<https://blog.csdn.net/luotuo44/article/details/38317797>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38317797**](http://blog.csdn.net/luotuo44/article/details/38317797)

# 日志处理：

        在Libevent的源码中，经常会见到形如event\_warn、event\_msgx、event\_err之类的函数。这通常出现在代码中一些值是不合理时。这些函数就是Libevent的日志函数。它能把这些不合理的情况打印出来，告知用户。

## 定制日志回调函数：

        Libevent在默认情况下，会将这些日志信息输出到终端上。这当然就不利于日后的观察。为此，Libevent允许用户定制自己的日志回调函数。所有的日志函数在最后输出信息时，都会调用日志回调函数的。所以用户可以通过定制自己的日志回调函数（在回调函数中把信息输出到一个文件上），方便日后的查看。定制回调函数就像设置自己信号处理函数那样，设置一个日志回调函数。当有日志时，Libevent库就会调用这个日志回调函数。

        回调函数的格式和日志定制函数如下所示：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **typedef** **void** (\*event\_log\_cb)(**int** severity, **const** **char** \*msg);
2. **void** event\_set\_log\_callback(event\_log\_cb cb);

        回调函数中的第一个参数severity是日志级别类型，有下面这些：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. #define EVENT\_LOG\_DEBUG 0
2. #define EVENT\_LOG\_MSG   1
3. #define EVENT\_LOG\_WARN  2
4. #define EVENT\_LOG\_ERR   3
6. /\* Obsolete names: these are deprecated, but older programs might use them.
7. \* They violate the reserved-identifier namespace. \*/
8. #define \_EVENT\_LOG\_DEBUG EVENT\_LOG\_DEBUG
9. #define \_EVENT\_LOG\_MSG EVENT\_LOG\_MSG
10. #define \_EVENT\_LOG\_WARN EVENT\_LOG\_WARN
11. #define \_EVENT\_LOG\_ERR EVENT\_LOG\_ERR

        值得注意的是，不能在你的日志回调函数里面调用任何Libevent提供的API函数，否则将发生未定义行为。

        在实现上，Libevent是通过定义一个全局函数指针变量来保存用户在日志定制函数中传入的参数cb。日志定制函数在实现上也是很简单的

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **static** event\_log\_cb log\_fn = NULL;
2. **void**
3. event\_set\_log\_callback(event\_log\_cb cb)
4. {
5. log\_fn = cb;
6. }

        从event\_set\_log\_callback的实现代码可以看到，并没有对这个参数cb做任何检查。

        Libevent的默认日志处理函数event\_log函数还是很简陋的，只是简单地根据参数判断日志记录的级别，然后把级别和日志内容输出。复杂一点的日志功能，可以参考[muduo日志](http://blog.csdn.net/luotuo44/article/details/19252535)功能。当然也有log4pp、log4xx这些把一个重量级的日志库。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **static** **void**
2. event\_log(**int** severity, **const** **char** \*msg)
3. {
4. **if** (log\_fn)
5. log\_fn(severity, msg);//调用用户的日志回调函数
6. **else** {
7. **const** **char** \*severity\_str;
8. **switch** (severity) {
9. **case** \_EVENT\_LOG\_DEBUG:
10. severity\_str = "debug";
11. **break**;
12. **case** \_EVENT\_LOG\_MSG:
13. severity\_str = "msg";
14. **break**;
15. **case** \_EVENT\_LOG\_WARN:
16. severity\_str = "warn";
17. **break**;
18. **case** \_EVENT\_LOG\_ERR:
19. severity\_str = "err";
20. **break**;
21. **default**:
22. severity\_str = "???";
23. **break**;
24. }
25. (**void**)fprintf(stderr, "[%s] %s\n", severity\_str, msg);//输出到标准错误终端上
26. }
27. }

        从event\_log函数中可以看到，当函数指针log\_fn不用NULL时，就调用log\_fn指向的函数。否则就直接向stderr输出日志信息。所以，设置自己的日志回调函数后，如果想恢复Libevent默认的日志回调函数，只需再次调用event\_set\_log\_callback函数，参数设置为NULL即可。

## 日志API以及日志消息处理流程：

        Libevent的日志API的使用也是挺简单的。首先，使用者确定要记录的日志的级别和错误类型，然后调用对应的日志函数。有下面这些可供选择的日志函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **void** event\_err(**int** eval, **const** **char** \*fmt, ...);
2. **void** event\_warn(**const** **char** \*fmt, ...);
3. **void** event\_sock\_err(**int** eval, evutil\_socket\_t sock, **const** **char** \*fmt, ...);
4. **void** event\_sock\_warn(evutil\_socket\_t sock, **const** **char** \*fmt, ...);
6. **void** event\_errx(**int** eval, **const** **char** \*fmt, ...);
7. **void** event\_warnx(**const** **char** \*fmt, ...);
8. **void** event\_msgx(**const** **char** \*fmt, ...);
9. **void** \_event\_debugx(**const** **char** \*fmt, ...);

        这些函数都是声明在log-internal.h文件中。所以用户并不能使用之，这些函数都是Libevent内部使用的。

        这些函数内部实现都差不多。下面是其中几个实现

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **void**
2. event\_warn(**const** **char** \*fmt, ...)
3. {
4. **va\_list** ap;
6. va\_start(ap, fmt);
7. \_warn\_helper(\_EVENT\_LOG\_WARN, strerror(errno), fmt, ap);
8. va\_end(ap);
9. }
11. **void**
12. event\_sock\_err(**int** eval, evutil\_socket\_t sock, **const** **char** \*fmt, ...)
13. {
14. **va\_list** ap;
15. **int** err = evutil\_socket\_geterror(sock);
17. va\_start(ap, fmt);
18. \_warn\_helper(\_EVENT\_LOG\_ERR, evutil\_socket\_error\_to\_string(err), fmt, ap);
19. va\_end(ap);
20. event\_exit(eval);
21. }

        可以看到，主要是设置调用的日记级别，把可变参数用va\_list变量记录，然后调用\_warn\_helper这个辅助函数。其中，event\_warn和event\_warnx的区别是\_warn\_helper函数的第二个参数分别是strerror(errno)和NULL。

        而\_warn\_helper这个辅助函数也是蛮简单的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **static** **void**
2. \_warn\_helper(**int** severity, **const** **char** \*errstr, **const** **char** \*fmt, **va\_list** ap)
3. {
4. **char** buf[1024];
5. **size\_t** len;
7. //如果有可变参数，就把可变参数格式化到一个缓存区buf中。
8. **if** (fmt != NULL)
9. evutil\_vsnprintf(buf, **sizeof**(buf), fmt, ap);
10. **else**
11. buf[0] = '\0';
13. //如果有额外的信息描述，把这些信息追加到可变参数的后面。
14. **if** (errstr) {
15. len = strlen(buf);
16. //-3是因为还有另外三个字符，冒号、空格和\0。
17. **if** (len < **sizeof**(buf) - 3) {
18. evutil\_snprintf(buf + len, **sizeof**(buf) - len, ": %s", errstr);
19. }
20. }
22. //把缓存区的数据作为一条日志记录，调用Libevent的日志函数。
23. event\_log(severity, buf);//这个函数也是最前面的说到的那个日志处理函数
24. }

        这些日志函数的使用，如同printf函数那样，支持%s, %d之类的格式化输出。比如：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. event\_warnx("Far too many %s (%d)", "wombats", 99);

        值得注意的是，也如同printf那样，可变参数中的参数要和第一个参数中的格式要求相匹配。在这些日志函数的声明中，如果是GNU 的编译器，将会检查是否匹配。其声明如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. #ifdef \_\_GNUC\_\_
2. #define EV\_CHECK\_FMT(a,b) \_\_attribute\_\_((format(printf, a, b)))
3. #define EV\_NORETURN \_\_attribute\_\_((noreturn))
4. #else
5. #define EV\_CHECK\_FMT(a,b)
6. #define EV\_NORETURN
7. #endif
9. #define \_EVENT\_ERR\_ABORT ((int)0xdeaddead)
11. **void** event\_err(**int** eval, **const** **char** \*fmt, ...) EV\_CHECK\_FMT(2,3) EV\_NORETURN;
12. **void** event\_warn(**const** **char** \*fmt, ...) EV\_CHECK\_FMT(1,2);
13. **void** event\_sock\_err(**int** eval, evutil\_socket\_t sock, **const** **char** \*fmt, ...) EV\_CHECK\_FMT(3,4) EV\_NORETURN;
14. **void** event\_sock\_warn(evutil\_socket\_t sock, **const** **char** \*fmt, ...) EV\_CHECK\_FMT(2,3);
15. **void** event\_errx(**int** eval, **const** **char** \*fmt, ...) EV\_CHECK\_FMT(2,3) EV\_NORETURN;
16. **void** event\_warnx(**const** **char** \*fmt, ...) EV\_CHECK\_FMT(1,2);
17. **void** event\_msgx(**const** **char** \*fmt, ...) EV\_CHECK\_FMT(1,2);
18. **void** \_event\_debugx(**const** **char** \*fmt, ...) EV\_CHECK\_FMT(1,2);

# 错误处理：

        Libevent库运行的时候有可能会致命发生错误，此时，Libevent的默认行为是终止程序。同日志处理一样，用户也是可以用Libevent定制自己的错误处理函数。错误处理函数的格式和定制函数如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38317797) [copy](https://blog.csdn.net/luotuo44/article/details/38317797)

1. **typedef** **void** (\*event\_fatal\_cb)(**int** err);
3. **void** event\_set\_fatal\_callback(event\_fatal\_cb cb);

        定制函数的内部实现原理和前面的日志处理函数一样，都是通过一个全局函数指针变量存储用户的错误处理函数。

        在默认情况下，Libevent处理这些致命错误时会粗暴地杀死程序，但大多数情况下，Libevent在调用这个致命处理函数前都会调用前面的日志记录函数，其级别是\_EVENT\_LOG\_ERR。此时，虽然程序突然死了，但还是可以在日志中找到一些信息。用户的错误处理函数也应该杀死程序。

        如果要定制自己的日志处理函数和错误处理函数，那么应该在程序的一开始位置就进行定制。

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luotuo44/article/details/38317797