**多线程内存模型**

下面这篇文章对C++中的内存模型做了一个介绍，其思想完全使用于JMM，可以帮助理解JMM

内存模型描述了多线程通过共享内存进行交互的行为

#### 1. 顺序一致性模型（Sequential Consistency）

在介绍C++多线程模型之前，让我们先介绍一下最基本的顺序一致性模型。对多线程程序来说，最直观，最容易被理解的执行方式就是顺序一致性模型。顺序一致性的提出者Lamport给出的定义是：  
“… the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”  
从这个定义中我们可以看出，顺序一致性主要约定了两件事情：  
（1）从单个线程的角度来看，每个线程内部的指令都是按照程序规定的顺序（program order）来执行的；  
（2）从整个多线程程序的角度来看，整个多线程程序的执行顺序是按照某种交错顺序来执行的，且是全局一致的；

下面我们通过一个例子来理解顺序一致性。假设我们有两个线程（线程1和线程2），它们分别运行在两个CPU核上，有两个初始值为0的全局共享变量x和y，两个线程分别执行下面两条指令：  
初始条件： x = y = 0;

|  |  |
| --- | --- |
| 线程 1 | 线程 2 |
| x = 1; | y=1; |
| r1 = y; | r2 = x; |

因为多线程程序交错执行的顺序是不确定的，所以该程序可能有如下几种执行顺序：

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| 顺序 1 | 顺序 2 | 顺序 3 |
| x = 1;  r1 = y;  y = 1;  r2 = x;  结果:r1==0 and r2 == 1 | y = 1;  r2 = x;  x = 1;  r1 = y;  结果: r1 == 1 and r2 == 0 | x = 1;  y = 1;  r1 = y;  r2 = x;  结果: r1 == 1 and r2 == 1 |

顺序一致性模型的第一个约定要求每个线程内部的语句都是按照程序规定的顺序执行，例如，线程1里面的两条语句在该线程中一定是x=1先执行，r1=y后执行。顺序一致性的第二个约定要求多线程程序按照某种顺序执行，且所有线程看见的整体执行顺序必须一致，即该多线程程序可以按照顺序1、顺序2或者顺序3（以及其他可能的执行顺序）执行，且线程1和线程2所观察到的整个程序的执行顺序是一致的（例如，如果线程1“看见”整个程序的执行顺序是顺序 1，那么线程2“看见”的整个程序的执行顺序也必须是顺序1，而不能是顺序2或者顺序3）。依照顺序一致性模型，虽然这个程序还可能按其他的交错顺序执行，但是r1和r2的值却只可能出现上面三种结果，而不可能出现r1和r2同时为0的情况。

然而，尽管顺序一致性模型非常易于理解，但是它却对CPU和编译器的性能优化做出了很大的限制，所以常见的多核CPU和编译器大都没有实现顺序一致性模型。例如，编译器可能会为了隐藏一部分读操作的延迟而做如下优化，把线程1中对y的读操作（即r1=y）调换到x=1之前执行：

初始条件：x=y=0;

|  |  |
| --- | --- |
| 线程 1 | 线程 2 |
| r1 = y; | y=1; |
| x = 1; | r2 = x; |

在这种情况下，该程序如果按下面的顺序执行就可能就会出现r1和r2都为0这样的违反顺序一致性的结果：

|  |
| --- |
| 顺序 4 |
| r1 = y; y = 1; r2 = x; x = 1; |

那么为什么编译器会做这样的乱序优化呢？因为读一个在内存中而不是在cache中的共享变量需要较长的时钟周期，所以编译器就“自作聪明”的让读操作先执行，从而隐藏掉一些指令执行的延迟，从而提高程序的性能。实际上，这种优化是串行时代非常普遍的，因为它对单线程程序的语义是没有影响的。但是在进入多核时代后，编译器缺少语言级的内存模型的约束，导致其可能做出违法顺序一致性规定的多线程语义的错误优化。同样的，多核CPU中的写缓冲区（store buffer）也可能实施乱序优化：它会把要写入内存的值先在缓冲区中缓存起来，以便让该写操作之后的指令先执行，进而出现违反顺序一致性的执行顺序。

因为现有的多核CPU和编译器都没有遵守顺序一致模型，而且C/C++的现有标准中都没有把多线程考虑在内，所以给编写多线程程序带来了一些问题。例如，为了正确地用C++实现Double-Checked Locking，我们需要使用非常底层的内存栅栏（Memory Barrier）指令来显式地规定代码的内存顺序性（memory ordering）[5]。然而，这种方案依赖于具体的硬件，因此可移植性很差；而且它过于底层，不方便使用。

#### 2. C++多线程内存模型

为了更容易的进行多线程编程，程序员希望程序能按照顺序一致性模型执行；但是顺序一致性对性能的损失太大了，CPU和编译器为了提高性能就必须要做优化。为了在易编程性和性能间取得一个平衡，一个新的模型出炉了：sequential consistency for data race free programs，它就是即将到来的C++1x标准中多线程内存模型的基础。对C++程序员来说，随着C++1x标准的到来，我们终于可以依赖高级语言内建的多线程内存模型来编写正确的、高性能的多线程程序。

C++内存模型可以被看作是C++程序和计算机系统（包括编译器，多核CPU等可能对程序进行乱序优化的软硬件）之间的契约，它规定了多个线程访问同一个内存地址时的语义，以及某个线程对内存地址的更新何时能被其它线程看见。这个模型约定：没有数据竞跑的程序是遵循顺序一致性的。该模型的核心思想就是由程序员用同步原语（例如锁或者C++1x中新引入的atomic类型的共享变量）来保证你程序是没有数据竞跑的，这样CPU和编译器就会保证程序是按程序员所想的那样执行的（即顺序一致性）。换句话说，程序员只需要恰当地使用具有同步语义的指令来标记那些真正需要同步的变量和操作，就相当于告诉CPU和编译器不要对这些标记好的同步操作和变量做违反顺序一致性的优化，而其它未被标记的地方可以做原有的优化。编译器和CPU的大部分优化手段都可以继续实施，只是在同步原语处需要对优化做出相应的限制；而且程序员只需要保证正确地使用同步原语即可，因为它们最终表现出来的执行效果与顺序一致性模型一致。由此，C++多线程内存模型帮助我们在易编程性和性能之间取得了一个平衡。

在C++1x标准之前，C++是在建立在单线程语义上的。为了进行多线程编程，C++程序员通过使用诸如Pthreads，Windows Thread等C++语言标准之外的线程库来完成代码设计。以Pthreads为例，它提供了类似pthread\_mutex\_lock这样的函数来保证对共享变量的互斥访问，以防止数据竞跑。人们不禁会问，Pthreads这样的线程库我用的好好的，干嘛需要C++引入的多线程，这不是多此一举么？其实，以线程库的形式进行多线程编程在绝大多数应用场景下都是没有问题的。然而，线程库的解决方案也有其先天缺陷。第一，如果没有在编程语言中定义内存模型的话，我们就不能清楚的定义到底什么样的编译器/CPU优化是合法的，而程序员也不能确定程序到底会怎么样被优化执行。例如，Pthreads标准中并未对什么是数据竞跑（Data Race）做出精确定义，因此C++编译器可能会进行一些错误优化从而导致数据竞跑[3]。第二，绝大多数情况下线程库能正确的完成任务，而在极少数对性能有更高要求的情况下（尤其是需要利用底层的硬件特性来实现高性能Lock Free算法时）需要更精确的内存模型以规定好程序的行为。简而言之，把内存模型集成到编程语言中去是比线程库更好的选择。

#### 3. C++1x中引入的atomic类型

C++作为一种高性能的系统语言，其设计目标之一就在于提供足够底层的操作，以满足对高性能的需求。在这个前提之下，C++1x除了提供传统的锁、条件变量等同步机制之外，还引入了新的atomic类型。相对于传统的mutex锁来说，atomic类型更底层，具备更好的性能，因此能用于实现诸如Lock Free等高性能并行算法。有了atomic类型，C++程序员就不需要像原来一样使用汇编代码来实现高性能的多线程程序了。而且，把atomic类型集成到C++语言中之后，程序员就可以更容易地实现可移植的多线程程序，而不用再依赖那些平台相关的汇编语句或者线程库。

对常见的数据类型，C++1x都提供了与之相对应的atomic类型。以bool类型举例，与之相对应的atomic\_bool类型具备两个新属性：原子性与顺序性。顾名思义，原子性的意思是说atomic\_bool的操作都是不可分割的，原子的；而顺序性则指定了对该变量的操作何时对其他线程可见。在C++1x中，为了满足对性能的追求，atomic类型提供了三种顺序属性：sequential consistency ordering（即顺序一致性），acquire release ordering以及relaxed ordering。因为sequential consistency是最易理解的模型，所以默认情况下所有atomic类型的操作都会使sequential consistency顺序。当然，顺序一致性的性能相对来说比较差，所以程序员还可以使用对顺序性要求稍弱一些的acquire release ordering与最弱的relaxed ordering。