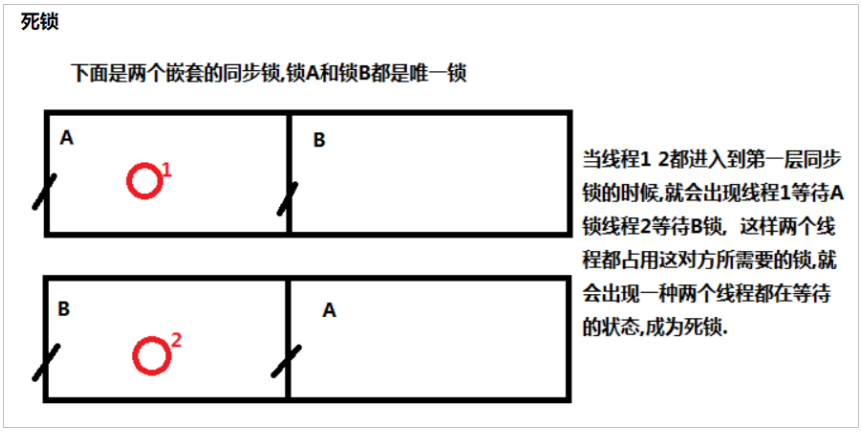
死锁

一、基本介绍

当线程任务中出现了多个同步(多个锁) 时，如果同步中嵌套了其他的同步。这时容易引发一种现象：程序出现无限等待，这种现象我们称为死锁。



二、死锁的四个必要条件

- 互斥条件: 一个资源每次只能被一个进程使用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件: 进程已获得的资源，在未使用完之前，不能强行剥夺。
- 循环等待条件: 若干进程之间形成一种头尾相接的循环等待资源关系。

三、线程通信（解决死锁）

1、等待与唤醒(wait()、notify()、notifyAll())

- wait（） :等待，将正在执行的线程释放其执行资格 和 执行权，并存储到线程池中。

- notify（）：唤醒，唤醒线程池中被wait（）的线程，一次唤醒一个，而且是任意的。
- notifyAll（）： 唤醒全部：可以将线程池中的所有wait() 线程都唤醒。

所谓唤醒的意思就是让 线程池中的线程具备执行资格。必须注意的是，这些方法都是在 同步中才有效。

同时这些方法在使用时必须标明所属锁，这样才可以明确出这些方法操作的到底是哪个锁上的线程。

等待中的线程必须由notify()方法显式地唤醒，否则它会永远地等待下去。

2、使用注意：

这三个方法均只能使用在同步代码块或者同步方法中。

使用时必须标识它们所操作的线程持有的锁，因为等待和唤醒必须是同一锁下的线程；

3、sleep和wait的异同：

相同点：一旦执行方法以后，都会使得当前的进程进入阻塞状态

不同点：

1.两个方法声明的位置不同，Thread类中声明sleep，Object类中声明wait。

2.调用的要求不同，sleep可以在任何需要的场景下调用，wait必须使用在同步代码块或者同步方法中

3.关于是否释放同步监视器，如果两个方法都使用在同步代码块或同步方法中，sleep不会释放，wait会释放

4、jdk1.5中，提供了多线程的升级解决方案为：

（1）将同步synchronized替换为显式的Lock操作；

（2）将Object类中的wait(), notify(),notifyAll()替换成了Condition对象，该对象可以通过Lock锁对象获取；

（3）一个Lock对象上可以绑定多个Condition对象，这样实现了本方线程只唤醒对方线程，而jdk1.5之前，一个同步只能有一个锁，不同的同步只能用锁来区分，且锁嵌套时容易死锁。

```
class Resource{
    private String name;
    private int count=1;
    private boolean flag=false;
    private Lock lock = new ReentrantLock();/*Lock是一个接口，ReentrantLock是该
```

```
接口的一个直接子类。*/
        private Condition condition_pro=lock.newCondition(); /*创建代表生产者方面的
Condition对象*/
        private Condition condition_con=lock.newCondition(); /*使用同一个锁，创建代
表消费者方面的Condition对象*/

        public void set(String name){
            lock.lock();//锁住此语句与lock.unlock()之间的代码
            try{
                while(flag)
                    condition_pro.await(); //生产者线程在conndition_pro对象上等待
                this.name=name+"---"+count++;
                System.out.println(Thread.currentThread().getName()+"...生产
者..."+this.name);
                flag=true;
                 condition_con.signalAll();
            }
            finally{
                lock.unlock(); //unlock()要放在finally块中。
            }
        }
        public void out(){
            lock.lock(); //锁住此语句与lock.unlock()之间的代码
            try{
                while(!flag)
                    condition_con.await(); //消费者线程在conndition_con对象上等待
            System.out.println(Thread.currentThread().getName()+"...消费
者..."+this.name);
            flag=false;
            condition_pro.signqlAll(); /*唤醒所有在condition_pro对象下等待的线程，也就是
唤醒所有生产者线程*/
            }
            finally{
                lock.unlock();
            }
        }
    }
```

四、经典例题：生产者/消费者问题：

生产者（Priductor）将产品交给店员（Clerk），而消费者（Customer）从店员处取走产品，店员一次只能持有固定数量的产品（比如20个），如果生产者视图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产：如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。

这里可能出现两个问题：

生产者比消费者快的时候，消费者会漏掉一些数据没有收到。

消费者比生产者快时，消费者会去相同的数据。

2、代码实现

定义缓冲区（店员）

```
/**
 * 线程通信的应用：生产者/消费者问题
 *
 * 1.是否是多线程问题？是的，有生产者线程和消费者线程（多线程的创建，四种方式）
 * 2.多线程问题是否存在共享数据？ 存在共享数据----产品（同步方法，同步代码块，lock锁）
 * 3.多线程是否存在线程安全问题？ 存在----都对共享数据产品进行了操作。（三种方法）
 * 4.是否存在线程间的通信，是，如果生产多了到20时，需要通知停止生产（wait）。（线程之间的通信问题，需要wait，notify等）
 *
 * */


    class Clerk{

        private int productCount = 0;


        //生产产品
        public synchronized void produceProduct() {

            if(productCount<20) {
                productCount++;

                System.out.println(Thread.currentThread().getName()+":开始生产第"+productCount+"个产品");
                notify();
            }else{
                //当有20个时，等待wait
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
        //消费产品
        public synchronized void consumeProduct() {
            if (productCount>0){
                System.out.println(Thread.currentThread().getName()+":开始消费
第"+productCount+"个产品");
                productCount--;
                notify();
            }else{
                //当0个时等待
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

定义生产者

```
    class Producer extends Thread{//生产者线程

        private Clerk clerk;

        public Producer(Clerk clerk) {
            this.clerk = clerk;
        }

        @Override
        public void run() {

            try {
                sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+";开始生产产品.....");

            while(true){
                clerk.produceProduct();
            }
        }
    }
```

定义消费者线程

```
class Consumer implements Runnable{//消费者线程
```

```
    private Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName()+":开始消费产品");

        while(true){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            clerk.consumeProduct();
        }

    }
}
```

测试

```
public class ProductTest {

    public static void main(String[] args){
        Clerk clerk = new Clerk();

        Producer p1 = new Producer(clerk);
        p1.setName("生产者1");

        Consumer c1 = new Consumer(clerk);
        Thread t1 = new Thread(c1);
        t1.setName("消费者1");

        p1.start();
        t1.start();

    }

}
```