Libonev模块设计文档

修订历史

版本	修订日期	修订描述	作者	备注
0.1	2016-01-23	基于libev的网络接口封装设计	肖冰	无

1 系统设计

1.1 综述

libonev是基于libev开源网络库封装的一套网络功能接口，主要提供网络读写功能，管理并维护I/O线程池。借鉴OceanBase0.4版本中使用的libeasy，旨在利用libev的事件驱动模型实现多线程收发包、处理IO，结合socket协议封装网络读写，形成一套高并发的事件库，使得OceanBase可以更方便的处理网络请求。特别是针对小数据包IO请求数较多的情况，多IO线程的处理模型能够保证数据包的处理能力，每个线程收到网络包后立即处理，减少上下文切换。

如OB中，UPS有多个网络读写线程，每个线程通过Linux epoll监测一个socket集合上的网络读写事件。每个socket只能同时分配一个线程。当网络读写线程收到网络包后，立即调用任务处理函数，如果处理时间很短，可以很快完成并回复客户端，不需要加锁，避免了上下文切换。UPS中大部分任务为短任务，比如随机读取内存表。另外还有少量任务需要等待共享资源上的锁，可以将这些任务加入到长任务队列中，交给专门的长任务处理线程处理。

在高性能服务器设计中，可以说事件库决定了系统的整体性能. libev饱受赞誉，部署在IBM Cloud和Amazon EC2环境中的应用程序都有使用。它除了将IO事件、定时器和信号统一放在事件这一套框架下处理，还支持自己的一些watcher，支持一些复杂的功能。对libev的使用专注于对事件模型的设计，各个事件在event loop中注册不同的观测器，响应不同的子事件，通过回调函数进行读写等操作。libonev主要使用以下libev定义的watcher类型：

watcher类型	简介
ev_io	IO可读可写，用于读写观测器
ev_async	激活线程， 用于线程观测器
ev_timer	定时器，用于监听观测器与超时观测器

向libev注册感兴趣的事件，比如socket可读事件，libev会对所注册的时间的源进行管理，并在事件发生时触发相应的程序。libonev对此封装了事件模型与网络协议，关键的数据结构包括：

数据结构	介绍	对libev的封装
onev_connection_e	封装TCP连接	event loop, 读写观测器，超时观测器
onev_baseth_e	封装基础线程信息，包括io信息	event loop, 线程观测器
onev_io_thread_e	io线程，包含基础线程信息，用于监听	event loop, 线程观测器，监听观测器
onev_request_e	对应应用层一个具体的包，贯穿整个请求的处理流程	无
onev_session_e	封装发往服务器端的请求包	超时观测器
onev_listen_e	用于管理对端口的监听，关联io线程	读观测器
onev_pool_e	内存池，模仿nginx的内存池实现，非全局可以有多个	无
onev_buf_e	用于管理连接的输入输出缓冲区	无

OceanBase在服务器启动和请求处理的时候封装libonev的IO线程以及自己定义的处理方法，其中IO线程的创建和连接的建立在服务器启动时全部初始化完毕，接下来响应连接上的请求，按服务器和客户端触发不同的请求处理函数。如MS在响应MySQL客户端的请求时将libonev作为服务器端使用其IO线程池和自己的工作线程池。

由于每个网络读写线程处理一部分预先分配的套接字，这就可能出现某个套接字上请求特别多而导致负载不均衡的情况。libonev同样实现网络模块内部自动在网络读写线程之间执行负载均衡的功能，将套接字从负载较高的线程切换到负载较低的线程。

1.2 名词解释

- libev: 一个开源的事件驱动库，基于epoll，kqueue等操作系统基础
- watcher: 观测器，libev定义，不同的事件通过不同的观测器完成响应
- event loop: 注册观测器的事件驱动框架，libev定义，框架内通过观测器结构和相应的接口箱注册观测器
- nginx: 面向性能设计的轻量级HTTP服务器，特点为占用内存小、并发能力强
- MS: MergerServer， 接收并解析用户的SQL请求， 需要向其他server发送查询计划
- UPS: UpdateServer, 存储增量更新数据， 主备之间需要同步

1.3 功能

- 处理io，包括io的创建、释放，io线程的启动，信号的处理，io的启动、等待与停止，统计io状态信息。
- 处理连接，包括连接的建立、断开，资源的释放，添加监听端口，接受连接，发送请求，分别按照客户端和服务端不同角色处理请求，超时控制，线程切换，负载均衡，回调唤醒。
- 处理线程池，包括基础线程的创建，事件循环的分配，初始化观测器，线程池的创建，线程与线程池的唤醒。
- 处理请求，包括请求的创建、销毁，请求的处理，唤醒异步观测器，添加缓冲区，同步异步发送请求。
- 处理日志，包括按照不同的日志级别输出日志。

2 模块设计

libonev模块主要分为两个子模块来封装。其中，element模块集成了网络框架中的基础数据结构，包括内存管理、哈希处理、线程池管理和时间管理等。conn模块封装了网络交互的具体方法，并对外提供使用网络框架的接口，包括对io、对请求、对连接、对线程池以及对日志的处理方法。下面对element模块以及conn的核心子模块–io模块和connection模块进行设计的详细介绍。

2.1 element子模块设计

libonev同OceanBase0.4版本中的网络库一样采用每个线程一个event loop的设计模式，内存资源按连接(connection)进行管理，连接之间的资源互不干涉。每个连接上可以有多个消息(message)通过链表连接起来，每个消息又可以有多个请求(request)组成，同样通过链表连接起来。 各结构上的启动及处理通过函数指针的形式来管理。

libonev的链表实现借鉴Linux内核的链表实现。内存池的管理借鉴nginx的实现。

2.1.1 关键结构

一个request相当于一个请求包，贯穿着请求的输入、处理和输出整个流程。包含请求的输入输出队列，以及发送时所在的message或session信息即连接和内存池信息。

一个session封装一个客户端发往服务器端的请求包，添加其所在的连接、地址、内存池和io线程，携带数据包标识符，设置处理函数和超时观测器。

一个message封装多个请求包，并设置用于解包的输入缓冲区，对应连接的输入缓冲区。

基础线程封装io线程的基础信息，包括线程标识符，事件循环，线程观测器，锁以及启动和处理函数指针。

io线程除了基础线程信息外，还注册监听观测器，关联各状态的连接与请求，如已建立但尚未监听的连接，已建立并且监听的连接，已经完成但是没有发送的请求，以及完全发送完毕的请求，统计已经完成已经正在处理的请求数量，维护客户端成员链表。

io线程上的每一个连接的事件循环体都对应到该线程的事件循环，每一个连接设置自己的读写观测器以及超时观测器，在连接建立时注册回调函数。每一个连接还需要设置socket协议参数，关联socket读写函数，维护请求队列、客户端队列、输出缓冲区以及统计信息。

io_handler集成用户实现的函数，包括封包解包、处理数据包、连接与断开连接、获取数据包标识符等等。

listen结构体设置读观测器，当listen fd上有可读事件时，IO线程相应的read_watcher会被触发，从而回调accept函数接收连接，切换线程继续监听。

2.2 io子模块设计

2.2.1 结构

io模块是应用程序的入口模块，也是整个网络框架处理流程的启动模块。每一次流程启动后，将产生一个全局的io链表，每一个io结构关联到一个io线程，设置自己的socket协议参数以及负载保护参数，关联两个监听，一个用于正常的io处理，一个用于线程切换。

2.2.2 关键处理方法

io初始化时对其线程池中的每个io线程进行初始化，设置监听观测器每100ms触发一次listen事件切换监听。后续线程切换时，重置观测器并启动可以改变监听观测器的触发频率。

io线程的执行体函数，是启动事件循环的地方，当io只有一个线程并且被监听或者io监听多线程并且端口重用时，即启动读观测器，此时的读观测器设置了accept回调函数，当有可读数据时被触发，进而进入connection核心处理模块进行连接的建立与请求的处理。

io模块还对外提供io的启动、等待、停止以及资源释放接口。默认io启动时，启动信号处理机制，用于反应用户动作，创建并启动系统线程。等待io时，检测其线程池中各西安城之间是否存在死锁。停止和释放资源时，均对线程池中的每个线程执行停止并释放资源。

2.2.3 流程

初始化io线程池中的每个线程和相应的数据结构。

针对OB的服务器端封装：

1. libonev的IO处理线程从connection上读客户端发送的请求数据
2. 调用io_handler的decode方法从缓冲区中解析获得OB需要的packet
3. 调用io_handler的process方法处理

针对OB的客户端封装：

1. 将每个发往libonev服务器端的请求包都封装成一个session,客户端将这个session放入连接的队列中然后返回
2. 收到包后，客户端再将这个session从连接的发送队列中删除，这里使用hash加快查找
3. libonev由于采用了libev实现的watcher机制，超时粒度基于包

2.3 connection子模块设计

connection模块是网络请求的核心处理模块，几乎封装所有的事件循环逻辑与回调处理流程。从创建连接到接收连接，从增加监听到切换线程，从发送请求到处理请求，围绕事件驱动模型的响应模式，设计回调函数功能，在不同的element上注册，在适当的时间点启动，当事件发生时被触发进而完成处理。

2.3.1 回调函数

针对一个标准的Client-Server网络服务模型，设计关键的回调函数如下：

回调函数	注册观测器	功能描述
accept	监听的读观测器	监听端口时注册，io线程启动时启动，被可读事件触发，执行socket accept，新建连接并初始化读写观测器和超时观测器，切换监听
wakeup	线程观测器	线程初始化时注册并启动，io线程收到信号后被触发，唤醒io线程启动所有连接上的观测器，最终发送处理完的请求
read	连接的读观测器	建立或接受连接时注册，读数据时启动，连接的输入缓冲区不为空时触发，新建message，执行socket read从连接读数据到message，处理请求
write	连接的写观测器	建立或接受连接时注册，写数据时启动，服务端连接的输出缓冲区不为空时触发，将缓冲区写到socket, 如果是在连接上的客户端，就启动该链接的读观测器和超时观测器；如果是在连接上的服务端，并且请求队列不为空，则处理请求后再次启动写观测器
timeout	连接的超时观测器	建立或接受连接时注册，读写数据或处理请求时启动，计时器超时触发，客户端连接的请求列表为空时停止

2.3.2 关键处理方法

listen操作定时抢占listen的锁，抢到锁就把当前的listen线程改成自己，并添加这个线程的EV_READ事件，以当数据到来时再次回调accept函数。如果抢不到锁就停止对EV_READ事件的监听。

线程池中的某线程触发accept对listenfd做accept操作，接受fd的socket连线成功后，会创建connection对象，并注册对这个fd的读写超时事件。

处理读事件，当fd可读时触发readable回调函数，新建一个message对应用户解包函数的一个单位。数据读到message后会判断当前是服务器端还是客户端，如果是客户端会调用do_response来处理请求，如果是服务器端会调用do_request来处理请求。

服务器端处理messege首先调用用户实现的解包函数进行解包，即对message的缓冲区解析数据。对于每一个解析出来的数据包，都会创建request对象，把数据包挂到request的输入包队列中去，之后将request挂到message的请求队列当中去，然后调用process request 。

对于需要工作线程处理的请求，使用process request方法，遍历message的请求队列，从message中取下request，调用用户实现的处理函数处理链表中的每一个请求。对于每一次处理完成后，调用request done，如果requeset的输出队列上不为空，即客户端需要得到响应，此时调用用户实现的封包函数，之后设置此次请求的cleanup方法。request done执行完毕后，调用write socket把数据通过socket写出去。

2.3.3 TCP流程

初始化io及用户回调函数集合之后，首先要增加监听地址，打开监听端口，把线程池中的每个线程都加上对这个listenfd的读事件，将回调函数设置为accept操作，使得在开始阶段，线程池的每个线程都会响应accept事件。之后线程池中的某个线程触发对listenfd的accept操作，接受fd的socket连线成功后，新建连接，在该连接的event loop上注册相关事件如读、写和超时事件。调用用户实现的回调函数on_connect后，切换listen实现线程间的负载均衡。当socket有数据到来时，会触发读事件的回调函数。数据全部读到定义的数据结构当中以后，服务器/客户端会启动相应的事件观测器响应请求。

3 模块接口

3.1对外接口

服务器或客户端在启动前调用onev_create_io函数创建io对象。完成存放io对象的内存空间的分配以及线程池规模的设置。同时设置TCP参数及负载保护参数，并初始化io线程池中的每个线程，为每一个线程分配一个event loop.

```
/**
 * @brief onev_create_io
 * Initialize io object, including its thread pool.
 * @param io, the io object that is going to be initialized
 * @param io_thread_count, the count of io thread
 * @return io or NULL
 */
onev_io_e
*onev_create_io(onev_io_e *io, int io_thread_count)
```

服务器端调用onev_io_set_uthread_start函数为io线程设置用户线程启动函数和参数。

```
/**
 * @brief onev_io_set_uthread_start
 * Set user thread start function for listen.
 * @param eio
 * @param on_utstart,implementated in user programs if needed
 * @param args, usurally 'this'
 */
void
onev_io_set_uthread_start(onev_io_e *eio, onev_io_uth_start_pe *o
n_utstart, void *args)
```

在启动io之前要通过调用onev_connection_add_listen打开监听端口.初始化读观测器，关注监听的读事件，设置回调函数。

```
/**
 * @brief onev_connection_add_listen
 * Add listen port. Called when start io.
 * @param io, the starting io object
 * @param host, server name or IP address or NULL
 * @param port
 * @param handler
 * @return connection or NULL
 */
onev_listen_e
*onev_connection_add_listen(onev_io_e *io, const char *host, int po
rt, onev_io_handler_pe *handler)
```

调用onev_start_io将io对象中所有线程池上的所有线程都启动。

```
/**
 * @brief onev_start_io
 * React signals.
 * Start all threads in all thread pools of io.
 * @param io
 * @return ONEV retcode
 */
int
onev_start_io(onev_io_e *io)
```

调用onev_stop_io函数停止io，唤醒所有io线程的线程观测器。

```
/**
 * @brief onev_stop_io
 * Stop io.
 * @param io
 * @return ONEV retcode
 */
int
onev_stop_io(onev_io_e *io)
```

调用onev_wait_io函数等待io，检测到线程资源发生死锁时，abort退出进程。

```
/**
 * @brief onev_wait_io
 * Wait io. Use spinlock.
 * @param io
 * @return ONEV retcode
 */
int
onev_wait_io(onev_io_e *io)
```

调用onev_destroy_io函数，将io从所在链表中删除，关闭所有监听，销毁所有线程并最后释放空间。

```
/**
 * @brief onev_destroy_io
 * Release io resource and space.
 * @param io
 */
void
onev_destroy_io(onev_io_e *io))
```

使用onev_pool_alloc接口，实际调用内部onev_pool_alloc_ex函数，从制定pool中分配制定大小的空间。

```
///allocate $size of memory from pool
#define onev_pool_alloc(pool, size)  onev_pool_alloc_ex(pool, size, sizeof(long))
```

调用onev_request_wakeup唤醒请求所在io线程的异步观测器。

```
/**
 * @brief onev_request_wakeup
 * Activates the async thread watcher.
 * @param r,the request to be sent
 */
void
onev_request_wakeup(onev_request_e *r)
```

调用onev_create_session新建session对象用于发送，并关联一个request。

```
/**
 * @brief onev_create_session
 * New a session. Associate with request.
 * @param size
 * @return session or NULL
 */
onev_session_e
*onev_create_session(int64_t size)
```

调用onev_session_set_timeout接口为session设置timeout,注意类型为ev_tstamp。

```
///type of session timeout: ev_tstamp
#define onev_session_set_timeout(s, t)  (s)->timeout = t
```

调用onev_client_dispatch函数将session发送到指定的io线程上去。计算合法的线程号，通过线程号选取线程，将session挂到线程的session链表当中去。最后激活该线程的异步观测器。

```
/**
 * @brief onev_client_dispatch
 * Compute index and get io thread.
 * Send session to address.
 * Signal async io thread watcher.
 * @param io, where to select io thread
 * @param addr, target address
 * @param s, session to be sent
 * @return ONEV retcode
 */
int
onev_client_dispatch(onev_io_e *io, onev_addr_e addr, onev_sessio
n_e *s)
```

调用onev_destroy_session通过销毁内部message的方式销毁session对象。

```
/**
 * @brief onev_destroy_session
 * Destroy session through messages.
 * @param data, pointer of the session to be destroyed.
 */
void
onev_destroy_session(void *data)
```

客户端向服务端发包时调用onev_client_send以同步的方式发送session，等待返回结果。

```
/**
 * @brief onev_client_send
 * Used when posting packet.
 * Synchronize. Wait for returned.
 * @param io, used in dispatch client
 * @param addr, target addr, used in dispatch client
 * @param s, to be sent
 * @return input packet
 */
void
*onev_client_send(onev_io_e *io, onev_addr_e addr, onev_session_e
*s)
```

为发送缓冲区分配内存后，需要根据包的大小调整分配的缓冲区设置，调用onev_buf_set_data根据data重新设置buffer.

```
/**
 * @brief onev_buf_set_data
 * Set data to buffer.
 * Associate with pool.
 * @param pool
 * @param b, the buffer needs setting
 * @param data, modify the buffer size
 * @param size, from configuration
 */
void
onev_buf_set_data(onev_pool_e *pool, onev_buf_e *b, const void *dat
a, uint32_t size)
```

缓冲区分配好以后调用onev_request_addbuf将缓冲区添加到具体的request对象当中去。

```
/**
 * @brief onev_request_addbuf
 * Push buffer to request(session)
 * @param r, request
 * @param b, buffer
 */
void
onev_request_addbuf(onev_request_e *r, onev_buf_e *b)
```

同步实现网络收发时，调用onev_client_wait_init对阻塞对象进行初始化，初始化链表成员变量，以及互斥锁和条件变量。

```
/**
 * @brief onev_client_wait_init
 * For synchronization.
 * Initialize wait object member vars, mutex and condition vars.
 * @param w, wait object
 */
void
onev_client_wait_init(onev_client_wait_e *w)
```

同步网络收发完成后，调用onev_client_wait_cleanup释放阻塞变量的资源，包括其链表成员变量。

```
/**
 * @brief onev_client_wait_cleanup
 * Release wait object. Including its list member.
 * @param w, wait object.
 */
void
onev_client_wait_cleanup(onev_client_wait_e *w)
```

客户端同步处理服务器端发包时，阻塞线程通过onev_client_wait_wakeup_request唤醒其他线程。同步网络收发中，某请求唤醒其他线程也主要通过其阻塞成员wobj进行，实际调用onev_client_wait_wakeup来在互斥的保护下执行。

```
/**
 * @brief onev_client_wait_wakeup_request
 * Call onev_client_wait_wakeup after updating reference.
 * To wakeup another thread once.
 * @param r, request
 */
void
onev_client_wait_wakeup_request(onev_request_e *r)
```

在互斥保护状态下，onev_client_wait_wakeup在阻塞变量维护中唤醒其他线程。

```
/**
 * @brief onev_client_wait_wakeup
 * Wakeup another thread under wait object.
 * @param w, wait object, usually from request
 */
void
onev_client_wait_wakeup(onev_client_wait_e *w)
```

删除连接后需要关闭socket，调用onev_connection_destroy_dispatch关闭单道读功能

```
/**
 * @brief onev_connection_destroy_dispatch
 * Close read function on the TCP socket of a connection.
 * @param c, connection.
 * @return 0
 */
int
onev_connection_destroy_dispatch(onev_connection_e *c)
```

对引用计数等需要进行原子加1时，调用onev_atomic_inc

```
///atomic increase,on,such as reference count
#define onev_atomic_inc(v) onev_atomic32_inc(v)
```

对pool空间加锁，简单将pool的flags变量设为1

```
/**
 * @brief onev_pool_set_lock
 * Simply set flags = 1.
 * @param pool
 */
void
onev_pool_set_lock(onev_pool_e *pool)
```

需要将libonev定义的地址转换成字符串时，调用onev_inet_addr_to_str函数，同时传入缓冲区及缓冲区大小。

```
/**
 * @brief onev_inet_addr_to_str
 * Convert onev address to string.
 * Under Ipv4 and Ipv6 domain separately.
 * @param addr, onev address
 * @param buffer, for print
 * @param len, buffer size
 * @return
 */
char
*onev_inet_addr_to_str(onev_addr_e *addr, char *buffer, int len)
```

调用onev_list_init函数采用Linux内核链表的实现方法初始化链表成员。

```
#define onev_list_init(ptr) do{  \
            (ptr)->prev = (ptr); \
            (ptr)->next = (ptr); \
}while (0)
```

输出日志时需要用到libonev对格式的处理，调用onev_localtime获得本地当前时间表达式。传入当前日志时间和分割时间，后者存储格式处理后的时间信息。

```
/**
 * @brief onev_localtime
 * No tzset in localtime for once call outside.
 * @param t, Calendar Time
 * @param tp, Broken-down Time
 * @return 1 if success, 0 if fail
 */
int
onev_localtime (const time_t *t, struct tm *tp)
```

调用onev_log_print_default函数写日志到文件。

```
/**
 * @brief onev_log_print_default
 * Write all content to file.
 * @param message, content
 */
void
onev_log_print_default(const char *message)
```

3.2内部回调函数

```
/**
 *Add listen through address.
 */
onev_listen_e *onev_add_listen_addr(onev_io_e *io, onev_addr_e add
r,onev_io_handler_pe *handler, int udp, void *args)
```

每个io线程的内部执行体函数。是onev_start_io内部执行函数，唯一启动事件循环之处。

```
/**
 * @brief onev_io_on_thread_start
 * Executor of io thread.
 * @param args
 * @return
 */
static void
*onev_io_on_thread_start(void *args)
```

连接处理中被监听事件触发，注册相关事件，触发连接的创建。

```
/**
 * Accpet connection. Process event.
 */
void onev_connection_on_accept(struct ev_loop *loop, ev_io *w, int
revents)
```

```
/**
 * Executor in onev_connection_on_accept()loop
 */
static int onev_connection_accept_one(struct ev_loop *loop, ev_io
*w)
```

4 使用限制条件和注意事项

Linux环境下需安装libtool。