ContentObserver 内容观察者，顾名思义是用来观察监测内容发生变化的，属于系统的一个组件

使用场景：保存数据到数据库中时，希望界面能够知道数据库中数据发生了变化，然后去刷新界面

具体使用场景：

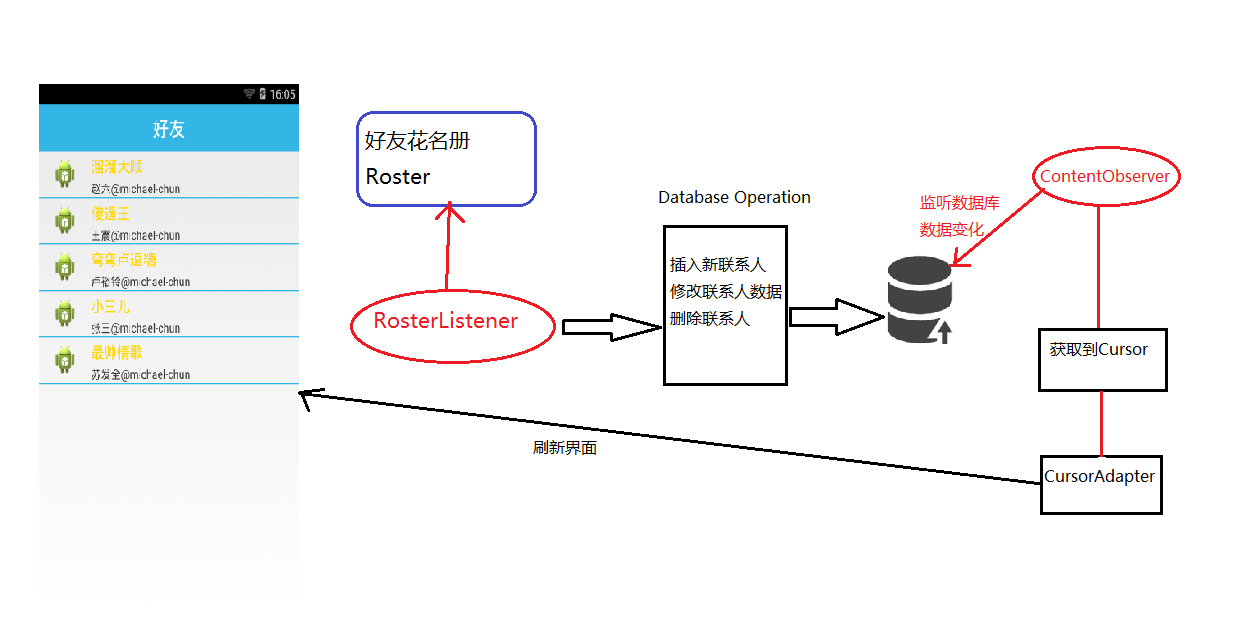
## 1.手机系统应用中的联系人应用

数据库中保存了联系人信息，每次增加、删除联系人或者修改联系人的备注、电话号码信息时，界面需要知道此时数据库中的联系人信息发生了变化，然后重新读取数据中的数据，然后去更新界面，系统采取的就是用内容观察者来实现

## 2.即时通讯软件（自己做的demo I信）的更新好友列表和消息记录模块

1）用户修改了好友的信息，数据库立即更新，界面需要知道数据中信息发生了变化，将新数据刷新到界面上，如上面的相似

2）用户和其他好友的聊天记录信息，通过GreenDao保存在数据库中，每次向好友发送、接收到好友的一条消息时，此消息便立即保存在数据库中，然后应该通知界面去刷新



内容观察者的作用：

1.监听数据的变化

2.将数据发生了变化这一消息，通知其他需要知道该变化的对象

## 和广播的比较

广播接收者也能实现这个功能，为什么不用广播呢？

1.广播接收者数据系统组件，资源消耗可能比较大

2.对于操作数据库这类操作，增删查改比较密集，使用广播的话需要发送很多的广播，比较消耗资源,而且可能存在广播消息接收延时的情况

3.上述第一则案例已经说了，系统的自带应用联系人也是通过内容观察者来实现数据同步的，咱还有啥不放心的呢！

## 使用内容观察者的步骤

说明：每个内容观察者在发送一个数据发生变化的消息，以及注册和反注册时都需要一个Uri，uri好比一种消息的标记，表示我这个内容观察者发送的是这类消息，你也要接受这类消息就需要在注册时使用一样的uri，否则你接收不到我发的消息，uri的命名必须以“content://”开头

1.创建一个内容观察者，在onChange方法内部实现监听到数据库中数据发生变化后应该执行的操作，如刷新界面

/\*\*

\* 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();

//此处实现数据发生变化后应该执行的操作

}

};

2.注册该内容观察者，使得该内容观察者生效，如在Activity创建的时候注册，在销毁的时候反注册

注册（在Activity的onCreate方法中执行）

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

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

反注册（在Activity的onDestroy方法中执行）

// Activity关闭时，取消注册内容观察者

getContentResolver().unregisterContentObserver(observer);

3.在执行数据库操作，导致数据库数据发生了变化的地方通知内容观察者 监听的数据发生了变化

imApp.getMsgRecordDao().insert(record);//此行代码表示数据库操作

// 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**);

在第一步中创建的内容观察者接收到该消息后，会执行其内部实现的onChange方法，onChange方法有两个，一个是在低版本上进行调用的API，另一个则是在高版本上调用的API（三个参数的那个）