零声教育出品 Mark 老师 QQ : 2548898954

概述

> libevent 和 libev 都是由 c 实现的异步事件库；注册异步事件，检测异步事件，根据事件的触发先后顺序，调用相对应回调函数处理事件；
>
> 处理的事件包括：网络 io 事件、定时事件以及信号事件；
>
> libevent 和 libev 主要封装了异步事件库与操作系统的交互；让用户无需关注平台检测处理事件的机制的差异，只需关注事件的具体处理；
>
> libevent 和 libev 对 window 支持比较差，由此产生了 libuv 库；libuv 基于 libev，在window 平台上更好的封装了 iocp；node.js 基于 libuv；
>
> 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；event 的数据结构设计太大，它包含了 io、时间以及信号处理全封装在一个结构体中，额外的组件如 http、dns、openssl 等实现质量差（容易产生安全问题），计时器不精确，不能很好地处理时间事件；
>
> libev 通过完全去除全局变量的使用，而是通过回调传参来传递上下文；并且根据不同事件类型构建不同的数据结构，以此来减低事件耦合性；
>
> libev 小而高效；只关注事件处理；

libevent

编译

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

evconnlistener

```
// 连接到达回调
typedef void (*evconnlistener_cb)(struct evconnlistener *, evutil_socket_t,
struct sockaddr *, int socklen, void *);

// listenfd 错误回调
typedef void (*evconnlistener_errorcb)(struct evconnlistener *, void *);

// 已经有 listenfd 的前提下，使用的接口
struct evconnlistener *evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd);

// 没有 listenfd 需要接口帮助构建
struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);

```

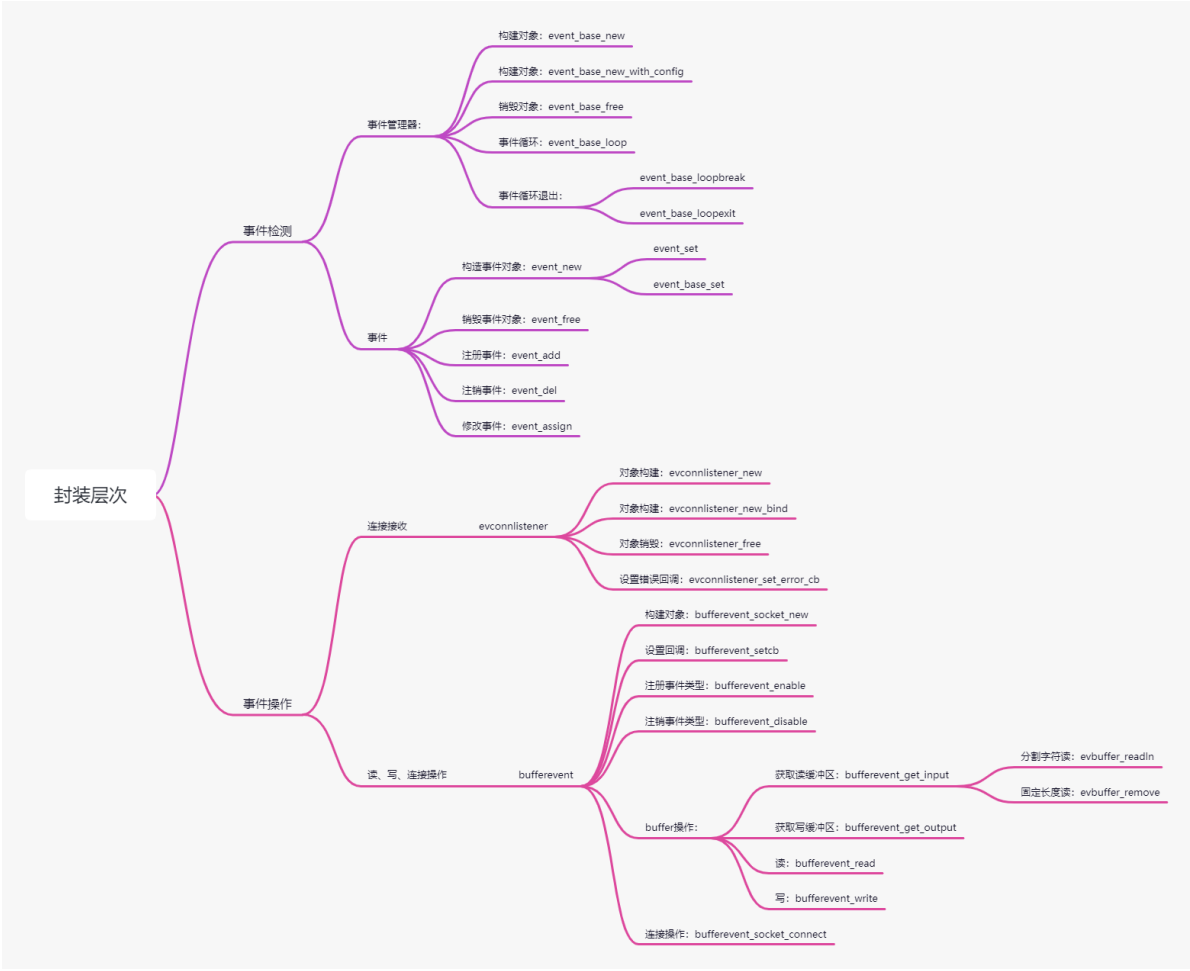
```
// 删除 evconnlistener 对象
void evconnlistener_free(struct evconnlistener *lev);

// 设置 listenfd 错误回调
void evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb);
```

bufferevent

```
struct bufferevent {
    /** Event base for which this bufferevent was created. */
    struct event_base *ev_base;
    /** Pointer to a table of function pointers to set up how this
        bufferevent behaves. */
    const struct bufferevent_ops *be_ops;

    /** A read event that triggers when a timeout has happened or a socket
        is ready to read data.  Only used by some subtypes of
        bufferevent. */
    struct event ev_read;
    /** A write event that triggers when a timeout has happened or a socket
        is ready to write data.  Only used by some subtypes of
        bufferevent. */
    struct event ev_write;

    /** An input buffer. Only the bufferevent is allowed to add data to
        this buffer, though the user is allowed to drain it. */
    struct evbuffer *input;

    /** An input buffer. Only the bufferevent is allowed to drain data
        from this buffer, though the user is allowed to add it. */
    struct evbuffer *output;

    struct event_watermark wm_read;
    struct event_watermark wm_write;

    bufferevent_data_cb readcb;
    bufferevent_data_cb writecb;
    /* This should be called 'eventcb', but renaming it would break
     * backward compatibility */
    bufferevent_event_cb errorcb;
    void *cbarg;

    struct timeval timeout_read;
    struct timeval timeout_write;

    /** Events that are currently enabled: currently EV_READ and EV_WRITE
        are supported. */
    short enabled;
};

// 新建 bufferevent 对象
struct bufferevent *bufferevent_socket_new(struct event_base *base,
evutil_socket_t fd, int options);

// 设置回调
void bufferevent_setcb(struct bufferevent *bufev,
```

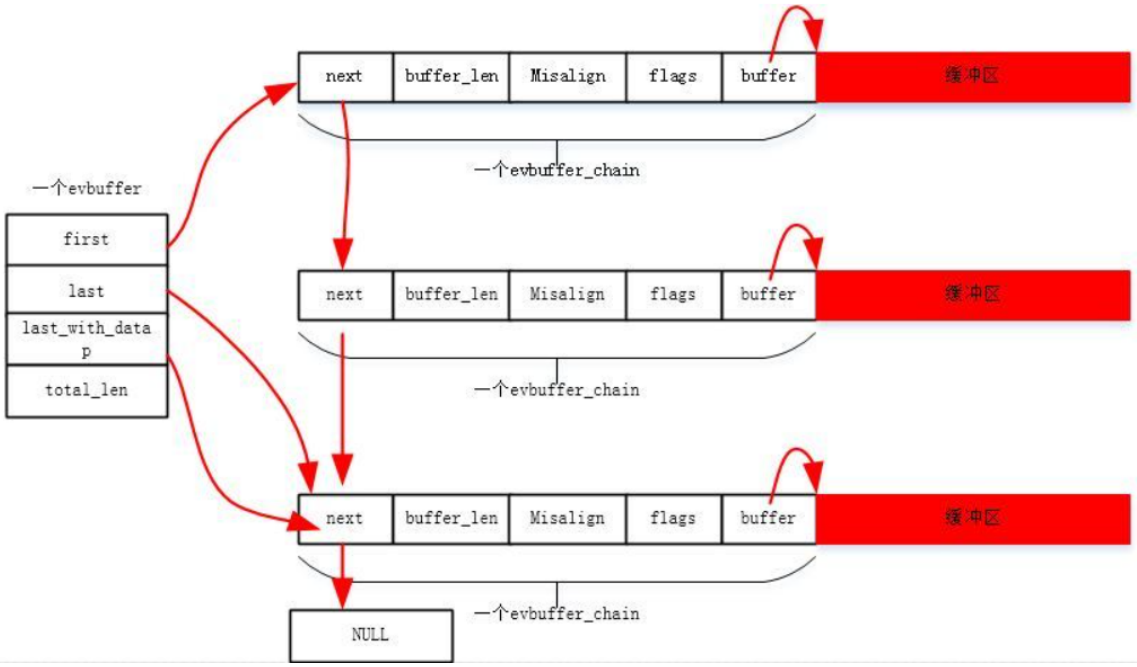
```
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg);

// 建立连接
int bufferevent_socket_connect(struct bufferevent *, const struct sockaddr
*, int);

// 释放 bufferevent 对象
void bufferevent_free(struct bufferevent *bufev);

// 获取 bufferevent 关联 fd
evutil_socket_t bufferevent_getfd(struct bufferevent *bufev);

// 写数据到写缓冲区
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);

// 从读缓冲区中读数据到 data 中
size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);

// 获取读缓冲区
struct evbuffer *bufferevent_get_input(struct bufferevent *bufev);

// 获取写缓冲区
struct evbuffer *bufferevent_get_output(struct bufferevent *bufev);

// 注册事件
int bufferevent_enable(struct bufferevent *bufev, short event);

// 注销事件
int bufferevent_disable(struct bufferevent *bufev, short event);

// 设置过滤器，比如处理对网络数据的加解密
struct bufferevent *
bufferevent_filter_new(struct bufferevent *underlying,
               bufferevent_filter_cb input_filter,
               bufferevent_filter_cb output_filter,
               int options,
               void (*free_context)(void *),
               void *ctx);
```

封装层次



evbuffer

> 图片来自网络，evbuffer_chain 中少了一个字段 off；



libev

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。
```

主要接口

ev_io_init

```
// 初始化 watcher 的 fd/events/callback
#define ev_io_init(ev,cb,fd,events) do { ev_init ((ev), (cb)); ev_io_set
((ev),(fd),(events)); } while (0)
```

ev_io_start

```
// 注册并绑定 io watcher 到 ev_loop
void ev_io_start(struct ev_loop *loop, ev_io *w);
```

ev_timer_start

```
// 注册并绑定 timer watcher 到 ev_loop
void ev_timer_start(struct ev_loop *loop, ev_timer *w);
```

ev_run

```
// 开启 ev_loop 的事件循环
int ev_run(struct ev_loop *loop, int flags);
```

重点

> 1. 掌握 libevent 封装层次；
> 2. 掌握 libevent 中 evbuffer 以及 readv、writev 的原理；
> 3. 掌握如何基于 libevent 构建应用；