更多C/C++相关视频：[程序员老廖的个人空间-程序员老廖个人主页-哔哩哔哩视频 (bilibili.com)]

0. 需求

1. 项目功能概述

这是一个高性能的多线程任务队列系统，主要提供以下功能：

1. 支持多个命名任务队列的创建和管理
2. 支持异步任务的提交和执行
3. 支持延迟任务的调度
4. 保证任务的FIFO（先进先出）顺序执行
5. 支持多线程并发提交任务
6. 线程安全的任务管理

项目源地址：

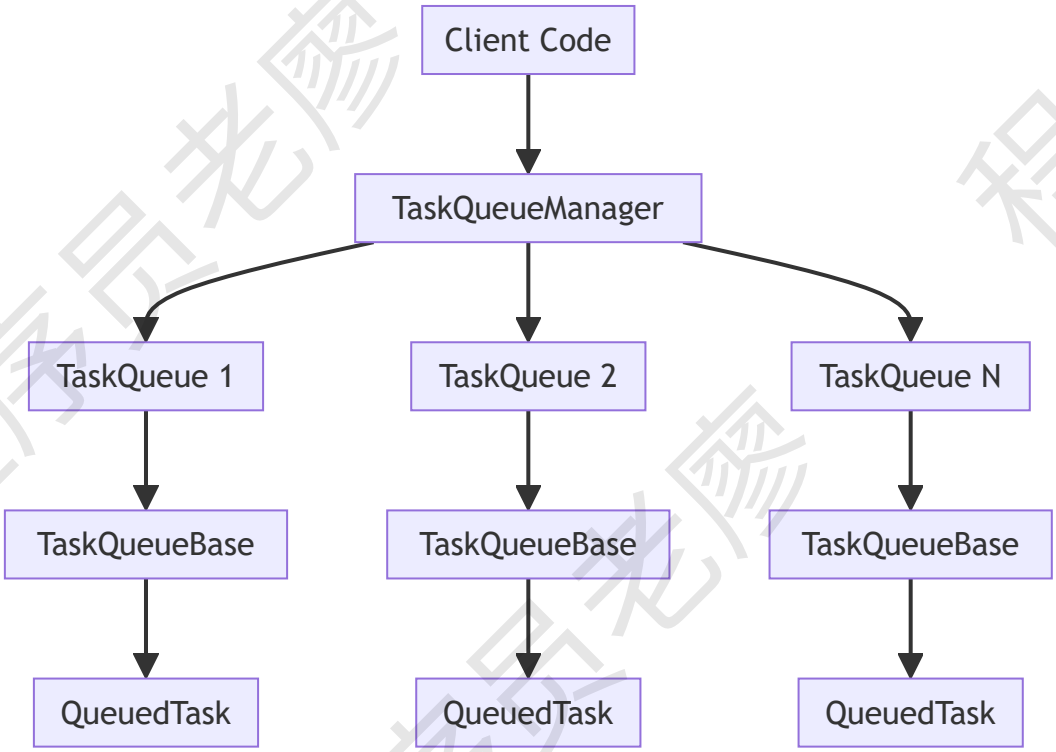
2 整体设计思路

1. **单例模式设计**
    - TaskQueueManager 采用单例模式
    - 使用 std::once_flag 确保线程安全的初始化
    - 通过全局宏 TQMgr 提供便捷访问
2. **线程安全考虑**
    - 使用互斥锁保护队列映射表
    - 所有对队列表的操作都是线程安全的
    - 使用智能指针管理资源生命周期
3. **任务队列管理**
    - 使用 unordered_map 存储命名任务队列
    - 支持动态创建和管理多个队列

- 每个队列都是独立的执行单元

3. 核心组件设计

3.1 系统架构图



不同的队列使用的线程是独立的，可以根据不同的业务投递到对应的队列，比如有些队列专门执行耗时的任务。

3.2 核心组件说明

TaskQueueManager

- 全局单例管理器
- 管理多个命名任务队列
- 提供队列的创建、获取、检查等功能
- 线程安全的队列管理

TaskQueue

- 任务队列的高层封装
- 提供任务提交接口
- 支持普通任务和延迟任务
- 支持Lambda表达式和自定义任务

TaskQueueBase

- 任务队列的底层实现
- 管理任务的实际执行
- 保证任务的FIFO顺序

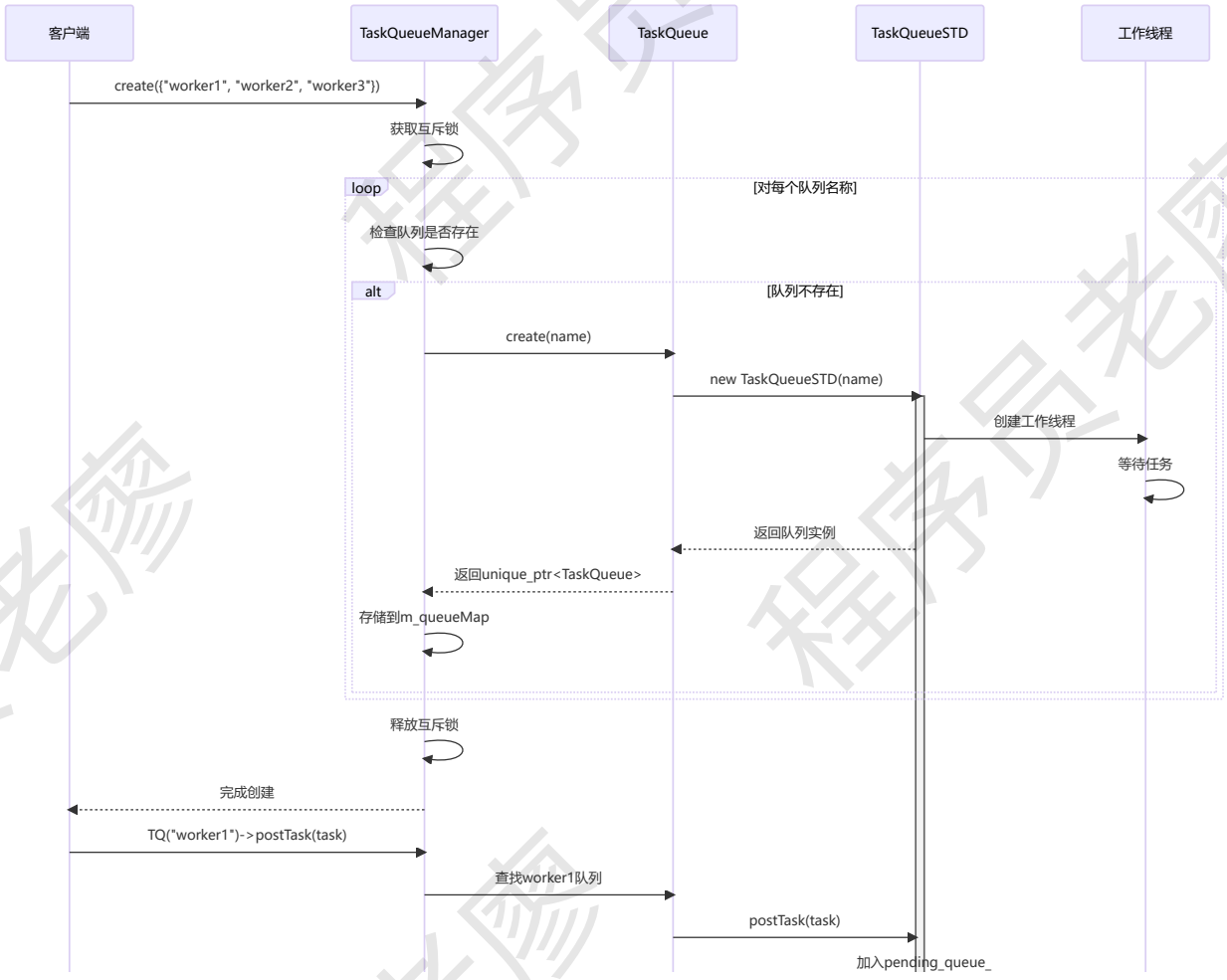
QueuedTask

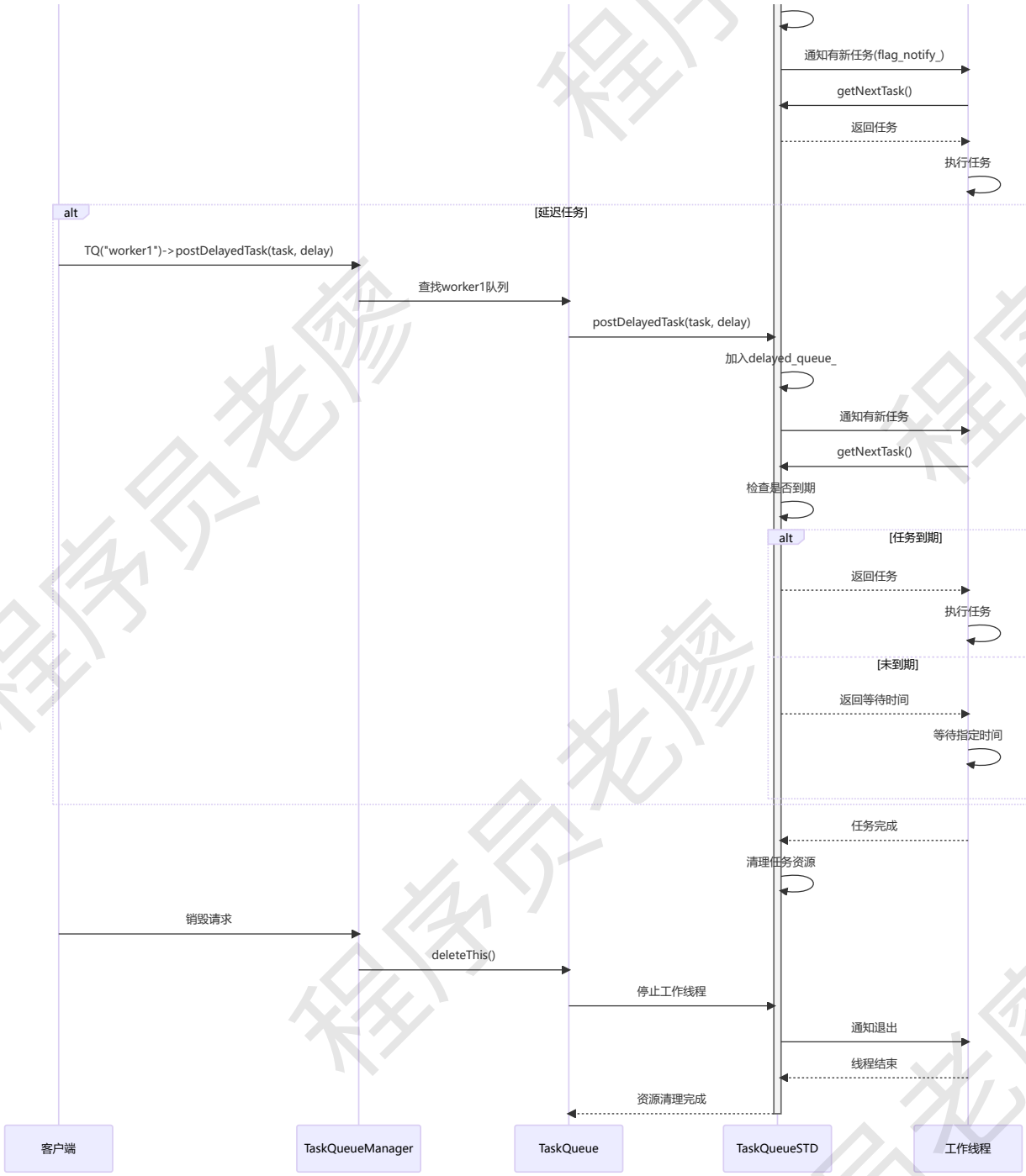
- 任务的基类
- 支持自定义任务实现
- 提供任务执行接口

TaskQueueSTD

- 单独的工作线程
- 支持即时任务和延迟任务
- FIFO（先进先出）执行顺序
- 基于事件的任务通知机制
- 线程安全的任务管理

4. 工作流程图

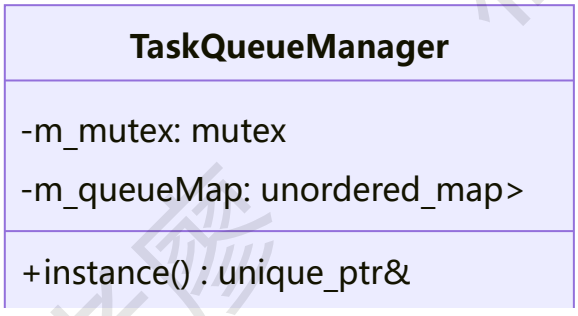

客户端
TaskQueueManager
TaskQueue
TaskQueueSTD
工作线程
create({"worker1", "worker2", "worker3"})
获取互斥锁
loop
[对每个队列名称]
检查队列是否存在
alt
[队列不存在]
create(name)
new TaskQueueSTD(name)
创建工作线程
等待任务
返回队列实例
返回unique_ptr<TaskQueue>
存储到m_queueMap
释放互斥锁
完成创建
TQ("worker1")->postTask(task)
查找worker1队列
postTask(task)
加入pending_queue_

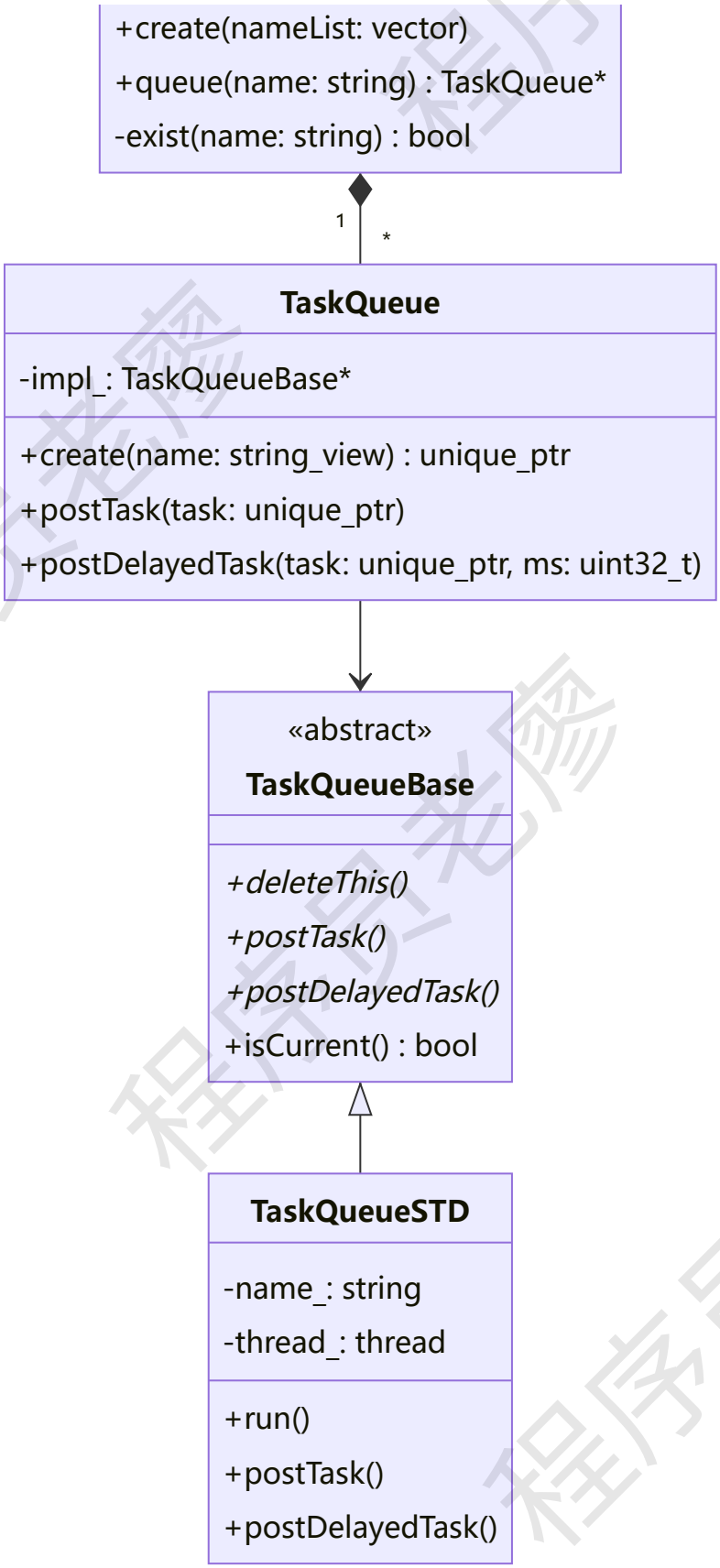

通知有新任务(flag_notify_)
getNextTask()
返回任务
执行任务
alt
[延迟任务]
TQ("worker1")->postDelayedTask(task, delay)
查找worker1队列
postDelayedTask(task, delay)
加入delayed_queue_
通知有新任务
getNextTask()
检查是否到期
alt
[任务到期]
返回任务
执行任务
[未到期]
返回等待时间
等待指定时间
任务完成
清理任务资源
销毁请求
deleteThis()
停止工作线程
通知退出
线程结束
资源清理完成
客户端
TaskQueueManager
TaskQueue
TaskQueueSTD
工作线程

TaskQueue实际是调用TaskQueueSTD。

```
std::unique_ptr<TaskQueue> TaskQueue::create(std::string_view name) {
    return std::make_unique<TaskQueue>(std::unique_ptr<TaskQueueBase,
TaskQueueDeleter>(new TaskQueueSTD(name)));
}
```

5. 类关系图

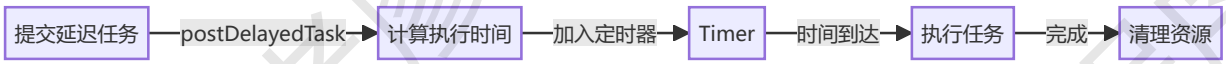



+create(nameList: vector)
+queue(name: string) : TaskQueue*
-exist(name: string) : bool
1
*
TaskQueue
-impl_: TaskQueueBase*
+create(name: string_view) : unique_ptr
+postTask(task: unique_ptr)
+postDelayedTask(task: unique_ptr, ms: uint32_t)
«abstract»
TaskQueueBase
+deleteThis()
+postTask()
+postDelayedTask()
+isCurrent() : bool
TaskQueueSTD
-name_: string
-thread_: thread
+run()
+postTask()
+postDelayedTask()

6. 关键特性实现

6.1 任务提交

6.2 延迟任务处理



7. 使用示例

```
// 1. 创建任务队列  这里创建了3个任务队列
TQMgr->create({"worker1", "worker2", "worker3"});

// 2. 提交普通任务
TQ("worker1")->postTask([](){
    cout << "执行普通任务" << endl;
});

// 3. 提交延迟任务
TQ("worker2")->postDelayedTask([](){
    cout << "执行延迟任务" << endl;
}, 1000);  // 1000ms后执行
```

更复杂的使用示例，重复执行某个任务，到达执行次数结束执行。

```
// 这里创建 一个新任务，重复执行三次，每次间隔1秒，即是没到三次时重新提交，先封装一个任务
    std::atomic<int> k{0};
    std::function<void()> task;   //显示声明
    task  = [&ev, &k, &task](){
        cout << "执行任务次数: " << ", k = " << ++k << endl;

         if (k < 3) {
            TQ("worker1")->postDelayedTask(task, 1000);
        } else {
            cout << "重复执行结束: " << ", k = " << k << endl;
            ev.set();
        }
    };

    TQ("worker1")->postDelayedTask(task, 1000);
```

8 关键实现细节

8.1 任务队列创建

1. **TaskQueueManager 单例实现**

```
std::unique_ptr<TaskQueueManager>& TaskQueueManager::instance() {
    static std::unique_ptr<TaskQueueManager> _instance = nullptr;
    static std::once_flag ocf;
    std::call_once(ocf, [](){
        _instance.reset(new TaskQueueManager());
    });
    return _instance;
}
```

2. **队列创建流程**

```
void TaskQueueManager::create(const std::vector<std::string>& nameList) {
    std::unique_lock<std::mutex> lock(m_mutex);
    for (const auto& name : nameList) {
        if (!exist(name)) {
            m_queueMap[name] = TaskQueue::create(name);
        }
    }
}
```

3. **具体队列创建**

```
std::unique_ptr<TaskQueue> TaskQueue::create(std::string_view name) {
    return std::make_unique<TaskQueue>(
        std::unique_ptr<TaskQueueBase, TaskQueueDeleter>(
            new TaskQueueSTD(name)
        )
    );
}
```

8.2 TaskQueueSTD 的实现原理

8.2.1 核心设计思路

TaskQueueSTD 是一个基于标准 C++ 实现的任务队列，具有以下特点：

1. 单独的工作线程
2. 支持即时任务和延迟任务
3. FIFO（先进先出）执行顺序
4. 基于事件的任务通知机制
5. 线程安全的任务管理

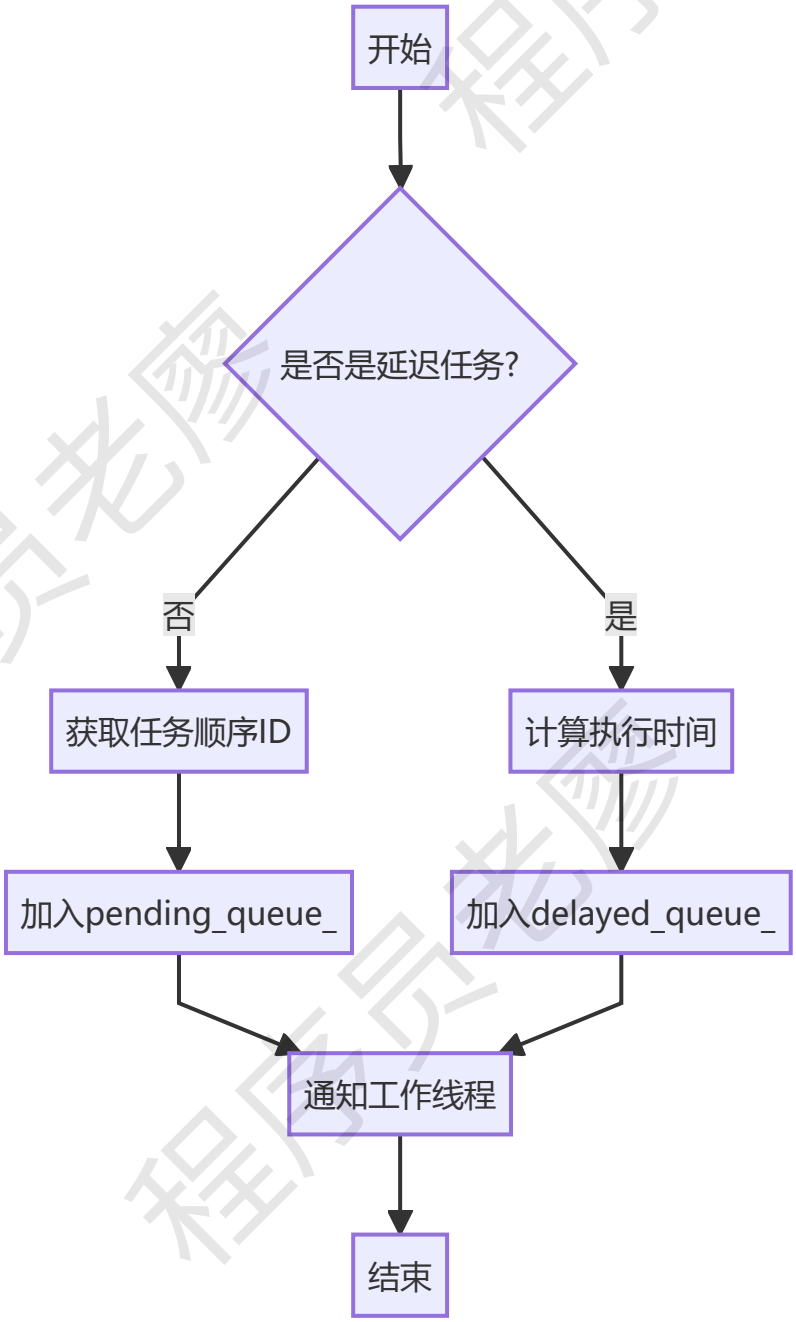
8.2.2 数据结构设计

8.2.3 关键组件实现

任务队列初始化

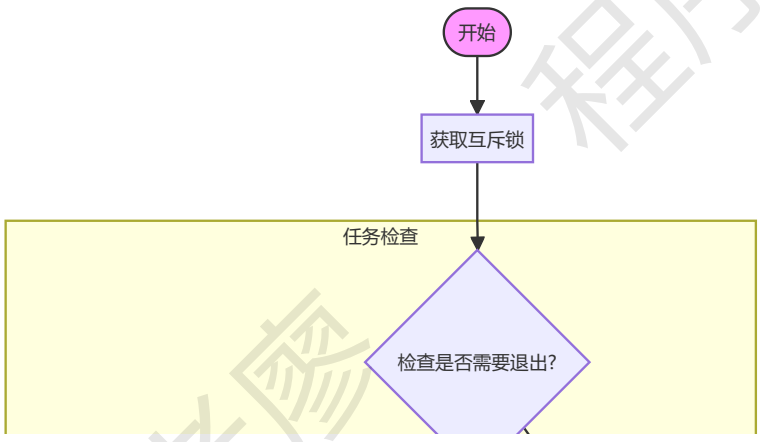
```
TaskQueueSTD::TaskQueueSTD(std::string_view queueName)
    : started_(false, false)
    , stopped_(false, false)
    , flag_notify_(false, false)
    , name_(queueName) {
    thread_ = std::thread([this]{
        CurrentTaskQueueSetter setCurrent(this);
        this->processTasks();
    });
    started_.wait(vi::Event::kForever);
}
```

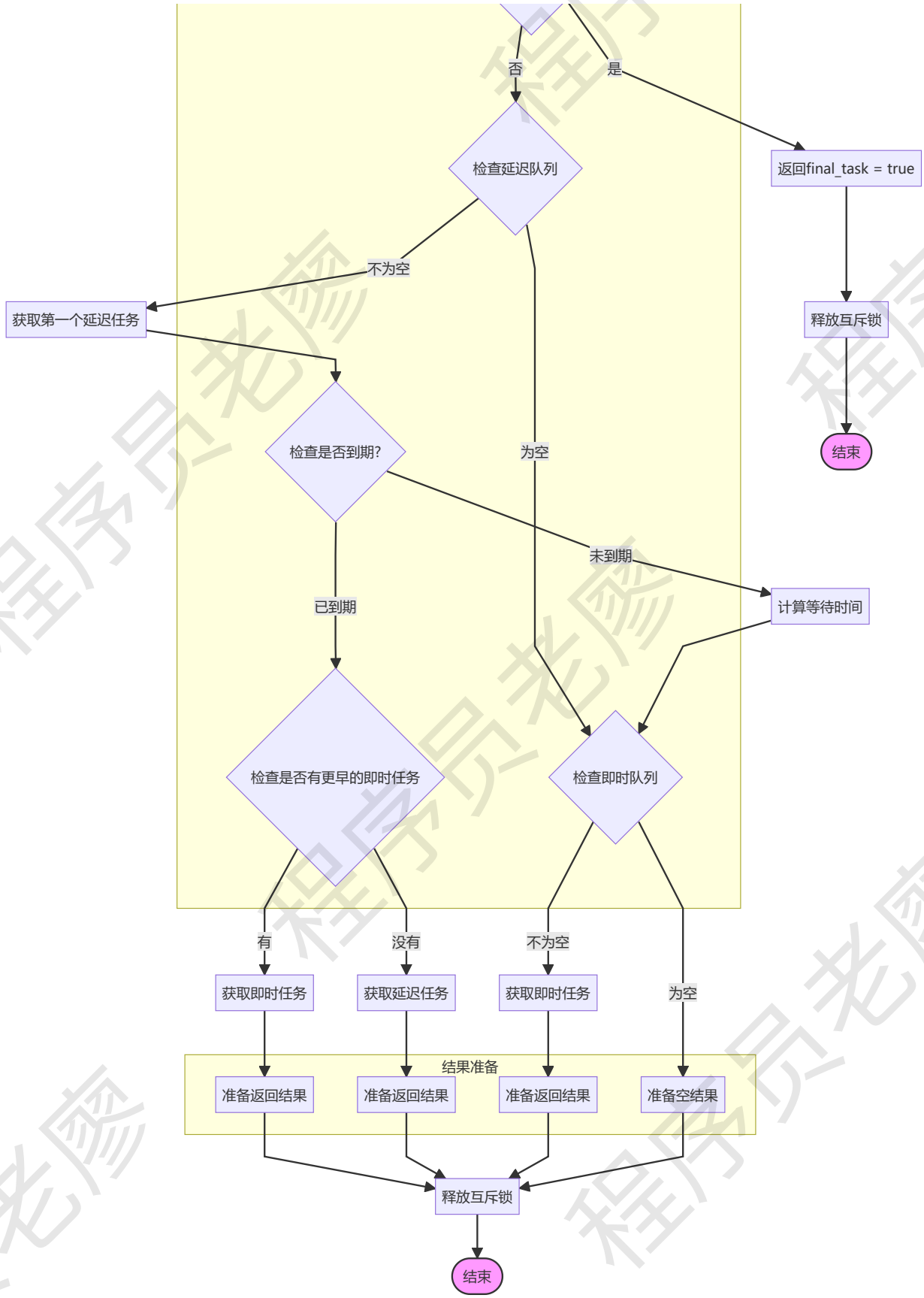
任务提交流程



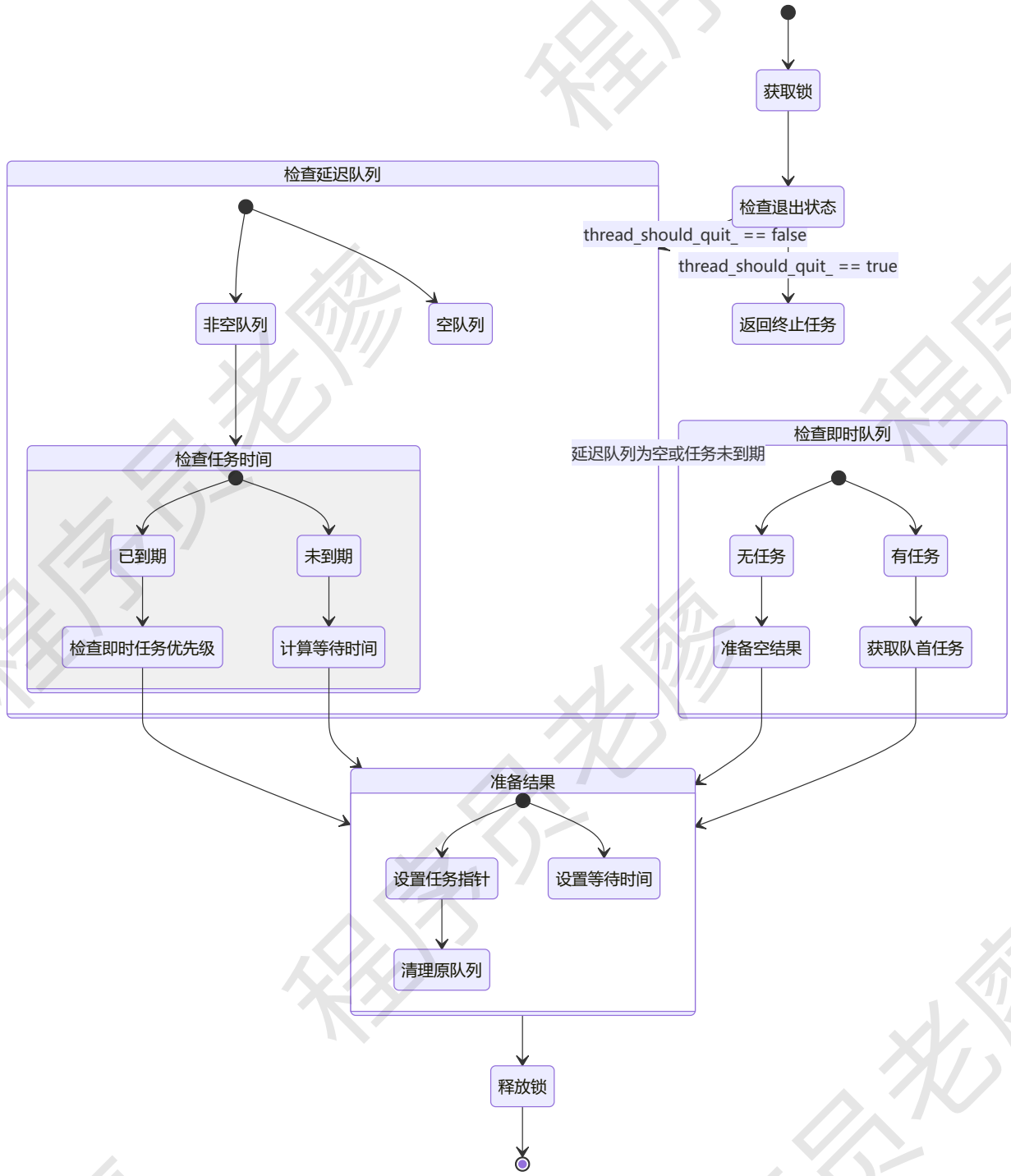
任务执行流程

askQueueSTD::getNextTask 的流程图：




否
是
检查延迟队列
返回final_task = true
不为空
获取第一个延迟任务
释放互斥锁
检查是否到期?
结束
为空
未到期
已到期
计算等待时间
检查是否有更早的即时任务
检查即时队列
有
没有
不为空
为空
获取即时任务
获取延迟任务
获取即时任务
结果准备
准备返回结果
准备返回结果
准备返回结果
准备空结果
释放互斥锁
结束

更详细的状态流转图：



关键处理逻辑说明：

1. **初始检查**
   - 获取互斥锁保护共享资源
   - 检查是否需要退出
   - 初始化结果结构
2. **延迟任务处理**
   - 检查延迟队列是否为空
   - 获取最早的延迟任务

- 检查任务是否到期
- 计算等待时间

3. **优先级处理**
   - 检查是否有更早的即时任务
   - 根据任务顺序ID决定执行顺序
   - 确保任务的FIFO顺序
4. **即时任务处理**
   - 检查即时队列是否为空
   - 获取队首任务
   - 准备返回结果
5. **结果准备**
   - 设置任务指针
   - 设置等待时间
   - 清理原队列中的任务
6. **资源清理**
   - 释放互斥锁
   - 返回结果

8.2.4 线程安全机制

1. **互斥锁保护**

```
std::mutex pending_mutex_;  // 保护任务队列访问
```

2. **事件同步**

```
vi::Event started_;      // 线程启动事件
vi::Event stopped_;      // 线程停止事件
vi::Event flag_notify_;  // 任务通知事件
```

9. 本项目C++新特性使用要点

现代C++特性在项目中的使用原因和优势：

`std::unique_ptr` 的使用

```
void postTask(std::unique_ptr<QueuedTask> task);
```

使用原因：

- 明确所有权语义：任务队列接管任务的所有权
- 防止内存泄漏：智能指针自动管理资源释放
- 避免共享所有权：任务只能被一个队列持有和执行
- 强制移动语义：不能意外地复制任务
- 零开销抽象：性能与原始指针相当

`std::move` 的使用

```
pending_queue_.push(std::pair<OrderId, std::unique_ptr<QueuedTask>>(order,
std::move(task)));
```

使用原因：

- 避免不必要的复制：直接转移资源所有权
- 提高性能：减少内存分配和复制操作
- 配合 `unique_ptr`：实现资源的转移
- 确保资源安全：防止多次释放同一资源

`std::forward` 的使用

```
template <class Closure>
void postTask(Closure&& closure) {
    postTask(ToQueuedTask(std::forward<Closure>(closure)));
}
```

使用原因：

- 完美转发：保持参数的值类别（左值/右值）
- 支持通用引用：使模板更灵活
- 优化性能：避免不必要的复制
- 类型推导：支持不同类型的任务封装

`string_view` 的使用

```
TaskQueueSTD(std::string_view queueName);
```

使用原因：

- 性能优化：避免字符串复制
- 灵活性：可以接受字符串字面量、`std::string` 等多种类型
- 只读访问：明确表明不会修改字符串
- 零开销抽象：不会产生额外的内存分配

其他现代C++特性的使用

a. Lambda表达式

```
thread_ = std::thread([this]{
    CurrentTaskQueueSetter setCurrent(this);
    this->processTasks();
});
```

- 简化代码：方便地创建临时函数对象
- 捕获上下文：访问外部变量
- 提高可读性：就地定义行为

b. RAII模式

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

- 资源安全：自动管理资源的生命周期
- 异常安全：确保资源正确释放
- 简化代码：避免手动加锁解锁

c. 类型推导（`auto`）

```
auto tick = milliseconds();
```

- 简化代码：避免冗长的类型声明
- 维护性：类型变化时无需修改代码
- 泛型编程：支持模板和泛型算法

总体设计意图

1. **安全性**：
   - 使用智能指针避免内存泄漏
   - 强制所有权语义防止资源误用
   - RAII确保资源正确管理
2. **性能**：
   - 移动语义避免不必要的复制
   - `string_view`减少字符串开销
   - 零开销抽象保持高效性
3. **可维护性**：
   - 现代C++特性提高代码可读性
   - 类型安全减少错误
   - 自动化资源管理简化代码

4. **灵活性**:
    - 模板和完美转发支持多种任务类型
    - Lambda表达式简化任务定义
    - 通用引用支持不同参数类型

这些特性的使用体现了现代C++的最佳实践，既保证了代码的安全性和性能，又提高了可维护性和灵活性。