linux 下的 log4cplus 应用

log4cplus 主要包括 layout、appender、loglevel 等；实现了 5 个等级的信息：DEBUG、INFO、WARNING、ERROR 和 FATAL。

Layouts ：布局器，控制输出消息的格式.

Appenders ：挂接器，与布局器紧密配合，将特定格式的消息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 logger。

Categories ：分类器，层次化（hierarchy）的结构，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。

使用 log4cplus 有六个基本步骤：

1. 实例化一个 appender 对象
2. 实例化一个 layout 对象
3. 将 layout 对象绑定(attach)到 appender 对象
4. 实例化一个 logger 对象,调用静态函数：log4cplus::Logger::getInstance("logger_name")

5. 将 appender 对象绑定(attach)到 logger 对象，如省略此步骤，标准输出（屏幕）appender 对象会绑定到 logger
6. 设置 logger 的优先级，如省略此步骤，各种有限级的消息都将被记录

下面的例子说明如何将日志写到文件中，在编译是要加上-llog4cplus -lthread

```
//Include .h file
#include <log4cplus/logger.h>
#include <log4cplus/fileappender.h>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <string>
#include <memory>
using namespace log4cplus;
using namespace log4cplus::helpers;
int main()
{
  SharedAppenderPtr pFileAppender(new FileAppender(("testlog.log")));
  pFileAppender->setName("LoggerName");
  std::auto_ptr<Layout> pPatternLayout(new PatternLayout(("%p:%D   -%m [%l]%n")));
  pFileAppender->setLayout(pPatternLayout);

  // 定义 Logger
  Logger pTestLogger = Logger::getInstance(("LoggerName"));

  // 将需要关联 Logger 的 Appender 添加到 Logger 上
  pTestLogger.addAppender(pFileAppender);
```

```
    // 输出日志信息
    //LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message");
    //LOG4CPLUS_DEBUG(pTestLogger, "This is a <Warn> log message");
    pTestLogger.setLogLevel(TRACE_LOG_LEVEL);

    LOG4CPLUS_TRACE(pTestLogger, "printMessages()");
    LOG4CPLUS_DEBUG(pTestLogger, "This is a DEBUG message");
    LOG4CPLUS_INFO(pTestLogger, "This is a INFO message");
    LOG4CPLUS_WARN(pTestLogger, "This is a WARN message");
    LOG4CPLUS_ERROR(pTestLogger, "This is a ERROR message");
    LOG4CPLUS_FATAL(pTestLogger, "This is a FATAL message");

    return 1;
}
```

下面的例子是输出到控制台的一个例子

```
#include <log4cplus/logger.h>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <memory>
#include <string>
using namespace log4cplus;
using namespace log4cplus::helpers;
using namespace std;
int main()
{
    // 定义一个控制台的 Appender
    SharedAppenderPtr pConsoleAppender(new ConsoleAppender());
    // 定义一个简单的 Layout,并绑定到 Appender
    auto_ptr<Layout> pSimpleLayout(new SimpleLayout());
    pConsoleAppender->setLayout(pSimpleLayout);
    // 定义 Logger,并设置优先级
    Logger pTestLogger = Logger::getInstance(("LoggerName"));
    pTestLogger.setLogLevel(WARN_LOG_LEVEL);
    // 将需要关联 Logger 的 Appender 添加到 Logger 上
    pTestLogger.addAppender(pConsoleAppender);
    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message...");

    return 0;
}
```

上面的例子中我使用的 log4cplus 的版本是 log4cplus-1.0.4。

log4cplus 基础

下载

可从网站 http://log4cplus.sourceforge.net 上去下载 log4cplus 1.0.4，具体地址 http://sourceforge.net/projects/log4cplus/files/log4cplus-stable/

安装

本地只有一个 vs2005 的工程，我的是 vs2003 的，所以要搞定工程的问题，去下个 CMake 很轻松就搞定你在工程上的问题。

基础

1. log4cplus 基本元素

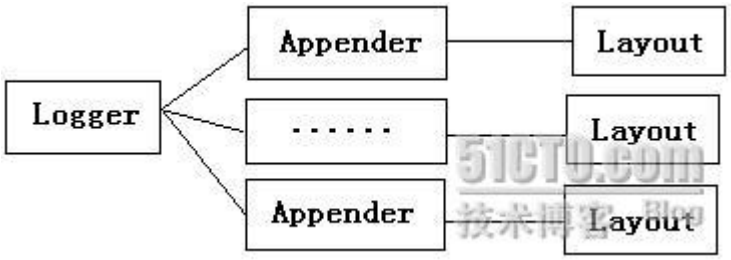
Layouts：控制输出消息的格式。

Appenders：输出位置。

Logger：日志对象。

Priorities：优先权，包括 TRACE, DEBUG, INFO, WARNING, ERROR, FATAL。

2. log4cplus 基本结构



3. 层次性

Logger 对象具有层次结构，按名称区分，如下代码：

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

对象 subtest 是 test 的子对象。

4. 优先级

log4cplus 优先级有低到高：

NOT_SET_LOG_LEVEL：接受缺省的 LogLevel，如果有父 logger 则继承它的 LogLevel

ALL_LOG_LEVEL：开放所有 log 信息输出

TRACE_LOG_LEVEL：开放 trace 信息输出(即 ALL_LOG_LEVEL)

DEBUG_LOG_LEVEL：开放 debug 信息输出

INFO_LOG_LEVEL：开放 info 信息输出

WARN_LOG_LEVEL：开放 warning 信息输出

ERROR_LOG_LEVEL：开放 error 信息输出

FATAL_LOG_LEVEL：开放 fatal 信息输出

OFF_LOG_LEVEL：关闭所有 log 信息输出

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。

5. Layout 格式输出

1.) SimpleLayout

是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"-"。

2.) TTCCLayout

其格式由时间，线程 ID，Logger 和 NDC 组成。

3.) PatternLayout

是一种有词法分析功能的模式布局器,类似正则表达式。以“%”作为开头的特殊预定义标识符，将产生特殊的格式信息。

（1）"%%"，转义为% 。

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

（3）"%D"，显示本地时间，比如："2004-10-16 18:55:45"，%d 显示标准时间。

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"main.cpp"

（5）"%L"，输出当前记录器所在的文件行号，比如"51"

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

（7）"%m"，输出原始信息。

（8）"%n"，换行符。

（9）"%p"，输出 LogLevel，比如"DEBUG"

（10）"%t"，输出记录器所在的线程 ID，比如 "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。

（12）格式对齐，比如"%-10m"时表示左对齐，宽度是 10，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。

6. Appender 输出位置

（1）控制台输出

ConsoleAppender

（2）文件输出

FileAppender / RollingFileAppender / DailyRollingFileAppender .

FileAppender :

实现了基本的文件操作功能，构造函数如下:

FileAppender(filename,mode,immediateFlush);

<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。缺省是 trunc，表示将先前文件删除。

<immediateFlush> 缓冲刷新标志.

RollingFileAppender:

RollingFileAppender(filename,maxFileSize,maxBackupIndex,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

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 。

DailyRollingFileAppender：

DailyRollingFileAppender(filename, schedule,immediateFlush, maxBackupIndex)

filename ：文件名

schedule ：存储频度

immediateFlush：缓冲刷新标志

maxBackupIndex：最大记录文件数

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,WEEKLY,DAILY,TWICE_DAILY,HOURLY,MINUTELY。

7. 使用步骤

a.) 生成 Appender 对象。

b.) 生成 Layout 对象，并绑定到 Appender。(可选)

c.) 生成 Logger 对象。

d.) 设置 Logger 优先级。(可选)

e.) 将需要关联 Logger 的 Appender 添加到 Logger 上。

f.) 使用 Logger 输出信息，所有大于设定的优先级的信息，并在所有挂接在该 Logger 对象上的 Appender 上以相应的 Layout 设定的格式显示出来。

8. 测试代码

1.） 一个基本框架的例子

```
//Include .h file
#include <log4cplus/logger.h>
#include <log4cplus/fileappender.h>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

// Link Lib
#ifndef _DEBUG
#pragma comment(lib,"log4cplusUS.lib")
#else
#pragma comment(lib,"log4cplusUSD.lib")
#endif

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个控制台的 Appender
    SharedAppenderPtr pConsoleAppender(new ConsoleAppender());

    // 定义一个简单的 Layout,并绑定到 Appender
    auto_ptr<Layout> pSimpleLayout(new SimpleLayout());
    pConsoleAppender->setLayout(pSimpleLayout);

    // 定义 Logger,并设置优先级
    Logger pTestLogger = Logger::getInstance(_T("LoggerName"));
    pTestLogger.setLogLevel(WARN_LOG_LEVEL);

    // 将需要关联 Logger 的 Appender 添加到 Logger 上
```

```
    pTestLogger.addAppender(pConsoleAppender);

    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message...");

    return 0;
}
```
运行结果，在控制台输出：
```
WARN - This is a <Warn> log message...
```

2.）一个精简模式的例子
```
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个控制台的 Appender
    SharedAppenderPtr pConsoleAppender(new ConsoleAppender());

    // 定义 Logger
    Logger pTestLogger = Logger::getInstance(_T("LoggerName"));

    // 将需要关联 Logger 的 Appender 添加到 Logger 上
    pTestLogger.addAppender(pConsoleAppender);

    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message...");

    return 0;
}
```
运行结果，在控制台输出：
```
WARN - This is a <Warn> log message...
```

3.）输出更多的信息内容
```
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个控制台的 Appender
    SharedAppenderPtr pConsoleAppender(new ConsoleAppender());

    // 定义 Logger
    Logger pTestLogger = Logger::getInstance(_T("LoggerName"));

    // 将需要关联 Logger 的 Appender 添加到 Logger 上
    pTestLogger.addAppender(pConsoleAppender);

    int n = 6 ;
    TCHAR *p = _T("TestString") ;
```

```
    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger,"This is a <Warn> log message..."<<n<<" "<<p);

  return 0;
}
```
运行结果，在控制台输出：
```
WARN - This is a <Warn> log message...6 TestString
```

4.）输出到日志文件
```
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个文件 Appender
    SharedAppenderPtr pFileAppender(new FileAppender(_T("d:\\testlog.log")));

    // 定义 Logger
    Logger pTestLogger = Logger::getInstance(_T("LoggerName"));

    // 将需要关联 Logger 的 Appender 添加到 Logger 上
    pTestLogger.addAppender(pFileAppender);

    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message...");

    return 0;
}
```
运行结果，在文件 d:\\testlog.log 中 输出：
```
WARN - This is a <Warn> log message...
```

5.）使用更多的格式控制
```
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义 1 个控制台的 Appender,3 个文件 Appender
    SharedAppenderPtr pConsoleAppender(new ConsoleAppender());
    SharedAppenderPtr pFileAppender1(new FileAppender(_T("d:\\testlog1.log")));
    SharedAppenderPtr pFileAppender2(new FileAppender(_T("d:\\testlog2.log")));
    SharedAppenderPtr pFileAppender3(new FileAppender(_T("d:\\testlog3.log")));

    // 定义一个简单的 Layout,并绑定到 pFileAppender1
    auto_ptr<Layout> pSimpleLayout(new SimpleLayout());
    pFileAppender1->setLayout(pSimpleLayout);

    // 定义一个 TTCLayout,并绑定到 pFileAppender2
    auto_ptr<Layout> pTTCLayout(new TTCCLayout());
```

```
    pFileAppender2->setLayout(pTTCLayout);

    // 定义一个 PatternLayout,并绑定到 pFileAppender3
    auto_ptr<Layout> pPatternLayout(new PatternLayout(_T("%d{%m/%d/%y
%H:%M:%S}  - %m [%l]%n")));
    pFileAppender3->setLayout(pPatternLayout);

    // 定义 Logger
    Logger pTestLogger = Logger::getInstance(_T("LoggerName"));

    // 将需要关联 Logger 的 Appender 添加到 Logger 上
    pTestLogger.addAppender(pConsoleAppender);
    pTestLogger.addAppender(pFileAppender1);
    pTestLogger.addAppender(pFileAppender2);
    pTestLogger.addAppender(pFileAppender3);

    // 输出日志信息
    LOG4CPLUS_WARN(pTestLogger, "This is a <Warn> log message...");

    return 0;
}
```

输出结果：

在控制台输出：

```
WARN - This is a <Warn> log message...
```

在 d:\testlog1.log 中输出：

```
WARN - This is a <Warn> log message...
```

在 d:\testlog2.log 中输出：

```
03-30-09 15:07:50,234 [1188] WARN LoggerName <> - This is a <Warn> log message...
```

在 d:\testlog3.log 中输出：

```
03/30/09 07:07:50 - This is a <Warn> log message...
[e:\study\testcode\vs2008\testlog4cplus\testlog4cplus\testlog4cplus.cpp:121]
```

9. 把设置移到配置文件中

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。

配置步骤：

1.）配置 Appender 名称

枚举：

```
log4cplus.appender.AppenderName1=log4cplus::ConsoleAppender
log4cplus.appender.AppenderName2=log4cplus::FileAppender
log4cplus.appender.AppenderName3=log4cplus::RollingFileAppender
log4cplus.appender.AppenderName4=log4cplus::DailyRollingFileAppender
log4cplus.appender.AppenderName5=log4cplus::SocketAppender
```

2.）配置 Layout

可以选择不设置、TTCCLayout、或 PatternLayout

设置 TTCCLayout 如下所示：

log4cplus.appender.AppenderName.layout=log4cplus::TTCCLayout

设置 PatternLayout 如下所示：

log4cplus.appender.AppenderName.layout=log4cplus::PatternLayout

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

3.）配置 Filter

可选择的 Filter: LogLevelMatchFilter,LogLevelRangeFilter 和 StringMatchFilter.

对 LogLevelMatchFilter 来说,过滤条件包括 LogLevelToMatch 和 AcceptOnMatch,只有当 log 信息的 LogLevel 值与 LogLevelToMatch 相同，且 AcceptOnMatch 为 true 时才会匹配。

对 LogLevelRangeFilter 来说，过滤条件包括 LogLevelMin、LogLevelMax 和 AcceptOnMatch，只有当 log 信息的 LogLevel 在 LogLevelMin、LogLevelMax 之间同时 AcceptOnMatch 为 true 时才会匹配。

对 StringMatchFilter 来说，过滤条件包括 StringToMatch 和 AcceptOnMatch，只有当 log 信息的 LogLevel 值与 StringToMatch 对应的 LogLevel 值与相同，且 AcceptOnMatch 为 true 时会匹配。

4.）配置 Logger

对于 RootLogger,如：

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

对于 non-root logger 来说：

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

5.）加载配置

PropertyConfigurator::doConfigure("cfg_filename");

6.）举例：

a.) 对 Appender 对象进行配置，包括设置 Appende 名称,Layout 和 Filter 及相关参数。

设置 Appender 名称及参数，如：

log4cplus.appender.AppenderName=log4cplus::FileAppender

log4cplus.appender.AppenderName.File=d:\testlog.log

设置 Layout 及参数，如：

log4cplus.appender.AppenderName.layout=log4cplus::PatternLayout

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

设置 Filter 及参数，如：

log4cplus.appender.AppenderName.filters.1=log4cplus::spi::LogLevelRangeFilter

log4cplus.appender.AppenderName.filters.1.LogLevelMin=DEBUG

log4cplus.appender.AppenderName.filters.1.LogLevelMax=INFO

log4cplus.appender.AppenderName.filters.1.AcceptOnMatch=true

log4cplus.appender.AppenderName.filters.2=log4cplus::spi::DenyAllFilter

b.) 设置 Logger 对象

log4cplus.rootLogger=TRACE,AppenderName1,AppenderName2

c.) 将以上配置保存到文件中，比如：d:\log4clpus.cfg 中,并如何使用：

```
int _tmain(int argc, _TCHAR* argv[])
{
	PropertyConfigurator::doConfigure(_T("d:\\log4cplus.cfg"));

	Logger logger = Logger::getRoot();

	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");

	return 0;
}
```

log4cplus 库简单使用

一.简介

log4cplus 是一个日志记录的库，目的很简单，就是把合适的信息送到正确的位置上去。在服务器程序上使用非常方便。

开发库下载地址可以去 baidu 搜一下，是开源的哦！

二.组成

Log4cplus 由 4 部分组成：

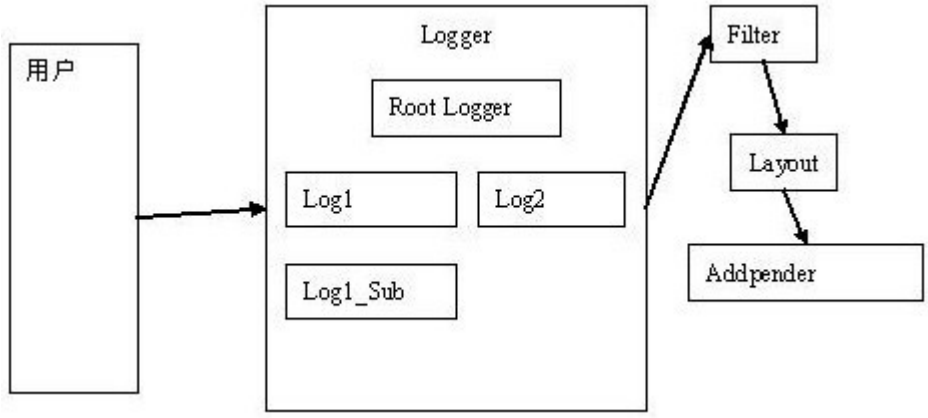
(1) Logger 日志模块,程序中唯一一个必须得使用的模块，解决了在哪里使用日志的问题

(2) Appenders 接收日志的各个设备,如控制台、文件、网络等。解决了输出到哪里去的问题

(3)Layout 格式化输出信息,解决了如何输出的问题。

(4)Filter 过滤器，解决哪些信息需要输出的问题，比如 DEBUG，WARR,INFO 等的输出控制.

Log4cplus 的主要部件关系图如下：



三.一些简单示例

(1) 各对象可以看下上图理解一下，以下为 严格实现步骤 1-6

```
#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <iostream>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <conio.h>
#include <log4cplus/helpers/sleep.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

int _tmain(int argc, _TCHAR* argv[])…{

    /**//* step 1: Instantiate an appender object */
    SharedObjectPtr<Appender> _append (new ConsoleAppender());
```

```
    _append->setName(LOG4CPLUS_TEXT("append for test"));
    /**//* step 2: Instantiate a layout object */
    //std::string pattern = ;
                                                std::auto_ptr<Layout>          _layout(new
PatternLayout(LOG4CPLUS_TEXT("%d{%m/%d/%y %H:%M:%S}  - %m [%l]%n")));
    /**//* step 3: Attach the layout object to the appender */
    _append->setLayout( _layout );
    /**//* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance(LOG4CPLUS_TEXT("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…");

    _getch();
    return 0;
}
```

(2) 简洁使用模式，appender 输出到屏幕

```
#include "stdafx.h"
#pragma comment(lib,"../bin/log4cplusD.lib")

#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <iostream>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <conio.h>
#include <log4cplus/helpers/sleep.h>

using namespace log4cplus;
using namespace log4cplus::helpers;

int _tmain(int argc, _TCHAR* argv[])…{

    /**//* step 1: Instantiate an appender object */
    SharedAppenderPtr _append(new ConsoleAppender());
    _append->setName(LOG4CPLUS_TEXT("append test"));

    /**//* step 4: Instantiate a logger object */
```

```
    Logger _logger = Logger::getInstance(LOG4CPLUS_TEXT("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…");
    _getch();
    return 0;
}
```

（3）简洁使用模式，appender 输出到屏幕

```
#include "stdafx.h"
#pragma comment(lib,"../bin/log4cplusD.lib")

#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <iostream>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <conio.h>
#include <log4cplus/helpers/sleep.h>

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

int _tmain(int argc, _TCHAR* argv[])…{
    /**//* step 1: Instantiate an appender object */
    SharedAppenderPtr _append(new ConsoleAppender());
    _append->setName(LOG4CPLUS_TEXT("append test"));

    /**//* step 4: Instantiate a logger object */
    Logger _logger = Logger::getInstance(LOG4CPLUS_TEXT("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) ;    //
不会被打印
    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 << 100000000) ;

    LOG4CPLUS_FATAL(_logger, "This is a double: "     << 1.2345234234);

    _getch();
    return 0;
}
```

（4）通过 loglog 来控制输出调试、警告或错误信息，appender 输出到屏幕。

```
#include "stdafx.h"
#pragma comment(lib,"../bin/log4cplusD.lib")

#include <log4cplus/logger.h>
#include <log4cplus/configurator.h>
#include <iostream>
#include <log4cplus/consoleappender.h>
#include <log4cplus/layout.h>
#include <conio.h>
#include <log4cplus/helpers/sleep.h>
#include <log4cplus/helpers/loglog.h>

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

⊟⊞void printMsgs(void)…{

    std::cout << "Entering printMsgs()…" << std::endl;

    LogLog::getLogLog()->debug(LOG4CPLUS_TEXT("This is a Debug statement…"));

   LogLog::getLogLog()->warn(LOG4CPLUS_TEXT("This is a Warning…"));

    LogLog::getLogLog()->error(LOG4CPLUS_TEXT("This is a Error…"));

    std::cout << "Exiting printMsgs()…" << std::endl << std::endl;
}

⊟⊞int _tmain(int argc, _TCHAR* argv[])…{
```

```
    /**//*
    LogLog 类实现了 debug, warn, error 函数用于输出调试、警告或错误信息，
    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。
    LogLog::getLogLog()->setInternalDebugging(false);
*/
    //默认级别是不打印 debug 信息
    printMsgs();

    std::cout << "Turning on debug…" << std::endl;

    LogLog::getLogLog()->setInternalDebugging(true);

    //打开 debug 方式
    //全部打印
    printMsgs();

    std::cout << "Turning on quiet mode…" << std::endl;

    LogLog::getLogLog()->setQuietMode(true);

    //屏蔽所有输出信息，所以不全打印各种级别信息
    printMsgs();

    _getch();
    return 0;
}
```

/**//*输入结果如下：

```
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()…

都有个 log4cplus:打头，可以自定义修改以下部分代码

```
LogLog::LogLog() : mutex(LOG4CPLUS_MUTEX_CREATE),     debugEnabled(false),
 quietMode(false),     PREFIX( LOG4CPLUS_TEXT("log4cplus: ") ),
   WARN_PREFIX( LOG4CPLUS_TEXT("log4cplus:WARN ") ),
   ERR_PREFIX( LOG4CPLUS_TEXT("log4cplus:ERROR ") ){}
*/
```

（5）文件模式，appender 输出到文件

```
 #include "stdafx.h"
 #pragma comment(lib,"../bin/log4cplusD.lib")

 #include <log4cplus/logger.h>
 #include <log4cplus/fileappender.h>
 #include <conio.h>

 using namespace log4cplus;

int _tmain(int argc, _TCHAR* argv[])…{
        /**//* step 1: Instantiate an appender object */
       SharedAppenderPtr _append(new FileAppender(LOG4CPLUS_TEXT("Test.log")));

      _append->setName(LOG4CPLUS_TEXT("file log test"));

       /**//* step 4: Instantiate a logger object */
                                                  Logger          _logger          =
Logger::getInstance(LOG4CPLUS_TEXT("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 #");
    }

    _getch();

}
```

log4cplus 使用小记

在项目中需要使用日志系统，以前的编程中也没有日志系统的经验。原本打算使用公共组件中的日志系统的，但是没有文档，也没有介绍，看着一堆函数茫然 了，像什么 rollback 之类的，后来才知道有按日期和大小回滚两种方式，也就是生成新的 log 文件。而且听同事说公共组件中的日志系统好像在多线程的 环境下会出现不输出日志的情况，所以最后只能找一些开源的日志程序来替代公共组件。

用于 C++的日志程序很多，log4cxx/log4cpp/log4cplus 还有另外的一些，最后选择了 log4cplus。原因很简单，这个是在 我的 ubuntu 上唯一顺利编译通过的。像 log4cxx 是 apache 的，还需要一些 apache 的相应软件包去支持。

软件下载在官方主页上 http://log4cplus.sourceforge.net/。编译安装和其他程序没有什么区别。完了之后在自己的程序加 上相应头文件的目录和库文件的相应路径。在 LD_LIBRARY_PATH 中添加相应的库目录。这样程序运行时才知道到哪里去找相应的库。

由于以前没有经验，拿到手根本不知道该怎么用，也不知道网上说的配置文件在哪里，后来终于搞清楚了。

使用起来还是很方便的，在每个类里面定义一个私有的 Logger logger。然后在类的初始化代码中添加：

```
PropertyConfigurator::doConfigure("./config/log4cplus.properties");
logger = Logger::getInstance("UpdateDatanode");
```

然后在相应位置放置一个 log4cplus.properties 文件，里面的内容为：

```
#############################################################
# Define the root logger
log4cplus.rootLogger=TRACE, consoleAppender, fileAppender

# Define a file appender named "consoleAppender"
log4cplus.appender.consoleAppender=log4cplus::ConsoleAppender
log4cplus.appender.consoleAppender.layout=log4cplus::PatternLayout
log4cplus.appender.consoleAppender.layout.ConversionPattern=%p        %D{%m/%d/%y
%H:%M:%S}   - %m [%l]%n

# Define a file appender named "fileAppender"
log4cplus.appender.fileAppender=log4cplus::RollingFileAppender
log4cplus.appender.fileAppender.MaxFileSize=200KB
log4cplus.appender.fileAppender.File=./log.log
log4cplus.appender.fileAppender.MaxBackupIndex=3
log4cplus.appender.fileAppender.layout=log4cplus::PatternLayout
log4cplus.appender.fileAppender.layout.ConversionPattern=%p        %D{%m/%d/%y
%H:%M:%S}   - %m [%l]%n
#############################################################
```

这样子后面就可以使用 LOG4CPLUS_DEBUG，LOG4CPLUS_ERROR 进行打印了。这个配置文件会分别将日志打印到屏幕和日志文件中去。在多线程的情况下也使用良好。

log4cplus 暂时就用了这基本的功能，还有很多东西可以学习研究下。

转一些相关的说明：

log4cplus 内容介绍

1. Logger 对象

Logger 对象具有层次结构，按名称区分，如下代码：

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

对象 subtest 是 test 的子对象。

2. 优先级

log4cplus 优先级有低到高：

NOT_SET_LOG_LEVEL ：接受缺省的 LogLevel，如果有父 logger 则继承它的 LogLevel

ALL_LOG_LEVEL ：开放所有 log 信息输出

TRACE_LOG_LEVEL ：开放 trace 信息输出(即 ALL_LOG_LEVEL)

DEBUG_LOG_LEVEL ：开放 debug 信息输出

INFO_LOG_LEVEL ：开放 info 信息输出

WARN_LOG_LEVEL ：开放 warning 信息输出

ERROR_LOG_LEVEL ：开放 error 信息输出

FATAL_LOG_LEVEL ：开放 fatal 信息输出

OFF_LOG_LEVEL ：关闭所有 log 信息输出

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。

3.Layout 格式输出

1.) SimpleLayout

是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"-"。

2.) TTCCLayout

其格式由时间，线程 ID，Logger 和 NDC 组成。

3.) PatternLayout

是一种有词法分析功能的模式布局器,类似正则表达式。以“%”作为开头的特殊预定义标识符，将产生特殊的格式信息。

（1）"%%"，转义为% 。

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

（3）"%D"，显示本地时间，比如："2004-10-16 18:55:45"，%d 显示标准时间。 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"main.cpp"
（5）"%L"，输出当前记录器所在的文件行号，比如"51"
（6）"%l"，输出当前记录器所在的文件名称和行号，比如"main.cpp:51"
（7）"%m"，输出原始信息。
（8）"%n"，换行符。
（9）"%p"，输出 LogLevel，比如"DEBUG"
（10）"%t"，输出记录器所在的线程 ID，比如 "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。
（12）格式对齐，比如"%-10m"时表示左对齐，宽度是 10，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。

4.) Appender 输出位置
（1）控制台输出
ConsoleAppender
（2）文件输出
FileAppender / RollingFileAppender / DailyRollingFileAppender .

FileAppender :
实现了基本的文件操作功能，构造函数如下:
FileAppender(filename,mode,immediateFlush);

<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。缺省是 trunc，表示将先前文件删除。
<immediateFlush> 缓冲刷新标志.

RollingFileAppender:

RollingFileAppender(filename,maxFileSize,maxBackupIndex,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

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 。

DailyRollingFileAppender：

DailyRollingFileAppender(filename, schedule,immediateFlush, maxBackupIndex)

filename ：文件名
schedule ：存储频度
immediateFlush：缓冲刷新标志
maxBackupIndex：最大记录文件数

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,WEEKLY,DAILY,TWICE_DAILY,HOURLY,MINUTELY。

开源日志系统 log4cplus

log4cplus 是 C++编写的开源的日志系统，功能非常全面，用到自己开发的工程中会比较专业的，本文介绍了 log4cplus 基本概念，以及如何安装，配置。

=== 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，通过将信息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；你可以选择将信息输出到屏幕、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。

=== 下载 ===

最新的 log4cplus 可以从以下网址下载 http://log4cplus.sourceforge.net
本文使用的版本为：1.0.2

=== 安装 ===

1. linux 下安装

```
tar xvzf log4cplus-x.x.x.tar.gz
cd log4cplus-x.x.x
./configure --prefix=/where/to/install
make
make install
```

这里我采用缺省安装路径：/usr/local，下文如无特别说明，均以此路径为准。

2. windows 下安装

不需要安装，有一个 msvc6 存放包括源代码和用例在内的开发工程（for VC6 only），使用之前请先编译"log4cplus_dll class"工程生成 dll，或者编译"log4cplus_static class"工程生成 lib.

=== 使用前的配置 ===

1. linux 下的配置

确保你的 Makefile 中包含 /usr/local/lib/liblog4cplus.a（静态库）或 -llog4cplus（动态库）即可，头文件在/usr/local/include/log4cplus 目录下。对于动态库，要想正常使用，还得将库安装路径加入到 LD_LIBRARY_PATH 中，我一般是这样做的：以管理员身份登录，在 /etc/ld.so.conf 中加入安装路径，这里是/usr/local/lib，然后执行 ldconfig 使设置生效即可。

2. windows 下的配置

将"log4cplus_dll class"工程或"log4cplus_static class"工程的 dsp 文件插入到你的工程中，或者直接把两个工程编译生成的库以及头文件所在目录放到你的工程的搜索路径中，如果你使用静态库，请在你的工程中"project/setting/C++"的 preprocessor definitions 中加入 LOG4CPLUS_STATIC。

=== 构成要素介绍 ===

虽然功能强大，应该说 log4cplus 用起来还是比较复杂的，为了更好地使用它，先介绍一下

它的基本要素。
Layouts ：布局器，控制输出消息的格式.
Appenders ：挂接器，与布局器紧密配合，将特定格式的消息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 logger。
Categories ：分类器，层次化（hierarchy）的结构，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。
Priorities ：优先权，包括 TRACE, DEBUG, INFO, WARNING, ERROR, FATAL。

本文介绍了 log4cplus 基本概念，以及如何安装，配置，下一篇将通过例子介绍如何使用 log4cplus。

本文介绍了使用 log4cplus 有六个步骤，并提供了一些例子引导你了解 log4cplus 的基本使用。

=== 基本使用 ===
使用 log4cplus 有六个基本步骤：
1. 实例化一个 appender 对象
2. 实例化一个 layout 对象
3. 将 layout 对象绑定(attach)到 appender 对象
4. 实例化一个 logger 对象,调用静态函数：log4cplus::Logger::getInstance("logger_name")
5. 将 appender 对象绑定(attach)到 logger 对象，如省略此步骤，标准输出（屏幕）appender 对象会绑定到 logger
6. 设置 logger 的优先级，如省略此步骤，各种有限级的消息都将被记录

下面通过一些例子来了解 log4cplus 的基本使用。
〖例 1〗
```
/*
    严格实现步骤 1-6，appender 输出到屏幕，其中的布局格式和 LogLevel 后面会详细解释。
*/
#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_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(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]
```

〖例 2〗

```
/*
        简洁使用模式，appender 输出到屏幕。
*/
#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...
```

〖例 3〗

```
/*
    iostream 模式，appender 输出到屏幕。
*/
#include <log4cplus/logger.h>
#include <log4cplus/consoleappender.h>
#include <iomanip> /* 其实这个东东还是放到 log4cplus 头文件中比较合适些，个人意见：）
*/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 << 100000000)
    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
```

〖例 4〗

```
/*
    调试模式，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()
{
      /*
     LogLog 类实现了 debug, warn, error  函数用于输出调试、警告或错误信息，
     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。
     LogLog::getLogLog()->setInternalDebugging(false);
      */
      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()...
```

需要指出的是，输出信息中总是包含"log4cplus:"前缀，有时候会感觉不爽，这是因为 LogLog

在实现时候死定了要这么写：

```
LogLog::LogLog()
  : mutex(LOG4CPLUS_MUTEX_CREATE),
    debugEnabled(false),
    quietMode(false),
    PREFIX( LOG4CPLUS_TEXT("log4cplus: ") ),
    WARN_PREFIX( LOG4CPLUS_TEXT("log4cplus:WARN ") ),
    ERR_PREFIX( LOG4CPLUS_TEXT("log4cplus:ERROR ") )
{
}
```

你可以把这些前缀换成自己看着爽的提示符号，然后重新编译，hihi。除非万不得已或者实在郁闷的不行，否则还是不要这样干。

〖例 5〗

```
/*      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 #
```

本文介绍了三种控制输出格式的布局管理器的概念和使用情况，通过掌握这些知识，可以更

有效地控制 log 系统输出尽可能贴近你需求的信息来。

=== 如何控制输出消息的格式 ===
前面已经讲过，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。其中：
1. SimpleLayout
是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"-"。
比如以下代码片段：

```
... ...
/* 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 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...")

... ...
```

将打印结果：
DEBUG - This is the simple formatted log message...

2. PatternLayout
是一种有词法分析功能的模式布局器，一提起模式就会想起正则表达式，这里的模式和正则表达式类似，但是
远比后者简单，能够对预定义的标识符（称为 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(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]

可以看出通过填写特定格式的模式字符串"pattern"，原始信息被包含到一堆有格式的信息当中了，这就使得

用户可以根据自身需要来定制显示内容。"pattern"可以包含普通字符串和预定义的标识符，其中：

（1）普通字符串，能够被直接显示的信息。

（2）预定义标识符，通过"%"与一个或多个字符共同构成预定义的标识符，能够产生出特定格式信息。

关于预定义标识符，log4cplus 文档中提供了详细的格式说明，我每种都试了一下，以上述代码为例，根据不同

的 pattern，各种消息格式使用情况列举如下：

（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" （因为我们是东 8 区，差 8 个小时啊）。

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"，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（刚接触 log4cplus 文档时还以为"%-5p"整个字符串代表 LogLevel 呢，呵呵）。

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（怎么得名的？Logger 里哪里有那个"C"的缩写啊！名字起得真够烂的，想扁人）。提供给那些想显示典型的信息（一般情况下够用了）又懒得配置 pattern 的同志们。
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)
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

当构造 TTCCLayout 对象时选择 GMT 时间格式时：

```
    ... ...

    /* 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

本文介绍了控制 log 信息格式的相关知识，下一部分将详细介绍 log 信息的几种文件操作方式。

将 log 信息记录到文件应该说是日志系统的一个基本功能，log4cplus 在此基础上，提供了更多的功能，可以按照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，依次类推；或者按照日期来决定是否转储。本文将详细介绍这些用法。

=== 如何将 log 记录到文件 ===
我们在例 5 中给出了一个将 log 记录到文件的例子，用的是 FileAppender 类实现的，log4cplus 提供了三个类用于文件操作，它们是 FileAppender 类、RollingFileAppender 类、DailyRollingFileAppender 类。

1. FileAppender 类
实现了基本的文件操作功能，构造函数如下：
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，呵呵，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例 5，这里不再赘述。

2. RollingFileAppender 类
构造函数如下：

```
log4cplus::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       {
    NDCContextCreator _context("loop");
    LOG4CPLUS_DEBUG(subTest, "Entering loop #" << i)
```

```
}

... ...
```

运行结果：
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.

3. DailyRollingFileAppender 类
构造函数如下：

```
DailyRollingFileAppender::DailyRollingFileAppender(const log4cplus::tstring& filename,
                DailyRollingFileSchedule schedule,
                bool immediateFlush,
                int maxBackupIndex)
```

filename　：文件名
schedule　：存储频度
immediateFlush：缓冲刷新标志
maxBackupIndex：最大记录文件数
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（每时）、MINUTELY（每分）。
maxBackupIndex 的含义同上所述，比如以下代码片段：

```
... ...

SharedAppenderPtr _append(new DailyRollingFileAppender("Test.log", MINUTELY,
true, 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      {
NDCContextCreator _context("loop");
LOG4CPLUS_DEBUG(subTest, "Entering loop #" << i)
}

... ...
```

运行结果：
运行后会以分钟为单位，分别生成名为 Test.log.2004-10-17-03-03 、Test.log.2004-10-17-03-04 和 Test.log.2004-10-17-03-05 这样的文件。
需要指出的是，刚看到按照频度（如 HOURLY、MINUTELY）转储这样的概念，以为 log4cplus 提供了内部定时器，感觉很奇怪，因为日志系统不应该主动记录，而 loging 事件总是应该被动触发的啊。仔细看了源代码后才知道这里的"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。

本部分详细介绍 log 信息的几种文件操作方式，下面将重点介绍一下如何有选择地控制 log 信息的输出。

日志系统的另一个基本功能就是能够让使用者按照自己的意愿来控制什么时候，哪些 log 信息可以输出。
如果能够让用户在任意时刻设置允许输出的 LogLevel 的信息就好了，log4cplus 通过 LogLevelManager、LogLog、Filter 三种方式实现了上述功能。

=== 优先级控制 ===
在研究 LogLevelManager 之前，首先介绍一下 log4cplus 中 logger 的存储机制，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( ... );
```

logger 的这种父子关联性会体现在优先级控制方面，log4cplus 将输出的 log 信息按照 LogLevel（从低到高）分为：
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
LogLevelManager 负责设置 logger 的优先级，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,我们举例说明：

〖例 6〗

```
#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.subtest 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.subtest 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
```

下面的例子演示了如何通过设置 LogLevel 来控制用户的 log 信息输出：

〖例 7〗

```
#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
```

用户也可以自行定义 LogLevel，操作比较简单，首先要定义 LEVEL 值，比如 HELLO_LOG_LEVEL 定义如下：

```
/* DEBUG_LOG_LEVEL   < HELLO_LOG_LEVEL < INFO_LOG_LEVEL */
const LogLevel HELLO_LOG_LEVEL = 15000;
```

然后定义以下宏即可：

```
/* define MACRO LOG4CPLUS_HELLO */
#define LOG4CPLUS_HELLO(logger, logEvent) \
    if(logger.isEnabledFor(HELLO_LOG_LEVEL)) { \
    log4cplus::tostringstream _log4cplus_buf; \
    _log4cplus_buf << logEvent; \
```

```
 logger.forcedLog(HELLO_LOG_LEVEL, _log4cplus_buf.str(), __FILE__, __LINE__); \
    }
```

不过 log4cplus 没有提供给用户一个接口来实现 LEVEL 值与字符串的转换，所以当带格式输出 LogLevel 字符串时候会显示"UNKNOWN"， 不够理想。比如用 TTCCLayout 控制输出的结果可能会如下所示：

10-17-04 11:17:51,124 [1075298944] UNKNOWN root <> - info

而不是期望的以下结果：

10-17-04 11:17:51,124 [1075298944] HELLO root <> - info

要想实现第二种结果，按照 log4cplus 现有的接口机制，只能改其源代码后重新编译，方法是在 loglevel.cxx 中加入：

#define _HELLO_STRING LOG4CPLUS_TEXT("HELLO")

然后修改 log4cplus::tstring defaultLogLevelToStringMethod(LogLevel ll)函数，增加一个判断：

case HELLO_LOG_LEVEL: return _HELLO_STRING;

重新编译 log4cplus 源代码后生成库文件，再使用时即可实现满意效果。

=== 调试模式 ===

即通过 loglog 来控制输出调试、警告或错误信息，见例 4，这里不再赘述。

=== 基于脚本配置来过滤 log 信息 ===

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，我将在全文的最后一部分中提到多线程环境中如何利用脚本配置来配合实现性能测试，本节将重点介绍基脚本实现过滤 log 信息的功能。

首先简单介绍一下脚本的语法规则：

包括 Appender 的配置语法和 logger 的配置语法，其中：

1.Appender 的配置语法:

（1）设置名称：

/*设置方法*/

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

例如（列举了所有可能的 Appender，其中 SocketAppender 后面会讲到）：

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

（2）设置 Filter：

包括选择过滤器和设置过滤条件，可选择的过滤器包括：LogLevelMatchFilter、LogLevelRangeFilter、

和 StringMatchFilter：

对 LogLevelMatchFilter 来说，过滤条件包括 LogLevelToMatch 和 AcceptOnMatch（true|false）， 只有

当 log 信息的 LogLevel 值与 LogLevelToMatch 相同，且 AcceptOnMatch 为 true 时才会匹

配。
LogLevelRangeFilter 来说，过滤条件包括 LogLevelMin、LogLevelMax 和 AcceptOnMatch，只有当 log 信息
的 LogLevel 在 LogLevelMin、LogLevelMax 之间同时 AcceptOnMatch 为 true 时才会匹配。
对 StringMatchFilter 来说，过滤条件包括 StringToMatch 和 AcceptOnMatch，只有当 log 信息的 LogLevel 值
与 StringToMatch 对应的 LogLevel 值与相同， 且 AcceptOnMatch 为 true 时会匹配。

过滤条件处理机制类似于 IPTABLE 的 Responsibility chain，（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。
（3）设置 Layout
可以选择不设置、TTCCLayout、或 PatternLayout
如果不设置，会输出简单格式的 log 信息。
设置 TTCCLayout 如下所示：
log4cplus.appender.ALL_MSGS.layout=log4cplus::TTCCLayout
设置 PatternLayout 如下所示：
log4cplus.appender.append_1.layout=log4cplus::PatternLayout
log4cplus.appender.append_1.layout.ConversionPattern=%d{%m/%d/%y %H:%M:%S,%Q} [%t] %-5p - %m%n

2.logger 的配置语法
包括 rootLogger 和 non-root logger。
对于 rootLogger 来说：
log4cplus.rootLogger=[LogLevel], appenderName, appenderName, ...
对于 non-root logger 来说：
log4cplus.logger.logger_name=[LogLevel|INHERITED], appenderName, appenderName, ...

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。

〖例 8〗
```
/*
 *      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

/*
 *      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;
}
```

运行结果：

```
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
```

本部分详细介绍了如何有选择地控制 log 信息的输出，最后一部分我们将介绍一下多线程、和 C/S 模式下该如何操作，顺便提一下 NDC 的概念。

log4cplus 在很多方面做的都很出色，但是使用过程有些地方感觉不爽。在继续吹捧之前我先把不爽之处
稍微提一提，然后继续介绍关于线程和套接字的知识。

=== 一些可以改进之处 ===

1. 用户自定义 LogLevel 的实现机制不够开放

在第五篇中曾经介绍过如何实现用户自行定义 LogLevel，为了实现比较理想的效果，甚至还需要改 log4cplus 的源代码。：（

2. 生成 Logger 对象的机制可以改进

我在使用时候，经常需要在不同的文件、函数中操作同一个 logger，虽然 log4cplus 实现了树状存储以及根据名称生成 Logger，却没有充分利用这样的特点确保同一个名称对应的 logger 对象的唯一性，比如以下代码：

```
    ... ...

    Logger logger1 = Logger::getInstance("test");
```

```
    Logger logger2 = Logger::getInstance("test");
    Logger * plogger1 = &logger1;
    Logger * plogger2 = &logger2;
    std::cout << "plogger1: " << plogger1 << std::endl << "plogger2: " << plogger2 << std::endl;

    ... ...
```

运行结果：

```
plogger1: 0xbfffe5a0
plogger2: 0xbfffe580
```

从结果可以看出，明明是同一个 Logger，但每次调用都会产生一个 Logger 副本，虽然结果是正确的（因为将存储和操作分开了），但是资源有些浪费，我看了一下 log4cplus 的代码，其实可以按照如下方式实现（示意性的）：

```
#include <iostream>
#include <string>
#include <map>
/* forward declaration */
class Logger;
class LoggerContainer
{
public:
    ~LoggerContainer();
    Logger * getinstance(const std::string & strLogger);
private:
    typedef std::map<:string,> LoggerMap;
    LoggerMap loggerPtrs;
};
class Logger
{
public:
  Logger() {std::cout << "ctor of Logger " << std::endl; }
    ~Logger() {std::cout << "dtor of Logger " << std::endl; }
    static Logger * getInstance( const std::string & strLogger)
    {
    static LoggerContainer defaultLoggerContainer;
    return defaultLoggerContainer.getinstance(strLogger);
    }
};
LoggerContainer::~LoggerContainer()
{
    /* release all ptr in LoggerMap */
```

```
      LoggerMap::iterator itr = loggerPtrs.begin();
      for( ; itr != loggerPtrs.end(); ++itr )
 {
 delete (*itr).second;
 }
}
Logger * LoggerContainer::getinstance(const std::string & strLogger)
{
    LoggerMap::iterator itr = loggerPtrs.find(strLogger);
    if(itr != loggerPtrs.end())
    {
    /* logger exist, just return it */
    return (*itr).second;
    }
    else
    {
    /* return a new logger */
    Logger * plogger = new Logger();
    loggerPtrs.insert(std::make_pair(strLogger, plogger));
    return plogger;
    }
}
int main()
{
      Logger * plogger1 = Logger::getInstance("test");
      Logger * plogger2 = Logger::getInstance("test");
      std::cout << "plogger1: " << plogger1 << std::endl << "plogger2: " << plogger2 <<
std::endl;
      return 0;
}
```

运行结果：

```
ctor of Logger
plogger1: 0x804fc30
plogger2: 0x804fc30
dtor of Logger
```

这里的 LoggerContainer 相当于 log4cplus 中的 Hierarchy 类，结果可以看出，通过同一个名称可以获取相同的 Logger 实例。

还有一些小毛病比如 RollingFileAppender 和 DailyRollingFileAppender 的参数输入顺序可以调整成统一方式等等，就不细说了。

本部分提到了使用 log4cplus 时候感觉不爽的地方，最后一部分将介绍一下 log4cplus 中线程和套接字实现情况经过短暂的熟悉过程，log4cplus 已经被成功应用到了我的项目中去了，效果还不错，:）除了上文提及的功能之外，下面将介绍 log4cplus 提供的线程和套接字的使

用情况。

=== NDC ===
首先我们先了解一下 log4cplus 中嵌入诊断上下文（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，呵呵。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）。 看了一下源代码，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();
```

当设定输出格式(Layout)为 TTCCLayout 时，输出如下：

```
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();

    ... ...
```

输出如下：

```
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。

=== 线程 ===

线程是 log4cplus 中的副产品， 而且仅作了最基本的实现，使用起来也异常简单，只要且必须要在派生类中重载 run 函数即可：

```
class TestThread : public AbstractThread
{
public:
    virtual void run();
};

void TestThread::run()
{
    /* do sth. */
    ... ...
}
```

log4cplus 的线程没有考虑同步、死锁，有互斥，实现线程切换的小函数挺别致的：

```
void log4cplus::thread::yield()
{
#if defined(LOG4CPLUS_USE_PTHREADS)
    ::sched_yield();
#elif defined(LOG4CPLUS_USE_WIN32_THREADS)
```

```
    ::Sleep(0);
#endif
}
```

=== 套接字 ===

套接字也是 log4cplus 中的副产品，在 namespace log4cplus::helpers 中，实现了 C/S 方式的日志记录。

1. 客户端程序需要做的工作：

```
/* 定义一个 SocketAppender 类型的挂接器 */
SharedAppenderPtr _append(new SocketAppender(host, 8888, "ServerName"));
/* 把_append 加入到 logger 中 */
Logger::getRoot().addAppender(_append);
/*  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】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);

    ... ...
```

2. 服务器端程序需要做的工作：

```
/* 定义一个 ServerSocket */
ServerSocket serverSocket(port);

/* 调用 accept 函数创建一个新的 socket 与客户端连接 */
Socket sock = serverSocket.accept();
```

此后即可用该 sock 进行数据 read/write 了,形如：

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

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

【注】 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。

```
/*
	严格实现步骤1-6，appender输出到屏幕，其中的布局格式和LogLevel后面会详细解释。

*/
#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 _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(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;
}
```

```
/*
    严格实现步骤1-6，appender输出到屏幕，其中的布局格式和LogLevel后面会详细解释。
*/
#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 _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(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;
}
```

封装 Log4cplus 后不能记录行号的问题

在使用 Log4cplus 的时候，在配置文件中配置了%l，就是记录日志信息所在的文件和行号。我们可以直接 LOG4CPLUS_ERROR(logger,LogMsg);来进行日志记录了。这样日志信息里面就会有我们日志信息所在文件和行号。

但是有的时候我们可能需要对 log4cplus 进行封装，封装后的代码可能会像下面的样子：

```
void _LogError(const char* fmt, ... )
{
    char LogMsg[4096];
    int strlen;
    memset(LogMsg,0x00,sizeof(LogMsg));

    va_list ap;
    va_start(ap,  fmt);
    strlen = vsprintf(LogMsg, fmt, ap);
    va_end(ap);
    LOG4CPLUS_ERROR(logger,LogMsg);
    return ;

}
```

但是这样的话，日志信息记录的文件名和行号就成了 _LogError 所在的文件和 LOG4CPLUS_ERROR(logger,LogMsg);行的行号。这样就达不到我们想要的目的了。

```
2010-01-06 00:52:30 [test.cpp:47]->
============================================================
2010-01-06 00:52:30 [test.cpp:47]-> XXXXX 日志
2010-01-06 00:52:30 [test.cpp:47]->
============================================================
2010-01-06 00:52:30 [test.cpp:47]-> 程序开始启动
```

想要记录到真正日志信息所在的文件和行，当然有办法：

我们可以定义两个全局变量：

```
char  *filename;
int  line;
```

然后定义一个宏：

```
#define  LogError  filename=__FILE__,line=__LINE__,_LogError
```

再把 _LogError 函数中的 LOG4CPLUS_ERROR(logger,LogMsg); 改成 logger.log(ERROR_LOG_LEVEL,LogMsg,filename,line); 这样就可以了。

```
2010-01-06 00:52:43 [test.cpp:60]->
============================================================
2010-01-06 00:52:43 [test.cpp:61]->      XXXXX 日志
2010-01-06 00:52:43 [test.cpp:62]->
============================================================
```

2010-01-06 00:52:43 [test.cpp:63]-> 程序开始启动

但是上述这种情况只适用于单线程。
如果要使用于多线程的环境中可以将 filename 和 line 实现为函数，就像 errno 在多线程的情况下的实现那样，其中要使用 thread specific 数据。

AIX 的 BUG 问题，无解

log4cplus BUG [AIX] Application seems to be locked on a mutex

Logging Framework for C++

[AIX] Application seems to be locked on a mutex - ID: 3056687
Last Update: Comment added (wilx)
Details:Hi all,

we encountered this situation not very often but only on AIX 5.3: the application seems to be lock when calling _global_lock_common

0xd010b824 _waitlock(??, ??) + 0x27c
0xd010bf28 _local_lock_common(??, ??, ??) + 0x138
0xd01195b8 _mutex_lock(??, ??, ??) + 0x258
0xdb07afa8 log4cplus::thread::Guard::Guard(pthread_mutex_t*)(0x315d49d8, 0x30215990) + 0x28
0xdb07b180 log4cplus::helpers::AppenderAttachableImpl::appendLoopOnAppenders(const log4cplus::spi::InternalLoggingEvent&) const(0x30215830, 0x315d4af0) + 0x48
0xdb2332e0 log4cplus::spi::LoggerImpl::callAppenders(const log4cplus::spi::InternalLoggingEvent&)(0x30225320, 0x315d4af0) + 0x68
0xdb231bd8 log4cplus::spi::LoggerImpl::forcedLog(int,const std::basic_string<char,std::char_traits<char>,std::allocator<char> >&,const char*,int)(0x30225320, 0x4e20, 0x315d4c10, 0xdc44ae28, 0x44) + 0xc0
0xdb068884 log4cplus::Logger::forcedLog(int,const std::basic_string<char,std::char_traits<char>,std::allocator<char> >&,const char*,int)(0x305d9f8c, 0x4e20, 0x315d4c10, 0xdc44ae28, 0x44) + 0x6c
...
This thread seems to be lock by :

0xd010c4c0 _global_lock_common(??, ??, ??) + 0x408
0xd0381610 _rec_mutex_lock(??) + 0x160
0xd01fb99c std::_Lock::_Wait()(??) + 0x5c
0xd01fb8dc std::_Lockit::_Lockit(int)(??, ??) + 0x1c
0xdb0945b4 std::basic_ostream<char,std::char_traits<char> >::sentry::sentry(std::basic_ostream<char,std::char_traits<char> >&)(0x5d3d1570, 0x3022c438) + 0x48
0xdb0dae74 std::basic_ostream<char,std::char_traits<char> >& std::operator<<<char,std::char_traits<char>,std::allocator<char> >(std::basic_ostream<char,std::char_traits<char> >&,const std::basic_string<char,std::char_traits<char>,std::allocator<char> >&)(0x3022c438, 0x5d3d1600) + 0xc0

0xdb1df660
log4cplus::pattern::PatternConverter::formatAndAppend(std::basic_ostream<char,std::char_traits<char> >&,const log4cplus::spi::InternalLoggingEvent&)(0x3022d230, 0x3022c438, 0x5d3d1940) + 0x208
0xdb1df7c0
log4cplus::PatternLayout::formatAndAppend(std::basic_ostream<char,std::char_traits<char> >&,const log4cplus::spi::InternalLoggingEvent&)(0x3022cd10, 0x3022c438, 0x5d3d1940) + 0x70
...

There are many other threads that are waiting for _global_lock_common.
The situation looks like the bug 2961084 , but no SIGNAL handling there.
I can't determine if the lock is due to log4cplus or AIX OS himself (or other).
Please, find in the attachment file, the whole stack.

Thanks you for the help,
Mikael Tintinger