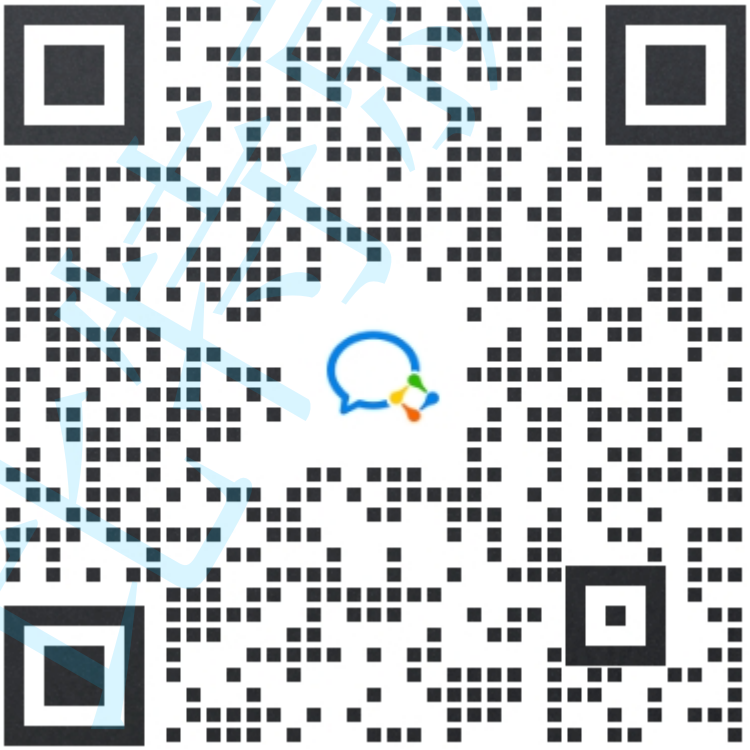
Spdlog 日志组件的安装及使用

版权说明



代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

介绍：

spdlog 是一个高性能、超快速、零配置的 C++ 日志库，它旨在提供简洁的 API 和丰富的功能，同时保持高性能的日志记录。它支持多种输出目标、格式化选项、线程安全以及异步日志记录。以下是对 spdlog 的详细介绍和使用方法。

github 链接：https://github.com/gabime/spdlog

特点

- **高性能**：spdlog 专为速度而设计，即使在高负载情况下也能保持良好的性能。
- **零配置**：无需复杂的配置，只需包含头文件即可在项目中使用。
- **异步日志**：支持异步日志记录，减少对主线程的影响。
- **格式化**：支持自定义日志消息的格式化，包括时间戳、线程 ID、日志级别等。
- **多平台**：跨平台兼容，支持 Windows、Linux、macOS 等操作系统。
- **丰富的 API**：提供丰富的日志级别和操作符重载，方便记录各种类型的日志。

安装

命令安装：

```
C++
dev@dev-host:~/workspace/spdlog$ sudo apt-get install libspdlog-
dev
```

原码安装：

```
Bash
dev@dev-host:~/workspace/spdlog$ git clone
https://github.com/gabime/spdlog.git

dev@dev-host:~/workspace/spdlog$ cd spdlog/
dev@dev-host:~/workspace/spdlog$ mkdir build && cd build
dev@dev-host:~/workspace/spdlog$ cmake -
```

```
DCMAKE_INSTALL_PREFIX=/usr ..
dev@dev-host:~/workspace/spdlog$ make && sudo make install
```

使用

包含头文件：

在你的 C++ 源文件中包含 spdlog 的头文件：

```
C++
#include <spdlog/spdlog.h>
```

日志输出等级枚举

```
C++
namespace level {
    enum level_enum : int {
        trace = SPDLOG_LEVEL_TRACE,
        debug = SPDLOG_LEVEL_DEBUG,
        info = SPDLOG_LEVEL_INFO,
        warn = SPDLOG_LEVEL_WARN,
        err = SPDLOG_LEVEL_ERROR,
        critical = SPDLOG_LEVEL_CRITICAL,
        off = SPDLOG_LEVEL_OFF,
        n_levels
    };

}
```

日志输出格式自定义：

可以自定义日志消息的格式：

```
C++
logger->set_pattern("%Y-%m-%d %H:%M:%S [%t] [%-7l] %v");

%t - 线程 ID（Thread ID）。
%n - 日志器名称（Logger name）。
%l - 日志级别名称（Level name），如 INFO, DEBUG, ERROR 等。
%v - 日志内容（message）。
```

```
%Y - 年（Year）。
%m - 月（Month）。
%d - 日（Day）。
%H - 小时（24-hour format）。
%M - 分钟（Minute）。
%S - 秒（Second）。
```

日志记录器类：

创建一个基本的日志记录器，并设置日志级别和输出模式：

```
C++
namespace spdlog {
class logger {
    logger(std::string name);
    logger(std::string name, sink_ptr single_sink)
    logger(std::string name, sinks_init_list sinks)
    void set_level(level::level_enum log_level);
    void set_formatter(std::unique_ptr<formatter> f);
    template<typename... Args>
    void trace(fmt::format_string<Args...> fmt, Args &&...args)
    template<typename... Args>
    void debug(fmt::format_string<Args...> fmt, Args &&...args)
    template<typename... Args>
    void info(fmt::format_string<Args...> fmt, Args &&...args)
    template<typename... Args>
    void warn(fmt::format_string<Args...> fmt, Args &&...args)
    template<typename... Args>
    void error(fmt::format_string<Args...> fmt, Args &&...args)
    template<typename... Args>
    void critical(fmt::format_string<Args...> fmt, Args &&...args)

    void flush(); //刷新日志
    //策略刷新--触发指定等级日志的时候立即刷新日志的输出
    void flush_on(level::level_enum log_level);
};
```

异步日志记录类：

为了异步记录日志，可以使用 `spdlog::async_logger`：

```C++

class async_logger final : public logger {
    async_logger(std::string logger_name,
                 sinks_init_list sinks_list,
                 std::weak_ptr<details::thread_pool> tp,
                 async_overflow_policy overflow_policy =
async_overflow_policy::block);
    async_logger(std::string logger_name,
                 sink_ptr single_sink,
                 std::weak_ptr<details::thread_pool> tp,
                 async_overflow_policy overflow_policy =
async_overflow_policy::block);

    //异步日志输出需要异步工作线程的支持，这里是线程池类
    class SPDLOG_API thread_pool {
        thread_pool(size_t q_max_items,
            size_t threads_n,
            std::function<void()> on_thread_start,
            std::function<void()> on_thread_stop);
        thread_pool(size_t q_max_items, size_t threads_n,
            std::function<void()> on_thread_start);
        thread_pool(size_t q_max_items, size_t threads_n);
    };
}

std::shared_ptr<spdlog::details::thread_pool> thread_pool() {
    return details::registry::instance().get_tp();
}

//默认线程池的初始化接口
inline void init_thread_pool(size_t q_size, size_t thread_count)

auto async_logger = spdlog::async_logger_mt("async_logger",
"logs/async_log.txt");
async_logger->info("This is an asynchronous info message");
```

日志记录器工厂类：

```C++
using async_factory =
async_factory_impl<async_overflow_policy::block>;
```

```
template<typename Sink, typename... SinkArgs>
inline std::shared_ptr<spdlog::logger> create_async(
    std::string logger_name,
    SinkArgs &&...sink_args)


//创建一个彩色输出到标准输出的日志记录器，默认工厂创建同步日志记录器
template<typename Factory = spdlog::synchronous_factory>
std::shared_ptr<logger> stdout_color_mt(
    const std::string &logger_name,
    color_mode mode = color_mode::automatic);
//标准错误
template<typename Factory = spdlog::synchronous_factory>
std::shared_ptr<logger> stderr_color_mt(
    const std::string &logger_name,
    color_mode mode = color_mode::automatic);
//指定文件
template<typename Factory = spdlog::synchronous_factory>
std::shared_ptr<logger> basic_logger_mt(
    const std::string &logger_name,
    const filename_t &filename,
    bool truncate = false,
    const file_event_handlers &event_handlers = {})
//循环文件
template<typename Factory = spdlog::synchronous_factory>
std::shared_ptr<logger> rotating_logger_mt(
    const std::string &logger_name,
    const filename_t &filename,
    size_t max_file_size,
    size_t max_files,
    bool rotate_on_open = false)
...
```

日志落地类

```
C++
namespace spdlog {
namespace sinks {
    class SPDLOG_API sink
    {
    public:
        virtual ~sink() = default;
        virtual void log(const details::log_msg &msg) = 0;
```

```
        virtual void flush() = 0;
        virtual void set_pattern(const std::string &pattern) = 0;
        virtual void
set_formatter(std::unique_ptr<spdlog::formatter> sink_formatter) =
0;

        void set_level(level::level_enum log_level);
    };

    using stdout_sink_mt;
    using stderr_sink_mt;
    using stdout_color_sink_mt;
    using stderr_color_sink_mt;
    //滚动日志文件-超过一定大小则自动重新创建新的日志文件
    sink_ptr rotating_file_sink(filename_t base_filename,
                        std::size_t max_size,
                        std::size_t max_files,
                        bool rotate_on_open = false,
                        const file_event_handlers &event_handlers =
{});
    using rotating_file_sink_mt = rotating_file_sink<std::mutex>;
    //普通的文件落地对啊 ing
    sink_ptr basic_file_sink(const filename_t &filename,
                        bool truncate = false,
                        const file_event_handlers &event_handlers =
{});
    using basic_file_sink_mt = basic_file_sink<std::mutex>;

    using kafka_sink_mt = kafka_sink<std::mutex>;
    using mongo_sink_mt = mongo_sink<std::mutex>;
    using tcp_sink_mt = tcp_sink<std::mutex>;
    using udp_sink_mt = udp_sink<std::mutex>;
    .....
    //*_st：单线程版本，不用加锁，效率更高。
    //*_mt：多线程版本，用于多线程程序是线程安全的。
}
}
```

全局接口：

```
C++
//输出等级设置接口
void set_level(level::level_enum log_level);
```

```
//日志刷新策略-每隔 N 秒刷新一次
void flush_every(std::chrono::seconds interval)
//日志刷新策略-触发指定等级立即刷新
void flush_on(level::level_enum log_level);
```

记录日志：

使用日志记录器记录不同级别的日志：

```C++
logger->trace("This is a trace message");
logger->debug("This is a debug message");
logger->info("This is an info message");
logger->warn("This is a warning message");
logger->error("This is an error message");
logger->critical("This is a critical message");
```

使用样例

```C++
#include <spdlog/spdlog.h>
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/async.h>

void multi_sink_example() {
    //创建一个标准输出的落地方向对象
    auto console_sink =
std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
    //该方向仅允许 warn 等级以上的日志输出
    console_sink->set_level(spdlog::level::warn);
    console_sink->set_pattern("[multi_sink_example] [%^%l%$] %v");
    //创建一个文件输出的落地方向对象
    auto file_sink =
std::make_shared<spdlog::sinks::basic_file_sink_mt>(
        "logs/multisink.txt", true);
    //该方向允许 trace 等级以上的日志输出
    file_sink->set_level(spdlog::level::trace);

    //构造一个日志器对象，用于输出日志
    spdlog::logger logger("multi_sink", {console_sink,
file_sink});
```

```
    //每个日志器都可以设置初步过滤等级，其次内部每个 sink 也可以设置自己独立的过滤等级
    logger.set_level(spdlog::level::debug);
    logger.set_pattern("%Y-%m-%d %H:%M:%S [%l] %v");
    logger.warn("this should appear in both console and file");
    logger.info("this message should not appear in the console, only in the file");
}

void async_example() {
    //void init_thread_pool(size_t q_size, size_t thread_count);
    spdlog::init_thread_pool(32768, 1);//设置默认线程池属性信息
    //通过工厂模式创建异步日志记录器的同时，会在内部创建默认线程池作为异步线程
    auto async_logger = spdlog::basic_logger_mt<spdlog::async_factory>(
        "async_file_logger", "logs/async_log.txt");
    async_logger->set_pattern("%Y-%m-%d %H:%M:%S [%l] %v");
    for (int i = 1; i < 101; ++i) {
        //需要注意的是，在多参数的时候，spdlog 并非使用 %s %d 这种通配符来匹配参数
        //而是使用 {}, spdlog 可以自己识别数据类型
        async_logger->info("Async message #{} {}", i, "hello");
    }
}

int main()
{
    //multi_sink_example();
    async_example();
    return 0;
}
```

```
C++
main : main.cc
    g++ -std=c++17 $^ -o $@ -lspdlog -lfmt
```

二次封装：

因为 spdlog 的日志输出对文件名和行号并不是很友好（也有可能是调研不到位...），以及因为 spdlog 本身实现了线程安全，如果使用默认日志器每次进行单例获取，效率会有降低，因此进行二次封装，简化使用。

- 日志的初始化封装接口
- 日志的输出接口封装宏

```C++
#include <spdlog/spdlog.h>
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/async.h>

namespace bite {
std::shared_ptr<spdlog::logger> g_logger;
void init_logger(const std::string &logger_name,
    const std::string &logger_file,
    spdlog::level::level_enum logger_level) {
    spdlog::flush_on(logger_level);
    spdlog::init_thread_pool(32768, 1);
    if (logger_file == "stdout")
        g_logger = spdlog::stdout_color_mt(logger_name);
    else
        g_logger =
spdlog::basic_logger_mt<spdlog::async_factory>(logger_name,
logger_file);
    g_logger->set_pattern("%H:%M:%S [%n][%-7l]%v");
    g_logger->set_level(logger_level);
}

#define DBG(format, ...) bite::g_logger-
>debug(std::string("[{:>10s}:{:<4d}] ")+format, __FILE__,
__LINE__, ##__VA_ARGS__);
#define INF(format, ...) bite::g_logger-
>info(std::string("[{:>10s}:{:<4d}] ")+format, __FILE__, __LINE__,
##__VA_ARGS__);
#define WRN(format, ...) bite::g_logger-
>warn(std::string("[{:>10s}:{:<4d}] ")+format, __FILE__, __LINE__,
##__VA_ARGS__);
#define ERR(format, ...) bite::g_logger-
>error(std::string("[{:>10s}:{:<4d}] ")+format, __FILE__,
__LINE__, ##__VA_ARGS__);

}
```

spdlog 与 glog 组件对比

glog 和 spdlog 都是流行的 C++ 日志库，它们各自具有不同的特点和优势。以下是对这两个库的对比分析，包括性能测试的结果和使用场景的考量。

glog

glog 是由 Google 开发的一个开源 C++ 日志库，它提供了丰富的日志功能，包括多种日志级别、条件日志记录、日志文件管理、信号处理、自定义日志格式等。glog 默认情况下是同步记录日志的，这意味着每次写日志操作都会阻塞直到日志数据被写入磁盘。

性能

根据张生荣的性能对比测试分析，glog 在同步调用的场景下的性能较 spdlog 慢。在一台低配的服务器上，glog 耗时 1.027 秒处理十万笔日志数据，而在固态硬盘上的耗时为 0.475 秒。

spdlog

spdlog 是一个开源的、高性能的 C++ 日志库，它支持异步日志记录，允许在不影响主线程的情况下进行日志写入。spdlog 旨在提供零配置的用户体验，只需包含头文件即可使用。它还支持多种输出目标、格式化选项和线程安全。

性能

在同样的性能测试中，spdlog 在同步调用的场景下比 glog 快。在低配服务器上的耗时为 0.135 秒，而在固态硬盘上的耗时为 0.057 秒。此外，spdlog 还提供了异步日志记录的功能，其简单异步模式的耗时为 0.158 秒。

对比总结

- **性能**：从性能测试结果来看，spdlog 在同步调用场景下的性能优于 glog。当涉及到大量日志数据时，spdlog 显示出更快的处理速度。
- **异步日志**：spdlog 支持异步日志记录，这在处理高负载应用程序时非常有用，可以减少日志操作对主线程的影响。
- **易用性**：spdlog 提供了更简单的集成和配置方式，只需包含头文件即可使用，而 glog 可能需要额外的编译和配置步骤。
- **功能**：glog 提供了一些特定的功能，如条件日志记录和信号处理，这些在某些场景下可能非常有用。

- **使用场景**：glog 可能更适合那些对日志性能要求不是特别高，但需要一些特定功能的场景。而 spdlog 则适合需要高性能日志记录和异步日志能力的应用程序。

在选择日志库时，开发者应根据项目的具体需求和性能要求来决定使用哪个库。如果项目对日志性能有较高要求，或者需要异步日志记录来避免阻塞主线程，spdlog 可能是更好的选择。如果项目需要一些特定的日志功能，或者已经在使用 glog 且没有显著的性能问题，那么继续使用 glog 也是合理的。

总结

spdlog 是一个功能强大且易于使用的 C++ 日志库，它提供了丰富的功能和高性能的日志记录能力。通过简单的 API，开发者可以快速地在项目中实现日志记录，同时保持代码的清晰和可维护性。无论是在开发阶段还是生产环境中，spdlog 都能提供稳定和高效的日志服务。