



Multithreading

--- Java Advanced Tutorial , Module 2

ALGOM HIGH-END IT TRAINING , JAVA ADVANCED

内容提要

多线程机制

线程的创建和启动

线程状态

线程调度(优先级)

线程同步

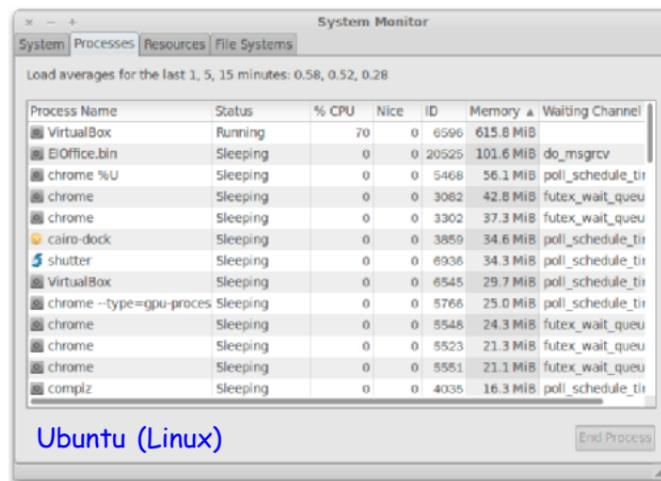
线程通信

定时任务

进程

什么是进程 (Process)

- 主流计算机操作系统都支持同时运行多个任务
- 每个任务通常就是一个程序
- 每个运行中的程序就是一个进程或者多个进程
- 我们能直观的“看到”进程



进程

■ 进程有什么特点

■ 独立性

- ◆ 进程是系统中独立存在的实体
- ◆ 可以拥有自己独立的资源
- ◆ 拥有自己私有的地址空间
 - 在未经允许的情况下，用户进程不能其他进程的地址空间

■ 动态性

- ◆ 进程是一个正在系统中活动的指令集合
 - 与进程不同的是，程序是一个静态的指令集合
- ◆ 进程具有自己的生命周期和各种不同的状态
 - 与进程不同的是，程序没有时间的概念，也就没有生命周期等概念

■ 并发性

- ◆ 多个进程可以在单个处理器上并发运行
- ◆ 并发运行的各个进程之间不会相互影响

并行与并发

■ 并行 : parallel

- 并行, 指的是多个进程在多个处理器上同时执行

- ◆ 正如同多辆汽车行使在多条马路上或者多个车道上
 - 多条马路, 可以与我们的多个 CPU 相对应
 - 而多个车道, 则可以与我们的单个 CPU 多个内核相对应

■ 并发 : concurrency

- 并发, 指同一时刻只能运行一条指令, 但是多个进程指令快速地轮换执行, 在某个时间段内, 看上去具有多个进程同时执行的效果

- 也就是说, 并发, 并不是真的是由单个CPU同时运行多个程序

- ◆ 正如同高速路上的收费站:
 - 假设收费站只有一个窗口, 一次只能收一辆车的过路费
 - 如果只看收费窗口, 的确是"单个线程"在运行
 - 而如果看某一段路, 有多辆车在跑, 就像收费窗口一次收了多辆车的过路费
 - 但事实上确是同一时刻, 只能收一辆车的过路费, 也只能让一辆车通过

- 现代的主流操作系统都支持多进程的并发

并发策略的实现方式

不同的并发策略

共享式的多任务操作策略

- ◆ 所谓共享式多任务，也称作协作式多任务
 - 如果一个任务获得了CPU时间片，除非它愿意放弃，否则它将永远霸占CPU
 - 所以，为了保证系统正常运作，各个任务之间需要协作
 - 某个任务获得CPU时间片后，使用一段时间后释放CPU，让其他任务有机会运行
- ◆ 这类操作系统，一般是早期的操作系统或者小型的操作系统，如
 - Windows 3.1、Mac OS 9、Symbian OS 等

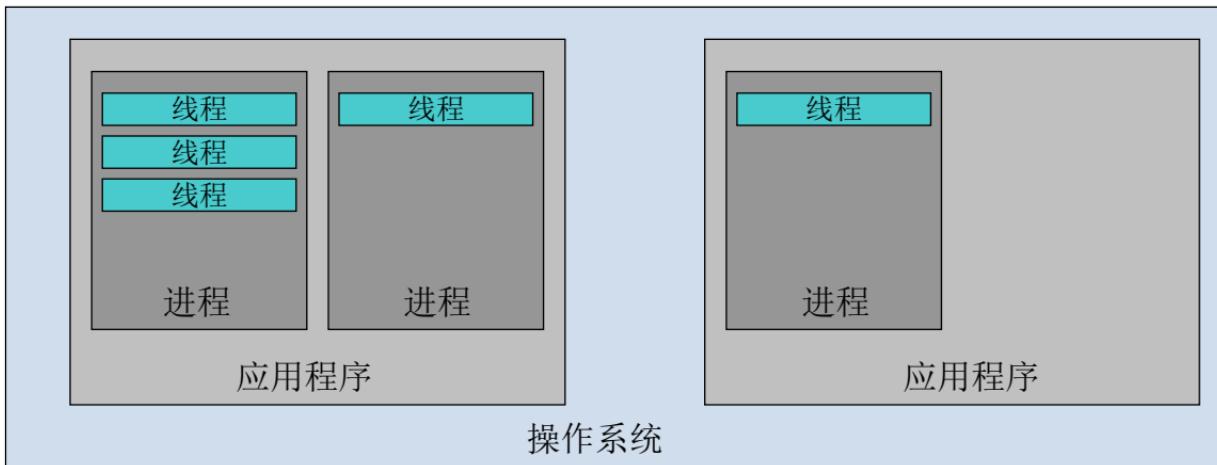
抢占式的多任务操作策略

- ◆ 所谓抢占式多任务是指
 - 总控制权在操作系统手中，直接中断而不事先和被中断程序协商
 - 操作系统会轮流询问每一个任务是否需要使用 CPU，需要使用的话就让它用
 - 不过在一定时间后，操作系统会剥夺当前任务的 CPU 使用权
 - 把它排在询问队列的最后，再去询问下一个任务
- ◆ 这类操作系统主要是大中型操作系统，如
 - Windows 95 之后的各个版本、Unix / Linux

线程

■ 什么是线程

- 线程 (Thread) 被称作 轻量级进程 (Lightweight Process)
- 线程是比进程更小一级的执行单元
 - ◆ 如同进程在操作系统中的地位一样，线程在进程中是独立的、并发的执行流



线程

■ 线程的特点

- 一个进程可以有多个线程，但至少有一个线程
 - ◆ 当进程被初始化后，主线程也就被创建了
 - 一般而言，一个应用程序仅要求有一个主线程(但可以有多个其它线程)
- 线程不能独立存在，必须属于某个进程
 - ◆ 线程可以拥有自己的堆栈、自己的程序计数器、自己的局部变量
 - ◆ 线程不能再独立拥有系统资源
 - 它将与父进程的其它线程共享该进程内所有的系统资源
- 线程可以独立完成某项任务
 - ◆ 也可以跟同一进程的其它线程一起完成某项任务
- 线程是独立运行的，它并不知道同进程内的其它线程的存在
- 线程的执行是抢占式的
- 一个线程可以创建和撤销另一个线程
 - ◆ 同一个进程中的多个线程之间可以并发执行

线程 vs 进程

线程和进程的区别与联系

进程：

- ◆ 每个进程都有独立的代码和数据空间(内存独立)
- ◆ 进程间的切换会有交大的开销

线程

- ◆ 可以看成是轻量级的进程
- ◆ 同一个进程中的线程共享代码和数据空间(栈内存独立，堆内存共享)
- ◆ 每个线程有独立的运行栈(方法调用栈)和程序计数器
- ◆ 线程切换的开销小

二者的联系

- ◆ 一个程序启动后，至少有一个进程
- ◆ 一个进程里可以包含多个线程，但是至少要有一个线程
- ◆ 线程不能脱离进程而独自存在

多线程

■ 多线程的优势

- 同一个进程的各线程间共享内存非常容易
 - ◆ 进程不能共享内存，因此进程间通信没那么容易
- 用多线程实现多任务并发比多进程效率高
 - ◆ 系统创建进程需要重新分配系统资源，因此效率不高
 - ◆ 而创建线程不需要重新分配系统资源，因此效率较高
- Java 语言简化了多线程编程
 - ◆ Java 语言内置多线程功能支持，而不是单纯地作为底层操作系统的调度方式
- 实际应用中，多线程应用非常广泛
 - ◆ 浏览器可以同时下载多个图片
 - ◆ 迅雷可以同时以 n 个线程下载一个文件
 - ◆ 一个 Web 服务器必须能同时响应多个用户的请求
 - ◆ JVM 本身则在后台提供了一个超级线程来回收垃圾

线程、进程、JVM

线程、进程与JVM的关系

- 通常我们所说的 JVM 有三层含义：
 - ◆ 抽象的 Java 虚拟机规范
 - ◆ 一个具体的 Java 虚拟机的实现
 - ◆ 一个运行中的 Java 虚拟机实例
- 一个运行中的 Java 虚拟机实例就是一个进程
 - ◆ 通过 Java 命令启动一个Java程序，就会启动一个新的进程(即JVM实例)
 - ◆ 下图为启动两个 Java 程序后，在任务管理器中的进程：

javaw.exe	Admin...	00	19,048 K	Java!
javaw.exe	Admin...	00	6,456 K	Java!

Win 7

java	Sleeping	0	0	27918	5.7 MiB	futex_wait_queu
java	Sleeping	0	0	27896	5.7 MiB	futex_wait_queu

Linux

线程、进程、JVM

线程、进程与JVM的关系

- JVM 中的线程分成两种
 - ◆ 守护线程
 - 通常是指由Java虚拟机自己使用的线程，比如 垃圾回收程序
 - Java 程序也可以将它创建的任意线程标记为守护线程
 - ◆ 非守护线程
 - 守护线程之外的其它线程，即为非守护线程，如 main 线程
 - 任何进程都有一个主线程，启动一个 Java 程序，main 线程即为主线程
- JVM 的生命周期
 - ◆ 当一个 Java 程序启动，其对应的 JVM 实例也就诞生了
 - ◆ 如果 JVM 中有非守护线程在运行，那么守护线程一定也要运行
 - ◆ 等到所有非守护线程都退出了，守护线程才退出
 - 如果安全管理器允许，非守护线程可以调用 System 或 Runtime 的 exit() 来退出
 - ◆ 当一个 JVM 实例中不再有任何线程运行，这个 JVM 实例也就走向消亡

创建并启动一个线程

■ 主线程

- 只要运行一个带有 main 方法的类，就会启动一个线程

■ 在 main 中创建一个线程

- 在进入 main 之后，再创建一个线程

```
Thread t = new Thread();
t.start();
System.out.println( t.getName() );
```

这里仅仅说明可以通过此种方式
创建一个线程(实际上毫无意义)

■ 如何获得当前线程

- 使用 Thread.currentThread() 可以获取当前线程的引用

```
Thread currentThread = Thread.currentThread(); //在那个里面调用就获取那个
System.out.println( "current thread : " + currentThread.getName() );
```

扩展**Thread**类实现自定义线程

实现步骤

- 重写 run 方法，其中包含该线程运行时所要执行的代码
- 通过线程对象的 start 方法启动一个线程

```
public class FirstThread extends Thread{  
    public void run() {  
        for(int i = 0 ; i < 100 ; i++){  
            System.out.println(i);  
        }  
    }  
    public static void main(String[] args) {  
        FirstThread ft = new FirstThread();  
        ft.start();  
    }  
}
```

1、JVM首先创建并启动主线程，
执行main()方法
2、main()方法创建了一个
FirstThread 对象 ft ，然后
调用该对象的 start() 方法启
动 FirstThread 线程

扩展**Thread**类实现自定义线程

■ 多个线程共享堆区的数据

■ 自定义一个线程类

```
public class MyThread extends Thread{  
    private int data = 0 ; //属于某个 MyThread 对象私有的数据  
    public void run() {  
        for( int i = 0 ; i < 10 ; i++ ){  
            System.out.println( Thread.currentThread().getName() + " : " + i );  
            data ++;  
            try {  
                Thread.sleep( 100 ); //让线程睡眠 100 毫秒  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        public void printData(){  
            System.out.println( this.data );  
        }  
    }  
}
```

扩展**Thread**类实现自定义线程

■ 多个线程共享堆区的数据

■ 共享同一个实例的实例变量

- ◆ 使用 主线程与自定义线程并发运行的方式即可验证

```
MyThread mt = new MyThread();
mt.start(); //启动一个线程
mt.run(); // main 线程中调用 mt 的 run() 方法
mt.printData(); // main 线程调用 mt 的 printData() 方法，输出 data 的值
```

■ 共享同一个实例对象

- ◆ 使用多个自定义线程并发的方式验证(它们共享一个实例对象)

```
MyThread mt = new MyThread();
Thread t1 = new Thread( mt );
t1.start();
Thread t2 = new Thread( mt );
t2.start();
mt.printData(); // main 线程调用 mt 的 printData() 方法，输出 data 的值
```

扩展**Thread**类实现自定义线程

■ 使用 Thread 实现自定义线程时需注意

■ 不要轻易覆盖 start 方法

- ◆ 如果任意覆盖 start 方法，那将导致调用 start 后无法启动线程
 - 覆盖了 start 方法，但没有调用 Thread 的 start()，那么该类的 start 方法将成为一个普通的方法，而不是一个用于启动线程的方法

■ 如果非要覆盖 start 方法

- ◆ 则在该方法内部第一行写 super.start()，以保证能够启动一个线程
 - 原因在于 Thread 中的 start 方法调用了一个本地方法 start0()
 - ◆ 然后再写其它的自定义代码
 - 这里，自定义代码有可能先于当前线程中的 run() 执行，也有可能在其后执行

通过实现 **Runnable** 接口实现自定义线程

实现步骤

- 实现 `java.lang.Runnable` 接口，实现 `run()` 方法

```
public class TestRunnable implements Runnable {  
    private int data = 0;  
    public void run() {  
        for( int i = 0 ; i < 10 ; i++ ){  
            System.out.println( Thread.currentThread().getName() + " : " + i );  
            data++;  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}  
} // 这里限于篇幅，省略了 printData 方法，你可以自己添加
```

通过实现 **Runnable** 接口实现自定义线程

实现步骤

- 实现 Runnable 接口的类不能作为线程直接启动
 - 它需要借助 Thread 构造创建一个线程对象，然后再启动之

```
public class Test {  
  
    public static void main(String[] args) {  
  
        TestRunnable tr = new TestRunnable(); // 不能使用 tr.start() 启动线程  
  
        Thread t1 = new Thread(tr);  
  
        t1.start(); // 启动线程  
    }  
}
```

通过实现 **Runnable** 接口实现自定义线程

共享数据

- 创建线程时使用 TestRunnable 的同一个实例

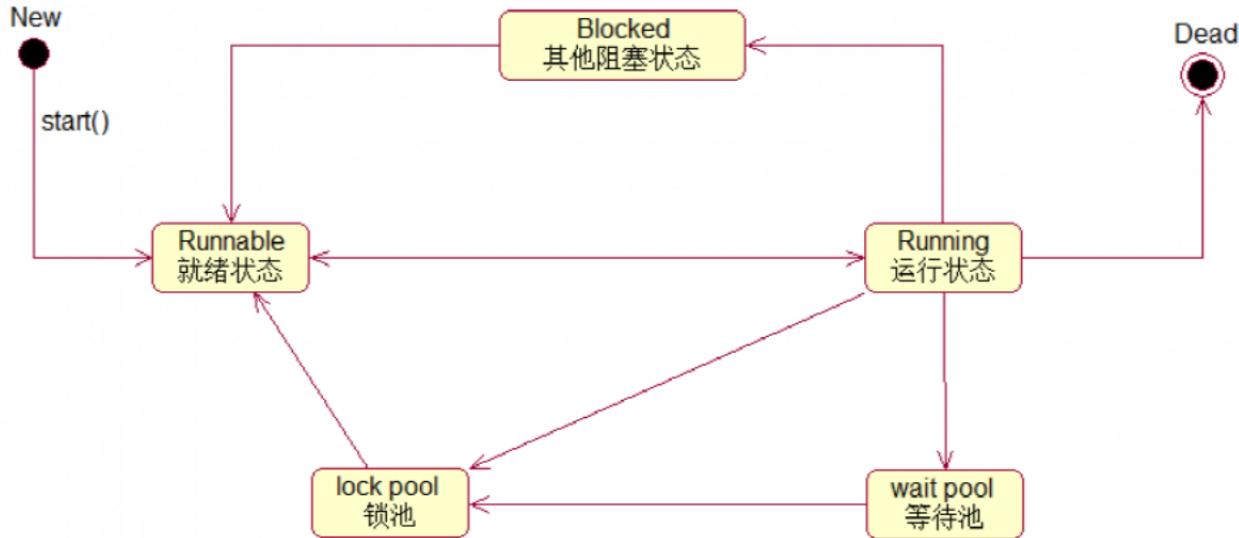
```
TestRunnable tr = new TestRunnable();
Thread t1 = new Thread( tr );           t1.start();
Thread t2 = new Thread( tr );           t2.start();
tr.printData(); // main 线程调用 mt 的 printData() 方法，输出 data 的值
```

- 使用不同实例，数据不共享

```
TestRunnable tr1 = new TestRunnable();
Thread t1 = new Thread( tr1 );           t1.start();
tr1.printData();
```

```
TestRunnable tr2 = new TestRunnable();
Thread t2 = new Thread( tr2 );           t2.start();
tr2.printData();
```

线程的状态图



线程的状态

■ 新建状态 (new)

- 使用 new 关键字创建的线程对象，处于新建状态
 - ◆ 注意这里说的使用 new 关键字是本质上使用new关键字
 - ◆ 很可能调用某个方法获得一个线程，只要这个方法中使用了new 关键字，那么这个线程就是新建状态
- 比如 Thread t = new Thread(); 语句就新建了一个线程对象

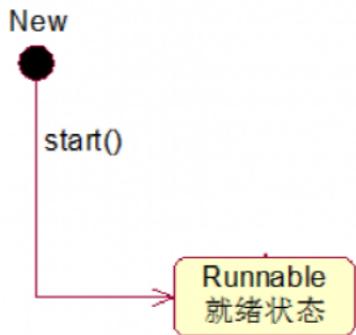
New



线程的状态

■ 就绪状态

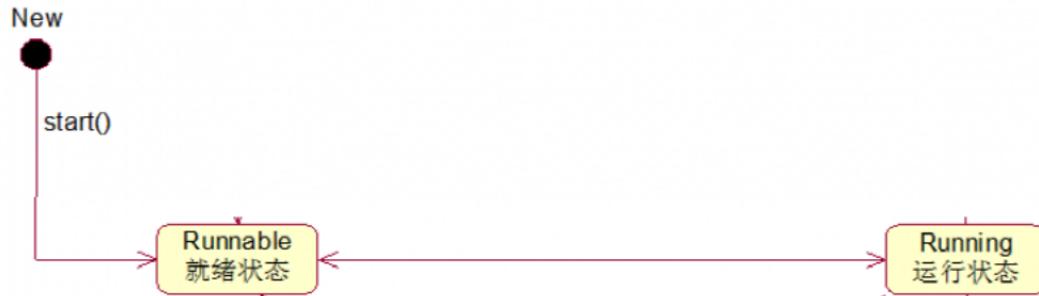
- 一个线程被创建后，并不是立即运行
- 其他线程调用该线程的 start() 方法，会使该线程处于就绪状态
 - 这时，JVM就会为它创建 方法调用栈 和 PC寄存器
 - 该状态的线程处于可运行池中，等待获得 CPU 的使用权



线程的状态

运行状态

- 该状态的线程占用 CPU，执行其程序代码
- 只有处于就绪状态的线程才能有机会转到该状态
- 单一 CPU，任一时刻只会有一个线程处于该状态
- 多CPU环境下，会有多个线程同时处于该状态



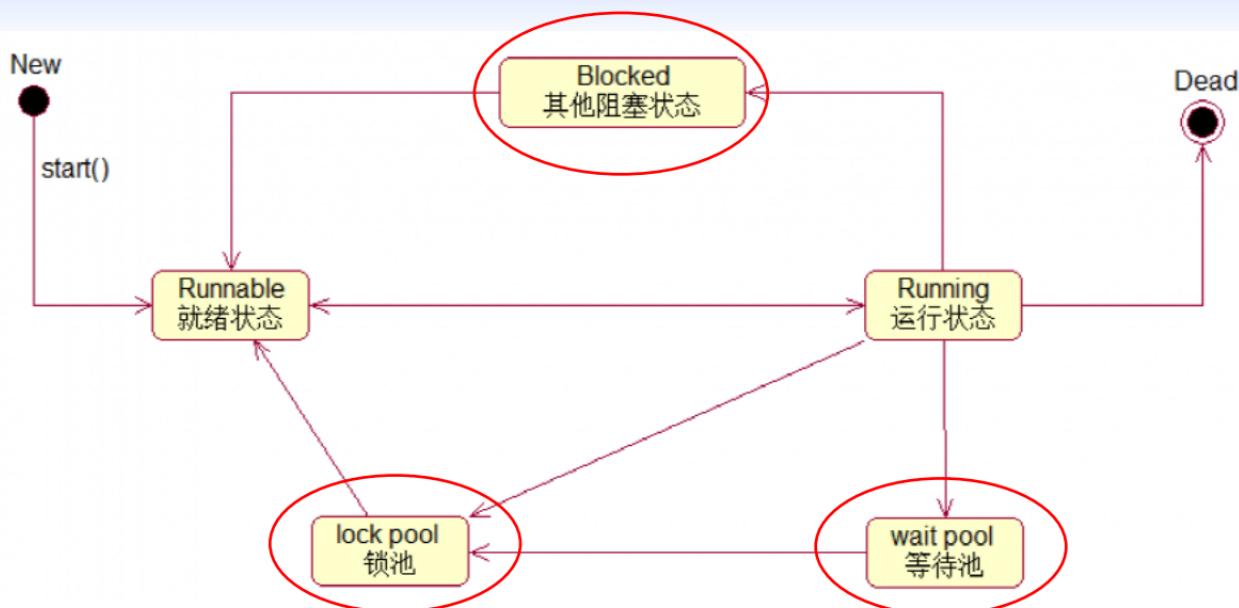
线程的状态

阻塞状态

- 指线程因为放弃CPU，暂停运行
- 线程处于阻塞状态，JVM不会给线程分配CPU
- 阻塞状态分3中情况：
 - ◆ 位于对象等待池中的阻塞状态
 - 处于运行状态的线程，某个对象调用了wait()方法
 - 参见线程通信
 - ◆ 处于对象锁池中的阻塞状态
 - 处于运行状态的线程，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，JVM会把这个线程放到这个对象的锁池中。（参见线程同步）
 - ◆ 其他阻塞状态
 - 线程执行了sleep()方法、或者调用了其他线程的join()方法、或者发出了I/O请求，就会进入该状态

线程的状态

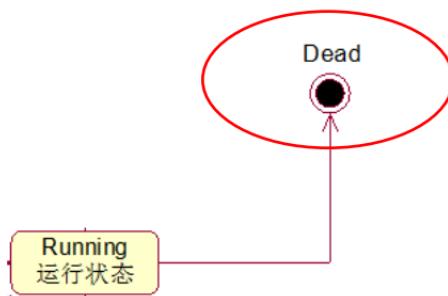
阻塞状态



线程的状态

死亡状态

- 当线程退出 run() 方法，线程就处于死亡状态
- 该线程的生命周期也就结束了
 - 线程退出 run() 方法，有可能是程序执行完毕，也有可能是程序异常中断
 - 无论是正常退出还是异常中断，都不会对其他线程造成影响



线程调度的相关概念

■ 线程调度

- 单个CPU，任意时刻只能执行一条机器指令
- 每个线程只有获得CPU的使用权才能执行指令
- **线程调度**就是按照特定的机制给多个线程分配CPU使用权

■ 调度模式

- 分时调度(协作)模式
 - ◆ 让所有线程都轮流获得CPU的使用权
 - ◆ 并且平均分配每个线程所占用的CPU时间片
- 抢占式调度模式
 - ◆ 优先让线程池中优先级高的线程占用CPU
 - ◆ 当优先级比较高的线程执行完毕后优先级较低的线程才有机会执行
 - ◆ 优先级相同的多个线程，谁先抢到时间片谁就运行

线程调度的相关概念

JVM 中的线程调度

- JVM采用抢占式调度模式
 - ◆ 即优先让线程池中优先级高的线程占用CPU
 - ◆ 如果多个线程的优先级相同，那么就随机选择一个线程，使其占用CPU
- 处于运行状态的线程会一直执行下去，直至它不得不放弃CPU
 - ◆ 以下情况会使线程放弃CPU：
 - JVM让其放弃CPU转入就绪状态，而让其他线程运行
 - 当前线程因为某些原因而进入阻塞状态
 - 线程运行结束
- 线程的调度不是跨平台的，不仅取决于JVM，还跟OS有关系

线程调度的实现方式

实现线程调度的方式

- 调整各个线程的优先级
 - ◆ 优先级高的线程获得较多的运行机会
 - ◆ 优先级低的线程获得较少的运行机会
- 让处于运行状态的线程调用 Thread.sleep() 方法
 - ◆ 处于运行状态的线程会转入睡眠状态，其它线程会获得CPU时间片
- 让处于运行状态的线程调用 Thread.yield() 方法
 - ◆ 处于运行状态的线程让位给同等优先级或优先级更高的线程运行
- 让处于运行状态的线程调用另一个线程的 join() 方法
 - ◆ 当前线程A调用另一个线程B的 join() 方法，当前线程A转入阻塞状态
 - ◆ 直到另一个线程B执行结束，线程A才恢复运行

线程的优先级

■ Thread 类中与线程优先级有关的方法和常量

■ 方法：

- ◆ 获取优先级： public final int getPriority()
- ◆ 设置优先级： public final void setPriority(int newPriority)

■ 常量：

- ◆ Thread.MAX_PRIORITY : 取值为 10 , 表示最高优先级
- ◆ Thread.MIN_PRIORITY : 取值为 1 , 表示最低优先级
- ◆ Thread.NORM_PRIORITY : 取值为 5 , 表示默认的优先级

线程的优先级

调整线程的优先级

- 扩展 Thread 类，实现一个线程
- 重写 run 方法
 - 其中循环输出线程的名程，然后睡眠 300 毫秒

```
public class Priority extends Thread{  
    public void run() {  
        for( int i = 0 ; i < 10 ; i++ ){  
            System.out.println( Thread.currentThread().getName() + "-" + i );  
            try {  
                Thread.sleep( 300 );  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

线程的优先级

■ 调整线程的优先级

- 创建 3 个线程
 - ◆ 为这三个线程名称，设置优先级(分别为 最低、默认、最高)
- 启动这 3 个线程

```
Priority pa = new Priority();
pa.setName("A");
pa.setPriority( Thread.MIN_PRIORITY ); // 最低优先级
pa.start();

Priority pb = new Priority();
pb.setName("B");
pb.setPriority( Thread.NORM_PRIORITY ); // 默认优先级
pb.start();

Priority pc = new Priority();
pc.setName("C");
pc.setPriority( Thread.MAX_PRIORITY ); // 最高优先级
pc.start();
```

线程睡眠 - Thread.sleep()

线程睡眠

- 当一个线程在运行中执行了 sleep() 方法后：
 - ◆ 它就会放弃 CPU 时间片，转到阻塞状态
 - ◆ 其它线程才有机会获得 CPU 时间片（不考虑优先级）
- Thread 中的 sleep 方法
 - ◆

```
public static native void sleep(long millis)
                           throws InterruptedException;
```

 - millis 参数用于设定睡眠时间，以毫秒为单位
 - ◆ 那个线程内部调用了该方法，就让那个线程转入睡眠（阻塞状态）

线程睡眠 - Thread.sleep()

■ 线程睡眠

■ 睡眠结束

- ◆ 自然醒：睡眠时间到了，就睡醒了
- ◆ 睡醒后不一定马上运行，而是转成就绪状态，等待时机

■ 中断睡眠

- ◆ 每个线程对象都有一个 interrupt() 方法，该方法用于中断睡眠
- ◆ 线程没睡够就被吵醒，称为线程中断
 - 线程被吵醒会哭的！中断线程会抛出 InterruptedException

线程睡眠 - Thread.sleep()

线程睡眠和中断睡眠举例

```
public class TestSleep extends Thread {  
    public void run() {  
        try {  
            Thread.sleep(6000);  
            System.out.println("呵呵，我睡醒了！！！");  
        } catch (InterruptedException e) {  
            System.out.println("谁吵醒我跟谁急！！！");  
            e.printStackTrace();  
        }  
        System.out.println("run 结束了!");  
    }  
  
    public static void main(String[] args) {  
        TestSleep ts = new TestSleep();  
        ts.start();  
        ts.interrupt(); // 中断 ts 的睡眠  
    }  
}
```

线程让步 - Thread.yield()

■ 线程运行中执行了 Thread.yield() 方法

- 如果有相同优先级或优先级更高的其他线程处于就绪状态
 - ◆ 把自己放到可运行池中，转为就绪状态
 - ◆ 把CPU时间片让给同等优先级或优先级更高的线程
 - ◆ 如果有多个同等优先级或更高优先级的线程，JVM会任意选取一个线程
 - 这里的任意选取一个，也包括当前线程本身
- 如果没有相同优先级或更高优先级的线程处于就绪状态
 - ◆ Thread.yield() 方法什么都不做，当前线程继续执行

线程让步 - Thread.yield()

线程让步举例

```
public class TestYield extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(currentThread() + " : i =" + i);  
            if (i == 5) {  
                Thread.yield(); // 当 i = 5 的时候让位  
                System.out.println( currentThread() + " 让位");  
            }  
        }  
    }  
    public static void main(String[] args) {  
        TestYield ty1 = new TestYield();    TestYield ty2 = new TestYield();  
        TestYield ty3 = new TestYield();    TestYield ty4 = new TestYield();  
        ty3.setPriority( Thread.NORM_PRIORITY + 2 );  
        ty1.start();      ty2.start();      ty3.start();      ty4.start();  
    }  
}  
//注意：整个程序的运行结果，结合优先级、线程状态作分析
```

sleep() 与 yield() 的异同

sleep() 与 yield() 方法的共同点

- 都属于 Thread 类的静态方法

- ◆ 可以使用 Thread 调用，也可以使用线程对象调用
- ◆ 不论是 sleep() 还是 yield() 都必须在线程内部调用
- ◆ 在线程外部调用这些方法，即便是使用某个线程对象调用，也不会对该线程对象有任何影响，反而是对该线程所在的线程有影响

```
public class MyThread extends Thread {
    public static void main(String[] args) {
        MyThread mt = new MyThread();           mt.start();
        mt.sleep( 1000 ); // 即便是基于 mt 调用了 sleep，对 mt 没有任何影响
    }                                         // 有影响的是 mt 所在的线程 main，即导致 main 睡眠1000ms
}
```

- 都会使处于运行状态的线程放弃CPU，给别的线程机会
 - ◆ 当前线程会放弃 CPU，或转入 就绪状态 或转入 阻塞状态

sleep() 与 yield() 的异同

■ sleep() 与 yield() 方法的区别

■ 是否考虑优先级

- ◆ sleep() 给其他线程运行的机会，不考虑其他线程的优先级，
 - 因此会给优先级较低的线程一个运行的机会
- ◆ yield() 只会给相同优先级或者更高优先级的线程一个运行的机会
 - 比当前线程优先级低的线程没有机会获得 CPU 时间片

■ 转入状态不同

- ◆ 线程调用 sleep(long millis) 后转入阻塞状态
- ◆ 线程调用 yield() 方法后直接转入就绪状态

sleep() 与 yield() 的异同

■ sleep() 与 yield() 方法的区别

■ 是否抛出异常

- ◆ sleep() 方法声明抛出 InterruptedException 异常
- ◆ yield() 方法不抛出任何异常

■ 可移植性

- ◆ sleep() 较 yield() 有更好的可移植性
- ◆ 不能依赖于 yield() 来提高程序的并发性能
- ◆ 推荐在测试时使用 yield()，其他情况能不用就不用

等待其他线程结束 - `join()`

■ 等待其他线程结束

- 当前运行的线程 A 调用另一个线程 B 的 `join()` 方法
 - 当前运行的线程 A 将转到阻塞状态
 - 当，另一个线程 B 运行结束了，A 才会恢复到就绪状态
-
- Thread 类的 `join()` 方法：
 - `public final void join() throws InterruptedException`
 - `public final synchronized void join(long millis)`
 - ◆ `millis` 参数用于设定当前线程被阻塞的时间，以毫秒为单位

等待其他线程结束 - join()

等待其他线程结束举例

```
public class TestJoin extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println( this.getName() + " : " + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        TestJoin tj = new TestJoin();  
        tj.start();  
        System.out.println("before join ");  
        try {  
            tj.join(); // 主线程调用 tj 的 join 方法  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("main end");  
    }  
}
```

设置为守护线程 - `setDaemon()`

■ 守护线程

- 也称之为后台线程、精灵线程
 - ◆ 当有非守护线程运行时，守护线程一定在运行
 - ◆ 所有的非守护线程都退出后，守护线程一定退出

```
public class DaemonThread extends Thread {  
    public void run() {  
        while( true ){    System.out.println("风吹鸡蛋壳"); }  
    }  
    public static void main(String[] args) throws Exception {  
        DaemonThread dt = new DaemonThread();  
        dt.setDaemon(true); // 设置 dt 线程为守护线程(必须在启动前设置)  
        System.out.println( dt.getName() + " 是守护线程吗: " + dt.isDaemon() );  
        dt.start(); // 启动线程  
        Thread.sleep( 10 ); // 主线程睡眠 10 毫秒  
        Thread main = Thread.currentThread();  
        System.out.println( main.getName() + " 是守护线程吗: " + main.isDaemon() );  
    }  
}
```

设置为守护线程 - `setDaemon()`

使用守护线程需注意

- 当所有非守护线程退出后，如果还有守护线程在运行，那么虚拟机就会终止这些守护线程
- 只有在启动线程 (`start()`) 之前，`setDaemon()` 才有效
 - ◆ 如果在线程启动之后才 `setDaemon()`，会引发 `IllegalThreadStateException`
- 非守护线程默认情况下创建的依然是非守护线程
- 守护线程默认情况下创建的依然是守护线程

线程安全问题

从银行取钱的问题

- 取钱的基本步骤
 - ◆ 验证帐号和密码
 - ◆ 输入取款金额
 - ◆ 判断账户余额是否大于取款金额
 - ◆ 如果余额大于取款金额，取款成功；否则，取款失败
- 对最后一步，如果可以取钱，那么可以拆分成两个部分
 - ◆ 银行帐号上的余额减少
 - ◆ 银行把现金支付给你
 - ◆ 这两个部分，应该是“成则同成，败则同败”（原子操作）
- 使用程序模拟上面的步骤
 - ◆ 使用 Account 模拟一个银行帐号，其中包括帐号、密码、余额
 - ◆ 建立一个线程类，用于模拟取钱过程
 - ◆ 建立测试类，在 main 方法中启动两个线程，模拟两个人取钱的过程

线程安全问题

■ 封装 Account 类

```
public class Account {  
    private String accountNo; // 帐号  
    private double balance; // 余额  
  
    public Account(){}
  
  
    public Account(String accountNo , double balance){  
        this.accountNo = accountNo;  
        this.balance = balance;  
    }  
  
    //此处省略 accountNo 和 balance 的 getter 和 setter  
  
}
```

线程安全问题

模拟取钱的线程类

```
public class GetMoney extends Thread {  
    private Account account;  
    private double money; // 交易额度  
    public GetMoney(){  
    }  
    public GetMoney(Account account, double money) {  
        this.account = account;  
        this.money = money;  
    }  
    public void run() {  
        System.out.println("本次取款金额为: " + money );  
        if( account.getBalance() > money ) {  
            System.out.println("取款前账户余额为: " + account.getBalance() );  
            System.out.println("请收好您的钞票[ " + money + "元] , 欢迎下次光临");  
            /*****  
             *  
             *  
             *  
             *  
             */  
            account.setBalance( account.getBalance() - money );  
            System.out.println("取款后账户余额为: " + account.getBalance() );  
        }else{  
            System.out.println("当前余额为: " + account.getBalance() + " , 余额不足, 不能取出" );  
        }  
    }  
}
```

线程安全问题

测试类 – TestGetMoney

```
public class TestGetMoney {  
    public static void main(String[] args) {  
        Account accb = new Account("令狐冲", 1000); // 传入帐号和账户余额  
        GetMoney a = new GetMoney( accb ,800 ); // 传入 Account 和 取款金额  
        a.start();  
        GetMoney b = new GetMoney( accb ,800 ); // 传入 Account 和 取款金额  
        b.start();  
    }  
}
```

您本次取款金额为: 800.0

取款前您的账户余额为: 1000.0

请收好您的钞票 [800.0] , 欢迎下次光临!

您当前的存款余额为: 200.0

您本次取款金额为: 800.0

您当前的存款余额为: 200.0, 不满足你本次要取出的金额

测试环境为
双核双线程CPU

线程安全问题

搞点破坏

- 在前面的 GetMoney 类中，把 `*****` 替换成如下代码

```
try {  
    Thread.sleep( 1 ); // 让处于运行中的线程睡眠 1 毫秒  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

- 之后再度执行 TestGetMoney 类，结果为：

您本次取款金额为: 800.0

取款前您的账户余额为: 1000.0

请收好您的钞票 [800.0]，欢迎下次光临！

您本次取款金额为: 800.0

取款前您的账户余额为: 1000.0

请收好您的钞票 [800.0 元]，欢迎下次光临！

您当前的存款余额为: 200.0

您当前的存款余额为: -600.0

测试环境为
双核双线程CPU

同步代码块

解决多线程下的线程安全问题 - 同步锁

- Java的多线程支持中引入了同步锁来解决前述问题
- 使用同步锁的通用方法就是使用同步代码块

同步代码块的语法

```
synchronized( object ){ // 这里的 object 即为同步锁  
    //这里就是需要同步的代码块  
}
```

同步的原理

- 一般以多条线程共享的资源(竞争资源)作为同步锁
- 同步的基本原理是：加锁 → 同步代码块顺利执行 → 释放锁

同步代码块

使用同步代码块解决前面的线程安全问题

- 以 GetMoney 中的 account 作为同步锁，修改 run() 方法

```
public void run() {  
    synchronized( account ){  
        System.out.println("本次取款金额为: " + money );  
        if( account.getBalance() > money ){  
            System.out.println("取款前账户余额为: " + account.getBalance() );  
            System.out.println("请收好您的钞票[ " + money + "元]，欢迎下次光临");  
            try {  
                Thread.sleep( 1 ); // 让处于运行中的线程睡眠 1 毫秒  
            } catch (InterruptedException e) { e.printStackTrace(); }  
            account.setBalance( account.getBalance() - money );  
            System.out.println( "取款后账户余额为: " + account.getBalance() );  
        }else{  
            System.out.println( "当前余额为: " + account.getBalance() + "，余额不足，不能取出" );  
        }  
    }  
}
```

同步代码块

■ 再度运行测试类 TestGetMoney

- 使用同步代码块后，结果为：

```
您本次取款金额为: 800.0
```

```
取款前您的账户余额为: 1000.0
```

```
请收好您的钞票 [ 800.0 ] , 欢迎下次光临!
```

```
您当前的存款余额为: 200.0
```

```
您本次取款金额为: 800.0
```

```
您当前的存款余额为: 200.0, 不满足你本次要取出的金额
```

- 对于以上结果，无论单核CPU、双核CPU，结果一样

同步方法

■ 同步方法

- 与同步代码块对应，Java多线程安全支持还提供了同步方法
- 同步方法是指被 synchronized 关键字修饰的方法
- 对于同步方法而言，无需显式指定同步锁
 - ◆ 同步方法的同步锁是 this，就是当前对象本身

■ 线程安全的类

- 使用同步方法可以非常方便地把某个类变成线程安全的类
- 线程安全的类，比如 StringBuffer 有以下特征
 - ◆ 该类的对象可以被多个线程安全地访问
 - ◆ 每个线程调用改对象的任意方法之后，都将得到正确的结果
 - ◆ 每个线程调用改对象的任意方法之后，该对象状态依然保持合理状态

同步方法

使用同步方法解决前面的线程安全问题

- 在 Account 类中增加一个对 balance 安全访问的方法 draw()

```
public synchronized void draw( double money ) {
    System.out.println("本次取款金额为: " + money + " 元");
    if( this.getBalance() > money ){
        System.out.println("取款前账户余额为: " + this.getBalance() );
        System.out.println("请收好您的钞票[" + money + "]，欢迎下次光临");
        try {
            Thread.sleep( 1 );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.balance -= money;
        System.out.println( "取款后账户余额为: " + this.getBalance() );
    } else {
        System.out.println( "当前余额为: " + account.getBalance() + "，余额不足，不能取出" );
    }
}
```

同步方法

■ 修改 GetMoney 中的 run 方法

```
public void run() {  
    account.draw( this.money );  
}
```

■ 运行测试类 TestGetMoney

您本次取款金额为: 800.0 元

取款前您的账户余额为: 1000.0

请收好您的钞票 [800.0 元] , 欢迎下次光临!

您当前的存款余额为: 200.0

您本次取款金额为: 800.0 元

您当前的存款余额为: 200.0, 不满足你本次要取出的金额

同步方法

实现线程安全的指导思想

- 可变类的线程安全是以牺牲程序的效率为代价的
- 只对那些会改变竞争资源的方法进行同步
 - ◆ 所谓的竞争资源，也就是共享资源
 - 前述例子中的 Account 的 balance 属性属于竞争资源
 - 而前例中的 Account 中的 accountNo 属性不属于竞争资源
 - ◆ 不要对所有的方法都进行同步
- 最好提供两个版本
 - ◆ 如果可变类有两种运行环境：单线程环境和多线程环境
 - ◆ 则应该为该类提供两个版本：单线程版本和多线程版本
 - 单线程版本用于单线程环境下保证性能
 - 多线程版本用于多线程环境下保证线程安全

线程同步的特征

■ 线程同步的特点

- 有同步代码块仍然有可能造成线程安全问题
 - ◆ 如果一个同步代码块和非同步代码块同时操纵共享资源，仍然会造成对共享资源的竞争
 - ◆ 程序员必须准确把握那些资源必须用同步代码块访问
- 每个对象都有惟一的同步锁
 - ◆ 前述举例中，如果有多个 Account 对象，每个对象都会有其独立的一把锁
- 在静态方法前面也可以使用 synchronized 关键字
 - ◆ 无论对普通方法还是对静态方法，而其同步锁是多个线程竞争的那个资源
 - ◆ 对于普通方法，其同步锁即为 this
 - ◆ 静态方法的同步锁，需要根据实际代码来确定(本质上还是被竞争的资源)
- 一个线程开始执行同步代码块时，不一定以不中断的方式运行
 - ◆ 当在线程中调用了 sleep() 或者 yield() 都有可能导致线程中断

同步与并发

身边的例子 - 从接水说起

设计饮水机类

- 指定饮水机的水量以及接水的方法

```
// 饮水机类
public class WaterDispenser {

    private int amount = 18900; // 饮水机水桶中水的剩余量(单位:ml)

    public void draw() {
        amount -= 50; // 每取一次水，剩余量减少 50ml
        String name = Thread.currentThread().getName(); // 获得当前线程的名字
        System.out.println(name + " 打水后，水剩余量：" + amount + " ml");
    }
}
```

同步与并发

身边的例子 - 从接水说起

- 设计需要接水的人对应的类(继承 Thread 类)
 - 每个接水的人在“获得”饮水机后期望接 300ml 水

```
public class Person extends Thread {  
    private WaterDispenser wd; // 饮水机  
    public void setWd(WaterDispenser wd) {  
        this.wd = wd;  
    }  
    public void run() {  
        synchronized (wd) { // 同步  
            for (int i = 0; i < 6; i++) {  
                wd.draw(); // 分 6 次打, 总共打300ml  
                Thread.yield();  
            }  
        }  
    }  
}
```

同步与并发

身边的例子 - 从接水说起

- 模拟 10 个人接水

```
public class TestDrawWater {  
    public static void main(String[] args) {  
        // 创建 饮水机 对象  
        WaterDispenser midea = new WaterDispenser();  
        // 创建 容量为 10 的 Person 数组  
        Person[] persons = new Person[ 10 ];  
        // 通过循环对 Person 数组各个元素初始化，并启动线程  
        for (int i = 0; i < persons.length; i++) {  
            persons[ i ] = new Person();  
            persons[ i ].setWd( midea );  
            persons[ i ].start() // 启动线程  
        }  
    }  
}
```

同步与并发

身边的例子 - 从接水说起

■ 同步

- ◆ 没有抢到饮水机的人只能等抢到饮水机的人打完了300ml才能进行下一次抢饮水机的动作
- ◆ 最后一个抢到饮水机的人一定很郁闷！（因为都接到水了没人跟他抢了）

■ 并发

- ◆ 为了提高并发性能，应该使同步代码块中的操作尽可能少
- ◆ 这样可以使一个线程尽快释放锁，减少其他线程等待锁的时间
- ◆ 举例请看后续的 Person2.java
 - 测试过程不变，还用 TestDrawWater 中的步骤测试

■ 以上可以看出，同步和并发是此消彼长的

同步与并发

理解并发

- 重新设计需要接水的人对应的类
 - 去掉其中的同步代码块

```
public class Person2 extends Thread {  
    private WaterDispenser wd; // 饮水机  
    public void setWd(WaterDispenser wd) {  
        this.wd = wd;  
    }  
    public void run() {  
        // synchronized (wd) { // 同步  
            for (int i = 0; i < 6; i++) {  
                wd.draw(); // 分十次打，总共打500ml  
                Thread.yield();  
            }  
        // }  
    }  
}
```

释放同步锁

■ 释放同步锁的锁定

- 当前线程的同步代码块、同步方法执行完毕
- 当前线程的同步代码块、同步方法遇到 break、return 语句
- 当前线程的同步代码块、同步方法异常结束
 - ◆ 比如出现了未处理的 Error 或者 Exception
- 当前线程执行同步代码块、同步方法时，程序执行了同步锁的 wait() 方法，当前线程暂停，并释放同步锁
 - ◆ 关于 wait() 方法在 下一节线程通信中讲解

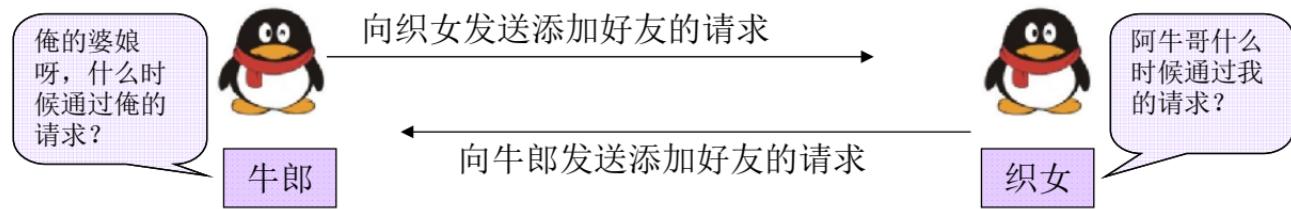
■ 不释放同步锁的情况

- 线程执行同步代码块、同步方法时，调用 Thread.sleep() 或 Thread.yield() 来暂停当前线程，当前线程不会释放同步锁
- 线程执行同步代码块时，其他线程调用了该线程的 suspend 方法将该线程挂起，该线程不会释放同步锁
 - ◆ suspend() 已被废除

死锁

什么是死锁

- 当一个线程A正在等待另一个线程B持有的锁
- 而线程B正在等待线程A持有的锁时，就会发生死锁



避免死锁

- 对于死锁现象，JVM不检测也不试图避免这种情况
- 避免死锁，程序员责无旁贷

线程通信

为什么要通信？

- 我们知道，不同的线程执行不同的任务
- 如果这些任务之间有某种联系，不同线程之间就必须通信
- 通信的目的是为了协同完成同一个工作
- 陌生的熟人：迅雷
 - ◆ 之所以号称以迅雷不及掩耳之势下载，依赖的就是多线程技术
 - ◆ 问题：
 - ◆ 迅雷的多个线程怎么样操作同一个文件？
 - ◆ 不同线程会不会重复下载的内容已经下载过的内容？
 - ◆ 同一个文件，会不会漏掉一部分，没有下载？



线程通信

解决问题

实例

- 用多线程下载一个 5 M 的文件：老鼠爱大米.mp3
- 假定最大活跃线程数为 3，用户设定的缓存大小为 1024 kb

一种实现可能

- 迅雷的 3 个线程A、B、C负责从网络读取数据
- 三个线程共同完成 1024 kb 的内容(对文件来说是连续的 1024 kb)
- 缓存满了，就输出到磁盘上(线程D)
- 然后继续读取下一个 1024 kb 的内容
- 以上可以看出，三个线程 A、B、C需要实现互相通信



线程通信的涵义

■ 我们研究的线程通信有以下涵义

- 多个线程协调运行 (依赖于Object的5个方法)
 - ◆ wait() : 导致当前线程等待
 - ◆ notify() : 唤醒在此同步锁上等待的单个线程(选择具有任意性)
 - ◆ notifyAll() : 唤醒在此同步锁上等待的所有线程
- 多线程通信
 - ◆ 通过访问共享变量的方式 (注:需要处理同步问题)
 - a) 通过内部类实现线程的共享变量
 - b) 通过实现 Runnable 接口实现线程的共享变量
 - ◆ 通过管道流
 - PipedInputStream / PipedOutputStream
 - PipedReader / PipedWriter
 - PipedSinkChannel / PipedSourceChannel (属于nio 范围)

多个线程协调运行

多线程协调运行的实现

- 必须借助 Object 类的 wait()、notify()、notifyAll() 方法实现
- 这三个方法，必须是基于同步锁的调用，而不是基于线程的调用
 - ◆ 对于 synchronized 修饰的方法
 - 因为该类的实例(this)即为同步锁
 - 因此可以在同步方法中使用 this 关键字调用这3个方法(或者省略this关键字直接使用)

```
public synchronized void testWait() {
    this.wait(); // 调用当前类的 wait() 方法
}
```

- ◆ 对于 synchronized 修饰的同步代码块
 - 其同步锁是 synchronized 后面的括号里面的对象
 - 因此必须通过该对象调用这3个方法

```
synchronized ( obj ) {
    obj.wait(); // 基于 obj 调用 wait()，即调用 obj 的 wait() 方法
}
```

多个线程协调运行

对三个方法的说明

■ wait()

- ◆ 导致当前线程等待(进入等待池)，直至其它线程调用该同步锁的 notify() 方法或者 notifyAll() 方法将其唤醒
- ◆ 调用该同步锁的 wait() 方法的当前线程会释放对该同步锁的锁定
- ◆ wait() 方法有三种重载形式：
 - wait() :一直等待，直到有其它线程通知唤醒
 - wait(long timeout) :在 timeout 毫秒后自动苏醒
 - wait(long timeout, int nanos) :在 timeout 毫秒 和 nanos 微妙后自动苏醒

■ notify()

- ◆ 唤醒在此同步锁上等待的单个线程
- ◆ 如果所有的线程都在此同步锁上等待，则会任意唤醒其中的一个线程
- ◆ 只有当前线程放弃对该同步锁的锁定后(使用wait()方法)，才可以执行被唤醒的线程

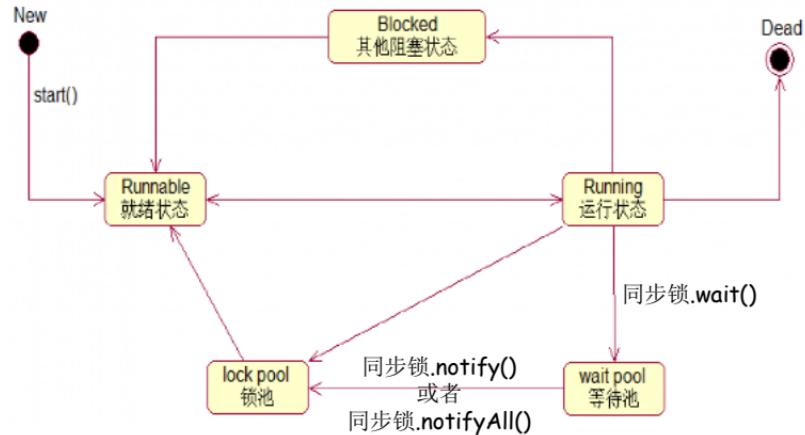
多个线程协调运行

对三个方法的说明

■ `notifyAll()`

- ◆ 唤醒正在等待该同步锁的所有线程
- ◆ 只有当前线程放弃对该同步锁的锁定后，才可以执行被唤醒的线程

图解三个方法的作用



多个线程协调运行举例

```
public class ICBCAccount {
```

帐号类 1

```
    private String accountNo; //帐号  
    private double balance; //余额  
    private char flag = '无'; //代表是否有钱的标记: 有- 帐号上有钱, 无- 帐号上没有钱
```

```
    public ICBCAccount( String accountNo , double balance ){  
        this.accountNo = accountNo;  
        if( balance > 0 ){  
            this.balance = balance;  
            this.flag = '有';  
        }  
    }
```

//取钱方法

```
    public synchronized void draw( double drawAmount ){  
        try {  
            if( this.flag == '无' ){  
                this.wait();  
            } else {  
                System.out.println( Thread.currentThread().getName() + " - 取钱[" + drawAmount + "]");  
                this.balance -= drawAmount;  
                System.out.println( Thread.currentThread().getName() + " - 余额[" + this.balance + "]");  
                this.flag = '无';  
                this.notifyAll();  
            }  
        } catch (InterruptedException e) { e.printStackTrace(); }  
    } //接下页
```

多个线程协调运行举例

//接上页

//存钱方法

```
public synchronized void deposit( double depositAmount ){  
    try {  
        if( this.flag == '有' ){  
            this.wait();  
        } else {  
            System.out.println( Thread.currentThread().getName() + " - 存钱["+ depositAmount +"]");  
            this.balance += depositAmount;  
            System.out.println( Thread.currentThread().getName() + " - 余额["+ this.balance +"]");  
            this.flag = '有';  
            this.notifyAll();  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

帐号类 2

多个线程协调运行举例

```
public class DrawThread extends Thread {  
    private ICBCAccount account;  
    private double drawAmount;  
    public DrawThread(String threadName ,  
                      ICBCAccount account,  
                      double drawAmount) {  
        super( threadName );  
        this.account = account;  
        this.drawAmount = drawAmount;  
    }  
    public void run() {  
        for( int i = 0 ; i < 100 ; i++ ){  
            account.draw(drawAmount);  
        }  
    }  
}
```

取钱者线程类

多个线程协调运行举例

```
public class DepositThread extends Thread {  
    private ICBCAccount account;  
    private double depositAmount;  
    public DepositThread(String threadName ,  
                         ICBCAccount account,  
                         double depositAmount) {  
        super( threadName );  
        this.account = account;  
        this.depositAmount = depositAmount;  
    }  
    public void run() {  
        for( int i = 0 ; i < 10 ; i++ ){  
            account.deposit( depositAmount );  
        }  
    }  
}
```

多个线程协调运行举例

```
public class ICBCHall {
    public static void main(String[] args) {
        ICBCAccount account = new ICBCAccount("奥巴马", 0);
        new DepositThread("甲", account, 800).start();
        new DrawThread("取钱者", account, 800).start();
        new DepositThread("乙", account, 800).start();
        new DepositThread("丙", account, 800).start();
    }
}
```

测试类(ICBC 营业厅)

注意：

线程最后的状态是在等待，即都处在等待池中，因为最后没有任何线程再调用同步锁的 `notify()` 或者 `notifyAll()` 来唤醒其中一个线程

最后获得运行机会的线程，因为执行了 `run` 中的代码，本质上是调用 `account` 的一个同步方法，从而进入了等待池，而这之后没有线程再度唤醒所有的线程或一个线程

最后的状态(被阻塞)，是在等待池，不是死锁现象，这点必须注意！

多线程通信

■ 使用管道流实现多线程通信

- 创建管道输入流和管道输出流 (使用 new 关键字)
 - ◆ PipedWriter pw = new PipedWriter();
 - ◆ PipedReader pr = new PipedReader();
- 使用管道输入流或者管道输出流的 connect 方法连接这对输入流和输出流
 - ◆ pw.connect(pr);
- 将管道输入流和管道输出流分别传给两个线程
 - ◆ new WriterThread("写数据线程" , pw).start();
 - ◆ new ReaderThread("读数据线程" , pr).start();
- 两个线程可以分别依赖各自的管道输入流、管道输出流进行通信
 - ◆ WriterThread 线程类的 run 方法中通过 pw 不断往外写数据
 - ◆ ReaderThread 线程类的 run 方法中通过 pr 不断地读取数据

定时器

■ 定时器用于定时执行特定的任务

- 通过 `java.util.Timer` 类的对象可以定时执行某项特定任务
 - ◆ `Timer` 类本身没有继承 `Thread` 也没有实现 `Runnable` 接口
 - ◆ 用于设定定时任务的常用方法
 - `schedule(TimerTask task, long delay, long period)`
 - » `task` : 即为需要定时执行的任务
 - » `delay` : 表示延迟执行的时间(单位为毫秒)
 - » `period` : 表示每次执行任务的间隔时间(单位为毫秒)
 - ◆ 取消任务的方法: `cancel()`
 - 所谓特定任务, 是指 `java.util.TimerTask` 类型的对象
 - ◆ `TimerTask` 本身是个抽象类
 - 想要得到一个 `TimerTask` 类型的对象, 必须使用 `TimerTask` 的子类
 - ◆ `TimerTask` 实现了 `Runnable` 接口, 但是没有实现其中的 `run()` 方法

定时器

定时器应用举例

```
public class TimerDemo {  
    public static void main(String[] args) {  
        final Timer timer = new Timer(); // 实例化Timer类  
        timer.schedule(new TimerTask() {  
            public void run() {  
                SimpleDateFormat f = new SimpleDateFormat("HH:mm:ss");  
                String currentTime = f.format( new Date() );  
                System.out.println( currentTime );  
                if ( currentTime.equals("22:39:00") ) {  
                    timer.cancel(); // 停止定时器  
                }  
            }  
        }, 0, 1000); // 定时器执行间隔时间，单位毫秒  
    }  
}
```



为之，则难者亦易矣；不为，则易者亦难矣！

www.oracleacademy.net