1 sleep 方法解析
2 yield 方法解析
3 wait 方法解析
4 notify() 方法解析
5 notifyAll() 方法解析
6 join() 方法解析
7 几个方法的区别
8 六种线程状态
    sleep 方法测试代码
    yield 方法测试代码
    wait 方法测试代码
    join 方法测试
参考

1 sleep 方法解析

- 通过调用 Thread.sleep(..) 方法来**操作当前线程，不能操作其它线程**
- 调用 sleep 方法后线程进入 Time_Waiting 状态（有限等待状态），且**不会释放锁**，让出 CPU 时间，当时间到了，进入可运行状态(RUNNABLE)
- 调用sleep方法后，线程如果被中断，会抛出异常，并清除中断状态
- 如果线程先被中断，后调用 sleep 方法，也会抛异常，并清除中断状态
- **中断一个非活动线程没有任何效果**（线程运行之前调用 interrupt() 方法没有任何意义）
- 调用interrupt后线程为中断状态

```
//使当前执行的线程休眠（暂时停止执行）指定的毫秒数，取决于系统计时器和调度程序的精度和准确性。线程不会丢失任何监视器的所有权
//如果有线程中断了当前的线程(调用Thread.sleep(..)方法的线程)，则会抛出 InterruptedException 异常，并且抛出此异常之后，当前线程的中断状态将被清除
public static native void sleep(long millis) throws InterruptedException;
//精确到纳秒级别
public static native void sleep(long millis, int nanos) throws InterruptedException;
//如果某个线程调用了 wait(..)、join(..)、sleep(..)方法被阻塞后调用了这个方法，会清除中断状态，并抛出异常(翻译)
//中断一个非活动线程没有任何效果
public void interrupt() {...}
```

2 yield 方法解析

- 和 sleep() 方法一样都是 Thread 类中的静态方法
- 声明让出 CPU 时间，但可能继续执行，不释放锁
- 一般不建议使用该方法

```
//向调度器声明可以让出 CPU 时间，调度器可以忽略该声明，继续让该线程执行
//Yield是一种启发式尝试，用于改善线程之间的相对进展，否则会过度利用CPU。 它的使用应与详细的分析和基准测试相结合，以确保它实际上具有所需的效果。
//这个方法很少使用。 它可能对调试或测试目的很有用，它可能有助于重现因竞争条件而产生的错误。 在设计并发控制结构（例如java.util.concurrent.locks包中的结构）时，它也可能很有用
public static native void yield();
```

3 wait 方法解析

- 调用 wait() 无参方法后，线程为 WATING 状态（无限等待状态），调用由此方法进入 Time_WATING 状态（有限等待状态）
- 当前线程必须拥有一个监视器（在 synchronized 块之内）对象
- 使当前线程等待，直到另一个线程为此对象(当前线程拥有的监视器对象)调用notify() 方法或 notifyAll() 方法，或者已经过了一定的实时时间
- 此方法使当前线程将**自身置于监视器对象的等待队列中，并且释放该监视器对象的锁**。出于线程调度目的，此线程将被禁用，并处于休眠状态，直到发生以下四种情况之一：
  - 别的线程调用了监视器对象的 notify() 方法后，会随机唤醒一个该监视器对象等待队列中的休眠线程
  - 别的线程调用了监视器对象的 notifyAll() 方法后，会唤醒该监视器对象等待队列中的所有休眠线程
  - 别的线程中断了该线程（**注意调用 wait() 方法后线程已经没有锁啦**，会直接进入异常流程）
  - 如果参数大于0，则大约过了这个参数的时间(毫秒)，线程会自动苏醒。如果参数为0，必须等待前三个通知。小于0直接抛异常
- 如果一个线程获得了多个监视器对象，调用一个监视器对象的 wait() 方法只会释放这一个监视器对象，不会释放其它
- 当线程被唤醒后，会和其它线程一起公平竞争该监视器，一旦有个线程竞争成功，其它线程都会处于，线程调用 wait() 方法后的那个状态(等待队列中休眠)
- **注意：** 即使没有上述说的四种情况，**线程也可能会唤醒**（实际上虽然很少很少发生），所以建议 wait() 方法放在 while 修饰的条件下，而不是 if，防止异常唤醒！
- 调用 wait() 方法后被中断会抛异常

```
public final native void wait(long timeout) throws InterruptedException;
//很奇怪， nanos 参数只要在 0< nanos< 999999 之内，该方法会直接调用 wait(++timeout)，即都只是多了一个毫秒数
public final native void wait(long timeout, int nanos) throws InterruptedException;
```

4 notify() 方法解析

- 当前线程必须拥有一个监视器（在 synchronized 块之内）对象
- 唤醒正在等待此对象监视器的单个线程。如果有任何线程正在等待这个对象，那么将选择唤醒其中一个线程。选择是任意的
- 被唤醒的线程必须等待当前线程放弃监视器（退出 synchronized 块）后，才能继续

```
public final native void notify();
```

5 notifyAll() 方法解析

- 唤醒正在等待此对象监视器的所有线程
- 其余和 notify() 方法一样

6 join() 方法解析

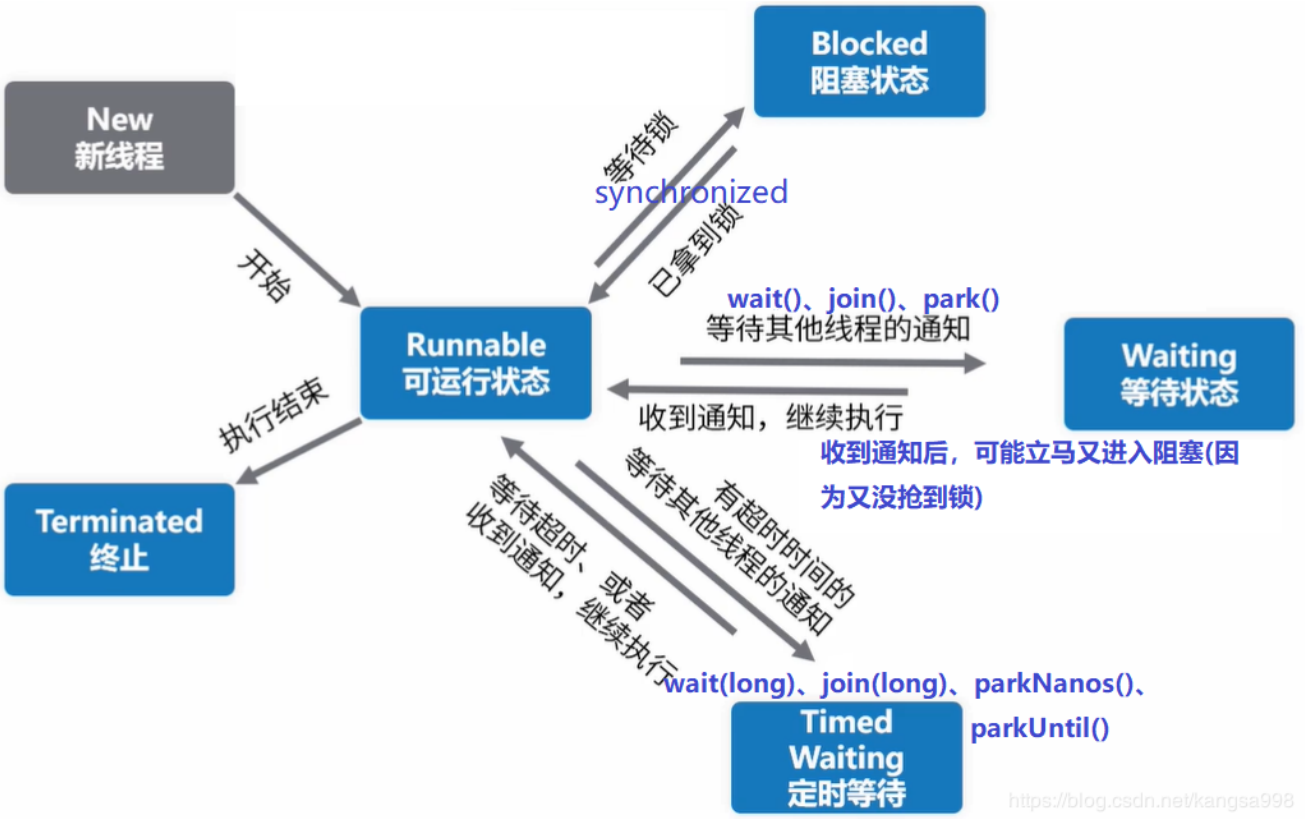
- **关键：** t.join(..) 方法已经被 synchronized 修饰了！所以当前线程调用某个线程对象的 join() 方法，说明当前对象获取了这个线程对象的监视器！
- 可以看出 join() 方法内部主要是调用 wait() 方法，并且使用 while 包围了 wait() 方法，这样当不正确唤醒当前线程时，**当条件不满足时，仍然会回归正轨！**
- 当 t.join() 方法参数为0时，调用这个方法的线程必须一直等到线程 t 死亡后，才能继续运行
- 当 t.join() 方法参数大于0时，调用这个方法的线程等到线程 t 死亡 或 等待参数的毫秒时间后，会继续运行

```
public final synchronized void join(long millis)
    throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {//判断this对象线程是否存活，如果存活就一直等待！只有被唤醒了，并且this
对象线程死了，才会退出 while！
            wait(0);// 当 this 线程死之前，会调用 notifyAll() 方法，唤醒所有等待线程，
            //所以，只有 this 线程死了，才会退出循环
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;//如果millis不为0，会从这里退出while循环！
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

7 几个方法的区别

- sleep() 和 yield() 方法是 Thread 类的静态本地方法
- join() 方法是 Thread 类的实例方法，内部调用 wait() 方法
- wait()、 notify()、notifyAll() 方法是 Object 类的实例本地方法（成员方法是类的所有方法）
- wait()、sleep()、join() 方法会抛中断异常，其它不会

8 六种线程状态




```
public enum State {
    //尚未启动的线程处于此状态。
    NEW,
    //在Java虚拟机中执行的线程处于此状态。但它可能正在等待来自操作系统的其他资源，例如 CPU 时间。
    RUNNABLE,
    //被阻塞等待监视器锁定的线程处于此状态。
    //执行 synchornized 代码块，抢监视器锁失败后，进入此状态
    BLOCKED,
    //无限期等待另一个线程执行*特定操作*的线程处于此状态。
    //调用 wait()、join()、LockSupport.park() 方法，进入无限等待状态
    WAITING,
    //在指定的等待时间内等待另一个线程执行操作的线程处于此状态。
    //调用 wait(long)、join(long)、parkNanos()、parkUntil() 方法进入有限等待状态
    TIMED_WAITING,
    //已退出的线程处于此状态。线程完全执行完毕后
    TERMINATED;
}
```

sleep 方法测试代码

```
    /***
     *  @Description: 调用 sleep 方法后线程进入 Time_Waiting 状态，且不会释放锁
     */
    public void 调用sleep线程的运行状态和锁释放情况() throws InterruptedException {
        Thread sleepTheard = new Thread(()->{
            System.out.println(Thread.currentThread().getName() + " 的状态是：" +
Thread.currentThread().getState().toString());
            try {
```

```
                synchronized (lock) {
                    System.out.println("获取锁了");
                    Thread.sleep(5000);
                    System.out.println("释放锁了");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"sleepThread");
        sleepTheard.start();

        Thread.sleep(1000);
        System.out.println(sleepTheard.getName() + " 的状态是：" +
sleepTheard.getState().toString());

        synchronized (lock) {
            System.out.println("才进入");
        }
        System.out.println(sleepTheard.getName() + " 的状态是：" +
sleepTheard.getState().toString());
    }

    /**
     * @Description: 调用sleep方法后，线程如果被中断，会抛出异常，并清除中断状态
     */
    public void 调用sleep被中断后的状态() throws InterruptedException {
        Thread sleepThread = new Thread(()->{
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                System.out.println("线程被中断抛异常后是否是中断状态：" +
Thread.currentThread().isInterrupted());
            }
        });

        sleepThread.start();
        Thread.sleep(200);
        sleepThread.interrupt();
    }

    /**
     * @Description: 中断一个非活动线程没有任何效果！
     */
    public void 中断一个非活动线程没有任何效果() throws InterruptedException {
        Thread sleepThread = new Thread(()->{
            try {
                Thread.sleep(1000);
                System.out.println("线程运行了");
            } catch (InterruptedException e) {
                e.printStackTrace();
                System.out.println("线程被中断抛异常后是否是中断状态：" +
Thread.currentThread().isInterrupted());
```

```
            }
        });

        sleepThread.interrupt();
        System.out.println("线程被中断后是否是中断状态：" + sleepThread.isInterrupted());

        sleepThread.start();
        Thread.sleep(2000);
    }

    /**
     * @Description: 调用interrupt后线程为中断状态
     */
    public void 调用interrupt后线程的中断状态() throws InterruptedException {
        Thread sleepThread = new Thread(()->{
            while (flag) {}
        });

        sleepThread.start();
        Thread.sleep(200);
        sleepThread.interrupt();
        System.out.println("调用interrupt后线程是否是中断状态：" +
sleepThread.isInterrupted());
        flag = Boolean.FALSE;
    }

    public void 先调用interrupt后调用sleep() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            while (flag) {

            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("线程被中断抛异常后是否是中断状态：" +
Thread.currentThread().isInterrupted());
                e.printStackTrace();
            }
        });
        t1.start();
        Thread.sleep(1000);

        t1.interrupt();
        System.out.println("线程t1中断状态为：" + t1.isInterrupted());

        flag = Boolean.FALSE;
        Thread.sleep(2000);
    }
```

yield 方法测试代码

```
public void yield声明让出CPU但可能继续执行() {
```

```
    Thread t1 = new Thread(() -> {
        while (true) {
            Thread.yield();
            System.out.print("1 ");
        }        });
    Thread t2 = new Thread(() -> {
        while (true) {
            Thread.yield();
            System.out.print("2 ");
        }
    });
    t1.start();
    t2.start();
    // 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 1 2 2 2 2 2 1 1 1 1 1 1 1 ,说明确实让出CPU了（有一段特别规律），但可能继续执行（有一点整个都是一个线程的值）
}
```

wait 方法测试代码

```
public void 调用wait方法后线程的状态() throws InterruptedException {
    Thread waitThread = new Thread(() -> {
        synchronized (lock) {
            try {
                System.out.println(Thread.currentThread().getName() + " 调用 wait 方法之间的状态是：" + Thread.currentThread().getState().toString());
                lock.wait(2000);
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    },"waitThread");
    waitThread.start();
    Thread.sleep(1000);
    System.out.println(waitThread.getName() + " 调用 wait 方法之后的状态是：" + waitThread.getState().toString());
    Thread.sleep(2000);
    System.out.println(waitThread.getName() + " 调用 wait 方法之后的状态是：" + waitThread.getState().toString());
    System.exit(0);

}
```

join 方法测试

```
public void 调用join方法线程完了才会往下走() throws InterruptedException {
    Thread t =  new Thread(() -> {
        try {
            Thread.sleep(40000);
            System.out.println("0");
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println(1);
    });
    t.start();
    t.join();
    System.out.println("1");

}
```

参考

JDK 1.8u171