

day08【线程安全、volatile关键字、原子性、并发包】

今日目标

- 线程安全
- volatile关键字
- 原子性
- 并发包

教学目标

- ☐ 能够解释安全问题的出现的原因
- ☐ 能够使用同步代码块解决线程安全问题
- ☐ 能够使用同步方法解决线程安全问题
- ☐ 能够说出volatile关键字的作用
- ☐ 能够说明volatile关键字和synchronized关键字的区别
- ☐ 能够理解原子类的工作机制
- ☐ 能够掌握原子类AtomicInteger的使用
- ☐ 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 线程安全

1.1 线程安全

如果有多个线程在同时运行，而这些线程可能会同时运行这段代码。程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

我们通过一个案例，演示线程的安全问题：

电影院要卖票，我们模拟电影院的卖票过程。假设要播放的电影是“葫芦娃大战奥特曼”，本次电影的座位共100个(本场电影只能卖100张票)。

我们来模拟电影院的售票窗口，实现多个窗口同时卖“葫芦娃大战奥特曼”这场电影票(多个窗口一起卖这100张票)

需要窗口，采用线程对象来模拟；需要票，Runnable接口子类来模拟

模拟票：

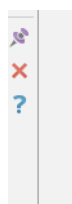
```
public class Ticket implements Runnable {
    private int ticket = 100;
    /*
     * 执行卖票操作
     */
    @Override
    public void run() {
        //每个窗口卖票的操作
        //窗口 永远开启
        while (true) {
            if (ticket > 0) { //有票 可以卖
                //出票操作
                //使用sleep模拟一下出票时间
                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--);
            }
        }
    }
}
```

测试类：

```
public class Demo {
    public static void main(String[] args) {
        //创建线程任务对象
        Ticket ticket = new Ticket();
        //创建三个窗口对象
        Thread t1 = new Thread(ticket, "窗口1");
        Thread t2 = new Thread(ticket, "窗口2");
        Thread t3 = new Thread(ticket, "窗口3");

        //同时卖票
        t1.start();
        t2.start();
        t3.start();
    }
}
```

结果中有一部分这样现象：



窗口3正在卖:6
窗口1正在卖:5
窗口2正在卖:5
窗口3正在卖:4
窗口1正在卖:3
窗口2正在卖:2
窗口3正在卖:1
窗口2正在卖:0
窗口1正在卖:-1

发现程序出现了两个问题:

1. 相同的票数,比如5这张票被卖了两回。
2. 不存在的票, 比如0票与-1票, 是不存在的。

这种问题, 几个窗口(线程)票数不同步了, 这种问题称为线程不安全。

线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作, 而无写操作, 一般来说, 这个全局变量是线程安全的; 若有多个线程同时执行写操作, 一般都需要考虑线程同步, 否则的话就可能影响线程安全。

1.2 线程同步

线程同步是为了解决线程安全问题。

当我们使用多个线程访问同一资源的时候, 且多个线程中对资源有写的操作, 就容易出现线程安全问题。

要解决上述多线程并发访问一个资源的安全性问题:也就是解决重复票与不存在票问题, Java中提供了同步机制(**synchronized**)来解决。

根据案例简述:

窗口1线程进入操作的时候, 窗口2和窗口3线程只能在外等着, 窗口1操作结束, 窗口1和窗口2和窗口3才有机会进入代码去执行。也就是说在某个线程修改共享资源的时候, 其他线程不能修改该资源, 等待修改完毕同步之后, 才能去抢夺CPU资源, 完成对应的操作, 保证了数据的同步性, 解决了线程不安全的现象。

为了保证每个线程都能正常执行原子操作Java引入了线程同步机制。

那么怎么去使用呢? 有三种方式完成同步操作:

1. 同步代码块。
2. 同步方法。
3. 锁机制。

1.3 同步代码块

- **同步代码块:** `synchronized` 关键字可以用于方法中的某个区块中, 表示只对这个区块的资源实行互斥访问。

格式:

```
synchronized(同步锁){  
    需要同步操作的代码  
}
```

同步锁:

对象的同步锁只是一个概念,可以想象为在对象上标记了一个锁.

1. 锁对象 可以是任意类型。
2. 多个线程对象 要使用同一把锁。

注意:在任何时候,最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着(BLOCKED)。

使用同步代码块解决代码:

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

    Object lock = new Object();
    /*
     * 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){
            synchronized (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--);
                }
            }
        }
    }
}
```

当使用了同步代码块后,上述的线程的安全问题,解决了。

1.4 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--);
        }
    }
}

```

1.5 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 volatile关键字

2.1. 看程序说结果

```

public class VolatileThread extends Thread {

    // 定义成员变量
    private boolean flag = false ;
    public boolean isFlag() { return flag;}

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 将flag的值更改为true
        this.flag = true ;
        System.out.println("flag=" + flag);

    }
}

public class VolatileThreadDemo { // 测试类

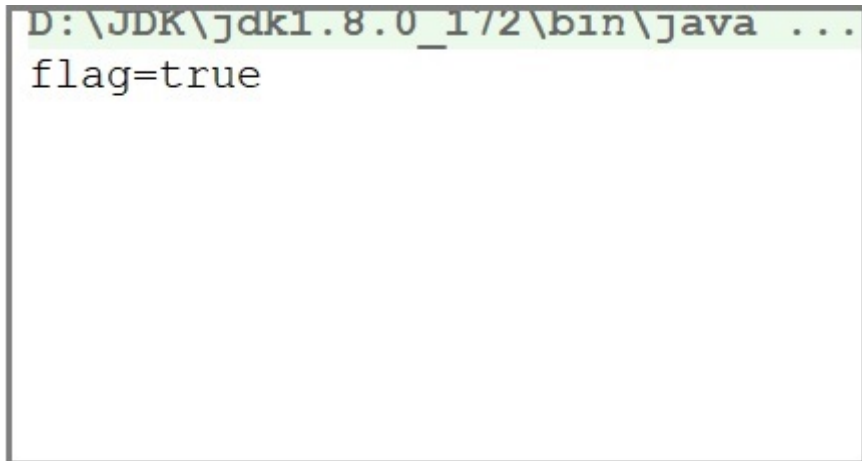
    public static void main(String[] args) {

        // 创建VolatileThread线程对象
        VolatileThread volatileThread = new VolatileThread() ;
        volatileThread.start();
    }
}

```

```
// main方法
while(true) {
    if(volatileThread.isFlag()) {
        System.out.println("执行了=====");
    }
}
}
```

结果：



我们看到，VolatileThread线程中已经将flag设置为true，但main()方法中始终没有读到，从而没有打印。

2.2. JMM

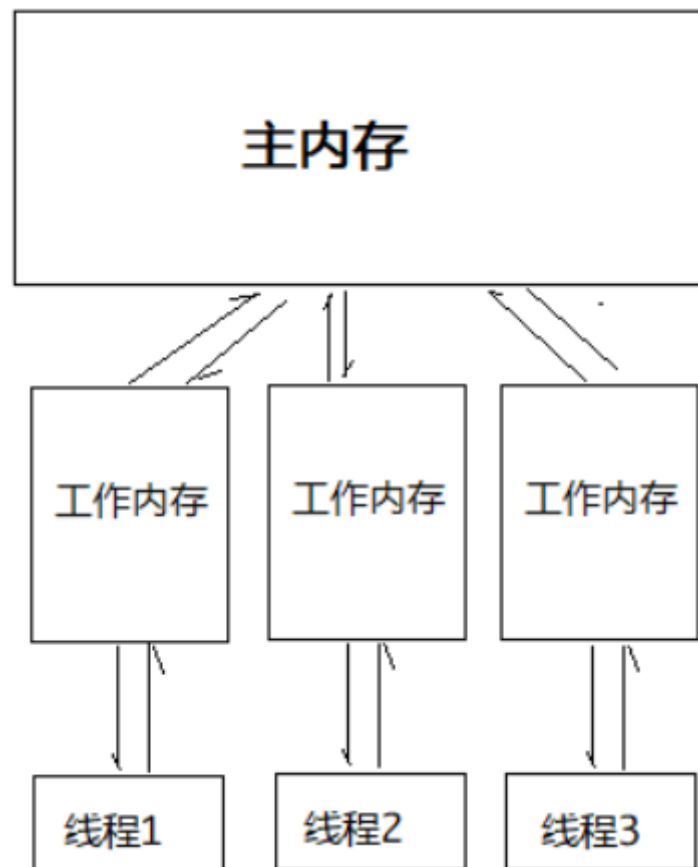
概述：JMM(Java Memory Model)Java内存模型,是java虚拟机规范中所定义的一种内存模型。

Java内存模型(Java Memory Model)描述了Java程序中各种变量(线程共享变量)的访问规则，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节。

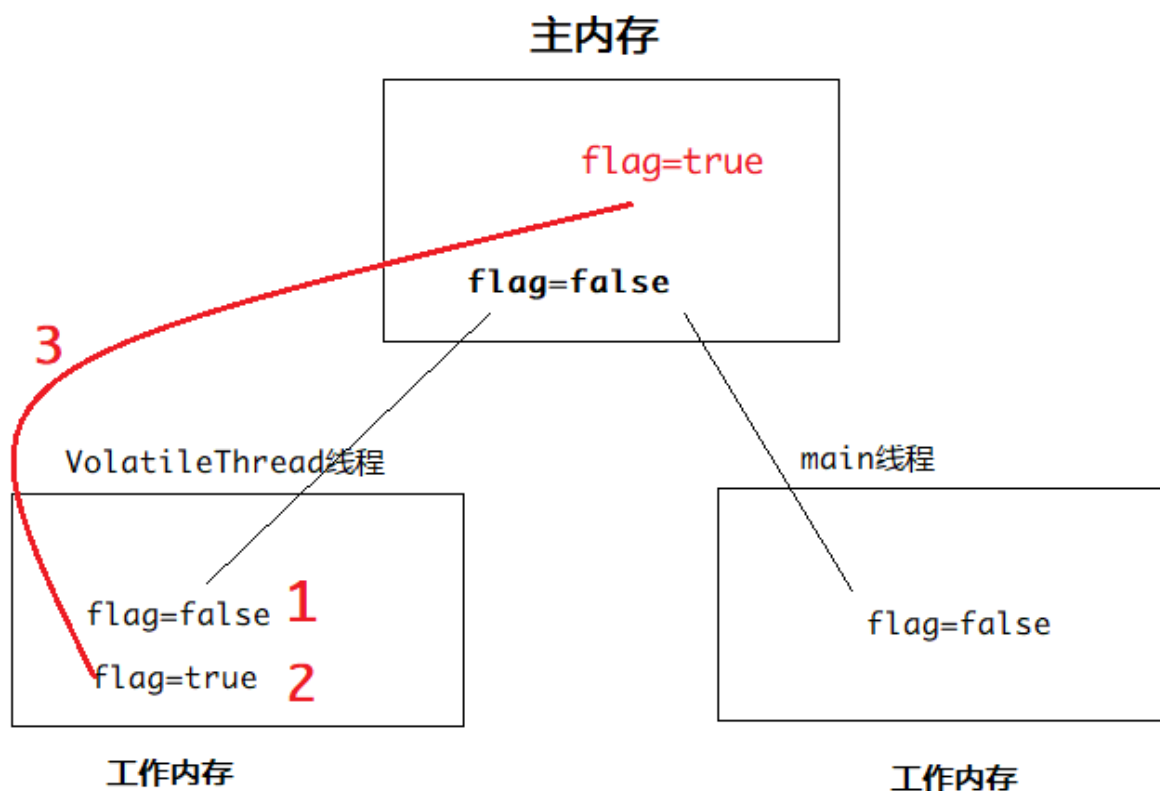
所有的共享变量都存储于主内存。这里所说的变量指的是实例变量和类变量。不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。每一个线程还存在自己的工作内存，线程

的工作内存，保留了被线程使用的变量的工作副本。线程对变量的所有的操作(读，取)都必须在工作内存中完成，而不能直接读写主内存中的变量，不同线程之间也不能直接访问

对方工作内存中的变量，线程间变量的值的传递需要通过主内存完成。



2.3. 问题分析



1. VolatileThread线程从主内存读取到数据放入其对应的工作内存
2. 将flag的值更改为true，但是这个时候flag的值还没有写会主内存
3. 此时main方法读取到了flag的值为false
4. 当VolatileThread线程将flag的值写回去后，但是main函数里面的while(true)调用的是系统比较底层的代码，速度快，快到没有时间再去读取主存中的值，

所以while(true)读取到的值一直是false。(如果有一个时刻main线程从主内存中读取到了主内存中flag的最新值，那么if语句就可以执行，main线程何时从主内存中读取最新的值，我们无法控制)

2.4. 问题处理

加锁

```
// main方法
while(true) {
    synchronized (volatileThread) {
        if(volatileThread.isFlag()) {
            System.out.println("执行了=====");
        }
    }
}
```

某一个线程进入synchronized代码块前后，执行过程入如下：

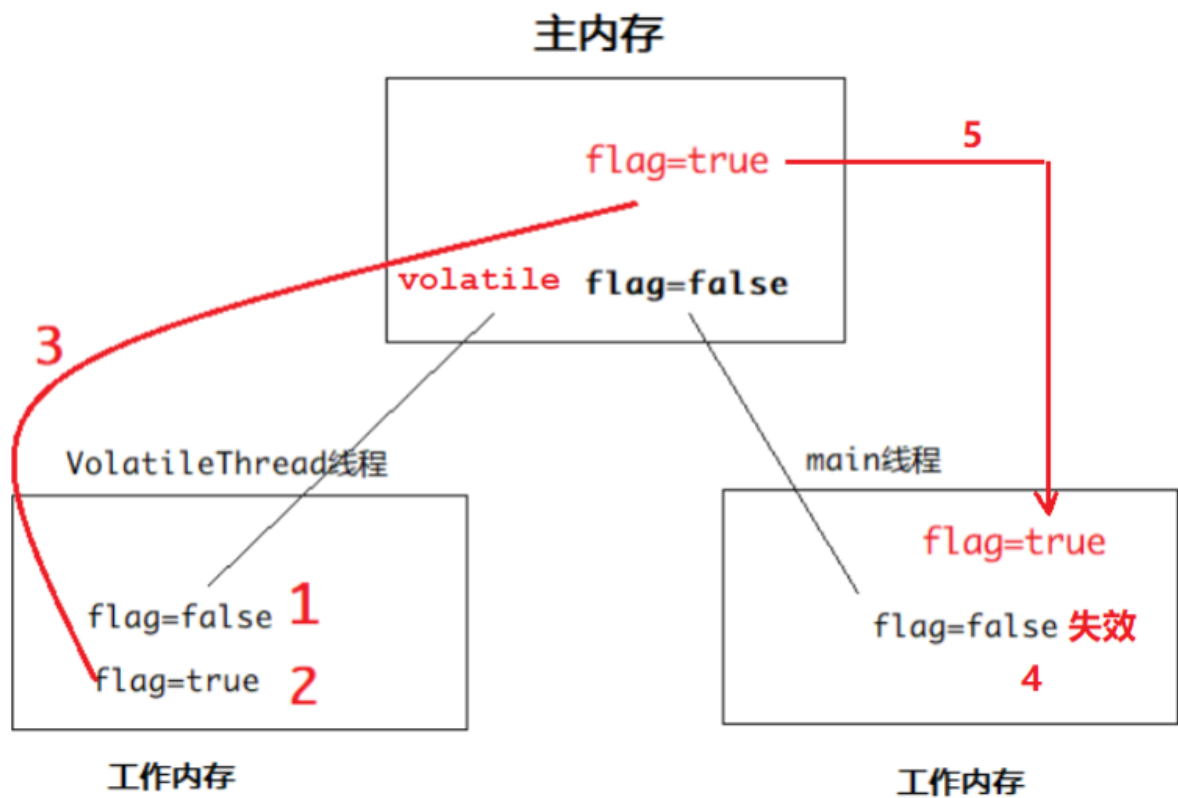
- a.线程获得锁
- b.清空工作内存
- c.从主内存拷贝共享变量最新的值到工作内存成为副本
- d.执行代码
- e.将修改后的副本的值刷新回主内存中
- f.线程释放锁

volatile关键字

使用volatile关键字：

```
private volatile boolean flag ;
```

工作原理：



1. VolatileThread线程从主内存读取到数据放入其对应的工作内存
2. 将flag的值更改为true，但是这个时候flag的值还没有写会主内存
3. 此时main方法main方法读取到了flag的值为false
4. 当VolatileThread线程将flag的值写回去后，失效其他线程对此变量副本
5. 再次对flag进行操作的时候线程会从主内存读取最新的值，放入到工作内存中

总结：volatile保证不同线程对共享变量操作的可见性，也就是说一个线程修改了volatile修饰的变量，当修改写回主内存时，另外一个线程立即看到最新的值。

但是volatile不保证原子性。

volatile与synchronized

- volatile只能修饰实例变量和类变量，而synchronized可以修饰方法，以及代码块。
- volatile保证数据的可见性，但是不保证原子性(多线程进行写操作，不保证线程安全);而synchronized是一种排他（互斥）的机制，

第三章 原子性

概述：所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，

多个操作是一个不可以分割的整体。

比如：从张三的账户给李四的账户转1000元，这个动作将包含两个基本的操作：从张三的账户扣除1000元，给李四的账户增加1000元。这两个操作必须符合原子性的要求，

要么都成功要么都失败。

3.1. 看程序说结果

```

public class volatileAtomicThread implements Runnable {

    // 定义一个int类型的遍历
    private int count = 0 ;

    @Override
    public void run() {

        // 对该变量进行++操作，100次
        for(int x = 0 ; x < 100 ; x++) {
            count++ ;
            System.out.println("count =====>>>> " + count);
        }
    }

}

public class volatileAtomicThreadDemo {

    public static void main(String[] args) {

        // 创建volatileAtomicThread对象
        volatileAtomicThread volatileAtomicThread = new volatileAtomicThread() ;

        // 开启100个线程对count进行++操作
        for(int x = 0 ; x < 100 ; x++) {
            new Thread(volatileAtomicThread).start();
        }

    }

}

```

执行结果：不保证一定是10000

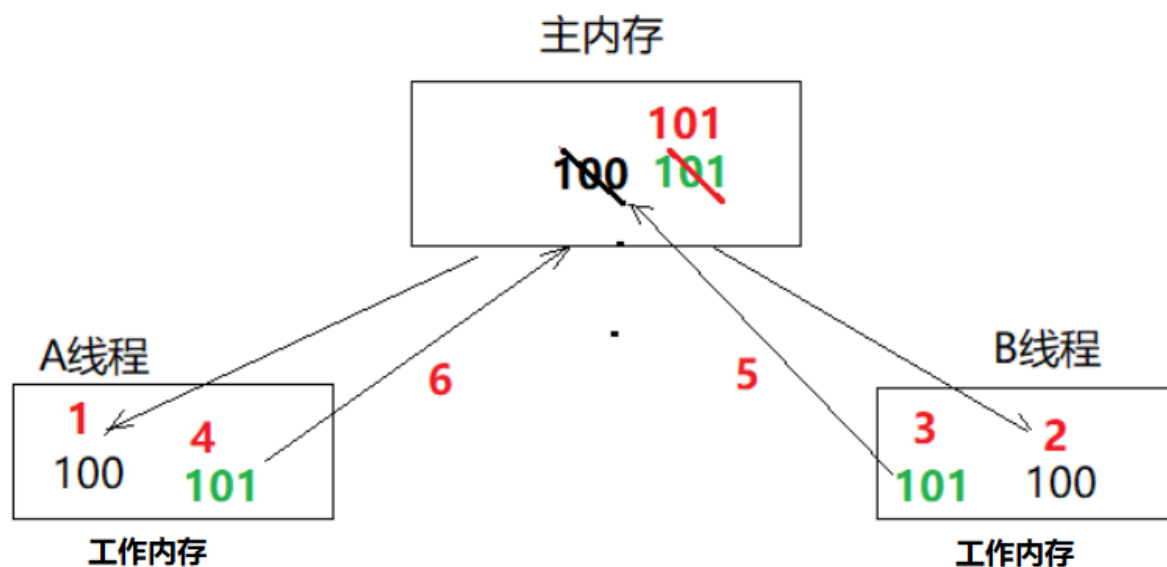
3.2. 问题原理说明

以上问题主要是发生在count++操作上：

count++操作包含3个步骤：

- 从主内存中读取数据到工作内存
- 对工作内存中的数据进行++操作
- 将工作内存中的数据写回到主内存

count++操作不是一个原子性操作，也就是说在某一个时刻对某一个操作的执行，有可能被其他的线程打断。



1) 假设此时x的值是100，线程A需要对改变量进行自增1的操作，首先它需要从主内存中读取变量x的值。由于CPU的切换关系，此时CPU的执行权被切换到了

B线程。A线程就处于就绪状态，B线程处于运行状态

2) 线程B也需要从主内存中读取x变量的值,由于线程A没有对x值做任何修改因此此时B读取到的数据还是100

3) 线程B工作内存中x执行了+1操作，但是未刷新之主内存中

4) 此时CPU的执行权切换到了A线程上，由于此时线程B没有将工作内存中的数据刷新到主内存，因此A线程工作内存中的变量值还是100，没有失效。

A线程对工作内存中的数据进行了+1操作

5) 线程B将101写入到主内存

6) 线程A将101写入到主内存

虽然计算了2次，但是只对A进行了1次修改。

3.3. volatile原子性测试

代码测试

```
// 定义一个int类型的变量
private volatile int count = 0 ;
```

小结：在多线程环境下，volatile关键字可以保证共享数据的可见性，但是并不能保证对数据操作的原子性（在多线程环境下volatile修饰的变量也是线程不安全的）。

在多线程环境下，要保证数据的安全性，我们还需要使用锁机制。

volatile的使用场景

- 开关控制

利用可见性特点，控制某一段代码执行或者关闭(比如今天课程的第一个案例)。

- 多个线程操作共享变量，但是是有一个线程对其进行写操作，其他的线程都是读

3.4. 问题解决

使用锁机制

我们可以给count++操作添加锁，那么count++操作就是临界区的代码，临界区只能有一个线程去执行，所以count++就变成了原子操作。

```
public class VolatileAtomicThread implements Runnable {

    // 定义一个int类型的变量
    private volatile int count = 0 ;
    private static final Object obj = new Object();

    @Override
    public void run() {

        // 对该变量进行++操作，100次
        for(int x = 0 ; x < 100 ; x++) {
            synchronized (obj) {
                count++ ;
                System.out.println("count =====>>>> " + count);
            }
        }

    }

}
```

原子类

概述：java从JDK1.5开始提供了java.util.concurrent.atomic包(简称Atomic包)，这个包中的原子操作类提供了一种用法简单，性能高效，线程安全地更新一个变量的方式。

AtomicInteger

原子型Integer，可以实现原子更新操作

<code>public AtomicInteger():</code>	初始化一个默认值为0的原子型Integer
<code>public AtomicInteger(int initialValue):</code>	初始化一个指定值的原子型Integer
<code>int get():</code>	获取值
<code>int getAndIncrement():</code>	以原子方式将当前值加1，注意，这里返回的是自增前的值。
<code>int incrementAndGet():</code>	以原子方式将当前值加1，注意，这里返回的是自增后的值。
<code>int addAndGet(int data):</code>	以原子方式将输入的数值与实例中的值
(AtomicInteger里的value)相加，并返回结果。	
<code>int getAndSet(int value):</code>	以原子方式设置为newValue的值，并返回旧值。

演示基本使用。

案例改造

使用AtomicInteger对案例进行改造.

```

public class VolatileAtomicThread implements Runnable {

    // 定义一个int类型的变量
    private AtomicInteger atomicInteger = new AtomicInteger();

    @Override
    public void run() {

        // 对该变量进行++操作, 100次
        for(int x = 0 ; x < 100 ; x++) {
            int i = atomicInteger.getAndIncrement();
            System.out.println("count =====>>> " + i);
        }

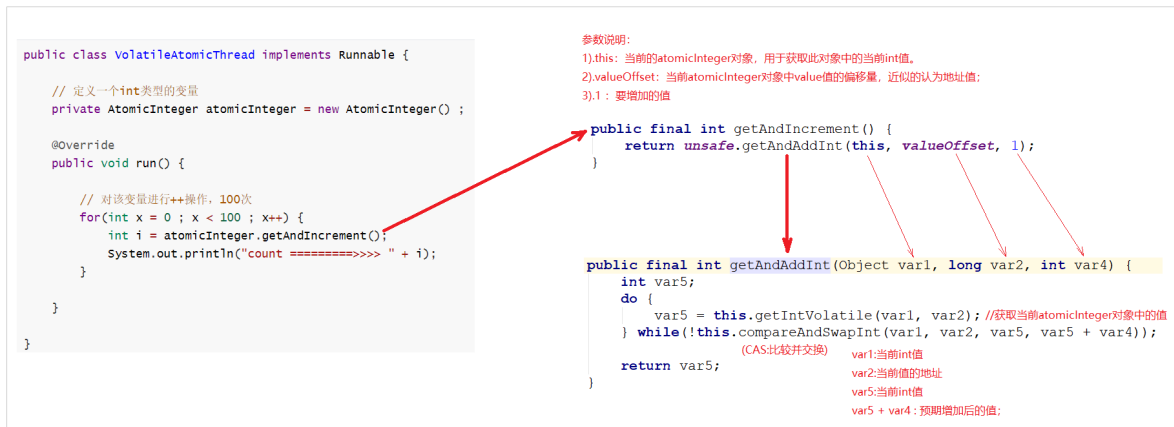
    }

}

```

原子类CAS机制

概述



CAS的全称是: Compare And Swap(比较再交换); 是现代CPU广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。CAS可以将read-modify-write

转换为原子操作, 这个原子操作直接由处理器保证。

CAS机制当中使用了3个基本操作数: 内存地址V, 旧的预期值A, 要修改的新值B。

举例:

1. 在内存地址V当中, 存储着值为10的变量。



2. 此时线程1想要把变量的值增加1。对线程1来说, 旧的预期值A=10, 要修改的新值B=11。



内存地址V

线程1: A = 10 B = 11

3. 在线程1要提交更新之前，另一个线程2抢先一步，把内存地址V中的变量值率先更新成了11。

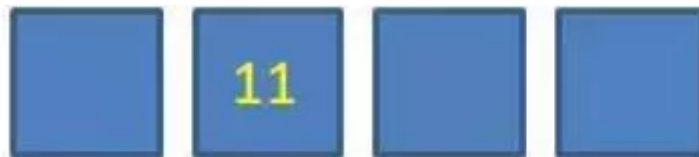


内存地址V

线程1: A = 10 B = 11

线程2: 把变量值更新为11

4. 线程1开始提交更新，首先进行A和地址V的实际值比较（Compare），发现A不等于V的实际值，提交失败。



内存地址V

线程1: A = 10 B = 11

A != V的值 (10 != 11)
提交失败！

线程2: 把变量值更新为11

5. 线程1重新获取内存地址V的当前值，并重新计算想要修改的新值。此时对线程1来说，A=11，B=12。这个重新尝试的过程被称为自旋。



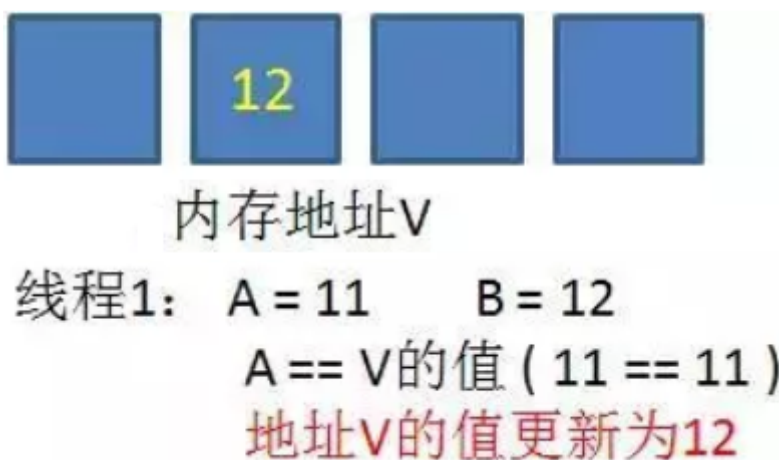
内存地址V

线程1: A = 11 B = 12

6. 这一次比较幸运，没有其他线程改变地址V的值。线程1进行Compare，发现A和地址V的实际值是相等的。



7. 线程1进行SWAP，把地址V的值替换为B，也就是12。



CAS与Synchronized

CAS和Synchronized都可以保证多线程环境下共享数据的安全性。那么他们两者有什么区别？

Synchronized是从悲观的角度出发：

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁

（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。因此Synchronized我们也将之称之为悲观锁。jdk中的ReentrantLock也是一种悲观锁。

CAS是从乐观的角度出发：

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据。

CAS这种机制我们也可以将其称之为乐观锁。

第四章 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。

4.1 ConcurrentHashMap

为什么要使用ConcurrentHashMap：

1. HashMap线程不安全，会导致数据错乱
2. 使用线程安全的Hashtable效率低下

基于以上两个原因，便有了ConcurrentHashMap的登场机会。

- **HashMap线程不安全演示。**

公有、静态的集合：

```
public class Const {  
    public static HashMap<String,String> map = new HashMap<>();  
}
```

线程，向map中写入数据：

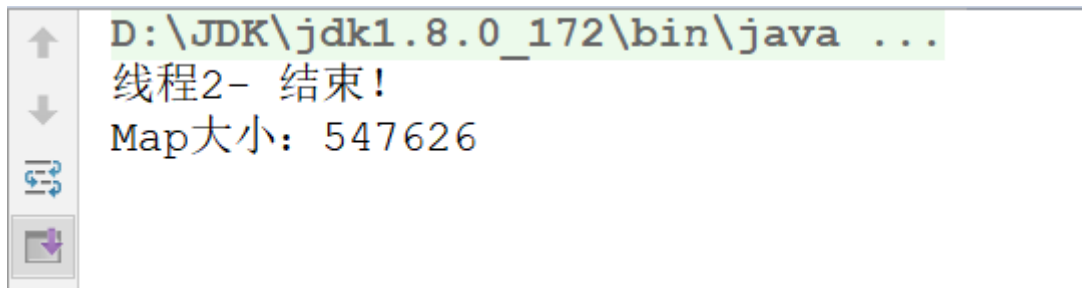
```
public void run() {  
    for (int i = 0; i < 500000; i++) {  
        Const.map.put(this.getName() + (i + 1), this.getName() + i + 1);  
    }  
    System.out.println(this.getName() + " 结束！");  
}
```

测试类：

```
public class Demo {  
    public static void main(String[] args) throws InterruptedException {  
        Thread1A a1 = new Thread1A();  
        Thread1A a2 = new Thread1A();  
        a1.setName("线程1-");  
        a2.setName("线程2-");  
  
        a1.start();  
        a2.start();  
        //休息10秒，确保两个线程执行完毕  
        Thread.sleep(1000 * 5);  
        //打印集合大小  
        System.out.println("Map大小: " + Const.map.size());  
    }  
}
```

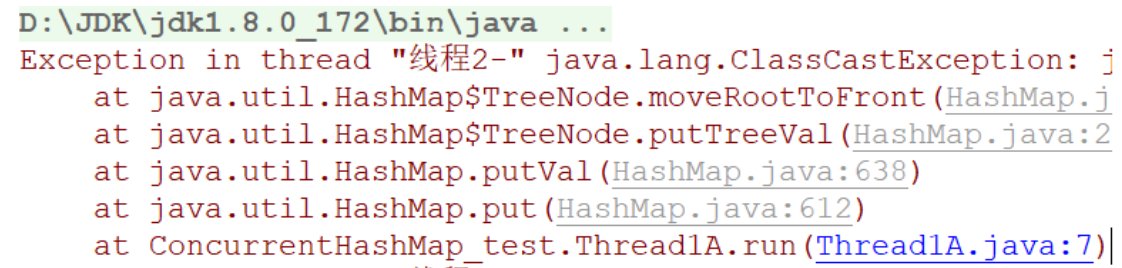
说明：两个线程分别向同一个map中写入50000个键值对，最后map的size应为：100000，但多运行几次会发现以下几种错误：

1. 假死：



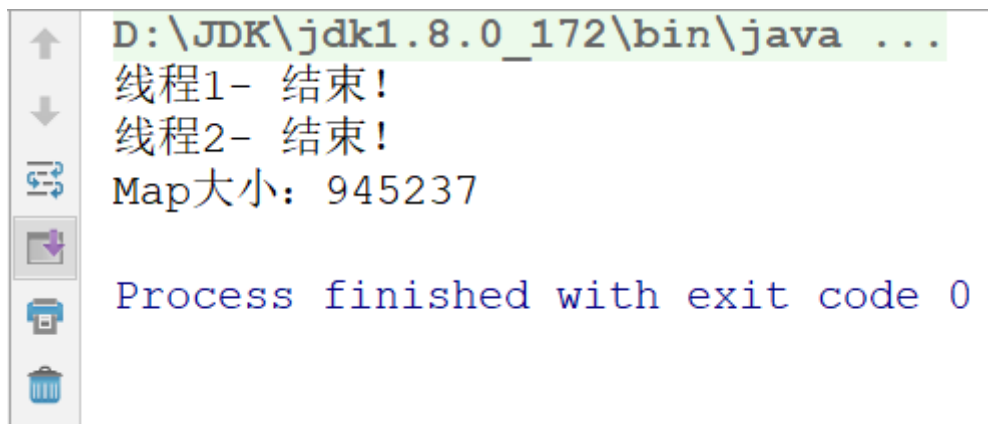
```
D:\JDK\jdk1.8.0_172\bin\java ...
线程2- 结束!
Map大小: 547626
```

2. 异常:



```
D:\JDK\jdk1.8.0_172\bin\java ...
Exception in thread "线程2-" java.lang.ClassCastException: j
at java.util.HashMap$TreeNode.moveRootToFront (HashMap.j
at java.util.HashMap$TreeNode.putTreeVal (HashMap.java:2
at java.util.HashMap.putVal (HashMap.java:638)
at java.util.HashMap.put (HashMap.java:612)
at ConcurrentHashMap_test.Thread1A.run (Thread1A.java:7)|
```

3. 错误结果:



```
D:\JDK\jdk1.8.0_172\bin\java ...
线程1- 结束!
线程2- 结束!
Map大小: 945237

Process finished with exit code 0
```

- 为了保证线程安全, 可以使用Hashtable。注意: 线程中加入了计时

公有、静态的集合:

```
public class Const {
    public static Hashtable<String,String> map = new Hashtable<>();
}
```

线程, 向map中写入数据:

```
public void run() {
    long start = System.currentTimeMillis();
    for (int i = 0; i < 500000; i++) {
        Const.map.put(this.getName() + (i + 1), this.getName() + i + 1);
    }
    long end = System.currentTimeMillis();
    System.out.println(this.getName() + " 结束! 用时: " + (end - start) +
        " 毫秒");
}
```

测试类:

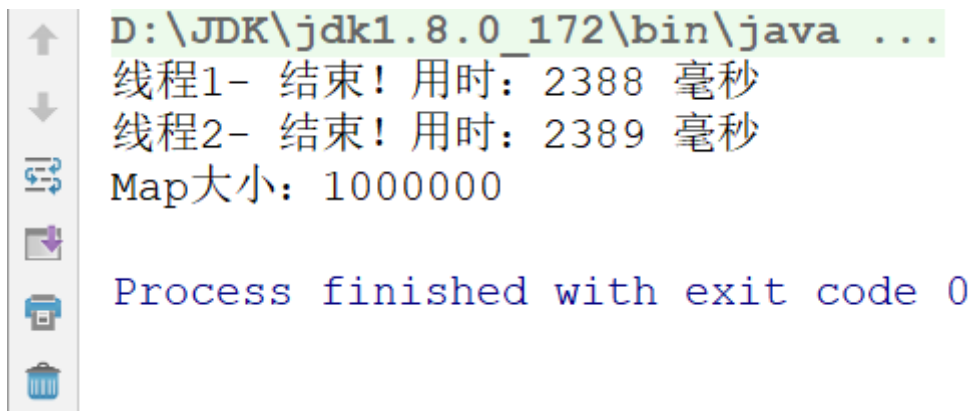
```

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Thread1A a1 = new Thread1A();
        Thread1A a2 = new Thread1A();
        a1.setName("线程1-");
        a2.setName("线程2-");

        a1.start();
        a2.start();
        //休息10秒，确保两个线程执行完毕
        Thread.sleep(1000 * 5);
        //打印集合大小
        System.out.println("Map大小: " + Const.map.size());
    }
}

```

执行结果：



```

D:\JDK\jdk1.8.0_172\bin\java ...
线程1- 结束! 用时: 2388 毫秒
线程2- 结束! 用时: 2389 毫秒
Map大小: 1000000

Process finished with exit code 0

```

可以看到，Hashtable保证的线程安全，时间是2秒多。

- 再看ConcurrentHashMap

公有、静态的集合：

```

public class Const {
    public static ConcurrentHashMap<String,String> map = new
    ConcurrentHashMap<>();
}

```

线程，向map中写入数据：

```

public void run() {
    long start = System.currentTimeMillis();
    for (int i = 0; i < 500000; i++) {
        Const.map.put(this.getName() + (i + 1), this.getName() + i + 1);
    }
    long end = System.currentTimeMillis();
    System.out.println(this.getName() + " 结束! 用时: " + (end - start) +
    " 毫秒");
}

```

测试类：

```

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

```

```

Thread1A a1 = new Thread1A();
Thread1A a2 = new Thread1A();
a1.setName("线程1-");
a2.setName("线程2-");

a1.start();
a2.start();
//休息10秒，确保两个线程执行完毕
Thread.sleep(1000 * 5);
//打印集合大小
System.out.println("Map大小: " + Const.map.size());
}
}

```

执行结果：

```

D:\JDK\jdk1.8.0_172\bin\java ...
线程1- 结束! 用时: 1684 毫秒
线程2- 结束! 用时: 1700 毫秒
Map大小: 1000000

Process finished with exit code 0

```

ConcurrentHashMap仍能保证结果正确，而且提高了效率。

HashTable效率低下原因：

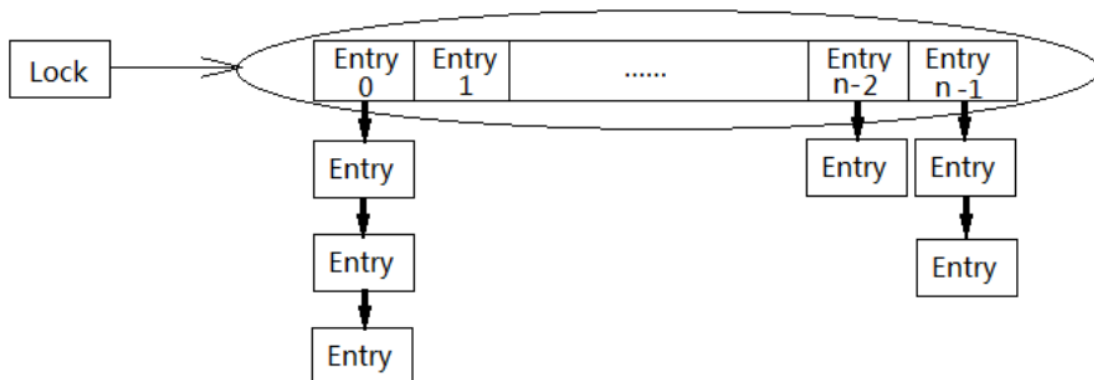
```

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

```

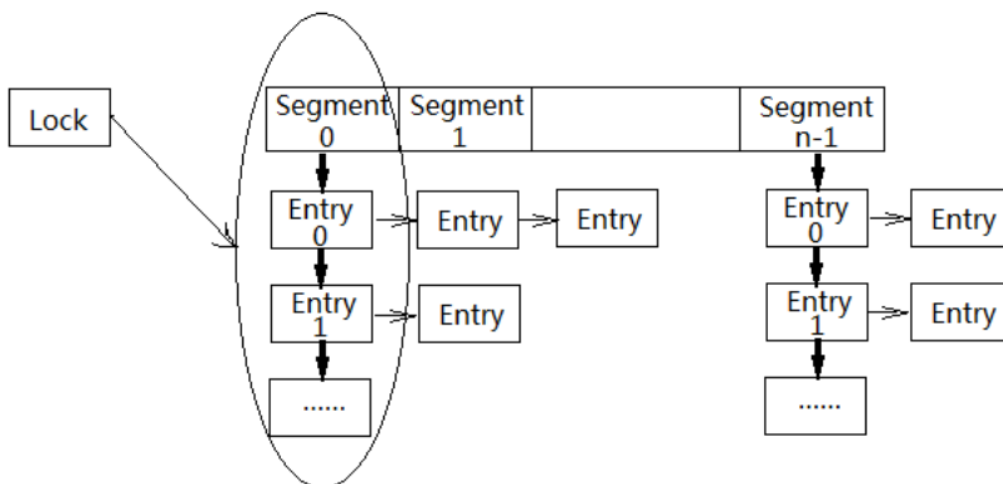
HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

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



4.2 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。

4.3 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();
        }
    }
}
```

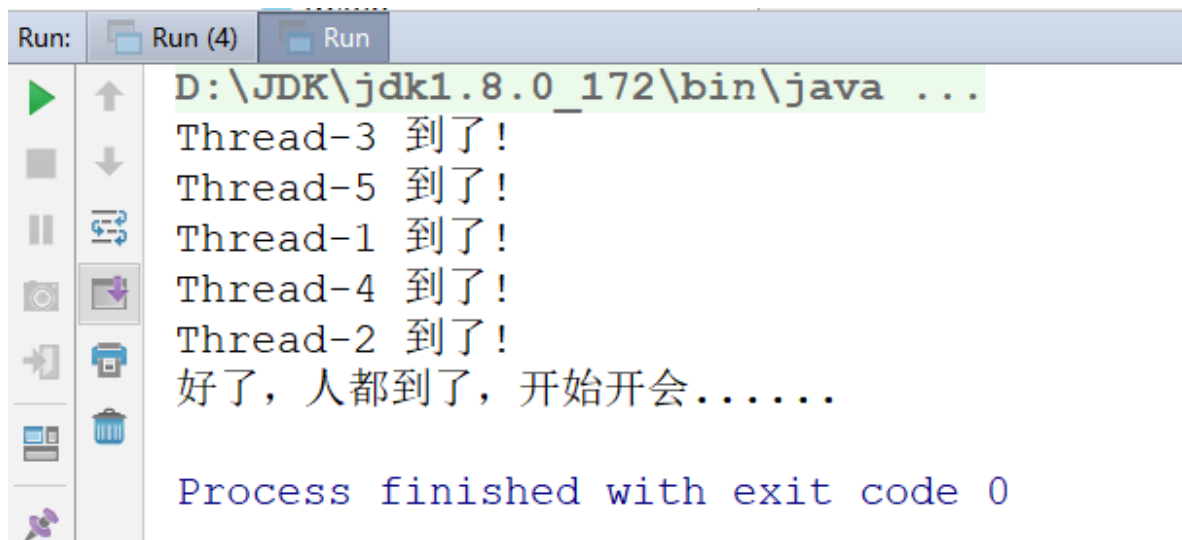
- 2). 制作开会线程：

```
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。

4.4 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
```

只能有1个线程执行

- 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

4.5 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

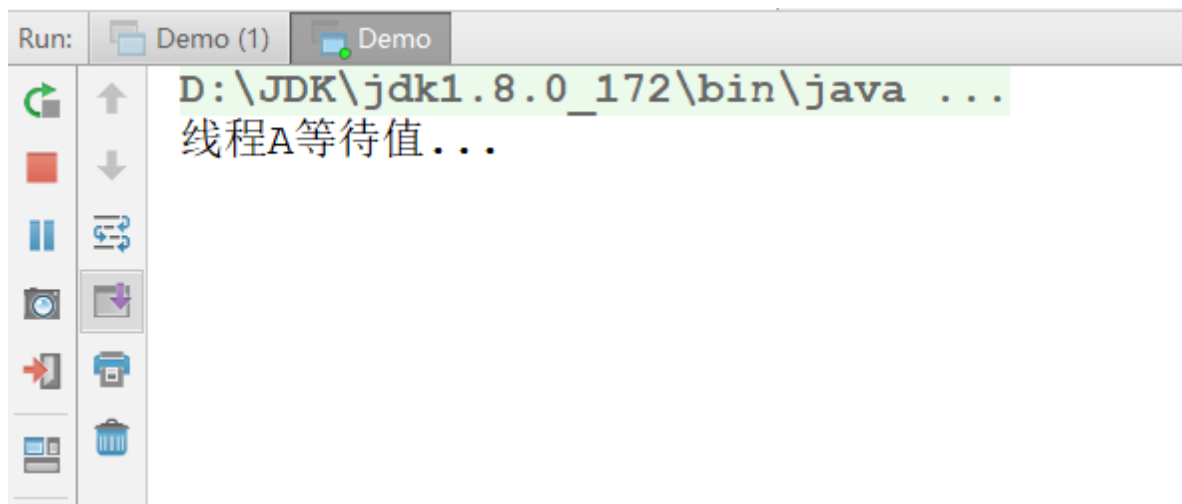
1).制作线程A，并能够接收一个Exchanger对象：

```
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:

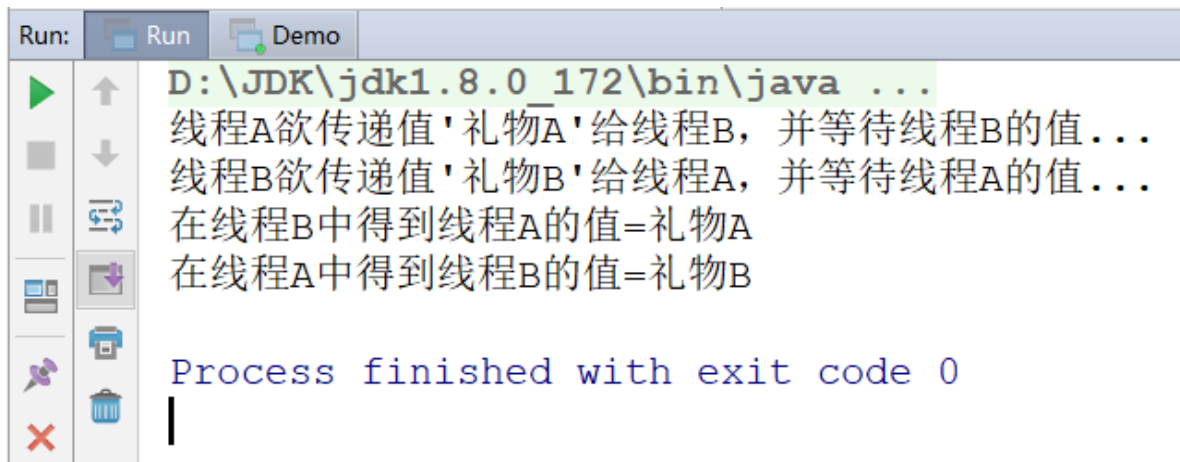
```
public class ThreadB extends Thread {
    private Exchanger<String> exchanger;
    public ThreadB(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }
    @Override
    public void run() {
        try {
            System.out.println("线程B欲传递值'礼物B'给线程A，并等待线程A的值...");
            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:



• 示例三: exchange方法的超时

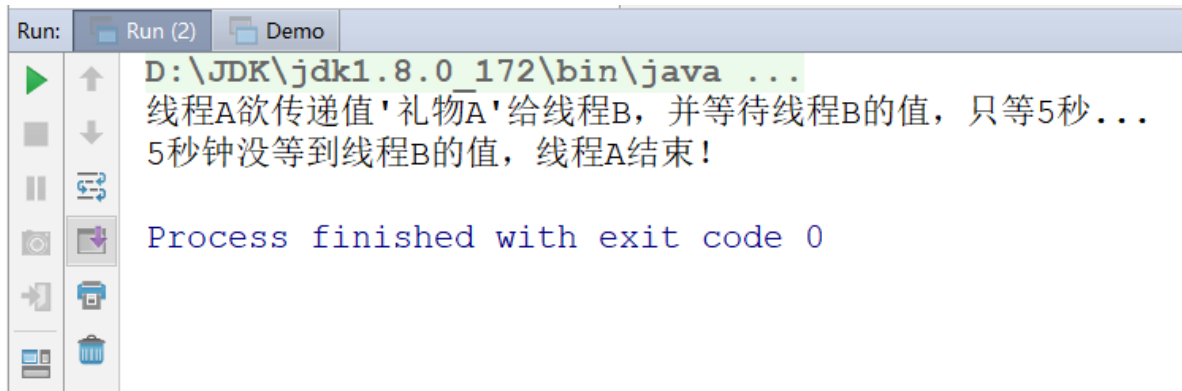
1).制作线程A:

```
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,