Java 语言的多线程技术

马希荣 （银川宁夏大学计算中心 750021）

摘要：Java 语言的一个重要特点就是支持多线程机制，目前最新发展起来的操作系统，如 windows NT，windows 95 等都采用多线程概念，把线程视为基本的执行单位。语言自身支持多线程机制可以为程序设计者在运用多线程功能上带来方便，Java 语言在这方面开了先河。

关键词：Java 线程 多线程 操作系统 进程

一、引言

就计算机的发展史而言，多线程概念的产生和采用多线程操作系统的出现是近几年的事情。随着个人计算机微处理器的飞速发展，个人计算机上的操作系统也纷纷采用多任务和分时设计，将早期只有大型计算机才具有的系统特性，带给了个人计算机。

Java 语言支持多线程机制，使应用程序能够并行处理多个任务，Java 语言中的基本函数库中的基本函数已定义了 Thread 这个基本类，内置了一组方法，使程序设计者编写多线程程序时，只要继承这个类，就可以利用原编写的方法，生成一个新的线程，执行一个线程，终止一个线程的任务，或者查看执行状态。

由于目前网络上传输文件或程序的速度仍然受到相当大的限制，利用 java 程序的多线程功能，可以在播放一个动画文件时，以多线程的方式同时传送每一帧图片，这样就提高了网络上的传输效率。

二、什么是多线程

可以在同一时间执行多个程序的操作系统，都有进程的概念，一个进程就是一个执行中的程序。每个进程之间是独立的，独占一块内存空间、一组系统资源，各进程是不知道彼此存在的。例如在 Unix 系统中，派生出一个进程的进程实际上是将父进程的代码段和数据段重新拷贝到子进程的相应空间。父进程和子进程相互独立，除非利用某些通信渠道来进行交流，或者通过操作系统产生交互作用。进程的概念较好地解决了多任务问题，但开销庞大。

为了减少不必要的系统负担，线程的概念便应运而生。所谓线程，也是一个执行的程序，但是与进程不同的是，多个线程共享一块内存空间和一组系统资源，而且线程本身的数据，通常只有微处理器的寄存数据，以及一个供程序执行时使用的堆栈。所以系统在产生一个线

程，或者多个线程之间进行切换时，负担比进程轻便的多，这样线程又称为轻负荷进程(light－weight process)。

三、Java 的多线程机制

新兴的操作系统，如 windows NT，windows 95 等，大多都采用了多线程概念，把线程视为基本的执行单位。在这种情况下，无线程功能的程序语言，就没有把相关的功能设计到其语言的结构之中，因此，造成了程序设计运用多线程功能上的不便和其他问题。若程序语言自身已支持了多线程概念，程序设计就不必再关心该操作系统的多线程功能如何使用了，当然在网络上可能有许多操作系统要使用 java 语言写的程序，而这些操作系统所提供的多线程功能不尽相同，这样 java 的多线程程序，有时候可能会遇到执行上的困难，但 java 语言的线程是强制性的，同时是分时来执行的。

1. 线程的创建

在 Java 语言中，创建线程有两种方式。一是实现一个 Runnable 接口，二是扩充 Thread 类。

采用实现接口 Runnable 创建线程的一种方法是：定义一个类为 Runnable 的实现，并传递此对象的一个引用给 Thread 的构造函数。

例如：
```
public class mxr extends Applet{
    public void init()
    {
    mator = new mator();
    new Thread(mator).start();
    }
    }
    Class mator implements Runnable{
    ...
    public void run()
    {
    ...
    }
    }
```

类也可以包含一个消息委托的线程，其实现方法如下例：
```
public class mxr extends Applet{
public void init()
{
mator = new mator();
mator.star();
}
}
Class mator implements Runnable{
Thread thread;
Public mator() { thread = new Thread(this);}
Public void start( ) {thread.start();}
Public void stop() {thread.stop();}
{…}
}
}
```

通过继承类 Thread 实现多线程很简单，只要继承类 Thread 重载方法 run()，然后为新类的每个实例调用 start()即可。当此类的方法 run()返回或 stop()被调用时，线程消亡。下面的例子通过扩充 Thread 类，并用该线程自己的实现覆盖 Thread.run()，产生一个新类 mxr.run()方法是 mxr 类线程所作的全部操作。

```
import java.mxr.*;
public class mxr extends Thread{
public void run()
{…}
}
```

2. 线程的启动和停止以及挂起和恢复

线程一旦创建之后，调用 start()方法启动该线程，即调用其目标的 run()方法，然后立即自调用 start()返回，与此同时，该线程开始执行。

线程在运行过程中，会遇到调用 sleep()方法，它将正在运行的线程挂起一定的时间。当然还可以调用 stop()方法，终止一个线程的执行。线程一经 stop()方法被终止执行，它就不能用 start()方法重新启动．若是用 sleep()方法暂停一个线程的执行，那么这一线程就将睡眠一段特定的时间后开始执行。这一方式不适合需要在发生某一特定事件时启动一线程的情况。这时可以调用 suspend()方法，允许一线程暂时停止执行，调用 suspend()方法将允许被挂起的线程重新起动。下述例子是建立线程的典型例子，说明了 start()，suspend()，sleep()等方法的使用。

```
// TM.java
public class mxr extends java.applet.Apllet{
```

```
MyThread thread;
public void init(){
thread = new MyThread();
thread.start();
}
public void start(){thread.resume();}
public void stop(){thread.suspend();}
}
class MyThread extends Thread{
public void run(){
// 此线程要执行的代码
}
}
```

3.线程的调度

Java 专门设置一线程调度者,监控所有程序的全部线程,并确定线程的执行顺序。调度者根据线程的两个属性来决定一线程是否执行,即线程的优先级和精灵标志。调度者根据全部线程的优先级确定一个执行线程,优先级高的线程在优先级低的线程之间执行,若只有精灵线程在执行,那么 Java 虚拟机将退出。

调度者可采用两种调度方式:占先式和非占先式。在占先式中,采用时间片机制,当一线程执行一特定时间片后便被挂起。经过一特定的时间后,调度者将恢复一被挂起的线程。在非占先方式中,一旦一线程被调度执行,该线程将被执行下去,直至结束。线程可在任意长时间内掌握系统控制权。Yield()方法可迫使调度者执行某一等待线程。

用 Getpriority()方法可获得一线程当前优先级的值。优先级的范围为 1 - 10,默认值设为 Thread.NORM - PRIORITY,其值为 5。一般情况下,严格依赖优先级不是一种好的策略。优先级只能作为对系统的暗示,告知系统哪些线程应优先于其他线程获得处理器。

精灵线程有时被称为服务线程,通常以低优先级运行,提供一些基本服务。一个在不停运行的精灵线程的例子是垃圾收集线程。

四、线程的应用实例

下面的例子就是一个产生三个线程的多线程程序,每个线程输出四行信息.

```
// Mxrprog.java
class MyFirstMultiThredProg{
public static void main(String args[]){
int i;
MyFirstMultiThreadClass [ ] aMtcArray = new MyFirstThreadClass[3];
System.out.println("第一个线程");
for (i=0;i<3;i++)
aMTCArray[i] = new MyFirstMultiThreadClass[i];
for (i=0;i<3;i++)
aCMTArray[i].start();
WhileLoop:
while(true)
for (i=0;i<3;i++)
if (aMTCArray[i].isAlive()) continue WhileLoop;
break;
}
System.out.println("\t main() program! Bye bye...");
//继承类 Thread
class MyFirstMultiThreadClass extends Thread {
private int MySerialNum;
MyFirstMultiThreadClass(int SerialNum){
super();
MySerialNum = SerialNum;
}
//线程开始执行
public void run(){
int i;
for (i=0;i<4;i++)
System.out.println("线程" + MySerialNum);
System.out.println("线程" + MySerialNum + "再见");
}
}
```

五、多线程的问题

利用 Java 中的多线程方式来多任务地执行程序有许多好处,而且现行的操作系统也都提供充足的支持给

编程人员，但是多线程的程序仍不多见，最主要的原因就是多任务执行的程序，对各个进程或线程都会共用的数据，必须加以严格的控制，以避免多个进程或线程同时修改相同的数据，造成数据前后不一致而导致错误。如果在管理数据的时候算法没有考虑清楚，还常会造成死锁(Deadlock)。

Java 语言为解决死锁定和资源和协调问题，提供了对数据同步化(synchronijation)支持，为了完成同步化工作，java 采用了监控器和状态变量的技巧，加以改变后设计出许多基本的原语。在语言上，若在一个类的方法前面加上 synchronized 保留字，一组标记了 synchronized 的方法在执行时不会有其他方法在同一时间一起执行。

1. 锁定对象数据

解决资源协调问题最简单的方法，是在任何时刻只允许一个线程来取用会产生问题的资源。在其他语言中，资源协调问题一般都是靠操作系统所提供的系统调试来实现这样的功能。由于 Java 提供了同步化(synchronized)关键字，可以用 synchronized 来锁定某个对象，这样就可以保证同一时刻只有一个线程可以取到这个资源。锁定方式如下：

```
synchronized(锁定的对象){
// Synchronzed 程序块
...
}
```

用 synchronized 锁定的是对象，如果使用的不是同一个对象的话，还是有可能有好几个线程在同一段程序代码中执行(不会造成资源协调问题)。因为 Synchronized 的锁定是基于对象的，而不是锁定某段程序代码。

当一同步方法正被某线程执行时，Java 会自动地阻止一线程激活另一同步方法。当一同步方法运行结束，正在等待的线程才可运行所执行的方法。任一时刻只允许一个同步方法进入，当一同步方法运行结束时，监控者解锁，另一同步方法才有机会开始执行。

2. 锁定类数据

用 Synchronized 关键字来协调对象实体中的数据时不会造成任何问题。然而类数据并不属于任何一个对象实体。类数据只有一份，只要是该类产生的对象实体，都可以使用。如果仅仅锁定了某个对象实体，其他线程仍然可能使用另外的同类的对象实体来使用同样的数据，同样会造成问题。但 Java 提供了类似的锁定方式：

```
Synchronized(new 欲锁定的类().getclass()){
// 代码
...
}
```

这样就可以安全地使用类数据了。当然，上面的程序会先产生一个欲锁定类的实体对象，然后调用方法 getclass()以取得其类对象，再加以锁定，而刚才产生的那个实体对象就等待着垃圾回收和线程来处理了。

六、结束语

多线程编程有一定的挑战性。不正确的同步策略所导致的问题是与时间相关的，所以这些问题不易察觉，一旦发生又很难加以诊断。线程程序设计的大多数错误可划分成两类：

·一个对象的方法改变一个对象的状态，多个线程执行这一方法。

·两个或更多的同步方法彼此依赖，令导致死锁。

编写多线程程序，避免死锁问题，可以遵循如下的一些简单原则：

·用线程完成的任务应是一个可和其他任务并行执行的任务。这意味着每一线程要有一定义得很好的任务，很容易确定代码中应同步的位置。

·当在一个对象的同步方法中激活另一同步方法时，要注意死锁的问题。在多个线程独立的执行这些方法时，可能会导致死锁。

·上锁的时间要尽可能短。

支持线程，允许程序同时执行多个任务，这是 Java 语言非常重要的一个特性。Java 语言提供的线程技术，其有很强的灵活性，既增强了程序的功能，又提高了程序的性能。

(来稿时间：1997 年 1 月)