Table of Contents

[一. Logging介绍 2](#_Toc47667644)

[二. 使用说明 2](#_Toc47667645)

[1. 输出日志信息 2](#_Toc47667646)

[2. 写入文档内容 5](#_Toc47667647)

[3. 日志和文档内容混合写入 6](#_Toc47667648)

[三. 核心组件 8](#_Toc47667649)

[1. 挂接器 10](#_Toc47667650)

[2. 布局器 12](#_Toc47667651)

[3. 嵌入诊断上下文NDC 13](#_Toc47667652)

[4. 脚本配置 14](#_Toc47667653)

[5. 其他 17](#_Toc47667654)

# Logging介绍

Logging是C++编写的日志系统，它提供了在C++程序中使用日志和跟踪调试的功能。使用logging系统能够非常便利地将日志或者跟踪调试信息写入字符流、内存字符串队列、文件、回滚文件、调试器、Windows日志、本地syslog和远程syslogserver中。程序移植自Java的日志处理跟踪项目log4j，并保持了API上的一致，源代码受Apache Public License 2.0保护，具有跨平台、线程安全、灵活、以及多粒度控制的特点，通过将日志划分优先级使其可以面向程序调试、运行、测试、和维护等全生命周期。

**Logging日志系统有例如以下优势：**

• 提供了可扩展的多种日志记录方式：控制台、滚动日志、频度日志、socket日志、数据库、服务器等等；

• 提供了NDC（嵌套诊断上下文），可用于多线程、多场景的跟踪调试；

• 提供了完整的日志动态优先级控制，可随时调整须要记录的日志优先级；

• 可通过配置文件完毕全部配置并动态载入和实时监听；

• 性能优秀，内存占用小；

• 代码级的平台无关性，支持Windows with VS2017、Linux with GCC v8.3，适用于大多数主流的操作系统和开发工具；

• 条理清晰，逻辑简单，极易上手，日志操作仅需三步：初始化、创建日志文件、写入日志。

• 应用灵活，具备复杂日志系统的一切高级应用，也支持进一步应用封装。

• 支持文档文件写入，也就是说，Logging不仅仅只具备写入日志信息功能，他还是一款非常优秀的文档生成工具（xml、html、csv等等）。

# 使用说明

Logging日志系统具备传统的日志信息操作，也适用于文档写入，同时具备控制台输出和日志文档输出功能，文档按照设定大小和备份个数滚动保存。

## 输出日志信息

Logging日志系统使用简单，极易上手，日志操作仅需要三步骤：初始化、创建日志记录器、写入日志。

第1步：用Initializer类初始化logging系统

#include <logging/logging.h>

int main()

{

logging::Initializer initializer; // 用Initializer类进行初始化

.

.

.

}

注意：要在主程序main()函数的初始部分声明initializer的对象（仅声明一次即可），initializer会自动初始化logging系统，在程序结束时也会自动关闭所有内存、线程等资源。即使在程序出现严重异常导致程序中断，也能完整保存最后一次写入的日志信息，保证了跟踪日志的完整性。

第2步：创建logger日志记录器

auto logger = logging::create(LOGGING\_TEXT("../output/demo1.log"));

logging::create()函数参数：

filename设置日志文件路径，文件扩展名可以任意设定，需要存储xml文档，尾缀扩展名改为.xml即可。

maxFileSize设置输出日志文件大小（默认10MB），以byte为单位，如果超出设定值将按照maxBackupIndex设置卷数进行备份。

maxBackupIndex设置最大滚动卷数（默认10），如果超出设定卷数，将会从第1卷进行覆盖。

LOGGING\_EXPORT Logger create(const logging::tstring& filename,

long maxFileSize = 10 \* 1024 \* 1024, // Size = 10 MB

int maxBackupIndex = 10); // Backup = 10

日志记录器（Logger）是日志处理的核心组件，通过logging::create()函数得到日志记录器（Logger），可以在控制台和滚动文件中同步输出相同格式信息，输出日志格式默认为：

年/月/日 时:分:秒,毫秒 [级别] 信息 [源文件路径:行号]

第3步：使用宏将日志输出

LOGGING\_TRACE(logger, "This is the trace log message...");

LOGGING\_DEBUG(logger, "This is the debug log message...");

LOGGING\_INFO(logger, "This is the info log message...");

LOGGING\_WARN(logger, "This is the warn log message...");

LOGGING\_ERROR(logger, "This is the error log message...");

LOGGING\_FATAL(logger, "This is the fatal log message...");

logging日志级别从高到低排序：FATAL、ERROR、WARN、INFO、DEBUG、TRACE

通过定义日志级别，可以控制到应用程序中相应级别的日志信息的开关。

logging::create函数得到日志记录器（Logger）默认级别为ALL\_LOG\_LEVEL。

比如定义了INFO级别，则应用程序中所有DEBUG和TRACE级别的日志信息将不被打印出来，也是说大于等于的级别的日志才输出。

#include <logging/logging.h>

int main()

{

logging::Initializer initializer;

auto logger = logging::create(LOGGING\_TEXT("../output/demo1.log"));

logger.setLogLevel(logging::INFO\_LOG\_LEVEL); // 设置日志级别

for (size\_t i = 0; i < 10; i++)

{

LOGGING\_TRACE(logger, "Entering loop #" << i << " This is the trace log message...");

LOGGING\_DEBUG(logger, "Entering loop #" << i << " This is the debug log message...");

LOGGING\_INFO(logger, "Entering loop #" << i << " This is the info log message...");

LOGGING\_WARN(logger, "Entering loop #" << i << " This is the warn log message...");

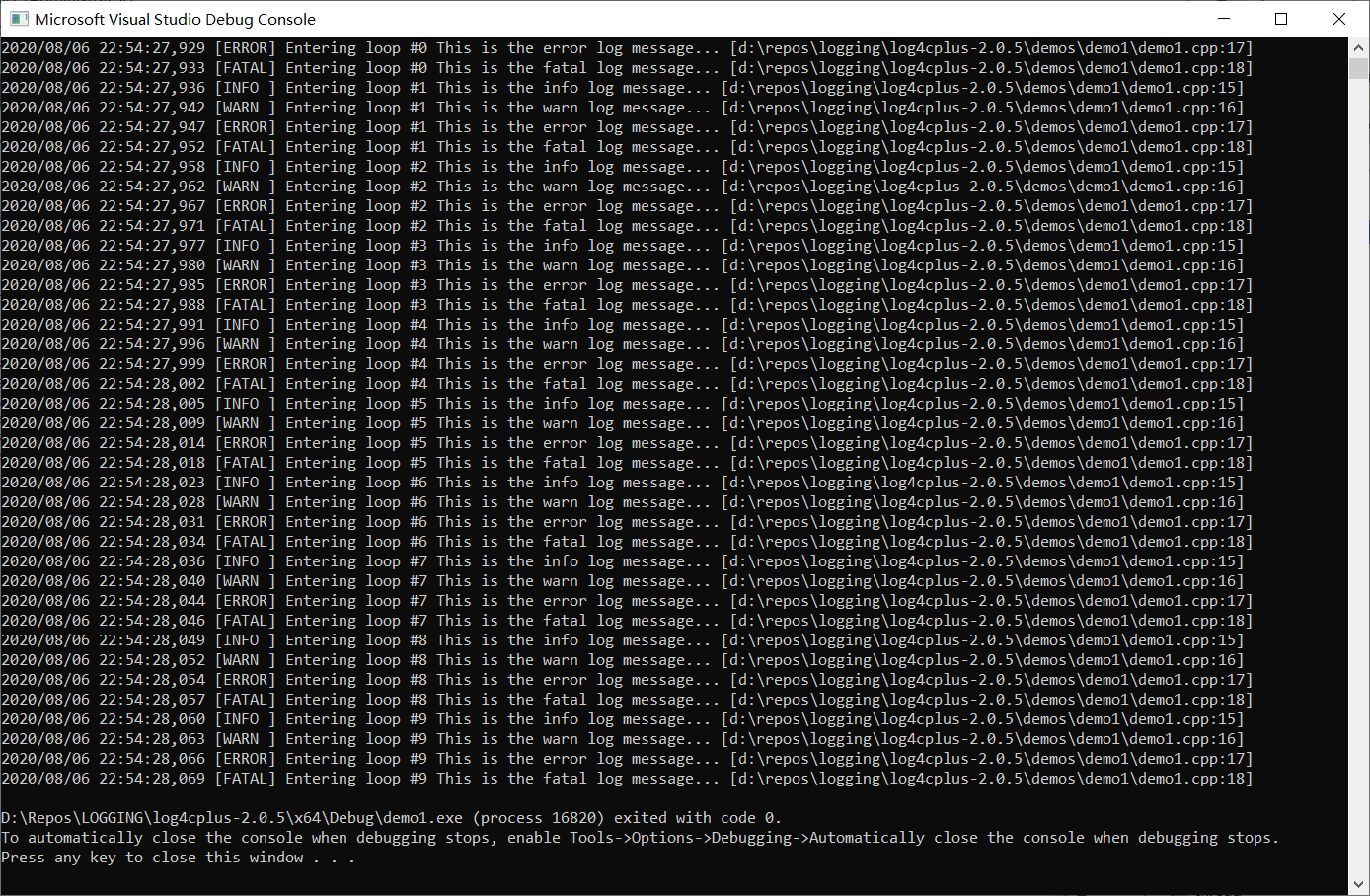
LOGGING\_ERROR(logger, "Entering loop #" << i << " This is the error log message...");

LOGGING\_FATAL(logger, "Entering loop #" << i << " This is the fatal log message...");

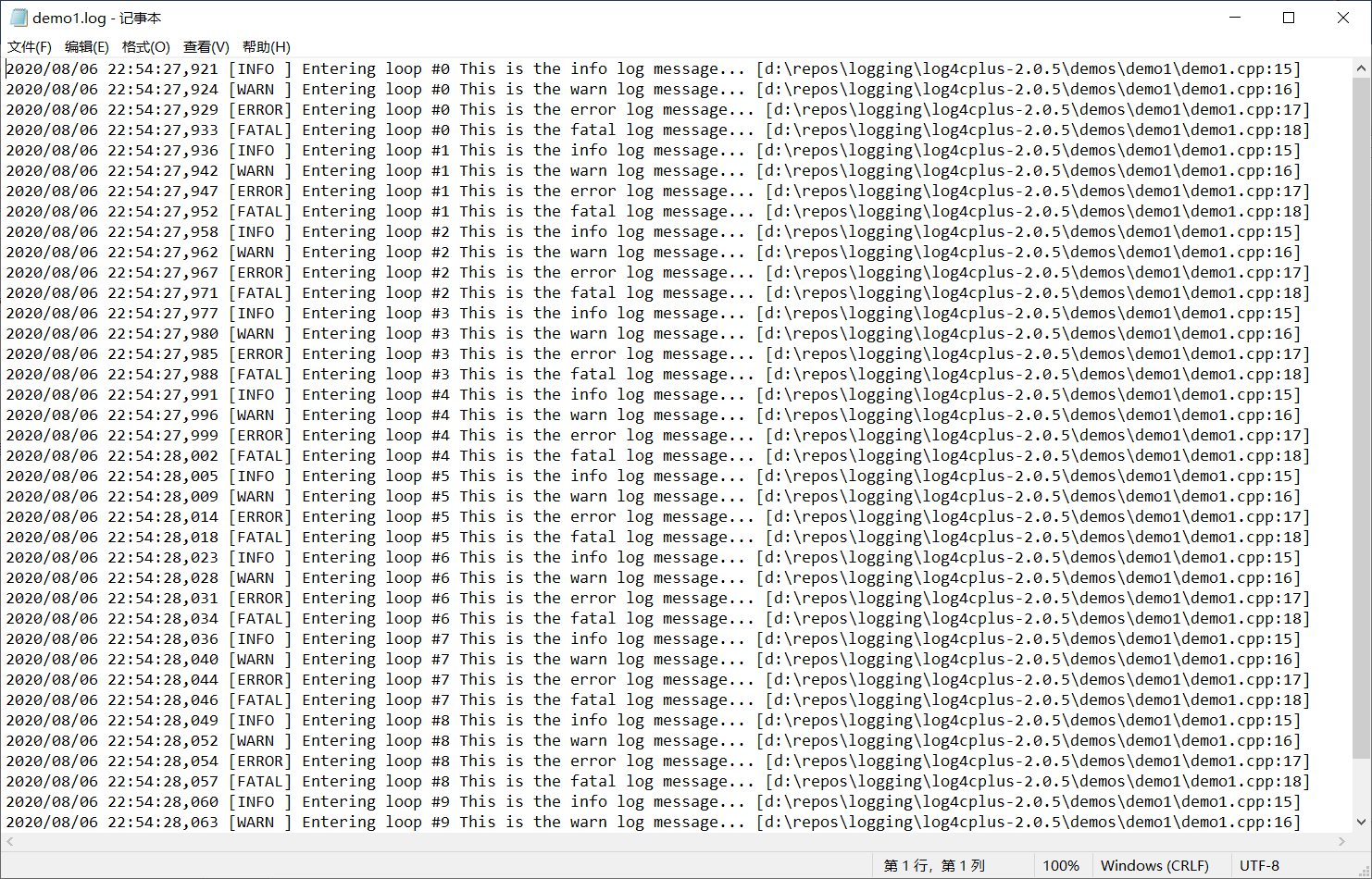
}

}

以下是控制台显示结果：



以下是日志文件保存结果：



Logging选用C++的流机制进行日志输出，为了区分包含<<运算符和不包含<<运算符的日志内容，分别提供了LOGGING \_XXXX和LOGGING \_XXXX\_STR两种日志输出宏。 另外，日志输出宏LOGGING\_TRACE\_METHOD主要用来跟踪方法的调用轨迹。

#include <logging/logging.h>

#include <iomanip>

using namespace std;

using namespace logging;

int main()

{

logging::Initializer initializer;

SharedAppenderPtr appender(new ConsoleAppender());

auto root = Logger::getRoot();

root.addAppender(appender);

LOGGING\_DEBUG(root, "This is" << " a reall" << "y long message." << std::endl << "Just testing it out" << std::endl << "What do you think?");

LOGGING\_DEBUG(root, "This is a bool: " << true);

LOGGING\_INFO(root, "This is a char: " << 'x');

LOGGING\_INFO(root, "This is a short: " << static\_cast<short>(-100));

LOGGING\_INFO(root, "This is a unsigned short: " << static\_cast<unsigned short>(100));

LOGGING\_INFO(root, "This is a int: " << 1000);

LOGGING\_INFO(root, "This is a unsigned int: " << 1000u);

LOGGING\_INFO(root, "This is a long(hex): " << std::hex << 100000000l);

LOGGING\_INFO(root, "This is a unsigned long: " << 100000000ul);

LOGGING\_WARN(root, "This is a float: " << 1.2345f);

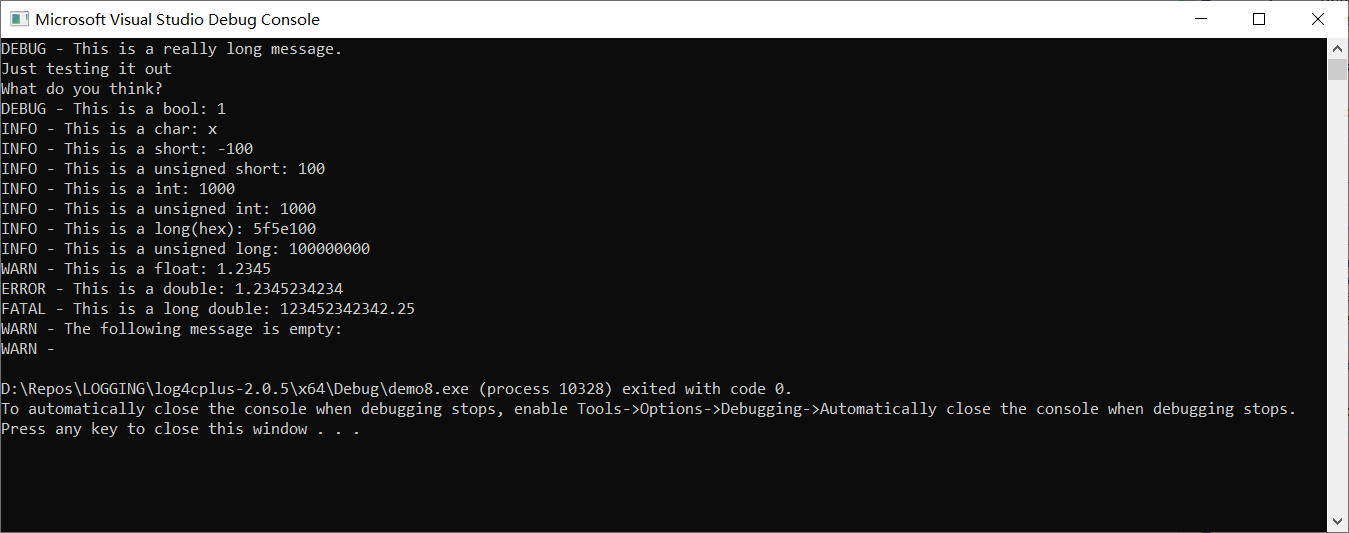
LOGGING\_ERROR(root, "This is a double: " << setprecision(15) << 1.2345234234);

LOGGING\_FATAL(root, "This is a long double: " << setprecision(15) << 123452342342.25L);

LOGGING\_WARN(root, "The following message is empty:");

LOGGING\_WARN(root, "");

}



## 写入文档内容

通过logger.write()函数，可以将信息原样写入文档（一次写入，多次写入），写入文档不包含任何格式。

#include "..\include\logging\logging.h"

int main()

{

logging::Initializer initializer;

auto logger = logging::create(LOGGING\_TEXT("../output/demo2.txt"));

for (size\_t j = 0; j < 10; j++)

{

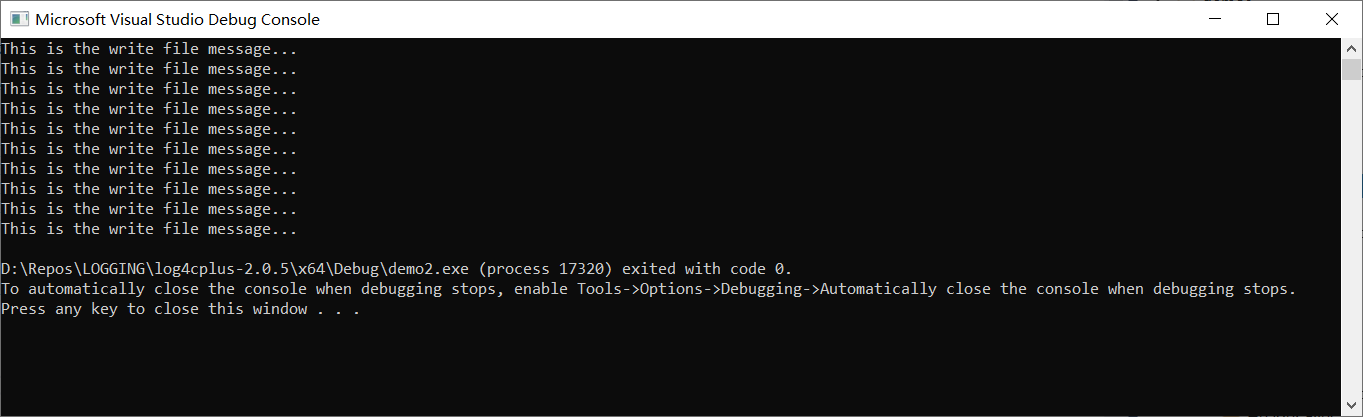
// 写入文档内容

logger.write(LOGGING\_TEXT("This is the write file message..."));

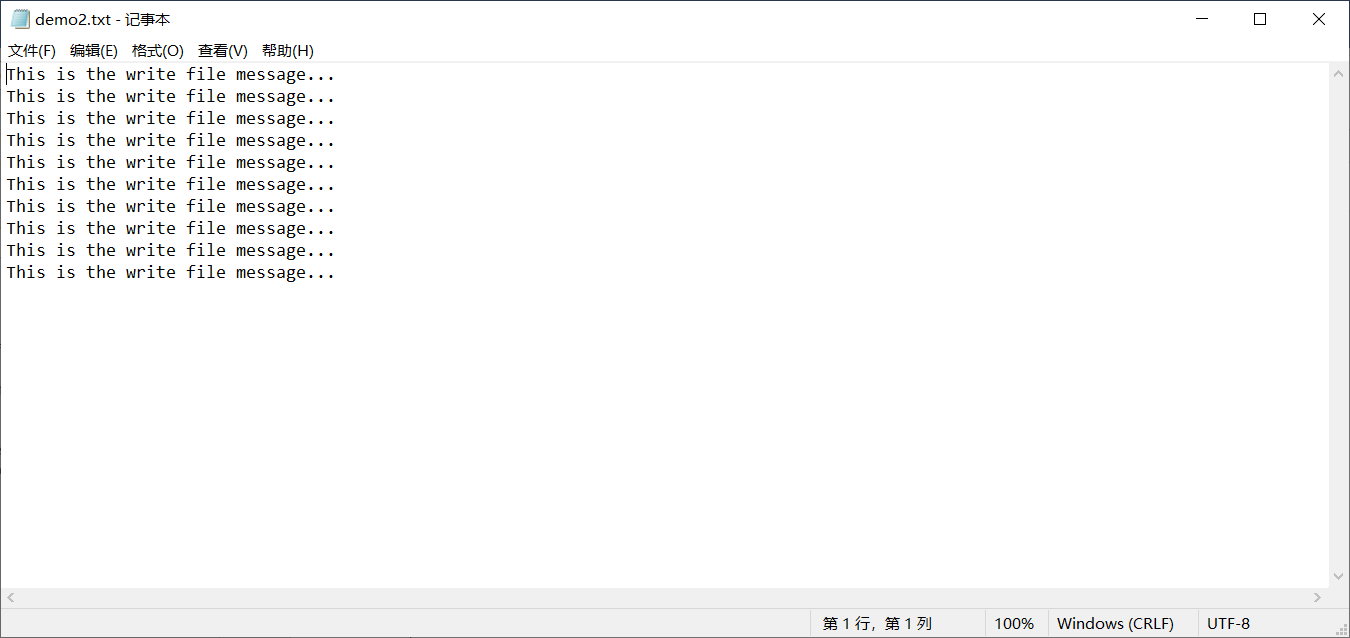
}

}

以下是控制台显示结果：



以下是日志文件保存结果：



## 日志和文档内容混合写入

#include <logging/logging.h>

int main()

{

logging::Initializer initializer;

auto logger = logging::create(LOGGING\_TEXT("../output/demo3.txt"));

for (size\_t i = 0; i < 10; i++)

{

LOGGING\_TRACE(logger, "Entering loop #" << i << " This is the trace log message...");

LOGGING\_DEBUG(logger, "Entering loop #" << i << " This is the debug log message...");

LOGGING\_INFO(logger, "Entering loop #" << i << " This is the info log message...");

LOGGING\_WARN(logger, "Entering loop #" << i << " This is the warn log message...");

LOGGING\_ERROR(logger, "Entering loop #" << i << " This is the error log message...");

LOGGING\_FATAL(logger, "Entering loop #" << i << " This is the fatal log message...");

for (size\_t j = 0; j < 10; j++)

{

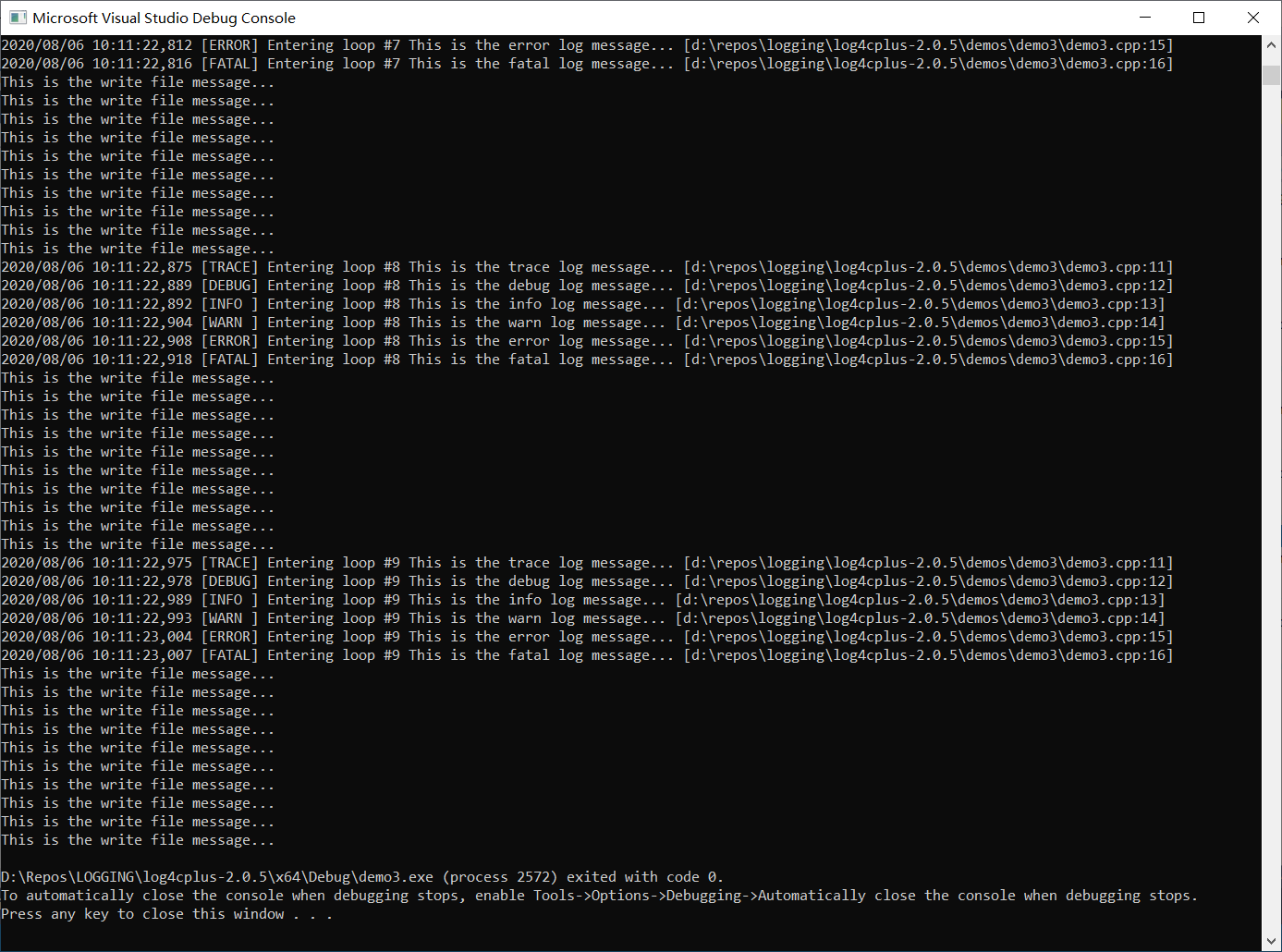
logger.write(LOGGING\_TEXT("This is the write file message..."));

}

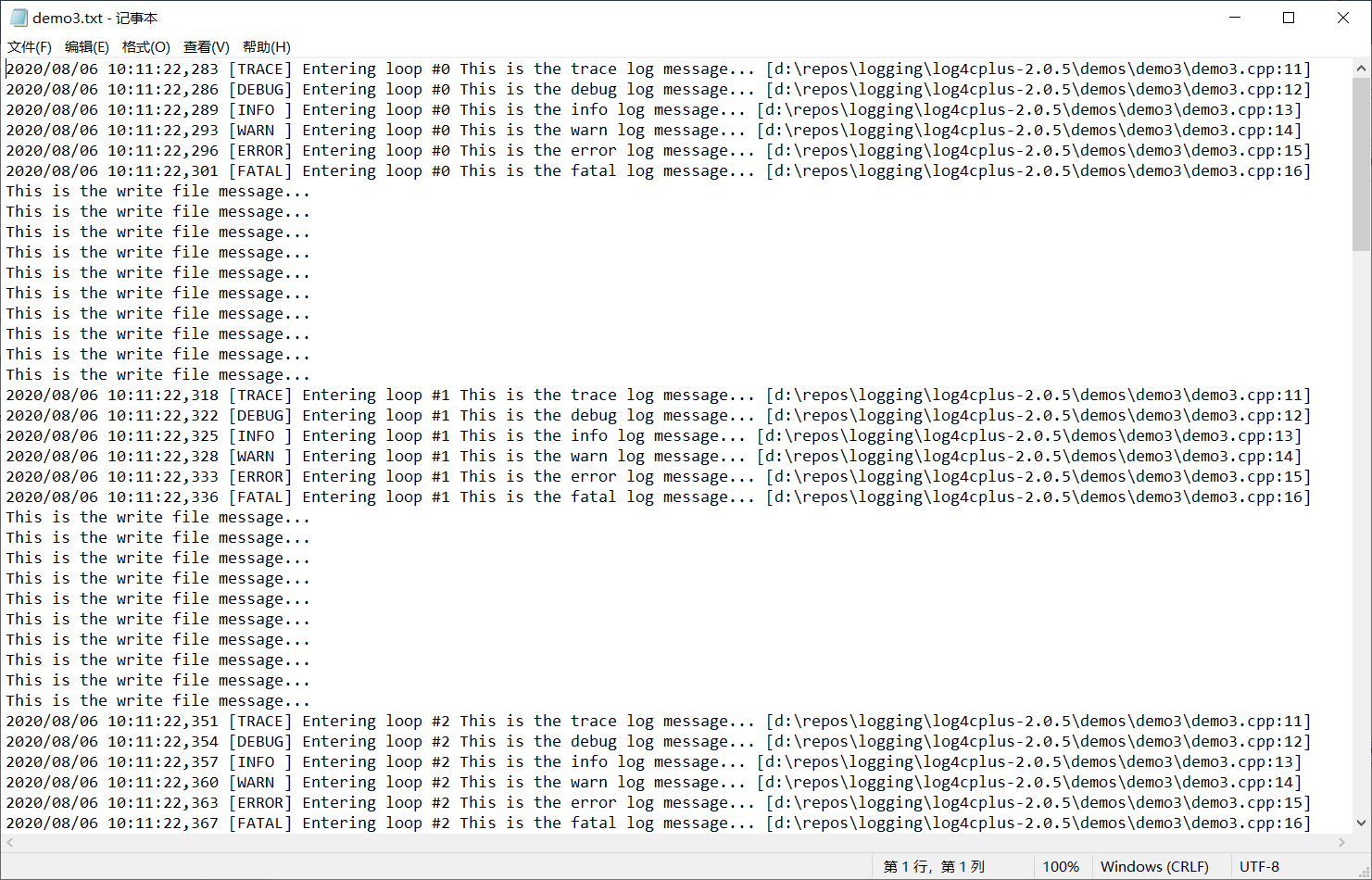
}

}

以下是控制台显示结果：



以下是日志文件保存结果：

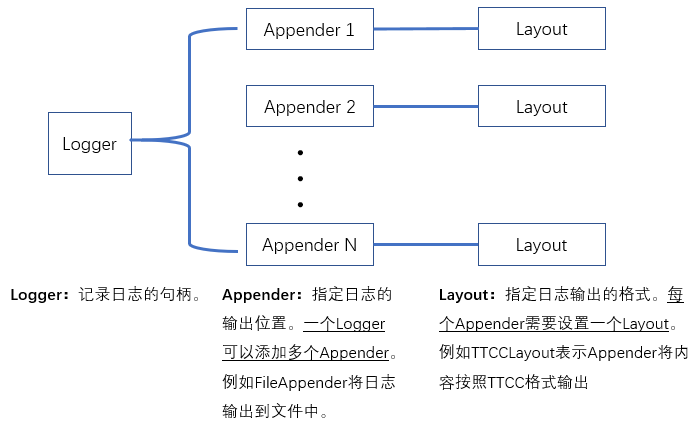


# 核心组件

Logging日志系统是[Apache Log4j 2](http://logging.apache.org/log4j/2.x/)的C++完整的继承实现，提供接口和使用逻辑与Log4j基本保持一致。

更多详细内容可以查阅[Log4j 2 API Documentation](http://logging.apache.org/log4j/2.x/log4j-api/index.html)文档。核心组件如下：

|  |  |
| --- | --- |
| 类 | 功能 |
| Logger | 记录器。保存并跟踪对象日志信息变更的实体，当你需要对一个对象进行记录时，就需要生成一个logger。 |
| Appender | 挂接器。与布局器和过滤器紧密配合，将特定格式的消息过滤后输出到所挂接的设备终端（如：控制台，文件、远程服务器等）。一个Logger可添加多个Appender，从而向多个位置输出日志。 |
| Layout | 布局器。用于指定日志输出的格式。每个Appender需要设置一个Layout。 |
| Filter | 过滤器。过滤输出消息，比如FATAL、ERROR、WARN、INFO、DEBUG、TRACE等的输出控制 |
| Hierarchy | 分类器。层次化的树型结构，用于对被记录信息的分类，层次中每一个节点维护一个logger的所有信息。 |



Logging系统的使用步骤：

第1步：用Initializer类初始化logging系统。

第2步：实例化一个封装了输出介质的Appender对象。

第3步：实例化一个封装了输出格式的Layout对象。

第4步：将Layout对象绑定(attach)到Appender对象；如省略此步骤，简单布局器SimpleLayout对象会绑定到Appender。

第5步：通过Logger::getInstance()获得Logger实例，并设置其日志输出等级阈值，如省略此步骤，默认等级为DEBUG（只有TRACE不被输出）。

第6步：将Appender对象绑定(attach)到Logger对象。

第7步：使用宏输出日志，日志级别从高到低排序分别为：FATAL、ERROR、WARN、INFO、DEBUG、TRACE。FATAL宏的名为LOGGING\_FATAL()，ERROR的宏的名为LOGGING\_ERROR()，以此类推。

注：一个Logger实例被配置后，将一直存在于程序中，在程序的任何地方都可通过实例名称获取到这个Logger，不用重新配置。

简单的控制台输出日志程序：

#include <logging/logging.h>

using namespace logging;

int main()

{

logging::Initializer initializer;

SharedAppenderPtr appender(new ConsoleAppender());

auto logger = Logger::getInstance(LOGGING\_TEXT("logger"));

logger.addAppender(SharedAppenderPtr(appender.get()));

for (size\_t i = 0; i < 10; i++)

{

LOGGING\_TRACE(logger, "Entering loop #" << i << " This is the trace log message...");

LOGGING\_DEBUG(logger, "Entering loop #" << i << " This is the debug log message...");

LOGGING\_INFO(logger, "Entering loop #" << i << " This is the info log message...");

LOGGING\_WARN(logger, "Entering loop #" << i << " This is the warn log message...");

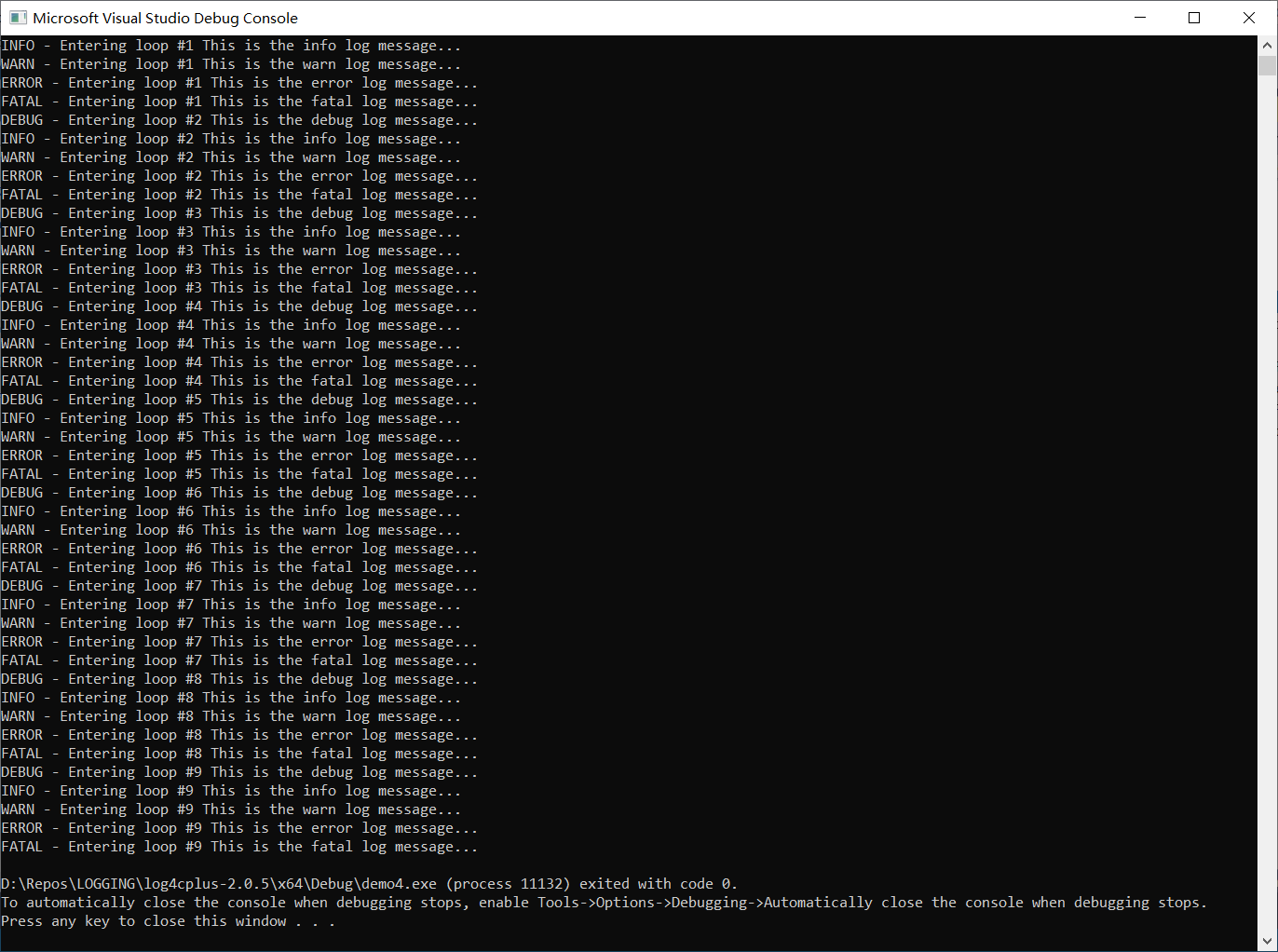
LOGGING\_ERROR(logger, "Entering loop #" << i << " This is the error log message...");

LOGGING\_FATAL(logger, "Entering loop #" << i << " This is the fatal log message...");

}

}

控制台显示结果如下：



展开后的控制台日志程序：

#include <logging/logging.h>

using namespace logging;

int main()

{

logging::Initializer initializer;

SharedAppenderPtr appender(new ConsoleAppender());

appender->setName(LOGGING\_TEXT("appender"));

Layout \* layout = new TTCCLayout;

appender->setLayout(std::unique\_ptr<Layout>(layout));

auto logger = Logger::getInstance(LOGGING\_TEXT("logger"));

logger.setLogLevel(logging::INFO\_LOG\_LEVEL);

logger.addAppender(SharedAppenderPtr(appender.get()));

for (size\_t i = 0; i < 10; i++)

{

LOGGING\_TRACE(logger, "Entering loop #" << i << " This is the trace log message...");

LOGGING\_DEBUG(logger, "Entering loop #" << i << " This is the debug log message...");

LOGGING\_INFO(logger, "Entering loop #" << i << " This is the info log message...");

LOGGING\_WARN(logger, "Entering loop #" << i << " This is the warn log message...");

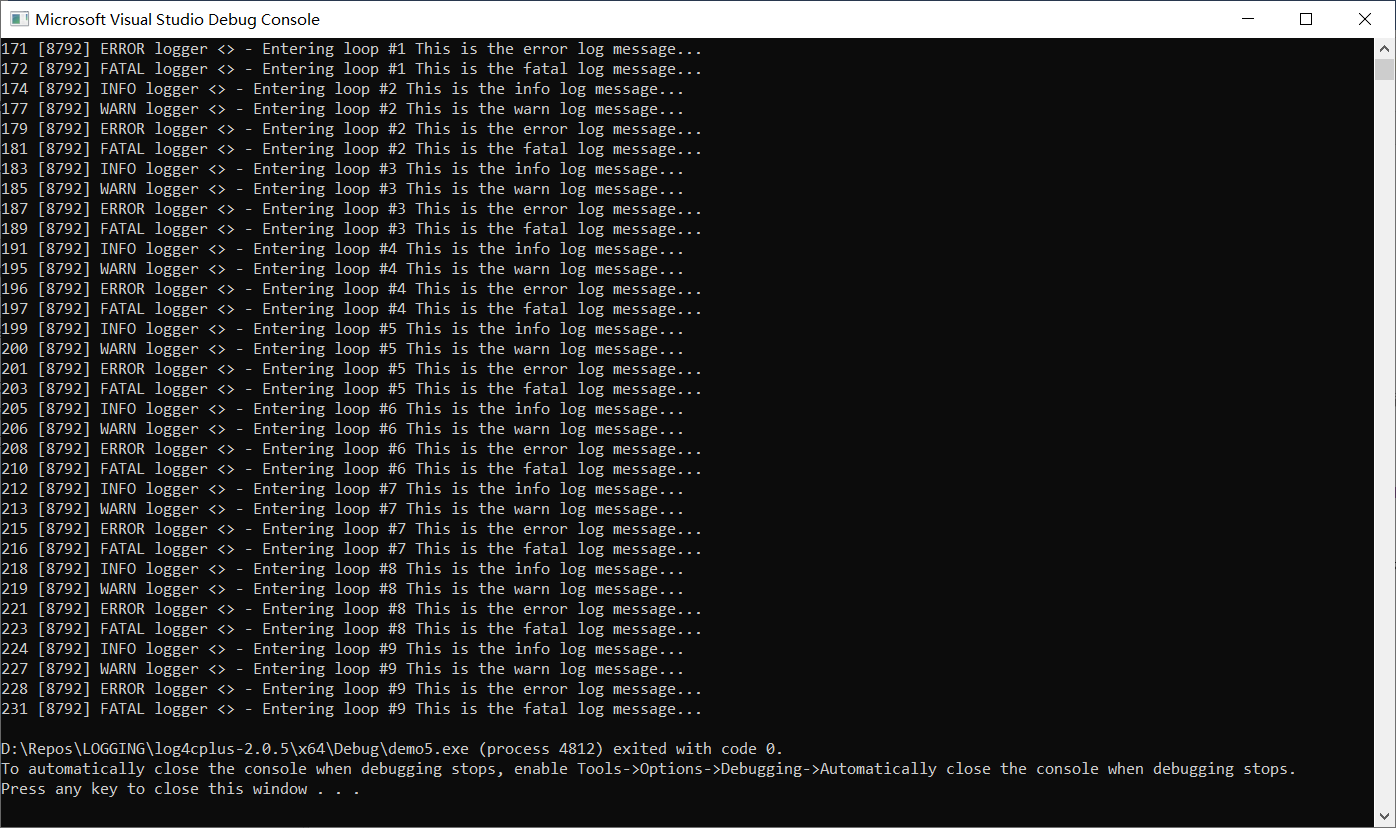
LOGGING\_ERROR(logger, "Entering loop #" << i << " This is the error log message...");

LOGGING\_FATAL(logger, "Entering loop #" << i << " This is the fatal log message...");

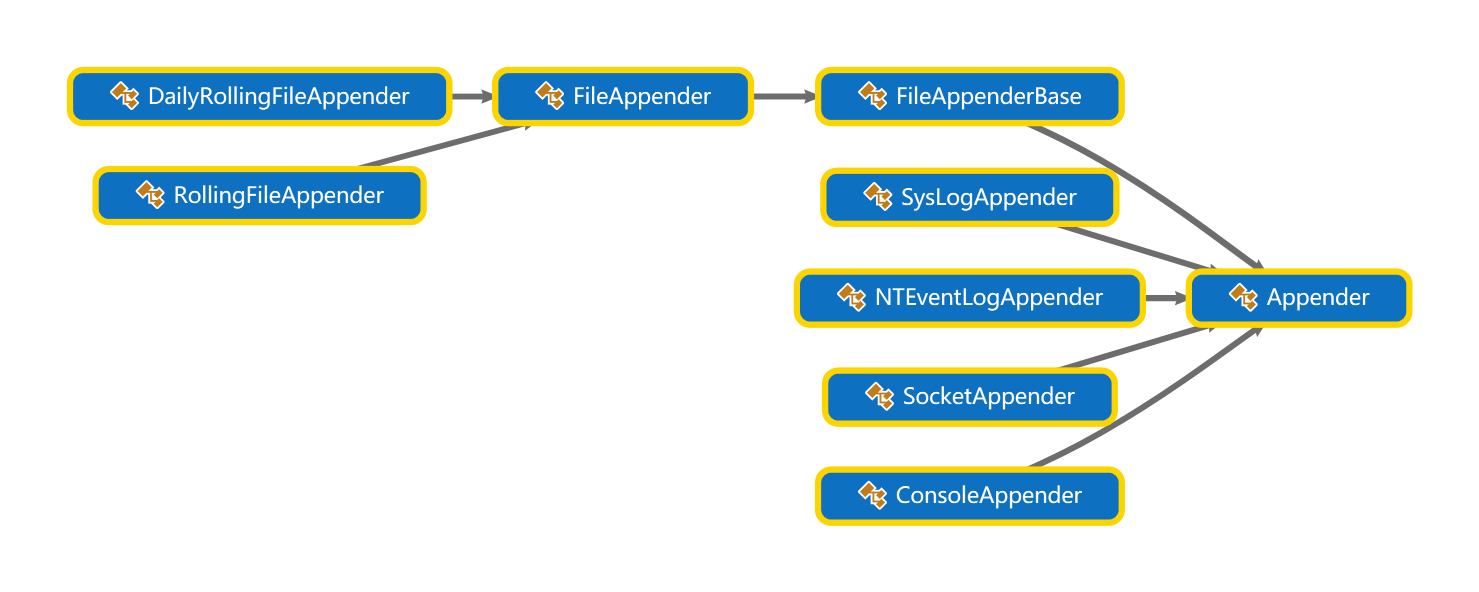
}

}

控制台显示结果如下：



## 挂接器

Logging系统支持的Appender的类型，关于文件日志主要有两种：RollingFileAppender和DailyRollingFileAppender

RollingFileAppender类可以根据预先设定的大小来决定是否转储，当超过该大小，后续log信息会另存到新文件中，除了定义每个记录文件的大小之外，还要确定在RollingFileAppender类对象构造时最多需要多少个这样的记录文件(maxBackupIndex+1)，当存储的文件数目超过maxBackupIndex+1时，会删除最早生成的文件，保证整个文件数目等于maxBackupIndex+1。

RollingFileAppender(const logging::tstring& filename,

long maxFileSize = 10 \* 1024 \* 1024, // Size = 10 MB

int maxBackupIndex = 1, // Backup = 1

bool immediateFlush = true,

bool createDirs = false);

需要指出的是，RollingFileAppender中隐含定义了文件的最小尺寸是200K，只有大于200K的设置才生效，<= 200k的设置都会被认为是200K。

DailyRollingFileAppender类可以根据预先设定的频度来决定是否转储，当超过该频度，后续log信息会另存到新文件中，这里的频度包括：MONTHLY（每月）、WEEKLY（每周）、DAILY（每日）、TWICE\_DAILY（每两天）、HOURLY（每时）、MINUTELY（每分）。

DailyRollingFileAppender(const logging::tstring& filename,

DailyRollingFileSchedule schedule = DAILY,

bool immediateFlush = true,

int maxBackupIndex = 10,

bool createDirs = false,

bool rollOnClose = true,

const logging::tstring& datePattern = logging::tstring());

需要指出的是这里的"频度"并不是你写入文件的速度，其实是否转储的标准并不依赖你写入文件的速度，而是依赖于写入的那一时刻是否满足了频度条件，即是否超过了以分钟、小时、周、月为单位的时间刻度，如果超过了就另存。

以上文件挂接器参数中，immediateFlush为缓冲刷新标志 ，默认是true，建议设置为true，以保证程序在出现异常中断的前一刻可以保留完整的跟踪信息。

#include <logging/logging.h>

using namespace logging;

int main()

{

logging::Initializer initializer;

SharedAppenderPtr appender(new DailyRollingFileAppender(LOGGING\_TEXT("../output/demo6.log")));

tstring pattern = LOGGING\_TEXT("%d{%m/%d/%y %H:%M:%S,%Q} [%t] %-5p %c{2} %%%x%% - %X{key} - %m [%l]%n");

appender->setLayout(std::unique\_ptr<Layout>(new PatternLayout(pattern)));

auto logger = Logger::getInstance(LOGGING\_TEXT("logger"));

Logger::getRoot().addAppender(appender);

for (size\_t i = 0; i < 10; i++)

{

LOGGING\_TRACE(logger, "Entering loop #" << i << " This is the trace log message...");

LOGGING\_DEBUG(logger, "Entering loop #" << i << " This is the debug log message...");

LOGGING\_INFO(logger, "Entering loop #" << i << " This is the info log message...");

LOGGING\_WARN(logger, "Entering loop #" << i << " This is the warn log message...");

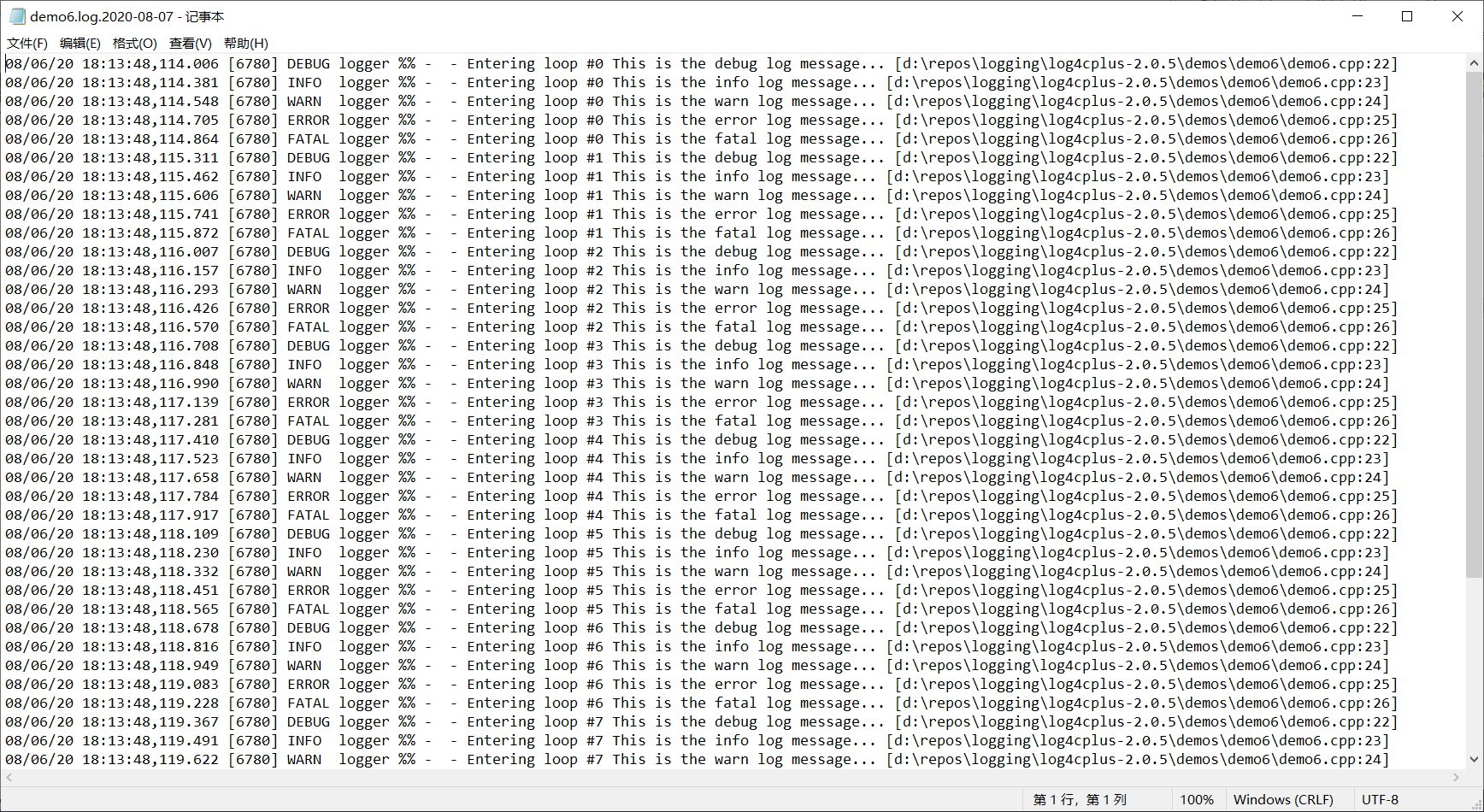
LOGGING\_ERROR(logger, "Entering loop #" << i << " This is the error log message...");

LOGGING\_FATAL(logger, "Entering loop #" << i << " This is the fatal log message...");

}

}

日志文件信息如下：

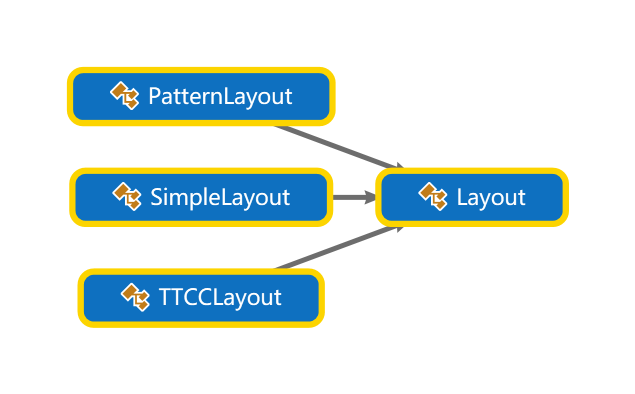


关于更多的挂接器应用（如AsyncAppender、CallbackAppender、SocketAppender、SysLogAppender等）可以查阅[Log4j 2 API Documentation](http://logging.apache.org/log4j/2.x/log4j-api/index.html)文档。

## 布局器

Logging系统支持的Layout的类型 3种，分别为PatternLayout、SimpleLayout和TTCCLayout。其中的PatternLayout可以让用户自定义输出格式。

SimpleLayout:一种简单格式的布局器，在输出的原始信息之前加上LogLevel和一个" - "，如果初始化时没有将布局器附加到挂接器，则默认使用SimpleLayout。



PatternLayout是一种有词法分析功能的模式布局器，能够对预定义的标识符进行解析，转换成特定格式输出。支持的转换标识符主要包括：

（1）"%%"，转义为%, 即，std::string pattern = "%%" 时输出"%"。

（2）"%c"，输出logger名称，比如std::string pattern ="%c" 时输出: "test\_logger.subtest"， 也可以控制logger名称的显示层次，比如"%c{1}"时输出"test\_logger"，其中数字表示层次。

（3）"%D"，显示本地时间，当std::string pattern ="%D" 时输出:"2004-10-16 18:55:45"，%d显示标准时间，所以当std::string pattern ="%d" 时输出"2004-10-16 10:55:45" （因为北京时间位于东8区，差8个小时）。

可以通过%d{...}定义更详细的显示格式，比如%d{%H:%M:%s}表示要显示小时:分钟：秒。大括号中可显示的预定义标识符如下：

%a -- 表示礼拜几，英文缩写形式，比如"Fri"

%A -- 表示礼拜几，比如"Friday"

%b -- 表示几月份，英文缩写形式，比如"Oct"

%B -- 表示几月份，"October"

%c -- 标准的日期＋时间格式，如 "Sat Oct 16 18:56:19 2004"

%d -- 表示今天是这个月的几号(1-31)"16"

%H -- 表示当前时刻是几时(0-23)，如 "18"

%I -- 表示当前时刻是几时(1-12)，如 "6"

%j -- 表示今天是哪一天(1-366)，如 "290"

%m -- 表示本月是哪一月(1-12)，如 "10"

%M -- 表示当前时刻是哪一分钟(0-59)，如 "59"

%p -- 表示现在是上午还是下午， AM or PM

%q -- 表示当前时刻中毫秒部分(0-999)，如 "237"

%Q -- 表示当前时刻中带小数的毫秒部分(0-999.999)，如 "430.732"

%S -- 表示当前时刻的多少秒(0-59)，如 "32"

%U -- 表示本周是今年的第几个礼拜，以周日为第一天开始计算(0-53)，如 "41"

%w -- 表示礼拜几，(0-6, 礼拜天为0)，如 "6"

%W -- 表示本周是今年的第几个礼拜，以周一为第一天开始计算(0-53)，如 "41"

%x -- 标准的日期格式，如 "10/16/04"

%X -- 标准的时间格式，如 "19:02:34"

%y -- 两位数的年份(0-99)，如 "04"

%Y -- 四位数的年份，如 "2004"

%Z -- 时区名，比如 "GMT"

（4）"%F"，输出当前记录器所在的文件名称，比如std::string pattern ="%F" 时输出: "main.cpp"。

（5）"%L"，输出当前记录器所在的文件行号，比如std::string pattern ="%L" 时输出: "51"

（6）"%l"，输出当前记录器所在的文件名称和行号，比如std::string pattern ="%l" 时输出"main.cpp:51"。

（7）"%m"，输出原始信息，比如std::string pattern ="%m" 时输出: "teststr"，这种实现机制可以确保原始信息被嵌入到带格式的信息中。

（8）"%n"，换行符，没什么好解释的。

（9）"%p"，输出LogLevel，比如std::string pattern ="%p" 时输出: "DEBUG"。

（10）"%t"，输出记录器所在的线程ID，比如std::string pattern ="%t" 时输出: "1075298944"。

（11）"%x"，嵌套诊断上下文NDC (nested diagnostic context) 输出，从堆栈中弹出上下文信息，NDC可以用对不同源的log信息（同时地）交叉输出进行区分，关于NDC方面的详细介绍会在下文中提到。

（12）格式对齐，比如std::string pattern ="%-10m"时表示左对齐，宽度是10，此时会输出"teststr "，当然其它的控制字符也可以相同的方式来使用，比如"%-12d"，"%-5p"等等。

TTCCLayout是在PatternLayout基础上发展的一种缺省的带格式输出的布局器，其格式由时间，线程ID，Logger和NDC 组成（consists of Time, Thread, Logger and nested diagnostic Context information, hence the name），TTCCLayout在构造时，有机会选择显示本地时间或GMT时间，缺省是按照本地时间显示：TTCCLayout::TTCCLayout(bool use\_gmtime = false)。

## 嵌入诊断上下文NDC

嵌入诊断上下文（Nested Diagnostic Context），即NDC。对logging系统而言，当输入源可能不止一个，而只有一个输出时，往往需要分辩所要输出消息的来源，比如服务器处理来自不同客户端的消息时就需要作此判断，NDC可以为交错显示的信息打上一个标记(stamp)，使得辨认工作看起来比较容易些。这个标记是线程特有的，利用了线程局部存储机制，称为线程私有数据（Thread-Specific Data，或TSD）

#include <logging/logging.h>

using namespace logging;

int main()

{

logging::Initializer initializer;

SharedAppenderPtr appender(new ConsoleAppender());

appender->setLayout(std::unique\_ptr<Layout>(new TTCCLayout));

auto logger = Logger::getInstance(LOGGING\_TEXT("logger"));

logger.addAppender(SharedAppenderPtr(appender.get()));

NDC& \_ndc = logging::getNDC();

\_ndc.push(LOGGING\_TEXT("\_ndc"));

LOGGING\_DEBUG(logger, "this is a NDC test");

\_ndc.pop();

LOGGING\_DEBUG(logger, "There should be no NDC...");

\_ndc.remove();

// 另外一种更简单的使用方法是在线程中直接用NDCContextCreator，不必显式地调用push / pop了.

// 而且当出现异常时，能够确保push与pop的调用是匹配的。

NDCContextCreator \_first\_ndc(LOGGING\_TEXT("\_first\_ndc"));

LOGGING\_DEBUG(logger, "this is a NDC test");

\_ndc.pop();

LOGGING\_DEBUG(logger, "There should be no NDC...");

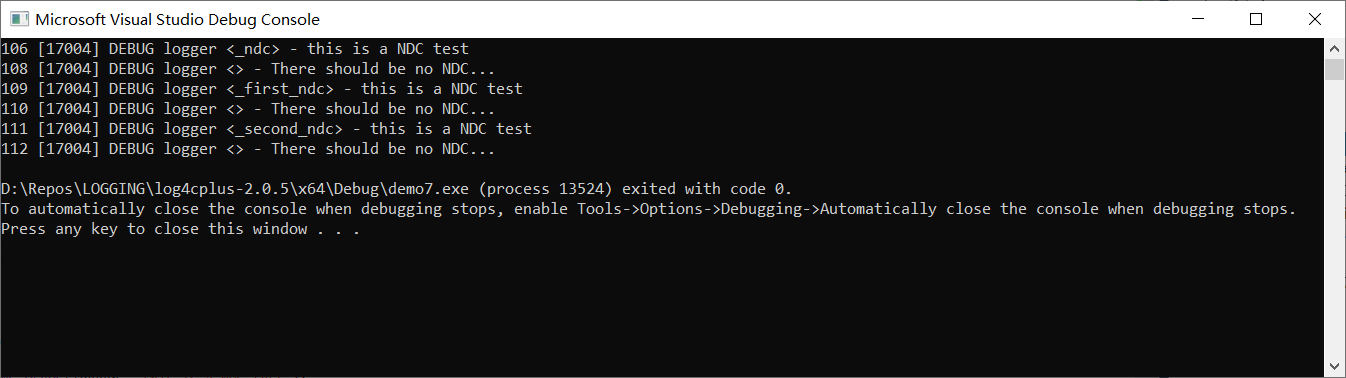
NDCContextCreator \_second\_ndc(LOGGING\_TEXT("\_second\_ndc"));

LOGGING\_DEBUG(logger, "this is a NDC test");

\_ndc.remove();

LOGGING\_DEBUG(logger, "There should be no NDC...");

}



## 脚本配置

除了通过程序实现对log环境的配置之外，还可以通过PropertyConfigurator实现了基于脚本配置的功能。通过脚本可以完成对logger、appender和layout的配置，因此可以解决怎样输出，输出到哪里的问题。

基本配置

基本配置语法主要针对包括rootLogger和non-root logger。

* 根Logger的配置

语法：logging.rootLogger=[LogLevel], appenderName, appenderName, ...

* 非根Logger的配置

语法：logging.logger.logger\_name=[LogLevel|INHERITED], appenderName, appenderName, ...

说明：INHERITED表示继承父Logger的日志级别。

高级配置

**Appender配置**

语法：

logging.appender.appenderName=fully.qualified.name.of.appender.class

fully.qualified.name.of.appeneder.class可用值：

ConsoleAppender控制台输出

FileAppender一般文件输出

RollingFileAppender文件大小输出

DailyRollingFileAppender频度输出

SocketAppender网络端口输出

文件通用选项：

ImmediateFlush是否立即刷新（默认为true）logging.appender.ALL\_MSGS.ImmediateFlush=true

File使用的文件名logging.appender.ALL\_MSGS.File=all\_msgs.log

Append是否追加到之前的文件logging.appender.ALL\_MSGS.Append=true

ReopenDelay先将日志缓存起来，等指定时间之后再往文件中插入，减少文件的保存次数logging.appender.ALL\_MSGS.ReopenDelay=10【单位为秒】

UseLockFile是否使用加锁的方式去写文件，默认是false，logging.appender.ALL\_MSGS.UseLockFile=true

LockFile使用的加锁文件名，logging.appender.ALL\_MSGS.LockFile=fuck\_are\_you.lock[文件名没有具体要求]

Locale使用的字符集logging.appender.ALL\_MSGS.Locale=chs 或 en

Threshold指定日志消息的输出最低层次logging.appender.ALL\_MSGS.Threshold=DEBUG

DailyRollingFileAppender相关配置：

Schedule文件保存频率 可选值：MONTHLY, WEEKLY, DAILY,TWICE\_DAILY, HOURLY, MINUTELY，logging.appender.ALL\_MSGS.Schedule=MINUTELY

MaxBackupIndex最多文件个数logging.appender.ALL\_MSGS. MaxBackupIndex=10

DatePattern指定文件名的日期格式1)'.'yyyy-MM: 每月

2)'.'yyyy-ww: 每周

3)'.'yyyy-MM-dd: 每天

4)'.'yyyy-MM-dd-a: 每天两次

5)'.'yyyy-MM-dd-HH: 每小时

6)'.'yyyy-MM-dd-HH-mm: 每分钟

logging.appender.ALL\_MSGS.DatePattern='.'yyyy-ww

RollingFileAppender相关配置：

MaxFileSize最大文件大小，当小于200kb的时候，默认为200kb，单位有（MB、KB），logging.appender.ALL\_MSGS. MaxFileSize=10

MaxBackupIndex最多文件个数logging.appender.ALL\_MSGS. MaxBackupIndex=10

**Filter配置**

Appender可以附加Filter组成的链表，如果Filter链中存在过滤器Filter， logging在输出日志之前将调用链表中Filter的过滤方法decide(),根据该方法的返回值决定是否过滤该输出日志。

语法：

logging.appender.appenderName.Filters.FilterNumber=fully.qualified.name.of.Filter.class

logging.appender.appenderName.Filters.FilterNumber.FilterCondition=value.of.FilterCondition

logging.appender.appenderName.Filters.AcceptOnMatch=true|false

举例：

logging.appender.append\_1.filters.1=logging::spi::LogLevelMatchFilter

logging.appender.append\_1.filters.1.LogLevelToMatch=TRACE

logging.appender.append\_1.filters.1.AcceptOnMatch=true

目前logging提供的过滤器包括DenyAllFilter 、LogLevelMatchFilter、LogLevelRangeFilter、和StringMatchFilter。

* LogLevelMatchFilter根据特定的日志级别进行过滤。

过滤条件包括LogLevelToMatch和AcceptOnMatch（true|false）， 只有当日志的LogLevel值与LogLevelToMatch相同，且AcceptOnMatch为true时才会匹配。

* LogLevelRangeFilter根据根据日志级别的范围进行过滤。

过滤条件包括LogLevelMin、LogLevelMax和AcceptOnMatch，只有当日志的LogLevel在LogLevelMin、LogLevelMax之间同时AcceptOnMatch为true时才会匹配。

* StringMatchFilter根据日志内容是否包含特定字符串进行过滤。

过滤条件包括StringToMatch和AcceptOnMatch，只有当日志包含StringToMatch字符串 且AcceptOnMatch为true时会匹配。

* DenyAllFilter则过滤掉所有消息。

过滤条件处理机制类似于Linux中IPTABLE的Responsibility chain机制，（即先deny、再allow）不过执行顺序刚好相反，后写的条件会被先执行，比如：

logging.appender.append\_1.filters.1=logging::spi::LogLevelMatchFilter

logging.appender.append\_1.filters.1.LogLevelToMatch=TRACE

logging.appender.append\_1.filters.1.AcceptOnMatch=true

#logging.appender.append\_1.filters.2=logging::spi::DenyAllFilter

会首先执行filters.2的过滤条件，关闭所有过滤器，然后执行filters.1，仅匹配TRACE信息。

**Layout配置**

可以选择不设置、TTCCLayout、或PatternLayout，如果不设置，会输出SimpleLayout格式的日志。

设置TTCCLayout的语法：logging.appender.ALL\_MSGS.layout=logging::TTCCLayout

设置PatternLayout的语法：logging.appender.append\_1.layout=logging::PatternLayout

举例：

logging.appender.append\_1.layout.ConversionPattern=%d{%m/%d/%y %H:%M:%S,%Q} [%t] %-5p - %m%n

## 其他

以上应用已满足当前项目需要，有关过滤器、分类器、定制记录器、定制日志级别、定制LogLog等更多细节可以查阅[Log4j 2 API Documentation](http://logging.apache.org/log4j/2.x/log4j-api/index.html)文档，模块内均有对应的继承实现。