目录

[1. Android四大基本组件 2](#_Toc518913498)

[1.1 Activity 2](#_Toc518913499)

[1.2 Service 3](#_Toc518913500)

[1.3 Broadcast Receiver 3](#_Toc518913501)

[1.4 Content Provider 4](#_Toc518913502)

[2. 布局Layout 4](#_Toc518913503)

[3. 组件Widget 5](#_Toc518913504)

[3.1 列表视图ListView 5](#_Toc518913505)

[3.1.1 Adapter之ViewHolder的使用 5](#_Toc518913506)

[3.2 RecyclerView 10](#_Toc518913507)

[4. Intent 10](#_Toc518913508)

[4.1 Intent Filter 11](#_Toc518913509)

[5. Android消息处理 11](#_Toc518913510)

[5.1事件与监听器 11](#_Toc518913511)

[5.2 Handler、Looper、MessageQueue和Message理解 11](#_Toc518913512)

[6. 系统级服务 12](#_Toc518913513)

[6.1 Notification用法 12](#_Toc518913514)

[7. 资源访问 13](#_Toc518913515)

[7.1 R类 13](#_Toc518913516)

[7.2 Android中的数据存取 13](#_Toc518913517)

[7.3 res资源文件的使用 13](#_Toc518913518)

[8.Gradle详解 19](#_Toc518913519)

[8.1 Gradle简介 19](#_Toc518913520)

[8.2 Groovy介绍 19](#_Toc518913521)

[碎片知识积累 20](#_Toc518913522)

[DPI、DIP、PX、DP、SP的区别 20](#_Toc518913523)

[Log图文详解 20](#_Toc518913524)

[相关术语的解析 21](#_Toc518913525)

# 1. Android四大基本组件

## 1.1 Activity

Activity代表一个单独的屏幕或一个和用户交互的可视化界面。在其之上是一个Window对象，再上是一个布局容器对象Layout，再上是各种展示组件widget。

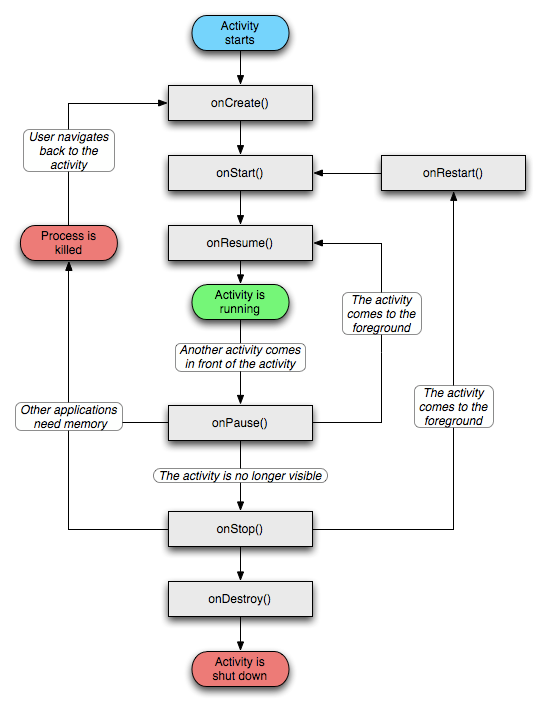
***1.1.1 Activity生命周期——***

打开应用时先后执行了onCreate()->onStart()->onResume三个方法

当我们按BACK键时，我们这个应用程序将结束，这时候我们将先后调用onPause()->onStop()->onDestory()三个方法

当我们按HOME的时候，Activity先后执行了onPause()->onStop()这两个方法

当我们再次启动ActivityDemo应用程序时，则先后分别执行了onRestart()->onStart()->onResume()三个方法



***1.1.2 Activity的调用——***

Activity的调用与激活（除了MainActivity）离不开Intent，同时也离不开主叫Activity和被叫Activity之间的关系。主叫Activity可以使用两种方法startActivity或startActivityForResult来调用被叫Activity。

第一种方法主叫方直接传递Intent对象给被叫方，将其激活，之后二者没有其他联系。被叫方可以通过getIntent方法获得传来的Intent对象，进而获取其他信息。

第二种方法主叫方调用被叫方只是将其视为一个处理，通过Intent对象传递信息给被叫方，但是等待被叫方处理完成返回结果。被叫方必须从本Activity对象调用setResult方法定义返回的处理结果（仍可以Intent形式封装数据），此时主叫方可以覆盖onActivityResult方法定义其接收到结果后的行为，该行为在被叫方执行finish方法结束后自动响应。

（更多细节和参数设定需参考reference）

## 1.2 Service

Service和Activity是并列的，亦即二者运作机理类似、调用过程类似，只是Service是一个后台进程一般不能显示。

详细而言，Service在以下几点上和Activity类似：  
①可在AndroidManif.xml中以<service>元素声明或用代码，用<intent-filter>指定可以相应的intent参数，如action  
②声明Service类时，需实现onCreate、onStart等事件回调函数  
③可使用Intent来启动Service对象

1.2.1 Service的远程调用

Android中各个组件运行在各自的进程中，彼此之间不能直接相互访问。只能通过间接方式传递参数，或远程调用。

Service可以实现轻量级远程调用lightweight RPC，即使用其他的基本组件来访问一个Service中的内容。此时一般使用bindService方法来绑定一个Service，该Service当中通过定义onBind方法返回一个由AIDL（Android Interface Definition Language）定义的接口的实现（亦即Service中的一个成员），以便另一端组件使用该实现。

AIDL实际上和Java语法大致相似，只是用AIDL定义的接口文件扩展名为.aidl，使用特定工具即可将其自动转换为该接口的Java文件。该Java文件实际上自动添加了一些内容以便该接口可以用于RPC。

客户进程实际上通过重写ServiceConnection中的onServiceConnected等函数来对Service返回内容进行操作。ServiceConnection对象将是bindService方法的参数。

## 1.3 Broadcast Receiver

BR是一个广播接收器，功能不言自明。广播与BR是该功能实现的两部分。广播就是一个发布intent的操作。BR和Service与Activity接收intent的方式类似。

仍以Activity组件为标准进行比较：  
①可在AndroidManif.xml中以<receiver>元素声明或用代码声明BR，用<intent-filter>指定可以相应的intent参数，如action  
②BR类是继承BroadcastReceiver类定义的，使用onReceive方法定义接收到intent后的动作。  
③广播是以intent为参数的sendBroadcast方法发布的。发布intent的位置可以是Activity之中。

1.3.1 标准广播

Android中存在标准广播消息，对于这些广播只需在<intent-filter>中定义相应的action即可用自定义BR接收。对于自定义的广播intent，可任意设定其action参数。

|  |  |
| --- | --- |
| 名称 | 意义 |
| ACTION\_BOOT\_COMPLETED | 系统启动完成 |
| ACTION\_TIME\_CHANGED | 时间改变 |
| ACTION\_DATE\_CHANGED | 日期改变 |
| ACTION\_TIMEZONE\_CHANGED | 时区改变 |
| ACTION\_BATTERY\_LOW | 低电量 |
| ACTION\_MEDIA\_EJECT | 插入或拔出外部媒体 |
| ACTION\_MEDIA\_BUTTON | 按下媒体按钮 |
| ACTION\_PACKAGE\_ADDED | 添加包 |
| ACTION\_PACKAGE\_REMOVED | 删除包 |

1.3.2 代码注册BR

一般在Activity.onResume() 方法中通过Context.registerReceiver ()方法来注册BR。在Activity.onPause() 中使用unregisterReceiver()来注销BR。

## 1.4 Content Provider

CP是跨应用程序共享数据的唯一方法。针对常用的数据类型（音视频、图片、通讯录等），android中已经定义了大量的CP类，位于android.provider。对于自定义数据，可自定义CP类。CP定义位于android.content，是一个抽象类。自定义的CP必须实现几个关于增删查改的通用方法，见1.4.1。

CP顾名思义是服务端（或组件）实现的对象，而客户端调用CP需要在客户端本地实现ContentResolver。CR和CP的方法是对应的，只是一个间接操作CP的客户端对象。可以推论，CR可以有很多个，且在不同客户进程中与同一个CP交互。

CR和CP之间通过特殊格式的URI进行对应或连接，类似于网址。已经被android定义的CP有既定义的URI。

1.4.1 ContentProvider常用方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| insert(Uri, ContentValues) | 插入 |
| delete(Uri, String, String[]) | 删除 |
| update(Uri, ContentValues, String, String[]) | 修改 |
| query(Uri, String[], String, String[], String) | 查询 |
| getType(Uri) | 获得MIME数据类型 |
| onCreate() | 当CP创建时调用 |
| getContext() | 获得Context对象 |

注：在CR中也是用同原型的方法操作数据内容

1.4.2 自定义ContentProvider

创建CP的步骤如下：  
①创建保存数据的文件或数据库SQLite，以及相关数据库Table类  
②定义CP的实现类  
③将实现类声明在AndroidManif.xml中，必须添加android:authorities属性。该属性也是其URI中的“域名”

# 2. 布局Layout

即对Activity中的组件如何排列进行模式定义的模板。布局之间可以内外嵌套与并列。

常见布局种类：  
线性布局Linearlayout——按照水平或垂直方向一字排列  
帧布局FrameLayout——后定义的组件覆盖前一组件，像视频帧一样按坐标定义  
表格布局TableLayout——组件像在表格中排列  
相对布局RelativeLayout——按照组件之间的相对位置描述来进行排列  
绝对布局AbsoluteLayout——通过确切坐标来定义位置，Android 2.0API后过期

实现布局定义有两种方式，一种是在资源文件layout文件夹中定义布局xml文件，然后在Activity的onCreate方法中声明该布局。另一种是硬代码方式定义。前者更加推荐。不同的布局方式使用的xml标签或属性不同，需自行另外了解记忆。

# 3. 组件Widget

## 3.1 列表视图ListView

ListView就是一个列表，可以用作菜单功能也可以单纯作为展示。

ListView的布局可以有两种方式，一种按照普通组件在layout中进行描述，另一种将Activity直接继承ListActivity，产生一个被单个ListView完全填充的Activity。不过也可以对ListActivity自定义布局，只是布局中必须含有一个id为android:id/list的ListView。因为该ListView即是ListActivity自带的那个组件。同时可选的，可以在ListActivity自定义布局中包含一个view对象，如TextView，并声明id为android:id/empty。此时当ListView对象内容为空时，该View对象显示并覆盖ListView组件。

ListView组件需要和一组数据产生联系，该数据是用户提供的列表项，可以是一个字符串数组或其他的。将数据和View组件建立关系的是Adapter对象，根据不同类型的数据要使用不同类型的Adapter。其作用就是将数据转化为可以显示在ListView中的内容。

### 3.1.1 Adapter之ViewHolder的使用

ViewHolder在Android自定义的适配器中使用。目的：优化资源，节省空间，避免重复绘制view而引起的不必要的内存损耗。

我自己以前的写法：

public class PlateAdapter extends BaseAdapter {

    private List**<Plate>** list;

    private Context context;

    public PlateAdapter(List**<Plate>** list, Context context) {

        super();

        this.list = list;

        this.context = context;

    }

    @Override

    public int getCount() {

        // TODO Auto-generated method stub

        return list.size();

    }

    @Override

    public Object getItem(int arg0) {

        // TODO Auto-generated method stub

        return list.get(arg0);

    }

    @Override

    public long getItemId(int arg0) {

        // TODO Auto-generated method stub

        return arg0;

    }

    @Override

    public View getView(int arg0, View arg1, ViewGroup arg2) {

        // TODO Auto-generated method stub

        if(arg1 == null){

            arg1 = LayoutInflater.from(context).inflate(R.layout.select\_car\_type\_list\_item, null);

        }

        TextView text = (TextView)arg1.findViewById(R.id.text);

        text.setText(list.get(arg0).getPlateType());

        return arg1;  }

}

学习过ViewHolder之后的写法：

@SuppressWarnings("unused")

public class NoticeAdapter  extends BaseAdapter{

    private Context \_context;

    private List**<ExamNotice>** \_list;

    public NoticeAdapter(Context context, List**<ExamNotice>** list) {

        super();

        this.\_context = context;

        this.\_list = list;

    }

    public void set\_list(List**<ExamNotice>** \_list) {

        this.\_list = \_list;  }

    @Override

    public int getCount() {

        return \_list.size();  }

    @Override

    public Object getItem(int arg0) {

        return \_list.get(arg0);  }

    @Override

    public long getItemId(int arg0) {

        return arg0;  }

    @Override

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

        Holder holder;

        if(null==convertView){

            convertView=View.inflate(\_context, R.layout.notice\_item, null);

            holder=new Holder();

            holder.studyPlanName=(TextView)convertView.findViewById(R.id.xxjh\_item\_name);

            holder.studyPlanDate=(TextView)convertView.findViewById(R.id.xxjh\_item\_date);

            convertView.setTag(holder);

        }else{

            holder=(Holder)convertView.getTag();

        }

        ExamNotice notice=(ExamNotice) getItem(position);

        holder.studyPlanName.setText(notice.getNoticeTitle());

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

        String noticeDate = sdf.format(notice.getNoticeDate());

        holder.studyPlanDate.setText(noticeDate);

        return convertView;  }

    private static class Holder{

        public TextView studyPlanName,studyPlanDate;  }

}

看一下官方的API：

1. /\*
2. \* Copyright (C) 2008 The Android Open Source Project
3. \*
4. \* Licensed under the Apache License, Version 2.0 (the "License");
5. \* you may not use this file except in compliance with the License.
6. \* You may obtain a copy of the License at
7. \*
8. \*      http://www.apache.org/licenses/LICENSE-2.0
9. \*
10. \* Unless required by applicable law or agreed to in writing, software
11. \* distributed under the License is distributed on an "AS IS" BASIS,
12. \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13. \* See the License for the specific language governing permissions and
14. \* limitations under the License.
15. \*/
17. package com.example.android.apis.view;
19. import android.app.ListActivity;
20. import android.content.Context;
21. import android.os.Bundle;
22. import android.view.LayoutInflater;
23. import android.view.View;
24. import android.view.ViewGroup;
25. import android.widget.BaseAdapter;
26. import android.widget.TextView;
27. import android.widget.ImageView;
28. import android.graphics.BitmapFactory;
29. import android.graphics.Bitmap;
30. import com.example.android.apis.R;
32. /\*\*
33. \* Demonstrates how to write an efficient list adapter. The adapter used in this example binds
34. \* to an ImageView and to a TextView for each row in the list.
35. \*
36. \* To work efficiently the adapter implemented here uses two techniques:
37. \* - It reuses the convertView passed to getView() to avoid inflating View when it is not necessary
38. \* - It uses the ViewHolder pattern to avoid calling findViewById() when it is not necessary
39. \*
40. \* The ViewHolder pattern consists in storing a data structure in the tag of the view returned by
41. \* getView(). This data structures contains references to the views we want to bind data to, thus
42. \* avoiding calls to findViewById() every time getView() is invoked.
43. \*/
44. public class List14 extends ListActivity {
46. private static class EfficientAdapter extends BaseAdapter {
47. private LayoutInflater mInflater;
48. private Bitmap mIcon1;
49. private Bitmap mIcon2;
51. public EfficientAdapter(Context context) {
52. // Cache the LayoutInflate to avoid asking for a new one each time.
53. mInflater = LayoutInflater.from(context);
55. // Icons bound to the rows.
56. mIcon1 = BitmapFactory.decodeResource(context.getResources(), R.drawable.icon48x48\_1);
57. mIcon2 = BitmapFactory.decodeResource(context.getResources(), R.drawable.icon48x48\_2);
58. }
60. /\*\*
61. \* The number of items in the list is determined by the number of speeches
62. \* in our array.
63. \*
64. \* @see android.widget.ListAdapter#getCount()
65. \*/
66. public int getCount() {
67. return DATA.length;
68. }
70. /\*\*
71. \* Since the data comes from an array, just returning the index is
72. \* sufficent to get at the data. If we were using a more complex data
73. \* structure, we would return whatever object represents one row in the
74. \* list.
75. \*
76. \* @see android.widget.ListAdapter#getItem(int)
77. \*/
78. public Object getItem(int position) {
79. return position;
80. }
82. /\*\*
83. \* Use the array index as a unique id.
84. \*
85. \* @see android.widget.ListAdapter#getItemId(int)
86. \*/
87. public long getItemId(int position) {
88. return position;
89. }
91. /\*\*
92. \* Make a view to hold each row.
93. \*
94. \* @see android.widget.ListAdapter#getView(int, android.view.View,
95. \*      android.view.ViewGroup)
96. \*/
97. public View getView(int position, View convertView, ViewGroup parent) {
98. // A ViewHolder keeps references to children views to avoid unneccessary calls
99. // to findViewById() on each row.
100. ViewHolder holder;
102. // When convertView is not null, we can reuse it directly, there is no need
103. // to reinflate it. We only inflate a new View when the convertView supplied
104. // by ListView is null.
105. if (convertView == null) {
106. convertView = mInflater.inflate(R.layout.list\_item\_icon\_text, null);
108. // Creates a ViewHolder and store references to the two children views
109. // we want to bind data to.
110. holder = new ViewHolder();
111. holder.text = (TextView) convertView.findViewById(R.id.text);
112. holder.icon = (ImageView) convertView.findViewById(R.id.icon);
114. convertView.setTag(holder);
115. } else {
116. // Get the ViewHolder back to get fast access to the TextView
117. // and the ImageView.
118. holder = (ViewHolder) convertView.getTag();
119. }
121. // Bind the data efficiently with the holder.
122. holder.text.setText(DATA[position]);
123. holder.icon.setImageBitmap((position & 1) == 1 ? mIcon1 : mIcon2);
125. return convertView;
126. }
128. static class ViewHolder {
129. TextView text;
130. ImageView icon;
131. }
132. }
134. @Override
135. public void onCreate(Bundle savedInstanceState) {
136. super.onCreate(savedInstanceState);
137. setListAdapter(new EfficientAdapter(this));
138. }
140. private static final String[] DATA = Cheeses.sCheeseStrings;
141. }

## 3.2 RecyclerView

RecyclerView可以说是ListView或GridView的一个替代，如其名所示，它只是具备了回收再利用View的功能，其他的功能完全由开发人员定制。

# 4. Intent

Intent是不同组件之间通信的媒介，可以用来启动Activity、Service和Broadcast。启动不同组件的方法如下：  
Activity——startActivity(), startActivityForResult()  
Service——startService(), bindService()  
Broadcasts——sendBroadcast(), sendOrderedBroadcast(), sendStickyBroadcast()

Intent的组成部分和属性如下：   
Action——一般表示一种组件动作，现在通常的含义就如其字面意义——动作，即一个操作、一个功能按键等。再具体而言，执行一个搜索命令或执行一个点赞命令就叫一个action。本质上是字符串常量。系统定义的Action常量包括Activity和Braodcast两种，也可以自定义Action。  
Data——一般和Action匹配，用以表示被动作的对象信息，如URI和MIME  
Category——表示Intent的一种分类信息  
Type——  
Component/Class——表示或指向另一个组件，组件名称由ComponentName类封装  
Extra——一个附加在Intent上的额外信息用于不同组件之间的信息传递，表示成一个键值对。当有多个键值对需要作为Extra数据连接在Intent上时，可以使用Bundle类封装键值对，再将Bundle对象设为Extra

## 4.1 Intent Filter

启动组件之前需要定位组件，分为显式Intent和隐式Intent。前者通过Component名称属性来直接定位需要启动的组件，一般只适用于同一程序内部。后者更常见，需要借助Intent Filter

每个组件都定义有一系列针对Intent属性的Intent Filter（在AndroidManifest.xml中），每个Intent可能有一系列定义了的属性。通过简单的匹配完成隐式定位目标组件。

Intent Filter中常见使用三种属性的标签，其标签名和标签属性如下：  
Action——android:name，Action标签必须非空  
Category——android:name，Category标签必须包含android.intent.category.DEFAULT，除非是特殊的情况，如LAUNCHER等，否则Category测试无效  
Data——用android:scheme, android:host, android:port, android:path表示URI格式scheme://host:port/path。用android:mimeType表示MIME类型

# 5. Android消息处理

## 5.1事件与监听器

Android系统和Java使用相同的事件处理机制，包括三个方面：  
event object——事件，事件本身的对象  
event source——事件源，产生事件的源头对象  
event listener——事件监听器，包含处理事件或响应事件的方法的对象

事件对象包含事件源的引用，事件源包含监听器的引用（列表）和针对事件调用监听器的方法，监听器只包含处理对应事件的方法。

Android中常见的事件监听器接口类：  
单击事件View.OnClickListener——需要实现onClick()方法  
焦点更改View.OnFocusChangeListener——onFocusChange()方法  
按键View.OnKeyListener——onKey()  
触碰View.OnTouchListener——onTouch()  
创建上下文菜单View.OnCreateContextMenuListener——onCreateContextMenu()

## 5.2 Handler、Looper、MessageQueue和Message理解

应用启动的时候会先调用ActivityThread的main方法，main方法会调用Looper.prepareMainLooper()创建Looper对象。一个线程最多只能创建一个Looper对象，Looper创建的同时会创建MessageQueue，一个Looper对应一个MessageQueue。除了主线程中自动创建Looper，其他子线程中必须先调用Looper.prepare()方法为本线程创建Looper对象。

Handler对象与Looper对象是绑定的，亦即与线程是绑定的。Handler()的构造方法可以使用默认的形式采用当前线程Looper，也可以传递其他线程Looper用于绑定。在其他线程中可通过handler.sendMessage()把消息发送到handler所在线程的MessageQueue中。MessageQueue借助Message对象（next）成员变量，实现单向链表，并且用一个mMessages对象表示当前待处理的消息。Looper.loop（）方法从MessageQueue中获取当前Message进行处理。子线程中初始化Handler最后要显示调用Looper.loop（）方法，其实相当于启动获取Message方法的循环，主线程中会默认调用。

Looper对象中通过不断执行queue.next()方法来对MessageQueue进行出队操作，Message被获取后，调用msg.target.dispatchMessage(msg)方法来将Message返回给target（即对应的Handler对象）。具体的操作是，若msg.callback与mCallback不为空，则调用handler.handleMessage(msg)处理消息，这个方法是用户创建handler对象时可重写的方法。至于msg.callback是一个Runnable对象，用于在handler所绑定线程中进行操作。

额外的，Handler类中的post(Runnable r)方法还是调用sendMessageDelayed方法，而且Runnable还被封装成了Message。经过类似上述的过程后，r被作为msg.callback执行run()。

额外的，View类中的post(Runnable action)方法借助Handler的post方法实现。

额外的，Activity类中的runOnUiThread()方法也是借助Handler的post方法实现，若所在线程就是主线程（即UI线程）则直接run()。

# 6. 系统级服务

系统级服务指的是手机系统自带的应用和服务。一般使用getSystemService(String service)返回系统级服务对象，即可直接在四大组件代码中使用。系统级服务一般需要附在某个已存在的且特定类型参数的intent之上与intent的本身目的协同工作，通过PendingIntent类获得这样的intent。当系统级服务被触发时，intent自动被执行所需的操作，无需再显式发布。

常用的系统级服务如下：

1. NotificationManager

String参数设为NOTIFICATION\_SERVICE。和Notification类一起使用，调用手机系统的消息提醒功能。包括震动、闪光灯等功能。

2. AlarmManager

String参数设为ALARM\_SERVICE。调用手机系统的闹钟功能。和发送给Broadcast Receiver的intent协同。

## 6.1 Notification用法

见Android常用技术积累

# 7. 资源访问

## 7.1 R类

R.java或R类，是（一般）由Android SDK在编译阶段通过aapt工具生成的，即系统自动生成，大部分情况下开发者只需调用即可。它主要用于管理资源文件res，调用它的成员可以获取资源文件id等。

R类一般生成在和每个MainActivity相同的包下。不在此路径下的类引用R类需要import，而资源文件id也在同一尺度内保持唯一性。

## 7.2 Android中的数据存取

7.2.1 Preference

Preference是一种轻量级数据存取方法，指的是被存取的数据比较少，一般以键值对形式将数据保存在一个xml文件中。

实现Preference功能的是android.content.SharedPreferences类和其内部类SharedPreferences.Editor。类的本体负责提供获取数据的方法，内部类负责提供修改数据的方法。

7.2.2 File

Android中也定义了一些方法获取Java的标准文件输入输出流。原理上和PC上的File类原理相似，只是使用Android中定义方法来操作。  
FileInputStream Context.openFileInput()  
InputStream Resources.openRawResource()

7.2.3 SQLiteDatabase

SQLite是一个嵌入式数据库引擎，针对资源有限的移动设备而言非常高效。可以在android代码中使用SQLiteDatabase（即数据库对象类）中的方法对其直接进行数据库相关操作。显然，该数据库是没有独立进程的。

SQLiteOpenHelper是一个该类型数据库的帮助类，是一个部分抽象类。只需实现onCreate和onUpgrade等事件回调方法，其他的有关数据库创建与管理的方法已经被定义。

## 7.3 res资源文件的使用

7.3.1 strings.xml

1) Resource strings and placeholders

Imagine you want to have in your strings.xml the copy for a precomposed tweet. In the end you want something like this:

“Groovy Calamari (http://groovycalamari.com) by @sdelamo is the best newsletter in the world.”

However, you don’t want the newsletter name, url and author twitter username to be hardcoded.

Fortunately we can use placeholders. Move the values to constants:  
static final String NEWSLETTER\_NAME = "Groovy Calamari";  
static final String NEWSLETTER\_URL = "http://groovycalamari.com";  
static final String NEWSLETTER\_AUTHOR = "@sdelamo";

Add to *strings.xml* the text:  
%1$s (%2$s) by %3$s is the best newsletter in the world

Then you can just call it with:  
String message = getApplicationContext().getString(R.string.share\_text, NEWSLETTER\_NAME, NEWSLETTER\_URL, NEWSLETTER\_AUTHOR);

Placeholder即占位符，其格式和字符串格式化输出string format的规律相似。

7.3.2 attr与style.xml以及?和@的符号使用

1）@和？的区别

style="?android:attr/progressBarStyleHorizontal"

style="@android:style/Widget.ProgressBar.Horizontal"

在设置style的时候既可以使用@也可以使用？，他们有什么区别呢？？

使用@表示使用固定的style，而不会跟随Theme改变，这style可以在对应的style.xml中找到。使用？表示从Theme中查找引用的资源名，这个google叫预定义样式，用在多主题时的场景，属性值会随着主题而改变。（？需要和attr配合使用）

例如上面的progressBarStyleHorizontal，查看\platforms\android-23\data\res\values\themes.xml文件，可以看到在不同的theme中，progressBarStyleHorizontal引用的style是不同的。看下面的示例：

在Theme中

<item name="progressBarStyleHorizontal">@style/Widget.ProgressBar.Horizontal</item>

Theme.Holo中为

<item name="progressBarStyleHorizontal">@style/Widget.Holo.ProgressBar.Horizontal</item>

2）style和attr使用区别

style

<!--使用自定义的style-->

@style/Widget.AppCompat.ProgressBar.Horizontal

<!--使用系统自带的style-->

@android:style/Widget.ProgressBar.Horizontal

attr

<!--使用自定义的 ，下面两种方式等效-->

"?attr/属性"

"?属性"

<!--使用系统自带的 ，下面两种方式等效-->

"?android:属性"

"?android:attr/属性"

另外：当引用系统自带的style和attr时

"@android:style/主题"和"@style/android:主题"等同

"?android:attr/属性"和"?attr/android:属性"等同

3）自定义随主题改变的属性

①如果是自定义控件，请在style.xml中或attrs.xml中声明属性：

<declare-styleable name="SunnyAttr">

<attr name="sunnyTextColor" format="reference"/>

<attr name="sunnyBgColor" format="reference"/>

<attr name="sunnyTextColorWhite" format="color"/>

<attr name="sunnyTextColorRed" format="reference"/>

<attr name="textColor" format="reference"></attr>

</declare-styleable>

如代码第五行所示，必须指明format为reference。这样自定义控件的属性就可以在xml使用，如果不明白，[查看这里](http://blog.csdn.net/xx326664162/article/details/63254622)

②在Theme中使用自定义的属性，可以再多个主题中定义不同的属性值

<!-- Base application theme. -->

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">

<!-- Customize your theme here. -->

<item name="colorPrimary">@color/colorPrimary</item>

<item name="colorPrimaryDark">@color/colorPrimaryDark</item>

<item name="colorAccent">@color/colorAccent</item>

<item name="sunnyTextColorRed">@color/sunnyTextColorRed</item>

</style>

③在对应的属性color,drawable等里面加入相应的资源

<color name="sunnyTextColorRed">#FFFF0000</color>

④这样就可以在xml中使用自定义控件的自定义属性，这个属性会随着主题而改变：

见下面第五行代码

<com.smartbracelet.sunny.sunnydemo3.SunnyTextView

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_centerInParent="true"

android:text="设置界面"

app:sunnyTextColor="?attr/sunnyTextColorRed"

/>

7.3.3 Layout文件xml

1）LayoutInflater之inflate方法介绍

inflate()是一个以layout资源文件为参数，创建视图对象并返回view类型的函数。

inflate方法从大范围来看，分两种，三个参数的构造方法和两个参数的构造方法。在这两类中又有细分，OK，那我们就把各种情况都来演示一遍。

三个参数的inflate方法方法头如下：

**public** View inflate(@LayoutRes **int** resource, @Nullable ViewGroup root, **boolean** attachToRoot)    
好，这里主要分为三种情况，分别来看

①root不为null，attachToRoot为true

当root不为null，attachToRoot为true时，表示将resource指定的布局添加到root中，添加的过程中resource所指定的的布局的根节点的各个属性都是有效的。比如下面一个案例，我的Activity的布局如下：

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    android:layout\_width="match\_parent"

    android:layout\_height="match\_parent"

    android:orientation="vertical"

    android:id="@+id/ll"

    tools:context="org.sang.layoutinflater.MainActivity">

</LinearLayout>

我还有一个布局linearlayout.xml如下：

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:id="@+id/ll"

    android:layout\_width="200dp"

    android:layout\_height="200dp"

    android:background="@color/colorPrimary"

    android:gravity="center"

    android:orientation="vertical">

    <Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content" />

</LinearLayout>

我现在想把这个linearlayout.xml布局文件添加到我的activity的布局中，那么我可以这么做：

@Override

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

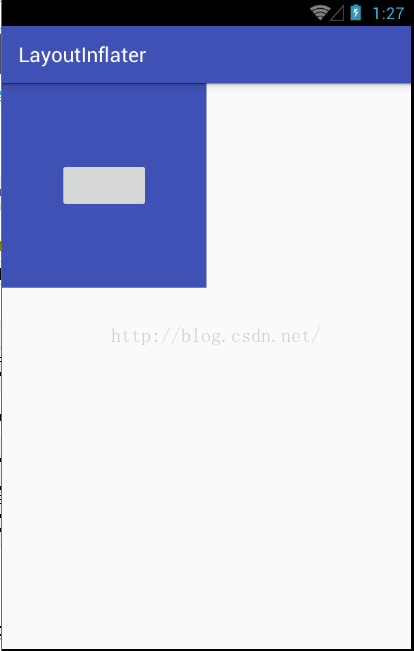
    setContentView(R.layout.activity\_main);

    LinearLayout ll = (LinearLayout) findViewById(R.id.ll);

    LayoutInflater inflater = LayoutInflater.from(**this**);

    inflater.inflate(R.layout.linearlayout, ll,**true**);  }

小伙伴们注意到，这里我都没写将inflate出来的View添加到ll中的代码，但是linearlayout布局文件就已经添加进来了，这就是因为我第三个参数设置为了true，表示将第一个参数所指定的布局添加到第二个参数的View中。最终显示效果如下：



如果我作死多写这么一行代码，如下：

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

    setContentView(R.layout.activity\_main);

    LinearLayout ll = (LinearLayout) findViewById(R.id.ll);

    LayoutInflater inflater = LayoutInflater.from(**this**);

    View view = inflater.inflate(R.layout.linearlayout, ll, **true**);

ll.addView(view);  }

这个时候再运行，系统会抛如下异常：

java.lang.IllegalStateException: The specified child already has a parent. You must call removeView() on the child's parent first.

原因就是因为当第三个参数为true时，会自动将第一个参数所指定的View添加到第二个参数所指定的View中。

②root不为null，attachToRoot为false

如果root不为null，而attachToRoot为false的话，表示不将第一个参数所指定的View添加到root中，那么这个时候有的小伙伴可能就有疑问了，既然不添加到root中，那我还写这么多干嘛？我第二个参数直接给null不就可以了？其实不然，这里涉及到另外一个问题：我们在开发的过程中给控件所指定的layout\_width和layout\_height到底是什么意思？该属性的表示一个控件在容器中的大小，就是说这个控件必须在容器中，这个属性才有意义，否则无意义。这就意味着如果我直接将linearlayout加载进来而不给它指定一个父布局，则inflate布局的根节点的layout\_width和layout\_height属性将会失效（因为这个时候linearlayout将不处于任何容器中，那么它的根节点的宽高自然会失效）。如果我想让linearlayout的根节点有效，又不想让其处于某一个容器中，那我就可以设置root不为null，而attachToRoot为false。这样，指定root的目的也就很明确了，即root会协助linearlayout的根节点生成布局参数，只有这一个作用。OK，还是上面的布局文件，如果我想将之添加到activity的布局中又该如何呢？

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

    setContentView(R.layout.activity\_main);

    LinearLayout ll = (LinearLayout) findViewById(R.id.ll);

    LayoutInflater inflater = LayoutInflater.from(**this**);

    View view = inflater.inflate(R.layout.linearlayout, ll, **false**);

    ll.addView(view);  }

大家注意，这个时候我需要手动的将inflate加载进来的view添加到ll容器中，因为inflate的最后一个参数false表示不将linealayout添加到ll中。显示效果和上文一样，不再贴图。

③root为null

当root为null时，不论attachToRoot为true还是为false，显示效果都是一样的。当root为null表示我不需要将第一个参数所指定的布局添加到任何容器中，同时也表示没有任何容器来来协助第一个参数所指定布局的根节点生成布局参数。我还是使用上文提到的linearlayout，我们来看下面一段代码：

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_main);

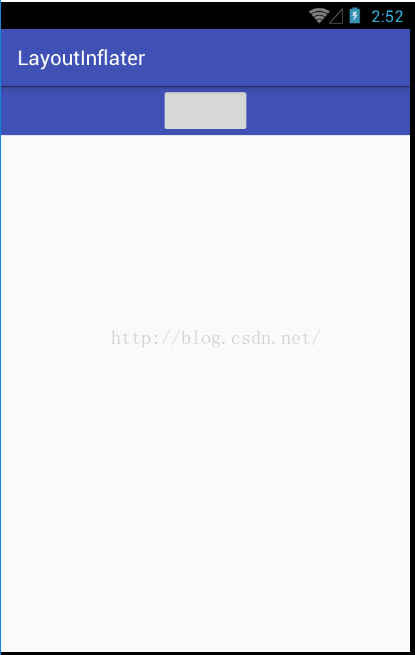
        LinearLayout ll = (LinearLayout) findViewById(R.id.ll);

        LayoutInflater inflater = LayoutInflater.from(**this**);

        View view = inflater.inflate(R.layout.linearlayout, **null**, **false**);

        ll.addView(view);  }

当第二个参数为null，第三个参数为false时（即使为true显示效果也是一样的，这里以false为例），由于在inflate方法中没有将linearlayout添加到某一个容器中，所以我需要手动添加，另外由于linearlayout并没有处于某一个容器中，所以它的根节点的宽高属性会失效，显示效果如下：



小伙伴们注意，这个时候不管我给linearlayout的根节点的宽高设置什么，都是没有效果的，它都是包裹button，如果我修改button，则button会立即有变化，因为button是处于某一个容器中的。

两个参数的inflate方法就很简单了，我们来稍微看一点点源码：

**public** View inflate(XmlPullParser parser, @Nullable ViewGroup root) {

**return** inflate(parser, root, root != **null**);

}

这是两个参数的inflate方法，大家注意两个参数实际上最终也是调用了三个参数。

两个参数的inflate方法分为如下两种情况：

a. root为null，等同于1.3所述情况。

b. root不为null，等同于1.1所述情况。

c. 为什么Activity布局的根节点的宽高属性会生效？

inflate方法我们已经说完了，小伙伴们可能有另外一个疑问，那为什么Activity布局的根节点的宽高属性会生效？其实原因很简单，大部分情况下我们一个Activity页面由两部分组成（Android的版本号和应用主题会影响到Activity页面组成，这里以常见页面为例），我们的页面中有一个顶级View叫做DecorView，DecorView中包含一个竖直方向的LinearLayout，LinearLayout由两部分组成，第一部分是标题栏，第二部分是内容栏，内容栏是一个FrameLayout，我们在Activity中调用setContentView就是将View添加到这个FrameLayout中，所以给大家一种错觉仿佛Activity的根布局很特殊，其实不然。

# 8.Gradle详解

## 8.1 Gradle简介

Gradle是一种项目管理和构建框架，最最简单的构建工具就是make了。make就是根据Makefile文件中写的规则，执行对应的命令，然后得到目标产物。

在Gradle爆红之前，常用的构建工具是ANT，然后又进化到Maven。ANT和Maven这两个工具其实也还算方便，现在还有很多地方在使用。但是二者都有一些缺点，所以让更懒得人觉得不是那么方便。比如，Maven编译规则是用XML来编写的。XML虽然通俗易懂，但是很难在xml中描述if{某条件成立，编译某文件}/else{编译其他文件}这样有不同条件的任务。

Gradle支持编程解决前任项目构建框架无法解决的问题。Gradle选择了Groovy。Groovy基于Java并拓展了Java。 Java程序员可以无缝切换到使用Groovy开发程序。Groovy说白了就是把写Java程序变得像写脚本一样简单。写完就可以执行，Groovy内部会将其编译成Java class然后启动虚拟机来执行。

Gradle另外一个特点就是它是一种DSL，即Domain Specific Language，领域相关语言。

## 8.2 Groovy介绍

8.2.1 Groovy编程语言特点概述

**1）一般**

Groovy注释标记和Java一样，支持//或者/\*\*/

Groovy语句可以不用分号结尾。

Groovy可以像Java一样定义package并沿用类似的编程结构，也可以直接在一个文件中书写操作代码而无需置于某个类或主方法中，即脚本式使用。

**2）变量定义**

Groovy中支持动态类型，即定义变量的时候可以不指定其类型。Groovy中，变量定义可以使用关键字def。注意，虽然def不是必须的，但是为了代码清晰，建议还是使用def关键字。def关键字起到的作用不仅这些，详见后文Groovy脚本理解。

**3）函数**

函数定义时，参数的类型也可以不指定。函数的返回值也可以是无类型的。无类型的函数定义，必须使用def关键字。如果指定了函数返回类型，则可不必加def关键字来定义函数。

函数返回值：Groovy的函数里，可以不使用return xxx来设置xxx为函数返回值。如果不使用return语句的话，则函数里最后一句代码的执行结果被设置成返回值。注意，如果函数定义时候指明了返回值类型的话，函数中则必须返回正确的数据类型，否则运行时报错。如果使用了动态类型的话，你就可以返回任何类型了。

函数调用的时候还可以省略参数外的括号，以及参数之间的逗号。注意，虽然写代码的时候，对于函数调用可以不带括号，但是Groovy经常把属性和函数调用混淆。调用函数要不要带括号，我个人意见是如果这个函数是Groovy API或者Gradle API中比较常用的，比如println，就可以不带括号。否则还是带括号。

**4）字符串**

Groovy对字符串支持相当强大，充分吸收了一些脚本语言的优点：

1  单引号''中的内容严格对应Java中的String，不对$符号进行转义  
def singleQuote='I am $ dolloar' //输出就是I am $ dolloar

2  双引号""的内容则和脚本语言的处理有点像，如果字符中有$号的话，则它会$表达式先求值。  
def doubleQuoteWithoutDollar = "I am one dollar" //输出 I am one dollar  
def x = 1  
def doubleQuoteWithDollar = "I am $x dolloar" //输出I am 1 dolloar

3 三个引号'''xxx'''中的字符串支持随意换行 比如  
def multieLines = ''' begin  
line 1   
line 2  
end '''

**5）类**

如果不声明public/private等访问权限的话，Groovy中类及其变量默认都是public的。

8.2.2 Groovy中的数据类型

**1）基本数据类型**

作为动态语言，Groovy世界中的所有事物都是对象。所以，int，boolean这些Java中的基本数据类型，在Groovy代码中其实对应的是它们的包装数据类型。比如int对应为Integer，boolean对应为Boolean。

**2）容器类**

①List类

变量定义：List变量由[]定义，比如

def aList = [5,'string',true] //List由[]定义，其元素可以是任何对象

变量存取：可以直接通过索引像一般数组存取，而且不用担心索引越界。如果索引超过当前链表长度，List会自动延长列表至该索引并往该索引添加元素null。

注意：形如[5,'string',true]的实体在Groovy中已经视为一个List类型的常量对象，可以如一般对象一样调用成员函数。

②Map类

变量定义：Map变量由[:]定义，比如

def aMap = ['key1':'value1','key2':true]

Map由[:]定义，注意其中的冒号。冒号左边是key，右边是Value。key必须是字符串，value可以是任何对象。另外，key与value可以用''或""包起来，也可以不用引号包起来。不过Key要是不使用引号包起来的话，也会带来一定混淆。

变量存取：  
println aMap.keyName <==这种表达方法好像key就是aMap的一个成员变量一样  
println aMap['keyName'] <==这种表达方法更传统一点  
aMap.anotherkey = "i am map" <==为map添加新元素

③Range类

Range是Groovy对List的一种拓展，变量定义和大体的使用方法如下：

def aRange = 1..5 <==Range类型的变量 由begin值+两个点+end值表示左边这个aRange包含1,2,3,4,5这5个值

如果不想包含最后一个元素，则

def aRangeWithoutEnd = 1..<5 <==包含1,2,3,4这4个元素

println aRange.from

println aRange.to

**3）闭包**

闭包，英文叫Closure，是Groovy中非常重要的一个数据类型或者说一种概念了。它是一种数据类型，它代表了一段可执行的代码。其外形如下：

def aClosure = {//闭包是一段代码，所以需要用花括号括起来..   
Stringparam1, int param2 -> //这个箭头很关键。前面是参数定义，箭头后面是代码   
println"this is code" //这是代码，最后一句是返回值，   
//也可以使用return，和Groovy中普通函数一样  
}

简而言之，Closure的定义格式是：

def xxx = {paramters -> code} //或者

def xxx = {无参数，纯code} 这种case不需要->符号

说实话，从C/C++语言的角度看，闭包和函数指针很像。定义好后，要调用它的方法就是：  
闭包对象.call(参数)  或者更像函数指针调用的方法：  
闭包对象(参数)    
比如：  
aClosure.call("this is string",100) 或者   
aClosure("this is string", 100)

上面就是一个闭包的定义和使用。在闭包中，还需要注意一点：

如果闭包没定义参数的话，则隐含有一个参数，这个参数名字叫it。it代表闭包的参数。

比如：

def greeting = { "Hello, $it!" }

assert greeting('Patrick') == 'Hello, Patrick!'

等同于：

def greeting = { it -> "Hello, $it!" }

assert greeting('Patrick') == 'Hello, Patrick!'

但是，如果在闭包定义时，采用下面这种写法，则表示闭包没有参数！

def noParamClosure = { -> true }

这个时候，我们就不能给noParamClosure传参数了！

闭包在Groovy中大量使用，比如很多类都定义了一些函数，这些函数最后一个参数都是一个闭包，大体的功能都是将闭包作为一个函数指针回调。闭包的具体内容可由用户自定义，然而闭包的参数与返回值显然都受该函数的上下文限定的，需要用户参考API文档来确定。

8.2.3 Groovy脚本理解

**1）Groovy脚本的实现**

Java中，我们最熟悉的是类。但是我们在Java的一个源码文件中，不能不写class（interface或者其他....），而Groovy可以像写脚本一样，把要做的事情都写在xxx.groovy中，而且可以通过groovy xxx.groovy直接执行这个脚本。

Groovy会先把xxx.groovy中的内容转换成一个Java类。执行 groovyc -d classes test.groovy

groovyc是groovy的编译命令，-d classes用于将编译得到的class文件拷贝到classes文件夹下。

xxx.groovy被转换成了一个xxx类，它从script派生。每一个脚本都会生成一个static main函数。这样，当我们groovy xxx.groovy的时候，其实就是用java去执行这个main函数。脚本中的所有代码都会放到xxx类的run函数中。如果脚本中定义了函数，则函数会被定义在xxx类中。

groovyc是一个比较好的命令，读者要掌握它的用法。然后利用jd-gui来查看对应class的Java源码。

**2）Groovy脚本的变量作用域**

因为Groovy的上述特性，脚本中的变量和函数在实际执行时可能并不在相同作用域。使用不同的声明变量的方法可以决定脚本内容生成的Java类最终的形态。

①使用类型或def声明变量

def x = 1或 int x = 1

变量是run()方法中的局部变量，不能被脚本中的函数调用，因为这些函数和run均为平等的成员函数。

②不使用def和类型，直接声明变量

x = 1

在run的执行过程中，将x作为一个属性添加到xxx类的实例对象中了。然后在脚本函数执行时，先获取这个属性。x并不是xxx类的成员，因此其他脚本无法直接访问x。

③脚本中使用Field注释

import groovy.transform.Field; //必须要先import

@Field x = 1 <==在x前面加上@Field标注，这样，x就彻彻底底是test的成员变量了。

**3）Groovy的文件I/O操作**

整体说来，Groovy的I/O操作是在原有Java I/O操作上进行了更为简单方便的封装，并且使用Closure来简化代码编写。

def targetFile = new File(文件名)  <==File对象还是要创建的。

1. 读文件

·读该文件中的每一行：eachLine的唯一参数是Closure。Closure的参数是文件每行的内容

targetFile.eachLine{

String oneLine ->

printlnoneLine }

·直接得到文件内容：

targetFile.getBytes() <==文件内容一次性读出，返回类型为byte[]

·使用InputStream：

def ism = targetFile.newInputStream()

//操作ism，最后记得关掉

ism.close

·使用闭包操作inputStream：

targetFile.withInputStream{ ism ->

操作ism. 不用close。Groovy会自动替你close

}

1. 写文件

和读文件差不多。这里给个例子，告诉大家如何copy文件。

def srcFile = new File(源文件名)

def targetFile = new File(目标文件名)

targetFile.withOutputStream{ os->

srcFile.withInputStream{ ins->

os << ins //用OutputStream的<<操作符重载完成从inputstream到OutputStream的输出

}

}

Groovy I/O操作相关类的SDK地址：

java.io.File: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/File.html**

java.io.InputStream: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/InputStream.html**

java.io.OutputStream: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/OutputStream.html**

java.io.Reader: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/Reader.html**

java.io.Writer: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/Writer.html**

java.nio.file.Path: **http://docs.groovy-lang.org/latest/html/groovy-jdk/java/nio/file/Path.html**

**4）Groovy的XML文件操作**

Groovy中，XML的解析提供了和XPath类似的方法，名为GPath。这是一个类，提供相应API。

举例说明：

test.xml文件：

<response version-api="2.0">

<value>

<books>

<book available="20" id="1">

<title>Don Xijote</title>

<author id="1">Manuel De Cervantes</author>

</book>

<book available="14" id="2">

<title>Catcher in the Rye</title>

<author id="2">JD Salinger</author>

</book>

<book available="13" id="3">

<title>Alice in Wonderland</title>

<author id="3">Lewis Carroll</author>

</book>

<book available="5" id="4">

<title>Don Xijote</title>

<author id="4">Manuel De Cervantes</author>

</book>

</books>

</value>

</response>

Groovy脚本：

//第一步，创建XmlSlurper类

def xparser = new XmlSlurper()

def targetFile = new File("test.xml")

//GPath出场

GPathResult gpathResult =xparser.parse(targetFile)

//现在我要访问id=4的book元素。

//gpathResult代表根元素response。通过e1.e2.e3这种格式就能访问到各级子元素

def book4 = gpathResult.value.books.book[3]

//得到book4的author元素

def author = book4.author

//再来获取元素的属性和textvalue

assert author.text() == ' Manuel De Cervantes '

//获取属性更直观

author.@id == '4' 或者 author['@id'] == '4'

//属性一般是字符串，可通过toInteger转换成整数

author.@id.toInteger() == 4

## 8.3 Gradle组件

Gradle中，每一个待编译的工程都叫一个Project。每一个Project在构建的时候都包含一系列的Task。比如一个Android APK的编译可能包含：Java源码编译Task、资源编译Task、JNI编译Task、lint检查Task、打包生成APK的Task、签名Task等。一个Project包含多少个Task由编译脚本指定的插件决定。插件就是用来定义Task，并具体执行这些Task的东西。

每一个Library和每一个App都是单独的Project。每一个Project在其根目录下都需要有一个build.gradle。build.gradle文件就是该Project的编译脚本，类似于Makefile。

Multi-Projects Build即多项目共同构建，需要在工程根目录也添加一个build.gradle，是为root project。这个build.gradle一般用于配置其他子Project。此build.gradle有没有都可以。另外根目录需要一个名为settings.gradle。这个文件名字必须是settings.gradle用来告诉Gradle，这个multiprojects包含多少个子Project。

Task和Task之间往往是有关系的，这就是所谓的依赖关系。某Task必须先等待其依赖的Task完成，才能执行该Task。

## 8.4 Gradle命令介绍

**1.gradle projects查看工程信息**

执行gradle projects命令，用于查看当前目录下工程信息

**2.gradle tasks查看任务信息**

执行gradle [project-path]:tasks来查看项目的task信息，若为项目自身目录，可省略路径

**3.gradle task-name执行任务**

执行gradle task-name从tasks信息中选取对应名称的任务来执行

## 8.5 Gradle工作流程

Gradle工作包含三个阶段：

①首先是初始化阶段。对multi-project build而言，就是执行settings.gradle

1. Initiliazation phase的下一个阶段是Configration阶段。

③Configration阶段的目标是解析每个project中的build.gradle。在这两个阶段之间，我们可以通过API加一些定制化的Hook，即一些希望在该阶段执行的函数。

④Configuration阶段完了后，整个project以及内部的Task关系就确定了。Configuration会建立一个有向图来描述Task之间的依赖关系。我们也可以添加一个HOOK，当Task关系图建立好后，执行一些操作。

⑤最后一个阶段就是执行任务了。当然，任务执行完后，我们还可以加Hook。

## 8.6 Gradle编程模型及API讲解

Gradle执行时会将不同的脚本转换为不同的Java对象，共有三种对象：  
Gradle对象：当我们执行gradle xxx或者什么的时候，gradle会从默认的配置脚本中构造出一个Gradle对象。在整个执行过程中，只有这么一个对象。Gradle对象的数据类型就是Gradle。我们一般很少去定制这个默认的配置脚本。  
Project对象：每一个build.gradle会转换成一个Project对象。  
Settings对象：显然，每一个settings.gradle都会转换成一个Settings对象。

对于其他gradle文件，除非定义了class，否则会转换成一个实现了Script接口的对象，类似一般的Groovy脚本。

8.6.1 Gradle对象

详见文档即可https://docs.gradle.org/current/dsl/

8.6.2 Project对象

在Gradle术语中，Project对象对应的是Build Script。

①加载插件

加载插件是调用它的apply函数。apply其实是Project实现的PluginAware接口定义的

常用的apply函数原型为apply(Map<String,?> options)。

脚本中一般为：

apply plugin: 'com.android.library' <==如果是编译Library，则加载此插件

apply plugin: 'com.android.application' <==如果是编译Android APP，则加载此插件

apply from: rootProject.getRootDir().getAbsolutePath() + "/utils.gradle" 加载根目录下的脚本

传入的参数实际上是一个Map，但是它的Key限定于以下三种：

from：脚本类型，接收路径名

plugin：插件类型，插件的实现类或id名称

to：对象类型

②设置属性

Gradle提供了一种名为extra property的方法。extra property是额外属性的意思，在第一次定义该属性的时候需要通过ext前缀来标示它是一个额外的属性。定义好之后，后面的存取就不需要ext前缀了。ext属性支持Project和Gradle对象。

除了ext.xxx=value这种定义方法外，还可以使用ext{}这种书写方法。ext{}不是ext(Closure)对应的函数调用。但是ext{}中的{}确实是闭包。

③Task介绍

Task是Gradle中的一种数据类型，它代表了一些要执行或者要干的工作。不同的插件可以添加不同的Task。每一个Task都需要和一个Project关联。

//Task是和Project关联的，所以，我们要利用Project的task函数来创建一个Task

task myTask <==myTask是新建Task的名字

task myTask { configure closure }

task myType << { task action } <==注意，<<符号是doLast的缩写

task myTask(type: SomeType)

task myTask(type: SomeType) { configure closure }

④Script Block

SB实际上就是一个带闭包参数的函数，这些函数常用于配置project

常用的SB有subprojects，buildscript，repositories，dependencies等。

# 碎片知识积累

## DPI、DIP、PX、DP、SP的区别

px ：其实就是像素单位，比如我们通常说的手机分辨列表800\*400都是px的单位

sp ：同dp相似，还会根据用户的字体大小偏好来缩放

dp ：虚拟像素，在不同的像素密度的设备上会自动适配

dip：同dp，即device independent pixels，设备独立像素单位

dpi：全称是dots per inch，对角线每英寸的像素点的个数，也称像素密度

用法：使用dp作为单位描述组件的几何尺寸，可以使得组件在不同像素密度的机器上自动匹配合适的大小（保证视觉比例不变）。  
使用px作为单位，绘制单像素分割线，否则其他单位会使分割线模糊。  
使用sp单位描述字体的尺寸，可以使得该APP的字体和系统全局字体匹配并协调变化。若使用dp也能实现跨机器自适应的效果，但不具备系统全局协调性。

## Log图文详解

android.util.Log常用的方法有以下5个：Log.v() Log.d() Log.i() Log.w() 以及 Log.e() 。根据首字母对应VERBOSE，DEBUG,INFO, WARN，ERROR。Log信息在LogCat视窗中查看。

1、Log.v 的调试颜色为黑色的，任何消息都会输出，这里的v代表verbose啰嗦的意思，平时使用就是Log.v("","");两个参数均为字符串类型，第一个是tag标签，第二个是message信息。其他的Log函数都是一样的原型。

2、Log.d的输出颜色是蓝色的，仅输出debug调试的意思，但他会输出上层的信息，过滤起来可以通过DDMS的Logcat标签来选择.

3、Log.i的输出为绿色，一般提示性的消息information，它不会输出Log.v和Log.d的信息，但会显示i、w和e的信息

4、Log.w的意思为橙色，可以看作为warning警告，一般需要我们注意优化Android代码，同时选择它后还会输出Log.e的信息。

5、Log.e为红色，可以想到error错误，这里仅显示红色的错误信息，这些错误就需要我们认真的分析，查看栈的信息了。

Log函数可以由我们自己在适当地方调用，用以方便维护系统。在Android编译器内部也会使用这些函数给我们发送信息。

## 相关术语的解析

Dalvik： Android特有的虚拟机,和JVM不同,Dalvik虚拟机非常适合在移动终端上使用!

AVD：Android Virtual Machine安卓虚拟设备,就是安卓的模拟器

ADT：Android Development Tools安卓开发工具

ADB：Android Debug Bridge

SDK：(software development kit)软件开发工具包,就是安卓系统,平台架构等的工具集合,如adb.exe

DDMS：(dalvik debug monitor service)安卓调试工具

adb：安卓调试桥,在sdk的platform-tools目录下,功能很多,命令行必备

DX工具：将.class转换成.dex文件

AAPT：(android asset packing tool),安卓资源打包工具

R.java文件：由aapt工具根据App中的资源文件自动生成,可以理解为资源字典

AndroidManifest.xml：app包名 + 组件声明 + 程序兼容的最低版本 + 所需权限等程序的配置文件

AS自动构建的项目基础像一个框架，框架即使用XML或Java注释来配置一些框架本身提供的辅助开发和编译的工具。否则所有的程序都必须也只能从单纯的源代码写出全部的部分。可以说框架有的时候是开发工具库，有时候是可重用模块。

界面设计

AS的界面布局依靠layouts（ViewGroup对象）和widgets（View对象）管理。前者就是界面布局模式，后者是UI组件（按钮文本框等）。ViewGroup对象管理View对象和其他子ViewGroup对象（虽然子ViewGroup实现的设计完全可以被单纯的View替代）。

AS的界面设计Design和Blueprint使用方法见官方Trainning网页。