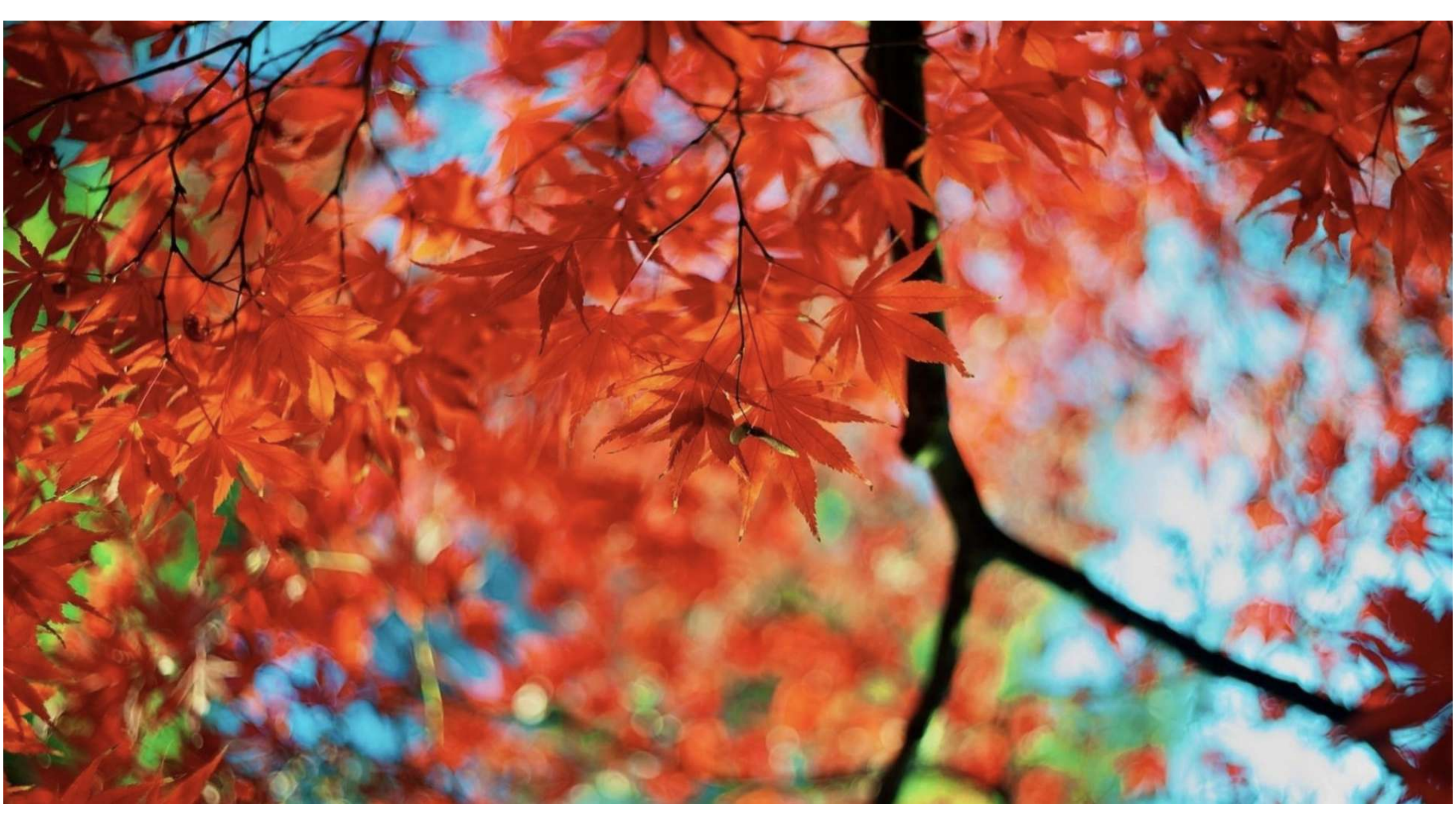


> 主要讲解volatile的相关知识，以及容易遇到的坑。

volatile变量的特性

保证可见性，不保证原子性：

- 当写一个volatile变量时，JMM会把该线程本地内存中的变量强制刷新到主内存中去；
- 这个写会操作会导致其他线程中的volatile变量缓存无效。

禁止指令重排，我们回顾一下，重排序需要遵守一定规则：

- 重排序操作不会对存在数据依赖关系的操作进行重排序。比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
- 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变。 比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

volatile禁止指令重排规则

使用volatile关键字修饰共享变量便可以禁止这种重排序。若用volatile修饰共享变量，在编译时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序，volatile禁止指令重排序也有一些规则：

- 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
- 在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

> 即执行到volatile变量时，其前面的所有语句都执行完，后面所有语句都未执行。且前面语句的结果对volatile变量及其后面语句可见。

volatile禁止指令重排分析

> 该部分相关内容，我直接copy上一篇文章，不是为了凑篇幅，因为有同学没有看上一篇文章，直接看这篇，为了能让每一篇文章能独立成章，可能会引用之前文章中的内容。

先看下面未使用volatile的代码：

```
class ReorderExample {
  int a = 0;
  boolean flag = false;
  public void writer() {
      a = 1;                   //1
      flag = true;             //2
  }
  Public void reader() {
      if (flag) {                //3
          int i =  a * a;        //4
          System.out.println(i);
      }
  }
}
```

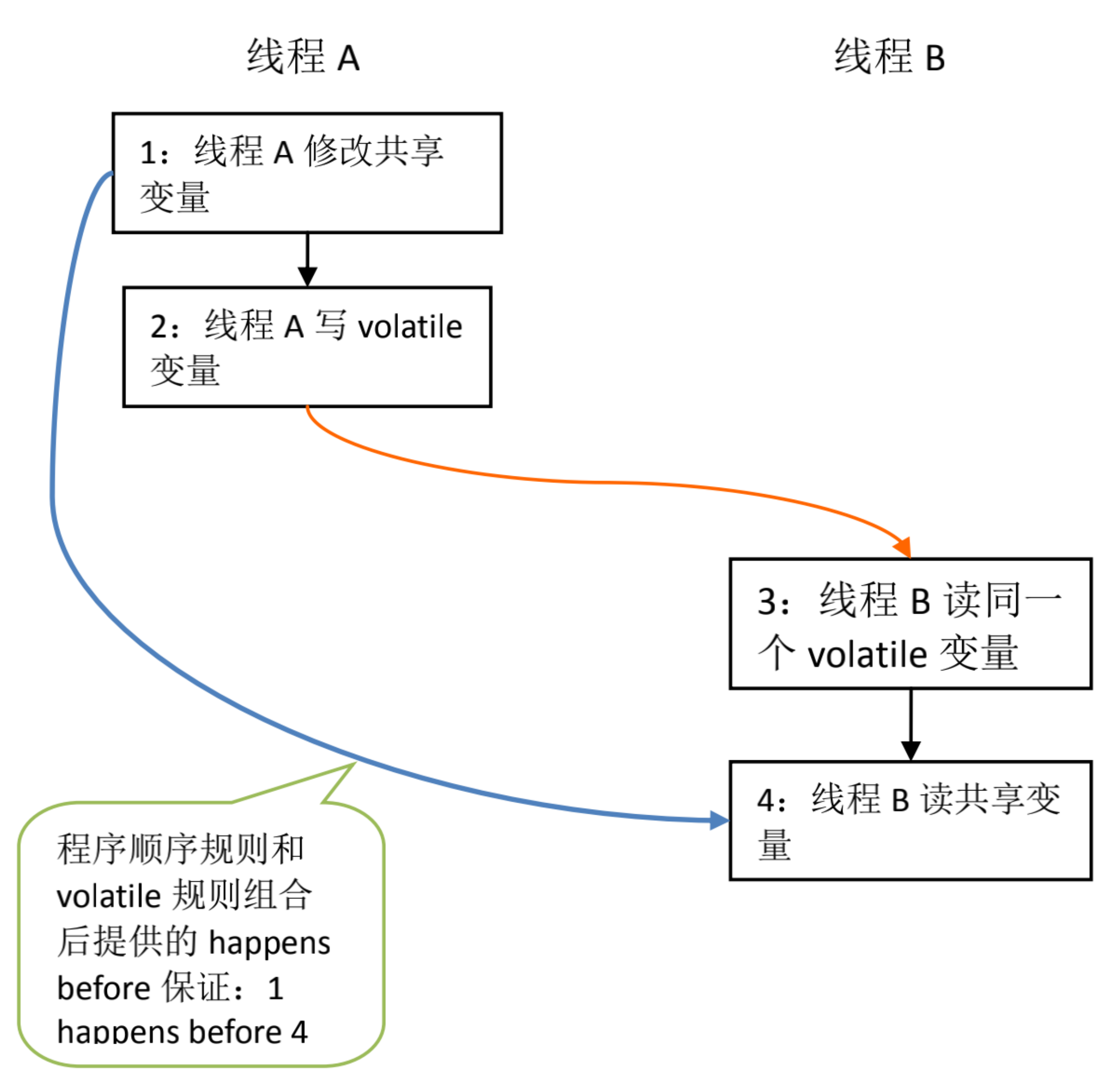
因为重排序影响，所以最终的输出可能是0，，具体分析请参考我的上一篇文章《Java并发编程系列1-基础知识》，如果引入volatile，我们再看一下代码：

```
class ReorderExample {
  int a = 0;
  boolean volatile flag = false;
  public void writer() {
      a = 1;                   //1
      flag = true;             //2
  }
  Public void reader() {
      if (flag) {                //3
          int i =  a * a;        //4
          System.out.println(i);
      }
  }
}
```

这个时候，volatile禁止指令重排序也有一些规则，这个过程建立的happens before关系可以分为两类：

1. 根据程序次序规则，1 happens before 2; 3 happens before 4。
2. 根据volatile规则，2 happens before 3。
3. 根据happens before 的传递性规则，1 happens before 4。

上述happens before关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示volatile规则；蓝色箭头表示组合这些规则后提供的happens before保证。

这里A线程写一个volatile变量后，B线程读同一个volatile变量。A线程在写volatile变量之前所有可见的共享变量，在B线程读同一个volatile变量后，将立即变得对B线程可见。

volatile不适用场景

volatile不适合复合操作

下面是变量自加的示例：

```
public class volatileTest {
    public volatile int inc = 0;
    public void increase() {
        inc++;
    }
    public static void main(String[] args) {
        final volatileTest test = new volatileTest();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("inc output:" + test.inc);
    }
}
```

测试输出：

```
inc output:8182
```

因为inc++不是一个原子性操作，可以由读取、加、赋值3步组成，所以结果并不能达到10000。

解决方法

采用synchronized：

```
public class volatileTest1 {
    public int inc = 0;
    public synchronized void increase() {
        inc++;
    }
    public static void main(String[] args) {
        final volatileTest1 test = new volatileTest1();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add synchronized, inc output:" + test.inc);
    }
}
```

采用Lock：

```
public class volatileTest2 {
    public int inc = 0;
    Lock lock = new ReentrantLock();
    public void increase() {
        lock.lock();
        inc++;
        lock.unlock();
    }
    public static void main(String[] args) {
        final volatileTest2 test = new volatileTest2();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add lock, inc output:" + test.inc);
    }
}
```

采用AtomicInteger：

```
public class volatileTest3 {
    public AtomicInteger inc = new AtomicInteger();
    public void increase() {
        inc.getAndIncrement();
    }
    public static void main(String[] args) {
        final volatileTest3 test = new volatileTest3();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<100;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add AtomicInteger, inc output:" + test.inc);
    }
}
```

三者输出都是1000，如下：

```
add synchronized, inc output:1000
add lock, inc output:1000
add AtomicInteger, inc output:1000
```

单例模式的双重锁为什么要加volatile

先看一下单例代码：

```
public class penguin {
    private static volatile penguin m_penguin = null;
    // 避免通过new初始化对象
    private void penguin() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin getInstance() {      //1
        if (null == m_penguin) {               //2
            synchronized(penguin.class) {      //3
                if (null == m_penguin) {       //4
                    m_penguin = new penguin(); //5
                }
            }
        }
        return m_penguin;                      //6
    }
}
```

在并发情况下，如果没有volatile关键字，在第5行会出现问题。instance = new TestInstance();可以分解为3行伪代码：

```
a. memory = allocate() //分配内存
b. ctorInstanc(memory) //初始化对象
c. instance = memory   //设置instance指向刚分配的地址
```

上面的代码在编译运行时，可能会出现重排序从a-b-c排序为a-c-b。在多线程的情况下会出现以下问题。当线程A在执行第5行代码时，B线程进来执行到第2行代码。假设此时A执行的过程中发生了指令重排序，即先执行了a和c，没有执行b。那么由于A线程执行了c导致instance指向了一段地址，所以B线程判断instance不为null，会直接跳到第6行并返回一个未初始化的对象。

何时使用volatile

大家对volatile变量的学习，关于重排序的规则，这个仅做了解即可，更重要的是掌握它的正常使用场景。那对于锁和volatile，我们什么时候才会去使用volatile呢？我们先回顾一下volatile和锁的区别：

> 加锁机制既可以保证可见性，又可以保证原子性，而volatile变量只能保证可见性。

也就是我们只在可见性上，才去使用volatile变量，比如在多线程情况下，我们需要对某个操作的完成、发生中断或者状态的标志，就可以声明为volatile，因为volatile可以保证该变量在所有线程的可见性。但是如果对于稍微复杂的操作，比如i++等复合操作，就不要使用volatile变量，如果你想通过volatile的“禁止指令重排规则”来保证volatile变量的前后变量代码的顺序性，建议你不要这样做，一方面有很多坑，另一方面别人也不理解你的代码，比如我示例中的“volatile禁止指令重排分析”，即使我讲了一遍，我自己都很容易忘，你还指望别人能理解么？如果你偏要这样去“炫技”，那你就是给别人埋坑了！就好比一行很简单的代码，有位程序员偏要用“&|”操作，结果埋个大坑，给部门造成S级的线上事故。

下面是《Java并发编程实战》的描述，我摘抄一下，其实有第一条不理解，仅做记录：

> 当且仅当满足以下所有条件时，才应该使用volatile变量：
>
> - 对变量的写操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
> - 该变量不会与其它状态变量一起纳入不可变的条件中。
> - 在访问变量时不需要加锁。

总结

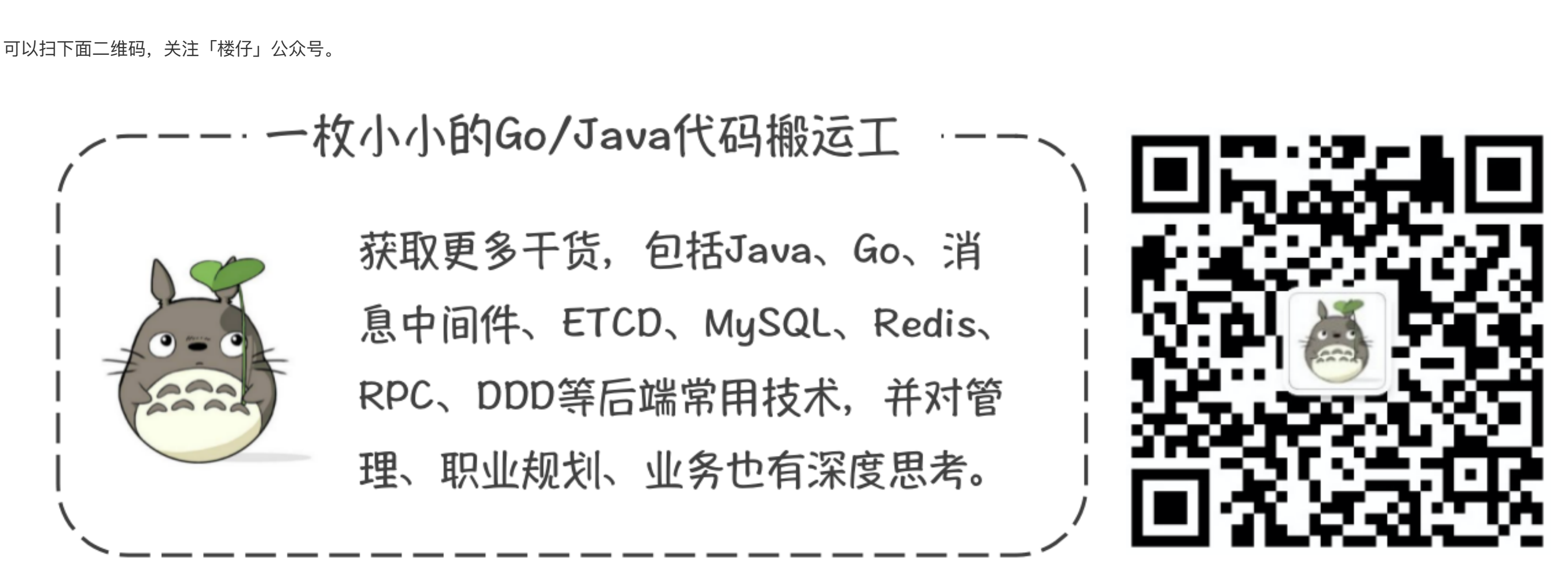
volatile可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在JVM底层volatile是采用“内存屏障”来实现的。观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令，lock前缀指令实际上相当于一个内存屏障（也称内存栅栏），内存屏障会提供3个功能：

- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 它会强制将对缓存的修改操作立即写入主存；
- 如果是写操作，它会导致其他CPU中对应的缓存行无效。

最后也讲解了volatile不适用的场景，以及解决的方法，并解释了单例模式为何需要使用volatile。

学习交流

可以扫下面二维码，关注「楼仔」公众号。



长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏🏻👏🏻👏🏻

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。