

# Java多线程系列分享（一）Java多线程基础

## 线程与进程

1 线程： 进程中负责程序执行的执行单元 线程本身依靠程序进行运行 线程是程序中的顺序控制流，只能使用分配给程序的资源和环境

2 进程： 执行中的程序

一个进程至少包含一个线程

3 单线程： 程序中只存在一个线程，实际上主方法就是一个主线程

4 多线程： 在一个程序中运行多个任务

目的是更好地使用CPU资源

## 线程的实现

### 1.继承Thread类

在java.lang包中定义, 继承Thread类必须重写run()方法

```
class MyThread extends Thread{
    private static int num = 0;

    public MyThread(){
        num++;
    }

    @Override
    public void run() {
        System.out.println("主动创建的第"+num+"个线程");
    }
}
```

创建好了自己的线程类之后，就可以创建线程对象了，然后通过start()方法去启动线程。注意，不是调用run()方法启动线程，run方法中只是定义需要执行的任务，如果调用run方法，即相当于在主线程中执行run方法，跟普通的方法调用没有任何区别，此时并不会创建一个新的线程来执行定义的任务。

```
public class Test {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

start()方法调用和run()方法调用的区别:

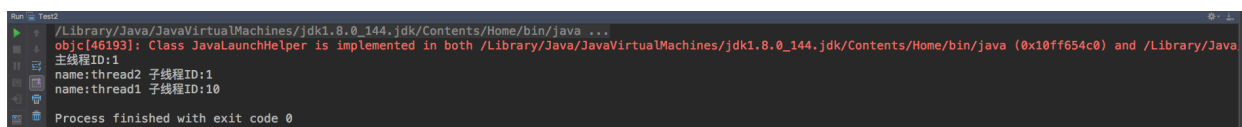
```
public class Test {
    public static void main(String[] args) {
        System.out.println("主线程ID:"+Thread.currentThread().getId());
        MyThread thread1 = new MyThread("thread1");
        thread1.start();
        MyThread thread2 = new MyThread("thread2");
        thread2.run();
    }
}

class MyThread extends Thread{
    private String name;

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

    @Override
    public void run() {
        System.out.println("name:"+name+" 子线程ID:"+Thread.currentThread().getId());
    }
}
```

运行结果:



```

/Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java ...
objc[46193]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java (0x10ff654c0) and /Library/Java
主线程ID:1
name:thread2 子线程ID:1
name:thread1 子线程ID:10
Process finished with exit code 0

```

从输出结果可以得出以下结论:

1) thread1和thread2的线程ID不同, thread2和主线程ID相同, 说明通过run方法调用并不会创建新的线程, 而是在主线程中直接运行run方法, 跟普通的方法调用没有任何区别;

2) 虽然thread1的start方法调用在thread2的run方法前面调用，但是先输出的是thread2的run方法调用的相关信息，说明新线程创建的过程不会阻塞主线程的后续执行。

## 2.实现Runnable接口

在Java中创建线程除了继承Thread类之外，还可以通过实现**Runnable**接口来实现类似的功能。实现Runnable接口必须重写其**run ()** 方法。

```
public class Test {
    public static void main(String[] args) {
        System.out.println("主线程ID: "+Thread.currentThread().getId());
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}

class MyRunnable implements Runnable{
    public MyRunnable() {
    }

    @Override
    public void run() {
        System.out.println("子线程ID: "+Thread.currentThread().getId());
    }
}
```

Runnable的中文意思是“任务”，顾名思义，通过实现Runnable接口，我们定义了一个子任务，然后将子任务交由Thread去执行。注意，这种方式必须将Runnable作为Thread类的参数，然后通过Thread的start方法来创建一个新线程来执行该子任务。如果调用Runnable的run方法的话，是不会创建新线程的，这跟普通的方法调用没有任何区别。事实上，查看Thread类的实现源代码会发现Thread类是实现了Runnable接口的。在Java中，这2种方式都可以用来创建线程去执行子任务，具体选择哪一种方式要看自己的需求。直接继承Thread类的话，可能比实现Runnable接口看起来更加简洁，但是由于Java只允许单继承，所以如果自定义类需要继承其他类，则只能选择实现Runnable接口。

## 3.使用ExecutorService、Callable、Future实现有返回结果的多线程

多线程后续会学到，这里暂时先知道一下有这种方法即可。ExecutorService、Callable、Future这个对象实际上都是属于Executor框架中的功能类。有返回值的任务必须实现Callable接口

(java.util.concurrent)，类似的，无返回值的任务必须实现Runnable接口 (java.lang)。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。下面提供了一个完整的有返回结果的多线程测试例子。代码如下：

```
/**
 * 有返回值的线程
```

```

*/
@SuppressWarnings("unchecked")
public class Test {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        System.out.println("----程序开始运行----");
        Date date1 = new Date();

        int taskSize = 5;
        // 创建一个线程池
        ExecutorService pool = Executors.newFixedThreadPool(taskSize);
        // 创建多个有返回值的任务
        List<Future> list = new ArrayList<Future>();
        for (int i = 0; i < taskSize; i++) {
            Callable c = new MyCallable(i + " ");
            // 执行任务并获取Future对象
            Future f = pool.submit(c);
            // System.out.println(">>>" + f.get().toString());
            list.add(f);
        }
        // 关闭线程池
        pool.shutdown();

        // 获取所有并发任务的运行结果
        for (Future f : list) {
            // 从Future对象上获取任务的返回值，并输出到控制台
            System.out.println(">>>" + f.get().toString());
        }

        Date date2 = new Date();
        System.out.println("----程序结束运行----， 程序运行时间【"
            + (date2.getTime() - date1.getTime()) + "毫秒】");
    }
}

class MyCallable implements Callable<Object> {
    private String taskNum;

    MyCallable(String taskNum) {
        this.taskNum = taskNum;
    }

    public Object call() throws Exception {
        System.out.println(">>>" + taskNum + "任务启动");
        Date dateTmp1 = new Date();
        Thread.sleep(1000);
        Date dateTmp2 = new Date();
        long time = dateTmp2.getTime() - dateTmp1.getTime();
        System.out.println(">>>" + taskNum + "任务终止");
    }
}

```

```
        return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";
    }
}
```

运行结果：

```
-----程序开始运行-----
>>>0 任务启动
>>>1 任务启动
>>>2 任务启动
>>>3 任务启动
>>>4 任务启动
>>>2 任务终止
>>>1 任务终止
>>>4 任务终止
>>>0 任务终止
>>>3 任务终止
>>>0 任务返回运行结果,当前任务时间【1002毫秒】
>>>1 任务返回运行结果,当前任务时间【1002毫秒】
>>>2 任务返回运行结果,当前任务时间【1002毫秒】
>>>3 任务返回运行结果,当前任务时间【1001毫秒】
>>>4 任务返回运行结果,当前任务时间【1001毫秒】
-----程序结束运行-----，程序运行时间【1008毫秒】
```

上述代码中的Executors类，提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

public static ExecutorService **newFixedThreadPool(int nThreads)** 创建固定数目线程的线程池。

public static ExecutorService **newCachedThreadPool()** 创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService **newSingleThreadExecutor()** 创建一个单线程化的Executor。

public static ScheduledExecutorService **newScheduledThreadPool(int corePoolSize)** 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

## 线程的状态

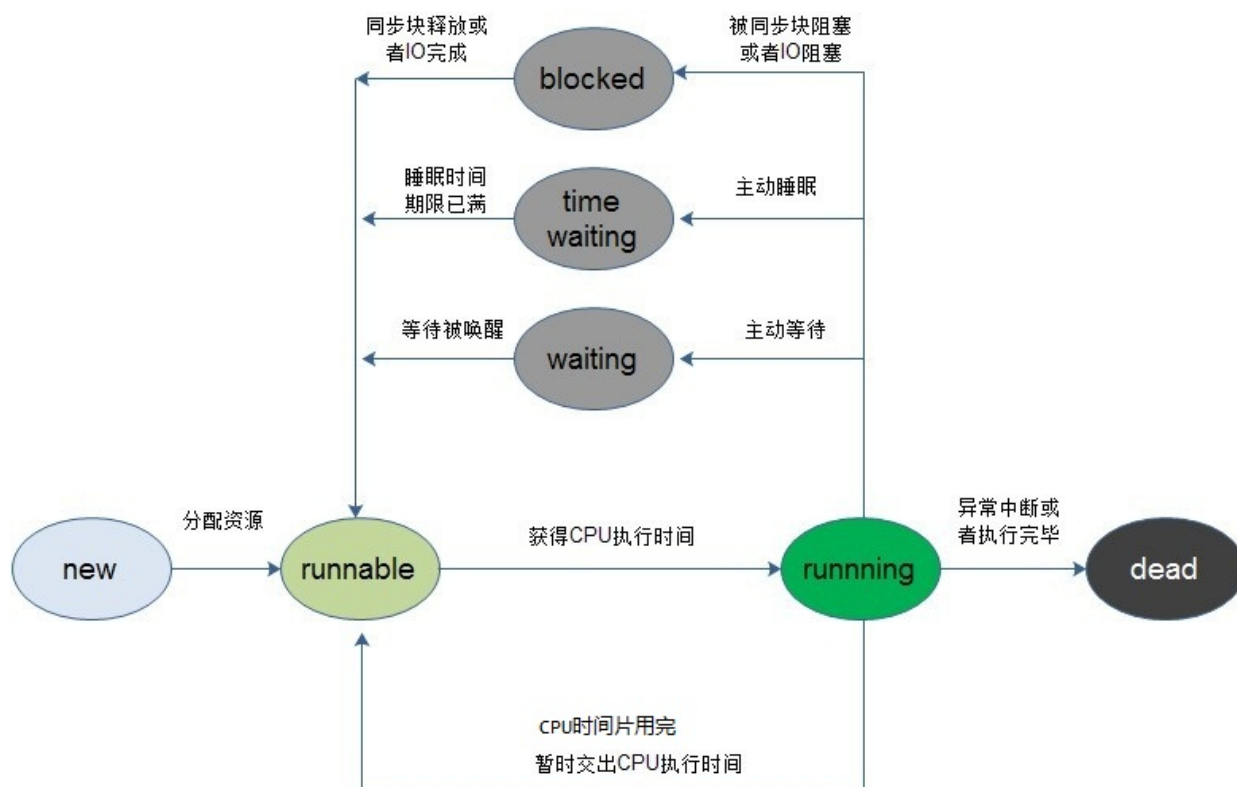
- 创建 (new) 状态: 准备好了一个多线程的对象
- 就绪 (runnable) 状态: 调用了 `start()` 方法, 等待CPU进行调度
- 运行 (running) 状态: 执行 `run()` 方法
- 阻塞 (blocked) 状态: 暂时停止执行, 可能将资源交给其它线程使用
- 终止 (dead) 状态: 线程销毁

当需要新起一个线程来执行某个子任务时, 就创建了一个线程。但是线程创建之后, 不会立即进入就绪状态, 因为线程的运行需要一些条件 (比如内存资源, 程序计数器、Java栈、本地方法栈都是线程私有的, 所以需要为线程分配一定的内存空间), 只有线程运行需要的所有条件满足了, 才进入就绪状态。

当线程进入就绪状态后, 不代表立刻就能获取CPU执行时间, 也许此时CPU正在执行其他的事情, 因此它要等待。当得到CPU执行时间之后, 线程便真正进入运行状态。

线程在运行状态过程中, 可能有多个原因导致当前线程不继续运行下去, 比如用户主动让线程睡眠 (睡眠一定的时间之后再重新执行)、用户主动让线程等待, 或者被同步块给阻塞, 此时就对应着多个状态: time waiting (睡眠或等待一定的事件)、waiting (等待被唤醒)、blocked (阻塞)。

当由于突然中断或者子任务执行完毕, 线程就会被消亡。



sleep和wait的区别:

- sleep是Thread类的方法,wait是Object类中定义的方法.
- Thread.sleep不会导致锁行为的改变, 如果当前线程是拥有锁的, 那么Thread.sleep不会让线程释放锁.
- Thread.sleep和Object.wait都会暂停当前的线程. OS会将执行时间分配给其它线程. 区别是, 调用wait后, 需要别的线程执行notify/notifyAll才能够重新获得CPU执行时间.

## 上下文切换

对于单核CPU来说，CPU在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

由于可能当前线程的任务并没有执行完毕，所以在切换时需要保存线程的运行状态，以便下次重新切换回来时能够继续切换之前的状态运行。举个简单的例子：比如一个线程A正在读取一个文件的内容，正读到文件的一半，此时需要暂停线程A，转去执行线程B，当再次切换回来执行线程A的时候，我们不希望线程A又从文件的开头来读取。

因此需要记录线程A的运行状态，那么会记录哪些数据呢？因为下次恢复时需要知道在这之前当前线程已经执行到哪条指令了，所以需要记录程序计数器的值，另外比如说线程正在进行某个计算的时候被挂起了，那么下次继续执行的时候需要知道之前挂起时变量的值时多少，因此需要记录CPU寄存器的状态。所以一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

对于线程的上下文切换实际上就是 **存储和恢复CPU状态的过程**，它使得线程执行能够从中断点恢复执行。

虽然多线程可以使得任务执行的效率得到提升，但是由于在线程切换时同样会带来一定的开销代价，并且多个线程会导致系统资源占用的增加，所以在进行多线程编程时要注意这些因素。

## 线程的常用方法

---

类别	方法签名	简介
线程的创建	Thread ()	
	Thread (String name)	创建传入名称的线程
	Thread(Runnable target)	
	Thread(Runnable target, String name)	
线程的方法	public void start()	使该线程开始执行；Java 虚拟机调用该线程的 run 方法。
	public void run()	如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。
	setName(String name)	改变线程名称，使之与参数 name 相同。
	setPriority(int priority)	更改线程的优先级。
	setDaemon(boolean on)	将该线程标记为守护线程或用户线程。
	join(long millisec)	等待该线程终止的时间最长为 millis 毫秒。
	interrupt()	中断线程。
	isAlive()	测试线程是否处于活动状态。
	yield()	暂停当前正在执行的线程对象，并执行其他线程。
	sleep(long millisec)	在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
获取线程的引用	currentThread()	返回对当前正在执行的线程对象的引用。

## currentThread()方法



currentThread()方法可以返回代码段正在被哪个线程调用的信息。

## sleep()方法

sleep相当于让线程睡眠，交出CPU，让CPU去执行其他的任务。但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。

如果方法不加锁

```
public class Test {

    private int i = 10;
    private Object object = new Object();

    public static void main(String[] args) throws IOException {
        Test test = new Test();
        MyThread thread1 = test.new MyThread();
        MyThread thread2 = test.new MyThread();
        thread1.start();
        thread2.start();
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            i++;
            System.out.println("i:"+i);
            try {
                System.out.println("线程"+Thread.currentThread().getName()+"进入睡眠状态");
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                // TODO: handle exception
            }
            System.out.println("线程"+Thread.currentThread().getName()+"睡眠结束");
            i++;
            System.out.println("i:"+i);
        }
    }
}
```

运行结果

```
i:11  
线程Thread-0进入睡眠状态  
i:12  
线程Thread-1进入睡眠状态  
线程Thread-0睡眠结束  
线程Thread-1睡眠结束  
i:14  
i:13
```

方法加锁

```

public class Test {

    private int i = 10;
    private Object object = new Object();

    public static void main(String[] args) throws IOException {
        Test test = new Test();
        MyThread thread1 = test.new MyThread();
        MyThread thread2 = test.new MyThread();
        thread1.start();
        thread2.start();
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            synchronized (object) {
                i++;
                System.out.println("i:"+i);
                try {
                    System.out.println("线程"+Thread.currentThread().getName()+"进入睡眠状态");
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    // TODO: handle exception
                }
                System.out.println("线程"+Thread.currentThread().getName()+"睡眠结束");
                i++;
                System.out.println("i:"+i);
            }
        }
    }
}

```

运行结果

```
i:11
线程Thread-0进入睡眠状态
线程Thread-0睡眠结束
i:12
i:13
线程Thread-1进入睡眠状态
线程Thread-1睡眠结束
i:14
```

从上面输出结果可以看出，当Thread-0进入睡眠状态之后，Thread-1并没有去执行具体的任务。只有当Thread-0执行完之后，此时Thread-0释放了对象锁，Thread-1才开始执行。

注意，如果调用了sleep方法，必须捕获InterruptedException异常或者将该异常向上层抛出。当线程睡眠时间满后，不一定会立即得到执行，因为此时可能CPU正在执行其他的任务。所以说调用sleep方法相当于让线程进入阻塞状态。

## yield()方法

调用yield方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。

注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。

```
public class MyThread extends Thread{
    @Override
    public void run() {
        long beginTime=System.currentTimeMillis();
        int count=0;
        for (int i=0;i<50000000;i++){
            count=count+(i+1);
            //Thread.yield();
        }
        long endTime=System.currentTimeMillis();
        System.out.println("用时: "+(endTime-beginTime)+" 毫秒! ");
    }
}

public class Run {
    public static void main(String[] args) {
        MyThread t= new MyThread();
        t.start();
    }
}
```

执行结果

用时：3毫秒

如果将 `//Thread.yield();`的注释去掉，执行结果如下：

用时：16080 毫秒！

## 对象方法：

### isAlive()方法

方法isAlive()的作用是测试线程是否处于活动状态。活动状态就是线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是“存活”的。

```
public class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("run="+this.isAlive());
    }
}

public class RunTest {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread=new MyThread();
        System.out.println("begin =="+myThread.isAlive());
        myThread.start();
        System.out.println("end =="+myThread.isAlive());
    }
}
```

执行结果：

```
begin ==false
run=true
end ==false
```

方法isAlive()的作用是测试线程是否处于活动状态。什么是活动状态呢？活动状态就是线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是“存活”的。

### join()方法

在很多情况下，主线程创建并启动了线程，如果子线程中进行大量耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到join()方法了。方法join()的作用是等待线程对象销毁。

```

public class Thread4 extends Thread{
    public Thread4(String name) {
        super(name);
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(getName() + " " + i);
        }
    }
    public static void main(String[] args) throws InterruptedException {
        // 启动子进程
        new Thread4("new thread").start();
        for (int i = 0; i < 10; i++) {
            if (i == 5) {
                Thread4 th = new Thread4("joined thread");
                th.start();
                th.join();
            }
            System.out.println(Thread.currentThread().getName() + " " +
i);
        }
    }
}

```

执行结果：

```
main 0
main 1
main 2
main 3
main 4
new thread 0
new thread 1
new thread 2
new thread 3
new thread 4
joined thread 0
joined thread 1
joined thread 2
joined thread 3
joined thread 4
main 5
main 6
main 7
main 8
main 9
```

由上可以看出main主线程等待joined thread线程先执行完了才结束的。如果把th.join()这行注释掉，运行结果如下：

```
main 0
main 1
main 2
main 3
main 4
main 5
main 6
main 7
main 8
main 9
new thread 0
new thread 1
new thread 2
new thread 3
new thread 4
joined thread 0
joined thread 1
joined thread 2
joined thread 3
joined thread 4
```

## getName和setName

用来得到或者设置线程名称。

## getPriority和setPriority

用来获取和设置线程优先级。

## setDaemon和isDaemon

用来设置线程是否成为守护线程和判断线程是否是守护线程。

在Java线程中有两种线程，一种是User Thread（用户线程），另一种是Daemon Thread(守护线程)。Daemon的作用是为其他线程的运行提供服务，比如说GC线程。其实User Thread线程和Daemon Thread守护线程本质上来说去没啥区别的，唯一的区别之处就在虚拟机的离开：如果User Thread全部撤离，那么Daemon Thread也就没啥线程好服务的了，所以虚拟机也就退出了。

守护线程并非虚拟机内部可以提供，用户也可以自行的设定守护线程，方法：`public final void setDaemon(boolean on)`；但是有几点需要注意：

- `thread.setDaemon(true)`必须在`thread.start()`之前设置，否则会跑出一个`IllegalThreadStateException`异常。你不能把正在运行的常规线程设置为守护线程。
- 在Daemon线程中产生的新线程也是Daemon的。
- 不是所有的应用都可以分配给Daemon线程来进行服务，比如读写操作或者计算逻辑。因为在Daemon Thread还没来的及进行操作时，虚拟机可能已经退出了。

守护线程和用户线程的区别在于：守护线程依赖于创建它的线程，而用户线程则不依赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。



```

public class Test6 {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.setDaemon(true);
        thread.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("我离开thread对象也不再打印了，也就是停止了");
    }
}

class MyThread extends Thread {
    private int i = 1;

    @Override
    public void run() {
        while (true) {
            i++;
            System.out.println("i=" + (i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

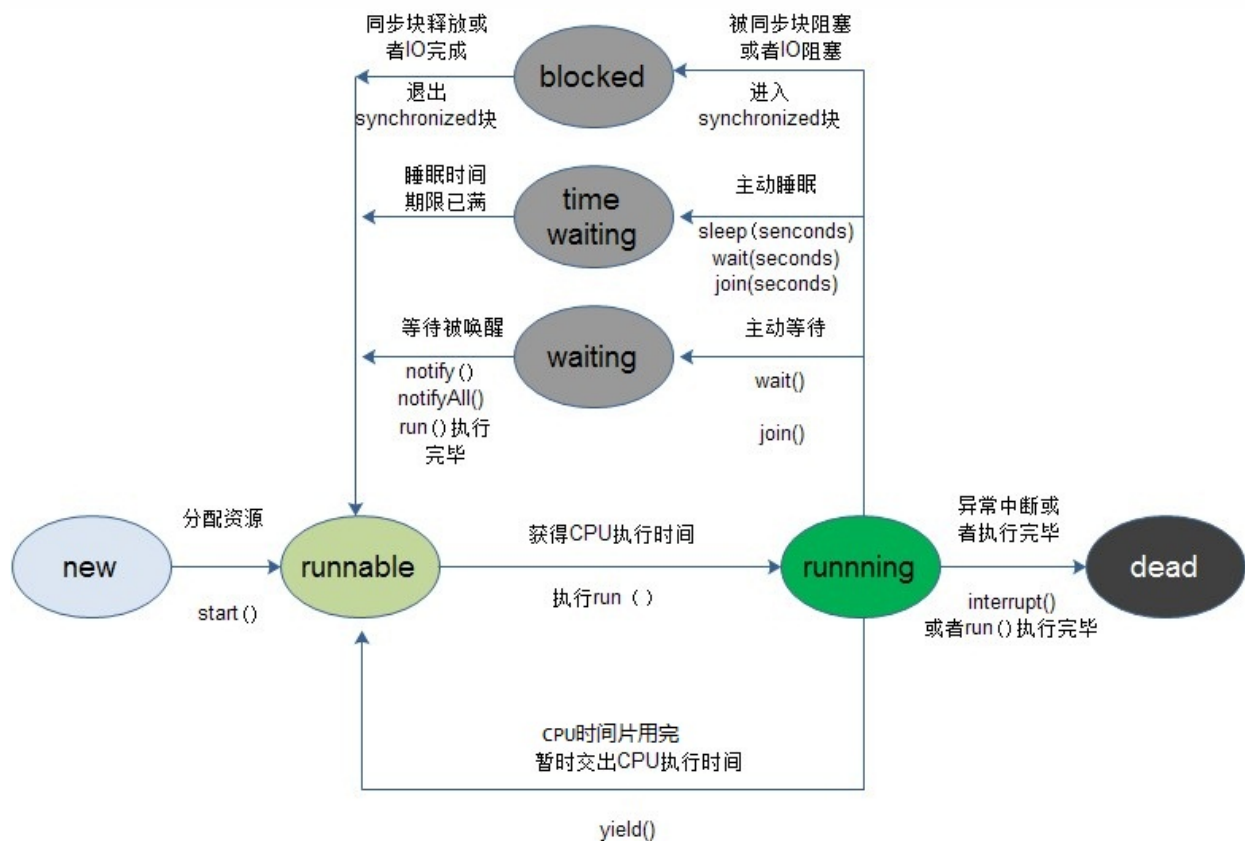
执行结果：

```

i=2
i=3
i=4
i=5
i=6
我离开thread对象也不再打印了，也就是停止了

```

**Thread类中的方法调用到底会引起线程状态发生的变化图**



## 停止线程

停止一个线程可以使用 `Thread.stop()` 方法，但最好不用它。该方法是不安全的，已被弃用。在Java中有以下3种方法可以终止正在运行的线程：

- 使用退出标志，使线程正常退出，也就是当 `run` 方法完成后线程终止
- 使用 `stop` 方法强行终止线程，但是不推荐使用这个方法，因为 `stop` 和 `suspend` 及 `resume` 一样，都是作废过期的方法，使用他们可能产生不可预料的结果。
- 使用 `interrupt` 方法中断线程，但这个不会终止一个正在运行的线程，还需要加入一个判断才可以完成线程的停止。