PDF-XChange
Click to buy NOW!
www.docu-track.com

多线程学习总结

首先，让我们来明白几个概念

进程： 正在执行的一个独立的程序，对于单 CPU 来说，进程在宏观上是并发的，但在微观上是串发的！

线程： 在一个进程内部的‘可以并发执行的多个函数’， 一个进程中可有 N 个线程

对于进程，每一个进程都是独占数据空间的，（是一个程序的执行流程）。

而对于线程，它生活在进程内部，它们共享进程空间，线程是由操作系统（OS）来维护的。

那么在 JAVA 中，如何来创建一个线程？

有两种方式可以选取择：

1. 一种方法是将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。

```
如： class ThreadA extends Thread {
        Private String name;
        Public ThreadA(String name) { this.name = name; }
        Public void run() {
            //具体的代码
        }
    }
```

2. 创建线程的另一种方法是声明实现 Runnable 接口的类。该类然后实现 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递。

```
如： class ThreadB implements Runnable {
        ....
        Public void run() {
            //具体的代码
        }
    }
```

接下来可以来分别创建一个线程类对像

```
....
Thread a = new ThreadA(“a”); //创建一个线程
a.start(); //启动一个线程

Runnable r = new ThreadB(); //创建一个 Runnable 的实例 r
Thread b = new Thread(r); //使用一个 Runnable 的实例来创建一个线程
b.start(); //启动一个线程
```

由上可以看出，在 JAVA 中，线程也是一个类（Thread），实际的线程就是一个对像。

现在我们得要弄清楚一个概念：

线程与线程对像可不是一个概念。

EX： Thread t = new Thread();

t 代表了一个线程，它只是一个线程对像，而不是线程，它可以到操作系统（OS）底层去初始化一个线程， 既调用 t.start() 方法。

JAVA 虚拟机是一个虚拟机进程，所有的 JAVA 程序都是跑在此虚拟机进程中的线程，由 OS 来调度！ 实际上，JAVA 程序的 MAIN 方法就是一个主线程，一旦运行一个 JAVA 程序，主

PDF-XChange Click to buy NOW! www.docu-track.com

线程就会开始运行。所以有人说 JAVA 是基于线程运行的！
注： 虚拟机进程要等到所在非守护线程都结束了，它才会结束。在 JAVA 虚拟机中有一个垃圾回收线程，就是一个守护线程。
再次早明： 实现 Runnable 接口的类不是一个线程类，只能是用它去构造一个线程对像。

注意： 线程，对像（线程对像）， 方法 之间的关系
线程： 哪个线程 ？
对像： 哪个线程对像？
方法： 哪个方法？
如 在 main() 中创建成了一个 Thread 类对像 t, 并调用了 t.start()。//表示： 主线程对 t 线程对像调用了 start() 方法。

例子：

```
Public class TestThread {
    Public static void main(String[] args) {
        Thread t1 = new MyThread();
        Runnable r = new YouThread();
        Thread t2 = new Thread(r);
        t1.start(); //启动 t1 线程
        t2.start();//启动 t2 线程
                // main 线程结束。
    }
}
Class MyThread extends Thread {
    Public void run() {
        For(int i=0;i<1000;i++)
            System.out.println( “ #### “ );
    }
}
Class YouThread implements Runnable {
    Public void run() {
        For(int i=0;i<1000;i++)
            System.out.println( “￥￥￥ ” );
    }
}
```

上例中，主线程 main 的任务就是启动 t1 和 t2 线程，在 t2.start() 后它就结束了。但可能 t1,t2 还在运行，所以进程不会退出。
注： 虚拟机进程要等到所在非守护线程都结束了，它才会结束.

Thread 类
Thread 类位于 java.lang 包中，有一方法可以设置线程的优先级。
setPriority(int i); 它的参数为一个整数。它有三个值：

Thread.MAX_PRIORITY	最高优先级	10
Thread.MIN_PRIORITY	最低优先级	1

Thread.NORM_PRIORITY　　一般优先级　　5

注：依平台不同，OS 对优先级的处理方式不一样，而且划分等级也可能不一样，所以这种方式不能保证线程之间的执行顺序，不鼓励使用。

线程间的各种状态的发生和如何来控制线程的状态？

1. 方法 yield() 只要一个线程调用了 yield()方法。它就会立既放弃 CPU 的时间片，从运行直接进入可运行态。!
2. 阻塞态

   有三种方式可导致一个线程从运行态变到阻塞态

   A. 等待数据输入。如 I/O 流中的 一些 read 方法

   B. 调用了 sleep() 方法。休眠指定时间

   C. 对另一个线程调用了 join() 方法。如 t2.join()。则表示当前线程要等到 t2 结束后才会打破阻塞。

   相应的， 要从阻塞态变到可运行态。

   A. 数据输入结束

   B. Sleep()方法所休眠时间到达

   C. 直到 t2 线程运行结束。

注： sleep() 方法会抛出一个 InterruptedException 异常，如果在 run() 方法中调用了 sleep() 方法， 则此异常是不能抛出的，因为父类或 Runnable 接口的 run() 都没有抛出异常，根据子方法绝不能比父类方法抛出更多的例外原则，所以必需使用 try . .. catch 做出处理。

Join()方法可以说是把两个线程合并成了一个线程，它也会抛出一个 InterruptedException 异常.

3. 线程同步

当一个或多个线程并发地访问同一共享资源的时候，就可能会造成共享资源的原子操作分开被执行，这样的话就会导致共享资源产生的数据不一致或数据不完整的情况，这种共享资源被称着临界资源。!

那么如何来解决这个问题呢？？？

同步锁机制。

现在，我们来回忆一下对像。我们之前所说的对象，拥有属性和方法。现在，我们要给它增加点东西了。就是： 一个对像除了属性和方法外，还拥有互斥锁，锁池，和等待队列。

既然每一个对象都有一把锁，如果把这个锁分配给了一个线程，则在此线程使用期间，其它线程就不能获得这个锁资源了，直到那个线程释放了这个锁资源。

注： 锁，锁池，等待队列是对象自有的，用来分配给线程使用的！ 简单变量是没有的！

实现同步的关键字： synchronized

它可以修饰代码块和方法。

1. 修饰方法： 就是对当前对像加锁，也叫内同步法。

   如: synchronized void m() { // 凡是要进入执行此代码块的线程，都必需拿到此对象
   . . . .// 的锁，否则此线程进入阻塞（对象的锁池），拿到锁的
   . . . .// 线程执行完后会释放锁资源（OS 分配）
   }

2. 修饰代码块： 对‘对象’加锁， 也叫外同步法。

如：

```
Void  m ()  {
    . . . //
    synchronized (对象名) { // (this: 也就表示对当前对象加锁，这与内同步是一样的
            //代码块
    }
    . . . //
}
```

由上可以看出，使用外同步加锁拥有更多的灵活性，可以使用不同对象的锁进行操作。内同步只能对当前对象加锁。

*** 静态方法允许加 synchronized ，但构造方法不允许，因为没有对象。

现在，有了上面的细致分析后，让我们来回顾一下: Verctor 和 ArrayList 这两个集合类的区别：

Vertor 是线程安全的，因为 Vector 中的所有方法都是用 synchronized 修饰的，所以才能保证它是线程安全的，但性能不高。

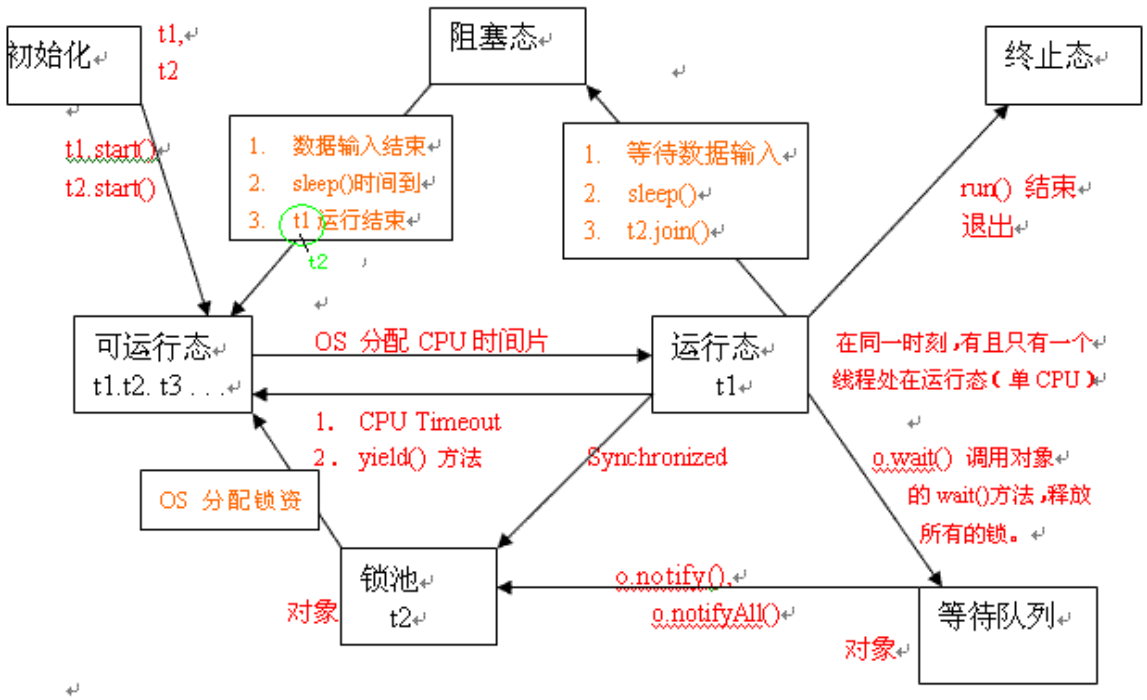
ArrayList 是线程不安全的，因为它的方法都不是 synchronized 修饰的，所以如果要实现 ArrayList 的线程安全，可以采用外同步法：

```
如：  . . .      . . .
        Synchronized (obj) {
                Obj.add(“234”);
                . . . . . .
        }  // 同样可以实现 代码块的线程安全
```

? 这样有人就要问了，那我们为什么要用 ArrayList 来代替 Vertor 呢？

--- 这是因为加锁是要消耗宝贵资源的，消耗 CPU 时间，自然效率就降低了，所以应该对该加锁的加锁，不该加锁的绝不要加》!!

现在，我们该画画线程状态图了！如下：



PDF-XChange Click to buy NOW! www.docu-track.com

注： notify() 和 notifyAll() 方法只纯性通知其它在等待队列中的线程，只是把它们‘唤醒‘，状态转到锁池，它不会让线程放弃任何的锁资源。

注： 拥有此对象的锁的线程进入到其它对象的锁池是不会放弃自己拥有的锁资源的，这也是产生死锁的原因。

3. 锁池状态

也是一种特殊的阻塞状态，

当某个线程要访问已加锁的临界资源，而又没有此对象的锁，则此线程就会进入对象的锁池，等待 OS 的调配，来得到锁标记。

4. 等待队列

也是一程特殊的阻塞！

当某个线程调用了对象的 wait() 方法，那么此线程就会释放它所拥有的所有锁标记，并自动进入这个对象的等待队列，直到有另一个线程调用了对象的 notify() 或 notifyAll()方法，则此线程才会从对象的等待队列中进入到锁池中，等待 OS 来分配锁标记。

注：wait() 和 notifyAll()方法是进程间通信必不可少的，当多个线程同时对临界资源进行操作时，就必需要相互协调，以达到正确运行的目的！

通常，线程的同步（synchronzied）和通信（wait(); notifyAll()）可以保证多线程程序的正常驻执行！

现在，让我们来写一个多线程同步的例子

```
Public class TestThread {
    Public static void main(String[] args) {
        Object obj = new Object();//生成一个对象，供线程使用。
        For(int i=0;i<5;i++) { //循环几次，就生成几个线程。
            New MyThread(obj);
        }
    }
}
Class MyThread extends Thread {
    Private static int id = 0;
    Private static int number = 10;
    Private Object obj;
    Public MyThread(Object obj) {
        Super(“”+ ++id);
        This.obj = obj;
        Start(); //在构造时启动
    }
    Public String toString() {
        Return “THREAD#”+getName() ;//获得线程名
    }
    Public void run() {
        Synchronized(obj) {//对传进来的对象加锁，以保证此对象对所有线程共享。
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
            While(number <= 100) {
                System.out.println(this+" print-> "+number);
                Number += 10;
                Obj.notifyAll();
                System.out.println("It's turn to next Thread. I wait . . .");
                If(number <= 100) {
                    Try {
                        Obj.wait();
                        Sleep(1000);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
}
```

最后一个问题： 死锁

产生死锁的原因： 在不释放自己已有的锁标记的情况下，要求得到互相对象的锁标记，从而造成死锁。

线程间的通信：

例子：

一个线程打印数：（1-52），另一个线程打印字母：（A-Z）。要求以如下格式输出：12A34B56C。 。 。 5152Z。 采用线程间通信完成，不可造成死锁。

```
Public class TestNumber {
    Public static void main(String[] args) {
        Object o = new Object();
        Thread t1 = new ThreadA(o);
        Thread t2 = new ThreadB(o);
        t1.start();
        t2.start();
    }
}
Class ThreadA extends Thread {
    Private Object o;
    Public ThreadA(Object o) {
        This.o = o;
    }
    Public void run() {
        Synchronized(o) {
            For(int i=0;i<=26;i++) {
                System.out.print(i*2-1);
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
                System.out.print(i*2);
                o.notifyAll();
                try {
                    if(I != 26) o.wait() ;
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
    Class ThreadB extends Thread {
        Private Object o;
        Public ThreadB(Object o){
            This.o = o;
        }
        Public void run() {
            Synchronized(o) {
                For(char c = 'A';c <= 'Z';c++) {
                    System.out.print(c);
                    o.notifyAll();
                    try {
                        if(c != 'Z')
                            o.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

额外知识：

我们知道，Thread 类中有一个 interrupt() 方法，可以终止线程，对一个线程调用了 interrupt() 方法，会使 wait() 或 sleep() 等方法产生一个 InterruptedException 异常，所以，可以利用一个标志位，来控制一个线程终止！

例子：

```
Public class TestInterrupt {
    Public static void main(String[] args) {
        Threard t1 = new Thread();
        t.start();
        int index = 0;
        while(true) {
            if(index++ == 500) {
                t1.stopThread();
```

```
                    t1.interrupt();
                    break;
                }
                System.out.println(Thread.currentThread().getName());
            }
            System.out.println("main() exit");
        }
    }
Class Thread1 extends Thread {
    Private boolean bstop = false;
    Public synchronized void run() {
        While(!bstop) {
            Try {
                //System.out.println(". . . .");
                Wait();
            } Catch(InterruptedException e) {
                    e.printStackTrace();
                    if(bstop) return ; //终止线程
            }
            System.out.println(getName());
        }
    }
    Public void stopThread() {
        bstop = true;
    }
}
```

到此， 我们的多线程总结完成，希望对大家有所帮助，谢谢！

作者： 叶加飞 (Steven Ye)
mailto: leton.ye@gmail.com