

# 第六讲：服务层组件连接器的设计与实现

知春路遇上八里桥

<2024-05-23 Thu>



① 前情提要

② 数据结构

③ 代码分析



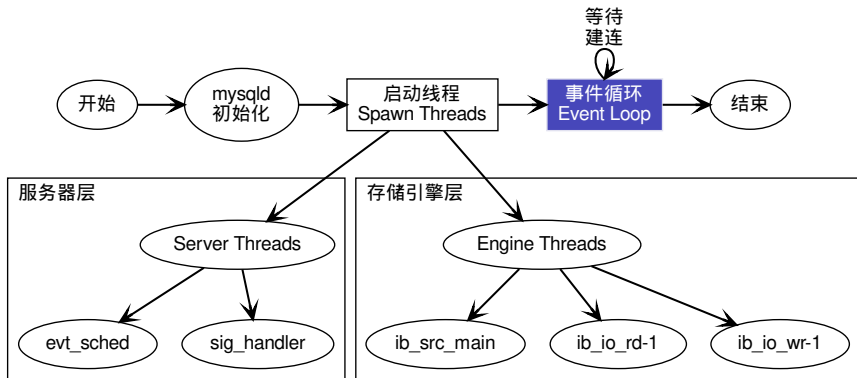
1

## 前情提要



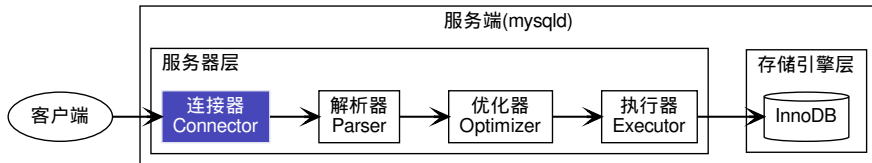
# 启动流程

MySQL 初始化完成之后，便进入一个死循环中，接受客户端请求，并完成客户端的命令



# 执行流程

- 客户端发送 QUERY 字符串后，首先就和连接器进行交互，所以我们先介绍连接器的实现<sup>1</sup>



<sup>1</sup>查询缓存在 8.0.x 已经移除，这里简历了流程图

2

## 数据结构



# 再看事件循环

- 调用连接器中实现的事件循环函数

- ▶ ★ sql/conn\_handler/connection\_acceptor.h

```
61 void connection_event_loop() {
62     Connection_handler_manager *mgr =
63         Connection_handler_manager::get_instance();
64     while (!connection_events_loop_aborted()) {
65         Channel_info *channel_info = m_listener->listen_for_connection_event();
66         if (channel_info != nullptr) mgr->process_new_connection(channel_info);
67     }
68 }
```

- ▶ 需要关注以下两点 ↓

- ① m\_listener->listen\_for\_connection\_event() 监听并接受连接，这里返回 Channel\_info 对象
- ② process\_new\_connection() 处理一个新来的连接

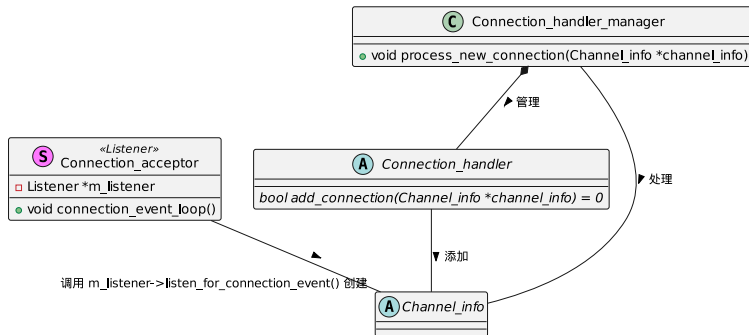
- 连接器的实现基本在目录 sql/conn\_handler 中

- ▶ channel\_info.h
- ▶ connection\_acceptor.h
- ▶ socket\_connection.h
- ▶ ...



# 基本处理步骤及核心类

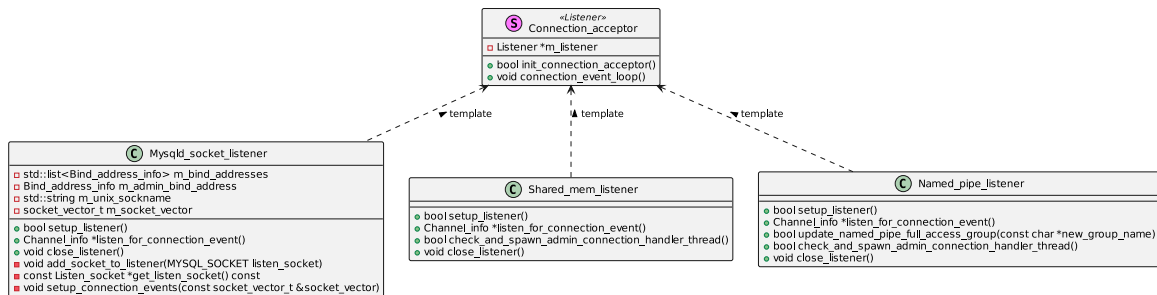
- ① Connection\_acceptor 用于监听客户端的连接请求
- ② Connection\_acceptor 创建 Channel\_Info 作为连接信道
- ③ Connection\_handler\_manager 指派给 Connection\_Handler 进行调度
- ④ 这吐槽一下 MySQL 的代码风格<sup>2</sup>



<sup>2</sup>[https://dev.mysql.com/doc/dev/mysql-server/8.0.37/PAGE\\_CODING\\_GUIDELINES.html](https://dev.mysql.com/doc/dev/mysql-server/8.0.37/PAGE_CODING_GUIDELINES.html)



# Listener 监听器



- **Connection\_acceptor** 以模版方式实现了 Listener 监听器,


▶ ★ `sql/conn_handler/connection_acceptor.h`

```
42  template <typename Listener>
43  class Connection_acceptor {
44      Listener *m_listener;
```

- **Mysql\_socket\_listener** 实现 Socket 的方式监听客户端的连接事件
- **Shared\_mem\_listener** 通过共享内存的监听方式
- **Named\_pipe\_listener** 通过命名管道来监听和接收客户请求



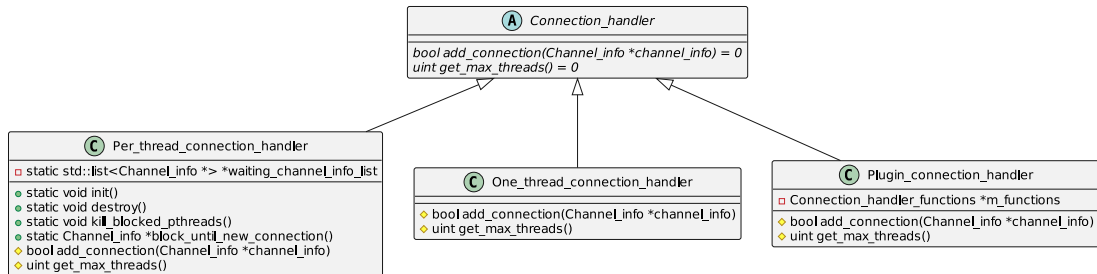
# Connection\_handler\_manager 连接处理器的管理器

 Connection_handler_manager
<ul style="list-style-type: none"><li>static Connection_handler_manager* m_instance</li><li>Connection_handler *m_connection_handler</li><li>Connection_handler *m_saved_connection_handler</li></ul>
<ul style="list-style-type: none"><li>static Connection_handler_manager* get_instance()</li><li>static bool init()</li><li>void process_new_connection(Channel_info *channel_info)</li><li>void load_connection_handler(Connection_handler *conn_handler)</li><li>bool unload_connection_handler()</li></ul>

- Connection\_handler\_manager 是单例类<sup>3</sup>，用来管理连接处理器
  - ▶ 静态变量 m\_instance 保存该单例的实例
  - ▶ 通过静态方法 get\_instance() 可以获取唯一实例
- init() 初始化管理器
- process\_new\_connection() 处理新的请求
- load\_connection\_handler(Connection\_handler \*conn\_handler) 装载处理器
- unload\_connection\_handler() 卸载处理器

<sup>3</sup>[https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classConnection\\_\\_handler\\_\\_manager.html](https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classConnection__handler__manager.html)

# Connection\_handler 连接处理器

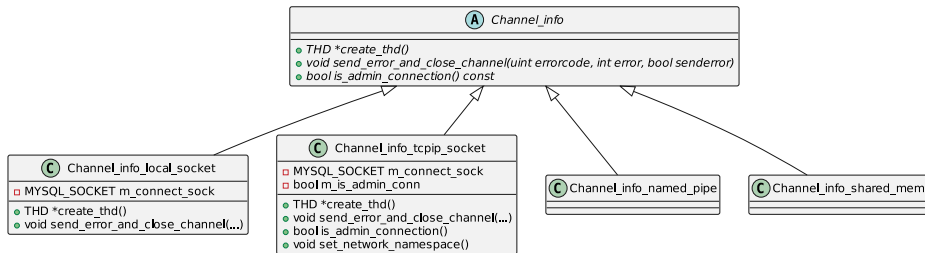


- Channel\_handler 连接处理器的抽象类<sup>4</sup>
- 通过 thread\_handling 参数可以设置线程池的类型
  - ▶ one-thread-per-connection 不启用线程池【默认】
  - ▶ pool-of-threads 启用线程池
- Per\_thread\_connection\_handler 不启用线程池，每一个连接用单独的线程处理
- One\_thread\_connection\_handler 启用线程池，所有连接用同一个线程处理
- Plugin\_connection\_handler 支持由 Plugin 具体实现的 handler，例如线程池

<sup>4</sup>[https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classConnection\\_\\_handler.html](https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classConnection__handler.html)



# Channel\_info 连接信道



- Channel\_info 连接信道的抽象类<sup>5</sup>，包含以下四个具体实现类
- Channel\_info\_local\_socket 与本地方式与服务器进行交互
- Channel\_info\_tcpip\_socket 以 TCP/IP 方式与服务器进行交互
  - ▶ TCP\_socket 描述 TCP socket 信息
  - ▶ Unix\_socket 描述 Unix socket 信息
- Channel\_info\_named\_pipe 命名管道的信道
- Channel\_info\_shared\_mem 共享内存模式的信道

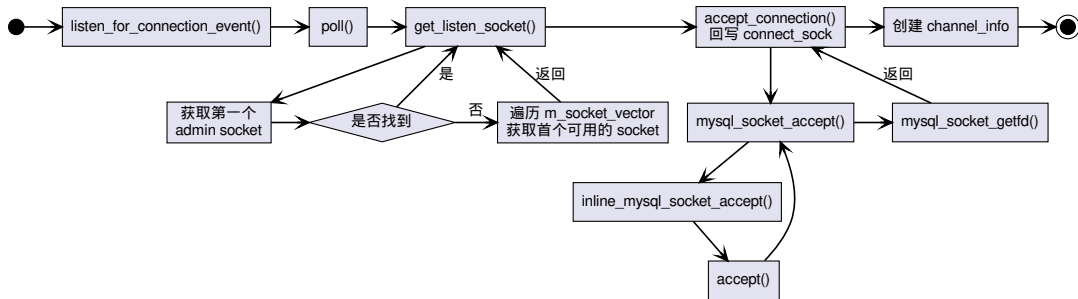
<sup>5</sup>[https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classChannel\\_\\_\\_info.html](https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classChannel___info.html)

3

## 代码分析



# 监听器处理流程梳理



# 监听器实现 Mysqlsocket\_listener

- 获取请求，通过 poll 方式来实现多路复用获取 socket

- ▶ ★ sql/conn\_handler/socket\_connection.cc

```
1348 Channel_info *Mysqlsocket_listener::listen_for_connection_event() {
1349     #ifdef HAVE_POLL
1350         int retval = poll(&m_poll_info.m_fds[0], m_socket_vector.size(), -1);
1351     #else
1352         m_select_info.m_read_fds = m_select_info.m_client_fds;
1353         int retval = select((int)m_select_info.m_max_used_connection,
1354                             &m_select_info.m_read_fds, 0, 0, 0);
1355     #endif
```

- 接收请求，获取有效的 socket 后进入 accept\_connection()

- ▶ ★ sql/conn\_handler/socket\_connection.cc

```
1398     if (accept_connection(listen_socket->m_socket, &connect_sock)) {
1399     #ifdef HAVE_SETNS
1400         if (!network_namespace_for_listening_socket.empty())
1401             (void)restore_original_network_namespace();
1402     #endif
1403         return nullptr;
1404     }
```

- 在 connection\_event\_loop() 死循环提供监听逻辑



# 管理器初始化

- `init()` 初始化管理器，在服务器启动时调用

- ▶ ★ `sql/mysql.cc`

```
11202  if (Connection_handler_manager::init()) {  
11203      LogErr(ERROR_LEVEL, ER_CONNECTION_HANDLING_OOM);  
11204      return 1;  
11205  }
```

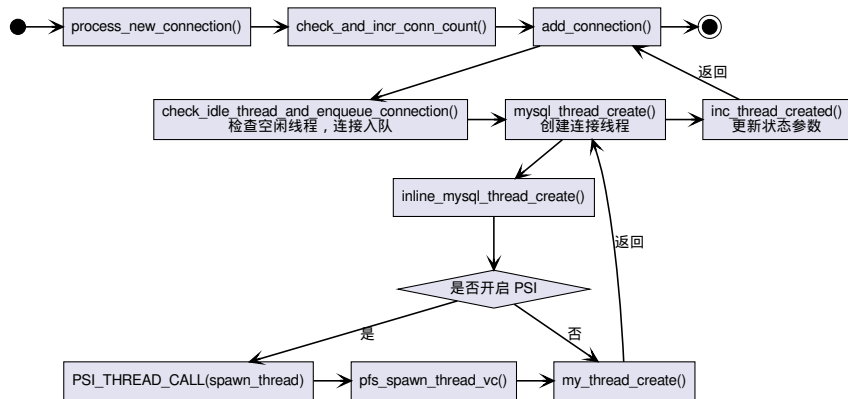
- 具体调用栈见下图

```
(gdb) i b  
Num      Type      Disp Enb Address      What  
1         breakpoint keep y 0x00005555913804b in Connection_handler_manager::init() at /opt/src/mysql-server/sql/con  
breakpoint already hit 1 time  
(gdb) bt  
#0 Connection_handler_manager::init () at /opt/src/mysql-server/sql/conn_handler/connection_handler_manager.cc:150  
#1 0x0000555558ba0eac in get_options (argc_ptr=0x55555e630470 <remaining_argc>, argv_ptr=0x55555e630478 <remaining_argv>  
#2 0x0000555558b8e39f in init_common_variables () at /opt/src/mysql-server/sql/mysql.cc:4930  
#3 0x0000555558b989de in mysqld_main (argc=9, argv=0x55555eca0350) at /opt/src/mysql-server/sql/mysql.cc:7682  
#4 0x0000555558b8590d in main (argc=2, argv=0x7fffffff088) at /opt/src/mysql-server/sql/main.cc:26  
(gdb) █
```





# 处理新连接流程梳理



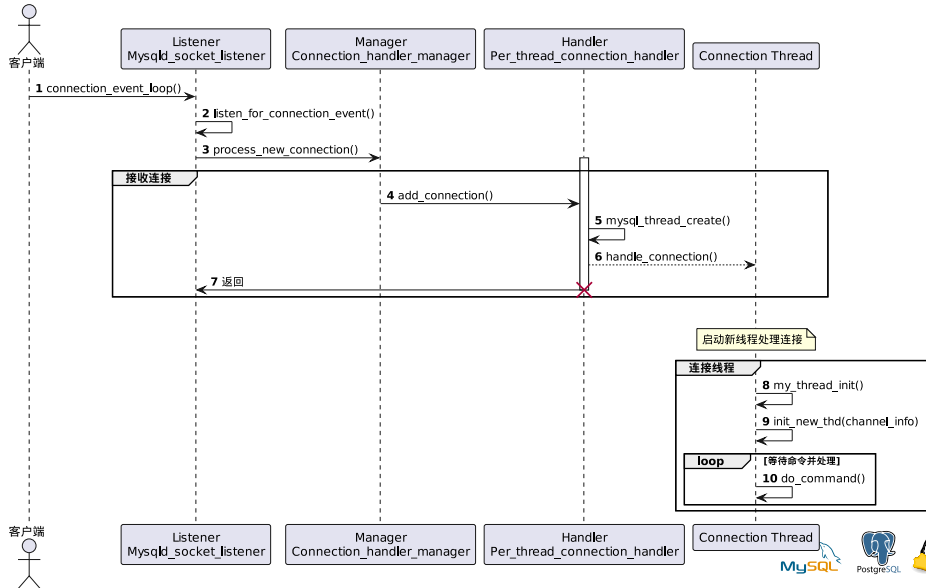
# 处理新连接的实现

- `check_and_incr_conn_count()` 增加连接计数，如果超过连接上限则拒绝
- `Per_thread_connection_handler::add_connection()` 添加当前信道
  - ▶ 如果有空闲的线程，那么将直接利用空闲的线程
  - ▶ 否则将创建一个新的线程来处理新连接
- `mysql_thread_create()` 创建并启动线程
  - ▶ 通过宏定义调用 `inline_mysql_thread_create()`
  - ▶ `PSI_THREAD_CALL(spawn_thread)` 宏调用 `pfs_spawn_thread_vc()`
    - ① 添加一些仪表盘的设置
  - ▶ 调用 `my_thread_create()` 后通过 `pthread_create()` 创建新线程
- `handle_connection()` 作为线程的入口函数
  - ▶ 首先是初始化线程需要的内存
  - ▶ 然后创建一个 THD 对象 `init_new_thd()`
  - ▶ 创建/或重用 `psi` 对象，并加到 `thd` 对象
  - ▶ 将 `thd` 对象加到 `thd manager` 中
  - ▶ 调用 `thd_prepare_connection()` 做验证
- 最后调用 `do_command()` 处理命令

```
302  while (thd_connection_alive(thd)) {  
303      if (do_command(thd)) break;  
304  }
```



# 连接建立极简流程



# 结束

