

volatile和synchronized详解

volatile和synchronized详解

一、JMM简介

- 1.什么是JMM?
- 2.JMM的三大特性:
 - 1.原子性
 - 2.可见性
 - 3.有序性
- 3.关于同步的规定:
- 4.解释说明

二、volatile关键字

- 1.volatile简介:
- 2.三大特性
 - 1>.保证可见性:
 - 2>.保证有序性(禁止指令重排序)
有序性的实现原理:
 - 3>.不保证原子性:
解决方式:

三、synchronized关键字

- 1.synchronized简介:
锁的类型:
- 2.实际应用:
- 3.实现原理
 - ObjectMonitor中的关键属性:
 - 同步方法和同步代码块的实现原理:
- 4.三大特性:
- 5.锁升级过程:
 - synchronized锁优化:
 - synchronized锁升级:

四、volatile和synchronized的区别:

五、实际应用

- 多线程下的单例模式:

一、JMM 简介

1.什么是JMM?

JMM 是Java内存模型（Java Memory Model），简称JMM。它本身只是一个抽象的概念，并不真实存在，它描述的是一种规则或规范。通过这组规范，定义了程序中对各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。

计算机在执行程序时，每条指令都是在CPU中执行的。而执行指令的过程中，势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程，跟CPU执行指令的速度比起来要慢的多（硬盘 < 内存 < 缓存cache < CPU）。因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。也就是当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时，就可以直接从它的高速缓存中读取数据或向其写入数据了。当运算结束之后，再将高速缓存中的数据刷新到主存当中。

2.JMM的三大特性：

1. 原子性

一个或多个操作，要么全部执行（执行的过程是不会被任何因素打断的），要么全部不执行。

2.可见性

只要有一个线程对共享变量的值做了修改，其他线程都将马上收到通知，立即获得最新值。

3.有序性

有序性可以总结为：在本线程内观察，所有的操作都是有序的；而在一个线程内观察另一个线程，所有操作都是无序的。前半句指 as-if-serial 语义：线程内似表现为串行，后半句是指：“指令重排序现象”和“工作内存与主内存同步延迟现象”。处理器为了提高程序的运行效率，提高并行效率，可能会对代码进行优化。编译器认为，重排序后的代码执行效率更优。这样一来，代码的执行顺序就未必是编写代码时候的顺序了，在多线程的情况下就可能会出错。

在代码顺序结构中，我们可以直观的指定代码的执行顺序，即从上到下按序执行。但编译器和CPU处理器会根据自己的决策，对代码的执行顺序进行重新排序，优化指令的执行顺序，提升程序的性能和执行速度，使语句执行顺序发生改变，出现重排序，但最终结果看起来没什么变化（在单线程情况下）。

有序性问题 指的是在多线程的环境下，由于执行语句重排序后，重排序的这一部分没有一起执行完，就切换到了其它线程，导致计算结果与预期不符的问题。这就是编译器的编译优化给并发编程带来的程序有序性问题。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行进入。

3.关于同步的规定：

- 1.线程解锁前，必须把共享变量的值刷新回主内存。
- 2.线程加锁前，必须将主内存的最新值读取到自己的工作内存。
- 3.加锁解锁是同一把锁。

4.解释说明

在JVM中，栈负责运行（主要是方法），堆负责存储（比如new的对象）。由于JVM运行程序的实体是线程，而每个线程在创建时，JVM都会为其创建一个工作内存（有些地方称为栈空间），工作内存是每个线程的私有数据区域。而JAVA内存模型中规定，所有变量都存储在主内存中，主内存是共享内存区域，所有线程都可以访问。

但线程对变量的操作（读取赋值等）必须在自己的工作内存中进行。首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后，再将变量写回到主内存。由于不能直接操作主内存中的变量，各个线程的工作内存中存储着主内存中的变量副本，因此，不同的线程之间无法直接访问对方的工作内存，线程间的通信（传值）必须通过主内存来完成。

二、volatile关键字

1.volatile简介：

volatile 是 JVM 提供的轻量级的同步机制。volatile 关键字可以保证并发编程三大特征（原子性、可见性、有序性）中的**可见性和有序性，不能保证原子性**。

2.三大特性

1>.保证可见性：

加了volatile关键字修饰的变量，只要有一个线程将主内存中的变量值做了修改，其他线程都将马上收到通知，立即获得最新值。当写线程写一个volatile变量时，JMM会把该线程对应的本地工作内存中的共享变量值刷新到主内存。当读线程读一个volatile变量时，JMM会把该线程对应的本地工作内存置为无效，线程将到主内存中重新读取共享变量。

volatile语义实现原理：

先来看两个与CPU相关的专业术语：

- 内存屏障（memory barriers）：一组处理器指令，用于实现对内存操作的顺序限制。
- 缓存行（cache line）：CPU高速缓存中可以分配的最小存储单位。处理器填写缓存行时会加载整个缓存行。

volatile可见性的实现是借助了CPU的lock指令，lock指令在多核处理器下，可以将当前处理器的缓存行的数据写回到系统内存，同时使其他CPU里缓存了该内存地址的数据置为无效。通过在写volatile的机器指令前加上lock前缀，使写volatile具有以下两个原则：

- 1. 写volatile时处理器会将缓存写回到主内存。
- 2. 一个处理器的缓存写回到内存，会导致其他处理器的缓存失效。

代码验证：

例如：

int number = 0; 此时number变量是没有可见性的。

volatile int number = 0; 前面添加了volatile关键字之后，可以解决可见性问题。

没加volatile关键字之前：

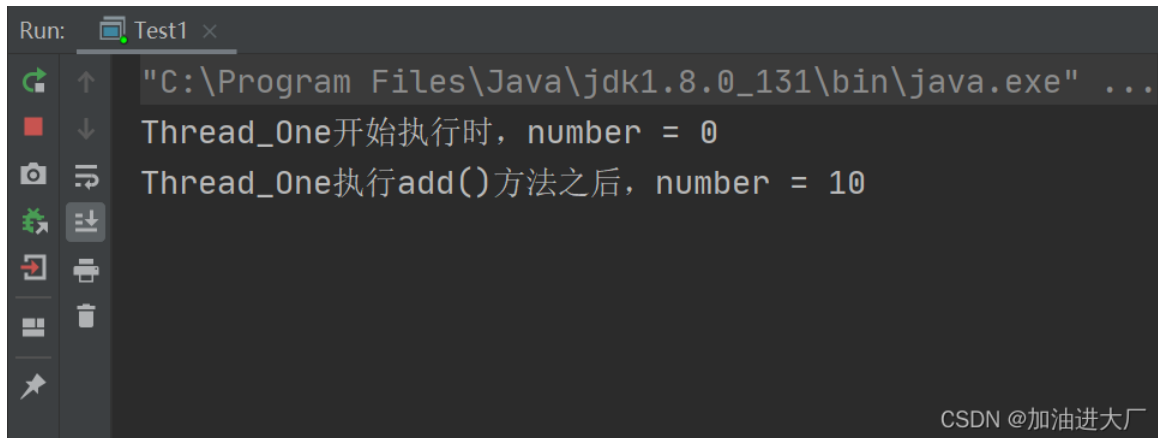
```
1  /**
2   * 普通类:
3   * 为了验证volatile的可见性
4   */
5  public class Test1 {
6      int number = 0;
7
8      public void add(){
9          this.number = 10;
10     }
11
12
13     public static void main(String[] args) {
14         Test1 test1 = new Test1();
15     }
```

```

16
17 //创建第一个线程
18 new Thread(() -> {
19     System.out.println(Thread.currentThread().getName()+"开始执行时, number = "+test1.number);
20
21     try{ Thread.sleep(3000);}catch (Exception e){e.printStackTrace();}
22     test1.add();//暂停3秒后, 修改number的值。
23     System.out.println(Thread.currentThread().getName()+"执行add()方法之后, number = "+test1.number);
24
25 }, "Thread_One").start();
26
27
28 //第二个是main线程
29 while (test1.number == 0){
30     //如果第二个main线程 可以监测到number值的改变, 就会跳出当前循环, 执行后续程序。
31 }
32
33 System.out.println(Thread.currentThread().getName()+"程序结束! ");
34
35 }
}

```

运行结果: (程序卡死在 循环while (test1.number == 0)里, 跳不出来)



Run: Test1 ×

```

"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
Thread_One开始执行时, number = 0
Thread_One执行add()方法之后, number = 10

```

CSDN @加油进大厂

加上volatile关键字之后:

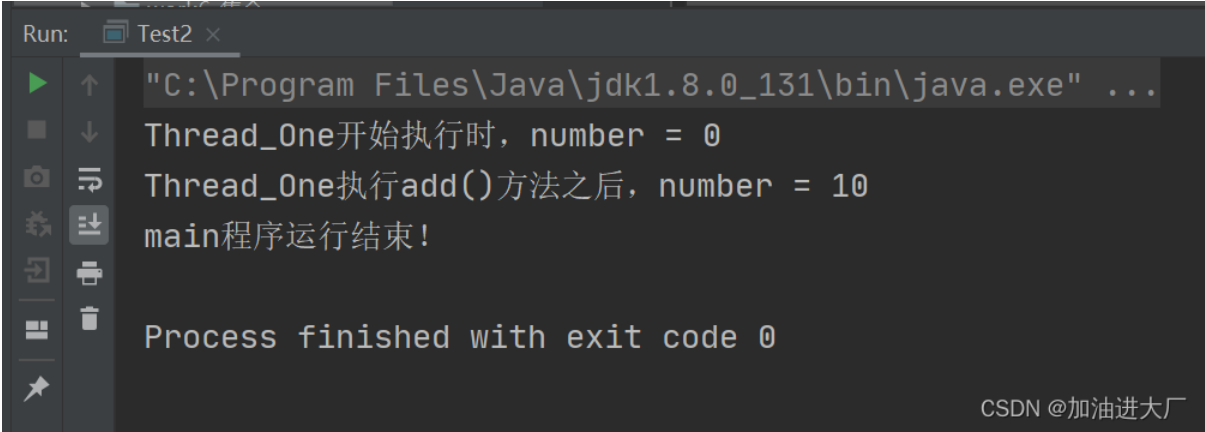
```

1 /**
2  * 变量上加了volatile关键字:
3  * 为了验证volatile的可见性。
4  */
5 public class Test2 {
6     volatile int number = 0;
7
8     public void add(){
9         this.number = 10;
10    }
11
12
13    public static void main(String[] args) {
14        Test2 test2 = new Test2();
15
16        //创建第一个线程
17        new Thread(() -> {
18            System.out.println(Thread.currentThread().getName()+"开始执行时, number = "+test2.number);
19
20            try{ Thread.sleep(3000);}catch (Exception e){e.printStackTrace();}
21            test2.add();//暂停3秒后, 修改number的值。
22            System.out.println(Thread.currentThread().getName()+"执行add()方法之后, number = "+test2.number);
23
24        }, "Thread_One").start();
25
26
27        //第二个是main线程
28        while (test2.number == 0){
29            //由于变量number上加了volatile关键字,
30            // 使得第二个main线程可以监测到number值的改变, 从而跳出了循环。
31        }
32
33        System.out.println(Thread.currentThread().getName()+"程序运行结束! ");
34
35
36

```

```
}  
}  
}
```

运行结果: (循环while (test1.number == 0)可以正常结束)



2>.保证有序性(禁止指令重排序)

简单说明:

计算机在执行程序时, 为了提高计算性能, 编译器和处理器常常会对指令进行重排序, 一般分为如下3种:

源代码 ——> 编译器优化的重排 ——> 指令并行的重排 ——> 内存系统的重排 ——> 最终执行的指令



解释说明:

- 单线程环境下, 可以确保程序最终执行结果和代码顺序执行结果的一致性(单线程环境下不用关注指令重排, 因为是否重排都不会出错)。处理器在进行重排序时, 必须要考虑指令之间的数据依赖性。
- 多线程环境中, 线程交替执行。由于编译器优化重排的存在, 两个线程中使用的变量能否保证一致性是无法确定的, 结果也就无法预测。而用volatile关键字修饰的变量, 可以禁止指令重排序, 从而避免多线程环境下, 程序出现乱序执行的现象。

有序性的实现原理:

volatile有序性的保证就是通过禁止指令重排序来实现的。指令重排序包括编译器和处理器重排序, JMM会分别限制这两种指令重排序。

volatile通过加内存屏障来实现禁止指令重排序。JMM为volatile加内存屏障有以下4种情况:

- 在每个volatile写操作的前面插入一个StoreStore屏障, 防止写volatile与后面的写操作重排序。
- 在每个volatile写操作的后面插入一个StoreLoad屏障, 防止写volatile与后面的读操作重排序。
- 在每个volatile读操作的后面插入一个LoadLoad屏障, 防止读volatile与后面的读操作重排序。
- 在每个volatile读操作的后面插入一个LoadStore屏障, 防止读volatile与后面的写操作重排序。

volatile写是在前面和后面分别插入内存屏障, 而volatile 读操作是在后面插入两个内存屏障。

内存屏障	解释说明
StoreStore屏障	禁止上面的普通写和下面的volatile 写重排序。
StoreLoad屏障	防止上面的volatile 写与下面可能存在的volatile 读/写重排序。
LoadLoad屏障	禁止下面所有的普通读操作和上面的volatile读重排序。
LoadStore屏障	禁止下面所有的普通写操作和上面的volatile读重排序。

3>.不保证原子性:

原子性指的是, 当某个线程正在执行某件事情的过程中, 是不允许被外来线程打断的。也就是说, 原子性的特点是要么不执行, 一旦执行就必须全部执行完毕。而volatile是不能保证原子性的, 即执行过程中是可以被其他线程打断甚至是加塞的。

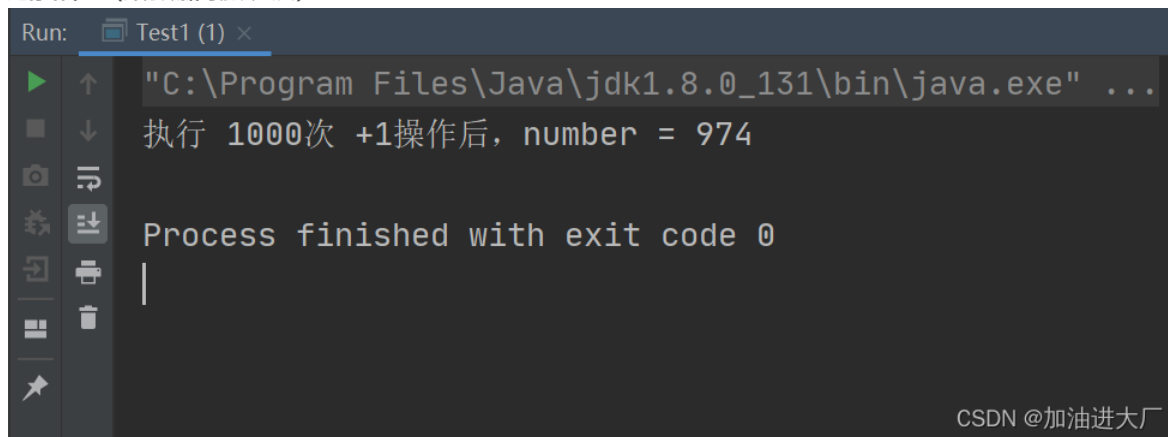
所以, volatile变量的原子性与synchronized的原子性是不同的。synchronized的原子性是指, 只要声明为synchronized的方法或代码块, 在执行上就是原子操作的。而volatile是不修饰方法或代码块的, 它只用来修饰变量, 对于单个volatile变量的读和写操作都具有原子性, 但类似于volatile++这种复合操作不具有原子性。所以volatile的原子性是受限制的。并且在多线程环境中, volatile并不能保证原子性。

代码验证:

加上volatile关键字，不能保证原子性:

```
1  /**
2   * 变量上加了volatile关键字:
3   * 为了验证volatile的 不保证原子性。
4   */
5  public class Test1 {
6
7      volatile int number = 0;
8
9      public void add(){
10         number++;
11     }
12
13
14     public static void main(String[] args) {
15         Test1 test1 = new Test1();
16
17         //创建10个线程
18         for (int i = 0; i < 10; i++){
19             new Thread(() -> {
20                 //每个线程执行1001次+1操作
21                 for (int j = 0; j < 100; j++){
22                     test1.add();
23                 }
24             }, "Thread_" + (i+1)).start();
25         }
26
27         //如果正在运行的线程数>2个(除了main线程和GC线程以外，还有其他线程正在运行)
28         while(Thread.activeCount() > 2){
29             Thread.yield(); //礼让其他线程，暂不执行后续程序
30         }
31
32         System.out.println("执行 1000次 +1操作后, number = "+test1.number);
33     }
34 }
35 }
36 }
```

运行结果: (部分数据可能会丢失)



解决方式:

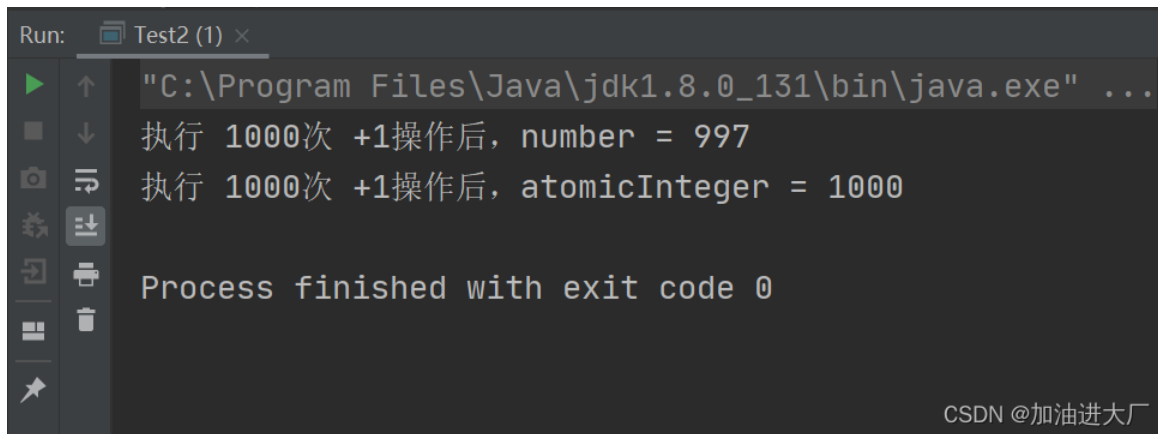
方式一: 方法上加 synchronized 关键字。

方式二: 利用AtomicInteger类实现原子性。代码如下:

```
1  import java.util.concurrent.atomic.AtomicInteger;
2
3  /**
4   * 变量上加了volatile关键字,
5   * 但 不能保证原子性 的 解决方式。
6   */
7  public class Test2 {
8
9      volatile int number = 0;
10
11     //解决方式一: 方法上加 synchronized 关键字
12     public void add(){
13         number++;
14     }
15 }
```

```
14     }
15
16     //解决方式二: 如下
17     AtomicInteger atomicInteger = new AtomicInteger();
18     public void addMyAtomic(){
19         //每调用一次此方法, 加个一。
20         atomicInteger.getAndIncrement();
21     }
22
23
24
25     public static void main(String[] args) {
26         Test2 test2 = new Test2();
27
28
29         //创建10个线程
30         for (int i = 0; i < 10; i++){
31             new Thread() -> {
32                 //每个线程执行1001次+1操作
33                 for (int j = 0; j < 100; j++){
34                     test2.add(); //调用不能保证原子性的方法
35                     test2.addMyAtomic(); //调用可以保证原子性的方法。
36                 }
37             }, "Thread_" + (i+1)).start();
38         }
39
40         //如果正在运行的线程数>2个(除了main线程和GC线程以外, 还有其他线程正在运行)
41         while(Thread.activeCount() > 2){
42             Thread.yield(); //礼让其他线程, 暂不执行后续程序
43         }
44
45         System.out.println("执行 1000次 +1操作后, number = " + test2.number);
46         System.out.println("执行 1000次 +1操作后, atomicInteger = " + test2.atomicInteger);
47     }
48 }
49 }
```

运行结果对比:



```
Run: Test2 (1) x
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
执行 1000次 +1操作后, number = 997
执行 1000次 +1操作后, atomicInteger = 1000
Process finished with exit code 0
CSDN @加油进大厂
```

三、synchronized关键字

1.synchronized简介:

synchronized是Java多线程中经常使用的一个关键字。synchronized可以保证原子性、可见性、有序性。它包括两种用法:synchronized 方法和 synchronized 代码块。它可以用来给对象、方法或代码块进行加锁。当它锁定一个方法或者一个代码块时,同一时刻最多只有一个线程可以执行这段代码,其他线程想在此时调用该方法只能排队等候。当它锁定一个对象时,同一时刻最多只有一个线程可以对这个类进行操作,没有获得锁的线程,在该类所有对象上的任何操作都不能进行。

锁的类型:

- **synchronized 是悲观锁的实现**,因为 synchronized 修饰的代码,每次执行时都会进行加锁操作,同时只允许一个线程进行操作,所以它是悲观锁的实现。
- **synchronized 是非公平锁,并且是不可设置的**。这是因为非公平锁的吞吐量大于公平锁,并且是主流操作系统线程调度的基本选择,所以这也是 synchronized 使用非公平锁原因。
- 同时, **synchronized是一个典型的可重入锁**,可重入锁最大的作用是避免死锁。

2.实际应用:

例1:

```
1 public synchronized void synMethod(){
2     //方法体
3 }
```

这时,线程获得的是成员锁,即一次只能有一个线程进入该方法,其他线程要想在此时调用该方法,只能排队等候,当前线程(就是在synchronized方法内部的线程)执行完该方法后,别的线程才能进入。

例2:

```
1 public Object synMethod(Object a1){
2     synchronized(a1){
3         //一次只能有一个线程进入
4     }
5 }
```

对某一代码块使用 synchronized后跟括号,括号里是变量。如上所示,此时,一次只有一个线程进入该代码块,此时,线程获得的是成员锁。

例3:

```
1 public class MyThread implements Runnable{
2     public static void main(String args[]){
3         Thread t1=new Thread(mt,"t1");
4         Thread t2=new Thread(mt,"t2");
5         t1.start();
6         t2.start();
7     }
8     public void run(){
9         synchronized(this){
10             System.out.println(Thread.currentThread().getName());
11         }
12     }
13 }
```

如果synchronized后面括号里是一个对象,此时,线程获得的是对象锁。如果线程进入,则得到当前对象锁,那么其他没有获得锁的线程,在该类所有对象上的任何操作都不能进行。

例4:

```
1 class ArrayWithLockOrder{
2     public ArrayWithLockOrder(int[] a){
3         synchronized(ArrayWithLockOrder.class){
4             //代码逻辑
5         }
6     }
7 }
```

如果synchronized后面括号里是类,此时线程获得的是对象锁。如果其他线程进入,则线程在该类中所有操作不能进行,包括静态变量和静态方法。实际上,对于含有静态方法和静态变量的代码块的同步,我们通常选用例4来加锁。

3. 实现原理

为了解决线程安全问题, Java提供了同步机制、互斥锁机制,这个机制保证了在同一时刻只有一个线程能访问共享资源。这个机制的保障来源于监视锁Monitor。每个Object对象中都内置了一个monitor对象,monitor对象存在于每个Java对象的对象头中(存储的是指针)。monitor相当于一个许可证,线程拿到许可证即可以进行操作,没有拿到则需要阻塞等待。任何对象都有一个monitor与之相关联,当monitor被持有之后,他将处于锁定状态。线程执行到monitorenter指令时,会尝试获取对象所对应的monitor的所有权,即尝试获取对象的锁。

•ObjectMonitor中的关键属性:

_owner: 指向持有ObjectMonitor对象的线程。

_WaitSet: 存放处于wait状态的线程队列。

_EntryList: 存放处于等待锁block状态的线程队列。

_recursions: 锁的重入次数。

_count: 用来记录该线程获取锁的次数。

解释说明:

·当多个线程同时访问,这些线程会被放进_EntryList队列,此时线程处于blocked状态。

·当一个线程获得了对象的monitor后,就可以进入running状态执行方法,此时, ObjectMonitor对象的 _Owner指向当前线程, _count加1表示当前对象锁被一个线程获取。

·如果running状态的线程调用wait()方法时,当前线程释放monitor对象,进入waiting状态, ObjectMonitor对象的 _owner变为null, _count减1,同时线程进入 _WaitSet队列。直到有线程调用notify()方法唤醒该线程,则该线程进入 _EntryList队列,竞争到锁再进入 _Owner区。

·如果当前线程执行完毕，那么也释放monitor对象，ObjectMonitor对象的_owner变为null，_count减1

•同步方法和同步代码块的实现原理：

同步代码块：

monitorenter指令插入到同步代码块的开始位置，monitorexit指令插入到同步代码块的结束位置，JVM需要保证每一个monitorenter都有一个monitorexit与之相对应。执行monitorenter指令进入同步代码块，执行monitorexit指令退出同步代码块。但实际应用中，往往是有2个monitorexit指令，第一个是正常退出时执行的，第二个是发生异常时执行的。

同步方法：

synchronized方法的字节码中，并没有看到monitorenter和monitorexit指令。synchronized方法会被翻译成普通方法的调用和返回指令，如:invokevirtual、areturn指令。在JVM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件的方法表中，会将该方法的access_flags字段中的synchronized标志为1，表示该方法是同步方法。并将调用该方法的对象或该方法所属的Class，在JVM的内部对象表示Klass做为锁对象。

4.三大特性：

synchronized保证原子性：

- 1.通过monitorenter和monitorexit指令，可以保证被synchronized修饰的代码在同一时间只能被一个线程访问，在锁未释放之前，无法被其他线程访问到。
- 2.即使在执行过程中，由于某种原因，比如CPU时间片用完，线程1放弃了CPU，但是它并没有进行解锁。而由于synchronized的锁是可重入的，这就保证下一个时间片还是只能被他自己获取到，还是会继续执行代码。直到所有代码执行完为止。

synchronized保证可见性：

对于一个被synchronized修饰的变量，在其解锁之前，必须先要把此变量同步回主存当中。

synchronized保证有序性：

尽管synchronized无法禁止指令重排和处理器优化，但是可以通过单线程机制来保证有序性。由于synchronized修饰的代码，在同一时刻只能被同一线程访问，从根本上避免了多线程的情况。而单线程环境下，在本线程内观察到的所有操作都是天然有序的，所以synchronized可以通过单线程的方式来保证程序的有序性。

5.锁升级过程：

synchronized锁优化：

在Java的早期版本中，synchronized属于重量级锁，效率低下，因为操作系统实现线程之间的切换时，需要从用户态转换到核心态，这个状态之间的转换需要较长的时间，时间成本相对较高。在jdk1.6之后，Java官方从JVM层面对synchronized进行优化。为了减少获得锁和释放锁所带来的性能消耗，引入了偏向锁和轻量级锁。其中，synchronized锁升级是锁优化的一种具体实现方式。

synchronized锁升级：

synchronized锁升级：无锁 → 偏向锁 → 轻量级锁 → 重量级锁

无锁 → 偏向锁：

当锁对象第一次被线程获取的时候，虚拟机会将锁对象的对象头中的锁标志位设置为01，并将偏向锁标志设置为1。线程通过CAS的方式将自己的ID值放置到对象头中（因为在这个过程中有可能会有其他线程来竞争锁，所以要通过CAS的方式，一旦有竞争就会升级为轻量级锁）。如果成功，线程就成功的获得到了该对象的偏向锁。这样每次再进入该锁对象的时候，就不用进行任何的同步操作，直接比较当前锁对象的对象头是不是该线程的ID，如果是就可以直接进入。

偏向锁 → 轻量级锁：

偏向锁是一种无竞争锁，一旦出现了竞争，大多数情况下就会升级为轻量级锁。现在我们假设有线程1持有偏向锁，线程2来竞争偏向锁，会经历以下几个过程：

- 1. 首先线程2会先检查偏向锁标记，如果是1，说明当前是偏向锁，那么JVM会找到线程1，查看线程1是否还活着。
- 2. 如果线程1已经执行完毕，即线程1已经不存在了（线程1自己不会主动去释放偏向锁），那么先将偏向锁置为0，对象头设置为无锁的状态，用CAS的方式尝试将线程2的ID放入到对象头中，不进行锁升级，还是偏向锁。
- 3. 如果线程1还活着，就先暂停线程1，将锁标志位变成00（轻量级锁），将偏向锁变成了轻量级锁，然后继续执行线程1，此时线程2采用CAS的方式尝试获取锁。

轻量级锁 → 重量级锁：

一旦竞争加剧（如自旋次数或自旋线程数超过阈值），轻量级锁就会膨胀为重量级锁，锁的状态变成10。此时，对象头中存储的就是指向重量级锁的栈帧的指针。而其他等待锁的线程要进入阻塞状态，等重量级锁被释放后，再被唤醒，然后去竞争锁。

重型锁可以认为是，直接对应操作系统底层中的互斥量，通过使用互斥量来进行锁的竞争。由于直接使用底层操作系统的调度，会消耗大量性能，所以称之为重型锁

锁	优点	缺点	使用场景
偏向锁	加锁和解锁不需要CAS操作；没有额外的性能消耗；和执行非同步方法相比，仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于基本没有线程竞争锁的同步场景。
轻量级锁	竞争的线程不会阻塞，使用自旋，提高了程序的响应速度。	如果线程一直不能获取锁,长时间的自旋，会造成CPU的消耗。	适用于少量线程竞争锁对象,且线程持有锁的时间不长，同步块执行速度较快，追求响应速度的场景。
重量级锁	线程竞争不使用CPU自旋，不会因为CPU空转而消耗CPU资源。	线程阻塞，响应时间较长，在多线程下，频繁的获取锁、释放锁，会带来巨大的性能消耗。	很多线程竞争锁,且锁持有的时间较长，追求吞吐量的场景。

四、volatile和synchronized的区别：

应用范围：

volatile关键字是对变量进行上锁，锁住的是单个变量，而synchronized还能对方法以及代码块进行上锁。

是否保证原子性：

在多线程环境下，volatile可以保证可见性和有序性，不能保证原子性，而synchronized在保证可见性和有序性的基础上，还可以保证原子性。

volatile变量的原子性与synchronized的原子性是不同的。synchronized的原子性是指，只要声明为synchronized的方法或代码块，在执行上就是原子操作，synchronized能保证被锁住的整个代码块的原子性。而volatile是不修饰方法或代码块的，它只用来修饰变量，对于单个volatile变量的读和写操作都具有原子性，但类似于volatile++这种复合操作不具有原子性。所以volatile的原子性是受限制的。所以，在多线程环境中，volatile并不能保证原子性。

使用场景：

volatile主要用于解决共享变量的数据可见性问题，而synchronized主要用于保证访问数据的同步性（同时也能保证可见性）。

保证有序性的方式：

volatile的有序性是通过禁止指令重排序来实现的。synchronized无法禁止指令重排，但是可以通过单线程机制来保证有序性。由于synchronized修饰的代码，在同一时刻只能被一个线程访问，从根本上避免了多线程的情况。而单线程环境下，在本线程内观察到的所有操作都是天然有序的，所以synchronized可以通过单线程的方式来保证程序的有序性。

性能方面：

volatile是线程同步的轻量级实现，性能高于synchronized。多线程访问volatile修饰的变量时不会发生阻塞（主要是因为volatile采用CAS加锁），而访问synchronized修饰的资源时会发生阻塞。

五、实际应用

•多线程下的单例模式：

关于单例模式的详细介绍：[单例模式（Singleton）](#)

这里只是简单的介绍一下：

```
1  /**
2   * 单例模式 singleton :
3   * 实现此类只能创建一个唯一的实例对象。
4   */
5  public class Singleton {
6
7      /**
8       * 1. 私有的构造方法,
9       * 可以确保外部类无法通过构造方法随便new出新的实例对象
10     */
11     private Singleton(){}
12
13
14     /**
15      * 2. 将唯一的一个 实例对象instance，作为属性。
16      *
17      * 静态的static，可以保证此实例对象是唯一存在的，
18      * 且外部类不需要通过创建此Singleton类，就可以直接获得instance对象。
19      *
20      * 私有的private，可保证其他外部类，不能随便使用这个唯一的实例对象，保证数据的安全。
21      *
22      * 添加 volatile :防止在给属性赋值的时候，JVM指定重排序(禁止指令重排序)。
23      */
24     private static volatile Singleton instance ;
25
26
27
28     /**
29      * 3. 向外部类 提供 可以获得这个唯一实例对象的方法。
30      *
31      * 之所以是 静态的static，也是为了让外部类不需要创建对象，就可以直接使用这个方法。
32      *
33      * 为临界资源添加锁 synchronized，
34      * 既 解决了 多线程的安全问题，又提高了工作效率。
35      */
36     public static Singleton getInstance(){
37         if (instance == null){
38             /**      锁定的资源 为当前Singleton类      */
39             synchronized(Singleton.class){
40                 if (instance == null){
41                     System.out.println("创建了一个对象！");
42                     instance = new Singleton();
43                 }
44             }
45         }
46     }
```

```
47 |         return instance;
    |     }

```

文章知识点与官方知识档案匹配，可进一步学习相关知识

Java技能树 首页 概览 139739 人正在系统学习中

“相关推荐”对你有帮助么？

-  非常没帮助
-  没帮助
-  一般
-  有帮助
-  非常有帮助