iLOG3 日志函数库

用户指南

（如打开本文档后文档结构图混乱，可在打开文档时连续按 ESC 键）

版本修订

版本	日期	修订人	内容
1.0.0	2014-01-12	calvin	创建
1.0.1	2014-02-03	calvin	新增 日志句柄集合相关章节
1.0.2	2014-02-09	calvin	新增 配置文件接口层相关章节

目录索引

1 前言

最近想开发一个类似 nginx 的东东，需要一个日志模块。自己曾经开发过两个日志函数库，第一个很简单，只有五个日志等级入口函数外加一个写文件函数，后来实在不适应公司平台日志接口需求就又开发了第二版，基于流行的“记录器-输出器-布局器”结构，还实现了外部配置文件和日志转档等等高级功能，使用了很多年，虽说没出什么问题但感觉过于太臃肿，痛定思痛，重新开发第三版。

网上搜了很多关于设计日志函数库的话题，总结一下作为此版的基本设计准则：

• 作为软件的重要基础模块，日志函数库应尽量实现的轻便、简易和稳定，在保证高可靠性的前提下，高性能低影响是首要设计目标。提供多层 API 以及可选的外部配置文件方式使用。

• 基本功能必不可少，如日志等级、行日志和十六进制块日志等，对性能和易用性影响较大的高级功能可不加尽量不加。（要不要实现日志转档是一个很纠结的问题，我一直坚持认为用运维 shell 来实现会更好，但迫于同类日志库的压力，我还是实现了）

• 日志模块至少应分为两部分：核心和外部配置，中间用 API 胶合起来，便于扩展和灵活替换

• 线程安全

经过两个晚上的奋力疾书和三个晚上的修修补补终成此版。比我预期的稍稍“丰满”一些，我已经在极力控制规模了，我尽力了。

2 概述

iLOG3 是一个轻便易用、概念简单，高性能、多层接口、原生跨平台、（规范使用时）线程安全的遵循 LGPL 开源协议的标准 c 日志函数库。

基本特性如下：

• 原生跨平台，这意味着你的软件在日志层面上是可轻松移植的，目前支持 WINDOWS & UNIX & Linux ， iLOG3 会在不同的操作系统上做相应的实现和优化

• 五类日志等级

• 变参的日志函数和日志宏

• 行日志风格方案选配

• 输出介质有文件、标准输出、标准错误输出、syslogd 或 WINDOWS EVENT、自定义介质

高级特性如下：

• 支持日志选项组合

• 支持按日志文件大小、每天、每小时转档

• 支持行日志风格自定义回调函数，很容易定制自己的行日志格式

• 支持日志文件的打开、输出、关闭自定义回调函数，很容易扩展成日志输出到远程日志服务器落地

• 线程安全、简易 MDC、基于线程本地存储的缺省全局日志句柄

分层实施“**日志句柄层(LOG)->日志句柄集合层(LOGS)->配置文件接口层(LOGCONF、LOGSCONF)**”。其实大部分用户的日志需求很简单，一个进程写一个日志文件（使用日志句柄层函数即可），但也考虑到另外一些用户有多个输出对象需求（使用日志句柄集合层函数即可），还有用户喜欢用外部配置文件来配置日志（使用配置文件接口层函数即可），不同用户在不同项目场景中使用 iLOG3 的不同层接口，不至于杀鸡用牛刀、小刀砍大树。

我还开发了一个姐妹函数库 iLOG3CONF_SML 来支持用 SML 标记语言配置文件来配置日志句柄，有兴趣的朋友也可以调用日志句柄层或日志句柄集合层函数开发自己的 iLOG3CONF_*，实现用 XML 或现在流行的 json 或自己项目统一配置文件格式，来实现用外部配置文件配置 iLOG3 日志句柄环境。

此外，源代码结构也比较简单，只有三对源文件，便于搬运、嵌入和修改。（后面有源码分析章节帮助读者理解和 fork 自己的版本）

3 编译安装

3.1 依赖库

在安装前必须知道 iLOG3 所依赖的库环境。据我所知，只要有基本的 c 编译环境即可，

如果有 pthread（UNIX & Linux）就支持线程安全了，如果按 c99 编译，就能多支持几个写日志宏方便用户使用，即使以上如果都没有，也不妨碍编译和基本使用，只是使用功能少了些。

3.2 用自带 makefile 或 VC6 工程文件编译安装

拿到源代码包，在合适的目录里解开并进入

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

iLOG3 没有采用 automake 安装，原因是 automake 复杂到够我学一学期了，于是我写了一个类似 automake 的东东 mktpl，iLOG3 的所有 makefile 都由 mktpl 自动生成各操作系统对应 makefile，用户对应使用即可。

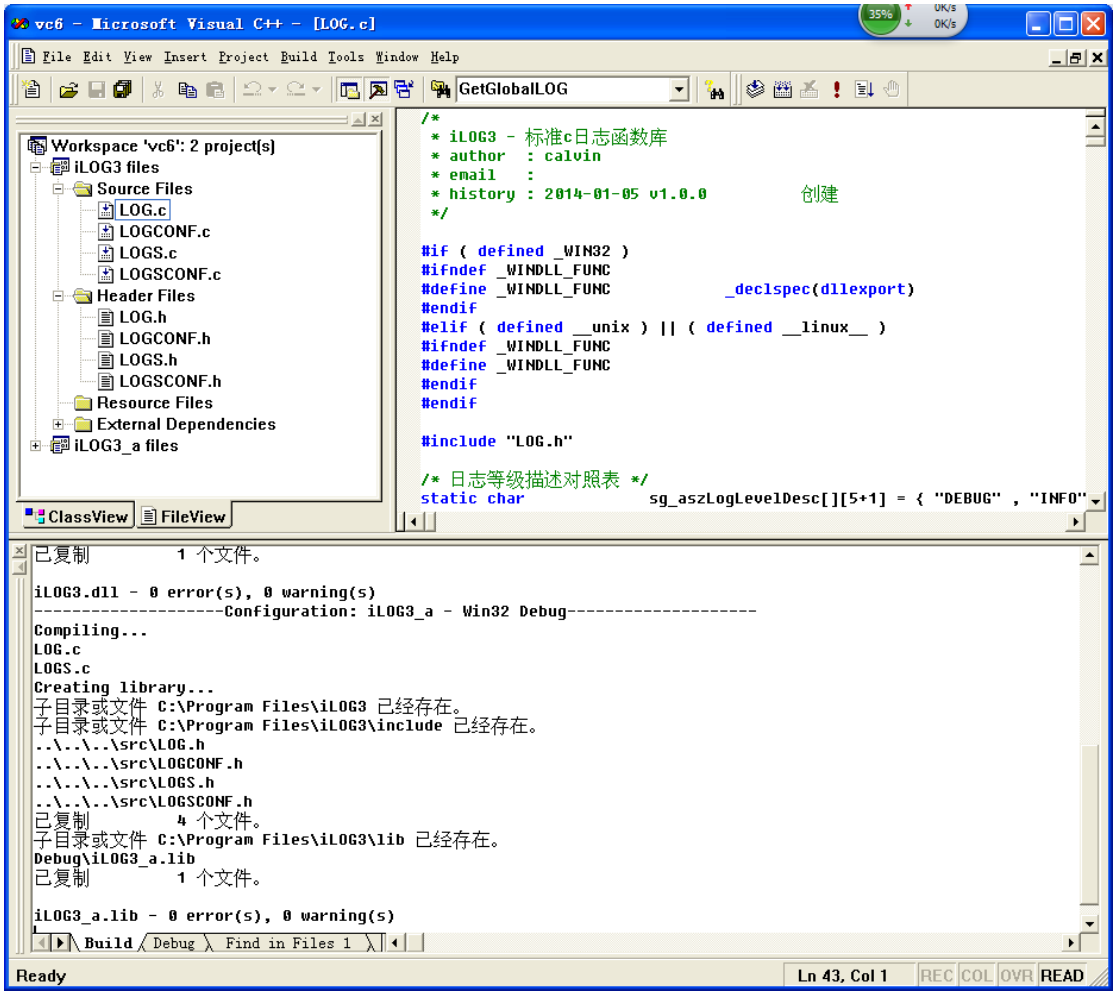
假如目标环境是 Linux，在 src 目录中打开 makefile.Linux，修改其中的 HDERINST 指向头文件安装目录（一般为/usr/local/include），修改其中的 LIBINST 和 LIBINST2 指向库文件安装目录（一般为/usr/local/lib）。

然后还是用 makefile.Linux 编译链接安装，顺利的话出现如下输出表示安装完成

```
$ 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 上有 makefile.MinGW 和 VC6 工程文件两种编译安装方式。其中 VC6 编译安装的话请重击文件"iLOG3-x.x.x\src\vc6\vc6.dsw"，重击菜单"Project"->"Settings..."->"Post-Build step"，设置好安装目标目录，最后重击"Build"->"Rebuild All"，底部 Output 窗口出现编译结果如下输出



安装目标路径如下：

```
C:\Program Files\iLOG3
├──bin
|       iLOG3.dll
|
├──include
|       LOG.h
|       LOGS.h
```

```
|       LOGCONF.h
|       LOGSCONF.h
|
└──lib
        iLOG3.lib
        iLOG3_a.lib
```

友情提示：在 WINDOWS 环境变量 PATH 中加上 iLOG3.dll 所在目录

至此，iLOG3 在你的系统里编译安装完成。

3.3 自己手动编译安装

如果你想自己手动编译安装，或是想做个 automake，很简单，记住只要把 LOG.c、LOGS.c 编译成 libiLOG3.so(libiLOG3.a)，以 Linux 环境链接动态库为例

```
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
```

如果你不需要配置文件接口层，追求小而精（特别是在嵌入式环境），把 LOGCONF.c 和 LOGSCONF.c 去除吧。

如果你只有写一个日志文件的需求，不需要日志句柄集合层，去除 LOGS.c，只编译 LOG.c 一个文件即可。

不加编译选项-std=c99 主要少了一批可变参数写日志宏。不加链接选项-lpthread（同时头文件中去掉”#include <pthread.h>”）少了线程安全和基于线程本地存储的缺省全局日志句柄。还是那句话，不妨碍编译和基本使用。

复制 libiLOG3.so 到库文件目录，复制*.hh 到头文件目录。搞定！

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

4 基本使用

4.1 一点点概念

我们先从日志句柄层开始，造好了一楼再造二楼。

就像操作标准文件流 FILE *fp 一样，**写日志要先创建一个日志句柄，设置一堆日志句柄属性，然后不停的写日志...最后销毁日志句柄**。一个日志句柄对应一个日志输出对象（文件或标准输出或其它），对应一个日志过滤等级，对应一个行日志风格方案。以上就是使用前需要掌握的全部概念，很简单吧。

有人会问我如果同时写两个日志文件怎么办？答曰：创建两个日志句柄，该写哪个写哪个，直截了当，简单就是美，没有分类，没有记录器没有输出器没有布局器，没有继承，一切完全“私人定制”。好吧，我承认我受控于当初的轻便简易设计准则，其实我很想加入很多很酷的概念，但一想到自己写的一千多行的山寨版 nginx 核心却要配五千多行的日志模块就不寒而栗。

日志句柄集合也可以实现同时有多个输出对象（或文件），具体见后面章节。

废话到此为止，上示例

4.2 第一个示例

（本文所有示例都取自于源代码包自带的测试例程 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 ; /* 日志句柄指针 */

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

```
	/* 创建日志句柄 */
	g = CreateLogHandle() ;
	if( g == NULL )
	{
		printf( "创建日志环境失败 errno[%d]\n" , errno );
		return -1;
	}
	else
	{
		printf( "创建日志环境成功\n" );
	}

	/* 设置日志输出文件名 */
	SetLogOutput( g , LOG_OUTPUT_FILE , "test_hello.log" , LOG_NO_OUTPUTFUNC );
	/* 设置当前日志过滤等级 */
	SetLogLevel( g , LOG_LEVEL_INFO );
	/* 设置当前行日志风格方案 */
	SetLogStyles( g , LOG_STYLES_HELLO , LOG_NO_STYLEFUNC);

	/* 以不同日志等级写行日志 */
	DebugLog( g , __FILE__ , __LINE__ , "hello iLOG3" ); /* 这行日志因等级不够，被华丽的过滤了 */
	InfoLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	WarnLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	ErrorLog( g , __FILE__ , __LINE__ , "hello iLOG3" );
	FatalLog( g , __FILE__ , __LINE__ , "hello iLOG3" );

	/* 以不同日志等级写十六进制块日志 */
	DebugHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen ); /* 又一个被华丽的过滤 */
	InfoHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	WarnHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	ErrorHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	FatalHexLog( g , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );

	/* 销毁日志句柄 */
	DestroyLogHandle( g );
	printf( "释放日志句柄\n" );

	return 0;
}

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

```
}
```

该示例中首先创建日志句柄，设置日志句柄环境如设置日志文件名，然后写日志，最后销毁日志句柄。

成功编译链接运行，产生日志文件如下：（注意：word 排版有错位）

```
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 | 缓冲区[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 | 缓冲区[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 | 缓冲区[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 | 缓冲区[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    ................
```

很简单，是吧？恭喜你已经学会使用 iLOG3 日志函数库的全部内容的一半了，真的，相信我。

一个记忆小技巧：日志句柄类型为 LOG，对应头文件 LOG.h，日志句柄集合类型为 LOGS，对应头文件 LOGS.h，顾名思义，LOGS 就是多个 LOG 集合体。

4.3 行日志风格方案选配

可能你觉得行日志风格太简单了，包含东东太少了，比如你还想加入毫秒以观察你开发的高性能应用服务器的耗时，你还想加入进程 id，哦，还有线程 id...好了，到此为止，目前我只能提供这些了。

修改行日志风格方案宏

```
#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 )
```

再次编译链接和运行

```
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 | 缓冲区[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 | 缓冲区[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 | 缓冲区[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 | 缓冲区[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    ................
```

完整行日志风格组合宏列表在《iLOG3 日志函数库参考手册》，有兴趣可以去看看。

5 高级特性

5.1 设置日志选项

iLOG3 日志选项集中在这里设置，且**必须在其它句柄设置函数前调用**。

日志输出模式宏决定打开关闭日志的时机，打开后不关闭有助于提高输出性能但当文件移走或删除后会有些小别扭，每次都打开关闭日志比较保险但性能较低，侦测文件变动则用于平衡可靠和性能要求。

```
LOG_OPTION_OPEN_AND_CLOSE     每次都打开日志、写日志、关闭日志
LOG_OPTION_CHANGE_TEST        侦测文件变动（缺省自动设置）
LOG_OPTION_OPEN_ONCE          日志打开一次不关闭
```

还存在一些有趣的选项

```
LOG_OPTION_FILENAME_SET_OUTPUT          自动根据文件名重置输出类型
LOG_OPTION_FILENAME_APPEND_DOT_LOG      日志输出文件名后自动加上".log"
```

比如选项 FILENAME_SET_OUTPUT 会根据日志文件名中包含的关键字调整输出类型，下例中从环境变量 TEST_FUNNY_LOG_OUTPUT 中获取文件名，如果设置环境变量 TEST_FUNNY_LOG_OUTPUT 为”#stdout#”再执行程序，日志都会跑到标准输出上去了。

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

选项 FILENAME_APPEND_DOT_LOG 会把得到的文件名最后自动加上”.log”，方便应用设置日志文件名

```
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 日志文件转档

日志文件转档是个大主题。iLOG3 日志函数库提供了三种转档方案供大家选用：

- 按日志文件大小转档
- 按天转档

• 按小时转档

函数 SetLogRotateMode 是用来设置转档方案的

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

mode 有以下取值：

```
#define LOG_ROTATEMODE_NONE          0 /* 不转档 */
#define LOG_ROTATEMODE_SIZE          1 /* 按日志文件大小转档 */
#define LOG_ROTATEMODE_PER_DAY       2 /* 按天转档 */
#define LOG_ROTATEMODE_PER_HOUR      3 /* 按小时转档 */
```

当设置为按日志文件大小转档时，还应调用函数 SetLogRotateSize 设置转档文件大小，一般实际转档大小都是略大于该大小值。由于大小转档优化算法需要一个系数，你可以调用函数 SetLogRotatePressureFactor 设置该系数大约为”写该日志文件的进程数*线程数*2”，当然，你不设置其实也没多大关系，最多转档文件写过头了点。按日志文件大小转档时，转档后文件名格式为”原日志文件名.序号”来排列。代码示例如下：

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

按天转档时，转档后文件名格式为”原日志文件名.年年年年月月日日”。代码示例如下：

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

按小时转档时，转档后文件名格式为”原日志文件名.年年年年月月日日_小时”。代码示例如下：

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

5.3 自定义行日志风格

我老觉得有人会质疑 iLOG3 的行日志风格过于方案化而缺乏完全的自定义灵活性。好吧，那我就设置一个回调函数给你，你爱怎么设置行日志风格就怎么设置。

设置日志句柄环境设置时代码中加入

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

编写回调函数如下：

```
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;
}
```

编译链接又运行，得到像这样的日志：

```
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 自定义日志输出类型

iLOG3 的日志输出对象不仅仅是文件，还支持标准输出、标准错误输出、syslog 或 WINDOWS EVENT 以及自定义日志输出函数等输出类型。

打开日志时设置 output 为 LOG_OUTPUT_STDOUT 宏就能把日志输出到屏幕上，标准错误输出同理

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

LOG_OUTPUT_SYSLOG 宏在 UNIX&Linux 上代表输出到 syslog ，在 WINDOWS 上代表 WINDOWS EVENT，代码如下：

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

自定义日志输出函数见下一节

5.5 自定义日志的打开、输出、关闭

如果是作为一个本地日志落地的话，iLOG3 完美完成了任务，但是我的野心很大，我要

把 iLOG3 折腾成可以作为一个远程日志落地的本地通讯客户端，我为了满足自己的私欲又一次无耻的增加了上百行代码。

下面是代码示例，把设置日志输出函数调用修改成这样，函数列表正式展现：

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

编写回调函数如下，这里我用 printf 代替假想的通讯连接、通讯数据发送和通讯断开

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

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

	g->open_flag = 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 ;

	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( g->open_flag == 0 )
		return 0;

	printf( "MyCloseLogFinally\n" );

	g->open_flag = 0 ;
	return 0;
}
```

编译又链接运行，显示如下：

```
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
```

这里只是演示一个框架，具体的通讯细节由你填充。

5.6 线程安全

一个日志函数库如果不能做到线程安全那就不算是一个通用的日志函数库。

如果你开发的是多进程应用，无需考虑线程安全，直接简单粗暴的把日志句柄声明成全局变量吧，如果是多线程就得考虑线程安全性问题了。

iLOG3 的线程安全编码方式至少有三种，一种是用全局日志句柄+自己实现互斥机制，第二种看下一段。

首先，日志函数库中不能调用线程不安全库函数，我也一直在查找，找到一个就用线程安全的版本替换之。然后，存储上，如果你要做到一个变量在线程间隔离，最简单的做法是声明成局部变量。iLOG3 日志句柄也一样，你声明一个自动的局部的 LOG 结构指针，调用函数 CreateLogHandle 创建日志句柄结构传回给它指向，它就是线程安全的。

但这样使用会给应用代码带来一定的麻烦，如果你的环境支持线程本地存储(TLS)，恭喜您，你可以启用基于线程本地存储的全局日志句柄来帮你自动隔离，我搞了一个缺省 TLS 全局日志句柄用来简化代码编写，具体见后面章节。

5.6.1 MDC

讳莫如深的名词，其实就是你自定义一些字符串值用于格式化到行日志中，但是又要做到线程间不影响，只要把这些字符串塞进已经线程隔离的日志句柄中，是不是就线程隔离了呢。

添加代码，往日志句柄 iLOG3 中压入标签”COREIB1_SERVICE”到一号索引的单元格中

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

然后在行日志风格组合宏中加入索引一标签配置

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

那么日志句柄 event 的行日志输出中就有”COREIB1_SERVICE”段了

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

标签最多可以设置三个，除非你修改了宏并重编了日志函数库源码

```
/* 自定义标签数量 */
#define LOG_MAXCNT_CUST_LABEL          3
```

标签值长度最长 32 字符，除非你修改了宏并重编了日志函数库源码

```
/* 自定义标签最大长度 */
#define LOG_MAXLEN_CUST_LABEL          32
```

你确信你真的需要那么多标签？

下一节是激动人心的...

5.6.2 基于线程本地存储的缺省全局日志句柄

前面讲到的日志句柄线程安全编码的一种方式是利用局部变量来实现，你是不是觉得太没技术含量了，接下来的技术含量喂你到饱。现在打开自己常用的搜索引擎，键入关键词”线程 TLS”，轻按回车，学习一下什么叫”线程本地存储”。

现在我假设你已经饱了，我们回来继续看 iLOG3 日志函数库。用局部变量来达到日志句柄线程安全会给使用者带来一些不爽，即把前面创建的日志句柄在你的各大函数中来回传递，利用线程本地存储就能把日志句柄声明成线程私有全局变量，听不懂？你就当它是全局变量好了。

iLOG3 日志函数库有一个线程本地存储的缺省全局日志句柄，变量名不能告诉你，这是秘密，你只要使用基于线程本地存储的缺省全局日志句柄版本的函数和宏就可以省却前面讲过的后面会讲到的所有函数的第一个参数即日志句柄，只要函数名最后加一个”G”就行了，像这样

```
CreateLogHandleG();
SetLogOutputG( LOG_OUTPUT_FILE , "test_hello.log");
SetLogLevelG( LOG_LEVEL_INFO );
```

```
SetLogStylesG( LOG_STYLES_HELLO , LOG_NO_STYLESFUNC );
DestroyLogHandleG();
```

是不是很省心，如果你想使用自己创建的日志句柄作为线程本地存储日志句柄，在创建它后用这个函数来替换缺省的即可

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

友情提示：线程本地存储并不是所有编译器都支持，据我所知 WINDOWS 上的 MS 编译器、Linux 上的 gcc 编译器和 AIX 上的 xlc 编译器是没问题的，其它的请大家试用后麻烦告诉我一声，我马上补充进去。

5.7 一个常用示例

可能绝大多数场合应用都希望日志句柄能做到全局访问，这样方便源代码结构，这里给出一个比较常用的代码示例

```
#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 ;

	/* 创建日志句柄 */
	if(CreateLogHandleG() == NULL )
	{
		printf( "创建日志环境失败 errno[%d]\n" , errno );
		return -1;
	}
	else
	{
		printf( "创建日志环境成功\n" );
	}

	/* 设置日志输出文件名 */
	SetLogOutputG( LOG_OUTPUT_FILE , "test_hello.log" , LOG_NO_OUTPUTFUNC );
	/* 设置当前日志过滤等级 */
```

```
	SetLogLevelG( LOG_LEVEL_INFO );
	/* 设置当前行日志风格方案 */
	SetLogStylesG( LOG_STYLES_HELLO , LOG_NO_STYLEFUNC);

	/* 以不同日志等级写行日志 */
	DebugLogG( __FILE__ , __LINE__ , "hello iLOG3" ); /* 这行日志因等级不够，被华丽的过滤了 */
	InfoLogG( __FILE__ , __LINE__ , "hello iLOG3" );
	WarnLogG( __FILE__ , __LINE__ , "hello iLOG3" );
	ErrorLogG( __FILE__ , __LINE__ , "hello iLOG3" );
	FatalLogG( __FILE__ , __LINE__ , "hello iLOG3" );

	/* 以不同日志等级写十六进制块日志 */
	DebugHexLogG( __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen ); /* 又一个被华丽的过滤 */
	InfoHexLogG( __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	WarnHexLogG( __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	ErrorHexLogG( __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	FatalHexLogG( __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );

	/* 销毁日志句柄 */
	DestroyLogHandleG();
	printf( "释放日志句柄\n" );

	return 0;
}

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

6 日志句柄集合层

前面讲的统统是日志句柄层（LOG）的使用说明，在小型项目或高性能场景中足够用了，有些用户比较喜欢折腾日志，我很早以前也这样，比如一个进程处理一笔交易，写进程详细日志(dog.log)，每笔交易处理完后写一行事件简约日志(event.log)，报错时写错误日志(error.log)，这里需要三个日志句柄，我把它们整合起来包装成一个日志句柄集合，名字就叫...LOGS，同时把写日志函数名或宏名中的 Log 改成 Logs 就成了写日志句柄集合的函数和宏，很好记，不是吗？

以下示例说明了日志句柄集合的使用，涉及多种使用方法，希望你能看懂。

```
#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( "创建日志句柄集合失败 errno[%d]\n" , errno );
		return -1;
	}

	g = CreateLogHandle() ;
	if( g == NULL )
	{
		printf( "创建日志句柄失败 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( "得到日志句柄失败 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( "创建日志句柄集合成功\n" );

	DebugLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	InfoLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	WarnLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	ErrorLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );
	FatalLogs( gs , __FILE__ , __LINE__ , "hello iLOG3" );

	DebugHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	InfoHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	WarnHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	ErrorHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );
	FatalHexLogs( gs , __FILE__ , __LINE__ , buffer , buflen , "缓冲区[%ld]" , buflen );

	DestroyLogHandle( RemoveOutLogFromLogs( gs , "FILE" ) );
	printf( "销毁日志句柄\n" );

	DestroyLogsHandle( gs );
	printf( "销毁日志句柄集合\n" );

	return 0;
}

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

创建日志句柄集合，再创建两个日志句柄，add 进去，用 g_id 来标识唯一日志句柄防止搞错，其中第一个日志句柄是创建设置完再 add 进去，第二个是先创建后马上 add 进去，然后再慢悠悠设置日志句柄环境。整好日志句柄集合后用 Logs 版本的函数写日志，一次调用同时输出多个日志句柄。最后销毁日志句柄集合，其中有一个日志句柄是移出日志句柄集合

后用日志句柄销毁函数销毁。当然实际使用时不用这么麻烦，这只是为了演示罢了。

7 配置文件接口层

7.1 自带的简单配置文件接口层

自带的配置文件接口层实现了比较简单的日志配置格式，好吧，没有比它更简单的了，看

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
```

一个这样的配置文件配置单个日志句柄，很容易看出来，每行设置一个属性，每个属性由一个 key 和一个 value 组成，像 output 或 log_buffersize 则有多个 value，中间都用白字符（空格、TAB）隔开，完整 key 列表见参考手册。

在代码中这样写

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

就能从配置文件构建日志句柄，代替原来的调用函数构建，是不是很方便，但是如果你用到回调函数，设置回调函数还是要调用函数来设置的。

我一般把可执行程序名和日志配置文件名命名成一样，貌似这是 c#的一种风格，比如可执行文件 test 和它的日志配置文件 test.conf 放在一个目录里，当然，大项目则应该把日志配置统一放在配置文件目录里。

下面是日志句柄集合配置

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
```

每个配置块之间用空行隔开，配置块开头多了个属性 id 对应日志句柄在日志句柄集合中的 g_id。

调用代码如下：

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

都只是多了一个’s’，你看懂了吗？

一个小技巧，类似日志选项 LOG_OPTION_FILENAME_APPEND_DOT_LOG 的作用

```
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] );
}
```

7.2 配置文件接口展望

自带的配置文件接口层只是提供了最基本的配置机制，同时也演示了如何解析某种格式配置后使用配置辅助函数转换和构建日志句柄环境。我在稍后开发的函数库 iLOG3CONF_SML 实现了用 SML 语言作为 iLOG3 配置文件格式的高级使用，而且还支持配置内嵌，在我的山寨 nginx（目前暂定名为 hetao）主配置中是这样配置日志环境的

```
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 ;
        }
}
```

代码中这样获取配置构建日志句柄集合

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

然后就能刷刷刷的写 access.log、error.log 了，很爽吧 ^_^

8 源码分析

我前面说了，本章节是为了帮助读者理解源码和 fork 自己的版本而写得，如果你对源码没兴趣可以直接跳到下一章。

好了，iLOG3 的实现文件源代码已经接近 2500 行了，这超出了我当初的轻便设计初衷，就此打住，恕不接待新需求了。

回顾自己的代码，空行控是我的风格，线程本地存储的缺省全局日志句柄版本的函数又使得源码重复了一倍，我就当代码只有 1000 行不到吧。看，这么多功能的日志函数库源代码只保持千行之内，还轻便吧？额~

虽说千行代码直接放在你面前是不礼貌的，但我感觉我写的还是挺容易阅读，毕竟编码的时候我经常考虑代码的易读性，即使再优美的代码只要不便于一眼看出来我就改改改。下面我带大家来剖析 iLOG3 源代码，希望大家会喜欢。

日志句柄的源代码主要从三条线来分析，第一条线是日志句柄的创建销毁，第二条线是日志句柄的环境设置，第三条线是实际写日志。至于日志句柄集合则没什么好说的，只是简单做个集合遍历而已。

8.1 第一条线：日志句柄的创建销毁

首当其冲的是创建日志句柄函数 CreateLogHandle，无非就是在堆上申请一块内存当作 LOG 日志句柄（结构体变量）使用，初始化云云

```
/* 创建日志句柄 */
LOG *CreateLogHandle()
{
	LOG		*g = NULL ;
	int		nret ;

#if ( defined _WIN32 )
#elif ( defined __unix ) || ( defined __linux__ )
#else
	return LOG_RETURN_ERROR_NOTSUPPORT;
#endif

	g = (LOG *)malloc( sizeof(LOG) ) ;
	if( g == NULL )
		return NULL;
	memset( g , 0x00 , sizeof(LOG) );

	SetLogLevel( g , LOG_LEVEL_DEFAULT );
	SetLogStyles( g , LOG_STYLES_DEFAULT , LOG_NO_STYLEFUNC );
	SetLogOptions( g , LOG_OPTION_CHANGE_TEST );

	g->rotate_mode = LOG_ROTATEMODE_NONE ;
	g->rotate_file_no = 1 ;
	g->rotate_file_count = LOG_ROTATE_SIZE_FILE_COUNT_DEFAULT ;
	g->pressure_factor = LOG_ROTATE_SIZE_PRESSURE_FACTOR_DEFAULT ;

	g->logbuf.buf_size = 0 ;
	g->logbuf.bufbase = NULL ;
	g->logbuf.bufptr = NULL ;
	g->logbuf.buf_remain_len = 0 ;
	nret = SetLogBufferSize( g , LOG_BUFSIZE_DEFAULT , LOG_BUFSIZE_MAX ) ;
	if( nret )
	{
```

```
		DestroyLogHandle( g );
		return NULL;
	}

	g->hexlogbuf.buf_size = 0 ;
	g->hexlogbuf.bufbase = NULL ;
	g->hexlogbuf.bufptr = NULL ;
	g->hexlogbuf.buf_remain_len = 0 ;
	nret = SetHexLogBufferSize( g , LOG_HEXLOG_BUFSIZE_DEFAULT , LOG_HEXLOG_BUFSIZE_MAX ) ;
	if( nret )
	{
		DestroyLogHandle( g );
		return NULL;
	}

	nret = CreateMutexSection( g ) ;
	if( nret )
	{
		DestroyLogHandle( g );
		return NULL;
	}

	return g;
}
```

成员 logbuf 和 hexlogbuf 是行日志和十六进制块日志格式化缓冲区，此外还用函数 CreateMutexSection 创建了系统级的互斥区用于转档时的临界区。如果创建失败则调用销毁函数 DestroyLogHandle。

销毁日志句柄则反过来。第一条线完了。

8.2 第二条线：日志句柄的环境设置

当日志句柄创建后就是设置其环境了。一般首先设置输出日志文件名

```
/* 设置日志输出 */
int SetLogOutput( LOG *g , int output , char *log_pathfilename , funcOpenLog *pfuncOpenLogFirst , funcOpenLog
*pfuncOpenLog , funcWriteLog *pfuncWriteLog , funcChangeTest *pfuncChangeTest , funcCloseLog *pfuncCloseLog ,
funcCloseLog *pfuncCloseLogFinally )
{
	char		pathfilename[ MAXLEN_FILENAME + 1 ] ;
	long		pathfilename_len ;
```

```
char            env_key[ MAXLEN_FILENAME + 1 ] ;
long            env_key_len ;
char            *env_val = NULL ;
long            env_val_len ;

char            *p1 = NULL , *p2 = NULL ;

int             nret = 0 ;

if( g == NULL )
        return LOG_RETURN_ERROR_PARAMETER;
if( log_pathfilename == NULL || log_pathfilename[0] == '\0' )
{
        memset( pathfilename , 0x00 , sizeof(pathfilename) );
}
else
{
        memset( pathfilename , 0x00 , sizeof(pathfilename) );
        strncpy( pathfilename , log_pathfilename , sizeof(pathfilename)-1 );
        pathfilename_len = strlen(pathfilename) ;

        p1 = strchr( pathfilename , '$' );
        while( p1 )
        {
                /* 展开环境变量 */
                p2 = strchr( p1 + 1 , '$' ) ;
                if( p2 == NULL )
                        return LOG_RETURN_ERROR_PARAMETER;

                memset( env_key , 0x00 , sizeof(env_key) );
                env_key_len = p2 - p1 + 1 ;
                strncpy( env_key , p1 + 1 , env_key_len - 2 );
                env_val = getenv( env_key ) ;
                if( env_val == NULL )
                        return LOG_RETURN_ERROR_PARAMETER;

                env_val_len = strlen(env_val) ;
                if( pathfilename_len + ( env_val_len - env_key_len ) > sizeof(pathfilename)-1 )
                        return LOG_RETURN_ERROR_PARAMETER;

                memmove( p2+1 + ( env_val_len - env_key_len ) , p2+1 , strlen(p2+1) + 1 );
                memcpy( p1 , env_val , env_val_len );
                pathfilename_len += env_val_len - env_key_len ;
```

```
                p1 = strchr( p1 + ( env_val_len - env_key_len ) , '$' );
        }
    }

    if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_SET_OUTPUT_BY_FILENAME ) )
    {
        if( strncmp( pathfilename , "#stdout#" , 8 ) == 0 )
        {
            memmove( pathfilename , pathfilename+8 , strlen(pathfilename+8)+1 );
            output = LOG_OUTPUT_STDOUT ;
        }
        else if( strncmp( pathfilename , "#stderr#" , 8 ) == 0 )
        {
            memmove( pathfilename , pathfilename+8 , strlen(pathfilename+8)+1 );
            output = LOG_OUTPUT_STDERR ;
        }
        else if( strncmp( pathfilename , "#syslog#" , 8 ) == 0 )
        {
            memmove( pathfilename , pathfilename+8 , strlen(pathfilename+8)+1 );
            output = LOG_OUTPUT_SYSLOG ;
        }
    }

    if( output == LOG_OUTPUT_FILE && TEST_ATTRIBUTE( g->log_options ,
LOG_OPTION_FILENAME_APPEND_DOT_LOG ) )
    {
        if( strlen(pathfilename) + 4 > sizeof(pathfilename) - 1 )
            return LOG_RETURN_ERROR_PARAMETER;

        strcat( pathfilename , ".log" );
    }

    if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) || TEST_ATTRIBUTE( g->log_options ,
LOG_OPTION_CHANGE_TEST ) )
    {
        if( g->pfuncCloseLogFinally )
        {
            nret = g->pfuncCloseLogFinally( g , & (g->open_handle) ) ;
            if( nret )
                return nret;
        }
    }

    if( TEST_NOT_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_AND_CLOSE )
```

```
	&& TEST_NOT_ATTRIBUTE( g->log_options , LOG_OPTION_CHANGE_TEST )
	&& TEST_NOT_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) )
{
	g->log_options |= LOG_OPTION_OPEN_DEFAULT ;
}

memset( g->log_pathfilename , 0x00 , sizeof(g->log_pathfilename) );
strncpy( g->log_pathfilename , pathfilename , sizeof(g->log_pathfilename)-1 );

g->output = output ;
if( g->output == LOG_OUTPUT_STDOUT )
{
	g->fd = STDOUT_HANDLE ;
	g->pfuncOpenLogFirst = NULL ;
	g->pfuncOpenLog = NULL ;
	g->pfuncWriteLog = & WriteLog_write ;
	g->pfuncChangeTest = NULL ;
	g->pfuncCloseLog = NULL ;
	g->pfuncCloseLogFinally = NULL ;

	g->log_options &= ~LOG_OPTION_OPEN_AND_CLOSE ;
	g->log_options &= ~LOG_OPTION_CHANGE_TEST ;
	g->log_options |= LOG_OPTION_OPEN_ONCE ;

	g->open_flag = 1 ;
}
else if( output == LOG_OUTPUT_STDERR )
{
	g->fd = STDERR_HANDLE ;
	g->pfuncOpenLogFirst = NULL ;
	g->pfuncOpenLog = NULL ;
	g->pfuncWriteLog = & WriteLog_write ;
	g->pfuncChangeTest = NULL ;
	g->pfuncCloseLog = NULL ;
	g->pfuncCloseLogFinally = NULL ;

	g->log_options &= ~LOG_OPTION_OPEN_AND_CLOSE ;
	g->log_options &= ~LOG_OPTION_CHANGE_TEST ;
	g->log_options |= LOG_OPTION_OPEN_ONCE ;

	g->open_flag = 1 ;
}
else if( output == LOG_OUTPUT_SYSLOG )
{
```

```
#if ( defined _WIN32 )
		g->pfuncOpenLogFirst = & OpenLog_RegisterEventSource ;
		g->pfuncOpenLog = NULL ;
		g->pfuncWriteLog = & WriteLog_ReportEvent ;
		g->pfuncChangeTest = NULL ;
		g->pfuncCloseLog = NULL ;
		g->pfuncCloseLogFinally = & CloseLog_DeregisterEventSource ;
#elif ( defined __unix ) || ( defined __linux__ )
		g->pfuncOpenLogFirst = & OpenLog_openlog ;
		g->pfuncOpenLog = NULL ;
		g->pfuncWriteLog = & WriteLog_syslog ;
		g->pfuncChangeTest = NULL ;
		g->pfuncCloseLog = NULL ;
		g->pfuncCloseLogFinally = & CloseLog_closelog ;
#endif

		g->log_options &= ~LOG_OPTION_OPEN_AND_CLOSE ;
		g->log_options &= ~LOG_OPTION_CHANGE_TEST ;
		g->log_options |= LOG_OPTION_OPEN_ONCE ;

		g->open_flag = 0 ;
	}
	else if( output == LOG_OUTPUT_FILE )
	{
#if ( defined _WIN32 )
		if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) || TEST_ATTRIBUTE( g->log_options ,
LOG_OPTION_CHANGE_TEST ) )
		{
			g->pfuncOpenLogFirst = & OpenLog_OpenFile ;
			g->pfuncOpenLog = NULL ;
			g->pfuncWriteLog = & WriteLog_WriteFile ;
			g->pfuncChangeTest = & ChangeTest_interval ;
			g->pfuncCloseLog = NULL ;
			g->pfuncCloseLogFinally = & CloseLog_CloseHandle ;
		}
		else
		{
			g->pfuncOpenLogFirst = NULL ;
			g->pfuncOpenLog = & OpenLog_OpenFile ;
			g->pfuncWriteLog = & WriteLog_WriteFile ;
			g->pfuncChangeTest = NULL ;
			g->pfuncCloseLog = & CloseLog_CloseHandle ;
			g->pfuncCloseLogFinally = NULL ;
		}
```

```
#elif ( defined __unix ) || ( defined __linux__ )
		if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) || TEST_ATTRIBUTE( g->log_options , LOG_OPTION_CHANGE_TEST ) )
		{
			g->pfuncOpenLogFirst = & OpenLog_open ;
			g->pfuncOpenLog = NULL ;
			g->pfuncWriteLog = & WriteLog_write ;
			g->pfuncChangeTest = & ChangeTest_interval ;
			g->pfuncCloseLog = NULL ;
			g->pfuncCloseLogFinally = & CloseLog_close ;
		}
		else
		{
			g->pfuncOpenLogFirst = NULL ;
			g->pfuncOpenLog = & OpenLog_open ;
			g->pfuncWriteLog = & WriteLog_write ;
			g->pfuncChangeTest = NULL ;
			g->pfuncCloseLog = & CloseLog_close ;
			g->pfuncCloseLogFinally = NULL ;
		}
#endif
		g->open_flag = 0 ;
	}
	else if( output == LOG_OUTPUT_CALLBACK )
	{
		SetLogOutputFuncDirectly( g , pfuncOpenLogFirst , pfuncOpenLog , pfuncWriteLog , pfuncChangeTest , pfuncCloseLog , pfuncCloseLogFinally );
		g->open_flag = 0 ;
	}

	if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) || TEST_ATTRIBUTE( g->log_options , LOG_OPTION_CHANGE_TEST ) )
	{
		if( g->pfuncOpenLogFirst )
		{
			nret = g->pfuncOpenLogFirst( g , g->log_pathfilename , & (g->open_handle) ) ;
			if( nret )
				return nret;
		}
	}

	return 0;
}
```

日志句柄中成员 output 存放输出类型，LOG_OUTPUT_FILE 代表文件，这时

log_pathfilename 存放日志输出文件名 LOG_OUTPUT_CALLBACK 代表启用自定义回调函数，并设置后面六个函数指针。还有很多其它输出类型见《参考手册》中的“日志输出类型宏”。

设置日志等级函数则比较简单

```
/* 设置日志等级 */
int SetLogLevel( LOG *g , int log_level )
{
	if( g == NULL )
		return LOG_RETURN_ERROR_PARAMETER;
	g->log_level = log_level ;
	return 0;
}
```

设置行日志风格时进行了格式化预处理，即分析传入的行日志风格组合宏，构建一个行日志风格函数数组，等后面每次实际写日志前的准备日志缓冲区时，只要顺序执行该行日志风格函数数组即可，大大减少了写日志时处理行日志风格配置的耗时。

```
/* 设置行日志风格 */
int SetLogStyles( LOG *g , long log_styles , funcLogStyle *pfuncLogStyle )
{
	if( g == NULL )
		return LOG_RETURN_ERROR_PARAMETER;

	g->log_styles = log_styles ;
	if( g->log_styles == LOG_STYLE_CALLBACK )
	{
		SetLogStyleFuncDirectly( g , pfuncLogStyle );
		return 0;
	}

	/* 构造行风格函数数组 */
	g->style_func_count = 0 ;

	if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_DATETIMEMS ) )
	{
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_DATETIMEMS ;
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR ;
	}
	else if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_DATETIME ) )
	{
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_DATETIME ;
```

```
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR ;
    }
    else if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_DATE ) )
    {
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_DATE ;
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR ;
    }
    if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_LOGLEVEL ) )
    {
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_LOGLEVEL ;
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR ;
    }
    if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL1 )
        || TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL2 )
        || TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL3 ) )
    {
        if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL1 ) )
        {
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_CUSTLABEL1 ;
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SPACE ;
        }
        if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL2 ) )
        {
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_CUSTLABEL2 ;
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SPACE ;
        }
        if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_CUSTLABEL3 ) )
        {
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_CUSTLABEL3 ;
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SPACE ;
        }
        g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR2 ;
    }
    if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_PID )
        || TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_TID )
        || TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_SOURCE ) )
    {
        if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_PID ) )
        {
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_PID ;
        }
        if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_TID ) )
        {
            g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_TID ;
```

```
		}
		if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_SOURCE ) )
		{
			g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SOURCE ;
		}
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_SEPARATOR ;
	}
	if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_FORMAT ) )
	{
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_FORMAT ;
	}
	if( TEST_ATTRIBUTE( g->log_styles , LOG_STYLE_NEWLINE ) )
	{
		g->pfuncLogStyles[ g->style_func_count++ ] = LogStyle_NEWLINE ;
	}

	g->pfuncLogStyle = & LogStyle_FuncArray ;

	return 0;
}
```

此外像设置日志选项、设置日志自定义标签、设置日志转档模式等函数比较简单，只是纯粹的赋值日志句柄中的成员而已，不讲解了。

第二条线完了，总结为“设置日志句柄环境”。

8.3 第三条线：写日志

我们从写错误日志函数作为样例切入。先放上使用者代码

```
ErrorLog( g , __FILE__ , __LINE__ , "hello iLOG3\n" );
```

进入函数 ErrorLog

```
/* 写错误日志 */
int ErrorLog( LOG *g , char *c_filename , long c_fileline , char *format , ... )
{
	WRITELOGBASE( g , LOG_LEVEL_ERROR )
	return 0;
}

/* 代码宏 */
#define WRITELOGBASE(_g_,_log_level_) \
	va_list		valist; \
	int		nret ; \
```

```
	if( (_g_) == NULL ) \
		return LOG_RETURN_ERROR_PARAMETER; \
	if( (_g_)->output == LOG_OUTPUT_FILE && (_g_)->log_pathfilename[0] == '\0' ) \
		return 0; \
	if( TEST_LOGLEVEL_NOTENOUGH( _log_level_ , (_g_) ) ) \
		return 0; \
	va_start( valist , format ); \
	nret = WriteLogBase( (_g_) , c_filename , c_fileline , _log_level_ , format , valist ) ; \
	va_end( valist ); \
	if( nret < 0 ) \
		return nret;
```

前半段无非是如果日志文件名为空或日志等级不够，则立即返回不写了。后半段通过一个变参写法调用写行日志基函数 WriteLogBase，该函数作为写不同等级日志函数的公共下层函数。

```
/* 写日志基函数 */
int WriteLogBase( LOG *g , char *c_filename , long c_fileline , int log_level , char *format , va_list valist )
{
	long		writelen ;
	int		nret ;

	if( format == NULL )
		return 0;

	/* 初始化行日志缓冲区 */
	g->logbuf.buf_remain_len = g->logbuf.buf_size - 1 - 1 ;
	g->logbuf.bufptr = g->logbuf.bufbase ;

	/* 填充行日志缓冲区 */
	if( g->pfuncLogStyle )
	{
		nret = g->pfuncLogStyle( g , & (g->logbuf) , c_filename , c_fileline , log_level , format , valist ) ;
		if( nret )
			return nret;
	}

	/* 打开文件*/
	if( g->open_flag == 0 )
	{
		if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_CHANGE_TEST ) || TEST_ATTRIBUTE( g->log_options ,
LOG_OPTION_OPEN_ONCE ) )
		{
			if( g->pfuncOpenLogFirst )
			{
```

```
                        nret = g->pfuncOpenLogFirst( g , g->log_pathfilename , & (g->open_handle) ) ;
                        if( nret )
                                return nret;
                }
        }
        else if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_AND_CLOSE ) )
        {
                /* 打开日志文件 */
                if( g->pfuncOpenLog )
                {
                        nret = g->pfuncOpenLog( g , g->log_pathfilename , & (g->open_handle) ) ;
                        if( nret )
                                return nret;
                }
        }
}

/* 导出日志缓冲区 */
if( g->pfuncWriteLog )
{
        nret = g->pfuncWriteLog( g , & (g->open_handle) , log_level , g->logbuf.bufbase , g->logbuf.buf_size-1-1 -
g->logbuf.buf_remain_len , & writelen ) ;
        if( nret )
                return nret;
}

/* 关闭日志 */
if( g->output == LOG_OUTPUT_FILE || g->output == LOG_OUTPUT_CALLBACK )
{
        if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_CHANGE_TEST ) )
        {
                if( g->pfuncChangeTest )
                {
                        nret = g->pfuncChangeTest( g , & (g->test_handle) ) ;
                        if( nret )
                                return nret;
                }
        }
        else if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_ONCE ) )
        {
        }
        else if( TEST_ATTRIBUTE( g->log_options , LOG_OPTION_OPEN_AND_CLOSE ) )
        {
                /* 关闭日志文件 */
```

```
                if( g->pfuncCloseLog )
                {
                        nret = g->pfuncCloseLog( g , & (g->open_handle) ) ;
                        if( nret )
                                return nret;
                }

        }
    }

    /* 如果输出到文件 */
    if( g->output == LOG_OUTPUT_FILE )
    {
        /* 日志转档侦测 */
        if( g->rotate_mode == LOG_ROTATEMODE_SIZE && g->log_rotate_size > 0 )
        {
                g->skip_count--;
                if( g->skip_count < 1 )
                {
                        RotateLogFileSize( g , writelen );
                }
        }
        else if( g->rotate_mode == LOG_ROTATEMODE_PER_DAY )
        {
                RotateLogFilePerDate( g );
        }
        else if( g->rotate_mode == LOG_ROTATEMODE_PER_HOUR )
        {
                RotateLogFilePerHour( g );
        }
    }

    /* 清空一级缓存 */
    g->cache1_tv.tv_sec = 0 ;
    g->cache1_stime.tm_mday = 0 ;

    return 0;
}
```

该函数做了以下工作

- 初始化行日志缓冲区
- 填充行日志缓冲区，函数是设置行日志风格时预置好的
- 打开日志文件，函数是设置日志输出时预置好的

• 导出日志缓冲区，函数是设置日志输出时预置好的

• 关闭日志文件，函数是设置日志输出时预置好的

• 最后清空一级缓存

转档函数里的算法以及行日志风格函数里的两级缓存算法就留给读者自己咀嚼了。

写十六进制块日志类同。

好了，恭喜您已经基本看完 2500 行代码了，现在是不是觉得 iLOG3 很轻便呢 ^_^

9 同类日志函数库比较

9.1 性能比较

由于 iLOG3 内部做了相当多的优化，自以为性能还是不错的。下面就以现在网上比较流行的同类日志函数库最新版本 log4c(v1.2.4)、zlog(v 1.2.12)和 iLOG3(v1.0.0)做一个简单的压测，最后给一个比较简单的总结，仅供参考。

压测环境在我的 2007 年买的老爷机 XP 里的虚拟机 VMWARE 里装的 Red Hat Enterprise Linux 5.4 里进行，压测场景为单进程开 20 个线程，每个线程写 10 万条日志，每行日志格式大致是”日期 时间 | PID:TID:源代码文件名:源行号 | loglog”，开压！

iLOG3 压测目录：iLOG3-1.0.0/test

iLOG3 压测命令：rm -f *.log* ; sleep 1 ; time ./test_press_mpt 1 10 100000 ; wc test_press_mpt.log* ; head -1 test_press_mpt.log.1

iLOG3 首压屏幕输出：

```
real    0m19.191s
user    0m1.822s
sys     0m17.340s
  23186   185488  1762139 test_press_mpt.log
 148674  1189392 11210480 test_press_mpt.log.1
 137974  1103792 10486024 test_press_mpt.log.2
 137974  1103792 10486024 test_press_mpt.log.3
 138270  1106160 10486208 test_press_mpt.log.4
 137976  1103808 10486176 test_press_mpt.log.5
```

```
 137973   1103784 10485950 test_press_mpt.log.6
 137973   1103784 10485949 test_press_mpt.log.7
1000000   8000000 75888950 total
2014-01-17 17:45:20 | INFO   | 27907:3086486416:test_press_mpt.c:129 | 1
```

zlog 压测目录：zlog-1.2.12/test

zlog 压测命令：rm -f *.log* ; sleep 1 ; time ./test_press_zlog 1 10 100000 ; wc press.log* ; head -1 press.log.0

zlog 首压屏幕输出：

```
real    0m20.531s
user    0m0.760s
sys     0m19.718s
 102559    820472   7999602 press.log
 128205   1025640   9999990 press.log.0
 128205   1025640   9999990 press.log.1
 128205   1025640   9999990 press.log.2
 128205   1025640   9999990 press.log.3
 128207   1025656  10000146 press.log.4
 128208   1025664  10000224 press.log.5
 128206   1025648  10000068 press.log.6
1000000   8000000  78000000 total
2014-01-17 17:47:10 | INFO    | 27983:3065871248:test_press_zlog.c:25 | loglog
```

log4c 压测目录：log4c-1.2.4/tests/log4c

log4c 压测命令：rm -f logrfmt* ; sleep 1 ; time ./test_rollingfile_appender_mt ; wc logrfmt* ; head -1 logrfmt.0

log4c 首压屏幕输出：

```
real    0m22.265s
user    0m2.810s
sys     0m19.390s
  37026    222156   3628548 logrfmt.0
 106997    641982  10485706 logrfmt.1
 106997    641982  10485706 logrfmt.2
 106997    641982  10485706 logrfmt.3
 106997    641982  10485706 logrfmt.4
 106997    641982  10485706 logrfmt.5
 106997    641982  10485706 logrfmt.6
 106997    641982  10485706 logrfmt.7
 106997    641982  10485706 logrfmt.8
 106997    641982  10485706 logrfmt.9
 999999   5999994  97999902 total
```

```
20140117 09:55:39.838 FATAL       root- (28547:3086404496:test_rollingfile_appender_mt.c:258) loglog
```

经过多次轮流压测，取中间段结果如下：

iLOG3	zlog	log4c
real 0m16.416s user 0m0.964s sys 0m15.413s	real 0m20.124s user 0m0.665s sys 0m19.421s	real 0m18.377s user 0m1.263s sys 0m17.085s
real 0m19.263s user 0m1.754s sys 0m17.475s	real 0m23.113s user 0m1.319s sys 0m21.740s	real 0m18.061s user 0m0.988s sys 0m17.033s
real 0m17.015s user 0m1.053s sys 0m15.933s	real 0m20.518s user 0m0.696s sys 0m19.786s	real 0m18.397s user 0m1.159s sys 0m17.198s
real 0m17.014s user 0m1.074s sys 0m15.897s	real 0m23.463s user 0m1.404s sys 0m22.002s	real 0m20.384s user 0m2.166s sys 0m18.171s
real 0m16.310s user 0m0.958s sys 0m15.323s	real 0m20.821s user 0m0.799s sys 0m19.977s	real 0m18.138s user 0m1.118s sys 0m16.978s
real 0m16.702s user 0m1.023s sys 0m15.648s	real 0m20.405s user 0m0.732s sys 0m19.606s	real 0m18.892s user 0m1.265s sys 0m17.577s
real 0m19.848s user 0m1.976s sys 0m17.815s	real 0m21.158s user 0m0.842s sys 0m20.257s	real 0m18.279s user 0m1.147s sys 0m17.081s
real 0m19.746s user 0m2.010s sys 0m17.688s	real 0m24.842s user 0m1.648s sys 0m23.125s	real 0m22.422s user 0m2.918s sys 0m19.434s
real 0m16.799s	real 0m19.998s	real 0m22.143s

user 0m1.079s sys 0m15.691s	user 0m0.679s sys 0m19.298s	user 0m2.788s sys 0m19.313s
real 0m17.524s user 0m1.343s sys 0m16.145s	real 0m21.028s user 0m1.022s sys 0m19.973s	real 0m22.894s user 0m2.903s sys 0m19.946s

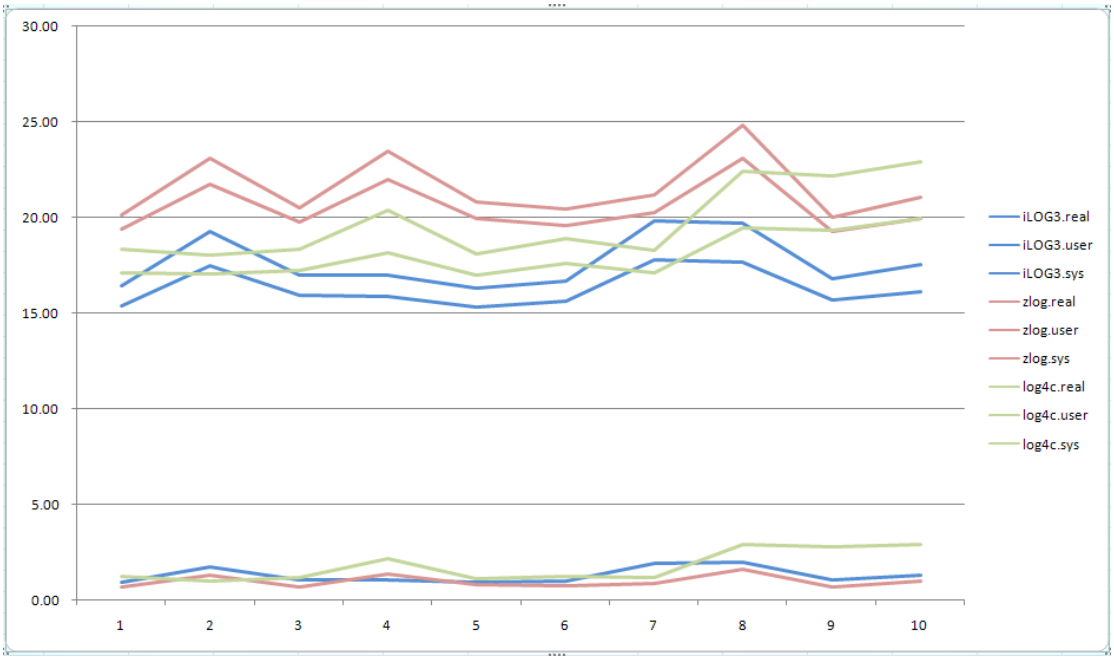
把以上数据摘录到三个文件中用命令处理

```
cat iLOG3.perf | awk '{print $2}' | tr 'ms' '  ' | awk '{print $2}' | while read F1 ; do read F2 ; read F3 ; read F4 ; printf "%s %s %s\n" $F1 $F2 $F3; done
```

数据转置后导入 excel

	A	B	C	D
1		real	user	sys
2	iLOG3	16.416	0.964	15.413
3		19.263	1.754	17.475
4		17.015	1.053	15.933
5		17.014	1.074	15.897
6		16.31	0.958	15.323
7		16.702	1.023	15.648
8		19.848	1.976	17.815
9		19.746	2.01	17.688
10		16.799	1.079	15.691
11		17.524	1.343	16.145
12				
13	zlog	20.124	0.665	19.421
14		23.113	1.319	21.74
15		20.518	0.696	19.786
16		23.463	1.404	22.002
17		20.821	0.799	19.977
18		20.405	0.732	19.606
19		21.158	0.842	20.257
20		24.842	1.648	23.125
21		19.998	0.679	19.298
22		21.028	1.022	19.973
23				
24	log4c	18.377	1.263	17.085
25		18.061	0.988	17.033
26		18.397	1.159	17.198
27		20.384	2.166	18.171
28		18.138	1.118	16.978
29		18.892	1.265	17.577
30		18.279	1.147	17.081
31		22.422	2.918	19.434
32		22.143	2.788	19.313
33		22.894	2.903	19.946

画耗时曲线图，横轴为测试序号，纵轴为耗时（秒）



可见总耗时 iLOG3 最少，其次是 log4c，最多的是 zlog

用户态耗时控制的最好的是 zlog，iLOG3 和 log4c 差不多其次

系统调用耗时 iLOG3 最少，log4c 略多，zlog 最多

另外在测试过程中发现，日志文件增长中把它移走或删除，log4c 不会重建日志文件，zlog 和 iLOG3 会马上重建，怀疑 log4c 每次写完日志行后没有关闭文件，依经验，每次关闭文件比不关闭文件耗时一般多一倍耗时，由此大致可以最后判定

当可靠情况下的性能，log4c 是三者中最低的，iLOG3 略占性能优势。zlog 因为实现功能较多，速度稍慢，希望 zlog 作者能继续优化，下一版赶超我，呵呵。

9.2 功能比较

	iLOG3	zlog	log4c
概念模型	一个日志句柄对应一个日志文件、一个过滤等级的简单模型；如需也可以使用日志句柄集合	syslog 分类模型	“记录器-输出器-布局器”和继承的 log4j 模型
日志等级	有 5 级	可自定义	有 11 级
可外部文件配置	有，自带的简单格式配置文件，后续开发的基于 SML 标记语言格式	有，自带检查器；与函数库整合在一起，无依	有；与函数库整合在一起，无依赖库

	配置文件（依赖库较多），基于配置辅助函数可以开发更多配置格式的配置文件	赖库	
配置文件格式	自带的简单配置格式；SML 或支持第三方开发	只有自带格式	只有 XML
可接口 API 配置日志环境	有	无	有
行日志风格配置	基于方案	完全自定义	完全自定义
写十六进制块日志	有	有	（未知）
输出介质	标准输出、标准错误输出、文件、syslog 或 WINDOWS EVENT、自定义输出对象回调函数	标准输出、标准错误输出、文件、syslog、自定义输出对象回调函数	更全
输出文件名表达	可以包含环境变量	可以包含环境变量、日期时间等	可以包含环境变量、日期时间等
日志转档	可以按照日志文件大小、每天、每小时三种方案转档	只能按照日志文件大小转档	多种转档选择
日志缺失时重建	有	有	（或许无）
线程安全	是	是	是
线程 MDC	有	有	（应该有）
全局线程安全日志对象	WINDOWS、AIX 和 Linux 上支持	有	（未知）
自定义行日志风格回调函数	有	无	（未知）
自定义日志打开、输出、关闭回调函数	全有	只有日志输出回调函数	（未知）
代码规模(.c)	约 2500 行	约 6612 行	约 3638 行

代码规模(.h)	约 550 行	约 1185 行	约 2566 行
库文件大小(.so）	约 90 KB	约 237 KB	约 212 KB
库文件大小(.a）	约 110 KB	约 361 KB	约 339 KB
使用接口方式	根据项目场景和日志规模需求，可以通过日志句柄层 API 或日志句柄集合层 API 来使用，也可以通过外部配置文件使用	只能通过外部配置文件来使用	通过 XML 配置文件或函数 API 来使用
跨操作系统平台	原生支持 UNIX&Linux&WINDOWS 三大主流平台，自带编译脚本和工程文件	原生不支持 WINDOWS，需要第三方移植	原生不支持 WINDOWS，需要第三方移植
其它依赖	有一个标准 c 编译器就行&& POSIX 标准 pthread	POSIX 标准 pthread，还要一个 C99 兼容的 vsnprintf	（未知）
其它优势	•国人开发，中文资料努力编写中 •源代码文件只有三对，可选择性编译	•国人开发，中英文资料齐全 • 函数库自诊断 •用户自定义多少条日志后 fsync 数据到硬盘	• 标准 c 里的老牌经典日志函数库，虽然国外人开发，但网上中文资料也颇多

总结：崇尚老牌经典的选 log4c、需要功能丰富的用 zlog，喜欢轻便&多层选择使用&原生跨平台的直接拖走 iLOG3

10 源代码包结构

```
iLOG3-x.x.x.tar.gz
|   AUTHORS
|   ChangeLog
|   INSTALL
|   LICENSE
|   makefile
|   makefile.Linux
|   makefile.AIX
```

```
|   makefile.MinGW
|   README
├──doc /* 文档主目录 */
|       iLOG3 日志函数库参考手册.doc
|       iLOG3 日志函数库用户指南.doc
├──src /* 源代码主目录 */
|   |   LOG.c /* 日志句柄实现源代码文件 */
|   |   LOG.h /* 日志句柄头文件 */
|   |   LOGS.c /* 日志句柄集合实现源代码文件 */
|   |   LOGS.h /* 日志句柄集合头文件 */
|   |   makefile
|   |   makefile.Linux /* Linux 环境编译 makefile */
|   |   makefile.AIX /* AIX 环境编译 makefile */
|   |   makefile.MinGW /* WINDOWS-MinGW 环境编译 makefile */
|   └──vc6
|       |   vc6.dsw /* WINDOWS-VC6 工作区文件，用于 VC6 编译 */
|       |   vc6.ncb
|       |   vc6.opt
|       ├──iLOG3
|       |   |   iLOG3.dsp
|       |   |   iLOG3.plg
|       └──iLOG3_a
|           |   LOG_a.dsp
|           |   LOG_a.plg
└──test /* 测试例程主目录 */
    |   makefile
|   |   makefile.Linux
|   |   makefile.AIX
|   |   makefile.MinGW
    |   test_hello.c /* 第一个测试示例 */
    |   test_demo.c /* 更复杂的测试示例 */
    |   test_stylesfunc.c /* 自定义行日志风格回调函数测试 */
    |   test_openfunc.c /* 自定义打开文件回调函数测试 */
    |   test_funny.c /* 一些有趣的选项测试 */
    |   test_press.c /* 单进程单线程压力测试 */
    |   test_press_mpt.c /* 多进程多线程压力测试 */
    |   test_press_mpt_tls.c /*基于线程本地存储的缺省全局日志句柄的多进程多线程压力测试*/
    |   test_memoryleak.c /*内存泄露测试 */
    └──vc6
        |   vc6.dsw /* 测试例程 WINDOWS-VC6 工作区文件，用于 VC6 编译*/
        |   vc6.ncb
        |   vc6.opt
        ├──test_hello
        |   |   test_hello.dsp /* test_hello 例程 vc6 工程文件，下同 */
```

```
|   |   test_hello.plg
|   |   test_mode.log
├──test_memoryleak
|   |   test_memoryleak.dsp
|   |   test_memoryleak.plg
├──test_openfunc
|   |   test_openfunc.dsp
|   |   test_openfunc.plg
├──test_press
|   |   test_press.dsp
|   |   test_press.plg
├──test_press_mpt
|   |   test_press_mpt.dsp
|   |   test_press_mpt.plg
├──test_press_mpt_tls
|   |   test_press_mpt_tls.dsp
|   |   test_press_mpt_tls.plg
└──test_stylesfunc
    |   test_stylesfunc.dsp
    |   test_stylesfunc.plg
```