

# 面试必问的volatile，你了解多少？



占小狼  [关注](#)

2017.12.27 00:51 字数 3395 阅读 2820 评论 20 喜欢 84

占小狼 转载请注明原创出处，谢谢！

## 前言

Java中volatile这个热门的关键字，在面试中经常会被提及，在各种技术交流群中也经常被讨论，但似乎讨论不出一个完美的结果，带着种种疑惑，准备从JVM、C++、汇编的角度重新梳理一遍。

volatile的两大特性：禁止重排序、内存可见性，这两个概念，不太清楚的同学可以看这篇文章 -> [java volatile关键字解惑](#)

概念是知道了，但还是很迷糊，**它们到底是如何实现的？**

本文会涉及到一些汇编方面的内容，如果多看几遍，应该能看懂。

## 重排序

为了理解重排序，先看一段简单的代码

```
public class VolatileTest {

    int a = 0;
    int b = 0;

    public void set() {
        a = 1;
        b = 1;
    }

    public void loop() {
        while (b == 0) continue;
        if (a == 1) {
            System.out.println("i'm here");
        } else {
            System.out.println("what's wrong");
        }
    }
}
```

VolatileTest类有两个方法，分别是set()和loop()，假设线程B执行loop方法，线程A执行set方法，会得到什么结果？

答案是不确定，因为这里涉及到了编译器的重排序和CPU指令的重排序。

## 编译器重排序

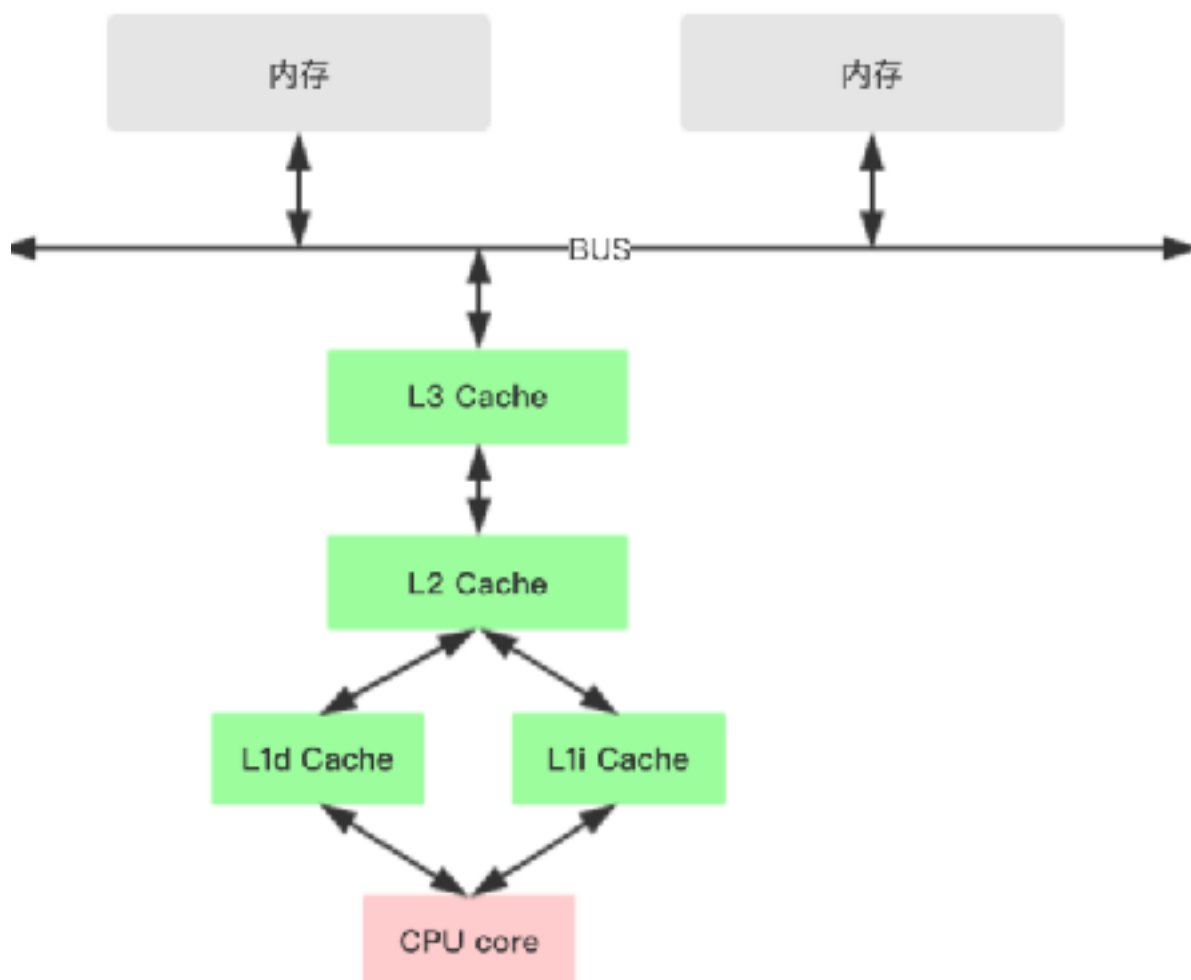
编译器在不改变单线程语义的前提下，为了提高程序的运行速度，可以对字节码指令进行重新排序，所以代码中a、b的赋值顺序，被编译之后可能就变成了先设置b，再设置a。

因为对于线程A来说，先设置哪个，都不影响自身的结果。

## CPU指令重排序

CPU指令重排序又是怎么回事？

在深入理解之前，先看看x86的cpu缓存结构。



- 1、各种寄存器，用来存储本地变量和函数参数，访问一次需要1cycle，耗时小于1ns；
- 2、L1 Cache，一级缓存，本地core的缓存，分成32K的数据缓存L1d和32k指令缓存L1i，访问L1需要3cycles，耗

时大约1ns；

3、L2 Cache，二级缓存，本地core的缓存，被设计为L1缓存与共享的L3缓存之间的缓冲，大小为256K，访问L2需要12cycles，耗时大约3ns；

4、L3 Cache，三级缓存，在同插槽的所有core共享L3缓存，分为多个2M的段，访问L3需要38cycles，耗时大约12ns；

当然了，还有平时熟知的DRAM，访问内存一般需要65ns，所以CPU访问一次内存和缓存比较起来显得很慢。

对于不同插槽的CPU，L1和L2的数据并不共享，一般通过MESI协议保证Cache的一致性，但需要付出代价。

在MESI协议中，每个Cache line有4种状态，分别是：

#### 1、M(Modified)

这行数据有效，但是被修改了，和内存中的数据不一致，数据只存在于本Cache中

#### 2、E(Exclusive)

这行数据有效，和内存中的数据一致，数据只存在于本Cache中

#### 3、S(Shared)

这行数据有效，和内存中的数据一致，数据分布在很多Cache中

#### 4、I(Invalid)

这行数据无效

每个Core的Cache控制器不仅知道自己的读写操作，也监听其它Cache的读写操作，假如有4个Core：

1、Core1从内存中加载了变量X，值为10，这时Core1中缓存变量X的cache line的状态是E；

2、Core2也从内存中加载了变量X，这时Core1和Core2缓存变量X的cache line状态转化成S；

3、Core3也从内存中加载了变量X，然后把X设置成了20，这时Core3中缓存变量X的cache line状态转化成M，其它Core对应的cache line变成I（无效）

当然了，不同的处理器内部细节也是不一样的，比如Intel的core i7处理器使用从MESI中演化出的MESIF协议，F(Forward)从Share中演化而来，一个cache line如果是F状态，可以把数据直接传给其它内核，这里就不纠结了。

CPU在cache line状态的转化期间是阻塞的，经过长时间的优化，在寄存器和L1缓存之间添加了LoadBuffer、StoreBuffer来降低阻塞时间，LoadBuffer、StoreBuffer，合称排序缓冲(Memoryordering Buffers (MOB))，Load缓冲64长度，store缓冲36长度，Buffer与L1进行数据传输时，CPU无须等待。

1、CPU执行load读数据时，把读请求放到LoadBuffer，这样就不用等待其它CPU响应，先进行下面操作，稍后再处理这个读请求的结果。

2、CPU执行store写数据时，把数据写到StoreBuffer中，待到某个适合的时间点，把StoreBuffer的数据刷到主存中。

因为StoreBuffer的存在，CPU在写数据时，真实数据并不会立即表现到内存中，所以对于其它CPU是不可见的；同样的道理，LoadBuffer中的请求也无法拿到其它CPU设置的最新数据；

由于StoreBuffer和LoadBuffer是异步执行的，所以在外面看来，先写后读，还是先读后写，没有严格的固定顺序。

## 内存可见性如何实现

从上面的分析可以看出，其实是CPU执行load、store数据时的异步性，造成了不同CPU之间的内存不可见，那么如何做到CPU在load的时候可以拿到最新数据呢？

## 设置volatile变量

写一段简单的java代码，声明一个volatile变量，并赋值

```
public class VolatileTest {  
  
    static volatile int i;  
  
    public static void main(String[] args){  
        i = 10;  
    }  
}
```

这段代码本身没什么意义，只是想看看加了volatile之后，编译出来的字节码有什么不同，执行 `javap -verbose VolatileTest` 之后，结果如下：

```
static volatile int i;  
flags: ACC_STATIC, ACC_VOLATILE
```

让人很失望，没有找类似关键字synchronize编译之后的字节码指令（monitorenter、monitorexit），volatile编译之后的赋值指令putstatic没有什么不同，唯一不同是变量i的修饰flags多了一个 ACC\_VOLATILE 标识。

不过，我觉得可以从这个标识入手，先全局搜下 ACC\_VOLATILE，无从下手的时候，先看看关键字在哪里被使用了，果然在accessFlags.hpp文件中找到类似的名字。

```
ss AccessFlags VALUE_OBJ_CLASS_SPEC {  
friend class VMStructs;  
private:  
int _flags;  
public:  
/ Java access flags  
bool is_public      () const { return (_flags & JVM_ACC_PUBLIC      ) != 0; }  
bool is_private     () const { return (_flags & JVM_ACC_PRIVATE     ) != 0; }  
bool is_protected   () const { return (_flags & JVM_ACC_PROTECTED   ) != 0; }  
bool is_static      () const { return (_flags & JVM_ACC_STATIC      ) != 0; }  
bool is_final       () const { return (_flags & JVM_ACC_FINAL       ) != 0; }  
bool is_synchronized() const { return (_flags & JVM_ACC_SYNCHRONIZED) != 0; }  
bool is_super       () const { return (_flags & JVM_ACC_SUPER       ) != 0; }  
  
bool is_volatile     () const { return (_flags & JVM_ACC_VOLATILE     ) != 0; }  
  
bool is_transient   () const { return (_flags & JVM_ACC_TRANSIENT   ) != 0; }  
bool is_native      () const { return (_flags & JVM_ACC_NATIVE      ) != 0; }  
bool is_interface   () const { return (_flags & JVM_ACC_INTERFACE   ) != 0; }  
bool is_abstract    () const { return (_flags & JVM_ACC_ABSTRACT    ) != 0; }  
bool is_strict      () const { return (_flags & JVM_ACC_STRICT      ) != 0; }
```

通过 is\_volatile() 可以判断一个变量是否被volatile修饰，然后再全局搜"is\_volatile"被使用的地方，最后在 bytecodeInterpreter.cpp 文件中，找到putstatic字节码指令的解释器实现，里面有 is\_volatile() 方法。

```

CASE(_putfield):
CASE(_putstatic):
{
    u2 index = Bytes::get_native_u2(pc+1);
    ConstantPoolCacheEntry* cache = cp->entry_at(index);
    if (!cache->is_resolved((Bytecodes::Code)opcode)) {
        CALL_VM(InterpreterRuntime::resolve_get_put(THREAD, (Bytecodes::Code)opcode),
                handle_exception());
        cache = cp->entry_at(index);
    }
}

if VM_JVMTI
if /* VM_JVMTI */

// QQQ Need to make this as inlined as possible. Probably need to split all the bytecode cases
// out so c++ compiler has a chance for constant prop to fold everything possible away.

oop obj;
int count;
TosState tos_type = cache->flag_state();

count = -1;
if (tos_type == ltos || tos_type == dtos) {
    -count;
}

```

当然了，在正常执行时，并不会走这段逻辑，都是直接执行字节码对应的机器码指令，这段代码可以在debug的时候使用，不过最终逻辑是一样的。

其中cache变量是java代码中变量i在常量池缓存中的一个实例，因为变量i被volatile修饰，所以 cache->is\_volatile() 为真，给变量i的赋值操作由 release\_int\_field\_put 方法实现。

再来看看 release\_int\_field\_put 方法

```

Line void oopDesc::release_int_field_put(int offset, jint contents) {
OrderAccess::release_store(int_field_addr(offset), contents);
}

```

内部的赋值动作被包了一层， OrderAccess::release\_store 究竟做了魔法，可以让其它线程读到变量i的最新值。

```

Line void      OrderAccess::release_store(volatile jint* p, jint v) {
*p = v;
}

```

奇怪，在OrderAccess::release\_store的实现中，第一个参数强制加了一个volatile，很明显，这是c/c++的关键字。

c/c++中的volatile关键字，用来修饰变量，通常用于语言级别的 [memory barrier](#)，在"The C++ Programming Language"中，对volatile的描述如下：

A volatile specifier is a hint to a compiler that an object may change its value in ways not specified by the language so that aggressive optimizations must be avoided.

volatile是一种类型修饰符，被volatile声明的变量表示随时可能发生变化，每次使用时，都必须从变量i对应的内存地址读取，编译器对操作该变量的代码不再进行优化，下面写两段简单的c/c++代码验证一下

```
#include <iostream>

int foo = 10;
int a = 1;
int main(int argc, const char * argv[]) {
    // insert code here...
    a = 2;
    a = foo + 10;
    int b = a + 20;
    return b;
}
```

代码中的变量i其实是无效的，执行 `g++ -S -O2 main.cpp` 得到编译之后的汇编代码如下：

```
main:                                ## @main
    .cfi_startproc
# BB#0:
    pushq   %rbp
cfi0:
    .cfi_def_cfa_offset 16
cfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
cfi2:
    .cfi_def_cfa_register %rbp
    movl    _foo(%rip), %eax
    leal    10(%rax), %ecx
    movl    %ecx, _a(%rip)
    addl    $30, %eax
                                ## kill: %EAX<def> %EAX<kill>
    popq    %rbp
    retq
    .cfi_endproc

.section    __DATA,__data
.globl    _foo                                ## @foo
.p2align   2
foo:
    .long   10                                ## 0xa

.globl    _a                                ## @a
.p2align   2
a:
    .long   1                                ## 0x1
```

可以发现，在生成的汇编代码中，对变量a的一些无效负责操作果然都被优化掉了，如果在声明变量a时加上volatile

```

#include <iostream>

int foo = 10;
volatile int a = 1;
int main(int argc, const char * argv[]) {
    // insert code here...
    a = 2;
    a = foo + 10;
    int b = a + 20;
    return b;
}

```

再次生成汇编代码如下：

```

_main:                                     ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
_cfi0:
    .cfi_def_cfa_offset 16
_cfi1:|
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
_cfi2:
    .cfi_def_cfa_register %rbp
    movl    $2, _a(%rip) → a赋值为2
    movl    _foo(%rip), %eax
    addl    $10, %eax
    movl    %eax, _a(%rip) → a赋值为12
    movl    _a(%rip), %eax → 重新读取a
    addl    $20, %eax
    popq    %rbp
    retq
    .cfi_endproc

```

和第一次比较，有以下不同：

1、对变量a赋值2的语句，也保留了下来，虽然是无效的动作，所以volatile关键字可以禁止指令优化，其实这里发挥了编译器屏障的作用；

编译器屏障可以避免编译器优化带来的内存乱序访问的问题，也可以手动在代码中插入编译器屏障，比如下面的代码和加volatile关键字之后的效果是一样



```

#include <iostream>

int foo = 10;
int a = 1;
int main(int argc, const char * argv[]) {
    // insert code here...
    a = 2;
    __asm__ volatile (" : : : \"memory\"); //编译器屏障
    a = foo + 10;
    __asm__ volatile (" : : : \"memory\");
    int b = a + 20;
    return b;
}

```

编译之后，和上面类似

```

main:                                     ## @main
    .cfi_startproc
# BB#0:
    pushq   %rbp
cfi0:
    .cfi_def_cfa_offset 16
cfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
cfi2:
    .cfi_def_cfa_register %rbp
    movl    $2, _a(%rip)
    ## InlineAsm Start
    ## InlineAsm End
    movl    _foo(%rip), %eax
    addl    $10, %eax
    movl    %eax, _a(%rip)
    ## InlineAsm Start
    ## InlineAsm End
    movl    _a(%rip), %eax
    addl    $20, %eax
    popq    %rbp
    retq
    .cfi_endproc

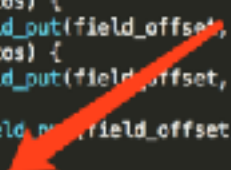
```

2、其中 `_a(%rip)` 是变量a的每次地址，通过 `movl $2, _a(%rip)` 可以把变量a所在的内存设置成2，关于RIP，可以查看 [x64下PIC的新寻址方式：RIP相对寻址](#)

所以，每次对变量a的赋值，都会写入到内存中；每次对变量的读取，都会从内存中重新加载。

感觉有点跑偏了，让我们回到JVM的代码中来。

```
// Now store the result
//
int field_offset = cache->f2_as_index();
if (cache->is_volatile()) {
    if (tos_type == itos) {
        obj->release_int_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == atos) {
        VERIFY_OOP(STACK_OBJECT(-1));
        obj->release_obj_field_put(field_offset, STACK_OBJECT(-1));
        OrderAccess::release_store(&BYTE_MAP_BASE[(uintptr_t)obj >> CardTableModRefB5::card_shift], 0);
    } else if (tos_type == btos) {
        obj->release_byte_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ltos) {
        obj->release_long_field_put(field_offset, STACK_LONG(-1));
    } else if (tos_type == ctos) {
        obj->release_char_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == stos) {
        obj->release_short_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ftos) {
        obj->release_float_field_put(field_offset, STACK_FLOAT(-1));
    } else {
        obj->release_double_field_put(field_offset, STACK_DOUBLE(-1));
    }
}
OrderAccess::storeload();
```



执行完赋值操作后，紧接着执行 `OrderAccess::storeload()`，这又是啥？

其实这就是经常会念叨的内存屏障，之前只知道念，却不知道是如何实现的。从CPU缓存结构分析中已经知道：一个load操作需要进入LoadBuffer，然后再去内存加载；一个store操作需要进入StoreBuffer，然后再写入缓存，这两个操作都是异步的，会导致不正确的指令重排序，所以在JVM中定义了一系列的内存屏障来指定指令的执行顺序。

JVM中定义的内存屏障如下，JDK1.7的实现

```
// Implementation of class OrderAccess.

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

- 1、loadload屏障 ( load1 , loadload , load2 )
- 2、loadstore屏障 ( load , loadstore , store )

这两个屏障都通过 `acquire()` 方法实现

```

line void OrderAccess::acquire() {
volatile intptr_t local_dummy;
#ifdef AMD64
__asm__ volatile ("movq 0(%%rsp), %0" : "=r" (local_dummy) : : "memory");
#else

```

其中 `__asm__`，表示汇编代码的开始。

`volatile`，之前分析过了，禁止编译器对代码进行优化。

把这段指令编译之后，发现没有看懂....最后的"memory"是编译器屏障的作用。

在LoadBuffer中插入该屏障，清空屏障之前的load操作，然后才能执行屏障之后的操作，可以保证load操作的数据在下个store指令之前准备好

3、storestore屏障 ( store1，storestore，store2 )

通过"release()"方法实现：

```

inline void OrderAccess::release() {
    // Avoid hitting the same cache-line from
    // different threads.
    volatile jint local_dummy = 0;
}

```

在StoreBuffer中插入该屏障，清空屏障之前的store操作，然后才能执行屏障之后的store操作，保证store1写入的数据在执行store2时对其它CPU可见。

4、storeload屏障 ( store，storeload，load )

对java中的volatile变量进行赋值之后，插入的就是这个屏障，通过"fence()"方法实现：

```

line void OrderAccess::fence() {
if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
__asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
#else
__asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
}
}

```

看到这个有没有很兴奋？

通过 `os::is_MP()` 先判断是不是多核，如果只有一个CPU的话，就不存在这些问题了。

storeload屏障，完全由下面这些指令实现

```
__asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
```

为了试验这些指令到底有什么用，我们再写点c++代码编译一下


```
#include <iostream>

int foo = 10;

int main(int argc, const char * argv[]) {
    // insert code here...
    volatile int a = foo + 10;
    // __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
    volatile int b = foo + 20;

    return 0;
}
```

为了变量a和b不被编译器优化掉，这里使用了volatile进行修饰，编译后的汇编指令如下：



```
fi2:
.cfi_def_cfa_register %rbp
movl    __foo(%rip), %eax
leal    10(%rax), %ecx
movl    %ecx, -8(%rbp)
addl    $20, %eax
movl    %eax, -4(%rbp)
xorl    %eax, %eax
popq    %rbp
retq
.cfi_endproc

.section    __DATA,__data
.globl    _foo          ## @foo
.p2align    2
foo:
.long    10             ## 0xa
```

从编译后的代码可以发现，第二次使用foo变量时，没有从内存重新加载，使用了寄存器的值。

把 \_\_asm\_\_ volatile \*\*\* 指令加上之后重新编译

```

Lcfi2:
    .cfi_def_cfa_register %rbp
    movl    foo(%rip), %eax
    addl    $10, %eax
    movl    %eax, -8(%rbp)
    ## InlineAsm Start
    lock
    addl    $0, (%rsp)
    ## InlineAsm End
    movl    _foo(%rip), %eax
    addl    $20, %eax
    movl    %eax, -4(%rbp)
    xorl    %eax, %eax
    popq    %rbp
    retq
    .cfi_endproc

    .section    __DATA,__data
    .globl    _foo
    .p2align    2
    foo:

```

相比之前，这里多了两个指令，一个lock，一个addl。

lock指令的作用是：在执行lock后面指令时，会设置处理器的LOCK#信号（这个信号会锁定总线，阻止其它CPU通过总线访问内存，直到这些指令执行结束），这条指令的执行变成原子操作，之前的读写请求都不能越过lock指令进行重排，相当于一个内存屏障。

还有一个：第二次使用foo变量时，从内存中重新加载，保证可以拿到foo变量的最新值，这是由如下指令实现

```
__asm__ volatile ( : : : "cc", "memory");
```

同样是编译器屏障，通知编译器重新生成加载指令(不可以从缓存寄存器中取)。

## 读取volatile变量

同样在 `bytecodeInterpreter.cpp` 文件中，找到getstatic字节码指令的解释器实现。

```

CASE(_getfield):
CASE(_getstatic):
{
    u2 index;
    ConstantPoolCacheEntry* cache;
    index = Bytes::get_native_u2(pc+1);

    // QQQ Need to make this as inlined as possible. Probably need to
    // split all the bytecode cases out so c++ compiler has a chance
    // for constant prop to fold everything possible away.

    cache = cp->entry_at(index);
    if (!cache->is_resolved((Bytecodes::Code)opcode)) {
        CALL_VM(InterpreterRuntime::resolve_get_put(THREAD, (Bytecodes::Code)opcode),
                handle_exception);
        cache = cp->entry_at(index);
    }

VM_JVMTI
/* VM_JVMTI */

    oop obj;
    if ((Bytecodes::Code)opcode == Bytecodes::_getstatic) {
        obj = (oop) cache->f1_as_instance();
        MORE_STACK(1); // Assume single slot push
    } else {
        obj = (oop) STACK_OBJECT(-1);
        CHECK_NULL(obj);
    }

    //
    // Now store the result on the stack
    //
    TosState tos_type = cache->flag_state();
    int field_offset = cache->f2_as_index();
    if (cache->is_volatile()) {
        if (tos_type == atos) {

```

通过 `obj->obj_field_acquire(field_offset)` 获取变量值

```

line oop oopDesc::obj_field_acquire(int offset) const {
return UseCompressedOops ?
    decode_heap_oop({narrowOop(
        OrderAccess::load_acquire(obj_field_addr<narrowOop>(offset)))
    : decode_heap_oop({oop(
        OrderAccess::load_ptr_acquire(obj_field_addr<oop>(offset))});

```

最终通过 `OrderAccess::load_acquire` 实现

```

inline jint OrderAccess::load_acquire(volatile jint* p) { return *p; }

```

底层基于C++的volatile实现，因为volatile自带了编译器屏障的功能，总能拿到内存中的最新值。

长按下方二维码关注占小狼



每一次的思考  
都想与你分享

小礼物走一走，来简书关注我

赞赏支持

java进阶干货

举报文章 © 著作权归作者所有



占小狼

写了 156869 字，被 13095 人关注，获得了 7516 个喜欢

关注

微信公众号：占小狼的博客 如果读完觉得有收获的话，欢迎点赞加关注

喜欢 | 84

更多分享

下载简书 App ▶

随时随地发现和创作内容





登录 后发表评论

20条评论

只看作者

按喜欢排序 按时间正序 按时间倒序



匠心零度

2楼 · 2017.12.26 19:15

太厉害，赶紧记笔记！

赞 回复

skillchen：@匠心零度 哈哈，大佬你也要加油啊，多更新点宝典

2017.12.28 06:34 回复

匠心零度：@skillchen 多谢支持，嗯嗯 加油

2017.12.28 06:37 回复

占小狼：@匠心零度 大佬你也要加油啊，多更新点宝典

2017.12.28 07:23 回复

添加新评论 | 还有1条评论，[展开查看](#)



心若有愧、四面是海

3楼 · 2017.12.26 19:40

在学校期间真应该好好学C和汇编的,哈哈

顺便提及几处个人认为可能是笔误的地方:

1. 代码中的变量i其实是无效的 => 代码中的变量a其实是无效的
2. 对变量a的一些无效负责操作果然都被优化掉了 => 赋值

赞 回复

占小狼：@心若有愧、四面是海 哈哈，真的是

2017.12.26 19:58 回复

心若有愧、四面是海：@占小狼 向您学习

2017.12.26 20:07 回复

添加新评论





埃尔文放弃

4楼 · 2017.12.27 00:23

一脸懵逼

赞 回复



吴世浩

5楼 · 2017.12.27 08:47

1:CPU执行store写数据时，把数据写到StoreBuffer中，待到某个适合的时间点；

2:每次使用时，都必须从变量i对应的内存地址读取

上面1是把改变数据先写StoreBuffer，那么还没刷到内存，2中说是从对应内存地址取？还没写到内存，从内存地址取的也不对吧？

赞 回复

占小狼：@吴世浩 2是针对c++中volatile修饰的变量，1是针对普通变量

2017.12.27 11:37 回复

添加新评论



土牛肉干

6楼 · 2017.12.28 06:57

写的太棒了👏狼哥威武，

赞 回复



IT聊天交友圈

7楼 · 2017.12.29 10:05

干货！收藏

赞 回复



statels0

8楼 · 2017.12.29 13:09

给大佬跪了

赞 回复



statels0

9楼 · 2017.12.29 13:13

狼哥 分享了

赞 回复

---



AbnerRao

10楼 · 2018.01.05 14:36

写的太棒了，简直看不懂，太深了☺加强学习

赞 回复

---



奔跑的IT男

11楼 · 2018.01.13 19:01

深入浅出，看的很舒服

赞 回复

---



马丁路德没有gold

12楼 · 2018.01.24 17:44

写的很好 之前只知道 内存屏障但是没了解这么深入 赞\(\ge\le)/

赞 回复

---



王小军008

13楼 · 2018.01.26 09:40

可怕

赞 回复

---



goforloneliness

14楼 · 2018.02.09 04:37

吾荒废的c和汇编就靠汝来拯救了

赞 回复