JUC锁: LockSupport详解

LockSupport是锁中的基础，是一个提供锁机制的工具类，所以先对其进行分析。

面试问题去理解

- 为什么LockSupport也是核心基础类? AQS框架借助于两个类： Unsafe(提供CAS操作)和LockSupport(提供park/unpark操作)
- 写出分别通过wait/notify和LockSupport的park/unpark实现同步?
- LockSupport.park()会释放锁资源吗? 那么Condition.await()呢?
- Thread.sleep()、Object.wait()、Condition.await()、LockSupport.park()的区别? 重点
- 如果在wait()之前执行了notify()会怎样?
- 如果在park()之前执行了unpark()会怎样?

LockSupport简介

LockSupport用来创建锁和其他同步类的基本线程阻塞原语。简而言之，当调用LockSupport.park时，表示当前线程将会等待，直至获得许可，当调用LockSupport.unpark时，必须把等待获得许可的线程作为参数进行传递，好让此线程继续运行。

LockSupport源码分析

类的属性

```
public class LockSupport {
    // Hotspot implementation via intrinsics API
    private static final sun.misc.Unsafe UNSAFE;
    // 表示内存偏移地址
    private static final long parkBlockerOffset;
    // 表示内存偏移地址
    private static final long SEED;
    // 表示内存偏移地址
    private static final long PROBE;
    // 表示内存偏移地址
    private static final long SECONDARY;

    static {
        try {
            // 获取Unsafe实例
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            // 线程类类型
            Class<?> tk = Thread.class;
```

```
            // 获取Thread的parkBlocker字段的内存偏移地址
            parkBlockerOffset = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("parkBlocker"));
            // 获取Thread的threadLocalRandomSeed字段的内存偏移地址
            SEED = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("threadLocalRandomSeed"));
            // 获取Thread的threadLocalRandomProbe字段的内存偏移地址
            PROBE = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("threadLocalRandomProbe"));
            // 获取Thread的threadLocalRandomSecondarySeed字段的内存偏移地址
            SECONDARY = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("threadLocalRandomSecondarySeed"));
        } catch (Exception ex) { throw new Error(ex); }
    }
}
```

说明: UNSAFE字段表示sun.misc.Unsafe类，查看其源码，点击在这里，一般程序中不允许直接调用，而long型的表示实例对象相应字段在内存中的偏移地址，可以通过该偏移地址获取或者设置该字段的值。

类的构造函数

```
// 私有构造函数，无法被实例化
private LockSupport() {}
```

说明: LockSupport只有一个私有构造函数，无法被实例化。

核心函数分析

在分析LockSupport函数之前，先引入sun.misc.Unsafe类中的park和unpark函数，因为LockSupport的核心函数都是基于Unsafe类中定义的park和unpark函数，下面给出两个函数的定义:

```
public native void park(boolean isAbsolute, long time);
public native void unpark(Thread thread);
```

说明: 对两个函数的说明如下:

- park函数，阻塞线程，并且该线程在下列情况发生之前都会被阻塞:
- ① 调用unpark函数，释放该线程的许可。
- ② 该线程被中断。
- ③ 设置的时间到了。并且，当time为绝对时间时，isAbsolute为true，否则，isAbsolute为false。当time为0时，表示无限等待，直到unpark发生。
- unpark函数，释放线程的许可，即激活调用park后阻塞的线程。这个函数不是安全的，调用这个函数时要确保线程依旧存活。

park函数

park函数有两个重载版本，方法摘要如下

```
public static void park();
public static void park(Object blocker);
```

说明: 两个函数的区别在于park()函数没有没有blocker，即没有设置线程的parkBlocker字段。park(Object)型函数如下。

```
public static void park(Object blocker) {
    // 获取当前线程
    Thread t = Thread.currentThread();
    // 设置Blocker
    setBlocker(t, blocker);
    // 获取许可
    UNSAFE.park(false, 0L);
    // 重新可运行后再此设置Blocker
    setBlocker(t, null);
}
```

说明: 调用park函数时，首先获取当前线程，然后设置当前线程的parkBlocker字段，即调用setBlocker函数，之后调用Unsafe类的park函数，之后再调用setBlocker函数。那么问题来了，为什么要在此park函数中要调用两次setBlocker函数呢? 原因其实很简单，调用park函数时，当前线程首先设置好parkBlocker字段，然后再调用Unsafe的park函数，此后，当前线程就已经阻塞了，等待该线程的unpark函数被调用，所以后面的一个setBlocker函数无法运行，unpark函数被调用，该线程获得许可后，就可以继续运行了，也就运行第二个setBlocker，把该线程的parkBlocker字段设置为null，这样就完成了整个park函数的逻辑。如果没有第二个setBlocker，那么之后没有调用park(Object blocker)，而直接调用getBlocker函数，得到的还是前一个park(Object blocker)设置的blocker，显然是不符合逻辑的。总之，必须要保证在park(Object blocker)整个函数执行完后，该线程的parkBlocker字段又恢复为null。所以，park(Object)型函数里必须要调用setBlocker函数两次。setBlocker方法如下。

```
private static void setBlocker(Thread t, Object arg) {
    // 设置线程t的parkBlocker字段的值为arg
    UNSAFE.putObject(t, parkBlockerOffset, arg);
}
```

说明: 此方法用于设置线程t的parkBlocker字段的值为arg。

另外一个无参重载版本，park()函数如下。

```
public static void park() {
    // 获取许可，设置时间为无限长，直到可以获取许可
    UNSAFE.park(false, 0L);
}
```

说明: 调用了park函数后，会禁用当前线程，除非许可可用。在以下三种情况之一发生之前，当前线程都将处于休眠状态，即下列情况发生时，当前线程会获取许可，可以继续运行。

- 其他某个线程将当前线程作为目标调用 unpark。
- 其他某个线程中断当前线程。
- 该调用不合逻辑地(即毫无理由地)返回。

parkNanos函数

此函数表示在许可可用前禁用当前线程，并最多等待指定的等待时间。具体函数如下。

```
public static void parkNanos(Object blocker, long nanos) {
    if (nanos > 0) { // 时间大于0
        // 获取当前线程
        Thread t = Thread.currentThread();
        // 设置Blocker
        setBlocker(t, blocker);
        // 获取许可，并设置了时间
        UNSAFE.park(false, nanos);
        // 设置许可
        setBlocker(t, null);
    }
}
```

说明: 该函数也是调用了两次setBlocker函数，nanos参数表示相对时间，表示等待多长时间。

parkUntil函数

此函数表示在指定的时限前禁用当前线程，除非许可可用, 具体函数如下:

```
public static void parkUntil(Object blocker, long deadline) {
    // 获取当前线程
    Thread t = Thread.currentThread();
    // 设置Blocker
    setBlocker(t, blocker);
    UNSAFE.park(true, deadline);
    // 设置Blocker为null
    setBlocker(t, null);
}
```

说明: 该函数也调用了两次setBlocker函数，deadline参数表示绝对时间，表示指定的时间。

unpark函数

此函数表示如果给定线程的许可尚不可用，则使其可用。如果线程在 park 上受阻塞，则它将解除其阻塞状态。否则，保证下一次调用 park 不会受阻塞。如果给定线程尚未启动，则无法保证此操作有任何效果。具体函数如下:

```
public static void unpark(Thread thread) {
    if (thread != null) // 线程为不空
        UNSAFE.unpark(thread); // 释放该线程许可
}
```

说明: 释放许可，指定线程可以继续运行。

LockSupport示例说明

使用*wait/notify*实现线程同步

```
class MyThread extends Thread {

    public void run() {
        synchronized (this) {
            System.out.println("before notify");
            notify();
            System.out.println("after notify");
```

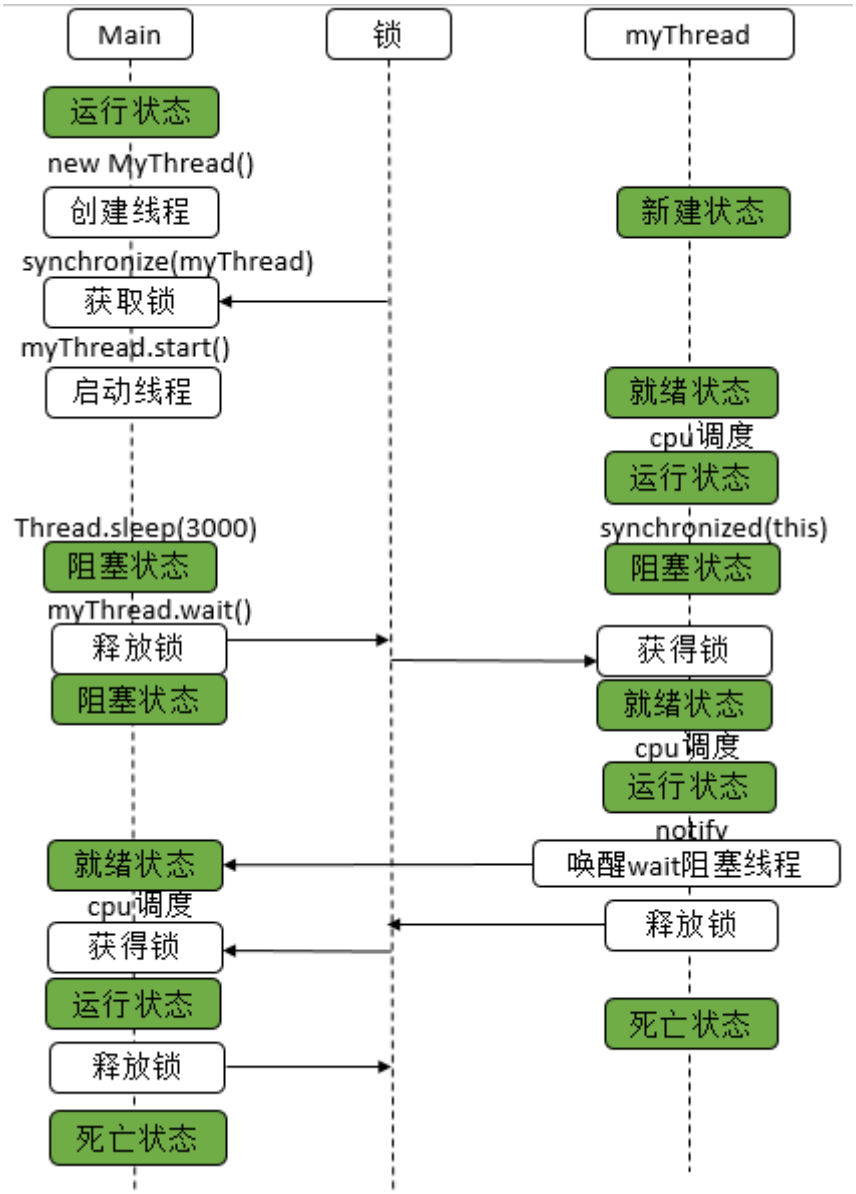
```
        }
    }
}

public class WaitAndNotifyDemo {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        synchronized (myThread) {
            try {
                myThread.start();
                // 主线程睡眠3s
                Thread.sleep(3000);
                System.out.println("before wait");
                // 阻塞主线程
                myThread.wait();
                System.out.println("after wait");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果

```
before wait
before notify
after notify
```

说明: 具体的流程图如下:



使用wait/notify实现同步时，必须先调用wait，后调用notify，如果先调用notify，再调用wait，将起不了作用。具体代码如下

```
class MyThread extends Thread {
    public void run() {
        synchronized (this) {
            System.out.println("before notify");
            notify();
            System.out.println("after notify");
        }
    }
}

public class WaitAndNotifyDemo {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        myThread.start();
        // 主线程睡眠3s
        Thread.sleep(3000);
        synchronized (myThread) {
            try {
                System.out.println("before wait");
                // 阻塞主线程
```

```
                myThread.wait();
                System.out.println("after wait");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果:

```
before notify
after notify
before wait
```

说明: 由于先调用了notify，再调用的wait，此时主线程还是会一直阻塞。

使用*park/unpark*实现线程同步

```
import java.util.concurrent.locks.LockSupport;

class MyThread extends Thread {
    private Object object;

    public MyThread(Object object) {
        this.object = object;
    }

    public void run() {
        System.out.println("before unpark");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 获取blocker
        System.out.println("Blocker info " + LockSupport.getBlocker((Thread) object));
        // 释放许可
        LockSupport.unpark((Thread) object);
        // 休眠500ms，保证先执行park中的setBlocker(t, null);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 再次获取blocker
        System.out.println("Blocker info " + LockSupport.getBlocker((Thread) object));

        System.out.println("after unpark");
    }
}

public class test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread(Thread.currentThread());
        myThread.start();
```

```
        System.out.println("before park");
        // 获取许可
        LockSupport.park("ParkAndUnparkDemo");
        System.out.println("after park");
    }
}
```

运行结果:

```
before park
before unpark
Blocker info ParkAndUnparkDemo
after park
Blocker info null
after unpark
```

说明: 本程序先执行park，然后在执行unpark，进行同步，并且在unpark的前后都调用了getBlocker，可以看到两次的结果不一样，并且第二次调用的结果为null，这是因为在调用unpark之后，执行了Lock.park(Object blocker)函数中的setBlocker(t, null)函数，所以第二次调用getBlocker时为null。

上例是先调用park，然后调用unpark，现在修改程序，先调用unpark，然后调用park，看能不能正确同步。具体代码如下

```
import java.util.concurrent.locks.LockSupport;

class MyThread extends Thread {
    private Object object;

    public MyThread(Object object) {
        this.object = object;
    }

    public void run() {
        System.out.println("before unpark");
        // 释放许可
        LockSupport.unpark((Thread) object);
        System.out.println("after unpark");
    }
}

public class ParkAndUnparkDemo {
    public static void main(String[] args) {
        MyThread myThread = new MyThread(Thread.currentThread());
        myThread.start();
        try {
            // 主线程睡眠3s
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("before park");
        // 获取许可
        LockSupport.park("ParkAndUnparkDemo");
        System.out.println("after park");
    }
}
```

运行结果:

```
before unpark
after unpark
before park
after park
```

说明: 可以看到，在先调用unpark，再调用park时，仍能够正确实现同步，不会造成由wait/notify调用顺序不当所引起的阻塞。因此park/unpark相比wait/notify更加的灵活。

中断响应

看下面示例

```
import java.util.concurrent.locks.LockSupport;

class MyThread extends Thread {
    private Object object;

    public MyThread(Object object) {
        this.object = object;
    }

    public void run() {
        System.out.println("before interrupt");
        try {
            // 休眠3s
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Thread thread = (Thread) object;
        // 中断线程
        thread.interrupt();
        System.out.println("after interrupt");
    }
}

public class InterruptDemo {
    public static void main(String[] args) {
        MyThread myThread = new MyThread(Thread.currentThread());
        myThread.start();
        System.out.println("before park");
        // 获取许可
        LockSupport.park("ParkAndUnparkDemo");
        System.out.println("after park");
    }
}
```

运行结果:

```
before park
before interrupt
after interrupt
after park
```

说明: 可以看到，在主线程调用park阻塞后，在myThread线程中发出了中断信号，此时主线程会继续运行，也就是说明此时interrupt起到的作用与unpark一样。

更深入的理解

*Thread.sleep()*和*Object.wait()*的区别

首先，我们先来看看Thread.sleep()和Object.wait()的区别，这是一个烂大街的题目了，大家应该都能说上来两点。

- Thread.sleep()不会释放占有的锁，Object.wait()会释放占有的锁；
- Thread.sleep()必须传入时间，Object.wait()可传可不传，不传表示一直阻塞下去；
- Thread.sleep()到时间了会自动唤醒，然后继续执行；
- Object.wait()不带时间的，需要另一个线程使用Object.notify()唤醒；
- Object.wait()带时间的，假如没有被notify，到时间了会自动唤醒，这时又分好两种情况，一是立即获取到了锁，线程自然会继续执行；二是没有立即获取锁，线程进入同步队列等待获取锁；

其实，他们俩最大的区别就是Thread.sleep()不会释放锁资源，Object.wait()会释放锁资源。

*Thread.sleep()*和*Condition.await()*的区别

Object.wait()和Condition.await()的原理是基本一致的，不同的是Condition.await()底层是调用LockSupport.park()来实现阻塞当前线程的。

实际上，它在阻塞当前线程之前还干了两件事，一是把当前线程添加到条件队列中，二是“完全”释放锁，也就是让state状态变量变为0，然后才是调用LockSupport.park()阻塞当前线程。

*Thread.sleep()*和*LockSupport.park()*的区别

LockSupport.park()还有几个兄弟方法——parkNanos()、parkUtil()等，我们这里说的park()方法统称这一类方法。

- 从功能上来说，Thread.sleep()和LockSupport.park()方法类似，都是阻塞当前线程的执行，且都不会释放当前线程占有的锁资源；
- Thread.sleep()没法从外部唤醒，只能自己醒过来；
- LockSupport.park()方法可以被另一个线程调用LockSupport.unpark()方法唤醒；
- Thread.sleep()方法声明上抛出了InterruptedException中断异常，所以调用者需要捕获这个异常或者再抛出；
- LockSupport.park()方法不需要捕获中断异常；
- Thread.sleep()本身就是一个native方法；
- LockSupport.park()底层是调用的Unsafe的native方法；

*Object.wait()*和*LockSupport.park()*的区别

二者都会阻塞当前线程的运行，他们有什么区别呢? 经过上面的分析相信你一定很清楚了，真的吗? 往下看!

- Object.wait()方法需要在synchronized块中执行；
- LockSupport.park()可以在任意地方执行；
- Object.wait()方法声明抛出了中断异常，调用者需要捕获或者再抛出；

- LockSupport.park()不需要捕获中断异常；
- Object.wait()不带超时的，需要另一个线程执行notify()来唤醒，但不一定继续执行后续内容；
- LockSupport.park()不带超时的，需要另一个线程执行unpark()来唤醒，一定会继续执行后续内容；
- 如果在wait()之前执行了notify()会怎样? 抛出IllegalMonitorStateException异常；
- 如果在park()之前执行了unpark()会怎样? 线程不会被阻塞，直接跳过park()，继续执行后续内容；

park()/unpark()底层的原理是“二元信号量”，你可以把它相像成只有一个许可证的Semaphore，只不过这个信号量在重复执行unpark()的时候也不会再增加许可证，最多只有一个许可证。

*LockSupport.park()*会释放锁资源吗*?*

不会，它只负责阻塞当前线程，释放锁资源实际上是在Condition的await()方法中实现的。