zlog使用攻略

难易 著[1][2]

March 27, 2012

[1]适用于 zlog v0.9.
[2]有问题，写邮件到HardySimpson1984@gmail.com

Contents

Chapter 1

zlog是什么？

zlog是一个高性能、线程安全、灵活、概念清晰的纯C日志函数库。

事实上，在C的世界里面没有特别好的日志函数库（就像JAVA里面的的log4j，或者C++的log4cxx）。C程序员都喜欢用自己的轮子。printf就是个挺好的轮子，但没办法通过配置改变日志的格式或者输出文件。syslog是个系统级别的轮子，不过速度慢，而且功能比较单调。

所以我写了zlog。

zlog有这些特性：

- syslog风格的配置文件，易学易用
- 可以灵活配置日志输出的格式，类似于log4j的pattern layout
- 纲目分类模型，比log4j系列的继承模型更加清晰
- 多种输出，包括动态文件、静态文件、stdout、stderr、syslog
- 可以在运行时动态刷新配置，只需要调用函数zlog_update()
- 高性能，在我的笔记本上达到72'000条日志每秒，大概是syslog(3)配合rsyslogd的200倍速度
- 用户可以自定义等级，无需改变库代码6.3
- 多线程和多进程写同一个日志的情况下，安全转档（转档就是把太大的日志重命名）
- 可以精确到微秒

- 如果一个程序默认只用一个分类输出，zlog提供了简单的调用包装dzlog
- MDC，一个log4j风格的键-值对的表，可以扩展用户自定义的字段
- 自诊断，可以在运行时输出zlog自己的日志和配置状态
- 不依赖其他库，只要是个POSIX系统就成

相关链接：

软件下载：https://github.com/HardySimpson/zlog/zipball/master

详细使用攻略：https://github.com/HardySimpson/zlog/raw/master/download/ZlogUsersGuide-CN-0.9rc1.pdf

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

作者博客：http://my.oschina.net/HardySimpson/blog

邮箱：HardySimpson1984@gmail.com

Chapter 2

zlog不是什么？

zlog的目标是成为一个简而精的日志函数库，她不会直接支持网络输出或者写入数据库，她不会直接支持日志内容的过滤和解析。

原因很明显，日志库是被应用程序调用的，所有花在日志库上的时间都是应用程序运行时间的一部分，而上面说的这些操作都很费时间，会拖慢应用程序的速度。这些事儿应该在别的进程或者别的机器上做。

如果你需要这些特性，我建议使用rsyslog，一个系统级别的日志搜集、过滤、存储软件，当然这是单独的进程，不是应用程序的一部分。

目前我正在考虑怎么让zlog和rsyslog合适的对接，传统的syslog(3)行，但不够优雅。当然我也考虑让应用程序挂一些输出函数到zlog上。有什么好想法，告诉我！

Chapter 3

Hello World

3.1 编译和安装zlog

如果是从Git上拉下来，而且你想试试自己环境的auto tools的话(https://github.com/HardySimpson/zlog):

```
$ sh autogen.sh
```

一般来说

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

如果--enable-test用了的话，测试程序会被编译。这些程序是很好的zlog的使用案例，如果你在别的目录编译，编译完之后要把测试目录的配置复制过来，例如：

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

3.2 应用程序调用和链接zlog

应用程序使用zlog很简单，只要在C文件里面加一行。

```
#include ‘‘zlog.h’’
```

如果你的系统有pkgconfig，可以在makefile里面这么写

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

如果没有的话，手工链接的命令是

```
$ 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 代码

这些代码在$(top_builddir)/test/test_hello.c, test_hello.conf

1. 写一个C文件：

```
$ 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. 写一个配置文件，放在和test_hello.c同样的目录下:

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

my_cat.DEBUG            >stdout; simple
```

3. 编译、然后运行!

```
$ 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 更简单的Hello World

这个例子在$(top_builddir)/test/test_default.c, test_default.conf. 源代码是：

```
#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;
    }
```

配置文件是test_default.conf，和test_hello.conf一模一样，最后执行程序的输出也一样。区别在于这里用了dzlog API，内含一个默认的zlog_category_t。详见5.6。

Chapter 4

配置文件

大部分的zlog的行为取决于配置文件：把日志打到哪里去，用什么格式，怎么转档。配置文件是zlog的黑话，我尽量把这个黑话设计的简单明了。

4.1 全局设置

全局设置以@开头。全局设置可以不写，使用内置的默认值。语法：@[键][n个空格或tab][值]

如果ignore_error_format_rule是true，zlog_init()将会忽略错误的format和rule。否则zlog_init()将会严格的检查所有format和rule的语法 ，碰到错误的就返回-1。默认值：

```
@ignore_error_format_rule    false
```

zlog在堆上为每个线程申请缓存。buf_size_min是单个缓存的最小值。写日志的时候，如果日志长度大于缓存，缓存会自动扩充到buf_size_max，日志再长超过buf_size_max就会截断。如果 buf_size_max 是 0，意味着不限制缓存，每次扩充buf_size=2*buf_size。缓存大小可以加上 KB，MB 或 GB这些单位。默认值：

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

rotate_lock_file是一个posix文件锁文件。用来在多进程情况下，保证日志安全转档。zlog会在zlog_init()时候创建这个文件。确认你执行

程序的用户有权限创建和读写这个文件。如果有多个用户需要转档同一个日志文件，确认这个锁文件对于多个用户都可读写。默认值：

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

default_format是缺省的日志格式。内置的缺省格式会产生类似这样的输出：

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

可以把缺省格式设成你喜欢的样子，默认值：

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

用户自定义级别

zlog内置的级别是：

```
@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
```

语法：

@level[n个空格或tab][级别的字符串] = [级别数值][syslog级别，可选]

级别数值要在 [1,253]内， 越大代表越重要。syslog级别是可选，不设的话默认是LOG_DEBUG。详见6.3。

4.2 分类(Category)、规则(Rule)和格式(Format)

C代码片段是：

```
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 */
```

以及配置文件是：

```
&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.
```

最后的输出是：

```
hello, zlog
```

分类(Category)用于区分不同的输入。代码中的分类变量的名字是一个字符串，在一个程序里面可以通过获取不同的分类名的category用来后面输出不同分类的日志，用于不同的目的。

格式(Format)是用来描述输出日志的格式，比如是否有带有时间戳，是否包含文件位置信息等，上面的例子里面的格式simple就是简单的用户输入的信息+换行符。

规则(Rule)则是把分类、级别、输出文件、格式组合起来，决定一条代码中的日志是否输出，输出到哪里，以什么格式输出。简单而言，规则里面的分类字符串和代码里面的分类变量的名字一样就匹配，当然还有其他情况，见后面的4.3.2。

所以，当程序运行这样一条语句的时候

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

首先zlog库会找到c对应的分类的名字，然后看这个分类在配置文件中有那几条规则是匹配的，目前这个分类拥有的规则是

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

然后检查目前的日志级别是否满足来决定是否输出，因为INFO级别大于DEBUG，所以这条日志要被输出。并且根据这条规则，会被输出到stdout(标准输出)，并且输出的格式是simple这个格式，也就是配置文件中这条决定的

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

最后在屏幕上输出

```
hello, zlog
```

这就是整个过程。用户只需要写自己的消息，最后日志发送到哪里，以什么格式，都由zlog库来完成。

分类变量和规则(rule)是松耦合的。如果有两条上面的规则，则ZLOG_INFO(c, "hello, zlog")会打印出两条''hello, zlog''。如果规则被误写为''my_ca.debug >stdout; simple''，没有东西会被输出。松耦合是同时易错并灵活的。当然，zlog有办法弥补这一点，见4.3.2。

规则(rule)和格式(format)是紧耦合的，如果规则(rule)使用了一条不存在的格式(format)，zlog_init()会失败，除非在配置文件中设置''@ignore_error_format_rule false''。

格式(format)的语法是：

&[格式名][n个空格或tab]"[转换格式串]"

规则的语法是：

[选择器][n个空格或tab][输出器]

[选择器] = [分类字符串].[级别]

[输出器] = [输出], [文件大小个数限制, 可选]; [format(格式)名, 可选]

4.3 选择输入

一条日志最终会不会被zlog打印，取决于源代码中的分类变量、级别和配置文件中的分类字符串、级别是否匹配。

[选择器] = [分类名].[级别]

4.3.1 级别匹配

zlog有6个默认的级别："DEBUG", "INFO", "NOTICE", "WARN", "ERROR"和"FATAL"。就像其他的日志函数库那样， aa.DEBUG意味着任何大于等于DEBUG级别的日志会被输出。当然还有其他的表达式。配置文件中的级别是大小写不敏感的。

表达式	含义
*	所有等级
aa.debug	代码内等级>=debug
aa.=debug	代码内等级==debug
aa.!debug	代码内等级!=debug

用户可以自定义等级，详见6.3。

4.3.2 分类匹配

终于到这里了，zlog的代码分类变量和配置文件中规则的分类字符串的匹配是比较特殊的。表面看起来和log4j差不多，但内涵不一样。我称之为纲目分类模型。

在log4j里面有父logger和子logger。比如说一个log4j的配置文件是这样。

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

aa是一个父logger。aa.bb和aa.cc是aa，aa.bb继承了aa's等级和输出(appender)，等级是DEBUG输出是A1。aa.cc继承并覆盖了aa的级别。所以aa.cc的等级是INFO输出还是A1。那么，如果我现在想把所有的aa的日志输出到一个特别的文件里面去，并保持原来的aa.bb、aa.cc的输出不变呢？log4j的表达式就会很复杂，还要引入阈值(threshold)，把阈值设到输出上去……

在zlog里面，所有的规则之间都是独立的，没有父子关系。纲目分类的关系表现在分类字符串中间的下划线。举例：

```
#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’’
```

没有继承，只有4条独立的规则

如果代码里面的分类名是''aa_bb''。代码就像这样：

```
category_t ab;
ab = zlog_get_category(‘‘aa_bb’’);
ZLOG_DEBUG(ab, ‘‘ab’s debug’’);
ZLOG_ERROR(ab, ‘‘ab’s error’’);
```

配置中，rule 1的分类字符串''aa_bb''和rule 3的分类字符串''aa_''，与代码中的名字为''aa_bb''的分类变量是匹配的。于是代码中属于aa_bb分类变量、>=DEBUG日志输出到aa_bb.log，属于aa_bb分类变量、>=ERROR会被输出到 aa_error.log。ERROR等级的日志会被同时写在两个文件里面。不过rule 4的变量字符串是''aa''，它不匹配分类变量''aa_bb''，它精确匹配拥有''aa''名字的分类变量。

这就是纲目分类模型。规则和规则之间是分开的。一个代码分类变量可以匹配多个规则分类字符串，一个规则可以属于多个代码分类变量。规则中的纲分类字符串(以下划线结尾的)匹配代码中的目分类变量，纲分类字符串的范围包括了了目分类字符串。这样，用户可以选择任意范围的纲目分类字符串来输出，而不影响其他规则的行为。

事实上，在zlog_get_category()被调用的时候，并不保证代码分类变量一定有相匹配的规则分类字符串。分类变量可以有很多与之匹配的规则，也有可能一条都没有，这取决于配置文件是怎么写的。当配置文件改变并调用zlog_update()后，代码分类变量和规则分类字符串的匹配会被重新计算。根据上面所说的匹配方式，每个分类变量都会从新的配置文件里面找到自己的新规则。

分类误写的问题，可以通过垃圾箱规则来弥补。规则的分类名为"!"的就是垃圾箱规则，匹配那些找不到规则的代码分类。例如zlog_get_category(''xx'')遍历所有规则，但没有发现匹配的规则，然后有一个规则的分类名是'!'，于是匹配这条垃圾箱规则。垃圾箱规则是唯一的，只有最后一条带"!"的规则才是有用的。用户可用这个表达式，在运行的时候把误写的都找出来。这个规则可以这么写：

```
!.*                ‘‘/var/log/zlog.mismatch.log’’
```

用表格来总结分类匹配

总结	配置文件规则分类	匹配的代码分类	不匹配的代码分类
*匹配所有	*.*	aa, aa_bb, aa_cc, xx, yy ...	NONE
以_结尾的分类匹配纲、目分类	aa_.*	aa, aa_bb, aa_cc, aa_bb_cc	xx, yy
不以_结尾的匹配目分类	aa.*	aa	aa_bb, aa_cc, aa_bb_cc
!匹配那些没有找到规则的分类	!.*	xx	aa(as it matches rules above)

4.4 输出动作

目前zlog支持4种输出，语法是：

[输出器] = [输出], [文件大小个数限制，可选]; [format(格式)名，可选]

动作	输出字段	文件限制字段
标准输出	>stdout	无意义
标准错误输出	>stderr	无意义
输出到syslog	>syslog	syslog设施(facilitiy)：LOG_USER(default), LOG_LOCAL[0-7] 必填
文件	''文件路径''	文件大小个数限制 1000, 1k, 2M, 1G... 3m*2, 4k*3...

文件路径 可以是相对路径或者绝对路径，被双引号"包含。转换格式串可以用在文件路径上。例如文件路径是 ''%E(HOME)/log/out.log''，

环境变量$HOME是/home/harry，那最后的输出文件是/home/harry/log/output.log。转换格式串详见 4.5。

文件限制 控制文件的大小和个数。zlog根据这个字段来转档，当日志文件太大的时候。例如

```
‘‘%E(HOME)/log/out.log’’,1M*3
```

如果out.log被写到1M大，转档动作为：

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

如果新的日志文件再次被写满，转档动作为：

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

下一次的转档会把最旧的日志文件删除掉， *3意味着zlog会保留3个文件：

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

最旧的文件有最大的序列号

格式名 是可选的，如果不写，用全局配置里面的默认格式：

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

这种格式会输出类似这样的日志：

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

规则可以用自己的格式，一个格式可以被多条规则使用。如果在某条规则前写这样的格式：

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

最后的输出是：

```
hello, zlog
```

格式(format)的语法是：

&[格式名][n个空格或tab]"[转换格式串]"

4.5 转换格式串

转换格式串的设计是从C的printf函数里面抄来的。一个转换格式串由文本字符和转换说明组成。

转换格式串用在规则的日志文件路径和输出格式(format)中。

你可以把任意的文本字符放到转换格式串里面。

每个转换说明都是以百分号(%)打头的，后面跟可选的宽度修饰符，最后以转换字符结尾。转换字符决定了输出什么数据，例如分类名、级别、时间日期、进程号等等。宽度修饰符控制了这个字段的最大最小宽度、左右对齐。下面是简单的例子。

如果转换格式串是：

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

源代码中的写日志语句是：

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

将会输出：

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

可以注意到，在文本字符和转换说明之间没有显式的分隔符。zlog解析的时候知道哪里是转换说明的开头和结尾。在这个例子里面%-5p这个转换说明决定了日志级别要被左对齐，占5个字符宽。

4.5.1 转换字符

可以被辨认的转换字符是

字符	效果	例子
%c	分类名	aa_bb
%d	打日志的时间。这个后面要跟一对小括号()内含说明具体的日期格式。就像%d(%F)或者%d(%m-%d %T)。如果不跟小括号，默认是%d(%F %T)。括号内的格式和 strftime(3)的格式一致，就是增加了%us表示微秒，还有 %ms表示毫秒。详见4.5.3	%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	环境变量，后面也要跟小括号，内涵环境变量的键。变量的值在zlog_init()的时候就求出来了	%E(HOME) /home/harry
%F	源代码文件名，来源于__FILE__宏。在某些编译器下 __FILE__是绝对路径。用$f来去掉目录只保留文件名，或者编译器有选项可以调节	test_hello.c
%f	源代码文件名，输出$F最后一个'/'后面的部分。当然这会有一定的性能损失	test_hello.c
%H	主机名，来源于 gethostname(2)	zlog-dev
%L	源代码行数，来源于__LINE__宏	135
%m	用户日志，用户从zlog函数输入的日志。	hello, zlog
%M	MDC (mapped diagnostic context)，每个线程一张键值对表，输出键相对应的值。后面必需跟跟一对小括号()内含键。例如%M(clientNumber)，clientNumbe是键。 详见 6.1	%M(clientNumber) 12345
%n	换行符，目前还不支持windows换行符	\n
%p	进程ID，来源于getpid()	2134
%V	日志级别，大写	INFO
%v	日志级别，小写	info
%t	线程ID，来源于pthread_self().	7636
%%	一个百分号	%
%[其他字符]	解析为错误，zlog_init()将会失败	

4.5.2 宽度修饰符

一般来说数据按原样输出。不过，有了宽度修饰符，就能够控制最小字段宽度、最大字段宽度和左右对齐。当然这要付出一定的性能代价。

可选的宽度修饰符放在百分号和转换字符之间。

第一个可选的宽度修饰符是左对齐标识，减号(-)。然后是可选的最

小字段宽度，这是一个十进制数字常量，表示最少有几个字符会被输出。如果数据本来没有那么多字符，将会填充空格（左对齐或者右对齐）直到最小字段宽度为止。默认是填充在左边也就是右对齐。当然你也可以使用左对齐标志，指定为填充在右边来左对齐。填充字符为空格(space)。如果数据的宽度超过最小字段宽度，则按照数据的宽度输出，永远不会截断数据。

这种行为可以用最大字段宽度来改变。最大字段宽度是放在一个句点号(.)后面的十进制数字常量。如果数据的宽度超过了最大字段宽度，则尾部多余的字符（超过最大字段宽度的部分）将会被截去。 最大字段宽度是8，数据的宽度是10，则最后两个字符会被丢弃。假如这种行为和C的printf是一样的，把后面的部分截断。

下面是各种宽度修饰符和分类转换字符配合一起用的例子。

宽度修饰符	左对齐	最小字段宽度	最大字段宽度	附注
%20c	否	20	无	左补充空格，如果分类名小于20个字符长。
%-20c	是	20	无	右补充空格，如果分类名小于20个字符长。
%.30c	无	无	30	如果分类名大于30个字符长，取前30个字符，去掉后面的。
%20.30c	否	20	30	如果分类名小于20个字符长，左补充空格。如果在20-30之间，按照原样输出。如果大于30个字符长，取前30个字符，去掉后面的。
%-20.30c	是	20	30	如果分类名小于20个字符长，右补充空格。如果在20-30之间，按照原样输出。如果大于30个字符长，取前30个字符，去掉后面的。

4.5.3 时间字符

这里是转换字符d支持的时间字符。

其中2个是zlog特有的，取自gettimeofday(2)

词	效果	例子
%ms	毫秒，3位数字字符串	013
%us	微秒，6位数字字符串	002323

其他的字符都是由strftime(2)生成的，在我的linux操作系统上支持的是：

字符	效果	例子
%a	一星期中各天的缩写名，根据locale显示	Wed
%A	一星期中各天的全名，根据locale显示	Wednesday
%b	缩写的月份名，根据locale显示	Mar
%B	月份全名，根据locale显示	March
%c	当地时间和日期的全表示， 根据locale显示	Thu Feb 16 14:16:35 2012
%C	世纪（年/100），2位的数字(SU)	20
%d	一个月中的某一天（01-31）	06
%D	相当于%m/%d/%y.（呃，美国人专用，美国人要知道在别的国家%d/%m/%y 才是主流。也就是说在国际环境下这个格式容易造成误解，要少用）(SU)	02/16/12
%e	就像%d，一个月中的某一天，但是头上的0被替换成空格(SU)	6
%F	相当于%Y-%m-%d（ISO 8601日期格式）(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) 大意是采用%V定义的年，如果那年的前几天不算新年的第一周，就算上一年	2012
%g	相当于%G，就是不带世纪（00-99). (TZ)	12
%h	相当于%b(SU)	Feb
%H	小时，24小时表示(00-23)	14
%I	小时，12小时表示(01-12)	02
%j	一年中的各天(001-366)	047
%k	小时，24小时表示(0-23)； 一位的前面为空格（可和%H比较）(TZ)	15

%l	小时，12小时表示(0-12)； 一位的前面为空格 (可和%比较)(TZ)	3
%m	月份(01-12)	02
%M	分钟(00-59)	11
%n	换行符 (SU)	\n
%p	"AM" 或 "PM"，根据当时的时间，根据locale显示相应的值，例如''上午“、” 下午 “ 。 中午是"PM"，凌晨是"AM"	PM
%P	相当于%p不过是小写，根据locale显示相应的值 (GNU)	pm
%r	时间+后缀AM或PM。在POSIX locale下相当于%I:%M:%S %p. (SU)	03:11:54 PM
%R	小时(24小时制):分钟 (%H:%M) (SU) 如果要带秒的，见%T	15:11
%s	Epoch以来的秒数，也就是从1970-01-01 00:00:00 UTC. (TZ)	1329376487
%S	秒(00-60). (允许60是为了闰秒)	54
%t	制表符tab(SU)	
%T	小时(24小时制):分钟:秒 (%H:%M:%S) (SU)	15:14:47
%u	一周的天序号(1-7)，周一是1，另见%w (SU)	4
%U	一年中的星期序号(00-53)，周日是一周的开始，一年中第一个周日所在的周是第01周。另见%V和%W	07
%V	ISO 8601星期序号(01-53)，01周是第一个至少有4天在新年的周。另见%U 和%W(SU)	07
%w	一周的天序号(0-6)，周日是0。另见%u	4
%W	一年中的星期序号(00-53)，周一是一周的开始，一年中第一个周一所在的周是第01周。另见%V和%W	07
%x	当前locale下的偏好日期	02/16/12
%X	当前locale下的偏好时间	15:14:47
%y	不带世纪数目的年份(00-99)	12
%Y	带世纪数目的年份	2012

%z	当前时区相对于GMT时间的偏移量。采用RFC 822-conformant来计算(话说我也不知道是啥) (using "%a, %d %b %Y %H:%M:%S %z"). (GNU)	+0800
%Z	时区名(如果有的话)	CST
%%	一个百分号	%

4.6 配置文件工具

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

zlog-chk-conf 尝试读取配置文件，检查语法，然后往屏幕上输出这些配置文件是否正确。我建议每次创建或者改动一个配置文件之后都用一下这个工具。输出可能是这样：

```
$ ./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
```

这个告诉你配置文件zlog.conf的126行，是错的。第一行进一步告诉你[aaa]不是一条正确的规则。

zlog-chk-conf可以同时分析多个配置文件，举例：

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

还有一个有用的工具是

```
$ 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
```

这个工具会产生一个模板配置文件，可以带中文或者英文的注释，注释是这一章内容的浓缩总结。

Chapter 5

zlog应用编程接口(API)

zlog的所有函数都是线程安全的，使用的时候只需要

```
#include ‘‘zlog.h’’
```

5.1 初始化和清理

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

zlog_init() 从配置文件中读取配置信息到内存。如果conf_file为NULL，所有日志以内置格式写到标准输出上。如果zlog_init()失败，详细错误会被记录在ZLOG_PROFILE_ERROR(see 6.2)，并且返回-1。每个进程只有第一次调用zlog_init()是有效的，后面的多余调用都会失败。

zlog_update()是用来重载配置，并根据这个配置文件来重计算内部的分类规则匹配、重建每个线程的缓存。可以在配置文件发生改变后调用这个函数。这个函数使用次数不限。如果conf_file 为NULL，会重载上一次zlog_init()或者zlog_update()使用的配置文件。如果失败，返回-1，同时zlog后续无法使用。

zlog_fini()清理所有zlog API申请的内存，关闭它们打开的文件。

5.2 分类(Category)操作

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

zlog_get_category()从全局的分类表里面取得一个分类，如果没有的话，创建一个。成功返回分类的指针，失败则返回NULL。category_name需要和配置文件的规则想对应，详见4.3。不用担心内存释放，zlog_fini()最后会进行清理。

5.3 写日志函数

```
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);
```

这3个函数是用户写日志的函数，输入的数据对应于配置文件中的%m。a_cat来自于调用zlog_get_category()。

zlog()和vzlog()根据format输出，就像printf(3)和vprintf(3)。

vzlog()相当于zlog()，只是它用一个va_list类型的参数args，而不是一堆类型不同的参数。vzlog() 内部使用了 va_copy 宏，args的内容在vzlog()后保持不变，可以参考stdarg(3)。

hzlog()有点不一样，它产生下面这样的输出，长度为buf_len的内存buf以16进制的形式表示出来。

```
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
```

参数file、line一般填写为__FILE__，__LINE__宏。这两个参数表明了日志是在哪里记录的。

level 是一个int型变量，默认在zlog.h里面定义为

```
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 写日志宏

为了简单使用，在zlog.h内定义了如下宏

zlog()的宏

```
#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)
```

vzlog()的宏

```
#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)
```

hzlog()的宏

```
#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操作

MDC(Mapped Diagnostic Context)是一个每线程拥有的键-值表，所以和分类没啥关系。操作函数为

```
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);
```

键和值都是字符串，长度不能超过MAXLEN_PATH(一般来说是1024)。记住这个表是和线程绑定的，每个线程有自己的表，所以在一个线程内的操作不会影响其他线程。

zlog_put_mdc()成功返回0，失败返回-1。zlog_get_mdc()成功返回value的指针，失败或者没有相应的key返回NULL。

5.6 dzlog应用编程接口(API)

dzlog是忽略分类(zlog_category_t)的一组zlog接口。它采用内置的一个默认分类，这个分类置于锁的保护下。这些接口也是线程安全的。忽略了分类，意味着用户不需要操心创建、存储、传输zlog_category_t类型的变量。当然也可以在用dzlog接口的同时用一般的zlog接口函数。

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

dzlog_init()和zlog_init()一样做初始化，就是多需要一个默认分类名(default_category_name)的参数。zlog_update()、 zlog_fini()可以和以前一样使用，用来刷新配置，或者清理。

dzlog_set_category()是用来改变默认分类用的。上一个分类会被替换成新的。同样不用担心内存释放的问题，zlog_fini()最后会清理。

```
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);
```

这是dzlog相应的3个写日志的函数。

```
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)
```

同样也有相应的宏，一般都用这组来写。

Chapter 6

高阶使用

6.1 MDC

MDC是什么？在log4j里面解释为Mapped Diagnostic Context。听起来是个很复杂的技术，其实MDC就是一个键-值对表。一旦某次你设置了，后面库可以帮你自动打印出来，或者成为文件名的一部分。让我们看一个例子，来自于$(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;

}
```

配置文件

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

输出

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

你可以看到zlog_put_mdc()在表里面设置键''myname''对应值''Zhang''，然后在配置文件里面%M(myname)指出了在日志的哪个位置需要把值打出来。第二次，键''myname''的值被覆盖写成''Li''，然后日志里面也有相应的变化。

MDC什么时候有用呢？往往在用户需要在同样的日志行为区分不同的业务数据的时候。比如说，c源代码是

```
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’’);
```

在配置文件里面是

```
&format ‘‘%M(customer_name) %m%n’’
```

当程序同时处理两个用户的时候，打出来的日志可能是

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

这样，你就可以用grep命令把这两个用户的日志分开来了

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

或者，还有另外一条路，一开始在文件名里面做区分，看配置文件：

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

这就会产生3个日志文件。

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

这是一条近路，如果用户知道自己在干什么。

MDC是每个线程独有的，所以可以把一些线程专有的变量设置进去。如果单单为了区分线程，可以用转换字符里面的%t来搞定。

6.2 诊断zlog本身

OK，至今为止，我假定zlog库本身是不出毛病的。zlog帮助用户程序写日志，帮助程序员debug程序。但是如果zlog内部出错了呢？怎么知道错在哪里呢？其他的程序可以用日志库来debug，但日志库自己怎么debug？答案很简单，zlog有自己的日志——诊断日志。这个日志通常是关闭的，可以通过环境变量来打开。

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

诊断日志只有两个级别debug和error。设置好环境变量后. 再跑test_hello程序3.3，然后debug日志为

```
$ 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日志没产生，因为没有错误发生。

你可以看出来，debug日志展示了zlog是怎么初始化还有清理的。不过在ZLOG_INFO()执行的时候没有日志打出来，这是从效率出发。

如果zlog库有任何问题，都会打日志到ZLOG_PROFILE_ERROR所指向的错误日志。比如说，在ZLOG_INFO()上用一个错误的printf的语法：

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

然后编译执行程序，ZLOG_PROFILE_ERROR的日志会是

```
$ 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.
```

这样，用户就能知道为啥期待的输出没有产生，然后搞定这个问题。

运行时诊断会带来一定的性能损失。一般来说，我在生产环境把ZLOG_PROFILE_ERROR打开，ZLOG_PROFILE_DEBUG关闭。

还有另外一个办法来诊断zlog。我们都知道，zlog_init()会把配置信息读入内存。在整个写日志的过程中，这块内存保持不变。如果用户程序因为某种原因损坏了这块内存，那么就会造成问题。还有可能是内存中的信息和配置文件的信息不匹配。所以我设计了一个函数，把内存的信息展现到ZLOG_PROFILE_ERROR指向的错误日志。

代码见$(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()就是这个函数。配置文件很简单。

```
$ 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
```

然后zlog.error.log会是

```
$ 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
```

这里显示了所有的等级，如果用户有自定义等级，也会在这里展示。

```
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-
```

这里显示了哪一个配置文件会被读进来，以及这个文件最后修改时间。如果你对配置文件和内存中的配置信息是否一致有疑问，先比较修改时间和文件路径。在这里例子里面，`ls -l test_profile.conf`，然后和上面的数据比较。

```
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
```

这里显示一些全局配置

```
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]
```

规则和格式不是那么明显

```
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是线程表，目前看来有一个线程。
cmap是分类表，有一个分类 -- my_cat

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

最后把dzlog默认的分类打印出来

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

6.3 用户自定义等级

这里我把用户自定义等级的几个步骤写下来。

1. 在配置文件中定义新的等级

```
$ 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;
```

内置的默认等级是(这些不需要写在配置文件里面)

```
@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
```

这样在zlog看来，一个整数(30)还有一个等级字符串(TRACE)代表了等级。这个整数必须位于[1,253]之间，其他数字是非法的。数字越大代表越重要。现在TRACE比DEBUG重要(30>20)，比INFO等级低(30<40)。在这样的定义后，TRACE就可以在下面的配置文件里面用了。例如这句话：

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

意味着等级>=TRACE的，包括INFO, NOTICE, WARN, ERROR, FATAL会被写到标准输出。

格式里面的转换字符''%V''会产生等级字符串的大写输出，''%v''会产生小写的等级字符串输出。

另外，在等级的定义里面，LOG_DEBUG是指当需要输出到syslog的时候，自定义的TRACE等级会以LOG_DEBUG输出到syslog。

2. 在源代码里面直接用新的等级是这么搞的

```
zlog(cat, __FILE__, __LINE__, 30, ‘‘test %d’’, 1);
```

为了简单使用，创建一个.h头文件

```
$ 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. 这样ZLOG_TRACE就能在.c文件里面用了

```
$ 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. 最后我们能看到输出

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

正是我们所期待的，配置文件只允许>=TRACE等级的日志输出到屏幕上。%V''和''%v''也显示了正确的结果。