4 数据库代理服务器设计

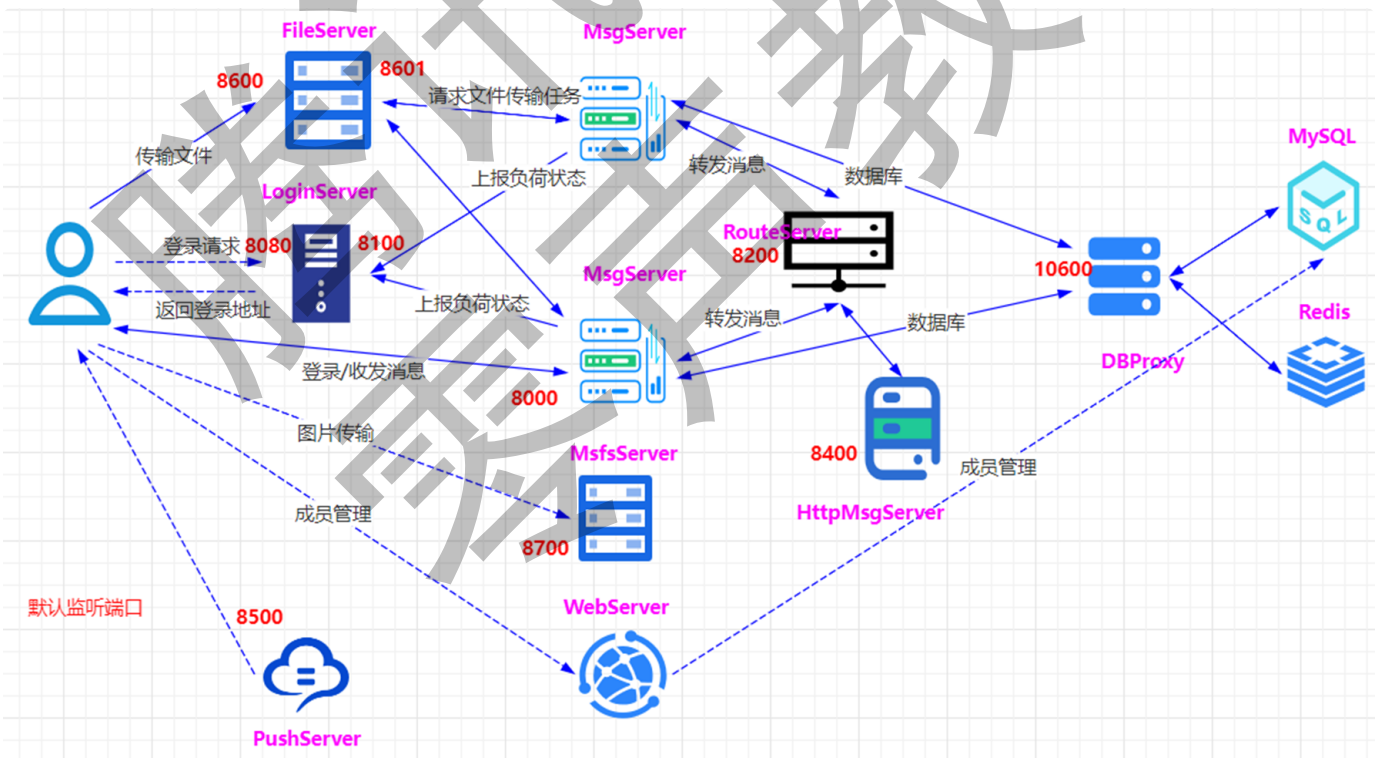
零声学院 https://0voice.ke.qq.com

讲师 Darren老师 QQ326873713

班主任 柚子老师 QQ2690491738

2022年06月25日

db_proxy_server是TeamTalk服务器端最后端的程序，它连接着关系型数据库mysql和nosql内存数据库redis，其位置如下图所示。



1 概述

dbproxyserver.conf文件，DB_SERVER主要分为以下几个部分：

1、TeamTalk_Matser MySQL主数据库

2、TeamTalk_Slave　MySQL从数据库（单机配置时涉及都是同一个数据库）

3、unread 未读信息实例 Redis 数据库

4、group_set 群组设置实例 redis数据库

5、token 实例 redis数据库

6、sync实例　redis数据库 同步功能

7、group_member redis 数据库

每个数据库（不管是MySQL还是Redis）实例都会预先打开于数据库的两个链接，不需要在每次使用的时候再打开，使用结束后释放，节省了数据打开和释放需要的时间和资源，在当前可用连接数不够的情况下再新增一个数据库链接，动态调节DB_Server的负载，同时限定了每个实例的最大可用连接数，由于系统资源是有限的，当业务比较繁忙时不能无限制创建新的连接，避免耗尽系统资源，这种场景下，当没有可用连接的时候，新的业务请求必须等待，等待可用的连接，然后再执行相应的业务操作。

DB_Server采用了多线程，在DB_Server启动的时候预先分配了配置文件中指定的线程数，用来处理具体的数据库请求，当一个请求到达DB_Server时，DB_Server将该请求封装成DB相关的额任务类，然后随机加入到预先启动的线程的任务列表中，有线程回调函数不停执行具体的任务请求，这就是整个DB_Serve的设计思路。

配置文件加载

MySQL数据库初始化

Redis缓存初始化

Redis持久化恢复。

2 main函数流程分析

```
int main()
{
    //1. 初始化redis连接 CacheManager::getInstance()，初始化的时CacheManager::Init()去读取配置文件dbproxyserver.conf

    //2. 初始化mysql连接 CDBManager::getInstance()，初始化的时CDBManager::Init()去读取配置文件dbproxyserver.conf
```

// 3. 初始化CAudioModel(语言消息)、CGroupMessageModel（群消息）、CGroupModel（群成员）、CMessageModel（单聊消息）、CSessionModel（会话）、CRelationModel（会话关系ID）、CUserModel（用户ID）、CFileModel（在线/离线文件）

//4. 初始化线程池和任务队列 init_proxy_conn， db_proxy_server作为server，没收到一个处理事务都封装成task交给线程池进行处理（g_thread_pool.AddTask(pTask)），但回发数据时还是在epoll所在的loop进行（proxy_loop_callback）。

//4.1 signal(SIGTERM, sig_handler); 信号设置，让db_proxy_server能够平滑退出

//5. 启动从mysql同步数据到redis工作 total_user_updated last_update_group更新时间，主要是同步群组成员doSyncGroupChat（作为一个线程运行）

//6. 在端口10600上启动侦听，监听新连接

//7. 主线程进入循环，监听新连接的到来以及出来新连接上的数据收发

}

3 文件说明

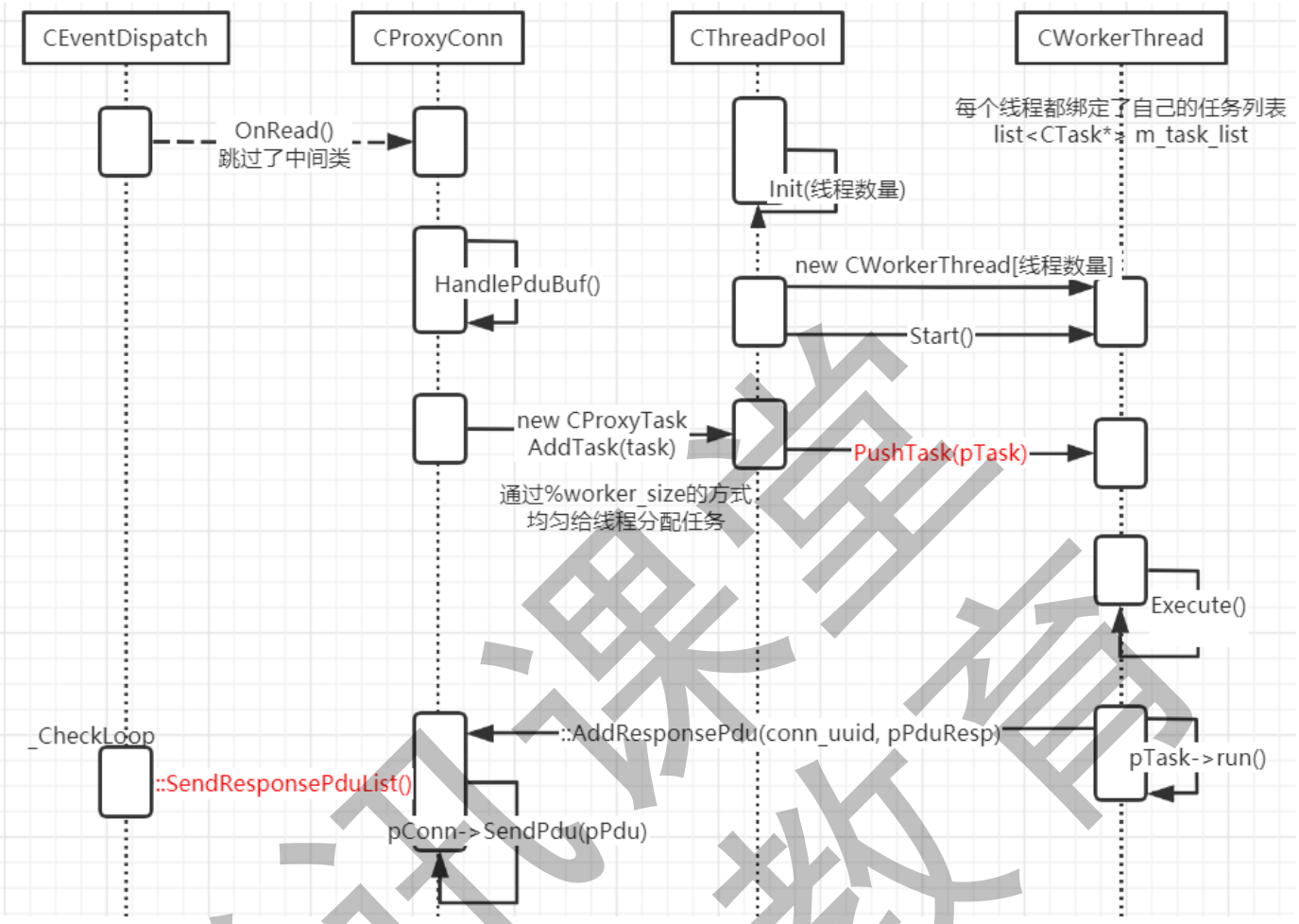
文件说明

- CachePool.h和CachePool.cpp：redis连接池
- DBPool.h和DBPool.cpp：MySQL连接池
- HandlerMap.h和HandlerMap.cpp：将每个业务和处理函数进行绑定，比如 m_handler_map.insert(make_pair(uint32_t(CID_OTHER_VALIDATE_REQ), DB_PROXY::doLogin));， DB_PROXY::doLogin是处理用户登录的。
- ProxyConn.h和ProxyConn.cpp：其他server连接db_proxy_server时都产生一个或者多个CProxyConn
- ProxyTask.h和ProxyTask.cpp：对应业务任务的封装，需要传递从参数：
  - conn_uuid：对应连接ProxyConn的id，每个ProxyConn都有唯一的id进行绑定，在epoll所在主循环回发数据的时候根据该id查找到对应的ProxyConn。CProxyConn* pConn = get_proxy_conn_by_uuid(pResp->conn_uuid);
  - pdu_handler：对应的业务处理函数
  - CImPdu：需要处理的pdu
- SyncCenter.h和SyncCenter.cpp：从MySQL数据库同步群主成员到Redis缓存。
- business目录：主要是数据库的操作，Model数据库操作，Action业务逻辑，以DB_PROXY::为命名空间的函数为业务入口函数：

- AudioModel.h/cpp：音频消息业务
- DepartAction.h/cpp和DepartModel.h/cpp：部门信息
- FileAction.h/cpp和FileModel.h/cpp：文件传输
- GroupAction.h/cpp和GroupModel.h/cpp：群相关操作
- GroupMessageModel.h/cpp：群消息
- InterLogin.h/cpp：登录数据库验证
- Login.h/cpp：登录逻辑处理
- MessageContent.h/cpp：消息处理（业务入口）
- MessageCounter.h/cpp：未读消息（业务入口）
- MessageModel.h/cpp: 消息处理实体（操作数据库）
- RecentSession.h/cpp：会话管理（业务入口）
- RelationModel.h/cpp：会话管理（操作数据库）
- SessionModel.h/cpp：会话处理（操作数据库）
- UserAction.h/cpp：用户管理（业务入口）
- UserModel.h/cpp：用户管理（操作数据库）

4 响应流程

重点关注：接收数据，处理数据，回发处理后的数据。



5 db_proxy_server退出机制

不中断当前的执行，而是在一个单独的线程中处理signal，以便mainloop()有机会优雅地停止？

使用信号的方式退出db_proxy_server，具体响应流程：

1. kill -SIGTERM pid
2. sig_handler进行响应，并通知各个连接db_proxy_server的组件（CID_OTHER_STOP_RECV_PACKET），各个组件收到（CID_OTHER_STOP_RECV_PACKET，响应函数为_HandleStopReceivePacket）后将其连接的db_proxy_server通道m_bOpen = false;

比如msg_server

```
void CDBServConn::_HandleStopReceivePacket(CImPdu* pPdu)
{
    log("HandleStopReceivePacket, from %s:%d.",
            g_db_server_list[m_serv_idx].server_ip.c_str(), g_db_server_list[m_serv_idx].server_port);

    m_bOpen = false;
}
```

3. CSyncCenter::getInstance()->stopSync()，将doSyncGroupChat所在的线程停止。
4. 然后注册定时器，4秒后调用exit_callback退出。

```
(1) 在另一个终端sudo kill -15 8180 (db_proxy_server的pid)

(2) db_proxy_server的log显示
11:40:59,424  <ProxyConn.cpp>|<59>|<sig_handler>,receive SIGTERM, prepare for exit
11:41:03,580 <ProxyConn.cpp>|<52>|<exit_callback>,exit_callback...
```

6 CSyncCenter

目前**主要是群里有人发消息后，更新其他人对应的最近会话信息**

1. CSyncCenter::getInstance()->init()；初始化m_nLastUpdateGroup的时间
2. CSyncCenter::getInstance()->startSync()，开启doSyncGroupChat线程，在线程里面循环执行任务
3. CSyncCenter::doSyncGroupChat入口函数，**主要是群里有人发消息后，更新其他人对应的最近会话信息**
4. 获取最近更新的群的更新时间，以getLastUpdateGroup(实质是m_nLastUpdateGroup)为基准，并以group_id为key保存到mapChangedGroup
5. 将当前的时间updateLastUpdateGroup更新到m_nLastUpdateGroup，并更新到redis 缓存的last_update_group（属于unread所在的db）
6. 遍历mapChangedGroup，将每个群的成员getGroupUser取出来，然后getSessionId获取会话ID更新每个群成员的对应群的会话。

7 redis数据库

redis cache划分

数据库	名称	最大连接数	主机	说明
1	unread	16	127.0.0.1:6379	未读消息计数器
2	group_set	16	127.0.0.1:6379	群组设置
3	sync	16	127.0.0.1:6379	同步控制
4	token	16	127.0.0.1:6379	推送相关的token
5	group_member	16	127.0.0.1:6379	群组成员

unread key范例

SQL 复制代码

```
127.0.0.1:6379[1]> KEYS *
 1) "3_1_im_user_group"
 2) "last_update_group"
 3) "2_2_im_user_group"
 4) "3_2_im_user_group"
 5) "msg_id_1"
 6) "1_im_group_msg"
 7) "msg_id_2"
 8) "2_im_group_msg"
 9) "group_msg_id_1"
10) "1_1_im_user_group"
11) "1_2_im_user_group"
12) "total_user_update"
13) "2_1_im_user_group"
14) "group_msg_id_2"
15) "msg_id_3"
```

group_set范例

127.0.0.1:6379[2]> KEYS *

1) "group_set_1"

2) "group_set_2"

group_member 范例

```
127.0.0.1:6379[5]> KEYS *
1) "group_member_2"
127.0.0.1:6379[5]> HGETALL group_member_2
1) "1"
2) "1585215616"
3) "2"
4) "1585215616"
5) "3"
6) "1585221276"
127.0.0.1:6379[5]>
```

7.1 unread

对于群组未读消息，独立开来看每个人都是不一样的。所以设计key的时候是nGroupId+nUserId结合。

清除群组消息计数器

bool CGroupMessageModel::clearMessageCount(uint32_t nUserId, uint32_t nGroupId)
key设计：int2string(nGroupId) + GROUP_TOTAL_MSG_COUNTER_REDIS_KEY_SUFFIX;
群ID加固定后缀。

增加群消息计数

bool CGroupMessageModel::incMessageCount(uint32_t nUserId, uint32_t nGroupId)

群消息id

uint32_t CGroupMessageModel::getMsgId(uint32_t nGroupId)

获取用户所有群的未读消息之和

void CGroupMessageModel::getUnReadCntAll(uint32_t nUserId, uint32_t &nTotalCnt)

重置群消息id

bool CGroupMessageModel::resetMsgId(uint32_t nGroupId)

增加个人未读消息

void CMessageModel::incMsgCount(uint32_t nFromId, uint32_t nToId)

获取未读消息总数和内容

void CMessageModel::getUnreadMsgCount(uint32_t nUserId, uint32_t &nTotalCnt, list<IM::BaseDefine::UnreadInfo>& lsUnreadCount)

获取消息id

uint32_t CMessageModel::getMsgId(uint32_t nRelateId)

获取未读消息总数

void CMessageModel::getUnReadCntAll(uint32_t nUserId, uint32_t &nTotalCnt)

重置消息id

bool CMessageModel::resetMsgId(uint32_t nRelateId)

从cache里面加载上次同步的时间信息等

void CSyncCenter::init()

更新同步时间

void CSyncCenter::updateTotalUpdate(uint32_t nUpdated)

更新上次同步群组信息时间

void CSyncCenter::updateLastUpdateGroup(uint32_t nUpdated)

清除用户未读计数

void CUserModel::clearUserCounter(uint32_t nUserId, uint32_t nPeerId, IM::BaseDefine::SessionType nSessionType)

7.2 group_set

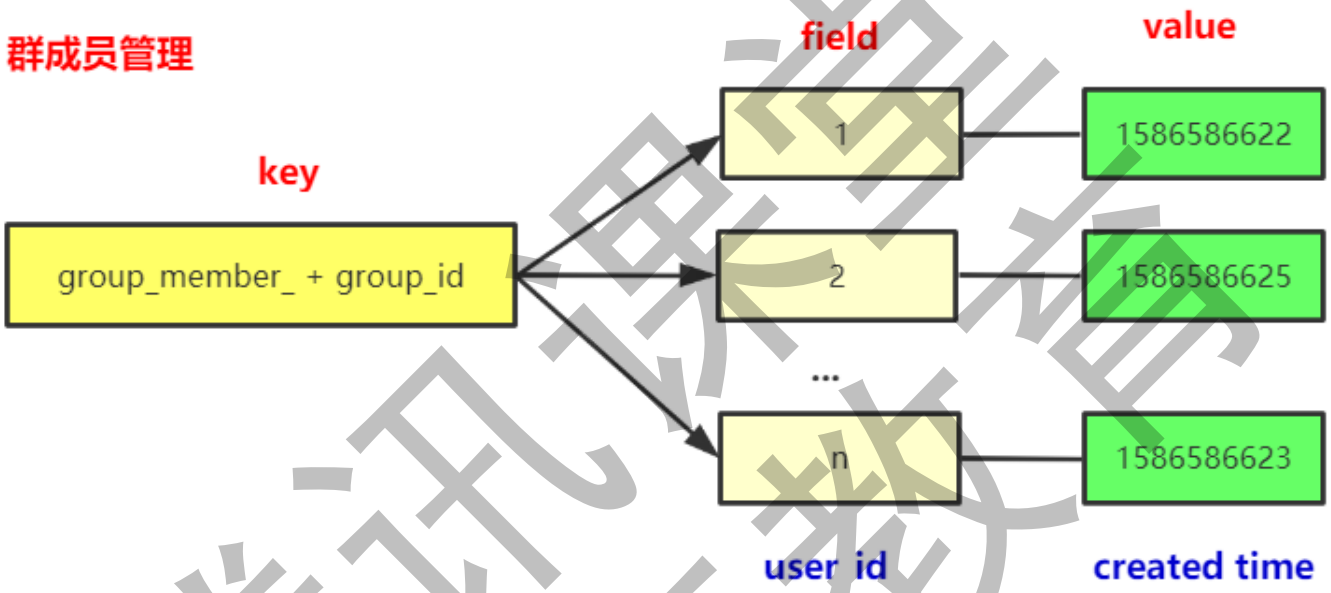
bool CGroupModel::setPush(uint32_t nUserId, uint32_t nGroupId, uint32_t nType, uint32_t nStatus)
void CGroupModel::getPush(uint32_t nGroupId, list<uint32_t>& lsUser, list<IM::BaseDefine::ShieldStatus>& lsPush)

7.3 token

void setDevicesToken(CImPdu* pPdu, uint32_t conn_uuid)
void getDevicesToken(CImPdu* pPdu, uint32_t conn_uuid)

7.4 group_member

群成员管理的redis缓存设计，以hash为存储结构。key使用group_*member* + group_id，hash里面的field使用user_id，value则对应创建时间。



插入新成员

bool CGroupModel::insertNewMember(uint32_t nGroupId, set<uint32_t>& setUsers)

判断是否在群里

bool CGroupModel::isInGroup(uint32_t nUserId, uint32_t nGroupId)

删除成员

bool CGroupModel::removeMember(uint32_t nGroupId, set<uint32_t> &setUser, list<uint32_t>& lsCurUserId)

删除重复的成员

void CGroupModel::removeRepeatUser(uint32_t nGroupId, set<uint32_t> &setUser)

获取群组用户成员id

void CGroupModel::getGroupUser(uint32_t nGroupId, list<uint32_t> &lsUserId)

是否是有效群id

bool CGroupModel::isValidateGroupId(uint32_t nGroupId)
判断"group_member_"+int2string(nGroupId); 作为key是否存储在redis中，如果不存在则认为群无效。

获取成员加入时间

uint32_t CGroupModel::getUserJoinTime(uint32_t nGroupId, uint32_t nUserId)

清除群成员

void CGroupModel::clearGroupMember(uint32_t nGroupId)

8 具体业务分析

8.1 登录

DB_PROXY::doLogin登录请求验证

请求命令：CID_OTHER_VALIDATE_REQ
回应命令：CID_OTHER_VALIDATE_RSP

1. 请求结构

C 复制代码

```
message IMValidateReq{
    //cmd id:   0x0703
    required string user_name = 1;  // 用户名
    required string password = 2;   // 用户密码
    optional bytes attach_data = 20;
}
```

2. 响应结构

```
message IMValidateRsp{
    //cmd id:   0x0704
    required string user_name = 1;  // 用户名
    required uint32 result_code = 2; // 返回码，0为正常
    optional string result_string = 3; // 返回说明
    optional IM.BaseDefine.UserInfo user_info = 4; // 用户详细信息
    optional bytes attach_data = 20;
}
```

```
message UserInfo{
    required uint32 user_id = 1;
    required uint32 user_gender = 2;    //// 用户性别,男：1 女：2 人妖/外星人：0
    required string user_nick_name = 3; //绰号
    required string avatar_url = 4;
    required uint32 department_id = 5;
    required string email = 6;
    required string user_real_name = 7; //真名
    required string user_tel = 8;
    required string user_domain = 9;    //用户名拼音
    required uint32 status = 10;        //0:在职  1. 试用期 2. 正式 3. 离职 4.实习，  client端需要对“离职”进行不展示
    optional string sign_info = 11;
}
```

3. 处理逻辑

- 检测用户登录密码错误情况；
- 验证用户名和用户密码是否匹配
  - 如果匹配则读取用户信息，则设置IMValidateRsp，返回码设置为0，代表正常
  - 如果验证失败，则设置IMValidateRsp，返回码设置为2，代表验证失败

CInterLoginStrategy::doLogin具体的操作：

- 通过"select * from IMUser where name='" + strName + "' and status=0" 查询相应用户的信息，strName是传入的用户名，status=0表示该user的状态是正常的。
- 读取用户信息，然后对比密码，特别需要注意的是：数据库存储的password = md5(用户传递的password(已经用md5加密) + salt（混淆码）)，所以对比密码的时候 （用户传递的password(已经

用md5加密）+ salt）做md5计算后再和数据库读取出来的password做对比，即是：

```
string strInPass = strPass + strSalt;
char szMd5[33];
CMd5::MD5_Calculate(strInPass.c_str(), strInPass.length(), szMd5);
string strOutPass(szMd5);  再将该计算结果和数据库存储的password做对比
```

- 用户名密码匹配成功则返回true，匹配失败则返回false。

DB_PROXY::doPushShield 勿扰模式设置

请求命令：CID_LOGIN_REQ_PUSH_SHIELD

回应命令：CID_LOGIN_RES_PUSH_SHIELD

1. 请求结构

```
message IMPushShieldReq {
    //cmd id:           0x010c
    required uint32 user_id = 1;    // 用户ID
    required uint32 shield_status = 2;// 1:开启, 0: 关闭
    optional bytes attach_data = 20;  // 服务端用, 客户端不用设置
}
```

2. 响应结构

```
message IMPushShieldRsp {
    //cmd id:           0x010d
    required uint32 user_id = 1;     // 用户ID
    required uint32 result_code = 2; // 值：  0:successed 1:failed
    optional uint32 shield_status = 3;  // 值：  如果result_code值为0(successed),
                                        // 则shield_status值设置，  1:开启, 0:关闭

    optional bytes attach_data = 20;    // 服务端用，客户端不用设置
}
```

3. 处理逻辑

- 该逻辑比较简单，根据用户user_id更新push_shield_status字段即可。
- 具体处理的函数CUserModel::updatePushShield，使用"update IMUser set `push_shield_status`="+ int2string(shield_status) + ", `updated`=" + int2string(now) + " where id="+int2string(user_id)，本质上来讲就是更新user_id对应的push_shield_status状态。
- 封装IMPushShieldRsp回复请求端
  - 如果更新成功则IMPushShieldRsp的result_code设置为0
  - 如果更新失败则IMPushShieldRsp的result_code设置为1

DB_PROXY::doQueryPushShield查询勿扰状态

请求命令：CID_LOGIN_REQ_QUERY_PUSH_SHIELD

回应命令：CID_LOGIN_RES_QUERY_PUSH_SHIELD

1. 请求结构

```
// 如果用户重新安装app，第一次启动登录成功后，app主动查询
// 服务端返回IMQueryPushShieldRsp
message IMQueryPushShieldReq {
    //cmd id:           0x010e
    required uint32 user_id = 1;
    optional bytes attach_data = 20;// 服务端用，客户端不用设置
}
```

2. 回应结构

```
message IMQueryPushShieldRsp {
    //cmd id:           0x010f
    required uint32 user_id = 1;
    required uint32 result_code = 2; // 值： 0:successed 1:failed
    optional uint32 shield_status = 3;  // 值： 1:开启， 0:关闭
    optional bytes attach_data = 20;
}
```

3. 处理逻辑

- 该逻辑比较简单，根据用户user_id查询push_shield_status字段即可。
- 具体处理的函数CUserModel::getPushShield，使用"select push_shield_status from IMUser where id="+int2string(user_id)，本质上来讲就是查询user_id对应的push_shield_status状态。
- 封装IMQueryPushShieldRsp回复请求端
  - 如果查询成功则IMQueryPushShieldRsp的result_code设置为0
  - 如果查询失败则IMQueryPushShieldRsp的result_code设置为1

8.2 最近会话

DB_PROXY::getRecentSession获取最近会话

请求命令：CID_BUDDY_LIST_RECENT_CONTACT_SESSION_REQUEST
回应命令：CID_BUDDY_LIST_RECENT_CONTACT_SESSION_RESPONSE

1. 请求结构

```
message IMRecentContactSessionReq{
    //cmd id:       0x0201
    required uint32 user_id = 1;  // 用户id
    required uint32 latest_update_time = 2; // 最近更新时间
    optional bytes attach_data = 20;
}
```

2. 回应结构

一个登陆用户对应多个会话，所以使用repeated IM.BaseDefine.ContactSessionInfo进行描述

```
message IMRecentContactSessionRsp{
    //cmd id:       0x0202
    required uint32 user_id = 1;  // 用户id
    repeated IM.BaseDefine.ContactSessionInfo contact_session_list = 2; // 最近会话列表
    optional bytes attach_data = 20;
}
```

注意ContactSessionInfo的session_id是对方用户ID，不是会话ID

```
message ContactSessionInfo{
    required uint32 session_id = 1;  //对方用户ID，主要这个不是会话ID
    required SessionType session_type = 2; // 会话类型 1：单聊，2：群聊
    required SessionStatusType session_status = 3; // 会话状态 0:正常，1：已删除
    required uint32 updated_time = 4;   // 最新时间
    required uint32 latest_msg_id = 5;  // 最后的消息id
    required bytes latest_msg_data = 6; // 最后的消息数据
    required MsgType latest_msg_type = 7; // 最后的消息类型 0x1: 文本单聊，0x2:语言单聊，
                                          // 0x11:文本群聊
    required uint32 latest_msg_from_user_id = 8; // 最后消息来自哪个用户ID，主要是群聊有用
}
```

3. 对应数据库

SQL | 复制代码

```sql
CREATE TABLE `IMRecentSession` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `userId` int(11) unsigned NOT NULL COMMENT '用户id',
    `peerId` int(11) unsigned NOT NULL COMMENT '对方id',
    `type` tinyint(1) unsigned DEFAULT '0' COMMENT '类型，1-用户,2-群组',
    `status` tinyint(1) unsigned DEFAULT '0' COMMENT '用户:0-正常，1-用户A删除,群组:0-正常，1-被删除',
    `created` int(11) unsigned NOT NULL DEFAULT '0' COMMENT '创建时间',
    `updated` int(11) unsigned NOT NULL DEFAULT '0' COMMENT '更新时间',
    PRIMARY KEY (`id`),
    KEY `idx_userId_peerId_status_updated` (`userId`,`peerId`,`status`,`updated`),
    KEY `idx_userId_peerId_type` (`userId`,`peerId`,`type`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

4. 处理逻辑

- 根据用户ID（user_id）和最近本地更新的时间lastTime 读取IMRecentSession表
- 具体处理函数CSessionModel::getRecentSession，使用"select * from IMRecentSession where userId = " + int2string(nUserId) + " and status = 0 and updated >" + int2string(lastTime) + " order by updated desc limit 100"； 即是获取> 客户端最后更新时间的IMRecentSession，并进行将序（最新的时间排在前面）和限制最多读取100条数据。
- 将读取到的IMRecentSession 信息填充ContactSessionInfo，每个会话填充一个ContactSessionInfo结构，这里只填充了 session_id对端用户ID，session_status会话状态，session_type会话类型，updated_time会话更新时间。**那还有latest_msg_from_user_id最后发消息的用户ID，最后的消息ID latest_msg_id，最后的消息latest_msg_data，最后的消息类型latest_msg_type来自哪里呢?** 这部分信息需要从消息表获取。
- 获取上步提到的还没有获取的信息，具体处理函数CSessionModel::fillSessionMsg，单聊消息调用CMessageModel::getLastMsg，群聊消息调用CGroupMessageModel::getLastMsg，读取相应的消息后继续封装ContactSessionInfo的剩余未填充的字段**latest_msg_from_user_id、latest_msg_id、latest_msg_data、latest_msg_type。**
  - 单聊CMessageModel::getLastMsg：使用"select msgId,type,content from " + "IMMessage_" + int2string(nRelateId % 8) + " force index (idx_relateId_status_created) where relateId= " + int2string(nRelateId) + " and status = 0 order by created desc, id desc limit 1"；根据fromId和toId映射的nRelateId，查找对应的"IMMessage_" + int2string(nRelateId % 8)消息表，只获取最新的一条数据。获取的字段为消息id msgId,消息类型 type，消息内容content
  - 群聊CGroupMessageModel::getLastMsg：使用"select msgId, type,userId, content from " + "IMGroupMessage_" + int2string(nGroupId % 8)+ " where groupId = " +

int2string(nGroupId) + " and status = 0 order by created desc, id desc limit 1"；从IMGroupMessage_消息表获取群聊最新的一条消息，获取的字段为消息ID msgId，消息类型type，发言人userId，消息内容content。

- DB_PROXY::getRecentSession：回到该函数根据获取数据填充IMRecentContactSessionRsp后回复请求者。

DB_PROXY::deleteRecentSession删除最近会话

请求命令：CID_BUDDY_LIST_REMOVE_SESSION_REQ
回复命令：CID_BUDDY_LIST_REMOVE_SESSION_RES

1. 请求结构

SQL 复制代码

```
message IMRemoveSessionReq{
  //cmd id:   0x0206
  required uint32 user_id = 1;   // 用户ID
  required IM.BaseDefine.SessionType session_type = 2;  // 会话类型 1：单聊，2：群聊
  required uint32 session_id = 3; // 对方user_id，或者群id
  optional bytes attach_data = 20;
}
```

2. 回应结构

SQL 复制代码

```
message IMRemoveSessionRsp{
  //cmd id:   0x0207
  required uint32 user_id = 1;  // 用户id
  required uint32 result_code = 2; // 返回码，0：正常，1：失败
  required IM.BaseDefine.SessionType session_type = 3;// 会话类型 1：单聊，2：群聊
  required uint32 session_id = 4;// 对方user_id，或者群id
  optional bytes attach_data = 20;
}
```

3. 处理逻辑

- DB_PROXY::deleteRecentSession入口

- 调用CSessionModel::getSessionId函数检测user_id和peer_id(从session_id获取)之间的sessionId是否存在，通过"select id from IMRecentSession where userId=" + int2string(nUserId) + " and peerId=" + int2string(nPeerId) + " and type=" + int2string(nType) 进行查询。
- 如果存在则调用CSessionModel::removeSession函数进行删除，通过"update IMRecentSession set status = 1, updated="+int2string(nNow)+" where id=" + int2string(nSessionId); 更新status的值，置为1则表明该会话被删除，这时候客户端看不到该会话，但数据库还是做保留。
- 根据是否删除成功封装IMRemoveSessionRsp回复请求端。

8.3 用户信息

DB_PROXY::getUserInfo 获取用户信息

请求命令：CID_BUDDY_LIST_USER_INFO_REQUEST
回复命令：CID_BUDDY_LIST_USER_INFO_RESPONSE

1. 请求结构

SQL 复制代码

```sql
message IMUsersInfoReq{
  //cmd id:    0x0204
  required uint32 user_id = 1;      // 用户ID
  repeated uint32 user_id_list = 2; // 请求的用户ID列表，这里也说明可以批量请求用户信息
  optional bytes attach_data = 20;
}
```

2. 回应结构

SQL 复制代码

```sql
message IMUsersInfoRsp{
  //cmd id:    0x0205
  required uint32 user_id = 1;  // 用户ID
  repeated IM.BaseDefine.UserInfo user_info_list = 2; // 请求到的用户信息
  optional bytes attach_data = 20;
}
```

```SQL
message UserInfo{
  required uint32 user_id = 1;
  required uint32 user_gender = 2;  //// 用户性别,男: 1 女: 2 人妖/外星人: 0
  required string user_nick_name = 3; //绰号
  required string avatar_url = 4;
  required uint32 department_id = 5;
  required string email = 6;
  required string user_real_name = 7; //真名
  required string user_tel = 8;
  required string user_domain = 9;  //用户名拼音
  required uint32 status = 10;        //0:在职  1. 试用期 2. 正式 3. 离职 4. 实习,  client端需要对“离职”进行不展示
  optional string sign_info = 11;
}
```

3. 处理逻辑

- DB_PROXY::getUserInfo入口
- 从IMUsersInfoReq解析出用请求的用户ID，将其插入到list里面
- 然后调用CUserModel::getUsers 读取相应用户ID的用户信息，使用select * from IMUser where id in (" + strClause + ")"， 其中strClause是要请求的用户id，使用","隔开。
- 封装到IMUsersInfoRsp 回复请求端

DB_PROXY::getChangedUser获取所有更新的用户信息

请求命令：CID_BUDDY_LIST_ALL_USER_REQUEST

回应命令：CID_BUDDY_LIST_ALL_USER_RESPONSE

1. 请求结构

```SQL
message IMAllUserReq{
  //cmd id:   0x0208
  required uint32 user_id = 1;  // 用户ID
  required uint32 latest_update_time = 2; // > latest_update_time时间后的用户
  optional bytes attach_data = 20;
}
```

请求晚于latest_update_time更新的用户，请求后返回的不仅是用户ID，且包括user_name、等等其他的信息。

2. 回应结构

```
message IMAllUserRsp{
  //cmd id:   0x0209
  required uint32 user_id = 1;
  required uint32 latest_update_time = 2;
  repeated IM.BaseDefine.UserInfo user_list = 3;
  optional bytes attach_data = 20;
}
```

```
message UserInfo{
  required uint32 user_id = 1;
  required uint32 user_gender = 2;  //// 用户性别,男：1 女：2 人妖/外星人：0
  required string user_nick_name = 3; //绰号
  required string avatar_url = 4;
  required uint32 department_id = 5;
  required string email = 6;
  required string user_real_name = 7; //真名
  required string user_tel = 8;
  required string user_domain = 9;  //用户名拼音
  required uint32 status = 10;          //0:在职  1. 试用期 2. 正式 3. 离职 4. 实习，  client端需要对“离职”进行不展示
  optional string sign_info = 11;
}
```

3. 处理逻辑

- DB_PROXY::getChangedUser入口
- 解析客户端最近的更新时间latest_update_time，和CSyncCenter::getInstance()->getLastUpdate()的时间进行对比，如果晚于CSyncCenter::getInstance()->getLastUpdate()则去数据库：
  - 先用CUserModel::getChangedId获取更新的用户ID列表，使用 "select id, updated from IMUser where updated>=" + int2string(nLastTime)， 从这里的设计看，没有对每次请求的用户数量进行限制。
  - 然后使用上一步获取的用户ID列表，调用CUserModel::getUsers获取对应id的用户信息，
- 将获取到的用户信息封装到IMAllUserRsp回复请求者，注意IMAllUserRsp的latest_update_time字段。

DB_PROXY::getChgedDepart 获取所有更新的部门信息

请求命令：CID_BUDDY_LIST_DEPARTMENT_REQUEST

响应命令：CID_BUDDY_LIST_DEPARTMENT_RESPONSE

1. 请求结构

SQL 复制代码

```
message IMDepartmentReq{
  //cmd id:   0x0210
  required uint32 user_id = 1;
  required uint32 latest_update_time = 2;
  optional bytes attach_data = 20;
}
```

请求晚于latest_update_time更新的部门信息，请求后返回的不仅是用户ID，且包括user_name、等等其他的信息。

2. 回应结构

SQL 复制代码

```
message IMDepartmentRsp{
  //cmd id:   0x0211
  required uint32 user_id = 1;
  required uint32 latest_update_time = 2;  // 最后更新的时间
  repeated IM.BaseDefine.DepartInfo dept_list = 3; // 部门列表信息
  optional bytes attach_data = 20;
}
```

SQL 复制代码

```
message DepartInfo{
  required uint32 dept_id = 1;
  required uint32 priority = 2;
  required string dept_name = 3;
    required uint32 parent_dept_id = 4;
    required DepartmentStatusType dept_status = 5;
}
```

3. 处理逻辑

- DB_PROXY::getChgedDepart入口，原理和getChangedUser大致相同。
- 解析客户端最近的更新时间latest_update_time，到数据对比IMDepart的updated时间：
  - 先用CDepartModel::getChgedDeptId获取更新的用户ID列表，使用 "select id, updated from IMDepart where updated>=" + int2string(nLastTime)， 从这里的设计看，没有对每次请求的用户数量进行限制。
  - 然后使用上一步获取的部门ID列表，调用CDepartModel::getDepts获取对应id的部门信息，
- 将获取到的部门信息封装到IMDepartmentRsp回复请求者，注意IMDepartmentRsp的latest_update_time字段。

DB_PROXY::changeUserSignInfo更新用户签名信息

请求命令：CID_BUDDY_LIST_CHANGE_SIGN_INFO_REQUEST
回复命令：CID_BUDDY_LIST_CHANGE_SIGN_INFO_RESPONSE

1. 请求结构

SQL 复制代码

```
message IMChangeSignInfoReq{
    //cmd id:    0x0213
     required uint32 user_id = 1;
     required string sign_info = 2;
     optional bytes attach_data = 20;
}
```

2. 回应结构

SQL 复制代码

```
message IMChangeSignInfoRsp{
     //cmd id:    0x0214
     required uint32 user_id = 1;
     required uint32 result_code = 2;
     optional string sign_info = 3;       // 此字段服务端用，客户端直接忽略
     optional bytes attach_data = 20;
}
```

3. 处理逻辑

- DB_PROXY::changeUserSignInfo入口，该逻辑比较简单
- 调用CUserModel::updateUserSignInfo进行更新签名，"update IMUser set `sign_info`='" + sign_info + "', `updated`=" + int2string(now) + " where id="+int2string(user_id); 就是一个修改

的过程，但是需要注意的是不只是更新sign_info，而且需要更新updated。

- 将更新结果封装到IMChangeSignInfoRsp回复给请求者。

8.4 消息内容

DB_PROXY::sendMessage 发送消息

请求命令：CID_MSG_DATA

回复命令：CID_MSG_DATA

包括单聊、群聊消息的处理，在db_proxy_server的主要处理：

1. 设置消息创建时间
2. 更新会话ID，如果没有存在会话ID则增加会话ID
3. 设置消息id msgId

1. 请求结构和回应结构一样

SQL 复制代码

```
message IMMsgData{
  //cmd id:   0x0301
  required uint32 from_user_id = 1;       //消息发送方
  required uint32 to_session_id = 2;      //消息接受方，单聊是对端user_id，群聊是group_id
  required uint32 msg_id = 3;         // 消息ID由服务器设置
  required uint32 create_time = 4;    // 消息时间由服务器设置
  required IM.BaseDefine.MsgType msg_type = 5; // 消息类型
  required bytes msg_data = 6;    // 消息内容
  optional bytes attach_data = 20;
}
```

2. 处理逻辑

这里我们主要讲述 文本单聊和文本群聊，两种分开来进行讲解，

先讲解RelationId，主要是针对单聊之间user_id的关系映射或者群聊时user_id和group_id之间的关系映射，在存储的时候主要根据两者的id写入数据库，使用"insert into IMRelationShip (`smallId`,`bigId`,`status`,`created`,`updated`) values(?,?,?,?,?) ，smallId和bigId代表产生关系的用户ID，大的用户ID则为bigId，小的用户ID则为smallId。目的是为了方便处理聊天消息方便：

- 单聊时两者之间的消息使用同一份数据。

MSG_TYPE_SINGLE_TEXT：单聊

- DB_PROXY::sendMessage入口
- 设置消息时间(uint32_t)time(NULL)
- 调用CRelationModel::getRelationId获取RelationId，如果没有则创建
- 调用CMessageModel::getMsgId获取消息MsgId，以"msg_id_" + int2string(nRelateId)为key，使用string类型的redis进行存储，每次调用getMsgId对应"msg_id_" + int2string(nRelateId)的value做+1操作；
- 使用CMessageModel::sendMessage将消息写入数据库，"insert into " + "IMMessage_" + int2string(nRelateId % 8)+ " (`relateId`, `fromId`, `toId`, `msgId`, `content`, `status`, `type`, `created`, `updated`) values(?, ?, ?, ?, ?, ?, ?, ?, ?)";
  - 并且调用incMsgCount记录未读消息计数
- 调用CSessionModel::updateSession为fromId和toId更新会话，分别为updateSession(nSessionId, nNow)和updateSession(nPeerSessionId, nNow)，是两端一定要注意。

MSG_TYPE_GROUP_TEXT：群聊

- DB_PROXY::sendMessage入口
- 设置消息时间(uint32_t)time(NULL)
- 调用CRelationModel::getRelationId获取RelationId，如果没有则创建
- 调用CGroupMessageModel::getMsgId获取消息MsgId，以"group_msg_id_" + int2string(nGroupId)为key，使用string类型的redis进行存储，每次调用getMsgId对应"group_msg_id_" + int2string(nGroupId)的value做+1操作；
- 使用CGroupMessageModel::sendMessage将消息写入数据库，"insert into " + "IMGroupMessage_" + int2string(nGroupId % 8)+ " (`groupId`, `userId`, `msgId`, `content`, `type`, `status`, `updated`, `created`) "\
  "values(?, ?, ?, ?, ?, ?, ?, ?)";
  - 并调用CGroupModel::updateGroupChat更新群最后聊天时间，使用"update IMGroup set lastChated=" + int2string(nNow) + " where id=" + int2string(nGroupId);
  - 并调用incMessageCount增加群消息计数，group_id + _im_group_msg 对应field的count计数+1； 并将对应的群消息计数更新给发消息的user，对应user_id_ + group_id + _im_user_group
  - 并调用CGroupMessageModel::clearMessageCount清除当前用户在当前群组发送消息的未读计数，本质上就是将自己已读计数 = 群组消息计数。并不是真正的设置为0
- 调用CSessionModel::updateSession为fromId和group_id之间的关系更新会话，和单聊不一样，需要更新当前user_id和group_id之间的会话即可。

补充：未读消息在unread，所在的db索引为1

127.0.0.1:6379[> select 1
ok

```
127.0.0.1:6379[1]> KEYS *
1) "3_1_im_user_group"
2) "last_update_group"
3) "2_2_im_user_group"
4) "3_2_im_user_group"
5) "msg_id_1"
6) "1_im_group_msg"
7) "msg_id_2"
8) "2_im_group_msg"
9) "group_msg_id_1"
10) "1_1_im_user_group"
11) "1_2_im_user_group"
12) "total_user_update"
13) "2_1_im_user_group"
14) "group_msg_id_2"
15) "msg_id_3"
```

DB_PROXY::getMessage 获取消息

请求命令：CID_MSG_LIST_REQUEST

回复命令：CID_MSG_LIST_RESPONSE

根据msgId和msgCnt获取消息起始位置和消息数量

1. 请求结构

SQL | 复制代码

```
message IMGetMsgListReq{
  //cmd id:   0x0309
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2; // 会话类型 1：单聊，2：群聊
  required uint32 session_id = 3;  // 单聊：对端user_id；群聊：group_id
  required uint32 msg_id_begin = 4;  // 消息起始id，设置为0的时候代表获取最新的msg_cnt条消息
  required uint32 msg_cnt = 5;   // 消息数量
  optional bytes attach_data = 20;
}
```

2. 回应结构

```SQL
message IMGetMsgListRsp{
  //cmd id:   0x030a
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2; // 会话类型 1：单聊，2：群聊
  required uint32 session_id = 3; // 单聊：对端user_id；群聊：group_id
  required uint32 msg_id_begin = 4; // 消息起始id
  repeated IM.BaseDefine.MsgInfo msg_list = 5; // 多个消息
  optional bytes attach_data = 20;
}
```

```SQL
  required uint32 msg_id = 1;
  required uint32 from_session_id = 2;   //发送的用户id
  required uint32 create_time = 3;
  required MsgType msg_type = 4;
  required bytes msg_data = 5;
}
```

3. 处理逻辑

SESSION_TYPE_SINGLE单聊

- 具体操作，调用CMessageModel::getMessage获取单聊消息，使用"select * from " + "IMMessage_" + int2string(nRelateId % 8)+ " force index (idx_relateId_status_created) where relateId= " + int2string(nRelateId) + " and status = 0 and msgId <=" + int2string(nMsgId)+ " order by created desc, id desc limit " + int2string(nMsgCnt);，本质就是查询IMMessage_表获取消息，关键点在于msgId <=" + int2string(nMsgId)和nMsgCnt的设置。 需要注意的是如果是想获取最新的n个消息则将msgId=0 使用"select * from " + "IMMessage_" + int2string(nRelateId % 8)+ " force index (idx_relateId_status_created) where relateId= " + int2string(nRelateId) + " and status = 0 order by created desc, id desc limit " + int2string(nMsgCnt);
- 将查询到的消息封装到MsgInfo，
- 再将多个MsgInfo封装到IMGetMsgListRsp回复给请求端。

SESSION_TYPE_GROUP群聊

- 具体操作，调用CGroupMessageModel::getMessage获取群聊消息，使用"select * from " + "IMGroupMessage_" + int2string(nGroupId % 8)+ " where groupId = " + int2string(nGroupId) + " and msgId<=" + int2string(nMsgId) + " and status = 0 and created>="+ int2string(nUpdated) + " order by created desc, id desc limit " + int2string(nMsgCnt); ，本质就是查询IMGroupMessage_表获取消息， 关键点在于msgId <=" + int2string(nMsgId)和nMsgCnt的设置。 需要注意的是如果是想获取最新的n个消息则将msgId=0 使用"select * from " + "IMGroupMessage_" + int2string(nGroupId % 8)+ " where groupId = " + int2string(nGroupId) + " and status = 0 and created>="+ int2string(nUpdated) + " order by created desc, id desc limit " + int2string(nMsgCnt);
- 将查询到的消息封装到MsgInfo，
- 再将多个MsgInfo封装到IMGetMsgListRsp回复给请求端。

DB_PROXY::getUnreadMsgCounter未读消息数

请求命令：CID_MSG_UNREAD_CNT_REQUEST
回复命令：CID_MSG_UNREAD_CNT_RESPONSE

既然有未读消息数量的说法，那未读消息数量是在哪里设置的？是单聊sendMessage时调用incMsgCount进行增加的。

未读消息数量是存储在redis cache，未读消息消息在存储在mysql数据库。
主要分两部分：

- 单聊的未读消息数量
- 群聊的未读消息数量

1. 请求结构

SQL | 复制代码

```
message IMUnreadMsgCntReq{
  //cmd id:   0x0307
  required uint32 user_id = 1;
  optional bytes attach_data = 20;
}
```

2. 回应结构

SQL 复制代码

```
message IMUnreadMsgCntRsp{
  //cmd id:   0x0308
  required uint32 user_id = 1;
  required uint32 total_cnt = 2;  // 总的未读消息数量
  repeated IM.BaseDefine.UnreadInfo unreadinfo_list = 3; // 不同fromId对应的未读信息
  optional bytes attach_data = 20;
}
```

SQL 复制代码

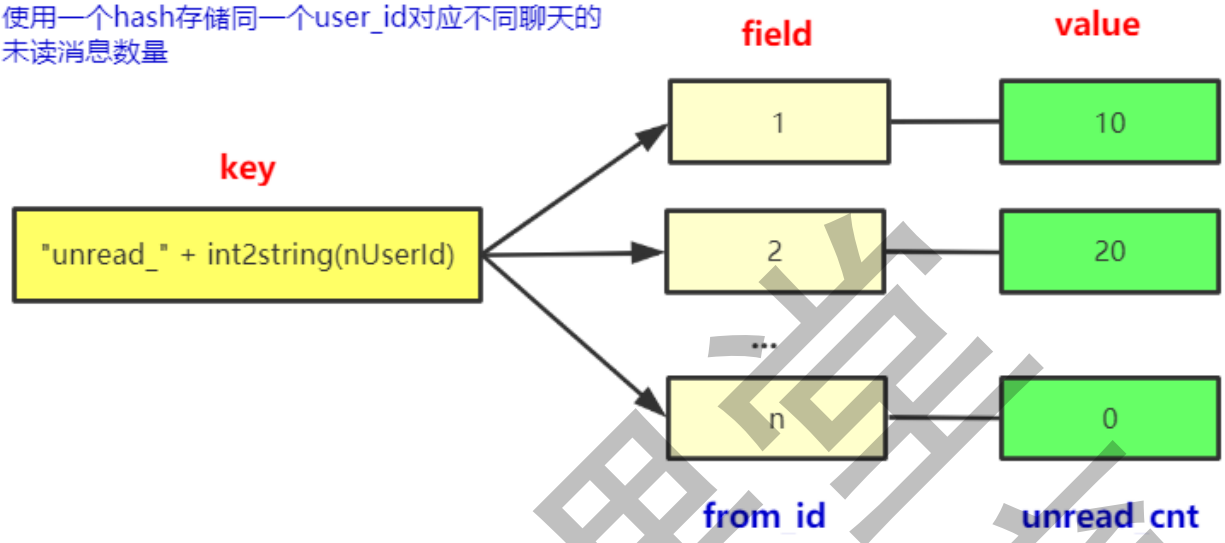
```
message UnreadInfo{
  required uint32 session_id = 1;
  required SessionType session_type = 2; // 会话类型: 单聊、群聊
  required uint32 unread_cnt = 3;  // 未读消息数量
  required uint32 latest_msg_id = 4;
  required bytes latest_msg_data = 5;
    required MsgType latest_msg_type = 6;
    required uint32 latest_msg_from_user_id = 7;          //来自哪一个用户id
}
```

3. 处理逻辑

单聊的未读消息

未读消息计数
key设计："unread_" + int2string(nUserId)

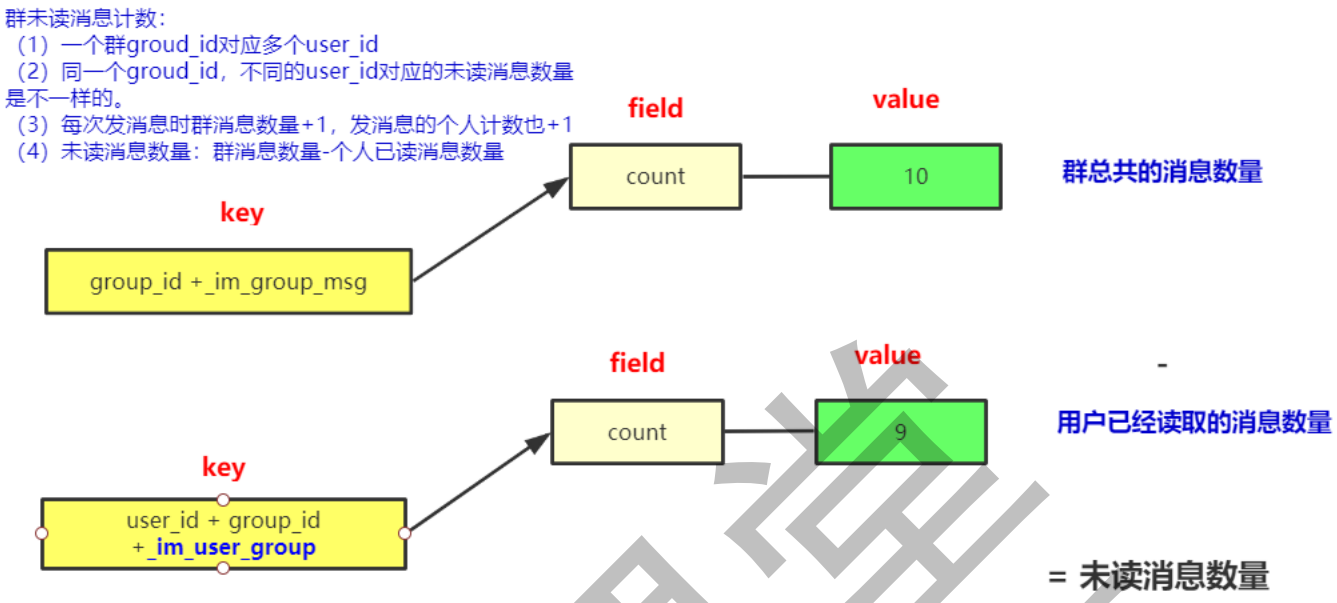
使用一个hash存储同一个user_id对应不同聊天的未读消息数量



未读消息数量存储在redis

- getUnreadMsgCounter入口函数
  - CMessageModel::getUnreadMsgCount：获取单聊未读消息数量，主要从 "unread_" + int2string(nUserId)进行读取，并将对应的from_id和unread_cnt设置到UnreadInfo，
    - 并调用CMessageModel::getLastMsg获取对应from_id的最新消息，使用"select msgId,type,content from " + strTableName + " force index (idx_relateId_status_created) where relateId= " + int2string(nRelateId) + " and status = 0 order by created desc, id desc limit 1"; 包含最后一条消息的 msgId, 消息类型type, 消息内容content

群聊的未读消息



getUnreadMsgCounter入口函数

- CGroupMessageModel::getUnreadMsgCount：获取群聊的信息
  - 先分析user_id有多少个群，并获取对应的群ID，使用CGroupModel::getUserGroupIds获取；
  - 获取group_id +_im_group_msg对应field （count）的value，即是该群的消息数量
  - 获取user_id + group_id +**_im_user_group**对应field （count）的value，即是该用户读取该群的消息数量
  - 该群的消息数量-用户已读该群的消息数量，即是用户在该群的未读消息数量
  - 并且调用CGroupMessageModel::getLastMsg获取该群聊最新的消息，包括msgId、消息内容strMsgData

DB_PROXY::clearUnreadMsgCounter 清除未读消息数量

请求命令：CID_MSG_READ_ACK

回应命令：无

1. 请求结构

```SQL
message IMMsgDataReadAck{
  //cmd id:   0x0303
  required uint32 user_id = 1;      //发送此信令的用户id
  required uint32 session_id = 2;
  required uint32 msg_id = 3;
  required IM.BaseDefine.SessionType session_type = 4;
}
```

2. 回应结构 无
3. 处理逻辑

clearUnreadMsgCounter函数入口

- 具体执行：CUserModel::clearUserCounter
  - 单聊：删除 unread_user_id对应filed（peerId），使用pCacheConn->hdel("unread_" + int2string(nUserId), int2string(nPeerId));
  - 群聊：group_id + _im_group_msg对应的 count，将其设置成群组 group_id +_im_group_msg对应field

DB_PROXY::getMessageById 根据消息id批量获取消息

请求命令：CID_MSG_GET_BY_MSG_ID_REQ
回复命令：CID_MSG_GET_BY_MSG_ID_RES
问题：和getMessage有什么不同的使用场景？

1. 请求结构

```SQL
message IMGetMsgByIdReq{
  //cmd id:     0x030d
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2;
  required uint32 session_id = 3;
  repeated uint32 msg_id_list = 4; // 请求消息id列表
  optional bytes attach_data = 20;
}
```

2. 回应结构

```SQL
message IMGetMsgByIdRsp{
  //cmd id:   0x030e
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2;
  required uint32 session_id = 3; // 单聊是对端user_id，群聊是group_id
  repeated IM.BaseDefine.MsgInfo msg_list = 4; // 消息内容列表
  optional bytes attach_data = 20;
}
```

3. 处理逻辑

- DB_PROXY::getMessageById 入口函数
- 分SESSION_TYPE_SINGLE单聊和SESSION_TYPE_GROUP群聊，
  - 单聊调用CMessageModel::getMsgByMsgId：本质就是给定MsgId查询IMMessage_x消息表，使用"select * from " + "IMMessage_" + int2string(nRelateId % 8)+ " where relateId=" + int2string(nRelateId) + " and status=0 and msgId in (" + strClause + ") order by created desc, id desc **limit 100**"； strClause 为消息msgId，以“,”隔开，并限定每次最多获取100条消息。
  - 群聊调用CGroupMessageModel::getMsgByMsgId：本质就是给定MsgId查询IMMessage_x消息表，使用"select * from " + "**IMGroupMessage_**" + int2string(nGroupId % 8)+ " where groupId=" + int2string(nGroupId) + " and msgId in (" + strClause + ") and status=0 and created >= " + int2string(nUpdated) + " order by created desc, id desc **limit 100**"; strClause 为消息msgId，以“,”隔开，并限定每次最多获取100条消息。
- 然后将查询到的结果封装到IMGetMsgByIdRsp，回复请求者

DB_PROXY::getLatestMsgId 获取最新的消息id

请求命令：CID_MSG_GET_LATEST_MSG_ID_REQ

回复命令：CID_MSG_GET_LATEST_MSG_ID_RSP

1. 请求结构

```SQL
message IMGetLatestMsgIdReq{
  //cmd id:     0x030b
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2;
  required uint32 session_id = 3;
  optional bytes attach_data = 20;
}
```

2. 回应结构

```SQL
message IMGetLatestMsgIdRsp{
  //cmd id:   0x030c
  required uint32 user_id = 1;
  required IM.BaseDefine.SessionType session_type = 2;
  required uint32 session_id = 3;
  required uint32 latest_msg_id = 4;
  optional bytes attach_data = 20;
}
```

3. 处理逻辑

- DB_PROXY::getLatestMsgId入口
- 分单聊SESSION_TYPE_SINGLE和SESSION_TYPE_GROUP群聊
  - 单聊调用CMessageModel::getLastMsg，使用"select msgId,type,content from " + "IMMessage_" + int2string(nRelateId % 8)+ " force index (idx_relateId_status_created) where relateId= " + int2string(nRelateId) + " and status = 0 order by created desc, id desc limit 1";
  - 群聊调用CGroupMessageModel::getLastMsg，使用"select msgId, type,userId, content from " + "IMGroupMessage_" + int2string(nGroupId % 8)+ " where groupId = " + int2string(nGroupId) + " and status = 0 order by created desc, id desc limit 1";
- 然后将查询到的结果封装到IMGetLatestMsgIdReq，回复请求者

8.5 群组

DB_PROXY::getNormalGroupList 获取用户加入的群组ID

请求命令：CID_GROUP_NORMAL_LIST_REQUEST

回复命令：CID_GROUP_NORMAL_LIST_RESPONSE

请求获知用户已经加入的群组，因此返回响应的时候需要附带群组相关的ID，但要注意的是这里只获取了group_id和version，并不是群的所有信息。

1. 请求结构

```SQL
message IMNormalGroupListReq{
  //cmd id:     0x0401
  required uint32 user_id = 1;
  optional bytes attach_data = 20;
}
```

2. 回应结构

```SQL
message IMNormalGroupListRsp{
  //cmd id:     0x0402
  required uint32 user_id = 1;
  repeated IM.BaseDefine.GroupVersionInfo group_version_list = 2; // 群组信息列表
  optional bytes attach_data = 20;
}
```

GroupVersionInfo 只包含了group_id和**version**。

```SQL
message GroupVersionInfo{
  required uint32 group_id = 1;
  required uint32 version = 2;
}
```

3. 处理逻辑

- DB_PROXY::getNormalGroupList入口函数
- 调用CGroupModel::getUserGroup获取相应的群GroupVersionInfo
  - 先调用CGroupModel::getUserGroupIds获取**group_id列表**，实质是查询IMGroupMember表，使用"select groupId from IMGroupMember where userId=" + int2string(nUserId) + " and status = 0 order by updated desc, id desc";
  - 然后通过获取的的group_id调用CGroupModel::getGroupVersion查询对应的信息，实质查询IMGroup表，使用 strSql = "select id,version from IMGroup where id in (" + strClause + ")";

if(0 != nGroupType) // 是否限定群类型

```
{
    strSql += " and type="+int2string(nGroupType);
}
strSql += " order by updated desc";
```

- 然后将查询到的结果封装到IMGetLatestMsgIdReq，回复请求者

DB_PROXY::getGroupInfo 获取批量群组信息

请求命令：CID_GROUP_INFO_REQUEST

回复命令：CID_GROUP_INFO_RESPONSE

根据请求者提供的group_id list，然后返回对应的群信息

1. **请求结构**

SQL 复制代码

```sql
message IMGroupInfoListReq{
  //cmd id:      0x0403
  required uint32 user_id = 1;
  repeated IM.BaseDefine.GroupVersionInfo group_version_list = 2;
  optional bytes attach_data = 20;
}
```

2. **回应结构**

SQL 复制代码

```sql
message IMGroupInfoListRsp{
  //cmd id:      0x0404
  required uint32 user_id = 1;
  repeated IM.BaseDefine.GroupInfo group_info_list = 2;
  optional bytes attach_data = 20;
}
```

```
message GroupInfo{
  required uint32 group_id = 1;
  required uint32 version = 2;  // 群版本
  required string group_name = 3; // 群名字
  required string group_avatar = 4; // 群头像地址
  required uint32 group_creator_id = 5; // 群创建者
  required GroupType group_type = 6; // 1:普通群, 2:临时群
  required uint32 shield_status = 7; //1: shield  0: not shield
  repeated uint32 group_member_list = 8; // 成员列表
}
```

3. 处理逻辑

- DB_PROXY::getGroupInfo入口函数
- 解析请求者的group_id和version 列表，并调用CGroupModel::isValidateGroupId在redis cache验证group_id的合法性
- 调用CGroupModel::getGroupInfo获取对应group_id的具体信息，实质是以group_id列表查询IMGroup表，使用"select * from IMGroup where id in (" + strClause + ") order by updated desc"， 其中strClause 为对应group_id列表，以“,”隔开。
  - 这里特别需要注意的点，version的作用是什么？什么时候会修改到group_id对应的version? version实际是每次修改后都对其进行+1操作，以便客户端在拉取信息时可以使用本地的version和服务器的version进行对比，如果本地的version<服务器的version则说明群有更新。具体是在修改群成员时调用 CGroupModel::modifyGroupMember （增加、删除），然后调用CGroupModel::incGroupVersion，最终操作到MySQL数据库"update IMGroup set version=version+1 where id="+int2string(nGroupId)
  - 并将具体的群信息封装到list<IM::BaseDefine::GroupInfo，此时还没有包含群成员信息
  - 然后调用CGroupModel::fillGroupMember获取群成员信息，遍历group_id列表，调用CGroupModel::getGroupUser获取对应group_id的群成员，实质是从redis cache获取，即是获取group_member_ + group_id为key所在hash结果的field（field存储了群成员的user_id），
- 将群信息以及对应的群成员封装到IMGroupInfoListRsp，回复请求者。

DB_PROXY::createGroup 创建群组

请求命令：CID_GROUP_CREATE_REQUEST
回复命令：CID_GROUP_CREATE_RESPONSE

1. 请求结构

```
message IMGroupCreateReq{
  //cmd id:     0x0405
  required uint32 user_id = 1; // 创建者
  //默认是创建临时群，且客户端只能创建临时群
  required IM.BaseDefine.GroupType group_type = 2 [default =
GROUP_TYPE_TMP];
  required string group_name = 3;   // 群名
  required string group_avatar = 4; // 群头像
  repeated uint32 member_id_list = 5; // 群员列表
  optional bytes attach_data = 20;
}
```

2. 回应结构

```
message IMGroupCreateRsp{
  //cmd id:     0x0406
  required uint32 user_id = 1;
  required uint32 result_code = 2; // 返回创建结果
  optional uint32 group_id = 3; // 群ID
  required string group_name = 4; // 群名称
  repeated uint32 user_id_list = 5; // 对应的群成员
  optional bytes attach_data = 20;
}
```

3. 处理逻辑

- DB_PROXY::createGroup入口
- 从IMGroupCreateReq解析出user_id，group_name和member_id_list （member_id_list 解析后存储到set去，防止member_id重复）
- 调用CGroupModel::createGroup创建群，分（1）创建群；（2）重置群消息ID；（3）清空group_id对应的member；（4）将member插入group_id对应的群：
  - (1) 创建群：CGroupModel::insertNewGroup(uint32_t nUserId, const string& strGroupName, const string& strGroupAvatar, uint32_t nGroupType, uint32_t nMemberCnt, uint32_t& nGroupId)，主要还包含了群成员人数。实质是插入IMGroup数据库，使用"insert into IMGroup(`name`, `avatar`, `creator`, `type`,`**userCnt**`, `status`, `version`, `lastChated`, `updated`, `created`) "\
"values(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"
  - (2) 重置群消息ID：CGroupMessageModel::resetMsgId，实质是将redis cache对应的"group_msg_id_" + int2string(nGroupId) key对应的value置为0
  - (3) 清空group_id对应的member：调用CGroupModel::clearGroupMember，这里又包含了2部

分的清除，1是清除IMGroupMember表对应的数据；2是清除redis 对应的"group_member_" + int2string(nGroupId)。

- (4) 将member插入group_id对应的群：CGroupModel::insertNewMember，这里也包含2部分的插入，1是操作IMGroupMember数据库，2是操作redis更新member
  - **MySQL数据库操作：先使用** "select userId from IMGroupMember where groupId=" + int2string(nGroupId) + " and userId in (" + strClause + ")"查询要插入用户是否已经在IMGroupMember，如果已经存在则加入set<uint32_t> setHasUser；**然后使用**"update IMGroupMember set status=0, updated="+int2string(nCreated)+" where groupId=" + int2string(nGroupId) + " and userId in (" + strClause + ")";更新其操作时间；**最后**剔除掉已经在群里面的，使用"insert into IMGroupMember(`groupId`, `userId`, `status`, `created`, `updated`) values\
    (?,?,?,?,?)";插入对应的member
  - **Redis 缓存操作**：将member id 作为field，created time作为value设置到"group_member_"+int2string(nGroupId)对应的hash结果，pCacheConn->hset(strKey, int2string(*it), int2string(nCreated));

- 将相应的信息封装到IMGroupCreateRsp，返回给请求者。

DB_PROXY::modifyMember 修改群组成员

请求命令：CID_GROUP_CHANGE_MEMBER_REQUEST
回应命令：CID_GROUP_CHANGE_MEMBER_RESPONSE
增加或者删除，请求的时候需要附带对应的 member_id，可存在多个 member_id，所以使用repeated的方式描述member_id_list。

1. 请求结构

SQL | 复制代码

```
message IMGroupChangeMemberReq{
  //cmd id:     0x0407
  required uint32 user_id = 1;
  required IM.BaseDefine.GroupModifyType change_type = 2;//0x01:增加; 0x02:删除
  required uint32 group_id = 3;
  repeated uint32 member_id_list = 4;
  optional bytes attach_data = 20;
}
```

2. 回应结构

```SQL
message IMGroupChangeMemberRsp{
  //cmd id:     0x0408
  required uint32 user_id = 1;
  required IM.BaseDefine.GroupModifyType change_type = 2;
  required uint32 result_code = 3;
  required uint32 group_id = 4;
  repeated uint32 cur_user_id_list = 5; //现有的成员id
  repeated uint32 chg_user_id_list = 6; //变动的成员id,add: 表示添加成功的id, del: 表示删除的id
  optional bytes attach_data = 20;
}
```

需要注意的是：cur_user_id_list = 5; //现有的成员id

3. 处理逻辑

- DB_PROXY::modifyMember
- 增加或者删除具体都是调用CGroupModel::modifyGroupMember(uint32_t nUserId, uint32_t nGroupId, IM::BaseDefine::GroupModifyType nType, set<uint32_t>& setUserId, list<uint32_t>& lsCurUserId)
- 先检测修改权限：使用CGroupModel::hasModifyPermission，实质是使用"select creator, type from IMGroup where id="+ int2string(nGroupId)查询群创建者和群类型，如果是普通群GROUP_TYPE_NORMAL则需要群创建者creator才有权限增加群成员，如果是临时群GROUP_TYPE_TMP且是增加操作则其他成员都可以进行
- 增加成员： 调用CGroupModel::addMember(uint32_t nGroupId, set<uint32_t> &setUser, list<uint32_t>& lsCurUserId)
  - (1)先在redis缓存检测要增加的成员是否已经就在群里面，调用CGroupModel::removeRepeatUser，实质去去查询"group_member_"+int2string(nGroupId)对应的hash，如果已经存在则从setUser剔除
  - (2)插入新成员：调用 CGroupModel::insertNewMember，具体参考创建群时的操作
  - (3)获取目前群所有的成员：调用CGroupModel::getGroupUser，实质是从redis缓存里面读取，操作 "group_member_" + int2string(nGroupId)对应的hash。
  - 退出CGroupModel::addMember时，，set<uint32_t> &setUser保存的是增加或者删除member id，list<uint32_t>& lsCurUserId保存的是当前群组的所有id。
- 删除成员：调用CGroupModel::removeMember(uint32_t nGroupId, set<uint32_t> &setUser, list<uint32_t>& lsCurUserId)，并且紧接着调用CGroupModel::removeSession，将要删除的member id对应的会话都删除
  - (1)先从MySQL数据库删除：根据member id使用"update IMGroupMember set status=1 where groupId =" + int2string(nGroupId) + " and userId in(" + strClause + ")";
  - (2)再从redis缓存删除： 操作"group_member_"+ int2string(nGroupId)，删除member id对应

的field
    - (3)获取目前群所有的成员：调用CGroupModel::getGroupUser，实质是从redis缓存里面读取，操作 "group_member_" + int2string(nGroupId)对应的hash。
    - 退出CGroupModel::removeMember时，，set<uint32_t> &setUser保存的是增加或者删除member id，list<uint32_t>& lsCurUserId保存的是当前群组的所有id。
- 不管是增加还是删除成员，则对应群的version版本都进行+1操作，调用CGroupModel::incGroupVersion
- 并情况 要增加或者删除成员对应的未读计数器，调用CUserModel::clearUserCounter(uint32_t nUserId, uint32_t nPeerId, IM::BaseDefine::SessionType nSessionType)，理论上该：增加时清除未读计数器；删除时应该将其对应的计数器删除掉。
- 将结果封装到IMGroupChangeMemberRsp回复请求者

8.6 文件传输

DB_PROXY::hasOfflineFile是否有离线文件

请求命令：CID_FILE_HAS_OFFLINE_REQ

回复命令：CID_FILE_HAS_OFFLINE_RES

1. 请求结构

SQL | 复制代码

```
message IMFileHasOfflineReq{
  //cmd id:   0x0509
  required uint32 user_id = 1;
  optional bytes attach_data = 20;
}
```

2. 回应结构

SQL | 复制代码

```
message IMFileHasOfflineRsp{
  //cmd id: 0x050a
  required uint32 user_id = 1;
  repeated IM.BaseDefine.OfflineFileInfo offline_file_list = 2;
  repeated IM.BaseDefine.IpAddr ip_addr_list = 3;
  optional bytes attach_data = 20;
}
```

OfflineFileInfo相应的信息存储到MySQL数据库

```SQL
message OfflineFileInfo{
    required uint32 from_user_id = 1;
    required string task_id = 2;
    required string file_name = 3;
    required uint32 file_size = 4;
}
```

```SQL
message IpAddr{
  required string ip = 1;
  required uint32 port = 2;
}
```

3. 处理逻辑

- DB_PROXY::hasOfflineFile
- 调用CFileModel::getOfflineFile(uint32_t userId, list<IM::BaseDefine::OfflineFileInfo>& lsOffline)获取离线文件信息，离线文件可以有多个。实质是查询"select * from IMTransmitFile where told="+int2string(userId) + " and status=0 order by created"; 通过told 查询是否属于自己的离线文件。
- 将结果封装到IMFileHasOfflineRsp回复请求者

DB_PROXY::addOfflineFile 加入离线文件

请求命令：CID_FILE_ADD_OFFLINE_REQ

回复命令：无

1. 请求结构

```SQL
message IMFileAddOfflineReq{
  //cmd id: 0x050b
  required uint32 from_user_id = 1;
  required uint32 to_user_id = 2;
  required string task_id = 3;
  required string file_name = 4;
  required uint32 file_size = 5;
}
```

2. **回应结构**

   无

3. **处理逻辑**

- DB_PROXY::addOfflineFile
- 具体调用CFileModel::addOfflineFile(uint32_t fromId, uint32_t toId, string& taskId, string& fileName, uint32_t fileSize)，使用"insert into IMTransmitFile (`fromId`,`toId`,`fileName`,`size`,`taskId`,`status`,`created`,`updated`) values(?,?,?,?,?,?,?,?)";

DB_PROXY::delOfflineFile 删除离线文件

请求命令：CID_FILE_DEL_OFFLINE_REQ

回复命令：无

1. 请求结构

```SQL
message IMFileDelOfflineReq{
  //cmd id: 0x050c
  required uint32 from_user_id = 1;
  required uint32 to_user_id = 2;
  required string task_id = 3;
}
```

2. 回应结构
3. 处理逻辑

- DB_PROXY::delOfflineFile入口
- 具体调用CFileModel::delOfflineFile(uint32_t fromId, uint32_t toId, string& taskId)，使用"delete from IMTransmitFile where fromId=" + int2string(fromId) + " and toId="+int2string(toId) + " and taskId='" + taskId + "'";

9 附录

SIGTERM信号

1.kill pid、kill –15 pid 、kill –SIGTERM pid

系统会发送一个SIGTERM的信号给对应的程序。当程序接收到该signal后，将会发生以下的事情（取决于对信号的处理）

- 程序立刻停止
- 当程序释放相应资源后再停止
- 程序可能仍然继续运行

大部分程序接收到SIGTERM信号后，会先释放自己的资源，然后在停止。但是也有程序可以在接受到信号量后，做一些其他的事情，并且这些事情是可以配置的。如果程序正在等待IO，可能就不会立马做出相应。也就是说，SIGTERM多半是会被阻塞的、忽略。