C++基础面试题精选

> 基于你的简历：熟练使用C/C++，掌握STL容器/算法、C++11/14特性、内存管理、多线程编程

一、STL容器与算法

Q1: vector的底层实现原理？扩容机制是什么？

标准回答：

vector底层是动态数组：

- 连续内存存储
- 随机访问O(1)
- 尾部插入均摊O(1)

扩容机制：

1. 当前容量用满时触发扩容
2. 一般扩容为原来的1.5倍或2倍（不同编译器可能不同）
3. 分配新内存，拷贝/移动旧元素，释放旧内存

性能优化：

```
vector<int> vec;
vec.reserve(1000);  // 预分配内存，避免多次扩容
```

与你项目的联系： "在文件传输系统中，我用vector管理Buffer对象池，预先reserve一定数量避免频繁扩容。"

Q2: map和unordered_map的区别？如何选择？

标准回答：

特性	map	unordered_map
底层实现	红黑树	哈希表
时间复杂度	O(log n)	平均O(1)，最坏O(n)
有序性	有序	无序
内存占用	较小	较大（需要额外的桶）

选择建议：

- 需要有序 → map
- 需要高效查找 → unordered_map

- 频繁范围查询 → map

与你项目的联系： "文件传输系统中，文件元数据用unordered_map存储，因为只需要通过文件名快速查找，不需要有序性。"

Q3: 迭代器失效的场景有哪些?

标准回答：

vector迭代器失效：

- 插入/删除元素时，插入点之后的迭代器失效
- 扩容时，所有迭代器失效

map迭代器失效：

- 删除元素时，只有指向被删元素的迭代器失效
- 插入元素不会导致迭代器失效

安全删除：

```
// vector安全删除
for (auto it = vec.begin(); it != vec.end(); ) {
    if (条件) {
        it = vec.erase(it);  // erase返回下一个有效迭代器
    } else {
        ++it;
    }
}

// map安全删除（C++11之前）
for (auto it = m.begin(); it != m.end(); ) {
    if (条件) {
        m.erase(it++);  // 先拷贝迭代器，再删除
    } else {
        ++it;
    }
}

// map安全删除（C++11）
for (auto it = m.begin(); it != m.end(); ) {
    if (条件) {
        it = m.erase(it);  // C++11 erase返回下一个迭代器
    } else {
        ++it;
    }
}
```

二、C++11/14特性

Q4: 智能指针有哪几种？区别是什么？

标准回答：

shared_ptr（共享所有权）：

- 引用计数管理
- 多个shared_ptr可以指向同一对象
- 最后一个shared_ptr析构时释放对象
- 线程安全（引用计数的增减）

unique_ptr（独占所有权）：

- 不可拷贝，只能移动
- 性能更高（无引用计数开销）
- 适合明确的单一所有权场景

weak_ptr（弱引用）：

- 不增加引用计数
- 解决shared_ptr循环引用问题
- 需要通过lock()转换为shared_ptr使用

使用场景：

```
// 优先使用unique_ptr
std::unique_ptr<Obj> obj = std::make_unique<Obj>();

// 需要共享所有权时用shared_ptr
std::shared_ptr<Obj> obj = std::make_shared<Obj>();

// 避免循环引用
class Node {
    std::shared_ptr<Node> next;    // 强引用
    std::weak_ptr<Node> prev;      // 弱引用，避免循环
};
```

与你项目的联系： "在Raft项目中，节点之间的通信连接使用shared_ptr管理，因为多个线程可能同时使用同一个连接。"

Q5: 右值引用和移动语义的作用？

标准回答：

右值引用（T&&）：

- 绑定到临时对象（右值）
- 允许"窃取"资源而非拷贝

移动语义的优势：

```
class Buffer {
    char* data;
    size_t size;
public:
    // 拷贝构造（深拷贝，开销大）
    Buffer(const Buffer& other) {
        size = other.size;
        data = new char[size];
        memcpy(data, other.data, size);
    }

    // 移动构造（转移资源，开销小）
    Buffer(Buffer&& other) noexcept {
        data = other.data;
        size = other.size;
        other.data = nullptr;  // 窃取资源
        other.size = 0;
    }
};

// 使用场景
Buffer createBuffer() {
    Buffer buf(1024);
    return buf;  // 触发移动而非拷贝
}

Buffer buf = createBuffer();  // 移动语义，高效
```

std::move的作用：

- 将左值转换为右值引用
- 明确表示"可以移动"

```
Buffer buf1(1024);
Buffer buf2 = std::move(buf1);  // 移动而非拷贝
// buf1不再可用
```

与你项目的联系： "文件传输系统中，Buffer对象在线程间传递时使用std::move避免拷贝大块内存，提高性能。"

Q6: lambda表达式的捕获方式有哪些?

标准回答：

捕获方式：

```
int x = 10;
int y = 20;

// 值捕获（拷贝）
auto f1 = [x]() { return x; };

// 引用捕获
auto f2 = [&x]() { x++; };

// 捕获所有（值）
auto f3 = [=]() { return x + y; };

// 捕获所有（引用）
auto f4 = [&]() { x++; y++; };

// 混合捕获
auto f5 = [=, &x]() { x++; return y; };

// C++14 初始化捕获
auto f6 = [ptr = std::make_unique<int>(42)]() { return *ptr; };
```

注意事项：

- 值捕获默认是const的，需要加mutable才能修改
- 引用捕获要注意对象生命周期
- 捕获this指针访问成员变量

```
class Foo {
    int value = 42;
public:
    auto getFunc() {
        return [this]() { return value; };  // 捕获this
    }
};
```

与你项目的联系： "在Raft项目的RPC实现中，使用lambda作为回调函数，捕获请求上下文处理响应。"

三、内存管理

Q7: new/delete和malloc/free的区别?

标准回答：

特性	new/delete	malloc/free
类型	C++运算符	C库函数
调用构造/析构	是	否

特性	new/delete	malloc/free
返回类型	具体类型指针	void*
失败处理	抛出异常	返回NULL
大小计算	自动	需要手动指定
重载	可以	不可以

使用示例：

```
// new/delete
Obj* obj = new Obj();         // 调用构造函数
delete obj;                   // 调用析构函数

Obj* arr = new Obj[10];       // 调用10次构造
delete[] arr;                 // 调用10次析构

// malloc/free
Obj* obj = (Obj*)malloc(sizeof(Obj));  // 只分配内存
free(obj);                             // 只释放内存
```

不能混用：

```
Obj* obj = new Obj();
free(obj);            // ❌ 错误：不会调用析构函数

Obj* obj = (Obj*)malloc(sizeof(Obj));
delete obj;           // ❌ 错误：delete期望通过new分配
```

Q8: 内存泄漏的常见场景和检测方法?

标准回答：

常见场景：

1. new后忘记delete
2. 异常抛出导致delete未执行
3. 循环引用（shared_ptr）
4. 容器中存储指针忘记释放

检测方法：

1. valgrind

```
valgrind --leak-check=full ./program
```

2. AddressSanitizer（编译时选项）

```
g++ -fsanitize=address -g program.cpp
./a.out
```

3. 智能指针（预防）

```
// ❌ 可能泄漏
void func() {
    Obj* obj = new Obj();
    if (error) return;  // 忘记delete
    delete obj;
}

// ✅ 自动释放
void func() {
    std::unique_ptr<Obj> obj = std::make_unique<Obj>();
    if (error) return;  // 自动析构
}
```

与你项目的联系： "在文件传输系统开发中，我使用valgrind检测内存泄漏，发现了Buffer对象池在某些异常情况下没有正确归还对象的问题，最后通过RAII模式解决。"

Q9: 什么是RAII？如何应用？

标准回答：

RAII（Resource Acquisition Is Initialization）：

- 资源获取即初始化
- 利用对象生命周期管理资源
- 构造函数获取资源，析构函数释放资源

经典应用：

1. 智能指针

```
std::unique_ptr<int> ptr(new int(42));
// 自动释放
```

2. 锁管理

```
std::mutex mtx;
```

```
void func() {
    std::lock_guard<std::mutex> lock(mtx);
    // 临界区代码
    if (error) return;  // 自动解锁
    // ...
}  // 离开作用域自动解锁
```

3. 文件管理

```
class FileGuard {
    FILE* fp;
public:
    FileGuard(const char* path) {
        fp = fopen(path, "r");
    }
    ~FileGuard() {
        if (fp) fclose(fp);
    }
};
```

与你项目的联系： "在并发控制中，使用std::lock_guard管理读写锁，保证即使出现异常也能正确解锁，避免死锁。"

四、多线程编程

Q10: mutex、lock_guard、unique_lock的区别?

标准回答:

mutex（基础互斥锁）：

```
std::mutex mtx;
mtx.lock();
// 临界区
mtx.unlock();  // 必须手动解锁，容易忘记
```

lock_guard（RAII封装）：

```
std::mutex mtx;
{
    std::lock_guard<std::mutex> lock(mtx);
    // 临界区
}  // 自动解锁
```

- 构造时加锁，析构时解锁

- 不可移动，不可复制
- 不能手动unlock

unique_lock（灵活的锁）：

```
std::mutex mtx;
std::unique_lock<std::mutex> lock(mtx);

// 可以提前解锁
lock.unlock();

// 可以重新加锁
lock.lock();

// 可以转移所有权
std::unique_lock<std::mutex> lock2 = std::move(lock);
```

- 可以延迟加锁、尝试加锁、定时加锁
- 可以配合条件变量使用
- 开销比lock_guard略大

使用场景：

- 简单加锁 → lock_guard
- 需要配合条件变量 → unique_lock
- 需要手动控制锁的生命周期 → unique_lock

与你项目的联系： "文件传输系统中，双缓冲日志使用unique_lock配合条件变量实现缓冲区切换；读写锁保护元数据哈希表时使用lock_guard足够简单高效。"

Q11: 条件变量如何使用？为什么要配合mutex？

标准回答：

基本用法：

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

// 生产者
void producer() {
    std::unique_lock<std::mutex> lock(mtx);
    // 准备数据
    ready = true;
    cv.notify_one();  // 通知消费者
}

// 消费者
```

```
void consumer() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return ready; });  // 等待条件满足
    // 处理数据
}
```

为什么要配合mutex?

1. 保护共享变量（如ready）
2. wait()内部会：
    - 原子地释放锁并进入等待
    - 被唤醒后重新获取锁

为什么用while不用if?

```
// ❌ 错误
cv.wait(lock);
if (ready) { /* ... */ }

// ✅ 正确
cv.wait(lock, []{ return ready; });
// 或
while (!ready) {
    cv.wait(lock);
}
```

原因：可能发生虚假唤醒（spurious wakeup）

与你项目的联系： "文件传输系统的双缓冲日志：前端线程写满buffer A后通知后端线程；后端线程等待条件变量，被唤醒后交换buffer并刷盘。"

```
// 伪代码
void frontendThread() {
    std::unique_lock<std::mutex> lock(mtx);
    while (bufferA.full()) {
        cv.wait(lock);  // 等待bufferA被处理
    }
    bufferA.write(log);
    if (bufferA.full()) {
        cv.notify_one();  // 通知后端处理
    }
}

void backendThread() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []{ return bufferA.full(); });
        std::swap(bufferA, bufferB);  // 交换buffer
        lock.unlock();
```

```
            bufferB.flush();  // 刷盘
        }
    }
```

Q12: 读写锁的使用场景？和互斥锁有什么区别?

标准回答：

互斥锁（mutex）：

- 读写都互斥
- 同一时间只有一个线程访问

读写锁（shared_mutex / pthread_rwlock）：

- 读读共享
- 读写互斥
- 写写互斥

C++实现：

```
#include <shared_mutex>

std::shared_mutex rw_mtx;

// 读操作（共享锁）
void read() {
    std::shared_lock<std::shared_mutex> lock(rw_mtx);
    // 多个线程可以同时读
}

// 写操作（独占锁）
void write() {
    std::unique_lock<std::shared_mutex> lock(rw_mtx);
    // 独占访问
}
```

适用场景：

- 读操作远多于写操作
- 读操作耗时较长

与你项目的联系： "文件元数据哈希表用pthread_rwlock保护：多个线程可以同时查询文件信息（共享锁），但上传新文件更新元数据时需要独占锁。这样在读多写少的场景下，性能比用mutex好很多。"

五、综合应用题

Q13: 实现一个线程安全的单例模式

标准回答：

C++11 局部静态变量（推荐）：

```
class Singleton {
private:
    Singleton() {}
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

public:
    static Singleton& getInstance() {
        static Singleton instance;  // C++11保证线程安全
        return instance;
    }
};
```

双重检查锁定（了解即可）：

```
class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;
    Singleton() {}

public:
    static Singleton* getInstance() {
        if (instance == nullptr) {  // 第一次检查（无锁）
            std::lock_guard<std::mutex> lock(mtx);
            if (instance == nullptr) {  // 第二次检查（有锁）
                instance = new Singleton();
            }
        }
        return instance;
    }
};
```

面试官可能追问：为什么局部静态变量是线程安全的?

C++11标准规定：如果多个线程同时进入声明静态局部变量的语句，只有一个线程会执行初始化，其他线程会阻塞等待。

Q14: 实现一个简单的线程池

标准回答思路：

```
class ThreadPool {
private:
    std::vector<std::thread> threads;          // 线程列表
    std::queue<std::function<void()>> tasks;   // 任务队列
    std::mutex mtx;                            // 保护任务队列
    std::condition_variable cv;                // 通知工作线程
    bool stop = false;                         // 停止标志

public:
    ThreadPool(size_t numThreads) {
        for (size_t i = 0; i < numThreads; ++i) {
            threads.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();  // 执行任务
                }
            });
        }
    }

    template<typename F>
    void enqueue(F&& task) {
        {
            std::unique_lock<std::mutex> lock(mtx);
            tasks.emplace(std::forward<F>(task));
        }
        cv.notify_one();
    }

    ~ThreadPool() {
        {
            std::unique_lock<std::mutex> lock(mtx);
            stop = true;
        }
        cv.notify_all();
        for (auto& t : threads) {
            t.join();
        }
    }
};
```

关键点：

1. 任务队列 + 互斥锁保护
2. 条件变量通知工作线程
3. 析构时优雅退出（通知所有线程并join）

与你项目的联系： "文件传输系统中，主线程负责epoll事件监听，工作线程池处理文件读写等耗时操作，避免阻塞事件循环。"

六、性能优化相关

Q15: 如何避免内存碎片？

标准回答：

内存碎片产生原因：

- 频繁分配和释放不同大小的内存块
- 长时间运行后，空闲内存分散

解决方案：

1. 对象池

```
class BufferPool {
    std::vector<Buffer*> pool;
    std::mutex mtx;
public:
    Buffer* acquire() {
        std::lock_guard<std::mutex> lock(mtx);
        if (pool.empty()) {
            return new Buffer();
        }
        Buffer* buf = pool.back();
        pool.pop_back();
        return buf;
    }

    void release(Buffer* buf) {
        std::lock_guard<std::mutex> lock(mtx);
        pool.push_back(buf);
    }
};
```

2. 预分配

```
std::vector<int> vec;
vec.reserve(1000);  // 预分配，避免多次扩容
```

3. 内存池/自定义分配器

```
class MemoryPool {
    char* pool;
    size_t size;
    size_t offset = 0;
public:
    void* allocate(size_t n) {
        if (offset + n > size) return nullptr;
        void* ptr = pool + offset;
        offset += n;
        return ptr;
    }
};
```

与你项目的联系： "文件传输系统中实现了简单的Buffer对象池，预先创建100个Buffer对象复用，避免频繁new/delete导致的内存碎片。"

七、常见陷阱

Q16: 以下代码有什么问题?

```
std::vector<int> vec{1, 2, 3, 4, 5};
for (auto it = vec.begin(); it != vec.end(); it++) {
    if (*it == 3) {
        vec.erase(it);
    }
}
```

答案:

erase后迭代器失效，继续使用会导致未定义行为。

正确写法:

```
for (auto it = vec.begin(); it != vec.end(); ) {
    if (*it == 3) {
        it = vec.erase(it);  // erase返回下一个有效迭代器
    } else {
        ++it;
    }
}
```

Q17: 以下代码有什么问题?

```
std::thread t([]{
    std::cout << "Hello" << std::endl;
});
```

答案：

线程对象离开作用域时，如果没有join()或detach()，会调用std::terminate()终止程序。

正确写法：

```
std::thread t([]{
    std::cout << "Hello" << std::endl;
});
t.join();  // 或 t.detach();
```

八、快速复习清单

STL必知必会

- [ ] vector扩容机制
- [ ] map vs unordered_map
- [ ] 迭代器失效场景

C++11/14必知必会

- [ ] 智能指针（shared_ptr/unique_ptr/weak_ptr）
- [ ] 右值引用和移动语义
- [ ] lambda表达式捕获方式

内存管理必知必会

- [ ] new/delete vs malloc/free
- [ ] 内存泄漏检测方法
- [ ] RAII原则

多线程必知必会

- [ ] mutex/lock_guard/unique_lock
- [ ] 条件变量使用
- [ ] 读写锁场景
- [ ] 线程安全单例
- [ ] 线程池原理

面试技巧

1. **结合项目经验回答**：不要只说理论，要联系自己的实践

2. **说明使用场景**：不同技术适用不同场景，展现权衡能力
3. **坦承不足**：不确定的地方可以说"我的理解是XX，不确定对不对"
4. **主动延伸**：回答完可以补充相关知识，展现知识广度

记住：面试官不是要你答对每道题，而是看你的思维方式和学习能力！