

## Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

Java并发编程系列：

- Java 并发编程：核心理论
- Java并发编程：Synchronized及其实现原理
- Java并发编程：Synchronized底层优化（轻量级锁、偏向锁）
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join).
- Java 并发编程：volatile的使用及其原理

### 一、线程的状态

Java中线程中状态可分为五种：New（新建状态），Runnable（就绪状态），Running（运行状态），Blocked（阻塞状态），Dead（死亡状态）。

New：新建状态，当线程创建完成时为新建状态，即new Thread(...)，还没有调用start方法时，线程处于新建状态。

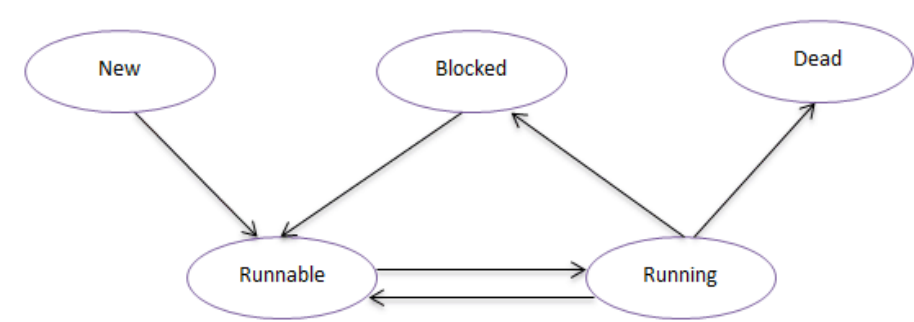
Runnable：就绪状态，当调用线程的的start方法后，线程进入就绪状态，等待CPU资源。处于就绪状态的线程由Java运行时系统的线程调度程序(thread scheduler)来调度。

Running：运行状态，就绪状态的线程获取到CPU执行权以后进入运行状态，开始执行run方法。

Blocked：阻塞状态，线程没有执行完，由于某种原因（如，I/O操作等）让出CPU执行权，自身进入阻塞状态。

Dead：死亡状态，线程执行完成或者执行过程中出现异常，线程就会进入死亡状态。

这五种状态之间的转换关系如下图所示：



有了对这五种状态的基本了解，现在我们来看看Java中是如何实现这几种状态的转换的。

### 二、wait/notify/notifyAll方法的使用

#### 1、wait方法：

void wait()	Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void wait(long timeout)	Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void wait(long timeout, int nanos)	Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

JDK中一共提供了这三个版本的方法，

（1）wait()方法的作用是将当前运行的线程挂起（即让其进入阻塞状态），直到notify或notifyAll方法来唤醒线程。

（2）wait(long timeout)，该方法与wait()方法类似，唯一的区别就是在指定时间内，如果没有notify或notifAll方法的唤醒，也会自动唤醒。

（3）至于wait(long timeout,long nanos)，本意在于更精确的控制调度时间，不过从目前版本来看，该方法貌似没有完整的实现该功能，其源码(JDK1.8)如下：

```
1 public final void wait(long timeout, int nanos) throws InterruptedException {
2     if (timeout < 0) {
3         throw new IllegalArgumentException("timeout value is negative");
4     }
```

#### 公告

昵称：liuxiaopeng  
园龄：2年6个月  
粉丝：249  
关注：2  
[+加关注](#)

#### 搜索

找找看

谷歌搜索

#### 最新随笔

- Spring Boot实战：拦截器与过滤器
- Spring Boot实战：静态资源处理
- Spring Boot实战：集成Swagger2
- Spring Boot实战：Restful API的构建
- Spring Boot实战：数据库操作
- Spring Boot实战：逐行释义HelloWorld
- Java集合类：AbstractCollection源码解析
- Java集合：整体结构
- Java 并发编程：volatile的使用及其原理
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

#### 我的标签

- Java(15)
- Spring(7)
- spring boot(7)
- 并发编程(4)

```
5
6     if (nanos < 0 || nanos > 999999) {
7         throw new IllegalArgumentException(
8             "nanosecond timeout value out of range");
9     }
10
11     if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {
12         timeout++;
13     }
14
15     wait(timeout);
16 }
```

从源码来看，JDK8中对纳秒的处理，只做了四舍五入，所以还是按照毫秒来处理的，可能在未来的某个时间点会用到纳秒级别的精度。虽然JDK提供了这三个版本，其实最后都是调用wait(long timeout)方法来实现的，wait()方法与wait(0)等效，而wait(long timeout,int nanos)从上面的源码可以看到也是通过wait(long timeout)来完成的。下面我们通过一个简单的例子来演示wait()方法的使用：

```
1 package com.paddx.test.concurrent;
2
3 public class WaitTest {
4
5     public void testWait(){
6         System.out.println("Start-----");
7         try {
8             wait(1000);
9         } catch (InterruptedException e) {
10             e.printStackTrace();
11         }
12         System.out.println("End-----");
13     }
14
15     public static void main(String[] args) {
16         final WaitTest test = new WaitTest();
17         new Thread(new Runnable() {
18             @Override
19             public void run() {
20                 test.testWait();
21             }
22         }).start();
23     }
24 }
```

这段代码的意图很简单，就是程序执行以后，让其暂停一秒，然后再执行。运行上述代码，查看结果：

```
Start-----
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at com.paddx.test.concurrent.WaitTest.testWait(WaitTest.java:8)
    at com.paddx.test.concurrent.WaitTest$1.run(WaitTest.java:20)
    at java.lang.Thread.run(Thread.java:745)
```

这段程序并没有按我们的预期输出相应结果，而是抛出了一个异常。大家可能会觉得奇怪为什么会抛出异常？而抛出的IllegalMonitorStateException异常又是什么？我们可以看一下JDK中对IllegalMonitorStateException的描述：

Thrown to indicate that a thread has attempted to wait on an object's **monitor or to notify other threads waiting on an object's** monitor without owning the specified monitor.

这句话的意思大概就是：线程试图等待对象的监视器或者试图通知其他正在等待对象监视器的线程，但本身没有对应的监视器的所有权。其实这个问题在《[Java并发编程：Synchronized及其实现原理](#)》一文中有提到过，wait方法是一个本地方法，其底层是通过一个叫做监视器锁的对象来完成的。所以上面之所以会抛出异常，是因为在调用wait方式时没有获取到monitor对象的所有权，那如何获取monitor对象所有权？Java中只能通过Synchronized关键字来完成，修改上述代码，增加Synchronized关键字：

```
1 package com.paddx.test.concurrent;
2
3 public class WaitTest {
4
5     public synchronized void testWait()//增加Synchronized关键字
6     {
7         System.out.println("Start-----");
8         try {
9             wait(1000);
10        } catch (InterruptedException e) {
11            e.printStackTrace();
12        }
13        System.out.println("End-----");
14    }
15
16    public static void main(String[] args) {
17        final WaitTest test = new WaitTest();
18        new Thread(new Runnable() {
19            @Override
20            public void run() {
21                test.testWait();
22            }
23        }).start();
24    }
25 }
```

Rest(2)
Restful(1)
过滤器(1)
集合框架(1)
静态资源(1)
拦截器(1)
更多

随笔档案
2018年1月 (5)
2017年12月 (1)
2016年6月 (1)
2016年5月 (3)
2016年4月 (4)
2016年3月 (3)

积分与排名
积分 - 40237
排名 - 9732

最新评论
1. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）  博主写得很棒  --还好可以改名字
2. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）  @就这个名引用在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。针对这句话有个疑问，如果我能保.....  --还好可以改名字
3. Re:Java8内存模型—永久代(PermGen)和元空间(Metaspace)  感谢分享，写的很好！  --我不将就
4. Re:Spring Boot实战：集成Swagger 2  学习了 菜鸟飘过~~~  --祎祎家斌斌

```
22         }).start();
23     }
24 }
```



现在再运行上述代码，就能看到预期的效果了：

```
Start-----
End-----
```

所以，通过这个例子，大家应该很清楚，wait方法的使用必须在同步的范围内，否则就会抛出IllegalMonitorStateException异常，wait方法的作用就是阻塞当前线程等待notify/notifyAll方法的唤醒，或等待超时后自动唤醒。

## 2、notify/notifyAll方法

void notify()	Wakes up a single thread that is waiting on this object's monitor.
void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

有了对wait方法原理的理解，notify方法和notifyAll方法就很容易理解了。既然wait方式是通过对象的monitor对象来实现的，所以只要在同一对象上去调用notify/notifyAll方法，就可以唤醒对应对象monitor上等待的线程了。notify和notifyAll的区别在于前者只能唤醒monitor上的一个线程，对其他线程没有影响，而notifyAll则唤醒所有的线程，看下面的例子很容易理解这两者的差别：



```
1 package com.paddx.test.concurrent;
2
3 public class NotifyTest {
4     public synchronized void testWait(){
5         System.out.println(Thread.currentThread().getName() +" Start-----");
6         try {
7             wait(0);
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        System.out.println(Thread.currentThread().getName() +" End-----");
12    }
13
14    public static void main(String[] args) throws InterruptedException {
15        final NotifyTest test = new NotifyTest();
16        for(int i=0;i<5;i++) {
17            new Thread(new Runnable() {
18                @Override
19                public void run() {
20                    test.testWait();
21                }
22            }).start();
23        }
24
25        synchronized (test) {
26            test.notify();
27        }
28        Thread.sleep(3000);
29        System.out.println("-----分割线-----");
30
31        synchronized (test) {
32            test.notifyAll();
33        }
34    }
35 }
```



输出结果如下：

```
Thread-0 Start-----
Thread-1 Start-----
Thread-2 Start-----
Thread-3 Start-----
Thread-4 Start-----
Thread-0 End-----
-----分割线-----
Thread-4 End-----
Thread-3 End-----
Thread-2 End-----
Thread-1 End-----
```

从结果可以看出：调用notify方法时只有线程Thread-0被唤醒，但是调用notifyAll时，所有的线程都被唤醒了。

最后，有两点点需要注意：

- （ 1 ）调用wait方法后，线程是会释放对monitor对象的所有权的。
- （ 2 ）一个通过wait方法阻塞的线程，必须同时满足以下两个条件才能被真正执行：

- 线程需要被唤醒（ 超时唤醒或调用notify/notifyll ）。
- 线程唤醒后需要竞争到锁（ monitor ）。

## 三、sleep/yield/join方法解析

上面我们已经清楚了wait和notify方法的使用和原理，现在我们再来看另外一组线程间协作的方法。这组方法跟上面方法的最明显区别是：这几个方法都位于Thread类中，而上面三个方法都位于Object类中。至于为什么，大家可以先思考一下。现在我们逐个分析sleep/yield/join方法：

5. Re:Spring Boot实战：拦截器与过滤器

这两个使用场景有啥区别？

--四度空间的平面

### 阅读排行榜

1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(52682)

2. Java并发编程：Synchronized及其实现原理(43925)

3. Java 并发编程：volatile的使用及其原理(24437)

4. Java 并发编程：核心理论(22467)

5. Java并发编程：Synchronized底层优化（ 偏向锁、轻量级锁 ）(21924)

### 评论排行榜

1. Java并发编程：Synchronized及其实现原理(21)

2. Java 并发编程：volatile的使用及其原理(16)

3. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(13)

4. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(12)

5. 通过反编译深入理解Java String及intern(10)

### 推荐排行榜

1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(36)

2. Java并发编程：Synchronized及其实现原理(33)

3. 从字节码层面看“HelloWorld”(26)

4. Java 并发编程：核心理论(21)

5. Spring Boot实战：逐行释义HelloWorld(15)

1、sleep

sleep方法的作用是让当前线程暂停指定的时间（毫秒），sleep方法是最简单的方法，在上述的例子中也用到过，比较容易理解。唯一需要注意的是其与wait方法的区别。最简单的区别是，wait方法依赖于同步，而sleep方法可以直接调用。而更深层次的区别在于sleep方法只是暂时让出CPU的执行权，并不释放锁。而wait方法则需要释放锁。



```
1 package com.paddx.test.concurrent;
2
3 public class SleepTest {
4     public synchronized void sleepMethod(){
5         System.out.println("Sleep start-----");
6         try {
7             Thread.sleep(1000);
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        System.out.println("Sleep end-----");
12    }
13
14    public synchronized void waitMethod(){
15        System.out.println("Wait start-----");
16        synchronized (this){
17            try {
18                wait(1000);
19            } catch (InterruptedException e) {
20                e.printStackTrace();
21            }
22        }
23        System.out.println("Wait end-----");
24    }
25
26    public static void main(String[] args) {
27        final SleepTest test1 = new SleepTest();
28
29        for(int i = 0;i<3;i++){
30            new Thread(new Runnable() {
31                @Override
32                public void run() {
33                    test1.sleepMethod();
34                }
35            }).start();
36        }
37
38
39        try {
40            Thread.sleep(10000);//暂停十秒，等上面程序执行完成
41        } catch (InterruptedException e) {
42            e.printStackTrace();
43        }
44        System.out.println("-----分割线-----");
45
46        final SleepTest test2 = new SleepTest();
47
48        for(int i = 0;i<3;i++){
49            new Thread(new Runnable() {
50                @Override
51                public void run() {
52                    test2.waitMethod();
53                }
54            }).start();
55        }
56
57    }
58 }
```



执行结果：

Sleep start-----  
Sleep end-----  
Sleep start-----  
Sleep end-----  
Sleep start-----  
Sleep end-----  
-----分割线-----  
Wait start-----  
Wait start-----  
Wait start-----  
Wait end-----  
Wait end-----  
Wait end-----

这个结果的区别很明显，通过sleep方法实现的暂停，程序是顺序进入同步块的，只有当上一个线程执行完成的时候，下一个线程才能进入同步方法，sleep暂停期间一直持有monitor对象锁，其他线程是不能进入的。而wait方法则不同，当调用wait方法后，当前线程会释放持有的monitor对象锁，因此，其他线程还可以进入到同步方法，线程被唤醒后，需要竞争锁，获取到锁之后再继续执行。

2、yield方法

yield方法的作用是暂停当前线程，以便其他线程有机会执行，不过不能指定暂停的时间，并且也不能保证当前线程马上停止。yield

方法只是将Running状态转变为Runnable状态。我们还是通过一个例子来演示其使用：



```
1 package com.paddx.test.concurrent;
2
3 public class YieldTest implements Runnable {
4     @Override
5     public void run() {
6         try {
7             Thread.sleep(100);
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        for(int i=0;i<5;i++){
12            System.out.println(Thread.currentThread().getName() + ": " + i);
13            Thread.yield();
14        }
15    }
16
17    public static void main(String[] args) {
18        YieldTest runn = new YieldTest();
19        Thread t1 = new Thread(runn,"FirstThread");
20        Thread t2 = new Thread(runn,"SecondThread");
21
22        t1.start();
23        t2.start();
24
25    }
26 }
```



运行结果如下：

FirstThread: 0  
SecondThread: 0  
FirstThread: 1  
SecondThread: 1  
FirstThread: 2  
SecondThread: 2  
FirstThread: 3  
SecondThread: 3  
FirstThread: 4  
SecondThread: 4

这个例子就是通过yield方法来实现两个线程的交替执行。不过请注意：这种交替并不一定能得到保证，源码中也对这个问题进行说明：

```
/**
 * A hint to the scheduler that the current thread is willing to yield
 * its current use of a processor. The scheduler is free to ignore this
 * hint.
 *
 * <p> Yield is a heuristic attempt to improve relative progression
 * between threads that would otherwise over-utilise a CPU. Its use
 * should be combined with detailed profiling and benchmarking to
 * ensure that it actually has the desired effect.
 *
 * <p> It is rarely appropriate to use this method. It may be useful
 * for debugging or testing purposes, where it may help to reproduce
 * bugs due to race conditions. It may also be useful when designing
 * concurrency control constructs such as the ones in the
 * {@link java.util.concurrent.locks} package.
 */
```


这段话主要说明了三个问题：

- 调度器可能会忽略该方法。
- 使用的时候要仔细分析和测试，确保能达到预期的效果。
- 很少有场景要用到该方法，主要使用的地方是调试和测试。


3、join方法

void join()	Waits for this thread to die.
void join(long millis)	Waits at most millis milliseconds for this thread to die.
void join(long millis, int nanos)	Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

join方法的作用是父线程等待子线程执行完成后再执行，换句话说就是将异步执行的线程合并为同步的线程。JDK中提供三个版本的join方法，其实现与wait方法类似，join()方法实际上执行的join(0)，而join(long millis, int nanos)也与wait(long millis, int nanos)的实现方式一致，暂时对纳秒的支持也是不完整的。我们可以看下join方法的源码，这样更容易理解：



```
1 public final void join() throws InterruptedException {
2     join(0);
3 }
4
5 public final synchronized void join(long millis)
6     throws InterruptedException {
```



```
7         long base = System.currentTimeMillis();
8         long now = 0;
9
10        if (millis < 0) {
11            throw new IllegalArgumentException("timeout value is negative");
12        }
13
14        if (millis == 0) {
15            while (isAlive()) {
16                wait(0);
17            }
18        } else {
19            while (isAlive()) {
20                long delay = millis - now;
21                if (delay <= 0) {
22                    break;
23                }
24                wait(delay);
25                now = System.currentTimeMillis() - base;
26            }
27        }
28    }
29
30    public final synchronized void join(long millis, int nanos)
31        throws InterruptedException {
32
33        if (millis < 0) {
34            throw new IllegalArgumentException("timeout value is negative");
35        }
36
37        if (nanos < 0 || nanos > 999999) {
38            throw new IllegalArgumentException(
39                "nanosecond timeout value out of range");
40        }
41
42        if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
43            millis++;
44        }
45
46        join(millis);
47    }
```



大家重点关注一下join(long millis)方法的实现，可以看出join方法就是通过wait方法来将线程的阻塞，如果join的线程还在执行，则将当前线程阻塞起来，直到join的线程执行完成，当前线程才能执行。不过有一点需要注意，这里的join只调用了wait方法，却没有对应的notify方法，原因是Thread的start方法中做了相应的处理，所以当join的线程执行完成以后，会自动唤醒主线程继续往下执行。下面我们通过一个例子来演示join方法的作用：

( 1 ) 不使用join方法：



```
1 package com.paddx.test.concurrent;
2
3 public class JoinTest implements Runnable{
4     @Override
5     public void run() {
6
7         try {
8             System.out.println(Thread.currentThread().getName() + " start-----");
9             Thread.sleep(1000);
10            System.out.println(Thread.currentThread().getName() + " end-----");
11        } catch (InterruptedException e) {
12            e.printStackTrace();
13        }
14    }
15
16    public static void main(String[] args) {
17        for (int i=0;i<5;i++) {
18            Thread test = new Thread(new JoinTest());
19            test.start();
20        }
21
22        System.out.println("Finished~~~");
23    }
24 }
```



执行结果如下：

```
Thread-0 start-----
Thread-1 start-----
Thread-2 start-----
Thread-3 start-----
Finished~~~
Thread-4 start-----
Thread-2 end-----
Thread-4 end-----
Thread-1 end-----
Thread-0 end-----
```



Thread-3 end-----

( 2 ) 使用join方法 :



```
1 package com.paddx.test.concurrent;
2
3 public class JoinTest implements Runnable{
4     @Override
5     public void run() {
6
7         try {
8             System.out.println(Thread.currentThread().getName() + " start-----");
9             Thread.sleep(1000);
10            System.out.println(Thread.currentThread().getName() + " end-----");
11        } catch (InterruptedException e) {
12            e.printStackTrace();
13        }
14    }
15
16    public static void main(String[] args) {
17        for (int i=0;i<5;i++) {
18            Thread test = new Thread(new JoinTest());
19            test.start();
20            try {
21                test.join(); //调用join方法
22            } catch (InterruptedException e) {
23                e.printStackTrace();
24            }
25        }
26
27        System.out.println("Finished~~~");
28    }
29 }
```



执行结果如下 :

Thread-0 start-----  
Thread-0 end-----  
Thread-1 start-----  
Thread-1 end-----  
Thread-2 start-----  
Thread-2 end-----  
Thread-3 start-----  
Thread-3 end-----  
Thread-4 start-----  
Thread-4 end-----  
Finished~~~

对比两段代码的执行结果很容易发现，在没有使用join方法之间，线程是并发执行的，而使用join方法后，所有线程是顺序执行的。

## 四、总结

本文主要详细讲解了wait/notify/notifyAll和sleep/yield/join方法。最后回答一下上面提出的问题：wait/notify/notifyAll方法的作用是实现线程间的协作，那为什么这三个方法不是位于Thread类中，而是位于Object类中？位于Object中，也就相当于所有类都包含这三个方法（因为Java中所有的类都继承自Object类）。要回答这个问题，还是得回过来看wait方法的实现原理，大家需要明白的是，wait等待的到底是什么东西？如果对上面内容理解的比较好的话，我相信大家应该很容易知道wait等待其实是对象monitor，由于Java中的每一个对象都有一个内置的monitor对象，自然所有的类都理应有wait/notify方法。

作者：liuxiaopeng

博客地址：<http://www.cnblogs.com/paddix/>

声明：转载请在文章页面明显位置给出原文连接。


好文要顶

关注我

收藏该文







liuxiaopeng  
关注 - 2  
粉丝 - 249

+加关注

« 上一篇：Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）  
» 下一篇：Java 并发编程：volatile的使用及其原理

posted @ 2016-05-04 08:15 liuxiaopeng 阅读(12300) 评论(12) 编辑 收藏

### 评论列表

#1楼 2016-05-04 09:36 JavaNewWorld

终于明白了，赞一个！

支持(0) 反对(0)

#2楼 2016-05-04 10:07 Ryan^大兵

赞一个

支持(0) 反对(0)

#3楼 2016-05-04 12:01 码魂

这个图哪来的?不太对.或者说画的太粗.

支持(0) 反对(0)

#4楼 2016-05-04 12:32 lulipro

正好要看并发编程，顶一个！

支持(0) 反对(0)

#5楼[楼主] 2016-05-04 13:00 liuxiaopeng

@ 码魂  
嗯，这个图只是粗略的表示一下状态之间的转换关系，阻塞可以细分成多种情况~

支持(0) 反对(0)

#6楼 2016-05-05 15:25 Anonymous\_lin

试试lock

支持(0) 反对(0)

#7楼 2016-08-19 10:31 dracularking

“我相信大家应该很容易知道wait等待其实是对象monitor，由于Java中的每一个对象都有一个内置的monitor对象，自然所有的类都理应有wait/notify方法。”

我感觉这种设计从OOAD的角度去理解的话还是有些费解的  
当前的情况是  
object.wait();  
从OOAD角度出发，一般会被理解为 object在wait，但实际上是线程在等待锁，而不是锁在等待，等待什么呢？虽然这样写，看上去很简洁。  
  
我觉得更好的符合OOAD的设计应该是这样的：  
thread.wait(object);//线程等待锁，描述贴切  
期待讨论。

支持(0) 反对(0)

#8楼 2017-03-21 11:42 Brilan

很好的文章，讲得很细，但是有一个疑问

Join方法的作用是：父线程等待子线程完成之后再执行。  
但在在使用join的并发例子中。测试用例里面，父线程是main函数所在线程，子线程应该有5个，同时这5个子线程之间没有父子关系，并发执行的，为何最后会变成顺序执行？这里有点不懂。  
  
父线程会等待5个子线程执行结束在执行，但是5个子线程之间应该会会并列执行，不会顺序执行？求大神解救。

支持(0) 反对(0)

#9楼[楼主] 2017-03-21 22:20 liuxiaopeng

@ Brilan  
这个的主要原因，是因为在循环里先执行了start()，接下来再执行join()，这个时候，循环就暂停了，需要等这次join的线程执行完成才能进入下一个循环，执行下一个线程的start和join，所以就导致顺序执行了。你可以先把5个线程都start了，再执行join，这样可以看到并发执行，但是这样其实也存在问题，因为join没办法并发执行，只有等第一个join的线程执行完才能再执行其他的join操作。

支持(1) 反对(0)

#10楼 2017-03-30 20:08 Stef\_ToBeC

对于刚刚学并发编程的我，觉得你写得太好了，我本来就是一个思维很发散的人，你能解答我目前对于并发的疑惑，太感激了…

支持(0) 反对(0)

#11楼 2017-04-30 17:24 Paranoia\_WXY

博主的文章写的很不错，对初学者来说讲的非常清楚。有个问题想要问一下。

```
1 public synchronized void waitMethod(){
2     System.out.println("Wait start-----");
3     synchronized (this){
4         try {
5             wait(1000);
6         } catch (InterruptedException e) {
7             e.printStackTrace();
8         }
9     }
```



```
10         System.out.println("Wait end-----");
11     }
```

这段代码是在比较sleep和wait方法代码片段中的wait方法，我想问一下方法已经标注了synchronized，为什么里面还用了synchronized代码块。去掉synchronized代码块，最后效果是一样的。这样做有什么用意？

支持(0) 反对(0)

#12楼 2017-11-23 09:28 辛巴国王

卤煮貌似很长时间不更新了，盼勤更文哇~~~

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【活动】2050 大会 - 年青人因科技而团聚（ 5.26-27杭州·云栖小镇 ）

【活动】华为云全新一代云服务器·限时特惠5.6折

【推荐】腾讯云多款高规格服务器，免费申请试用6个月