BY:

zhaoyihang873@gmail.com

总结：

1.单例实现Logger实例全局唯一

2.由时间戳来翻滚日志目录

3.异步IO日志系统，用多线程和无锁队列来实现

（1）日志线程从无锁队列获取处理信息，其他线程将要处理信息放到无锁队列

（2）日志类的析构函数不要忘记回收线程资源

（3）日志信息从类中分离定义到结构体中

4.线程安全的设计

无锁队列模板类使用了 C++11 中的原子操作，实现了一个线程安全的队列

5.崩溃安全的设计

注册一个 SIGSEGV 信号处理函数，在程序崩溃时记录堆栈信息，并将其写入日志文件，handle_sigsegv 函数用于处理 SIGSEGV 信号，调用 log_stack_trace 函数记录堆栈信息，最后调用 std::exit 函数退出程序

1.用枚举来定义日志等级

```
public:
    enum Level
    {
        DEBUG = 0,
        INFO,
        WARN,
        ERROR,
        FATAL,
        LEVEL_COUNT
    };
```

```
const char* Logger::s_level[LEVEL_COUNT] =
{
    "DEBUG",
    "INFO",
    "WARN",
    "ERROR",
    "FATAL"
};
```

2.用单例模式来设计Logger类

1.将默认方法声明到private里面

```
private:
    Logger();
    ~Logger();
```

在**类外**面定义

```
Logger::Logger() : m_max(0), m_len(0), m_level(DEBUG)
{
}

Logger::~Logger()
{
    close();
}
```

2.声明一个私有的静态指针来指向唯一的实例

```
private:
    static Logger *m_instance;
```

在**类外**面初始化这个静态指针

```
Logger *Logger::m_instance = NULL;
```

3.声明一个公共的方法来获取到这个实例

```
public:
    static Logger* instance();
```

定义这个公共方法

```
Logger* Logger::instance()
{
    if (m_instance == NULL)
        m_instance = new Logger();
    return m_instance;
}
```

3.设计文件流--写，翻滚

std::ofstream的seekp() 函数

`std::ofstream` 是 C++ 标准库中用于写入文件的输出流类。它提供了 `seekp()` 函数，用于设置文件指针的位置。

`seekp()` 函数有两个参数：

1. `pos`：要设置的文件指针的位置。
2. `mode`：指定偏移量的起始位置，可以是 `std::ios::beg`（文件开头）、`std::ios::cur`（当前位置）或 `std::ios::end`（文件末尾）。

`seekp()` 函数可以用于以下操作：

1. 移动文件指针到指定位置。
   例如，使用 `seekp(10, std::ios::beg)` 将文件指针移动到文件开头后的第10个字节处。
2. 获取文件指针的当前位置。
   例如，使用 `seekp(0, std::ios::cur)` 获取文件指针的当前位置。
3. 将文件指针移动到文件末尾，以便在文件末尾附加数据。
   例如，使用 `seekp(0, std::ios::end)` 将文件指针移动到文件末尾。

在使用 `seekp()` 函数之前，需要先打开文件并创建 `std::ofstream` 对象。例如：

```
std::ofstream outfile("filename.txt");

// 将文件指针移动到文件末尾
outfile.seekp(0, std::ios::end);

// 在文件末尾写入数据
outfile << "Hello, world!" << std::endl;

// 关闭文件
outfile.close();
```

在上面的代码中，我们首先打开一个名为 "filename.txt" 的文件，并创建一个 `std::ofstream` 对象。然后，我们使用 `seekp()` 函数将文件指针移动到文件末尾，并在文件末尾写入数据。最后，我们关闭文件。

`tellp()` 函数是 `std::ofstream` 类的成员函数

用于返回输出流的当前位置，也就是输出指针的当前位置。它没有参数，返回一个 `std::streampos` 类型的值，表示当前位置相对于流的开头的偏移量。

以下是一个使用 `tellp()` 函数的示例：

```
#include <iostream>
#include <fstream>

int main() {
  std::ofstream outfile("example.txt");
  outfile << "Hello, world!" << std::endl;
  //可以将 std::streampos 视为一种类似于 long 或 long long 的整数类型。
  std::streampos pos = outfile.tellp();
  std::cout << "Current position: " << pos << std::endl;
  outfile.close();
  return 0;
}
```

在上面的示例中，我们首先创建了一个名为 `example.txt` 的输出文件流对象 `outfile`，然后在文件中写入了一些数据。接着，我们使用 `tellp()` 函数获取输出指针的当前位置，并将其存储在 `pos` 变量中。最后，我们关闭文件流并输出当前位置。

需要注意的是，`tellp()` **函数返回的偏移量是一个相对于流的开头的值。**如果您需要将输出指针移动到指定的位置，可以使用 `seekp()` 函数。

每次open都需要获取已有的长度为了方便翻滚

```
void Logger::open(const string &filename)
{
    m_filename = filename;
    m_fout.open(filename, ios::app);
    if (m_fout.fail())
    {
        throw std::logic_error("open log file failed: " + filename);
    }
    m_fout.seekp(0, ios::end);
    std::streampos m_len = m_fout.tellp();
}
```

```
if (m_max > 0 && m_len >= m_max)
    {
        rotate();
    }
```

实现翻滚

```
void Logger::rotate()
{
    close();
    time_t ticks = time(NULL);
    //线程安全
```

```
    struct tm* ptm;
    localtime_r(&ticks, ptm);

    char timestamp[32];
    memset(timestamp, 0, sizeof(timestamp));
    strftime(timestamp, sizeof(timestamp), ".%Y-%m-%d_%H-%M-%S", ptm);
    string filename = m_filename + timestamp;
    if (rename(m_filename.c_str(), filename.c_str()) != 0)
    {
        throw std::logic_error("rename log file failed: " +
string(strerror(errno)));
    }
    open(m_filename);
}
```

`time(NULL)` 函数返回从 1970 年 1 月 1 日 00:00:00 UTC 到当前时间的秒数，也称为 Unix 时间戳。然后将该时间戳传递给 `localtime()` 函数，该函数将其转换为本地时间，并将结果存储在 `struct tm` 结构体中。

`struct tm` 结构体包含以下字段：

```
struct tm {
    int tm_sec;   // 秒 (0-60)
    int tm_min;   // 分 (0-59)
    int tm_hour;  // 时 (0-23)
    int tm_mday;  // 日 (1-31)
    int tm_mon;   // 月 (0-11)
    int tm_year;  // 年 (自 1900 起的年数)
    int tm_wday;  // 星期几 (0-6，0 表示星期日)
    int tm_yday;  // 自新年以来的天数 (0-365)
    int tm_isdst; // 是否为夏令时 (正数表示是，0 表示不是，负数表示不确定)
};
```

在上述代码中，`ptm` 变量是一个指向 `struct tm` 结构体的指针，它指向存储当前本地时间的结构体。您可以使用 `ptm` 指针来访问 `struct tm` 结构体中的各个字段，以获取有关当前本地时间的更多信息。

注意

`localtime()` 函数返回的指针指向的结构体是静态分配的，因此不应该尝试释放它，也不应该同时使用多个指向该结构体的指针。此外，`localtime()` 函数是非线程安全的，如果您需要在多个线程中使用它，请考虑使用 `localtime_r()` 函数或其他线程安全的日期和时间函数。

时间戳转换函数

`strftime()` 函数是 C 标准库中的一个函数，用于将时间戳格式化为指定格式的字符串。它的函数原型如下：

Copy

```
size_t strftime(char* str, size_t maxsize, const char* format, const struct tm*
timeptr);
```

参数说明：

- `str`：指向一个字符数组的指针，用于存储格式化后的字符串。
- `maxsize`：`str` 指向的字符数组的最大长度。
- `format`：指向一个以 `%` 开头的格式字符串，用于指定输出格式。
- `timeptr`：指向一个 `struct tm` 结构体的指针，包含要格式化的时间信息。

`strftime()` 函数将 `timeptr` 指向的 `struct tm` 结构体中的时间信息，按照 `format` 字符串指定的格式进行格式化，并将结果存储在 `str` 指向的字符数组中。函数返回值是生成的字符串长度（不包括空字符）。

`format` 字符串中的 `%` 后面可以跟一些特定的字符，用于指定输出格式。下面是一些常用的格式字符：

- `%Y`：年份，如 2022。
- `%m`：月份，如 04。
- `%d`：日期，如 27。
- `%H`：24 小时制的小时数，如 23。
- `%M`：分钟数，如 59。
- `%S`：秒数，如 30。
- `%a`：星期几的缩写，如 Mon。
- `%A`：星期几的全称，如 Monday。
- `%b`：月份的缩写，如 Apr。
- `%B`：月份的全称，如 April。
- `%c`：完整的日期和时间，如 Mon Apr 27 23:59:30 2022。
- `%p`：上午或下午，如 AM 或 PM。
- `%r`：12 小时制的时间，如 11:59:30 PM。
- `%x`：日期，如 04/27/22。
- `%X`：时间，如 23:59:30。

下面是一个示例代码：

```
#include <stdio.h>
#include <time.h>

int main() {
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    char buffer[80];
    strftime(buffer, 80, "Now it's %I:%M%p.", tm_info);
    printf("%s\n", buffer);
    return 0;
}
```

在上面的示例代码中，我们首先使用 `time()` 函数获取当前时间的时间戳，然后使用 `localtime()` 函数将时间戳转换为本地时间，并将结果存储在 `struct tm` 结构体 `tm_info` 中。接着，我们使用 `strftime()` 函数将 `tm_info` 指向的时间信息格式化为指定格式的字符串，将结果存储在 `buffer` 数组中。最后，我们使用 `printf()` 函数输出 `buffer` 数组中的字符串。

重命名函数

需要翻滚的时候重命名

`rename()` 函数是 C 标准库中的一个函数，用于重命名文件或将文件移动到另一个目录下。它的函数原型如下：

Copy

```
int rename(const char* oldname, const char* newname);
```

参数说明：

- `oldname`：指向一个以 null 结尾的字符串，表示要重命名或移动的文件的原始名称。
- `newname`：指向一个以 null 结尾的字符串，表示要将文件重命名或移动到的新名称或新路径。

`rename()` 函数将通过 `oldname` 指定的文件重命名为 `newname` 所指向的新名称。如果 `newname` 中包含了路径信息，那么文件将被移动到指定的路径下。如果 `newname` 中不包含路径信息，那么文件将被重命名为指定的新名称。

`rename()` 函数返回值为 0 表示操作成功，否则表示操作失败。可能的错误情况包括文件不存在、权限不足等。

下面是一个示例代码：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int ret = rename("oldfile.txt", "newfile.txt");
    if (ret != 0) {
        printf("Error renaming file.\n");
        exit(EXIT_FAILURE);
    }
    printf("File renamed successfully.\n");
    return 0;
}
```

在上面的示例代码中，我们使用 `rename()` 函数将名为 `oldfile.txt` 的文件重命名为 `newfile.txt`。如果函数返回值不为 0，说明操作失败，我们将输出错误信息并退出程序，否则输出操作成功的信息。

4.设计key函数log

```
void Logger::log(Level level, const char* file, int line, const char* format,
...)
{
    if (m_level > level)
    {
        return;
    }

    if (m_fout.fail())
    {
        throw std::logic_error("open log file failed: " + m_filename);
```

```
    }

    time_t ticks = time(NULL);
    struct tm* ptm = localtime(&ticks);
    char timestamp[32];
    memset(timestamp, 0, sizeof(timestamp));
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", ptm);

    int len = 0;
    const char * fmt = "%s %s %s:%d ";
    //获取len的小技巧
    len = snprintf(NULL, 0, fmt, timestamp, s_level[level], file, line);
    if (len > 0)
    {
        char * buffer = new char[len + 1];
        snprintf(buffer, len + 1, fmt, timestamp, s_level[level], file, line);
        buffer[len] = 0;
        m_fout << buffer;
        delete buffer;
        m_len += len;
    }

    va_list arg_ptr;
    va_start(arg_ptr, format);
    len = vsnprintf(NULL, 0, format, arg_ptr);
    va_end(arg_ptr);
    if (len > 0)
    {
        char * content = new char[len + 1];
        va_start(arg_ptr, format);
        vsnprintf(content, len + 1, format, arg_ptr);
        va_end(arg_ptr);
        content[len] = 0;
        m_fout << content;
        delete content;
        m_len += len;
    }

    m_fout << "\n";
    m_fout.flush();

    if (m_max > 0 && m_len >= m_max)
    {
        rotate();
    }
}
```

---

snprintf格式化写入函数

`snprintf` 是一个 C 标准库中的函数，用于格式化字符串并将其写入缓冲区中。它可以避免缓冲区溢出问题，因为它允许你指定要写入的最大字符数。如果格式化后的字符串超过了指定的最大字符数，`snprintf` 会根据指定的最大字符数截断字符串，并在缓冲区的末尾添加一个空字符。

`snprintf` 的函数原型如下：

```
int snprintf(char *str, size_t size, const char *format, ...);
```

其中：

- `str`：要写入的缓冲区的地址。
- `size`：缓冲区的最大字符数。
- `format`：格式化字符串。
- `...`：可变参数，用于指定要格式化的值。

`snprintf` 的返回值是写入缓冲区的字符数，不包括空字符。如果写入的字符数超过了指定的最大字符数，返回值将等于指定的最大字符数。

以下是一个使用 `snprintf` 的示例：

```
#include <cstdio>

int main() {
    char buffer[128];
    int num = 42;
    float f = 3.14f;
    const char* str = "Hello, world!";

    int count = snprintf(buffer, sizeof(buffer), "num = %d, f = %f, str = %s",
num, f, str);
    if (count >= sizeof(buffer)) {
        // 缓冲区溢出，字符串被截断
        printf("Output truncated.\n");
    }
    printf("%s\n", buffer);

    return 0;
}
```

在这个示例中，我们定义了一个缓冲区 `buffer`，并使用 `snprintf` 格式化字符串并将其写入缓冲区中。我们使用 `%d`、`%f` 和 `%s` 格式说明符分别代表整数、浮点数和字符串。我们还使用可变参数 `num`、`f` 和 `str` 分别提供要格式化的值。在 `snprintf` 调用后，我们检查写入的字符数是否超过了缓冲区的大小，如果超过了，则打印一条消息。最后，我们打印缓冲区中的字符串。

输出应该类似于以下内容：

```
num = 42, f = 3.140000, str = Hello, world!
```

`vsnprintf` 使用一个 `va_list` 类型的参数代替 `snprintf` 函数中的可变参数列表

`vsnprintf` 是一个 C 标准库函数，与 `snprintf` 函数相似，也可以格式化字符串并将其写入缓冲区中。不同之处在于，`vsnprintf` 使用一个 `va_list` 类型的参数代替 `snprintf` 函数中的可变参数列表。这使得它可以与 `va_start`、`va_arg` 和 `va_end` 等函数一起使用，以便于处理可变参数列表。

`vsnprintf` 的函数原型如下：

```
int vsnprintf(char* str, size_t size, const char* format, va_list ap);
```

其中：

- `str`：要写入的缓冲区的地址。
- `size`：缓冲区的最大字符数。
- `format`：格式化字符串。
- `ap`：指向 `va_list` 的指针。

`vsnprintf` 的返回值是写入缓冲区的字符数，不包括空字符。如果写入的字符数超过了指定的最大字符数，返回值将等于指定的最大字符数。

以下是一个使用 `vsnprintf` 的示例：

```
#include <cstdio>
#include <cstdarg>

int main() {
    char buffer[128];
    const char* format = "num = %d, f = %f, str = %s";
    int num = 42;
    float f = 3.14f;
    const char* str = "Hello, world!";

    va_list args;
    va_start(args, format);
    int count = vsnprintf(buffer, sizeof(buffer), format, args);
    va_end(args);

    if (count >= sizeof(buffer)) {
        // 缓冲区溢出，字符串被截断
        printf("Output truncated.\n");
    }
    printf("%s\n", buffer);

    return 0;
}
```

在这个示例中，我们使用 `vsnprintf` 格式化字符串并将其写入缓冲区中。我们使用 `%d`、`%f` 和 `%s` 格式说明符分别代表整数、浮点数和字符串。我们使用 `va_list` 类型的变量 `args` 代替了 `snprintf` 中的可变参数列表，并使用 `va_start`、`va_arg` 和 `va_end` 函数处理可变参数列表。在 `vsnprintf` 调用后，我们检查写入的字符数是否超过了缓冲区的大小，如果超过了，则打印一条消息。最后，我们打印缓冲区中的字符串。

输出应该类似于以下内容：

```
num = 42, f = 3.140000, str = Hello, world!
```

5.实现异步IO，线程安全和崩溃安全

可以使用多线程和无锁队列来实现。具体地，可以创建一个专门的日志线程，该线程负责将日志写入磁盘文件，而其他线程只需要将日志信息放入一个无锁队列中，由日志线程来处理即可。

```
#include <iostream>
#include <fstream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <chrono>

// 日志级别枚举类
enum class LogLevel {
    INFO,
    WARNING,
    ERROR
};

// 日志信息结构体
struct LogMessage {
    LogLevel level;     // 日志级别
    std::string message;    // 日志内容
    std::chrono::system_clock::time_point timestamp;    // 日志时间
};
//////////////////////////////////////////////////////
// 无锁队列模板类
template<typename T>
class LockFreeQueue {
public:
    LockFreeQueue() : head_(nullptr), tail_(nullptr) {}
    ~LockFreeQueue() {
        while (head_) {
            Node* tmp = head_;
            head_ = tmp->next_;
            delete tmp;
        }
    }
    void push(const T& value) {
        Node* node = new Node(value);
        Node* prev_tail = tail_.exchange(node, std::memory_order_acq_rel);
        if (prev_tail) {
            prev_tail->next_ = node;
        } else {
            head_ = node;
        }
    }
    bool pop(T& value) {
        Node* prev_head = head_.load(std::memory_order_relaxed);
        while (prev_head) {
            Node* next = prev_head->next_;
            if (head_.compare_exchange_weak(prev_head, next,
std::memory_order_release, std::memory_order_relaxed)) {
                value = prev_head->value_;
                delete prev_head;
                return true;
            }
        }
```

```
        return false;
    }

private:
    struct Node {
        T value_;
        Node* next_;
        Node(const T& value) : value_(value), next_(nullptr) {}
    };
    std::atomic<Node*> head_;
    std::atomic<Node*> tail_;
};

// 日志类
class Logger {
public:
    Logger() : stop_(false) {
        // 创建日志线程
        thread_ = std::thread([this]() {
            run();
        });
    }
    ~Logger() {
        // 停止日志线程
        stop_ = true;
        cv_.notify_all();
        thread_.join();
    }
    void log(LogLevel level, const std::string& message) {
        LogMessage log_message = { level, message,
std::chrono::system_clock::now() };
        queue_.push(log_message);
        cv_.notify_one();
    }

private:
    void run() {
        // 打开日志文件
        std::ofstream log_file("log.txt", std::ios::app);
        if (!log_file) {
            std::cerr << "Failed to open log file" << std::endl;
            return;
        }
        // 循环处理日志信息
        while (!stop_) {
            LogMessage log_message;
            if (queue_.pop(log_message)) {
                // 将日志信息写入文件
                std::string level_str;
                switch (log_message.level) {
                case LogLevel::INFO:
                    level_str = "INFO";
                    break;
                case LogLevel::WARNING:
                    level_str = "WARNING";
                    break;
                case LogLevel::ERROR:
                    level_str = "ERROR";
```

```
                    break;
                }
                auto time =
std::chrono::system_clock::to_time_t(log_message.timestamp);
                log_file << "[" << level_str << "] "
                         << std::ctime(&time) << " "
                         << log_message.message << std::endl;
            } else {
                // 队列为空，等待通知
                std::unique_lock<std::mutex> lock(mutex_);
                cv_.wait(lock);
            }
        }
        // 关闭日志文件
        log_file.close();
    }

    LockFreeQueue<LogMessage> queue_;   // 日志信息队列
    std::thread thread_;    // 日志线程
    std::mutex mutex_;  // 互斥量
    std::condition_variable cv_;    // 条件变量
    bool stop_; // 是否停止标志
};

// 全局日志对象
Logger g_logger;

// 记录堆栈信息的函数
void log_stack_trace() {
    // TODO: 实现堆栈信息记录
    g_logger.log(LogLevel::ERROR, "Stack trace not implemented");
}

// 捕获 SIGSEGV 信号的处理函数
void handle_sigsegv(int sig) {
    g_logger.log(LogLevel::ERROR, "Caught SIGSEGV signal");
    log_stack_trace();
    std::exit(sig);
}

int main() {
    // 注册 SIGSEGV 信号处理函数
    std::signal(SIGSEGV, handle_sigsegv);

    // 输出日志信息
    g_logger.log(LogLevel::INFO, "Starting program");

    // 模拟多线程输出日志
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back([i]() {
            for (int j = 0; j < 10; ++j) {
                g_logger.log(LogLevel::INFO, "Thread " + std::to_string(i) + "
message " + std::to_string(j));
            }
        });
    }
    for (auto& thread : threads) {
```

```
        thread.join();
    }

    // 输出日志信息
    g_logger.log(LogLevel::INFO, "Program finished");

    return 0;
}
```