Thread 与 Object中的重要方法详解

问题?

考考你

1. 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？而sleep定义在Thread类里？
2. 用3种方式实现生产者模式
3. Java SE 8和Java 1.8和JDK 8是什么关系，是同一个东西吗？
4. Join和sleep和wait期间线程的状态分别是什么？为什么？

1. 概览

方法概览

类	方法名	简介
Thread	sleep相关	本表格的“相关”，指的的重载方法，也就是方法名相同，但是参数不同，例如sleep有多个方法，只是参数不同，实际作用大同小异
	join	等待其他线程执行完毕
	yield相关	放弃已经获取到的CPU资源
	currentThread	获取当前执行线程的引用
	start, run相关	启动线程相关
	interrtupt相关	中断线程
	stop()，suspend(),resuem()相关	已废弃
Object	wait/notify/notifyAll相关	让线程暂时休息和唤醒

3. wait / notify / notify all 方法

3.1 作用、用法

1. 阻塞阶段的使用

直到以下4种情况之一发生时，才会被唤醒

- 另一个线程调用这个对象的notify()方法且刚好被唤醒的是本线程；
- 另一个线程调用这个对象的notifyAll()方法；
- 过了wait(long timeout)规定的超时时间，如果传入0就是永久等待；
- 线程自身调用了interrupt()

4.展示 wait 和 notify的基本用法

4.1 wait 方法会 释放锁资源

```
/**
 * 展示 wait 和 notify的基本用法， 1. 研究代码的执行顺序 2. 证明wait释放锁资源
 */
public class Wait {

    // 1. 定义object方法
    public static Object object = new Object();

    static class Thread1 extends Thread{
        @Override
        public void run() {
            // 使用synchronize进行代码块的同步
            synchronized (object){
                System.out.println(Thread.currentThread().getName() + "开始执行
了");

                try {
                    object.wait();  //wait 方法会释放锁资源， 可以被中断，所以需要捕获中
断异常
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("线程" + Thread.currentThread().getName() +
"获得了锁");
            }
        }
    }

    static class Thread2 extends Thread{
        @Override
        public void run() {
            synchronized (object){
                // 调用同一个对象进行 唤醒 wait的线程
```

```
                object.notify();
                System.out.println("线程" + Thread.currentThread().getName() +
"调用了 notify()");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {

        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();
        thread1.start();
        Thread.sleep(10);
        thread2.start();
    }
}
```

4.2 notify notifyAll

```
/**
 *  1.  3 个线程，  线程1 和线程2 首先被阻塞，线程3唤醒他们。 notify notifyAll
 *  2.  展示 start 先执行不代表线程先启动 （由操作系统线程进行调度）
 */
public class WaitNotifyall implements Runnable{

    private static  final Object resourceA = new Object();
    @Override
    public void run() {
        synchronized (resourceA){
            System.out.println(Thread.currentThread().getName() + " 获取到了资源");


            try {
                System.out.println(Thread.currentThread().getName() + " 即将等
待");
                resourceA.wait();

                System.out.println(Thread.currentThread().getName() + " 即将结
束");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Runnable r = new WaitNotifyall();

        Thread threadA = new Thread(r);
        Thread threadB = new Thread(r);

        Thread threadC = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (resourceA){
//                    resourceA.notifyAll();  //进行资源的唤醒
                    resourceA.notify();  //进行资源的唤醒
```

```
                    System.out.println("线程C 进行了唤醒");
                }
            }
        });

        threadA.start();
        threadB.start();
        Thread.sleep(200);// 注释后可以看到 先执行 start 方法不一定会优先执行
        threadC.start();

    }
}
```

4.3 只释放 monitor

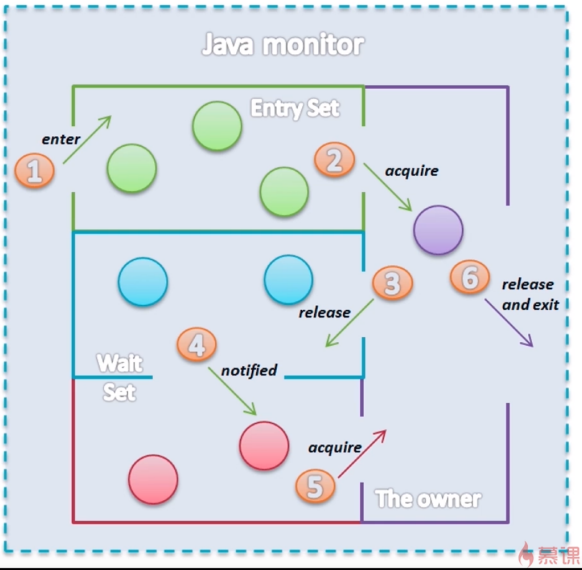
```
/**
 *  证明 wait 只释放当前的 那把锁
 */
public class WaitNotifyReleaseOwnMonitor {

    private static volatile Object resourceA = new Object();
    private static volatile Object resourceB = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread threadA = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (resourceA) {
                    System.out.println("ThreadA 获取到了 A锁");
                    synchronized (resourceB) {
                        System.out.println("ThreadA 获取到了 B锁");
                        try {
                            System.out.println("ThreadA 释放了 A锁");
                            resourceA.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        });

        Thread threadB = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (resourceA) {
                    System.out.println("ThreadB 获取到了 A锁");
                    System.out.println("ThreadB 尝试获取 B锁");
                    synchronized (resourceB) {
                        System.out.println("ThreadB 获取到了 B锁");
                    }
                }
            }
        });
        threadA.start();
        Thread.sleep(1000);
        threadB.start();
```

```
    }
}
```

5. 特点

wait原理

- 入口集 Entry Set
- 等待集 Wait Set



状态转化的特殊情况

- 从Object.wait()状态刚被唤醒时，通常不能立刻抢到monitor锁，那就会从Waiting先进入Blocked状态，抢到锁后再转换到Runnable状态（官方文档）
- 如果发生异常，可以直接跳到终止Terminated状态，不必再遵循路径，比如可以从Waiting直接到Terminated