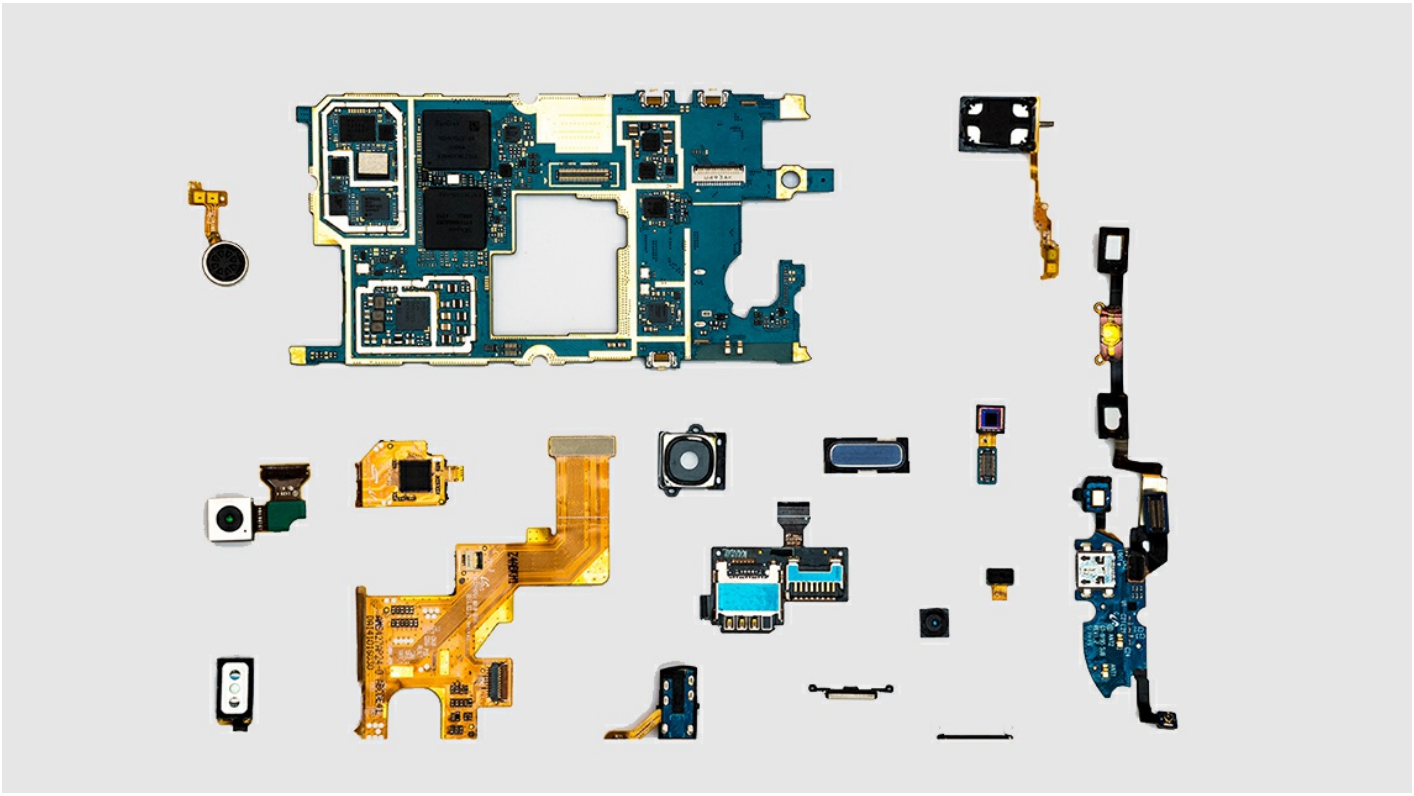
20讲内存模型和atomic：理解并发的复杂性



你好，我是吴咏炜。

上一讲我们讨论了一些并发编程的基本概念，今天我们来讨论一个略有点绕的问题，C++ 里的内存模型和原子量。

C++98 的执行顺序问题

C++98 的年代里，开发者们已经了解了线程的概念，但 C++ 的标准里则完全没有提到线程。从实践上，估计大家觉得不提线程，C++ 也一样能实现多线程的应用程序吧。不过，很多聪明人都忽略了，下面的事实可能会产生不符合直觉预期的结果：

- 为了优化的必要，编译器是可以调整代码的执行顺序的。唯一的要求是，程序的“可观测”外部行为是一致的。
- 处理器也会对代码的执行顺序进行调整（所谓的 CPU 乱序执行）。在单处理器的情况下，这种乱序无法被程序观察到；但在多处理器的情况下，在另外一个处理器上运行的另一个线程就可能会察觉到这种不同顺序的后果了。

对于上面的后一点，大部分开发者并没有意识到。原因有好几个方面：

- 多处理器的系统在那时还不常见
- 主流的 x86 体系架构仍保持着较严格的内存访问顺序
- 只有在数据竞争（data race）激烈的情况下才能看到“意外”的后果

举一个例子，假设我们有两个全局变量：

```
int x = 0;
int y = 0;
```

然后我们在一个线程里执行：

```
x = 1;
y = 2;
```

在另一个线程里执行：

```
if (y == 2) {
  x = 3;
  y = 4;
}
```

想一下，你认为上面的代码运行完之后，`x`、`y` 的数值有几种可能?

你如果认为有两种可能，1、2 和 3、4 的话，那说明你是按典型程序员的思维模式看问题的——没有像编译器和处理器一样处理问题。事实上，1、4 也是一种结果的可能。有两个基本的原因可以造成这一后果：

- 编译器没有义务一定按代码里给出的顺序产生代码。事实上，跟据上下文调整代码的执行顺序，使其最有利于处理器的架构，是优化中很重要的一步。就单个线程而言，先执行 `x = 1` 还是先执行 `y = 2` 完全是件无关紧要的事：它们没有外部“可观察”的区别。
- 在多处理器架构中，各个处理器可能存在缓存不一致性问题。取决于具体的处理器类型、缓存策略和变量地址，对变量 `y` 的写入有可能先反映到主内存中去。之所以这个问题似乎并不常见，是因为常见的 x86 和 x86-64 处理器是在顺序执行方面做得最保守的——大部分其他处理器，如 ARM、DEC Alpha、PA-RISC、IBM Power、IBM z/架构和 Intel Itanium 在内存序问题上都比较“松散”。x86 使用的内存模型基本提供了顺序一致性（sequential consistency）；相对的，ARM 使用的内存模型就只是松散一致性（relaxed consistency）。较为严格的描述，请查看参考资料 [1] 和里面提供的进一步资料。

虽说 Intel 架构处理器的顺序一致性比较好，但在多处理器（包括多核）的情况下仍然能够出现写读序列变成读写序列的情况，产生意料之外的后果。参考资料 [2] 中提供了完整的例子，包括示例代码。对于缓存不一致性问题的一般中文介绍，可以查看参考资料 [3]。

双重检查锁定

在多线程可能对同一个单件进行初始化的情况下，有一个双重检查锁定的技巧，可基本示意如下：

```
// 头文件
class singleton {
public:
  static singleton* instance();
  …
private:
  static singleton* inst_ptr_;
};

// 实现文件
singleton* singleton::inst_ptr_ =
  nullptr;

singleton* singleton::instance()
{
  if (inst_ptr_ == nullptr) {
    lock_guard lock;  // 加锁
    if (inst_ptr_ == nullptr) {
      inst_ptr_ = new singleton();
    }
  }
  return inst_ptr_;
}
```

这个代码的目的是消除大部分执行路径上的加锁开销。原本的意图是：如果 `inst_ptr_` 没有被初始化，执行才会进入加锁的路径，防止单件被构造多次；如果 `inst_ptr_` 已经被初始化，那它就会被直接返回，不会产生额外的开销。虽然看上去很美，但它一样有着上面提到的问题。Scott Meyers 和 Andrei Alexandrecu 详尽地分析了这个用法 [4]，然后得出结论：即使花上再大的力气，这个用法仍然有着非常多的难以填补的漏洞。本质上还是上面说的，优化编译器会努力击败你试图想防止优化的努力，而多处理器会以令人意外的方式让代码走到错误的执行路径上去。他们分析得非常详细，建议你可以花时间学习一下。

volatile

在某些编译器里，使用 `volatile` 关键字可以达到内存同步的效果。但我们必须记住，这不是 `volatile` 的设计意图，也不能通用地达到内存同步的效果。`volatile` 的语义只是防止编译器“优化”掉对内存的读写而已。它的合适用法，目前主要是用来读写映射到内存地址上的 I/O 操作。

由于 `volatile` 不能在多处理器的环境下确保多个线程能看到同样顺序的数据变化，在今天的通用应用程序中，不应该再看到 `volatile` 的出现。

C++11 的内存模型

为了从根本上消除这些漏洞，C++11 里引入了适合多线程的内存模型。我们可以在参考资料 [5] 里了解更多的细节。跟我们开发密切相关的是：现在我们有了原子对象（atomic）和使用原子对象的获得（acquire）、释放（release）语义，可以真正精确地控制内存访问的顺序性，保证我们需要的内存序。

内存屏障和获得、释放语义

拿刚才的那个例子来说，如果我们希望结果只能是 1、2 或 3、4，即满足程序员心中的完全存储序（total store ordering），我们需要在 `x = 1` 和 `y = 2` 两句语句之间加入内存屏障，禁止这两句语句交换顺序。我们在此种情况下最常用的两个概念是“获得”和“释放”：

- **获得**是一个对内存的**读**操作，当前线程的任何后面的读写操作都不允许重排到这个操作的**前面**去。
- **释放**是一个对内存的**写**操作，当前线程的任何前面的读写操作都不允许重排到这个操作的**后面**去。

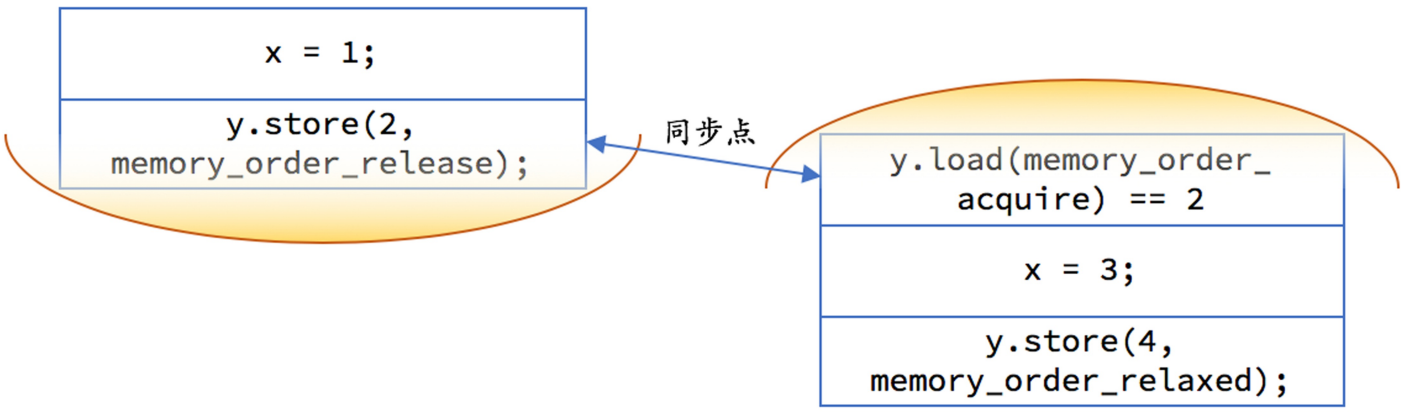
具体到我们上面的第一个例子，我们需要把 `y` 声明成 `atomic<int>`。然后，我们在线程 1 需要使用释放语义：

```
x = 1;
y.store(2, memory_order_release);
```

在线程 2 我们对 `y` 的读取应当使用获得语义，但存储只需要松散内存序即可：

```
if (y.load(memory_order_acquire) ==
    2) {
  x = 3;
  y.store(4, memory_order_relaxed);
}
```

我们可以用下图示意一下，每一边的代码都不允许重排越过黄色区域，且如果 `y` 上的释放早于 `y` 上的获取的话，释放前对内存的修改都在另一个线程的获取操作后可见：



事实上，在我们把 `y` 改成 `atomic<int>` 之后，两个线程的代码一行不改，执行结果都会是符合我们的期望的。因为 `atomic` 变量的写操作缺省就是释放语义，读操作缺省就是获得语义。即

- `y = 2` 相当于 `y.store(2, memory_order_release)`
- `y == 2` 相当于 `y.load(memory_order_acquire) == 2`

但是，缺省行为可能是对性能不利的：我们并不需要在任何情况下都保证操作的顺序性。

另外，我们应当注意一下，acquire 和 release 通常都是配对出现的，目的是保证如果对同一个原子对象的 release 发生在 acquire 之前的话，release 之前发生的内存修改能够被 acquire 之后的内存读取全部看到。

atomic

刚才是对 atomic 用法的一个非正式介绍。下面我们对 atomic 做一个稍完整些的说明（更完整的见 [6]）。

C++11 在 <atomic> 头文件中引入了 `atomic` 模板，对原子对象进行了封装。我们可以将其应用到任何类型上去。当然对于不同的类型效果还是有所不同的：对于整型量和指针等简单类型，通常结果是无锁的原子对象；而对于另外一些类型，比如 64 位机器上大小不是 1、2、4、8（有些平台/编译器也支持对更大的数据进行无锁原子操作）的类型，编译器会自动为这些原子对象的操作加上锁。编译器提供了一个原子对象的成员函数 `is_lock_free`，可以检查这个原子对象上的操作是否是无锁的。

原子操作有三类：

- 读：在读取的过程中，读取位置的内容不会发生任何变动。
- 写：在写入的过程中，其他执行线程不会看到部分写入的结果。
- 读-修改-写：读取内存、修改数值、然后写回内存，整个操作的过程中间不会有其他写入操作插入，其他执行线程不会看到部分写入的结果。

<atomic> 头文件中还定义了内存序，分别是：

- `memory_order_relaxed`：松散内存序，只用来保证对原子对象的操作是原子的
- `memory_order_consume`：目前不鼓励使用，我就不说明了
- `memory_order_acquire`：获得操作，在读取某原子对象时，当前线程的任何后面的读写操作都不允许重排到这个操作的前面去，并且其他线程在对同一个原子对象释放之前的所有内存写入都在当前线程可见
- `memory_order_release`：释放操作，在写入某原子对象时，当前线程的任何前面的读写操作都不允许重排到这个操作的后面去，并且当前线程的所有内存写入都在对同一个原子对象进行获取的其他线程可见
- `memory_order_acq_rel`：获得释放操作，一个读-修改-写操作同时具有获得语义和释放语义，即它前后的任何读写操作都不允许重排，并且其他线程在对同一个原子对象释放之前的所有内存写入都在当前线程可见，当前线程的所有内存写入都在对同一个原子对象进行获取的其他线程可见
- `memory_order_seq_cst`：顺序一致性语义，对于读操作相当于获取，对于写操作相当于释放，对于读-修改-写操作相当于获得释放，**是所有原子操作的默认内存序**

`atomic` 有下面这些常用的成员函数：

- 默认构造函数（只支持零初始化）
- 拷贝构造函数被删除
- 使用内置对象类型的构造函数（不是原子操作）
- 可以从内置对象类型赋值到原子对象（相当于 `store`）
- 可以从原子对象隐式转换成内置对象（相当于 `load`）
- `store`，写入对象到原子对象里，第二个可选参数是内存序类型
- `load`，从原子对象读取内置对象，有个可选参数是内存序类型
- `is_lock_free`，判断对原子对象的操作是否无锁（是否可以用处理器的指令直接完成原子操作）
- `exchange`，交换操作，第二个可选参数是内存序类型（这是读-修改-写操作）
- `compare_exchange_weak` 和 `compare_exchange_strong`，两个比较加交换（CAS）的版本，你可以分别指定成功和失败时的内存序，也可以只指定一个，或使用默认的最安全内存序（这是读-修改-写操作）
- `fetch_add` 和 `fetch_sub`，仅对整数和指针内置对象有效，对目标原子对象执行加或减操作，返回其原始值，第二个可选参数是内存序类型（这是读-修改-写操作）
- `++` 和 `--`（前置和后置），仅对整数和指针内置对象有效，对目标原子对象执行增一或减一，操作使用顺序一致性语义，并注意返回的不是原子对象的引用（这是读-修改-写操作）

- += 和 -=，仅对整数和指针内置对象有效，对目标原子对象执行加或减操作，返回操作之后的数值，操作使用顺序一致性语义，并注意返回的不是原子对象的引用（这是读-修改-写操作）

有了原子对象之后，我们可以轻而易举地把 [第 2 讲] 中的 shared_count 变成线程安全。我们只需要包含 <atomic> 头文件，并把下面这行

```
long count_;
```

修改成

```
std::atomic_long count_;
```

即可（atomic_long 是 atomic<long> 的类型别名）。不过，由于我们并不需要 ++ 之后计数值影响其他行为，在 add_count 中执行简单的 ++、使用顺序一致性语义略有浪费。更好的做法是将其实现成：

```
void add_count() noexcept
{
  count_.fetch_add(
    1, std::memory_order_relaxed);
}
```

is_lock_free 的可能问题

注意，macOS 上在使用 Clang 时似乎不支持对需要加锁的对象使用 is_lock_free 成员函数，此时链接会出错。而 GCC 在这种情况下，需要确保系统上装了 libatomic。以 CentOS 7 下的 GCC 7 为例，我们可以使用下面的语句来安装：

```
sudo yum install devtoolset-7-libatomic-devel
```

然后，用下面的语句编译可以通过：

```
g++ -pthread test.cpp -latomic
```

Windows 下使用 MSVC 则没有问题。

mutex

上一讲我们已经讨论了互斥量。今天，我们只需要补充两点：

- 互斥量的加锁操作（lock）具有获得语义
- 互斥量的解锁操作（unlock）具有释放语义

有了目前讲过的这些知识，我们终于可以实现一个真正安全的双重检查锁定了：

```
// 头文件
class singleton {
public:
  static singleton* instance();
  …
private:
  static mutex lock_;
  static atomic<singleton*>
    inst_ptr_;
};

// 实现文件
mutex singleton::lock_;
atomic<singleton*>
  singleton::inst_ptr_;

singleton* singleton::instance()
{
  singleton* ptr = inst_ptr_.load(
    memory_order_acquire);
  if (ptr == nullptr) {
    lock_guard<mutex> guard{lock_};
    ptr = inst_ptr_.load(
      memory_order_relaxed);
    if (ptr == nullptr) {
      ptr = new singleton();
      inst_ptr_.store(
        ptr, memory_order_release);
    }
  }
  return inst_ptr_;
}
```

有个小地方注意一下：为了和 inst_ptr_.load 语句对称，我在 inst_ptr_.store 时使用了释放语义；不过，由于互斥量解锁本身具有释放语义，这么做并不是必需的。

并发队列的接口

在结束这一讲之前，我们来检查一下并发对编程接口的冲击。回想我们之前讲到标准库里 queue 有下面这样的接口：

```
template <typename T>
class queue {
public:
  …
  T& front();
  const T& front() const;
  void pop();
  …
}
```

我们之前还问过为什么 pop 不直接返回第一个元素。可到了并发的年代，我们不禁要问，这样的接口设计到底明智吗?

会不会在我们正在访问 `front()` 的时候，这个元素就被 pop 掉了?

事实上，上面这样的接口是不可能做到并发安全的。并发安全的接口大概长下面这个样子：

```
template <typename T>
class queue {
public:
  …
  void wait_and_pop(T& dest)
  bool try_pop(T& dest);
  …
}
```

换句话说，要准备好位置去接收；然后如果接收成功了，才安安静静地在自己的线程里处理已经被弹出队列的对象。接收方式还得分两种，阻塞式的和非阻塞式的……

那我为什么要在内存模型和原子量这一讲里讨论这个问题呢？因为并发队列的实现，经常是用原子量来达到无锁和高性能的。单生产者、单消费者的并发队列，用原子量和获得、释放语义就能简单实现。对于多生产者或多消费者的情况，那实现就比较复杂了，一般会使用 `compare_exchange_strong` 或 `compare_exchange_weak`。讨论这个话题的复杂性，就大大超出了本专栏的范围了。你如果感兴趣的话，可以查看下面几项内容：

- nvwa::fc_queue [7] 给出了一个单生产者、单消费者的无锁并发定长环形队列，代码长度是几百行的量级。
- moodycamel::ConcurrentQueue [8] 给出了一个多生产者、多消费者的无锁通用并发队列，代码长度是几千行的量级。
- 陈皓给出了一篇很棒的对无锁队列的中文描述 [9]，推荐阅读。

内容小结

在这一讲里，我们讨论了 C++ 对并发的底层支持，特别是内存模型和原子量。这些底层概念，是在 C++ 里写出高性能并发代码的基础。

课后思考

在传统 PC 上开发的程序员，应当比较少接触具有松散或弱内存一致性的系统，但原子量和普通变量的区别还是很容易在代码中表现出来的。请你尝试一下多个线程对一个原子量和一个普通全局变量做多次增一操作，观察最后的结果。

在 Intel 处理器架构上，唯一可见的重排是多处理器下的写读操作。大力推荐你尝试一下参考资料 [2] 中的例子（Windows 和 Linux 下可直接运行；macOS 下需要使用我的修改版本或备用下载链接来覆盖下载代码中的 gcc/ordering.cpp），并修改预定义宏。另外一种改法就是把代码中的 `X`、`Y` 的类型改成 `atomic_int`，重排也就消失了。

如果遇到任何特别问题，欢迎留言与我交流。

参考资料

[1] Wikipedia, “Memory ordering”. https://en.wikipedia.org/wiki/Memory_ordering

[1a] 维基百科, “内存排序”. https://zh.wikipedia.org/zh-cn/内存排序

[2] Jeff Preshing, “Memory reordering caught in the act”. https://preshing.com/20120515/memory-reordering-caught-in-the-act/

[3] 王欢明, 《多处理器编程：从缓存一致性到内存模型》. https://zhuanlan.zhihu.com/p/35386457

[4] Scott Meyers and Andrei Alexandrescu, “C++ and the perils of double-checked locking”. https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

[5] cppreference.com, “Memory model”. https://en.cppreference.com/w/cpp/language/memory_model

[5a] cppreference.com, “内存模型”. https://zh.cppreference.com/w/cpp/language/memory_model

[6] cppreference.com, “std::atomic”. https://en.cppreference.com/w/cpp/atomic/atomic

[6a] cppreference.com, “std::atomic”. https://zh.cppreference.com/w/cpp/atomic/atomic

[7] 吴咏炜, nvwa. https://github.com/adah1972/nvwa

[8] Cameron Desrochers, moodycamel::ConcurrentQueue. https://github.com/cameron314/concurrentqueue

[9] 陈皓, 《无锁队列的实现》. https://coolshell.cn/articles/8239.html

精选留言

prowu

吴老师，您好！有两个问题请帮忙解答下：

1、在解释相关memory_order_acquire, memory_order_release等时，都有提到“当前线程可见”，这个“可见”该怎么理解？

2、可以帮忙总结下，在什么场景下需要保证内存序，比如：满足了以下条件，就需要考虑是否保证内存序了：

（1）多线程环境下

（2）存在多个变量是可多个线程共享的，比如：类成员变量、全局变量

（3）这多个共享变量在实现逻辑上存在相互依赖的关系

（4）...

谢谢！

2020-01-14 08:56

作者回复

1. “可见”，可以理解成获得和释放操作的两个线程能观察到相同的内存修改结果。

2. 原则上任何多线程访问的变量应该要么是原子量，要么有互斥量来保护，这样最安全。特别要考虑内存序的，当然就是有多个有逻辑相关性的共享变量了。对于单个的变量，比如检查线程是否应该退出的布尔变量，只要消除了编译器优化，不需要保证访问顺序也可以正常工作；这样原子量可以使用 relaxed 的访问方式。

2020-01-14 09:44

木瓜777

您好，看了这篇后，对互斥量和原子量的使用 有些不明白，什么时候应该用互斥量，什么时候用原子量，什么时候一起使用?

2020-01-12 10:15

作者回复

用原子量的地方，粗想一下，你用锁都可以。但如果锁导致阻塞的话，性能比起原子量那是会有好几个数量级的差异了。锁即使不导致阻塞，性能也会比原子量低——锁本身的实现就会用到原子量，是个复杂的复合操作。

反过来不成立，用互斥量的地方不能都改用原子量。原子量本身没有阻塞机制，没有保护代码段的功能。

2020-01-13 14:37

tt

感觉这里的无锁操作就像分布式系统里面谈到的乐观锁，普通的互斥量就像悲观锁。只是CPU级的乐观锁由CPU提供指令集级别的支持。

内存重排会引起内存数据的不一致性，尤其是在多CPU的系统里。这又让我想起分布式系统里讲的CAP理论。

多线程就像分布式系统里的多个节点，每个CPU对自己缓存的写操作在CPU同步之前就造成了主内存中数据的值在每个CPU缓存中的不一致，相当于分布式系统中的分区。

我大概看了参考文献一眼，因为一级缓存相对主内存速度有数量级上的优势，所以各个缓存选择的策略相当于分布式系统中的可用性，即保留了AP（分区容错性与可用性，放弃数据的一致性），然后在涉及到缓存数据一致性问题上，相当于采取了最终一致性。

其实我觉得不论是什么系统，时间颗足够小的话，都会存在数据的不一致，只是CPU的速度太快了，所以看起来都是最终一致性。在保证可用性的时候，整个程序的某个变量或内存中的值看起来就是进行了重排。

分布式系统中将多个节点解耦的方式是用异步、用对列。生产者把变化事件写到对列里就返回，然后由消费者取出来异步的实施这些操作，达到数据的最终一致性。

看资料里，多CPU同步时，也有在CPU之间引入对列。当需要“释放前对内存的修改都在另一个线程的获取操作后可见”时，我的理解就是用了所谓的“内存屏障”强制让消费者消费完对列里的"CPU级的事物"。所以才会在达到严格内存序的过程中降低了程序的性能。

也许，这个和操作系统在调度线程时，过多的上下文切换会导致系统性能降低有关系。

2020-01-10 23:07

作者回复

思考得挺深入，很好。

操作系统的上下文切换和内存序的关系我略有不同意见。内存屏障的开销我查下来大概是 100、200 个时钟周期，也就是约 50 纳秒左右吧。而 Linux 的上下文切换开销约在 1 微秒多，也就是两者之前的性能差异超过 20 倍。因此，内存屏障不太可能是上下文切换性能开销的主因。

上下文切换实际需要做的事情非常多，那应该才是主要原因。

2020-01-11 17:23

禾桃

和大家分享一个链接

操作系统中锁的实现原理

https://mp.weixin.qq.com/s/6MRi_UEcMybKn4YXi6qWng

2020-01-14 22:26

作者回复

这篇太简单了，基本上只是覆盖尝试加锁这一步（大致是 compare_exchange_strong）。而且，现代操作系统上谁会用关中断啊。

最关键的是，一个线程在加锁失败时会发生什么。操作系统会挂起这个线程，并在锁释放时可能会重新唤起这个线程。文中完全没有提这个。

2020-01-15 22:26

禾桃

is_lock_free，判断对原子对象的操作是否无锁（是否可以用处理器的指令直接完成原子操作）

#1
这里的处理器的指令指的是，
“lock cmpxchg”?

#2
“是否可以用处理器的指令直接完成原子操作”, 这里的直接指的是仅使用“处理器的指令吗?

#3
能麻烦给个is_not_lock_free的对原子对象的操作的大概什么样子吗?

谢谢!

2020-01-12 21:03

作者回复

#1

不一定。比如，对于 store，生成可能就只是 mov 指令加个 mfence。

#2

是。

#3

你可以对比一下编译器生成的汇编代码：

https://godbolt.org/z/UHsDRj

2020-01-13 13:57

花晨少年

这一节讲的实在是太好了，我对前几节的编译器模版相关的不是很感冒，要是能把这期更深入的细节探讨一下，多做几节，就更好了。

```
singleton* singleton::instance()
{
@a
if (inst_ptr_ == nullptr) {//@1
@b
lock_guard lock; // 加锁
```

```
if (inst_ptr_ == nullptr) {
@c
inst_ptr_ = new singleton();//@2
@d
}
}
return inst_ptr_;
}
```

有个问题，就是对double check那个例子的疑惑，会出现什么问题?
inst_ptr_应该就两种状态，null和非null。
如果线程1在@b处，等待锁，这个时候线程2不管在@c或者@d处，线程a获得锁的时候，都不会进入@c，因为inst_ptr已经非空。
如果线程1在@a处，线程2在@2处，执行new操作，难道@2这个语句有什么问题吗，难道@2不是一个原子操作，会导致线程1已经得到线程2分配的对象地址，而内存还没有准备好吗？如果是这种情况的话，
那么下面加入了原子操作后，也没有解决new问题啊，

```
singleton* singleton::instance()
{
singleton* ptr = inst_ptr_.load(
memory_order_acquire);
if (ptr == nullptr) {
lock_guard<mutex> guard{lock_};
ptr = inst_ptr_.load(
memory_order_relaxed);
if (ptr == nullptr) {
ptr = new singleton();
inst_ptr_.store(
ptr, memory_order_release);
}
}
return inst_ptr_;
}
```

2020-01-12 20:17

作者回复

看参考资料4吧。如果嫌太长，就只看代码，编译器和处理器眼里允许重排成的样子。

简单说，就是赋值顺序的问题。至少在某些处理器上，其他线程可能先看到 inst_ptr_ 被修改，再看到单件的构造完成。

2020-01-13 14:26

陈志恒

专栏里面的评论都满地是宝，这就是比啃书本强太多的地方，大家可以讨论请教。文章需要复习，评论也同样需要复习，看看是否有了新的想法 。

在阅读的时候，我心里也有前面几个读者的关于锁、互斥量、原子操作的区别与联系的疑问 。

我尝试说一下我的理解：站在需求的角度
1.对单独没有逻辑联系的变量，直接使用原子量的relaxed就够了，没必要加上内存序
2.对于有联系的多个多线程中的变量，这时就需要考虑使用原子量的内存序
3.对于代码段的保护，由于原子量没有阻塞，所以必须使用互斥量和锁来解决
ps：互斥量+锁的操作 可取代 原子量。反之不可。

另外，还产生新的疑问：
1.互斥量的定义中，一个互斥量只允许在多线程中加一把锁，那么是否可以说互斥量只有和锁配合达到保护代码段的作用，互斥量还有其他单独的用法吗?
2.更近一步，原子量+锁，是否可以完成对代码段的保护？而吴老师也在评论区里提到：锁是由原子量构成的。

望老师解答，纠正。

2020-02-05 12:41

作者回复

你从需求方面理解的 1、2、3 我觉得都对，很好！

“互斥量只有和锁配合”这个提法我觉得很怪：互斥量是个对象，（加/解）锁是互斥量支持的动作——如果你指 lock_guard 之类的类，那是辅助的 RAII 对象，目的只是自动化互斥量上的对应操作而已。

你可能是被“操作系统中锁的实现原理”这样的提法带偏了。没有作为名字的专门锁对象，只有互斥量、条件变量、原子量。我也被带偏了，我在某个评论里说“锁”的时候，指的就是互斥量加锁。

2020-02-05 23:37

花晨少年

https://en.cppreference.com/w/cpp/atomic/memory_order最后一段讲解
memory_order_seq_cst提到，如果要保证最后的断言"assert(z.load() != 0);"不会发生，必须使用memory_order_seq_cst，这里很不理解。
下面是代码

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

std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};

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

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

void read_x_then_y()
{
while (!x.load(std::memory_order_seq_cst))//@1
;
if (y.load(std::memory_order_seq_cst)) {//@2
++z;
}
}
```

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

int main()
{
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); // will never happen
}
```

把代码全部改成memory_order_acq_rel操作为什么不可以?
按照memory_order_acq_rel的描述，在其他线程中,@2的所有操作应该都不会被重排到@1之前，
@4的操作也不会被重排到@3之前，
那如果是这样的话，也能确保断言永远不会发生。

2020-01-13 10:52

作者回复

memory_order_seq_cst 不是拿来和 memory_order_acq_rel 对比的，而是和 memory_order_relaxed 对比的。正如我在另外一个回答里说的，这里使用 memory_order_acq_rel 可能是非法的。比如 load，只能使用 relaxed、acquire 和 seq_cst，并且后两者是等价的。

2020-01-13 14:30

禾桃

```
void add_count() noexcept
{
count_.fetch_add(
1, std::memory_order_relaxed);
}

void add_count() noexcept
{
count_.fetch_add(
1, std::memory_order_seq_cst);
}
```

std::memory_order_seq_cst 比std::memory_order_relaxed，
性能方面的浪费，具体指的是什么?

谢谢!

2020-01-12 20:24

作者回复

好问题。这个问题我之前没细究，但现在仔细一看，常见架构上内存序参数对 fetch_add 是没影响的……似乎读-修改-写操作里，一般都是实现成顺序一致的。

也有例外，如 Power、Raspbian Buster、RISC-V：

https://godbolt.org/z/Du85RX

2020-01-13 14:11

花晨少年

介绍memory_order_seq_cst时，说这是所有原子操作的默认内存序，但是在文章前面又说

y = 2 相当于 y.store(2, memory_order_release)
y == 2 相当于 y.load(memory_order_acquire) == 2
?
有点凌乱，这里。

2020-01-12 19:53

作者回复

别漏了前面那几句：

「`memory_order_seq_cst`：顺序一致性语义，对于读操作相当于获取，对于写操作相当于释放」

2020-01-13 09:56

花晨少年

memory_order_acq_rel只能作用到读取-修改-写操作吗，貌似单纯的读或者写操作也可以用这个order.
那这个order和seq_cst貌似并没有很大的区别，
不明白这两个order的不止区别是什

2020-01-12 19:48

作者回复

按标准的规定，store 只能用 relaxed、release 或 seq_cst，load 只能用 relaxed、acquire 或 seq_cst，等等。其他组合在标准中明确说是未定义行为，就算能过也有点凑巧，不保证换个编译器或甚至换个版本还能继续工作。

不要这么做。

2020-01-13 14:16

李亮亮

C++真是博大精深

2020-01-11 18:42

作者回复

计算的世界真是复杂。C++是为了性能，让你能够看到这些复杂性而已。对性能没那么关注的，可以把这些复杂性隐藏掉。

2020-01-12 09:55

禾桃

Preshing

“In particular, each processor is allowed to delay the effect of a store past any load from a different location. “

这里的”delay”指的是1已经被写到X_cpu_cache, 但是还没有没到推送到X_memeory?

#1
X = 1;
asm volatile("" ::: "memory"); // Prevent memory reordering
r1 = Y;

上面的代码,能确保cpu会先执行store，（至少先写到X_cpu_cache,无法保证1被推送到X_memory)，然后再read?

#2
X = 1;
asm volatile("mfence" ::: "memory");
r1 = Y;

上面的代码,能确保cpu会先执行store（包括把1写到X_cpu_cache，再推送至X_memoery), 然后再read?

上面的代码，cpu 执行到mfence时，会确保1从X_cpu_cache推送到X_memory, 然后再去读Y?

谢谢!

2020-01-10 23:01

作者回复

delay部分和第二个问题的回答是“是”。

第一个问题你这么说似乎也对，但这个asm语句的主要目的是防止编译器做出任何重排，而没有对处理器提出要求。结果是会跟你说的一样。

2020-01-11 13:48