多线程（一）：C++11 atomic和内存序

多线程很难学，就像大横按对于所有吉他初学者一样，是一道划分能力的门槛。培养成熟的多线程编程思想，能梳理清楚多线程的逻辑，这就需要下无数年的苦功。对于多线程的各种api，涉及到源码原理的不多，各路论坛能查到的资料也语焉不详，让我学得极其痛苦，所以干脆自己写一个多线程系列，方便自己复习，也能借大家参考一下。

和atomic的初次相见并没有一见倾心的浪漫

初次学习atomic的时候，一头雾水，我不知道这个东西有怎样的意义，也不懂底层的实现原理，而且和其他概念混杂在一起后产生了更多的疑惑：

- 原子变量和原子操作的关系是什么，底层实现？
- atomic和能解决多线程的什么问题？
- atomic和锁的联系和区别？
- atomic和volatile（c/c++）的联系和区别？
- 为什么atomic的成员方法有内存序参数？
- 不同的内存序有什么区别？
- 内存序和内存屏障是一个东西吗？
- 什么是无锁编程，无锁编程就是用atomic来代替锁吗？

如果你也有这些疑惑，让我们来一个个来解决这些问题。

什么是原子变量

atomic是一个类模板，在我的认知中，更愿意把它看做是一个拓展封装类，封装一个原有的类型，并拓展新的api给用户，好比share_ptr之于原生指针，适配器queue之于容器deque。

atomic也是如此，当我们写下std::atomic<int>的时候，意味着将int拓展成原子类型，将int类型的++，--等都变成原子操作，同时拓展了诸如fetch_add，fetch_sub等原子加减方法供用户使用。

特化成员函数

fetch_add	原子地将参数加到存储于原子对象的值，并返回先前保有的值 (公开成员函数)
fetch_sub	原子地从存储于原子对象的值减去参数，并获得先前保有的值 (公开成员函数)
fetch_and	原子地进行参数和原子对象的值的逐位与，并获得先前保有的值 (公开成员函数)
fetch_or	原子地进行参数和原子对象的值的逐位或，并获得先前保有的值 (公开成员函数)
fetch_xor	原子地进行参数和原子对象的值的逐位异或，并获得先前保有的值 (公开成员函数)
operator++ **operator++**(int) **operator--** **operator--**(int)	令原子值增加或减少一 (公开成员函数)
operator+= **operator-=** **operator&=** **operator|=** **operator^=**	加、减，或与原子值进行逐位与、或、异或 (公开成员函数)

然而atomic变量并不是所有成员方法都是原子操作。像赋值运算符=操作并不提供原子性。

成员函数

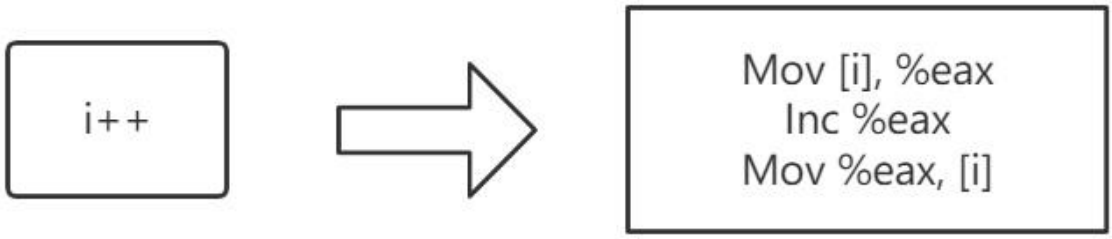
(构造函数)	构造原子对象 (公开成员函数)
operator=	存储值于原子对象 (公开成员函数)
is_lock_free	检查原子对象是否免锁 (公开成员函数)
store	原子地以非原子对象替换原子对象的值 (公开成员函数)
load	原子地获得原子对象的值 (公开成员函数)
operator T	从原子对象加载值 (公开成员函数)
exchange	原子地替换原子对象的值并获得它先前持有的值 (公开成员函数)
compare_exchange_weak compare_exchange_strong	原子地比较原子对象与非原子参数的值，若相等则进行交换，若不相等则进行加载 (公开成员函数)
wait (C++20)	阻塞线程直至被提醒且原子值更改 (公开成员函数)
notify_one (C++20)	提醒至少一个在原子对象上的等待中阻塞的线程 (公开成员函数)
notify_all (C++20)	提醒所有在原子对象上的等待中阻塞的线程 (公开成员函数)

所以我们可以先对原子变量下一个初步的定义：**即某些成员方法是原子操作的对象。**

什么是原子操作

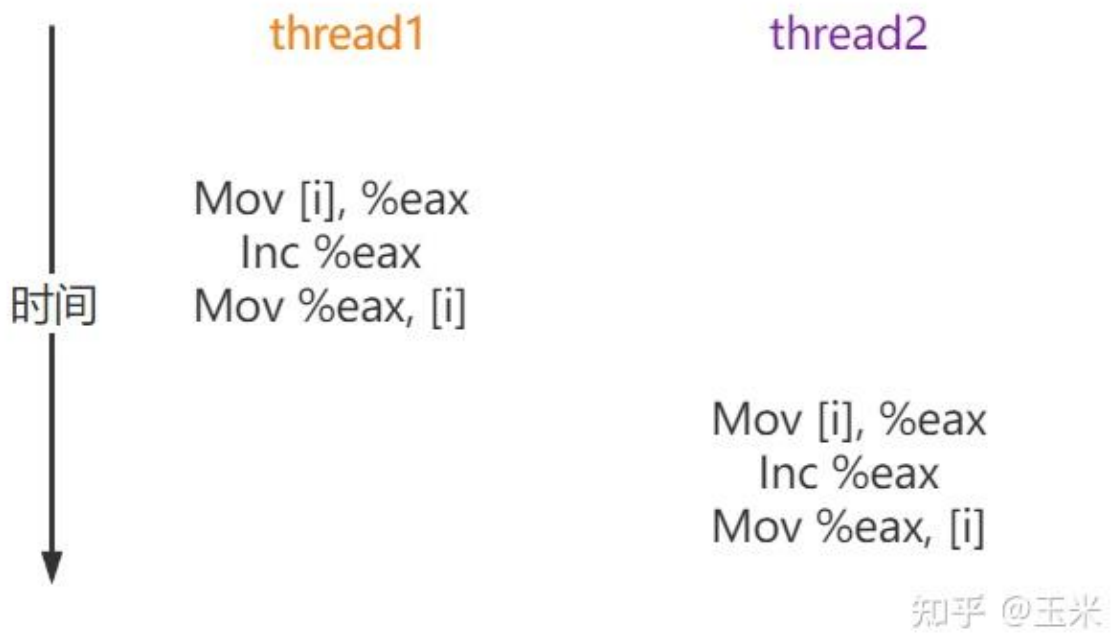
什么是原子操作呢，回想起第一次字节面试，面试官问我 `++i` 和 `i++` 哪个操作是原子的，我觉得 `++i` 不用像 `i++` 一样，在自增的同时还要拷贝一个自增前的副本作为返回值，所以应该是原子的，当时面试官笑而不语，后来我才发现，不论哪个都不是原子操作。（真心腹黑.....）

原子操作和高层（编程语言层级在C以上，姑且叫它高层）代码的实现并没有关系，哪怕 `++i` 或是 `i++` 在高层只是一条单独的语句，当翻译成底层代码（汇编）的时候，就需要更多指令来完成。

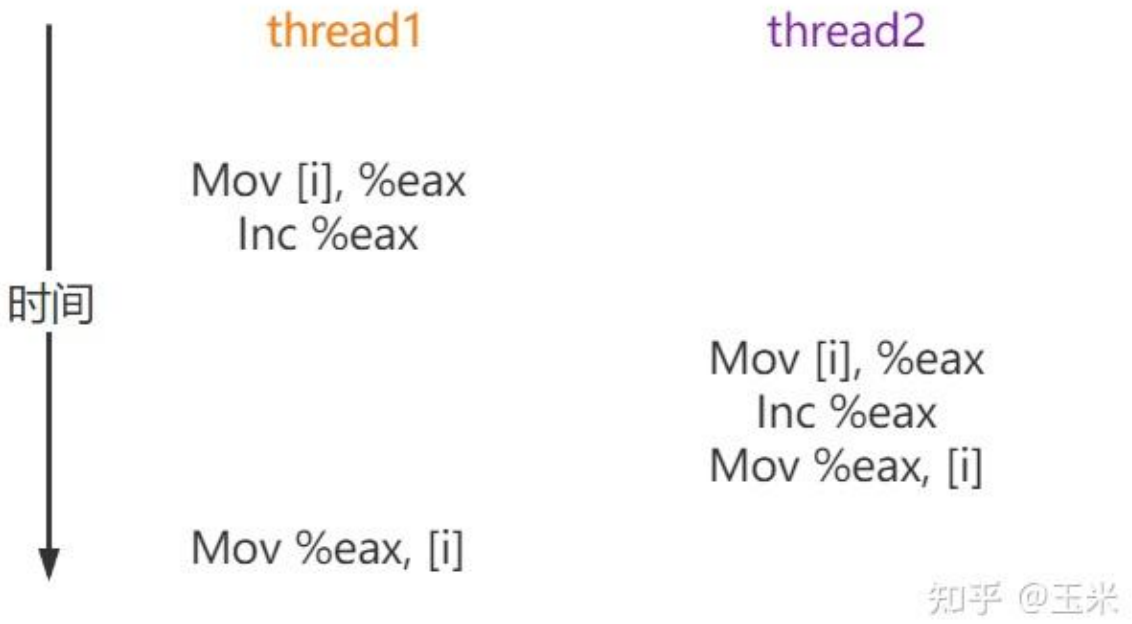


我们需要使用mov指令，从内存中，将i的值读到寄存器eax中。使用inc指令自增寄存器eax的值。然后使用mov指令，将寄存器eax的值拷贝回内存。

想象这样一个场景，两个线程同时共享变量 `i = 0`,并都执行 `i++` 操作，我们当然希望两个线程都按照以下的顺序来执行，最后得到 `i` 的结果为2。



然而因为在汇编层面代码 `i ++` 并不是单一指令，很可能出现以下情况：



当线程1自增完寄存器中的 i 后，并没有写回到内存中的 i ，线程2抢占了cpu，从内存中依然读的是初始内存的 i=0 到寄存器中，自增，并写回到内存，此时内存 i= 1。线程1恢复执行，再次给内存的 i 写入1，所以最后的结果依旧还是 i = 1而不是 i=2。

这会给程序员带来强烈的不安，写代码写得似乎都没有安全感了，当你在某个线程中对一个共享的变量进行更新，哪怕只是自增，你都无法保证这个自增的动作是可靠的，因为cpu看到的代码和你看到的代码不同，由于多线程导致这个更新没有发生实在太常见了。

原子操作正是为了解决这个问题，在任何一个线程中，只要你对原子变量进行++操作，一定能保证在当前线程的这个++动作一定会完成，不会受其它线程的影响。

所以我们可以给原子操作下个定义：**即在高层代码的一个原子操作，不论在底层是怎么实现的，有多少条指令，在底层这些指令执行期间，都不会受到其它线程或者读写任务的干扰。**所以当我们在两个线程对原子变量 i 分别进行自增，最后的结果一定是2。

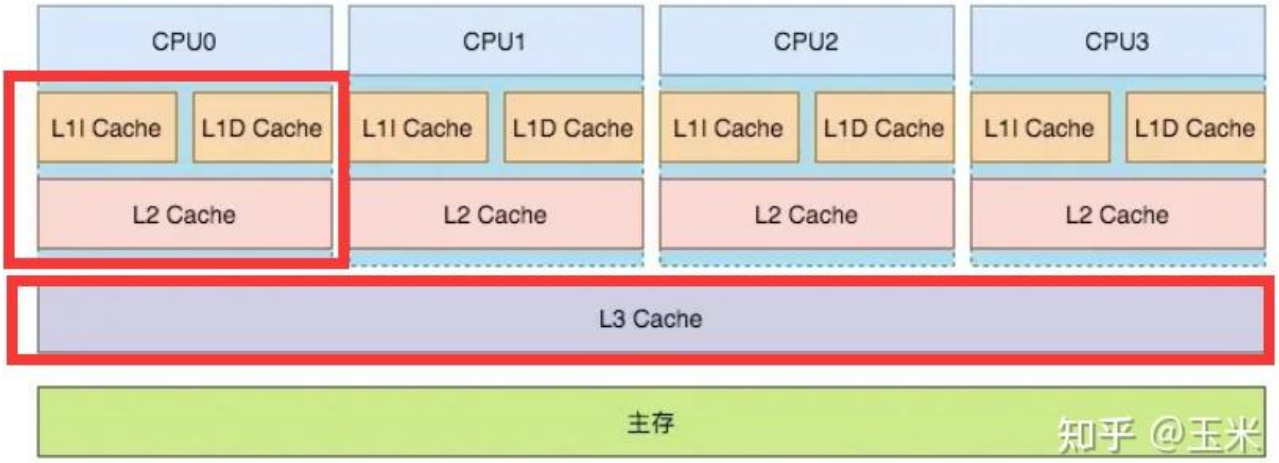
原子操作底层实现

我自己对原子操作的定义是，其底层的诸多指令会被捆绑在一起完成，不受其它线程影响，那么具体该怎么才能实现这种捆绑呢。

加锁，这是大家最容易想到的方法，也是最简单的方法。先不说底层汇编，在高层语言解决实际的业务问题，我们需要某段代码在同一个时间只允许一个进程/线程访问，也就是临界区，一般会lock住mutex来进入临界区，完成业务后unlock mutex来退出临界区。

但是互斥锁mutex有个严重的问题，就是效率不高，如果有两个线程t1和t2，当t1拿到了互斥锁，t2也想要这个锁，发现这个锁已经被t1占有了，t2不会在t1身边徘徊等锁，而是直接陷入阻塞态，直到t1释放了锁，t2才会被唤醒，进入到就绪队列等待调度，并再次去尝试获得锁。这将会消耗大量的资源在重复的“睡眠”，“等待”，“唤醒”的操作中。

其次，在保存和恢复上下文的过程中，还会存在cache失效的可能。



根据局部性原理，cpu设置了三级缓存，这些小而快的单元用于存储主存中经常访问的一些数据，当上下文切换的时候，旧的缓存自然对于新的任务无用，需要重新映射，所以当用了太多互斥锁，会导致线程频繁切换（陷入阻塞态，等待唤醒），继而引起cache无限失效。

那可不可以不用互斥锁实现指令捆绑呢?

可以，互斥锁是在高层语言级别实现的捆绑，事实上硬件本身就提供了指令实现底层语言的捆绑，比如Intel X86系列的cpu提供的“lock”指令前缀。

如果你的代码最后会运行在x86的cpu上，我们甚至可以不用atomic变量的fetch_add，而通过gcc内嵌汇编代码实现一个自己的fetch_add。

fetch_add功能是将increment增量加到指针ptr指向的变量，并返回ptr指向变量的旧值，代码如下所示:

```
static int my_fetch_add(int *ptr, int increment){
    int old_value = *ptr;
    __asm__ volatile("lock; xadd %0, %1 \n\t"
                     : "=r"(old_value), "=m"(*ptr)
                     : "0"(increment), "m"(*ptr)
                     : "cc", "memory");
    return old_value;
}
```

我们可以按这个格式来逐行分析代码。

注意汇编代码部分的前缀是lock；也是整个指令的原子精髓所在，前缀lock用于锁定前端串行总线FSB，保证了指令执行时不会受到其他处理器的干扰。

- 汇编指令为xadd，即交换相加，%0和%1为占位符。
- 输出部分为
  - “=r”：=表示old_value是一个输出操作数，r代表将old_value存放在任意一个寄存器
  - “=m”：=表示*ptr是一个输出操作数，m代表操作数在内存中
- 输入部分为
  - “0”：表示将占位符%0指向操作数increment
  - “m”：将ptr放在任意一个寄存器。
- 会被修改的部分
  - “cc”：通知gcc，嵌入式汇编更新了condition code register
  - “memory”：代表该指令执行完会修改内存的值

要注意一点，占位符的%0本应该是输出部分的第一个操作数（也就是old_value），但是因为输入部分的“0”约束将%0指向了increment。

总地来看，整条指令等价于 “xaddr increment *ptr”。交换相加会将increment和*ptr交换，并把相交的值加到交换后第一个操作数上面，也就是*ptr。

之后将新的*ptr写入到内存，并返回*ptr的原值old_value。

我们再来自己实现一个Compare&Swap（CAS）原子操作,CAS的原语逻辑是，将*ptr和*ptr的试探值cmp做比较，如果相等，就让*ptr=new_val，并返回*ptr的旧值old_value，代码如下。

```
static int cas (int *ptr, int cmp, int new_val){
    int old_value = *ptr;
    __asm__ volatile ("lock; cmpxchg %2, %3"
                      : "=a" (old_value), "=m" (*ptr)
                      : "r" (new_val), "m" (*ptr), "0" (cmp)
                      : "cc", "memory");
    return old_value;
}
```

直接看汇编十分晦涩，是因为cmpxchg这条指令比较奇怪，有个隐藏的%0参数，假设有三个寄存器eax，ebx，ecx对应cmpxchg的%1，%2，%3参数。

cmpxchg %ebx, %ecx的意思是：如果eax(%0)与ecx(%3)相等，则将ebx（%2）的值送入到ecx(%3)。

其中%0指的是试探值cmp，%2指的是new_val，%3指向的是*ptr。

看不懂也不要紧，总之最后这条汇编实现CAS的功能。我也写了一个demo测试了下，发现a自增到了100。

```
int main(){
    int a = 0;
    for(int i =0;i<100;++i){
        cas(&a,a,a+1);
    }
    cout<<a<<endl;
    return 0;
}
```

是的，我利用CAS函数实现了自增++，又利用循环实现了+任意数，而且每个++操作都是原子的，在多线程下也依然能保证该操作可靠，不受其它线程影响地自增完。

所以，理论上任何原子操作++ -- fetch add fetch sub都可以通过CAS变化而成。

可以说，搞懂了CAS对于理解原子操作有极大的帮助，这就是我引出CAS的原因。

哪怕我没看过atomic的源码，也可以在x86机器上利用这样一个原子CAS自己实现一个atomic类。

举一反三，对于x86外的其它机器，自然也会有其它的底层指令有前缀lock类似的功能去实现底层的捆绑，虽然未必是CAS实现，可能是LL/SC（load link/store conditional）之类的特殊指令，但是我们只要先理解其中一种底层实现即可。

CAS可能会导致ABA问题，LL/SC则不会，不过这不是本章的重点，对于这些区别容后再叙。

那难道就没有机器不支持原子指令的?

答案是有，或者有些你自定义的特殊变量套用std::atomic，那么c++11的atomic变量的原子操作就会使用高层语言的mutex来实现，狡猾的地方就在于标准库将这些都封装好了，你不需要清楚地知道内部是怎么实现。不过atomic还是留了一个接口，告诉你底层到底有没有用锁来实现。

注意，使用高层语言的互斥锁而非底层lock指令实现的atomic操作，效率自然不高。

内存序的意义

是否感觉在atomic配合原子操作已经能保证操作的可靠性，相当完美了，为什么还需要有内存序这样的东西?

有人说，是因为单核多线程和多核多线程的原因。

老版x86平台上的lock指令会锁住系统总线，那么不论是哪个核的线程都没法干扰到另一个线程的原子操作，但是随着核心逐渐增多，如果还像以前一样总是lock总线效率就低了，所以在p6以后，即便声明的是lock信号，只会lock到cache级别，而不是总线。这样的话，一个线程t1执行 i++其值只同步到缓存而非内存，若是另一个线程t2执行 i++ 时从内存拿，就会让这个原子操作变得不可靠了。所以内存序是用来解决多核线程问题的。

是这样吗?

当然不是。目前的已经有MESI机制来解决缓存一致性的问题了，当多个核的cache共享一个变量 i 的时候，其中一个核在cache对变量 i 进行修改后更新到内存，会通知其它核将变量 i 标记为失效，下次访问必须重新从内存拿。所以lock缓存配合MESI依旧可以保证原子操作的可靠性。

内存序解决的问题在于，你的高层代码不可能只有一条语句 i ++。

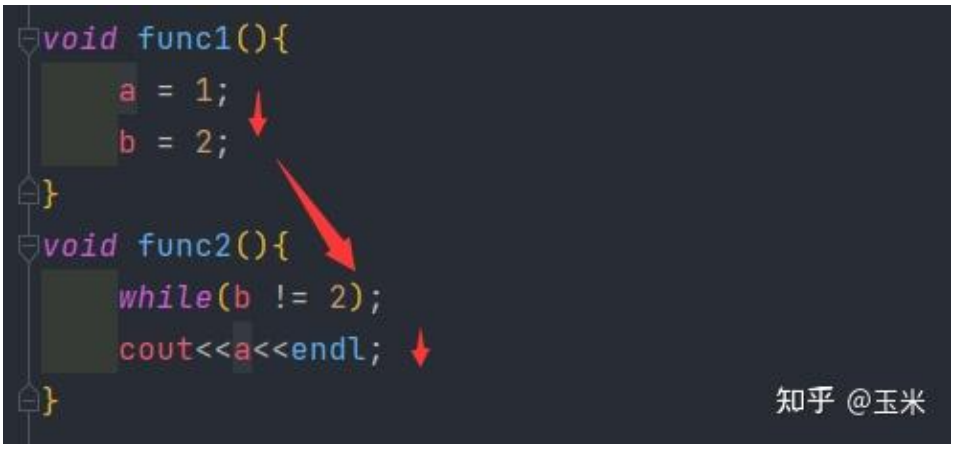
atomic操作只解决了高层语言某个变量某个动作的原子性。

那么多个变量多个语句呢? 看下述代码

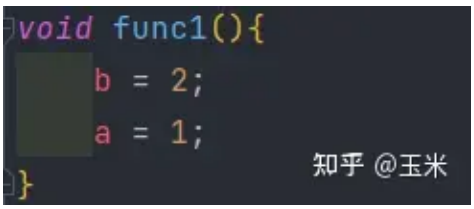
```
#include<iostream>
#include<thread>
using namespace std;

int a = 0;
int b = 0;
void func1(){
    a = 1;
    b = 2;
}
void func2(){
    while(b ≠ 2);
    cout<<a<<endl;
}

int main(){
    std::thread t1(func1);
    std::thread t2(func2);
    t1.join();
    t2.join();
    return 0;
}
```

cout<<a 的结果是什么呢，答案是a既有可能是0，也有可能是1。

我们希望代码以这样的逻辑顺序运行，事实上却并非如此



在cpu的角度，单线程fun1运行的时候，a，b之间并没有依赖关系，那么可能出于某些底层实际运行效率的考量，可能是指令执行级别的乱序优化，流水线、乱序执行、分支预测等等，总之cpu会合理重排一些代码。



实际上线程1可能会先赋值b，再赋值给a，或者按你规定的顺序先赋值a，再赋值b，都有可能。

然而a，b在线程1中的赋值顺序没有依赖关系，在线程2中却并非如此，线程2希望当b=2的时候，a此时必然已经赋值为1。

这就是为什么需要内存序的根本原因，我们利用内存序可以限制cpu对指令执行顺序的重排程度，防止单线程指令的合理重排在多线程的环境下出现顺序上的错误。

所以到此，我们可以对atomic和volatile的做一个区分。

volatile变量的意义在于每次读写都会从内存读或者写内存，解决的是编译器重排的问题。

volatile只能保证涉及每个volatile变量的代码的相对顺序不会被编译器重排，至于volatile变量的代码和其他非volatile变量的代码之间的相对顺序并不保证，且无法保证cpu不会继续重排你的代码。

（注意这里的volatile是c/c++的volatile，java的volatile是有优化的）

而atomic原子操作，我们先狭义上认为此时atomic只是实现了汇编指令的捆绑，也就是我刚刚写的内嵌汇编代码。既然捆绑了，自然也不会被编译器重排。此时我们可以粗浅认为volatile和atomic此时的功能有些类似。

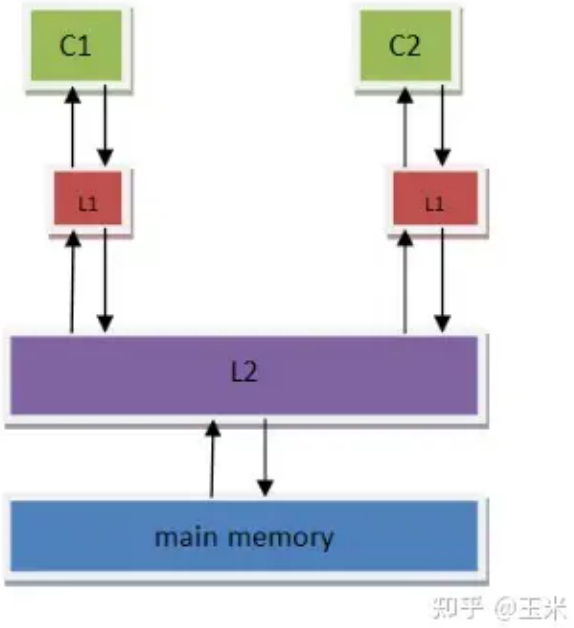
但是c++11的真正atomic操作，是有内存序参数的，用于避免cpu的重排，**可以说atomic+内存序两者叠加才真正在lock-free（免锁）情况下实现了高层代码顺序和底层代码执行顺序的统一。**

内存序的区分

要理解不同的内存序，不妨从几种硬件层面的内存模型来入手，会更好理解。

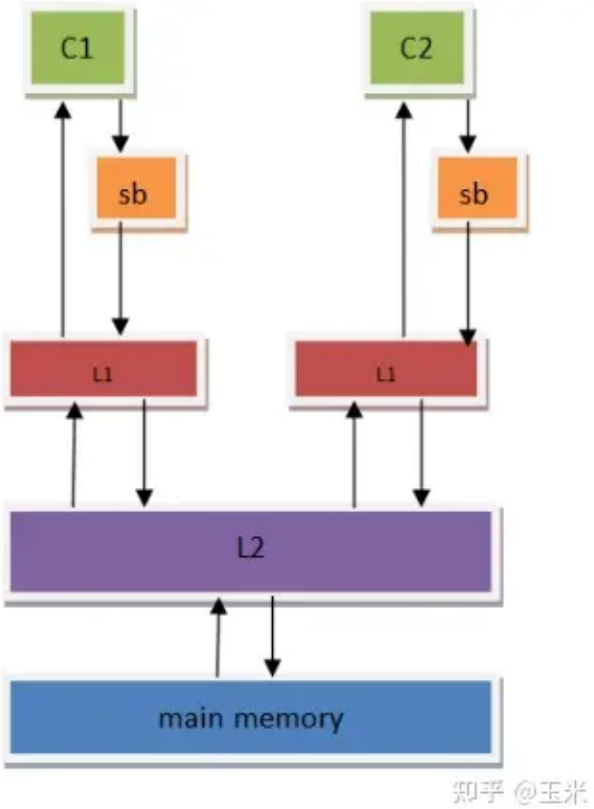
内存一致性模型（memory consistency model）用于描述多线程对共享存储器的访问行为，在不同的内存一致性模型里，多线程对共享存储器的访问行为有非常大的差别。

- 顺序存储模型（强定序模型）



在顺序存储器模型里，cpu不会重排代码，且多核cache通过MESI协议进行同步，MP（多核）会严格按照代码指令流来执行代码。

- 完全存储定序（TSO）



x86 CPU 就是这种内存模型，为了提高效率，其在L1缓存之前加了一个store buffer，因此写数据指令执行时，会先把更新后的数据放到store buffer里后立刻返回执行下一条指令，store buffer的数据则会慢慢被写到L1 cache中，如果有多条写操作指令，会严格按照FIFO的次序执行。

但无论是否FIFO，总之store buffer的存在已经导致MESI的同步被破坏，写指令立刻返回，后续的指令（比如读操作）可能在store buffer数据还没更新到所有cache和内存之前就执行，这就会出现store-load乱序。

- 部分存储定序（PSO）

比如ARM的处理器，为了继续提高效率，在TSO的基础上，放弃写操作的FIFO，所以会导致多个写操作之间的逻辑顺序被破坏。

```
a = 1;
b = 2;
```

在TSO的内存模型上，我们能保证b=2时，a一定 = 1，但是在PSO上，不能保证，这就会出现store-store乱序。

- 宽松内存模型（RMO）

在PSO的基础上，继续丧心病狂地放宽限制来提高效率，打破读写操作之间的顺序。

除了store-load乱序，store-store乱序，还会出现load-load，load-store乱序。

简单来说就是RMO的cpu说，你的高层代码翻译到我这里来，我想怎么排就怎么排，我按我自己心意来执行，就是开心就是玩儿。

到此，我们终于能稍微明白内存序的作用了。**内存序就是高层对底层cpu内存模型的一种封装**。

c++在执行atomic操作时，传入的不同内存序参数，就是在告诉你，它会模拟上述哪一种内存模型来处理代码执行顺序。

memory的总体分类和内存序的对应：

- **memory_order_seq_cst:**
  这是所有atomic操作内存序参数的默认值，语义上就是要求底层提供顺序一致性模型，不存在任何重排，可以解决一切问题，但是效率最低。
- **memory_order_release/acquire/consume:**
  提供release、acquire或者consume，release语意的一致性保障
  它的语义是：我们允许cpu或者编译器做一定的指令乱序重排，但是由于tso，pso的存在，可能产生的store-load乱序store-store乱序导致问题，那么涉及到多核交互的时候，就需要手动使用release，acquire去避免这样的这个问题了。简单来说就是允许大部分写操作乱序（只要不影响代码正确性的话），对于乱序影响正确性的那些部分，程序员自己使用对应的内存序代码来控制。
- **memory_order_relaxed:**
  这种内存序对应的就是RMO，完全放开，让编译器和cpu自由搞，很容易出问题，除非你的代码是那种不论怎么重排都不影响正确性的逻辑，那么选择这种内存序确实能提升最大性能。

综上，最实用的还是memory_order_release和memory_order_acquire这两种内存序，兼顾了效率和代码的正确性。

- memory_order_release

如果用了这种内存序，保证在本行代码之前，有任何写内存的操作，都是不能放到本行语句之后的。

也就是可以让程序员可保证一段代码的写顺序。

假设我们还是希望a=1的执行在b=2之前（对于所有共享ab的线程来说都是一致的），可以这样实现。

```
//-将下面代码用release控制
int a = 0;
int b = 0;
void func1(){
    a = 1;
    b = 2;
}

//------------------------------分割线----------------------------

int a = 0;
std::atomic<int> b(0);
void func1(){
    a = 1;
    b.store(2,std::memory_order_release);//-a的写操作不会重排到b的写操作之后
}
```

- memory_order_acquire

如果用这种内存序，保证在本行代码之后，有任何读内存的操作，都不能放到本行语句之前。

也就是可以让程序员可保证一段代码的读顺序。

对于线程2而言，我们无法保证cout<<a<<endl;会不会重排到while(b≠2);之前，所以可以这样修改代码

```
int a = 0;
int b = 0;
void func1(){
    a = 1;
    b = 2;
}
void func2(){
    while(b ≠ 2);
```

```
    cout<<a<<endl;
}

//------------------------------分割线----------------------------

int a = 0;
std::atomic<int> b(0);
void func1(){
    a = 1;
    b.store(2,std::memory_order_release);//-a的写操作不会重排到b的写操作之后
}
void func2(){
    while(b.load(std::memory_order_acquire) != 2);
    cout<<a<<endl;//-a的读操作不会重排到b的读操作之前
}
```

c++内存屏障

那么内存屏障是什么，简单来说，就是我们希望上述的代码在逻辑上更纯粹，我们希望a和b就是纯纯的两个非原子int，而不是让b变成原子变量来保证执行顺序。

内存屏障就可以想象成用一个匿名的原子变量来保证执行顺序，不需要让b变成原子变量了，代码如下：

```
int a = 0;
int b = 0;
void func1(){
    a = 1;
    std::atomic_thread_fence(std::memory_order_release);
    b = 2;
}
void func2(){
    while(b != 2);
    std::atomic_thread_fence(std::memory_order_acquire);
    cout<<a<<endl;
}
```

使用release屏障，相当于写操作a=1不会重排到b之后。

使用acquire屏障，相当于读操作cout<<a不会重排while(b !=2)之前。

和刚刚实现了一样的功能。

对于无锁编程的理解

我个人认为lock-free不能简单理解成无锁，因为本身CAS就是一个自旋锁的机制，我感觉无锁和有锁更像是互斥锁和自旋锁的区别，或者说悲观锁和乐观锁的区别。

在执行 i++的时候，互斥锁觉得本线程在执行汇编的三条语句时，一定会被其它线程干扰，所以干脆在i++之前就加锁，自增后解锁，代码如下：

```
int i;
mutex m;
void func1(){
    lock_guard<mutex> lock(m);
    i++;
}
```

atomic则是利用CAS的机制，我先判断 i 是不是=旧值，如果=旧值说明没被其它线程干扰，于是 i 更新成new_value，这就有点乐观的意思了，因为atomic优先觉得本线程是没有被其它线程干扰的，大不了compare不成功，就不更新新的值呗。

参考

C++ 中的 volatile, atomic 及 memory barrier

内存一致性模型_langren388的博客-CSDN博客_内存一致性模型

如何理解 C++11 的六种 memory order?