[**漫谈C++11多线程内存模型**](http://blog.csdn.net/cszhouwei/article/details/11730559)

标签： [C++](http://www.csdn.net/tag/C%2b%2b)[Memory Reordering](http://www.csdn.net/tag/Memory%20Reordering)

2013-09-16 13:54 12524人阅读 [评论](http://blog.csdn.net/cszhouwei/article/details/11730559#comments)(8) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/cszhouwei/article/details/11730559#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C/C++（8） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/cszhouwei/article/details/11730559)

**写在前面**

*“*[***C++***](http://lib.csdn.net/base/cplusplus)*11 feels like a new language” - Bjarne Stroustrup*

的确，[**c++**](http://lib.csdn.net/base/cplusplus)11核心已经发生了巨大的变化，它现在支持Lambda表达式、对象类型自动推断、统一初始化语法、Deleted和Defaulted函数、nullptr、委托构造函数、右值引用等等，本文主要讨论C++11对于多线程编程的支持。

**一些例子**

为何C++多线程编程需要对标准进行修订，基于多线程库如POSIX、boost.Thread的大量代码不是都工作得很好吗？详见《[Threads Cannot be Implemented as a Library](http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf)》，简单概括如下：

*因为C++03标准是单线程的，所以即便是完全符合标准的编译器也可能各个脑袋里面只装着一个线程，于是在对代码作优化的时候总是一不小心就可能做出危害多线程正确性的优化来。*

**简单示例：**

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. x = y = 0;
2. Thread1     Thread2
3. x = 1;      y = 1;
4. r1 = y;     r2 = x;

理论上来说，r1==r2==0这种输出是不可能的（此处不展开推理，请各位看官自行分析），但现实往往是残酷的，编译器只需把Thread1中的x=1和r1=y操作互换即可。

**极端示例：**

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. **for** (…) {
2. …
3. **if** (mt) pthread\_mutex\_lock(…);
4. x = … x …
5. **if** (mt) pthread\_mutex\_unlock(…);
6. }

对于x的访问已经被pthread\_mutex\_lock/pthread\_mutex\_unlock包围了，这下总算安全了吧？在Hans Boehm的paper中提到，编译器可以运用“Register Promotion”的技术进行优化，对此，POSIX线程库也无能为力：

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. r = x;
2. **for** (…) {
3. …
4. **if** (mt) { x = r; pthread\_mutex\_lock(…); r = x; }
5. r = … r …
6. **if** (mt) { x = r; pthread\_mutex\_unlock(…); r = x; }
7. }
8. x = r;

**Memory Model**

那么，究竟如何才能编写出正确的多线程代码呢？最简单的办法就是禁止编译器作任何优化：所有的操作严格按照“Program Order”执行，所有的操作都触发“Cache Coherence”以确保它们的副作用在跨线程间的“Memory Visibility”顺序。

但这样做显然是不切实际的，需要付出巨大的效率代价。于是编译器说：“不如这样，你来告诉我哪些数据是线程间共享的。这样，我就可以在必要的时候保守优化，一般情况下全力优化。只要你保证自己的程序是正确同步的，那我保证程序执行时就是你要的那个样子。”这样一个在程序员和语言间的约定，就是“Memory Model”。

*PS：上面的例子可能会给大家一个错觉，这一切都是编译器的错，实际上这里的“罪魁祸首”是一种称为“Memory Reordering”的存在，而“Compiler Reordering”仅是其中的一个来源，另一个就是更为底层的“Processor Reordering”，因此才需要前文提到的“Cache Coherence”。本文不打算对硬件做过多的探讨，有兴趣的读者推荐阅读《*[*Memory Barriers: a Hardware View for Software Hackers*](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.5245&rep=rep1&type=pdf)*》，非常精彩！*

**Memory Order**

好了，下面让我们切入正题，看看C++11到底给我们带来了什么？

* [Atomic operations library](http://en.cppreference.com/w/cpp/atomic)
* [Thread support library](http://en.cppreference.com/w/cpp/thread)

“Thread support library”可以简单想象成POSIX线程库的OO版本，对常用的Threads、Mutex、Condition Variables、Futures等概念进行了很好的封装，其实它的前身就是[Boost::Thread](http://www.boost.org/doc/libs/1_54_0/doc/html/thread.html)，本文略过不提。

“Atomic operations library”顾名思义，其实就是原子操作库。而在以往，我们往往需要借助汇编语言或者第三方线程库方能实现。atomic对于多线程编程，尤其是lock-free[**算法**](http://lib.csdn.net/base/datastructure)，其重要性不言而喻，有了std::atomic库，我们终于可以摆脱那些繁琐的汇编代码了！

*PS：乐衷于lock-free编程的读者需要注意的一点，并非所有的atomic内置类型操作均是lock-free的，与具体平台相关，可以调用is\_lock\_free接口进行查询。*

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. **void** store( T desired, memory\_order = std::memory\_order\_seq\_cst );
2. T load( memory\_order = std::memory\_order\_seq\_cst ) **const**;
3. T exchange( T desired, memory\_order = std::memory\_order\_seq\_cst );
4. … (略)

查看std::atomic接口可以发现，几乎每个方法都有一个类型为memory\_order的默认参数，默认值是std::memory\_order\_seq\_cst。

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. **enum** memory\_order {
2. memory\_order\_relaxed,
3. memory\_order\_consume,
4. memory\_order\_acquire,
5. memory\_order\_release,
6. memory\_order\_acq\_rel,
7. memory\_order\_seq\_cst
8. };

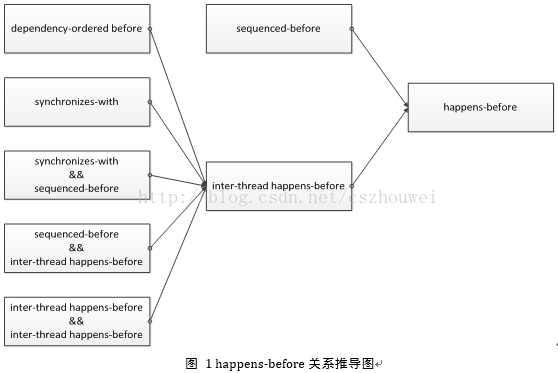
第一次接触memory order的读者看到这里估计已经晕了，不幸的是我们还必须引入更多概念才能讲清楚。首先，我们必须铭记在心的是，c++11引入这些概念本质上是为了解决 “visible side-effects”的问题，用通俗的话来讲：

*线程1执行写操作A之后，如何可靠并高效地保证线程2执行读操作B时，操作A的结果是完整可见的？*

为了解决上述问题，C++11引入了“happens-before”关系，其比较完整的定义如下：

[*Let A and B represent operations performed by a multithreaded process. If A happens-before B, then the memory effects of A effectively become visible to the thread performing B before B is performed.*](http://preshing.com/20130702/the-happens-before-relation)

OK，现在问题就转化为：如何在A、B两个操作之间建立起happens-before关系呢？下面为大家奉上一张新鲜出炉的关系推导图谱，此图信息量巨大，请仔细琢磨回味：



* sequenced-before（线程内）

*在同一个线程内，操作A先于操作B*

* dependency-ordered before（线程间）

case 1：*线程1的操作A对变量M执行“release”写，线程2的操作B对变量M执行“consume”读，并且操作B读取到的值源于操作A之后的“release”写序列中的任何一个（包括操作A本身）*

case 2：*线程1的操作A 与线程2的操作X之间存在dependency-ordered before关系，同时线程2的操作B“depends on”操作X（所谓B“depends on”A，这里就不给出精确的定义，举个直观的例子：B=M[A]）*

* synchronizes-with（线程间）

*线程1的操作A对变量M执行“release”写，线程2的操作B对变量M执行“acquire”读，并且操作B读取到的值源于操作A之后的“release”写序列中的任何一个（包括操作A本身）*

基础知识铺垫到此结束，下面我们总算可以来具体谈谈不同memory order的使用了！

**代码分析**

* Relaxed ordering

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. x = y = 0;
2. // Thread 1:
3. r1 = y.load(memory\_order\_relaxed); // A
4. x.store(r1, memory\_order\_relaxed); // B
5. // Thread 2:
6. r2 = x.load(memory\_order\_relaxed); // C
7. y.store(42, memory\_order\_relaxed); // D

简单来说，标记为memory\_order\_relaxed的atomic操作对于memory order几乎不作保证，它们唯一的承诺就是“atomicity”，当然，不能破坏“modification order”的一致性要求。

对于上述代码片段而言，输出r1 == r2 == 42是合法的。这里，我们可以推导出的关系只有A sequenced-before B、C sequenced-before D，仅此而已。

*TIPS：Relaxed ordering比较适用于“计数器”一类的原子变量，不在意memory order的场景。*

* Release-Acquire ordering

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. #include
2. #include
3. #include
4. #include
6. std::atomic<std::string\*> ptr;
7. **int** data;
9. **void** producer()
10. {
11. std::string\* p = **new** std::string("Hello");          // A
12. data = 42;                          // B
13. ptr.store(p, std::memory\_order\_release);            // C
14. }
16. **void** consumer()
17. {
18. std::string\* p2;
19. **while** (!(p2 = ptr.load(std::memory\_order\_acquire)))         // D
20. ;
21. assert(\*p2 == "Hello");                 // E
22. assert(data == 42);                     // F
23. }
25. **int** main()
26. {
27. std::**thread** t1(producer);
28. std::**thread** t2(consumer);
29. t1.join(); t2.join();
30. }

首先，我们可以直观地得出如下关系：A sequenced-before B sequenced-before C、C synchronizes-with D、D sequenced-before E sequenced-before F。

利用前述happens-before推导图，不难得出A happens-before E、B happens-before F，因此，这里的E、F两处的assert永远不会fail。

*TIPS：Release-Acquire ordering难度系数与性能指数相对均衡，属于实现lock-free算法的首选。*

* Release-Consume ordering

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. **void** producer()
2. {
3. std::string\* p = **new** std::string("Hello");          // A
4. data = 42;                          // B
5. ptr.store(p, std::memory\_order\_release);            // C
6. }
8. **void** consumer()
9. {
10. std::string\* p2;
11. **while** (!(p2 = ptr.load(std::memory\_order\_consume)))     // D
12. ;
13. assert(\*p2 == "Hello");                 // E
14. assert(data == 42);                     // F
15. }

这次我们把D处修改为memory\_order\_consume，情况又会有何不同呢？首先，基本的关系对毋庸置疑：A sequenced-before B sequenced-before C、C dependency-ordered before D、D sequenced-before E sequenced-before F。

那么我们还能那么轻易地推导出A happens-before E、B happens-before F吗？答案是：A、E关系成立，而B、F关系破裂。根据我们之前的定义，E depends-on D，从而可以推导出，接着就是水到渠成了。反观D、F之间并不存在这种依赖关系。因此，这里的E永远不会fail，而F有可能fail。

*TIPS：Release-Consume ordering难度系数最高，强烈不推荐初学者使用，很多大师级人物都在这上面栽过跟头，当然，它的系统开销可能小于Release-Acquire ordering，适用于极致追求性能的场景，前提是你得能够hold住它。*

* Sequentially-consistent ordering

**[cpp]** [view plain](http://blog.csdn.net/cszhouwei/article/details/11730559) [copy](http://blog.csdn.net/cszhouwei/article/details/11730559)

1. std::atomic x = ATOMIC\_VAR\_INIT(**false**);
2. std::atomic y = ATOMIC\_VAR\_INIT(**false**);
3. std::atomic z = ATOMIC\_VAR\_INIT(0);
5. **void** write\_x()
6. {
7. x.store(**true**, std::memory\_order\_seq\_cst);   // A
8. }
10. **void** write\_y()
11. {
12. y.store(**true**, std::memory\_order\_seq\_cst);   // B
13. }
15. **void** read\_x\_then\_y()
16. {
17. **while** (!x.load(std::memory\_order\_seq\_cst))
18. ;
19. **if** (y.load(std::memory\_order\_seq\_cst)) {
20. ++z;
21. }
22. }
24. **void** read\_y\_then\_x()
25. {
26. **while** (!y.load(std::memory\_order\_seq\_cst))
27. ;
28. **if** (x.load(std::memory\_order\_seq\_cst)) {
29. ++z;
30. }
31. }
33. **int** main()
34. {
35. std::**thread** a(write\_x);
36. std::**thread** b(write\_y);
37. std::**thread** c(read\_x\_then\_y);
38. std::**thread** d(read\_y\_then\_x);
39. a.join(); b.join(); c.join(); d.join();
40. assert(z.load() != 0);          // C
41. }

所谓的Sequentially-consistent ordering，其实就是“顺序一致性”，它是最严格的memory order，除了满足前面所说的Release-Acquire/Consume约束之外，所有的线程对于该顺序必须达成一致。

这里，C处的assert是永远不会faile的。反证法：在线程c的世界里，如果++z未执行，需要操作A先于操作B完成；在线程d的世界里，如果++z未执行，需要操作B先于操作A完成。由于这些操作都是memory\_order\_seq\_cst类型，因此，所有的线程需要达成一致，出现矛盾。

*TIPS：Sequentially-consistent ordering难度系数最低，潜在开销可能最大，最符合人类常规思维模型，因此，在多线程编程中最易推理，最不容易出错，强烈推荐初学者使用，当出现性能瓶颈时再考虑优化。*

**聊聊volatile**

很多读者可能比较奇怪，为啥突然会聊到volatile的话题？其实volatile在C/C++阵营里的争论从来没有停止过，而我们公司的代码库里，同样可以发现volatile大量地用于多线程/多进程编程，这真的可行吗？

这里我不打算展开进行讨论，仅列出一些个人理解：

* 禁止编译器优化（禁用寄存器优化，直接读写内存）
* 无法保证atomicity（机器字长内的变量可以保证？注意内存对齐！）
* 无法保证memory order（对于Processor Reordering无能为力）

为何我们的代码仍然work呢？这和具体硬件平台相关，x86平台属于strongly-ordered模型，绝大多数场景下，Release-Acquire ordering是可以自动获得的。至于atomicity的问题，一定要留意机器字长以及内存对齐。

推荐使用场景：信号处理函数中使用到的信号标志变量。关于volatile更多精彩讨论推荐阅读《[Nine ways to break your systems code using volatile](http://blog.regehr.org/archives/28)》。

*PS：这里的讨论仅限于C/C++，不同语言对于volatile赋予的语义并不相同，比如*[***Java***](http://lib.csdn.net/base/java)*中的volatile是保证顺序一致性的。*

**写在最后**

本文主要致力于梳理清楚C++11的多线程Memory Order概念，帮助大家更好地理解多线程lock-free代码应该如何编写，如何分析其正确性。其实更高级的同步工具，如mutex、spinlock等，同样可以提供Release-Acquire ordering，只是并非本文关注的重点，故略过不提。而且限于篇幅所限，对于硬件层面的Processor Reordering几乎未作解释，后面有机会再写个硬件篇和大家一起探讨吧！

PS*：个人水平有限，理解偏差在所难免，欢迎讨论交流！*