并发基础（3）： java 线程优先级小试牛刀

一、概述

多线程的优先级，小伙伴们应该都或多或少的用过或者见到过，但是，对于具体用法可能还是有点不太清楚，这篇文章就对这个问题进行一个探讨，也欢迎小伙伴们一起留言讨论。

在不同的 JVM 中（JVM 也算是一个操作系统），有着不同的 CPU 调度算法，对于大部分的 JVM 来说，优先级也是调度算法中的一个参数。

所以，**线程优先级在一定程度上，对线程的调度执行顺序有所影响，但不能用于保证线程的执行顺序，因为优先级仅仅是其中一个参数而已，其他参数还可能有线程的等待时间、执行时间等。**而且操作系统也可抗能可以完全不用理会 JAVA 线程对于优先级的设定。

线程优先级的范围一般是 1~10，默认是 5，但也有的 JVM 不是这个范围。所以，一般也尽量不要设置优先级为数字，可以使用 Thread 类的 3 个静态字段：

static int MAX_PRIORITY ： 线程可以具有的最高优先级。

static int MIN_PRIORITY ： 线程可以具有的最低优先级。

static int NORM_PRIORITY ： 分配给线程的默认优先级。

同时。对于需要较多 CPU 时间的线程需要设置较低的优先级，这样可以确保处理器不会被独占。

二、实例应用

一直在思考怎么设计，才能用简单明了的例子来证明优先级对线程的执行顺序有影响，最后为了严谨，还是不得不用到线程锁，可能对初学者来说，有点难理解。不过，思路是很清晰的：就是如何让 10 个线程一起同时并发。首先让创建的 10 个线程依次进入对象锁的池中等待，然后当 10 个线程创建完后，main 线程（主线程）同时唤醒这 10 个线程，于是 10 个线程同时一起并发竞争 CPU，只计算 5 次，看看线程的结束的先后顺序。（**注意：之所以线程的执行次数限制在 5 次，而不是无限，是因为会发生线程饥饿，高优先级线程占用着 CPU，导致低优先级的线程无法被调度！！**）

```
public class Test2 {
    //obj 对象 用于作为对象锁
    static String obj="";
    public static void main(String[] args) {
   //创建十个不同优先级的线程
      for(int i=1;i<10;i++){
          Thread t = new Thread(new Task(),"Thread_"+i);
          t.setPriority(i);
        //线程启动，进入就绪队列
```

```
        t.start();
        try {
        //当前线程--main 线程 休眠 100 毫秒，确保线程 t 已经创建完成，并能运行到等待获取锁处
         Thread.sleep(100);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
     }
     synchronized (obj) {
        //main 线程获取对象锁后，唤醒等待在该对象池上的所有线程--就是上面创建的 10 个线程
        obj.notifyAll();
     }
}
}
  //实现 Runnable 接口
  class Task implements Runnable{
    @Override
    public void run() {
        synchronized (Test2.obj) {
            try {
                //在对象 Test2.obj 上等待，
                Test2.obj.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //线程唤醒后，执行下面的代码
        int count=0;
        //计算 5 次，不能是 while(true),不限制执行次数，否则会发生线程饥饿
        while(count<5){
            count++;
            System.out.println(Thread.currentThread().getName());
            //每次计算完后，让出 CPU，重新进入就绪队列，与其他线程一起竞争 CPU
            Thread.yield();
        }
        System.out.println(Thread.currentThread().getName()+"结束");
    }
  }
```

测试结果：

从结果可以看出，**虽然线程的结束顺序不是完全按照优先级高低，但也基本是优先级高的线程结束的较快，被 CPU 调度的概率越大。**

```
Thread_9
   Thread_5
   Thread_6
   Thread_8
   Thread_7
   Thread_9
   Thread_8
   Thread_7
   Thread_9
   Thread_8
   Thread_9
   Thread_7
   Thread_9
   Thread_7
   Thread_8
   Thread_9 结束
   Thread_8
   Thread_7
   Thread_8 结束
   Thread_7 结束
   Thread_4
   Thread_5
   Thread_6
   Thread_3
   Thread_5
   Thread_6
   Thread_5
   Thread_6
   Thread_5
   Thread_6
   Thread_5 结束
   Thread_6 结束
   Thread_4
   Thread_2
   Thread_3
   Thread_4
   Thread_1
```

```
Thread_3
Thread_4
Thread_3
Thread_4
Thread_3
Thread_4 结束
Thread_3 结束
Thread_2
Thread_1
Thread_2
Thread_1
Thread_2
Thread_1
Thread_2
Thread_1
Thread_1 结束
Thread_2 结束
```