服务器引擎

网络库和RPC

目录

CONTENT

01 概述

02 网络库设计

03 RPC设计

04 总结展望

01

概述

背景

目的

封装业务无关的可复用高性能底层,可以在不修改底层模块的情况下快速的写各种游戏业务

为PMServer提供底层网络通信支持

定位

一个高性能的分区服分地图的跨平台MMOARPG服务器引擎

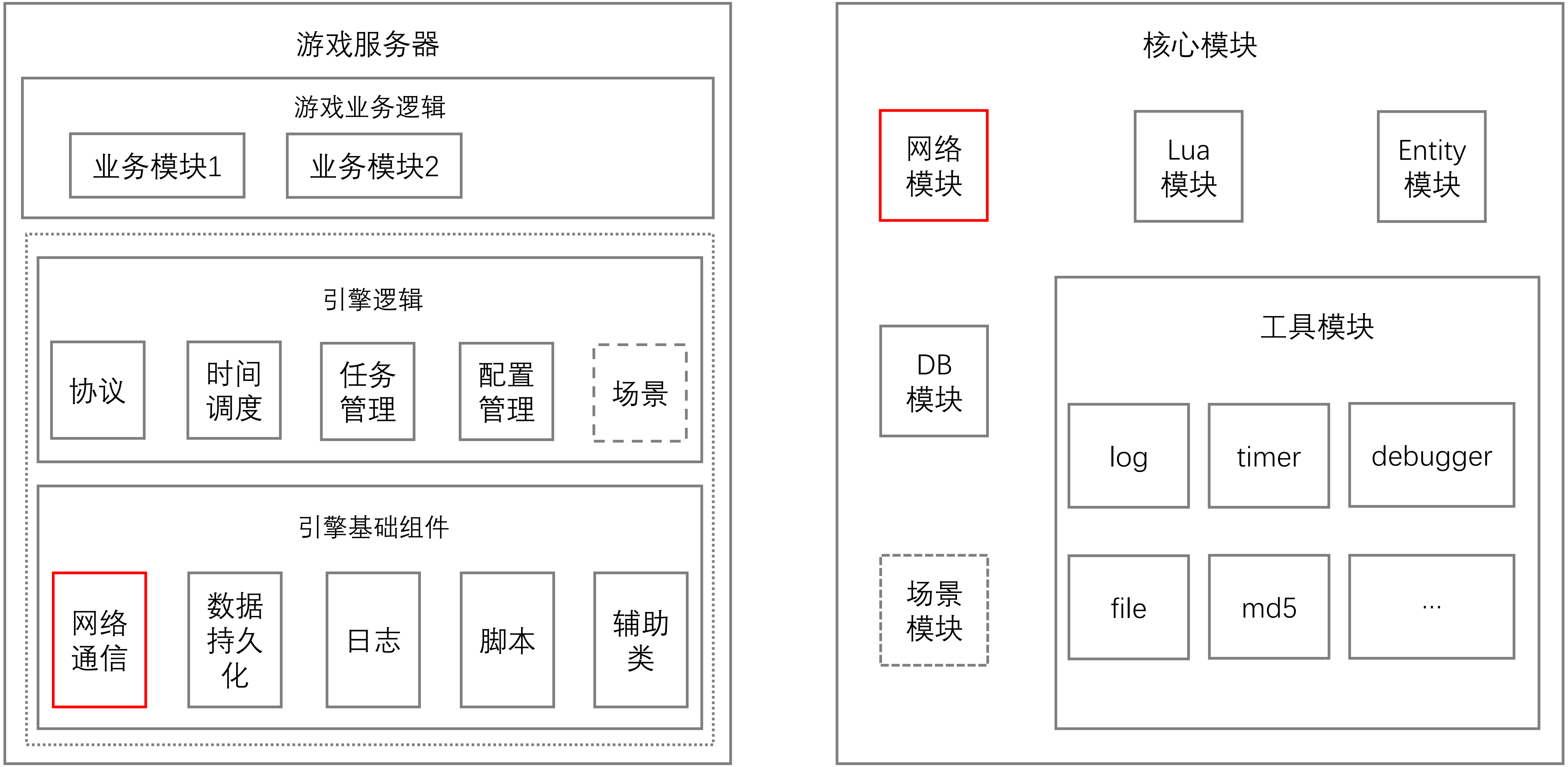
PMServer网络库

要求

代码稳定安全，易读易维护，简洁高效，跨平台

同上

逻辑分层


游戏服务器
游戏业务逻辑
业务模块1
业务模块2
引擎逻辑
协议
时间调度
任务管理
配置管理
场景
引擎基础组件
网络通信
数据持久化
日志
脚本
辅助类
核心模块
网络模块
Lua模块
Entity模块
DB模块
场景模块
工具模块
log
timer
debugger
file
md5
…

基础知识

网络I/O模型

阻塞式I/O
非阻塞式I/O
多路复用I/O
信号驱动I/O
异步I/O

《UNIX网络编程》

多线程编程

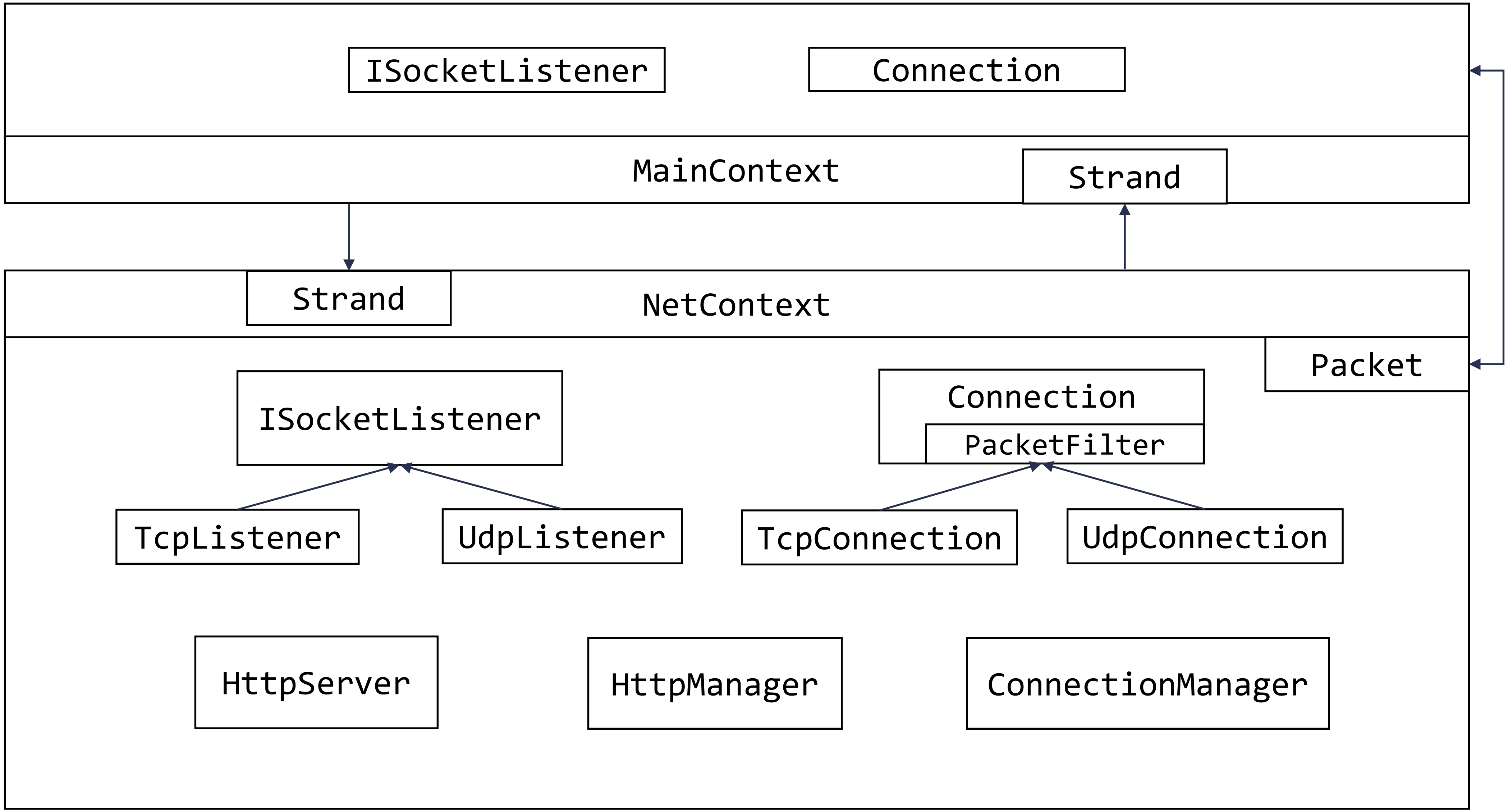
线程管理
共享数据
基于锁的并发数据结构
内存模型和原子操作
无锁并发数据结构

《C++并发编程实战》

02

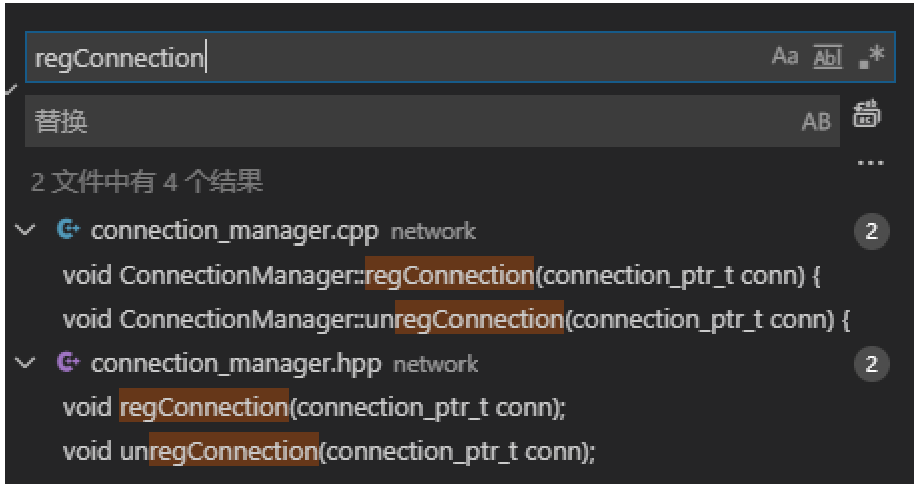
网络库设计

旧架构分析


ISocketListener
Connection
MainContext
Strand
Strand
NetContext
Packet
ISocketListener
Connection
PacketFilter
TcpListener
UdpListener
TcpConnection
UdpConnection
HttpServer
HttpManager
ConnectionManager

存在问题

对象所有权混乱



2 文件中有 4 个结果

connection_manager.cpp network 2

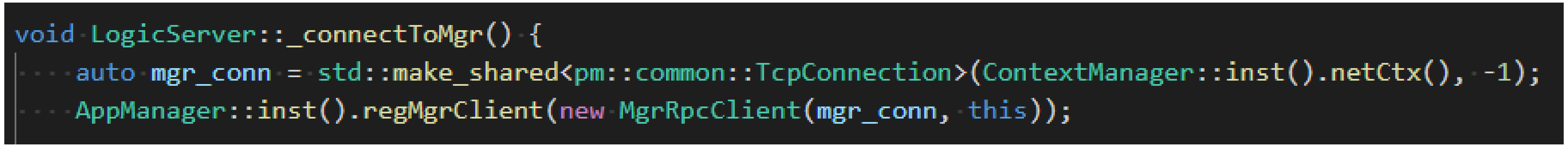
void ConnectionManager::regConnection(connection_ptr_t conn) {

void ConnectionManager::unregConnection(connection_ptr_t conn) {

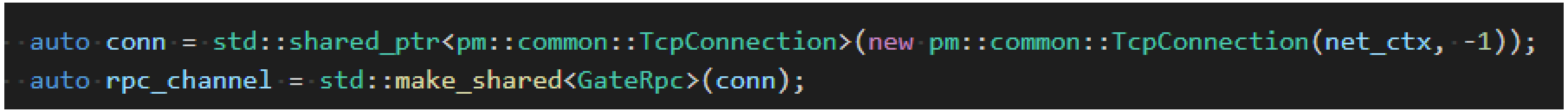
connection_manager.hpp network 2

void regConnection(connection_ptr_t conn);

void unregConnection(connection_ptr_t conn);



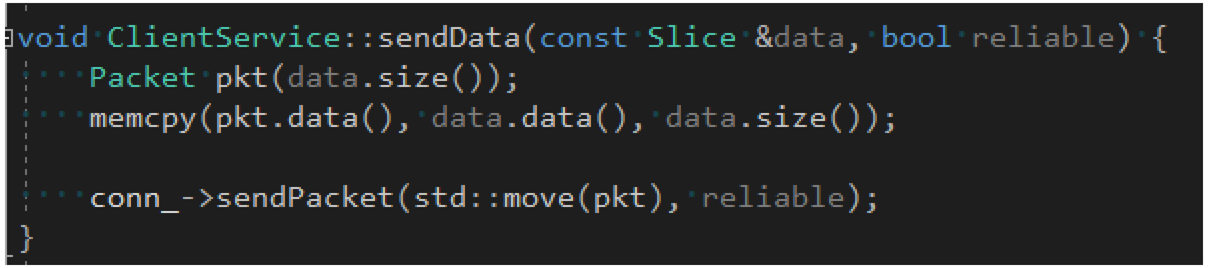
```
void LogicServer::_connectToMgr() {
    auto mgr_conn = std::make_shared<pm::common::TcpConnection>(ContextManager::inst().netCtx(), -1);
    AppManager::inst().regMgrClient(new MgrRpcClient(mgr_conn, this));
```



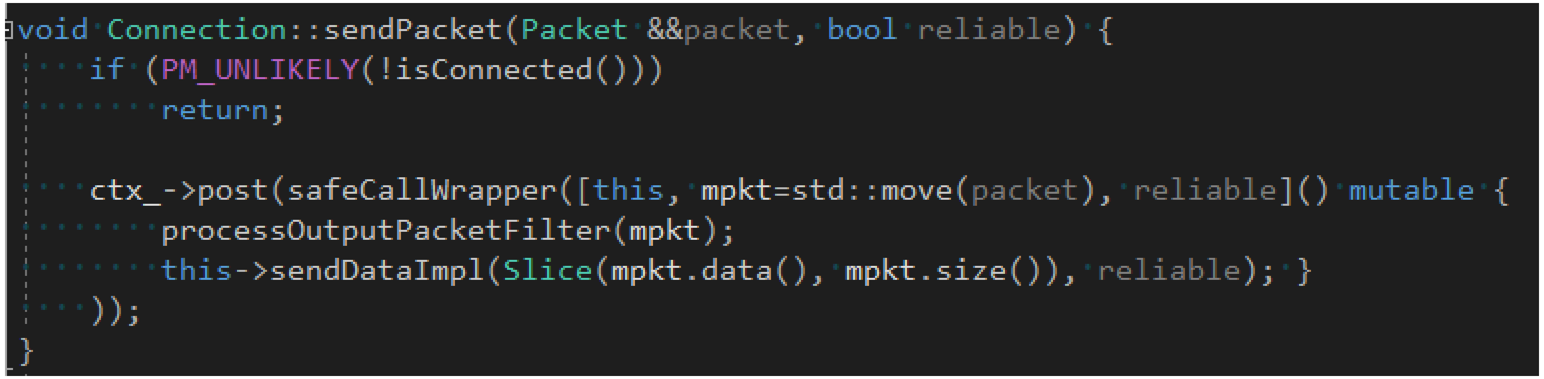
```
 auto conn = std::shared_ptr<pm::common::TcpConnection>(new pm::common::TcpConnection(net_ctx, -1));
 auto rpc_channel = std::make_shared<GateRpc>(conn);
```

存在问题

跨线程对象调用

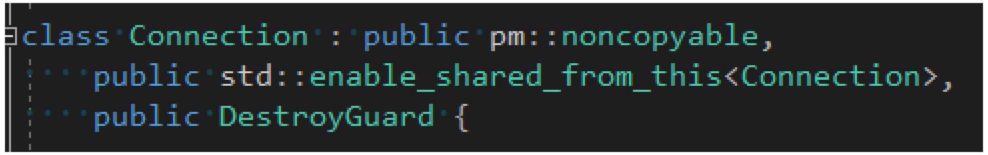


```
void ClientService::sendData(const Slice &data, bool reliable) {
    Packet pkt(data.size());
    memcpy(pkt.data(), data.data(), data.size());

    conn_->sendPacket(std::move(pkt), reliable);
}
```

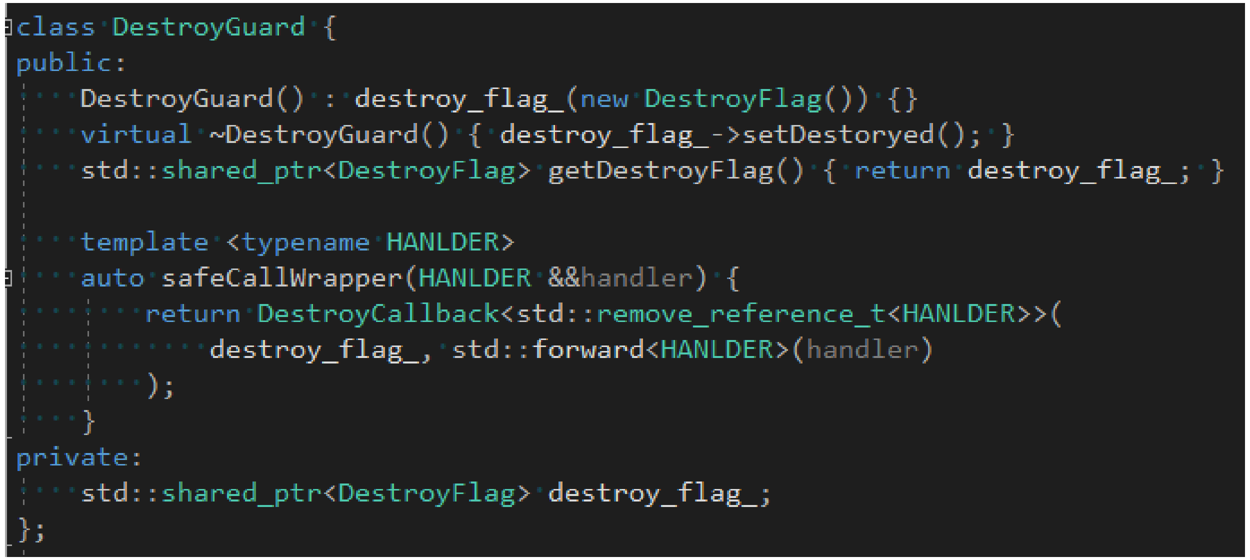


```
void Connection::sendPacket(Packet &&packet, bool reliable) {
    if (PM_UNLIKELY(!isConnected()))
        return;

    ctx_->post(safeCallWrapper([this, mpkt=std::move(packet), reliable]() mutable {
        processOutputPacketFilter(mpkt);
        this->sendDataImpl(Slice(mpkt.data(), mpkt.size()), reliable); }
    ));
}
```

存在问题

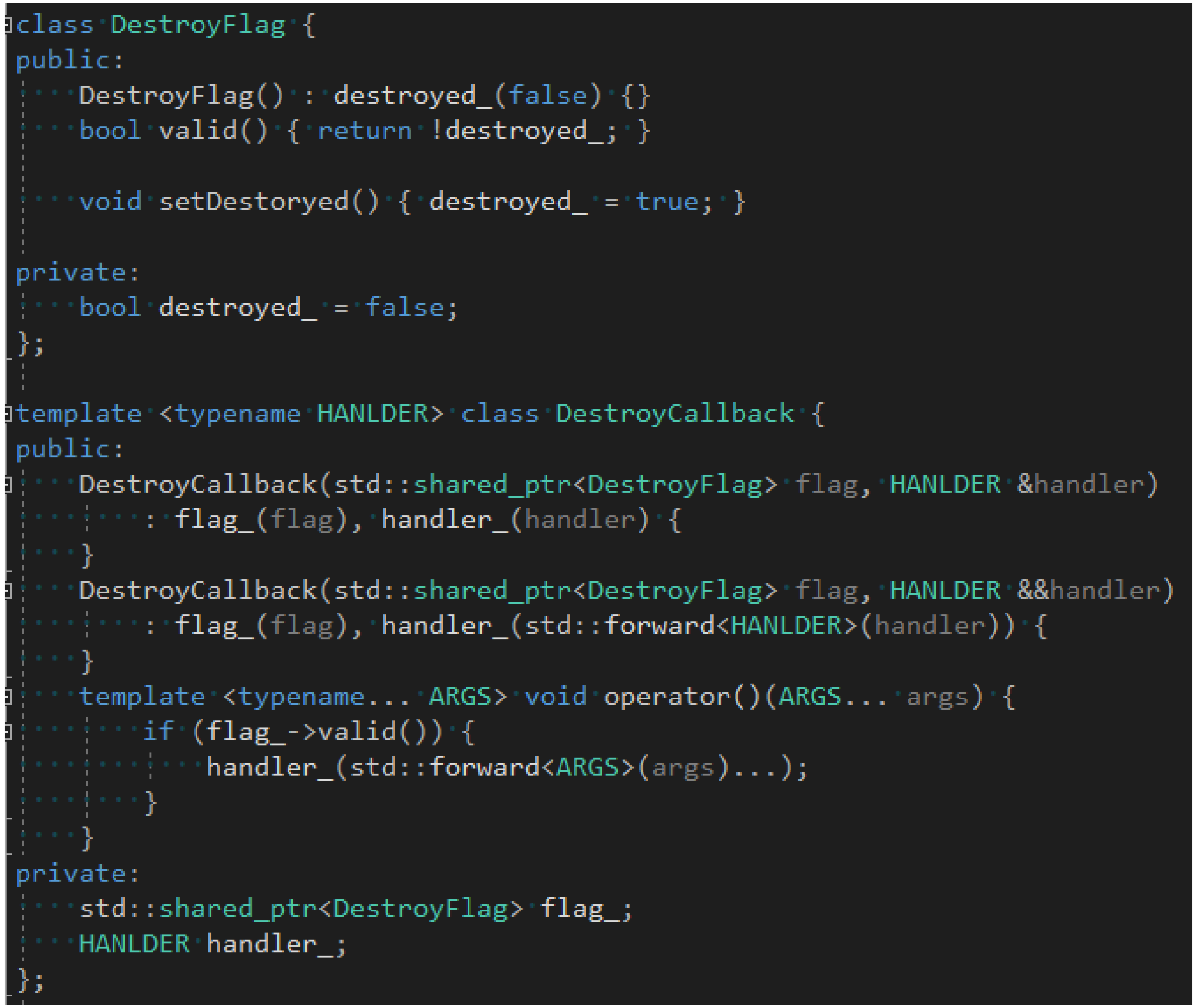


```
class Connection : public pm::noncopyable,
    public std::enable_shared_from_this<Connection>,
    public DestroyGuard {
```



```
class DestroyGuard {
public:
    DestroyGuard() : destroy_flag_(new DestroyFlag()) {}
    virtual ~DestroyGuard() { destroy_flag_->setDestoryed(); }
    std::shared_ptr<DestroyFlag> getDestroyFlag() { return destroy_flag_; }

    template <typename HANLDER>
    auto safeCallWrapper(HANLDER &&handler) {
        return DestroyCallback<std::remove_reference_t<HANLDER>>(
            destroy_flag_, std::forward<HANLDER>(handler)
        );
    }
private:
    std::shared_ptr<DestroyFlag> destroy_flag_;
};
```

DestroyGuard并没有起到作用



```
class DestroyFlag {
public:
    DestroyFlag() : destroyed_(false) {}
    bool valid() { return !destroyed_; }

    void setDestoryed() { destroyed_ = true; }

private:
    bool destroyed_ = false;
};

template <typename HANLDER> class DestroyCallback {
public:
    DestroyCallback(std::shared_ptr<DestroyFlag> flag, HANLDER &handler)
        : flag_(flag), handler_(handler) {
    }
    DestroyCallback(std::shared_ptr<DestroyFlag> flag, HANLDER &&handler)
        : flag_(flag), handler_(std::forward<HANLDER>(handler)) {
    }
    template <typename... ARGS> void operator()(ARGS... args) {
        if (flag_->valid()) {
            handler_(std::forward<ARGS>(args)...);
        }
    }
private:
    std::shared_ptr<DestroyFlag> flag_;
    HANLDER handler_;
};
```

重构目标

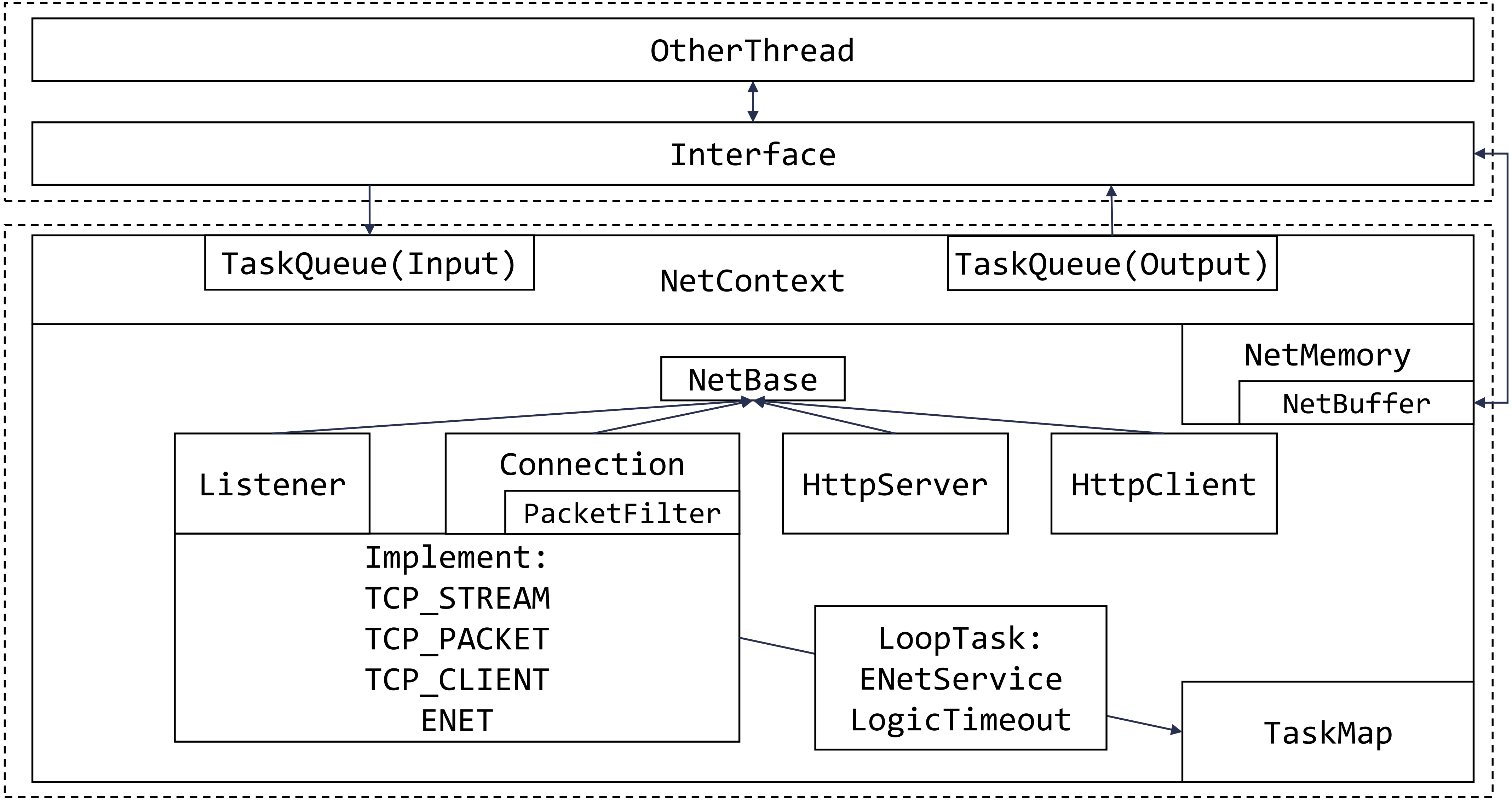
优点保留:

1. 清晰的继承关系
2. 易用的接口
3. 完善的具体实现

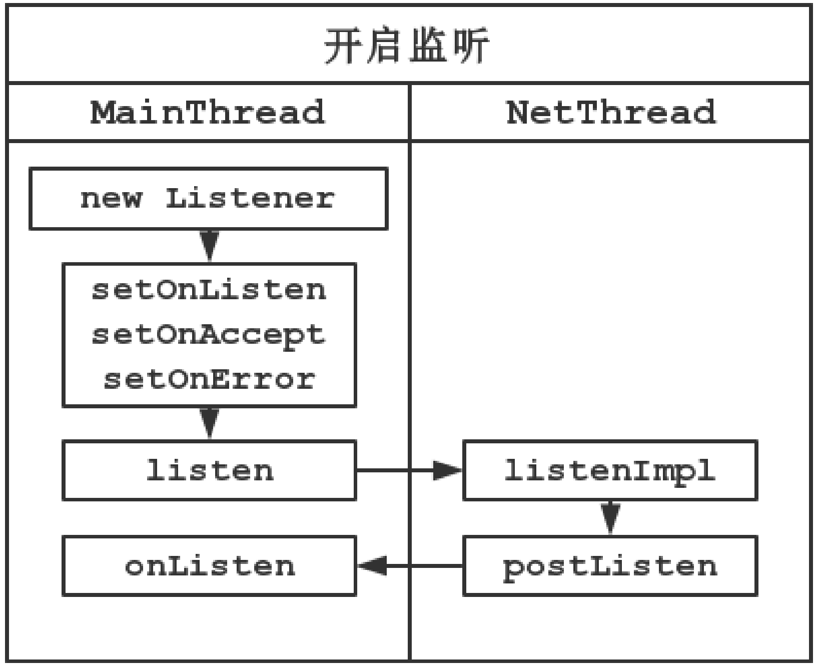
缺点改进:

1. 明确对象所有权
2. 避免跨线程调用
3. 提高代码可读性，易维护

架构设计


OtherThread
Interface
TaskQueue(Input)
NetContext
TaskQueue(Output)
NetMemory
NetBuffer
NetBase
Listener
Connection
PacketFilter
HttpServer
HttpClient
Implement:
TCP_STREAM
TCP_PACKET
TCP_CLIENT
ENET
LoopTask:
ENetService
LogicTimeout
TaskMap

流程图



开启监听

MainThread

NetThread

new Listener

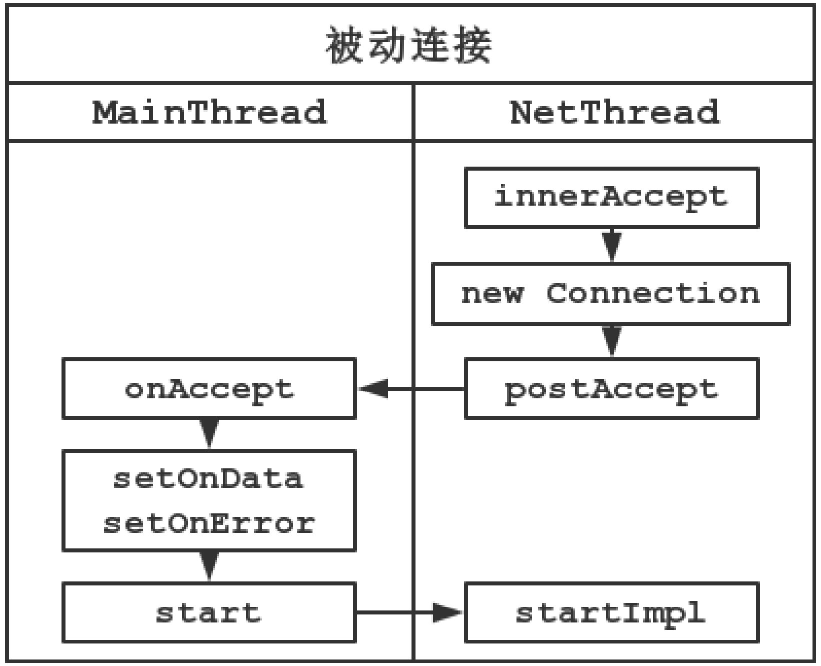
setOnListen
setOnAccept
setOnError

listen

listenImpl

postListen

onListen



被动连接

MainThread

NetThread

innerAccept

new Connection

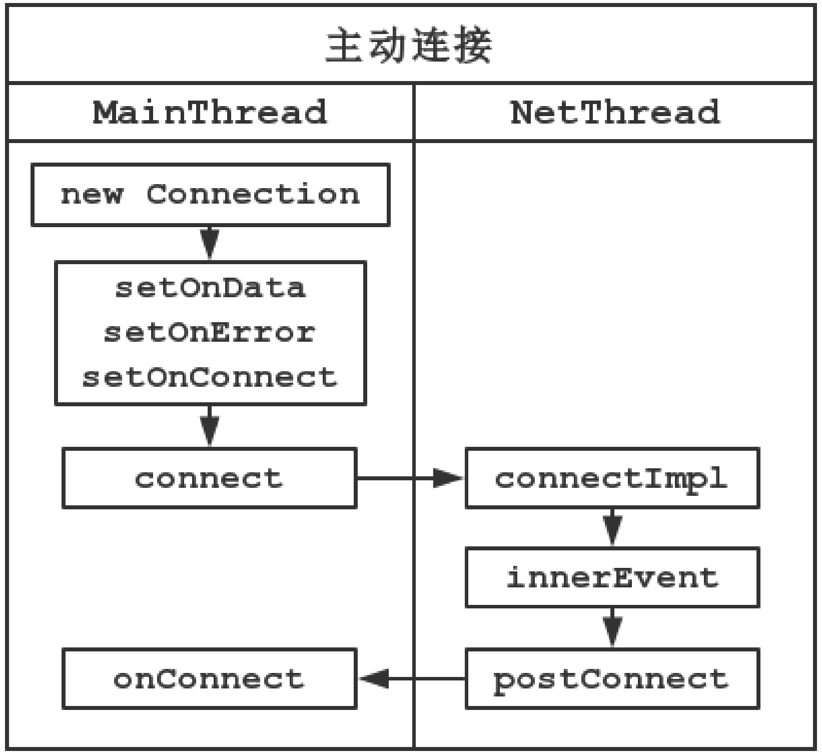
postAccept

onAccept

setOnData
setOnError

start

startImpl



主动连接

MainThread

NetThread

new Connection

setOnData
setOnError
setOnConnect

connect

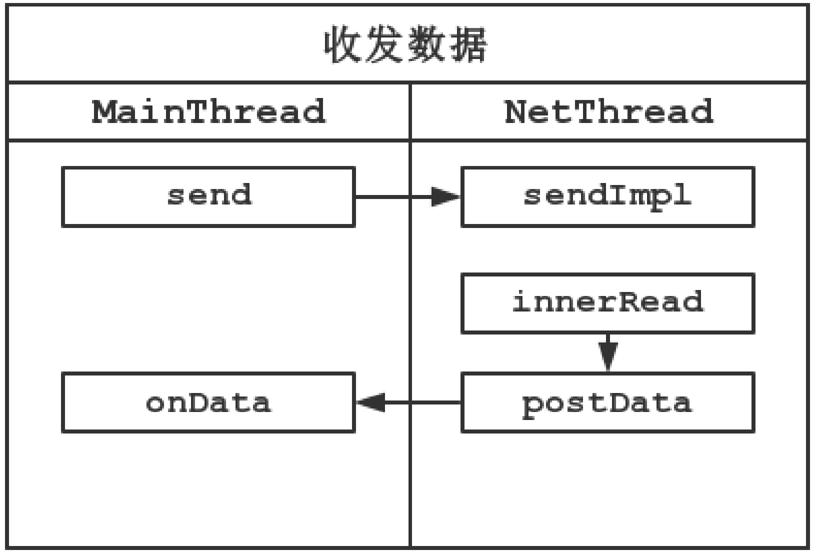
connectImpl

innerEvent

postConnect

onConnect

流程图



收发数据

MainThread

NetThread

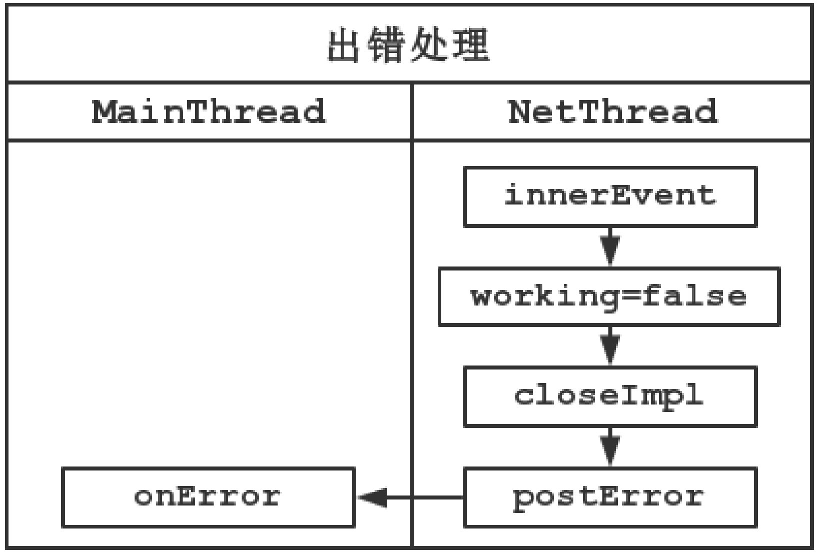
send

sendImpl

innerRead

onData

postData



出错处理

MainThread

NetThread

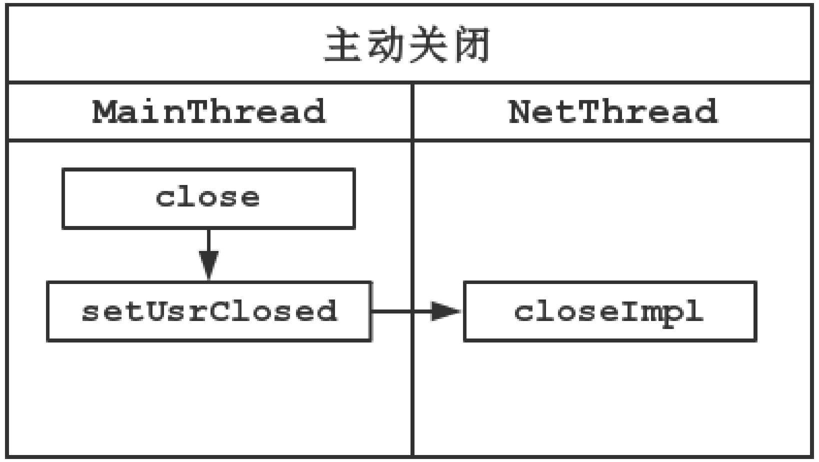
innerEvent

working=false

closeImpl

onError

postError



主动关闭

MainThread

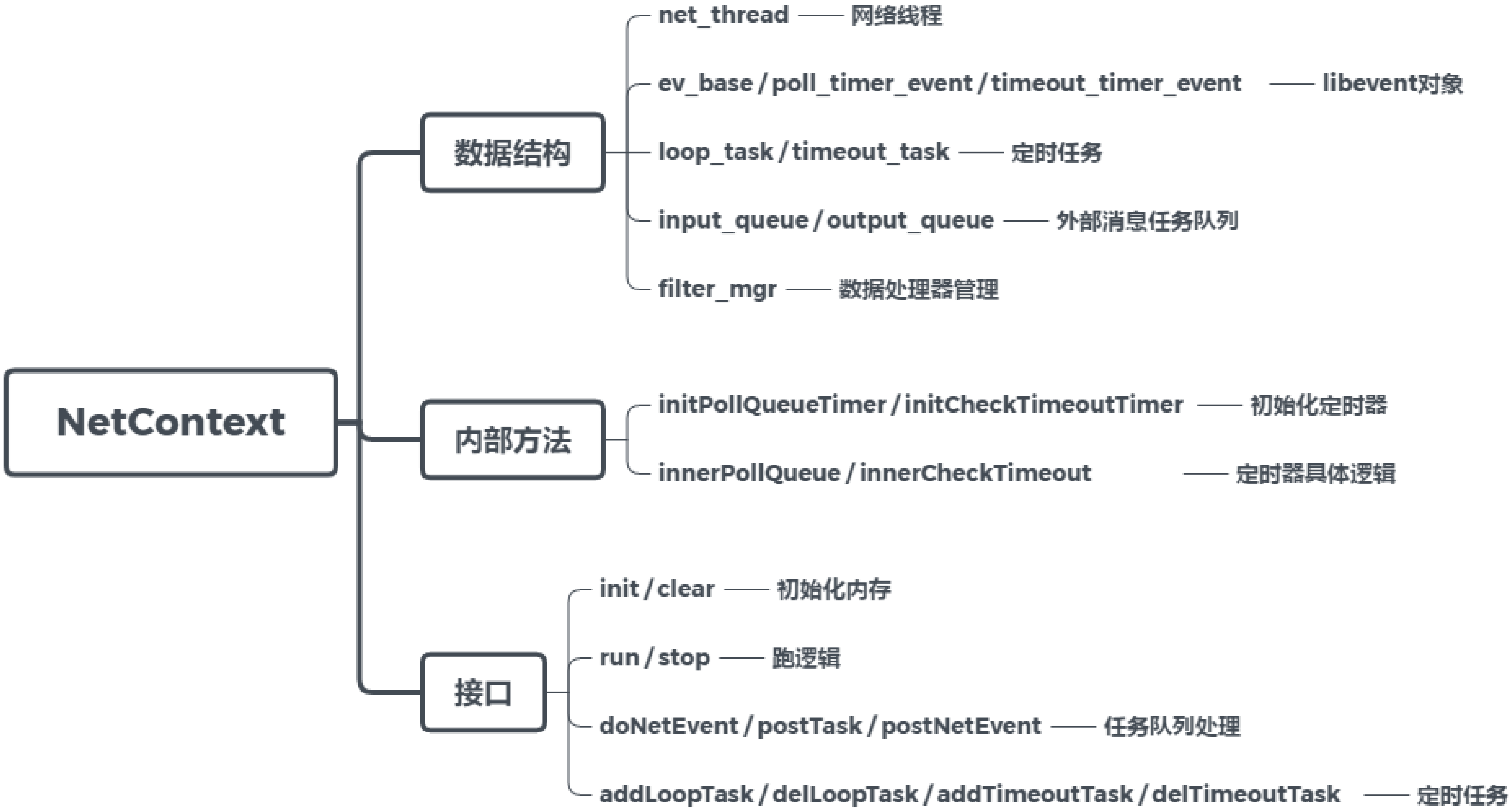
NetThread

close

setUsrClosed

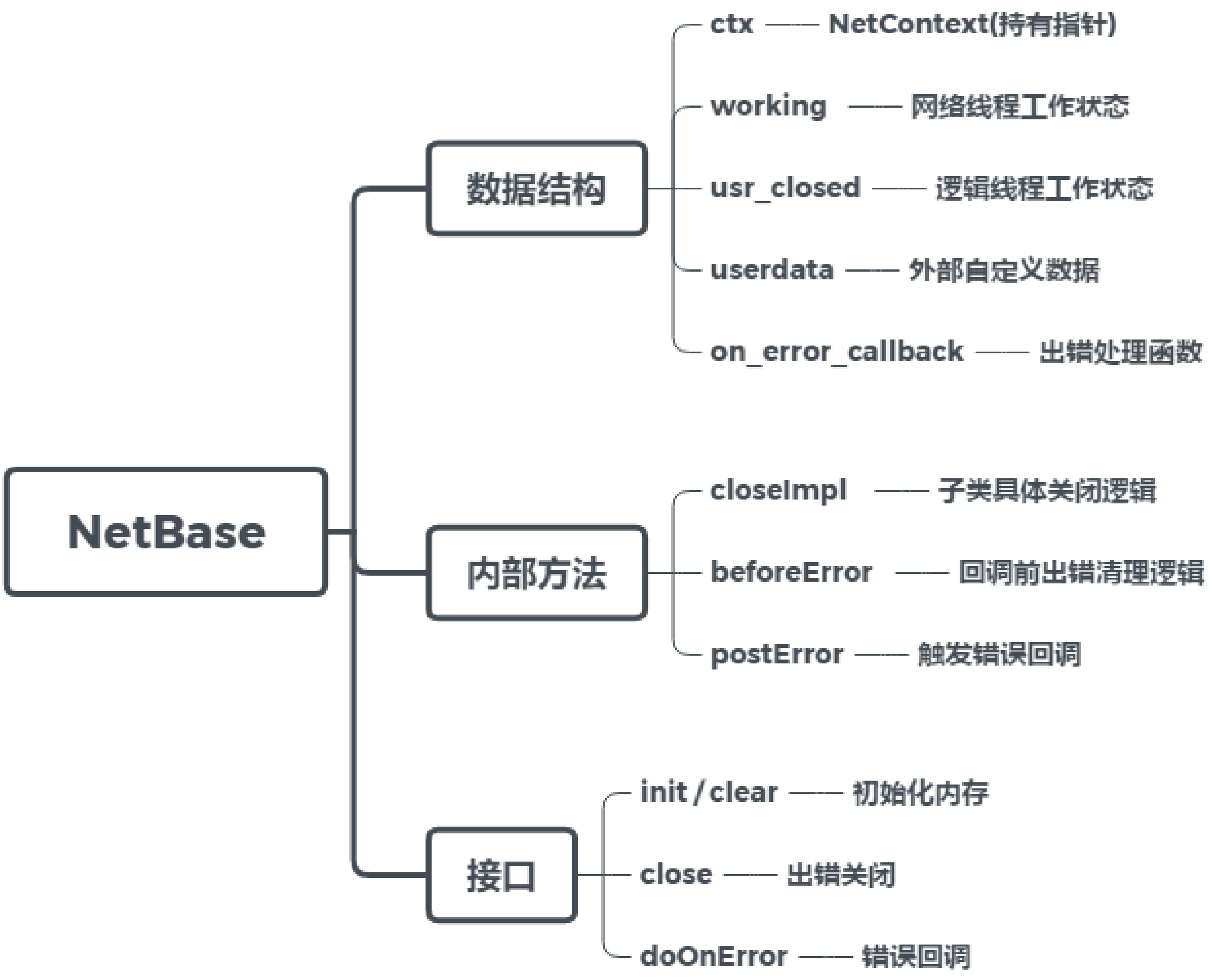
closeImpl

线程管理



- NetContext
  - 数据结构
    - net_thread —— 网络线程
    - ev_base / poll_timer_event / timeout_timer_event —— libevent对象
    - loop_task / timeout_task —— 定时任务
    - input_queue / output_queue —— 外部消息任务队列
    - filter_mgr —— 数据处理器管理
  - 内部方法
    - initPollQueueTimer / initCheckTimeoutTimer —— 初始化定时器
    - innerPollQueue / innerCheckTimeout —— 定时器具体逻辑
  - 接口
    - init / clear —— 初始化内存
    - run / stop —— 跑逻辑
    - doNetEvent / postTask / postNetEvent —— 任务队列处理
    - addLoopTask / delLoopTask / addTimeoutTask / delTimeoutTask —— 定时任务

继承模型

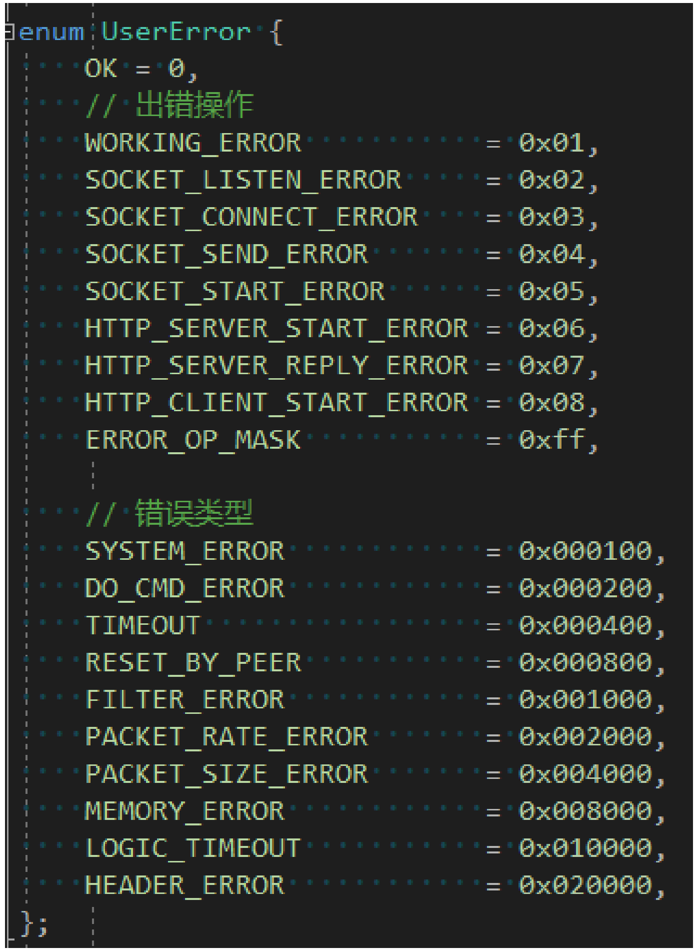


- NetBase
  - 数据结构
    - ctx —— NetContext(持有指针)
    - working —— 网络线程工作状态
    - usr_closed —— 逻辑线程工作状态
    - userdata —— 外部自定义数据
    - on_error_callback —— 出错处理函数
  - 内部方法
    - closeImpl —— 子类具体关闭逻辑
    - beforeError —— 回调前出错清理逻辑
    - postError —— 触发错误回调
  - 接口
    - init / clear —— 初始化内存
    - close —— 出错关闭
    - doOnError —— 错误回调

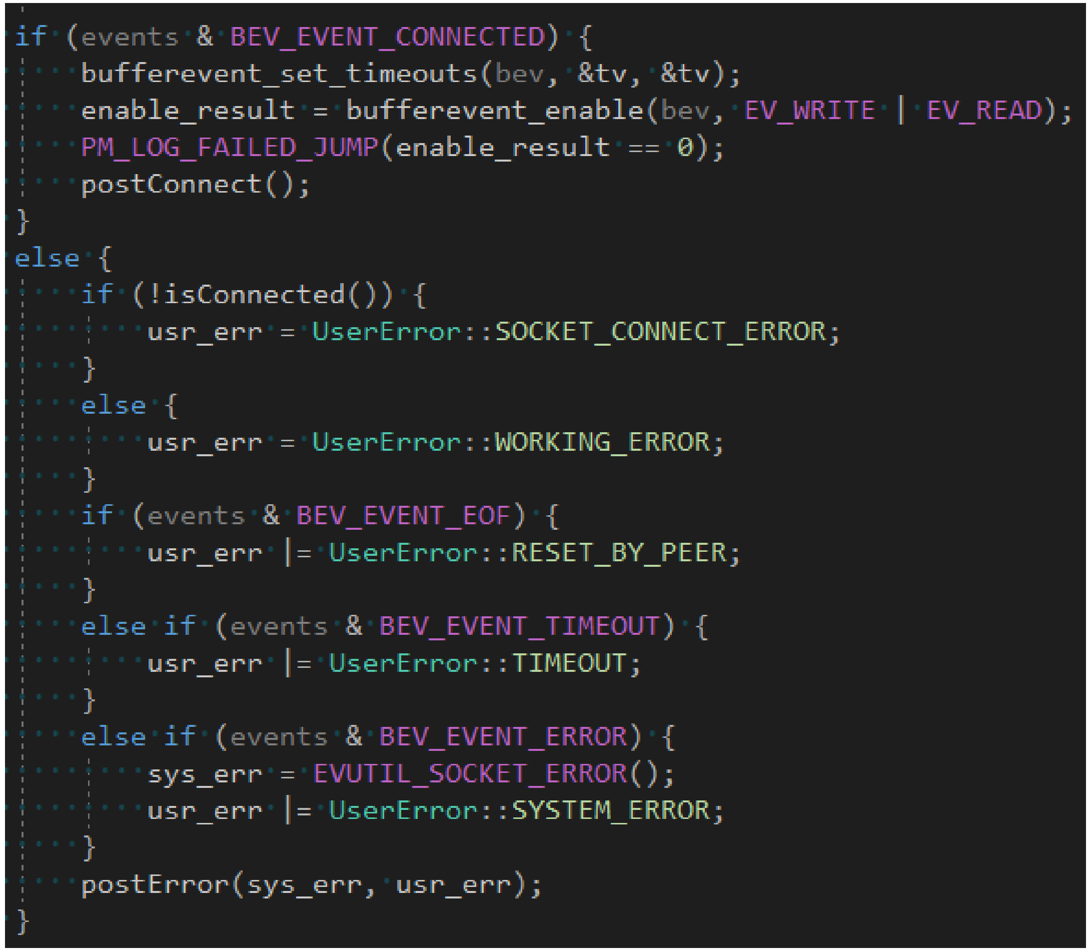
为什么只有onError没有onClose?

close是主动行为，逻辑调用了close之后，就可以认为已经close了，不需要等回调
error是网络线程执行异常，需要通知逻辑

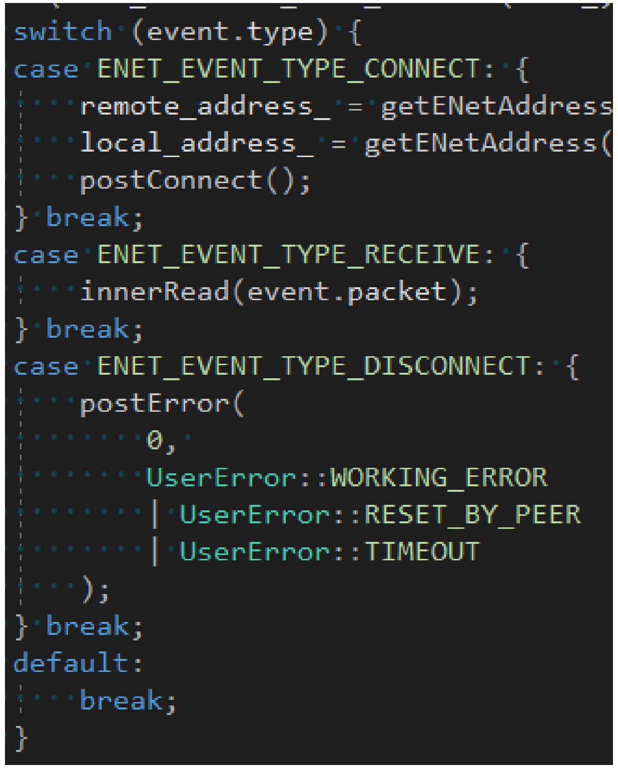
错误处理



```
enum UserError {
    OK = 0,
    // 出错操作
    WORKING_ERROR              = 0x01,
    SOCKET_LISTEN_ERROR        = 0x02,
    SOCKET_CONNECT_ERROR       = 0x03,
    SOCKET_SEND_ERROR          = 0x04,
    SOCKET_START_ERROR         = 0x05,
    HTTP_SERVER_START_ERROR    = 0x06,
    HTTP_SERVER_REPLY_ERROR    = 0x07,
    HTTP_CLIENT_START_ERROR    = 0x08,
    ERROR_OP_MASK              = 0xff,

    // 错误类型
    SYSTEM_ERROR               = 0x000100,
    DO_CMD_ERROR               = 0x000200,
    TIMEOUT                    = 0x000400,
    RESET_BY_PEER              = 0x000800,
    FILTER_ERROR               = 0x001000,
    PACKET_RATE_ERROR          = 0x002000,
    PACKET_SIZE_ERROR          = 0x004000,
    MEMORY_ERROR               = 0x008000,
    LOGIC_TIMEOUT              = 0x010000,
    HEADER_ERROR               = 0x020000,
};
```

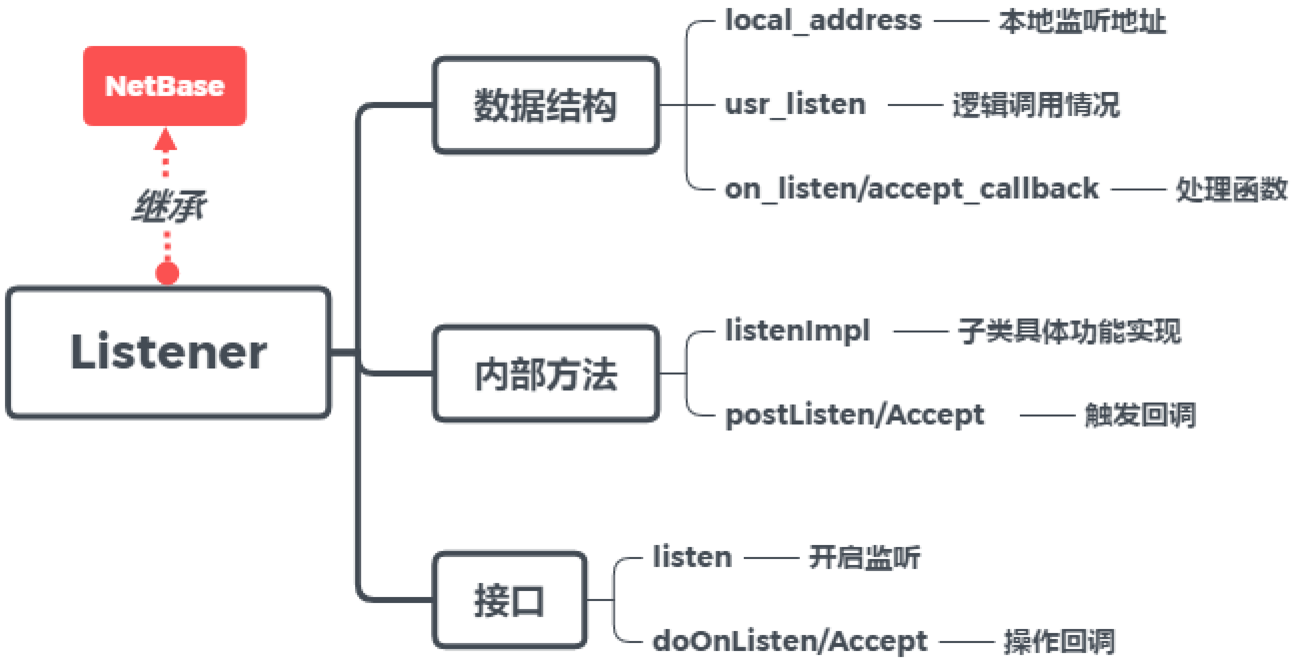


```
if (events & BEV_EVENT_CONNECTED) {
    bufferevent_set_timeouts(bev, &tv, &tv);
    enable_result = bufferevent_enable(bev, EV_WRITE | EV_READ);
    PM_LOG_FAILED_JUMP(enable_result == 0);
    postConnect();
}
else {
    if (!isConnected()) {
        usr_err = UserError::SOCKET_CONNECT_ERROR;
    }
    else {
        usr_err = UserError::WORKING_ERROR;
    }
    if (events & BEV_EVENT_EOF) {
        usr_err |= UserError::RESET_BY_PEER;
    }
    else if (events & BEV_EVENT_TIMEOUT) {
        usr_err |= UserError::TIMEOUT;
    }
    else if (events & BEV_EVENT_ERROR) {
        sys_err = EVUTIL_SOCKET_ERROR();
        usr_err |= UserError::SYSTEM_ERROR;
    }
    postError(sys_err, usr_err);
}
```

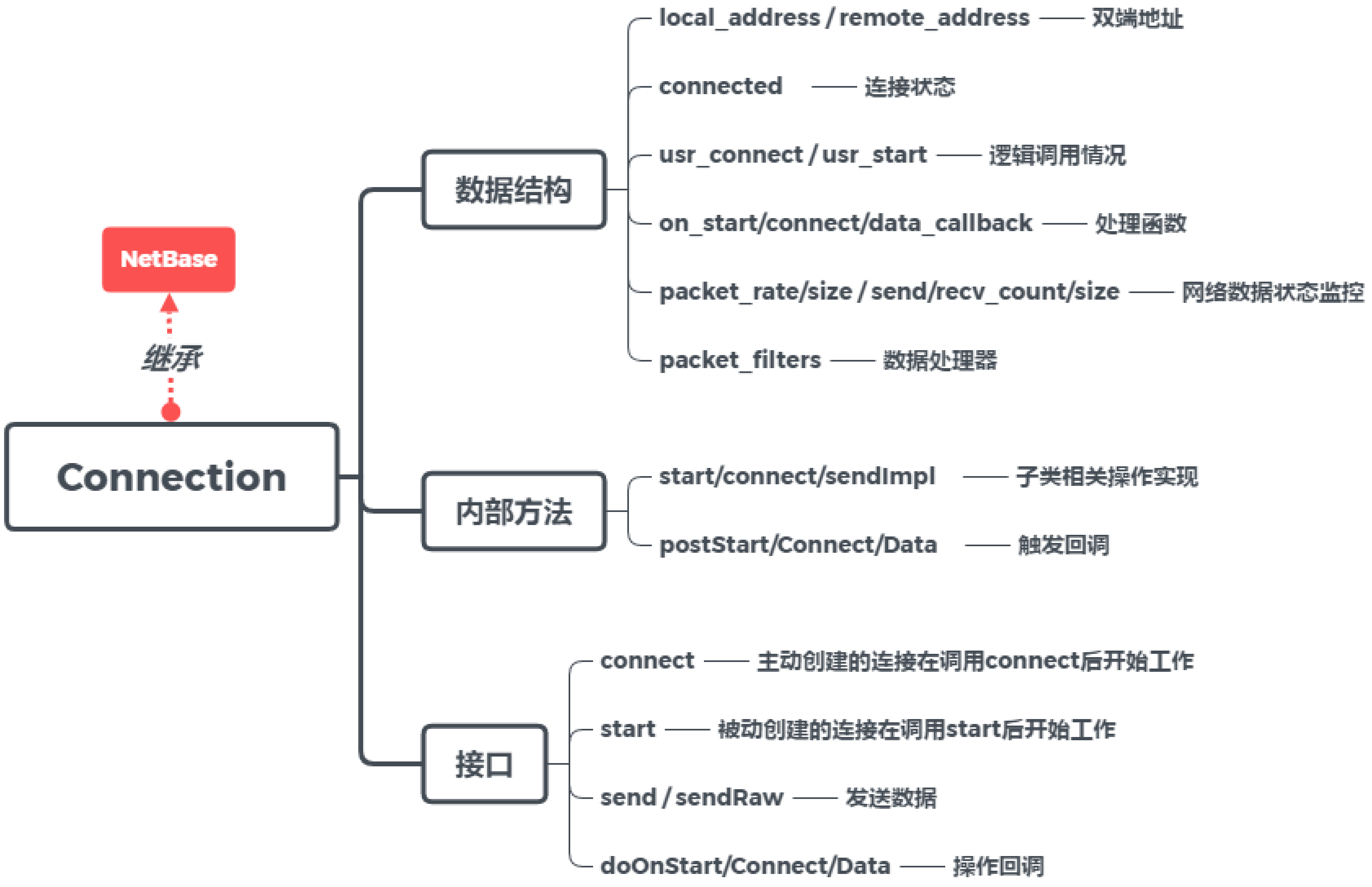


```
switch (event.type) {
case ENET_EVENT_TYPE_CONNECT: {
    remote_address_ = getENetAddress
    local_address_ = getENetAddress(
    postConnect();
} break;
case ENET_EVENT_TYPE_RECEIVE: {
    innerRead(event.packet);
} break;
case ENET_EVENT_TYPE_DISCONNECT: {
    postError(
        0,
        UserError::WORKING_ERROR
        | UserError::RESET_BY_PEER
        | UserError::TIMEOUT
    );
} break;
default:
    break;
}
```

继承模型


NetBase
继承
Listener
数据结构
local_address —— 本地监听地址
usr_listen —— 逻辑调用情况
on_listen/accept_callback —— 处理函数
内部方法
listenImpl —— 子类具体功能实现
postListen/Accept —— 触发回调
接口
listen —— 开启监听
doOnListen/Accept —— 操作回调

继承模型



NetBase

继承

Connection

数据结构

- local_address / remote_address —— 双端地址
- connected —— 连接状态
- usr_connect / usr_start —— 逻辑调用情况
- on_start/connect/data_callback —— 处理函数
- packet_rate/size / send/recv_count/size —— 网络数据状态监控
- packet_filters —— 数据处理器

内部方法

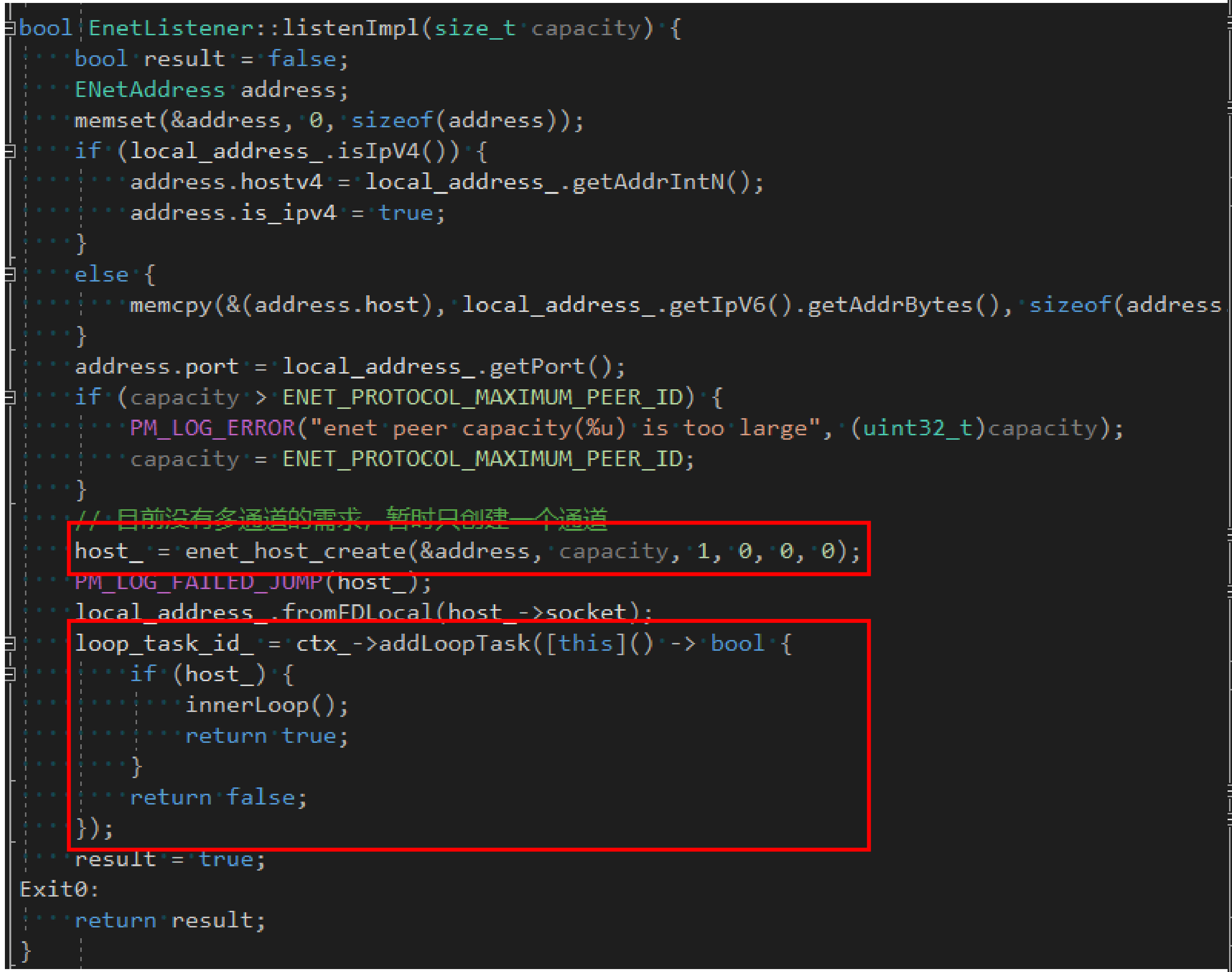
- start/connect/sendImpl —— 子类相关操作实现
- postStart/Connect/Data —— 触发回调

接口

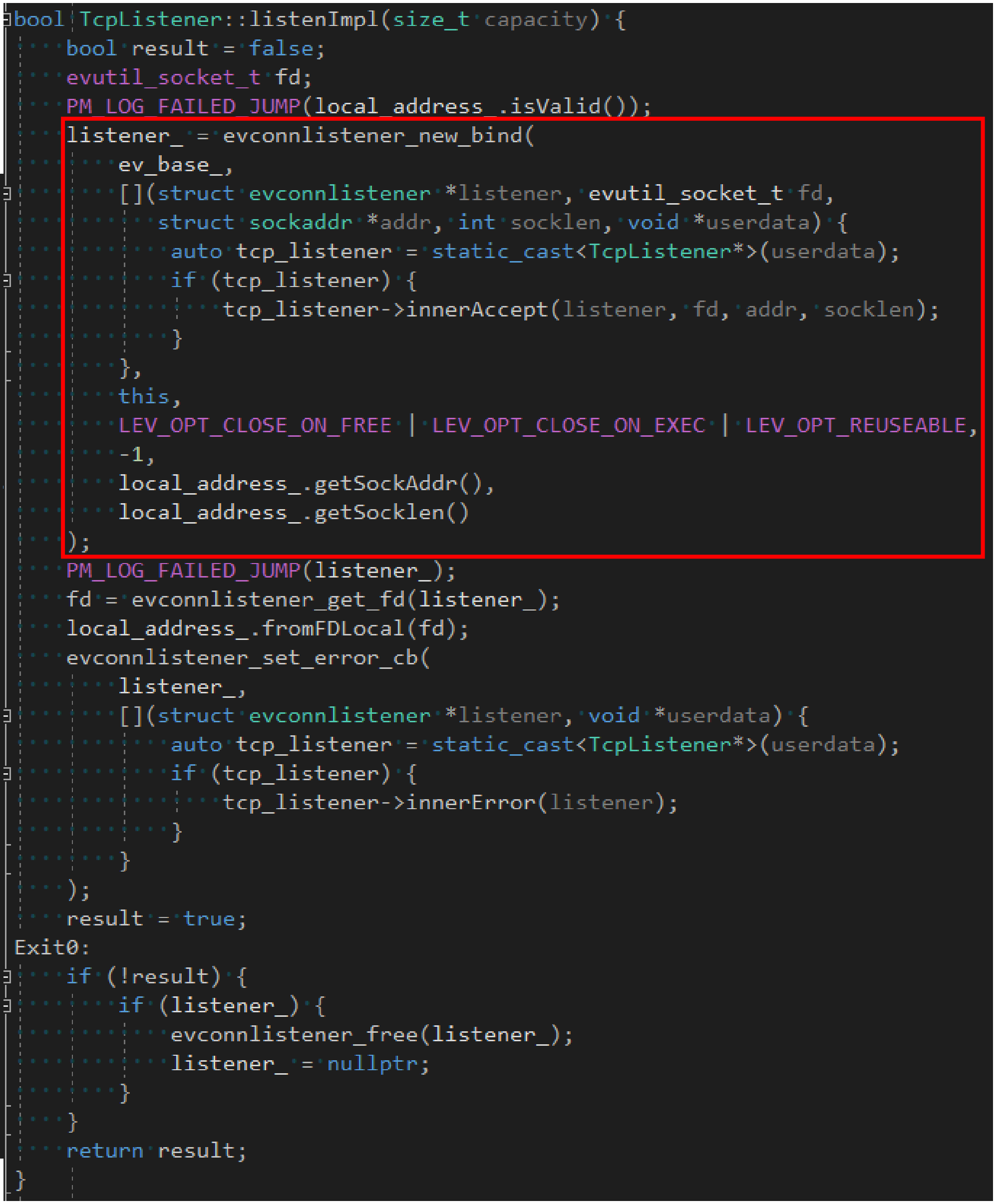
- connect —— 主动创建的连接在调用connect后开始工作
- start —— 被动创建的连接在调用start后开始工作
- send / sendRaw —— 发送数据
- doOnStart/Connect/Data —— 操作回调

继承模型

具体子类实现具体逻辑



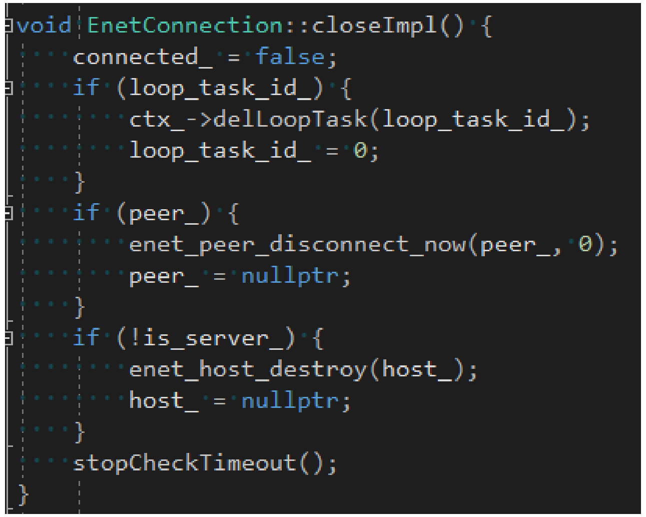
```cpp
bool EnetListener::listenImpl(size_t capacity) {
    bool result = false;
    ENetAddress address;
    memset(&address, 0, sizeof(address));
    if (local_address_.isIpV4()) {
        address.hostv4 = local_address_.getAddrIntN();
        address.is_ipv4 = true;
    }
    else {
        memcpy(&(address.host), local_address_.getIpV6().getAddrBytes(), sizeof(address
    }
    address.port = local_address_.getPort();
    if (capacity > ENET_PROTOCOL_MAXIMUM_PEER_ID) {
        PM_LOG_ERROR("enet peer capacity(%u) is too large", (uint32_t)capacity);
        capacity = ENET_PROTOCOL_MAXIMUM_PEER_ID;
    }
    // 目前没有多通道的需求，暂时只创建一个通道
    host_ = enet_host_create(&address, capacity, 1, 0, 0, 0);
    PM_LOG_FAILED_JUMP(host_);
    local_address_.fromFDLocal(host_->socket);
    loop_task_id_ = ctx_->addLoopTask([this]() -> bool {
        if (host_) {
            innerLoop();
            return true;
        }
        return false;
    });
    result = true;
Exit0:
    return result;
}
```



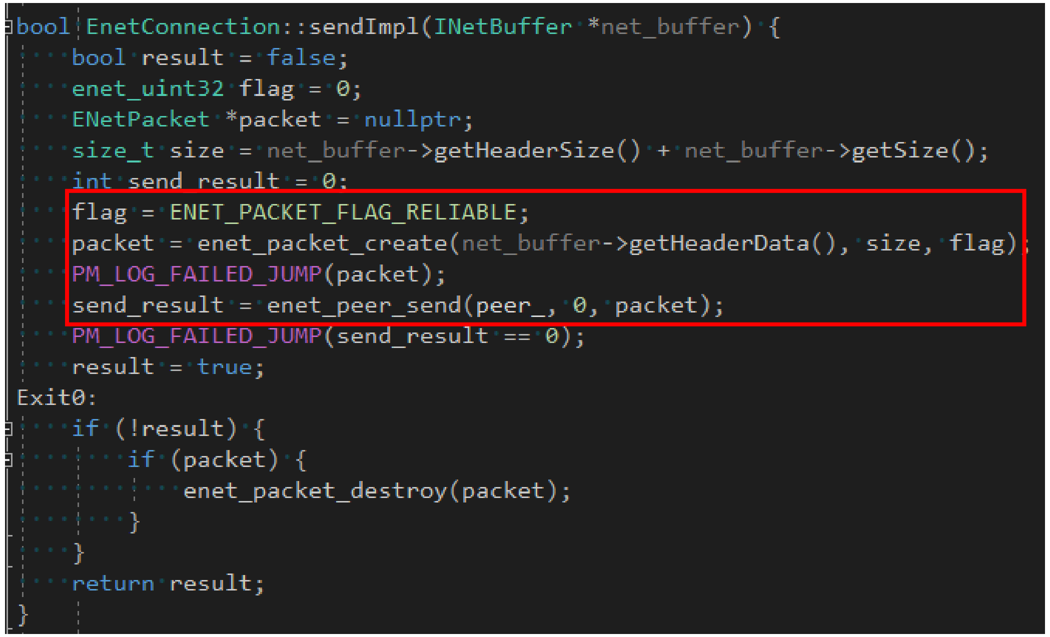
```cpp
bool TcpListener::listenImpl(size_t capacity) {
    bool result = false;
    evutil_socket_t fd;
    PM_LOG_FAILED_JUMP(local_address_.isValid());
    listener_ = evconnlistener_new_bind(
        ev_base_,
        [](struct evconnlistener *listener, evutil_socket_t fd,
            struct sockaddr *addr, int socklen, void *userdata) {
            auto tcp_listener = static_cast<TcpListener*>(userdata);
            if (tcp_listener) {
                tcp_listener->innerAccept(listener, fd, addr, socklen);
            }
        },
        this,
        LEV_OPT_CLOSE_ON_FREE | LEV_OPT_CLOSE_ON_EXEC | LEV_OPT_REUSEABLE,
        -1,
        local_address_.getSockAddr(),
        local_address_.getSocklen()
    );
    PM_LOG_FAILED_JUMP(listener_);
    fd = evconnlistener_get_fd(listener_);
    local_address_.fromFDLocal(fd);
    evconnlistener_set_error_cb(
        listener_,
        [](struct evconnlistener *listener, void *userdata) {
            auto tcp_listener = static_cast<TcpListener*>(userdata);
            if (tcp_listener) {
                tcp_listener->innerError(listener);
            }
        }
    );
    result = true;
Exit0:
    if (!result) {
        if (listener_) {
            evconnlistener_free(listener_);
            listener_ = nullptr;
        }
    }
    return result;
}
```

继承模型

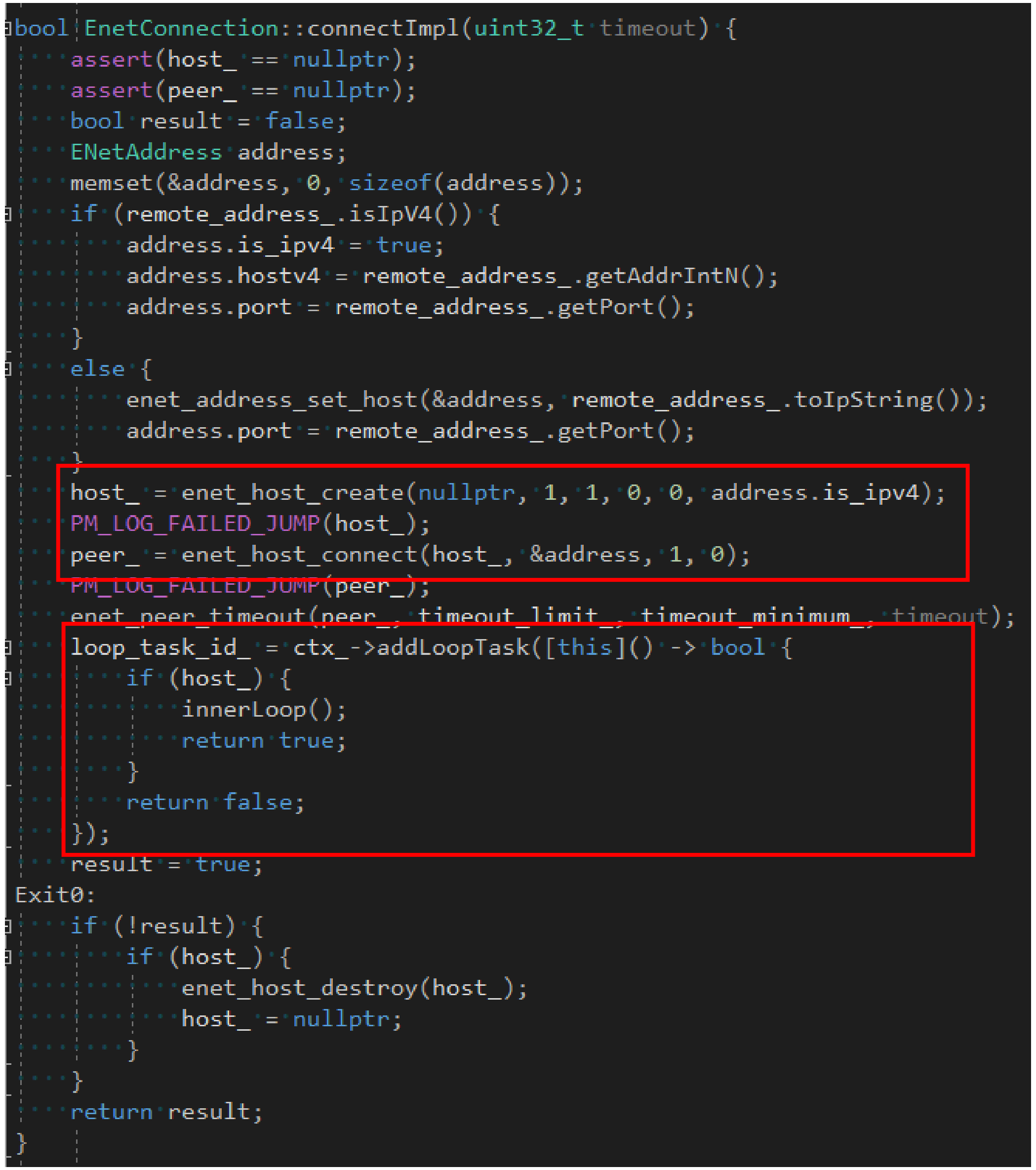
具体子类实现具体逻辑



```cpp
void EnetConnection::closeImpl() {
    connected_ = false;
    if (loop_task_id_) {
        ctx_->delLoopTask(loop_task_id_);
        loop_task_id_ = 0;
    }
    if (peer_) {
        enet_peer_disconnect_now(peer_, 0);
        peer_ = nullptr;
    }
    if (!is_server_) {
        enet_host_destroy(host_);
        host_ = nullptr;
    }
    stopCheckTimeout();
}
```



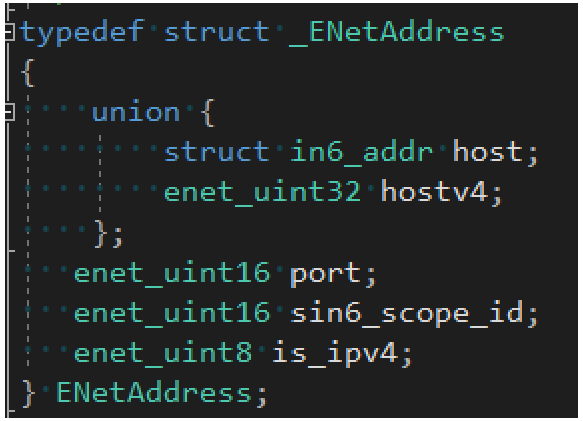
```cpp
bool EnetConnection::sendImpl(INetBuffer *net_buffer) {
    bool result = false;
    enet_uint32 flag = 0;
    ENetPacket *packet = nullptr;
    size_t size = net_buffer->getHeaderSize() + net_buffer->getSize();
    int send_result = 0;
    flag = ENET_PACKET_FLAG_RELIABLE;
    packet = enet_packet_create(net_buffer->getHeaderData(), size, flag);
    PM_LOG_FAILED_JUMP(packet);
    send_result = enet_peer_send(peer_, 0, packet);
    PM_LOG_FAILED_JUMP(send_result == 0);
    result = true;
Exit0:
    if (!result) {
        if (packet) {
            enet_packet_destroy(packet);
        }
    }
    return result;
}
```



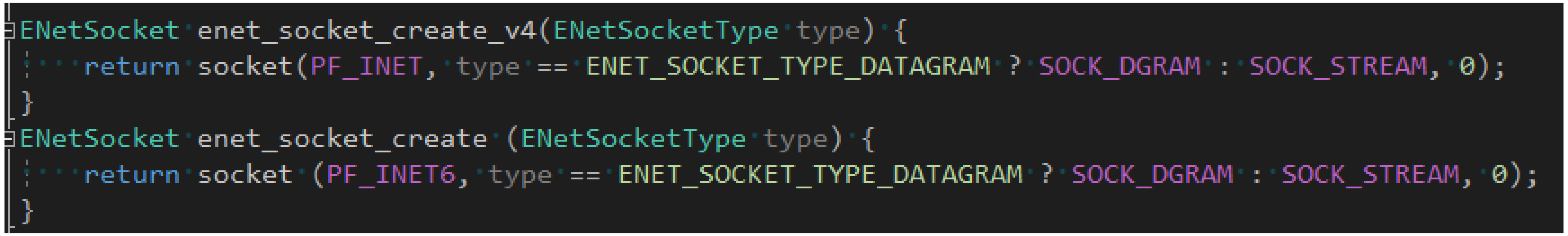
```cpp
bool EnetConnection::connectImpl(uint32_t timeout) {
    assert(host_ == nullptr);
    assert(peer_ == nullptr);
    bool result = false;
    ENetAddress address;
    memset(&address, 0, sizeof(address));
    if (remote_address_.isIpV4()) {
        address.is_ipv4 = true;
        address.hostv4 = remote_address_.getAddrIntN();
        address.port = remote_address_.getPort();
    }
    else {
        enet_address_set_host(&address, remote_address_.toIpString());
        address.port = remote_address_.getPort();
    }
    host_ = enet_host_create(nullptr, 1, 1, 0, 0, address.is_ipv4);
    PM_LOG_FAILED_JUMP(host_);
    peer_ = enet_host_connect(host_, &address, 1, 0);
    PM_LOG_FAILED_JUMP(peer_);
    enet_peer_timeout(peer_, timeout_limit_, timeout_minimum_, timeout);
    loop_task_id_ = ctx_->addLoopTask([this]() -> bool {
        if (host_) {
            innerLoop();
            return true;
        }
        return false;
    });
    result = true;
Exit0:
    if (!result) {
        if (host_) {
            enet_host_destroy(host_);
            host_ = nullptr;
        }
    }
    return result;
}
```

ENet IPv6处理

Enet兼容IPv4和IPv6

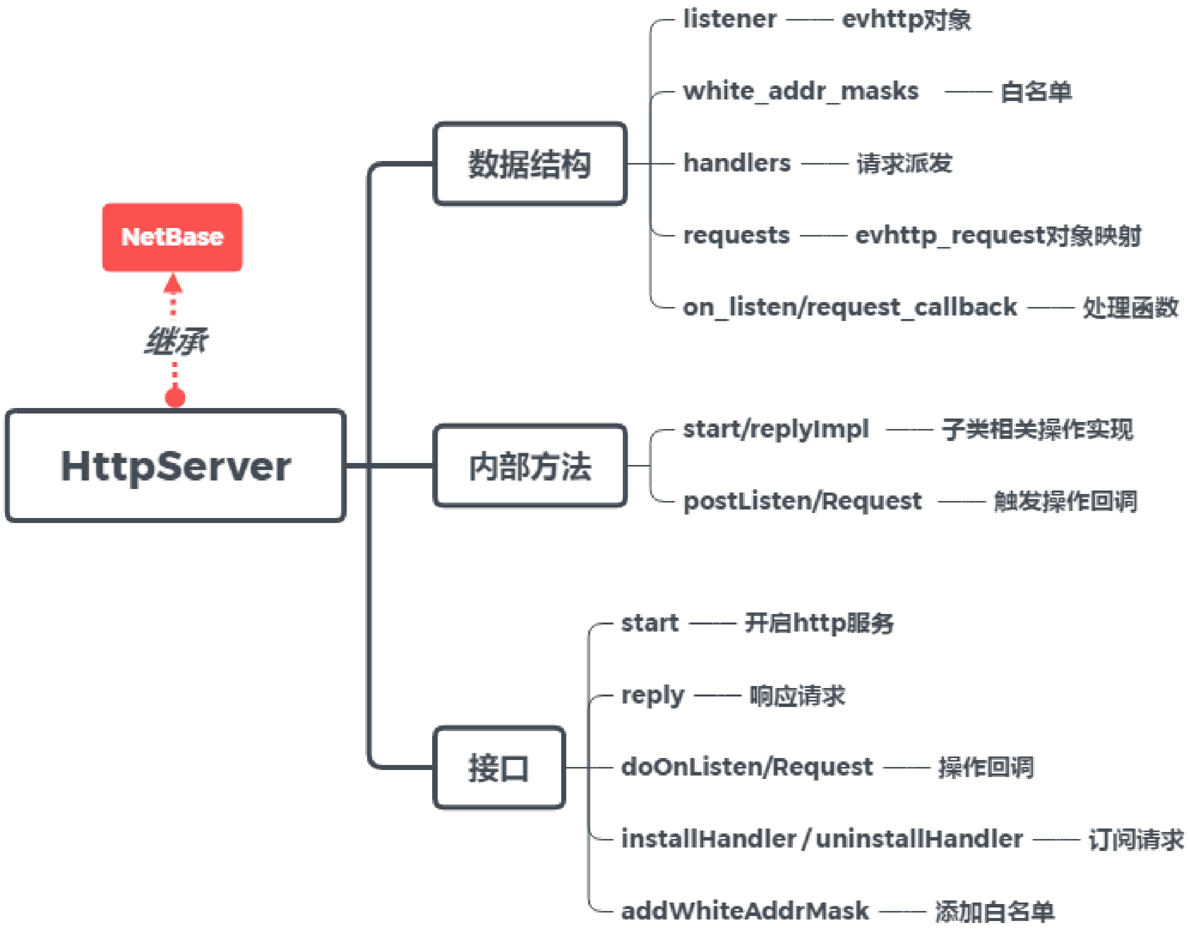


```
typedef struct _ENetAddress
{
    union {
        struct in6_addr host;
        enet_uint32 hostv4;
    };
    enet_uint16 port;
    enet_uint16 sin6_scope_id;
    enet_uint8 is_ipv4;
} ENetAddress;
```

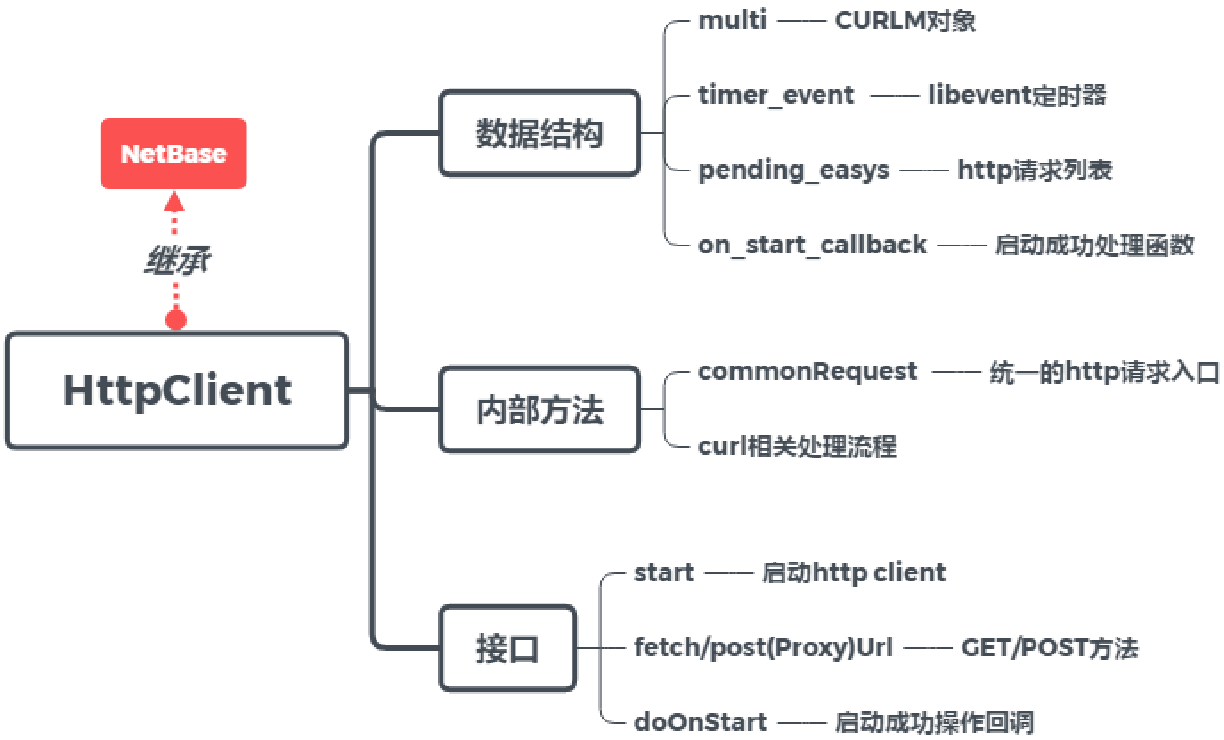


```
ENetSocket enet_socket_create_v4(ENetSocketType type) {
    return socket(PF_INET, type == ENET_SOCKET_TYPE_DATAGRAM ? SOCK_DGRAM : SOCK_STREAM, 0);
}
ENetSocket enet_socket_create (ENetSocketType type) {
    return socket (PF_INET6, type == ENET_SOCKET_TYPE_DATAGRAM ? SOCK_DGRAM : SOCK_STREAM, 0);
}
```

继承模型



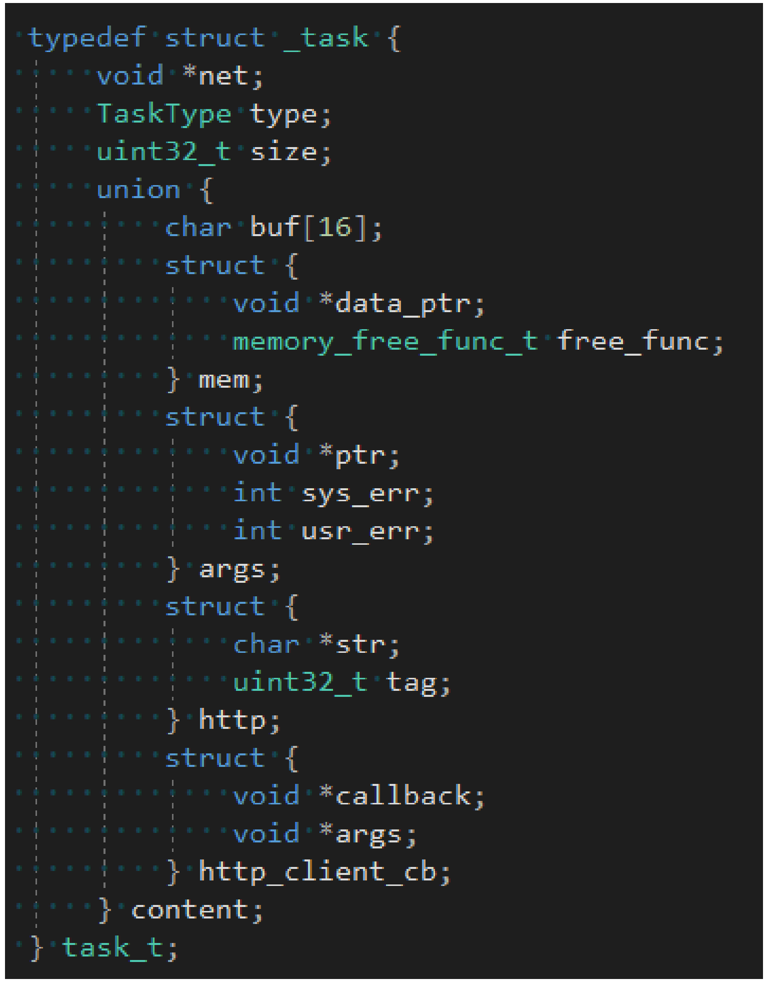
- NetBase
  - 继承
- HttpServer
  - 数据结构
    - listener —— evhttp对象
    - white_addr_masks —— 白名单
    - handlers —— 请求派发
    - requests —— evhttp_request对象映射
    - on_listen/request_callback —— 处理函数
  - 内部方法
    - start/replyImpl —— 子类相关操作实现
    - postListen/Request —— 触发操作回调
  - 接口
    - start —— 开启http服务
    - reply —— 响应请求
    - doOnListen/Request —— 操作回调
    - installHandler / uninstallHandler —— 订阅请求
    - addWhiteAddrMask —— 添加白名单



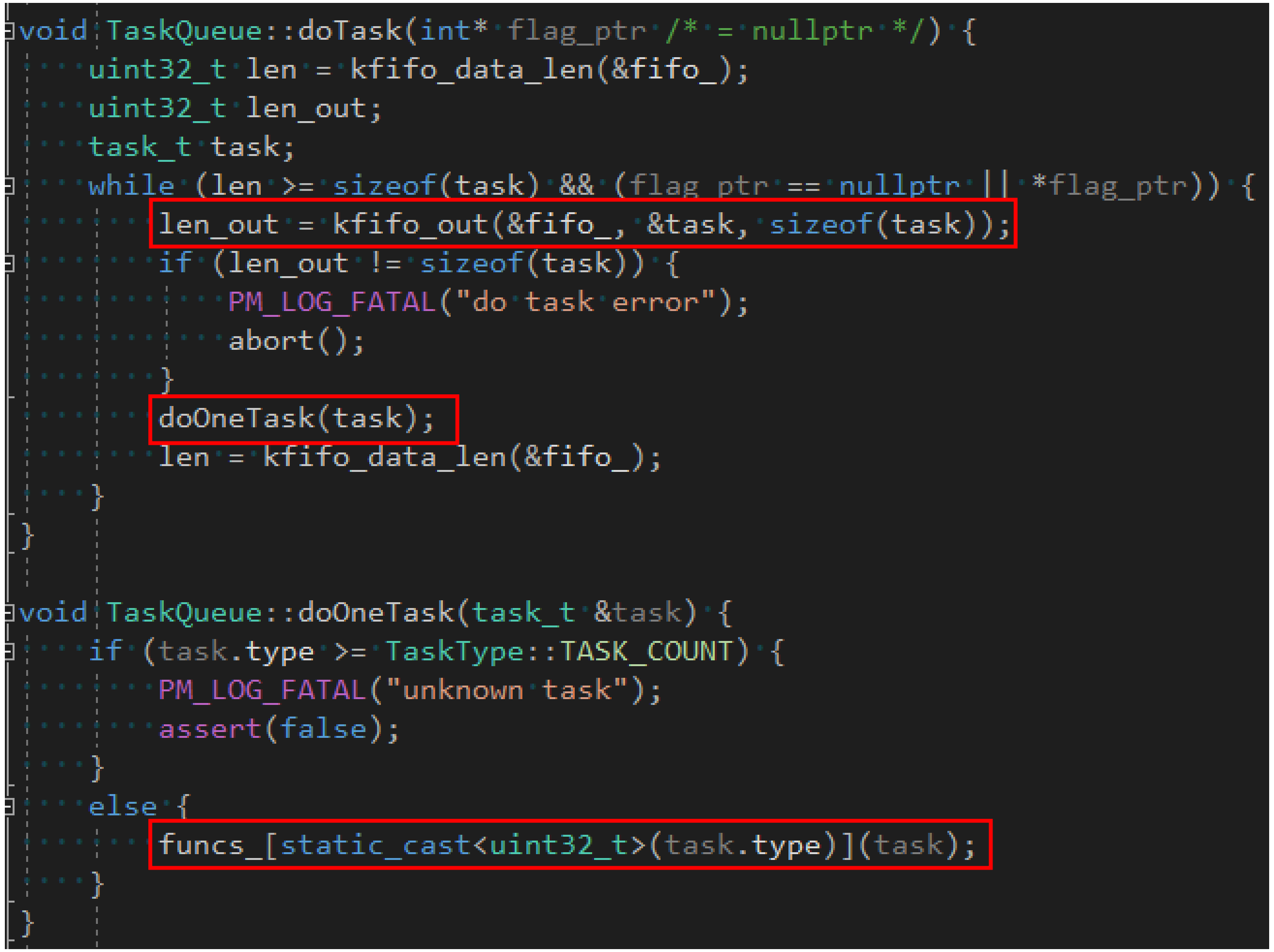
- NetBase
  - 继承
- HttpClient
  - 数据结构
    - multi —— CURLM对象
    - timer_event —— libevent定时器
    - pending_easys —— http请求列表
    - on_start_callback —— 启动成功处理函数
  - 内部方法
    - commonRequest —— 统一的http请求入口
    - curl相关处理流程
  - 接口
    - start —— 启动http client
    - fetch/post(Proxy)Url —— GET/POST方法
    - doOnStart —— 启动成功操作回调

任务队列

task_t数据结构

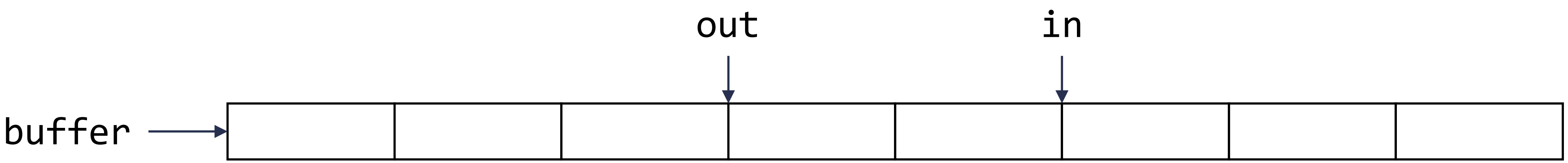


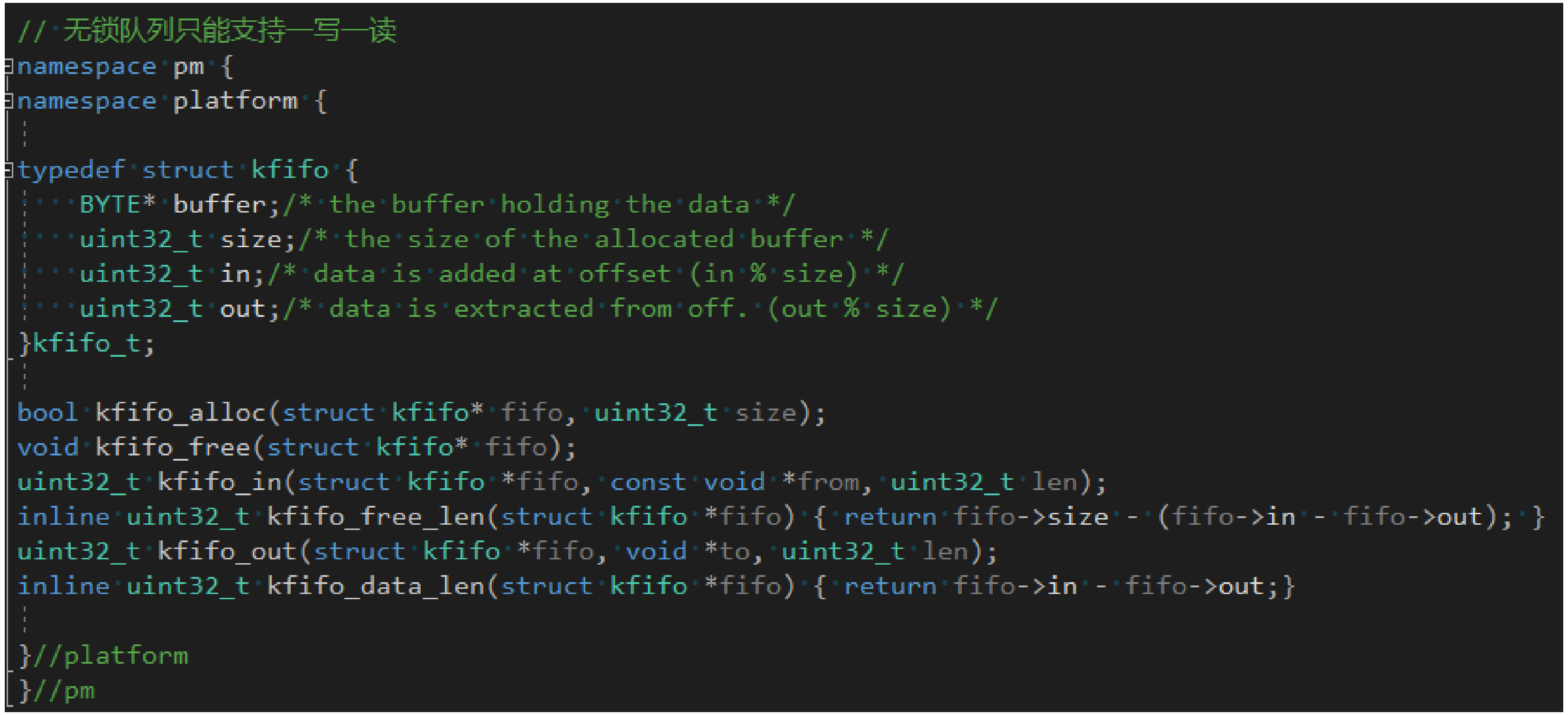
```
typedef struct _task {
    void *net;
    TaskType type;
    uint32_t size;
    union {
        char buf[16];
        struct {
            void *data_ptr;
            memory_free_func_t free_func;
        } mem;
        struct {
            void *ptr;
            int sys_err;
            int usr_err;
        } args;
        struct {
            char *str;
            uint32_t tag;
        } http;
        struct {
            void *callback;
            void *args;
        } http_client_cb;
    } content;
} task_t;
```



```
void TaskQueue::doTask(int* flag_ptr /* = nullptr */) {
    uint32_t len = kfifo_data_len(&fifo_);
    uint32_t len_out;
    task_t task;
    while (len >= sizeof(task) && (flag_ptr == nullptr || *flag_ptr)) {
        len_out = kfifo_out(&fifo_, &task, sizeof(task));
        if (len_out != sizeof(task)) {
            PM_LOG_FATAL("do task error");
            abort();
        }
        doOneTask(task);
        len = kfifo_data_len(&fifo_);
    }
}

void TaskQueue::doOneTask(task_t &task) {
    if (task.type >= TaskType::TASK_COUNT) {
        PM_LOG_FATAL("unknown task");
        assert(false);
    }
    else {
        funcs_[static_cast<uint32_t>(task.type)](task);
    }
}
```

任务队列

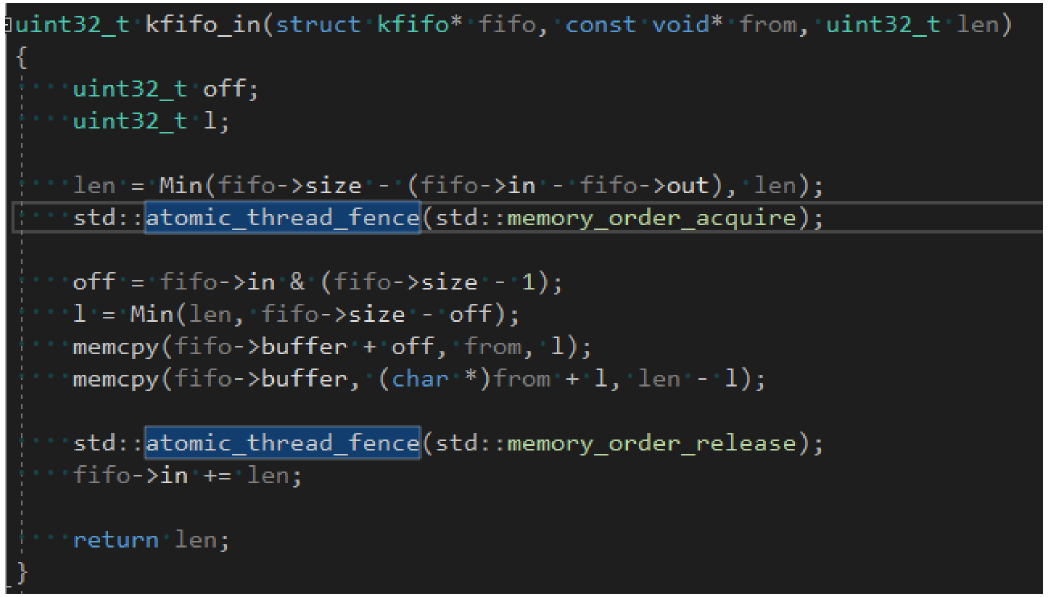




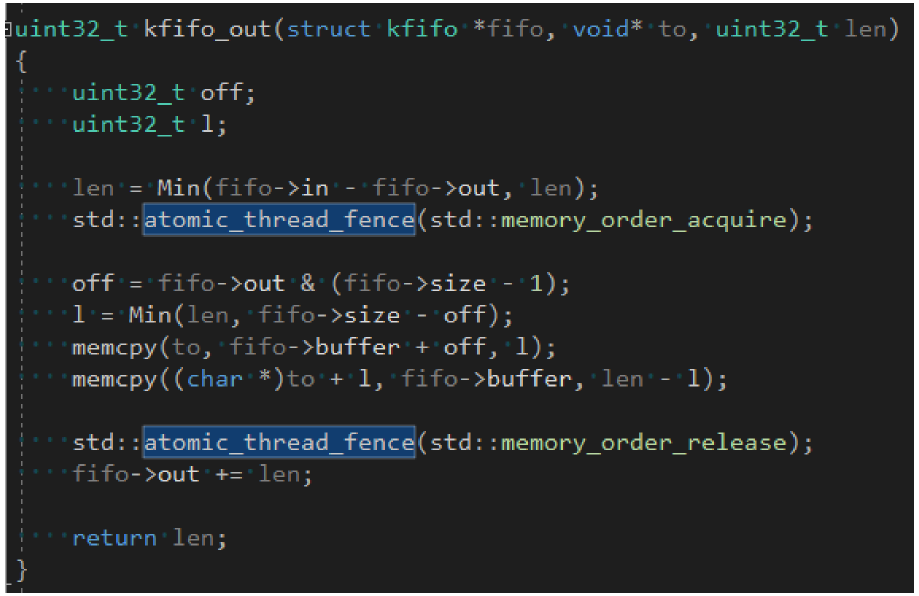
```
// 无锁队列只能支持一写一读
namespace pm {
namespace platform {

typedef struct kfifo {
    BYTE* buffer;/* the buffer holding the data */
    uint32_t size;/* the size of the allocated buffer */
    uint32_t in;/* data is added at offset (in % size) */
    uint32_t out;/* data is extracted from off. (out % size) */
}kfifo_t;

bool kfifo_alloc(struct kfifo* fifo, uint32_t size);
void kfifo_free(struct kfifo* fifo);
uint32_t kfifo_in(struct kfifo *fifo, const void *from, uint32_t len);
inline uint32_t kfifo_free_len(struct kfifo *fifo) { return fifo->size - (fifo->in - fifo->out); }
uint32_t kfifo_out(struct kfifo *fifo, void *to, uint32_t len);
inline uint32_t kfifo_data_len(struct kfifo *fifo) { return fifo->in - fifo->out;}

}//platform
}//pm
```

任务队列

无锁队列，移植自 Linux 内核的无锁队列实现，单生产者单消费者场景



```
uint32_t kfifo_in(struct kfifo* fifo, const void* from, uint32_t len)
{
    uint32_t off;
    uint32_t l;

    len = Min(fifo->size - (fifo->in - fifo->out), len);
    std::atomic_thread_fence(std::memory_order_acquire);

    off = fifo->in & (fifo->size - 1);
    l = Min(len, fifo->size - off);
    memcpy(fifo->buffer + off, from, l);
    memcpy(fifo->buffer, (char *)from + l, len - l);

    std::atomic_thread_fence(std::memory_order_release);
    fifo->in += len;

    return len;
}
```



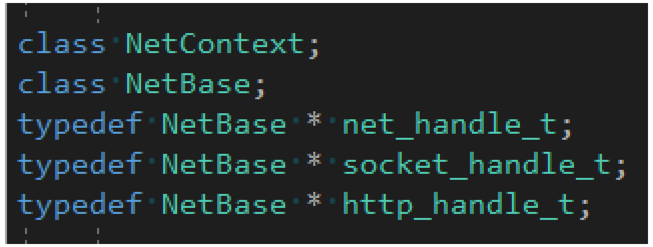
```
uint32_t kfifo_out(struct kfifo *fifo, void* to, uint32_t len)
{
    uint32_t off;
    uint32_t l;

    len = Min(fifo->in - fifo->out, len);
    std::atomic_thread_fence(std::memory_order_acquire);

    off = fifo->out & (fifo->size - 1);
    l = Min(len, fifo->size - off);
    memcpy(to, fifo->buffer + off, l);
    memcpy((char *)to + l, fifo->buffer, len - l);

    std::atomic_thread_fence(std::memory_order_release);
    fifo->out += len;

    return len;
}
```

atomic_thread_fence作用？

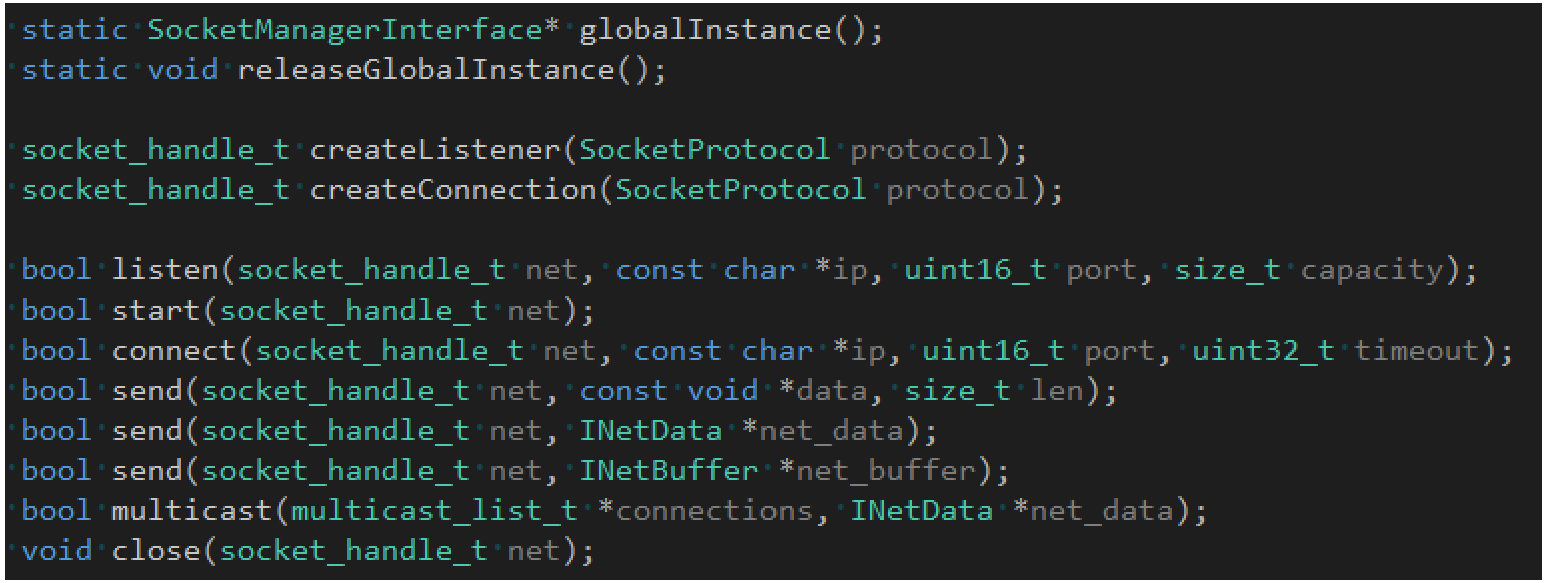
队列如何循环？为什么in-out即大小？

接口设计

C风格接口，外部持有指针，统一通过接口操作，隐藏跨线程交互细节

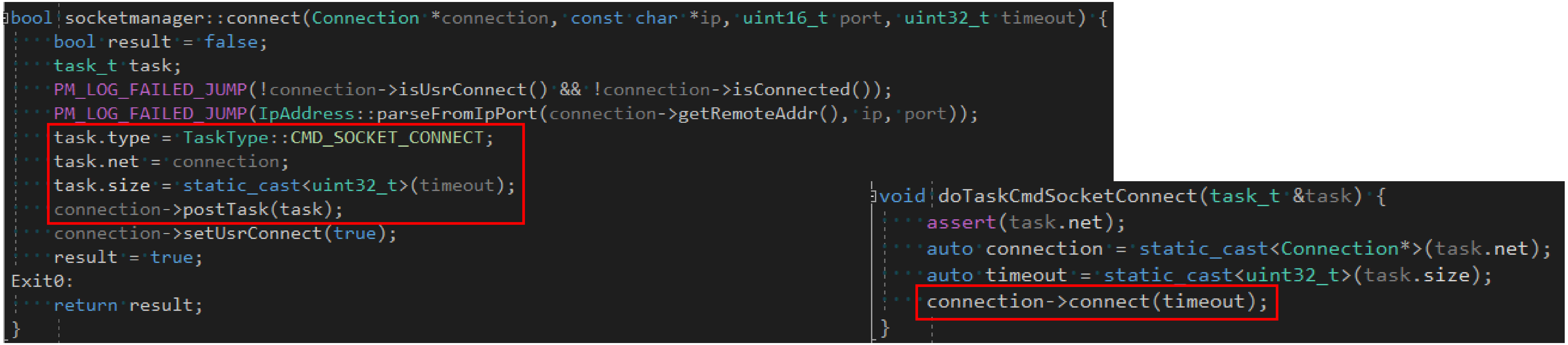


```
class NetContext;
class NetBase;
typedef NetBase * net_handle_t;
typedef NetBase * socket_handle_t;
typedef NetBase * http_handle_t;
```



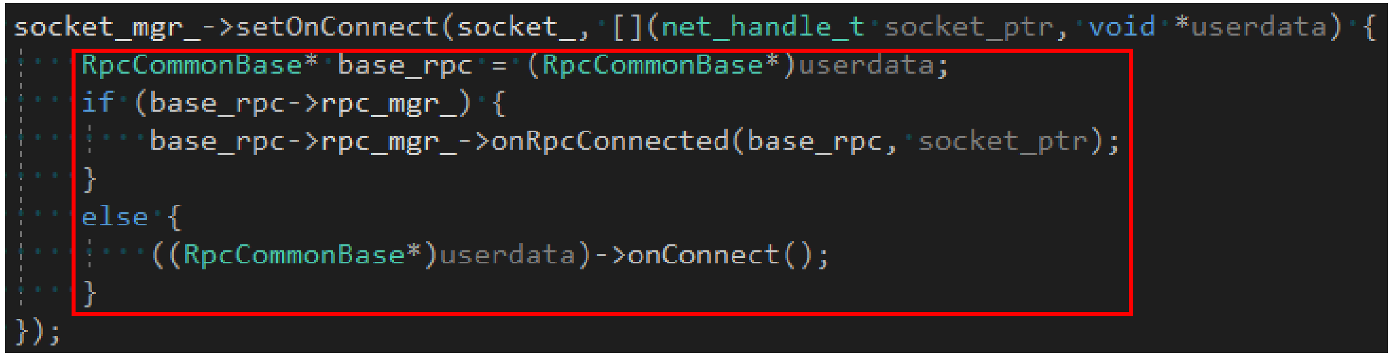
```
static SocketManagerInterface* globalInstance();
static void releaseGlobalInstance();

socket_handle_t createListener(SocketProtocol protocol);
socket_handle_t createConnection(SocketProtocol protocol);

bool listen(socket_handle_t net, const char *ip, uint16_t port, size_t capacity);
bool start(socket_handle_t net);
bool connect(socket_handle_t net, const char *ip, uint16_t port, uint32_t timeout);
bool send(socket_handle_t net, const void *data, size_t len);
bool send(socket_handle_t net, INetData *net_data);
bool send(socket_handle_t net, INetBuffer *net_buffer);
bool multicast(multicast_list_t *connections, INetData *net_data);
void close(socket_handle_t net);
```

线程隔离

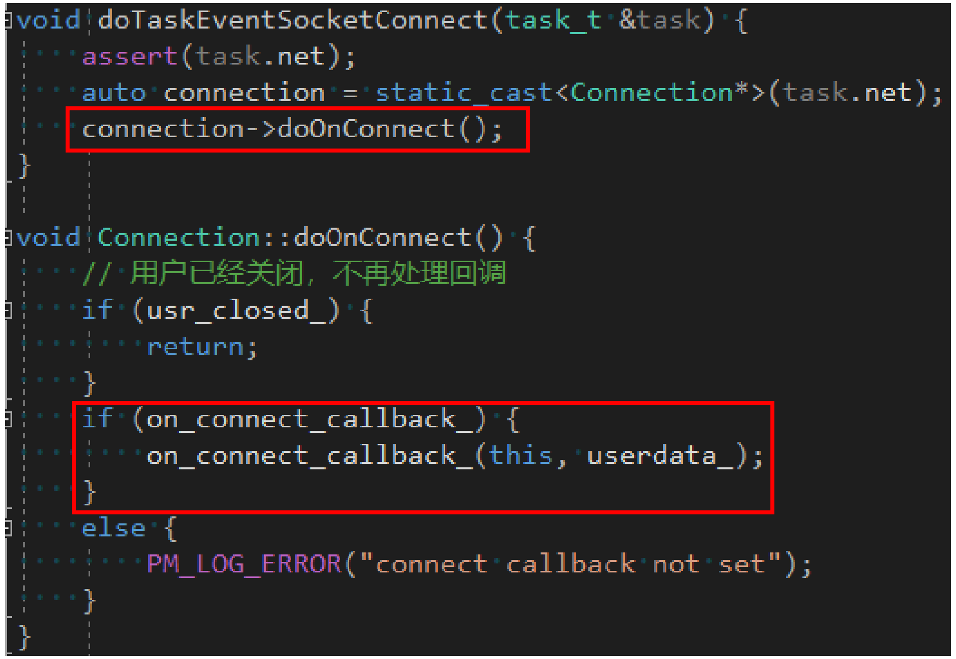


```
bool socketmanager::connect(Connection *connection, const char *ip, uint16_t port, uint32_t timeout) {
    bool result = false;
    task_t task;
    PM_LOG_FAILED_JUMP(!connection->isUsrConnect() && !connection->isConnected());
    PM_LOG_FAILED_JUMP(IpAddress::parseFromIpPort(connection->getRemoteAddr(), ip, port));
    task.type = TaskType::CMD_SOCKET_CONNECT;
    task.net = connection;
    task.size = static_cast<uint32_t>(timeout);
    connection->postTask(task);
    connection->setUsrConnect(true);
    result = true;
Exit0:
    return result;
}
```

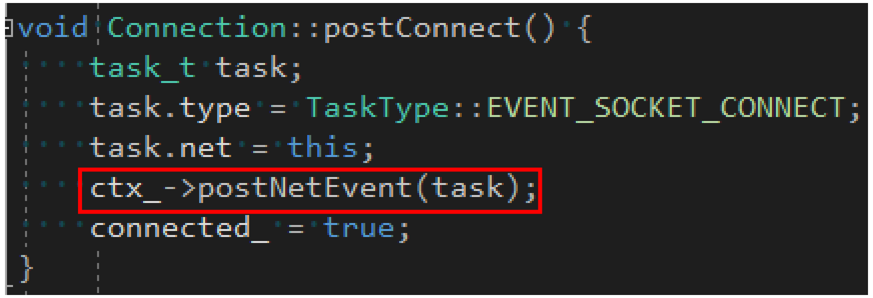
```
void doTaskCmdSocketConnect(task_t &task) {
    assert(task.net);
    auto connection = static_cast<Connection*>(task.net);
    auto timeout = static_cast<uint32_t>(task.size);
    connection->connect(timeout);
}
```



```
socket_mgr_->setOnConnect(socket_, [](net_handle_t socket_ptr, void *userdata) {
    RpcCommonBase* base_rpc = (RpcCommonBase*)userdata;
    if (base_rpc->rpc_mgr_) {
        base_rpc->rpc_mgr_->onRpcConnected(base_rpc, socket_ptr);
    }
    else {
        ((RpcCommonBase*)userdata)->onConnect();
    }
});
```



```
void doTaskEventSocketConnect(task_t &task) {
    assert(task.net);
    auto connection = static_cast<Connection*>(task.net);
    connection->doOnConnect();
}

void Connection::doOnConnect() {
    // 用户已经关闭，不再处理回调
    if (usr_closed_) {
        return;
    }
    if (on_connect_callback_) {
        on_connect_callback_(this, userdata_);
    }
    else {
        PM_LOG_ERROR("connect callback not set");
    }
}
```

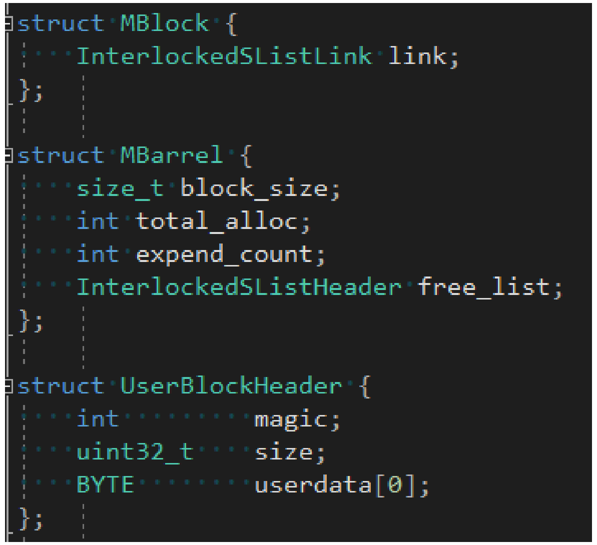


```
void Connection::postConnect() {
    task_t task;
    task.type = TaskType::EVENT_SOCKET_CONNECT;
    task.net = this;
    ctx_->postNetEvent(task);
    connected_ = true;
}
```

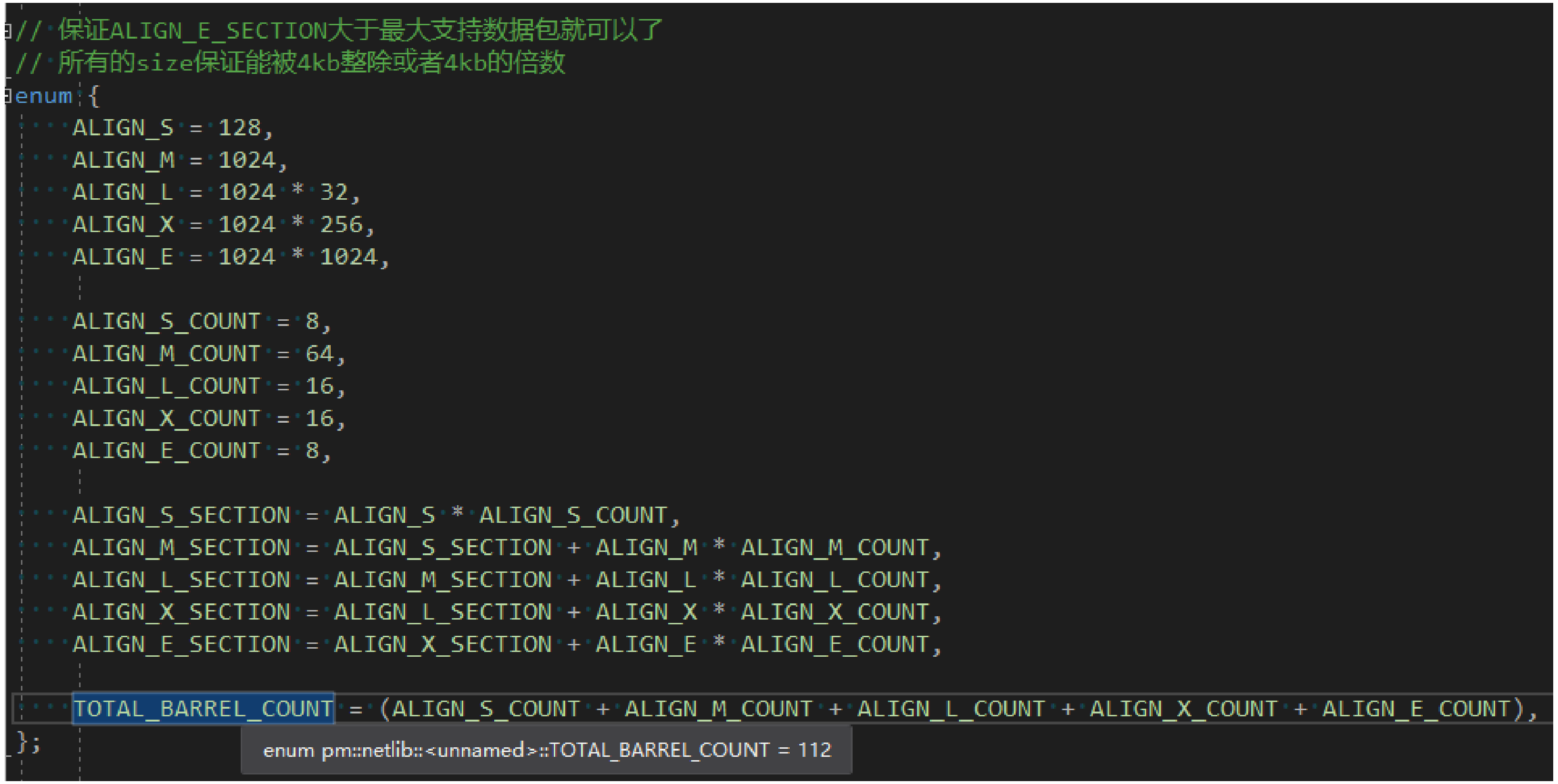
内存管理

网络线程内存特点：
1. 使用时间短
2. 跨线程：一个线程申请另一个线程释放

设计：
1. 分桶管理
2. 动态扩容
3. 无锁（CAS操作）



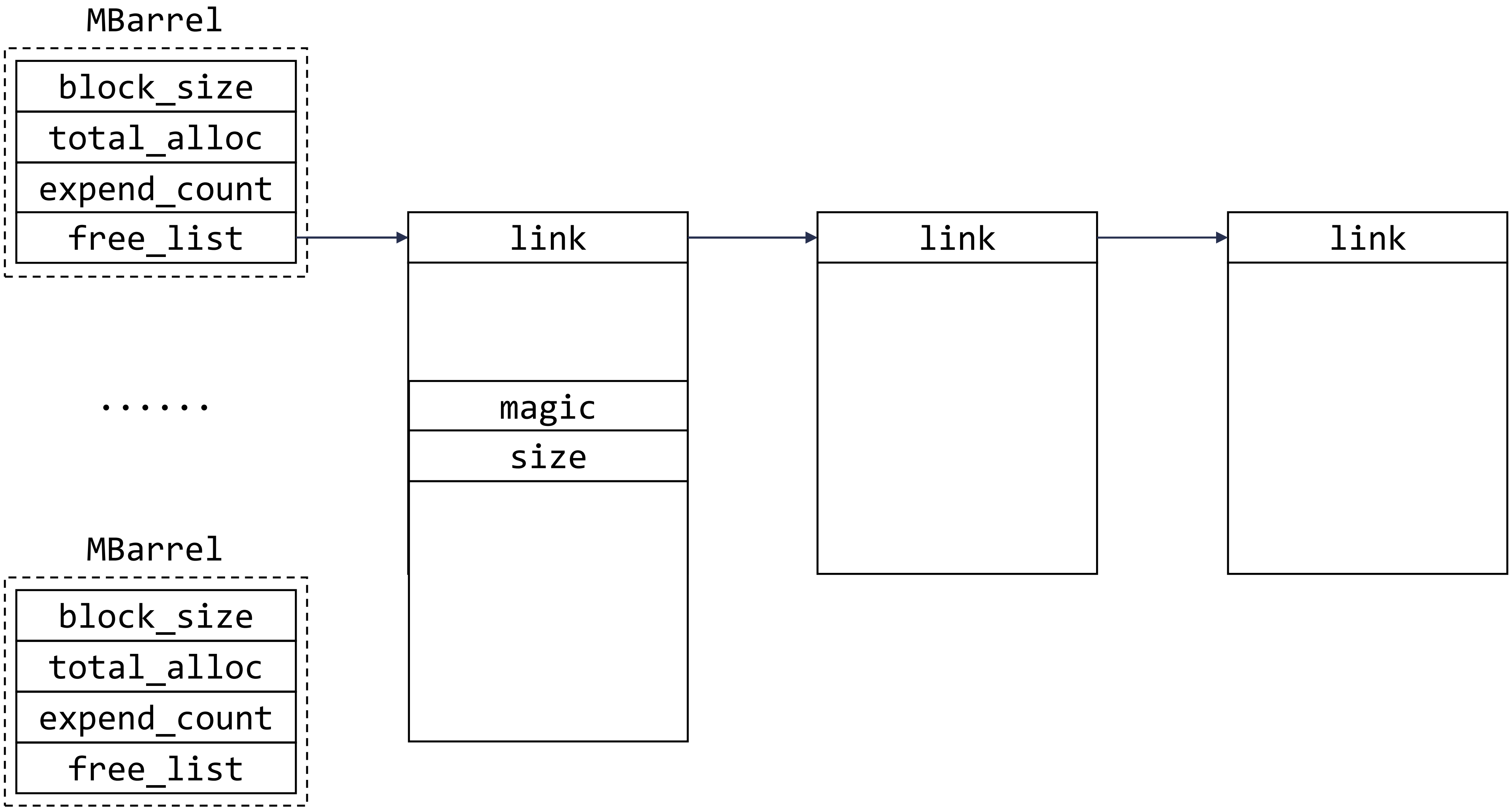
```
struct MBlock {
    InterlockedSListLink link;
};

struct MBarrel {
    size_t block_size;
    int total_alloc;
    int expend_count;
    InterlockedSListHeader free_list;
};

struct UserBlockHeader {
    int         magic;
    uint32_t    size;
    BYTE        userdata[0];
};
```



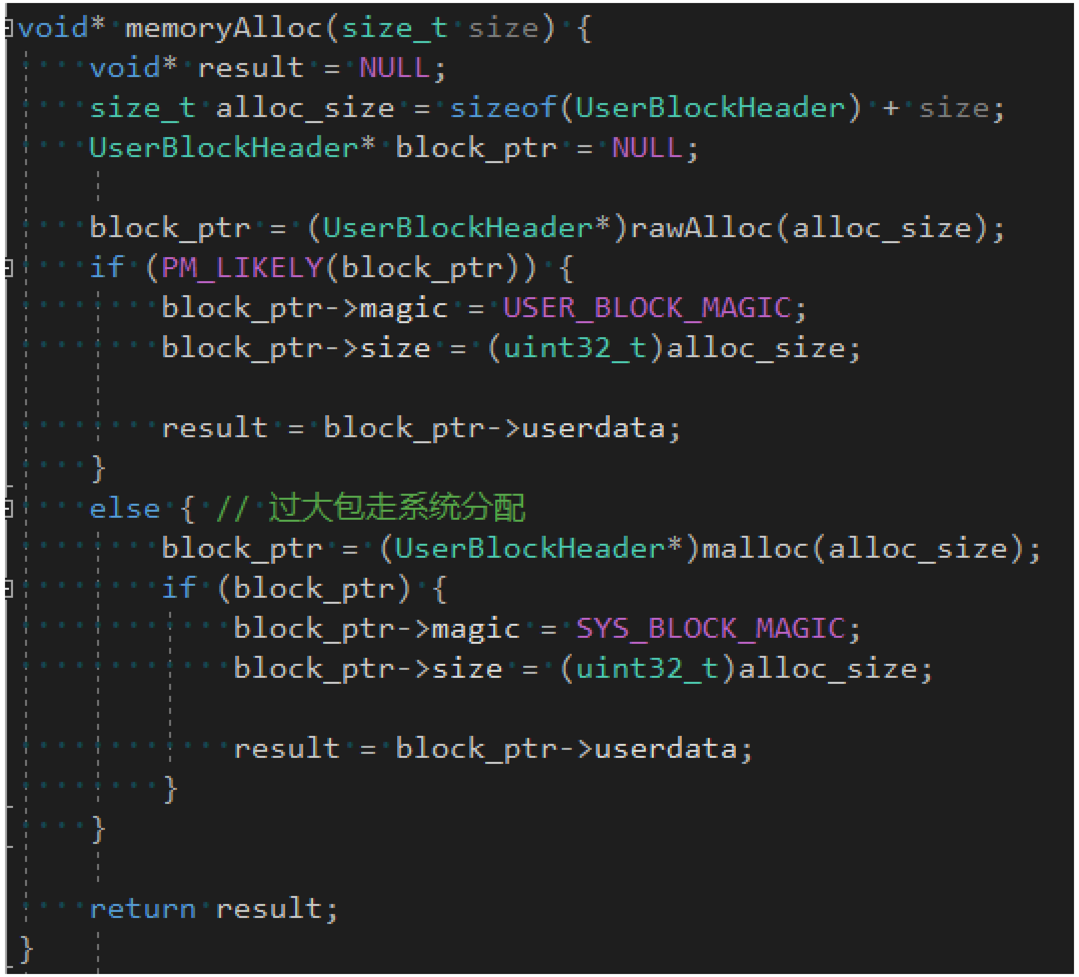
```
// 保证ALIGN_E_SECTION大于最大支持数据包就可以了
// 所有的size保证能被4kb整除或者4kb的倍数
enum {
    ALIGN_S = 128,
    ALIGN_M = 1024,
    ALIGN_L = 1024 * 32,
    ALIGN_X = 1024 * 256,
    ALIGN_E = 1024 * 1024,

    ALIGN_S_COUNT = 8,
    ALIGN_M_COUNT = 64,
    ALIGN_L_COUNT = 16,
    ALIGN_X_COUNT = 16,
    ALIGN_E_COUNT = 8,

    ALIGN_S_SECTION = ALIGN_S * ALIGN_S_COUNT,
    ALIGN_M_SECTION = ALIGN_S_SECTION + ALIGN_M * ALIGN_M_COUNT,
    ALIGN_L_SECTION = ALIGN_M_SECTION + ALIGN_L * ALIGN_L_COUNT,
    ALIGN_X_SECTION = ALIGN_L_SECTION + ALIGN_X * ALIGN_X_COUNT,
    ALIGN_E_SECTION = ALIGN_X_SECTION + ALIGN_E * ALIGN_E_COUNT,

    TOTAL_BARREL_COUNT = (ALIGN_S_COUNT + ALIGN_M_COUNT + ALIGN_L_COUNT + ALIGN_X_COUNT + ALIGN_E_COUNT),
};
```

enum pm::netlib::<unnamed>::TOTAL_BARREL_COUNT = 112

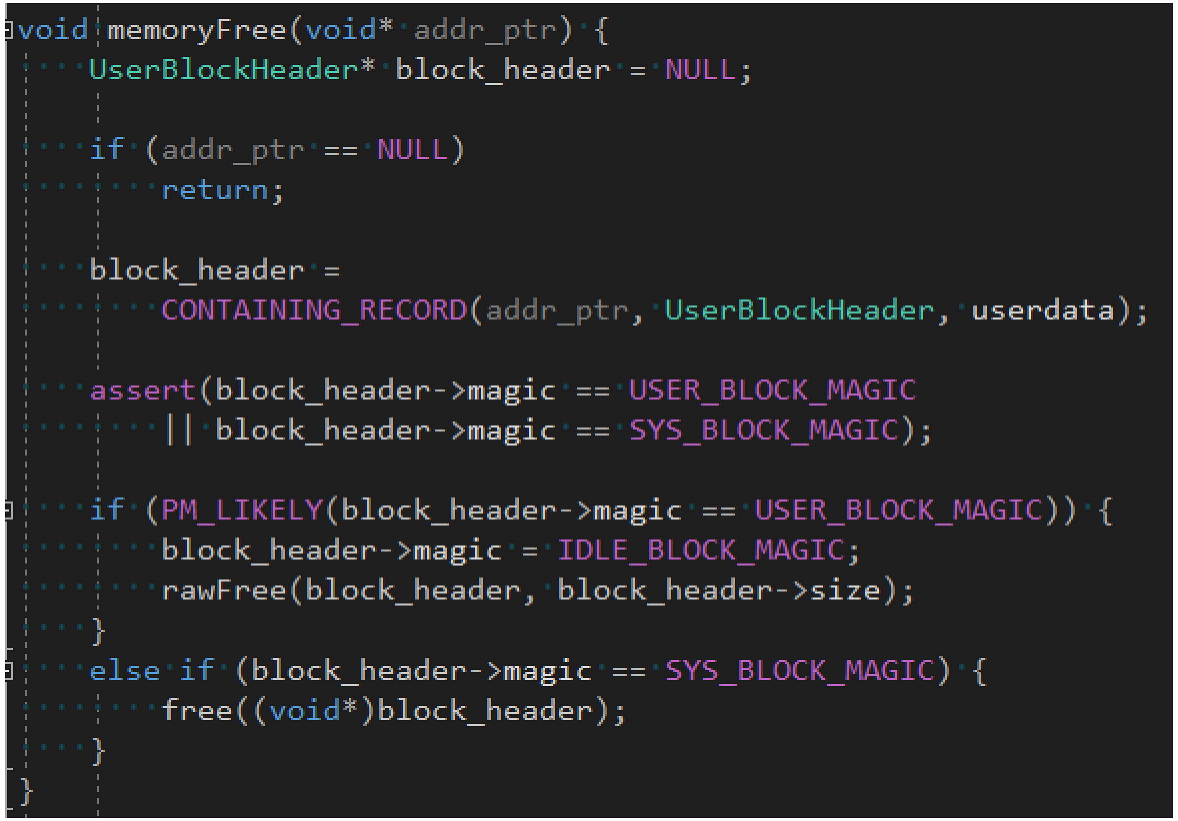
分桶管理


MBarrel
block_size
total_alloc
expend_count
free_list
......
MBarrel
block_size
total_alloc
expend_count
free_list
link
magic
size
link
link

分桶管理

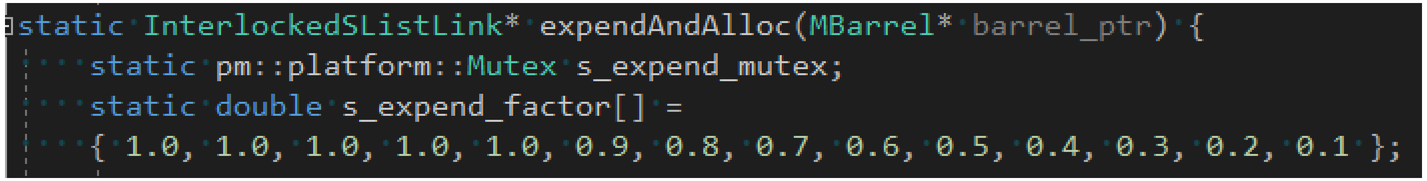


```
void* memoryAlloc(size_t size) {
    void* result = NULL;
    size_t alloc_size = sizeof(UserBlockHeader) + size;
    UserBlockHeader* block_ptr = NULL;

    block_ptr = (UserBlockHeader*)rawAlloc(alloc_size);
    if (PM_LIKELY(block_ptr)) {
        block_ptr->magic = USER_BLOCK_MAGIC;
        block_ptr->size = (uint32_t)alloc_size;

        result = block_ptr->userdata;
    }
    else { // 过大包走系统分配
        block_ptr = (UserBlockHeader*)malloc(alloc_size);
        if (block_ptr) {
            block_ptr->magic = SYS_BLOCK_MAGIC;
            block_ptr->size = (uint32_t)alloc_size;

            result = block_ptr->userdata;
        }
    }

    return result;
}
```

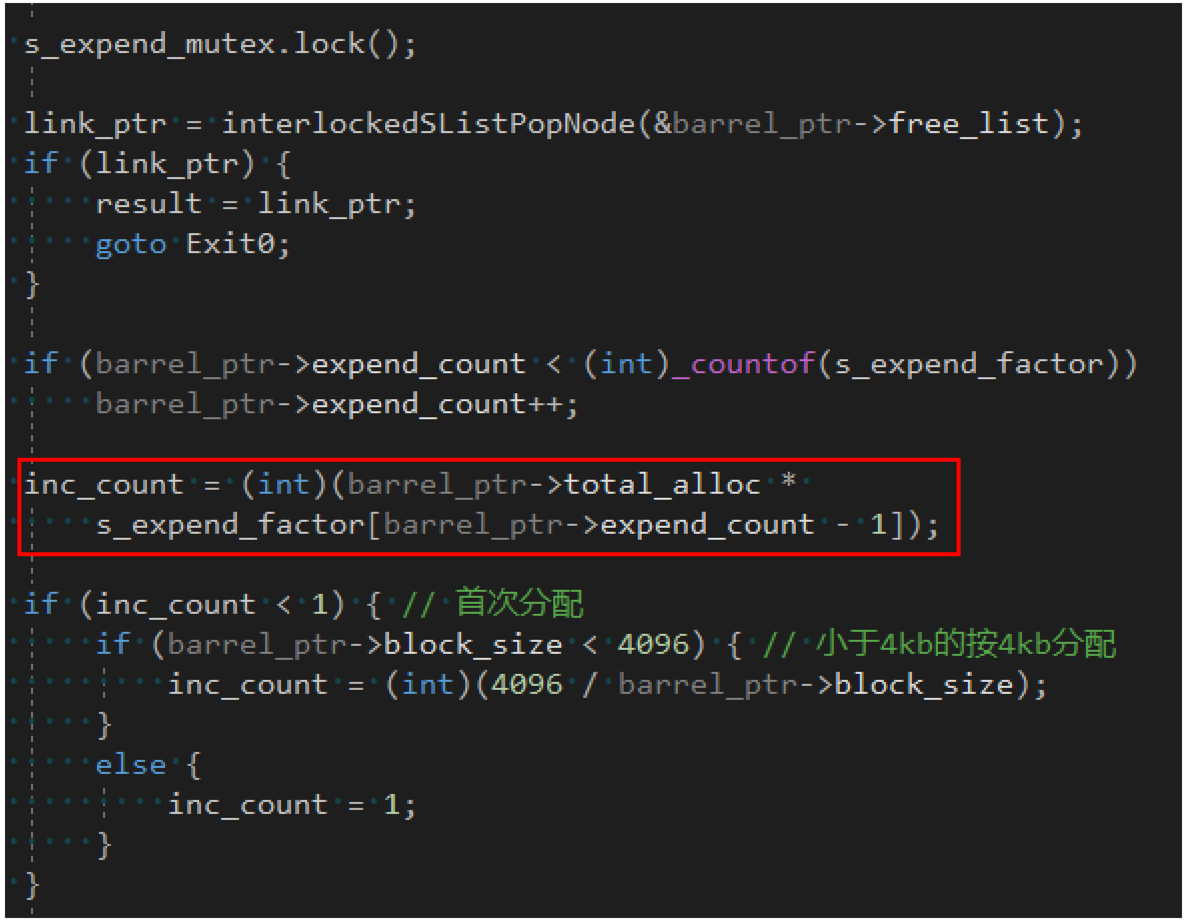


```
void memoryFree(void* addr_ptr) {
    UserBlockHeader* block_header = NULL;

    if (addr_ptr == NULL)
        return;

    block_header =
        CONTAINING_RECORD(addr_ptr, UserBlockHeader, userdata);

    assert(block_header->magic == USER_BLOCK_MAGIC
        || block_header->magic == SYS_BLOCK_MAGIC);

    if (PM_LIKELY(block_header->magic == USER_BLOCK_MAGIC)) {
        block_header->magic = IDLE_BLOCK_MAGIC;
        rawFree(block_header, block_header->size);
    }
    else if (block_header->magic == SYS_BLOCK_MAGIC) {
        free((void*)block_header);
    }
}
```

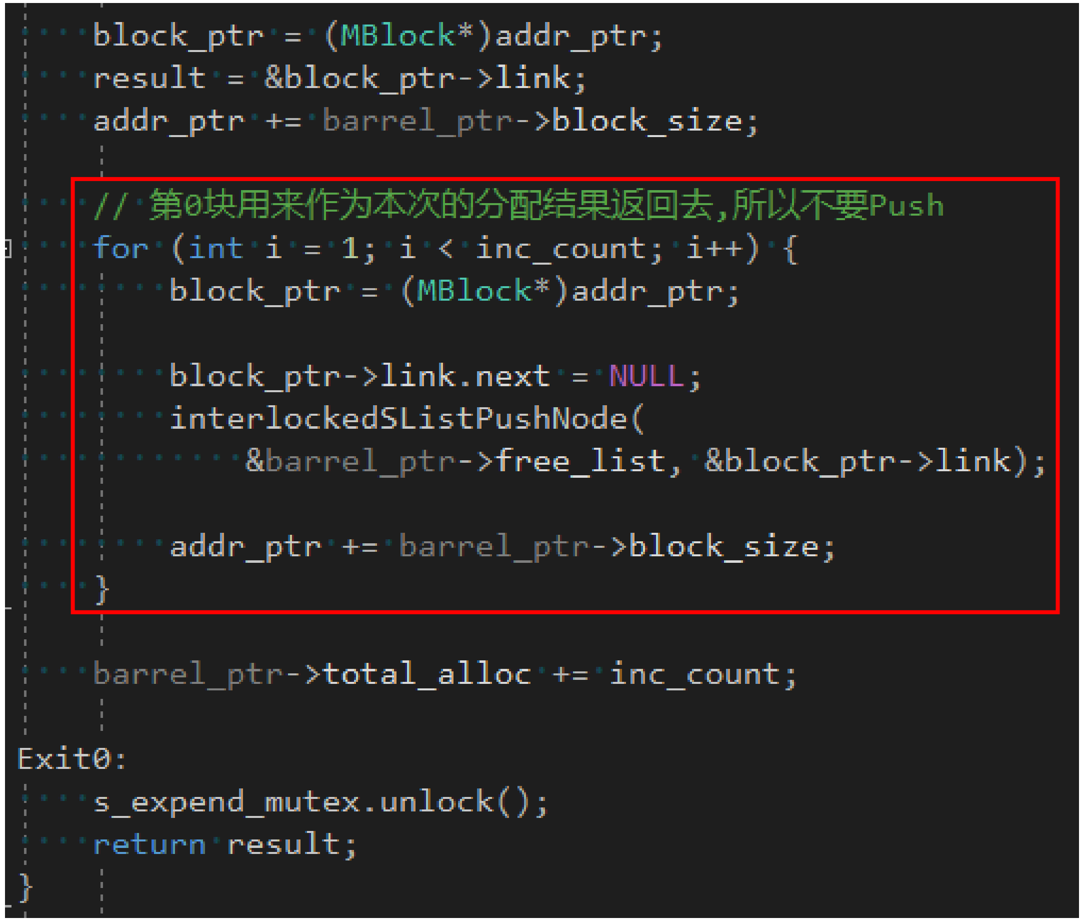
动态扩容



```
static InterlockedSListLink* expendAndAlloc(MBarrel* barrel_ptr) {
    static pm::platform::Mutex s_expend_mutex;
    static double s_expend_factor[] =
    { 1.0, 1.0, 1.0, 1.0, 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 };
```

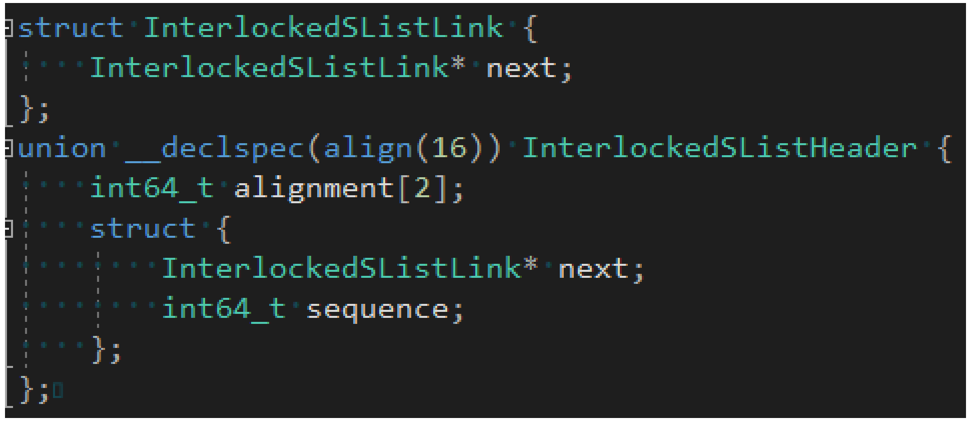


```
s_expend_mutex.lock();

link_ptr = interlockedSListPopNode(&barrel_ptr->free_list);
if (link_ptr) {
    result = link_ptr;
    goto Exit0;
}

if (barrel_ptr->expend_count < (int)_countof(s_expend_factor))
    barrel_ptr->expend_count++;

inc_count = (int)(barrel_ptr->total_alloc *
    s_expend_factor[barrel_ptr->expend_count - 1]);

if (inc_count < 1) { // 首次分配
    if (barrel_ptr->block_size < 4096) { // 小于4kb的按4kb分配
        inc_count = (int)(4096 / barrel_ptr->block_size);
    }
    else {
        inc_count = 1;
    }
}
```



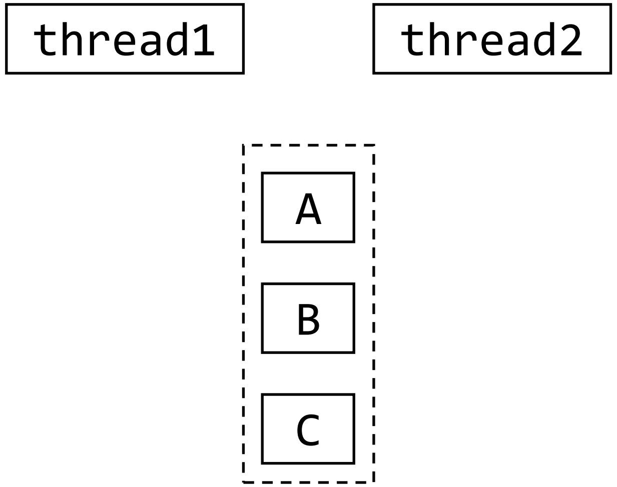
```
    block_ptr = (MBlock*)addr_ptr;
    result = &block_ptr->link;
    addr_ptr += barrel_ptr->block_size;

    // 第0块用来作为本次的分配结果返回去,所以不要Push
    for (int i = 1; i < inc_count; i++) {
        block_ptr = (MBlock*)addr_ptr;

        block_ptr->link.next = NULL;
        interlockedSListPushNode(
            &barrel_ptr->free_list, &block_ptr->link);

        addr_ptr += barrel_ptr->block_size;
    }

    barrel_ptr->total_alloc += inc_count;

Exit0:
    s_expend_mutex.unlock();
    return result;
}
```

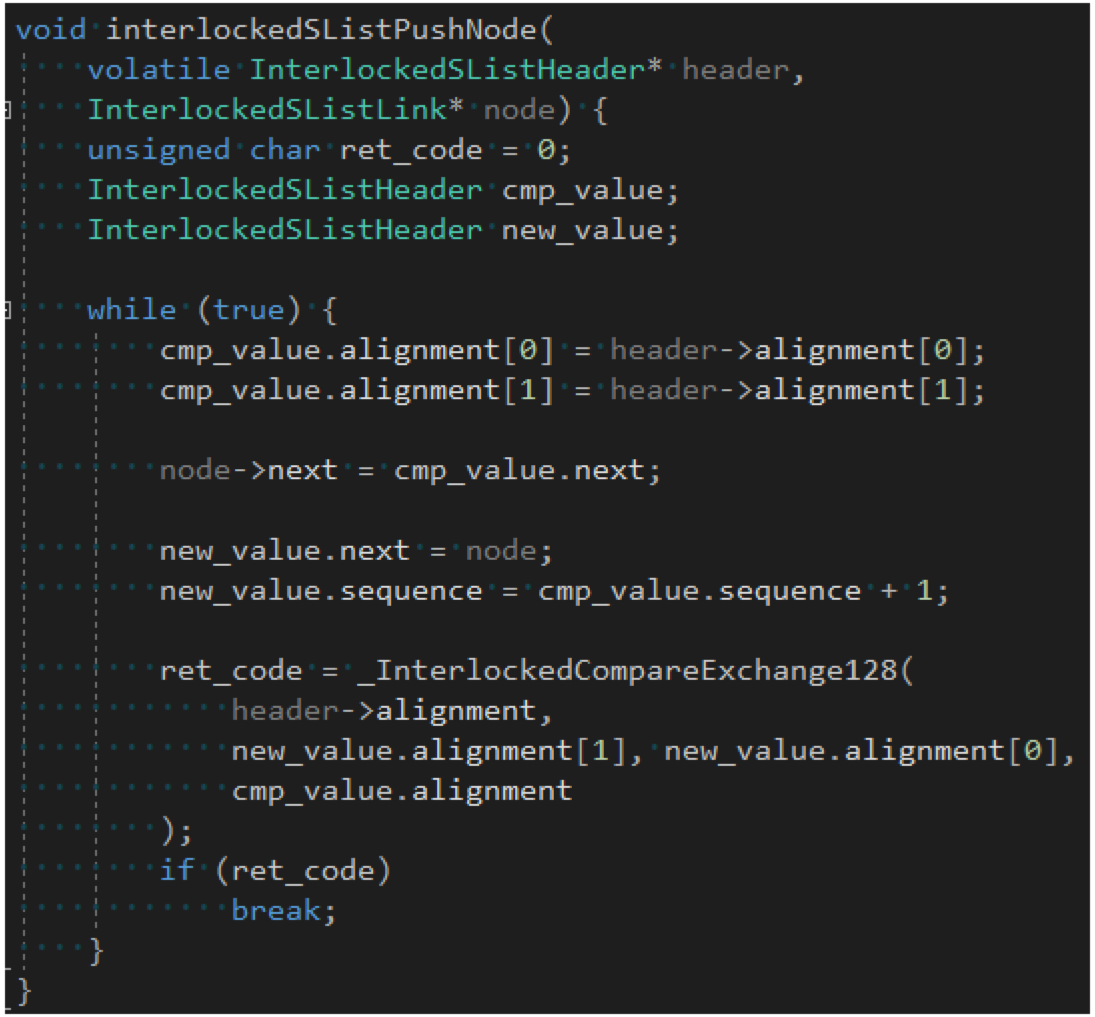
CAS操作



```
struct InterlockedSListLink {
    InterlockedSListLink* next;
};
union __declspec(align(16)) InterlockedSListHeader {
    int64_t alignment[2];
    struct {
        InterlockedSListLink* next;
        int64_t sequence;
    };
};
```

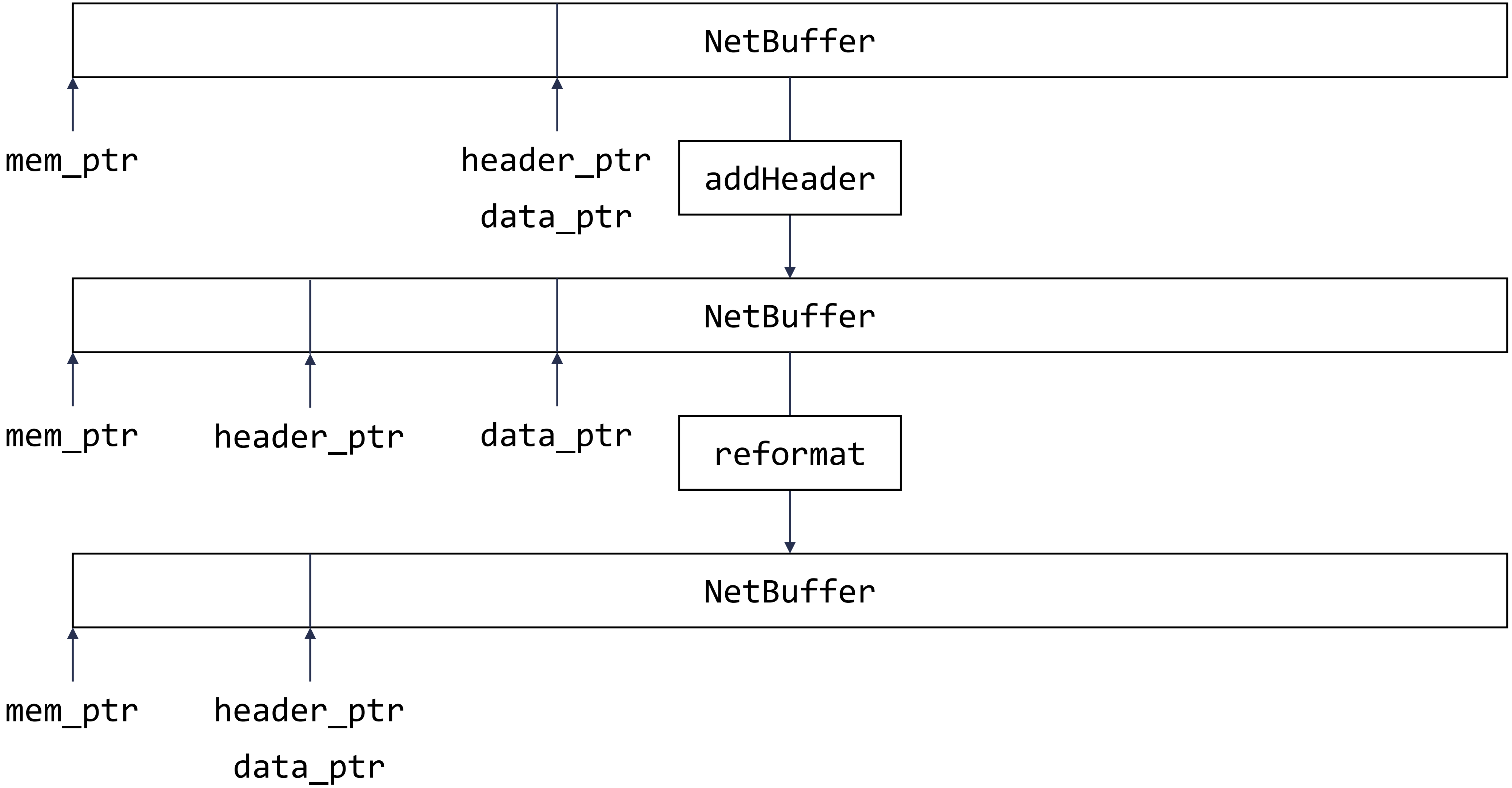
sequence作用?
处理ABA问题



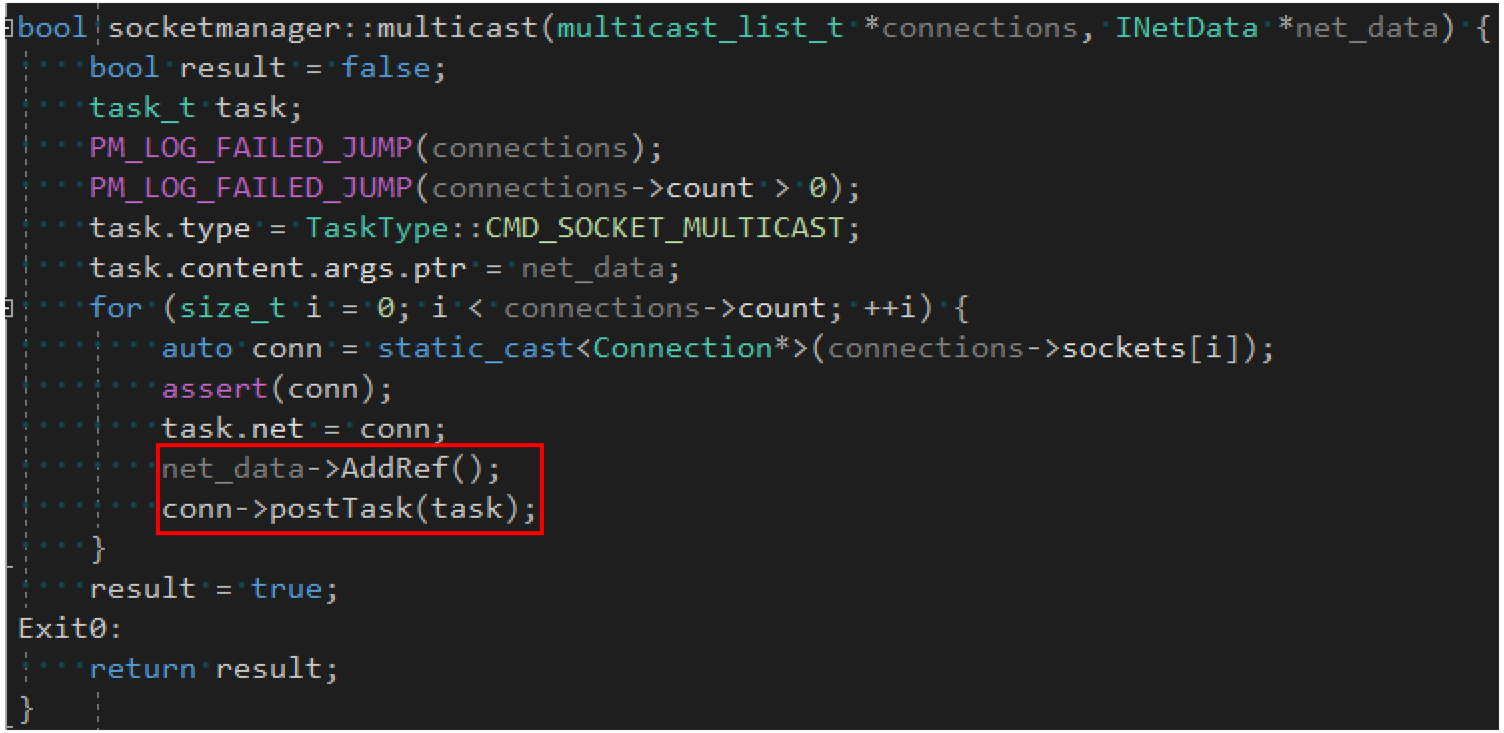


```
void interlockedSListPushNode(
    volatile InterlockedSListHeader* header,
    InterlockedSListLink* node) {
    unsigned char ret_code = 0;
    InterlockedSListHeader cmp_value;
    InterlockedSListHeader new_value;

    while (true) {
        cmp_value.alignment[0] = header->alignment[0];
        cmp_value.alignment[1] = header->alignment[1];

        node->next = cmp_value.next;

        new_value.next = node;
        new_value.sequence = cmp_value.sequence + 1;

        ret_code = _InterlockedCompareExchange128(
            header->alignment,
            new_value.alignment[1], new_value.alignment[0],
            cmp_value.alignment
        );
        if (ret_code)
            break;
    }
}
```

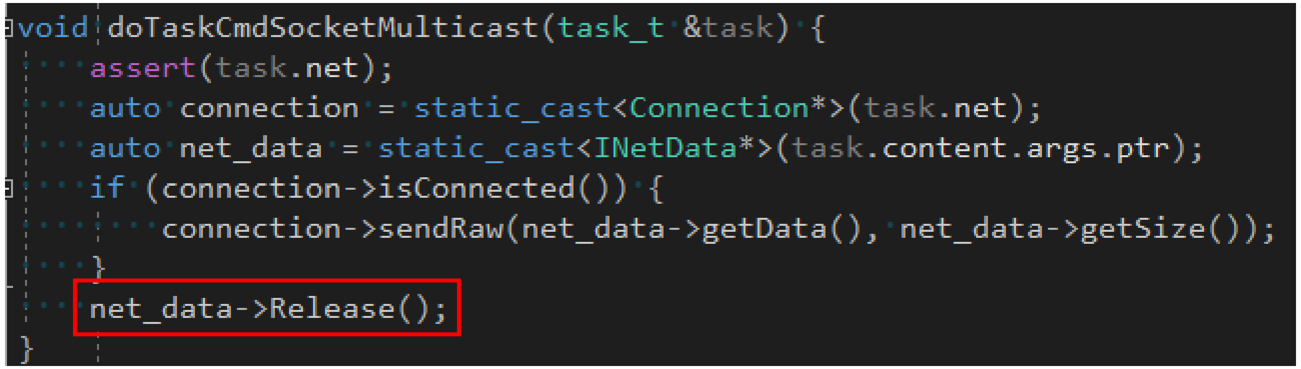
内存管理


NetBuffer
mem_ptr
header_ptr
data_ptr
addHeader
NetBuffer
mem_ptr
header_ptr
data_ptr
reformat
NetBuffer
mem_ptr
header_ptr
data_ptr

广播优化



```
bool socketmanager::multicast(multicast_list_t *connections, INetData *net_data) {
    bool result = false;
    task_t task;
    PM_LOG_FAILED_JUMP(connections);
    PM_LOG_FAILED_JUMP(connections->count > 0);
    task.type = TaskType::CMD_SOCKET_MULTICAST;
    task.content.args.ptr = net_data;
    for (size_t i = 0; i < connections->count; ++i) {
        auto conn = static_cast<Connection*>(connections->sockets[i]);
        assert(conn);
        task.net = conn;
        net_data->AddRef();
        conn->postTask(task);
    }
    result = true;
Exit0:
    return result;
}
```

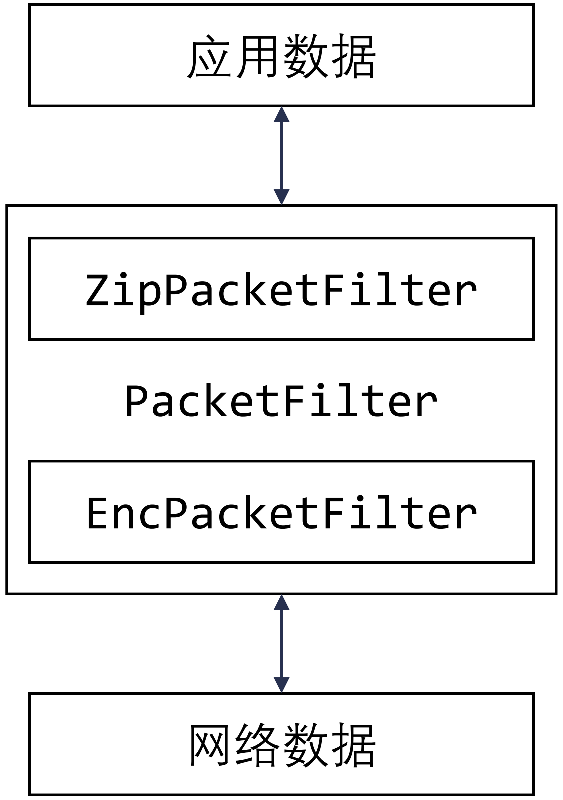


```
void doTaskCmdSocketMulticast(task_t &task) {
    assert(task.net);
    auto connection = static_cast<Connection*>(task.net);
    auto net_data = static_cast<INetData*>(task.content.args.ptr);
    if (connection->isConnected()) {
        connection->sendRaw(net_data->getData(), net_data->getSize());
    }
    net_data->Release();
}
```

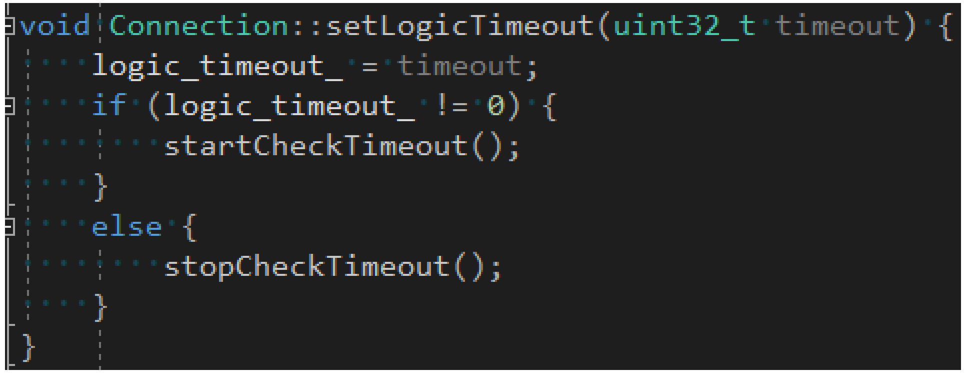
周边设施

数据处理

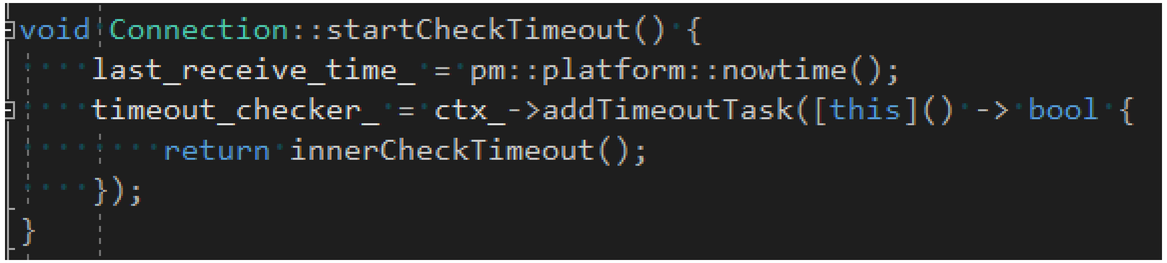
1. 网络线程执行
2. 压缩解压/加密解密



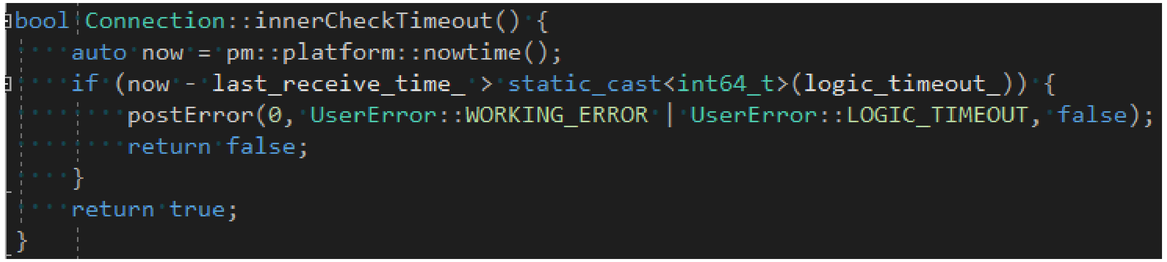
应用层心跳



```
void Connection::setLogicTimeout(uint32_t timeout) {
    logic_timeout_ = timeout;
    if (logic_timeout_ != 0) {
        startCheckTimeout();
    }
    else {
        stopCheckTimeout();
    }
}
```



```
void Connection::startCheckTimeout() {
    last_receive_time_ = pm::platform::nowtime();
    timeout_checker_ = ctx_->addTimeoutTask([this]() -> bool {
        return innerCheckTimeout();
    });
}
```

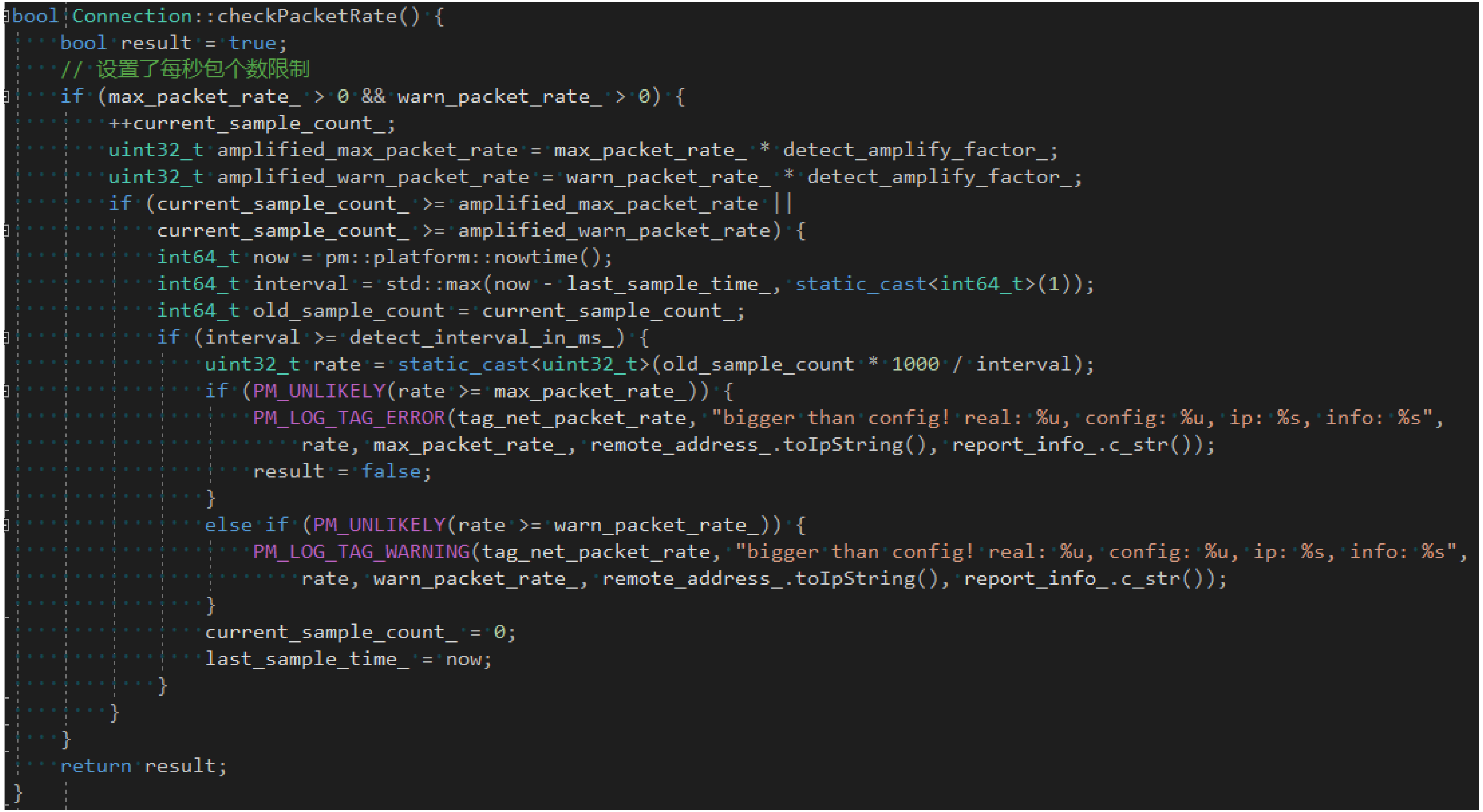


```
bool Connection::innerCheckTimeout() {
    auto now = pm::platform::nowtime();
    if (now - last_receive_time_ > static_cast<int64_t>(logic_timeout_)) {
        postError(0, UserError::WORKING_ERROR | UserError::LOGIC_TIMEOUT, false);
        return false;
    }
    return true;
}
```

周边设施

报警机制

1. 单包大小限制
2. 时间段包数限制

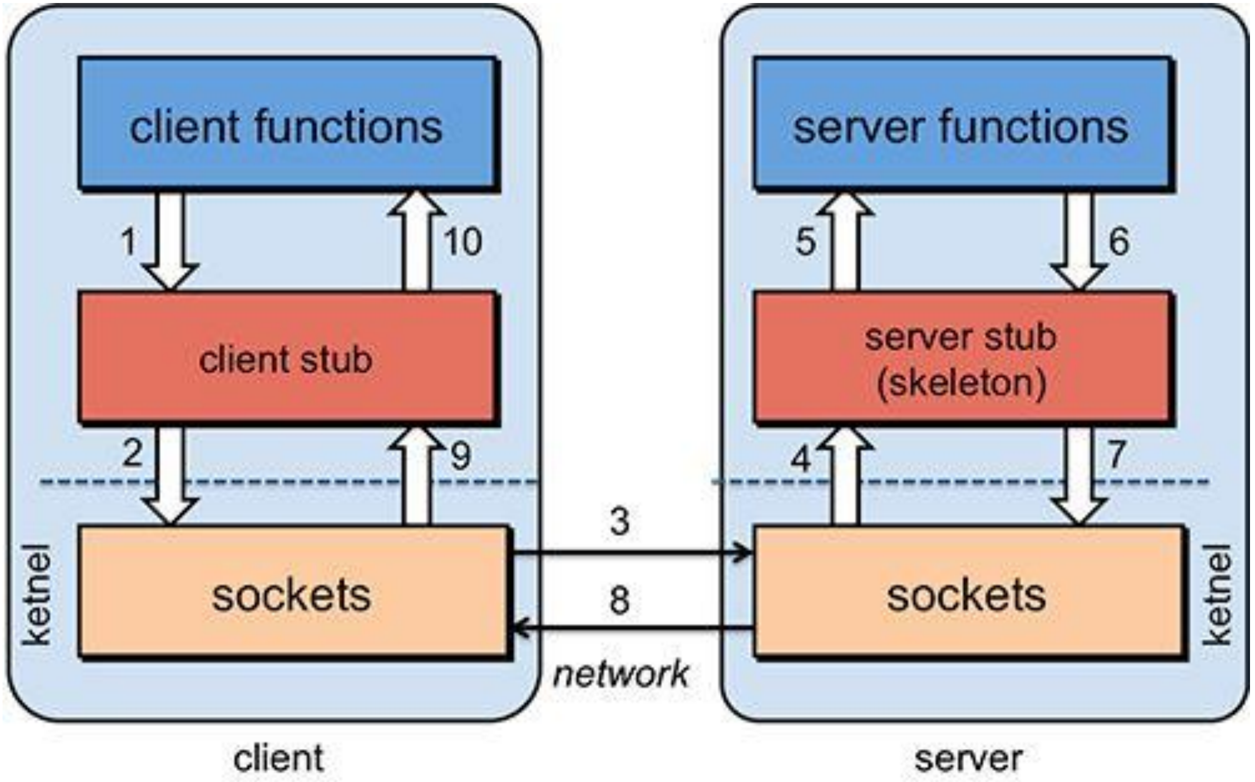


```cpp
bool Connection::checkPacketRate() {
    bool result = true;
    // 设置了每秒包个数限制
    if (max_packet_rate_ > 0 && warn_packet_rate_ > 0) {
        ++current_sample_count_;
        uint32_t amplified_max_packet_rate = max_packet_rate_ * detect_amplify_factor_;
        uint32_t amplified_warn_packet_rate = warn_packet_rate_ * detect_amplify_factor_;
        if (current_sample_count_ >= amplified_max_packet_rate ||
            current_sample_count_ >= amplified_warn_packet_rate) {
            int64_t now = pm::platform::nowtime();
            int64_t interval = std::max(now - last_sample_time_, static_cast<int64_t>(1));
            int64_t old_sample_count = current_sample_count_;
            if (interval >= detect_interval_in_ms_) {
                uint32_t rate = static_cast<uint32_t>(old_sample_count * 1000 / interval);
                if (PM_UNLIKELY(rate >= max_packet_rate_)) {
                    PM_LOG_TAG_ERROR(tag_net_packet_rate, "bigger than config! real: %u, config: %u, ip: %s, info: %s",
                        rate, max_packet_rate_, remote_address_.toIpString(), report_info_.c_str());
                    result = false;
                }
                else if (PM_UNLIKELY(rate >= warn_packet_rate_)) {
                    PM_LOG_TAG_WARNING(tag_net_packet_rate, "bigger than config! real: %u, config: %u, ip: %s, info: %s",
                        rate, warn_packet_rate_, remote_address_.toIpString(), report_info_.c_str());
                }
                current_sample_count_ = 0;
                last_sample_time_ = now;
            }
        }
    }
    return result;
}
```

03

RPC设计

RPC简介



Call Id映射

序列化/反序列化

网络传输

技术选型

网络传输

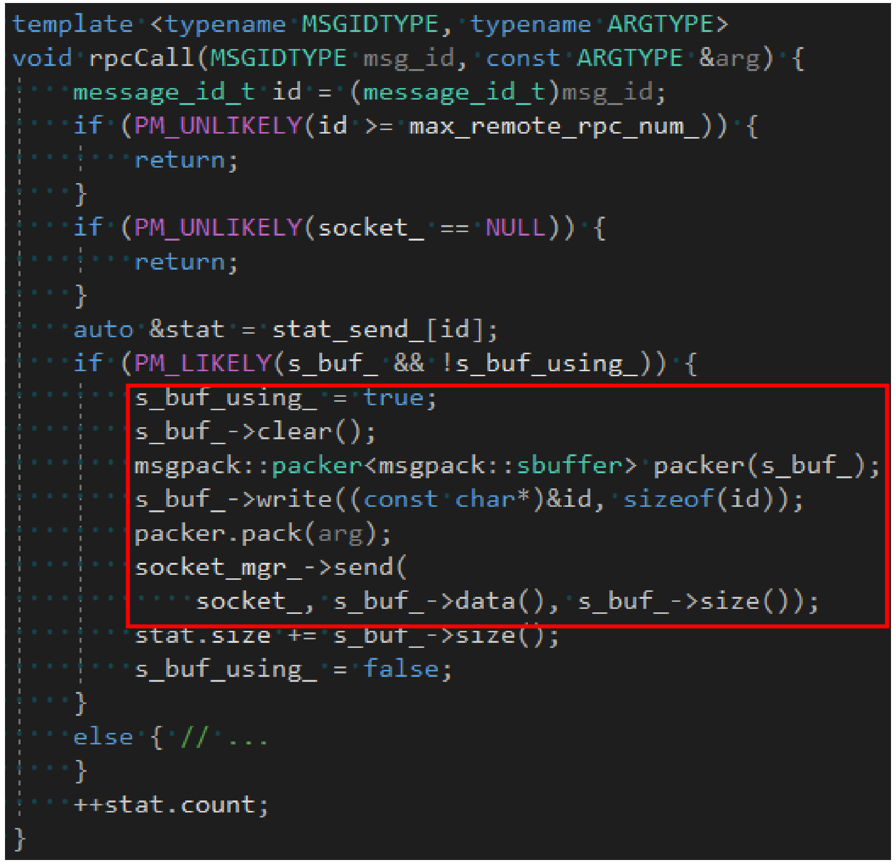
netlib

序列化/反序列化

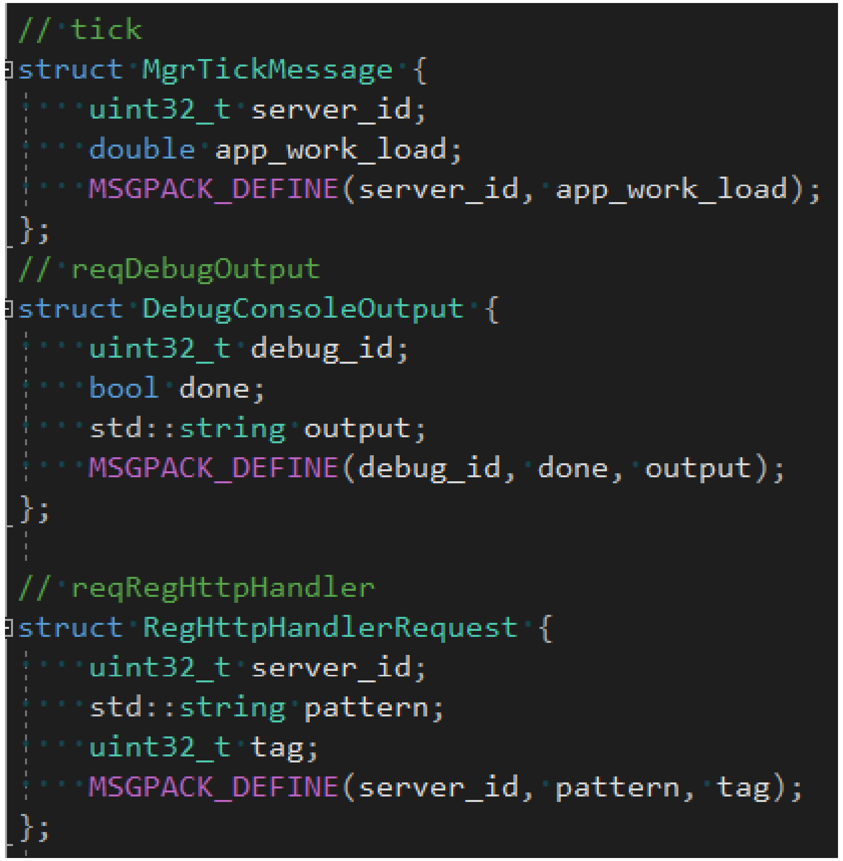
MessagePack

Call Id映射

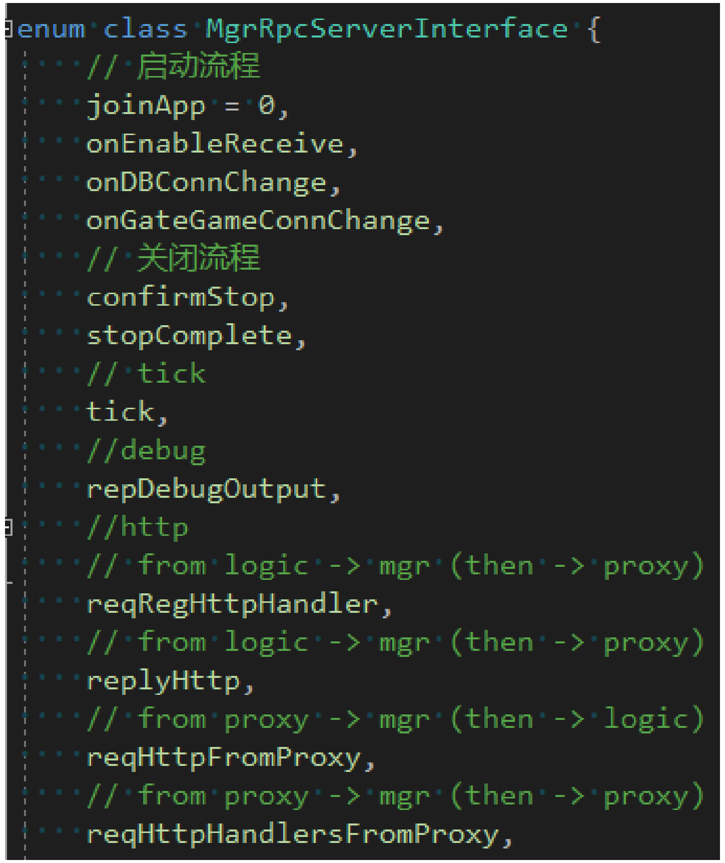
C++ enum



```
template <typename MSGIDTYPE, typename ARGTYPE>
void rpcCall(MSGIDTYPE msg_id, const ARGTYPE &arg) {
    message_id_t id = (message_id_t)msg_id;
    if (PM_UNLIKELY(id >= max_remote_rpc_num_)) {
        return;
    }
    if (PM_UNLIKELY(socket_ == NULL)) {
        return;
    }
    auto &stat = stat_send_[id];
    if (PM_LIKELY(s_buf_ && !s_buf_using_)) {
        s_buf_using_ = true;
        s_buf_->clear();
        msgpack::packer<msgpack::sbuffer> packer(s_buf_);
        s_buf_->write((const char*)&id, sizeof(id));
        packer.pack(arg);
        socket_mgr_->send(
            socket_, s_buf_->data(), s_buf_->size());
        stat.size += s_buf_->size();
        s_buf_using_ = false;
    }
    else { // ...
    }
    ++stat.count;
}
```

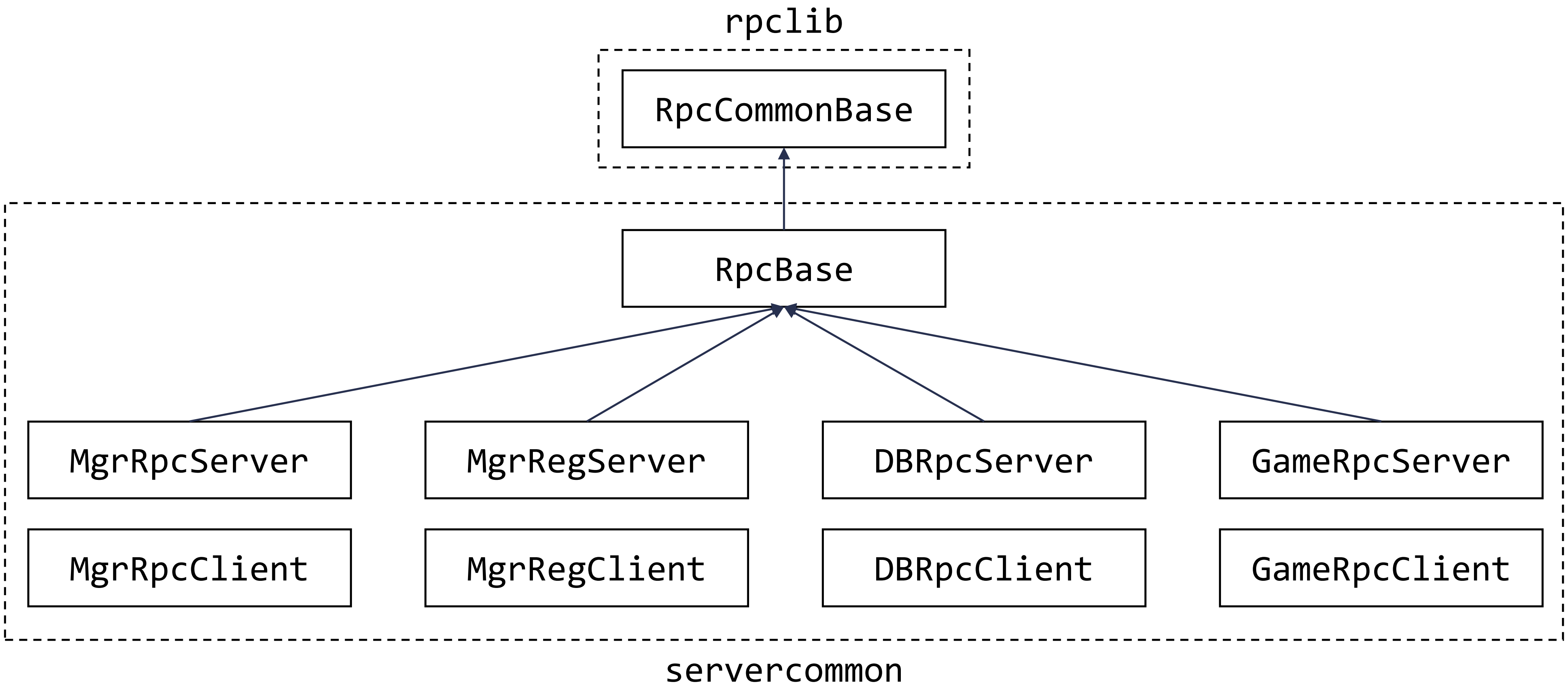


```
// tick
struct MgrTickMessage {
    uint32_t server_id;
    double app_work_load;
    MSGPACK_DEFINE(server_id, app_work_load);
};
// reqDebugOutput
struct DebugConsoleOutput {
    uint32_t debug_id;
    bool done;
    std::string output;
    MSGPACK_DEFINE(debug_id, done, output);
};

// reqRegHttpHandler
struct RegHttpHandlerRequest {
    uint32_t server_id;
    std::string pattern;
    uint32_t tag;
    MSGPACK_DEFINE(server_id, pattern, tag);
};
```



```
enum class MgrRpcServerInterface {
    // 启动流程
    joinApp = 0,
    onEnableReceive,
    onDBConnChange,
    onGateGameConnChange,
    // 关闭流程
    confirmStop,
    stopComplete,
    // tick
    tick,
    //debug
    repDebugOutput,
    //http
    // from logic -> mgr (then -> proxy)
    reqRegHttpHandler,
    // from logic -> mgr (then -> proxy)
    replyHttp,
    // from proxy -> mgr (then -> logic)
    reqHttpFromProxy,
    // from proxy -> mgr (then -> proxy)
    reqHttpHandlersFromProxy,
```

继承模型



rpclib

RpcCommonBase

RpcBase

MgrRpcServer

MgrRegServer

DBRpcServer

GameRpcServer

MgrRpcClient

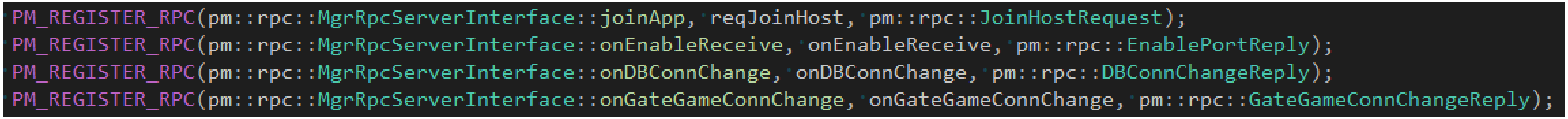
MgrRegClient

DBRpcClient

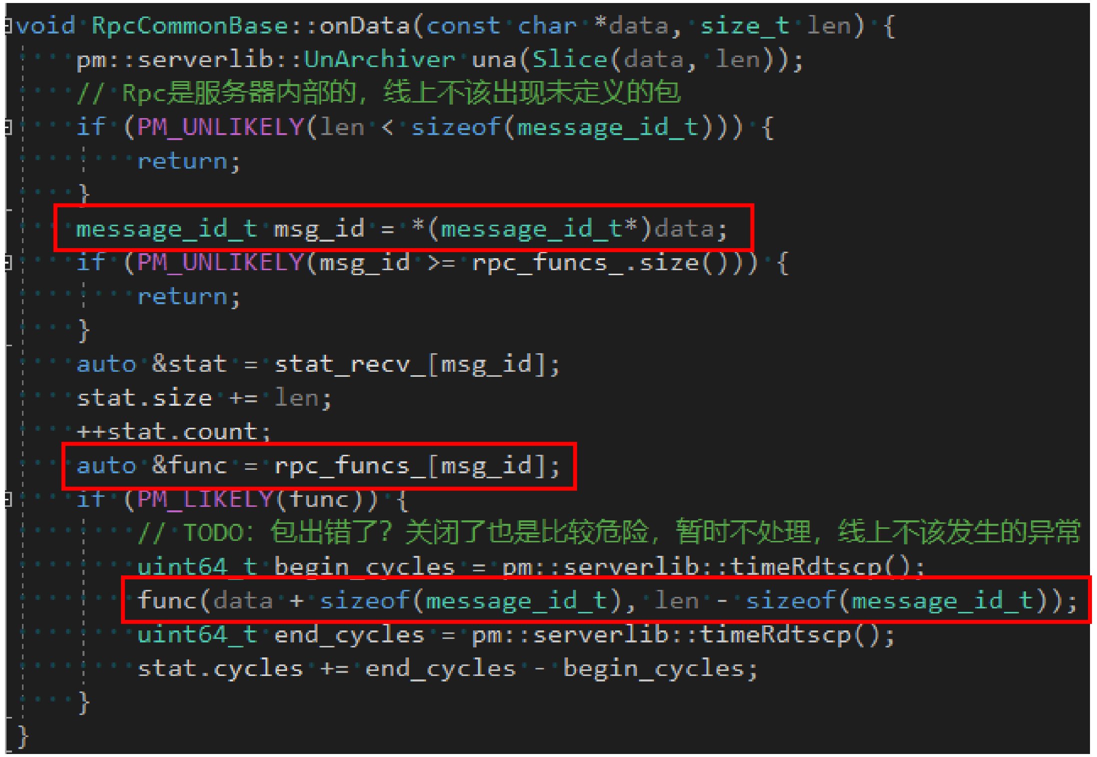
GameRpcClient

servercommon

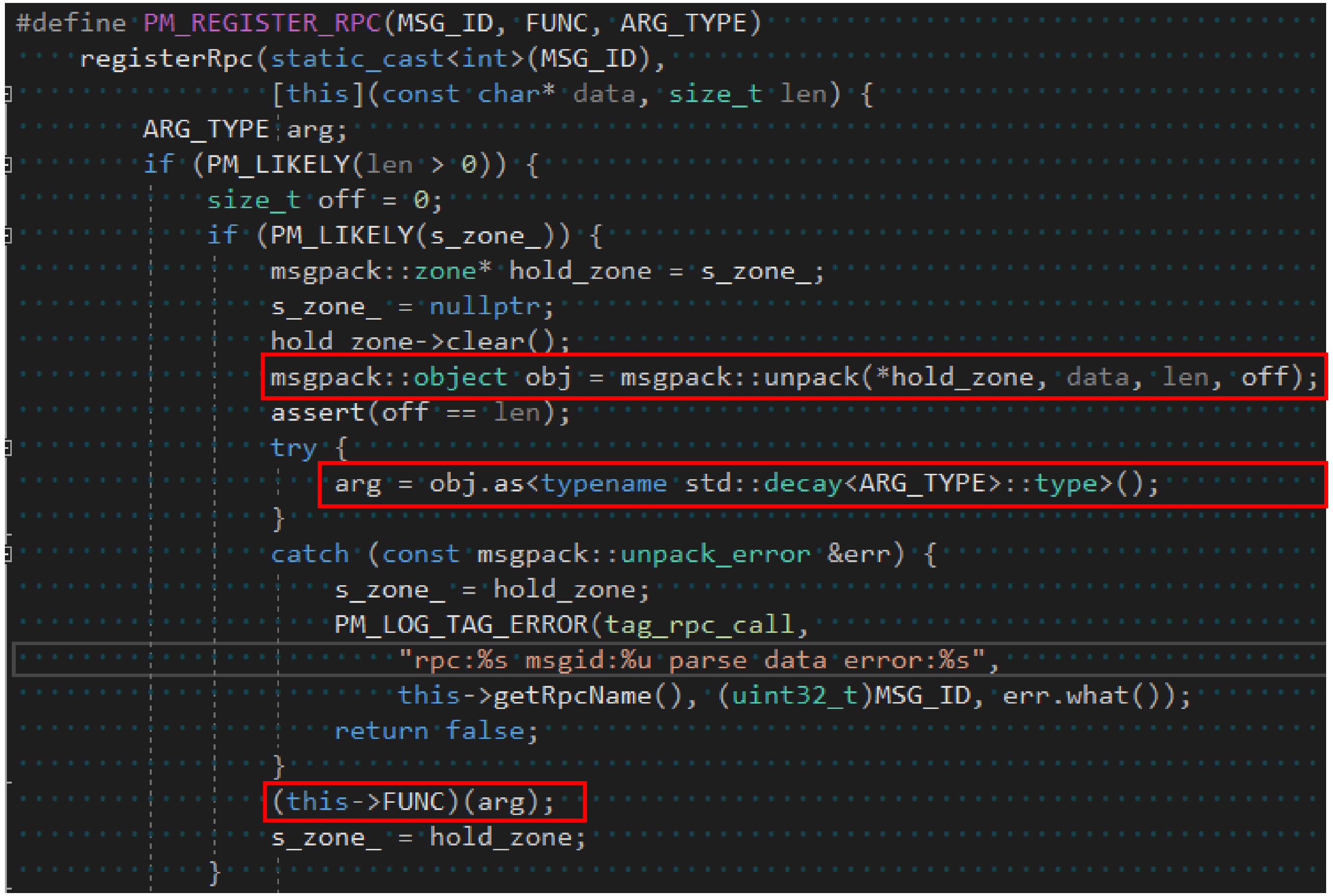
RPC实现



```
PM_REGISTER_RPC(pm::rpc::MgrRpcServerInterface::joinApp, reqJoinHost, pm::rpc::JoinHostRequest);
PM_REGISTER_RPC(pm::rpc::MgrRpcServerInterface::onEnableReceive, onEnableReceive, pm::rpc::EnablePortReply);
PM_REGISTER_RPC(pm::rpc::MgrRpcServerInterface::onDBConnChange, onDBConnChange, pm::rpc::DBConnChangeReply);
PM_REGISTER_RPC(pm::rpc::MgrRpcServerInterface::onGateGameConnChange, onGateGameConnChange, pm::rpc::GateGameConnChangeReply);
```

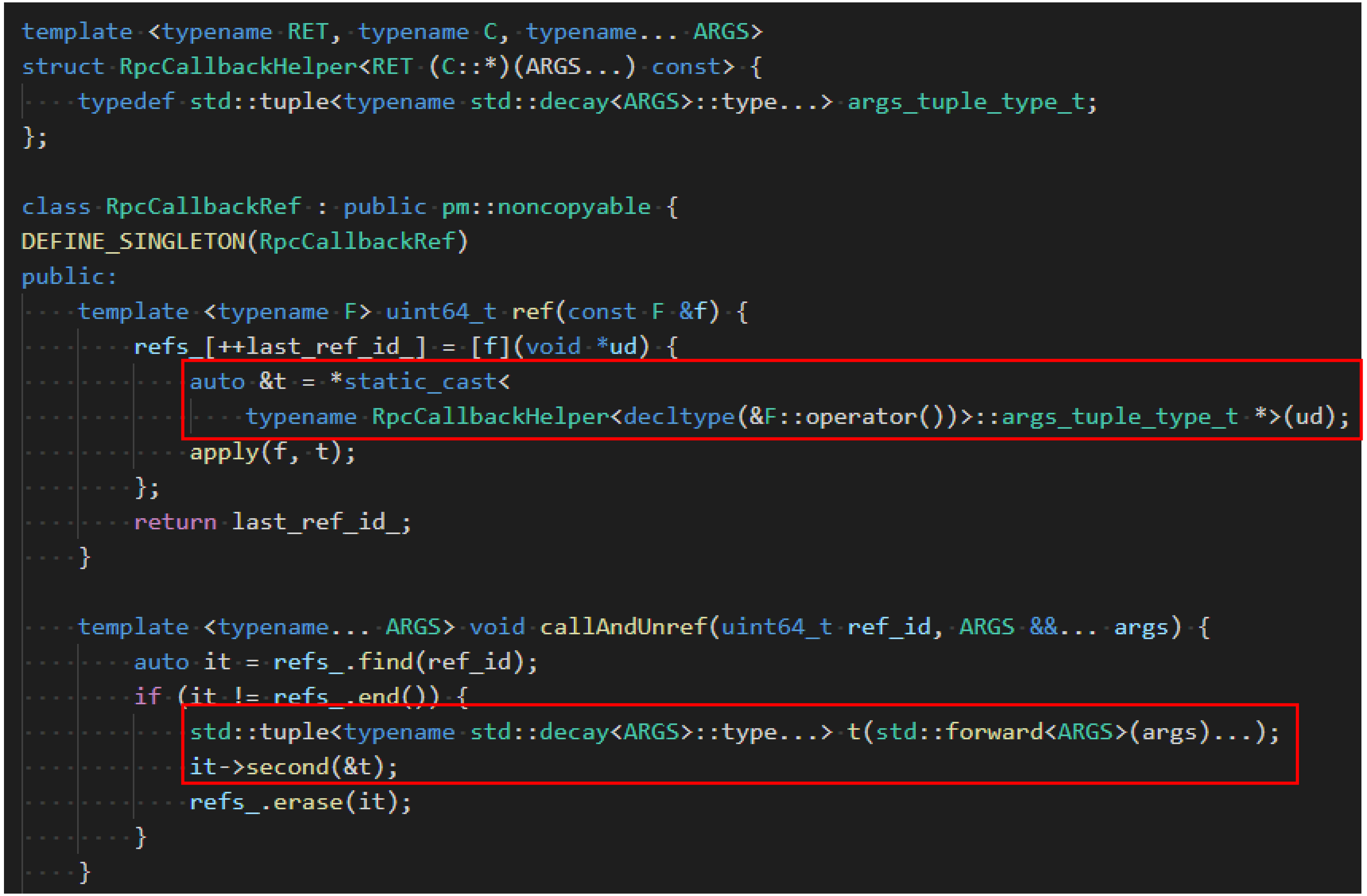


```
void RpcCommonBase::onData(const char *data, size_t len) {
    pm::serverlib::UnArchiver una(Slice(data, len));
    // Rpc是服务器内部的，线上不该出现未定义的包
    if (PM_UNLIKELY(len < sizeof(message_id_t))) {
        return;
    }
    message_id_t msg_id = *(message_id_t*)data;
    if (PM_UNLIKELY(msg_id >= rpc_funcs_.size())) {
        return;
    }
    auto &stat = stat_recv_[msg_id];
    stat.size += len;
    ++stat.count;
    auto &func = rpc_funcs_[msg_id];
    if (PM_LIKELY(func)) {
        // TODO：包出错了？关闭了也是比较危险，暂时不处理，线上不该发生的异常
        uint64_t begin_cycles = pm::serverlib::timeRdtscp();
        func(data + sizeof(message_id_t), len - sizeof(message_id_t));
        uint64_t end_cycles = pm::serverlib::timeRdtscp();
        stat.cycles += end_cycles - begin_cycles;
    }
}
```



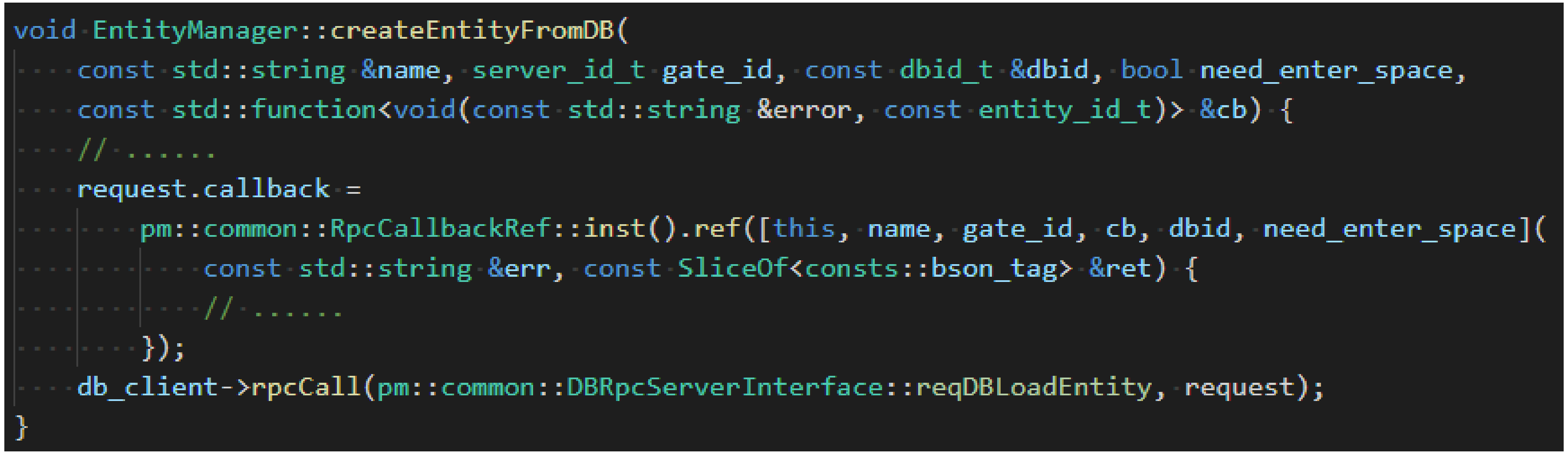
```
#define PM_REGISTER_RPC(MSG_ID, FUNC, ARG_TYPE)
    registerRpc(static_cast<int>(MSG_ID),
                [this](const char* data, size_t len) {
        ARG_TYPE arg;
        if (PM_LIKELY(len > 0)) {
            size_t off = 0;
            if (PM_LIKELY(s_zone_)) {
                msgpack::zone* hold_zone = s_zone_;
                s_zone_ = nullptr;
                hold_zone->clear();
                msgpack::object obj = msgpack::unpack(*hold_zone, data, len, off);
                assert(off == len);
                try {
                    arg = obj.as<typename std::decay<ARG_TYPE>::type>();
                }
                catch (const msgpack::unpack_error &err) {
                    s_zone_ = hold_zone;
                    PM_LOG_TAG_ERROR(tag_rpc_call,
                        "rpc:%s msgid:%u parse data error:%s",
                        this->getRpcName(), (uint32_t)MSG_ID, err.what());
                    return false;
                }
                (this->FUNC)(arg);
                s_zone_ = hold_zone;
            }
```

RPC回调（旧）

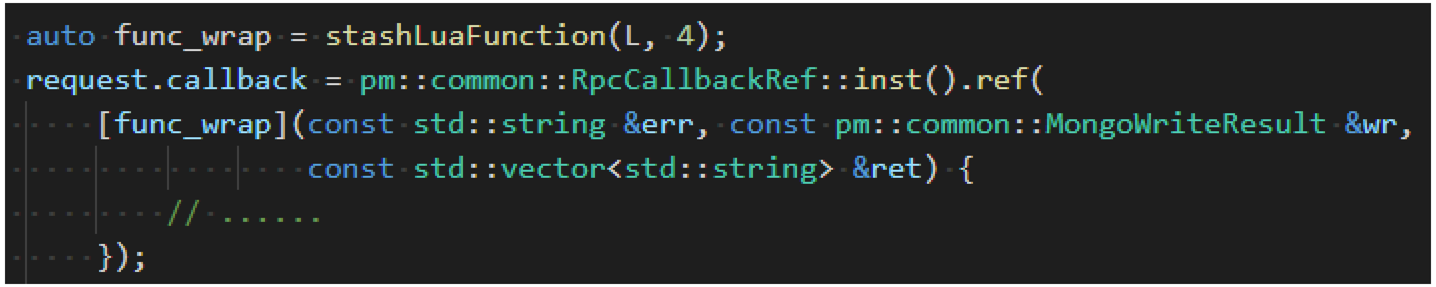


```
template <typename RET, typename C, typename... ARGS>
struct RpcCallbackHelper<RET (C::*)(ARGS...) const> {
    typedef std::tuple<typename std::decay<ARGS>::type...> args_tuple_type_t;
};

class RpcCallbackRef : public pm::noncopyable {
DEFINE_SINGLETON(RpcCallbackRef)
public:
    template <typename F> uint64_t ref(const F &f) {
        refs_[++last_ref_id_] = [f](void *ud) {
            auto &t = *static_cast<
                typename RpcCallbackHelper<decltype(&F::operator())>::args_tuple_type_t *>(ud);
            apply(f, t);
        };
        return last_ref_id_;
    }

    template <typename... ARGS> void callAndUnref(uint64_t ref_id, ARGS &&... args) {
        auto it = refs_.find(ref_id);
        if (it != refs_.end()) {
            std::tuple<typename std::decay<ARGS>::type...> t(std::forward<ARGS>(args)...);
            it->second(&t);
            refs_.erase(it);
        }
    }
```

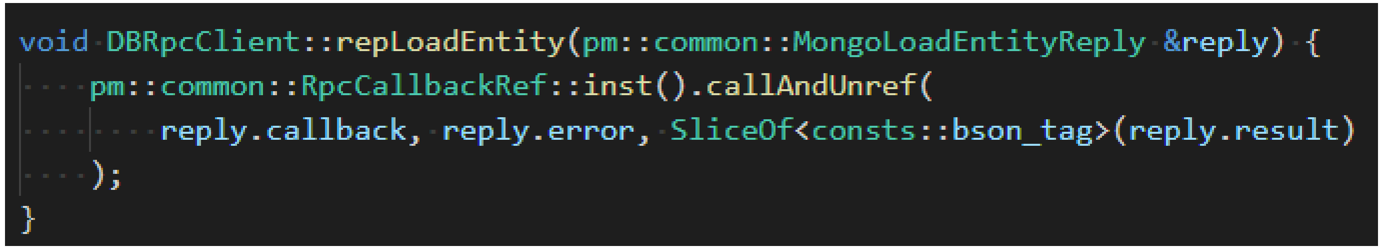
RPC回调（旧）



```
void EntityManager::createEntityFromDB(
    const std::string &name, server_id_t gate_id, const dbid_t &dbid, bool need_enter_space,
    const std::function<void(const std::string &error, const entity_id_t)> &cb) {
    // ......
    request.callback =
        pm::common::RpcCallbackRef::inst().ref([this, name, gate_id, cb, dbid, need_enter_space](
            const std::string &err, const SliceOf<consts::bson_tag> &ret) {
            // ......
        });
    db_client->rpcCall(pm::common::DBRpcServerInterface::reqDBLoadEntity, request);
}
```

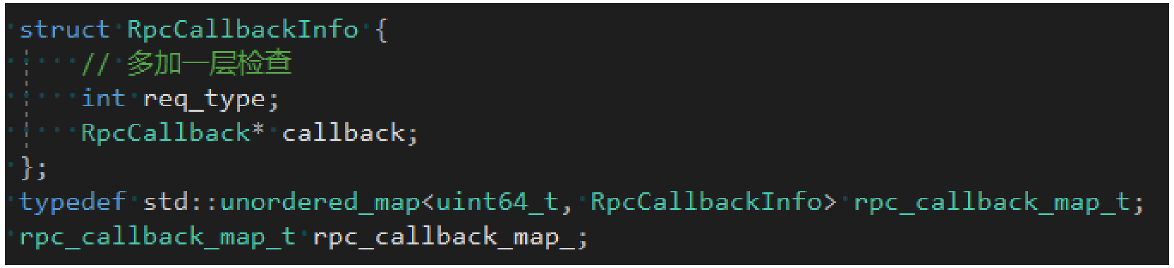


```
auto func_wrap = stashLuaFunction(L, 4);
request.callback = pm::common::RpcCallbackRef::inst().ref(
    [func_wrap](const std::string &err, const pm::common::MongoWriteResult &wr,
                const std::vector<std::string> &ret) {
        // ......
    });
```

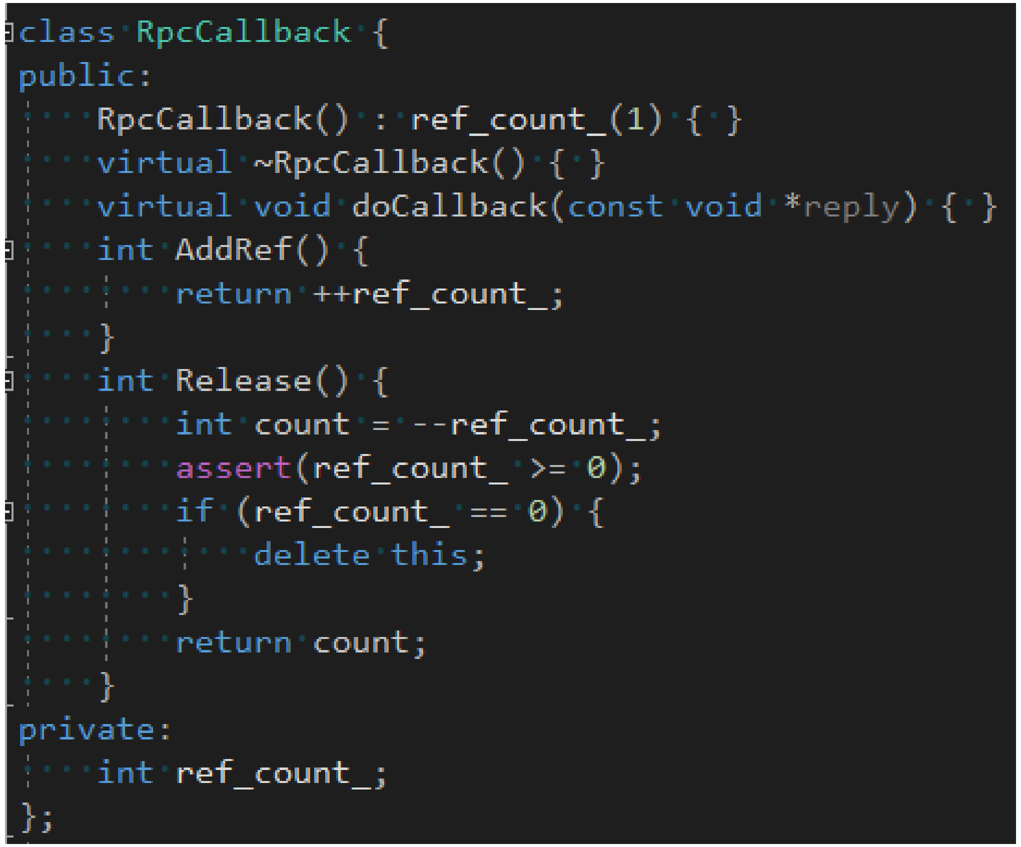


```
void DBRpcClient::repLoadEntity(pm::common::MongoLoadEntityReply &reply) {
    pm::common::RpcCallbackRef::inst().callAndUnref(
        reply.callback, reply.error, SliceOf<consts::bson_tag>(reply.result)
    );
}
```

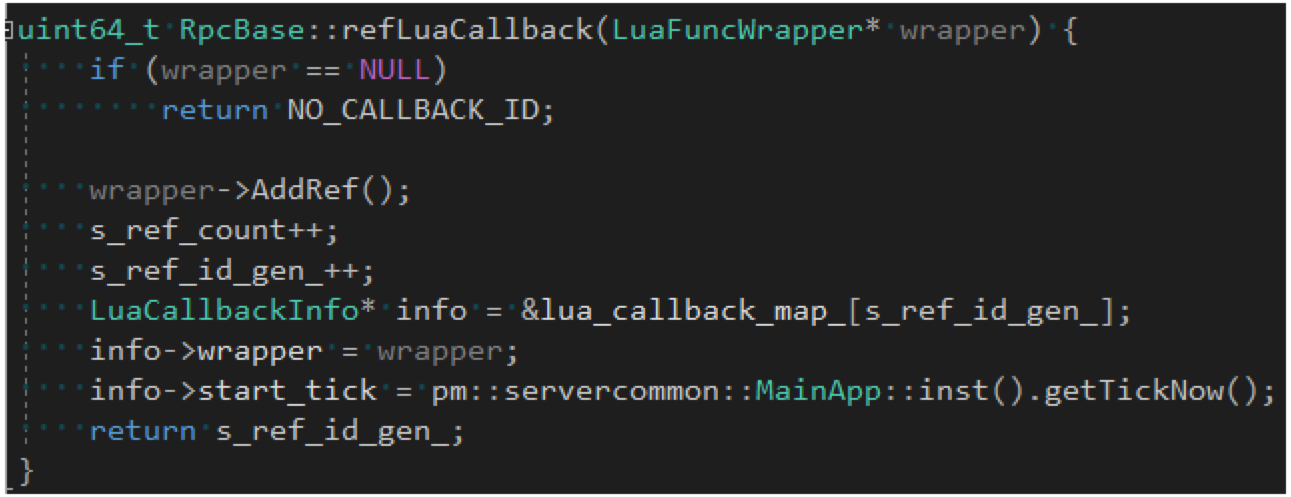
RPC回调（新）



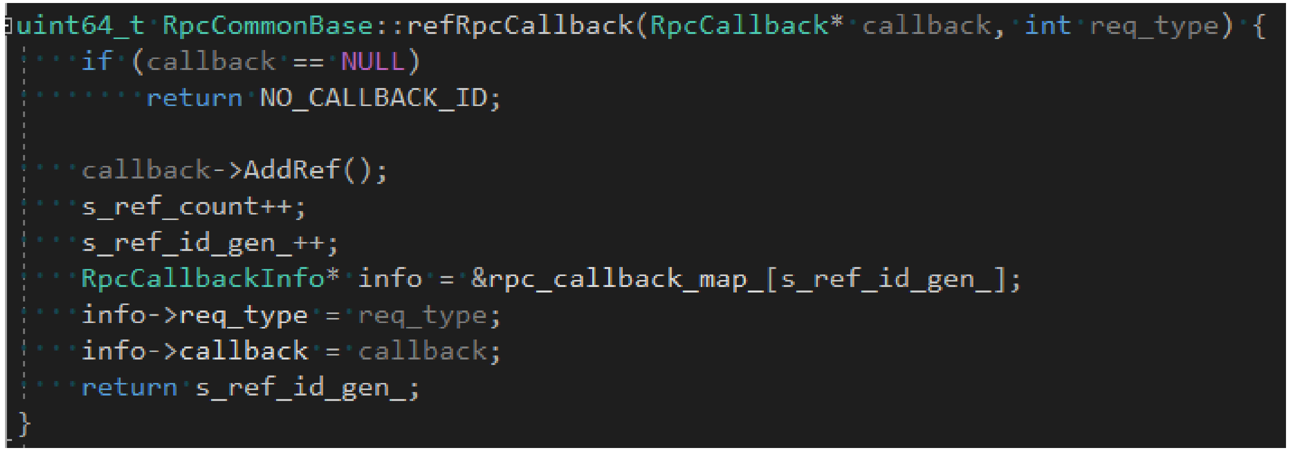
```
struct RpcCallbackInfo {
    // 多加一层检查
    int req_type;
    RpcCallback* callback;
};
typedef std::unordered_map<uint64_t, RpcCallbackInfo> rpc_callback_map_t;
rpc_callback_map_t rpc_callback_map_;
```



```
class RpcCallback {
public:
    RpcCallback() : ref_count_(1) { }
    virtual ~RpcCallback() { }
    virtual void doCallback(const void *reply) { }
    int AddRef() {
        return ++ref_count_;
    }
    int Release() {
        int count = --ref_count_;
        assert(ref_count_ >= 0);
        if (ref_count_ == 0) {
            delete this;
        }
        return count;
    }
private:
    int ref_count_;
};
```

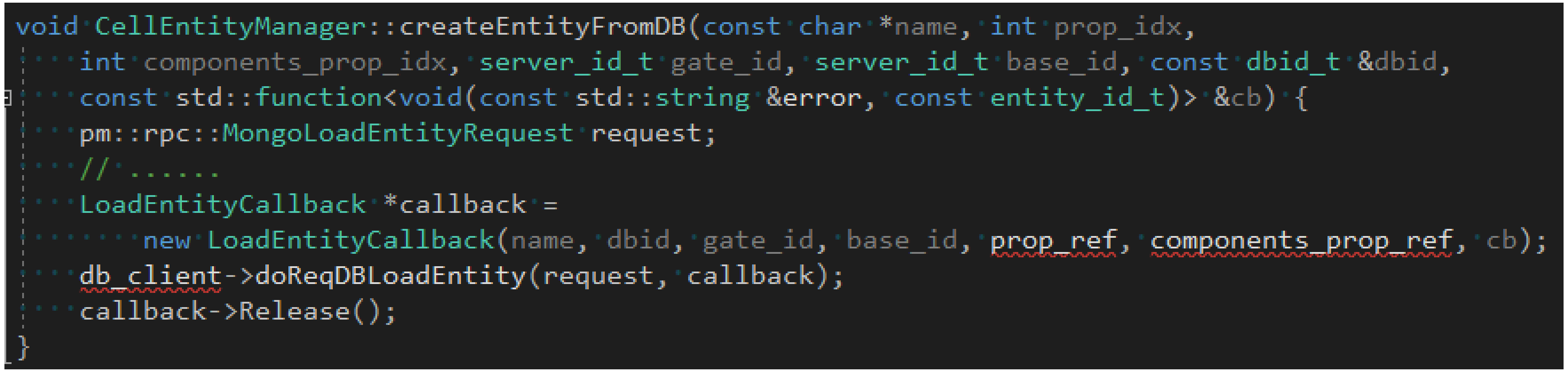


```
uint64_t RpcBase::refLuaCallback(LuaFuncWrapper* wrapper) {
    if (wrapper == NULL)
        return NO_CALLBACK_ID;

    wrapper->AddRef();
    s_ref_count++;
    s_ref_id_gen_++;
    LuaCallbackInfo* info = &lua_callback_map_[s_ref_id_gen_];
    info->wrapper = wrapper;
    info->start_tick = pm::servercommon::MainApp::inst().getTickNow();
    return s_ref_id_gen_;
}
```

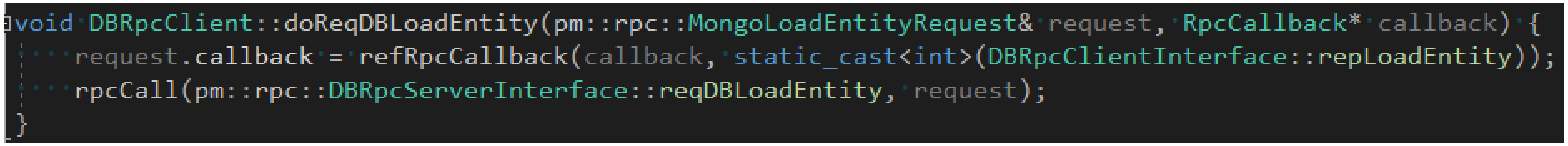


```
uint64_t RpcCommonBase::refRpcCallback(RpcCallback* callback, int req_type) {
    if (callback == NULL)
        return NO_CALLBACK_ID;

    callback->AddRef();
    s_ref_count++;
    s_ref_id_gen_++;
    RpcCallbackInfo* info = &rpc_callback_map_[s_ref_id_gen_];
    info->req_type = req_type;
    info->callback = callback;
    return s_ref_id_gen_;
}
```

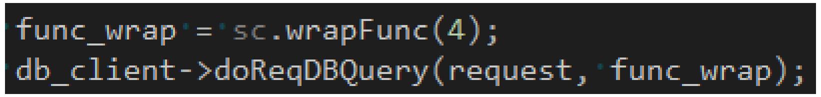
RPC回调（新）



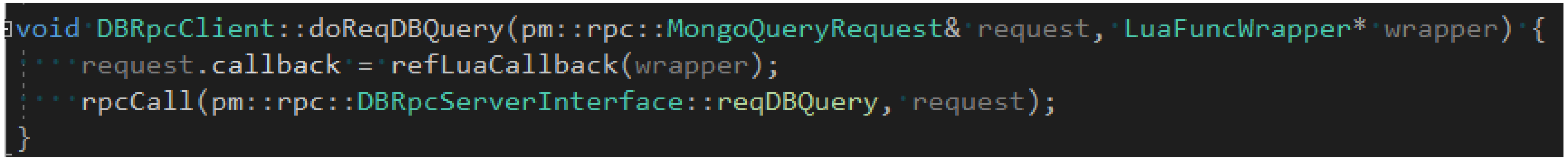
```cpp
void CellEntityManager::createEntityFromDB(const char *name, int prop_idx,
    int components_prop_idx, server_id_t gate_id, server_id_t base_id, const dbid_t &dbid,
    const std::function<void(const std::string &error, const entity_id_t)> &cb) {
    pm::rpc::MongoLoadEntityRequest request;
    // ......
    LoadEntityCallback *callback =
        new LoadEntityCallback(name, dbid, gate_id, base_id, prop_ref, components_prop_ref, cb);
    db_client->doReqDBLoadEntity(request, callback);
    callback->Release();
}
```



```cpp
void DBRpcClient::doReqDBLoadEntity(pm::rpc::MongoLoadEntityRequest& request, RpcCallback* callback) {
    request.callback = refRpcCallback(callback, static_cast<int>(DBRpcClientInterface::repLoadEntity));
    rpcCall(pm::rpc::DBRpcServerInterface::reqDBLoadEntity, request);
}
```

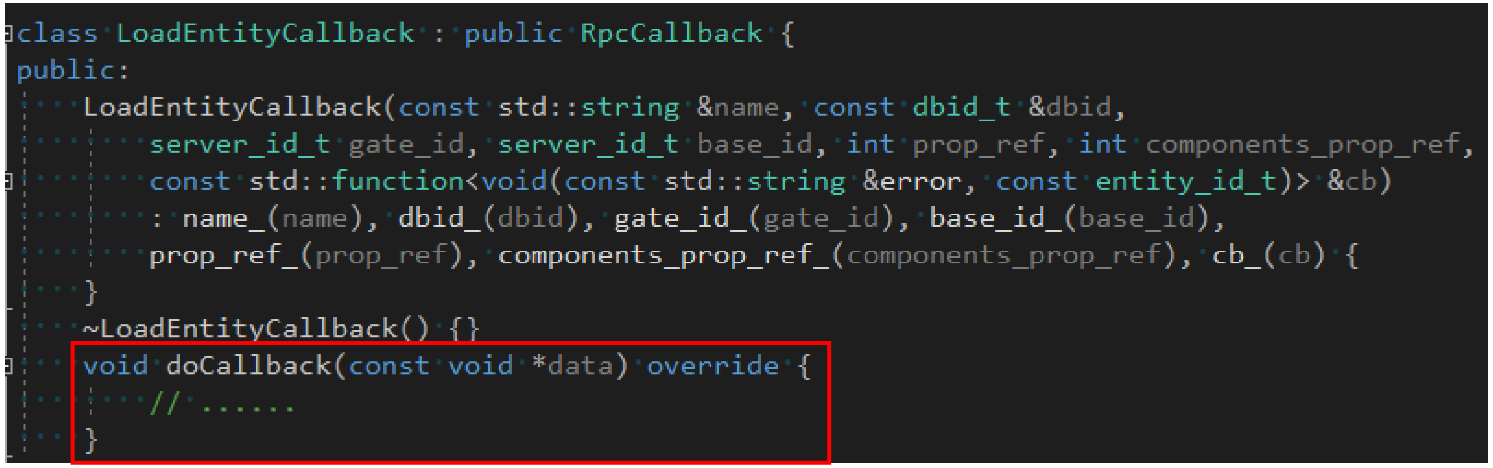


```lua
func_wrap = sc.wrapFunc(4);
db_client->doReqDBQuery(request, func_wrap);
```

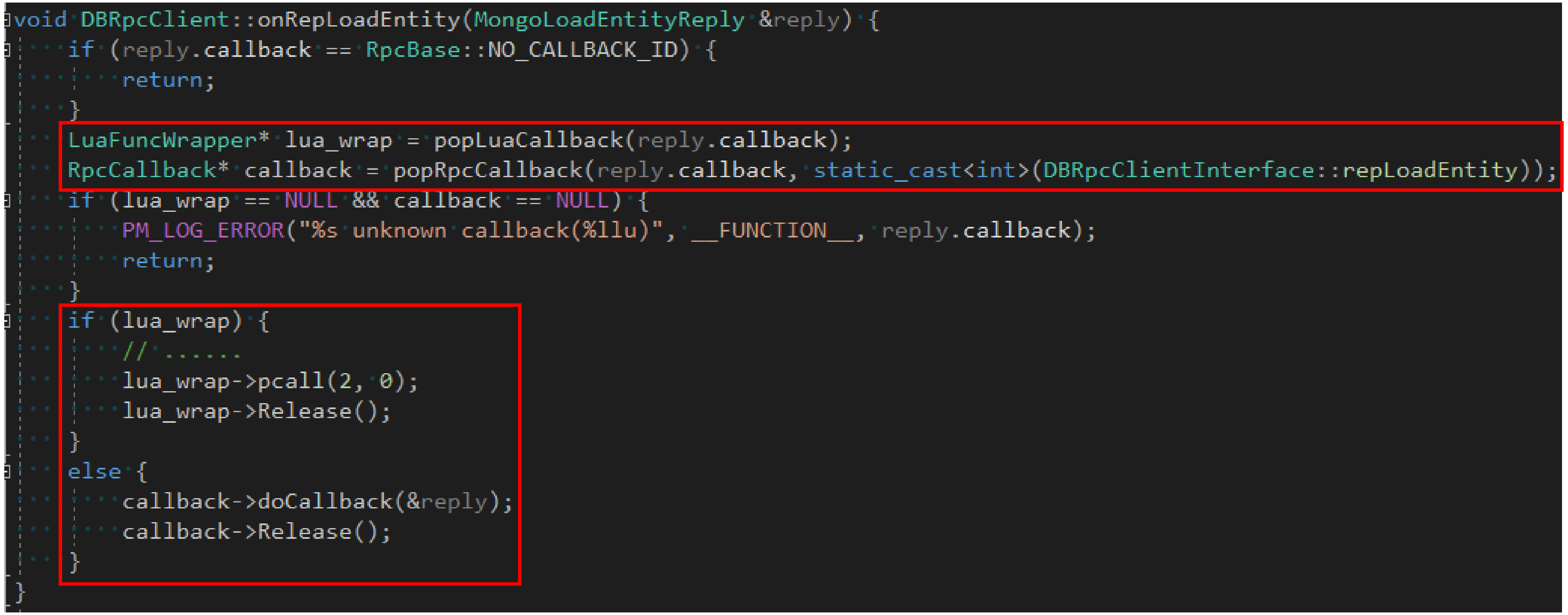


```cpp
void DBRpcClient::doReqDBQuery(pm::rpc::MongoQueryRequest& request, LuaFuncWrapper* wrapper) {
    request.callback = refLuaCallback(wrapper);
    rpcCall(pm::rpc::DBRpcServerInterface::reqDBQuery, request);
}
```

RPC回调（新）

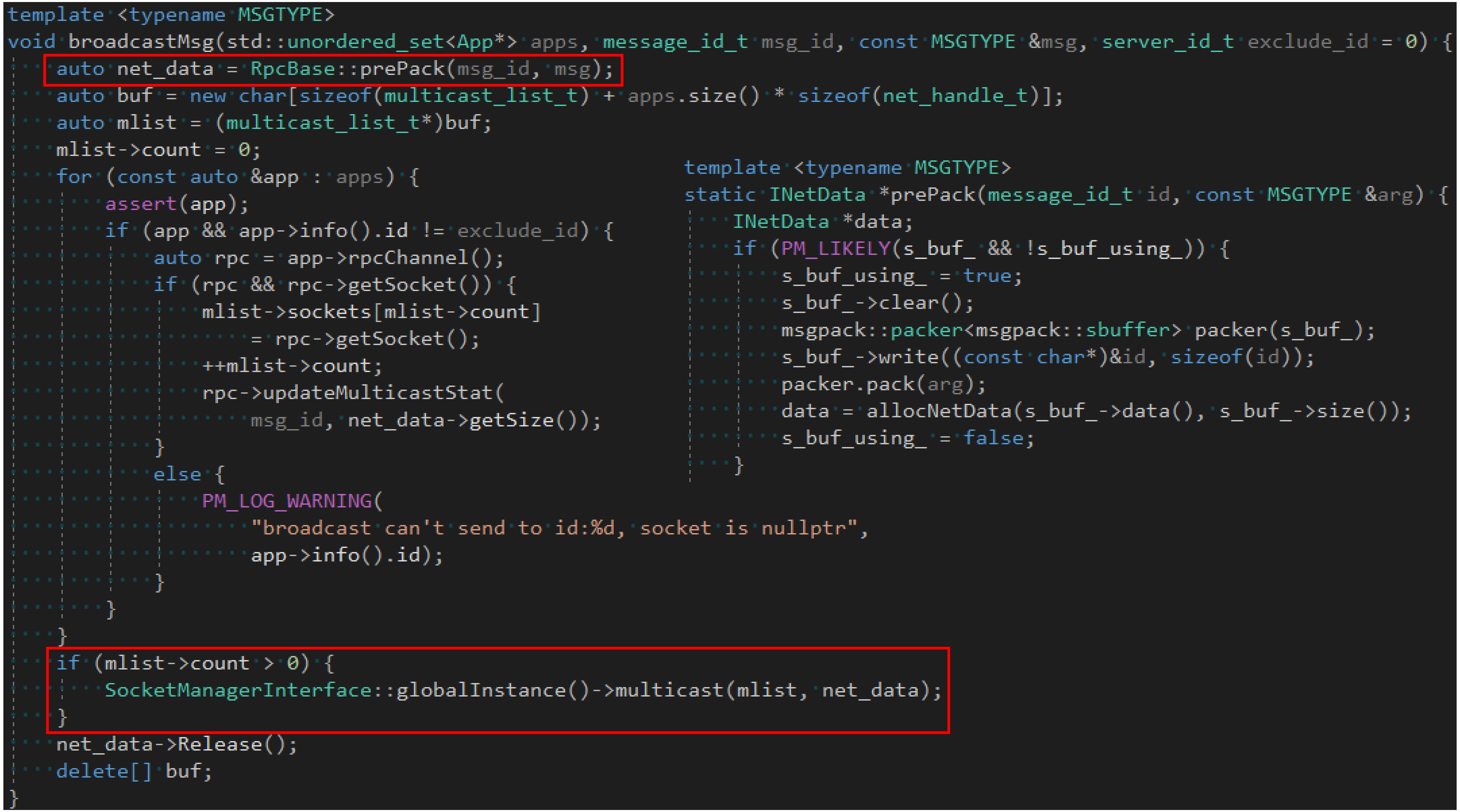


```
class LoadEntityCallback : public RpcCallback {
public:
    LoadEntityCallback(const std::string &name, const dbid_t &dbid,
        server_id_t gate_id, server_id_t base_id, int prop_ref, int components_prop_ref,
        const std::function<void(const std::string &error, const entity_id_t)> &cb)
        : name_(name), dbid_(dbid), gate_id_(gate_id), base_id_(base_id),
        prop_ref_(prop_ref), components_prop_ref_(components_prop_ref), cb_(cb) {
    }
    ~LoadEntityCallback() {}
    void doCallback(const void *data) override {
        // ......
    }
```



```
void DBRpcClient::onRepLoadEntity(MongoLoadEntityReply &reply) {
    if (reply.callback == RpcBase::NO_CALLBACK_ID) {
        return;
    }
    LuaFuncWrapper* lua_wrap = popLuaCallback(reply.callback);
    RpcCallback* callback = popRpcCallback(reply.callback, static_cast<int>(DBRpcClientInterface::repLoadEntity));
    if (lua_wrap == NULL && callback == NULL) {
        PM_LOG_ERROR("%s unknown callback(%llu)", __FUNCTION__, reply.callback);
        return;
    }
    if (lua_wrap) {
        // ......
        lua_wrap->pcall(2, 0);
        lua_wrap->Release();
    }
    else {
        callback->doCallback(&reply);
        callback->Release();
    }
}
```

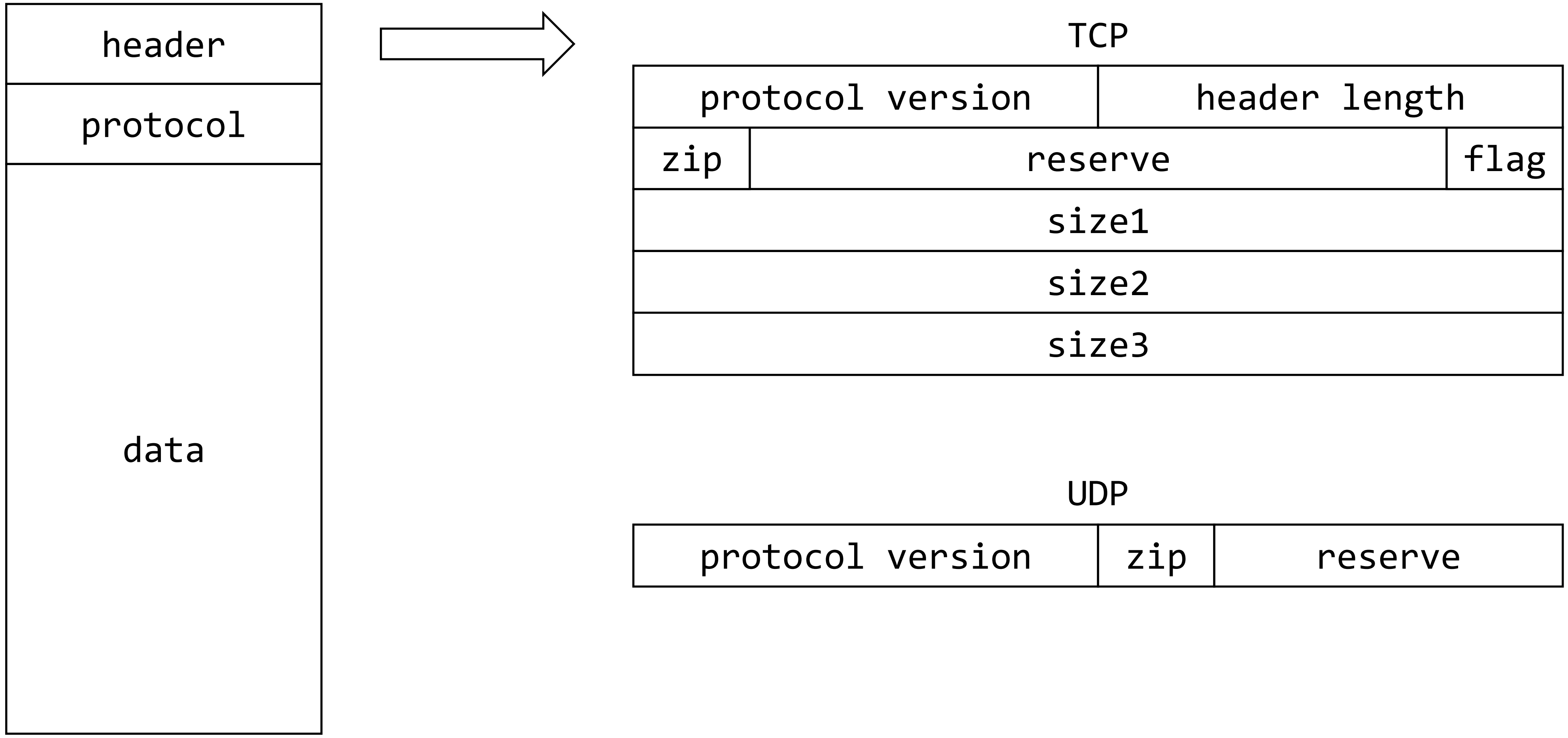
广播优化



```cpp
template <typename MSGTYPE>
void broadcastMsg(std::unordered_set<App*> apps, message_id_t msg_id, const MSGTYPE &msg, server_id_t exclude_id = 0) {
    auto net_data = RpcBase::prePack(msg_id, msg);
    auto buf = new char[sizeof(multicast_list_t) + apps.size() * sizeof(net_handle_t)];
    auto mlist = (multicast_list_t*)buf;
    mlist->count = 0;
    for (const auto &app : apps) {
        assert(app);
        if (app && app->info().id != exclude_id) {
            auto rpc = app->rpcChannel();
            if (rpc && rpc->getSocket()) {
                mlist->sockets[mlist->count]
                    = rpc->getSocket();
                ++mlist->count;
                rpc->updateMulticastStat(
                    msg_id, net_data->getSize());
            }
            else {
                PM_LOG_WARNING(
                    "broadcast can't send to id:%d, socket is nullptr",
                    app->info().id);
            }
        }
    }
    if (mlist->count > 0) {
        SocketManagerInterface::globalInstance()->multicast(mlist, net_data);
    }
    net_data->Release();
    delete[] buf;
}
```

```cpp
template <typename MSGTYPE>
static INetData *prePack(message_id_t id, const MSGTYPE &arg) {
    INetData *data;
    if (PM_LIKELY(s_buf_ && !s_buf_using_)) {
        s_buf_using_ = true;
        s_buf_->clear();
        msgpack::packer<msgpack::sbuffer> packer(s_buf_);
        s_buf_->write((const char*)&id, sizeof(id));
        packer.pack(arg);
        data = allocNetData(s_buf_->data(), s_buf_->size());
        s_buf_using_ = false;
    }
```

客户端协议



header

protocol

data

TCP

protocol version | header length

zip | reserve | flag

size1

size2

size3

UDP

protocol version | zip | reserve

04

总结展望

总结

模块设计需要高内聚低耦合

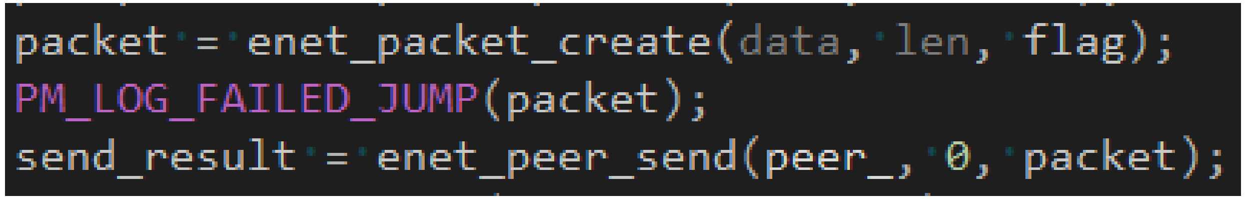
接口清晰易用，隐藏内部细节

内部实现代码简洁，易于维护

底层模块需要抠细节

后续优化

通过ENET_PACKET_FLAG_NO_ALLOCATE设置以及托管enet的内存分配



```
packet = enet_packet_create(data, len, flag);
PM_LOG_FAILED_JUMP(packet);
send_result = enet_peer_send(peer_, 0, packet);
```

NetMemory内存释放

接入DNS，ip地址支持域名

......

谢谢