作者：代兴　　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

Java 语言的 volatile 关键字说明

Java 语言的 volatile 关键字，属于编译期屏障、运行期屏障。
使用 volatile 修饰的变量，禁止编译期重排序，禁止运行期重排序，插入内存屏障指令，读操作从内存读取，写操作写到内存。

查看 JVM 源码中的 volatile

文件 openjdk/hotspot/src/cpu/x86/vm/assembler_x86.hpp。

```
  // Serializes memory and blows flags
  void membar(Membar_mask_bits order_constraint) {
    if (os::is_MP()) {
      // We only have to handle StoreLoad
      if (order_constraint & StoreLoad) {
        // All usable chips support "locked" instructions which suffice
        // as barriers, and are much faster than the alternative of
        // using cpuid instruction. We use here a locked add [esp],0.
        // This is conveniently otherwise a no-op except for blowing
        // flags.
        // Any change to this code may need to revisit other places in
        // the code where this idiom is used, in particular the
        // orderAccess code.
        lock();
        addl(Address(rsp, 0), 0);// Assert the lock# signal here
      }
    }
  }
```

文件 openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_64.cpp。

```
// Volatile variables demand their effects be made known to all CPU's
// in order.  Store buffers on most chips allow reads & writes to
// reorder; the JMM's ReadAfterWrite.java test fails in -Xint mode
// without some kind of memory barrier (i.e., it's not sufficient that
// the interpreter does not reorder volatile references, the hardware
// also must not reorder them).
//
// According to the new Java Memory Model (JMM):
// (1) All volatiles are serialized wrt to each other.  ALSO reads &
//     writes act as aquire & release, so:
// (2) A read cannot let unrelated NON-volatile memory refs that
//     happen after the read float up to before the read.  It's OK for
```

```
//     non-volatile memory refs that happen before the volatile read to
//     float down below it.
// (3) Similar a volatile write cannot let unrelated NON-volatile
//     memory refs that happen BEFORE the write float down to after the
//     write.  It's OK for non-volatile memory refs that happen after the
//     volatile write to float up before it.
//
// We only put in barriers around volatile refs (they are expensive),
// not _between_ memory refs (that would require us to track the
// flavor of the previous memory refs).  Requirements (2) and (3)
// require some barriers before volatile stores and after volatile
// loads.  These nearly cover requirement (1) but miss the
// volatile-store-volatile-load case.  This final case is placed after
// volatile-stores although it could just as well go before
// volatile-loads.
void TemplateTable::volatile_barrier(Assembler::Membar_mask_bits
                                     order_constraint) {
  // Helper function to insert a is-volatile test and memory barrier
  if (os::is_MP()) { // Not needed on single CPU
    __ membar(order_constraint);
  }
}
```

代码逻辑：
判断条件 `if (os::is_MP())`，多核 CPU 才需要内存屏障。
判断条件 `if (order_constraint & StoreLoad)`，x86 CPU 属于强内存模型，StoreLoad 才需要内存屏障。
内存屏障使用 lock 指令，更轻量。

查看 JDK 源码中的 volatile

Java 类 AtomicLong。

```
public class AtomicLong extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 1927816293512124184L;

    // setup to use Unsafe.compareAndSwapLong for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicLong.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile long value;
```

Java 类 ConcurrentSkipListMap。

```
public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable {
    /**
     * The topmost head index of the skiplist.
     */
    private transient volatile HeadIndex<K,V> head;

    static final class Node<K,V> {
        final K key;
        volatile Object value;
        volatile Node<K,V> next;

    static class Index<K,V> {
        final Node<K,V> node;
        final Index<K,V> down;
        volatile Index<K,V> right;
```

volatile 修饰的属性，保证可见性。多个线程并发，一个线程修改属性的值，其他线程可以看到属性的最新值。注意：volatile 不能保证原子性，多个线程并发写值，可能导致覆盖写。

用 Java 分析 volatile 的影响

编写代码： NumVolatile.java

```
package org.test3.volatilex;

public class NumVolatile {
    // 数字。有 volatile 的场景。
    private static volatile long numWithVolatile = 0;

    // 线程的任务。有 volatile 的场景
    private static class NumTaskWithVolatile implements Runnable {
        @Override
        public void run() {
            for (int m = 0; m < 9000000; m++) {
                numWithVolatile = numWithVolatile + 1;
            }
        }
    }

    // 数字。没有 volatile 的场景。
    private static long numWithoutVolatile = 0;

    // 线程的任务。没有 volatile 的场景
    private static class NumTaskWithoutVolatile implements Runnable {
```

```java
        @Override
        public void run() {
            for (int m = 0; m < 9000000; m++) {
                numWithoutVolatile = numWithoutVolatile + 1;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        // 多个线程。有 volatile 的场景
        Thread thread1 = new Thread(new NumTaskWithVolatile());
        Thread thread2 = new Thread(new NumTaskWithVolatile());

        // 等待线程执行完成。查看耗时
        long beginMillis = System.currentTimeMillis();
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        long costMillis = System.currentTimeMillis() - beginMillis;
        String tmp = "有 volatile 的场景  耗时=" + costMillis + "  num=" + numWithVolatile;
        System.out.println(tmp);

        //----------------
        // 多个线程。没有 volatile 的场景
        Thread thread3 = new Thread(new NumTaskWithoutVolatile());
        Thread thread4 = new Thread(new NumTaskWithoutVolatile());

        // 等待线程执行完成。查看耗时
        long beginMillisB = System.currentTimeMillis();
        thread3.start();
        thread4.start();
        thread3.join();
        thread4.join();
        long costMillisB = System.currentTimeMillis() - beginMillisB;
        String tmpB = "没有 volatile 的场景  耗时=" + costMillisB + "  num=" + numWithoutVolatile;
        System.out.println(tmpB);
    }
}
```

运行代码：

```
有 volatile 的场景  耗时=225  num=12303621
没有 volatile 的场景  耗时=15  num=9573743
```

分析结果：
代码逻辑为 2 个线程，并发修改一个整数变量，区分有 volatile 的场景、没有 volatile 的场景。
有 volatile 的场景，耗时为 225 毫秒，数字为 12303621。没有 volatile 的场景，耗时为 15 毫秒，数字为 9573743。两者耗时相差十多倍。
代码 `numWithVolatile = numWithVolatile + 1` ，先读出 numWithVolatile 的值，然后更新 numWithVolatile 的

值。
没有 volatile 的场景，线程读变量、写变量使用高速缓存，高速缓存和内存使用异步刷新，所以速度快。
有 volatile 的场景，线程读变量从内存加载变量的最新值，线程写变量把修改的值写到内存，再加上缓存同步、内存屏障指令，所以速度慢。
如果使用 CAS 指令，累加的数字为 18000000。实际的数字为 12303621、9573743，因为发生了写值覆盖。volatile 使得数据与内存同步更多，减少了写值覆盖，所以数字更接近 18000000。