zlog[1]使用手册

难易 著[2][3]

July 19, 2012

[1]星星之火，可以燎原——毛泽东
[2]适用于 zlog v1.1.x
[3]有问题，在github上开个issue，或者写邮件到HardySimpson1984@gmail.com

Contents

Chapter 1

zlog是什么？

zlog是一个高可靠性、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在效率、功能、安全性上大大超过了log4c，并且是用c写成的，具有比较好的通用性。

zlog有这些特性：

- syslog分类模型，比log4j模型更加直接了当
- 日志格式定制，类似于log4j的pattern layout
- 多种输出，包括动态文件、静态文件、stdout、stderr、syslog、用户自定义输出函数
- 运行时手动、自动刷新配置文件（同时保证安全）
- 高性能，在我的笔记本上达到25万条日志每秒，大概是syslog(3)配合rsyslogd的1000倍速度
- 用户自定义等级
- 多线程和多进程环境下保证安全转档
- 精确到微秒
- 简单调用包装dzlog（一个程序默认只用一个分类）
- MDC，线程键-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(当然还要一个C99兼容的vsnprintf)

相关链接：

软件下载：https://github.com/downloads/HardySimpson/zlog/zlog-latest-stable.tar.gz
源代码：git@github.com:HardySimpson/zlog.git
使用手册(pdf)：https://github.com/downloads/HardySimpson/zlog/UsersGuide-CN.pdf
问题讨论区：https://github.com/HardySimpson/zlog/issues
英文主页：http://hardysimpson.github.com/zlog/
中文主页：http://www.oschina.net/p/zlog
作者博客：http://my.oschina.net/HardySimpson/blog
邮箱：HardySimpson1984@gmail.com

1.1 zlog 1.1 发布说明

1. zlog是基于POSIX的。目前我手上有的环境只有AIX和linux。在其他的系统下（FreeBSD, NetBSD, OpenBSD, OpenSolaris, Mac OS X...）估计也能行，有问题欢迎探讨。

2. zlog使用了一个C99兼容的vsnprintf。也就是说如果缓存大小不足，vsnprintf将会返回目标字符串应有的长度（不包括'\0'）。如果在你的系统上vsnprintf不是这么运作的，zlog就不知道怎么扩大缓存。如果在目标缓存不够的时候vsnprintf返回-1，zlog就会认为这次写入失败。幸运的是目前大多数c标准库符合C99标准。glibc 2.1,libc on AIX, libc on freebsd...都是好的，不过glibc2.0不是。在这种情况下，用户需要自己来装一个C99兼容的vsnprintf，来crack这个函数库。我推荐ctrio, 或者C99-snprintf。只要改buf.c就行，祝好运！

3. zlog 1.1在库方面是和zlog 1.0二进制兼容的，在配置文件方面有两个的改动，

   (a) %d(%ms,%us)不再是合法的，%d.%ms是合法的。%ms毫秒和%us微秒提出来作为单独的转换字符串。

   (b) 增加了默认时间串%D，输出"2012-02-14 17:03:12"这样的时间。

   之所以改变这些，是为了简化代码，以及能提升30%的性能。所以如果你是从1.0.x升级而且使用动态库的话，无须重新编译程序，只要稍微改动一下配置文件即可。

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，不会直接支持网络输出或者写入数据库，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zlog已经支持用户自定义输出，可以自己实现一个输出函数，自由的把日志输出到其他进程或者其他机器。而把日志的分类匹配、日志格式成型的工作交给zlog。

Chapter 3

Hello World

3.1 编译和安装zlog

下载zlog-latest-stable.tar.gz

```
$ tar -zxvf zlog-latest-stable.tar.gz
$ cd zlog-1.0.x/
$ ./configure --prefix=[where u wanna install it] \
      --enable-test
$ make
$ sudo make install
```

--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。

```
# add one line
$ sudo vi /etc/ld.so.conf
/usr/local/lib
$ sudo ldconfig
```

在你的程序运行之前，保证libzlog.so在系统的动态链接库加载器可以找到的目录下。上面的命令适用于linux，别的系统自己想办法。

- 对于zlog库开发者，如果你在别的目录而不是在源代码目录编译(平行编译，parallel building)，编译完之后要把源代码测试目录的配置文件复制过来。否则运行测试程序会报错，找不到配置文件。

```
$ tar -zxvf zlog-x.x.x.tar.gz
$ mkdir build && cd build
$ ../zlog-1.0.x/configure --prefix=[where u wanna install it] \
      --enable-test
```

```
$ cp ../zlog-1.0.x/test/*.conf ./test/
$ cd test
$ ./test_hello
hello, zlog
```

- 对于zlog库开发者，如果是从github上直接下载，那就不会有configure这个脚本，因为这个脚本是由auto tools从configure.ac等源文件生成的。这时候你的环境上就需要有automake, autoconf等工具来生成configure

```
$ git clone git@github.com:HardySimpson/zlog.git
```

或者从这个地址下载zip包 https://github.com/HardySimpson/zlog/zipball/master
解压后，执行命令来生成configure

```
$ cd zlog
$ ./autogen.sh
```

后面的步骤就和本节一开始描述的一样了。

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

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 (!c) {
    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
[formats]
simple = "%m%n"
[rules]
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6.5。

Chapter 4

Syslog 模型

4.1 分类(Category)、规则(Rule)和格式(Format)

zlog有3个重要的概念：分类(Category)、规则(Rule)和格式(Format)。

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。

所以，当程序执行下面的语句的时候

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

zlog会找到c的名字是"my_cat"，对应的配置文件中的规则是

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

然后库会检查，目前这条日志的级别是否符合规则中的级别来决定是否输出。因为INFO>=DEBUG，所以这条日志会被输出。并且根据这条规则，会被输出到stdout（标准输出），输出的格式是simple，在配置文件中定义是

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

最后在屏幕上打印

```
hello, zlog
```

这就是整个过程。用户要做就是写自己的信息。日志往哪里输出，以什么格式输出，都是库和配置文件来完成的。

4.2 syslog模型和log4j模型的区别

好，那么目前这个模型和syslog有什么关系呢？至今为止，这个模型还是比较像log4j。log4j的模型里面有logger, appender和layout。区别在于，在log4j里面，代码中的logger和配置中的logger是一一对应的，并且一个logger有唯一的级别。一对一关系是log4j, log4cxx, log4cpp, log4cplus, log4net的唯一选择。

但这种模型是不灵活的，他们发明了过滤器（filters）来弥补，但这只能把事情弄得更加混乱。所以让我们把目光转回syslog的模型，这是一个设计的很简易正确的模型。

继续上一节的例子，如果在zlog的配置文件中有这么2行规则：

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

然后，一行代码会产生两行输出：

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

现在一个代码中的分类对应配置文件中的两条规则。log4j的用户可能会说："这很好，但是只要在log4j里面放两个appender也能做的一样。"所以继续看下一个例子：

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

代码是：

```
ZLOG_INFO(c, "info, zlog");
ZLOG_DEBUG(c, "debug, zlog");
```

最后，在aa.log中只有一条日志

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

但在bb.log里面有两条

```
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
```

从这个例子能看出来区别。log4j无法轻易的做到这一点。在zlog里面，一个分类可以对应多个规则，每个规则有自己的级别、输出和格式。这就让用户能按照需求过滤、多渠道输出自己的所有日志。

4.3 扩展syslog模型

所以到现在你能看出来zlog的模型更像syslog的模型。不幸的是，在syslog里面，设施（facility）是个int型，而且必须从系统定义的那几种里面选择。zlog走的远一点，用一个字符串来标识分类。

syslog有一个通配符"*"，匹配所有的设施（facility）。zlog里面也一样，"*"匹配所有分类。这提供了一个很方便的办法来重定向你的系统中各个组件的错误。只要这么写：

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

zlog强大而独有的特性是上下级分类匹配。如果你的分类是这样的：

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

然后配置文件是这样的

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

这两条规则都匹配c分类"my_cat"。通配符"_" 表示上级分类。 "my_"是"my_cat"和"my_dog"的上级分类。还有一个通配符是"!"，详见5.5.2

Chapter 5

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。这是个配置文件例子：

```
# comments
[global]
strict init = true
buffer min = 1024
buffer max = 2MB
rotate lock file = /tmp/zlog.lock
default format = "%D.%us %-6P (%c:%F:%L) - %m%n"
file perms = 600

[levels]
TRACE = 10
CRIT = 130, LOG_CRIT

[formats]
simple = "%m%n"
normal = "%D %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;
```

有关单位：当设置内存大小或者大数字时，可以设置1k 5GB 4M这样的单位：

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

单位是大小写不敏感的，所以1GB 1Gb 1gB是等效的。

5.1 全局参数

全局参数以[global]开头。[]代表一个节的开始，四个小节的顺序不能变，依次为global-levels-formats-rules。这一节可以忽略不写。语法为

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

- strict init

  如果"strict init"是true，zlog_init()将会严格检查所有的格式和规则，任何错误都会导致zlog_init() 失败并且返回-1。当"strict init"是false的时候，zlog_init()会忽略错误的格式和规则。 这个参数默认为true。

- reload conf period

  这个选项让zlog能在一段时间间隔后自动重载配置文件。重载的间隔以每进程写日志的次数来定义。当写日志次数到了一定值后，内部将会调用zlog_reload()进行重载。每次zlog_reload()或者zlog_init()之后重新计数累加。因为zlog_reload()是原子性的，重载失败继续用当前的配置信息，所以自动重载是安全的。默认值是0，自动重载是关闭的。

- buffer min

- buffer max

  zlog在堆上为每个线程申请缓存。"buffer min"是单个缓存的最小值，zlog_init()的时候申请这个长度的内存。写日志的时候，如果单条日志长度大于缓存，缓存会自动扩充，直到到"buffer max"。 单条日志再长超过"buffer max"就会被截断。如果 "buffer max" 是0，意味着不限制缓存，每次扩充为原先的2倍，直到这个进程用完所有内存为止。缓存大小可以加上 KB, MB 或 GB这些单位。默认来说"buffer min"是 1K ， "buffer max" 是2MB。

- rotate lock file

  这个选项指定了一个锁文件，用来保证多进程情况下日志安全转档。zlog会在zlog_init()时候以读写权限打开这个文件。确认你执行程序的用户有权限创建和读写这个文件。转档日志的伪代码是：

```
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用于多线程， fcntl_lock用于多进程。fcntl_lock是POSIX建议锁。详见man 3 fcntl。这个锁是全系统有效的。在某个进程意外死亡后，操作系统会释放此进程持有的锁。这就是我为什么用fcntl锁来保证安全转档。进程需要对锁文件有读写权限。

默认来说，rotate lock file = self。在这种情况下，zlog不会创建任何锁文件，用配置文件作为锁文件。fcntl是建议锁，所以用户可以自由的修改存储他们的配置文件。一般来说，单个日志文件不会被不同操作系统用户的进程转档，所以用配置文件作为锁文件是安全的。

如果你设置其他路径作为锁文件，例如/tmp/zlog.lock，zlog会在zlog_init()的时候创建这个文件。如果有多个操作系统用户的进程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。默认值是/tmp/zlog.lock。

- default format

  这个参数是缺省的日志格式，默认值为：

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

  这种格式产生的输出类似这样：

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

- file perms

  这个指定了创建日志文件的缺省访问权限。必须注意的是最后的产生的日志文件的权限为"file perms"& ~umask。默认为600，只允许当前用户读写。

- fsync period

  在每条规则写了一定次数的日志到文件后，zlog会调用fsync(3)来让操作系统马上把数据写到硬盘。次数是每条规则单独统计的，并且在zlog_reload()后会被清0。必须指出的是，在日志文件名是动态生成或者被转档的情况下，zlog不能保证把所有文件都搞定，zlog只fsync()那个时候刚刚write()的文件描述符。这提供了写日志速度和数据安全性之间的平衡。例子：

```
$ 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
```

如果你极度在乎安全而不是速度的话，用同步IO文件，见5.5.3。默认值是0，由操作系统来决定什么时候刷缓存到文件。

5.2 日志等级自定义

这一节以[levels]开始。用于定义用户自己的日志等级，建议和用户自定义的日志记录宏一起使用。这一节可以忽略不写。语法为：

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

(level int)必须在[1,253]这个范围内，越大越重要。(syslog level)是可选的，如果不设默认为LOG_DEBUG。

详见7.3。

5.3 格式(Formats)

这一节以[formats]开始。用来定义日志的格式。语法为：

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

很好理解，(name)被后面的规则使用。(name)必须由数字和字母组成，下划线"_"也算字母。(actual format)前后需要有双引号。 (actual formats)可以由转换字符组成，见下一节。

5.4 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。

5.4.1 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。详见5.4.3	%d(%F) 2011-12-01 %d(%m-%d %T) 12-01 17:17:42 %d(%T) 17:17:42.035 %d 2012-02-14 17:03:12
%D	相当于%d(%F %T)，zlog默认时间串，在性能上比%d有一定的提升	2012-02-14 17:03:12
%ms	毫秒，3位数字字符串 取自gettimeofday(2)	013
%us	微秒，6位数字字符串 取自gettimeofday(2)	002323

%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$f来去掉目录只保留文件名，或者编译器有选项可以调节	test_hello.c 或者在某些编译器下 /home/zlog/src/test/test_hello.c
%f	源代码文件名，输出$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7.1	%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
%U	调用函数名，来自于__func__(C99)或者__FUNCTION__(gcc)，如果编译器支持的话。	main
%V	日志级别，大写	INFO
%v	日志级别，小写	info
%t	16进制表示的线程ID，来源于pthread_self() "0x%x",(unsigned int) pthread_t	0xba01e700
%T	相当于%t,不过是以长整型表示的 "%lu", (unsigned long) pthread_t	140633234859776
%%	一个百分号	%
%[其他字符]	解析为错误，zlog_init()将会失败	

5.4.2 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。

5.4.3 时间字符

这里是转换字符d支持的时间字符。

所有字符都是由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 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) 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	%

5.5 规则(Rules)

这一节以[rules]开头。这个描述了日志是怎么被过滤、格式化以及被输出的。这节可以忽略不写，不过这样就没有日志输出了，嘿嘿。语法是：

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

当zlog_init()被调用的时候，所有规则都会被读到内存中。当zlog_get_category()被调用，规则就被被分配给分类（5.5.2）。在实际写日志的时候，例如ZLOG_INFO()被调用的时候，就会比较这个INFO和各条规则的等级，来决定这条日志会不会通过这条规则输出。当zlog_reload()被调用的时候，配置文件会被重新读入，包括所有的规则，并且重新计算分类对应的规则。

5.5.1 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7.3。

5.5.2 分类匹配

分类必须由数字和字母组成，下划线"_"也算字母。

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_结尾的分类匹配本级及下级分类	aa_.*	aa, aa_bb, aa_cc, aa_bb_cc	xx, yy
不以_结尾的精确匹配分类名	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)

5.5.3 输出动作

目前zlog支持若干种输出，语法是：
[输出], [附加选项, 可选]; [format(格式)名, 可选]

动作	输出字段	附加选项
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...
同步IO文件	-"文件路径"	
用户自定义输出	$name	"path" 动态或者静态的用于record输出

- stdout, stderr, syslog

  如表格描述，其中只有sylog的附加选项是有意义并必须写的。

  值得注意的是，zlog在写日志的时候会用这样的语句

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

  而如果你的程序是个守护进程，在启动的时候把STDOUT_FILENO，也就是1的文件描述符关掉的话，会发生什么结果呢？

日志会被写到新的1的文件描述符所代表的文件里面！我收到过邮件，说zlog把日志写到自己的配置文件里面去了！

所以，千万不要在守护进程的规则里面加上>stdout或>stderr。这会产生不可预料的结果……

- 文件
  - 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 5.4。

    zlog的文件功能极为强大，例如

    1. 输出到命名管道(FIFO)，必须在调用前由mkfifo(1)创建

       ```
       *.*    "/tmp/pipefile"
       ```

    2. 输出到NULL，也就是不输出

       ```
       *.*    "/dev/null"
       ```

    3. 每线程一个日志，在程序运行的目录下

       ```
       *.*    "%T.log"
       ```

    4. 输出到有进程号区分的日志，每天，在$HOME/log目录，每1GB转档一次，保持5个日志文件。

       ```
       *.*    "%E(HOME)/log/aa.%p.%d(%F).log",1GB * 5
       ```

    5. aa_及下级分类，每个分类一个日志

       ```
       aa_.*      "/var/log/%c.log"
       ```

  - 文件转档

    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。如果不写*3,则意味这一直转档下去，不会删除任何旧的日志。

- 同步IO文件

  在文件路径前加上一个"-"就打开了同步IO选项。在打开文件(open)的时候，会以O_SYNC选项打开，这时候每次写日志操作都会等操作系统把数据写到硬盘后才返回。这个选项极为耗时：

```
$ 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
```

- 格式名

  是可选的，如果不写，用全局配置里面的默认格式：

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

- 用户自定义输出详见7.4

5.6 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 is nul
```

```
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] fail
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
```

Chapter 6

zlog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"
```

6.1 初始化和清理

总览

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

描述

zlog_init()从配置文件confpath中读取配置信息到内存。如果confpath为NULL，会寻找环境变量ZLOG_CONF_PATH的值作为配置文件名。如果环境变量ZLOG_CONF_PATH也没有，所有日志以内置格式写到标准输出上。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并不做任何事情。

zlog_reload()从confpath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、并设置原有的用户自定义输出函数。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confpath为NULL，会重载上一次zlog_init()或者zlog_reload()使用的配置文件。如果zlog_reload()失败，上一次的配置依然有效。所以zlog_reload()具有原子性。

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

返回值

如果成功，zlog_init()和zlog_reload()返回0。失败的话，zlog_init()和zlog_reload()返回-1。详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.2 分类(Category)操作

总览

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

描述

zlog_get_category()从zlog的全局分类表里面找到分类，用于以后输出日志。如果没有的话，就建一个。然后它会遍历所有的规则，寻找和cname匹配的规则并绑定。

配置文件规则中的分类名匹配cname的规律描述如下：

1. * 匹配任意cname。
2. 以下划线_结尾的分类名同时匹配本级分类和下级分类。例如aa_匹配aa, aa_, aa_bb, aa_bb_cc这几个cname。
3. 不以下划线_结尾的分类名精确匹配cname。例如aa_bb匹配aa_bb这个cname。
4. ! 匹配目前还没有规则的cname。

每个zlog_category_t *对应的规则，在zlog_reload()的时候会被自动重新计算。不用担心内存释放，zlog_fini() 最后会清理一切。

返回值

如果成功，返回zlog_category_t的指针。如果失败，返回NULL。详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.3 写日志函数及宏

总览

```
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);
```

描述

这3个函数是实际写日志的函数，输入的数据对应于配置文件中的%m。category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      0123456789A
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 - te
0000000003   72 61 72 79 20 77 72 61 70 70 65 72 20 73 63 72   rary wrapper
```

参数file和line填写为__FILE__和__LINE__这两个宏。这两个宏标识日志是在哪里发生的。参数func 填写为__func__或者__FUNCTION__，如果编译器支持的话，如果不支持，就填写为"<unkown>"。

level是一个整数，应该是在下面几个里面取值。

```
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;
```

每个函数都有对应的宏，简单使用。例如：

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

所有的宏列表:

```
/* 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)
```

返回值

这些函数不返回。如果有错误发生，详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.4 MDC操作

总览

```
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);
```

描述

MDC(Mapped Diagnostic Context)是一个每线程拥有的键-值表，所以和分类没什么关系。

key和value是字符串，长度不能超过MAXLEN_PATH(1024)。如果超过MAXLEN_PATH(1024)的话，会被截断。

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调用不会影响其他线程。

返回值

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。如果有错误发生，详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.5 dzlog接口

总览

```
int dzlog_init(char *confpath, char *cname);
int dzlog_set_category(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);
```

描述

dzlog是忽略分类(zlog_category_t)的一组简单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，这样会更爽。

dzlog_init()和zlog_init()一样做初始化，就是多需要一个默认分类名cname的参数。zlog_reload()、 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。

dzlog的宏也定义在zlog.h里面。更简单的写法。

```
DZLOG_FATAL(format, ...)
DZLOG_ERROR(format, ...)
DZLOG_WARN(format, ...)
DZLOG_NOTICE(format, ...)
DZLOG_INFO(format, ...)
DZLOG_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_NOTICE(buf, buf_len)
HDZLOG_INFO(buf, buf_len)
HDZLOG_DEBUG(buf, buf_len)
```

返回值

成功情况下dzlog_init()和dzlog_set_category()返回0。失败情况下dzlog_init()和 dzlog_set_category()返回-1。详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.6 用户自定义输出

总览

```
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(char *name, zlog_record_fn record);
```

描述

zlog允许用户自定义输出函数。输出函数需要绑定到某条特殊的规则上。这种规则的例子是：

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

zlog_set_record()做绑定动作。规则中输出段有$name的，会被用来做用户自定义输出。输出函数为record。这个函数需要为zlog_record_fn的格式。

zlog_msg_t结构的各个成员描述如下：

path来自规则的逗号后的字符串，这个字符串会被动态的解析，输出当前的path，就像动态文件路径一样。

buf和len 是zlog格式化后的日志信息和长度。

所有zlog_set_record()做的绑定在zlog_reload()使用后继续有效。

返回值

成功情况下zlog_set_record()返回0。失败情况下zlog_set_record()返回-1。详细错误会被写在由环境变量ZLOG_PROFILE_ERROR指定的错误日志里面。

6.7 调试和诊断

总览

```
void zlog_profile(void);
```

描述

环境变量ZLOG_PROFILE_ERROR指定zlog本身的错误日志。

环境变量ZLOG_PROFILE_DEBUG指定zlog本身的调试日志。

zlog_profile()打印所有内存中的配置信息到ZLOG_PROFILE_ERROR，在运行时。可以把这个和配置文件比较，看看有没有问题。

Chapter 7

高阶使用

7.1 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
[formats]
mdc_format=     "%d(%F %X.%ms) %-6V (%c:%F:%L) [%M(myname)] - %m%n"
[rules]
*.*             >stdout; mdc_format
```

输出

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

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。

7.2 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[Mar 13 2
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，这是为了效率。

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 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], sr
```

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

然后zlog.error.log会是

```
$ 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[2012-06-01 11:2
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) %V [%
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/zlog.lock
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 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
[global]
default format  =                "%V %v %m%n"
[levels]
TRACE = 30, LOG_DEBUG
[rules]
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)

```
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
```

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__, sizeof(__FILE__)-1, \
__func__, sizeof(__func__)-1,__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, ...) \
        zlog(cat, __FILE__, sizeof(__FILE__)-1, \
        __func__, sizeof(__func__)-1, __LINE__, \
        ZLOG_LEVEL_TRACE, format, ## __VA_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。

7.4 用户自定义输出

这里我把用户自定义输出的几个步骤写下来。

1. 在配置文件里面定义规则

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

2. 绑定一个函数到$myoutput，并使用之

```
#include <stdio.h>
#include "zlog.h"
int output(zlog_msg_t *msg)
{
    printf("[mystd]:[%s][%s][%ld]\n", str2, msg, (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. 最后我们发现用户自定义输出的函数能用了！

```
$ ./test_record
[mystd]:[ mypath my_cat 2012-07-19 12:23:08][hello, zlog
][12]
```

正如你所见，msglen是12，zlog生成的msg在最后有一个换行符。

4. 用户自定义输出可以干很多神奇的事情，就像某个用户(flw@newsmth.net)的需求

   (a) 日志文件名为 foo.log
   (b) 如果 foo.log 超过 100M，则生成一个新文件，其中包含的就是 foo.log 目前的内容而 foo.log 则变为空，重新开始增长
   (c) 如果距离上次生成文件时已经超过 5 分钟，则即使不到 100M，也应当生成一个新文件
   (d) 新文件名称可以自定义，比如加上设备名作为前缀、日期时间串作为后缀
   (e) 我希望这个新文件可以被压缩，以节省磁盘空间或者网络带宽。

   但愿他能顺利写出这个需求的代码！在多进程或者多线程的情况下！上帝保佑他！

Chapter 8

尾声

好酒！所有人生问题的终极起源和终极答案。

荷马.辛普森