# 零声学院 Mark 老师 QQ: 2548898954

# 服务端事件组成

- **网络io事件**
  - **linux：epoll、poll、select**
  - **mac：kqueue**
  - **window：iocp**
- **定时事件**
  - **红黑树**
  - **最小堆：二叉树、四叉树**
  - **跳表**
  - **时间轮**
- **信号事件**

# libevent 与 libev 概述：

## 概述

> libevent和libev都是c语言实现的异步事件库；通过注册异步事件，库检测事件触发，从而库根据发生事件的先后顺序，调用相应回调函数进行处理；
>
> 事件包括：网络io事件，定时事件，信号事件；
>
> 事件循环：等待并分发事件；用于管理事件；
>
> libevent 和 libev 主要封装了异步事件库与操作系统的交互；让用户不用关注平台的差异，只需着手事件的具体处理；
>
> libevent 和 libev 对window支持都比较差，因此产生 libuv 库，libuv 基于 libev，但是 window 上封装了 iocp；node.js基于libuv；

## 区别

> 从设计理念出发，libev 是为了改进 libevent 中的一些架构决策，例如，全局变量的使用使得在多线程环境中很难安全地使用 libevent，watcher 的数据结构设计太大，因为它们将 I/O、时间和信号处理放在一个结构体中，额外的组件如 http 、dns、openssl， 服务器由于实现质量差以及由此产生的安全问题，计时器不精确，不能很好地处理时间事件。
>
> libev 通过不使用全局变量，而是对所有回调函数传参的方式传递上下文；并且根据不同事件类型构建不同的数据结构，这样以来减低事件的耦合性；
>
> libev 小而高效；只关注事件处理；

# libevent

## 编译

```
aclocal
libtoolize --force
autoheader
automake --add-missing
autoconf
./configure && make && make install
```

## 特色

### bufferevent

> 提供 bufferevent，进一步提供管理读写事件（包括连接断开事件），以及读写数据缓冲；
>
> - bufferevent_socket_new
> - bufferevent_socket_connect
> - bufferevent_free
> - bufferevent_setcb
> - bufferevent_enable
> - bufferevent_disable
> - bufferevent_get_input
> - bufferevent_get_output
> - bufferevent_write
> - bufferevent_write_buffer
> - bufferevent_read
> - bufferevent_read_buffer

### evconnlistener

> 提供了监听和接受 tcp 连接的方法
>
> - evconnlistener_new
> - evconnlistener_new_bind
> - evconnlistener_free

```
typedef void (*evconnlistener_cb)(struct evconnlistener *, evutil_socket_t,
struct sockaddr *, int socklen, void *);

struct evconnlistener *evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd);

struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);

void evconnlistener_free(struct evconnlistener *lev);

/** Flag: Indicates that we should not make incoming sockets nonblocking
 * before passing them to the callback. */
#define LEV_OPT_LEAVE_SOCKETS_BLOCKING  (1u<<0)
/** Flag: Indicates that freeing the listener should close the underlying
 * socket. */
#define LEV_OPT_CLOSE_ON_FREE       (1u<<1)
```

```
/** Flag: Indicates that we should set the close-on-exec flag, if possible */
#define LEV_OPT_CLOSE_ON_EXEC       (1u<<2)
/** Flag: Indicates that we should disable the timeout (if any) between when
 * this socket is closed and when we can listen again on the same port. */
#define LEV_OPT_REUSEABLE       (1u<<3)
/** Flag: Indicates that the listener should be locked so it's safe to use
 * from multiple threadcs at once. */
#define LEV_OPT_THREADSAFE      (1u<<4)
/** Flag: Indicates that the listener should be created in disabled
 * state. Use evconnlistener_enable() to enable it later. */
#define LEV_OPT_DISABLED        (1u<<5)
/** Flag: Indicates that the listener should defer accept() until data is
 * available, if possible.  Ignored on platforms that do not support this.
 *
 * This option can help performance for protocols where the client transmits
 * immediately after connecting.  Do not use this option if your protocol
 * _doesn't_ start out with the client transmitting data, since in that case
 * this option will sometimes cause the kernel to never tell you about the
 * connection.
 *
 * This option is only supported by evconnlistener_new_bind(): it can't
 * work with evconnlistener_new_fd(), since the listener needs to be told
 * to use the option before it is actually bound.
 */
#define LEV_OPT_DEFERRED_ACCEPT     (1u<<6)
/** Flag: Indicates that we ask to allow multiple servers (processes or
 * threads) to bind to the same port if they each set the option.
 *
 * SO_REUSEPORT is what most people would expect SO_REUSEADDR to be, however
 * SO_REUSEPORT does not imply SO_REUSEADDR.
 *
 * This is only available on Linux and kernel 3.9+
 */
#define LEV_OPT_REUSEABLE_PORT      (1u<<7)
```

# libevent的主要接口

## event_base_new

> 初始化 libevent；对应理解 epoll_create

```
struct event_base *event_base_new(void);
```

## event_new

> 创建事件，初始化event和相应的回调函数

```
struct event *
event_new(struct event_base *base, evutil_socket_t fd, short events, void (*cb)
(evutil_socket_t, short, void *), void *arg)
```

## event_set

> 设置事件

```
void
event_set(struct event *ev, evutil_socket_t fd, short events,
      void (*callback)(evutil_socket_t, short, void *), void *arg)
```

## event_base_set

> 建立 `event` 与 `event_base` 的映射关系；

```
int event_base_set(struct event_base *eb, struct event *ev);
```

## event_add

> 注册事件，包括时间事件；相当于 epoll_ctl；

```
int
event_add(struct event *ev, const struct timeval *tv)
```

## event_del

> 注销事件

```
int
event_del(struct event *ev)
```

## event_base_loop

> 进入事件循环

```
int
event_base_loop(struct event_base *base, int flags)
```

## 注意

> `event_new` 相当于 `malloc + event_set + event_base_set`

# libev的主要数据结构

## EV_WATCHER

```
/* shared by all watchers */
#define EV_WATCHER(type)            \\
  int active;    /* 表示 watcher 是否活跃，active = 1 表示还没被 stop 掉 */ \\
  int pending;   /* 存储 watcher 在 pendings 中的索引。大于零表示还没被处理。
                  * watcher 的回调函数被调用后，会设置为 0。 */ \\
  int priority;  /* 事件的优先级 */ \\
  void *data;    /* 回调函数所需要的数据 */ \\
  void (*cb)(EV_P_ struct type *w, int revents);  /* 回调函数 */
```

作用：不同事件类型的共有信息。

## EV_WATCHER_LIST

```
#define EV_WATCHER_LIST(type)           \\
  EV_WATCHER (type)             \\
  struct ev_watcher_list *next;  /* 同一个文件描述符上可以被注册多个 watcher，比如：监听是否可读/可写 */
作用：watcher 链表
```

## ev_io

```
typedef struct ev_io
{
  EV_WATCHER_LIST (ev_io)

  int fd;
  int events;
} ev_io;
// 作用：记录 IO 事件的基本信息。
// ev_io 相比 ev_watcher 增加了 next, fd, events 的属性。
```

## ANFD

```
/* file descriptor info structure */
typedef struct
{
  WL head;              /* 同一个 fd 上的所有 ev_watcher 事件 */
  unsigned char events; /* the events watched for，通常被设置成所有 ev_watcher->events 的或集。 */
  unsigned char reify;  /* flag set when this ANFD needs reification (EV_ANFD_REIFY, EV__IOFDSET)
                           * 默认值为 0，当调用 ev_io_start 后，reify 会被设置为 `w->events & EV__IOFDSET | EV_ANFD_REIFY`。
                           * 如果 reify 未被设置，则把 fd 添加到 fdchanges 中去。*/
  ...
} ANFD;

// 作用：
// 解决根据 fd 快速找到与其相关的事件。
// libev 的方法是用 anfds 数组来存所有 fd 信息的结构体，然后以 fd 值为索引直接找到对应的结构体。
```

## ANPENDING

```
/* stores the pending event set for a given watcher */
typedef struct
{
  W w;
  int events; /* the pending event set for the given watcher */
} ANPENDING;

// 作用：存储已准备好的 watcher，等待回调函数被调用。
```

## ev_loop

```
struct ev_loop {
  double ev_rt_now; /* 当前的时间戳 */

  int backend; /* 采用哪种多路复用方式, e.g. SELECT/POLL/EPOLL */
  int activecnt; /* total number of active events ("refcount") */
  int loop_done; /* 事件循环结束的标志，signal by ev_break */

  int backend_fd; /* e.g. epoll fd, created by epoll_create*/
  void (*backend_modify)(EV_P_ int fd, int oev, int nev); /* 对应 epoll_ctl */
  void (*backend_poll)(EV_P_ ev_tstamp timeout); /* 对应 epoll_wait */

  void (*invoke_cb)(struct ev_loop *loop);

  ANFD *anfds; /* 把初始化后的 ev_io 结构体绑定在 anfds[fd].head 事件链表上，方便根据 fd
直接查找。*/

  int *fdchanges; /* 存放需要 epoll 监听的 fd */
  ANPENDING *pendings [NUMPRI]; /* 存放等待被调用 callback 的 watcher */
}
// 作用：基本包含了 loop 循环所需的所有信息，为让注释更容易理解采用 epoll 进行说明。
```

# libev的主要接口

## ev_io_init

```
#define ev_io_init(ev,cb,fd,events) do { ev_init ((ev), (cb)); ev_io_set ((ev),
(fd),(events)); } while (0)
```

> 初始化 watcher 的 fd/events/callback

## ev_io_start

```
void ev_io_start(struct ev_loop *loop, ev_io *w)
```

> 注册并绑定 io watcher 到 ev_loop

## ev_timer_start

```
void ev_timer_start(struct ev_loop *loop, ev_timer *w)
```

> 注册并绑定 timer watcher 到 ev_loop

## ev_run

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

> 开启改 ev_loop 的事件循环