C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

讲师介绍--专业来自专注和实力

King老师

系统架构师，曾供职著名创业公司系统架构师，微软亚洲研究院、创维集团全球研发中心。国内第一代商业Paas平台开发者。著有多个软件专利，参与多个开源软件维护。在全球化，高可用的物联网云平台架构与智能硬件设计方面有丰富的研发与实战经验。

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

开源框架log4cpp和日志模块实现

1.日志写入逻辑

2.Log4cpp日志框架

3.Log4cpp范例讲解

4.muduo日志库分析

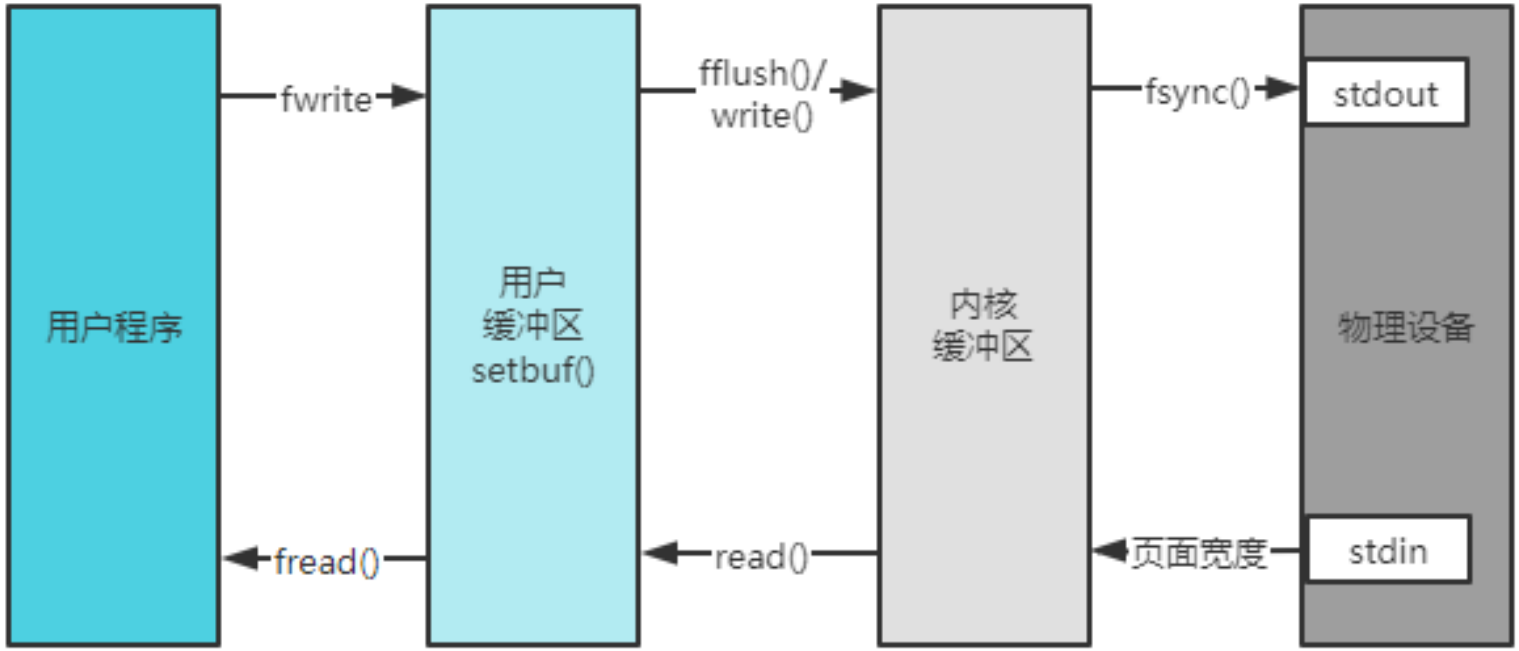
1 日志写入逻辑1

```cpp
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    FILE *file = fopen("0-fwrite_test.log", "wt");

    for(int i = 0; i < 10; i++) {

        ostringstream oss;
        oss<<"NO." << i<<" Root Error Message!\n";
        fwrite(oss.str().c_str(), 1, oss.str().size(),  file);
    }

    fclose(file);
}
```

```
Breakpoint 2, 0x00007ffff71dd3b0 in write () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) bt
#0  0x00007ffff71dd3b0 in write () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x00007ffff715ec0f in _IO_file_write () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x00007ffff7160419 in _IO_do_write () from /lib/x86_64-linux-gnu/libc.so.6
#3  0x00007ffff715f9c0 in _IO_file_close_it () from /lib/x86_64-linux-gnu/libc.so.6
#4  0x00007ffff71533ff in fclose () from /lib/x86_64-linux-gnu/libc.so.6
#5  0x0000000000400f3c in main () at /mnt/hgfs/log/src-log4cpp/0-fwrite_test.cpp:16
```

write为什么会出现，应用程序调用的是fwrite

1 日志写入逻辑2



fwrite 1000000 line, fwrite:10000, fflush:1, buf_size:2900, need time:3128ms, ops:319693

write 1000000 line, write:10000, fsync:1, buf_size:2900, need time:1784ms, ops:560538

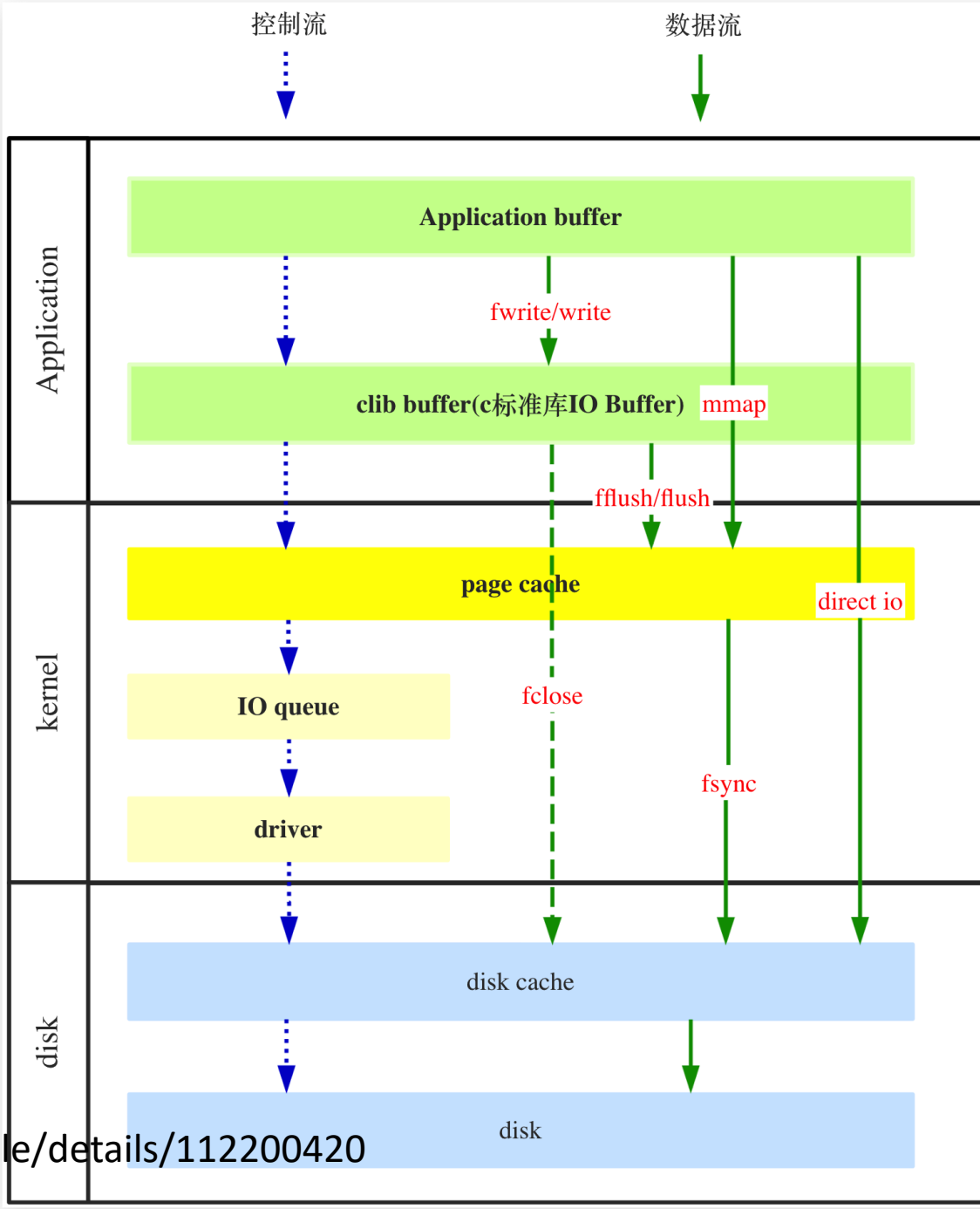
fwrite 1000000 line, fwrite:50000, fflush:1, buf_size:580, need time:4240ms, ops:**235849**

write 1000000 line, write:50000, fsync:1, buf_size:580, need time:7005ms, ops:142755

```
fwrite 100000 line, fwrite:100, fflush:100, buf_size:28000, need time:76ms, ops:1315789
write  100000 line, write:100,  fsync:100, buf_size:28000, need time:63ms, ops:1587301
```

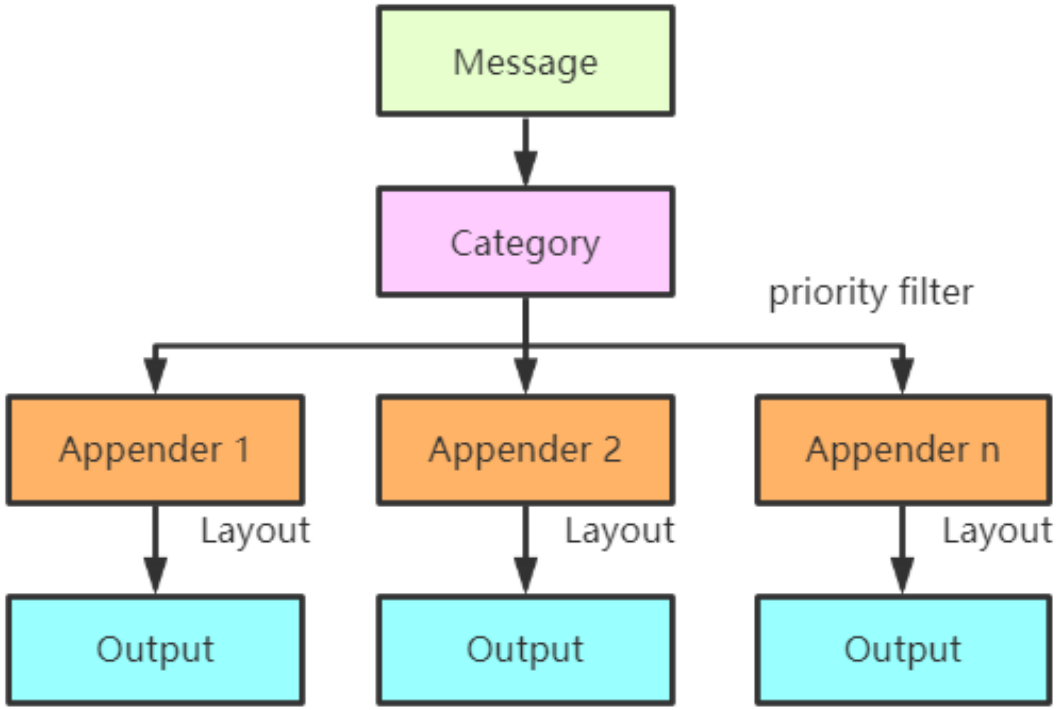
1 日志写入逻辑3-总结

- fwrite有缓存，write没有缓存。
- write是系统调用，每次需要将数据写到磁盘，写的大小是要求的大小，依然涉及频繁的用户态和内核态切换。
- fwrite是库函数，每次将数据写入到缓冲区，等缓冲区满了，一次写入磁盘。或者使用fflush冲洗缓冲区。
- 功能
  - fflush:是把C库中的缓冲调用write函数写到磁盘[其实是写到内核的缓冲区]。
  - fsync：是把内核缓冲刷到磁盘上。



参考：https://blog.csdn.net/hilaryfrank/article/details/112200420

2 Log4cpp日志框架



Log4cpp中最重要概念有Category（种类）、Appender（附加器）、Layout（布局）、Priorty（优先级）

2.1 下载和编译

见pdf文档：开源框架log4cpp和日志模块实现-资料.pdf

2. 2 日志级别

log4cpp为例，具体的级别看具体的日志库

- EMERG
- FATAL
- ALERT
- CRIT
- ERROR
- WARN
- NOTICE
- INFO
- DEBUG

有些日志库 DEBUG和INFO的级别是反过来的。

2. 3 日志格式化1

比如：20210410 14:18:15.299684Z 30836 INFO NO.1506710 Root Error Message! - main_log_test.cc:47
格式：年月日 时分秒 微妙 时区 日志级别 日志内容 文件名 行号

比如Log4cpp的log4j.appender.A2.layout.ConversionPattern=%d %m %n

- %% - 转义字符'%'
- %c - Category
- %d - 日期；日期可以进一步设置格式，用花括号包围，例如%d{%H:%M:%S,%l}。
  日期的格式符号与ANSI C函数strftime中的一致。但增加了一个格式符号%l，表示毫秒，占三个十进制位。
- %m - 消息
- %n - 换行符；会根据平台的不同而不同，但对用户透明。
- %p - 优先级
- %r - 自从layout被创建后的毫秒数
- %R - 从1970年1月1日开始到目前为止的秒数
- %u - 进程开始到目前为止的时钟周期数
- %x - NDC
- %t - **线程id**

2.3 日志格式化2-layout

- BasicLayout::format
- PassThroughLayout::format 支持自定义的布局，我们可以继承他实现自定义的日志格式
- PatternLayout::format log4cpp支持用户配置日志格式
- SimpleLayout::format 比BasicLayout还简单的日志格式输出。

PassThroughLayout

直通布局。顾名思义，这个就是没有布局的“布局”，你让它写什么它就写什么，它不会为你添加任何东西，连换行符都懒得为你加。

SimpleLayout

简单布局。它只会为你添加“优先级”的输出。相当于PatternLayout格式化为：“%p: %m%n”。

BasicLayout

基本布局。它会为你添加“时间”、“优先级”、“种类”、“NDC”。相当于PatternLayout格式化为：“%R %p %c %x: %m%n”

2.3 日志格式化3-layout

PatternLayout

格式化布局。它的使用方式类似C语言中的printf，使用格式化它符串来描述输出格式。目前支持的转义定义如下：

%% - 转义字符'%'

%c - Category

%d - 日期；日期可以进一步设置格式，用花括号包围，例如%d{%H:%M:%S,%l}。

日期的格式符号与ANSI C函数strftime中的一致。但增加了一个格式符号%l，表示毫秒，占三个十进制位。

%m - 消息

%n - 换行符；会根据平台的不同而不同，但对用户透明。

%p - 优先级

%r - 自从layout被创建后的毫秒数

%R - 从1970年1月1日开始到目前为止的秒数

%u - 进程开始到目前为止的时钟周期数

%x - NDC

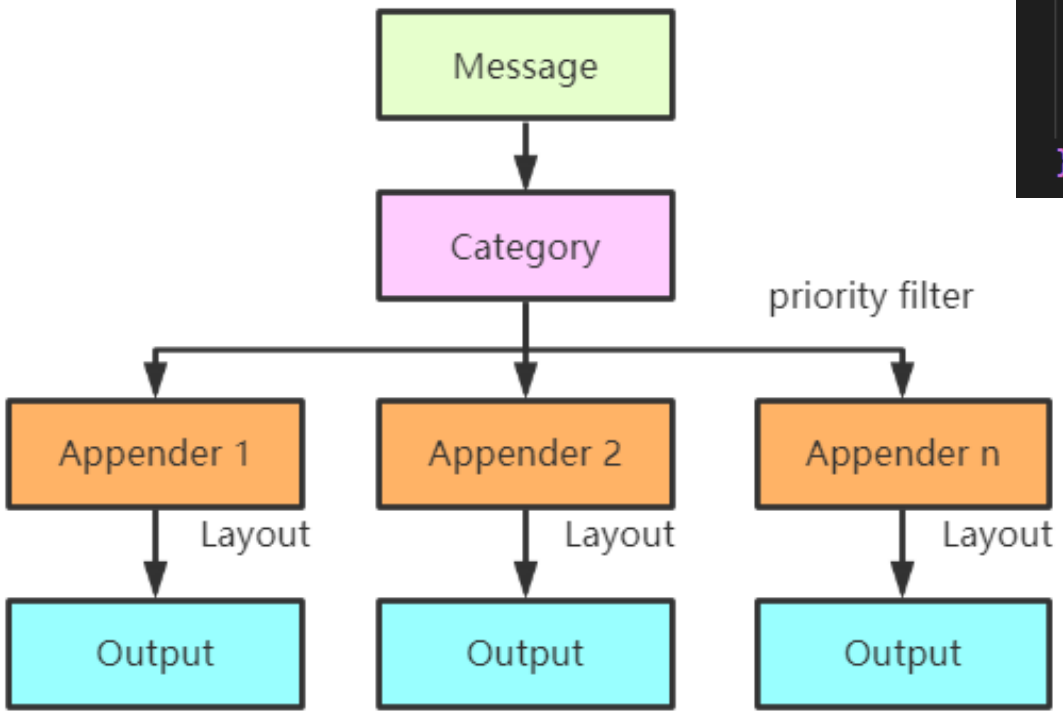
%t - 线程id

2.4 日志输出

日志不同的输出方式，以log4cpp为例：

- 日志输出到控制台，比如ConsoleAppender
- 日志输出到本地文件，比如FileAppender
- 日志通过网络输出到远程服务器，比如**RemoteSyslogAppender**

```
void FileAppender::_append(const LoggingEvent& event) {
    std::string message(_getLayout().format(event));
    if (!::write(_fd, message.data(), message.length())) {
        // XXX help! help!
    }
}
```



2.5 日志回滚

比如以下功能：

- 本地日志支持最大文件限制
- 当本地日志到达最大文件限制的时候新建一个文件
- 每天至少一个文件

比如：

```
RollingFileAppender(const std::string& name,
                    const std::string& fileName,
                    size_t maxFileSize = 10*1024*1024,
                    unsigned int maxBackupIndex = 1,
                    bool append = true,
                    mode_t mode = 00644);
```

2.6 配置文件

比如lo4cpp的配置文件

```
# 文件名: test_log4cpp2.conf
# a simple test config
#定义了3个category sub1, sub2, sub1.sub2
log4j.rootCategory=DEBUG, rootAppender
log4j.category.sub1=,A1
log4j.category.sub2=INFO
#log4j.category.sub1.sub2=ERROR, A2
log4j.category.sub1.sub2=, A2
# 设置sub1.sub2 的additivity属性
log4j.additivity.sub1.sub2=true
#定义rootAppender类型和layout属性
log4j.appender.rootAppender=org.apache.log4j.ConsoleAppender
log4j.appender.rootAppender.layout=org.apache.log4j.BasicLayout
#定义A1的属性
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.fileName=A1.log
log4j.appender.A1.layout=org.apache.log4j.SimpleLayout
#定义A2的属性
log4j.appender.A2=org.apache.log4j.ConsoleAppender
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
#log4j.appender.A2.layout.ConversionPattern=The message '%m' at time %d%n
log4j.appender.A2.layout.ConversionPattern=%d %m %n
```

3 Log4cpp范例讲解

- 配置文件剖析
- Log4cpp调用栈分析
- 性能测试
- 日志回滚
- 日志库瓶颈分析

3. 1配置文件剖析

重点分析3-test_log4cpp.conf

Category有2个参数，日志级别,Appender。
留空说明是继承父的设置，不为空则是自己的级别

```
# a simple test config
#定义了3个category sub1, sub2, sub1.sub2
log4j.rootCategory=DEBUG, rootAppender
log4j.category.sub1=,A1
log4j.category.sub2=INFO
#log4j.category.sub1.sub2=ERROR, A2
log4j.category.sub1.sub2=, A2
# 设置sub1.sub2 的additivity属性
log4j.additivity.sub1.sub2=true
#定义rootAppender类型和layout属性
log4j.appender.rootAppender=org.apache.log4j.ConsoleAppender
log4j.appender.rootAppender.layout=org.apache.log4j.BasicLayout
#定义A1的属性
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.fileName=A1.log
log4j.appender.A1.layout=org.apache.log4j.SimpleLayout
#定义A2的属性
log4j.appender.A2=org.apache.log4j.ConsoleAppender
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
#log4j.appender.A2.layout.ConversionPattern=The message '%m' at time %d%n
log4j.appender.A2.layout.ConversionPattern=%d %m %n
```

日志级别留空说明sub1日志级别和root一致；appender为A1，则后续要初始化A1

日志级别自定义为INFO；appender和root一致

日志级别留空说明sub1.sub2日志级别和sub1一致；appender为A2，则后续要初始化A2

如果值为true，则该Category的Appender包含了父Category的Appender, 即是日志也从root的appender输出
如果值为false，则该Category的Appender取代了父Category的Appender

初始化root的appender属性

初始化A1的appender属性

初始化A2的appender属性

=true

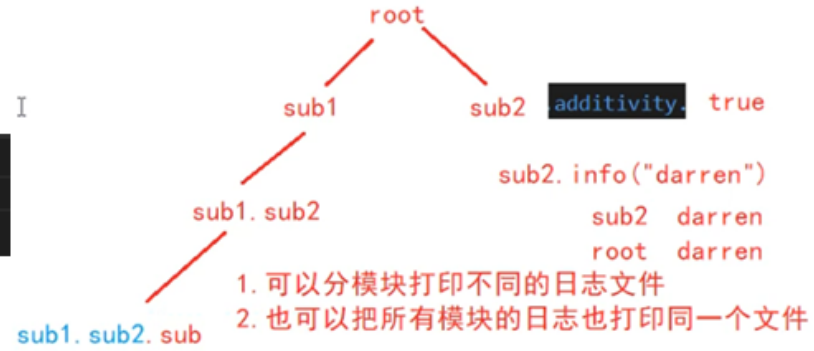
A1.log

```
DEBUG    : sub1 This is some debug
DEBUG    : sub1_sub2 This is some debug
ERROR    : sub1 This is some error
ERROR    : sub1_sub2 This is some error
```

=false

A1.log

```
DEBUG    : sub1 This is some debug
ERROR    : sub1 This is some error
```



3. 2 Log4cpp调用栈分

Category::callAppenders

```
void Category::callAppenders(const LoggingEvent& event) throw() {
    threading::ScopedLock lock(_appenderSetMutex);
    {
        if (!_appender.empty()) {
            for(AppenderSet::const_iterator i = _appender.begin();
                i != _appender.end(); i++) {
                (*i)->doAppend(event);
            }
        }
    }
    if (getAdditivity() && (getParent() != NULL)) {
        getParent()->callAppenders(event);      // 父类的appender
    }
}
```

FileAppender::_append

```
void FileAppender::_append(const LoggingEvent& event) {
    std::string message(_getLayout().format(event)); // 根据自己的layout

    // 通过一个变量去累加
    // write_bytes += message.length();
    if (!::write(_fd, message.data(), message.length())) {
        // XXX help! help!
    }
}
```

RollingFileAppender::_append

```
void RollingFileAppender::_append(const LoggingEvent& event) {
    FileAppender::_append(event);
    off_t offset = ::lseek(_fd, 0, SEEK_END);       // 获取文件大小，也是影响效率的关键
    if (offset < 0) {
        // XXX we got an error, ignore for now
    } else {
        if(static_cast<size_t>(offset) >= _maxFileSize) {
            rollOver();
        }
    }
}
```

3.3 性能测试

同步写入

```
FileAppender----------------->
begin_time: 1640855533530
end_time: 1640855534742
need the time: 1640855534742 1640855533530, 1212毫秒, ops = 8250ops/s

RollingFileAppender----------------->
begin_time: 1640855534742
end_time: 1640855538330
need the time: 1640855538330 1640855534742, 3588毫秒, ops = 2787ops/s
```

异步写入

```
Get message from Memory Queue!
----------------------------------------
begin_time: 1640855586054
end_time: 1640855586110
need the time: 1640855586110 1640855586054, 56毫秒

RollingFileAppender----------------->
begin_time: 1640855586110
need the time: 1640855586138 1640855586110, 28毫秒, ops = 1785714ops/s
```

3. 4 日志回滚

以4-RollingFileAppender.cpp为范例

1.删除log.5
2. log.4 ->log.5
   log.3 ->log.4
   log.2 ->log.3
   log.1 ->log.2
   log ->log.1
新建一个.log

3.5 日志性能分析

- 实时写入磁盘 单笔 write
  - 多行（100行）日记累积再写入 -> 累积了99行，下一行一直不来。
    - （1）单独起一个定时器（比如1秒）去刷新
    - （2）同一个日志管理线程去刷新数据

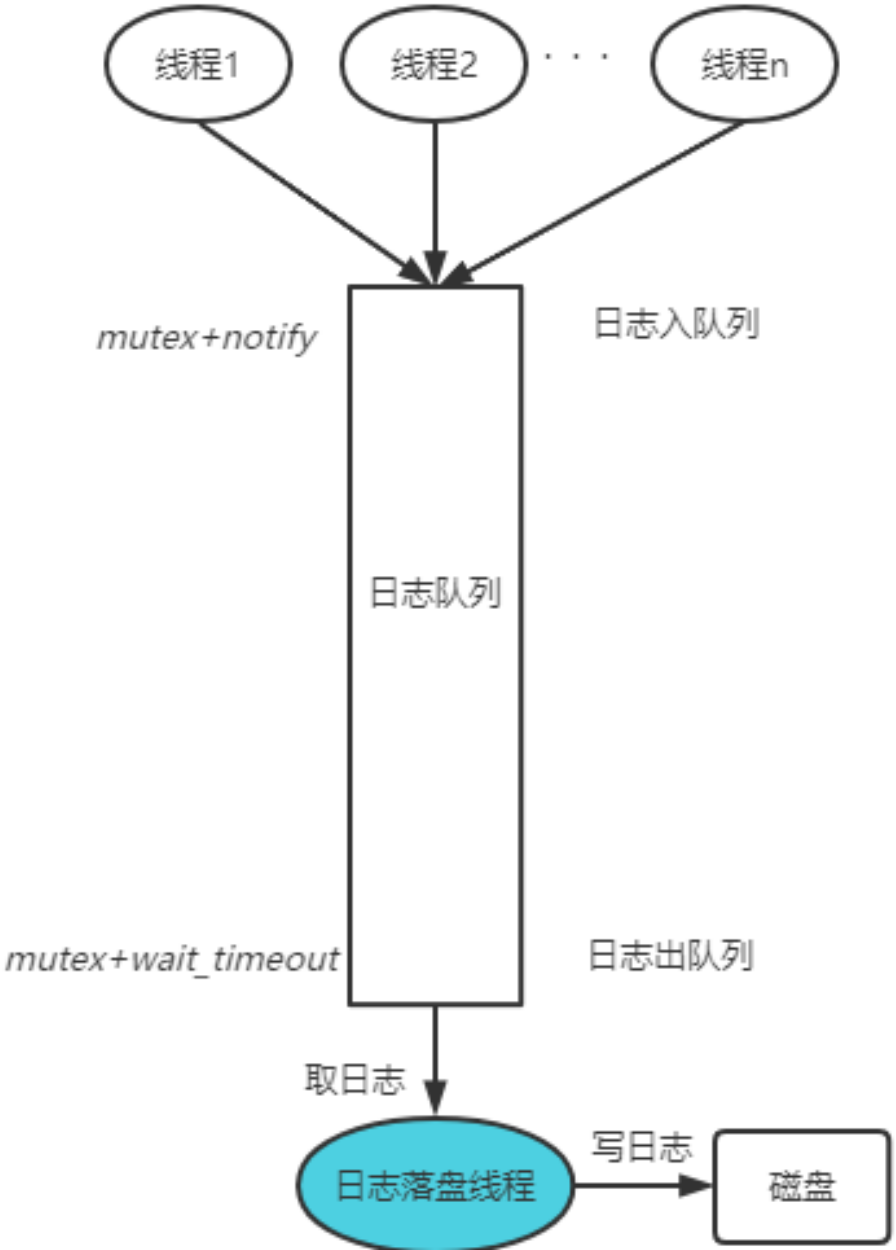
    注：实际glog他是累积一定的行数或者过了一定的时间间隔就让刷新。
- 回滚日志每次都取读取日志文件大小 肯定不能每次读取文件大小

4 muduo日志库分析

重点

- 性能分析
- 日志批量写入
- 批量唤醒写线程
- 写日志用notify+wait_timeout方式触发日志的写入
- 锁的粒度，双缓冲，双队列
- buffer默认4M缓冲区，buffers是buffer队列，push、pop时使用move语义减少内存拷贝
- coredump日志丢失找回分析

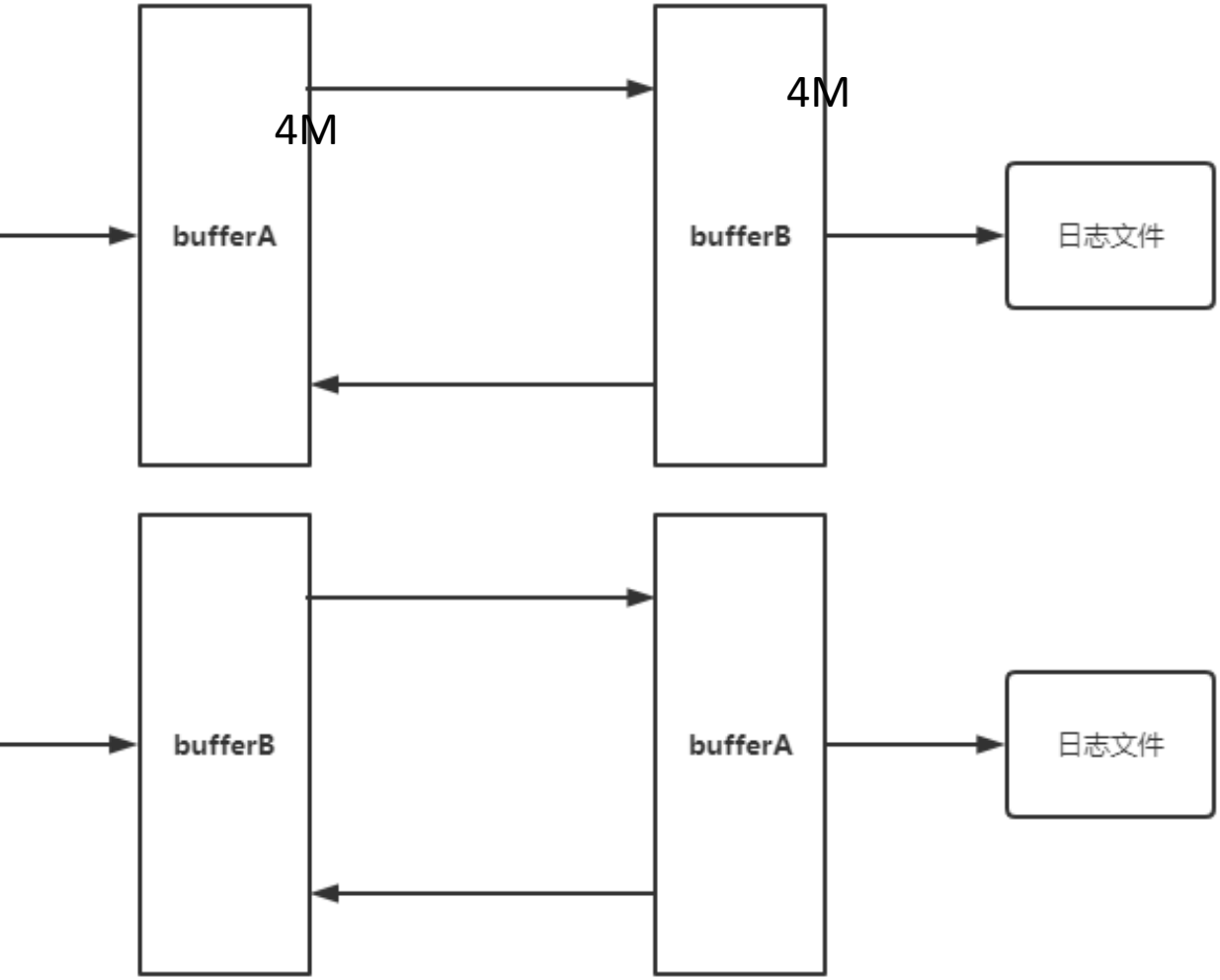
4.1 异步日志机制



1. 该怎么唤醒日志落盘线程读取日志写入磁盘
   1. ~~日志落盘线程 sleep(1s)~~
2. 每次发notify对性能有没有影响
   1. Mutex 多线程互斥
   2. ~~要不要发notify，累积多行日志再发送？~~
3. 缓存多条日志才唤醒日志落盘线程？

4. 日志写入磁盘时是批量写入还是单条写入
   1. **批量写入**

4.1 双缓冲机制

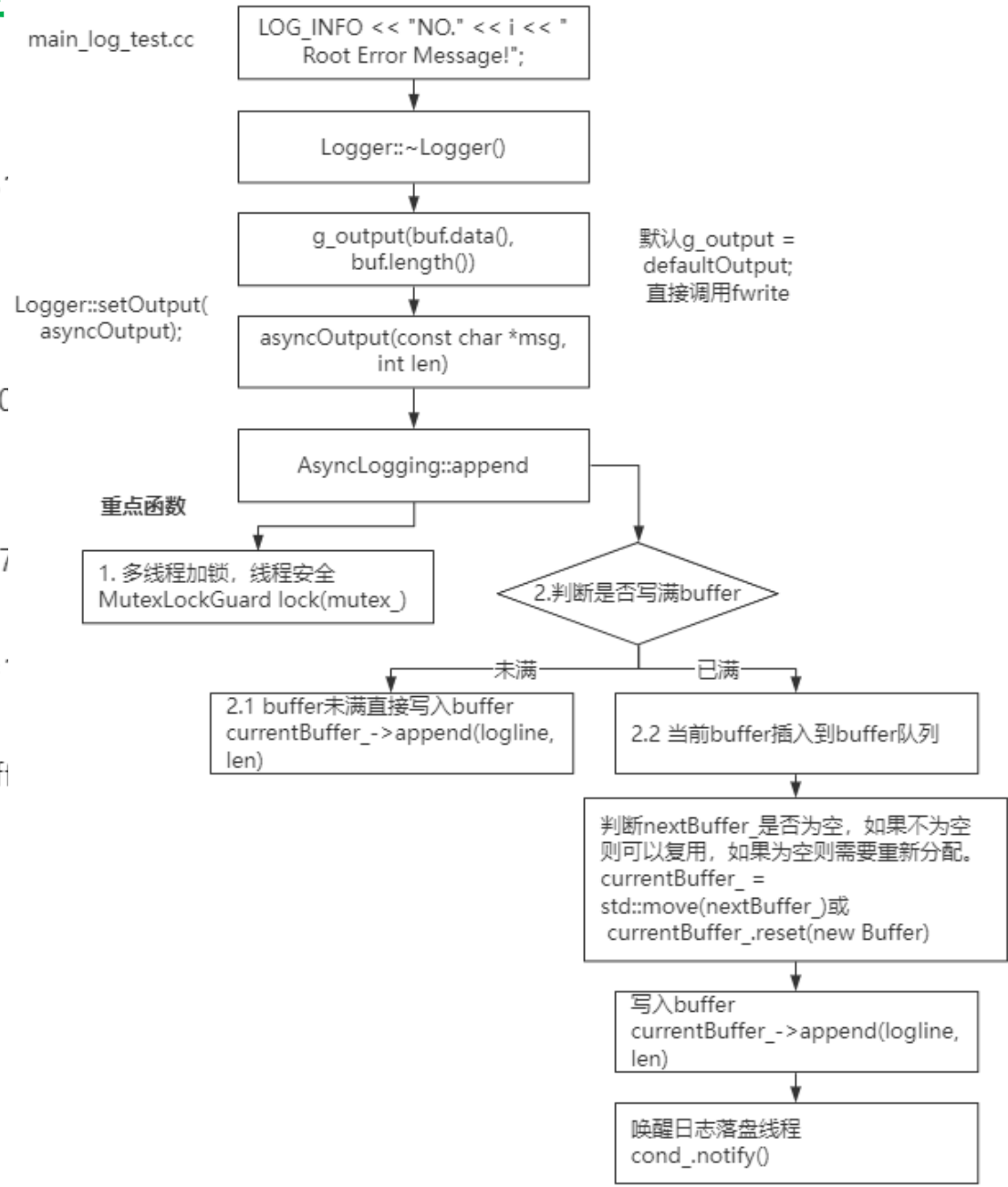


1. 日志notify的问题
   1. 写满1个buffer才发一次notify唤醒日志落盘线程
   2. 超时通过wait_timeout唤醒日志落盘线程，buffer只要有数据就写入到磁盘
2. 避免buffer不断分配
3. Buffer默认大小4M字节

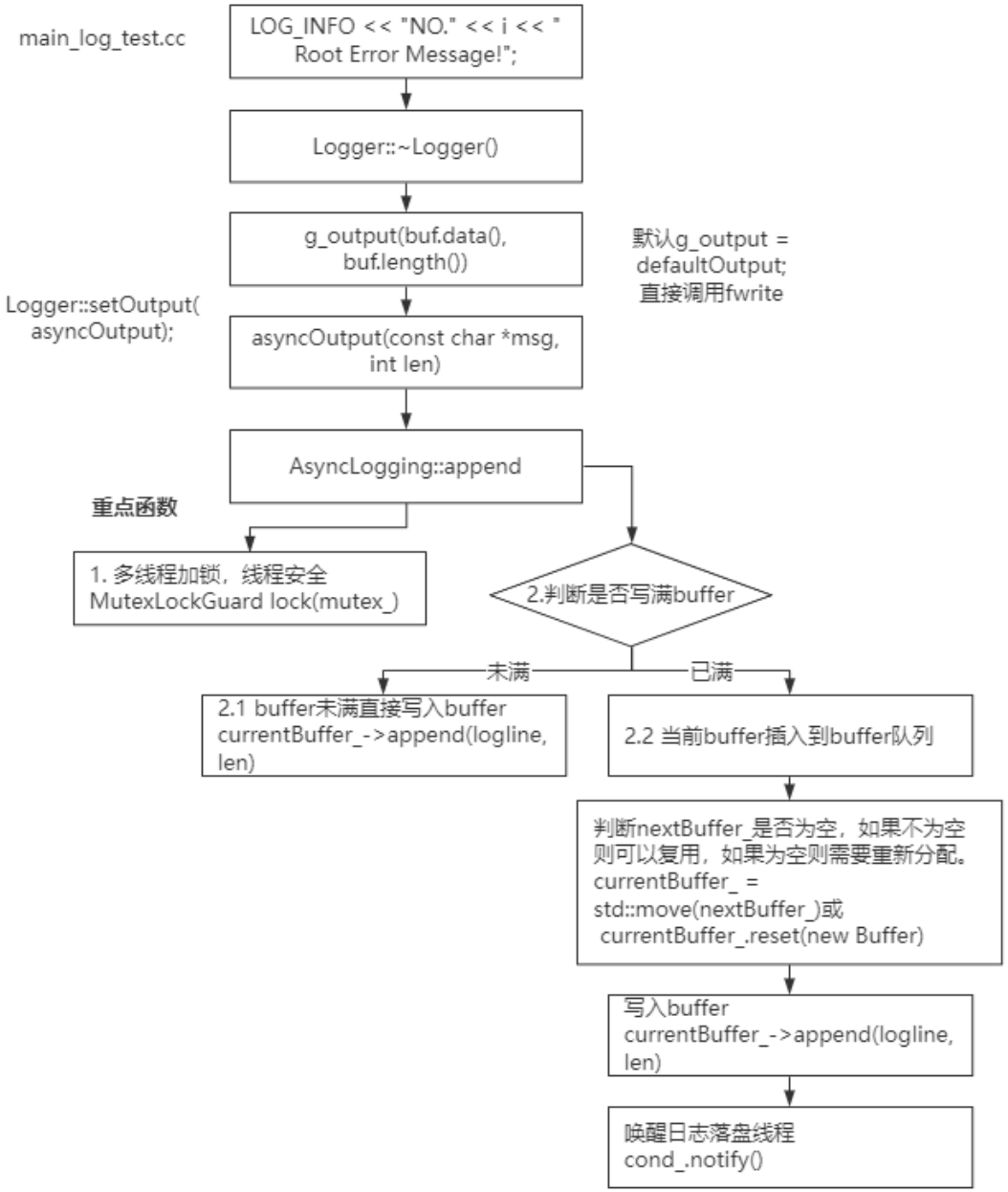
```
AsyncLogging::append
AsyncLogging::threadFunc
```

4.2 前台日志写入栈1–堆栈

```
#0  AsyncLogging::append (this=0x7fffffffd0d0,
    logline=0x7fffffffd2f8 "20210410 14:38:47.161334Z 3
main_log_test.cc:47\n", len=85)
    at /root/0voice/log/muduo_log/AsyncLogging.cc:35
#1  0x000000000042d74f in asyncOutput (
    msg=0x7fffffffd2f8 "20210410 14:38:47.161334Z 311
main_log_test.cc:47\n", len=85)
    at /root/0voice/log/muduo_log/main_log_test.cc:18
#2  0x0000000000425791 in Logger::~Logger (this=0x7
/root/0voice/log/muduo_log/Logging.cc:200
#3  0x000000000042d9fa in testLogPerformance (argc=
/root/0voice/log/muduo_log/main_log_test.cc:47
#4  0x000000000042de17 in main (argc=1, argv=0x7ffff
/root/0voice/log/muduo_log/main_log_test.cc:79
```



4.2 前台日志写入栈2-流程图



4.3 后台日志(落盘)写入栈

#5 0x000000000042d0e0 in **FileUtil::AppendFile::write** (this=0x7ffff0001c80,
logline=0x7ffff6b7e018 "20210410 09:54:23.342297Z 30836 INFO NO.0 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342314Z 30836 INFO NO.1 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342315Z 3083"..., len=3999942)
at /root/0voice/log/muduo_log/FileUtil.cc:63
#6 0x000000000042cfcd in **FileUtil::AppendFile::append** (this=0x7ffff0001c80,
logline=0x7ffff6b7e018 "20210410 09:54:23.342297Z 30836 INFO NO.0 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342314Z 30836 INFO NO.1 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342315Z 3083"..., len=3999942)
at /root/0voice/log/muduo_log/FileUtil.cc:34
#7 0x000000000043028b in **LogFile::append_unlocked** (this=0x7ffff67ab9b0,
logline=0x7ffff6b7e018 "20210410 09:54:23.342297Z 30836 INFO NO.0 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342314Z 30836 INFO NO.1 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342315Z 3083"..., len=3999942)
at /root/0voice/log/muduo_log/LogFile.cc:66
#8 0x000000000043014d in **LogFile::append** (this=0x7ffff67ab9b0,
logline=0x7ffff6b7e018 "20210410 09:54:23.342297Z 30836 INFO NO.0 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342314Z 30836 INFO NO.1 Root Error Message! -
main_log_test.cc:47\n20210410 09:54:23.342315Z 3083"..., len=3999942)
at /root/0voice/log/muduo_log/LogFile.cc:47
#9 0x00000000004267d7 in **AsyncLogging::threadFunc** (this=0x7fffffffd0d0) at
/root/0voice/log/muduo_log/AsyncLogging.cc:107

4.4 总结

双缓冲
双队列
锁的粒度
move语义
批量日志插入队列
唤醒或超时日志读取写入磁盘

4.5 作业

分析日志崩溃后，怎么通过coredump查找丢失的日志。

1. 需要熟悉gdb
2. 熟悉muduo日志库原理

学院介绍

办学宗旨:

一切只为渴望更优秀的你

办学愿景:

让世界没有难学的技术,
让工程师的生活丰富多彩