miniob启动和建立连接过程

1 server启动

点击运行，根据启动参数，会执行main函数。main函数主要做了几件事，解析命令行参数，配置全局参数，根据参数启动server，开启监听。

1.1 g_server = init_server();

根据参数配置选项，并在最后根据是否使用标准输入输出，建立server。

```
  Server *server = nullptr;
  if (server_param.use_std_io) {
    server = new CliServer(server_param);   命令行有CLI
  } else {
    server = new NetServer(server_param);
  }

  return server;
}
```

1.2 g_server->serve();

根据创建的server（CLi或者是Net），开启监听。

2 CLI

如果启动方式用`./bin/observer -f ../etc/observer.ini -P cli`这行参数，那么会使用CLI启动，也就是命令行交互模式。

如果是CLI启动方式，会直接跳过多线程，直接使用SqlTaskHandler处理请求。在这个过程中没有设计建立线程，这是我们主要学习的流程。

```
int CliServer::serve()
{
  CliCommunicator communicator;

  RC rc = communicator.init( fd: STDIN_FILENO,  session: make_unique<Session>( other: Session::default_session()),  addr: "stdin");
  if (OB_FAIL(rc)) {                                                                  构建stdio的输入输出的管理者
    LOG_WARN("failed to init cli communicator. rc=%s", strrc(rc));
    return -1;
  }

  started_ = true;

  SqlTaskHandler task_handler;   直接使用SqlTaskHandler管理sql，session
  while (started_ && !communicator.exit()) {
    rc = task_handler.handle_event(&communicator);
    if (OB_FAIL(rc)) {
      started_ = false;
    }
  }

  started_ = false;
  return 0;
}
```

在while循环中，调用handle_event方法，处理sql。

```cpp
RC SqlTaskHandler::handle_event(Communicator *communicator)
{
  SessionEvent *event = nullptr;
  RC rc = communicator->read_event([&]event);
  if (OB_FAIL(rc)) {...}

  if (nullptr == event) {...}

  session_stage_.handle_request2(event);

  SQLStageEvent sql_event(event, sql:event->query());

  rc = handle_sql(&sql_event);
  if (OB_FAIL(rc)) {...}

  bool need_disconnect = false;

  rc = communicator->write_result(event, [&]need_disconnect);
  LOG_INFO("write result return %s", strrc(rc));
  event->session()->set_current_request(nullptr);
  Session::set_current_session(nullptr);

  delete event;

  if (need_disconnect) {...}
  return RC::SUCCESS;
}
```

```cpp
RC SqlTaskHandler::handle_sql(SQLStageEvent *sql_event)
{
  RC rc = query_cache_stage_.handle_request(sql_event);
  if (OB_FAIL(rc)) {...}

  rc = parse_stage_.handle_request(sql_event);
  if (OB_FAIL(rc)) {...}

  rc = resolve_stage_.handle_request(sql_event);
  if (OB_FAIL(rc)) {...}

  rc = optimize_stage_.handle_request(sql_event);
  if (rc != RC::UNIMPLENMENT && rc != RC::SUCCESS) {...}

  rc = execute_stage_.handle_request(sql_event);
  if (OB_FAIL(rc)) {...}

  return rc;
}
```

缓存处理

解析sql字符串

解析内部数据结构为进一步使用的数据结构

优化

执行

结束

上面这个流程也是我们主要学习的流程，重点关注一个字符串是如何通过parser变成各种node，各种node怎么变成stmt，stmt怎么递归的生成物理计划和逻辑计划，最后再执行。当然这不是本文的主要讲解过程，本文主要简单了解服务端怎么接受命令。

3 Net

```
int NetServer::serve()
{
  thread_handler_ = ThreadHandler::create( name: server_param_.thread_handling.c_str());
  if (thread_handler_ == nullptr) {...}

  RC rc = thread_handler_->start();
  if (OB_FAIL(rc)) {...}

  int retval = start();
  if (retval == -1) {...}

  if (!server_param_.use_std_io) {
    struct pollfd poll_fd;
    poll_fd.fd      = server_socket_;
    poll_fd.events  = POLLIN;
    poll_fd.revents = 0;

    while (started_) {
      int ret = poll(&poll_fd,  nfds: 1,  timeout: 500);
      if (ret < 0) {...} else if (0 == ret) {...}

      if (poll_fd.revents & (POLLERR | POLLHUP | POLLNVAL)) {...}

      this->accept(server_socket_);
    }
  }

  thread_handler_->stop();
  thread_handler_->await_stop();
  delete thread_handler_;
  thread_handler_ = nullptr;

  started_ = false;
  LOG_INFO( fmt: "NetServer quit");
  return 0;
}
```

创建线程池，有one_per和java_thread两种，默认是前者

使用poll监听事件，500ms内有事件则返回事件的fd，后续调用accept

ver::serve

```
void NetServer::accept(int fd)
{
  struct sockaddr_in addr;
  socklen_t          addrlen = sizeof(addr);

  int ret = 0;

  int client_fd = ::accept(fd, (struct sockaddr *)&addr, &addrlen);
  if (client_fd < 0) {...}

  char ip_addr[24];
  if (inet_ntop( af: AF_INET, &addr.sin_addr, ip_addr,  len: sizeof(ip_addr)) == nullptr) {...}
  stringstream address;
  address << ip_addr << ":" << addr.sin_port;
  string addr_str = address.str();

  ret = set_non_block(client_fd);
  if (ret < 0) {...}

  if (!server_param_.use_unix_socket) {...}

  Communicator *communicator = communicator_factory_.create(server_param_.protocol);

  RC rc = communicator->init(client_fd,  session: make_unique<Session>( other: Session::default_session()), addr_str);
  if (rc != RC::SUCCESS) {...}

  LOG_INFO("Accepted connection from %s\n", communicator->addr());

  rc = thread_handler_->new_connection(communicator);
  if (OB_FAIL(rc)) {...}
}
```

使用线程池创建新的connection

3.1 线程池的start

one_thread_per_connection的比较简单，直接返回成功即可，因为每个连接都创建新的线程。

java的稍微复杂一些。创建线程池之后，会启动一个工作线程监听事件，有事件产生，会将其放入线程池执行。

```
RC JavaThreadPoolThreadHandler::start()
{
  if (nullptr != event_base_) {...}

  // 在多线程场景下使用libevent，先执行这个函数
  evthread_use_pthreads();
  // 创建一个event_base对象，这个对象是libevent的主要对象，所有的事件都会注册到这个对象中
  event_base_ = event_base_new();
  if (nullptr == event_base_) {...}

  // 创建线程池
  // 这里写死了线程池的大小，实际上可以从配置文件中读取
  int ret = executor_.init("SQL",  // name
                           2,      // core size
                           8,      // max size
                           60*1000 // keep alive time
                           );
  if (0 != ret) {...}

  // libevent 的监测消息循环主体，要放在一个线程中执行
  // event_loop_thread 是运行libevent 消息监测循环的函数，会长期运行，并且会放到线程池中占据一个线程
  auto event_worker = std::bind(&JavaThreadPoolThreadHandler::event_loop_thread, this);
  ret = executor_.execute(event_worker);
  if (0 != ret) {...}

  return RC::SUCCESS;
}
```

libevent回调线程

```
  // event_loop_thread 是运行libevent 消息监测循环的函数，会长期运行，并且会放到线程池中占据一个线程
  auto event_worker = std::bind(&JavaThreadPoolThreadHandler::event_loop_thread, this);
  ret = executor_.execute(event_worker);
  if (0 != ret) {...}

  return RC::SUCCESS;
}

/** @brief libevent 的消息事件回调函数 ... */
static void event_callback(evutil_socket_t fd, short event, void *arg){...}

void JavaThreadPoolThreadHandler::handle_event(EventCallbackAg *ag){...}

void JavaThreadPoolThreadHandler::event_loop_thread()
{
  LOG_INFO("event base dispatch begin");
  // event_base_dispatch 也可以完成这个事情。
  // event_base_loop 会等待所有事件都结束
  // 如果不增加 EVLOOP_NO_EXIT_ON_EMPTY 标识，当前事件都处理完成后，就会退出循环
  // 加上这个标识就是即使没有事件，也等在这里
  event_base_loop(event_base_, EVLOOP_NO_EXIT_ON_EMPTY);
  LOG_INFO("event base dispatch end");
}
```

start()函数，绑定了一个事件和执行的函数

这个线程实际上执行了非常重要的任务：它负责处理所有注册到 libevent 事件循环中的事件。这包括但不限于：

- 监听网络连接请求。
- 接收和发送数据。
- 处理定时任务。

- 监听文件描述符的状态变化。

这个线程通常会一直运行，因为它需要持续监测和响应事件，这是事件驱动的网络服务的基础。如果线程停止了，那么这些事件就无法得到处理，服务也就无法正常工作。

对该函数的解释：

```
/**
  Wait for events to become active, and run their callbacks.

  This is a more flexible version of event_base_dispatch().

  By default, this loop will run the event base until either there are no more
  pending or active events, or until something calls event_base_loopbreak() or
  event_base_loopexit().  You can override this behavior with the 'flags'
  argument.

  @param eb the event_base structure returned by event_base_new() or
     event_base_new_with_config()
  @param flags any combination of EVLOOP_ONCE | EVLOOP_NONBLOCK
  @return 0 if successful, -1 if an error occurred, or 1 if we exited because
     no events were pending or active.
  @see event_base_loopexit(), event_base_dispatch(), EVLOOP_ONCE,
     EVLOOP_NONBLOCK
  */
EVENT2_EXPORT_SYMBOL
int event_base_loop(struct event_base *, int);
```

等待事件到来并执行他们的回调

3.2 new_connection

3.2.1 one_per

每次创建一个新线程，并且立刻投入使用。

```
RC OneThreadPerConnectionThreadHandler::new_connection(Communicator *communicator)
{
  lock_guard guard([&]lock_);

  auto iter = thread_map_.find(communicator);
  if (iter != thread_map_.end()) {...}

  Worker *worker = new Worker([&]*this, communicator);
  thread_map_[communicator] = worker;
  return worker->start();
}
```

```
RC start()
{
  thread_ = new thread(f:std::ref([&]*this));
  return RC::SUCCESS;
}

RC stop()
{
  running_ = false;
  return RC::SUCCESS;
}

RC join(){...}

void operator()()
{
  LOG_INFO("worker thread start. communicator = %p", communicator_);
  int ret = thread_set_name("SQLWorker");
  if (ret != 0) {...}

  struct pollfd poll_fd;
  poll_fd.fd = communicator_->fd();
  poll_fd.events = POLLIN;
  poll_fd.revents = 0;

  while (running_) {
    int ret = poll(&poll_fd, nfds:1, timeout:500);
    if (ret < 0) {...} else if (0 == ret) {...}

    if (poll_fd.revents & (POLLERR | POLLHUP | POLLNVAL)) {...}

    RC rc = task_handler_.handle_event(communicator_);
    if (OB_FAIL(rc)) {...}
  }

  LOG_INFO("worker thread stop. communicator = %p", communicator_);
  host_.close_connection(communicator_); /// 连接关闭后，当前对象会被删除
```

创建一个新线程，并且会调用线程()重载方法

和之前处理sql的方式一

但是这样这个线程处理完这个工作不是会空转了吗？我猜测会在某个地方更改running_状态使其停止。

更正，可能会在处理完请求后break。

```
while (running_) {
  int ret = poll(&poll_fd, nfds:1, timeout:500);
  if (ret < 0) {
    LOG_WARN("poll error. fd = %d, ret = %d, error=%s", poll_fd.fd, ret, strerror(errno));
    break;
  } else if (0 == ret) {...}

  if (poll_fd.revents & (POLLERR | POLLHUP | POLLNVAL)) {
    LOG_WARN("poll error. fd = %d, revents = %d", poll_fd.fd, poll_fd.revents);
    break;
  }

  RC rc = task_handler_.handle_event(communicator_);
  if (OB_FAIL(rc)) {...}
}

LOG_INFO("worker thread stop. communicator = %p", communicator_);
host_.close_connection(communicator_); /// 连接关闭后，当前对象会被删除
}
```

处理完之后会更改fd状态，使其break掉，关闭线程。

3.2.2 java_thread

java_thread也会每次在事件到达时调用new_connection，但是和每次都创建线程不同，当有事件到达时，会创建一个可读事件，通知event_base，之后会有线程来取任务。

```
RC JavaThreadPoolThreadHandler::new_connection(Communicator *communicator)
{
  int fd = communicator->fd();
  LOG_INFO("new connection. fd=%d", fd);
  EventCallbackAg *ag = new EventCallbackAg;
  ag->host = this;
  ag->communicator = communicator;
  ag->ev = nullptr;
  /// 创建一个libevent事件对象。其中EV_READ表示可读事件，就是客户端发消息时会触发事件。
  /// EV_ET 表示边缘触发，有消息时只会触发一次，不会重复触发。这个标识在Linux平台上是支持的，但是有些平台不支持。
  /// 使用EV_ET边缘触发时需要注意一个问题，就是每次一定要把客户端发来的消息都读取完，直到read返回EAGAIN为止。
  /// 我们这里不使用边缘触发。
  /// 注意这里没有加 EV_PERSIST，表示事件触发后会自动从event_base中删除，需要自己再手动加上这个标识。这是有必
  /// 要的，因为客户端发出一个请求后，我们再返回客户端消息之前，不再希望接收新的消息。
  struct event *ev = event_new(event_base_, fd, EV_READ, event_callback, ag);
  if (nullptr == ev) {...}
  ag->ev = ev;

  lock_.lock();
  event_map_[communicator] = ag;
  lock_.unlock();

  int ret = event_add(ev, timeout: nullptr);
  if (0 != ret) {...}
  LOG_TRACE("add event success. fd=%d, communicator=%p", fd, communicator);

  return RC::SUCCESS;
}
```

所以new_connction的操作是添加事件。

事件的消费端也在同一个类下。static方法，也就是说当注册事件，发送事件的同时，该函数会被event_base调用。

```
/**
 * @brief libevent 的消息事件回调函数
 * @details 当libevent检测到某个连接上有消息到达时，比如客户端发消息过来、客户端断开连接等，就会
 * 调用这个回调函数。
 * 按照libevent的描述，我们不应该在回调函数中执行比较耗时的操作，因为回调函数是运行在libevent的
 * 事件检测主循环中。
 * @param fd 有消息的连接的文件描述符
 * @param event 事件类型，比如EV_READ表示有消息到达，EV_CLOSED表示连接断开
 * @param arg 我们注册给libevent的回调函数的参数
 */
static void event_callback(evutil_socket_t fd, short event, void *arg)
{
  if (event & (EV_READ | EV_CLOSED)) {
    LOG_TRACE("got event. fd=%d, event=%d", fd, event);
    EventCallbackAg *ag = (EventCallbackAg *)arg;
    JavaThreadPoolThreadHandler *handler = ag->host;
    handler->handle_event(ag);              线程池处理任务
  } else {
    LOG_ERROR("unexpected event. fd=%d, event=%d", fd, event);
  }
}
```

```cpp
void JavaThreadPoolThreadHandler::handle_event(EventCallbackAg *ag)
{
  /*
  当前函数是一个libevent的回调函数。按照libevent的要求，我们不能在这个函数中执行比较耗时的操作，
  因为它是运行在libevent主消息循环处理函数中的。

  我们这里在收到消息时就把它放到线程池中处理。
  */

  // sql_handler 是一个回调函数
  auto sql_handler = [this, ag]()->void {
    RC rc = sql_task_handler_.handle_event(ag->communicator); // 这里会有接收消息、处理请求然后返回结果一条龙服务
    if (RC::SUCCESS != rc) {...} else if (0 != event_add(ag->ev, timeout: nullptr)) {...} else {
      // 添加event后就不应该再访问communicator了，因为可能会有另一个线程处理当前communicator
      // 或者就需要加锁处理并发问题
      // LOG_TRACE("add event. fd=%d, communicator=%p", event_get_fd(ag->ev), this);
    }
  };

  executor_.execute(sql_handler);
}
```

之前执行event_loop的工作线程。他主要负责消息处理，并将其提交到线程池中。

和之前的处理方式没有什么不同。

至此，不管是CLI或者是Net的接受方式都已经知道了，CLI采用同server单线程的处理方式，Net的one_per采用每次accept都创建新线程的方式，java_thread采用libevent监听消息然后交由线程池的方式。

当然，这不是我们主要需要关注的部分，简单了解即可。