
第六章 Android 进程间通信

这篇文章介绍 Android 框架层提供的一些通信机制 Broadcast、Intent、Content Provider，主要用于应用程序开发时提供跨进程或应用程序内部的通信，优点是接口简单，使用方便，但使用时可能有一些局限性，比如性能和返回数据。这些机制底层可能依赖 Binder、ASH 机制，对于库和框架层的开发人员来说也会更多的用到底层的机制，这些内容会放在另一篇《Android 内核驱动》中介绍。

6.1 Broadcast Receiver

什么是 Broadcast

在 android 中，通过广播（broadcast）可以通知其他广播接受者某个事件发生了。比如电源不足，信号不好等。首先，我们看一个简单的 demo，该 demo 实现了一个自定义 broadcast。

发送端这个 activity 中创建了一个按钮，当按钮被按下的时候通过 `sendBroadcast()` 发送一个 broadcast。

```
public class BroadcastTest extends Activity {
    public static final String NEW_LIFEFROM_DETECTED =
        "com.android.broadcasttest.NEW_LIFEFROM";

    public void onCreate(Bundle savedInstanceState) {
        .....
        Button btn0 = (Button) findViewById(R.id.btn0);
        btn0.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                Intent it = new Intent(NEW_LIFEFROM_DETECTED);
                sendBroadcast(it);
            }
        });
    }
    .....
}
```

接收端在 `onReceive()` 中实现了当接收到 broadcast 所做的动作。

```
Public class MyBroadcastReceiver extends BroadcastReceiver {
    .....
    Public void onReceive(Context context, Intene intent){
        // TODO
    }
    .....
}
```

在 receiver 的 action 中定义了该 receiver 能够接受的广播，Manifest.xml 定义部分：

```
<receiver android:name=".MyBroadcastReceiver">
    <intent-filter>
        <action android:name="com.android.broadcasttest.NEW_LIFEFROM" />
    </intent-filter>
</receiver>
```

Android 中是如何实现 Broadcast 机制

Broadcast 机制是基于一种注册方式的，Broadcast Receiver 将其特征描述并注册在系统中。根据注册时机，可以分为两类，网上有人称之为冷注册和热注册。

- 冷注册，就是 Broadcast Receiver 的相关信息写在配置文件中，系统会负责在相关事件发生的时候及时通知到该 Broadcast Receiver。这种模式适合于这样的场景：某事件发生 -> 通知 Broadcast -> 启动相关处理应用。比如，监听来电、邮件、短信之类的，都隶属于这种模式。
- 热注册，顾名思义，注册这样的事情都是由应用自己来处理的，通常是在 onResume 事件中通过 registerReceiver 进行注册，在 onPause 等事件中通过 unregisterReceiver 反注册，通过这种方式使其能够在运行期间保持对相关事件的关注。比如，一款优秀的词典软件，可能会有在运行期间关注网络状况变化的需求，使其可以在有廉价网络的时候优先使用网络查询词汇，在其他情况下，首先通过本地词库来查词。而这样的监听，只需要在其工作状态下保持就好，不运行的时候，管你是天大的网路变化，与我何干。其模式可以归结为：启动应用 -> 监听事件 -> 发生时进行处理。

前面的 Demo 中用的就是冷注册。热注册的 code 如下：

OnCreate 时，声明一个 BroadcastReceiver。

```
BroadcastReceiver mReceiver=new BroadcastReceiver(){
    Public void onReceive(Context context,Intent intent){
        //收到 Broadcast 会跑到这里
    }
}
```

OnResume 时，通过 registerReceiver 注册。

```
IntentFilter mfilter=new IntentFilter();
mfilter.addAction(Intent.ACTION_SCREEN_OFF);
registerReceiver(mReceiver,mfilter);
```

OnPause 时，通过 unregisterReceiver 反注册。

```
unregisterReceiver(mReceiver);
```

registerReceiver， unregisterReceiver 的内部实现，可以参考 [frameworks/base/services/java/com/android/server/am/ActivityManagerService.java](http://androidappdocs.appspot.com/reference/android/server/am/ActivityManagerService.java)。

除了接收消息的一方有多种模式，发送者也有很重要的选择权。通常，发送者有两类：

- 系统本身，Android 定义了一组的 Standard Broadcast Actions，称为系统 Broadcast 消息，参考 <http://androidappdocs.appspot.com/reference/android/content/Intent.html>。
- 自定义应用通过的接口 Context.sendBroadcast 或 Context.sendOrderedBroadcast 也可以发送 Broadcast。前者发出的称为 Normal broadcast，所有关注该消息的 Receiver，都有

机会获得并进行处理；后者放出的称作 **Ordered broadcasts**，顾名思义，接受者需要按资排辈，排在后面的能否收到广播，需要看前面的处理方式。

需要注意的是，当 **Broadcast Receiver** 接收到相关的消息，在 **OnReceive** 中不要执行很消耗时间的操作，通常把消耗时间的操作放到一个 **Service** 中，在 **OnReceive** 中启动该 **Service**。

虽然 **Broadcast** 整个逻辑不复杂，却是足够有用和好用，它统一了 **Android** 的事件广播模型，更多 **Broadcast Receiver** 相关内容，可以参考：

<http://androidappdocs.appspot.com/reference/android/content/BroadcastReceiver.html>。

6.2 Intent

什么是 Intent

Intent 是一种运行时绑定（run-time binding）机制，它能在程序运行过程中连接两个不同的组件。通过 **Intent**，你的程序可以向 **Android** 表达某种请求或者意愿，**Android** 会根据意愿的内容选择适当的组件来完成请求。比如，有一个 **Activity** 希望打开网页浏览器查看某一网页的内容，那么这个 **Activity** 只需要发出 **WEB_SEARCH_ACTION** 给 **Android**，**Android** 就会根据 **Intent** 的请求内容，查询各组件注册时声明的 **IntentFilter**，找到网页浏览器的 **Activity** 来浏览网页。

Android 的三个基本组件——**Activity**，**Service** 和 **Broadcast Receiver**——都是通过 **Intent** 机制激活的，不同类型的组件有不同的传递 **Intent** 方式：

- 要激活一个新的 **Activity**，或者让一个现有的 **Activity** 做新的操作，可以通过调用 **Context.startActivity()** 或者 **Activity.startActivityForResult()** 方法。
- 要启动一个新的 **Service**，或者向一个已有的 **Service** 传递新的指令，调用 **Context.startService()** 方法或者调用 **Context.bindService()** 方法将调用此方法的上下文对象与 **Service** 绑定。
- **Context.sendBroadcast()**、**Context.sendOrderBroadcast()**、**Context.sendStickyBroadcast()** 这三个方法可以发送 **Broadcast Intent**。发送之后，所有已注册的并且拥有与之相匹配 **IntentFilter** 的 **BroadcastReceiver** 就会被激活。

Intent 一旦发出，**Android** 都会准确找到相匹配的一个或多个 **Activity**，**Service** 或者 **BroadcastReceiver** 作响应。所以，不同类型的 **Intent** 消息不会出现重叠，即 **Broadcast** 的 **Intent** 消息只会发送给 **BroadcastReceiver**，而决不会发送给 **Activity** 或者 **Service**。由 **startActivity()** 传递的消息也只会发给 **Activity**，由 **startService()** 传递的 **Intent** 只会发送给 **Service**。

Intent 的构成

要在不同的 **activity** 之间传递数据，就要在 **intent** 中包含相应的内容，一般来说数据中最基

本的应该包括:

- **Action:** 用来指明要实施的动作是什么, 比如说 ACTION_VIEW, ACTION_EDIT 等。具体的可以查阅 android SDK-> reference 中的 Android.content.intent 类, 里面的 constants 中定义了所有的 action。

一些常用的 Action:

ACTION_CALL	activity	启动一个电话.
ACTION_EDIT	activity	显示用户编辑的数据.
ACTION_MAIN	activity	作为 Task 中第一个 Activity 启动
ACTION_SYNC	activity	同步手机与数据服务器上的数据.
ACTION_BATTERY_LOW	broadcast receiver	电池电量过低警告.
ACTION_HEADSET_PLUG	broadcast receiver	插拔耳机警告
ACTION_SCREEN_ON	broadcast receiver	屏幕变亮警告.
ACTION_TIMEZONE_CHANGED	broadcast receiver	改变时区警告.

- **Data:** 要事实的具体的数据, 一般由一个 Uri 变量来表示

简单的 Action, Data 的例子:

```
Uri uri = Uri.parse("http://www.google.com");
Intent it = new Intent(Intent.ACTION_VIEW, uri);
startActivity(it);
```

- **Category:** 一个字符串, 包含了关于处理该 intent 的组件的种类的信息。一个 intent 对象可以有任意个 category。intent 类定义了许多 category 常数:

CATEGORY_BROWSABLE	目标 activity 可以使用浏览器来显示-例如图片或电子邮件消息
CATEGORY_GADGET	该 activity 可以被包含在另外一个装载小工具的 activity 中
CATEGORY_HOME	该 activity 显示主屏幕, 也就是用户按下 Home 键看到的界面
CATEGORY_LAUNCHER	该 activity 可以作为一个 Task 的第一个 activity, 并且列在应用程序启动器中
CATEGORY_PREFERENCE	该 activity 是一个选项面板

addCategory()方法为一个 intent 对象增加一个 category,
removeCategory 删除一个 category,
getCategories()获取 intent 所有的 category.

- **Type:** 显式指定 Intent 的数据类型 (MIME) (多用途互联网邮件扩展, Multipurpose Internet Mail Extensions)。比如, 一个组件是可以显示图片数据的而不能播放声音文件。很多情况下, data 类型可在 URI 中找到, 比如 content:开头的 URI, 表明数据由设备上的 content provider 提供。但是通过设置这个属性, 可以强制采用显式指定的类型而不再进行推导。

MIME 类型有 2 种形式:

- 单个记录的格式: `vnd.android.cursor.item/vnd.yourcompanyname.contenttype`, 如:
`content://com.example.transportationprovider/trains/122` (一条列车信息的 uri) 的 MIME 类型是 `vnd.android.cursor.item/vnd.example.rail`
- 多个记录的格式: `vnd.android.cursor.dir/vnd.yourcompanyname.contenttype`, 如:
`content://com.example.transportationprovider/trains` (所有列车信息) 的 MIME 类型是 `vnd.android.cursor.dir/vnd.example.rail`

- **component**: 指定 Intent 的目标组件的类名称。通常 Android 会根据 Intent 中包含的其它属性的信息, 比如 `action`、`data/type`、`category` 进行查找, 最终找到一个与之匹配的目标组件。但是, 如果 `component` 这个属性有指定的话, 将直接使用它指定的组件, 而不再执行上述查找过程。指定了这个属性以后, Intent 的其它所有属性都是可选的。例如:

```
Intent it = new Intent(Activity.Main.this, Activity2.class);
startActivity(it);
```

- **extras**: 附加信息, 例如 `ACTION_TIMEZONE_CHANGED` 的 intent 有一个 "time-zone" 附加信息来指明新的时区, 而 `ACTION_HEADSET_PLUG` 有一个 "state" 附加信息来指示耳机是被插入还是被拔出。intent 对象有一系列 `put...()` 和 `set...()` 方法来设定和获取附加信息。这些方法和 `Bundle` 对象很像。事实上附加信息可以使用 `putExtras()` 和 `getExtras()` 作为 `Bundle` 来读和写。例如:

```
//用 Bundle 传递数据
Intent it = new Intent(Activity.Main.this, Activity2.class);
Bundle bundle=new Bundle();
bundle.putString("name", "This is from MainActivity!");
it.putExtras(bundle);
startActivity(it);
//获得数据
Bundle bundle=getIntent().getExtras();
String name=bundle.getString("name");
```

intent 的解析

在应用中, 我们可以以两种形式来使用 Intent:

- **直接 Intent**: 指定了 `component` 属性的 Intent (调用 `setComponent(ComponentName)` 或者 `setClass(Context, Class)` 来指定)。通过指定具体的组件类, 通知应用启动对应的组件。
- **间接 Intent**: 没有指定 `component` 属性的 Intent。这些 Intent 需要包含足够的信息, 这样系统才能根据这些信息, 在所有可用的组件中, 确定满足此 Intent 的组件。

对于直接 Intent, Android 不需要去做解析, 因为目标组件已经很明确, Android 需要解析的是那些间接 Intent, 通过解析将 Intent 映射给可以处理此 Intent 的 Activity、Service 或 Broadcast Receiver。

Intent 解析机制主要是通过查找已注册在 `AndroidManifest.xml` 中的所有 `<intent-filter>` 及其中定义的 Intent, 通过 `PackageManager` (注: `PackageManager` 能够得到当前设备上所安装的

application package 的信息) 来查找能处理这个 Intent 的 component。在这个解析过程中, Android 是通过 Intent 的 action、type、category 这三个属性来进行判断的, 判断方法如下:

- 如果 Intent 指定了 action, 则目标组件的 IntentFilter 的 action 列表中就必须包含有这个 action, 否则不能匹配;
- 如果 Intent 没有提供 type, 系统将从 data 中得到数据类型。和 action 一样, 目标组件的数据类型列表中必须包含 Intent 的数据类型, 否则不能匹配。
- 如果 Intent 中的数据不是 content:类型的 URI, 而且 Intent 也没有明确指定 type, 将根据 Intent 中数据的 scheme (比如 http:或者 mailto:) 进行匹配。同上, Intent 的 scheme 必须出现在目标组件的 scheme 列表中。
- 如果 Intent 指定了一个或多个 category, 这些类别必须全部出现在组建的类别列表中。比如 Intent 中包含了两个类别: LAUNCHER_CATEGORY 和 ALTERNATIVE_CATEGORY, 解析得到的目标组件必须至少包含这两个类别。

下面举例说明 Intent 如何定义及如何被解析。完整代码请参考 development/samples/notepad, 以及 <http://www.cnblogs.com/phinecos/archive/2009/08/26/1554684.html>

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name">
```

声明 provider

```
    <provider android:name="NotePadProvider"
        android:authorities="com.google.provider.NotePad"
    />
```

第一个 Activity——NoteList, 这个 activity 声明了三个 intent filter

```
<activity android:name="NotesList" android:label="@string/title_notes_list">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.GET_CONTENT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>
```

- 第一个 filter 使这个 Activity 出现在 launcher 并作为程序的主 activity
- 第二个 filter, 当 type 为 vnd.android.cursor.dir/vnd.google.note (便笺目录) 时, 可以用这个 activity 查看(android.app.action.VIEW)、编辑(android.app.action.EDIT)、选择(android.app.action.PICK)。

- 第三个 filter，当 type 为 vnd.android.cursor.item/vnd.google.note（便笺记录）时，可以用这个 activity 读取记录内容（android.app.action.GET_CONTENT）。

第二个 Activity——NoteEditor，声明了两个 intent filter

```
<activity android:name="NoteEditor"
    android:theme="@android:style/Theme.Light"
    android:label="@string/title_note"
    android:screenOrientation="sensor"
    android:configChanges="keyboardHidden|orientation"
>
    <!-- This filter says that we can view or edit the data of a single note -->
    <intent-filter android:label="@string/resolve_edit">
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="com.android.notepad.action.EDIT_NOTE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>

    <!-- This filter says that we can create a new note inside of a directory of
    notes. -->
    <intent-filter>
        <action android:name="android.intent.action.INSERT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
</activity>
```

- 第一个 filter 查看、编辑便笺
- 第二个 filter 新建便笺

第三个 Activity——TitleEditor，这个 activity 用于修改标题，除支持缺省类别，还支持另外两个类别：android.intent.category.ALTERNATIVE 和 android.intent.category.SELECTED_ALTERNATIVE。实现了这两个类别之后，其它 Activity 就可以调用 queryIntentActivityOptions(ComponentName, Intent[], Intent, int) 查询这个 Activity 提供的 action，而不需要了解它的具体实现；或者调用 addIntentOptions(int, int, ComponentName, Intent[], Intent, int, MenuItem[]) 建立动态菜单。

```
<activity android:name="TitleEditor"
    android:label="@string/title_edit_title"
    android:theme="@android:style/Theme.Dialog"
    android:windowSoftInputMode="stateVisible">
    <intent-filter android:label="@string/resolve_title">
        <action android:name="com.android.notepad.action.EDIT_TITLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.ALTERNATIVE" />
        <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>
```

在这个 intent-filter 中有一个明确的名称（通过 android:label="@string/resolve_title" 指定），在用户浏览数据的时候，如果这个 Activity 是数据的一个可选操作，指定明确的名称可以为用户提供更好控制界面。

有了这个功能，下面的 Intent 就会被解析到 TitleEditor 这个 Activity，显示并且允许用户编辑标识为 ID 的便笺的标题。

```
action=com.google.android.notepad.action.EDIT_TITLE
data=content://com.google.provider.NotePad/notes/{ID}
```

更多详细内容请参考 <http://developer.android.com/reference/android/content/Intent.html>

6.3 Content Provider

Android 应用程序可以使用文件或 SQLite 数据库来存储数据。Content Provider 提供了一种多应用间数据共享的方式，比如：联系人信息可以被多个应用程序访问。Content Provider 实现了一组用于提供其他应用程序存取数据的标准方法的类。

应用程序可以在 Content Provider 中执行如下操作：

- 查询数据
- 修改数据
- 添加数据
- 删除数据

标准的 Content Provider

Android 提供了一些已经在系统中实现的标准 Content Provider，比如联系人信息，图片库等等，你可以用这些 Content Provider 来访问设备上存储的联系人信息，图片等等。

1、 查询数据

在 Content Provider 中使用的查询字符串有别于标准的 SQL 查询。很多诸如 select, add, delete, modify 等操作都使用一种特殊的 URI 来进行，这种 URI 由 3 个部分组成：

- “content://”
- 代表数据的路径
- 一个可选的标识数据的 ID

以下是一些示例 URI：

content://media/internal/images 这个 URI 将返回设备上存储的所有图片
content://contacts/people/ 这个 URI 将返回设备上的所有联系人信息
content://contacts/people/45 这个 URI 返回单个结果（ID 为 45 的联系人记录）

Android 提供了一系列的帮助类（在 android.provider 包下），里面包含了很多以类变量形式给出的查询字符串，这种方式更容易让我们理解，参见下例：

```
MediaStore.Images.Media.INTERNAL_CONTENT_URI  
Contacts.People.CONTENT_URI
```

因此，如上面 content://contacts/people/45 这个 URI 就可以写成如下形式：

```
Uri person = ContentUris.withAppendedId(People.CONTENT_URI, 45);
```

然后执行数据查询：

```
Cursor cur = managedQuery(person, null, null, null);
```

这个查询返回一个包含所有数据字段的 Cursor，可以通过迭代 Cursor 来获取所有的数据：

```
public class ContentProviderDemo extends Activity {  
    @Override
```



```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    displayRecords();
}

private void displayRecords() {
    //该数组中包含了所有要返回的字段
    String columns[] = new String[] { People.NAME, People.NUMBER };
    Uri mContacts = People.CONTENT_URI;
    Cursor cur = managedQuery(
        mContacts,
        columns, // 要返回的数据字段
        null, // WHERE 子句
        null, // WHERE 子句的参数
        null // Order-by子句
    );
    if (cur.moveToFirst()) {
        String name = null;
        String phoneNo = null;
        do {
            name = cur.getString(cur.getColumnIndex(People.NAME));
            phoneNo = cur.getString(cur.getColumnIndex(People.NUMBER));
            Toast.makeText(this,
                name + " " + phoneNo,
                Toast.LENGTH_LONG
            ).show();
        } while (cur.moveToNext());
    }
}
}

```

2、修改记录

我们可以使用 `ContentResolver.update()`方法来修改数据:

```
updateRecord(10, "XYZ");
```

更改第 10 条记录的 `name` 字段值为 “XYZ”

```

private void updateRecord(int recNo, String name) {
    Uri uri = ContentUris.withAppendedId(People.CONTENT_URI, recNo);
    ContentValues values = new ContentValues();
    values.put(People.NAME, name);
    getContentResolver().update(uri, values, null, null);
}

```

3、添加记录

我们可以调用 `ContentResolver.insert()`方法来增加记录,该方法接受一个要增加的记录的目标 `URI`, 以及一个包含了新记录值的 `Map` 对象,调用后的返回值是新记录的 `URI`, 包含记录号。

```

private void insertRecords(String name, String phoneNo) {
    ContentValues values = new ContentValues();
    values.put(People.NAME, name);
    Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
    Log.d("ANDROID", uri.toString());
    Uri numberUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);
    values.clear();
    values.put(Contacts.Phones.TYPE, People.Phones.TYPE_MOBILE);
    values.put(People.NUMBER, phoneNo);
    getContentResolver().insert(numberUri, values);
}

```

这样我们就可以调用 `insertRecords(name, phoneNo)`的方式来向联系人信息簿中添加联系人姓名和电话号码。

4、删除记录

`getContextResolver.delete()`方法可以用来删除记录。如删除设备上所有的联系人信息:

```
private void deleteRecords() {
    Uri uri = People.CONTENT_URI;
    getContextMenu().delete(uri, null, null);
}
```

指定 `WHERE` 条件语句来删除特定的记录, 删除 `name` 为 'XYZ' 的记录:

```
getContextResolver().delete(uri, "NAME=" + "'XYZ'", null);
```

创建 Content Provider:

要创建我们自己的 `Content Provider` 的话, 我们需要遵循以下几步:

1、 创建一个继承自 `ContentProvider` 的类

2、 定义一个 `public static final Uri` 类型的类变量, 你必须为其指定一个唯一的字符串值, 最好是类的全名称, 如

```
public static final Uri CONTENT_URI =
    Uri.parse("content://com.mycom.MyContentProvider");
```

3、 创建数据存储系统。大多数 `Content Provider` 使用 `Android` 文件系统或 `SQLite` 数据库来保持数据, 但是你也可以以任何你想要的方式来存储。

4、 定义返回给客户端的数据列名。如果你正在使用 `Android` 数据库, 则数据列的使用方式就和你以往所熟悉的其他数据库一样。但是, 你必须为其定义一个 `_id` 列, 它用来表示每条记录的唯一性。

5、 如果你要存储字节型数据, 比如位图文件等, 那保存该数据的数据列其实是一个表示实际保存文件的 `URI` 字符串, 客户端通过它来读取对应的文件数据, 处理这种数据类型的 `Content Provider` 需要实现一个名为 `_data` 的字段, `_data` 字段列出了该文件在 `Android` 文件系统上的精确路径。这个字段不仅是供客户端使用, 而且也可以供 `ContentResolver` 使用。客户端可以调用 `ContentResolver.openOutputStream()` 方法来处理该 `URI` 指向的文件资源, 如果是 `ContentResolver` 本身的话, 由于其持有的权限比客户端要高, 所以它能直接访问该数据文件。

6、 声明 `public static String` 型的变量, 用于指定要从 `Cursor` 处返回的数据列。

7、 查询返回一个 `Cursor` 类型的对象。所有执行写操作的方法如 `insert()`, `update()` 以及 `delete()` 都将被监听。我们可以通过使用 `ContentResolver().notifyChange()` 方法来通知监听器关于数据更新的信息。

8、 在 `AndroidManifest.xml` 中标明 `Content Provider`。

-
- 9、如果你要处理的数据类型是一种比较新的类型，你就必须先定义一个新的 MIME 类型，以供 `ContentProvider.getType(uri)` 来返回。

自定义 Provider

定义了 Content Provider 的 `CONTENT_URI`，以及数据列：

```
public class MyUsers {
    public static final String AUTHORITY = "com.wissen.MyContentProvider";
    // BaseColumn 类中已经包含了 _id 字段
    public static final class User implements BaseColumns {
        public static final Uri CONTENT_URI =
            Uri.parse("content://com.wissen.MyContentProvider");

        // 表数据列
        public static final String USER_NAME = "USER_NAME";
    }
}
```

基于上面的类来定义实际的 Content Provider 类：

```
public class MyContentProvider extends ContentProvider {
    private SQLiteDatabase sqlDB;
    private DatabaseHelper dbHelper;
    private static final String DATABASE_NAME = "Users.db";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_NAME = "User";
    private static final String TAG = "MyContentProvider";
    private static final String DATABASE_CREATE =
        "create table " + TABLE_NAME +
        " (_id integer primary key autoincrement, USER_NAME text);";
}
```

Android 为 SQLite 提供了便利的 `SQLiteOpenHelper` 方便自动创建新的数据库或者升级数据库

```
private static class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DATABASE_CREATE); //创建表
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }
}
```

继续 `MyContentProvider` 类

```
@Override
public boolean onCreate() {
    dbHelper = new DatabaseHelper(getContext());
    return (dbHelper == null) ? false : true;
}

@Override
```

```

public Uri insert(Uri uri, ContentValues contentvalues) {
    sqlDB = dbHelper.getWritableDatabase();
    long rowId = sqlDB.insert(TABLE_NAME, "", contentvalues);
    if (rowId > 0) {
        Uri rowUri =
            ContentUris.appendId(MyUsers.User.CONTENT_URI.buildUpon(),
                rowId).build();

        getContext().getContentResolver().notifyChange(rowUri, null);
        return rowUri;
    }
    throw new SQLException("Failed to insert row into " + uri);
}

@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    qb.setTables(TABLE_NAME);
    Cursor c = qb.query(db, projection, selection,
        null, null, null, sortOrder);
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}

@Override
public int update(Uri uri, ContentValues contentvalues,
    String s, String[] as) {
    return 0;
}

@Override
public int delete(Uri uri, String s, String[] as) {
    return 0;
}

@Override
public String getType(Uri uri) {
    return null;
}
}

```

创建 Content Provider 后，需要在 manifest 文件中声明，android 系统才能知道它，当其他应用需要调用该 Content Provider 时才能创建或者调用它。

```

<provider android:name="com.wissen.MyContentProvider"
    android:authorities="com.wissen.MyContentProvider">
</provider>

```

使用自定义的 Content Provider

```

public class MyContentDemo extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        insertRecord("MyUser");
        displayRecords();
    }

    private void insertRecord(String userName) {
        ContentValues values = new ContentValues();
        values.put(MyUsers.User.USER_NAME, userName);
        getContentResolver().insert(MyUsers.User.CONTENT_URI, values);
    }

    private void displayRecords() {

```

```
String columns[] = new String[] {
    MyUsers.User._ID,
    MyUsers.User.USER_NAME
};
Uri myUri = MyUsers.User.CONTENT_URI;
Cursor cur = managedQuery(myUri, columns, null, null, null );
if (cur.moveToFirst()) {
    String id = null;
    String userName = null;

    do {
        id = cur.getString(cur.getColumnIndex(MyUsers.User._ID));
        userName = cur.getString(
            cur.getColumnIndex(
                MyUsers.User.USER_NAME));
        Toast.makeText(
            this,
            id + " " + userName,
            Toast.LENGTH_LONG
        ).show();
    } while (cur.moveToNext());
}
}
```