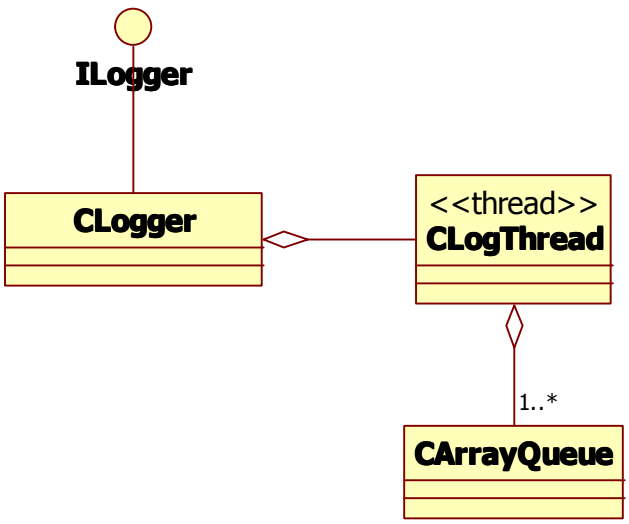
开源 mooon logger 介绍

1. 什么是 mooon logger？

logger 是 mooon 中 sys 库下的一个轻量的写本地日志工具，具备使用简单、结构简单、实现高效和线程安全（非多进程安全）等特点，主要功能如下：

1) 自动在日志行前添加日期、时间、线程号和日志级别；
2) 支持可变参数的文本日志和二进制日志；
3) 支持最大 64K 的一条日志（预先分配大小为 512 字节，这个值可设置）；
4) 支持 debug、info、error、warn 和 fatal 多种日志级别；
5) 支持独立的 trace 日志，方便程序调试，独立的开关控制跟踪日志，允许关闭其它级别的日志，只输出跟踪日志；
6) 日志级别可动态调整；
7) 支持同时将日志打印到标准输出；
8) 可控制是否自动追加换行符，如果行末尾已经有换行符，则不会添加；
9) 可控制是否自动追加结尾点号，如果行末尾已经有结尾点号或换行符，则不会添加；
10) 支持滚动日志，允许设置备份个数和单个日志文件大小。

2. 类结构



目前有两种 ILogger 的实现，一种是基于 log4cxx 的，对应于 plugin_log4cxx，另一个是 sys 库内置的 CLogger，本文只介绍 CLogger 的实现。如果你喜欢，可以直接以类的方式使用 CLogger，而不以接口的方式使用 ILogger。

3. ILogger 接口

```
/**
  * 日志接口
  */
class ILogger
{
public:
    /** 是否允许同时在标准输出上打印日志 */
    virtual void enable_screen(bool enabled) = 0;
    /** 是否允许跟踪日志，跟踪日志必须通过它来打开 */
    virtual void enable_trace_log(bool enabled) = 0;
    /** 是否自动在一行后添加结尾的点号，如果最后已经有点号，则不会再添加 */
    virtual void enable_auto_adddot(bool enabled) = 0;
    /** 是否自动添加换行符，如果已经有换行符，则不会再自动添加换行符 */
    virtual void enable_auto_newline(bool enabled) = 0;
    /** 设置日志级别，跟踪日志级别不能通过它来设置 */
    virtual void set_log_level(log_level_t log_level) = 0;
    /** 设置单个文件的最大建议大小 */
    virtual void set_single_filesize(uint32_t filesize) = 0;
    /** 设置日志文件备份个数，不包正在写的日志文件 */
    virtual void set_backup_number(uint16_t backup_number) = 0;

    /** 是否允许二进制日志 */
    virtual bool enabled_bin() = 0;
    /** 是否允许 Debug 级别日志 */
    virtual bool enabled_debug() = 0;
    /** 是否允许 Info 级别日志 */
    virtual bool enabled_info() = 0;
    /** 是否允许 Warn 级别日志 */
    virtual bool enabled_warn() = 0;
    /** 是否允许 Error 级别日志 */
    virtual bool enabled_error() = 0;
    /** 是否允许 Fatal 级别日志 */
    virtual bool enabled_fatal() = 0;
    /** 是否允许 Trace 级别日志 */
    virtual bool enabled_trace() = 0;

    virtual void log_debug(const char* format, ...) = 0;
    virtual void log_info(const char* format, ...)  = 0;
    virtual void log_warn(const char* format, ...)  = 0;
    virtual void log_error(const char* format, ...) = 0;
    virtual void log_fatal(const char* format, ...) = 0;
```

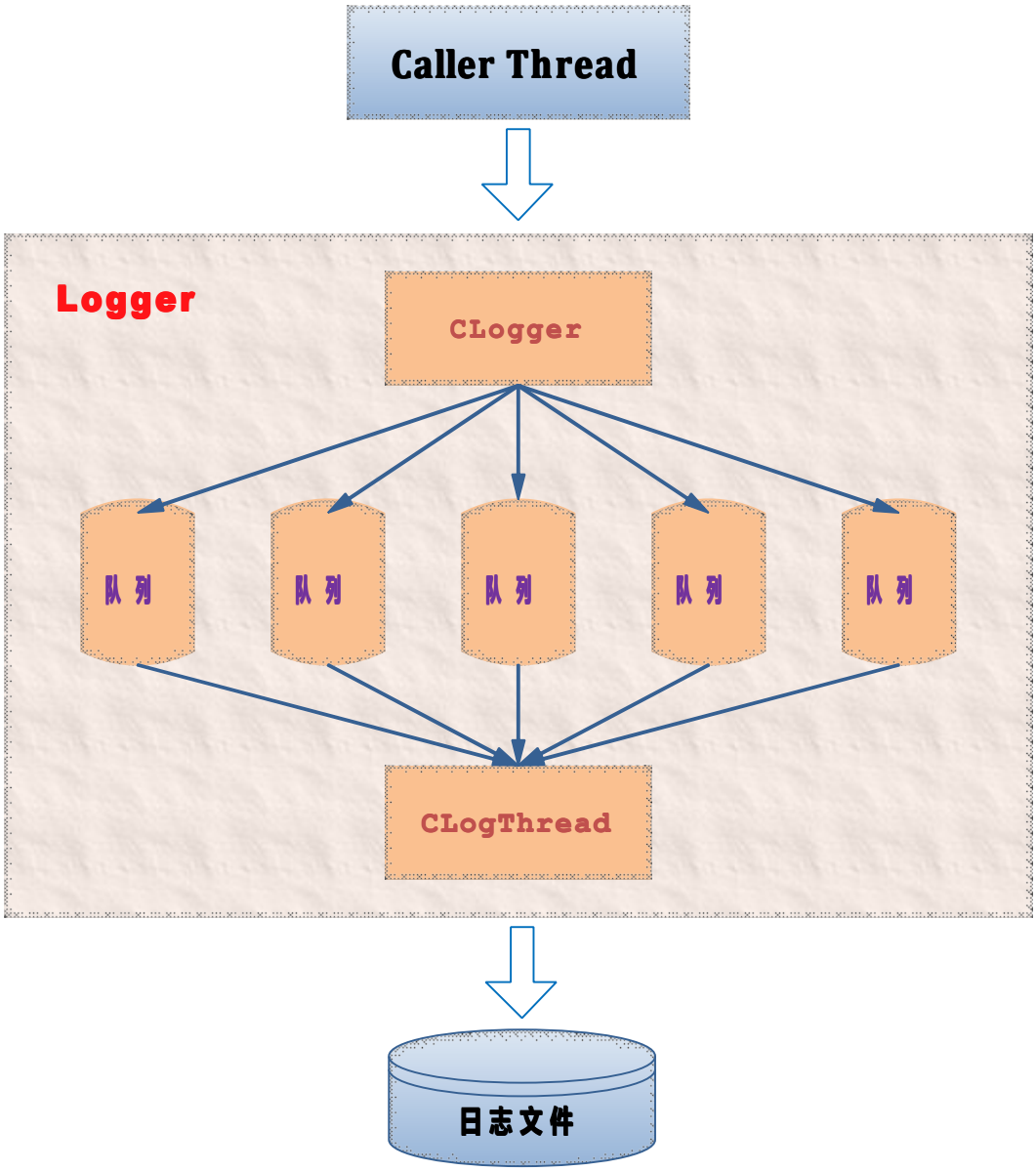
```
    virtual void log_trace(const char* format, ...) = 0;
    virtual void log_bin(const char* log, uint16_t size) = 0;

    /** 写二进制日志 */
    virtual void bin_log(const char* log, uint16_t size) = 0;
};
```

4. 日志宏

MYLOG_DEBUG(format, ...)
MYLOG_INFO(format, ...)
MYLOG_WARN(format, ...)
MYLOG_ERROR(format, ...)
MYLOG_FATAL(format, ...)
MYLOG_TRACE(format, ...)
MYLOG_BIN(log, size)

5. 工作原理



Logger 创建一个独立的线程，专门用来将队列中的日志写入磁盘。而日志线程可以配置多个队列，每个调用者线程的日志可以配置总是只存储到同一个队列，也可以不分队列存储，不分队列存放效率是最高的，但同一个线程输出的日志将是无时间顺序（秒级顺序错乱）的。

引入多队列，可以降低调用者线程和日志线程之间的碰撞，从而保证高效。

日志并不一定是按一条条写入磁盘的，日志线程总是一次性的借助 **writev** 系统调用将一个队列中当前所有的日志一次性写入磁盘，这样就减少了 I/O 次数。

但同时，Logger 仍旧保持了日志写入磁盘的及时性，在调用者线程和日志线程间设置了通知事件，如果所有队列均为空，则日志线程进入睡觉等待状态，任意调用线程将日志存入日志队列后，调用线程会检查日志线程是否处于睡觉等待状态，如果是则唤醒它，否则只是将日志放入队列。

6. 日志格式

日志格式为：[YYYY-MM-DD HH:MM:SS][8 位十六进制线程号][日志级别名称]日志内容，如：

[2010-09-01 21:10:20][0x8E92B4A1][INFO]测试日志.
[2010-09-01 21:10:23][0x6342B4A1][INFO]测试日志

7. 使用示例

```
#include <sys/logger.h>

int main()
{
    sys::CLogger* logger = new sys::CLogger();

    try
    {
        if (logger->create(".", "test.log"))
        {
            fprintf(stderr, "Can not create log file.\n");
            return 1;
        }
    }
    catch (CSyscallException& ex)
    {
        fprintf(stderr, "Failed to start log thread for %d at %s:%d"
            , ex.get_errcode(), ex.get_filename(), ex.get_linenumber());
        return 1;
    }

    // 下面这一句初始化全局的日志器，以方便使用日志宏，
    // 否则使用日志宏时，日志将打印到标准输出上。
    sys::ILogger* sys::g_logger = logger;

    // 允许同时在屏幕上输出日志
    logger->enable_screen(true);

    // 写日志，要使用日志宏，必须保证已经初始化 sys::g_logger
    MYLOG_DEBUG("%s", "测试日志");

    // 不再需要日志器了，比如进程退出之前
```

```
    logger->destroy(); // 日志器销毁之前会保证队列中的所有日志都写入文件
    delete logger;
    return 0;
}
```

8. 项目门户

8.1. 开发论坛

http://bbs.hadoopor.com/index.php?gid=67

8.2. 代码位置

http://code.google.com/p/mooon/

8.3. 项目邮箱

eyjian@qq.com 和 eyjian@gmail.com