

# 第 7 章 跨程序共享数据，探究 内容提供器

在上一章中我们学了 Android 数据持久化的技术，包括文件存储、SharedPreferences 存储、以及数据库存储。不知道你有没有发现，使用这些持久化技术所保存的数据都只能在当前应用程序中访问。虽然文件和 SharedPreferences 存储中提供了 `MODE_WORLD_READABLE` 和 `MODE_WORLD_WRITEABLE` 这两种操作模式，用于供给其他的应用程序访问当前应用的数据，但这两种模式在 Android 4.2 版本中都被废弃了。为什么呢？因为 Android 官方已经不再推荐使用这种方式来实现跨程序数据共享的功能，而是应该使用更加安全可靠的内容提供器技术。

可能你会有些疑惑，为什么要将我们程序中的数据共享给其他程序呢？当然，这个要视情况而定的，比如说账号和密码这样的隐私数据显然是不能共享给其他程序的，不过一些可以让其他程序进行二次开发的基础性数据，我们还是可以选择将其共享的。例如系统的电话簿程序，它的数据库中保存了很多的联系人信息，如果这些数据都不允许第三方的程序进行访问的话，恐怕很多应用的功能都要大打折扣了。除了电话簿之外，还有短信、媒体库等程序都实现了跨程序数据共享的功能，而使用的技术当然就是内容提供器了，下面我们就来对这一技术进行深入的探讨。

## 7.1 内容提供器简介

内容提供器（Content Provider）主要用于在不同的应用程序之间实现数据共享的功能，它提供了一套完整的机制，允许一个程序访问另一个程序中的数据，同时还能保证被访数据的安全性。目前，使用内容提供器是 Android 实现跨程序共享数据的标准方式。

不同于文件存储和 SharedPreferences 存储中的两种全局可读写操作模式，内容提供器可以选择只对哪一部分数据进行共享，从而保证我们程序中的隐私数据不会有泄漏的风险。

内容提供器的用法一般有两种，一种是使用现有的内容提供器来读取和操作相应程序中的数据，另一种是创建自己的内容提供器给我们程序的数据提供外部访问接口。那么接下来我们就一个一个开始学习吧，首先从使用现有的内容提供器开始。

## 7.2 访问其他程序中的数据

当一个应用程序通过内容提供者对其数据提供了外部访问接口，任何其他的应用程序都可以对这部分数据进行访问。Android 系统中自带的电话簿、短信、媒体库等程序都提供了类似的访问接口，这就使得第三方应用程序可以充分地利用这部分数据来实现更好的功能。下面我们就来看一看，内容提供者到底是如何使用的。

### 7.2.1 ContentResolver 的基本用法

对于每一个应用程序来说，如果想要访问内容提供者中共享的数据，就一定要借助 ContentResolver 类，可以通过 Context 中的 getContentResolver() 方法获取到该类的实例。ContentResolver 中提供了一系列的方法用于对数据进行 CRUD 操作，其中 insert() 方法用于添加数据，update() 方法用于更新数据，delete() 方法用于删除数据，query() 方法用于查询数据。有没有似曾相识的感觉？没错，SQLiteDatabase 中也是使用的这几个方法来进行 CRUD 操作的，只不过它们在方法参数上稍微有一些区别。

不同于 SQLiteDatabase，ContentResolver 中的增删改查方法都是不接收表名参数的，而是使用一个 Uri 参数代替，这个参数被称为内容 URI。内容 URI 给内容提供者中的数据建立了唯一标识符，它主要由两部分组成，权限（authority）和路径（path）。权限是用于对不同的应用程序做区分的，一般为了避免冲突，都会采用程序包名的方式来进行命名。比如某个程序的包名是 com.example.app，那么该程序对应的权限就可以命名为 com.example.app.provider。路径则是用于对同一应用程序中不同的表做区分的，通常都会添加到权限的后面。比如某个程序的数据库里存在两张表，table1 和 table2，这时就可以将路径分别命名为/table1 和/table2，然后把权限和路径进行组合，内容 URI 就变成了 com.example.app.provider/table1 和 com.example.app.provider/table2。不过，目前还很难辨认出这两个字符串就是两个内容 URI，我们还需要在字符串的头部加上协议声明。因此，内容 URI 最标准的格式写法如下：

```
content://com.example.app.provider/table1
content://com.example.app.provider/table2
```

有没有发现，内容 URI 可以非常清楚地表达出我们想要访问哪个程序中哪张表里的数据。也正是因此，ContentResolver 中的增删改查方法才都接收 Uri 对象作为参数，因为使用表名的话系统将无法得知我们期望访问的是哪个应用程序里的表。

在得到了内容 URI 字符串之后，我们还需要将它解析成 Uri 对象才可以作为参数传入。解析的方法也相当简单，代码如下所示：

```
Uri uri = Uri.parse("content://com.example.app.provider/table1")
```

只需要调用 Uri.parse() 方法，就可以将内容 URI 字符串解析成 Uri 对象了。

现在我们就可以使用这个 Uri 对象来查询 table1 表中的数据了，代码如下所示：

```
Cursor cursor = getContentResolver().query(
    uri,
    projection,
    selection,
    selectionArgs,
    sortOrder);
```

这些参数和 SQLiteDatabase 中 query() 方法里的参数很像，但总体来说要简单一些，毕竟这是在访问其他程序中的数据，没必要构建过于复杂的查询语句。下表对使用到的这部分参数进行了详细的解释。

query()方法参数	对应 SQL 部分	描述
uri	from table_name	指定查询某个应用程序下的某一张表
projection	select column1, column2	指定查询的列名
selection	where column = value	指定 where 的约束条件
selectionArgs	-	为 where 中的占位符提供具体的值
orderBy	order by column1, column2	指定查询结果的排序方式

查询完成后返回的仍然是一个 Cursor 对象，这时我们就可以将数据从 Cursor 对象中逐个读取出来了。读取的思路仍然是通过移动游标的位置来遍历 Cursor 的所有行，然后再取出每一行中相应列的数据，代码如下所示：

```
if (cursor != null) {
    while (cursor.moveToNext()) {
        String column1 = cursor.getString(cursor.getColumnIndex("column1"));
        int column2 = cursor.getInt(cursor.getColumnIndex("column2"));
    }
    cursor.close();
}
```

掌握了最难的查询操作，剩下的增加、修改、删除操作就更不在话下了。我们先来看看如何向 table1 表中添加一条数据，代码如下所示：

```
ContentValues values = new ContentValues();
values.put("column1", "text");
values.put("column2", 1);
getContentResolver().insert(uri, values);
```

可以看到，仍然是将待添加的数据组装到 ContentValues 中，然后调用 ContentResolver

的 `insert()` 方法，将 `Uri` 和 `ContentValues` 作为参数传入即可。

现在如果我们想要更新这条新添加的数据，把 `column1` 的值清空，可以借助 `ContentResolver` 的 `update()` 方法实现，代码如下所示：

```
ContentValues values = new ContentValues();
values.put("column1", "");
getContentResolver().update(uri, values, "column1 = ? and column2 = ?", new
String[] { "text", "1" });
```

注意上述代码使用了 `selection` 和 `selectionArgs` 参数来对想要更新的数据进行约束，以防止所有的行都会受影响。

最后，可以调用 `ContentResolver` 的 `delete()` 方法将这条数据删除掉，代码如下所示：

```
getContentResolver().delete(uri, "column2 = ?", new String[] { "1" });
```

到这里为止，我们就把 `ContentResolver` 中的增删改查方法全部学完了。是不是感觉非常简单？因为这些知识早在上一章中学习 `SQLiteDatabase` 的时候你就已经掌握了，所需特别注意的就只有 `uri` 这个参数而已。那么接下来，我们就利用目前所学的知识，看一看如何读取系统电话簿中的联系人信息。

## 7.2.2 读取系统联系人

由于我们之前一直使用的都是模拟器，电话簿里面并没有联系人存在，所以现在需要自己手动添加几个，以便稍后进行读取。打开电话簿程序，界面如图 7.1 所示。

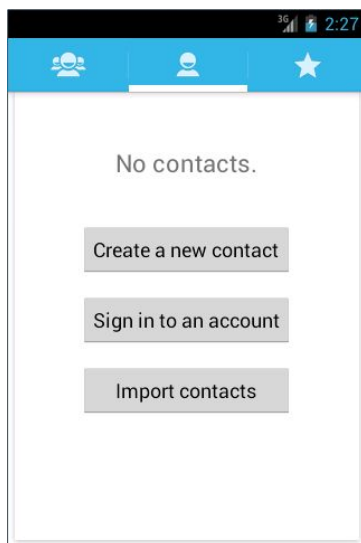


图 7.1

可以看到，目前电话簿里是没有任何联系人的，我们可以通过点击 **Create a new contact** 按钮来对联系人进行创建。这里就先创建两个联系人吧，分别填入他们的姓名和手机号，如图 7.2 所示。

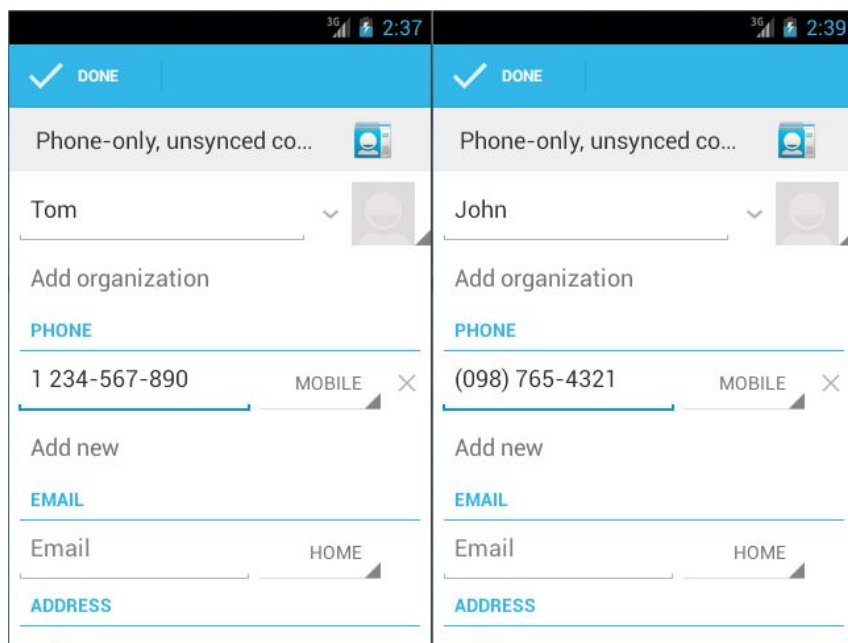


图 7.2

这样准备工作就做好了，现在新建一个 **ContactsTest** 项目，让我们开始动手吧。

首先还是来编写一下布局文件，这里我们希望读取出来的联系人信息能够在 **ListView** 中显示，因此，修改 **activity\_main.xml** 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ListView
        android:id="@+id/contacts_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </ListView>

</LinearLayout>
```

简单起见，LinearLayout 里就只放置了一个 ListView。接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    ListView contactsView;

    ArrayAdapter<String> adapter;

    List<String> contactsList = new ArrayList<String>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        contactsView = (ListView) findViewById(R.id.contacts_view);
        adapter = new ArrayAdapter<String>(this, android.R.layout.
simple_list_item_1, contactsList);
        contactsView.setAdapter(adapter);
        readContacts();
    }

    private void readContacts() {
        Cursor cursor = null;
        try {
            // 查询联系人数据
            cursor = getContentResolver().query(
                ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                null, null, null, null);
            while (cursor.moveToNext()) {
                // 获取联系人姓名
                String displayName = cursor.getString(cursor.getColumnIndex(
                    ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
                // 获取联系人手机号
                String number = cursor.getString(cursor.getColumnIndex(
                    ContactsContract.CommonDataKinds.Phone.NUMBER));
                contactsList.add(displayName + "\n" + number);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        } finally {
            if (cursor != null) {
                cursor.close();
            }
        }
    }
}

```

在 `onCreate()` 方法中，我们首先获取了 `ListView` 控件的实例，并给它设置好了适配器，然后就去调用 `readContacts()` 方法。下面重点看下 `readContacts()` 方法，可以看到，这里使用了 `ContentResolver` 的 `query()` 方法来查询系统的联系人数据。不过传入的 `Uri` 参数怎么有些奇怪啊，为什么没有调用 `Uri.parse()` 方法去解析一个内容 `URI` 字符串呢？这是因为 `ContactsContract.CommonDataKinds.Phone` 类已经帮我们做好了封装，提供了一个 `CONTENT_URI` 常量，而这个常量就是使用 `Uri.parse()` 方法解析出来的结果。接着我们对 `Cursor` 对象进行遍历，将联系人姓名和手机号这些数据逐个取出，联系人姓名这一列对应的常量是 `ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME`，联系人手机号这一列对应的常量是 `ContactsContract.CommonDataKinds.Phone.NUMBER`。两个数据都取出之后，将它们进行拼接，并且中间加上换行符，然后将拼接后的数据添加到 `ListView` 里。最后千万不要忘记将 `Cursor` 对象关闭掉。

这样就结束了吗？还差一点点，读取系统联系人也是需要声明权限的，因此修改 `AndroidManifest.xml` 中的代码，如下所示：

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.contactstest"
    android:versionCode="1"
    android:versionName="1.0" >
    .....

    <uses-permission android:name="android.permission.READ_CONTACTS" />
    .....

</manifest>

```

加入了 `android.permission.READ_CONTACTS` 权限，这样我们的程序就可以访问到系统的联系人数据了。现在才算是大功告成，让我们来运行一下程序吧，效果如图 7.3 所示。

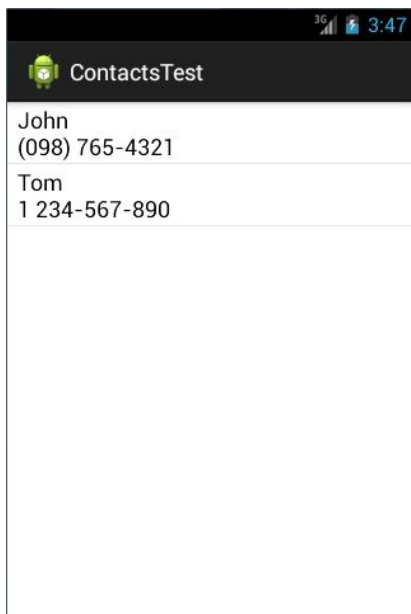


图 7.3

刚刚添加的两个联系人的数据都成功读取出来了！说明跨程序访问数据的功能确实是实现了。

经验值：+5000

目前经验值：41905

级别：资深鸟

赢得宝物：战胜内容提供猪。拾取内容提供猪掉落的宝物，一小瓶具有舒缓神经作用的烧酒、猪皮 Android 战袍一套、大号搓澡巾一条、神肤佳一块、神柔洗发露一瓶、神洁士牙膏和牙刷一套，还有一个看起来质量不错的小杯子（应该是漱口用的）。我很惊讶，作为一头猪，居然会随身携带这些东西，即使作为人，随身携带这些东西也是让人侧目的。不过这并不让人讨厌，因为这显然说明它是一头很讲卫生的猪。一头如此讲卫生的猪显然是应该认真对待的，所以我决定将这些东西还给它。后来我才知道，内容提供猪不仅讲卫生，而且非常懂礼貌，是神界公认的好孩子，它热衷社区义工，常利用周末时间去给上了年纪的爷爷奶奶们念报纸。

## 7.3 创建自己的内容提供器

在上一节当中，我们学习了如何在自己的程序中访问其他应用程序的数据。总体来说思路还是非常简单的，只需要获取到该应用程序的内容 URI，然后借助 ContentResolver 进行



CRUD 操作就可以了。可是你有没有想过，那些提供外部访问接口的应用程序都是如何实现这种功能的呢？它们又是怎样保证数据的安全性，使得隐私数据不会泄漏出去？学习完本节的知识后，你的疑惑将会被一一解开。

### 7.3.1 创建内容提供器的步骤

前面已经提到过，如果想要实现跨程序共享数据的功能，官方推荐的方式就是使用内容提供器，可以通过新建一个类去继承 `ContentProvider` 的方式来创建一个自己的内容提供器。`ContentProvider` 类中有六个抽象方法，我们在使用子类继承它的时候，需要将这六个方法全部重写。新建 `MyProvider` 继承自 `ContentProvider`，代码如下所示：

```
public class MyProvider extends ContentProvider {

    @Override
    public boolean onCreate() {
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
        return 0;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
}
```

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

}
```

在这六个方法中，相信大多数你都已经非常熟悉了，我再来简单介绍一下吧。

1. onCreate()

初始化内容提供器的时候调用。通常会在这里完成对数据库的创建和升级等操作，返回 true 表示内容提供器初始化成功，返回 false 则表示失败。注意，只有当存在 ContentResolver 尝试访问我们程序中的数据时，内容提供器才会被初始化。

2. query()

从内容提供器中查询数据。使用 uri 参数来确定查询哪张表，projection 参数用于确定查询哪些列，selection 和 selectionArgs 参数用于约束查询哪些行，sortOrder 参数用于对结果进行排序，查询的结果存放在 Cursor 对象中返回。

3. insert()

向内容提供器中添加一条数据。使用 uri 参数来确定要添加到的表，待添加的数据保存在 values 参数中。添加完成后，返回一个用于表示这条新记录的 URI。

4. update()

更新内容提供器中已有的数据。使用 uri 参数来确定更新哪一张表中的数据，新数据保存在 values 参数中，selection 和 selectionArgs 参数用于约束更新哪些行，受影响的行数将作为返回值返回。

5. delete()

从内容提供器中删除数据。使用 uri 参数来确定删除哪一张表中的数据，selection 和 selectionArgs 参数用于约束删除哪些行，被删除的行数将作为返回值返回。

6. getType()

根据传入的内容 URI 来返回相应的 MIME 类型。

可以看到，几乎每一个方法都会带有 Uri 这个参数，这个参数也正是调用 ContentResolver 的增删改查方法时传递过来的。而现在，我们需要对传入的 Uri 参数进行解析，从中分析出调用方期望访问的表和数据。

回顾一下，一个标准的内容 URI 写法是这样的：

```
content://com.example.app.provider/table1
```

这就表示调用方期望访问的是 com.example.app 这个应用的 table1 表中的数据。除此之外，我们还可以在这个内容 URI 的后面加上一个 id，如下所示：

```
content://com.example.app.provider/table1/1
```

这就表示调用方期望访问的是 com.example.app 这个应用的 table1 表中 id 为 1 的数据。

内容 URI 的格式主要就只有以上两种，以路径结尾就表示期望访问该表中所有的数据，以 id 结尾就表示期望访问该表中拥有相应 id 的数据。我们可以使用通配符的方式来分别匹配这两种格式的内容 URI，规则如下。

1. \*: 表示匹配任意长度的任意字符
2. #: 表示匹配任意长度的数字

所以，一个能够匹配任意表的内容 URI 格式就可以写成：

```
content://com.example.app.provider/*
```

而一个能够匹配 table1 表中任意一行数据的内容 URI 格式就可以写成：

```
content://com.example.app.provider/table1/#
```

接着，我们再借助 UriMatcher 这个类就可以轻松地实现匹配内容 URI 的功能。UriMatcher 中提供了一个 addURI() 方法，这个方法接收三个参数，可以分别把权限、路径和一个自定义代码传进去。这样，当调用 UriMatcher 的 match() 方法时，就可以将一个 Uri 对象传入，返回值是某个能够匹配这个 Uri 对象所对应的自定义代码，利用这个代码，我们就可以判断出调用方期望访问的是哪张表中的数据了。修改 MyProvider 中的代码，如下所示：

```
public class MyProvider extends ContentProvider {

    public static final int TABLE1_DIR = 0;

    public static final int TABLE1_ITEM = 1;

    public static final int TABLE2_DIR = 2;

    public static final int TABLE2_ITEM = 3;

    private static UriMatcher uriMatcher;

    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("com.example.app.provider", "table1", TABLE1_DIR);
        uriMatcher.addURI("com.example.app.provider ", "table1/#", TABLE1_ITEM);
        uriMatcher.addURI("com.example.app.provider ", "table2", TABLE2_ITEM);
        uriMatcher.addURI("com.example.app.provider ", "table2/#", TABLE2_ITEM);
    }
    .....
}
```

```

@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
    switch (uriMatcher.match(uri)) {
        case TABLE1_DIR:
            // 查询table1表中的所有数据
            break;
        case TABLE1_ITEM:
            // 查询table1表中的单条数据
            break;
        case TABLE2_DIR:
            // 查询table2表中的所有数据
            break;
        case TABLE2_ITEM:
            // 查询table2表中的单条数据
            break;
        default:
            break;
    }
    .....
}
.....
}

```

可以看到，MyProvider 中新增了四个整型常量，其中 TABLE1\_DIR 表示访问 table1 表中的所有数据，TABLE1\_ITEM 表示访问 table1 表中的单条数据，TABLE2\_DIR 表示访问 table2 表中的所有数据，TABLE2\_ITEM 表示访问 table2 表中的单条数据。接着在静态代码块里我们创建了 UriMatcher 的实例，并调用 addURI() 方法，将期望匹配的内容 URI 格式传递进去，注意这里传入的路径参数是可以使用通配符的。然后当 query() 方法被调用的时候，就会通过 UriMatcher 的 match() 方法对传入的 Uri 对象进行匹配，如果发现 UriMatcher 中某个内容 URI 格式成功匹配了该 Uri 对象，则会返回相应的自定义代码，然后我们就可以判断出调用方期望访问的到底是什么数据了。

上述代码只是以 query() 方法为例做了个示范，其实 insert()、update()、delete() 这几个方法的实现也是差不多的，它们都会携带 Uri 这个参数，然后同样利用 UriMatcher 的 match() 方法判断出调用方期望访问的是哪张表，再对该表中的数据进行相应的操作就可以了。

除此之外，还有一个方法你会比较陌生，即 getType() 方法。它是所有的内容提供者都必须提供的一个方法，用于获取 Uri 对象所对应的 MIME 类型。一个内容 URI 所对应的 MIME 字符串主要由三部分组成，Android 对这三个部分做了如下格式规定。

1. 必须以 vnd 开头。
2. 如果内容 URI 以路径结尾，则后接 android.cursor.dir/，如果内容 URI 以 id 结尾，则后接 android.cursor.item/。
3. 最后接上 vnd.<authority>.<path>。

所以，对于 content://com.example.app.provider/table1 这个内容 URI，它所对应的 MIME 类型就可以写成：

```
vnd.android.cursor.dir/vnd.com.example.app.provider.table1
```

对于 content://com.example.app.provider/table1/1 这个内容 URI，它所对应的 MIME 类型就可以写成：

```
vnd.android.cursor.item/vnd.com.example.app.provider.table1
```

现在我们可以继续完善 MyProvider 中的内容了，这次来实现 getType() 方法中的逻辑，代码如下所示：

```
public class MyProvider extends ContentProvider {
    .....
    @Override
    public String getType(Uri uri) {
        switch (uriMatcher.match(uri)) {
            case TABLE1_DIR:
                return "vnd.android.cursor.dir/vnd.com.example.app.provider.
table1";
            case TABLE1_ITEM:
                return "vnd.android.cursor.item/vnd.com.example.app.provider.
table1";
            case TABLE2_DIR:
                return "vnd.android.cursor.dir/vnd.com.example.app.provider.
table2";
            case TABLE2_ITEM:
                return "vnd.android.cursor.item/vnd.com.example.app.provider.
table2";
            default:
                break;
        }
        return null;
    }
}
```

到这里，一个完整的内容提供器就创建完成了，现在任何一个应用程序都可以使用 `ContentResolver` 来访问我们程序中的数据。那么前面所提到的，如何才能保证隐私数据不会泄漏出去呢？其实多亏了内容提供器的良好机制，这个问题在不知不觉中已经被解决了。因为所有的 CRUD 操作都一定要匹配到相应内容 URI 格式才能进行的，而我们当然不可能向 `UriMatcher` 中添加隐私数据的 URI，所以这部分数据根本无法被外部程序访问到，安全问题也就不存在了。

好了，创建内容提供器的步骤你也已经清楚了，下面就来实战一下，真正体验一回跨程序数据共享的功能。

### 7.3.2 实现跨程序数据共享

简单起见，我们还是在上一章中 `DatabaseTest` 项目的基础上继续开发，通过内容提供器来给它加入外部访问接口。打开 `DatabaseTest` 项目，首先将 `MyDatabaseHelper` 中使用 `Toast` 弹出创建数据库成功的提示去除掉，因为跨程序访问时我们不能直接使用 `Toast`。然后添加一个 `DatabaseProvider` 类，代码如下所示：

```
public class DatabaseProvider extends ContentProvider {

    public static final int BOOK_DIR = 0;

    public static final int BOOK_ITEM = 1;

    public static final int CATEGORY_DIR = 2;

    public static final int CATEGORY_ITEM = 3;

    public static final String AUTHORITY = "com.example.databasetest.provider";

    private static UriMatcher uriMatcher;

    private MyDatabaseHelper dbHelper;

    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI(AUTHORITY, "book", BOOK_DIR);
        uriMatcher.addURI(AUTHORITY, "book/#", BOOK_ITEM);
        uriMatcher.addURI(AUTHORITY, "category", CATEGORY_DIR);
        uriMatcher.addURI(AUTHORITY, "category/#", CATEGORY_ITEM);
    }
}
```

```
@Override
public boolean onCreate() {
    dbHelper = new MyDatabaseHelper(getContext(), "BookStore.db", null, 2);
    return true;
}

@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
    // 查询数据
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    Cursor cursor = null;
    switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            cursor = db.query("Book", projection, selection, selectionArgs,
null, null, sortOrder);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            cursor = db.query("Book", projection, "id = ?", new String[]
{ bookId }, null, null, sortOrder);
            break;
        case CATEGORY_DIR:
            cursor = db.query("Category", projection, selection,
selectionArgs, null, null, sortOrder);
            break;
        case CATEGORY_ITEM:
            String categoryId = uri.getPathSegments().get(1);
            cursor = db.query("Category", projection, "id = ?", new String[]
{ categoryId }, null, null, sortOrder);
            break;
        default:
            break;
    }
    return cursor;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
```

```
// 添加数据
SQLiteDatabase db = dbHelper.getWritableDatabase();
Uri uriReturn = null;
switch (uriMatcher.match(uri)) {
    case BOOK_DIR:
    case BOOK_ITEM:
        long newBookId = db.insert("Book", null, values);
        uriReturn = Uri.parse("content://" + AUTHORITY + "/book/" +
newBookId);
        break;
    case CATEGORY_DIR:
    case CATEGORY_ITEM:
        long newCategoryId = db.insert("Category", null, values);
        uriReturn = Uri.parse("content://" + AUTHORITY + "/category/" +
newCategoryId);
        break;
    default:
        break;
}
return uriReturn;
}

@Override
public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
    // 更新数据
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    int updatedRows = 0;
    switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            updatedRows = db.update("Book", values, selection, selectionArgs);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            updatedRows = db.update("Book", values, "id = ?", new String[]
{ bookId });
            break;
        case CATEGORY_DIR:
            updatedRows = db.update("Category", values, selection,
selectionArgs);
    }
```



```
        break;
    case CATEGORY_ITEM:
        String categoryId = uri.getPathSegments().get(1);
        updatedRows = db.update("Category", values, "id = ?", new String[]
{ categoryId });
        break;
    default:
        break;
    }
    return updatedRows;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    // 删除数据
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    int deletedRows = 0;
    switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            deletedRows = db.delete("Book", selection, selectionArgs);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            deletedRows = db.delete("Book", "id = ?", new String[] { bookId });
            break;
        case CATEGORY_DIR:
            deletedRows = db.delete("Category", selection, selectionArgs);
            break;
        case CATEGORY_ITEM:
            String categoryId = uri.getPathSegments().get(1);
            deletedRows = db.delete("Category", "id = ?", new String[]
{ categoryId });
            break;
        default:
            break;
    }
    return deletedRows;
}
```

```
@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            return "vnd.android.cursor.dir/vnd.com.example.databasetest.
provider.book";
        case BOOK_ITEM:
            return "vnd.android.cursor.item/vnd.com.example.databasetest.
provider.book";
        case CATEGORY_DIR:
            return "vnd.android.cursor.dir/vnd.com.example.databasetest.
provider.category";
        case CATEGORY_ITEM:
            return "vnd.android.cursor.item/vnd.com.example.databasetest.
provider.category";
    }
    return null;
}
}
```

代码虽然很长，不过不用担心，这些内容都非常容易理解，因为使用到的全部都是上一小节中我们学到的知识。首先在类的一开始，同样是定义了四个常量，分别用于表示访问 Book 表中的所有数据、访问 Book 表中的单条数据、访问 Category 表中的所有数据和访问 Category 表中的单条数据。然后在静态代码块里对 UriMatcher 进行了初始化操作，将期望匹配的几种 URI 格式添加了进去。

接下来就是每个抽象方法的具体实现了，先来看下 onCreate()方法，这个方法的代码很短，就是创建了一个 MyDatabaseHelper 的实例，然后返回 true 表示内容提供者初始化成功，这时数据库就已经完成了创建或升级操作。

接着看一下 query()方法，在这个方法中先获取到了 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要访问哪张表，再调用 SQLiteDatabase 的 query() 进行查询，并将 Cursor 对象返回就好了。注意当访问单条数据的时候有一个细节，这里调用了 Uri 对象的 getPathSegments() 方法，它会将内容 URI 权限之后的部分以 “/” 符号进行分割，并把分割后的结果放入到一个字符串列表中，那这个列表的第 0 个位置存放的就是路径，第 1 个位置存放的就是 id 了。得到了 id 之后，再通过 selection 和 selectionArgs 参数进行约束，就实现了查询单条数据的功能。

再往后就是 insert()方法，同样它也是先获取到了 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要往哪张表里添加数据，再调用 SQLiteDatabase 的 insert() 方法进行

添加就可以了。注意 insert()方法要求返回一个能够表示这条新增数据的 URI，所以我们还需要调用 Uri.parse()方法来将一个内容 URI 解析成 Uri 对象，当然这个内容 URI 是以新增数据的 id 结尾的。

接下来就是 update()方法了，相信这个方法中的代码已经完全难不倒你了。也是先获取 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要更新哪张表里的数据，再调用 SQLiteDatabase 的 update()方法进行更新就好了，受影响的行数将作为返回值返回。

下面是 delete()方法，是不是感觉越到后面越轻松了？因为你已经渐入佳境，真正地找到窍门了。这里仍然是先获取到 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要删除哪张表里的数据，再调用 SQLiteDatabase 的 delete()方法进行删除就好了，被删除的行数将作为返回值返回。

最后是 getType()方法，这个方法中的代码完全是按照上一节中介绍的格式规则编写的，相信已经没有什么解释的必要了。

这样我们就将内容提供器中的代码全部编写完了，不过离实现跨程序数据共享的功能还差了一小步，因为还需要将内容提供器在 AndroidManifest.xml 文件中注册才可以，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.databasetest"
    android:versionCode="1"
    android:versionName="1.0" >
    .....
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        .....
        <provider
            android:name="com.example.databasetest.DatabaseProvider"
            android:authorities="com.example.databasetest.provider" >
            </provider>
        </application>
    </manifest>
```

可以看到，这里我们使用了<provider>标签来对 DatabaseProvider 这个内容提供器进行注册，在 android:name 属性中指定了该类的全名，又在 android:authorities 属性中指定了该内容提供器的权限。

现在 DatabaseTest 这个项目就已经拥有了跨程序共享数据的功能了，我们赶快来尝试一

下。首先需要将 DatabaseTest 程序从模拟器中删除掉，以防止上一章中产生的遗留数据对我们造成干扰。然后运行一下项目，将 DatabaseTest 程序重新安装在模拟器上了。接着关闭掉 DatabaseTest 这个项目，并创建一个新项目 ProviderTest，我们就将通过这个程序去访问 DatabaseTest 中的数据。

还是先来编写一下布局文件吧，修改 activity\_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/add_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add To Book" />

    <Button
        android:id="@+id/query_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Query From Book" />

    <Button
        android:id="@+id/update_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update Book" />

    <Button
        android:id="@+id/delete_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Delete From Book" />

</LinearLayout>
```

布局文件很简单，里面放置了四个按钮，分别用于添加、查询、修改和删除数据的。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private String newId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button addData = (Button) findViewById(R.id.add_data);
        addData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 添加数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book");
                ContentValues values = new ContentValues();
                values.put("name", "A Clash of Kings");
                values.put("author", "George Martin");
                values.put("pages", 1040);
                values.put("price", 22.85);
                Uri newUri = getContentResolver().insert(uri, values);
                newId = newUri.getPathSegments().get(1);
            }
        });
        Button queryData = (Button) findViewById(R.id.query_data);
        queryData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 查询数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book");
                Cursor cursor = getContentResolver().query(uri, null, null,
null, null);
                if (cursor != null) {
                    while (cursor.moveToNext()) {
                        String name = cursor.getString(cursor.
getColumnIndex("name"));
                        String author = cursor.getString(cursor.
getColumnIndex("author"));
                        int pages = cursor.getInt(cursor.getColumnIndex
("pages"));
```

```

        double price = cursor.getDouble(cursor.
getColumnIndex("price"));
        Log.d("MainActivity", "book name is " + name);
        Log.d("MainActivity", "book author is " + author);
        Log.d("MainActivity", "book pages is " + pages);
        Log.d("MainActivity", "book price is " + price);
    }
    cursor.close();
}
});
Button updateData = (Button) findViewById(R.id.update_data);
updateData.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        // 更新数据
        Uri uri = Uri.parse("content://com.example.databasetest.
provider/book/" + newId);
        ContentValues values = new ContentValues();
        values.put("name", "A Storm of Swords");
        values.put("pages", 1216);
        values.put("price", 24.05);
        getResolver().update(uri, values, null, null);
    }
});
Button deleteData = (Button) findViewById(R.id.delete_data);
deleteData.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        // 删除数据
        Uri uri = Uri.parse("content://com.example.databasetest.
provider/book/" + newId);
        getResolver().delete(uri, null, null);
    }
});
}
}

```

可以看到，我们分别在这四个按钮的点击事件里面处理了增删改查的逻辑。添加数据的时候，首先调用了 `Uri.parse()` 方法将一个内容 URI 解析成 `Uri` 对象，然后把要添加的数据都存放到 `ContentValues` 对象中，接着调用 `ContentResolver` 的 `insert()` 方法执行添加操作就可以了。注意 `insert()` 方法会返回一个 `Uri` 对象，这个对象中包含了新增数据的 `id`，我们通过 `getPathSegments()` 方法将这个 `id` 取出，稍后会用到它。

查询数据的时候，同样是调用了 `Uri.parse()` 方法将一个内容 URI 解析成 `Uri` 对象，然后调用 `ContentResolver` 的 `query()` 方法去查询数据，查询的结果当然还是存放在 `Cursor` 对象中的。之后对 `Cursor` 进行遍历，从中取出查询结果，并一一打印出来。

更新数据的时候，也是先将内容 URI 解析成 `Uri` 对象，然后把想要更新的数据存放到 `ContentValues` 对象中，再调用 `ContentResolver` 的 `update()` 方法执行更新操作就可以了。注意这里我们为了不想让 `Book` 表中其他的行受到影响，在调用 `Uri.parse()` 方法时，给内容 URI 的尾部增加了一个 `id`，而这个 `id` 正是添加数据时所返回的。这就表示我们只希望更新刚刚添加的那条数据，`Book` 表中的其他行都不会受影响。

删除数据的时候，也是使用同样的方法解析了一个以 `id` 结尾的内容 URI，然后调用 `ContentResolver` 的 `delete()` 方法执行删除操作就可以了。由于我们在内容 URI 里指定了一个 `id`，因此只会删掉拥有相应 `id` 的那行数据，`Book` 表中的其他数据都不会受影响。

现在运行一下 `ProviderTest` 项目，会显示如图 7.4 所示的界面。

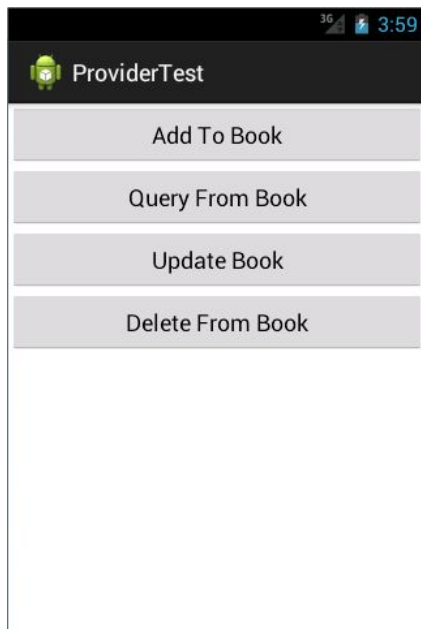


图 7.4

点击一下 Add To Book 按钮，此时数据就应该已经添加到 DatabaseTest 程序的数据库中了，我们可以通过点击 Query From Book 按钮来检查一下，打印日志如图 7.5 所示。

Tag	Text
MainActivity	book name is A Clash of Kings
MainActivity	book author is George Martin
MainActivity	book pages is 1040
MainActivity	book price is 22.85

图 7.5

然后点击一下 Update Book 按钮来更新数据，再点击一下 Query From Book 按钮进行检查，结果如图 7.6 所示。

Tag	Text
MainActivity	book name is A Storm of Swords
MainActivity	book author is George Martin
MainActivity	book pages is 1216
MainActivity	book price is 24.05

图 7.6

最后点击 Delete From Book 按钮删除数据，此时再点击 Query From Book 按钮就查询不到数据了。

由此可以看出，我们的跨程序共享数据功能已经成功实现了！现在不仅是 ProviderTest 程序，任何一个程序都可以轻松访问 DatabaseTest 中的数据，而且我们还丝毫不用担心隐私数据泄漏的问题。

到这里，与内容提供器相关的重要内容就基本全部介绍完了，下面就让我们再次进入本书的特殊环节，学习一下关于 Git 更多的用法。

经验值：+7000

目前经验值：48905

级别：资深鸟

获赠宝物：拜会自定义内容提供猪。自定义内容提供猪是内容提供猪的堂兄，有着和内容提供猪一样的好脾气，友善、爱干净、助人为乐。唯一的不同是自定义内容提供猪喜欢神飞丝牌子的洗发露。自定义内容提供猪善于经商，在乡下开了一个连锁的便民小超市，已在周边神县开了 5 家连锁店。他向我赠送了全套神洁品牌的洗漱用品。说实话，我确实好多天没洗澡了，差不多已经有一个多星期没有洗澡了，倒不是住不起店，事实上我现在比较富有，



一路上斩获的物资让我换了不少盘缠，没有住店，日夜兼程的原因是想早点实现目标，成为 Android 开发高手。但现在望着手中的洗浴用品，我突然感到我累了，于是我当下选择住店，洗个澡，也干净一下，不能让猪兄瞧不起不是。洗完澡后，我惊讶地发现我身体的光芒比拜别 Git 领主时已增亮了许多。酣睡一夜。继续前进。

## 7.4 Git 时间，版本控制工具进阶

在上一次的 Git 时间里，我们学习了关于 Git 最基本的用法，包括安装 Git、创建代码仓库，以及提交本地代码。本节中我们将要学习更多的使用技巧，不过在开始之前先要把准备工作做好。

所谓的准备工作就是要给一个项目创建代码仓库，这里就选择在 ProviderTest 项目中创建吧，打开 Git Bash，进入到这个项目的根目录下面，然后执行 git init 命令，如图 7.7 所示。

```
Tony@TONY-PC ~  
$ cd f:  
  
Tony@TONY-PC /f  
$ cd codes/AndroidFirstLine/ProviderTest  
  
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest  
$ git init  
Initialized empty Git repository in f:/codes/AndroidFirstLine/ProviderTest/.git/
```

图 7.7

这样准备工作就已经完成了，让我们继续开始 Git 之旅吧。

### 7.4.1 忽略文件

代码仓库现在已经是创建好了，接下来我们应该去提交 ProviderTest 项目中的代码。不过在提交之前你也许应该思考一下，是不是所有的文件都需要加入到版本控制当中呢？

在第一章介绍 Android 项目结构的时候有提到过，bin 目录和 gen 目录下的文件都是会自动生成的，我们不应该将这部分文件添加到版本控制当中，否则有可能会对文件的自动生成造成影响，那么如何才能实现这样的效果呢？

Git 提供了一种可配性很强的机制来允许用户将指定的文件或目录排除在版本控制之外，它会检查代码仓库的根目录下是否存在一个名为.gitignore的文件，如果存在的话就去一行行读取这个文件中的内容，并把每一行指定的文件或目录排除在版本控制之外。注意.gitignore中指定的文件或目录是可以使用“\*”通配符的。

现在，我们在 ProviderTest 项目的根目录下创建一个名为.gitignore的文件，然后编辑这个文件中的内容，如图 7.8 所示。

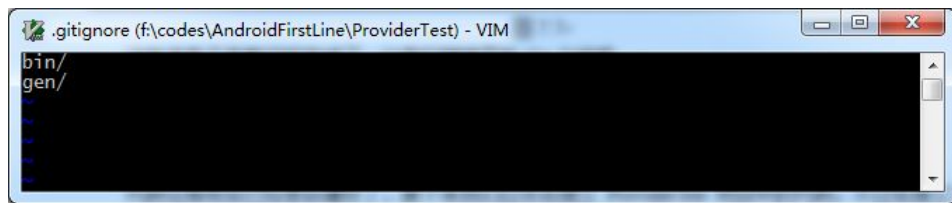


图 7.8

这样就表示把 bin 目录和 gen 目录下的所有文件都忽略掉，从而使用得它们不会加入到版本控制当中。

排除了 bin 和 gen 这两个目录以后，我们就可以提交代码了，先使用 add 命令将所有的文件进行添加，如下所示：

```
git add .
```

然后执行 commit 命令完成提交，如下所示：

```
git commit -m "First commit."
```

### 7.4.2 查看修改内容

在进行了第一次代码提交之后，我们后面还可能会对项目不断地进行维护，添加新功能等。比较理想的情况是每当完成了一小块功能，就执行一次提交。但是当某个功能牵扯到的代码比较多时，有可能写到后面的时候我们就已经忘记前面修改了什么东西了。遇到这种情况时不用担心，Git 全部都帮你记着呢！下面我们就来学习一下，如何使用 Git 来自上次提交后文件修改的内容。

查看文件修改情况的方法非常简单，只需要使用 status 命令就可以了，在项目的根目录下输入如下命令：

```
git status
```

然后 Git 会提示目前项目中没有任何可提交的文件，因为我们刚刚才提交过嘛。现在对 ProviderTest 项目中的代码稍做一下改动，修改 MainActivity 中的代码，如下所示：

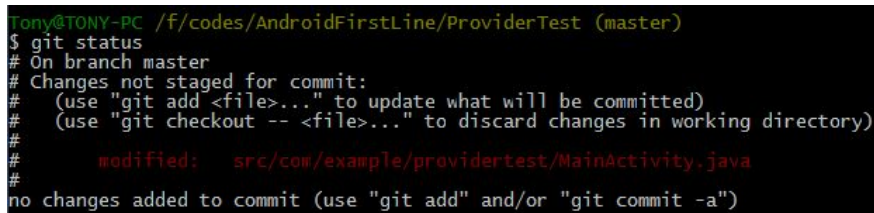
```
public class MainActivity extends Activity {  
    .....  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        .....  
        addData.setOnClickListener(new OnClickListener() {  
            @Override  
            public void onClick(View v) {
```

```

.....
values.put("price", 55.55);
.....
    }
    });
    .....
}
}
}

```

这里仅仅是在添加数据的时候，将书的价格由 22.85 改成了 55.55。然后重新输入 `git status` 命令，这次结果如图 7.9 所示。



```

Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   src/com/example/providertest/MainActivity.java
#
no changes added to commit (use "git add" and/or "git commit -a")

```

图 7.9

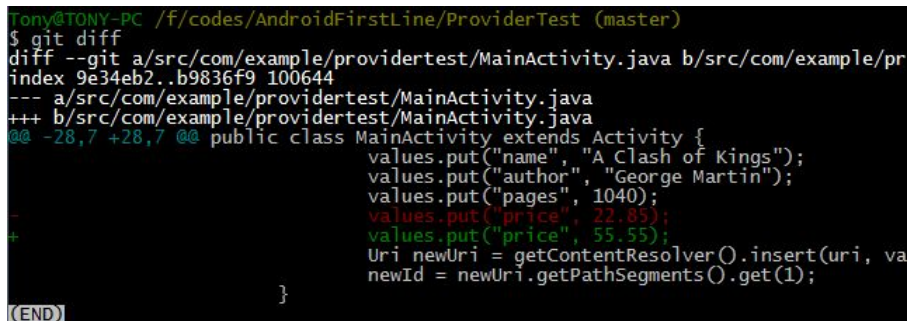
可以看到，Git 提醒我们 `MainActivity.java` 这个文件已经发生了更改，那么如何才能看到更改的内容呢？这就需要借助 `diff` 命令了，用法如下所示：

**git diff**

这样可以查看到所有文件的更改内容，如果你只想查看 `MainActivity.java` 这个文件的更改内容，可以使用如下命令：

**git diff src/com/example/providertest/MainActivity.java**

命令的执行结果如图 7.10 所示。



```

Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git diff
diff --git a/src/com/example/providertest/MainActivity.java b/src/com/example/pr
index 9e34eb2..b9836f9 100644
--- a/src/com/example/providertest/MainActivity.java
+++ b/src/com/example/providertest/MainActivity.java
@@ -28,7 +28,7 @@ public class MainActivity extends Activity {
     values.put("name", "A Clash of Kings");
     values.put("author", "George Martin");
     values.put("pages", 1040);
-    values.put("price", 22.85);
+    values.put("price", 55.55);
     Uri newUri = getContentResolver().insert(uri, va
     newId = newUri.getPathSegments().get(1);
}
(END)

```

图 7.10

其中，减号代表删除的部分，加号代表添加的部分。从图中我们就可以明显地看出，书的价格由 22.85 被修改成了 55.55。

### 7.4.3 撤销未提交的修改

有的时候我们代码可能会写得过于草率，以至于原本正常的功能，结果反倒被我们改出了问题。遇到这种情况时也不用着急，因为只要代码还未提交，所有修改的内容都是可以撤销的。

比如在上一小节中我们修改了 MainActivity 里一本书的价格，现在如果想要撤销这个修改就可以使用 checkout 命令，用法如下所示：

```
git checkout src/com/example/providertest/MainActivity.java
```

执行了这个命令之后，我们对 MainActivity.java 这个文件所做的一切修改就应该都被撤销了。重新运行 git status 命令检查一下，结果如图 7.11 所示。

```
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git status
# On branch master
nothing to commit, working directory clean
```

图 7.11

可以看到，当前项目中没有任何可提交的文件，说明撤销操作确实是成功了。

不过这种撤销方式只适用于那些还没有执行过 add 命令的文件，如果某个文件已经被添加过了，这种方式就无法撤销其更改的内容，我们来做个试验瞧一瞧。

首先仍然是将 MainActivity 中那本书的价格改成 55.55，然后输入如下命令：

```
git add .
```

这样就把所有修改的文件都进行了添加，可以输入 git status 来检查一下，结果如图 7.12 所示。

```
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   src/com/example/providertest/MainActivity.java
#
```

图 7.12

现在我们再执行一遍 checkout 命令，你会发现 MainActivity 仍然是处于添加状态，所修改的内容无法撤销掉。

这种情况应该怎么办？难道我们还没法后悔了？当然不是，只不过对于已添加的文件我

们应该先对其取消添加，然后才可以撤回提交。取消添加使用的是 `reset` 命令，用法如下所示：

```
git reset HEAD src/com/example/providertest/MainActivity.java
```

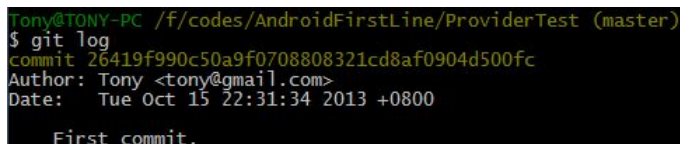
然后再运行一遍 `git status` 命令，你就会发现 `MainActivity.java` 这个文件重新变回了未添加状态，此时就可以使用 `checkout` 命令来将修改的内容进行撤销了。

#### 7.4.4 查看提交记录

当 `ProviderTest` 这个项目开发了几个月之后，我们可能已经执行过上百次的提交操作了，这个时候估计你早就已经忘记每次提交都修改了哪些内容。不过没关系，忠实的 `Git` 一直都帮我们清清楚楚地记录着呢！可以使用 `log` 命令查看历史提交信息，用法如下所示：

```
git log
```

由于目前我们只执行过一次提交，所以能看到的信息很少，如图 7.13 所示。



```
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git log
commit 26419f990c50a9f0708808321cd8af0904d500fc
Author: Tony <tony@gmail.com>
Date: Tue Oct 15 22:31:34 2013 +0800

    First commit.
```

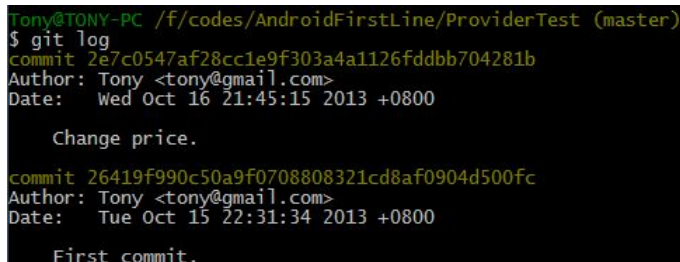
图 7.13

可以看到，每次提交记录都会包含提交 id、提交人、提交日期，以及提交描述这四个信息。那么我们再次将书价修改成 55.55，然后执行一次提交操作，如下所示：

```
git add .
```

```
git commit -m "Change price."
```

现在重新执行 `git log` 命令，结果如图 7.14 所示。



```
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git log
commit 2e7c0547af28cc1e9f303a4a1126fddbb704281b
Author: Tony <tony@gmail.com>
Date: Wed Oct 16 21:45:15 2013 +0800

    Change price.

commit 26419f990c50a9f0708808321cd8af0904d500fc
Author: Tony <tony@gmail.com>
Date: Tue Oct 15 22:31:34 2013 +0800

    First commit.
```

图 7.14

当提交记录非常多的时候，如果我们只想查看其中一条记录，可以在命令中指定该记录的 id，并加上 `-1` 参数表示我们只想看到一行记录，如下所示：

```
git log 2e7c0547af28cc1e9f303a4a1126fddbb704281b -1
```

而如果想要查看这条提交记录具体修改了什么内容，可以在命令中加入-p 参数，命令如下：

```
git log 2e7c0547af28cc1e9f303a4a1126fddbb704281b -1 -p
```

查询出的结果如图 7.15 所示，其中减号代表删除的部分，加号代表添加的部分。

```
Tony@TONY-PC /f/codes/AndroidFirstLine/ProviderTest (master)
$ git log 2e7c0547af28cc1e9f303a4a1126fddbb704281b -1 -p
commit 2e7c0547af28cc1e9f303a4a1126fddbb704281b
Author: Tony <tony@gmail.com>
Date:   Wed Oct 16 21:45:15 2013 +0800

    Change price.

diff --git a/src/com/example/providertest/MainActivity.java b/src/com/example/pr
index 9e34eb2..b9836f9 100644
--- a/src/com/example/providertest/MainActivity.java
+++ b/src/com/example/providertest/MainActivity.java
@@ -28,7 +28,7 @@ public class MainActivity extends Activity {
     values.put("name", "A Clash of Kings");
     values.put("author", "George Martin");
     values.put("pages", 1040);
-    values.put("price", 22.85);
+    values.put("price", 55.55);
     Uri newUri = getContentResolver().insert(uri, va
     newId = newUri.getPathSegments().get(1);
}

(END)
```

图 7.15

好了，本次的 Git 时间就到这里，下面我们来对本章中所学的知识做个回顾吧。

## 7.5 小结与点评

本章的内容比较少，而且很多时候都是在使用上一章中学习的数据库知识，所以理解这部分内容对你来说应该还是比较轻松的吧。在本章中，我们主要学习了内容提供器的相关内容，以实现跨程序数据共享的功能。现在你不仅知道了如何去访问其他程序中的数据，还学会了怎样创建自己的内容提供器来共享数据，收获还是挺大的吧。

不过每次在创建内容提供器的时候，你都需要提醒一下自己，我是不是应该这么做？因为只有真正需要将数据共享出去的时候我们才应该创建内容提供器，仅仅是用于程序内部访问的数据就没有必要这么做，所以千万别对它进行滥用。

在连续学了几章系统机制方面的内容之后是不是感觉有些枯燥？那么下一章中我们就来换换口味，学习一下 Android 多媒体方面的知识吧。

经验值：+3000

目前经验值：51905

级别：资深鸟

获赠宝物：拜会 Git 领主儿子的属地。获赠不少盘缠。