## conn ------🡪 XMPPConnection长连接对象

ImApp------🡪保存连接对象conn、用户名信息，并且继承了Application的子类，用于为其他页面提供直接获取长连接conn

## 多个页面之间进行跳转后，需要共同使用连接对象conn和服务器进行相应的操作，所以在conn连接建立好之后，需要将该对象进行保存，如保存在一个Application的子类ImApp中，在其他页面直接通过getApplication方法然后强转后拿到该子类对象，从而获取该连接对象conn，也需要保存用户名等其他信息

## 客户端

### 1.连接服务器（在应用程序打开欢迎界面的时候）

通过连接配置对象配置服务器ip、端口号、服务器主机名信息，然后通过连接对象进行服务的连接，此操作在子线程中运行

// 创建连接配置对象，设置服务器主机IP、端口号、服务器名

config = **new** ConnectionConfiguration("192.168.1.106", 5222, "michael-chun");

conn = **new** XMPPConnection(config);

// 连接服务器

conn.connect();

### 2.提交用户名密码进行登录（用户点击登录按钮执行）

通过连接对象调用login方法，子线程中执行

conn.login(username, password);

### 3.用户登录成功后，从服务器获取联系人好友信息

1.在其他页面直接获取conn长连接

ImApp imApp = (ImApp) getApplication();

XMPPConnection conn = imApp.getConn();

2.通过长连接调用获取用户联系人的方法

// 1.获取包含所有的用户的花名册

Roster roster = conn.getRoster();

// 2.获取所有的联系人信息

Collection<RosterEntry> entries = roster.getEntries();

// 3.将获取的所有联系人信息保存在集合中

List<RosterEntry> entrys = **new** ArrayList<>(entries);

// 4.将获取的所有联系人集合传递给适配器，通过适配器去填充显示当前用户列表的ListView

==========================================================

获取所有联系人的分组

List<RosterGroup> groups = **new** ArrayList<RosterGroup>(roster.getGroups());

根据联系人分组获取到所有的联系人

**for**(RosterGroup group : groups){

System.***out***.println("联系人组名是： " + group.getName()+"=============");

List<RosterEntry> entryes = **new** ArrayList<>(group.getEntries());

**for**(RosterEntry entry : entrys){

System.***out***.println("用户名 ： " + entry.getName() + "昵称 ：" + entry.getUser() + "用户当前状态: "+ entry.getStatus());

}

}

## 监听好友的状态，并更新好友列表

/\*\*

1.获取好友花名册 （子线程中执行）

// 获取包含所有的用户的花名册

Roster roster = imApp.getConn().getRoster();

2.创建ListView的适配器，为ListView设置适配器 （主线程中）

**private** ContactsListAdapter contactsListAdapter ;

contactsListAdapter = **new** ContactsListAdapter(entrysList, ContactListActivity.**this**); contacts\_list.setAdapter(contactsListAdapter);

\* 3.为花名册创建并设置监听器

/\*\*

\* 创建一个用于监听 保存了好友列表的花名册的监听器，只要花名册中的用户上下线信息、账号信息发生变化都会触发该监听器

\* A listener that is fired any time a roster is changed or

\* the presence of a user in the roster is changed.

\*/

RosterListener rosterListener = **new** RosterListener() {

// 花名册中好友上下线信息发生变动的回调方法

@Override

**public** **void** presenceChanged(Presence presence) {

updateContactsList();

}

// 花名册中好友账号信息发生变动的回调方法,如修改了备注，其他设备也需要同步

@Override

**public** **void** entriesUpdated(Collection<String> addresses) { updateContactsList();

}

// 花名册中好友被删除时的回调方法

@Override

**public** **void** entriesDeleted(Collection<String> addresses) {

updateContactsList();

}

// 花名册中新添加了好友的回调方法

@Override

**public** **void** entriesAdded(Collection<String> addresses) {

updateContactsList();

}

};

\* 4.创建用于更新花名册联系人的方法，在主线程中用适配器更新好友列表，在子线程中获取好友 花名册 Roster

\*/

/\*\*

\* 更新好友列表的方法

\*/

**private** **void** updateContactsList() { ThreadUtils.*runInThread*(**new** Runnable() {

@Override

**public** **void** run() {

// 获取包含所有的用户的花名册

Roster roster = imApp.getConn().getRoster();

// 为花名册设置监听器，监听花名册中好友的上下线、账号信息变化以及删除、添加好友

roster.addRosterListener(rosterListener);

// 每次更新添加当前花名册联系人时，清除上次list集合中的所有联系人，否则会造成数据重复显示

entrysList.clear();

entrysList.addAll(roster.getEntries());

// 在主线程中更新用户列表

ThreadUtils.*runUnThread*(**new** Runnable() {

@Override

**public** **void** run() { **if**(contactsListAdapter == **null**){

contactsListAdapter = **new** ContactsListAdapter(entrysList, ContactListActivity.**this**); contacts\_list.setAdapter(contactsListAdapter);

}**else**{ contactsListAdapter.notifyDataSetChanged();

}

}

});

}

});}

|  |  |
| --- | --- |
| Message | 消息对象 |
| Chat | 聊天工具 发送接收数据 |
| ChatManager | 获取聊天工具的消息管理器 |
| MessageListener | 接收消息 的监听器 |

## 发送、接收消息

1.通过长连接获取聊天工具管理器

chatManager = imApp.getConn().getChatManager();

2.通过聊天工具管理器获取聊天工具，并为其设置接收消息的监听器，实现处理消息的

chat = chatManager.createChat(toAccount, **null**);

3.发送信息

chat.sendMessage(content);

4.创建监听好友发来消息的监听器

/\*\*

\*监听接收到好友发来的消息的监听器

\*/

MessageListener msgListener = **new** MessageListener() {

@Override

**public** **void** processMessage(Chat chat,**final** Message message) {

ThreadUtils.*runUnThread*(**new** Runnable() {

@Override

**public** **void** run() {

// 打印封装好的消息的xml内容

System.***out***.println(message.toXML());

// 此处执行处理接收到的新的消息的操作

**if**(message.getBody() != **null**){

Toast.*makeText*(ChatActivity.**this**, message.getBody(), 0).show();

}

}

});

}

};

5.设置监听好友发来消息的监听器

chat.addMessageListener(msgListener);

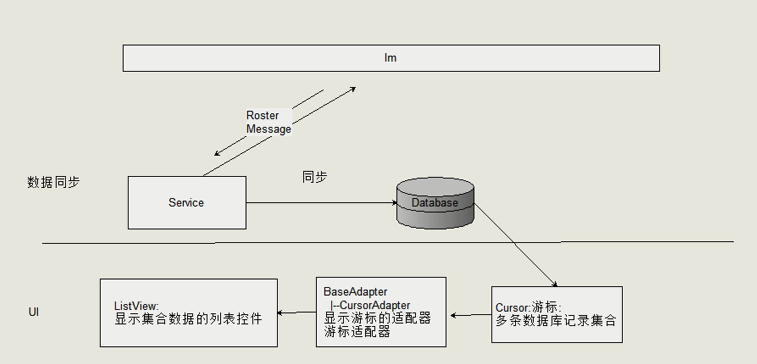
## 在聊天界面显示消息记录

在发送、接收消息的时候将消息对象保存到List集合中，每次发送、接收新的消息的时候就通过适配器Adapter拿着List去更新界面（notifyDataSetChanged）

适配器中要更新两种消息的View，一种是发送的，另一种是接收到的，在适配器内进行具体的实现，具体请见ChatMessageAdapter类中的实现

===========================================================

QQ实现消息列表和用户信息变化的刷新的原理图如下，等待时间很少，用户几乎感觉不到数据的加载



关闭聊天界面时，也需要去监听用户联系人信息及消息的变化，创建核心服务在后台进行监听，有变化则将数据同步到数据库中，再将数据库中的信息刷新到界面

在核心服务中完成Roster联系人花名册的获取，及所有联系人信息的入库，以及更新界面操作

1.GreenDao框架自动生成用户联系人Contact、ContactDao、DaoMaster、DaoSession类

2.获取Roster花名册中所有的联系人的RosterEntry实体类

3.每个联系人对应的的实体类RosterEntry中的属性，封装生成Contact类，并保存到数据库中，在核心服务中进行代码实现，代码如下：

rosterEntryList = **new** ArrayList<RosterEntry>(imApp.getConn().getRoster().getEntries());

// 将好友列表中的所有好友信息保存到数据库中

**for**(RosterEntry entry : rosterEntryList){

System.***out***.println("EntryName : "+entry.getName() + "EntryUser : "+entry.getUser());

String nick = NickUtils.*getNick*(entry.getUser(), entry.getName());

Contact entity = **new** Contact();

entity.setAccount(entry.getUser());

entity.setNick(nick); entity.setAvator(R.drawable.***ic\_launcher***); entity.setSpell(PinYinUtils.*getPinYin*(nick));

// 查询数据库中是否保存了该好友的信息

List<Contact> resultList = contactDao.queryRaw("where ACCOUNT = ?", entry.getUser());

**if**(resultList.size() == 0){

// 没有保存，则直接保存好友的信息

contactDao.insert(entity);

System.***out***.println("直接保存：======= "+entity.toString());

}**else**{

// 通过ID查找出已经保存的联系人信息记录

Contact existedContact = contactDao.load(resultList.get(0).getId()); existedContact.setNick(entity.getNick()); existedContact.setSpell(entity.getSpell()); existedContact.setAvator(entity.getAvator());

// 已经保存了好友的信息，则更新好友的信息 contactDao.insertOrReplace(existedContact);

System.***out***.println("修改保存：======= existedContact ： "+ "Nick : " +existedContact.getNick() +"Account : "+existedContact.getAccount() );

}

}

4.在ImApp对象创建的时候，获取联系人数据库对象，并作为成员变量保存起来

sqliteDatabase = **new** DaoMaster.DevOpenHelper(**this**, "im.db", **null**).getWritableDatabase();

DaoMaster master = **new** DaoMaster(sqliteDatabase);

DaoSession session = master.newSession();

contactDao = session.getContactDao();

5.打开联系人列表界面时，查询联系人数据库，获取结果集Cursor，传递给适配器

/\*\*

\* 更新好友列表的方法

\*/

**private** **void** updateContactsList() {

SQLiteDatabase sqliteDatabase = imApp.getSqliteDatabase();

// 根据昵称的拼音的首字母，对查询到的结果进行排序

Cursor cursor = sqliteDatabase.rawQuery("select \* from CONTACT order by spell ASC ;", **new** String[]{});

contactsListAdapter = **new** ContactsListAdapter(**this**, cursor, **true**);

contacts\_list.setAdapter(contactsListAdapter);

}

6.创建CursorAdapter适配器对象，对接收到的Cursor结果集进行处理，将每条数据显示到界面上

CursorAdapter的内部实现

@Override

**public** View getView(**int** position, View convertView, ViewGroup parent) {

Cursor cursor = getCursor();

cursor.moveToPosition(position);

String nick = cursor.getString(cursor.getColumnIndex("NICK"));

String account = cursor.getString(cursor.getColumnIndex("ACCOUNT"));

**int** avator = cursor.getInt(cursor.getColumnIndex("AVATOR"));

View view = **null**;

ViewHolder holder = **null** ;

**if**(convertView != **null** ){

view = convertView ;

holder = (ViewHolder) view.getTag() ;

}**else**{

view = View.*inflate*(context , R.layout.***buddy\_list\_item***, **null**);

holder = **new** ViewHolder() ;

view.setTag(holder);

}

holder.contact\_photo = (ImageView) view.findViewById(R.id.***contact\_photo***);

holder.contact\_nickname = (TextView) view.findViewById(R.id.***contact\_nickname***);

holder.contact\_desc = (TextView) view.findViewById(R.id.***contact\_desc***);

holder.contact\_nickname.setText(nick);

holder.contact\_desc.setText(account);

holder.contact\_photo.setImageResource(avator);

**return** view ;

}

**private** **class** ViewHolder {

**private** ImageView contact\_photo ;

**private** TextView contact\_nickname ;

**private** TextView contact\_desc ;

} @Override

**public** View newView(Context context, Cursor cursor, ViewGroup parent) {

**return** **null**;

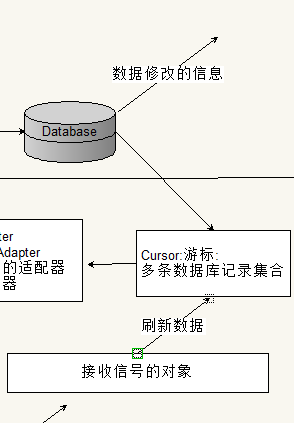
}

@Override

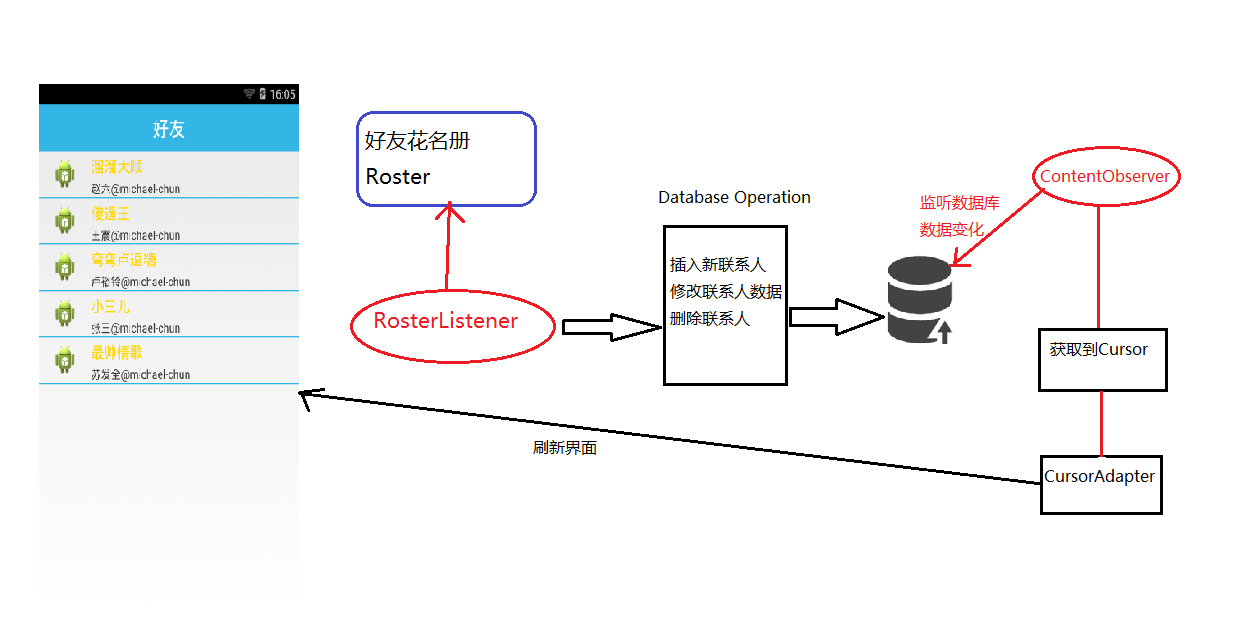
**public** **void** bindView(View view, Context context, Cursor cursor) {

}

## 设置Roster监听器，监听到新增、删除联系人，或修改好友信息（备注、昵称等）



Roster花名册联系人信息发生变化----🡪RosterListener监听到Roster数据发生变化 ---🡪更新数据库中保存的联系人信息-🡪数据库库信息发生变化--🡪ContentOberser观察到数据库信息发生变化--🡪重新获取数据库中的联系人信息,拿到结果集Cursor--🡪将结果集Cursor交给用于显示联系人的ListView的适配器CursorAdapter---🡪CursorAdapter拿到新的Cursor，更新界面联系人列表，过程如下图所示：



1.监听Roster花名册联系人信息的变化，将数据同步到数据库

2.设置数据库信息发生变化的监听器，通知界面刷新

1)在核心服务打开的时候将联系人数据保存到数据库中

roster = imApp.getConn().getRoster();

// 为roster设置监听器

roster.addRosterListener(rosterListener);

contactDao = imApp.getContactDao();

ThreadUtils.*runInThread*(**new** Runnable() {

@Override

**public** **void** run() {

rosterEntryList = **new** ArrayList<RosterEntry>(roster.getEntries());

// 将好友列表中的所有好友信息保存到数据库中

**for**(RosterEntry entry : rosterEntryList){

insertOrUpdateDBContact(entry);

}

}

});

2)创建Roster监听器

/\*\*

\* 监听联系人信息发生变化的监听器，每次有响应的事件触发时，同步数据到数据库

\*/

**private** RosterListener rosterListener = **new** RosterListener(){

@Override

**public** **void** entriesAdded(Collection<String> addresses) {

// 将Roster中新添加的用户数据保存到数据库中

**for**(String account : addresses){

System.***out***.println("=========entriesAdded==========");

System.***out***.println("addressItem : " + account);

RosterEntry addedEntry = roster.getEntry(account);

insertOrUpdateDBContact(addedEntry);

}

}

@Override

**public** **void** entriesUpdated(Collection<String> addresses) {

// Roster中的某条联系人信息发生了修改，同步跟新数据库中的联系人信息

**for**(String account : addresses){

System.***out***.println("=========entriesUpdated==========");

System.***out***.println("addressItem : " + account);

RosterEntry entry = roster.getEntry(account);

insertOrUpdateDBContact(entry);

}

}

@Override

**public** **void** entriesDeleted(Collection<String> addresses) {

// Roster中联系人删除了，数据库中的联系人的表中也要删除被删除的联系人对应的那条记录

**for**(String account : addresses){

System.***out***.println("=========entriesDeleted==========");

System.***out***.println("addressItem : " + account);

SQLiteDatabase db = imApp.getSqliteDatabase();

db.execSQL("delete from CONTACT where ACCOUNT = ?;", **new** String[]{account});

}

Uri uri = Uri.*parse*("content://"+ContactDao.**class**.getSimpleName());

// 通知所有的监听上面的uri的内容观察者 数据库数据发生了变化

getContentResolver().notifyChange(uri, **null**);

}

@Override

**public** **void** presenceChanged(Presence presence) {

Toast.*makeText*(CoreService.**this**, "someOne login or logout", 0).show();

}

};

3)同步数据到数据库

/\*\*

\* 更新数据库中保存联系人的表中的数据，没有则插入，有则更新

\* **@param** entry

\*/

**private** **void** insertOrUpdateDBContact(RosterEntry entry) {

System.***out***.println("EntryName : "+entry.getName() + "EntryUser : "+entry.getUser());

String nick = NickUtils.*getNick*(entry.getUser(), entry.getName());

Contact entity = **new** Contact();

entity.setAccount(entry.getUser());

entity.setNick(nick);

entity.setAvator(R.drawable.***ic\_launcher***);

entity.setSpell(PinYinUtils.*getPinYin*(nick));

// 查询数据库中是否保存了该好友的信息

List<Contact> resultList = contactDao.queryRaw("where ACCOUNT = ?", entry.getUser());

**if**(resultList.size() == 0){

// 没有保存，则直接保存好友的信息

contactDao.insert(entity);

System.***out***.println("直接保存：======= "+entity.toString());

}**else**{

// 通过ID查找出已经保存的联系人信息记录

Contact existedContact = contactDao.load(resultList.get(0).getId());

existedContact.setNick(entity.getNick());

existedContact.setSpell(entity.getSpell());

existedContact.setAvator(entity.getAvator());

// 已经保存了好友的信息，则更新好友的信息

contactDao.insertOrReplace(existedContact);

System.***out***.println("修改保存：======= existedContact ： "+ "Nick : " +existedContact.getNick() +"Account : "+existedContact.getAccount() );

}

Uri uri = Uri.*parse*("content://"+ContactDao.**class**.getSimpleName());

// 通知所有的监听上面的uri的内容观察者 数据库数据发生了变化

getContentResolver().notifyChange(uri, **null**);

}

4)设置内容观察者，监听数据库数据变化

5)在显示联系人列表的Activity中，创建内容观察者

**private** ContentObserver observer = **new** ContentObserver(**new** Handler()) {

@Override

**public** **void** onChange(**boolean** selfChange) {

**super**.onChange(selfChange);

}

@SuppressLint("NewApi")

@Override

**public** **void** onChange(**boolean** selfChange, Uri uri) {

**super**.onChange(selfChange, uri);

updateContactsList();

}

};

6)注册内容观察者

Uri uri = Uri.*parse*("content://"+ContactDao.**class**.getSimpleName());

getContentResolver().registerContentObserver(uri, **true**, observer);

7)更新好友列表

/\*\*

\* 更新好友列表的方法

\*/

**private** **void** updateContactsList() {

**if**(contactsListAdapter == **null**){

SQLiteDatabase sqliteDatabase = imApp.getSqliteDatabase();

// 根据昵称的拼音的首字母，对查询到的结果进行排序

Cursor cursor = sqliteDatabase.rawQuery("select \* from CONTACT order by spell ASC ;", **new** String[]{});

contactsListAdapter = **new** ContactsListAdapter(**this**, cursor, **true**);

contacts\_list.setAdapter(contactsListAdapter);

}**else**{

SQLiteDatabase sqliteDatabase = imApp.getSqliteDatabase();

// 根据昵称的拼音的首字母，对查询到的结果进行排序

Cursor newCursor = sqliteDatabase.rawQuery("select \* from CONTACT order by spell ASC ;", **new** String[]{});

Cursor oldCursor = contactsListAdapter.swapCursor(newCursor);

oldCursor.close();

}

}

8)继承了CursorAdapter的，用于向联系人列表提供数据的适配器类的主要内部实现方法，newView和bindView是空实现，不用实现

@Override

**public** View getView(**int** position, View convertView, ViewGroup parent) {

Cursor cursor = getCursor();

cursor.moveToPosition(position);

String nick = cursor.getString(cursor.getColumnIndex("NICK"));

String account = cursor.getString(cursor.getColumnIndex("ACCOUNT"));

**int** avator = cursor.getInt(cursor.getColumnIndex("AVATOR"));

View view = **null**;

ViewHolder holder = **null** ;

**if**(convertView != **null** ){

view = convertView ;

holder = (ViewHolder) view.getTag() ;

}**else**{

view = View.*inflate*(context , R.layout.***buddy\_list\_item***, **null**);

holder = **new** ViewHolder() ;

view.setTag(holder);

}

holder.contact\_photo = (ImageView) view.findViewById(R.id.***contact\_photo***);

holder.contact\_nickname = (TextView) view.findViewById(R.id.***contact\_nickname***);

holder.contact\_desc = (TextView) view.findViewById(R.id.***contact\_desc***);

holder.contact\_nickname.setText(nick);

holder.contact\_desc.setText(account);

holder.contact\_photo.setImageResource(avator);

**return** view ;

}

**private** **class** ViewHolder {

**private** ImageView contact\_photo ;

**private** TextView contact\_nickname ;

**private** TextView contact\_desc ;}

## 保存消息记录到数据库

一.发送聊天消息时，将聊天消息对象入库

二.接收到好友发来的消息时，将消息对象入库

具体步骤：

1.通过GreenDao创建聊天消息的实体类、数据库操作的Dao

Schema schema = **new** Schema(1, "com.jgsu.imessage.dao") ;

//联系人实体

Entity entity = schema.addEntity("Contact");

entity.addStringProperty("account");

entity.addStringProperty("nick");

entity.addIntProperty("avator");

entity.addStringProperty("spell");

entity.addIdProperty().primaryKey().autoincrement() ;

//消息记录实体

Entity entity2 = schema.addEntity("MsgRecord");

entity2.addIdProperty().primaryKey().autoincrement() ;

entity2.addStringProperty("session\_id");

entity2.addStringProperty("session\_name");

entity2.addIntProperty("avator");

entity2.addStringProperty("from\_id");

entity2.addStringProperty("from\_nick");

entity2.addIntProperty("from\_avator");

entity2.addStringProperty("content");

entity2.addStringProperty("msg\_type");

entity2.addStringProperty("status");

entity2.addStringProperty("unread");

entity2.addLongProperty("time");

**try** {

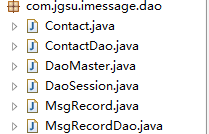
**new** DaoGenerator().generateAll(schema, "../I信/src");

}**catch** (Exception e) {

e.printStackTrace();

}

创建好的实体类及其Dao



2.通过GreenDao创建数据库操作的Dao，保存在ImApp类中

sqliteDatabase = **new** DaoMaster.DevOpenHelper(**this**, "im.db", **null**).getWritableDatabase();

DaoMaster master = **new** DaoMaster(sqliteDatabase);

DaoSession session = master.newSession();

// 数据库中联系人所在的表的数据操作的 DAO

contactDao = session.getContactDao();

// 数据库中消息记录所在的表的数据操作的 DAO

msgRecordDao = session.getMsgRecordDao();

### 发送消息入库

3.在核心服务中，创建一个保存聊天工具Chat的集合chatSet，并在服务中实现发送消息的sendMsg方法，创建并设置监听收到好友消息的监听器，sendMsg方法中，发送完消息后，将消息入库

/\*\*

\* 用于保存和其他好友进行聊天的聊天工具Chat对象的集合

\*/

**private** HashMap<String ,Chat> chatSet = **new** HashMap<String ,Chat>();

**public** **void** sendMsg(Message msg) **throws** XMPPException{

Chat chat = chatSet.get(msg.getTo());

**if**(chat == **null**){

chat = imApp.getConn().getChatManager().createChat(msg.getTo(), **null**);

chatSet.put(msg.getTo(), chat);

// 设置聊天监听器，监听好友发来的消息

chat.addMessageListener(msgListener);

}

chat.sendMessage(msg);

MsgRecord entity = **new** MsgRecord();

entity.setFrom\_id(msg.getFrom());

entity.setFrom\_nick(NickUtils.*getNick*(msg.getFrom(), **null**));

entity.setSession\_id(msg.getTo());

entity.setSession\_name(msg.getSubject());

entity.setAvator(R.drawable.***imessage***);

entity.setContent(msg.getBody());

entity.setMsg\_type(msg.getType().toString());

entity.setTime(System.*currentTimeMillis*());

// 将已经发送的消息保存到数据库中

imApp.getMsgRecordDao().insert(entity);

System.***out***.println("==========消息记录入库============");

}

/\*\*

\*监听接收到好友发来的消息的监听器

\*/

MessageListener msgListener = **new** MessageListener() {

@Override

**public** **void** processMessage(Chat chat,**final** Message message) {

ThreadUtils.*runUnThread*(**new** Runnable() {

@Override

**public** **void** run() {

**if**(message.getBody() != **null**){

System.***out***.println("接收到了消息了！！！！=================》》》》》");

// 接收到消息时，将消息对象保存到List集合中，通过适配器通知界面更新消息列表

message.setSubject(TimeUtils.*getCurrentTime*());

message.setTo(NickUtils.*getCorrectedUserName*(message.getTo()));

message.setFrom(NickUtils.*getCorrectedUserName*(message.getFrom()));

MsgRecord record = **new** MsgRecord();

record.setSession\_id(message.getFrom());

record.setAvator(R.drawable.***imessage***);

record.setContent(message.getBody());

record.setFrom\_id(message.getFrom());

Cursor cursor = imApp.getSqliteDatabase().rawQuery("select \* from CONTACT where ACCOUNT = ? ;", **new** String[]{message.getFrom()});

cursor.moveToFirst();

String nick = cursor.getString(cursor.getColumnIndex("NICK"));

record.setFrom\_nick(nick);

record.setSession\_name(nick);

record.setMsg\_type(message.getType().toString());

record.setTime(System.*currentTimeMillis*());

imApp.getMsgRecordDao().insert(record);

// 告诉内容观察者消息记录数据库信息发生了变化，通知界面更新

Uri uri = Uri.*parse*("content://"+MsgRecordDao.**class**.getSimpleName());

getContentResolver().notifyChange(uri, **null**);

}

}

});

}

};

4.在聊天界面中调用上一步实现的发送消息的方法，由于上述方法在核心服务中实现，所以在ImApp创建的时候保存一个核心服务的对象，实现set、get方法，在核心服务被创建的时候，将自己设置到ImApp类中，保存自己的引用，供其他实现模块调用

核心服务的onCreate方法中：// 将核心服务对象（当前类的实例）保存到ImApp类中

imApp.setCoreService(**this**);

聊天界面的发送消息的按钮的点击事件中调用sendMsg方法：

// 调用在核心服务CoreService中实现的发送消息+保存消息记录到数据库的方法

imApp.getCoreService().sendMsg(msg);

### 接收的消息入库

1.在核心服务中创建的接收消息的监听器中实现消息的入库

/\*\*

\*监听接收到好友发来的消息的监听器

\*/

MessageListener msgListener = **new** MessageListener() {

@Override

**public** **void** processMessage(Chat chat,**final** Message message) {

ThreadUtils.*runUnThread*(**new** Runnable() {

@Override

**public** **void** run() {

**if**(message.getBody() != **null**){

System.***out***.println("接收到了消息了！！！！=================》》》》》");

// 接收到消息时，将消息对象保存到List集合中，通过适配器通知界面更新消息列表

message.setSubject(TimeUtils.*getCurrentTime*());

message.setTo(NickUtils.*getCorrectedUserName*(message.getTo()));

message.setFrom(NickUtils.*getCorrectedUserName*(message.getFrom()));

MsgRecord record = **new** MsgRecord();

record.setSession\_id(message.getFrom());

record.setAvator(R.drawable.***imessage***);

record.setContent(message.getBody());

record.setFrom\_id(message.getFrom());

Cursor cursor = imApp.getSqliteDatabase().rawQuery("select \* from CONTACT where ACCOUNT = ? ;", **new** String[]{message.getFrom()});

cursor.moveToFirst();

String nick = cursor.getString(cursor.getColumnIndex("NICK"));

record.setFrom\_nick(nick);

record.setSession\_name(nick);

record.setMsg\_type(message.getType().toString());

record.setTime(System.*currentTimeMillis*());

imApp.getMsgRecordDao().insert(record);

// 告诉内容观察者消息记录数据库信息发生了变化，通知界面更新

Uri uri = Uri.*parse*("content://"+MsgRecordDao.**class**.getSimpleName());

getContentResolver().notifyChange(uri, **null**);

}

}

});

}

};

## 实现动态显示消息记录的动态更新

思路：创建内容观察者，观察数据库的变化，如果保存聊天记录的数据库发生了变化，则再次查询数据库，更新界面

1.在发送、接收到消息，执行入库操作之后，通过内容观察者发送一个数据变化消息

imApp.getMsgRecordDao().insert(record);

// 告诉内容观察者消息记录数据库信息发生了变化，通知界面更新

Uri uri = Uri.*parse*("content://"+MsgRecordDao.**class**.getSimpleName());

getContentResolver().notifyChange(uri, **null**);

2.聊天界面的Activity中创建一个内容观察者，并在观察者的onChange方法中实现重新查询数据库，更新聊天列表的操作

/\*\*

\* 监听数据库中消息记录内容发生变化的内容观察者

\*/

**private** ContentObserver observer = **new** ContentObserver(**new** Handler()) {

@Override

**public** **void** onChange(**boolean** selfChange) {

**super**.onChange(selfChange);

}

@SuppressLint("NewApi")

@Override

**public** **void** onChange(**boolean** selfChange, Uri uri) {

**super**.onChange(selfChange, uri);

updateMsgListView();

}

};

4.查询数据库与更新消息列表的updateMsgListView的具体实现

/\*\*

\* 更新消息列表

\*/

**private** **void** updateMsgListView(){

String sql = "select \* from MSG\_RECORD where SESSION\_ID = ? order by TIME ASC";

**if**(msgListAdapter == **null**){

Cursor cursor = imApp.getSqliteDatabase().rawQuery(sql, **new** String[]{toAccount});

msgListAdapter = **new** ChatMessageAdapter(**this**, cursor, **true**);

*lv\_chat\_content*.setAdapter(msgListAdapter);

**if**(cursor.getCount() > 0){

*lv\_chat\_content*.setSelection(cursor.getCount() - 1);

}

}**else**{

Cursor newCursor = imApp.getSqliteDatabase().rawQuery(sql, **new** String[]{toAccount});

Cursor oldCursor = msgListAdapter.swapCursor(newCursor);

oldCursor.close();

**if**(oldCursor.getCount() > 0){

*lv\_chat\_content*.setSelection(oldCursor.getCount() - 1);

}

}

}

5.在聊天界面创建的时候注册内容观察者，使内容观察者生效，在关闭时反注册

注册：

uri = Uri.*parse*("content://"+MsgRecordDao.**class**.getSimpleName());

getContentResolver().registerContentObserver(uri, **true**, observer);

反注册：

// Activity关闭时，取消注册内容观察者 getContentResolver().unregisterContentObserver(observer);

6.消息列表适配器的实现 ，由于数据源来于查询数据库后得到的Cursor结果集，所以适配器继承了CursorAdapter

全部实现代码：

**public** **class** ChatMessageAdapter **extends** CursorAdapter {

**public** ChatMessageAdapter(Context context, Cursor cursor, **boolean** autoRequery) {

**super**(context, cursor, autoRequery);

**this**.context = context ;

Activity activity = (Activity)context ;

imApp = (ImApp) ((Activity)context).getApplication();

}

**private** Context context ;

**private** ImApp imApp ;

// 设置ListView显示的Item的种类

@Override

**public** **int** getViewTypeCount() {

**return** 2 ;

}

// 获取消息的类型，0 从当前用户发送出去的消息，1 代表本机当前用户接收到的用户消息

@Override

**public** **int** getItemViewType(**int** position) {

Cursor cursor = getCursor();

cursor.moveToPosition(position);

String from\_id = cursor.getString(cursor.getColumnIndex("FROM\_ID"));

// 判断发来的消息记录的发送者和当前用户是否是同一个人，如果是的话，则将消息显示到界面的右边，不是的话则显示到界面的左边

**if**(from\_id.equals(imApp.getUsername()+"@michael-chun")){

**return** 0 ;

}**else** {

**return** 1 ;

}

}

@Override

**public** **int** getCount() {

**return** getCursor().getCount() ;

}

@SuppressLint("ViewHolder")

@Override

**public** View getView(**int** position, View convertView, ViewGroup parent) {

Cursor cursor = getCursor();

cursor.moveToPosition(position);

**int** type = getItemViewType(position) ;

// 处理消息

**if**(0 == type){

convertView = setSendView(position, convertView);

}**else**{

// 接收的消息

convertView = setReceiveView(position, convertView);

}

**return** convertView ;

}

**private** **class** ViewHolder {

**private** ImageView head ;

**private** TextView time ;

**private** TextView content ;

}

/\*\*

\* 创建发送的消息对应界面上的view

\* **@param** chatMessage 消息对象

\* **@param** convertView

\* **@return**

\*/

**private** View setSendView(**int** position,View convertView){

ViewHolder holder = **null** ;

// 发送的消息

**if**(convertView == **null**){

convertView = View.*inflate*(context, R.layout.***item\_chat\_send***, **null**);

holder = **new** ViewHolder() ;

holder.content = (TextView) convertView.findViewById(R.id.***content***);

holder.head = (ImageView) convertView.findViewById(R.id.***head***);

holder.time = (TextView) convertView.findViewById(R.id.***time***);

convertView.setTag(holder);

}**else**{

holder = (ViewHolder) convertView.getTag();

}

Cursor cursor = getCursor();

cursor.moveToPosition(position); //游标移动到指定的行

**int** avator = cursor.getInt(cursor.getColumnIndex("FROM\_AVATOR"));

String content = cursor.getString(cursor.getColumnIndex("CONTENT"));

**long** time = cursor.getLong(cursor.getColumnIndex("TIME"));

holder.content.setText(content);

holder.time.setText(TimeUtils.*millSecond2FomatedTime*(time));

holder.head.setImageResource(avator);

**return** convertView ;

}

/\*\*

\* 创建接收的消息对应界面上的view

\* **@param** chatMessage 消息对象

\* **@param** convertView

\* **@return**

\*/

**private** View setReceiveView(**int** position,View convertView){

ViewHolder holder = **null** ;

// 发送的消息

**if**(convertView == **null**){

convertView = View.*inflate*(context, R.layout.***item\_chat\_receive***, **null**);

holder = **new** ViewHolder() ;

holder.content = (TextView) convertView.findViewById(R.id.***content***);

holder.head = (ImageView) convertView.findViewById(R.id.***head***);

holder.time = (TextView) convertView.findViewById(R.id.***time***);

convertView.setTag(holder);

}**else**{

holder = (ViewHolder) convertView.getTag();

}

Cursor cursor = getCursor();

cursor.moveToPosition(position);

**int** avator = cursor.getInt(cursor.getColumnIndex("FROM\_AVATOR"));

String content = cursor.getString(cursor.getColumnIndex("CONTENT"));

**long** time = cursor.getLong(cursor.getColumnIndex("TIME"));

holder.content.setText(content);

holder.time.setText(TimeUtils.*millSecond2FomatedTime*(time));

holder.head.setImageResource(avator);

**return** convertView ;

}

@Override

**public** View newView(Context context, Cursor cursor, ViewGroup parent) {

**return** **null**;

}

@Override

**public** **void** bindView(View view, Context context, Cursor cursor) {

}

## 处理被聊情况时产生的bug

1.主动聊天（主动向好友发送消息）

2.被动聊天（好友主动向自己发送消息）

由于在主动聊天中，每次和好友聊天前，代码中实现了先创建一个chat聊天工具，然后利用该聊天工具发送消息给好友，设置消息监听器接收信息

但是在被动聊天时，好友发送过来一个消息，但是前提是没有建立一个和好友的聊天工具chat，也没有创建一个该聊天工具接收消息的监听器，所以产生了接收不到被聊时，好友发送过来的消息

解决办法：1.通过ChatManager设置一个监听Chat聊天工具被创建时的监听器

2.在监听器中根据createdLocally参数判断是主动聊天还是被动聊天

3.如果是被动聊天，获取被聊好友的用户名，在保存了所有聊天工具的集合中查询是否存在该聊天对象，不存在的话则把该监听器方法中的chat参数作为与被聊好友的聊天工具，然后将该聊天工具对象保存到集合中

具体代码实现如下：

// 处理被聊天的情况（别人找我聊天）

// 通过ChatManager设置一个监听chat创建时的监听器

imApp.getConn().getChatManager().addChatListener(**new** ChatManagerListener() {

@Override

**public** **void** chatCreated(Chat chat, **boolean** createdLocally) {

// createdLocally ----true 表示自己主动找别人聊天

// createdLocally ----false 表示别人主动找自己聊天，则chat未被创建，所以产生接收不到别人主动找我聊天的bug

// 由于别人主动找我聊天，chat还没有创建，则直接把方法中的参数chat对象作为聊天工具

**if**(!createdLocally){

// 在保存chat对象的集合中查找是否存在与被聊对象的chat工具

String beiLiaoUser = chat.getParticipant() ; //获取被聊的好友的用户名

String filtedUser = NickUtils.*getCorrectedUserName*(beiLiaoUser); //去掉好友名字后面的终端号，如smack等

Chat beiLiaoChat = chatSet.get(filtedUser);

**if**(beiLiaoChat == **null**){

// 如果为空，则直接将传递过来的参数chat作为聊天工具，并设置监听器，之前没有接收到被聊消息就是因为没有chat工具和没有设置监听器

// 设置被聊天的监听器

chat.addMessageListener(msgListener);

chatSet.put(filtedUser, chat);//将聊天工具保存到集合中

}

}

}

});