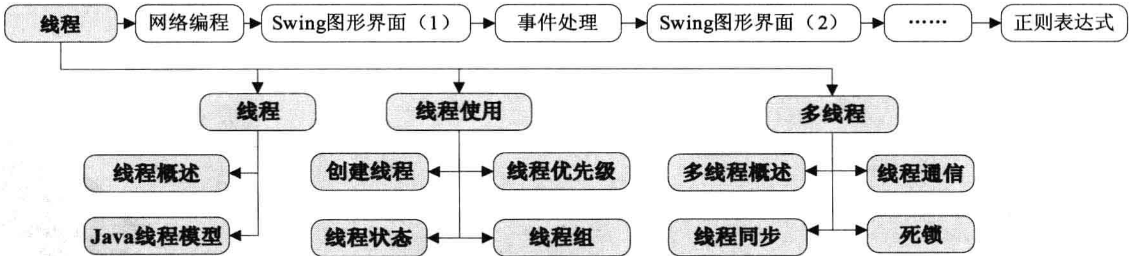
理论篇

第 1 章　线　　程

本章目标

- 理解线程的基本概念
- 理解 Java 的线程模型
- 掌握 Java 线程的状态和状态转换
- 掌握线程的创建和使用
- 掌握线程优先级的使用
- 掌握线程组的使用
- 理解多线程的概念
- 掌握 Java 的多线程实现
- 掌握线程的同步技巧
- 掌握线程的通信方式
- 理解死锁的概念

学习导航



任务描述

【描述 1.D.1】

通过 Thread 类获取程序的主线程。

【描述 1.D.2】

通过创建 Thread 类的子类，演示线程的创建。

【描述 1.D.3】

通过实现 Runnable 接口，演示线程的创建。

【描述 1.D.4】

演示线程的创建、运行和停止三个状态之间的切换。

【描述 1.D.5】

利用线程的 sleep 状态实现周期性打印信息。

【描述 1.D.6】

利用线程的 interrupt 方法实现线程中断。

【描述 1.D.7】

演示线程优先级的设置及使用。

【描述 1.D.8】

演示线程组的创建和使用。

【描述 1.D.9】

一个简单的多线程例子，能根据用户提供的字符，打印其后连续的 30 个字符。

【描述 1.D.10】

通过多线程，演示不使用同步机制可能出现的情况。

【描述 1.D.11】

通过生产/消费模型，演示线程通信机制的应用。

【描述 1.D.12】

使用资源竞争模型，演示死锁的产生。

1.1 线程基础

线程（Thread）是独立于其他线程运行的程序执行单元。在 Java 体系中，线程在多任务处理中起着举足轻重的作用。

1.1.1 线程概述

线程（轻量级程序）类似于一个程序，也有开始、执行和结束。它是运行在程序内部的一个比进程还要小的单元。使用线程的主要原因在于可以在一个程序中同时运行多个任务。每个 Java 程序都至少有一个线程——主线程。当一个 Java 程序启动时，JVM 会创建主线程，并在该线程中调用程序的 main()方法。

多线程就是同时有多个线程在执行。在多 CPU 的计算机中，多线程的实现是真正的物理上的同时执行，而对于单 CPU 的计算机而言，实现的只是逻辑上的同时执行。在每个时刻，真正执行的只有一个线程，由操作系统进行线程管理调度，但由于 CPU 的速度很快，让人感到像是多个线程在同时执行。

进程是指一种“自包容”的运行程序，有自己的地址空间；线程是进程内部单一的一个顺序控制流。基于进程的特点是允许计算机同时运行两个或更多的程序。基于线程的多任务处理环境中，线程是最小的处理单位。多线程程序在更低的层次中引入多任务处理。

多进程与多线程是多任务的两种类型。多线程与多进程的主要区别在于，线程是一个进程中一段独立的控制流，一个进程可以拥有若干个线程。在多进程设计中各个进程之间的数据块是相互独立的，一般彼此不影响，要通过信号、管道等进行交流。而在多线程设计中，各个线程不一定独立，同一任务中的各个线程共享程序段、数据段等资源。

多线程比多进程更方便于共享资源，而 Java 提供的同步机制解决了线程之间的数据完整性问题，使得多线程设计更易发挥作用。在 Java 程序设计中，动画设计及多媒体应用都会广泛地使用到多线程。

引入线程的优点是：

- 充分利用 CPU 资源。
- 简化编程模型。
- 简化异步事件处理。
- 使 GUI 更有效率。
- 节约成本。

1.1.2 Java 线程模型

Java 的线程模型是面向对象的。在 Java 中建立线程有两种方法：一种是继承 Thread 类；另一种是实现 Runnable 接口，并通过 Thread 和实现 Runnable 的类来建立线程。

Java 通过 Thread 类将线程所必需的功能都封装了起来。要想建立一个线程，必须要有一

个线程执行函数，这个线程执行函数对应 Thread 类的 run()方法；Thread 类还有一个 start()方法，这个方法负责启动线程，当调用 start()方法后，如果线程启动成功，将自动调用 Thread 类的 run()方法。因此，任何继承 Thread 的 Java 类都可以通过 Thread 类的 start()方法来建立线程。如果想运行自己的线程执行函数，那就要重写 Thread 类的 run()方法。

Java 线程模型中还提供了一个标识某个 Java 类是否可作为线程类的接口——Runnable，该接口只有一个抽象方法 run()，也就是 Java 线程模型的线程执行函数。

这两种方法从本质上说是一致的，即都是通过 Thread 类来建立线程，并运行 run()方法的。但它们的区别在于：由于 Java 不支持多继承，因此，这个线程类如果继承了 Thread，就不能再继承其他的类了，而通过实现 Runnable 接口的方法来建立线程，这样的线程类可以在必要的时候继承和业务有关的类，形成清晰的数据模型。

1.2　线程使用

在 Java 中创建线程有几种方法。每个 Java 程序至少包含一个线程：主线程。其他线程都是通过 Thread 构造器或实例化继承类 Thread 的类来创建的。

下述代码用于实现任务描述 1.D.1，通过 Thread 类获取程序的主线程。

【描述 1.D.1】 MainThread.java

```
class MainThread {
    public static void main(String args[]) {
        // 调用 Thread 类的 currentThread()方法获取当前线程
        Thread t = Thread.currentThread();
        System.out.println("主线程是: " + t);
    }
}
```

1.2.1　创建线程

在 Java 中创建线程有两种方法：使用 Thread 类和使用 Runnable 接口。在使用 Runnable 接口时需要建立一个 Thread 实例。因此，无论是通过 Thread 类还是 Runnable 接口建立线程，都必须建立 Thread 类或它的子类的实例。

1. Thread

Java 线程可以通过直接实例化 Thread 对象或实例化继承 Thread 的对象来创建其他线程。Thread 子类需要重写 run 方法，并在 run 方法内部定义线程的功能语句，其常用方法及使用说明见表 1-1。

试读结束：需要全本请在线购买： www.ertongbook.com

表 1-1　Thread 类的方法列表

方法名	功能说明
Thread()	构造默认的线程对象
Thread(Runnable target)	使用传递的 Runnable 构造线程对象
Thread(Runnable target,String name)	使用传递的 Runnable 构造名为 name 的线程对象
Thread(ThreadGroup group,Runnable target, String name)	使用传递的 Runnable 在 group 线程组内构造名为 name 的线程对象
final String getName()	返回线程的名称
final boolean isAlive()	如果线程是激活的，则返回 true
final void setName(String name)	将线程的名称设置为由 name 指定的名称
set\getPriority()	设置得到线程优先级
final void join()	等待线程结束
static void sleep(long millis)	用于将线程挂起一段时间，单位毫秒
void start()	调用 run( )方法启动线程，开始线程的执行
void stop()	停止线程，已经不建议使用
void interrput()	中断线程
static int activeCount()	返回激活的线程数
static void yield()	使正在执行的线程临时暂停，并允许其他线程执行

下述代码用于实现任务描述 1.D.2，通过创建 Thread 类的子类，演示线程的创建。

【描述 1.D.2】 ThreadDemo.java

```
class Thread1 extends Thread {
    public void run() {
        // 获取当前线程的名字
        System.out.println(Thread.currentThread().getName());
    }
}
class Thread2 extends Thread {
    public Thread2(String name){
        super(name);
    }
    public void run() {
        // 获取当前线程的名字
        System.out.println(Thread.currentThread().getName());
    }
}
public class ThreadDemo {
    public static void main(String[] args) {
        Thread1 thread1 = new Thread1();
        // 构造名为 thread2 的线程对象
        Thread2 thread2 = new Thread2("thread2");
```

```
        thread1.start();
        thread2.start();
        // 获取主线程的名字
        System.out.println("["+Thread.currentThread().getName()+"]");
    }
}
```

可能的执行结果：

```
Thread-0
[main]
thread2
```

上述代码定义了两个线程 Thread1 和 Thread2，在创建 thread1 对象时并未指定线程名，因此，所输出的线程名是系统的默认值。而对于输出结果，不仅不同机器之间的结果可能不同，而且在同一机器上多次运行同一程序也可能生成不同结果。另外，线程的执行次序是不定的，除非使用同步机制以强制按特定的顺序执行。

注意 Thread 类的 start 方法不能多次调用，如不能调用两次 thread1.start()方法，否则会抛出一个 IllegalThreadStateException 异常。

2. Runnable

创建线程的另一种方式是实现 Runnable 接口。实现 Runnable 接口的类必须使用 Thread 类的实例才能创建线程。通过 Runnable 接口创建线程的步骤如下：

01 实例化实现 Runnable 接口的类。

02 建立一个 Thread 对象，并将第一步实例化后的对象作为参数传入 Thread 类的构造方法。

03 通过 Thread 类的 start()方法建立线程。

下述代码用于实现任务描述 1.D.3，通过实现 Runnable 接口，演示线程的创建。

【描述 1.D.3】 RunnableDemo.java

```
class Thread3 implements Runnable {
    // 重写 run 方法
    public void run() {
        // 获取当前线程的名字
        System.out.println(Thread.currentThread().getName());
        for (int i=0;i<30;i++){
            System.out.print("A");
        }
    }
}
```

```
public class RunnableDemo {
    public static void main(String[] args) {
        Thread3 th3 = new Thread3();
        // 将实现 Runnable 的类的实例传入构造函数
        Thread thread = new Thread(th3);
        thread.start();
        // 获取主线程的名字
        System.out.println("["+Thread.currentThread().getName()+"]");
        for (int i=0;i<30;i++){
            System.out.print("C");
        }
    }
}
```

可能的执行结果：

```
Thread-0
AAAAAAAAAAAAAAAAAA[main]
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAA
```

注意 不要直接调用 Thread 类或 Runnable 对象的 run()方法启动，应该通过 Thread 的 start()方法启动线程。

1.2.2 线程状态

线程的状态分为 7 种：born（新生状态）、runnable（就绪状态）、running（运行状态）、waiting（等待状态）、sleeping（睡眠状态）、blocked（阻塞状态）和 dead（死亡状态）。图 1-1 展示了线程的状态和状态之间的转换。

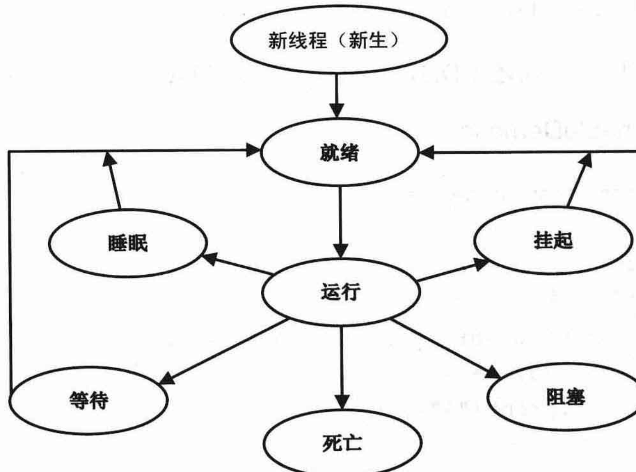


图 1-1 线程状态之间的转换

1. born

当使用 new 来新建一个线程时，一个新的线程就诞生了。除非在构造函数中调用了 start()方法，否则这个新线程将作为一个新对象呆在内存中基本上什么事情也不做。当在方法中对这个线程调用了 start()方法，或者这个线程在程序的其他任何位置使自己的状态由 born 改变为 runnable 之后，调度程序就可以把处理器分配给这个线程。

线程处于等待状态时，可以通过 Thread 类的方法来设置线程的各种属性，如线程的优先级（setPriority）、线程名（setName）和线程的类型（setDaemon）等。

2. runnable、running

把处理器分配给一个处于 runnable 的线程之后，这个线程的状态就变成了 running。如果一个处于 running 状态的线程能够运行到结束或因某个未捕获异常而使线程终止时，它的状态就会变为 dead。否则，这个线程的命运就取决于当前是否还有其他线程等待处理器运行。如果这个线程在当前所有处于 runnable 状态的线程中具有最高优先级，那么它就会继续执行，除非这个线程被运行程序的平台划分为时间片。当线程被划分为时间片时，一个处于 running 状态的线程将分配到一个固定间隔的处理器时间。具有相同优先级的线程的时间片调度将导致这些线程被轮流执行。如果正在执行的线程的代码包含了 yield()方法，那么这个处于 running 状态的线程将停止运行，并把处理器交给其他进程。

可以通过 Thread 类的 isAlive()方法来判断线程是否处于运行状态。当线程处于运行状态时，isAlive()返回 true，当 isAlive()返回 false 时，线程可能处于等待状态，也可能处于停止状态。

下述代码用于实现任务描述 1.D.4，演示线程的创建、运行和停止三个状态之间的切换。

【描述 1.D.4】 LifeThread1.java

```
class LifeThread1 extends Thread {
    public void run() {
        int i = 0;
        while ((++i) < 1000)
            ;
    }
}
public class LifeCycle1 {
    public static void main(String[] args) throws Exception {
        LifeThread1 thread1 = new LifeThread1();
        System.out.println("等待状态[isAlive: " + thread1.isAlive() + "]");
        thread1.start();
        System.out.println("运行状态[isAlive: " + thread1.isAlive() + "]");
        // 等线程 thread1 结束后再继续执行
        thread1.join();
        System.out.println("线程结束[isAlive: " + thread1.isAlive() + "]");
    }
}
```

执行结果如下：

```
等待状态[isAlive: false]
运行状态[isAlive: true]
线程结束[isAlive: false]
```

3. blocked

在线程试图执行某个不能立即完成的任务，并且该线程必须等待其他任务的完成时才能继续，则该线程进入阻塞状态（blocked）。例如，在线程发出输入/输出请求时，这时操作系统将阻塞该线程的执行，直至完成 I/O 操作，操作完成时，该线程便回到就绪状态，可以再次调度该线程，使其执行。

4. sleeping

在线程执行的过程中，可以通过 sleep 方法使线程暂时停止执行，使其进入 sleep 状态。

下述代码用于实现任务描述 1.D.5，利用线程的 sleep 状态实现周期性打印信息。

【描述 1.D.5】 SleepDemo.java

```
class TimeThread extends Thread {
   // 安全退出控制位
   boolean flag = true;
   public void run() {
      while (flag) {
         System.out.println("TimeThread!");
         try {
            // 休眠 1 秒
            Thread.sleep(1000);
         } catch (Exception e) {
            System.out.println(e);
         }
      }
   }
   // 安全退出控制方法
   public void safeStop() {
      flag = false;
   }
}
public class SleepDemo {
   public static void main(String args[]) {
      TimeThread thread = new TimeThread();
      thread.start();
      try {
         // 主线程等待键盘输入
```

```
            System.in.read();
            // 控制 thread 安全控制位，使线程结束
            thread.safeStop();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

上述代码运行期间会每隔 1 秒钟输出“TimeThread!”，直到用户在主线程环境下按下键盘键。

在使用 sleep 方法时有两点需要注意：

- sleep 方法的参数是毫秒。
- 在使用 sleep 方法时必须使用 throws 或放入 try{...}catch{...}语句。

另外，还可以使用 suspend 和 resume 方法方便地挂起和唤醒线程，但这两个方法可能会造成一些不可预料的事情，已被标识为 deprecated，尽量不要使用这两个方法来操作线程。

注意 如果一个线程包含了很长的循环，在循环的每次迭代之后把这个线程切换到 sleep 状态是一种很好的策略，这可以保证其他线程不必等待很长时间才能轮到处理器执行。

5. waiting

如果某个线程的执行条件还未满足，可以调用 wait 方法使其进入等待状态。一旦线程处于等待状态，在另一个线程对其等待对象调用 notify 或 notifyAll 方法时，则线程回到就绪状态。

如果对某个线程调用 interrupt 方法，则该线程会根据线程状态抛出 InterruptedException 异常，对异常进行处理，可以再次调用该线程。

6. dead

线程在下列情况下会结束：

- 线程运行到 run()方法的结尾。
- 线程抛出一个未捕获异常或 Error，或使用 interrupt()方法中断线程。

如在任务描述 1.D.5 中，线程运行期间可以通过调用 safeStop()方法将线程的 flag 控制位设置为 false，结束 while 循环，从而结束线程。

另一种可控的结束线程的方式就是抛出异常或执行 interrupt()方法。

下述代码用于实现任务描述 1.D.6，利用线程的 interrupt 方法实现线程中断。

试读结束：需要全本请在线购买： www.ertongbook.com

【描述 1.D.6】 InterruptDemo.java

```
class InterruptThread extends Thread {
    public void run() {
        try {
            System.out.println("在 20 秒之内按任意键中断线程!");
            // 延迟 20 秒
            sleep(2000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
public class InterruptDemo {
    public static void main(String[] args) throws Exception {
        Thread thread = new InterruptThread();
        thread.start();
        System.in.read();
        thread.interrupt();
        thread.join();
        System.out.println("线程退出!");
    }
}
```

执行结果如下：

```
在 20 秒之内按任意键中断线程!
sleep interrupted
线程退出!
```

在调用 interrupt()方法后，sleep()方法抛出异常，然后输出错误信息：sleep interrupted。

注意 在 Thread 类中有两个方法可以判断线程是否通过 interrupt 方法被终止。一个是静态的方法 interrupted()，另一个是非静态的方法 isInterrupted()。

1.2.3 线程优先级

线程的优先级代表该线程的重要程度，当有多个线程同时处于可执行状态并等待获得 CPU 时间时，线程调度系统根据各个线程的优先级来决定 CPU 分配时间，优先级高的线程有更大的机会获得 CPU 时间。

每一个线程都有一个优先级。默认情况下优先级通过静态常量 Thread.NORM_PRIORITY 定义，值为 5。每个新线程均继承创建线程的优先级。线程的优先级可以通过 setPriority()/getPriority()方法设置和获取，可将线程的优先级设置为 MIN_PRIORITY（1）和

MAX_PRIORITY（10）之间的值。

下述代码用于实现任务描述 1.D.7，演示线程优先级的设置及使用。

【描述 1.D.7】 PriorityDemo.java

```
class Thread4 extends Thread {
    Thread4(String name) {
        super(name);
    }
    public synchronized void run() {
        System.out.println(this.getName());
        try {
            sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(this.getName() + " 结束!");
    }
}
public class PriorityDemo {
    public static void main(String args[]) throws InterruptedException {
        Thread4 thread1 = new Thread4("first");
        Thread4 thread2 = new Thread4("second");
        Thread4 thread3 = new Thread4("third");
        // 设置优先级为最大,10
        thread1.setPriority(Thread.MAX_PRIORITY);
        thread2.setPriority(Thread.MAX_PRIORITY - 1);
        thread3.setPriority(Thread.MAX_PRIORITY - 2);
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

可能的执行结果：

```
first
second
third
second 结束!
third 结束!
first 结束!
```

当线程睡眠后，优先级就失效了，若想保证优先级继续有效，可考虑使用 join()方法解决。

注意　线程优先级高度依赖于系统。线程优先级不能保证线程的执行次序，应尽量避免使用线程优先级作为构建任务执行顺序的绝对标准。

1.2.4　线程组

某些程序拥有多个线程，如果将它们按照功能归类，可以方便地进行管理。Java 允许创建线程组，实现多线程管理。通过线程组可对其所有线程同时进行操作。例如，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。

Java 的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。

所有线程都隶属于一个线程组，在创建之初，线程被限制到一个系统线程组里，用户也可在创建线程时明确指定组。Java 的 ThreadGroup 类提供了大量的方法，方便对线程组树中的每一个线程组及线程组中的每一个线程进行操作，其常用方法及使用说明见表 1-2。

表 1-2　ThreadGroup 类的方法列表

方法名	功能说明
ThreadGroup(String name)	创建一个新的线程组
int activeCount()	返回线程组中活动线程的数量
int enumerate(Thread []list)	返回所有活动线程的引用
ThreadGroup getParent()	得到这个线程组的父线程组
void interrupt()	中断此线程组和它的子线程组中的所有线程
void list()	打印当前线程组的信息

下述代码用于实现任务描述 1.D.8，演示线程组的创建和使用。

【描述 1.D.8】 ThreadGroupDemo.java

```
class Thread5 extends Thread {
    Thread5(ThreadGroup g, String name) {
        super(g, name);
    }
}
class Thread6 extends Thread5 {
    Thread6(ThreadGroup g, String name) {
        super(g, name);
        // 调用构造函数的同时启动线程
        start();
    }
    public void run() {
        ThreadGroup group = getThreadGroup().getParent().getParent();
        group.list();
```

```
    }
}

public class ThreadGroupDemo {
    public static void main(String[] args) {
        ThreadGroup first = new ThreadGroup("first");
        // 线程组 second 属于线程组 first
        ThreadGroup second = new ThreadGroup(first, "second");
        // 线程组 third 属于线程组 second
        ThreadGroup third = new ThreadGroup(second, "third");
        // 声明线程 one，并使之属于 first 线程组
        Thread one = new Thread5(first, "one");
        // 声明线程 two，并使之属于 third 线程组
        Thread two = new Thread6(third, "two");
    }
}
```

执行结果如下：

```
java.lang.ThreadGroup[name=first,maxpri=10]
    java.lang.ThreadGroup[name=second,maxpri=10]
        java.lang.ThreadGroup[name=third,maxpri=10]
            Thread[two,5,third]
```

1.3 多线程

1.3.1 多线程概述

线程的最主要功能是多任务处理，即多线程。多线程也就是在主线程中有多个线程在运行，多个线程的执行是并发的，在逻辑上“同时”，而不管是否是物理上的“同时”。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、同步等问题是需要重点留意的。

下述代码用于实现任务描述 1.D.9，是一个简单的多线程例子，能根据用户提供的字符，打印其后连续的 30 个字符。

【描述 1.D.9】 MultiThreadDemo.java

```
public class MultiThreadDemo extends Thread {
    // 存储随机字符
    char c;
    public MultiThreadDemo(String name, char c) {
        super(name);
```

```
        this.c = c;
    }
    public void run() {
        int k;
        char ch = c;
        System.out.println();
        System.out.print(getName() + " : ");
        for (k = 0; k <= 30; k++) {
            ch = (char) (c + k);
            System.out.print(ch + " ");
        }
        System.out.println(getName() + " end!");
    }
    public static void main(String args[]) {
        MultiThreadDemo th1 = new MultiThreadDemo("th1", 'A');
        MultiThreadDemo th2 = new MultiThreadDemo("th2", 'a');
        MultiThreadDemo th3 = new MultiThreadDemo("th3", '0');
        // 启动线程
        th1.start();
        th2.start();
        th3.start();
        // 取得当前活动线程的总数
        System.out.println("activecount = " + Thread.activeCount());
    }
}
```

可能的运行结果：

```
activecount = 4
th1 : A B C D
th2 : a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~  th2 end!
E F G
th3 : 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B H I J K L M N O P Q R S T U V W X Y
Z [ \ ] ^ _ th1 end!
C D E F G H I J K L M N th3 end!
```

上述运算结果在每次运行时都会有所不同，也就是说线程的执行是乱序的，所以在使用多线程时要尤其注意。

1.3.2 线程同步

在 Java 中提供了线程同步的概念以保证某个资源在某一时刻只能由一个线程访问，以保证共享数据及操作的完整性。

Java 使用监控器（也称对象锁）实现同步。每个对象都有一个监控器，使用监控器可以

保证一次只允许一个线程执行对象的同步语句。即在对象的同步语句执行完毕前，其他试图执行当前对象的同步语句的线程都将处于阻塞状态，只有线程在当前对象的同步语句执行完毕后，监控器才会释放对象锁，并让最高优先级的阻塞线程处理它的同步语句。

也可以这样理解对象锁，当拨打公共信息服务台时，接话员（共享对象）可以被多个客户访问（提供服务），但每一次接话员只能为一个客户服务。当其为某个客户服务时，其状态为“忙碌”（获取了对象锁），其他客户只能等待。当接话员为当前的客户服务结束时，其状态就变成“空闲”（释放了对象锁），现在就可以继续为其他客户服务了。

下述代码用于实现任务描述 1.D.10，通过多线程演示不使用同步机制可能出现的情况。

【描述 1.D.10】 NoSyn.java

```
class Share {
    void print(String str) {
        System.out.print("[" + str);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.println("]");
    }
}
class Caller implements Runnable {
    String str;
    Share share;
    Thread thread;
    public Caller(Share share, String str) {
        this.share = share;
        this.str = str;
        // 实例化线程
        thread = new Thread(this);
        thread.start();
    }
    public void run() {
        share.print(str);
    }
}
public class NoSyn {
    public static void main(String args[]) throws InterruptedException {
        Share share = new Share();
        Caller call1 = new Caller(share, "A");
        Caller call2 = new Caller(share, "B");
        Caller call3 = new Caller(share, "C");
```


试读结束：需要全本请在线购买： www.ertongbook.com