LockSupport

2022年10月21日 22:03

LockSupport是java.util.concurrent.locks包下的一个子类，LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。

LockSupport中的**park**()和**unpark**()的作用分别是阻塞线程和解除阻塞线程

所有方法 | 静态方法 | 具体的方法

Modifier and Type	Method and Description
static Object	getBlocker(Thread t) 返回提供给最近调用尚未解除阻塞的park方法的阻止程序对象，如果不阻止则返回null。
static void	park() 禁止当前线程进行线程调度，除非许可证可用。
static void	park(Object blocker) 禁止当前线程进行线程调度，除非许可证可用。
static void	parkNanos(long nanos) 禁用当前线程进行线程调度，直到指定的等待时间，除非许可证可用。
static void	parkNanos(Object blocker, long nanos) 禁用当前线程进行线程调度，直到指定的等待时间，除非许可证可用。
static void	parkUntil(long deadline) 禁用当前线程进行线程调度，直到指定的截止日期，除非许可证可用。
static void	parkUntil(Object blocker, long deadline) 禁用当前线程进行线程调度，直到指定的截止日期，除非许可证可用。
static void	unpark(Thread thread) 为给定的线程提供许可证（如果尚未提供）。

1、3种让线程等待和唤醒的方法

有三种方式可以做到让线程等待和唤醒：

- 使用Object中的**wait**()方法让线程等待，使用Object中的**notify**()方法唤醒线程
- 使用JUC包中的**condition**的**await**()方法让线程等待，使用**signal**()方法唤醒线程
- **LockSupport**类可以阻塞当前线程以及唤醒指定被阻塞的线程

下面请看三种方式如何使用，以及他们的弊端

（1）Object类中的wait和notify方法实现线程等待和唤醒

正常情况：

```
Object objectLock = new Object();

// 新建一个线程t1
new Thread(() -> {
    synchronized (objectLock) {
        System.out.println(Thread.currentThread().getName() + "\t----come in");

        try {
            // 线程等待，如果该线程阻塞，其他线程可以持有锁
            objectLock.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 线程被唤醒之后，打印一句话
        System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
    }
}, "t1").start();

// 暂停1秒钟线程
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

```
new Thread(() -> {
    synchronized (objectLock) {
        // 唤醒持有objectLock这把锁的线程
        objectLock.notify();
        System.out.println(Thread.currentThread().getName() + "\t----唤醒其他线程");
    }
} ,"t2").start();
```

正常情况下，t1会被t2唤醒，控制台打印如下：

```
t1      ----come in
t2      ----唤醒其他线程
t1      ----被唤醒
```

异常情况1：wait和notify不在同步代码块或者同步方法中使用

以下代码相比于上面的代码，仅仅把synchronized注释掉了

```
Object objectLock = new Object();

// 新建一个线程t1
new Thread(() -> {
    //synchronized (objectLock) {
        System.out.println(Thread.currentThread().getName() + "\t----come in");
        try {
            // 线程等待，如果该线程阻塞，其他线程可以持有锁
            objectLock.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 线程被唤醒之后，打印一句话
        System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
    //}
}, "t1").start();

// 暂停1秒钟线程
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}

new Thread(() -> {
    //synchronized (objectLock) {
        // 唤醒持有objectLock这把锁的线程
        objectLock.notify();
        System.out.println(Thread.currentThread().getName() + "\t----唤醒其他线程");
    //}
} ,"t2").start();
```

如果不在同步代码块或者同步方法中使用wait和notify，那么将会报以下错：

```
t1  ----come in
Exception in thread "t1" java.lang.IllegalMonitorStateException Create breakpoint
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:502)
    at com.wzq.lockSupports.ObjectWaitAndNotify.lambda$main$0(ObjectWaitAndNotify.java:24) <1 internal line>
Exception in thread "t2" java.lang.IllegalMonitorStateException Create breakpoint
    at java.lang.Object.notify(Native Method)
    at com.wzq.lockSupports.ObjectWaitAndNotify.lambda$main$1(ObjectWaitAndNotify.java:44) <1 internal line>
```

即如果使用wait和notify，必须把它包在synchronized之间

异常情况2：先notify唤醒再wait阻塞线程

```
Object objectLock = new Object();

new Thread(() -> {
```

```
    // 暂停1秒钟线程
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    synchronized (objectLock) {
        System.out.println(Thread.currentThread().getName() + "\t----come in");

        try {
            // 线程等待，如果该线程阻塞，其他线程可以持有锁
            objectLock.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 线程被唤醒之后，打印一句话
        System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
    }
}, "t1").start();

new Thread(() -> {
    synchronized (objectLock) {
        // 唤醒持有ObjectLock锁的t1线程
        objectLock.notify();
        System.out.println(Thread.currentThread().getName() + "\t----唤醒其他线程");
    }
} ,"t2").start();
```

在这种情况下：程序不会结束，因为t1线程一直在wait

所以，使用Object类中的wait和notify方法有弊端，必须这样才可以正常使用：

- wait和notify方法必须要在同步块或者方法里面，且成对出现使用
- 先wait后notify才可以正常运行

（2）Condition接口中的await后signal方法实现线程的等待和唤醒

正常情况：

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

// t1 线程负责等待
new Thread(() -> {
    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + "\t----come in");
        condition.await();
        System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
} ,"t1").start();

// 暂停几秒钟线程
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}

// t2线程，负责唤醒t1线程
new Thread(() -> {
```

```
    lock.lock();
    try {
        condition.signal(); // 唤醒其他线程
        System.out.println(Thread.currentThread().getName() + "----唤醒其他线程");
    } finally {
        lock.unlock();
    }
} ,"t2").start();
```

正常情况下，线程能够被正常唤醒，输出如下：

```
t1      ----come in
t2      ----唤醒其他线程
t1      ----被唤醒
```

异常情况1：注释掉lock锁块，即不在lock和unlock对里面使用Condition的await和signal

这种方法直接报错：

```
t1  ----come in
Exception in thread "t1" java.lang.IllegalMonitorStateException <4 internal lines>
    at com.wzq.lockSupports.ConditionAwaitAndSignal.lambda$main$0(ConditionAwaitAndSignal.java:26) <1 internal line>
Exception in thread "t2" java.lang.IllegalMonitorStateException <1 internal line>
    at com.wzq.lockSupports.ConditionAwaitAndSignal.lambda$main$1(ConditionAwaitAndSignal.java:46) <1 internal line>
```

condition.await()和condition.signal()都出发了IllegalMonitorStateException异常

只有在lock、unlock对里面，才能正确调用condition中的线程等待和唤醒方法

异常情况2：先唤醒再等待

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

new Thread(() -> {
    // 先睡一秒
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // 上锁
    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + "\t----come in");
        // 使线程等待
        condition.await();
        System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}, "t1").start();

new Thread(() -> {
    lock.lock();
    try {
        // 唤醒线程
        condition.signal();
        System.out.println(Thread.currentThread().getName() + "\t----唤醒线程");
    } finally {
        lock.unlock();
    }
}, "t2").start();
```

在这种情况下，程序永远不会停止，因为t1线程一直在等待

所以，要想使用condition的线程等待和唤醒方法，需要先获取锁；一定要先await后signal

上述两个对象Object和Condition使用的限制条件：

- 线程先要获得并持有锁，必须在锁块（synchronized或lock）中
- 必须要先等待后唤醒，线程才能够被唤醒

4、LockSupport类中的park等待和unpark唤醒

LockSupport通过**park**和**unpark**方法（**都是静态方法**）来实现阻塞和唤醒线程的操作！它的官网解释是这样的：

LockSupport是用来创建锁和其他同步类基本线程阻塞原语。
LockSupport类是用了一种**Permit（许可）**的概念来做到阻塞和唤醒线程的功能，每个线程都有一个许可（Permit）
但与Semaphore不同，许可的累加上限是1。（即每个线程只有一份许可证，即使给多份许可证，线程拿到的也只有一个）

主要方法：

- 阻塞：调用**park / park(Object blocker)**，阻塞当前线程/阻塞传入的具体线程
  permit许可证默认没有，不能放行。所以一开始调用park()方法当前线程就会阻塞。
  直到别的线程给当前线程发放permit，park方法才会被唤醒
- 唤醒：调用**unpark(Thread thread)**，唤醒处于阻塞状态的指定线程
  调用unpark(thread)方法后，就会将thread线程的许可证permit发放，自动唤醒park线程，
  即之前阻塞中的LockSupport.park()方法会立即返回。

LockSupport可以解决Object和Lock如前文所述的痛点。

正常调用：

```
Thread t1 = new Thread(() -> {
    System.out.println(Thread.currentThread().getName() + "\t----come in");
    // 使用park，等待其他线程发许可证
    LockSupport.park();
    System.out.println(Thread.currentThread().getName() + "\t----被唤醒");
}, "t1");
t1.start();

// 暂停几秒线程（main）
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}

new Thread(() -> {
    LockSupport.unpark(t1);
    System.out.println(Thread.currentThread().getName()+"\t----唤醒线程");
} ,"t2").start();
```

提前唤醒线程：

```
Thread t1 = new Thread(() -> {
    // t1线程先休眠1秒钟
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // 打印被阻塞前的时间
    System.out.println(Thread.currentThread().getName() + "\t" + System.currentTimeMillis());
    LockSupport.park(); // 线程等待
    System.out.println(Thread.currentThread().getName() + "\t" + System.currentTimeMillis() + "\t被唤醒");
```

```
}, "t1");
t1.start();

// t2线程唤醒t1线程
new Thread(() -> {
    LockSupport.unpark(t1); // 唤醒t1线程
    System.out.println(Thread.currentThread().getName() + "\t----唤醒线程");
}, "t2").start();
```

```
t2  ----唤醒线程
t1  1666528460189
t1  1666528460189  被唤醒
```

时间一致，说明t1根本没有等待拿到许可证，直接继续执行

但是LockSupport的park和unpark必须成双成对出现！

重点说明：

许可证最多只能有一个！不会累计，park之后就没有许可证了，即使发多个许可证，也只能持有一个。以下这个程序会一直阻塞。

```
Thread t1 = new Thread(() -> {
    // t1线程先休眠1秒钟
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // 打印被阻塞前的时间
    System.out.println(Thread.currentThread().getName() + "\t" + System.currentTimeMillis());
    LockSupport.park(); // 这里拿到许可证，就没有许可证了
    System.out.println("--------");
    LockSupport.park(); // 已经没有许可证了，所以他会一直等待
    System.out.println(Thread.currentThread().getName() + "\t" + System.currentTimeMillis() + "\t被唤醒");
}, "t1");
t1.start();

// t2线程唤醒t1线程
new Thread(() -> {
    // 许可证不会累计，即使发4个，也只有1个
    LockSupport.unpark(t1);
    LockSupport.unpark(t1);
    LockSupport.unpark(t1);
    LockSupport.unpark(t1);
    System.out.println(Thread.currentThread().getName() + "\t----唤醒线程");
}, "t2").start();
```

总结：

LockSupport是用来创建锁和其他同步类的基本线程阻塞原语

LockSupport是一个线程阻塞工具类，所有的方法都是静态方法，可以让线程在任意位置阻塞，阻塞之后也有对应的唤醒方法。

归根结底，LockSupport调用的Unsafe中的native代码

LockSupport提供park和unpark方法实现阻塞线程和解除线程阻塞的过程。

LockSupport和每个使用它的线程都有一个许可（permit）关联。

每个线程都有一个相关的permit，permit最多只有一个，重复调用unpark也不会积累凭证。

形象的理解

线程阻塞需要消耗凭证（permit），这个凭证最多只有1个。

当调用park方法时：

- 如果有凭证，则会直接消耗掉这个凭证然后正常退出；
- 如果无凭证，就必须阻塞等待凭证可用；

而unpark则相反，它会增加一个凭证，但凭证最多只能有1个，累加无效。

2、面试题

为什么可用突破wait/notify的原有调用顺序?

因为unpark获得了一个凭证，之后再调用park方法，就可以名正言顺的凭证消费，故不会阻塞。
先放了凭证后续可以通畅无阻。

为什么唤醒两次后阻塞两次，但最终结果还会阻塞线程?

因为凭证的数量最多为1，连续调用两次unpark和调用一个unpark效果一样，只会增加一个凭证；
而调用两次park却需要消费两个凭证，证不够，不能放行。