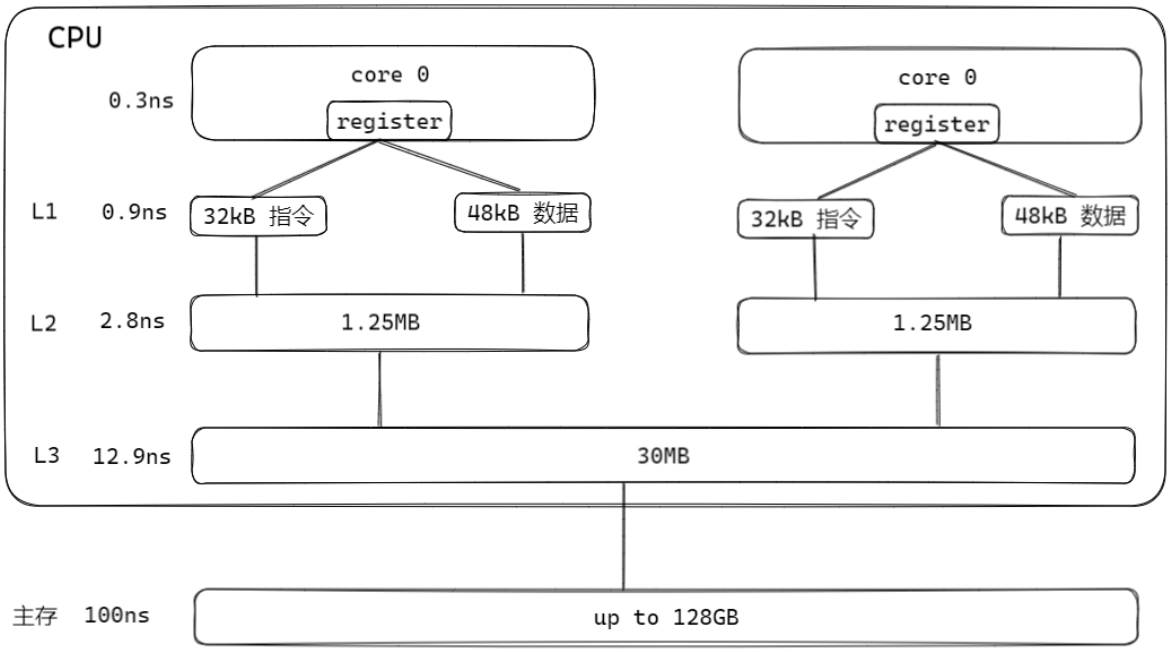
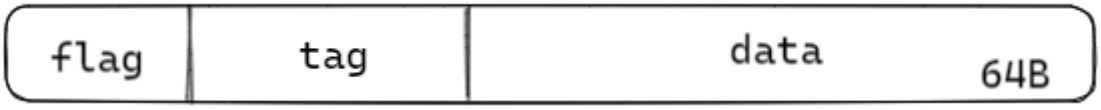
零声教育出品 Mark 老师 QQ:2548898954

存储体系结构

cpu 缓存



cache line



缓存一致性协议 MESI

> MESI 协议是一个基于失效的缓存一致性协议，支持 write-back 写回缓存的常用协议。
>
> 主要原理：通过总线嗅探策略（将读写请求通过总线广播给所有核心，核心根据本地状态进行响应）

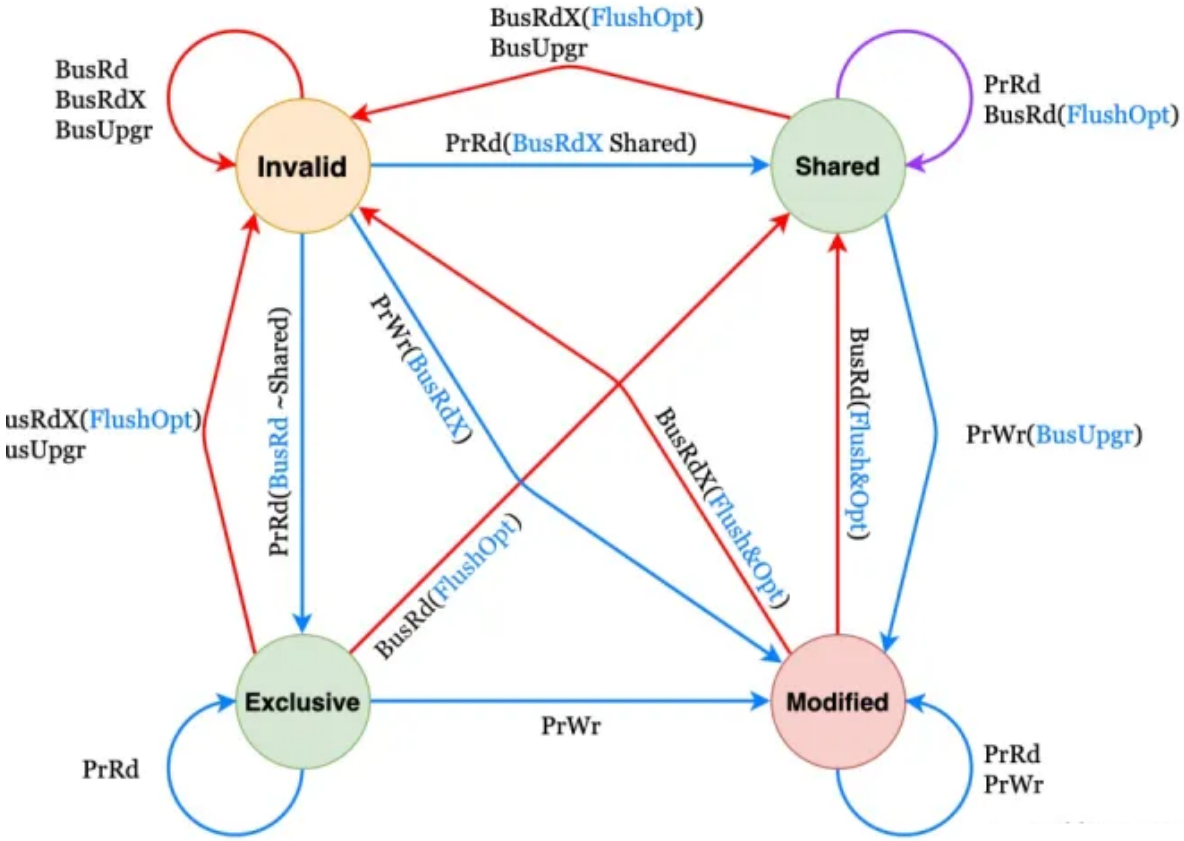
状态

- Modified（M）：某数据已修改但是没有同步到内存中。如果其他核心要读该数据，需要将该数据从缓存同步到内存中，并将状态转为 S。
- Exclusive（E）：某数据只在该核心当中，此时缓存和内存中的数据一致。
- Shared（S）：某数据在多个核心中，此时缓存和内存中的数据一致。
- Invaliddate（I）：某数据在该核心中以失效，不是最新数据。

事件

- **PrRd**：核心请求从换存块中读出数据；
- **PrWr**：核心请求向缓存块写入数据；
- **BusRd**：总线嗅探器收到来自其他核心的读出缓存请求；
- **BusRdX**：总线嗅探器收到另一核心写一个其不拥有的缓存块的请求；
- **BusUpgr**：总线嗅探器收到另一核心写一个其拥有的缓存块的请求；
- **Flush**：总线嗅探器收到另一核心把一个缓存块写回到主存的请求；
- **FlushOpt**：总线嗅探器收到一个缓存块被放置在总线以提供给另一核心的请求，和 Flush 类似，但只不过是从缓存到缓存的传输请求。

状态机


BusRdX(FlushOpt)
BusUpgr
BusRd
BusRdX
BusUpgr
PrRd
BusRd(FlushOpt)
Invalid
PrRd(BusRdX Shared)
Shared
usRdX(FlushOpt)
usUpgr
PrRd(BusRd ~Shared)
PrWr(BusRdX)
BusRdX(Flush&Opt)
BusRd(Flush&Opt)
PrWr(BusUpgr)
BusRd(FlushOpt)
Exclusive
PrWr
Modified
PrRd
PrRd
PrWr

状态机详解

当前状态	事件	响应
M	PrRd	• 无总线事务生成 • 状态保持不变 • 读操作为缓存命中
	PrWr	• 无总线事务生成 • 状态保持不变 • 写操作为缓存命中
	BusRd	• 状态变为共享(S)Shared • 发出总线FlushOpt信号并发出块的内容，接收者为最初发出BusRd的缓存与主存控制器（回写主存）
	BusRdX	• 状态变为无效(I)Invalid • 发出总线FlushOpt信号并发出块的内容，接收者为最初发出BusRd的缓存与主存控制器（回写主存）
E	PrRd	• 无总线事务生成 • 状态保持不变 • 读操作为缓存命中
	PrWr	• 无总线事务生成 • 状态变为已修改(M)Modified • 向缓存块中写入修改后的值
	BusRd	• 状态变为共享(S)Shared • 发出总线FlushOpt信号并发出块的内容
	BusRdX	• 状态变为无效 • 发出总线FlushOpt信号并发出块的内容
S	PrRd	• 无总线事务生成 • 状态保持不变 • 读操作为缓存命中
	PrWr	• 发出总线事务BusUpgr信号 • 状态转换为已修改(M)Modified • 其他缓存看到BusUpgr总线信号，标记其副本为为无效(I)Invalid
	BusRd	• 状态变为共享(S)Shared • 可能发出总线FlushOpt信号并发出块的内容（设计时决定那个共享的缓存发出数据）
	BusRdX	• 状态变为无效(I)Invalid • 可能发出总线FlushOpt信号并发出块的内容（设计时决定那个共享的缓存发出数据）
I	PrRd	• 给总线发BusRd信号 • 其他处理器看到BusRd，检查自己是否有有效的数据副本，通知发出请求的缓存 • 如果其他缓存有有效的副本，其中一个缓存发出数据，状态变为(S)Shared • 如果其他缓存都没有有效的副本，从主存获得数据，状态变为(E)Exclusive
	PrWr	• 给总线发BusRdX信号 • 状态转换为(M)Modified • 如果其他缓存有有效的副本，其中一个缓存发出数据；否则从主存获得数据 • 如果其他缓存有有效的副本，见到BusRdX信号后无效其副本 • 向缓存块中写入修改后的值
	BusRd	• 状态保持不变，信号忽略
	BusRdX/BusUpgr	• 状态保持不变，信号忽略

原子变量

> 原子变量是一种多线程编程中常用的同步机制。它能确保对共享变量的操作在执行时不会被其他线程的操作干扰，从而避免竞态条件。
>
> 原子变量具备原子性，也就是要么全部完成，要么全部未完成。C/C++ 标准库提供了丰富的原子类型。

- `std::atomic<T>`
- `is_lock_free`：是否支持无锁操作；
- `store(T desired, std::memory_order order)`：用于将指定的值存储到原子对象中；
- `load(std::memory_order order)`：用于获取原子变量的当前值。
- `exchange(std::atomic<T>* obj, T desired)`：访问和修改包含的值，将包含的值替换并返回它前面的值。如果替换成功，则返回原来的值。

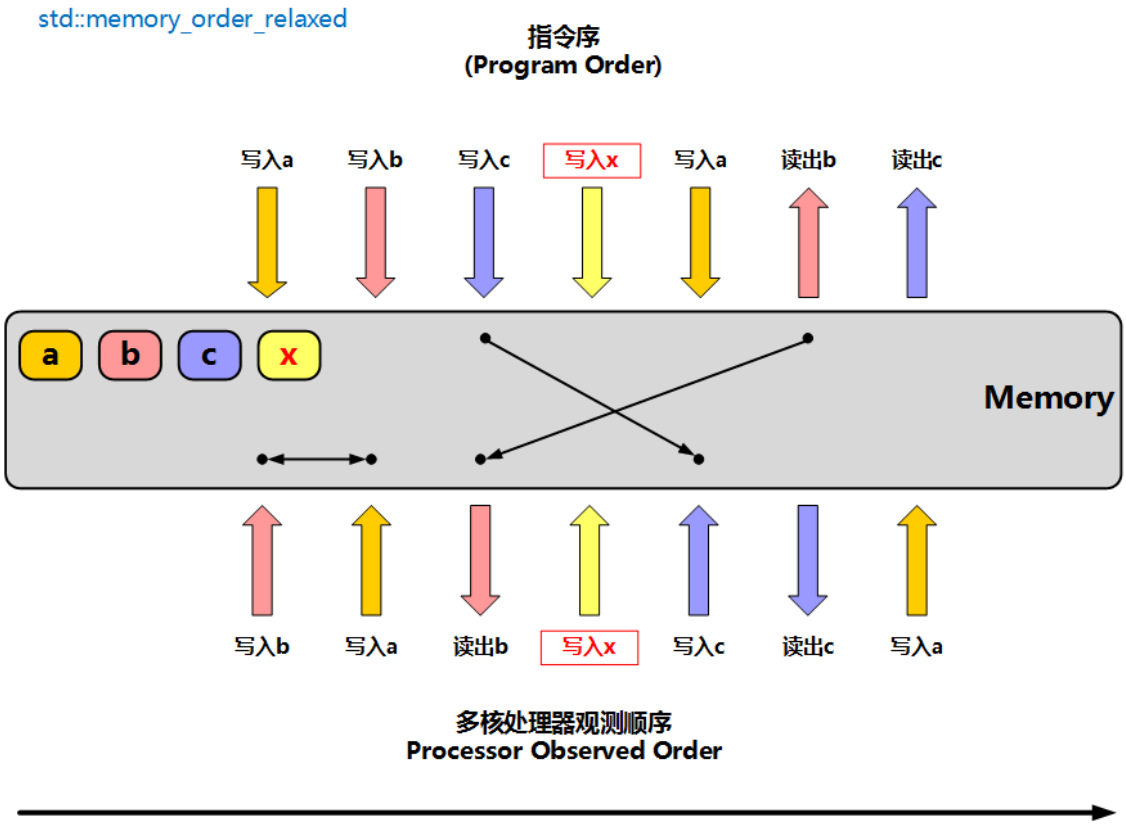
- `compare_exchange_weak(T& expected, T val, memory_order success, memory_order failure)`：比较一个值和一个期望值是否相等，如果相等则将该值替换成一个新值，并返回 `true`；否则不做任何操作并返回 `false`。注意，`compare_exchange_weak` 函数是一个弱化版本的原子操作函数，因为在某些平台上它可能会失败并重试。如果需要保证严格的原子性，则应该使用 `compare_exchange_strong` 函数。

- `compare_exchange_strong(T& expected, T val, memory_order success, memory_order failure)`
- `fetch_add`
- `fetch_sub`
- `fetch_add`
- `fetch_or`
- `fetch_xor`

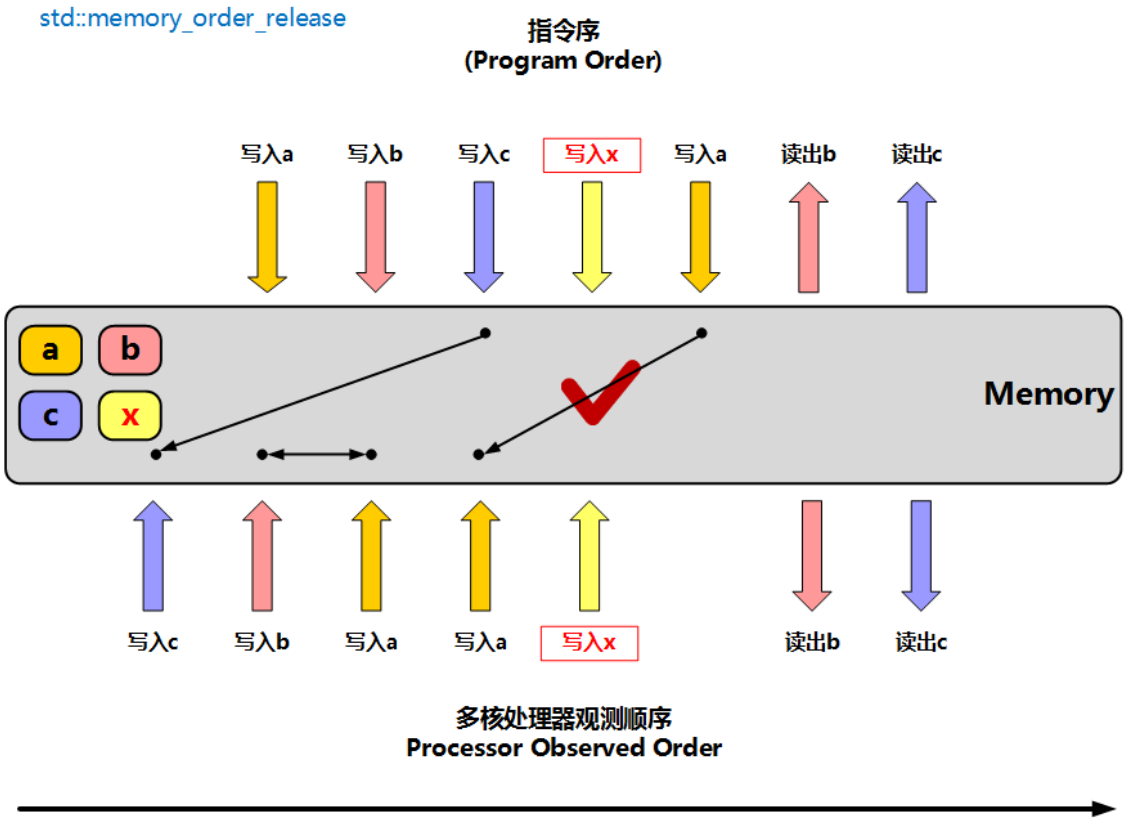
内存模型

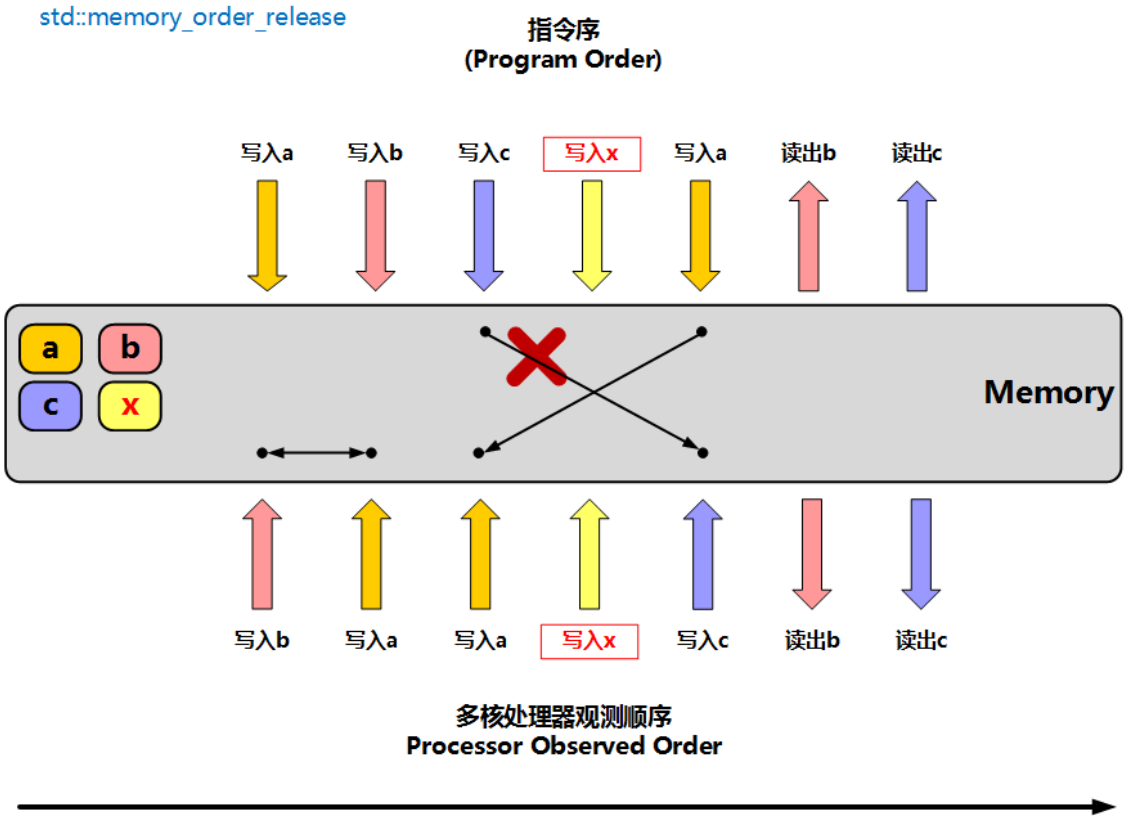
> 这里所指内存模型对应缓存一致性模型，作用是对同一时间的读写操作进行排序。在不同的 CPU 架构上，这些模型的具体实现方式可能不同，但是 C++11 帮你屏蔽了内部细节，不用考虑内存屏障，只要符合上面的使用规则，就能得到想要的效果。可能有时使用的模型粒度比较大，会损耗性能，当然还是使用各平台底层的内存屏障粒度更准确，效率也会更高，对程序员的功底要求也高。

- **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，在不需要保证顺序时使用；

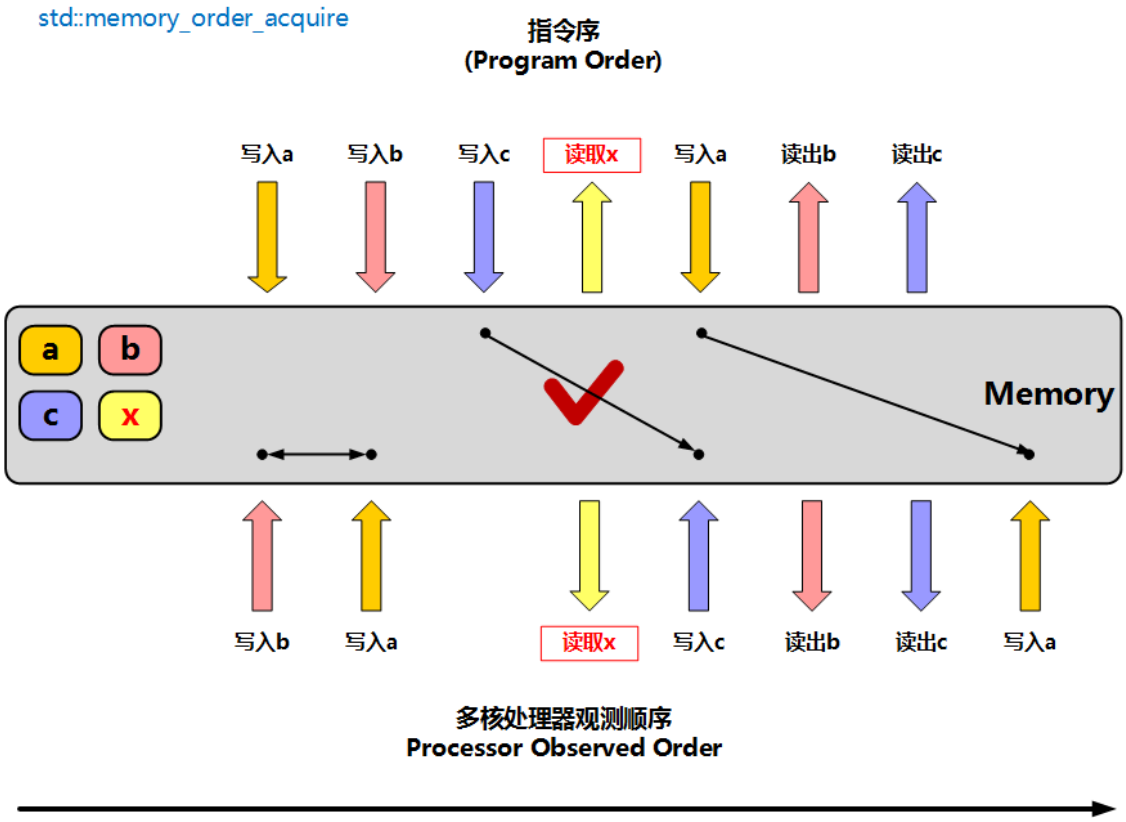


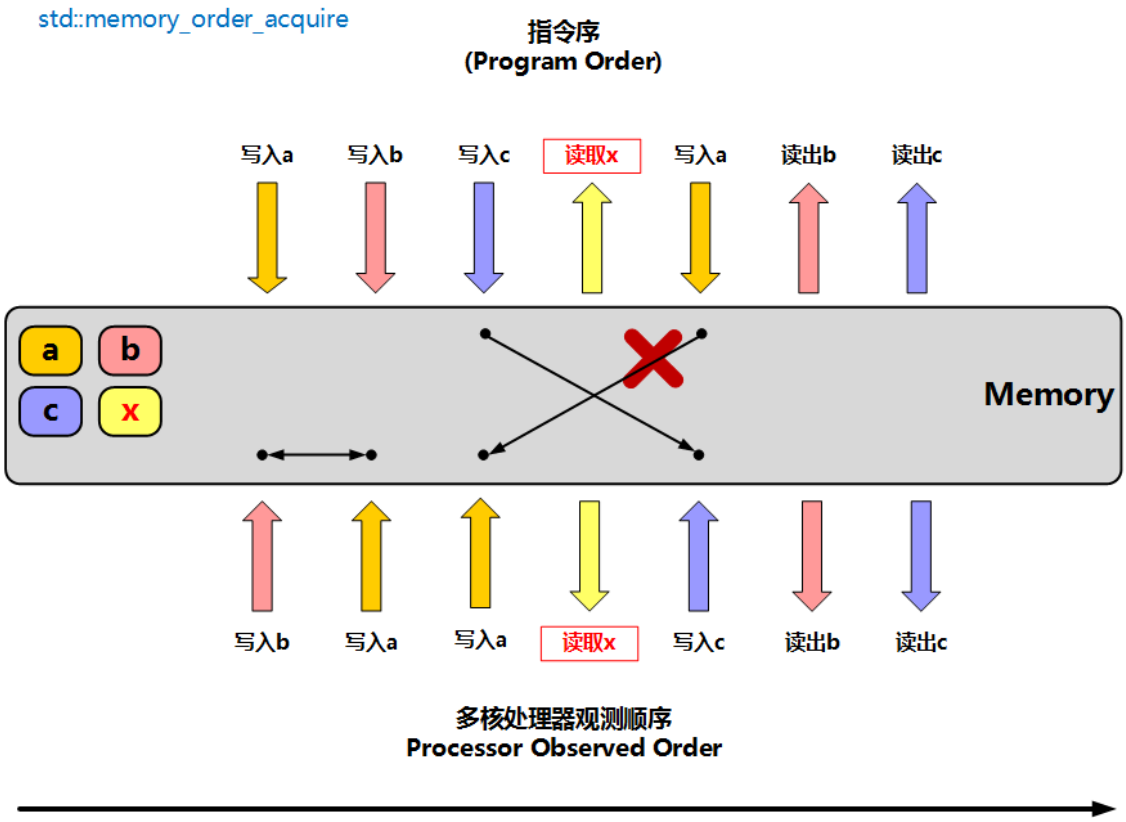
- **memory_order_release**：**释放操作**，在写入某原子对象时，当前线程的任何前面的读写操作都不允许重排到这个操作的后面去，并且当前线程的所有内存写入都在对**同一个原子**对象进行获取的其他线程可见；通常与 memory_order_acquire 或 memory_order_consume 配对使用；





- **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_consume**：同 memory_order_acquire 类似，区别是它仅对依赖于该原子变量操作涉及的对象，比如这个操作发生在原子变量 a 上，而 s = a + b；那 s 依赖于 a，但 b 不依赖于 a；当然这里也有循环依赖的问题，例如：t = s + 1，因为 s 依赖于 a，那 t 其实也是依赖于 a 的；在大多数平台上，这只会影响编译器的优化；**不建议使用**；

- **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得，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**，并且会对所有使用此模型的原子操作建立一个**全局顺序**，保证了**多个原子变量**的操作在所有线程里观察到的操作顺序相同，当然它是最慢的同步模型。