iLOG3

User Guide

Revised version

version	date	author	content
1.0.0	2014-01-12	calvin	create
1.0.1	2014-02-03	calvin	add Log handle collection
1.0.2	2014-02-09	calvin	add The interface layer configuration file
1.0.3	2014-05-18	calvin	modify Custom log open, write, close add Subsequent modification log file name

indexes

1 Introduction

Recently I want to develop a similar nginx, need a log module. I had developed two log function library, the first is very simple, only five levels of log entry functions plus a written document, then is not to meet the needs of company platform log interface and development of the second edition, based on the popular "recorder - follower - layout" structure, also implements the external configuration files and log files, and so on advanced features, use for many years, no problem but feel too too bloated, then develop the third edition.

The search on internet a lot of topics about design log function library, sum up as the basic design principles for this topic :

- As the important foundation of software module, the log function library should try to realize the lightweight, simple and stable, on the premise of guarantee high reliability, high performance low impact is the primary design goals. Provide multiple API and optional way of external configuration file to use
- Basic function is indispensable, such as log level, line journal and hexadecimal block, etc., to a greater influence on the performance and ease of use of advanced features don't add do not add as much as possible. (do I want to realize turn log file is a very tangled problem, I always insist that use ops shell to implement will be better, but the same kind of logging library under the pressure, I am still achieved)
- Log module at least should be divided into two parts: core and external configuration, intermediate and together with the API is advantageous for the extension and flexible replacement
- Thread safety

After two nights of struggling to code and three night tinkering to this topic. Than I had expected a little "fullness", I have been trying to control the size, I tried my best.

2 Overview

iLOG3 is a portable and easy to use, simple concept, high performance, multiple interfaces, native cross-platform, thread safety , following the LGPL source protocol standard c log function library.

The basic features are as follows ：

• Native cross-platform, which means that your software on the log module is easily transplanted, currently supported WINDOWS & UNIX & Linux, iLOG3 will accordingly implementation and optimization on different operating systems

• Five log levels

• Variable parameter of the journal functions and macros

• Log style scheme selection

• Output media file, the standard output and standard error output, syslogd or WINDOWS EVENT, custom medium

Advanced features as follows ：

• Advanced log options

• According to turn archives by log file size, every day, every hour

• Support line logging style custom callback function, it is easy to customize your own line of log format

•Support of the log file open, output, close the custom callback function, it is easy to extend into the log output to a remote server

• Thread safety, simple MDC, based on the default thread local storage global logging handle

Layered implementation "**LOG handle layer (LOG) - > LOG handle collection (LOGS) - > config file interface layer (LOGCONF, LOGSCONF)**". Actually most users log demand is very simple, a process to write a log file (using the logging handle layer functions), but also consider the other users have more than one output object demand (using the logging handle collection layer functions), and users tend to use external configuration file to configure the log (using a configuration file interface layer functions), different users in different project scene using iLOG3 different layers of the interface.

I also developed a sister function library iLOG3CONF_SML to support with SML markup language (XML) configuration file to configure the logging handle, interested friends can call log handle layer or log handle collection function to develop their own iLOG3CONF_ *, implementation in XML or json or in fashion now own project unified configuration file format, to achieve with an external configuration file configuration iLOG3 log handle to the environment.

In addition, the source code structure is relatively simple, only three pair of the source file, easy to handle, embedding and modification. (source behind analysis chapter to help readers understand and fork own version)

3 Compiler install

3.1 Dependency

Before installation must know iLOG3 rely on library environment. As far as I know, so long as has the basic c compiler environment can, if there is a pthread (UNIX and Linux) to support thread safety, if c99 compilation, can support several writing log macro convenient user use, even if the above all have no, don't hinder compilation and basic use, just use less function.

3.2 With a makefile or VC6 project file compiler install

To get the source code package, in the appropriate directory and enter

```
$ tar xvzf iLOG3-x.x.x.tar.gz
$ cd iLOG3-x.x.x
```

iLOG3 do decline automake, the reason is automake complex to enough I learn a semester, so I wrote a similar automake named MKTPL, all iLOG3 makefile by MKTPL automatically generate the corresponding operating system makefile, user corresponding use.

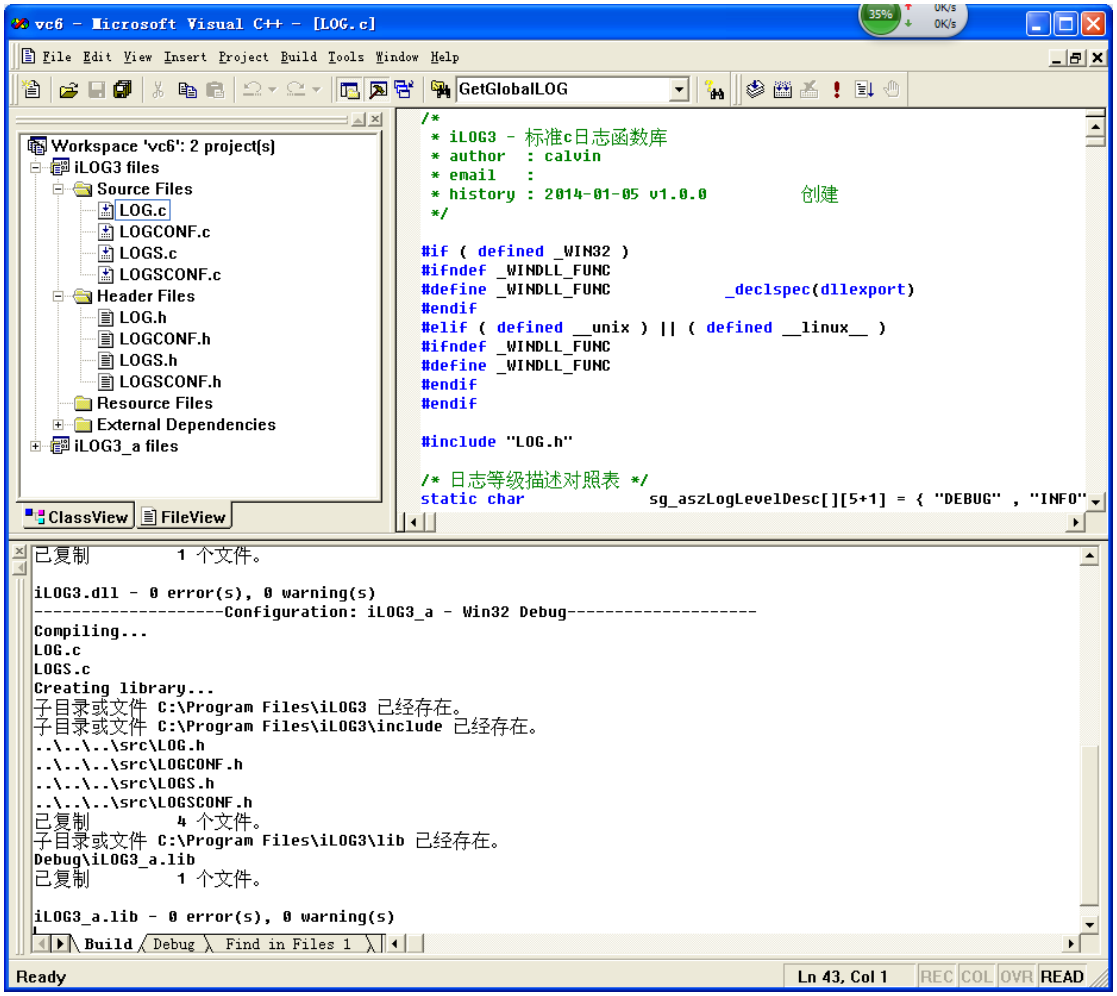
If the target environment is Linux, open the makefile.Linux in the SRC directory, modify the HDERINST pointing in the header file installation directory (usually /usr/local/include), modify the LIBINST and LIBINST2 pointing in the direction of the library files installation

directory (usually /usr/local/lib).

Then use a makefile.Linux compiled links installation, goes well in the following output said the installation is complete

```
$ make -f makefile.Linux clean
rm -f LOG.o
rm -f LOGS.o
rm -f LOGCONF.o
rm -f LOGSCONF.o
rm -f libiLOG3.so
rm -f libiLOG3.a
$ make -f makefile.Linux
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOG.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGS.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGCONF.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGSCONF.c
gcc -g -fPIC -O2 -std=c99 -o libiLOG3.so LOG.o LOGS.o LOGCONF.o LOGSCONF.o -shared -L. -lpthread
ar rv libiLOG3.a LOG.o LOGS.o LOGCONF.o LOGSCONF.o
ar: creating libiLOG3.a
a - LOG.o
a - LOGS.o
a - LOGCONF.o
a - LOGSCONF.o
$ sudo make -f makefile.Linux install
make[1]: Entering directory `/home/calvin/exsrc/iLOG3-1.0.0/src'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/calvin/exsrc/iLOG3-1.0.0/src'
cp -f libiLOG3.so /usr/local/lib/
cp -f LOG.h /usr/local/include/
cp -f LOGS.h /usr/local/include/
cp -f LOGCONF.h /usr/local/include/
cp -f LOGSCONF.h /usr/local/include/
```

WINDOWS have two compile installation mode, the makefile.MinGW and VC6 project file. With VC6 compiler installed please big menu "iLOG3 - X.X.X \ SRC \ VC6 \ VC6 DSW," big menu "Project" - > "Settings..." Post - Build step "- >", set up the installation target directory, the final blow "Build" - > "Rebuild All", compile the Output window appears at the bottom of the results the following Output



Install the target path is as follows：

```
C:\Program Files\iLOG3
├──bin
|       iLOG3.dll
|
├──include
|       LOG.h
|       LOGS.h
|       LOGCONF.h
|       LOGSCONF.h
|
└──lib
        iLOG3.lib
        iLOG3_a.lib
```

Helpful hints: adding iLOG3 DLL directory in WINDOWS environment variable PATH.

At this point, iLOG3 compiler installed in your system.

3.3 Manually compile installation

If you want to manually compile installation, or do you want to be automake, very simple, remember as long as making the LOG.C, LOGS.C LOGCONF.c LOGSCONF.c into libiLOG3. (or libiLOG3.a), in Linux environment dynamic link library, for example

```
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOG.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGS.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGCONF.c
gcc -g -fPIC -O2 -std=c99 -Wall -Werror -I. -c LOGSCONF.c
gcc -g -fPIC -O2 -std=c99 -o libiLOG3.so LOG.o LOGS.o LOGCONF.o LOGSCONF.o -shared -L. -lpthread
```

If you don't need to configure file interface layer, the pursuit of small and exquisite (especially in embedded environment), to remove LOGCONF.C and LOGSCONF.C.

If you only have to write a LOG file, do not need to LOG handle collection layer, remove LOGS.C, only compile the LOG.c file.

Do not add compiler options -STD = c99 is mainly a batch of variable parameter less writing log macros. Do not add link options - lpthread (header file at the same time remove the "# include <pthread.h>") less thread safety and based on the default thread local storage global log handle. Again, don't interfere with compilation and basic usage.

Reproduction libiLOG3. So to library file directory, copy *.h at the end of the file directory. Done!

```
cp -f libiLOG3.so /usr/local/lib/
cp -f *.h /usr/local/include/
```

4 Basic use

4.1 A little concept

Let's start from log handle layers, built the reengineering on the second floor on the ground floor.

Like standard file operation FILE * fp, **write a journal must create a handle, set a few of logging handle properties, and then keep writing log... Final destruction log handle**. A log handle corresponding to a log output object (file or standard output, or other), corresponds to a log filter rating, corresponds to a line of log style. Above all is the need to master before use concept, is very simple?

Someone ask me How to write two log files at the same time? Answer: create two log handle, straightforward, simple is beauty, without classification, there is no recorder output without layout, there is no inheritance, all completely "private custom". Ok, I admit that I am controlled by original light simple design rules, in fact I very want to add a lot of cool concept, but the thought of writing your own more than one thousand lines of similar nginx core will tremble with more than five thousand lines of log module.

Log handle collection can be achieved at the same time have more than one output object (or file), see a later chapter.

Useless talk so far, the examples

4.2 The first sample

All examples taken from the source code package's own test routine test/test_*.c

```
#include <stdio.h>

#include "LOG.h"

#define LOG_STYLES_HELLO    ( LOG_STYLE_DATETIMEMS | LOG_STYLE_LOGLEVEL | LOG_STYLE_PID | LOG_STYLE_TID | LOG_STYLE_SOURCE | LOG_STYLE_FORMAT | LOG_STYLE_NEWLINE )

int test_hello()
{
	LOG		*g = NULL ; /* Log handle pointer */

	char		buffer[ 64 + 1 ] = "" ;
	long		buflen = 64 ;

	/* Create log handle */
	g = CreateLogHandle() ;
	if( g == NULL )
```

```
	{
		printf( "Failed to create logging environment ,errno[%d]\n" , errno );
		return -1;
	}
	else
	{
		printf( " Create log environment successfully \n" );
	}

	/* Set the log output file name */
	SetLogOutput( g , LOG_OUTPUT_FILE , "test_hello.log" , LOG_NO_OUTPUTFUNC );
	/* Set the current log filter level */
	SetLogLevel( g , LOG_LEVEL_INFO );
	/* Set the current line of log style */
	SetLogStyles( g , LOG_STYLES_HELLO , LOG_NO_STYLEFUNC);

	/* In a different log level   write the log line */
	WriteDebugLog( g , __FILE__ , __LINE__ , "hello iLOG3" ); /* This line of log for rating is not enough, was magnificent
filter */
	WriteInfoLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteWarnLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteErrorLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteFatalLog( g , __FILE__ , __LINE__ , "hello iLOG3" );

	/* In a different log level   write the log block */
	WriteDebugHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "buflen[%ld]" , buflen ); /* And a filter */
	WriteInfoHexLog( g , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteWarnHexLog( g , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteErrorHexLog( g , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteFatalHexLog( g , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );

	/* Destruction of log handle */
	DestroyLogHandle( g );
	printf( " Release the logging handle \n" );

	return 0;
}

int main()
{
	return -test_hello();
}
```

If your compiler support C99 (etc. add '-std=gnu99' for gcc) , use macro to output log simplely

```
	/* In a different log level  write the log line */
	DEBUGLOG( g , "hello iLOG3" ); /* 这行日志因等级不够，被华丽的过滤了 */
	INFOLOG( g , "hello iLOG3" );
	WARNLOG( g , "hello iLOG3" );
	ERRORLOG( g , "hello iLOG3" );
	FATALLOG( g , "hello iLOG3" );

	/* In a different log level  write the log block */
	DEBUGHEXLOG( g , buffer , buflen , "缓冲区[%ld]" , buflen ); /* 又一个被华丽的过滤 */
	INFOHEXLOG( g , buffer , buflen , "缓冲区[%ld]" , buflen );
	WARNHEXLOG( g , buffer , buflen , "缓冲区[%ld]" , buflen );
	ERRORHEXLOG( g , buffer , buflen , "缓冲区[%ld]" , buflen );
	FATALHEXLOG( g , buffer , buflen , "缓冲区[%ld]" , buflen );
```

This example first create log handle, set the log handle environment such as setting the log file name, and then write the log, finally destroyed the log handle.

Compile successfully link operation, produce the log files are as follows: (note: word typesetting dislocation)

```
2014-01-14 21:01:09 | INFO  | test_hello.c:50 | hello iLOG3
2014-01-14 21:01:09 | WARN  | test_hello.c:51 | hello iLOG3
2014-01-14 21:01:09 | ERROR | test_hello.c:52 | hello iLOG3
2014-01-14 21:01:09 | FATAL | test_hello.c:53 | hello iLOG3
2014-01-14 21:01:09 | INFO  | test_hello.c:56 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0123456789ABCDEF
0x00000000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
2014-01-14 21:01:09 | WARN  | test_hello.c:57 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0123456789ABCDEF
0x00000000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
2014-01-14 21:01:09 | ERROR | test_hello.c:58 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0123456789ABCDEF
0x00000000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00000030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
2014-01-14 21:01:09 | FATAL | test_hello.c:59 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0123456789ABCDEF
0x00000000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
0x00000010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

Very simple, isn't it? Congratulations you have learned to use iLOG3 half of the entire contents of the log function library, really, believe me.

Tip: a memory log handle type for the log, the corresponding header file the LOG.h, log handle collection types for LOGS, corresponding to the header file LOGS.h, as the name implies, LOGS is multiple log collection.

4.3 Log style scheme selection

May be you think log style are too simple, contain too little, for example, if you still want to join the milliseconds to see you develop the high performance of the application server time consuming, you still want to be a part of the process id, oh, and thread id... Ok, so far, I can only provide these at the moment.

Modify the line of log style scheme macros

```
#define LOG_STYLES_HELLO   ( LOG_STYLE_DATETIMEMS | LOG_STYLE_LOGLEVEL | LOG_STYLE_PID | LOG_STYLE_TID | LOG_STYLE_SOURCE | LOG_STYLE_FORMAT | LOG_STYLE_NEWLINE )
```

Compile links and run again

```
2014-01-14 21:08:06.468000 | INFO   | 3148:3596:test_hello.c:50 | hello iLOG3
2014-01-14 21:08:06.484000 | WARN   | 3148:3596:test_hello.c:51 | hello iLOG3
2014-01-14 21:08:06.484000 | ERROR | 3148:3596:test_hello.c:52 | hello iLOG3
2014-01-14 21:08:06.484000 | FATAL | 3148:3596:test_hello.c:53 | hello iLOG3
2014-01-14 21:08:06.484000 | INFO   | 3148:3596:test_hello.c:56 | buflen[64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0123456789ABCDEF
0x00000000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
2014-01-14 21:08:06.484000 | WARN   | 3148:3596:test_hello.c:57 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0123456789ABCDEF
0x00000000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
2014-01-14 21:08:06.484000 | ERROR | 3148:3596:test_hello.c:58 | buflen [64]
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0123456789ABCDEF
```

```
0x00000000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
2014-01-14 21:08:06.484000 | FATAL | 3148:3596:test_hello.c:59 | buflen [64]
              0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0123456789ABCDEF
0x00000000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

A complete line of log style combination macro list in 《iLOG3 Reference Manual.doc》, are interested can go and see.

5 Advanced features

5.1 Set the log options

iLOG3 log settings option on here, and **must be in front of the other handle set function calls**.

Log output mode of macro decision opened and closed to the timing of the log, after opening not to close to improve output performance but when the file will remove or delete some small at odds, opening and closing every time log safe but the performance is low, detect file changes is used to balance reliability and performance requirements.

LOG_OPTION_OPEN_AND_CLOSE	Every time open the log, write the log, close the log
LOG_OPTION_CHANGE_TEST	Automatic detect file changes (default setting)
LOG_OPTION_OPEN_ONCE	Log opened not to close

There are some interesting options

LOG_OPTION_FILENAME_SET_OUTPUT	Depending on the type of output filename reset automatically
LOG_OPTION_FILENAME_APPEND_DOT_LOG	Log output filename automatically after plus ".log"

```
    SetLogOptions( g2 , LOG_OPTION_SET_OUTPUT_BY_FILENAME );
    SetLogOutput( g2 , LOG_OUTPUT_FILE , "$TEST_FUNNY_LOG_OUTPUT$" , LOG_NO_OUTPUTFUNC );
```

```
int main( int argc , char *argv[] )
{
```

```
	...
	SetLogOptions( g , LOG_OPTION_FILENAME_APPEND_DOT_LOG );
	SetLogOutput( g , LOG_OUTPUT_FILE , argv[0] , LOG_NO_OUTPUTFUNC );
	...
}
```

5.2 Log file rolling

Log file rolling profile is a big topic. ILOG3 log function library provides three turn archives solution for everybody to choose:

- According to the log file size turn archives
- According to the days turn archives
- According to the hours turn archives

The function SetLogRotateMode is used to set the turn archives scheme

```
int SetLogRotateMode( LOG *g , int rotate_mode );
```

The mode has the following values:

```
#define LOG_ROTATEMODE_NONE		0 /* no rolling */
#define LOG_ROTATEMODE_SIZE		1 /* According to the log file size turn archives */
#define LOG_ROTATEMODE_PER_DAY	2 /* According to the days turn archives */
#define LOG_ROTATEMODE_PER_HOUR	3 /* According to the hours turn archives */
```

When set to turn according to the log file size, you should also call a function SetLogRotateSize settings file size, the actual rolling file size is slightly bigger than the value. Due to the size of turning gear optimization algorithm needs a coefficient, you can call the function SetLogRotatePressureFactor set the coefficient is about "process count * thread count * 2", of course, you don't set up also doesn't matter much, actually most turn archives file write too. Round of gears, according to the log file size after format for " filename. serial_number" to line up. The code examples are as follows:

```
SetLogRotateMode(g, LOG_ROTATEMODE_SIZE );
SetLogRotateSize( g , 10*1024*1024 );
SetLogRotatePressureFactor(g, 20 );
```

Turn archives according to the day when turning gear after the name of the file format for " filename. YYYYMMDD". The code examples are as follows:

```
SetLogRotateMode( g , LOG_ROTATEMODE_PER_DAY );
```

Turn archives by the hour, turning gear after the name of the file format for

"filename.YYYYMMDD_HH". The code examples are as follows:

```
SetLogRotateMode(g, LOG_ROTATEMODE_PER_HOUR );
```

5.3 Custom log style

I always feel that someone will question iLOG3 line logging style too scheme is changed and the lack of a fully custom flexibility. Well, I will give you set a callback function, you can set line log style what you love to set up.

The handle to set the log environment settings to join in your code

```
SetLogStyles( g , LOG_STYLE_CALLBACK , & MyLogStyleFunc );
```

Write a callback function is as follows:

```
funLogStyleFunc MyLogStyleFunc ;
int MyLogStyleFunc( LOG *g , char *c_filename , long c_fileline , int log_level , char *format , va_list valist )
{
	long		tt ;
	struct tm	stime ;
	long		len ;

	time( & tt );
	LOCALTIME( tt , stime )

	len = SNPRINTF( g->log_bufptr , g->log_bufremainlen , "%02d%02d %02d%02d%02d " , stime.tm_mon+1 ,
stime.tm_mday , stime.tm_hour , stime.tm_min , stime.tm_sec ) ;
OFFSET_BUFPTR(g->log_bufptr,p->log_bufremainlen,len)
	len = VSNPRINTF( p->log_bufptr , p->log_bufremainlen , format , valist ) ;
OFFSET_BUFPTR(p->log_bufptr,p->log_bufremainlen,len)
	len = SNPRINTF( p->log_bufptr , p->log_bufremainlen , "%s" , LOG_NEWLINE ) ;
	OFFSET_BUFPTR(p->log_bufptr,p->log_bufremainlen,len)

	return 0;
}
```

Compile and link and run, get a log like this:

```
0114 231519 DEBUG hello iLOG3
0114 231519 INFO hello iLOG3
0114 231519 WARN hello iLOG3
0114 231519 ERROR hello iLOG3
0114 231519 FATAL hello iLOG3
```

5.4 Custom log output

iLOG3 output common object is file, you can join absolute or relative path in filename , may also directly to join environment variables, such as

```
SetLogOutput( g , LOG_OUTPUT_FILE , ”$HOME$/log/myoutput.log” , LOG_NO_OUTPUTFUNC );
```

iLOG3 log output object is not only a file, also supports standard output and standard error output, syslog or WINDOWS EVENT and output type, such as custom log output function.

Open the log output is set when for LOG_OUTPUT_STDOUT macro can make the log output to the screen, the standard error output in the same way

```
SetLogOutput( g , LOG_OUTPUT_STDOUT , NULL , LOG_NO_OUTPUTFUNC );
```

LOG_OUTPUT_SYSLOG macro on UNIX&Linux representing the output to the syslog, represents a WINDOWS in WINDOWS EVENT, the code is as follows:

```
SetLogOutput( g , LOG_OUTPUT_SYSLOG , “test” , LOG_NO_OUTPUTFUNC );
```

Custom log output function to see the next section

5.5 Custom log open, output, close

If as a local log output, iLOG3 finished the task perfectly, but I'm under a lot of ambition, I am going to put iLOG3 into can be used as a remote logs over landing local messaging clients, I once again in order to meet their own lusts shameless added hundreds of lines of code.

Here is the code examples, to set the log output function call to modify as such:

```
SetLogOptions( press , LOG_OPTION_OPEN_ONCE ); /* In front of SetLogOutput */
SetLogOutput( g , LOG_OUTPUT_CALLBACK , "127.0.0.1:514" , & MyOpenLogFirst , NULL , & MyWriteLog , NULL , NULL , &
MyCloseLogFinally );
```

Write a callback function as follows, here I use ‘printf’ instead of a hypothetical data connection, communication and disconnects

```
funcOpenLog MyOpenLogFirst ;
int MyOpenLogFirst( LOG *g , char *log_pathfilename , void **open_handle )
{
	if(IsLogOpened(g) == 1 )
		return 0;

	printf( "MyOpenLogFirst[%s]\n" , log_pathfilename );
```

```
	SetOpenFlag(g,1);
	return 0;
}

funcWriteLogBuffer MyWriteLog ;
long MyWriteLog (LOG *g , void **open_handle , int log_level , char *buf , long len )
{
	char   *p_log_level_desc = NULL ;

	if(IsLogOpened(g) == 0 )
		return 0;

	ConvertLogLevel_itoa( log_level , & p_log_level_desc );
	printf( "MyWriteLog [%s][%ld][%s]\n" , p_log_level_desc , len , buf );

	return len;
}

funcCloseLog MyCloseLogFinally ;
int MyCloseLogFinally( LOG *g , void **open_handle )
{
	if(IsLogOpened(g) == 0 )
		return 0;

	printf( "MyCloseLogFinally\n" );

	SetOpenFlag(g,0);
	return 0;
}
```

Compile and link and run, show as follows:

```
MyOpenLogFirst [127.0.0.1:514]
MyWriteLogBuffer[INFO][77][2014-01-14 23:34:32.546000 | INFO    | 2280:3640:test_openfunc.c:64 | INFO-DOG
]
MyWriteLogBuffer[WARN][77][2014-01-14 23:34:32.546000 | WARN    | 2280:3640:test_openfunc.c:65 | WARN-DOG
]
MyWriteLogBuffer[ERROR][78][2014-01-14 23:34:32.546000 | ERROR | 2280:3640:test_openfunc.c:66 | ERROR-DOG
]
MyWriteLogBuffer[FATAL][78][2014-01-14 23:34:32.546000 | FATAL | 2280:3640:test_openfunc.c:67 | FATAL-DOG
]
MyCloseLogFinally
```

Here only demonstrate a framework, specific details filled by you.

5.6 Subsequent modification log file name

Either call a function or by configuration, application always has the demand of the subsequent amendment log file name, such as iterative communication model hope every child thread to write different log file, then by changes after the child thread is created out of the current output log file name, without having to reopen the log handle.

Using the log output type macro LOG_OUTPUT_NOSET to call a function SetLogOutput can directly modify the log file name in order to improve the operation efficiency.

```
char  pathfilename[ MAXLEN_FILENAME + 1 ] ;
memset( pathfilename , 0x00 , sizeof(pathfilename) );
snprintf( pathfilename , sizeof(pathfilename)-1 , "$HOME$/log/thread%ld.log" , (unsigned long)pthread_self() );
SetLogOutput( g , LOG_OUTPUT_NOSET , pathfilename , LOG_NO_OUTPUTFUNC );
```

5.7 Thread-safe

If you can't do a log function library thread-safe that would not have been a general log function library.

If you develop multi-processes without considering thread safety, simple and crude directly handle the log statement into a global variable, if it is multi-threaded is a thread safety issues to consider.

iLOG3 thread-safe code means there are at least three, one is to use global log handle + realize the mutex mechanism, a second look at the next paragraph.

First of all, don't call log function library thread unsafe library function, I have been looking for, find a used thread-safe version to replace it. Then, storage, if you want to do a variable in the isolation between threads, the simplest approach is to declare a local variable. you declare a local LOG structure pointer, automatic calling function CreateLogHandle back to it to create log handle structure, it is thread safe.

But such use will bring some trouble to the application code, if your environment support thread local storage (TLS), congratulations you, you can enable global log handle based on the thread local storage to quarantine automatically for you, I have a default TLS global log handle is used to simplify the code, specific see a later chapter.

5.7.1 MDC

Taboo word, it is your custom some string value for formatting to the line in the log, but also to do not affect between threads, as long as the string into the handle to the log has been isolated thread, the thread is isolated.

Add code, in the past will handle iLOG3 medium voltage into the label "COREIB1_SERVICE" into the cell of index number one

```
SetLogCustLabel(g, 1 , "COREIB1_SERVICE" );
```

Then log style in the combination of macro index in a tag configuration

```
#define LOG_STYLES_EVENT    ( LOG_STYLE_DATETIME | LOG_STYLE_CUSTLABEL1 | LOG_STYLE_FORMAT |
LOG_STYLE_NEWLINE )
```

Then log in line handle event log output "COREIB1_SERVICE" section

```
2014-01-14 21:52:52 | COREIB1_SERVICE | hello iLOG3
```

Tags can be set up to 5 most unless you have modified the macro and made up the log function library source code

```
#define LOG_MAXCNT_CUST_LABEL          5
```

Label value 64 characters in length, unless you have modified the macro and made up the log function library source code

```
#define LOG_MAXLEN_CUST_LABEL          64
```

Are you sure you really need so many labels?

The next section is exciting...

5.7.2 Based on the default thread local storage global log handle

Said before the log handle thread-safe code one way is to use a local variable to achieve, do you think it's no technical content, the technical content feed you to the full. Now open your commonly used search engine, type in key words "thread TLS", press ‘enter’, to learn what is called "thread local storage".

I assume you have enough, now we come back to see iLOG3. Use local variables to achieve

log handle thread-safe can bring some to the user, namely in front of the handle to create the log are passed back and forth in your major function, using the handle to the thread local storage can log statements into thread private global variables, could not understand? It is a global variable.

iLOG3 has a thread local storage of the default log handle, global variable name can't tell you, it's a secret, as long as you use the default based on the thread local storage global log handle version of the function and macro can dispense front talked about behind talk about all the functions of the first parameter to the log handle, as long as the function name finally have to do is add a "G", like this

```
CreateLogHandleG();
SetLogOutputG( LOG_OUTPUT_FILE , "test_hello.log");
SetLogLevelG( LOG_LEVEL_INFO );
SetLogStylesG( LOG_STYLES_HELLO , LOG_NO_STYLESFUNC );
DestroyLogHandleG();
```

Whether very worry, if you want to use their create log as a thread local storage log handle, after it was created using this function to replace the default

```
void SetGlobalLOG( LOG *g )
```

Helpful hints: thread local storage and not all compilers support, as far as I know MS compiler on the WINDOWS, Linux on the GCC compiler and AIX XLC compiler is no problem, the rest of the trial, please trouble to tell me, after I'll add in.

5.8 A common example

Most occasions application may wish to log handle can do global access, so convenient source structure, presented here is a more commonly used code examples:

```
#include <stdio.h>

#include “LOG.h”

#define LOG_STYLES_HELLO   ( LOG_STYLE_DATETIMEMS | LOG_STYLE_LOGLEVEL | LOG_STYLE_PID | LOG_STYLE_TID | LOG_STYLE_SOURCE | LOG_STYLE_FORMAT | LOG_STYLE_NEWLINE )

int test_hello()
{
	char		buffer[ 64 + 1 ] = "" ;
	long		buflen = 64 ;
```

```
	/* Create log handle */
	if(CreateLogHandleG() == NULL )
	{
		printf( "Failed to create logging environment ,errno[%d]\n" , errno );
		return -1;
	}
	else
	{
		printf( " Create log environment successfully \n" );
	}

	/* Set the log output file name */
	SetLogOutputG( LOG_OUTPUT_FILE , "test_hello.log" , LOG_NO_OUTPUTFUNC );
	/* Set the current log filter level */
	SetLogLevelG( LOG_LEVEL_INFO );
	/* Set the current line of log style */
	SetLogStylesG( LOG_STYLES_HELLO , LOG_NO_STYLEFUNC);

	/* In a different log level write the log line */
	WriteDebugLogG( __FILE__ , __LINE__ , "hello iLOG3" ); /* This line of log for rating is not enough, was magnificent
filter */
	WriteInfoLogG( __FILE__ , __LINE__ , "hello iLOG3" ); /* If you enable the c99 macro INFOLOGG can also use the log to
write logs, dispense a __FILE__ and __LINE__ two parameters */
	WriteWarnLogG( __FILE__ , __LINE__ , "hello iLOG3" );
	WriteErrorLogG( __FILE__ , __LINE__ , "hello iLOG3" );
	WriteFatalLogG( __FILE__ , __LINE__ , "hello iLOG3" );

	/* In a different log level write the log block */
	WriteDebugHexLogG( __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen ); /* And a filter */
	WriteInfoHexLogG( __FILE__ , __LINE__ , buffer , buflen , "buflen[%ld]" , buflen );
	WriteWarnHexLogG( __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteErrorHexLogG( __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteFatalHexLogG( __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );

	/* Destruction of log handle */
	DestroyLogHandleG();
	printf( " Release the logging handle \n" );

	return 0;
}

int main()
{
```

```
	return -test_hello();
}
```

Use macro to reduce codes

```
	/* In a different log level write the log line */
	DEBUGLOGG( "hello iLOG3" ); /* 这行日志因等级不够，被华丽的过滤了 */
	INFOLOGG( "hello iLOG3" ); /* 如果启用 c99 还可以使用日志宏 INFOLOGG 来写日志，省却__FILE__,__LINE__两个参数 */
	WARNLOGG( "hello iLOG3" );
	ERRORLOGG( "hello iLOG3" );
	FATALLOGG( "hello iLOG3" );

	/* In a different log level write the log block */
	DEBUGHEXLOGG( buffer , buflen , "缓冲区[%ld]" , buflen ); /* 又一个被华丽的过滤 */
	INFOHEXLOGG( buffer , buflen , "缓冲区[%ld]" , buflen );
	WARNHEXLOGG( buffer , buflen , "缓冲区[%ld]" , buflen );
	ERRORHEXLOGG( buffer , buflen , "缓冲区[%ld]" , buflen );
	FATALHEXLOGG( buffer , buflen , "缓冲区[%ld]" , buflen );
```

6 Log handle collection layer

Are all speaking in front of the instructions on the use of the log handle layer (LOG), on small projects or high performance enough to use in the scene, some users prefer for LOG, I so long ago, such as a process to deal with a deal to write process detailed LOG (dog.log), each transaction processing after the contracted write a line of events (event.log), write error log when an error (error.log), three LOG handle is needed here, I wrap them together as a handle to a LOG collection, the name is... LOGS, and at the same time to write the log function or macro name in the log into LOGS is writing Log handle collection of functions and macros, very easy to remember, isn't it?

The following example illustrates the use of log handle collection, involving a variety of method of use, hope you can understand.

```
#include <stdio.h>
#include <errno.h>

#include "LOGS.h"

#define LOG_STYLES_HELLO	( LOG_STYLE_DATETIMEMS | LOG_STYLE_LOGLEVEL | LOG_STYLE_PID | LOG_STYLE_TID |
```

```
LOG_STYLE_SOURCE | LOG_STYLE_FORMAT | LOG_STYLE_NEWLINE )

int test_logs()
{
	LOGS		*gs = NULL ;
	LOG		*g = NULL ;

	long		index ;
	char		*g_id = NULL ;

	char		buffer[ 64 + 1 ] = "" ;
	long		buflen = sizeof(buffer) - 1 ;

	int		nret ;

	gs = CreateLogsHandle() ;
	if( gs == NULL )
	{
		printf( "Failed to create log handle collection ,errno[%d]\n" , errno );
		return -1;
	}

	g = CreateLogHandle() ;
	if( g == NULL )
	{
		printf( "Failed to create log handle ,errno[%d]\n" , errno );
		return -1;
	}

	SetLogOutput( g , LOG_OUTPUT_FILE , "test_logs.log" , LOG_NO_OUTPUTFUNC );
	SetLogLevel( g , LOG_LEVEL_INFO );
	SetLogStyles( g , LOG_STYLES_HELLO , LOG_NO_STYLEFUNC );

	AddLogToLogs( gs , "FILE" , g );

	AddLogToLogs( gs , "STDERR" , CreateLogHandle() );

	g = GetLogFromLogs( gs , "STDERR" ) ;
	if( g == NULL )
	{
		printf( "Get the log handle failure ,errno[%d]\n" , errno );
		return -1;
	}
```

```
	SetLogOutput( g , LOG_OUTPUT_STDERR , "" , LOG_NO_OUTPUTFUNC );
	SetLogLevel( g , LOG_LEVEL_ERROR );
	SetLogStyles( g , LOG_STYLES_HELLO , LOG_NO_STYLEFUNC );

	printf( " Create log handle collection of success \n" );

	WriteDebugLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteInfoLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteWarnLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteErrorLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	WriteFatalLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );

	WriteDebugHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "buflen[%ld]" , buflen );
	WriteInfoHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteWarnHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteErrorHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );
	WriteFatalHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , " buflen [%ld]" , buflen );

	DestroyLogHandle( RemoveOutLogFromLogs( gs , "FILE" ) );
	printf( " Destruction of log handle \n" );

	DestroyLogsHandle( gs );
	printf( " Destruction of logging handle collection \n" );

	return 0;
}

int main()
{
	return -test_logs();
}
```

Log handle collection macro

```
#define WRITELOGS( _g_ , _log_level_ , ... )	WriteLevelLogs( _g_ , __FILE__ , __LINE__ , _log_level_ , __VA_ARGS__ );
#define DEBUGLOGS( _g_ , ... )			WriteDebugLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );
#define INFOLOGS( _g_ , ... )		WriteInfoLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );
#define NOTICELOGS( _g_ , ... )			WriteNoticeLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );
#define WARNLOGS( _g_ , ... )		WriteWarnLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );
#define ERRORLOGS( _g_ , ... )			WriteErrorLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );
#define FATALLOGS( _g_ , ... )		WriteFatalLogs( _g_ , __FILE__ , __LINE__ , __VA_ARGS__ );

#define WRITEHEXLOGS( _g_ , _log_level_ , _buf_ , _buf_size_ , ... )	WriteLevelHexLogs( _g_ , __FILE__ , __LINE__ ,
_log_level_ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define DEBUGHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )		WriteDebugHexLogs( _g_ , __FILE__ , __LINE__ ,
_buf_ , _buf_size_ , __VA_ARGS__ );
```

```
#define INFOHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )                WriteInfoHexLogs( _g_ , __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define NOTICEHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )                    WriteNoticeHexLogs( _g_ , __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define WARNHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )                WriteWarnHexLogs( _g_ , __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define ERRORHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )               WriteErrorHexLogs( _g_ , __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define FATALHEXLOGS( _g_ , _buf_ , _buf_size_ , ... )               WriteFatalHexLogs( _g_ , __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );

#define WRITELOGSG( _log_level_ , ... )     WriteLevelLogsG( __FILE__ , __LINE__ , _log_level_ , __VA_ARGS__ );
#define DEBUGLOGSG( ... )               WriteDebugLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );
#define INFOLOGSG( ... )                WriteInfoLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );
#define NOTICELOGSG( ... )              WriteNoticeLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );
#define WARNLOGSG( ... )                WriteWarnLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );
#define ERRORLOGSG( ... )               WriteErrorLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );
#define FATALLOGSG( ... )               WriteFatalLogsG( __FILE__ , __LINE__ , __VA_ARGS__ );

#define WRITEHEXLOGSG( _log_level_ , _buf_ , _buf_size_ , ... )      WriteLevelHexLogsG( __FILE__ , __LINE__ , _log_level_ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define DEBUGHEXLOGSG( _buf_ , _buf_size_ , ... )               WriteDebugHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define INFOHEXLOGSG( _buf_ , _buf_size_ , ... )         WriteInfoHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define NOTICEHEXLOGSG( _buf_ , _buf_size_ , ... )              WriteNoticeHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define WARNHEXLOGSG( _buf_ , _buf_size_ , ... )                WriteWarnHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define ERRORHEXLOGSG( _buf_ , _buf_size_ , ... )               WriteErrorHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
#define FATALHEXLOGSG( _buf_ , _buf_size_ , ... )               WriteFatalHexLogsG( __FILE__ , __LINE__ , _buf_ , _buf_size_ , __VA_ARGS__ );
```

7 The configuration file interface layer

7.1 The interface layer with simple configuration file

configuration file interface layer implements the simple log configuration format, ok, no more simple than it, look

test_logconf.conf

```
output          FILE   test_logconf.log
level           INFO
styles          DATETIME|LOGLEVEL|PID|TID|SOURCE|FORMAT|NEWLINE
options         CHANGE_TEST
rotate_mode     SIZE
rotate_size     10MB
log_bufsize     1MB 5MB
```

Writing in the code

```
#include "LOGCONF.h"
...
g = CreateLogHandleFromConfig( "test_logconf.conf" , NULL ) ;
...
```

Build log from the config file handle, instead of calling a function to build, is very convenient.

Below is the log handle collection configuration

test_logsconf.conf

```
id              hello
output          FILE   test_logconf.log
level           INFO
styles          DATETIME|LOGLEVEL|PID|TID|SOURCE|FORMAT|NEWLINE
options         CHANGE_TEST
rotate_mode     SIZE
rotate_size     10MB
log_bufsize     1MB    5MB

id              stdout
output          STDOUT
level           INFO
styles          DATETIME|LOGLEVEL|PID|TID|SOURCE|FORMAT|NEWLINE
```

With a blank line between each configuration block, block the beginning more than a property id corresponding log in log handle handle g_id in the collection.

```
#include "LOGSCONF.h"
...
gs = CreateLogsHandleFromConfig( "test_logsconf.conf" , NULL ) ;
...
```

Is more than a 's', you understand?

A tip

```
int test_logsconf( char *program )
```

```
{
	...
#if ( defined _WIN32 )
	if( strstr( program , ".exe" ) )
	{
		strstr( program , ".exe" )[0] = '\0' ;
	}
#endif
	gs = CreateLogsHandleFromConfig( program , ".conf" ) ;
	...
}

int main( int argc , char *argv[] )
{
	return -test_logsconf( argv[0] );
}
```

Another trick

```
output		FILE		""
```

```
g = CreateLogHandleFromConfig( "test_logconf.conf" , NULL ) ;
SetLogOutput( g , LOG_OUTPUT_NOSET , "test_logconf_1.log" , LOG_NO_OUTPUTFUNC );
```

7.2 Looking forward

Configuration file interface layer just provides the most basic configuration mechanism, at the same time also shows how to parse a format configuration using the configuration auxiliary function after transformation and build log handle to the environment. I later in the development of library iLOG3CONF_SML realized with SML language as a senior iLOG3 configuration file format to use, but also the supporting configuration embedded, in my fortress nginx (called hetao) master configuration was the configuration log environment

```
hetao
{
	logs
	{
		log ( id = access )
		{
			output = FILE ;
```

```
            filename = "$HOME$/log/access.log" ;
            level = DEBUG ;
            styles = "DATETIME|LOGLEVEL|PID|TID|FORMAT|NEWLINE" ;
            rotate_mode = SIZE ;
            rotate_size = 10MB ;
        }

        log ( id = error )
        {
            output = FILE ;
            filename = "$HOME$/log/error.log" ;
            level = ERROR ;
            styles = "DATETIME|LOGLEVEL|SOURCE|FORMAT|NEWLINE" ;
            rotate_mode = SIZE ;
            rotate_size = 10MB ;
        }
    }

    module
    {
        connection ( filename = "$HOME$/so/hetao_module_connection_epoll.so" )
        {
        }

        protocol ( filename = "$HOME$/so/hetao_module_protocol_http.so" )
        {
            max_header_size = 10KB ;

            root = "$HOME$/www" ;
            index_page = "index.html" ;
            error_pages
            {
                error_page { error = 403 ; page = "$HOME$/www/403.html" } ;
                error_page { error = 404 ; page = "$HOME$/www/404.html" } ;
                error_page { error = 500 ; page = "$HOME$/www/500.html" } ;
                error_page { error = 505 ; page = "$HOME$/www/505.html" } ;
            }
        }
    }

    server
    {
        listen_ip = "127.0.0.1" ;
        listen_port = 8080 ;
```

```
            max_connections = 1024 ;
        }
}
```

Code so that access to configure the build log handle collection

```
gs = CreateLogsHandleFromConfig_SML( "hetao.conf" , "/hetao/logs" , NULL ) ;
```

Then can brush brush to write access.. access.log, error.log, very bright ^_^

8 If you prefer simple log function to log function library

iLOG3 comes with independent simple log function (iLOG3 lite version) for you to use, just copy src/LOGC.h, SRC/LOGC.c to your project directory to use directly.

There are many projects are using this smart mode, such as tcpdaemon.