- 线程同步机制
- 半同步半反应堆线程池
- http连接处理
- 定时器处理非活动连接
- 日志系统
- 数据库模块

线程同步机制

// 使用RAII机制实现信号量的创建和销毁,并实现PV操作 // 使用RAII机制实现互斥锁机制,以实现多线程同步

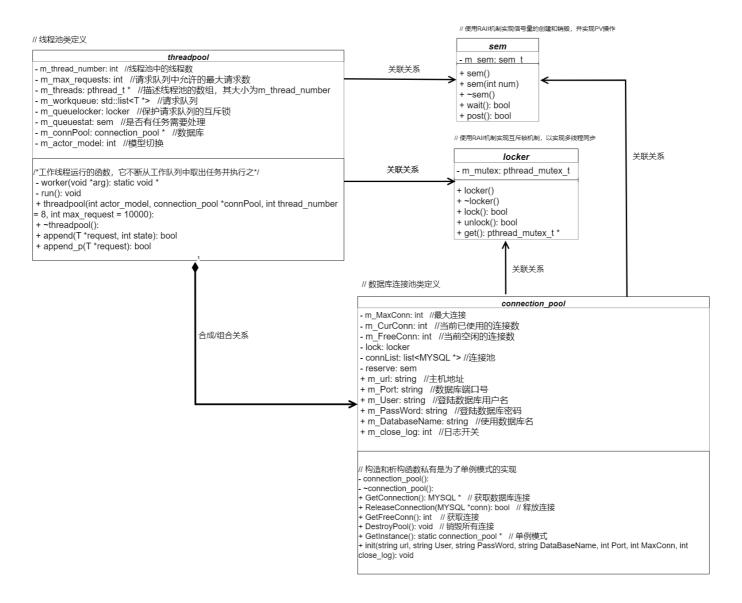
sem
- m sem: sem t
+ sem()
+ sem(int num)
+ ~sem()
+ wait(): bool
+ post(): bool

locker
- m_mutex: pthread_mutex_t
+ locker() + ~locker() + lock(): bool + unlock(): bool + get(): pthread_mutex_t *

// 使用RAII机制实现条件变量锁机制,实现多线程同步

cond - m_cond: pthread_cond_t + cond() + ~cond() + wait(pthread_mutex_t *m_mutex): bool + timewait(pthread_mutex_t *m_mutex, struct timespec t): bool + signal(): bool + broadcast(): bool

半同步半反应堆线程池



http连接处理

依赖关系

```
http_conn
    m_sockfd: int
 - III SUCKUS: IIII
- III advess: sockaddr_in
- III aread_buffREAD_BUFFER_SIZE]; char # 存储读取的请求报文数据
- III read_jox! chan # 《缓中区中》—read_buf中数据的最后一个字节的下一个位置
- III changed idx: long # / 缓中区中,是有数据的设置m_checked_idx
- III start_line: int # / III read_buf中已经解析的字符个数
 - III_sate_inits. III | Table | Table
 // 以下为解析请求报文中对应的6个变量
- m_real_file[FiLENAME_LEN]: char // 存储读取文件的名称
- m_url: char *
  - m version: char
 - m_host: char *
- m_content_length: long
- m_linger: bool
   - m_file_address: char * // 读取服务器上的文件地址
- m_file_stat: struct stat
   - m iv[2]: struct iovec // io向量机制iovec
 - m_Iv[2]: struct lovec // lo向量机制lovec - m_Iv_count: int - cgi: int // 是否启用的POST - m_string: char* // 存储请求头数据- bytes_to_send: int // 用余发送字节数 - bytes_to_send: int // 日发送字节数 - doc_root: char* - m_users: map~string, string>
  - m_close_log: int
- sql_user[100]: char
- sql_passwd[100]: char
- sql_name[100]: char
 + FILENAME_LEN=200: static const int // 设置读取文件的名称m_real_file大小
+ READ_BUFFER_SIZE=2048: static const int // 设置读缓冲区m_read_buf大小
POST,
HEAD,
                PUT,
DELETE,
                  TRACE
                  OPTIONS,
                  CONNECT
                 PATH
 」,
//主状态机的状态
//主状でいます。
+ enum CHECK_STATE{
CHECK_STATE_REQUESTLINE=0,
CHECK_STATE_HEADER,
CHECK_STATE_CONTENT
 //报文解析的结果
+ enum HTTP_CODE{
NO_REQUEST,
                 GET_REQUEST
BAD_REQUEST
                  NO RESOURCE
                  FORBIDDEN_REQUEST,
                  FILE REQUEST.
                INTERNAL_ERROR,
CLOSED_CONNECTION
 //从状态机的状态
      enum LINE_STATUS{
LINE_OK=0,
                  LINE_BAD
                 LINE OPEN
       timer_flag: int
 + improv: int
 + m_epollfd: static int
+ m_user_count: static int
+ mysql: MYSQL *
   + m state: int
  - get_line(); char*
- parse_line(); LINE_STATUS // 从状态机读取一行,分析是请求报文的哪一部分
- unmap(); void
// 根据响应报文格式,生成对应8个部分,以下函数均由do_request调用
   add_response(const char *format, ...); bool
- add_content(const char *content); bool
- add_status_line(int status, const char *title); bool
- add_headers(int content_length); bool
  - add_content_type(): bool
- add_content_length(int content_length): bool
- add_linger(): bool
- add_blank_line(): bool
+ http_conn()
+ ~http_conn()
+ ~http_conn()
+ init(int sockfd, const sockaddr_in &addr, char*, int, int, string user, string passwd, string sqiname): void // 初始化套接字地址, 函数内部会调用私有方法init
+ close_conn(bool real_close = true): void // 关闭http连接
 + close_configuor real_close = true; void // 天闪fitt
+ process(); void
+ read_once(); bool // 读取浏览器端发来的全部数据
+ write(); bool // 响应报文写入函数
 + get_address(): sockaddr_in *
+ initmysql_result(connection_pool *connPool): void // 同步线程初始化数据库读取表
```

// sql_connection_pool.h 数据库连接池类定义

```
connection_pool

m_MaxConn: int //最大连接

m_CurConn: int //当前已使用的连接数

m_FreeConn: int //当前空闲的连接数

lock: locker

connList: list<MYSQL *> //连接池

reserve: sem

# m_url: string //李陆数据库第口号

# m_User: string //李陆数据库用户名

# m_PassWord: string //李陆数据库中户名

# m_DatabaseName: string //使用数据库名

# m_close_log: int //日志开关

// 构造和析构函数私有是为了单例模式的实现

connection_pool():

-connection_pool():

-connection_pool():

GetConnection(): MYSQL * // 获取数据库连接

# ReleaseConnection(MYSQL * // 求取数据库连接

# ReleaseConnection(): // 获取连接

# CetFreeConn(): int // 获取连接

# CetInstance(): static connection_pool * // 单例模式

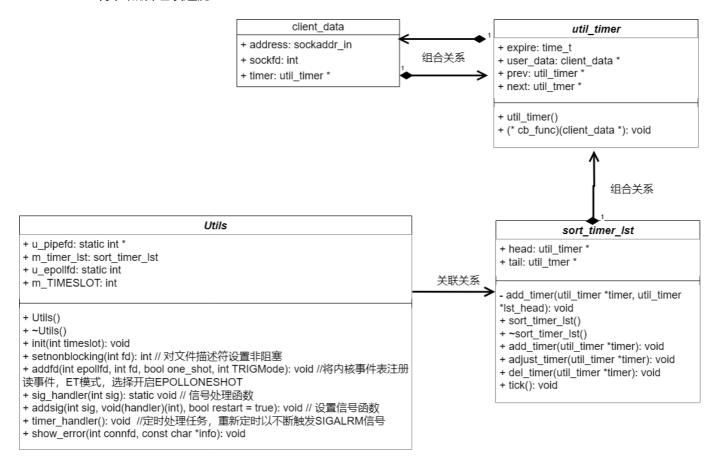
init(string url, string User, string PassWord, string DataBaseName, int Port, int MaxConn, int close_log): void
```

定时器处理非活动连接

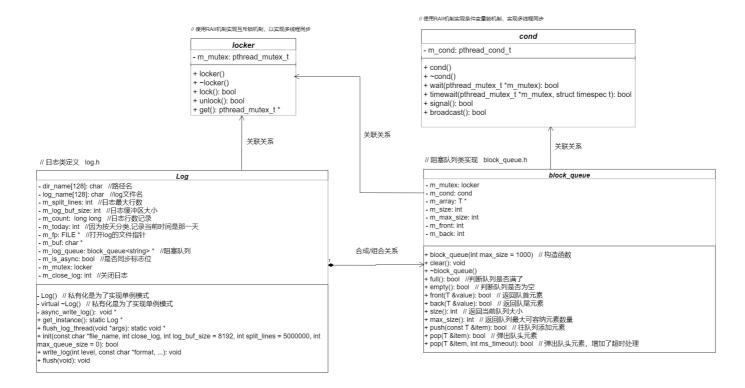
项目中将连接资源、定时事件和超时时间封装为定时器类**util_timer**,其中client_data为连接资源,具体包括客户端套接字地址、文件描述符和定时器。

定时器容器**sort_timer_lst**为带头尾结点的升序双向链表,具体的为每个连接创建一个定时器,将其添加到链表中,并按照超时时间升序排列。执行定时任务时,将到期的定时器从链表中删除。类中各函数具体逻辑如下:

- 1. add timer函数,将目标定时器添加到链表中,添加时按照升序添加
 - 1. 若当前链表中只有头尾节点,直接插入
 - 2. 否则,将定时器按升序插入
- 2. adjust_timer函数, 当定时任务发生变化,调整对应定时器在链表中的位置
 - 1. 客户端在设定时间内有数据收发,则当前时刻对该定时器重新设定时间,这里只是往后延长超时时 间
 - 2. 被调整的目标定时器在尾部,或定时器新的超时值仍然小于下一个定时器的超时,不用调整
 - 3. 否则先将定时器从链表取出, 重新插入链表
- 3. del timer函数将超时的定时器从链表中删除
 - 1. 常规双向链表删除结点
- 4. 使用统一事件源,SIGALRM信号每次被触发,主循环中调用一次定时任务处理函数 tick(),处理链表容器中到期的定时器
 - 1. 遍历定时器升序链表容器,从头结点开始依次处理每个定时器,直到遇到尚未到期的定时器
 - 2. 若当前时间小于定时器超时时间, 跳出循环, 即未找到到期的定时器
 - 3. 若当前时间大于定时器超时时间,即找到了到期的定时器,执行回调函数,然后将它从链表中删除,然后继续遍历



日志系统



数据库模块

