链滴

JMM、Volatile（Juc-10）

作者：yscxy

原文链接：https://ld246.com/article/1606983682184

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

概述

谈谈你对volatile的理解

1. 保证可见性
2. 不保证原子性
3. 禁止指令重排

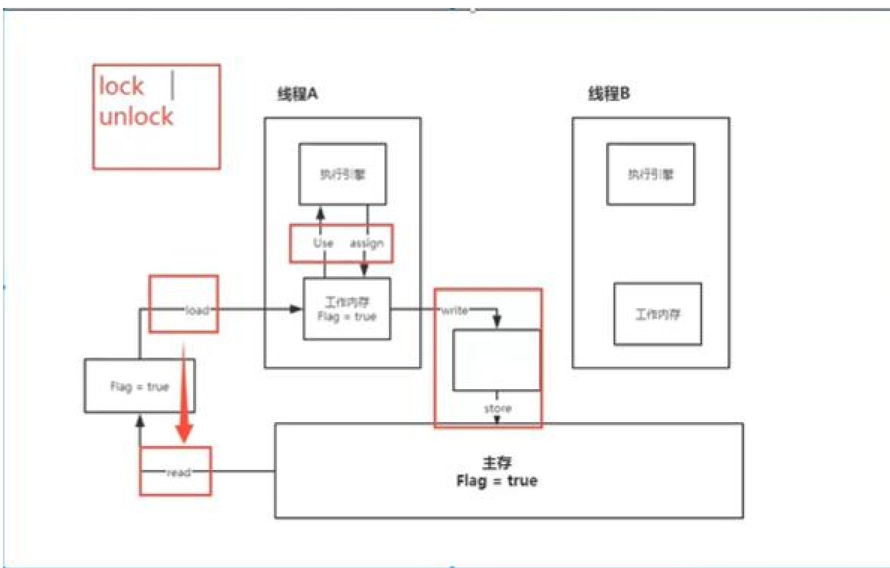
什么是JMM

java内存模型，是一个不存在的东西，是一个概念，一个约定！

关于JMM一些同步的约定：

1. 线程解锁前一定把共享变量立刻刷回主储存
2. 线程加锁钱必须把主存最新值copy到工作内存中
3. 加锁一定是同一把锁

线程，工作内存，主内存一共有4组8种操作



内存交互操作

内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

-
    - lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
    - unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
    - read （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用

- load　　（载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中

- use　　（使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚机遇到一个需要使用到变量的值，就会使用到这个指令

- assign　（赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存变量副本中

- store　（存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存，以便后续的write使用

- write　（写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入内存的变量中

JMM对这八种指令的使用，制定了如下规则：

-

- 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必write

- 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存

- 不允许一个线程将没有assign的数据从工作内存同步回主内存

- 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼量实施use、store操作之前，必须经过assign和load操作

- 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才解锁

- 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前必须重新load或assign操作初始化变量的值

- 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的量

- 对一个变量进行unlock操作之前，必须把此变量同步回主内存

JMM对这八种操作规则和对volatile的一些特殊规则就能确定哪里操作是线程安全，哪些操作是程不安全的了。但是这些规则实在复杂，很难在实践中直接分析。所以一般我们也不会通过上述规则行分析。更多的时候，使用java的happen-before规则来进行分析。

问题

程序不知道主内存的值已经被修改过了,类似这样的代码就会出现死循环

```
package net.yscxy.tvolatile;

import java.util.concurrent.TimeUnit;

/**
 * @Author WangFuKun
 * @create 2020/12/2 20:21
 */
public class JmmTest {
    static  int number = 0;

    public static void main(String[] args) throws InterruptedException {//main线程
```

```
        new Thread(() -> { //线程1对主内存的变化是不可见的，不知道的
            while (number == 0) {
            }
        }).start();
        TimeUnit.SECONDS.sleep(1);
        number = 1;
        System.out.println(number);
    }

}
```

volatile

1. 保证可见性

```
package net.yscxy.tvolatile;

import java.util.concurrent.TimeUnit;

/**
 * @Author WangFuKun
 * @create 2020/12/2 20:21
 */
public class JmmTest {
    static volatile int number = 0;

    public static void main(String[] args) throws InterruptedException {//main线程
        new Thread(() -> { //线程1对主内存的变化是不可见的，不知道的
            while (number == 0) {
            }
        }).start();
        TimeUnit.SECONDS.sleep(1);
        number = 1;
        System.out.println(number);
    }

}
```

2. 不保证原子性

原子性：不可分割

也就是线程A在执行任务的时候是不能被打扰和被分割的，要么同时成功，要么同时失败

```
package net.yscxy.tvolatile;

/**
 * @Author WangFuKun
 * @create 2020/12/2 21:39
 */
//测试不保证原子性
public class VDemo02 {
    private volatile static int number = 0;
```

```
    public  static void add() {
        number++;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    add();
                }
            }).start();
        }
        //因为默认有jc线程和主线程这两个线程一直活跃着
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        System.out.println(number);
    }
}
```

如果不加lock或者synchronized如何保证原子性

方案：使用原子类，解决原子性问题AtomicInteger

AtomicInteger 他的底层用的是CAS

```
package net.yscxy.tvolatile;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * @Author WangFuKun
 * @create 2020/12/2 21:39
 */
//测试不保证原子性
public class VDemo02 {
    //使用原子类进行解决
    private volatile static AtomicInteger number = new AtomicInteger();

    public synchronized static void add() {
        //number++
        //当然这里并不简单的是一个加一操作
        number.getAndIncrement(); //AtomicInteger +1方法，CAS
    }
```

```java
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    add();
                }
            }).start();
        }
        //因为默认有jc线程和主线程这两个线程一直活跃着
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        System.out.println(number);
    }
}
```

这些类的低层直接和操作系统挂钩，在内存中修改值，Unsafe是一个很特殊的存在