第 3 章

多线程通信

前面探讨了多线程的基础知识、多线程之间如何安全地访问共享资源等。这一章着重探讨一下多线程之间是如何交互的。

3.1 wait()与 notify()

java.lang.Object 类中内置了用于线程通信的方法 wait()、notify()与 notifyAll()。

```
public class Object {
    public final void wait() throws InterruptedException {}
    public final native void wait(long timeout)
                throws InterruptedException;
    public final native void notify();
    public final native void notifyAll();
}
```

调用 Object 对象的 wait()方法，会导致当前线程进入 WAITING 等待状态，直到另外一个线程，调用该对象的 notify()或 notifyAll()方法，处于 WAITING 状态的线程才重新转为 RUNNABLE 状态。

调用对象的 wait()方法前，当前线程必须要拥有指定对象的监视器锁。调用 wait()方法后，会释放对象的监视器锁，然后当前线程进入 WAITING 状态。

```
synchronized(obj) {
   obj.wait();
}
```

3.1.1 阻塞当前线程

参见如下代码，在主函数中启动一个线程，当 i=5 的时候，调用 object 对象的 wait()方法，这会导致当前线程 8 被阻塞（不是主线程被阻塞）。

```
public static void main(String[] args) {
```

```
    Object object = new Object();
    new Thread(new Runnable() {
        public void run() {
           for(int i=0;i<10;i++) {
              System.out.println(Thread.currentThread().getId() + ",i=" +i);
              if(i==5) {
                 synchronized(object) {
                     try {
                        System.out.println(Thread.currentThread().getId()
                                          + "开始等待...");
                        object.wait();
                     } catch (Exception e) {
                        e.printStackTrace();
                     }
                }
              }
           }
        }
    }).start();
}
```

程序运行结果如下，线程 8 被阻塞后，进入长期等待状态。

```
8,i=0
8,i=1
8,i=2
8,i=3
8,i=4
8,i=5
8 开始等待...
```

在主函数中，增加一个新的线程，延迟 1 秒后，发送 notify()通知。

```
new Thread(new Runnable() {
    public void run() {
        System.out.println(Thread.currentThread().getId() + " running...");
        try {
          Thread.sleep(1000);
           synchronized(object) {
               System.out.println(Thread.currentThread().getId()
                                  + ", 发送 notify 通知...");
               object.notify();
           }
        } catch (Exception e) {}
```

```
        }
    }).start();
```

重新运行程序，运行结果如下。线程 8 进入 WAITING 状态后，线程 9 延迟 1 秒后发出了 notify()通知。线程 8 接到通知后继续运行直至结束。

```
8,i=0
8,i=1
8,i=2
8,i=3
8,i=4
8,i=5
8 开始等待...
9 running...
9，发送 notify 通知...
8,i=6
8,i=7
8,i=8
8,i=9
```

注意，不管是调用 object 的 wait()方法，还是调用 object 的 notify()方法，都要提前获得 object 对象的监视器锁：

```
synchronized(object) {
    ...
}
```

3.1.2 案例分析：厨师与侍者 1

假设有一个小饭店，里面只有一个厨师和一个侍者（服务员）。厨师只有收到服务员的通知才开始做菜，没有工作时厨师就处于等待状态。服务员只有收到顾客的订单，才会通知厨师工作，没有订单时，服务员也处于等待状态。厨师做完菜，会通知服务员取餐。使用多线程，模拟这个小饭店的工作状况。

厨师什么时候做菜，应该做什么菜，由服务员根据订单通知厨师。而服务员什么时候可以取菜，则是看厨师什么时候做完菜，做完后才会通知服务员取菜。

操作步骤如下：

（1）新建订单类，模拟最多 10 个订单。

```
class Order{
    private static int i=0;
    private int m_count;
    public Order(){
```

```
        m_count = i++;
        if(m_count==10){
            System.out.println("没有食物了，结束!");
            System.exit(0);
        }
    }
}
```

（2）新建饭店类，成员变量订单表示饭店当前是否存在订单。

```
class Restaurant {
    public Order order;
}
```

（3）新建厨师类 Chef，Chef 启动后，就处于等待 waiter 通知的状态。饭店当前订单为空时，模拟生成新订单。

```
class Chef extends Thread{
   private Restaurant restaurant;
   private Waiter waiter;
   public Chef(Restaurant restaurant,Waiter waiter){
       this.restaurant = restaurant;
       this.waiter = waiter;
   }
   public void run(){
      while(true){
         if(restaurant.order == null){
            restaurant.order = new Order();
            System.out.println("厨师-" + Thread.currentThread().getId()
                               + ",接到新订单");
            synchronized(waiter){
               System.out.println("厨师-" + Thread.currentThread().getId()
                                  + ",通知 waiter 取食物");
               waiter.notify();
            }
            try {
               Thread.sleep(1000);
            } catch (Exception e) {
            }
         }

} } }
```

（4）新建侍者类 Waiter。厨师做完菜后，通知侍者取餐。

```
class Waiter extends Thread{
    private Restaurant restaurant;
    public Waiter(Restaurant r){
        restaurant = r;
    }
    public void run() {
        while(restaurant.order == null){
            synchronized(this){
                try {
                System.out.println("Waiter-" + Thread.currentThread().getId()
                                + ",等待中");
                wait();
                restaurant.order = null;
                System.out.println("Waiter-" + Thread.currentThread().getId()
                            + ",收到通知，取走订单 ");
            } catch (Exception e) {    }
        }
} } }
```

（5）代码测试。

```
public static void main(String[] args) {
    Restaurant restaurant = new Restaurant();
    Waiter waiter = new Waiter(restaurant);
    waiter.start();
    Chef chef = new Chef(restaurant,waiter);
    chef.start();
}
```

程序运行结果如下：

```
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
```

```
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
厨师-9,接到新订单
厨师-9,通知 waiter 取食物
Waiter-8,收到通知，取走订单
Waiter-8,等待中
没有食物了，结束!
```

3.1.3　案例分析：厨师与侍者 2

3.1.2 节的厨师与侍者案例引自 *Thinking in Java*，都是在厨师类中模拟创建的订单，然后通知服务员（侍者）取菜。这个流程过于简单了，与实际情况不符。本节在 3.1.2 节的基础上，做进一步优化。模拟顾客在饭店通过服务员点菜，然后服务员通知厨师做菜；厨师做菜完成后会通知服务员取菜；厨师与服务员无事时，则处于空闲等待状态。

实现步骤如下：

（1）定义订单类，设置订单编号和订单内容。

```
public class Order implements Serializable{
  private String dno;
  private String info;
```

```
    public String getDno() {
        return dno;
    }
    public String getInfo() {
        return info;
    }
    public void setInfo(String info) {
        this.info = info;
    }
    public Order(String dno) {
        this.dno = dno;
    }
}
```

（2）定义饭店类，模拟顾客进入饭店，随机生成订单。

```
class Restaurant implements Runnable {
    private Waiter waiter;
    public void setWaiter(Waiter waiter) {
        this.waiter = waiter;
    }
    public void run() {
        while(true) {
            int rand = (int)(Math.random()*5000);
            try {
                System.out.println("饭店等待顾客中---------------");
                Thread.sleep(rand);
                String dno = "d" + System.currentTimeMillis();
                Order order = new Order(dno);
                order.setInfo("宫保鸡丁一份...");
                System.out.println("顾客来了，通知服务员点菜，生成订单：" + dno );
                synchronized (waiter) {
                    waiter.setOrder(order);  //把订单给服务员
                    waiter.setMsgID(1);
                    waiter.notify();
                }

            } catch (Exception e) {
            }
        }
    }
}
```

（3）创建 Waiter 类，它可以接收新订单通知，也可以接收取菜通知。

```
class Waiter implements Runnable{
    private Order order;
    private Chef chef;
    private int msgID = 1; //如果是新订单通知 ID=1, 如果是取菜通知 ID=2
    public void setMsgID(int msgID) {
        this.msgID = msgID;
    }
    public void setChef(Chef chef) {
        this.chef = chef;
    }
    public void setOrder(Order order) {
        this.order = order;
    }
    public void run() {
        while(true) {
            synchronized(this) {
                try {
                    System.out.println("服务员空闲等待中...");
                    this.wait();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            if(msgID == 1) {
                //服务员收到新订单通知
                System.out.println("waiter 收到订单：" + this.order.getDno()
                                    + "," + this.order.getInfo());
                //通知厨师做菜
                synchronized(chef) {
                    System.out.println("waiter 通知厨师做菜...");
                    chef.setOrder(order);
                    chef.notify();
                }
            }else {
                //服务员收到了取菜通知
                System.out.println("waiter 取菜给顾客....");
            }
        }   }   }
```

（4）定义厨师类。厨师收到服务员的做菜通知后开始烹饪，菜做完后通知服务员取菜。

```
class Chef implements Runnable{
    private Order order;
```

```
    private Waiter waiter;
    public void setOrder(Order order) {
        this.order = order;
    }
    public void setWaiter(Waiter waiter) {
        this.waiter = waiter;
    }
    public void run() {
        while(true) {
            synchronized(this) {
                try {
                    System.out.println("厨师空闲等待中...");
                    this.wait();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            //厨师收到订单通知
            int rand = (int)(Math.random()*800);
            try {
                Thread.sleep(rand);
            } catch (Exception e) {
            }
            System.out.println("厨师做菜完成,通知 waiter 取菜...");
            synchronized(this.waiter) {
                waiter.setMsgID(2);
                waiter.notify();
            }
     }   }   }
```

（5）在主函数中，模拟饭店的点菜流程，进行测试。

```
public static void main(String[] args) {
    Chef chef = new Chef();
    Waiter waiter = new Waiter();
    waiter.setChef(chef);
    chef.setWaiter(waiter);
    Restaurant rest = new Restaurant();
    rest.setWaiter(waiter);
    new Thread(waiter).start();
    new Thread(chef).start();
    new Thread(rest).start();
}
```

程序运行结果如下，本节案例与真实项目环境更加贴近。

```
厨师空闲等待中...
服务员空闲等待中...
饭店等待顾客中---------------
顾客来了，通知服务员点菜，生成订单：d1597404292617
waiter 收到订单：d1597404292617,宫保鸡丁一份...
waiter 通知厨师做菜...
服务员空闲等待中...
饭店等待顾客中---------------
厨师做菜完成,通知 waiter 取菜...
厨师空闲等待中...
waiter 取菜给顾客...
服务员空闲等待中...
顾客来了，通知服务员点菜，生成订单：d1597404293698
饭店等待顾客中---------------
waiter 收到订单：d1597404293698,宫保鸡丁一份...
waiter 通知厨师做菜...
服务员空闲等待中...
厨师做菜完成,通知 waiter 取菜...
厨师空闲等待中...
```

3.1.4　案例分析：两个线程交替输出信息

本节案例是让两个线程交替输出信息：一个线程输出 aa，另一个线程输出 bb。

这个案例的关键在于共享对象的状态为开关项，当某个线程锁定对象并读取对象状态时，另一个线程必须等待。

操作步骤如下：

（1）新建状态类 State，设置布尔值 bRet 作为开关项。

```
class State {
     public boolean bRet = false;
}
```

（2）新建任务类 PrintA，锁定状态对象，并判断开关项为真时进入等待状态。

```
class PrintA implements Runnable {
private State state;
public PrintA(State state){
     this.state = state;
}
public void run(){
      while (true) {
```

```
            try {
                 synchronized (state) {
                     if(state.bRet){
                         state.wait();
                     }
                     System.out.println("aa...");
                     Thread.sleep(1000);
                     state.bRet = true;
                     state.notify();
                 }
            }catch(Exception e){
                  e.printStackTrace();
            }
}}}
```

（3）新建任务类 PrintB，锁定状态对象，并判断开关项为假时进入等待状态。

```
class PrintB implements Runnable {
    private State state;
    public PrintB(State state){
         this.state = state;
    }
         public void run(){
             while (true) {
                 try {
                      synchronized(state) {
                          if(!state.bRet){
                             state.wait();
                          }
                          System.out.println("bb...");
                          Thread.sleep(1000);
                          state.bRet = false;
                          state.notify();
                      }
                 }catch(Exception e){
                       e.printStackTrace();
                 }
    } } }
```

（4）代码测试。

```
public static void main(String[] args) {
    State state = new State();
    new Thread(new PrintA(state)).start();
```

```
        new Thread(new PrintB(state)).start();
    }
```

测试结果如下，交替输出 aa...和 bb...，循环往复。

```
aa...
bb...
aa...
bb...
aa...
```

3.2　join 线程排队

线程 A 调用线程 B 对象的 join()方法，会导致线程 A 的运行中断，直到线程 B 运行完毕或超时，线程 A 才继续运行。

```
public class Thread {
    public final void join()
            throws InterruptedException { }
    public final synchronized void join(long millis)
            throws InterruptedException {}
}
```

需要注意的是，join(long millis)方法中传入的超时参数不能为负数，否则将抛出 IllegalArgumentException 异常。如果 millis 参数为 0，则表示永远等待。

3.2.1　加入者与休眠者

分别定义加入者 Joiner 与休眠者 Sleeper。在 Sleeper 运行过程中，如果 Joiner 加入 Sleeper 的运行中，会导致 Sleeper 的运行被阻塞，直到 Joiner 运行完毕，Sleeper 才会继续运行。

开发步骤如下：

（1）新建类 Joiner，循环输出 10 个 k 值。

```
class Joiner extends Thread{
   public void run() {
      System.out.println("Joiner 线程 id="
                         + Thread.currentThread().getId()+  " run...");
      try {
         for(int i=0;i<10;i++){
            Thread.sleep(100);
            System.out.println("线程"
                       + Thread.currentThread().getId() + "---k=" + i);
```

```
            }
        } catch (Exception e) {
        }
        System.out.println(Thread.currentThread().getId()+  "  end...");
    }
}
```

（2）新建类 Sleeper，循环输出 10 个 i 值。当 i=5 时，joiner 加入。

```
class Sleeper extends Thread{
    private Joiner joiner;
    public void setJoiner(Joiner joiner) {
        this.joiner = joiner;
    }
    public void run() {
        System.out.println("Sleeper 线程 id="
                + Thread.currentThread().getId() +  " run...");
        try {
            for(int i=0;i<10;i++){
                if(i==5 && joiner != null) {
                    System.out.println("joiner 加入,线程" +
                                    Thread.currentThread().getId() + "被阻塞");
                    joiner.join();
                }
                Thread.sleep(100);
                System.out.println("线程"
                            + Thread.currentThread().getId() + "---i=" + i);
            }
        } catch (Exception e) {    }
        System.out.println(Thread.currentThread().getId()+  " end...");
    }
}
```

（3）代码测试，同时启动 Joiner 与 Sleeper 两个线程。当 i=5 时，Joiner 加入 Sleeper，这将导致 Sleeper 的运行被阻塞。

```
public static void main(String[] args) {
    Sleeper sleeper = new Sleeper();
    Joiner joiner = new Joiner();
    sleeper.setJoiner(joiner);
    sleeper.start();
    joiner.start();
}
```

代码运行结果如下，在 Joiner 加入前，Joiner 与 Sleeper 两个线程同时运行。当 Joiner 加入后，Sleeper 被阻塞。直到 Joiner 运行结束，Sleeper 才重新被激活，并运行到结束。

```
Sleeper 线程 id=8 run...
Joiner 线程 id=9 run...
线程 8---i=0
线程 9---k=0
线程 8---i=1
线程 9---k=1
线程 8---i=2
线程 9---k=2
线程 8---i=3
线程 9---k=3
线程 8---i=4
joiner 加入,线程 8 被阻塞
线程 9---k=4
线程 9---k=5
线程 9---k=6
线程 9---k=7
线程 9---k=8
线程 9---k=9
9  end...
线程 8---i=5
线程 8---i=6
线程 8---i=7
线程 8---i=8
线程 8---i=9
8 end...
```

3.2.2　案例：紧急任务处理

在工作的过程中经常会遇到这种情况：当你正在处理日常工作时，突然之间来了紧急任务。这时你被迫停下手里的工作优先处理紧急任务，等到紧急任务处理完成后再继续刚才未完成的工作。

操作步骤如下：

（1）新建线程 Worker 代表日常任务的完成。当出现紧急任务时，日常任务被停止，直到紧急任务结束才继续。

```
class Worker extends Thread {
    private UrgentTask joiner;
    public void run() {
        int i = 0;
```

```
            while(i < 9){
                try {
                    if(i == 8){
                        UrgentTask urgent = joiner;
                        System.out.println("突然接到了紧急工作,需要去完成...");
                        urgent.start();
                        urgent.join();
                    }
                    System.out.println("我正在做日常工作,完成度"
                                        + (i * 10) + "%...");
                    TimeUnit.SECONDS.sleep(1);
                    i += 2;
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("我已经完成了日常工作,完成度 100%...");
        }
        public void setJoiner(UrgentTask joiner) {
              this.joiner = joiner;
        }
    }
```

（2）新建类 UrgentTask，表示紧急任务。

```
    class UrgentTask extends Thread {
        public void run() {
            int i = 0;
            while(i < 9){
                try{
                    System.out.println("紧急任务处理,完成度" + (i * 10) + "%+++");
                    TimeUnit.SECONDS.sleep(1);
                    i += 3;
                }catch(Exception e){
                    e.printStackTrace();
                }
            }
            System.out.println("紧急工作完成度 100%+++");
        }
    }
```

（3）分别创建日常工作对象和紧急任务对象。当日常工作进行到一半时，紧急任务加入。

```
public static void main(String[] args) {
      try {
            Worker worker = new Worker();
            UrgentTask j = new UrgentTask();
            worker.start();
            TimeUnit.SECONDS.sleep(3);
            worker.setJoiner(j);
      } catch (Exception e) {
            e.printStackTrace();
      }
}
```

程序运行结果如下：

```
我正在做日常工作,完成度 0%...
我正在做日常工作,完成度 20%...
我正在做日常工作,完成度 40%...
我正在做日常工作,完成度 60%...
突然接到了紧急工作,需要去完成...
紧急任务处理,完成度 0%+++
紧急任务处理,完成度 30%+++
紧急任务处理,完成度 60%+++
紧急工作完成度 100%+++
我正在做日常工作,完成度 80%...
我已经完成了日常工作,完成度 100%...
```

3.2.3　join 限时阻塞

join(long millis)就是指定阻塞时间，超时自动解锁。

修改 3.2.1 节中 Sleeper 的代码，原来的加入代码为 joiner.join()，现在修改为 joiner.join(300)，即 Joiner 只阻塞 Sleeper 线程 300 ms。

其他代码不变，测试结果如下。从测试结果可以看出，Sleeper 被阻塞 300 ms 后继续运行，无须等到 Joiner 运行完毕。

```
Sleeper 线程 id=8 run...
Joiner 线程 id=9 run...
线程 8---i=0
线程 9---k=0
线程 8---i=1
线程 9---k=1
线程 8---i=2
线程 9---k=2
线程 8---i=3
```

```
线程 9---k=3
线程 8---i=4
joiner 加入,线程 8 被阻塞
线程 9---k=4
线程 9---k=5
线程 9---k=6
线程 9---k=7
线程 8---i=5
线程 9---k=8
线程 8---i=6
线程 9---k=9
9  end...
线程 8---i=7
线程 8---i=8
线程 8---i=9
8 end...
```

参考 join(long millis)方法的源码如下，join()方法的底层实现依赖的是 wait()方法。执行新线程 join()后，被阻塞的线程进入 WAITING 等待状态。join()延时结束后，通过 notify()通知，会唤醒处于 WAITING 状态的休眠线程。

```
public final synchronized void join(long millis)
throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;
    if (millis < 0) {
        throw new IllegalArgumentException("超时值为负");
    }
    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

3.3　线程中断

线程类 Thread 中的停止线程的 stop()方法与暂停线程的 suspend()方法已被废弃，如果想停止一个正在运行中的线程，可以尝试使用 interrupt()方法。

interrupt()方法并不能马上停止线程的运行，它只是给线程设置一个中断状态值，这相当于一个停止线程运行的建议，线程是否能够停止，由操作系统和 CPU 决定。

isInterrupted()方法用于判断当前线程是否处于中断状态。

```
public class Thread implements Runnable {
     public final void stop() {}
     public final void suspend() {}
     public void interrupt() {}
     public boolean isInterrupted() {}
}
```

3.3.1　中断运行态线程

启动一个线程，延迟 500 ms 后，调用这个线程对象的 interrupt()方法。观察这个线程的中断状态，查看这个线程是否能够停止。

```
public static void main(String[] args) {
   Thread t = new Thread(new Runnable() {
      public void run() {
         for(int i=0;i<10;i++) {
            if(Thread.currentThread().isInterrupted()) {
               System.out.println("收到中断通知，结束线程...");
               break;
            }else {
               System.out.println(Thread.currentThread().getId()
                                  + ",i=" + i);
               try {
                    Thread.sleep(100);
               } catch (Exception e) {   }
            }
         }
      }
   });
   t.start();
   try {
      Thread.sleep(500);
   } catch (Exception e) {   }
```

```
        t.interrupt();
        System.out.println(t.getId() + "中断状态:" + t.isInterrupted());
    }
```

反复运行上面的代码，可能得到完全不同的三种结果，分别如下：

（1）调用线程的 interrupt()方法后，最容易出现的结果就是中断状态为 false，也就是说 interrupt()方法没有起到任何效果。

```
8,i=0
8,i=1
8,i=2
8,i=3
8,i=4
8 中断状态:false
8,i=5
8,i=6
8,i=7
8,i=8
8,i=9
```

（2）调用线程的 interrupt()方法后，线程中断状态可能被设置为 true。但是线程仍然继续运行，并没有被停止。

```
8,i=0
8,i=1
8,i=2
8,i=3
8,i=4
8 中断状态:true
8,i=5
8,i=6
8,i=7
8,i=8
8,i=9
```

（3）调用线程的 interrupt()方法后，线程真的停止运行了。这种情况出现的次数较少，需要反复尝试才可以看到。

```
8,i=0
8,i=1
8,i=2
8,i=3
8,i=4
8 中断状态:true
```

```
收到中断通知，结束线程...
```

3.3.2　中断阻塞态线程

修改 3.3.1 节中的代码，让线程启动后进入 WAITING 状态。这时调用线程的 interrupt()方法，观察程序运行状态。

```
public static void main(String[] args) {
    Object obj = new Object();
    Thread t = new Thread(new Runnable() {
        public void run() {
           for(int i=0;i<10;i++) {
              if(Thread.currentThread().isInterrupted()) {
                 System.out.println("收到中断通知，结束线程...");
                 break;
              }else {
                 System.out.println(Thread.currentThread().getId()
                                    + ",i=" + i);
                 try {
                 Thread.sleep(100);
                 synchronized(obj) {
                      obj.wait();
                 }
                 } catch (Exception e) {
                     e.printStackTrace();
                 }
              }

           }
       }
    });
    t.start();
}
```

没有调用线程对象的 interrupt()方法时，程序启动后进入 WAITING 状态。输出结果如下所示：

```
8,i=0
8 中断状态:false
```

调用线程对象的 interrupt()方法，程序运行结果发生变化。

```
public static void main(String[] args) {
      ...
```