volatile 实现原理

volatile 关键字保证了内存的可见性，但是不保证原子性。
volatile 为访问变量加入了内存屏障，防止指令重排序。
被 volatile 修饰的变量在编译之后的指令中，定义变量的 flags 会加上 ACC_VOLATILE 标志。
ACC_VOLATILE 标志被定义在 JVM 源码的 accessFlags.hpp 中。

```
// accessFlags.hpp
bool is_volatile () const { return (_flags & JVM_ACC_VOLATILE) != 0; }

// bytecodeInterpreter.cpp
// 存储变量时，判断是否被 volatile 修饰
if (cache -> is_volatile()) {
    // 判断数据类型，根据不同的数据类型执行不同的方法
    if (tos_type == itos) {
        // int 类型
        obj -> release_int_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == atos) { // obj 对象类型
        VERIFY_OOP(STACK_OBJECT(-1));
        obj -> release_obj_field_put(field_offset, STACK_OBJECT(-1));
        OrderAccess::release_store(&BYTE_MAP_BASE[(uintptr_t)obj >> CardTableModRefBS::card_shift],
    } else if (tos_type == btos) {
        // byte 类型
        obj -> release_byte_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ltos) {
        // long 类型
        obj -> release_long_field_put(field_offset, STACK_LONG(-1));
    } // ... char, short, float, double 省略
    // 执行完毕后，执行下面这个方法
    OrderAccess::storeload();
}

// oop.inline.cpp
// release_int_field_put 方法在此文件中
inline void oopDesc::release_int_field_put(int offset, jint contents) { OrderAccess::release_store(

// release_store 方法在 orderAccess.hpp 中定义
// orderAccess.hpp
static void release_store(volatile jint* p, jint v); //还有对其他数据类型的定义

// 具体的实现根据不同的操作系统CPU进行实现 Linux, Windows.. 等等 CPU
// 例如: orderAccess_linux_x86.inline.hpp
inline void OrderAccess::release_store(volatile jint* p, jint v) { *p = v; } // 此处 volatile, 语言级
```

1. 对每个 volatile 变量的写操作的前面会插入 storestore barrier。
2. 对每个 volatile 变量的写操作后会插入 storeload barrier。
3. 对每个 volatile 读操作之前插入 loadload barrier。
4. 对每个 volatile 读操作之后插入 loadstore barrier。

以上代码第 23 行证实了第二点：(其他的可以在其他源码中找到)

```
OrderAccess::storeload(); // 这是在写操作完毕之后，执行的方法。

// 该方法在 orderAccess_linux_x86.inline.hpp 中（此处只看这一个实现）
inline void OrderAccess::loadload() { acquire(); }
```

```
inline void OrderAccess::storestore() { release(); }
inline void OrderAccess::loadstore() { acquire(); }
inline void OrderAccess::storeload() { fence(); }

// fence() 方法中会在指令之前加入 lock 前缀。（汇编指令）。。。汇编指令没有深入研究，不贴代码了。
```

所以 volatile 修饰符可以保证内存的可见性（内存屏障）。但是无法保证高并发下的原子性。
例如：

```
package com.keanu.io.study.concurrency;

public class VolatileDemo {

    volatile int i = 0;

    public void incr() {
        i++;
    }

     public static void main(String[] args) {
         new VolatileDemo().incr();
     }
}
```

被 JVM 编译之后（使用 javap -c 指令查看字节码）：

```
public class com.keanu.io.study.concurrency.VolatileDemo {
  volatile int i;

  public com.keanu.io.study.concurrency.VolatileDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iconst_0
       6: putfield      #2                  // Field i:I
       9: return

  public void incr();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2                  // Field i:I
       5: iconst_1
       6: iadd
       7: putfield      #2                  // Field i:I
      10: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #3                  // class com/keanu/io/study/concurrency/VolatileDemo
       3: dup
       4: invokespecial #4                  // Method "<init>":()V
       7: invokevirtual #5                  // Method incr:()V
      10: return
}
```

可以看到，编译之后的指令中，i++（复合操作） 的操作分成了 3步，以上代码的 17，19，20 行。

1. getfield
2. iadd
3. putfield

当有多个线程同时执行时，有可能同一时间有多个线程同时执行了 getfield 指令，可能就会有一个线程拿到的是旧值，这就造成了原子性问题。

可以通过 synchronized 关键字来解决，避免线程并行执行。synchronized 实现原理可以参照这篇文章：https://keanu96.github.io/deep-in-synchronized/