第 10 章 多线程编程

教学目标：

本章介绍 Java 的多线程编程。多线程是 Java 语言的一个很重要的特点。本 章将首先介绍线程的概念以 及使用线程的优点，然后详 细介绍如何通过继 承 Thread 类或者实现 Runnable 接口实现 Java 的线程。针对多线程编程中的一些问 题例如线程调度、线程同步等问题我们也在本章中加以讨论。本章最后介绍了 线程组的的概念及应用。

教学重点与难点：

 线程的概念

 Thread 类和 Runnable 接口

 线程优先级

 生产者-消费者问题和线程同步

 死锁和避免死锁

 线程组

10.1 多线程概述

多线程是 Java 语言的特色之一。利用 Java 相关的 API，我们可以很容易的编写出一个 多线程的程序。多线程能够有效的提高程序的处理效率。本章从多线程的概念和多线程的 优点开始讲述。

10.1.1 什么是多线程

Java 程序通过流控制来执行程序流，程序中单个顺序的流控制称为线程，多线程则指 的是在单个程序中可以同时运行多个不同的线程执行不同的任务。

线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度， 区别在于线程没有独立的存储空间，而是和所属进程中的其它线程共享一个存储空间，这 使得线程间的通信远较进程简单。

多个线程的执行是并发的，也就是在逻辑上“同时”，而不管是否是物理上的“同时”。 如果系统只有一个 CPU，那么真正的“同时”是不可能的，但是由于 CPU 的速度非常快， 用户感觉不到其中的区别，因此我们也不用关心它，只需要设想各个线程是同时执行即可。

多线程和传统的单线程在程序设计上最大的区别在于，由于各个线程的控制流彼此独 立，使得各个线程之间的代码是乱序执行的，由此带来的线程调度，同步等问题，将在本 章中加以探讨。

10.1.2 为什么要使用多线程

在 Java 语言产生前，传统的程序设计语言的程序同一时刻只能单任务操作，效率非常 低，例如程序往往在接收数据输入时发生阻塞，只有等到程序获得数据后才能继续运行。 随着 Internet 的迅猛发展，这种状况越来越不能让人们忍受：如果网络接收数据阻塞，后 台程序就处于等待状态而不继续任何操作，而这种阻塞是经常会碰到的，此时 CPU 资源被 白白的闲置起来。如果在后台程序中能够同时处理多个任务，则能充分的利用 CPU 资源。 多线程是 Java 语言的一个很重要的特点。在一个 Java 程序中，我们可以同时并行运行多 个相对独立的线程，例如，我们如果创建一个线程来进行数据输入输出，而创建另一个线 程在后台进行其它的数据处理，如果输入输出线程在接收数据时阻塞，而处理数据的线程 仍然在运行。多线程程序设计大大提高了程序执行效率和处理能力。

10.2 建立 Java 线程

Java 提供了类 java.lang.Thread 来方便多线程编程，这个类提供了大量的方法来方便 我们控制自己的各个线程。除此之外，通过 Java 的计时器 Timer 能够实现简单的任务线程， 而 Runnable 接口在无法继承 Thread 类的情况下提供另外一种实现线程的方法。下面我们 将分别介绍它们。

10.2.1 计时器

java.util.Timer 和 java.util.TimerTask 可 以使 开发人 员很 容易实 现简 单的任 务。 在 J2SDK1.3 中引入这两个类之前，开发人员必须编写自己的调度程序，这需要花费很大精力 来处理线程和复杂的 Object.wait()方法。不过，Java 计时器没有足够的能力来满足许多应 用程序的计划要求。

下面的例子利用 Timer 在一段时间后启动一个线程。

首先我们定义一个类 RemindTask，这个类继承自 TimerTask 类，并且覆盖了 run()方法， 在 run()方法中我们编写我们要做的事情，在这个例子中即打印“Time’s up!”，然后调用 cancel()方法取消计时器。在 Reminder 类的构造函数中，我们创建一个 Timer 的对象，然 后调用 schedule()方法：

timer.schedule(new RemindTask(), seconds\*1000);

它的作用是在 seconds\*1000 毫秒后启动任务(new RemindTask())。因此，这个例子将在 运行 5 秒后打印出“Time’s up!”。

import java.util.Timer;

import java.util.TimerTask;

public class Reminder { Timer timer;

public Reminder(int seconds) {

timer = new Timer();

timer.schedule(new RemindTask(), seconds\*1000);

}

class RemindTask extends TimerTask {

public void run() { System.out.println("Time's up!"); timer.cancel();

}

}

public static void main(String args[]) { new Reminder(5); System.out.println("Task scheduled.");

}

}

下面这个程序将每 1 秒钟重复做一件事情，即打印警告信息“Alert!”，重复三次后打

印“Time’s up!”并且退出程序，注意 schedule()方法的调用参数：

timer.schedule(new RemindTask(), 0, 1\*1000);

它每隔 1\*1000 毫秒调用 RemindTask 线程的 run()方法。

import java.util.\*;

public class Alert { Timer timer; public Alert() {

timer = new Timer();

timer.schedule(new RemindTask(),

0,

1\*1000);

}

class RemindTask extends TimerTask {

int numWarning = 3;

public void run() {

if (numWarning > 0) { System.out.println("Alert!"); numWarning--;

} else {

System.out.println("Time's up!"); System.exit(0);

}

}

}

public static void main(String args[]){

new Alert();

}

}

程序的输出如下：

Alert! Alert! Alert! Time's up!

10.2.2 Thread 类

如果要做的不是一个简单的重复作业，则 Timer 就显得力不从心了。这时

java.lang.Thread 类将派上用场。下面介绍 Thread 类的使用。

**1.** 创建线程

任何顺序执行的程序都有开头、结尾以及一系列执行语句。线程也同这种程序类似， 所不同的是线程本身不能运行，它只能用于程序中。线程就是程序中单独顺序的流控制。

Java 提供 Thread 类来创建线程。下面是一个创建启动一个线程的语句：

Thread thread1=new Thread();

thread1.start();

从这个例子，我们可以通过 Thread()构造方法创建一个线程，并启动该线程。事实上， 启动线程将启动线程的 run()方法，而 Thread 类中的 run()方法没有任何操作语句，所以这 个线程没有任何操作。要使线程实现预定功能，必须定义自己的 run()方法。Java 中通常有 两种方式定义 run()方法：

通过定义一个 Thread 类的子类，在该子类中覆盖 run()方法。Thread 子类的实例对象

就是一个线程，显然，该线程有我们自己设计的线程体 run()方法，启动线程就启动了子类 中重写的 run()方法。

通过 Runnable 接口，在该接口中定义 run()方法的接口。所谓接口跟类非常类似，主 要用来实现特殊功能，如复杂关系的多重继承功能。在此，我们定义一个实现 Runnable() 接 口的类，在该类中定义自己的 run()方法，然后以该类的实例对象为参数调用 Thread 类的 构造方法来创建一个线程。

线程的所有活动都是通过线程 run()方法来实现的。在一个线程被建立并初始化以后，

Java 的运行时系统就自动调用 run()方法，正是通过 run()方法才使得建立线程的目的得以 实现。

下面的程序说明了如何通过继承 Thread 类实现线程。在例子中 myThread 类继承了 Thread 类并覆盖了 Thread 类的 run()方法，在 main()方法中我们生成 myThread 的实例并调 用 start()方法启动这个线程。完整的代码如下：

class myThread extends Thread {

public void run() {

for (int i = 0; i < 10; i++) { System.out.println("Hello" + i);

}

}

}

public class threadSample{

public static void main(String args[]){ myThread thread = new myThread(); thread.start();

}

}

这个程序的输出如下：

Hello0

Hello1

Hello2

Hello3

Hello4

Hello5

Hello6

Hello7

Hello8

Hello9

下面我们再实现一个稍微复杂一点的例子。这个例子中，myThread 类的 run()方法打 印一些信息并且进入睡眠，睡眠时间是随时的。在 main()方法中将创建两个 myThread 类 的对象并分别启动它们。代码如下：

class myThread extends Thread {

myThread(String str){

super(str);

}

public void run() {

for (int i = 0; i < 10; i++) { System.out.println(getName() + " : Hello" + i); try {

sleep((int)(Math.random() \* 1000));

} catch (InterruptedException e) {}

}

}

}

public class multiThreadSample{

public static void main(String args[]){

myThread thread1 = new myThread("Thread 1"); myThread thread2 = new myThread("Thread 2"); thread1.start();

thread2.start();

}

}

**2.** 线程命名

每个线程体都有一个名字，Thread 缺省的名称是一个短线连字符和一个零开始的数字 符号，如果不接受 Java 的缺省线程名称则可以选择使用自己的。为了能够自定义名称， Thread 提供带有 name 参数构造函数。在例子 multiThreadSample 中，我们对线程体的构造 如下：

myThread thread1 = new myThread("Thread 1");

myThread thread2 = new myThread("Thread 2");

我们为 myThread 的对象 thread1 和 thread2 分别命名为“Thread 1”和“Thread 2”，除 此之外，Thread 类还提供了方法 setName()用于设置一个线程的名字。Thread 也提供一个 getName()方法返回当前名称。例如在例子中：

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

线程命名最大的作用在于程序调试。

**3.** 线程状态

在上面的例子 multiThreadSample.java 中，首先我们创建一个 Thread 的对象：

myThread thread1 = new myThread("Thread 1");

在启动这个线程之前，这个线程处于创建状态，它仅仅是一个空的线程对象，系统不 为它分配资源。处于这种状态时只能启动或终止该线程，调用除这两种以外的其它方法都 会失败并且会引起非法状态处理。

之后我们进行调用：

thread1.start();

其中 start()方法产生了运行这个线程所需的系统资源，安排其运行，并调用线程体的 run()方法，这样就使得该线程处于可运行( Runnable )状态。需要注意的是这一状态并不是 运行中状态（Running )，因为线程也许实际上并未真正运行。由于很多计算机都是单处理 器的，所以要在同一时刻运行所有的处于可运行状态的线程是不可能的，Java 的运行系统 必须实现调度来保证这些线程共享处理器。但是在大多数情况下，可运行状态也就是运行 中，当一个线程正在运行时，它是可运行的，并且也是当前正运行的线程。

当下面 四种 情况发 生时 ，线程 就进 入不可 运行 状态： 调用 了 sleep() 方法；调 用了

suspend()方法；为等候一个条件变量，线程调用 wait()方法；输入输出流中发生线程阻塞。 例如，在 multiThreadSample.java 中，我们在 run()方法中进行了 sleep()调用：

try {

sleep((int)(Math.random() \* 1000));

} catch (InterruptedException e) {}

在这个 例子中我 们可以看 到通过调 用 sleep() 方法使 得 myThread 线 程休止 了 (int)(Math.random() \* 1000)毫秒，这时即使处理器空闲，也不能执行该线程。sleep()调用结 束以后，myThread 又成为可运行的了。

对于上面四种情况，都有特定的返回可运行状态的方法与之对应，如下所述： 如果线程处于睡眠状态中，sleep()方法中的参数为休息时间，当这个时间过去后，线

程即为可运行的；

如果线程在等待条件变量，那么要停止等待的话，需要该条件变量所在的对象调用

notify()或 notifyAll()方法；

如果在 I/O 流中发生线程阻塞，则特定的 I/O 指令将结束这种不可运行状态。 线程的终止一般可通过两种方法实现：自然撤消或是被停止。

自然撤消是指从线程的 run()方法正常退出，而调用 stop()方法也可以停止当前线程。 在 Thread 类的程序接口中提供了 isAlive()方法，如果线程已被启动并且未被终止，那

么 isAlive()返回 true。如果返回 false，则该线程是新创建或是已被终止，若返回 true，则 该线程是可运行或是不可运行的，但是我们不能通过 isAlive()方法的返回值判断这个线程 是可运行的还是处于不可运行状态。

**4. Join()**方法

有些情况下，当新创建一个线程并启动它之后，希望等待这个线程结束之后再作其他 事情。例如下面的程序，线程的 run()函数的功能是打印 5 行字符，而 main()函数希望在线 程打印完 5 行字符之后打印“Thread finished”的信息。程序代码如下：

class myThread extends Thread {

public void run() {

for (int i = 0; i < 5; i++) { System.out.println("Hello" + i);

}

}

}

public class threadSample2{

public static void main(String args[]){ myThread thread = new myThread(); thread.start(); System.out.println("Thread finished.");

}

}

但是运行这个程序的结果却和设想不一样，一个可能的输出是：

Thread finished. Hello0

Hello1

Hello2

Hello3

Hello4

为什么会这 样？因为程 序中的 main() 函数执 行 thread.start() 后马上 就接着执 行

System.out.println("Thread finished.")，而这时 thread 线程可能尚未执行完毕。 为了正确打印信息，我们可以通过不停的调用 isAlive()方法来判线程是否结束，只有

结束后才继续 main()函数的工作。例如，上面的程序可以改成以下这样：

class myThread extends Thread {

public void run() {

for (int i = 0; i < 5; i++) { System.out.println("Hello" + i);

}

}

}

public class threadSample2{

public static void main(String args[]){

myThread thread = new myThread();

thread.start();

while(thread.isAlive())

try{

Thread.sleep(10);

}catch (InterruptedException e){};

System.out.println("Thread finished.");

}

}

main()函数循环调用 isAlive()函数判断 thread 的状态，如果线程 thread 函数还没结束， 则主线程调用 Thread.sleep(10)休眠 10 毫秒。可以看到，这次打印的信息是正确的。

事实上，while 循环、isAlive()方法和 sleep()方法结合起来是很有用的，Java 语言提供 了 join()方法用于等待一个线程的停止。join()方法又三个形式：join()，join(long millis)和 join(long millis, int nanos)。后两个 join()方法其实上在等待线程停止的过程中加入了超时的 概念，如果超过参数所指定的时间线程还没退出，则 join()方法返回，调用 join()的线程继 续。因此，上面的程序又可以改成下面的样子：

class myThread extends Thread {

public void run() {

for (int i = 0; i < 5; i++) { System.out.println("Hello" + i);

}

}

}

public class threadSample3{

public static void main(String args[]){ myThread thread = new myThread(); thread.start();

try{

thread.join();

}catch (InterruptedException e){};

System.out.println("Thread finished.");

}

}

**5.** 访问线程

Thread 类提供了一个静态方法 currentThread()可以用于返回正在运行的线程体引用。 这个方法有时候是很重要的。例如下面这个程序例子中，myThread 线程体的 run()方法调 用 foo 类的 doSomething()方法打印一些信息，这些信息包括调用线程的名字，但是 foo 类 中并没有保存对线程题的引用，因此我们只能通过 Thread.currentThread().getName()得到调 用者的名字 。这个例子 只是简单的 说明了 currentThread()方法的必要 性，因此只 是调用 getName()得到当前运行线程的名字。事实上，在实际应用中，调用 currentThread()方法得 到当前线程后可用以其他很多目的。

i);

class foo{

static void doSomething(){

for (int i = 0; i < 10; i++) System.out.println(Thread.currentThread().getName() + " : Hello" +

}

}

class myThread extends Thread {

public void run() {

foo.doSomething();

}

}

public class ThreadAccess{

public static void main(String args[]){ myThread thread1 = new myThread(); myThread thread2 = new myThread(); thread1.start();

thread2.start();

}

}

Thread 类还提供了静态方法 activeCount()和 enumerate()分别用于得到当前同一线程组 中的活动线程数目和这些线程副本。请看如下例子：

class myThread extends Thread {

public void run() {

try{

sleep(5000);

}catch(Exception e){};

}

}

public class ThreadAccess2{

public static void main(String args[]){ myThread thread1 = new myThread(); myThread thread2 = new myThread(); thread1.start();

thread2.start();

int n = Thread.activeCount(); System.out.println("Active threads count: " + n);

Thread tarray[] = new Thread[Thread.activeCount()]; Thread.enumerate(tarray);

System.out.println("Active threads: ");

for(int i=0; i< tarray.length; i++) System.out.println(tarray[i].getName());

}

}

这个程序的输出是：

Active threads count: 3

Active threads:

main

Thread-0

Thread-1

需要注意的是，在一般情况下，当重申一个数组时不要依靠 activeCount()的返回值。 如 果这 样做， 程序 将冒掷 出一 个 NullPointerException 对 象的 风险 。为什 么呢 ？在调 用 activeCount() 和 enumerate(Thread [] thdarray) 之间， 一个或更多 线程可能结 束。结果 ， enumerate(Thread [] thdarray) 能够复 制少数线程 引用进它的 数组。因此 ，仅考虑 将 activeCount() 的返回值作为数组可能大小的最大值。同样，考虑 将 enumerate(Thread [] thdarray)的返回值作为在一个程序对那种方法调用时活跃线程的数目

10.2.3 Runnable 接口

前面介绍了通过继承 Thread 类来引入多线程。然而，有些情况下却不能够这么作。Java 语言不允许一个类继承多个类，如果某个类已经声明成继承其他类，那么它将不可以继承 Thread 类。幸运的是，Java 的设计者已经意识到不可能创建 Thread 子类的情形总会发生的。 这导致产生 java.lang.Runnable 接口和带 Runnable 参数的 Thread 构造器，如 Thread(Runnable target)。

Runnable 接口声明了唯一一个方法：void run()。这个方法和 Thread 的 run()方法一样

并作为线程的执行入口服务。因为 Runnable 是一个接口，任何类都能通过将一个 implements 子句包含进类头和提供一个适当的 run()方法实现接口。在执行时间，程序代码能从那个类 创建一个对象或 runnable 并将 runnable 的引用传递给一个适当的 Thread 构造器。构造器和 Thread 对象一起存贮这个引用并确保一个新线程在调用 Thread 对象的 start()方法后调用 runnable 的 run()方法。

abstract class foo {

abstract void printInfo();

}

i);

class MyRunnable extends foo implements Runnable {

void printInfo(){

for (int i=0; i<5; i++) { System.out.println(Thread.currentThread().getName() + " : Hello" +

try { Thread.sleep(500);

} catch (InterruptedException e) { e.printStackTrace(); }

}

}

public void run () {

printInfo();

}

}

public class RunnableDemo{

public static void main(String args[]){ MyRunnable thread1 = new MyRunnable(); MyRunnable thread2 = new MyRunnable(); new Thread(thread1).start();

new Thread(thread2).start();

}

}

RunnableDemo 由类 RunnableDemo，foo 和 MyRunnable 组成。类 RunnableDemo 通过

创建一个 MyRunnable 对象驱动应用程序。foo 和 MyRunnable 组成了一个基于 foo 层次的 类。foo 是抽象的因为它提供一个抽象的 printInfo()方法。MyRunnable 类扩展 foo。由于 MyRunnable 继承了 foo 类，因此它不能同时继承 Thread 类，因此只能通过让之实现 Runnable 接口实现多线程。在 RunnableDemo 类的 main()方法中，我们创建了两个 MyRunnable 的对 象 thread1 和 thread2，并且把它们绑定到 Thread 类的对象上并调用 Thread 的 start()方法启 动线程：

MyRunnable thread1 = new MyRunnable(); MyRunnable thread2 = new MyRunnable(); new Thread(thread1).start();

new Thread(thread2).start();

10.3 线程优先级

多线程程序设计涉及到的一个问题是线程的优先级。线程优先级的大小决定了线程获 取 CPU 时间的机会。Java 语言提供了设计线程优先级的方法。

10.3.1 线程优先级和线程调度

线程调度有两种方式：抢占式和非抢占式。抢占式系统在任何给定的时间内将运行最 高优先级的线程，系统中的所有线程都有自己的优先级。

线程的优先级（Priority）决定该线程的重要程序有多大，即线程运行的顺序以及从处 理器中获得的时间数量如何。

Java 线程的优先级可由用户来设置，线程的优先级是由一个整数表示，这个整数越大 则 线 程 的 优 先 级 越 大 。 Java 优 先 级 的 范 围 在 Thread.MIN\_PRIORITY 和 Thread.MAX\_PRIORITY 之间。缺省情况下线程的优先级是 Thread.NORM\_PRIORITY。 Thread 类提供了 setPriority()和 getPriority()方法来设置和读取优先级。

**Java 虚拟机是抢占式的，它能保证运行优先级最高的线程。在 Java 虚拟机中，如把**

**一个线程的优先级改为最高，那么他将取代当前正在运行的线程，除非这个线程结束运行 或者进入休眠状态，否则将一直占用所有的处理器的时间**。

下面是一个模仿赛跑的程序。Runner 类的 run()方法中对自身的私有成员 step 做自增 的操作，然后进入睡眠 1 毫秒。Race 类的 main()方法中创建了 Runner 类的两个对象 runner[0] 和 runner[1] ， 分 别 把 它 们 的 优 先 级 设 置 为 Thread.MIN\_PRIORITY 和 Thread.MAX\_PRIORITY，然后调用 start()方法启动它们。main()方法每隔 5 秒钟就查询一 下这两个对象的 step 值，然后分别打印这两个对象的 step 值。可以看到，这个赛跑程序是 不公平的，因为 runner[1]的优先级比较高，所以它获取 CPU 的机会就大，可以预料，在一 段时间后，runner[1]的 step 值将比 runner[0]的 step 值大。完成的程序代码如下：

class Runner extends Thread {

private int step = 0;

public void run() {

while(step < 50000){

step ++;

try{

Thread.sleep(1);

}catch(Exception e){};

}

}

public int getStep(){

return step;

}

}

public class Race{

public static void main(String args[]){ Runner runner[] = { new Runner(),

new Runner()};

runner[0].setPriority(Thread.MIN\_PRIORITY);

runner[1].setPriority(Thread.MAX\_PRIORITY);

runner[0].start();

runner[1].start();

while(runner[0].isAlive() && runner[1].isAlive()){

try{

Thread.sleep(5000);

}catch(Exception e){};

System.out.println("Runner 0: " + runner[0].getStep()); System.out.println("Runner 1: " + runner[1].getStep());

}

}

}

运行这个程序，可以看到输出如下：

|  |  |  |
| --- | --- | --- |
| Runner | 0: | 3664 |
| Runner | 1: | 4976 |
| Runner | 0: | 7601 |
| Runner | 1: | 9958 |
| Runner | 0: | 11541 |
| Runner | 1: | 14939 |
| Runner | 0: | 15457 |
| Runner | 1: | 19928 |
| Runner | 0: | 19376 |
| Runner | 1: | 24909 |
| Runner | 0: | 23290 |

Runner 1: 29899

如果遇到两个优先级相同的线程，调度器将以轮转调度（Round-Robin）算法选择一个 线程运行。这个被选择的线程将一直运行直至以下某个条件为真：

(1) 一个具有更高线程优先级的线程进入可运行状态。

(2) 这个线程结束运行。

(3) 在分时操作系统上，时间片到期。分时操作系统把 CPU 时间分成一个一个的时间 片，操作系统轮流地把每个时间片分给各个并发线程，每个线程一次只能运行一个时间片。 当时间片计数到时后，系统选择另一个线程并分给它时间片，让其投入运行，如此循环反 复。

当这些条件的某个为真时，另外一个线程将有机会得到 CPU 资源而运行。 下面的程序说明了这种情况，runner[0]和 runner[1]的优先级都设为 2，代码如下：

class Runner extends Thread {

private int step = 0;

public void run() {

while(step < 50000){

step ++;

try{

Thread.sleep(1);

}catch(Exception e){};

}

}

public int getStep(){

return step;

}

}

public class Race{

public static void main(String args[]){ Runner runner[] = { new Runner(),

new Runner()};

runner[0].setPriority(2);

runner[1].setPriority(2);

runner[0].start();

runner[1].start();

while(runner[0].isAlive() && runner[1].isAlive()){

try{

Thread.sleep(5000);

}catch(Exception e){};

System.out.println("Runner 0: " + runner[0].getStep()); System.out.println("Runner 1: " + runner[1].getStep());

}

}

}

程序的运行结果表明两个线程基本上是轮流执行的，这也表明程序所在的运行平台是 分时操作系统。下面是这个程序的运行结果：

|  |  |  |
| --- | --- | --- |
| Runner | 0: | 4029 |
| Runner | 1: | 4030 |
| Runner | 0: | 7989 |
| Runner | 1: | 7990 |
| Runner | 0: | 11399 |
| Runner | 1: | 11400 |
| Runner | 0: | 13492 |
| Runner | 1: | 13493 |
| Runner | 0: | 15503 |
| Runner | 1: | 15504 |
| Runner | 0: | 17045 |
| Runner | 1: | 17045 |
| Runner | 0: | 20908 |
| Runner | 1: | 20909 |

10.3.2 用户线程和守护线程

在 Java 中比较特殊的线程是被称为守护（Daemon）线程的低级别线程。这个线程具 有最低的优先级，用于为系统中的其它对象和线程提供服务。将一个用户线程设置为守护 线程的方式是在线程对象创建之前调用线程对象的 setDaemon()方法。典型的守护线程例子 是 JVM 中的系统资源自动回收线程，它始终在低级别的状态中运行，用于实时监控和管理 系统中的可回收资源。

只有当创建中的线程是 daemon 时，这个新的守护线程才会自动成为 daemon。从调度

的角度来说，守护线程与用户线程是类似的。Java 虚拟机仅仅只在用户线程退出后才检查 某个特定的线程是不是守护线程。如果 Java 虚拟机发现内存中只有守护线程而没有用户 线程时，它会自动退出。守护线程的存在只是为了服务于用户线程。垃圾回收线程是 Java 虚拟机的基准实现中的标准守护线程。只有在某一线程对象处于创建和启动之间时，函数 void setDaemon ()才能被调用。

当 Java 虚拟机启动时，通常只有一个非守护线程（在有些已作指定的类中，这一线

程常典型地调用名为 main 的函数）。Java 虚拟机继续执行线程，直到以下任何一种情况 发生：

（1）类 Runtime 的退出函数被调用，并且安全管理程序允许其退出。

（2）所有的非守护线程都因为各种原因而终止。这些原因包括：运行调用返回或是发 生越界的意外等。

10.4 线程同步

线程同步是多线程编程中很重要的技术。在实际应用中，几乎所有多线程都必须考虑 同步的问题。本小节通过对生产者-消费者问题的讨论，探讨线程同步的方法。最后还介绍 了饥饿和死锁的概念。

10.4.1 生产者-消费者问题

线程同步中一个经典问题是生产者-消费者问题。下面我们讨论这个问题在没有同步和 使用了同步以后出现的情况。

**1.** 问题描述

前面所提到的线程都是独立的，而且异步执行，也就是说每个线程都包含了运行时所 需要的数据或方法，而不需要外部的资源或方法，也不必关心其它线程的状态或行为。但 是经常有一些同时运行的线程需要共享数据，例如一个线程向文件写数据，而同时另一个 线程从同一文件中读取数据，因此就必须考虑其它线程的状态与行为，这时就需要实现同 步来得到预期结果。

我们把系统中使用某类资源的线程称为消费者，产生或释放同类资源的线程称为生产

者，下面我们来讨论关于线程的同步问题的一般模型，即生产者－消费者问题。 可以想象在一个 Java 的应用程序中，生产者线程向文件中写数据，消费者从文件中读

数据，这样，在这个程序中同时运行的两个线程共享同一个文件资源。

下面我们通过一个程序来模拟生产者 - 消费者问题。在这个例子中，生产者产生 从 “Book0”到“Book9”的 10 个字符串，代表 10 本书，将它们存储在一个 BookCase 的对 象中，表示把书放入书架中。然后调用 sleep()方法使生产者线程在一个随机产生的 0 到 100 秒的时间段内睡眠：

public class Producer extends Thread {

private BookCase bookcase;

private int number;

public Producer(BookCase b, int number) {

bookcase = b;

this.number = number;

}

public void run() {

for (int i = 0; i < 10; i++) { String book = "Book"+i; bookcase.put(book);

System.out.println("Producer " + this.number

+ " put: " + book);

try {

sleep((int)(Math.random() \* 100));

} catch (InterruptedException e) { }

}

}

}

消费者线程则不断地从 BookCase 对象中取这些书并打印所取到的书：

public class Consumer extends Thread {

private BookCase bookcase;

private int number;

public Consumer(BookCase b, int number) {

bookcase = b;

this.number = number;

}

public void run() { String book = "";

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

book = bookcase.get(); System.out.println("Consumer " + this.number

+ " got: " + book);

}

}

}

在这个程序中，我们通过一个 producerConsumerTest 类来驱动生产者和消费者。

public class ProducerConsumerTest {

public static void main(String[] args) { BookCase b = new BookCase (); Producer p1 = new Producer(b, 1); Consumer c1 = new Consumer(b, 1);

p1.start();

c1.start();

}

}

在这个例子中，生产者与消费者通过 BookCase 对象来共享数据。但是我们发现不论

是生产者线程还是消费者线程都无法保证生产者每放入一本书，消费者就能及时取得这本 书并且只取一次。下面将说明没有同步的情况下有可能发生的事情。

**2.** 没有同步的情况

在没有同步的情况下，如果生产者的速度比消费者快，那么生产者就会在消费者取走 某本书之前放入下一本书，因此消费者将漏掉这本书，如：

Consumer 1 got: Book3

Producer 1 put: Book4

Producer 1 put: Book5

Consumer 1 got: Book5

另外一个问题是：如果消费者比生产者快，那么消费者就有可能在生产者还没放入新 书时访问 CubbyHole 多次。这种情况下消费者将多次取到同一本书，例如：

Producer 1 put: Book4

Consumer 1 got: Book4

Consumer 1 got: Book4

Producer 1 put: Book5

Consumer 连续取到两次“Book4”之后 Producer 才放入“Book5”。 上面两种输出结果都不是我们所希望的那样：生产者每放入一本书，消费者就取这本

书。象这种异步执行的多线程，由于希望同时进入同一对象中而发生错误结果的情况称为 竞争条件(race condition)。

为了避免上述情况发生，就必须使生产者线程向 BookCase 中存储数据和消费者线程 从 BookCase 中取数据同步起来。为达到这一目的，必须采取两种措施：

不能让生产者和消费者同时访问 BookCase。在 Java 线程中，我们可以通过锁住一个 对象来做到这一点。档一个对象被某个线程锁定后，如果有第二个线程调用同步的方法方 位这个对象时，第二个线程就会阻塞，直到对象锁被释放。

生产者和消费者能够彼此之间进行协调。生产者在放入新书后要通知消费者，而消费 者在取出一本书后也要通知生产者。Thread 类提供了 wait()、notify()和 notifyAll()方法来完 成这些事情。wait()方法用于在某个条件上进行等待，而 notify()和 notifyAll()则用于在条件 改变之后通知其他线程。

**3.** 使用同步

以下的 BookCase 实现了同步，注意 put()方法和 get()方法。在下一小节中我们将详细 说明 get()和 put()方法实现同步的具体过程。

public class BookCase {

private String Book;

private boolean available = false;

public synchronized String get() {

while (available == false) {

try {

wait();

} catch (InterruptedException e) { }

}

available = false;

notifyAll();

return Book;

}

public synchronized void put(String newBook) {

while (available == true) {

try {

wait();

} catch (InterruptedException e) { }

}

Book = newBook; available = true; notifyAll();

}

}

运行 ProducerConsumerTest，可以看到，输出的结果是正确的。

Producer 1 put: Book0

Consumer 1 got: Book0

Consumer 1 got: Book1

Producer 1 put: Book1

Consumer 1 got: Book2

Producer 1 put: Book2

Consumer 1 got: Book3

Producer 1 put: Book3

Consumer 1 got: Book4

Producer 1 put: Book4

Consumer 1 got: Book5

Producer 1 put: Book5

Consumer 1 got: Book6

Producer 1 put: Book6

Consumer 1 got: Book7

Producer 1 put: Book7

Consumer 1 got: Book8

Producer 1 put: Book8

Consumer 1 got: Book9

Producer 1 put: Book9

10.4.2 对象锁

**我们把一次只允许一个线程使用的共享资源称为临界资源（Critical Resource）**，**而在 每个进程中访问临界资源的程序段称为临界区（Critical Section）。**

对象通过 synchronized 关键字来声明线程必须获得监视器才能进行对自己的访问。 在我们的生产者-消费者例子中，BookCase 的 put()方法和 get()方法是临界区。在生产

者放入书的同时消费者不能取书，而在消费者取书的同时生产者不能放书。因此 put()方法 和 get()方法必须用 synchronized 关键字标志起来。如下所示：

public class BookCase {

private String Book;

private boolean available = false;

public synchronized String get() {

}

public synchronized void put(String newBook) {

}

}

BookCase 类的 put() 方法和 get() 方法 都标志有 synchronized 关 键字， 则系统将 给

BookCase 类的每一个对象都赋予一个唯一的锁。无论什么时候，只要某个线程进入标有 synchronized 关键字的片断，这个对象就被该线程锁住。如果在此同时其他线程也准备调 用同一个对象标有 synchronized 的方法，它将被暂停直到当前线程离开 synchronized 代码 片断。

所以，生产者调用 BookCase 的 put()方法时，它就锁住 BookCase，以阻止消费者访问

它的 get()方法：

public synchronized void put(String newBook) {

//生产者获得锁

……

//生产者释放锁

}

put()方法返回时，生产者释放锁。

类似的，消费者调用 get()方法时，它就锁住 BookCase 以阻止生产者调用 put()方法：

public synchronized String get() {

//消费者获得锁

……

//消费者释放锁

}

锁的获得和释放都是由 Java 运行时环境自动完成的。

10.4.3 wait()、notify()和 notifyAll()方法

当线程在继续执行前需要等待一个条件时，仅有 synchronized 关键字是不够的。虽然 synchronized 关键字阻止并发访问一个对象，但它没有实现线程之间的通讯。为此，Object 提供了 wait()、notify()和 notifyAll()方法。获得对象锁的线程可以通过调用该对象的 wait() 方法主动释放锁，**等待在该对象的线程等待队列上**，此时其他线程可以得到锁从而访问该 对象，之后可以通过调用 notify()和 notifyAll()方法来唤醒先前因调用 wait()方法而等待的 线程。一般情况下，对于 wait()、notify()和 notifyAll()方法的调用都是根据一定的条件来进 行的。

notifyAll()方法和 notify()方法的区别的是 notifyAll()唤醒所有等待的线程，这些线程中

的一个经过竞争进入临界区，其它的继续等待。

类 BookCase 有两个变量：Book 是 BookCase 中当前内容，布尔变量 available 指示当 前内容是否可以取出。只有当 available 是真时，消费者才能取数据。为实现两线程同步， 必须保证：

(1) 消费者接收数据的前提是 available 为真，即数据单元内容不空； (2) 生产者发送数据的前提是数据单元内容为空。 基于这样的原则，一个 BookCase 的实现可能如下：

public synchronized String get() {

if (available == true) {

available = false;

return Book;

}

}

public synchronized void put(String newBook) {

if (available == false) {

available = true; Book = newBook;

}

}

但是这样的实现却是不对的。让我们考虑 get()方法，当生产者还没放入书、available

为假时，get()什么都没有做。同样，当消费者没取走书、available 为真时，put()什么都没 做。很自然的，我们会想到：消费者应该等待生产者放入书，而生产者放入书后通知消费 者取书。同样，生产者应当等待消费者取走书，而消费者也应该通知生产者书已经取走了。

在下面这个实现中，我们通过调用对象的 notifyAll()和 wait()方法来实现上述的功能。

public class BookCase {

private String Book;

private boolean available = false;

public synchronized String get() {

while (available == false) {

try {

wait();

} catch (InterruptedException e) { }

}

available = false;

notifyAll();

return Book;

}

public synchronized void put(String newBook) {

while (available == true) {

try {

wait();

} catch (InterruptedException e) { }

}

Book = newBook; available = true; notifyAll();

}

}

get()方法包含了一个 while 循环，结束条件是 available 为真。如果 available 为假的话，

消费者就知道生产者还没有产生新的数据，将继续等待。while()循环中调用 wait()方法， 等待生产者线程发送消息。当 put()方法调用 notifyAll()时，消费者线程被唤醒并继续 while() 循环。

同样，生产者调用 put()方法，在 available 为真时调用 wait()进入睡眠，它将一直等待

到消费者取走书。

可能出现的问题是： 在 get()方法的开始，available 为假，消费者线程必须等待生产 者发送数据，那么，如果消费者占据锁，生产者如何发送呢？同样，若数据未被消费者取 走，而生产者占据锁，消费者怎样获得数据呢？**在 Java 中是这样处理的：当一个线程进入 等待状态后，锁就会自动释放，而当它被唤醒后，该线程又占据锁。这样就使等待的线程 有机会进入临界区。**

10.4.4 饥饿和死锁

为了进入临界区，线程必须取得对象的锁。假设现在有两个线程要独占的访问两个对 象：对象 A 和对象 B，任一个线程要访问其中一个对象都需要获得对象锁。如果在某一个 时刻，第一个线程已经持有对象 A 的锁并正在企图获得对象 B 的锁，而同时，第二个线程 已经持有对象 B 的锁并正在企图获得对象 A 的锁。那么它们各自占有对象 A 和对象 B 的 锁，并且同时等待另外一个线程释放另外一个对象的锁，结果是，这两个线程将一直等待， 这就形成了死锁。

死锁问题的一个著名问题是哲学家问题：一个房间内有 5 个哲学家，他们的生活就是

思考和进食。房间里有一张圆桌，中间放着一盘通心粉（假定通心粉无限多）。桌子周围放 有五把椅子，分别属于五位哲学家每两位哲学家之间有一把叉子，哲学家进食时必须同时 使用左右两把叉子。如果在某个时间，哲学家同时拿起他们左边的叉子，而拿起叉子后就 不主动放下，那么所有的哲学家都会在等待他们右边叉子的过程中饿死。

为了避免死锁，一般应该遵循以下一些原则：资源归类，将各种资源归入若干个不同

的资源类(Resource Group)中。资源排序，在不同资源类之间规定次序，对不同资源类中的 资源采用线性按序申请的方法。针对性优化，对同一资源类中的资源，采用适当的方法。 如前面所述的问题，如果对象 A 和 B 的锁总是按字母顺序取得，则不会出现死锁。

在编写多线程代码时，死锁是最难处理的问题之一。 因为死锁可能在最意想不到的地

方发生，所以查找和修正它既费时又费力。例如，试考虑下面这段锁定了多个对象的代码。

public int sumArrays(int[] a1, int[] a2){

int value = 0;

int size = a1.length;

if (size == a2.length) {

synchronized(a1) { //A

synchronized(a2) { //B

for (int i=0; i<size; i++)

value += a1[i] + a2[i];

}

}

}

return value;

}

这段代码在求和操作中访问两个数组对象之前正确地锁定了这两个数组对象。它形式

简短，编写也适合所要执行的任务；但不幸的是，它有一个潜在的问题。这个问题就是它 埋下了死锁的种子，除非您在不同的线程中对相同的对象调用该方法时格外小心。要查看

潜在的死锁，请考虑如下的事件序列：

创建两个数组对象，ArrayA 和 ArrayB。线程 1 用下面的调用来调用 sumArrays 方

法：

sumArrays(ArrayA, ArrayB);

线程 2 用下面的调用来调用 sumArrays 方法：

sumArrays(ArrayB, ArrayA);

线程 1 开始执行 sumArrays 方法并在//A 处获得对参数 a1 的锁，对于这个调用而 言，它就是对 ArrayA 对象的锁。然后在//B 处，在线程 1 获得对 ArrayB 的锁之前被抢先。

线程 2 开始执行 sumArrays 方法并在 //A 处获得对参数 a1 的锁，对于这个调用而 言，它就是对 ArrayB 对象的锁。

然后线程 2 在//B 处试图获取对参数 a2 的锁，它是对 ArrayA 对象的锁。因为这个锁当 前由线程 1 持有，所以线程 2 被阻塞。

线程 1 开始执行并在//B 处试图获取对参数 a2 的锁，它是对 ArrayB 对象的锁。因为 这个锁当前由线程 2 持有，所以线程 1 被阻塞。

现在两个线程都被死锁。 避免这种问题的一种方法是让代码按固定的全局顺序获取锁。 在本例中，如果线程 1

和线程 2 **按相同的顺序对参数调用 sumArrays 方法，就不会发生死锁。但是，这一技术要 求，多线程代码的程序员在调用那些锁定作为参数传入的对象的方法时需要格外小心**。在 您遇到这种死锁并不得不进行调试之前，使用这一技术的应用程序似乎不切实际。

另外，**也可以将锁定顺序嵌入对象的内部。这允许代码查询它准备为其获得锁的对象，**

**以确定正确的锁定顺序。只要即将锁定的所有对象都支持锁定顺序表示法，并且获取锁的 代码遵循这一策略，就可避免这种潜在死锁的情况。**

**在对象中嵌入锁定顺序的缺点是，这种实现将使内存需求和运行时成本增加**。另外， 在上例中应用这一技术需要在数组中有一个包装对象，用来存放锁定顺序信息。例如，试 考虑下面的代码，它由前面的示例修改而来，其中实现了锁定顺序技术：

class ArrayWithLockOrder{

private static long num\_locks = 0;

private long lock\_order;

private int[] arr;

public ArrayWithLockOrder(int[] a)

{

arr = a;

synchronized(ArrayWithLockOrder.class) {

num\_locks++; // 锁数加 1。

lock\_order = num\_locks; // 为此对象实例设置唯一的 lock\_order。

}

}

public long lockOrder()

{

return lock\_order;

}

public int[] array()

{

return arr;

}

}

class SomeClass implements Runnable

{

public int sumArrays(ArrayWithLockOrder a1, ArrayWithLockOrder a2)

{

int value = 0;

ArrayWithLockOrder first = a1; // 保留数组引用的一个

ArrayWithLockOrder last = a2; // 本地副本。

int size = a1.array().length;

if (size == a2.array().length)

{

if (a1.lockOrder() > a2.lockOrder()) // 确定并设置对象的锁定

{ // 顺序。

first = a2;

last = a1;

}

synchronized(first) { // 按正确的顺序锁定对象。

synchronized(last) {

int[] arr1 == a1.array(); int[] arr2 == a2.array(); for (int i=0; i<size; i++)

value += arr1[i] + arr2[i];

}

}

}

return value;

}

public void run() {

//...

}

}

在第一个示例中，ArrayWithLockOrder 类是作为数组的一个包装提供的。每创建该类

的一个新对象，该类就将 static num\_locks 变量加 1。一个单独的 lock\_order 实例变量被 设置为 num\_locks static 变量的当前值。这可以保证，对于该类的每个对象，lock\_order 变 量都有一个独特的值。lock\_order 实例变量充当此对象相对于该类的其他对象的锁定顺序 指示器。

请注意，static num\_locks 变量是在 synchronized 语句中进行操作的。这是必须的，因

为对象的 每个实例 共享该对 象的 static 变量。因此，当 两个线程 同时创建

ArrayWithLockOrder 类的一个对象时，如果操作 static num\_locks 变量的代码未作同步处

理，该变量就可能被破坏。对此代码作同步处理可以保证，对于 ArrayWithLockOrder 类的 每个对象，lock\_order 变量都有一个独特的值。

此外还更新了 sumArrays 方法，以使它包括确定正确锁定顺序的代码。在请求锁之前， 将查询每个对象以获得它的锁定顺序。 编号较小的首先被锁定。此代码可以保证，不管各 对象是以什么顺序传给此方法，它们总是被以相同的顺序锁定。

实现嵌入的锁定顺序需要投入更多的工作，使用更多的内存，并会延长执行时间。但 是，如果代码中可能存在这些类型的死锁，这样做是值得的。如果无法承受额外的内存和 执行开销，或者不能接受 num\_locks 或 lock\_order 重新开始的可能性，则在建立锁定对象 的预定义顺序时应该仔细斟酌。

10.5 线程组

线程组顾名思义是对线程进行分组。Java 中规定，线程组中可以包括若干线程和线程 组。线程组中的线程可以访问本线程组，但不可以越过线程组的限制再访问外层线程/线程 组。这样可以将线程的操作限制在给定范围。

**每个线程都隶属于唯一一个线程组，这个线程组在线程创建时指定并在线程的整个生 命期内都不能更改。你可以通过调用包含 ThreadGroup 类型参数的 Thread 类构造函数来 指定线程属的线程组，若没有指定，则线程缺省地隶属于名为 system 的系统线程组。**

**在 Java 中，除了预建的系统线程组外，所有线程组都必须显式创建**。

在 Java 中，除系统线程组外的每个线程组又隶属于另一个线程组，你可以在创建线 程组时指定其所隶属的线程组，若没有指定，则缺省地隶属于系统线程组。这样，所有线 程组组成了一棵以系统线程组为根的树。

**Java 允许我们对一个线程组中的所有线程同时进行操作，比如我们可以通过调用线程 组的相应方法来设置其中所有线程的优先级，也可以启动或阻塞其中的所有线程。**

线程组机制的另一个重要作用是线程安全。线程组机制允许我们通过分组来区分有不 同安全特性的线程，对不同组的线程进行不同的处理，还可以通过线程组的分层结构来支 持不对等安全措施的采用。**Java 的 ThreadGroup 类提供了大量的方法来方便我们对线程 组树中的每一个线程组以及线程组中的每一个线程进行操作。**

10.5.1 创建一个线程组

下面通过一个程序来说明线程组的创建和它的一些方法的使用，程序代码如下：

import java.lang.\*;

import java.io.\*;

public class ThreadGroupDemo{

public static void main(String[] args){

ThreadGroup grp1=new ThreadGroup("Group1"); ThreadGroup grp2=new ThreadGroup("Group2"); System.out.println("Group1: "+grp1.toString());

Thread thread1=new Thread(grp1,new AThread());

ThreadGroup parent=grp1.getParent(); System.out.println("Group1's parent: " + parent); System.out.println("ParentGroup's info:"); parent.list();

System.out.println("ActiveGroupCount in ParentGroup: "

+ parent.activeGroupCount()); System.out.println("ActiveCount in ParentGroup: "

+ parent.activeCount());

thread1.start();

System.out.println("Current thread's Group: "

+ Thread.currentThread().getThreadGroup());

}

}

class AThread extends Thread{

public AThread(){

super();

}

public void run(){

System.out.println("I am Thread: " + this.toString());

}

}

程序工作是这样的：先创建两个 ThreadGroup 对象，再在第一个 ThreadGroup 对象中 创建一个线程。然后就是打印一些信息。先输出了第一个 ThreadGroup 对象的有关信息， 再显示其父线程组的信息。然后调用 list()方法打印该父线程组中各级成员列表。然后调用 activeGroupCount()方法和 activeCount()方法统计了列表中线程组与线程的数目，之后是当 前线程的所在线程组描述，最后是关于一个线程的信息。

程序的开始我们创建了线程组对象，构造函数的参数为线程组名。ThreadGroup 类另

外还有一个构造函数：

public ThreadGroup(ThreadGroup parent,String name)

比前一种形式多一个参数指明该线程组的父线程组。程序创建了一个线程。这个构造 函数看来也许有些奇特，它的形式为：

public Thread(ThreaqdGroup group,Runnable target)

意即创建这样一个线程，它属于线程组 gruop，并利用 target 的 run()方法。例子中 target 是一个 AThread 对象。而 AThread 是 Thread 的子类，由于 Thread 类实现了 Runnable 接口， 因此 AThread 的对象就可以作为 Runnable 的一个实例，从而担当 target 了。这里我们再解 释一下为什么会有 Thread-0 和 Thread-1 两个线程。其中一个是我们创建并使之运行的 thread1，另一个就是程序中 new AThread()的结果：

Thread thread1=new Thread(grp1,new AThread());

程序的输出如下：

Group1: java.lang.ThreadGroup[name=Group1,maxpri=10] Group1's parent: java.lang.ThreadGroup[name=main,maxpri=10] ParentGroup's info: java.lang.ThreadGroup[name=main,maxpri=10]

Thread[main,5,main] Thread[Thread-0,5,main] java.lang.ThreadGroup[name=Group1,maxpri=10]

Thread[Thread-1,5,Group1]

java.lang.ThreadGroup[name=Group2,maxpri=10] ActiveGroupCount in ParentGroup: 2

ActiveCount in ParentGroup: 3

Current thread's Group: java.lang.ThreadGroup[name=main,maxpri=10] I am Thread: Thread[Thread-0,5,main]

10.5.2 线程组的方法

线程组类的方法与线程类有不少相似之处。如 getName()得到线程组的名字。isDaemon() 判断线程组是否为守护线程组。setDaemon()设置线程组为守护线程组（当参数为 true）或 用户线程组（参数为 false）。checkAccess()判断线程是否允许被更改。

parentOf(ThreadGroup g)方法可以用于判断这个线程组是否是另外一个线程组的父线 程组。

ThreadGroup 的方法 activeGroupCount()计算它包含的线程组数目，而 activeCount()计 算它包含的线程的数目。

线程可以用 getThreadGroup()来访问它所在的线程组。但越界访问线程组之外的线程 或线程组是不允许的。

线程组类还有几种重载形式的 enumerate()方法用以把组内活动线程组或线程的引用拷 贝进指定数组。返值为活动线程组/线程的数目。例如，在上面的例子中，若用

int ct=grp1.enumerate(threadList);

其中 threadList 是线程数组，则会把 Thread-1 的引用拷贝入 threadList，ct 则被赋值为

1。

getMaxPriority()用于获取线程组的最大优先级。setMaxPriority()方法则用于设置线程

组的最大优先级。

对于 setMaxPriority()方法，这个方法对于线程组中优先级大于参数 pri 的线程没有作 用。这个方法首先调用 checkAccess()方法判断这个线程组的线程是否允许被更改，如果不 允 许 则 会 抛 出 一 个 例 外 。 然 后 判 断 setMaxPriority() 方 法 的 参 数 pri 是 否 在 Thread.MIN\_PRIORITY 和 Thread.MAX\_PRIORITY 之间，如果不是，则什么都不改变， 否则这个线程组的对象的优先级将被设成参数 pri 和该线程组的父线程组所允许的最大优 先级中较小的一个，对于没有父进程组的系统进程组，它们的线程优先级将被直接设置程 pri。这个过程将对线程组的所有子线程组递归调用。

10.6 本章小结

程序中单个顺序的流控制称为线程，多线程则指的是在单个程序中可以同时运行多个 不同的线程执行不同的任务。多线程是 Java 语言的一个很重要的特点，多线程程序设计大 大提高了程序执行效率和处理能力。

Java 提供了类 java.lang.Thread 来方便多线程编程除此之外，通过 Java 的计时器 Timer

和 Runnable 接口也可用以创建 Java 线程。 计时器适用于简单的重复性任务，更复杂的任务则需要 Thread 类或 Runnable 接口。

线程有 4 个状态：**创建、可运行、阻塞（不可运行）和中止状态，线程在这四个状态中转 换，构成了线程的生命周期。**

线程的优先级决定了线程获取 CPU 时间的能力，Java 线程的优先级可由用户来设置。 Java 优先级的范围在 Thread.MIN\_PRIORITY 和 Thread.MAX\_PRIORITY 之间。Thread 类 提供了 setPriority()和 getPriority()方法来设置和读取优先级。

在某些情况下，有一些**同时运行的线程需要共享数据**，例如一个线程向文件写数据， 而同时另一个线程从同一文件中读取数据，因此就必须考虑其它线程的状态与行为，这时 就需要实现同步来得到预期结果。对象锁、wait()和 notify()是线程同步常用的方法。使用 锁应当注意避免死锁的情况发生。

线程组顾名思义是对线程进行分组。线程组的作用是方便对一系列线程同时进行操作，

线程组机制的另一个重要作用是线程安全。

10.7 上机练习

1. 验证本章各小节的例子。

2. 实现读者-写者问题的同步。问题描述如下：对共享资源的读写操作，任一时刻写 者最多只允许一个，而读者则允许多，读和写不能同时进行，而但多个读者同时读则是允 许的。

3. 编写一个程序用于模拟哲学家问题，注意避免哲学家饿死的情况。

10.8 习题

1. 现在有如下的一个类：

class Counter {

public static void main(String[] args) {

Thread t = new Thread(new CounterBehavior());

t.start();

}

}

请给出 CounterBehavior 的实现，使得该程序能输出从 1 到 100 的整数，并且隔一秒输

出一次。

2. 现在有如下一个类：

class Counter {

public static void main(String[] args) {

new Counter().go();

}

void go() {

// A

Thread t = new Thread(a);

t.start();

}

}

以下哪些代码可以填入 A 处，使得程序能输出 1 到 100 的整数：

(1)

Runnable a = new Runnable() {

public void run() {

for (int i = 1; i <= 100; i++) { System.out.println(i);

}

}

};

(2)

a implements Runnable {

public void run() {

for (int i = 1; i <= 100; i++) { System.out.println(i);

}

}

};

(3)

Thread a = new Thread() {

public void run() {

for (int i = 1; i <= 100; i++) { System.out.println(i);

}

}

};

3. 现在有如下一个类：

class RunTest implements Runnable {

public static void main(String[] args) { RunTest rt = new RunTest();

Thread t =new Thread(rt);

//A

}

public void run() { System.out.println("running");

}

void go() {

start(1);

}

void start(int i) {

}

}

在 A 处填写代码，使得程序能输出“running”。

4. 编译和运行下面的程序，请指出结果。

public class X extends Thread implements Runable{

public void run(){ System.out.println("this is run()");

}

public static void main(String args[])

{

Thread t=new Thread(new X());

t.start();

}

}

5. 关于 wait()和 notify()，哪些说法是正确的？

(1) 如果多个线程同时等待一个锁，notify()只能唤醒等待时间最长的那个线程。

(2) 如果多个线程同时等待一个锁，我们无法判断 notify()将唤醒哪个线程。

(3) notify()是在 Thread 类中定义的。

(4) notify()只能在 while 循环中被调用。

6.下 面哪个方法可以在任何时候被任何线程调用?

(1) notify()；(2) wait()；(3) notifyAll()；(4) sleep()；(5) yield()；(6) synchronized(this)