面向对象程序设计Java

江春华

电子科技大学信息与软件工程学院

内　容

第8章 多线程设计

1. 多线程机制
2. 多线程实现方法
3. 多线程状态及调度
4. 线程同步
5. 线程通信

多线程机制

- 线程就是应用程序中的一个可执行线索，多线程就是同一个应用程序中有多个可执行线索，它们可以并发执行。
- 多线程就是同一程序中多个任务的并发实现。
- 同类的多个线程是共享一块内存空间和一组系统资源，而线程本身的数据通常只有微处理器的寄存器数据，以及一个供程序执行时使用的堆栈。

多线程机制

- 线程与进程相似，是一段完成某个特定功能的代码，是程序中单个顺序的流控制。
- 系统在产生一个线程，或者在各个线程之间切换时，负担要比进程小的多。
- 一个进程中可包含多个线程，线程被称为轻负荷进程(light-weight process)。

多线程机制

- 进程与线程之间的差别主要体现在两个方面：

  - 作为基本的执行单元，线程的划分比进程小，因此，支持多线程的系统要比只支持多进程的系统并发程度高。

  - 进程把内存空间作为自己的资源之一，每个进程均有自己的内存单元。

    线程却共享内存单元，通过共享的内存空间来交换信息，从而有利于提高执行效率。

多线程机制

❖线程由三部分组成：

- 虚拟的CPU，封装在Java.lang.Thread类中。
- CPU所执行的代码，传递给Thread类。
- CPU所处理的数据，传递给Thread类。

- 建立Thread对象时，必须提供执行代码和代码所处理的数据。
- Java对象模型要求程序代码只能写成类的成员方法。
- 数据只能作为方法中的变量或类的成员存在。
- 规则要求为线程提供的代码和数据以类的实例形式出现。

多线程机制

- 一个建立线程的例子：

```
public class SimpleRunnable implements Runnable{
   private String message;
   public static void main(String args[]){
      SimpleRunnable r = new SimpleRunnable("Hello");
      Thread t1 = new Thread(r);
      t1.start();
   }
   public SimpleRunnable(String message){
      this.message = message;
   }
   public void run(){
      System.out.println(message);
   }
}
```

线程开始执行时，它在**public void run()**方法中执行。该方法是定义的线程执行起点，像应用程序从**main()**开始

首先main()方法构造**SimpleRunnable**类的实例。实例有自己的一个String数据，初始化为**"Hello"**。由实例t1传入**Thread**类构造器，这是线程运行时处理的数据。
执行的代码是实例方法**run()**。

多线程实现方法

- 创建新线程有两种方法：
  - 生成Thread子类。
  - 生成一个类，声明实现Runnable接口。

实现Thread子类方法的多线程

- 用这种方法生成新线程，可以按以下步骤进行：

1. 生成**Thread**类的子类。

```
class MyThread extends Thread
```

2. 在子类中覆盖**run()**方法。

```
public void run()
```

3. 生成子类的对象，并且调用start()方法启动新线程。

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

start()方法将调用**run()**方法执行线程。

实现Thread子类方法的多线程示例

```
class FirstThread extends Thread {
    public void run() {
      try{
          System.out.println("First thread starts running.");
         for(int i=0; i<6; i++) {
             System.out.println("First " + i);
            sleep(1000);
         }
         System.out.println("First thread finishes running.");
      } catch (InterruptedException e) {}
    }
}
```

实现Thread子类方法的多线程示例

```
class SecondThread extends Thread {
   public void run() {
      try{
         System.out.println("\tSecond thread starts running.");
         for(int i=0; i<6; i++) {
            System.out.println("\tSecond " + i);
            sleep(1000);
         }
         System.out.println("\tSecond thread finishes running");
       }catch (InterruptedException e) {}
    }
}
```

实现Thread子类方法的多线程示例

```
public class ThreadTest1 {
}
```

结果为:

```
First thread starts running.
First 0
        Second thread starts running.
        Second 0
First 1
        Second 1
First 2
        Second 2
First 3
        Second 3
First 4
        Second 4
First 5
        Second 5
First thread finishes running.
  Second thread finished.
```

实现Runnable接口方法的多线程

- 使用这种方法创建新线程，要完成以下几步：
  - 程序中某个类声明实现Runnable接口，并且在这个类中实现run()方法。
  - 生成这个类的对象。
  - 用Thread(Runnable target)构造器生成Thread对象，其中target是声明实现了Runnable接口的对象，并且用start()方法启动线程。

实现Runnable接口多线程示例

```
class FirstThread implements Runnable {
    public void run() {
      try {
        System.out.println("First thread starts running.");
        for(int i=0; i<6; i++) {
            System.out.println("First " + i);
           Thread.sleep(1000);
        }
        System.out.println("First thread finishes running.");
      } catch (InterruptedException e) {}
   }
}
```

实现Runnable接口多线程示例

```
class SecondThread implements Runnable {
    public void run() {
      try {
        System.out.println("\tSecond thread starts running.");
        for(int i=0; i<6; i++) {
            System.out.println("\tSecond " + i);
            Thread.sleep(1000);
        }
        System.out.println("\tSecond thread finished.");
      }catch(InterruptedException e) {}
   }
}
```

实现Runnable接口多线程示例

结果为:

```
First thread starts running.
First 0
        Second thread starts running.
        Second 0
First 1
        Second 1
First 2
        Second 2
First 3
        Second 3
First 4
        Second 4
First 5
        Second 5
First thread finishes running.
        Second thread finished.
```

多线程状态及调度

- 线程的状态：新生态、可执行态、阻塞态、停止态。
- 一个线程被创建以后，它就有了生命期，在生命期内，可以用来完成一项任务。线程在创建后到销毁之前总处于这四种态之一。
- 新生态：线程生成之后立即进入这个状态。线程对象已被分配内存空间，其私有数据已被初始化，但该线程还未被调度，可用`start()`方法调度。新生线程一旦被调度，就将切换到可执行状态。

多线程状态及调度

- 可执行态：处于可执行环境中，随时可以被调度而执行。它可细分为两个子状态：
  - 执行状态，已获得CPU，正在执行；
  - 就绪状态，只等待处理器资源。这两个子状态的过渡由执行调度器来控制。
- 阻塞态：由某种原因引起线程暂停执行的状态。
- 停止态：线程执行完毕或另一线程调用stop()方法使其停止时，进入这种停止状态，它表示线程已退出执行状态，并且不再进入可执行状态。

多线程状态及调度

- 应用程序中的多个线程能够并发执行，即线程数在多于处理机数时是串行地执行，那么如何来决定哪一个线程先执行？
- Java引入了优先级的概念，优先级就是线程获得CPU而执行的优先程度，优先级越高，获得CPU的权力越大，执行的机会越多，执行的时间也越长。
- Java把优先级划分为10级，用1至10的整数表示，数值越大，优先级越高。

线程的控制

- 线程的状态转换关系图。

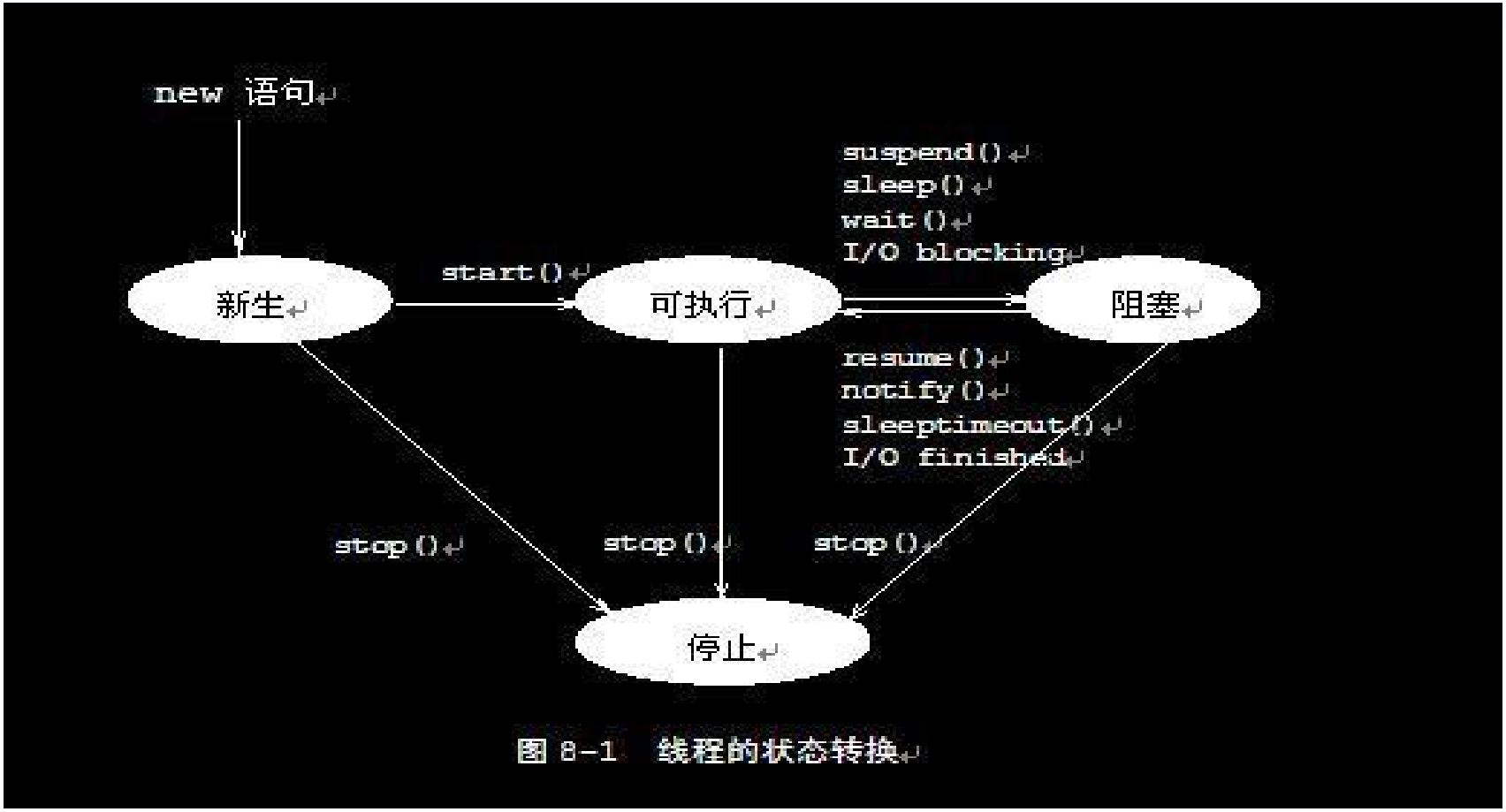


图 8-1 线程的状态转换

多线程状态及调度

- 在Thread类中定义了三个优先级常量：MIN_PRIORITY，MAX_PRIORITY和NORM_PRIORITY，其值分别为1，10，5。
- 如果应用程序没有为线程分配优先级，则Java系统为其赋值为NORM_PRIORITY。
- 可以通过Thread类的setPriority(int a)方法来修改系统自动设置的线程优先级。

多线程状态及调度

- 调度就是分配CPU资源，确定线程的执行顺序。
- Java采用抢占式调度方式，即高优先级线程具有剥夺低优先级线程执行的权力。
- 如果一个低优先线程正在执行，这时出现一个高优先级线程，那么低优先级线程就只能停止执行，放弃CPU，推回到等待队列中，等待下一轮执行，而让高优先级线程立即执行。

多线程状态及调度

- 如果线程具有相同的优先级，则按“先来先服务”的原则调度。
- 让高优先级线程执行一段时间后，能够交出使用权，放弃CPU。有两个方法可以达到这一目的：
  - 调用**sleep()**方法，暂时进入睡眠状态，从而让出CPU，使有相同优先级线程和低优先级线程有执行的机会。
  - 调用**yield()**而放弃CPU，这时和它有相同优先级的线程就有执行的机会。

多线程状态及调度

- Thread类定义控制线程执行的方法：
  - start()：用于调用run()方法使线程开始执行。
  - stop()：立即停止线程执行，其内部状态清零，放弃占用资源。
  - wait()：使线程处于等待状态。线程等待某个条件调用wait()方法。
  - notify()：使线程脱离阻塞状态。在条件变量所在的对象中调用notify()方法即可使线程脱离阻塞状态。
  - sleep()：调整线程执行时间，参数指定睡眠时间。
  - yield()：暂停调度线程并将其放在等待队列末尾，等待下一轮执行，使同优先级的其它线程有机会执行。

线程同步

- Java使用同步方法和同步状态来协调资源。
- 多线程提高了程序的并发度，但是有时候是不安全的或者不合逻辑的。则需要多线程同步。
- 线程同步是多线程编程的一个相当重要的技术。
- 多线程同步控制机制：保证同一时刻只有一个线程访问数据资源。

线程同步

- 同步锁：Java用锁标志(lock flag)的手段，对被访问的数据进行同步限制，从而实现对数据的保护。
- 把所有被保护资源都加上锁标志，线程必须取得锁标志才能访问被保护的资源。
- Java规定：被宣布为同步（使用**synchronized**关键字）的方法、对象或类数据，在任何一个时刻只能被一个线程使用。

线程同步

```
while (true) {
  synchronized(this) {
    t1.setText(Integer.toString(count1++));
    t2.setText(Integer.toString(count2++));
  }
}
```

- 在工
  资源
- **sy**
  它来
- 用**synchr**ed修饰的方法和代码段称为方法同步和代码段同步，它意味着同一时刻该方法或代码段只能被一个线程执行，其它想执行该方法或代码段的线程必须等待。
- 方法同步仅在该方法前加上**synchronized**修饰符即可。

线程同步

- 同步操作是以牺牲CPU资源为代价的。
- 正确使用同步可以减少线程间的相互干扰，提高程序的稳定性和可靠性。
- Java程序中多个线程可以通过消息来实现互动联系的，通常可以用notify()或notifyAll()方法唤醒其它一个或所有线程。
- 使用wait()方法来使该线程处于阻塞状态，等待其它的线程用notify()唤醒。

线程通信

- wait方法和notify方法是Java同步机制中重要的组成部分。
- 结合与synchronized关键字使用，可以建立很多优秀的同步模型。
- 同步分为类级别和对象级别，分别对应着类锁和对象锁。
- 如果static的方法被synchronized关键字修饰，则在这个方法被执行前必须获得类锁。对象锁类似。

wait和notify的应用示例

```
class TestThread extends Thread{
   private int time = 0 ;
   public TestThread(Runnable r,String name){
     super(r,name);
   }
   public int getTime(){ return time; }
   public int increaseTime (){ return ++time; }
}
public class DemoThread implements Runnable{
   public DemoThread() {
    TestThread t1 = new TestThread(this,"1");
    TestThread t2 = new TestThread(this,"2");
    t2.start();
    t1.start();
   }
   public static void main(String[] args) {
     new DemoThread();
   }
```

wait和notify的应用示例

```
public void run(){
   Test
   try{
      if
         
      }
      wh
         
         
         
         
      }
   }cat
   }
}
```

结果为:

```
@time in thread1=1
@time in thread1=2
**************************
@time in thread2=1
@time in thread2=2
**************************
@time in thread1=3
@time in thread1=4
**************************
@time in thread2=3
@time in thread2=4
**************************
```

思考问题

1. Java是如何实现多线程处理的？
2. Java多线程有哪几种状态，是什么样的调度方式？
3. 为何要有线程同步，它们是如何实现同步和通信的？

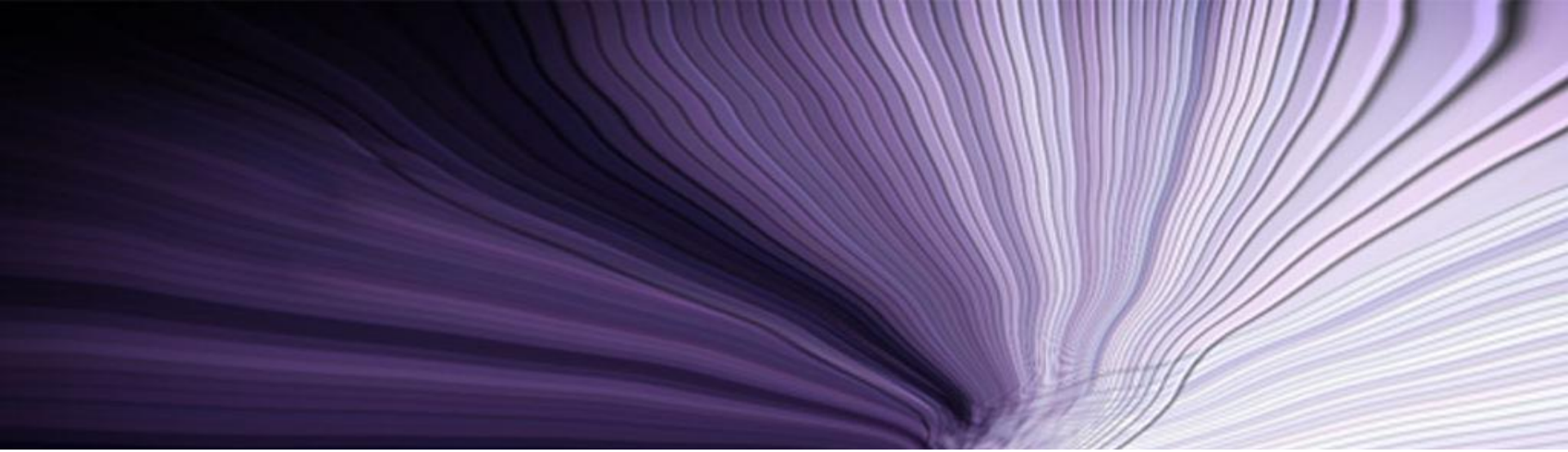
第8章作业

本章习题

习题1-6题

1-4题必做

5-6题选做



Q&A

电子科技大学信息与软件工程学院