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.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");
        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：

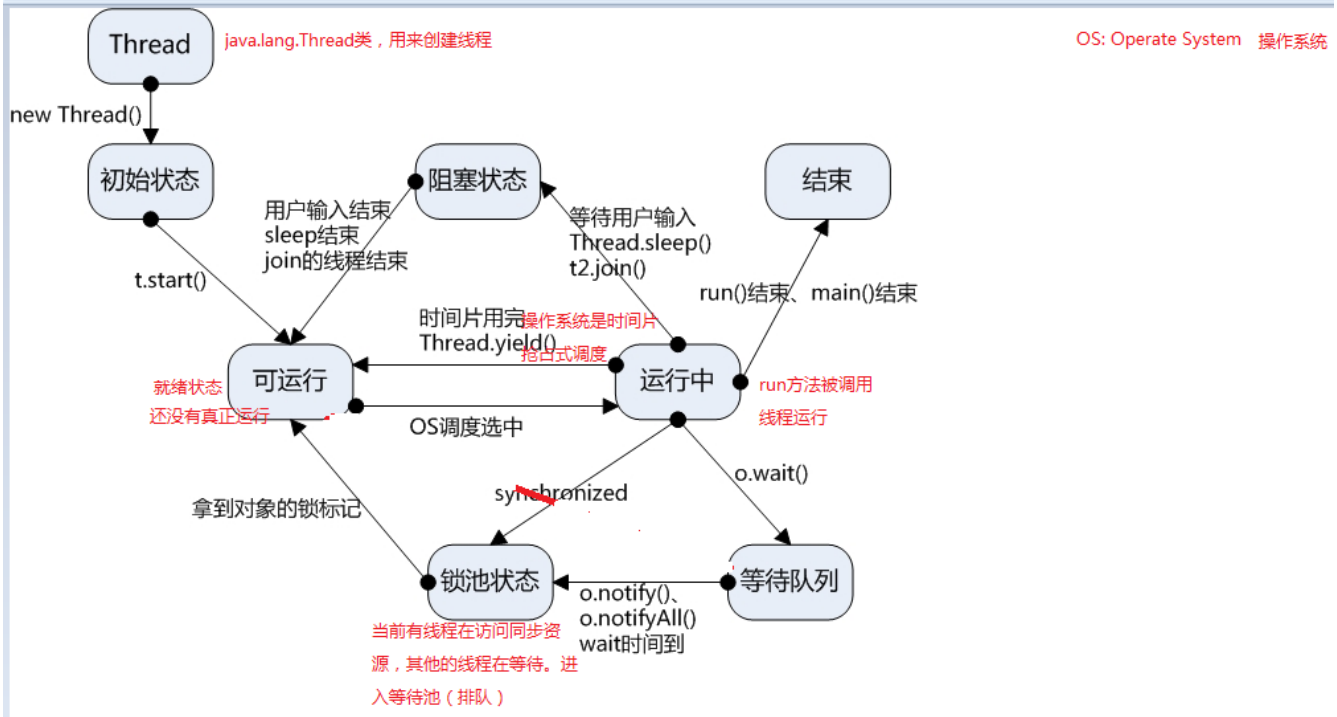
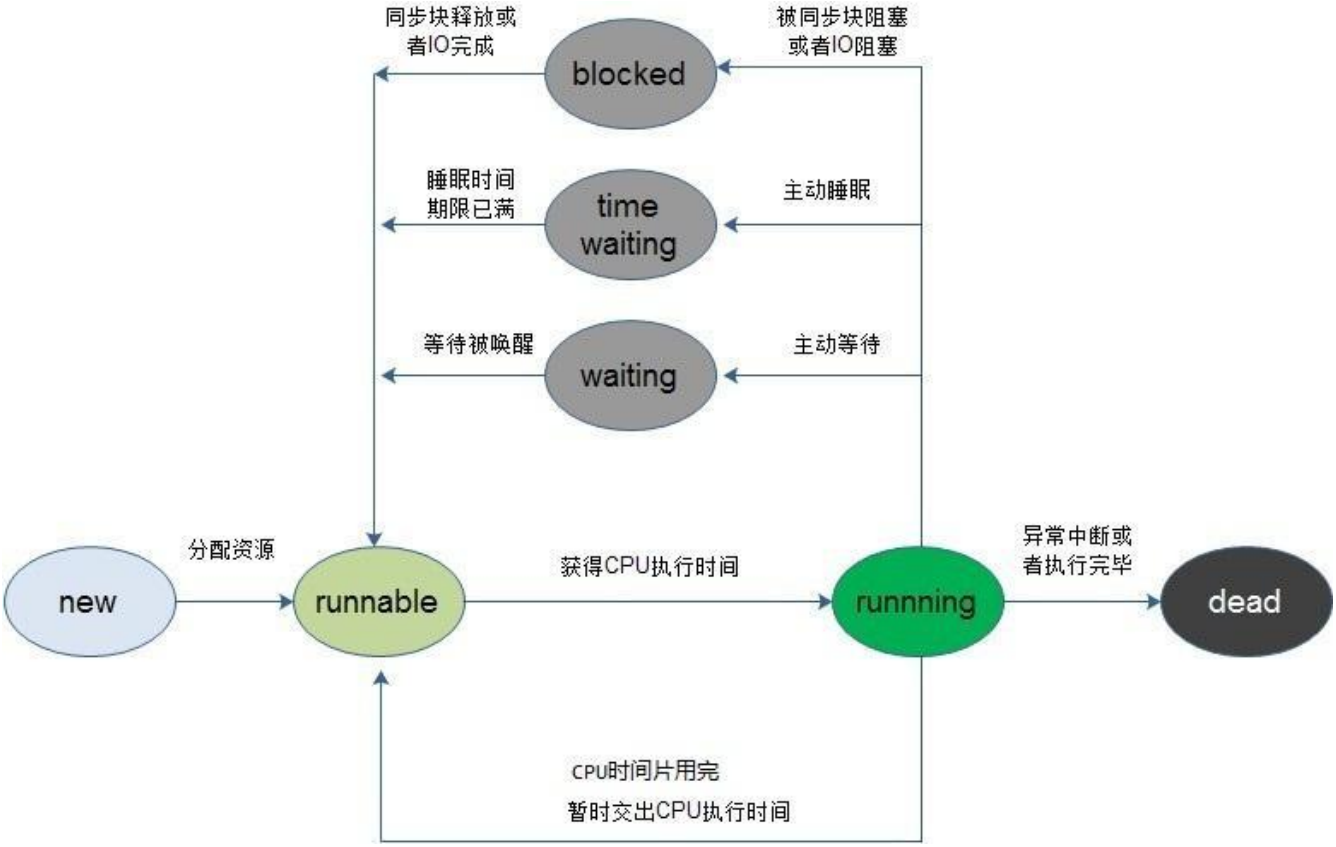


图2：



- 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、notity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) {

                }
                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。

相当与是点了重置按钮。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: 71 线程3: 72 线程3: 73 线程3: 74 线程3: 75

```
public class work3 {
```

```
    public static void main(String[] args) {
        Object obj = new Object();
        for(int i=1;i<=3;i++){//传入对象锁 和 i值

            new Thread(new MyThread2(obj,i), "线程"+i).start();
        }
    }
}
class MyThread2 implements Runnable{
    private static int i = 0;//打印的值 1-75
    private static int count=0;//计数 三次一轮回
    private Object obj;
    private int n;//接参 i值
    public MyThread2(Object obj,int n) {
        this.obj=obj;
        this.n = n;
    }


    @Override
    public void run() {
        synchronized (obj) {
            while(i<75){//i++ 在代码块 所以到74就可以了

                obj.notifyAll();//唤醒所有线程

                if(count%3!=(n-1)){ //找出下一个线程  不正确的线程等待

                    try {
                        obj.wait();
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
                i++;
                System.out.println(Thread.currentThread().getName()+": "+i);
                if(i%5 == 0){ //打印了五次后 此线程让出资源，等待
                    try {
                        count++; //count是static修饰 ，为了共享
                        System.out.println();
                        obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

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。

五、线程的优先级

一、线程优先级的介绍

java 中的线程优先级的范围是1～10，默认的优先级是5。

“高优先级线程”会优先于“低优先级线程”执行。

优先级高的线程，大概率会先获取到CPU资源。操作系统在调度线程的时候，会优先执行优先级高的线程。

每个线程都有一个优先级。“高优先级线程”会优先于“低优先级线程”执行。在一些运行的主线程中创建新的子线程时，子线程的优先级被设置为等于“创建它的主线程的优先级”，

这就是线程优先级的“继承性”。

子线程的优先级继承了创建它的父线程的优先级。除非：单独给子线程设置优先级。

二、获取线程优先级、设置线程的优先级

int getPriority（ ）获取线程的优先级 void setPriority(int priority) 设置线程的优先级，线程的优先级1-10. Thread 类中定义了三个常量。

-

`static int`	`MAX_PRIORITY` **线程可以拥有的最大优先级。**
`static int`	`MIN_PRIORITY` 线程可以拥有的最小优先级。
`static int`	`NORM_PRIORITY` 分配给线程的默认优先级。

```
package com.psfd.thread;

public class ThreadA extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++)
            //获取优先级
            System.out.println(String.format("%s的线程，优先级为%d，第%d此循环",
Thread.currentThread().getName(),
```

```
                    Thread.currentThread().getPriority(), (i + 1)));
        }
    }
}
```

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

        //设置优先级
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);

        t1.start();
        t2.start();
        System.out.println("主线程结束");
    }
}
```

结果：
主线程开始
主线程结束
A的线程，优先级为1，第1此循环
B的线程，优先级为10，第1此循环
B的线程，优先级为10，第2此循环
A的线程，优先级为1，第2此循环
B的线程，优先级为10，第3此循环
A的线程，优先级为1，第3此循环
B的线程，优先级为10，第4此循环
A的线程，优先级为1，第4此循环
B的线程，优先级为10，第5此循环
A的线程，优先级为1，第5此循环

从结果上来看，并没有体现出“优先级”的作用。
其原因是：现在的CPU都是多核心的CPU。极有可能两个线程，由不同的CPU在执行。
但是，在单核CPU的情况下，体现的非常明显。

三、守护线程

1、概念

是一种特殊的线程，在背后提供服务。

在java中，最著名的守护线程就是：GC线程（垃圾回收线程）

java 中有两种线程：**用户线程**和**守护线程**。可以通过isDaemon()方法来区别它们：如果返回false，则说明该线程是“用户线程”；否则就是“守护线程”。用户线程一般用于执行用户级任务，而守护线程也就是“后台线程”，一般用来执行后台任务。需要注意的是：Java虚拟机在“用户线程”都结束后会后退出。

2、使用

boolean isDaemon() 获取一个线程是否是守护线程

void setDaemon(boolean) 设置一个线程为守护线程

```
package com.psfd.thread;

public class ThreadA extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(String.format("%s的线程，优先级为%d，第%d此循环",
Thread.currentThread().getName(),
                    Thread.currentThread().getPriority(), (i + 1)));
        }
        System.out.println(Thread.currentThread().getName() + "任务结束");
    }
}



package com.psfd.thread;

public class ThreadB implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            System.out.println(Thread.currentThread().getName() + i);
        }

        System.out.println(Thread.currentThread().getName() + "任务结束");
    }

}
```

```java
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("用户线程A");
        ThreadA t2 = new ThreadA();
        t2.setName("用户线程B");

        ThreadB t3Obj = new ThreadB();
        Thread t3 = new Thread(t3Obj);
        t3.setName("守护线程");
        //设置t3线程为守护线程。
        t3.setDaemon(true);


        t1.start();
        t2.start();
        t3.start();
        System.out.println("主线程结束");
    }
}
```

结果分析：
如果t3设置为守护线程，那么在t1，t2,和主线程执行结束后，JVM会自动的终止守护线程t3的执行。
如果t3也是用户线程，而不是守护线程，那么，JVM会等t3在执行完毕。

结论：

在程序中，当所有的用户线程执行结束后（包含了主线程），守护线程自动结束。

作业：

练题1:

编写程序实现,子线程循环3次,接着主线程循环5次,接着再子线程循环3次,主线程循环5次,如此反复3次.

练习题2:

设计四个线程,其中两个线程每次对变量i加1,另外两个线程每次对i减1.

练习题3:

自己编写代码,实现生产者-消费者模型功能.内容自由发挥,只需要表达思想.

用一个集合来装产品，生产者负责生产产品，消费负责消费产品。

模拟汽车的生产和消费。

练习题4：

现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？

练习题5：

有一个抽奖池,该抽奖池中存放了奖励的金额,该抽奖池用一个数组int[] arr = {10,5,20,50,100,200,500,800,2,80,300};创建两个抽奖箱(线程)设置线程名称分别为“抽奖箱1”，“抽奖箱2”，随机从arr数组中获取奖项元素并打印在控制台上,格式如下:

抽奖箱1 抽出奖金5元

抽奖箱2 抽出奖金10元

....

...

练习题6：

某公司组织年会,会议入场时有两个入口,在入场时每位员工都能获取一张双色球彩票,假设公司有100个员工,利用多线程模拟年会入场过程, 并分别统计每个入口入场的人数,以及每个员工拿到的彩票的号码。

线程运行后打印格式如下： 编号为: 2 的员工 从后门 入场! 拿到的双色球彩票号码是: [17, 24, 29, 30, 31, 32, 07] 编号为: 1 的员工 从后门 入场! 拿到的双色球彩票号码是: [06, 11, 14, 22, 29, 32, 15] //..... 从后门入场的员工总共: 13 位员工 从前门入场的员工总共: 87 位员工

练习题7：

编写多线程应用程序，模拟多个人通过一个山洞的模拟。这个山洞每次只能通过一个人，每个人通过山洞的时间为5秒，随机生成10个人，同时准备过此山洞，显示一下每次通过山洞人的姓名。

练习题8：

目前有三个类，里面都有打印的方法，类A的该方法可以打印出10个A,类B的方法可以打印出10个B,类C的方法可以打印出10个C.请利用多线程（提示wait,notify）实现轮流打印出ABC.