线程间通信方式：

1.同步

这里的同步是指多个线程通过synchronized关键字这种方式来实现线程间的通信。

```
public class MyObject {

    synchronized public void methodA() {
        //do something....
    }

    synchronized public void methodB() {
        //do some other thing
    }
}

public class ThreadA extends Thread {

    private MyObject object;
//省略构造方法
    @Override
    public void run() {
        super.run();
        object.methodA();
    }
}

public class ThreadB extends Thread {

    private MyObject object;
//省略构造方法
    @Override
    public void run() {
        super.run();
        object.methodB();
    }
}

public class Run {
    public static void main(String[] args) {
        MyObject object = new MyObject();

        //线程A与线程B 持有的是同一个对象:object
        ThreadA a = new ThreadA(object);
        ThreadB b = new ThreadB(object);
        a.start();
        b.start();
    }
}
```

由于线程A和线程B持有同一个MyObject类的对象object，尽管这两个线程需要调用不同的方法，但是它们是同步执行的。比如线程B需要等待线程A执行完了methodA()方法之后，它才能执行methodB()方法。这样，线程A和线程B就实现了通信。

这种方式，本质就是“共享内存”方式的通信。多个线程需要访问同一个共享变量，谁拿到了锁，谁就可以执行。

2.while轮询方式。

```
public class MyList {

    private List<String> list = new ArrayList<String>();
    public void add() {
        list.add("elements");
    }
    public int size() {
        return list.size();
    }
}

import mylist.MyList;

public class ThreadA extends Thread {

    private MyList list;

    public ThreadA(MyList list) {
        super();
        this.list = list;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                list.add();
                System.out.println("添加了" + (i + 1) + "个元素");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

import mylist.MyList;

public class ThreadB extends Thread {

    private MyList list;

    public ThreadB(MyList list) {
        super();
        this.list = list;
```

```
49      }
50
51      @Override
52      public void run() {
53          try {
54              while (true) {
55                  if (list.size() == 5) {
56                      System.out.println("==5, 线程b准备退出了");
57                      throw new InterruptedException();
58                  }
59              }
60          } catch (InterruptedException e) {
61              e.printStackTrace();
62          }
63      }
64  }
65
66  import mylist.MyList;
67  import extthread.ThreadA;
68  import extthread.ThreadB;
69
70  public class Test {
71
72      public static void main(String[] args) {
73          MyList service = new MyList();
74
75          ThreadA a = new ThreadA(service);
76          a.setName("A");
77          a.start();
78
79          ThreadB b = new ThreadB(service);
80          b.setName("B");
81          b.start();
82      }
83  }
```

在这种方式下，线程A不断地改变条件，线程ThreadB不停的通过while语句检测(list.size == 5)是否成立，从而实现了线程间的通信，但是这种方式会浪费cpu资源。一直会在轮询。还有就是轮询还不一定可见，volatile关键字的可见性，线程都是把变量读取到本地线程栈空间，然后再去修改的本地变量。因此，如果线程B每次都在取本地的条件变量，尽管另外一个线程已经改变了轮询的条件，它也察觉不到，这样也造成了死循环。

```
 4 public class MyList {
 5
 6     private static List<String> list = new ArrayList<String>();
 7
 8     public static void add() {
 9         list.add("anyString");
10     }
11
12     public static int size() {
13         return list.size();
14     }
15 }
16
17
18 public class ThreadA extends Thread {
19
20     private Object lock;
21
22     public ThreadA(Object lock) {
23         super();
24         this.lock = lock;
25     }
26
27     @Override
28     public void run() {
29         try {
30             synchronized (lock) {
31                 if (MyList.size() != 5) {
32                     System.out.println("wait begin "
33                             + System.currentTimeMillis());
34                     lock.wait();
35                     System.out.println("wait end  "
36                             + System.currentTimeMillis());
37                 }
38             }
39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         }
42     }
43 }
44
45
46 public class ThreadB extends Thread {
47     private Object lock;
48
49     public ThreadB(Object lock) {
50         super();
51         this.lock = lock;
52     }
```

```
    @Override
    public void run() {
        try {
            synchronized (lock) {
                for (int i = 0; i < 10; i++) {
                    MyList.add();
                    if (MyList.size() == 5) {
                        lock.notify();
                        System.out.println("已经发出了通知");
                    }
                    System.out.println("添加了" + (i + 1) + "个元素!");
                    Thread.sleep(1000);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Run {

    public static void main(String[] args) {

        try {
            Object lock = new Object();

            ThreadA a = new ThreadA(lock);
            a.start();

            Thread.sleep(50);

            ThreadB b = new ThreadB(lock);
            b.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

当A的条件没有满足时，线程A会调用wait()放弃CPU，并进入阻塞状态。--不像while轮询那样占用CPU。

当条件满足时，线程B调用notify()通知线程A，所谓通知线程A，就是唤醒线程A，并让它进入可运行状态。这种方式的好处就是CPU利用率提高了。

但是也有一些缺点：比如，线程B先执行，一下子添加了5个元素并调用notify()发送了通知，而此时线程A还执行，当线程A执行调用wait时，那么永远不会唤醒了。因为线程B已经发了通知了，以后不再发通知。说明:通知过早，会打乱程序的执行逻辑。

```
wait begin
添加了1个元素!
添加了2个元素!
添加了3个元素!
添加了4个元素!
已经发出了通知
添加了5个元素!
添加了6个元素!
添加了7个元素!
添加了8个元素!
添加了9个元素!
添加了10个元素!
wait end
```

这个结果就体现了notify()执行后不会立即释放CPU资源，而是等到同步块执行完成才释放。