

# Android Interview Preparation – Coding

deepwaterooo

2021 年 12 月 16 日

## 目录

<b>1 全面了解 Activity</b>	<b>1</b>
1.1 Activity 是什么?	1
1.2 Activity 的内部调用过程	1
1.2.1 activity 被回收的状态和信息保存和恢复过程	2
1.3 启动模式	3
1.3.1 启动模式什么?	3
1.3.2 为什么要定义启动模式?	3
1.3.3 怎样定义启动模式?	3
1.4 Intent Filter	5
1.4.1 intent-filter 的三大属性	5
1.4.2 Activity 种 Intent Filter 的匹配过程	5
1.5 开发中 Activity 的一些问题	5
<b>2 Fragment 全解析</b>	<b>6</b>
2.1 概述	6
2.2 Fragment 的生命周期	6
2.3 将 fragment 添加到 activity 之中	8
2.3.1 在 activity 的布局文件里声明 fragment	8
2.3.2 通过编码将 fragment 添加到已存在的 ViewGroup 中	9
2.3.3 添加没有界面的 fragment	10
2.4 fragment 事务后台栈	10
2.5 与 Activity 交互	10
<b>3 Service 全面总结</b>	<b>11</b>
3.1 什么是服务?	11
3.2 Service 与 Thread 的区别	11
3.3 服务的分类	11
3.3.1 按运行分类	11
3.3.2 按使用分类	12
3.4 Service 生命周期	12
3.5 在 manifest 中声明服务	14
3.6 startService 启动服务	15
3.7 bindService 启动服务	15
<b>4 IntentService 使用详解和实例但介绍</b>	<b>17</b>
4.1 IntentService 定义	17
4.2 IntentService 与 Service 的区别	18
4.3 IntentService 实例介绍	18

<b>5 Android 启动过程图解</b>	<b>19</b>
5.0.1	20
5.0.2	20
5.0.3	20
5.0.4	20
5.0.5	21
5.0.6	21
<b>6 Android 异步消息处理机制（Handler、Looper、MessageQueue）源码解析</b>	<b>21</b>
6.0.1	21
6.0.2	21
6.0.3	22
6.0.4	22
6.0.5	23
6.0.6	24
6.0.7	25
<b>7 Android 数据存储五种方式使用与总结</b>	<b>25</b>
7.0.1	25
7.0.2	25
7.0.3	26
7.0.4 3. 2 使用外部存储（sdcard）	27
7.0.5	28
7.0.6 HttpURLConnection	28
7.0.7 HttpClient	29
7.0.8 Android 提供的其他网络访问框架	30
7.0.9	30
7.0.10	30
<b>8 Android 缓存机制</b>	<b>30</b>
8.1 实现原理	30
8.2 Bitmap 的缓存	30
8.2.1 使用内存缓存	30
8.2.2 使用磁盘缓存	31
8.2.3 设备配置参数改变时加载问题	32
8.3 使用 SQLite 进行缓存	32
8.4 文件缓存	33
<b>9 Android 异步任务机制之 AsyncTask</b>	<b>33</b>
9.0.1	33
9.0.2	33
9.0.3	33
9.0.4	34
9.0.5	35
<b>10Android 自定义 View 入门</b>	<b>35</b>
10.1一、实现步骤	35
10.2二、哪些方法需要被重写	35
10.3三. 自定义控件的三种方式	36
10.4四. 自定义属性的两种方法	36
10.5五. 自定义随手指移动的小球 (小例子)	37

<b>11Android 自定义 ViewGroup 入门实践</b>	<b>38</b>
11.11.View 绘制流程	38
11.2 2.onMeasure 过程	39
11.3 3.onLayout 过程	39
11.44. 示例程序	40
<b>12Android View 事件分发机制源码分析</b>	<b>41</b>
12.11. 涉及到事件响应的常用方法构成	42
12.2 2.android 事件处理涉及到的三个重要函数	42
12.3 3.View 源码分析	43
12.4 4.ViewGroup 源码分析	44
12.5 5. 总结	45
<b>13BroadcastReceiver 使用总结</b>	<b>46</b>
13.1BroadcastReceiver 的定义	46
13.2BroadcastReceiver 使用注意	46
13.3BroadcastReceiver 的注册	46
13.3.1静态方式	46
13.3.2动态方式	47
13.3.3静态注册和动态注册的区别	47
13.4发送 BroadcastReceiver	47
13.4.1普通广播	47
13.4.2有序广播	47
13.4.3异步广播	47
13.4.4异步有序广播	48
13.5总结	48
13.6一些常用的系统广播的 action 和 permission	48
<b>14ContentProvider 实例详解</b>	<b>49</b>
14.11.ContentProvider 是什么?	49
14.2 2.URL	49
14.2.12. 1 MIME	50
14.2.22. 2 ContentUriis	50
14.2.32. 3 UriMatcher	50
14.33.ContentProvider 的主要方法	51
14.4 4.ContentResolver	51
14.55.ContentObserver	52
14.66. 实例说明	52
<b>15Android SQLite 的使用入门</b>	<b>54</b>
15.1概述	54
15.2SQLite 的特性	54
15.3Android 中使用 SQLite	54
15.3.1创建数据库	55
15.3.2创建表和索引	56
15.3.3添加数据	56
15.3.4更新数据 (修改)	56
15.3.5删除数据	56
15.3.6查询数据	57
<b>16Android 名企面试题及答案整理 (一)</b>	<b>57</b>

17AIDL 的使用情况和实例介绍62

17.1AIDL 是什么?62

17.2什么情况下要使用 AIDL62

17.3定义一个 AIDL 接口的步骤62

17.4用一个实例来分步骤说明63

17.4.1在 server 项目中建立.aidl 文件63

17.4.2在 server 项目中建立服务类63

17.4.3在 server 项目 AndroidManifest 中申明 Service64

17.4.4把 server 项目中的 aidl 文件带包拷贝到 client 项目中（包名要相同）64

# 1 全面了解 Activity

## 1.1 Activity 是什么？

- 我们都知道 android 中有四大组件（Activity 活动，Service 服务，Content Provider 内容提供者，BroadcastReceiver 广播接收器），Activity 是我们用的最多也是最基本的组件，因为应用的所有操作都与用户相关，Activity 提供窗口来和用户进行交互。

官方文档这么说：

- An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View).
- 大概的意思（原谅我）：activity 是独立平等的，用来处理用户操作。几乎所有的 activity 都是用来和用户交互的，所以 activity 类会创建了一个窗口，开发者可以通过 setContentView(View) 的接口把 UI 放到给窗口上。
- Android 中的 activity 全都归属于 task 管理。task 是多个 activity 的集合，这些 activity 按照启动顺序排队存入一个栈（即“back stack”）。android 默认会为每个 App 维持一个 task 来存放该 app 的所有 activity，task 的默认 name 为该 app 的 package name。
- 当然我们也可以在 AndroidMainfest.xml 中申明 activity 的 taskAffinity 属性来自定义 task，但不建议使用，如果其他 app 也申明相同的 task，它就有可能启动到你的 activity，带来各种安全问题（比如拿到你的 Intent）。

## 1.2 Activity 的内部调用过程

- 上面已经说了，系统通过堆栈来管理 activity，当一个新的 activity 开始时，它被放置在堆栈的顶部和成为运行活动，以前的 activity 始终保持低于它在堆栈，而不会再次到达前台，直到新的活动退出。
- 还是上这张官网的 activity\_lifecycle 图：
- ![这里写图片描述](http://img.blog.csdn.net/20160425171711054)
- 首先打开一个新的 activity 实例的时候，系统会依次调用
  - > onCreate() -> onStart() -> onResume() 然后开始 running
- running 的时候被覆盖了（从它打开了新的 activity 或是被锁屏，但是它 \*\* 依然在前台 \*\* 运行，lost focus but is still visible），系统调用 onPause();

- 该方法执行 **activity** 暂停，通常用于提交未保存的更改到持久化数据，停止动画和其他的东西。但这个 **activity** 还是完全活着（它保持所有的状态和成员信息，并保持连接到 **\*\*窗口管理器\*\***）
- 接下来它有三条出路
  - 用户返回到该 **activity** 就调用 **onResume()** 方法重新 **running**
  - 用户回到桌面或是打开其他 **activity**，就会调用 **onStop()** 进入停止状态（保留所有的状态和成员信息，**\*\*对用户不可见\*\***）
  - 系统内存不足，拥有更高限权的应用需要内存，那么该 **activity** 的进程就可能会被系统回收。（回收 **onPause()** 和 **onStop()** 状态的 **activity** 进程）要想重新打开就必须重新创建一遍。

如果用户返回到 **onStop()** 状态的 **activity**（又显示在前台了），系统会调用 **> onRestart() -> onStart() -> onResume()** 然后重新 **running** 在 **activity** 结束（调用 **finish()**）或是被系统杀死之前会调用 **onDestroy()** 方法释放所有占用的资源。

- **activity** 生命周期中三个嵌套的循环
  - **activity** 的完整生存期会在 **onCreate()** 调用和 **onDestroy()** 调用之间发生。
  - **activity** 的可见生存期会在 **onStart()** 调用和 **onStop()** 调用之间发生。系统会在 **activity** 的整个生存期内多次调用 **onStart()** 和 **onStop()**，因为 **activity** 可能会在显示和隐藏之间不断地来回切换。
  - **activity** 的前后台切换会在 **onResume()** 调用和 **onPause()** 之间发生。
- 因为这个状态可能会经常发生转换，为了避免切换迟缓引起的用户等待，这两个方法中的代码应该相当地轻量化。

### 1.2.1 **activity** 被回收的状态和信息保存和恢复过程

---

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        if (savedInstanceState != null){ // 判断是否有以前的保存状态信息
            savedInstanceState.get("Key");
        }
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        // TODO Auto-generated method stub
        // 可能被回收内存前保存状态和信息，
        Bundle data = new Bundle();
        data.putString("key", "last words before be kill");
        outState.putAll(data);
        super.onSaveInstanceState(outState);
    }
    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        if (savedInstanceState != null){ // 判断是否有以前的保存状态信息
            savedInstanceState.get("Key");
        }
        super.onRestoreInstanceState(savedInstanceState);
    }
}
```

---

#### 1. **onSaveInstanceState()** 方法

- 在 `activity` 可能被回收之前调用，用来保存自己的状态和信息，以便回收后重建时恢复数据（在 `onCreate()` 或 `onRestoreInstanceState()` 中恢复）。旋转屏幕重建 `activity` 会调用该方法，但其他情况在 `onPause()` 和 `onStop()` 状态的 `activity` 不一定会调用，下面是该方法的文档说明。
- One example of when `onPause()` and `onStop()` is called and not this method is when a user navigates back from activity B to activity A: there is no need to call `onSaveInstanceState()` on B because that particular instance will never be restored, so the system avoids calling it. An example when `onPause()` is called and not `onSaveInstanceState()` is when activity B is launched in front of activity A: the system may avoid calling `onSaveInstanceState()` on activity A if it isn't killed during the lifetime of B since the state of the user interface of A will stay intact.
- 也就是说，系统灵活的来决定调不调用该方法，但是如果调用就一定发生在 **`onStop()`** 方法之前，但并不保证发生在 **`onPause()`** 的前面还是后面。

## 2. `onRestoreInstanceState()` 方法

- 这个方法在 `onStart()` 和 `onPostCreate()` 之间调用，在 `onCreate()` 中也可以状态恢复，但有时候需要所有布局初始化完成后再恢复状态。
- `onPostCreate()`：一般不实现这个方法，当程序的代码开始运行时，它调用系统做最后的初始化工作。

## 1.3 启动模式

### 1.3.1 启动模式什么？

- 简单的说就是定义 `activity` 实例与 `task` 的关联方式。

### 1.3.2 为什么要定义启动模式？

- 为了实现一些默认启动（`standard`）模式之外的需求：
  - 让某个 `activity` 启动一个新的 `task`（而不是被放入当前 `task`）
  - 让 `activity` 启动时只是调出已有的某个实例（而不是在 `back stack` 顶创建一个新的实例）
  - 或者，你想在用户离开 `task` 时只保留根 `activity`，而 `back stack` 中的其它 `activity` 都要清空

### 1.3.3 怎样定义启动模式？

- 定义启动模式的方法有两种：

#### 1. 使用 `manifest` 文件

- 在 `manifest` 文件中 `activity` 声明时，利用 `activity` 元素的 `launchMode` 属性来设定 `activity` 与 `task` 的关系。

---

```
<activity
    android:launchMode="standard"
/>
```

---

- 注意：你用 `launchMode` 属性为 `activity` 设置的模式可以被启动 `activity` 的 `intent` 标志所覆盖。

(a) 有哪些启动模式?

- "standard" (默认模式)
  - 当通过这种模式来启动 Activity 时,Android 总会为目标 Activity 创建一个新的实例,并将该 Activity 添加到当前 Task 栈中。这种方式不会启动新的 Task,只是将新的 Activity 添加到原有的 Task 中。
- "singleTop"
  - 该模式和 standard 模式基本一致,但有一点不同:当将要被启动的 Activity 已经位于 Task 栈顶时,系统不会重新创建目标 Activity 实例,而是直接复用 Task 栈顶的 Activity。
- "singleTask"
  - Activity 在同一个 Task 内只有一个实例。
  - 如果将要启动的 Activity 不存在,那么系统将会创建该实例,并将其加入 Task 栈顶;
  - 如果将要启动的 Activity 已存在,且存在栈顶,直接复用 Task 栈顶的 Activity。
  - 如果 Activity 存在但是没有位于栈顶,那么此时系统会把位于该 Activity 上面的所有其他 Activity 全部移出 Task,从而使得该目标 Activity 位于栈顶。
- "singleInstance"
  - 无论从哪个 Task 中启动目标 Activity,只会创建一个目标 Activity 实例且会用一个全新的 Task 栈来装载该 Activity 实例(全局单例)。
  - 如果将要启动的 Activity 不存在,那么系统将会先创建一个全新的 Task,再创建目标 Activity 实例并将该 Activity 实例放入此全新的 Task 中。
  - 如果将要启动的 Activity 已存在,那么无论它位于哪个应用程序,哪个 Task 中;系统都会把该 Activity 所在的 Task 转到前台,从而使该 Activity 显示出来。

## 2. 使用 Intent 标志

- 在要启动 activity 时,你可以在传给 startActivity() 的 intent 中包含相应标志,以修改 activity 与 task 的默认关系。

---

```
Intent i = new Intent(this, NewActivity.class);
i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(i);
```

---

(a) 可以通过标志修改的默认模式有哪些?

- FLAG\_ACTIVITY\_NEW\_TASK
  - 与"singleTask"模式相同,在新的 task 中启动 activity。如果要启动的 activity 已经运行于某 task 中,则那个 task 将调入前台。
- FLAG\_ACTIVITY\_SINGLE\_TOP
  - 与"singleTop"模式相同,如果要启动的 activity 位于 back stack 顶,系统不会重新创建目标 Activity 实例,而是直接复用 Task 栈顶的 Activity。
- FLAG\_ACTIVITY\_CLEAR\_TOP
  - 此种模式在 launchMode 中没有对应的属性值。
  - 如果要启动的 activity 已经在当前 task 中运行,则不再启动一个新的实例,且所有在其上面的 activity 将被销毁。

(b) 关于启动模式的一些建议

- 一般不要改变 activity 和 task 默认的工作方式。如果你确定有必要修改默认方式,请保持谨慎,并确保 activity 在启动和从其它 activity 返回时的可用性,多做测试和安全方面的工作。

## 1.4 Intent Filter

- android 的 3 个核心组件——Activity、services、广播接收器——是通过 intent 传递消息的。intent 消息用于在运行时绑定不同的组件。
- 在 Android 的 AndroidManifest.xml 配置文件中可以通过 intent-filter 节点为一个 Activity 指定其 Intent Filter，以便告诉系统该 Activity 可以响应什么类型的 Intent。

### 1.4.1 intent-filter 的三大属性

#### 1. Action

- 一个 Intent Filter 可以包含多个 Action，Action 列表用于标示 Activity 所能接受的“动作”，它是一个用户自定义的字符串。

```
<intent-filter >
  <action android:name="android.intent.action.MAIN" />
  <action android:name="com.scu.amazing7Action" />
  .....
</intent-filter>
```

在代码中使用以下语句便可以启动该 Intent 对象: `Intent i=new Intent(); i.setAction("com.scu.amazing7Action")`。Action 列表中包含了“com.scu.amazing7Action”的 Activity 都将会匹配成功

2. URL 在 intent-filter 节点中,通过 data 节点匹配外部数据,也就是通过 URI 携带外部数据给目标组件。`<data android:mimeType="mimeType" android:scheme="scheme" android:host="host" android:port="port" android:path="path"/>` 注意: 只有 data 的所有的属性都匹配成功时 URI 数据匹配才会成功
3. Category 为组件定义一个类别列表,当 Intent 中包含这个类别列表的所有项目时才会匹配成功。`<intent-filter . . . > <action android:name="code android.intent.action.MAIN" /> <category android:name="code android.intent.category.LAUNCHER" /> </intent-filter>`

### 1.4.2 Activity 种 Intent Filter 的匹配过程

加载所有的 Intent Filter 列表 去掉 action 匹配失败的 Intent Filter 去掉 url 匹配失败的 Intent Filter 去掉 Category 匹配失败的 Intent Filter 判断剩下的 Intent Filter 数目是否为 0。如果为 0 查找失败返回异常; 如果大于 0, 就按优先级排序, 返回最高优先级的 Intent Filter

## 1.5 开发中 Activity 的一些问题

- 一般设置 Activity 为非公开的

```
<activity
  android:exported="false" />
```

- 注意: 非公开的 Activity 不能设置 intent-filter, 以免被其他 activity 唤醒 (如果拥有相同的 intent-filter)。
  - 不要指定 activity 的 taskAffinity 属性
  - 不要设置 activity 的 LaunchMode (保持默认)
- 注意 Activity 的 intent 最好也不要设定为 FLAG\_ACTIVITY\_NEW\_TASK
  - 在匿名内部类中使用 this 时加上 activity 类名 (类名.this, 不一定是当前 activity)



- 设置 activity 全屏

\* 在其 onCreate() 方法中加入:

```
// 设置全屏模式
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
// 去除标题栏
requestWindowFeature(Window.FEATURE_NO_TITLE);
```

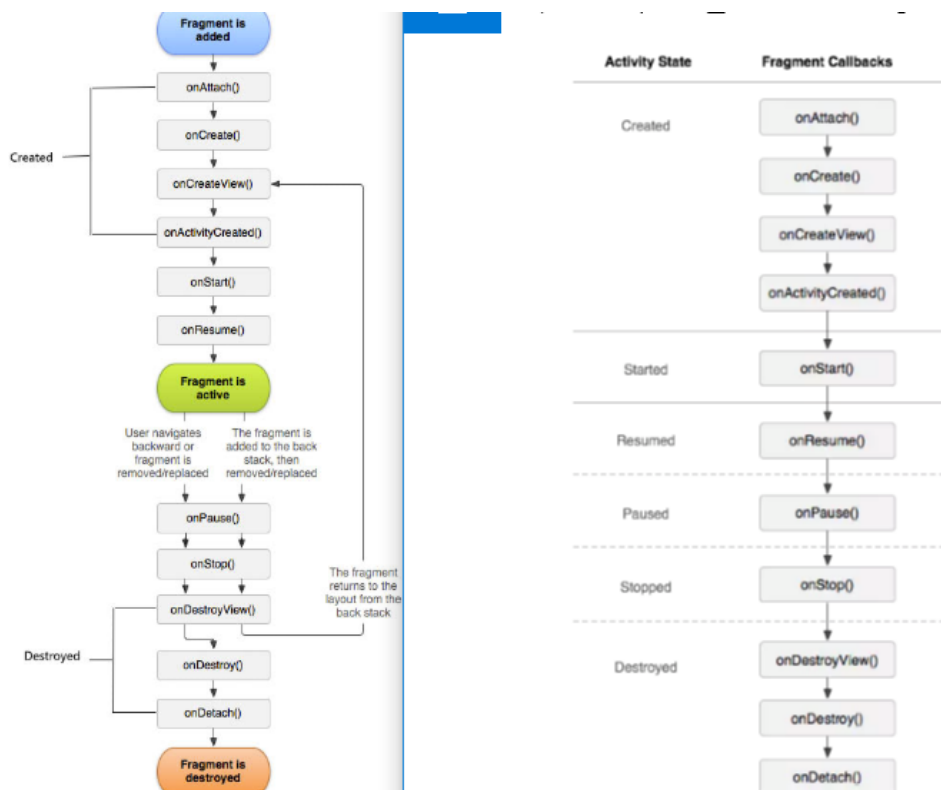
## 2 Fragment 全解析

### 2.1 概述

- **Fragment** 是 **Activity** 中用户界面的一个行为或者是一部分。主要是支持在大屏幕上动态和更为灵活的去组合或是交换 UI 组件, 通过将 **activity** 的布局分割成若干个 **fragment**, 可以在运行时编辑 **activity** 的呈现, 并且那些变化会被保存在由 **activity** 管理的后台栈里面。
- **Fragment** 必须总是被嵌入到一个 **activity** 之中, 并且 **fragment** 的生命周期直接受其宿主 **activity** 的生命周期的影响。你可以认为 **fragment** 是 **activity** 的一个模块零件, 它有自己的生命周期, 接收它自己的输入事件, 并且可以在 **activity** 运行时添加或者删除。
- 应该将每一个 **fragment** 设计为模块化的和可复用化的 **activity** 组件。也就是说, 你可以在多个 **activity** 中引用同一个 **fragment**, 因为 **fragment** 定义了它自己的布局, 并且使用它本身生命周期回调的行为。

### 2.2 Fragment 的生命周期

- 先看 fragment 生命周期图:



- **fragment** 所生存的 **activity** 生命周期直接影响着 **fragment** 的生命周期，由此针对 **activity** 的每一个生命周期回调都会引发一个 **fragment** 类似的回调。例如，当 **activity** 接收到 **onPause()** 时，这个 **activity** 之中的每个 **fragment** 都会接收到 **onPause()**。

[这有 **Activity** 的详细说明](<http://blog.csdn.net/amazing7/article/details/51244219>)

- **Fragment** 有一些额外的生命周期回调方法（创建和销毁 **fragment** 界面）。
  - **onAttach()**
    - \* 当 **fragment** 被绑定到 **activity** 时调用（**Activity** 会被传入）。
  - **onCreateView()**
    - \* 将本身的布局构建到 **activity** 中去（**fragment** 作为 **activity** 界面的一部分）
  - **onActivityCreated()**
    - \* 当 **activity** 的 **onCreate()** 函数返回时被调用。
  - **onDestroyView()**
    - \* 当与 **fragment** 关联的视图体系正被移除时被调用。
  - **onDetach()**
    - \* 当 **fragment** 正与 **activity** 解除关联时被调用。
- 当 **activity** 接收到它的 **onCreate()** 回调时，**activity** 之中的 **fragment** 接收到 **onActivityCreated()** 回调。
  - 一旦 **activity** 处于 **resumed** 状态，则可以在 **activity** 中自由的添加或者移除 **fragment**。因此，只 \*\* 有当 **activity** 处于 **resumed** 状态时 \*\*，**fragment** 的生命周期才可以独立变化。**fragment** 会在 **activity** 离开恢复状态时 再一次被 **activity** 推入它的生命周期中。
- 管理 **fragment** 生命周期与管理 **activity** 生命周期很相像。像 **activity** 一样，**fragment** 也有三种状态：
  - **Resumed**
    - \* **fragment** 在运行中的 **activity** 可见。
  - **Paused**
    - \* 另一个 **activity** 处于前台且得到焦点，但是这个 **fragment** 所在的 **activity** 仍然可见（前台 **activity** 部分透明，或者没有覆盖全屏）。
  - **Stopped**
    - \* **fragment** 不可见。要么宿主 **activity** 已经停止，要么 **fragment** 已经从 **activity** 上移除，但已被添加到后台栈中。一个停止的 **fragment** 仍然活着（所有状态和成员信息仍然由系统保留着）。但是，它对用户来讲已经不再可见，并且如果 **activity** 被杀掉，它也将被杀掉。
    - \* 如果 **activity** 的进程被杀掉了，在 **activity** 被重新创建时，你需要恢复 **fragment** 状态。可以执行 **fragment** 的 **onSaveInstanceState()** 来保存状态（注意在 **fragment** 是在 **onCreate()**，**onCreateView()**，或 **onActivityCreated()** 中进行恢复）。
- 在生命周期方面，**activity** 与 **fragment** 之间一个 很重要的不同，就是在各自的后台栈中是如何存储的。
  - 当 **activity** 停止时，默认情况下 **activity** 被安置在由系统管理的 **activity** 后台栈中；
  - **fragment** 仅当在一个事务被移除时，通过显式调用 **addToBackStack()** 请求保存的实例，该 **fragment** 才被置于由宿主 **activity** 管理的后台栈。

- 要创建一个 **fragment**，必须创建一个 **fragment** 的子类。一般情况下，我们至少需要实现以下几个 **fragment** 生命周期方法：

- **onCreate()**

- \* 在创建 **fragment** 时系统会调用此方法。在实现代码中，你可以初始化想要在 **fragment** 中保持的那些必要组件，当 **fragment** 处于暂停或者停止状态之后可重新启用它们。

- **onCreateView()**

- \* 在第一次为 **fragment** 绘制用户界面时系统会调用此方法。为 **fragment** 绘制用户界面，这个函数必须要返回所绘出的 **fragment** 的根 **View**。如果 **fragment** 没有用户界面可以返回空。

---

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.example_fragment, container, false);
}
```

---

- **inflate()** 函数需要以下三个参数：

- \* 要 **inflate** 的布局的资源 ID。
  - \* 被 **inflate** 的布局的父 **ViewGroup**。
  - \* 一个布尔值，表明在 **inflate** 期间被 **inflat** 的布局是否应该附上 **ViewGroup**（第二个参数 **container**）。（在这个例子中传入的是 **false**，因为系统已经将被 **inflate** 的布局插入到容器中（**container**）——传入 **true** 会在最终的布局里创建一个多余的 **ViewGroup**。）

- **onPause()**

- \* 系统回调用该函数作为用户离开 **fragment** 的第一个预兆（尽管这并不总意味着 **fragment** 被销毁）。在当前用户会话结束之前，通常要在这里提交任何应该持久化的变化（因为用户可能不再返回）。

## 2.3 将 **fragment** 添加到 **activity** 之中

- 可以通过在 **activity** 布局文件中声明 **fragment**，用 **fragment** 标签把 **fragment** 插入到 **activity** 的布局中，或者用应用程序源码将它添加到一个存在的 **ViewGroup** 中。
- 但 **fragment** 并不是一个定要作为 **activity** 布局的一部分，**fragment** 也可以为 **activity** 隐身工作。

### 2.3.1 在 **activity** 的布局文件里声明 **fragment**

- 可以像为 **view** 一样为 **fragment** 指定布局属性。例如：

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.test.FragmentOne"
        android:id="@+id/fo"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

---

- **fragment** 标签中的 **android:name** 属性指定了布局中实例化的 **Fragment** 类。
- 当系统创建 **activity** 布局时，它实例化了布局文件中指定的每一个 **fragment**，并为它们调用 **onCreateView()** 函数，以获取每一个 **fragment** 的布局。系统直接在 **<fragment>** 元素的位置插入 **fragment** 返回的 **View**。
- 注意：每个 **fragment** 都需要一个唯一的标识，如果重启 **activity**，系统可用来恢复 **fragment**（并且可用来捕捉 **fragment** 的事务处理，例如移除）。
- 为 **fragment** 提供 ID 有三种方法：
  - 用 **android:id** 属性提供一个唯一的标识。
  - 用 **android:tag** 属性提供一个唯一的字符串。
  - 如果上述两个属性都没有，系统会使用其容器视图（**view**）的 ID。

### 2.3.2 通过编码将 **fragment** 添加到已存在的 **ViewGroup** 中

- 在 **activity** 运行的任何时候，你都可以将 **fragment** 添加到 **activity** 布局中。
- 要管理 **activity** 中的 **fragment**，可以使用 **FragmentManager**。可以通过在 **activity** 中调用 **getFragmentManager()** 获得。使用 **FragmentManager** 可以做如下事情，包括：
  - 使用 **findFragmentById()**（用于在 **activity** 布局中提供有界面的 **fragment**）或者 **findFragmentByTag()** 获取 **activity** 中存在的 **fragment**（用于有界面或者没有界面的 **fragment**）。
  - 使用 **popBackStack()**（模仿用户的 **BACK** 命令）从后台栈弹出 **fragment**。
  - 使用 **addOnBackStackChangeListener()** 注册一个监听后台栈变化的监听器。
- 在 **Android** 中，对 **Fragment** 的事务操作都是通过 **FragmentTransaction** 来执行。操作大致可以分为两类：
  - 显示：**add()** **replace()** **show()** **attach()**
  - 隐藏：**remove()** **hide()** **detach()**
  - 说明：
    - \* 调用 **show()** & **hide()** 方法时，**Fragment** 的生命周期方法并不会被执行，仅仅是 **Fragment** 的 **View** 被显示或者隐藏。
    - \* 执行 **replace()** 时（至少两个 **Fragment**），会执行第二个 **Fragment** 的 **onAttach()** 方法、执行第一个 **Fragment** 的 **onPause()-onDetach()** 方法，同时 **containerView** 会 **detach** 第一个 **Fragment** 的 **View**。
    - \* **add()** 方法执行 **onAttach()-onResume()** 的生命周期，相对的 **remove()** 就是执行完成剩下的 **onPause()-onDetach()** 周期。
- 可以像下面这样从 **Activity** 中取得 **FragmentTransaction** 的实例：

---

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

---

- 可以用 **add()** 函数添加 **fragment**，并指定要添加的 **fragment** 以及要将其插入到哪个视图（**view**）之中（注意 **commit** 事务）：

---

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

---

### 2.3.3 添加没有界面的 **fragment**

- 也可以使用 **fragment** 为 **activity** 提供后台动作，却不呈现多余的用户界面。
- 想要添加没有界面的 **fragment**，可以使用 `add(Fragment, String)`（为 **fragment** 提供一个唯一的字符串“tag”，而不是视图（view）ID）。这样添加了 **fragment**，但是，因为还没有关联到 **activity** 布局中的视图（view），收不到 `onCreateView()` 的调用。所以不需要实现这个方法。
- 对于无界面 **fragment**，字符串标签是 \*\* 唯一识别 \*\* 它的方法。如果之后想从 **activity** 中取到 **fragment**，需要使用 `findFragmentByTag()`。

## 2.4 **fragment** 事务后台栈

- 在调用 `commit()` 之前，可以将事务添加到 **fragment** 事务后台栈中（通过调用 `addToBackStack()`）。这个后台栈由 **activity** 管理，并且允许用户通过按 BACK 键回退到前一个 **fragment** 状态。
- 下面的代码中一个 **fragment** 代替另一个 **fragment**，并且将之前的 **fragment** 状态保留在后台栈中：

---

```
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction = getFragmentManager().beginTransaction();
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);
transaction.commit();
```

---

- 注意：
  - 如果添加多个变更事务（例如另一个 `add()` 或者 `remove()`）并调用 `addToBackStack()`，那么在调用 `commit()` 之前的所有应用的变更被作为一个单独的事务添加到后台栈中，并且 BACK 键可以将它们一起回退。
  - 当移除一个 **fragment** 时，如果调用了 `addToBackStack()`，那么之后 **fragment** 会被停止，如果用户回退，它将被恢复过来。
  - 调用 `commit()` 并不立刻执行事务，相反，而是采取预约方式，一旦 **activity** 的界面线程（主线程）准备好便可运行起来。然而，如果有必要的话，你可以从界面线程调用 `executePendingTransactions()` 立即执行由 `commit()` 提交的事务。
  - 只能在 **activity** 保存状态（当用户离开 **activity** 时）之前用 `commit()` 提交事务。如果你尝试在那时之后提交，会抛出一个异常。这是因为如果 **activity** 需要被恢复，提交后的状态会被丢失。对于这类丢失提交的情况，可使用 `commitAllowingStateLoss()`

## 2.5 与 **Activity** 交互

- **Activity** 中已经有了该 **Fragment** 的引用，直接通过该引用进行交互。
  - 如果没引用可以通过调用 **fragment** 的函数 `findFragmentById()` 或者 `findFragmentByTag()`，从 **FragmentManager** 中获取 **Fragment** 的索引，例如：

---

```
ExampleFragment fragment = (ExampleFragment) getFragmentManager().findFragmentById(R.id.example_fragment);
```

---

- 在 **Fragment** 中可以通过 `getActivity` 得到当前绑定的 **Activity** 的实例。
- 创建 **activity** 事件回调函数，在 **fragment** 内部定义一个回调接口，宿主 **activity** 来实现它。

## 3 Service 全面总结

### 3.1 什么是服务？

- **Service** 是一个应用程序组件，它能够在后台执行一些耗时较长的操作，并且不提供用户界面。服务能被其它应用程序的组件启动，即使用户切换到另外的应用时还能保持后台运行。此外，应用程序组件还能与服务绑定，并与服务进行交互，甚至能进行进程间通信（IPC）。比如，服务可以处理网络传输、音乐播放、执行文件 I/O、或者与 **content provider** 进行交互，所有这些都是后台进行的。

### 3.2 Service 与 Thread 的区别

- 服务仅仅是一个组件，即使用户不再与你的应用程序发生交互，它仍然能在后台运行。因此，应该只在需要时才创建一个服务。
- 如果你需要在主线程之外执行一些工作，但仅当用户与你的应用程序交互时才会用到，那你应该创建一个新的线程而不是创建服务。比如，如果你需要播放一些音乐，但只是当你的 **activity** 在运行时才需要播放，你可以在 **onCreate()** 中创建一个线程，在 **onStart()** 中开始运行，然后在 **onStop()** 中终止运行。还可以考虑使用 **AsyncTask** 或 **HandlerThread** 来取代传统的 **Thread** 类。
- 由于无法在不同的 **Activity** 中对同一 **Thread** 进行控制，这个时候就要考虑用服务实现。如果你使用了服务，它默认就运行于应用程序的主线程中。因此，如果服务执行密集计算或者阻塞操作，你仍然应该在服务中创建一个新的线程来完成（避免 ANR）。

### 3.3 服务的分类

#### 3.3.1 按运行分类

- 前台服务
  - 前台服务是指那些经常会被用户关注的服务，因此内存过低时它不会成为被杀的对象。前台服务必须提供一个状态栏通知，并会置于“正在进行的”（“Ongoing”）组之下。这意味着只有在服务被终止或从前台移除之后，此通知才能被解除。
  - 例如，用服务来播放音乐的播放器就应该运行在前台，因为用户会清楚地知晓它的运行情况。状态栏通知可能会标明当前播放的歌曲，并允许用户启动一个 **activity** 来与播放器进行交互。
  - 要把你的服务请求为前台运行，可以调用 **startForeground()** 方法。此方法有两个参数：唯一标识通知的整数值、状态栏通知 **Notification** 对象。例如：

---

```
Notification notification = new Notification(R.drawable.icon,
                                           getText(R.string.ticker_text),
                                           System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
                                                       notificationIntent, 0);
notification.setLatestEventInfo(this, getText(R.string.notification_title),
                                getText(R.string.notification_message),
                                pendingIntent);
startForeground(ONGOING_NOTIFICATION, notification);
```

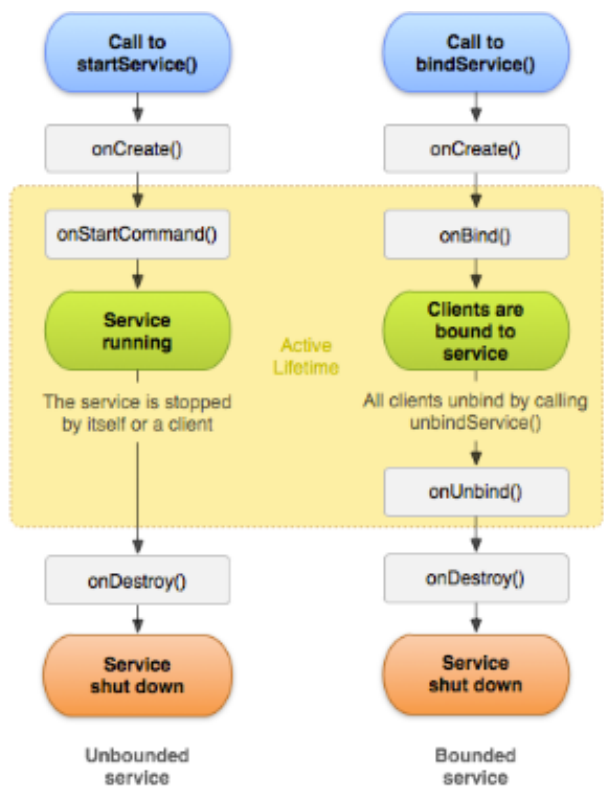
---

- 要从前台移除服务，请调用 **stopForeground()** 方法，这个方法接受个布尔参数，表示是否同时移除状态栏通知。此方法不会终止服务。不过，如果服务在前台运行时被你终止了，那么通知也会同时被移除。
- 后台服务

3.3.2 按使用分类

- 本地服务
  - 用于应用程序内部，实现一些耗时任务，并不占用应用程序比如 Activity 所属线程，而是单开线程后台执行。
  - 调用 Context.startService() 启动，调用 Context.stopService() 结束。在内部可以调用 Service.stopSelf() 或 Service.stopSelfResult() 来自自己停止。
- 远程服务
  - 用于 Android 系统内部的应用程序之间，可被其他应用程序复用，比如天气预报服务，其他应用程序不需要再写这样的服务，调用已有的即可。可以定义接口并把接口暴露出来，以便其他应用进行操作。客户端建立到服务对象的连接，并通过那个连接来调用服务。调用 Context.bindService() 方法建立连接，并启动，以调用 Context.unbindService() 关闭连接。多个客户端可以绑定至同一个服务。如果服务此时还没有加载，bindService() 会先加载它。

3.4 Service 生命周期



- Service 生命周期方法:

```
public class ExampleService extends Service {
    int mStartMode; // 标识服务被杀死后的处理方式
    IBinder mBinder; // 用于客户端绑定的接口
    boolean mAllowRebind; // 标识是否使用 onRebind
    @Override
    public void onCreate() {
        // 服务正被创建
    }
}
```



```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // 服务正在启动, 由 startService() 调用引发
    return mStartMode;
}
@Override
public IBinder onBind(Intent intent) {
    // 客户端用 bindService() 绑定服务
    return mBinder;
}
@Override
public boolean onUnbind(Intent intent) {
    // 所有的客户端都用 unbindService() 解除了绑定
    return mAllowRebind;
}
@Override
public void onRebind(Intent intent) {
    // 某客户端正用 bindService() 绑定到服务,
    // 而 onUnbind() 已经被调用过了
}
@Override
public void onDestroy() {
    // 服务用不上了, 将被销毁
}
}

```

- 请注意 `onStartCommand()` 方法必须返回一个整数。这个整数是描述系统在杀死服务之后应该如何继续运行。`onStartCommand()` 的返回值必须是以下常量之一：

- \* `START_NOT_STICKY`

- 如果系统在 `onStartCommand()` 返回后杀死了服务，则不会重建服务了，除非还存在未发送的 `intent`。当服务不再是必需的，并且应用程序能够简单地重启那些未完成的工作时，这是避免服务运行的最安全的选项。

- \* `START_STICKY`

- 如果系统在 `onStartCommand()` 返回后杀死了服务，则将重建服务并调用 `onStartCommand()`，但不会再次送入上一个 `intent`，而是用 `null intent` 来调用 `onStartCommand()`。除非还有启动服务的 `intent` 未发送完，那么这些剩下的 `intent` 会继续发送。这适用于媒体播放器（或类似服务），它们不执行命令，但需要一直运行并随时待命。

- \* `START_REDELIVER_INTENT`

- 如果系统在 `onStartCommand()` 返回后杀死了服务，则将重建服务并用上一个已送过的 `intent` 调用 `onStartCommand()`。任何未发送完的 `intent` 也都会依次送入。这适用于那些需要立即恢复工作的活跃服务，比如下载文件。

- 服务的生命周期与 `activity` 的非常类似。不过，更重要的是你需密切关注服务的创建和销毁环节，因为后台运行的服务是不会引起用户注意的。
- 服务的生命周期——从创建到销毁——可以有两种路径：

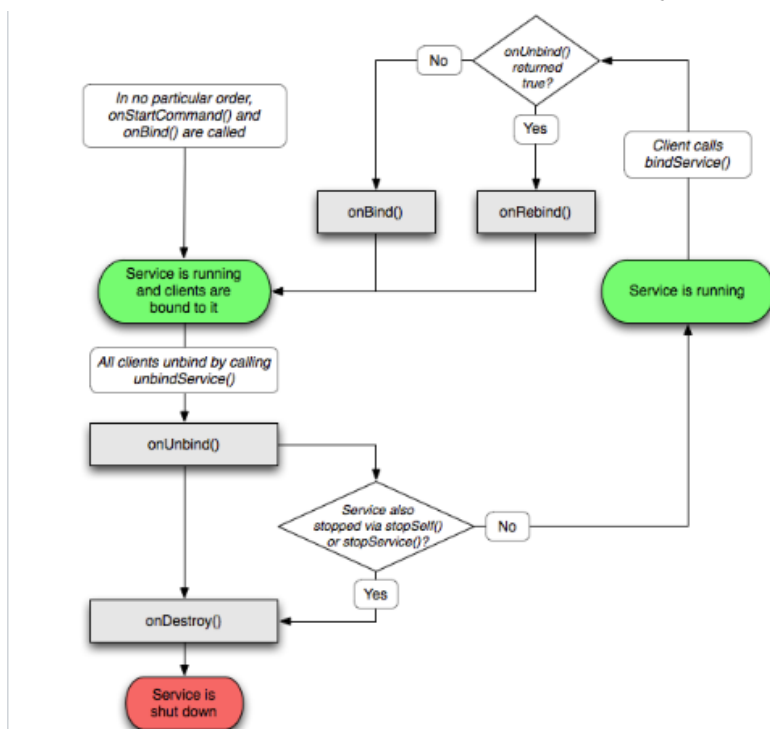
- 一个 `started` 服务

- \* 这类服务由其它组件调用 `startService()` 来创建。然后保持运行，且必须通过调用 `stopSelf()` 自行终止。其它组件也可通过调用 `stopService()` 终止这类服务。服务终止后，系统会把它销毁。
- \* 如果一个 `Service` 被 `startService` 方法多次启动，那么 `onCreate` 方法只会调用一次，`onStart` 将会被调用多次（对应调用 `startService` 的次数），并且系统只会创建 `Service` 的一个实例（因此你应该知道只需要一次 `stopService` 调用）。该 `Service` 将会一直在后台运行，而不管对应程序的 `Activity` 是否在运行，直到被调用 `stopService`，或自身的 `stopSelf` 方法。当然如果系统资源不足，`android` 系统也可能结束服务。



## - 一个 bound 服务

- \* 服务由其它组件（客户端）调用 `bindService()` 来创建。然后客户端通过一个 `IBinder` 接口与服务进行通信。客户端可以通过调用 `unbindService()` 来关闭联接。多个客户端可以绑定到同一个服务上，当所有的客户端都解除绑定后，系统会销毁服务。（服务不需要自行终止。）
- \* 如果一个 `Service` 被某个 `Activity` 调用 `Context.bindService` 方法绑定启动，不管调用 `bindService` 调用几次，`onCreate` 方法都只会调用一次，同时 `onStart` 方法始终不会被调用。当连接建立之后，`Service` 将会一直运行，除非调用 `Context.unbindService` 断开连接或者之前调用 `bindService` 的 `Context` 不存在了（如 `Activity` 被 `finish` 的时候），系统将会自动停止 `Service`，对应 `onDestroy` 将被调用。



- 这两条路径并不是完全隔离的。也就是说，你可以绑定到一个已经用 `startService()` 启动的服务上。例如，一个后台音乐服务可以通过调用 `startService()` 来启动，传入一个指明所需播放音乐的 `Intent`。之后，用户也许需要用播放器进行一些控制，或者需要查看当前歌曲的信息，这时一个 `activity` 可以通过调用 `bindService()` 与此服务绑定。在类似这种情况下，`stopService()` 或 `stopSelf()` 不会真的终止服务，除非所有的客户端都解除了绑定。
  - 当在旋转手机屏幕的时候，当手机屏幕在“横”“竖”变换时，此时如果你的 `Activity` 如果会自动旋转的话，旋转其实是 `Activity` 的重新创建，因此旋转之前的使用 `bindService` 建立的连接便会断开（`Context` 不存在了）。

## 3.5 在 manifest 中声明服务

- 无论是什么类型的服务都必须在 `manifest` 中申明，格式如下：

```
<manifest ... >
  <application ... >
    <service android:name=".ExampleService" />
  </application>
</manifest>
```

- Service 元素的属性有：

android:name	服务类名
android:label	服务的名字. 如果此项不设置, 那么默认显示的服务名则为类名
android:icon	服务的图标
android:permission	申明此服务的权限, 这意味着只有提供了该权限的应用才能控制或连接此服务
android:process	表示该服务是否运行在另外一个进程, 如果设置了此项, 那么将会在包名后面加上这段字符串表示另一进程的名字
android:enabled	如果此项设置为 true, 那么 Service 将会默认被系统启动, 不设置默认此项为 false
android:exported	表示该服务是否能够被其他应用程序所控制或连接, 不设置默认此项为 false

- android:name 是唯一必需的属性——它定义了服务的类名。与 activity 一样，服务可以定义 intent 过滤器，使得其它组件能用隐式 intent 来调用服务。如果你想让服务只能内部使用（其它应用程序无法调用），那么就不必（也不应该）提供任何 intent 过滤器。
- 此外，如果包含了 android:exported 属性并且设置为“false”，就可以确保该服务是你应用程序的私有服务。即使服务提供了 intent 过滤器，本属性依然生效。

### 3.6 startService 启动服务

- 从 activity 或其它应用程序组件中可以启动一个服务，调用 startService() 并传入一个 Intent（指定所需启动的服务）即可。

---

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

---

- 服务类：

---

```
public class MyService extends Service {
    // onBind 是 Service 的虚方法，因此我们不得不实现它。
    // 返回 null，表示客户端不能建立到此服务的连接。
    @Override
    public IBinder onBind(Intent intent) {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        //接受传递过来的 intent 的数据
        return START_STICKY;
    };
    @Override
    public void onDestroy() {
        super.onDestroy();
    }
}
```

---

- 一个 started 服务必须自行管理生命周期。也就是说，系统不会终止或销毁这类服务，除非必须恢复系统内存并且服务返回后一直维持运行。因此，服务必须通过调用 stopSelf() 自行终止，或者其它组件可通过调用 stopService() 来终止它。

### 3.7 bindService 启动服务

- 当应用程序中的 activity 或其它组件需要与服务进行交互，或者应用程序的某些功能需要暴露给其它应用程序时，你应该创建一个 bound 服务，并通过进程间通信（IPC）来完成。

- 方法如下:

---

```
Intent intent=new Intent(this,BindService.class);
bindService(intent, ServiceConnection conn, int flags)
```

---

- 注意 bindService 是 Context 中的方法, 当没有 Context 时传入即可。
- 在进行服务绑定的时, 其 flags 有:
  - Context.BIND\_AUTO\_CREATE
    - \* 表示收到绑定请求的时候, 如果服务尚未创建, 则即刻创建, 在系统内存不足需要先摧毁优先级组件来释放内存, 且只有驻留该服务的进程成为被摧毁对象时, 服务才被摧毁
  - Context.BIND\_DEBUG\_UNBIND
    - \* 通常用于调试场景中判断绑定的服务是否正确, 但容易引起内存泄漏, 因此非调试目的的时候不建议使用
  - Context.BIND\_NOT\_FOREGROUND
    - \* 表示系统将阻止驻留该服务的进程具有前台优先级, 仅在后台运行。
- 服务类:

---

```
public class BindService extends Service {
    // 实例化 MyBinder 得到 mybinder 对象;
    private final MyBinder binder = new MyBinder();
    // 返回 Binder 对象。
    @Override
    public IBinder onBind(Intent intent) {
        // TODO Auto-generated method stub
        return binder;
    }
    // 新建内部类 MyBinder, 继承自 Binder(Binder 实现 IBinder 接口),
    // MyBinder 提供方法返回 BindService 实例。
    public class MyBinder extends Binder{
        public BindService getService(){
            return BindService.this;
        }
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // TODO Auto-generated method stub
        return super.onUnbind(intent);
    }
}
```

---

- 启动服务的 activity 代码:

---

```
public class MainActivity extends Activity {
    // 是否绑定
    boolean mIsBound = false;
    BindService mBoundService;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        doBindService();
    }
    // 实例化 ServiceConnection 接口的实现类, 用于监听服务的状态
    private ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            BindService mBoundService = ((BindService.MyBinder) service).getService();
        }
    }
}
```

---

```

        @Override
        public void onServiceDisconnected(ComponentName name) {
            mBoundService = null;
        }
    };
    // 绑定服务
    public void doBindService() {
        bindService(new Intent(MainActivity.this, BindService.class), conn, Context.BIND_AUTO_CREATE);
        mIsBound = true;
    }
    // 解除绑定服务
    public void doUnbindService() {
        if (mIsBound) {
            // Detach our existing connection.
            unbindService(conn);
            mIsBound = false;
        }
    }
    @Override
    protected void onDestroy() {
        // TODO Auto-generated method stub
        super.onDestroy();
        doUnbindService();
    }
}

```

- 注意在 AndroidManifest.xml 中对 Service 进行显式声明
- 判断 Service 是否正在运行:

```

private boolean isServiceRunning() {
    ActivityManager manager = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
    {
        if ("com.example.demo.BindService".equals(service.service.getClassName())) {
            return true;
        }
    }
    return false;
}

```

## 4 IntentService 使用详解和实例但介绍

### 4.1 IntentService 定义

- IntentService 继承与 Service，用来处理异步请求。客户端可以通过 startService(Intent) 方法传递请求给 IntentService。IntentService 在 onCreate() 函数中通过 HandlerThread 单独开启一个线程来依次处理所有 Intent 请求对象所对应的任务。
- 这样以免事务处理阻塞主线程（A N R）。执行完所一个 Intent 请求对象所对应的工作之后，如果没有新的 Intent 请求达到，则 \*\*自动停止 \*\*Service；否则执行下一个 Intent 请求所对应的任务。
- IntentService 在处理事务时，还是采用的 Handler 方式，创建一个名叫 ServiceHandler 的内部 Handler，并把它直接绑定到 HandlerThread 所对应的子线程。ServiceHandler 把一个 intent 所对应的事务都封装到叫做 \*\*onHandleIntent\*\* 的虚函数；因此我们直接实现虚函数 onHandleIntent，再在里面根据 Intent 的不同进行不同的事务处理就可以了。
- 另外，IntentService 默认实现了 Onbind() 方法，返回值为 null。
- 使用 IntentService 需要实现的两个方法：
  - 构造函数

- \* `IntentService` 的构造函数一定是 **\*\* 参数为空 \*\*** 的构造函数，然后再在其中调用 `super("name")` 这种形式的构造函数。因为 `Service` 的实例化是系统来完成的，而且系统是用参数为空的构造函数来实例化 `Service` 的
- 实现虚函数 `onHandleIntent`
  - \* 在里面根据 `Intent` 的不同进行不同的事务处理。
  - \* 好处：处理异步请求的时候可以减少写代码的工作量，比较轻松地实现项目的需求。

## 4.2 `IntentService` 与 `Service` 的区别

- `Service` 不是独立的进程，也不是独立的线程，它是依赖于应用程序的主线程的，不建议在 `Service` 中编写耗时的逻辑和操作，否则会引起 ANR。
- `IntentService` 它创建了一个独立的工作线程来处理所有的通过 `onStartCommand()` 传递给服务的 intents（把 intent 插入到工作队列中）。通过工作队列把 intent 逐个发送给 `onHandleIntent()`。
- 不需要主动调用 `stopSelf()` 来结束服务。因为，在所有的 intent 被处理完后，系统会自动关闭服务。
- 默认实现的 `onBind()` 返回 `null`。

## 4.3 `IntentService` 实例介绍

- 首先是 `myIntentService.java`

---

```
public class myIntentService extends IntentService {
    //-----必须实现-----
    public myIntentService() {
        super("myIntentService");
        // 注意构造函数参数为空，这个字符串就是 worker thread 的名字
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        //根据 Intent 的不同进行不同的事务处理
        String taskName = intent.getExtras().getString("taskName");
        switch (taskName) {
            case "task1":
                Log.i("myIntentService", "do task1");
                break;
            case "task2":
                Log.i("myIntentService", "do task2");
                break;
            default:
                break;
        }
    }
    //-----用于打印生命周期-----
    @Override
    public void onCreate() {
        Log.i("myIntentService", "onCreate");
        super.onCreate();
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i("myIntentService", "onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public void onDestroy() {
        Log.i("myIntentService", "onDestroy");
        super.onDestroy();
    }
}
```

---

- 然后记得在 Manifest.xml 中注册服务

---

```
<service android:name=".myIntentService">
    <intent-filter >
        <action android:name="cn.scu.finch"/>
    </intent-filter>
</service>
```

---

- 最后在 Activity 中开启服务

---

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        //同一服务只会开启一个 worker thread, 在 onHandleIntent 函数里依次处理 intent 请求。
        Intent i = new Intent("cn.scu.finch");
        Bundle bundle = new Bundle();
        bundle.putString("taskName", "task1");
        i.putExtras(bundle);
        startService(i);
        Intent i2 = new Intent("cn.scu.finch");
        Bundle bundle2 = new Bundle();
        bundle2.putString("taskName", "task2");
        i2.putExtras(bundle2);
        startService(i2);
        startService(i); //多次启动
    }
}
```

---

- 运行结果:

---

```
myIntentService onCreate()
myIntentService onStartCommand()
myIntentService onStartCommand()
myIntentService do task1
myIntentService onStartCommand()
myIntentService do task2
myIntentService do task1
myIntentService onDestroy()
```

---

- IntentService 在 onCreate() 函数中通过 HandlerThread 单独开启一个线程来依次处理所有 Intent 请求对象所对应的任务。
- 通过 onStartCommand() 传递给服务 intent 被 \*\* 依次 \*\* 插入到工作队列中。工作队列又把 intent 逐个发送给 onHandleIntent()。
- 注意:
  - 它只有一个工作线程，名字就是构造函数的那个字符串，也就是“myIntentService”，我们知道多次开启 service，只会调用一次 onCreate 方法（创建一个工作线程），多次 onStartCommand 方法（用于传入 intent 通过工作队列再发给 onHandleIntent 函数做处理）。

## 5 Android 启动过程图解

Android 手机开机执行过程图:![这里写图片描述](<http://img.blog.csdn.net/20160603133621981>)

从开机到桌面的过程为: **Bootloader** \*\*Kernel\*\* \*\*Init 进程 \*\* Zygote **SystemService** **ServiceManager** **Home Launcher** Android 服务包括系统服务和应用服务，系统服务是指 Android 系统在启动过程就已经启动实现了的服务，对于系统服务又分为 Java 服务和本地服务，Java 服务是由 Java 代码编写而成，由 SystemServer 进程提供，而本地服务是由 C/C++ 实现的服务，由 Init 进程在系统启动时启动的服务。应用服务是由开发者自行实现的某些特定服务。1、Bootloader

### 5.0.1

当电源按下，引导芯片代码开始从预定义的地方（固化在 ROM）开始执行。加载引导程序到 RAM，然后执行。BootLoader 是在操作系统内核运行之前运行。可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境带到一个合适状态，以便为最终调用操作系统内核准备好正确的环境。2、Kernel

### 5.0.2

Android 内核启动时，会设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找”init”文件，然后启动 root 进程或者系统的第一个进程。3、init 进程

### 5.0.3

init 进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序 init 的方式，完成引导进程。init 始终是第一个进程。启动过程就是代码 init.c 中 main 函数执行过程：system.c 在函数中执行了：\*\* 文件夹建立 \*\*，\*\* 挂载 \*\*，\*\*rc 文件解析 \*\*，\*\* 属性设置 \*\*，\*\* 启动服务 \*\*，\*\* 执行动作 \*\*，\*\*socket 监听 \*\*.....

- rc 文件解析

.rc 文件是 Android 使用的初始化脚本文件，Android 中有特定的格式以及规则。4、Zygote

### 5.0.4

所有的应用程序进程以及系统服务进程（SystemServer）都是由 Zygote 进程孕育（fork）出来的，zygote 本身是 Native 应用程序，与驱动内核无关。我们知道，Android 系统是基于 Linux 内核的，而在 Linux 系统中，所有的进程都是 init 进程的子孙进程，也就是说，所有的进程都是直接或者间接地由 init 进程 fork 出来的。Zygote 进程也不例外，它是在系统启动的过程，由 init 进程创建的（在系统启动脚本 system/core/rootdir/init.rc 文件中）。在 Java 中，不同的虚拟机实例会为不同的应用分配不同的内存。假如 Android 应用应该尽可能快地启动，但如果 Android 系统为每一个应用启动不同的 Dalvik 虚拟机实例，就会消耗大量的内存以及时间。因此，为了克服这个问题，Android 系统创造了”Zygote”。Zygote 是一个虚拟机进程，预加载以及初始化核心库类，让 Dalvik 虚拟机共享代码、降低内存占用和启动时间。**Zygote** 进程包含两个主要模块：. Socket 服务端，该 Socket 服务端用于接收启动新的 Dalvik 进程命令。. Framework 共享类及共享资源，当 Zygote 进程启动后，会装载一些共享类和资源，共享类是在 preload-classes 文件中定义的，共享资源是在 preload-resources 文件中定义。因为其他 Dalvik 进程是由 Zygote 进程孵化出来的，因此只要 Zygote 装载好了这些类和资源后，新的 Dalvik 进程就不需要在装载这些类和资源了，它们共享 Zygote 进程的资源 and 类。**Zygote** 启动分为两个阶段：. \*\* 虚拟机启动 — 通过 native 启动 \*\*

- startVm(&mJavaVM, &env) 启动虚拟机
- onVmCreated(env) 虚拟机启动后的初始化
- startReg(env) 注册 JNI 函数
- env->CallStaticVoidMethod(startClass, startMeth, strArray) 调用 ZygoteInit 类的 main 函数开创 java 世界

. \*\*SystemServer 进程 — 通过 Java 启动 \*\*

- registerZygoteSocket() 为 zygote 进程注册监听 socket

- preload() 加载常用的 JAVA 类和系统资源
- startSystemServer() 启动 SystemServer 进程
- runSelectLoopMode() 进入循环监听模式
- closeServerSocket() 进程退出时，关闭 socket 监听 5、启动系统服务

### 5.0.5

Zygote 创建新的进程去启动系统服务。你可以在 ZygoteInit 类的” startSystemServer” 方法中找到源代码。核心服务：

- 启动电源管理器；
- 

创建 Activity 管理器； 启动电话注册； 启动包管理器；设置 Activity 管理服务为系统进程；启动上下文管理器；启动系统 Context Providers；启动电池服务；启动定时管理器；启动传感服务；启动窗口管理器；启动蓝牙服务；启动挂载服务。其他服务： 6、引导完成

### 5.0.6

一旦系统服务在内存中跑起来了,Android 就完成了引导过程。在这个时候“ACTION\_BOOT\_COMPLETED”开机启动广播就会发出去。

## 6 Android 异步消息处理机制（Handler、Looper、MessageQueue）源码解析

### 1、Handler 的由来

#### 6.0.1

当程序第一次启动的时候，Android 会同时启动一条主线程（Main Thread）来负责处理与 UI 相关的事件，我们叫做 UI 线程。Android 的 UI 操作并不是线程安全的（出于性能优化考虑），意味着如果多个线程并发操作 UI 线程，可能导致线程安全问题。为了解决 Android 应用多线程问题—Android 平台只允许 UI 线程修改 Activity 里的 UI 组建，就会导致新启动的线程无法改变界面组建的属性值。\*\*简单的说：\*\*当主线程队列处理一个消息超过 5 秒,android 就会抛出一个 ANP(无响应) 的异常, 所以, 我们需要把一些要处理比较长的消息, 放在一个单独线程里面处理, 把处理以后的结果, 返回给主线程运行, 就需要用的 Handler 来进行线程建的通信。

#### 6.0.2

2.1 让线程延时执行主要用到的两个方法：

- final boolean postAtTime(Runnable r, long uptimeMillis)
- final boolean postDelayed(Runnable r, long delayMillis)

2.2 让任务在其他线程中执行并返回结果分为两个步骤：

- 在新启动的线程中发送消息

使用 Handler 对象的 sendMessage() 方法或者 SendEmptyMessage() 方法发送消息。



- 在主线程中获取处理消息

重写 Handler 类中处理消息的方法 (void handleMessage(Message msg))，当新启动的线程发送消息时，消息发送到与之关联的 MessageQueue。而 Handler 不断地从 MessageQueue 中获取并处理消息。3、Handler 更新 UI 线程一般使用

### 6.0.3

- 首先要进行 Handler 申明，复写 handleMessage 方法 (放在主线程中)

```
private Handler handler = new Handler() { @Override public void handleMessage(Message msg) { // TODO 接收消息并且去更新 UI 线程上的控件内容 if (msg.what == UPDATE) { // 更新界面上的 textview tv.setText(String.valueOf(msg.obj)); } super.handleMessage(msg); } };
```

- 子线程发送 Message 给 ui 线程表示自己任务已经执行完成，主线程可以做相应的操作了。

```
new Thread() { @Override public void run() { // TODO 子线程中通过 handler 发送消息给 handler 接收，由 handler 去更新 TextView 的值 try { //do something
```

```
    Message msg = new Message(); msg.what = UPDATE; msg.obj = " 更新后的值" ; handler.sendMessage(msg); } } catch (InterruptedException e) { e.printStackTrace(); } } }.start();
```

4、Handler 原理分析

### 6.0.4

#### 4.1 Handler 的构造函数

- public Handler()

```
public Handler(Callback callback) public Handler(Looper looper) public Handler(Looper looper, Callback callback)
```

- 第 1 个和第 2 个构造函数都没有传递 Looper，这两个构造函数都将通过调用 Looper.myLooper() 获取当前线程绑定的 Looper 对象，然后将该 Looper 对象保存到名为 mLooper 的成员字段中。

下面来看 1 个函数源码：113 public Handler() { 114 this(null, false); 115 } 127 public Handler(Callback callback) { 128 this(callback, false); 129 } //他们会调用 Handler 的内部构造方法 188 public Handler(Callback callback, boolean async) { 189 if (FIND\_POTENTIAL\_LEAKS) { 190 final Class<? extends Handler> klass = getClass(); 191 if ((klass.isAnonymousClass() || klass.isMemberClass()

klass.isLocalClass()) &&

```
192 (klass.getModifiers() & Modifier.STATIC) == 0) { 193 Log.w(TAG, "The following Handler class should be static or leaks might occur: " + 194 klass.getCanonicalName()); 195 } 196 } 197 /***** 198 mLooper = Looper.myLooper(); 199 if (mLooper == null) { 200 throw new RuntimeException( 201 "Can't create handler inside thread that has not called Looper.prepare()"); 202 } 203 mQueue = mLooper.mQueue; 204 mCallback = callback; 205 mAsynchronous = async; 206 }
```

我们看到暗红色的重点部分：通过 Looper.myLooper() 获取了当前线程保存的 Looper 实例，又通过这个 Looper 实例获取了其中保存的 MessageQueue (消息队列)。\*\* 每个 Handler 对应一个 Looper 对象，产生一个 MessageQueue\*\*

- 第 1 个和第 2 个构造函数传递了 Looper 对象，这两个构造函数会将该 Looper 保存到名为 mLooper 的成员字段中。

下面来看 个函数源码: 136 public Handler(Looper looper) { 137 this(looper, null, false); 138 }  
 147 public Handler(Looper looper, Callback callback) { 148 this(looper, callback, false); 149 }  
 //他们会调用 Handler 的内部构造方法 227 public Handler(Looper looper, Callback callback, boolean async) { 228 mLooper = looper; 229 mQueue = looper.mQueue; 230 mCallback = callback; 231 mAsynchronous = async; 232 }

- 第 个和第 个构造函数还传递了 Callback 对象, Callback 是 Handler 中的内部接口, 需要实现其内部的 handleMessage 方法, Callback 代码如下:

80 public interface Callback { 81 public boolean More ... handleMessage(Message msg); 82 }  
 Handler.Callback 是用来处理 Message 的一种手段, 如果没有传递该参数, 那么就应该重写 Handler 的 handleMessage 方法, 也就是说为了使得 Handler 能够处理 Message, 我们有两种办法:

1. 向 Hanlder 的构造函数传入一个 Handler.Callback 对象, 并实现 Handler.Callback 的 handleMessage 方法  
 2. 无需向 Hanlder 的构造函数传入 Handler.Callback 对象, 但是需要重写 Handler 本身的 handleMessage 方法

也就是说无论哪种方式, 我们都得通过某种方式实现 handleMessage 方法, 这点与 Java 中对 Thread 的设计有异曲同工之处。

4.2 Handle 发送消息的几个方法源码  
 public final boolean sendMessage(Message msg) { return sendMessageDelayed(msg, 0); }  
 public final boolean sendEmptyMessageDelayed(int what, long delayMillis) { Message msg = Message.obtain(); msg.what = what; return sendMessageDelayed(msg, delayMillis); }  
 public final boolean sendMessageDelayed(Message msg, long delayMillis) { if (delayMillis < 0) { delayMillis = 0; } return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis); }  
 public boolean sendMessageAtTime(Message msg, long uptimeMillis) { MessageQueue queue = mQueue; if (queue == null) { RuntimeException e = new RuntimeException( this + " sendMessageAtTime() called with no mQueue"); Log.w("Looper", e.getMessage(), e); return false; } return enqueueMessage(queue, msg, uptimeMillis); }  
 我们可以看出他们最后都调用了 sendMessageAtTime(), 然后返回了 enqueueMessage 方法, 下面看一下此方法源码:  
 626 private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {  
 //把当前的 handler 作为 msg 的 target 属性 627 msg.target = this; 628 if (mAsynchronous) { 629 msg.setAsynchronous(true); 630 } 631 return queue.enqueueMessage(msg, uptimeMillis); 632 }  
 在该方法中有两件事需要注意:

1. msg.target = this 该代码将 Message 的 target 绑定为当前的 Handler
2. queue.enqueueMessage

变量 queue 表示的是 Handler 所绑定的消息队列 MessageQueue, 通过调用 queue.enqueueMessage(msg, uptimeMillis) 我们将 Message 放入到消息队列中。过下图可以看到完整的方法调用顺序: [这里写图片描述](<http://img.blog.csdn.net/20160516125245477>)

## 6.0.5

我们一般在主线程申明 Handler, 有时我们需要继承 Thread 类实现自己的线程功能, 当我们在里面申明 Handler 的时候会报错。其原因是主线程中已经实现了两个重要的 Looper 方法, 下面看一看 ActivityThread.java 中 main 方法的源码: public static void main(String[] args) {  
 //.....省略 5205 Looper.prepareMainLooper();//> 5206 5207 ActivityThread thread = new ActivityThread(); 5208 thread.attach(false); 5209 5210 if (sMainThreadHandler == null) { 5211 sMainThreadHandler = thread.getHandler(); 5212 } 5213 5214 AsyncTask.init(); 5215 5216 if (false) { 5217 Looper.myLooper().setMessageLogging(new 5218 LogPrinter(Log.DEBUG, "ActivityThread")); 5219 } 5220 5221 Looper.loop();//> 5222 5223 throw new RuntimeException("Main thread loop unexpectedly exited"); 5224 } 5225 }  
 5.1 首先看 prepare() 方法 70 public static void prepare() { 71 prepare(true); 72 } 73 74 private static void prepare(boolean quitAllowed) {  
 //证了一个线程中只有一个 Looper 实例 75 if (sThreadLocal.get() != null) { 76 throw new RuntimeException("Only one Looper may be created per thread"); 77 } 78 }

sThreadLocal.set(new Looper(quitAllowed)); 79 } 该方法会调用 Looper 构造函数同时实例化出 MessageQueue 和当前 thread. 186 private Looper(boolean quitAllowed) { 187 mQueue = new MessageQueue(quitAllowed); 188 mThread = Thread.currentThread(); 189 } 182 public static MessageQueue myQueue() { 183 return myLooper().mQueue; 184 } prepare() 方法中通过 ThreadLocal 对象实现 Looper 实例与线程的绑定。(不清楚的可以查看 [ThreadLocal 的使用规则和源码分析](<http://blog.csdn.net/amazing7/article/details/51313851>))

### 5.2 loop() 方法

```
109 public static void loop() { 110 final Looper me = myLooper(); 111 if (me == null) { 112 throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread."); 113 } 114 final MessageQueue queue = me.mQueue; 115 118 Binder.clearCallingIdentity(); 119 final long ident = Binder.clearCallingIdentity(); 120 121 for (;;) { 122 Message msg = queue.next(); // might block 123 if (msg == null) { 124 125 return; 126 } 127 129 Printer logging = me.mLogging; 130 if (logging != null) { 131 logging.println("«»»> Dispatching to " + msg.target + " " + 132 msg.callback + ": " + msg.what); 133 } //重点 **** 135 msg.target.dispatchMessage(msg); 136 137 if (logging != null) { 138 logging.println("«««< Finished to " + msg.target + " " + msg.callback); 139 } 140 142 // identity of the thread wasn't corrupted. 143 final long newIdent = Binder.clearCallingIdentity(); 144 if (ident != newIdent) { 145 Log.wtf(TAG, "Thread identity changed from 0x" + Long.toHexString(ident) + " to 0x" + Long.toHexString(newIdent) + " while dispatching to " + msg.target.getClass().getName() + " " + msg.callback + " what=" + msg.what); 150 } 151 152 msg.recycleUnchecked(); 153 } 154 }
```

首先 looper 对象不能为空，就是说 loop() 方法调用必须在 prepare() 方法的后面。Looper 一直在不断的从消息队列中通过 MessageQueue 的 next 方法获取 Message，然后通过代码 msg.target.dispatchMessage(msg) 让该 msg 所绑定的 Handler (Message.target) 执行 dispatchMessage 方法以实现对 Message 的处理。Handler 的 dispatchMessage 的源码如下：93 public void dispatchMessage(Message msg) { 94 if (msg.callback != null) { 95 handleCallback(msg); 96 } else { 97 if (mCallback != null) { 98 if (mCallback.handleMessage(msg)) { 99 return; 100 } 101 } 102 handleMessage(msg); 103 } 104 }

我们可以看到 Handler 提供了三种途径处理 Message，而且处理有前后优先级之分：首先尝试让 postXXX 中传递的 Runnable 执行，其次尝试让 Handler 构造函数中传入的 Callback 的 handleMessage 方法处理，最后才是让 Handler 自身的 handleMessage 方法处理 Message。

### 6、如何在子线程中使用 Handler

## 6.0.6

Handler 本质是从当前的线程中获取到 Looper 来监听和操作 MessageQueue，当其他线程执行完成后回调当前线程。子线程需要先 prepare() 才能获取到 Looper，是因为在子线程只是一个普通的线程，其 ThreadLocal 中没有设置过 Looper，所以会抛出异常，而在 Looper 的 prepare() 方法中 sThreadLocal.set(new Looper()) 是设置了 Looper 的。

### 6.1 实例代码

定义一个类实现 Runnable 接口或继承 Thread 类（一般不继承）。class Rub implements Runnable {

```
public Handler myHandler; // 实现 Runnable 接口的线程体 @Override public void run() {
```

、调用 **Looper** 的 **prepare()** 方法为当前线程创建 **Looper** 对象并，创建 **Looper** 对象时，它的构造器会自动的创建相对应的 **MessageQueue** Looper.prepare();

、创建 **Handler** 子类的实例，重写 **handleMessage()** 方法，该方法处理除当前线程以外线程的消息 myHandler = new Handler() { @Override public void handleMessage(Message msg) { String ms = ""; if (msg.what == 0x777) {

```
} }
```

}; //、调用 Looper 的 loop() 方法来启动 Looper 让消息队列转动起来 Looper.loop(); } } 注意分成三步： 1. 调用 Looper 的 prepare() 方法为当前线程创建 Looper 对象，创建 Looper 对象时，它的构造器会创建与之配套的 MessageQueue。 2. 有了 Looper 之后，创建 Handler 子类实例，重写 HandlerMessage() 方法，该方法负责处理来自于其他线程的消息。 3. 调用 Looper 的 loop() 方法启动 Looper。然后使用这个 handler 实例在任何其他线程中发送消息，最终处理消息的代码都会在你创建 Handler 实例的线程中运行。

### 7、总结

## 6.0.7

**\*\*Handler\*\*:** 发送消息, 它能把消息发送给 Looper 管理的 MessageQueue。  
处理消息, 并负责处理 Looper 分给它的消息。**\*\*Message\*\*:** Handler 接收和处理的  
消息对象。**\*\*Looper\*\*:** 每个线程只有一个 Looper, 它负责管理对应的 MessageQueue,  
会不断地从 MessageQueue 取出消息, 并将消息分给对应的 Handler 处理。  
主线程中, 系统已经初始化了一个 Looper 对象, 因此可以直接创建 Handler 即可, 就可以通过  
Handler 来发送消息、处理消息。程序自己启动的子线程, 程序必须自己创建一个 Looper 对象, 并  
启动它, 调用 Looper.prepare() 方法。prapare() 方法: 保证每个线程最多只有一个 Looper 对象。  
looper() 方法: 启动 Looper, 使用一个死循环不断取出 MessageQueue 中的消息, 并将取出的消  
息分给对应的 Handler 进行处理。 MessageQueue: 由 Looper 负责管理, 它采用先进先出的方  
式来管理 Message。 Handler 的构造方法, 会首先得到当前线程中保存的 Looper 实例, 进而与  
Looper 实例中的 MessageQueue 想关联。 Handler 的 sendMessage 方法, 会给 msg 的 target  
赋值为 handler 自身, 然后加入 MessageQueue 中。

## 7 Android 数据存储五种方式使用与总结

### 1、概述

#### 7.0.1

Android 提供了 5 种方式来让用户保存持久化应用程序数据。根据自己的需求来做选择,  
比如数据是否是应用程序私有的, 是否能被其他程序访问, 需要多少数据存储空间等, 分别是:

使用 SharedPreferences 存储数据    文件存储数据    SQLite 数据库存储数据    使用  
ContentProvider 存储数据    网络存储数据    Android 提供了一种方式来暴露你的数据 (甚至是私  
有数据) 给其他应用程序 - ContentProvider。它是一个可选组件, 可公开读写你应用程序数据。2、  
SharedPreferences 存储

#### 7.0.2

SharedPreferences 类提供了一个总体框架, 使您可以保存和检索的任何基本数据类型 (boolean,  
float, int, long, string) 的持久键-值对 (基于 XML 文件存储的 “key-value” 键值对数据)。通常  
用来存储程序的一些配置信息。其存储在 “data/data/程序包名/shared\_prefs 目录下。xml 处理时  
Dalvik 会通过自带底层的本地 XML Parser 解析, 比如 XMLpull 方式, 这样对于内存资源占用比较  
好。 **2.1** 我们可以通过以下两种方法获取 SharedPreferences 对象 (通过 Context):

- getSharedPreferences (String name, int mode)

当我们有多个 SharedPreferences 的时候, 根据第一个参数 name 获得相应的 SharedPreferences  
对象。 getPreferences (int mode) 如果你的 Activity 中只需要一个 SharedPreferences 的时候  
使用。这里的 mode 有四个选项: Context.MODE\_PRIVATE 该 SharedPreferences 数据只能被本  
应用程序读、写。Context.MODE\_WORLD\_READABLE 该 SharedPreferences 数据能被其他应用  
程序读, 但不能写。Context.MODE\_WORLD\_WRITEABLE 该 SharedPreferences 数据能被其他应  
用程序读和写。Context.MODE\_MULTI\_PROCESS sdk2.3 后添加的选项, 当多个进程同时读写同  
一个 SharedPreferences 时它会检查文件是否修改。 **2.2** 向 Shared Preferences 中 \*\* 写入值 \*\*

首先要通过 SharedPreferences.Editor 获取到 Editor 对象; 然后通过 Editor 的 putBoolean() 或  
putString() 等方法存入值; 最后调用 Editor 的 commit() 方法提交; //Use 0 or MODE\_PRIVATE  
for the default operation SharedPreferences settings = getSharedPreferences("fanrunqi", 0);  
SharedPreferences.Editor editor = settings.edit(); editor.putBoolean("isAmazing", true); // 提  
交本次编辑 editor.commit(); 同时 Edit 还有两个常用的方法:

- editor.remove(String key) : 下一次 commit 的时候会移除 key 对应的键值对

editor.clear(): 移除所有键值对

**2.3 从 Shared Preferences 中 \*\* 读取值 \*\*** 读取值使用 Shared-Preference 对象的 getBoolean() 或 getString() 等方法就行了 (没 Editor 啥子事)。SharedPreferences settings = getSharedPreferences("fanrunqi", 0); boolean isAmazing = settings.getBoolean("isAmazing");

**2.4 Shared Preferences 的优缺点** 看起来 Preferences 是很轻量级的应用, 使用起来也很方便, 简洁。但存储数据类型比较单一 (只有基本数据类型), 无法进行条件查询, 只能在不复杂的存储需求下使用, 比如保存配置信息等。

### 3、文件数据存储

## 7.0.3

1. 3.1 使用内部存储 当文件被保存在内部存储中时, 默认情况下, 文件是应用程序私有的, 其他应用不能访问。当用户卸载应用程序时这些文件也跟着被删除。文件默认存储位置: /data/data/包名/files/文件名。

2. 3.1.1 创建和写入一个内部存储的私有文件: 调用 Context 的 openFileOutput() 函数, 填入文件名和操作模式, 它会返回一个 FileOutputStream 对象。通过 FileOutputStream 对象的 write() 函数写入数据。FileOutputStream 对象的 close() 函数关闭流。例如: String FILENAME = "a.txt"; String string = "fanrunqi"; try { FileOutputStream fos = openFileOutput(FILENAME, Context.MODE\_PRIVATE); fos.write(string.getBytes()); fos.close(); } catch (Exception e) { e.printStackTrace(); } 在 openFileOutput(String name, int mode) 方法中

- name 参数: 用于指定文件名称, 不能包含路径分隔符 “/”, 如果文件不存在, Android 会自动创建它。
- mode 参数: 用于指定操作模式, 分为四种:
- Context.MODE\_PRIVATE = 0

为默认操作模式, 代表该文件是私有数据, 只能被应用本身访问, 在该模式下, 写入的内容会覆盖原文件的内容。

- Context.MODE\_APPEND = 32768

该模式会检查文件是否存在, 存在就往文件追加内容, 否则就创建新文件。

- Context.MODE\_WORLD\_READABLE = 1

表示当前文件可以被其他应用读取。

- MODE\_WORLD\_WRITEABLE

表示当前文件可以被其他应用写入。

3. 3.1.2 读取一个内部存储的私有文件: 调用 openFileInput(), 参数中填入文件名, 会返回一个 FileInputStream 对象。使用流对象的 read() 方法读取字节 调用流的 close() 方法关闭流例如: String FILENAME = "a.txt"; try { FileInputStream inStream = openFileInput(FILENAME); int len = 0; byte[] buf = new byte<sup>1</sup>; StringBuilder sb = new StringBuilder(); while ((len = inStream.read(buf)) != -1) { sb.append(new String(buf, 0, len)); } inStream.close(); } catch (Exception e) { e.printStackTrace(); } 其他一些经常用到的方法:

- getFilesDir(): 得到内存储文件的绝对路径
- getDir(): 在内存储空间中 \*\* 创建 \*\* 或 \*\* 打开一个已经存在 \*\* 的目录

<sup>1</sup>DEFINITION NOT FOUND.

- `deleteFile()`: 删除保存在内部存储的文件。
  - `fileList()`: 返回当前由应用程序保存的文件的数组（内存存储目录下的全部文件）。
4. 3.1. 3 保存编译时的静态文件 如果你想在应用编译时保存静态文件，应该把文件保存在项目的 `**res/raw/**` 目录下，你可以通过 `openRawResource()` 方法去打开它（传入参数 `R.raw.filename`），这个方法返回一个 `InputStream` 流对象你可以读取文件但是不能修改原始文件。`InputStream is = this.getResources().openRawResource(R.raw.filename);`
5. 3.1. 4 保存内存缓存文件 有时候我们只想缓存一些数据而不是持久化保存，可以使用 `getCacheDir()` 去打开一个文件，文件的存储目录（`/data/data/包名/cache`）是一个应用专门来保存临时缓存文件的内存目录。当设备的内部存储空间比较低的时候，Android 可能会删除这些缓存文件来恢复空间，但是你不应该依赖系统来回收，要自己维护这些缓存文件把它们的大小限制在一个合理的范围内，比如 1 MB。当你卸载应用的时候这些缓存文件也会被移除。

### 7.0.4 3. 2 使用外部存储（sdcard）

因为内部存储容量限制，有时候需要存储数据比较大的时候需要用到外部存储，使用外部存储分为以下几个步骤：

- 3.2.1 添加外部存储访问权限 首先，要在 `AndroidManifest.xml` 中加入访问 `SDCard` 的权限，如下：  
`<!-- 在 SDCard 中创建与删除文件权限 --> <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />`  
`<!-- 往 SDCard 写入数据权限 --> <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />`
- 3.2.2 检测外部存储的可用性 在使用外部存储时我们需要检测其状态，它可能被连接到计算机、丢失或者只读等。下面代码将说明如何检查状态：`//获取外存储的状态 String state = Environment.getExternalStorageState(); if (Environment.MEDIA_MOUNTED.equals(state)) { // 可读可写 mExternalStorageAvailable = mExternalStorageWriteable = true; } else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) { // 可读 } else { // 可能有很多其他的状态，但是我们只需要知道，不能读也不能写 }`
- 3.2.3 访问外部存储器中的文件 **\*\* 1、如果 API 版本大于或等于 8 \*\***，使用

- `getExternalFilesDir (String type)`

该方法打开一个外存储目录，此方法需要一个类型，指定你想要的子目录，如类型参数 `DIRECTORY_MUSIC` 和 `DIRECTORY_RINGTONES`（传 `null` 就是你应用程序的文件目录的根目录）。通过指定目录的类型，确保 Android 的媒体扫描仪将扫描分类系统中的文件（例如，铃声被确定为铃声）。如果用户卸载应用程序，这个目录及其所有内容将被删除。例如：`File file = new File(getExternalFilesDir(null), "fanrunqi.jpg");` 2、如果 **API 版本小于 8**（7 或者更低）

- `getExternalStorageDirectory ()`

通过该方法打开外存储的根目录，你应该在以下目录下写入你的应用数据，这样当卸载应用程序时该目录及其所有内容也将被删除。`Android/data/<package_name>/files` 读写数据：`if (Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {`

`File sdCardDir = Environment.getExternalStorageDirectory();//获取 SDCard 目录"/sdcard"`  
`File saveFile = new File(sdCardDir,"a.txt");`

`//写数据 try { FileOutputStream fos= new FileOutputStream(saveFile); fos.write("fanrunqi".getBytes()); fos.close(); } catch (Exception e) { e.printStackTrace(); }`

`//读数据 try { FileInputStream fis= new FileInputStream(saveFile); int len =0; byte[] buf = new byte[1024]; StringBuffer sb = new StringBuffer(); while((len=fis.read(buf))!=-1){ sb.append(new String(buf, 0, len)); } fis.close(); } catch (Exception e) { e.printStackTrace(); } }` 我们也可以在 `/Android/data/package_name/cache/` 目录下做外部缓存。部分翻译于：

[android-data-storage](<http://www.android-doc.com/guide/topics/data/data-storage.html>) 4、网络存储数据



## 7.0.5

### 7.0.6 HttpURLConnection

HttpURLConnection 是 Java.net 包中提供的 API，我们知道 Android SDK 是基于 Java 的，所以当然优先考虑 HttpURLConnection 这种最原始最基本的 API，其实大多数开源的联网框架基本上也是基于 JDK 的 HttpURLConnection 进行的封装罢了，掌握 HttpURLConnection 需要以下几个步骤：1、将访问的路径转换成 URL。

- URL url = new URL(path);

2、通过 URL 获取连接。

- HttpURLConnection conn = (HttpURLConnection) url.openConnection();

3、设置请求方式。

- conn.setRequestMethod(GET);

4、设置连接超时时间。

conn.setConnectTimeout(5000); 5、设置请求头的信息。

- conn.setRequestProperty(User-Agent, Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0));

7、针对不同的响应码，做不同的操作（请求码 200，表明请求成功，获取返回内容的输入流）工具类：public class StreamTools { /\*\*

- 将输入流转换成字符串
- 
- @param is
- 从网络获取的输入流
- @return

```
*/ public static String streamToString(InputStream is) { try { ByteArrayOutputStream baos =
new ByteArrayOutputStream(); byte[] buffer = new byte1; int len = 0; while ((len = is.read(buffer))
!= -1) { baos.write(buffer, 0, len); } baos.close(); is.close(); byte[] byteArray = baos.toByteArray();
return new String(byteArray); } catch (Exception e) { Log.e(tag, e.toString()); return null; } }
}
```

1. HttpURLConnection 发送 GET 请求 public static String loginByGet(String username, String password) { String path = <http://192.168.0.107:8080/WebTest/LoginServlet?username=> + username + &password= + password; try { URL url = new URL(path); HttpURLConnection conn = (HttpURLConnection) url.openConnection(); conn.setConnectTimeout(5000); conn.setRequestMethod(GET); int code = conn.getResponseCode(); if (code == 200) { InputStream is = conn.getInputStream(); // 字节流转换成字符串 return StreamTools.streamToString(is); } else { return 网络访问失败; } } catch (Exception e) { e.printStackTrace(); return 网络访问失败; } }
2. HttpURLConnection 发送 POST 请求 public static String loginByPost(String username, String password) { String path = <http://192.168.0.107:8080/WebTest/LoginServlet>; try { URL url = new URL(path); HttpURLConnection conn = (HttpURLConnection) url.openConnection();

```
conn.setConnectTimeout(5000); conn.setRequestMethod(POST); conn.setRequestProperty(Content-Type, application/x-www-form-urlencoded); String data = username + "&password=" + password; conn.setRequestProperty(Content-Length, data.length() + ); // POST 方式, 其实就是浏览器把数据写给服务器 conn.setDoOutput(true); // 设置可输出流 OutputStream os = conn.getOutputStream(); // 获取输出流 os.write(data.getBytes()); // 将数据写给服务器 int code = conn.getResponseCode(); if (code == 200) { InputStream is = conn.getInputStream(); return StreamTools.streamToString(is); } else { return 网络访问失败; } } catch (Exception e) { e.printStackTrace(); return 网络访问失败; } }
```

## 7.0.7 HttpClient

HttpClient 是开源组织 Apache 提供的 Java 请求网络框架, 其最早是为了方便 Java 服务器开发而诞生的, 是对 JDK 中的 HttpURLConnection 各 API 进行了封装和简化, 提高了性能并且降低了调用 API 的繁琐, Android 因此也引进了这个联网框架, 我们不再需要导入任何 jar 或者类库就可以直接使用, 值得注意的是 Android 官方已经宣布不建议使用 HttpClient 了。

1. HttpClient 发送 GET 请求 1、创建 HttpClient 对象 2、创建HttpGet 对象, 指定请求地址 (带参数) 3、使用 HttpClient 的 execute(), 方法执行 HttpGet 请求, 得到 HttpResponse 对象 4、调用 HttpResponse 的 getStatusLine().getStatusCode() 方法得到响应码 5、调用的 HttpResponse 的 getEntity().getContent() 得到输入流, 获取服务端写回的数据 public static String loginByHttpClientGet(String username, String password) { String path = <http://192.168.0.107:8080/WebTest/LoginServlet?username=>

- username + "&password=" + password;

```
HttpClient client = new DefaultHttpClient(); // 开启网络访问客户端 HttpGet httpGet = new HttpGet(path); // 包装一个 GET 请求 try { HttpResponse response = client.execute(httpGet); // 客户端执行请求 int code = response.getStatusLine().getStatusCode(); // 获取响应码 if (code == 200) { InputStream is = response.getEntity().getContent(); // 获取实体内容 String result = StreamTools.streamToString(is); // 字节流转字符串 return result; } else { return 网络访问失败; } } catch (Exception e) { e.printStackTrace(); return 网络访问失败; } }
```

2. HttpClient 发送 POST 请求 1, 创建 HttpClient 对象 2, 创建HttpPost 对象, 指定请求地址 3, 创建 List, 用来装载参数 4, 调用 HttpPost 对象的 setEntity() 方法, 装入一个 UrlEncodedFormEntity 对象, 携带之前封装好的参数 5, 使用 HttpClient 的 execute() 方法执行 HttpPost 请求, 得到 HttpResponse 对象 6, 调用 HttpResponse 的 getStatusLine().getStatusCode() 方法得到响应码 7, 调用的 HttpResponse 的 getEntity().getContent() 得到输入流, 获取服务端写回的数据 public static String loginByHttpClientPOST(String username, String password) { String path = <http://192.168.0.107:8080/WebTest/LoginServlet>; try { HttpClient client = new DefaultHttpClient(); // 建立一个客户端 HttpPost httpPost = new HttpPost(path); // 包装 POST 请求 // 设置发送的实体参数 List parameters = new ArrayList(); parameters.add(new BasicNameValuePair(username, username)); parameters.add(new BasicNameValuePair(password, password)); httpPost.setEntity(new UrlEncodedFormEntity(parameters, UTF-8)); HttpResponse response = client.execute(httpPost); // 执行 POST 请求 int code = response.getStatusLine().getStatusCode(); if (code == 200) { InputStream is = response.getEntity().getContent(); String result = StreamTools.streamToString(is); return result; } else { return 网络访问失败; } } catch (Exception e) { e.printStackTrace(); return 访问网络失败; } } “ 参考: [Android 开发请求网络方式详解](<http://www.2cto.com/kf/201501/368943.html>)



## 7.0.8 Android 提供的其他网络访问框架

HttpClient 和 HttpURLConnection 的两种网络访问方式编写网络代码，需要自己考虑很多，获取数据或许可以，但是如果要手机本地数据上传至网络，根据不同的 web 端接口，需要组织不同的数据内容上传，给手机端造成了很大的工作量。目前有几种快捷的网络开发开源框架，给我们提供了非常大的便利。下面是这些项目 Github 地址，有文档和 Api 说明。[android-async-http](https://github.com/loopj/android-async-http) [http-request](https://github.com/kevinsawicki/http-request) [okhttp](https://github.com/square/okhttp) 5、SQLite 数据库存储数据

## 7.0.9

前面的文章 [SQLite 的使用入门](http://blog.csdn.net/amazing7/article/details/51375012) 已经做了详细说明，这里就不在多说了。6、使用 ContentProvider 存储数据

## 7.0.10

同样可以查看 [ContentProvider 实例详解](http://blog.csdn.net/amazing7/article/details/51324022)

# 8 Android 缓存机制

移动开发本质上就是手机和服务器之间进行通信，需要从服务端获取数据。反复通过网络获取数据是比较耗时的，特别是访问比较多的时候，会极大影响了性能，Android 中可通过缓存机制来减少频繁的网络操作，减少流量、提升性能。

## 8.1 实现原理

把不需要实时更新的数据缓存下来，通过时间或者其他因素 来判别是读缓存还是网络请求，这样可以缓解服务器压力，一定程度上提高应用响应速度，并且支持离线阅读。

## 8.2 Bitmap 的缓存

在许多的情况下 (像 ListView, GridView 或 ViewPager 之类的组件) 我们需要一次性加载大量的图片，在屏幕上显示的图片 and 所有待显示的图片有可能需要马上就在屏幕上无限制的进行滚动、切换。像 ListView, GridView 这类组件，它们的子项当不可见时，所占用的内存会被回收以供正在前台显示子项使用。垃圾回收器也会释放你已经加载了的图片占用的内存。如果你想让你的 UI 运行流畅的话，就不应该每次显示时都去重新加载图片。保持一些内存和文件缓存就变得很有必要了。

### 8.2.1 使用内存缓存

通过预先消耗应用的一点内存来存储数据，便可快速的为应用中的组件提供数据，是一种典型的以 \*\* 空间换时间 \*\* 的策略。LruCache 类 (Android v4 Support Library 类库中开始提供) 非常适合来做图片缓存任务，它可以使用一个 LinkedHashMap 的强引用来保存最近使用的对象，并且当它保存的对象占用的内存总和超出了为它设计的最大内存时会把 \*\* 不经常使用 \*\* 的对象成员踢出以供垃圾回收器回收。给 LruCache 设置一个合适的内存大小，需考虑如下因素：

- 还剩余多少内存给你的 activity 或应用使用
- 屏幕上需要一次性显示多少张图片 and 多少图片在等待显示
- 手机的大小和密度是多少 (密度越高的设备需要越大的缓存)
- 图片的尺寸 (决定了所占用的内存大小)

- 图片的访问频率（频率高的在内存中一直保存）
- 保存图片的质量（不同像素的在不同情况下显示）

具体的要根据应用图片使用的具体情况来找到一个合适的解决办法，一个设置 LruCache 例子：

```
private LruCache<String, Bitmap> mMemoryCache; @Override protected void onCreate(Bundle savedInstanceState) { ... // 获得虚拟机能提供的最大内存，超过这个大小会抛出 Out-Of-Memory 的异常 final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024); // 用 1 / 8 的内存大小作为内存缓存 final int cacheSize = maxMemory / 8; mMemoryCache = new LruCache<String, Bitmap>(cacheSize) { @Override protected int sizeOf(String key, Bitmap bitmap) { // 这里返回的不是 item 的个数，是 cache 的 size（单位 1024 个字节） return bitmap.getByteCount() / 1024; } }; ... } public void addBitmapToMemoryCache(String key, Bitmap bitmap) { if (getBitmapFromMemCache(key) == null) { mMemoryCache.put(key, bitmap); } } public Bitmap getBitmapFromMemCache(String key) { return mMemoryCache.get(key); } 当为 ImageView 加载一张图片时，会先在 LruCache 中看看有没有缓存这张图片，如果有的话直接更新到 ImageView 中，如果没有的话，一个后台线程会被触发来加载这张图片。 public void loadBitmap(int resId, ImageView imageView) { final String imageKey = String.valueOf(resId); // 查看下内存缓存中是否缓存了这张图片 final Bitmap bitmap = getBitmapFromMemCache(imageKey); if (bitmap != null) { mImageView.setImageBitmap(bitmap); } else { mImageView.setImageResource(R.drawable.sample3); } BitmapWorkerTask task = new BitmapWorkerTask(mImageView); task.execute(resId); } } 在图片加载的 Task 中，需要把加载好的图片加入到内存缓存中。 class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> { ... // 在后台完成 @Override protected Bitmap doInBackground(Integer... params) { final Bitmap bitmap = decodeSampledBitmapFromResource(getResources(), params[0], 100, 100); addBitmapToMemoryCache(String.valueOf(params[0]), bitmap); return bitmap; } ... }
```

## 8.2.2 使用磁盘缓存

内存缓存能够快速的获取到最近显示的图片，但不一定就能够获取到。当数据集过大时很容易把内存缓存填满（如 GridView）。你的应用也有可能被其它的任务（比如来电）中断进入到后台，后台应用有可能会被杀死，那么相应的内存缓存对象也会被销毁。当你的应用重新回到前台显示时，你的应用又需要一张一张的去加载图片了。磁盘文件缓存能够用来处理这些情况，保存处理好的图片，当内存缓存不可用的时候，直接读取在硬盘中保存好的图片，这样可以有效的减少图片加载的次数。读取磁盘文件要比直接从内存缓存中读取要慢一些，而且需要在一个 UI 主线程外的线程中进行，因为磁盘的读取速度是不能够保证的，磁盘文件缓存显然也是一种以 \*\*空间换时间\*\* 的策略。如果图片使用非常频繁的话，一个 ContentProvider 可能更适合代替去存储缓存图片，比如图片 gallery 应用。下面是一个 DiskLruCache 的部分代码：

```
private DiskLruCache mDiskLruCache; private final Object mDiskCacheLock = new Object(); private boolean mDiskCacheStarting = true; private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB private static final String DISK_CACHE_SUBDIR = "thumbnails"; @Override protected void onCreate(Bundle savedInstanceState) { ... // 初始化内存缓存 ... // 在后台线程中初始化磁盘缓存 File cacheDir = getDiskCacheDir(this, DISK_CACHE_SUBDIR); new InitDiskCacheTask().execute(cacheDir); ... } class InitDiskCacheTask extends AsyncTask<File, Void, Void> { @Override protected Void doInBackground(File... params) { synchronized (mDiskCacheLock) { File cacheDir = params[0]; mDiskLruCache = DiskLruCache.open(cacheDir, DISK_CACHE_SIZE); mDiskCacheStarting = false; // 结束初始化 mDiskCacheLock.notifyAll(); // 唤醒等待线程 } return null; } } class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> { ... // 在后台解析图片 @Override protected Bitmap doInBackground(Integer... params) { final String imageKey = String.valueOf(params[0]); // 在后台线程中检测磁盘缓存 Bitmap bitmap = getBitmapFromDiskCache(imageKey); ... }
```

<sup>2</sup>DEFINITION NOT FOUND.

```

if (bitmap == null) { // 没有在磁盘缓存中找到图片 final Bitmap bitmap = decodeSampled-
BitmapFromResource( getResources(), params2, 100, 100)); } // 把这个 final 类型的 bitmap 加
到缓存中 addBitmapToCache(imageKey, bitmap); return bitmap; } ... } public void addBitmap-
ToCache(String key, Bitmap bitmap) { // 先加到内存缓存 if (getBitmapFromMemCache(key) ==
null) { mMemoryCache.put(key, bitmap); } // 再加到磁盘缓存 synchronized (mDiskCacheLock)
{ if (mDiskLruCache != null && mDiskLruCache.get(key) == null) { mDiskLruCache.put(key,
bitmap); } } } public Bitmap getBitmapFromDiskCache(String key) { synchronized (mDiskCach-
eLock) { // 等待磁盘缓存从后台线程打开 while (mDiskCacheStarting) { try { mDiskCache-
Lock.wait(); } catch (InterruptedException e) { } } if (mDiskLruCache != null) { return mDiskL-
ruCache.get(key); } } return null; } public static File getDiskCacheDir(Context context, String
uniqueName) { // 优先使用外缓存路径, 如果没有挂载外存储, 就使用内缓存路径 final String
cachePath = Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
|| !isExternalStorageRemovable() ? getExternalCacheDir(context).getPath(): context.getCacheDir().getP-
ath(); return new File(cachePath + File.separator + uniqueName); } 不能在 UI 主线程中进行这项操-
作, 因为初始化磁盘缓存也需要对磁盘进行操作。上面的程序片段中, 一个锁对象确保了磁盘缓存
没有初始化完成之前不能够对磁盘缓存进行访问。内存缓存在 UI 线程中进行检测, 磁盘缓存在 UI
主线程外的线程中进行检测, 当图片处理完成之后, 分别存储到内存缓存和磁盘缓存中。

```

### 8.2.3 设备配置参数改变时加载问题

由于应用在运行的时候设备配置参数可能会发生改变, 比如设备朝向改变, 会导致 Android 销毁你的 Activity 然后按照新的配置重启, 这种情况下, 我们要避免重新去加载处理所有的图片, 让用户能有一个流畅的体验。使用 Fragment 能够把内存缓存对象传递到新的 activity 实例中, 调用 `setRetainInstance(true)` 方法来保留 Fragment 实例。当 activity 重新创建好后, 被保留的 Fragment 依附于 activity 而存在, 通过 Fragment 就可以获取到已经存在的内存缓存对象了, 这样就可以快速的获取到图片, 并设置到 ImageView 上, 给用户一个流畅的体验。下面是一个示例程序片段: `private LruCache<String, Bitmap> mMemoryCache; @Override protected void onCreate(Bundle savedInstanceState) { ... RetainFragment mRetainFragment = RetainFragment.findOrCreateRetainFragment(getFragmentManager()); mMemoryCache = RetainFragment.mRetainFragment.mMemoryCache; if (mMemoryCache == null) { mMemoryCache = new LruCache<String, Bitmap>(cacheSize) { ... //像上面例子中那样初始化缓存 } mRetainFragment.mRetainedCache = mMemoryCache; } ... } class RetainFragment extends Fragment { private static final String TAG = "RetainFragment"; public LruCache<String, Bitmap> mRetainedCache; public RetainFragment() {} public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) { RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG); if (fragment == null) { fragment = new RetainFragment(); } return fragment; } @Override public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); // 使得 Fragment 在 Activity 销毁后还能够保留下来 setRetainInstance(true); } } 可以在不适用 Fragment (没有界面的服务类 Fragment) 的情况下旋转设备屏幕。在保留缓存的情况下, 你应该能发现填充图片到 Activity 中几乎是瞬间从内存中取出而没有任何延迟的感觉。任何图片优先从内存缓存获取, 没有的话再到硬盘缓存中找, 如果都没有, 那就以普通方式加载图片。参考: [Caching Bitmaps](http://developer.android.com/training/displaying-bitmaps/cache-bitmap.html) [LruCache](http://developer.android.com/reference/android/util/LruCache.html)`

## 8.3 使用 SQLite 进行缓存

网络请求数据完成后, 把文件的相关信息 (如 url (一般作为唯一标示), 下载时间, 过期时间) 等存放到数据库。下次加载的时候根据 url 先从数据库中查询, 如果查询到并且时间未过期, 就根据路径读取本地文件, 从而实现缓存的效果。注意: 缓存的数据库是存放在 `/data/data/<package>/databases/` 目录下, 是占用内存空间的, 如果缓存累计, 容易浪费内存, 需要及时清理缓存。

## 8.4 文件缓存

思路和一般缓存一样，把需要的数据存储在文件中，下次加载时判断文件是否存在和过期（使用 `File.lastModified()` 方法得到文件的最后修改时间，与当前时间判断），存在并未过期就加载文件中的数据，否则请求服务器重新下载。注意，无网络环境下就默认读取文件缓存中的。

## 9 Android 异步任务机制之 AsyncTask

- 在 Android 中实现异步任务机制有两种方式，\*\*Handler\*\* 和 \*\*AsyncTask\*\*。
- 
- Handler 已经在上一篇文章 [异步消息处理机制（Handler、Looper、MessageQueue）源码解析](<http://blog.csdn.net/amazing7/article/details/51424038#reply>) 说过了。
- 
- 本篇就说说 AsyncTask 的异步实现。1、什么时候使用 AsyncTask

### 9.0.1

在上一篇文章已经说了，主线程主要负责控制 UI 页面的显示、更新、交互等。为了有更好的用户体验，UI 线程中的操作要求越短越好。我们把耗时的操作（例如网络请求、数据库操作、复杂计算）放到单独的子线程中操作，以避免主线程的阻塞。但是在子线程中不能更新 UI 界面，这时候需要使用 handler。但如果耗时的操作太多，那么我们需要开启太多的子线程，这就会给系统带来巨大的负担，随之也会带来性能方面的问题。在这种情况下我们就可以考虑使用类 AsyncTask 来异步执行任务，不需要子线程和 handler，就可以完成异步操作和刷新 UI。不要随意使用 AsyncTask，除非你必须要与 UI 线程交互。默认情况下使用 Thread 即可，要注意需要将线程优先级调低。AsyncTask 适合处理短时间的操作，长时间的操作，比如下载一个很大的视频，这就需要你使用自己的线程来下载，不管是断点下载还是其它的。2、AsyncTask 原理

### 9.0.2

AsyncTask 主要有二个部分：一个是与主线程的交互，另一个就是线程的管理调度。虽然可能多个 AsyncTask 的子类的实例，但是 AsyncTask 的内部 Handler 和 ThreadPoolExecutor 都是进程范围内共享的，其都是 static 的，也即属于类的，类的属性的作用范围是 CLASSPATH，因为一个进程一个 VM，所以是 AsyncTask 控制着进程范围内所有的子类实例。AsyncTask 内部会创建一个进程作用域的线程池来管理要运行的任务，也就就是说当你调用了 AsyncTask 的 execute() 方法后，AsyncTask 会把任务交给线程池，由线程池来管理创建 Thread 和运行 Therad。3、AsyncTask 介绍

### 9.0.3

Android 的 AsyncTask 比 Handler 更轻量级一些（只是代码上轻量一些，而实际上要比 handler 更耗资源），适用于简单的异步处理。Android 之所以有 Handler 和 AsyncTask，都是为了不阻塞主线程（UI 线程），因为 UI 的更新只能在主线程中完成，因此异步处理是不可避免的。AsyncTask：对线程间的通讯做了包装，是后台线程和 UI 线程可以简易通讯：后台线程执行异步任务，将 result 告知 UI 线程。使用 AsyncTask 分为两步：继承 AsyncTask 类实现自己的类 `public abstract class AsyncTask<Params, Progress, Result> {`

- Params: 输入参数，对应 excute() 方法中传递的参数。如果不需要传递参数，则直接设为 void 即可。

- 
- **Progress:** 后台任务执行的百分比
- 
- **Result:** 返回值类型, 和 `doInBackground()` 方法的返回值类型保持一致。

复写方法最少要重写以下这两个方法:

- `doInBackground(Params...)`

在 **\*\* 子线程 \*\*** (其他方法都在主线程执行) 中执行比较耗时的操作, 不能更新 UI, 可以在该方法中调用 `publishProgress(Progress...)` 来更新任务的进度。`Progress` 方法是 `AsyncTask` 中一个 `final` 方法只能调用不能重写。

- `onPostExecute(Result)`

使用在 `doInBackground` 得到的结果处理操作 UI, 在主线程执行, 任务执行的结果作为此方法的参数返回。有时根据需求还要实现以下三个方法:

- `onProgressUpdate(Progress...)`

可以使用进度条增加用户体验度。此方法在主线程执行, 用于显示任务执行的进度。

- `onPreExecute()`

这里是最终用户调用 `Excute` 时的接口, 当任务执行之前开始调用此方法, 可以在这里显示进度对话框。

- `onCancelled()`

用户调用取消时, 要做的操作 4、`AsyncTask` 示例

#### 9.0.4

按照上面的步骤定义自己的异步类: `public class MyTask extends AsyncTask<String, Integer, String> { //执行的第一个方法用于在执行后台任务前做一些 UI 操作 @Override protected void onPreExecute() {`

`}`

`//第二个执行方法, 在 onPreExecute() 后执行, 用于后台任务, 不可在此方法内修改 UI @Override protected String doInBackground(String... params) { //处理耗时操作 return " 后台任务执行完毕"; }`

`/* 这个函数在 doInBackground 调用 publishProgress(int i) 时触发, 虽然调用时只有一个参数但是这里取到的是一个数组, 所以要用 progresss2来取值第 n 个参数就用 progress[n] 来取值 */ @Override protected void onProgressUpdate(Integer... progresses) { //"loading..." + progresses2 + "%" super.onProgressUpdate(progress); }`

`*doInBackground 返回时触发, 换句话说, 就是 doInBackground 执行完后触发这里的 result 就是上面 doInBackground 执行后的返回值, 所以这里是 " 后台任务执行完毕" * @Override protected void onPostExecute(String result) {`

`}`

`//onCancelled 方法用于在取消执行中的任务时更改 UI @Override protected void onCancelled() {`

`} }` 在主线程申明该类的对象, 调用对象的 `execute()` 函数开始执行。`MyTask t = new MyTask(); t.execute();`//这里没有参数 5、使用 `AsyncTask` 需要注意的地方

## 9.0.5

- AsyncTask 内部的 Handler 需要和主线程交互，所以 AsyncTask 的实例必须在 UI 线程中创建
- AsyncTaskResult 的 doInBackground(mParams) 方法执行异步任务运行在子线程中，其他方法运行在主线程中，可以操作 UI 组件。
- 一个 AsyncTask 任务只能被执行一次。
- 运行中可以随时调用 AsyncTask 对象的 cancel(boolean) 方法取消任务，如果成功，调用 isCancelled() 会返回 true，并且不会执行 onPostExecute() 方法了，而是执行 onCancelled() 方法。
- 对于想要立即开始执行的异步任务，要么直接使用 Thread，要么单独创建线程池提供给 AsyncTask。默认的 AsyncTask 不一定会立即执行你的任务，除非你提供给他一个单独的线程池。如果不与主线程交互，直接创建一个 Thread 就可以了。

## 10 Android 自定义 View 入门

- 在 android 应用开发过程中，固定的一些控件和属性可能满足不了开发的需求，所以在一些特殊情况下，我们需要自定义控件与属性。

### 10.1 一、实现步骤

1. 继承 View 类或其子类
2. 复写 view 中的一些函数
3. 为自定义 View 类增加属性（两种方式）
4. 绘制控件（导入布局）
5. 响应用户事件
6. 定义回调函数（根据自己需求来选择）

### 10.2 二、哪些方法需要被重写

- onDraw()

view 中 onDraw() 是个空函数，也就是说具体的视图都要覆写该函数来实现自己的绘制。对于 ViewGroup 则不需要实现该函数，因为作为容器是“没有内容”的（但必须实现 dispatchDraw() 函数，告诉子 view 绘制自己）。

- onLayout()

主要是为 ViewGroup 类型布局子视图用的，在 View 中这个函数为空函数。

- onMeasure()

用于计算视图大小（即长和宽）的方式，并通过 setMeasuredDimension(width, height) 保存计算结果。

- onTouchEvent

定义触屏事件来响应用户操作。还有一些不常用的方法：

- onKeyDown 当按下某个键盘时
-



**onKeyUp** 当松开某个键盘时    **onTrackballEvent** 当发生轨迹球事件时    **onSizeChange()** 当该组件的大小被改变时    **onFinishInflate()** 回调方法,当应用从 XML 加载该组件并用它构建界面之后调用的方法    **onWindowFocusChanged(boolean)** 当该组件得到、失去焦点时    **onAttachedToWindow()** 当把该组件放入到某个窗口时    **onDetachedFromWindow()** 当把该组件从某个窗口上分离时触发的方法    **onWindowVisibilityChanged(int)**: 当包含该组件的窗口的可见性发生改变时触发的方法    **View** 的绘制流程绘制流程函数调用关系如下图: [这里写图片描述](<http://img.blog.csdn.net/20160617150747985>) 我们调用 **requestLayout()** 的时候,会触发 **measure** 和 **layout** 过程,调用 **invalidate**,会执行 **draw** 过程。

### 10.3 三. 自定义控件的三种方式

1. 继承已有的控件当要实现的控件和已有的控件在很多方面比较类似,通过对已有控件的扩展来满足要求。2. 继承一个布局文件一般用于自定义组合控件,在构造函数中通过 **inflater** 和 **addView()** 方法加载自定义控件的布局文件形成图形界面(不需要 **onDraw** 方法)。3. 继承 **view** 通过 **onDraw** 方法来绘制出组件界面。

### 10.4 四. 自定义属性的两种方法

1. 在布局文件中直接加入属性,在构造函数中去获得。布局文件:  

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android" android:layout_width="match_parent" android:layout_height="match_parent">
    <com.example.demo.myView android:layout_width="wrap_content" android:layout_height="wrap_content"
        Text="@string/hello_world" />
</RelativeLayout>
```

获取属性值: `public myView(Context context, AttributeSet attrs) { super(context, attrs); // TODO Auto-generated constructor stub
int textId = attrs.getAttributeResourceValue(null, "Text", 0);
String text = context.getResources().getText(textId).toString(); }`
2. 在 **res/values/** 下建立一个 **attrs.xml** 来声明自定义 **view** 的属性。可以定义的属性有:

```
<declare-styleable name="名称"> //参考某一资源 ID (name 可以随便命名)
    <attr name="background" format="reference" /> //颜色值
    <attr name="textColor" format="color" /> //布尔值
    <attr name="focusable" format="boolean" /> //尺寸值
    <attr name="layout_width" format="dimension" /> //浮点值
    <attr name="fromAlpha" format="float" /> //整型值
    <attr name="frameDuration" format="integer" /> //字符串
    <attr name="text" format="string" /> //百分数
    <attr name="pivotX" format="fraction" /> //枚举值
    <attr name="orientation">
        <enum name="horizontal" value="0" />
        <enum name="vertical" value="1" />
    </attr> //位或运算
    <attr name="windowSoftInputMode">
        <flag name="stateUnspecified" value="0" />
        <flag name="stateUnchanged" value="1" />
    </attr> //多类型
    <attr name="background" format="reference|color" />
</declare-styleable>
```

- **attrs.xml** 进行属性声明

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="myView">
        <attr name="text" format="string"/>
        <attr name="textColor" format="color"/>
    </declare-styleable>
</resources>
```

- 添加到布局文件

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android" android:layout_width="match_parent"
    android:layout_height="match_parent" xmlns:myview="http://schemas.android.com/apk/com.example.demo">
    <com.example.demo.myView android:layout_width="wrap_content" android:layout_height="wrap_content"
        myview:text = "test" myview:textColor = "#ff0000" />
</RelativeLayout>
```

这里注意命名空间: **xmlns: 前缀 = "http://schemas.android.com/apk/res/res-auto"**,  
前缀:**TextColor** 使用属性。

- 在构造函数中获取属性值

```

public myView(Context context, AttributeSet attrs) { super(context, attrs); // TODO Auto-generated
constructor stub TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.myView);
String text = a.getString(R.styleable.myView_text); int textColor = a.getColor(R.styleable.myView_text
Color.WHITE);
    a.recycle(); } 或者: public myView(Context context, AttributeSet attrs) { super(context,
attrs); // TODO Auto-generated constructor stub TypedArray a = context.obtainStyledAttributes(attrs,
R.styleable.myView); int n = a.getIndexCount(); for(int i=0;i<n;i++){ int attr = a.getIndex(i);
switch (attr) { case R.styleable.myView_text:
    break; case R.styleable.myView_textColor:
    break;
} } a.recycle(); }

```

## 10.5 五. 自定义随手指移动的小球 (小例子)

`` 实现上面的效果我们大致需要分成这几步

- 在 `res/values/` 下建立一个 `attrs.xml` 来声明自定义 `view` 的属性
- 一个继承 `View` 并复写部分函数的自定义 `view` 的类
- 一个展示自定义 `view` 的容器界面

1. 自定义 `view` 命名为 `myView`, 它有一个属性值, 格式为 `color`、`<?xml version="1.0" encoding="utf-8"?> <resources> <declare-styleable name="myView"> <attr name="TextColor" format="color"/> </declare-styleable> </resources>` 2. 在构造函数获取获得 `view` 的属性配置和复写 `onDraw` 和 `onTouchEvent` 函数实现绘制界面和用户事件响应。 `public class myView extends View{ //定义画笔和初始位置 Paint p = new Paint(); public float currentX = 50; public float currentY = 50; public int textColor; public myView(Context context, AttributeSet attrs) { super(context, attrs); //获取资源文件里面的属性, 由于这里只有一个属性值, 不用遍历数组, 直接通过 R 文件拿出 color 值 //把属性放在资源文件里, 方便设置和复用 TypedArray array = context.obtainStyledAttributes(attrs,R.styleable.myView_textColor = array.getColor(R.styleable.myView_TextColor,Color.BLACK); array.recycle(); } @Override protected void onDraw(Canvas canvas) { super.onDraw(canvas); //画一个蓝色的圆形 p.setColor(Color.BLUE); canvas.drawCircle(currentX,currentY,30,p); //设置文字和颜色, 这里的颜色是资源文件 values 里面的值 p.setColor(textColor); canvas.drawText("BY finch",currentX-30,currentY+50,p); } @Override public boolean onTouchEvent(MotionEvent event) { currentX = event.getX(); currentY = event.getY(); invalidate(); //重新绘制图形 return true; } }` 这里通过不断的更新当前位置坐标和重新绘制图形实现效果, 要注意的是使用 `TypedArray` 后一定要记得 `recycle()`。 否则会对下次调用产生影响。 [这里写图片描述](http://img.blog.csdn.net/20160503144335969) 3. 把 `myView` 加入到 `activity_main.xml` 布局里面 `<?xml version="1.0" encoding="utf-8"?> <RelativeLayout 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" xmlns:myview="http://schemas.android.com/apk/res-auto" android:paddingBottom="@dimen/activity_vertical_margin" android:paddingLeft="@dimen/activity_horizontal_margin" android:paddingRight="@dimen/activity_horizontal_margin" android:paddingTop="@dimen/activity_vertical_margin" tools:context="finch.scu.cn.myview.MainActivity"> <finch.scu.cn.myview.myView android:layout_width="match_parent" android:layout_height="match_parent" myview:TextColor="#ff0000" /> </RelativeLayout>` 4. 最后是 `MainActivity` `public class MainActivity extends AppCompatActivity { @Override protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity_main); } }`

- 具体的 `view` 要根据具体的需求来, 比如我们要侧滑删除的 `listview` 我们可以继承 `listview`, 监听侧滑事件, 显示删除按钮实现功能。



## 11 Android 自定义 ViewGroup 入门实践

对自定义 view 还不是很了解的码友可以先看 [自定义 View 入门](<http://blog.csdn.net/Amazing7/article/details/51303289>) 这篇文章，本文主要对自定义 ViewGroup 的过程的梳理，废话不多说。

### 11.1 1.View 绘制流程

ViewGroup 也是继承于 View，下面看看绘制过程中依次会调用哪些函数。 ! [这里写图片描述](<http://img.blog.csdn.net/20160617180933525>) 说明：

- measure() 和 onMeasure()

在 View.java 源码中：`public final void measure(int widthMeasureSpec,int heightMeasureSpec){ ... onMeasure ... }` `protected void onMeasure(int widthMeasureSpec,int heightMeasureSpec) { setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec), getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec)); }` 可以看出 measure() 是被 final 修饰的，这是不可被重写。onMeasure 在 measure 方法中调用的，当我们继承 View 的时候通过重写 onMeasure 方法来测量控件大小。 layout() 和 onLayout(),draw() 和 onDraw() 类似。

- dispatchDraw()

View 中这个函数是一个空函数，ViewGroup 复写了 dispatchDraw() 来对其子视图进行绘制。自定义的 ViewGroup 一般不对 dispatchDraw() 进行复写。

- requestLayout()

当布局变化的时候，比如方向变化，尺寸的变化，会调用该方法，在自定义的视图中，如果某些情况下希望重新测量尺寸大小，应该手动去调用该方法，它会触发 measure() 和 layout() 过程，但不会进行 draw。自定义 ViewGroup 的时候一般复写

- onMeasure() 方法：

•

计算 childView 的测量值以及模式，以及设置自己的宽和高 onLayout() 方法， 对其所有 child-View 的位置进行定位 View 树:![这里写图片描述](<http://img.blog.csdn.net/20160617181010906>)

树的遍历是有序的，由父视图到子视图，每一个 ViewGroup 负责测绘它所有的子视图，而最底层的 View 会负责测绘自身。

- measure:

自上而下进行遍历，根据父视图对子视图的 MeasureSpec 以及 ChildView 自身的参数，通过 getChildMeasureSpec(parentHeightMeasure,mPaddingTop+mPaddingBottom, lp.height) 获取 ChildView 的 MeasureSpec，回调 ChildView.measure 最终调用 setMeasuredDimension 得到 ChildView 的尺寸：mMeasuredWidth 和 mMeasuredHeight

- Layout :

也是自上而下进行遍历的，该方法计算每个 ChildView 的 ChildLeft,ChildTop; 与 measure 中得到的每个 ChildView 的 mMeasuredWidth 和 mMeasuredHeight，来对 ChildView 进行布局。  
`child.layout(left,top,left+width,top+height)`

## 11.2 2 .onMeasure 过程

measure 过程会为一个 View 及所有子节点的 mMeasuredWidth 和 mMeasuredHeight 变量赋值, 该值可以通过 getMeasuredWidth() 和 getMeasuredHeight() 方法获得。**onMeasure** 过程传递尺寸的两个类:

- **ViewGroup.LayoutParams** (ViewGroup 自身的布局参数)

用来指定视图的高度和宽度等参数,使用 view.getLayoutParams() 方法获取一个视图 LayoutParams, 该方法得到的就是其所在父视图类型的 LayoutParams, 比如 View 的父控件为 RelativeLayout, 那么得到的 LayoutParams 类型就为 RelativeLayoutParams。

- 具体值
- 

MATCH\_PARENT 表示子视图希望和父视图一样大 (不包含 padding 值) WRAP\_CONTENT 表示视图为正好能包裹其内容大小 (包含 padding 值)

- **MeasureSpecs**

测量规格, 包含测量要求和尺寸的信息, 有三种模式:

- UNSPECIFIED
- 

父视图不对子视图有任何约束, 它可以达到所期望的任意尺寸。比如 ListView、ScrollView, 一般自定义 View 中用不到

- EXACTLY
- 

父视图为子视图指定一个确切的尺寸, 而且无论子视图期望多大, 它都必须在该指定大小的边界内, 对应的属性为 match\_parent 或具体值, 比如 100dp, 父控件可以通过 MeasureSpec.getSize(measureSpec) 直接得到子控件的尺寸。

- AT\_MOST
- 父视图为子视图指定一个最大尺寸。子视图必须确保它自己所有子视图可以适应在该尺寸范围内, 对应的属性为 wrap\_content, 这种模式下, 父控件无法确定子 View 的尺寸, 只能由子控件自己根据需求去计算自己的尺寸, 这种模式就是我们自定义视图需要实现测量逻辑的情况。

## 11.3 3 .onLayout 过程

子视图的具体位置都是相对于父视图而言的。View 的 onLayout 方法为空实现, 而 ViewGroup 的 onLayout 为 abstract 的, 因此, 如果自定义的自定义 ViewGroup 时, 必须实现 onLayout 函数。

在 layout 过程中, 子视图会调用 getMeasuredWidth() 和 getMeasuredHeight() 方法获取到 measure 过程得到的 mMeasuredWidth 和 mMeasuredHeight, 作为自己的 width 和 height。然后调用每一个子视图的 layout(l, t, r, b) 函数, 来确定每个子视图在父视图中的位置。

## 11.4 4. 示例程序

先上效果图：![这里写图片描述](http://img.blog.csdn.net/20160617215723283) 代码中有详细的注释，结合上文中的说明，理解应该没有问题。这里主要贴出核心代码。FlowLayout.java 中 (参照阳神的慕课课程)

- onMeasure 方法 @Override protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) { // 获得它的父容器为它设置的测量模式和大小 int sizeWidth = MeasureSpec.getSize(widthMeasureSpec); int modeWidth = MeasureSpec.getMode(widthMeasureSpec); int sizeHeight = MeasureSpec.getSize(heightMeasureSpec); int modeHeight = MeasureSpec.getMode(heightMeasureSpec); // 用于 wrap\_content 情况下，来记录父 view 宽和高 int width = 0; int height = 0; // 取每一行宽度的最大值 int lineWidth = 0; // 每一行的高度累加 int lineHeight = 0; // 获得子 view 的个数 int cCount = getChildCount(); for (int i = 0; i < cCount; i++) { View child = getChildAt(i); // 测量子 View 的宽和高 (子 view 在布局文件中是 wrap\_content) measureChild(child, widthMeasureSpec, heightMeasureSpec); // 得到 LayoutParams MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams(); // 根据测量宽度加上 Margin 值算出子 view 的实际宽度 (上文中有说明) int childWidth = child.getMeasuredWidth() + lp.leftMargin + lp.rightMargin; // 根据测量高度加上 Margin 值算出子 view 的实际高度 int childHeight = child.getMeasuredHeight() + lp.topMargin + lp.bottomMargin; // 这里的父 view 是有 padding 值的，如果再添加一个元素就超出最大宽度就换行 if (lineWidth + childWidth > sizeWidth - getPaddingLeft() - getPaddingRight()) { // 父 view 宽度 = 以前父 view 宽度、当前行宽的最大值 width = Math.max(width, lineWidth); // 换行了，当前行宽 = 第一个 view 的宽度 lineWidth = childWidth; // 父 view 的高度 = 各行高度之和 height += lineHeight; // 换行了，当前行高 = 第一个 view 的高度 lineHeight = childHeight; } else { // 叠加行宽 lineWidth += childWidth; // 得到当前行最大的高度 lineHeight = Math.max(lineHeight, childHeight); } // 最后一个控件 if (i == cCount - 1) { width = Math.max(lineWidth, width); height += lineHeight; } } /\*\*
  - EXACTLY 对应 match\_parent 或具体值
  - AT\_MOST 对应 wrap\_content
  - 在 FlowLayout 布局文件中
  - android:layout\_width="fill\_parent"
  - android:layout\_height="wrap\_content"
  - 
  - 如果是 MeasureSpec.EXACTLY 则直接使用父 ViewGroup 传入的宽和高，否则设置为自己计算的宽和高。\*/ setMeasuredDimension( modeWidth == MeasureSpec.EXACTLY ? sizeWidth : width + getPaddingLeft() + getPaddingRight(), modeHeight == MeasureSpec.EXACTLY ? sizeHeight : height + getPaddingTop() + getPaddingBottom() ); }

- onLayout 方法 // 存储所有的 View private List<List<View>> mAllViews = new ArrayList<List<View>>(); // 存储每一行的高度 private List<Integer> mLineHeight = new ArrayList<Integer>(); @Override protected void onLayout(boolean changed, int l, int t, int r, int b) { mAllViews.clear(); mLineHeight.clear(); // 当前 ViewGroup 的宽度 int width = getWidth(); int lineWidth = 0; int lineHeight = 0; // 存储每一行所有的 childView List<View> lineViews = new ArrayList<View>(); int cCount = getChildCount(); for (int i = 0; i < cCount; i++) { View child = getChildAt(i); MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams(); int childWidth = child.getMeasuredWidth(); int childHeight = child.getMeasuredHeight(); lineWidth = childWidth + lp.leftMargin + lp.rightMargin; lineHeight = Math.max(lineHeight,

```

childHeight + lp.topMargin lp.bottomMargin); lineViews.add(child); // 换行，在 onMeasure 中 childWidth 是加上 Margin 值的 if (childWidth + lineWidth + lp.leftMargin + lp.rightMargin > width - getPaddingLeft() - getPaddingRight()) { // 记录行高 mLineHeight.add(lineHeight); // 记录当前行的 Views mAllViews.add(lineViews); // 新行的行宽和行高 lineWidth = 0; lineHeight = childHeight + lp.topMargin + lp.bottomMargin; // 新行的 View 集合 lineViews = new ArrayList<View>(); } } // 处理最后一行 mLineHeight.add(lineHeight); mAllViews.add(lineViews); // 设置子 View 的位置 int left = getPaddingLeft(); int top = getPaddingTop(); // 行数 int lineNum = mAllViews.size(); for (int i = 0; i < lineNum; i++) { // 当前行的所有的 View lineViews = mAllViews.get(i); lineHeight = mLineHeight.get(i); for (int j = 0; j < lineViews.size(); j++) { View child = lineViews.get(j); // 判断 child 的状态 if (child.getVisibility() == View.GONE) { continue; } MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams(); int lc = left + lp.leftMargin; int tc = top + lp.topMargin; int rc = lc + child.getMeasuredWidth(); int bc = tc + child.getMeasuredHeight(); // 为子 View 进行布局 child.layout(lc, tc, rc, bc); left = lc + child.getMeasuredWidth() + lp.leftMargin lp.rightMargin; } left = getPaddingLeft(); top += lineHeight; } } /**

```

- 因为我们只需要支持 margin，所以直接使用系统的 MarginLayoutParams

```

*/ @Override public LayoutParams generateLayoutParams(AttributeSet attrs) { return new MarginLayoutParams(getContext(), attrs); }

```

- 以及 MainActivity.java

```

public class MainActivity extends Activity { LayoutInflater mInflater; @InjectView(R.id.id_flowlayout1)
FlowLayout idFlowlayout1; @InjectView(R.id.id_flowlayout2) FlowLayout idFlowlayout2; private String[] mVals = new String[] { "Do", "one thing", "at a time", "and do well.", "Never", "forget", "to say", "thanks.", "Keep on", "going ", "never give up." }; @Override protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity_main); ButterKnife.inject(this); mInflater = LayoutInflater.from(this); initFlowlayout2(); } public void initFlowlayout2() { for (int i = 0; i < mVals.length; i++) { final RelativeLayout rl2 = (RelativeLayout) mInflater.inflate(R.layout.flow_layout, idFlowlayout2, false); TextView tv2 = (TextView) rl2.findViewById(R.id.tv); tv2.setText(mVals[i]); rl2.setTag(i); idFlowlayout2.addView(rl2); rl2.setOnClickListener() { @Override public void onClick(View v) { int i = (int) v.getTag(); addViewToFlowlayout1(i); rl2.setBackgroundResource(R.drawable.flow_layout_disable_bg); rl2.setOnClickListener() { @Override public void onClick(View v) { int i = (int) v.getTag(); idFlowlayout2.removeView(rl2); } } } } } public void addViewToFlowlayout1(int i){ RelativeLayout rl1 = (RelativeLayout) mInflater.inflate(R.layout.flow_layout, idFlowlayout1, false); ImageView iv = (ImageView) rl1.findViewById(R.id.iv); iv.setVisibility(View.VISIBLE); TextView tv1 = (TextView) rl1.findViewById(R.id.tv); tv1.setText(mVals[i]); rl1.setTag(i); idFlowlayout1.addView(rl1); rl1.setOnClickListener(new View.OnClickListener() { @Override public void onClick(View v) { int i = (int) v.getTag(); idFlowlayout1.removeView(v); View view = idFlowlayout2.getChildAt(i); view.setClickable(true); view.setBackgroundResource(R.drawable.flow_layout_disable_bg); } } } } }

```

- 这个项目源码已经上传，想要看源码的朋友可以  
> 点击 [FlowLayout](<https://github.com/fanrunqi/FlowLayout>) >
- 如果有什么疑问可以给我留言，不足之处欢迎在 github 上指出，谢谢！

## 12 Android View 事件分发机制源码分析

- 在 android 开发中会经常遇到滑动冲突（比如 ScrollView 或是 SlidingMenu 与 ListView 的嵌套）的问题，需要我们深入的了解 android 事件响应机制才能解决，事件响应机制已经是 android 开发者必不可少的知识。

## 12.1 1. 涉及到事件响应的常用方法构成

用户在手指与屏幕接触过程中通过 `MotionEvent` 对象产生一系列事件，它有四种状态：

- `MotionEvent.ACTION_DOWN` : 手指按下屏幕的瞬间（一切事件的开始）
- `MotionEvent.ACTION_MOVE` : 手指在屏幕上移动
- `MotionEvent.ACTION_UP` : 手指离开屏幕瞬间
- `MotionEvent.ACTION_CANCEL` : 取消手势，一般由程序产生，不会由用户产生

Android 中的事件 `onClick`, `onLongClick`, `onScroll`, `onFling` 等等，都是由许多个 `Touch` 事件构成的（一个 `ACTION_DOWN`, `n` 个 `ACTION_MOVE`, 1 个 `ACTION_UP`）。android 事件响应机制是先分发 \*\*（先由外部的 **View** 接收，然后依次传递给其内层的最小 **View**）再 \*\* 处理（从最小 `View` 单元（事件源）开始依次向外层传递。）的形式实现的。复杂性表现在：可以控制每层事件是否继续传递（分发和拦截协同实现），以及事件的具体消费（事件分发也具有事件消费能力）。

## 12.2 2 .android 事件处理涉及到的三个重要函数

- 事件分发: **`public boolean dispatchTouchEvent(MotionEvent ev)`**

当有监听到事件时，首先由 `Activity` 进行捕获，进入事件分发处理流程。（因为 `activity` 没有事件拦截，`View` 和 `ViewGroup` 有）会将事件传递给最外层 `View` 的 `dispatchTouchEvent(MotionEvent ev)` 方法，该方法对事件进行分发。

- `return true` : 表示该 `View` 内部消化掉了所有事件。
- `return false` : 事件在本层不再继续进行分发，并交由 \*\* 上层 \*\* 控件的 `onTouchEvent` 方法进行消费（如果本层控件已经是 `Activity`，那么事件将被系统消费或处理）。
- 如果事件分发返回系统默认的 `super.dispatchTouchEvent(ev)`，事件将分发给本层的事件拦截 `onInterceptTouchEvent` 方法进行处理
- 事件拦截: **`public boolean onInterceptTouchEvent(MotionEvent ev)`**
  - `return true` : 表示将事件进行拦截，并将拦截到的事件交由本层控件的 `onTouchEvent` 进行处理；
  - `return false` : 则表示不对事件进行拦截，事件得以成功分发到子 `View`。并由子 `View` 的 `dispatchTouchEvent` 进行处理。
  - 如果返回 `super.onInterceptTouchEvent(ev)`，默认表示拦截该事件，并将事件传递给当前 `View` 的 `onTouchEvent` 方法，和 `return true` 一样。
- 事件响应: **`public boolean onTouchEvent(MotionEvent ev)`**

在 `dispatchTouchEvent`（事件分发）返回 `super.dispatchTouchEvent(ev)` 并且 `onInterceptTouchEvent`（事件拦截返回 `true` 或 `super.onInterceptTouchEvent(ev)`）的情况下，那么事件会传递到 `onTouchEvent` 方法，该方法对事件进行响应。

- 如果 `return true`，表示 `onTouchEvent` 处理完事件后消费了此次事件。此时事件终结；
- 如果 `return false`，则表示不响应事件，那么该事件将会不断向上层 `View` 的 `onTouchEvent` 方法传递，直到某个 `View` 的 `onTouchEvent` 方法返回 `true`，如果到了最顶层 `View` 还是返回 `false`，那么认为该事件不消耗，则在同一个事件系列中，当前 `View` 无法再次接收到事件，该事件会交由 `Activity` 的 `onTouchEvent` 进行处理；

- 如果 `return super.dispatchTouchEvent(ev)`, 则表示不响应事件, 结果与 `return false` 一样。
- 从以上过程中可以看出, `dispatchTouchEvent` 无论返回 `true` 还是 `false`, 事件都不再进行分发, 只有当其返回 `super.dispatchTouchEvent(ev)`, 才表明其具有向下层分发的愿望, 但是是否能够分发成功, 则需要经过事件拦截 `onInterceptTouchEvent` 的审核。事件是否向上传递处理是由 `onTouchEvent` 的返回值决定的。

![这里写图片描述](http://img.blog.csdn.net/20160428161104339) (图来自网络)

## 12.3 3.View 源码分析

Android 中 `ImageView`、`textView`、`Button` 等继承于 `View` 但没有重写的 `dispatchTouchEvent` 方法, 所以都用的 `View` 的该方法进行事件分发。看 `View` 重要函数部分源码: `public boolean dispatchTouchEvent(MotionEvent event) { //返回 true, 表示该 View 内部消化掉了所有事件。返回 false, 表示 View 内部只处理了 ACTION_DOWN 事件, 事件继续传递, 向上级 View(ViewGroup) 传递。if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED && mOnTouchListener.onTouch(this, event)) { //此处的 onTouch 方式就是回调的我们注册 OnTouchListener 时重写的 onTouch() 方法 return true; } return onTouchEvent(event); }` 首先进行三个条件的判断: (1) 查看是否给 `button` 设置了 `OnTouchListener()` 事件; (2) 控件是否 `Enable`; (控件默认都是 `enable` 的) (3) `button` 里面实现的 `OnTouchListener` 监听里的 `onTouch()` 方法是否返回 `true`; 如果条件都满足, 则该事件被消耗掉, 不再进入 `onTouchEvent` 中处理。否则将事件将交给 `onTouchEvent` 方法处理。`public boolean onTouchEvent(MotionEvent event) { ...`

`/* 当前 onTouch 的组件必须是可点击的比如 Button, ImageButton 等等, 此处 CLICKABLE 为 true, 才会进入 if 方法, 最后返回 true。如果是 ImageView、TextView 这些默认为不可点击的 View, 此处 CLICKABLE 为 false, 最后返回 false。当然会有特殊情况, 如果给这些 View 设置了 onClick 监听器, 此处 CLICKABLE 也将为 true */`

```
if(((viewFlags & CLICKABLE) = CLICKABLE || (viewFlags & LONG_CLICKABLE) = LONG_CLICKABLE)
{ switch (event.getAction()) { case MotionEvent.ACTION_UP: ... if (!post(mPerformClick))
{ performClick();// 实际就是回调了我们注册的 OnClickListener 中重新的 onClick() 方法 } ...
break;
```

```
case MotionEvent.ACTION_DOWN: ... break;
case MotionEvent.ACTION_CANCEL: ... break;
case MotionEvent.ACTION_MOVE: ... break; } return true; }
return false; } public boolean performClick() { ... // if (li != null && li.mOnClickListener
!= null) { ... li.mOnClickListener.onClick(this); return true; }
return false; } public void setOnClickListener(OnClickListener l) { if (!isClickable()) {
setClickable(true); } getListenerInfo().mOnClickListener = l; }
```

- 只有我们注册 `OnTouchListener` 时重写的
- `onTouch()` 方法中
- 
- 返回 `false` —> 执行 `onTouchEvent` 方法—> 导致 `onClick()` 回调方法执行
- 

返回 `true` —> `onTouchEvent` 方法不执行—> 导致 `onClick()` 回调方法不会执行

## 12.4 4.ViewGroup 源码分析

Android 中诸如 `LinearLayout` 等的五大布局控件,都是继承自 `ViewGroup`,而 `ViewGroup` 本身是继承自 `View`,所以 `ViewGroup` 的事件处理机制对这些控件都有效。部分源码:`public boolean dispatchTouchEvent(MotionEvent ev) { final int action = ev.getAction(); final float xf = ev.getX(); final float yf = ev.getY(); final float scrolledXFloat = xf + mScrollX; final float scrolledYFloat = yf + mScrollY; final Rect frame = mTempRect;`

`//这个值默认是 false, 然后我们可以通过 requestDisallowInterceptTouchEvent(boolean disallowIntercept) 方法 //来改变 disallowIntercept 的值 boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;`

`//这里是 ACTION_DOWN 的处理逻辑 if (action == MotionEvent.ACTION_DOWN) { //清除 mMotionTarget, 每次 ACTION_DOWN 都很设置 mMotionTarget 为 null if (mMotionTarget != null) { mMotionTarget = null; }`

`//disallowIntercept 默认是 false, 就看 ViewGroup 的 onInterceptTouchEvent() 方法 if (disallowIntercept || !onInterceptTouchEvent(ev)) { //第一点 ev.setAction(MotionEvent.ACTION_DOWN); final int scrolledXInt = (int) scrolledXFloat; final int scrolledYInt = (int) scrolledYFloat; final View[] children = mChildren; final int count = mChildrenCount; //遍历其子 View for (int i = count - 1; i >= 0; i-) { //第二点 final View child = children[i];`

`//如果该子 View 是 VISIBLE 或者该子 View 正在执行动画, 表示该 View 才 //可以接受到 Touch 事件 if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE`

`child.getAnimation() != null) {`

`//获取子 View 的位置范围 child.getHitRect(frame);`

`//如 Touch 到屏幕上的点在该子 View 上面 if (frame.contains(scrolledXInt, scrolledYInt)) { // offset the event to the view's coordinate system final float xc = scrolledXFloat - child.mLeft; final float yc = scrolledYFloat - child.mTop; ev.setLocation(xc, yc); child.mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;`

`//调用该子 View 的 dispatchTouchEvent() 方法 if (child.dispatchTouchEvent(ev)) { // 如果 child.dispatchTouchEvent(ev) 返回 true 表示 //该事件被消费了,设置 mMotionTarget 为该子 View mMotionTarget = child; //直接返回 true return true; } / The event didn't get handled, try the next view. / Don't reset the event's location, it's not // necessary here. } } }`

`//判断是否为 ACTION_UP 或者 ACTION_CANCEL boolean isUpOrCancel = (action == MotionEvent.ACTION_UP || (action == MotionEvent.ACTION_CANCEL);`

`if (isUpOrCancel) { //如果是 ACTION_UP 或者 ACTION_CANCEL, 将 disallowIntercept 设置为默认的 false //假如我们调用了 requestDisallowInterceptTouchEvent() 方法来设置 disallowIntercept 为 true //当我们抬起手指或者取消 Touch 事件的时候要将 disallowIntercept 重置为 false //所以说上面的 disallowIntercept 默认在我们每次 ACTION_DOWN 的时候都是 false mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT; }`

`/ The event wasn't an ACTION_DOWN, dispatch it to our target if / we have one. final View target = mMotionTarget; //mMotionTarget 为 null 意味着没有找到消费 Touch 事件的 View, 所以我们需要调用 ViewGroup 父类的 //dispatchTouchEvent() 方法, 也就是 View 的 dispatchTouchEvent() 方法 if (target == null) { / We don't have a target, this means we're handling the / event as a regular view. ev.setLocation(xf, yf); if ((mPrivateFlags & CANCEL_NEXT_UP_EVENT) != 0) { ev.setAction(MotionEvent.ACTION_CANCEL); mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT; } return super.dispatchTouchEvent(ev); }`

`//这个 if 里面的代码 ACTION_DOWN 不会执行, 只有 ACTION_MOVE //ACTION_UP 才会走到这里, 假如在 ACTION_MOVE 或者 ACTION_UP 拦截的 //Touch 事件, 将 ACTION_CANCEL 派发给 target, 然后直接返回 true //表示消费了此 Touch 事件 if (!disallowIntercept && onInterceptTouchEvent(ev)) { final float xc = scrolledXFloat - (float) target.mLeft; final float yc = scrolledYFloat - (float) target.mTop; mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT; ev.setAction(MotionEvent.ACTION_CANCEL); ev.setLocation(xc, yc);`



```

    if (!target.dispatchTouchEvent(ev)) { } // clear the target mMotionTarget = null; / Don't
    dispatch this event to our own view, because we already / saw it when intercepting; we just
    want to give the following // event to the normal onTouchEvent(). return true; }
    if (isUpOrCancel) { mMotionTarget = null; }
    / finally offset the event to the target's coordinate system and / dispatch the event. fi
    nal float xc = scrolledXFloat - (float) target.mLeft; final float yc = scrolledYFloat - (float) tar
    get.mTop; ev.setLocation(xc, yc);
    if ((target.mPrivateFlags & CANCEL_NEXT_UP_EVENT) != 0) { ev.setAction(MotionEvent.ACTION
    target.mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT; mMotionTarget = null; }
    //如果没有拦截 ACTION_MOVE, ACTION_DOWN 的话, 直接将 Touch 事件派发给 target re
    turn target.dispatchTouchEvent(ev); }

```

- 1、dispatchTouchEvent 作用：决定事件是否由 onInterceptTouchEvent 来拦截处理。

返回 super.dispatchTouchEvent 时，由 onInterceptTouchEvent 来决定事件的流向返回 false 时，会继续分发事件，自己内部只处理了 ACTION\_DOWN 返回 true 时，不会继续分发事件，自己内部处理了所有事件（ACTION\_DOWN, ACTION\_MOVE, ACTION\_UP）

- 2、onInterceptTouchEvent 作用：拦截事件，用来决定事件是否传向子 View

返回 true 时，拦截后交给自己的 onTouchEvent 处理返回 false 时，拦截后交给子 View 来处理

- 3、onTouchEvent 作用：事件最终到达这个方法

返回 true 时，内部处理所有的事件，换句话说，后续事件将继续传递给该 view 的 onTouchEvent() 处理返回 false 时，事件会向上传递，由 onTouchEvent 来接受，如果最上面 View 中的 onTouchEvent 也返回 false 的话，那么事件就会消失

## 12.5 5. 总结

- 如果 ViewGroup 找到了能够处理该事件的 View，则直接交给子 View 处理，自己的 onTouchEvent 不会被触发；
- 可以通过复写 onInterceptTouchEvent(ev) 方法，拦截子 View 的事件（即 return true），把事件给自己处理，则会执行自己对应的 onTouchEvent 方法。
- 子 View 可以通过调用 getParent().requestDisallowInterceptTouchEvent(true); 阻止 ViewGroup 对其 MOVE 或者 UP 事件进行拦截；
- 一个点击事件产生后，它的传递过程如下：

Activity->Window->View。顶级 View 接收到事件之后，就会按相应规则去分发事件。如果一个 View 的 onTouchEvent 方法返回 false，那么将会交给父容器的 onTouchEvent 方法进行处理，逐级往上，如果所有的 View 都不处理该事件，则交由 Activity 的 onTouchEvent 进行处理。

- 如果某一个 View 开始处理事件，如果他不消耗 ACTION\_DOWN 事件（也就是 onTouchEvent 返回 false），则同一事件序列比如接下来进行 ACTION\_MOVE，则不会再交给该 View 处理。
- ViewGroup 默认不拦截任何事件。
- 诸如 TextView、ImageView 这些不作为容器的 View，一旦接受到事件，就调用 onTouchEvent 方法，它们本身没有 onInterceptTouchEvent 方法。正常情况下，它们都会消耗事件（返回 true），除非它们是不可点击的（clickable 和 longClickable 都为 false），那么就会交由父容器的 onTouchEvent 处理。

- 点击事件分发过程如下 dispatchTouchEvent—>OnTouchListener 的 onTouch 方法—>onTouchEvent—>OnClickListener 的 onClick 方法。也就是说，我们平时调用的 setOnClickListener，优先级是最低的，所以，onTouchEvent 或 OnTouchListener 的 onTouch 方法如果返回 true，则不响应 onClick 方法...

## 13 BroadcastReceiver 使用总结

### 13.1 BroadcastReceiver 的定义

广播是一种广泛运用的在应用程序之间传输信息的机制，主要用来监听系统或者应用发出的广播信息，然后根据广播信息作为相应的逻辑处理，也可以用来传输少量、频率低的数据。在实现开机启动服务和网络状态改变、电量变化、短信和来电时通过接收系统的广播让应用程序作出相应的处理。BroadcastReceiver 自身并不实现图形用户界面，但是当它收到某个通知后，BroadcastReceiver 可以通过启动 Service、启动 Activity 或是 NotificationManager 提醒用户。

### 13.2 BroadcastReceiver 使用注意

当系统或应用发出广播时，将会扫描系统中的所有广播接收者，通过 action 匹配将广播发送给相应的接收者，接收者收到广播后将会产生一个广播接收者的实例，执行其中的 onReceive() 这个方法；特别需要注意的是这个实例的生命周期只有 10 秒，如果 10 秒内没执行结束 onReceive()，系统将会报错。在 onReceive() 执行完毕之后，该实例将会被销毁，所以不要在 onReceive() 中执行耗时操作，也不要里面创建子线程处理业务（因为可能子线程没处理完，接收者就被回收了，那么子线程也会跟着被回收掉）；正确的处理方法就是通过 in 调用 activity 或者 service 处理业务。

### 13.3 BroadcastReceiver 的注册

BroadcastReceiver 的注册方式有且只有两种，一种是静态注册（推荐使用），另外一种动态注册，广播接收者在注册后就开始监听系统或者应用之间发送的广播消息。**\*\* 接收短信广播示例 \*\***：定义自己的 BroadcastReceiver 类 public class MyBroadcastReceiver extends BroadcastReceiver {

```
// action 名称 String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED" ;
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals( SMS_RECEIVED )) { // 一个 receiver 可以接收多个 action 的，
    即可以有多个 intent-filter，需要在 onReceive 里面对 intent.getAction(action name) 进行判断。}
    } }
```

#### 13.3.1 静态方式

在 AndroidManifest.xml 的 application 里面定义 receiver 并设置要接收的 action。< receiver android:name = ".MyBroadcastReceiver" > < intent-filter android:priority = "777" > <action android:name = "android.provider.Telephony.SMS\_RECEIVED" > < intent-filter > </ receiver > 这里的 priority 取值是 -1000 到 1000，值越大优先级越高，同时注意加上系统接收短信的权限。"" < uses-permission android:name = "android.permission.RECEIVE\_SMS" /> 静态注册的广播接收者是一个常驻在系统中的全局监听器，当你在应用中配置了一个静态的 BroadcastReceiver，安装了应用后而无论应用是否处于运行状态，广播接收者都是已经常驻在系统中了。同时应用里的所有 receiver 都在清单文件里面，方便查看。要销毁掉静态注册的广播接收者，可以通过调用 PackageManager 将 Receiver 禁用。

### 13.3.2 动态方式

在 Activity 中声明 BroadcastReceiverReceiver 的扩展对象，在 onResume 中注册， onPause 中卸载。

```
public class MainActivity extends Activity { MyBroadcastReceiver receiver; @Override protected void onResume() { // 动态注册广播 (代码执行到这才会开始监听广播消息，并对广播消息作为相应的处理) receiver = new MyBroadcastReceiver(); IntentFilter intentFilter = new IntentFilter("android.provider.Telephony.SMS_RECEIVED"); registerReceiver( receiver , intentFilter); super.onResume(); } @Override protected void onPause() { // 撤销注册 (撤销注册后广播接收者将不会再监听系统的广播消息) unregisterReceiver(receiver); super.onPause(); } }
```

### 13.3.3 静态注册和动态注册的区别

1、静态注册的广播接收者一经安装就常驻在系统之中，不需要重新启动唤醒接收者；动态注册的广播接收者随着应用的生命周期，由 registerReceiver 开始监听，由 unregisterReceiver 撤销监听，如果应用退出后，没有撤销已经注册的接收者应用应用将会报错。2、当广播接收者通过 intent 启动一个 activity 或者 service 时，如果 intent 中无法匹配到相应的组件。动态注册的广播接收者将会导致应用报错而静态注册的广播接收者将不会有任何报错，因为自从应用安装完成后，广播接收者跟应用已经脱离了关系。

## 13.4 发送 BroadcastReceiver

发送广播主要有两种类型：

### 13.4.1 普通广播

应用在需要通知各个广播接收者的情况下使用，如开机启动使用方法： sendBroadcast() Intent intent = new Intent("android.provider.Telephony.SMS\_RECEIVED"); //通过 intent 传递少量数据 intent.putExtra("data", "finch"); // 发送普通广播 sendBroadcast(Intent); 普通广播是完全异步的，可以在同一时刻（逻辑上）被所有接收者接收到，所有满足条件的 BroadcastReceiver 都会随机地执行其 onReceive() 方法。同级别接收是先后是随机的；级别低的收到广播；消息传递的效率比较高，并且无法中断广播的传播。

### 13.4.2 有序广播

应用在需要有特定拦截的场景下使用，如黑名单短信、电话拦截。 使用方法： sendOrderedBroadcast(intent, receiverPermission); receiverPermission：一个接收器必须持以接收您的广播。如果为 null，不经许可的要求（一般都为 null）。//发送有序广播 sendOrderedBroadcast(intent, null); 在有序广播中，我们可以在前一个广播接收者将处理好的数据传送给后面的广播接收者，也可以调用 abortBroadcast() 来终结广播的传播。 public void onReceive(Context arg0, Intent intent) { //获取上一个广播的 bundle 数据 Bundle bundle = getResultExtras(true); //true: 前一个广播没有结果时创建新的 Bundle; false: 不创建 Bundle bundle.putString("key", "777"); //将 bundle 数据放入广播中传给下一个广播接收者 setResultExtras(bundle); //终止广播传给下一个广播接收者 abortBroadcast(); } 高级别的广播收到该广播后，可以决定把该广播是否截断掉。同级别接收是先后是随机的，如果先接收到的把广播截断了，同级别的例外的接收者是无法收到该广播。

### 13.4.3 异步广播

使用方法： sendStickyBroadcast()：发出的 Intent 当接收 Activity（动态注册）重新处于 onResume 状态之后就能重新接受到其 Intent。（the Intent will be held to be re-broadcast to future receivers）。就是说 sendStickyBroadcast 发出的最后一个 Intent 会被保留，下次当 Activity 处于活跃的时候又会接受到它。发这个广播需要权限：<uses-permission android:name="android.permission.BROADCAST\_"

/> 卸载该广播: `removeStickyBroadcast(intent)`; 在卸载之前该 `intent` 会保留, 接收者在可接收状态都能获得。

#### 13.4.4 异步有序广播

使用方法: `sendStickyOrderedBroadcast(intent, resultReceiver, scheduler, initialCode, initialData, initialExtras)`: 这个方法具有有序广播的特性也有异步广播的特性; 同时需要限权: `<uses-permission android:name="android.permission.BROADCAST_STICKY" />`

### 13.5 总结

- 静态广播接收的处理器是由 `PackageManagerService` 负责, 当手机启动或者新安装了应用的时候, `PackageManagerService` 会扫描手机中所有已安装的 APP 应用, 将 `AndroidManifest.xml` 中有关注册广播的信息解析出来, 存储至一个全局静态变量当中。
- 动态广播接收的处理器是由 `ActivityManagerService` 负责, 当 APP 的服务或者进程起来之后, 执行了注册广播接收的代码逻辑, 即进行加载, 最后会存储在一个另外的全局静态变量中。需要注意的是:

1、这个并非是一成不变的, 当程序被杀死之后, 已注册的动态广播接收器也会被移出全局变量, 直到下次程序启动, 再进行动态广播的注册, 当然这里面的顺序也已经变更了一次。2、这里也并没完整的进行广播的排序, 只记录的注册的先后顺序, 并未有结合优先级的处理。

- 广播发出的时候, 广播接收者接收的顺序如下:

1. 当广播为 **\*\* 普通广播 \*\*** 时, 有如下的接收顺序: 无视优先级动态优先于静态同优先级的动态广播接收器, **\*\* 先注册的大于后注册的 \*\*** 同优先级的静态广播接收器, **\*\* 先扫描的大于后扫描的 \*\*** 2. 如果广播为 **\*\* 有序广播 \*\***, 那么会将动态广播处理器和静态广播处理器合并在一起处理广播的消息, 最终确定广播接收的顺序: 优先级高的先接收 同优先级的动静态广播接收器, **\*\* 动态优先于静态 \*\*** 同优先级的动态广播接收器, **\*\* 先注册的大于后注册的 \*\*** 同优先级的静态广播接收器, **\*\* 先扫描的大于后扫描的 \*\***

### 13.6 一些常用的系统广播的 **action** 和 **permission**

- 开机启动

```
<action android:name="android.intent.action.BOOT_COMPLETED"/> <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

- 网络状态

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/> <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/> 网络是否可用的方法: public static boolean isNetworkAvailable(Context context) { ConnectivityManager mgr = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE); NetworkInfo[] info = mgr.getAllNetworkInfo(); if (info != null) { for (int i = 0; i < info.length; i++) { if (info[i].getState() == NetworkInfo.State.CONNECTED) { return true; } } } return false; }
```

- 电量变化

```
<action android:name="android.intent.action.BATTERY_CHANGED"/> BroadcastReceiver 的 onReceive 方法: public void onReceive(Context context, Intent intent) { int currLevel = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 0); //当前电量
```

```
int total = intent.getIntExtra(BatteryManager.EXTRA_SCALE, 1); //总电量 int percent = currLevel * 100 / total; Log.i(TAG, "battery: " + percent + "%"); }
```

## 14 ContentProvider 实例详解

### 14.1 1.ContentProvider 是什么？

ContentProvider（内容提供者）是 Android 的四大组件之一，管理 android 以结构化方式存放的数据，以相对安全的方式封装数据（表）并且提供简易的处理机制和统一的访问接口供 \*\* 其他程序 \*\* 调用。Android 的数据存储方式总共有五种，分别是：Shared Preferences、网络存储、文件存储、外储存储、SQLite。但一般这些存储都只是在单独的一个应用程序之中达到一个数据的共享，有时候我们需要操作其他应用程序的一些数据，就会用到 ContentProvider。而且 Android 为常见的一些数据提供了默认的 ContentProvider（包括音频、视频、图片和通讯录等）。但注意 ContentProvider 它也只是个中间人，真正操作的数据源可能是数据库，也可以是文件、xml 或网络等其他存储方式。

### 14.2 2.URL

URL（统一资源标识符）代表要操作的数据，可以用来标识每个 ContentProvider，这样你可以通过指定的 URI 找到想要的 ContentProvider，从中获取或修改数据。在 Android 中 URI 的格式如下图所示：![这里写图片描述](<http://img.blog.csdn.net/20160505154322045>)

- A

schema，已经由 Android 所规定为：content://。

- B

主机名（Authority），是 URI 的授权部分，是唯一标识符，用来定位 ContentProvider。

- C 部分和 D 部分：是每个 ContentProvider 内部的路径部分

- C

指向一个对象集合，一般用表的名字，如果没有指定 D 部分，则返回全部记录。

- D

指向特定的记录，这里表示操作 user 表 id 为 7 的记录。如果要操作 user 表中 id 为 7 的记录的 name 字段，D 部分变为 \*\*/7/name\*\* 即可。

- URI 模式匹配通配符

- 

- \*: 匹配的任意长度的任何有效字符的字符串。

- 

- #: 匹配的任意长度的数字字符的字符串。

- 

- 如：

- 

- content://com.example.app.provider/\*

- 匹配 provider 的任何内容 url

- 

- content://com.example.app.provider/table3/#

- 匹配 table3 的所有行

### 14.2.1 2. 1 MIME

MIME 是指定某个扩展名的文件用一种应用程序来打开,就像你用浏览器查看 PDF 格式的文件,浏览器会选择合适的应用来打开一样。Android 中的工作方式跟 HTTP 类似,ContentProvider 会根据 URI 来返回 MIME 类型,ContentProvider 会返回一个包含两部分的字符串。MIME 类型一般包含两部分,如:

- text/html

text/css text/xml application/pdf 分为类型和子类型,Android 遵循类似的约定来定义 MIME 类型,每个内容类型的 Android MIME 类型有两种形式:多条记录(集合)和单条记录。集合记录: vnd.android.cursor.dir/自定义单条记录: vnd.android.cursor.item/自定义 vnd 表示这些类型和子类型具有非标准的、供应商特定的形式。Android 中类型已经固定好了,不能更改,只能区别是集合还是单条具体记录,子类型可以按照格式自己填写。在使用 Intent 时,会用到 MIME,根据 Mimetype 打开符合条件的活动。下面分别介绍 Android 系统提供了两个用于操作 Uri 的工具类: ContentUri 和 UriMatcher。

### 14.2.2 2. 2 ContentUri

ContentUri 包含一个便利的函数 withAppendedId() 来向 URI 追加一个 id。 Uri uri = Uri.parse("content://cn.scu.myprovider/user") Uri resultUri = ContentUri.withAppendedId(uri, 7); //生成后的 Uri 为: content://cn.scu.myprovider/user/7 同时提供 parseId(uri) 方法用于从 URL 中获取 ID: Uri uri = Uri.parse("content://cn.scu.myprovider/user/7") long personid = ContentUri.parseId(uri); //获取的结果为:7

### 14.2.3 2. 3 UriMatcher

UriMatcher 本质上是一个文本过滤器,用在 contentProvider 中帮助我们过滤,分辨出查询者想要查询哪个数据表。举例说明:

- 第一步,初始化:

```
UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH); //常量 UriMatcher.NO_MATCH 表示不匹配任何路径的返回码
```

- 第二步,注册需要的 Uri:

```
//USER 和 USER_ID 是两个 int 型数据 matcher.addURI("cn.scu.myprovider", "user", USER);  
matcher.addURI("cn.scu.myprovider", "user/#",USER_ID); //如果 match() 方法匹配 content://cn.scu.my  
路径,返回匹配码为 USER
```

- 第三部,与已经注册的 Uri 进行匹配:

```
/*
```

- 如果操作集合,则必须以 vnd.android.cursor.dir 开头
- 如果操作非集合,则必须以 vnd.android.cursor.item 开头
- \*/

```
@Override public String getType(Uri uri) { Uri uri = Uri.parse("content://" + "cn.scu.myprovider"  
+ "/user"); switch(matcher.match(uri)){ case USER: return "vnd.android.cursor.dir/user"; case  
USER_ID: return "vnd.android.cursor.item/user"; } }
```

### 14.3 3.ContentProvider 的主要方法

- `public boolean onCreate()`

ContentProvider 创建后 或 打开系统后其它应用第一次访问该 ContentProvider 时调用。

- `public Uri insert(Uri uri, ContentValues values)`

外部应用向 ContentProvider 中添加数据。

- `public int delete(Uri uri, String selection, String[] selectionArgs)`

外部应用从 ContentProvider 删除数据。

- `public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs):`

外部应用更新 ContentProvider 中的数据。

- `public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)`

供外部应用从 ContentProvider 中获取数据。

- `public String getType(Uri uri)`

该方法用于返回当前 Uri 所代表数据的 MIME 类型。

### 14.4 4.ContentResolver

ContentResolver 通过 URI 来查询 ContentProvider 中提供的数据。除了 URI 以外，还必须知道需要获取的数据段的名称，以及此数据段的数据类型。如果你需要获取一个特定的记录，你必须知道当前记录的 ID，也就是 URI 中 D 部分。ContentResolver 类提供了与 ContentProvider 类相同签名的四个方法：

- `public Uri insert(Uri uri, ContentValues values) //添加`
- `public int delete(Uri uri, String selection, String[] selectionArgs) //删除`
- 
- `public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) //更新`
- 
- `public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)//获取`

实例代码:  
`ContentResolver resolver = getContentResolver(); Uri uri = Uri.parse("content://cn.scu.mypr  
//添加一条记录 ContentValues values = new ContentValues(); values.put("name", "fanrunqi");  
values.put("age", 24); resolver.insert(uri, values); //获取 user 表中所有记录 Cursor cursor =  
resolver.query(uri, null, null, null, "userid desc"); while(cursor.moveToNext()){ //操作 } //把 id  
为 1 的记录的 name 字段值更改新为 finch ContentValues updateValues = new ContentValues();  
updateValues.put("name", "finch"); Uri updateIdUri = ContentUris.withAppendedId(uri, 1); re-  
solver.update(updateIdUri, updateValues, null, null); //删除 id 为 2 的记录 Uri deleteIdUri =  
ContentUris.withAppendedId(uri, 2); resolver.delete(deleteIdUri, null, null);`



## 14.5 5.ContentObserver

ContentObserver(内容观察者), 目的是观察特定 Uri 引起的数据库的变化, 继而做一些相应的处理, 它类似于数据库技术中的触发器 (Trigger), 当 ContentObserver 所观察的 Uri 发生变化时, 便会触发它。下面是使用内容观察者监听短信的例子: `public class MainActivity extends Activity {`

```
    @Override protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
```

```
    //注册观察者 Observer this.getContentResolver().registerContentObserver(Uri.parse("content://sms/"),
    SMSObserver(new Handler()));
```

```
    }
```

```
    private final class SMSObserver extends ContentObserver {
```

```
    public SMSObserver(Handler handler) { super(handler);
```

```
    }
```

```
    @Override public void onChange(boolean selfChange) {
```

```
        Cursor cursor = MainActivity.this.getContentResolver().query( Uri.parse("content://sms/inbox"),
        null, null, null, null);
```

```
        while (cursor.moveToNext()) { StringBuilder sb = new StringBuilder();
```

```
        sb.append("address=").append( cursor.getString(cursor.getColumnIndex("address")));
```

```
        sb.append(";subject=").append( cursor.getString(cursor.getColumnIndex("subject")));
```

```
        sb.append(";body=").append( cursor.getString(cursor.getColumnIndex("body")));
```

```
        sb.append(";time=").append( cursor.getLong(cursor.getColumnIndex("date")));
```

```
        System.out.println("——has Received SMS:." + sb.toString());
```

```
    }
```

```
    }
```

```
    } } 同时可以在 ContentProvider 发生数据变化时调用 getContentResolver().notifyChange(uri,
    null) 来通知注册在此 URI 上的访问者。public class UserContentProvider extends ContentProvider
    { public Uri insert(Uri uri, ContentValues values) { db.insert("user", "userid", values); getCon-
    text().getContentResolver().notifyChange(uri, null); } }
```

## 14.6 6. 实例说明

数据源是 SQLite, 用 ContentResolver 操作 ContentProvider。![这里写图片描述](<http://img.blog.csdn.net/20160505195143099>) Constant.java (储存一些常量) `public class Constant {`

```
    public static final String TABLE_NAME = "user";
```

```
    public static final String COLUMN_ID = "_id"; public static final String COLUMN_NAME
    = "name";
```

```
    public static final String AUTHORITY = "cn.scu.myprovider"; public static final int ITEM
    = 1; public static final int ITEM_ID = 2;
```

```
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/user"; public static
    final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/user";
```

```
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/user");
```

```
} DBHelper.java(操作数据库) public class DBHelper extends SQLiteOpenHelper {
```

```
    private static final String DATABASE_NAME = "finch.db"; private static final int DATABASE_VERSION
    = 1;
```

```
    public DBHelper(Context context) { super(context, DATABASE_NAME, null, DATABASE_VERSION)
    }
```

```
    @Override public void onCreate(SQLiteDatabase db) throws SQLException { //创建表格
    db.execSQL("CREATE TABLE IF NOT EXISTS " + Constant.TABLE_NAME + "(" + Constant.COLUMN_ID
    + " INTEGER PRIMARY KEY AUTOINCREMENT," + Constant.COLUMN_NAME + " VARCHAR
    NOT NULL);"); }
```

```

@Override public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
throws SQLException { //删除并创建表格 db.execSQL("DROP TABLE IF EXISTS "+ Constant.TABLE_NAME);
onCreate(db); } } MyProvider.java(自定义的 ContentProvider) public class MyProvider
extends ContentProvider {
    DBHelper mDbHelper = null; SQLiteDatabase db = null;
    private static final UriMatcher mMatcher; static{ mMatcher = new UriMatcher(UriMatcher.NO_MATCH);
mMatcher.addURI(Constant.AUTHORITY,Constant.TABLE_NAME, Constant.ITEM); mMatcher.addURI(
Constant.TABLE_NAME+"/#", Constant.ITEM_ID); }
    @Override public String getType(Uri uri) { switch (mMatcher.match(uri)) { case Constant.ITEM: return Constant.CONTENT_TYPE; case Constant.ITEM_ID: return Constant.CONTENT_TYPE; default: throw new IllegalArgumentException("Unknown URI"+uri); } }
    @Override public Uri insert(Uri uri, ContentValues values) { // TODO Auto-generated method stub long rowId; if(mMatcher.match(uri)!=Constant.ITEM){ throw new IllegalArgumentException("Unknown URI"+uri); } rowId = db.insert(Constant.TABLE_NAME,null,values); if(rowId>0){ Uri noteUri=ContentUris.withAppendedId(Constant.CONTENT_URI, rowId); getApplicationContext().getContentResolver().notifyChange(noteUri, null); return noteUri; }
        throw new SQLException("Failed to insert row into " + uri); }
    @Override public boolean onCreate() { // TODO Auto-generated method stub mDbHelper = new DBHelper(getApplicationContext());
        db = mDbHelper.getReadableDatabase();
        return true; }
    @Override public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) { // TODO Auto-generated method stub Cursor c = null; switch (mMatcher.match(uri)) { case Constant.ITEM: c = db.query(Constant.TABLE_NAME, projection, selection, selectionArgs, null, null, sortOrder); break; case Constant.ITEM_ID: c = db.query(Constant.TABLE_NAME, projection,Constant.COLUMN_ID + "=" +uri.getLastPathSegment(), selectionArgs, null, null, sortOrder); break; default: throw new IllegalArgumentException("Unknown URI"+uri); }
        c.setNotificationUri(getApplicationContext().getContentResolver(), uri); return c; }
    @Override public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) { // TODO Auto-generated method stub return 0; } @Override public int delete(Uri uri, String selection, String[] selectionArgs) { // TODO Auto-generated method stub return 0; }
} MainActivity.java(ContentResolver 操作) public class MainActivity extends Activity { private ContentResolver mContentResolver = null; private Cursor cursor = null; @Override protected void onCreate(Bundle savedInstanceState) { // TODO Auto-generated method stub super.onCreate(savedInstanceState); setContentView(R.layout.activity_main);
        TextView tv = (TextView) findViewById(R.id.tv);
        mContentResolver = getContentResolver(); tv.setText(" 添加初始数据"); for (int i = 0; i < 10; i++) { ContentValues values = new ContentValues(); values.put(Constant.COLUMN_NAME, "fanrunqi"+i); mContentResolver.insert(Constant.CONTENT_URI, values); }
        tv.setText(" 查询数据"); cursor = mContentResolver.query(Constant.CONTENT_URI, new String[]{Constant.COLUMN_ID,Constant.COLUMN_NAME}, null, null, null); if (cursor.moveToFirst()) { String s = cursor.getString(cursor.getColumnIndex(Constant.COLUMN_NAME)); tv.setText(" 第一个数据: "+s); } }
    } 最后在 manifest 申明 <provider android:name="MyProvider" android:authorities="cn.scu.myprovider" android:exported="true"/>
/> [本文中代码下载](http://download.csdn.net/detail/amazing7/9511234)

```

## 15 Android SQLite 的使用入门

### 15.1 概述

Android 也提供了几种方法用来保存数据,使得这些数据即使在程序结束以后依然不会丢失。这些方法有:

- 文本文件:

可以保存在应用程序自己的目录下, 安装的每个 app 都会在/data/data/目录下创建个文件夹, 名字和应用程序中 AndroidManifest.xml 文件中的 package 一样。

- SDcard 保存:
- Preferences 保存:

这也是一种经常使用的数据存储方法,因为它们对于用户而言是透明的,并且从应用安装的时候就存在了。

- Assets 保存:

用来存储一些只读数据, Assets 是指那些在 assets 目录下的文件, 这些文件在你将你的应用编译打包之前就要存在, 并且可以在应用程序运行的时候被访问到。但有时候我们需要对保存的数据进行一些复杂的操作, 或者数据量很大, 超出了文本文件和 Preference 的性能的范围, 所以需要一些更加高效的方法来管理, 从 Android1.5 开始, Android 就自带 SQLite 数据库了。SQLite 它是一个独立的, 无需服务进程, 支持事务处理, 可以使用 SQL 语言的数据库。

### 15.2 SQLite 的特性

#### 1、ACID 事务 ACID:

- 指数据库事务正确执行的四个基本要素的缩写。包含:原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)。一个支持事务(Transaction)的数据库, 必需具有这四种特性, 否则在事务过程(Transaction processing)当中无法保证数据的正确性, 交易过程极可能达不到交易方的要求。

2、零配置-无需安装和管理配置 3、储存在单一磁盘文件中的一个完整的数据库 4、数据库文件可以在不同字节顺序的机器间自由的共享 5、支持数据库大小至 2TB 6、足够小, 大致 3 万行 C 代码, 250K 7、比一些流行的数据库在大部分普通数据库操作要快 8、简单, 轻松的 API 9、包含 TCL 绑定, 同时通过 Wrapper 支持其他语言的绑定

- <http://www.sqlite.org/tclsqlite.html>

10、良好注释的源代码, 并且有着 90% 以上的测试覆盖率 11、独立: 没有额外依赖 12、Source 完全的 Open, 你可以用于任何用途, 包括出售它 13、支持多种开发语言, C, PHP, Perl, Java, ASP.NET, Python

### 15.3 Android 中使用 SQLite

Activites 可以通过 Content Provider 或者 Service 访问一个数据库。

### 15.3.1 创建数据库

Android 不自动提供数据库。在 Android 应用程序中使用 SQLite，必须自己创建数据库，然后创建表、索引，填充数据。Android 提供了 SQLiteOpenHelper 帮助你创建一个数据库，你只要继承 SQLiteOpenHelper 类根据开发应用程序的需要，封装创建和更新数据库使用的逻辑就行了。

SQLiteOpenHelper 的子类，至少需要实现三个方法：`public class DatabaseHelper extends SQLiteOpenHelper { /**`

- `@param context` 上下文环境（例如，一个 Activity）
- `@param name` 数据库名字
- `@param factory` 一个可选的游标工厂（通常是 Null）
- `@param version` 数据库模型版本的整数
- 
- 会调用父类 SQLiteOpenHelper 的构造函数

```
*/ public DatabaseHelper(Context context, String name, CursorFactory factory, int version) {  
    super(context, name, factory, version);  
} /**
```

- 在数据库第一次创建的时候会调用这个方法
- 

```
* 根据需要对传入的 SQLiteDatabase 对象填充表和初始化数据。*/ @Override public void onCreate(SQLiteDatabase db) { } /**
```

- 当数据库需要修改的时候（两个数据库版本不同），Android 系统会主动的调用这个方法。
- 一般我们在这个方法里边删除数据库表，并建立新的数据库表。

```
*/ @Override public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) { //三个参数，一个 SQLiteDatabase 对象，一个旧的版本号和一个新的版本号 } @Override public void  
onOpen(SQLiteDatabase db) { // 每次成功打开数据库后首先被执行 super.onOpen(db); } } 继承  
SQLiteOpenHelper 之后就拥有了以下两个方法：
```

- `getReadableDatabase()` 创建或者打开一个查询数据库
- `getWritableDatabase()` 创建或者打开一个可写数据库

`DatabaseHelper database = new DatabaseHelper(context);` //传入一个上下文参数 `SQLiteDatabase db = null; db = database.getWritableDatabase();` 上面这段代码会返回一个 `SQLiteDatabase` 类的实例，使用这个对象，你就可以查询或者修改数据库。`SQLiteDatabase` 类为我们提供了很多种方法，而较常用的方法如下：

- `(int) delete(String table, String whereClause, String[] whereArgs)`

删除数据行

- `(long) insert(String table, String nullColumnHack, ContentValues values)` 添加数据行
- `(int) update(String table, ContentValues values, String whereClause, String[] whereArgs)`

更新数据行

- (void) execSQL(String sql) 执行一个 SQL 语句，可以是一个 select 或其他 sql 语句
- (void) close() 关闭数据库
- (Cursor) query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

查询指定的数据表返回一个带游标的数据集。各参数说明：**table**：表名称 **columns**：列名称数组 **selection**：条件子句，相当于 where **selectionArgs**：条件语句的参数数组 **groupBy**：分组 **having**：分组条件 **orderBy**：排序类 **limit**：分页查询的限制 **Cursor**：返回值，相当于结果集 **ResultSet**

- (Cursor) rawQuery(String sql, String[] selectionArgs)

运行一个预置的 SQL 语句，返回带游标的数据集（与上面的语句最大的区别就是防止 SQL 注入）当你完成了对数据库的操作（例如你的 Activity 已经关闭），需要调用 SQLiteDatabase 的 Close() 方法来释放掉数据库连接。

### 15.3.2 创建表和索引

为了创建表和索引，需要调用 SQLiteDatabase 的 execSQL() 方法来执行 DDL 语句。如果没有异常，这个方法没有返回值。例如，你可以执行如下代码：db.execSQL("CREATE TABLE user(\_id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT, password TEXT);"); 这条语句会创建一个名为 user 的表，表有一个列名为 \_id，并且是主键，这列的值是会自动增长的整数。另外还有两列：username( 字符) 和 password( 字符)。SQLite 会自动为主键列创建索引。通常情况下，第一次创建数据库时创建了表和索引。要删除表和索引，需要使用 execSQL() 方法调用 DROP INDEX 和 DROP TABLE 语句。

### 15.3.3 添加数据

有两种方法可以给表添加数据。可以使用 execSQL() 方法执行 INSERT, UPDATE, DELETE 等语句来更新表的数据。execSQL() 方法适用于所有不返回结果的 SQL 语句。例如：String sql = "insert into user(username,password) values ('finch','123456');"; 插入操作的 SQL 语句 db.execSQL(sql); 执行 SQL 语句 "" 使用 SQLiteDatabase 对象的 insert()。ContentValues cv = new ContentValues(); cv.put("username","finch"); //添加用户名 cv.put("password","123456"); //添加密码 db.insert("user",null,cv); 行插入操作

### 15.3.4 更新数据（修改）

使用 SQLiteDatabase 对象的 update() 方法。ContentValues cv = new ContentValues(); cv.put("password","654321"); //添加要更改的字段及内容 String whereClause = "username=?"; //修改条件 String[] whereArgs = {"finch"}; //修改条件的参数 db.update("user",cv,whereClause,whereArgs); 行修改该方法有四个参数：表名；列名和值的 ContentValues 对象；可选的 WHERE 条件；可选的填充 WHERE 语句的字符串，这些字符串会替换 WHERE 条件中的 “?” 标记，update() 根据条件，更新指定列的值。使用 execSQL 方式的实现 String sql = "update [user] set password = '654321' where username='finch'"; //修改的 SQL 语句 db.execSQL(sql); //执行修改 ""

### 15.3.5 删除数据

使用 SQLiteDatabase 对象的 delete() 方法。String whereClause = "username=?"; //删除的条件 String[] whereArgs = {"finch"}; //删除的条件参数 db.delete("user",whereClause,whereArgs); //执行删除 使用 execSQL 方式的实现 String sql = "delete from user where username='finch'"; //删除操作的 SQL 语句 db.execSQL(sql); //执行删除操作

### 15.3.6 查询数据

使用 `rawQuery()` 直接调用 `SELECT` 语句 `Cursor c = db.rawQuery("select * from user where username=?", new String[]{"finch"}); if(cursor.moveToFirst()) { String password = c.getString(c.getColumnIndex("password")); ... }` 返回值是一个 `cursor` 对象，这个对象的方法可以迭代查询结果。如果查询是动态的，使用这个方法就会非常复杂。例如，当你需要查询的列在程序编译的时候不能确定，这时候使用 `query()` 方法会方便很多。通过 `query` 实现查询 `query()` 方法用 `SELECT` 语句构建查询。`SELECT` 语句内容作为 `query()` 方法的参数，比如：要查询的表名，要获取的字段名，`WHERE` 条件，包含可选的位置参数，去替代 `WHERE` 条件中位置参数的值，`GROUP BY` 条件，`HAVING` 条件。除了表名，其他参数可以是 `null`。所以代码可写成：`Cursor c = db.query("user", null, null, null, null, null, null); // 查询并获得游标 if(c.moveToFirst()) { // 判断游标是否为空 for(int i=0; i<c.getCount(); i++) { c.moveTo(i); // 移动到指定记录 String username = c.getString(c.getColumnIndex("username")); String password = c.getString(c.getColumnIndex("password")); } }`

1. 使用游标 不管你是否如何执行查询，都会返回一个 `Cursor`，这是 `Android` 的 `SQLite` 数据库游标，使用游标，你可以：

- 通过使用 `getCount()` 方法得到结果集中有多少记录；
- 通过 `moveToFirst()`, `moveToNext()`, 和 `isAfterLast()` 方法遍历所有记录；
- 通过 `getColumnNames()` 得到字段名；
- 通过 `getColumnIndex()` 转换成字段号；
- 通过 `getString()`, `getInt()` 等方法得到给定字段当前记录的值；
- 通过 `requery()` 方法重新执行查询得到游标；
- 通过 `close()` 方法释放游标资源；

例如，下面代码遍历 `user` 表：`Cursor result=db.rawQuery("SELECT _id, username, password FROM user"); result.moveToFirst(); while (!result.isAfterLast()) { int id=result.getInt(0); String name=result.getString(1); String password =result.getString(2); // do something useful with these result.moveToNext(); } result.close();`

## 16 Android 名企面试题及答案整理（一）

- 下面异常是属于 `Runtime Exception` 的是 (abcd)(多选)

A、`ArithmeticException`      B、`IllegalArgumentException`      C、`NullPointerException`  
D、`BufferUnderflowException`

解析：Java 提供了两类主要的异常：`runtime exception` 和 `checked exception`。`checked` 异常也就是我们经常遇到的 `IO` 异常，以及 `SQL` 异常都是这种异常。对于这种异常，`JAVA` 编译器强制要求我们必需对出现的这些异常进行 `catch`。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆 `catch` 块去处理可能的异常。出现运行时异常后，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，到最上层，如果是多线程就由 `Thread.run()` 抛出，如果是单线程就被 `main()` 抛出。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是 `Exception` 的子类，也有一般异常的特点，是可以被 `Catch` 块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。

编译时被检查的异常和运行时异常的区别：编译被检查的异常在函数内被抛出，函数必须要声明，否编译失败。声明的原因：是需要调用者对该异常进行处理。运行时异常如果在函数内被抛出，在函数上不需要声明。

- Math.round(11.5) 等于多少 ( ). Math.round(-11.5) 等于多少 (c)

A、11 ,-11 B、11 ,-12 C、12 ,-11 D、12 ,-12

解析: Math.ceil() 用作向上取整。Math.floor() 用作向下取整。Math.round() 我们数学中常用到的四舍五入取整。

- 对一些资源以及状态的操作保存, 最好是保存在生命周期的哪个函数中进行 (d)

A、 onPause() B、 onCreate() C、 onResume() D、 onStart() 解析:

![这里写图片描述](http://img.blog.csdn.net/20160411181000798) 系统杀死程序会调用 onSaveInstanceState(Bundle) 进行数据保存,这里保存的数据会出现在在程序下一次 onStart(Bundle) 这个 Bundle 中, onStart 时可以将 Bundle 中数据取出。

- Intent 传递数据时, 下列的数据类型哪些可以被传递 (abcd)(多选)

A、 Serializable B、 charsequence C、 Parcelable D、 Bundle 解析: Serializable : 将 Java 对象序列化为二进制文件的 Java 序列化技术是 Java 系列技术中一个较为重要的技术点, 在大部分情况下, 开发人员只需要了解被序列化的类需要实现 Serializable 接口, 使用 ObjectOutputStream 和 ObjectInputStream 进行对象的读写。charsequence : 实现了这个接口的类有: CharBuffer、String、StringBuffer、StringBuilder 这个四个类。CharBuffer 为 nio 里面用的一个类, String 实现这个接口理所当然, StringBuffer 也是一个 CharSequence, StringBuilder 是 Java 抄袭 C# 的一个类, 基本和 StringBuffer 类一样, 效率高, 但是不保证线程安全, 在不需要多线程的环境下可以考虑。提供这么一个接口, 有些处理 String 或者 StringBuffer 的类就不用重载了。但是这个接口提供的方法有限, 只有下面几个: charat、length、subSequence、toString 这几个方法, 感觉如果有必要, 还是重载的比较好, 避免用 instanceof 这个操作符。Parcelable : android 提供了一种新的类型: Parcel。本类被用作封装数据的容器, 封装后的数据可以通过 Intent 或 IPC 传递。除了基本类型以外, 只有实现了 Parcelable 接口的类才能被放入 Parcel 中。是 GOOGLE 在安卓中实现的另一种序列化, 功能和 Serializable 相似, 主要是序列化的方式不同 Bundle 是将数据传递到另一个上下文中或保存或回复你自己状态的数据存储方式。它的数据不是持久化状态。

- 下列属于 SAX 解析 xml 文件的优点的是 (b)

A、 将整个文档树在内存中, 便于操作, 支持删除, 修改, 重新排列等多种功能

B、不用事先调入整个文档, 占用资源少 C、整个文档调入内存, 浪费时间和空间 D、不是长久驻留在内存, 数据不是持久的, 事件过后, 若没有保存数据, 数据就会消失解析: 在 Android 中提供了三种解析 XML 的方式:SAX(Simple API XML),DOM(Document Object Model), 以及 Android 推荐的 Pull 解析方式。SAX: 是事件驱动型 XML 解析的一个标准接口, 简单地说就是对文档进行顺序扫描, 当扫描到文档 (document) 开始与结束、元素 (element) 开始与结束、文档 (document) 结束等地方时通知事件处理函数, 由事件处理函数做相应动作, 然后继续同样的扫描, 直至文档结束。DOM: 即对象文档模型, 它是将整个 XML 文档载入内存 (所以效率较低, 不推荐使用), 使用 DOM API 遍历 XML 树、检索所需的数据, 每一个节点当做一个对象。Pull: 运行方式与 SAX 解析器相似。它提供了类似的事件, SAX 解析器的工作方式是自动将事件推入事件处理器进行处理, 因此你不能控制事件的处理主动结束; 而 Pull 解析器的工作方式为允许你的应用程序代码主动从解析器中获取事件, 正因为是主动获取事件, 因此可以在满足了需要的条件后不再获取事件, 结束解析。pull 是一个 while 循环, 随时可以跳出, 而 sax 是只要解析了, 就必须解析完成。

- 在 android 中使用 Menu 时可能需要重写的方法有 (ac)。(多选)

A、 onCreateOptionsMenu()

B、 onCreateMenu() C、 onOptionsItemSelected() D、 onItemSelected() 解析: android 中有三种菜单

1. 选项菜单 Options menus : 一个 Activity 只能有一个选项菜单, 在按下 Menu 键时, 显示在屏幕下方。



重写 `onCreateContextMenu` 用以创建上下文菜单 重写 `onContextItemSelected` 用以响应上下文菜单 2. 上下文菜单 Context menus : 为 Activity 中的任何一个视图注册一个上下文菜单, “长按”出现。重写 `onCreateOptionsMenu` 用以创建选项菜单 重写 `onOptionsItemSelected` 用以响应选项菜单

3. 弹出式菜单 Popup menus : 依赖于 Activity 中的某个一个视图

- android 关于 service 生命周期的 `onCreate()` 和 `onStart()` 说法正确的是 (ad)(多选题)

A、当第一次启动的时候先后调用 `onCreate()` 和 `onStart()` 方法

B、当第一次启动的时候只会调用 `onCreate()` 方法 C、如果 service 已经启动, 将先后调用 `onCreate()` 和 `onStart()` 方法 D、如果 service 已经启动, 只会执行 `onStart()` 方法, 不在执行 `onCreate()` 方法  
解析: ![这里写图片描述](http://img.blog.csdn.net/20160411181016027) 1). 被启动的服务的生命周期: 如果一个 Service 被某个 Activity 调用 `Context.startService` 方法启动, 那么不管是否有 Activity 使用 `bindService` 绑定或 `unbindService` 解除绑定到该 Service, 该 Service 都在后台运行。如果一个 Service 被 `startService` 方法多次启动, 那么 `onCreate` 方法只会调用一次, `onStart` 将会被调用多次 (对应调用 `startService` 的次数), 并且系统只会创建 Service 的一个实例 (因此你应该知道只需要一次 `stopService` 调用)。该 Service 将会一直在后台运行, 而不管对应程序的 Activity 是否在运行, 直到被调用 `stopService`, 或自身的 `stopSelf` 方法。当然如果系统资源不足, android 系统也可能结束服务。2). 被绑定的服务的生命周期: 如果一个 Service 被某个 Activity 调用 `Context.bindService` 方法绑定启动, 不管调用 `bindService` 调用几次, `onCreate` 方法都只会调用一次, 同时 `onStart` 方法始终不会被调用。当连接建立之后, Service 将会一直运行, 除非调用 `Context.unbindService` 断开连接或者之前调用 `bindService` 的 Context 不存在了 (如 Activity 被 finish 的时候), 系统将会自动停止 Service, 对应 `onDestroy` 将被调用。3). 被启动又被绑定的服务生命周期: 如果一个 Service 又被启动又被绑定, 则该 Service 将会一直在后台运行。并且不管如何调用, `onCreate` 始终只会调用一次, 对应 `startService` 调用多少次, Service 的 `onStart` 便会调用多少次。调用 `unbindService` 将不会停止 Service, 而必须调用 `stopService` 或 Service 的 `stopSelf` 来停止服务。4). 当服务被停止时清除服务: 当一个 Service 被终止 (1、调用 `stopService`; 2、调用 `stopSelf`; 3、不再有绑定的连接 (没有被启动)) 时, `onDestroy` 方法将会被调用, 在这里你应当做一些清除工作, 如停止在 Service 中创建并运行的线程。特别注意: 1、你应当知道在调用 `bindService` 绑定到 Service 的时候, 你就应当保证在某处调用 `unbindService` 解除绑定 (尽管 Activity 被 finish 的时候绑定会自动解除, 并且 Service 会自动停止); 2、你应当注意使用 `startService` 启动服务之后, 一定要使用 `stopService` 停止服务, 不管你是否使用 `bindService`; 3、同时使用 `startService` 与 `bindService` 要注意到, Service 的终止, 需要 `unbindService` 与 `stopService` 同时调用, 才能终止 Service, 不管 `startService` 与 `bindService` 的调用顺序, 如果先调用 `unbindService` 此时服务不会自动终止, 再调用 `stopService` 之后服务才会停止, 如果先调用 `stopService` 此时服务也不会终止, 而再调用 `unbindService` 或者之前调用 `bindService` 的 Context 不存在了 (如 Activity 被 finish 的时候) 之后服务才会自动停止; 4、当在旋转手机屏幕的时候, 当手机屏幕在 “横” “竖” 变换时, 此时如果你的 Activity 如果会自动旋转的话, 旋转其实是 Activity 的重新创建, 因此旋转之前的使用 `bindService` 建立连接便会断开 (Context 不存在了), 对应服务生命周期与上述相同。5、在 sdk 2.0 及其以后的版本中, 对应的 `onStart` 已经被否决变为 `onStartCommand`, 不过之前的 `onStart` 任然有效。这意味着, 如果你开发的应用程序用的 sdk 为 2.0 及其以后的版本, 那么你应当使用 `onStartCommand` 而不是 `onStart`。

- 下面是属于 `GLSurfaceView` 特性的是 (abc)(多选)

A、管理一个 surface, 这个 surface 就是一块特殊的内存, 能直接排版到 android 的视图 view 上。

B、管理一个 `EGL display`, 它能让 `opengl` 把内容渲染到上述的 surface 上。C、让渲染器在独立的线程里运作, 和 UI 线程分离。D、可以直接从内存或者 DMA 等硬件接口取得图像数据解析: Android 游戏当中主要的除了控制类外就是显示类 `View`。 `SurfaceView` 是从 `View` 基类中派生出来的显示类。 android 游戏开发中常用的三种视图是: `view`、`SurfaceView` 和 `GLSurfaceView`。 `View`: 显示

视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在 UI 主线程内更新画面，速度较慢。**SurfaceView**：基于 view 视图进行拓展的视图类，更适合 2D 游戏的开发；是 view 的子类，类似使用双缓冲机制，在新的线程中更新画面所以刷新界面速度比 view 快。**GLSurfaceView**：基于 SurfaceView 视图再次进行拓展的视图类，专用于 3D 游戏开发的视图；是 SurfaceView 的子类，OpenGL 专用。**GLSurfaceView** 提供了下列特性：1. 管理一个 surface，这个 surface 就是一块特殊的内存，能直接排版到 android 的视图 view 上。2. 管理一个 EGL display，它能让 opengl 把内容渲染到上述的 surface 上。3. 用户自定义渲染器 (render)。4. 让渲染器在独立的线程里运作，和 UI 线程分离。5. 支持按需渲染 (on-demand) 和连续渲染 (continuous)。6. 一些可选工具，如调试。

- 关于 ContentValues 类说法正确的是 (a)

A、他和 Hashtable 比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是 String 类型，而值都是基本类型

B、他和 Hashtable 比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是任意类型，而值都是基本类型 C、他和 Hashtable 比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名，可以为空，而值都是 String 类型 D、他和 Hashtable 比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是 String 类型，而值也是 String 类型解析：ContentValues 和 HashTable 类似都是一种存储的机制但是两者最大的区别就在于，contentvalues Key 只能是 String 类型，values 只能存储基本类型的数据，像 string,int 之类的，不能存储对象这种东西。ContentValues 常用在数据库中的操作。HashMap 是 Hashtable 的轻量级实现（非线性安全的实现），他们都完成了 Map 接口，主要区别在于 HashMap 允许空（null）键值（key），由于非线性安全，效率上可能高于 Hashtable。HashMap 允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许。

- 下面退出 Activity 错误的方法是 (c) A、finish()

B、抛异常强制退出 C、System.exit() D、onStop() 解析：![这里写图片描述](http://img.blog.csdn.net/20160411181031423) finish()：在你的 activity 动作完成的时候，或者 Activity 需要关闭的时候，调用此方法。当你调用此方法的时候，系统只是将最上面的 Activity 移出了栈，并没有及时的调用 onDestory() 方法，其占用的资源也没有被及时释放。因为移出了栈，所以当你点击手机上面的“back”按键的时候，也不会再找到这个 Activity。finish 函数仅仅把当前 Activity 退出了，但是并没有释放他的资源。安卓系统自己决定何时从内存中释放应用程序。当系统没有可用内存到时候，会按照优先级，释放部分应用。onDestory()：系统销毁了这个 Activity 的实例在内存中占据的空间。在 Activity 的生命周期中，onDestory() 方法是他生命的最后一步，资源空间等就被回收了。当重新进入此 Activity 的时候，必须重新创建，执行 onCreate() 方法。System.exit(0)：退出整个应用程序（不仅仅是当前 activity）。将整个进程直接 Kill 掉。

- 关于 res/raw 目录说法正确的是 (a)

A、这里的文件是原封不动的存储到设备上不会转换为二进制的格式

B、这里的文件是原封不动的存储到设备上会转换为二进制的格式 C、这里的文件最终以二进制的格式存储到指定的包中 D、这里的文件最终不会以二进制的格式存储到指定的包中解析：res/raw 和 assets 的相同点：两者目录下的文件在打包后会原封不动的保存在 apk 包中，不会被编译成二进制。res/raw 和 assets 的不同点：1.res/raw 中的文件会被映射到 R.java 文件中，访问的时候直接使用资源 ID 即 R.id.filename；assets 文件夹下的文件不会被映射到 R.java 中，访问的时候需要 AssetManager 类。

- android 中常用的四个布局是 framlayout, linenarlayout, relativelayout 和 tablelayout。
- android 的四大组件是 activiey, service, broadcast 和 contentprovide。
- activity 一般会重载 7 个方法用来维护其生命周期，除了 onCreate(),onStart(),onDestory() 外还有 onpause,onresume,onstop, onrestart。
- android 的数据存储的方式 sharedpreference, 文件,SQLite,contentprovider, 网络。

- 程序运行的结果是：good and gbc

```
public class Example { String str = new String("good"); char[] ch = {'a','b','c'}; public static void
main(String args[]) { Example ex = new Example(); ex.change(ex.str, ex.ch); System.out.print(ex.str +
" and "); System.out.print(ex.ch); } public void change(String str, char ch[]) { str = "test ok"; ch[2] = 'g';
} } 解析： public void change(String str, char ch[]) str 是按值传递，所以在函数中对它的操作只生
效于它的副本，与原字符串无关。ch 是按址传递，在函数中根据地址，可以直接对字符串进行操作。
```

- 在 android 中，请简述 jni 的调用过程。1) 安装和下载 Cygwin，下载 Android NDK 2) 在 ndk 项目中 JNI 接口的设计 3) 使用 C/C++ 实现本地方法 4) JNI 生成动态链接库 .so 文件 5) 将动态链接库复制到 java 工程，在 java 工程中调用，运行 java 工程即可

- Android 应用程序结构：

Linux Kernel (Linux 内核)、Libraries (系统运行库或者是 c/c++ 核心库)、Application Framework (开发框架包)、Applications (核心应用程序)

![这里写图片描述](http://img.blog.csdn.net/20160411181237328)

- 请继承 SQLiteOpenHelper 实现创建一个版本为 1 的 “diaryOpenHelper.db” 的数据库，同时创建一个 “diary” 表 (包含一个 \_id 主键并自增长，topic 字符型 100 长度，content 字符型 1000 长度)，在数据库版本变化时请删除 diary 表，并重新创建出 diary 表。

```
public class DBHelper extends SQLiteOpenHelper { public final static String DATABASENAME
="diaryOpenHelper.db"; public final static int DATABASEVERSION = 1; //创建数据库 public DB-
Helper(Context context, String name, CursorFactory factory, int version) { super(context, name,
factory, version); } //创建表等机构性文件 public void onCreate(SQLiteDatabase db) { String sql
="create table diary (" + "_id integer primary key autoincrement," + "topic varchar(100)," +
"content varchar(1000)" + ")"; db.execSQL(sql); } //若数据库版本有更新，则调用此方法 public
void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) { String sql = "drop table
if exists diary"; db.execSQL(sql); this.onCreate(db); } }
```

- 页面上现有 ProgressBar 控件 progressBar，请用书写线程以 10 秒的时间完成其进度显示工作。

```
public class ProgressBarStu extends Activity { private ProgressBar progressBar = null;
protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); set-
ContentView(R.layout.progressbar); //从这到下是关键 progressBar = (ProgressBar)findViewById(R.id.p
Thread thread = new Thread(new Runnable() { @Override public void run() { int progress-
BarMax = progressBar.getMax(); try { while(progressBarMax != progressBar.getProgress()) {
int stepProgress = progressBarMax / 10; int currentProgress = progressBar.getProgress(); pro-
gressBar.setProgress(currentProgress + stepProgress); Thread.sleep(1000); } } catch (InterruptedException) { // TODO Auto-generated catch block e.printStackTrace(); } } }); thread.start(); //关键结束
} }
```

- onFreeze() renamed to onSaveInstanceState(), 以便恢复在 onCreate(Bundle) 里面设置的状态。
- 如果后台的 Activity 由于某原因被系统回收了，onSaveInstanceState() 在被系统回收之前 (onPause() 前面) 保存当前状态。

当你的程序中某一个 Activity A 在运行时，主动或被动地运行另一个新的 Activity B，这个时候 A 会执行 onSaveInstanceState()。B 完成以后又会来找 A，这个时候就有两种情况：一是 A 被回收，二是 A 没有被回收，被回收的 A 就要重新调用 onCreate() 方法，不同于直接启动的是这回 onCreate() 里是带上了参数 savedInstanceState；而没被回收的就直接执行 onResume()，跳过 onCreate() 了。

- **ContentProvider:**

提供了我们在应用程序之前共享数据的一种机制，而我们知道每一个应用程序都是运行在不同的应用程序的，数据和文件在不同应用程序之间达到数据的共享不是没有可能，而是显得比较复杂，而正好 Android 中的 ContentProvider 则达到了这一需求，比如有时候我们需要操作手机里的联系人，手机里的多媒体等一些信息，我们都可以用到这个 ContentProvider 来达到我们所需。1)、ContentProvider 为存储和获取数据提供了统一的接口。ContentProvide 对数据进行封装，不用关心数据存储的细节。使用表的形式来组织数据。

2)、使用 ContentProvider 可以在不同的应用程序之间共享数据。3)、Android 为常见的一些数据提供了默认的 ContentProvider (包括音频、视频、图片和通讯录等)。总的来说使用 ContentProvider 对外共享数据的好处是统一了数据的访问方式。Uri 为系统的每一个资源给其一个名字，比方说通话记录。每一个 ContentProvider 都拥有一个公共的 URI，这个 URI 用于表示这个 ContentProvider 所提供的数据。

- 请解释下 Android 程序运行时权限与文件系统权限的区别。运行时权限 Dalvik( android 授权) 文件系统 linux 内核授权

- 什么是 ANR 如何避免它?

在 Android 里，应用程序的响应性是由 Activity Manager 和 Window Manager 系统服务监视的。当它监测到以下情况中的一个时，Android 就会针对特定的应用程序显示 ANR：在 5 秒内没有响应输入的事件（例如，按键按下，屏幕触摸）BroadcastReceiver 在 10 秒内没有执行完毕。Android 应用程序通常是运行在一个单独的线程（例如，main）里。这意味着你的应用程序所做的事情如果在主线程里占用了太长的时间的话，就会引发 ANR 对话框，因为你的应用程序并没有给自己机会来处理输入事件或者 Intent 广播。在主线程里尽量少的做事情，比如高耗时的计算和网络、数据库等潜在的耗时操作都应该放在子线程来完成。

## 17 AIDL 的使用情况和实例介绍

### 17.1 AIDL 是什么?

- AIDL (Android Interface Definition Language)，Android 接口定义语言，Android 提供的 IPC (Inter Process Communication，进程间通信) 的一种独特实现。

### 17.2 什么情况下要使用 AIDL

- 使用 AIDL 只有在你允许来自不同应用的客户端跨进程通信访问你的 service，并且想要在你的 service 种处理 \*\*多线程\*\* 的时候才是必要的。如果你不需要执行不同应用之间的 IPC 并发，你应该通过实现 Binder 建立你的接口，或者如果你想执行 IPC，但是不需要处理多线程。那么使用 Messenger 实现你的接口。

### 17.3 定义一个 AIDL 接口的步骤

- 必须在一个 .aidl 文件中使用 java 编程语言语法定义你的 AIDL 接口，然后在提供 service 的应用中和任何绑定到这个 service 的应用中的源代码中（在 src 目录下）保存它。
- 当你编译包含 .aidl 文件的应用时，Android SDK 工具基于这个 .aidl 文件生成一个 IBinder 接口，并且把它保存到项目的 gen 目录下。service 必须恰当的实现这个 IBinder 接口之后客户端应用可以绑定到这个服务上，然后从 IBinder 调用方法来执行 IPC。
- 使用 AIDL 建立一个邻接的 service 需要遵循下面的步骤：

- 1. 建立 .aidl 文件

- \* 这个文件使用方法签名定义了语言接口
- 2. 实现这个接口
  - \* Android SDK 工具基于你的.aidl 文件使用 java 语言生成一个接口这个接口有一个内部抽象类，叫做 Stub，它是继承 Binder 并且实现你 AIDL 接口的你必须继承这个 Stub 类并且实现这些方法
- 3. 暴露这个接口给客户端
  - \* 实现一个 service 并且覆盖 onBind() 方法返回你的 Stub 实现类。
- 你的.aidl 文件必须被复制到其他应用程序中来让他们访问你 service 的接口，你必须维护原始接口的支持（向后兼容）。

## 17.4 用一个实例来分步骤说明

### 17.4.1 在 server 项目中建立.aidl 文件

![这里写图片描述](<http://img.blog.csdn.net/20160504180944041>)

- AIDL 使用一个简单的语法让你声明一个带有一个或者多个带有参数和返回值方法的接口参数和返回值可以是任何类型，甚至是 AIDL 生成的接口。
- IService.aidl

---

```
package com.example.aidl;
interface IService {
    String hello(String name);
}
```

---

### 17.4.2 在 server 项目中建立服务类

- 当你编译你的应用时，Android SDK 工具生成一个.java 接口文件用你的.aidl 文件命名生成的接口包含一个名字为 Stub 的子类，这是一个它父类的抽象实现，并且声明了.aidl 中所有的方法。
- Stub 也定义了一些辅助的方法，最显著的就是 asInterface()，它是用来接收一个 IBinder（通常 IBinder 传递给客户端的 onServiceConnected() 回调方法）并且返回一个 Stub 接口的实例。
- 一旦你为 service 实现了接口，你需要把它暴露给客户端，这样他们才能绑定到上面为了给你的 service 暴露接口，继承 Service 并且实现 onBind() 方法返回一个你实现生成的 Stub 类。
- AIDLService.java

---

```
public class AIDLService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public IBinder onBind(Intent intent) {
        // Return the interface
        return new IService.Stub() {
            @Override
            public String hello(String name) throws RemoteException {
                // TODO Auto-generated method stub
                return "hello"+name;
            }
        };
    }
}
```

---

### 17.4.3 在 **server** 项目 **AndroidManifest** 中申明 **Service**

```
<service
    android:name="com.example.service.AIDLService" >
    <intent-filter>
        <action android:name="android.intent.action.AIDLService" />
    </intent-filter>
</service>
```

### 17.4.4 把 **server** 项目中的 **aidl** 文件带包拷贝到 **client** 项目中（包名要相同）

![这里写图片描述](http://img.blog.csdn.net/20160504181850678)

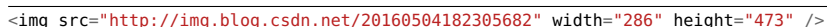
- MainActivity.java

```
public class MainActivity extends Activity {
    IService RemoteService; // 监听服务
    private ServiceConnection mConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            // TODO Auto-generated method stub
            Log.i("mConnection", service+"");
            RemoteService = IService.Stub.asInterface(service);

            try {
                String s= RemoteService.hello("finch");
                Toast.makeText(MainActivity.this, s, Toast.LENGTH_LONG).show();
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void onServiceDisconnected(ComponentName name) {
            // TODO Auto-generated method stub
        }
    };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initService();
    }
    // 连接服务
    private void initService() {
        Intent i = new Intent( );
        i.setAction("android.intent.action.AIDLService");
        boolean ret = bindService(i, mConnection, Context.BIND_AUTO_CREATE);
    }
    // 断开服务
    private void releaseService() {
        unbindService(mConnection);
        mConnection = null;
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        releaseService();
    }
}
```

- 运行结果:



[文章中 AIDL 例子代码下载](http://download.csdn.net/detail/amazing7/9510133)