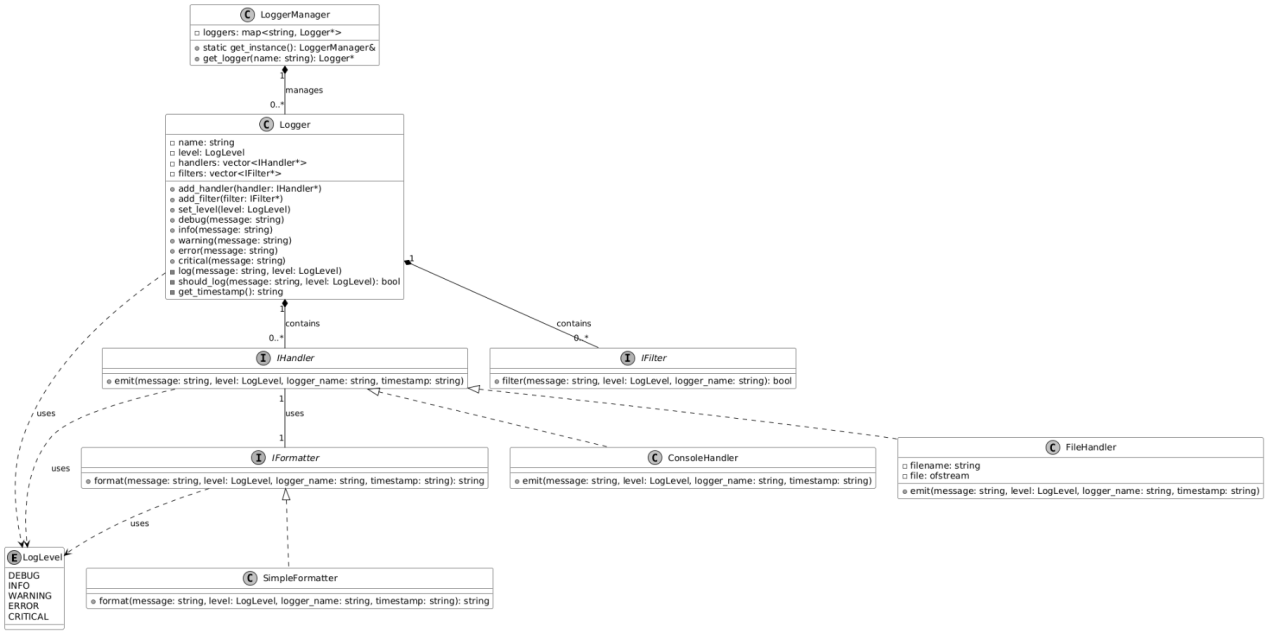
**日志设计**

**1.日志等级的设定：**

enum LogLevel { DEBUG = 0, INFO, WARNING, ERROR, FATAL, UNKNOWN };



**2.格式化器的设计与实现：**

首先：格式化器的接口（包含格式化方法format）

// 日志格式化器接口

class IFormatter {

public:

virtual ~IFormatter() = default;

virtual std::string format(const std::string& message, LogLevel level,

const std::string& logger\_name,

const std::string& timestamp) = 0;

};

设计一个简易的格式化器类：

// 简易格式化器

// example: [2025:06:16][error][hostscan.log]: "errmessage"

class SimpleFormatter : public IFormatter {

public:

std::string format(const std::string& message, LogLevel level,

const std::string& logger\_name,

const std::string& timestamp) override

{

std::string level\_str;

switch (level) {

case LogLevel::DEBUG:

level\_str = "DEBUG";

break;

case LogLevel::INFO:

level\_str = "INFO";

break;

case LogLevel::ERROR:

level\_str = "ERROR";

break;

case LogLevel::FATAL:

level\_str = "FATAL";

break;

case LogLevel::UNKNOWN:

level\_str = "UNKNOWN";

break;

}

return "[" + timestamp + "]" + "[" + level\_str + "]" + "[" + logger\_name +

"]" + ": \"" + message + "\"";

}

};

**3.输出处理器的设计与实现：**

首先，规定处理器的接口（包含输出方法emit）：

class IHandler {

protected:

std::shared\_ptr<IFormatter> formatter;

public:

IHandler(std::shared\_ptr<IFormatter> \_formatter) : formatter(\_formatter) {}

virtual ~IHandler() = default;

virtual void emit(const std::string& message, LogLevel level,

const std::string& logger\_name,

const std::string& timestamp) = 0;

};

然后具体实现两个输出处理器：控制台输出、文件输出：

class ConsoleHandler : public IHandler {

public:

ConsoleHandler(std::shared\_ptr<IFormatter> \_formatter)

: IHandler(\_formatter) {}

void emit(const std::string& message, LogLevel level,

const std::string& logger\_name,

const std::string& timestamp) override {

std::string formatted =

formatter->format(message, level, logger\_name, timestamp);

// 根据日志级别选择输出流

if (level == LogLevel::ERROR || level == LogLevel::FATAL ||

level == LogLevel::UNKNOWN) {

std::cerr << formatted << std::endl;

} else {

std::cout << formatted << std::endl;

}

}

};

// 文件输出目标

class FileHandler : public IHandler {

private:

std::ofstream file;

std::string filename;

public:

FileHandler(const std::string \_filename,

std::shared\_ptr<IFormatter> \_formatter)

: IHandler(\_formatter), filename(\_filename) {

file.open(filename, std::ios::app);

if (!file.is\_open()) {

std::cerr << "无法打开日志文件：" << filename << std::endl;

}

}

~FileHandler() {

if (file.is\_open()) {

file.close();

}

}

void emit(const std::string& message, LogLevel level,

const std::string& logger\_name,

const std::string& timestamp) override {

if (!file.is\_open()) return;

std::string formatted =

formatter->format(message, level, logger\_name, timestamp);

file << formatted << std::endl;

}

};

**4.日志过滤器的设计与实现**

// 过滤器接口

class IFilter {

public:

virtual ~IFilter() = default;

virtual bool filter(const std::string& message, LogLevel level,

const std::string& logger\_name) = 0;

};

**5.日志记录类的实现：**

包含属性：

日志名string:name、日志等级LogLevel:level、

处理器序列vector<shared\_ptr<IHandler>>:handlers、

过滤器序列vector<shared\_ptr<IFilter>>:filters、

并发控制互斥锁mutex:mutex

私有方法：

当前获取时间戳：string get\_timestamp()const{}

检测是否应该记入日志：bool should\_log(const string& message, LogLevel level){}

实际记录的日志：void log(const string& message, LogLevel level){}

公共方法：

添加处理器：void add\_handler();

添加过滤器：void add\_filter();

设置日志级别：void set\_level();

日志记录方法：

void debug();

void info();

void warning();

void error();

void fatal();

void unknown();

// 日志记录器

class Logger {

private:

// 获取当前时间戳

std::string get\_timestamp() const {

auto now = std::time(nullptr);

char buf[80];

std::strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", std::localtime(&now));

return std::string(buf);

}

// 检测是否应该记录该日志

bool should\_log(const std::string& message, LogLevel level) const {

if (level < this->level) return false;

for (const auto& filter : filters) {

// 如果被其中一个过滤器过滤了,就不输出

if (!filter->filter(message, level, name)) return false;

}

return true;

}

// 实际记录的日志

void log(const std::string& message, LogLevel level) {

if (!should\_log(message, level)) return;

std::string timestamp = get\_timestamp();

// 使用互斥锁保证线程安全

{

std::lock\_guard<std::mutex> lock(mutex);

for (const auto& handler : handlers) {

handler->emit(message, level, name, timestamp);

}

}

}

public:

Logger(const std::string& \_name, LogLevel \_level = LogLevel::INFO)

: name(\_name), level(\_level) {}

// 添加处理器

void add\_handler(std::shared\_ptr<IHandler> handler) {

handlers.push\_back(handler);

}

// 添加过滤器

void add\_filter(std::shared\_ptr<IFilter> filter) {

filters.push\_back(filter);

}

// 设置日志级别

void set\_level(LogLevel level) { this->level = level; }

// 日志记录方法

void debug(const std::string& message) { log(message, LogLevel::DEBUG); }

void info(const std::string& message) { log(message, LogLevel::INFO); }

void warning(const std::string& message) { log(message, LogLevel::WARNING); }

void error(const std::string& message) { log(message, LogLevel::ERROR); }

void fatal(const std::string& message) { log(message, LogLevel::FATAL); }

void unknown(const std::string& message) { log(message, LogLevel::UNKNOWN); }

private:

std::string name;

LogLevel level;

std::vector<std::shared\_ptr<IHandler>> handlers;

std::vector<std::shared\_ptr<IFilter>> filters;

static std::mutex mutex;

};

std::mutex Logger::mutex;//静态成员外部定义

**6.日志管理器的设计与实现**

// 日志管理器-单例模式

class LoggerManager {

public:

static LoggerManager& get\_instance() { return instance; }

std::shared\_ptr<Logger> get\_logger(const std::string& name) {

auto it = loggers.find(name);

if (it != loggers.end()) return it->second;

auto new\_logger = std::make\_shared<Logger>(name);

loggers[name] = new\_logger;

return new\_logger;

}

private:

static LoggerManager instance;

std::map<std::string, std::shared\_ptr<Logger>> loggers;

LoggerManager() = default;

LoggerManager(const LoggerManager&) = delete;

LoggerManager& operator=(const LoggerManager&) = delete;

};

LoggerManager LoggerManager::instance;

**7.使用示例**

int main() {

// 获取日志记录器

auto logger = LoggerManager::get\_instance().get\_logger("main");

// 设置日志级别

logger->set\_level(LogLevel::DEBUG);

// 添加控制台处理器

auto console\_formatter = std::make\_shared<SimpleFormatter>();

auto console\_handler = std::make\_shared<ConsoleHandler>(console\_formatter);

logger->add\_handler(console\_handler);

// 添加文件处理器

auto file\_formatter = std::make\_shared<SimpleFormatter>();

auto file\_handler = std::make\_shared<FileHandler>("app.log", file\_formatter);

logger->add\_handler(file\_handler);

// 记录不同级别的日志

logger->debug("这是一条调试信息");

logger->info("这是一条普通信息");

logger->warning("这是一条警告信息");

logger->error("这是一条错误信息");

logger->fatal("这是一条严重错误信息");

logger->unknown("这是一条未知错误信息");

return 0;

}