一.可重入锁

可重入锁又名递归锁
是指在同一个线程在外层方法获取锁的时候，再进去该线程的内层方法会自动获取锁（前提：同一个锁对象），不会因为之前已经获取过还没释放而阻塞。
java中的reentrantLock和Synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。

```
public class Demo2 {

    private static Object lock = new Object();

    public static void main(String[] args) {
        test();
    }

    public static void test() {
        new Thread(() -> {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + "外");
                synchronized (lock) {
                    System.out.println(Thread.currentThread().getName() + "中");
                    synchronized (lock) {
                        System.out.println(Thread.currentThread().getName() +
"内");
                    }
                }
            }
        }).start();
    }
}
```

synchronized的可重入实现原理

每个锁对象拥有一个锁计数器和一个指向持有该锁的线程的指针。
当执行monitorenter时，如果目标锁对象的计数器为0，那么说明他没有被其他线程锁持有，Java虚拟机会将改锁对象的持有线程设置为当前线程，并且将其计数器+1.
在目标锁对象的计数器不为0的情况下，如果锁对象的持有线程是当前线程，那么Java虚拟机可以将其计数器+1，否则需要等待，直至持有线程释放锁。
当执行monitorexit时，Java虚拟机则需要将锁对象的计数器-1.计数器为0代表释放锁。

```
public class Demo3 {

    private static Lock lock=new ReentrantLock();

    public static void main(String[] args) {
        m1();
    }
    public static void m1(){
        new Thread(()->{
            try {
                    lock.lock();
```

```
                System.out.println("第一次加锁");
                try {
                    lock.lock();
                    System.out.println("第2次加锁");
                }finally {
                    lock.unlock();
                }
            }finally {
                lock.unlock();
            }

        },"m1").start();
    }
}
```

二，LockSuport

用于创建锁和其他同步类的基本线程阻塞原语。
本质就是线程阻塞和唤醒wait notify的加强版
park() unpark()

线程阻塞唤醒的三种方法

1.使用Object中的wait()让线程等待，使用Object的notify()唤醒线程。

```
public class Demo4 {

    private static Object lock=new Object();

    public static void main(String[] args) {
        new Thread(()->{
            synchronized (lock){
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();

        new Thread(()->{

            synchronized (lock){
                lock.notify();
            }

        },"B").start();
    }

}
```

存在的问题：
必须在synchronized代码块里，顺序不能反。
2.使用juc中的Condition的await()让线程等待，使用signal()方法唤醒线程。

```
public class Demo5 {

    private static Lock lock = new ReentrantLock();
    private static Condition condition = lock.newCondition();

    public static void main(String[] args) {

        new Thread(() -> {
            try {
                lock.lock();
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }

        }, "A").start();

        new Thread(() -> {
            try {
                lock.lock();
                condition.signal();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }

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

存在的问题：
必须在lock代码块里面使用，使用顺序不能反。
3.LockSupport类park()可以阻塞当前线程以及unpark(Thread)唤醒指定被阻塞的线程。
LockSupport类使用了一种名为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只有1和0两个值，默认是0.
permit默认时0，所以一开始调用park()方法，当前线程就会阻塞，直到别的线程将当前线程的permit设置为1时，park方法会被唤醒，然后将permit再次设置为0并返回。

```
public class Demo6 {

    public static void main(String[] args) {

        Thread A = new Thread(() -> {
            System.out.println("A");
            LockSupport.park();
            //LockSupport.park();
            System.out.println("...");
        });
        A.start();

        new Thread(()->{
            System.out.println("B");
            LockSupport.unpark(A);
```

```
            LockSupport.unpark(A);
        }).start();
    }
}
```

三，总结

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。permit相当于1，0的开关，默认是0，调用一次unpark就加1变成1，调用一次park会消费permit，也就是将1变成o，同时park立即返回。
如再次调用park会变成阻塞(因为permit为零了会阻塞在这里，一直到permit变为1)，这时调用unpark会把permit置为1。
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。

为什么可以先唤醒线程后阻塞线程?

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。

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和调用一次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。