

day17【线程安全解决、并发包】

今日内容

- 线程安全问题解决
- 并发包

教学目标

- ☐ 能够使用同步方法解决线程安全问题
- ☐ 能够说明volatile关键字和synchronized关键字的区别
- ☐ 能够描述ConcurrentHashMap类的作用
- ☐ 能够描述CountDownLatch类的作用
- ☐ 能够描述CyclicBarrier类的作用
- ☐ 能够表述Semaphore类的作用
- ☐ 能够描述Exchanger类的作用

第一章 synchronized关键字

1.1 同步方法

- **同步方法**:使用synchronized修饰的方法,就叫做同步方法,保证A线程执行该方法的时候,其他线程只能在方法外等着。

格式:

```
public synchronized void method(){  
    可能会产生线程安全问题的代码  
}
```

同步锁是谁?

对于非static方法,同步锁就是this。

对于static方法,我们使用当前方法所在类的字节码对象(类名.class)。

使用同步方法代码如下:

```
public class Ticket implements Runnable{  
    private int ticket = 100;  
    /*  
     * 执行卖票操作  
     */  
    @Override  
    public void run() {  
        //每个窗口卖票的操作  
    }  
}
```

```

        //窗口 永远开启
        while(true){
            sellTicket();
        }
    }

    /**
     * 锁对象 是 谁调用这个方法 就是谁
     * 隐含 锁对象 就是 this
     */
    public synchronized void sellTicket(){
        if(ticket>0){//有票 可以卖
            //出票操作
            //使用sleep模拟一下出票时间
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            //获取当前线程对象的名字
            String name = Thread.currentThread().getName();
            System.out.println(name+"正在卖:"+ticket--);
        }
    }
}

```

1.2 Lock锁

java.util.concurrent.locks.Lock 机制提供了比synchronized代码块和synchronized方法更广泛的锁定操作,同步代码块/同步方法具有的功能Lock都有,除此之外更强大

Lock锁也称同步锁, 加锁与释放锁方法化了, 如下:

- public void lock():加同步锁。
- public void unlock():释放同步锁。

使用如下:

```

public class Ticket implements Runnable{
    private int ticket = 100;

    Lock lock = new ReentrantLock();
    /**
     * 执行卖票操作
     */
    @Override
    public void run() {
        //每个窗口卖票的操作
        //窗口 永远开启
        while(true){
            lock.lock();
            if(ticket>0){//有票 可以卖
                //出票操作
            }
        }
    }
}

```

```

        //使用sleep模拟一下出票时间
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        //获取当前线程对象的名字
        String name = Thread.currentThread().getName();
        System.out.println(name+"正在卖:"+ticket--);
    }
    lock.unlock();
}
}
}
}

```

第二章 并发包

在JDK的并发包里提供了几个非常有用的并发容器和并发工具类。供我们在多线程开发中进行使用。

2.1 CopyOnWriteArrayList

- ArrayList的线程不安全：

1. 定义线程类：

```

public class MyThread extends Thread {
    public static List<Integer> list = new ArrayList<>(); //线程不安全的
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            list.add(i);
        }
        System.out.println("添加完毕! ");
    }
}

```

2. 定义测试类：

```

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();

        Thread.sleep(1000);

        System.out.println("最终集合的长度: " + MyThread.list.size());
    }
}

```

最终结果可能会抛异常，或者最终集合大小是不正确的。

- CopyOnWriteArrayList是线程安全的：

1. 定义线程类：

```
public class MyThread extends Thread {  
    //    public static List<Integer> list = new ArrayList<>(); //线程不安全的  
    //改用：线程安全的List集合：  
    public static CopyOnWriteArrayList<Integer> list = new  
    CopyOnWriteArrayList<>();  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10000; i++) {  
            list.add(i);  
        }  
        System.out.println("添加完毕！");  
    }  
}
```

2. 测试类：

```
public class Demo {  
    public static void main(String[] args) throws InterruptedException {  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
  
        t1.start();  
        t2.start();  
  
        Thread.sleep(1000);  
  
        System.out.println("最终集合的长度：" + MyThread.list.size());  
    }  
}
```

结果始终是正确的。

2.2 CopyOnWriteArraySet

- HashSet仍然是线程不安全的：

1. 线程类：

```

public class MyThread extends Thread {
    public static Set<Integer> set = new HashSet<>(); //线程不安全的
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            set.add(i);
        }
        System.out.println("添加完毕! ");
    }
}

```

2. 测试类:

```

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();

        //主线程也添加10000个
        for (int i = 10000; i < 20000; i++) {
            MyThread.set.add(i);
        }
        Thread.sleep(1000 * 3);
        System.out.println("最终集合的长度: " + MyThread.set.size());
    }
}

```

最终结果可能会抛异常，也可能最终的长度是错误的！！

• CopyOnWriteArraySet是线程安全的:

1. 线程类:

```

public class MyThread extends Thread {
    // public static Set<Integer> set = new HashSet<>(); //线程不安全的
    //改用: 线程安全的Set集合:
    public static CopyOnWriteArraySet<Integer> set = new
CopyOnWriteArraySet<>();

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            set.add(i);
        }
        System.out.println("添加完毕! ");
    }
}

```

2. 测试类:

```

public class Demo {
    public static void main(String[] args) throws InterruptedException {

```

```

        MyThread t1 = new MyThread();
        t1.start();

        //主线程也添加10000个
        for (int i = 10000; i < 20000; i++) {
            MyThread.set.add(i);
        }
        Thread.sleep(1000 * 3);
        System.out.println("最终集合的长度: " + MyThread.set.size());
    }
}

```

可以看到结果总是正确的！！

2.3 ConcurrentHashMap

- **HashMap是线程不安全的。**

1. 线程类：

```

public class MyThread extends Thread {
    public static Map<Integer, Integer> map = new HashMap<>();
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            map.put(i, i);
        }
    }
}

```

2. 测试类：

```

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();

        for (int i = 10000; i < 20000 ; i++) {
            MyThread.map.put(i, i);
        }
        Thread.sleep(1000 * 2);

        System.out.println("map最终大小: " + MyThread.map.size());
    }
}

```

运行结果可能会出现异常、或者结果不准确！！

- **Hashtable是线程安全的，但效率低：**

我们改用JDK提供的一个早期的线程安全的Hashtable类来改写此例，注意：我们加入了"计时"。

1. 线程类：

```
public class MyThread extends Thread {
    public static Map<Integer, Integer> map = new Hashtable<>();
    @Override
    public void run() {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            map.put(i, i);
        }
        long end = System.currentTimeMillis();
        System.out.println((end - start) + " 毫秒");
    }
}
```

2. 测试类：

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new MyThread().start(); //开启1000个线程
        }

        Thread.sleep(1000 * 20); //由于每个线程执行时间稍长，所以这里多停顿一会

        System.out.println("map的最终大小: " + MyThread.map.size());
    }
}
```

3. 最终打印结果：

```
...
...
15505 毫秒
15509 毫秒
15496 毫秒
15501 毫秒
15539 毫秒
15540 毫秒
15542 毫秒
15510 毫秒
15541 毫秒
15502 毫秒
15533 毫秒
15647 毫秒
15544 毫秒
15619 毫秒
map的最终大小: 100000
```

能看到结果是正确的，但耗时较长。

- 改用ConcurrentHashMap

1. 线程类:

```
public class MyThread extends Thread {
    public static Map<Integer, Integer> map = new ConcurrentHashMap<>();

    @Override
    public void run() {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            map.put(i, i);
        }
        long end = System.currentTimeMillis();
        System.out.println((end - start) + " 毫秒");
    }
}
```

2. 测试类:

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();
        }

        Thread.sleep(1000 * 20);

        System.out.println("map的最终大小: " + MyThread.map.size());
    }
}
```

3. 最终结果:

```
...
...
3995 毫秒
3997 毫秒
4007 毫秒
4007 毫秒
4008 毫秒
4010 毫秒
4019 毫秒
4022 毫秒
4026 毫秒
3985 毫秒
4152 毫秒
4292 毫秒
map的最终大小: 100000
```

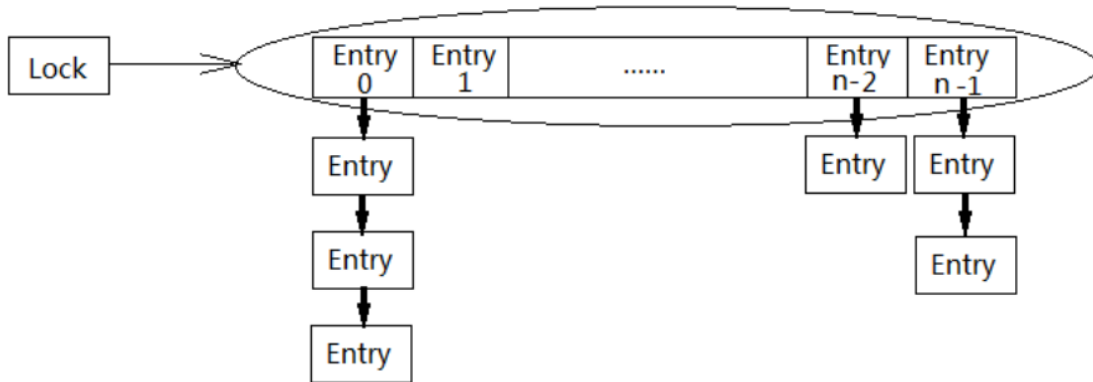
可以看到效率提高了很多!!!

- HashTable效率低下原因:


```
public synchronized V put(K key, V value)
public synchronized V get(Object key)
```

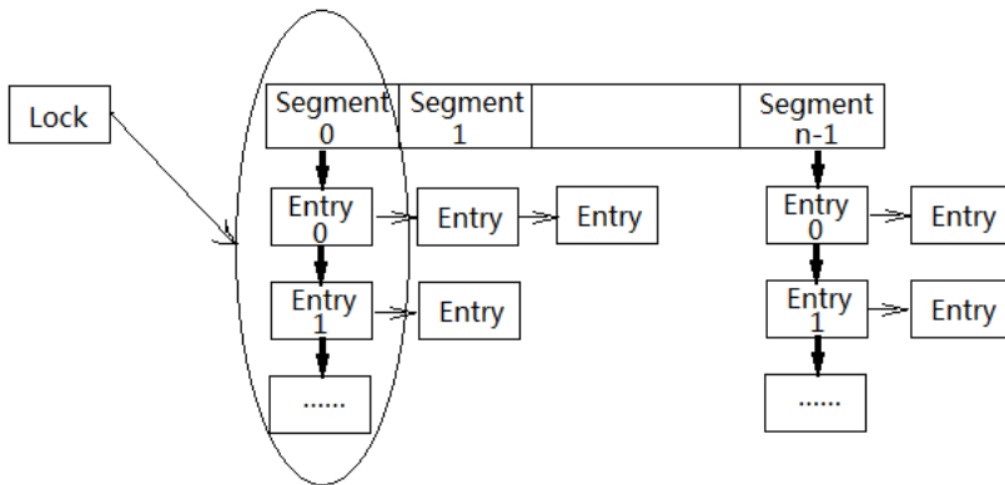
HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

Hashtable: 锁定整个哈希表，一个操作正在进行时，其它操作也同时锁定，效率低下：



ConcurrentHashMap高效的原因：CAS + 局部(synchronized)锁定

ConcurrentHashMap: 局部锁定，只锁定桶。当对当前元素锁定时，它元素不锁定



2.4 CountdownLatch

CountDownLatch允许一个或多个线程等待其他线程完成操作。

例如：线程1要执行打印：A和C，线程2要执行打印：B，但线程1在打印A后，要线程2打印B之后才能打印C，所以：线程1在打印A后，必须等待线程2打印完B之后才能继续执行。

CountDownLatch构造方法：

```
public CountdownLatch(int count)// 初始化一个指定计数器的CountDownLatch对象
```

CountDownLatch重要方法：

```
public void await() throws InterruptedException// 让当前线程等待
public void countDown() // 计数器进行减1
```

- 示例 1). 制作线程1:

```
public class ThreadA extends Thread {
    private CountdownLatch down ;
    public ThreadA(CountDownLatch down) {
        this.down = down;
    }
    @Override
    public void run() {
        System.out.println("A");
        try {
            down.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("C");
    }
}
```

- 2). 制作线程2:

```
public class ThreadB extends Thread {
    private CountdownLatch down ;
    public ThreadB(CountDownLatch down) {
        this.down = down;
    }
    @Override
    public void run() {
        System.out.println("B");
        down.countDown();
    }
}
```

- 3).制作测试类:

```
public class Demo {
    public static void main(String[] args) {
        CountdownLatch down = new CountdownLatch(1); //创建1个计数器
        new ThreadA(down).start();
        new ThreadB(down).start();
    }
}
```

- 4). 执行结果： 会保证按： A B C的顺序打印。

说明：

CountDownLatch中count down是倒数的意思，latch则是门闩的含义。整体含义可以理解为倒数的门栓，似乎有一点“三二一，芝麻开门”的感觉。

CountDownLatch是通过一个计数器来实现的，每当一个线程完成了自己的任务后，可以调用countDown()方法让计数器-1，当计数器到达0时，调用CountDownLatch。

await()方法的线程阻塞状态解除，继续执行。

2.5 CyclicBarrier

概述

CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

例如：公司召集5名员工开会，等5名员工都到了，会议开始。

我们创建5个员工线程，1个开会线程，几乎同时启动，使用CyclicBarrier保证5名员工线程全部执行后，再执行开会线程。

CyclicBarrier构造方法：

```
public CyclicBarrier(int parties, Runnable barrierAction)// 用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景
```

CyclicBarrier重要方法：

```
public int await()// 每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞
```

- 示例代码： 1). 制作员工线程：

```
public class PersonThread extends Thread {
    private CyclicBarrier cbRef;
    public PersonThread(CyclicBarrier cbRef) {
        this.cbRef = cbRef;
    }
    @Override
    public void run() {
        try {
            Thread.sleep((int) (Math.random() * 1000));
            System.out.println(Thread.currentThread().getName() + " 到了！");
            cbRef.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

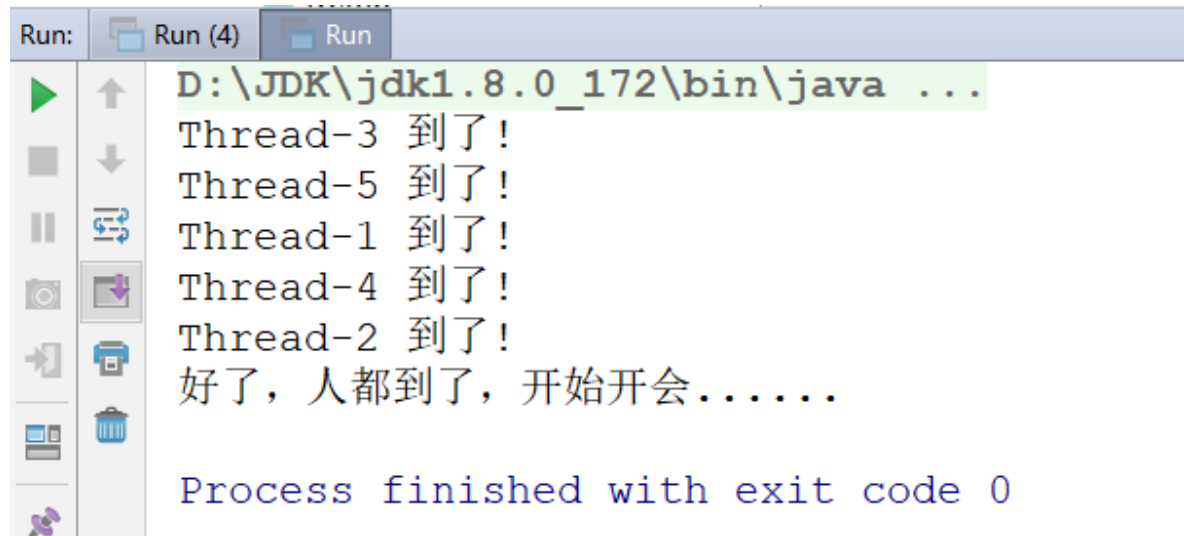
- 2). 制作开会线程：

```
public class MeetingThread extends Thread {
    @Override
    public void run() {
        System.out.println("好了，人都到了，开始开会.....");
    }
}
```

3). 制作测试类:

```
public class Demo {
    public static void main(String[] args) {
        CyclicBarrier cbRef = new CyclicBarrier(5, new MeetingThread()); //等待5个
        线程执行完毕，再执行MeetingThread
        PersonThread p1 = new PersonThread(cbRef);
        PersonThread p2 = new PersonThread(cbRef);
        PersonThread p3 = new PersonThread(cbRef);
        PersonThread p4 = new PersonThread(cbRef);
        PersonThread p5 = new PersonThread(cbRef);
        p1.start();
        p2.start();
        p3.start();
        p4.start();
        p5.start();
    }
}
```

4). 执行结果:



```
Run: Run (4) Run
D:\JDK\jdk1.8.0_172\bin\java ...
Thread-3 到了!
Thread-5 到了!
Thread-1 到了!
Thread-4 到了!
Thread-2 到了!
好了，人都到了，开始开会.....
Process finished with exit code 0
```

使用场景

使用场景: CyclicBarrier可以用于多线程计算数据，最后合并计算结果的场景。

需求: 使用两个线程读取2个文件中的数据，当两个文件中的数据都读取完毕以后，进行数据的汇总操作。

2.6 Semaphore

Semaphore的主要作用是控制线程的并发数量。

synchronized可以起到"锁"的作用，但某个时间段内，只能有一个线程允许执行。

Semaphore可以设置同时允许几个线程执行。

Semaphore字面意思是信号量的意思，它的作用是控制访问特定资源的线程数目。

Semaphore构造方法：

```
public Semaphore(int permits)           permits 表示许可线程的数量
public Semaphore(int permits, boolean fair) fair 表示公平性，如果这个设为
true 的话，下次执行的线程会是等待最久的线程
```

Semaphore重要方法：

```
public void acquire() throws InterruptedException 表示获取许可
public void release()                             release() 表示释放许可
```

- 示例一：同时允许1个线程执行

1). 制作一个Service类：

```
public class Service {
    private Semaphore semaphore = new Semaphore(1); // 1表示许可的意思，表示最多允许1个
    线程执行acquire()和release()之间的内容
    public void testMethod() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName()
                + " 进入 时间=" + System.currentTimeMillis());
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName()
                + " 结束 时间=" + System.currentTimeMillis());
            semaphore.release();
            // acquire()和release()方法之间的代码为"同步代码"
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

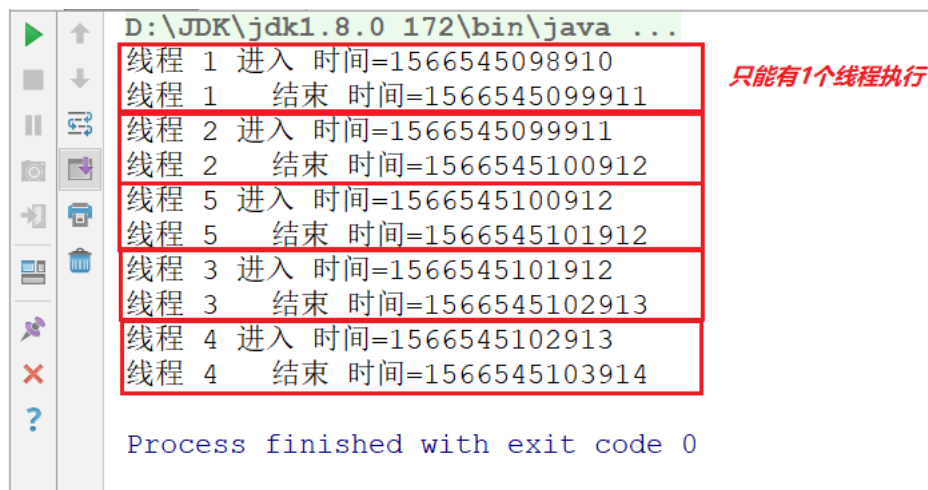
2). 制作线程类：

```
public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.testMethod();
    }
}
```

3). 测试类:

```
public class Demo {  
    public static void main(String[] args) {  
        Service service = new Service();  
        //启动5个线程  
        for (int i = 1; i <= 5; i++) {  
            ThreadA a = new ThreadA(service);  
            a.setName("线程 " + i);  
            a.start(); //5个线程会同时执行Service的testMethod方法，而某个时间段只能有1个  
            线程执行  
        }  
    }  
}
```

4). 结果:



```
D:\JDK\jdk1.8.0_172\bin\java ...  
线程 1 进入 时间=1566545098910  
线程 1 结束 时间=1566545099911  
线程 2 进入 时间=1566545099911  
线程 2 结束 时间=1566545100912  
线程 5 进入 时间=1566545100912  
线程 5 结束 时间=1566545101912  
线程 3 进入 时间=1566545101912  
线程 3 结束 时间=1566545102913  
线程 4 进入 时间=1566545102913  
线程 4 结束 时间=1566545103914  
  
Process finished with exit code 0
```

- 示例二：同时允许2个线程同时执行 1). 修改Service类，将new Semaphore(1)改为2即可：

```
public class Service {  
    private Semaphore semaphore = new Semaphore(2); //2表示许可的意思，表示最多允许2个  
    线程执行acquire()和release()之间的内容  
    public void testMethod() {  
        try {  
            semaphore.acquire();  
            System.out.println(Thread.currentThread().getName()  
                + " 进入 时间=" + System.currentTimeMillis());  
            Thread.sleep(5000);  
            System.out.println(Thread.currentThread().getName()  
                + " 结束 时间=" + System.currentTimeMillis());  
            semaphore.release();  
            //acquire()和release()方法之间的代码为"同步代码"  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

2). 再次执行结果:

```
D:\JDK\jdk1.8.0_172\bin\java ...
线程 1 进入 时间=1566545296522
线程 2 进入 时间=1566545296522
线程 2 结束 时间=1566545297523
线程 1 结束 时间=1566545297523
线程 3 进入 时间=1566545297523
线程 4 进入 时间=1566545297523
线程 3 结束 时间=1566545298523
线程 4 结束 时间=1566545298523
线程 5 进入 时间=1566545298523
线程 5 结束 时间=1566545299523

Process finished with exit code 0
```

允许2个线程同时执行

2.7 Exchanger

概述

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。

这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

Exchanger构造方法：

```
public Exchanger()
```

Exchanger重要方法：

```
public V exchange(V x)
```

- 示例一：exchange方法的阻塞特性

1).制作线程A，并能够接收一个Exchanger对象：

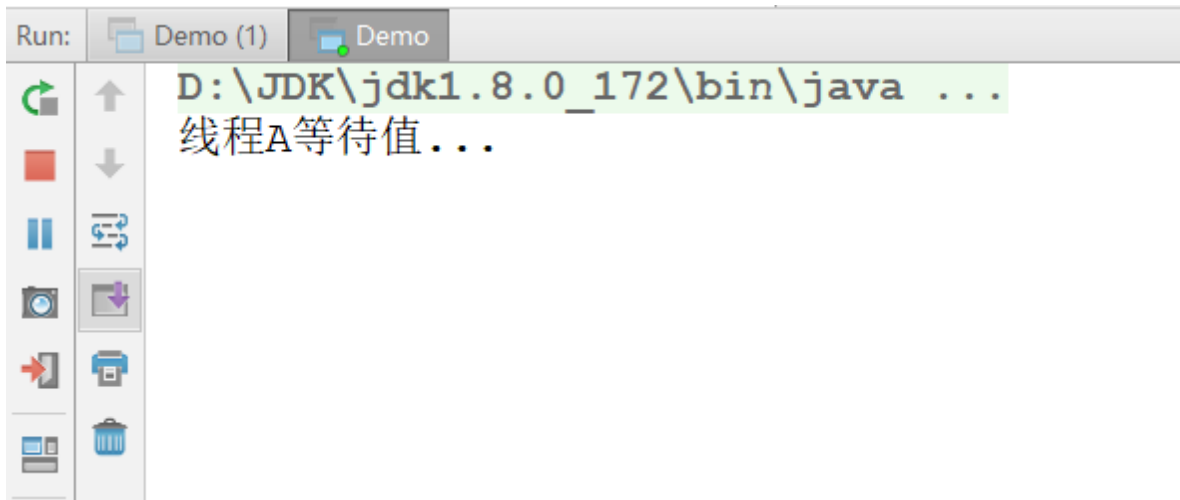
```
public class ThreadA extends Thread {
    private Exchanger<String> exchanger;
    public ThreadA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }
    @Override
    public void run() {
        try {
            System.out.println("线程A欲传递值'礼物A'给线程B，并等待线程B的值...");
            System.out.println("在线程A中得到线程B的值=" + exchanger.exchange("礼物A"));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

2). 制作main()方法:

```
public class Demo {  
    public static void main(String[] args) {  
        Exchanger<String> exchanger = new Exchanger<String>();  
        ThreadA a = new ThreadA(exchanger);  
        a.start();  
    }  
}
```

3). 执行结果:



- 示例二: exchange方法执行交换

1). 制作线程A:

```
public class ThreadA extends Thread {  
    private Exchanger<String> exchanger;  
    public ThreadA(Exchanger<String> exchanger) {  
        super();  
        this.exchanger = exchanger;  
    }  
    @Override  
    public void run() {  
        try {  
            System.out.println("线程A欲传递值'礼物A'给线程B, 并等待线程B的值...");  
            System.out.println("在线程A中得到线程B的值=" + exchanger.exchange("礼物A"));  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

2). 制作线程B:

```
public class ThreadB extends Thread {  
    private Exchanger<String> exchanger;
```



```

public ThreadB(Exchanger<String> exchanger) {
    super();
    this.exchanger = exchanger;
}
@Override
public void run() {
    try {
        System.out.println("线程B欲传递值'礼物B'给线程A，并等待线程A的值...");
        System.out.println("在线程B中得到线程A的值=" + exchanger.exchange("礼物
B"));

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

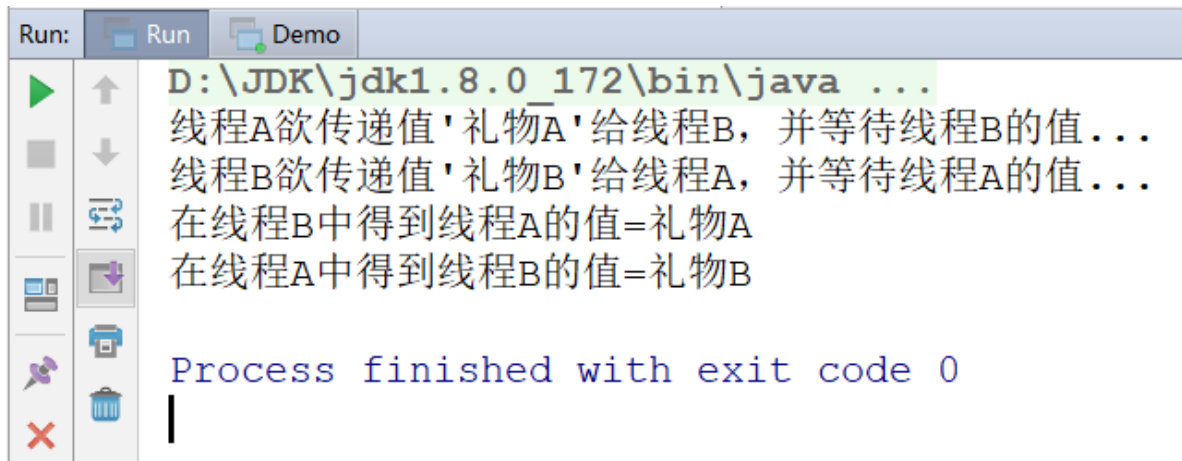
3).制作测试类:

```

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Exchanger<String> exchanger = new Exchanger<String>();
        ThreadA a = new ThreadA(exchanger);
        ThreadB b = new ThreadB(exchanger);
        a.start();
        b.start();
    }
}

```

4).执行结果:



```

Run: Demo
D:\JDK\jdk1.8.0_172\bin\java ...
线程A欲传递值'礼物A'给线程B，并等待线程B的值...
线程B欲传递值'礼物B'给线程A，并等待线程A的值...
在线程B中得到线程A的值=礼物A
在线程A中得到线程B的值=礼物B
Process finished with exit code 0

```

- 示例三: exchange方法的超时

1).制作线程A:

```

public class ThreadA extends Thread {
    private Exchanger<String> exchanger;
    public ThreadA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }
    @Override

```

```

    public void run() {
        try {
            System.out.println("线程A欲传递值'礼物A'给线程B，并等待线程B的值，只等5秒...");
            System.out.println("在线程A中得到线程B的值 =" + exchanger.exchange("礼物A",5, TimeUnit.SECONDS));
            System.out.println("线程A结束! ");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
            System.out.println("5秒钟没等到线程B的值，线程A结束! ");
        }
    }
}

```

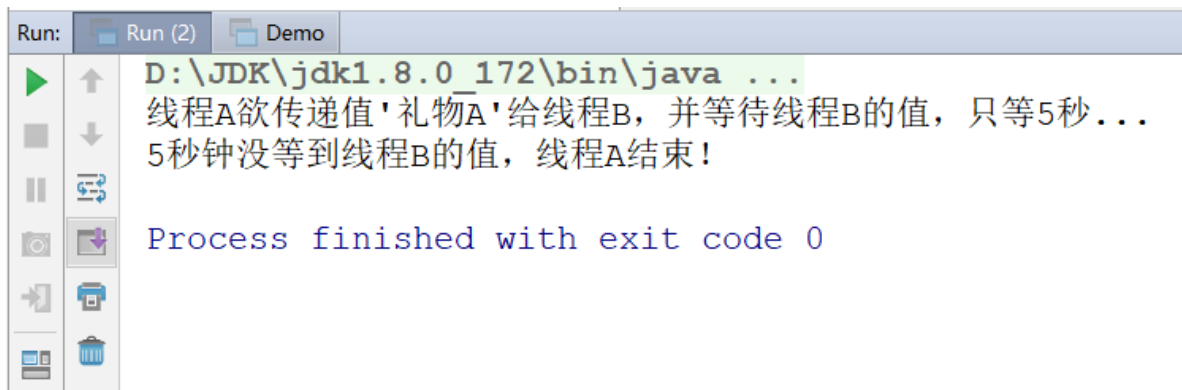
2).制作测试类:

```

public class Run {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        ThreadA a = new ThreadA(exchanger);
        a.start();
    }
}

```

3).测试结果:



使用场景

使用场景：可以做数据校对工作

需求：比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水。为了避免错误，采用AB岗两人进行录入，录入到两个文件中，系统需要加载这两个文件，

并对两个文件数据进行校对，看看是否录入一致，