C/C++ Volatile 关键词剖析

何登成

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。

2 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

void main ()
{
int a = 5;
int b = 10;
int c = 20;
int d;
 	
scanf("%d", &c);
 	
a = fn(c);
	call dword ptr [__imp__scanf (12A20A8h)]
	mov eax,dword ptr [esp+8]
	add esp,8

b = a + 1;	lea ecx,[eax+1]

d = fn(b);
 	
cout << a << b << c << d;
}	

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

b = a + 1;	mov eax,dword ptr [esp+0Ch] ... inc eax

d = fn(b);
 	
cout << a << b << c << d;
}	

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。

2.1 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。

3 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

void main ()
{
int a;
int b;
int c;
 	
a = 1;
b = 2;
c = 3;
 	
printf("%d, %d, %d", a, b, c);
}
	push 3
	push 2
	push 1
	call ...

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

void main ()
{
      volatile int      a;
      volatile int      b;
      volatile int      c;

      a = 1;                            mov       eax,dword ptr [esp]
      b = 2;                            mov       ecx,dword ptr [esp+4]
      c = 3;                            mov       edx,dword ptr [esp+8]

      printf("%d, %d, %d", a, b, c);    push      eax
                                        push      ecx
                                        push      edx
                                        call      ...
}
```

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，然后在调用 printf()函数。

3.1 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。

4 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;
      }
}
```

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

```
// cordering.c

int A, B;
void foo()
{
      A = B + 1;
      B = 0;
}
```

汇编

```
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
```

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
// cordering.c	gcc -O2 -S -masm=intel cordering.c

int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	mov DWORD PTR B[rip], 0
	add eax, 1
	mov DWORD PTR A[rip], eax
	ret

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 之前复制，Volatile 变量 B，并未组织编译器优化的发生，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
// cordering.c	gcc -O2 -S -masm=intel cordering.c

volatile int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	add eax, 1
	mov DWORD PTR A[rip], eax
	mov DWORD PTR B[rip], 0
	ret

同时将 A，B 两个变量都声明为 volatile 变量，再来看看对应的汇编。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。

4.1 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 ()                                  Thread2 ()
{                                           {
      // do something;                              if (flag == true)
      // 实际情况，肯定更加复杂                     {
顺                                                      // assert something happens;
序      flag = true;                                    // 实际情况，在假设something已经
交
换                                                      // 发生的前提下，做接下来的工作
      something = 1;                                    assert (something == 1);
}
                                                        // do other things,depends on sth
                                                        other things;
                                                  }
                                            }
```

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。

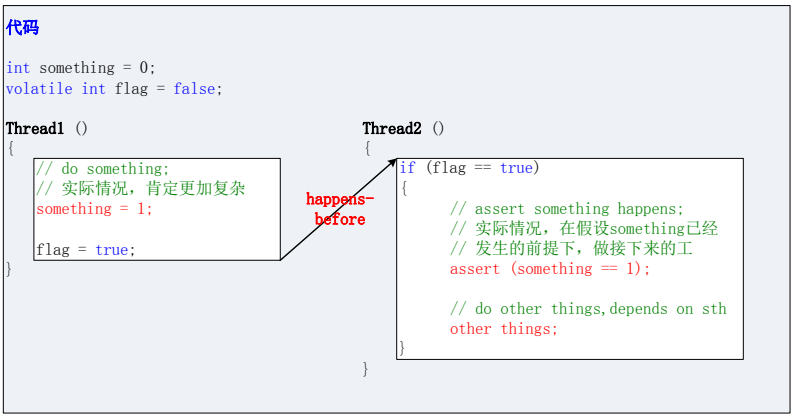
Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
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：



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。

4.2 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。

5 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：

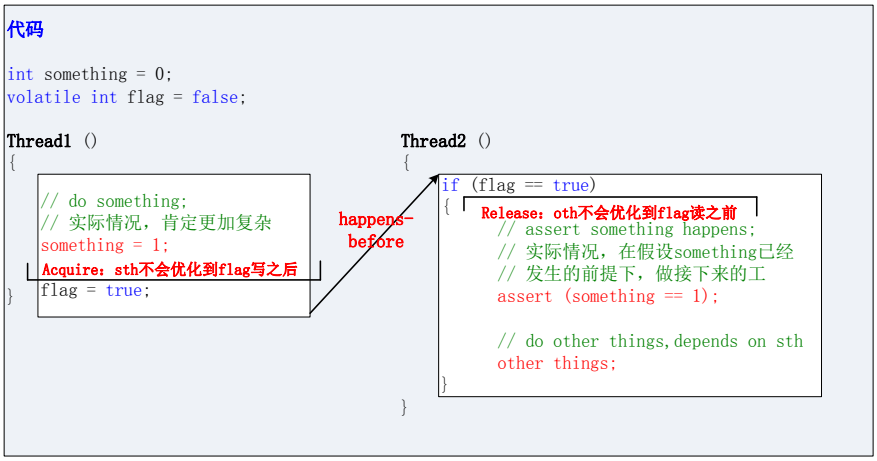
- 对于 Java Volatile 变量的写操作，带有 Release 语义，所有 Volatile 变量写操作之前的针对其他任何变量的读写操作，都不会被编译器、CPU 优化后，乱序到 Volatile 变量的写操作之后执行。

- 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：



6 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-mapeed 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。

7 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++).