在认识volatile之前，我们先来了解一下Java并发编程的三大性质即：是原子性、可见性以及有序性。

原子性 原子在化学中反应上是不可在分割的粒子。因此原子性指的是一个不可以被分割的操作，即这个操作在执行过程中不能被中断，要么全部不执行，要么全部执行。且一旦开始执行，不会被其他线程打断。

可见性 指的是一个线程修改了共享变量后，另外线程能立即感知这个变量被修改。

有序性 指程序按照代码的先后顺序执行。有时候为了优化性能，编译器会对字节码指令进行重排序。但是能保证重排序后的执行结果与重排序之前是一致的。

volatitle经常被用到并发编程的场景中。它的作用有两个，即：

保证可见性；

保证有序性。

但是，要注意volatile关键字并不能保证原子性。接下来我们对volatile的两个作用进行详细分析。

保证可见性

由于每个线程都有自己的工作空间，导致多线程的场景下会出现缓存不一致性的问题。即，当两个线程共用一个共享变量时，如果其中一个线程修改了这个共享变量的值。但是由于另外一个线程在自己的工作内存中已经保留了一份该共享变量的副本，因此它无法感知该变量的值已经被修改。

看下面的一个例子：

public class VolatileDemo {

private static boolean ready;

public static class MyThread extends Thread {

@Override

public void run() {

System.out.println("MyThread is running...");

while (!ready) ; // 如果ready为false，则死循环

System.out.println("MyThread is end");

}

}

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

new MyThread().start();

Thread.sleep(1000);

ready = true;

System.out.println("ready = " + ready);

Thread.sleep(5000);

System.out.println("main thread is end.");

}

}

代码中定义了一个boolean类型的成员变量ready,其默认值为false。在MyThread线程中判断如果ready为false时则进行死循环。接下来在main方法中开启MyThread线程，并在睡眠1s后将ready修改为true。正常情况下ready修改为true后MyThread线程中的死循环则会停止，并打印“MyThread is end"。但是来看下运行效果跟我们猜想是否一致，打印日志如下：

MyThread is running...

ready = true

main thread is end.

可以看见当ready被修改为true后，MyThread线程依然未结束。通过这一例子也证实了MyThread线程中的ready副本并没有得到及时的更新。

那么接下来我们将成员变量ready使用volatile关键字修饰后，再运行看打印日志：

MyThread is running...

MyThread is end

ready = true

main thread is end.

可见，当在主线程中修改了ready为true后，MyThread线程立即感知了ready的变化，并结束了死循环。从这个例子中也可以看见volatile确实能有效的保证多个线程共享变量的可见性。

保证有序性

我们知道，编译器为了优化程序性能，可能会在编译时对字节码指令进行重排序。重排序后的指令在单线程中运行时没有问题的，但是如果在多线程中，重排序后的代码则可能会出现问题。因此，一般在多线程并发情况下我们都应该禁止指令重排序的优化。而volatile关键字就可以禁止编译器对字节码进行重排序。volatile保证有序性在我们平时开发中有一个很常见的例子，即双重锁校验的单例模式下需要使用volatile关键字来禁止指令重排序。我们来看下代码：

public class DoubleCheckLock {

private volatile static DoubleCheckLock instance;

private DoubleCheckLock(){}

public static DoubleCheckLock getInstance(){

//第一次检测

if (instance==null){

//同步

synchronized (DoubleCheckLock.class){

if (instance == null){

//多线程环境下可能会出现问题的地方

instance = new DoubleCheckLock();

}

}

}

return instance;

}

}

如果上述代码中没有给instance加上volatile关键字会怎么呢？我们不妨来分析一下，首先我们应该清楚instance = new DoubleCheckLock();这一操作并不是一个原子操作，实例化对象的字节指令可以分为三步，如下：

1.分配对象内存：memory = allocate();

2.初始化对象：instance(memory);

3.instance指向刚分配的内存地址：instance = memory;

而由于编译器的指令重排序，以上指令可能会出现以下顺序：

1.分配对象内存：memory = allocate();

2.instance指向刚分配的内存地址：instance = memory;

3.初始化对象：instance(memory);

以优化后的字节码指令来看双重锁校验的代码是否有问题呢？不难发现，如果线程1第一次调用单例方法，在该线程的时间片轮转结束后执行到了优化后的第二个指令，即instance被赋值，但是还未被分配初始化对象。此时，线程2抢到了CPU时间片，同时调用了getInstance方法，第一次校验就发现instance不为null，遂将其返回。在得到这个单例后调用单例的方法，此时必定出现空指针异常。

因此，可见指令重排序在多线程并发的情况下是会出现问题的。此时，我们便可以通过volatile关键字来禁止编译器的优化，从而避免空指针的出现。

volatile不保证原子性

对于原子操作，volatile关键字是无能为力的。如果需要保证原子操作，则需要使用synchronized关键字、Lock锁 或者Autom相关类来确保操作的原子性。