Chapter 3 第3章 线程的管理

线程启动之后，通常需要对线程的运行进行管理，本章主要介绍线程管理的方法。

3.1 线程数目的确定

在并行编程过程中，很多情况下会有这样的问题：在程序中创建多少个线程最合适？创建多少个线程才能使程序性能最佳？

线程数目的多少对多核处理器性能的发挥有一定的影响。如果创建线程的数目较少，则会导致多核处理器的多个处理核处于闲置状态，性能得不到完全发挥，浪费了系统的资源。如果创建线程的数目过多，则由于多核处理器的处理能力是一定的，多余的线程不能马上得到执行，浪费了软件资源，这也会在一定程度上影响程序的性能。可见，在并行程序中创建线程的数目过多或过少都不好。

在解决一个大的问题时，通常希望创建足够多的线程数，以满足计算的需求，然而在很多情况下，在程序中设定线程数为多核处理器可以同时处理的最大线程数，Java 中提供了相关的方法，可用于获取处理器可以同时处理的最大线程数，该方法如下：

```
int nthreads = Runtime.getRuntime().availableProcessors();
```

其中，availableProcessors()方法得到的是 Java 虚拟机可以使用的逻辑意义上的处理核数 N，显然 N 是一个整型值。例如，如果一个多核处理器有 4 个处理核，并且该处理器不支持超线程，则通过调用方法 availableProcessors()得到的值为 4；如果该多核处理器支持超线程，则通过调用方法 availableProcessors()得到的值为 8。

通过查看 Windows 操作系统的任务管理器可以查看处理器的逻辑核数。图 3-1 展示了两颗 Intel Xeon CPU E5-2650 多核处理器的 CPU 处理核的使用情况，每颗 CPU 有 8 个处理核，每个核均支持超线程，从图 3-1 可以看出，该处理器可以支持 32 个线程同时运行。

在选择线程数时，可以根据具体情况决定。如果程序中所有任务都是计算密集型任务，则设定线程数为多核处理器可以同时处理的最大线程数。计算密集型指程序的大部分时间用于计算，而非等待或阻塞。如果阻塞的时间占很大的比例，则不属于计算密集型。

如果程序中的大部分任务是 I/O 密集型任务，则应创建尽可能多的线程，这是因为当一个线程执行的任务遇到 I/O 操作时，该线程将阻塞，这时处理器将进行环境切换，继而转为其他线程执行，这样可以让处理核一直处于忙碌状态，以提高处理器的利用率。

线程数的选取和程序阻塞时间也有一定的关系，设定任务的阻塞时间占任务完成总时间的比例为 f，显然，$0 \leqslant f < 1$。计算线程总数 t 的公式可以表示如下：

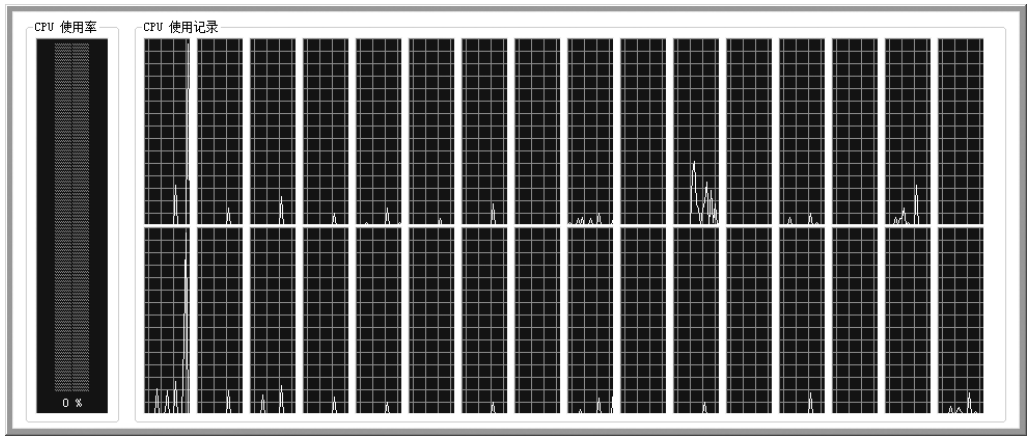

图 3-1　16 核 CPU 支持 32 线程性能情况

$$t = \frac{\text{处理器的逻辑处理核数 } N}{1-f}$$

如果一个程序中的所有任务都会有 $f=50\%$的阻塞时间，则此时设定的线程数应为多核处理器逻辑核数的两倍。当 $f=0$ 时，表示每个任务都处于一直计算的忙碌状态，没有任何等待，则可以设定线程数 t 为多核处理器的逻辑处理核数；当 f 接近 1 时，表示很多任务都有相当多的阻塞时间，应尽可能多地设定线程数。

3.2　线程运行的控制

线程启动以后可以管理线程，使线程休眠、等待或中断执行等。

3.2.1　等待线程执行完毕

在某些情况下，需要让某一个线程等待另一个线程执行结束后再开始执行该线程。例如，线程 1 和线程 2 共同完成一个计算并输出结果的任务，线程 1 完成相关的计算，线程 2 用于输出结果，显然，线程 2 需要等待线程 1 计算出结果后再输出。

可以使用线程类的方法 join()实现上述功能，当调用某个线程类对象实例的 join()方法后，将会等待该线程类对象执行结束。

方法 join()的定义形式如下：

```
//没有参数的方法
• public final void join() throws InterruptedException

//参数 millis 为等待的毫秒数
• public final synchronized void join(long millis) throws InterruptedException

//参数 millis 为等待的毫秒数，nanos 为等待的纳秒数
• public final synchronized void join(long millis, int nanos) throws InterruptedException
```

从上面的方法定义可以看出，方法 join()可能会抛出 InterruptedException 异常，所以使用 join()方法的代码需要包含在 try…catch…代码语句中，形式如下：

```
Thread t = new Thread();
try{
    t.join();
}catch(InterruptedException e){
    //…
}
```

下面通过一个例题演示方法 join()的使用。

【例 3-1】 定义 3 个线程，线程 A 用于产生若干随机数，线程 B 用于计算这些数的和，线程 C 用于输出结果，只有当线程 A 完成后，线程 B 才能计算，计算完成后，线程 C 才能输出。

【解题分析】

题意明确规定了线程之间的先后关系，可以通过方法 join()控制。需要注意的是，在线程 B 执行前，线程 A 需要执行完毕，所以在线程 B 的 run()方法中加入线程 A 的 join()方法。

【程序代码】

```
//Producer.java
package book.ch3.join;
//定义类 Producer,实现了 Runnable 接口,代表题目中的线程 A
public class Producer implements Runnable{
    //定义数组 arr
    int[] arr;
    //构造方法定义
    public Producer(int[] arr){
        //对类属性赋值
        this.arr = arr;
    }
    //run()方法定义
    public void run(){
        //对数组元素进行初始化赋值
        for(int i=0; i<arr.length; i++){
            arr[i] = (int)(Math.random() * 100);
        }
    }
}
//Worker.java
package book.ch3.join;
//定义类 Worker,该类实现了 Runnable 接口,代表题目中的线程 B
public class Worker implements Runnable {
    //定义数组 arr
    int[] arr;
    //定义线程对象 thread
```

```
    Thread thread;
    //构造方法定义
    public Worker(int[] arr, Thread thread){
        this.arr = arr;
        this.thread = thread;
    }
    //run()方法定义
    public void run(){
        //使用 join()方法,等待 thread 线程执行完毕
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //数组中各元素求和
        int sum = 0;
        for(int i=0; i<arr.length; i++){
            sum += arr[i];
        }
        //将求和结果 sum 赋给全局静态变量 sum
        Index.sum = sum;
    }
}
//PrintTask.java
package book.ch3.join;
//定义类 PrintTask,该类实现了 Runnable 接口,代表题目中的线程 B
public class PrintTask implements Runnable{
    //定义线程对象 thread
    Thread thread;
    //构造方法定义,对类属性 thread 赋值
    PrintTask(Thread thread){
        this.thread = thread;
    }
    //run()方法定义
    public void run(){
        //通过 join()方法,让 thread 线程等待
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //输出结果
        System.out.println("sum="+Index.sum);
    }
}
```

```
//Index.java
package book.ch3.join;
public class Index {
    //静态全局变量 sum,初始值为 0
    static int sum = 0;
    public static void main(String[] args) {
        //定义数组 arr,大小为 100000
        int[] arr = new int[100000];
        //创建线程 p 并启动
        Thread p = new Thread(new Producer(arr));
        p.start();
        //创建线程 w 并启动
        Thread w = new Thread(new Worker(arr, p));
        w.start();
        //创建线程 o 并启动
        Thread o = new Thread(new PrintTask(w));
        o.start();
        //等待线程结束
        try {
            o.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //校验结果
        int t = 0;
        for (int i = 0; i < arr.length; i++) {
            t += arr[i];
        }
        if (t == sum) {
            System.out.println("验证通过!");
        } else {
            System.out.println("验证失败. t=" + t + ", sum=" + sum);
        }
    }
}
```

【运行结果】

程序运行结果如图 3-2 所示。由于数据随机产生,因此 sum 的计算结果每次都不同。

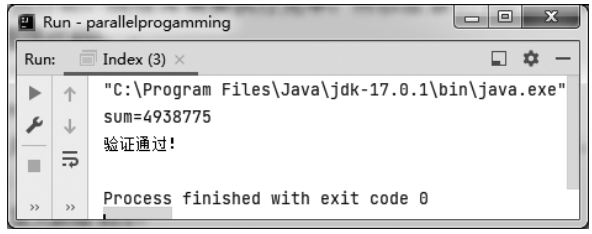

图 3-2 运行结果

【相关讨论】

该例让一个线程等待另一个线程执行完毕，也可以让一个线程等待若干线程执行完毕，读者可以自行练习。

3.2.2　休眠

方法 sleep()用于使一个线程暂停运行一段固定的时间，暂停时间的具体长短由 sleep()方法的参数给出。方法 sleep()的定义形式如下：

```
//参数 millis 指明了休眠的毫秒数
• public static native void sleep(long millis) throws InterruptedException

//参数 millis 指明了休眠的毫秒数，nanos 指明了休眠的纳秒数
• public static void sleep(long millis, int nanos) throws InterruptedException
```

方法 sleep()可能会抛出 InterruptedException 异常，因此需要将该方法放入 try…catch…语句中。

在线程暂停执行的这段时间中，CPU 的时间片会让给其他线程，从而使其他线程可以交由 CPU 执行。

线程的调度是按照线程的优先级顺序进行的。当高优先级的线程存在时，低优先级的线程获得 CPU 的机会很小。有时，高优先级的线程需要与低优先级的线程同步，此时高优先级的线程将会让出 CPU，使低优先级的线程有机会运行。高优先级的线程可以通过在它的 run()方法中调用 sleep()方法使自己退出 CPU 休眠一段时间，休眠结束后，如果条件具备，线程即可进入运行状态。

【例 3-2】　输出 0～10，每输出一个数字后线程休眠 1s。

【解题分析】

在输出一个数字后，调用线程的方法 sleep()，sleep 的参数 millis 为毫秒数，故应使用 1000ms。

【程序代码】

```
//Worker.java
package book.ch3.sleep;
public class Worker extends Thread {
    public void run() {
        //循环 11 次，每次间隔半秒输出 i 的值
        for (int i = 0; i <= 10; i++) {
            System.out.print(" " + i);
            //线程休眠 1000ms
            try{
                sleep(1000);
            }catch(InterruptedException e){  }
        }
    }
}
//Index.java
package book.ch3.sleep;
public class Index {
```

```
    public static void main(String[] args) {
        //创建两个线程对象 worker1 和 worker2
        Thread worker1 = new Worker();
        Thread worker2 = new Worker();
        worker1.start();
        worker2.start();
    }
}
```

【运行结果】

程序运行结果如图 3-3 所示。

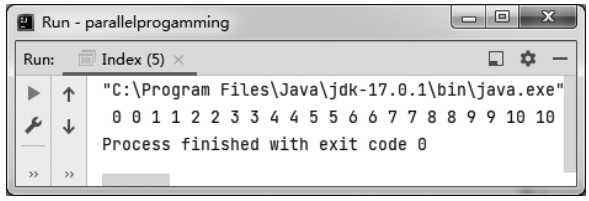

图 3-3 运行结果

【相关讨论】

如果在同步语句中使用 sleep()方法，则线程在睡眠期间不会丢掉对于任何监视器的持有权。

3.2.3 中断

一个线程除了正常执行结束外，也可以人为地中断线程的执行，线程的中断可以使用 interrupt()方法。

除非是一个线程正在尝试中断它本身，否则中断的请求一般都会被接受。如果一个线程由于调用 wait()方法或 join()方法正处于阻塞队列中，则中断请求不会被响应，并会抛出 InterruptedException 异常。

在早期的 Thread 类方法中，还可以使用 stop()方法终止一个线程的执行，但 stop()方法现在已不推荐使用。

在程序中调用线程的方法 interrupt()后，通常需要在线程的 run()方法中使用类 Thread 的方法 isInterrupted()进行判断，并根据判断结果执行相应的操作。

需要注意的是，interrupt()和 isInterrupted()是两个非常类似的方法。interrupt()方法是线程的一个静态方法，调用该方法会清除线程的中断状态；isInterrupted()是一个实例方法，主要用于检查是否被中断，调用该方法不会清除线程的中断状态。

【例 3-3】 在线程中从 2000 年开始输出所有闰年，直到线程被中断。

【解题分析】

闰年的判断方法：如果某一年份能够被 4 整除，并且不能被 100 整除，则该年份是闰年；如果某一年份能够被 400 整除，则该年份也是闰年。

线程中需要通过无限循环不断地输出闰年，在主线程中等待一段时间后，向线程发出中断请求，线程中通过方法 isInterrupted()判断是否处于中断状态。

【程序代码】

```
//LeapYearPrinter.java
package book.ch3.interrupt;
public class LeapYearPrinter extends Thread{
    public void run(){
        //从 2000 年开始
        int year = 2000;
        System.out.println("闰年包括:");
        while(true){
            //闰年判断
            if(year%4==0&&year%100!=0 || year%400==0){
                System.out.println("闰年:"+year);
            }
            //判断线程是否被中断,如果是,则输出信息并返回
            if(isInterrupted()){
                System.out.println("线程类 LeapYearPrinter 已经被中断.");
                return;
            }
            year++;
        }
    }
}
//Index.java
package book.ch3.interrupt;
public class Index {
    public static void main(String[] args) {
        //创建线程对象
        Thread newThread = new LeapYearPrinter();
        //启动线程
        newThread.start();
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //通过 interrupt 方法中断线程
        newThread.interrupt();
    }
}
```

【程序分析】

线程不断输出闰年,并让线程在休眠 1ms 后中断线程。这里选择休眠 1ms 是为了使输出在可控的范围内,避免输出过多,读者可以自行调节。

【运行结果】

程序运行结果如图 3-4 所示。

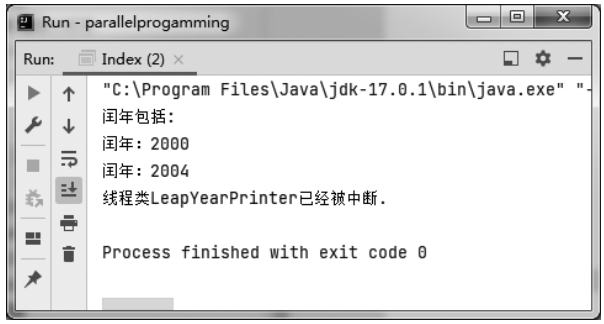

图 3-4 运行结果

JDK 1.0 提供了线程的方法 stop()和 suspend(),这两个方法通常用于控制线程的停止和挂起,其中,方法 stop()用来直接终止线程,方法 suspend()会一直阻塞当前线程,直到调用该线程的 resume()方法。由于缺乏安全性和容易导致死锁等原因,这两个方法从 JDK 2.0 版本开始不再推荐使用。

3.2.4 让出 CPU 的使用权

为了防止某个线程独占 CPU 资源,可以让当前执行的线程让出 CPU 的使用权,yield()方法可以实现该功能。

方法 yield()用于使当前线程让出 CPU 的使用权,但是这并不能保证 CPU 接下来调用的不是该线程。

该方法常用于线程的调试和测试环境,用于发现由于竞争条件而引起的错误,也可以用于一些并行的数据结构设计中。

【例 3-4】 令线程从 1 开始输出到 5,当输出到 3 时,令线程让出 CPU 的使用权。

【解题分析】

使用循环输出 1～5,当值为 3 时,使用 yield()方法阻塞线程。

【程序代码】

```
//Worker.java
package book.ch3.yield;
//扩展 Thread 创建类 Worker
public class Worker extends Thread {
    public void run() {
        System.out.println(this.getName() + "开始执行");
        for (int i = 1; i <= 5; i++) {
            //当输出到 3 时,调用线程 yield 方法
            if (i == 3) {
                Thread.yield();
                System.out.println(this.getName() + "让出了 CPU 的使用权");
            }
            System.out.println(this.getName() + "正在输出" + i);
```

```
        }
        System.out.println(this.getName() + "结束");
    }
}
//Index.java
package book.ch3.yield;
public class Index {
    public static void main(String[] args) {
        //创建两个线程并启动
        Thread t1 = new Worker();
        Thread t2 = new Worker();
        t1.start();
        t2.start();
    }
}
```

【运行结果】

程序运行结果如图 3-5 所示。

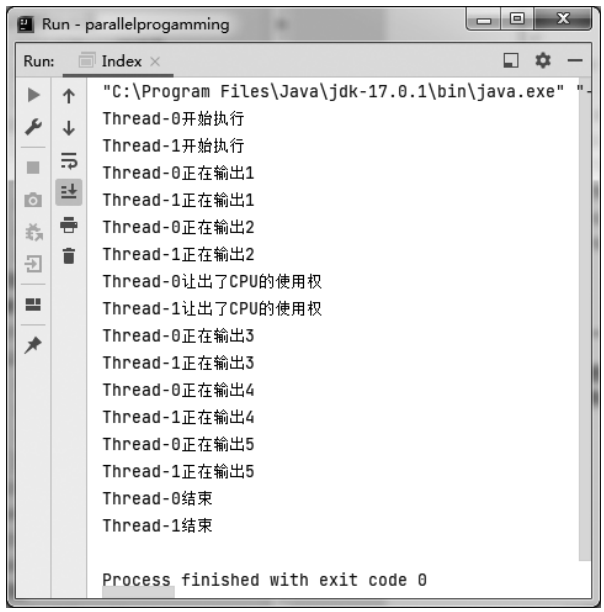

图 3-5　运行结果

【相关讨论】

从运行结果可以看出，在输出 2 以后，线程 1 和线程 2 分别让出了 CPU 的使用权，但在下次执行时又获取了 CPU 的使用权，然后继续执行。

3.3 守护线程

前面使用的线程一般称为用户线程(user thread),Java 中还有一类特殊的线程,称为守护线程(daemon thread)。在 Java 虚拟机中,守护线程的典型例子是垃圾收集器(garbage collector,GC)。

守护线程与其他线程没有太大的不同,它的唯一作用是为用户线程提供服务。当只剩下守护线程时,虚拟机会退出,这主要是因为没有可服务的线程,守护线程的运行就没有必要了。

在守护线程中,通常设定了一个无限的循环,用于等待服务的请求或完成某个任务,守护线程一般不会承担重要的任务,这主要是因为一方面守护线程具有较低的优先级,不确定守护线程在什么时候可以获得 CPU 的时间片,另一方面不确定守护线程在什么时候结束运行。

可以将一个线程变为守护线程,方法是设置线程的属性方法 setDaemon(),该方法定义的形式如下:

```
public final void setDaemon(boolean on)
```

在将线程设置为守护线程时,可能会抛出异常 IllegalThreadStateException 和 SecurityException,如果当前线程是活动的(alive),也就是说,该线程已经通过方法 start()等途径启动,则抛出异常 IllegalThreadStateException。在设置为守护线程时,会通过方法 checkAccess()判断当前线程是否可以被修改,如果不可以,则抛出异常 SecurityException。

例如:

```
Thread thrd = new Thread();
thrd.setDaemon(true);
thrd.start();
```

如果将方法 setDaemon()的参数值设置为 true,则将该线程标记为守护线程,否则为用户线程。

需要注意的是,该方法必须在调用线程的 start()方法之前调用,一旦线程启动,将无法修改线程的守护状态。如果父线程是守护线程,那么子线程也将是守护线程。可以通过线程的 isDaemon()方法判断某个线程是否为守护线程。

【例 3-5】 使用守护线程完成数据维护的任务,在某一时刻使用守护线程删除队尾的数据。

【解题分析】

通过方法 setDaemon()设置一个守护线程,在守护线程中通过一个无限循环监控队列的数据变化,在某一时刻删除队尾的数据。

【程序代码】

```
//Worker.java
package book.ch3.daemon;
//引入类 LinkedList
import java.util.LinkedList;
public class Worker extends Thread {
    //域属性 list
    private LinkedList<Integer> list;
```

```
    //构造方法定义
    public Worker(LinkedList<Integer> list) {
        this.list = list;
    }
    //重写 run()方法
    @Override
    public void run() {
        //循环 10 次
        for (int i = 0; i < 10; i++) {
            //生成随机数据
            int newData = (int) (Math.random() * 1000);
            //向列表中添加数据
            list.addFirst(newData);
            System.out.println("新的数据" + newData
                               + "被插入列表, Size=" + list.size());
            //休眠 1s
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
//Cleaner.java
package book.ch3.daemon;
import java.util.LinkedList;
public class Cleaner extends Thread {
    //域属性 list
    private LinkedList<Integer> list;
    //构造方法定义
    public Cleaner(LinkedList<Integer> list) {
        //对域属性 list 赋值
        this.list = list;
        //设置为守护线程
        this.setDaemon
(true);
    }
    //重写 run()方法
    @Override
    public void run() {
        while (true) {
            //5s 后开始移除数据
            try {
```

```
                sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            while (true) {
                if (!list.isEmpty()) {
                    list.removeLast();
                    System.out.println("一个数据已经被移除。");
                }
            }
        }
    }
}
//Index.java
package book.ch3.daemon;
import java.util.LinkedList;
public class Index {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        Thread worker = new Worker(list);
        Thread cleaner = new Cleaner(list);
        worker.start();
        cleaner.start();
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run(){
                System.out.println("Java 虚拟机退出");
            }
        });
    }
}
```

【程序分析】

本例包含两个线程类定义，一个为用户线程 Worker，另一个为守护线程 Cleaner。在构造方法中，通过方法 setDaemon()指明该线程为守护线程，在 run()方法中设置一个无限循环，用于不断对队列的大小进行监控，当队列非空时，移除队尾的数据。

【运行结果】

程序运行结果如图 3-6 所示。从图中可以看出，5s 后守护线程开始执行。

【相关讨论】

由上例可见，守护线程可以帮助用户线程完成一些额外的处理工作，由于它的优先级较低，因此一般在 Worker 休眠时执行。

上面的程序只使用了一个用户线程，也可以使用两个及两个以上的线程，但因为共享链表，如果读者将线程设置为多个，则应使用同步机制保护共享数据。

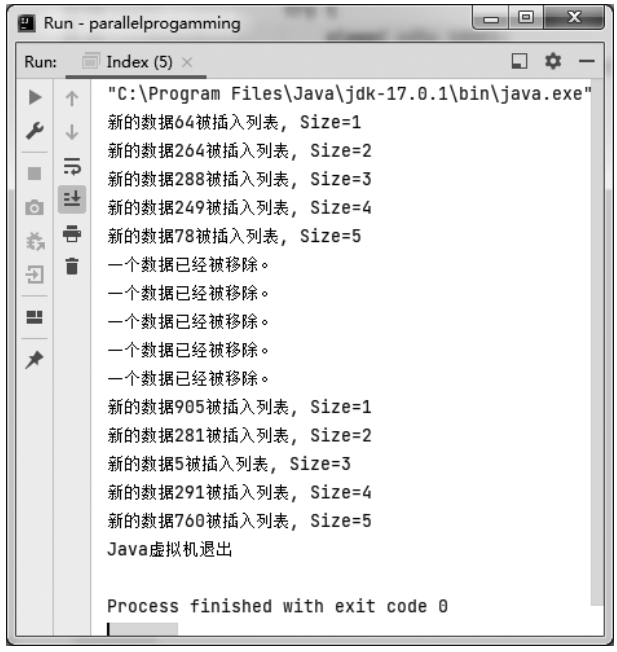

图 3-6　运行结果

3.4　线程分组

如果有若干正在做同一工作的线程，为了方便对这些线程进行管理，可以对这些线程进行分组，从而把分到同一组的若干线程作为一个整体进行操作。

在线程类 Thread 的构造方法中，可以指明线程属于哪一个分组，形式如下：

```
public Thread(ThreadGroup group, Runnable target)
```

其中，参数 group 可以指明该线程属于哪一个线程组。

线程组代表线程的集合，使用类 ThreadGroup 创建。类 ThreadGroup 从 JDK 1.0 开始就已经发布，在包 java.lang 下，而不在包 java.util.concurrent 下。

类 ThreadGroup 常用的构造方法主要有如下两个：

```
//创建一个线程组，通过参数 name 指明线程组的名字，该 name 的值应该是唯一的，可以和其他线程组区分
• public ThreadGroup(String name)

//创建一个线程组，参数 parent 指明了该线程组的父线程组，参数 name 指明了线程组的名字
• public ThreadGroup(ThreadGroup parent, String name)
```

一个线程组可以包含其他线程组，线程组之间形成了一种树状结构，除了初始创建的线程组外，其他线程组都有一个父线程组。一个线程允许访问所属线程组的相关信息，不允许访问父线程组的信息。类 ThreadGroup 的常用方法如表 3-1 所示。

表 3-1 类 ThreadGroup 的常用方法

方 法	含 义
public final String getName()	获取当前线程组的名字
public final ThreadGroup getParent()	获取当前线程组的父线程组
public final int getMaxPriority()	获取当前线程组的最大优先权，属于该线程组的所有线程的优先权不能大于该值
public final boolean isDaemon()	返回当前线程组是否为一个守护线程组
public final void setDaemon(boolean daemon)	将当前线程组设置为一个守护线程组
public int activeCount()	获取当前线程组及其子线程组中活动的线程数
public int enumerate(Thread list[])	将此线程组及其子线程组中的所有活动线程复制到指定数组中
public final void stop()	停止线程组中线程的执行

下面通过例题演示类 ThreadGroup 的用法。

【例 3-6】 使用线程组操作 10 个线程，需要分别创建子线程组和父线程组，并向其中添加 5 个线程，通过 stop()方法停止整个线程组的运行。

【解题分析】

如果想使用线程组操作线程，则需要把线程加入线程组，线程组之间可以形成树状结构，为了指明父线程组和子线程组，需要在线程组创建时通过参数指明父线程组。

【程序代码】

```
//Worker.java
package book.ch3.ThreadGroup;
public class Worker implements Runnable {
    //域属性 counter,用于计数
    int counter = 0;
    @Override
    public void run() {
        //循环若干次,Integer.MAX_VALUE 为整数的最大值
        for(int i=0; i<Integer.MAX_VALUE; i++){
            counter++;
        }
        System.out.println(Thread.currentThread().getName()+"已经停止执行");
    }
}
//Index.java
package book.ch3.ThreadGroup;
public class Index {
    public static void main(String[] args) {
        //线程数
        int threadNum = 5;
        //线程组 parentGroup
```

```
        ThreadGroup parentGroup = new ThreadGroup("父线程组");
        //线程组 parentGroup 的子线程组
        ThreadGroup childGroup = new ThreadGroup(parentGroup,"子线程组");
        Worker worker = new Worker();
        //向父线程组中加入线程
        Thread[] threads = new Thread[threadNum * 2];
        for (int i = 0; i < threadNum; i++) {
            threads[i] = new Thread(childGroup, worker);
            threads[i].start();
        }
        //向子线程组中加入线程
        System.out.println(threadNum+"个线程被加入到子线程组 ");
        for (int i = 0; i < threadNum; i++) {
            threads[threadNum + i] = new Thread(parentGroup, worker);
            threads[threadNum + i].start();
        }
        System.out.println(threadNum+"个线程被加入到父线程组 ");
        //输出活动线程数
        System.out.println("在" + parentGroup.getName()
                    + "中活动线程数为: " + parentGroup.activeCount());
        //调用 stop()方法停止子线程组线程的执行
        childGroup.stop();
        System.out.println("子线程组已经被停止");
        //调用 stop()方法停止父线程组线程的执行
        parentGroup.stop();
        System.out.println("父线程组已经被停止");
    }
}
```

【程序分析】

程序中创建了父线程组和子线程组，为了区分父线程组和子线程组，通过继承关系表明父线程组，然后在创建线程时通过参数指明哪个线程放入了哪个线程组。

【运行结果】

程序运行结果如图 3-7 所示。

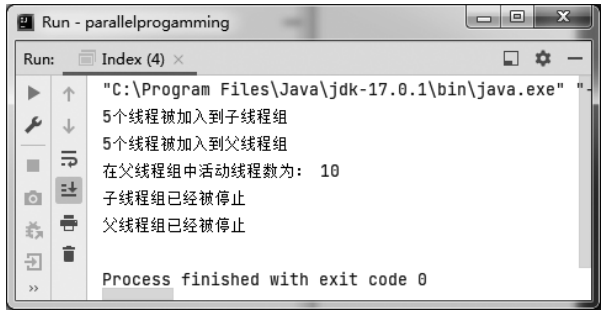

图 3-7　运行结果

【相关讨论】

当有多个线程时，将线程分别加入不同的线程组有利于线程的管理。

3.5 线程本地化

类 ThreadLocal 是一个非常有用的类，Java 通过类 ThreadLocal 实现线程本地对象，使用类 ThreadLocal 将会使变量在每个线程的私有区域内有一个拷贝（或称副本），每个线程都可以相对独立地改变自己的副本，而不会影响其他线程的副本。值得说明的是，ThreadLocal 并不表示一个线程，而是表示线程的一个局部变量。

在类 ThreadLocal 的内部实现机制上，它使用一个哈希表（Hashmap）维护线程的局部变量，哈希表中的键（key）为线程对象，值（value）对应线程的变量副本。类 ThreadLocal 使用原子整型变量 AtomicInteger 作为哈希表的哈希码（Hash code），原子类型保证了在多线程环境下不会导致哈希码的混乱。

类 ThreadLocal 的构造方法如下：

```
//用于创建一个线程本地变量
• ThreadLocal()
```

类 ThreadLocal 提供了方法 set()和 get()，用于设置和读取线程的本地值。一个线程首次获取一个线程本地对象值时将调用方法 initialValue()，该方法用于对每个线程对象进行初始化。

表 3-2 类 ThreadLocal 的常用方法

方　法	说　明
T get()	该方法返回线程本地变量的值
protected T initialValue()	用于设置当前线程本地变量的初始值
void remove()	移除本地变量
void set(T value)	设置本地变量的值

【例 3-7】 定义一个类，用于给每个线程分配一个唯一的 ID。

【解题分析】

ID 是一个线程的标识，可以区分线程，可以使用类 ThreadLocal 给每个线程分配一个唯一的 ID。这里不直接使用类 ThreadLocal，而是定义一个 ThreadLocal 类的子类，并重写方法 initialValue()。

【程序代码】

```
package book.ch3.local;
public class ThreadID {
    //定义私有的静态整型变量 nextID
    private static volatile int nextID = 0;
    //定义一个内部类 ThreadLocalID，它从类 ThreadLocal 继承，ThreadLocal 的尖括号内为 Integer。
在该内部类内，重写了方法 initialValue()
    private static class ThreadLocalID extends ThreadLocal<Integer> {
```

```
        protected synchronized Integer initialValue() {
            return nextID++;
        }
    }
    private static ThreadLocalID threadID = new ThreadLocalID();
    public static int get() {
        return threadID.get();
    }
    public static void set(int index) {
        threadID.set(index);
    }
}
```

【例 3-8】　使用类 ThreadLocal 为每个线程增加时间戳。

【解题分析】

可以为每个线程的启动记录时间情况，为此定义一个 ThreadLocal 实例，并重写它的 initialValue() 方法。

【程序代码】

```
//Worker.java
package book.ch3.threadlocal;
import java.util.Date;
public class Worker extends Thread {
    //定义时间戳,用于记录线程的创建时间
    ThreadLocal<Date> timeStamp = new ThreadLocal<Date>(){
        protected Date initialValue(){
            return new Date();
        }
    };
    @Override
    public void run(){
        System.out.println(getName()+"线程启动于"+timeStamp.get());
    }
}
//Index.java
package book.ch3.threadlocal;
public class Index {
    public static void main(String[] args){
        Thread t1 = new Worker();
        Thread t2 = new Worker();
        t1.start();
        t2.start();
    }
}
```

【运行结果】

程序运行结果如图 3-8 所示。

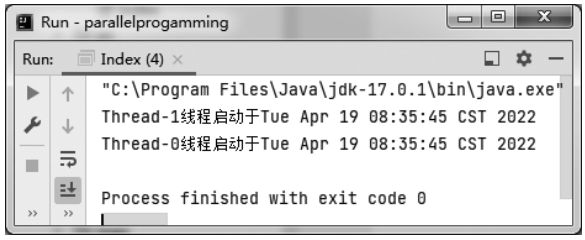

图 3-8 运行结果

【相关讨论】

线程本地化可以让线程拥有属于自己的资源，在 Wloka 等的论文 *Refactoring for reentrancy*[①] 中，通过线程本地化操作可以实现程序的可重入性重构，在他们的方法中，有时甚至不需要使用同步控制，有兴趣的读者可以自行阅读。

3.6 线程开销问题

使用多线程编写的程序可以最大程度地发挥多核处理器的处理能力，提高硬件资源的利用率。引入多线程在提升程序性能的同时，也会引入一些额外的性能开销，例如线程的创建和销毁、线程之间的同步控制、线程之间的切换和调度策略都是增加这一开销的来源。有时，如果使用线程不当，不仅不会带来性能提升，反而会使性能下降。下面通过一个例子说明这个问题。

【例 3-9】 频繁创建多个线程，在创建后只做少量的工作就立即结束，观察程序的执行时间情况。

【解题分析】

创建线程，让线程做很少的工作，例如只输出一条信息，这样在创建线程之后，线程将立刻结束。创建多个这样的线程实例，观察程序的运行时间，并与串行执行的时间进行对比。

【程序代码】

```
//Printer.java
package book.ch3.badperformance;
//定义类 Printer
public class Printer extends Thread {
    @Override
    public void run(){
        //输出线程正在运行的信息
        System.out.println(this.getName()+"正在运行");
    }
}
```

① Wloka J, Sridharan M, Tip F. Refactoring for reentrancy. Proceedings of the European Software Engineering Conference and the ACM Sigsoft International Symposium on Foundations of Software Engineering(ESEC/FSE), 2009, Amsterdam, the Netherlands, August. 173-182.

```
//Index.java
package book.ch3.badperformance;
public class Index {
public static void main(String[] args) {
        //定义线程数
        int threadNum = 100;
        //开始时间
        long start1 = System.nanoTime();
        //为了便于对线程操作,定义线程数组,并生成每一个线程对象
        Thread[] threads = new Thread[threadNum];
        for(int i=0; i<threadNum; i++){
            threads[i] = new Printer();
            threads[i].start();
        }
        //等待这些线程执行结束
        for(int i=0;i<threadNum; i++){
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //多个线程处理的结束时间
        long end1 = System.nanoTime();
        System.out.println("使用线程的执行时间为"+(end1-start1)+"纳秒");
        //串行执行的开始时间
        long start2 = System.nanoTime();
        for(int i=0;i<threadNum; i++){
            System.out.println("正在输出:"+i);
        }
        //串行执行的结束时间
        long end2 = System.nanoTime();
        System.out.println("串行的执行时间为"+(end2-start2)+"纳秒");
    }
}
```

【程序分析】

通过在程序中创建大量线程让线程执行,比较串行和并行处理的时间。

【运行结果】

程序运行结果的部分截图如图 3-9 所示。

【结果分析】

从程序的执行结果可以看出,使用线程的执行时间比串行的执行时间要多,这是因为在并行程序和串行程序执行同样工作的情况下,线程的创建和启动需要耗费更多的时间,此外,线程的上下文切换、同步、线程阻塞操作等也都会带来开销。