开源框架log4cpp和日志模块实现

零声教育 https://ke.qq.com/course/420945?tuin=137bb271
Darren QQ：326873713

重点内容

- 日志配置
- 日志格式化
- 日志输出方式
- 日志的级别分级
- 发生时间和具体线程信息
- 线程安全
- 日志回滚
- 日志来不及写入怎么办

Log4cpp主要用来分析一个日志库的架构
mudo日志主要分析性能以及coredump找回丢失的日志

1 日志库设计性能关键点

1.1 重点

- 批量写入和每笔写入性能的区别
- 批量写入和每笔写入实时性的区别

参考程序 **1-file_test.cpp** 测试。

fwrite 1000000 line, fwrite:10, fflush:1, buf_size:2900000, need time:409ms, ops:2444987
write 1000000 line, write:10, fsync:1, buf_size:2900000, need time:408ms, ops:2450980
fwrite 1000000 line, fwrite:100, fflush:1, buf_size:290000, need time:437ms, ops:2288329
write 1000000 line, write:100, fsync:1, buf_size:290000, need time:419ms, ops:2386634
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:20000, fflush:1, buf_size:1450, need time:4198ms, ops:238208
write 1000000 line, write:20000, fsync:1, buf_size:1450, need time:3043ms, ops:328623
fwrite 1000000 line, fwrite:28571, fflush:1, buf_size:1015, need time:4213ms, ops:237360
write 1000000 line, write:28571, fsync:1, buf_size:1015, need time:4221ms, ops:236910
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
当每次写入的数据变小的时候，write性能急剧下降。

1.2 IO接口

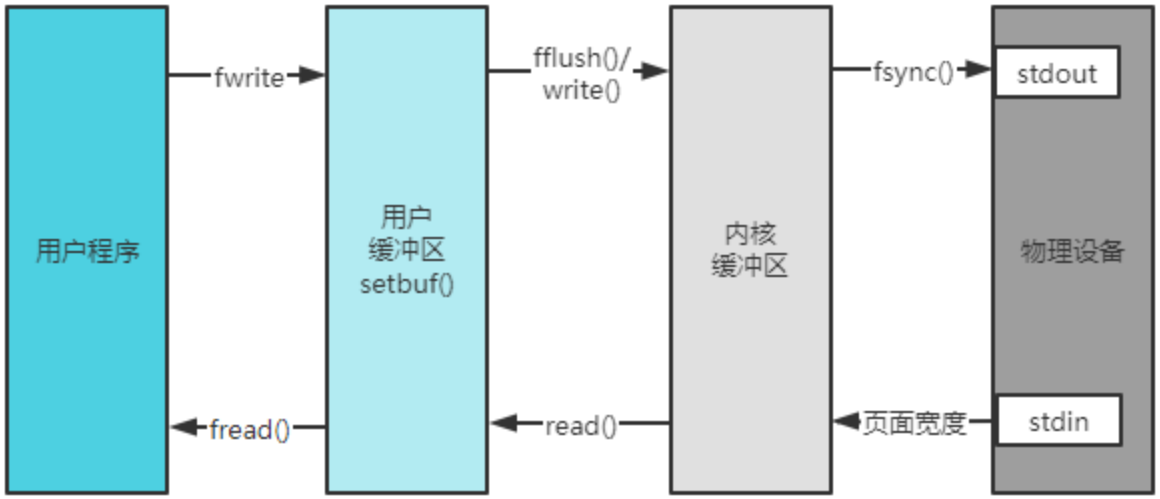
read/write/fsync：

- linux底层操作；
- 内核调用， 涉及到进程上下文的切换，即用户态到核心态的转换，这是个比较消耗性能的操作。

fread/fwrite/fflush：

- c语言标准规定的io流操作，建立在read/write/fsync之上
- 在用户层， 又增加了一层缓冲机制，用于减少内核调用次数，但是增加了一次内存拷贝。

两者之间的关系，见下图：



对于输入设备，调用fsync/fflush将清空相应的缓冲区，其内数据将被丢弃;
对于输出设备或磁盘文件，fflush只能保证数据到达内核缓冲区，并不能保证数据到达物理设备，因此应该在调用fflush后，调用fsync(fileno(stream))，确保数据存入磁盘。

setbuf() 函数的声明。

```
void setbuf(FILE *stream, char *buffer)
```

如果setbuf(fd, NULL);则不缓存。

flush函数和fsync函数对比

1. fflush接受一个参数FILE *.

```
fflush(FILE *);
```

fflush是libc.a中提供的方法，是用来将流中未写的数据传送到内核。如果参数为null，将导致所有流冲洗。

2. fsync接受的时一个Int型的文件描述符

```
fsync(int fd);
```

fsync是系统提供的系统调用。将数据写到磁盘上

fflush/fsync 功能区别

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。

c库缓冲-----fflush----------〉内核缓冲---------fsync-----〉磁盘

sync()、fflush()、fsync()这3个函数的别

a、三者的用途不一样：

- sync,是同步整个系统的磁盘数据的.
- fsync是同步打开的一个文件到缓冲区数据到磁盘上.
- fflush是刷新打开的流的.

b、同样是同步，但三者的同步等级不一样：.

- sync, 将缓冲区数据写回磁盘, 保持同步.(无参数)
- fsync, 将缓冲区的数据写到文件中.(有一个参数 int fd)
- fflush, 将文件流里未写出的数据立刻写出

2 日志库的常见功能

2.1 支持日志级别

比如

Bash 复制代码

```
enum LogLevel
  {
    TRACE,
    DEBUG,
    INFO,
    WARN,
    ERROR,
    FATAL,
    NUM_LOG_LEVELS,
  };
```

在使用日志库的时候要注意不同日志库的级别有差异，比如有些日志库 DEBUG和INFO的级别是反过来的。

2.2 日志格式化

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

2.3 日志输出

日志不同的输出方式

- 日志输出到控制台
- 日志输出到本地文件
- 日志通过网络输出到远程服务器
- ...

不同输出方式是或的关系，支持多种同时输出

2.4 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

2.5 日志配置文件

比如lo4cpp的配置文件

Bash 复制代码

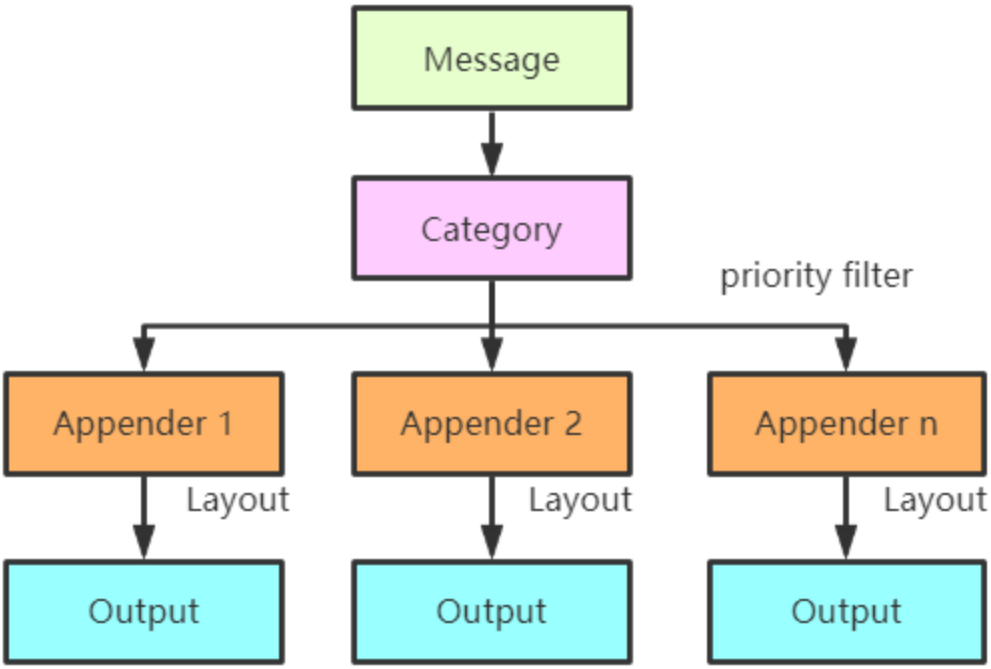
```
# 文件名: test_log4cpp2.conf
# a simple test config
#定义了3个category sub1, sub2, sub1.sub2
log4j.rootCategory=DEBUG, rootAppender
log4j.category.sub1=,A1
log4j.category.sub2=INFO
#log4j.category.sub1.sub2=ERROR, A2
log4j.category.sub1.sub2=, A2
# 设置sub1.sub2 的additivity属性
log4j.additivity.sub1.sub2=true
#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使用

log4cpp是个基于LGPL的开源项目，移植自Java的日志处理跟踪项目log4j，并保持了API上的一致。其类似的支持库还包括Java(log4j)，C++(log4cpp、log4cplus)，C(log4c)，python(log4p)等。

Log4cpp中最重要概念有Category（种类）、Appender（附加器）、Layout（布局）、Priorty（优先级）、NDC（嵌套的诊断上下文）。

Category、Appender与Layout三者的关系如下图所示：



3.1 下载和编译

下载地址：https://sourceforge.net/projects/log4cpp/files/latest/download 我下载的是：log4cpp-1.1.3.tar.gz

解压：

tar zxf log4cpp-1.1.3.tar.gz

编译：

Bash 复制代码

```
cd log4cpp
./configure
make
make check
sudo make install
sudo ldconfig
```

默认安装路径：

头文件：/usr/local/include/log4cpp

库文件：lqf@ubuntu:~/0voice/log/log4cpp$ ls –al /usr/local/lib/liblog4cpp.

liblog4cpp.a liblog4cpp.la liblog4cpp.so liblog4cpp.so.5 liblog4cpp.so.5.0.6

3.2 测试范例

3.2.1 手动使用log4cpp的基本步骤

手动使用log4cpp的基本步骤如下：

1. 实例化一个layout 对象；
2. 初始化一个appender 对象；
3. 把layout对象附着在appender对象上；
4. 调用log4cpp::Category::getInstance("name"). 实例化一个category对象；
5. 把appender对象附到category上（根据additivity的值取代其他appender或者附加在其他appender后）。
6. **设置category的优先级**；

C++ 复制代码

```cpp
// FileName: 2-test_log4cpp.cpp
#include "log4cpp/Category.hh"
#include "log4cpp/FileAppender.hh"
#include "log4cpp/BasicLayout.hh"
int main(int argc, char *argv[])
{
    // 1实例化一个layout 对象
    log4cpp::Layout *layout = new log4cpp::BasicLayout();
    // 2. 初始化一个appender 对象
    log4cpp::Appender *appender = new
log4cpp::FileAppender("FileAppender", "./test_log4cpp1.log");
    // 3. 把layout对象附着在appender对象上
    appender->setLayout(layout);
    // 4. 实例化一个category对象
    log4cpp::Category &warn_log =
        log4cpp::Category::getInstance("darren");
    // 5. 设置additivity为false，替换已有的appender
    warn_log.setAdditivity(false);
    // 5. 把appender对象附到category上
    warn_log.setAppender(appender);
    // 6. 设置category的优先级，低于此优先级的日志不被记录
    warn_log.setPriority(log4cpp::Priority::WARN);
    // 记录一些日志
    warn_log.info("Program info which cannot be wirten");
    warn_log.debug("This debug message will fail to write");
    warn_log.alert("Alert info");
    // 其他记录日志方式
    warn_log.log(log4cpp::Priority::WARN, "This will be a logged
warning");
    log4cpp::Priority::PriorityLevel priority;
    bool this_is_critical = true;
    if (this_is_critical)
        priority = log4cpp::Priority::CRIT;
    else
        priority = log4cpp::Priority::DEBUG;
    warn_log.log(priority, "Importance depends on context");

    warn_log.critStream() << "This will show up << as "
                          << 1 << " critical message";
    // clean up and flush all appenders
    log4cpp::Category::shutdown();
    return 0;
}
```

编译：g++ -o 2-test_log4cpp 2-test_log4cpp.cpp -llog4cpp

这里的输出格式是：BasicLayout::format

3.2.2 配置文件驱动方式使用步骤

基本使用步骤是：

1. 读取解析配置文件；
2. 实例化category对象；
3. 正常使用这些category对象进行日志处理；

配置文件：

C++ 复制代码

```
# 文件名: 3-test_log4cpp.conf
# a simple test config
#定义了3个category sub1, sub2, sub1.sub2
# category 有两个参数 日志级别,Appender
log4cpp.rootCategory=DEBUG, rootAppender
# log4cpp.category.sub1设置日志级别默认和root一致, Appender为A1
log4cpp.category.sub1=,A1
# log4cpp.category.sub2设置为INFO，Appender默认使用root的
log4cpp.category.sub2=INFO
#log4cpp.category.sub1.sub2=ERROR, A2
log4cpp.category.sub1.sub2=, A2
# 设置sub1.sub2 的additivity属性，该属性默认值为true
# 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
log4cpp.additivity.sub1=false
# sub1.sub2的日志也从sub1的appender输出
log4cpp.additivity.sub1.sub2=true
#定义rootAppender类型和layout属性
log4cpp.appender.rootAppender=org.apache.log4cpp.ConsoleAppender
log4cpp.appender.rootAppender.layout=org.apache.log4cpp.BasicLayout
#定义A1的属性
log4cpp.appender.A1=org.apache.log4cpp.FileAppender
log4cpp.appender.A1.fileName=A1.log
log4cpp.appender.A1.layout=org.apache.log4cpp.SimpleLayout
#定义A2的属性
log4cpp.appender.A2=org.apache.log4cpp.ConsoleAppender
log4cpp.appender.A2.layout=org.apache.log4cpp.PatternLayout
#log4cpp.appender.A2.layout.ConversionPattern=The message '%m' at time
# log4cpp.appender.A2.layout.ConversionPattern=%d %m %n
# %d 时间戳 %t 线程名 %x NDC  %p 优先级 %m log message 内容  %n 回车换行
log4cpp.appender.A2.layout.ConversionPattern=%d %p %x - %m%n
```

源码：

C++ 复制代码

```cpp
// FileName: test_log4cpp2.cpp
// Test log4cpp by config file.
#include "log4cpp/Category.hh"
#include "log4cpp/PropertyConfigurator.hh"
#include "log4cpp/NDC.hh"
// 编译: g++ -o 3-test_log4cpp  3-test_log4cpp.cpp -llog4cpp -lpthread

void test(log4cpp::Category &category)
{
    log4cpp::NDC::push(__FUNCTION__); // 记录NDC信息
    category.info("零声学院");
    log4cpp::NDC::pop();
}
int main(int argc, char *argv[])
{
    // 1 读取解析配置文件
    // 读取出错，完全可以忽略，可以定义一个缺省策略或者使用系统缺省策略
    // BasicLayout输出所有优先级日志到ConsoleAppender
    try
    {
        log4cpp::PropertyConfigurator::configure("./3-test_log4cpp.conf");
    }
    catch (log4cpp::ConfigureFailure &f)
    {
        std::cout << "Configure Problem " << f.what() << std::endl;
        return -1;
    }

    // 2 实例化category对象
    // 这些对象即使配置文件没有定义也可以使用，不过其属性继承其父category
    // 通常使用引用可能不太方便，可以使用指针，以后做指针使用
    // log4cpp::Category* root = &log4cpp::Category::getRoot();
    log4cpp::Category &root = log4cpp::Category::getRoot();
    log4cpp::Category &sub1 =
log4cpp::Category::getInstance(std::string("sub1"));
    log4cpp::Category &sub2 =
log4cpp::Category::getInstance(std::string("sub2"));
    log4cpp::Category &sub1_sub2 =
log4cpp::Category::getInstance(std::string("sub1.sub2"));

    // 3 正常使用这些category对象进行日志处理。
    root.debug("root debug");
    root.info("root info");
    root.notice("root notice");
    root.warn("root warn");
    root.error("root error");
    root.crit("root crit");
    root.alert("root alert");
```

```
46        root.fatal("root fatal");
47        root.emerg("root emerg");
48
49        printf("-----------------------------------\n");
50
51        sub1.debug("sub1 debug");
52        sub1.info("sub1 info");
53        sub1.notice("sub1 notice");
54        sub1.warn("sub1 warn");
55        sub1.error("sub1 error");
56        sub1.crit("sub1 crit");
57        sub1.alert("sub1 alert");
58        sub1.fatal("sub1 fatal");
59        sub1.emerg("sub1 emerg");
60        printf("-----------------------------------\n");
61
62        sub2.debug("sub2 debug");
63        sub2.info("sub2 info");
64        sub2.notice("sub2 notice");
65        sub2.warn("sub2 warn");
66        sub2.error("sub2 error");
67        sub2.crit("sub2 crit");
68        sub2.alert("sub2 alert");
69        sub2.fatal("sub2 fatal");
70        sub2.emerg("sub2 emerg");
71        printf("-----------------------------------\n");
72
73        sub1_sub2.debug("sub1_sub2 debug");
74        sub1_sub2.info("sub1_sub2 info");
75        sub1_sub2.notice("sub1_sub2 notice");
76        sub1_sub2.warn("sub1_sub2 warn");
77        sub1_sub2.error("sub1_sub2 error");
78        sub1_sub2.crit("sub1_sub2 crit");
79        sub1_sub2.alert("sub1_sub2 alert");
80        sub1_sub2.fatal("sub1_sub2 fatal");
81        sub1_sub2.emerg("sub1_sub2 emerg");
82        // clean up and flush all appenders
83        log4cpp::Category::shutdown();
84        return 0;
85    }
```

编译：g++ -o 3-test_log4cpp 3-test_log4cpp.cpp -llog4cpp -lpthread

这里的难点是 Category的层级问题。

Category类

Category英中翻译过来是“种类”，但我的理解它应该带有“归类”的意思。它的机制更像是“部门组织机构”，具有树型结构。它将日志按“区域”划分。

Log4cpp有且只一个根Category，可以有多个子Category组成树型结构。Category具有Priority、additivity属性与若干方法。下面列出所有的属性与常用方法。

属性：Name

Category的名称。不可重复。

相关方法

```
virtual const std::string& getName() const throw();
```

属性：Priority

日志的优先级（详情请见Priority章节）：

- 对于根Category，必须指定Priority。(priority < Priority::NOTSET)
- 对于非根Category，可以不指定Priority，此时优先级继承自父Category。
- 对于非根Category，也可以指定Priority，此时优先级覆盖父Category的优先级。

相关方法

```
//设置当前Category的优先级
virtual void setPriority(Priority::Value priority);
//获取当前对象的优先级
virtual Priority::Value getPriority() const throw();
// 设置root Category的优先级
static void setRootPriority(Priority::Value priority);
// 获取root Category的优先级
static Priority::Value getRootPriority() throw();
// 获取当前category继承表中的优先级，如果当前的优先值没有设置的话，则找他的父亲，
// 如果父亲没有找到的话，他会继续向上找，因为root Category的优先值默认为Priority::INFO，
// 所以肯定是能够找到的
virtual Priority::Value getChainedPriority() const throw();
// 返回当前拥有priority优先级
virtual bool isPriorityEnabled(Priority::Value priority) const throw();
```

属性：additivity （叠加、相加性）

每个Category都有一个additivity属性，该属性默认值为true。

- 如果值为true，则该Category的Appender包含了父Category的Appender。
- 如果值为false，则该Category的Appender取代了父Category的Appender。

相关方法

```
virtual void setAdditivity(bool additivity);
    virtual bool getAdditivity() const throw();
```

属性：parent

上级Category。根Category的parent为空。

相关方法

```
virtual Category* getParent() throw();
virtual const Category* getParent() const throw();
```

方法：getRoot

静态方法。取得根Category。

```
static Category& getRoot();
```

方法：getInstance

静态方法。取得指定名称（参数name）的Category，如果不存在，则自动创建一个以name命名，parent为rootCategory，Priority为INFO的Category（说白了就是一个工厂方法，不要和单例模式混淆了）。

```
static Category& getInstance(const std::string& name);
```

方法：exists

静态方法。判断是否存在指定名称（参数name）的Category。如果存在，则返回相应的Category（指针），否则返回空指针。

```
static Category* exists(const std::string& name);
```

方法：shutdown

静态方法。从所有的Category中移除所有的Appender。

```
static void shutdown();
```

方法：shutdownForced

静态方法。从所有的Category中移除所有的Appender，并且删除所有的Appender。（shutdown方法只是不使用Appender，并没有彻底从内存中销毁Appender）。

```
static void shutdownForced();
```

方法：Appender相关

```
// 添加一个Appender到Category中。
// 该方法将把Appender的所有权交给Category管理
virtual void addAppender(Appender* appender);  //appender的生命周期被函数内部（Category）接管
// 添加一个Appender到Category中。
// 但是该方法并不把Appender的所有权交给Category管理
virtual void addAppender(Appender& appender);
```

```
// 获取指定名字的Appender（指针）。如果不存在则返回空指针
virtual Appender* getAppender(const std::string& name) const;
// 获取所有的Appender，以指针的方式存储在set中
virtual AppenderSet getAllAppenders() const;
// 删除所有的Appender
virtual void removeAllAppenders();
// 删除指定的Appender
virtual void removeAppender(Appender* appender);
// 判断指定Appender是否被当前Category拥有所有权。
// 如果是的话，在删除该Appender时，将同时销毁它。
virtual bool ownsAppender(Appender* appender) const throw();
```

注意：

在使用virtual void addAppender(Appender* appender);方法时，Category会接管appender的所有权，并确保会在合适的时机销毁它，所以不要再在外面调用delete方法去销毁它。

如：

```
log4cpp::Appender*appender = new log4cpp::OstreamAppender("default", &std::cout);
root.addAppender(appender);
delete appender;  // Error! Don't delete it
```

方法：日志相关

Category的日志输出方式有两种，一种是简单的传递std::string类型的字符串，一种是采用类似c api中的printf，可以格式化生成字符串。

```
// 以指定的优先级生成日志
virtual void log(Priority::Value priority,
                 const std::string& message) throw();
// 以指定的优先级与格式化生成日志，类似c api中的printf方法
virtual void log(Priority::Value priority, const char* stringFormat,
                 ...) throw();
// 格式化生成日志的一种变体，直接传递va_list参数
virtual void logva(Priority::Value priority, const char* stringFormat,
                   va_list va) throw();
```

除了以上三种通用日志三种方法外，Log4cpp还指供了多种预定义Priority的方法，如：

```
void debug(const char* stringFormat, ...) throw();
void info(const char* stringFormat, ...) throw();
......
```

例子：

```
log4cpp::Category& rootCategory = log4cpp::Category::getRoot();
rootCategory.error("This is error message\n");
rootCategory.log(log4cpp::Priority::DEBUG, "This is %d message\n", 8);
```

方法： CategoryStream

除了C风格的方法外，Log4cpp还指供了c++风格的流式输出方法。模彷std::iostream，Log4cpp也指供了CategoryStream辅助类。每个CategoryStream对象封装一种优先级的日志输出，并提供了对“<<”符号的重载函数。CategoryStream对象内部借由std::ostringstream来实现流式格式化输出生成日志消息(std::string)。

```
// 获取（生成）指定优先级的CategoryStream
virtual CategoryStream getStream(Priority::Value priority);
// 对运算符“<<”的重载。也是获取（生成）指定优先级的CategoryStream
virtual CategoryStream operator<<(Priority::Value priority);
// 快捷方法。还有infoStram()等预定义优先级的快捷方法。
inline CategoryStream debugStream();
.....
```

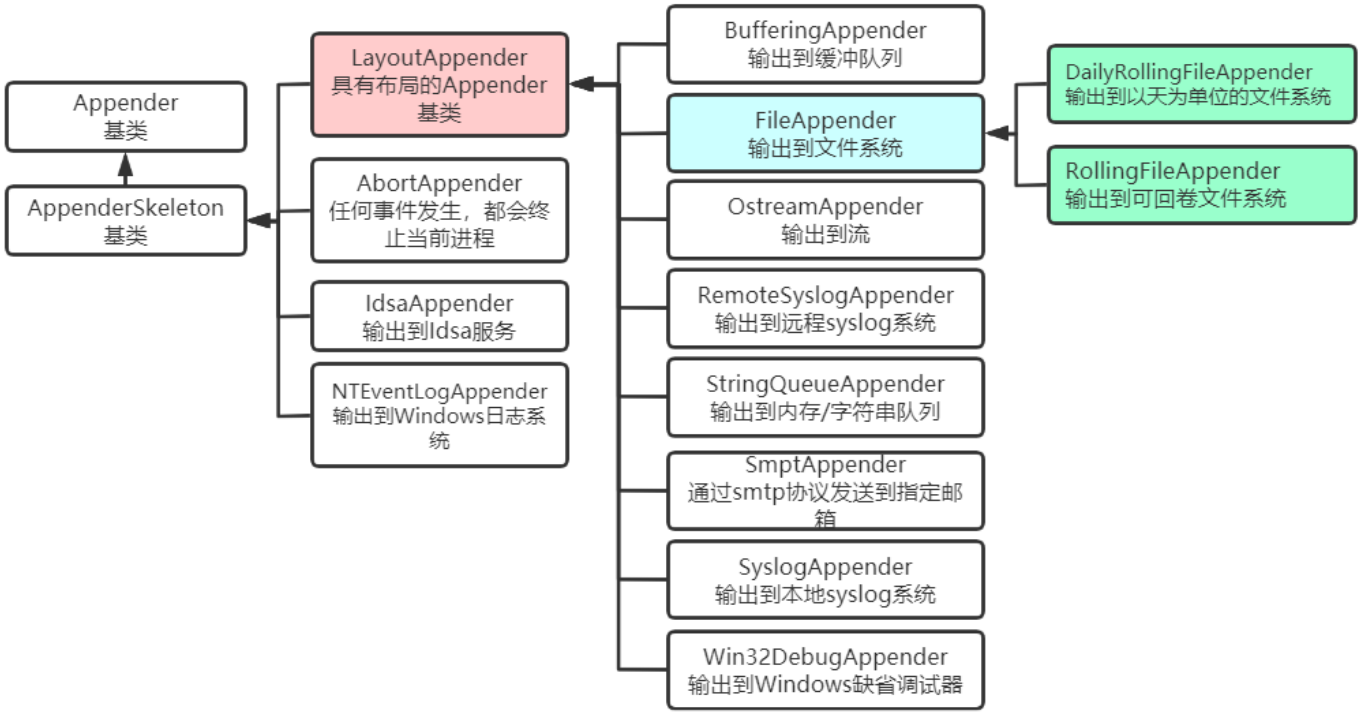
在使用时，CategoryStream，应该只作为临时对象，因为它总是在对象销毁时才会真正将日志输出(flush)。当然你可以强制调用CategoryStream的flush方法，但这样会使代码很难看（日志代码跟实际业务无关，代码行应该尽可能的少，并且可以随时屏蔽日志功能）。

例子：

```
log4cpp::Category& root = log4cpp::Category::getRoot();
root << log4cpp::Priority::ERROR << "This is error message";
root.debugStream() << "This is " << 8 << " message";
```

Appender类

Appender负责将日志写入相应的设备，比如控制台、文件、调试器、Windows日志、syslog等。



FileAppender

作用：输出到文件系统。

构造函数：

```
FileAppender(const std::string& name, const std::string& fileName,
             bool append = true, mode_t mode = 00644);
```

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
filename	string		文件名。 可以是绝对路径，也可以是相对路径。 当文件不存在时，Log4cpp会自动创建文件，但是不会自动创建目录。所以，当filename中的路径不存在时，操作将会失败。
append	boolean	true	是否从现有文件中追加。如果为false，则将会覆盖当前文件。
mode	int	00644	打开文件的权限模式。 参考c api中的open函数。

DailyRollingFileAppender

作用：一种特例化的FileApppender，文件系统以天为单位进行管理，当生成日志时的日期发生变化时，将会生成新的日志文件。

没有限制每个文件的大小

构造函数：

```
DailyRollingFileAppender(const std::string& name,
                         const std::string& fileName,
                         unsigned int maxDaysToKeep = maxDaysToKeepDefault,
                         bool append = true,
                         mode_t mode = 00644);
```

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
filename	string		文件名。 可以是绝对路径，也可以是相对路径。 当文件不存在时，Log4cpp会自动创建文件，但是不会自动创建目录。所以，当filename中的路径不存在时，操作将会失败。
append	boolean	true	是否从现有文件中追加。如果为false，则将会覆盖当前文件。
mode	int	00644	打开文件的权限模式。 参考c api中的open函数。
maxDaysToKeep	int	30	**最多保留多少天的日志**。 超出时间范围的日志文件将被删除。

值得借鉴的在于如果判断是新的一天，以及如何做到保留多少天的日志。

RollingFileAppender

作用：一种特例化的FileAppender，文件系统是文件大小为单位，当日志文件的大小达到预设值时，将会生成新的日志文件。

构造函数：

```
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);

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
filename	string		文件名。 可以是绝对路径，也可以是相对路径。 当文件不存在时，Log4cpp会自动创建文件，但是不会自动创建目录。所以，当filename中的路径不存在时，操作将会失败。
append	boolean	true	是否从现有文件中追加。如果为false，则将会覆盖当前文件。
mode	int	00644	打开文件的权限模式。 参考c api中的open函数。
maxFileSize	int	10M	文件大小最大值。 当日志文件的大小达到该值时，将会生成新的日志文件。 注意：在临界值时，Log4cpp保证了同一条日志并不会输出到两个文件中，以保留可读性。
maxBackupIndex	int	1	保留的日志文件数，该值必须大于或等于1。 当文件数量超出该值时，较早期的文件将被删除。

值得借鉴的是**maxBackupIndex保持多少个日志文件数量。**

对于这里检测文件大小（lseek(_fd, 0, SEEK_END)）的方式效率太低，不值得借鉴。

OstreamAppender

作用：输出到指定流，需指定std:ostream对象。可以是std::cout或std::cerr或其它派生自std::ostream的流对象。

构造函数：OstreamAppender(const std::string& name, std::ostream* stream);

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
stream	std:ostream		可以是std::cout或std::cerr或其它派生自std::ostream的流对象。

范例：

```
log4cpp::OstreamAppender * osAppender = new
log4cpp::OstreamAppender("osAppender",&std::cout);
```

RemoteSyslogAppender

作用：输出到远程syslog系统。

构造函数：

```
RemoteSyslogAppender(const std::string& name,
                     const std::string& syslogName,
                     const std::string& relayer,
                     int facility = LOG_USER,
                     int portNumber = 514);
```

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
syslogName	string		Syslog服务名称
relayer	string		主机名。可以是域名，也可以是IP地址。
facility	int	8	优先级
portNumber	int	514	Syslog的网络服务端口。

SmptAppender

作用：通过smtp协议发送到指定邮箱。

构造函数：

```
SmptAppender(const std::string& name, const std::string& host, const std::string& from,
             const std::string& to, const std::string& subject);
```

相关参数：

名称	类型	默认值	描述
name	string		Appender名称
host	string		Smtp服务器地址
from	string		发件人
to	string		收件人
subject	string		主题

StringQueueAppender

作用：输出到内存/字符串队列。

构造函数：StringQueueAppender(const std::string& name);

相关参数：

名称	类型	默认值	描述
name	string		Appender名称

SyslogAppender

作用：输出到本地syslog系统。

相关参数：SyslogAppender(const std::string& name, const std::string& syslogName,
int facility = LOG_USER);

名称	类型	默认值	描述
name	string		Appender名称
syslogName	string		Syslog服务名称
facility	int	8	优先级

BufferingAppender

作用：输出到缓存队列

Win32DebugAppender

作用：输出到Windows缺省调试器。
相关参数：

名称	类型	默认值	描述
name	string		Appender名称

IdsaAppender

作用：输出到Idsa服务。
相关参数：

名称	类型	默认值	描述
name	string		Appender名称
idsaName	string		Idsa服务名

NTEventLogAppender

输出到Windows日志系统。
相关参数：

名称	类型	默认值	描述
name	string		Appender名称
sourceName	string		Windows日志中的“事件来源”名称

Layout类

Layout控制输出日志的显示样式。Log4cpp内置了4种Layout。
对于不同的日志布局，大家参考：

- 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”。

```
const std::string& priorityName = Priority::getPriorityName(event.priority);
    message.width(Priority::MESSAGE_SIZE);message.setf(std::ios::left);
    message << priorityName << ": " << event.message << std::endl;
```

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”

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

Priority优先级

日志的优先级。Log4cpp内置了10种优先级。

```
typedef enum {
    EMERG  = 0,
    FATAL  = 0,
    ALERT  = 100,
    CRIT   = 200,
    ERROR  = 300,
    WARN   = 400,
    NOTICE = 500,
    INFO   = 600,
    DEBUG  = 700,
    NOTSET = 800
} PriorityLevel;
```

取值越小，优先级越高。例如一个Category的优先级为INFO(600)，则包括EMEGR、FATAL、ALERT、CRIT、ERROR、WARN、INFO等小于或等于600的日志都会被输出，而DEBUG等大于600的日志则不会被输出。
注意1：优先级是一个整数，不一定要只取上面的预定义值，比如说101也是可以的。
注意2：对于根Category，优先级不能大于或等于NOTSET(800)。

NDC

NDC是Nested Diagnostic Contexts的英文缩定，意思就是“嵌套的诊断上下文”，有了它我们可以更好的跟踪程序运行轨迹。

NDC是以线程为基础的，每个线程拥有且只有一个NDC。NDC看起来像一个“栈”。

NDC的几个常用方法是push、pop、get、clear。

Push：把一个字符串压入NDC栈。

Pop：从NDC栈中退出上一次压入的字符串。

Get：取得当前NDC栈中的字符串，每次压入的字符串用空格隔开。

Clear：清空NDC栈中的字符串。

典型的NDC用法可以在每个函数入口处调用NDC::push(__FUNCTION__); 在函数出口处调用NDC::pop(); 在PatternLayout中指定%x参数。这样就可以在日志中清晰的知道函数的调用情况。

Log4cpp配置文件格式说明

log4cpp有3个主要的组件：categories（类别）、appenders（附加目的地）、和 layouts（布局），layout类控制输出日志消息的显示样式（看起来像什么）。log4cpp当前提供以下layout格式：

- log4cpp::IdsaAppender // 发送到IDS或者logger,
- log4cpp::FileAppender // 输出到文件
- log4cpp::RollingFileAppender // 输出到回卷文件，即当文件到达某个大小后回卷

- log4cpp::OstreamAppender // 输出到一个ostream类
- log4cpp::RemoteSyslogAppender // 输出到远程syslog服务器
- log4cpp::StringQueueAppender // 内存队列
- log4cpp::SyslogAppender // 本地syslog
- log4cpp::Win32DebugAppender // 发送到缺省系统调试器
- log4cpp::NTEventLogAppender // 发送到win 事件日志

category 类真正完成记录日志功能，两个主要组成部分是appenders和priority（优先级）。优先级控制哪类日志信息可以被这个category记录，当前优先级分为：NOTSET, DEBUG, INFO, NOTICE, WARN, ERROR, CRIT, ALERT 或 FATAL/EMERG 。每个日志信息有个优先级，每个category有个优先级，当消息的优先级大于等于category的优先级时，这个消息才会被category记录，否则被忽略。优先级的关系如下：
NOTSET < DEBUG < INFO < NOTICE < WARN < ERROR < CRIT < ALERT < FATAL = EMERG

category类和appender的关系是，多个appender附在category上，这样一个日志消息可以同时输出到多个设备上。
category被组织成一个树，子category创建时优先级缺省NOTSET，category缺省会继承父category的appender。而如果不希望这种appender的继承关系，log4cpp允许使用additivity 标签，为false时新的appender取代category的appender列表。

log4cpp可以用手动方式使用，也可以使用配置文件使用，用配置文件的方式简单，便捷。所有，现在都用配置文件的方式。

appender属性（本质是根据不同的appender有不同的熟悉）：
appender类型
fileName 文件名
layout 日志输出格式

比如FileAppender

```
FileAppender(const std::string& name, const std::string& fileName,
             bool append = true, mode_t mode = 00644);
```

即是在配置文件的时候，配置以上属性（name，fileName，append ，mode）。

日志级别

log 的优先级别解读，参阅源码 log4cpp-1.1.3\include\log4cpp\Priority.hh

由高到低

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

对应到 Category 相应函数，参阅源码 log4cpp-1.1.3\include\log4cpp\Category.hh

- `Category::emerg()`
- `Category::fatal()`
- `Category::alert()`
- `Category::crit()`
- `Category::error()`
- `Category::warn()`
- `Category::notice()`
- `Category::info()`
- `Category::debug()`

以上函数都有 2 个重载函数，可分别接受格式化字串或 std::string，例如 debug()，有

- `void debug(const char* stringFormat, ...) throw();`
- `void debug(const std::string& message) throw();`

关于优先级别使用的建议

开发运行时，设为 DEBUG 级，而正式运营时，则设为 NOTICE ；
一定要显示出来的信息则可以用 NOTICE 或以上级别；
跟踪函数运行痕迹的信息用 INFO 级别；
运行时调试的信息用 DEBUG 级别；
举例说明

C++ 复制代码

```
void initialize(int argc, char* argv[])
{
  log.info("initialize() : argc=%d", argc);
  for (int i=0; i < argc; ++i)
  {
    log.debug("initialize() : argv[%d]=%s", i, argv[i]);
  }
  log.notice("initialize() : done");
}
```

log4cpp 的 category 分为 rootCategory 和其它自定义的 category。
而每个 category 都可以输出到多个 appender。并且 category 也是有包含关系的。
例如 rootCategory 就是所有 category 的根。而自定义的 category 也可以在配置文件中定义其包含关系。
先看一个 rootCategory 的配置

```
log4cpp.rootCategory=DEBUG, console, sample
```

这个定义里，指定了 rootCategory 的 log 优先级是 DEBUG，其 appender 有 2 个，分别是 console 和 sample。
即是说，等号右边内容以逗号分隔，第一项是优先级别，接下来的都是 appender 名字，可以有一个或多个。

现在来看看自定义的 category

```
log4cpp.category.demo=DEBUG, sample
```

这里定义了一个名字为 demo 的 category，其优先级为 DEBUG，appender 为 sample。
注意， category 和 appender 名字可以完全相同。

再来看看有包含关系的 category 的定义

```
log4cpp.category.demo.son=DEBUG, son
log4cpp.category.demo.daughter=DEBUG, daughter
```

以上定义了 2 个 category，名字分别为 son 和 daughter，其父 category 为 demo。
son 产生的 log 会写到 son 和 demo 的 appender 中。同理，daughter 的 log 会写到 daughter 和 demo 的 appender 中。

现在来看看 appender 的定义。appender 有很多种，这里只介绍几种，分别是

```
ConsoleAppender : 控制台输出，即 std::cout
Win32DebugAppender : VC IDE 的输出，即 ::OutputDebugString
FileAppender : 文件输出
RollingFileAppender : 回滚文件输出
```

现在看一个 ConsoleAppender 的例子

```
log4cpp.appender.console=ConsoleAppender
log4cpp.appender.console.layout=PatternLayout
log4cpp.appender.console.layout.ConversionPattern=%d [%p] - %m%n
```

以上信息解释为：一个名为 console 的 appender，其类型为 ConsoleAppender，即 控制台输出 log 输出的布局是 指定的样式
输出的格式 是 "%d [%p] - %m%n" (稍后再解释)

再看一个 FileAppender 的例子

```
log4cpp.appender.sample=FileAppender
log4cpp.appender.sample.fileName=sample.log
log4cpp.appender.sample.layout=PatternLayout
log4cpp.appender.sample.layout.ConversionPattern=%d [%p] - %m%n
```

以上信息解释为：一个名为 sample 的 appender，其类型为 FileAppender，即 文件输出指定的 log 文件名为 sample.log，输出的布局是 指定的样式，输出的格式 是 "%d [%p] - %m%n"
对应 category 和 appender 的配置方式，可以发现
category 是 "log4cpp.category." + "category name"
category 名字可以用 "." 分隔，以标识包含关系
appender 是 "log4cpp.appender." + "appender name"
appender 名字 不能 用 "." 分隔，即是说 appender 是没有包含关系的
现在看一个完整的配置文件例子

```
#定义 root category 的属性
log4cpp.rootCategory=DEBUG, console
#定义 console 属性
log4cpp.appender.console=ConsoleAppender
log4cpp.appender.console.layout=PatternLayout
log4cpp.appender.console.layout.ConversionPattern=%d [%p] - %m%n
#定义 sample category 的属性
log4cpp.category.sample=DEBUG, sample
```

```
#定义 sample appender 的属性
log4cpp.appender.sample=FileAppender
log4cpp.appender.sample.fileName=sample.log
log4cpp.appender.sample.layout=PatternLayout
log4cpp.appender.sample.layout.ConversionPattern=%d [%p] - %m%n
#定义 sample.son category 的属性
log4cpp.category.sample.son=DEBUG, son
#定义 son appender 的属性
log4cpp.appender.son=FileAppender
log4cpp.appender.son.fileName=son.log
log4cpp.appender.son.layout=PatternLayout
log4cpp.appender.son.layout.ConversionPattern=%d [%p] - %m%n
#定义 sample.daughter category 的属性
log4cpp.category.sample.daughter=DEBUG, daughter
#定义 daughter appender 的属性
log4cpp.appender.daughter=FileAppender
log4cpp.appender.daughter.fileName=daughter.log
log4cpp.appender.daughter.layout=PatternLayout
log4cpp.appender.daughter.layout.ConversionPattern=%d [%p] - %m%n
```

ConversionPattern 参数解读，参阅源码 log4cpp-1.1.3\src\PatternLayout.cpp

```
%m log message 内容, 即 用户写 log 的具体信息
%n 回车换行
%c category 名字
%d 时间戳
%p 优先级
%r 距离上一次写 log 的间隔, 单位毫秒
%R 距离上一次写 log 的间隔, 单位秒
%t 线程号
%u 处理器时间
%x NDC ?
```

窃以为，以下格式就足够了，即输出 "时间 [线程号] 优先级 - log内容 回车换行"

```
%d [%t] %p - %m%n
```

配置的知识就差不多了，现在看看实际代码应用

首先是初始化 log4cpp 的配置，例如我的配置文件名是 log4cpp.properties

C++ 复制代码

```cpp
try
{
  log4cpp::PropertyConfigurator::configure("log4cpp.properties");
}
catch (log4cpp::ConfigureFailure & f)
{
  std::cerr << "configure problem " << f.what() << std::endl;
}
```

初始化完成后，就可以这样用了（具体的应用技巧，你自己摸索吧）

C++ 复制代码

```cpp
log4cpp::Category & log =
log4cpp::Category::getInstance(std::string("sample"));
log.debug("test debug log");
log.info("test info log");
// 用 sample.son
log4cpp::Category & log =
log4cpp::Category::getInstance(std::string("sample.son"));
log.debug("test debug log of son");
log.info("test info log of son");
// 用 sample.daughter
log4cpp::Category & log =
log4cpp::Category::getInstance(std::string("sample.daughter"));
log.debug("test debug log of daughter");
log.info("test info log of daughter");
```

4 Log4cpp源码分析

参考：《log4cpp源码详解手册》

log4cpp::FileAppender::_append 调用栈

结合4-RollingFileAppender.cpp

Breakpoint 3, 0x00007ffff7b9ba9a in log4cpp::FileAppender::_append(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
(gdb) bt
#0 0x00007ffff7b9ba9a in log4cpp::FileAppender::_append(log4cpp::LoggingEvent const&) () from /usr/local/lib/liblog4cpp.so.5
#1 0x00007ffff7b92b9c in log4cpp::AppenderSkeleton::doAppend(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
#2 0x00007ffff7ba5bb4 in log4cpp::Category::callAppenders(log4cpp::LoggingEvent const&) () from /usr/local/lib/liblog4cpp.so.5
#3 0x00007ffff7ba5e34 in log4cpp::Category::_logUnconditionally2(int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) () from /usr/local/lib/liblog4cpp.so.5
#4 0x00007ffff7ba6736 in log4cpp::Category::error(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) () from /usr/local/lib/liblog4cpp.so.5
#5 0x0000000000401b77 in main (argc=1, argv=0x7fffffffe4d8) at **4-RollingFileAppender.cpp**:100

log4cpp::RollingFileAppender::_append

结合4-RollingFileAppender.cpp
Breakpoint 2, 0x00007ffff7b9d7b8 in log4cpp::RollingFileAppender::_append(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
(gdb) bt
#0 0x00007ffff7b9d7b8 in log4cpp::RollingFileAppender::_append(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
#1 0x00007ffff7b92b9c in log4cpp::AppenderSkeleton::doAppend(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
#2 0x00007ffff7ba5bb4 in log4cpp::Category::callAppenders(log4cpp::LoggingEvent const&) () from /usr/local/lib/liblog4cpp.so.5
#3 0x00007ffff7ba5e34 in log4cpp::Category::_logUnconditionally2(int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) () from /usr/local/lib/liblog4cpp.so.5
#4 0x00007ffff7ba6736 in log4cpp::Category::error(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) () from /usr/local/lib/liblog4cpp.so.5

#5 0x0000000000401b9e in main (argc=1, argv=0x7fffffffe4d8) at **4-RollingFileAppender.cpp:100**

重点

C++ 复制代码

```cpp
void RollingFileAppender::rollOver() {
        ::close(_fd);
        if (_maxBackupIndex > 0) {
            std::ostringstream filename_stream;
            filename_stream << _fileName << "." << std::setw( _maxBackupIndexWidth ) << std::setfill( '0' ) << _maxBackupIndex << std::ends;
            // remove the very last (oldest) file
            std::string last_log_filename = filename_stream.str();
            // std::cout << last_log_filename << std::endl; // removed by request on sf.net #140
            ::remove(last_log_filename.c_str());

            // rename each existing file to the consequent one 修改文件序号
            for(unsigned int i = _maxBackupIndex; i > 1; i--) {
                filename_stream.str(std::string());
                filename_stream << _fileName << '.' << std::setw( _maxBackupIndexWidth ) << std::setfill( '0' ) << i - 1 << std::ends;  // set padding so the files are listed in order
                ::rename(filename_stream.str().c_str(), last_log_filename.c_str());
                last_log_filename = filename_stream.str();
            }
            // new file will be numbered 1
            ::rename(_fileName.c_str(), last_log_filename.c_str());
        }
        _fd = ::open(_fileName.c_str(), _flags, _mode);
    }
```

log4cpp::StringQueueAppender::_append

#0 0x00007ffff7b9e692 in log4cpp::StringQueueAppender::_append(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5
#1 0x00007ffff7b92b9c in log4cpp::AppenderSkeleton::doAppend(log4cpp::LoggingEvent const&) ()
  from /usr/local/lib/liblog4cpp.so.5

#2 0x00007ffff7ba5bb4 in log4cpp::Category::callAppenders(log4cpp::LoggingEvent const&) () from /usr/local/lib/liblog4cpp.so.5
#3 0x00007ffff7ba5e34 in log4cpp::Category::_logUnconditionally2(int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) () from /usr/local/lib/liblog4cpp.so.5
#4 0x00007ffff7ba5d54 in log4cpp::Category::_logUnconditionally(int, char const*, __va_list_tag*) ()
   from /usr/local/lib/liblog4cpp.so.5
#5 0x00007ffff7ba66c9 in log4cpp::Category::error(char const*, ...) () from /usr/local/lib/liblog4cpp.so.5
#6 0x000000000040288b in main (argc=1, argv=0x7fffffffe4d8) at **5-StringQueueAppender.cpp**:42

5 mudo日志库分析

重点

- 性能分析
- 文件批量写入
- 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mov语义减少内存拷贝
- coredump日志丢失找回分析

见课程分析。
源码
muduo日志库是C++ stream风格，这样用起来更自然，不必费心保持格式字符串和参数类型的一致性，可以随用随写，而且是类型安全的。

前台日志写入栈

#0 AsyncLogging::append (this=0x7fffffffd0d0,
   logline=0x7fffffffd2f8 "20210410 14:38:47.161334Z 31101 INFO NO.0 Root Error Message! - 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 31101 INFO NO.0 Root Error Message! – 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=0x7fffffffd2e0, __in_chrg=<optimized out>) at /root/0voice/log/muduo_log/Logging.cc:200
#3 0x000000000042d9fa in testLogPerformance (argc=1, argv=0x7fffffffe4e8) at /root/0voice/log/muduo_log/main_log_test.cc:47
#4 0x000000000042de17 in main (argc=1, argv=0x7fffffffe4e8) at /root/0voice/log/muduo_log/main_log_test.cc:79

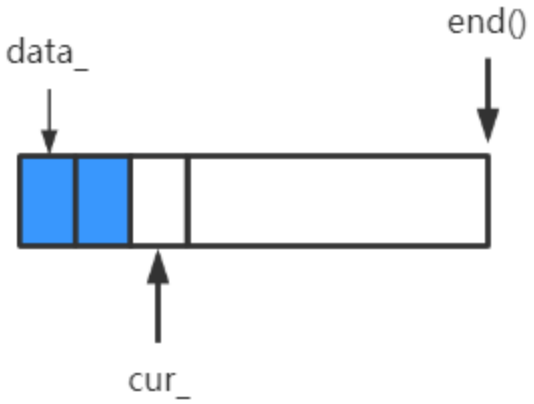
后台日志写入栈

#0 write () at ../sysdeps/unix/syscall-template.S:84
#1 0x00007ffff72d0c0f in _IO_new_file_write (f=0x7ffff00009a0, data=0x7ffff6b7e018, n=3997696) at fileops.c:1263
#2 0x00007ffff72d139a in new_do_write (to_do=3997696,
data=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"..., fp=0x7ffff00009a0) at fileops.c:518
#3 _IO_new_file_xsputn (f=0x7ffff00009a0, data=<optimized out>, n=3999942) at fileops.c:1342
#4 0x00007ffff72d020a in __GI_fwrite_unlocked (buf=<optimized out>, size=1, count=3999942, fp=<optimized out>) at iofwrite_u.c:43
#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

类设计

FixedBuffer的设计



预先分配好的buffer

LogStream类设计

负责数据的格式化

格式化后的数据存储到Buffer buffer_;

Logger类设计

负责对接 LOG_INFO << "NO." << i << " Root Error Message!";

日志前端
Logger –> Impl –> LogStream –> operator << –> FixedBuffer –> g_output [g_output]

LogFile负责文件写入逻辑

AppendFile 真正写入文件

AsyncLogging 负责日志异步逻辑和线程循环

日志滚动原理

- 一种是日志文件大小达到预设值（一秒中最多创建一个文件）
- 另一种是时间到达超过当天。

coredump查找丢失的日志

https://www.yuque.com/docs/share/b29be8e3-765c-4937-891e-a2d6663b381d?# 《muduo异步日志——core dump查找未落盘的日志》

6 升级gdb版本

gdb不正常才升级。

问题：

Type "apropos word" to search for commands related to "word"...

Reading symbols from ./main_log_test...Dwarf Error: wrong version in compilation unit header (is 5, should be 2, 3, or 4) [in module /mnt/hgfs/log/log/build/main_log_test]

在讲c++11的时候把gcc升级到了11.2版本，然后发现老版本的gdb出现不匹配。

下载：

wget http://ftp.gnu.org/gnu/gdb/gdb-11.1.tar.gz

解压：

tar -zxvf gdb-11.1.tar.gz

编译gdb过程中需要使用texinfo, 先安装texinfo

sudo apt-get install texinfo

编译：

Bash 复制代码

```
cd  gdb-11.1
./configure
make
sudo make install
```

gdb --version