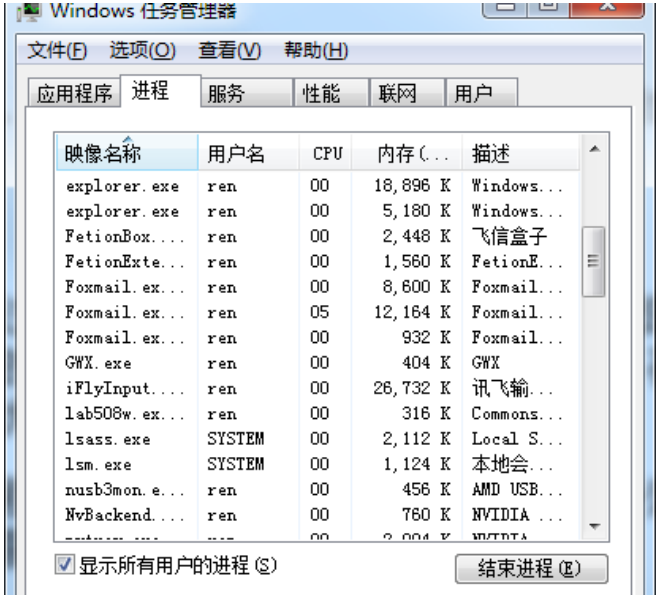
Java软件设计基础

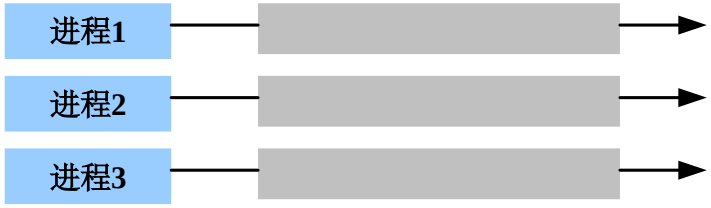
第十章 Java线程机制

1. Java中的多线程机制

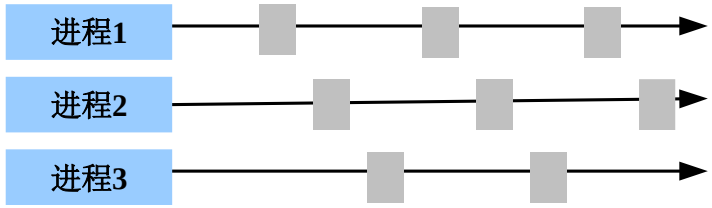
- 进程
  - 进程是程序的一次执行过程，是系统进行调度和资源分配的一个独立单位，每个进程都有自己独立的一块内存空间、一组系统资源。在进程概念中，每个进程的内部数据和状态都是完全独立的。
  - 计算机上同时运行多个程序叫多进程并发。操作系统按照一定的策略调度各个进程执行，以便最大限度的利用计算机的各种资源。



- 进程的并发(Concurrency)与并行(Parallel)



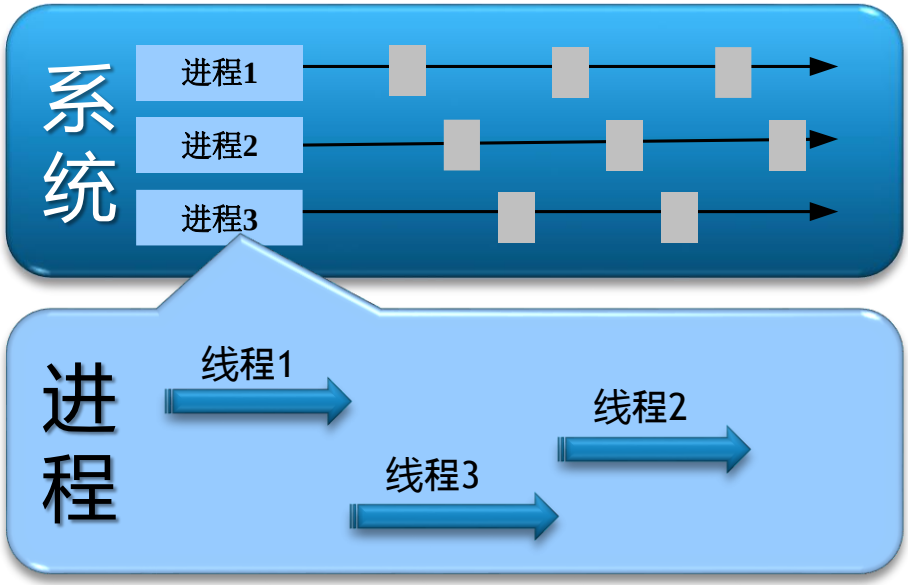
多个进程运行在多CPU上



多个进程共享单CPU

◆ 并行是指同一时刻有多条指令在多个处理器上同时执行，并发指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

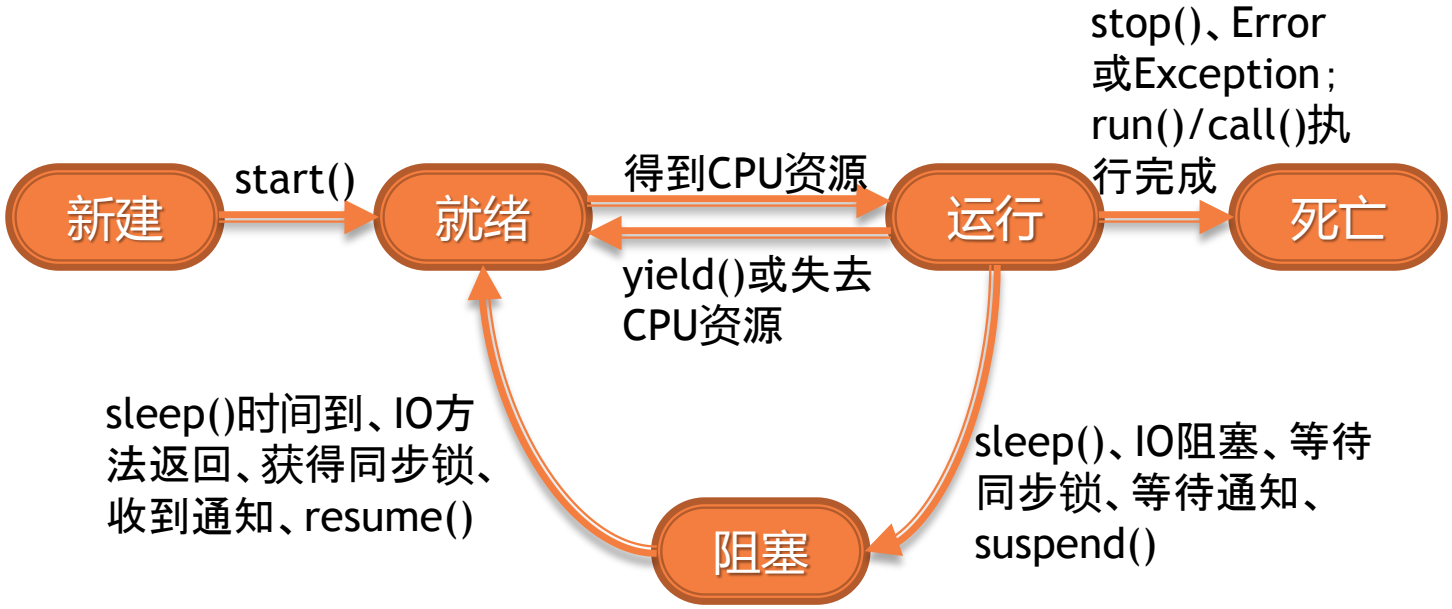
- 多线程
  - 多线程扩展了多进程的概念，使得同一个进程可以同时并发处理多个任务。
  - 线程(Thread)是进程中某个单个顺序的控制流，也被称为轻量进程(Lightweight Process)。



线程共享其父进程的系统资源；线程拥有运行必需的一些数据结构。

◆ 如果计算机只有一个CPU，那么在任何时刻只有一个线程处于运行状态；在多处理器的机器上会有多个线程并行执行；当线程数大于处理器数时，依然会有多个线程在一个处理器上轮换的现象。

- 线程的生命周期
  - 同进程一样，线程也有从创建、运行到消亡的过程，这称为线程的生命周期。



新建

- new()：用new关键字创建一个线程对象后，该线程对象就处于新建状态，这是线程已被创建但未开始执行的特殊状态。系统不为它分配资源，但有自己的内存空间。

就绪

- start：使线程处于就绪状态
  - JVM会为其创建方法调用栈和程序计数器，一旦获得CPU资源，线程就进入运行状态，并自动调用自己的run()方法。

- 不要显式的去调用**run()**方法，否则系统将把**run()**方法作为一个普通方法调用，而非线程的执行体。
- 只能对新建状态的线程调用**start()**方法，否则将引起**IllegalThreadStateException**异常。
- 不要试图对一个已经死亡的线程调用**start()**方法使它重新启动，死亡的线程不可再次最为线程执行。

运行 阻塞

- 线程开始运行以后不可能一直处于运行状态（除非它的执行体足够短，瞬间就执行结束了）。一般来说运行过程中的中断目的是使得其他线程获得执行的机会，其调度细节取决于底层平台采用的策略。
  - 抢占式调度策略：
    - 现代桌面和服务器操作系统大多采用，会给每个可执行线程一个小时间段来处理任务，当该时间段用完后系统会剥夺其占用的资源，让其他线程获得执行机会，在选择下一个线程时系统会考虑线程的优先级。
  - 协作式调度策略
    - 小型设备如手机等大多采用，只有当一个线程调用了sleep()或yield()方法时才会放弃所占用的资源——也就是必须由运行中的线程主动放弃。

处于阻塞、等待或计时等待状态的线程不运行任何代码，且消耗最少的资源。

2. 线程的创建与实现

当程序作为一个应用程序运行时，**Java**解释器为**main**方法启动一个线程。当程序作为一个**applet**运行时，**Web**浏览器启动一个线程来运行**applet**。

- 线程的三种创建方法

继承创建Thread类的子类来实现

```
Class MyThread extends Thread{
        public void run(){
        ……//该线程要实现什么任务
        }
}//自定义线程类
```

```
MyThread t1=new MyThread();
t1.start();
```

① 定义Thread类的子类，并重写run()方法，其方法体代表线程需要完成的任务（以后统一称为执行体）；

② 创建该Thread子类的实例；
③ 调用start方法启动该线程。

```java
import java.util.*;

public class MyThread extends Thread{
    public void run(){
        int r=new Random().nextInt(100);
        System.out.println(getName()+"produced the random number:"+r);
    }
    public static void main(String[] args) {
        for(int i=0;i<10;i++){
            new MyThread().start();
        }
    }
}
```

```
Thread-8produced the random number:9
Thread-6produced the random number:98
Thread-9produced the random number:25
Thread-3produced the random number:22
Thread-2produced the random number:67
Thread-7produced the random number:10
Thread-4produced the random number:88
Thread-5produced the random number:23
Thread-0produced the random number:81
Thread-1produced the random number:0
```

```
Thread-4produced the random number:40
Thread-0produced the random number:0
Thread-7produced the random number:70
Thread-3produced the random number:83
Thread-2produced the random number:92
Thread-1produced the random number:89
Thread-5produced the random number:6
Thread-6produced the random number:77
Thread-9produced the random number:59
Thread-8produced the random number:41
```

利用Runnable接口来实现

```
Class MyTask implements Runnable{
        public void run(){
        ……//实现Runnable接口中的方法
        }
}//自定义任务类
```

```
MyTask task=new MyTask();
Thread t1=new Thread(task);
t1.start();
```

① 定义实现了Runnable接口的实现类，并实现该接口的run()方法即执行体；

② 创建该实现类的实例；
③ 以该实例为作为Thread的参数创建线程对象；
④ 调用start方法启动该线程。

```java
class Task implements Runnable{
    public void run(){
        int r=new Random().nextInt(100);
        System.out.println(Thread.currentThread().getName()+"produced the random number:"+r);
    }
}
public class MyThread{
    public static void main(String[] args) {
        Task t1=new Task();
        for(int i=0;i<10;i++){
            new Thread(t1).start();
        }
    }
}
```

Thread提供的两个方法：
currentThread方法：静态方法，返回当前正在执行的线程对象；
getName方法：实例方法，返回调用该方法的线程名字

利用Callable接口来实现

```
Class MyTask implements Callable{
      public 返回值类型 call() throws 异常列表{
      ……//实现Call方法，表明该任务要做什么
}
}//自定义任务类
```

```
MyTask mt=new MyTask();
FutureTask ft=new FutureTask(mt);
Thread t1=new Thread(ft);
t1.start();
```

① 定义实现了Callable接口的实现类，并实现该接口的call()方法即执行体；

② 创建该实现类的实例；
③ 以该实例为作为FutureTask的参数创建对象，该FutureTask对象封装了call方法的返回值；
④ 以该FutureTask对象作为Thread的参数创建线程；
⑤ 调用start方法启动该线程。

call()方法可以有返回值，可以抛出异常；
FutureTask对象可通过get()方法来获得call方法的返回值

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

class MyTask implements Callable{
    public String call() throws Exception{
        int r=new Random().nextInt(100);
        return Thread.currentThread().getName()+" produced the random number:"+r;
    }
}

public class CallableAndFuture {
    public static void main(String[] args) {
        MyTask mt=new MyTask();
        FutureTask ft;
        try{
            for(int i=0;i<10;i++){
                ft = new FutureTask(mt);
                new Thread(ft).start();
                System.out.println(ft.get());
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }catch (ExecutionException e){
            e.printStackTrace();
        }
    }
}
```

- 三种创建方式的对比

使用接口的方式：可以从其他类继承；可以共享同一个任务对象（非常适合多个相同线程处理同一份资源的情况）；将CPU、代码和数据分开，形成清晰的风格；

使用继承Thread类的方式：编写简单；无法继承其他类；

也可以声明一个Thread类的子类，并实现Runnable接口。然后在客户端程序中用这个类生成一个对象，并调用它的start方法来启动线程。不推荐使用以上模式，因为它将任务和任务运行的机制混在一起。将任务从线程中分离出来是比较好的设计。

3. 控制线程

- 通过设置线程的优先级
  - Java将线程的优先级分为10个等级，用1~10表示，数字越大表示优先级越高。优先级高的线程获得较多的执行机会。
  - Thread类中定义了三个成员变量：

MIN_PRIORITY：	**最低级(1级)**
MAX_PRIORITY：	**最高级(10级)**
NORM_PRIORITY：	**普通优先级(5级)**

    - 当线程对象被创建时，默认优先级与其父线程的优先级相同，默认情况下main线程具有普通优先级，所以由main线程创建的子线程优先级为5。
    - 线程可以通过调用setPriority方法改变优先级。

```java
public void run(){
    for(int i=0;i<50;i++){
        System.out.println(getName()+",其优先级为："+getPriority()+",循环变量的值为："+i);
    }
}
```

```java
public static void main(String[] args) {
    Thread.currentThread().setPriority(6);
    for(int i=0;i<30;i++){
        if(i==10){
            PriorityTest low=new PriorityTest("低级");
            low.start();
            System.out.println(low.getPriority());
            low.setPriority(Thread.MIN_PRIORITY);
        }
        if(i==20){
            PriorityTest high=new PriorityTest("高级");
            high.start();
            System.out.println(high.getPriority());
            high.setPriority(Thread.MAX_PRIORITY);
        }
    }
}
```

优先级高的线程获得了更多的执行机会

```
高级,其优先级为：10,循环变量的值为：8
高级,其优先级为：10,循环变量的值为：9
高级,其优先级为：10,循环变量的值为：10
低级,其优先级为：1,循环变量的值为：2
高级,其优先级为：10,循环变量的值为：11
低级,其优先级为：1,循环变量的值为：3
高级,其优先级为：10,循环变量的值为：12
高级,其优先级为：10,循环变量的值为：13
低级,其优先级为：1,循环变量的值为：4
高级,其优先级为：10,循环变量的值为：14
低级,其优先级为：1,循环变量的值为：5
```

JVM以及操作系统会优先处理优先级别高的线程，但不代表这些线程一定会先完成。

设定优先级只能建议系统更快的处理（与底层操作系统有关），而不能强制。

在运行时，并非按照方法分界，而是按照机器码/汇编码分界。也就是说不保证任何一段代码是被完整而不打断而执行的。

- sleep()方法
  - 是Thread类的静态方法，使线程睡眠一段时间并进入阻塞状态。

```
sleep(long ms);                 //睡眠ms毫秒
sleep(long ms, int ns);         //睡眠ms毫秒+ns纳秒
```

  - 该方法受到系统计时器和线程调度器的精度与准确度的影响。
  - 正在运行的线程可以在它的run()方法体中调用sleep()方法来使自己放弃处理器资源而休眠一段时间，长短由sleep()方法的参数决定。
  - 如果线程在休眠时被打断，JVM就抛出InterruptedException异常，因此必须在try-catch语句块中调用sleep()方法，或抛出该异常。

```
class ThreadEx2 extends Thread{
    String s;
    int m,i=0;
    ThreadEx2(String s){
        this.s=s;
    }
    public void run(){
        try{
            sleep((int)(500*Math.random()));
            System.out.println(s+"finished");
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
    public static void main(String args[]){
        ThreadEx2 a=new ThreadEx2("a  ");
        ThreadEx2 b=new ThreadEx2("b  ");
        a.start();b.start();
        System.out.println("main is finished");
    }
}
```

Ex5

```
E:\javadoc\Chp12>java ThreadEx2
main is finished
b  finished
a  finished

E:\javadoc\Chp12>java ThreadEx2
main is finished
a  finished
b  finished
```

- yield()方法
  - 是Thread类的静态方法，可让当前正在执行的线程暂停并转入就绪状态。
  - 当某个线程调用了yield方法暂停后，只有优先级与当前线程相同，或者优先级比当前线程更高的处于就绪状态的线程才会获得执行机会。

> 完全有可能的是，某个线程调用了**yield**方法以后立刻又获得了处理器资源开始运行。

yield与sleep	sleep方法会将线程转入阻塞状态，直到时间结束才转入就绪状态；yield是直接进入就绪状态
	调用sleep方法要求捕获或抛出异常；
	sleep方法有更好的可移植性，通常不建议使用yield方法来控制并发线程的执行。

```
public class  YieldTest extends Thread{
    public YieldTest(String name){
        super(name);
    }
    public void run(){
        for(int i=0;i<50;i++){
            System.out.println(getName()+","+i);
            if(i==20){
                  Thread.yield();
            }
        }
    }
    public static void main(String[] args) throws Exception{
        YieldTest yt1=new YieldTest(“1");
        YieldTest yt2=new YieldTest(“2");
        yt1.start();
        yt2.start();
    }
}
```

Ex 6

- join()方法
  - 表示当前线程等待调用该方法的线程结束后再恢复执行。
  - 一个线程A在占有CPU资源期间，可以让其他线程B调用join()方法与本线程联合，即“B.join();”，这样称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线程将立即中断运行，一直等到它联合的线程B执行完毕，A线程才再重新排队等待CPU资源。
  - 如果A准备联合的线程B已经结束，则“B.join();”语句将不起任何作用。

通常由使用线程的程序调用，以将大问题分成许多小问题，每个小问题分配一个线程，当所有的小问题都得到解决后，再调用主线程来进一步操作。

```
public class ThreadJoin{
    public static void main(String argsp[]){
        TJoin a = new TJoin();
        a.threadA.start(); a.threadB.start();
    }
}
class TJoin implements Runnable{
    Thread threadA,threadB;
    String content[]={"各就位","预备","跑"};
    public TJoin(){
        threadA=new Thread(this); threadB=new Thread(this);
        threadA.setName("运动员");threadB.setName("发令员");
    }
    public void run(){
        if(Thread.currentThread()==threadA){
            System.out.println("等待"+threadB.getName()+"发令");
            try{threadB.join();}
            catch(InterruptedException e){return;}
            System.out.println("开始跑!");
        }
        else if(Thread.currentThread()==threadB){
            System.out.println(threadB.getName()+"发令:");
            for(int i=0;i<content.length;i++){
                System.out.println(content[i]);
                try{threadB.sleep(1000);}
                catch(InterruptedException e){return;}
            }
        }
    }
}
```

Ex7

运行结果

```
等待发令员发令
发令员发令:
各就位
预备
跑
开始跑!
```

```
import java.util.concurrent.*;
public class SumTask implements Callable{
	int i,j;
	public SumTask(int i,int j){
		this.i=i;
		this.j=j;
	}
	public Integer call() throws Exception{
		int sum=0;
		for(int count=i;count<=j;count++){
			sum+=count;
		}
		return sum;
	}
	public static void main(String[] args) throws Exception{
		int sum=0;
		for(int i=0;i<10;i++){
			SumTask st=new SumTask(i*10+1,(i+1)*10);
			FutureTask ft=new FutureTask(st);
			Thread t=new Thread(ft);
			t.start();
			t.join();
			sum+=((Integer)ft.get()).intValue();
		}
		System.out.println(sum);
	}
}
```

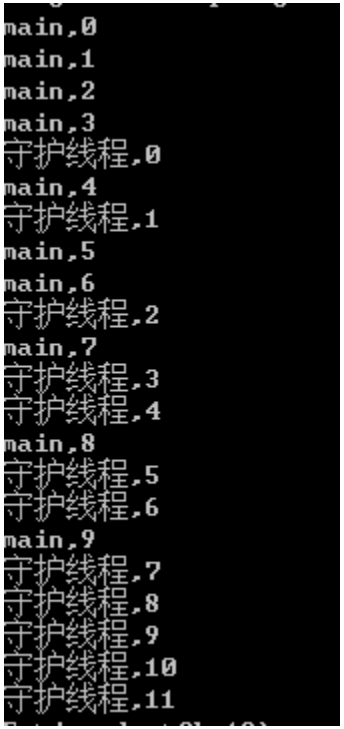
Ex8

将1~100的求和分为1~10、10~20……分为10个子线程计算，当10个子线程结束后汇总计算总和。

- setDaemon()方法
  - 比较特殊的一类线程是被称作后台(Daemon)线程的低级别线程（也叫守护线程、精灵线程），JVM中的垃圾回收线程就是典型的后台线程。
  - 一般来说前台线程创建的子线程默认为前台线程；后台线程创建的子线程默认为后台线程；
  - setDaemon方法可将一个线程设置为后台线程，isDaemon方法用于判断某线程是否为后台线程；
  - 当所有的前台线程死亡时，后台线程随之死亡。设置某线程为后台线程必须在其调用start方法之前设置，否则会产生IllegalThreadStateException异常。

```
public class DaemonThread extends Thread{
	public DaemonThread(String s){
		super(s);
	}
	public void run(){
		for(int i=0;i<100;i++){
			System.out.println(getName()+","+i);
		}
	}
	public static void main(String[] args) {
		DaemonThread dt=new DaemonThread("守护线程");
		dt.setDaemon(true);
		dt.start();
		for(int i=0;i<10;i++){
			System.out.println(Thread.currentThread().getName()+","+i);
		}
	}
}
```

Ex9



从上面的结果可以看出主线程（前台线程）结束后，即使后台线程执行体中应该运行至99，事实上却无法达成，因为前台线程运行结束后JVM会主动退出，因而后台线程也就被结束了。

4. 线程的同步

- 线程同步概念的提出
  - Java应用程序的多个线程共享同一进程的数据资源，多个用户线程在并发运行过程中可能会同时访问具有敏感性的内容。一个线程需要修改该内容时，另一个线程也要修改该内容，系统需要对以上情况作出使当的处理。
  - Java中定义了线程同步的概念，用来实现共享数据的一致性维护。
  - 线程同步的使用主要是保证一个进程中多个线程的协调工作，使得线程安全的共享数据，转换和控制线程的执行，保证内存的一致性。
- 同步方法
  - 例：对应同一银行帐户的取款问题。

```
public class BankEx{
	public static void main(String args[]) throws InterruptedException{
		BankSave bs=new BankSave();
		Operator o1=new Operator(bs,“爸爸");
		Operator o2=new Operator(bs,“孩子");
		Thread t1=new Thread(o1);
		Thread t2=new Thread(o2);
		t1.start(); t2.start();
		Thread.currentThread().sleep(500);
	}
}
class BankSave{
	private static int money=1000;
	public void add(int i){
		money=money+i;
		System.out.println(“爸爸存入"+i+"元，余额"+money+"元");
	}
	public void get(int i){
		if(i<=money){
		money=money-i;
		System.out.println(“孩子取走"+i+"元，余额"+money+"元");
	}
	else System.out.println("余额不足");
	}
	public int showMoney(){return money;}
}
```

```
class Operator implements Runnable{
    String name;
    BankSave bs;
    public Operator(BankSave bs,String name){
        this.bs=bs;
        this.name=name;
    }
    public static void oper(String name,BankSave bs){
        if(name.equals("爸爸")){
            try{
                for(int i=0;i<5;i++){
                    Thread.currentThread().sleep((int)(Math.random()*300));
                    bs.add(1000);
                }
            }catch(InterruptedException e){return; }
        }
        else try{
            for(int i=0;i<5;i++){
                Thread.currentThread().sleep((int)(Math.random()*300));
                bs.get(1000);
            }
        }catch(InterruptedException e){return; }
    public void run(){oper(name,bs); }
}
```

```
爸爸存入1000元，余额9000元
孩子取出1000元，余额9000元
```

public static synchronized void oper….
在程序的运行中，如果某一线程调用经synchronized修饰的方法，在该线程结束此方法之前，其他所有线程都不能运行该方法，只有等待到该方法被释放为止。

- 同步代码块
  - 上述结果因为程序中可能有两个线程同时修改银行帐户的对象，因此也可以使用银行帐户作为同步监视器，使得任何修改该银行帐户的操作无法有两个或以上的线程同时进行。

```
synchronized (bs){
    针对银行帐户进行修改的操作……
}
```

  - 线程可以执行代码块中的操作之前，必须要先获得同步监视器的锁定，执行完成后，会释放对该同步监视器的锁定，以便于其他线程执行。

synchronized关键字可以修饰方法、代码块，但不能修饰构造方法、成员变量等；

如果作为类修饰符，表明该类中的方法都是**synchronized**的，但尽量不要对所有方法都进行同步，只对那些会改变竞争资源的方法进行同步。

- 同步锁Lock
  - Java SE 5.0引入，通过显式定义同步锁对象来实现同步。在实现线程安全的控制中，比较常用的是ReentrantLock。

```
public class c{
        Lock myLock=new ReentrantLock();
        public void f(){
                myLock.lock();
                try{
                        修改数据的操作
                }
                finally{
                        myLock.unlock();
                }
        }
}
```

- 死锁
  - 当两个或多个线程等待一个不可能满足的条件时会发生死锁。
  - 死锁是发生在线程间相互阻塞的现象，允许多个线程并发访问共享资源时，必须提供同步机制，然而如果对这种机制使用不当，可能会出现线程永远被阻塞的现象。
    - 例如两个线程分别等待对方各自占有的一个资源，就会产生死锁。

5. 线程通信

- 当线程在系统内运行时，线程的调度具有一定的透明性，程序通常无法准确控制线程执行顺序，Java提供了一些通信机制以协调运行。
  - wait()：等待方法
    - 可让当前线程释放锁，并进入等待状态，直到其他线程进入同一监视器并调用notify()方法为止；
    - wait方法可以无参或带时间参数，即等待多久；
  - notify()
    - 可唤醒同一同步监视器中调用了wait()方法的单个线程，只有当前线程放弃对此同步监视器的锁定，被唤醒的线程才可以执行。
  - notifyAll()
    - 唤醒此同步监视器中调用了wait()方法的所有线程，只有当前线程放弃对此同步监视器的锁定，被唤醒的线程才可以执行。

- 上述三个方法属于**Object**类。
- 对于使用同步修饰符的方法，因为该类的默认实例（**this**）就是同步监视器，所以可以在同步方法中直接调用上述三个方法
- 对于使用同步修饰符修饰的代码块，同步监视器是修饰符后括号里的对象，所以需要由该对象调用这三个方法。

- 实例：模拟三人买票
  - 说明：
    - 售货员最初只有一张五元钱，电影票五元一张。
    - 排队的三人中，A有一张二十元人民币，B有一张十元人民币，C有一张五元人民币。
    - 如果给售票员的钱不是零钱，而售票员又没有零钱找，那么此人必须等待，并允许后面的人买票，以便售票员获得零钱。
  - 分析
    - A/B/C三人 买票就像三个线程，每个线程必须满足以上条件（售票员有足够的找零或者提供售票员的是五元零钱）才能执行；
    - A/B/C三人同时只能有一人向售票员买票；不满足条件时需要等待。

售票窗口类

```
class Cinema implements Runnable{
        Thread A,B,C;
        TicketSeller seller;
        Cinema(){
                A=new Thread(this);B=new Thread(this);C=new Thread(this);
                A.setName("A");B.setName("B");C.setName("C");
                seller=new TicketSeller();
        }
        public void run(){
                if(Thread.currentThread()==A){seller.sellTicket(20);}
                else if(Thread.currentThread()==B){seller.sellTicket(10);}
                else if(Thread.currentThread()==C){seller.sellTicket(5);}
        }
}
```

```
class TicketSeller{
        int five=1,ten=0,twenty=0;
        public synchronized  void sellTicket(int receiveMoney){
                if(receiveMoney==5){
                        five++;
                        System.out.println(Thread.currentThread().getName()+"给我五元钱，售票一张，无找零");
                }
                else if(receiveMoney==10){
                        while(five<1){
                                try{
                                        System.out.println(Thread.currentThread().getName()+" 请您一旁等待");
                                        wait();
                                        System.out.println(Thread.currentThread().getName()+"结束等待");
                                }catch(InterruptedException e){
                                        return;
                                }
                        }
                        five--;
                        ten++;
                        System.out.println(Thread.currentThread().getName()+"给我十元钱，售票一张，找零五元");
                }
```

```
                else if(receiveMoney==20){
                        while(five<1||ten<1){
                                try{
                                        System.out.println(Thread.currentThread().getName()+" 请您一旁等待");
                                        wait();
                                        System.out.println(Thread.currentThread().getName()+"结束等待");
                                }catch(InterruptedException e){
                                        return;
                                }
                        }
                        five--;
                        ten--;
                        twenty++;

        System.out.println(Thread.currentThread().getName()+"给我二十元钱，售票一张，找零十五元");
                }
                notifyAll();
        }
}
public class TicketEx{
        public static void main(String args[]){
                Cinema a = new Cinema();
                a.A.start();
                a.B.start();
                a.C.start();
        }
}
```

运行结果

```
A 请您一旁等待
B给我十元钱，售票一张，找零五元
A结束等待
A 请您一旁等待
C给我五元钱，售票一张，无找零
A结束等待
A给我二十元钱，售票一张，找零十五元
```

- 使用Condition提供的方法控制线程通信
  - Condition实例被绑定在一个Lock对象上，它提供await()、signal()、signalAll()方法用于Lock对象上线程的等待和唤醒。
  - 可以让得到Lock对象却无法继续执行的线程释放Lock对象。
  - 上例关键部分可以改为：

```
class TicketSeller{
    private final Lock lock=new ReentrantLock();
    private final Condition cond=lock.newCondition();
    int five=1,ten=0,twenty=0;
    public  void sellTicket(int receiveMoney){
        lock.lock();
        try{
            if(receiveMoney==5){……//略}
            else if(receiveMoney==10) ){
                cond.await();
            ……//略}
            else if(receiveMoney==20) {
                cond.await();
            ……//略}
            cond.signalAll();
        }finally{
            lock.unlock();
        }
    }
}
```

- BlockingQueue(阻塞队列)控制线程通信
  - 当生产者试图向BlockingQueue中放入元素时，如果队列已满，则该线程阻塞；
  - 当消费者试图从BlockingQueue取出元素时，如果队列为空，则该线程阻塞。

```
class Producer extends Thread{
	private BlockingQueue<String> bq;
	public Producer(BlockingQueue<String> bq){
		this.bq=bq;
	}
	public void run(){
		String[] sArr=new String[]{"产品1","产品2","产品3"};
		for(int i=0;i<3;i++){
			try{
				Thread.sleep(200);
				bq.put(sArr[i%3]);
			}catch(Exception e){
				e.printStackTrace();
			}
			System.out.println(getName()+"生产完成,放入："+bq);
		}
	}
}
```

```
class Consumer extends Thread{
	private BlockingQueue<String> bq;
	public Consumer(BlockingQueue<String> bq){
		this.bq=bq;
	}
	public void run(){
		while(true){
			try{
				Thread.sleep(200);
				bq.take();
			}catch(Exception e){
				e.printStackTrace();
			}
			System.out.println(getName()+"有货，取货完成"+bq);
		}
	}
}
```

```
Thread-0生产完成,放入：[产品3]
Thread-3有货，取货完成[]
Thread-2生产完成,放入：[产品2]
Thread-3有货，取货完成[]
Thread-1生产完成,放入：[产品3]
Thread-3有货，取货完成[]
Thread-2生产完成,放入：[产品3]
```

主方法中创建了3个生产者和1个消费者线程，并将bq长度设为1。
可以看出，由于容量为1，生产者线程和消费者线程自动交替进行

- 实例：进度条
  - 说明：当一个线程从源文件向目标文件复制数据时，进度条在另一个线程中同时更新，需要创建一个复制文件的线程，只要复制文件中的一些字节，进度条的当前值就被更新，从而显示复制过程的进度。

run方法体

```
BufferedInputStream in=null;
BufferedOutputStream out=null;
try{
        File inf=new File(jfc1.getText());
        File outf=new File(jfc2.getText());
        in =new BufferedInputStream(new FileInputStream(inf));
        out =new BufferedOutputStream(new FileOutp
        long totalBytes=in.available();
        jpb.setValue(0);
        jpb.setMaximum(100);
        int r;
        long bytesRead=0;
        byte[] b=new byte[10];
        while((r=in.read(b,0,b.length))!=-1){
                out.write(b,0,r);
                bytesRead+=r;
                currentValue=(int)(bytesRead*100/totalBytes);
                jpb.setValue(currentValue);
```