拜托，线程间的通信真的很简单。

Simen郎
微信搜索公众号：小郎码知答

1、为什么需要线程通信

线程是操作系统调度的最小单位，有自己的栈空间，可以按照既定的代码逐步的执行，但是如果每个线程间都孤立的运行，那就会造资源浪费。所以在现实中，我们需要这些线程间可以按照指定的规则共同完成一件任务，所以这些线程之间就需要互相协调，这个过程被称为线程的通信。

线程的通信可以被定义为：

> 线程通信就是当多个线程共同操作共享的资源时，互相告知自己的状态以避免资源争夺。

2、线程通信的方式

线程通信主要可以分为三种方式，分别为**共享内存**、**消息传递**和**管道流**。每种方式有不同的方法来实现

- 共享内存：线程之间共享程序的公共状态，线程之间通过读-写内存中的公共状态来隐式通信。

> volatile共享内存

- 消息传递：线程之间没有公共的状态，线程之间必须通过明确的发送信息来显示的进行通信。

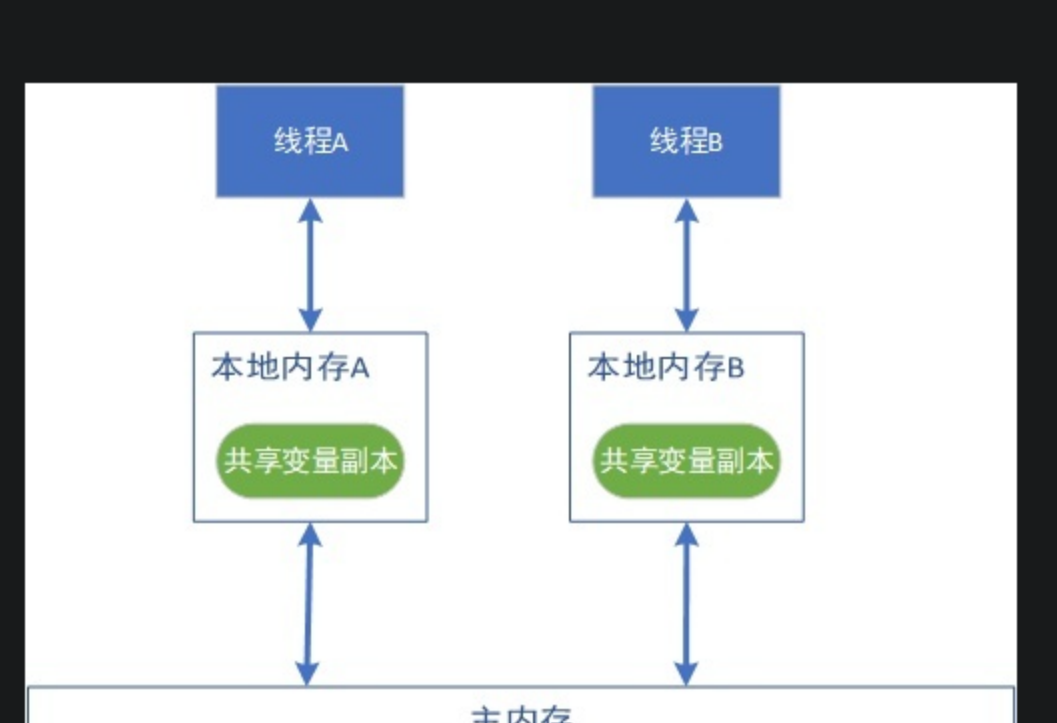
> wait/notify等待通知方式
> join方式

- 管道流

> 管道输入/输出流的形式

2.1共享内存

在学习Volatile之前，我们先了解下Java的内存模型，



在java中，所有堆内存中的所有的数据（实例域、静态域和数组元素）存放在主内存中可以在线程之间共享，一些局部变量、方法中定义的参数存放在本地内存中不会在线程间共享。线程之间的共享变量存储在主内存中，本地内存存储了共享变量的副本。如果线程A要和线程B通信，则需要经过以下步骤

> ①线程A把本地内存A更新过的共享变量刷新到主内存中
> ②线程B到内存中去读取线程A之前已更新过的共享变量。

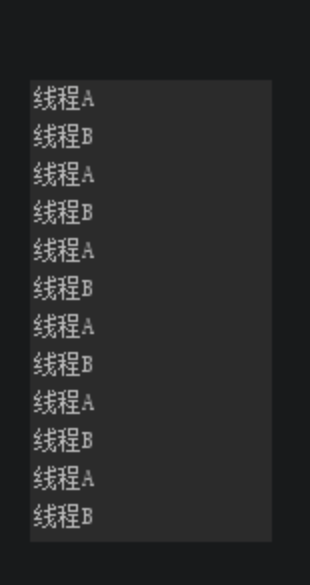
这保证了线程间的通信必须经过主内存。下面引出我们要学习的关键字**volatile**

volatile有一个关键的特性：保证内存可见性，即多个线程访问内存中的同一个被volatile关键字修饰的变量时，当某一个线程修改完该变量后，需要先将这个最新修改的值写回到主内存，从而保证下一个读取该变量的线程取得的就是主内存中该数据的最新值，这样就保证线程之间的透明性，便于线程通信。

代码实现

```java
/**
 * @Author: Simon Lang
 * @Date: 2020/5/5 15:13
 */
public class TestVolatile {
    private static volatile boolean flag=true;
    public static void main(String[] args){
        new Thread(new Runnable() {
            public void run() {
                while (true){
                    if(flag){
                        System.out.println("线程A");
                        flag=false;
                    }
                }
            }
        }).start();


        new Thread(new Runnable() {
            public void run() {
                while (true){
                    if(!flag){
                        System.out.println("线程B");
                        flag=true;
                    }
                }
            }
        }).start();
    }
}
```

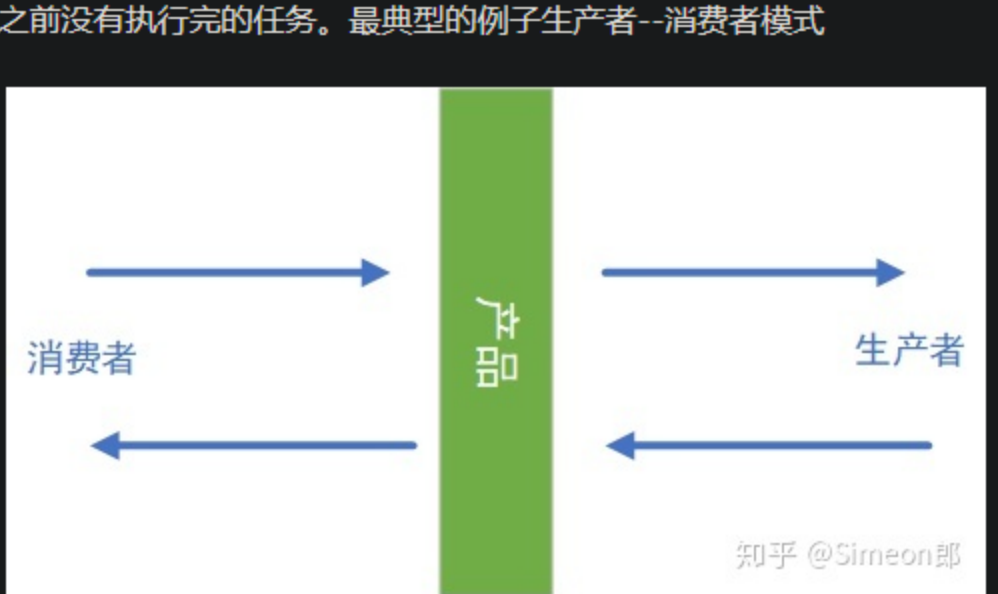
测试结果：线程A和线程B交替执行



2.2消息传递

2.2.1wait/notify等待通知方式

从字面上理解，等待通知机制就是将处于等待状态的线程将由其它线程发出通知后重新获取CPU资源，继续执行之前没有执行完的任务。最典型的例子生产者--消费者模式



有一个产品队列，生产者想要在队列中添加产品，消费者需要从队列中取出产品，如果队列为空，消费者应该等待生产者添加产品后才进行消费，队列为满时，生产者需要等待消费者消费一部分产品后才能继续生产。队列可以认为是java模型里的临界资源，生产者和消费者认为是不同的线程，它们需要交替的占用临界资源来进行各自方法的执行，所以就需要线程间通信。

生产者--消费者模型主要为了方便复用和解耦，java语言实现线程之间的通信协作的方式是等待/通知机制

等待/通知机制提供了三个方法用于线程间的通信

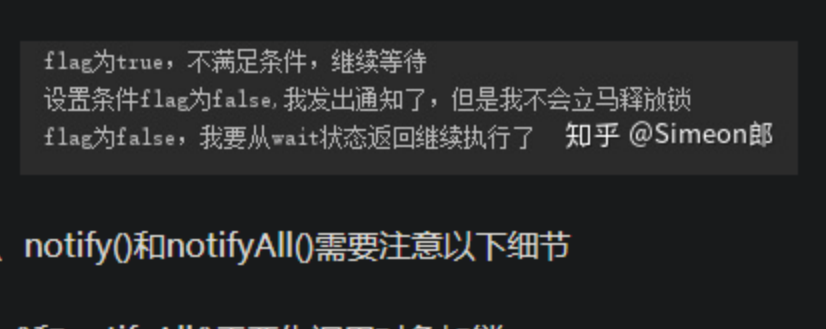
wait()当前线程释放锁并进入等待(阻塞)状态notify()唤醒一个正在等待相应对象锁的线程，使其进入就绪队列，以便在当前线程释放锁后继续竞争锁notifyAll()唤醒所有正在等待相应对象锁的线程，使其进入就绪队列，以便在当前线程释放锁后继续竞争锁

等待/通知机制是指一个线程A调用了对象Object的wait()方法进入等待状态，而另一线程B调用了对象Object的notify()或者notifyAll()方法，当线程A收到通知后就可以从对象Object的wait()方法返回，进而执行后序的操作。线程间的通信需要对象Object来完成，对象中的wait()、notify()、notifyAll()方法就如同开关信号，用来完成等待方和通知方的交互。

测试代码

```java
public class WaitNotify {
    static boolean flag=true;
    static Object lock=new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread waitThread=new Thread(new WaitThread(),"WaitThread");
        waitThread.start();
        TimeUnit.SECONDS.sleep(1);
        Thread notifyThread=new Thread(new NotifyThread(),"NotifyThread");
        notifyThread.start();
    }
    //等待线程
    static class WaitThread implements Runnable{
        public void run() {
            //加锁
            synchronized (lock){
                //条件不满足时，继续等待，同时释放lock锁
                while (flag){
                    System.out.println("flag为true，不满足条件，继续等待");
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //条件满足
                System.out.println("flag为false，我要从wait状态返回继续执行了");

            }

        }
    }
    //通知线程
    static class NotifyThread implements Runnable{

        public void run() {
            //加锁
            synchronized (lock){
                //获取lock锁，然后进行通知，但不会立即释放lock锁，需要该线程执行完毕
                lock.notifyAll();
                System.out.println("设置flag为false,我发出通知了，但是我不会立马释放锁");
                flag=false;
            }
        }
    }
}
```

测试结果



NOTE：使用wait()、notify()和notifyAll()需要注意以下细节

- 使用wait()、notify()和notifyAll()需要先调用对象加锁
- 调用wait()方法后，线程状态由Running变成Waiting，并将当前线程放置到对象的等待队列
- notify()和notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()和notifyAll()的线程释放锁之后等待线程才有机会从wait()返回
- notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll()方法则是将等待队列中所有的线程全部转移到同步队列，被移到的线程状态由Waiting变为Blocked。
- 从wait()方法返回的前提是获得调用对象的锁

其实等待通知机制有有一个经典的范式，该范式可以分为两部分，分别是等待方（消费者）和通知方（生产者）

- 等待方

```
synchronized(对象){
while(条件不满足){
对象.wait()
}
对应的处理逻辑
}
```

- 通知方

```
synchronized(对象){
改变条件
对象.notifyAll
}
```

2.2.2join方式

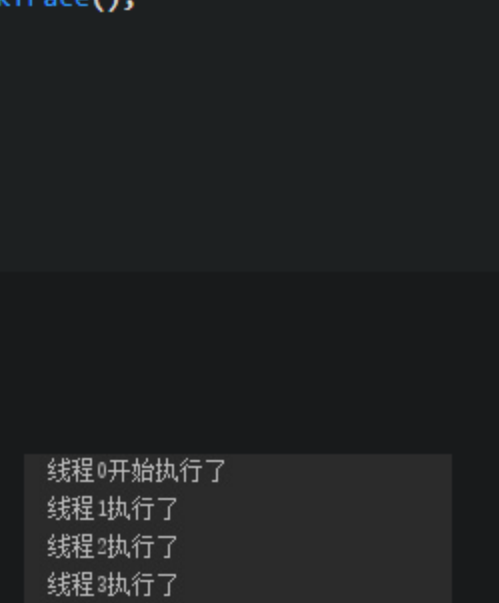
在很多应用场景中存在这样一种情况，主线程创建并启动子线程后，如果子线程要进行很耗时的计算，那么主线程将比子线程先结束，但是主线程需要子线程的计算的结果来进行自己下一步的计算，这时主线程就需要等待子线程，java中提供可join()方法解决这个问题。

join()方法的作用是：在当前线程A调用线程B的join()方法后，会让当前线程A阻塞，直到线程B的逻辑执行完成，A线程才会解除阻塞，然后继续执行自己的业务逻辑，这样做可以节省计算机中资源。

测试代码

```java
public class TestJoin {
    public static void main(String[] args){
        Thread thread=new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("线程0开始执行了");
            }
        });
        thread.start();
        for (int i=0;i<10;i++){
            JoinThread jt=new JoinThread(thread,i);
            jt.start();
            thread=jt;
        }

    }

    static class JoinThread extends Thread{
        private Thread thread;
        private int i;

        public JoinThread(Thread thread,int i){
            this.thread=thread;
            this.i=i;
        }

        @Override
        public void run() {
            try {
                thread.join();
                System.out.println("线程"+(i+1)+"执行了");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

测试结果



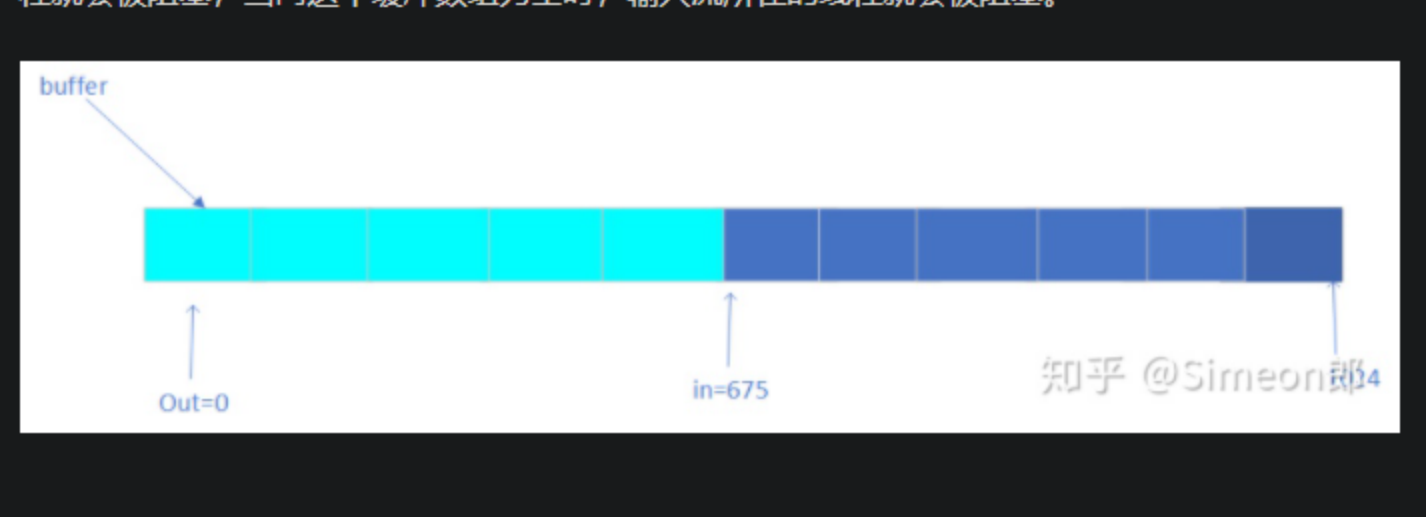
NOTE：每个线程的终止的前提是前驱线程的终止，每个线程等待前驱线程终止后，才从join方法返回，实际上，这里涉及了等待/通知机制，即下一个线程的执行需要接受前驱线程结束的通知。

2.3管道输入/输出流

管道流是是一种使用比较少的线程间通信方式，管道输入/输出流和普通文件输入/输出流或者网络输出/输出流不同之处在于，它主要用于线程之间的数据传输，传输的媒介为管道。

管道输入/输出流主要包括4种具体的实现：PipedOutputStrean、PipedInputStrean、PipedReader和PipedWriter，前两种面向字节，后两种面向字符。

java的管道的输入和输出实际上使用的是一个循环缓冲数组来实现的，默认为1024，输入流从这个数组中读取数据，输出流从这个数组中写入数据，当这个缓冲数组已满的时候，输出流所在的线程就会被阻塞，当向这个缓冲数组为空时，输入流所在的线程就会被阻塞。

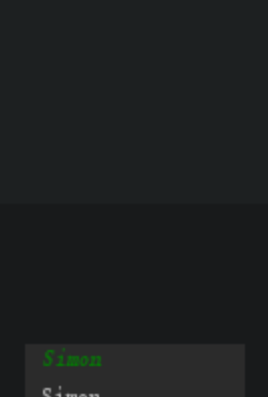


> buffer：缓冲数组，默认为1024
> out：从缓冲数组中读数据
> in：从缓冲数组中写数据

测试代码

```java
public class TestPip {
    public static void main(String[] args) throws IOException {
        PipedWriter writer  = new PipedWriter();
        PipedReader reader = new PipedReader();
        //使用connect方法将输入流和输出流连接起来
        writer.connect(reader);
        Thread printThread = new Thread(new Print(reader) , "PrintThread");
        //启动线程printThread
        printThread.start();
        int receive = 0;
        try{
            //读取输入的内容
            while((receive = System.in.read()) != -1){
                writer.write(receive);
            }
        }finally {
            writer.close();
        }
    }

    private static class Print implements Runnable {
        private PipedReader reader;

        public Print(PipedReader reader) {
            this.reader = reader;
        }

        @Override
        public void run() {
            int receive = 0;
            try{
                while ((receive = reader.read()) != -1){
                    //字符转换
                    System.out.print((char) receive);
                }
            }catch (IOException e) {
                System.out.print(e);
            }
        }
    }
}
```

测试结果



NOTE：对于Piped类型的流，必须先进性绑定，也就是调用connect()方法，如果没有将输入/输出流绑定起来，对于该流的访问将抛出异常。

如果有什么疑问可以联系我，（公众号有我的联系方式）让我们每天一起每天进步一点点

参考文献

[1]方腾飞.Java并发编程的艺术

[2]voidcn.com/article/p-hw...

[3]blog.csdn.net/canot/art...

编辑于 2020-05-07

线程 多线程 并发