理解 C++ 的 Memory Order

2017-12-04 | 并发编程 |

为什么需要 Memory Order

如果不使用任何同步机制（例如 mutex 或 atomic），在多线程中读写同一个变量，那么，程序的结果是难以预料的。简单来说，编译器以及 CPU 的一些行为，会影响到程序的执行结果：

- 即使是简单的语句，C++ 也不保证是原子操作。
- CPU 可能会调整指令的执行顺序。
- 在 CPU cache 的影响下，一个 CPU 执行了某个指令，不会立即被其它 CPU 看见。

原子操作说的是，一个操作的状态要么就是未执行，要么就是已完成，不会看见中间状态。例如，在 C++11 中，下面程序的结果是未定义的：

```
1            int64_t i = 0;     // global variable
2
3  Thread-1:              Thread-2:
4  i = 100;               std::cout << i;
```

C++ 并不保证 `i = 100` 是原子操作，因为在某些 CPU Architecture 中，写入 `int64_t` 需要两个 CPU 指令，所以 Thread-2 可能会读取到 `i` 在赋值过程的中间状态。

另一方面，为了优化程序的执行性能，CPU 可能会调整指令的执行顺序。为阐述这一点，下面的例子中，让我们假设所有操作都是原子操作：

```
1            int x = 0;     // global variable
```

```
2            int y = 0;     // global variable
3
4  Thread-1:              Thread-2:
5  x = 100;               while (y != 200)
6  y = 200;                   ;
7                         std::cout << x;
```

如果 CPU 没有乱序执行指令，那么 Thread-2 将输出 `100` 。然而，对于 Thread-1 来说， `x = 100;` 和 `y = 200;` 这两个语句之间没有依赖关系，因此，Thread-1 允许调整语句的执行顺序：

```
1  Thread-1:
2  y = 200;
3  x = 100;
```

在这种情况下，Thread-2 将输出 `0` 或 `100` 。

CPU cache 也会影响到程序的行为。下面的例子中，假设从时间上来讲，A 操作先于 B 操作发生：

```
1            int x = 0;     // global variable
2
3  Thread-1:                      Thread-2:
4  x = 100;    // A               std::cout << x;    // B
```

尽管从时间上来讲，A 先于 B，但 CPU cache 的影响下，Thread-2 不能保证立即看到 A 操作的结果，所以 Thread-2 可能输出 `0` 或 `100` 。

从上面的三个例子可以看到，多线程读写同一变量需要使用同步机制，最常见的同步机制就是 `std::mutex` 和 `std::atomic` 。然而，从性能角度看，通常使用 `std::atomic` 会获得更好的性能。

C++11 为 `std::atomic` 提供了 4 种 memory ordering:

- Relaxed ordering
- Release-Acquire ordering
- Release-Consume ordering

- Sequentially-consistent ordering

默认情况下， `std::atomic` 使用的是 Sequentially-consistent ordering。但在某些场景下，合理使用其它三种 ordering，可以让编译器优化生成的代码，从而提高性能。

Relaxed ordering

在这种模型下， `std::atomic` 的 `load()` 和 `store()` 都要带上 `memory_order_relaxed` 参数。Relaxed ordering 仅仅保证 `load()` 和 `store()` 是原子操作，除此之外，不提供任何跨线程的同步。

先看看一个简单的例子：

```
                    std::atomic<int> x = 0;     // global variable
                    std::atomic<int> y = 0;     // global variable

Thread-1:                                       Thread-2:
r1 = y.load(memory_order_relaxed); // A        r2 = x.load(memory_order_relax
x.store(r1, memory_order_relaxed); // B        y.store(42, memory_order_relax
```

执行完上面的程序，可能出现 `r1 == r2 == 42` 。理解这一点并不难，因为编译器允许调整 C 和 D 的执行顺序。如果程序的执行顺序是 D -> A -> B -> C，那么就会出现 `r1 == r2 == 42` 。

如果某个操作只要求是原子操作，除此之外，不需要其它同步的保障，就可以使用 Relaxed ordering。程序计数器是一种典型的应用场景：

```
#include <cassert>
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> cnt = {0};

void f()
{
    for (int n = 0; n < 1000; ++n) {
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
```

```
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }

    assert(cnt == 10000);    // never failed

    return 0;
}
```

Release-Acquire ordering

在这种模型下， `store()` 使用 `memory_order_release` ，而 `load()` 使用 `memory_order_acquire` 。这种模型有两种效果，第一种是可以限制 CPU 指令的重排：

- 在 `store()` 之前的所有读写操作，不允许被移动到这个 `store()` 的后面。
- 在 `load()` 之后的所有读写操作，不允许被移动到这个 `load()` 的前面。

除此之外，还有另一种效果：假设 Thread-1 `store()` 的那个值，成功被 Thread-2 `load()` 到了，那么 Thread-1 在 `store()` 之前对内存的所有写入操作，此时对 Thread-2 来说，都是可见的。

下面的例子阐述了这种模型的原理：

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<bool> ready{ false };
int data = 0;

void producer()
{
    data = 100;                                     // A
    ready.store(true, std::memory_order_release);   // B
```

```
}

void consumer()
{
    while (!ready.load(std::memory_order_acquire))    // C
        ;
    assert(data == 100); // never failed              // D
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);

    t1.join();
    t2.join();

    return 0;
}
```

让我们分析一下这个过程：

- 首先 A 不允许被移动到 B 的后面。
- 同样 D 也不允许被移动到 C 的前面。
- 当 C 从 while 循环中退出了，说明 C 读取到了 B `store()` 的那个值，此时，Thread-2 保证能够看见 Thread-1 执行 B 之前的所有写入操作（也即是 A）。

参考资料

- C++ atomics and memory ordering
- cppreference.com - std::memory_order
- Atomic Usage examples
- C++11 introduced a standardized memory model. What does it mean?
- bRPC - Memory fence
- Acquire and Release Semantics

#Concurrency