C++ 并发编程

从线程到高性能并发系统

核心内容

线程基础 · 互斥锁 · 条件变量

线程池设计 · 原子操作 · 内存序

读写锁 · 无锁编程 · 异步日志实战

面试准备 · 项目实战 · 并发优化

作者：Aweo

2025 年 10 月

Contents

1 线程基础

1.1 为什么需要多线程？

单线程的局限：

- 无法利用多核 CPU
- I/O 阻塞时 CPU 空闲
- 无法同时处理多个任务

多线程的优势：

- 并行计算，提高 CPU 利用率
- 异步 I/O，隐藏延迟
- 提升程序响应性

1.2 线程创建与管理

1.2.1 创建线程的三种方式

```cpp
#include <iostream>
#include <thread>

// 方式1：普通函数
void worker_function(int id) {
    std::cout << "Worker " << id << " running" << std::endl;
}

// 方式2：函数对象
class WorkerFunctor {
public:
    void operator()(int id) {
        std::cout << "Functor worker " << id << std::endl;
    }
};

// 方式3：Lambda表达式（最常用）
int main() {
    // 方式1：普通函数
    std::thread t1(worker_function, 1);

    // 方式2：函数对象
    WorkerFunctor functor;
    std::thread t2(functor, 2);

    // 方式3：Lambda（推荐）
    std::thread t3([](int id) {
        std::cout << "Lambda worker " << id << std::endl;
    }, 3);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

1.2.2 join vs detach

```cpp
#include <thread>
#include <chrono>

void task() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Task completed" << std::endl;
}

int main() {
    // join：等待线程完成
    {
        std::thread t(task);
        t.join();  // 阻塞，等待t完成
        std::cout << "After join" << std::endl;
    }

    // detach：分离线程
    {
        std::thread t(task);
        t.detach();  // 立即返回，t在后台运行
        std::cout << "After detach" << std::endl;
        // 注意：主线程退出时，detach的线程可能还在运行
    }

    return 0;
}
```

面试重点：

- **join**：阻塞等待线程结束，保证资源安全释放
- **detach**：线程独立运行，主线程不再管理（危险：可能访问已销毁的资源）
- 必须调用 **join** 或 **detach**，否则析构时会调用 std::terminate()

1.2.3 参数传递

```cpp
#include <thread>
#include <string>

void func1(int x, const std::string& str) {
    std::cout << x << " " << str << std::endl;
}

void func2(int& x) {  // 引用参数
    x = 100;
}

int main() {
    // 1. 按值传递（默认）
    std::string s = "hello";
    std::thread t1(func1, 42, s);
    t1.join();

    // 2. 按引用传递（必须使用std::ref）
    int value = 10;
    std::thread t2(func2, std::ref(value));
```

```
    t2.join();
    std::cout << "value = " << value << std::endl;  // 输出100

    // 3. 移动语义（unique_ptr等只能移动的类型）
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    std::thread t3([](std::unique_ptr<int> p) {
        std::cout << *p << std::endl;
    }, std::move(ptr));
    t3.join();

    return 0;
}
```

面试重点：

- 默认按值传递（拷贝）
- 引用传递必须用 `std::ref` 或 `std::cref`
- 只移动类型（unique_ptr）必须用 `std::move`

1.2.4 获取线程信息

```cpp
#include <thread>

int main() {
    std::thread t([]() {
        // 获取当前线程ID
        std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;

        // 线程休眠
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        // 让出CPU时间片
        std::this_thread::yield();
    });

    // 检查线程是否可join
    if (t.joinable()) {
        t.join();
    }

    // 获取硬件并发线程数
    unsigned int n = std::thread::hardware_concurrency();
    std::cout << "Hardware threads: " << n << std::endl;

    return 0;
}
```

2 互斥锁与同步

2.1 数据竞争问题

```cpp
#include <thread>
#include <iostream>

int counter = 0;  // 共享变量

void increment() {
    for (int i = 0; i < 100000; ++i) {
        ++counter;  // 非原子操作！
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

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

    // 期望：200000，实际：可能小于200000（数据竞争）
    std::cout << "Counter: " << counter << std::endl;

    return 0;
}
```

问题分析：++counter 不是原子操作，实际包含 3 步：

1. 读取 counter 的值到寄存器
2. 寄存器值+1
3. 写回内存

两个线程可能同时读到相同的值，导致数据丢失。

2.2 std::mutex（互斥锁）

2.2.1 基本用法

```cpp
#include <thread>
#include <mutex>
#include <iostream>

int counter = 0;
std::mutex mtx;  // 互斥锁

void increment() {
    for (int i = 0; i < 100000; ++i) {
        mtx.lock();      // 加锁
        ++counter;       // 临界区
        mtx.unlock();    // 解锁
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
```

```
19
20      t1.join();
21      t2.join();
22
23      std::cout << "Counter: " << counter << std::endl;  // 正确：200000
24
25      return 0;
26  }
```

问题：手动 lock/unlock 容易出错（忘记 unlock、异常导致 unlock 未执行）

2.3 std::lock_guard（RAII 锁）

```cpp
#include <mutex>

std::mutex mtx;
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // 构造时加锁
        ++counter;
        // 析构时自动解锁
    }
}
```

优点：

- RAII 管理，构造加锁、析构解锁
- 异常安全（自动解锁）
- 不能手动 unlock（作用域结束才释放）

2.4 std::unique_lock（灵活的锁）

```cpp
#include <mutex>

std::mutex mtx;

void flexible_locking() {
    std::unique_lock<std::mutex> lock(mtx);

    // 1. 可以手动解锁
    // ... 临界区 ...
    lock.unlock();

    // 2. 非临界区代码
    // ... do something ...

    // 3. 重新加锁
    lock.lock();
    // ... 临界区 ...

    // 4. 转移锁的所有权
    std::unique_lock<std::mutex> lock2 = std::move(lock);

    // 5. 延迟加锁
    std::unique_lock<std::mutex> lock3(mtx, std::defer_lock);
```

```
    // ... 暂时不加锁 ...
    lock3.lock();  // 需要时加锁
}
```

unique_lock vs lock_guard：

特性	lock_guard	unique_lock
手动 unlock	❌	✅
延迟加锁	❌	✅
转移所有权	❌	✅
与条件变量配合	❌	✅
开销	低	稍高

使用建议：
- 简单场景用 lock_guard（性能更好）
- 需要灵活控制或条件变量时用 unique_lock

2.5 死锁问题

2.5.1 死锁产生的条件

```cpp
std::mutex mtx1, mtx2;

// 线程1
void thread1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lock2(mtx2);  // 等待mtx2
}

// 线程2
void thread2() {
    std::lock_guard<std::mutex> lock2(mtx2);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lock1(mtx1);  // 等待mtx1
}
// 死锁！线程1等待mtx2，线程2等待mtx1
```

死锁的四个必要条件：
1. 互斥：资源不能共享
2. 持有并等待：持有资源的同时等待其他资源
3. 不可抢占：资源不能被强制释放
4. 循环等待：形成环路

2.5.2 解决方案 1：固定加锁顺序

```cpp
// 总是按相同顺序获取锁
void thread1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::lock_guard<std::mutex> lock2(mtx2);
}

void thread2() {
    std::lock_guard<std::mutex> lock1(mtx1);  // 相同顺序
    std::lock_guard<std::mutex> lock2(mtx2);
}
```

2.5.3 解决方案 2：std::lock（原子地获取多个锁）

```cpp
void transfer(Account& from, Account& to, int amount) {
    // 同时锁定两个账户，避免死锁
    std::lock(from.mtx, to.mtx);
```

```
4
5       // 锁已经获取，使用adopt_lock表示已拥有锁
6       std::lock_guard<std::mutex> lock1(from.mtx, std::adopt_lock);
7       std::lock_guard<std::mutex> lock2(to.mtx, std::adopt_lock);
8
9       from.balance -= amount;
10      to.balance += amount;
11  }
```

2.5.4 解决方案 3：std::scoped_lock（C++17，推荐）

```
1  void transfer(Account& from, Account& to, int amount) {
2      // C++17：一行代码解决死锁
3      std::scoped_lock lock(from.mtx, to.mtx);
4
5      from.balance -= amount;
6      to.balance += amount;
7  }
```

C++

2.6 条件变量（std::condition_variable）

条件变量用于线程间同步，一个线程等待条件满足，另一个线程通知条件已满足。

2.6.1 生产者-消费者模型

```
1   #include <queue>
2   #include <mutex>
3   #include <condition_variable>
4
5   template<typename T>
6   class ThreadSafeQueue {
7       std::queue<T> queue_;
8       mutable std::mutex mtx_;
9       std::condition_variable cv_;
10
11  public:
12      // 生产者：向队列添加元素
13      void push(T value) {
14          {
15              std::lock_guard<std::mutex> lock(mtx_);
16              queue_.push(std::move(value));
17          }
18          cv_.notify_one();  // 通知一个等待的消费者
19      }
20
21      // 消费者：从队列取出元素
22      T pop() {
23          std::unique_lock<std::mutex> lock(mtx_);
24
25          // 等待队列非空
26          cv_.wait(lock, [this] { return !queue_.empty(); });
27
28          T value = std::move(queue_.front());
29          queue_.pop();
30          return value;
31      }
32
```

C++

```cpp
    // 带超时的pop
    bool try_pop(T& value, std::chrono::milliseconds timeout) {
        std::unique_lock<std::mutex> lock(mtx_);

        if (!cv_.wait_for(lock, timeout, [this] { return !queue_.empty(); })) {
            return false;  // 超时
        }

        value = std::move(queue_.front());
        queue_.pop();
        return true;
    }
};

// 使用示例
int main() {
    ThreadSafeQueue<int> queue;

    // 生产者线程
    std::thread producer([&queue]() {
        for (int i = 0; i < 10; ++i) {
            queue.push(i);
            std::cout << "Produced: " << i << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    });

    // 消费者线程
    std::thread consumer([&queue]() {
        for (int i = 0; i < 10; ++i) {
            int value = queue.pop();
            std::cout << "Consumed: " << value << std::endl;
        }
    });

    producer.join();
    consumer.join();

    return 0;
}
```

条件变量工作原理：

1. **wait**：
   - 原子地释放锁并进入等待状态
   - 被通知时重新获取锁
   - 检查条件（防止虚假唤醒）
2. **notify_one**：唤醒一个等待线程
3. **notify_all**：唤醒所有等待线程

面试重点：为什么 wait 需要 while 循环（或 lambda）？

```cpp
// ❌ 错误：没有循环检查
cv_.wait(lock);
if (!queue_.empty()) {
```

```
    // 可能队列已空（虚假唤醒或其他线程取走了）
}

// ✅ 正确：使用while循环
while (queue_.empty()) {
    cv_.wait(lock);
}

// ✅ 更好：使用lambda（内部是while循环）
cv_.wait(lock, [this] { return !queue_.empty(); });
```

虚假唤醒：条件变量可能在没有 notify 的情况下被唤醒（系统信号等）

2.7 条件变量实战：带超时的任务队列

```cpp
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <chrono>

class TaskQueue {
    std::queue<std::function<void()>> tasks_;
    std::mutex mtx_;
    std::condition_variable cv_;
    bool stop_ = false;

public:
    void submit(std::function<void()> task) {
        {
            std::lock_guard<std::mutex> lock(mtx_);
            if (stop_) return;
            tasks_.push(std::move(task));
        }
        cv_.notify_one();
    }

    bool get_task(std::function<void()>& task,
                  std::chrono::milliseconds timeout) {
        std::unique_lock<std::mutex> lock(mtx_);

        // 等待任务或超时
        if (!cv_.wait_for(lock, timeout,
                          [this] { return !tasks_.empty() || stop_; })) {
            return false;  // 超时
        }

        if (stop_) return false;  // 停止

        task = std::move(tasks_.front());
        tasks_.pop();
        return true;
    }

    void shutdown() {
        {
```

```
            std::lock_guard<std::mutex> lock(mtx_);
            stop_ = true;
        }
        cv_.notify_all();  // 唤醒所有等待线程
    }
};
```

2.8 notify_one vs notify_all

C++

```
std::mutex mtx;
std::condition_variable cv;
bool ready = false;

// 场景1：只需要一个线程处理
void worker() {
    std::unique_lock<std::mutex> lock(mtx);
    cv_.wait(lock, [] { return ready; });
    // 处理任务
}

// 使用notify_one（只唤醒一个线程）
void producer() {
    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }
    cv.notify_one();  // 只需要一个worker处理
}

// 场景2：所有线程都需要知道
void worker_all() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    // 所有worker都需要执行
}

// 使用notify_all（唤醒所有线程）
void producer_all() {
    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }
    cv.notify_all();  // 所有worker都需要知道
}
```

选择建议：

- 生产者-消费者：用 notify_one（只需要一个消费者处理）
- 屏障同步（所有线程等待某个事件）：用 notify_all

3 线程池设计与实现

3.1 为什么需要线程池？

问题：频繁创建和销毁线程的开销很大

- 线程创建：内核分配资源（栈空间、PCB 等）
- 线程销毁：内核回收资源
- 每次创建/销毁耗时约 100-1000 微秒

线程池优势：

- 复用线程，避免频繁创建/销毁
- 控制并发数量，避免过多线程
- 统一管理，提高程序稳定性

3.2 线程池基本架构

```
            Tasks Queue
                |
                v
      +--------------------+
      |   Task Queue       |
      | [任务1][任务2]...   |
      +--------------------+
                |
        +-------+-------+
        |       |       |
        v       v       v
    Thread  Thread  Thread  <- Worker Threads
```

3.3 简单线程池实现

C++

```cpp
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <future>

class ThreadPool {
    std::vector<std::thread> workers_;           // 工作线程
    std::queue<std::function<void()>> tasks_;    // 任务队列
    std::mutex mtx_;                             // 互斥锁
    std::condition_variable cv_;                 // 条件变量
    bool stop_ = false;                          // 停止标志

public:
    // 构造函数：创建指定数量的工作线程
    ThreadPool(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this] {
                while (true) {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(mtx_);

```

```
27                          // 等待任务或停止信号
28                          cv_.wait(lock, [this] {
29                              return stop_ || !tasks_.empty();
30                          });
31
32                          if (stop_ && tasks_.empty()) {
33                              return;  // 退出线程
34                          }
35
36                          task = std::move(tasks_.front());
37                          tasks_.pop();
38                      }
39
40                      task();  // 执行任务
41                  }
42              });
43          }
44      }
45
46      // 析构函数：停止所有线程
47      ~ThreadPool() {
48          {
49              std::unique_lock<std::mutex> lock(mtx_);
50              stop_ = true;
51          }
52          cv_.notify_all();
53
54          for (auto& worker : workers_) {
55              worker.join();
56          }
57      }
58
59      // 提交任务
60      template<typename F>
61      void submit(F&& f) {
62          {
63              std::unique_lock<std::mutex> lock(mtx_);
64              if (stop_) {
65                  throw std::runtime_error("submit on stopped ThreadPool");
66              }
67              tasks_.emplace(std::forward<F>(f));
68          }
69          cv_.notify_one();
70      }
71  };
72
73  // 使用示例
74  int main() {
75      ThreadPool pool(4);  // 创建4个工作线程
76
77      for (int i = 0; i < 10; ++i) {
78          pool.submit([i] {
79              std::cout << "Task " << i << " running on thread "
80                        << std::this_thread::get_id() << std::endl;
```

```
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        });
    }

    return 0;  // 析构时等待所有任务完成
}
```

3.4 支持返回值的线程池（std::future）

C++

```
class ThreadPoolWithFuture {
    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex mtx_;
    std::condition_variable cv_;
    bool stop_ = false;

public:
    ThreadPoolWithFuture(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(mtx_);
                        cv_.wait(lock, [this] { return stop_ || !tasks_.empty(); });
                        if (stop_ && tasks_.empty()) return;
                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }
                    task();
                }
            });
        }
    }

    ~ThreadPoolWithFuture() {
        {
            std::unique_lock<std::mutex> lock(mtx_);
            stop_ = true;
        }
        cv_.notify_all();
        for (auto& worker : workers_) worker.join();
    }

    // 提交任务并返回future
    template<typename F, typename... Args>
    auto submit(F&& f, Args&&... args)
        -> std::future<typename std::result_of<F(Args...)>::type>
    {
        using return_type = typename std::result_of<F(Args...)>::type;

        // 打包任务
        auto task = std::make_shared<std::packaged_task<return_type()>>(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );
```

```
47
48          std::future<return_type> result = task->get_future();
49
50          {
51              std::unique_lock<std::mutex> lock(mtx_);
52              if (stop_) {
53                  throw std::runtime_error("submit on stopped ThreadPool");
54              }
55              tasks_.emplace([task]() { (*task)(); });
56          }
57          cv_.notify_one();
58
59          return result;
60      }
61  };
62
63  // 使用示例
64  int main() {
65      ThreadPoolWithFuture pool(4);
66
67      // 提交返回int的任务
68      auto result = pool.submit([](int x) {
69          return x * x;
70      }, 10);
71
72      std::cout << "Result: " << result.get() << std::endl;  // 输出：100
73
74      // 提交多个任务
75      std::vector<std::future<int>> futures;
76      for (int i = 0; i < 10; ++i) {
77          futures.push_back(pool.submit([](int x) {
78              std::this_thread::sleep_for(std::chrono::milliseconds(100));
79              return x * x;
80          }, i));
81      }
82
83      // 获取所有结果
84      for (auto& f : futures) {
85          std::cout << f.get() << " ";
86      }
87      std::cout << std::endl;
88
89      return 0;
90  }
```

面试重点：你的项目中线程池是如何实现的？

- 固定大小的线程池：创建 4 个工作线程
- 任务队列：使用 std::queue + mutex + condition_variable
- **RAII** 管理：析构时自动停止并等待所有任务完成
- 应用场景：文件传输服务器，工作线程处理文件传输

3.5 线程池优化技巧

3.5.1 1. 动态调整线程数

```C++
class DynamicThreadPool {
    std::atomic<size_t> active_threads_{0};
    size_t max_threads_;

    void maybe_add_thread() {
        if (tasks_.size() > active_threads_ &&
            workers_.size() < max_threads_) {
            workers_.emplace_back([this] { worker_thread(); });
        }
    }
};
```

3.5.2 2. 任务优先级

```C++
struct Task {
    int priority;
    std::function<void()> func;

    bool operator<(const Task& other) const {
        return priority < other.priority;  // 优先队列：大顶堆
    }
};

std::priority_queue<Task> tasks_;  // 使用优先队列
```

3.5.3 3. 线程本地队列（减少锁竞争）

```C++
// 每个线程有自己的任务队列
thread_local std::queue<Task> local_queue_;

// 工作窃取：空闲线程从其他线程偷取任务
```

4 原子操作与内存序

4.1 为什么需要原子操作？

```cpp
// 问题：非原子操作的数据竞争
int counter = 0;
std::mutex mtx;

// 方式1：使用互斥锁（开销大）
void increment_with_mutex() {
    std::lock_guard<std::mutex> lock(mtx);
    ++counter;
}

// 方式2：使用原子操作（无锁，更高效）
std::atomic<int> atomic_counter{0};

void increment_atomic() {
    ++atomic_counter;  // 原子操作，无需加锁
}
```

原子操作的优势：

- 无锁，避免线程阻塞
- 硬件支持，性能更高
- 适合简单的计数、标志位等场景

4.2 std::atomic 基本用法

```cpp
#include <atomic>

std::atomic<int> counter{0};
std::atomic<bool> flag{false};

void worker() {
    // 原子递增
    counter++;           // 等价于 counter.fetch_add(1)
    counter += 5;        // 等价于 counter.fetch_add(5)

    // 原子比较交换（CAS: Compare-And-Swap）
    int expected = 10;
    int desired = 20;
    if (counter.compare_exchange_strong(expected, desired)) {
        // 成功：counter原来是10，现在设置为20
    } else {
        // 失败：counter不是10，expected被更新为counter的实际值
    }

    // 原子读写
    int value = counter.load();
    counter.store(100);

    // 原子交换
    int old_value = counter.exchange(200);
}
```

4.3 常用原子操作

```cpp
std::atomic<int> x{0};

// 1. fetch_add：原子地加然后返回旧值
int old = x.fetch_add(10);  // x = 10, old = 0

// 2. fetch_sub：原子地减然后返回旧值
old = x.fetch_sub(3);  // x = 7, old = 10

// 3. fetch_and、fetch_or、fetch_xor（位运算）
std::atomic<unsigned int> flags{0};
flags.fetch_or(0x01);   // 设置第0位
flags.fetch_and(~0x01); // 清除第0位

// 4. exchange：原子地交换并返回旧值
old = x.exchange(100);  // x = 100, old = 7

// 5. compare_exchange_weak / compare_exchange_strong
int expected = 100;
bool success = x.compare_exchange_strong(expected, 200);
// 如果x == expected，则x = 200，返回true
// 否则expected = x，返回false
```

4.4 CAS（Compare-And-Swap）详解

原理：原子地比较并交换，是无锁编程的基础。

```cpp
// CAS伪代码
bool compare_and_swap(int* ptr, int expected, int desired) {
    if (*ptr == expected) {
        *ptr = desired;
        return true;
    } else {
        expected = *ptr;  // 更新expected为实际值
        return false;
    }
    // 整个过程是原子的！
}
```

应用：无锁栈

```cpp
template<typename T>
class LockFreeStack {
    struct Node {
        T data;
        Node* next;
    };

    std::atomic<Node*> head_{nullptr};

public:
    void push(const T& data) {
        Node* new_node = new Node{data, nullptr};

        // CAS循环
```

```
        new_node->next = head_.load();
        while (!head_.compare_exchange_weak(new_node->next, new_node)) {
            // 失败：head_已被其他线程修改，重试
            // new_node->next已被更新为head_的新值
        }
    }

    bool pop(T& result) {
        Node* old_head = head_.load();

        while (old_head &&
               !head_.compare_exchange_weak(old_head, old_head->next)) {
            // 失败：重试
        }

        if (!old_head) return false;

        result = old_head->data;
        delete old_head;  // 注意：实际需要处理ABA问题和内存回收
        return true;
    }
};
```

compare_exchange_weak vs compare_exchange_strong：

- **weak**：可能虚假失败（即使值相等也可能失败），但性能更好
- **strong**：不会虚假失败，但可能稍慢
- 使用建议：在循环中使用 weak，单次 CAS 使用 strong

4.5 内存序（Memory Order）

C++提供 6 种内存序，控制原子操作的可见性和顺序：

```
enum class memory_order {
    relaxed,       // 最宽松：只保证原子性
    consume,       // 很少使用
    acquire,       // 获取语义
    release,       // 释放语义
    acq_rel,       // 获取-释放语义
    seq_cst        // 顺序一致性（默认，最严格）
};
```

4.5.1 memory_order_relaxed（宽松序）

特点：只保证原子性，不保证顺序

```
std::atomic<int> x{0}, y{0};

// 线程1
void thread1() {
    x.store(1, std::memory_order_relaxed);
    y.store(1, std::memory_order_relaxed);
}

// 线程2
void thread2() {
    while (!y.load(std::memory_order_relaxed));  // 等待y == 1
```

```
    // 此时x可能还是0！（CPU乱序执行）
}
```

使用场景：计数器（只关心最终值）

```C++
std::atomic<int> counter{0};

void increment() {
    counter.fetch_add(1, std::memory_order_relaxed);  // 性能最好
}
```

4.5.2 memory_order_acquire / release（获取-释放序）

特点：建立 happens-before 关系

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

// 线程1：生产者
void producer() {
    data = 42;  // 1
    ready.store(true, std::memory_order_release);  // 2
    // 保证：1 happens-before 2
}

// 线程2：消费者
void consumer() {
    while (!ready.load(std::memory_order_acquire));  // 3
    assert(data == 42);  // 4：一定成立
    // 保证：2 happens-before 3，3 happens-before 4
    // 因此：1 happens-before 4
}
```

使用场景：自旋锁、生产者-消费者

4.5.3 memory_order_seq_cst（顺序一致性，默认）

特点：最严格，保证全局一致的顺序

```C++
// 默认使用seq_cst
x.store(1);  // 等价于 x.store(1, std::memory_order_seq_cst)
int v = x.load();  // 等价于 x.load(std::memory_order_seq_cst)
```

优点：最安全，符合直觉 缺点：性能最差

4.5.4 性能对比与选择

| 内存序 | 性能 | 保证 | 使用场景 ||---|---|---|---| relaxed | 最快 | 只保证原子性 | 简单计数器 || acquire/release | 中等 | happens-before | 自旋锁、生产者-消费者 || seq_cst | 最慢 | 全局一致顺序 | 默认、复杂同步 |

建议：

- 不确定时用 seq_cst（默认）
- 性能关键且理解内存序时，使用 acquire/release 或 relaxed

5 读写锁与自旋锁

5.1 std::shared_mutex（读写锁，C++17）

特点：允许多个读线程同时访问，写线程独占

```cpp
#include <shared_mutex>
#include <unordered_map>

class ThreadSafeCache {
    std::unordered_map<std::string, std::string> cache_;
    mutable std::shared_mutex mtx_;  // 读写锁

public:
    // 读操作：共享锁（多个线程可以同时读）
    std::string get(const std::string& key) const {
        std::shared_lock<std::shared_mutex> lock(mtx_);
        auto it = cache_.find(key);
        return it != cache_.end() ? it->second : "";
    }

    // 写操作：独占锁（只有一个线程可以写）
    void set(const std::string& key, const std::string& value) {
        std::unique_lock<std::shared_mutex> lock(mtx_);
        cache_[key] = value;
    }
};
```

对应你的简历项目：

```
文件元数据使用哈希表+读写锁实现细粒度并发控制，支持多线程同时读取
```

优势：

- 读多写少的场景性能提升明显
- 多个读线程不会互相阻塞

面试重点：什么时候使用读写锁？

- 读操作远多于写操作（读写比 > 10:1）
- 临界区较大（读写锁开销比普通 mutex 大）
- 例如：缓存、配置管理、元数据查询

5.2 自旋锁（Spinlock）

原理：通过 busy-wait 循环等待，不进入睡眠

```cpp
class Spinlock {
    std::atomic_flag flag_ = ATOMIC_FLAG_INIT;

public:
    void lock() {
        while (flag_.test_and_set(std::memory_order_acquire)) {
            // 自旋等待
            // 可选：std::this_thread::yield(); // 让出CPU
        }
    }

    void unlock() {
```

```
        flag_.clear(std::memory_order_release);
    }
};

// 使用示例
Spinlock spinlock;
int counter = 0;

void increment() {
    spinlock.lock();
    ++counter;
    spinlock.unlock();
}
```

自旋锁 **vs** 互斥锁：

特性	自旋锁	互斥锁
等待方式	busy-wait（消耗 CPU）	睡眠（不消耗 CPU）
上下文切换	无	有
适用场景	临界区很小（几十条指令）	临界区较大
多核	必须	不限

使用建议：

- 临界区非常小且执行时间可预测：自旋锁
- 临界区较大或可能阻塞：互斥锁
- 单核系统：不要用自旋锁（无意义）

6 异步日志实战（对应你的简历项目）

6.1 异步日志的设计目标

问题：同步日志阻塞业务线程

```C++
// 同步日志（慢！）
void handle_request() {
    // 处理请求
    process_data();

    // 写日志（阻塞，可能几毫秒）
    fwrite(log_buffer, size, 1, logfile);  // I/O阻塞
    fflush(logfile);
}
```

解决：异步日志，业务线程只写内存，日志线程负责刷盘

6.2 双缓冲异步日志实现

对应你的简历：

```
采用双缓冲设计，前端线程无锁写入，后端线程批量刷盘，通过条件变量实现零拷贝缓冲区切换
```

```C++
#include <vector>
#include <mutex>
#include <condition_variable>
#include <thread>
#include <fstream>
#include <chrono>

class AsyncLogger {
    using Buffer = std::vector<char>;

    Buffer front_buffer_;  // 前端缓冲区（业务线程写入）
    Buffer back_buffer_;   // 后端缓冲区（日志线程刷盘）

    std::mutex mtx_;
    std::condition_variable cv_;
    std::thread log_thread_;

    std::ofstream logfile_;
    bool stop_ = false;

    static constexpr size_t BUFFER_SIZE = 4 * 1024 * 1024;  // 4MB

public:
    AsyncLogger(const std::string& filename)
        : logfile_(filename, std::ios::app)
    {
        front_buffer_.reserve(BUFFER_SIZE);
        back_buffer_.reserve(BUFFER_SIZE);

        // 启动日志线程
        log_thread_ = std::thread([this] { log_thread_func(); });
    }

```

```
34      ~AsyncLogger() {
35          {
36              std::lock_guard<std::mutex> lock(mtx_);
37              stop_ = true;
38          }
39          cv_.notify_one();
40          log_thread_.join();
41      }
42
43      // 业务线程调用：快速写入前端缓冲区
44      void log(const std::string& message) {
45          std::lock_guard<std::mutex> lock(mtx_);
46
47          front_buffer_.insert(front_buffer_.end(),
48                               message.begin(), message.end());
49          front_buffer_.push_back('\n');
50
51          // 缓冲区满或超时，通知日志线程
52          if (front_buffer_.size() >= BUFFER_SIZE) {
53              cv_.notify_one();
54          }
55      }
56
57  private:
58      // 日志线程：批量刷盘
59      void log_thread_func() {
60          while (true) {
61              {
62                  std::unique_lock<std::mutex> lock(mtx_);
63
64                  // 等待：缓冲区有数据 或 超时 或 停止
65                  cv_.wait_for(lock, std::chrono::seconds(3), [this] {
66                      return !front_buffer_.empty() || stop_;
67                  });
68
69                  if (stop_ && front_buffer_.empty()) {
70                      break;  // 退出
71                  }
72
73                  // 零拷贝交换缓冲区（swap只交换指针）
74                  front_buffer_.swap(back_buffer_);
75              }
76
77              // 写入磁盘（不持有锁，不阻塞业务线程）
78              if (!back_buffer_.empty()) {
79                  logfile_.write(back_buffer_.data(), back_buffer_.size());
80                  logfile_.flush();
81                  back_buffer_.clear();
82              }
83          }
84      }
85  };
86
87  // 使用示例
```

```
int main() {
    AsyncLogger logger("app.log");

    // 业务线程：快速写日志（无阻塞）
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back([&logger, i] {
            for (int j = 0; j < 1000; ++j) {
                logger.log("Thread " + std::to_string(i) +
                           " log " + std::to_string(j));
            }
        });
    }

    for (auto& t : threads) {
        t.join();
    }

    return 0;  // 析构时等待日志全部写完
}
```

关键技术点：

1. 双缓冲：
   - 前端缓冲区：业务线程写入（持有锁时间短）
   - 后端缓冲区：日志线程刷盘（不持有锁）
2. 零拷贝：
   - swap()只交换指针，不复制数据
3. 条件变量：
   - 缓冲区满时立即刷盘
   - 超时时也刷盘（避免日志延迟）
4. 批量写入：
   - 积累多条日志一次性写入磁盘
   - 减少系统调用次数

性能对比：

- 同步日志：每条日志约 1-10ms（磁盘 I/O）
- 异步日志：每条日志约几微秒（内存操作）
- 性能提升 **100-1000** 倍

面试重点：你的项目中异步日志如何实现？

- 双缓冲设计
- 前端无锁写入（持锁时间极短）
- 后端批量刷盘
- 条件变量通知
- swap 零拷贝切换
- 消除日志 I/O 对传输性能的影响

7 常见面试问题总结

7.1 线程基础

1. join 和 detach 的区别？

- **join**：阻塞等待线程结束，保证资源安全
- **detach**：线程独立运行，主线程不再管理（危险）
- 必须调用其中一个，否则析构时 terminate

2. 如何向线程传递参数？

- 默认按值传递（拷贝）
- 引用必须用 `std::ref`
- 只移动类型（unique_ptr）用 `std::move`

3. std::thread 的拷贝和移动？

- 不可拷贝（deleted）
- 可移动（move）

7.2 锁相关

1. lock_guard vs unique_lock？

| 特性 | lock_guard | unique_lock | |——|————|————-| | 手动 unlock | ❌ | ✅ | | 延迟加锁 | ❌ | ✅ | | 条件变量 | ❌ | ✅ | | 开销 | 低 | 稍高 |

2. 如何避免死锁？

- 固定加锁顺序
- 使用 `std::lock` 或 `std::scoped_lock` 同时获取多个锁
- 超时机制（try_lock_for）
- 避免持有锁时调用外部代码

3. 什么是虚假唤醒？

- 条件变量可能在没有 notify 的情况下被唤醒
- 必须用 while 循环（或 lambda）检查条件
- `cv.wait(lock, []{ return condition; })`内部就是 while 循环

4. 读写锁适用场景？

- 读多写少（读写比 > 10:1）
- 临界区较大
- 例如：缓存、配置、元数据查询

7.3 线程池

1. 为什么需要线程池？

- 避免频繁创建/销毁线程的开销
- 控制并发数量
- 统一管理

2. 你的项目中线程池如何实现？

- 固定大小（4 个工作线程）
- 任务队列（queue + mutex + cv）
- RAII 管理（析构自动停止）
- 应用：文件传输服务器

3. 如何优雅地关闭线程池？

- 设置 stop 标志
- notify_all 唤醒所有等待线程
- 等待所有线程 join
- 处理剩余任务（可选）

7.4 原子操作

1. 原子操作 **vs** 锁？

- 原子操作：无锁，性能更高，适合简单操作
- 锁：适合复杂临界区

2. CAS 是什么？应用场景？

- Compare-And-Swap：原子地比较并交换
- 无锁编程的基础
- 应用：无锁栈/队列、引用计数

3. 内存序是什么？

- 控制原子操作的可见性和顺序
- relaxed：只保证原子性
- acquire/release：happens-before
- seq_cst：全局一致（默认）

4. compare_exchange_weak vs strong？

- weak：可能虚假失败，性能更好，循环中使用
- strong：不会虚假失败，单次 CAS 使用

7.5 并发问题排查

1. 如何排查死锁？

- GDB：`info threads` + `thread apply all bt`
- pstack：查看所有线程栈
- 日志：记录加锁顺序
- 工具：helgrind（valgrind 的一部分）

2. 如何排查数据竞争？

- ThreadSanitizer（TSan）：gcc/clang -fsanitize=thread
- helgrind（valgrind）
- 代码审查：检查共享变量的访问

3. 如何定位性能瓶颈？

- perf：CPU profiling
- 锁竞争分析：perf lock
- 火焰图：可视化性能热点

7.6 项目相关（针对你的简历）

1. 异步日志的双缓冲如何实现？

- 前端缓冲区：业务线程写入
- 后端缓冲区：日志线程刷盘
- swap 零拷贝切换
- 条件变量通知

2. 读写锁如何使用？

- 文件元数据：读多写少

- `shared_lock`：多线程同时读取
- `unique_lock`：写操作独占

3. 对象池如何实现？

- 预分配内存
- 空闲列表管理
- 加锁保护（或使用无锁栈）

4. 如何处理并发竞态？

- GDB 调试：设置条件断点
- 日志：记录关键状态
- 加锁：保护共享资源
- 原子操作：简单的计数/标志

7.7 最终建议

面试准备：

1. 熟练掌握 mutex、condition_variable、atomic
2. 理解线程池原理，能手写基本版本
3. 深入了解你项目中的并发技术（异步日志、读写锁）
4. 准备好性能数据（为什么用异步日志？提升多少？）
5. 了解并发问题排查工具（GDB、TSan、perf）

回答技巧：

1. 先说结论，再解释原理
2. 结合项目实际经验
3. 对比不同方案（锁 vs 原子操作）
4. 提到性能优化（双缓冲、对象池）
5. 展示问题排查能力

加分项：

1. 了解内存序和无锁编程
2. 阅读过优秀并发库源码（muduo、folly）
3. 能讲出遇到的并发 Bug 和解决过程
4. 了解现代并发技术（C++20 coroutine）
5. 性能调优经验（锁优化、线程数调优）