基于 Java 多线程的并发机制的研究和实现

王金海，江务学
（天津工业大学 信息与通讯工程学院，河北 天津 300160）

摘 要：针对高可靠性、高质量的 Java 并行多任务程序设计，分析了 Java 多线程机制的原理及其实现技术，研究了程序并发过程中的同步机制和交互通信机制，比较了基于操作系统级和基于 Java 多线程级并发机制的实现结构，总结了并发程序中死锁预防的一些编程规则和策略。所构造的一个具有完全意义上的并发同步的框架实例有一定的实用价值。

关键词：Java；多线程；并发机制；同步；死锁
中图分类号：TP311.52 文献标识码：A 文章编号：1005－3751(2004)03－0034－04

Study and Implementation of Concurrent Multitask Based on Java Multithread

WANG Jin-hai, JIANG Wu-xue
(School of Information and Communication, Tianjin Polytechnic University, Tianjin 300160, China)

Abstract: For the development of high reliable and qualitative Java concurrent programs, this paper analyses Java multithread mechanism and its realization, studies the concurrent mechanism based on Java synchronization and interactive communication mechanism, compares the concurrent structure based on opearting system and based on Java multithread, sums up some concurrent programming rules and strategies to prevent deadlock. A frame instance based on entire synchronization is presented, which has come practical values.

Key words: Java; multithread; concurrent mechanism; synchronization; deadlock

随着采用了内核级多线程结构的现代操作系统和处理机的体系结构的不断推进，并行多任务程序设计技术也迅猛发展，并在现实应用中占有举足轻重的地位。当前，支持并发多任务处理的有 C＋＋，Delphi，Java 等语言开发工具。Java 语言和 Java 虚拟机提供了完全意义上的多线程机制，其内置语言级的多线程机制可以方便地实现多个并行程序的开发。Java 多线程机制实现了多任务在宏观上同时执行，提供了并发过程中实现临界资源保护的同步机制和通信机制，还提供了由于同步可能造成的死锁的预防措施。笔者以 Windows 2000 为平台，以 JDK1.3 为开发环境，对上述问题进行分析、研究及实现。

1 进程与线程

1.1 概念比较

进程是一段静态代码即程序在处理机上的一次动态的运行过程，是一个具有产生、发展和消亡的过程。线程是比进程更小的执行单位，一个进程有一个或多个线程，线程也是一个具有创建、存在和消亡的动态过程。

收稿日期：2003－08－22
作者简介：王金海(1966－)，男，江西南昌人，副教授，硕士生导师，博士，研究方向为精密测量仪器与智能化、高级系统软件技术、计算机通讯。

1.2 职能比较

线程是进程的重要组成部分。传统的进程身兼两职：作为资源的分配和 CPU 的调度的基本单位，为了更好地实现开发程序的并发性，让进程摆脱繁重的任务，将其职能分离出来，让称为线程的实体分担，这也就产生了线程。进程是资源分配的基本单位，线程是 CPU 调度的基本单位。进程申请并获取所需资源，其对应的一些线程便在这些资源内活动并利用之[1]。

1.3 系统开销比较

因为进程和线程在上述职能的区别，所以线程之间转换及其间的切换都比进程做等量工作所需要的系统开销小，而且更容易实现多线程之间的同步和通信[2]。

2 线程状态及控制方法

Java 中的线程具有动态的生命周期，一个完整的周期经历五种状态，其间的转换及相应的线程控制方法如图 1 所示。

(1)新生态(Newborn)：创建了线程类子类的实例并初始化后，该对象就处于新生态，此时有了相应的存储空间和相应进程的资源。

(2)就绪态(Runnable)：处于新生态的线程被启动后，就处于就绪态，即进入就绪队列等待 CPU 时间片的到来。

此时已具备了运行的条件。

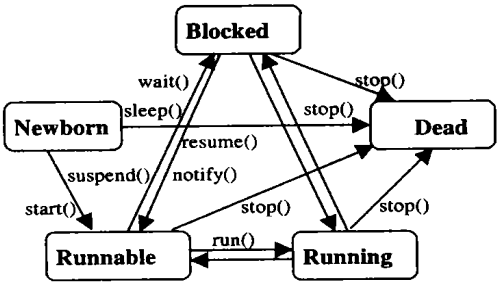

图 1　线程状态转换及控制方法图

(3)运行态(Running): 处于就绪态的线程获得 CPU 时间片即处于运行态, 此时执行对应线程的 run()方法的代码来完成相应的操作。

(4)阻塞态(Blocked): 正在运行或处于就绪态的线程由于某种原因, 让出 CPU 并暂时终止自己的执行, 即进入阻塞态, 只有当被阻塞的原因被解除时方可转入就绪或运行态。

(5)死亡态(Dead): 一个线程完成了其所有操作或被提前强行终止时即死亡, 此时线程不能再被恢复和执行。

3　基于 **Thread** 子类和 **Runnable** 接口的线程创建

在大型实用软件的开发过程中, 通常需要多个任务在宏观上同时执行, 同时也需要充分利用系统资源和提高程序的执行效率, 多线程能提供完好的解决方案。Java 是一种完全意义上的面向对象的程序设计语言, 提供了丰富的类库(包)和应用接口, 来方便用户开发利用。Java 中创建线程有两种方法。

3.1　继承 **Thread** 类

Java.lang.Thread 是系统提供的用来表示进程的类, Thread 类中的许多方法提供了比较完整的多线程处理的功能。可以用 Thread 来定义子类, 构造用户的线程。例如:

```
public MyThread extends Thread
{
   …//相关属性和方法定义
   public void run()
   {
   …//线程体代码
   }
   public static void main(String args[])
   { Thread t=new MyThread();
     …
     t.start();//启动线程 t
     …
   }
}
```

其中 run()方法至关重要, 也是整个线程的核心, 其代码是线程所要执行的内容, start()方法的作用是调用 run()方法, 启动线程。

3.2　实现 **Runnable** 接口

Java 中提供了一个实现多线程的接口 Runnable, 该接口只有一个方法 run(), 用户通过重载该方法以实现线程的相关操作, 有了 Runnable 接口就可被系统自动识别并执行。例如:

```
public class MyThread implements Runnable
{
   …//相关属性定义
   public void run()
   {
   …//线程所要执行操作的代码
   }
}
```

启动的方法有两种, 但其实质还是一致的, 如下所示:

```
MyThread my= new MyThread();
Thread t= new Thread(my);
t.start();
```

或

```
MyThread my= new MyThread();
new Thread(my).start;
```

3.3　两种线程创建方法的比较

第一种方法层次清晰, 逻辑分明, 使用简便, 实质上也实现了 Runnable 接口; 第二种方法可以弥补第一种方法的不足, 即可以实现多重继承(Java 中的类只能单重继承), 例如在小程序 Applet 中的应用。Runnable 接口还要以 Thread 类为框架来实现。

4　**Java** 并发机制的分析和研究

4.1　基于 **synchronized** 和 **monitor** 的互斥机制

并发性的引入, 在多道程序设计中, 为了达到某种实际目的, 程序中总会存在着一些不能"同时"(实质是宏观上的同时)被两个以上的线程访问的程序段, 即临界资源。Java 中利用 synchronized 锁定标志来定义临界资源, 这里的临界资源可以是方法, 也可以是代码段。在程序中有 synchronized 标志的方法或代码段在任一时刻只能有一个线程可以进入访问, 即实现了临界资源的互斥访问。在实现原理上有引入了类似于信号量机制的管程(Monitor), Java 为每一个对象都分配一个管程, 其作用是负责管理线程对临界资源的访问。一个线程访问某临界资源, 就获得了该临界资源所属对象的管程, 并且管程"上锁", 其它想访问该临界资源的线程必须等待至管程"开锁"(见图 2)。

4.2　基于 **wait**()和 **notify**()的交互通信机制

synchronized 解决了临界资源的互斥访问问题, 在现实应用中, 还会要求线程之间进行协同合作工作, 即在对临界资源的互斥访问的同时, 线程之间还要相互通信和互通消息。如经典的生产者－消费者问题、写者－读者问题、网络传输中的发送端－接受端问题等。Java 为了解决

多线程之间的交互问题，提供了 3 个标准的 Object 类的方法：wait（），notify（）和 notifyA ll（）。wait（）作用是使当前运行的线程由运行态转为阻塞态，进入等待队列中的 wati（）集中，并且释放管程；notify（）的作用是唤醒 wait（）集中的队首线程（或按照一定的算法来唤醒某个线程），使线程有阻塞态转为可运行态；notifyAll（）的作用是将所有 wati（）集中的线程都唤醒。Wait（）和 notify（）的配对使用能很好地实现互斥中的通信机制，以切实解决上述列举的问题[3]。

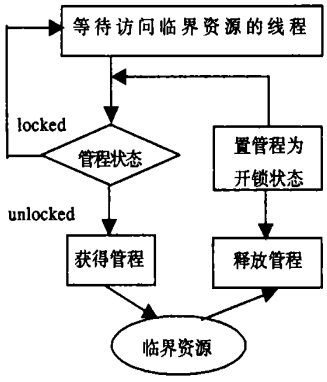

图 2 lockedunlocked 互斥机制实现原理图

4.3 基于互斥和交互通信机制的完全意义的同步

同步是指多个线程在一些关键点上的互相等待和互通消息。Java 的 synchronized 实现了线程的互斥机制，wait（）和 notify（）实现了线程间的互通消息的通信机制，分别解决了数据的一致性和多线程之间的协同合作的问题，实现了完全意义上的同步。图 3 将 Java 的同步机制和操作系统级的有信号灯和 P、V 原语实现的同步机制作一简单比较。

```
synchronized(this)          …
{…                          P(empty)
  wait();                   P(mutex)
  …                           CS;
notify();                   V(mutex)
…                           P(full)
}                             …
```

图 3 Java 同步机制与操作系统同步机制的比较图

5 基于完全同步的一个并发多任务的实用例程

5.1 实用例程

构造一个共享缓冲区类 SharedBuffer，其内有相关的数据属性，一个 send（）发送方法，一个 receive（）接收方法，均为同步方法，两个方法内通过 wait（）和 notify（）来实现其间的交互通信。另构造由 Thread 类继承而来的两个子类，分别是 Sender 和 Receiver，产生两个线程 t1 和 t2，实现具体的发送和接收的操作。例程如下：

```
public class FrameSample
{
  public static void main(String args[])
  {
    SharedBuffer buffer= new SharedBuffer();
    Sender t1= new Sender(buffer);
    Receiver t2= new Receiver(buffer);
    t1. start();
    t2. start();
  }
}// 主类 FrameSample，buffer 为共享对象实例
class Sender extends Thread
{ SharedBuffer theBuffer;
  public Sender(SharedBuffer s)
  { theBuffer= s;}
  public void run()
  { char c;
    for(int i=0; i<5; i++)
    { c=(char)(Math. random() *26+'A');
      theBuffer. send(c);
      System. out. println("Sender: "+c);
    }
  }
}// 发送者类
class Receiver extends Thread
{ SharedBuffer theBuffer;
  public Receiver(SharedBuffer s)
  { theBuffer= s;}
  public void run()
  { char c;
    for(int i=0; i<5; i++)
    { c=theBuffer. receive();
      System. out. println("Receiver: "+c);
    }
  }
}// 接收者类
class SharedBuffer
{ private int index=0;
  private char buf[]= new char[5];
  public synchronized void send(char c)
  { while(index==buf. length)
    { try{this. wait();}
      catch(InterruptedException e){}
    }
    this. notify(); buf[index]=c; index++;
  }
  public synchronized char receive()
  { while(index==0)
    { try{this. wait();}
      catch(InterruptedException e){}
    }
    this. notify(); index--; return buf[index];
  }
}// 共享缓冲区类
```

5.2 几点说明

（1）程序实例具有完全意义的并发性。对共享缓冲区互斥访问，能否向缓冲区发送数据和从缓冲区取数据是通

过 wait()和 notify()来互通消息的。完整的例程代码基于 Windows 2000, JDK1.3 环境通过; 例中若不使用 synchronized 来实现互斥, 将会出现数据的丢失和数据重复错误, 若不使用 wati()和 notify()就会出现异常和使 CPU 的效率明显降低[3, 5]。

(2)对临界资源的深入理解。现实世界中, 临界资源是指某些共享设备、共享数据段和数据结构等, 例程中对应的是 buffer。而计算机世界是对现实世界的一种抽象和封装, 由具体的机制决定的, 对应的临界资源就是程序的代码段, 例程中对应的是 send()和 receive()方法。实质上, 两世界是互通一致的, 之所以 buffer 是临界资源, 是通过 send()和 receive()的操作表现出来并实现的。

6　死锁预防

线程的同步机制中, 可能会出现这样一种现象, 几个线程各自拥有所需的部分资源, 而又同时需要别的线程所拥有的部分资源, 并等待着别的线程释放, 否则都不能向前推进, 这样一种由于资源竞争和分配而造成的一种僵持的现象即为死锁。Java 的并发机制是基于 JVM 级的, 类似于操作系统的同步机制, 死锁具有很大的隐藏性, 用户使用时, 可能很容易导致死锁故障, 而 Java 又不能避免死锁和恢复死锁, 只能在使用时对死锁进行预防[4, 5]。下面提供一些方法和策略, 可以用来预防死锁。

(1)线程的任务划分要明确、合理。多线程的并发是死锁产生的根源, 必须结合实际需要和 Java 多线程机制的特性合理安排各个线程的任务; 保证资源的均衡性; 尽量减少线程数。

(2)正确使用同步机制。synchronized 的使用是产生用户级死锁的直接原因, 慎重考虑同步方法之间的调用; 尽量避免同步方法之间的嵌套调用。

(3)正确使用通信机制。wait()和 notify()方法能解决一定的死锁问题, 通常 wait()和 notify()配对使用, 适当的时候考虑是否要用 notifyAll()方法。

(4)临界资源要精练。临界资源是产生死锁的物质基础, 尽量使临界资源精练, 以减少资源竞争而产生的冲突。

7　结束语

多线程技术功能强大, 为用户编程提供了极大的现实问题的解决策略。文中从原理和应用的角度探讨了 Java 多线程机制的实现技术, 从操作系统级分析了多线程技术中的同步通信问题, 总结了一些死锁预防的方法和策略, 并构造了一个完全同步的并行多任务的框架实例。在实际软件开发过程中, 可以快速开发出高可靠性高质量的并行多任务的程序, 具有一定的实用价值。

参考文献:

[1] 王海英. 一种改进的多线程模型[J]. 东北林业大学学报, 2002, 30(1): 95－97.

[2] 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统[M]. 西安: 西安电子科技大学出版社, 2000.

[3] 徐绍忠, 王　秉, 刘小虎. Java 并发机制研究[J]. 计算机工程, 2002, 28(4): 73－75.

[4] Horstmann C S, Cornell G. Java2 核心技术[M]. 朱　志, 王怀等译. 北京: 机械工业出版社, 2001.

[5] Deitel H M, Deitel P J. Java How to Program(4th Edition)[M]. 北京: 电子工业出版社, 2002.

(上接第 33 页)

生成动态连接库的方法来进行 VC 与 ML 之间的数据交互, 将两种开发环境便捷地来, 充分利用它们的优势来满足工程中的实际需要, 这对子解决现实工程问题有一定的意义。

表 1　典型节点数据

时刻 / 节点	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	12.86	34.41	50.13	61.74	70.44	76.99	81.90	85.58	88.35	90.44	92.01	93.21
3	100	100	100	100	100	100	100	100	100	100	100	100	100
11	0	8.81	28.39	45.03	57.77	67.38	74.61	80.04	84.12	87.18	89.48	91.23	92.55
21	0	46.21	62.95	72.40	79.11	84.10	87.85	90.67	92.78	94.36	95.55	96.46	97.14
29	0	18.55	37.60	52.48	63.77	72.26	78.65	83.45	87.06	89.76	91.80	93.34	94.51
42	0	32.18	51.62	63.72	72.45	78.97	83.87	87.54	90.30	92.37	93.93	95.11	96.01
51	0	77.60	87.30	90.86	93.17	94.87	96.14	97.09	97.81	98.34	98.75	99.05	99.28
61	0	38.23	56.43	67.38	75.26	81.13	85.55	88.87	91.35	93.22	94.63	95.69	96.50
71	0	27.59	47.59	60.57	70.02	77.09	82.39	86.38	89.37	91.61	93.30	94.58	95.55
101	0	6.95	25.45	42.63	56.00	66.12	73.73	79.45	83.75	86.97	89.40	91.24	92.63

参考文献:

[1] 王　晖. 精通 Visual C++ 6.0[M]. 北京: 电子工业出版社, 1999.

[2] 张智星. Matlab 程序设计与应用[M]. 北京: 清华大学出版社, 2002.

[3] 杨世铭. 传热学[M]. 北京: 高等教育出版社, 1987.