C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

扫一扫 升职加薪

班主任:柚子/2690491738

讲师介绍--专业来自专注和实力

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

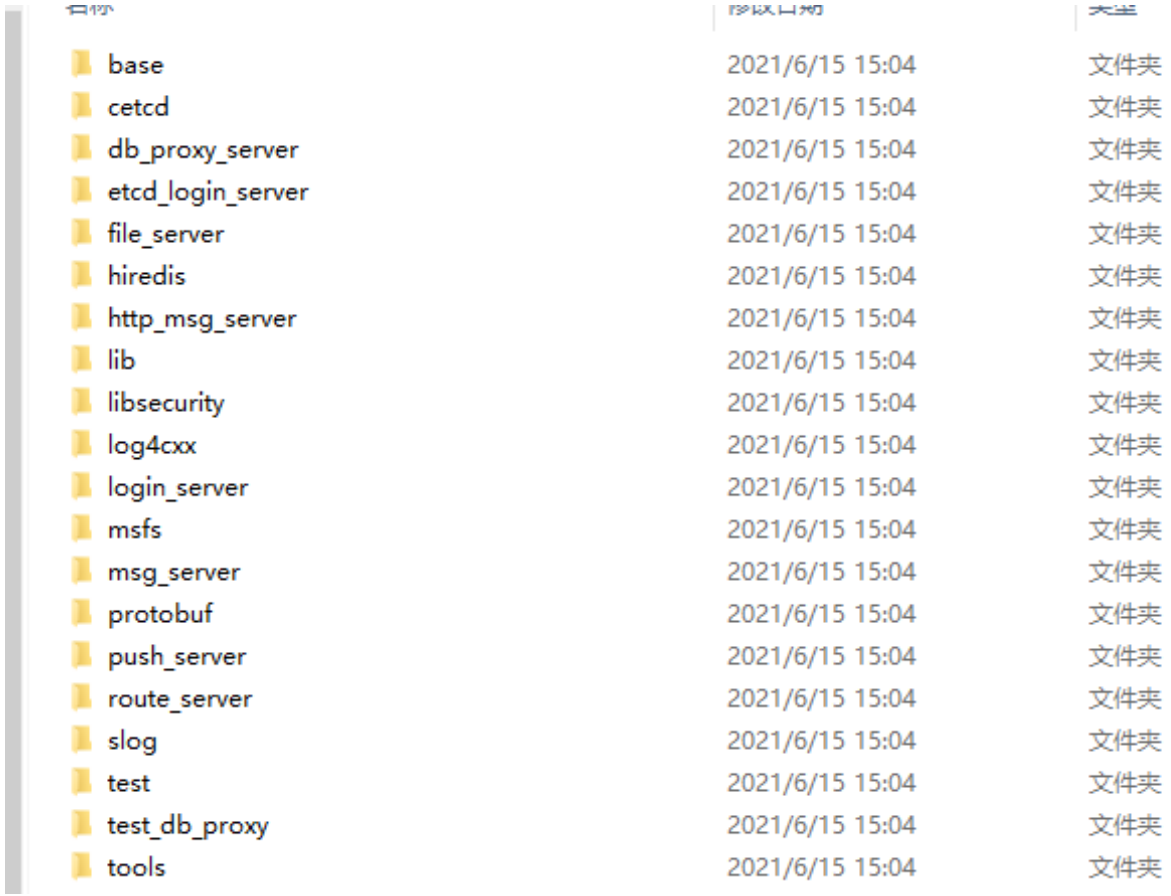
2 IM登录服务器和消息服务器设计

1. 通信协议设计
2. reactor模型
3. login_server分析
4. 数据库设计
5. 登录业务分析
6. msg_server分析

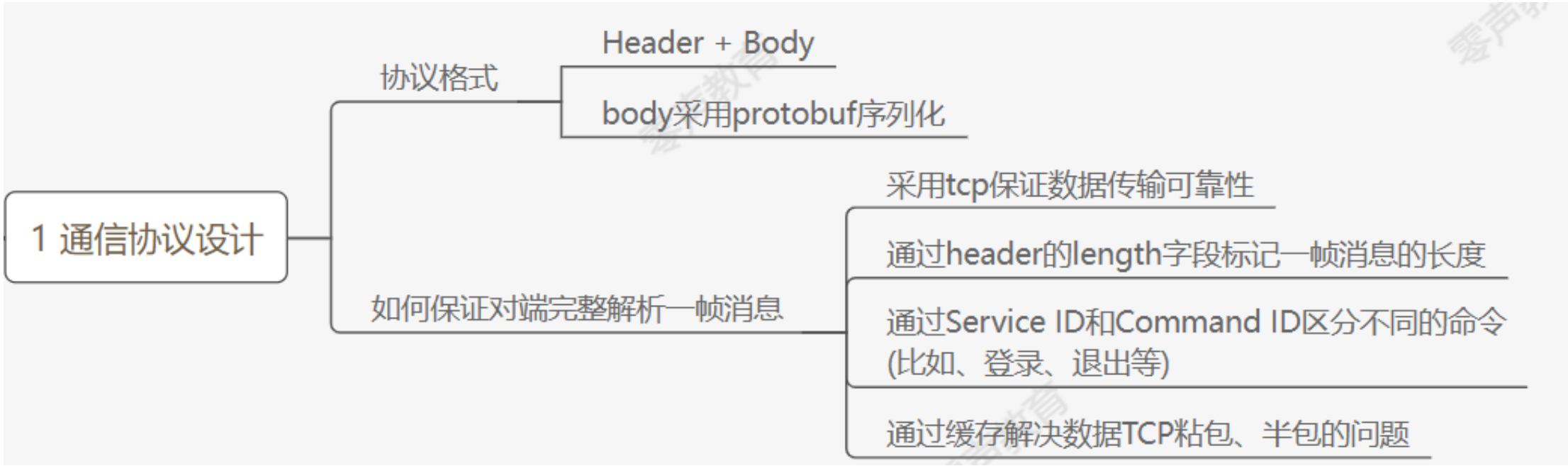
0 源码分布

0voice_im\server\src

- base 基础组件



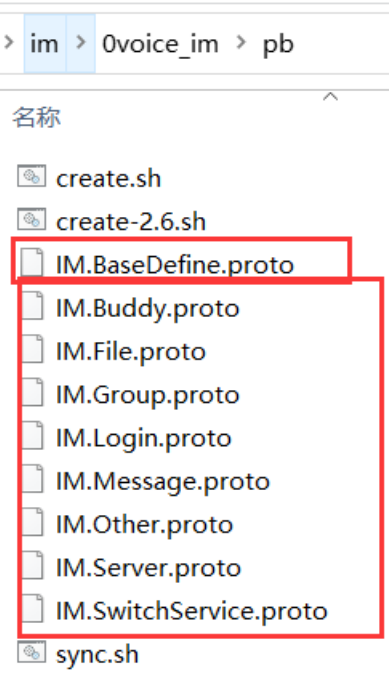
1.1 通信协议设计

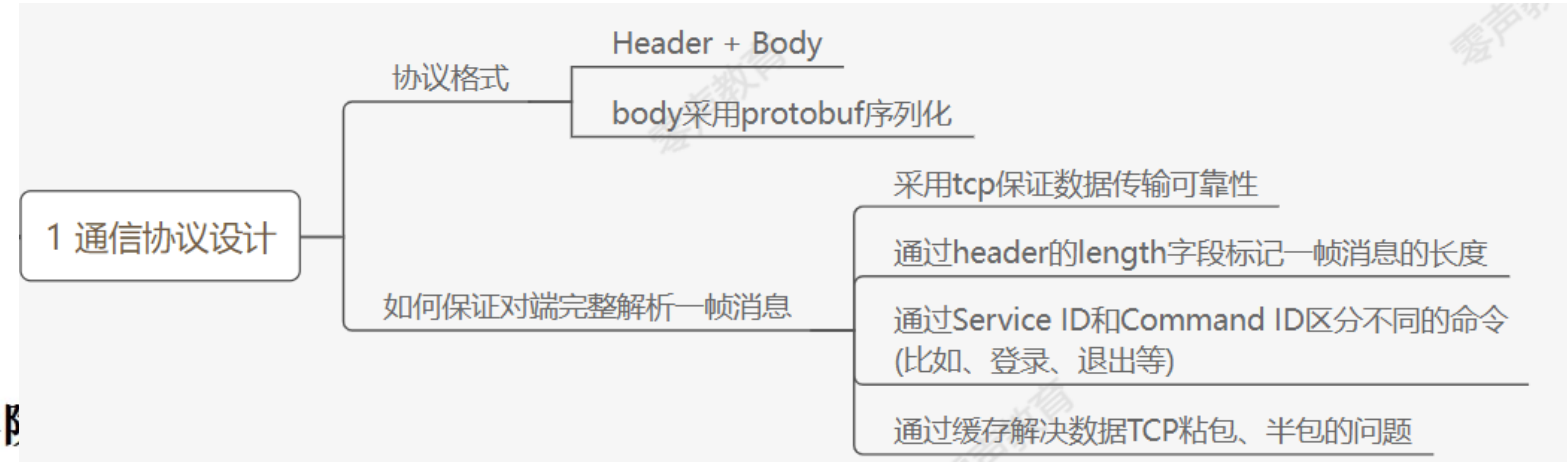

1 通信协议设计
协议格式
Header + Body
body采用protobuf序列化
如何保证对端完整解析一帧消息
采用tcp保证数据传输可靠性
通过header的length字段标记一帧消息的长度
通过Service ID和Command ID区分不同的命令(比如、登录、退出等)
通过缓存解决数据TCP粘包、半包的问题

1.1 通信协议设计–格式

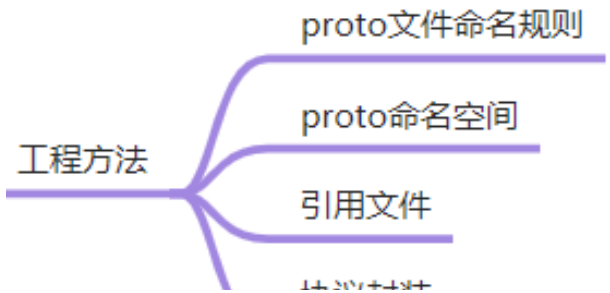
Header(length 50)+body Header(length 100)+body

字段	类型	长度（字节）	说明
length	unsigned int	4	整个包的长度包括 协议头 + BODY
version	unsigned short	2	通信协议的版本号
appid	unsigned short	2	对外SDK提供服务时，用来识别不同的客户
service_id	unsigned short	2	对应命令的分组类比，比如login和msg是不同分组
command_id	unsigned short	2	分组里面的子命令，比如login和login response
seq_num	unsigned short	2	包序号
reserve	unsigned short	2	预留字节
body	unsigned char[]	n	具体的协议数据

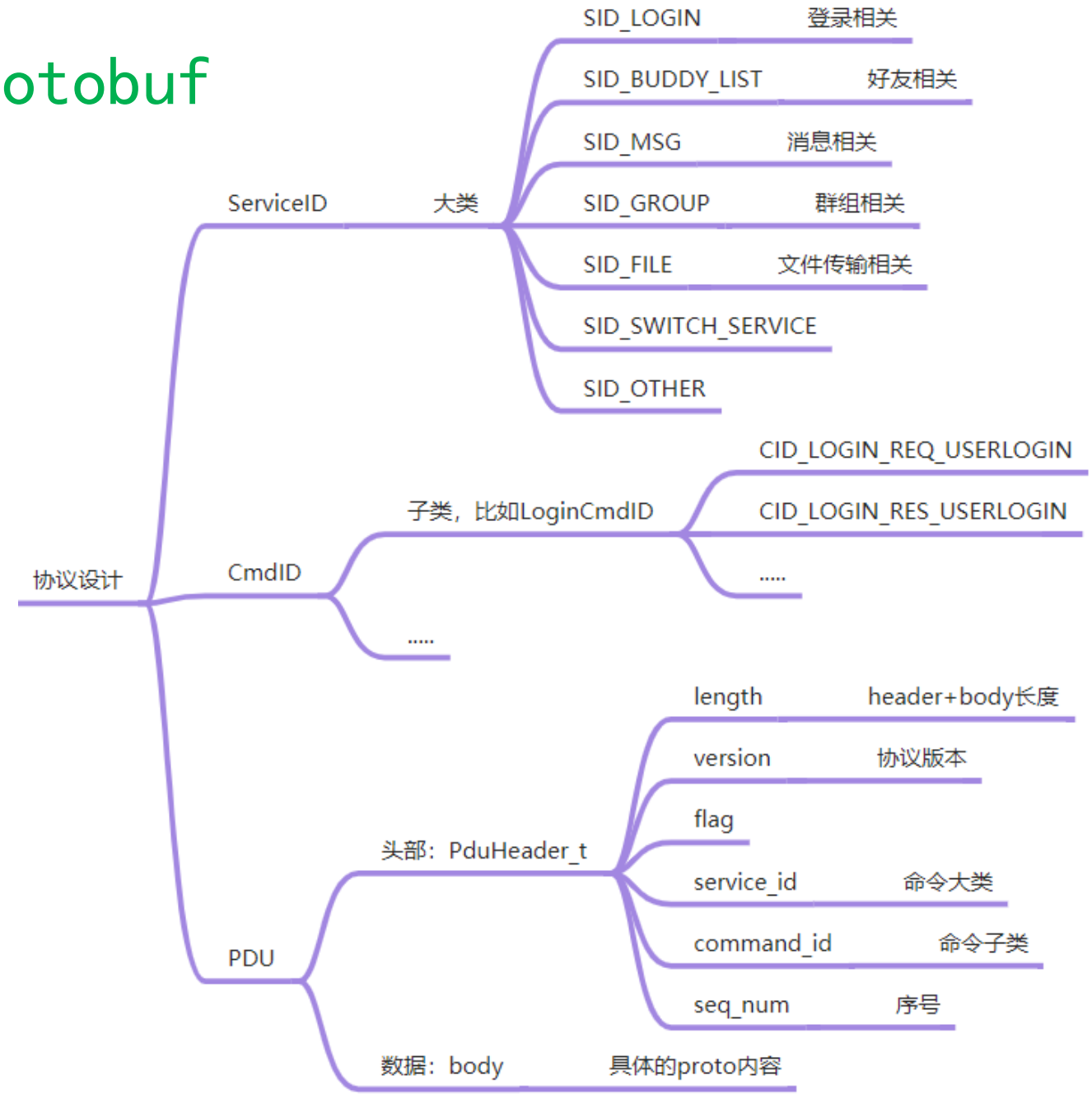




1.2 通信协议设计-protobuf



create.sh	2
IM.BaseDefine.proto	2
IM.Buddy.proto	2
IM.File.proto	2
IM.Group.proto	2
IM.Login.proto	2
IM.Message.proto	2
IM.Other.proto	2
IM.Server.proto	2
IM.SwitchService.proto	2
sync.sh	2



1.3 通信协议设计–结构体

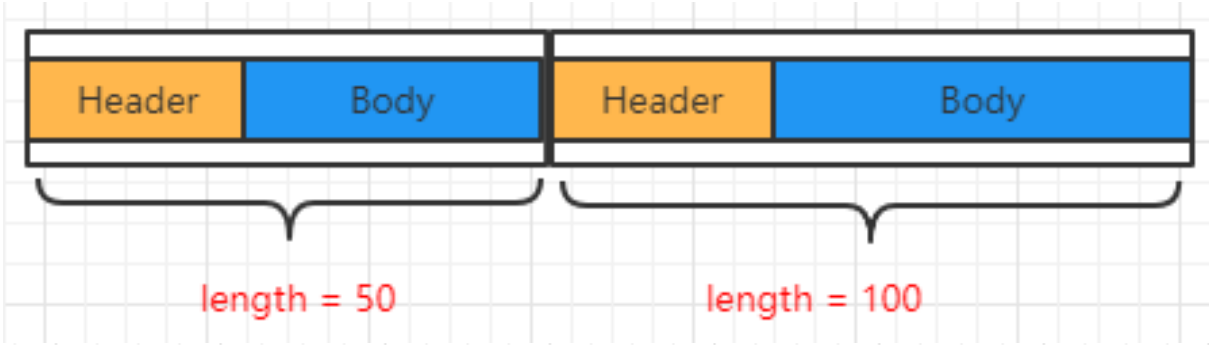
```
typedef struct {
    uint32_t      length;            // the whole pdu length 代表这个包
    uint16_t      version;     // pdu version number
    uint16_t      flag;          // not used
    uint16_t      service_id;        //
    uint16_t      command_id;   //
    uint16_t      seq_num;    // 包序号
    uint16_t   reversed;   // 保留
} PduHeader_t;
```

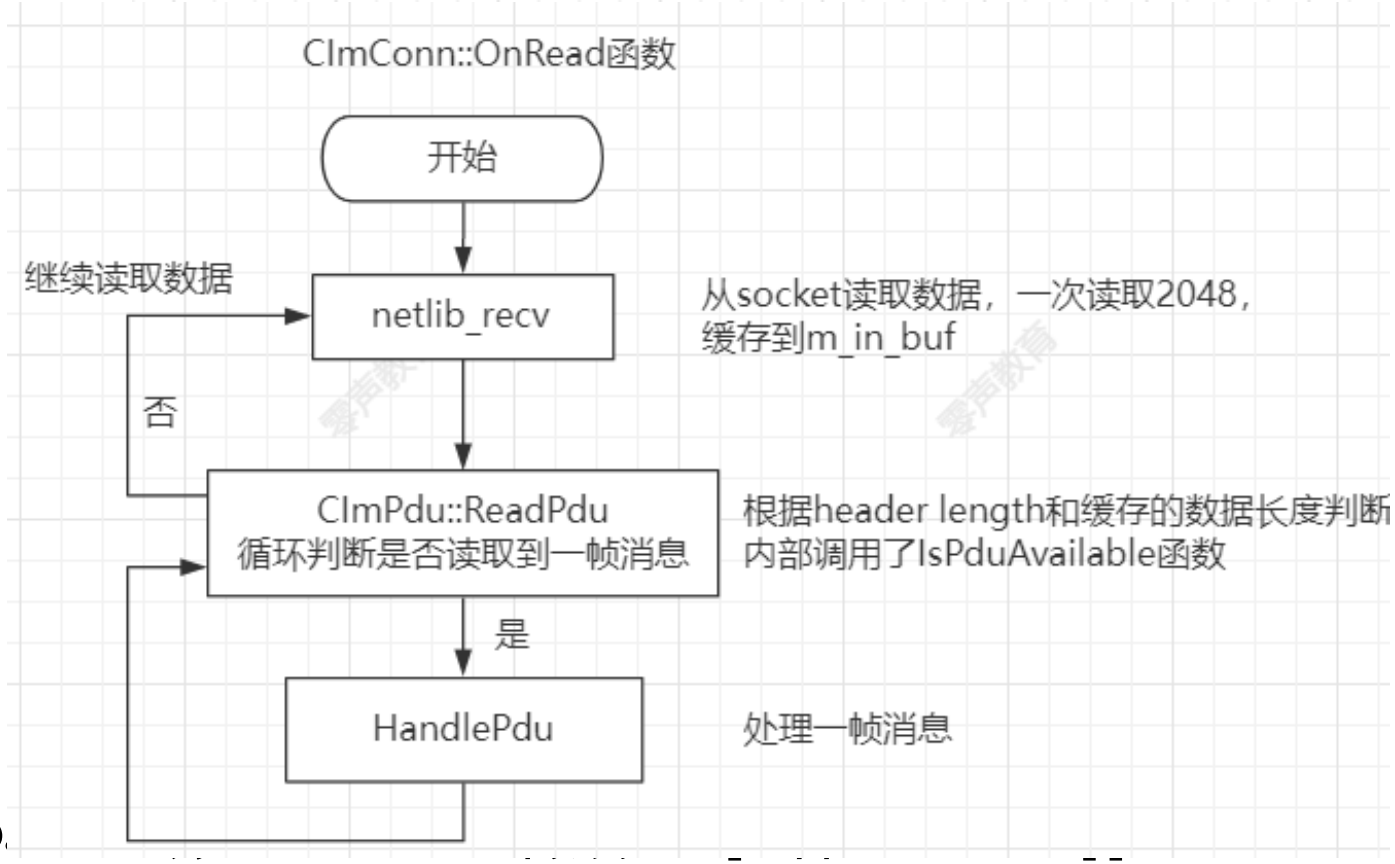
Header + body

1.4 通信协议设计–包完整性判断1

150字节
Length=50;

1. Socket TCP（可靠性）
2. 协议设计的边界处理 header + body，通过header的length字段（4字节）解决
3. 通过Service ID和Command ID区分不同的命令
4. 通过缓存解决数据TCP粘包、半包的问题





1.4 通信协议设计–包完整性判断2

CImPdu::ReadPdu

```
CImPdu* CImPdu::ReadPdu(uchar_t *buf, uint32_t len)
{
    uint32_t pdu_len = 0;
    if (!IsPduAvailable(buf, len, pdu_len))
        return NULL;

    uint16_t service_id = CByteStream::ReadUint16(buf + 8);
    uint16_t command_id = CByteStream::ReadUint16(buf + 10);
    CImPdu* pPdu = NULL;

    pPdu = new CImPdu();
    //pPdu->_SetIncomingLen(pdu_len);
    //pPdu->_SetIncomingBuf(buf);
    pPdu->Write(buf, pdu_len);
    pPdu->ReadPduHeader(buf, IM_PDU_HEADER_LEN);

    return pPdu;
}
```

判断缓存数据是否够一个包

构建一个包

```
bool CImPdu::IsPduAvailable(uchar_t* buf, uint32_t len, uint32_t& pdu_len)
{
    if (len < IM_PDU_HEADER_LEN)
        return false;

    pdu_len = CByteStream::ReadUint32(buf);
    if (pdu_len > len)
    {
        //log("pdu_len=%d, len=%d\n", pdu_len, len);
        return false;
    }

    if(0 == pdu_len)
    {
        throw CPduException(1, "pdu_len is 0");
    }

    return true;
}
```

CImConn::OnRead

```
for (;;)
{
    uint32_t free_buf_len = m_in_buf.GetAllocSize() - m_in_buf.GetWriteOffset();
    if (free_buf_len < READ_BUF_SIZE)
        m_in_buf.Extend(READ_BUF_SIZE);
    log_debug("handle = %u, netlib_recv into, time = %u\n", m_handle, get_tick_count());
    int ret = netlib_recv(m_handle, m_in_buf.GetBuffer() + m_in_buf.GetWriteOffset(), READ_BUF_
    if (ret <= 0)
        break;

    m_recv_bytes += ret;
    m_in_buf.IncWriteOffset(ret);

    m_last_recv_tick = get_tick_count();
}
CImPdu* pPdu = NULL;
try
{
    while ( ( pPdu = CImPdu::ReadPdu(m_in_buf.GetBuffer(), m_in_buf.GetWriteOffset()) ) )
    {
        uint32_t pdu_len = pPdu->GetLength();
        log_debug("handle = %u, pdu_len into = %u\n",m_handle, pdu_len);
        HandlePdu(pPdu);

        m_in_buf.Read(NULL, pdu_len);
        delete pPdu;
        pPdu = NULL;
        ++g_recv_pkt_cnt;
    }
```

将数据读取到缓存

更新缓存数据的位置

读取到完整包

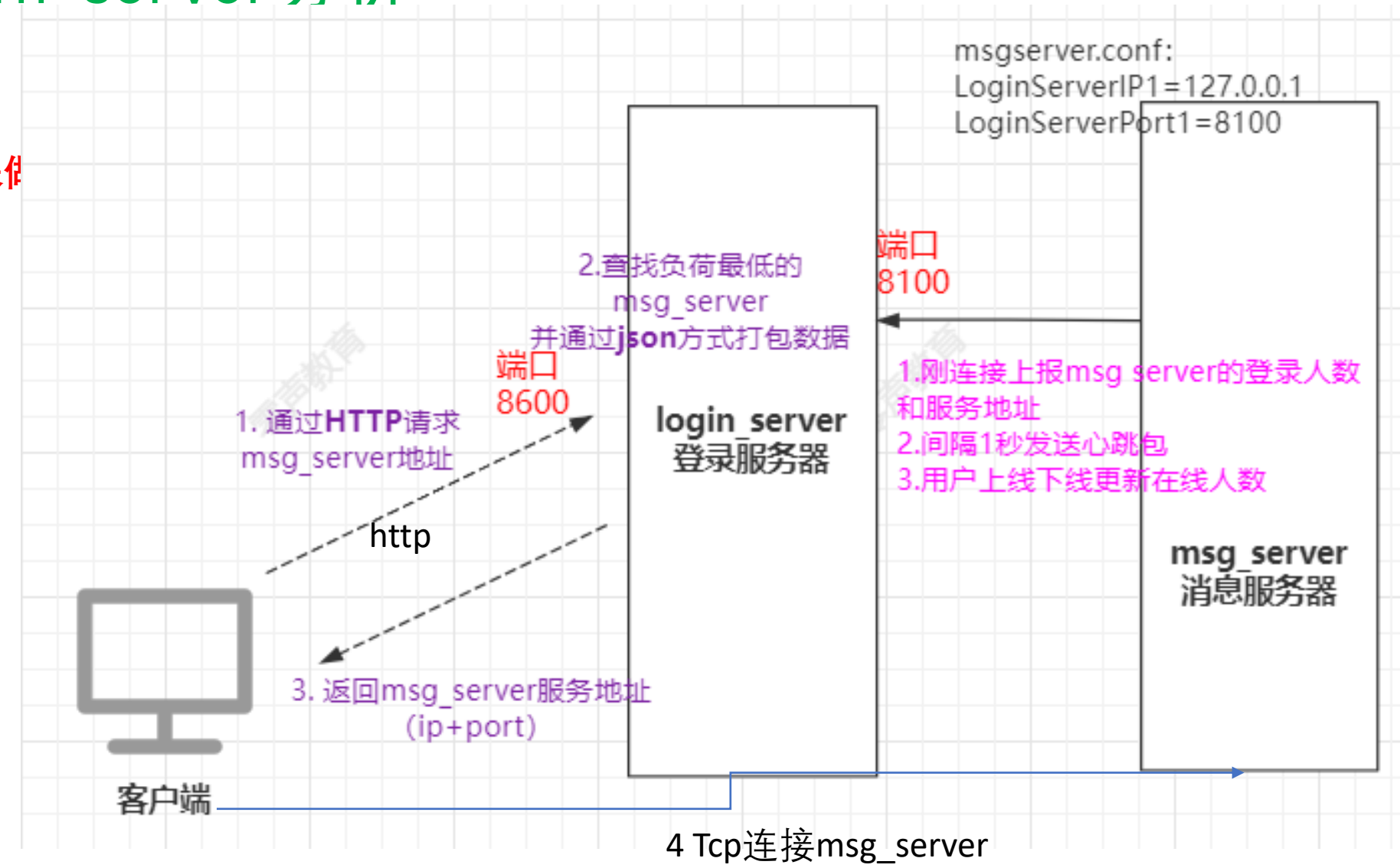
处理包

将包的数据出队列

2 login server分析

login_server

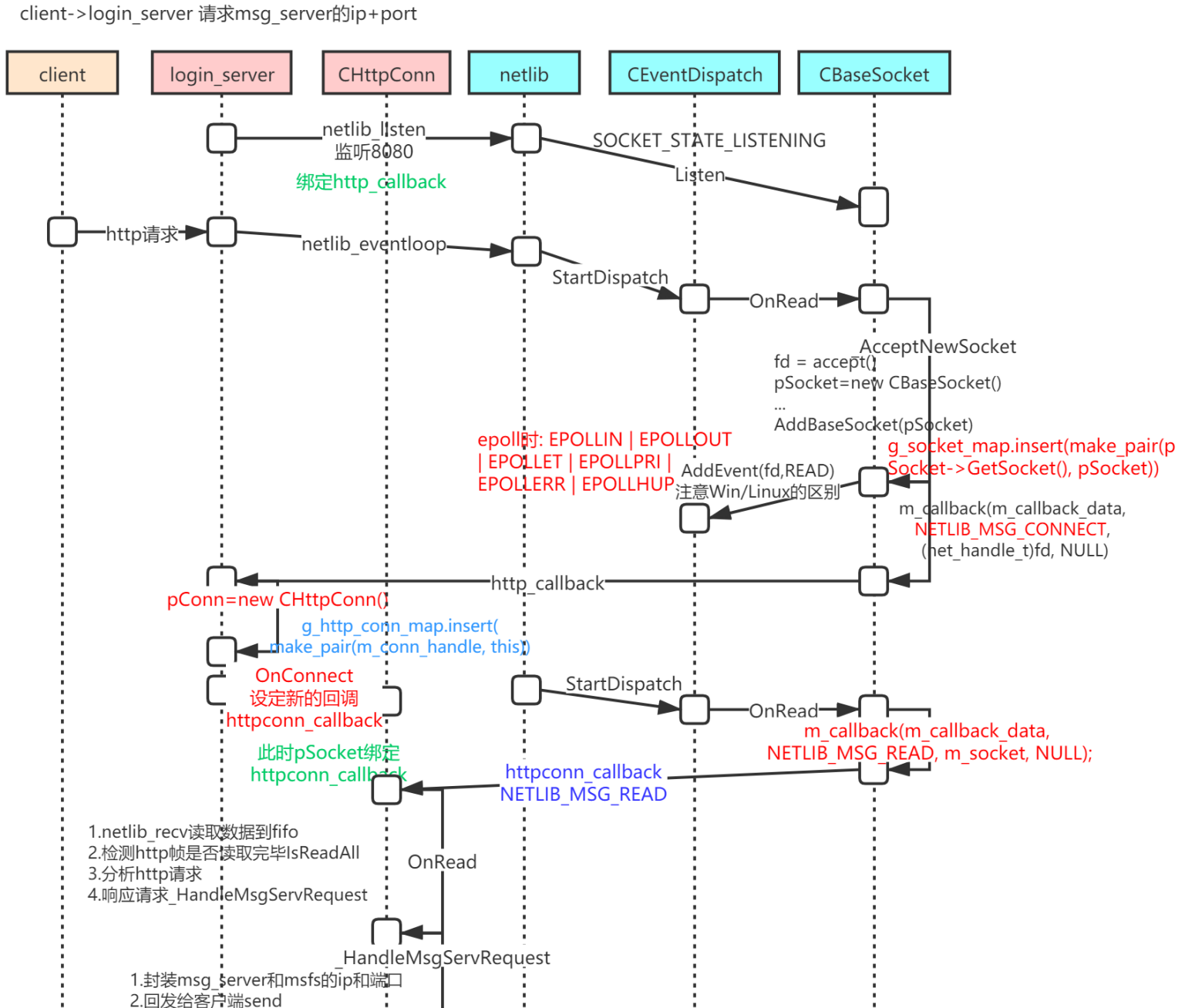
这个服务是用来做负载均衡的



2 login_server

login_server

这个服务是用来做负载均衡的



3 reactor模型-理念1

Reactor 模式是处理并发 I/O 比较常见的一种模式，用于同步 I/O，中心思想是将所有要处理的 I/O 事件注册到一个中心 I/O 多路复用器上，同时主线程 / 进程阻塞在多路复用器上；

一旦有 I/O 事件到来或是准备就绪 (文件描述符或 socket 可读、写)，多路复用器返回并将事先注册的相应 I/O 事件分发到对应的处理器中。

Reactor 模型有三个重要的组件：

- 多路复用器：由操作系统提供，在 linux 上一般是 select, poll, epoll 等系统调用。
- 事件分发器：将多路复用器中返回的就绪事件分到对应的处理函数中。
- 事件处理器：负责处理特定事件的处理函数。

具体流程如下：

1. 注册读就绪事件和相应的事件处理器；
2. 事件分离器等待事件；
3. 事件到来，激活分离器，分离器调用事件对应的处理器；
4. 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

3 reactor模型-理念2

CBaseSocket：管理socket io，作为client或者server都需要实例化一个CBaseSocket
CEventDispatch：是reactor的触发器，epoll相关的函数都在此调用。
netlib：是对外提供了调用的api，它封装了CEventDispatch。

3.1 reactor模型-CBaseSocket

CBaseSocket：管理socket io，作为client或者server都需要实例化一个CBaseSocket

hash_map<net_handle_t, CBaseSocket*> SocketMap; 管理所有的连接，以fd为key，
CBaseSocket对象为value，对应处理：

1. void AddBaseSocket(CBaseSocket* pSocket)　　增
2. void RemoveBaseSocket(CBaseSocket* pSocket)　　删
3. CBaseSocket* FindBaseSocket(net_handle_t fd)　　查

socket自定义状态：

SOCKET_STATE_IDLE： CBaseSocket()
SOCKET_STATE_LISTENING： Listen()　　作为server的时候用
SOCKET_STATE_CONNECTING : Connect()　　作为client的时候用
SOCKET_STATE_CONNECTED: OnWrite()　　作为client的时候用
SOCKET_STATE_CLOSING: OnClose()

```
enum
{
    NETLIB_MSG_CONNECT = 1,
    NETLIB_MSG_CONFIRM,
    NETLIB_MSG_READ,
    NETLIB_MSG_WRITE,
    NETLIB_MSG_CLOSE,
    NETLIB_MSG_TIMER,
    NETLIB_MSG_LOOP
};
```

fd事件

3.2 reactor模型-CEventDispatch

CEventDispatch是reactor的触发器，epoll相关的函数都在此调用。

Timer相关
AddTimer：加入定时事件
RemoveTimer：删除定时事件
_CheckTimer：检测定时事件 _私有函数

Loop相关
AddLoop：加入循环事件
_CheckLoop：检测循环事件

epoll相关
AddEvent：　　添加fd事件
RemoveEvent：删除fd事件
StartDispatch：进入reactor主循环
StopDispatch：停止reactor主循环

3.3 reactor模型-api netlib

netlib是对外的api，封装了CEventDispatch。

fd相关

- netlib_listen：监听
- netlib_connect：连接
- netlib_send：发送
- netlib_recv：接收
- netlib_close：关闭
- netlib_option：参数设置
  - NETLIB_OPT_SET_CALLBACK 设置回调函数

```
void CHttpConn::OnConnect(net_handle_t handle)
{
    log("OnConnect, handle=%d\n", handle);
    m_sock_handle = handle;
    m_state = CONN_STATE_CONNECTED;
    g_http_conn_map.insert(make_pair(m_conn_handle, this));

    netlib_option(handle, NETLIB_OPT_SET_CALLBACK, (void*)httpconn_callback);
    netlib_option(handle, NETLIB_OPT_SET_CALLBACK_DATA, reinterpret_cast<void *>(m_conn_handle) );
    netlib_option(handle, NETLIB_OPT_GET_REMOTE_IP, (void*)&m_peer_ip);
}
```

定时器相关

- netlib_register_timer：注册定时器
- netlib_delete_timer：删除定时器

循环相关

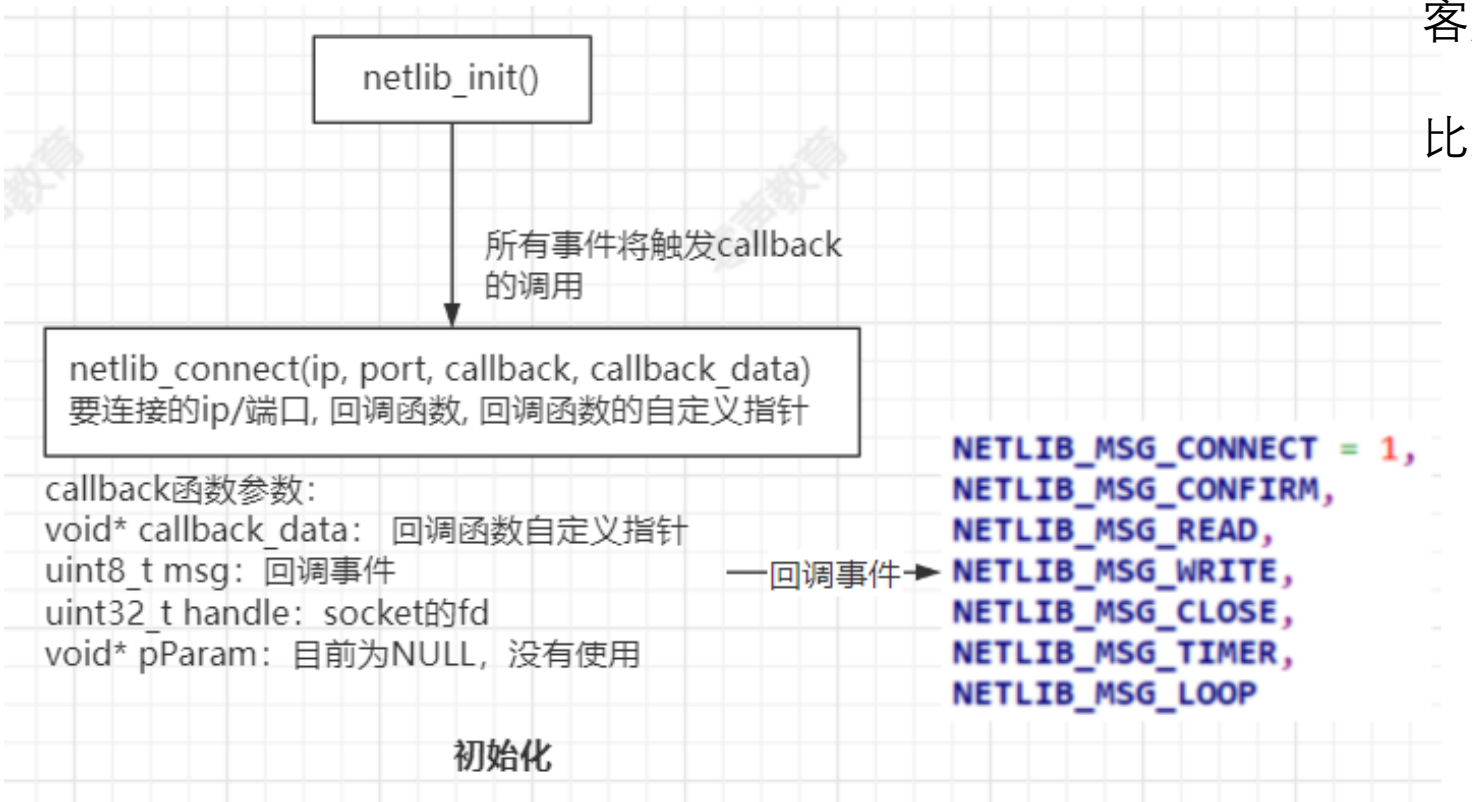
- netlib_add_loop：可以加入需要循环处理的事务到reactor
- netlib_eventloop：进入reactor主循环

3.4 reactor模型-作为客户端1

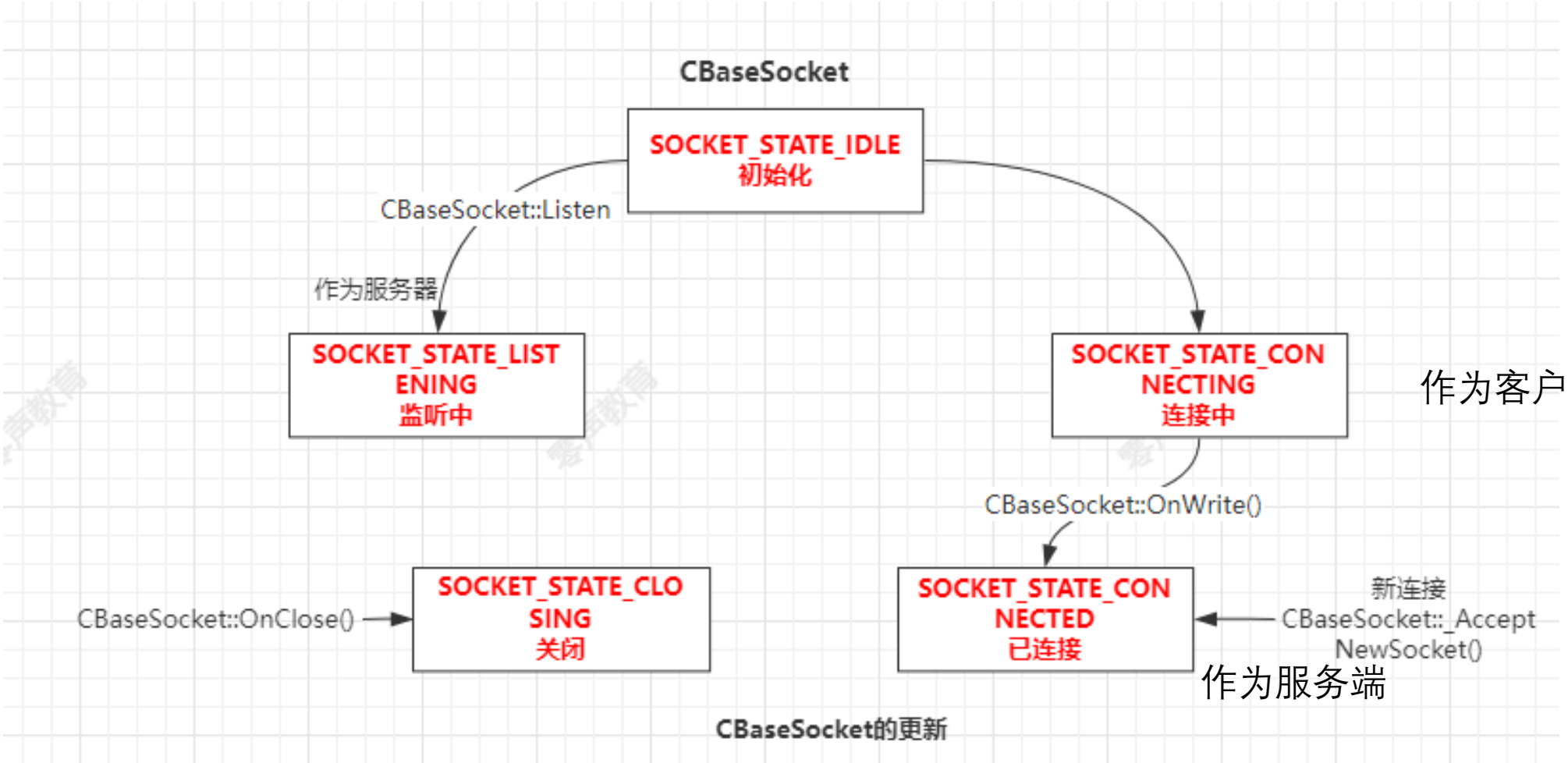
作为客户端，去连接服务器。**比如msg_server如何和login_server保持通信**

一个reactor可以多个
客户端 连接 服务器业务。

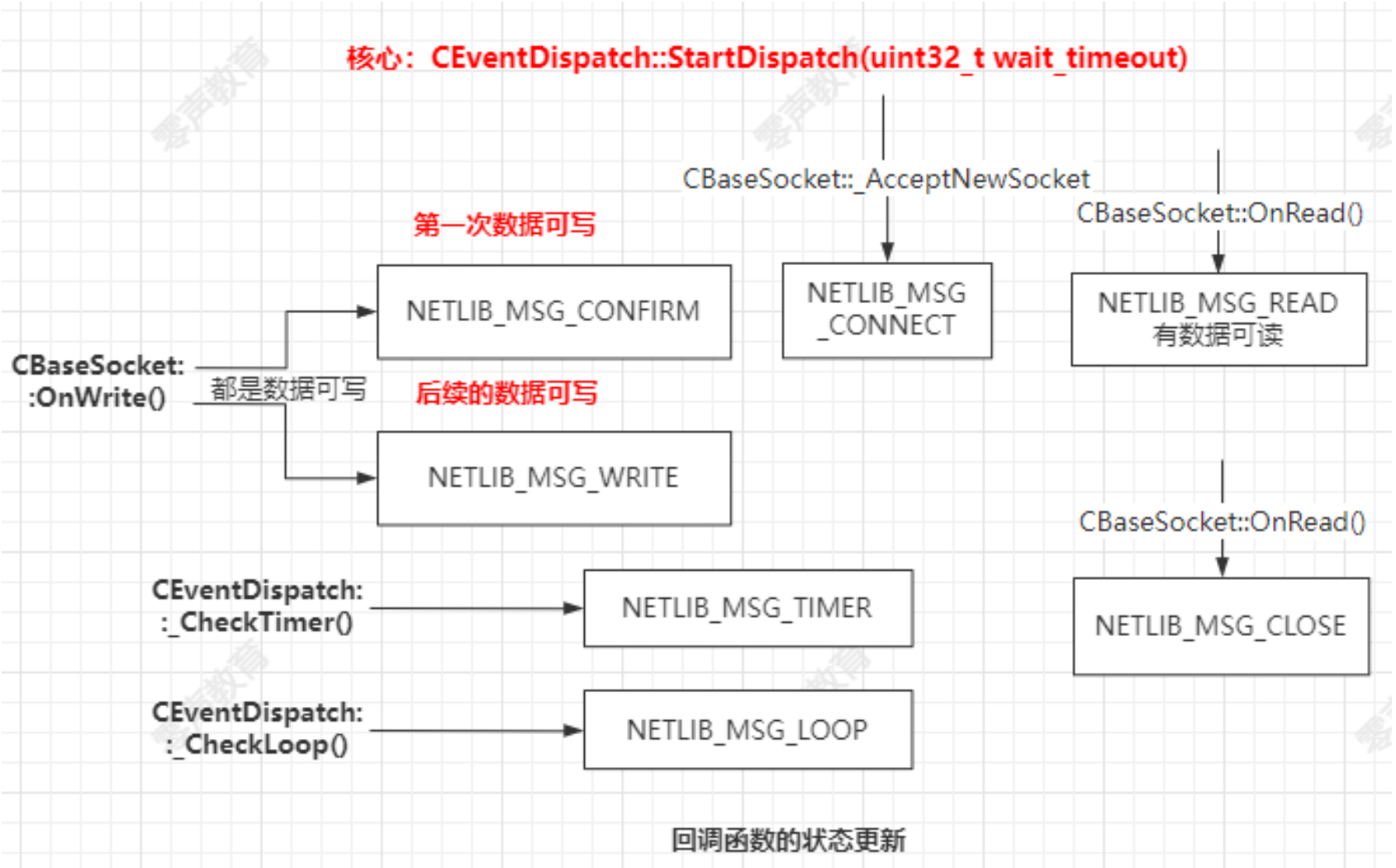
比如msg_server
- login
- file
- route



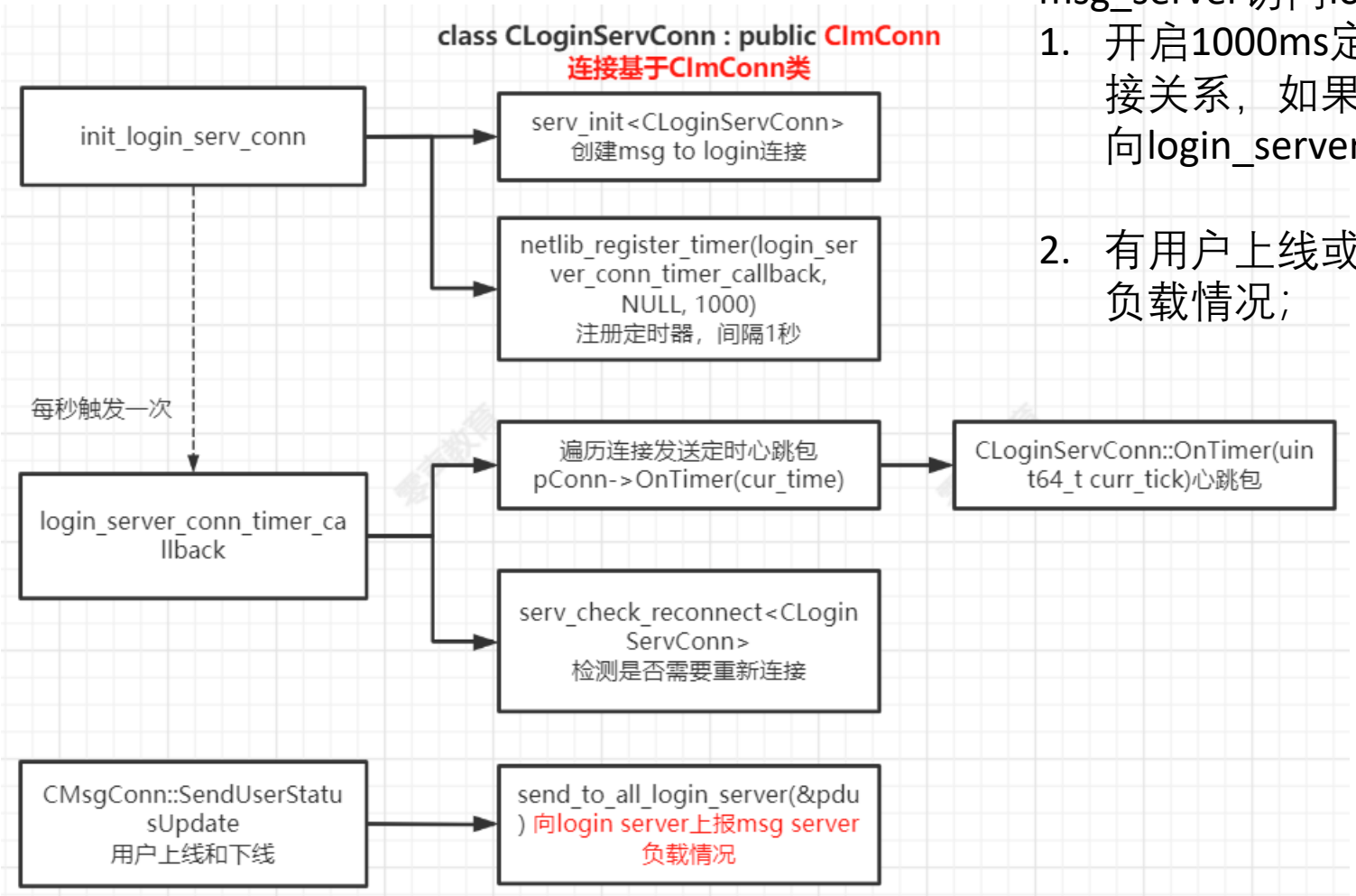
3.4 reactor模型-作为客户端2



3.4 reactor模型-作为客户端3



3.4 reactor模型-作为客户端4-应用范例



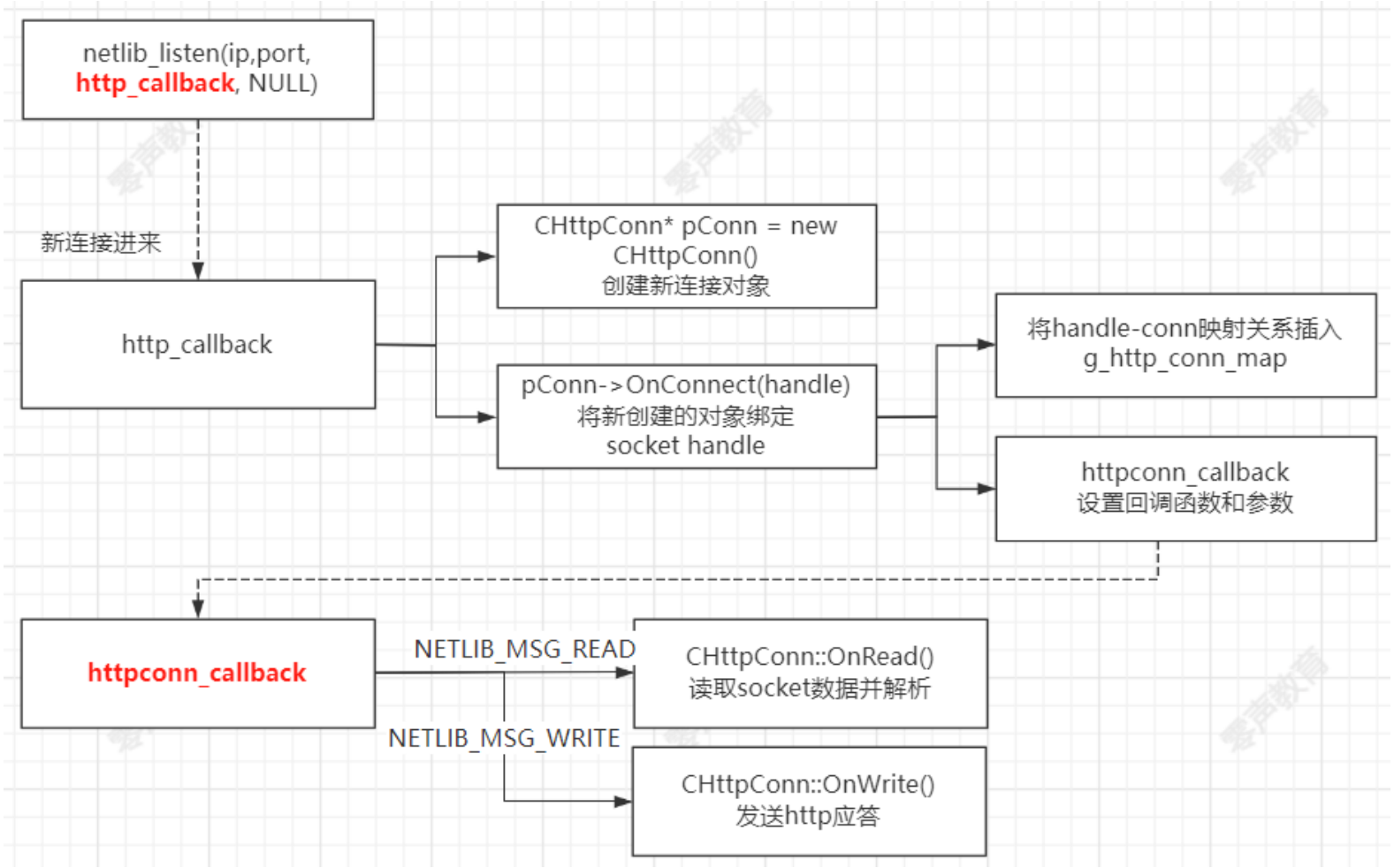
msg_server访问login_server：

1. 开启1000ms定时器，定时检测login_server之间的连接关系，如果没有连接重连，如果已经连接则定时向login_server发送心跳包；

2. 有用户上线或者下线时向msg_server上报目前其的负载情况；

3.4 reactor模型-作为服务端

作为服务端，如何响应客户端的连接并处理。



3.5 reactor模型-http服务和IM协议的区别

Listen 设置回调
netlib_option： NETLIB_OPT_SET_CALLBACK 设置回调函数

login_server.cpp

```
netlib_listen(http_listen_ip_list.GetItem(i), http_port, http_callback, NULL);
```

```
49: void http_callback(void* callback_data, uint8_t msg, uint32_t handle, void* pParam)
50: {
51:     if (msg == NETLIB_MSG_CONNECT)
52:     {
53:         // 这里是不是觉得很奇怪,为什么new了对象却没有释放?
54:         // 实际上对象在被Close时使用delete this的方式释放自己
55:         CHttpConn* pConn = new CHttpConn();
56:         pConn->OnConnect(handle);
57:     }
58:     else
59:     {
60:         log_error("!!!error msg: %d ", msg);
61:     }
62: }
```

```
void CHttpConn::OnConnect(net_handle_t handle)
{
    log("OnConnect, handle=%d\n", handle);
    m_sock_handle = handle;
    m_state = CONN_STATE_CONNECTED;
    g_http_conn_map.insert(make_pair(m_conn_handle, this));

    netlib_option(handle, NETLIB_OPT_SET_CALLBACK, (void*)httpconn_callback);
    netlib_option(handle, NETLIB_OPT_SET_CALLBACK_DATA, reinterpret_cast<void *>(m_conn_handle));
    netlib_option(handle, NETLIB_OPT_GET_REMOTE_IP, (void*)&m_peer_ip);
}
```

```
netlib_listen(msg_server_listen_ip_list.GetItem(i), msg_server_port, msg_serv_callback, NULL);
```

```
void msg_serv_callback(void* callback_data, uint8_t msg, uint32_t handle, void* pP
{
    log("msg_server come in");

    if (msg == NETLIB_MSG_CONNECT)
    {
        CLoginConn* pConn = new CLoginConn();
        pConn->OnConnect2(handle, LOGIN_CONN_TYPE_MSG_SERV);
    }
    else
    {
        log_error("!!!error msg: %d ", msg);
    }
}
```

```
void CLoginConn::OnConnect2(net_handle_t handle, int conn_type)
{
    m_handle = handle;
    m_conn_type = conn_type;
    ConnMap_t* conn_map = &g_msg_serv_conn_map;
    if (conn_type == LOGIN_CONN_TYPE_CLIENT) {
        conn_map = &g_client_conn_map;                // 初始化map
    }else {
        conn_map->insert(make_pair(handle, this));          pdu
    }
    netlib_option(handle, NETLIB_OPT_SET_CALLBACK, (void*)imconn_callback);
    netlib_option(handle, NETLIB_OPT_SET_CALLBACK_DATA, (void*)conn_map);
}
```

4 数据库设计


数据库设计
IMAdmin 后台管理员表
IMAudio 存储语音地址
IMDepart 存储部门信息
IMDiscovery 发现配置表
IMGroup 群组表
IMGroupMember 群成员表
IMGroupMessage_(x) 群消息表,x代表第几张表
IMMessage_(x) 单聊消息表，x代表第几张表
IMRrecentSession 最近联系人(会话)表
IMRelationShip 用户关系表
IMUser 用户表

5.1 登录业务

登录账号验证与状态同步

- 用户名密码校验
- 在线状态设置+路由状态更新
- 旧客户端账号下线
- 通知好友，移动时代，24小时在线

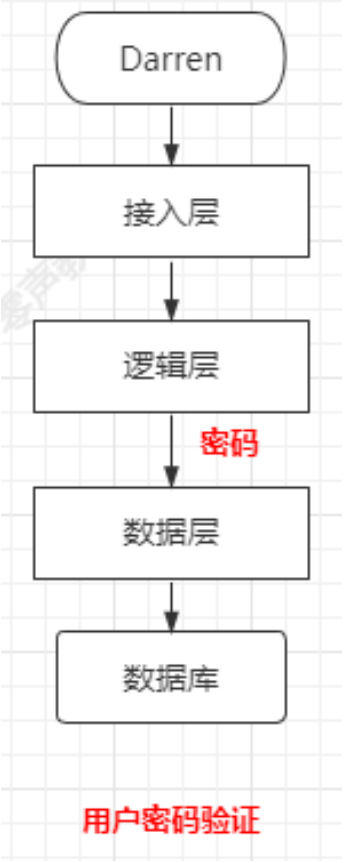
登录数据准备

- 分组信息
- 好友列表
- 群组列表
- 好友信息
- 未读消息
- 其他...

5.1 登录业务-续1

登录账号验证与状态同步

- 用户名密码校验



明文密码不能用

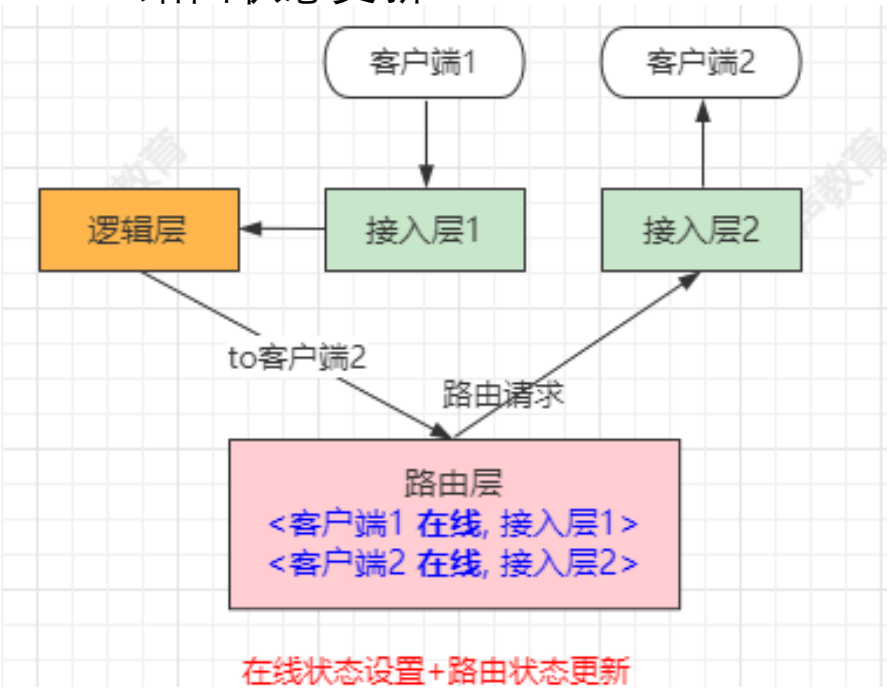
客户端 md5(password)

password_md5
这个密码能不能之间存入数据库

服务端
md5(password_md5+混淆码)

数据库存储:
1. md5(password_md5+混淆码)
2. 混淆码

- 在线状态设置+路由状态更新
  - 客户端状态
  - 服务端状态
  - 路由状态更新



5.1 登录业务-续2 旧客户端账号下线

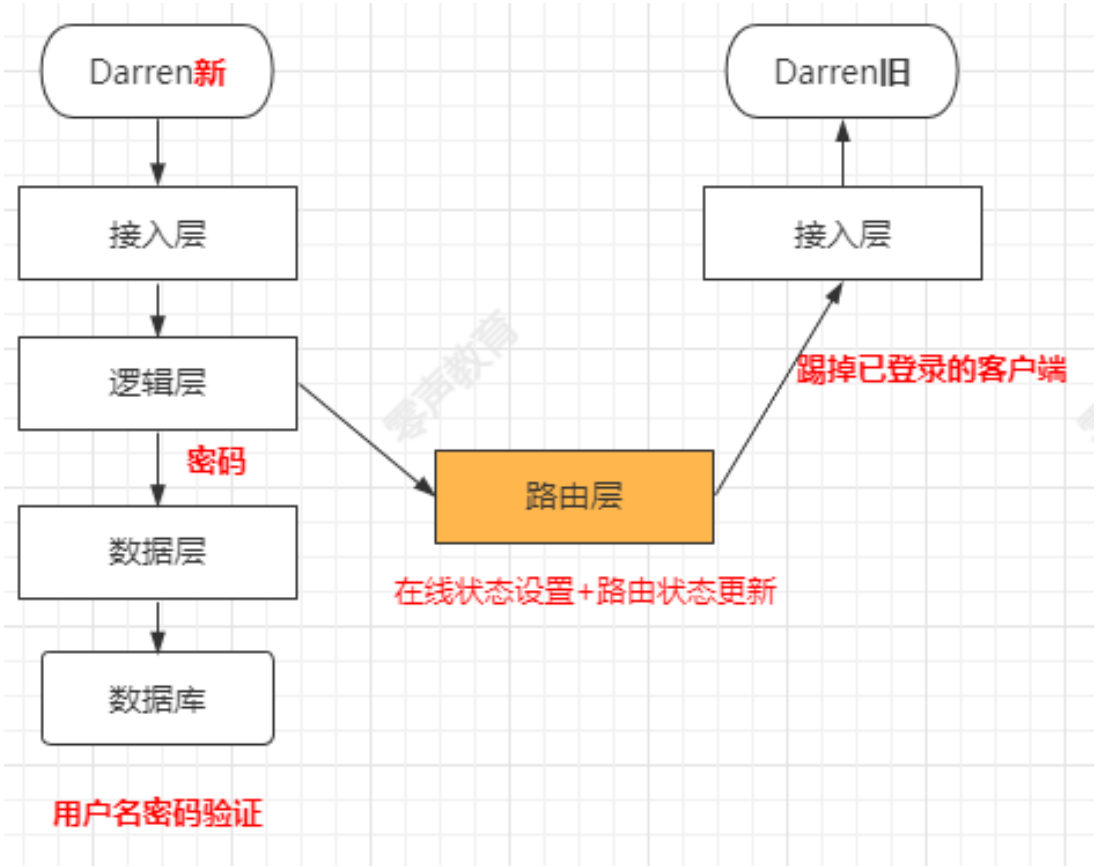
登录账号验证与状态同步

- 如果账号已经登录，则将其强迫下线

1. 办公室pc登录
   （1）手机再登录没有问题
   （2）如果在家里的pc登录，办公室下线

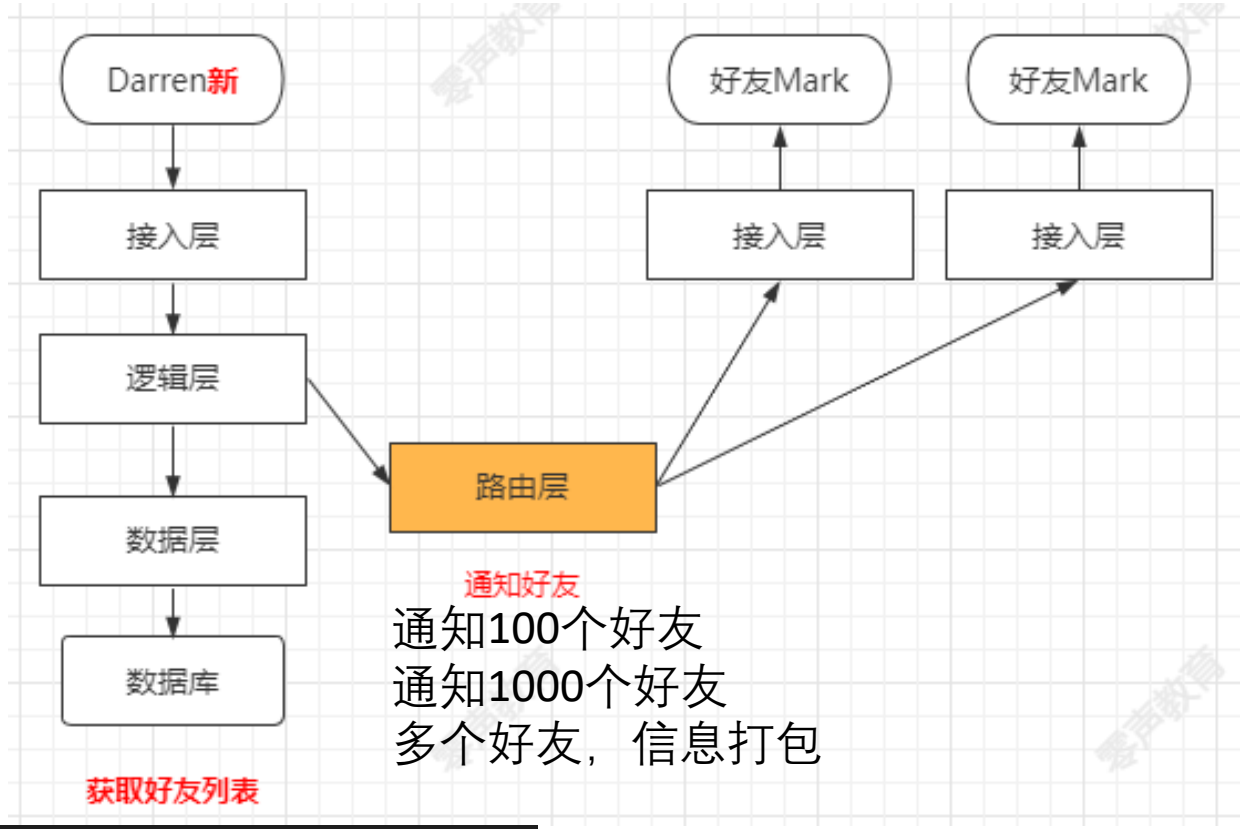
2 手机登录
   （1）换个手机再登录，之前的手机下线
   (2)如果使用pc登录，没有问题，提示pc有登录

根据实际需求提供策略。



5.1 登录业务-续3 通知好友

登录账号验证与状态同步
- 通知好友




```
28 message IMUsersInfoReq{
29     //cmd id:       0x0204
30     required uint32 user_id = 1;
31     repeated uint32 user_id_list = 2;
32     optional bytes attach_data = 20;
33 }
34
```

Darren 1
秋香 2
柚子 3

5.1 登录业务-续4 同步信息

即时通讯 消息重要还是 好友重要?
微信先从QQ导入的好友关系，**可以接收qq的信息**

同步信息
- 好友列表
- 分组列表
- 群组列表
- 未读消息

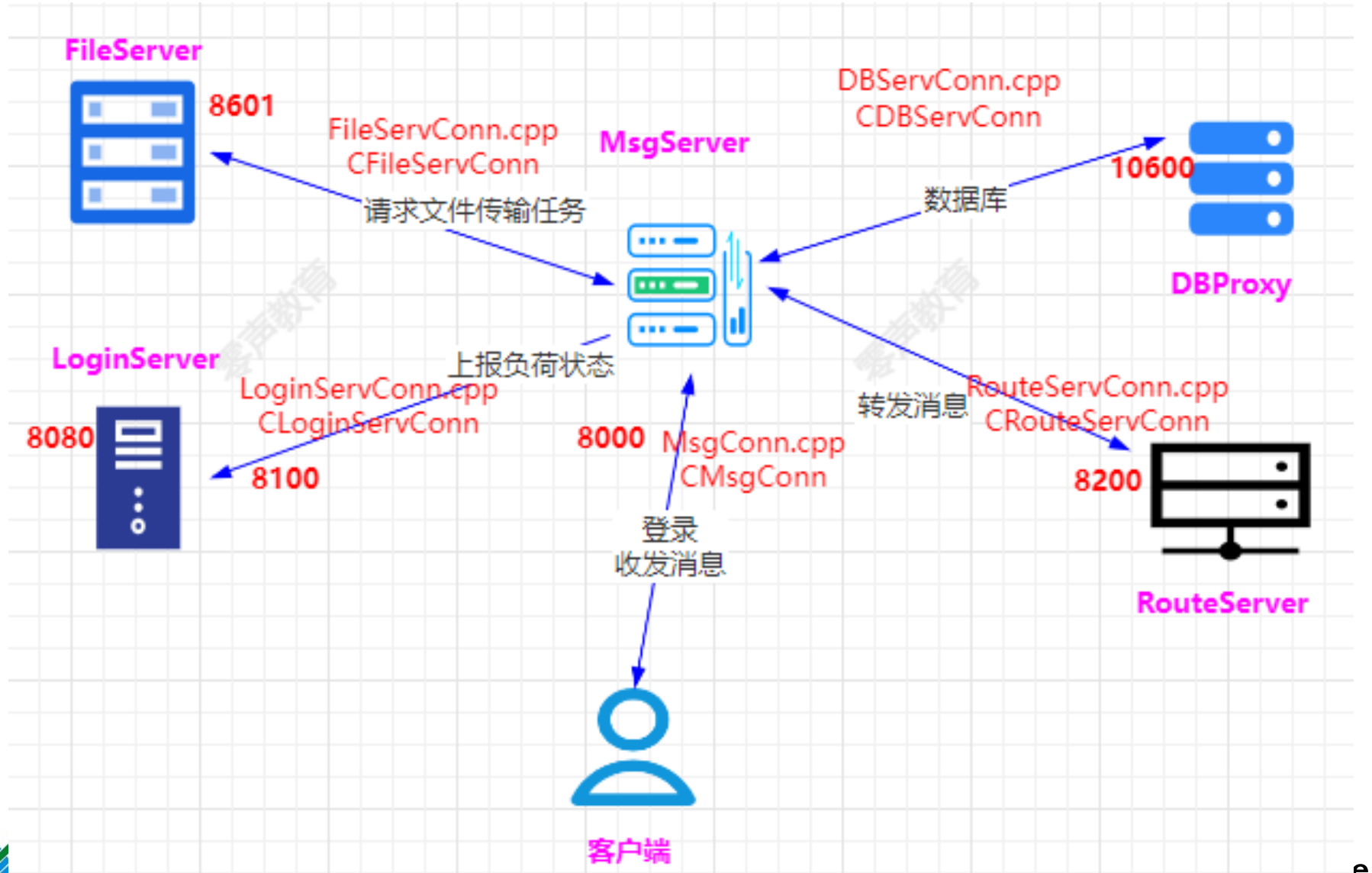
特别是登录微信

特别是换了新手机，登录过程特别慢。

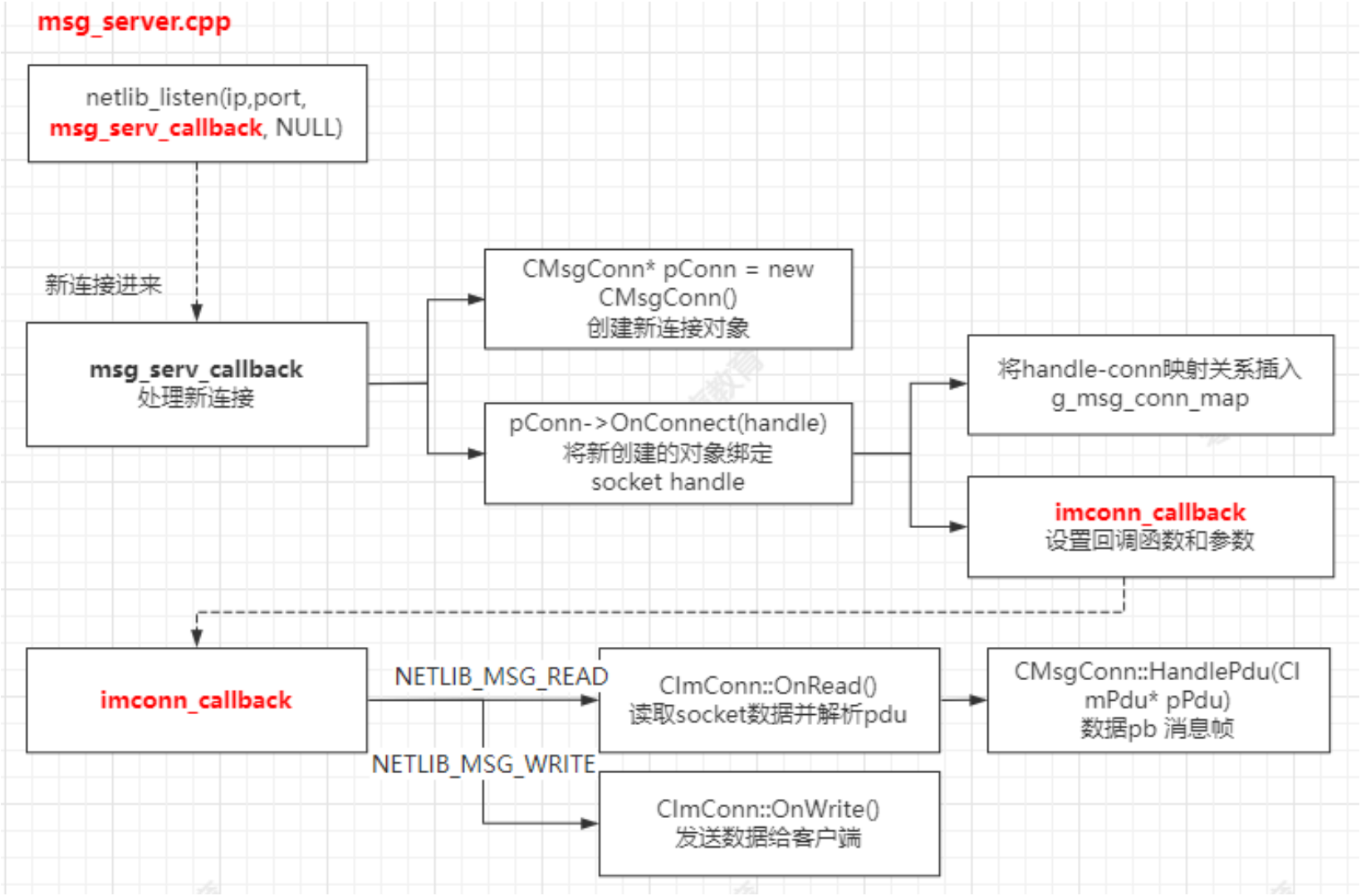
作业：是不是每次登录都需要全部拉取：
- 好友列表　　时间更新节点
- 分组列表
- 未读消息

怎么做到不全部拉取，在dbproxy讲解。

6.0 msg_server对接逻辑



6.0 msg_server对接逻辑

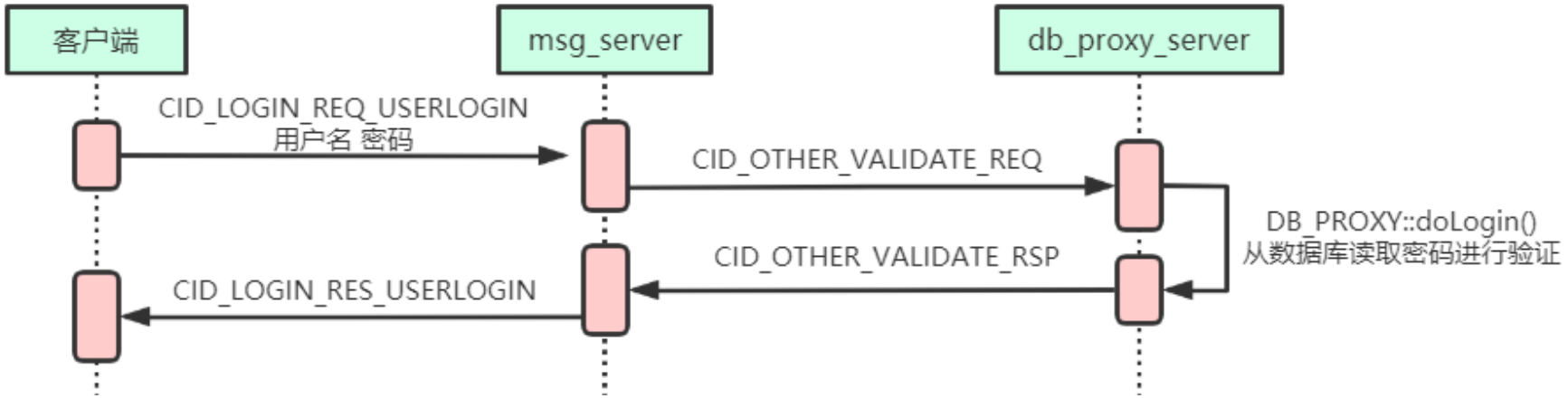


6.1 msg_server登录流程

结合proto文件

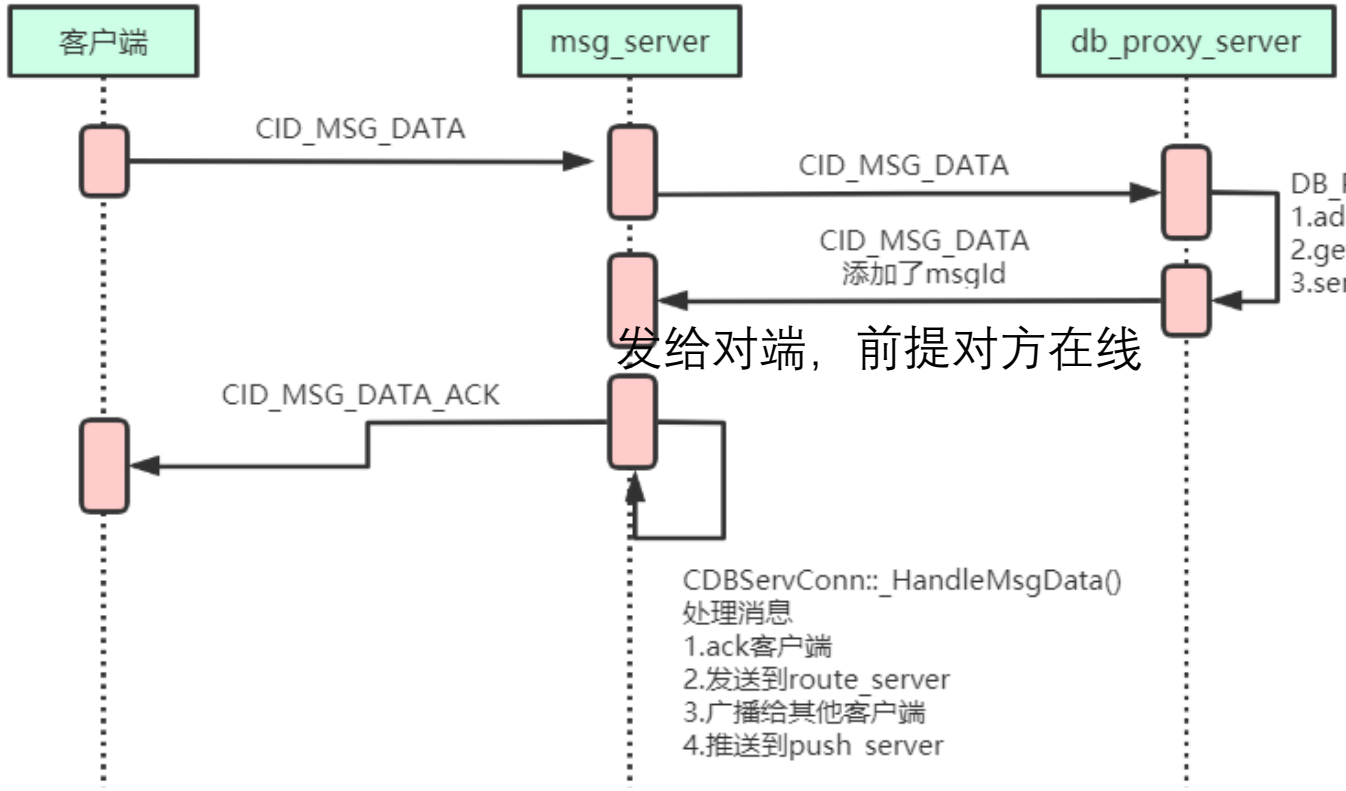
0voice_im › pb

```
message IMLoginReq{
    //cmd id:        0x0103
    required string user_name = 1;
    required string password = 2;
    required IM.BaseDefine.UserStatType online_status =
    required IM.BaseDefine.ClientType client_type = 4;
    optional string client_version = 5;
}
```




```
message IMLoginRes{
    //cmd id:        0x0104
    required uint32 server_time = 1;
    required IM.BaseDefine.ResultType result_code = 2;
    optional string result_string = 3;
    optional IM.BaseDefine.UserStatType online_status = 4;
    optional IM.BaseDefine.UserInfo user_info = 5;
}
```

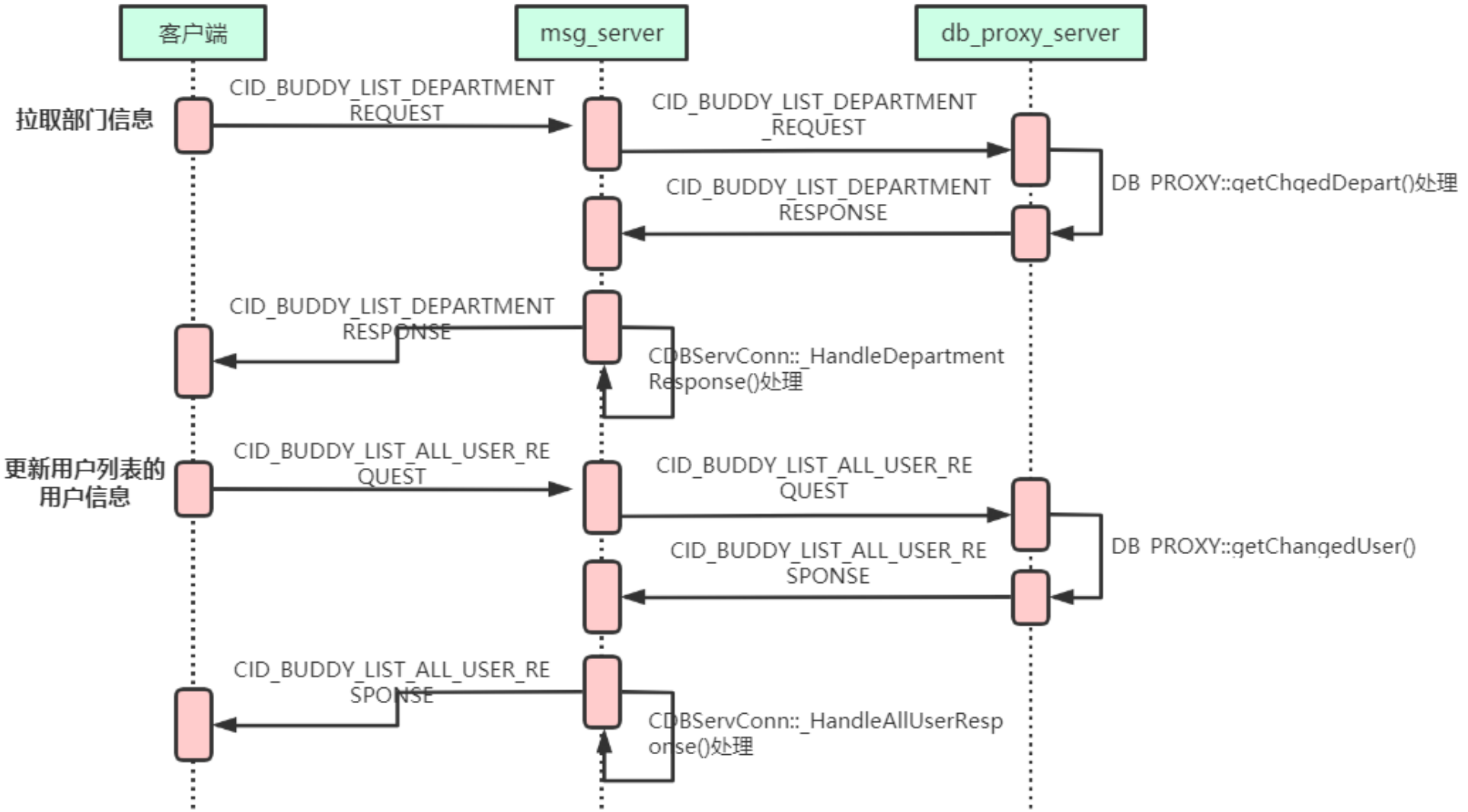
6.2 msg_server发送消息



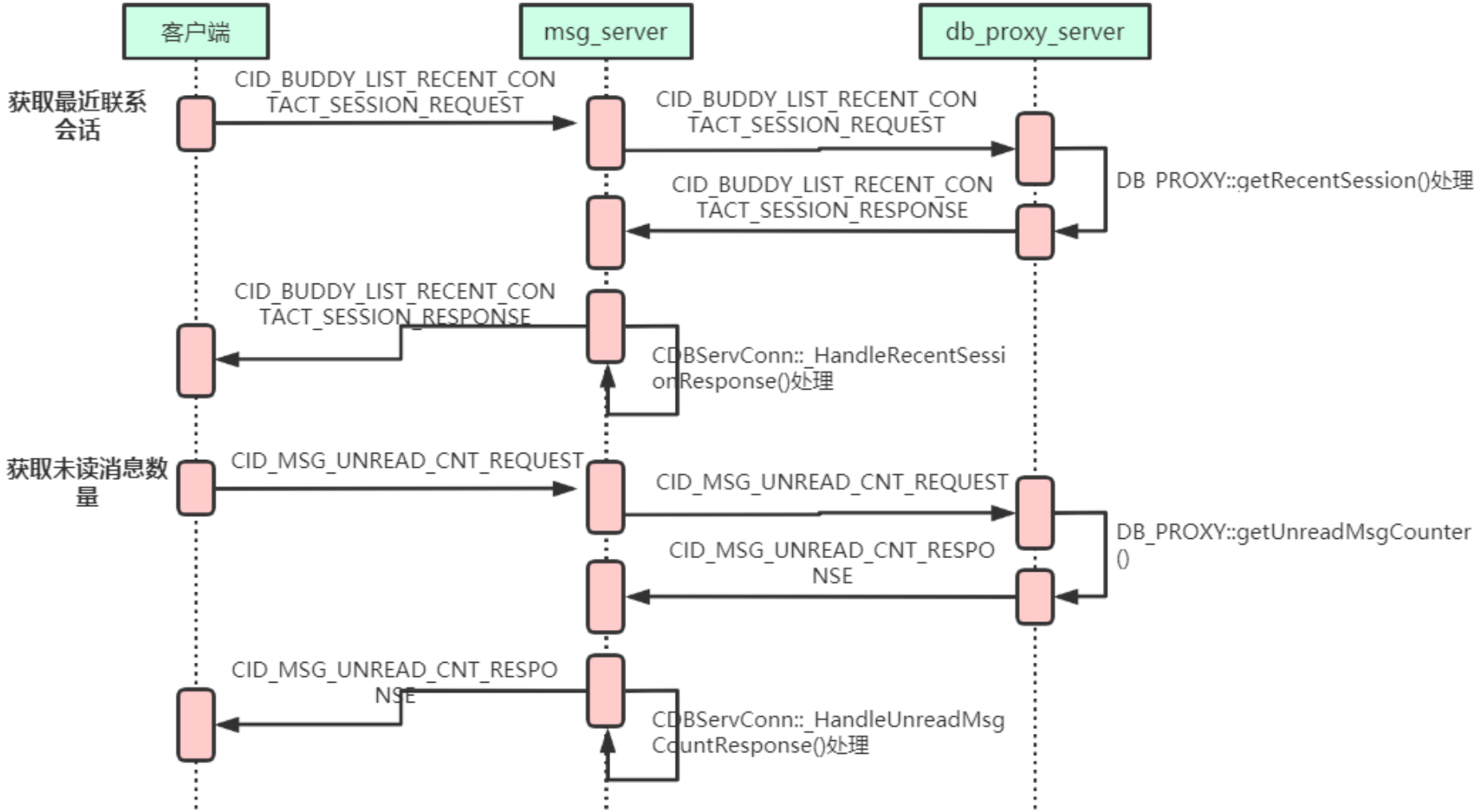

```
7  //service id  0x0003
8  message IMMsgData{
9      //cmd id:        0x0301
10     required uint32 from_user_id = 1;  //消息发送
11     required uint32 to_session_id = 2; //消息接收
12     required uint32 msg_id = 3;
13     required uint32 create_time = 4;
14     required IM.BaseDefine.MsgType msg_type = 5;
15     required bytes msg_data = 6;
16     optional bytes attach_data = 20;
17 }
```

```
19 message IMMsgDataAck{
20     //cmd id:        0x0302
21     required uint32 user_id = 1; //发送此信令的用户id
22     required uint32 session_id = 2;
23     required uint32 msg_id = 3;
24     required IM.BaseDefine.SessionType session_type = 4;
25 }
```

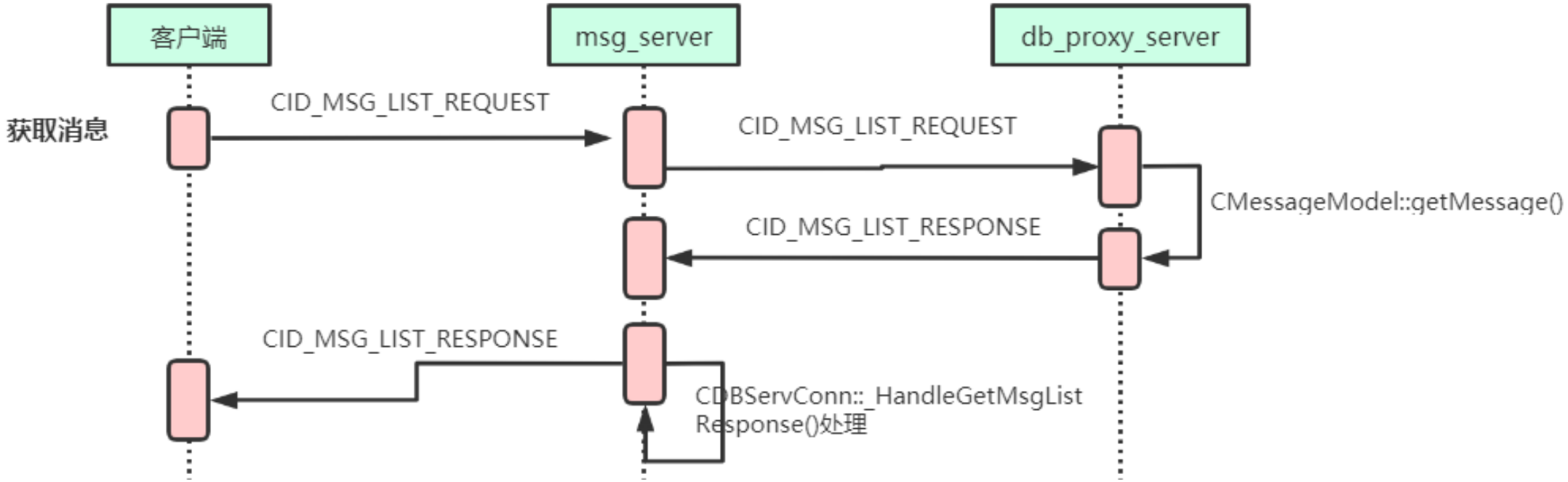
6.3 msg_server登录请求响应过程-更新信息1



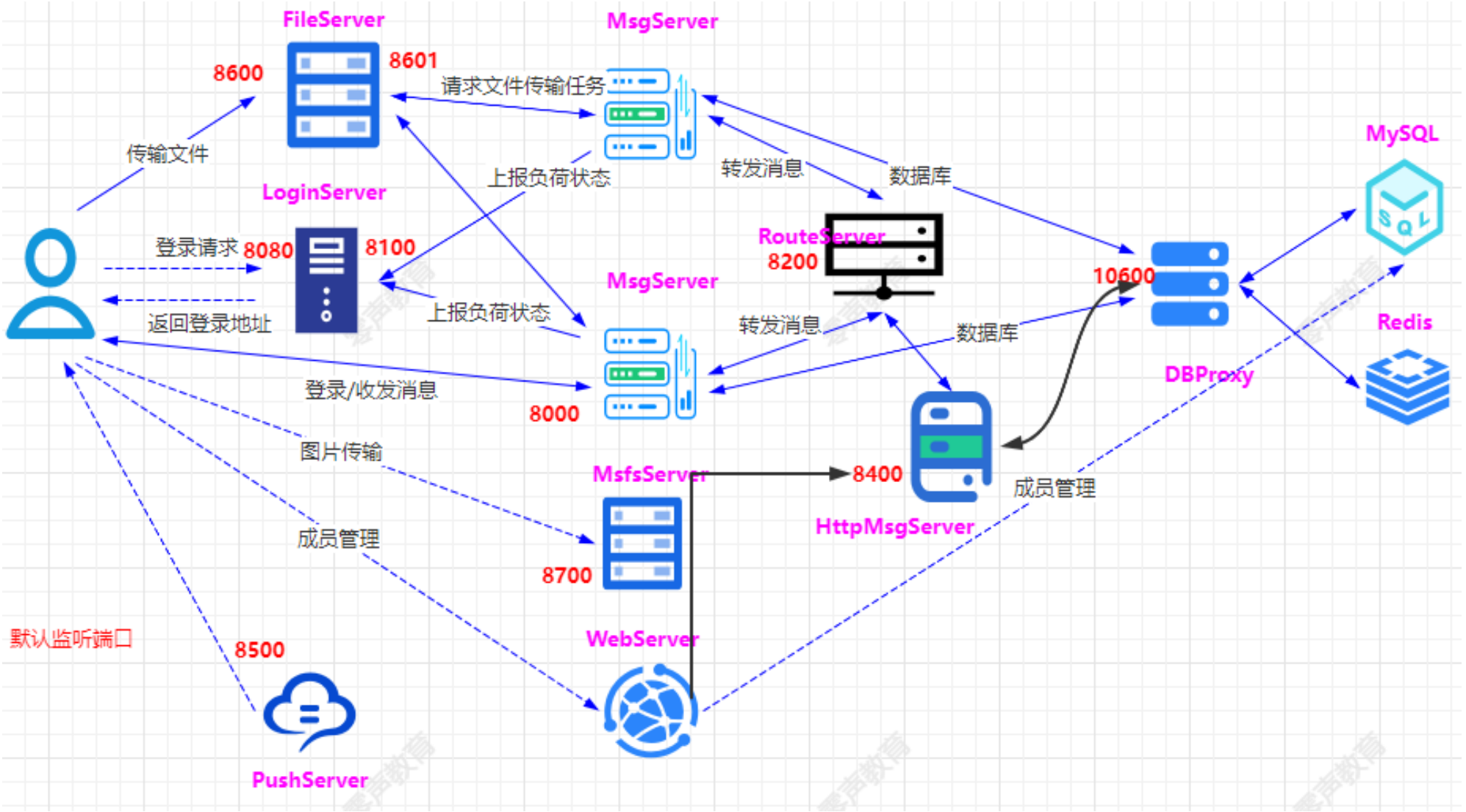
6.3 msg_server登录请求响应过程2



6.3 msg_server登录请求响应过程3



7 框架图


FileServer
MsgServer
8600
8601
请求文件传输任务
传输文件
上报负荷状态
转发消息
数据库
MySQL
LoginServer
登录请求
8080
8100
RouteServer
8200
10600
MsgServer
返回登录地址
上报负荷状态
转发消息
数据库
Redis
登录/收发消息
8000
DBProxy
图片传输
MsfsServer
8400
成员管理
HttpMsgServer
成员管理
8700
默认监听端口
8500
WebServer
PushServer