线程通信

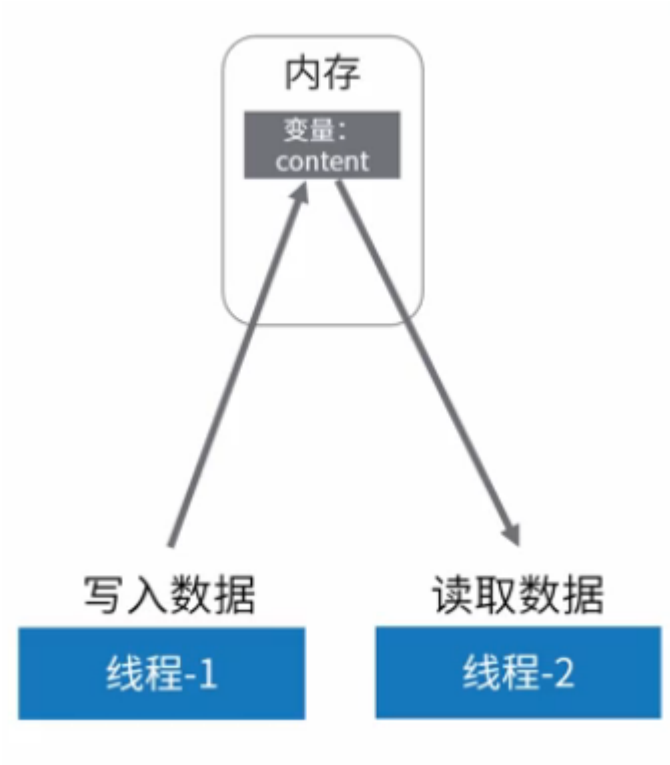
通信的方式

要想实现多个线程之间的协同，如：线程执行先后顺序，获取某个线程执行的结果等等。涉及到线程通信，有以下四种。

1. 文件共享
2. 网络共享
3. 共享变量
4. JDK提供的线程协调API

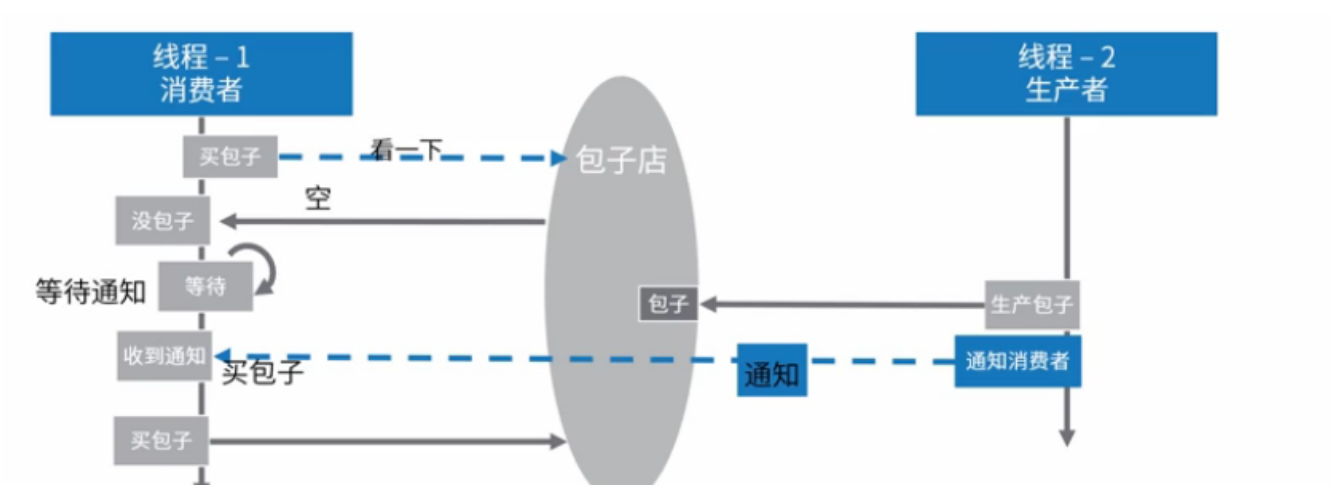
文件共享

共享变量



线程协作-JDK API

JDK中对于需要多线程协作完成某一任务的场景，提供了对应的API支持。多线程协作的典型场景是生产者-消费者模型。



被弃用的suspend和resume

调用suspend挂起目标线程，通过resume可以恢复线程执行。被弃用的主要原因是，容易写出死锁的代码。

```
/** 正常的suspend/resume */
    public void suspendResumeTest() throws Exception {
        // 启动线程
        Thread consumerThread = new Thread(() -> {
            if (baozidian == null) { // 如果没包子，则进入等待
                System.out.println("1、进入等待");
```

```
                Thread.currentThread().suspend();
            }
            System.out.println("2、买到包子，回家");
        });
        consumerThread.start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        consumerThread.resume();
        System.out.println("3、通知消费者");
    }
```

suspend和resume死锁示例

1. 在同步代码中使用

```
/** 死锁的suspend/resume。 suspend并不会像wait一样释放锁，故此容易写出死锁代码 */
    public void suspendResumeDeadLockTest() throws Exception {
        // 启动线程
        Thread consumerThread = new Thread(() -> {
            if (baozidian == null) { // 如果没包子，则进入等待
                System.out.println("1、进入等待");
                // 当前线程拿到锁，然后挂起
                synchronized (this) {
                    Thread.currentThread().suspend();
                }
            }
            System.out.println("2、买到包子，回家");
        });
        consumerThread.start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        // 争取到锁以后，再恢复consumerThread
        synchronized (this) {
            consumerThread.resume();
        }
        System.out.println("3、通知消费者");
    }
```

2. resume比suspend先执行

```
/** 导致程序永久挂起的suspend/resume */
    public void suspendResumeDeadLockTest2() throws Exception {
        // 启动线程
        Thread consumerThread = new Thread(() -> {
            if (baozidian == null) {
                System.out.println("1、没包子，进入等待");
                try { // 为这个线程加上一点延时
                    Thread.sleep(5000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
                // 这里的挂起执行在resume后面
                Thread.currentThread().suspend();
            }
            System.out.println("2、买到包子，回家");
        });
        consumerThread.start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        consumerThread.resume();
        System.out.println("3、通知消费者");
        consumerThread.join();
    }
```

wait/notify机制

这些方法只能由同一对象锁的持有者线程调用，也就是写在同步代码块里面，否则会抛出IllegalMonitorStateException异常。

wiat方法导致当前线程等待，加入该对象的等待集合中，并且放弃当前持有的对象锁。notify/notifyAll方法唤醒一个或所有正在等待这个对象锁的线程。

注意：虽然wait会释放锁，但是对顺序有要求，如果在notify被调用之后，才开始wait方法的调用，线程会永远处于WAITING状态。

1. 正常的wait/notify

```
/** 正常的wait/notify */
    public void waitNotifyTest() throws Exception {
        // 启动线程
        new Thread(() -> {
                synchronized (this) {
                    while (baozidian == null) { // 如果没包子，则进入等待
                    try {
                        System.out.println("1、进入等待");
                        this.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            System.out.println("2、买到包子，回家");
        }).start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        synchronized (this) {
            this.notifyAll();
            System.out.println("3、通知消费者");
        }
    }
```

2. 导致死锁的wait/notify

```
/** 会导致程序永久等待的wait/notify */
    public void waitNotifyDeadLockTest() throws Exception {
        // 启动线程
        new Thread(() -> {
            if (baozidian == null) { // 如果没包子，则进入等待
                try {
                    Thread.sleep(5000L);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                }
                synchronized (this) {
                    try {
                        System.out.println("1、进入等待");
                        this.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            System.out.println("2、买到包子，回家");
        }).start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        synchronized (this) {
            this.notifyAll();
            System.out.println("3、通知消费者");
        }
    }
```

park/unpark机制

线程调用park则等待“许可”，upark方法为指定线程提供“许可”。不要求park和unpark的顺序，但是也不能在同步代码块中调用。多次调用unpark后，在调用park，线程会继续运行。但不会叠加，连续多层次调用park方法，第一次会拿到“许可”直接运行，后续调用会进入等待。

1. 正常的park/unpark

```
/** 正常的park/unpark */
    public void parkUnparkTest() throws Exception {
        // 启动线程
        Thread consumerThread = new Thread(() -> {
            while (baozidian == null) { // 如果没包子，则进入等待
                System.out.println("1、进入等待");
                LockSupport.park();
            }
            System.out.println("2、买到包子，回家");
        });
        consumerThread.start();
        // 3秒之后，生产一个包子
```

```
        Thread.sleep(3000L);
        baozidian = new Object();
        LockSupport.unpark(consumerThread);
        System.out.println("3、通知消费者");
    }
```

2. 死锁的park/unpark

```
/** 死锁的park/unpark */
    public void parkUnparkDeadLockTest() throws Exception {
        // 启动线程
        Thread consumerThread = new Thread(() -> {
            if (baozidian == null) { // 如果没包子，则进入等待
                System.out.println("1、进入等待");
                // 当前线程拿到锁，然后挂起
                synchronized (this) {
                    LockSupport.park();
                }
            }
            System.out.println("2、买到包子，回家");
        });
        consumerThread.start();
        // 3秒之后，生产一个包子
        Thread.sleep(3000L);
        baozidian = new Object();
        // 争取到锁以后，再恢复consumerThread
        synchronized (this) {
            LockSupport.unpark(consumerThread);
        }
        System.out.println("3、通知消费者");
    }
```

伪唤醒

在线程通信中，不能使用if来做唤醒条件的判断，而应该在while循环中做条件判断。原因是处于等待状态的线程可能会受到错误的**警报和伪唤醒**，如果不再循环中检查等待条件，程序就会在没有满足结束条件的情况下退出，伪唤醒是指线程并非因为notify，notifyAll，unpark等api调用而唤醒，是更底层的原因导致的。