JUC并发编程

JUC并发包API 包介绍

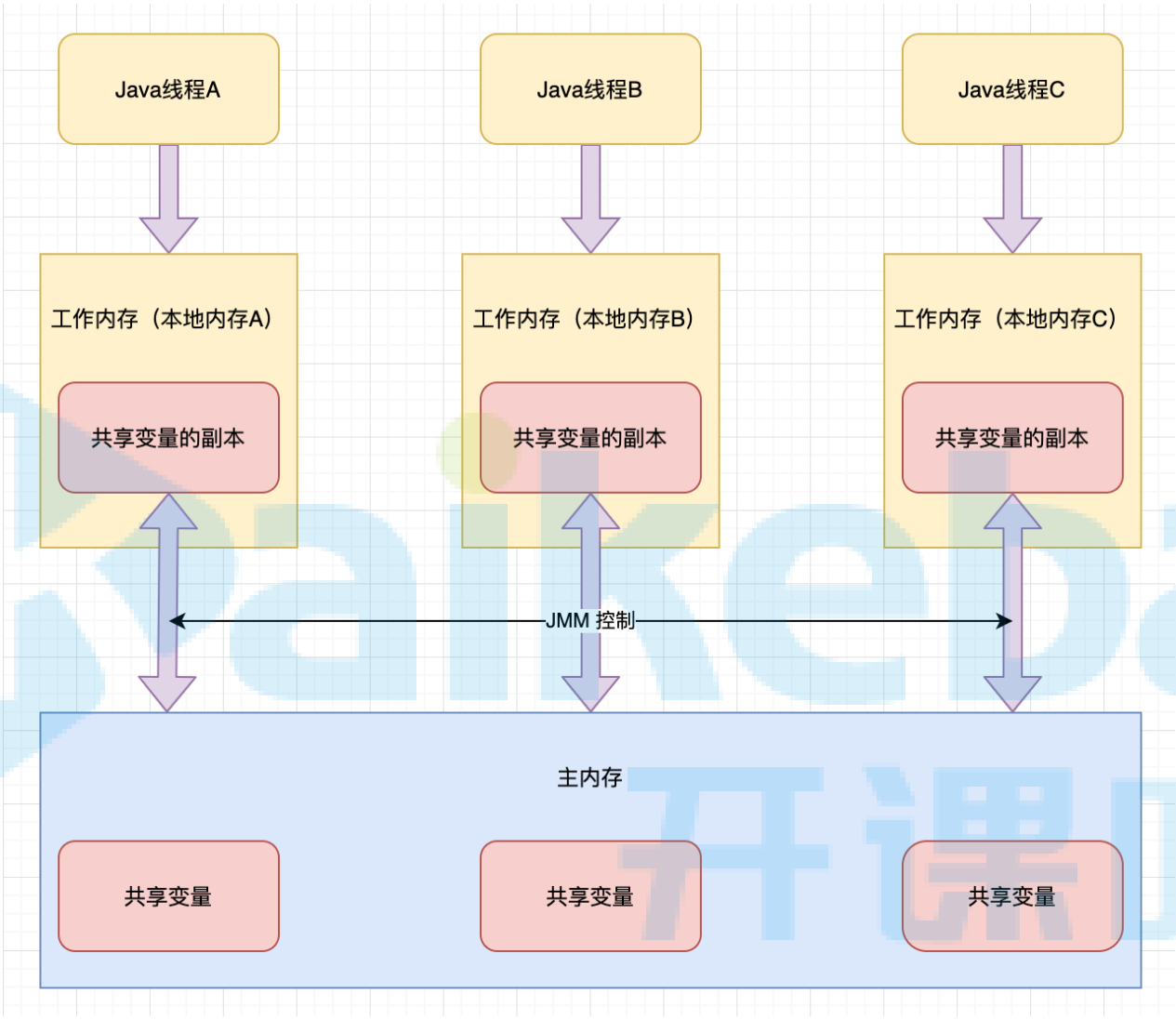
> java.util.concurrent:
>
> 1. 并发与并行的不同?
>
>   1. 并发，如同，秒杀一样，多个线程访问同一个资源
>   2. 并行，一堆事情 一块去做，如同，一遍烧热水，一个拆方便面包装
>
> java.util.concurrent.atomic
>
> 1. AtomicInteger 原子性引用
>
> java.util.concurrentlocks
>
> 1. Lock接口
> 2. ReentrantLock 可重入锁
> 3. ReadWriteLock 读写锁

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：



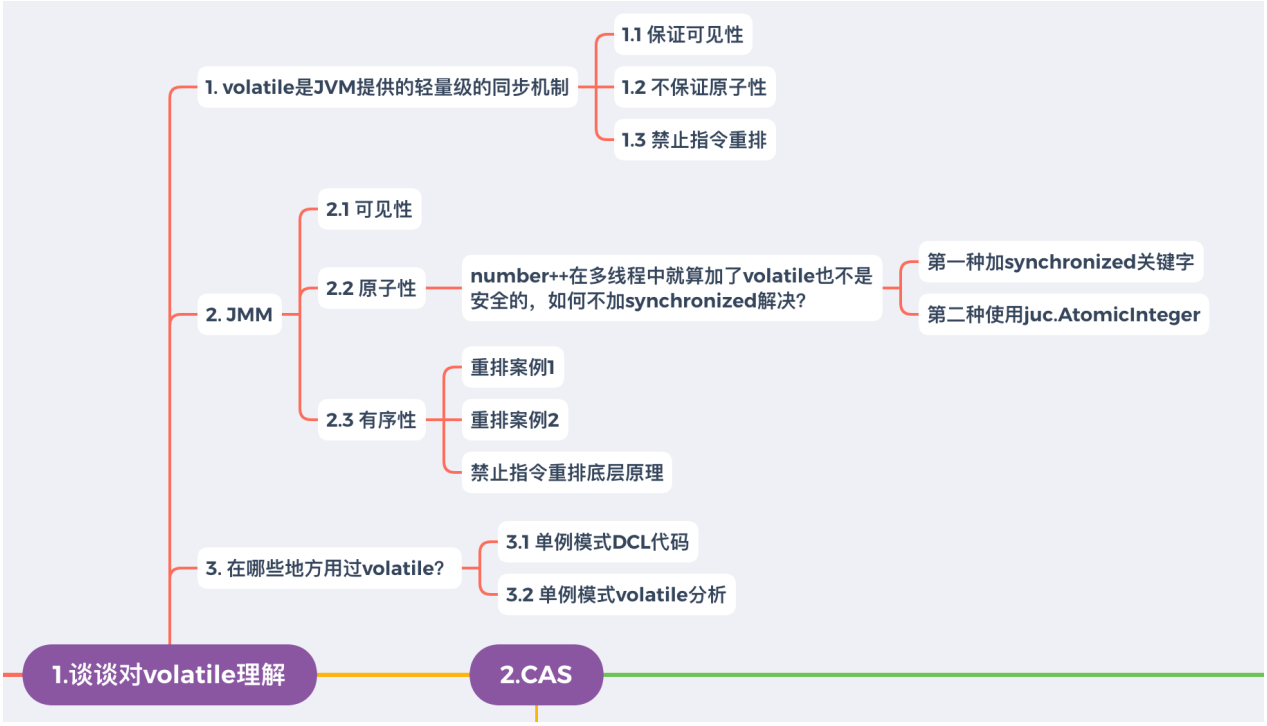
JMM可能带来**可见性**、**原子性**和**有序性**问题。

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提供的一种**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

```
class MyData {
    int number = 0;
    //volatile int number = 0;

    public void setTo60() {
        this.number = 60;
    }

}

public class VolatileDemo {
    public static void main(String[] args) {
        volatileVisibilityDemo();
    }

    //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
    private static void volatileVisibilityDemo() {
        System.out.println("可见性测试");
        MyData myData = new MyData();//资源类
        //启动一个线程操作共享数据
```

```
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "\t 执行");
            try {
                TimeUnit.SECONDS.sleep(3);
                myData.setTo60();
                System.out.println(Thread.currentThread().getName() + "\t 更新number值：" + myData.number);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "ThreadA").start();
        while (myData.number == 0) {
            //main线程持有共享数据的拷贝，一直为0
        }
        System.out.println(Thread.currentThread().getName() + "\t main获取number值：" + myData.number);
    }
}
```

`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：

```
可见性测试
ThreadA	执行
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：

```
可见性测试
ThreadA	执行
ThreadA	更新number值：60
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。

```
class MyData{
    //int number=0;
```

```
    volatile int number=0;

    //此时number前面已经加了volatile，但是不保证原子性
    public void addPlusPlus(){
        number++;
    }
}
public class VolatileDemo {
    public static void main(String[] args) {
        //volatileVisibilityDemo();
        atomicDemo();
    }

    private static void atomicDemo() {
        System.out.println("原子性测试");
        MyData myData=new MyData();
        for (int i = 1; i <= 20; i++) {
            new Thread(()->{
                for (int j = 0; j <1000 ; j++) {
                    myData.addPlusPlus();
                }
            },String.valueOf(i)).start();
        }
        while (Thread.activeCount()>2){
            Thread.yield();
        }
        System.out.println(Thread.currentThread().getName()+"\t int类型最终number值："+myData.number);
    }
}
```

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。

```
javap -c 包名.类名
```

```
javap -c MyData

public void addPlusPlus();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2                  // Field number:I   //读
     5: iconst_1                                              //++常量1
     6: iadd                                                  //加操作
     7: putfield      #2                  // Field number:I   //写操作
    10: return

```

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

```
class MyData{
    //int number=0;
    volatile int number=0;

    AtomicInteger atomicInteger=new AtomicInteger();

    public void setTo60(){
        this.number=60;
    }

    //此时number前面已经加了volatile，但是不保证原子性
    public void addPlusPlus(){
```

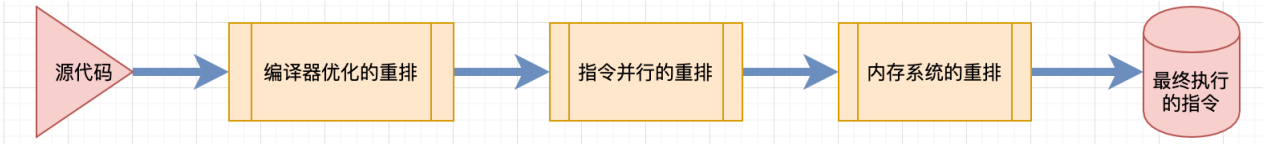
```
        number++;
    }

    public void addAtomic(){
        atomicInteger.getAndIncrement();
    }
}

public class VolatileDemo {
    public static void main(String[] args) {
        //volatileVisibilityDemo();
        atomicDemo();
    }

    private static void atomicDemo() {
        System.out.println("原子性测试");
        MyData myData=new MyData();
        for (int i = 1; i <= 20; i++) {
            new Thread(()->{
                for (int j = 0; j <1000 ; j++) {
                    myData.addPlusPlus();
                    myData.addAtomic();
                }
            },String.valueOf(i)).start();
        }
        while (Thread.activeCount()>2){
            Thread.yield();
        }
        System.out.println(Thread.currentThread().getName()+"\t int类型最终number值："+myData.number);
        System.out.println(Thread.currentThread().getName()+"\t AtomicInteger类型最终number值："+myData.atomicInteger);
    }
}
```

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。

```
原子性测试
main     int类型最终number值：17751
main     AtomicInteger类型最终number值：20000
```

有序性

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。

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

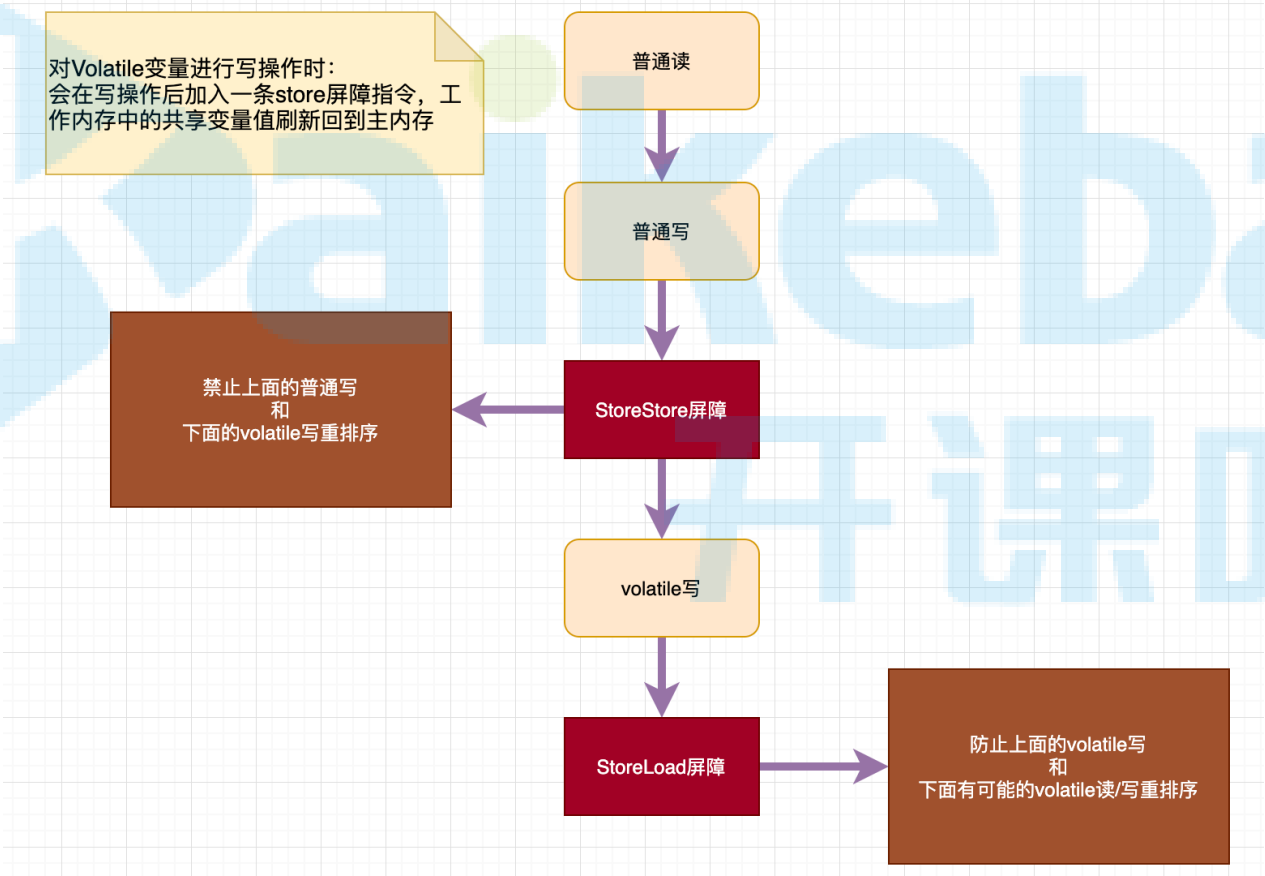
```
public class ResortSeqDemo {

    int a=0;
    boolean flag=false;
    /*
    多线程下flag=true可能先执行，还没走到a=1就被挂起。
    其它线程进入method02的判断，修改a的值=5，而不是6。
     */
    public void method01(){
        a=1;
        flag=true;
    }
    public void method02(){
        if (flag){
            a+=5;
            System.out.println("*****最终值a: "+a);
        }
    }

    public static void main(String[] args) {
        ResortSeqDemo resortSeq = new ResortSeqDemo();

```

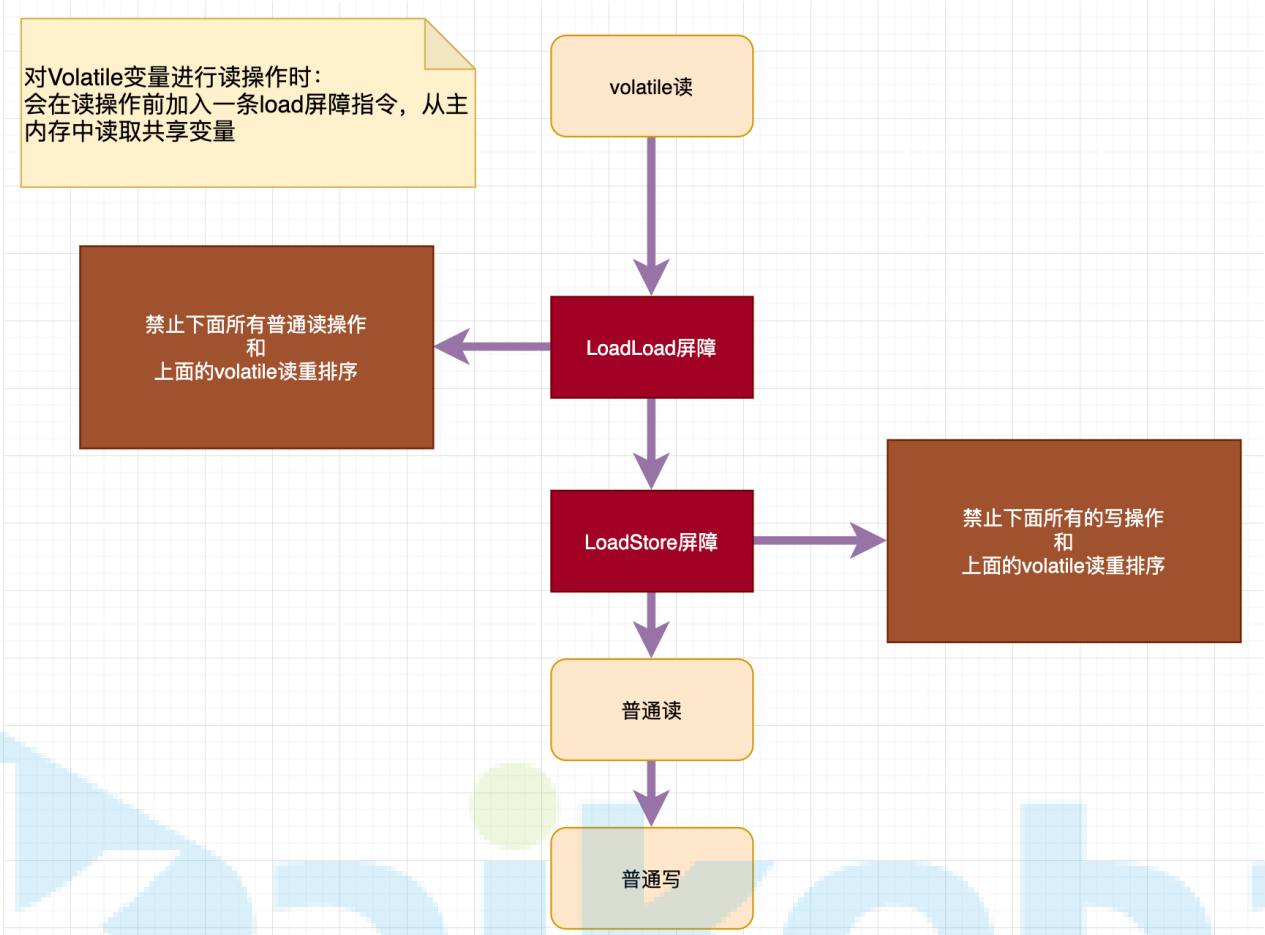
```
23          new Thread(()->{resortSeq.method01();},"ThreadA").start();
24          new Thread(()->{resortSeq.method02();},"ThreadB").start();
25      }
26  }
```

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。





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。

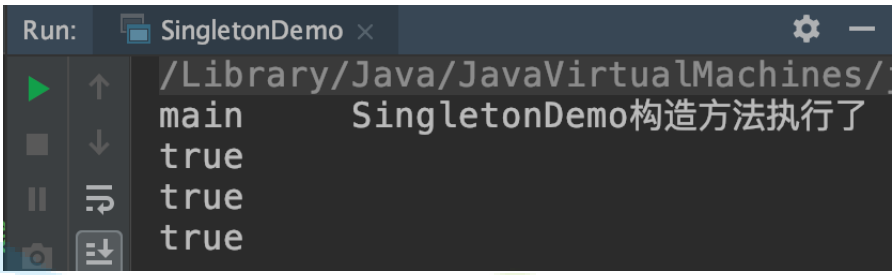
哪些地方用到过volatile?

单例模式的安全问题

- 传统

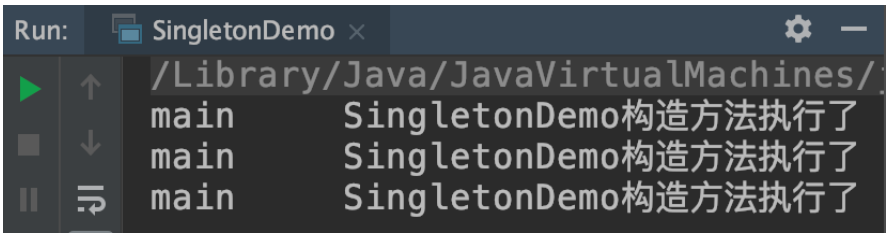
```
public class SingletonDemo {
    private static SingletonDemo instance = null;

    private SingletonDemo() {
        System.out.println(Thread.currentThread().getName() +"\t SingletonDemo构造方法执行了");
    }

    public static SingletonDemo getInstance(){
        if (instance == null) {
            instance = new SingletonDemo();
        }
        return instance;
    }
```

```
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  }
```



- 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  }
```



- 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。

```
public class SingletonDemo {
    private static SingletonDemo instance = null;

    private SingletonDemo() {
        System.out.println(Thread.currentThread().getName() +"\t
SingletonDemo构造方法执行了");
    }

    public static SingletonDemo getInstance(){
        if (instance == null) {
            synchronized (SingletonDemo.class){
                if (instance == null) {
                    instance = new SingletonDemo();
                }
            }
        }
        return instance;
    }

    public static void main(String[] args) {
        //多线程操作
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                SingletonDemo.getInstance();
            },Thread.currentThread().getName()).start();
        }

    }
}
```

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

```
instance = new SingletonDemo();

public static thread.SingletonDemo getInstance();
  Code:
     0: getstatic     #11       // Field instance:Lthread/SingletonDemo;
     3: ifnonnull     37
     6: ldc           #12       // class thread/SingletonDemo
     8: dup
     9: astore_0
    10: monitorenter
    11: getstatic     #11       // Field instance:Lthread/SingletonDemo;
    14: ifnonnull     27
    17: new           #12       // class thread/SingletonDemo 步骤1
    20: dup
    21: invokespecial #13       // Method "<init>":()V 步骤2
    24: putstatic     #11       // Field instance:Lthread/SingletonDemo;步骤3

底层Java Native Interface中的C语言代码内容，开辟空间的步骤
memory = allocate();    //步骤1.分配对象内存空间
instance(memory);       //步骤2.初始化对象
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。

```
memory = allocate();     //步骤1. 分配对象内存空间
instance(memory);        //步骤2.初始化对象
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。

```
memory = allocate();     //步骤1. 分配对象内存空间
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！
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。

```
public static SingletonDemo getInstance(){
    if (instance == null) {
        synchronized (SingletonDemo.class){
            if (instance == null) {
                instance = new SingletonDemo(); //多线程情况下，可能发生指令重排
            }
        }
    }
    return instance;
}
```

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。

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