

何登成的技术博客

追求技术的道路上，10年如一日

搜索

- [首页](#)
- [关于我](#)

[RSS 订阅](#)

© 2012-2014 何登成的技术博客

C/C++ Volatile关键词深度剖析

十二 2nd, 2013

[发表评论](#) | [Trackback](#)

[1 背景 1](#)

[2 Volatile: 易变的 1](#)

[2.1 小结 2](#)

[3 Volatile: 不可优化的 3](#)

[3.1 小结 4](#)[4 Volatile: 顺序性 4](#)[4.1 happens-before 6](#)[4.2 小结 7](#)[5 Volatile: Java增强 8](#)[6 Volatile的起源 9](#)[7 参考资料 9](#)

1. 背景

前几天，发了一条如下的微博 (关于C/C++ Volatile关键词的使用建议):



此微博，引发了朋友们的大量讨论：赞同者有之；批评者有之；当然，更多的朋友，是希望我能更详细的解读C/C++ Volatile关键词，来佐证我的微博观点。而这，正是我写这篇博文的初衷：本文，将详细分析C/C++ Volatile关键词的功能 (有多种功能)、Volatile关键词在多线程编程中存在的问题、Volatile关键词与编译器/CPU的关系、C/C++ Volatile与Java Volatile的区别，以及Volatile关键词的起源，希望对大家更

好的理解、使用C/C++ Volatile，有所帮助。

Volatile，词典上的解释为：易失的；易变的；易挥发的。那么用这个关键词修饰的C/C++变量，应该也能够体现出”易变”的特征。大部分人认识Volatile，也是从这个特征出发，而这也是本文揭秘的C/C++ Volatile的第一个特征。

1. Volatile：易变的

在介绍C/C++ Volatile关键词的”易变”性前，先让我们看看以下的两个代码片段，以及他们对应的汇编指令 (以下用例的汇编代码，均为VS 2008编译出来的Release版本)：

- **测试用例一：非Volatile变量**

代码	汇编
<pre>void main () { int a = 5; int b = 10; int c = 20; int d; scanf("%d", &c); a = fn(c); b = a + 1; d = fn(b); cout << a << b << c << d; }</pre>	<pre>call dword ptr [__imp__scanf (12A20A8h)] mov eax,dword ptr [esp+8] add esp,8 lea ecx,[eax+1] </pre>

b = a + 1;这条语句，对应的汇编指令是：lea ecx, [eax + 1]。由于变量a，在前一条语句a = fn(c)执行时，被缓存在了寄存器eax中，因此b = a + 1；语句，可以直接使用仍旧在寄存器eax中的a，来进行计算，对应的也就是汇编：[eax + 1]。

- 测试用例二：Volatile变量

代码

汇编

```
void main ()
{
    volatile int a = 5;
    int        b = 10;
    int        c = 20;
    int        d;

    scanf("%d", &c);

    a = fn(c);

    b = a + 1;

    d = fn(b);

    cout << a << b << c << d;
}
```

```
mov     ecx,dword ptr [esp+8]
mov     dword ptr [esp+0Ch],ecx

mov     eax,dword ptr [esp+0Ch]
...
inc     eax
```

与测试用例一唯一的不同之处，是变量a被设置为volatile属性，一个小小的变化，带来的是汇编代码上很大的变化。a = fn(c)执行后，寄存器ecx中的a，被写回内存：mov dword ptr [esp+0Ch], ecx。然后，在执行b = a + 1; 语句时，变量a有重新被从内存中读取出来：mov eax, dword ptr [esp + 0Ch]，而不再直接使用寄存器ecx中的内容。

1. 小结

从以上的两个用例，就可以看出C/C++ Volatile关键词的第一个特性：**易变性**。所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取。volatile的这个特性，相信也是大部分朋友所了解的特性。

在了解了C/C++ Volatile关键词的”易变”特性之后，再让我们接着继续来剖析Volatile的下一个特性：”不可优化”特性。

1. Volatile：不可优化的

与前面介绍的”易变”性类似，关于C/C++ Volatile关键词的第二个特性：”不可优化”性，也通过两个对比的代码片段来说明：

- 测试用例三：非Volatile变量

代码	汇编
<pre>void main () { int a; int b; int c; a = 1; b = 2; c = 3; printf("%d, %d, %d", a, b, c); }</pre>	<pre>push 3 push 2 push 1 call ...</pre>

在这个用例中，非volatile变量a, b, c全部被编译器优化掉了 (optimize out)，因为编译器通过分析，发觉a, b, c三个变量是无用的，可以进行常量替换。最后的汇编代码相当简介，高效率。

- 测试用例四：Volatile变量

代码	汇编
<pre>void main () { volatile int a; volatile int b; volatile int c; a = 1; b = 2; c = 3; printf("%d, %d, %d", a, b, c); }</pre>	<pre>mov eax, dword ptr [esp] mov ecx, dword ptr [esp+4] mov edx, dword ptr [esp+8] push eax push ecx push edx call ...</pre>

测试用例四，与测试用例三类似，不同之处在于，a，b，c三个变量，都是volatile变量。这个区别，反映到汇编语言中，就是三个变量仍旧存在，需要将三个变量从内存读入到寄存器之中，然后再调用printf()函数。

1. 小结

从测试用例三、四，可以总结出C/C++ Volatile关键词的第二个特性：“不可优化”特性。volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。相对于前面提到的第一个特性：“易变”性，“不可优化”特性可能知晓的人会相对少一些。但是，相对于下面提到的C/C++ Volatile的第三个特性，无论是“易变”性，还是“不可优化”性，都是Volatile关键词非常流行的概念。

1. Volatile：顺序性

C/C++ Volatile关键词前面提到的两个特性，让Volatile经常被解读为一个为多线程而生的关键词：一个全局变量，会被多线程同时访问/修

改，那么线程内部，就不能假设此变量的不变性，并且基于此假设，来做一些程序设计。当然，这样的假设，本身并没有什么问题，多线程编程，并发访问/修改的全局变量，通常都会建议加上Volatile关键词修饰，来防止C/C++编译器进行不必要的优化。但是，很多时候，C/C++ Volatile关键词，在多线程环境下，会被赋予更多的功能，从而导致问题的出现。

回到本文背景部分我的那篇微博，我的这位朋友，正好犯了一个这样的问题。其对C/C++ Volatile关键词的使用，可以抽象为下面的伪代码：

代码

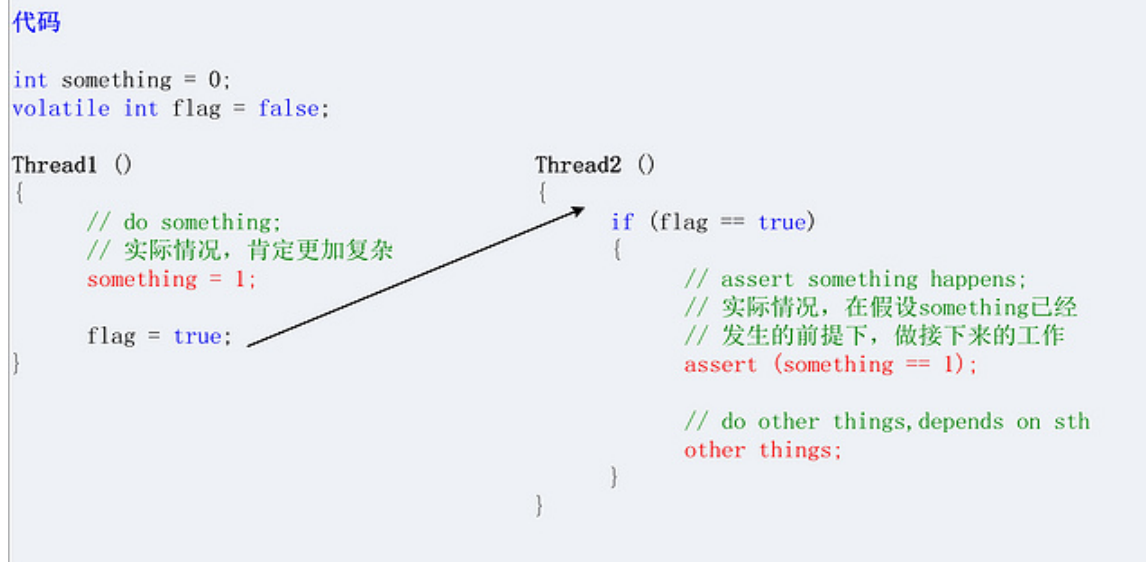
```
int something = 0;
volatile int flag = false;

Thread1 ()
{
    // do something;
    // 实际情况，肯定更加复杂
    something = 1;

    flag = true;
}

Thread2 ()
{
    if (flag == true)
    {
        // assert something happens;
        // 实际情况，在假设something已经
        // 发生的前提下，做接下来的工作
        assert (something == 1);

        // do other things, depends on sth
        other things;
    }
}
```

A diagram showing two threads, Thread1 and Thread2. Thread1 has a line of code 'flag = true;' with an arrow pointing from it to the 'if (flag == true)' condition in Thread2, illustrating the state change and its effect on the other thread's execution flow.

这段伪代码，声明另一个Volatile的flag变量。一个线程(Thread1)在完成一些操作后，会修改这个变量。而另外一个线程(Thread2)，则不断读取这个flag变量，由于flag变量被声明了volatile属性，因此编译器在编译时，并不会每次都从寄存器中读取此变量，同时也不会通过各种激进的优化，直接将if (flag == true)改写为if (false == true)。只要flag变量在Thread1中被修改，Thread2中就会读取到这个变化，进入if条件判断，然后进入if内部进行处理。在if条件的内部，**由于flag == true，那么假设Thread1中的something操作一定已经完成了**，在基于这个假设的基础上，继续进行下面的other things操作。

通过将flag变量声明为volatile属性，很好的利用了本文前面提到的C/C++ Volatile的两个特性：“易变”性；“不可优化”性。按理说，这是一个对于volatile关键词的很好应用，而且看到这里的朋友，也可以去检查检查自己的代码，我相信肯定会有这样的使用存在。

但是，这个多线程下看似对于C/C++ Volatile关键词完美的应用，实际上却是有大问题的。问题的关键，就在于前面标红的文字：**由于flag = true，那么假设Thread1中的something操作一定已经完成了**。flag == true，为什么能够推断出Thread1中的something一定完成了？其实既然我把这作为一个错误的用例，答案是一目了然的：**这个推断不能成立，你不能假设看到flag == true后，flag = true;这条语句前面的something一定已经执行完成了**。这就引出了C/C++ Volatile关键词的第三个特性：顺序性。

同样，为了说明C/C++ Volatile关键词的”顺序性”特征，下面给出三个简单的用例 (注：与上面的测试用例不同，下面的三个用例，基于的是Linux系统，使用的是”GCC: (Debian 4.3.2-1.1) 4.3.2”):

- 测试用例五：非Volatile变量

代码	汇编
<pre>// cordering.c int A, B; void foo() { A = B + 1; B = 0; }</pre>	<pre>gcc -O2 -S -masm=intel cordering.c cat cordering.s mov eax, DWORD PTR B[rip] mov DWORD PTR B[rip], 0 add eax, 1 mov DWORD PTR A[rip], eax ret</pre>

一个简单的示例，全局变量A, B均为非volatile变量。通过gcc O2优化进行编译，你可以惊奇的发现，A, B两个变量的赋值顺序被调换了！！！在对应的汇编代码中，B = 0语句先被执行，然后才是A = B + 1语句被执行。

在这里，我先简单的介绍一下C/C++编译器最基本优化原理：**保证一段程序的输出，在优化前后无变化**。将此原理应用到上面，可以发现，虽然gcc优化了A, B变量的赋值顺序，但是foo()函数的执行结果，优化前后没有发生任何变化，仍旧是A = 1; B = 0。因此这么做是可行的。

- 测试用例六：一个Volatile变量

代码	汇编
<pre>// cordering.c int A; volatile int B; void foo() { A = B + 1; B = 0; }</pre>	<pre>gcc -O2 -S -masm=intel cordering.c mov eax, DWORD PTR B[rip] mov DWORD PTR B[rip], 0 add eax, 1 mov DWORD PTR A[rip], eax ret</pre>

此测试，相对于测试用例五，最大的区别在于，变量B被声明为volatile变量。通过查看对应的汇编代码，B仍旧被提前到A之前赋值，Volatile变量B，并未阻止编译器优化的发生，编译后仍旧发生了乱序现象。

如此看来，C/C++ Volatile变量，与非Volatile变量之间的操作，是可能被编译器交换顺序的。

通过此用例，已经能够很好的说明，本章节前面，通过flag == true，来假设something一定完成是不成立的。在多线程下，如此使用volatile，会产生很严重的问题。但是，这不是终点，请继续看下面的测试用例七。

• 测试用例七：两个Volatile变量

代码	汇编
<pre>// cordering.c volatile int A; volatile int B; void foo() { A = B + 1; B = 0; }</pre>	<pre>gcc -O2 -S -masm=intel cordering.c mov eax, DWORD PTR B[rip] add eax, 1 mov DWORD PTR A[rip], eax mov DWORD PTR B[rip], 0 ret</pre>

同时将A，B两个变量都声明为volatile变量，再看看对应的汇编。奇迹发生了，A，B赋值乱序的现象消失。此时的汇编代码，与用户代码顺序高度一致，先赋值变量A，然后赋值变量B。

如此看来，C/C++ Volatile变量间的操作，是不会被编译器交换顺序的。

1. happens-before

通过测试用例六，可以总结出：C/C++ Volatile变量与非Volatile变量间的操作顺序，有可能被编译器交换。因此，上面多线程操作的伪代码，在实际运行的过程中，就有可能变成下面的顺序：

```
代码

int something = 0;
volatile int flag = false;

Thread1 ()
{
    // do something;
    // 实际情况，肯定更加复杂

    flag = true;
    something = 1;
}

Thread2 ()
{
    if (flag == true)
    {
        // assert something happens;
        // 实际情况，在假设something已经

        // 发生的前提下，做接下来的工作
        assert (something == 1);

        // do other things, depends on sth
        other things;
    }
}
```

顺序交换

由于Thread1中的代码执行顺序发生变化，flag = true被提前到something之前进行，那么整个Thread2的假设全部失效。由于something未执行，但是Thread2进入了if代码段，整个多线程代码逻辑出现问题，导致多线程完全错误。

细心的读者看到这里，可能要提问，根据测试用例七，C/C++ Volatile变量间，编译器是能够保证不交换顺序的，那么能不能将something中所有的变量全部设置为volatile呢？这样就阻止了编译器的乱序优化，从而也就保证了这个多线程程序的正确性。

针对此问题，很不幸，仍旧不行。将所有的变量都设置为volatile，首先能够阻止编译器的乱序优化，这一点是可以肯定的。但是，别忘了，编译器编译出来的代码，最终是要通过CPU来执行的。目前，市场上有各种不同体系架构的CPU产品，CPU本身为了提高代码运行的效率，也会对代码的执行顺序进行调整，这就是所谓的CPU Memory Model (CPU内存模型)。关于CPU的内存模型，可以参考这些资料：[Memory Ordering From Wiki](#)；[Memory Barriers Are Like Source Control Operations From Jeff Preshing](#)；[CPU Cache and Memory Ordering](#)

[From 何登成](#)。下面，是截取自Wiki上的一幅图，列举了不同CPU架构，可能存在的指令乱序。

Memory ordering in some architectures^{[2][3]}

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	xSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

从图中可以看到，X86体系(X86, AMD64)，也就是我们目前使用最广的CPU，也会存在指令乱序执行的行为：StoreLoad乱序，读操作可以提前到写操作之前进行。

因此，回到上面的例子，哪怕将所有的变量全部都声明为volatile，哪怕杜绝了编译器的乱序优化，但是针对生成的汇编代码，CPU有可能仍旧会乱序执行指令，导致程序依赖的逻辑出错，volatile对此无能为力。

其实，针对这个多线程的应用，真正正确的做法，是构建一个happens-before语义。关于happens-before语义的定义，可参考文章：[The Happens-Before Relation](#)。下面，用图的形式，来展示happens-before语义：

代码

```
int something = 0;
volatile int flag = false;
```

Thread1 ()

{

```
// do something;
// 实际情况, 肯定更加复杂
something = 1;
```

```
flag = true;
```

}

Thread2 ()

{

```
if (flag == true)
```

{

```
// assert something happens;
// 实际情况, 在假设something已经
// 发生的前提下, 做接下来的工
assert (something == 1);
```

```
// do other things, depends on sth
other things;
```

}

}

happens-
before

如图所示, 所谓的happens-before语义, 就是保证Thread1代码块中的所有代码, 一定在Thread2代码块的第一条代码之前完成。当然, 构建这样的语义有很多方法, 我们常用的Mutex、Spinlock、RWLock, 都能保证这个语义 (关于happens-before语义的构建, 以及为什么锁能保证happens-before语义, 以后专门写一篇文章进行讨论)。但是, C/C++ Volatile关键词不能保证这个语义, 也就意味着C/C++ Volatile关键词, 在多线程环境下, 如果使用的不够细心, 就会产生如同我这里提到的错误。

1. 小结

C/C++ Volatile关键词的第三个特性: **“顺序性”**, 能够保证Volatile变量间的顺序性, 编译器不会进行乱序优化。Volatile变量与非Volatile变量的顺序, 编译器不保证顺序, 可能会进行乱序优化。同时, C/C++ Volatile关键词, 并不能用于构建happens-before语义, 因此在进行多线程程序设计时, 要小心使用volatile, 不要掉入volatile变量的使用陷阱之中。

1. Volatile: Java增强

在介绍了C/C++ Volatile关键词之后，再简单介绍一下Java的Volatile。与C/C++的Volatile关键词类似，Java的Volatile也有这三个特性，但最大的不同在于：第三个特性，“顺序性”，Java的Volatile有很极大的增强，Java Volatile变量的操作，附带了Acquire与Release语义。所谓的Acquire与Release语义，可参考文章：[Acquire and Release Semantics](#)。(这一点，后续有必要的话，可以写一篇文章专门讨论)。Java Volatile所支持的Acquire、Release语义，如下：

- 对于Java Volatile变量的写操作，带有Release语义，所有Volatile变量写操作之前的针对其他任何变量的读写操作，都不会被编译器、CPU优化后，乱序到Volatile变量的写操作之后执行。
- 对于Java Volatile变量的读操作，带有Acquire语义，所有Volatile变量读操作之后的针对其他任何变量的读写操作，都不会被编译器、CPU优化后，乱序到Volatile变量的读操作之前进行。

通过Java Volatile的Acquire、Release语义，对比C/C++ Volatile，可以看出，Java Volatile对于编译器、CPU的乱序优化，限制的更加严格了。Java Volatile变量与非Volatile变量的一些乱序操作，也同样被禁止。

由于Java Volatile支持Acquire、Release语义，因此Java Volatile，能够用来构建happens-before语义。也就是说，前面提到的C/C++ Volatile在多线程下错误的使用场景，在Java语言下，恰好就是正确的。如下图所示：

代码

```
int something = 0;
volatile int flag = false;
```

Thread1 ()

{

```
// do something;
// 实际情况, 肯定更加复杂
something = 1;
Release: sth不会优化到flag写之后
flag = true;
```

}

Thread2 ()

{

```
if (flag == true)
{
    Acquire: oth不会优化到flag读之前
    // assert something happens;
    // 实际情况, 在假设something已经
    // 发生的前提下, 做接下来的工
    assert (something == 1);

    // do other things, depends on sth
    other things;
}
```

}

happens-
before

1. Volatile的起源

C/C++的Volatile关键词，有三个特性：易变性；不可优化性；顺序性。那么，为什么Volatile被设计成这样呢？要回答这个问题，就需要从Volatile关键词的产生说起。（注：这一小节的内容，参考自[C++ and the Perils of Double-Checked Locking](#)论文的第10章节：volatile: A Brief History。这是一篇顶顶好的论文，值得多次阅读，强烈推荐！）

Volatile关键词，最早出现于19世纪70年代，被用于处理memory-mapped I/O (MMIO)带来的问题。在引入MMIO之后，一块内存地址，既有可能是真正的内存，也有可能被映射到一个I/O端口。相对的，读写一个内存地址，既有可能操作内存，也有可能读写的是一个I/O设备。MMIO为什么需要引入Volatile关键词？考虑如下的一个代码片段：


```
unsigned int *p = GetMagicAddress ();  
unsigned int a, b;  
  
a = *p;                (1)  
b = *p;                (2)  
  
*p = a;                (3)  
*p = b;                (4)
```

在此代码片段中，指针p既有可能指向一个内存地址，也有可能指向一个I/O设备。如果指针p指向的是I/O设备，那么(1)，(2)中的a，b，就会接收到I/O设备的连续两个字节。但是，p也有可能指向内存，此时，编译器的优化策略，就可能会判断出a，b同时从同一内存地址读取数据，在做完(1)之后，直接将a赋值给b。对于I/O设备，需要防止编译器做这个优化，不能假设指针b指向的内容不变——易变性。

同样，代码(3)，(4)也有类似的问题，编译器发现将a，b同时赋值给指针p是无意义的，因此可能会优化代码(3)中的赋值操作，仅仅保留代码(4)。对于I/O设备，需要防止编译器将写操作给彻底优化消失了——“不可优化”性。

对于I/O设备，编译器不能随意交互指令的顺序，因为顺序一变，写入I/O设备的内容也就发生了变化了——“顺序性”。

基于MMIO的这三个需求，设计出来的C/C++ Volatile关键词，所含有的特性，也就是本文前面分析的三个特性：易变性；不可优化性；顺序性。

1. 参考资料

[1] Wiki. [Volatile variable](#).

[2] Wiki. [Memory ordering](#).

[3] Scott Meyers; Andrei Alexandrescu. [C++ and the Perils of Double-Checked Locking](#).

- [4] Jeff Preshing. [Memory Barriers Are Like Source Control Operations](#).
- [5] Jeff Preshing. [The Happens-Before Relation](#).
- [6] Jeff Preshing. [Acquire and Release Semantics](#).
- [7] 何登成. [CPU Cache and Memory Ordering——并发程序设计入门](#).
- [8] Bartosz Milewski. [Who ordered sequential consistency?](#)
- [9] Andrew Haley. [What are we going to do about volatile?](#)
- [10] Java Glossary. [volatile](#).
- [11] stackoverflow. [Why is volatile not considered useful in multithreaded C or C++ programming?](#)
- [12] msdn. [Volatile fields](#).
- [13] msdn. [volatile \(C++\)](#).
- [14] 刘未鹏. [《C++0x漫谈》系列之：多线程内存模型](#).

标签: [C/C++](#), [Java](#), [volatile](#)
[MySQL 加锁处理分析](#) [排队论及其应用浅析](#)

1. 
hailong
十二 2nd, 201318:18
[回复](#) | [引用](#) | [#1](#)

Volatile 变量和non-Volatile变量 会被编译器优化掉，之前还真不知道！长见识了。如果这样，禁止掉优化嘛，也可以

- o 
hedengcheng
十二 2nd, 201318:40
[回复](#) | [引用](#) | [#2](#)

禁掉优化，你可以试试。debug与release版本的性能差距有多大，很多就是编译器优化的功劳。



eagle.dai

十二 2nd, 201322:01

[回复](#) | [引用](#) | [#3](#)

```
#pragma optimize("", off)
```

...

```
#pragma optimize("", on)
```

VC下面可以禁止一块代码优化，GCC下面也应该有类似的方法



eagle.dai

十二 2nd, 201322:03

[回复](#) | [引用](#) | [#4](#)

当然，这个只能解决编译器的reordering带了的的问题，CPU也会做这种事情



hedengcheng

十二 2nd, 201322:40

[回复](#) | [引用](#) | [#5](#)

禁用编译器的优化，并不是一个好方法，毕竟，对于代码性能来说，编译器的优化，提升太明显了。

2.



jiayy

十二 2nd, 201319:12

[回复](#) | [引用](#) | [#6](#)

如果是gcc编译器，在取flag的地方加上 BARRIER 指令是否就可以保证顺序性了？

```
#define BARRIER() do{ asm volatile("" ::: "memory");}while(0)
```



hedengcheng
十二 2nd, 201322:38
[回复](#) | [引用](#) | [#7](#)

barrier是可以的。不过，你的这个是compiler barrier，更强一点的话，应该同时加上CPU barrier，例如mfence指令。



Yatao Li
十二 3rd, 201300:57
[回复](#) | [引用](#) | [#8](#)

lock xor eax, eax比mfence快

3.

zz
十二 2nd, 201321:09
[回复](#) | [引用](#) | [#9](#)

赞一下！~ 不过最后Java volatile 图片中左右两边acquire和release是不是弄反了啊~ 左边写flag应该是release，右边读flag应该是acquire吧~ 😊



hedengcheng
十二 2nd, 201322:37
[回复](#) | [引用](#) | [#10](#)

错了，谢谢指正！

4.

Thomson
十二 2nd, 201323:23
[回复](#) | [引用](#) | [#11](#)

同赞！非常全面的总结。使用mfence来实现happen-before比使用lock的性能要好吧？中间提到foo函数里面的memory write reordering, foo执行完后 A=1, B=0, 但是因为进入foo之前A,B的值可能已经不是初值，第一次看到这里还想了一下为什么一定会是1和0. 这里的reordering主要也就是把读和写隔开，消除相邻指令的data dependence, 选gcc是因为VC++不会做这个优化吗？



o

hedengcheng
十二 3rd, 201308:26
[回复](#) | [引用](#) | [#12](#)

vc，我没做这个测试，因为原来这个用例已经有了，我就直接在gcc下做了钱。



5.

hiproz
十二 3rd, 201301:06
[回复](#) | [引用](#) | [#13](#)

清晰 明了，一气呵成。特支持一下，期待更多精良大作。



6.

xiaoyang
十二 3rd, 201308:10
[回复](#) | [引用](#) | [#14](#)

好文章，支持，以前觉得自己入门了，现在发现还很远，最起码的几个名词都没听说过。



7.

东青
十二 3rd, 201309:46
[回复](#) | [引用](#) | [#15](#)

很不错，尤其是最后一段从volatile关键字的起源说起，让我们知道了他的来龙去脉，更好的加深了理解。



8.

victor
十二 3rd, 201309:53

[回复](#) | [引用](#) | [#16](#)

lz你好，第一个例子我用gcc -S t.c 编译为什么观察不到你提到的那种变化，这个与你的那个fn（）函数没关系吧，想知道为什么



o

hedengcheng
十二 3rd, 2013 11:09

[回复](#) | [引用](#) | [#17](#)

第一个例子，我用的是vc编译器，gcc没试过，因此尚不知道原因。



■

Adam
十二 9th, 2013 17:44

[回复](#) | [引用](#) | [#18](#)

是不是用-S只是进行了汇编，没有经过优化阶段，应该对编译完生成的.o文件，再反汇编看。



o

hedengcheng
十二 3rd, 2013 11:10

[回复](#) | [引用](#) | [#19](#)

跟fn函数是有关的，我可以把这个test完整代码贴出来。



9.

[tengfei](#)

十二 3rd, 2013 12:46

[回复](#) | [引用](#) | [#20](#)

好文章啊，受益了。赞。



10.

catmonkeyxu

十二 3rd, 2013 14:01

[回复](#) | [引用](#) | [#21](#)

受教了。那么能够实现happens-before语义性能最快的方式是什么呢？似乎就是lock指令？



hedengcheng
十二 3rd, 201318:41

[回复](#) | [引用](#) | [#22](#)

嗯，其实个人感觉性能真没有那么明显的差别，更大的差别，还是在于对和错。

11.



shunruo
十二 3rd, 201321:56

[回复](#) | [引用](#) | [#23](#)

VC中，volatile 是带Acquire 和 release 语义的。

12.



colin陶
十二 5th, 201313:26

[回复](#) | [引用](#) | [#24](#)

太赞了...菜鸟路过...^^学习学习,转走了哈,大牛



colin陶
十二 5th, 201313:27

[回复](#) | [引用](#) | [#25](#)

http://blog.csdn.net/lin_credible/article/details/17128057

13.



zym
十二 13th, 201321:50

[回复](#) | [引用](#) | [#26](#)

对于测试用例五，测试了一下gcc 4.7.2也是如此。准确的说应该是下面这个顺序：

1. A=B

2.B=0

3.A=A+1

可见B为0时A还没有加1！但我个人认为这里的gcc的优化是有严重问题的，甚至可以当作是bug了。

同时测试了clang 3.3，即使在-O2下也绝对不会把B=0优化到A=B+1前面。



hedengcheng

十二 15th, 2013 17:17

[回复](#) | [引用](#) | [#27](#)

这个不算bug。编译器只要保证你程序达到的效果不变，中间的执行过程，是可以任意优化的。你可以试试，一段简单的代码，分别用debug，o1-3 release编译。debug，o1可能还跟代码基本一致，但是，o2-3跟代码的顺序，差异就会很大了。



[Wizmann](#)

四 24th, 2014 09:17

[回复](#) | [引用](#) | [#28](#)

c++不知道有多线程这个东西，所以只保证单线程下运行的正确。（这个问题应该在C++11得到了改变

14.



邓波

十二 18th, 2013 16:29

[回复](#) | [引用](#) | [#29](#)

多线程访问共享数据如果涉及写操作当然要加锁。这不是明摆着么？呵呵。

15.




fusijie

十二 21st, 2013 08:32

[回复](#) | [引用](#) | [#30](#)

赞，令人十分享受的文章。

16. [并发编程系列之一：锁的意义 | 何登成的技术博客](#)
十二 24th, 201317:21
[#31](#)
17. [2013年个人微博推荐技术资料汇总 | 何登成的技术博客](#)
十二 25th, 201322:08
[#32](#)
18. 
leo
二 13th, 201406:41
[回复](#) | [引用](#) | [#33](#)

对x86而言，貌似那个volatile例子不太适合，几乎不太可能出现你说的问题。
x86下，store之间不会有乱序出现，所以不会出现 flag = true then something = 1。

- o 
hedengcheng
二 13th, 201412:47
[回复](#) | [引用](#) | [#34](#)

嗯，x86不会出现storestore乱序

19. 
clr
三 15th, 201422:24
[回复](#) | [引用](#) | [#35](#)

关于顺序性，我不确定你说的“CPU乱序执行”会影响volatile 变量之间的执行顺序。正如你后面所说的，volatile一开始是由MMIO引入，所以，不管是CPU还是编译器都应该能够确保volatile变量间执行的顺序性，否则，往某一个端口的输出或者读取就会仍然无法得到保证。

20. 
莫铭
三 22nd, 201413:17

[回复](#) | [引用](#) | [#36](#)

接触到很多东西，以前模糊不明的也稍清晰点了，非常感谢，期待更多的好文章。

21.



aCayF

五 18th, 201420:11

[回复](#) | [引用](#) | [#37](#)

提一个疑惑，还望前辈能帮忙解答一下，前面所说的volatile关键字会强制将数据写到内存中去，这里强制的意思是就算cache用了write-back的方式，在这次操作中也会保证被强制写到内存而不只写到cache为止吗？

o



hedengcheng

五 21st, 201418:41

[回复](#) | [引用](#) | [#38](#)

如果你将cache当作内存的一部分，就可以理解了。所谓的强制写内存，读内存，是相对于register而言的。

22.



[apprentice89](#)

五 21st, 201410:03

[回复](#) | [引用](#) | [#39](#)

清晰易懂，非常感谢！

23.



OWR

七 1st, 201411:55

[回复](#) | [引用](#) | [#40](#)

Very Good!

Thank you

昵称 (必填)

电子邮箱 (我们会为您保密) (必填) 网址[订阅评论](#)

Recent Posts

- [一个最不可思议的MySQL死锁分析](#)
- [2013年个人微博推荐技术资料汇总](#)
- [并发编程系列之一：锁的意义](#)
- [MySQL 加锁处理分析](#)
- [C/C++ Volatile关键词深度剖析](#)

Tag Cloud

[2PC](#) [5.6](#) [add index](#) [AIO](#) [B+-Tree](#) [Block](#) [Buffer Pool](#) [Checkpoint](#) [CPU](#) [Crash Recovery](#) [Database](#) [Data Cache](#) [Deadlock](#) [Double Write](#) [Group Commit](#) [InnoDB](#) [Insert Buffer](#) [join](#) [Linux](#) [Lock](#) [MariaDB](#) [MVCC](#) [MySQL](#) [MySQL BUG](#) [online](#) [Oracle](#) [Page](#) [Percona](#) [Range Query](#) [ReadView](#) [Redo](#) [Rollback Segment](#) [Row](#) [transaction](#) [Trasaction](#) [Undo](#) [XA](#) [XtraDB](#) [中断](#) [分布式事务](#) [多版本](#) [数据库](#) [数据库内核分享](#) [日志](#) [软中断](#)

Categories

- [C/C++](#) (2)
- [Falcon](#) (1)
- [Hardware](#) (3)
- [Optimizer](#) (2)
- [Performance](#) (1)
- [Programming](#) (7)
- [Test](#) (1)

- [TPCC](#) (1)
- [数据库](#) (41)
 - [InnoDB](#) (34)
 - [MySQL Server](#) (27)
 - [Oracle](#) (6)
 - [PostgreSQL](#) (1)
- [数据库内核分享](#) (14)
- [数据库研发](#) (6)
- [杂谈](#) (3)

Archives

- [2014年一月](#)
- [2013年十二月](#)
- [2013年十一月](#)
- [2013年七月](#)
- [2013年四月](#)
- [2013年三月](#)
- [2013年二月](#)
- [2013年一月](#)
- [2012年十二月](#)
- [2012年十一月](#)
- [2012年十月](#)
- [2012年九月](#)
- [2012年八月](#)
- [2012年七月](#)
- [2012年六月](#)
- [2012年五月](#)
- [2012年四月](#)

Blogroll

- [ACM Queue](#)
- [All Things Distributed](#)
- [Bartosz Milewski's Programming Cafe](#)
- [Brendan's blog](#)

- [Dimitrik's Weblog](#)
- [Dr.Dobb's](#)
- [Gustavo Duarte](#)
- [High Scalability](#)
- [Jeremy Cole's Blog](#)
- [Jonathan Lewis's Oracle Scratchpad](#)
- [LinkedIn Engineering](#)
- [Mark Callaghan's High Availability MySQL](#)
- [Mechanical Sympathy](#)
- [MySQL Performance Blog](#)
- [Paul E. McKenney's Journal](#)
- [Perspectives](#)
- [preshing on programming](#)
- [Sutter's Mill](#)
- [The Pith of Performance](#)
- [The Twitter Engineering Blog](#)
- [Transactions on InnoDB](#)
- [Wired Enterprise – IT Happens](#)
- [云风的Blog](#)
- [左耳朵耗子的酷壳](#)
- [旺旺的数据库思考者之家](#)
- [霸爷的系统技术非业余研究](#)
- [首席的弯曲评论](#)

Powered by [WordPress](#) | Theme by [NeoEase](#) | Valid [XHTML 1.1](#) and [CSS 3](#)

- [注册](#)
- [登录](#)
- [置顶](#)