zlog User's Guide

by Hardy Simpson[1][2]

March 27, 2012

[1]This Guide is for zlog v0.9.

[2]If you have comments or error corrections, write email to HardySimpson1984@gmail.com

Contents

Chapter 1

What is zlog?

zlog is a high efficent, thread safe, flexsible, clear model, pure c logging library

Actually, in the c world there is NO good logging library for application like logback in java or log4cxx in c++. printf can work, but can not be easily redirected or reformat, syslog is slow and is designed for system use.

So I write zlog.

zlog has feartures below:

- syslog style configure file, easy for understand and use
- log format Customization, like pattern layout in log4j
- range-category model, which is more flexible and more clear than hierarchy model of log4j
- multiple output, include static file path, dynamic file path, stdout, stderr, syslog
- runtime refreash configure to change output flow or output format, just need to call one function – zlog_update()
- high efficieny, on my laptop, record 720'000 log per second, about 200 times faster than syslog(3) with rsyslogd
- user can define his own log level without change and rebuild library6.3
- safely rotate log file by size when multiple process or multiple threads write to one same log file

- accurate to microseconds
- dzlog, a default category log API for easy use
- MDC, a log4j style key-value map, expand user defined field in dynamic log file path or log format, is also useful in multi-thread programming
- self debugable, can output zlog's self debug&error log at runtime
- Not depend on any other 3rd party library, just base on POSIX system.

Links:

Download: https://github.com/HardySimpson/zlog/zipball/master

UsersGuide: https://github.com/HardySimpson/zlog/raw/master/download/ZlogUsersGuide-EN-0.9rc1.pdf

Homepage: https://github.com/HardySimpson/zlog

Author's Blog(in Chinese): http://my.oschina.net/HardySimpson/blog

Author's Email: HardySimpson1984@gmail.com

Chapter 2

What is not zlog

The goal of zlog is becoming a simple, fast log library for application. It will not support too much output like send log to other machine through net or save it to database. It will not parse content of log and filter them.

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

If you want all these features, I recommend rsyslog, a excellent syslog daemon implementation. It is a independent process receives logs from other process or machines, and parse and store logs. Still, it has a distance from user application.

So now I think about how to connect zlog and rsyslog, the tranditional syslog(3) is one way, but maybe there is another way more efficient. Also I think about let user hook his own output function to zlog. If you have a good idea, tell me!

Chapter 3

Hello World

3.1 Build and Installation zlog

If checking out from Git(https://github.com/HardySimpson/zlog) and you want to try auto tools in your environment to re-generate configure shell.

```
$ sh autogen.sh
```

Then (Normally, begin here)

```
$ ./configure --prefix=[where u wanna install it] \
      --enable-test
$ make
$ make install
```

If –enable-test is used, then all the test program under test directory will be built. These are also good example codes which show how to use zlog. If zlog is built in other path other than source path, you need to copy all configure file to build path by hand, to run test program successfully, for example:

```
$ cp $(top_srcdir)/test/*.conf $(top_builddir)/test/
$ cd $(top_builddir)/test/
$ ./test_hello
hello, zlog
```

3.2 In User's application Call and Link zlog

To use zlog, only add one line below in source c file or cpp file.

```
#include ‘‘zlog.h’’
```

If your system has pkgconfig then you can just add this to your makefile

```
CFLAGS += $(shell pkg-config --cflags zlog)
LDFLAGS += $(shell pkg-config --libs zlog)
```

If has not, the hand-made link command is

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

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 (!my_cat) {
```

```
            printf("get cat fail\n");
            zlog_fini();
            return -2;
        }
        ZLOG_INFO(c, "hello, zlog");
        zlog_fini();
        return 0;
    }
```

2. write a configure file in the same path as test_hello.c:

```
$ vi test_hello.conf
&simple                 "%m%n"

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 equals test_hello.conf, and the output of test_default equals test_hello. The difference is, test_default use dzlog API, which has a default *zlog_ cateogy_ t* inside and easy for use. See 5.6 for more details.

Chapter 4

Configure File

Most actions of zlog library are dependent upon configure file: where to output the log, how log is rotate, the output format, etc... Configure File is a special language tells how library works. I design it as clear as I can.

4.1 Global Setting

Global setting begins with @. All global setting could be not written, for use default value. The full sytanx is:

```
@[key][n space or tab][value]
```

If ignore_error_format_rule is true, zlog_init() will omit error syntax of formats and rules. Else if ignore_error_format_rule is false, zlog_init() will check sytnax of all formats and rules strictly, and any error will cause zlog_init() failed and return -1. Default, ignore_error_format_rule is false.

```
@ignore_error_format_rule    false
```

zlog allocates one log buffer in each thread. buf_size_min indicates size of buffer malloced at init time. While loging, if log content size > buf_size, buffer will expand automaticly, till buf_size_max, and log content is truncated. buf_size_max setting to 0 means buf_size is unlimited, everytime buf_size = 2*buf_size, till process use up all it's memory. Size can append with unit KB, MB or GB suffix, so 1024 equals 1KB. Default, buf_size_min is 1K and buf_size_max is 2MB.

```
@buf_size_min           1024
@buf_size_max           2MB
```

rotate_lock_file is a lock file for rotate a log safely between multi-process. zlog will create the file at zlog_init(). Make sure your program has permission to create and read-write the file. If programs run by different users who need to write and rotater a same log file, make sure that each program has permission to create and read-write the same lock file. Default, rotate_lock_file is /tmp/zlog.lock

```
@rotate_lock_file       /tmp/zlog.lock
```

default_format is used by rules without format. That cause each rule without format specified, would yield output like this:

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

You can set it to change the default behavior.The inner default format:

```
@default_format "%d(%F %T) %V [%p:%F:%L] %m%n"
```

User can defined his own levels, and suggest to be used with user-defined macros in source file.

The inner default levels are

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

The syntax is

```
@level[n tabs or spaces][level string] = [level int], [syslog level, optional
```

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

see 6.3 for more details.

4.2 Category, Rule and Format

This is part of C source file:

```
zlog_init("test_hello.conf");
/* read conf file to memory */

c = zlog_get_category("my_cat");
/* match "my_cat" to c */

ZLOG_INFO(c, "hello, zlog");
/* logging action "hello, zlog" which bonds to c */
```

And this is part of conf file:

```
&simple                  "%m%n"
# format, begein with a &
my_cat.DEBUG             >stdout; simple
# rule, all my_cat's log and level bigger than debug
# output to standard output, use simple format decribed above.
```

And this is the output:

```
hello, zlog
```

Category is designed for different input. In source code name of category variable is a charactor string. In program, get different category for log will distinguish them from each other.

Format describes log pattern, like with or without time stamp, source file, source line.

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

So when excute this sentence in source file

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

zlog library will find the name of c is "my_cat", which has one rule. That is

```
my_cat.DEBUG             >stdout; simple
```

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

```
&simple                    "%m%n"
```

At last show in screen

```
hello, zlog
```

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

Category variable and rule are loose coupled. If there are 2 lines of same rules above, thus *ZLOG_INFO(c, "hello, zlog")* will cause to lines of output "*hello, zlog*". If mistaking the rule, write it as "my_ca.debug >stdout; simple", nothing will be output to screen. Loose coupling are both fallible and flexible. Still, zlog has a feature to make up for this deficiency, see 4.3.2.

Rule and format are tightly coupled. If a rule using a format that is not mentioned before, zlog_init() will be failed, unless "@ignore_error_format_rule false" is set at the beginning of configure file. One format can be used in mutilple rules.

The syntax of format is:

```
&[name][n tab or space]‘‘[conversion pattern]’’
```

The syntax of rule is:

```
[selector][n tab or space][action]
[selector] = [category string].[level]
[action] = [output], [file limitation,optional]; [format name, optional]
```

4.3 Select Input

Whether or not a log will be output by zlog, depend on category variable and level in c source file match category string and level in rule or not.

```
[selector] = [category string].[level]
```

4.3.1 Level Matching

There are six default level in zlog, "DEBUG", "INFO", "NOTICE", "WARN", "ERROR" and "FATAL". As all other log library, aa.DEBUG means all log of level that is greater than or equals to DEBUG will be output. Still, there are more expressions. Levels in configure file are case insensitive, both capital or lowercase is accepted.

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

User can define his own level, see6.3.

4.3.2 Category Matching

Finally, We arrive at here. The matching between source category variable and rule category string from configure file is quite different from other logging library like log4j. I call it range-category model.

In log4j, there are father logger and child logger. For Example, if the configure file is like this:

```
log4j.logger.aa=DEBUG, A1
log4j.logger.aa.bb=
log4j.logger.aa.cc=INFO
```

aa is a father logger. aa.bb and aa.cc is the child of aa, aa.bb inherits aa's level and appender, which are DEBUG and A1. aa.cc inherits and overwrite aa's level. So aa.cc has the level of INFO and appender of A1. Then, what will happen if I want all aa's log output to a special file (and keep output of aa.bb and aa.cc not changed)? It is complicate to give expression here, maybe need set threshold, bonds it to appender...

In zlog, all rules are independent, NO heritage. Range-category relationship is express as a underline in category string. For example:

```
#rule 1
aa_bb.DEBUG         "/var/log/aa_bb.log"

#rule 2
```

```
aa_cc.INFO          "/var/log/aa_cc.log"

#rule 3
aa_.ERROR           "/var/log/aa_error.log"

#rule 4
aa.*                "/var/log/aa.log"
```

There is no heirship here, just 4 sperate rules.

If the name of category in source is "*aa_ bb*". The c source file is like this.

```
category_t ab;
ab = zlog_get_category("aa_bb");
ZLOG_DEBUG(ab, "ab's debug");
ZLOG_ERROR(ab, "ab's error");
```

Then category string in rule 1 and rule 3, which are "aa_bb" and "aa_", match the category variable in source file, whose name is "aa_bb". These two rules determine that aa_bb's >=DEBUG log action in source will be output to *aa_ bb.log,* and aa_bb's >=ERROR log action in source will be output to *aa_ error.log*. ERROR log will be written in both files. But rule 4 doesn't match the category. It's category string is "aa", which accurately matches the category variable whose name is"aa".

That is the range-category model. Rule and rule are sperate. Each source category variable has its rules, one rule may belong to different soure category variable. Rule-super-category-string(with '_') matches all source-sub-category-variable, so super-category-string range contains sub-cateogry-string range. At a result, user could select any range in configure file, matches big or small of categories for output, but not affect other rules' action.

In fact, when *zlog_ get_ category()* is called, there is NO guarantee that the category must contains rules. It may have many matching rules, or may have none. That depends on how configure file is written. After the configure file is changed and *zlog_ update()* is called, the relationship between categories and rules will be re-calculated. Each Category will get its new matching rules, in the way described above.

The compensation for category nomatching is the wastebin rule. Category name of wastebin rule is "!". This rule, whose category string is "!", matches category that doesn't has any rules. For example, when *zlog_ get_ category("xx")*

goes through all rules and find there is no rule matches *"xx"*, and there is a rule's category string is "!". Wastebin rule matches category *"xx"* and output it's log to a special file. Wastebin rule is unique, only the last "!"rule works. User can find his mistake at runtime. The wastebin rule maybe write like this

```
!.*                    "/var/log/zlog.nomatch.log"
```

Let's conclude with a table here.

summarize	category string from configure file	matched category	no matched category
* 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 end 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)

4.4 Output Action

Now, zlog supports 4 ways of output, the syntax is

```
[action] = [output], [file limitation,optional]; [format name, optional]
```

output action	output	file limitation
to standard out	>stdout	no meaning
to standard error	>stderr	no meaning
to syslog	>syslog	syslog facilitiy, can be: LOG_USER(default), LOG_LOCAL[0-7] This is not optional.
to file	"[file path]"	file limitation, can be: 1000, 1k, 2M, 1G... 3m*2, 4k*3...

[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 at last. See 4.5 for more details.

[file limitation] controls log file size and count. zlog rotate log file when the file is too large by this value. Let the action is

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

After a out.log is filled by programs 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 time 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.

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

```
@default_format "%d(%F %T) %P [%p:%F:%L] %m%n"
```

That cause each rule without format specified, would yield output like this:

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

Each rule can use its own format. One format can be used in mutilple rules. . If before the rule write such a format:

```
&simple "%m%n''
my_cat.*    >stdout; simple
```

Then the output will be:

```
hello, zlog
```

The syntax of format is:

```
&[name][n tab or space]‘‘[conversion pattern]''
```

4.5 Conversion pattern

The conversion pattern is closely related to the conversion pattern of the printf function in C. 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) %-5P [%p:%F:%L] %m%n".
```

Then the statements

```
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.

4.5.1 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 parenthesis. 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 admits the same syntax as the strftime(3), but add %us and %ms for millisecond and microsecond. see 4.5.3for 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

%E	Used to output the environment variable. The key of environment variable should be enclose between parenthesis.Vaule is fixed at zlog_init()	%E(HOME) /home/harry
%F	Used to output the file name where the logging request was issued. The file name comes from __FILE__ macro. Some compiler take __FILE__ as the absolute path. Use $f to strip path and remain file name. Or complier has option to switch.	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
%H	Used to output the hostname of system, which is from gethostname(2)	zlog-dev
%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 6.1 for more detail.	%M(clientNumber) 12345

%n	Outputs unix newline character, I do not support windows line separator now.	\n
%p	Used to output the id of the process that generated the logging event, which comes from getpid().	2134
%V	Used to output the level of the logging event, capital.	INFO
%v	Used to output the level of the logging event, lowercase.	info
%t	Used to output the id of the thread that generated the logging event, which comes from pthread_self().	7636
%%	the sequence %% outputs a single percent sign.	%
%[other char]	parse as a wrong syntax, will cause zlog_init() fail	

4.5.2 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, it 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.
%.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.

4.5.3 Time Character

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

There are 2 speical time words support by zlog itself, which comes from gettimeofday(2)

word	effect	example
%ms	The millisecond, 3-digit integer string	013
%us	The microsecond, 6-digit integer string	002323

Other Character is supported by strftime(3) in library. The Character support 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. (Yecch — for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in 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 corre- sponding 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.	%

4.6 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 try to read conf files, and check their syntax, and output to screen whether it is correct. I suggest using this tools each time you create or change a configure 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
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[
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
```

It tells you that line 126 in you configure file, zlog.conf, is wrong, and the 1st line further tells you that [aaa] is not a right rule.

zlog-chk-conf can read mutiple configure files at the same time. For example

```
$ zlog-chk-conf zlog.conf ylog.conf
--[zlog.conf] syntax right
--[ylog.conf] syntax right
```

The other useful tool is

```
$ zlog-gen-conf -h
Useage: zlog-gen-conf [conf file]
If no filename is specified, use zlog.conf as default
-c  Chinese comment(UTF-8)
          if envrionment is GBK, use
          $ iconv -f UTF-8 -t GBK xx.conf > yy.conf
          $ mv yy.conf xx.conf
-e  Enligsh comment
-h, show help message
```

It will generate a configure file template for you, with comment if needed. The comment is a summary of this chapter.

Chapter 5

zlog API

All API of zlog are thread safe. To use them, just need to

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

5.1 init and finish

```
int zlog_init(char *conf_file);
int zlog_update(char *conf_file);
void zlog_fini(void);
```

zlog_ init() read configuration from file. If *conf_ file* is NULL, all log will be output to stdout with inner format. If *zlog_ init()* fail, it will record a error log in ZLOG_PROFILE_ERROR(see 6.2), and returns -1. Only the first time call *zlog_ init()* per process is effective, other times will fail.

zlog_ update() is designed to reload *conf_ file*. From the conf-file it re-calculate category-rule relationship, rebuild thread buffers. It can be call at runtime when configure file changed, and can be use unlimit times. If conf_file is NULL, reload the last configure file *zlog_ init()* or *zlog_ update()* specified. If *zlog_ update()* failed, return -1 and zlog will be unuseable.

zlog_ fini() release all memory zlog API applied, close opened files. It can be call unlimit times.

5.2 category operation

```
typedef struct zlog_category_t zlog_category_t;
```

```
zlog_category_t *zlog_get_category(char *category_name);
```

zlog_get_category() get a category from global table for future log, if none, create it. If success, return the address of zlog_category_t, else return NULL. *category_name* should correspond to configure setting, see 4.3. No need to worry about category's memory release, *zlog_fini()* will clean up at last.

5.3 log functions

```
void zlog(zlog_category_t * a_cat,
            char *file, long line,
            int level,
            char *format, ...);
void vzlog(zlog_category_t * a_cat,
            char *file, long line,
            int level,
            char *format, va_list args);
void hzlog(zlog_category_t * a_cat,
            char *file, long line,
            int level,
            void *buf, size_t buf_len);
```

These 3 functions are the real log function producing user message, which corresponds to %m is configure file. *a_cat* 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()*, respectively, 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      0123456789AB
0000000001  23 21 20 2f 62 69 6e 2f 62 61 73 68 0a 0a 23 20   #! /bin/bash..
```

```
0000000002   74 65 73 74 5f 68 65 78 20 2d 20 74 65 6d 70 6f   test_hex - tem
0000000003   72 61 72 79 20 77 72 61 70 70 65 72 20 73 63 72   rary wrapper s
```

The parameter *file* and *line* are usually filled with __FILE__ and __LINE__ macro. These indicate where log event happens.

level is a int, which usually is in

```
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;
```

5.4 log macros

For easy writing, there are some log macros

macros of zlog()

```
#define ZLOG_FATAL(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_FATAL, format, ##args)
#define ZLOG_ERROR(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_ERROR, format, ##args)
#define ZLOG_WARN(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_WARN, format, ##args)
#define ZLOG_NOTICE(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_NOTICE, format, ##args)
#define ZLOG_INFO(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_INFO, format, ##args)
#define ZLOG_DEBUG(cat, format, args...) \
zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_DEBUG, format, ##args)
```

macros of vzlog()

```
#define VZLOG_FATAL(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_FATAL, format, args)
#define VZLOG_ERROR(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_ERROR, format, args)
#define VZLOG_WARN(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_WARN, format, args)
#define VZLOG_NOTICE(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_NOTICE, format, args)
#define VZLOG_INFO(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_INFO, format, args)
#define VZLOG_DEBUG(cat, format, args) \
vzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_DEBUG, format, args)
```

macros of hzlog()

```
#define HZLOG_FATAL(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_FATAL, buf, buf_len)
#define HZLOG_ERROR(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_ERROR, buf, buf_len)
#define HZLOG_WARN(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_WARN, buf, buf_len)
#define HZLOG_NOTICE(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_NOTICE, buf, buf_len)
#define HZLOG_INFO(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_INFO, buf, buf_len)
#define HZLOG_DEBUG(cat, buf, buf_len) \
hzlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_DEBUG, buf, buf_len)
```

5.5 MDC operation

MDC(Mapped Diagnostic Context) is a thread key-value map, so it has nothing to do with category. the fuctions are

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

key and value are all strings, which are no longer than MAXLEN_PATH(typically 1024). One thing should remeber is that the map bonds to a thread, thus in one thread if you set a key-value pair will not affect other threads.

zlog_put_mdc() returns 0 for success, -1 for fail. zlog_get_mdc returns poiter of value for success, NULL for fail or key not exist.

5.6 dzlog API

dzlog consists of some simlple fuctions that omit *zlog_category_t*. It use a default category inside and put the category under the protect of lock. It is thread safe also. Omit category means that users need not to create, save, tranfer *zlog_category_t* variables. Still, user can get and use other category through normal API for flexibility.

```
int dzlog_init(char *conf_file, char *default_category_name);
int dzlog_set_category(char *default_category_name);
```

dzlog_init() is just as *zlog_init()*, but need a *default_category_name* for inner default_category. *zlog_update()* and *zlog_fini()* can be used as before, to refresh conf_file, or release all.

dzlog_set_category() is designed for change default category. The last default category is instead by new one, and still don't worry about memory releasing, all category will be clean up at *zlog_fini()*.

```
void dzlog(char *file, long line, int level, char *format, ...);
void vdzlog(char *file, long line, int level, char *format, va_list args);
void hdzlog(char *file, long line, int level, void *buf, size_t buf_len);
```

These 3 are dzlog functions, which bind to the default category.

```
DZLOG_FATAL(format, args...)
DZLOG_ERROR(format, args...)
DZLOG_WARN(format, args...)
DZLOG_NOTICE(format, args...)
DZLOG_INFO(format, args...)
DZLOG_DEBUG(format, args...)
```

```
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_NOTICE(buf, buf_len)
HDZLOG_INFO(buf, buf_len)
HDZLOG_DEBUG(buf, buf_len)
```

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

Chapter 6

Advance Using

6.1 MDC

What is MDC? In log4j it is short for Mapped Diagnostic Context. That sounds like a complicate terminology. MDC is just a key-value map. Once you set it by function, library will print it to file every time a log event happens, or become part of log file path. Let's see a 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
&mdc_format    "%d(%F %X.%ms) %-6V (%c:%F:%L) [%M(myname)] - %m%n"
*.*             >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
```

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

When should MDC be used? That mainly depends on when user need to seprate same log action with different scenes. 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 program process two customer at the same time, the output maybe:

```
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 two one customer from another, by use grep afterwards

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

Or, there is another way, sperate them to different log file when log action is taken, in .conf

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

It will produce 3 log

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

That's a quick way, if user kown what he is doing.

About MDC, another thing is that map belongs to thread, each thread has it's own map. In one thread *zlog_ mdc_ put()* will not affect other thread's map. Still, if you just want to distinguish on thread from another, use %t in conversion charactor is enough.

6.2 Profile zlog Itself

OK, till now, I suspect that zlog library never fails, it help user's application to write log and debug user's application. But if zlog itself has some problem, how to find it out? Other program debug through log library, and how can a log library debug itself? The answer is the same, zlog library has its own log. This profile log is usually shut down, and can be open by set environment variables.

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

profile log just has 2 levels, debug and error. After Settion, run test_hello program in 3.3, then the debug log is

```
$ more zlog.debug.log
03-13 09:46:56 DEBUG (7503:zlog.c:115) ------zlog_init start, compile time[Ma
03-13 09:46:56 DEBUG (7503:spec.c:825) spec:[0x7fdf96b7c010][%d(%F %T)][%F %T
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 writen 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 overfl
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.
```

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

Runtime profile will make efficency lost. Normally, I keep ZLOG_PROFILE_ERROR on and ZLOG_PROFILE_DEBUG off at my environment.

There is still another way to profile zlog libraray. As we all know, *zlog_ init()* read configure file to memory . Through all log actions, the configure structure memory keeps unchanged. There is possibility that this memory is damaged by other fuctions in user's application, or the memory doesn't equal what configure file describes. So I design a function to show this memory at runtime, 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 configure file is simple

```
$ cat test_profile.conf
@ignore_error_format_rule false
@buf_size_min 1024
@buf_size_max 0
@rotate_lock_file /tmp/zlog.lock
```

```
&simple "%m%n"

my_cat.* >stdout; simple
```

Then zlog.error.log is

```
$ cat /tmp/zlog.error.log
03-13 11:15:19 ERROR (25631:zlog.c:802) ------zlog_profile start------
03-13 11:15:19 ERROR (25631:zlog.c:803) init_flag:[1]
03-13 11:15:19 ERROR (25631:level.c:246) level:* = 0, 6
03-13 11:15:19 ERROR (25631:level.c:246) level:DEBUG = 20, 7
03-13 11:15:19 ERROR (25631:level.c:246) level:INFO = 40, 6
03-13 11:15:19 ERROR (25631:level.c:246) level:NOTICE = 60, 5
03-13 11:15:19 ERROR (25631:level.c:246) level:WARN = 80, 4
03-13 11:15:19 ERROR (25631:level.c:246) level:ERROR = 100, 3
03-13 11:15:19 ERROR (25631:level.c:246) level:FATAL = 120, 1
03-13 11:15:19 ERROR (25631:level.c:246) level:UNKNOWN = 254, 3
03-13 11:15:19 ERROR (25631:level.c:246) level:! = 255, 6
```

That shows the levels. If user has defined levels, shows here also.

```
03-13 11:15:19 ERROR (25631:conf.c:442) ---conf[0x7fbcbed14520]---
03-13 11:15:19 ERROR (25631:conf.c:443) file:[test_profile.conf],mtime:[2012-
```

That shows which configure file is read, and what's it last modified time. When you have doubt with conf file and conf memory not equals, check their time and path first. In this example, use ls -l test_profile.conf, and compare them.

```
03-13 11:15:19 ERROR (25631:conf.c:444) ignore_error_format_rule:[0]
03-13 11:15:19 ERROR (25631:conf.c:445) buf_size_min:[1024]
03-13 11:15:19 ERROR (25631:conf.c:446) buf_size_max:[0]
03-13 11:15:19 ERROR (25631:conf.c:447) rotate_lock_file:[/tmp/zlog.lock]
03-13 11:15:19 ERROR (25631:conf.c:449) default_format:
03-13 11:15:19 ERROR (25631:format.c:183) format:[0x14e82e0][default]-[%d(%F
```

Here shows some global setting.

```
03-13 11:15:19 ERROR (25631:conf.c:452) ---rules[0x14d5120]---
03-13 11:15:19 ERROR (25631:rule.c:718) rule:[0x14f8320][my_cat.0]-[,0][0x14f
03-13 11:15:19 ERROR (25631:conf.c:457) ---formats[0x14d5420]---
03-13 11:15:19 ERROR (25631:format.c:183) format:[0x14f0300][simple]-[%m%n]
```

rules and formats are not so distinct.

```
03-13 11:15:19 ERROR (25631:thread.c:260) ---tmap[0x7fbcbed16570]---
03-13 11:15:19 ERROR (25631:thread.c:263) thread:[140448634054400]
03-13 11:15:19 ERROR (25631:category.c:273) ---cmap[0x7fbcbed16578]---
03-13 11:15:19 ERROR (25631:category.c:276) category:[my_cat]
```

tmap is thread map. It has one thread.
cmap is category map. It has one category – my_cat

```
03-13 11:15:19 ERROR (25631:zlog.c:810) default_category[my_cat]
```

default category of dzlog is printed at last.

```
03-13 11:15:19 ERROR (25631:zlog.c:812) ------zlog_profile end------
```

6.3 User defines level

Here I write down all steps of how user define own levels.

1. define levels in configure file.

```
$ cat $(top_builddir)/test/test_level.conf
@ignore_error_format_rule false
@buf_size_min 1024
@buf_size_max 0
@rotate_lock_file /tmp/zlog.lock
@default_format "%V %v %m%n"
```

```
@level TRACE = 30, LOG_DEBUG

my_cat.TRACE >stdout;
```

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

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

Now in zlog, a integer(30) and a level string(TRACE) represents a level. Note that this integer must be in [1,253], other number is illegal. More larger, more important.Now 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 generate captial of level string and "%v" for lowercase of level string.

Besides, in level defination LOG_DEBUG means when use >syslog in 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__, __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, args...) \
    zlog(cat, __FILE__, __LINE__, ZLOG_LEVEL_TRACE, format, ##args)
#endif
```

3. Now, 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 configure file only allows >=TRACE ouput to screen. And "%V" and "%v" works well.