线程详解

目录(?)[-]

本文主要讲了java中多线程的使用方法、线程同步、线程数据传递、线程状态及相应的一些线程函数用法、概述等。

首先讲一下进程和线程的区别：

进程：每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销，一个进程包含1--n个线程。

线程：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。

线程和进程一样分为五个阶段：创建、就绪、运行、阻塞、终止。

多进程是指操作系统能同时运行多个任务（程序）。

多线程是指在同一程序中有多个顺序流在执行。

在java中要想实现多线程，有两种手段，一种是继续Thread类，另外一种是实现Runable接口。

一、扩展java.lang.Thread类

```
package com.multithread.learning;
/**
*@functon 多线程学习
*@author 林炳文
*@time 2015.3.9
*/
class Thread1 extends Thread{
 private String name;
   public Thread1(String name) {
     this.name=name;
   }
 public void run() {
      for (int i = 0; i < 5; i++) {
          System.out.println(name + "运行 ： " + i);
          try {
              sleep((int) Math.random() * 10);
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }

}
```

```
}
public class Main {

 public static void main(String[] args) {
  Thread1 mTh1=new Thread1("A");
  Thread1 mTh2=new Thread1("B");
  mTh1.start();
  mTh2.start();

 }

}
```

输出：

A运行 ： 0

B运行 ： 0

A运行 ： 1

A运行 ： 2

A运行 ： 3

A运行 ： 4

B运行 ： 1

B运行 ： 2

B运行 ： 3

B运行 ： 4

再运行一下：

A运行：0
B运行：0
B运行：1
B运行：2
B运行：3
B运行：4
A运行：1
A运行：2
A运行：3
A运行：4

说明：

程序启动运行main时候，java虚拟机启动一个进程，主线程main在main()调用时候被创建。随着调用MitiSay的两个对象的start方法，另外两个线程也启动了，这样，整个应用就在多线程下运行。

注意：start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。

从程序运行的结果可以发现，多线程程序是乱序执行。因此，只有乱序执行的代码才有必要设计为多线程。

Thread.sleep()方法调用目的是不让当前线程独自霸占该进程所获取的CPU资源，以留出一定时间给其他线程执行的机会。

实际上所有的多线程代码执行顺序都是不确定的，每次执行的结果都是随机的。

但是start方法重复调用的话，会出现java.lang.IllegalThreadStateException异常。

```
 Thread1 mTh1=new Thread1("A");
 Thread1 mTh2=mTh1;
 mTh1.start();
 mTh2.start();
```

输出：

```
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Unknown Source)
    at com.multithread.learning.Main.main(Main.java:31)
A运行 ： 0
A运行 ： 1
A运行 ： 2
A运行 ： 3
A运行 ： 4
```

二、实现java.lang.Runnable接口

```
/**
 *@functon 多线程学习
 *@author 林炳文
 *@time 2015.3.9
 */
package com.multithread.runnable;
class Thread2 implements Runnable{
 private String name;

 public Thread2(String name) {
  this.name=name;
 }

 @Override
 public void run() {
    for (int i = 0; i < 5; i++) {
          System.out.println(name + "运行 ： " + i);
          try {
           Thread.sleep((int) Math.random() * 10);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
       }

 }

}
public class Main {

 public static void main(String[] args) {
  new Thread(new Thread2("C")).start();
  new Thread(new Thread2("D")).start();
 }

}
```

输出：

C运行 ： 0

D运行 ： 0

D运行 ： 1

C运行 ： 1

D运行 ： 2

C运行 ： 2

D运行 ： 3

C运行 ： 3

D运行 ： 4

C运行 ： 4

说明：

Thread2类通过实现Runnable接口，使得该类有了多线程类的特征。run（）方法是多线程程序的一个约定。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是扩展Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

三、Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

```
package com.multithread.learning;
/**
 *@functon 多线程学习,继承Thread，资源不能共享
 *@author 林炳文
 *@time 2015.3.9
 */
class Thread1 extends Thread{
 private int count=5;
 private String name;
   public Thread1(String name) {
      this.name=name;
   }
 public void run() {
       for (int i = 0; i < 5; i++) {
           System.out.println(name + "运行  count= " + count--);
           try {
               sleep((int) Math.random() * 10);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
       }

 }
}
```

```
public class Main {

 public static void main(String[] args) {
  Thread1 mTh1=new Thread1("A");
  Thread1 mTh2=new Thread1("B");
  mTh1.start();
  mTh2.start();

 }

}
```

输出：

```
B运行  count= 5
A运行  count= 5
B运行  count= 4
B运行  count= 3
B运行  count= 2
B运行  count= 1
A运行  count= 4
A运行  count= 3
A运行  count= 2
A运行  count= 1
```

从上面可以看出，不同的线程之间count是不同的，这对于卖票系统来说就会有很大的问题，当然，这里可以用同步来作。这里我们用Runnable来做下看看

```
/**
 *@functon 多线程学习 继承runnable，资源能共享
 *@author 林炳文
 *@time 2015.3.9
 */
package com.multithread.runnable;
class Thread2 implements Runnable{
   private int count=15;
 @Override
 public void run() {
   for (int i = 0; i < 5; i++) {
    System.out.println(Thread.currentThread().getName() + "运行  count= " + count--);
          try{
           Thread.sleep((int) Math.random() * 10);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
      }

}
```

```
}
public class Main {

 public static void main(String[] args) {

  Thread2 my = new Thread2();
      new Thread(my, "C").start();//同一个mt，但是在Thread中就不可以，如果用同一个实例化对象mt，就会出现异常
      new Thread(my, "D").start();
      new Thread(my, "E").start();
 }

}
```

输出：

C运行 count= 15

D运行 count= 14

E运行 count= 13

D运行 count= 12

D运行 count= 10

D运行 count= 9

D运行 count= 8

C运行 count= 11

E运行 count= 12

C运行 count= 7

E运行 count= 6

C运行 count= 5

E运行 count= 4

C运行 count= 3

E运行 count= 2

这里要注意每个线程都是用同一个实例化对象，如果不是同一个，效果就和上面的一样了！

总结：

实现**Runnable**接口比继承**Thread**类所具有的优势：

1）：适合多个相同的程序代码的线程去处理同一个资源

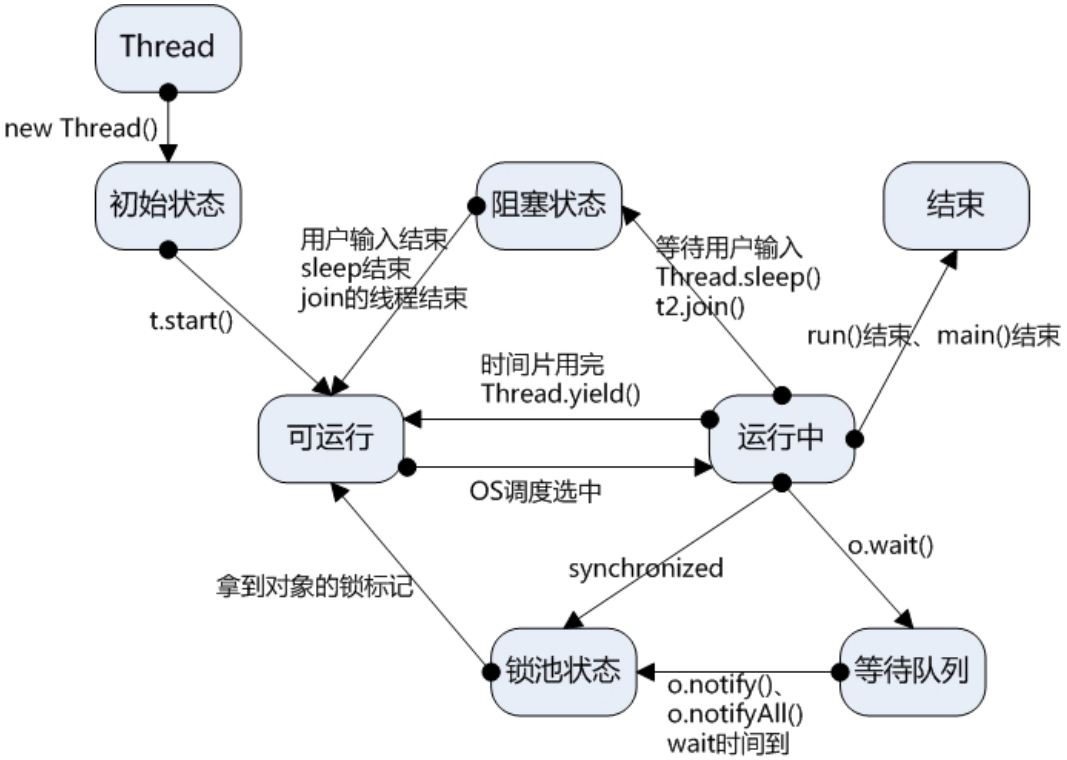
2）：可以避免**java**中的单继承的限制

3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

提醒一下大家：**main**方法其实也是一个线程。在**java**中所以的线程都是同时启动的，至于什么时候，哪个先执行，完全看谁先得到**CPU**的资源。

在**java**中，每次程序运行至少启动**2**个线程。一个是**main**线程，一个是垃圾收集线程。因为每当使用**java**命令执行一个类的时候，实际上都会启动一个ＪＶＭ，每一个ｊＶＭ实习在就是在操作系统中启动了一个进程。

四、线程状态转换



1、新建状态（New）：新创建了一个线程对象。

2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3、运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。

（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5、死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

五、线程调度

线程的调度

1、调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX_PRIORITY

线程可以具有的最高优先级，取值为10。

static int MIN_PRIORITY

线程可以具有的最低优先级，取值为1。

static int NORM_PRIORITY

分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

六、常用函数说明

①sleep(long millis): 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）

②join():指等待t线程终止。

使用方式。

join是Thread类的一个方法，启动线程后直接调用，即join()的作用是：“等待该线程终止”，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子线程结束了才能执行。

```
Thread t = new AThread(); t.start(); t.join();
```

为什么要用join()方法

在很多情况下，主线程生成并起动了子线程，如果子线程里要进行大量的耗时的运算，主线程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也就是主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。

不加join。

```
/**
 *@functon 多线程学习,join
 *@author 林炳文
 *@time 2015.3.9
 */
package com.multithread.join;
class Thread1 extends Thread{
 private String name;
    public Thread1(String name) {
     super(name);
       this.name=name;
    }
```

```
 public void run() {
  System.out.println(Thread.currentThread().getName() + " 线程运行开始!");
        for (int i = 0; i < 5; i++) {
            System.out.println("子线程"+name + "运行 : " + i);
            try {
                sleep((int) Math.random() * 10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName() + " 线程运行结束!");
 }
}

public class Main {

 public static void main(String[] args) {
  System.out.println(Thread.currentThread().getName()+"主线程运行开始!");
  Thread1 mTh1=new Thread1("A");
  Thread1 mTh2=new Thread1("B");
  mTh1.start();
  mTh2.start();
  System.out.println(Thread.currentThread().getName()+ "主线程运行结束!");

 }

}
```

输出结果：

main主线程运行开始!

main主线程运行结束!

B 线程运行开始!

子线程B运行 : 0

A 线程运行开始!

子线程A运行 : 0

子线程B运行 : 1

子线程A运行 : 1

子线程A运行 : 2

子线程A运行 : 3

子线程A运行 : 4

A 线程运行结束!

子线程B运行 : 2

子线程B运行 : 3

子线程B运行 : 4

B 线程运行结束!

发现主线程比子线程早结束

加join

```
public class Main {

 public static void main(String[] args) {
  System.out.println(Thread.currentThread().getName()+"主线程运行开始!");
  Thread1 mTh1=new Thread1("A");
  Thread1 mTh2=new Thread1("B");
  mTh1.start();
  mTh2.start();
  try {
   mTh1.join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  try {
   mTh2.join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  System.out.println(Thread.currentThread().getName()+ "主线程运行结束!");

 }

}
```

运行结果：

main主线程运行开始!

A 线程运行开始!

子线程A运行 : 0

B 线程运行开始!

子线程B运行 : 0

子线程A运行 : 1

子线程B运行 : 1

子线程A运行 : 2

子线程B运行 : 2

子线程A运行 : 3

子线程B运行 : 3

子线程A运行 : 4

子线程B运行 : 4

A 线程运行结束!

主线程一定会等子线程都结束了才结束

③yield():暂停当前正在执行的线程对象，并执行其他线程。

Thread.yield()方法作用是：暂停当前正在执行的线程对象，并执行其他线程。

yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

结论：yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。可看上面的图。

```
/**
 *@functon 多线程学习 yield
 *@author 林炳文
 *@time 2015.3.9
 */
package com.multithread.yield;
class ThreadYield extends Thread{
    public ThreadYield(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 1; i <= 50; i++) {
            System.out.println("" + this.getName() + "-----" + i);
            // 当i为30时，该线程就会把CPU时间让掉，让其他或者自己的线程执行（也就是谁先抢到谁执行）
            if (i ==30) {
                this.yield();
            }
        }

}
}

public class Main {

 public static void main(String[] args) {

  ThreadYield yt1 = new ThreadYield("张三");
    ThreadYield yt2 = new ThreadYield("李四");
        yt1.start();
        yt2.start();
 }

}
```

运行结果：

第一种情况：李四（线程）当执行到30时会CPU时间让掉，这时张三（线程）抢到CPU时间并执行。

第二种情况：李四（线程）当执行到30时会CPU时间让掉，这时李四（线程）抢到CPU时间并执行。

sleep()和yield()的区别

sleep()和yield()的区别):sleep()使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会被执行；yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

sleep 方法使当前运行中的线程睡眼一段时间，进入不可运行状态，这段时间的长短是由程序设定的，yield 方法使当前线程让出 CPU 占有权，但让出的时间是不可设定的。实际上，yield()方法对应了如下操作：先检测当前是否有相同优先级的线程处于同可运行状态，如有，则把 CPU 的占有权交给此线程，否则，继续运行原来的线程。所以yield()方法称为“退让”，它把运行机会让给了同等优先级的其他线程

另外，sleep 方法允许较低优先级的线程获得运行机会，但 yield() 方法执行时，当前线程仍处在可运行状态，所以，不可能让出较低优先级的线程些时获得 CPU 占有权。在一个运行系统中，如果较高优先级的线程没有调用 sleep 方法，又没有受到 I\O 阻塞，那么，较低优先级线程只能等待所有较高优先级的线程运行结束，才有机会运行。

④setPriority(): 更改线程的优先级。

MIN_PRIORITY = 1

NORM_PRIORITY = 5

MAX_PRIORITY = 10

用法：

```
Thread4 t1 = new Thread4("t1");
Thread4 t2 = new Thread4("t2");
t1.setPriority(Thread.MAX_PRIORITY);
t2.setPriority(Thread.MIN_PRIORITY);
```

⑤interrupt():中断某个线程，这种结束方式比较粗暴，如果t线程打开了某个资源还没来得及关闭也就是run方法还没有执行完就强制结束线程，会导致资源无法关闭

要想结束进程最好的办法就是用sleep()函数的例子程序里那样，在线程类里面用以个boolean型变量来控制run()方法什么时候结束，run()方法一结束，该线程也就结束了。

⑥wait()

Obj.wait()，与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作，从语法角度来说就是Obj.wait(),Obj.notify必须在synchronized(Obj){...}语句块内。从功能上来说wait就是说线程在获取对象锁后，主动释放对象锁，同时本线程休眠。直到有其它线程调用对象的notify()唤醒该线程，才能继续获取对象锁，并继续执行。相应的notify()就是对对象锁的唤醒操作。但有一点需要注意的是notify()调用后，并不是马上就释放对象锁的，而是在相应的synchronized(){}语句块执行结束，自动释放锁后，JVM会在wait()对象锁的线程中随机选取一线程，赋予其对象锁，唤醒线程，继续执行。这样就提供了在线程间同步、唤醒的操作。Thread.sleep()与Object.wait()二者都可以暂停当前线程，释放CPU控制权，主要的区别在于Object.wait()在释放CPU同时，释放了对象锁的控制。

单单在概念上理解清楚了还不够，需要在实际的例子中进行测试才能更好的理解。对Object.wait()，Object.notify()的应用最经典的例子，应该是三线程打印ABC的问题了吧，这是一道比较经典的面试题，题目要求如下：

建立三个线程，A线程打印10次A，B线程打印10次B,C线程打印10次C，要求线程同时运行，交替打印10次ABC。这个问题用Object的wait()，notify()就可以很方便的解决。代码如下：

```
/**
 * wait用法
 * @author DreamSea
 * @time 2015.3.9
 */
```

```
package com.multithread.wait;
public class MyThreadPrinter2 implements Runnable {

    private String name;
    private Object prev;
    private Object self;

    private MyThreadPrinter2(String name, Object prev, Object self) {
        this.name = name;
        this.prev = prev;
        this.self = self;
    }

    @Override
    public void run() {
        int count = 10;
        while (count > 0) {
            synchronized (prev) {
                synchronized (self) {
                    System.out.print(name);
                    count--;

                    self.notify();
                }
                try {
                    prev.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

        }
    }

    public static void main(String[] args) throws Exception {
        Object a = new Object();
        Object b = new Object();
        Object c = new Object();
        MyThreadPrinter2 pa = new MyThreadPrinter2("A", c, a);
        MyThreadPrinter2 pb = new MyThreadPrinter2("B", a, b);
        MyThreadPrinter2 pc = new MyThreadPrinter2("C", b, c);


        new Thread(pa).start();
        Thread.sleep(100);  //确保按顺序A、B、C执行
```

```
        new Thread(pb).start();
        Thread.sleep(100);
        new Thread(pc).start();
        Thread.sleep(100);
        }
}
```

输出结果：

ABCABCABCABCABCABCABCABCABCABC

先来解释一下其整体思路，从大的方向上来讲，该问题为三线程间的同步唤醒操作，主要的目的就是ThreadA->ThreadB->ThreadC->ThreadA循环执行三个线程。为了控制线程执行的顺序，那么就必须要确定唤醒、等待的顺序，所以每一个线程必须同时持有两个对象锁，才能继续执行。一个对象锁是prev，就是前一个线程所持有的对象锁。还有一个就是自身对象锁。主要的思想就是，为了控制执行的顺序，必须要先持有prev锁，也就前一个线程要释放自身对象锁，再去申请自身对象锁，两者兼备时打印，之后首先调用self.notify()释放自身对象锁，唤醒下一个等待线程，再调用prev.wait()释放prev对象锁，终止当前线程，等待循环结束后再次被唤醒。运行上述代码，可以发现三个线程循环打印ABC，共10次。程序运行的主要过程就是A线程最先运行，持有C,A对象锁，后释放A,C锁，唤醒B。线程B等待A锁，再申请B锁，后打印B，再释放B，A锁，唤醒C，线程C等待B锁，再申请C锁，后打印C，再释放C,B锁，唤醒A。看起来似乎没什么问题，但如果你仔细想一下，就会发现有问题，就是初始条件，三个线程按照A,B,C的顺序来启动，按照前面的思考，A唤醒B，B唤醒C，C再唤醒A。但是这种假设依赖于JVM中线程调度、执行的顺序。

wait和sleep区别

共同点：

1. 他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

2. wait()和sleep()都可以通过interrupt()方法 打断线程的暂停状态 ，从而使线程立刻抛出InterruptedException。

如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep /join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出InterruptedException 。

不同点：

1. Thread类的方法：sleep(),yield()等

Object的方法：wait()和notify()等

2. 每个对象都有一个锁来控制同步访问。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3. wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

4. sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

所以sleep()和wait()方法的最大区别是：

sleep()睡眠时，保持对象锁，仍然占有该锁；

而wait()睡眠时，释放对象锁。

但是wait()和sleep()都可以通过interrupt()方法打断线程的暂停状态，从而使线程立刻抛出InterruptedException（但不建议使用该方法）。

sleep（）方法

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CUP的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会;

sleep()是Thread类的Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个Synchronized块中调用Sleep()方法是，线程虽然休眠了，但是对象的机锁并木有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。

在sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级。

wait（）方法

wait()方法是Object类里的方法；当一个线程执行到wait()方法时，它就进入到一个和该对象相关的等待池中，同时失去（释放）了对象的机锁（暂时失去机锁，wait(long timeout)超时时间到后还需要返还对象锁）；其他线程可以访问；

wait()使用notify或者notifyAlll或者指定睡眠时间来唤醒当前等待池中的线程。

wiat()必须放在synchronized block中，否则会在program runtime时扔出"java.lang.lllegalMonitorStateException"异常。

七、常见线程名词解释

主线程：JVM调用程序main()所产生的线程。

当前线程：这个是容易混淆的概念。一般指通过Thread.currentThread()来获取的进程。

后台线程：指为其他线程提供服务的线程，也称为守护线程。JVM的垃圾回收线程就是一个后台线程。**用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束**

前台线程：是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过isDaemon()和setDaemon()方法来判断和设置一个线程是否为后台线程。

线程类的一些常用方法：

sleep(): 强迫一个线程睡眠N毫秒。
isAlive(): 判断一个线程是否存活。
join(): 等待线程终止。
activeCount(): 程序中活跃的线程数。
enumerate(): 枚举程序中的线程。
currentThread(): 得到当前线程。
isDaemon(): 一个线程是否为守护线程。
setDaemon(): 设置一个线程为守护线程。(用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束)
setName(): 为线程设置一个名称。
wait(): 强迫一个线程等待。
notify(): 通知一个线程继续运行。
setPriority(): 设置一个线程的优先级。

八、线程同步

1、synchronized关键字的作用域有二种：

1）是某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

2）是某个类的范围，synchronized static aStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。

2、除了方法前用synchronized关键字，synchronized关键字还可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。用法是: synchronized(this){/*区块*/}，它的作用域是当前对象；

3、synchronized关键字是不能继承的，也就是说，基类的方法synchronized f(){} 在继承类中并不自动是synchronized f(){}，而是变成了f(){}。继承类需要你显式的指定它的某个方法为synchronized方法；

Java对多线程的支持与同步机制深受大家的喜爱，似乎看起来使用了synchronized关键字就可以轻松地解决多线程共享数据同步问题。到底如何？——还得对synchronized关键字的作用进行深入了解才可定论。

总的说来，synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，也就是平时说的同步方法和同步语句块。如果再细的分类，synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。

在进一步阐述之前，我们需要明确几点：

A. 无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁——而且同步方法很可能还

会被其他线程的对象访问。

B．每个对象只有一个锁（lock）与之相关联。

C．实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

接着来讨论synchronized用到不同地方对代码产生的影响：

假设P1、P2是同一个类的不同对象，这个类中定义了以下几种情况的同步块或同步方法，P1、P2就都可以调用它们。

1． 把synchronized当作函数修饰符时，示例代码如下：

```
Public synchronized void methodAAA()
{
//….
}
```

这也就是同步方法，那这时synchronized锁定的是哪个对象呢？它锁定的是调用这个同步方法对象。也就是说，当一个对象P1在不同的线程中执行这个同步方法时，它们之间会形成互斥，达到同步的效果。但是这个对象所属的Class所产生的另一对象P2却可以任意调用这个被加了synchronized关键字的方法。

上边的示例代码等同于如下代码：

```
public void methodAAA()
{
synchronized (this)      //  (1)
{
      //…..
}
}
```

(1)处的this指的是什么呢？它指的就是调用这个方法的对象，如P1。可见同步方法实质是将synchronized作用于object reference。——那个拿到了P1对象锁的线程，才可以调用P1的同步方法，而对P2而言，P1这个锁与它毫不相干，程序也可能在这种情形下摆脱同步机制的控制，造成数据混乱：（

2．同步块，示例代码如下：

```
          public void method3(SomeObject so)
            {
                  synchronized(so)
{
      //…..
}
}
```

这时，锁就是so这个对象，谁拿到这个锁谁就可以运行它所控制的那段代码。当有一个明确的对象作为锁时，就可以这样写程序，但当没有明确的对象作为锁，只是想让一段代码同步时，可以创建一个特殊的instance变量（它得是一个对象）来充当锁：

```
class Foo implements Runnable
{
      private byte[] lock = new byte[0];  // 特殊的instance变量
    Public void methodA()
{
      synchronized(lock) { //… }
}
//…..
}
```

注：零长度的byte数组对象创建起来将比任何对象都经济——查看编译后的字节码：生成零长度的byte[]对象只需3条操作码，而Object lock = new Object()则需要7行操作码。

3．将synchronized作用于static 函数，示例代码如下：

```
    Class Foo
{
public synchronized static void methodAAA()  // 同步的static 函数
{
//….
}
public void methodBBB()
{
     synchronized(Foo.class)  //  class literal(类名称字面常量)
}
    }
```

代码中的methodBBB()方法是把class literal作为锁的情况，它和同步的static函数产生的效果是一样的，取得的锁很特别，是当前调用这个方法的对象所属的类（Class，而不再是由这个Class产生的某个具体对象了）。

记得在《Effective Java》一书中看到过将 Foo.class和 P1.getClass()用于作同步锁还不一样，不能用P1.getClass()来达到锁这个Class的目的。P1指的是由Foo类产生的对象。

可以推断：如果一个类中定义了一个synchronized的static函数A，也定义了一个synchronized 的instance函数B，那么这个类的同一对象Obj在多线程中分别访问A和B两个方法时，不会构成同步，因为它们的锁都不一样。A方法的锁是Obj这个对象，而B的锁是Obj所属的那个Class。

1、线程同步的目的是为了保护多个线程反问一个资源时对资源的破坏。

2、线程同步方法是通过锁来实现，每个对象都有切仅有一个锁，这个锁与一个特定的对象关联，线程一旦获取了对象锁，其他访问该对象的线程就无法再访问该对象的其他非同步方法。

3、对于静态同步方法，锁是针对这个类的，锁对象是该类的Class对象。静态和非静态方法的锁互不干预。一个线程获得锁，当在一个同步方法中访问另外对象上的同步方法时，会获取这两个对象锁。

4、对于同步，要时刻清醒在哪个对象上同步，这是关键。

5、编写线程安全的类，需要时刻注意对多个线程竞争访问资源的逻辑和安全做出正确的判断，对“原子”操作做出分析，并保证原子操作期间别的线程无法访问竞争资源。

6、当多个线程等待一个对象锁时，没有获取到锁的线程将发生阻塞。

7、死锁是线程间相互等待锁锁造成的，在实际中发生的概率非常的小。真让你写个死锁程序，不一定好使，呵呵。但是，一旦程序发生死锁，程序将死掉。

九、线程数据传递

在传统的同步开发模式下，当我们调用一个函数时，通过这个函数的参数将数据传入，并通过这个函数的返回值来返回最终的计算结果。但在多线程的异步开发模式下，数据的传递和返回和同步开发模式有很大的区别。由于线程的运行和结束是不可预料的，因此，在传递和返回数据时就无法象函数一样通过函数参数和return语句来返回数据。

9.1、通过构造方法传递数据

在创建线程时，必须要建立一个Thread类的或其子类的实例。因此，我们不难想到在调用start方法之前通过线程类的构造方法将数据传入线程。并将传入的数据使用类变量保存起来，以便线程使用(其实就是在run方法中使用)。下面的代码演示了如何通过构造方法来传递数据：

```
package mythread;
public class MyThread1 extends Thread
{
private String name;
public MyThread1(String name)
{
this.name = name;
}
public void run()
{
System.out.println("hello " + name);
}
public static void main(String[] args)
{
Thread thread = new MyThread1("world");
thread.start();
}
}
```

由于这种方法是在创建线程对象的同时传递数据的，因此，在线程运行之前这些数据就就已经到位了，这样就不会造成数据在线程运行后才传入的现象。如果要传递更复杂的数据，可以使用集合、类等数据结构。使用构造方法来传递数据虽然比较安全，但如果要传递的数据比较多时，就会造成很多不便。由于Java没有默认参数，要想实现类似默认参数的效果，就得使用重载，这样不但使构造方法本身过于复杂，又会使构造方法在数量上大增。因此，要想避免这种情况，就得通过类方法或类变量来传递数据。

9.2、通过变量和方法传递数据

向对象中传入数据一般有两次机会，第一次机会是在建立对象时通过构造方法将数据传入，另外一次机会就是在类中定义一系列的public的方法或变量（也可称之为字段）。然后在建立完对象后，通过对象实例逐个赋值。下面的代码是对MyThread1类的改版，使用了一个setName方法来设置 name变量：

```
package mythread;
public class MyThread2 implements Runnable
{
private String name;
public void setName(String name)
{
this.name = name;
}
public void run()
{
System.out.println("hello " + name);
}
public static void main(String[] args)
{
MyThread2 myThread = new MyThread2();
```

```
myThread.setName("world");
Thread thread = new Thread(myThread);
thread.start();
}
}
```

9.3、通过回调函数传递数据

上面讨论的两种向线程中传递数据的方法是最常用的。但这两种方法都是main方法中主动将数据传入线程类的。这对于线程来说，是被动接收这些数据的。然而，在有些应用中需要在线程运行的过程中动态地获取数据，如在下面代码的run方法中产生了3个随机数，然后通过Work类的process方法求这三个随机数的和，并通过Data类的value将结果返回。从这个例子可以看出，在返回value之前，必须要得到三个随机数。也就是说，这个 value是无法事先就传入线程类的。

```
package mythread;
class Data
{
public int value = 0;
}
class Work
{
public void process(Data data, Integer numbers)
{
for (int n : numbers)
{
data.value += n;
}
}
}
public class MyThread3 extends Thread
{
private Work work;
public MyThread3(Work work)
{
this.work = work;
}
public void run()
{
java.util.Random random = new java.util.Random();
Data data = new Data();
int n1 = random.nextInt(1000);
int n2 = random.nextInt(2000);
int n3 = random.nextInt(3000);
work.process(data, n1, n2, n3); // 使用回调函数
System.out.println(String.valueOf(n1) + "+" + String.valueOf(n2) + "+"
+ String.valueOf(n3) + "=" + data.value);
}
public static void main(String[] args)
{
```

```
Thread thread = new MyThread3(new Work());
thread.start();
}
}
```