

Java 并发编程：volatile的使用及其原理

Java并发编程系列：

- Java 并发编程：核心理论
- Java并发编程：Synchronized及其实现原理
- Java并发编程：Synchronized底层优化（轻量级锁、偏向锁）
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join).
- Java 并发编程：volatile的使用及其原理

一、volatile的作用

在《Java并发编程：核心理论》一文中，我们已经提到过可见性、有序性及原子性问题，通常情况下我们可以通过Synchronized关键字来解决这些问题，不过如果对Synchronized原理有了解的话，应该知道Synchronized是一个比较重量级的操作，对系统的性能有比较大的影响，所以，如果有其他解决方案，我们通常都避免使用Synchronized来解决问题。而volatile关键字就是Java中提供的另一种解决可见性和有序性问题的方案。对于原子性，需要强调一点，也是大家容易误解的一点：对volatile变量的单次读/写操作可以保证原子性的，如long和double类型变量，但是并不能保证i++这种操作的原子性，因为本质上i++是读、写两次操作。

二、volatile的使用

关于volatile的使用，我们可以通过几个例子来说明其使用方式和场景。

1、防止重排序

我们从一个最经典的例子来分析重排序问题。大家应该都很熟悉单例模式的实现，而在并发环境下的单例实现方式，我们通常可以采用双重检查加锁（DCL）的方式来实现。其源码如下：



```
1 package com.paddx.test.concurrent;
2
3 public class Singleton {
4     public static volatile Singleton singleton;
5
6     /**
7      * 构造函数私有，禁止外部实例化
8      */
9     private Singleton() {};
10
11     public static Singleton getInstance() {
12         if (singleton == null) {
13             synchronized (singleton) {
14                 if (singleton == null) {
15                     singleton = new Singleton();
16                 }
17             }
18         }
19         return singleton;
20     }
21 }
```



现在我们分析一下为什么要在变量singleton之间加上volatile关键字。要理解这个问题，先要了解对象的构造过程，实例化一个对象其实可以分为三个步骤：

- （1）分配内存空间。
- （2）初始化对象。
- （3）将内存空间的地址赋值给对应的引用。

但是由于操作系统可以对指令进行重排序，所以上面的过程也可能会变成如下过程：

- （1）分配内存空间。
- （2）将内存空间的地址赋值给对应的引用。
- （3）初始化对象

公告

昵称：liuxiaopeng
园龄：2年6个月
粉丝：249
关注：2
[+加关注](#)

搜索

找找看

谷歌搜索

最新随笔

- Spring Boot实战：拦截器与过滤器
- Spring Boot实战：静态资源处理
- Spring Boot实战：集成Swagger2
- Spring Boot实战：Restful API的构建
- Spring Boot实战：数据库操作
- Spring Boot实战：逐行释义HelloWorld
- Java集合类：AbstractCollection源码解析
- Java集合：整体结构
- Java 并发编程：volatile的使用及其原理
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)


我的标签

- Java(15)
- Spring(7)
- spring boot(7)
- 并发编程(4)


如果是这个流程，多线程环境下就可能将一个未初始化的对象引用暴露出来，从而导致不可预料的结果。因此，为了防止这个过程的重排序，我们需要将变量设置为volatile类型的变量。

2、实现可见性


可见性问题主要指一个线程修改了共享变量值，而另一个线程却看不到。引起可见性问题的主要原因是每个线程拥有自己的一个高速缓存区——线程工作内存。volatile关键字能有效的解决这个问题，我们看下下面的例子，就可以知道其作用：



```
1 package com.paddx.test.concurrent;
2
3 public class VolatileTest {
4     int a = 1;
5     int b = 2;
6
7     public void change(){
8         a = 3;
9         b = a;
10    }
11
12    public void print(){
13        System.out.println("b="+b+";a="+a);
14    }
15
16    public static void main(String[] args) {
17        while (true){
18            final VolatileTest test = new VolatileTest();
19            new Thread(new Runnable() {
20                @Override
21                public void run() {
22                    try {
23                        Thread.sleep(10);
24                    } catch (InterruptedException e) {
25                        e.printStackTrace();
26                    }
27                    test.change();
28                }
29            }).start();
30
31            new Thread(new Runnable() {
32                @Override
33                public void run() {
34                    try {
35                        Thread.sleep(10);
36                    } catch (InterruptedException e) {
37                        e.printStackTrace();
38                    }
39                    test.print();
40                }
41            }).start();
42
43        }
44    }
45 }
```



直观上说，这段代码的结果只可能有两种：b=3;a=3 或 b=2;a=1。不过运行上面的代码（可能时间上要长一点），你会发现除了上两种结果之外，还出现了第三种结果：




```
.....
b=2;a=1
b=2;a=1
b=3;a=3
b=3;a=3
b=3;a=1
b=3;a=3
b=2;a=1
b=3;a=3
b=3;a=3
.....
```

为什么会出现b=3;a=1这种结果呢？正常情况下，如果先执行change方法，再执行print方法，输出结果应该为b=3;a=3。相反，如果先执行的print方法，再执行change方法，结果应该是 b=2;a=1。那b=3;a=1的结果是怎么出来的？原因就是第一个线程将值a=3修改后，但是对第二个线程是不可见的，所以才出现这一结果。如果将a和b都改成volatile类型的变量再执行，则再也不会出现b=3;a=1的结果了。

3、保证原子性

关于原子性的问题，上面已经解释过。volatile只能保证对单次读/写的原子性。这个问题可以看下JLS中的描述：




```
17.7 Non-Atomic Treatment of double and long

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.
```



Rest(2)
Restful(1)
过滤器(1)
集合框架(1)
静态资源(1)
拦截器(1)
更多

随笔档案
2018年1月 (5)
2017年12月 (1)
2016年6月 (1)
2016年5月 (3)
2016年4月 (4)
2016年3月 (3)

积分与排名
积分 - 40379
排名 - 9716

最新评论
1. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁） 博主写得很棒 --还好可以改名字
2. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁） @就这个名引用在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。针对这句话有个疑问，如果我能保..... --还好可以改名字
3. Re:Java8内存模型—永久代(PermGen)和元空间(Metaspace) 感谢分享，写的很好！ --我不将就
4. Re:Spring Boot实战：集成Swagger 2 学习了 菜鸟飘过~~~ --祎祎家斌斌

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid possible complications.

这段话的内容跟我前面的描述内容大致类似。因为long和double两种数据类型的操作可分为高32位和低32位两部分，因此普通的long或double类型读/写可能不是原子的。因此，鼓励大家将共享的long和double变量设置为volatile类型，这样能保证任何情况下对long和double的单次读/写操作都具有原子性。

关于volatile变量对原子性保证，有一个问题容易被误解。现在我们就通过下列程序来演示一下这个问题：



```
1 package com.paddx.test.concurrent;
2
3 public class VolatileTest01 {
4     volatile int i;
5
6     public void addI(){
7         i++;
8     }
9
10    public static void main(String[] args) throws InterruptedException {
11        final VolatileTest01 test01 = new VolatileTest01();
12        for (int n = 0; n < 1000; n++) {
13            new Thread(new Runnable() {
14                @Override
15                public void run() {
16                    try {
17                        Thread.sleep(10);
18                    } catch (InterruptedException e) {
19                        e.printStackTrace();
20                    }
21                    test01.addI();
22                }
23            }).start();
24        }
25
26        Thread.sleep(10000);//等待10秒，保证上面程序执行完成
27
28        System.out.println(test01.i);
29    }
30 }
```



大家可能会误认为对变量加上关键字volatile后，这段程序就是线程安全的。大家可以尝试运行上面的程序。下面是我本地运行的结果：



可能每个人运行的结果不相同。不过应该能看出，volatile是无法保证原子性的（否则结果应该是1000）。原因也很简单，i++其实是一个复合操作，包括三步骤：

- （1）读取i的值。
- （2）对i加1。
- （3）将i的值写回内存。

volatile是无法保证这三个操作是具有原子性的，我们可以通过AtomicInteger或者Synchronized来保证+1操作的原子性。

注：上面几段代码中多处执行了Thread.sleep()方法，目的是为了增加并发问题的产生几率，无其他作用。

三、volatile的原理

通过上面的例子，我们基本应该知道了volatile是什么以及怎么使用。现在我们再来看看volatile的底层是怎么实现的。

1、可见性实现：

在前文中已经提及过，线程本身并不直接与主内存进行数据的交互，而是通过线程的工作内存来完成相应的操作。这也是导致线程间数据不可见的本质原因。因此要实现volatile变量的可见性，直接从这方面入手即可。对volatile变量的写操作与普通变量的主要区别有两点：

- （1）修改volatile变量时会强制将修改后的值刷新的主内存中。
- （2）修改volatile变量后会导致其他线程工作内存中对应的变量值失效。因此，再读取该变量值的时候就需要重新从读取主内存中的值。

通过这两个操作，就可以解决volatile变量的可见性问题。

2、有序性实现：

在解释这个问题前，我们先来了解一下Java中的happen-before规则，JSR 133中对Happen-before的定义如下：

Two actions can be ordered by a happens-before relationship.If one action happens before another, then the first is visible to and ordered before the second.

5. Re:Spring Boot实战：拦截器与过滤器
这两个使用场景有啥区别？
--四度空间的平面

阅读排行榜
1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(52862)
2. Java并发编程：Synchronized及其实现原理(44063)
3. Java 并发编程：volatile的使用及其原理(24488)
4. Java 并发编程：核心理论(22502)
5. Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）(21988)

评论排行榜
1. Java并发编程：Synchronized及其实现原理(21)
2. Java 并发编程：volatile的使用及其原理(16)
3. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(13)
4. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(12)
5. 通过反编译深入理解Java String及intern(10)

推荐排行榜
1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(36)
2. Java并发编程：Synchronized及其实现原理(33)
3. 从字节码层面看“HelloWorld”(26)
4. Java 并发编程：核心理论(21)
5. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(15)

通俗一点说就是如果a happen-before b，则a所做的任何操作对b是可见的。（这一点大家务必记住，因为happen-before这个词容易被误解为是时间的前后）。我们再来看看JSR 133中定义了哪些happen-before规则：

- Each action in a thread happens before every subsequent action in that thread.
 - An unlock on a monitor happens before every subsequent lock on that monitor.
 - A write to a volatile field happens before every subsequent read of that volatile.
 - A call to start() on a thread happens before any actions in the started thread.
 - All actions in a thread happen before any other thread successfully returns from a join() on that thread.
 - If an action a happens before an action b, and b happens before an action c, then a happens before c.

翻译过来为：

- 同一个线程中的，前面的操作 happen-before 后续的操作。（即单线程内按代码顺序执行。但是，在不影响在单线程环境执行结果的前提下，编译器和处理器可以进行重排序，这是合法的。换句话说，这一是规则无法保证编译重排和指令重排）。
- 监视器上的解锁操作 happen-before 其后续的加锁操作。（Synchronized 规则）
- 对volatile变量的写操作 happen-before 后续的读操作。（volatile 规则）
- 线程的start() 方法 happen-before 该线程所有的后续操作。（线程启动规则）
- 线程所有的操作 happen-before 其他线程在该线程上调用 join 返回成功后的操作。
- 如果 a happen-before b，b happen-before c，则a happen-before c（传递性）。

这里我们主要看下第三条：volatile变量的保证有序性的规则。《[Java并发编程：核心理论](#)》一文中提到过重排序分为编译器重排序和处理器重排序。为了实现volatile内存语义，JMM会对volatile变量限制这两种类型的重排序。下面是JMM针对volatile变量所规定的重排序规则表：

Can Reorder	2nd operation		
1st operation	Normal Load Normal Store	Volatile Load	Volatile Store
Normal Load Normal Store			No
Volatile Load	No	No	No
Volatile store		No	No

3、内存屏障

为了实现volatile可见性和happen-befor的语义。JVM底层是通过一个叫做“内存屏障”的东西来完成。内存屏障，也叫做内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。下面是完成上述规则所要求的内存屏障：

Required barriers	2nd operation			
1st operation	Normal Load	Normal Store	Volatile Load	Volatile Store
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store			StoreLoad	StoreStore

- （1）LoadLoad 屏障

执行顺序：Load1—>Loadload—>Load2

确保Load2及后续Load指令加载数据之前能访问到Load1加载的数据。
- （2）StoreStore 屏障

执行顺序：Store1—>StoreStore—>Store2

确保Store2以及后续Store指令执行前，Store1操作的数据对其它处理器可见。
- （3）LoadStore 屏障


执行顺序： Load1—>LoadStore—>Store2

确保Store2和后续Store指令执行前，可以访问到Load1加载的数据。
- （4）StoreLoad 屏障

执行顺序: Store1—> StoreLoad—>Load2

确保Load2和后续的Load指令读取之前，Store1的数据对其他处理器是可见的。

最后我可以通过一个实例来说明一下JVM中是如何插入内存屏障的：



```
1 package com.paddx.test.concurrent;
2
3 public class MemoryBarrier {
4     int a, b;
5     volatile int v, u;
6
7     void f() {
8         int i, j;
9
10        i = a;
11        j = b;
12        i = v;
13        //LoadLoad
14        j = u;
15        //LoadStore
16        a = i;
17        b = j;
18        //StoreStore
19        v = i;
20        //StoreStore
21        u = j;
```

```
22         //StoreLoad
23         i = u;
24         //LoadLoad
25         //LoadStore
26         j = b;
27         a = i;
28     }
29 }
```



四、总结

总体来说volatile的理解还是比较困难的，如果不是特别理解，也不用急，完全理解需要一个过程，在后续的文章中也还会多次看到volatile的使用场景。这里暂且对volatile的基础知识和原来有一个基本的了解。总体来说，volatile是并发编程中的一种优化，在某些场景下可以代替Synchronized。但是，volatile的不能完全取代Synchronized的位置，只有在一些特殊的场景下，才能适用volatile。总的来说，必须同时满足下面两个条件才能保证在并发环境的线程安全：

- (1) 对变量的写操作不依赖于当前值。
- (2) 该变量没有包含在具有其他变量的不变式中。

作者：liuxiaopeng

博客地址：<http://www.cnblogs.com/paddix/>

声明：转载请在文章页面明显位置给出原文连接。

标签: Java, 并发编程

好文要顶

关注我

收藏该文



liuxiaopeng

关注 - 2

粉丝 - 249

+加关注

« 上一篇：Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

» 下一篇：Java集合：整体结构

posted @ 2016-05-17 08:01 liuxiaopeng 阅读(24488) 评论(16) 编辑 收藏

评论列表

#1楼 2016-05-17 09:09 Haart

提醒博主：按照JLS，除了long 和 double之外的类型，读写操作本身就是原子的，不需要通过volatile修饰。

再进一步说，如果需要确保原子操作，使用AtomicLong之类的同步类更好，这个同步类内置了很多原子操作，包括自增和自减。

再进一步说，唯一需要用到volatile的场合是为了避免活性失败，但是又不想用锁和同步类。

支持(3) 反对(0)

#2楼 2016-05-17 09:50 sujiehao

好文章就爱顶，我顶=。=

支持(0) 反对(0)

#3楼 2016-05-17 10:15 辛巴国王

博主厉害，深度好文，学习了！

支持(0) 反对(0)

#4楼[楼主] 2016-05-17 20:58 liuxiaopeng

@ sujiehao

谢谢~

支持(0) 反对(0)

#5楼[楼主] 2016-05-17 20:58 liuxiaopeng

@ 辛巴国王

共同学习~

支持(0) 反对(0)

#6楼[楼主] 2016-05-17 21:14 liuxiaopeng

@ Haart

AtomicLong确实可以保证原子性，也提供了很多比较方便的方法。不过如果只是单纯的对long/double读/写的话，是可以用volatile的。

至于volatile使用的唯一场景是“避免活性失败”，在jdk 1.5之前应该是，但是jdk 1.5之后，volatile的语义是有加强的，所以应用场景也有所扩大。

支持(2) 反对(0)

#7楼 2017-02-13 14:47 sharp666

volatile非线程安全，类似的容易犯错的是把线程安全的ConcurrentHashMap按先get判断是否存在再put的用法
楼主有没有微信啥的？交流交流

支持(0) 反对(0)

#8楼 2017-03-23 15:22 坐在家里晒太阳

您好，楼主，我有几个问题想请教一下，在实现可见性那个例子中，出现b=3 a=1的原因是：就是第一个线程将值a=3修改后，但是对第二个线程是不可见的，所以才出现这一结果，既然b=3已经对第二个线程是可见的状态了，为什么比b=a之前的操作a=3会成为不可见的呢？还有一点就是既然a=3修改后对线程二是不可见的，我是不是还可以考虑到第四种情况，就是b=a修改之后，对第二线程是不可见的，然后输出的结果是b=2,a=3呢，请大神解答一下。

支持(0) 反对(0)

#9楼 2017-03-23 15:35 坐在家里晒太阳

```
b=2;a=1
b=2;a=1
b=2;a=1
b=2;a=3
b=3;a=3
b=3;a=3
b=3;a=3
. . .
```

好吧，是有这种可能。。。

支持(0) 反对(0)

#10楼 2017-03-24 17:25 awkejiang

double-checked locking的单例写的有问题吧，应该是

```
1 class Singleton {
2     private static volatile Singleton singleton;
3
4     private Singleton() {}
5
6     public static Singleton getInstance() {
7         if (singleton == null) {
8             synchronized (Singleton.class) {
9                 if (singleton == null) {
10                     singleton = new Singleton();
11                 }
12             }
13         }
14         return singleton;
15     }
16 }
```

支持(0) 反对(0)

#11楼 2017-04-02 12:13 Stef_ToBeC

单例那里，既然说synchronized区域能确保操作的有序性，为什么还要用volatile？

支持(2) 反对(0)

#12楼 2017-04-24 11:05 mdong

感谢分享，这是我见过的总结的最好的文章，赞。

支持(0) 反对(0)

#13楼 2017-07-15 10:23 JohnNaruto

volatile实现可见性:变量a和b加上volatile，自己写程序运行后，还是会出现b=3;a=1的结果了，何解？

```
1 public class VolatileDemo01 {
2     volatile int a = 1;
3     volatile int b = 2;
4
5     public void change() {
6         a = 3;
7         b = a;
8     }
9
10    public void print() {
11        System.out.println("a=" + a + "; b=" + b);
12    }
13 }
14
15
16 public static void test02() {
17     while (true) {
18         final VolatileDemo01 test = new VolatileDemo01();
19
```

```
20         new Thread(new Runnable() {
21             @Override
22             public void run() {
23                 try {
24                     Thread.sleep(10);
25                 } catch (InterruptedException e) {
26                     e.printStackTrace();
27                 }
28                 test.change();
29             }
30         }).start();
31
32         new Thread(new Runnable() {
33             @Override
34             public void run() {
35                 try {
36                     Thread.sleep(10);
37                 } catch (InterruptedException e) {
38                     e.printStackTrace();
39                 }
40                 test.print();
41             }
42         }).start();
43     }
44 }
```

支持(1) 反对(0)

#14楼 2017-07-15 10:26 JohnNaruto

@ awkejiang
为啥将对象锁更改成类锁，何解，说出你的理由

支持(0) 反对(0)

#15楼 2017-07-18 17:53 yhanliang

@ JohnNaruto
是的,我试验了一把 也是这样的,依然会出现 b=3;a=1

支持(0) 反对(0)

#16楼 2017-09-05 14:12 sinama

@JohnNaruto 我也是加了volatile还是会出现b=3；a=1的情况， 怎么解决；
@liuxiaopeng 2、实现可见性z那段代码跑过了吗，我这还是有问题呢，你qq多少， 可以加你吗

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【活动】2050 大会 - 年青人因科技而团聚（ 5.26-27杭州·云栖小镇）

【活动】华为云全新一代云服务器·限时特惠5.6折

【推荐】腾讯云多款高规格服务器，免费申请试用6个月