最新版Web服务器项目详解 - 10 日志系统（下）

原创 互联网猿 两猿社 2020-05-06 08:30

本文内容

日志系统分为两部分，其一是单例模式与阻塞队列的定义，其二是日志类的定义与使用。

本篇将介绍日志类的定义与使用，具体的涉及到基础API，流程图与日志类定义，功能实现。

基础API，描述fputs，可变参数宏__VA_ARGS__，fflush

流程图与日志类定义，描述日志系统整体运行流程，介绍日志类的具体定义

功能实现，结合代码分析同步、异步写文件逻辑，分析超行、按天分文件和日志分级的具体实现

基础API

为了更好的源码阅读体验，这里对一些API用法进行介绍。

fputs

```
1  #include <stdio.h>
2  int fputs(const char *str, FILE *stream);
```

- str，一个数组，包含了要写入的以空字符终止的字符序列。
- stream，指向FILE对象的指针，该FILE对象标识了要被写入字符串的流。

可变参数宏__VA_ARGS__

__VA_ARGS__是一个可变参数的宏，定义时宏定义中参数列表的最后一个参数为省略号，在实际使用时会发现有时会加##，有时又不加。

```
//最简单的定义
#define my_print1(...)  printf(__VA_ARGS__)

//搭配va_list的format使用
#define my_print2(format, ...) printf(format, __VA_ARGS__)
#define my_print3(format, ...) printf(format, ##__VA_ARGS__)
```

__VA_ARGS__宏前面加上##的作用在于，当可变参数的个数为0时，这里printf参数列表中的的##会把前面多余的","去掉，否则会编译出错，建议使用后面这种，使得程序更加健壮。

fflush

```
#include <stdio.h>
int fflush(FILE *stream);
```

fflush()会强迫将缓冲区内的数据写回参数stream 指定的文件中，如果参数stream 为NULL，fflush()会将所有打开的文件数据更新。

在使用多个输出函数连续进行多次输出到控制台时，有可能下一个数据再上一个数据还没输出完毕，还在输出缓冲区中时，下一个printf就把另一个数据加入输出缓冲区，结果冲掉了原来的数据，出现输出错误。

在prinf()后加上fflush(stdout); 强制马上输出到控制台，可以避免出现上述错误。

流程图与日志类定义

流程图

- 日志文件
  - 局部变量的懒汉模式获取实例
  - 生成日志文件，并判断同步和异步写入方式
- 同步
  - 判断是否分文件
  - 直接格式化输出内容，将信息写入日志文件
- 异步
  - 判断是否分文件
  - 格式化输出内容，将内容写入阻塞队列，创建一个写线程，从阻塞队列取出内容写入日志文件

日志类定义

通过局部变量的懒汉单例模式创建日志实例，对其进行初始化生成日志文件后，格式化输出内容，并根据不同的写入方式，完成对应逻辑，写入日志文件。

日志类包括但不限于如下方法，

- 公有的实例获取方法
- 初始化日志文件方法
- 异步日志写入方法，内部调用私有异步方法
- 内容格式化方法
- 刷新缓冲区
- …

```
class Log
{
public:
    //C++11以后,使用局部变量懒汉不用加锁
    static Log *get_instance()
    {
        static Log instance;
        return &instance;
    }

    //可选择的参数有日志文件、日志缓冲区大小、最大行数以及最长日志条队列
    bool init(const char *file_name, int log_buf_size = 8192, int split_lines = 50

    //异步写日志公有方法，调用私有方法async_write_log
    static void *flush_log_thread(void *args)
    {
        Log::get_instance()->async_write_log();
    }

    //将输出内容按照标准格式整理
    void write_log(int level, const char *format, ...);

    //强制刷新缓冲区
    void flush(void);

private:
    Log();
    virtual ~Log();

    //异步写日志方法
    void *async_write_log()
    {
        string single_log;

        //从阻塞队列中取出一条日志内容，写入文件
        while (m_log_queue->pop(single_log))
        {
            m_mutex.lock();
            fputs(single_log.c_str(), m_fp);
            m_mutex.unlock();
        }
    }

private:
    char dir_name[128];     //路径名
    char log_name[128];     //log文件名
    int m_split_lines;      //日志最大行数
    int m_log_buf_size;     //日志缓冲区大小
    long long m_count;      //日志行数记录
    int m_today;            //按天分文件,记录当前时间是那一天
    FILE *m_fp;             //打开log的文件指针
    char *m_buf;            //要输出的内容
    block_queue<string> *m_log_queue; //阻塞队列
    bool m_is_async;                  //是否同步标志位
    locker m_mutex;            //同步类
};


//这四个宏定义在其他文件中使用，主要用于不同类型的日志输出
#define LOG_DEBUG(format, ...) Log::get_instance()->write_log(0, format, __VA_ARGS
#define LOG_INFO(format, ...) Log::get_instance()->write_log(1, format, __VA_ARGS_
#define LOG_WARN(format, ...) Log::get_instance()->write_log(2, format, __VA_ARGS_
#define LOG_ERROR(format, ...) Log::get_instance()->write_log(3, format, __VA_ARGS

#endif
```

日志类中的方法都不会被其他程序直接调用，末尾的四个可变参数宏提供了其他程序的调用方法。

前述方法对日志等级进行分类，包括DEBUG，INFO，WARN和ERROR四种级别的日志。

功能实现

init函数实现日志创建、写入方式的判断。

write_log函数完成写入日志文件中的具体内容，主要实现日志分级、分文件、格式化输出内容。

生成日志文件 && 判断写入方式

通过单例模式获取唯一的日志类，调用init方法，初始化生成日志文件，服务器启动按当前时刻创建日志，前缀为时间，后缀为自定义log文件名，并记录创建日志的时间day和行数count。

写入方式通过初始化时**是否设置队列大小**（表示在队列中可以放几条数据）来判断，若队列大小为0，则为同步，否则为异步。

```cpp
//异步需要设置阻塞队列的长度，同步不需要设置
bool Log::init(const char *file_name, int log_buf_size, int split_lines, int max_q
{
    //如果设置了max_queue_size,则设置为异步
    if (max_queue_size >= 1)
    {
        //设置写入方式flag
        m_is_async = true;

        //创建并设置阻塞队列长度
        m_log_queue = new block_queue<string>(max_queue_size);
        pthread_t tid;

        //flush_log_thread为回调函数,这里表示创建线程异步写日志
        pthread_create(&tid, NULL, flush_log_thread, NULL);
    }

    //输出内容的长度
    m_log_buf_size = log_buf_size;
    m_buf = new char[m_log_buf_size];
    memset(m_buf, '\0', sizeof(m_buf));

    //日志的最大行数
    m_split_lines = split_lines;

    time_t t = time(NULL);
    struct tm *sys_tm = localtime(&t);
    struct tm my_tm = *sys_tm;

    //从后往前找到第一个/的位置
    const char *p = strrchr(file_name, '/');
    char log_full_name[256] = {0};

    //相当于自定义日志名
    //若输入的文件名没有/，则直接将时间+文件名作为日志名
    if (p == NULL)
    {
        snprintf(log_full_name, 255, "%d_%02d_%02d_%s", my_tm.tm_year + 1900, my_t
    }
    else
    {
        //将/的位置向后移动一个位置，然后复制到logname中
        //p - file_name + 1是文件所在路径文件夹的长度
        //dirname相当于./
        strcpy(log_name, p + 1);
        strncpy(dir_name, file_name, p - file_name + 1);

        //后面的参数跟format有关
        snprintf(log_full_name, 255, "%s%d_%02d_%02d_%s", dir_name, my_tm.tm_year
    }

    m_today = my_tm.tm_mday;

    m_fp = fopen(log_full_name, "a");
    if (m_fp == NULL)
    {
        return false;
    }

    return true;
}
```

日志分级与分文件

日志分级的实现大同小异，一般的会提供五种级别，具体的，

- Debug，调试代码时的输出，在系统实际运行时，一般不使用。
- Warn，这种警告与调试时终端的warning类似，同样是调试代码时使用。
- Info，报告系统当前的状态，当前执行的流程或接收的信息等。
- Error和Fatal，输出系统的错误信息。

上述的使用方法仅仅是个人理解，在开发中具体如何选择等级因人而异。项目中给出了除Fatal外的四种分级，实际使用了Debug，Info和Error三种。

超行、按天分文件逻辑，具体的，

- 日志写入前会判断当前day是否为创建日志的时间，行数是否超过最大行限制
  - 若为创建日志时间，写入日志，否则按当前时间创建新log，更新创建时间和行数
  - 若行数超过最大行限制，在当前日志的末尾加count/max_lines为后缀创建新log

将系统信息格式化后输出，具体为：格式化时间 + 格式化内容

```cpp
void Log::write_log(int level, const char *format, ...)
{
    struct timeval now = {0, 0};
    gettimeofday(&now, NULL);
    time_t t = now.tv_sec;
    struct tm *sys_tm = localtime(&t);
    struct tm my_tm = *sys_tm;
    char s[16] = {0};

    //日志分级
    switch (level)
    {
    case 0:
        strcpy(s, "[debug]:");
        break;
    case 1:
        strcpy(s, "[info]:");
        break;
    case 2:
        strcpy(s, "[warn]:");
        break;
    case 3:
        strcpy(s, "[erro]:");
        break;
    default:
        strcpy(s, "[info]:");
        break;
    }


    m_mutex.lock();

    //更新现有行数
    m_count++;

    //日志不是今天或写入的日志行数是最大行的倍数
    //m_split_lines为最大行数
    if (m_today != my_tm.tm_mday || m_count % m_split_lines == 0)
    {
        char new_log[256] = {0};
        fflush(m_fp);
        fclose(m_fp);
        char tail[16] = {0};

        //格式化日志名中的时间部分
        snprintf(tail, 16, "%d_%02d_%02d_", my_tm.tm_year + 1900, my_tm.tm_mon +

        //如果是时间不是今天,则创建今天的日志，更新m_today和m_count
        if (m_today != my_tm.tm_mday)
        {
            snprintf(new_log, 255, "%s%s%s", dir_name, tail, log_name);
            m_today = my_tm.tm_mday;
            m_count = 0;
        }
        else
        {
            //超过了最大行，在之前的日志名基础上加后缀, m_count/m_split_lines
            snprintf(new_log, 255, "%s%s%s.%lld", dir_name, tail, log_name, m_cou
        }
        m_fp = fopen(new_log, "a");
    }

    m_mutex.unlock();

    va_list valst;
    //将传入的format参数赋值给valst，便于格式化输出
```

```
67      va_start(valst, format);
68
69      string log_str;
70      m_mutex.lock();
71
72      //写入内容格式：时间 + 内容
73      //时间格式化，snprintf成功返回写字符的总数，其中不包括结尾的null字符
74      int n = snprintf(m_buf, 48, "%d-%02d-%02d %02d:%02d:%02d.%06ld %s ",
75                       my_tm.tm_year + 1900, my_tm.tm_mon + 1, my_tm.tm_mday,
76                       my_tm.tm_hour, my_tm.tm_min, my_tm.tm_sec, now.tv_usec, s);
77
78      //内容格式化，用于向字符串中打印数据、数据格式用户自定义，返回写入到字符数组str中的字
79      int m = vsnprintf(m_buf + n, m_log_buf_size - 1, format, valst);
80      m_buf[n + m] = '\n';
81      m_buf[n + m + 1] = '\0';
82
83      log_str = m_buf;
84
85      m_mutex.unlock();
86
87      //若m_is_async为true表示异步，默认为同步
88      //若异步,则将日志信息加入阻塞队列,同步则加锁向文件中写
89      if (m_is_async && !m_log_queue->full())
90      {
91          m_log_queue->push(log_str);
92      }
93      else
94      {
95          m_mutex.lock();
96          fputs(log_str.c_str(), m_fp);
97          m_mutex.unlock();
98      }
99
100     va_end(valst);
101 }
```

如果本文对你有帮助， 阅读原文 star一下服务器项目，我们需要你的星星^_^.

完。