在程序中使用多线程的目的是什么？

1、提高效率，增加任务的吞吐量

2、提升CPU等资源的利用率，减少CPU的空转

多线程的应用在日常开发中很多，带来了很多的便利，让我们以前研究下在多线程场景中要注意问题吧，一般主要从这三个方面考虑

1、原子性

2、可见性

3、有序性

如果不能保证原子性、可见性和顺序性会有什么问题？这些问题怎么解决呢？让我们一起来看下

1、原子性

原子性的操作是不可被中断的一个或一系列操作。

个人理解，严格的原子性的操作，其他线程获取操作的变量时，**只能获取操作前的变量值和操作后的变量值**，不能获取到操作过程中的中间值，在操作过程中其他操作需要获取变量值，需要进入**阻塞状态**等待操作结束。

如果不能保证原子性会有什么问题呢

让我们一块看一个简单例子吧

首先写一个简单的线程，代码如下

**public** **class** ThreadDemo **implements** Runnable{

**int** no = 0;

@Override

**public** **void** run() {

no++;

System.***out***.println(no);

}

**public** **static** **void** main(String[] args)

{

ThreadDemo demo = **new** ThreadDemo();

**for**(**int** i=0;i<1000;i++)

{

Thread task = **new** Thread(demo);

task.start();

}

}

}

Main函数中启动了1000个线程，每个线程都对no进行了一次+1操作，理想情况下no最后的结果应该是1000，可是我的运行最终结果只有996，执行了1000次+1操作，最后的结果为什么不是1000呢，这就要说到no++操作不是原子的问题了（no可见性的问题在下一个小结讨论）

先对no++这个操作分解，可以分为三步：取值，加一，赋值，这三个操作都是原子的，不过合在一起就不行了。两个线程A、B一起来操作no，no初始值是1,线程A读取no值是1，然后做no+1，这时线程B对 no取值还是1，然后A将2赋值给no，B操作no+1结果是2，也将2赋值给no。对1做了两次+1操作，最后结果是2。这个过程可以参考下图



所以，需要保证这种不可分割操作的原子性，那要怎么做才能保证原子性呢，有两种方式

1、加锁synchronized，保证同一时间只有一个线程操作变量，其他线程需等待操作结束才能使用临界资源

2、使用CAS操作，变量计算前保留一份旧值a，计算完成后结果值为b，把b刷到内存之前先比较a是否和内存中变量一致，如果一致，就把内存中的变量赋值为b，不一样，重新获取内存中变量值，重复一遍操作，一直到a和内存中一致，操作结束。

Lock和原子类（AtomicInteger等）是通过使用unsafe的compareAndSwap方法实现CAS操作保证原子性的。

二、可见性

线程变量的可见性问题，需要从操作系统的CPU、缓存、内存的矛盾开始说起。读写性能上 CPU>缓存>内存>I/O。CPU/缓存/内存的结构看下图。



CPU和内存之间隔着缓存和CPU寄存器。缓存还分为一级、二级、三级缓存。CPU的读写性能上要大于内存，为了提高效率会将数据先取到缓存中，CPU处理完数据后会先放到缓存中，然后同步到内存中。

如果不理解CPU缓存这部分内容的话，可以简单的认为每个线程都有自己的本地工作内存，变量会先缓存到本地工作内存中使用，修改后会先修改工作内存中的存储，然后在同步到主内存中。结构如下图



这种内存结构会引起什么问题呢，现在有一个变量var，线程A对var做了一次修改，刚放到缓存（工作内存）还未同步到内存时，另外一个线程B也来使用var，读取到的还是var未修改值。

共享的变量需要保证可见性，怎么保证共享变量的可见性呢

1、加锁（加锁是万能的操作）synchronized和Lock都可以保证。

线程在加锁时，会清空工作内存中共享变量的值，共享变量使用是需要从主内存中重新获取。

线程解锁是，会把共享变量重新刷新到主内存中。

2、使用volatile修饰共享变量，volatile修饰的共享变量在修改后会立即被更新到内存中，其他线程使用共享变量会去内存中读取

优先使用volatile来解决可见性问题，加锁需要消耗的资源太多。

3、有序性

为了优化程序性能，编译器、处理器和运行时会对代码指令进行重排，重排过程中会遵循as-if-serial语义，即不影响单线程的运行结果。

扩展一下 指令重排为什么会提高程序性能呢，我个人理解是CPU是多核处理的，为了保证处理器资源的充分利用，对代码指令进行乱序处理，即可以多个处理器并行处理指令，防止不相关的指令需要等待上一个指令结束才能开始。

代码执行顺序被重排会是什么效果呢，举个简单例子

**int** a =0;

a=2;

**int** b =1;

**int** c= a+b;

这段代码中变量a和b 是相互不影响的，优化后可以是如下代码，只要保证执行结果不变，有依赖的变量c在变量a和b之后处理即可

**int** b =1;

**int** a =0;

a=2;

**int** c= a+b;

在单线程中这样是没有问题的，如果是多线程呢，看下如下代码

**public** **static** **void** main(String[] args)

{

Object val =**null**;

**boolean** finish =**false**;

Runnable task1 = **new** Runnable() {

Object val;

**boolean** finish;

**public** **void** setVal(Object val,**boolean** finish){**this**.val= val;**this**.finish = finish;}

@Override

**public** **void** run() {

**if**(**this**.finish)

{

System.*out*.println(**this**.val.toString());

}

}

};

Runnable task2 = **new** Runnable() {

Object val;

**boolean** finish;

**public** **void** setVal(Object val,**boolean** finish){**this**.val= val;**this**.finish = finish;}

@Override

**public** **void** run() {

val = **new** Object();

finish =**true**;

}

};

}

分别创建两个任务task1和task2,他们共享两个变量val和finish，按现在的顺序执行的话，task1不会出现val为null时被使用的情况。不过进行了指令重排之后呢，task2中val和finish操作顺序调整对单线程来说是没有任何影响的，所以task2的代码可能会变成

@Override

**public** **void** run() {

finish =**true**;

val = **new** Object();

}

这样task1中，就会出现finish为true，val为null的情况了。

那么怎么保证多个线程中的代码顺序一致性呢

1、加锁（还是加锁）synchronized和Lock，保证同一时刻只有一个线程进行操作

2、使用volatile修饰变量，在JMM中volatile的读和写都会插入内存屏障来禁止处理器的重排

这样原子性、可见性、有序性就基本讲完了，其中有很多的知识点没有详细的说，例如：

CAS、volatile、synchronized、lock等等，这些会在后边的文章中慢慢研究。

注：**如果弄不清楚原子性和可见性的区别，只要记住下边两点内容**

1、原子性针对完整的操作过程，其他操作只能获取操作前或操作后的变量数据

2、可见性主要是变量修改，变量修改后，马上刷新到内存中，而其他线程能感知到变量的修改