# 概述

# 原理

Libevent确实方便了开发人员，对于定时器、信号处理、关心的文件或者套接字，只需要挂载到event\_base上面，设置好对应的回调函数和参数就可以了，当对应的事件发生时，Libevent会自动调度相应的回调函数进行处理。

## 创建event\_base

struct event\_base \* main\_base = event\_base\_new(void);

　　主要是根据参数，创建event\_base结构体，然后初始化一些数据，如果对默认的参数不满意需要个性化配置的话，可以先创建event\_config，然后调用event\_base\_new\_with\_config来创建。其中在eventops这个变量中，按照优先级顺序排序罗列了常见的IO复用模型，比如kqueue、epoll、poll、select等，由于Libevent是跨平台的，这些IO复用在有些平台可能是不可用的，同时你还可以在event\_config中选择过滤某些不想要的模型。

　　当选定了某个IO复用模型之后，其操作结构eventop就被添加到base->evsel中，然后调用其特定的init初始化函数。这些操作跟文件系统file\_operations结构极为的类似。

　　接着看看epoll类提供了哪些操作：

const struct eventop epollops = {

"epoll",

epoll\_init,

epoll\_nochangelist\_add,

epoll\_nochangelist\_del,

epoll\_dispatch,

epoll\_dealloc,

1, /\* need reinit \*/

EV\_FEATURE\_ET|EV\_FEATURE\_O1|EV\_FEATURE\_EARLY\_CLOSE,

0

};

在初始化函数epoll\_init当中，基本就类似epoll使用时候标准化的准备工作：首先调用epoll\_create创建epfd，然后预先创建INITIAL\_NEVENT(32)个空间用于存放epoll\_event，如果使用了timerfd，则再调用timerfd\_create创建对应的timerfd。最后**这些fd以及epoll\_event都存放在struct epollop当中，然后作为epoll\_init函数的返回保存在base->evbase上**。

struct epollop {

struct epoll\_event \*events; //数组

int nevents;

int epfd;

#ifdef USING\_TIMERFD

int timerfd;

#endif

};

在创建event\_base的最后，还调用了event\_base\_priority\_init进行了一个初始化操作，如果有多个优先级，就有对应的多个等待队列挂靠在base->activequeues上面，而base->nactivequeues记录了优先级的数目。

## 创建listen套接字，并建立connect事件侦听

### 基本过程

listener = evconnlistener\_new\_bind(srvopt.main\_base, accept\_conn\_cb, NULL,

LEV\_OPT\_CLOSE\_ON\_FREE|LEV\_OPT\_REUSEABLE, -1/\*backlog\*/,

(struct sockaddr\*)&sin, sizeof(sin));

这个算是个简化版的函数，你可以自己先手动建立和绑定socket，然后再调用evconnlistener\_new建立connect事件侦听。这个函数给socket设置了一个高大上的符号SO\_KEEPALIVE（SO\_KEEPALIVE 保持连接检测对方主机是否崩溃，避免服务器永远阻塞于TCP连接的输入），SOCKET开发还是有很多参数的，比如之前sshinner网络一直出问题，消息不能及时的被发送接收，最后跟踪shadowsockets-libev发现，是要给socket添加TCP\_NODELAY参数，问题才得以解决。

在evconnlistener\_new函数中，首先调用listen，然后分配evconnlistener\_event这个数据结构，base作为struct evconnlistener类型传递给用户空间，而listener主要作为内部隐藏的数据结构，为通用的struct event数据类型。

struct evconnlistener\_event {

struct evconnlistener base;

struct event listener;

};

struct evconnlistener {

const struct evconnlistener\_ops \*ops;

void \*lock;

evconnlistener\_cb cb;

evconnlistener\_errorcb errorcb;

void \*user\_data;

unsigned flags;

short refcnt;

int accept4\_flags;

unsigned enabled : 1;

};

struct event {

struct event\_callback ev\_evcallback;

/\* for managing timeouts \*/

union {

TAILQ\_ENTRY(event) ev\_next\_with\_common\_timeout;

int min\_heap\_idx;

} ev\_timeout\_pos;

evutil\_socket\_t ev\_fd;

struct event\_base \*ev\_base;

union {

/\* used for io events \*/

struct {

LIST\_ENTRY (event) ev\_io\_next;

struct timeval ev\_timeout;

} ev\_io;

/\* used by signal events \*/

struct {

LIST\_ENTRY (event) ev\_signal\_next;

short ev\_ncalls;

/\* Allows deletes in callback \*/

short \*ev\_pncalls;

} ev\_signal;

} ev\_;

short ev\_events;

short ev\_res; /\* result passed to event callback \*/

struct timeval ev\_timeout;

};

通过上面的数据结构可以清晰的发现，调用evconnlistener\_new\_bind函数作为参数提供的回调函数和参数，都被赋值给了evconnlistener\_event->base.cb和base.user\_data上面。接下来调用了两个比较重要的函数：

event\_assign(&lev->listener, base, fd, EV\_READ|EV\_PERSIST,

listener\_read\_cb, lev);

evconnlistener\_enable(&lev->base);

evconnlistener\_enable(&lev->base)通过追根述源是调用了event\_listener\_enable，最后调用了event\_add(&lev\_e->listener, NULL)。而event\_assign和event\_add都是比较重要的函数，event\_assign类似于event\_new的作用，只不过参数是一个已经初始化了的struct event，而event\_add则是把event由initialized状态变成pending状态，以便开始接收事件，其实后面可以发现，bufferevent\_enable等接口，底层也是调用的event\_add实现的。

### event\_assign调用

event\_assign(&lev->listener, base, fd, EV\_READ|EV\_PERSIST,

listener\_read\_cb, lev);

初看上面比较奇怪，在evconnlistener\_new这个函数的上半部分已经设置了一个base.cb和base.user\_data了，怎么下面又调用一个event\_assign来设置一个listener\_read\_cb回调呢？其实上面是用户提供的callback和args，但是这并没有直接跟某个事件相关联，而下面的event\_assign却是设置了&lev->listener(标准的struct event类型)为固定的listener\_read\_cb回调函数，当发生了EV\_READ就会被自动调用。然后在listener\_read\_cb中，我们发现：

cb = lev->cb; //

user\_data = lev->user\_data; //

cb(lev, new\_fd, (struct sockaddr\*)&ss, (int)socklen, user\_data);

errorcb(lev, user\_data);

所以说，其实Libevent内部根本没有什么诸如LISTEN的事件，还是用的标准EV\_READ(因为最最底层的epoll异步只能监听read/write/except事件)，只是做了个封装，当连接之后激活EV\_READ的回调函数，而出错了就调用err\_callback函数，所以accept\_conn\_cb实际是被手动调用的。

### event\_add调用

evconnlistener\_enable的调用被翻译到event\_add函数，其实不光光是这里，后面最常用的bufferevent\_enable这类函数，其实也是翻译到底层的event\_add函数(event\_add\_nolock\_)上面。

struct evconnlistener\_event \*lev\_e =

EVUTIL\_UPCAST(lev, struct evconnlistener\_event, base);

return event\_add(&lev\_e->listener, NULL);

#define EVUTIL\_UPCAST(ptr, type, field) \

((type \*)(((char\*)(ptr)) - evutil\_offsetof(type, field)))

event\_add有两个参数，后面一个参数是struct timeval的超时参数，如果是NULL就表示无限期等待，这里先就不考虑这种情况。其中最主要做的事情就是调用evmap\_io\_add\_/evmap\_signal\_add\_函数将事件加入到base当中：

// 对照上面，返回-1 error, 0 没有实际操作, 1 真实添加了

res = evmap\_io\_add\_(base, ev->ev\_fd, ev);

res = evmap\_signal\_add\_(base, (int)ev->ev\_fd, ev);

// 上面函数的核心操作

GET\_IO\_SLOT\_AND\_CTOR(ctx, io, fd, evmap\_io, evmap\_io\_init,

evsel->fdinfo\_len);

evsel->add(base, ev->ev\_fd,

old, (ev->ev\_events & EV\_ET) | res, extra);

首先，由于Libevent的设计是跨平台的，而Windows和Linux对socket和fd的表达和处理方式不同，GET\_IO\_SLOT\_AND\_CTOR的行为也有差异：

Windows使用的是hashtable维护着struct event\_map\_entry结构，而Linux平台就直接是用的指针数组（数组，元素类型是指针），用fd作为偏移来索引，指针指向的结构按需分配，十分的简洁高效。

/\* Used to map signal numbers to a list of events. If EVMAP\_USE\_HT is not

defined, this structure is also used as event\_io\_map, which maps fds to a

list of events.

\*/

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;

};

/\*\* Mapping from file descriptors to enabled (added) events \*/

struct event\_io\_map io;

/\*\* Mapping from signal numbers to enabled (added) events. \*/

struct event\_signal\_map sigmap;

在每一个event\_base结构体中，都定义了struct event\_signal\_map类型的两个成员io和sigmap(Linux平台)，用于信号量和FD与events之间的事件映射。然后看GET\_IO\_SLOT\_AND\_CTOR(GET\_SIGNAL\_SLOT\_AND\_CTOR)这个宏，查询或者创建fd对应的struct evmap\_io对象ctx，将当前的事件和之前的事件进行合并，并调用evsel->add进行更新(最终反应到底层epoll上面就是epoll\_ctl命令进行操作了)，并把当前的event结构体插入到前面找到的ctx->events链表当中。

## 建立主事件循环

无论是主线程，还是对于每个线程池用自己的event\_base，最终都会调用这个函数作为主循环进行事件处理。

event\_base\_loop(main\_base, 0);

先不考虑那些FLAG(控制何时推出啊啥的)，在event\_base\_loop主要做的事情是：

/\* F1 \*/ event\_queue\_make\_later\_events\_active(base);

/\* F2 \*/ res = evsel->dispatch(base, tv\_p);

/\* F3 \*/ timeout\_process(base);

F1的函数主要是在Libevent中引用了Deferred Callback机制，操作上就是从event\_base的active\_later\_queue队列中将事件取出来，然后添加到activequeues[evcb->evcb\_pri]对应优先级队列上面。

F2对于epoll类型，就是调用epoll\_dispatch函数：首先调用epoll\_apply\_changes(base);对event\_base->changelist上面挂靠的所有对fd的事件修改都执行底层修改使之生效；然后用res = epoll\_wait(epollop->epfd, events, epollop->nevents, timeout);获取被激活的事件；对获取到的每个fd的事件，提取被激活的事件类型，然后调用evmap\_io\_active\_(base, events[i].data.fd, ev | EV\_ET);函数处理。

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;

GET\_IO\_SLOT(ctx, io, fd, evmap\_io);

LIST\_FOREACH(ev, &ctx->events, ev\_io\_next) {

if (ev->ev\_events & events)

event\_active\_nolock\_(ev, ev->ev\_events & events, 1);

}

}

执行的结果就是，这个fd对应的evmap\_io上的所有事件，以及Deferred Callback事件，都被收集添加到event\_base->activequeues[evcb->evcb\_pri]队列中去。

于是乎，最后的好戏就是：

if (N\_ACTIVE\_CALLBACKS(base)) {

int n = event\_process\_active(base);

在event\_process\_active中调用了event\_process\_active\_single\_queue。当然作者考虑的细节还是比较细腻的：如果当前被调度的活动事件过多，就考虑在timer和maxcb两个维度上限制本轮的事件处理量，而在event\_process\_active\_single\_queue中，会不断从事件链表中取出事件处理（包括执行对应的回调函数）。从实现方式上看来，按照优先级从高到低的顺序，每一轮只处理一个最高优先级非空事件队列中的事件，然后就返回了。这样看来，如果高优先级的事件太多太活跃，那么低优先级的事件还是会有被饿死的风险。

## 基于bufferevent的普通socket读写事件

bufferevent使得网络的开发变的很方便，无论是从事件还是底层的evbuffer都提供了一套丰富灵活的接口。但是需要注意的是bufferevent目前只能用于TCP连接的类型，对于UDP只能手动建立struct event事件，然后设置事件和回调函数了，而且在回调函数中，一般也只能调用sendto/recvfrom等操作接口。

struct event\_base \*base = evconnlistener\_get\_base(listener);

struct bufferevent \*bev =

bufferevent\_socket\_new(base, fd, 0 /\*BEV\_OPT\_CLOSE\_ON\_FREE\*/);

bufferevent\_setcb(bev, bufferread\_cb, NULL, bufferevent\_cb, NULL);

bufferevent\_enable(bev, EV\_READ|EV\_WRITE);

上面算是在网络开发中用的最频繁的了，比如在listener的connection回调函数中，接收到一个新的套接字fd，那么就对这个套接字设置bufferevent事件，设置对应的回调函数。

在bufferevent\_socket\_new函数中：

struct bufferevent\_private \*bufev\_p;

struct bufferevent \*bufev;

bufferevent\_init\_common\_(bufev\_p, base, &bufferevent\_ops\_socket, options);

bufev\_p->bev;

event\_assign(&bufev->ev\_read, bufev->ev\_base, fd,

EV\_READ|EV\_PERSIST|EV\_FINALIZE, bufferevent\_readcb, bufev);

event\_assign(&bufev->ev\_write, bufev->ev\_base, fd,

EV\_WRITE|EV\_PERSIST|EV\_FINALIZE, bufferevent\_writecb, bufev);

evbuffer\_add\_cb(bufev->output, bufferevent\_socket\_outbuf\_cb, bufev);

evbuffer\_freeze(bufev->input, 0);

evbuffer\_freeze(bufev->output, 1);

跟之前的struct evconnlistener\_event一样，这里返回给用户空间可用的struct bufferevent也是struct bufferevent\_private的一部分，不过相比listener的单个struct event，这里的bufferevent内容成员要复杂的多，其中我们比较熟悉的有：input、output两个evbuffer，可以调用evbuffer的族函数进行相关的高级处理；此外还设置了be\_ops为bufferevent\_ops\_socket，而enabled使能的时间中默认为EV\_WRITE，所以EV\_READ需要手动enable；接下来设置bufev\_private->deferred的callback回调函数和调用参数为bufferevent\_run\_deferred\_callbacks\_locked和bufev\_private。

最后的两个event\_assign分别将ev\_read和ev\_write两个event的回调函数设置为了bufferevent\_readcb/bufferevent\_writecb。由于EV\_WRITE默认是使能的，所以还调用了evbuffer\_add\_cb设置其默认的回调函数为bufferevent\_socket\_outbuf\_cb。为了安全，还将两个evbuffer先冻结起来，准备工作还未就绪，所以此时还不允许数据传输。

其实，正如上面的例子，对于bufferevent，通常的写操作就使用其默认的callback就可以了，实际开发当中我们最关心的是读事件，因为我们要接收数据处理数据（即便只是转发操作），而写数据只要准备好要发送的数据，底层的写就让其自动处理就可以了。

说到底，这里的bufferevent和evconnlistener类似，也是采用了两段式设计：在bufferevent中的ev\_read/write被激活调度的时候，其自动执行的是bufferevent\_readcb/writecb函数，在这些标准函数中会做一些的预先处理操作，比如evbuffer\_read/evbuffer\_write\_atmost的读写，到最后通过bufferevent\_trigger\_nolock\_调用用户设定的回调函数。然后你可能感兴趣EV\_WRITE默认的bufferevent\_socket\_outbuf\_cb干了啥？查看其代码，其实也就是：检查确保当前ev\_write是否是pending的，如果不是就bufferevent\_add\_event\_变成pending状态就好了。我们知道，epoll事件最底层是操作系统直接驱动的，所以如果底层驱动发现socket是可写的，就可以调度底层发送数据了，因此这个函数实际上其实啥都没做。

最后的bufferevent\_enable(bev, EV\_READ|EV\_WRITE);跟之前的evconnlistener\_enable也大差不差的，最终都是通过be\_socket\_enable->bufferevent\_add\_event\_->event\_add方式，将event加入到对应的event\_base上面去，使之变为pending状态。

# 应用

## Memcache

### 原理

1、main [memcached.c]

memcached启动时候执行memcached.c中的main函数，在加载了好长的初始化配置之后，定义并初始化event\_base；

static struct event\_base \*main\_base;

main\_base = event\_init();

然后通过调用memcached\_thread\_init，创建工作者线程：

memcached\_thread\_init(settings.num\_threads, main\_base);

创建定时器clock\_handler(0, 0, 0);，这个基于Libevent创建的定时器每一秒钟执行一次，用以更新current\_time这个表示自从进程启动后的时间长度。

然后针对服务端参数指定的侦听(ip:port/unix socket)类型，分别调用server\_socket\_unix/server\_sockets函数，绑定指定地址，并为创建的socket添加connect事件，核心代码如下

// unix socket

listen\_conn = conn\_new(sfd, conn\_listening, EV\_READ|EV\_PERSIST,1,

local\_transport, main\_base)))

// tcp

listen\_conn\_add=conn\_new(sfd,conn\_listening,EV\_READ|EV\_PERSIST,1,

transport, main\_base))

这个conn\_new不仅仅在这里用以侦听套接字分配资源、创建事件侦听，之后所有客户端连接的套接字也会用这个函数。这个函数最终回调的响应函数是event\_handler，然后最终调用函数drive\_machine，这个函数内部是一个复杂的有限状态机，会处理所有与套接字相关的连接、关闭、读写等操作。

listen套接字当接收到客户请求的时候，如果连接OK，并且没有超过最大连接数目，就调用dispatch\_conn\_new接收请求。这个函数中，会轮询选择要添加的工作线程，然后创建一个等待item，并添加到对应线程的new\_conn\_queue队列上去，然后向这个线程的读取队列里面写入’c’一个字节表明有一个新的请求，然后对应线程管道读事件就会被触发，执行处理回调函数。

主线main\_base进入Libevent事件循环中：

/\* enter the event loop \*/

if (event\_base\_loop(main\_base, 0) != 0) {

retval = EXIT\_FAILURE;

}

2、memcached\_thread\_init [thread.c]

上面我们关注的核心在于调用memcached\_thread\_init这个函数创建nthreads个工作者线程。

typedef struct {

pthread\_t thread\_id; /\* unique ID of this thread \*/

struct event\_base \*base; /\* libevent handle this thread uses \*/

struct event notify\_event; /\* listen event for notify pipe \*/

int notify\_receive\_fd; /\* receiving end of notify pipe \*/

int notify\_send\_fd; /\* sending end of notify pipe \*/

struct thread\_stats stats; /\* Stats generated by this thread \*/

struct conn\_queue \*new\_conn\_queue; /\*queue of new connections to handle\*/

cache\_t \*suffix\_cache; /\* suffix cache \*/

} LIBEVENT\_THREAD;

void memcached\_thread\_init(int nthreads, struct event\_base \*main\_base) {

...

threads = calloc(nthreads, sizeof(LIBEVENT\_THREAD));

dispatcher\_thread.base = main\_base;

dispatcher\_thread.thread\_id = pthread\_self();

for (i = 0; i < nthreads; i++) {

int fds[2];

threads[i].notify\_receive\_fd = fds[0];

threads[i].notify\_send\_fd = fds[1];

setup\_thread(&threads[i]);

/\* Reserve three fds for the libevent base, and two for the pipe \*/

stats.reserved\_fds += 5;

}

/\* Create threads after we've done all the libevent setup. \*/

for (i = 0; i < nthreads; i++) {

create\_worker(worker\_libevent, &threads[i]);

}

...

}

上面把非核心的代码剔除掉，就可以看清memcached\_thread\_init所做的具体工作了。

1）为每个线程创建LIBEVENT\_THREAD结构体，并把自我分发线程的信息记录在dispatcher\_thread中；

2）对每个线程的结构体LIBEVENT\_THREAD初始化，然后通过pipe创建匿名管道，pipefd[0]指向读端，而pipefd[1]指向写端，然后每个线程就通过这个匿名管道同其他线程进行通信；

static void setup\_thread(LIBEVENT\_THREAD \*me) {

...

me->base = event\_init();

/\* Listen for notifications from other threads \*/

event\_set(&me->notify\_event, me->notify\_receive\_fd,

EV\_READ | EV\_PERSIST, thread\_libevent\_process, me);

event\_base\_set(me->base, &me->notify\_event);

event\_add(&me->notify\_event, 0);

me->new\_conn\_queue = malloc(sizeof(struct conn\_queue));

cq\_init(me->new\_conn\_queue);

me->suffix\_cache = cache\_create("suffix", SUFFIX\_SIZE, sizeof(char\*), NULL, NULL);

...

}

其中的setup\_thread函数中，为每一个线程创建一个event\_base，然后添加之前管道的读写时间侦听；同时每个线程还创建了一个等待队列，所有的新请求会添加到这个等待队列上面去。

在匿名管道的读写事件的相应函数thread\_libevent\_process上，会尝试读取一个字节,如果是上面写入的’c’，就表明有待处理的请求，然后就从等待队列new\_conn\_queue中取出一个item，然后处理。处理的方式就是确认这个连接，分配相应的资源，然后再丢到上面的那个event\_handler->drive\_machine的状态机中去!

3）调用create\_worker(worker\_libevent, &threads[i]);实行真正创建线程操作，其内部就是一个pthread\_create；

### 总结

Memcached工作方式可以描述如下：

软件启动的时候，创建event\_base，并且根据设置类型创建侦听的tcp/udp socket或者unix socket，然后为这些套接字创建读侦听事件，加入到event\_base上，等待客户端连接；

创建工作线程池，每个工作线程创建自己的event\_base；创建一个等待队列，新连接的客户请求都会挂在这个队列上；创建一个匿名管道，并为管道创建读写侦听事件；

当新的客户端连接上来有请求时候，主线程的侦听事件回调函数会被激活，条件满足后接受这个连接，然后选取一个工作线程，创建等待item挂到其队列上，然后向其管道写入一个c，对应线程管道读事件被激活，读取一个c，并从队列中取出一个请求处理；

Memcache对所有socket的处理都是event\_handler->drive\_machine中处理的。

可以说，memcached在线程池在等待连接和事件处理中都充分利用了Libevent的异步事件，所以效率是非常之高的。自己的那个线程池，主要是将所有的任务都放到一个链表队列中，当线程发现没有任务的时候，就会用pthread\_cond\_wait阻塞睡眠，当主线程发现等待的任务太多，就会用pthread\_cond\_signal唤醒睡眠线程（不会惊群）。

总体会让人感觉，把任务事先分给各个队列，吞吐量要大一些，让任务阻塞在select、poll、epoll上，会比自己控制睡眠唤醒要高效可靠！

## TDSQL