**Thrift异步IO服务器源码分析**

06 Dec 2014

最近在使用 [libevent](http://libevent.org/) 开发项目，想起之前写 [Thrift源码剖析](http://yanyiwu.com/work/2014/10/17/thrift-source-code-illustration.html) 的时候说到关于 TNonblockingServer 以后会单独写一篇解析， 现在是时候了，就这篇了。

以下内容依然是基于 [thrift-0.9.0](http://archive.apache.org/dist/thrift/0.9.0/thrift-0.9.0.tar.gz) 。

**概述**

现在随着 Node.js 的兴起，很多人着迷 eventloop ， 经常是不明真相就会各种追捧，其实 eventloop 只是 一种高并发的解决方案。Thrift 的 TNonblockingServer 就是该解决方案的典型实现之一。

而且，Thrift 的 TNonblockingServer 实现代码干净，注释丰富， 并没有用到什么奇淫巧计，核心就是使用 [libevent](http://libevent.org/) 进行异步 驱动和状态的转换，只要有些 [libevent](http://libevent.org/) 经验的人就很容易 能看懂。

想进一步了解 [libevent](http://libevent.org/) 可以看看 [C1000K之Libevent源码分析](http://yanyiwu.com/work/2014/12/10/asyncronous-io-libevent.html) 。

**事件注册**

Thrift 使用 [libevent](http://libevent.org/) 作为服务的事件驱动器， libevent 其实就是 epoll 更高级的封装而已(在linux下是epoll)，而 struct event 事件是 [libevent](http://libevent.org/) 编程的最小单元，只要是使用 [libevent](http://libevent.org/) 就会使用到它，或者是包装它。

整个 TNonblockingServer 有三个关键的地方和 libevent 有关。

1. listener\_event

第一种是服务的监听事件也就是服务负责 listen 和 accept 的 主 socket ，如下。

// Register the server event

event\_set(&serverEvent\_,

listenSocket\_,

EV\_READ | EV\_PERSIST,

TNonblockingIOThread::listenHandler,

server\_);

当新的连接请求进来的时候，TNonblockingIOThread::listenHandler 函数被 触发，在 TNonblockingIOThread::listenHandler 里主要负责 accept 新连接。

2. pipe\_event

第二种比较有意思，这个事件对应的文件描述符是 socket pair ，使用 evutil\_socketpair 创建，其实也是调用linux接口 socketpair 搞出来的。这个东西不是之前理解的 网络通信套接字，在这里可以把它理解成一个管道来使用，如下:

// Create an event to be notified when a task finishes

event\_set(&notificationEvent\_,

getNotificationRecvFD(),

EV\_READ | EV\_PERSIST,

TNonblockingIOThread::notifyHandler,

**this**);

代码里面的 getNotificationRecvFD 就是拿这个 socket pair 管道的读文件描述符， 也就是当这个 socketpair 管道有数据可读时，该事件就会被触发，也就是回调函数 TNonblockingIOThread::notifyHandler 会被调用。

其实第二种事件非常好理解，可以类比多线程编程里面的任务队列， 不同线程之间共享着同一个任务队列来进行消息的传递。 而在 TNonblockingServer 里面，则通过该管道进行消息的传递。

3. connection\_event

第三种是每个连接的状态变化事件，每一个 TConnection 代表一个连接， 每一个 TConnection 含有一个 socket 文件描述符，并且当 TConnection 生成之后，会为它注册一个事件，负责对该 socket 的异步读写。 如下：

event\_set(&event\_, tSocket\_->getSocketFD(), eventFlags\_,

TConnection::eventHandler, this);

注意到，每个连接都会注册一个 第三种事件， 也就是说，程序的整个运行过程中，假设并发连接数为 n ， 则第三种事件的数量也为 n，而第一种和第二种始终 只有一个事件。 所以真个程序运行过程中事件的数量是【2 + n】。

**socket状态转移**

因为是异步编程，每个socket都必须设置为非阻塞。 当可读的事件发生时，则读，可写的事件发生时，则写。 读和写两种操作会互相交替进行，所以我们需要用 状态值来进行不同的逻辑处理。

TNonblockingServer 里的状态值有以下三种：

/// Three states **for** sockets: **recv** frame size, **recv** data, **and** **send** mode

enum TSocketState {

SOCKET\_RECV\_FRAMING,

SOCKET\_RECV,

SOCKET\_SEND

};

需要补充说明的是，要和 Thrift 的 TNonblockingServer 通信，则客户端 需要使用

shared\_ptr<**TTransport**> transport(new TFramedTransport(socket));

来作为传输工具，就是因为 TNonblockingServer 的 socket recv 数据是 按 frame 来一帧帧的接受。所以第一个状态值 SOCKET\_RECV\_FRAMING 代表进入该状态就是有帧头（数据包的大小）可以读取， 而第二个状态值 SOCKET\_RECV 代表有数据可以读取，先读完帧头才读该数据。 第三个状态 SOCKET\_SEND 代表有数据可以发送。

每次 rpc 调用的过程的状态转移先后过程如下：

SOCKET\_RECV\_FRAMING -> SOCKET\_RECV -> SOCKET\_SEND

这三个状态都有可能被重复调用，取决于数据包的大小。

每次 socket 状态转移靠 workSocket 函数完成：

/\*\*

\* Libevent handler called (via our static wrapper) when the connection

\* socket had something happen. Rather than use the flags [libevent] passed,

\* we use the connection state to determine whether we need to read or

\* write the socket.

\*/

**void** workSocket();

**app状态转移**

上面的 socket 状态转移，是针对每个连接的数据收发状态转移， 和 socket 紧密相关，而这里的 app状态转移则是针对整个 rpc 远程函数调用(不过每次rpc调用其实也是建立在某个连接的基础之上)。

app状态的代码如下：

**enum** TAppState {

APP\_INIT,

APP\_READ\_FRAME\_SIZE,

APP\_READ\_REQUEST,

APP\_WAIT\_TASK,

APP\_SEND\_RESULT,

APP\_CLOSE\_CONNECTION

};

状态的转移顺序如下：

1. APP\_INIT 初始状态。
2. APP\_READ\_FRAME\_SIZE 读取帧数据。
3. APP\_READ\_REQUEST 读取请求的数据，并根据请求的数据 进行数据的解析和任务的生成，并且将任务扔进线程池。
4. APP\_WAIT\_TASK 等待任务的完成
5. APP\_SEND\_RESULT 任务已经完成，将任务结果发送。
6. APP\_CLOSE\_CONNECTION 关闭连接。

每次app状态转移由 TConnetion::transition 函数完成：

/\*\*

\* This is called when the application transitions from one state into

\* another. This means that it has finished writing the data that it needed

\* to, or finished receiving the data that it needed to.

\*/

**void** transition();

状态3 -> 状态4 -> 状态5 转移很关键，涉及到线程池和主线程的交互。 请看下文。

**任务的线程池**

总所周知的是，异步服务器最适合的场景是高并发，IO 密集型程序。 对于 CPU 密集型的应用场景一般使用多线程服务来解决。 而对于 RPC 服务，TNonblockingServer 想使用异步 IO 来应对高并发。 但是对于 rpc 远程函数调用，如果被方法的函数是 CPU 密集型的函数， 则运行该函数的过程整个主线程就会被阻塞，也就是传说中的 【block the whole world】， 对于此，TNonblockingServer 的解决方法是将该函数包装成一个任务， 然后扔进线程池，以此来避免主线程的阻塞。

线程池本身没什么好说的，但是在 TNonblockingServer 里 面需要了解的就是 线程池和主线程的交互：

当 TConnetion 的 app状态 进入 APP\_READ\_REQUEST 之后 读取完请求数据之后，则将任务包装好扔进线程池。 并且将状态改变(APP\_READ\_REQUEST -> APP\_WAIT\_TASK)：

// The application **is** now waiting on the task to finish

appState\_ = APP\_WAIT\_TASK;

并且将该连接标识为 Idle ，如下函数：

// Set this connection idle so that [libevent] doesn't process more

// data on it while we're still waiting for the threadmanager to

// finish this task

setIdle();

setIdle 的目的在于将该连接对应的 socket事件标志位清空， 也就是在 Idle阶段不再关心该 socket是否有数据可读或者可写。

而当线程池里的某个 Task 运行完毕后，则会触发主线程的 pipe\_event (上文中的已注册事件种的第二种事件)，告知主线程任务已完成。 如下：

// Signal completion back to the libevent thread via a pipe

**if** (!connection\_->notifyIOThread()) {

**throw** TException("TNonblockingServer::Task::run: failed write on notify pipe");

}

主线程收到通知之后，则会从 状态4(APP\_WAIT\_TASK) 转 移向 状态5(APP\_SEND\_RESULT) ，进入向 客户端发送函数调用结果的过程。

**总结**

Thrift 的 TNonblockingServer 注释很丰富，原理清晰。 个人认为基本上是事件驱动服务器的入门教科书级代码了， 事件驱动服务器核心在于状态转移， 因为事件驱动的原因，每次转换 事件我们都需要保存当前的状态。 没啥，都是状态而已。

哦对了，在下读源码的时候习惯加 cout ，然后跑起来看结果， 文末有一份运行示例可以帮助理解，有兴趣的可以看看， 修改后的源码在 [MyTNonblockingServer](http://7viirv.com1.z0.glb.clouddn.com/4f6bc786c4_TNonblockingServer.cpp) 。

**运行示例**

TNonblockingServer.cpp945TNonblockingServer::handleEvent

TNonblockingServer.cpp990Create a new TConnection **for** this client **socket**.

TNonblockingServer.cpp1015iothreadnumber = 0

TNonblockingServer.cpp714APP\_INIT

TNonblockingServer.cpp442TNonblockingServer::TConnection::workSocket()

TNonblockingServer.cpp448SOCKET\_RECV\_FRAMING

TNonblockingServer.cpp493size known; now get the rest of the frame

TNonblockingServer.cpp736APP\_READ\_FRAME\_SIZE

TNonblockingServer.cpp442TNonblockingServer::TConnection::workSocket()

TNonblockingServer.cpp498SOCKET\_RECV

TNonblockingServer.cpp523 We are done reading, move onto the **next** state

TNonblockingServer.cpp590APP\_READ\_REQUEST

TNonblockingServer.cpp625setIdle

TNonblockingServer.cpp374connection\_->notifyIOThread()

TNonblockingServer.cpp1401notifyHandler

TNonblockingServer.cpp1415TNonblockingIOThread::notifyHandler

TNonblockingServer.cpp661APP\_WAIT\_TASK

TNonblockingServer.cpp442TNonblockingServer::TConnection::workSocket()

TNonblockingServer.cpp535SOCKET\_SEND

TNonblockingServer.cpp564writeBufferPos\_ done

TNonblockingServer.cpp698APP\_SEND\_RESULT

TNonblockingServer.cpp714APP\_INIT

TNonblockingServer.cpp442TNonblockingServer::TConnection::workSocket()

TNonblockingServer.cpp448SOCKET\_RECV\_FRAMING

转载请注明出处: [Thrift异步IO服务器源码分析](http://yanyiwu.com/work/2014/12/06/thrift-tnonblockingserver-analysis.html)