

Java并发编程之volatile关键字解析

赞 | 0

收藏 | 3

java 并发 jvm volatile **ziwenxie** 4月30日发布

804 次浏览

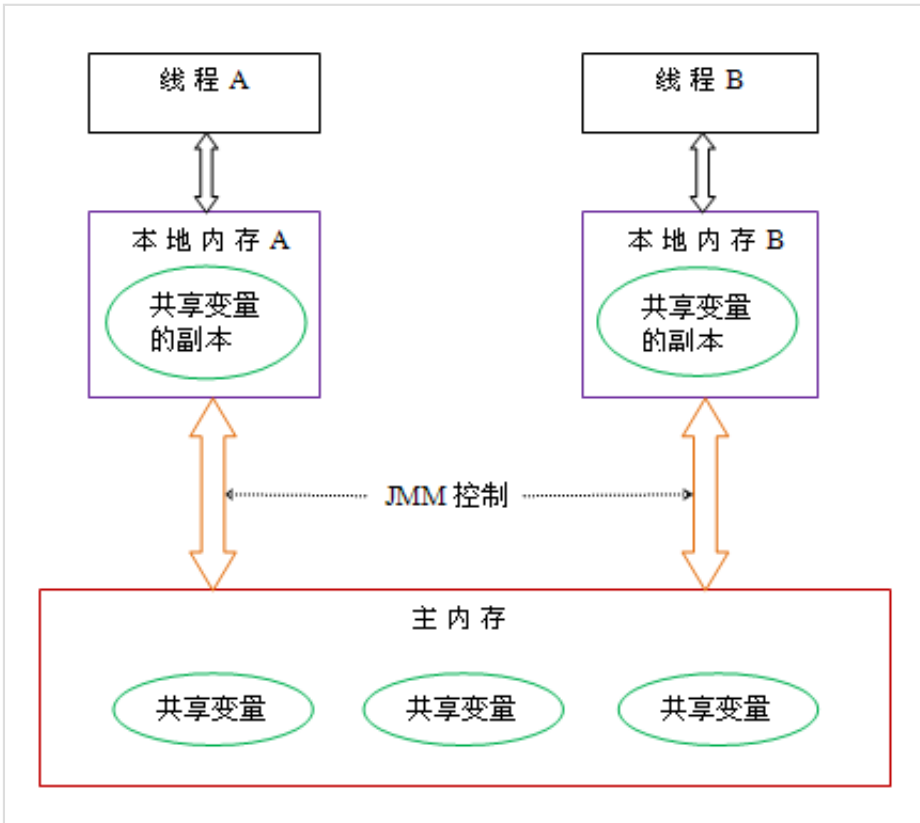
本文为作者原创，转载请声明[博客出处](#):)

引言

`volatile` 关键字虽然从字面上理解起来比较简单，但是要用好不是一件容易的事情。本文我们就从JVM内存模型开始，了解一下 `volatile` 的应用场景。

JVM内存模型

在了解 `volatile` 之前，我们有必要对JVM的内存模型有一个基本的了解。Java的内存模型规定了所有的变量都存储在主内存中（即物理硬件的内存），每条线程还具有自己的工作内存（工作内存可能位于处理器的高速缓存之中），线程的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取，赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量）。不同的线程之间无法直接访问对方工作内存之间的变量，线程间变量值的传递需要通过主内存来完成。



p.s: 对于上面提到的副本拷贝，比如假设线程中访问一个10MB的对象，并不会把这10MB的内存复制一份拷贝出来，实际上这个对象的引用，对象中某个在线程访问到的字段是有可能存在拷贝的，但不会有虚拟机实现把整个对象拷贝一次。

在并发编程中，我们通常会遇到以下三个问题：原子性，可见性，有序性，下面我们我们来具体看一下这三个特性与 `volatile` 之间的联系：

有序性



```

public class TestCase {
    public static int number;
    public static boolean isinitied;

    public static void main(String[] args) {
        new Thread(
            () -> {
                while (!isinitied) {
                    Thread.yield();
                }
                System.out.println(number);
            }
        ).start();
        number = 20;
        isinitied = true;
    }
}

```

对于上面的代码我们上面的本意是想输出 20，但是如果运行的话可以发现输出的值可能会是 0。这是因为有时候为了提供程序的效率，JVM会做进行及时编译，也就是可能会对指令进行重排序，将 `isInitied = true;` 放在 `number = 20;` 之前执行，在单线程下面这样做没有任何问题，但是在多线程下则会出现重排序问题。如果我们将 `number` 声名为 `volatile` 就可以很好的解决这个问题，这可以禁止JVM进行指令重排序，也就意味着 `number = 20;` 一定会在 `isInitied = true` 前面执行。

可见性

比如对于变量 `a`，当线程一要修改变量`a`的值，首先需要将`a`的值从主存复制过来，再将`a`的值加一，再将`a`的值复制回主存。在单线程下面，这样的操作没有任何的问题，但是在多线程下面，比如还有一个线程二，在线程一修改`a`的值的时候，也从主存将`a`的值复制过来进行加一，随后线程一和线程二先后将`a`的值复制回主存，但是主存中`a`的值最终将只会加一而不是加二。

使用 `volatile` 可以解决这个问题，它可以保证在线程一修改`a`的值之后立即将修改值同步到主存中，这样线程二拿到的`a`的值就是线程一已经修改过的`a`的值了。对`volatile`变量执行写操作时，会在写操作后加入一条 `store` 屏障指令，对`volatile`变量执行读操作时，会在写操作后加入一条 `load` 屏障指令。

线程写`volatile`变量过程：

1. 改变线程工作内存中`volatile`变量副本的值；
2. 将改变后的副本的值从工作内存刷新到主内存。

线程读`volatile`变量过程：

1. 从主内存中读取`volatile`变量的最新值到工作内存中；
2. 从工作内存中读取`volatile`变量副本。

原子性

原子性是指CPU在执行一条语句的时候，不会中途转去执行另外的语句。比如 `i = 1` 就是一个原子操作，但是 `++i` 就不是一个原子操作了，因为它要求首先读取 `i` 的值，然后修改 `i` 的值，最后将值写入主存中。

但是 `volatile` 却不能保证程序的原子性，下面我们通过一个实例来验证一下：

```

public class TestCase {
    public volatile int v = 0;
    public static final int threadCount = 20;

    public void increase() {
        v++;
    }

    public static void main(String[] args) {
        TestCase testCase = new TestCase();
        for (int i=0; i<threadCount; i++) {
            new Thread(
                () -> {
                    for (int j=0; j<1000; j++) {
                        testCase.increase();
                    }
                }
            ).start();
        }

        while (Thread.activeCount() > 1) {
            Thread.yield();
        }
        System.out.println(testCase.v);
    }
}

```

输出结果：

18921

上面我们的本意是想让输出 20000，但是运行程序后，结果可能会小于 20000。因为 `v++` 它本身并不是一个原子操作，它是分为多个步骤的，而且 `volatile` 本身也并不能保证原子性。

上面的程序使用 `synchronized` 则可以很好的解决，只需要声明 `public synchronized void increase()` 就行了。

或者使用lock也行：

```

Lock lock = new ReentrantLock();

public void increase() {
    lock.lock();
    try {
        v++;
    } finally {
        lock.unlock();
    }
}

```

或者将 `v` 声明为 `AtomicInteger v = new AtomicInteger();`。在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的自增，自减，以及加法操作，减法操作进行了封装，保证这些操作是原子性操作。

单例模式

下面我们通过单例模式来看一下 `volatile` 的一个具体应用：

```
class Singleton {
    private volatile static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null)
                    instance = new Singleton();
            }
        }
        return instance;
    }

    public static void main(String[] args) {
        Singleton.getInstance();
    }
}
```

上面 `instance` 必须要用 `volatile` 修饰，因为 `new Singleton` 是分为三个步骤的：

- 1. 给instance指向的对象分配内存，并设置初始值为null（根据JVM类加载机制的原理，对于静态变量这一步应该在 `new Singleton` 之前就已经完成了）。
- 2. 执行构造函数真正初始化instance
- 3. 将instance指向对象分配内存空间（分配内存空间之后instance就是非null了）

在我们的步骤2, 3之间的顺序是可以颠倒的，如果线程一在执行步骤3之后并没有执行步骤2，但是被线程二抢占了，线程二得到的 `instance` 是非null，但是instance却还没有初始化。而使用volatile则可以保证程序的有序性。

References

UNDERSTANDING THE JVM
JAVA CONCURRENCY IN PRACTICE

4月30日发布 ...



赞赏支持

赞 | 0

收藏 | 3

如果觉得我的文章对你有用，请随意赞赏

你可能感兴趣的文章

volatile的使用及DCL模式 3 收藏，1k 浏览

Java 多线程（6）：volatile 关键字的使用 5 收藏，334 浏览

Java并发，volatile+不可变容器对象能保证线程安全么？！ 242 浏览

评论

默认排序 时间排序



文明社会，理性评论

发表评论



ziwenxie
307 声望

关注作者

发布于专栏

ziwenxie

12 人关注

关注专栏

网站相关

关于我们
服务条款
帮助中心
声望与权限
编辑器语法
每周精选
社区服务中心




联系合作

联系我们
加入我们
合作伙伴
媒体报道
建议反馈

常用链接

笔记插件:
Chrome
笔记插件:
Firefox
订阅：问答 / 文章
文档镜像
D-DAY 技术沙龙
黑客马拉松
Hackathon
域名搜索注册
周边店铺

社区日志

产品技术日志
社区运营日志
市场运营日志
团队日志
社区访谈



内容许可

除特别说明外，用户内容均采用 知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议 进行许可
本站由 又拍云 提供 CDN 存储服务



手机扫一扫
下载官方 App