- 基础总结篇之六：ContentProvider之读写联系人
- 基础总结篇之八：创建及调用自己的ContentProvider
- ContentProvider的getType()的作用

ContentProvider

Android 中，`ContentProvider` 是一种数据封装器，适合在不同进程中共享数据。下面我们来通过两个部分来介绍如何使用 `ContentProvider`

- 使用系统 `ContentProvider`：读取联系人
- 自定义 `ContentProvider`

使用系统 `ContentProvider` 读取联系人

Android 中联系人的数据存储在 `/data/data/com.android.providers.contacts` 下的 databases 下，在开始前我们需要了解 `android.provider.ContactsContract` 这个类，它定义了各种联系人相关的 URL 和每一种类型信息的属性信息。

下面是完整的例子，我们在 androidTest 下新建一个测试用例类 `ContractsReadTest`，完整代码如下：

```
/**
 * @Desc :
 * @Author : Ramon
 * @create 2021/3/12 23:58
 */
@RunWith(AndroidJUnit4::class)
class ContractsReadTest {
    companion object {
        private const val TAG = "ContractsReadTest"

        // content://com.android.contacts/contacts
        private val CONTRACTS_URL: Uri = ContactsContract.Contacts.CONTENT_URI

        // content://com.android.contacts/data/phones
        private val PHONES_URL: Uri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI

        // content://com.android.contacts/data/emails
        private val EMAIL_URI: Uri = ContactsContract.CommonDataKinds.Email.CONTENT_URI

        private val _ID = ContactsContract.Contacts._ID
        private val DISPLAY_NAME = ContactsContract.Contacts.DISPLAY_NAME
        private val HAS_PHONE_NUMBER = ContactsContract.Contacts.HAS_PHONE_NUMBER
        private val CONTACT_ID = ContactsContract.Data.CONTACT_ID

        private val PHONE_NUMBER = ContactsContract.CommonDataKinds.Phone.NUMBER
        private val PHONE_TYPE = ContactsContract.CommonDataKinds.Phone.TYPE
        private val EMAIL_DATA = ContactsContract.CommonDataKinds.Email.DATA
        private val EMAIL_TYPE = ContactsContract.CommonDataKinds.Email.TYPE
    }
```

```
    @Test
    fun testReadContracts() {
        val appContext = InstrumentationRegistry.getInstrumentation().targetContext
        val contentResolver = appContext.contentResolver
        val c = contentResolver.query(CONTRACTS_URL, null, null, null, null)
        c?.let { cursor ->
            while (c.moveToNext()) {
                val _id = cursor.getInt(cursor.getColumnIndex(_ID))
                val displayName = cursor.getString(cursor.getColumnIndex(DISPLAY_NAME))

                Log.i(TAG, "display name: $displayName")

                // 电话和 email，一个人可能对应多个
                val phones = arrayListOf<String>()
                val emails = arrayListOf<String>()

                // where clause
                val seletion = "$CONTACT_ID=$_id"

                // 获取手机号
                val hasPhoneNumber =
cursor.getInt(cursor.getColumnIndex(HAS_PHONE_NUMBER))
                if (hasPhoneNumber > 0) {
                    val phoneCursor = contentResolver.query(PHONES_URL, null, seletion,
null, null)
                    phoneCursor?.let { pCursor ->
                        while (pCursor.moveToNext()) {
                            val phoneNumber =
pCursor.getString(pCursor.getColumnIndex(PHONE_NUMBER))
                            val phoneType =
pCursor.getInt(pCursor.getColumnIndex(PHONE_TYPE))
                            // 将联系人添加到列表
                            phones.add("${getPhoneTypeNameById(phoneType)}:
$phoneNumber")
                        }
                        pCursor.close()
                    }

                    Log.i(TAG, "phones = $phones")
                }

                // 获取邮箱
                val emCursor = contentResolver.query(EMAIL_URI, null, seletion, null,
null)
                emCursor?.let {emailCursor ->
                    while (emailCursor.moveToNext()) {
                        val emailData =
emailCursor.getString(emailCursor.getColumnIndex(EMAIL_DATA))
                        val emailType =
emailCursor.getInt(emailCursor.getColumnIndex(EMAIL_TYPE))
                        emails.add("${getEmailTypeNameById(emailType)}: $emailData")
                    }
                    emailCursor.close()
                    Log.i(TAG, "emails = $emails")
                }
            }
```

```
            cursor.close()
        }
    }

    private fun getPhoneTypeNameById(typeId: Int): String {
        return when (typeId) {
            ContactsContract.CommonDataKinds.Phone.TYPE_HOME -> "home"
            ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE -> "mobile"
            ContactsContract.CommonDataKinds.Phone.TYPE_WORK -> "work"
            else -> "none"
        }
    }

    private fun getEmailTypeNameById(typeId: Int): String {
        return when (typeId) {
            ContactsContract.CommonDataKinds.Email.TYPE_HOME -> "home"
            ContactsContract.CommonDataKinds.Email.TYPE_WORK -> "work"
            ContactsContract.CommonDataKinds.Email.TYPE_OTHER -> "other"
            else -> "none"
        }
    }
}
```

接下来需要在清单文件中声明读取联系人的权限

```
<!-- 读取联系人 -->
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
```

运行测试用例，在 Log 中可以看到联系人被读出来了。

如果我们是在一个 `Activity` 里读取联系人，可以使用 `ContentResolver` 直接读取，还可以使用 `Activity` 的 `managedQuery` 来读取，来看下这个方法的实现

```
public final Cursor managedQuery(Uri uri,String[] projection,String selection,String[]
selectionArgs,String sortOrder){
        Cursor c = getContentResolver().query(uri, projection, selection, selectionArgs,
sortOrder);
        if (c != null) {
            startManagingCursor(c);
        }
        return c;
}
```

它还是使用了 `ContentResolver` 进行查询操作，但是多了一步 `startManagingCursor` 的操作，它会根据 `Activity` 的生命周期对 `Cursor` 对象进行管理，避免了一些因 `Cursor` 是否释放引起的问题。

向系统 `ContentProvider` 添加联系人

在 AndroidTest 中新建一个测试类 `ContactsWriteTest`，完整代码如下：

```
@RunWith(AndroidJUnit4::class)
class ContactsWriteTest {
    companion object {
        private const val TAG = "ContactsWriteTest"

        // content://com.android.contacts/raw_contacts
        private val RAW_CONTACTS_URI: Uri = ContactsContract.RawContacts.CONTENT_URI
        // content://com.android.contacts/data
        private val DATA_URI = ContactsContract.Data.CONTENT_URI

        private const val ACCOUNT_TYPE = ContactsContract.RawContacts.ACCOUNT_TYPE
        private const val ACCOUNT_NAME = ContactsContract.RawContacts.ACCOUNT_NAME

        private const val RAW_CONTACT_ID = ContactsContract.Data.RAW_CONTACT_ID
        private const val MIMETYPE = ContactsContract.Data.MIMETYPE

        private const val NAME_ITEM_TYPE =
            ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE
        private const val DISPLAY_NAME =
            ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME

        private const val PHONE_ITEM_TYPE =
            ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE
        private const val PHONE_NUMBER = ContactsContract.CommonDataKinds.Phone.NUMBER
        private const val PHONE_TYPE = ContactsContract.CommonDataKinds.Phone.TYPE
        private const val PHONE_TYPE_HOME =
ContactsContract.CommonDataKinds.Phone.TYPE_HOME
        private const val PHONE_TYPE_MOBILE =
ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE

        private const val EMAIL_ITEM_TYPE =
            ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE
        private const val EMAIL_DATA = ContactsContract.CommonDataKinds.Email.DATA
        private const val EMAIL_TYPE = ContactsContract.CommonDataKinds.Email.TYPE
        private const val EMAIL_TYPE_HOME =
ContactsContract.CommonDataKinds.Email.TYPE_HOME
        private const val EMAIL_TYPE_WORK =
ContactsContract.CommonDataKinds.Email.TYPE_WORK
        private const val AUTHORITY = ContactsContract.AUTHORITY

    }

    @Test
    fun testWriteContact() {
        val operations = arrayListOf<ContentProviderOperation>()

        var operation = ContentProviderOperation.newInsert(RAW_CONTACTS_URI)
            .withValue(ACCOUNT_TYPE, null)
            .withValue(ACCOUNT_NAME, null)
            .build()

        operations.add(operation)

        // 添加联系人名称操作
        operation = ContentProviderOperation.newInsert(DATA_URI)
            .withValueBackReference(RAW_CONTACT_ID, 0)
            .withValue(MIMETYPE, NAME_ITEM_TYPE)
```

```
            .withValue(DISPLAY_NAME, "Ramon Lee")
            .build()

        operations.add(operation)

        // 添加家庭座机号码
        operation = ContentProviderOperation.newInsert(DATA_URI)
            .withValueBackReference(RAW_CONTACT_ID, 0)
            .withValue(MIMETYPE, PHONE_ITEM_TYPE)
            .withValue(PHONE_TYPE, PHONE_TYPE_HOME)
            .withValue(PHONE_NUMBER, "3360075")
            .build()

        operations.add(operation)

        // 添加移动手机号码
        operation = ContentProviderOperation.newInsert(DATA_URI)
            .withValueBackReference(RAW_CONTACT_ID, 0)
            .withValue(MIMETYPE, PHONE_ITEM_TYPE)
            .withValue(PHONE_TYPE, PHONE_TYPE_MOBILE)
            .withValue(PHONE_NUMBER, "15900962200")
            .build()
        operations.add(operation)

        // 添加家庭邮箱
        operation = ContentProviderOperation.newInsert(DATA_URI)
            .withValueBackReference(RAW_CONTACT_ID, 0)
            .withValue(MIMETYPE, EMAIL_ITEM_TYPE)
            .withValue(EMAIL_TYPE, EMAIL_TYPE_HOME)
            .withValue(EMAIL_DATA, "xxxx.gmail.com")
            .build()

        operations.add(operation)

        // 添加工作邮箱
        operation = ContentProviderOperation.newInsert(DATA_URI)
            .withValueBackReference(RAW_CONTACT_ID, 0)
            .withValue(MIMETYPE, EMAIL_ITEM_TYPE)
            .withValue(EMAIL_TYPE, EMAIL_TYPE_WORK)
            .withValue(EMAIL_DATA, "xxxx.ten.com")
            .build()

        operations.add(operation)

        val resolver =
InstrumentationRegistry.getInstrumentation().targetContext.contentResolver
        // 批量执行，返回结果
        val results = resolver.applyBatch(AUTHORITY, operations)
        for (i in results.indices) {
            Log.i(TAG, "result $i = ${results[i]}")
        }
    }
}
```

上面我们把插入联系人的操作分为了几个 `ContentProviderOperation` 来操作，`withValueBackReference(RAW_CONTACT_ID, 0)` 表示引用了第一项操作的 id 值。

遇到一个报错，因为把 Test 方法写到了 Companion object 里面去了...

```
Test class should have exactly one public zero-argument constructor
    at
org.junit.runners.BlockJUnit4ClassRunner.validateZeroArgConstructor(BlockJUnit4ClassRunne
r.java:171)
    at
org.junit.runners.BlockJUnit4ClassRunner.validateConstructor(BlockJUnit4ClassRunner.java:
148)
    at
org.junit.runners.BlockJUnit4ClassRunner.collectInitializationErrors(BlockJUnit4ClassRunn
er.java:127)
    at org.junit.runners.ParentRunner.validate(ParentRunner.java:416)
    at org.junit.runners.ParentRunner.<init>(ParentRunner.java:84)
    at org.junit.runners.BlockJUnit4ClassRunner.<init>(BlockJUnit4ClassRunner.java:65)
```

自定义 ContentProvider

自定义前需要了解的两个知识点

authority：称为授权，这是一个唯一标识的字符串，有了这个 `ContentProvider` 才能提供一组 URL，才能向外界共享服务。

```
<provider android:name=".SomeProvider"
            android:authorities="com.your-company.SomeProvider"/>
```

MIME: 多用途 Internet 邮件扩展(`Multipurpose Internet Mail Extensions`), ContentProvider 负责返回给定 URL 的 MIME 类型。 MIME 类型包含两部分，类型和子类型,如 `text/html，text/css，text/xml`

Android 也遵循类似的定义来定义 MIME 类型。

- 对于单条记录，MIME 类似 `vnd.android.cursor.item/vnd.your-company.content-type`
- 对于多条记录，MIME 类似 `vnd.android.cursor.dir/vnd.your-company.comtent-type`

其中 vnd 表示这些类型和子类型具有非标准的，供应商特定的形式， `content-type` 可以根据 ContentProvider 的功能来定，比如日记可以为 note，日程安排可以为 schedule

例：创建一个记录 Person 的 ContentProvider

访问者可以根据下面的路径找到 ContentProvider

```
content://ramon.lee.PersonProvider/persons/3
```

分解下上面这个 URL

- **content**: schema

- **ramon.lee.PersonProvider**: authority
- **persons/3**: path
- **3**: ID

操作者可以根据 `[BASE_URL]/persons` 来操作集合，也可以通过 `[BASE_URL]/persons/#` 的形式操作单个 person。

1. 创建 PersonProvider 类，实现 query insert delete update getType 方法

```
class PersonProvider : ContentProvider() {
    companion object {
        private const val TAG = "PersonProvider"
        private const val AUTHORITY = "ramon.lee.PersonProvider"
        private const val PERSON_ALL = 0
        private const val PERSON_ONE = 1
        private const val CONTENT_TYPE = "vnd.android.cursor.dir/vnd.scott.person"
        private const val CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.scott.person"
        private const val PERSON_TABLE = "person"
    }

    private val matcher: UriMatcher = UriMatcher(UriMatcher.NO_MATCH)
    private var helper: DBHelper? = null
    private var db: SQLiteDatabase? = null

    init {
        matcher.addURI(AUTHORITY, "persons", PERSON_ALL)    //匹配记录集合
        matcher.addURI(AUTHORITY, "persons/#", PERSON_ONE);    //匹配单条记录
    }

    //数据改变后立即重新查询
    private val NOTIFY_URI =
        Uri.parse("content://$AUTHORITY/persons")

    override fun onCreate(): Boolean {
        helper = DBHelper(context!!)
        return true
    }

    override fun getType(uri: Uri): String? {
        return when (matcher.match(uri)) {
            PERSON_ALL -> CONTENT_TYPE
            PERSON_ONE -> CONTENT_ITEM_TYPE
            else -> throw IllegalArgumentException("Unknown URI: $uri")
        }
    }

    override fun query(
        uri: Uri, projection: Array<String>?, selection: String?,
        selectionArgs: Array<String>?, sortOrder: String?
    ): Cursor? {
        db = helper?.readableDatabase
        var selection = selection
        var selectionArgs: Array<String>? = selectionArgs
        when (matcher.match(uri)) {
            PERSON_ALL -> Log.i(TAG, "do nothing")
```

```
            PERSON_ONE -> {
                val _id = ContentUris.parseId(uri)
                selection = "_id = ?"
                selectionArgs = arrayOf(_id.toString())
            }
            else -> throw IllegalArgumentException("Unknown URI: $uri")
        }
        return db?.query(PERSON_TABLE, projection, selection, selectionArgs, null, null,
sortOrder)
    }

    override fun insert(uri: Uri, values: ContentValues?): Uri? {
        val match = matcher.match(uri)
        if (match != PERSON_ALL) {
            throw IllegalArgumentException("Wrong URI: $uri")
        }
        db = helper?.writableDatabase
        var value = values
        if (value == null) {
            value = ContentValues()
            value.put("name", "no name")
            value.put("age", "1")
            value.put("info", "no info")
        }
        val rowId = db?.insert(PERSON_TABLE, null, value)
        rowId?.let {
            if (rowId > 0) {
                notifyDataChanged()
                return ContentUris.withAppendedId(uri, rowId)
            }
        }
        return null
    }

    override fun delete(uri: Uri, selection: String?, selectionArgs: Array<String>?): Int
{
        db = helper?.writableDatabase
        var selection = selection
        var selectionArgs = selectionArgs
        val match = matcher.match(uri)
        when(match) {
            PERSON_ALL -> Log.i(TAG, "do nothing")
            PERSON_ONE -> {
                val _id = ContentUris.parseId(uri)
                selection = "_id = ?"
                selectionArgs = arrayOf(_id.toString())
            }
        }
        val count = db?.delete(PERSON_TABLE, selection, selectionArgs)
        count?.let {
            if (count > 0) {
                notifyDataChanged()
            }
            return count
        } ?: run {
            return 0
        }
    }
```

```
    override fun update(
        uri: Uri, values: ContentValues?, selection: String?,
        selectionArgs: Array<String>?
    ): Int {
        db = helper?.writableDatabase
        var selection = selection
        var selectionArgs = selectionArgs
        val match = matcher.match(uri)
        when(match) {
            PERSON_ALL -> Log.i(TAG, "do nothing")
            PERSON_ONE -> {
                val _id = ContentUris.parseId(uri)
                selection = "_id = ?"
                selectionArgs = arrayOf(_id.toString())
            }
            else -> throw IllegalArgumentException("Unknown URI: $uri ")
        }
        val count = db?.update(PERSON_TABLE, values, selection, selectionArgs)
        count?.let {
            if (count > 0) {
                notifyDataChanged()
            }
            return count
        } ?: run {
            return 0
        }
    }

    //通知指定URI数据已改变
    private fun notifyDataChanged() {
        context?.contentResolver?.notifyChange(NOTIFY_URI, null)
    }
}
```

在上面的类中，我们定义了授权地址 `ramon.lee.PersonProvider`，基于这个授权我们使用了 `URL_MATCHER` 对其路径进行了匹配， `[BASE_URI]/persons` 和 `[BASE_URI]/persons/#` 分别表示操作集合和操作单个数据，在增删改查方法中我们根据匹配的结果做不同的处理。

getType 有什么用呢?

用处1：

在 `getType` 方法中，会根据传入的 URL 返回不同的 MIME 字符串，字符串需要符合以下规定

- 如果是单条记录应该返回以 `vnd.android.cursor.item/` 为首的字符串
- 如果是多条记录，应该返回 `vnd.android.cursor.dir/` 为首的字符串

具体使用，在 Activity 中配置

```
<activity android:name=".activity.ProviderTestActivity">
    <intent-filter>
        <action android:name="ramon.lee.fourcomponent.activity.ProviderTestActivity "/>
        <category android:name="android.intent.category.DEFAULT"/>
```

```
        <data android:mimeType="vnd.android.cursor.dir/vnd.scott.person"/>
    </intent-filter>
    <intent-filter>
        <action android:name="ramon.lee.fourcomponent.activity.ProviderTestActivity "/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="vnd.android.cursor.item/vnd.scott.person"/>
    </intent-filter>
</activity>
```

然后隐式启动这个 Activity

```
val intent = Intent("ramon.lee.fourcomponent.activity.ProviderTestActivity")
val uri = Uri.parse("content://ramon.lee.PersonProvider/persons/1")
intent.data = uri
startActivity(intent)
```

再回顾下我们定义的 `UriMatcher` 和 `getType` 方法

```
private const val CONTENT_TYPE = "vnd.android.cursor.dir/vnd.scott.person"
private const val CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.scott.person"

init {
    matcher.addURI(AUTHORITY, "persons", PERSON_ALL)    //匹配记录集合
    matcher.addURI(AUTHORITY, "persons/#", PERSON_ONE);    //匹配单条记录
}

override fun getType(uri: Uri): String? {
    return when (matcher.match(uri)) {
        PERSON_ALL -> CONTENT_TYPE
        PERSON_ONE -> CONTENT_ITEM_TYPE
        else -> throw IllegalArgumentException("Unknown URI: $uri")
    }
}
```

隐式启动会根据去调用 `getType` 获取 `MIME` 类型，然后判断和 `Activity` 是否匹配，只有匹配才能正确打开 `Activity`

总结： 隐式调用 `activity`，传入 `Uri` 类型 `data`，为了判断 `Activity` 是否能处理这个 `Intent` 请求。

用法2：

还有比如我们在 `query` 方法中有可能是查询全部集合，有可能是查询单条记录，那么我们返回的 Cursor 或是集合类型，或是单条记录，这个跟 `getType` 返回的 `MIME` 类型是一致的。

另外，我们还使用了 `notifyChange` 来通知数据改变。

2. Person 和 DBHelper 定义如下

```
data class Person (
        var _id: Int = 0,
        val name: String,
        val age: Int,
        val info: String)
```

```
class DBHelper(context: Context) :
    SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
    companion object {
        private const val DATABASE_NAME = "provider.db"
        private const val DATABASE_VERSION = 1
    }

    override fun onCreate(db: SQLiteDatabase?) {
        val sql = "CREATE TABLE IF NOT EXISTS person" +
            "(_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "name VARCHAR, " +
                "age INTEGER, " +
                "info TEXT)"
        db?.execSQL(sql)
    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {
        db?.execSQL("DROP TABLE IF EXISTS person")
        onCreate(db)
    }
}
```

3. 在 Manifest 中声明这个 Provider

```
<provider
    android:name=".provider.PersonProvider"
    android:authorities="ramon.lee.PersonProvider"
    android:enabled="true"
    android:exported="true" />
```

4. 创建测试 Activity，进行增删改查

```
class ProviderTestActivity : AppCompatActivity() {
    companion object {
        private const val TAG = "ProviderTestActivity"
        private const val AUTHORITY = "ramon.lee.PersonProvider"
        private val PERSON_ALL_URI: Uri = Uri.parse("content://$AUTHORITY/persons")
    }

    private var binding: ActivityProviderTestBinding? = null
    private var resolver: ContentResolver? = null
    private var personAdapter: PersonAdapter? = null
    private var obserable: PersonObserver? = null
```

```
    private var persons = arrayListOf<Person>()

    private val handler = object: Handler() {
        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            query()
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = DataBindingUtil.setContentView(this, R.layout.activity_provider_test)
        resolver = contentResolver
        personAdapter = PersonAdapter(persons)
        binding?.let {
            it.recycler.apply {
                layoutManager = LinearLayoutManager(this@ProviderTestActivity)
                adapter = personAdapter
            }
        }
        binding?.onClick = ClickAdapter()
        obserable = PersonObserver(handler)
        resolver?.registerContentObserver(PERSON_ALL_URI, true, obserable!!)
    }

    /**
     * 初始化一些数据
     */
    private fun init() {
        val persons = ArrayList<Person>()
        val person1 = Person(name = "Rick",age =  22, info = "gender")
        val person2 = Person(name = "Leo",age =  21, info = "actor")
        val person3 = Person(name = "Luke",age =  20, info = "student")

        persons.add(person1)
        persons.add(person2)
        persons.add(person3)

        for (i in persons.indices) {
            val value = ContentValues()
            value.put("name", persons[i].name)
            value.put("age", persons[i].age)
            value.put("info", persons[i].info)
            resolver?.insert(PERSON_ALL_URI, value)
        }
    }

    /**
     * 查询所有记录
     */
    private fun query() {
        val c: Cursor? = resolver?.query(PERSON_ALL_URI, null, null, null, null)
        val results = arrayListOf<Person>()
        c?.let {
            while (c.moveToNext()) {
                val _id = c.getInt(c.getColumnIndex("_id"))
                val name = c.getString(c.getColumnIndex("name"))
                val age = c.getInt(c.getColumnIndex("age"))
```

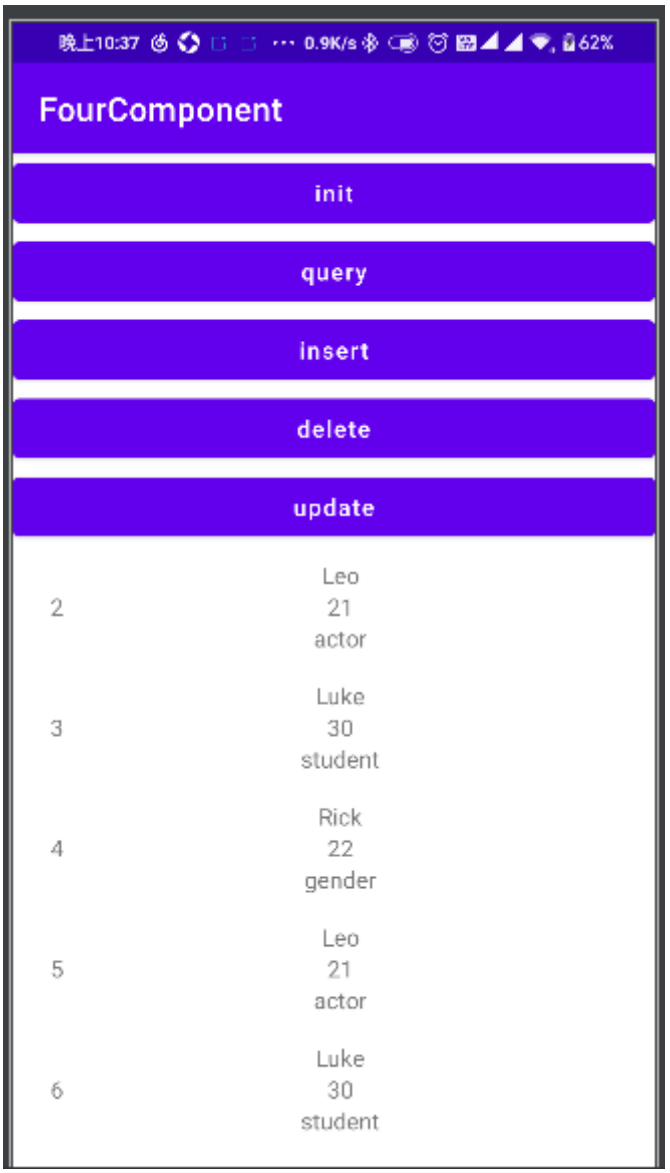
```
                val info = c.getString(c.getColumnIndex("info"))
                results.add(Person(_id, name, age, info))
            }
            c.close()
            persons.clear()
            persons.addAll(results)
            personAdapter?.notifyDataSetChanged()
        } ?: run {
            Log.i(TAG, "cursor is null")
        }
    }

    /**
     * 插入一条记录
     */
    private fun insert() {
        val person = Person(name = "Anna", age = 18, info = "princess")
        val value = ContentValues()
        value.put("name", person.name)
        value.put("age", person.age)
        value.put("info", person.info)
        resolver?.insert(PERSON_ALL_URI, value)
    }

    /**
     * 更新一条记录
     */
    private fun update() {
        // 将指定 name 的 age 更新为 30
        val value = ContentValues()
        value.put("age", 30)
        resolver?.update(PERSON_ALL_URI, value, "name = ?", arrayOf("Luke"))
    }

    /**
     * 删除一条记录
     */
    private fun delete() {
        val delUri = ContentUris.withAppendedId(PERSON_ALL_URI, 1)
        resolver?.delete(delUri, null, null)
    }

    override fun onDestroy() {
        resolver?.unregisterContentObserver(obserable!!)
        super.onDestroy()
    }

    inner class ClickAdapter {
        fun click(view: View) {
            when (view.id) {
                R.id.btn_init -> init()
                R.id.btn_query -> query()
                R.id.btn_insert -> insert()
                R.id.btn_update -> update()
                R.id.btn_delete -> delete()
            }
        }
```

```
    }

    class PersonAdapter(private val items: List<Person>):
RecyclerView.Adapter<PersonAdapter.ViewHolder>() {
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
            return
ViewHolder(DataBindingUtil.inflate(LayoutInflater.from(parent.context),
R.layout.item_person_info, parent, false))
        }

        override fun getItemCount(): Int {
            return items.size
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            holder.bind(items[position])
        }

        inner class ViewHolder(val binding: ItemPersonInfoBinding):
RecyclerView.ViewHolder(binding.root) {
            fun bind(person: Person) {
                binding.person = person
                binding.executePendingBindings()
            }
        }
    }
}
```

5. 其中用到了 ContentObserver 来监听 Provider 改变

```
class PersonObserver(val handler: Handler) : ContentObserver(handler) {
    private val TAG = "PersonObserver"
    override fun onChange(selfChange: Boolean) {
        super.onChange(selfChange)
        Log.i(TAG, "data changed, try to query")
        handler.sendMessage(Message())
    }
}
```

通过上面的方法，当 `Provider` 有更新时，我们可以立即刷新界面，最终效果如下。


晚上10:37
0.9K/s
62%
FourComponent
init
query
insert
delete
update
2
Leo
21
actor
3
Luke
30
student
4
Rick
22
gender
5
Leo
21
actor
6
Luke
30
student