C++11内存模型完全解读-从硬件层面和内存模型规则层面双重解读

C++11标准引入了一套新的内存模型，这套模型中共有三种模型，分别为sequentially-consistent模型、acquire-release模型和relaxed模型。三种模型对内存序的约束力是不一样的，sequentially-consistent的约束力最强，但是执行效率也最低。relaxed模型约束力最差，执行效率最高。acquire-release模型约束力居中，属于半约束，执行效率也居中。

本文假设你已经对C++的这些内存模型有一些基本的概念，并且也了解一些硬件方面的知识，比如缓存一致性协议(MESI)和store-buffer以及invalidate-queue。相关描述在网上有很多，比如硬件角度看内存屏障，为什么需要内存屏障(尤其是这个，这里面通过例子讲了很多重要的知识点，强烈推荐，看懂这个，你就会恍然大悟)以及知乎专栏的一系列文章(知乎上很多大牛都有相关文章，强烈建议去看看)等等。

另外，有很多博客和书籍也专门讨论了C++11的内存模型，比如大名鼎鼎的《C++ Concurrency In Action 2nd》，这本书比较详细的介绍和讨论了C++11的内存模型以及锁的使用，此书强烈建议看英文原版，中文版在内存模型那些章节翻译的太差，网上有很多英文pdf，加上谷歌翻译，有道翻译，基本就能看懂了。此外，还有Jeff Preshing在2012年的一些文章，也都特别的好。另外，有一些不懂的问题如果去StackOverFlow上查询的话，可能会获得很多意外的惊喜！

一、预备知识

在开始之前，还需要准备一些预备知识，或者一些口头约定，以方便下文的讨论。

1. 同步点：

对于一个原子类型变量a，如果a在线程1中进行store(写)操作，在线程2中进行load(读)操作，则线程1的store和线程2的load构成原子变量a的一对同步点，其中的store操作和load操作就分别是一个同步点。

可以看出，同步点具有三个条件：

- 必须是一对原子变量操作中的一个，且一个操作是store，另一个操作是load；
- 这两个操作必须针对同一个原子变量；
- 这两个操作必须分别在两个线程中。

2. synchronized-with(同步)：

对于一对同步点来说，当写操作写入一个值x后，另一个同步点的读操作在某一时刻读到了这个变量的值x，则此时就认为这两个同步点之间发生了同步关系。

同步关系具有两方面含义：

- 针对的是一对同步点之间的一种状态的描述；

- 只有当读取的值是另一个同步点写入的值的时候，这两个同步点之间才发生同步；

也就是说，如果读取的值不是另外一个同步点写入的值，则此时这两个同步点之间并没有发生同步。

3. happens-before(先于发生):

当线程1中的操作A先执行，而线程2中的操作B后执行时，A就happens-beforeB。happens-before是用来表示两个线程中两个操作被执行的先后顺序的一种描述。

happens-bofore有三个个特点:

- 可传递性。如果Ahappens-beforeB，Bhappens-beforeC，则有Ahappens-beforeC；
- 当store操作A与load操作B发生同步时，则Ahappens-beforeB；
- happens-before一般用于描述分别位于两个线程中的操作之间的顺序。

4. sequenced-before:

如果在单个线程内操作A发生在操作B之前，则表示为Asequenced-beforeB。这个关系是描述单个线程内两个操作之前的先后执行顺序的，与happens-before是相对的。

此外，sequenced-before也具有可传递性，并且sequenced-before与happences-before之间也具有可传递性：如果线程1中操作Asequenced-before操作B，而操作Bhappences-before线程2中的操作C，操作Csequenced-before线程2中的操作D，则有操作Ahappences-before操作D。

二、内存模型简述

1. relaxed order:

当程序员所写的代码被编译器翻译成机器语言时，编译器可能会为了优化性能来重排程序员所写的代码，比如:

```
int a = 0;
int b = 0;

void func()
{
        int t = 1;
        a = t;
        b = 2;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

- 9
- 10
- 11

编译器最终优化后的代码可能是这样子的：

```
int a = 0;
int b = 0;

void func()
{
        b = 2;
        a = 1;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

在单线程中，这种优化是无关紧要的，因为这两个变量是不相关的，谁先谁后，最后结果一样。但是，如果在多线程环境中，比如另一个线程通过b的值来输出a的值：

```
void func2()
{
        while (b != 2);
        std::cout << a << std::endl;
}
```

- 1
- 2
- 3
- 4
- 5
- 6

假如func()与func2()是在不同线程中执行，则func2()中的输出结果可能就不是1，因为编译器可能改变了func()中的代码顺序。

即使编译器没有重排你的代码，最终CPU执行的时候可能也会不一样(这里假设你已经了解缓存一致性协议(MESI)和store-buffer以及invalidate-queue)。变量a的值1可能暂时存储到CPU1的store-buffer中，变量b的值2可能存储到CPU2的cacheline中，然后func2()可能是在CPU2上执行，此时CPU2从cacheline上读取b的值，发现是2，因此while循环退出，执行输出语句。但是此时a的最新值1在CPU1的store-buffer中，因此CPU2上看不到a的值1，只能看到a的值0，因此就会输出0，而不是输出1。

如果a和b都是原子变量，且其store操作和load操作都是用的relaxed内存序，则其执行过程跟上述非原子变量类似。
relaxed内存序模型允许编译器对代码的任意优化和重排，也允许CPU的指令重排，relaxed唯一保证的是原

子变量上的操作都是原子性的，即一个操作不会被中断，是排他性的，只有当一个操作完成后，才能执行另一个操作，即使是多线程。但是其他方面就不能保证了，例如上面分析的那样。

上面是从硬件层面来分析的，下面从内存模型规则方面来分析。

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x, y;
std::atomic<int> z;

void write_x_then_y()
{
        x.store(true, std::memory_order_relaxed); // 1
        y.store(true, std::memory_order_relaxed); // 2
}

void read_y_then_x()
{
        while(!y.load(std::memory_order_relaxed)); // 3
        if(x.load(std::memory_order_relaxed)) // 4
                ++z;
}

int main()
{
        x = false;
        y = false;
        z = 0;
        std::thread a(write_x_then_y);
        std::thread b(read_y_then_x);
        a.join();
        b.join();
        assert(z.load()!= 0); // 5
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

上述代码最终在表达式5处有可能会触发assert，因为x和y用的relaxed，所以1和2处的代码可能会被重排，导致y = true时，x仍然为false。从代码中可以看出，2和3这两个操作分别是一对同步点，所以当3处读取的值为2处写入的那个值时(即3处读取的值为true时)，2和3发生了同步，且表达式2happences-before表达式3。但是，由于使用的relaxed内存序，所以表达式1没有sequenced-before表达式2，表达式3也没有sequenced-before表达式4。因此，表达式1并没有happens-before表达式4，因此最终无法确定表达式4一定会在表达式1被执行前执行，最终导致z的值可能仍然为0。

这里需要多啰嗦一点，上面说“无法确定表达式4一定会在表达式1被执行前执行”，其实更准确的说，应该是：**表达式4在执行的时候，其所属线程可能还看不到另一个线程中表达式1对x值的修改动作。**也就是说，表达式4在执行的时候，表达式1或许已经执行了，但是x的新值并没有被同步，导致表达式4所属CPU(或线程)并没有**感知**到x值的修改，这也是**线程感知内存模型**的由来。**因此，下文中如果涉及到线程间的操作的先后执行，更严格意义上来说是线程间的操作可被感知。**

例如线程1中有三个操作A，B，C，是按顺序执行的，但是在线程2看来，线程1中的这三个操作顺序可能是CBA，BCA，ACB等等，线程3看到的可能又是另一番景象。即使是两个线程执行同一块汇编指令，最终的顺序可能都不一样。这种情况下，唯一能保证的是所有的线程对同一个原子变量的修改顺序的感知是一样的。比如原子变量a，假如先执行a = 2，再执行 a = 6，最后a = -1，则任何线程看到a的值的变化顺序都是2，6，-1，而不会是任何其他顺序。但是不同线程在某一个时刻同时观察这个变量时，可能看到的值是不一样的。比如在某个时刻，线程1看到的a值是2，而在同一时刻，线程B看到的值可能是6，甚至是-1。当然对于不同变量间的相互顺序，那就不确定了。

综上所述，relaxed模型不保证代码执行顺序，只保证原子变量上操作的原子性(即排他性)。事实上，原子变量上操作的原子性对于其他两个模型也都是保证的。

2. acquire-release order:

当原子变量同步点的store操作是memory_order_release或memory_order_acq_rel时，而对应的另一个同步点的load操作是memory_order_acquire或memory_order_acq_rel或memory_order_consume时，此时就是acquire-release内存序模型。标准规定：

1. *在release之前的所有store操作绝不会重排到(不管是编译器对代码的重排还是CPU指令重排)此release对应的操作之后，也就是说如果release对应的store操作完成了，则C++标准能够保证此release之前的所有store操作肯定已经先完成了，或者说可被感知了；*
2. *在acquire之后的所有load操作或者store操作绝对不会重排到此acquire对应的操作之前，也就是说只有当执行完此acquire对应的load操作之后，才会执行后续的读操作或者写操作。*

```
// 这里的变量既有普通全局变量，又有原子类型的全局变量
int a = 0;
```

```
float b = 0.0;
short c = 0.0;
double d = 0.0;
char e = 's';
std::atomic<int> ai{0};
std::atomic<bool> go{false};

void write()
{
        int t = 1; //1
        a = t + 1; // 2
        b = 45.9; // 3
        c = 25; // 4
        ai.store(45, std::memory_order_relaxed); // 5
        go.store(true, std::memory_order_release); //6
        d = 10.0; // 7
        e = 'g'; // 8
}

void read()
{
        std::cout << a << std::endl; // 9
        while (!g.load(std::memory_order_acquire)); // 10
        std::cout << b << c << ai << std::endl; // 11
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29

表达式6处的std::memory_order_release能够保证上面的1,2,3,4,5表达式的执行一定是在表达式6之前完成。一旦go的值变成true了，那么可以肯定1,2,3,4,5表达式所对应的值也已经存储完成了，且其他线程是能够获取到这些改变后的值的，而不会因为在Cache中没有同步而造成不一样的情况(当然只有当检测到go位true之后才能如此确定)。**不过，对于1,2,3,4,5这几个表达式，它们5个之间的执行顺序可以任由编译器重排或者处理器乱序执行，它们5个相互之间是无约束的。**此外，对于表达式7和8来说，它俩就没有限制，它俩可以任由编译器重排，且可以重排到表达式6之上。release内存序只对其前面的写操作有作用。另外，对于10之后的所有读或者写操作，都会等到10这个表达式完成后才执行，但是表达式9与表达式10之间就没有顺序要求，编译器或者CPU可以将9重排到10之后执行。acquire内存序只对其后面的读或者写操作才有作用。
一个简单的记忆方法是：**我先读，我后写。**
下面使用硬件方面的知识来探究这种模型的可能实现方式(比较复杂(如果你看完为什么需要内存屏障这篇文章，可能就比较容易理解了)，如果想简单理解，可以直接跳过这里的硬件逻辑解释，看下面的内存模型规则的解释，比较简单且通用)：

> *当对一个store操作使用release时，首先会阻止编译器将此store操作之前的任意store操作重排到此store操作之后，也就是说生成的汇编指令中，上述代码的1~5的汇编指令都会在6的汇编指令之前；其次，编译器会在此store操作**执行之前**插入一个内存屏障(memory barrier，又称内存栅栏)指令，而且是写内存屏障(store memory barrier，smb)指令。此指令会告诉CPU在执行后续的store之前必须先把store-buffer中的数据flush，或者是通过stall一段时间直到store-buffer清空，或者使用store-buffer把后续将要写入cacheline中的值也缓存到store-buffer中(而不是直接写入到cacheline中)，因为如果变量本来就在此CPU的cacheline中且处于M或E状态，一般正常情况下是可以直接将新值写入到cacheline中而无需与store-buffer交互的。此文中假设smb命令要求后续的写操作都要写到store-buffer中。*
> *当对一个load操作使用acquire时，首先会阻止编译器将load操作之后的任何store或load操作重排到此acquire对应的load操作之前，且也会在此load操作**执行之后**插入一个读内存屏障(read memory barrier，rmb)。rmb会要求将此CPU的invalidate-queue中的invalidate消息全部执行完后再执行其他操作。来看看它们是怎么解决乱序的，首先假设上述代码中6和10处都是用的relaxed内存序，且编译器为执行代码重排，则有：*
>
> 1. *根据上述代码，假设write()函数在CPU1上执行，初始时变量a，b和go是在CPU1的cacheline上且处于E状态，c和ai在CPU2的cacheline上且处于E状态；*
> 2. *当执行2和3表达式时，由于a和b就在CPU1的cacheline上且是E状态，因此写入的新值并不会缓存到CPU1的store-buffer中，而是直接写到cacheline中，且也不需要发送invalidate消息给其他CPU的cache；*
> 3. *当执行表达式4和5时，由于CPU1上的cacheline中没有这两个变量，因此其会先将对应的值按照FIFO(先进先出)的方式写入到store-buffer中，并发送read-invalidate消息给CPU2，要求CPU2给出这两个变量的地址和值，并要求CPU2将这两个变量的cacheline变成I状态(store-buffer中的值什么时候才能刷回到cacheline中呢？当store-buffer中对应的值的invalidate-ack消息(由其他CPU发过来的)被收到时，才会刷回到cacheline里)。假如CPU2比较繁忙，它会将CPU1发来的invalidate消息存储到自己私有的invalidate-queue中，并立即返回变量c和ai的值(都是0)和地址以及invalidate-ack消息；*
> 4. *CPU1在接收到invalidate-ack消息前，就可以继续执行表达式6(比如可能是由于指令并发的原因，一个时钟周期内可以执行多条指令)。由于go的值在CPU1的cacheline里面且处于E状态，因此CPU1可以直接将true值存储到cacheline中并且不需要向其他CPU发送invalidate消息；*
> 5. *CPU2此时接收到了CPU1上一步发送的关于变量c和ai的invalidate消息，并且CPU2并没有立即处理这两个invalidate消息，而是将消息存到了CPU2的私有invalidate-queue中，并立即返回c和ai的地址和值以及invalidate-ack消息；*
> 6. *假如CPU2执行的是read()函数，此时已经执行到了while循环处，CPU2需要读取go的值，但是CPU2发现本地cacheline里面没有go的值，因此就向CPU1发送read消息(这是CPU2在执行了第5步之后执行的)。*
> 7. *CPU1首先接收到了CPU2返回来的c和ai的地址和值以及invalidate-ack消息，因此便将store-buffer中的c和ai的新值刷回到CPU1的cacheline中。现在CPU1的cacheline里面有*

了c和ai的值且是最新的。因此CPU1中c和ai对应的cacheline的状态会改为M。但是记住，此时CPU2中的cacheline里也有c和ai的值，且状态为E，因为CPU2中的invalidate-queue中的invalidate消息还未被执行；

8. *CPU1随后又接收到CPU2读取go值的消息，因此就把cacheline中最新的go = true传给了CPU2；*
9. *CPU2接收到了最新的go值，从而退出了while循环，因此开始执行后续的c和ai的读取操作。由于c和ai本来就在CPU2的cacheline里且并不是I状态，所以CPU2直接就从CPU2的cacheline里拿取c和ai的值并打印。因此，打印出来的c和ai的值仍然是未修改前的值；*
10. *此时CPU2才开始执行invalidate-queue中的invalidate消息，把CPU2的c和ai的cacheline状态改为I，但是现在一切都晚了。*

假如6和10处分别使用了对应的内存序标记，则就会加上对应的内存屏障操作。CPU1写入go的新值前，会遇到smb，因此后面的go值并不会直接写入到cacheline中，而是写入到store-buffer中，且处于store-buffer中c和ai的值的后面。因此后面如果将store-buffer的值刷回到cacheline中时，肯定是先把c和ai的值刷回去，然后才把go的值刷回去，所以其他CPU发现CPU1中的go的cacheline变成true之前，一定会先感知到CPU1中c和ai的变化。当while循环读取go值后，会遇到rmb指令，就会刷新CPU2中的invalidate-queue消息，等invalidate-queue中的invalidate消息执行完成后才继续执行对应的内存读操作，所以CPU2在读取c和ai值之前，就会执行CPU1的要求，将c和ai的cacheline修改为I状态，这样，后面读取c和ai时，CPU2就会去CPU1上读取。这样，就不会出问题了。

上面大概就是硬件层面的实现方式，这里只是可能的方式，并不一定，读者不要认为所有CPU架构都是这么玩的。

下面我们从C++的内存模型规则来分析，内存模型规则才是所有人都应该要理解并记住的，因为这是标准所能保证的，是更高层次的一种抽象。

*由release内存序的规则可知，表达式1~5都是sequenced-before表达式6的。表达式6与表达式10构成了一对同步点，因此当表达式10获取的值是表达式6写进去的那个值的时候，表达式10和表达式6就构成了同步关系(synchronized-with)。根据上面对happens-before的描述可知，此时表达式6happens-before表达式10。又根据acquire的规则可知，表达式10是sequenced-before表达式11的，**所以这里的先于发生构成一条链子**。由happens-before与sequenced-before的可传递性得知：表达式1~5最终是happens-before表达式11的。因此，表达式11读取的值一定是表达式1~5更新进去的值。*

为了更加深刻的理解这个内存序模型，再看一个例子：

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
        x.store(true, std::memory_order_release); // 1
}

void write_y()
{
        y.store(true, std::memory_order_release); // 2
}
```

```
void read_x_then_y()
{
        while(!x.load(std::memory_order_acquire)); // 3
        if(y.load(std::memory_order_acquire)) // 4
                ++z;
}

void read_y_then_x()
{
        while(!y.load(std::memory_order_acquire)); // 5
        if(x.load(std::memory_order_acquire)) // 6
                ++z;
}

int main()
{
        x = false;
        y = false;
        z = 0;

        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);

        a.join();
        b.join();
        c.join();
        d.join();

        assert(z.load() != 0); // 7
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50

上面代码尽管x和y的store操作使用的是release语义，x和y的load操作使用的是acquire语义，但是最终表达式7仍然可能会触发assert(如果全都改成seq_cst，最后就能保证一定不会出现assert了)。让我们来用内存模型的规则分析一下：

> *从代码中可以看出，表达式1和3构成一对同步点，2和5构成另一对同步点。当某个时刻表达式3读出x的值为true时(一定会在某个时刻读到true，因为是while循环)，此时1和3就发生了同步，那么此时表达式1happens-before表达式3，所以表达式1上面的所有store操作(然鹅此处没有任何操作)就会先于表达式4发生。然而，对当前1和3发生的同步来说，表达式2并不是在表达式3的上面执行，所以表达式2并没有happens-before表达式4(如果表达式2是在表达式1的上面执行的话，那就有happences-before关系了)，所以表达式4读取的y值仍然可能是false，导致z值为0。同理，表达式2会happens-before表达式5，但是表达式1没有happens-before表达式6，所以表达式6处读取的x值仍然可能为0。因此，最后，z的值就为0。*
> *所以记住：**acquire-release模型只对当前发生了同步的两个同步点及其同步点处前后的其他共享变量的内存操作有执行顺序的约束，对其他额外的线程，额外的位置没有这种顺序约束，并且对曾经在其他地方发生过同步的原子变量也没有约束。**(这里的"曾经"会在下一小节的seq_cst有解释的)*
> *此外，还可以再多分析一点。其实表达式1与表达式6也构成了一对同步点，表达式2与表达式4也构成了一对同步点。但是，1和6不一定会发生同步(1和3一定会发生同步)，2和4也不一定会发生同步(2和5一定会发生同步)。也就是说，表达式1一定会在程序运行的某个时刻与表达式3发生同步，从而使表达式1happens-before表达式3。但是在整个程序运行过程中，表达式1不一定会与表达式6发生同步，因为表达式6不是循环，导致表达式6处读取的x值即使不为true，也会执行过去，从而结束。同理，表达式2一定会在某个时刻与表达式5发生同步，从而使表达式2happens-before表达式5，但是表达式2不一定会与表达式4发生同步。因此，可以想象，如果把表达式4和6处都改为while循环，则表达式2和4也会发生同步，同理表达式1和6也会发生同步，最终z值一定不会为0。(哈哈，这里是显而易见的呀，都不需要用这个方法来分析，因为毕竟是while循环，只有x和y值为true时，才能执行下面的++z*

语句。这本来就是天然正确的。不过，这里通过上面这种方法的分析，也能确定此分析方法是正确的，且能更清晰的认识这种方法以及acquire-release内存序模型)

acquire-release的可传递性:

看下面例子:

```
std::atomic<int> data[5];
std::atomic<bool> sync1(false), sync2(false);

void thread_1()
{
        data[0].store(42, std::memory_order_relaxed);
        data[1].store(97, std::memory_order_relaxed);
        data[2].store(17, std::memory_order_relaxed);
        data[3].store(-141, std::memory_order_relaxed);
        data[4].store(2003, std::memory_order_relaxed);
        sync1.store(true, std::memory_order_release); // 1
}

void thread_2()
{
        while(!sync1.load(std::memory_order_acquire)); // 2
        sync2.store(true, std::memory_order_release); // 3
}

void thread_3()
{
        while(!sync2.load(std::memory_order_acquire)); // 4
        assert(data[0].load(std::memory_order_relaxed) == 42);
        assert(data[1].load(std::memory_order_relaxed) == 97);
        assert(data[2].load(std::memory_order_relaxed) == 17);
        assert(data[3].load(std::memory_order_relaxed) == -141);
        assert(data[4].load(std::memory_order_relaxed) == 2003);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29

上面例子中thread_3中的assert永远不会触发，因为acquire-release具有可传递性。

由代码可知，thread_1中对data的所有store操作都sequenced-before表达式1。当表达式2读取的值为true时，表达式1就happences-before表达式2。在thread_2中，表达式2sequenced-before表达式3(因为acquire语义的关系)。当表达式4读取的值为true时，表达式3就happences-before表达式4。最后，在thread_3中，表达式4sequenced-beforedata的load操作。因此，data的所有store操作sequenced-before表达式1happences-before表达式2sequenced-before表达式3happences-before表达式4sequenced-before对data的所有load操作。根据sequenced-before和happences-before的可传递性得知，对data的所有store操作都happences-before对data的所有load操作。

因此，尽管thread_2中没有涉及到对data的任何数据操作，最后都能确定thread_1中对data的store操作是先于thread_3中对data的load操作的，这就是acquire-release的可传递性。

其实，上面的功能完全可以只用一个原子变量配合RMW操作来实现：

```
std::atomic<int> sync(0);

void thread_1()
{
        // ...
        sync.store(1, std::memory_order_release); // 1
}

void thread_2()
{
        int expected = 1;
        while(!sync.compare_exchange_strong(expected, 2, std::memory_order_acq_rel)) // 2
                expected = 1;
}

void thread_3()
{
        while(sync.load(std::memory_order_acquire) < 2); // 3
        // ...
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

注意，thread_2中RMW操作使用的是memory_order_acq_rel内存序标记，这种情况下，表达式1与表达式2中的load部分可以构成一对同步点(构成第一个acquire-release)并且能够在某时刻发生同步，表达式2中的store部分与表达式3可以构成另一对同步点(构成第二个acquire-release)并在某一时刻发生同步。其实，这里的RWM操作即使是用relaxed内存序标记，最后也能实现表达式1之前的store操作happences-before表达式3之后的load操作，这就是下面介绍的release-sequence rule。不过如果RMW操作没有使用acq_rel或者seq_cst，则RWM操作的上下位置的其他操作是与表达式1或者表达式3周围的操作构不成某种顺序约束关系的。

在acquire-release 模型中，有一个规则叫做release-sequence rule。这个规则其实类似于上面的acquire-release可传递性，它的大意为：
线程1中对原子变量进行store操作，线程2对此原子变量执行RMW(读-修改-写，read-modify-write)操作，线程3对此原子变量也执行RMW操作，线程4与线程3类似…线程n对此原子变量执行load操作或者RMW操作。其中上述所有的store操作都是用的release、acq_rel或seq_cst语义之一，而所有的load操作都用的是acquire、acq_rel、seq_cst或consume语义之一，且除了最后一个线程中的RMW不能用relaxed语义外，其他任何中间线程的RMW操作都可以用任何内存序语义，包括relaxed。此时，对于所有的这些线程，**如果一个线程load得到的值是上一个线程store进去的(即线程n读取的值是线程n-1写进去的，线程n-1读取的值是线程n-2写进去的…)，则这些线程间的操作构成了release-sequence，并且此时可以得出线程1的store操作与线程n的load操作是构成了同步关系，它俩之间具有happens-before的关系。** 注意，中间线程如果用的是relaxed的RMW操作，则中间的那个线程与其他线程之间无同步关系，也无happens-before关系(也就是说此时中间的这些线程的同步点上下如果有对共享变量的load或者store操作，则这些操作是没有顺序可言的，这些操作与其他线程中对应的操作也无顺序可言，编译器可以任意优化重排)，但是即使这样，这些线程连接起来，其首和尾就构成了happens-before关系。
此外，如果中间线程的RMW操作全部都是acquire、acq_rel或seq_cst语义之一，则所有的中间线程都与第一个store线程构成happens-before关系。

例如下面代码，其中x初始值为0：
```
// 线程 1:
A; // 表示对共享变量的一系列内存操作
x.store(2, memory_order_release);

// 线程 2:
B; // 与A类似
int n = x.fetch_add(1, memory_order_relaxed);
C; //与A类似

// 线程 3:
int m = x.load(memory_order_acquire);
D; // 与A类似
```

- 1
- 2
- 3

- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

上述代码中n的值可能为0或2，而m的值可能为0，1，2，3。

假如n = 2且m = 3，则此时就构成了release-sequence，因为线程2读取的值是线程1存储进去的，而线程3读取的值是线程2存储进去的，此时就可以确定在执行D之前，A一定已经先被执行完了，A是happens-beforeD的。但是B和C的顺序就无法保证，因为其用的是relaxed语义，且B和C这两个表达式与A和D之间无法构成happens-before关系。

此外，如果某时刻能确定m = 2，则也能得出A是happens-beforeD的，因为m的获取的值直接就是线程1存入的值。此时是无需考虑线程2的。也就是说，只要m = 2或m = 3，则就能确定A是happens-beforeD的。

另外，如果线程2的fetch_add()是用的是非relaxed且非release，如果能确定n = 2，则能得出Ahappens-beforeC。同理，如果fetch_add()是用的是release、acq_rel、seq_cst，并且发现m = 1或m = 3，则此时能得出Bhappens-beforeD。

另外，release-acquire可用于实现锁，锁操作中的lock操作相当于acquire，锁的unlock操作相当于release。它们构成了一块互斥区域，用于保护数据。

consume：

consume语义是一种弱的acquire，它只对关联变量进行约束，这个实际编程中基本不用，且在某些情况下会自动进化成acquire语义(比如当用consume语义修饰的load操作在if条件表达式中时)。另外，C++17标准明确说明这个语义还未完善，建议直接使用acquire，且在《C++ Concurrency In Action 2nd》中作者也建议不要使用这个语义。因此，这里就不讨论了。

3. sequence-consistent order(seq_cst)：

(下文对此内存序模型的解释不够友好，我以后想办法解释的更通用更容易理解一点)

这种内存模型具有最强约束力，它不允许编译器对相关变量进行重排序，并且，它会在CPU的各个Cache之间产生大量的同步，以产生一致性的顺序，因此其效率也是最低的。其核心思想是：**任何线程中使用了acq_rel标记的原子变量的内存操作对于其他任何线程都是可感知的。**也就是说，如果使用了acq_rel的内存操作A在线程1中被执行了，则其他任何线程都能感知到操作A对原子变量的值的修改，而不会因为值缓存在store-buffer中而无法感知。

从硬件角度来看的话，使用此内存序语义修饰load或store或RMW操作时，就像是在这个操作的前面和后面都插入了smb指令和rmb指令，以实现最大的同步。或者你可以假想成那些用seq_cst语义修饰的原子变量的store操作的之前的所有其他变量store操作都直接将值写到了内存中，而用seq_cst修饰的原子变量的load操作之后的所有其他变量load操作都直接从内存中拿取值，这些过程中的值根本不会缓存到Cache中(只是假想，这样有助于记忆和理解)。

从内存模型规则的角度来看的话，不管是load操作还是store操作，只要是用了此内存序标记，其前面的任何操作都不会重排到此操作的后面(当然其前面的那些操作相互之间是可以重排的，无影响，类似于

acquire-release语义)，且此操作后面的任何操作都不会重排到此操作的前面(同理，此操作后面的那些操作相互之间可以重排)，且一旦某个内存操作完成了，其他任何线程都能感知到。

注意，seq_cst内存序中acquire-release的规则仍然在，只不过多加了一些强制排序的规则，即**曾经在其他地方发生过同步的原子变量也能继续参与本次的同步。**你可以直接想象成所有用seq_cst修饰的store或load或RMW操作都是在同一个线程中执行的。这样，如果曾经某个原子变量有过store操作，那么在此之后，所有再load此原子变量的操作都能拿到曾经store进去的值，因为是在同一个线程中按顺序执行的(假想的)，然后，再在此基础上运用acquire-release的规则去分析原子此原子变量store和对应的load操作前后的其他变量的内存操作的顺序就可以了。

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
        x.store(true, std::memory_order_seq_cst); // 1
}

void write_y()
{
        y.store(true, std::memory_order_seq_cst); // 2
}

void read_x_then_y()
{
        while(!x.load(std::memory_order_seq_cst)); // 3
        if(y.load(std::memory_order_seq_cst)) // 4
                ++z;
}

void read_y_then_x()
{
        while(!y.load(std::memory_order_seq_cst)); // 5
        if(x.load(std::memory_order_seq_cst)) // 6
                ++z;
}

int main()
{
        x = false;
        y = false;
        z = 0;

        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);

        a.join();
        b.join();
        c.join();
        d.join();
```

```
        assert(z.load() != 0); // 7
}
```

上面代码最后的表达式7永远不会触发assert，这是C++标准所保证的。

首先，因为用的是seq_cst内存序语义，所以表达式3一定sequenced-boefore表达式4，表达式5一定sequenced-boefore表达式6。且1和3发生同步时，表达式1也会happens-before表达式3，同理在2和4发生同步时，表达式2也会happens-before表达式4。

假如表达式3在某一时刻退出循环，则表达式1一定已经被执行了，此时由于表达式1用的是seq_cst语义，**因此所有线程都能感知到表达式1被执行了，且所有线程在读取x值时，都能得到表达式1对x更新后的值。**假如此时表达式2还未被执行(可能是线程b还未被操作系统调度起来)，因此表达式4处的if就会判断失败，所以++z不会被执行。然后过了一段时间，表达式5终于在某个时刻读取到y值为true了，然后就开始执行表达式6。由上面分析知，刚才表达式1对x值的变更会立刻被线程d所感知。因此表达式6读取x值时，读取的就是表达式1存储进去的值，因此会执行++z。从另一方面来看，当2和5发生同步的时候，由于表达式1**曾经**与3发生过同步，因此在2和5发生同步的时刻，表达式1也能参与同步。假如表达式1上面有很多其他共享变量的store操作，则此时这些store操作一定发生在表达式6后面的内存操作执行之前。或者想象成是在单线程中执行的，因为x的store(表达式1)是先发生的，所以后续不管什么时候对x进行load(不管是表达式3还是表达式6)，其都能拿取到表达式1存储的值，因为是在单线程中的(假想的)。

由上面分析可以看出，seq_cst内存序模型与acquire-release模型很像，都需要同步点发生同步的时候，才能确定同步点前后的其他内存操作具有happens-before关系。不过，比acquire-release模型更强的是，对曾经发生过同步的原子变量来说，也能参与到后面的同步上。其实这里感觉比较难理解，建议seq_cst模型用**线程感知**的概念或**假想成单线程的情况**来理解，可能会轻松点。

三、一些例子解读

1. 如果有两个load操作同时对应于一个store操作会如何?

```
std::vector<int> vi;
std::atomic<bool> ready{false};

void write()
{
        for (int i = 0; i < 10; ++i) // 1
        vi.push_back(i * 10);

        ready.store(true, std::memory_order_release); // 2
}

void read1()
{
        while (!ready.load(std::memory_order_acquire)); // 3
        process1(vi); // 4
}

void read1()
{
        while (!ready.load(std::memory_order_acquire)); // 5
        process2(vi); // 6
}
```

   - 1
   - 2
   - 3
   - 4

- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

这种情况下，分别有两对同步点，2和3构成一对，2和5构成另一对。因此，4和6都会在1执行完成后再执行，1happens-before4并且1happens-before6。但是，4和6之间没有先于关系。如果4和6都是只读的操作，则没啥影响。但是如果4和6中有修改操作，则4和6之间就有可能产生数据竞争。

2. 如果有两个RWM操作同时对应于一个store操作会如何?

```
std::vector<int> queue_data;
std::atomic<int> count{ 0 };

void populate_queue()
{
        unsigned const number_of_items = 20;
        queue_data.clear();
        for (unsigned i = 0; i < number_of_items; ++i)
        {
                queue_data.push_back(i);
        }

        count.store(number_of_items, std::memory_order_release);
}

void consume_queue_items()
{
        auto id = std::this_thread::get_id();
        std::string s;
        std::stringstream is;
        is << id;

        std::string str_id;
        is >> str_id;
        thread_local std::ofstream out(str_id + ".txt");
        if (!out.good())
                return;

        while (true)
        {
                int item_index;
```

```
                if ((item_index = count.fetch_sub(1, std::memory_order_acq_rel)) <= 0)
                {
                        std::this_thread::yield();
                        continue;
                }

                out << queue_data[item_index - 1] << '\n';
        }
}

void execute()
{
        std::thread a(populate_queue);
        std::thread b(consume_queue_items);
        std::thread c(consume_queue_items);
        a.join();
        b.join();
        c.join();
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36

- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51

线程b和线程c虽然执行的是同一块汇编代码，但是它们分属两个线程。因此，线程b和c中的fetch_sub所读取的值有可能是线程a中的store存储进去的，也有可能是线程b或c的fetch_sub写进去的。所以线程b或c可能直接跟线程a中的代码有happens-before关系，也可能与线程b或c中的代码有happens-before关系。如果是后者，则就构成了release-sequence。这也是《C++ Concurrency In Action 2nd》中用于解释release-sequence rule的一个例子。对于上述代码，只能确定线程b和c中对queue_data的读取操作一定晚于线程a中对queue_data的写入操作，但是线程b和c之间的顺序就不确定了。

四、总结

至此，三种内存序模型都大概的梳理了一遍。可以看出，acquire-release模型是最复杂的。本文中对这些内存模型的描述可能不是那么的直白，也可能描述的不是那么到位，建议读者看看《C++ Concurrency In Action 2nd》这本书，里面作者也进行了详尽的描述，只不过他没有怎么涉及硬件层面的描述。关于硬件层面，那就多到知乎上找找相关的高并发、缓存一致性协议等相关文章或者专栏，然后结合本文，可能就会加深你对C++11内存模型的理解。

如果有读者对本文有什么不清楚的地方，欢迎留言，我们一起讨论，一起学习！！！

五、关于三种内存序的一些Tips

1. RWM操作中含有两种操作，一个load操作，另一个store操作。然而，RWM操作的函数只支持传入一个内存序形参来同时表示store操作和load操作(那些CAS操作能够传入两个内存序形参并不是分别给load操作和store操作用的，是分别对应操作失败和操作成功的)，因此，如果RWM函数中内存序传入的是acquire，consume内存序，则此内存序只能对RWM中的load部分有效，如果传入的是release，则只对RWM中的store有效，如果传入的是acq_rel或者seq_cst，则能同时对RWM中的load和store生效，并且其中的load对应的就是acquire，store对应的就是release；

2. 如果seq_cst与acquire或者release配对使用(即seq_cst修饰同一对同步点中的一个，acquire或者release修饰这一对同步点中的另一个)，则如果seq_cst修饰的是store操作，则此store操作相当于用的是release语义，如果seq_cst修饰的是load操作，则此load操作相当于用的是acquire语义；

3. 对于函数std::atomic_thread_fence()，如果传入release标记的话，此fence函数应该放在store函数的前面，如果传入的是acquire标记的话，此fence应该放在load函数的后面。此外，两个fence同步的话，同步点在fence处，因此fence函数只能**对处于fence两边的内存操作的顺序进行约束**，当内存操作都处于fence函数的同一边时，是起不到约束作用的；

4. 如果原子操作已经发生了happences-before关系，则fence函数并不影响其已经产生的关系；

5. 原子操作能阻止数据竞争，但是不能阻止条件竞争，数据竞争可能会产生未定义行为。关于数据竞争和条件竞争的区别，请看博文；

6. 注意，在使用原子变量之前，最好先使用is_lock_free()判断一下其内部是否是用锁实现的，如果是的，那么最好不要使用原子变量，而是直接就使用锁，因为锁更不容易出错，且容易维护。这种情况下原子操作的效率并不比锁的效率高。当然，最好是使用相关的宏，直接在编译期就能确定是不是，这样能写出效率更好的代码。此外，C++17的is_always_lock_free()函数也是编译期执行的，也可优先考虑使用；

7. 不要把std::atomic<>与那些可选名称如std::atomic_bool等混用，因为可能导致代码不可移植。尽量使用std::atomic<>而不是那些可选的替代类型(除非是在C接口中使用)；

8. 原子类型的那些复合赋值操作符(如+=)返回的是当前原子变量存储的新值，而这些操作符对应的成员函数版本(如fetch_add())返回的是原子变量的旧值。例外情况是，前置递增(或递减)运算符返回的是新值，后置递增(或递减)运算符返回的是旧值。注意，它们返回的全都不是引用，而是右值；

9. std::atomic_flag必须用ATOMIC_FLAG_INIT初始化，它将std::atomic_flag初始化为clear状态，test_and_set()能将其转成set状态，而clear()能将其恢复成clear状态；

10. 注意，在弱CAS的平台上(即平台不支持直接使用CPU指令来实现原子类型的比较并交换(compare-and-swap)操作)，compare_exchange_strong()函数可能内部是使用compare_exchange_weak()函数并加上一个循环来实现的。因此，如果用compare_exchange_strong()函数时要主动加一个循环，可以考虑看看能不能直接使用compare_exchange_weak()函数，这样或许能少加一层循环；

11. compare_exchange_xxx()函数可以传入两个内存序标记，一个是用于指定函数返回true时(表示进行了store动作)的内存序操作，一个是用于指定返回false时(表示未进行store动作)的内存序操作。第一个内存序形参是用于指定返回true时(函数执行成功)的内存序操作，第二个是用于返回false(函数执行失败)的。当传入两个内存序标记时，成功的内存序的限制级别必须大于等于失败的内存序限制级别，不能小于。由于失败时没有进行store操作，因此失败的内存序标记不能是release或者acq_rel，可以指定为其他任何类型，包括relaxed。此外，当然也可以只传入一个内存序形参，只传入一个的话，它是用于指定成功的内存序操作的。此时，失败的内存序会自动使用与成功相同的内存序标记，不过会把其上的release部分的语义给排除掉(也就是语义退化)，比如如果只传入一个release内存序用于成功，则失败的内存序(第二个内存序形参)会自动变成relaxed(将release退化成relaxed)，如果是acq_rel用于成功，则失败的自动变成acquire(将acq_rel退化成acquire)，如果两个都不指定，则成功和失败都统一使用seq_cst；

12. 整形原子变量没有乘法、除法和位移动操作符；

13. 如果想把std::atomic<>模板用于自定义类型(UDT)，则UDT不能含有虚函数，不能有虚基类，且赋值运算符必须是编译器合成的。此外，UDT的基类以及它的非静态数据成员也必须符合这些条件。这些条件会允许编译器使用memcpy或者等价的操作(也就是连续内存拷贝)来执行赋值操作。最后，UDT原子类型的compare_exchange操作必须可以用按位比较(就像使用memcmp()一样)，而不能有任何自定义的比较操作。如果UDT类型提供的比较操作有不同的语义，或者其中含有一些填充bit位不参与比较，则即使数值比较上是相等的，compare_exchange操作也会失败。综合看起

来，就UDT就像是C语言中的struct类似，就是存储一些数据的集合体，即使包含有内置类型数组都可以；

14. 注意，浮点原子类型在使用compare_exchange等函数时可能会出错，因为浮点数的表示形式可能不一样。此外，浮点原子类型没有任何算数运算操作(比如+=，-=等)；

15. C++标准特地为std::shared_ptr<>准备了全局原子操作函数的重载版本，所以在多线程之间使用std::shared_ptr<>时应该使用原子操作函数来操作。不过，没有std::shared_ptr<>的原子类型，其存储的类型仍然是std::shared_ptr<>(至少C++17标准中还没有)，但是在存储的时候或者获取的时候要调用相应的全局原子操作函数，例如：

```
// 智能指针，用于在线程之间传递数据
std::shared_ptr<my_data> p;

void update_global_data()
{
        std::shared_ptr<my_data> local(new my_data);

        // 1 必须使用全局原子写函数来存储
        std::atomic_store(&p, local);
}

void process_global_data()
{
        // 2 在使用的时候，也要先通过全局原子读函数将其转成普通智能指针，再使用
        std::shared_ptr<my_data> local = std::atomic_load(&p);
        process_data(local);
}
```

    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9
    - 10
    - 11
    - 12
    - 13
    - 14
    - 15
    - 16
    - 17
    - 18
    - 19

16. 一般来说，在同一个表达式中的多个操作之间的顺序是不确定的，但有些是确定的，比如逗号表达式，或者一个表达式的结果作为另一个表达式的参数时。

C++ 多线程与内存模型资料汇总