

JUC并发编程「第三讲」

前言

随着互联网公司招聘优质工程师的需求日渐提高，对程序员的技术要求也是水涨船高，不再是仅懂得API调用就能拿高薪offer。

想要追求高薪，必须要精业务、懂原理、挖源码、会性能调优才能够搞定。

其各互联网一面试题，浮出水面，最典型、问到最多的技术为Java并发编程、JVM

JUC并发包API 包介绍

The screenshot shows the Oracle Java SE 14 API documentation for the `java.util.concurrent` package. The page is titled "模块java.base" and includes a search bar. The "配套" (Related) section lists several packages, with `java.util.concurrent` highlighted in a red box. The description for `java.util.concurrent` is: "实用程序类通常在并发编程中很有用。" (Utility classes are often useful in concurrent programming.)

出口产品	描述
<code>java.util</code>	包含集合框架、一些国际化支持类、服务加载程序、属性、随机数生成、字符串解析和扫描类、base64编码和解码、位数组以及几个其他实用程序类。
<code>java.util.concurrent</code>	实用程序类通常在并发编程中很有用。
<code>java.util.concurrent.atomic</code>	一个小的类工具包，支持对单个变量进行无锁线程安全编程。
<code>java.util.concurrent.locks</code>	接口和类提供了用于锁定和等待条件的框架，这些条件不同于内置的同步和监视器。

java.util.concurrent:

1. 并发与并行的不同？

1. 并发，如同，秒杀一样，多个线程访问同一个资源
2. 并行，一堆事情一块去做，如同，一遍烧热水，一个拆方便面包装

java.util.concurrent.atomic

1. AtomicInteger 原子性引用

java.util.concurrent.locks

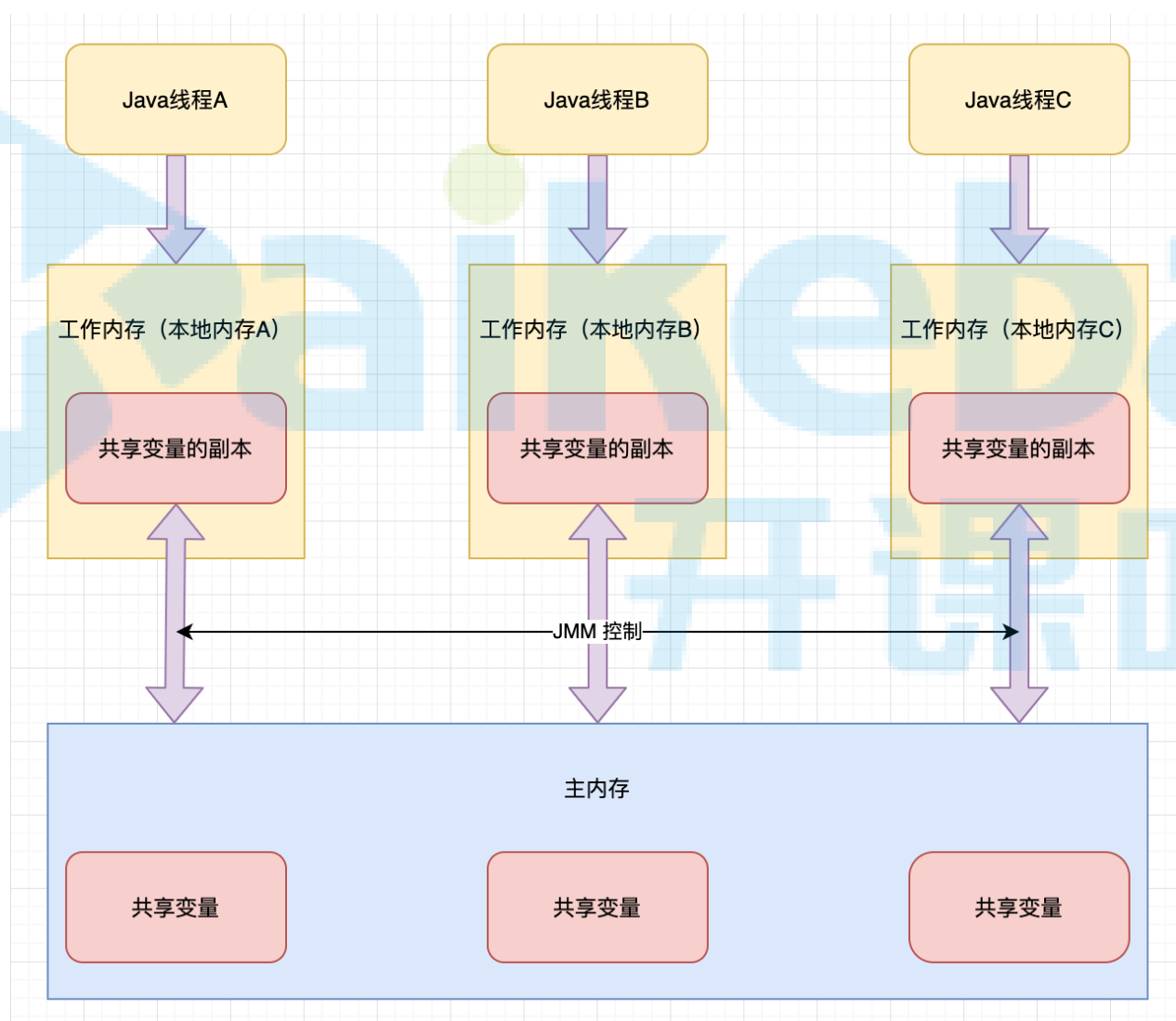
1. Lock接口
2. ReentrantLock 可重入锁
3. ReadWriteLock 读写锁

JMM (Java Memory Model)

JMM是指Java内存模型，不是JVM，不是所谓的栈、堆、方法区。

每个Java线程都有自己的工作内存。操作数据，首先从主内存中读，得到一份拷贝，操作完毕后再写回到主内存。

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存（有些地方成为栈空间），工作内存是每个线程的私有数据区域，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作（读取赋值等）必须在工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，各个线程中的工作内存中存储着主内存中的变量副本拷贝，因此不同的线程间无法访问对方的工作内存，线程间的通信（传值）必须通过主内存来完成，期简要访问过程如下图：



JMM可能带来可见性、原子性和有序性问题。

所谓可见性，就是某个线程对主内存内容的更改，应该立刻通知到其它线程。

所谓原子性，是指一个操作是不可分割的，不能执行到一半，就不执行了。

所谓有序性，就是指令是有序的，不会被重排。

volatile关键字



volatile 关键字是Java提供了一种轻量级同步机制。

- 它能够保证可见性和有序性
- 但是不能保证原子性
- 禁止指令重排

可见性

可见性测试

```
1 class MyData {
2     int number = 0;
3     //volatile int number = 0;
4
5     public void setTo60() {
6         this.number = 60;
7     }
8
9 }
10
11 public class VolatileDemo {
12     public static void main(String[] args) {
13         volatileVisibilityDemo();
14     }
15 }
```

```

15
16 //volatile可以保证可见性，及时通知其它线程主物理内存的值已被修改
17 private static void volatileVisibilityDemo() {
18     System.out.println("可见性测试");
19     MyData myData = new MyData(); //资源类
20     //启动一个线程操作共享数据
21     new Thread(() -> {
22         System.out.println(Thread.currentThread().getName() + "\t 执
行");
23         try {
24             TimeUnit.SECONDS.sleep(3);
25             myData.setTo60();
26             System.out.println(Thread.currentThread().getName() + "\t
更新number值: " + myData.number);
27         } catch (InterruptedException e) {
28             e.printStackTrace();
29         }
30     }, "ThreadA").start();
31     while (myData.number == 0) {
32         //main线程持有共享数据的拷贝，一直为0
33     }
34     System.out.println(Thread.currentThread().getName() + "\t main获取
number值: " + myData.number);
35 }
36 }

```

MyData 类是资源类，一开始number变量没有用volatile修饰，所以程序运行的结果是：

```

1 可见性测试
2 ThreadA  执行
3 ThreadA  更新number值: 60

```

虽然一个线程把number修改成了60，但是main线程持有的仍然是最开始的0，所以一直循环，程序不会结束。

如果对number添加了volatile修饰，运行结果是：

```

1 可见性测试
2 ThreadA  执行
3 ThreadA  更新number值: 60
4 main     main获取number值: 60

```

可见某个线程对number的修改，会立刻反映到主内存上。

原子性

原子性指的是什么意思？

不和分割，完整性，也即某个线程正则做某个具体业务时，中间不可以被加塞或者被分割。需要整体完整，要么同时成功，要么同时失败。

```
1  class MyData{
2      //int number=0;
3      volatile int number=0;
4
5      //此时number前面已经加了volatile，但是不保证原子性
6      public void addPlusPlus(){
7          number++;
8      }
9  }
10 public class VolatileDemo {
11     public static void main(String[] args) {
12         //volatileVisibilityDemo();
13         atomicDemo();
14     }
15
16     private static void atomicDemo() {
17         System.out.println("原子性测试");
18         MyData myData=new MyData();
19         for (int i = 1; i <= 20; i++) {
20             new Thread(()->{
21                 for (int j = 0; j <1000 ; j++) {
22                     myData.addPlusPlus();
23                 }
24                 },String.valueOf(i)).start();
25         }
26         while (Thread.activeCount(>2)){
27             Thread.yield();
28         }
29         System.out.println(Thread.currentThread().getName()+"\t int类型最终
30         number值: "+myData.number);
31     }
32 }
```

volatile并不能保证操作的原子性。这是因为，比如一条number++的操作，会形成3条指令。

```
1  javap -c 包名.类名
```

```

1  javap -c MyData
2
3  public void addPlusPlus();
4      Code:
5          0: aload_0
6          1: dup
7          2: getfield      #2                // Field number:I //读
8          5: iconst_1
9          6: iadd
10         7: putfield      #2                // Field number:I //写操作
11         10: return
12

```

假设有3个线程，分别执行number++，都先从主内存中拿到最开始的值，number=0，然后三个线程分别进行操作。假设线程0执行完毕，number=1，也立刻通知到了其它线程，但是此时线程1、2已经拿到了number=0，所以结果就是写覆盖，线程1、2将number变成1。

解决的方式就是：

1. 对 addPlusPlus() 方法加锁。
2. 使用 java.util.concurrent.AtomicInteger 类。

Constructors

Constructor and Description

AtomicInteger()

Creates a new AtomicInteger with initial value 0.

AtomicInteger(int initialValue)

Creates a new AtomicInteger with the given initial value.

getAndIncrement

public final int getAndIncrement()

Atomically increments by one the current value.

Returns:

the previous value

```

1  class MyData{
2      //int number=0;
3      volatile int number=0;
4
5      AtomicInteger atomicInteger=new AtomicInteger();
6
7      public void setTo60(){
8          this.number=60;
9      }
10
11      //此时number前面已经加了volatile，但是不保证原子性
12      public void addPlusPlus(){

```

```

13         number++;
14     }
15
16     public void addAtomic(){
17         atomicInteger.getAndIncrement();
18     }
19 }
20
21 public class VolatileDemo {
22     public static void main(String[] args) {
23         //volatileVisibilityDemo();
24         atomicDemo();
25     }
26
27     private static void atomicDemo() {
28         System.out.println("原子性测试");
29         MyData myData=new MyData();
30         for (int i = 1; i <= 20; i++) {
31             new Thread(()->{
32                 for (int j = 0; j <1000 ; j++) {
33                     myData.addPlusPlus();
34                     myData.addAtomic();
35                 }
36             },String.valueOf(i)).start();
37         }
38         while (Thread.activeCount(>2)){
39             Thread.yield();
40         }
41         System.out.println(Thread.currentThread().getName()+"\t int类型最终
number值: "+myData.number);
42         System.out.println(Thread.currentThread().getName()+"\t
AtomicInteger类型最终number值: "+myData.atomicInteger);
43     }
44 }

```

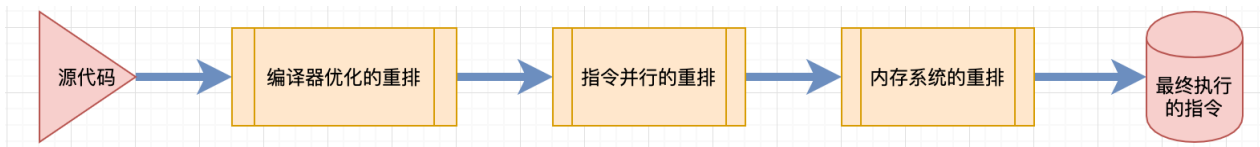
结果：可见，由于 `volatile` 不能保证原子性，出现了线程重复写的问题，最终结果比20000小。而 `AtomicInteger` 可以保证原子性。

1	原子性测试
2	main int类型最终number值: 17751
3	main AtomicInteger类型最终number值: 20000

有序性

[有序性案例](#)

计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排，一般分以下三种：



单线程环境里面确保程序最终执行结果和代码顺序执行的结果一致；

处理器在进行重排序时必须要考虑指令之间的数据依赖性；

多线程环境中线程交替执行，由于编译器优化重排的存在，两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。

volatile可以保证有序性，也就是防止指令重排序。

所谓指令重排序，就是出于优化考虑，CPU执行指令的顺序跟程序员自己编写的顺序不一致。就好比一份试卷，题号是老师规定的，是程序员规定的，但是考生（CPU）可以先做选择，也可以先做填空。

```
1  int x = 11; //语句1
2  int y = 12; //语句2
3  x = x + 5;  //语句3
4  y = x * x;  //语句4
```

以上例子，可能出现的执行顺序有1234、2134、1342，这三个都没有问题，最终结果都是x = 16, y=256。但是如果是4开头，就有问题了，y=0。这个时候就不需要指令重排序。

观看下面代码，在多线程场景下，说出最终值a的结果是多少？ 5或者6

我们采用 **volatile** 可实现禁止指令重排优化，从而避免多线程环境下程序出现乱序执行的现象

```
1  public class ResortSeqDemo {
2
3      int a=0;
4      boolean flag=false;
5      /*
6       多线程下flag=true可能先执行，还没走到a=1就被挂起。
7       其它线程进入method02的判断，修改a的值=5，而不是6。
8       */
9      public void method01(){
10         a=1;
11         flag=true;
12     }
13     public void method02(){
14         if (flag){
15             a+=5;
16             System.out.println("*****最终值a: "+a);
17         }
18     }
19
20     public static void main(String[] args) {
21         ResortSeqDemo resortSeq = new ResortSeqDemo();
22     }
```



```

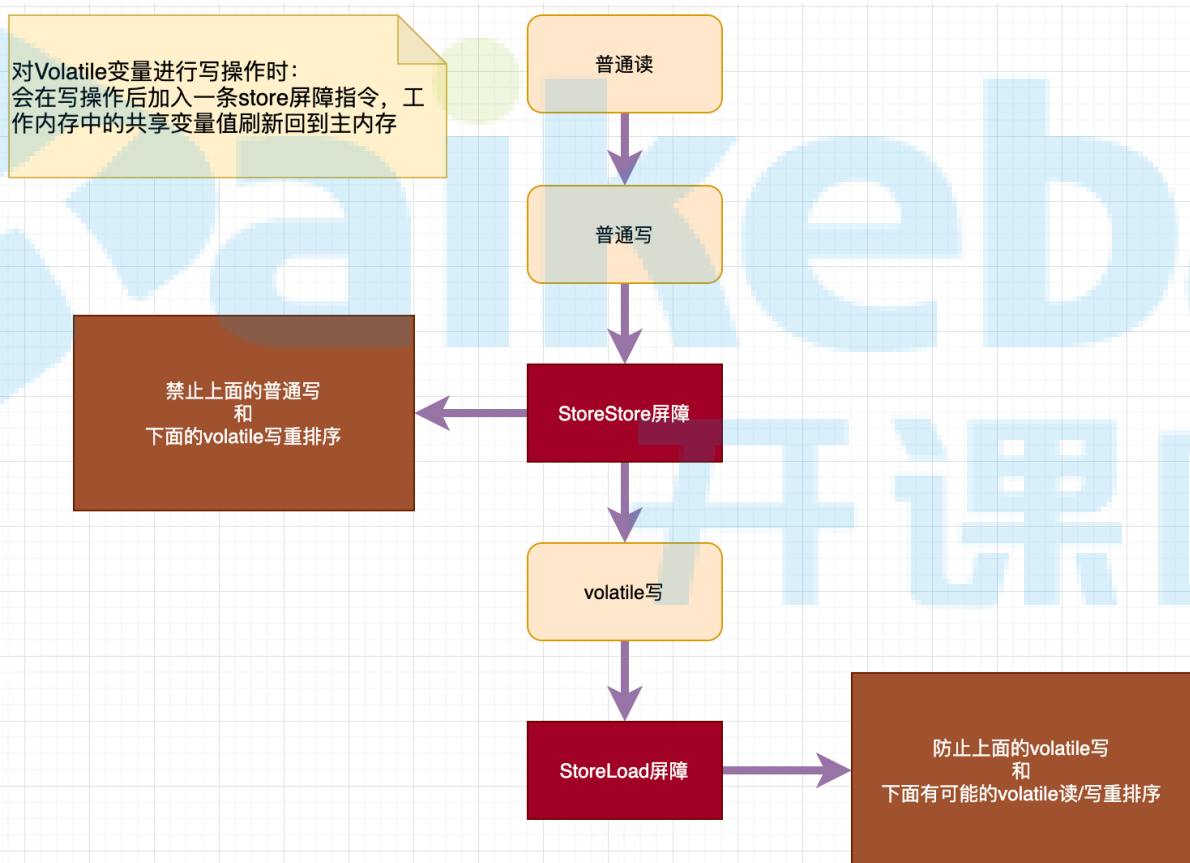
23     new Thread()->{resortSeq.method01();}, "ThreadA").start();
24     new Thread()->{resortSeq.method02();}, "ThreadB").start();
25 }
26 }

```

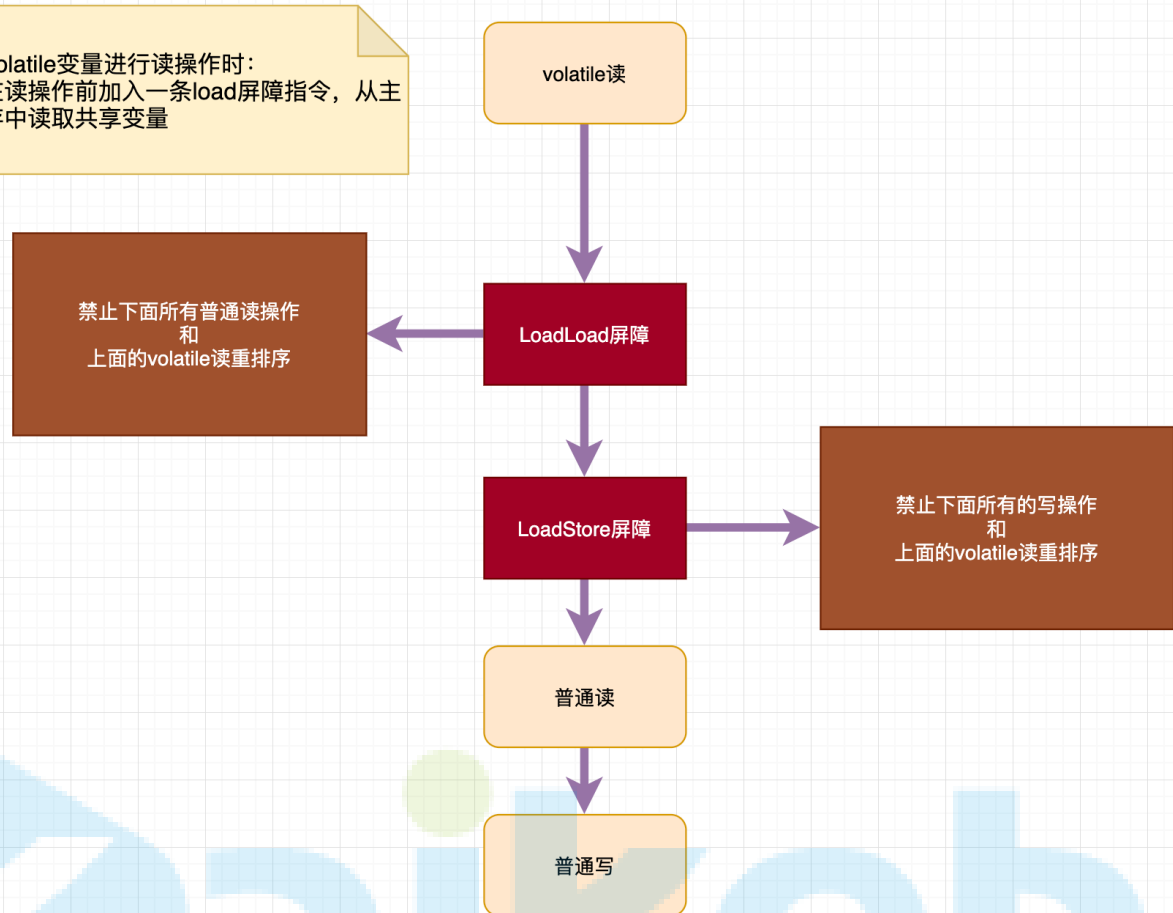
为什么volatile 可实现禁止指令重排优化，从而避免多线程环境下程序出现乱序执行的现象？说说它的原理

我们先来了解一个概念，**内存屏障**（Memory Barrier）又称内存栅栏，是一个CPU指令，volatile底层就是用CPU的**内存屏障**（Memory Barrier）指令来实现的，它有两个作用

- 一个是保证特定操作的顺序性
- 二是保证变量的可见性。



对Volatile变量进行读操作时：
会在读操作前加入一条load屏障指令，从主
内存中读取共享变量



由于编译器和处理器都能够执行指令重排优化。所以，如果在指令间插入一条Memory Barrier则会告诉编译器和CPU，不管什么指令都不能和这条Memory Barrier指令重排序，也就是说通过插入内存屏障可以禁止在内存屏障前后的指令进行重排序优化。内存屏障另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读到这些数据的最新版本。

哪些地方用到过volatile?

单例模式的安全问题

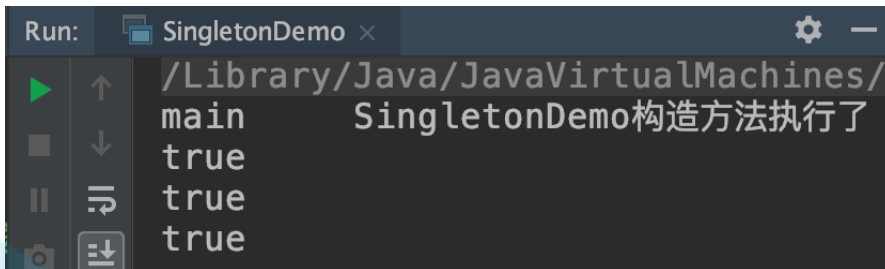
- 传统

```
1 public class SingletonDemo {
2     private static SingletonDemo instance = null;
3
4     private SingletonDemo() {
5         System.out.println(Thread.currentThread().getName() + "\t"
6         SingletonDemo构造方法执行了");
7     }
8
9     public static SingletonDemo getInstance(){
10         if (instance == null) {
11             instance = new SingletonDemo();
12         }
13         return instance;
14     }
15 }
```

```

14
15     public static void main(String[] args) {
16         //main线程操作
17         System.out.println(SingletonDemo.getInstance() ==
SingletonDemo.getInstance());
18         System.out.println(SingletonDemo.getInstance() ==
SingletonDemo.getInstance());
19         System.out.println(SingletonDemo.getInstance() ==
SingletonDemo.getInstance());
20     }
21 }

```



```

Run: SingletonDemo x
/Library/Java/JavaVirtualMachines/
main SingletonDemo构造方法执行了
true
true
true

```

- 改为多线程操作测试

```

1     public class SingletonDemo {
2         private static SingletonDemo instance = null;
3
4         private SingletonDemo() {
5             System.out.println(Thread.currentThread().getName() + "\t"
SingletonDemo构造方法执行了");
6         }
7
8         public static SingletonDemo getInstance(){
9             if (instance == null) {
10                 instance = new SingletonDemo();
11             }
12             return instance;
13         }
14
15         public static void main(String[] args) {
16             //多线程操作
17             for (int i = 0; i < 10; i++) {
18                 new Thread()->{
19                     SingletonDemo.getInstance();
20                 }, Thread.currentThread().getName()).start();
21             }
22         }
23     }
24 }

```

```
Run: SingletonDemo x
/Library/Java/JavaVirtualMachines/
main SingletonDemo构造方法执行了
main SingletonDemo构造方法执行了
main SingletonDemo构造方法执行了
```

- 调整后，采用常见的DCL（Double Check Lock）双端检查模式加了同步，但是在多线程下依然会有线程安全问题。

```
1 public class SingletonDemo {
2     private static SingletonDemo instance = null;
3
4     private SingletonDemo() {
5         System.out.println(Thread.currentThread().getName() + "\t"
6             SingletonDemo构造方法执行了");
7     }
8
9     public static SingletonDemo getInstance(){
10         if (instance == null) {
11             synchronized (SingletonDemo.class){
12                 if (instance == null) {
13                     instance = new SingletonDemo();
14                 }
15             }
16         }
17         return instance;
18     }
19
20     public static void main(String[] args) {
21         //多线程操作
22         for (int i = 0; i < 10; i++) {
23             new Thread()->{
24                 SingletonDemo.getInstance();
25             },Thread.currentThread().getName()).start();
26         }
27     }
28 }
```

```
Run: SingletonDemo x
/Library/Java/JavaVirtualMachines/
main SingletonDemo构造方法执行了
Process finished with exit code 0
```

这个漏洞比较tricky，很难捕捉，但是是存在的。 `instance=new SingletonDemo();` 可以大致分为三步

```
1  instance = new SingletonDemo();
2
3  public static thread.SingletonDemo getInstance();
4      Code:
5          0: getstatic      #11          // Field instance:Lthread/SingletonDemo;
6          3: ifnonnull        37
7          6: ldc              #12          // class thread/SingletonDemo
8          8: dup
9          9: astore_0
10         10: monitorenter
11         11: getstatic      #11          // Field instance:Lthread/SingletonDemo;
12         14: ifnonnull        27
13         17: new            #12          // class thread/SingletonDemo 步骤1
14         20: dup
15         21: invokespecial  #13          // Method "<init>":()V 步骤2
16         24: putstatic      #11          // Field instance:Lthread/SingletonDemo;步
    骤3
17
18  底层Java Native Interface中的C语言代码内容，开辟空间的步骤
19  memory = allocate();      //步骤1.分配对象内存空间
20  instance(memory);          //步骤2.初始化对象
21  instance = memory;         //步骤3.设置instance指向刚分配的内存地址，此时instance
    != null
```

剖析：

在多线程的环境下，由于有指令重排序的存在，DCL（双端检锁）机制不一定线程安全，我们可以加入 `volatile` 可以禁止指令重排。

原因在与某一个线程执行到第一次检测，读取到的 `instance` 不为 `null` 时，`instance` 的引用对象可能没有完成初始化。

```
1  memory = allocate();      //步骤1. 分配对象内存空间
2  instance(memory);          //步骤2.初始化对象
3  instance = memory;         //步骤3.设置instance指向刚分配的内存地址，此时instance !=
    null
```

步骤2和步骤3不存在数据依赖关系，而且无论重排前还是重排后，程序的执行结果在单线程中并没有改变，因此这种重排优化是允许的。

```
1  memory = allocate();      //步骤1. 分配对象内存空间
2  instance = memory;         //步骤3.设置instance指向刚分配的内存地址，此时instance !=
    null，但是对象还没有初始化完成！
3  instance(memory);          //步骤2.初始化对象
```

但是指令重排只会保证串行语义的执行一致性（单线程），并不关心多线程的语义一致性。所以，当一条线程访问instance不为null时，由于instance实例未必已初始化完成，也就造成了线程安全问题。

```
1 public static SingletonDemo getInstance(){
2     if (instance == null) {
3         synchronized (SingletonDemo.class){
4             if (instance == null) {
5                 instance = new SingletonDemo(); //多线程情况下，可能发生指令重
排
6             }
7         }
8     }
9     return instance;
10 }
```

如果发生指定重排，那么，

1. 此时内存已经分配，那么 `instance=memory` 不为null。
2. 碰巧，若遇到线程此时挂起，那么 `instance(memory)` 还未执行，对象还未初始化。
3. 导致了 `instance!=null`，所以两次判断都跳过，最后返回的 `instance`` 没有任何内容，还没初始化。

解决的方法就是对 `singletondemo` 对象添加上 `volatile` 关键字，禁止指令重排。

```
1 private static volatile SingletonDemo singletonDemo=null;
```

CAS

开课吧

2.CAS

3.ABA问

1. 比较并交换

2. CAS底层原理？谈谈对Unsafe的理解？为什么不用synchronized也能实现++操作？

AtomicInteger.getAndIncrement源码

sun.misc.Unsafe类

Unsafe.getAndAddInt

CAS

底层汇编

简单总结

3. CAS的缺点？

循环时间长，开销大

只能保证一个共享变量的原子操作

会引出ABA问题

看下面代码进行思考，此时number前面是加了volatile关键字修饰的，volatile不保证原子性，那么使用AtomicInteger是如何保证原子性的？这里的原理是什么？CAS

```
1 class MyData {
2
3     volatile int number = 0;
4     AtomicInteger atomicInteger=new AtomicInteger();
5
6     public void addPlusPlus(){
7         number++;
8     }
9
10    public void addAtomic(){
11        atomicInteger.getAndIncrement();
12    }
13
14    public void setTo60() {
15        this.number = 60;
16    }
17 }
```

CAS的全称为Compare-And-Swap，比较并交换，是一种很重要的同步思想。它是一条CPU并发原语。

它的功能是判断主内存某个位置的值是否为跟期望值一样，相同就进行修改，否则一直重试，直到一致为止。这个过程是原子的。

看下面这段代码，思考运行结果是

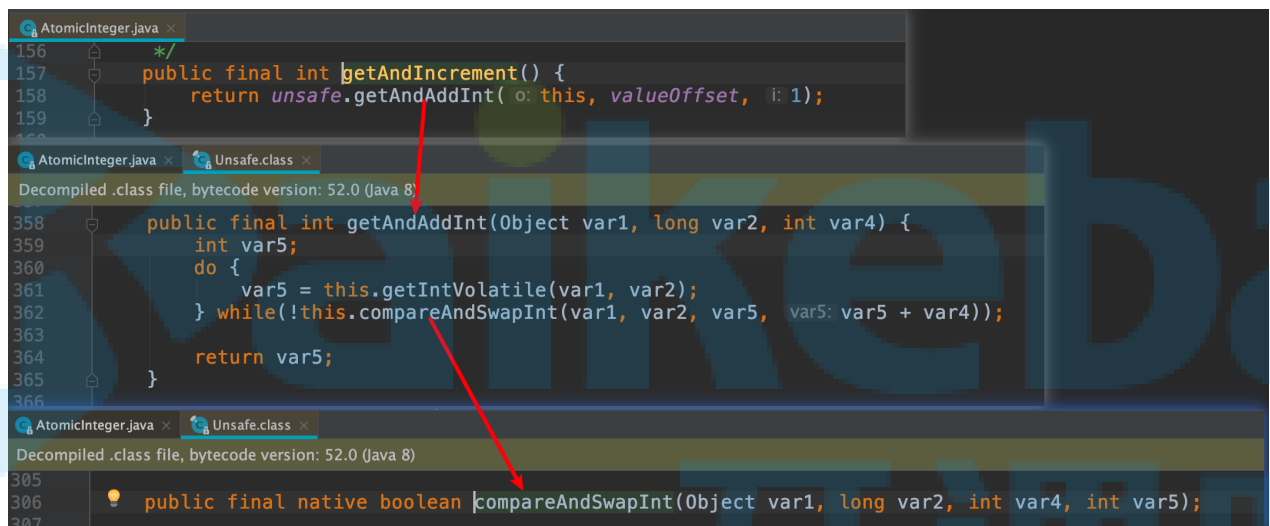
```

1 public class CASDemo {
2     public static void main(String[] args) {
3         AtomicInteger atomicInteger=new AtomicInteger(5);
4         System.out.println(atomicInteger.compareAndSet(5, 2020)+"\t 当前数据
值 : "+ atomicInteger.get());
5         //修改失败
6         System.out.println(atomicInteger.compareAndSet(5, 1024)+"\t 当前数据
值 : "+ atomicInteger.get());
7     }
8 }

```

第一次修改，期望值为5，主内存也为5，修改成功，为2020。第二次修改，期望值为5，主内存为2020，修改失败，需要重新获取主内存的值。

查看 `AtomicInteger.getAndIncrement()` 方法，发现其没有加 `synchronized` 也实现了同步。这是为什么？



CAS并发原语体现在JAVA语言中就是`sun.misc.Unsafe`类中的各个方法。看方法源码，调用`Unsafe`类中的CAS方法，JVM会帮我们实现出CAS汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。再次强调，由于CAS是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行是连续的，在执行过程中不允许被中断，也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题。

CAS底层原理


```

public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
}

```

AtomicInteger内部的重要参数

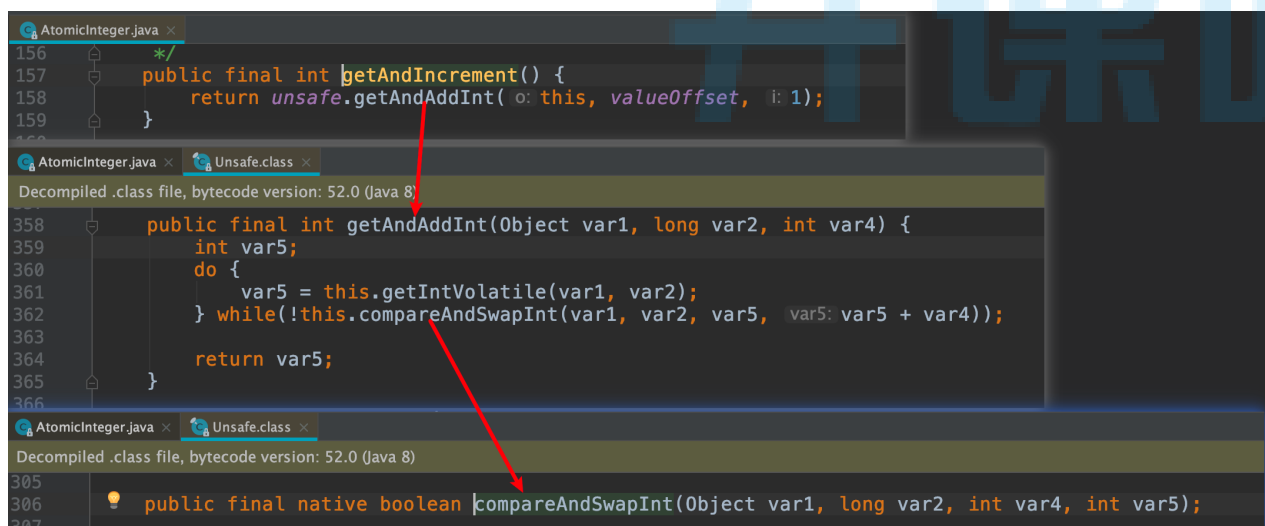
1. Unsafe

是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地（native）方法来访问，Unsafe相当于一个后面，基于该类可以直接操作特定内存的数据。Unsafe类存在于sun.misc包中，其内部方法操作可以像C的指针一样直接操作内存，因为Java中CAS操作的执行依赖于Unsafe类的方法。

注意Unsafe类中的所有方法都是native修饰的，也就是说Unsafe类中的方法都直接调用操作系统底层资源执行相应任务

2. 变量valueOffset，表示该变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的。
3. 变量value用volatile修饰，保证了多线程之间的内存可见性。

AtomicInteger.getAndIncrement() 调用了 Unsafe.getAndAddInt() 方法。Unsafe 类的大部分方法都是 native 的，用来像C语言一样从底层操作内存。



C语句代码JNI，对应java方法 `public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5)`

```

1 UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject
  unsafe, jlong obj, jlong offset, jint e, jint x))
2     UnsafeWrapper("Unsafe_CompareAndSwapInt");
3     oop p = JNIHandles::resolve(obj);
4     jint* addr = (jint *)index_oop_from_field_offset_long(p, offset);
5     return (jint)(Atomic::cmpxchg(x, addr, e) == e);
6 UNSAFE_END
7
8 //先想办法拿到变量value在内存中的地址addr。
9 //通过Atomic::cmpxchg实现比较替换，其中参数x是即将更新的值，参数e是原内存的值。

```

这个方法的var1和var2，就是根据对象和偏移量得到在主内存的快照值var5。然后 `compareAndSwapInt` 方法通过var1和var2得到当前主内存的实际值。如果这个实际值跟快照值相等，那么就更新主内存的值为var5+var4。如果不等，那么就一直循环，一直获取快照，一直对比，直到实际值和快照值相等为止。

参数介绍

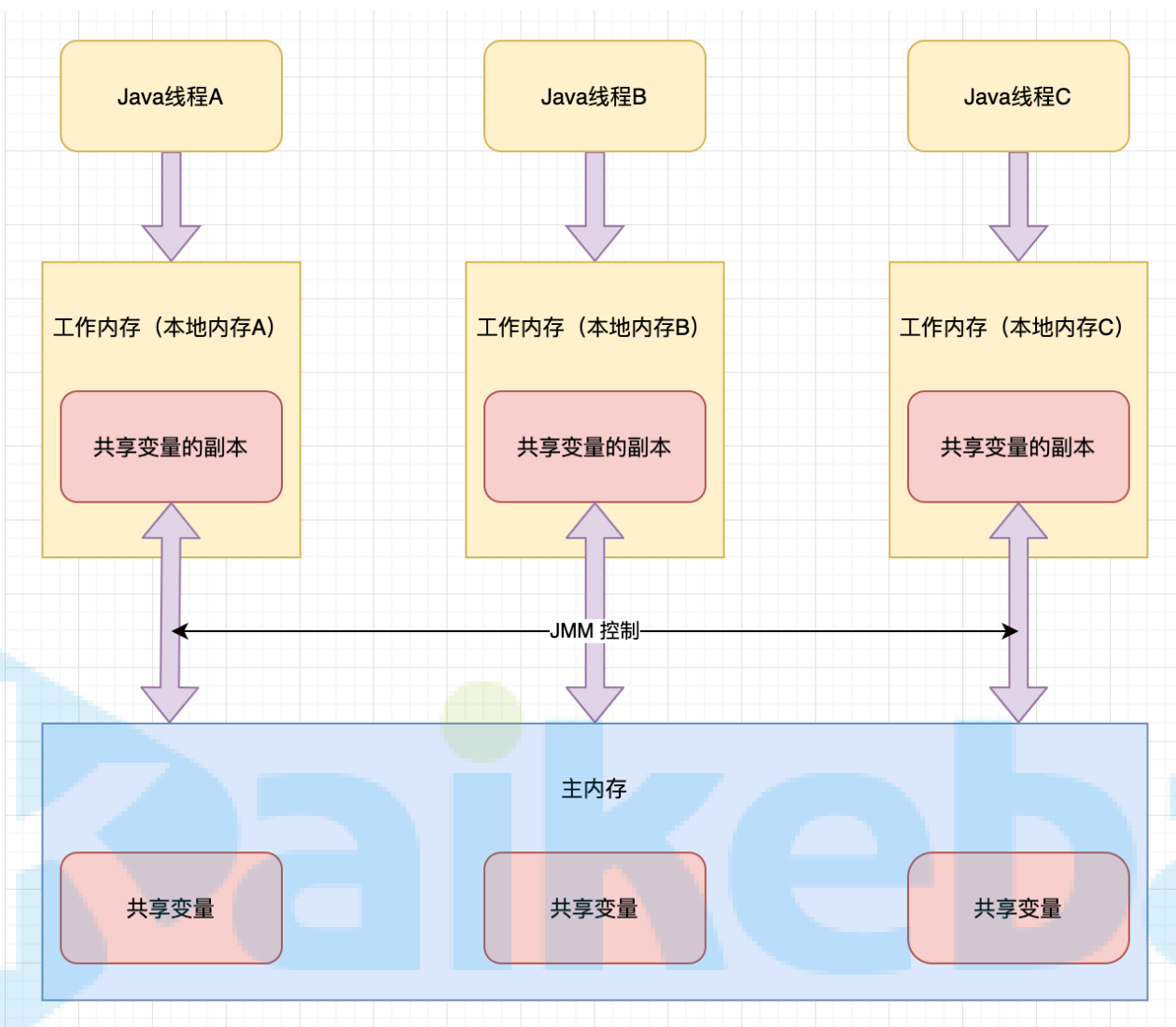
var1 AtomicInteger对象本身

var2 该对象值的引用地址

var4 需要变动的数量

var5 是通过var1和var2，根据对象和偏移量得到在主内存的快照值var5

比如有A、B两个线程，一开始都从主内存中拷贝了原值为3，A线程执行到 `var5=this.getIntVolatile`，即var5=3。此时A线程挂起，B修改原值为4，B线程执行完毕，由于加了volatile，所以这个修改是立即可见的。A线程被唤醒，执行 `this.compareAndSwapInt()` 方法，发现这个时候主内存的值不等于快照值3，所以继续循环，重新从主内存获取。



CAS缺点

```
Unsafe.class x
Decompiled .class file, bytecode version: 52.0 (Java 8)
358     public final int getAndAddInt(Object var1, long var2, int var4) {
359         int var5;
360         do {
361             var5 = this.getIntVolatile(var1, var2);
362         } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));
363
364         return var5;
365     }
```

CAS实际上是一种自旋锁，

1. 一直循环，开销比较大。我们可以看到getAndAddInt方法执行时，有个do while，如果CAS失败，会一直进行尝试。如果CAS长时间一直不成功，可能会给CPU带来很大的开销。
2. 对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是，对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁来保证原子性。
3. 引出了ABA问题。

CAS会导致"ABA问题"

高频面试题

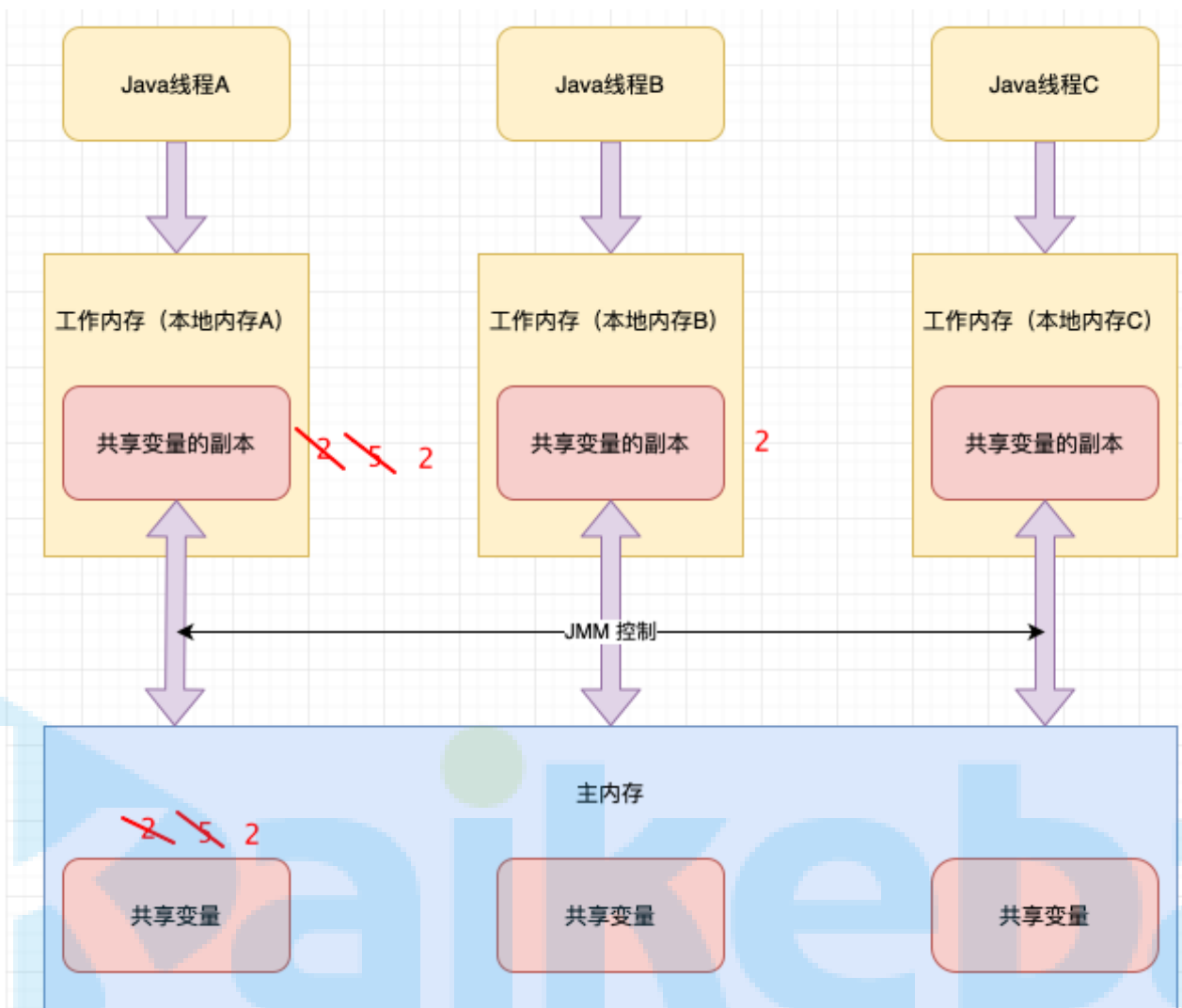
1. 原子类AtomicInteger的ABA问题谈谈？原子更新引用你知道吗？
2. 我们知道ArrayList是线程不安全，请编码写一个不安全的案例并给出解决方案
3. 公平锁/非公平锁/可重入锁/递归锁/自旋锁谈谈你的理解？请手写一个自旋锁
4. CountDownLatch/CyclicBarrier/Semaphore使用过吗？
5. 阻塞队列知道吗？
6. 线程池用过吗？ThreadPoolExecutor谈谈你的理解？生产上你如何设置合理参数
7. 死锁编码及定位分析

所谓ABA问题，就是CAS算法实现需要取出内存中某时刻的数据并在当下时刻比较并替换，这里存在一个时间差，那么这个时间差可能带来意想不到的问题。

比如，一个线程B从内存位置Value中取出2，这时候另一个线程A也从内存位置Value中取出2，并且线程A进行了一些操作将值变成了5，然后线程A又再次将值变成了2，这时候线程B进行CAS操作发现内存中仍然是2，然后线程B操作成功。

尽管线程B的CAS操作成功，但是不代表这个过程就是没有问题的。

开课吧



有这样的需求，比如CAS，只注重头和尾，只要首尾一致就接受。

但是有的需求，还看重过程，中间不能发生任何修改，这就引出了 `AtomicReference` 原子引用。

AtomicReference原子引用

`AtomicInteger` 对整数进行原子操作，如果是一个POJO呢？可以用 `AtomicReference` 来包装这个POJO，使其操作原子化。

```

1 public class AtomicReferenceDemo {
2     public static void main(String[] args) {
3         User user1 = new User("Jack",25);
4         User user2 = new User("Tom",21);
5
6         AtomicReference<User> atomicReference = new AtomicReference<>();
7
8         atomicReference.set(user1);
9
10
11         System.out.println(atomicReference.compareAndSet(user1,user2)+"\t"+atomic
Reference.get()); // true
12
13         System.out.println(atomicReference.compareAndSet(user1,user2)+"\t"+atomic
Reference.get()); //false
14     }
15 }

```

```

Run: AtomicReferenceDemo x
/Library/Java/JavaVirtualMachines/jdk1
true   User(username=Tom, age=21)
false  User(username=Tom, age=21)

```

ABA问题的解决(AtomicStampedReference 类似于时间戳)

1	ThreadA	100	1	2020	2
2					
3	ThreadB	100	1	111	2
				100	3

使用 `AtomicStampedReference` 类可以解决ABA问题。这个类维护了一个“版本号”Stamp，在进行CAS操作的时候，不仅要比较当前值，还要比较版本号。只有两者都相等，才执行更新操作。

解决ABA问题的关键方法：

```

132
133  /**
134   * Atomically sets the value of both the reference and stamp
135   * to the given update values if the
136   * current reference is {@code ==} to the expected reference
137   * and the current stamp is equal to the expected stamp.
138   *
139   * @param expectedReference the expected value of the reference
140   * @param newReference the new value for the reference
141   * @param expectedStamp the expected value of the stamp
142   * @param newStamp the new value for the stamp
143   * @return {@code true} if successful
144   */
145  @ public boolean compareAndSet(V expectedReference,
146                                V newReference,
147                                int expectedStamp,
148                                int newStamp) {
149      Pair<V> current = pair;
150      return
151          expectedReference == current.reference &&
152          expectedStamp == current.stamp &&
153          ((newReference == current.reference &&
154            newStamp == current.stamp) ||
155          casPair(current, Pair.of(newReference, newStamp)));
156  }
157

```

参数说明:

1	V	expectedReference,	预期值引用
2	V	newReference,	新值引用
3	int	expectedStamp,	预期值时间戳
4	int	newStamp,	新值时间戳

```

1  public class ABADemo {
2      static AtomicReference<Integer> atomicReference = new
AtomicReference<>(100);
3      static AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(100, 1);
4
5      public static void main(String[] args) {
6          System.out.println("=====ABA问题的产生=====");
7
8          new Thread(() -> {
9              atomicReference.compareAndSet(100, 101);
10             atomicReference.compareAndSet(101, 100);
11         }, "t1").start();
12
13         new Thread(() -> {
14             try {
15                 TimeUnit.SECONDS.sleep(1);
16             } catch (InterruptedException e) {
17                 e.printStackTrace();
18             }
19         }

```

```

20         System.out.println(atomicReference.compareAndSet(100, 2020) +
"\t" + atomicReference.get().toString());
21     }, "t2").start();
22
23     try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e)
{ e.printStackTrace(); }
24
25     System.out.println("=====ABA问题的解决=====");
26     new Thread(() -> {
27         int stamp = atomicStampedReference.getStamp();
28         System.out.println(Thread.currentThread().getName() + "\t第一次
版本号: " + stamp);
29
30         try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException
e) { e.printStackTrace(); }
31         atomicStampedReference.compareAndSet(100,101,
32
atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);
33         System.out.println(Thread.currentThread().getName() + "\t第二次
版本号: " + atomicStampedReference.getStamp());
34
35         atomicStampedReference.compareAndSet(101,100,
36
atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);
37         System.out.println(Thread.currentThread().getName() + "\t第三次
版本号: " + atomicStampedReference.getStamp());
38     }, "t3").start();
39
40     new Thread(() -> {
41         int stamp = atomicStampedReference.getStamp();
42         System.out.println(Thread.currentThread().getName() + "\t第一次
版本号: " + stamp);
43
44         try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException
e) { e.printStackTrace(); }
45         boolean result=atomicStampedReference.compareAndSet(100,2020,
46             stamp,stamp+1);
47         System.out.println(Thread.currentThread().getName()+"\t修改成功
与否: "+result+" 当前最新版本号"+atomicStampedReference.getStamp());
48         System.out.println(Thread.currentThread().getName()+"\t当前实际
值: "+atomicStampedReference.getReference());
49     }, "t4").start();
50     }
51 }

```



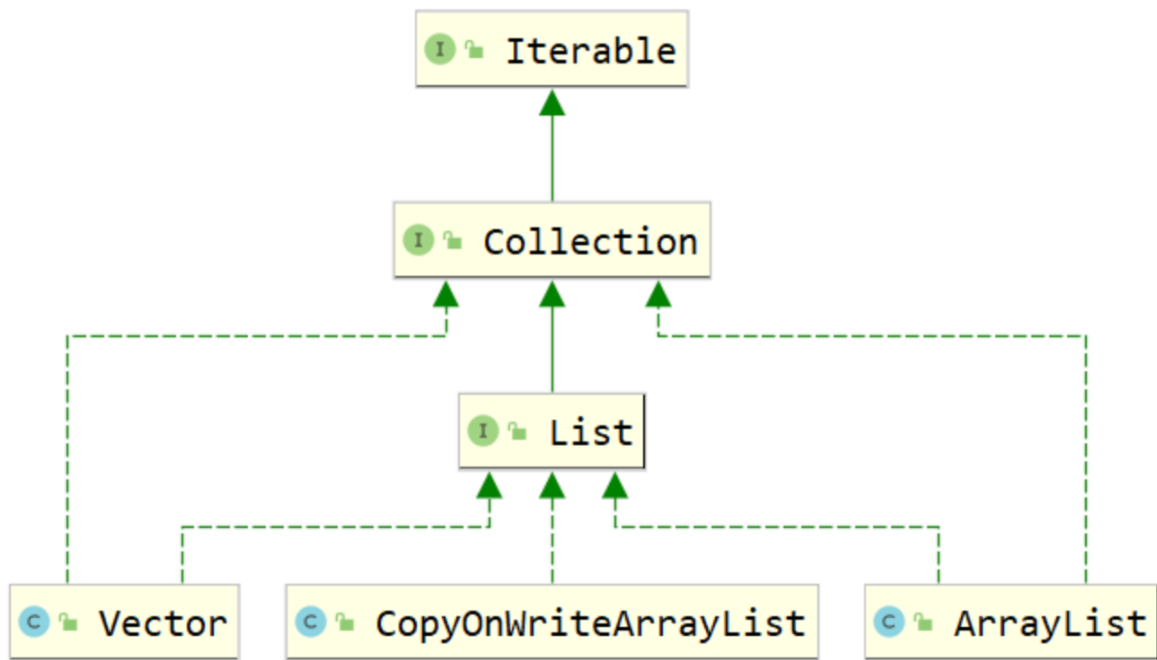
```
Run: ABADemo x
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.
=====ABA问题的产生=====
true      2020
=====ABA问题的解决=====
t3 第一次版本号: 1
t4 第一次版本号: 1
t3 第二次版本号: 2
t3 第三次版本号: 3
t4 修改成功与否: false 当前最新版本号3
t4 当前实际值: 100
```

集合类不安全问题

ArrayList与CopyOnWriteArrayList

ArrayList 不是线程安全类，在多线程同时写的情况下，会抛出
java.util.ConcurrentModificationException 异常。

```
1 private static void listNotSafe() {
2     //List<String> list=new ArrayList<>();
3     List<String> list = new CopyOnWriteArrayList<>();
4     for (int i = 1; i <= 30; i++) {
5         new Thread(() -> {
6             list.add(UUID.randomUUID().toString().substring(0, 8));
7             System.out.println(Thread.currentThread().getName() + "\t" +
list);
8         }, String.valueOf(i)).start();
9     }
10 }
```



<https://blog.csdn.net/lizongxia>

解决方法:

1. 使用 `Vector` (`ArrayList` 所有方法加 `synchronized`, 太重)。
2. 使用 `Collections.synchronizedList()` 转换成线程安全类。
3. 使用 `java.concurrent.CopyOnWriteArrayList` (推荐)。

CopyOnWriteArrayList

这是JUC的类, 通过写时复制来实现读写分离。比如其 `add()` 方法, 就是先复制一个新数组, 长度为原数组长度+1, 然后将新数组最后一个元素设为添加的元素。

```
1 public boolean add(E e) {
2     final ReentrantLock lock = this.lock;
3     lock.lock();
4     try {
5         //得到旧数组
6         Object[] elements = getArray();
7         int len = elements.length;
8         //复制新数组
9         Object[] newElements = Arrays.copyOf(elements, len + 1);
10        //设置新元素
11        newElements[len] = e;
12        //设置新数组
13        setArray(newElements);
14        return true;
15    } finally {
16        lock.unlock();
17    }
18 }
```

写时复制：

CopyOnWrite容器即写时复制的容器。

原理：

往一个容器添加元素的时候，不直接往当前容器Object[]添加，而是现将当前容器Object[]进Copy，

复制出一个新的容器Object[] newElements，然后新的容器Object[] newElements里添加元素，添加完元素之后，

再将原容器的引用指向新的容器setArray(newElements);。这样做的好处是可以对CopyOnWrite容器进行并发的读，

而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器

Set与CopyOnWriteArraySet

```
1 private static void setNoSafe() {
2     //Set<String> set=new HashSet<>();
3     Set<String> set = new CopyOnWriteArraySet<>();
4     for (int i = 1; i <= 30; i++) {
5         new Thread(() -> {
6             set.add(UUID.randomUUID().toString().substring(0, 8));
7             System.out.println(Thread.currentThread().getName() + "\t" +
8                 set);
9             }, String.valueOf(i)).start();
10    }
```

HashSet和HashMap

HashSet底层是用HashMap实现的。既然是用HashMap实现的，那HashMap.put()需要传两个参数，而HashSet.add()只传一个参数，这是为什么？实际上HashSet.add()就是调用的HashMap.put()，只不过Value被写死了，是一个private static final Object对象。

跟List类似，HashSet和TreeSet都不是线程安全的，与之对应的有CopyOnWriteSet这个线程安全类。这个类底层维护了一个CopyOnWriteArrayList数组。

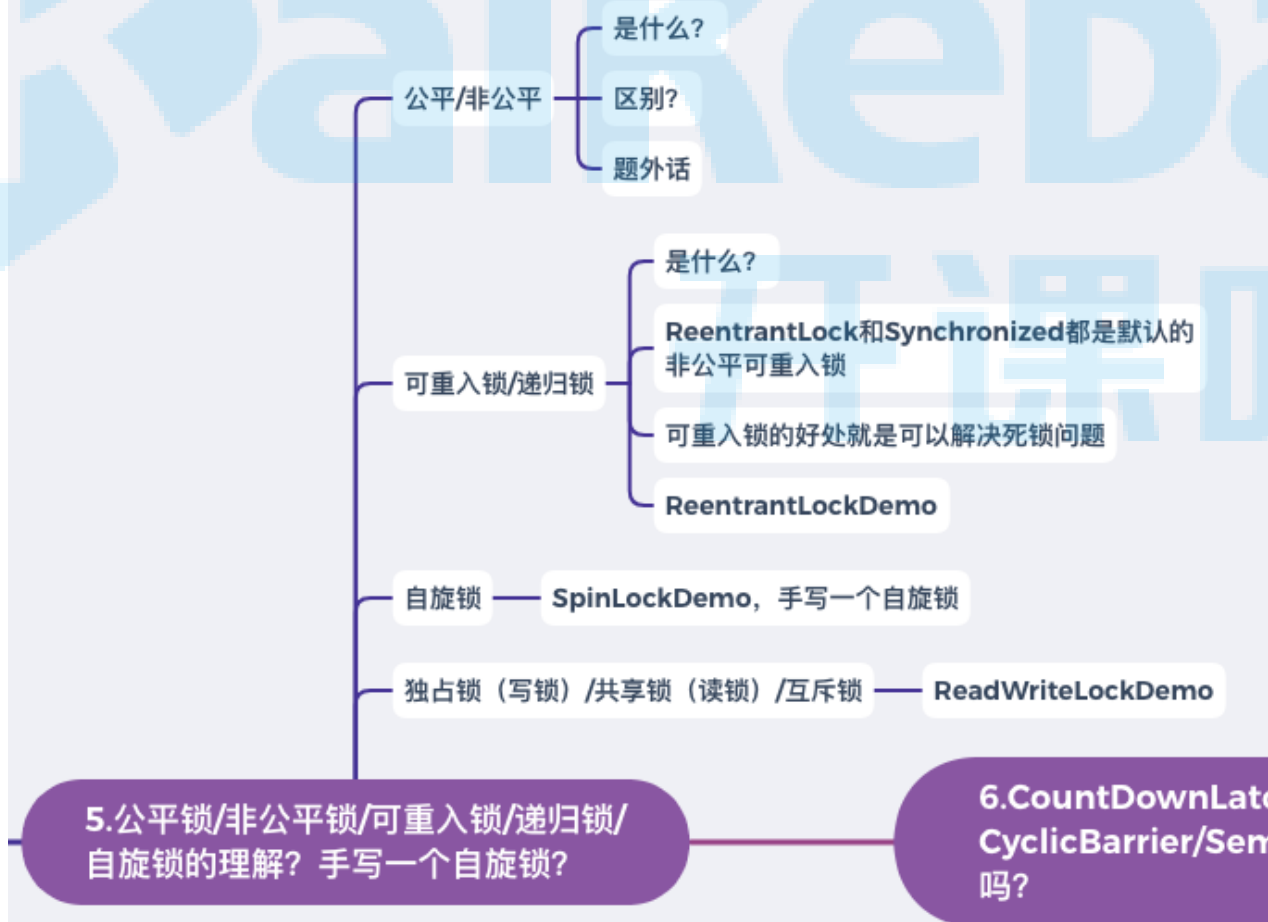
```
1 private final CopyOnWriteArrayList<E> al;
2 public CopyOnWriteArraySet() {
3     al = new CopyOnWriteArrayList<E>();
4 }
```

Map与ConcurrentHashMap

HashMap 不是线程安全的, Hashtable 是线程安全的, 但是跟 Vector 类似, 太重量级。所以也有类似 CopyOnWriteMap, 只不过叫 ConcurrentHashMap。

```
1 private static void mapNotSafe() {
2     //Map<String,String> map=new HashMap<>();
3     Map<String, String> map = new ConcurrentHashMap<>();
4     for (int i = 1; i <= 30; i++) {
5         new Thread(() -> {
6             map.put(Thread.currentThread().getName(),
7                 UUID.randomUUID().toString().substring(0, 8));
8             System.out.println(Thread.currentThread().getName() + "\t" +
9                 map);
10        }, String.valueOf(i)).start();
11    }
12 }
```

Java锁



多线程8锁

- 题目:

- 1 1.标准访问, 请问先打印邮件还是短信?
- 2 2.邮件方法暂停4秒钟, 请问先打印邮件还是短信?
- 3 3.新增一个普通方法hello (), 请问先打印邮件还是hello?
- 4 4.两部手机, 请问先打印邮件还是短信?
- 5 5.两个静态同步方法, 同一部手机, 请问先打印邮件还是短信?
- 6 6.两个静态同步方法, 2部手机, 请问先打印邮件还是短信?
- 7 7.1个普通同步方法, 1个静态同步方法, 1部手机, 请问先打印邮件还是短信?
- 8 8.1个普通同步方法, 1个静态同步方法, 2部手机, 请问先打印邮件还有短信?

● 解析:

```
1 class Phone{
2     public static synchronized void sendEmail(){
3         try { TimeUnit.SECONDS.sleep(4); } catch (InterruptedException e)
4         {e.printStackTrace(); }
5         System.out.println("=====sendEmail");
6     }
7
8     public synchronized void sendMessage(){
9         System.out.println("=====sendMessage");
10    }
11
12    public void hello(){
13        System.out.println("sayHello");
14    }
15 }
```

```
16 /**
17  * 1. 标准访问, 请问先打印邮件还是短信? 邮件
18  * 2. 邮件方法暂停4秒钟, 请问先打印邮件还是短信? 邮件
19  *     对象锁
20  *     一个对象里面如果有多个synchronized方法, 某一个时刻内, 只要一个线程去调用其中
21  *     的一个synchronized方法了,
22  *     其他的线程都只能等待, 换句话说, 某一个时刻内, 只能有唯一一个线程去访问这些
23  *     synchronized方法,
24  *     锁的是当前对象this, 被锁定后, 其他的线程都不能进入到当前对象的其他的
25  *     synchronized方法
26  * 3. 新增一个普通方法hello (), 请问先打印邮件还是hello? hello
27  *     加个普通方法后发现和同步锁无关
28  * 4. 两部手机, 请问先打印邮件还是短信? hello
29  *     换成两个对象后, 不是同一把锁了, 情况立刻变化
30  * 5. 两个静态同步方法, 同一部手机, 请问先打印邮件还是短信? 邮件
31  * 6. 两个静态同步方法, 2部手机, 请问先打印邮件还是短信? 邮件
32  *     全局锁
33  *     synchronized实现同步的基础: java中的每一个对象都可以作为锁。
34  *     具体表现为一下3中形式。
35  *     对于普通同步方法, 锁是当前实例对象, 锁的是当前对象this,
36  *     对于同步方法块, 锁的是synchronized括号里配置的对象。
```

```

34  *      对于静态同步方法，锁是当前类的class对象
35  *      7. 1个普通同步方法，1个静态同步方法，1部手机，请问先打印邮件还是短信？ 短信
36  *      8. 1个普通同步方法，1个静态同步方法，2部手机，请问先打印邮件还是短信？ 短信
37  *      当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。
38  *      也就是说如果一个实例对象的普通同步方法获取锁后，该实例对象的其他普通同步方法必须等待获取锁的方法释放锁后才能获取锁，
39  *      可是别的实例对象的普通同步方法因为跟该实例对象的普通同步方法用的是不同的锁，
40  *      所以无需等待该实例对象已获取锁的普通同步方法释放锁就可以获取他们自己的锁。
41  *
42  *      所有的静态同步方法用的也是同一把锁--类对象本身，
43  *      这两把锁(this/class)是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有静态条件的。
44  *      但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，
45  *      而不管是同一个实例对象的静态同步方法之间，
46  *      还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象
47  */
48  public class LockDemo {
49      public static void main(String[] args) {
50          Phone phone = new Phone();
51          Phone phone2 = new Phone();
52
53          new Thread()->{
54              phone.sendEmail();
55          }, "A").start();
56
57          try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e)
58          {e.printStackTrace(); }
59
60          new Thread()->{
61              phone2.sendMessage();
62          }, "B").start();
63
64          try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e)
65          {e.printStackTrace(); }
66
67          /*
68          new Thread()->{
69              phone.hello();
70          }, "C").start();
71          */
72      }
73  }

```

人工窗口排队购票(回顾)

```

2  * 题目：三个售票员  卖出   30张票
3  *
4  */
5  class Ticket{//资源类
6      //票
7      private int number = 30;
8
9      public synchronized void saleTicket(){
10         if (number > 0) {
11             System.out.println(Thread.currentThread().getName()+"\t卖出
第: "+(number--)+"\t还剩下: "+number);
12         }
13     }
14 }
15
16 public class SaleTicketDemo {
17     public static void main(String[] args) {
18         Ticket ticket = new Ticket();
19
20         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "A").start();
21         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "B").start();
22         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "C").start();
23     }
24 }

```

公平锁非公平锁（火车站人工窗口排队购票）

```

1  Lock lock = new ReentrantLock(true);

```

```

/**
 * Creates an instance of {@code ReentrantLock}.
 * This is equivalent to using {@code ReentrantLock(false)}.
 */
public ReentrantLock() {
    sync = new NonfairSync();
}

```

```

/**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy
 */
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

```

概念：所谓公平锁，就是多个线程按照申请锁的顺序来获取锁，类似排队，先到先得。而非公平锁，则是多个线程抢夺锁，会导致优先级反转或饥饿现象。

区别：

- 公平锁在获取锁时先查看此锁维护的等待队列，为空或者当前线程是等待队列的队首，则直接占有锁，否则插入到等待队列，FIFO原则。
- 非公平锁比较粗鲁，上来直接先尝试占有锁，失败则采用公平锁方式。非公平锁的优点是吞吐量比公平锁更大。

`synchronized` 和 `juc.ReentrantLock` 默认都是非公平锁。`ReentrantLock` 在构造的时候传入 `true` 则是公平锁。

```

1  /**
2   * 题目：三个售票员  卖出  30张票
3   *
4   */
5  class Ticket{//资源类
6      //票
7      private int number = 30;
8
9      Lock lock = new ReentrantLock();
10
11     public void saleTicket(){
12         lock.lock();
13
14         try{
15             if (number > 0) {
16                 System.out.println(Thread.currentThread().getName()+"\t卖出
第: "+(number--)+"\t还剩下: "+number);
17             }
18         } catch (Exception e) {
19             e.printStackTrace();
20         } finally {

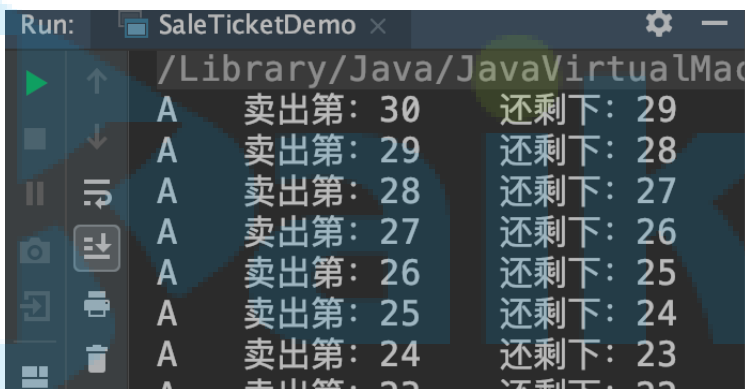
```



```

21         lock.unlock();
22     }
23 }
24 }
25
26 public class SaleTicketDemo {
27     public static void main(String[] args) {
28         Ticket ticket = new Ticket();
29
30         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "A").start();
31         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "B").start();
32         new Thread()->{ for (int i = 1; i <= 30 ; i++)
ticket.saleTicket(); }, "C").start();
33     }
34 }

```



Thread	卖出第 (Sold)	还剩下 (Remaining)
A	30	29
A	29	28
A	28	27
A	27	26
A	26	25
A	25	24
A	24	23
A	23	22

可重入锁/递归锁

```

public sync void method01()
{
    method02();
}

public sync void method02()
{
}

```

可重入锁又叫递归锁，指的同一个线程在**外层方法**获得锁时，进入**内层方法**会自动获取锁。也就是说，线程可以进入任何一个它已经拥有锁的代码块。比如 `method01` 方法里面有 `method02` 方法，两个方法都有同一把锁，得到了 `method01` 的锁，就自动得到了 `method02` 的锁。

就像有了家门的锁，厕所、书房、厨房就为你敞开了。可重入锁可以**避免死锁**的问题。

详见[ReentrantLockDemo](#)。

```
1 public class ReentrantLockDemo {
2     public static void main(String[] args) {
3         PhonePlus phonePlus = new PhonePlus();
4
5         syncTest(phonePlus);
6     }
7
8     private static void syncTest(PhonePlus phonePlus) {
9         new Thread()->{
10             phonePlus.sendSMS();
11         }, "t1").start();
12
13         new Thread()->{
14             phonePlus.sendSMS();
15         }, "t2").start();
16
17         Thread t3 = new Thread(phonePlus);
18         Thread t4 = new Thread(phonePlus);
19         t3.start();
20         t4.start();
21     }
22 }
23
24 class PhonePlus implements Runnable {
25     //Synchronized TEST
26
27     public synchronized void sendSMS() {
28         System.out.println(Thread.currentThread().getId() + "\t" +
29 "sendSMS()");
30         sendEmail();
31     }
32
33     public synchronized void sendEmail() {
34         System.out.println(Thread.currentThread().getId() + "\t" +
35 "sendEmail()");
36     }
37
38     //Reentrant TEST
39     Lock lock = new ReentrantLock();
40
41     public void method1() {
42         lock.lock();
43         try {
44             System.out.println(Thread.currentThread().getId() + "\t" +
45 "method1()");
46             method2();
47         } finally {
```

```

45         lock.unlock();
46     }
47 }
48
49 public void method2() {
50     lock.lock();
51     try {
52         System.out.println(Thread.currentThread().getId() + "\t" +
"method2()");
53     } finally {
54         lock.unlock();
55     }
56 }
57
58 @Override
59 public void run() {
60     method1();
61 }
62 }

```

锁的配对

锁之间要配对，加了几把锁，最后就得解开几把锁，下面的代码编译和运行都没有任何问题。但锁的数量不匹配会导致死循环。

```

1  lock.lock();
2  lock.lock();
3  try{
4      someAction();
5  }finally{
6      lock.unlock();
7  }

```

自旋锁

所谓自旋锁，就是尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取。自己在那儿一直循环获取，就像“自旋”一样。这样的好处是减少线程切换的上下文开销，缺点是会消耗CPU。CAS底层的 `getAndAddInt` 就是自旋锁思想。

```

1  //跟CAS类似，一直循环比较。
2  while (!atomicReference.compareAndSet(null, thread)) { }

```

```
Unsafe.class x
Decompiled .class file, bytecode version: 52.0 (Java 8)
358     public final int getAndAddInt(Object var1, long var2, int var4) {
359         int var5;
360         do {
361             var5 = this.getIntVolatile(var1, var2);
362         } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));
363
364         return var5;|
365     }
```

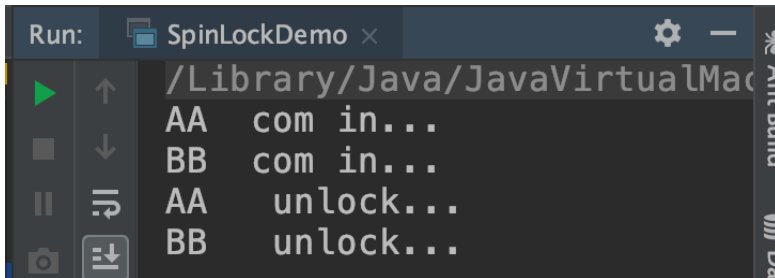
详见[SpinLockDemo](#)。

```
1  /**
2   * 题目：实现一个自旋锁
3   * 自旋锁好处：循环比较获取直到成功为止，没有类似wait的阻塞。
4   *
5   * 通过CAS操作完成自旋锁，A线程先进来调用myLock方法自己持有锁5秒钟，
6   * B随后进来后发现当前有线程持有锁，不是null，所以只能通过自选等待，直到A释放锁后B随后
  抢到。
7   */
8  public class SpinLockDemo {
9      //原子引用线程
10     AtomicReference<Thread> atomicReference = new AtomicReference<>();
11
12     public void myLock(){
13         Thread thread = Thread.currentThread();
14         System.out.println(Thread.currentThread().getName()+"\t"+"com
in...");
15         while(!atomicReference.compareAndSet(null, thread)){ }
16     }
17
18     public void myUnLock(){
19         Thread thread = Thread.currentThread();
20         atomicReference.compareAndSet(thread, null);
21         System.out.println(Thread.currentThread().getName()+"\t"+"
unlock...");
22     }
23
24     public static void main(String[] args) {
25         SpinLockDemo spinLockDemo = new SpinLockDemo();
26
27         new Thread()->{
28             spinLockDemo.myLock();
29             try { TimeUnit.SECONDS.sleep(5); } catch (InterruptedException
e) {e.printStackTrace(); }
30             spinLockDemo.myUnLock();
31             }, "AA").start();
32
33         try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e)
{e.printStackTrace(); }
34     }
```

```

35         new Thread()->{
36             spinLockDemo.myLock();
37             try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException
e) {e.printStackTrace(); }
38             spinLockDemo.myUnLock();
39             }, "BB").start();
40     }
41 }

```



读写锁/独占/共享

读锁是共享的，写锁是独占的。 `juc.ReentrantLock` 和 `synchronized` 都是独占锁，独占锁就是一个锁只能被一个线程所持有。有的时候，需要读写分离，那么就要引入读写锁，即 `juc.ReentrantReadWriteLock`。

独占锁：指该锁一次只能被一个线程所持有。对 `ReentrantLock` 和 `Synchronized` 而言都是独占锁

共享锁：指该锁可被多个线程所持有

对 `ReentrantReadWriteLock` 其读锁是共享锁，其写锁是独占锁。

读锁的共享锁可保证并发读是非常高效的，读写、写读、写写的过程是互斥的。

比如缓存，就需要读写锁来控制。缓存就是一个键值对，以下Demo模拟了缓存的读写操作，读的 `get` 方法使用了 `ReentrantReadWriteLock.ReadLock()`，写的 `put` 方法使用了 `ReentrantReadWriteLock.WriteLock()`。这样避免了写被打断，实现了多个线程同时读。

[ReadWriteLockDemo](#)

```

1  /**
2   * 多个线程同时读一个资源类没有任何问题，所以为了满足并发量，读取共享资源应该可以同时进行。
3   * 但是，如果有一个线程想去写共享资料，就不应该再有其他线程可以对该资源进行读或写
4   * 小总结：
5   *      读-读 能共存
6   *      读-写 不能共存
7   *      写-写 不能共存
8   */
9  public class ReadWriteLockDemo {
10     public static void main(String[] args) {
11         MyCache cache = new MyCache();

```

```

12
13         //写
14         for (int i = 1; i <= 5; i++) {
15             final int tempInt = i;
16             new Thread()->{
17                 cache.put(tempInt + "", tempInt + "");
18             }, String.valueOf(i)).start();
19         }
20
21         //读
22         for (int i = 1; i <= 5; i++) {
23             final int tempInt = i;
24             new Thread()->{
25                 cache.get(tempInt+"");
26             }, String.valueOf(i)).start();
27         }
28     }
29 }
30
31 class MyCache{
32     //缓存更新快，需要用volatile修饰，保证可见性，不保证原子性，一个线程修改后，通知更
    新
33     private Map<String, Object> map = new HashMap<>();
34     private ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
35
36     public void put(String key, Object value){
37         rwLock.writeLock().lock();
38         try{
39             System.out.println(Thread.currentThread().getName()+"\t正在写
    入: "+ key);
40             //模拟网络传输
41             try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException
    e) {e.printStackTrace(); }
42
43             map.put(key, value);
44             System.out.println(Thread.currentThread().getName()+"\t写入完
    成");
45         } catch (Exception e) {
46             e.printStackTrace();
47         } finally {
48             rwLock.writeLock().unlock();
49         }
50     }
51
52     public Object get(String key){
53         Object result = null;
54
55         rwLock.readLock().lock();
56         try{

```

```
57         System.out.println(Thread.currentThread().getName() + "\t正在读  
取:" + key);  
58         //模拟网络传输  
59         try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException  
e) {e.printStackTrace(); }  
60  
61         result = map.get(key);  
62         System.out.println(Thread.currentThread().getName()+"\t读取完  
成: "+result);  
63     } catch (Exception e) {  
64         e.printStackTrace();  
65     } finally {  
66         rwLock.readLock().unlock();  
67     }  
68     return result;  
69 }  
70 }
```

并发编程常用辅助类

开 课 吧

6.CountDownLatch/ CyclicBarrier/Semaphore使用过 吗?

7.阻塞队列知道吗?

CountDownLatch

让一些线程阻塞直到另一些线程完成操作后才被唤醒（班长关门）

主要有两个方法，`await`方法会被阻塞。`countDown`会让计数器-1，不会阻塞。将计数器变为0时，调用`await`方法的线程会被唤醒，继续执行。

CountDownLatchDemo

CyclicBarrier

`CyclicBarrier`字面上就是可循环使用的屏障。当一组线程得到一个屏障（同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会打开，所有被屏障拦截的线程才会继续工作。进入屏障通过`await`方法。

CyclicBarrierDemo — 七龙珠

Semaphore

信号量主要用于两个目的，一个是多个共享资源的互斥使用，一个是并发线程数的控制。

SemaphoreDemo — 争车位

CountDownLatch

`CountDownLatch` 内部维护了一个计数器，只有当计数器==0时，某些线程才会停止阻塞，开始执行。

```
CountDownLatch.java x
188
189     private final Sync sync;
190
191     /**
192      * Constructs a {@code CountDownLatch} initialized with the given count.
193      *
194      * @param count the number of times {@link #countDown} must be invoked
195      *             before threads can pass through {@link #await}
196      * @throws IllegalArgumentException if {@code count} is negative
197      */
198     public CountDownLatch(int count) {
199         if (count < 0) throw new IllegalArgumentException("count < 0");
200         this.sync = new Sync(count);
201     }
```

`CountDownLatch` 主要有两个方法，`countDown()` 来让计数器-1，`await()` 来让线程阻塞。

当 `count==0` 时，阻塞线程自动唤醒。

案例 班长关门：main线程是班长，6个线程是学生。只有6个线程运行完毕，都离开教室后，main线程班长才会关教室门。

关于 `CountDownLatch` 的使用，请看[CountDownLatchDemo](#)。

```
1  /**
2   * CountDownLatch主要有两个方法，当一个或多个线程调用await方法时，这些线程会阻塞。
3   * 其他线程调用countDown方法会将计数器减1(调用countDown方法的线程不会阻塞)，
4   * 当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行
5   */
6  public class CountDownLatchDemo {
7      public static void main(String[] args) throws InterruptedException {
8          CountDownLatch countDownLatch = new CountDownLatch(6);
9
10         for (int i = 1; i <= 6; i++) {
11             new Thread()->{
12                 System.out.println(Thread.currentThread().getName() + "\t
上完自习，离开教室");
13                 countDownLatch.countDown();
14             }.start();
15         }
16         countDownLatch.await();
17         System.out.println(Thread.currentThread().getName() + "\t班长最后关门
走人");
18     }
19 }
20 }
```

CyclicBarrier

`CountDownLatch` 是减，而 `CyclicBarrier` 是加，理解了 `CountDownLatch`，`CyclicBarrier` 就很容易。比如召集7颗龙珠才能召唤神龙，详见[CyclicBarrierDemo](#)。

```
1  public class CyclicBarrierDemo {
2      public static void main(String[] args) {
3
4          CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
5              System.out.println("====召唤神龙");
6          });
7
8          for (int i = 1; i <= 7; i++) {
9              final int tempInt = i;
10             new Thread()->{
11                 System.out.println(Thread.currentThread().getName() + "\t
收集到第" + tempInt + "颗龙珠");
12
13                 try {
14                     cyclicBarrier.await();
15                 } catch (Exception e) {
```

```

16         e.printStackTrace();
17     }
18     }, String.valueOf(i)).start();
19 }
20 }
21 }

```

```

Run: CyclicBarrierDemo x
/Library/Java/JavaVirtualMa
1   收集到第1颗龙珠
5   收集到第5颗龙珠
4   收集到第4颗龙珠
3   收集到第3颗龙珠
2   收集到第2颗龙珠
7   收集到第7颗龙珠
6   收集到第6颗龙珠
====召唤神龙

```

Semaphore

`CountDownLatch` 的问题是**不能复用**。比如 `count=3`，那么加到3，就不能继续操作了。

而 `Semaphore` 可以解决这个问题，比如6辆车3个停车位，对于 `CountDownLatch` 只能停**3辆车**，而 `Semaphore` 可以停6辆车，车位空出来后，其它车可以占有，这就涉及到了 `Semaphore.acquire()` 和 `Semaphore.release()` 方法。

```

1  /**
2   * 在信号量上我们定义两种操作：
3   * acquire（获取）当一个线程调用acquire操作时，他要么通过成功获取信号量（信号量减
4   * 1），要么一直等待下去，直到有线程释放信号量，或超时。
5   * release（释放）实际上会将信号量加1，然后唤醒等待的线程。
6   *
7   * 信号量主要用于两个目的，一个是用于多个共享资源的互斥使用，另一个用于并发线程数的控制
8   */
9  public class SemaphoreDemo {
10     public static void main(String[] args) {
11         Semaphore semaphore = new Semaphore(3); //模拟资源类，有3个空车位
12
13         for (int i = 1; i <= 6; i++) {
14             new Thread(()->{
15                 try{
16                     //占有资源
17                     semaphore.acquire();
18
19                     System.out.println(Thread.currentThread().getName()+"\t抢到车位");
20                 } catch (InterruptedException e) {
21                     e.printStackTrace();
22                 }
23             }).start();
24         }
25     }
26 }

```

```

19         try { TimeUnit.SECONDS.sleep(3); } catch
(InterruptedException e) {e.printStackTrace(); }
20
    System.out.println(Thread.currentThread().getName()+"\t停车3秒后离开车位");
21        } catch (Exception e) {
22            e.printStackTrace();
23        } finally {
24            //释放资源
25            semaphore.release();
26        }
27        }, String.valueOf(i)).start();
28    }
29 }
30 }

```

Run: SemaphoreDemo x

```

/Library/Java/JavaVirtualMa
1  抢到车位
3  抢到车位
2  抢到车位
2  停车3秒后离开车位
1  停车3秒后离开车位
3  停车3秒后离开车位
5  抢到车位
4  抢到车位
6  抢到车位
4  停车3秒后离开车位
5  停车3秒后离开车位
6  停车3秒后离开车位

```