



# Libev 官方文档学习笔记 - 01: 概述和 ev\_loop

[linux](#) [c](#) [异步io](#) [libev](#) [bsd](#) 发布于 2016-08-04

请注意这是 **libev** 而不是 libevent 的文章！

自从接触到 libev 之后，就深深赞同作者精简的设计理念，于是就爱上了 libev 这样简单的I/O库。此外，libev 的大小也比 libevent 小得多并且自由得多。虽然我在公司的项目用的异步 I/O 库还是以 libevent 和 libubox 为主，但是个人业余的工程中，往往用的是 libev 而不是 libevent。

可惜的是，貌似是因为 libev 是单人维护，而且不支持 Windows 等原因，并不如 libevent 甚至是 libuv 等受欢迎，国内的研究资料也并不多。

但是呢，老子是 Linux / BSD 开发者，我就喜欢！

阅读本文最好有 libevent 基础，因为基本概念和 libevent 是类似的

库	事件循环	具体事件
libevent	event_loop	event
libev	ev_loop	watcher

其他的在这里我就不重复了。

关于 libevent 请参见我的文章：[Libevent官方文档学习笔记](#)

本文地址：<https://segmentfault.com/a/1190000006173864>

## Reference

- [libev官网](#)
- [libev - a high performance full featured event loop written in c](#)
- [\[Linux 网络编程\] Libev 初步](#)

## 概述

### Features

- ev\_io：支持 Linux 的select、poll、epoll；BSD 的kqueue；Solaris 的event port mechanisms
- ev\_signal：支持各种信号处理、同步信号处理
- ev\_timer：相对事件处理
- ev\_periodic：排程时间表
- ev\_child：进程状态变化事件
- ev\_start：监视文件状态
- ev\_fork：有限的fork事件支持

## 时间显示

Libev 使用一个`ev_tstamp`数据类型来表示1970年以来的秒数，实际类型是 C 里面的`double`类型。

## 错误事件

Libev 使用三种层级的错误：

- 1. **操作系统错误**：调用`ev_set_syserr_cb`所设置的回调。默认行为是调用`abort()`
- 2. **参数错误**：调用`assert`
- 3. **内部错误 (bug)**：内部调用`assert`

## 全局（配置）函数

以下函数可以在任意时间调用，用于配置 libev 库：

```
ev_tstamp ev_time ();
```

返回当前的时间。

```
void ev_sleep (ev_tstamp interval);
```

休眠一段指定的时间。如果`interval`小于等于0，则立刻返回。最大支持一天，也就是86400秒

```
int ev_version_major ();
int ev_version_minor ();
```

可以调用这两个函数，并且与系统与定义的`EV_VERSION_MAJOR`和`EV_VERSION_MINOR`作对比，判断是否应该支持该库

```
unsigned int ev_supported_backends ();
unsigned int ev_recommand_backends ();
unsigned int ev_embeddable_backends ();
```

返回该 libev 库支持的和建议的后端列表

```
void ev_set_allocator ( void *(*cb)(void *ptr, long size)throw() );
```

重新设置`realloc`函数。对于一些系统（至少包括 BSD 和 Darwin）的 `realloc` 函数可能不正确，libev 已经给了替代方案。

```
void ev_set_syserr_cb ( void (*cb)(const char *msg)throw() );
```

设置系统错误的 callback。默认调用`perror()`并`abort()`

```
void ev_feed_signal (int signum)
```

模拟一个`signal`事件出来

## 控制 event loops 的函数

Event loop 用一个结构体`struct ev_loop *`描述。Libev 支持两类 loop，一是 default loop，支持 child process event；动态创建的 event loops 就不支持这个功能

```
struct ev_loop *ev_default_loop (unsigned int flags);
```

初始化 default loops。如果已经初始化了，那么直接返回并且忽略 flags。注意这个函数并不是线程安全的。只有这个 loop 可以处理 `ev_child` 事件。

```
struct ev_loop *ev_loop_new (unsigned int flags);
```

这个函数是线程安全的。一般而言，每个 thread 使用一个 loop。以下说明 flag 项的各个值：

- `EVFLAG_AUTO`：默认值，常用
- `EVFLAG_NOENV`：指定 libev 不使用 `LIBEV_FLAGS` 环境变量。常用于调试和测试
- `EVFLAG_FORKCHECK`：与 `ev_loop_fork()` 相关，本文暂略
- `EVFLAG_NOINOTIFY`：在 `ev_stat` 监听中**不使用** `inotify` API
- `EVFLAG_SIGNALFD`：在 `ev_signal` 监听中使用 `signalfd` API
- `EVFLAG_NOSIGMASK`：使 libev 避免修改 signal mask。这样的话，你要使 signal 是非阻塞的。在未来的 libev 中，这个 mask 将会是默认值。
- `EVBACKEND_SELECT`：通用后端
- `EVBACKEND_POLL`：除了 Windows 之外的所有后端都可以用
- `EVBACKEND_EPOLL`：Linux 后端
- `EVBACKEND_KQUEUE`：大多数 BSD 的后端
- `EVBACKEND_DEVPOLL`：Solaris 8 后端
- `EVBACKEND_PORT`：Solaris 10 后端

```
void ev_loop_destroy (struct ev_loop *loop);
```

销毁 `ev_loop`。注意这里要将所有的 IO 清除光之后再调用，因为这个函数并不中止所有活跃（active）的 IO。部分 IO 不会被清除，比如 signal。这些需要手动清除。这个函数一般和 `ev_loop_new` 一起出现在同一个线程中。

```
void ev_loop_fork (struct ev_loop *loop);
```

这个函数导致 `ev_run` 的子过程重设已有的 backend 的 kernel state。重用父进程创建的 loop。可以和 `pthread_atfork()` 配合使用。

需要在每一个需要在 fork 之后重用的 loop 中调用这个函数。必须在恢复之前或者调用 `ev_run()` 之前调用。如果是在 `fork` 之后创建的 loop，不需要调用。

使用 pthread 的代码例如下：

```
static void post_fork_child (void)
{
    ev_loop_fork (EV_DEFAULT);
}
...
pthread_atfork (NULL, NULL, post_fork_child);
```

```
int ev_is_default_loop (struct ev_loop *loop);
```

判断当前 loop 是不是 default loop。

```
unsigned int ev_iteration (struct ev_loop *loop);
```

返回当前的 loop 的迭代数。等于 libev pool 新事件的数量 (?)。这个值对应`ev_prepare`和`ev_check`调用，并在 prepare 和 check 之间增一。

```
unsigned int ev_depth (struct ev_loop *loop);
```

返回`ev_run()`进入减去退出次数的差值。

注意，导致`ev_run`异常退出的调用（setjmp / longjmp, pthread\_cancel, 抛出异常等）均不会导致该值减一。

```
unsigned int ev_backend (struct ev_loop *loop);
```

返回`EVBKEND_*`值

```
ev_tstamp ev_now (loop)
```

得到当前的“event loop time”。在 callback 调用期间，这个值是不变的。

```
void ev_new_update (loop)
```

更新从`ev_now()`中返回的时间。不必要的话，不要使用，因为这个函数的开销相对是比较大的。

```
void ev_suspend (struct ev_loop *loop);  
void ev_resume (struct ev_loop *loop);
```

暂停当前的 loop，使其刮起当前的所有工作。同时其 timeout 也会暂停。如果恢复后，timer 会从上一次暂停状态继续及时——这一点对于实现一些要连同时间也一起冻结的功能时，非常有用。

注意已经 resume 的loop不能再 resume，反之已经 suspend 的 loop 不能再 suspend。

```
bool ev_run (struct ev_loop *loop, int flags);
```

初始化 loop 结束后，调用这个函数开始 loop。如果 flags == 0，直至 loop 没有活跃的时间或者是调用了 ev\_bread 之后停止。

Loop 可以是异常使能的，你可以在 callback 中调用`longjmp`来终端回调并且跳出 ev\_run，或者通过抛出 C++ 异常。这些不会导致 ev\_depth 值减少。

`EVRUN_NOWAIT`会检查并且执行所有未解决的 events，但如果没有就绪的时间，ev\_run 会立刻返回。`EVRUN_ONCE`会检查所有的 events，在至少每一个 event 都执行了一次事件迭代之后才返回。但有时候，使用`ev_prepare/ev_check`更好。

以下是`ev_run`的大致工作流程：

- loop depth ++
- 重设`ev_break`状态
- 在首次迭代之前，调用所有 pending watchers

LOOP：

- 如果置了`EVFLAG_FORKCHECK`，则检查 fork，如果检测到 fork，则排队并调用所有的 fork watchers
- 排队并且调用所有 ready 的watchers
- 如果`ev_break`被调用了，则直接跳转至 FINISH
- 如果检测到了 fork，则分离并且重建 kernel state
- 使用所有未解决的变化更新 kernel state
- 更新`ev_now`的值
- 计算要 sleep 或 block 多久

- 如果指定了的话, sleep
- loop iteration ++
- 阻塞以等待事件
- 排队所有未处理的I/O事件
- 更新ev\_now的值, 执行 time jump 调整
- 排队所有超时事件
- 排队所有定期事件
- 排队所有优先级高于 pending 事件的 idle watchers
- 排队所有 check watchers
- 按照上述顺序的逆序, 调用 watchers (check watchers -> idle watchers -> 定期事件 -> 计时器超时事件 -> fd事件)。信号和 child watchers 视为 fd watchers。
- 如果ev\_break被调用了, 或者使用了EVRUN\_ONCE或者EVRUN\_NOWAIT, 则如果没有活跃的 watchers, 则 FINISH, 否则 continue

FINISH:

- 如果是EVBREAK\_ONE, 则重设 ev\_break 状态
- loop depth --
- return

```
void ev_break (struct ev_loop *loop, how);
```

中断 loop。参数可以是 EVBREAK\_ONE (执行完一个内部调用后返回) 或EVBREAK\_ALL (执行完所有)。

下一次调用 ev\_run 的时候, 相应的标志会清除

```
void ev_ref (struct ev_loop *loop);
void ev_unref (struct ev_loop *loop);
```

类似于 **Objective-C** 中的引用计数, 只要 reference count 不为0, ev\_run 函数就不会返回。

在做 start 之后要 unref; stop 之前要 ref。

```
void ev_set_io_collect_interval (struct ev_loop *loop, ev_tstamp interval);
void ev_set_timeout_collect_interval (struct ev_loop *loop, ev_tstamp interval);
```

两个值均默认为0, 表示尽量以最小的延迟调用 callback。但这是理想的情况, 实际上, 比如 select 这样低效的系统调用, 由于可以一次性读取很多, 所以可以适当地进行延时。通过使用比较高的延迟, 但是增加每次处理的数据量, 以提高 CPU 效率。

```
void ev_invoke_pending (struct ev_loop *loop);
```

调用所有的 pending 的 watchers。这个除了可以在 callback 中调用 (少见) 之外, 更多的是在重载的函数中使用。参见下一个函数

```
void ev_set_invoke_pending_cb (struct ev_loop *loop, void (*invoke_pending_cb)(EV_P));
```

重载 ev\_loop 调用 watchers 的函数。新的回调应调用 ev\_invoke\_pending。如果要恢复默认值, 则置喙 ev\_invoke\_pending 即可。

```
int ev_pending_count (struct ev_loop *loop);
```

返回当前有多少个 pending 的 watchers。

```
void ev_set_loop_release_cb (struct ev_loop *loop,
                             void (*release)(EV_P)throw(),
                             void (*acquire)(EV_P)throw());
```

这是一个 lock 操作，你可以自定义 lock。其中 release 是 unlock，acquire 是 lock。release 是在 loop 挂起以等待events 之前调用，并且在开始回调之前调用 acquire。

```
void ev_set_userdata (struct ev_loop *loop, void *data);
void *ev_userdata (struct ev_loop *loop);
```

设置 / 读取 loop 中的用户 data。这一点和 libevent 很不同，libevent 的参数 / 用户数据是以 event 为单位的，而 libev 的原生用户数据是以 loop 为单位的。

```
void ev_verify (struct ev_loop *loop);
```

验证当前 loop 的设置。如果发现问题，则打印 error msg 并 abort()。

## 系列篇

[Libev 官方文档学习笔记（1）——概述和 ev loop（本文）](#)

[Libev 官方文档学习笔记（2）——watcher 基础](#)

[Libev 官方文档学习笔记（3）——常用 watcher 接口](#)

[使用 libev 构建 TCP 响应服务器的简单流程](#)

阅读 19.6k • 更新于 2017-10-27

👍 赞 5

🔖 收藏 13

¥ 赞赏

🔗 分享

本作品系原创， 采用《署名-非商业性使用-相同方式共享 4.0 国际》许可协议



amc

🔖 734



关注作者

### 11 条评论

得票 • 时间



撰写评论 ...

提交评论



**majianfei1023**： 连 异步 是什么都不懂吧？

👍 • 回复 • 2017-05-05

**golotv**： 你发言只有一堆问句，屁点营养没有，你懂你讲明白，在国内装什么逼呢。

👍 • 回复 • 2017-05-31



**amc**（作者）： 这篇文章不是介绍“异步”的概念的，主要还是在讲 libev。如果你觉得不懂的话，可以移步参考资料。当然，如果你有什么疑问，我能够解答的话，我也会试试看回答的

👍 • 回复 • 2017-05-05



**majianfei1023**： 谢谢你啊，不过不用了。我只是对你的这句“虽然我在公司的项目用的异步 I/O 库还是以 libevent 和 libubox 为主”，有疑问而已。请问libevent是异步IO库吗？

👍 • 回复 • 2017-05-08

**amc**： 真要抠严格定义的话，应该说“异步 I/O”是一种编程思路，这相对于同步的 I/O 编程思路而言。科班出身的软件专业往往会简单



学到 socket, bind, connect, accept, read, write 等等一路下来的 API。正好，如果真的是按顺序这样调用下来，然后写了一个 server / client 的话，可以算是一种同步 I/O 的编程思路。

 · [回复](#) · 2017-05-08

**amc**：而异步 I/O 则不一样。某种程度上，异步 I/O 是##事件驱动##的。程序不是阻塞着等待某一个事件，也不是主动轮询某个事件。在这种模式下，程序应该是“异步地”等待系统 API 或者是其他的 API 对你发出通知，然后程序在这个通知中做响应的操作。操作完后，程序主动返回，把 CPU 使用权交回

 · [回复](#) · 2017-05-08

**amc**：各 UNIX 和类 UNIX 的发行版都提供了 **select** 系统调用，可以理解为 **select** “可以用于实现异步 I/O”。所以 **select** 也可以算是异步 I/O API。但由于 **select** 在高并发下的性能低下，各 UNIX 和 类UNIX 发行版都另外再实现了可用于实现 异步 I/O 的系统 API，比如 Linux ，就提供了 **epoll**

 · [回复](#) · 2017-05-08

[展开显示更多](#)

## 推荐阅读

### NIO、BIO、AIO 与 PHP 实现

最近看到NIO，AIO，Netty，Promise话题很热，我作为一个phper也想来凑凑热闹，凑着凑着发现周围怎么都是javaer，jser。那么...

[小白要生发](#) · 阅读 7.6k · 57 赞 · 4 评论

### 高性能网络编程(二)：上一个10年，著名的C10K并发连接问题

对于高性能即时通讯技术（或者说互联网编程）比较关注的开发者，对C10K问题（即单机1万个并发连接问题）应该都有所了解。”...

[JackJiang](#) · 阅读 24.2k · 35 赞 · 3 评论

### PHP并发IO编程之路

并发 IO 问题一直是服务器端编程中的技术难题，从最早的同步阻塞直接 Fork 进程，到 Worker 进程池/线程池，到现在的异步IO、...

[leo su](#) · 阅读 1.3k · 28 赞

### 基于汇编的 C/C++ 协程 - 背景知识

近几年来，协程在 C/C++ 服务器中的解决方案开始涌现。本文主要阐述以汇编实现上下文切换的协程方案，并且说明其在异步开...

[amc](#) · 阅读 3.6k · 6 赞

### Libevent 官方文档学习笔记（1. libevent core部分）

初入libevent的人，很可能是第一次接触异步编程。Libevent的编程思想，建议还是多看前人的程序，或者是看libevent本身的文档...

[amc](#) · 阅读 14.4k · 6 赞

### Libev 官方文档学习笔记 - 03：常用 watcher 接口

这个 watcher 负责检测文件描述符（以下简称fd）是否可写入数据或者是读出数据。最好是将fd设置为非阻塞的。注意有时候...

[amc](#) · 阅读 8.3k · 4 赞

### Node.js异步I/O,事件驱动

Node.js以高效,轻量著称,具有非阻塞I/O,事件驱动的特性.非阻塞I/O很浅显的解释就是: 代码以单线程的方式执行,在遇到I/O操作时N...

[Leo](#) · 阅读 2.9k · 3 赞

### node - 非阻塞的异步 IO

每当我们提起 node.js 时总会脱口而出 事件驱动、非阻塞I/O 和 单线程，所以我总结了以下几点对这三项概念的阐述，不一定正确...

[kangkk](#) · 阅读 2.3k · 2 赞

## 后台 / 嵌入式全栈之路

用户专栏

曾经的嵌入式 / 后台开发一枚，现在开始走向架构。本专栏没有高深技术，只讲基础组件、工具，请放心食用

72 人关注

82 篇文章

关注专栏

专栏主页

产品

热门问答

热门专栏

热门课程

最新活动

技术圈

酷工作

移动客户端

课程

Java 开发课程

PHP 开发课程

Python 开发课程

前端开发课程

移动开发课程

资源

每周精选

用户排行榜

徽章

帮助中心

声望与权限

社区服务中心

合作

关于我们

广告投放

职位发布

讲师招募

联系我们

合作伙伴

关注

产品技术日志

社区运营日志

市场运营日志

团队日志

社区访谈

条款

服务条款

隐私政策

下载 App

Copyright © 2011-2020 SegmentFault.



浙ICP备 15005796号-2 浙公网安备 33010602002000号 杭州堆栈科技有限公司版权所有