一、认识多任务、多进程、单线程、多线程

要认识多线程就要从操作系统的原理说起。

以前古老的DOS操作系统（V 6.22）是单任务的，还没有线程的概念，系统在每次只能做一件事情。比如你在copy东西的时候不能rename文件名。为了提高系统的利用效率，采用批处理来批量执行任务。

现在的操作系统都是多任务操作系统，每个运行的任务就是操作系统所做的一件事情，比如你在听歌的同时还在用MSN和好友聊天。听歌和聊天就是两个任务，这个两个任务是“同时”进行的。一个任务一般对应一个进程，也可能包含好几个进程。比如运行的MSN就对应一个MSN的进程，如果你用的是windows系统，你就可以在任务管理器中看到操作系统正在运行的进程信息。

一般来说，当运行一个应用程序的时候，就启动了一个进程，当然有些会启动多个进程。启动进程的时候，操作系统会为进程分配资源，其中最主要的资源是内存空间，因为程序是在内存中运行的。在进程中，有些程序流程块是可以乱序执行的，并且这个代码块可以同时被多次执行。实际上，这样的代码块就是线程体。线程是进程中乱序执行的代码流程。当多个线程同时运行的时候，这样的执行模式成为并发执行。

多线程的目的是为了最大限度的利用CPU资源。

Java编写程序都运行在在Java虚拟机（JVM）中，在JVM的内部，程序的多任务是通过线程来实现的。每用java命令启动一个java应用程序，就会启动一个JVM进程。在同一个JVM进程中，有且只有一个进程，就是它自己。在这个JVM环境中，所有程序代码的运行都是以线程来运行。

一般常见的Java应用程序都是单线程的。比如，用java命令运行一个最简单的HelloWorld的Java应用程序时，就启动了一个JVM进程，JVM找到程序程序的入口点main()，然后运行main()方法，这样就产生了一个线程，这个线程称之为主线程。当main方法结束后，主线程运行完成。JVM进程也随即退出 。

对于一个进程中的多个线程来说，多个线程共享进程的内存块，当有新的线程产生的时候，操作系统不分配新的内存，而是让新线程共享原有的进程块的内存。因此，线程间的通信很容易，速度也很快。不同的进程因为处于不同的内存块，因此进程之间的通信相对困难。

实际上，操作的系统的多进程实现了多任务并发执行，程序的多线程实现了进程的并发执行。多任务、多进程、多线程的前提都是要求操作系统提供多任务、多进程、多线程的支持。

在Java程序中，JVM负责线程的调度。线程调度是值按照特定的机制为多个线程分配CPU的使用权。

调度的模式有两种：分时调度和抢占式调度。分时调度是所有线程轮流获得CPU使用权，并平均分配每个线程占用CPU的时间；抢占式调度是根据线程的优先级别来获取CPU的使用权。JVM的线程调度模式采用了抢占式模式。

所谓的“并发执行”、“同时”其实都不是真正意义上的“同时”。众所周知，CPU都有个时钟频率，表示每秒中能执行cpu指令的次数。在每个时钟周期内，CPU实际上只能去执行一条（也有可能多条）指令。操作系统将进程线程进行管理，轮流（没有固定的顺序）分配每个进程很短的一段是时间（不一定是均分），然后在每个线程内部，程序代码自己处理该进程内部线程的时间分配，多个线程之间相互的切换去执行，这个切换时间也是非常短的。因此多任务、多进程、多线程都是操作系统给人的一种宏观感受，从微观角度看，程序的运行是异步执行的。

用一句话做总结：虽然操作系统是多线程的，但CPU每一时刻只能做一件事，和人的大脑是一样的，呵呵。

二、Java与多线程

Java语言的多线程需要操作系统的支持。

Java 虚拟机允许应用程序并发地运行多个执行线程。Java语言提供了多线程编程的扩展点，并给出了功能强大的线程控制API。

在Java中，多线程的实现有两种方式：

扩展java.lang.Thread类

实现java.lang.Runnable接口

每个线程都有一个优先级，高优先级线程的执行优先于低优先级线程。每个线程都可以或不可以标记为一个守护程序。当某个线程中运行的代码创建一个新 Thread 对象时，该新线程的初始优先级被设定为创建线程的优先级，并且当且仅当创建线程是守护线程时，新线程才是守护程序。

当 Java 虚拟机启动时，通常都会有单个非守护线程（它通常会调用某个指定类的 main 方法）。Java 虚拟机会继续执行线程，直到下列任一情况出现时为止：

调用了 Runtime 类的 exit 方法，并且安全管理器允许退出操作发生。

非守护线程的所有线程都已停止运行，无论是通过从对 run 方法的调用中返回，还是通过抛出一个传播到 run 方法之外的异常。

三、扩展java.lang.Thread类

/\*\*

 \* File Name:   TestMitiThread.java

 \* Created by:  IntelliJ IDEA.

 \* Copyright:   Copyright (c) 2003-2006

 \* Date Time:   2007-5-17 10:03:12

 \* Readme:      通过扩展Thread类实现多线程

 \*/

public class TestMitiThread {

    public static void main(String[] rags) {

        System.out.println(Thread.currentThread().getName() + " 线程运行开始!");

        new MitiSay("A").start();

        new MitiSay("B").start();

        System.out.println(Thread.currentThread().getName() + " 线程运行结束!");

    }

}

class MitiSay extends Thread {

    public MitiSay(String threadName) {

        super(threadName);

    }

    public void run() {

        System.out.println(getName() + " 线程运行开始!");

        for (int i = 0; i < 10; i++) {

            System.out.println(i + " " + getName());

            try {

                sleep((int) Math.random() \* 10);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

        System.out.println(getName() + " 线程运行结束!");

    }

}

运行结果：

main 线程运行开始!

main 线程运行结束!

A 线程运行开始!

0 A

1 A

B 线程运行开始!

2 A

0 B

3 A

4 A

1 B

5 A

6 A

7 A

8 A

9 A

A 线程运行结束!

2 B

3 B

4 B

5 B

6 B

7 B

8 B

9 B

B 线程运行结束!

说明：

程序启动运行main时候，java虚拟机启动一个进程，主线程main在main()调用时候被创建。随着调用MitiSay的两个对象的start方法，另外两个线程也启动了，这样，整个应用就在多线程下运行。

在一个方法中调用Thread.currentThread().getName()方法，可以获取当前线程的名字。在mian方法中调用该方法，获取的是主线程的名字。

注意：start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。

从程序运行的结果可以发现，多线程程序是乱序执行。因此，只有乱序执行的代码才有必要设计为多线程。

Thread.sleep()方法调用目的是不让当前线程独自霸占该进程所获取的CPU资源，以留出一定时间给其他线程执行的机会。

实际上所有的多线程代码执行顺序都是不确定的，每次执行的结果都是随机的。

四、实现java.lang.Runnable接口

/\*\*

 \* 通过实现 Runnable 接口实现多线程

 \*/

public class TestMitiThread1 implements Runnable {

    public static void main(String[] args) {

        System.out.println(Thread.currentThread().getName() + " 线程运行开始!");

        TestMitiThread1 test = new TestMitiThread1();

        Thread thread1 = new Thread(test);

        Thread thread2 = new Thread(test);

        thread1.start();

        thread2.start();

        System.out.println(Thread.currentThread().getName() + " 线程运行结束!");

    }

    public void run() {

        System.out.println(Thread.currentThread().getName() + " 线程运行开始!");

        for (int i = 0; i < 10; i++) {

            System.out.println(i + " " + Thread.currentThread().getName());

            try {

                Thread.sleep((int) Math.random() \* 10);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

        System.out.println(Thread.currentThread().getName() + " 线程运行结束!");

    }

}

运行结果：

main 线程运行开始!

Thread-0 线程运行开始!

main 线程运行结束!

0 Thread-0

Thread-1 线程运行开始!

0 Thread-1

1 Thread-1

1 Thread-0

2 Thread-0

2 Thread-1

3 Thread-0

3 Thread-1

4 Thread-0

4 Thread-1

5 Thread-0

6 Thread-0

5 Thread-1

7 Thread-0

8 Thread-0

6 Thread-1

9 Thread-0

7 Thread-1

Thread-0 线程运行结束!

8 Thread-1

9 Thread-1

Thread-1 线程运行结束!

说明：

TestMitiThread1类通过实现Runnable接口，使得该类有了多线程类的特征。run（）方法是多线程程序的一个约定。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是扩展Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

五、读解Thread类API

static int MAX\_PRIORITY

          线程可以具有的最高优先级。

static int MIN\_PRIORITY

          线程可以具有的最低优先级。

static int NORM\_PRIORITY

          分配给线程的默认优先级。

构造方法摘要

Thread(Runnable target)

          分配新的 Thread 对象。

Thread(String name)

          分配新的 Thread 对象。

方法摘要

static Thread currentThread()

          返回对当前正在执行的线程对象的引用。

 ClassLoader getContextClassLoader()

          返回该线程的上下文 ClassLoader。

 long getId()

          返回该线程的标识符。

 String getName()

          返回该线程的名称。

 int getPriority()

          返回线程的优先级。

 Thread.State getState()

          返回该线程的状态。

 ThreadGroup getThreadGroup()

          返回该线程所属的线程组。

static boolean holdsLock(Object obj)

          当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。

 void interrupt()

          中断线程。

static boolean interrupted()

          测试当前线程是否已经中断。

 boolean isAlive()

          测试线程是否处于活动状态。

 boolean isDaemon()

          测试该线程是否为守护线程。

 boolean isInterrupted()

          测试线程是否已经中断。

 void join()

          等待该线程终止。

 void join(long millis)

          等待该线程终止的时间最长为 millis 毫秒。

 void join(long millis, int nanos)

          等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

 void resume()

          已过时。 该方法只与 suspend() 一起使用，但 suspend() 已经遭到反对，因为它具有死锁倾向。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。

 void run()

          如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。

 void setContextClassLoader(ClassLoader cl)

          设置该线程的上下文 ClassLoader。

 void setDaemon(boolean on)

          将该线程标记为守护线程或用户线程。

static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

          设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序。

 void setName(String name)

          改变线程名称，使之与参数 name 相同。

 void setPriority(int newPriority)

          更改线程的优先级。

 void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

          设置该线程由于未捕获到异常而突然终止时调用的处理程序。

static void sleep(long millis)

          在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。

static void sleep(long millis, int nanos)

          在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行）。

 void start()

          使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

 void stop()

          已过时。 该方法具有固有的不安全性。用 Thread.stop 来终止线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 ThreadDeath 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。stop 的许多使用都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。如果目标线程等待很长时间（例如基于一个条件变量），则应使用 interrupt 方法来中断该等待。

          已过时。 该方法具有固有的不安全性。请参阅 stop() 以获得详细信息。该方法的附加危险是它可用于生成目标线程未准备处理的异常（包括若没有该方法该线程不太可能抛出的已检查的异常）。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。

 void suspend()

          已过时。 该方法已经遭到反对，因为它具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，则在目标线程重新开始以前任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume 之前锁定该监视器，则会发生死锁。这类死锁通常会证明自己是“冻结”的进程。有关更多信息， String toString()

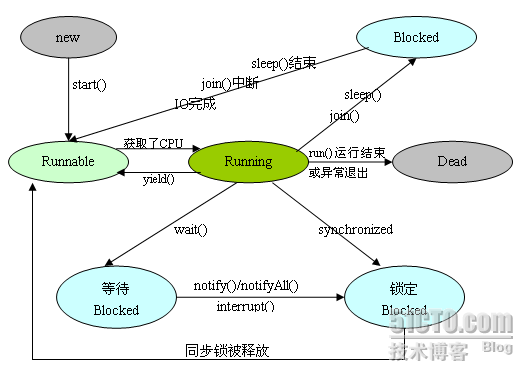
          返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

static void yield()

          暂停当前正在执行的线程对象，并执行其他线程。

六、线程的状态转换图

线程在一定条件下，状态会发生变化。线程变化的状态转换图如下：



1、新建状态（New）：新创建了一个线程对象。

2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3、运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。

（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5、死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

七、线程的调度

1、调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX\_PRIORITY

          线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

          线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

          分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

7、常见线程名词解释

主线程：JVM调用程序mian()所产生的线程。

当前线程：这个是容易混淆的概念。一般指通过Thread.currentThread()来获取的进程。

后台线程：指为其他线程提供服务的线程，也称为守护线程。JVM的垃圾回收线程就是一个后台线程。

前台线程：是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过isDaemon()和setDaemon()方法来判断和设置一个线程是否为后台线程。