



>



Select an Agent
to view its workspace

管理知识库

Knowledge Base Off

Seed 选择模式现在设置为 "Off"，你的知识库将不会被使用。

Reasoning Process >

这是一个非常典型的C++面向对象设计与多线程编程的综合题目。根据你提供的图片（项目3-轻量级多任务调度器），我将为你编写一个完整的C++解决方案。

为了确保代码可以方便地在一个文件中编译运行，同时满足所有设计模式（工厂、命令、策略、观察者）和功能要求（文件压缩、矩阵计算、HTTP请求、殚窗、日志等），我将使用 C++17 标准（用于文件系统和多线程）以及 Windows API（用于实现 Task D 的殚窗和 Task A/C 的系统调用）。

设计思路与架构映射

1. UI 层：使用控制台（Console）模拟菜单界面，充当 UI。
2. Observer 模式：`UI` 类订阅 `TaskScheduler`，当任务完成或有日志时，`Scheduler` 通知 UI 刷新显示。
3. Command/Strategy 模式：`ITask` 是接口，具体任务（如 `BackupTask`）是策略，`ScheduledTask` 封装了执行时间，充当命令。
4. Factory 模式：`TaskFactory` 根据用户输入创建具体的任务对象。
5. Singleton 模式：`TaskScheduler` 全局唯一。
6. RAII：`LogWriter` 利用构造函数打开文件，析构函数关闭文件。

完整代码 (main.cpp)

你可以将以下代码保存为 `main.cpp`，在 Windows 环境下使用 Visual Studio 或 MinGW (g++) 编译。

```
cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <thread>
#include <functional>
#include <fstream>
#include <filesystem>
#include <memory>
#include <chrono>
#include <ctime>
#include <iomanip>
#include <sstream>
#include <fstream>
#include <random>
#include <atomic>
#include <map>
#include <windows.h> // 用于 Task D 惰窗和 Sleep

// 使用 C++17 文件系统
namespace fs = std::filesystem;
using namespace std;

// =====
// 1. 日志模块 (RAII & 单例辅助)
// =====
class LogWriter {
private:
    ofstream logfile;
    mutex logMutex;
    LogWriter() {
        logfile.open("scheduler.log", ios::app);
        if (logfile.is_open()) {
            logfile << "--- Log System Started ---\n";
        }
    }
public:
    ~LogWriter() {
        if (logfile.is_open()) {
            logfile << "--- Log System Closed ---\n";
            logfile.close();
        }
    }
    // Meyers Singleton
    static LogWriter& Instance() {
        static LogWriter instance;
        return instance;
    }
    void Write(const string& msg) {
        lock_guard<mutex> lock(logMutex);
        auto now = chrono::system_clock::to_time_t(chrono::system_clock::now());
        if (logfile.is_open()) {
            logfile << put_time(localtime(&now), "[%Y-%m-%d %H:%M:%S] ") << msg << endl;
            logfile.flush(); // 确保写入
        }
    }
};

// =====
// 2. 观察者模式 (Observer Interface)
// =====
class IOObserver {
public:
    virtual void OnTaskEvent(const string& message) = 0;
    virtual ~IOObserver() = default;
};

// =====
// 3. 任务接口与策略 (Strategy Pattern)
// =====
class ITask {
public:
    virtual string GetName() const = 0;
    virtual void Execute() = 0;
    virtual ~ITask() = default;
};

// --- 具体任务实现 ---
// Task A: 文件备份 (模拟压缩)
class TaskBackup : public ITask {
public:
    string GetName() const override { return "Task A: File Backup"; }
    void Execute() override {
        // 模拟压缩操作，实际可用 system("tar -cf ...") 或 powershell
        LogWriter::Instance().Write("Starting Backup of C:\\Data...");

        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(2));

        // 实际上我们创建一个模拟的 zip 文件
        ofstream outfile("backup_data.zip");
        outfile << "Compressed Data Content";
        outfile.close();

        LogWriter::Instance().Write("Backup completed: Saved to backup_data.zip");
    }
};

// Task B: 矩阵乘法
class TaskMatrix : public ITask {
public:
    string GetName() const override { return "Task B: Matrix Multiplication"; }
    void Execute() override {
        auto start = chrono::high_resolution_clock::now();

        // 200x200 矩阵乘法
        int N = 200;
        vector<vector<double>> A(N, vector<double>(N, 1.0));
        vector<vector<double>> B(N, vector<double>(N, 2.0));
        vector<vector<double>> C(N, vector<double>(N, 0.0));

        for(int i=0; i<N; ++i)
            for(int j=0; j<N; ++j)
                for(int k=0; k<N; ++k)
                    C[i][j] += A[i][k] * B[k][j];

        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        cout << "Time taken: " << duration.count() << " seconds" << endl;
    }
};

// Task C: HTTP GET
class TaskHttp : public ITask {
public:
    string GetName() const override { return "Task C: HTTP GET"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task D: 惰窗 (FIFO Queue)
class TaskFifo : public ITask {
public:
    string GetName() const override { return "Task D: FIFO Queue"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task E: 线程池 (ThreadPool)
class TaskThreadPool : public ITask {
public:
    string GetName() const override { return "Task E: ThreadPool"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task F: 定时任务 (Scheduled Task)
class TaskScheduled : public ITask {
public:
    string GetName() const override { return "Task F: Scheduled Task"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task G: 命令 (Command)
class TaskCommand : public ITask {
public:
    string GetName() const override { return "Task G: Command"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task H: 策略 (Strategy)
class TaskStrategy : public ITask {
public:
    string GetName() const override { return "Task H: Strategy"; }
    void Execute() override {
        // 模拟耗时
        this_thread::sleep_for(chrono::seconds(1));
    }
};

// Task I: 观察者 (Observer)
class TaskObserver : public IOObserver {
public:
    void OnTaskEvent(const string& message) {
        cout << "Observer received message: " << message << endl;
    }
};

// Task J: 工厂 (Factory)
class TaskFactory {
public:
    ITask* CreateTask(string name) {
        if (name == "Task A") {
            return new TaskBackup();
        } else if (name == "Task B") {
            return new TaskMatrix();
        } else if (name == "Task C") {
            return new TaskHttp();
        } else if (name == "Task D") {
            return new TaskFifo();
        } else if (name == "Task E") {
            return new TaskThreadPool();
        } else if (name == "Task F") {
            return new TaskScheduled();
        } else if (name == "Task G") {
            return new TaskCommand();
        } else if (name == "Task H") {
            return new TaskStrategy();
        } else if (name == "Task I") {
            return new TaskObserver();
        } else {
            return nullptr;
        }
    }
};

// Task J: 调度器 (Scheduler)
class TaskScheduler {
private:
    queue<ITask*> tasks;
    IOObserver* observer;
public:
    TaskScheduler(IOObserver* observer) {
        this->observer = observer;
    }

    void AddTask(ITask* task) {
        tasks.push(task);
    }

    void Run() {
        while (!tasks.empty()) {
            ITask* task = tasks.front();
            task->Execute();
            tasks.pop();
        }
    }
};

// Main Function
int main() {
    IOObserver* observer = new TaskObserver();
    TaskScheduler scheduler(observer);

    TaskFactory factory;
    ITask* taskA = factory.CreateTask("Task A");
    ITask* taskB = factory.CreateTask("Task B");
    ITask* taskC = factory.CreateTask("Task C");
    ITask* taskD = factory.CreateTask("Task D");
    ITask* taskE = factory.CreateTask("Task E");
    ITask* taskF = factory.CreateTask("Task F");
    ITask* taskG = factory.CreateTask("Task G");
    ITask* taskH = factory.CreateTask("Task H");
    ITask* taskI = factory.CreateTask("Task I");

    scheduler.AddTask(taskA);
    scheduler.AddTask(taskB);
    scheduler.AddTask(taskC);
    scheduler.AddTask(taskD);
    scheduler.AddTask(taskE);
    scheduler.AddTask(taskF);
    scheduler.AddTask(taskG);
    scheduler.AddTask(taskH);
    scheduler.AddTask(taskI);

    scheduler.Run();
}

// End of main.cpp
```

开启新的话题

◇ 常规模式 ◆ GPT 4.1 ▲

x1 ▲

Nanobanana Pro is free



Flower-soil

订阅与设置

TRIAL

+ - ⊞ 器 ⊙ ⊜

+

