

线程的状态和API

一、多线程的代码运行顺序

当代码中使用了多线程，那么代码的顺序，不是程序执行的顺序。

```
package thread;

import com.psfed.thread.ThreadA;

public class Demo {

    public static void main(String[] args) {
        System.out.println("主线程开始");

        ThreadA t1 = new ThreadA("A");
        t1.setName("吴麻子的线程");
        t1.start();

        ThreadA t2 = new ThreadA("B");
        t2.start();
        System.out.println("主线程结束");

    }
}
```

为什么说代码的顺序，不再是程序执行的顺序？

因为启动了多线程以后，子线程和主线程，是同时运行的。所以，在结果来看，
System.out.println("主线程结束");这行代码可能会比子线程中的run方法先执行完。

二、继承Thread类和实现Runnable接口区别

```
//实现Runnable
public class ThreadB implements Runnable {

    private int count = 100;

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            count--;
            System.out.println(Thread.currentThread().getName() + "; count = " + count);
        }
    }
}
```

```

}

//继承Thread
package com.psfd.thread;

public class ThreadA extends Thread{

    private int count = 100;

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            count--;
            System.out.println(Thread.currentThread().getName() + "; count = " + count);
        }
    }
}

//调用
package thread;

import com.psfd.thread.ThreadA;
import com.psfd.thread.ThreadB;

public class Demo {

    public static void main(String[] args) {
        //继承Thread类
        ThreadA t1 = new ThreadA();
        t1.setName("A");
        t1.start();
        ThreadA t2 = new ThreadA();
        t2.setName("B");
        t2.start();

        //实现Runnable接口
        ThreadB obj = new ThreadB();
        Thread t3 = new Thread(obj);
        t3.setName("C");
        t3.start();
        Thread t4 = new Thread(obj);
        t4.setName("D");
        t4.start();
    }
}

```

总结：

- 1、推荐使用实现Runnable接口
- 2、实现Runnable接口，有助于资源的共享。
- 3、java的继承（extends）是单一继承，而implements可以实现多重继承。

当线程类还有其他父类的时候，extends非常困难。

三、基本API

1、run () 和start ()

```
//测试调用start方法和run方法的区别
package thread;

import com.psfd.thread.ThreadA;

public class Demo {

    public static void main(String[] args) {
        System.out.println("主线程开始");
        ThreadA t1 = new ThreadA("A");
        //A线程调用start
        t1.start();

        ThreadA t2 = new ThreadA("B");
        //B线程调用run
        t2.run();

        System.out.println("当前主线程：" + Thread.currentThread().getId());
        System.out.println("主线程结束");
    }
}

线程类：
package com.psfd.thread;

public class ThreadA extends Thread {

    private String name;

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

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("当前子线程：id= " + Thread.currentThread().getId() + ";name
= " + name);
        }
    }
}
```

```
}  
}
```

执行结果：

主线程开始

当前子线程：id= 10;name = A

当前子线程：id= 1;name = B

当前子线程：id= 10;name = A

当前子线程：id= 1;name = B

当前子线程：id= 10;name = A

当前子线程：id= 1;name = B

当前子线程：id= 1;name = B

当前子线程：id= 1;name = B

当前子线程：id= 10;name = A

当前主线程：1

当前子线程：id= 10;name = A

主线程结束

从结果分析：调用start方法的A线程，启动了一个新的线程（id和主线程的ID不一致）

而调用run方法，并没有启动新的线程，而是由main线程直接调用run方法。（id和主线程的ID一致的）

结论：

1、不管是调用start方法还是调用run方法，最终都会执行线程run方法

2、调用start方法，会启动一个新的线程

而调用run方法，是主线程的单线程运行，并没有启动新的线程。

2、Thread中基本方法

获取线程的基本信息

```
/**  
static Thread currentThread() 获取当前执行该段代码的线程  
long getId() 获取线程的ID  
String getName() 获取线程的名字  
int getPriority() 设置线程的优先级  
Thread.State getState() Thread.State是Thread的内部类  
boolean isAlive()  
boolean isDaemon() 守护线程（后台线程、精灵线程）  
setDaemon(boolean on) 设置一个线程为守护线程  
setName(String name)  
setPriority(int newPriority)  
static void sleep(long millis) 线程休眠（毫秒）  
**/  
  
System.out.println(String.format("主线程name=%s,id=%d",  
Thread.currentThread().getName(),Thread.currentThread().getId()));  
  
ThreadA t1 = new ThreadA("A");
```

```
t1.setName("吴麻子的线程");    //修改线程的名字
t1.start();
```

四、线程的状态（线程的生命周期）

1、看图

图1：

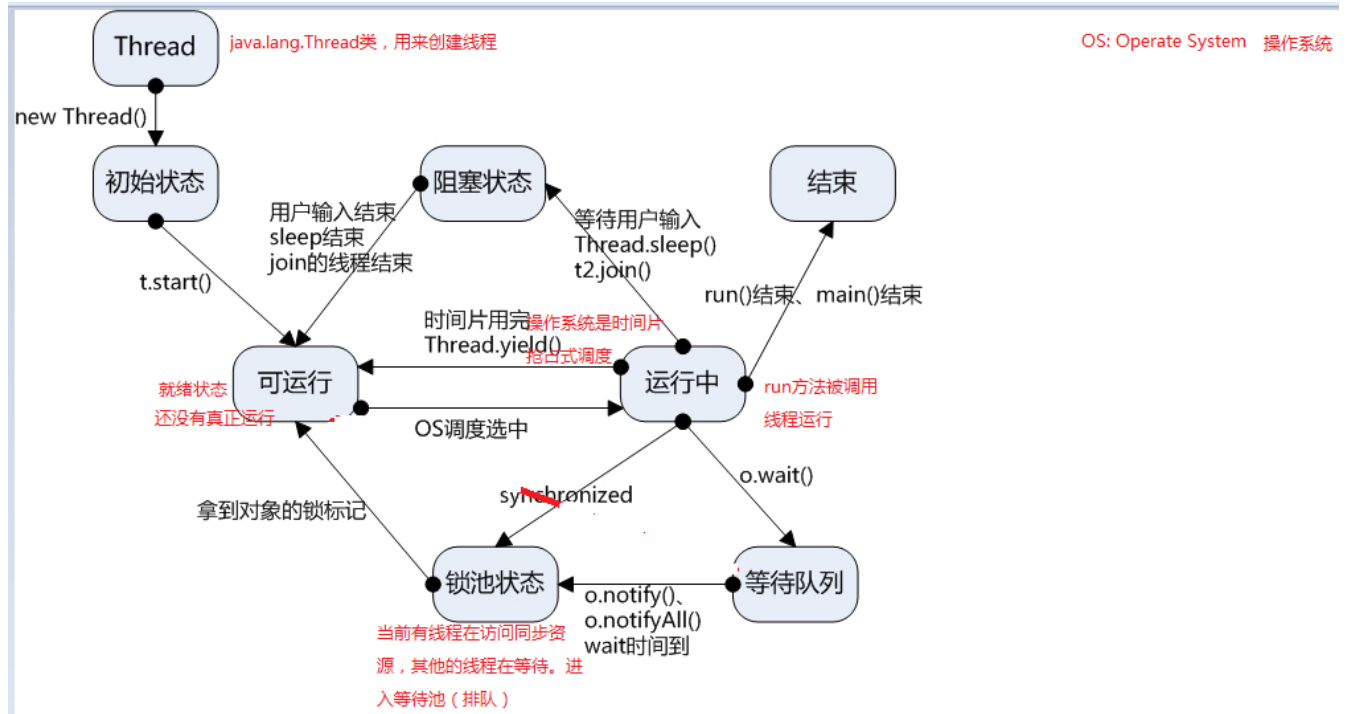
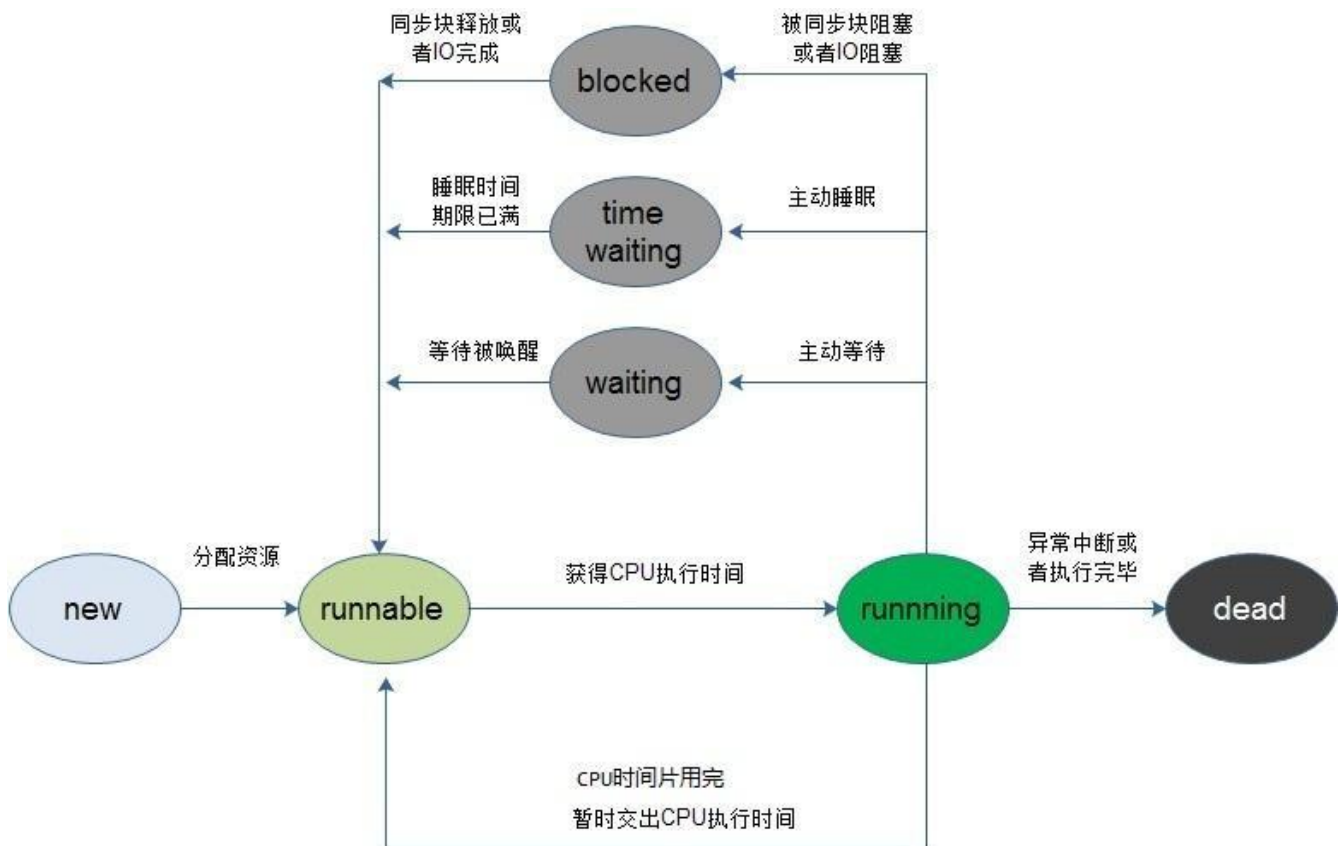


图2：



- 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资源, 操作系统进行调度
- 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

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。(wait会释放持有的锁)
- (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。(注意,sleep是不会释放持有的锁)

- 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, 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。

2、sleep和wait

sleep () :

java.lang.Thread中定义的方法

作用是在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行）。这个“正在执行的线程”是指this.currentThread()返回的线程。

wait () :

java.lang.Object中定义的方法

作用是让线程等待指定的毫秒数。

```
package thread;

import java.io.IOException;

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();
        thread1.setName("王麻子");
        MyThread thread2 = test.new MyThread();
        thread2.setName("吴麻子");
        thread1.start();
        thread2.start();
    }

    class MyThread extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                i++;
                System.out.println(Thread.currentThread().getName() + " i = " + i);
                try {
                    System.out.println("线程" + Thread.currentThread().getName() + "进入睡眠状态");

                    //sleep
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
```

```

        }
        System.out.println("线程" + Thread.currentThread().getName() + "睡眠结束");
        i++;
        System.out.println(Thread.currentThread().getName() + "i = " + i);
    }
}
}
}
}

```

//执行结果：

```

王麻子 i = 11
线程王麻子进入睡眠状态
线程王麻子睡眠结束
王麻子i = 12
吴麻子 i = 13
线程吴麻子进入睡眠状态
线程吴麻子睡眠结束
吴麻子i = 14

```

从结果得知：当“王麻子”睡眠以后，“吴麻子”是进不来的。因为sleep不会释放锁。

//将sleep改成wait

```

package thread;

import java.io.IOException;

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();
        thread1.setName("王麻子");
        MyThread thread2 = test.new MyThread();
        thread2.setName("吴麻子");
        thread1.start();
        thread2.start();
    }

    class MyThread extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                i++;
                System.out.println(Thread.currentThread().getName() + " i = " + i);
                try {
                    System.out.println("线程" + Thread.currentThread().getName() + "进入睡眠
状态");

                    //wait
                    Thread.currentThread().wait(3000);
                } catch (InterruptedException e) {

```



```

        }
        System.out.println("线程" + Thread.currentThread().getName() + "睡眠结束");
        i++;
        System.out.println(Thread.currentThread().getName() + "i = " + i);
    }
}
}
}
}

```

//运行结果

王麻子 i = 11

线程王麻子进入睡眠状态

吴麻子 i = 12

线程吴麻子进入睡眠状态

Exception in thread "王麻子" java.lang.IllegalMonitorStateException

at java.lang.Object.wait(Native Method)

at thread.Test\$MyThread.run(Test.java:27)

Exception in thread "吴麻子" java.lang.IllegalMonitorStateException

at java.lang.Object.wait(Native Method)

at thread.Test\$MyThread.run(Test.java:27)

不用理会异常。

王麻子wait以后，吴麻子乘虚而入。

说：wait方法会释放锁资源。

sleep和wait都是让线程暂来。但是他们是有区别的。

1、sleep()睡眠时，保持对象锁，仍然占有该锁；其他线程进不来同步的区域 2、而wait()睡眠时，释放对象锁。其他线程可以访问同步区域

作业：

龟兔赛跑问题

龟兔赛跑：2000米 要求：(1)兔子每 0.1 秒 5 米的速度，每跑20米休息1秒；(2)乌龟每 0.1 秒跑 2 米，不休息；(3)其中一个跑到终点后另一个不跑了！ 程序设计思路：(1)创建一个Animal动物类，继承Thread，编写一个running抽象方法，重写run方法，把running方法在run方法里面调用。(2)创建Rabbit兔子类和Tortoise乌龟类，继承动物类(3)两个子类重写running方法

3、notify和notifyAll

sleep方法，让线程休眠指定的时间。时间到了以后，自动进入就绪状态，等待CPU资源，然后运行。

wait方法：

wait()：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() 方法

wait(long timeout) 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() 方法，或者指定的时间过完。

notify：唤醒在此对象监视器上等待的单个线程

notifyall：唤醒在此对象监视器上等待的所有线程

```
package thread;

import java.io.IOException;

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();
        thread1.setName("王麻子");
        MyThread thread2 = test.new MyThread();
        thread2.setName("吴麻子");

        MyThread2 thread3 = test.new MyThread2();
        thread3.setName("张麻子");

        thread1.start();
        thread2.start();

        //休眠一秒，原因是：让前面两个线程先执行，让两个线程都进入等待状态。
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //唤醒线程
        thread3.start();
    }

    class MyThread extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                i++;
                System.out.println(Thread.currentThread().getName() + " i = " + i);
                try {
                    System.out.println("线程" + Thread.currentThread().getName() + "进入睡眠状态");
                    object.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```

        } catch (InterruptedException e) {

        }

        System.out.println("线程" + Thread.currentThread().getName() + "睡眠结束");
        i++;
        System.out.println(Thread.currentThread().getName() + "i = " + i);
    }
}

class MyThread2 extends Thread {
    @Override
    public void run() {
        synchronized (object) {
            //唤醒该对象 ( object ) 上等待的所有线程
            object.notifyAll();
        }
    }
}
}

```

//1、wait、notify、notifyAll都是Object的方法。任何对象都有这三个方法。

//2、当需要调用以上的方法的时候，一定要对竞争资源进行加锁，如果不加锁的话，则会报IllegalMonitorStateException 异常

//3、假设有三个线程执行了obj.wait()，那么obj.notifyAll()则能全部唤醒tread1，thread2，thread3，但是要继续执行obj.wait()的下一条语句，必须获得obj锁，因此，tread1，thread2，thread3只有一个有机会获得锁继续执行，例如tread1，其余的需要等待thread1释放obj锁之后才能继续执行

4、join方法

在一个线程中调用另一个线程的join方法。

其作用是：当前线程等待调用join方法的线程结束后，才往下执行。

使用场景：当前线程要使用指定线程（调用join方法的线程）的计算结果。必须要等到指定线程计算完成后才能往下走。

```

package thread;

import com.psfd.thread.ThreadA;
import com.psfd.thread.ThreadB;

public class Demo {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("主线程开始");
        //继承Thread类
        ThreadA t1 = new ThreadA();
        t1.setName("A");
        t1.start();
    }
}

```

```

        ThreadA t2 = new ThreadA();
        t2.setName("B");
        t2.start();

        t1.join();
        t2.join();

        System.out.println("主线程结束");
    }
}

```

5、yield

static void yield()

方法意思是：退让、让步。

yield方法也是让线程暂停，让出CPU资源，其他线程可以执行。

sleep：休眠。不会释放锁，更加不会释放CPU。sleep时间结束后，恢复到就绪状态，等待CPU资源再执行

wait：等待，会释放锁资源。要么等时间结束，要么等其他线程唤醒，回复到就绪状态，等待CPU资源再执行

yield：暂停线程，让出CPU资源和锁资源。当前线程立即回复到就绪状态，等待CPU资源再执行。

相当于是点了重置按钮。或者就是：磁盘的格式化。

调用yield方法，有时候是完全看不出效果来。原因是：调用了yield方法，立即回到就绪状态，如果恰好此时抢到了CPU资源（操作系统马上调度了它），立即就恢复执行。

6、stop

停止线程。可能会造成线程的死锁。

不建议使用。

作业：

1、启动3个线程打印递增的数字，线程1先打印1,2,3,4,5, 然后是线程2打印6,7,8,9,10, 然后是线程3打印11,12,13,14,15. 接着再由线程1打印16,17,18,19,20....以此类推, 直到打印到75. 程序的输出结果应该为: 线程1: 1 线程1: 2 线程1: 3 线程1: 4 线程1: 5

线程2: 6 线程2: 7 线程2: 8 线程2: 9 线程2: 10 ...

线程3: 11 线程3: 12 线程3: 13 线程3: 14 线程3: 15

2、创建两个线程，其中一个输出1-52，另外一个输出A-Z。

输出格式要求：12A 34B 56C 78D ...

对应关系：大写的英文字母，有26个。

按顺序：两个数字，对应一个字母。