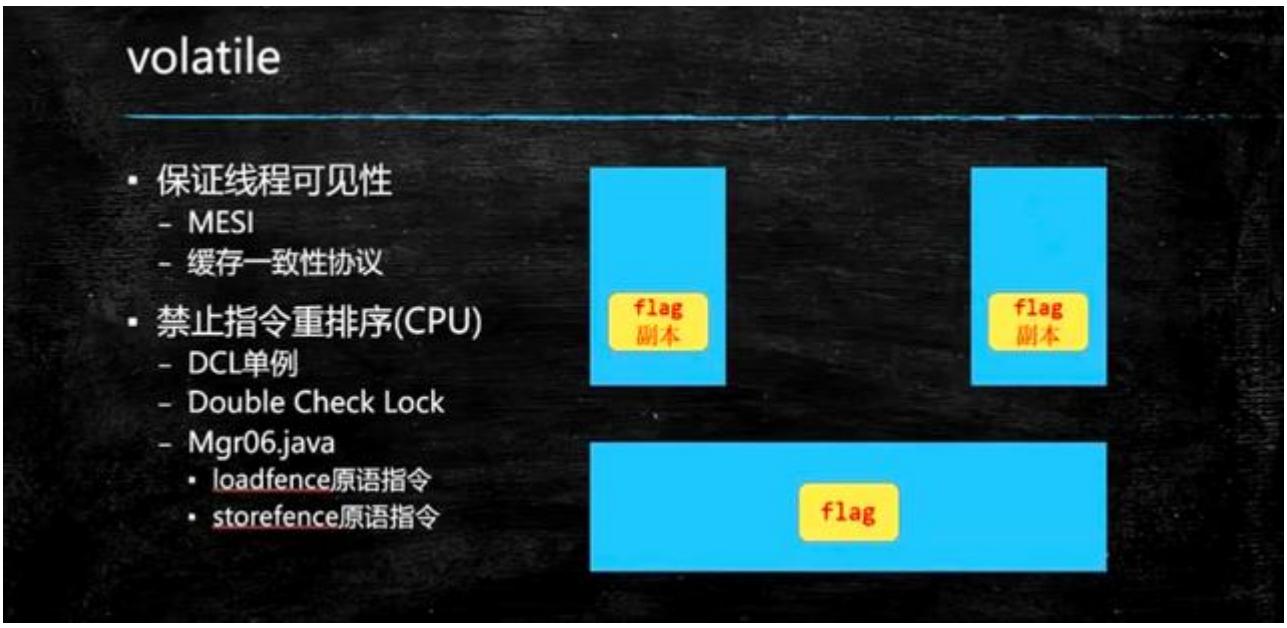
# volatile

我们先来看这个volatile的概念，volatile它是什么意思，现在像大的互联网企业的面试，基本上volatile是必会的，有时候他也不会太问，认为你应该会，但是中小企业也就开始问这方面的问题。

我们来看一下这个小程序，写了一个方法啊，首先定义了一个变量 布尔类型 等于true，这里模拟的是一个服务器的操作，我的值为true你就给我不间断的运行，什么时候为false你再停止。 测试new Thread启动一个线程，调用m方法，睡了一秒，最后running等于false，运行方法他是不会停止的。 如果你要把volatile打开，那么结果就是启动程序一秒之后他就会m end停止。(volatile就是不停的追踪这个值，时刻看什么时候发生了变化)

```
/*** volatile 关键字，使一个变量在多个线程间可见* A B线程都用到一个变量，java默认是A线程中保留-
```

volatile作用(由于面试常考到，必须记着)




**1：保证线程的可见性**

大家知道java里面是有堆内存的，堆内存是所有线程共享里面的内存，除了共享的内存之外呢，每个线程都有自己的专属的区域，都有自己的工作内存，如果说在共享内存里有一个值的话，当我们线程，某一个线程都要去访问这个值的时候，会将这个值copy一份，copy到自己的这个工作空间里头，然后对这个值的任何改变，首先是在自己的空间里进行改变，什么时候写回去，就是改完之后会马上写回去。

什么时候去检查有没有新的值，也不好控制。

在这个线程里面发生的改变，并没有及时的反应到另外一个线程里面，这就是线程之间的不可见，对这个变量值加了volatile之后就能够保证 一个线程的改变，另外一个线程马上就能看到。

大家可以去查这个词：MESI ，他的本质上是使用了cpu的一个叫做 高速缓存一致性协议

注：在和面《JVM》的课程中，老师深入讲解了MESI协议和volatile的底层实现。

**2：禁止指令重新排序**

指令重排序也是和cpu有关系，每次写都会被线程读到，加了volatile之后。cpu原来执行一条指令的时候它是一步一步的顺序的执行，但是现在的cpu为了提高效率，它会把指令并发的来执行，第一个指令执行到一半的时候第二个指令可能就已经开始执行了，这叫做流水线式的执行。在这种新的架构的设计基础之上呢想充分的利用这一点，那么就要求你的编译器把你的源码编译完的指令之后呢可能进行一个指令的重新排序。

这个是通过实际工程验证了，不仅提高了，而且提高了很多。-DCL单例

我们来聊一聊什么是单例，单例的意思就是我保证你在JVM的内存里头永远只有某一个类的一个实例，其实这个很容易理解，在我们工程当中有一些类真的没有必要new好多个对象，比如说权限管理者。

单例最简单的写法就是下面这种写法，是说我有一个类，定义了这个类的一个对象，然后一个对象呢

是在个类的内部的，同时我把Mgr01()这个类的构造方法设置成private意思就是别的不要去new我，只有我自己能new，理论上来说我就只有自己一个实例了，通过getInstance()访问这个实例，所以无论你调用多少次的getInstanc()本质上它就只有这一个对象，这种写法非常简洁也很容易理解，由JVM来保证永远只有这一个实例。

```
package com.mashibing.dp.singleton;/*** 饿汉式* 类加载到内存后，被实例化一个单例，JVM保证线程
```

但是有的人他会吹毛求疵，他会说我还没开始用这个对象呢，没用这个对象调这个方法你干嘛把他初始化了，你能不能什么时候开始用，调这个方法的时候你再给我初始化。所以呢，下面代码这个是和上一种一样的写法。

```
package com.mashibing.dp.singleton;/*** 跟01是一个意思**/public class Mgr02{private stati
```

所以另外产生这种懒汉式的单例，意思是说我getInstance()，什么时候我开始调用这个getInstace()的时候，我才对它进行初始化。当然，这个不要对它进行初始化两次，只能初始化一次才对，不然就成了俩对象了吗，所以上来之后先判断*INSTANCE == null* 的话我才初始化。

不过，更加吹毛求疵的事情又来了，我不单要求你我用的时候才进行初始化，我还要求你线程安全。显然我们下面03这个是不保证线程安全的，所以你多个线程访问的时候它一定会出问题，下来你自己可以实验实验。

```
/*** 懒汉式单例*/public class Mgr03{private static Mgr03 INSTANCE;private Mgr03(){}public
```

所以他要怎么做呢，我们要加一个synchronized解决，加一把锁嘛*public static synchronized* 这句话一旦加上就没问题了，因为这个里面从头到尾就只有一个线程运行，第一个线程发现它为空给它new了，第二个线程他无论怎么访问这个值已经永远不可能为空了，它只能是拿原来第一个线程初始化的部分，这是没问题的

```
/*** lazy loading* 也称懒汉式* 虽然达到了按需初始化的目的，但却能带来线程不安全的问题* 可以通过
```

开始进一步的吹毛求疵synchronized一下加在方法上这个代码太长了，说不定里面还有其他的业务逻辑，对于加锁这个事情，代码能锁的少的就要尽量锁的少。那么通过进一步的吹毛求疵又有了新的写法如下代码：线程判断，先别加锁，判断是否为空，如果为空在加锁初始化，更细粒度的一个锁，这叫做锁细化，也是锁优化的一步。很不幸的是这个写法是不对的，我们分析一下，第一个线程判断它为空，还没有执行下面的过程第二个线程来了，也判断它为空。第一个线程对它进行了加锁，synchronized完了之后呢把锁释放了，而第二个线程也是判断为空拿到这把锁也初始化了一遍，所以这种写法是有问题的。

```
public class Mgr05{private static Mgr05 INSTANCE;private Mgr05(){}public static Mgr05 ge
```

所以就产生了，我们今天要讲的volatile这个问题，这个问题是这样来产生的，看下面代码，叫做双重检查锁或者叫双重检查的单例，在这种双重检查判断的情况下刚才上面的说的线程问题就不会再有了，

分析一下：第一个线程来了判断ok，你确实是空值，然后进行下面的初始化过程，假设第一个线程把这个INSTANCE已经初始化了，第二个线程，第一个线程检查等于空的时候第二个线程检查也等于空，所以第二个线程在 *if(INSTANCE == null)* 这句话的时候停住了，暂停之后呢第一个线程已经把它初始化完了释放锁，第二个线程继续往下运行，往下运行的时候它会尝试拿这把锁，第一个线程已经释放了，它是可以拿到这把锁的，注意，拿到这把锁之后他还会进行一次检查，由于第一个线程已经把INSTANCE初始化了所以这个检查通过了，它不会在重新new一遍。因次，双重检查这个事儿是能够保证线程安全的。

就这个程序无论你运行多少遍，就算你在高并发的情况下运行，拿一百台机器同时访问这一台机子上的getInstance()，每个机器上跑个一万个线程，使劲儿跑，ok，这个程序运行的结果也会是正确的。

好，那么会有同学会说要不要加volatile？ 这是一道面试题：**你听说过单例模式吗，单例模式里面有一种叫双重检查的你了解吗，这个单例要不要加volatile？**答案是要加的，我们这个实验很难做出来

让它出错的情况，所以以前很多人就不加这个volatile他也不会出问题，不加volatile问题就会出现在指令重排序上，

第一个线程 *INSTANCE = new Mgr06()*经过我们的编译器编译之后呢的指令呢是分成三步 1.给指令申请内存 2.给成员变量初始化 3.是把这块内存的内容赋值给INSTANCE。既然有这个值了你在另外一个线程里头上来先去检查，你会发现这个值已经有了，你根本就不会进入锁那部分的代码。

加了volatile会怎么样呢，加了volatile指令重排序就不允许存在了。对这个对象上的指令重排序不允许存在，所以在这个时候一定是保证你初始化完了之后才会赋值给你这个变量，ok 这是volatile的含义。

```
/*** lazy loading* 也称懒汉式* 虽然达到了按需求初始化的目的，但却能带来线程不安全的问题* 可以通
```

三个步骤顺序有严格的规定吗，在JVM里面规定了八种原则，除了这些之外其他的指令都可以有重排序，保证原子性只是保证这些操作必须要么都完成之后其他才能访问，但是保证了原子性和保证重排序是两回事儿

到现在为止，volatile的两个含义已经说完了。

下面这个程序，如果不加volatile是一定会有问题的，结果是到不了10万的，原因很简单，count值改变之后只是被别的线程所看见，但是光看见没用，count++本身它不是一个原子性的操作，所以说volatile保证线程的可见性，并不能替代synchronized，保证不了原子性。要想解决这个问题，加上synchronized。

```
/*** volatile并不能保证多个线程共同修改running变量时所带来的的不一致问题，也就是说volatile不能
```

好，我们来看锁优化的一些问题，这个锁优化内容非常多啊，线程这块儿的内容也特别好玩儿，按顺序来讲是可能是1、2、3、4、5、6、7这样的内容，但是你要按照优化来讲，它这些优化可能会分布在不同的这个步骤之上，所以讲到某个优化的问题的时候我们就谈一次，讲到就谈一次。

锁优化其中有一个叫做把锁粒度变细 ，还有一个叫把锁粒度变粗，其实说的是一回事儿，什么意思呢，作为synchronized来说你这个锁呢征用不是很剧烈的前提下，你这个锁呢，粒度最好还是小一些。

下面程序是什么意思，如果是说m1方法他前面有一堆业务逻辑，后面有一堆业务逻辑，这个业务逻辑我用sleep来模拟了它，那么中间是你需要加锁的代码，那这个时候你不应该把锁加在整个方法上，只应该加在count++上(参见m2)，这很简单就叫做锁的细化。那什么时候需要将锁粗化呢，在征用特别频繁，由于你锁的粒度越变越细，好多小的细锁跑在你这个上面，这个方法，或者某一段业务逻辑里头，好，那你干脆不如弄成一把大锁，他的征用反而就没有那么频繁了，程序写的好，不会发生死锁。

```
/*** synchronized优化* 同步代码块中的语句越少越好* 比较m1和m2* @author mashibing*/package cc
```

下面有一个小概念，你在某一种特定的不小心的情况下你把o变成了别的对象了，这个时候线程的并发就会出问题。锁是在对象的头上两位来作为代表的，你这线程本来大家都去访问这两位了，结果突然把这把锁变成别的对象，去访问别的对象的两位了，这俩之间就没有任何关系了。因此，以对象作为锁的时候不让它发生改变，加fifinal。

```
/*** 锁定某对象o，如果o的属性发生改变，不影响锁的使用* 但是如果o变成另外一个对象，则锁定的对象发
```

好，那么到现在为止，我们volatile和synchronized都已经基本讲完了，稍微简单的回顾一下。

synchronized锁的是对象而不得代码，锁方法锁的是this，锁static方法锁的是class，锁定方法和非锁定方法是可以同时执行的，锁升级从偏向锁到自旋锁到重量级锁

volatile 保证线程的可见性，同时防止指令重排序。线程可见性在CPU的级别是用缓存一直性来保证的；禁止指令重排序CPU级别是你禁止不了的，那是人家内部运行的过程，提高效率的。但是在虚拟机级别你家volatile之后呢，这个指令重排序就可以禁止。严格来讲，还要去深究它的内部的话，它是加了读屏障和写屏障，这个是CPU的一个原语。

注：关于synchronized和volatile的底层实现，在老师的JVM课程中会有深入到CPU级别的讲解

# CAS

cas号称是无锁优化，或者叫自旋。这个名字无所谓，理解它是干什么的就行，概念这个东西是人为了描述问题解决问题而定义出来的，所以怎么定义不是很重要，重点是在解决问题上我们通过Atomic类(原子的)。由于某一些特别常见的操作，老是来回的加锁，加锁的情况特别多，所以干脆java就提供了这些常见的操作这么一些个类，这些类的内部就自动带了锁，当然这些锁的实现并不是synchronized重量级锁，而是CAS的操作来实现的(号称无锁)。

我们来举例几个简单的例子，凡是以Atomic开头的都是用CAS这种操作来保证线程安全的这么一些个类。AtomicInteger的意思就是里面包了一个Int类型，这个int类型的自增 *count++* 是线程安全的，还有拿值等等是线程安全的，由于我们在工作开发中经常性的有那种需求，一个值所有的线程共同访问它往上递增 ，所以jdk专门提供了这样的一些类。使用方法AtomicInteger如下代码

```
try{TimeUnit.SECONDS.sleep(3);}catch(InterruptedException e){e.printStackTrace();}//创建
```

好，我们来分许分析，它的内部实现的原理，主要是聊原理，它的用法看看API都会用。这个原理叫CAS操作，incrementAndGet() 调用了getAndAddInt

```
public final int incrementAndGet() { return U.getAndAddInt(this,VALUE,1)+1;}当然这个也是一
```

Unsafe这个类里面的方法CompareAndSetI(CAS),说一下字面意思，比较并且设定。这个比较并且设定的意思是什么呢，我原来想改变某一个值0 ，我想把它变成1，但是其中我想做到线程安全，就只能加锁synchronized ，不然线程就不安全。我现在可以用另外一种操作来替代这把锁，就是cas操作，你可以把它想象成一个方法，这个方法有三个参数，cas(V，Expected，NewValue)。


V第一个参数是要改的那个值；Expected第二个参数是期望当前的这个值会是几；NewValue要设定的新值。当前这个线程想改这个值的时候我期望你这值就是0，你不能是个1，如果是1就说明我这值不对，然后想把你变成1。这句话说的是什么意思呢，比如原来这个值变成3了，我这个线程想改这个值的时候我一定期望你现在是3 ，是3我才改，如果你在我改的过程中变成4了，那你跟我的期望值就对不上了，说明有另外一个线程改了这个值了，那我这个cas就重新再试一下，再试的时候我希望你这个值是4，在修改的时候期望值是4，没有其他的线程修改这个值，那好，我给你改成5，这就是cas操作，在本质上就是这么一个意思。

Expected如果对的上期望值，NewValue才会去对其修改，进行新的值设定的时候，这个过程之中来了一个线程把你的值改变了怎么办，我就可以再试一遍，或者失败，这个是cas操作。

当你判断的时候，发现是我期望的值，还没有进行新值设定的时候值发生了改变怎么办，cas是cpu的原语支持，也就是说cas操作是cpu指令级别上的支持，中间不能被打断。

**ABA问题**

一般的面试会问一下，了解这个ABA问题吗?

这个ABA问题是这样的，假如说你有一个值，我拿到这个值是1，想把它变成2，我拿到1用cas操作，期望值是1，准备变成2，这个对象Object，在这个过程中，没有一个线程改过我肯定是可以更改的，但是如果有一个线程先把这个1变成了2后来又变回1，中间值更改过，它不会影响我这个cas下面操作，这就是ABA问题。

这种问题怎么解决。如果是int类型的，最终值是你期望的，也没有关系，这种没关系可以不去管这个问题。如果你确实想管这个问题可以加版本号，做任何一个值的修改，修改完之后加一，后面检查的时候连带版本号一起检查。

如果是基础类型：无所谓。不影响结果值；

如果是引用类型：就像是你的女朋友和你分手之后又复合，中间经历了别的男人。

**Unsafe**


不需要加锁是怎么做到的呢，原因是使用了Unsafe这个类，关于这个类呢，你了解就行了，这个类里面的方法非常非常多，而且这个类除了用反射使用之外，其他不能直接使用，不能直接使用的原因，和ClassLoader是有关系的。先简单了解这个类。所有的Atomic操作内部下面都是CompareAndSetI这样的操作，那个CompareAndSetI就是在Unsafe这个类里面完成的。

回顾我们今天讲的内容，我们讲了volatile(1线程可见性，2指令重排序)；我们讲了CAS的原理，有人叫它无锁优化，有人叫乐观锁，cas会产生ABA问题；Unsafe了解。