这些是内存序（Memory Order） 参数，用来控制原子操作的内存可见性和顺序性。它们非常重要，我来详细解释一下：

内存序的作用

在多核处理器中，编译器和CPU会对指令进行重排序来优化性能，这可能导致多线程程序出现意想不到的行为。内存序就是用来控制这种重排序的。

主要的内存序类型

1. `std::memory_order_relaxed`

- 最宽松：只保证原子性，不保证顺序
- 适用于计数器等不需要严格顺序的场景

2. `std::memory_order_acquire`（用于load操作）

- 获取操作：确保之后的读/写操作不会被重排到acquire之前
- 相当于建立了一个"读屏障"

3. `std::memory_order_release`（用于store操作）

- 释放操作：确保之前的读/写操作不会被重排到release之后
- 相当于建立了一个"写屏障"

4. `std::memory_order_acq_rel`（用于read-modify-write操作）

- 同时具有acquire和release语义

5. `std::memory_order_seq_cst`（默认）

- 顺序一致性：最严格的内存序，保证所有线程看到相同的操作顺序
- 性能开销最大

在你的代码中的具体作用

```
void first(function<void()> printFirst) {
    printFirst();
    state.store(1, std::memory_order_release);  // 释放屏障
    cv.notify_all();
}

void second(function<void()> printSecond) {
    std::unique_lock<std::mutex> lock(m_mutex);
    cv.wait(lock, [this](){
        return state.load(std::memory_order_acquire) == 1;  // 获取屏障
    });
    // ...
}
```

这里的内存序建立了happens-before关系：

1. **`store(1, release)`** → 确保`printFirst()`的结果对其他线程可见
2. **`load(acquire)`** → 确保看到`state == 1`后，后续操作能看到之前的所有修改

如果不使用内存序会怎样？

默认是`memory_order_seq_cst`，虽然正确但性能较差：

```
// 默认（安全但慢）
state.store(1);  // 等价于 memory_order_seq_cst
state.load();    // 等价于 memory_order_seq_cst

// 优化版本（正确且更快）
state.store(1, std::memory_order_release);
state.load(std::memory_order_acquire);
```

实际应用建议

1. 初学者：使用默认的`seq_cst`，虽然慢但安全
2. 有经验者：在理解后使用`release/acquire`配对优化性能
3. 特定场景：计数器等可以使用`relaxed`

在你的LeetCode解法中，使用正确的内存序可以确保线程间的正确同步，同时获得更好的性能。