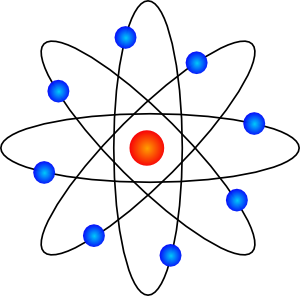
C++ 11 开发中的 Atomic 原子操作

<http://blog.jobbole.com/106516/>

原子操作在多线程开发中经常用到，比如在计数器，序列产生器等地方，这类情况下数据有并发的危险，但是用锁去保护又显得有些浪费，所以原子类型操作十分的方便。

原子操作虽然用起来简单，但是其背景远比我们想象的要复杂。其主要在于现代计算系统过于的复杂：多处理器、多核处理器、处理器又有核心独有以及核心共享的多级缓存，在这种情况下，一个核心修改了某个变量，其他核心什么时候可见是一个十分严肃的问题。

同时在极致最求性能的时代，处理器和编译器往往表现的很智能，进行极度的优化，比如什么乱序执行、指令重排等，虽然可以在当前上下文中做到很好的优化，但是放在多核环境下常常会引出新的问题来，这时候就必须提示编译器和处理器某种提示，告诉某些代码的执行顺序不能被优化。

所以这里说到的原子操作，基本都包含我们三个方面所关心的语义：操作本身是不可分割的(Atomicity)，一个线程对某个数据的操作何时对另外一个线程可见(Visibility)，执行的顺序是否可以被重排(Ordering)。  
[](http://jbcdn2.b0.upaiyun.com/2016/10/79ff21cd2d0b4750fefe2b0ab767f28f.png)

# 一、legacy GCC \_\_sync

据说在C++11标准出来之前，大家都诟病C++标准没有一个明确的内存模型，随着多线程开发的普及这个问题显得越来越迫切。当然各个C++编译器实现者也是各自为政，GCC自然是实用主义当道，于是根据Intel的开发手册老早就搞出了一系列的\_\_sync原子操作函数集合，这也是被广大程序员最为熟悉常用的操作了吧，罗列如下：

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | type \_\_sync\_fetch\_and\_OP (type \*ptr, type value, ...)  type \_\_sync\_OP\_and\_fetch (type \*ptr, type value, ...)  bool \_\_sync\_bool\_compare\_and\_swap (type \*ptr, type oldval, type newval, ...)  type \_\_sync\_val\_compare\_and\_swap (type \*ptr, type oldval, type newval, ...)  \_\_sync\_synchronize (...)    type \_\_sync\_lock\_test\_and\_set (type \*ptr, type value, ...)  void \_\_sync\_lock\_release (type \*ptr, ...) |

上面的OP操作包括add、sub、or、and、xor、nand这些常见的数学操作，而type表示的数据类型Intel官方允许的是int、long、long long的带符号和无符号类型，但是GCC扩展后允许任意1/2/4/8的标量类型；CAS的操作有两个版本分别返回bool表示是否成功，而另外一个在操作之前会先返回ptr地址处存储的值；\_\_sync\_synchronize直接插入一个full memory barrier，当然你也可能经常见到像asm volatile(“” ::: “memory”);这样的操作。前面的这些原子操作都是full barrier类型的，这意味着：任何内存操作的指令不允许跨越这些操作重新排序。

\_\_sync\_lock\_test\_and\_set用于将value的值写入ptr的位置，同时返回ptr之前存储的值，其内存模型是acquire barrier，意味着该操作之后的memory store指令不允许重排到该操作之前去，不过该操作之前的memory store可以排到该操作之后去，而\_\_sync\_lock\_release则更像是对前面一个操作锁的释放，通常意味着将0写入ptr的位置，该操作是release barrier，意味着之前的memory store是全局可见的，所有的memory load也都完成了，但是接下来的内存读取可能会被排序到该操作之前执行。可以这里比较绕，翻译起来也比较的拗口，不过据我所见，这里很多是用在自旋锁类似的操作上，比如：

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static volatile int \_sync;    static void lock\_sync() {  while(\_\_sync\_lock\_test\_and\_set(&\_sync, 1));  }  static void unlock\_sync() {  \_\_sync\_lock\_release(&\_sync);  } |

其实这里的1可以是任何non-zero的值，主要是用作bool的效果。

# 二、C++11 新标准中的内存模型

上面GCC那种full barrier的操作确实有效，但是就像当初系统内核从单核切换到多核用大颗粒锁一样的简单粗暴，先不说这种形势下编译器和处理器无法进行优化，光要变量使其对他处理器可见，就需要在处理间进行硬件级别的同步，显然是十分耗费资源的。在C++11新标准中规定的内存模型(memory model)颗粒要细化的多，如果熟悉这些内存模型，在保证业务正确的同时可以将对性能的影响减弱到最低。

原子变量的通用接口使用store()和load()方式进行存取，可以额外接受一个额外的memory order参数，而不传递的话默认是最强模式Sequentially Consistent。

根据执行线程之间对变量的同步需求强度，新标准下的内存模型可以分成如下几类：

## 2.1 Sequentially Consistent

该模型是最强的同步模式，参数表示为std::memory\_order\_seq\_cst，同时也是默认的模型。

C++

|  |  |
| --- | --- |
| 1  2  3 | -Thread 1- -Thread 2-  y = 1 if (x.load() == 2)  x.store (2); assert (y == 1) |

对于上面的例子，即使x和y是不相关的，通常情况下处理器或者编译器可能会对其访问进行重排，但是在seq\_cst模式下，x.store(2)之前的所有memory accesses都会happens-before在这次store操作。

另外一个角度来说：对于seq\_cst模式下的操作，所有memory accesses操作的重排不允许跨域这个操作，同时这个限制是双向的。

## 2.2 Acquire/Release

GCC的wiki可能讲的不太清楚，查看下面的典型Acquire/Release的使用例子：

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | std::atomic<int> a{0};  int b = 0;  -Thread 1-  b = 1;  a.store(1, memory\_order\_release);  -Thread 2-  while (a.load(memory\_order\_acquire) != 1)  /\*waiting\*/;  std::cout << b << '\n'; |

毫无疑问，如果是seq\_cst，那么上面的操作一定是成功的(打印变量b显示为1)。

a. memory\_order\_release保证在这个操作之前的memory accesses不会重排到这个操作之后去，但是这个操作之后的memory accesses可能会重排到这个操作之前去。通常这个主要是用于之前准备某些资源后，通过store+memory\_order\_release的方式”Release”给别的线程；

b. memory\_order\_acquire保证在这个操作之后的memory accesses不会重排到这个操作之前去，但是这个操作之前的memory accesses可能会重排到这个操作之后去。通常通过load+memory\_order\_acquire判断或者等待某个资源，一旦满足某个条件后就可以安全的“Acquire”消费这些资源了。

## 2.3 Consume

这是一个相比Acquire/Release更加宽松的内存模型，对非依赖的变量也去除了happens-before的限制，减少了所需同步的数据量，可以加快执行的速度。

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | -Thread 1-  n = 1  m = 1  p.store (&n, memory\_order\_release)  -Thread 2-  t = p.load (memory\_order\_acquire);  assert( \*t == 1 && m == 1 );  -Thread 3-  t = p.load (memory\_order\_consume);  assert( \*t == 1 && m == 1 ); |

线程2的assert会pass，而线程3的assert可能会fail，因为n出现在了store表达式中，算是一个依赖变量，会确保对该变量的memory access会happends-before在这个store之前，但是m没有依赖关系，所以不会同步该变量，对其值不作保证。

Comsume模式因为降低了需要在硬件之间同步的数量，所以理论上其执行的速度会比之上面的内存模型块一些，尤其在共享内存大规模数据量情况下，应该会有较明显的差异表现出来。

在这里，Acquire/Consume~Release这种线程间同步协作的机制就被完全暴露了，通常会形成Acquired/Consume来等待Release的某个状态更新。需要注意的是这样的通信需要两个线程间成对的使用才有意义，同时对于没有使用这个内存模型的第三方线程没有任何作用效果。

## 2.4 Relaxed

最宽松的模式，memory\_order\_relaxed没有happens-before的约束，编译器和处理器可以对memory access做任何的re-order，因此另外的线程不能对其做任何的假设，这种模式下能做的唯一保证，就是一旦线程读到了变量var的最新值，那么这个线程将再也见不到var修改之前的值了。

这种情况通常是在需要原子变量，但是不在线程间同步共享数据的时候会用，同时当relaxed存一个数据的时候，另外的线程将需要一个时间才能relaxed读到该值，在非缓存一致性的构架上需要刷新缓存。在开发的时候，如果你的上下文没有共享的变量需要在线程间同步，选用Relaxed就可以了。

## 2.5 小结

看到这里，你对Atomic原子操作，应当不仅仅停留在indivisable的层次了，因为所有的内存模型都能保证对变量的修改是原子的，C++11新标准的原子应该上升到了线程间数据同步和协作的问题了，跟前面的LockFree关系也比较密切。

手册上也这样告诫菜鸟程序员：除非你知道这是什么，需要减弱线程间原子上下文同步的耦合性增加执行效率，才考虑这里的内存模型来优化你的程序，否则还是老老实实的使用默认的memory\_order\_seq\_cst，虽然速度可能会慢点，但是稳妥些，万一由于你不成熟的优化带来问题，是很难去调试的。

# 三、C++11 GCC \_\_atomic

GCC实现了C++11之后，上面的\_\_sync系列操作就变成了Legacy而不被推荐使用了，而基于C++11的新原子操作接口使用\_\_atomic作为前缀。

对于普通的数学操作函数，其函数接口形式为：

C++

|  |  |
| --- | --- |
| 1  2 | type \_\_atomic\_OP\_fetch (type \*ptr, type val, int memorder);  type \_\_atomic\_fetch\_OP (type \*ptr, type val, int memorder); |

除此之外，还根据新标准提供了一些新的接口：

C++

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | type \_\_atomic\_load\_n (type \*ptr, int memorder);  void \_\_atomic\_store\_n (type \*ptr, type val, int memorder);  type \_\_atomic\_exchange\_n (type \*ptr, type val, int memorder);  bool \_\_atomic\_compare\_exchange\_n (type \*ptr, type \*expected, type desired, bool weak, int success\_memorder, int failure\_memorder);    bool \_\_atomic\_test\_and\_set (void \*ptr, int memorder);  void \_\_atomic\_clear (bool \*ptr, int memorder);    void \_\_atomic\_thread\_fence (int memorder);    bool \_\_atomic\_always\_lock\_free (size\_t size, void \*ptr);  bool \_\_atomic\_is\_lock\_free (size\_t size, void \*ptr); |

从函数名，看起来意思也很明了吧，上面的带\_n的后缀版本如果去掉\_n就是不用提供memorder的seq\_cst版本。最后的两个函数，是判断系统上对于某个长度的对象是否会产生lock-free的原子操作，一般long long这种8个字节是没有问题的，对于支持128位整形的构架就可以达到16字节无锁结构了。

Boost.Asio这里就不在罗列了，不过其中有一些例子比较好，基于内存模型的Wait-free的ring buffer、producer-customer的例子，可以去看看。

# 参考文献

[Chapter 45. Boost.Atomic](http://theboostcpplibraries.com/boost.atomic)  
[Chapter 5. Boost.Atomic](http://www.boost.org/doc/libs/1_61_0/doc/html/atomic.html)  
[6.52 Built-in Functions for Memory Model Aware Atomic Operations](https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html)  
[6.51 Legacy \_\_sync Built-in Functions for Atomic Memory Access](https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html)  
[Concurrent programming the fast and dirty way!](https://fotisl.com/blog/2009/11/concurrent-programming-the-fast-and-dirty-way/)  
[n3337.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf)  
[GCC wiki on atomic synchronization](https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync)