

1. 纲要

- a) 多线程的基本概念
- b) 线程的创建和启动
- c) 线程的生命周期
- d) 线程的调度
- e) 线程控制
- f) 线程的同步
- g) 守护线程
- h) 定时器的使用
- i) windows 的任务计划

2. 内容

1.1 多线程的基本概念

线程指进程中的一个执行场景，也就是执行流程，那么进程和线程有什么区别呢？

- 每个进程是一个应用程序，都有独立的内存空间
- 同一个进程中的线程共享其进程中的内存和资源

（共享的内存是堆内存和方法区内存，栈内存不共享，每个线程有自己的。）

1.什么是进程？

一个进程就是一个应用程序。在操作系统中每启动一个应用程序就会相应的启动一个进程。例如：千千静听进程，魔兽进程，Word 进程，QQ 进程，JVM 启动对应一个进程。

2.系统引入多进程的作用？

最初的计算机是“单进程的”，计算机只能运行一个应用程序，例如第一台计算机只有DOS 窗口。现代的计算机可以满足我们一边听音乐，一边玩游戏。现代的计算给我们人类感觉：多件事情一起运行。感觉是并行的（错觉）。

对于单核的计算机来讲，在某一个时间点上只能做一件事情，但是由于计算机的处理速度很高，多个进程之间完成频繁的切换执行，这个切换速度使人类产生了错觉，人类的错觉是：

多个进程在同时运行。

计算机引入多进程的作用：提高 **CPU 的使用率**。

重点：进程和进程之间的内存独立。

3.什么是线程？

线程是进程的一个执行场景。一个进程可以启动多个线程。

4.进程引入多线程的作用？

提高进程的使用率。

重点：线程和线程之间栈内存独立，堆内存和方法区内存共享。一个线程一个栈。

5.描述 java 程序的执行原理：

java 命令执行会启动 JVM，JVM 的启动表示启动一个应用程序，表示启动了一个进程。该进程会自动启动一个“主线程”，然后主线程负责调用某个类的 main 方法。所以 main 方法的执行是在主线程中执行的。然后通过 main 方法代码的执行可以启动其他的“分支线程”。所以，main 方法结束程序不一定结束，因为其他的分支线程有可能还在执行。

1.2 线程的创建和启动

Java 虚拟机的**主线程**入口是 main 方法，用户可以自己创建线程，创建方式有**两种**：

- 继承 Thread 类
- 实现 Runnable 接口（**推荐使用 Runnable 接口**）

1.2.1 继承 Thread 类

Thread 类中创建线程最重要的两个方法为：

```
public void run()
```

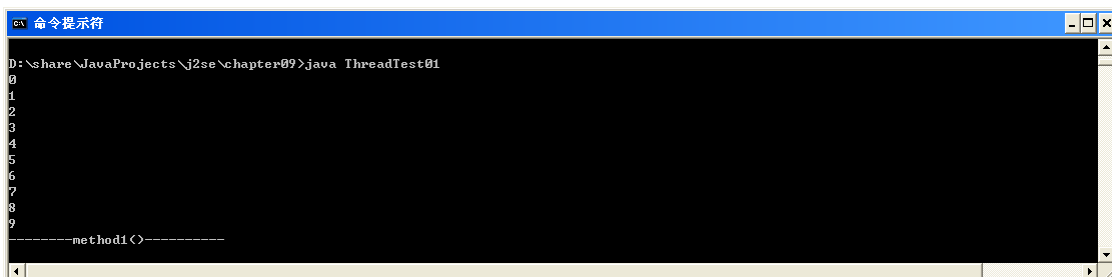
```
public void start()
```

采用 Thread 类创建线程，用户只需要继承 Thread，覆盖 Thread 中的 run 方法，父类 Thread 中的 run 方法没有抛出异常，那么子类也不能抛出异常，最后采用 start 启动线程即可

【示例代码】，不使用线程

```
public class ThreadTest01 {
```

```
public static void main(String[] args) {  
    Processor p = new Processor();  
    p.run();  
  
    method1();  
}  
  
private static void method1() {  
    System.out.println("-----method1()-----");  
}  
}  
  
class Processor {  
  
    public void run() {  
        for (int i=0; i<10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```



```
命令提示符  
D:\share\JavaProjects\j2se\chapter09>java ThreadTest01  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
-----method1()-----
```

以上顺序输出相应的结果（属于串行），也就是 run 方法完全执行完成后，才执行 method1 方法，也就是 method1 必须等待前面的方法返回才可以得到执行，这是一种“**同步编程模型**”

【代码示例】，使用线程

```
public class ThreadTest02 {

    public static void main(String[] args) {
        Processor p = new Processor();

        //手动调用该方法
        //不能采用 run 来启动一个场景（线程），
        //run 就是一个普通方法调用
        //p.run();

        //采用 start 启动线程，不是直接调用 run
        //start 不是马上执行线程，而是使线程进入就绪
        //线程的真正执行是由 Java 的线程调度机制完成的
        p.start();

        //只能启动一次
        //p.start();

        method1();
    }

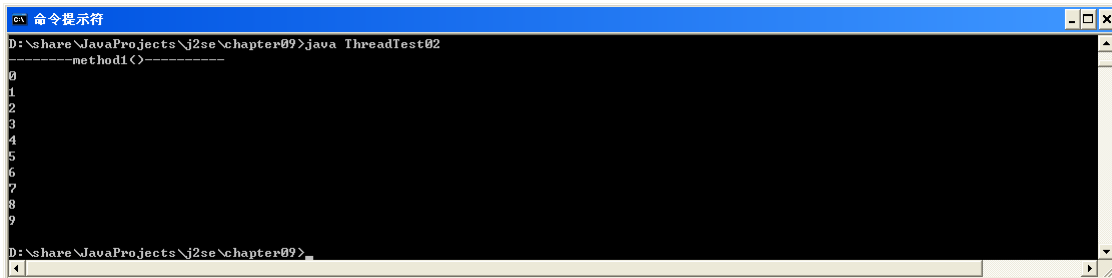
    private static void method1() {
        System.out.println("-----method1()-----");
    }
}

class Processor extends Thread {

    //覆盖 Thread 中的 run 方法，该方法没有异常
    //该方法是由 java 线程掉机制调用的
```

//我们不应该手动调用该方法

```
public void run() {  
    for (int i=0;i<10;i++) {  
        System.out.println(i);  
    }  
}
```



通过输出结果大家会看到，没有顺序执行，而在输出数字的同时执行了 `method1()` 方法，如果从效率上看，采用多线程的示例要快些，因为我们可以看作他是同时执行的，`method1()` 方法没有等待前面的操作完成才执行，这叫“**异步编程模型**”

1.2.2 实现 Runnable 接口

其实 `Thread` 对象本身就实现了 `Runnable` 接口，但一般建议直接使用 `Runnable` 接口来写多线程程序，因为接口会比类带来更多的好处

【示例代码】

```
public class ThreadTest03 {  
  
    public static void main(String[] args) {  
        //Processor r1 = new Processor();  
        Runnable r1 = new Processor();  
        //不能直接调用 run  
        //p.run();  
  
        Thread t1 = new Thread(r1);
```

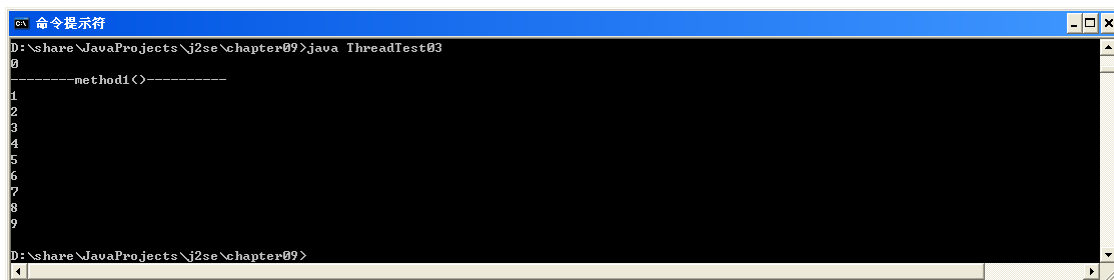
```
//启动线程
t1.start();

method1();
}

private static void method1() {
    System.out.println("-----method1()-----");
}
}

//实现 Runnable 接口
class Processor implements Runnable {

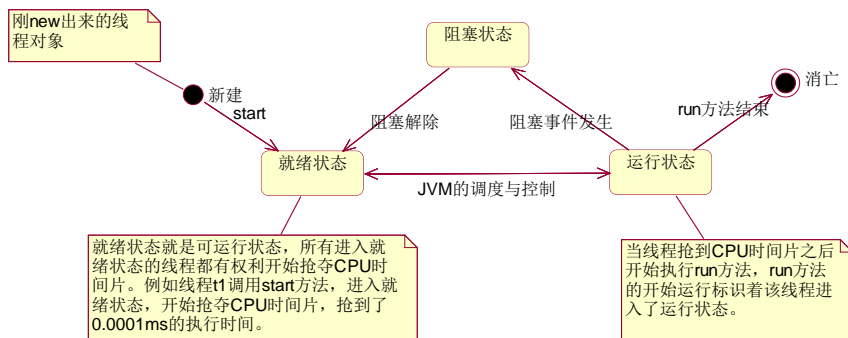
    //实现 Runnable 中的 run 方法
    public void run() {
        for (int i=0;i<10;i++) {
            System.out.println(i);
        }
    }
}
```



```
命令提示符
D:\Share\JavaProjects\j2se\chapter09>java ThreadTest03
0
-----method1()-----
1
2
3
4
5
6
7
8
9
D:\Share\JavaProjects\j2se\chapter09>
```

1.3 线程的生命周期

线程的生命周期存在五个状态：新建、就绪、运行、阻塞、死亡



新建：采用 new 语句创建完成

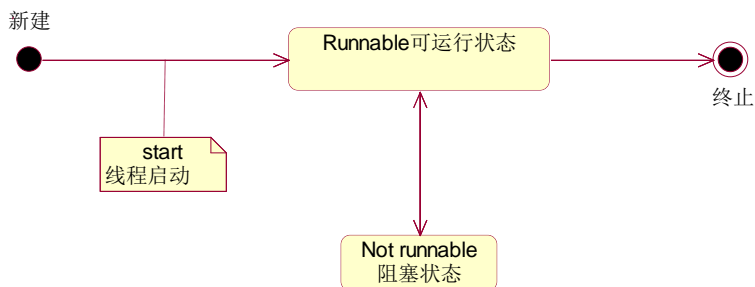
就绪：执行 start 后

运行：占用 CPU 时间

阻塞：执行了 wait 语句、执行了 sleep 语句和等待某个对象锁，等待输入的场所

终止：退出 run()方法

有的书籍上对线程的生命周期是做如下定义的。



1.4 线程的调度与控制

通常我们的计算机只有一个 CPU，CPU 在某一个时刻只能执行一条指令，线程只有得到 CPU 时间片，也就是使用权，才可以执行指令。在单 CPU 的机器上线程不是并行运行的，只有在多个 CPU 上线程才可以并行运行。Java 虚拟机要负责线程的调度，取得 CPU 的使用权，目前

有两种调度模型：分时调度模型和抢占式调度模型，Java 使用抢占式调度模型。

分时调度模型：所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间片

抢占式调度模型：优先级高的线程获取 CPU 的时间片相对多一些，如果线程的优先级相同，那么会随机选择一个

1.4.1 线程优先级

线程优先级主要分三种：MAX_PRIORITY(最高级);MIN_PRIORITY（最低级）
NOM_PRIORITY(标准)默认

```
public class ThreadTest04 {

    public static void main(String[] args) {

        Runnable r1 = new Processor();

        Thread t1 = new Thread(r1, "t1");

        //设置线程的优先级，线程启动后不能再次设置优先级
        //必须在启动前设置优先级
        //设置最高优先级
        t1.setPriority(Thread.MAX_PRIORITY);
        //启动线程
        t1.start();

        //取得线程名称
        //System.out.println(t1.getName());

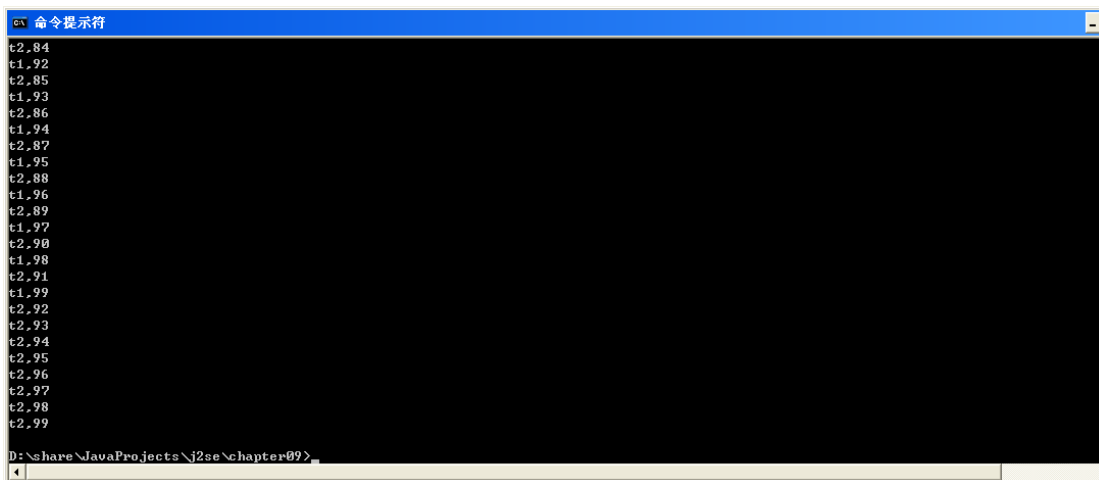
        Thread t2 = new Thread(r1, "t2");
        //设置最低优先级
        t2.setPriority(Thread.MIN_PRIORITY);
        t2.start();
    }
}
```



```
        System.out.println(Thread.currentThread().getName());
    }
}

class Processor implements Runnable {

    public void run() {
        for (int i=0;i<100;i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
        }
    }
}
```



```
命令提示符
t2,84
t1,92
t2,85
t1,93
t2,86
t1,94
t2,87
t1,95
t2,88
t1,96
t2,89
t1,97
t2,90
t1,98
t2,91
t1,99
t2,92
t2,93
t2,94
t2,95
t2,96
t2,97
t2,98
t2,99
D:\share\JavaProjects\j2se\chapter09>
```

从以上输出结果应该看可以看出，优先级高的线程（t1）会得到的 CPU 时间多一些，优先执行完成

1.4.2 Thread.sleep

sleep 设置休眠的时间,单位毫秒，当一个线程遇到 sleep 的时候，就会睡眠，进入到阻塞状态，放弃 CPU，腾出 cpu 时间片，给其他线程用，所以在开发中通常我们会这样做，使其他的线程能够取得 CPU 时间片，当睡眠时间到达了，线程会进入可运行状态，得到 CPU 时间片继续执行，如果线程在睡眠状态被中断了，将会抛出 InterruptedException

【示例代码】

```
public class ThreadTest05 {

    public static void main(String[] args) {
        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        Thread t2 = new Thread(r1, "t2");
        t2.start();
    }
}

class Processor implements Runnable {

    public void run() {
        for (int i=0; i<100; i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
            if (i % 10 == 0) {
                try {
                    //睡眠 100 毫秒，主要是放弃 CPU 的使用，将 CPU 时间片
                    //交给其他线程使用
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

1.4.3 Thread.yield、

它与 sleep()类似，只是不能由用户指定暂停多长时间，并且 yield()方法只能让同优先级的线程有执行的机会

```
public class ThreadTest06 {

    public static void main(String[] args) {

        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        Thread t2 = new Thread(r1, "t2");
        t2.start();
    }
}

class Processor implements Runnable {

    public void run() {
        for (int i=0; i<100; i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
            if (i % 10 == 0) {
                System.out.println("-----");
                //采用 yield 可以将 CPU 的使用权让给同一个优先级的线程
                Thread.yield();
            }
        }
    }
}
```

1.4.4 t.join();

当前线程可以调用另一个线程的 join 方法，调用后当前线程会被阻塞不再执行，直到被调用的线程执行完毕，当前线程才会执行

```
public class ThreadTest07 {

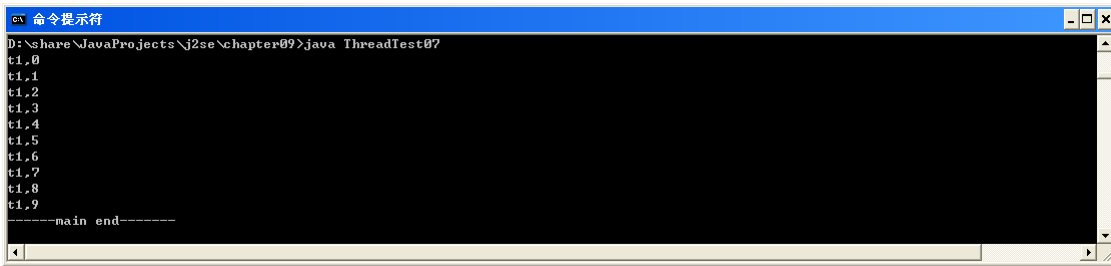
    public static void main(String[] args) {

        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("-----main end-----");
    }
}

class Processor implements Runnable {

    public void run() {
        for (int i=0;i<10;i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
        }
    }
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter09>java ThreadTest07
t1.0
t1.1
t1.2
t1.3
t1.4
t1.5
t1.6
t1.7
t1.8
t1.9
-----main end-----
```

1.4.5 interrupt（中断）

如果我们的线程正在睡眠，可以采用 interrupt 进行中断

```
public class ThreadTest08 {

    public static void main(String[] args) {
        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        try {
            //设置为 500 毫秒，没有出现中断异常，因为
            //500 毫秒之后再次调用 t1.interrupt()时，
            //此时的睡眠线程已经执行完成
            //如果 sleep 的时间设置的小一些，会出现中断异常，
            //因为存在睡眠线程
            Thread.sleep(500);
        } catch (Exception e) {
            e.printStackTrace();
        }
        //中断睡眠中的线程
        t1.interrupt();
    }
}
```

```
class Processor implements Runnable {

    public void run() {
        for (int i=1;i<100;i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
            if (i % 50 == 0) {
                try {
                    Thread.sleep(200);
                } catch (Exception e) {
                    System.out.println("-----中断-----");
                    break;
                }
            }
        }
    }
}
```

1.4.6 如何正确的停止一个线程

通常定义一个标记，来判断标记的状态停止线程的执行

```
public class ThreadTest09 {

    public static void main(String[] args) {
        //Runnable r1 = new Processor();
        Processor r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        try {
```

```
        Thread.sleep(20);
    } catch (Exception e) {}

    //停止线程
    r1.setFlag(true);
}

}

class Processor implements Runnable {

    //线程停止标记， true 为停止
    private boolean flag;

    public void run() {
        for (int i=1; i<100; i++) {
            System.out.println(Thread.currentThread().getName() + "," + i);
            //为 true 停止线程执行
            if (flag) {
                break;
            }
        }
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }
}
```

1.5 线程的同步（加锁）

1.5.1 为什么需要同步

【示例代码】，取得 0~10 的和采用相同的线程对象启用两个线程进行计算（共享一个对象启动两个线程）

```
public class ThreadTest10 {

    public static void main(String[] args) {
        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

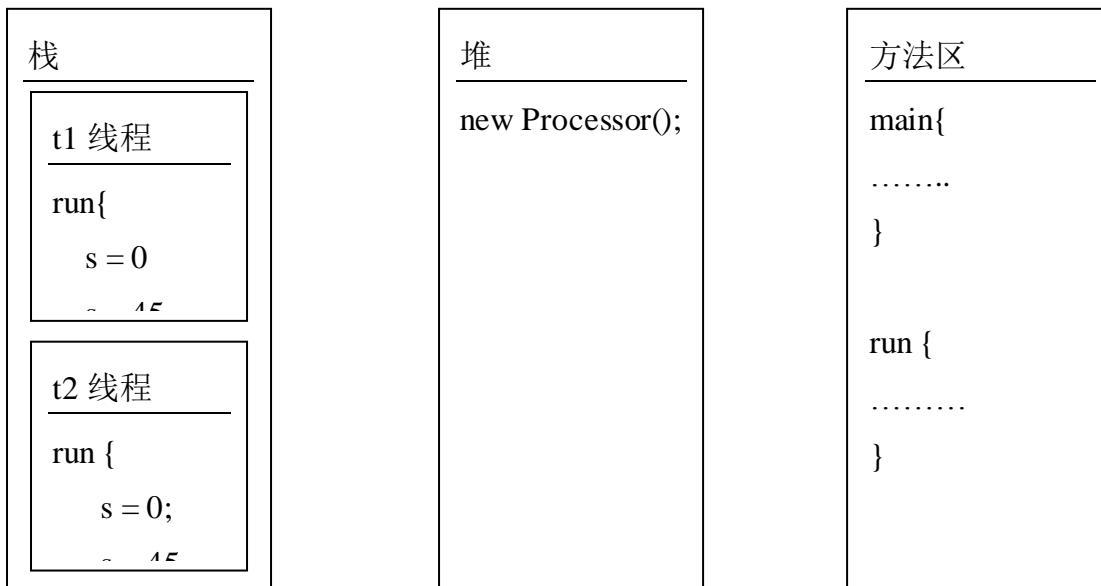
        Thread t2 = new Thread(r1, "t2");
        t2.start();
    }
}

class Processor implements Runnable {

    public void run() {
        //定义局部变量 s，作为累加变量
        int s = 0;
        for (int i=0; i<10; i++) {
            s+=i;
        }
        System.out.println(Thread.currentThread().getName() + ", s=" + s);
    }
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter09>java ThreadTest10
t1, s=45
t2, s=45
D:\share\JavaProjects\j2se\chapter09>
```



以上 t1 和 t2 并发执行，s 为每个线程的局部变量，位于各自的栈帧中，因为栈帧中的数据是不会互相干扰的，所有计算结果都为 45

【示例代码】，取得 0~10 的和采用两个线程进行计算，将 s 改为成员变量(共享一个对象启动两个线程)

```
public class ThreadTest11 {

    public static void main(String[] args) {
        Runnable r1 = new Processor();
```

```
Thread t1 = new Thread(r1, "t1");
t1.start();

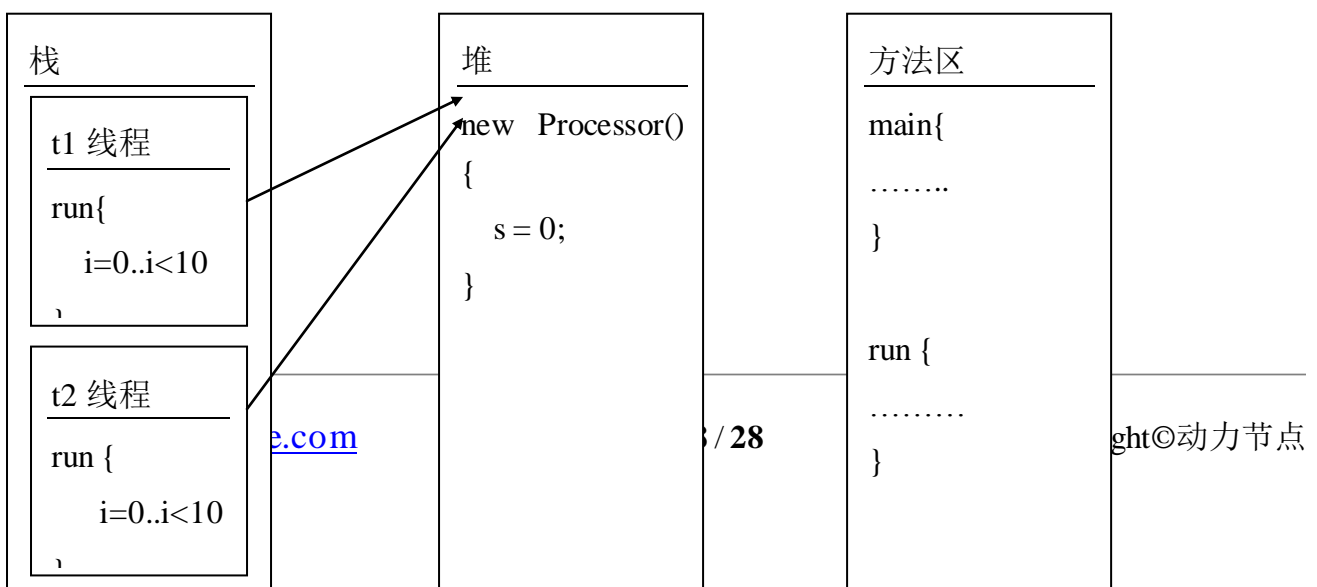
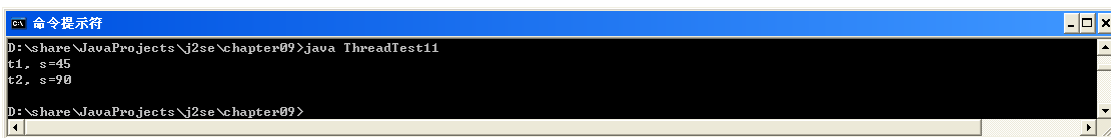
Thread t2 = new Thread(r1, "t2");
t2.start();

}

}

class Processor implements Runnable {
    //定义成员变量 s，作为累加变量
    private int s = 0;

    public void run() {
        for (int i=0; i<10; i++) {
            s+=i;
        }
        System.out.println(Thread.currentThread().getName() + ", s=" + s);
    }
}
```



为什么出现以上的问题，因为共享了同一个对象的成员变量 s，两个线程同时对其进行操作，所以产生了问题，此时称为此时 **Processor** 为线程不安全的，如果想得到正确的结果，必须采用线程同步，加锁，该变量不能共享使用

1.5.2 使用线程同步

线程同步，指某一个时刻，指允许一个线程来访问**共享资源**，线程同步其实是对**对象**加锁，如果对象中的方法都是同步方法，那么某一时刻只能执行一个方法，采用线程同步解决以上的问题，我们只要保证线程一操作 s 时，线程 2 不允许操作即可，只有线程一使用完成 s 后，再让线程二来使用 s 变量

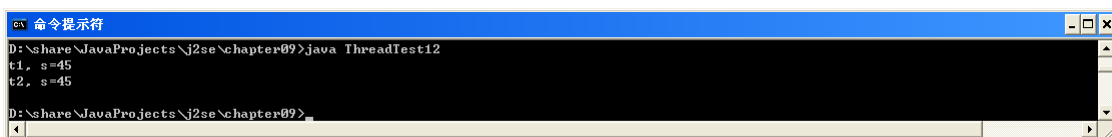
```
public class ThreadTest12 {  
  
    public static void main(String[] args) {  
        Runnable r1 = new Processor();  
        Thread t1 = new Thread(r1, "t1");  
        t1.start();  
  
        Thread t2 = new Thread(r1, "t2");  
        t2.start();  
  
    }  
}
```

```
}

class Processor implements Runnable {
    //定义成员变量 s，作为累加变量
    private int s = 0;

    //synchronized 是对对象加锁
    //采用 synchronized 同步最好只同步有线程安全的代码
    //可以优先考虑使用 synchronized 同步块
    //因为同步的代码越多，执行的时间就会越长，其他线程等待的时间就会越长
    //影响效率
    //public synchronized void run() {

        //使用同步块
        synchronized (this) {
            for (int i=0;i<10;i++) {
                s+=i;
            }
            System.out.println(Thread.currentThread().getName() + ", s=" + s);
            s = 0;
        }
        //.....
        //.....
    }
}
```



```
命令提示符
D:\Share\JavaProjects\j2se\chapter09>java ThreadTest12
t1, s=45
t2, s=45
D:\Share\JavaProjects\j2se\chapter09>
```

执行正确

以上示例，如果不采用线程同步如何解决？可以让每个线程创建一个对象，这样在堆中就不会出现对象的**状态共享**了，从而可以避免**线程安全**问题

1.5.3 为每一个线程创建一个对象来解决线程安全问题

```
public class ThreadTest13 {

    public static void main(String[] args) {

        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        //再次创建 Processor 对象
        //每个线程拥有自己的对象
        Runnable r2 = new Processor();
        Thread t2 = new Thread(r2, "t2");
        t2.start();

    }
}

class Processor implements Runnable {

    //定义成员变量 s，作为累加变量
    private int s = 0;

    public void run() {
        for (int i=0; i<10; i++) {
            s+=i;
        }
    }
}
```

```
System.out.println(Thread.currentThread().getName() + ", s=" + s);

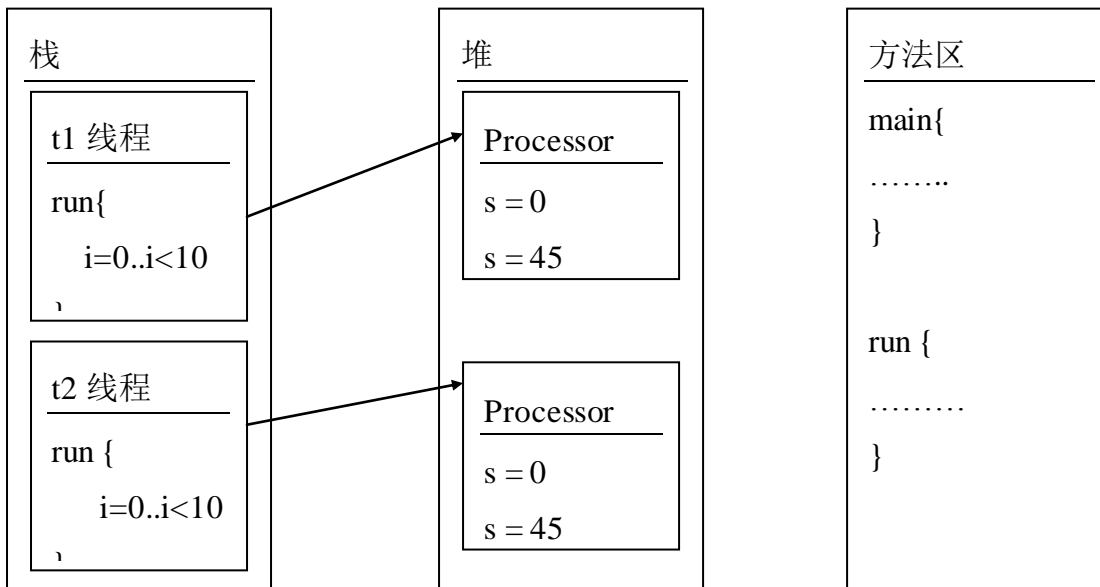
s = 0;

}

}
```

```
命令提示符
D:\share\JavaProjects\j2se\chapter09>java ThreadTest13
t1, s=45
t2, s=45
D:\share\JavaProjects\j2se\chapter09>
```

以上输出完全正确，每个线程操作的是自己的对象，没有操作共享的资源



1.6 守护线程

从线程分类上可以分为：**用户线程**（以上讲的都是用户线程），另一个是**守护线程**。守护线程是这样的，所有的用户线程结束生命周期，守护线程才会结束生命周期，只要有一个用户线程存在，那么守护线程就不会结束，例如 java 中著名的**垃圾回收器**就是一个守护线程，只有应用程序中所有的线程结束，它才会结束。

1.6.1 用户线程

```
public class DaemonThreadTest01 {

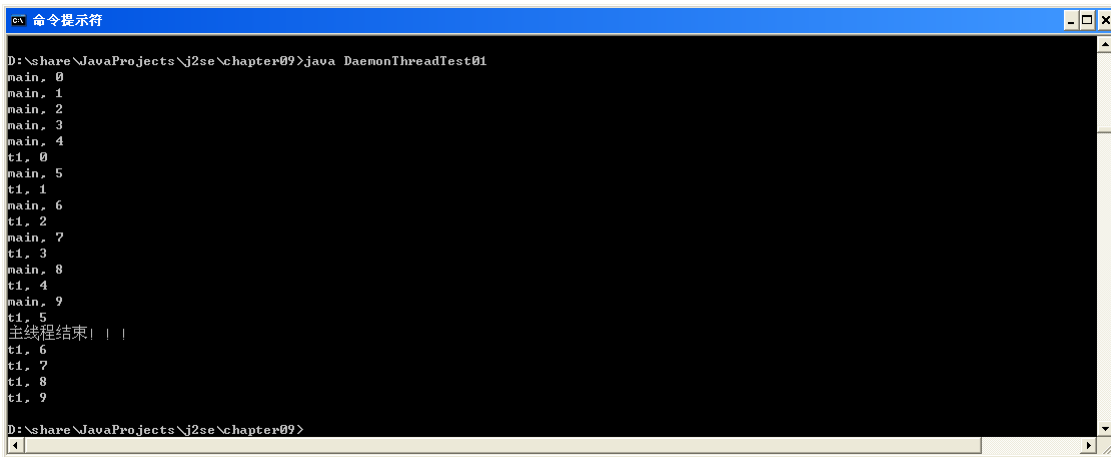
    public static void main(String[] args) {
        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");
        t1.start();

        for (int i=0;i<10;i++) {
            System.out.println(Thread.currentThread().getName() + ", " + i);
        }

        System.out.println("主线程结束!!!");
    }
}

class Processor implements Runnable {

    public void run() {
        for (int i=0;i<10;i++) {
            System.out.println(Thread.currentThread().getName() + ", " + i);
        }
    }
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter09>java DaemonThreadTest01
main. 0
main. 1
main. 2
main. 3
main. 4
t1. 0
main. 5
t1. 1
main. 6
t1. 2
main. 7
t1. 3
main. 8
t1. 4
main. 9
t1. 5
主线程结束!!!
t1. 6
t1. 7
t1. 8
t1. 9
D:\share\JavaProjects\j2se\chapter09>
```

以上可以看出，主线程执行结束了，但用户线程仍然将数据打印出来了

1.6.2 修改为守护（服务线程）线程

```
public class DaemonThreadTest02 {

    public static void main(String[] args) {

        Runnable r1 = new Processor();
        Thread t1 = new Thread(r1, "t1");

        //将当前线程修改为守护线程
        //在线程没有启动时可以修改以下参数
        t1.setDaemon(true);

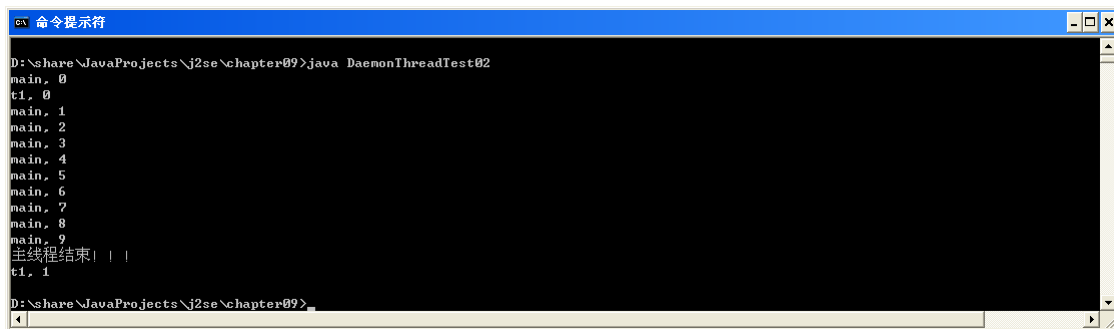
        t1.start();

        for (int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + ", " + i);
        }

        System.out.println("主线程结束!!!");
    }
}
```



```
}  
  
class Processor implements Runnable {  
  
    public void run() {  
        for (int i=0;i<10;i++) {  
            System.out.println(Thread.currentThread().getName() + ", " + i);  
        }  
    }  
}
```



```
命令提示符  
D:\Share\JavaProjects\j2se\chapter09>java DaemonThreadTest02  
main, 0  
t1, 0  
main, 1  
main, 2  
main, 3  
main, 4  
main, 5  
main, 6  
main, 6  
main, 7  
main, 8  
main, 9  
主线程结束 | |  
t1, 1  
D:\Share\JavaProjects\j2se\chapter09>
```

设置为守护线程后，当主线程结束后，守护线程并没有把所有的数据输出完就结束了，也即是说守护线程是为**用户线程服务的**，**当用户线程全部结束，守护线程会自动结束**

1.7 Timer 定时器

```
import java.util.*;  
import java.text.*;  
  
public class TimerTest01 {  
  
    public static void main(String[] args) throws Exception{  
        Timer t = new Timer();  
        Date    firsDate    =    new    SimpleDateFormat("yyyy-MM-dd  
HH:mm:ss").parse("2010-02-20 15:23:10");
```

```
t.schedule(new MyTimerTask(), firsDate, 1000*60*60*24);
    }
}

class MyTimerTask extends TimerTask {

    public void run() {
        System.out.println(new Date());
    }
}
```

以上程序在 2010-02-20 15:23:10 会输出，每个 24 小时输出一次

【代码示例】，采用匿名类实现以上功能

```
import java.util.*;
import java.text.*;

public class TimerTest02 {

    public static void main(String[] args) throws Exception{
        Timer t = new Timer();
        Date    firsDate    =    new    SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").parse("2010-02-20 15:29:10");
        t.schedule(new TimerTask() {
            public void run() {
                System.out.println(new Date());
            }
        },firsDate, 1000*2);
    }
}
```

以上程序在 2010-02-20 15:29:10 会输出，每个 2 秒钟输出一次

关于日程有专门的第三方开源产品，如： [Quartz](#)

1.8 window 定时器（任务）

1.9 重点掌握

1. 进程与线程的概念
2. 线程的两种实现方式（Thread,Runnable）
3. 了解线程的优先级
4. sleep 的含义
5. 如果正确的结束一个线程
6. 线程同步的含义（同步共享资源，局部变量不存在共享的问题）
7. 守护线程的概念
8. 了解 Timer
9. 了解 winodw 提供的计划

