

Log4cplus 使用指南

广

目 录

1 LOG4CPLUS 简介	4
2 安装方法	4
3 主要类说明	5
4 基本使用	5
4.1 基本步骤	5
4.2 使用示例	6
4.2.1 例1-标准使用	6
4.2.2 例2-简洁使用	7
4.2.3 例3-输出日志到控制台	8
4.2.4 例4-输出日志到文件	9
4.2.5 例5-使用loglog 输出日志	10
4.3 日志输出宏	12
5 输出格式控制	13
5.1 SIMPLELAYOUT	13
5.2 PATTERNLAYOUT	14
5.2.1 转换标识符	14
5.3 TTCCLayout	16
6 输出重定向	17
6.1 重定向到控制台	17
6.2 重定向到文件	18
6.2.1 FileAppender	18
6.2.2 RollingFileAppender	18
6.2.3 DailyRollingFileAppender	19
6.3 重定向到远程服务器	21
6.3.1 客户端程序需要做的工作	21
6.3.2 服务器端程序需要做的工作	21
6.3.3 例6-重定向到远程服务器	22
6.4 嵌入诊断上下文 NDC	27
7 输出过滤	29
7.1 利用日志级别进行输出过滤	29
7.1.1 日志级别管理	29
7.1.2 利用日志级别进行输出过滤	30

7.1.3 例7-日志的优先级.....	30
7.1.4 例8-运行时利用日志级别进行输出过滤.....	33
7.2 利用脚本配置进行输出过滤	36
7.3 LogLOG 的输出过滤	36
8 脚本配置	36
8.1 基本配置	36
8.1.1 根 Logger 的配置.....	36
8.1.2 非根 Logger 的配置.....	36
8.2 高级配置	37
8.2.1 Appender 配置.....	37
8.2.2 Filter 配置.....	37
8.2.3 Layout 配置.....	38
8.3.3 例9-脚本配置.....	38
8.3 脚本配置的动态加载	41
8.3.1 例10-使用线程监控脚本的更新.....	41
9 定制 LOG4CPLUS.....	43
9.1 定制日志级别	43
9.2 定制 LogLOG	45

1 Log4cplus 简介

log4cplus 是 C++编写的开源的日志系统，前身是 java 编写的 log4j 系统，受 Apache Software License 保护，作者是 Tad E. Smith。

log4cplus 具有线程安全、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。你可以选择将日志输出到屏幕、文件、NT event log、甚至是远程服务器；通过指定策略对日志进行定期备份等等。

2 安装方法

1- 解压: `gzip -cd log4cplus-x.x.x.tar.gz | tar -xf -`

2- 进入 log4cplus 根目录: `cd log4cplus-x.x.x`

3- 产生 Makefile: `./configure --prefix=/where/to/install --enable-threads=no`

如果需要指定安装路径可使用--prefix 参数，否则将缺省安装到/usr/local 目录下。另外，如果需要单线程版本可通过参数-enable-threads=no 指定，否则默认将安装多线程版本。

对于 HP-UNIX 平台用户，由于 aCC 编译器选项兼容性问题，请另外加入参数 CXXFLAGS="-AA -w"(单线程版本)或 CXXFLAGS="-AA -mt -w"(多线程版本)。

4- 创建: `make`

对于 HP-UNIX 用户，由于 aCC 编译器不包含-Wall 选项来显示所有警告，创建时将导致无效的-W 参数错误，请修改 /log4cplus-x.x.x/src 目录下的 Makefile，将 AM_CPPFLAGS = -Wall 行的-Wall 选项删除或注释掉。

此外，某些 HP-UNIX 平台的套接字连接接受函数 accept()第三个参数要求为 int*，而在 socket-unix.cxx 源文件 153 行实现中实际传入的是 socklen_t*类型，平台并不支持，也将导致编译错误。解决方法是将源代码该行中的传入参数强制转换为 int*类型即可。

注意 AIX 和 Linux 平台目前并没有上述两处创建错误。

对于 AIX 平台用户请保证创建时使用的编译器是 xlc 而不是 g++，否则将导致 log4cplus 脚本配置功能运行时产生段异常，生成 core 文件。有鉴于此，也请保证 HP-UNIX 用户尽量使用 aCC 编译器进行创建。

5- 创建/log4cplus/tests 目录下的测试用例: `make check`

6- 安装: `make install`

安装成功后将在/usr/local 目录或指定的目录下创建 include 和 lib 两个子目录及相应

文件。其中 `include` 目录包含头文件，`lib` 目录包含最终打包生成的静态和动态库。在动态连接 `log4cplus` 库时请使用 `-llog4cplus` 选项。

3 主要类说明

类名	说明
Filter	过滤器，过滤输出消息。
Layout	布局器，控制输出消息的格式。
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屏幕，文件等等)。
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 <code>logger</code> 。
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 <code>logger</code> 的所有信息。
LogLevel	优先权，包括 <code>TRACE</code> , <code>DEBUG</code> , <code>INFO</code> , <code>WARNING</code> , <code>ERROR</code> , <code>FATAL</code> 。

4 基本使用

4.1 基本步骤

使用 `log4cplus` 有六个基本步骤：

- (1) 实例化一个封装了输出介质的 `appender` 对象；
- (2) 实例化一个封装了输出格式的 `layout` 对象；
- (3) 将 `layout` 对象绑定(`attach`)到 `appender` 对象；

如省略此步骤，简单布局器 `SimpleLayout`(参见 5.1 小节)对象会绑定到 `logger`。

- (4) 实例化一个封装了日志输出 `logger` 对象,并调用其静态函数 `getInstance()`获得实例，`log4cplus::Logger::getInstance("logger_name")`；
- (5) 将 `appender` 对象绑定(`attach`)到 `logger` 对象；
- (6) 设置 `logger` 的优先级，如省略此步骤，各种有限级的日志都将被输出。

4.2 使用示例

下面通过一些例子来了解 log4cplus 的基本使用。

4.2.1 例 1-标准使用

```
/*
 *标准使用， 严格实现步骤 1-6。
 */
#include <log4cplus/logger.h>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

int main()
{
    /* step 1: Instantiate an appender object */
    SharedObjectPtr<Appender> _append (new ConsoleAppender());
    _append->setName("append for test");

    /* step 2: Instantiate a layout object */
    std::string pattern = "%d{%m/%d/%y %H:%M:%S} - %m [%l]%n";
    std::auto_ptr<Layout> _layout(new PatternLayout(pattern));

    /* step 3: Attach the layout object to the appender */
    _append->setLayout( _layout );

    /* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance("test");

    /* step 5: Attach the appender object to the logger */
    _logger.addAppender(_append);

    /* step 6: Set a priority for the logger */
}
```

```

_logger.setLogLevel(ALL_LOG_LEVEL);

/* log activity */
LOG4CPLUS_DEBUG(_logger, "This is the FIRST log message...")
sleep(1);
LOG4CPLUS_WARN(_logger, "This is the SECOND log message...")
return 0;
}

```

输出结果:

10/14/04 09:06:24 - This is the FIRST log message... [main.cpp:31]

10/14/04 09:06:25 - This is the SECOND log message... [main.cpp:33]

4.2.2 例 2-简洁使用

```

/*
 *简洁使用，仅实现步骤 1、4、5。
 */
#include <log4cplus/logger.h>
#include <log4cplus/consoleappender.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

int main()
{
    /* step 1: Instantiate an appender object */
    SharedAppenderPtr _append(new ConsoleAppender());
    _append->setName("append test");

    /* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance("test");

    /* step 5: Attach the appender object to the logger */
    _logger.addAppender(_append);
}

```

```
/* log activity */
LOG4CPLUS_DEBUG(_logger, "This is the FIRST log message...")
sleep(1);
LOG4CPLUS_WARN(_logger, "This is the SECOND log message...")

return 0;
}
```

输出结果:

DEBUG - This is the FIRST log message...

WARN - This is the SECOND log message...

4.2.3 例 3-输出日志到控制台

```
/*
 * iostream 模式， appender 输出到控制台。
 */
#include <log4cplus/logger.h>
#include <log4cplus/consoleappender.h>
#include <iomanip>

using namespace log4cplus;

int main()
{
    /* step 1: Instantiate an appender object */
    SharedAppenderPtr _append(new ConsoleAppender());
    _append->setName("append test");

    /* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance("test");

    /* step 5: Attach the appender object to the logger */
    _logger.addAppender(_append);

    /* log activity */
```



```

LOG4CPLUS_TRACE(_logger, "This is" << " just a t" << "est." << std::endl)
LOG4CPLUS_DEBUG(_logger, "This is a bool: " << true)
LOG4CPLUS_INFO(_logger, "This is a char: " << 'x')
LOG4CPLUS_WARN(_logger, "This is a int: " << 1000)
LOG4CPLUS_ERROR(_logger, "This is a long(hex): " << std::hex << 1000000000)
LOG4CPLUS_FATAL(_logger, "This is a double: " << std::setprecision(15) <<
1.2345234234)

return 0;
}

```

输出结果:

DEBUG - This is a bool: 1

INFO - This is a char: x

WARN - This is a int: 1000

ERROR - This is a long(hex): 5f5e100

FATAL - This is a double: 1.2345234234

4.2.4 例 4-输出日志到文件

```

/*
 *文件模式， appender 输出到文件。
 */
#include <log4cplus/logger.h>
#include <log4cplus/fileappender.h>

using namespace log4cplus;

int main()
{
    /* step 1: Instantiate an appender object */
    SharedAppenderPtr _append(new FileAppender("Test.log"));
    _append->setName("file log test");

    /* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance("test.subtestof_filelog");
}

```

```

/* step 5: Attach the appender object to the logger */
_logger.addAppender(_append);

/* log activity */
int i;
for( i = 0; i < 5; ++i )
{
    LOG4CPLUS_DEBUG(_logger, "Entering loop #" << i << "End line #")
}

return 0;
}

```

输出结果 (Test.log 文件):

```

DEBUG - Entering loop #0End line #
DEBUG - Entering loop #1End line #
DEBUG - Entering loop #2End line #
DEBUG - Entering loop #3End line #
DEBUG - Entering loop #4End line #

```

4.2.5 例 5-使用 loglog 输出日志

LogLog 类实现了 debug, warn, error 函数用于 logcplus 运行时显示 log4cplus 自身的调试、警告或错误信息, 是对标准输出的简单封装, 它也可以用来进行简单的日志输出。LogLog 同时提供了两个方法来进一步控制所输出的信息, 其中 setInternalDebugging() 方法用来控制是否屏蔽输出信息中的调试信息, 当输入参数为 false 则屏蔽, 缺省设置为 false。setQuietMode() 方法用来控制是否屏蔽所有输出信息, 当输入参数为 true 则屏蔽, 缺省设置为 false。

```

/*
    通过 loglog 来控制输出调试、警告或错误信息, appender 输出到屏幕。
*/
#include <iostream>
#include <log4cplus/helpers/loglog.h>

```

```
using namespace log4cplus::helpers;

void printMsgs(void)
{
    std::cout << "Entering printMsgs()..." << std::endl;
    LogLog::getLogLog()->debug("This is a Debug statement...");
    LogLog::getLogLog()->warn("This is a Warning...");
    LogLog::getLogLog()->error("This is a Error...");
    std::cout << "Exiting printMsgs()..." << std::endl << std::endl;
}

int main()
{
    printMsgs();

    std::cout << "Turning on debug..." << std::endl;
    LogLog::getLogLog()->setInternalDebugging(true);
    printMsgs();

    std::cout << "Turning on quiet mode..." << std::endl;
    LogLog::getLogLog()->setQuietMode(true);
    printMsgs();

    return 0;
}
```

输出结果:

```
Entering printMsgs()...
log4cplus:WARN This is a Warning...
log4cplus:ERROR This is a Error...
Exiting printMsgs()...
Turning on debug...
Entering printMsgs()...
log4cplus: This is a Debug statement...
log4cplus:WARN This is a Warning...
```

log4cplus:ERROR This is a Error...

Exiting printMsgs()...

Turning on quiet mode...

Entering printMsgs()...

Exiting printMsgs()...

注意输出信息中总是包含"log4cplus:"前缀，如果需要定制使其使用其他的前缀请参见 9.2 小节。

4.3 日志输出宏

log4cplus 在头文件 loggingmacros.h 中提供了以下的日志输出宏：

```
LOG4CPLUS_TRACE_METHOD(logger, logEvent)
```

```
LOG4CPLUS_TRACE(logger, logEvent)
```

```
LOG4CPLUS_TRACE_STR(logger, logEvent)
```

```
LOG4CPLUS_DEBUG(logger, logEvent)
```

```
LOG4CPLUS_DEBUG_STR(logger, logEvent)
```

```
LOG4CPLUS_INFO(logger, logEvent)
```

```
LOG4CPLUS_INFO_STR(logger, logEvent)
```

```
LOG4CPLUS_WARN(logger, logEvent)
```

```
LOG4CPLUS_WARN_STR(logger, logEvent)
```

```
LOG4CPLUS_ERROR(logger, logEvent)
```

```
LOG4CPLUS_ERROR_STR(logger, logEvent)
```

```
LOG4CPLUS_FATAL(logger, logEvent)
```

```
LOG4CPLUS_FATAL_STR(logger, logEvent)
```

其中 `logger` 为 `Logger` 实例名称，`logEvent` 为日志内容。由于 `log4cplus` 选用 C++ 的流机制进行日志输出，因此为了区分包含 `<<` 运算符和不包含 `<<` 运算符的日志内容，分别提供了 `LOG4CPLUS_XXXX` 和 `LOG4CPLUS_XXXX_STR` 两种日志输出宏。另外，日志输出宏 `LOG4CPLUS_TRACE_METHOD` 主要用来跟踪方法的调用轨迹。

5 输出格式控制

log4cplus 通过布局器 (Layouts) 来控制输出的格式, log4cplus 提供了三种类型的 Layouts, 分别是 SimpleLayout、PatternLayout、和 TTCCLayout。

5.1 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。

以下代码片段演示了如何使用 SimpleLayout。

```
... ..
/* step 1: Instantiate an appender object */
SharedObjectPtr _append (new ConsoleAppender());
_append->setName("append for test");

/* step 2: Instantiate a layout object */
std::auto_ptr<Layout> _layout(new log4cplus::SimpleLayout());

/* step 3: Attach the layout object to the appender */
_append->setLayout( _layout );

/* step 4: Instantiate a logger object */
Logger _logger = Logger::getInstance("test");

/* step 5: Attach the appender object to the logger */
_logger.addAppender(_append);

/* log activity */
LOG4CPLUS_DEBUG(_logger, "This is the simple formatted log message...")

... ..
```

输出结果:

DEBUG - This is the simple formatted log message...

5.2 PatternLayout

一种有词法分析功能的模式布局器，类似于 C 语言的 `printf()` 函数，能够对预定义的转换标识符（conversion specifiers）进行解析，转换成特定格式输出。

以下代码片段演示了如何使用 `PatternLayout`。

```
... ..
/* step 1: Instantiate an appender object */
SharedObjectPtr _append (new ConsoleAppender());
_append->setName("append for test");

/* step 2: Instantiate a layout object */
std::string pattern = "%d{%m/%d/%y %H:%M:%S} - %m [%l]%n";
std::auto_ptr<Layout> _layout(new PatternLayout(pattern));

/* step 3: Attach the layout object to the appender */
_append->setLayout( _layout );

/* step 4: Instantiate a logger object */
Logger _logger = Logger::getInstance("test_logger.subtest");

/* step 5: Attach the appender object to the logger */
_logger.addAppender(_append);
/* log activity */
LOG4CPLUS_DEBUG(_logger, "teststr")

... ..
```

输出结果：

```
10/16/04 18:51:25 - teststr [main.cpp:51]
```

5.2.1 转换标识符

`PatternLayout` 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", 即上述代码中 LOG4CPLUS_DEBUG 的第二个参数, 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。

5.3 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), 因而得名, 关于 NDC 请参见 6.4 小节。

以下代码片段演示了如何使用 TTCCLayout。

```
... ..
/* step 1: Instantiate an appender object */
SharedObjectPtr _append (new ConsoleAppender());
_append->setName("append for test");

/* step 2: Instantiate a layout object */
std::auto_ptr _layout(new TTCCLayout());
```



```

/* step 3: Attach the layout object to the appender */
_append->setLayout( _layout );

/* step 4: Instantiate a logger object */
Logger _logger = Logger::getInstance("test_logger");

/* step 5: Attach the appender object to the logger */
_logger.addAppender(_append);

/* log activity */
LOG4CPLUS_DEBUG(_logger, "teststr")

... ..

```

输出结果:

```
10-16-04 19:08:27,501 [1075298944] DEBUG test_logger <> - teststr
```

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)
```

如果需要构造 TTCCLayout 对象时选择 GMT 时间格式，则使用方式如下代码片断所示。

```

... ..

/* step 2: Instantiate a layout object */
std::auto_ptr_layout(new TTCCLayout(true));

... ..

```

输出结果:

```
10-16-04 11:12:47,678 [1075298944] DEBUG test_logger <> - teststr
```

6 输出重定向

6.1 重定向到控制台

log4cplus 默认将输出到控制台，提供 ConsoleAppender 用于操作。示例代码请参见 4.2.1、4.2.2 或 4.2.3 小节，这里不再赘述。

6.2 重定向到文件

log4cplus 提供了三个类用于文件操作，它们是 FileAppender 类、RollingFileAppender 类、DailyRollingFileAppender 类。

6.2.1 FileAppender

实现了基本的文件操作功能，构造函数如下：

```
FileAppender::FileAppender(const log4cplus::tstring& filename,
                           LOG4CPLUS_OPEN_MODE_TYPE mode =
                           LOG4CPLUS_FSTREAM_NAMESPACE::ios::trunc,
                           bool immediateFlush = true);
```

- filename : 文件名
- mode : 文件类型，可选择的文件类型包括 app、ate、binary、in、out、trunc，因为实际上只是对 stl 的一个简单包装，这里就不多讲了。缺省是 trunc，表示将先前文件删除。
- immediateFlush : 缓冲刷新标志，如果为 true 表示每向文件写一条记录就刷新一次缓存，否则直到 FileAppender 被关闭或文件缓存已满才更新文件，一般是要设置 true 的，比如你往文件写的过程中出现了错误（如程序非正常退出），即使文件没有正常关闭也可以保证程序终止时刻之前的所有记录都会被正常保存。

FileAppender 类的使用情况请参考 4.2.5 小节，这里不再赘述。

6.2.2 RollingFileAppender

实现可以滚动转储的文件操作功能，构造函数如下：

```
RollingFileAppender::RollingFileAppender(const log4cplus::tstring& filename,
                                           long maxFileSize,
                                           int maxBackupIndex,
                                           bool immediateFlush)
```

- filename : 文件名
- maxFileSize : 文件的最大尺寸
- maxBackupIndex : 最大记录文件数
- immediateFlush : 缓冲刷新标志

RollingFileAppender 类可以根据你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，你还要确定在

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。然后继续记录, 比如以下代码片段:

```
... ..

#define LOOP_COUNT 200000

SharedAppenderPtr _append(new RollingFileAppender("Test.log", 5*1024, 5));
_append->setName("file test");
_append->setLayout( std::auto_ptr(new TTCCLayout()) );
Logger::getRoot().addAppender(_append);

Logger root = Logger::getRoot();
Logger test = Logger::getInstance("test");
Logger subTest = Logger::getInstance("test.subtest");

for(int i=0; i    {
    NDCCContextCreator _context("loop");
    LOG4CPLUS_DEBUG(subTest, "Entering loop #" << i)
}

... ..
```

输出结果:

运行后会产生 6 个输出文件, Test.log、Test.log.1、Test.log.2、Test.log.3、Test.log.4、Test.log.5 其中 Test.log 存放着最新写入的信息, 而最后一个文件中并不包含第一个写入信息, 说明已经被不断更新了。

需要指出的是, 这里除了 Test.log 之外, 每个文件的大小都是 200K, 而不是我们想像中的 5K, 这是因为 log4cplus 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。

6.2.3 DailyRollingFileAppender

实现根据频度来决定是否转储的文件转储功能, 构造函数如下:

```
DailyRollingFileAppender::DailyRollingFileAppender(const log4cplus::tstring& filename,
```

```
bool immediateFlush,  
int maxBackupIndex)
```

- DailyRollingFileAppender 类可以根据你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（每小时）、MINUTELY（每分钟）。maxBackupIndex 的含义同上所述，比如以下代码片段：

• • • • •

20

17-03-04 和 Test.log.2004-10-17-03-05 这样的文件。

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。

6.3 重定向到远程服务器

log4cplus 提供了 SocketAppender，实现了 C/S 方式的日志记录，用于支持重定向到远程服务器。

6.3.1 客户端程序需要做的工作

- (1) 定义一个 SocketAppender 类型的挂接器

```
SharedAppenderPtr _append(new SocketAppender(host, 8888, "ServerName"));
```

- (2) 把该挂接器加入到 logger 中

```
Logger::getRoot().addAppender(_append);
```

- (3) SocketAppender 类型不需要 Layout，直接调用宏就可以将信息发往 loggerServer 了

```
LOG4CPLUS_INFO(Logger::getRoot(), "This is a test: ")
```

注意这里对宏的调用其实是调用了 SocketAppender::append()，里面有一个数据传输约定，即先发送一个后续数据的总长度，然后再发送实际的数据：

```
... ..

SocketBuffer buffer = convertToBuffer(event, serverName);
SocketBuffer msgBuffer(LOG4CPLUS_MAX_MESSAGE_SIZE);

msgBuffer.appendSize_t(buffer.getSize());
msgBuffer.appendBuffer(buffer);

... ..
```

6.3.2 服务器端程序需要做的工作

- (1) 定义一个 ServerSocket

```
ServerSocket serverSocket(port);
```

- (2) 调用 accept 函数创建一个新的 socket 与客户端连接

```
Socket sock = serverSocket.accept();
```

- (3) 此后即可用该 sock 进行数据 read/write 了,形如(完整代码见 6.3.3 小节):

```

SocketBuffer msgSizeBuffer(sizeof(unsigned int));

if(!clientsock.read(msgSizeBuffer))
{
    return;
}

unsigned int msgSize = msgSizeBuffer.readInt();

SocketBuffer buffer(msgSize);

if(!clientsock.read(buffer))
{
    return;
}

```

(4) 为了将读到的数据正常显示出来，需要将 `SocketBuffer` 存放的内容转换成 `InternalLoggingEvent` 格式：

```
log4cplus::spi::InternalLoggingEvent event = readFromBuffer(buffer);
```

然后输出：

```
Logger logger = Logger::getInstance(event.getLoggerName());
```

```
logger.callAppenders(event);
```

注意 `read/write` 是按照阻塞方式实现的，意味着对其调用直到满足了所接收或发送的个数才返回。

6.3.3 例 6-重定向到远程服务器

以下是服务器端代码。

```

#include <log4cplus/config.h>
#include <log4cplus/configurator.h>
#include <log4cplus/consoleappender.h>
#include <log4cplus/socketappender.h>
#include <log4cplus/helpers/loglog.h>
#include <log4cplus/helpers/socket.h>
#include <log4cplus/helpers/threads.h>

```

```
#include <log4cplus/spi/loggerimpl.h>
#include <log4cplus/spi/loggingevent.h>

#include <iostream>

using namespace std;
using namespace log4cplus;
using namespace log4cplus::helpers;
using namespace log4cplus::thread;

namespace loggingserver {
    class ClientThread : public AbstractThread {
    public:
        ClientThread(Socket clientsock)
            : clientsock(clientsock)
        {
            cout << "Received a client connection!!!!" << endl;
        }

        ~ClientThread()
        {
            cout << "Client connection closed." << endl;
        }

        virtual void run();

    private:
        Socket clientsock;
    };
}
```

```
int
main(int argc, char** argv)
{
    if(argc < 3) {
        cout << "Usage: port config_file" << endl;
        return 1;
    }
    int port = atoi(argv[1]);
    tstring configFile = LOG4CPLUS_C_STR_TO_TSTRING(argv[2]);

    PropertyConfigurator config(configFile);
    config.configure();

    ServerSocket serverSocket(port);
    while(1) {
        loggingserver::ClientThread *thr =
            new loggingserver::ClientThread(serverSocket.accept());
        thr->start();
    }

    return 0;
}

/////////////////////////////////////////////////////////////////
// loggingserver::ClientThread implementation
/////////////////////////////////////////////////////////////////

void
loggingserver::ClientThread::run()
{
    while(1) {
        if(!clientsock.isOpen()) {
```



```
        return;
    }
    SocketBuffer msgSizeBuffer(sizeof(unsigned int));
    if(!clientsock.read(msgSizeBuffer)) {
        return;
    }

    unsigned int msgSize = msgSizeBuffer.readInt();

    SocketBuffer buffer(msgSize);
    if(!clientsock.read(buffer)) {
        return;
    }

    spi::InternalLoggingEvent event = readFromBuffer(buffer);
    Logger logger = Logger::getInstance(event.getLoggerName());
    logger.callAppenders(event);
}
}
```

以下是客户端代码。

```
#include <log4cplus/logger.h>
#include <log4cplus/socketappender.h>
#include <log4cplus/loglevel.h>
#include <log4cplus/tstring.h>
#include <log4cplus/helpers/threads.h>
#include <iomanip>

using namespace std;
using namespace log4cplus;

int
main(int argc, char **argv)
```

```

{
    log4cplus::helpers::sleep(1);
    tstring serverName = (argc > 1 ? LOG4CPLUS_C_STR_TO_TSTRING(argv[1]) :
tstring());
//    tstring host = LOG4CPLUS_TEXT("192.168.2.10");
    tstring host = LOG4CPLUS_TEXT("127.0.0.1");
    SharedAppenderPtr append_1(new SocketAppender(host, 9998, serverName));
    append_1->setName( LOG4CPLUS_TEXT("First") );
    Logger::getRoot().addAppender(append_1);

    Logger root = Logger::getRoot();
    Logger test = Logger::getInstance( LOG4CPLUS_TEXT("socket.test") );

    LOG4CPLUS_DEBUG(root,      "This is"
                        << " a reall"
                        << "y long message." << endl
                        << "Just testing it out" << endl
                        << "What do you think?")

    test.setLogLevel(NOT_SET_LOG_LEVEL);
    LOG4CPLUS_DEBUG(test, "This is a bool: " << true)
    LOG4CPLUS_INFO(test, "This is a char: " << 'x')
    LOG4CPLUS_INFO(test, "This is a short: " << (short)-100)
    LOG4CPLUS_INFO(test, "This is a unsigned short: " << (unsigned short)100)
    log4cplus::helpers::sleep(0, 500000);
    LOG4CPLUS_INFO(test, "This is a int: " << (int)1000)
    LOG4CPLUS_INFO(test, "This is a unsigned int: " << (unsigned int)1000)
    LOG4CPLUS_INFO(test, "This is a long(hex): " << hex << (long)100000000)
    LOG4CPLUS_INFO(test, "This is a unsigned long: " << (unsigned long)100000000)
    LOG4CPLUS_WARN(test, "This is a float: " << (float)1.2345)
    LOG4CPLUS_ERROR(test, "This is a double: "
                        << setprecision(15)
                        << (double)1.2345234234)

    LOG4CPLUS_FATAL(test, "This is a long double: "

```

```

        << setprecision(15)
        << (long double)123452342342.342)

    return 0;
}

```

6.4 嵌入诊断上下文 NDC

log4cplus 中的嵌入诊断上下文 (Nested Diagnostic Context), 即 NDC。对 log 系统而言, 当输入源可能不止一个, 而只有一个输出时, 往往需要分辨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)。相关定义如下, 包括定义、初始化、获取、设置和清除操作:

linux pthread

```

#define LOG4CPLUS_THREAD_LOCAL_TYPE pthread_key_t*
#define LOG4CPLUS_THREAD_LOCAL_INIT ::log4cplus::thread::createPthreadKey()
#define LOG4CPLUS_GET_THREAD_LOCAL_VALUE( key ) pthread_getspecific(*key)
#define LOG4CPLUS_SET_THREAD_LOCAL_VALUE(key,value) \
    pthread_setspecific(*key, value)
#define LOG4CPLUS_THREAD_LOCAL_CLEANUP( key ) pthread_key_delete(*key)

```

win32

```

#define LOG4CPLUS_THREAD_LOCAL_TYPE DWORD
#define LOG4CPLUS_THREAD_LOCAL_INIT TlsAlloc()
#define LOG4CPLUS_GET_THREAD_LOCAL_VALUE( key ) TlsGetValue(key)
#define LOG4CPLUS_SET_THREAD_LOCAL_VALUE( key, value ) \
    TlsSetValue(key, static_cast(value))
#define LOG4CPLUS_THREAD_LOCAL_CLEANUP( key ) TlsFree(key)

```

使用起来比较简单, 在某个线程中:

```

NDC& ndc = log4cplus::getNDC();
ndc.push("ur ndc string");
LOG4CPLUS_DEBUG(logger, "this is a NDC test");

```

```

... ..

ndc.pop();

... ..

LOG4CPLUS_DEBUG(logger, "There should be no NDC...");
ndc.remove();

```

输出结果(当设定输出格式为 TTCCLayout 时):

10-21-04 21:32:58, [3392] DEBUG test - this is a NDC test

10-21-04 21:32:58, [3392] DEBUG test <> - There should be no NDC...

也可以在自定义的输出格式中使用 NDC(用%x) , 比如:

```

... ..

std::string pattern = "NDC:[%x] - %m %n";
std::auto_ptr_layout(new PatternLayout(pattern));

... ..

LOG4CPLUS_DEBUG(_logger, "This is the FIRST log message...")
NDC& ndc = log4cplus::getNDC();
ndc.push("ur ndc string");
LOG4CPLUS_WARN(_logger, "This is the SECOND log message...")
ndc.pop();
ndc.remove();

... ..

```

输出结果:

NDC:[] - This is the FIRST log message...

NDC:[ur ndc string] - This is the SECOND log message...

另外一种更简单的使用方法是在线程中直接用 NDCContextCreator:

```

NDCContextCreator _first_ndc("ur ndc string");

```

```
LOG4CPLUS_DEBUG(logger, "this is a NDC test")
```

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。

7 输出过滤

7.1 利用日志级别进行输出过滤

7.1.1 日志级别管理

log4cplus 将输出的 log 信息按照 LogLevel（从低到高）分为：

级别	说明
NOT_SET_LOG_LEVEL (-1)	接受缺省的 LogLevel，如果有父 logger 则继承它的 LogLevel
ALL_LOG_LEVEL (0)	开放所有 log 信息输出
TRACE_LOG_LEVEL (0)	开放 trace 信息输出（即 ALL_LOG_LEVEL）
DEBUG_LOG_LEVEL(10000)	开放 debug 信息输出
INFO_LOG_LEVEL (20000)	开放 info 信息输出
WARN_LOG_LEVEL (30000)	开放 warning 信息输出
ERROR_LOG_LEVEL(40000)	开放 error 信息输出
FATAL_LOG_LEVEL (50000)	开放 fatal 信息输出
OFF_LOG_LEVEL (60000)	关闭所有 log 信息输出

在 log4cplus 中，所有 logger 都通过一个层次化的结构（其实内部是 hash 表）来组织的，有一个 Root 级别的 logger，可以通过以下方法获取：

```
Logger root = Logger::getRoot();
```

用户定义的 logger 都有一个名字与之对应，比如：

```
Logger test = Logger::getInstance("test");
```

可以定义该 logger 的子 logger：

```
Logger subTest = Logger::getInstance("test.subtest");
```

注意 Root 级别的 logger 只有通过 getRoot 方法获取，Logger::getInstance("root") 获得的是它的子对象而已。有了这些具有父子关系的 logger 之后可分别设置其 LogLevel，比如：

```
root.setLogLevel( ... );
```

```
Test.setLogLevel( ... );  
subTest.setLogLevel( ... );
```

各个 logger 可以通过 setLogLevel 设置自己的优先级，当某个 logger 的 LogLevel 设置成 NOT_SET_LOG_LEVEL 时，该 logger 会继承父 logger 的优先级，另外，如果定义了重名的多个 logger，对其中任何一个的修改都会同时改变其它 logger。

7.1.2 利用日志级别进行输出过滤

log4cplus 支持编译时候和运行时刻利用日志级别进行输出过滤。编译时刻通过如下的预定义变量进行过滤：

```
#define LOG4CPLUS_DISABLE_FATAL  
#define LOG4CPLUS_DISABLE_WARN  
#define LOG4CPLUS_DISABLE_ERROR  
#define LOG4CPLUS_DISABLE_INFO  
#define LOG4CPLUS_DISABLE_DEBUG  
#define LOG4CPLUS_DISABLE_TRACE
```

运行时刻的过滤则通过使用 Logger 的 setLogLevel 设置日志级别进行过滤。

7.1.3 例 7-日志的优先级

```
#include "log4cplus/logger.h"  
#include "log4cplus/consoleappender.h"  
#include "log4cplus/loglevel.h"  
#include <iostream>  
  
using namespace std;  
using namespace log4cplus;  
  
int main()  
{  
    SharedAppenderPtr _append(new ConsoleAppender());  
    _append->setName("test");  
    Logger::getRoot().addAppender(_append);  
    Logger root = Logger::getRoot();  
  
    Logger test = Logger::getInstance("test");
```

```

Logger subTest = Logger::getInstance("test.subtest");
LogLevelManager& llm = getLogLevelManager();

cout << endl << "Before Setting, Default LogLevel" << endl;
LOG4CPLUS_FATAL(root, "root: " << llm.toString(root.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test: " << llm.toString(test.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test.subtest: " <<
llm.toString(subTest.getChainedLogLevel()))

cout << endl << "Setting test.subtest to WARN" << endl;
subTest.setLogLevel(WARN_LOG_LEVEL);
LOG4CPLUS_FATAL(root, "root: " << llm.toString(root.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test: " << llm.toString(test.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test.subtest: " <<
llm.toString(subTest.getChainedLogLevel()))

cout << endl << "Setting test to TRACE" << endl;
test.setLogLevel(TRACE_LOG_LEVEL);
LOG4CPLUS_FATAL(root, "root: " << llm.toString(root.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test: " << llm.toString(test.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test.subtest: " <<
llm.toString(subTest.getChainedLogLevel()))

cout << endl << "Setting test.subtest to NO_LEVEL" << endl;
subTest.setLogLevel(NOT_SET_LOG_LEVEL);
LOG4CPLUS_FATAL(root, "root: " << llm.toString(root.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test: " << llm.toString(test.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test.subtest: " <<
llm.toString(subTest.getChainedLogLevel()) << '\n')

cout << "create a logger test_bak, named \"test_\", too. " << endl;
Logger test_bak = Logger::getInstance("test");
cout << "Setting test to INFO, so test_bak also be set to INFO" << endl;

```

```
test.setLogLevel(INFO_LOG_LEVEL);
LOG4CPLUS_FATAL(root, "test: " << llm.toString(test.getChainedLogLevel()))
LOG4CPLUS_FATAL(root, "test_bak: " <<
llm.toString(test_bak.getChainedLogLevel()))

return 0;
}
```

输出结果:

Before Setting, Default LogLevel

FATAL - root: DEBUG

FATAL - test: DEBUG

FATAL - test.subtest: DEBUG

Setting test.subtest to WARN

FATAL - root: DEBUG

FATAL - test: DEBUG

FATAL - test.subtest: WARN

Setting test to TRACE

FATAL - root: DEBUG

FATAL - test: TRACE

FATAL - test.subtest: WARN

Setting test.subtest to NO_LEVEL

FATAL - root: DEBUG

FATAL - test: TRACE

FATAL - test.subtest: TRACE

create a logger test_bak, named "test_", too.

Setting test to INFO, so test_bak also be set to INFO

FATAL - test: INFO

FATAL - test_bak: INFO

7.1.4 例 8-运行时利用日志级别进行输出过滤

```
#include "log4cplus/logger.h"
#include "log4cplus/consoleappender.h"
#include "log4cplus/loglevel.h"
#include <iostream>

using namespace std;
using namespace log4cplus;

void ShowMsg(void)
{
    LOG4CPLUS_TRACE(Logger::getRoot(),"info")
    LOG4CPLUS_DEBUG(Logger::getRoot(),"info")
    LOG4CPLUS_INFO(Logger::getRoot(),"info")
    LOG4CPLUS_WARN(Logger::getRoot(),"info")
    LOG4CPLUS_ERROR(Logger::getRoot(),"info")
    LOG4CPLUS_FATAL(Logger::getRoot(),"info")
}

int main()
{
    SharedAppenderPtr _append(new ConsoleAppender());
    _append->setName("test");
    _append->setLayout(std::auto_ptr(new TTCCLayout()));
    Logger root = Logger::getRoot();
    root.addAppender(_append);

    cout << endl << "all-log allowed" << endl;
    root.setLogLevel(ALL_LOG_LEVEL);
    ShowMsg();

    cout << endl << "trace-log and above allowed" << endl;
    root.setLogLevel	TRACE_LOG_LEVEL);
```

```
ShowMsg();

cout << endl << "debug-log and above allowed" << endl;
root.setLogLevel(DEBUG_LOG_LEVEL);
ShowMsg();

cout << endl << "info-log and above allowed" << endl;
root.setLogLevel(INFO_LOG_LEVEL);
ShowMsg();

cout << endl << "warn-log and above allowed" << endl;
root.setLogLevel(WARN_LOG_LEVEL);
ShowMsg();

cout << endl << "error-log and above allowed" << endl;
root.setLogLevel(ERROR_LOG_LEVEL);
ShowMsg();

cout << endl << "fatal-log and above allowed" << endl;
root.setLogLevel(FATAL_LOG_LEVEL);
ShowMsg();

cout << endl << "log disabled" << endl;
root.setLogLevel(OFF_LOG_LEVEL);
ShowMsg();

return 0;
}
```

输出结果:

all-log allowed

10-17-04 10:11:40,587 [1075298944] TRACE root <> - info

10-17-04 10:11:40,590 [1075298944] DEBUG root <> - info

10-17-04 10:11:40,591 [1075298944] INFO root <> - info

10-17-04 10:11:40,591 [1075298944] WARN root <> - info
10-17-04 10:11:40,592 [1075298944] ERROR root <> - info
10-17-04 10:11:40,592 [1075298944] FATAL root <> - info

trace-log and above allowed

10-17-04 10:11:40,593 [1075298944] TRACE root <> - info
10-17-04 10:11:40,593 [1075298944] DEBUG root <> - info
10-17-04 10:11:40,594 [1075298944] INFO root <> - info
10-17-04 10:11:40,594 [1075298944] WARN root <> - info
10-17-04 10:11:40,594 [1075298944] ERROR root <> - info
10-17-04 10:11:40,594 [1075298944] FATAL root <> - info

debug-log and above allowed

10-17-04 10:11:40,595 [1075298944] DEBUG root <> - info
10-17-04 10:11:40,595 [1075298944] INFO root <> - info
10-17-04 10:11:40,596 [1075298944] WARN root <> - info
10-17-04 10:11:40,596 [1075298944] ERROR root <> - info
10-17-04 10:11:40,596 [1075298944] FATAL root <> - info

info-log and above allowed

10-17-04 10:11:40,597 [1075298944] INFO root <> - info
10-17-04 10:11:40,597 [1075298944] WARN root <> - info
10-17-04 10:11:40,597 [1075298944] ERROR root <> - info
10-17-04 10:11:40,598 [1075298944] FATAL root <> - info

warn-log and above allowed

10-17-04 10:11:40,598 [1075298944] WARN root <> - info
10-17-04 10:11:40,598 [1075298944] ERROR root <> - info
10-17-04 10:11:40,599 [1075298944] FATAL root <> - info

error-log and above allowed

10-17-04 10:11:40,599 [1075298944] ERROR root <> - info
10-17-04 10:11:40,600 [1075298944] FATAL root <> - info

fatal-log and above allowed

10-17-04 10:11:40,600 [1075298944] FATAL root <> - info

log disabled

7.2 利用脚本配置进行输出过滤

由于 log4cplus 脚本配置中可以设置日志的级别、过滤器 **Filter**，因此它也是进行输出过滤的一种很好的选择。脚本配置的使用具体参见第 8 节。

7.3 LogLog 的输出过滤

Loglog 可以使用 `setInternalDebugging()` 方法用来控制是否屏蔽输出信息中的调试信息，当输入参数为 `false` 则屏蔽，缺省设置为 `false`。另外方法 `setQuietMode()` 方法用来控制是否屏蔽所有输出信息，当输入参数为 `true` 则屏蔽，缺省设置为 `false`。具体用法请参见 4.2.5 小节。

8 脚本配置

除了通过程序实现对 log 环境的配置之外，log4cplus 通过 `PropertyConfigurator` 类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。

下面将简单介绍一下脚本的语法规则，包括基本配置语法和高级配置语法。

8.1 基本配置

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

8.1.1 根 Logger 的配置

语法：

```
log4cplus.rootLogger=[LogLevel], appenderName, appenderName, ...
```

8.1.2 非根 Logger 的配置

语法：

```
log4cplus.logger.logger_name=[LogLevel|INHERITED], appenderName,  
appenderName, ...
```

说明：INHERITED 表示继承父 `Logger` 的日志级别。

8.2 高级配置

8.2.1 Appender 配置

语法:

log4cplus.appender.appenderName=fully.qualified.name.of.appender.class

举例:

```
log4cplus.appender.append_1=log4cplus::ConsoleAppender
log4cplus.appender.append_2=log4cplus::FileAppender
log4cplus.appender.append_3=log4cplus::RollingFileAppender
log4cplus.appender.append_4=log4cplus::DailyRollingFileAppender
log4cplus.appender.append_4=log4cplus::SocketAppender
```

8.2.2 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, log4cplus 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:

log4cplus.appender.appenderName.Filter.FilterNumber=fully.qualified.name.of.Filter.class

log4cplus.appender.appenderName.Filter.FilterNumber.FilterCondition=value.of.FilterCondition

举例:

```
log4cplus.appender.append_1.filters.1=log4cplus::spi::LogLevelMatchFilter
log4cplus.appender.append_1.filters.1.LogLevelToMatch=TRACE
log4cplus.appender.append_1.filters.1.AcceptOnMatch=true
```

目前 log4plus 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:

```
log4cplus.appender.append_1.filters.1=log4cplus::spi::LogLevelMatchFilter
log4cplus.appender.append_1.filters.1.LogLevelToMatch=TRACE
log4cplus.appender.append_1.filters.1.AcceptOnMatch=true
#log4cplus.appender.append_1.filters.2=log4cplus::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。

8.2.3 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:

```
log4cplus.appender.ALL_MSGS.layout=log4cplus::TTCCLayout
```

设置 PatternLayout 的语法:

```
log4cplus.appender.append_1.layout=log4cplus::PatternLayout
```

举例:

```
log4cplus.appender.append_1.layout.ConversionPattern=%d{%m/%d/%y %H:%M:%S,%Q} [%t] %-5p - %m%n
```

8.3.3 例 9-脚本配置

脚本方式使用起来非常简单, 只要首先加载配置即可 (urconfig.properties 是自行定义的配置文件):

```
PropertyConfigurator::doConfigure("urconfig.properties");
```

下面我们通过例子体会一下 log4cplus 强大的基于脚本过滤 log 信息的功能。以下是 urconfig.properties 示例脚本配置内容。

```
/*
```

```
*      urconfig.properties
*/

log4cplus.rootLogger=TRACE, ALL_MSGS, TRACE_MSGS, DEBUG_INFO_MSGS,
FATAL_MSGS

log4cplus.appender.ALL_MSGS=log4cplus::RollingFileAppender
log4cplus.appender.ALL_MSGS.File=all_msgs.log
log4cplus.appender.ALL_MSGS.layout=log4cplus::TTCCLayout

log4cplus.appender.TRACE_MSGS=log4cplus::RollingFileAppender
log4cplus.appender.TRACE_MSGS.File=trace_msgs.log
log4cplus.appender.TRACE_MSGS.layout=log4cplus::TTCCLayout
log4cplus.appender.TRACE_MSGS.filters.1=log4cplus::spi::LogLevelMatchFilter
log4cplus.appender.TRACE_MSGS.filters.1.LogLevelToMatch=TRACE
log4cplus.appender.TRACE_MSGS.filters.1.AcceptOnMatch=true
log4cplus.appender.TRACE_MSGS.filters.2=log4cplus::spi::DenyAllFilter

log4cplus.appender.DEBUG_INFO_MSGS=log4cplus::RollingFileAppender
log4cplus.appender.DEBUG_INFO_MSGS.File=debug_info_msgs.log
log4cplus.appender.DEBUG_INFO_MSGS.layout=log4cplus::TTCCLayout
log4cplus.appender.DEBUG_INFO_MSGS.filters.1=log4cplus::spi::LogLevelRangeFilter
log4cplus.appender.DEBUG_INFO_MSGS.filters.1.LogLevelMin=DEBUG
log4cplus.appender.DEBUG_INFO_MSGS.filters.1.LogLevelMax=INFO
log4cplus.appender.DEBUG_INFO_MSGS.filters.1.AcceptOnMatch=true
log4cplus.appender.DEBUG_INFO_MSGS.filters.2=log4cplus::spi::DenyAllFilter

log4cplus.appender.FATAL_MSGS=log4cplus::RollingFileAppender
log4cplus.appender.FATAL_MSGS.File=fatal_msgs.log
log4cplus.appender.FATAL_MSGS.layout=log4cplus::TTCCLayout
log4cplus.appender.FATAL_MSGS.filters.1=log4cplus::spi::StringMatchFilter
log4cplus.appender.FATAL_MSGS.filters.1.StringToMatch=FATAL
log4cplus.appender.FATAL_MSGS.filters.1.AcceptOnMatch=true
log4cplus.appender.FATAL_MSGS.filters.2=log4cplus::spi::DenyAllFilter
```

以下是示例代码。

```
/*
 *    main.cpp
 */
#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <log4cplus/helpers/stringhelper.h>

using namespace log4cplus;

static Logger logger = Logger::getInstance("log");

void printDebug()
{
    LOG4CPLUS_TRACE_METHOD(logger, "::printDebug()");
    LOG4CPLUS_DEBUG(logger, "This is a DEBUG message");
    LOG4CPLUS_INFO(logger, "This is a INFO message");
    LOG4CPLUS_WARN(logger, "This is a WARN message");
    LOG4CPLUS_ERROR(logger, "This is a ERROR message");
    LOG4CPLUS_FATAL(logger, "This is a FATAL message");
}

int main()
{
    Logger root = Logger::getRoot();
    PropertyConfigurator::doConfigure("urconfig.properties");
    printDebug();

    return 0;
}
```

输出结果:

1. all_msgs.log


```

10-17-04 14:55:25,858 [1075298944] TRACE log <> - ENTER: ::printDebug()
10-17-04 14:55:25,871 [1075298944] DEBUG log <> - This is a DEBUG message
10-17-04 14:55:25,873 [1075298944] INFO log <> - This is a INFO message
10-17-04 14:55:25,873 [1075298944] WARN log <> - This is a WARN message
10-17-04 14:55:25,874 [1075298944] ERROR log <> - This is a ERROR message
10-17-04 14:55:25,874 [1075298944] FATAL log <> - This is a FATAL message
10-17-04 14:55:25,875 [1075298944] TRACE log <> - EXIT:  ::printDebug()

```

2. trace_msgs.log

```

10-17-04 14:55:25,858 [1075298944] TRACE log <> - ENTER: ::printDebug()
10-17-04 14:55:25,875 [1075298944] TRACE log <> - EXIT:  ::printDebug()

```

3. debug_info_msgs.log

```

10-17-04 14:55:25,871 [1075298944] DEBUG log <> - This is a DEBUG message
10-17-04 14:55:25,873 [1075298944] INFO log <> - This is a INFO message

```

4. fatal_msgs.log

```

10-17-04 14:55:25,874 [1075298944] FATAL log <> - This is a FATAL message

```

8.3 脚本配置的动态加载

多线程版本的 log4cplus 提供了实用类 `ConfigureAndWatchThread`，该类启动线程对配置脚本进行监控，一旦发现配置脚本被更新则立刻重新加载配置。

类 `ConfigureAndWatchThread` 的构造函数定义为：

```

ConfigureAndWatchThread(const log4cplus::tstring& propertyFile,
                        unsigned int millis = 60 * 1000);

```

第一个参数 `propertyFile` 为配置脚本的路径名，第二个参数为监控时两次更新检查相隔的时间，单位为毫秒 ms。

8.3.1 例 10-使用线程监控脚本的更新

```

#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <log4cplus/helpers/loglog.h>
#include <log4cplus/helpers/stringhelper.h>

```

```
using namespace std;
using namespace log4cplus;
using namespace log4cplus::helpers;

Logger log_1 = Logger::getInstance("test.log_1");
Logger log_2 = Logger::getInstance("test.log_2");
Logger log_3 = Logger::getInstance("test.log_3");

void printMsgs(Logger& logger)
{
    LOG4CPLUS_TRACE_METHOD(logger, "printMsgs()");
    LOG4CPLUS_DEBUG(logger, "printMsgs()");
    LOG4CPLUS_INFO(logger, "printMsgs()");
    LOG4CPLUS_WARN(logger, "printMsgs()");
    LOG4CPLUS_ERROR(logger, "printMsgs()");
}

int main()
{
    cout << "Entering main()..." << endl;
    LogLog::getLogLog()->setInternalDebugging(true);
    Logger root = Logger::getRoot();
    try {
        ConfigureAndWatchThread configureThread("log4cplus.properties", 5 * 1000);

        LOG4CPLUS_WARN(root, "Testing....")

        for(int i=0; i<100; ++i) {
            printMsgs(log_1);
            printMsgs(log_2);
            printMsgs(log_3);
            log4cplus::helpers::sleep(1);
        }
    }
```

```

    }
    catch(...) {
        cout << "Exception..." << endl;
        LOG4CPLUS_FATAL(root, "Exception occurred...")
    }

    cout << "Exiting main()..." << endl;
    return 0;
}

```

以下是配置脚本 log4cplus.properties 的内容。

```

log4cplus.rootLogger=INFO, STDOUT, R
log4cplus.logger.test=WARN
log4cplus.logger.test.log_1=FATAL
log4cplus.logger.test.log_2=FATAL
log4cplus.logger.test.log_3=WARN

log4cplus.appender.STDOUT=log4cplus::ConsoleAppender
log4cplus.appender.STDOUT.layout=log4cplus::PatternLayout
log4cplus.appender.STDOUT.layout.ConversionPattern=%d{ %m/%d/%y    %H:%M:%S }
[%t] %-5p %c{2} %%%x%% - %m [%l]%n

log4cplus.appender.R=log4cplus::RollingFileAppender
log4cplus.appender.R.File=output.log
#log4cplus.appender.R.MaxFileSize=5MB
log4cplus.appender.R.MaxFileSize=500KB
log4cplus.appender.R.MaxBackupIndex=5
log4cplus.appender.R.layout=log4cplus::TTCCLayout

```

9 定制 Log4cplus

9.1 定制日志级别

log4cplus 支持日志级别的定制。如果需要定义自己的优先级，则可以按以下步骤进行定制。

(1) 定义新日志级别对应的常量整数和输出宏。

```

/*
 * customloglevel.h
 */
#include <log4cplus/logger.h>
#include <log4cplus/helpers/loglog.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

const LogLevel CRITICAL_LOG_LEVEL = 45000;

#define LOG4CPLUS_CRITICAL(logger, logEvent) \
    if(logger.isEnabledFor(CRITICAL_LOG_LEVEL)) { \
        log4cplus::tostringstream _log4cplus_buf; \
        _log4cplus_buf << logEvent; \
        logger.forcedLog(CRITICAL_LOG_LEVEL,    _log4cplus_buf.str(),    __FILE__, \
        __LINE__); \
    }

```

(2)定义新日志级别对应的字符串、常量整数与字符串之间的转换函数，定义自己的初始化器将转换函数注册到 LogLevelManage。

```

/*
 * customloglevel.cxx
 */
#include "customloglevel.h"

#define _CRITICAL_STRING "CRITICAL"

tstring
criticalToStringMethod(LogLevel ll)
{
    if(ll == CRITICAL_LOG_LEVEL) {
        return _CRITICAL_STRING;
    }
}

```

```

    else {
        return tstring();
    }
}

LogLevel
criticalFromStringMethod(const tstring& s)
{
    if(s == _CRITICAL_STRING) return CRITICAL_LOG_LEVEL;

    return NOT_SET_LOG_LEVEL;
}

class CriticalLogLevelInitializer {
public:
    CriticalLogLevelInitializer() {
        getLogLevelManager().pushToStringMethod(criticalToStringMethod);
        getLogLevelManager().pushFromStringMethod(criticalFromStringMethod);
    }
};

CriticalLogLevelInitializer criticalLogLevelInitializer_;

```

(3)使用新定义的日志级别

9.2 定制 LogLog

LogLog 输出信息中总是包含"log4cplus:"前缀, 这是因为 LogLog 在实现时候在构造函数中进行了硬编码:

```

LogLog::LogLog()
: mutex(LOG4CPLUS_MUTEX_CREATE),
  debugEnabled(false),
  quietMode(false),
  PREFIX( LOG4CPLUS_TEXT("log4cplus: ") ),
  WARN_PREFIX( LOG4CPLUS_TEXT("log4cplus:WARN ") ),

```

```

/*
 * main.cxx
 */
#include "customloglevel.h"
#include <log4cplus/consoleappender.h>
#include <iomanip>
#include <iostream>

using namespace std;
using namespace log4cplus;

int
main()
{
    SharedAppenderPtr append_1(new ConsoleAppender());
    append_1->setName("First");
    Logger::getRoot().addAppender(append_1);

    Logger root = Logger::getRoot();
    LOG4CPLUS_CRITICAL(root, "This is a new logginglevel")

    return 0;
}

```

```
ERR_PREFIX( LOG4CPLUS_TEXT("log4cplus:ERROR ") )
```

```
{
}
```

可以把这些前缀换成自己需要的提示符号，然后重新编译。