zlog[1] User's Guide

Hardy Simpson[2][3]

November 30, 2022

[1]A single spark can start a prairie fire – Mao Zedong

[2]This Guide is for zlog v1.2.*

[3]If you have comments or error corrections, post a issue on github, or write email to HardySimpson1984@gmail.com

Contents

Chapter 1

What is zlog?

zlog is a reliable, high-performance, thread safe, flexible, clear-model, pure C logging library.

Actually, in the C world there was NO good logging library for applications like logback in java or log4cxx in c++. printf can work, but can not be redirected easily nor be reformatted. syslog is slow and is designed for system use.

So I wrote zlog.

It is faster, safer and more powerful than log4c. So it can be widely used.

zlog has these features:

- syslog model, better than log4j model
- log format customization
- multiple output, include static file path, dynamic file path, stdout, stderr, syslog, user-defined ouput
- runtime with manual or automatic refresh of configuration (done safely)
- high-performance, 250'000 logs/second on my laptop, about 1000 times faster than syslog(3) with rsyslogd
- user-defined log level
- safely rotate log file under multiple-process or multiple-thread conditions
- accurate to microseconds
- dzlog, a default category log API for easy use

- MDC, a log4j style key-value map
- self debuggable, can output zlog's self debug and error log at runtime
- Does not depend on any other 3rd party library, just base on POSIX system (including pthread) and a C99 compliant vsnprintf.

Links:
Homepage: http://hardysimpson.github.com/zlog
Downloads: https://github.com/HardySimpson/zlog/releases
Author's Email: HardySimpson1984@gmail.com

1.1 Compatibility Notes

1. zlog is based on POSIX-compatible systems. I have just GNU/linux and AIX environments to compile, test and run zlog. Still, I think zlog will work well on FreeBSD, NetBSD, OpenBSD, OpenSolaris, Mac OS X etc. Test runs of zlog on any system are welcome.

2. zlog uses a feature of C99 compliant vsnprintf. That is, if the buffer size of destination is not long enough, vsnprintf will return the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. If the vsnprintf on your system does not work like that, zlog can not know the right buffer size when a single log is longer than the buffer. Fortunately, glibc 2.1, libc on AIX, and libc on freebsd work correctly, while glibc 2.0 does not. In this case, user should crack zlog himself with a C99 compliant vsnprintf. I suggest ctrio, or C99-snprintf. The file buf.c should be cracked, good luck!

3. Some people offer versions of zlog for other platforms. Thanks!

   auto tools version: https://github.com/bmanojlovic/zlog

   cmake verion: https://github.com/lisongmin/zlog

   windows version: https://github.com/lopsd07/WinZlog

1.2 zlog 1.2 Release Notes

1. zlog 1.2 provides these features:

   (a) support for pipeline. Now zlog can send ouput log through programs like cronolog
   (b) Full rotation support, see 5.6
   (c) Other code compatible details, bug fixes.

2. zlog 1.2 is binary compatible with zlog 1.0. The differences are:

   (a) All zlog macros like ZLOG_INFO are shifted to lowercase versions, zlog_info. This big change is because I think it is easier for people to type. If you are using a previous version of zlog, please use a script to substitute all macros, and re-compile your program. Here is an example:

   ```
   sed -i -e 's/\b\w*ZLOG\w*\b/\L&\E/g' aa.c
   ```

   (a) Auto tools compile is abandoned. Auto tools is ugly so I dropped it. A simple makefile is in use, which requires gcc and gnu make. If this makefile does not work in your environment, you will need to write a suitable makefile for yourself. It should be quite easy for a geek.

Chapter 2

What zlog is not

The goal of zlog is to be a simple, fast log library for applications. It does not support output like sending the log to another machine through the net or saving it to database. It will not parse content of log and filter them.

The reason is obvious: the library is called by an application, so all time taken by the log library is part of the application's time. And database inserting or log content parsing takes a long time. These will slow down the application. These operation should be done in a different process or on a different machine.

If you want all these features, I recommend rsyslog, zLogFabric, Logstash. These have independent processes to receive logs from another process or machine, and to parse and store logs. These functions are separated from the user application.

Now 7.4 is supported by zlog. Just one output function need to be implemented: to transfer the log to the other process or machine. The work of category matching and log generating is left with the zlog library.

One possibility is to write a zlog-redis client. It send logs to redis on local or remote machines by user defined output. Then other processes can read logs from redis and write to disk. What do you think about this idea? I will be happy to discuss it with you.

Chapter 3

Hello World

3.1 Build and Installation zlog

Download:https://github.com/HardySimpson/zlog/archive/latest-stable.tar.gz

```
$ tar -zxvf zlog-latest-stable.tar.gz
$ cd zlog-latest-stable/
$ make
$ sudo make install
or
$ sudo make PREFIX=/usr/local/ install
```

PREFIX indicates where zlog is installed. After installation, change system settings to make sure your program can find the zlog library

```
$ sudo vi /etc/ld.so.conf
/usr/local/lib
$ sudo ldconfig
```

Before running a real program, make sure libzlog.so is in the directory where the system's dynamic lib loader can find it. The commands mentioned above are for linux. Other systems will need a similar set of actions.

- Beside the normal make, these are also available:

```
$ make 32bit # 32bit version on 64bit machine, libc6-dev-i386 is needed
$ make noopt # without gcc optimization
$ make doc   # lyx and hevea is needed
$ make test  # test code, which is also good example for zlog
```

- makefile of zlog is written in gnu make style. So if your platform is not linux, install a gnu make and gcc before trying to build zlog. Another way is to write a makefile in your platform's make style. This should be quite easy as zlog is not complicated.

3.2 Call and Link zlog in User's application

To use zlog, add one line to the source c file or cpp file:

```
#include "zlog.h"
```

zlog needs the pthread library. The link command is:

```
$ cc -c -o app.o app.c -I/usr/local/include
    # -I[where zlog.h is put]
$ cc -o app app.o -L/usr/local/lib -lzlog -lpthread
    # -L[where libzlog.so]
```

3.3 Hello World Example

This example can be found in $(top_builddir)/test/test_hello.c, test_hello.conf

1. Write a new c source file:

```
$ vi test_hello.c
#include <stdio.h>
#include "zlog.h"

int main(int argc, char** argv)
{
    int rc;
    zlog_category_t *c;
    rc = zlog_init("test_hello.conf");
    if (rc) {
            printf("init failed\n");
            return -1;
        }
        c = zlog_get_category("my_cat");
        if (!c) {
            printf("get cat fail\n");
            zlog_fini();
            return -2;
        }
        zlog_info(c, "hello, zlog");
        zlog_fini();
        return 0;
    }
```

2. Write a configuration file in the same path as test_hello.c:

```
$ vi test_hello.conf
[formats]
simple = "%m%n"
[rules]
my_cat.DEBUG    >stdout; simple
```

3. Compile and run it:

```
$ cc -c -o test_hello.o test_hello.c -I/usr/local/include
$ cc -o test_hello test_hello.o -L/usr/local/lib -lzlog
$ ./test_hello
hello, zlog
```

3.4 Simpler Hello World Example

This example can be found in $(top_builddir)/test/test_default.c, test_default.conf. The source code is

```
#include <stdio.h>
#include "zlog.h"
int main(int argc, char** argv)
{
        int rc;
        rc = dzlog_init("test_default.conf", "my_cat");
        if (rc) {
            printf("init failed\n");
            return -1;
        }
        dzlog_info("hello, zlog");
        zlog_fini();
        return 0;
    }
```

The configure file test_default.conf is the same as test_hello.conf, and the output of test_default is the same as that of test_hello. The difference is that test_default uses the dzlog API, which has a default *zlog_category_t* inside and is easier to use. See 6.5 for more details.

Chapter 4

Syslog model

4.1 Category, Rule and Format

In zlog, there are 3 important concepts: category, rule and format.

Category specifies different kinds of log entries. In the zlog source code, category is a (zlog_cateogory_t *) variable. In your program, different categories for the log entries will distinguish them from each other.

Format describes detail log patterns, such as: with or without time stamp, source file, source line.

Rule consists of category, level, output file (or other channel) and format. In brief, if the category string in a rule in the configuration file equals the name of a category variable in the source, then they match.

So when this sentence in the source file is executed:

```
zlog_category_t *c;
c = zlog_get_category("my_cat");
zlog_info(c, "hello, zlog");
```

zlog library uses the category name "my_cat" to match one rule in the configuration file. That is

```
[rules]
my_cat.DEBUG    >stdout; simple
```

Then the library will check if level is correct to decide whether the log will be output or not. As INFO>=DEBUG the log will be output, and as the rule says, it will be sent to stdout (standard output) in the format of simple, which is described as

```
[formats]
simple = "%m%n"
```

Lastly, zlog will show the zlog_info() content on the screen

```
hello, zlog
```

That's the whole story. The only thing a user need to do is to write the messages. Where the log will be output, or in which format, is done by zlog library.

4.2 Differences between syslog model and log4j model

Does zLog have anything to do with syslog? Until now, the model is more like log4j. As in log4j, there are concepts of logger, appender and layout. The difference is that in log4j, each logger in source code must correspond to one logger in the configuration file and has just one definite level. One-to-one relationship is the only choice for log4j, log4cxx, log4cpp, log4cplus log4net and etc...

But the log4j model is NOT flexible, they invent filters to make up for it, and that make things more worse. So let's get back to syslog model, which has an excellent design.

Continuing our example from the last section, if the zlog configuration file has 2 rules:

```
[rules]
my_cat.DEBUG     >stdout; simple
my_cat.INFO      >stdout;
```

Then they will generate 2 log outputs to stdout:

```
hello, zlog
2012-05-29 10:41:36 INFO [11288:test_hello.c:41] hello, zlog
```

You see that one category in the source code corresponds to two rules in the configuration file. Maybe log4j's user will say, "That's good, but 2 appender for one logger will do the same thing". So, let's see the next example of configure file:

```
[rules]
my_cat.WARN     "/var/log/aa.log"
my_cat.DEBUG    "/var/log/bb.log"
```

And the source code is:

```
zlog_info(c, "info, zlog");
zlog_debug(c, "debug, zlog");
```

Then, in aa.log, there is just one log

```
2012-05-29 10:41:36 INFO [11288:test_hello.c:41] info, zlog
```

But in bb.log, there will be two

```
2012-05-29 10:41:36 INFO [11288:test_hello.c:41] info, zlog
2012-05-29 10:41:36 DEBUG [11288:test_hello.c:42] debug, zlog
```

From this example, you see the difference. Log4j can not do it easily. In zlog, one category may correspond to mutiple rules, and rules can have different level, output, and format combinations. The user has an easy, clear way to filter and multi-ouput all logs on demand.

4.3 Expand syslog model

You can see that category in zlog is more like facility in syslog. Unfortunately, facility in sylog is an int, and the value of facility must be chosen from a limited system-defined range. zlog does better, making it a string variable.

In syslog, there is a special wildcard "*", which matches all facilities. It does the same thing in zlog. "*" matches all categories. That is a convenient way to make all errors generated by multiple components in your system redirect to one log file. Just write in the configuration file like this:

```
[rules]
*.error    "/var/log/error.log"
```

A unique feature of zlog is sub-category matching. If your source code has:

```
c = zlog_get_category("my_cat");
```

And the configuration file has rules :

```
[rules]
my_cat.*      "/var/log/my_cat.log"
my_.NOTICE    "/var/log/my.log"
```

These 2 rules match category "c" with the name "my_cat". The wildcard "_" is the way to represent a super category. "my_" is a super category for "my_cat" and "my_dog". There is also another wildcard "!". See 5.5.2 for more detail.

Chapter 5

Configure File

Most actions of zlog library are dependent upon the configuration file: where to output the log, how to rotate the log files, how to format the output, etc. The configuration file uses a domain specific language to control the library actions. Here is an example of zlog.conf:

```
# comments
[global]
strict init = true
reload conf period = 1M
buffer min = 1024
buffer max = 2MB
rotate lock file = /tmp/zlog.lock
default format = "%d.%ms %-6V (%c:%F:%L) - %m%n"
file perms = 600
fsync period = 1K

[levels]
TRACE = 10
CRIT = 130, LOG_CRIT

[formats]
simple = "%m%n"
normal = "%d(%F %T) %m%n"

[rules]
default.*                >stdout; simple
```

```
*.*                     "%12.2E(HOME)/log/%c.log", 1MB*12; simple
my_.INFO                >stderr;
my_cat.!ERROR           "/var/log/aa.log"
my_dog.=DEBUG           >syslog, LOG_LOCAL0; simple
my_mice.*               $user_define;
```

[] means a section's beginning, and the order of sections is fixed, using the sequence global-levels-formats-rules.

Note on units: when memory size or large number is needed, it is possible to specify it in the usual form of 1k 5GB 4M and so forth:

```
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 byte
```

units are case insensitive so 1GB 1Gb 1gB are all the same.

5.1 Global

Global section begins with [global]. This section can be omitted.The syntax is

```
(key) = (value)
```

- strict init

  If "strict init = true", zlog_init() will check syntax of all formats and rules strictly, and any error will cause zlog_init() to fail and return -1. When "strict init = false", zlog_init() will ignore syntax errors for formats and rules. The default is true.

- reload conf period

  This parameter causes the zlog library to reload the configuration file automatically after a period, which is measured by number of log times per process. When the number reaches the value, it calls zlog_reload() internally. The number is reset to zero at the last zlog_reload() or zlog_init(). As zlog_reload() is atomic, if zlog_reload() fails, zlog still runs with the current configuration. So reloading automatically the configuration is safe. The default is 0, which means never reload automatically.

- buffer min
- buffer max

  zlog allocates a log buffer in each thread. "buffer min" indicates size of buffer malloc'ed at init time. While logging, if one single log's content is longer than buffer size now, zlog will expand buffer automatically until "buffer max". Then, if the size is still longer than "buffer max", the log content will be truncated. If "buffer max" is 0, it means buffer size is unlimited, and each time zlog will expand buffer by twice its size, until the process uses all available memory. The value of these 2 parameters can appended with unit KB, MB or GB suffix, where 1024 equals 1KB. As default, "buffer min" is 1K and "buffer max" is 2MB.

- rotate lock file

  This specifies a lock file for rotating a log safely in multi-process situations. zlog will open the file at zlog_init() with the permission of read-write. The pseudo-code for rotating a log file is:

  ```
  write(log_file, a_log)
  if (log_file > 1M)

      if (pthread_mutex_lock succ && fcntl_lock(lock_file) succ)
         if (log_file > 1M) rotate(log_file);
         fcntl_unlock(lock_file);
         pthread_mutex_unlock;
  ```

  mutex_lock is for multi-thread and fcntl_lock is for multi-process. fcntl_lock is the POSIX advisory record locking. See man 3 fcntl for details. The lock is system-wide, and when a process dies unexpectedly, the operating system releases all locks owned by the process. That's why I chose fcntl lock for rotating log safely. The process needs read-write permisson for lock_file to lock it.

  By default, rotate lock file = self. This way, zlog does not create any lock file and sets the configuration file as the lock file. As fcntl is advisory, it does not really forbid programmers to change and store the configuration file. Generally speaking, one log file will not be rotated by processes run by different operating system users, so using the configuration file as lock file is safe.

If you choose another path as lock file, for example, /tmp/zlog.lock, zlog will create it at zlog_init(). Make sure your program has permission to create and read-write the file. If processes run by different operating system users need to write and rotate the same log file, make sure that each program has permission to create and read-write the same lock file.

- default format

  This parameter is used by rules without format specified. The default is

  ```
  "%d %V [%p:%F:%L] %m%n"
  ```

  It will yield output like this:

  ```

  2012-02-14 17:03:12 INFO [3758:test_hello.c:39] hello, zlog
  ```

- file perms

  This specifies all log file permissions when they are created. Note that it is affected by user's umask. The final file permission will be "file perms" & ~umask. The default is 600, which just allows user read and write.

- fsync period

  After a number of log times per rule (to file only), zlog will call fsync(3) after write() to tell the Operating System to write data to disk immediately from any internal system buffers. The number is incremented by each rule and will be reset to 0 after zlog_reload(). Note that when the file's path is generated dynamically or is rotated, zlog does not guarantee fsync() touch all files. It just does fsync() against the file descriptors that have have seen write() prior to the boundary time. It offers a balance between speed and data safety. An example:

  ```
  $ time ./test_press_zlog 1 10 100000
  real 0m1.806s
  user 0m3.060s
  sys  0m0.270s

  $ wc -l press.log
  1000000 press.log
  ```

```
$ time ./test_press_zlog 1 10 100000 #fsync period = 1K
real 0m41.995s
user 0m7.920s
sys  0m0.990s

$ time ./test_press_zlog 1 10 100000 #fsync period = 10K
real 0m6.856s
user 0m4.360s
sys  0m0.550s
```

If you want extreme safety but do not care about speed, use synchronous file I/O, see 5.5.3.The defualt is 0, which means let the operating system flush the output buffer when it wants.

5.2 Levels

This section begins with [levels] and allows the user to define application levels. You should match these values with user-defined macros in the source file. This section can be omitted.

The syntax is

```
(level string) = (level int), (syslog level, optional)
```

level int should in [1,253], higher numbers mean more important. syslog level is optional, if not set, use LOG_DEBUG

see 7.3 for more details.

5.3 Formats

This section begins with [formats], where the user can define preferred log patterns. The syntax is

```
(name) = "(actual formats)"
```

It is easy to understand, (name) will be used in the next section [rules]. The format (name) consists of letters and digits plus underscore "_". The (actual format) should be put in double quotes. It can be built up with conversion patterns, as described below.

5.4 Conversion pattern

The conversion pattern is closely related to the conversion pattern of the C printf function. A conversion pattern is composed of literal text and format control expressions called conversion specifiers.

Conversion pattern is used in both filepath of rule and pattern of format.

You are free to insert any literal text within the conversion pattern.

Each conversion specifier starts with a percent sign (%) and is followed by optional format modifiers and a conversion character. The conversion character specifies the type of data, e.g. category, level, date, thread id. The format modifiers control such things as field width, padding, left and right justification. The following is a simple example.

Let the conversion pattern be

```
"%d(%m-%d %T) %-5V [%p:%F:%L] %m%n".
```

Then the statement

```
zlog_info(c, "hello, zlog");
```

would yield the output

```
02-14 17:17:42 INFO  [4935:test_hello.c:39] hello, zlog
```

Note that there is no explicit separator between text and conversion specifiers. The pattern parser knows when it has reached the end of a conversion specifier when it reads a conversion character. In the example above the conversion specifier %-5p means the level of the logging event should be left justified to a width of five characters.

5.4.1 Conversion Characters

The recognized conversion characters are

conversion char	effect	example

%c	Used to output the category of the logging event.	aa_bb
%d()	Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between parentheses. For example, %d(%F) or %d(%m-%d %T). If no date format specifier is given then %d(%F %T) format is assumed. The date format specifier permits the same syntax as the strftime(2). see 5.4.3for more detail.	%d(%F) 2011-12-01 %d(%m-%d %T) 12-01 17:17:42 %d(%T).%ms 17:17:42.035 %d 2012-02-14 17:03:12 %d() 2012-02-14 17:03:12
%E()	Value of environment variables	%E(LOGNAME) simpson
%ms	The millisecond, 3-digit integer string comes from gettimeofday(2)	013
%us	The microsecond, 6-digit integer string comes from gettimeofday(2)	002323
%F	Used to output the file name where the logging request was issued. The file name comes from __FILE__ macro. Some compilers supply __FILE__ as the absolute path. Use %f to strip path and keep the file name. Some compilers have an option to control this feature.	test_hello.c or, under some compiler /home/zlog/src/test/test_hello.c
%f	Used to output the source file name, the string after the last '/' of $F. It will cause a little performance loss in each log event.	test_hello.c
%g()	Used to output the date of the logging event in UTC in stead of local time.	%g(%F) 2011-12-01 %g(%m-%d %T) 12-01 15:17:42 %g(%T.ms) 15:17:42.035 %g 2012-02-14 15:03:12 %g() 2012-02-14 15:03:12
%H	Used to output the hostname of system, which is from gethostname(2)	zlog-dev

%k	Used to output the kernel thread id. On Linux, that's the LWP using syscall(SYS_gettid) and on OSX, pthread_threadid_np.	2136
%L	Used to output the line number from where the logging request was issued, which comes from __LINE__ macro	135
%m	Used to output the application supplied message associated with the logging event.	hello, zlog
%M	Used to output the MDC (mapped diagnostic context) associated with the thread that generated the logging event. The M conversion character must be followed by the key for the map placed between parenthesis, as in %M(clientNumber) where clientNumber is the key. The value in the MDC corresponding to the key will be output.See 7.1 for more detail.	%M(clientNumber) 12345
%n	Outputs unix newline character, zLog does not support the MS-Windows line separator at this time.	\n
%p	Used to output the id of the process that generated the logging event, which comes from getpid().	2134
%U	Used to output the function name where the logging request was issued. It comes from __func__(C99) or __FUNCTION__(gcc) macro, with the support of the compiler.	main
%V	Used to output the level of the logging event, uppercase.	INFO
%v	Used to output the level of the logging event, lowercase.	info

%t	Used to output the hexadecimal form of the thread id that generated the logging event, which comes from pthread_self(). "%x",(unsigned int) pthread_t	ba01e700
%T	Equivalent to %t, but the long form "%lu", (unsigned long) pthread_t	140633234859776
%%	the sequence %% outputs a single percent sign.	%
%[other char]	parsed as a wrong syntax, will cause zlog_init() fail	

5.4.2 Format Modifier

By default, the relevant information is output as-is. However, with the aid of format modifiers it is possible to change the minimum field width, the maximum field width, and justification. It will cause a little performance loss in each log event.

The optional format modifier is placed between the percent sign and the conversion character.

The first optional format modifier is the left justification flag which is just the minus (-) character. Then comes the optional minimum field width modifier. This is a decimal constant that represents the minimum number of characters to output. If the data item requires fewer characters, it is padded on either the left or the right until the minimum width is reached. The default is to pad on the left (right justify) but you can specify right padding with the left justification flag. The padding character is space. If the data item is larger than the minimum field width, the field is expanded to accommodate the data. The value is never truncated.

This behavior can be changed using the maximum field width modifier which is designated by a period followed by a decimal constant. If the data item is longer than the maximum field, then the extra characters are removed from the beginning of the data item and not from the end. For example, if the maximum field width is eight and the data item is ten characters long, then the last two characters of the data item are dropped. This behavior equals the printf function in C where truncation is done from the end.

Below are various format modifier examples for the category conversion specifier.

format modifier	left justify	minimum width	maximum width	comment
%20c	false	20	none	Left pad with spaces if the category name is less than 20 characters long.
%-20c	true	20	none	Right pad with spaces if the category name is less than 20 characters long.
%020c	false	20	none	Left pad with 0's if the category name is less than 20 characters long.
%.30c	NA	none	30	Truncate from the end if the category name is longer than 30 characters.
%20.30c	false	20	30	Left pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the end.
%-20.30c	true	20	30	Right pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the end.

5.4.3 Time Character

Here is the Time Character support by Conversion Character *d*.

All Character is supported by strftime(3) in library. The Character supported on my linux system are

character	effect	example
%a	The abbreviated weekday name according to the current locale.	Wed
%A	The full weekday name according to the current locale.	Wednesday
%b	The abbreviated month name according to the current locale.	Mar
%B	The full month name according to the current locale.	March
%c	The preferred date and time representation for the current locale.	Thu Feb 16 14:16:35 2012

%C	The century number (year/100) as a 2-digit integer. (SU)	20
%d	The day of the month as a decimal number (range 01 to 31).	06
%D	Equivalent to %m/%d/%y. (for Americans only. Americans should note that in other countries %d/%m/%y is more common. This means that in an international context this format is ambiguous and should not be used.) (SU)	02/16/12
%e	Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space. (SU)	6
%F	Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)	2012-02-16
%G	The ISO 8601 week-based year (see NOTES) with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %Y, except that if the ISO week number belongs to the previous or next year, that year is used instead. (TZ)	2012
%g	Like %G, but without century, that is, with a 2-digit year (00-99). (TZ)	12
%h	Equivalent to %b. (SU)	Feb
%H	The hour as a decimal number using a 24-hour clock (range 00 to 23).	14
%I	The hour as a decimal number using a 12-hour clock (range 01 to 12).	02
%j	The day of the year as a decimal number (range 001 to 366).	047
%k	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.) (TZ)	15
%l	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.) (TZ)	3
%m	The month as a decimal number (range 01 to 12).	02

%M	The minute as a decimal number (range 00 to 59).	11
%n	A newline character. (SU)	\n
%p	Either "AM" or "PM" according to the given time value, or the corresponding strings for the current locale. Noon is treated as "PM" and midnight as "AM".	PM
%P	Like %p but in lowercase: "am" or "pm" or a corresponding string for the current locale. (GNU)	pm
%r	The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to %I:%M:%S %p. (SU)	03:11:54 PM
%R	The time in 24-hour notation (%H:%M). (SU) For a version including the seconds, see %T below.	15:11
%s	The number of seconds since the Epoch, that is, since 1970-01-01 00:00:00 UTC. (TZ)	1329376487
%S	The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)	54
%t	A tab character. (SU)	
%T	The time in 24-hour notation (%H:%M:%S). (SU)	15:14:47
%u	The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w. (SU)	4
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sun- day as the first day of week 01. See also %V and %W.	07
%V	The ISO 8601 week number (see NOTES) of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year. See also %U and %W. (SU)	07
%w	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.	4
%W	The week number of the current year as a decimal number, range 00 to 53, starting with the first Mon- day as the first day of week 01.	07
%x	The preferred date representation for the current locale without the time.	02/16/12

%X	The preferred time representation for the current locale without the date.	15:14:47
%y	The year as a decimal number without a century (range 00 to 99).	12
%Y	The year as a decimal number including the century.	2012
%z	The time-zone as hour offset from GMT. Required to emit RFC 822-conformant dates (using "%a, %d %b %Y %H:%M:%S %z"). (GNU)	+0800
%Z	The timezone or name or abbreviation.	CST
%%	A literal '%' character.	%

5.5 Rules

This section begins with [rules]. It decides how log actions are filtered, formatted and output. This section can be omitted, but there will result in no log output. The syntax is

```
    (category).(level)    (output), (option,optional); (format name, optional)
```

When zlog_init() is called, all rules will be read into memory. When zlog_get_category() is called, mutiple rules will be assigned to each category, in the way 5.5.2 describes. When logging is performed, the level between matched rules and INFO will be checked to decide whether this single log will be output through the rule. When zlog_reload() is called, the configuration file will be re-read into memory, including rules. All category rules will be re-calculated.

5.5.1 Level Matching

There are six default levels in zlog, "DEBUG", "INFO", "NOTICE", "WARN", "ERROR" and "FATAL". As in all other log libraries, aa.DEBUG means all logs of level greater than or equal to DEBUG will be output. Still, there are more expressions. Levels in the configuration file are not case sensitive; both capital or lowercase are accepted.

example expression	meaning
*	all [source level]
aa.debug	[source level]>=debug
aa.=debug	[source level]==debug
aa.!debug	[source level]!=debug

The level strings can be defined by the user. See7.3.

5.5.2 Category Matching

Category Matching is simple. The name of the category is made up of letters, digits, and/or the underscore "_".

summarize	category string from configure file	category matches	category not matches
* matches all	*.*	aa, aa_bb, aa_cc, xx, yy ...	NONE
string end with underline matches super-category and sub-categories	aa_.*	aa, aa_bb, aa_cc, aa_bb_cc	xx, yy
string not ending with underline accurately matches category	aa.*	aa	aa_bb, aa_cc, aa_bb_cc
! matches category that has no rule matched	!.*	xx	aa(as it matches rules above)

5.5.3 Output Action

zlog supports various output methods. The syntax is

```
(output action), (output option); (format name, optional)
```

output	output action	output option
to standard out	>stdout	no meaning
to standard error	>stderr	no meaning
to syslog	>syslog	syslog facility, can be: LOG_USER(default), LOG_LOCAL[0-7] This is required.
pipeline output	| cat	no meaning
to file	"(file path)"	rotation. see 5.6 for detail 10MB * 3 ~ "press.#r.log"
synchronous I/O file	-"(file path)"	
user-defined output	$name	"path" (dynamic or static) of record function

- stdout, stderr, syslog

  As the above table describes, only the syslog action has a meaningful output option and it must be set.

  Warning: NEVER use >stdout or >stderr when your program is a daemon process. A daemon process always closes its first file descriptor, and when >stdout is set, zlog will output a log like this

  ```
  write(STDOUT_FILENO, zlog_buf_str(a_thread->msg_buf), zlog_buf_len(a_thread-
  ```

  What will happen then? The log will be written to the file whose fd is now 1. I have received mail from someone who said zlog as a daemon wrote logs to the configuration file. So remember, daemon processes should not set any rule output to stdout, or stderr. It will generate undefined behavior. If you still want output logs to console when stdout is closed, use "/dev/tty" instead.

- pipeline output

  ```
  *.*    | /usr/bin/cronolog /www/logs/example_%Y%m%d.log ; normal
  ```

  This is an example of how zlog pipelines its output to cronolog. The implementation is simple. popen("/usr/bin/cronolog /www/logs/example_%Y%m%d.log","w") is called at zlog_init(), and forward logs will be written to the open descriptor in the "normal" format. Writing through pipeline and cronnolog is faster than dynamic file of zlog, as there is no need to open and close file descripter each time when logs are written to a pipe.

```
[rules]
*.*              "press%d(%Y%m%d).log"
$ time ./test_press_zlog 1 10 100000
real 0m4.240s
user 0m2.500s
sys 0m5.460s

[rules]
*.*                  | /usr/bin/cronolog press%Y%m%d.log
$ time ./test_press_zlog 1 10 100000
real 0m1.911s
user 0m1.980s
sys 0m1.470s
```

There are some limitations when using pipeline output:

– POSIX.1-2001 says that write(2)s of less than PIPE_BUF bytes must be atomic, On Linux, PIPE_BUF is 4096 bytes.
– When a single log is longer than PIPE_BUF, and multiple processes write logs through one pipe (parent calls zlog_init(), and forks many child processes), log interlacing will occur.
– Unrelated multiple processes can start multiple cronolog processes and write to the same log file. Even if a single log is not longer than PIPE_BUF, multiple cronologs will cause log interlace. As cronologs read log continuously, it doesn't know where is the split between log entries.

In summary, pipeline to a single log file:

– Single process writing, no limitation for length of one log. Multi-threads in one process, atomic writing is already assured by zlog.
– Related multiple processes, the length of one log should not longer than PIPE_BUF.
– Unrelated multiple processes, no matter how long a single log is, will cause interlace and is not safe.

- file
  - file path

    can be absolute file path or relative file path. It is quoted by double quotation marks. *Conversion pattern* can be used in file path. If the file path is "%E(HOME)/log/out.log" and the program environment $HOME is /home/harry, then the log file will be /home/harry/log/output.log. See 5.4 for more details.

    file of zlog is powerful, for example

    1. output to named pipe(FIFO), which must be created by mkfifo(1) before use

       ```
       *.*   "/tmp/pipefile"
       ```
    2. output to null, do nothing at all

       ```
       *.*    "/dev/null"
       ```
    3. output to console, in any case

       ```
       *.*    "/dev/tty"
       ```
    4. output a log to each tid, in the directory where the process running

       ```
       *.*     "%T.log"
       ```
    5. output to file with pid name, every day, in $HOME/log directory, rotate log at 1GB, keep 5 log files

       ```
       *.*     "%E(HOME)/log/aa.%p.%d(%F).log",1GB * 5
       ```
    6. each category of aa_ super category, output log with category name

       ```
       aa_.*      "/var/log/%c.log"
       ```
  - rotate action

    controls log file size and count. zlog rotates the log file when the file exceeds this value. For example, let the action be

    ```
    "%E(HOME)/log/out.log",1M*3
    ```

    and after out.log is filled to 1M, the rotation is

    ```
    out.log -> out.log.1
    out.log(new create)
    ```

    If the new log is full again, the rotation is

    ```
    out.log.1 -> out.log.2
    out.log -> out.log.1
    out.log(new create)
    ```

The next rotation will delete the oldest log, as *3 means just allows 3 file exist

```
unlink(out.log.2)
out.log.1 -> out.log.2
out.log -> out.log.1
out.log(new create)
```

So the oldest log has the biggest serial number. If *3 is not specified, it means rotation will continue and no old log will be deleted.

– synchronous I/O file

  Putting a minus sign '-' sets the synchronous I/O option. log file is opened with O_SYNC and every single log action will wait until the Operating System writes data to disk. It is painfully slow:

```
$ time ./test_press_zlog 100 1000
real 0m0.732s
user 0m1.030s
sys  0m1.080s
$ time ./test_press_zlog 100 1000  # synchronous I/O open
real 0m20.646s
user 0m2.570s
sys  0m6.950s
```

- format name

  It is optional. If not set, use zlog default format in global setting, which is:

```
[global]
default format = "%d %V [%p:%F:%L] %m%n"
```

- see 7.4 for more details for $.

5.6 Rotation

Why rotation? I have see more than once in a production environment, that the hard disk is full of logs and causes the system to stop working, or a single log file is too big to open or grep. Several ways to rotate and archive log files are possible:

1. Split log by date or time.

   For example, generate one log file per day.

```
aa.2012-08-02.log
aa.2012-08-03.log
aa.2012-08-04.log
```

In this case, the system administrator knows how much log will be produced one day. The sys admin is able to search log files based on the day. The best way to make this split is to let the zlog library do it. Another choice is using cronosplit to analyse the content of log file and split it. A bad way is using crontab+logrotate to daily move log files, which is not accurate, some logs will be put into the file for the previous day.

Using zlog, there is no need for external rotate action to complete the job. Setting time in the log file name works:

```
*.*  "aa.%d(%F).log"
```

or using cronolog for faster performace:

```
*.*  | cronolog aa.%F.log
```

2. Split log by size.

   Always suitable for development use. In this case, the program generates a lot of logs in a short period. But the text editor might not be able to open big files quickly. Although the split can be done using split tools afterwards, this requires extra steps. So a good way is to let the logging library do the rotation. There are two ways of rotation, as nlog describes, Sequence and Rolling. In case of Sequence:

```
aa.log (new)
aa.log.2 (less new)
aa.log.1
aa.log.0 (old)
```

   And in case of Rolling:

```
aa.log (new)
aa.log.0 (less new)
aa.log.1
aa.log.2 (old)
```

It's hard to say which one is most suitable.

If only some of the newest logs are useful to developers, logging library should do the cleanup work and delete the old log files. Some external tools can't find out which files are older.

The simplest rotation configuration for zlog is:

```
*.*     "aa.log", 10MB
```

It is Rolling. When aa.log is larger than 10MB, zlog will rename file like this:

```
aa.log.2 -> aa.log.3
aa.log.1 -> aa.log.2
aa.log.0 -> aa.log.1
aa.log -> aa.log.0
```

The configuration can be more complex:

```
*.*     "aa.log", 10MB * 0 ~ "aa.log.#r"
```

The 1st argument after the file name says when rotation will be triggered, in size.

The 2nd argument after the file name says how many archive files will be kept, (0 means keep all).

The 3rd argument after the file name shows the archive file name. #r is a sequence number for archive files. r is short for Rolling, and #s is short for sequence. Archive file name must contain one of #r or #s.

3. Split log by size, and add time tag to archive file.

```
aa.log
aa.log-20070305.00.log
aa.log-20070501.00.log
aa.log-20070501.01.log
aa.log-20071008.00.log
```

In this case, the log file is not usually viewed frequently, and is likely checked once a day. Of course, when one day's log is more than 100MB, you should consider storing in two files and add postfix numbers. For example if the date is used as part of the pattern (like 20070501):

The configuration of zlog is:

```
*.*     "aa.log", 100MB ~ "aa-%d(%Y%m%d).#2s.log"
Do rotation every 100MB. The archive file name also supports conversion stri
```

4. Compress, move and delete old archive.

   Compress should not be done by the logging library, because compress need time and CPU. The mission of the logging library is to cooperate with compress programs.

   For the 3 ways to split logs, way 1 and way 3 are easy to manage. It is easy to find old log file by name or by modify time. And then compress, move and delete old log files by crontab and shell.

   For second way, compress is useless, delete is needed and zlog already supports it.

   If you really want to rotate and compress log file at the same time, I suggest logrotate. It is an independent program and will not confuse the situation.

5. zlog support for external tools like logrotate.

The rotation support of zlog is very powerful, still there are several cases zlog can not handle. Like rotation by time, before or after rotation call some user-defined shells. That will make zlog too complex.

Under these circumstances, consider using external tools like logrotate. On linux, the problem is that when a tool renames the log file, the working process which uses an inode to reference the file will not automatically reopen the new file. The standard way is send a signal to the program and let it reopen the file. For syslogd the command is:

```
kill -SIGHUP ‘cat /var/run/syslogd.pid‘
```

For zlog as a library, it is not good to receive signals. zlog provide zlog_reload(), which reloads the configuration file and reopens all log files. So if you write a program and want to reopen a log file manually, you can write some code to do the job like this: after receiving a signal or command from client, call zlog_reload().

5.7 Configure File Tools

```
$ zlog-chk-conf -h
Useage: zlog-chk-conf [conf files]...
-q, suppress non-error message
-h, show help message
```

zlog-chk-conf tries to read configuration files, check their syntax, and output to screen whether it is correct or not. I suggest using this tool each time you create or change a configuration file. It will output like this

```
$ ./zlog-chk-conf zlog.conf
03-08 15:35:44 ERROR (10595:rule.c:391) sscanf [aaa] fail, category or level is
03-08 15:35:44 ERROR (10595:conf.c:155) zlog_rule_new fail [aaa]
03-08 15:35:44 ERROR (10595:conf.c:258) parse configure file[zlog.conf] line[126
03-08 15:35:44 ERROR (10595:conf.c:306) zlog_conf_read_config fail
03-08 15:35:44 ERROR (10595:conf.c:366) zlog_conf_build fail
03-08 15:35:44 ERROR (10595:zlog.c:66) conf_file[zlog.conf], init conf fail
03-08 15:35:44 ERROR (10595:zlog.c:131) zlog_init_inner[zlog.conf] fail

---[zlog.conf] syntax error, see error message above
```

This example tells you that [aaa] is not a correct rule and that line 126 in the configuration file is wrong. Later failure reports result from that fundamental failure.

Chapter 6

zlog API

All zlog APIs are thread safe. To use them, you just need to

```
#include "zlog.h"
```

6.1 initialize and finish

SYNOPSIS

```
int zlog_init(const char *confpath);
int zlog_reload(const char *confpath);
void zlog_fini(void);
```

DESCRIPTION zlog_init() reads configuration from the file confpath. If confpath is NULL, it looks for the environment variable ZLOG_CONF_PATH to find the configuration file. If $ZLOG_CONF_PATH is NULL also, all logs will be output to stdout with an internal format. Only the first call to zlog_init() per process is effective, subsequent calls will fail and do nothing.

zlog_reload() is designed to reload the configuration file. From the confpath it re-calculates the category-rule relationships, rebuilds thread buffers, and resets user-defined output function rules. It can be called at runtime when the configuration file is changed or you wish to use another configuration file. It can be called any number of times. If confpath is NULL, it reloads the last configuration file that zlog_init() or zlog_reload() specified. If zlog_reload() failed, the current configuration in memory will remain unchanged. So zlog_reload() is atomic.

zlog_fini() releases all zlog API memory and closes opened files. It can be called any number of times.

RETURN VALUE

On success , zlog_init() and zlog_reload() return zero. On error, zlog_init() and zlog_reload() return -1, and a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.2 category operation

SYNOPSIS

```
typedef struct zlog_category_s zlog_category_t;
zlog_category_t *zlog_get_category(const char *cname);
```

DESCRIPTION zlog_get_category() gets a category from zlog's category_table for a future log action. If the category cname does not exist it will be created. Then zlog goes through all rules as determined by the configuration. It returns a pointer to matched rules corresponding to cname.

This is how category string in rules matches cname:

1. * matches all cname.
2. category string which ends with underscore "_" matches super-category and sub-categories. For example, "aa_" matches cname like "aa", "aa_", "aa_bb", "aa_bb_cc".
3. category string which does not end with underscore "_"matches cname accurately. For example, "aa_bb" matches only a cname of "aa_bb".
4. ! matches cname that has no rule matched.

The rules for each category will be automatically re-calculated when zlog_reload() is called. No need to worry about category's memory release, zlog_fini() will clean up at the end.

RETURN VALUE

On success, return the address of zlog_category_t. On error, return NULL, and a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.3 log functions and macros

SYNOPSIS

```
void zlog(zlog_category_t * category,
            const char *file, size_t filelen,
            const char *func, size_t funclen,
            long line, int level,
            const char *format, ...);
void vzlog(zlog_category_t * category,
            const char *file, size_t filelen,
            const char *func, size_t funclen,
            long line, int level,
            const char *format, va_list args);
void hzlog(zlog_category_t * category,
            const char *file, size_t filelen,
            const char *func, size_t funclen,
            long line, int level,
            const void *buf, size_t buflen);
```

DESCRIPTION These 3 functions are the real log functions producing user messages, which correspond to %m is configuration file entries. category comes from zlog_get_category() described above.

zlog() and vzlog() produce output according to a format like printf(3) and vprintf(3).

vzlog() is equivalent to zlog(), except that it is called with a va_list instead of a variable number of arguments. vzlog() invokes the va_copy macro, the value of args remain unchanged after the call. See stdarg(3).

hzlog() is a little different, it produce output like this, the hexadecimal representation of buf and output len is buf_len

```
hex_buf_len=[5365]
                0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F      012345
0000000001      23 21 20 2f 62 69 6e 2f 62 61 73 68 0a 0a 23 20   #! /bin/
0000000002      74 65 73 74 5f 68 65 78 20 2d 20 74 65 6d 70 6f   test_hex
0000000003      72 61 72 79 20 77 72 61 70 70 65 72 20 73 63 72   rary wra
```

The parameter file and line are usually filled by the __FILE__ and __LINE__ macros. These indicate where the log event happened. The parameter func is filled with __func__ or __FUNCTION__, if the compiler supports it, otherwise it will be filled with "<unknown>".

level is an int in the current level list, which defaults to:

```
typedef enum {
    ZLOG_LEVEL_DEBUG = 20,
    ZLOG_LEVEL_INFO = 40,
    ZLOG_LEVEL_NOTICE = 60,
    ZLOG_LEVEL_WARN = 80,
    ZLOG_LEVEL_ERROR = 100,
    ZLOG_LEVEL_FATAL = 120
} zlog_level;
```

Each fuction has its macros for easy use. For example,

```
#define zlog_fatal(cat, format, args...) \
zlog(cat, __FILE__, sizeof(__FILE__)-1, \
__func__, sizeof(__func__)-1, __LINE__, \
ZLOG_LEVEL_FATAL, format, ##args)
```

The full list of macros is:

```
/* zlog macros */
/* zlog macros */
zlog_fatal(cat, format, ...)
zlog_error(cat, format, ...)
zlog_warn(cat, format, ...)
zlog_notice(cat, format, ...)
zlog_info(cat, format, ...)
zlog_debug(cat, format, ...)

/* vzlog macros */
vzlog_fatal(cat, format, args)
vzlog_error(cat, format, args)
vzlog_warn(cat, format, args)
vzlog_notice(cat, format, args)
vzlog_info(cat, format, args)
vzlog_debug(cat, format, args)
```

```
/* hzlog macros */
hzlog_fatal(cat, buf, buf_len)
hzlog_error(cat, buf, buf_len)
hzlog_warn(cat, buf, buf_len)
hzlog_notice(cat, buf, buf_len)
hzlog_info(cat, buf, buf_len)
hzlog_debug(cat, buf, buf_len)
```

RETURN VALUE

These functions return nothing. But if anything wrong happens, a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.4 MDC operation

SYNOPSIS

```
int zlog_put_mdc(const char *key, const char *value);
char *zlog_get_mdc(const char *key);
void zlog_remove_mdc(const char *key);
void zlog_clean_mdc(void);
```

DESCRIPTION MDC (Mapped Diagnostic Context) is a thread key-value map, and has nothing to do with category.

key and value are all strings, which should be no longer than MAXLEN_PATH(1024). If the input is longer than MAXLEN_PATH(1024), the input will be truncated.

One thing you should remember is that the map bonds to a thread, thus if you set a key-value pair in one thread, it will not affect other threads.

RETURN VALUE

zlog_put_mdc() returns 0 for success, -1 for fail. zlog_get_mdc() returns a pointer to value for success, NULL for fail or key not exist. If anything wrong happens, a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.5 dzlog API

SYNOPSIS

```
int dzlog_init(const char *confpath, const char *cname);
int dzlog_set_category(const char *cname);
void dzlog(const char *file, size_t filelen,
           const char *func, size_t funclen,
           long line, int level,
           const char *format, ...);
void vdzlog(const char *file, size_t filelen,
            const char *func, size_t funclen,
            long line, int level,
            const char *format, va_list args);
void hdzlog(const char *file, size_t filelen,
            const char *func, size_t funclen,
            long line, int level,
            const void *buf, size_t buflen);
```

DESCRIPTION dzlog consists of simple functions that omit *zlog_category_t*. It uses a default category internally and puts the category under lock protection. It is thread safe also. Omitting the category means that users need not create, save, or transfer *zlog_category_t* variables. Still, you can get and use other category values at the same time through the normal API for flexibility.

dzlog_init() is just as zlog_init(), but needs a cname for the internal default category. zlog_reload() and zlog_fini() can be used as before, to refresh conf_file, or release all.

dzlog_set_category() is designed for changing the default category. The last default category is replaced by the new one. Don't worry about releasing memory since all category allocations will be cleaned up at zlog_fini().

Macros are defined in zlog.h. They are the general way in simple logging.

```
dzlog_fatal(format, ...)
dzlog_error(format, ...)
dzlog_warn(format, ...)
dzlog_notice(format, ...)
dzlog_info(format, ...)
```

```
dezlog_debug(format, ...)

vdzlog_fatal(format, args)
vdzlog_error(format, args)
vdzlog_warn(format, args)
vdzlog_notice(format, args)
vdzlog_info(format, args)
vdzlog_debug(format, args)

hdzlog_fatal(buf, buf_len)
hdzlog_error(buf, buf_len)
hdzlog_warn(buf, buf_len)
hdzlog_noticebuf, buf_len)
hdzlog_info(buf, buf_len)
hdzlog_debug(buf, buf_len)
```

RETURN VALUE

On success, dzlog_init() and dzlog_set_category() return zero. On error, dzlog_init() and dzlog_set_category() return -1, and a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.6 User-defined Output

SYNOPSIS

```
 typedef struct zlog_msg_s {
         char *buf;
         size_t len;
         char *path;
} zlog_msg_t;
typedef int (*zlog_record_fn)(zlog_msg_t *msg);
int zlog_set_record(const char *rname, zlog_record_fn record);
```

DESCRIPTION zlog allows a user to define an output function. The output function bonds to a special kind of rule in configuration file. A typical rule is:

```
*.*     $name, "record path %c %d"; simple
```

zlog_set_record() does the bonding operation. Rules with the $rname will be output through user-defined function record. The callback function has the type of zlog_record_fn.

The members of struct zlog_msg_t are decribed below:

path comes from the second parameter of rule after $name, which is generated dynamically like the file path.

buf and len are zlog formatted log message and its length.

All settings of zlog_set_record() are kept available after zlog_reload().

RETURN VALUE

On success, zlog_set_record() returns zero. On error, it returns -1, and a detailed error log will be recorded to the log file indicated by ZLOG_PROFILE_ERROR.

6.7 debug and profile

SYNOPSIS `void zlog_profile(void);`

DESCRIPTION environment variable ZLOG_PROFILE_ERROR indicates zlog's error log path.

environment variable ZLOG_PROFILE_DEBUG indicates zlog's debug log path.

zlog_profile() prints all information in memory to zlog's error log file at runtime. You can compare it to the configuration file to find possible errors.

Chapter 7

Advanced Usage

7.1 MDC

What is MDC? In log4j it is short for Mapped Diagnostic Context. That sounds like a complicated terminology. MDC is just a key-value map. Once you set it by function, the zlog library will print it to file every time a log event happens, or become part of log file path. Let's see an example in $(top_builddir)/test/test_mdc.c.

```
$ cat test_mdc.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include "zlog.h"
int main(int argc, char** argv)
{
    int rc;
    zlog_category_t *zc;
    rc = zlog_init("test_mdc.conf");
    if (rc) {
        printf("init failed\n");
        return -1;
    }
    zc = zlog_get_category("my_cat");
    if (!zc) {
            printf("get cat fail\n");
            zlog_fini();
            return -2;
        }
        zlog_info(zc, "1.hello, zlog");
        zlog_put_mdc("myname", "Zhang");
        zlog_info(zc, "2.hello, zlog");
        zlog_put_mdc("myname", "Li");
        zlog_info(zc, "3.hello, zlog");
        zlog_fini();
        return 0;
    }
```

The configure file is

```
$ cat test_mdc.conf
[formats]
mdc_format=     "%d.%ms %-6V (%c:%F:%L) [%M(myname)] - %m%n"
[rules]
*.*             >stdout; mdc_format
```

And the output is

```
$ ./test_mdc
2012-03-12 09:26:37.740 INFO   (my_cat:test_mdc.c:47) [] - 1.hello, zlog
2012-03-12 09:26:37.740 INFO   (my_cat:test_mdc.c:51) [Zhang] - 2.hello, zlog
2012-03-12 09:26:37.740 INFO   (my_cat:test_mdc.c:55) [Li] - 3.hello, zlog
```

You can see zlog_put_mdc() function sets the map with key "myname" and value "Zhang", and in configuration file *%M(myname)* indicates where the value shows in each log. The second time, value of key "myname" is overwritten to "Li", and the log changes also.

When should MDC be used? That mainly depends on when a user need to separate same log action with different scenarios. For example, in .c

```
zlog_put_mdc("customer_name", get_customer_name_from_db() );
zlog_info("get in");
zlog_info("pick product");
zlog_info("pay");
zlog_info("get out");
```

in .conf

```
&format  "%M(customer_name) %m%n"
```

When the program processes two customers at the same time, the output could be:

```
Zhang get in
Li get in
Zhang pick product
Zhang pay
Li pick product
Li pay
Zhang get out
Li get out
```

Now you can distinguish one customer from another, by using grep afterwards

```
$ grep Zhang aa.log > Zhang.log
$ grep Li aa.log >Li.log
```

Or, another way is to seperate them to different log file when log action is taken. In .conf

```
*.* "mdc_%M(customer_name).log";
```

It will produce 3 logs

```
mdc_.log mdc_Zhang.log mdc_Li.log
```

That's a quick way, if you know what you are doing.

In MDC, the map belongs to a thread and each thread has it's own map. In one thread zlog_mdc_put() will not affect other thread's map. Still, if you want to distinguish one thread from another, using the %t conversion character is enough.

7.2 Profile zlog Itself

Until this point we have assumed that the zlog library never fails. It helps the application to write log entries and to debug the application. But if zlog itself has some problem, how can we find out? Other programs debug through the log library so how can a log library debug itself? The answer is the same, zlog library has its own log. This profile log is usually closed, and can be opened by setting environment variables.

```
$ export ZLOG_PROFILE_DEBUG=/tmp/zlog.debug.log
$ export ZLOG_PROFILE_ERROR=/tmp/zlog.error.log
```

profile log has just 2 levels, debug and error. After setting them, run test_hello program in 3.3, and the debug log will be

```
$ more zlog.debug.log
03-13 09:46:56 DEBUG (7503:zlog.c:115) ------zlog_init start, compile time[Mar 1
03-13 09:46:56 DEBUG (7503:spec.c:825) spec:[0x7fdf96b7c010][%d(%F %T)][%F %T 29
03-13 09:46:56 DEBUG (7503:spec.c:825) spec:[0x7fdf96b52010][ ][ 0][]
......
03-13 09:52:40 DEBUG (8139:zlog.c:291) ------zlog_fini end------
```

zlog.error.log is not created, as no error occurs.

As you can see, debug log shows how zlog is inited and finished, but no debug log is written when zlog_info() is executed. That's for efficency.

If there is anything wrong with zlog library, all will show in zlog.error.log. For example, using a wrong printf syntax in zlog()

```
zlog_info(zc, "%l", 1);
```

Then run the program, the zlog.error.log should be

```
$ cat zlog.error.log
03-13 10:04:58 ERROR (10102:buf.c:189) vsnprintf fail, errno[0]
03-13 10:04:58 ERROR (10102:buf.c:191) nwrite[-1], size_left[1024], format[%l]
03-13 10:04:58 ERROR (10102:spec.c:329) zlog_buf_vprintf maybe fail or overflow
03-13 10:04:58 ERROR (10102:spec.c:467) a_spec->gen_buf fail
03-13 10:04:58 ERROR (10102:format.c:160) zlog_spec_gen_msg fail
03-13 10:04:58 ERROR (10102:rule.c:265) zlog_format_gen_msg fail
03-13 10:04:58 ERROR (10102:category.c:164) hzb_log_rule_output fail
03-13 10:04:58 ERROR (10102:zlog.c:632) zlog_output fail, srcfile[test_hello.c],
```

Now, you could find the reason why the expected log doesn't generate, and fix the wrong printf syntax.

Runtime profiling causes a loss of efficency. Normally, I keep ZLOG_PROFILE_ERROR on and ZLOG_PROFILE_DEBUG off in my environment.

There is another way to profile the zlog library. zlog_init() reads the configuration file into memory. Throughout all log actions, the configure structure remains unchanged. There is possibility that this memory is damaged by other functions in an application, or the memory doesn't equal what the configuration file describes. So there is a function to show this memory at runtime and print it to ZLOG_PROFILE_ERROR.

see $(top_builddir)/test/test_profile.c

```
$ cat test_profile.c
#include <stdio.h>
#include "zlog.h"

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

    int rc;
    rc = dzlog_init("test_profile.conf", "my_cat");
    if (rc) {
        printf("init failed\n");
        return -1;
    }
    dzlog_info("hello, zlog");
    zlog_profile();
    zlog_fini();
    return 0;

}
```

zlog_profile() is the function. The configuration file is simple

```
$ cat test_profile.conf
[formats]
simple = "%m%n"
[rules]
my_cat.*                >stdout; simple
```

Then zlog.error.log is

```
$ cat /tmp/zlog.error.log
06-01 11:21:26 WARN  (7063:zlog.c:783) ------zlog_profile start------
06-01 11:21:26 WARN  (7063:zlog.c:784) init_flag:[1]
06-01 11:21:26 WARN  (7063:conf.c:75) -conf[0x2333010]-
06-01 11:21:26 WARN  (7063:conf.c:76) --global--
06-01 11:21:26 WARN  (7063:conf.c:77) ---file[test_profile.conf],mtime[2012-06-0
06-01 11:21:26 WARN  (7063:conf.c:78) ---strict init[1]---
06-01 11:21:26 WARN  (7063:conf.c:79) ---buffer min[1024]---
06-01 11:21:26 WARN  (7063:conf.c:80) ---buffer max[2097152]---
06-01 11:21:26 WARN  (7063:conf.c:82) ---default_format---
06-01 11:21:26 WARN  (7063:format.c:48) ---format[0x235ef60][default = %d(%F %T)
06-01 11:21:26 WARN  (7063:conf.c:85) ---file perms[0600]---
06-01 11:21:26 WARN  (7063:conf.c:87) ---rotate lock file[/tmp/zlog.lock]---
06-01 11:21:26 WARN  (7063:rotater.c:48) --rotater[0x233b7d0][0x233b7d0,/tmp/zlo
06-01 11:21:26 WARN  (7063:level_list.c:37) --level_list[0x2335490]--
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x23355c0][0,*,*,1,6]---
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x23375e0][20,DEBUG,debug,5,7]--
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x2339600][40,INFO,info,4,6]---
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x233b830][60,NOTICE,notice,6,5]
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x233d850][80,WARN,warn,4,4]---
06-01 11:21:26 WARN  (7063:level.c:37) ---level[0x233fc80][100,ERROR,error,5,3]-
```

7.3 User-defined Level

Here are all the steps to define your own levels.

1. Define levels in the configuration file.

   ```
   $ cat $(top_builddir)/test/test_level.conf
   [global]
   default format  =              "%V %v %m%n"
   [levels]
   TRACE = 30, LOG_DEBUG
   [rules]
   my_cat.TRACE            >stdout;
   ```

   The internal default levels are (no need to write them in the conf file):

```
DEBUG = 20, LOG_DEBUG
INFO = 40, LOG_INFO
NOTICE = 60, LOG_NOTICE
WARN = 80, LOG_WARNING
ERROR = 100, LOG_ERR
FATAL = 120, LOG_ALERT
UNKNOWN = 254, LOG_ERR
```

In zlog, an integer(30) and a level string(TRACE) represent a level. Note that this integer must be in [1,253], any other number is illegal. Higher numbers are more important. TRACE is more important than DEBUG(30>20), and less important than INFO(30<40). After the definition, TRACE can be used in rule of configure file. This sentence

```
my_cat.TRACE >stdout;
```

means that level >= TRACE, which is TRACE, INFO, NOTICE, WARN, ERROR, FATAL will be written to standard output.

The conversion charactor %V of format string generates the uppercase value of the level string and %v generates the lowercase value of the level string.

In the level definition LOG_DEBUG means when using >syslog in a rule, all TRACE log will output as syslog' s LOG_DEBUG level.

2. Using the new log level in source file, the direct way is like this

```
zlog(cat, __FILE__, sizeof(__FILE__)-1, \
__func__, sizeof(__func__)-1,__LINE__, \
30, "test %d", 1);
```

For easy use, create a .h file

```
$ cat $(top_builddir)/test/test_level.h
#ifndef __test_level_h
#define __test_level_h

#include "zlog.h"

enum {
    ZLOG_LEVEL_TRACE = 30,
    /* must equals conf file setting */
};
#define zlog_trace(cat, format, ...) \
        zlog(cat, __FILE__, sizeof(__FILE__)-1, \
        __func__, sizeof(__func__)-1, __LINE__, \
        ZLOG_LEVEL_TRACE, format, ## __VA_ARGS__)
#endif
```

3. Then zlog_trace can be used int .c file

```
$ cat $(top_builddir)/test/test_level.c
#include <stdio.h>
#include "test_level.h"
int main(int argc, char** argv)
{
    int rc;
    zlog_category_t *zc;

    rc = zlog_init("test_level.conf");
    if (rc) {
        printf("init failed\n");
        return -1;
    }
    zc = zlog_get_category("my_cat");
    if (!zc) {
        printf("get cat fail\n");
        zlog_fini();
        return -2;
    }
    zlog_trace(zc, "hello, zlog - trace");
    zlog_debug(zc, "hello, zlog - debug");
    zlog_info(zc, "hello, zlog - info");
    zlog_fini();
    return 0;
}
```

4. Now we can see the output

```
$ ./test_level
TRACE trace hello, zlog - trace
INFO info hello, zlog - info
```

That's just what we expect. The configuration file only allows >=TRACE ouput to screen. And %V and %v work as well.

7.4 User-defined Output

The goal of user-defined output is that zlog gives up some rights. zlog is only responsible for generating path and message dynamically as per the user's configuration, but leaves the log output, rotate and cleanup actions for the user to specify. You can do what ever you want by setting a function to special rules. Here are the steps to define your own output function.

1. Define output in rules of configure file.

```
$ cat test_record.conf
[formats]
simple = "%m%n"
[rules]
my_cat.*      $myoutput, " mypath %c %d";simple
```

2. Set an output function for myoutput, then use it

```
#include <stdio.h>
#include "zlog.h"
int output(zlog_msg_t *msg)
{

    printf("[mystd]:[%s][%s][%ld]\n", msg->path, msg->buf, (long)msg->len);
    return 0;
}

int main(int argc, char** argv)
{
    int rc;
    zlog_category_t *zc;
```

```
        rc = zlog_init("test_record.conf");
        if (rc) {
        printf("init failed\n");
        return -1;
    }
    zlog_set_record("myoutput", output);
    zc = zlog_get_category("my_cat");
    if (!zc) {
        printf("get cat fail\n");
        zlog_fini();
        return -2;
    }
    zlog_info(zc, "hello, zlog");
    zlog_fini();
    return 0;
}
```

3. Now we can see how the user-defined output() works

   ```
   $ ./test_record
   [mystd]:[ mypath my_cat 2012-07-19 11:01:12][hello, zlog
   ][12]
   ```

   As you can see, msglen is 12, and msg is formatted by zlog to contain a newline character.

4. There are many other things you can do with user-defined output functions. As one user(flw@newsmth.net) provided:

   (a) Log name is foo.log

   (b) If foo.log is larger than 100M, then generate a new logfile which contains all the contents of foo.log. And zlog truncates foo.log to 0 and re-appends to it when the next log happens.

   (c) When the time is over 5 minutes after last logging, even if foo.log is not larger than 100M, zlog still jumps to a new file.

   (d) The new file name should be defined by your own needs. For example add device number as prefix and time string as postfix.

(e) You might compress the new log file to save disk space and network bandwidth.

I wish him good luck trying to write such a function for multi-process or multi-thread cases!

Chapter 8

Epilog

Here's to alcohol, the cause of – and solution to – all life's problems.

Homer Simpson