**C1000K之Libevent源码分析**

10 Dec 2014

**简介**

说到异步IO,高并发之类的名词， 可能很多人第一反应就是 select, poll, epoll, kqueue 之类的底层代码库。 但是其实除非你要写一个 Nginx 性能级别的服务器， 否则直接使用 epoll 之类的还是太过底层， 诸多不便，要榨干整个异步编程的高并发性能还需要开发很多相关组件， 而 [Libevent](https://github.com/nmathewson/Libevent) 就是作为更好用的高性能异步编程网络库而生， 他帮你包装了各种 buffer 和 event， 甚至也提供了更加高层的 http 和 rpc 等接口， 可以让你脱离底层细节，更加专注于服务的其他核心功能的实现。 当然，要真正用好它，还是需要懂不少关于他的实现原理。

如果是第一次接触 [Libevent](https://github.com/nmathewson/Libevent) 的可以先看一篇非常好的入门文章： [Libevent-book](http://www.wangafu.net/~nickm/libevent-book) ， 文章主要从 C10K 问题的发展循序渐进， 分别讲了在高并发连接的情况下， 多线程解决方案， 多进程解放方案会遇到的问题， 从而引出为什么异步IO是当前解决高并发连接最有效的方案。

[ideawu](http://www.ideawu.net/) 实现的 C1000K 服务器 [icomet](http://github.com/ideawu/icomet) 核心就是基于 [Libevent](https://github.com/nmathewson/Libevent) 实现的。

本文源码分析基于 [Libevent](https://github.com/nmathewson/Libevent) master 分支 commit 6dba1694c89119c44cef03528945e5a5978ab43a 版本的代码。

**事件循环**

既然是异步IO，事件驱动，自然会有事件循环(event loop) 。 [Libevent](https://github.com/nmathewson/Libevent) 的事件循环是通过调用 event\_base\_dispatch 来实现， 其实 event\_base\_dispatch 函数也是调用了 event\_base\_loop， 代码如下：

**int**

event\_base\_dispatch(**struct** event\_base \*event\_base)

{

**return** (event\_base\_loop(event\_base, 0));

}

运行事件循环的函数肯定是阻塞函数， 拿 linux 平台来说， libevent 的事件循环其实就是循环调用 epoll\_wait 函数，

**int** epoll\_wait(**int** epfd, **struct** epoll\_event \*events,

**int** maxevents, **int** timeout);

在没有任何事件被触发的时候，epoll\_wait 是阻塞等待的， 而且，在 [Libevent](https://github.com/nmathewson/Libevent) 里，定时器的实现其实就是通过 epoll\_wait 的 timeout 参数实现的。

如果只有一个单线程的话，一旦调用了 event\_base\_dispatch 之后，这个线程就会被事件完完全全的霸占， 无法进行任何其他的操作。

如果我们需要临时手动激活任何其他事件的话， 则需要借助另一个线程来操作(因为主线程仍然在阻塞等待中)。

在 event\_active 函数的解释里面就有一句话说的就是这个事情：

One common use **in** multithreaded programs **is** to wake the thread running event\_base\_loop() **from** another thread.

**事件是 Libevent 的最小单位**

这里的事件指的就是 struct event 数据结构。

事件可以注册的各种信号如下：

#define EV\_TIMEOUT 0x01

Indicates that a timeout has occurred.

#define EV\_READ 0x02

Wait **for** a **socket** **or** FD to become readable.

#define EV\_WRITE 0x04

Wait **for** a **socket** **or** FD to become writeable.

#define EV\_SIGNAL 0x08

Wait **for** a POSIX signal to be raised.

#define EV\_PERSIST 0x10

Persistent event: won't get removed automatically when

activated.

#define EV\_ET 0x20

Select edge-triggered behavior, if supported by the backend.

几乎所有其它更上层的数据结构都是基于 struct event 的包装来完成的。

**核心数据结构**

就拿 基于 [Libevent](https://github.com/nmathewson/Libevent) 写一个 HTTP 服务器举例来说说， 编程时需要理解的几个核心数据结构是：

* evhttp\_request
* evhttp\_connection
* bufferevent
* evbuffer

从上到下是从高层到底层的关系， 下文的顺序也是从高层往底层分析。

**evhttp\_request**

**struct** evhttp\_request {

// 每个 evhttp\_request 都内含一个 evhttp\_connection 来负责数据传输

**struct** evhttp\_connection \*evcon;

//输入输出的两个buffer（这两个buffer的数据是从 evhttp\_connection 里面的 input\_buffer 和 output\_buffer 拷贝过来的。）

**struct** evbuffer \*input\_buffer; /\* read data \*/

ev\_int64\_t ntoread;

**unsigned** chunked:1, /\* a chunked request \*/

userdone:1; /\* the user has sent all data \*/

**struct** evbuffer \*output\_buffer; /\* outgoing post or data \*/

//和HTTP协议有关的各种回调函数：

**void** (\*cb)(**struct** evhttp\_request \*, **void** \*);

**void** \*cb\_arg;

**void** (\*chunk\_cb)(**struct** evhttp\_request \*, **void** \*);

**int** (\*header\_cb)(**struct** evhttp\_request \*, **void** \*);

**void** (\*error\_cb)(**enum** evhttp\_request\_error, **void** \*);

**void** (\*on\_complete\_cb)(**struct** evhttp\_request \*, **void** \*);

**void** \*on\_complete\_cb\_arg;

// 其它

// ......

};

**evhttp\_connection**

evhttp\_request 和 evhttp\_connection 的关系很简单，拿协议栈来对比的话， 前者代表的是 HTTP 协议，即应用层协议， 后者代表的是 TCP 协议，即传输层协议。 前者需要管理所有和 HTTP 相关的数据内容，比如 HTTP header 数据和 body 数据。

**struct** evhttp\_connection {

// socket文件描述符

evutil\_socket\_t fd;

// evhttp\_connection 含有一个bufferevent，基于它进行数据传输。

**struct** bufferevent \*bufev;

// 和数据传输有关的状态

**enum** evhttp\_connection\_state state;

//和 HTTP 协议无关的数据传输回调函数

**void** (\*cb)(**struct** evhttp\_connection \*, **void** \*);

**void** \*cb\_arg;

**void** (\*closecb)(**struct** evhttp\_connection \*, **void** \*);

**void** \*closecb\_arg;

// 其它

// ......

};

evhttp\_connection 结构里面含有 enum evhttp\_connection\_state state 变量， 这个和 [Thrift异步IO服务器源码分析](http://yanyiwu.com/work/2014/12/06/thrift-tnonblockingserver-analysis.html) 里面的 保持每个连接的状态是一个道理。 维护状态变化是异步IO服务编程的必要条件。

**enum** evhttp\_connection\_state {

EVCON\_DISCONNECTED, /\*\*< not currently connected not trying either\*/

EVCON\_CONNECTING, /\*\*< tries to currently connect \*/

EVCON\_IDLE, /\*\*< connection is established \*/

EVCON\_READING\_FIRSTLINE,/\*\*< reading Request-Line (incoming conn) or

\*\*< Status-Line (outgoing conn) \*/

EVCON\_READING\_HEADERS, /\*\*< reading request/response headers \*/

EVCON\_READING\_BODY, /\*\*< reading request/response body \*/

EVCON\_READING\_TRAILER, /\*\*< reading request/response chunked trailer \*/

EVCON\_WRITING /\*\*< writing request/response headers/body \*/

};

**bufferevent**

bufferevent 就是包装了 EV\_READ event 和 EV\_WRITE event ， 并且带有读写缓冲区(evbuffer)的更高层的单位。

**struct** bufferevent {

// 读写事件

**struct** **event** ev\_read;

**struct** **event** ev\_write;

// 读写缓冲区

**struct** evbuffer \*input;

**struct** evbuffer \*output;

// 注意三个回调函数是核心

bufferevent\_data\_cb readcb;

bufferevent\_data\_cb writecb;

bufferevent\_event\_cb errorcb;

**void** \*cbarg;

// 其他

// ......

};

我觉得上面代码就非常简洁易懂了， 两个读写时间没什么好说的， 两个读写缓冲区也是必须的（evbuffer的实现在后面会谈到）， 三个回调函数就是核心。 读和写的回调函数没什么好说的， 唯一需要注意的是 bufferevent\_event\_cb errorcb 这个回调函数是必须注册的， 它关系到当该 bufferevent 对应的时间发生读写外的任何行为（比如socket关闭）时， 都会触发。

整理一遍 bufferevent 的事件处理过程就是：

* 当可读事件发生时，调用 readcb 将 socket 的数据 通过 recv 读出来存入 input 缓冲区；
* 当可写事件发生时，调用 writecb 将 output 缓冲区里面的数据通过 socket send 发送出去；
* 当其他事件发生时，比如 socket close 发生，进行相应的数据清理退出工作。

**evbuffer**

基本上的异步IO服务里的buffer都是一个德行（包括Nginx也是这样）， 都是即是数组又是链表（类似C++ STL里面的deque）。 对于 libevent 来说， evbuffer 是一个链表，管理整个缓冲区的头指针和尾指针，

**struct** evbuffer {

**struct** evbuffer\_chain \*first;

**struct** evbuffer\_chain \*last;

size\_t total\_len;

//......

};

对于 evbuffer 这个链表的每个单元，也就是 evbuffer\_chain 来说， 则是数组（连续内存空间），

**struct** evbuffer\_chain {

**struct** evbuffer\_chain \*next;

size\_t buffer\_len;

size\_t off;

**unsigned** **char** \*buffer;

//......

};

**总结**

对于我个人而言，读源码的时候主要是从核心数据结构入手， 如果理解了这几个核心数据结构， 一般就能猜到这些数据结构的相关函数都有哪些。 可以围绕着这些结构去找相关的函数为己所用。

转载请注明出处: [C1000K之Libevent源码分析](http://yanyiwu.com/work/2014/12/10/asyncronous-io-libevent.html)