我们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

多线程在现在开发中用的很频繁，带来了很多的遍历，那使用多线程要注意问题呢，一起从这三个方面研究一下吧

1、原子性

2、可见性

3、顺序性

让我们分别看一下原子性、可见性和顺序性带来的问题和解决方案吧

1、什么是原子性，不保证原子性会有什么问题，保证原子性的解决方案有哪些

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，其他操作获取对应的变量的时候，**只能获取操作前的资源和操作后的变量值**，在操作过程中如果需要获取变量，将进入**阻塞状态**等待操作结束。

多线程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

首先实现一个Runnable

**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，可是我的运行结果只有996，明明执行了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可见性的问题在后边讨论）

对no++这个操作进行一下分解，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。那么线程A先取到no的值是1，然后对no+1，这时线程B来了拿到no的值也是1，然后A将2赋值给no，B操作no+1结果是2，将2赋值给no。对no做了两次+1操作，最后结果是2。这个过程可以参考下图



现在明白不保证原子性带来的问题的，那要怎么做才能保证每次+1操作的过程中不会被其他操作影响呢，有两种方式

1、加锁synchronized，保证只有一个线程在操作变量

2、使用CAS（Compare And Set）操作，变量操作前保留一份旧值A，操作完成后变量变为B，赋值之前比较A是否和内存中变量一致，如果一致，就把变量赋值为B，不一样，就重新取值重新走一遍操作流程，然后在比较赋值。

二、可见性是怎么引起的，怎么来解决呢

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。从性能上比较 CPU>缓存>内存>I/O。CPU/缓存/内存的结构可以用下图表示。



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

这个过程中如果变量A被CPU修改以后，放到了缓存中还没同步到内存，另外一个线程使用A，获取到的还是A原来的值。

不了解CPU缓存这部分话，可以理解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，修改后会先放到工作内存中，然后在同步到主内存中。结构如下图



Synchronized和cas