Libevent 源码分析

Allen Xu

January 5, 2015

1 Libevent 简介

本文档以 Libevent-2.0.21 版本为基础，分析其代码，始于 2014 年 12 月 19 日。有关 windows 平台的代码均不去关注，文档中出现的需要表述的代码中与 windows 平台有关的均已略去。

作者在分析 Libevent 源码时，使用的分析工具是 Understand-3.1。

Libevent is an event notification library for developing scalable network servers. The Libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, Libevent also support callbacks due to signals or regular timeouts. Libevent

Libevent is meant to replace the event loop found in event driven network servers. An application just needs to call event_dispatch() and then add or remove events dynamically without having to change the event loop.

Currently, Libevent supports /dev/poll, kqueue(2), select(2), poll(2), epoll(4), and evports. The internal event mechanism is completely independent of the exposed event API, and a simple update of Libevent can provide new functionality without having to redesign the applications. As a result, Libevent allows for portable application development and provides the most scalable event notification mechanism available on an operating system. Libevent can also be used for multithreaded programs. Libevent should compile on Linux, *BSD, Mac OS X, Solaris and, Windows.

usage Standard usage

Every program that uses Libevent must inclurde the <event2/event.h> header, and pass the -levent flag to the linker. (You can instead link -levent_core if you only want the main event and buffered IO-based code, and don't want to link any protocol code.)

setup Library setup

Before you call any other Libevent functions, you need to set up the library. If you're going to use Libevent from multiple threads in a multithreaded application, you need to initialize thread support – typically by using evthread_use_pthreads() or evthread_use_windows_threads(). See <event2/thread.h> for more information.

This is also the point where you can replace Libevent's memory management functions with event_set_mem_functions, and enable debug mode with event_enable_debug_mode().

Creating an event base

Next, you need to create an event_base structure, using event_base_new() or event_base_new_with_c The event_base is responsible for keeping track of which events are "pending" (that is to say, being watched to see if they become active) and which events are "active". Every event is associated with a single event_base.

event Event notification

For each file descriptor that you wish to monitor, you must create an event structure with event_new(). (You may also declare an event structure and call event_assign() to initialize the members of the structure.) To enable notification, you add the structure to

the list of monitored events by calling event_add(). The event structure must remain allocated as long as it is active, so it should generally be allocated on the heap.

loop Dispaching evets.

Finally, you call event_base_dispatch() to loop and dispatch events. You can also use event_base_loop() for more fine-grained control.

Currently, only one thread can be dispatching a given event_base at a time. If you want to run events in multiple threads at once, you can either have a single event_base whose events add work to a work queue, or you can create multiple event_base objects.

bufferevent I/O Buffers

Libevent provides a buffered I/O abstraction on top of the regular event callbacks. This abstraction is called a bufferevent. A bufferevent provides input and output buffers that get filled and drained automatically. The user of a buffered event no longer deals directly with the I/O, but instead is reading from input and writing to output buffers.

Once initialized via bufferevent_socket_new(), the bufferevent structure can be used repeatedly with bufferevent_enable() and bufferevent_disable(). Instead of reading and writing directly to a socket, you would call bufferevent_read() and bufferevent_write().

When read enabled the bufferevent will try to read from the file descriptor and call the read callback. The write callback is executed whenever the output buffer is drained below the write low watermark, which is 0 by default. See <event2/bufferevent*.h> for more information.

timers Timers

Libevent can also be used to create timers that invoke a callback after a certain amount of time has expired. The evtimer_new() function returns an event struct to use as a timer. To activate the timer, call evtimer_add(). Timers can be deactivated by calling evtimer_del().

摘自 Libevent 库中 include/event2/event.h

2 主要数据结构

通过主要数据结构的关系，来初步观察 libevent 对各类 event 的管理框架

event 结构体 include/event2/event_struct.h

```
struct event {
    TAILQ_ENTRY(event) ev_active_next; //激活队列
    TAILQ_ENTRY(event) ev_next; //注册事件队列
    /* for managing timeouts */
    union {
        TAILQ_ENTRY(event) ev_next_with_common_timeout;
        int min_heap_idx; //指明该event结构体在堆的位置
    } ev_timeout_pos; //仅用于定时事件处理器(event).EV_TIMEOUT类型

    //对于I/O事件，是文件描述符；对于signal事件，是信号值
    evutil_socket_t ev_fd;

    struct event_base *ev_base; //所属的event_base

    //因为信号和I/O是不能同时设置的。所以可以使用共用体以省内存
    //在低版本的Libevent，两者是分开的，不在共用体内。
    union {
    //无论是信号还是IO，都有一个TAILQ_ENTRY的队列。它用于这样的情景:
    //用户对同一个fd调用event_new多次，并且都使用了不同的回调函数。
    //每次调用event_new都会产生一个event*。这个xxx_next成员就是把这些
    //event连接起来的。

        /* used for io events */
        //用于IO事件
        struct {
            TAILQ_ENTRY(event) ev_io_next;
            struct timeval ev_timeout;
        } ev_io;

        /* used by signal events */
        //用于信号事件
        struct {
            TAILQ_ENTRY(event) ev_signal_next;
            short ev_ncalls; //事件就绪执行时，调用ev_callback的次数          /* Allows deletes in
         callback */
            short *ev_pncalls; //指针，指向次数
        } ev_signal;
    } _ev;

    short ev_events;//记录监听的事件类型 EV_READ EVTIMEOUT之类
    short ev_res;       /* result passed to event callback */ //记录了当前激活事件的类型
    //libevent用于标记event信息的字段，表明其当前的状态.
    //可能值为前面的EVLIST_XXX
    short ev_flags;

    //本event的优先级。调用event_priority_set设置
    ev_uint8_t ev_pri;
    ev_uint8_t ev_closure;
    struct timeval ev_timeout;//用于定时器,指定定时器的超时值

    /* allows us to adopt for different types of events */
    void (*ev_callback)(evutil_socket_t, short, void *arg); //回调函数
    void *ev_arg; //回调函数的参数
};
```

event 结构体里面有几个 TAILQ_ENTRY 队列节点类型。这里因为一个 event 是会同时处于多个队列之中。比如同一个文件描述符或者信号值对应的多个 event 会被连在一起，所有的被加入到 event_base 的 event 也会连在一起，所有被激活的 event 也会被连在

一起。所以会有多个 QAILQ_ENTRY。

event_base 结构体 event-internal.h

```
1  struct event_base {
2      /** Function pointers and other data to describe this event_base's backend. */
3      const struct eventop *evsel;
4      /** Pointer to backend-specific data. */
5      void *evbase;
6
7      /** List of changes to tell backend about at next dispatch.  Only used by the O(1) backends. */
8      struct event_changelist changelist;
9
10     /** Function pointers used to describe the backend that this event_base uses for signals */
11     const struct eventop *evsigsel;
12     /** Data to implement the common signal handelr code. */
13     struct evsig_info sig;
14
15     /** Number of virtual events */
16     int virtual_event_count;
17     /** Number of total events added to this event_base */
18     int event_count;
19     /** Number of total events active in this event_base */
20     int event_count_active;
21
22     /** Set if we should terminate the loop once we're done processing events. */
23     int event_gotterm;
24     /** Set if we should terminate the loop immediately */
25     int event_break;
26     /** Set if we should start a new instance of the loop immediately. */
27     int event_continue;
28
29     /** The currently running priority of events */
30     int event_running_priority;
31
32     /** Set if we're running the event_base_loop function, to prevent
33      * reentrant invocation. */
34     int running_loop;
35
36     /* Active event management. */
37     /** An array of nactivequeues queues for active events (ones that have triggered,
38     and whose callbacks need to be called).  Low priority numbers are more important,
39     and stall higher ones.*/
40     struct event_list *activequeues;
41     /** The length of the activequeues array */
42     int nactivequeues;
43
44     /* common timeout logic */
45
46     /** An array of common_timeout_list* for all of the common timeout values we know. */
47     struct common_timeout_list **common_timeout_queues;
48     /** The number of entries used in common_timeout_queues */
49     int n_common_timeouts;
50     /** The total size of common_timeout_queues. */
51     int n_common_timeouts_allocated;
52
53     /** List of defered_cb that are active.  We run these after the active events. */
54     struct deferred_cb_queue defer_queue;
55
56     /** Mapping from file descriptors to enabled (added) events */
57     struct event_io_map io;
58
59     /** Mapping from signal numbers to enabled (added) events. */
60     struct event_signal_map sigmap;
```

```
61
62      /** All events that have been enabled (added) in this event_base */
63      struct event_list eventqueue;
64
65      /** Stored timeval; used to detect when time is running backwards. */
66      struct timeval event_tv;
67
68      /** Priority queue of events with timeouts. */
69      struct min_heap timeheap;
70
71      /** Stored timeval: used to avoid calling gettimeofday/clock_gettime too often. */
72      struct timeval tv_cache;
73
74  #if defined(_EVENT_HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
75      /** Difference between internal time (maybe from clock_gettime) and gettimeofday. */
76      struct timeval tv_clock_diff;
77      /** Second in which we last updated tv_clock_diff, in monotonic time. */
78      time_t last_updated_clock_diff;
79  #endif
80
81  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
82      /* threading support */
83      /** The thread currently running the event_loop for this base */
84      unsigned long th_owner_id;
85      /** A lock to prevent conflicting accesses to this event_base */
86      void *th_base_lock;
87      /** The event whose callback is executing right now */
88      struct event *current_event;
89      /** A condition that gets signalled when we're done processing an event with waiters on it. */
90      void *current_event_cond;
91      /** Number of threads blocking on current_event_cond. */
92      int current_event_waiters;
93  #endif
94
95      /** Flags that this base was configured with */
96      enum event_base_config_flag flags;
97
98      /* Notify main thread to wake up break, etc. */
99      /** True if the base already has a pending notify, and we don't need to add any more. */
100     int is_notify_pending;
101     /** A socketpair used by some th_notify functions to wake up the main thread. */
102     evutil_socket_t th_notify_fd[2];
103     /** An event used by some th_notify functions to wake up the main thread. */
104     struct event th_notify;
105     /** A function used to wake up the main thread from another thread. */
106     int (*th_notify_fn)(struct event_base *base);
107 };
```

在一个 event loop 中只会有一个 event_base 结构体对象存在。

event_io_map 结构体 event-internal.h

```
1  #ifdef EVMAP_USE_HT
2  #include "ht-internal.h"
3  struct event_map_entry;
4  HT_HEAD(event_io_map, event_map_entry);
5  #else
6  #define event_io_map event_signal_map
7  #endif
```

其中宏 EVMAP_USE_HT 的定义在 If we're on win32, then file descriptors are not nice low densely packed integers. Instead, they are pointer-like windows handles, and we want to use a hashtable instead of an array to map fds to events.

```
#ifdef WIN32
#define EVMAP_USE_HT
#endif
```

event_signal_map 结构体 event-internal.h

```
struct event_signal_map {
    /* An array of evmap_io * or of evmap_signal *; empty entries are set to NULL. */
    void **entries;
    /* The number of entries available in entries */
    int nentries;
};
```

event 管理，框架图

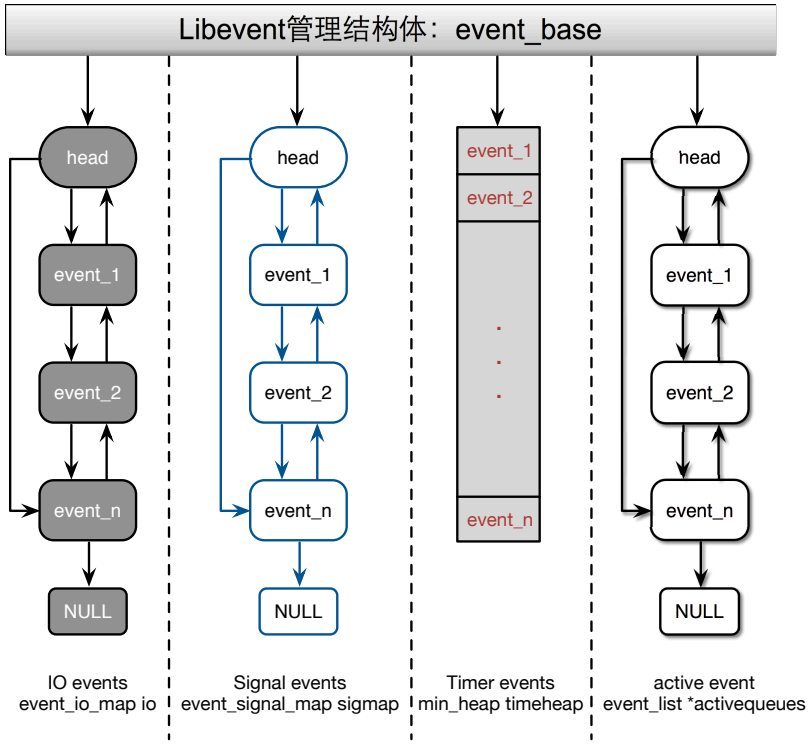

Figure 1: Events Management.

3 多线程、锁、条件变量

Libevent 原生是没有多线程模型的，需要开发者自己编写多线程相关的代码。有了多线程之后，会涉及到加锁、解锁、条件变量以及线程安全等相关内容。evthread、lock 相关的宏、函数、线程安全相关知识

4 Libevent 之 Reactor

本章主要是 Libevent 对多路 IO 复用机制的封装。

4.1 Reactor Vs Proactor

一般情况下，I/O 复用机制需要事件分离器 (event demultiplexor)。事件分离器的作用：将那些读写事件源分发给各读写事件的处理者。开发人员在开始的时候需要在分离器那里注册需要关注的事件，并提供相应的处理者 (event handlers)，或者是回调函数；事件分离器在适当的时候会将请求的事件分发给这些 handler 或者回调函数。

目前，事件分用器存在两种模式：Reactor 和 Proactor。Reactor 模式基于同步 I/O，而 Proactor 模式基于异步 I/O。在 Reactor 模式中，事件分离器等待某个事件的发生（比如文件描述符可读写、socket 可读写、定时器超时或者信号），事件分离器将此事件传给事先注册的事件处理函数或回调函数，由后者来做实际的读写操作。

而在 Proactor 模式中，事件处理者 (或由事件分离器发起) 直接发起一个异步读写操作 (相当于请求)，而实际工作是由操作系统完成的。发起时，需要提供的参数包括存放读到数据的缓冲区，读的数据大小，或者存放外发数据的缓冲区，以及这个请求完后的回调函数等信息。事件分离器得知了这个请求，它默默等待这个请求的完成，然后转发**完成事件**给相应的事件处理者或者回调。

为了更好地理解 Reactor 与 Proactor 两种模式的区别，下面用 read 操作的例子来看一下两者的步骤。下面是 Reactor 的做法：

1. 某事件处理者向事件分离器注册某个 socket 上的读事件；
2. 事件分离者等着这个事件的发生（可能会有一些其他事件）；
3. 当事件发生了，事件分离器被唤醒，将读事件分发给相应事件处理者；
4. 事件处理者于是去那个 socket 上读数据。若需要，它再次注册 socket 上的读事件，重复上面的步骤。

下面再来看看 Proactor（需要操作系统支持）是如何做的：

1. 事件处理者直接请求一个读操作，然后等待该读操作的完成；
2. 事件分离器等着这个读事件的完成，等待该事件的完成事件；
3. 事件分离器等待的同时，OS 已经在执行该读操作：读取目标数据，放入用户提供的缓冲区中，最后通知事件分离器，即完成事件;
4. 事件分离器通知之前的事件处理者: 该读事件已经完成;
5. 事件处理者即可处理该读操作获得的数据。若需要，事件处理者再次请求一个写操作，重复上述几个步骤。

按照大多数人的观点，Proactor 的性能会比 Reactor 好（暂时没有实验测试）。目前 windows、linux 都有对异步 IO 的支持。也有一些网络库已经支持 Proactor 模式。Libevent 采用的是 Reactor 模式，其实现了跨平台的多路 IO 复用接口封装。这使得用户可以在不同的平台使用统一的接口。接下来就讲一讲其跨平台的实现。[1]

[1] 由于 Libevent 库是一个通用的开源网络库，考虑到应用环境的多样性，故其实现了多路 IO 复用的封装。但在许多实际应用场景中，跨平台是没有必要的。

4.2 Reactor 相关结构体

下面代码是 Libevent 实现跨平台 IO 多路复用的相关数据结构：

```
//event-internal.h 文件
struct event_base {
    const struct eventop *evsel;     //多路 IO 复用函数指针结构体
    void *evbase;                    //
    ......
};

struct eventop {
    const char *name;                //多路 IO 复用函数的名字

    void *(* init )(struct event_base *);
    int  (*add)(struct event_base *, evutil_socket_t  fd,  short old,  short events, void *fdinfo );
    int  (*del)(struct event_base *, evutil_socket_t  fd,  short old,  short events, void *fdinfo );
    int  (*dispatch)(struct event_base *, struct timeval  *);
    void (*dealloc)(struct event_base *);

    int  need_reinit;                        //是否要重新初始化，0 表示不需要
    enum event_method_feature features;      //多路 IO 复用的特征，详见下文解释
    size_t  fdinfo_len;                      //额外信息的长度。有些多路 IO 复用函数需要额外的信息
};
```

event_base 结构体是 Libevent 中最核心的结构体，其详细介绍参见第2章。此处我们关注 evsel 和 evbase 成员。evsel 是一个 struct eventop 结构体指针，struct eventop 的成员是一些函数指针，实际上这些函数指针最终指向的就是多路 IO 复用函数中对应的函数。只需要给这些指针赋予相应的多路 IO 复用的函数即可。

- init Function to set up an event_base to use this backend. It should create a new structure holding whatever information is needed to run the backend, and return it. The returned pointer will get stored by event_init into the event_base.evbase field. On failure, this function should return NULL.
- add Enable reading/writing on a given fd or signal. 'events' will be the events that we're trying to enable: one or more of EV_READ, EV_WRITE, EV_SIGNAL, and EV_ET. 'old' will be those events that were enabled on this fd previously. 'fdinfo' will be a structure associated with the fd by the evmap; its size is defined by the fdinfo field below. It will be set to 0 the first time the fd is added. The function should return 0 on success and -1 on error.
- del As "add", except 'events' contains the events we mean to disable.
- dispatch 可以进入监听 Function to implement the core of an event loop. It must see which added events are ready, and cause event_active to be called for each active event (usually via event_io_active or such). It should return 0 on success and -1 on error.。
- dealloc Function to clean up and free our data from the event_base.

feature 成员指定多路 IO 复用函数应该满足哪些特征。所有的特征定义在一个枚举类型中，在使用中可以调用 event_config_avoid_method() 可以通过名字让 libevent 避免使用特定的可用后端。调用 event_config_require_feature() 让 libevent 不使用不能提供所有指定特征的后端，event_config_require_features() 可识别的特征值有：

```
//event.h 文件
enum event_method_feature {
    EV_FEATURE_ET = 0x01,     //支持边沿触发
    EV_FEATURE_O1 = 0x02,     //要求事件分派器时间复杂度为 O(1)，排除 select、poll 等
```

```
    EV_FEATURE_FDS = 0x04     //支持任意文件描述符，而不能仅仅支持套接字
};
```

4.3 多路 IO 复用机制

现有的许多平台都有不止一种多路 IO 复用机制，例如 Linux 就支持 select、poll、epoll，而 BSD 支持的包括 select、kqueue。关于各个多路 IO 复用机制此处没有详细介绍，网络上有许多很好的资料。Libevent 支持的多路 IO 复用机制包括：select、poll、epoll、kqueue、devpoll 等。其封装都在相对应的代码文件里，例如 select 的函数接口声明如下：

```
//select.c 文件
static void * select_init (struct event_base *);
static int select_add(struct event_base *, int, short old, short events, void*);
static int select_del(struct event_base *, int, short old, short events, void*);
static int select_dispatch(struct event_base *, struct timeval *);
static void select_dealloc(struct event_base *);

const struct eventop selectops = {
    "select",
     select_init ,
    select_add,
    select_del,
    select_dispatch,
    select_dealloc,
    0, EV_FEATURE_FDS, 0,
};
```

有些多路 IO 复用机制的实现相比这个两个会复杂一些，如 epoll、kqueue 等。

```
//epoll.c 文件
static void * epoll_init (struct event_base *);
static int epoll_dispatch(struct event_base *, struct timeval *);
static void epoll_dealloc(struct event_base *);

static const struct eventop epollops_changelist = {
    "epoll␣(with␣changelist)",
     epoll_init ,
    event_changelist_add,
    event_changelist_del,
    epoll_dispatch,
    epoll_dealloc,
    1,
    EV_FEATURE_ET|EV_FEATURE_O1,
    EVENT_CHANGELIST_FDINFO_SIZE
};

static int epoll_nochangelist_add(struct event_base *base, evutil_socket_t fd,
    short old, short events, void *p);
static int epoll_nochangelist_del(struct event_base *base, evutil_socket_t fd,
    short old, short events, void *p);

const struct eventop epollops = {
    "epoll",
     epoll_init ,
    epoll_nochangelist_add,
    epoll_nochangelist_del,
    epoll_dispatch,
    epoll_dealloc,
    1,
    EV_FEATURE_ET|EV_FEATURE_O1,
    0
```

```
};
```

注意到 epoll 定义了两个 struct eventop 结构，直观的观察，是前者是使用的基于 changelist 的事件添加删除函数，而后者使用 epoll 自带的相关函数接口。这个可以调用 event_config_set_flag() 让 libevent 在创建 event_base 时设置一个或者多个将在下面介绍的运行时标志，event_config_set_flag() 可识别的选项值有：

```
enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOCK = 0x01,
    EVENT_BASE_FLAG_IGNORE_ENV = 0x02,
    EVENT_BASE_FLAG_STARTUP_IOCP = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
    EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10
};
```

- EVENT_BASE_FLAG_NOLOCK：不要为 event_base 分配锁。设置这个选项可以为 event_base 节省一点用于锁定和解锁的时间，但是让在多个线程中访问 event_base 成为不安全的。
- EVENT_BASE_FLAG_IGNORE_ENV：选择使用的后端时，不要检测 EVENT_* 环境变量。使用这个标志需要三思：这会让用户更难调试你的程序与 libevent 的交互。
- EVENT_BASE_FLAG_STARTUP_IOCP：仅用于 Windows，让 libevent 在启动时就启用任何必需的 IOCP 分发逻辑，而不是按需启用。
- EVENT_BASE_FLAG_NO_CACHE_TIME：不是在事件循环每次准备执行超时回调时检测当前时间，而是在每次超时回调后进行检测。注意：这会消耗更多的 CPU 时间。
- EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST：告诉 libevent，如果使用 epoll 后端，可以安全地使用更快的基于 changelist 的后端。epoll-changelist 后端可以在后端的分发函数调用之间，同样的 fd 多次修改其状态的情况下，避免不必要的系统调用。但如果传递任何使用 dup() 或者其变体克隆的 fd 给 libevent, epoll-changelist 后端会触发一个内核 bug，导致不正确的结果。在不使用 epoll 后端的情况下，这个标志无效。也可以通过设置 EVENT_EPOLL_USE_CHANGELIST 环境变量来打开 epoll-changelist 选项。

【比较基于 changelist 和非 changelist 之间的区别】

4.4 选择多路 IO 复用后端

看到这里，想必已经知道，只需将对应平台的多路 IO 复用函数的全局变量赋值给 event_base 的 evsel 变量即可。可是怎么让 Libevent 根据不同的平台选择不同的多路 IO 复用函数呢？另外大部分 OS 都会实现 select、poll 和一个自己的高效多路 IO 复用函数。怎么从多个中选择一个甚至是最优的呢？下面看一下 Libevent 的解决方案：

```
//event.c 文件
#ifdef _EVENT_HAVE_EVENT_PORTS
extern const struct eventop evportops;
#endif
#ifdef _EVENT_HAVE_SELECT
extern const struct eventop selectops;
#endif
#ifdef _EVENT_HAVE_POLL
extern const struct eventop pollops;
#endif
#ifdef _EVENT_HAVE_EPOLL
```

```
12 extern const struct eventop epollops;
13 #endif
14 #ifdef _EVENT_HAVE_WORKING_KQUEUE
15 extern const struct eventop kqops;
16 #endif
17 #ifdef _EVENT_HAVE_DEVPOLL
18 extern const struct eventop devpollops;
19 #endif
20 #ifdef WIN32
21 extern const struct eventop win32ops;
22 #endif
23
24 static const struct eventop *eventops[] = {
25 #ifdef _EVENT_HAVE_EVENT_PORTS
26     &evportops,
27 #endif
28 #ifdef _EVENT_HAVE_WORKING_KQUEUE
29     &kqops,
30 #endif
31 #ifdef _EVENT_HAVE_EPOLL
32     &epollops,
33 #endif
34 #ifdef _EVENT_HAVE_DEVPOLL
35     &devpollops,
36 #endif
37 #ifdef _EVENT_HAVE_POLL
38     &pollops,
39 #endif
40 #ifdef _EVENT_HAVE_SELECT
41     &selectops,
42 #endif
43 #ifdef WIN32
44     &win32ops,
45 #endif
46     NULL
47 };
```

它根据宏定义判断当前的 OS 环境是否有某个多路 IO 复用函数。如果有，那么就把与之对应的 struct eventop 结构体指针放到一个全局数组中。有了这个数组，现在只需将数组的某个元素赋值给 evsel 变量即可。因为是条件宏，在编译器编译代码之前完成宏的替换，所以是可以这样定义一个数组的。这些宏都在 event-config.h 中有定义了，该文件是一个很基础和重要的文件。在文件的一开始有这样一句”This file was generated by autoconf when libevent was built”。这说明这个文件是在 Libevent 配置的时候生成的，即在编译 Libevent 之前就应该要生成该文件。

从数组的元素可以看到，低下标存的是高效多路 IO 复用函数。如果从低到高下标选取一个多路 IO 复用函数，那么将优先选择高效的。现在看一下 Libevent 如何选取后端：

```
1  //event.c文件
2  struct event_base *event_base_new_with_config(const struct event_config *cfg) {
3      int i;
4      struct event_base *base;
5      int should_check_environment;
6
7      /* 分配并清零 event_base 内存. event_base 所有成员均初始化为 0 */
8      if ((base = mm_calloc(1, sizeof(struct event_base))) == NULL) {
9          event_warn("%s: calloc", __func__);
10         return NULL;
11     }
12     ...……
13     should_check_environment =
14         !(cfg && (cfg->flags & EVENT_BASE_FLAG_IGNORE_ENV));
15     /* 遍历数组元素 */
```

```
16      for (i = 0; eventops[i] && !base->evbase; i++) {
17          if (cfg != NULL) {
18          /* 判断该后端是否被禁用 */
19          if (event_config_is_avoided_method(cfg, eventops[i]->name))
20              continue;
21          if ((eventops[i]->features & cfg->require_features) != cfg->require_features)
22              continue;
23          }
24
25          /* also obey the environment variables */
26          if (should_check_environment && event_is_method_disabled(eventops[i]->name))
27              continue;
28
29          /* 找到一个满足条件的多路 IO 复用后端 */
30          base->evsel = eventops[i];
31
32          base->evbase = base->evsel->init(base);
33      }
34
35      if (base->evbase == NULL) {
36          event_warnx("%s: no event mechanism available", __func__);
37          base->evsel = NULL;
38          event_base_free(base);
39          return NULL;
40      }
41      ……
42      return (base);
43  }
```

event_base_new_with_config 用于根据 event_config 来创建 event_base，将会在下一章详细介绍。此处关注其中的 for 循环，可以看到，首先从 eventops 数组中选出一个元素。如果设置了 event_config，那么就对这个元素 (即多路 IO 复用函数) 特征进行检测，看其是否满足 event_config 所描述的特征。找到第一个符合条件的 eventops 元素并赋值给 evsel，根据 for 循环的判断条件：eventops[i] && !base->evbase，此 for 循环结束。

【此处添加对 event_config 的解释】

后端数据存储结构体：在本文最前面列出的 event_base 结构体中，除了 evsel 变量外，还有一个 evbase 变量。这也是一个很重要的变量，而且也是用于跨平台的。像 select、poll、epoll 之类多路 IO 复用函数在调用时要传入一些数据，比如监听的文件描述符 fd，监听了的哪些事件。在 Libevent 中，这些数据不是保存在 event_base 这个结构体中的，而是存放在 evbase 这个指针指向的结构体中。

需要注意到 evbase 是 void 指针类型，这是由于不同的多路 IO 复用函数需要使用不同格式的数据，所以 Libevent 为每一个多路 IO 复用函数都定义了专门的结构体 (即结构体是不同的)，暂且称之为 IO 复用结构体。evbase 指向的就是这些结构体。由于这些结构体是不同的，所以要用一个 void 类型指针。epoll 的 IO 复用结构体，如下面代码所示：

```
1   /* epoll.c 文件 */
2   struct epollop {
3       struct epoll_event *events;
4       int nevents;
5       int epfd;
6   };
```

前面 event_base_new_with_config 的代码中，即该函数第 34 行，调用了 init 函数。这行代码就是用来赋值 evbase 的。下面是 epoll 对应的 init 函数：

```
1   /* epoll.c 文件 */
2   static void * epoll_init (struct event_base *base) {
3       int epfd;
4       struct epollop *epollop;
```

```
5
6       /*  Initialize  the kernel queue. (The size field  is  ignored since  2.6.8.)  */
7       if  ((epfd = epoll_create(32000)) == −1) {
8           if  (errno != ENOSYS)
9               event_warn(”epoll_create”);
10          return (NULL);
11      }
12
13      evutil_make_socket_closeonexec(epfd);
14
15      if  (!(epollop = mm_calloc(1, sizeof(struct epollop)))) {
16          close(epfd);
17          return (NULL);
18      }
19
20      epollop−>epfd = epfd;
21
22      /*  Initialize  fields  */
23      epollop−>events = mm_calloc(INITIAL_NEVENT, sizeof(struct epoll_event));
24      if  (epollop−>events == NULL) {
25          mm_free(epollop);
26          close(epfd);
27          return (NULL);
28      }
29      epollop−>nevents = INITIAL_NEVENT;
30
31      if  ((base−>flags & EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST) != 0 ||
32          ((base−>flags & EVENT_BASE_FLAG_IGNORE_ENV) == 0 &&
33          evutil_getenv(”EVENT_EPOLL_USE_CHANGELIST”) != NULL))
34          base−>evsel = &epollops_changelist;
35
36      evsig_init (base);
37
38      return (epollop);
39  }
```

经过上面的处理后，Libevent 在特定的 OS 下能使用到特定的多路 IO 复用函数。在之前说到的 evmap_io_add 和 evmap_signal_add 函数中都会调用 evsel->add。由于在新建 event_base 时就选定了对应的多路 IO 复用函数，给 evsel、evbase 变量赋值，所以 evsel->add 能把对应的 fd 和监听事件加到对应的 IO 复用结构体保存。由于有 evsel 和 evbase 这个两个指针变量，当初始化完成之后，再也不用担心具体使用的是哪个多路 IO 复用后端。evsel 结构体的函数指针提供了统一的接口，上层的代码要使用到多路 IO 复用函数的一些操作函数时，直接调用 evsel 结构体提供的函数指针即可。也正是如此，Libevent 实现了统一的跨平台 Reactor 接口。

5 Libevent 核心流程

本章主要介绍 Libevent 的核心事件流程。event.c 中的核心框架

5.1 简单示例

为了能够清晰的把握核心流程而不被细枝末节而打乱，首先看一个最简单的 libevent 的示例：

```
#include<unistd.h>
#include<stdio.h>
#include<thread.h>

#include<event2/event.h>

void example_cb(int fd, short events, void *arg) {
    char buf[512];
     printf ("In the example_cb\n");
    read(fd, buf, sizeof(buf));
}

int main() {
    /* 使用 event_base 默认配置 */
    struct event_base *base = event_base_new();

    struct event *example_ev = event_new(base, STDIN_FILENO,
                        EV_READ | EV_PERSIST, example_cb, NULL);

    event_add(example_ev, NULL); /* 没有超时 */

    event_base_dispatch(base);

    return 0;
}
```

这个例子已经包含了 Libevent 的基础工作流程。用 event_base_new 初始化一个 event_base 结构体，这个结构体是 libevent 中核心数据结构，他将原本独立的各个单元整合在一起。然后 event_new 创建一个 event 事件，该事件监听标准输入的读事件，并设置为永久事件 (EV_PERSIST)。再将此新创建的一个事件添加到 event_base 中。最后调用事件分派函数，开始一直监听已经添加到 event_base 中的事件。本章后面的内容将按照这样的主线进行分析。

5.2 创建 event_base

event_base_new 函数可以创建一个默认配置的 event_base 结构体。它先用 event_config_new 创建一个 event_config，这个配置结构体默认是空的，也就是说不包含任何配置信息，然后调用 event_base_new_with_config 函数创建默认配置的 event_base 结构体。下面先看一下 event_base_new_with_config：

```
/* event.c 文件 */
struct event_base * event_base_new(void) {
    struct event_base *base = NULL;
    struct event_config *cfg = event_config_new();
    if (cfg) {
        base = event_base_new_with_config(cfg);
        event_config_free(cfg);
    }
    return base;
}

struct event_base * event_base_new_with_config(const struct event_config *cfg) {
```

```
13      int i;
14      struct event_base *base;
15      int should_check_environment;
16
17  #ifndef _EVENT_DISABLE_DEBUG_MODE
18      event_debug_mode_too_late = 1;
19  #endif
20
21      if ((base = mm_calloc(1, sizeof(struct event_base))) == NULL) {
22          event_warn("%s: calloc", __func__);
23          return NULL;
24      }
25      detect_monotonic();
26      gettime(base, &base->event_tv);
27
28      min_heap_ctor(&base->timeheap);
29      TAILQ_INIT(&base->eventqueue);
30      base->sig.ev_signal_pair[0] = -1;
31      base->sig.ev_signal_pair[1] = -1;
32      base->th_notify_fd[0] = -1;
33      base->th_notify_fd[1] = -1;
34
35      event_deferred_cb_queue_init(&base->defer_queue);
36      base->defer_queue.notify_fn = notify_base_cbq_callback;
37      base->defer_queue.notify_arg = base;
38      if (cfg)
39          base->flags = cfg->flags;
40
41      evmap_io_initmap(&base->io);
42      evmap_signal_initmap(&base->sigmap);
43      event_changelist_init(&base->changelist);
44
45      base->evbase = NULL;
46
47      should_check_environment =
48          !(cfg && (cfg->flags & EVENT_BASE_FLAG_IGNORE_ENV));
49
50      /* 选择 IO 复用后端，前文已经讲解过 */
51      ……
52
53      if (base->evbase == NULL) {
54          event_warnx("%s: no event mechanism available", __func__);
55          base->evsel = NULL;
56          event_base_free(base);
57          return NULL;
58      }
59
60      if (evutil_getenv("EVENT_SHOW_METHOD"))
61          event_msgx("libevent using: %s", base->evsel->name);
62
63      /* allocate a single active event queue */
64      if (event_base_priority_init(base, 1) < 0) {
65          event_base_free(base);
66          return NULL;
67      }
68
69      /* prepare for threading */
70  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
71      //测试 evthread_lock_callbacks 结构中的 lock 指针函数是否为 NULL//即测试 Libevent 是否已经初始
            化为支持多线程模式。//由于一开始是用 mm_calloc 申请内存的，所以该内存区域的值为 0//对于
            th_base_lock 变量，目前的值为 NULL.
72      if (EVTHREAD_LOCKING_ENABLED() &&
73          (!cfg || !(cfg->flags & EVENT_BASE_FLAG_NOLOCK))) {
74          int r;
```

```
75          EVTHREAD_ALLOC_LOCK(base->th_base_lock,
76              EVTHREAD_LOCKTYPE_RECURSIVE);
77          base->defer_queue.lock = base->th_base_lock;
78          EVTHREAD_ALLOC_COND(base->current_event_cond);
79          r = evthread_make_base_notifiable(base);
80          if (r<0) {
81              event_warnx("%s: Unable to make base notifiable.", __func__);
82              event_base_free(base);
83              return NULL;
84          }
85      }
86  #endif
87
88  #ifdef WIN32
89      ......
90  #endif
91
92      return (base);
93  }
```

有时，需要对某些方面有些特殊的要求，此时就不能使用默认配置的 event_base 了，需要对 event_base 进行配置。这里用到了 event_config 结构体，【配置 event_base】。这个结构体主要是对 event_base 进行一些配置。

宏 EVTHREAD_LOCKING_ENABLED 主要是检测是否已经支持锁了。检测的方式也很简单，也就是检测 _evthread_lock_fns 全局变量中的 lock 成员变量是否不为 NULL。有关这个 _evthread_lock_fns 全局变量在【多线程、锁、条件变量】中讲解。

5.3 创建 event

现在 event_base 已经新建出来了。下面看一下 event_new 函数，它和前面的 event_base_new 一样，把主要是的初始化工作交给另一个函数。event_new 函数的工作只是创建一个 struct event 结构体，然后把它的参数原封不动地传给 event_assign，所以还是看 event_assign 函数。

```
1   struct event *event_new(struct event_base *base, evutil_socket_t fd, short events,
2                           void (*cb)(evutil_socket_t, short, void *), void *arg) {
3       struct event *ev;
4       ev = mm_malloc(sizeof(struct event));
5       if (ev == NULL)
6           return (NULL);
7       if (event_assign(ev, base, fd, events, cb, arg) < 0) {
8           mm_free(ev);
9           return (NULL);
10      }
11
12      return (ev);
13  }
14
15  int event_assign(struct event *ev, struct event_base *base, evutil_socket_t fd,
16                   short events, void (*callback)(evutil_socket_t, short, void *), void *arg) {
17      if (!base)
18          base = current_base;
19
20      _event_debug_assert_not_added(ev);
21
22      ev->ev_base = base;
23
24      ev->ev_callback = callback;
25      ev->ev_arg = arg;
26      ev->ev_fd = fd;
27      ev->ev_events = events;
```

```
28      ev->ev_res = 0;
29      ev->ev_flags = EVLIST_INIT;
30      ev->ev_ncalls = 0;
31      ev->ev_pncalls = NULL;
32
33      if (events & EV_SIGNAL) {
34          if ((events & (EV_READ|EV_WRITE)) != 0) {
35              event_warnx("%s: EV_SIGNAL is not compatible with "
36                  "EV_READ or EV_WRITE", __func__);
37              return -1;
38          }
39          ev->ev_closure = EV_CLOSURE_SIGNAL;
40      } else {
41          if (events & EV_PERSIST) {
42              evutil_timerclear(&ev->ev_io_timeout);
43              ev->ev_closure = EV_CLOSURE_PERSIST;
44          } else {
45              ev->ev_closure = EV_CLOSURE_NONE;
46          }
47      }
48
49      min_heap_elem_init(ev);
50
51      if (base != NULL) {
52          /* by default, we put new events into the middle priority */
53          ev->ev_pri = base->nactivequeues / 2;
54      }
55
56      _event_debug_note_setup(ev);
57
58      return 0;
59  }
```

从 event_assign 函数的名字可以得知它是进行赋值操作的。所以它能可以在 event 被初始化后再次调用。不过，初始化后再次调用的话，有些事情要注意。【注意事项】从上面的代码可看到：如果这个 event 是用来监听一个信号的，那么就不能让这个 event 监听读或者写事件。注意，此时 event 结构体的变量 ev_flags 的值是 EVLIST_INIT。对变量的追踪是很有帮助的。它指明了 event 结构体的状态。它通过以或运算的方式取下面的值：

```
1   //event_struct.h文件
2   #define EVLIST_TIMEOUT  0x01      //event从属于定时器队列或者时间堆
3   #define EVLIST_INSERTED 0x02      //event从属于注册队列
4   #define EVLIST_SIGNAL   0x04        //没有使用
5   #define EVLIST_ACTIVE   0x08        //event从属于活动队列
6   #define EVLIST_INTERNAL 0x10      //该event是内部使用的。信号处理时有用到
7   #define EVLIST_INIT 0x80          //event已经被初始化了
8
9   /* EVLIST_X_ Private space: 0x1000-0xf000 */
10  #define EVLIST_ALL  (0xf000 | 0x9f)     //所有标志。这个不能取
```

5.4 向 event_base 添加 event

创建完一个 event 结构体后，现在看一下 event_add。它同前面的函数一样，内部也是调用其他函数完成工作。因为它用到了锁，所以给出它的代码。

```
1   int event_add(struct event *ev, const struct timeval *tv) {
2       int res;
3
4       if (EVUTIL_FAILURE_CHECK(!ev->ev_base)) {
5           event_warnx("%s: event has no event_base set.", __func__);
6           return -1;
```

```
7       }
8
9       EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);
10
11      res = event_add_internal(ev, tv, 0);
12
13      EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);
14
15      return (res);
16  }
17
18  /* Implementation function to add an event.  Works just like event_add,
19   * except: 1) it requires that we have the lock.  2) if tv_is_absolute is set,
20   * we treat tv as an absolute time, not as an interval to add to the current
21   * time */
22  static inline int event_add_internal(struct event *ev, const struct timeval *tv, int tv_is_absolute) {
23      struct event_base *base = ev->ev_base;
24      int res = 0;
25      int notify = 0;
26
27      EVENT_BASE_ASSERT_LOCKED(base);
28      _event_debug_assert_is_setup(ev);
29
30      event_debug((
31          "event_add: event: %p (fd "EV_SOCK_FMT"), %s%s%scall %p",
32          ev,
33          EV_SOCK_ARG(ev->ev_fd),
34          ev->ev_events & EV_READ ? "EV_READ " : " ",
35          ev->ev_events & EV_WRITE ? "EV_WRITE " : " ",
36          tv ? "EV_TIMEOUT " : " ",
37          ev->ev_callback));
38
39      EVUTIL_ASSERT(!(ev->ev_flags & ~EVLIST_ALL));
40
41      /*
42       * prepare for timeout insertion further below, if we get a
43       * failure on any step, we should not change any state.
44       */
45      if (tv != NULL && !(ev->ev_flags & EVLIST_TIMEOUT)) {
46          if (min_heap_reserve(&base->timeheap,
47              1 + min_heap_size(&base->timeheap)) == -1)
48              return (-1);  /* ENOMEM == errno */
49      }
50
51      /* If the main thread is currently executing a signal event's
52       * callback, and we are not the main thread, then we want to wait
53       * until the callback is done before we mess with the event, or else
54       * we can race on ev_ncalls and ev_pncalls below. */
55  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
56      if (base->current_event == ev && (ev->ev_events & EV_SIGNAL)
57          && !EVBASE_IN_THREAD(base)) {
58          ++base->current_event_waiters;
59          EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lock);
60      }
61  #endif
62
63      if ((ev->ev_events & (EV_READ|EV_WRITE|EV_SIGNAL)) &&
64          !(ev->ev_flags & (EVLIST_INSERTED|EVLIST_ACTIVE))) {
65          if (ev->ev_events & (EV_READ|EV_WRITE))
66              res = evmap_io_add(base, ev->ev_fd, ev);
67          else if (ev->ev_events & EV_SIGNAL)
68              res = evmap_signal_add(base, (int)ev->ev_fd, ev);
69          if (res != -1)
```

```
70              event_queue_insert(base, ev, EVLIST_INSERTED);
71          if (res == 1) {
72              /* evmap says we need to notify the main thread. */
73              notify = 1;
74              res = 0;
75          }
76      }
77
78      /*
79       * we should change the timeout state only if the previous event
80       * addition succeeded.
81       */
82      if (res != -1 && tv != NULL) {
83          struct timeval now;
84          int common_timeout;
85
86          /*
87           * for persistent timeout events, we remember the
88           * timeout value and re-add the event.
89           *
90           * If tv_is_absolute, this was already set.
91           */
92          if (ev->ev_closure == EV_CLOSURE_PERSIST && !tv_is_absolute)
93              ev->ev_io_timeout = *tv;
94
95          /*
96           * we already reserved memory above for the case where we
97           * are not replacing an existing timeout.
98           */
99          if (ev->ev_flags & EVLIST_TIMEOUT) {
100             /* XXX I believe this is needless. */
101             if (min_heap_elt_is_top(ev))
102                 notify = 1;
103             event_queue_remove(base, ev, EVLIST_TIMEOUT);
104         }
105
106         /* Check if it is active due to a timeout.  Rescheduling
107          * this timeout before the callback can be executed
108          * removes it from the active list. */
109         if ((ev->ev_flags & EVLIST_ACTIVE) &&
110             (ev->ev_res & EV_TIMEOUT)) {
111             if (ev->ev_events & EV_SIGNAL) {
112                 /* See if we are just active executing
113                  * this event in a loop
114                  */
115                 if (ev->ev_ncalls && ev->ev_pncalls) {
116                     /* Abort loop */
117                     *ev->ev_pncalls = 0;
118                 }
119             }
120
121             event_queue_remove(base, ev, EVLIST_ACTIVE);
122         }
123
124         gettime(base, &now);
125
126         common_timeout = is_common_timeout(tv, base);
127         if (tv_is_absolute) {
128             ev->ev_timeout = *tv;
129         } else if (common_timeout) {
130             struct timeval tmp = *tv;
131             tmp.tv_usec &= MICROSECONDS_MASK;
132             evutil_timeradd(&now, &tmp, &ev->ev_timeout);
```

```
133                 ev->ev_timeout.tv_usec |=
134                     (tv->tv_usec & ~MICROSECONDS_MASK);
135             } else {
136                 evutil_timeradd(&now, tv, &ev->ev_timeout);
137             }
138 
139             event_debug((
140                  "event_add: timeout in %d seconds, call %p",
141                  (int)tv->tv_sec, ev->ev_callback));
142 
143             event_queue_insert(base, ev, EVLIST_TIMEOUT);
144             if (common_timeout) {
145                 struct common_timeout_list *ctl =
146                     get_common_timeout_list(base, &ev->ev_timeout);
147                 if (ev == TAILQ_FIRST(&ctl->events)) {
148                     common_timeout_schedule(ctl, &now, ev);
149                 }
150             } else {
151                 /* See if the earliest timeout is now earlier than it
152                  * was before: if so, we will need to tell the main
153                  * thread to wake up earlier than it would
154                  * otherwise. */
155                 if (min_heap_elt_is_top(ev))
156                     notify = 1;
157             }
158         }
159 
160         /* if we are not in the right thread, we need to wake up the loop */
161         if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
162             evthread_notify_base(base);
163 
164         _event_debug_note_add(ev);
165 
166         return (res);
167     }
```

event_add 函数只是对 event_base 加了锁，然后调用 event_add_internal 函数完成工作。所以函数 event_add 是线程安全的。event_add_internal 函数会调用 evmap_io_add 和 evmap_signal_add，把有相同文件描述符 fd 和信号值 sig 的 event 连在一个队列里面。成功之后，就会调用 event_queue_insert，向 event_base 注册事件。

【evmap_io_add 和 evmap_signal_add 函数】把要监听的 fd 或者 sig 添加到多路 IO 复用函数中，使得其是可以监听的。

```
1   //evmap.c文件
2   int evmap_io_add(struct event_base *base, evutil_socket_t fd, struct event *ev) {
3       const struct eventop *evsel = base->evsel;
4       struct event_io_map *io = &base->io;
5       struct evmap_io *ctx = NULL;
6       int nread, nwrite, retval = 0;
7       short res = 0, old = 0;
8       struct event *old_ev;
9 
10      ……
11 
12      //GET_IO_SLOT_AND_CTOR宏的作用就是让ctx指向struct event_map_entry结构体中的
            TAILQ_HEAD
13      //宏的展开，可以到http://blog.csdn.net/luotuo44/article/details/38403241查看
14      GET_IO_SLOT_AND_CTOR(ctx, io, fd, evmap_io, evmap_io_init,
15                           evsel->fdinfo_len);
16 
17      //同一个fd可以调用event_new,event_add
18      //多次。nread、nwrite就是记录有多少次。如果每次event_new的回调函数
19      //都不一样，那么当fd有可读或者可写时，这些回调函数都是会触发的。
```

```
    //对一个fd不能event_new、event_add太多次的。后面会进行判断
    nread = ctx->nread;
    nwrite = ctx->nwrite;

    if (nread)
        old |= EV_READ;
    if (nwrite)
        old |= EV_WRITE;

    if (ev->ev_events & EV_READ) {
        //记录是不是第一次。如果是第一次，那么就说明该fd还没被
        //加入到多路IO复用中。即还没被加入到像select、epoll这些
        //函数中。那么就要加入。这个在后面可以看到。
        if (++nread == 1)
            res |= EV_READ;
    }
    if (ev->ev_events & EV_WRITE) {
        if (++nwrite == 1)
            res |= EV_WRITE;
    }
    if (EVUTIL_UNLIKELY(nread > 0xffff || nwrite > 0xffff)) {
        event_warnx("Too many events reading or writing on fd %d",
                    (int)fd);
        return -1;
    }

    //把fd加入到多路IO复用中。
    if (res) {
        void *extra = ((char*)ctx) + sizeof(struct evmap_io);
        if (evsel->add(base, ev->ev_fd,
                       old, (ev->ev_events & EV_ET) | res, extra) == -1)
            return (-1);
        retval = 1;
    }

    //nread进行了++。把次数记录下来。下次对于同一个fd，这个次数就有用了
    ctx->nread = (ev_uint16_t) nread;
    ctx->nwrite = (ev_uint16_t) nwrite;

    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_io_next);

    return (retval);
}
```

代码中有两个计数 nread 和 nwrite，当其值为 1 时，就说明是第一次监听对应的事件。此时，就要把这个 fd 添加到多路 IO 复用函数中。这就完成 fd 与 select、poll、epoll 之类的多路 IO 复用函数的相关联。这完成对 fd 监听的第一步。

```
//
static void event_queue_insert(struct event_base *base, struct event *ev, int queue)
{
    EVENT_BASE_ASSERT_LOCKED(base);

    if (ev->ev_flags & queue) {
        /* Double insertion is possible for active events */
        if (queue & EVLIST_ACTIVE)
            return;

        event_errx(1, "%s: %p(fd "EV_SOCK_FMT") already on queue %x", __func__,
            ev, EV_SOCK_ARG(ev->ev_fd), queue);
        return;
    }

```

```
16      if (~ev->ev_flags & EVLIST_INTERNAL)
17          base->event_count++;
18
19      ev->ev_flags |= queue;
20      switch (queue) {
21      case EVLIST_INSERTED:
22          TAILQ_INSERT_TAIL(&base->eventqueue, ev, ev_next);
23          break;
24      case EVLIST_ACTIVE:
25          base->event_count_active++;
26          TAILQ_INSERT_TAIL(&base->activequeues[ev->ev_pri],
27              ev,ev_active_next);
28          break;
29      case EVLIST_TIMEOUT: {
30          if (is_common_timeout(&ev->ev_timeout, base)) {
31              struct common_timeout_list *ctl =
32                  get_common_timeout_list(base, &ev->ev_timeout);
33              insert_common_timeout_inorder(ctl, ev);
34          } else
35              min_heap_push(&base->timeheap, ev);
36          break;
37      }
38      default:
39          event_errx(1, "%s: unknown queue %x", __func__, queue);
40      }
41  }
```

这个函数的主要作为是把 event 加入到对应的队列中。在这里，是为了把 event 加入到 eventqueue 这个已注册队列中，即将 event 向 event_base 注册。注意，此时 event 结构体的 ev_flags 变量为 EVLIST_INIT | EVLIST_INSERTED 了。

5.5 监听 event

现在事件已经添加完毕，开始进入主循环 event_base_dispatch 函数。还是同样，该函数内部调用 event_base_loop 完成工作。

```
1   int event_base_dispatch(struct event_base *event_base) {
2       return (event_base_loop(event_base, 0));
3   }
4
5   int event_base_loop(struct event_base *base, int flags) {
6       const struct eventop *evsel = base->evsel;
7       struct timeval tv;
8       struct timeval *tv_p;
9       int res, done, retval = 0;
10
11      /* Grab the lock.  We will release it inside evsel.dispatch, and again
12       * as we invoke user callbacks. */
13      EVBASE_ACQUIRE_LOCK(base, th_base_lock);
14
15      if (base->running_loop) {
16          event_warnx("%s: reentrant invocation.  Only one event_base_loop"
17              " can run on each event_base at once.", __func__);
18          EVBASE_RELEASE_LOCK(base, th_base_lock);
19          return -1;
20      }
21
22      base->running_loop = 1;
23
24      clear_time_cache(base);
25
26      if (base->sig.ev_signal_added && base->sig.ev_n_signals_added)
27          evsig_set_base(base);
```

```
28
29      done = 0;
30
31  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
32      base->th_owner_id = EVTHREAD_GET_ID();
33  #endif
34
35      base->event_gotterm = base->event_break = 0;
36
37      while (!done) {
38          base->event_continue = 0;
39
40          /* Terminate the loop if we have been asked to */
41          if (base->event_gotterm) {
42              break;
43          }
44
45          if (base->event_break) {
46              break;
47          }
48
49          timeout_correct(base, &tv);
50
51          tv_p = &tv;
52          if (!N_ACTIVE_CALLBACKS(base) && !(flags & EVLOOP_NONBLOCK)) {
53              timeout_next(base, &tv_p);
54          } else {
55              /*
56               * if we have active events, we just poll new events
57               * without waiting.
58               */
59              evutil_timerclear(&tv);
60          }
61
62          /* If we have no events, we just exit */
63          if (!event_haveevents(base) && !N_ACTIVE_CALLBACKS(base)) {
64              event_debug(("%s: no events registered.", __func__));
65              retval = 1;
66              goto done;
67          }
68
69          /* update last old time */
70          gettime(base, &base->event_tv);
71
72          clear_time_cache(base);
73
74          res = evsel->dispatch(base, tv_p);
75
76          if (res == -1) {
77              event_debug(("%s: dispatch returned unsuccessfully.",
78                  __func__));
79              retval = -1;
80              goto done;
81          }
82
83          update_time_cache(base);
84
85          timeout_process(base);
86
87          if (N_ACTIVE_CALLBACKS(base)) {
88              int n = event_process_active(base);
89              if ((flags & EVLOOP_ONCE)
90                  && N_ACTIVE_CALLBACKS(base) == 0
```

```
                && n != 0)
                done = 1;
        } else if (flags & EVLOOP_NONBLOCK)
            done = 1;
    }
    event_debug(("%s: asked to terminate loop.", __func__));

done:
    clear_time_cache(base);
    base->running_loop = 0;

    EVBASE_RELEASE_LOCK(base, th_base_lock);

    return (retval);
}
```

在 event_base_loop 函数内部会进行加锁，这是因为这里要对 event_base 里面的多个队列进行一些数据操作 (增删操作)，此时要用锁来保护队列不被另外一个线程所破坏。上面代码中有两个函数 evsel->dispatch 和 event_process_active。前一个将调用多路 IO 复用函数，对 event 进行监听，并且把满足条件的 event 放到 event_base 的激活队列中。后一个则遍历这个激活队列的所有 event，逐个调用对应的回调函数。

整个流程如下图所示：

5.6 激活列表

将已激活 event 插入到激活列表，我们还是深入看看 Libevent 是怎么把 event 添加到激活队列的。dispatch 是一个函数指针，它的实现都包含是一个多路 IO 复用函数。这里选择 poll 这个多路 IO 复用函数来作分析。

```
static int
epoll_dispatch(struct event_base *base, struct timeval *tv)
{
    struct epollop *epollop = base->evbase;
    struct epoll_event *events = epollop->events;
    int i, res;
    long timeout = -1;

    if (tv != NULL) {
        timeout = evutil_tv_to_msec(tv);
        if (timeout < 0 || timeout > MAX_EPOLL_TIMEOUT_MSEC) {
            /* Linux kernels can wait forever if the timeout is
             * too big; see comment on MAX_EPOLL_TIMEOUT_MSEC. */
            timeout = MAX_EPOLL_TIMEOUT_MSEC;
        }
    }

    epoll_apply_changes(base);
    event_changelist_remove_all(&base->changelist, base);

    EVBASE_RELEASE_LOCK(base, th_base_lock);

    res = epoll_wait(epollop->epfd, events, epollop->nevents, timeout);

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    if (res == -1) {
        if (errno != EINTR) {
            event_warn("epoll_wait");
            return (-1);
        }

        return (0);
```

```
    }

    event_debug(("%s: epoll_wait reports %d", __func__, res));
    EVUTIL_ASSERT(res <= epollop->nevents);

    for (i = 0; i < res; i++) {
        int what = events[i].events;
        short ev = 0;

        if (what & (EPOLLHUP|EPOLLERR)) {
            ev = EV_READ | EV_WRITE;
        } else {
            if (what & EPOLLIN)
                ev |= EV_READ;
            if (what & EPOLLOUT)
                ev |= EV_WRITE;
        }

        if (!ev)
            continue;

        evmap_io_active(base, events[i].data.fd, ev | EV_ET);
    }

    if (res == epollop->nevents && epollop->nevents < MAX_NEVENT) {
        /* We used all of the event space this time.  We should
           be ready for more events next time. */
        int new_nevents = epollop->nevents * 2;
        struct epoll_event *new_events;

        new_events = mm_realloc(epollop->events,
            new_nevents * sizeof(struct epoll_event));
        if (new_events) {
            epollop->events = new_events;
            epollop->nevents = new_nevents;
        }
    }

    return (0);
}
```

events 数组的数据是在 evmap_io_add 函数中添加的，在 evmap_io_add 函数里面，有一个 evsel->add 调用，它会把数据 (fd 和对应的监听类型) 放到 events 数组中。

当主线程从 epoll 返回时，没有错误的话，就说明有些监听的事件发生了。在函数的后面，它会遍历这个 events 数组，查看哪个 fd 是有事件发生。如果事件发生，就调用 evmap_io_active(base, events[i].data.fd, ev | EV_ET)；在这个函数里面会把这个 fd 对应的 event 放到 event_base 的激活 event 队列中。下面是 evmap_io_active 的代码。

```
void evmap_io_active(struct event_base *base, evutil_socket_t fd, short events) {
    struct event_io_map *io = &base->io;
    struct evmap_io *ctx;
    struct event *ev;

#ifndef EVMAP_USE_HT
    EVUTIL_ASSERT(fd < io->nentries);
#endif
    GET_IO_SLOT(ctx, io, fd, evmap_io);

    EVUTIL_ASSERT(ctx);
    TAILQ_FOREACH(ev, &ctx->events, ev_io_next) {
        if (ev->ev_events & events)
            event_active_nolock(ev, ev->ev_events & events, 1);
    }
```

```
16  }
17
18  void event_active_nolock(struct event *ev, int res, short ncalls) {
19      struct event_base *base;
20
21      event_debug(("event_active: %p (fd "EV_SOCK_FMT"), res %d, callback %p",
22          ev, EV_SOCK_ARG(ev->ev_fd), (int)res, ev->ev_callback));
23
24
25      /* We get different kinds of events, add them together */
26      if (ev->ev_flags & EVLIST_ACTIVE) {
27          ev->ev_res |= res;
28          return;
29      }
30
31      base = ev->ev_base;
32
33      EVENT_BASE_ASSERT_LOCKED(base);
34
35      ev->ev_res = res;
36
37      if (ev->ev_pri < base->event_running_priority)
38          base->event_continue = 1;
39
40      if (ev->ev_events & EV_SIGNAL) {
41  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
42          if (base->current_event == ev && !EVBASE_IN_THREAD(base)) {
43              ++base->current_event_waiters;
44              EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lock);
45          }
46  #endif
47          ev->ev_ncalls = ncalls;
48          ev->ev_pncalls = NULL;
49      }
50
51      event_queue_insert(base, ev, EVLIST_ACTIVE);
52
53      if (EVBASE_NEED_NOTIFY(base))
54          evthread_notify_base(base);
55  }
```

event_queue_insert 将 ev 插入激活队列中，见之前对该函数的解释。经过这三个函数的调用，就可以把一个 fd 对应的所有符合条件的 event 插入到激活队列中。因为 Libevent 还对事件处理设有优先级，所以有一个激活数组队列，而不是只有一个激活队列。注意，此时 event 结构体的 ev_flags 变量为 EVLIST_INIT | EVLIST_INSERTED | EVLIST_ACTIVE 了。

5.7 处理激活事件

现在已经完成了将 event 插入到激活队列中。接下来就是遍历激活数组队列，把所有激活的 event 都处理即可。现在来追踪 event_process_active 函数。

```
1   /*
2    * Active events are stored in priority queues. Lower priorities are always
3    * process before higher priorities . Low priority events can starve high
4    * priority ones.
5    */
6
7   static int event_process_active(struct event_base *base) {
8       /* Caller must hold th_base_lock */
9       struct event_list *activeq = NULL;
10      int i, c = 0;
```

```
11
12      for (i = 0; i < base->nactivequeues; ++i) {
13          if (TAILQ_FIRST(&base->activequeues[i]) != NULL) {
14              base->event_running_priority = i;
15              activeq = &base->activequeues[i];
16              c = event_process_active_single_queue(base, activeq);
17              if (c < 0) {
18                  base->event_running_priority = -1;
19                  return -1;
20              } else if (c > 0)
21                  break; /* Processed a real event; do not
22                       * consider lower-priority events */
23              /* If we get here, all of the events we processed
24               * were internal. Continue. */
25          }
26      }
27
28      event_process_deferred_callbacks(&base->defer_queue,&base->event_break);
29      base->event_running_priority = -1;
30      return c;
31  }
32
33  /*
34    Helper for event_process_active to process all the events in a single queue,
35    releasing the lock as we go. This function requires that the lock be held
36    when it's invoked. Returns -1 if we get a signal or an event_break that
37    means we should stop processing any active events now. Otherwise returns
38    the number of non-internal events that we processed.
39  */
40  static int event_process_active_single_queue(struct event_base *base, struct event_list *activeq) {
41      struct event *ev;
42      int count = 0;
43
44      EVUTIL_ASSERT(activeq != NULL);
45
46      for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {
47          if (ev->ev_events & EV_PERSIST)
48              event_queue_remove(base, ev, EVLIST_ACTIVE);
49          else
50              event_del_internal(ev);
51          if (!(ev->ev_flags & EVLIST_INTERNAL))
52              ++count;
53
54          event_debug((
55               "event_process_active: event: %p, %s%scall %p",
56              ev,
57              ev->ev_res & EV_READ ? "EV_READ " : " ",
58              ev->ev_res & EV_WRITE ? "EV_WRITE " : " ",
59              ev->ev_callback));
60
61  #ifndef _EVENT_DISABLE_THREAD_SUPPORT
62          base->current_event = ev;
63          base->current_event_waiters = 0;
64  #endif
65
66          switch (ev->ev_closure) {
67          case EV_CLOSURE_SIGNAL:
68              event_signal_closure(base, ev);
69              break;
70          case EV_CLOSURE_PERSIST:
71              event_persist_closure(base, ev);
72              break;
73          default:
```

```
74              case EV_CLOSURE_NONE:
75                  EVBASE_RELEASE_LOCK(base, th_base_lock);
76                  (*ev->ev_callback)(
77                      ev->ev_fd, ev->ev_res, ev->ev_arg);
78                  break;
79              }
80
81              EVBASE_ACQUIRE_LOCK(base, th_base_lock);
82      #ifndef _EVENT_DISABLE_THREAD_SUPPORT
83              base->current_event = NULL;
84              if (base->current_event_waiters) {
85                  base->current_event_waiters = 0;
86                  EVTHREAD_COND_BROADCAST(base->current_event_cond);
87              }
88      #endif
89
90              if (base->event_break)
91                  return -1;
92              if (base->event_continue)
93                  break;
94          }
95          return count;
96      }
```

上面的代码，从高到低优先级遍历激活 event 优先级数组。对于激活的 event，要调用 event_queue_remove 将之从激活队列中删除掉。然后再对这个 event 调用其回调函数。event_queue_remove 函数的调用会改变 event 结构体的 ev_flags 变量的值。调用后，ev_flags 变量为 EVLIST_INIT | EVLIST_INSERTED。现在又可以等待下一次事件的到来了。

6 Timer

6.1 超时处理

如何成为超时 event：Libevent 允许创建一个超时 event，使用 evtimer_new 宏。

```
#define evtimer_assign(ev, b, cb, arg)  event_assign((ev), (b), -1, 0, (cb), (arg))
#define evtimer_new(b, cb, arg)     event_new((b), -1, 0, (cb), (arg))
#define evtimer_add(ev, tv)         event_add((ev), (tv))
#define evtimer_del(ev)             event_del(ev)
#define evtimer_pending(ev, tv)         event_pending((ev), EV_TIMEOUT, (tv))
#define  evtimer_initialized (ev)        event_initialized (ev)
```

从宏的实现来看，它一样是用到了一般的 event_new，并且不使用任何的文件描述符，另外还有其他的一些超时事件的处理宏，都是用的与之对应的 event 处理函数。从超时 event 宏的实现来看，无论是 evtimer 创建的 event 还是一般 event_new 创建的 event，都能使得 Libevent 进行超时监听。其实，使得 Libevent 对一个 event 进行超时监听的原因是：在调用 event_add 的时候，第二参数不能为 NULL，要设置一个超时值。如果为 NULL，那么 Libevent 将不会为这个 event 监听超时。下文统一称设置了超时值的 event 为超时 event。

超时 event 的原理：Libevent 对超时进行监听的原理不同于之前讲到的对信号的监听，Libevent 对超时的监听的原理是，多路 IO 复用函数都是有一个超时值。如果用户需要 Libevent 同时监听多个超时 event，那么 Libevent 就把超时值最小的那个作为多路 IO 复用函数的超时值。自然，当时间一到，就会从多路 IO 复用函数返回。此时对超时 event 进行处理即可。Libevent 运行用户同时监听多个超时 event，那么就必须要对这个超时值进行管理。Libevent 提供了小根堆和通用超时 (common timeout) 这两种管理方式。下文为了叙述方便，就假定使用的是小根堆。工作流程：下面来看一下超时 event 的工作流程。

设置超时值：首先调用 event_add 时要设置一个超时值，这样才能成为一个超时 event。

```
//event.c文件
//在event_add中，会把第三个参数设为0.使得使用的是相对时间
static inline int event_add_internal(struct event *ev, const struct timeval *tv, int tv_is_absolute) {
    struct event_base *base = ev->ev_base;
    int res = 0;
    int notify = 0;

    //tv不为NULL,就说明是一个超时event,在小根堆中为其留一个位置
    if (tv != NULL && !(ev->ev_flags & EVLIST_TIMEOUT)) {
        if (min_heap_reserve(&base->timeheap,
            1 + min_heap_size(&base->timeheap)) == -1)
            return (-1);  /* ENOMEM == errno */
    }

    ... //将IO或者信号event插入到对应的队列中。

    if (res != -1 && tv != NULL) {
        struct timeval now;

        //用户把这个event设置成EV_PERSIST。即永久event.
        //如果没有这样设置的话，那么只会超时一次。设置了，那么就
        //可以超时多次。那么就要记录用户设置的超时值。
        if (ev->ev_closure == EV_CLOSURE_PERSIST && !tv_is_absolute)
            ev->ev_io_timeout = *tv;

        //该event之前被加入到超时队列。用户可以对同一个event调用多次event_add
        //并且可以每次都用不同的超时值。
        if (ev->ev_flags & EVLIST_TIMEOUT) {
            /* XXX I believe this is needless. */
```

```
30              //之前为该event设置的超时值是所有超时中最小的。
31              //从下面的删除可知，会删除这个最小的超时值。此时多路IO复用函数
32              //的超时值参数就已经改变了。
33              if (min_heap_elt_is_top(ev))
34                  notify = 1; //要通知主线程。可能是次线程为这个event调用本函数
35
36              //从超时队列中删除这个event。因为下次会再次加入。
37              //多次对同一个超时event调用event_add,那么只能保留最后的那个。
38              event_queue_remove(base, ev, EVLIST_TIMEOUT);
39          }
40
41          //因为可以在次线程调用event_add。而主线程刚好在执行event_base_dispatch
42          if ((ev->ev_flags & EVLIST_ACTIVE) &&
43              (ev->ev_res & EV_TIMEOUT)) {//该event被激活的原因是超时
44
45              ...
46              event_queue_remove(base, ev, EVLIST_ACTIVE);
47          }
48
49          //获取现在的时间
50          gettime(base, &now);
51
52          //虽然用户在event_add时只需用一个相对时间，但实际上在Libevent内部
53          //还是要把这个时间转换成绝对时间。从存储的角度来说，存绝对时间只需
54          //一个变量。而相对时间则需两个，一个存相对值，另一个存参照物。
55          if (tv_is_absolute) { //该参数指明时间是否为一个绝对时间
56              ev->ev_timeout = *tv;
57          } else {
58              //参照时间 + 相对时间 ev_timeout存的是绝对时间
59              evutil_timeradd(&now, tv, &ev->ev_timeout);
60          }
61
62          //将该超时event插入到超时队列中
63          event_queue_insert(base, ev, EVLIST_TIMEOUT);
64
65          //本次插入的超时值，是所有超时中最小的。那么此时就需要通知主线程。
66          if (min_heap_elt_is_top(ev))
67              notify = 1;
68      }
69
70      //如果代码逻辑中是需要通知的。并且本线程不是主线程。那么就通知主线程
71      if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
72          evthread_notify_base(base);
73
74      return (res);
75  }
```

对于同一个 event，如果是 IO event 或者信号 event，那么将无法多次添加。但如果是一个超时 event，那么是可以多次添加的。并且对应超时值会使用最后添加时指明的那个，之前的统统不要，即替换掉之前的超时值。

代码中出现了多次使用了 notify 变量。这主要是用在：次线程在执行这个函数，而主线程在执行 event_base_dispatch。前面说到 Libevent 能对超时 event 进行监听的原理是：多路 IO 复用函数有一个超时参数。在次线程添加的 event 的超时值更小，又或者替换了之前最小的超时值。在这种情况下，都是要通知主线程，告诉主线程，最小超时值已经变了。关于通知主线程 evthread_notify_base，【evthread_notify_base 通知主线程】

代码中的第三个判断体中用到了 ev->ev_io_timeout。但 event 结构体中并没有该变量。其实，ev_io_timeout 是一个宏定义。

```
1  //event-internal.h文件
2  #define ev_io_timeout   _ev.ev_io.ev_timeout
```

要注意的一点是,在调用 event_add 时设定的超时值是一个时间段(可以认为隔多长时间就触发一次),相对于现在,即调用 event_add 的时间,而不是调用 event_base_dispatch 的时间。

调用多路 IO 复用函数等待超时:现在来看一下 event_base_loop 函数,看其是怎么处理超时 event 的。

```
//event.c文件
int
event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;
    struct timeval tv;
    struct timeval *tv_p;
    int res, done, retval = 0;

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    base->running_loop = 1;

    done = 0;
    while (!done) {
        tv_p = &tv;
        if (!N_ACTIVE_CALLBACKS(base) && !(flags & EVLOOP_NONBLOCK)) {
            // 根据Timer事件计算evsel->dispatch的最大等待时间(超时值最小)
            timeout_next(base, &tv_p);
        } else { //不进行等待
            //把等待时间置为0,即可不进行等待,马上触发事件
            evutil_timerclear(&tv);
        }

        res = evsel->dispatch(base, tv_p);

        //处理超时事件,将超时事件插入到激活链表中
        timeout_process(base);

        if (N_ACTIVE_CALLBACKS(base)) {
            int n = event_process_active(base);
        }
    }

done:
    base->running_loop = 0;
    EVBASE_RELEASE_LOCK(base, th_base_lock);

    return (retval);
}

//选出超时值最小的那个
static int timeout_next(struct event_base *base, struct timeval **tv_p)
{
    /* Caller must hold th_base_lock */
    struct timeval now;
    struct event *ev;
    struct timeval *tv = *tv_p;
    int res = 0;

    // 堆的首元素具有最小的超时值,这个是小根堆的性质。
    ev = min_heap_top(&base->timeheap);

    //堆中没有元素
    if (ev == NULL) {
        *tv_p = NULL;
        goto out;
```

```
58      }
59
60      //获取当然时间
61      if (gettime(base, &now) == -1) {
62          res = -1;
63          goto out;
64      }
65
66      // 如果超时时间<=当前时间，不能等待，需要立即返回
67      // 因为ev_timeout这个时间是由event_add调用时的绝对时间 + 相对时间。所以ev_timeout是
68      // 绝对时间。可能在调用event_add之后，过了一段时间才调用event_base_diapatch,所以
69      // 现在可能都过了用户设置的超时时间。
70      if (evutil_timercmp(&ev->ev_timeout, &now, <=)) {
71          evutil_timerclear(tv); //清零，这样可以让dispatcht不会等待，马上返回
72          goto out;
73      }
74
75      // 计算等待的时间=当前时间-最小的超时时间
76      evutil_timersub(&ev->ev_timeout, &now, tv);
77
78      EVUTIL_ASSERT(tv->tv_sec >= 0);
79      EVUTIL_ASSERT(tv->tv_usec >= 0);
80      event_debug(("timeout_next: in %d seconds", (int)tv->tv_sec));
81
82  out:
83      return (res);
84  }
```

上面代码的流程是：计算出本次调用多路 IO 复用函数的等待时间，然后调用多路 IO 复用函数中等待超时。

激活超了时的 event：上面代码中的 timeout_process 函数就是处理超了时的 event。

```
1   /* Activate every event whose timeout has elapsed. */
2   //把超时了的event，放到激活队列中。并且，其激活原因设置为EV_TIMEOUT
3   static void timeout_process(struct event_base *base) {
4       /* Caller must hold lock. */
5       struct timeval now;
6       struct event *ev;
7
8       if (min_heap_empty(&base->timeheap)) {
9           return;
10      }
11
12      gettime(base, &now);
13
14      //遍历小根堆的元素。之所以不是只取堆顶那一个元素，是因为当主线程调用多路IO复用函数
15      //进入等待时，次线程可能添加了多个超时值更小的event
16      while ((ev = min_heap_top(&base->timeheap))) {
17          //ev->ev_timeout存的是绝对时间
18          //超时时间比此刻时间大，说明该event还没超时。那么余下的小根堆元素更不用检查了。
19          if (evutil_timercmp(&ev->ev_timeout, &now, >))
20              break;
21
22          /* delete this event from the I/O queues */
23          //下面说到的del是等同于调用event_del.把event从这个event_base中(所有的队列都)
24          //删除。event_base不再监听之。
25          //这里是timeout_process函数。所以对于有超时的event，才会被del掉。
26          //对于有EV_PERSIST选项的event，在处理激活event的时候，会再次添加进event_base的。
27          //这样做的一个好处就是，再次添加的时候，又可以重新计算该event的超时时间(绝对时间)。
28          event_del_internal(ev);
29
30          event_debug(("timeout_process: call %p",ev->ev_callback));
31          //把这个event加入到event_base的激活队列中。
```

```
        //event_base的激活队列又有该event了。所以如果该event是EV_PERSIST的，是可以
        //再次添加进该event_base的
        event_active_nolock(ev, EV_TIMEOUT, 1);
    }
}
```

当从多路 IO 复用函数返回时，就检查时间小根堆，看有多少个 event 已经超时了。如果超时了，那就把这个 event 加入到 event_base 的激活队列中。并且把这个超时 del(删除) 掉，这主要是用于非 PERSIST 超时 event 的。删除一个 event 的具体操作可以查看这里。把一个 event 添加进激活队列后的工作流程可以参考【Libevent 工作流程探究】

处理永久超时 event：现在来看一下如果该超时 event 有 EV_PERSIST 选项，在后面是怎么再次添加进 event_base，因为前面的代码注释中已经说了，在选出超时 event 时，会把超时的 event 从 event_base 中 delete 掉。

```
//event.c文件
int
event_assign(struct event *ev, struct event_base *base, evutil_socket_t fd,
                    short events, void (*callback)(evutil_socket_t, short, void *), void *arg)
{
    ...
    if (events & EV_PERSIST) {
        ev->ev_closure = EV_CLOSURE_PERSIST;
    } else {
        ev->ev_closure = EV_CLOSURE_NONE;
    }

    return 0;
}

static int
event_process_active_single_queue(struct event_base *base,
    struct event_list *activeq)
{
    struct event *ev;

    //遍历同一优先级的所有event
    for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {

        //下面这个if else 是用于IO event的。这里贴出，是为了了解一些非超时event是
        //怎么处理永久事件(EV_PERSIST)的。
        //如果是永久事件，那么只需从active队列中删除。
        if (ev->ev_events & EV_PERSIST)
            event_queue_remove(base, ev, EVLIST_ACTIVE);
        else //不是的话，那么就要把这个event删除掉。
            event_del_internal(ev);


        switch (ev->ev_closure) {
        //这个case只对超时event的EV_PERSIST才有用。IO的没有用
        case EV_CLOSURE_PERSIST:
            event_persist_closure(base, ev);
            break;

        default: //默认是EV_CLOSURE_NONE
        case EV_CLOSURE_NONE:
            //没有设置EV_PERSIST的超时event，就只有一次的监听机会
            (*ev->ev_callback)(
                ev->ev_fd, ev->ev_res, ev->ev_arg);
            break;
        }
    }
}
```

```
49
50  static inline void
51  event_persist_closure(struct event_base *base, struct event *ev)
52  {
53      //在event_add_internal函数中，如果是超时event并且有EV_PERSIST，那么就会把ev_io_timeout
            设置成
54      //用户设置的超时时间(相对时间)。否则为0。即不进入判断体中。
55      //说明这个if只用于处理具有EV_PERSIST属性的超时event
56      if (ev->ev_io_timeout.tv_sec || ev->ev_io_timeout.tv_usec) {
57          struct timeval run_at, relative_to, delay, now;
58          ev_uint32_t usec_mask = 0;
59
60          gettime(base, &now);
61
62          //delay是用户设置的超时时间。event_add的第二个参数
63          delay = ev->ev_io_timeout;
64          //是因为超时才执行到这里，event可以同时监听多种事件。如果是由于可读而执行
65          //到这里，那么就说明还没超时。
66          if (ev->ev_res & EV_TIMEOUT) { //如果是因为超时而激活，那么下次超时就是本次超时的
67              relative_to = ev->ev_timeout; // 加上 delay 时间。
68          } else {
69              relative_to = now; //重新计算超时值
70          }
71
72          evutil_timeradd(&relative_to, &delay, &run_at);
73          //无论relative是哪个时间，run_at都不应该小于now。
74          //如果小于，则说明是用户手动修改了系统时间，使得gettime()函数获取了一个
75          //之前的时间。比如现在是9点，用户手动调回到7点。
76          if (evutil_timercmp(&run_at, &now, <)) {
77              //那么就以新的系统时间为准
78              evutil_timeradd(&now, &delay, &run_at);
79          }
80
81          //把这个event再次添加到event_base中。注意，此时第三个参数为1，说明是一个绝对时间
82          event_add_internal(ev, &run_at, 1);
83      }
84      EVBASE_RELEASE_LOCK(base, th_base_lock);
85      (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);//执行回调函数
86  }
```

这段代码的处理流程是：如果用户指定了 EV_PERSIST，那么在 event_assign 中就记录下来。在 event_process_active_single_queue 函数中会针对永久 event 进行调用 event_persist_closure 函数对之进行处理。在 event_persist_closure 函数中，如果是一般的永久 event，那么就直接调用该 event 的回调函数。如果是超时永久 event，那么就需要再次计算新的超时时间，并将这个 event 再次插入到 event_base 中。这段代码也指明了，如果一个 event 因可读而被激活，那么其超时时间就要重新计算。而不是之前的那个了。也就是说，如果一个 event 设置了 3 秒的超时，但 1 秒后就可读了，那么下一个超时值，就要重新计算设置，而不是 2 秒后。

从前面的源码分析也可以得到：如果一个 event 监听可读的同时也设置了超时值，并且一直没有数据可读，最后超时了，那么这个 event 将会被删除掉，不会再等。

6.2 时间管理

6.3 超时管理

前面的内容已经说到，如果要对多个超时 event 同时进行监听，就要对这些超时 event 进行集中管理，能够方便地 (时间复杂度小) 获取、加入、删除一个 event。在之前的 Libevent 版本，Libevent 使用小根堆管理这些超时 event。小根堆的插入和删除时间复杂度都是 O(logN)。在 2.0.4-alpha 版本时，Libevent 引入了一个叫 common-timeout 的东西来管理超时 event，要注意的是，它并不是替换小根堆，而是和小根堆配合使用

的。事实上，common-timeout 的实现要用到小根堆。

Libevent 的小根堆和数据结构教科书上的小根堆几乎是一样的。看一下数据结构和 Libevent 的小根堆源码，很容易就懂的。这样就不多讲了。本文主要讲一下 common-timeout。

common-timeout 的用途：要讲解 common-timeout，得先说明它的用途。前面说到它和小根堆是配合使用的。小根堆是用在：多个超时 event 的超时时长是随机的。而 common-timeout 则是用在：大量的超时 event 具有相同的超时时长。其中，超时时长是指 event_add 参数的第二个参数。要注意的是，这些大量超时 event 虽然有相同的超时时长，但它们的超时时间是不同的。因为超时时间 = 超时时长 + 调用 event_add 时间。毫无疑问，如果有相同超时时长的大量超时 event 放到小根堆上，那么效率比较低的。虽然小根堆的插入和删除的时间复杂度都是 O(logN)，但是如果有大量的 N，效率也是会下降很多。

common-timeout 的原理：common-timeout 的思想是，既然有大量的超时 event 具有相同的超时时长，那么就它们必定依次激活。如果把它们按照超时时间升序地放到一个队列中 (在 Libevent 中就是这样做的)，那么每次只需检查队列的第一个超时 event 即可。因为其他超时 event 肯定在第一个超时之后才超时的。前面说到 common-timeout 和小根堆是配合使用的。当从 common-timeout 中选出最早超时的那个 event 后，就把之插入到小根堆中。这样就可以通过小根堆完成超时监控了。具体的处理过程可以参考【超时 event 的处理】。通过这样处理后，就不用把大量的超时 event 都插入到小根堆中。

下面看一下 Libevent 的具体实现吧。

```
1  //event-internal.h文件
2  struct event_base {
3      ……
4      //因为可以有多个不同时长的超时event组。故得是数组
5      //因为数组元素是common_timeout_list指针，所以得是二级指针
6      struct common_timeout_list **common_timeout_queues;
7      //数组元素个数
8      int n_common_timeouts;
9      //已分配的数组元素个数
10     int n_common_timeouts_allocated;
11     ……
12 };
13
14 /* A list of events waiting on a given 'common' timeout value. Ordinarily,
15  * events waiting for a timeout wait on a minheap. Sometimes, however, a
16  * queue can be faster.
17  **/
18 struct common_timeout_list {
19     /* List of events currently waiting in the queue. */
20     //超时event队列。将所有具有相同超时时长的超时event放到一个队列里面
21     struct event_list events;
22     /* 'magic' timeval used to indicate the duration of events in this
23      * queue. 超时时长*/
24     struct timeval duration;
25     /* Event that triggers whenever one of the events in the queue is
26      * ready to activate 具有相同超时时长的超时event代表*/
27     struct event timeout_event;
28     /* The event_base that this timeout list is part of */
29     struct event_base *base;
30 };
```

在实际应用时，可能超时时长为 10 秒的有 1k 个超时 event，时长为 20 秒的也有 1k 个，这就需要一个数组。数组的每一个元素是 common_timeout_list 结构体指针。每一个 common_timeout_list 结构体就会处理所有具有相同超时时长的超时 event。common_timeout_list 结构体里面有一个 event 结构体成员，所以并不是从多个具有相同超时时长的超时 event 中选择一个作为代表，而是在内部有一个 event。common_timeout_list 是使用 struct event_list 结构体队列来管理 event，它是一种 TAILQ_QUEUE 队列，可

以参考最后一章【TAILQ_QUEUE 队列】

使用 common-timeout：现在来看看怎么使用 common-timeout。从上面的代码可以想到，如果要使用 common-timeout，就必须把超时 event 插入到 common_timeout_list 的 events 队列中。又因为其要求具有相同的超时时长，所以要插入的超时 event 要和某个 common_timeout_list 结构体有相同的超时时长。所以，我们还是来看一下怎么设置 common_timeout_list 结构体的超时时长。实际上，并不是设置。而是向 event_base 申请一个具有特定时长的 common_timeout_list。每申请一个，就会在 common_timeout_queues 数组中加入一个 common_timeout_list 元素。可以通过 event_base_init_common_timeout 申请。申请后，就可以直接调用 event_add 把超时 event 插入到 common-timeout 中。但问题是，common-timeout 和小根堆是共存的，event_add 又没有第三个参数作为说明，要插入到 common-timeout 还是小根堆。

common-timeout 标志：其实，event_add 是根据第二个参数，即超时时长值进行区分的。首先有一个基本事实，对一个 struct timeval 结构体, 成员 tv_usec 的单位是微秒，所以最大也就是 999999, 只需低 20 比特位就能存储了。但成员 tv_usec 的类型是 int 或者 long，肯定有 32 比特位。所以，就有高 12 比特位是空闲的。Libevent 就是利用那空闲的 12 个比特位做文章的。这 12 比特位是高比特位。Libevent 使用最高的 4 比特位作为标志位，标志它是一个专门用于 common-timeout 的时间，下文将这个标志称为 common-timeout 标志。次 8 比特位用来记录该超时时长在 common_timeout_queues 数组中的位置，即下标值。这也限制了 common_timeout_queues 数组的长度，最大为 2 的 8 次方，即 256。

为了方便地处理这些比特位，Libevent 定义了下面这些宏定义和一个判断函数。

```
1  /* Common timeouts are special timeouts that are handled as queues rather than
2   * in the minheap. This is more efficient than the minheap if we happen to
3   * know that we're going to get several thousands of timeout events all with
4   * the same timeout value.
5   *
6   * Since all our timeout handling code assumes timevals can be copied,
7   * assigned, etc, we can't use "magic pointer" to encode these common
8   * timeouts. Searching through a list to see if every timeout is common could
9   * also get inefficient . Instead, we take advantage of the fact that tv_usec
10  * is 32 bits long, but only uses 20 of those bits (since it can never be over
11  * 999999.) We use the top bits to encode 4 bites of magic number, and 8 bits
12  * of index into the event_base's aray of common timeouts.
13  */
14
15 #define MICROSECONDS_MASK       COMMON_TIMEOUT_MICROSECONDS_MASK
16 #define COMMON_TIMEOUT_IDX_MASK 0x0ff00000
17 #define COMMON_TIMEOUT_IDX_SHIFT 20
18 #define COMMON_TIMEOUT_MASK     0xf0000000
19 #define COMMON_TIMEOUT_MAGIC    0x50000000
20
21 #define COMMON_TIMEOUT_IDX(tv) \
22     (((tv)->tv_usec & COMMON_TIMEOUT_IDX_MASK)>>COMMON_TIMEOUT_IDX_SHIFT)
23
24 /** Return true iff if 'tv' is a common timeout in 'base' */
25 static inline int
26 is_common_timeout(const struct timeval *tv,
27     const struct event_base *base)
28 {
29     int idx;
30     //不具有common-timeout标志位，那么就肯定不是commont-timeout时间了
31     if ((tv->tv_usec & COMMON_TIMEOUT_MASK) != COMMON_TIMEOUT_MAGIC)
32         return 0;
33     idx = COMMON_TIMEOUT_IDX(tv);     //获取数组下标
34     return idx < base->n_common_timeouts;
35 }
```

代码最后面的那个判断函数，是用来判断一个给定的 struct timeval 时间，是否为

common-timeout 时间。在 event_add_internal 函数中会用之作为判断，然后根据判断结果来决定是插入小根堆还是 common-timeout，这也就完成了区分。申请并得到特定时长的 common-timeout：那么怎么得到一个具有 common-timeout 标志的时间呢？其实，还是通过前面说到的 event_base_init_common_timeout 函数。该函数将返回一个具有 common-timeout 标志的时间。

```
//event.c文件
//申请一个时长为duration的common_timeout_list
const struct timeval *
event_base_init_common_timeout(struct event_base *base,
    const struct timeval *duration)
{
    int i;
    struct timeval tv;
    const struct timeval * result=NULL;
    struct common_timeout_list *new_ctl;

    //这个时间的微秒位应该进位。用户没有将之进位。比如二进制的103，个位的3应该进位
    if (duration->tv_usec > 1000000) {
        //将之进位，因为下面会用到高位
        memcpy(&tv, duration, sizeof(struct timeval));
        if (is_common_timeout(duration, base))
            tv.tv_usec &= MICROSECONDS_MASK;//去除common-timeout标志
        tv.tv_sec += tv.tv_usec / 1000000; //进位
        tv.tv_usec %= 1000000;
        duration = &tv;
    }

    for (i = 0; i < base->n_common_timeouts; ++i) {
        const struct common_timeout_list *ctl =
            base->common_timeout_queues[i];
        //具有相同的duration，即之前有申请过这个超时时长。那么就不用分配空间。
        if (duration->tv_sec == ctl->duration.tv_sec &&
            duration->tv_usec ==
            (ctl->duration.tv_usec & MICROSECONDS_MASK)) {//要&这个宏，才能是正确的时间
            result = &ctl->duration;
            goto done;
        }
    }

    //达到了最大申请个数，不能再分配了
    if (base->n_common_timeouts == MAX_COMMON_TIMEOUTS) {
        goto done;
    }

    //新的超时时长，需要分配一个common_timeout_list结构体。

    //之前分配的空间已经用完了，要重新申请空间
    if (base->n_common_timeouts_allocated == base->n_common_timeouts) {
        int n = base->n_common_timeouts < 16 ? 16 :
            base->n_common_timeouts*2;
        struct common_timeout_list **newqueues =
            mm_realloc(base->common_timeout_queues,
            n*sizeof(struct common_timeout_queue *));
        if (!newqueues) {
            goto done;
        }
        base->n_common_timeouts_allocated = n;
        base->common_timeout_queues = newqueues;
    }

    //为该超时时长分配一个common_timeout_list结构体
    new_ctl = mm_calloc(1, sizeof(struct common_timeout_list));
```

```
58      if (!new_ctl) {
59          goto done;
60      }
61
62      //为这个结构体进行一些设置
63      TAILQ_INIT(&new_ctl->events);
64      new_ctl->duration.tv_sec = duration->tv_sec;
65      new_ctl->duration.tv_usec =
66          duration->tv_usec | COMMON_TIMEOUT_MAGIC | //为这个时间加入common-timeout标志
67          (base->n_common_timeouts << COMMON_TIMEOUT_IDX_SHIFT);//加入下标值
68
69      //对timeout_event这个内部event进行赋值。设置回调函数和回调参数。
70      evtimer_assign(&new_ctl->timeout_event, base,
71          common_timeout_callback, new_ctl);
72
73      new_ctl->timeout_event.ev_flags |= EVLIST_INTERNAL; //标志成内部event
74      event_priority_set(&new_ctl->timeout_event, 0); //优先级为最高级
75      new_ctl->base = base;
76      //放到数组对应的位置上
77      base->common_timeout_queues[base->n_common_timeouts++] = new_ctl;
78      result = &new_ctl->duration;
79
80  done:
81
82      return result;
83  }
```

该函数只是在 event_base 的 common_timeout_queues 数组中申请一个特定超时时长的位置。同时该函数也会返回一个 struct timeval 结构体指针变量，该结构体已经被赋予了 common-timeout 标志。以后使用该变量作为 event_add 的第二个参数，就可以把超时 event 插入到 common-timeout 中了。不应该也不能自己手动为 struct timeval 变量加入 common-timeout 标志。该函数中，也给内部的 event 进行了赋值, 设置了回调函数和回调参数。要注意的是回调参数是这个 common_timeout_list 结构体变量指针。在回调函数中，有了这个指针，就可以访问 events 变量，即访问到该结构体上的所有超时 event。于是就能手动激活这些超时 event。在 Libevent 的官方例子中，得到 event_base_init_common_timeout 的返回值后，就把它存放到另外一个 struct timeval 结构体中。而不是直接使用返回值作为 event_add 的参数。

将超时 event 存放到 common-timeout 中：现在已经向 event_base 申请了一个特定的超时时长，并得到了具有 common-timeout 标志的时间。那么，就调用 event_add 看看。

```
1   //event.c文件
2   static inline int
3   event_add_internal(struct event *ev, const struct timeval *tv,
4       int tv_is_absolute)
5   {
6       struct event_base *base = ev->ev_base;
7       int res = 0;
8       int notify = 0;
9
10      ... //加入到IO队列或者信号队列
11
12      if (res != -1 && tv != NULL) {
13          struct timeval now;
14          int common_timeout;
15
16          gettime(base, &now);
17
18          //判断这个时间是否为common-timeout标志
19          common_timeout = is_common_timeout(tv, base);
20          if (common_timeout) {
```

```
            struct timeval tmp = *tv;
            //只取真正的时间部分，common-timeout标志位和下标位不要
            tmp.tv_usec &= MICROSECONDS_MASK;
            //转换成绝对时间
            evutil_timeradd(&now, &tmp, &ev->ev_timeout);
            ev->ev_timeout.tv_usec |=
                (tv->tv_usec & ~MICROSECONDS_MASK); //加入标志位
        }

        event_queue_insert(base, ev, EVLIST_TIMEOUT);

        if (common_timeout) {
            struct common_timeout_list *ctl =
                get_common_timeout_list(base, &ev->ev_timeout);
            if (ev == TAILQ_FIRST(&ctl->events)) {
                common_timeout_schedule(ctl, &now, ev);
            }
        }
    }

    return (res);
}
```

由于在【超时 event 的处理】中已经对这个函数进行了一部分讲解，现在只讲有关 common-timeout 部分。

虽然上面的代码省略了很多东西，但是有一点要说明，当超时 event 被加入 common-timeout 时并不会设置 notify 变量的，即不需要通知主线程。

common-timeout 与小根堆的配合：从上面的代码可以看到，首先是为超时 event 内部时间 ev_timeout 加入 common-timeout 标志。然后调用 event_queue_insert 进行插入。但此时调用 event_queue_insert 插入，并不是插入到小根堆。它只是插入到 event_base 的 common_timeout_list 数组的一个队列中。下面代码可以看到这一点。

```
static void
event_queue_insert(struct event_base *base, struct event *ev, int queue)
{
    EVENT_BASE_ASSERT_LOCKED(base);

    if (ev->ev_flags & queue) {
        /* Double insertion is possible for active events */
        if (queue & EVLIST_ACTIVE)
            return;

        event_errx(1, "%s: %p(fd "EV_SOCK_FMT") already on queue %x", __func__,
            ev, EV_SOCK_ARG(ev->ev_fd), queue);
        return;
    }

    if (~ev->ev_flags & EVLIST_INTERNAL)
        base->event_count++;

    ev->ev_flags |= queue;
    switch (queue) {
    case EVLIST_INSERTED:
        TAILQ_INSERT_TAIL(&base->eventqueue, ev, ev_next);
        break;
    case EVLIST_ACTIVE:
        base->event_count_active++;
        TAILQ_INSERT_TAIL(&base->activequeues[ev->ev_pri],
            ev,ev_active_next);
        break;
    case EVLIST_TIMEOUT: {
        if (is_common_timeout(&ev->ev_timeout, base)) {
```

```
            struct common_timeout_list *ctl =
                get_common_timeout_list(base, &ev->ev_timeout);
            insert_common_timeout_inorder(ctl, ev);
        } else
            min_heap_push(&base->timeheap, ev);
        break;
    }
    default:
        event_errx(1, "%s: unknown queue %x", __func__, queue);
    }
}

static void //in order说明是有序的。
insert_common_timeout_inorder(struct common_timeout_list *ctl,
    struct event *ev)
{
    struct event *e;
    //虽然有相同超时时长，但超时时间却是 超时时长 + 调用event_add的时间。
    //所以是在不同的时间触发超时的。它们根据绝对超时时间，升序排在队列中。
    //一般来说，直接插入队尾即可。因为后插入的，绝对超时时间肯定大。
    //但由于线程抢占的原因，可能一个线程在evutil_timeradd(&now, &tmp, &ev->ev_timeout);
    //执行完，还没来得及插入，就被另外一个线程抢占了。而这个线程也是要插入一个
    //common-timeout的超时event。这样就会发生：超时时间小的反而后插入。
    //所以要从后面开始遍历队列，寻找一个合适的地方。
    TAILQ_FOREACH_REVERSE(e, &ctl->events,
        event_list, ev_timeout_pos.ev_next_with_common_timeout) {
        if (evutil_timercmp(&ev->ev_timeout, &e->ev_timeout, >=)) {
            TAILQ_INSERT_AFTER(&ctl->events, e, ev, //从队列后面插入
                ev_timeout_pos.ev_next_with_common_timeout);
            return; //插入后就返回
        }
    }

    //在队列头插入，只会发生在前面的寻找都没有寻找到的情况下
    TAILQ_INSERT_HEAD(&ctl->events, ev,
        ev_timeout_pos.ev_next_with_common_timeout);
}
```

既然 event_queue_insert 函数并没有完成插入到小根堆。那么就看 event_add_internal 的最后面的那个 if 判断。读者可能会问，为什么要插入到小根堆。其实，前面已经说到了。common-timeout 是采用一个代表的方式进行工作的。所以肯定要有一个代表被插入小根堆中，这也是 common-timeout 和小根堆的相互配合。

```
//event.c文件
if (common_timeout) {
    struct common_timeout_list *ctl =
        get_common_timeout_list(base, &ev->ev_timeout);
    if (ev == TAILQ_FIRST(&ctl->events)) {
        common_timeout_schedule(ctl, &now, ev);
    }
}

static void
common_timeout_schedule(struct common_timeout_list *ctl,
    const struct timeval *now, struct event *head)
{
    struct timeval timeout = head->ev_timeout;
    timeout.tv_usec &= MICROSECONDS_MASK; //清除common-timeout标志
    //用common_timeout_list结构体的一个event成员作为超时event调用event_add_internal
    //由于已经清除了common-timeout标志，所以这次将插入到小根堆中。
    event_add_internal(&ctl->timeout_event, &timeout, 1);
}
```

从判断可以看到，它判断要插入的这个超时 event 是否为这个队列的第一个元素。如果是的话，就说这个特定超时时长队列第一次有超时 event 要插入。这就要进行一些处理。在 common_timeout_schedule 函数中，我们可以看到，它将一个 event 插入到小根堆中了。并且也可以看到，代表者不是用户给出的超时 event 中的一个，而是 common_timeout_list 结构体的一个 event 成员。

将 common-timeout event 激活：现在来看一下当 common_timeout_list 的内部 event 成员被激活时怎么处理。它的回调函数为 common_timeout_callback。

```
1  //event.c文件
2  static void common_timeout_callback(evutil_socket_t fd, short what, void *arg) {
3      struct timeval now;
4      struct common_timeout_list *ctl = arg;
5      struct event_base *base = ctl->base;
6      struct event *ev = NULL;
7      EVBASE_ACQUIRE_LOCK(base, th_base_lock);
8      gettime(base, &now);
9      while (1) {
10         ev = TAILQ_FIRST(&ctl->events);
11
12         //该超时event还没到超时时间。不要检查其他了。因为是升序的
13         if (!ev || ev->ev_timeout.tv_sec > now.tv_sec ||
14             (ev->ev_timeout.tv_sec == now.tv_sec &&
15             (ev->ev_timeout.tv_usec&MICROSECONDS_MASK) > now.tv_usec))
16             break;
17
18         //一系列的删除操作。包括从这个超时event队列中删除
19         event_del_internal(ev);
20         //手动激活超时event。注意，这个ev是用户的超时event
21         event_active_nolock(ev, EV_TIMEOUT, 1);
22     }
23
24     //不是NULL，说明该队列还有超时event。那么需要再次common_timeout_schedule，进行监听
25     if (ev)
26         common_timeout_schedule(ctl, &now, ev);
27     EVBASE_RELEASE_LOCK(base, th_base_lock);
28 }
```

在回调函数中，会手动把用户的超时 event 激活。于是，用户的超时 event 就能被处理了。由于 Libevent 这个内部超时 event 的优先级是最高的，所以在接下来就会处理用户的超时 event，而无需等到下一轮多路 IO 复用函数调用返回后。

7 Signal 事件

信号（signal，信号）用来通知进程发生了异步事件，是在软件层次上对中断机制的模拟。在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是进程间通信机制中唯一的异步通信机制，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。进程之间可以互相通过系统调用 kill 等发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。Signal 是异步事件的经典事例，将 Signal 事件统一到系统的 I/O 多路复用中就不像 Timer 事件那么自然，Signal 事件的出现对于进程来讲是完全随机的，进程不能只是测试一个变量来判别一个信号是否发生，而是必须告诉内核“在此信号发生时，请执行如下的操作”。详见 signal 和 sigaction 函数。要将 Signal 事件的管理集成到系统的 IO 机制中，一般采用这样的思想：如果当 Signal 发生时，并不立即调用该信号相应的事件处理函数，而是通过管道或其他方式通知 I/O 复用程序将其加入到激活事件队列中，然后再统一与 I/O 事件以及 Timer 事件一起处理。其示意图可以见图【】。Libevent 也采用了这样的思想，不过它使用的通知方式是 socketpair。【此处是不是要添加 socketpair 的简单介绍。】

libevent 的信号集成过程如下：此处要插入一幅上述思想的示意图

简述几个步骤：

1. 创建 socketpair
2. 为这个 socketpair 读端创建一个 event，并将只加入 event_base 中
3. 设置信号捕获函数
4. 有信号发生，信号捕获函数在 socketpair 写端像写端写入信号的编号
5. IO 复用函数中监听到信号发生的事件，即可将之与其他事件一起处理

我们先来看一下 event_base 结构体中关于 Signal event 的相关成员，如下：

```
1  struct event_base{
2      const struct eventop *evsigsel;
3      struct evsig_info sig;        /* */
4       ...
5      struct event_signal_map sigmap; /**/
6       ...
7  };
8
9  struct evsig_info {
10     struct event ev_signal;
11     evutil_socket_t ev_signal_pair[2];
12     int ev_signal_added;
13     int ev_n_signals_added;
14
15 #ifdef _EVENT_HAVE_SIGACTION
16     struct sigaction **sh_old;
17 #else
18     ev_sighandler_t **sh_old;
19 #endif
20     int sh_old_max;
21 };
```

此结构体用于建立 signal 与 IO 复用后端的连接。其中 ev_signal 是一个 event 变量，用于监听 socketpair 读端 ev_signal_pair[1]；ev_signal_pair[2] 用于信号捕获函数与 IO 复用函数通信的 socketpair；ev_signal_added 用来标记是否已经将 ev_signal 加入到 event_base 中；ev_n_signals_added 要监听的信号数；sh_old 二级指针实际是一个数组，用户可能已经设置过某个信号的信号捕获函数，但 Libevent 也要为这个信号

设置另外一个信号捕获函数，此时就要保存用户之前设置的信号捕获函数。当 Libevent 不要监听这个信号时，就能够恢复用户之前的捕获函数，而用户的捕获函数需要存储起来，而且会有多个信号，所以得用一个数组保存。最后的 sh_old_max 用于存放 sh_old 的大小。

在 S.Reactor 中讲到过 event_base_new_with_config 函数，其中的 for 循环是获取平台的 IO 多路复用后端接口，在获取到后端之后，有一行如下代码：

```
/*event.c 第 552 行*/
base->evbase = base->evsel->init(base);
```

此行代码即调用 IO 多路复用后端的初始化 init 函数，在该函数的最后会调用 evsig_init 函数，以 epoll 函数为例，如下：

```
/*epoll.c 第 108 行*/
static void * epoll_init (struct event_base *base) {
    … /*创建 epoll 实例、epoll 监听事件初始化等操作*/

    evsig_init (base);

    return (epollop);
}
```

而正是这个 evsig_init 函数完成了创建 socketpair 并将 socketpair 的一个读端与 event_info 结构体中的 ev_signal 相关联。

```
/*signal.c 第 169 行*/
int evsig_init (struct event_base *base) {
    if (evutil_socketpair (AF_UNIX, SOCK_STREAM, 0, base->sig.ev_signal_pair) == -1) {
#ifdef WIN32
        …
#else
        event_sock_err(1, -1, "%s: socketpair", __func__);
#endif
        return -1;
    }

    evutil_make_socket_closeonexec(base->sig.ev_signal_pair[0]);
    evutil_make_socket_closeonexec(base->sig.ev_signal_pair[1]);
    base->sig.sh_old = NULL;
    base->sig.sh_old_max = 0;

    evutil_make_socket_nonblocking(base->sig.ev_signal_pair[0]);
    evutil_make_socket_nonblocking(base->sig.ev_signal_pair[1]);

    event_assign(&base->sig.ev_signal, base, base->sig.ev_signal_pair[1],
        EV_READ | EV_PERSIST, evsig_cb, base);

    base->sig.ev_signal.ev_flags |= EVLIST_INTERNAL;
    event_priority_set(&base->sig.ev_signal, 0);

    base->evsigsel = &evsigops;
    return 0;
}
```

此函数先调用 evutil_socketpair(此函数是对 socketpair 创建接口的跨平台封装，在 evutil.c 第 183 行) 创建一个 socketpair，再分别调用 evutil_make_socket_closeonexec 和 evutil_make_socket_nonblocking 两个函数向 socketpair 读写两端的 flags 中添加 FD_CLOEXEC 和 O_NONBLOCK 选项；然后调用 event_assign 函数，将 ev_signal_pair[1] (读端) 与 event_base->sig.ev_signal 这个 event 相关联，即用 event_base->sig.ev_signal 这个事件去监听 socketpair 读端 ev_signal_pair[1]，【后面肯定是会将这个事件加入到监听事件队列中的】。最后将此事件设置为内部事件、将其优先级设为 0(最高)、以及将信号操作函数结构体赋值。

【关于 FD_CLOEXEC 和 O_NONBLOCK 两个选项】

socketpair 的两个端都调用 evutil_make_socket_closeonexec 和 evutil_make_socket_nonblocking 函数。其实现分别如下：

```
int evutil_make_socket_closeonexec(evutil_socket_t fd) {
#if !defined(WIN32) && defined(_EVENT_HAVE_SETFD)
    int flags;
    if ((flags = fcntl(fd, F_GETFD, NULL)) < 0) {
        event_warn("fcntl(%d, F_GETFD)", fd);
        return -1;
    }
    if (fcntl(fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
        event_warn("fcntl(%d, F_SETFD)", fd);
        return -1;
    }
#endif
    return 0;
}

int evutil_make_socket_nonblocking(evutil_socket_t fd) {
#ifdef WIN32   略去
#else
    {
        int flags;
        if ((flags = fcntl(fd, F_GETFL, NULL)) < 0) {
            event_warn("fcntl(%d, F_GETFL)", fd);
            return -1;
        }
        if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
            event_warn("fcntl(%d, F_SETFL)", fd);
            return -1;
        }
    }
#endif
    return 0;
}
```

因为不能让子进程可以访问的这个 socketpair。因为子进程的访问可能会出现扰乱。比如，子进程往 socketpair 发送信息，使得父进程的多路 IO 复用函数误以为信号发生了；父进程确实发生了信号，也往 socketpair 发送了一个字节，但却被子进程接收了这个字节。父进程没有监听到可读。关于 close on exec，从 linux 内核 2.6.27 起，创建文件描述符的 flags 参数增加额外的选项，可以直接指定 O_NONBLOCK 和 FD_CLOEXEC 选项。FD_CLOEXEC 的功能是让程序 exec() 时，进程回自动关闭这个文件描述符，而文件描述符默认是被子进程继承的。

回到 evsig_init 函数，在返回之前可以看到 event_base 的一个成员 evsignal 被赋值为 evsigops，evsignal 是一个 IO 复用结构体，而 evsigops 是专门用于信号处理的 IO 复用结构体变量。定义如下：

```
static const struct eventop evsigops = {
    "signal",
    NULL,
    evsig_add,
    evsig_del,
    NULL,
    NULL,
    0, 0, 0
};
```

将信号 event 加入到 event_base: 前面的代码已经完成了“创建 socketpair 并将 socketpair 的一个读端于 ev_signal 相关联”。接下来看其他的工作。假如要对一个绑定了某个信号的 event 调用 event_add 函数，那么在 event_add 的内部会调用 event_add_internal

函数。而 event_add_internal 函数又会调用 evmap_signal_add 函数。有了前面几章的讲解，应该对这个流程不陌生。下面看看 evmap_signal_add 函数：

```
int evmap_signal_add(struct event_base *base, int sig, struct event *ev)
{
    const struct eventop *evsel = base->evsigsel;
    struct event_signal_map *map = &base->sigmap;
    struct evmap_signal *ctx = NULL;

    if (sig >= map->nentries) {
        if (evmap_make_space(
            map, sig, sizeof(struct evmap_signal *)) == -1)
            return (-1);
    }
    GET_SIGNAL_SLOT_AND_CTOR(ctx, map, sig, evmap_signal, evmap_signal_init,
        base->evsigsel->fdinfo_len);

    if (TAILQ_EMPTY(&ctx->events)) {
        if (evsel->add(base, ev->ev_fd, 0, EV_SIGNAL, NULL)
            == -1)
            return (-1);
    }

    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_signal_next);

    return (1);
}
```

上面函数的内部调用了 IO 复用结构体的 add 函数指针，即调用了 evsig_add。现在我们深入 evsig_add 函数。

```
static int evsig_add(struct event_base *base, evutil_socket_t evsignal, short old, short events, void
    *p)
{
    struct evsig_info *sig = &base->sig;
    (void)p;

    EVUTIL_ASSERT(evsignal >= 0 && evsignal < NSIG);

    /* catch signals if they happen quickly */
    EVSIGBASE_LOCK();
    if (evsig_base != base && evsig_base_n_signals_added) {
        event_warnx("Added a signal to event base %p with signals "
            "already added to event_base %p.  Only one can have "
            "signals at a time with the %s backend.  The base with "
            "the most recently added signal or the most recent "
            "event_base_loop() call gets preference; do "
            "not rely on this behavior in future Libevent versions.",
            base, evsig_base, base->evsel->name);
    }
    evsig_base = base;
    evsig_base_n_signals_added = ++sig->ev_n_signals_added;
    evsig_base_fd = base->sig.ev_signal_pair[0];
    EVSIGBASE_UNLOCK();

    event_debug(("%s: %d: changing signal handler", __func__, (int)evsignal));
    if (_evsig_set_handler(base, (int)evsignal, evsig_handler) == -1) {
        goto err;
    }

    event_base 第一次监听信号事件。要添加 ev_signal 到 event_base 中
    if (!sig->ev_signal_added) {
    注意，本函数的调用路径为 event_add->event_add_internal->evmap_signal_map->evsig_add
        所以这里是递归调用 event_add 函数。而 event_add 函数是会加锁的。所以需要锁为递归锁
```

```
        if (event_add(&sig->ev_signal, NULL))
            goto err;
        sig->ev_signal_added = 1;
    }

    return (0);

err:
    EVSIGBASE_LOCK();
    --evsig_base_n_signals_added;
    --sig->ev_n_signals_added;
    EVSIGBASE_UNLOCK();
    return (-1);
}
```

从后面的那个 if 语句可以得知，当 sig->ev_signal_added 变量为 0 时 (即用户第一次监听一个信号)，就会将 ev_signal 这个 event 加入到 event_base 中。从前面的“统一事件源”可以得知，这个 ev_signal 的作用就是通知 event_base，有信号发生了。只需一个 event 即可完成工作，即使用户要监听多个不同的信号，因为这个 event 已经和 socketpair 的读端相关联了。如果要监听多个信号，那么就在信号处理函数中往这个 socketpair 写入不同的值即可。event_base 能监听到可读，并可以从读到的内容可以判断是哪个信号发生了。从代码中也可得知，Libevent 并不会为每一个信号监听创建一个 event。它只会创建一个全局的专门用于监听信号的 event。这个也是“统一事件源”的工作原理。

设置信号捕抓函数：evsig_add 函数还调用了 _evsig_set_handler 函数完成设置 Libevent 内部的信号捕抓函数。/* Helper: set the signal handler for evsignal to handler in base, so that * we can restore the original handler when we clear the current one. */

```
int
_evsig_set_handler(struct event_base *base,
    int evsignal, void (__cdecl *handler)(int))
{
#ifdef _EVENT_HAVE_SIGACTION
    struct sigaction sa;
#else
    ev_sighandler_t sh;
#endif
    struct evsig_info *sig = &base->sig;
    void *p;

    /*
     * resize saved signal handler array up to the highest signal number.
     * a dynamic array is used to keep footprint on the low side.
     */
     数组的一个元素就存放一个信号。信号值等于其下标
    if (evsignal >= sig->sh_old_max) {
        int new_max = evsignal + 1;
        event_debug(("%s: evsignal (%d) >= sh_old_max (%d), resizing",
                __func__, evsignal, sig->sh_old_max));
        p = mm_realloc(sig->sh_old, new_max * sizeof(*sig->sh_old));
        if (p == NULL) {
            event_warn("realloc");
            return (-1);
        }

        memset((char *)p + sig->sh_old_max * sizeof(*sig->sh_old),
            0, (new_max - sig->sh_old_max) * sizeof(*sig->sh_old));

        sig->sh_old_max = new_max;
        sig->sh_old = p;
    }
```

```
34
35      注意 sh_old 是一个二级指针。元素是一个一级指针。为这个一级指针分配内存
36      /* allocate space for previous handler out of dynamic array */
37      sig->sh_old[evsignal] = mm_malloc(sizeof *sig->sh_old[evsignal]);
38      if (sig->sh_old[evsignal] == NULL) {
39          event_warn("malloc");
40          return (-1);
41      }
42
43      /* save previous handler and setup new handler */
44  #ifdef _EVENT_HAVE_SIGACTION
45      memset(&sa, 0, sizeof(sa));
46      sa.sa_handler = handler;
47      sa.sa_flags |= SA_RESTART;
48       sigfillset (&sa.sa_mask);
49
50      if (sigaction(evsignal, &sa, sig->sh_old[evsignal]) == -1) {
51          event_warn("sigaction");
52          mm_free(sig->sh_old[evsignal]);
53          sig->sh_old[evsignal] = NULL;
54          return (-1);
55      }
56  #else
57      if ((sh = signal(evsignal, handler)) == SIG_ERR) {
58          event_warn("signal");
59          mm_free(sig->sh_old[evsignal]);
60          sig->sh_old[evsignal] = NULL;
61          return (-1);
62      }
63      *sig->sh_old[evsignal] = sh;
64  #endif
65
66      return (0);
67  }
```

当我们对某个信号进行 event_new 和 event_add 后，就不应该再次设置该信号的信号捕抓函数。否则 event_base 将无法监听到信号的发生。通过在外部给这个进程发生信号的方式。可以看到，event_base 无法监听到该信号。所有信号会被 signal_handle 捕抓。可以通过下面的代码验证：

```
1   #include<unistd.h>
2   #include<stdio.h>
3   #include<signal.h>
4   #include<event.h>
5
6   void sig_cb(int fd, short events, void *arg) {
7       printf ("in␣the␣sig_cb\n");
8   }
9
10  void signal_handle(int sig) {
11      printf ("catch␣the␣sig␣%d\n", sig);
12  }
13
14  int main() {
15      struct event_base *base = event_base_new();
16
17      struct event *ev = evsignal_new(base, SIGUSR1, sig_cb, NULL);
18      event_add(ev, NULL);
19
20      signal(SIGUSR1, signal_handle);
21
22      printf ("pid␣=␣%d\n", getpid());
23
```

```
24      printf ("begin\n");
25      event_base_dispatch(base);
26      printf ("end\n");
27
28      return 0;
29  }
```

捕抓信号：前面的代码中有两个函数并没有讲，分别是信号捕抓函数 evsig_handler 和调用 event_assign 时的信号回调函数 evsig_cb。

```
1   static void __cdecl
2   evsig_handler(int sig)
3   {
4       int save_errno = errno;
5   #ifdef WIN32
6       int socket_errno = EVUTIL_SOCKET_ERROR();
7   #endif
8       ev_uint8_t msg;
9
10      if (evsig_base == NULL) {
11          event_warnx(
12              "%s: received signal %d, but have no base configured",
13              __func__, sig);
14          return;
15      }
16
17  #ifndef _EVENT_HAVE_SIGACTION
18      signal(sig, evsig_handler);
19  #endif
20
21      /* Wake up our notification mechanism */
22      msg = sig;
23      send(evsig_base_fd, (char*)&msg, 1, 0);
24      errno = save_errno;
25  #ifdef WIN32
26      EVUTIL_SET_SOCKET_ERROR(socket_errno);
27  #endif
28  }
```

从 evsig_handler 函数的实现可以看到，实现得相当简单。只是将信号对应的值写入到 socketpair 中。evsig_base_fd 是 socketpair 的写端，这是一个全局变量，在 evsig_add 函数中被赋值的。从“统一事件源”的工作原理来看，现在已经完成了对信号的捕抓，已经将该信号的当作 IO 事件写入到 socketpair 中了。现在 event_base 应该已经监听到 socketpair 可读了，并且会为调用回调函数 evsig_cb 了。下面看看 evsig_cb 函数。

```
1   /* Callback for when the signal handler write a byte to our signaling socket */
2   static void
3   evsig_cb(evutil_socket_t fd, short what, void *arg)
4   {
5       static char signals[1024];
6       ev_ssize_t n;
7       int i;
8       int ncaught[NSIG];
9       struct event_base *base;
10
11      base = arg;
12
13      memset(&ncaught, 0, sizeof(ncaught));
14
15      while (1) {
16          n = recv(fd, signals, sizeof(signals), 0);
17          if (n == -1) {
18              int err = evutil_socket_geterror(fd);
```

```
19              if (! EVUTIL_ERR_RW_RETRIABLE(err))
20                  event_sock_err(1, fd, "%s: recv", __func__);
21              break;
22          } else if (n == 0) {
23              /* XXX warn? */
24              break;
25          }
26          for (i = 0; i < n; ++i) {
27              ev_uint8_t sig = signals[i];
28              if (sig < NSIG)
29                  ncaught[sig]++;
30          }
31      }
32
33      EVBASE_ACQUIRE_LOCK(base, th_base_lock);
34      for (i = 0; i < NSIG; ++i) {
35          if (ncaught[i])
36              evmap_signal_active(base, i, ncaught[i]);
37      }
38      EVBASE_RELEASE_LOCK(base, th_base_lock);
39  }
```

该回调函数的作用是读取 socketpair 的所有数据，并将数据当作信号，再根据信号值调用 evmap_signal_active。有一点要注意，evsig_cb 这个回调函数并不是用户为监听一个信号调用 event_new 时设置的用户回调函数，而是 Libevent 内部为了处理信号而设置的内部回调函数。

激活信号 event:

虽然如此，但是现在的情况是：当有信号发生时，就会调用 evmap_signal_active 函数。

```
1   //event-internal.h 文件
2   #define ev_signal_next  _ev.ev_signal.ev_signal_next
3   #define ev_ncalls   _ev.ev_signal.ev_ncalls
4   #define ev_pncalls  _ev.ev_signal.ev_pncalls
5
6   //evmap.c 文件
7   //后两个参数分别是信号值和发生的次数
8   void evmap_signal_active(struct event_base *base, evutil_socket_t sig, int ncalls) {
9       struct event_signal_map *map = &base->sigmap;
10      struct evmap_signal *ctx;
11      struct event *ev;
12
13      //通过这个 fd 找到对应的 TAILQ_HEAD
14      GET_SIGNAL_SLOT(ctx, map, sig, evmap_signal);
15
16      //遍历该 fd 的队列
17      TAILQ_FOREACH(ev, &ctx->events, ev_signal_next)
18          event_active_nolock(ev, EV_SIGNAL, ncalls);
19  }
20
21  //event.c 文件
22  void event_active_nolock(struct event *ev, int res, short ncalls) {
23      struct event_base *base;
24
25      base = ev->ev_base;
26      ev->ev_res = res;
27
28      //这将停止处理低优先级的 event。一路回退到 event_base_loop 中
29      if (ev->ev_pri < base->event_running_priority)
30          base->event_continue = 1;
31
32      if (ev->ev_events & EV_SIGNAL) {
```

```
#ifndef _EVENT_DISABLE_THREAD_SUPPORT
        if (base->current_event == ev && !EVBASE_IN_THREAD(base)) {
            ++base->current_event_waiters;
            //由于此时是主线程执行，所以并不会进行这个判断里面
            EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lock);
        }
#endif
        ev->ev_ncalls = ncalls;
        ev->ev_pncalls = NULL;
    }

    //插入到激活队列中. 插入到队尾
    event_queue_insert(base, ev, EVLIST_ACTIVE);
}
```

通过 evmap_signal_active、event_active_nolock 和 event_queue_insert 这三个函数的调用后，就可以把一个 event 插入到激活队列了。

由于这些函数的执行本身就是在 Libevent 处理 event 的回调函数之中的 (Libevent 正在处理内部的信号处理 event)。所以并不需要从 event_base_loop 里的 while 循环里面再次执行一次 evsel->dispatch()，才能执行到这次信号 event。即无需等到下一次处理激活队列，就可以执行该信号 event 了。分析如下：首先要明确，现在执行上面三个函数相当于在执行 event 的回调函数。所以其是运行在 event_process_active 函数之中的。

```
//event.c文件
static int event_process_active(struct event_base *base) {
    /* Caller must hold th_base_lock */
    struct event_list *activeq = NULL;
    int i, c = 0;

    //从高优先级到低优先级遍历优先级数组
    for (i = 0; i < base->nactivequeues; ++i) {
        //对于特定的优先级，遍历该优先级的所有激活 event
        if (TAILQ_FIRST(&base->activequeues[i]) != NULL) {
            base->event_running_priority = i;
            activeq = &base->activequeues[i];
            c = event_process_active_single_queue(base, activeq);
            if (c < 0) {
                base->event_running_priority = -1;
                return -1;
            } else if (c > 0)
                break; /* Processed a real event; do not consider lower-priority events */
            /* If we get here, all of the events we processed were internal. Continue. */
        }
    }

    event_process_deferred_callbacks(&base->defer_queue,&base->event_break);
    base->event_running_priority = -1;
    return c;
}

static int event_process_active_single_queue(struct event_base *base, struct event_list *activeq) {
    struct event *ev;

    //遍历该优先级的所有 event，并处理之
    for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {

        ... //开始处理这个 event。会调用 event 的回调函数
    }
}
```

从上面的代码可以看到，Libevent 在处理内部的那个信号处理 event 的回调函数时，其

实是在 event_process_active_single_queue 的一个循环里面。因为 Libevent 内部的信号处理 event 的优先级最高优先级，并且在前面的将用户信号 event 插入到队列 (即 event_queue_insert)，在插入到队列的尾部。所以无论用户的这个信号 event 的优先级是多少，都是在 Libevent 的内部信号处理 event 的后面。所以在遍历上面两个函数的里外两个循环时，肯定会执行到用户的信号 event。

执行已激活信号 event：现在看看 Libevent 是怎么处理已激活的信号 event 的。

```
1  //event.c文件
2  static  inline  void event_signal_closure(struct event_base *base, struct event *ev) {
3      short ncalls;
4      int  should_break;
5
6      /*Allows deletes to  work */
7      ncalls  = ev->ev_ncalls;
8      if  (ncalls  != 0)
9          ev->ev_pncalls = &ncalls;
10
11     //while 循环里面会调用用户设置的回调函数。该回调函数可能会执行很久所以要解锁先
12     EVBASE_RELEASE_LOCK(base, th_base_lock);
13     //如果该信号发生了多次，那么就需要多次执行回调函数
14     while (ncalls) {
15         ncalls--;
16         ev->ev_ncalls = ncalls;
17         if  (ncalls  == 0)
18             ev->ev_pncalls = NULL;
19         (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);
20
21         EVBASE_ACQUIRE_LOCK(base, th_base_lock);
22         //其他线程调用 event_base_loopbreak 函数中断之
23         should_break = base->event_break;
24         EVBASE_RELEASE_LOCK(base, th_base_lock);
25
26         if  (should_break) {
27             if  (ncalls  != 0)
28                 ev->ev_pncalls = NULL;
29             return;
30         }
31     }
32 }
```

可以看到，如果对应的信号发生了多次，那么该信号 event 的回调函数将被执行多次。

8 连接监听 evconnlistener

使用 evconnlistener：基于 event 和 event_base 已经可以写一个 CS 模型了。但是对于服务器端来说，仍然需要用户自行调用 socket、bind、listen、accept 等步骤。这个过程有点繁琐，为此在 2.0.2-alpha 版本的 Libevent 推出了一些对应的封装函数。用户只需初始化 struct sockaddr_in 结构体变量，然后把它作为参数传给函数 evconnlistener_new_bind 即可。该函数会完成上面说到的那 4 个过程。下面的代码是一个使用例子。

```
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>

#include<stdio.h>
#include<string.h>

#include<event.h>
#include<listener.h>
#include<bufferevent.h>
#include<thread.h>

void listener_cb(evconnlistener * listener , evutil_socket_t fd,
                 struct sockaddr *sock, int socklen, void *arg);

void socket_read_cb(bufferevent *bev, void *arg);
void socket_error_cb(bufferevent *bev, short events, void *arg);

int main() {
    evthread_use_pthreads();//enable threads

    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(struct sockaddr_in));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(8989);

    event_base *base = event_base_new();
    evconnlistener * listener
            = evconnlistener_new_bind(base, listener_cb, base,
                                      LEV_OPT_REUSEABLE|LEV_OPT_CLOSE_ON_FREE |
        LEV_OPT_THREADSAFE,
                                      10, (struct sockaddr*)&sin, sizeof(struct sockaddr_in));

    event_base_dispatch(base);
    evconnlistener_free(listener);
    event_base_free(base);

    return 0;
}

//有新的客户端连接到服务器
//当此函数被调用时，libevent已经帮我们accept了这个客户端。该客户端的文件描述符为fd
void listener_cb(evconnlistener * listener , evutil_socket_t fd,
                 struct sockaddr *sock, int socklen, void *arg) {
    event_base *base = (event_base*)arg;

    //下面代码是为这个fd创建一个bufferevent
    bufferevent *bev = bufferevent_socket_new(base, fd,
                                              BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, socket_read_cb, NULL, socket_error_cb, NULL);
    bufferevent_enable(bev, EV_READ | EV_PERSIST);
}
```

```
53
54  void socket_read_cb(bufferevent *bev, void *arg) {
55      char msg[4096];
56
57      size_t len = bufferevent_read(bev, msg, sizeof(msg)-1 );
58
59      msg[len] = '\0';
60       printf ("server read the data %s\n", msg);
61
62      char reply[] = "I has read your data";
63      bufferevent_write(bev, reply, strlen(reply) );
64  }
65
66  void socket_error_cb(bufferevent *bev, short events, void *arg) {
67       if (events & BEV_EVENT_EOF)
68           printf ("connection closed\n");
69      else if (events & BEV_EVENT_ERROR)
70           printf ("some other error\n");
71
72      //这将自动close套接字和free读写缓冲区
73      bufferevent_free(bev);
74  }
75
76  //客户端代码
77  #include<sys/types.h>
78  #include<sys/socket.h>
79  #include<netinet/in.h>
80  #include<arpa/inet.h>
81  #include<errno.h>
82  #include<unistd.h>
83
84  #include<stdio.h>
85  #include<string.h>
86  #include<stdlib.h>
87
88  #include<event.h>
89  #include<event2/bufferevent.h>
90  #include<event2/buffer.h>
91  #include<event2/util.h>
92
93  int tcp_connect_server(const char* server_ip, int port);
94
95  void cmd_msg_cb(int fd, short events, void* arg);
96  void server_msg_cb(struct bufferevent* bev, void* arg);
97  void event_cb(struct bufferevent *bev, short event, void *arg);
98
99  int main(int argc, char** argv) {
100      if( argc < 3 ) {
101          //两个参数依次是服务器端的IP地址、端口号
102          printf ("please input 2 parameter\n");
103          return -1;
104      }
105
106      struct event_base *base = event_base_new();
107
108      struct bufferevent* bev = bufferevent_socket_new(base, -1,
109                                                       BEV_OPT_CLOSE_ON_FREE);
110
111      //监听终端输入事件
112      struct event* ev_cmd = event_new(base, STDIN_FILENO,
113                                       EV_READ | EV_PERSIST,
114                                       cmd_msg_cb, (void*)bev);
115
```

```
116     event_add(ev_cmd, NULL);
117
118     struct sockaddr_in server_addr;
119
120     memset(&server_addr, 0, sizeof(server_addr) );
121
122     server_addr.sin_family = AF_INET;
123     server_addr.sin_port = htons(atoi(argv[2]));
124     inet_aton(argv[1],  &server_addr.sin_addr);
125
126     bufferevent_socket_connect(bev, (struct sockaddr *)&server_addr,
127                                sizeof(server_addr));
128
129     bufferevent_setcb(bev, server_msg_cb, NULL, event_cb, (void*)ev_cmd);
130     bufferevent_enable(bev, EV_READ | EV_PERSIST);
131
132     event_base_dispatch(base);
133
134     printf("finished \n");
135     return 0;
136 }
137
138 void cmd_msg_cb(int fd, short events, void* arg) {
139     char msg[1024];
140
141     int ret = read(fd, msg, sizeof(msg));
142     if( ret < 0 )
143     {
144         perror("read fail ");
145         exit(1);
146     }
147
148     struct bufferevent* bev = (struct bufferevent*)arg;
149
150     //把终端的消息发送给服务器端
151     bufferevent_write(bev, msg, ret);
152 }
153
154 void server_msg_cb(struct bufferevent* bev, void* arg) {
155     char msg[1024];
156
157     size_t len = bufferevent_read(bev, msg, sizeof(msg));
158     msg[len] = '\0';
159
160     printf("recv %s from server\n", msg);
161 }
162
163 void event_cb(struct bufferevent *bev, short event, void *arg) {
164
165     if (event & BEV_EVENT_EOF)
166         printf("connection closed\n");
167     else if (event & BEV_EVENT_ERROR)
168         printf("some other error\n");
169     else if( event & BEV_EVENT_CONNECTED)
170     {
171         printf("the client has connected to server\n");
172         return ;
173     }
174
175     //这将自动close套接字和free读写缓冲区
176     bufferevent_free(bev);
177
178     struct event *ev = (struct event*)arg;
```

```
179     event_free(ev);
180 }
```

从上面代码可以看到，当服务器端监听到一个客户端的连接请求后，就会调用 listener_cb 这个回调函数。这个回调函数是在 evconnlistener_new_bind 函数中设置的。现在来看一下这个函数的参数有哪些，下面是其函数原型。

```
1  typedef void (*evconnlistener_cb)(struct evconnlistener *, evutil_socket_t, struct sockaddr *, int
       socklen, void *);
2
3  struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
4      evconnlistener_cb cb, void *ptr, unsigned flags, int backlog, const struct sockaddr *sa, int
           socklen);
```

- 第一个参数是很熟悉的 event_base，无论怎么样都是离不开 event_base 这个发动机的。
- 第二个参数是一个函数指针，该函数指针的格式如代码所示。当有新的客户端请求连接时，该函数就会调用。要注意的是：当这个回调函数被调用时，Libevent 已经帮我们 accept 了这个客户端。所以，该回调函数有一个参数是文件描述符 fd。我们直接使用这个 fd 即可。真是方便。这个参数是可以为 NULL 的，此时用户并不能接收到客户端。当用户调用 evconnlistener_set_cb 函数设置回调函数后，就可以了。
- 第三个参数是传给回调函数的用户参数，作用就像 event_new 函数的最后一个参数。
- 第四个参数 flags 是一些标志值，有下面这些：
  - LEV_OPT_LEAVE_SOCKETS_BLOCKING：默认情况下，当连接监听器接收到新的客户端 socket 连接后，会把该 socket 设置为非阻塞的。如果设置该选项，那么就把之客户端 socket 保留为阻塞的
  - LEV_OPT_CLOSE_ON_FREE：当连接监听器释放时，会自动关闭底层的 socket
  - LEV_OPT_CLOSE_ON_EXEC：为底层的 socket 设置 close-on-exec 标志
  - LEV_OPT_REUSEABLE: 在某些平台，默认情况下当一个监听 socket 被关闭时，其他 socket 不能马上绑定到同一个端口，要等一会儿才行。设置该标志后，Libevent 会把该 socket 设置成 reuseable。这样，关闭该 socket 后，其他 socket 就能马上使用同一个端口
  - LEV_OPT_THREADSAFE：为连接监听器分配锁。这样可以确保线程安全
- 第五个参数 backlog 是系统调用 listen 的第二个参数。最后两个参数就不多说了。

evconnlistener 的封装：接下来看一下 Libevent 是怎么封装 evconnlistener 的。用到的结构体：

```
1  // listener.c文件
2  struct evconnlistener_ops {///一系列的工作函数
3      int  (*enable)(struct evconnlistener *);
4      int  (*disable)(struct evconnlistener *);
5      void (*destroy)(struct evconnlistener *);
6      void (*shutdown)(struct evconnlistener *);
7      evutil_socket_t  (*getfd)(struct evconnlistener *);
8      struct event_base *(*getbase)(struct evconnlistener *);
9  };
10
```

```
struct evconnlistener {
    const struct evconnlistener_ops *ops;//操作函数
    void *lock;  //锁变量，用于线程安全
    evconnlistener_cb cb;//用户的回调函数
    evconnlistener_errorcb errorcb;//发生错误时的回调函数
    void *user_data;//回调函数的参数
    unsigned flags;//属性标志
    short refcnt;//引用计数
    unsigned enabled : 1;//位域为1.即只需一个比特位来存储这个成员
};

struct evconnlistener_event {
    struct evconnlistener base;
    struct event listener;  //内部event,插入到event_base
};
```

在 evconnlistener_event 结构体有一个 event 结构体。可以想象，在实现时必然是将服务器端的 socket fd 赋值给 struct event 类型变量 listener 的 fd 成员。然后将 listener 加入到 event_base，这样就完成了自动监听工作。这也回归到之前学过的内容。

下面看一下具体是怎么实现的。初始化服务器 socket：

```
struct evconnlistener *
evconnlistener_new_bind(struct event_base *base, evconnlistener_cb cb,
    void *ptr, unsigned flags, int backlog, const struct sockaddr *sa, int socklen) {
    struct evconnlistener * listener ;
    evutil_socket_t fd;
    int on = 1;
    int family = sa ? sa->sa_family : AF_UNSPEC;

    if (backlog == 0)
        return NULL;

    fd = socket(family, SOCK_STREAM, 0);
    if (fd == -1)
        return NULL;

    if (evutil_make_socket_nonblocking(fd) < 0) {
        evutil_closesocket(fd);
        return NULL;
    }

    if (flags & LEV_OPT_CLOSE_ON_EXEC) {
        if (evutil_make_socket_closeonexec(fd) < 0) {
            evutil_closesocket(fd);
            return NULL;
        }
    }

    if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, (void*)&on, sizeof(on))<0) {
        evutil_closesocket(fd);
        return NULL;
    }
    if (flags & LEV_OPT_REUSEABLE) {
        if (evutil_make_listen_socket_reuseable(fd) < 0) {
            evutil_closesocket(fd);
            return NULL;
        }
    }

    if (sa) {
        if (bind(fd, sa, socklen)<0) {
            evutil_closesocket(fd);
            return NULL;
```

```
        }
    }

    listener = evconnlistener_new(base, cb, ptr, flags, backlog, fd);
    if (! listener ) {
        evutil_closesocket(fd);
        return NULL;
    }

    return listener ;
}
```

evconnlistener_new_bind 函数申请一个 socket，然后对之进行一些有关非阻塞、重用、保持连接的处理、绑定到特定的 IP 和端口。最后把业务逻辑交给 evconnlistener_new 处理。

```
static const struct evconnlistener_ops evconnlistener_event_ops = {
    event_listener_enable,
    event_listener_disable,
    event_listener_destroy,
    NULL, /* shutdown */
    event_listener_getfd,
    event_listener_getbase
};

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

    if (backlog > 0) {
        if ( listen (fd, backlog) < 0)
            return NULL;
    } else if (backlog < 0) {
        if ( listen (fd, 128) < 0)
            return NULL;
    }

    lev = mm_calloc(1, sizeof(struct evconnlistener_event));
    if (! lev)
        return NULL;

    lev->base.ops = &evconnlistener_event_ops;
    lev->base.cb = cb;
    lev->base.user_data = ptr;
    lev->base.flags = flags;
    lev->base.refcnt = 1;

    if (flags & LEV_OPT_THREADSAFE) {
        EVTHREAD_ALLOC_LOCK(lev->base.lock, EVTHREAD_LOCKTYPE_RECURSIVE);
    }

    event_assign(&lev->listener, base, fd, EV_READ|EV_PERSIST,
        listener_read_cb, lev);

    evconnlistener_enable(&lev->base);

    return &lev->base;
}

int evconnlistener_enable(struct evconnlistener *lev) {
    int r;
    LOCK(lev);
```

```
48      lev->enabled = 1;
49      if (lev->cb)
50          r = lev->ops->enable(lev);
51      else
52          r = 0;
53      UNLOCK(lev);
54      return r;
55  }
56
57  static int event_listener_enable(struct evconnlistener *lev) {
58      struct evconnlistener_event *lev_e =
59          EVUTIL_UPCAST(lev, struct evconnlistener_event, base);
60      return event_add(&lev_e->listener, NULL);
61  }
```

几个函数的一路调用，思路还是挺清晰的。就是申请一个 socket，进行一些处理，然后用之赋值给 event。最后把之 add 到 event_base 中。event_base 会对新客户端的请求连接进行监听。

在 evconnlistener_enable 函数里面，如果用户没有设置回调函数，那么就不会调用 event_listener_enable。也就是说并不会 add 到 event_base 中。event_listener_enable 函数里面的宏 EVUTIL_UPCAST 可以根据结构体成员变量的地址推算出结构体的起始地址。有关这个宏，可以查看”结构体偏移量”。

处理客户端的连接请求：现在来看一下 event 的回调函数 listener_read_cb。

```
1   static void listener_read_cb(evutil_socket_t fd, short what, void *p) {
2       struct evconnlistener *lev = p;
3       int err;
4       evconnlistener_cb cb;
5       evconnlistener_errorcb errorcb;
6       void *user_data;
7       LOCK(lev);
8       while (1) {
9           struct sockaddr_storage ss;
10  #ifdef WIN32
11          int socklen = sizeof(ss);
12  #else
13          socklen_t socklen = sizeof(ss);
14  #endif
15          evutil_socket_t new_fd = accept(fd, (struct sockaddr*)&ss, &socklen);
16          if (new_fd < 0)
17              break;
18          if (socklen == 0) {
19              /* This can happen with some older linux kernels in
20               * response to nmap. */
21              evutil_closesocket(new_fd);
22              continue;
23          }
24
25          if (!( lev->flags & LEV_OPT_LEAVE_SOCKETS_BLOCKING))
26              evutil_make_socket_nonblocking(new_fd);
27
28          if (lev->cb == NULL) {
29              UNLOCK(lev);
30              return;
31          }
32          ++lev->refcnt;
33          cb = lev->cb;
34          user_data = lev->user_data;
35          UNLOCK(lev);
36          cb(lev, new_fd, (struct sockaddr*)&ss, (int)socklen,
37              user_data);
38          LOCK(lev);
```

```
39            if (lev->refcnt == 1) {
40                int freed = listener_decref_and_unlock(lev);
41                EVUTIL_ASSERT(freed);
42                return;
43            }
44            --lev->refcnt;
45        }
46        err = evutil_socket_geterror(fd);
47        if (EVUTIL_ERR_ACCEPT_RETRIABLE(err)) {
48            UNLOCK(lev);
49            return;
50        }
51        if (lev->errorcb != NULL) {
52            ++lev->refcnt;
53            errorcb = lev->errorcb;
54            user_data = lev->user_data;
55            UNLOCK(lev);
56            errorcb(lev, user_data);
57            LOCK(lev);
58            listener_decref_and_unlock(lev);
59        } else {
60            event_sock_warn(fd, "Error from accept() call");
61        }
62    }
```

这个函数所做的工作也比较简单，就是 accept 客户端，然后调用用户设置的回调函数。所以，用户回调函数的参数 fd 是一个已经连接好了的 socket。

上面函数说到了错误回调函数，可以通过下面的函数设置连接监听器的错误监听函数。

```
1   void evconnlistener_set_error_cb(struct evconnlistener *lev,
2                                    evconnlistener_errorcb errorcb) {
3       LOCK(lev);
4       lev->errorcb = errorcb;
5       UNLOCK(lev);
6   }
```

释放 evconnlistener：调用 evconnlistener_free 可以释放一个 evconnlistener。由于 evconnlistener 拥有一些系统资源，在释放 evconnlistener_free 的时候会释放这些系统资源。

```
1   void evconnlistener_free(struct evconnlistener *lev) {
2       LOCK(lev);
3       lev->cb = NULL;
4       lev->errorcb = NULL;
5       if (lev->ops->shutdown)
6           lev->ops->shutdown(lev);
7       listener_decref_and_unlock(lev);
8   }
9
10  static int listener_decref_and_unlock(struct evconnlistener *listener) {
11      int refcnt = --listener->refcnt;
12      if (refcnt == 0) {
13          listener->ops->destroy(listener);
14          UNLOCK(listener);
15          EVTHREAD_FREE_LOCK(listener->lock, EVTHREAD_LOCKTYPE_RECURSIVE);
16          mm_free(listener);
17          return 1;
18      } else {
19          UNLOCK(listener);
20          return 0;
21      }
22  }
```

```
23
24 static void event_listener_destroy(struct evconnlistener *lev) {
25     struct evconnlistener_event *lev_e =
26         EVUTIL_UPCAST(lev, struct evconnlistener_event, base);
27
28     event_del(&lev_e->listener);
29     if (lev->flags & LEV_OPT_CLOSE_ON_FREE)
30         evutil_closesocket(event_get_fd(&lev_e->listener));
31     event_debug_unassign(&lev_e->listener);
32 }
```

要注意一点，LEV_OPT_CLOSE_ON_FREE 选项关闭的是服务器端的监听 socket，而非那些连接客户端的 socket。

现在来说一下那个 listener_decref_and_unlock。前面注释说到，在函数 listener_read_cb 中，一般情况下是不会调用 listener_decref_and_unlock，但在多线程的时候可能会调用。这种特殊情况是：当主线程 accept 到一个新客户端时，会解锁，并调用用户设置的回调函数。此时，引用计数等于 2。就在这个时候，第二个线程执行 evconnlistener_free 函数。该函数会执行 listener_decref_and_unlock。明显主线程还在用这个 evconnlistener，肯定不能删除。此时引用计数也等于 2 也不会删除。但用户已经调用了 evconnlistener_free。Libevent 必须要响应。当第二个线程执行完后，主线程抢到 CPU，此时引用计数就变成 1 了，也就进入到 if 判断里面了。在判断体里面执行函数 listener_decref_and_unlock，并且完成删除工作。

总得来说，Libevent 封装的这个 evconnlistener 和一系列操作函数，还是比较简单的。思路也比较清晰。

9 evbuffer 与 bufferevent

10 log 日志系统及错误处理

11 其他

debug 系统
TAILQ_QUEUE 宏