

知识点列表

编号	名称	描述	级别
1	Java 多线程编程概念	理解 Java 多线程编程的概念，掌握什么叫程序、进程、线程，理解并发概念	*
2	Java 创建一个线程	掌握如何创建一个线程	**
3	线程的状态	理解线程状态图	**
4	线程状态管理	掌握 sleep()和 yield()的用法，以及调用该方法后线程处于的状态	**
5	线程的常用属性及方法	掌握线程常用属性及方法	*
6	两种方式创建线程	熟练掌握创建线程的两种方式	***
7	Sleep 状态的打断唤醒	理解并掌握 Sleep 状态，以及 IO 阻塞等	**
8	异步与同步	能够区分同步和异步的异同及优势	*
9	线程并发安全问题	理解并发问题，掌握解决并发安全问题的方法	*
10	Java 中同步的 API	了解 Java 提供的同步 API 方法	*

注： **"理解级别 ***"掌握级别 ****"应用级别

目录

1. Java 多线程编程概念 *	3
2. Java 创建一个线程 **	3
3. 线程的状态 **	4
4. 线程状态管理 **	6
5. 线程的常用属性及方法 **	7
6. 两种方式创建线程 ***	9
7. Sleep 状态的打断唤醒 **	11
8. 异步与同步 *	15
9. 线程并发安全问题 *	18
10. Java 中同步的 API *	26

1. Java 多线程编程概念 *

Java 语言的优势之一就是线程处理较为简单。

一般操作系统都支持同时运行多个任务，一个任务通常就是一个**程序**，每个运行中的程序被称为一个**进程**，当一个程序运行时，内部可能包含多个顺序执行流，每个顺序执行流就是一个**线程**。

- 1) **程序** 指令 + 数据的 byte 序列，如: qq.exe
- 2) **进程** 正在运行的程序，是程序动态的执行过程（运行于内存中）
- 3) **线程** 在进程内部，并发运行过程（Java 中的方法可以看做线程）
- 4) **并发** 进程是并发运行的，OS 将时间划分为很多时间片段（**时间片**），尽可能均匀分配给正在运行的程序，微观上进程走走停停，宏观上都在运行，这种都运行的现象叫并发，但是不是绝对意义上的“同时发生”

2. Java 创建一个线程 **

- 1) Thread 类
线程类（Thread）包含一个可以运行的过程（方法）：run()方法
- 2) 创建一个具体线程的步骤如下：
第一，继承 Thread 类
第二，覆盖 run 方法（就是更新运行过程），实现用户自己的过程
第三，创建线程实例（就是创建一个线程）
第四，使用线程实例的 start() 方法启动线程，启动以后线程会尽快的去并发执行 run()

【案例】基本线程演示

```
ThreadDemo.java X
1 package corejava.day10.ch05;
2 /** 基本线程演示 */
3 public class ThreadDemo {
4     public static void main(String[] args) {
5         Person1 p1 = new Person1(); //p1 线程实例
6         Person2 p2 = new Person2(); //p2 线程实例
7         p1.start();
8         p2.start();
9         System.out.println("main Over!");
10    }
11 }
```

```

12 class Person1 extends Thread{
13     public void run() {
14         for(int i=0; i<100; i++){
15             System.out.println("你是谁呀!");
16         }
17     }
18 }
19 class Person2 extends Thread{
20     public void run() {
21         for(int i=0; i<100; i++){
22             System.out.println("修理水管的!");
23         }
24     }

```

注：

- ✓ **main()方法**也是一个线程，之前学习的程序都是单线程的，从 main()方法开始执行
- ✓ 单核、双核、多核处理器的输出结果都不会一样

3. 线程的状态 **

线程的 5 中状态

1) **New** 新建状态

- 当程序使用 new 关键字创建了一个线程后，该线程就处于新建状态，此时线程还未启动，当线程对象调用 start()方法时，线程启动，进入 Runnable 状态

2) **Runnable** 可运行（就绪）状态

- 当线程处于 Runnable 状态时，表示线程准备就绪，等待获取 CPU

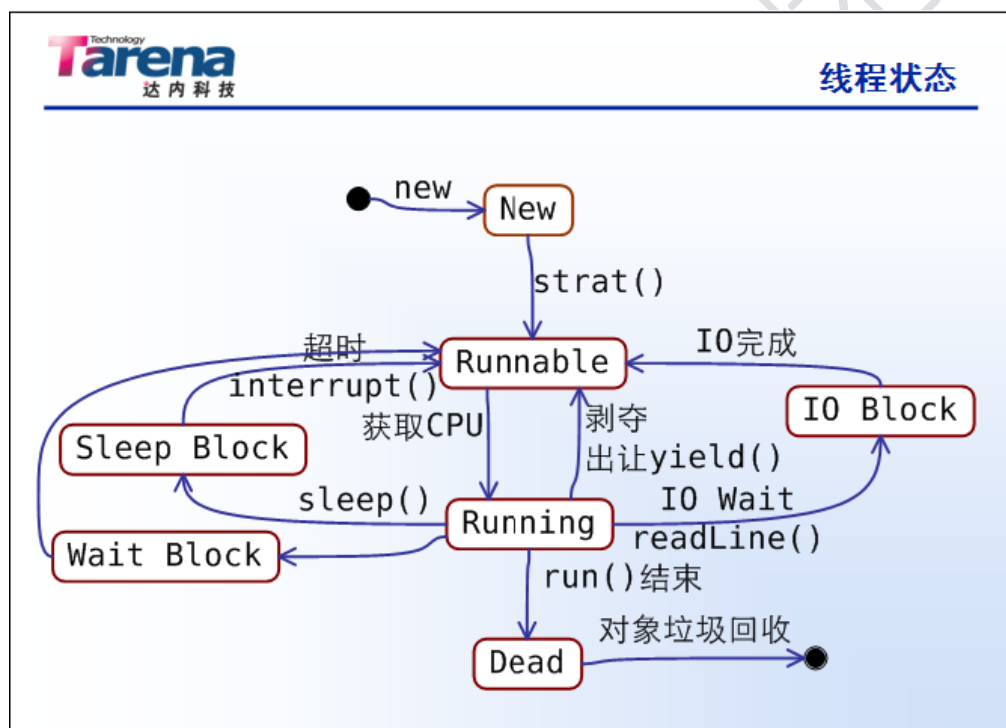
3) **Running** 运行（正在运行）状态

- 假如该线程获取了 CPU，则进入 Running 状态，开始执行线程体，即 run()方法中的内容
- 注意：
 - 如果系统只有 1 个 CPU，那么在任意时间点则只有 1 条线程处于 Running 状态；
 - 如果是双核系统，那么同一时间点会有 2 条线程处于 Running 状态
 - 但是，当线程数大于处理器数时，依然会是多条线程在同一个 CPU 上轮询执行
- 当一条线程开始运行时，如果它不是一瞬间完成，那么它不可能一直处于 Running 状态，线程在执行过程中会被中断，目的是让其它线程获得执行的机会，像这样**线程调度**的策略取决于底层平台。对于抢占式策略的平台而言，系统系统会给每个可执行的线程一小段时间来处理任务，当该时间段（**时间片**）用完，系统会剥夺该线程所占资源（CPU），让其他线程获得运行机会。
- 调用 **yield()方法**，可以使线程由 Running 状态进入 Runnable 状态

4) **Block** 阻塞（挂起）状态

- 当如下情况下，线程会进入阻塞状态：

- ✓ 线程调用了 sleep()方法主动放弃所占 CPU 资源
 - ✓ 线程调用了阻塞式 IO 方法（比如控制台输入方法），在该方法返回前，该线程被阻塞
 - ✓
 - 当正在执行的线程被阻塞时，其它线程就获得执行机会了。需要注意的是，**当阻塞结束时**，该线程将进入 Runnable 状态，而非直接进入 Running 状态
- 5) **Dead** 死亡状态
- 当线程的 run()方法执行结束，线程进入 Dead 状态
 - 需要注意的是，不要试图对一个已经死亡的线程调用 start()方法，线程死亡后将不能再次作为线程执行，系统会抛出 **IllegalThreadStateException** 异常



注：

- 1) new 运算创建线程后，线程进入 **New** 状态（初始状态）
- 2) 调用 start()方法后，线程从 New 状态进入 **Runnable** 状态（就绪状态）
 - start()方法是在 main()方法（**Running** 状态）中调用的
- 3) 线程结束后，进入 **Dead** 状态（死亡状态），被**对象垃圾回收**
- 4) main()方法结束后，其它线程，比如上例中 p1 和 p2 开始抢着进入 Running 状态
 - 由谁抢到是底层操作系统决定（操作系统分配时间片）
 - 单核处理器：在一个时间点上只有一个线程在 Running 状态；双核处理器：2 个
 - 如果 p1 进入 Running 状态，当操作系统分配给它的时间片到期时，p1 进入 Runnable

状态，p2 进入 Running 状态

- 在期间有可能其它的进程的线程获得时间片，那么 p1 和 p2 同时进入 Runnable 状态，等待操作系统分配时间片
- 5) 线程进入 Dead 状态后，只能被垃圾回收，不能再开始
- 6) 如果线程在运行过程中，自己调用了 **yield()方法**，则主动由 Running 状态进入 Runnable 状态

4. 线程状态管理 **

- 1) 让出 CPU **Thread.yield()**
当前线程让出处理器（离开 Running 状态），使当前线程进入 Runnable 状态等待
- 2) 休眠 **Thread.sleep(times)**
使当前线程从 Running 放弃处理器进入 Block 状态，休眠 times 毫秒，再返回到 Runnable
如果其他线程打断当前线程的 Block(sleep)，就会发生 InterruptedException。

【案例 1】Thread.yield()方法演示

```
ThreadDemo02.java
4 public static void main(String[] args) {
5     Person1 p1 = new Person1(); //p1 线程实例
6     Person2 p2 = new Person2(); //p2 线程实例
7     p1.start();
8     p2.start();
9     System.out.println("main Over!");
10 }
11 }
12 class Person1 extends Thread{
13     public void run() {
14         for(int i=0; i<100; i++){
15             System.out.println("你是谁呀!");
16             Thread.yield();
17         }
18     }
19 }
20 class Person2 extends Thread{
21     public void run() {
22         for(int i=0; i<100; i++){
23             System.out.println("修理水管的!");
24             Thread.yield();
25         }
26     }
27 }
```

5. 线程的常用属性及方法 **

- 1) **线程的优先级** (资源紧张时候, 尽可能优先)
 - `t3.setPriority(Thread.MAX_PRIORITY);` 设置为最高优先级
 - 默认有 10 优先级, 优先级高的线程获得执行 (进入 Running 状态) 的机会多. 机会的多少不能通过代码干预
 - 默认的优先级是 5
- 2) **后台线程** (守护线程, 精灵线程)
 - `t1.setDaemon(true);`
 - Java 进程的结束: 当前所有前台线程都结束时, Java 进程结束
 - 当前台线程结束时, 不管后台线程是否结束, 都要被停掉!
- 3) **获得线程名字**
`getName()`
- 4) **获得当前线程**
`Thread main = Thread.currentThread();`

【案例 1】Thread 优先级 `setPriority()` 方法

```

1 package corejava.day11.ch01;
2 /** 基本线程演示 */
3 public class ThreadDemo02 {
4     public static void main(String[] args) {
5         Person1 p1 = new Person1(); //p1 线程实例
6         Person2 p2 = new Person2(); //p2 线程实例
7         Person3 p3 = new Person3(); //p3 线程实例
8         p3.setPriority(Thread.MAX_PRIORITY); //最高优先级
9         p1.setPriority(Thread.MIN_PRIORITY); //最低优先级
10        p1.start();
11        p2.start();
12        p3.start();
13        System.out.println("main Over!");
14    }
15 }
16 class Person1 extends Thread{
17     public void run() {
18         for(int i=0; i<100; i++){
19             System.out.println("你是谁呀!");
20             Thread.yield();
21         }
22         System.out.println("#####你是谁呀!Over");
23     }
24 }
    
```

```

25 class Person2 extends Thread{
26     public void run() {
27         for(int i=0; i<100; i++){
28             System.out.println("修理水管的!");
29             Thread.yield();
30         }
31         System.out.println("#####修理水管的!Over");
32     }
33 }
34 class Person3 extends Thread{
35     public void run() {
36         for(int i=0; i<100; i++){
37             System.out.println("弹弓!");
38             Thread.yield();
39         }
40         System.out.println("#####弹弓!Over");
41     }
42 }

```

注：

- ✓ 线程优先级最高 (Thread.MAX_PRIORITY) 为 10，最低优先级 (Thread.MIN_PRIORITY) 为 1，默认 (Thread.NORM_PRIORITY) 为 5
- ✓ 一般情况下，优先级最高的线程最先结束，如本例中最先输出“弹弓! Over”，但是，线程的执行情况与系统资源和操作系统平台有关，如单核处理器，优先级最高的线程一般最先完成

【案例 2】设置后台线程（守护线程） setDaemon(true)

```

ThreadDemo02.java
1 package corejava.day11.ch02;
2 /** 基本线程演示 */
3 public class ThreadDemo02 {
4     public static void main(String[] args) {
5         Person1 p1 = new Person1(); //p1 线程实例
6         Person2 p2 = new Person2(); //p2 线程实例
7         p1.setDaemon(true); //守护线程
8         p1.start();
9         p2.start();
10        System.out.println("main Over!");
11    }
12 }

```



```

13 class Person1 extends Thread{
14     public void run() {
15         for(int i=0; i<1000; i++){
16             System.out.println("你是谁呀!");
17             Thread.yield();
18         }
19         System.out.println("#####你是谁呀!Over");
20     }
21 }
22 class Person2 extends Thread{
23     public void run() {
24         for(int i=0; i<100; i++){
25             System.out.println("修理水管的!");
26             Thread.yield();
27         }
28         System.out.println("#####修理水管的!Over");
29     }
30 }

```

注：

- ✓ Daemon 意为“精灵、守护神”
 - “守护神”就像“影子”，后台线程（守护线程）相当于“影子”
- ✓ 本例中，p1 被设置为后台线程，p2 和 main 是前台线程
- ✓ **Java 进程在全部前台线程结束时结束，而守护线程会被“提前杀掉”**
 - 本例中守护线程 p1 在前台线程 main 和 p2 结束前就被“提前杀掉”了，故不会输出“你是谁呀！”

6. 两种方式创建线程 ***

继承 Thread 类 (extends Thread) 或者实现 Runnable 接口 (implements Runnable)

1) 继承 Thread 类

■ 实现步骤：

- ✓ 继承 Thread 类, 覆盖 run()方法, 提供并发运程的过程
- ✓ 创建这个类的实例
- ✓ 使用 start() 方法启动线程

2) 实现 Runnable 接口

■ 实现步骤：

- ✓ 实现 Runnable 接口, 实现 run()方法, 提供并发运程的过程
- ✓ 创建这个类的实例, 用这个实例作为 Thread 构造器参数, 创建 Thread 类
- ✓ 使用 start() 方法启动线程

3) 使用内部类/匿名类创建线程

【案例】创建线程的方式

```

ThreadInitDemo.java
1 package corejava.day11.ch03;
2 /** 线程的创建方式 */
3 public class ThreadInitDemo {
4     public static void main(String[] args) {
5         /*1. 使用匿名内部类创建线程*/
6         Thread t1 = new Thread() { // 继承Thread类
7             public void run() {
8                 System.out.println("1##HI");
9             }
10        };
11        t1.start();
12
13        /*2. 使用Runnable 接口创建线程*/
14        //2.1 实现Runnable接口
15        Runnable runner = new Runnable() {
16            public void run() {
17                System.out.println("2##HI");
18            }
19        };
20        //2.2 在创建线程实例时候，将Runnable实例作为构造参数
21        Thread t2 = new Thread(runner);
22        t2.start();
23
24        /*3. 使用Runnable接口创建匿名类，创建线程实例*/
25        Thread t3 = new Thread(new Runnable() {
26            public void run() {
27                System.out.println("HI t3");
28            }
29        });
30        t3.start();
31
32        /*4. 创建匿名类实例，直接启动线程*/
33        new Thread() {
34            public void run() {
35                System.out.println("HI Thread");
36            }
37        }.start();
38
39        /*5. 创建匿名类实例，使用Runnable接口*/
40        new Thread(new Runnable() {
41            @Override
42            public void run() {
43                System.out.println("Hi,Runnable");
44            }
45        }).start();
    
```

```
46  
47     }  
48 }
```

注：

- ✓ 使用 Thread 或者 Runnable 两种方式都可以

7. Sleep 状态的打断唤醒 **

- 1) Thread.sleep(times)

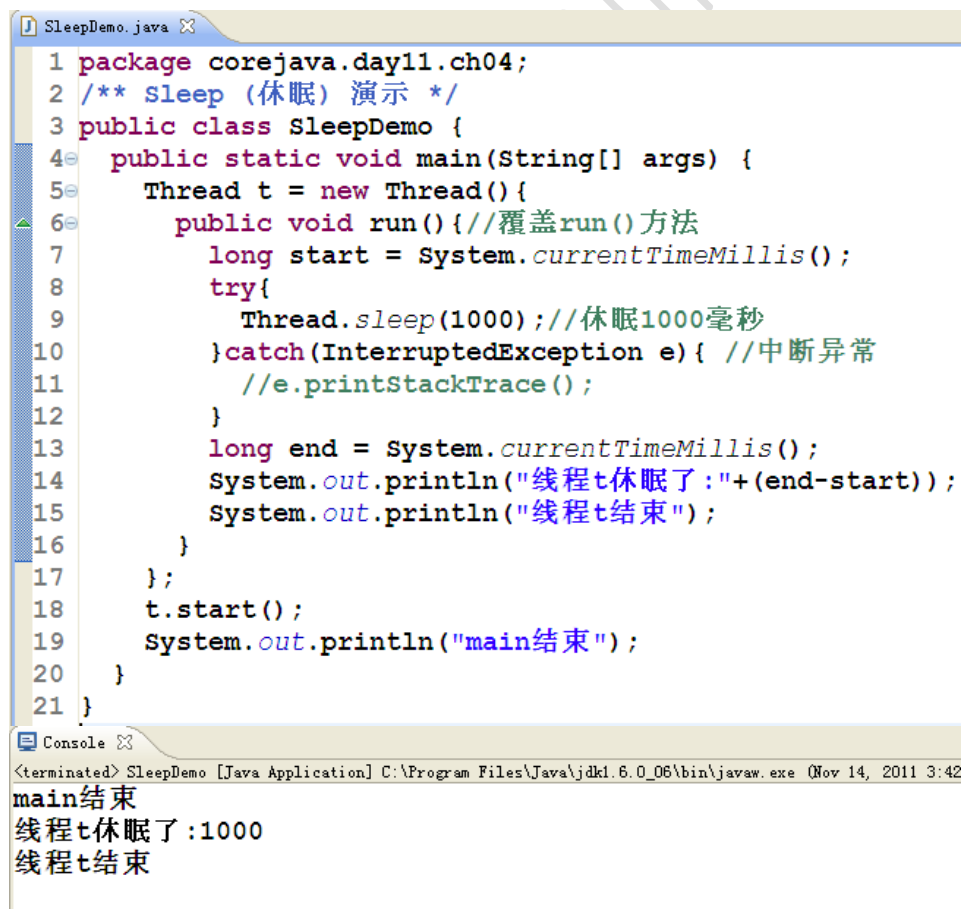
使当前线程从 Running 状态放弃处理器，进入 Block 状态，休眠 times（单位为毫秒），休眠结束后，再返回到 Runnable 状态

- 2) interrupt() 方法打断/中断

使用该方法可以让一个线程提前唤醒另外一个 sleep Block 的线程

- 3) 被唤醒线程会出现中断异常

【案例 1】线程休眠演示 sleep()



```

SleepDemo.java
1 package corejava.day11.ch04;
2 /** Sleep（休眠）演示 */
3 public class SleepDemo {
4     public static void main(String[] args) {
5         Thread t = new Thread() {
6             public void run() { //覆盖run()方法
7                 long start = System.currentTimeMillis();
8                 try {
9                     Thread.sleep(1000); //休眠1000毫秒
10                } catch (InterruptedException e) { //中断异常
11                    //e.printStackTrace();
12                }
13                long end = System.currentTimeMillis();
14                System.out.println("线程t休眠了:" + (end - start));
15                System.out.println("线程t结束");
16            }
17        };
18        t.start();
19        System.out.println("main结束");
20    }
21 }

Console
<terminated> SleepDemo [Java Application] C:\Program Files\Java\jdk1.6.0_06\bin\javaw.exe (Nov 14, 2011 3:42)
main结束
线程t休眠了:1000
线程t结束
    
```

注：

- ✓ InterruptedException 中断异常，在当前线程休眠被其它线程打断时发生

【案例 2】线程休眠打断演示 interrupt()

```

1 package corejava.day11.ch04;
2 /** Sleep (休眠) 演示
3  * 主线程休眠3秒，线程t休眠10秒，
4  * 主线程休眠3秒后，t的休眠被打断了
5  */
6 public class SleepDemo2 {
7     public static void main(String[] args) {
8         Thread t = new Thread() {
9             public void run() { //覆盖run()方法
10                 long start = System.currentTimeMillis();
11                 try {
12                     Thread.sleep(10000); //休眠10秒
13                 } catch (InterruptedException e) { //中断异常
14                     //e.printStackTrace();
15                 }
16                 long end = System.currentTimeMillis();
17                 System.out.println("线程t休眠了：" + (end - start));
18                 System.out.println("##线程t结束");
19             }
20         };
21         t.start();
22
23         try {
24             Thread.sleep(3000); //主线程休眠3秒
25             t.interrupt(); //在主线程中打断t的休眠
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29         System.out.println("main结束");
30     }
31 }

```

注：

- ✓ 程序执行步骤说明：
 - 主线程 main 首先执行，接着休眠 3 秒；线程 t 执行，接着准备休眠 10 秒
 - 当 3 秒时，主线程 main 休眠结束，继续执行代码
 - 接着，线程 t 的休眠被打断 (t.interrupt())，线程 t 结束
 - 主线程 main 结束
- ✓ 不能用 sleep() 方法计时，因为线程休眠结束后不是直接进入 Running 状态，而是进入 Runnable 状态等待系统分配时间片，所以会有差异，所以计时要使用系统提供的

【案例 3】线程休眠和中断演示

```

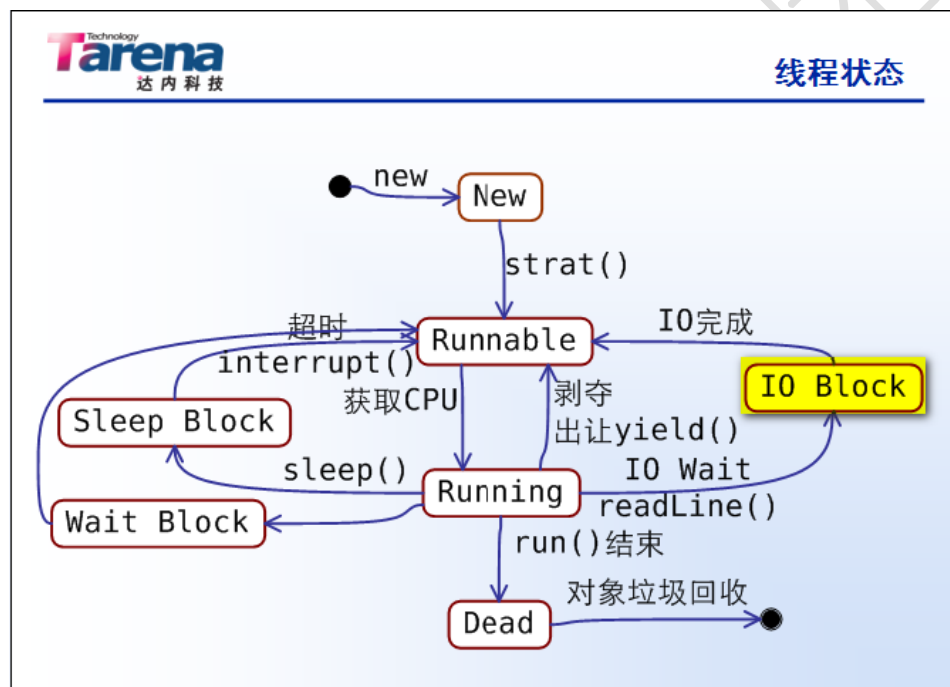
1 package corejava.day11.ch05;
2 /** Sleep的演示
3  * 一个线程代表睡觉
4  * 一个线程代表砸墙
5  * 砸墙线程打断睡觉线程
6  * */
7 public class SleepDemo2 {
8     public static void main(String[] args) {
9         //t1 代表睡觉线程, t1的睡觉时间一定比t2长
10        //如果被中断,t1就不再继续睡觉了
11        final Thread t1 = new Thread(){
12            public void run(){
13                for(int i=0; i<5; i++){
14                    System.out.println("去睡觉了!");
15                    try{
16                        Thread.sleep(5000);
17                    }catch(InterruptedException e){
18                        e.printStackTrace();
19                        System.out.println("干嘛哪! 破相了!");
20                        break;//如果被中断就不再睡觉了
21                    }
22                }
23            }
24        };
25        t1.start();
26
27        //t2 代表砸墙线程, t2砸完了, 将中断t1
28        Thread t2 = new Thread(){
29            public void run(){
30                for(int i=0; i<12; i++){
31                    System.out.println("砸墙, 咣当!");
32                    try {
33                        Thread.sleep(1000);
34                    } catch (InterruptedException e) {

```

```

35         e.printStackTrace();
36     }
37 }
38 System.out.println("终于砸穿了!");
39 t1.interrupt();
40 }
41 };
42 t2.start();
43 }
44 }
    
```

【案例 4】IO 阻塞现象演示



注：

- ✓ IO Block 是由 IO 操作时触发的，和 Sleep Block 不同，Sleep Block 是有 sleep()方法触发的
 - 触发方法如 readLine()、nextLine()、next()、nextInt()等
 - 一般 IO 读取方法都会发生 IO Block

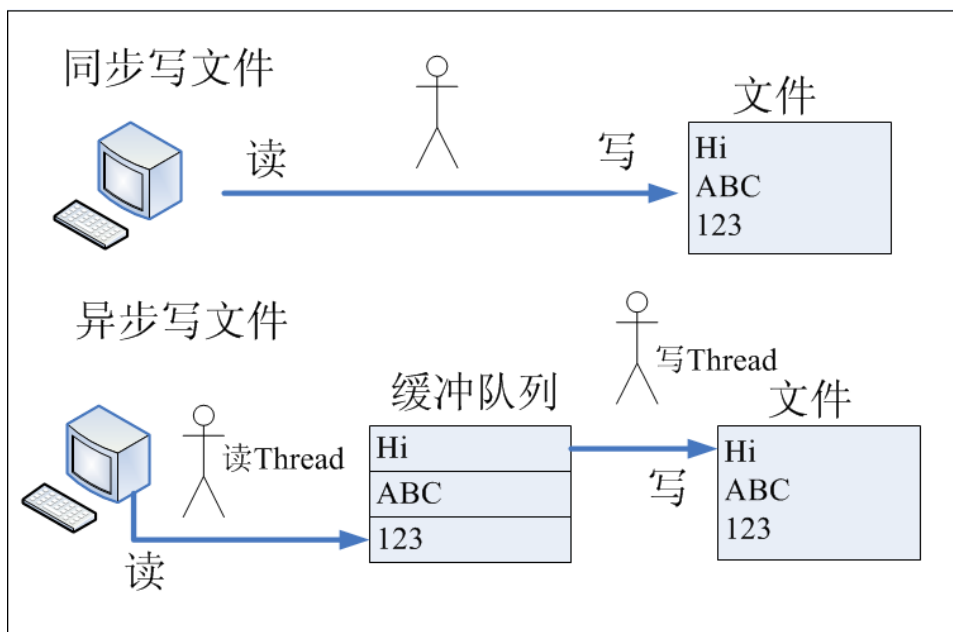
```

1 package corejava.day11.ch06;
2 import java.util.Scanner;
3 /** IO Block Demo
4  * 当 Block 发生时,线程停下,不运行,但没有结束
5  * 是挂起/阻塞(暂停)状态
6  * 挂起状态结束则返回到Runnable继续运行
7  */
8 public class IOBlockDemo {
9     public static void main(String[] args) {
10         Thread t = new Thread(){
11             public void run(){
12                 Scanner s = new Scanner(System.in);
13                 while(true){
14                     //s.nextLine() 会阻塞当前线程,直到IO完成为止
15                     //就是输入回车为止
16                     String str = s.nextLine();//相当于readLine()
17                     System.out.println(str);
18                     if(str.equalsIgnoreCase("q"))//忽略大小写
19                         break;
20                 }
21             }
22         };
23         t.start();
24     }

```

8. 异步与同步 *

- 1) 异步
并发, 各干自己的。如: 一群人上卡车
- 2) 同步
步调一致的处理。 如: 一群人上公交车



【案例 1】同步写文件

```

1 package corejava.day11.ch07;
2 import java.io.FileOutputStream;
6 /** 同步写文件
7  * 任务：读取控制台输入的信息，同步写入到一个文件中
8  * */
9 public class SyncWriterDemo {
10     public static void main(String[] args)
11         throws IOException {
12         Scanner s = new Scanner(System.in);
13         FileOutputStream file =
14             new FileOutputStream("demo.txt");
15         PrintWriter out = new PrintWriter(file);
16         while(true){
17             //读-写-读-写.... 同步读写
18             String str = s.nextLine();
19             out.println(str);
20             if(str.equalsIgnoreCase("q")){
21                 break;
22             }
23         }
24         out.close();
25     }
26 }

```


【案例 2】异步写文件

案例描述

- 1) 线程 1 负责将控制台信息读取到内存缓冲区（集合），如果控制台输入 quit 将结束输入，中断写出线程
- 2) 线程 2 负责将缓冲区中的信息写到硬盘文件，每次检查缓冲区是否有数据，如果有就写出，直到空为止，如果没有数据就休眠 5 秒，写出线程是后台线程，可以自动结束
- 3) 缓冲区采用队列(FIFO)的工作方式

参考代码

```

1 package corejava.day11.ch07;
2 import java.io.FileNotFoundException;
3
4
5
6
7
8 public class UnsyncWriterDemo {
9     public static void main(String[] args)
10         throws FileNotFoundException {
11         //1. 从缓冲区写入文件
12         //1.1 缓冲队列
13         final LinkedList<String> buf =
14             new LinkedList<String>();
15         //1.2 包装流
16         final PrintWriter out =
17             new PrintWriter(
18                 new FileOutputStream("unsync.txt"));
19         //1.3 负责将缓冲队列中的信息写入文件的线程
20         // 每5秒检查1次
21         final Thread writer = new Thread() {
22             public void run() {
23                 while(true) {
24                     if(buf.isEmpty()) {
25                         try {
26                             // System.out.println("空buf, 休眠5秒");
27                             out.flush();
28                             Thread.sleep(5000); //如果为空, 则休眠
29                         } catch (InterruptedException e) {
30                             e.printStackTrace();
31                         }
32                         continue;
33                     }
34                     String str = buf.removeFirst();
35                     // System.out.println("正在写入:"+str);
36                     out.println(str);
37                 }
38             }
39         };
40     }
41 }

```

```

39     };
40     //1.4 将写出线程设置为后台线程，可以自动结束
41     writer.setDaemon(true);
42
43     //2. 从控制台读取，写入缓冲区
44     //2.1 负责读取控制台信息到缓冲队列(LinkedList)的线程
45     final Thread reader = new Thread() {
46     public void run() {
47         Scanner s = new Scanner(System.in);
48         while(true) {
49             String str = s.nextLine();
50             //写到buf的最后，还可以用方法buf.push(str)
51             buf.addLast(str);
52             if(str.equalsIgnoreCase("Q")) {
53                 //中断唤醒写线程，在结束之前，再写一次
54                 writer.interrupt();
55                 break;
56             }
57         }
58     }
59     };
60
61     //3. 启动线程
62     reader.start();
63     writer.start();
64 }
65 }

```

9. 线程并发安全问题 *

- 1) 多个线程并发读写同一个临界资源时候会发生“**线程并发安全问题**”，如果保证多线程同步访问临界资源，就可以解决。
- 2) 常见的**临界资源**：
 - 多线程共享实例变量
 - 静态公共变量
- 3) 使用同步代码块解决线程并发安全问题
 - synchronized(同步监视器){
 }
 - 同步监视器 是一个任意对象实例. 是一个多个线程之间的互斥的锁机制. 多个线程要使用同一个"监视器"对象 实现同步互斥
 - 常见写法


```
synchronized(this){
                    }
                
```

- 如果方法的全部过程需要同步, 可以简单使用 `synchronized` 修饰方法, 相当于整个方法的 `synchronized(this)`
- 尽量减少同步范围, 提高并发效率

【案例 1】线程安全问题演示

```

1 package corejava.day11;
2 import corejava.day11.Table.Person;
3 public class SyncDemo {
4     public static void main(String[] args) {
5         Table table = new Table();
6         Person p1 = table.new Person();
7         Person p2 = table.new Person();
8         p1.start();
9         p2.start();
10        // p1和p2 共享同一个table, 桌子上有beans=20
11    }
12 }
13 class Table {
14     int beans = 20;
15     public int getBean() {
16         if (beans == 0)
17             throw new RuntimeException("没了!");
18         Thread.yield();
19         return beans--;
20     }
21     class Person extends Thread {
22     public void run() {
23         while (true) {
24             int bean = getBean();
25             System.out.println(getName() + "," + bean);
26             Thread.yield();
27         }
28     }
29 }
30 }

```

```

<terminated> SyncDemo [Java Application] C:\Java\jdk1.6.0_13\bin\javaw.exe (2011-11-30 上午11:09:15)
Thread-0, 3
Thread-1, 2
Thread-0, 1
Thread-1, 0
Thread-0, -1
Thread-1, -2
Thread-0, -3
Thread-1, -4
    
```

注：

- ✓ 在执行结果中出现了两个人 (p1 和 p2) 同时吃 1 个豆豆的情况，出现计数错误，原因就是**线程并发**造成的，这就是**线程安全问题**
- ✓ 控制台输出结果中，错误输出 (异常输出) 和标准输出出现的顺序常常不一样，原因就是 **Java 中的错误输出和标准输出是并发的**，Java 基于这样的设计是为了提高性能

执行结果分析

时序	线程 p1 状态	线程 p1 执行语句 Thread-0	临界资源 beans	线程 p2 状态	线程 p2 执行语句 Thread-1

1	Running	if (beans == 0) throw ...;	1	Runnable	
2	Running	Thread.yield();	1	Runnable	
3	Runnable		1	Running	if (beans == 0) throw ...;
4	Runnable		1	Running	Thread.yield();
5	Running	return beans--;	0	Runnable	
6	Running	输出: Thread-0, 1	0	Runnable	
7	Running	Thread.yield();	0	Runnable	
8	Runnable		-1	Running	return beans--;
9	Runnable		-1	Running	输出: Thread-0, 0
10	Runnable		-1	Running	Thread.yield();

11	Running	if (beans == 0) throw ...;	-1	Runnable	
12	Running	Thread.yield();	-1	Runnable	
13	Runnable		-1	Running	if (beans == 0) throw ...;
14	Runnable		-1	Running	Thread.yield();
15	Running	return beans--;	-2	Runnable	
16	Running	输出: Thread-0, -1	-2	Runnable	
17	Running	Thread.yield();	-2	Runnable	

注：

- ✓ 以上假设在一颗处理器上发生的一种假设的并发执行情况，实际上存在其它的并发时序
- ✓ 系统不仅仅只有两个线程，还有很多其他应用程序的线程一同参与并发
- ✓ 为了演示方便，加入代码 Thread.yield()，这是将并发问题放大了，即使不加，也存在着并发问题的风险

结论

判断桌上有没有豆子（if 判断）、取豆子、桌上豆子减 1 应该是一起操作的，否则不同线程并发访问同一方法 getBean()时就有风险

```

15 public int getBean() {
16     if (beans == 0)
17         throw new RuntimeException("没了!");
18     Thread.yield();
19     return beans--;
20 }
```

解决办法

- 1) 加 1 把“同步监视锁”
- 2) 使用 synchronized 关键字为读操作（代码 18 行）和写操作（代码第 21 行）加同步代码块

```

15  Object monitor = new Object();
16  public int getBean() {
17      synchronized (monitor) {
18          if (beans == 0)
19              throw new RuntimeException("没了!");
20          Thread.yield();
21          return beans--;
22      }
23  }

```

错误的写法

“同步监视锁”是共有的，只有一把，如下表示“每个调用 getBean() 方法的人都有一把锁”，是错误的写法，没有意义

```

25  public int getBean() {
26      Object monitor = new Object();
27      synchronized (monitor) {
28          if (beans == 0)
29              throw new RuntimeException("没了!");
30          Thread.yield();
31          return beans--;
32      }
33  }

```

其它写法 01

只要“同步监视锁”只有一把（一个对象）就可以，如下所示的写法也是可以的

```

16  public int getBean() {
17      synchronized (this) {
18          if (beans == 0)
19              throw new RuntimeException("没了!");
20          Thread.yield();
21          return beans--;
22      }
23  }

```

其它写法 02

直接使用 synchronized 关键字修饰方法，相当于 `synchronized(this){}`

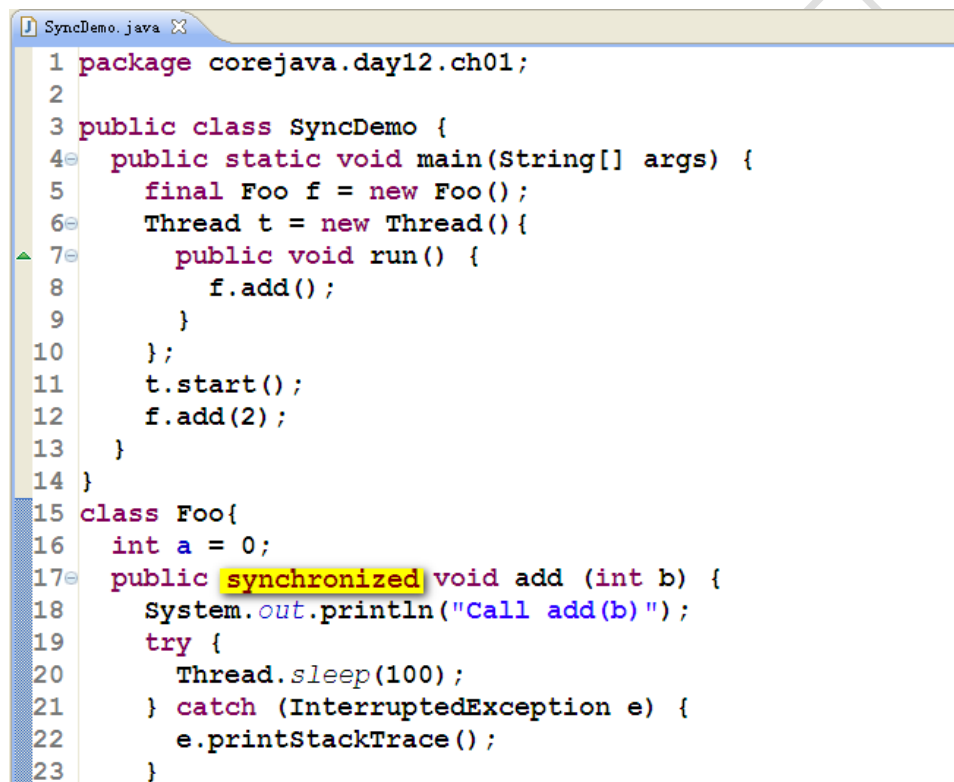
建议，尽量减少同步范围，提高并发效率

```

16 public synchronized int getBean() {
17     if (beans == 0)
18         throw new RuntimeException("没了!");
19     Thread.yield();
20     return beans--;
21 }

```

【案例 2】线程安全问题演示



```

SyncDemo.java
1 package corejava.day12.ch01;
2
3 public class SyncDemo {
4     public static void main(String[] args) {
5         final Foo f = new Foo();
6         Thread t = new Thread() {
7             public void run() {
8                 f.add();
9             }
10        };
11        t.start();
12        f.add(2);
13    }
14 }
15 class Foo {
16     int a = 0;
17     public synchronized void add (int b) {
18         System.out.println("Call add(b)");
19         try {
20             Thread.sleep(100);
21         } catch (InterruptedException e) {
22             e.printStackTrace();
23         }
24     }
25 }

```

```

24     }
25     a+=b;
26     System.out.println("Over Call add(b)");
27 }
28 public synchronized void add () {
29     System.out.println("Call add()");
30     try {
31         Thread.sleep(100);
32     } catch (InterruptedException e) {
33         e.printStackTrace();
34     }
35     a++;
36     System.out.println("Over Call add()");
37 }
38 }

```

Console

```

<terminated> SyncDemo [Java Application] C:\Program Files\Java\jdk1.6.0_06\bin\javaw.exe (Nov 16, 2011 4:54:11 PM)
Call add(b)
Over Call add(b)
Call add()
Over Call add()

```

执行结果分析

- 1) 第 4 行 程序执行入口 main()方法，**主线程启动**，进入 **Running 状态**
- 2) 第 5 行 堆内存中创建对象 Foo f，Java 所有对象会自带一把“锁”（锁默认为打开，打开和关闭对对象的访问和修改没有任何影响）
- 3) 第 6 行-第 10 行 堆内存中创建对象 Thread t
- 4) 第 11 行 线程 t 启动，进入 **Runnable 状态**
- 5) 第 12 行 如果主线程 main 此时在 **Running 状态**，则继续执行本行代码，调用对象 Foo f 的 synchroized add(int b)方法
- 6) 第 17 行 由于有 synchroized 关键字修饰，所以此时需要在 add(int b)方法上“**检查** this 对象上的锁是否已锁上”，此时是开着的，**上锁**，开始执行 add (int b) 方法内部代码
- 7) 第 19 行 输出语句，输出 **“Call add(b)”**
- 8) 第 20 行 Thread.sleep(100)，此时主线程 main 进入 **Block 状态**，锁仍是 **“上锁状态”**，处理器空闲
- 9) 第 11 行 假设线程 Thread t 抢到处理器，则由 Runnable 进入 **Running 状态**
- 10) 第 8 行 调用方法 f.add()
- 11) 第 28 行 执行方法 synchroized f.add()，首先“**检查** this 上的锁”，此时“this”是同一个对象 f，发现锁是“**锁上的**”（执行第 17 行代码时锁上的），Thread t 进入 **Block 状态**等待，处理器空闲给其它线程使用


```
public static void main(String[] args) {
    final Foo f = new Foo();
    Thread t = new Thread() {
        public void run() {
            f.add();
        }
    };
    t.start();
    f.add(2);
}
```

- 12) **第 20 行** 主线程 Thread.sleep(100)睡醒了，由 Block 进入 **Runnable 状态**，等待系统分配时间片，如果主线程抢到处理器，则进入 **Running 状态**
- 13) **第 25 行** 执行 a+=b；得到 a = 2
- 14) **第 26 行** 输出 **“Over Call add(b)”**，离开 synchroized 代码块，同时**“将锁打开”**，并通知 (notify) 处于 Block 状态的线程，处于 Block 状态的线程 Thread t 立即进入 **Runnable 状态**
主线程 main 结束，进入 **Dead 状态**，处理器空闲
- 15) **第 28 行** 处于 **Runnable 状态**的 Thread t 如果抢到处理器，进入 **Running 状态**，由于 add()方法有 synchroized 关键字修饰，所以此时线程 Thread t 需要在 add()方法上**“检查 this 对象上的锁是否已锁上”**，此时是开着的，**上锁**，开始执行 add () 方法内部代码
- 16) **第 29 行** 输出 **“Call add()”**
- 17) **第 31 行** Thread.sleep(100)，此时线程 Thread t 进入 **Block 状态**，锁仍是**“上锁状态”**，处理器空闲
- 18) **第 31 行** 线程 Thread t 执行 Thread.sleep(100)睡醒了，进入 **Runnable 状态**，等待进入 Running 状态
- 19) **第 31 行** 如果线程 Thread t 抢到处理器，进入 **Running 状态**，继续执行方法 add()代码
- 20) **第 35-36 行** a++；得 a=3，输出 **“Over Call add()”**
- 21) **第 37 行** 离开 synchroized 代码块，同时**“将锁打开”**，线程 Thread t 结束，进入 **Dead 状态**
- 22) **本段代码的核心价值**：保护 Foo 的属性 a 的值的修改
- 23) **本段代码的业务意义**：拍卖商品，add(int b)方法表示指定数额的加价，add()表示默认数额的加价，都修改 Foo 的属性 int a (拍卖价)，两个方法需要加同步锁

补充说明

this 对象上的“锁”相当于“接力棒”，一个时间点只有一个线程可以获得
如下相当于有了两把“锁”（f 和 f2）

由 f 调用的方法 f.add()和 f.add(2)不是并发的；

而因为有 2 把锁（f 和 f2），所以由 f 调用的方法和由 f2 调用的方法是并发的

```

3 public class SyncDemo {
4     public static void main(String[] args) {
5         final Foo f = new Foo();
6         final Foo f2 = new Foo();
7         Thread t = new Thread(){
8             public void run() {
9                 f.add();
10                f2.add();
11            }
12        };
13        t.start();
14        f.add(2);
15        f2.add(2);
16    }
17 }

```

输出结果如下所示：

```

Console
<terminated> SyncDemo [Java Application] C:\Program Files\Java\jdk1.6.0_06\bin\javaw.exe (Nov 17, 2011 11:53:
Call add(b)
Over Call add(b)
Call add(b)
Call add()
Over Call add(b)
Over Call add()
Call add()
Over Call add()

```

10. Java 中同步的 API *

- 1) StringBuffer 是同步的，synchronized append();
StringBuilder 不是同步的，append();
- 2) Vector 和 Hashtable 是同步的；ArrayList 和 HashMap 不是同步的
- 3) Collections.synchronizedList()
Collections.synchronizedMap()
ArrayList list = new ArrayList();
List synclist = Collections.synchronizedList(list);

StringBuffer 源码演示

```
corejava.Day12.txt corejava.Day11.txt *SyncDemo.java StringBuffer.class
/**
 * @see java.lang.String#valueOf(java.lang.Object)
 * @see #append(java.lang.String)
 */
public synchronized StringBuffer append(Object obj)
super.append(String.valueOf(obj));
    return this;
}

public synchronized StringBuffer append(String str)
super.append(str);
    return this;
}
```