

面试题精编版.....	7
一、 Activity.....	7
1、 什么是 Activity?.....	7
2、 请描述一下 Activity 生命周期.....	7
3、 常见的 Activity 类型有 FragmentActivitiy, ListActivity, TabAcitivty 等。请描述一下 Activity 生命周期.....	7
4、 如何保存 Activity 的状态?	8
5、 两个 Activity 之间跳转时必然会执行的是哪几个方法?	9
6、 横竖屏切换时 Activity 的生命周期.....	9
7、 如何将一个 Activity 设置成窗口的样式.....	9
8、 如何退出 Activity? 如何安全退出已调用多个 Activity 的 Application?	10
9、 Activity 的四种启动模式，singletop 和 singletask 区别是什么？一般书签的使用模式是 singletop，那为什么不使用 singletask?.....	11
10、 Android 中的 Context, Activity, Appliction 有什么区别?	11
11、 两个 Activity 之间传递数据，除了 intent，广播接收者，content provider 还有啥?	12
12、 Context 是什么?	12
二、 Service.....	13
1、 Service 是否在 main thread 中执行, service 里面是否能执行耗时的操作?.....	13
2、 Activity 怎么和 Service 绑定，怎么在 Activity 中启动自己对应的 Service?	13
3、 请描述一下 Service 的生命周期.....	14

4、 什么是 IntentService ? 有何优点?	15
5、 说说 Activity 、 Intent 、 Service 是什么关系.....	18
6、 Service 和 Activity 在同一个线程吗.....	18
7、 Service 里面可以弹吐司么.....	19
8、 什么是 Service 以及描述下它的生命周期。 Service 有哪些启动方法，有什么区别，怎样停用 Service ?	19
9、 在 service 的生命周期方法 onstartConmand() 可不可以执行网络操作? 如何在 service 中执行网络操作?	20
三、 Broadcast Receiver	20
1、 请描述一下 BroadcastReceiver	20
2、 在 manifest 和代码中如何注册和使用 BroadcastReceiver	21
3、 BroadCastReceiver 的生命周期.....	21
四、 ContentProvider	22
1、 请介绍下 ContentProvider 是如何实现数据共享的.....	22
2、 请介绍下 Android 的数据存储方式.....	23
3、 为什么要用 ContentProvider ? 它和 sql 的实现上有什么差别?	23
4、 说说 ContentProvider 、 ContentResolver 、 ContentObserver 之间的关系.....	23
五、 ListView	24
1、 ListView 如何提高其效率?	24
2、 当 ListView 数据集改变后，如何更新 ListView	24
3、 ListView 如何实现分页加载.....	24

4、	ListView 可以显示多种类型的条目吗.....	25
5、	ListView 如何定位到指定位置.....	26
6、	如何在 ScrollView 中如何嵌入 ListView.....	26
7、	ListView 中如何优化图片.....	27
8、	ListView 中图片错位的问题是如何产生的.....	29
9、	如何刷新 ListView 中单个 item 的数据，不刷新整个 ListView 的数据？	29
六、	Intent	29
1、	Intent 传递数据时，可以传递哪些类型数据？	29
2、	Serializable 和 Parcelable 的区别.....	30
3、	请描述一下 Intent 和 IntentFilter.....	31
七、	Fragment	33
1、	Fragment 跟 Activity 之间是如何传值的.....	33
2、	描述一下 Fragment 的生命周期.....	34
3、	Fragment 的 replace 和 add 方法的区别.....	35
4、	Fragment 如何实现类似 Activity 栈的压栈和出栈效果的？	36
5、	Fragment 在你们项目中的使用.....	37
6、	如何切换 fragement,不重新实例化.....	38
	Android 高级 (★★★)	40
一、	Android 性能优化 (11.9 更新)	40
1、	如何对 Android 应用进行性能分析.....	40
2、	什么情况下会导致内存泄露.....	47

3、 如何避免 OOM 异常.....	51
4、 Android 中如何捕获未捕获的异常.....	54
5、 ANR 是什么？怎样避免和解决 ANR（重要）	57
6、 Android 线程间通信有哪几种方式（重要）	59
7、 Devik 进程，linux 进程，线程的区别.....	59
8、 描述一下 android 的系统架构？	60
9、 android 应用对内存是如何限制的?我们应该如何合理使用内存？（2016.01.24）	61
10、 简述 android 应用程序结构是哪些？（2016.01.24）	62
11、 请解释下 Android 程序运行时权限与文件系统权限的区别？（2016.01.24）	65
12、 Framework 工作方式及原理，Activity 是如何生成一个 view 的，机制是什么？ （2016.01.24）	66
13、 多线程间通信和多进程之间通信有什么不同，分别怎么实现？（2016.01.24）	67
二、 Android 屏幕适配.....	68
1、 屏幕适配方式都有哪些.....	68
2、 屏幕适配的处理技巧都有哪些.....	75
3、 dp 和 px 之间的关系.....	78
三、 AIDL.....	79
1、 什么是 AIDL 以及如何使用.....	79
2、 AIDL 的全称是什么?如何工作?能处理哪些类型的数据？	80
四、 Android 中的事件处理.....	80
1、 Handler 机制.....	80

2、 事件分发机制.....	81
3、 子线程发消息到主线程进行更新 UI，除了 handler 和 AsyncTask，还有什么？	84
4、 子线程中能不能 new handler？为什么？	85
五、 Android 中的动画.....	86
1、 Android 中的动画有哪几类，它们的特点和区别是什么.....	86
2、 如何修改 Activity 进入和退出动画.....	86
3、 属性动画，例如一个 button 从 A 移动到 B 点，B 点还是可以响应点击事件，这个原理是什么？	87
六、 ContentObserver 内容观察者作用及特点.....	87
项目框架的使用（★★★）	92
一、 自我介绍.....	92
二、 开发中都使用过哪些框架、平台.....	92
三、 都使用过哪些自定义控件.....	98
四、 自定义控件：绘制圆环的实现过程.....	99
五、 自定义控件：摩天轮的实现过程.....	104
六、 自定义控件：可拖拽排序的 GridLayout 的实现过程.....	105
七、 流式布局的实现过程.....	106
八、 项目的流程.....	108
九、 项目中常见的问题（11.9 更新）	109
十、 即时通讯是是怎么做的？.....	126
十一、 设计模式六大原则.....	136

十二、 第三方登陆.....	138
十三、 第三方支付.....	140
十四、 常见框架分析.....	141
Java 面试题（10.23 更新）（★★）	159
一、 Java 基础（★★）	159
1、 Java 中引用类型都有哪些.....	159
2、 什么是重载，什么是重写，有什么区别？	160
3、 String、StringBuffer 和 StringBuilder 的区别	160
4、 关键字 final 和 static 是怎么使用的	161
5、 TCP/IP 协议簇分哪几层？ TCP、IP、XMPP、HTTP、分别属于哪一层？（2016.01.24）	163
二、 Java 中的设计模式.....	163
1、 你所知道的设计模式有哪些.....	163
2、 单例设计模式.....	164
3、 工厂设计模式.....	165
4、 建造者模式（Builder）	169
5、 适配器设计模式.....	170
6、 装饰模式（Decorator）	173
7、 策略模式（strategy）	174
8、 观察者模式（Observer）	176

Android 基础（2016.01.20 更新）（★★）**一、 Activity****1、什么是 Activity?**

四大组件之一,一般的,一个用户交互界面对应一个 activity

`setContentView()` ,// 要显示的布局

`button.setOnClickListener{`

`}, activity 是 Context 的子类,同时实现了 window.callback 和 keyevent.callback, 可以处理与窗体用户交互的事件.`

我开发常用的有 `FragmentActivity` `ListActivity` ,`PreferenceActivity` ,`TabActivity` 等...

2、请描述一下 Activity 生命周期

Activity 从创建到销毁有多种状态,从一种状态到另一种状态时会激发相应的回调方法,

这些回调方法包括: `onCreate` `onStart` `onResume` `onPause` `onStop` `onDestroy`

其实这些方法都是两两对应的, `onCreate` 创建与 `onDestroy` 销毁;

`onStart` 可见与 `onStop` 不可见; `onResume` 可编辑(即焦点)与 `onPause`;

如果界面有共同的特点或者功能的时候,还会自己定义一个 `BaseActivity`.
进度对话框的显示与销毁

3、常见的 Activity 类型有 FragmentActivity, ListActivity, TabActivity 等。请

描述一下 **Activity** 生命周期

Activity 从创建到销毁有多种状态,从一种状态到另一种状态时会激发相应的回调方法,

这些回调方法包括: `onCreate` `onStart` `onResume` `onPause` `onStop` `onDestroy`

其实这些方法都是两两对应的, `onCreate` 创建与 `onDestroy` 销毁;

onStart 可见与 onStop 不可见；onResume 可编辑（即焦点）与 onPause。

4、如何保存 Activity 的状态？

Activity 的状态通常情况下系统会自动保存的，只有当我们需要保存额外的数据时才需要使用到这样的功能。

一般来说，调用 onPause()和 onStop()方法后的 activity 实例仍然存在于内存中，activity 的所有信息和状态数据不会消失，当 activity 重新回到前台之后，所有的改变都会得到保留。

但是当系统内存不足时，调用onPause()和onStop()方法后的activity可能会被系统摧毁，此时内存中就不会存有该 activity 的实例对象了。如果之后这个 activity 重新回到前台，之前所作的改变就会消失。为了避免此种情况的发生，我们可以覆写onSaveInstanceState()方法。onSaveInstanceState()方法接受一个 Bundle 类型的参数，开发者可以将状态数据存储到这个 Bundle 对象中，这样即使 activity 被系统摧毁，当用户重新启动这个 activity 而调用它的 onCreate()方法时，上述的 Bundle 对象会作为实参传递给 onCreate()方法，开发者可以从 Bundle 对象中取出保存的数据，然后利用这些数据将 activity 恢复到被摧毁之前的状态。

需要注意的是，onSaveInstanceState()方法并不是一定会被调用的，因为有些场景是不需要保存状态数据的。比如用户按下 BACK 键退出 activity 时，用户显然想要关闭这个 activity，此时是没有必要保存数据以供下次恢复的，也就是 onSaveInstanceState()方法不会被调用。如果调用 onSaveInstanceState()方法，调用将发生在 onPause()或 onStop()方法之前。


```
@Override
protected void onSaveInstanceState(Bundle outState) {
    // TODO Auto-generated method stub
    super.onSaveInstanceState(outState);
}
```

5、两个 **Activity** 之间跳转时必然会执行的是哪几个方法？

一般情况下比如说有两个 **activity**, 分别叫 A,B, 当在 A 里面激活 B 组件的时候, A 会调用 **onPause()** 方法, 然后 B 调用 **onCreate()**, **onStart()**, **onResume()**。

这个时候 B 覆盖了窗体, A 会调用 **onStop()** 方法。如果 B 是个透明的, 或者是对话框的样式, 就不会调用 A 的 **onStop()** 方法。

6、横竖屏切换时 **Activity** 的生命周期

此时的生命周期跟清单文件里的配置有关系。

1. 不设置 **Activity** 的 **android:configChanges** 时, 切屏会重新调用各个生命周期默认首先销毁当前 **activity**, 然后重新加载。

2. 设置 **Activity**

android:configChanges="orientation|keyboardHidden|screenSize" 时, 切屏不会重新调用各个生命周期, 只会执行 **onConfigurationChanged** 方法。

通常在游戏开发, 屏幕的朝向都是写死的。

7、如何将一个 **Activity** 设置成窗口的样式

只需要给我们的 **Activity** 配置如下属性即可。

android:theme="@android:style/Theme.Dialog"

8、如何退出 **Activity**? 如何安全退出已调用多个 **Activity** 的 **Application**?

1、通常情况用户退出一个 **Activity** 只需按返回键，我们写代码想退出 **activity** 直接调用 **finish()**方法就行。

2、记录打开的 **Activity**:

每打开一个 **Activity**，就记录下来。在需要退出时，关闭每一个 **Activity** 即可。

```
//伪代码
List<Activity> lists ;//在 application全局的变量里面
lists = new ArrayList<Activity>();
lists.add(this);
for(Activity activity: lists)
{
    activity.finish();
}
lists.remove(this);
```

3、发送特定广播:

在需要结束应用时，发送一个特定的广播，每个 **Activity** 收到广播后，关闭即可。

//给某个 **activity** 注册接受接受广播的意图

```
registerReceiver(receiver, filter)
```

//如果过接受到的是 关闭 **activity** 的广播 就调用 **finish()**方法 把当前的 **activity** **finish()**掉

4、递归退出

在 打开新的 **Activity** 时使用 **startActivityForResult**，然后自己加标志，在 **onActivityResult** 中处理，递归关闭。

5、其实 也可以通过 **intent** 的 **flag** 来实现

intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP)激活一个新的 **activity**。此时如果该

任务栈中已经有该 **Activity**，那么系统会把这个 **Activity** 上面的所有 **Activity** 干掉。其实相当于给 **Activity** 配置的启动模式为 **SingleTop**。

9、**Activity** 的四种启动模式，**singletop** 和 **singletask** 区别是什么？一般书签的使用模式是 **singletop**，那为什么不使用 **singletask**？

singleTop 跟 **standard** 模式比较类似。唯一的区别就是，当跳转的对象是位于栈顶的 **activity**（应该可以理解为用户眼前所看到的 **activity**）时，程序将不会生成一个新的 **activity** 实例，而是直接跳到现存于栈顶的那个 **activity** 实例。拿上面的例子来说，当 **Act1** 为 **singleTop** 模式时，执行跳转后栈里面依旧只有一个实例，如果现在按返回键程序将直接退出。

singleTask 模式和 **singleInstance** 模式都是只创建一个实例的。在这种模式下，无论跳转的对象是不是位于栈顶的 **activity**，程序都不会生成一个新的实例（当然前提是栈里面已经有这个实例）。这种模式相当有用，在以后的多 **activity** 开发中，常会因为跳转的关系导致同个页面生成多个实例，这个在用户体验上始终有点不好，而如果你将对应的 **activity** 声明为 **singleTask** 模式，这种问题将不复存在。在主页的 **Activity** 很常用

10、**Android** 中的 **Context**，**Activity**，**Appliction** 有什么区别？

相同：**Activity** 和 **Application** 都是 **Context** 的子类。

Context 从字面上理解就是上下文的意思，在实际应用中它也确实是起到了管理上下文环境中各个参数和变量的总用，方便我们可以简单的访问到各种资源。

不同：维护的生命周期不同。**Context** 维护的是当前的 **Activity** 的生命周期，**Application** 维护的是整个项目的生命周期。

使用 **context** 的时候，小心内存泄露，防止内存泄露，注意一下几个方面：

1. 不要让生命周期长的对象引用 **activity context**，即保证引用 **activity** 的对象要与 **activity** 本身生命周期是一样的。
2. 对于生命周期长的对象，可以使用 **application, context**。
3. 避免非静态的内部类，尽量使用静态类，避免生命周期问题，注意内部类对外部对象引用导致的生命周期变化。

11、两个 **Activity** 之间传递数据，除了 **intent**，广播接收者，**content provider** 还有啥？

- 1) 利用 **static** 静态数据，**public static** 成员变量
- 2) 利用外部存储的传输，

例如 **File** 文件存储

SharedPreferences 首选项

Sqlite 数据库

12、**Context** 是什么？

- 1、它描述的是一个应用程序环境的信息，即上下文。

2、该类是一个抽象(**abstract class**)类，**Android** 提供了该抽象类的具体实现类 (**ContextImpl**)。

3、通过它我们可以获取应用程序的资源 and 类，也包括一些应用级别操作，例如：启动一个 **Activity**，发送广播，接受 **Intent**，信息，等。

二、 **Service**

1、**Service** 是否在 **main thread** 中执行, **service** 里面是否能执行耗时的操作?

默认情况,如果没有显示的指 **servic** 所运行的进程, **Service** 和 **activity** 是运行在当前 **app** 所在进程的 **main thread**(**UI** 主线程)里面。

service 里面不能执行耗时的操作(网络请求,拷贝数据库,大文件)

特殊情况 ,可以在清单文件配置 **service** 执行所在的进程 ,让 **service** 在另外的进程中执行

```
<service
    android:name="com.baidu.location.f"
    android:enabled="true"
    android:process=":remote" >
</service>
```

2、**Activity** 怎么和 **Service** 绑定，怎么在 **Activity** 中启动自己对应的 **Service**?

Activity 通过 **bindService(Intent service, ServiceConnection conn, int flags)** 跟 **Service** 进行绑定，当绑定成功的时候 **Service** 会将代理对象通过回调的形式传给 **conn**，

这样我们就拿到了 **Service** 提供的服务代理对象。

在 **Activity** 中可以通过 **startService** 和 **bindService** 方法启动 **Service**。一般情况下如果想获取 **Service** 的服务对象那么肯定需要通过 **bindService**（）方法，比如音乐播放器，第三方支付等。如果仅仅只是为了开启一个后台任务那么可以使用 **startService**（）方法。

3、请描述一下 **Service** 的生命周期

Service 有绑定模式和非绑定模式，以及这两种模式的混合使用方式。不同的使用方法生命周期方法也不同。

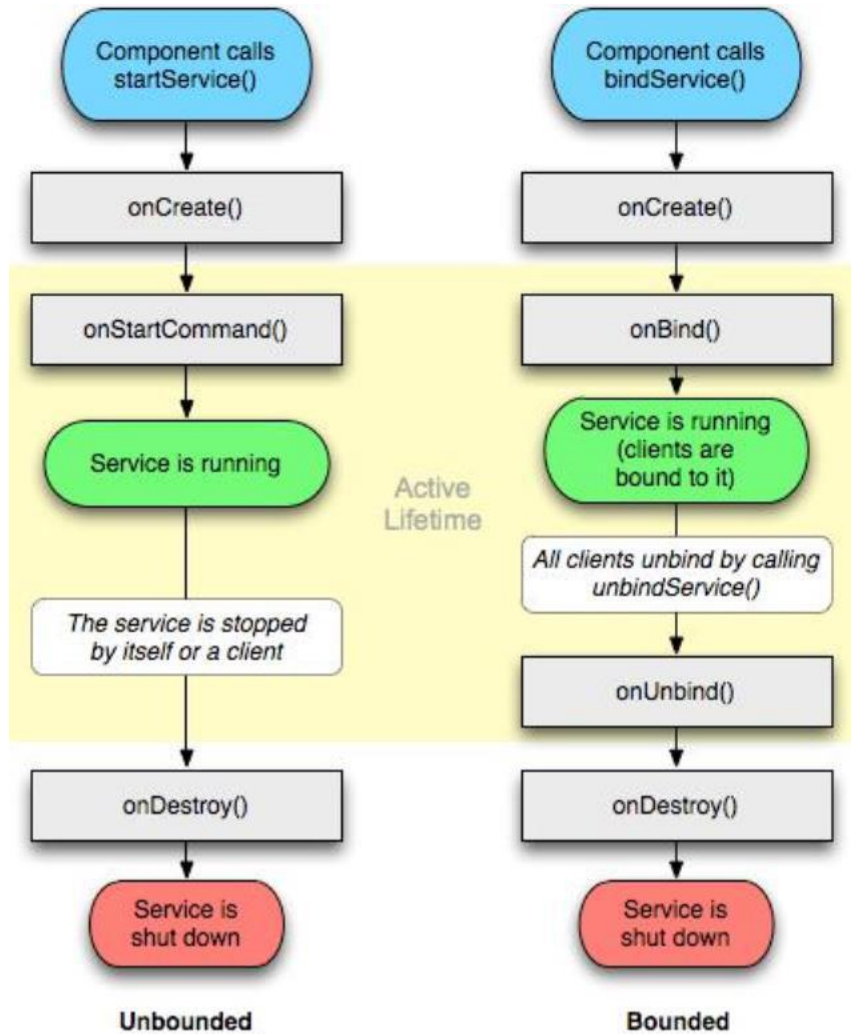
非绑定模式：当第一次调用 **startService** 的时候执行的方法依次为 **onCreate()**、**onStartCommand()**，当 **Service** 关闭的时候调用 **onDestory** 方法。

绑定模式：第一次 **bindService**（）的时候，执行的方法为 **onCreate()**、**onBind()** 解除绑定的时候会执行 **onUnbind()**、**onDestory()**。

上面的两种生命周期是在相对单纯的模式下的情形。我们在开发的过程中还必须注意 **Service** 实例只会有一个，也就是说如果当前要启动的 **Service** 已经存在了那么就不会再次创建该 **Service** 当然也不会调用 **onCreate**（）方法。

一个 **Service** 可以被多个客户进行绑定，只有所有的绑定对象都执行了 **onBind**（）方法后该 **Service** 才会销毁，不过如果有一个客户执行了 **onStart()**方法，那么这个时候如果所有的 **bind** 客户都执行了 **unBind()**该 **Service** 也不会销毁。

Service 的生命周期图如下所示，帮助大家记忆。



4、什么是 `IntentService`? 有何优点?

我们通常只会使用 `Service`，可能 `IntentService` 对大部分同学来说都是第一次听说。那么看了下面的介绍相信你就不再陌生了。如果你还是不了解那么在面试的时候你就坦诚说没用过或者不了解等。并不是所有的问题都需要回答上来的。

一、`IntentService` 简介

`IntentService` 是 `Service` 的子类，比普通的 `Service` 增加了额外的功能。先看 `Service` 本身存在两个问题：

`Service` 不会专门启动一条单独的进程，`Service` 与它所在应用位于同一个进程中；

Service 也不是专门一条新线程，因此不应该在 Service 中直接处理耗时的任务；

二、IntentService 特征

会创建独立的 worker 线程来处理所有的 Intent 请求；

会创建独立的 worker 线程来处理 onHandleIntent()方法实现的代码，无需处理多线程问题；

所有请求处理完后，IntentService 会自动停止，无需调用 stopSelf()方法停止

Service；

为 Service 的 onBind()提供默认实现，返回 null；

为 Service 的 onStartCommand 提供默认实现，将请求 Intent 添加到队列中；

使用 IntentService

本人写了一个 IntentService 的使用例子供参考。该例子中一个 MainActivity 一个 MyIntentService，这两个类都是四大组件当然需要在清单文件中注册。这里只给出核心代码：

MainActivity.java:

```
public void click(View view){
    Intent intent = new Intent(this, MyIntentService.class);
    intent.putExtra("start", "MyIntentService");
    startService(intent);
}
```

MyIntentService.java


```
public class MyIntentService extends IntentService {

    private String ex = "";
    private Handler mHandler = new Handler() {

        public void handleMessage(android.os.Message msg) {
            Toast.makeText(MyIntentService.this, "-e " + ex,
                Toast.LENGTH_LONG).show();
        }
    };

    public MyIntentService(){
        super("MyIntentService");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        ex = intent.getStringExtra("start");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        /**
         *模拟执行耗时任务
         *该方法是在子线程中执行的，因此需要用到 handler跟主线程进行通信
         */
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        mHandler.sendEmptyMessage(0);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

运行后效果如下：

7、Service 里面可以弹吐司么

可以的。弹吐司有个条件就是得有一个 **Context** 上下文，而 **Service** 本身就是 **Context** 的子类，因此在 **Service** 里面弹吐司是完全可以的。比如我们在 **Service** 中完成下载任务后可以弹一个吐司通知用户。

8、什么是 Service 以及描述下它的生命周期。Service 有哪些启动方法，有什么区别，怎样停用 Service?

在 **Service** 的生命周期中，被回调的方法比 **Activity** 少一些，只有 **onCreate**, **onStart**, **onDestroy**, **onBind** 和 **onUnbind**。

通常有两种方式启动一个 **Service**，他们对 **Service** 生命周期的影响是不一样的。

1. 通过 **startService**

Service 会经历 **onCreate** 到 **onStart**，然后处于运行状态，**stopService** 的时候调用 **onDestroy** 方法。

如果是调用者自己直接退出而没有调用 **stopService** 的话，**Service** 会一直在后台运行。

2. 通过 **bindService**

Service 会运行 **onCreate**，然后是调用 **onBind**，这个时候调用者和 **Service** 绑定在一起。调用者退出了，**Service** 就会调用 **onUnbind->onDestroyed** 方法。

所谓绑定在一起就共存亡了。调用者也可以通过调用 **unbindService** 方法来停止服务，这时候 **Service** 就会调用 **onUnbind->onDestroyed** 方法。

需要注意的是如果这几个方法交织在一起的话，会出现什么情况呢？

一个原则是 **Service** 的 **onCreate** 的方法只会被调用一次，就是你无论多少次的 **startService** 又 **bindService**，**Service** 只被创建一次。

如果先是 **bind** 了，那么 **start** 的时候就直接运行 **Service** 的 **onStart** 方法，如果先是 **start**，那么 **bind** 的时候就直接运行 **onBind** 方法。

如果 **service** 运行期间调用了 **bindService**，这时候再调用 **stopService** 的话，**service** 是不会调用 **onDestroy** 方法的，**service** 就 **stop** 不掉了，只能调用 **UnbindService**，**service** 就会被销毁

如果一个 **service** 通过 **startService** 被 **start** 之后，多次调用 **startService** 的话，**service** 会多次调用 **onStart** 方法。多次调用 **stopService** 的话，**service** 只会调用一次 **onDestroyed**

方法。

如果一个 **service** 通过 **bindService** 被 **start** 之后，多次调用 **bindService** 的话，**service** 只会调用一次 **onBind** 方法。

多次调用 **unbindService** 的话会抛出异常。

9、在 **service** 的生命周期方法 **onstartConmand()**可不可以执行网络操作？如何在 **service** 中执行网络操作？

可以直接在 **Service** 中执行网络操作,在 **onStartCommand()**方法中可以执行网络操作

三、 Broadcast Receiver

1、请描述一下 BroadcastReceiver

BroadCastReceiver 是 **Android** 四大组件之一，主要用于接收系统或者 **app** 发送的广播事件。

广播分两种：有序广播和无序广播。

内部通信实现机制：通过 **Android** 系统的 **Binder** 机制实现通信。

无序广播：完全异步，逻辑上可以被任何广播接收者接收到。优点是效率较高。缺点是一个接收者不能将处理结果传递给下一个接收者，并无法终止广播 **intent** 的传播。

有序广播：按照被接收者的优先级顺序，在被接收者中依次传播。比如有三个广播接收者 **A**，**B**，**C**，优先级是 $A > B > C$ 。那这个消息先传给 **A**，再传给 **B**，最后传给 **C**。每个接收者有权终止广播，比如 **B** 终止广播，**C** 就无法接收到。此外 **A** 接收到广播后可以对结果对象进行操作，当广播传给 **B** 时，**B** 可以从结果对象中取得 **A** 存入的数据。

在通过 **Context.sendOrderedBroadcast(intent, receiverPermission, resultReceiver, scheduler, initialCode, initialData, initialExtras)**时我们可以指定 **resultReceiver** 广播接收者，这个接收者我们可以认为是最终接收者，通常情况下如果比他优先级更高的接收者如果没有终止广播，那么他的 **onReceive** 会被执行两次，第一次是正常的按照优先级顺序执行，第二次是作为最终接收者接收。如果比他优先级高的接收者终止了广播，那么他依然能接收到广播。

在我们的项目中经常使用广播接收者接收系统通知，比如开机启动、**sd** 挂载、低电量、外播电话、锁屏等。

如果我们做的是播放器，那么监听到用户锁屏后我们应该将我们的播放之暂停等。

2、在 manifest 和代码中如何注册和使用 BroadcastReceiver

在清单文件中注册广播接收者称为静态注册，在代码中注册称为动态注册。静态注册的广播接收者只要 app 在系统中运行则一直可以接收到广播消息，动态注册的广播接收者当注册的 Activity 或者 Service 销毁了那么就接收不到广播了。

静态注册：在清单文件中进行如下配置

```
<receiver android:name=".BroadcastReceiver1" >
    <intent-filter>
        <action android:name="android.intent.action.CALL" >
        </action>
    </intent-filter>
</receiver>
```

动态注册：在代码中进行如下注册

```
receiver = new BroadcastReceiver();
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(CALL_ACTION);
context.registerReceiver(receiver, intentFilter);
```

3、BroadcastReceiver 的生命周期

- a. 广播接收者的生命周期非常短暂的，在接收到广播的时候创建，onReceive()方法结束之后销毁；
- b. 广播接收者中不要做一些耗时的工作，否则会弹出 Application No Response 错误对话框；
- c. 最好也不要再在广播接收者中创建子线程做耗时的工作，因为广播接收者被销毁后进程就成为了空进程，很容易被系统杀掉；

d. 耗时的较长的工作最好放在服务中完成;

4、Android 引入广播机制的用意

- a. 从 MVC 的角度考虑(应用程序内) 其实回答这个问题的时候还可以这样问，android 为什么要有那 4 大组件，现在的移动开发模型基本上也是照搬的 web 那一套 MVC 架构，只不过是改了点嫁妆而已。android 的四大组件本质上就是为了实现移动或者说嵌入式设备上的 MVC 架构，它们之间有时候是一种相互依存的关系，有时候又是一种补充关系，引入广播机制可以方便几大组件的信息和数据交互。
- b. 程序间互通消息(例如在自己的应用程序内监听系统来电)
- c. 效率上(参考 UDP 的广播协议在局域网的方便性)
- d. 设计模式上(反转控制的一种应用，类似监听者模式)

四、 ContentProvider

1、请介绍下 ContentProvider 是如何实现数据共享的

在 Android 中如果想将自己应用的数据（一般多为数据库中的数据）提供给第三发应用，那么我们只能通过 ContentProvider 来实现了。

ContentProvider 是应用程序之间共享数据的接口。使用的时候首先自定义一个类继承 ContentProvider，然后覆写 query、insert、update、delete 等方法。因为它是四大组件之一因此必须在 AndroidManifest 文件中进行注册。

把自己的数据通过 uri 的形式共享出去

android 系统下 不同程序 数据默认是不能共享访问

需要去实现一个类去继承 ContentProvider

```
public class PersonContentProvider extends ContentProvider{  
    public boolean onCreate(){  
  
    }  
}
```

query(Uri, String[], String, String[], String)

insert(Uri, ContentValues)

```
update(Uri, ContentValues, String, String[])  
delete(Uri, String, String[])  
}
```

```
<provider  
    android:exported="true"  
    android:name="com.itheima.contentProvider.provider.PersonContentPro  
vider" android:authorities="com.itheima.person" />
```

第三方可以通过 **ContentResolver** 来访问该 **Provider**。

2、请介绍下 Android 的数据存储方式

- a. File 存储
- b. SharedPreferences 存储
- c. ContentProvider 存储
- d. SQLiteDatabase 存储
- e. 网络存储

3、为什么要用 ContentProvider? 它和 sql 的实现上有什么差别?

ContentProvider 屏蔽了数据存储的细节,内部实现对用户完全透明,用户只需要关心操作数据的 **uri** 就可以了, **ContentProvider** 可以实现不同 **app** 之间共享。

Sql 也有增删改查的方法,但是 **sql** 只能查询本应用下的数据库。而 **ContentProvider** 还可以去增删改查本地文件.xml 文件的读取等。

4、说说 ContentProvider、ContentResolver、ContentObserver 之间的关系

- a. ContentProvider 内容提供者,用于对外提供数据
- b. ContentResolver.notifyChange(uri)发出消息

- c. `ContentResolver` 内容解析者，用于获取内容提供者提供的数据
- d. `ContentObserver` 内容监听器，可以监听数据的改变状态
- e. `ContentResolver.registerContentObserver()` 监听消息。

五、 `ListView`

1、`ListView` 如何提高其效率？

当 `convertView` 为空时，用 `setTag()` 方法为每个 `View` 绑定一个存放控件的 `ViewHolder` 对象。当 `convertView` 不为空，重复利用已经创建的 `view` 的时候，使用 `getTag()` 方法获取绑定的 `ViewHolder` 对象，这样就避免了 `findViewById` 对控件的层层查询，而是快速定位到控件。

- ① 复用 `ConvertView`，使用历史的 `view`，提升效率 200%
- ② 自定义静态类 `ViewHolder`，减少 `findViewById` 的次数。提升效率 50%
- ③ 异步加载数据，分页加载数据。
- ④ 使用 `WeakRefrence` 引用 `ImageView` 对象

2、当 `ListView` 数据集改变后，如何更新 `ListView`

使用该 `ListView` 的 `adapter` 的 `notifyDataSetChanged()` 方法。该方法会使 `ListView` 重新绘制。

3、`ListView` 如何实现分页加载

- ① 设置 `ListView` 的滚动监听器：`setOnScrollListener(new`

OnScrollListener{...})

在监听器中有两个方法：滚动状态发生变化的方法(`onScrollStateChanged`)和

`listView` 被滚动时调用的方法(`onScroll`)

② 在滚动状态发生改变的方法中，有三种状态：

手指按下移动的状态： `SCROLL_STATE_TOUCH_SCROLL`: // 触摸滑动

惯性滚动（滑翔（`fling`）状态）： `SCROLL_STATE_FLING`: // 滑翔

静止状态： `SCROLL_STATE_IDLE`: // 静止

对不同的状态进行处理：

分批加载数据，只关心静止状态：关心最后一个可见的条目，如果最后一个可见条目就是数据适配器（集合）里的最后一个，此时可加载更多的数据。在每次加载的时候，计算出滚动的数量，当滚动的数量大于等于总数量的时候，可以提示用户无更多数据了。

4、ListView 可以显示多种类型的条目吗

这个当然可以的，`ListView` 显示的每个条目都是通过 `baseAdapter` 的 `getView(int position, View convertView, ViewGroup parent)`来展示的，理论上我们完全可以让每个条目都是不同类型的 `view`。

比如：从服务器拿回一个标识为 `id=1`,那么当 `id=1` 的时候，我们就加载类型一的条目，当 `id=2` 的时候，加载类型二的条目。常见布局在资讯类客户端中可以经常看到。

除此之外 `adapter` 还提供了 `getViewTypeCount()` 和 `getItemViewType(int position)` 两个方法。在 `getView` 方法中我们可以根据不同的 `viewtype` 加载不同的布局文件。

5、ListView 如何定位到指定位置

可以通过 ListView 提供的 `lv.setSelection(listView.getPosition());`方法。

6、如何在 ScrollView 中如何嵌入 ListView

通常情况下我们不会在 ScrollView 中嵌套 ListView，但是如果面试官非让我嵌套的话也是可以的。

在 ScrollView 添加一个 ListView 会导致 listview 控件显示不全，通常只会显示一条，这是因为两个控件的滚动事件冲突导致。所以需要通过 listview 中的 item 数量去计算 listview 的显示高度，从而使其完整展示，如下提供一个方法供大家参考。

```
lv = (ListView) findViewById(R.id.lv);
adapter = new MyAdapter();
lv.setAdapter(adapter);
setListViewHeightBasedOnChildren(lv);

-----
public void setListViewHeightBasedOnChildren(ListView listView) {
    ListAdapter listAdapter = listView.getAdapter();
    if (listAdapter == null) {
        return;
    }
    int totalHeight = 0;
    for (int i = 0; i < listAdapter.getCount(); i++) {
        View listItem = listAdapter.getView(i, null, listView);
        listItem.measure(0, 0);
        totalHeight += listItem.getMeasuredHeight();
    }
    ViewGroup.LayoutParams params = listView.getLayoutParams();
    params.height = totalHeight + (listView.getDividerHeight() *
(listAdapter.getCount() - 1));
    params.height += 5; // if without this statement, the listview will
                        // little short
    listView.setLayoutParams(params);
}
```

现阶段最好的处理的方式是： 自定义 `ListView`，重载 `onMeasure()`方法，设置全部显示。

```
package com.meiya.ui;

import android.widget.ListView;

/**
 *
 * @Description:  scrollview中内嵌 listview的简单实现
 *
 * @File:  ScrollViewWithListView.java
 *
 *
 * @Version
 */
public class ScrollViewWithListView extends ListView {

    public ScrollViewWithListView(android.content.Context context,
    android.util.AttributeSet attrs) {
        super(context, attrs);
    }

    /**
     * Integer.MAX_VALUE >> 2,如果不设置，系统默认设置是显示两条
     */
    public void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        int expandSpec = MeasureSpec.makeMeasureSpec(Integer.MAX_VALUE >> 2,
        MeasureSpec.AT_MOST);
        super.onMeasure(widthMeasureSpec, expandSpec);
    }

}
```

7、ListView 中如何优化图片

图片的优化策略比较多。

1、处理图片的方式：

如果 **ListView** 中自定义的 **Item** 中有涉及到大量图片的，一定要对图片进行细心的处理，因为图片占的内存是 **ListView** 项中最头疼的，处理图片的方法大致有以下几种：

①、不要直接拿路径就去循环 **BitmapFactory.decodeFile**；使用 **Options** 保存图片大小、不要加载图片到内存去。

②、对图片一定要经过边界压缩尤其是比较大的图片，如果你的图片是后台服务器处理好的那就不需要了

③、在 **ListView** 中取图片时也不要直接拿个路径去取图片，而是以 **WeakReference**（使用 **WeakReference** 代替强引用。比如可以使用 **WeakReference mContextRef**）、**SoftReference**、**WeakHashMap** 等的来存储图片信息。

④、在 **getView** 中做图片转换时，产生的中间变量一定及时释放

2、异步加载图片基本思想：

1）、先从内存缓存中获取图片显示（内存缓冲）

2）、获取不到的话从 **SD** 卡里获取（**SD** 卡缓冲）

3）、都获取不到的话从网络下载图片并保存到 **SD** 卡同时加入内存并显示（视情况看是否要显示）

原理：

优化一：先从内存中加载，没有则开启线程从 **SD** 卡或网络中获取，这里注意从 **SD** 卡获取图片是放在子线程里执行的，否则快速滚屏的话会不够流畅。

优化二：于此同时，在 **adapter** 里有个 **busy** 变量，表示 **listview** 是否处于滑动状态，如果是滑动状态则仅从内存中获取图片，没有的话无需再开启线程去外存或网络获取图片。

优化三：**ImageLoader** 里的线程使用了线程池，从而避免了过多线程频繁创建和销毁，

如果每次总是 `new` 一个线程去执行这是非常不可取的，好一点的用的 `AsyncTask` 类，其实内部也是用到了线程池。在从网络获取图片时，先是将其保存到 `sd` 卡，然后再加载到内存，这么做的好处是在加载到内存时可以做个压缩处理，以减少图片所占内存。

8、ListView 中图片错位的问题是如何产生的

图片错位问题的本质源于我们的 `listview` 使用了缓存 `convertView`，假设一种场景，一个 `listview` 一屏显示九个 `item`，那么在拉出第十个 `item` 的时候，事实上该 `item` 是重复使用了第一个 `item`，也就是说在第一个 `item` 从网络中下载图片并最终要显示的时候，其实该 `item` 已经不在当前显示区域内了，此时显示的后果将可能在第十个 `item` 上输出图像，这就导致了图片错位的问题。所以解决之道在于可见则显示，不可见则不显示。

9、如何刷新 ListView 中单个 item 的数据，不刷新整个 ListView 的数据？

修改单个 `Item` 的数据,然后调用适配器的 `notifyDataSetChanged()` 方法

六、Intent

1、Intent 传递数据时，可以传递哪些类型数据？

`Intent` 可以传递的数据类型非常的丰富，`java` 的基本数据类型和 `String` 以及他们的数组形式都可以，除此之外还可以传递实现了 `Serializable` 和 `Parcelable` 接口的对象。

2、Serializable 和 Parcelable 的区别

在使用内存的时候，Parcelable 类比 Serializable 性能高，所以推荐使用 Parcelable 类。

1. Serializable 在序列化的时候会产生大量的临时变量，从而引起频繁的 GC。
2. Parcelable 不能使用在要将数据存储在磁盘上的情况。尽管 Serializable 效率低点，

但在这种情况下，还是建议你用 Serializable 。

实现：

1. Serializable 的实现，只需要继承 Serializable 即可。这只是给对象打了一个标记，系统会自动将其序列化。
2. Parcelable 的实现，需要在类中添加一个静态成员变量 CREATOR，这个变量需要继承 Parcelable.Creator 接口。

```
public class MyParcelable implements Parcelable {
    private int mData;

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(mData);
    }

    public static final Parcelable.Creator<MyParcelable> CREATOR
        = new Parcelable.Creator<MyParcelable>() {
        public MyParcelable createFromParcel(Parcel in) {
            return new MyParcelable(in);
        }

        public MyParcelable[] newArray(int size) {
            return new MyParcelable[size];
        }
    };

    private MyParcelable(Parcel in) {
        mData = in.readInt();
    }
}
```

3、请描述一下 Intent 和 IntentFilter

Android 中通过 Intent 对象来表示一条消息，一个 Intent 对象不仅包含有这个消息的目的地，还可以包含消息的内容，这好比一封 Email，其中不仅应该包含收件地址，还可以包含具体的内容。对于一个 Intent 对象，消息“目的地”是必须的，而内容则是可选项。

通过 Intent 可以实现各种系统组件的调用与激活。

IntentFilter: 可以理解为邮局或者是一个信笺的分拣系统...

这个分拣系统通过 3 个参数来识别

Action: 动作 view

Data: 数据 uri uri

Category: 而外的附加信息

Action 匹配

Action 是一个用户定义的字符串，用于描述一个 Android 应用程序组件，一个 IntentFilter 可以包含多个 Action。在 AndroidManifest.xml 的 Activity 定义时可以在其 <intent-filter> 节点指定一个 Action 列表用于标示 Activity 所能接受的“动作”，例如：

```
<intent-filter >

<action android:name="android.intent.action.MAIN" />

<action android:name="cn.itheima.action" />

.....

</intent-filter>
```

如果我们在启动一个 Activity 时使用这样的 Intent 对象：

```
Intent intent =new Intent();

intent.setAction("cn.itheima.action");
```

那么所有的 Action 列表中包含了“cn.itheima”的 Activity 都将会匹配成功。

Android 预定义了一系列的 Action 分别表示特定的系统动作。这些 Action 通过常量的方式定义在 android.content.Intent 中，以“ACTION_”开头。我们可以在 Android 提供的文档中找到它们的详细说明。

URI 数据匹配

一个 **Intent** 可以通过 **URI** 携带外部数据给目标组件。在 `<intent-filter>` 节点中，通过 `<data/>` 节点匹配外部数据。

mimeType 属性指定携带外部数据的数据类型，**scheme** 指定协议，**host**、**port**、**path** 指定数据的位置、端口、和路径。如下：

```
<data android:mimeType="mimeType" android:scheme="scheme"
      android:host="host" android:port="port" android:path="path"/>
```

电话的 uri tel: 12345

http://www.baidu.com

自己定义的 uri itcast://cn.itcast/person/10

如果在 **Intent Filter** 中指定了这些属性，那么只有所有的属性都匹配成功时 **URI** 数据匹配才会成功。

Category 类别匹配

`<intent-filter>` 节点中可以为组件定义一个 **Category** 类别列表，当 **Intent** 中包含这个列表的所有项目时 **Category** 类别匹配才会成功。

七、Fragment

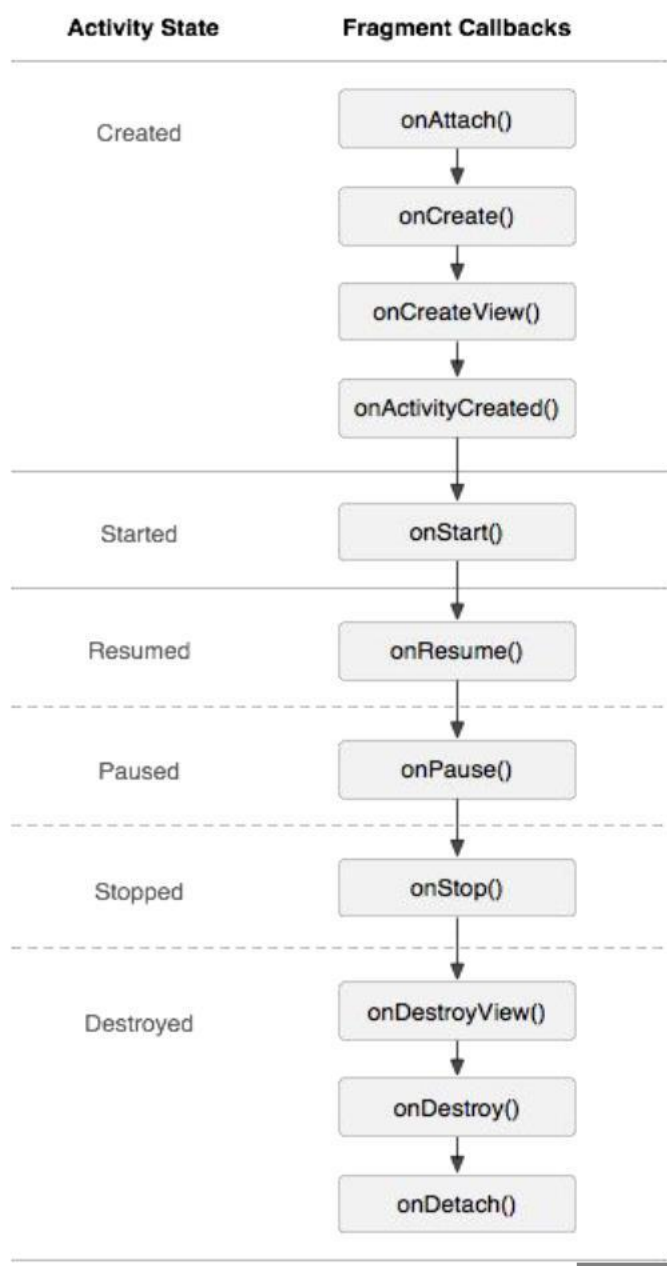
1、Fragment 跟 Activity 之间是如何传值的

当 **Fragment** 跟 **Activity** 绑定之后，在 **Fragment** 中可以直接通过 `getActivity()` 方法获取到其绑定的 **Activity** 对象，这样就可以调用 **Activity** 的方法了。在 **Activity** 中可以通过如下方法获取到 **Fragment** 实例

```
FragmentManager fragmentManager = getFragmentManager();  
Fragment fragment = fragmentManager.findFragmentByTag(tag);  
Fragment fragment = fragmentManager.findFragmentById(id);
```

获取到 **Fragment** 之后就可以调用 **Fragment** 的方法。也就实现了通信功能。

2、描述一下 **Fragment** 的生命周期



3、Fragment 的 replace 和 add 方法的区别

Fragment 本身并没有 replace 和 add 方法，这里的理解应该为使用 FragmentManager 的 replace 和 add 两种方法切换 Fragment 时有什么不同。

我们经常使用的一个架构就是通过 RadioGroup 切换 Fragment，每个 Fragment 就是一个功能模块。

```
case R.id.rb_1:
    rb_1.setBackgroundColor(Color.RED);
    transaction.show(fragment1);
// transaction.replace(R.id.fl, fragment1, "Fragment1");
break;
case R.id.rb_2:
    rb_2.setBackgroundColor(Color.YELLOW);
// transaction.replace(R.id.fl, fragment2, "Fragment2");
    transaction.show(fragment2);
    break;
case R.id.rb_3:
    rb_3.setBackgroundColor(Color.BLUE);
// transaction.replace(R.id.fl, fragment3, "Fragment3");
    transaction.show(fragment3);
    break;
```

实现这个功能可以通过 replace 和 add 两种方法。

Fragment 的容器一个 FrameLayout，add 的时候是把所有的 Fragment 一层一层的叠加到了 FrameLayout 上了，而 replace 的话首先将该容器中的其他 Fragment 去除掉然后将当前 Fragment 添加到容器中。

一个 Fragment 容器中只能添加一个 Fragment 种类，如果多次添加则会报异常，导致程序终止，而 replace 则无所谓，随便切换。

因为通过 add 的方法添加的 Fragment，每个 Fragment 只能添加一次，因此如果要想达到切换效果需要通过 Fragment 的 hide 和 show 方法结合者使用。将要显示的 show 出来，将其他 hide 起来。这个过程 Fragment 的生命周期没有变化。

通过 `replace` 切换 `Fragment`，每次都会执行上一个 `Fragment` 的 `onDestroyView`，新 `Fragment` 的 `onCreateView`、`onStart`、`onResume` 方法。

基于以上不同的特点我们在使用的使用一定要结合着生命周期操作我们的视图和数据。

4、Fragment 如何实现类似 Activity 栈的压栈和出栈效果的？

`Fragment` 的事物管理器内部维持了一个双向链表结构，该结构可以记录我们每次 `add` 的 `Fragment` 和 `replace` 的 `Fragment`，然后当我们点击 `back` 按钮的时候会自动帮我们实现退栈操作。

Add this transaction to the back stack. This means that the transaction will be remembered after it is complete and will reverse its operation when later popped off the stack.

Parameters:

nameAn optional name for this back stack state, or null.

```
transaction.addToBackStack("name");
```

//实现源码在 `BackStackRecord`中

```
public FragmentTransaction addToBackStack(String name) {
    if (!mAllowAddToBackStack) {
        throw new IllegalStateException(
            "This FragmentTransaction is not allowed to be added to the back stack.");
    }
    mAddToBackStack = true;
    mName = name;
    return this;
}
```

//上面的源码仅仅做了一个标记

除此之外因为我们要使用 `FragmentManager` 用的是 `FragmentActivity`，因此 `FragmentActivity` 的 `onBackPressed` 方法必定重新覆写了。打开看一下，发现确实如此。

```
/**
 * Take care of popping the fragment back stack or finishing t
 * as appropriate.
 */
public void onBackPressed() {
    if (!mFragments.popBackStackImmediate()) {
        finish();
    }
}
//mFragments的原型是FragmentManagerImpl，看看这个方法都干嘛了
```

```
@Override
public boolean popBackStackImmediate() {
    checkStateLoss();
    executePendingTransactions();
    return popBackStackState(mActivity mHandler, null, -1, 0);
}
```

//看看 popBackStackState方法都干了啥，其实通过名称也能大概了解只给几个片段吧，代码了

```
while (index >= 0) {
    //从后退栈中取出当前记录对象
    BackStackRecord bss = mBackStack.get(index);
    if (name != null && name.equals(bss.getName())) {
        break;
    }
    if (id >= 0 && id == bss.mIndex)
    { break;
    }
    index--;
}
```

5、Fragment 在你们项目中的使用

Fragment 是 android3.0 以后引入的概念，做局部内容更新更方便，原来为了到达

这一点要把多个布局放到一个 **activity** 里面，现在可以用多 **Fragment** 来代替，只有在需要的时候才加载 **Fragment**，提高性能。

Fragment 的好处：

- (1) **Fragment** 可以使你能够将 **activity** 分离成多个可重用的组件，每个都有它自己的生命周期和 **UI**。
- (2) **Fragment** 可以轻松得创建动态灵活的 **UI** 设计，可以适应于不同的屏幕尺寸。从手机到平板电脑。
- (3) **Fragment** 是一个独立的模块,紧紧地与 **activity** 绑定在一起。可以运行中动态地移除、加入、交换等。
- (4) **Fragment** 提供一个新的方式让你在不同的安卓设备上统一你的 **UI**。
- (5) **Fragment** 解决 **Activity** 间的切换不流畅，轻量切换。
- (6) **Fragment** 替代 **TabActivity** 做导航，性能更好。
- (7) **Fragment** 在 4.2.版本中新增嵌套 **fragment** 使用方法，能够生成更好的界面效果。

6、如何切换 **fragment**,不重新实例化

翻看了 **Android** 官方 **Doc**，和一些组件的源代码，发现 **replace()**这个方法只是在上一个 **Fragment** 不再需要时采用的简便方法。

正确的切换方式是 **add()**，切换时 **hide()**，**add()**另一个 **Fragment**；再次切换时，只需 **hide()**当前，**show()**另一个。

这样就能做到多个 **Fragment** 切换不重新实例化：

```
public void switchContent(Fragment from, Fragment to) {
```

```
if (mContent != to) {  
  
    mContent = to;  
  
    FragmentTransaction transaction  
    =mFragmentManager.beginTransaction().setCustomAnimations(android.R.anim.fade_in,  
    R.anim.slide_out);  
  
    if (!to.isAdded()) { // 先判断是否被 add 过  
  
        transaction.hide(from).add(R.id.content_frame, to).commit(); // 隐藏当前的  
fragment, add 下一个到 Activity 中  
  
    } else {  
  
        transaction.hide(from).show(to).commit(); // 隐藏当前的 fragment, 显示下一  
个  
  
    }  
  
}  
  
}
```

Android 高级 (★★★)

一、Android 性能优化 (11.9 更新)

1、如何对 Android 应用进行性能分析

一款 App 流畅与否安装在自己的真机里，玩几天就能有个大概的感性认识。不过通过专业的分析工具可以使我们更好的分析我们的应用。而在实际开发中，我们解决完当前应用所有 bug 后，就会开始考虑到新能的优化。

如果不考虑使用其他第三方性能分析工具的话，我们可以直接使用 `ddms` 中的工具，其实 `ddms` 工具已经非常的强大了。`ddms` 中有 `traceview`、`heap`、`allocation tracker` 等工具都可以帮助我们分析应用的方法执行时间效率和内存使用情况。

`traceview`

(一) `TraceView` 简介

`Traceview` 是 Android 平台特有的数据采集和分析工具，它主要用于分析 Android 中应用程序的 `hotspot`（瓶颈）。`Traceview` 本身只是一个数据分析工具，而数据的采集则需要使用 Android SDK 中的 `Debug` 类或者利用 `DDMS` 工具。

二者的用法如下：

开发者在一些关键代码段开始前调用 Android SDK 中 `Debug` 类的 `startMethodTracing` 函数，并在关键代码段结束前调用 `stopMethodTracing` 函数。这两个函数运行过程中将采集运行时间内该应用所有线程（注意，只能是 `Java` 线程）的函数执行情况，并将采集数据保存到 `/mnt/sdcard/` 下的一个文件中。开发者然后需要利用 SDK 中的 `Traceview` 工具来分析这些数据。

借助 Android SDK 中的 `DDMS` 工具。`DDMS` 可采集系统中某个正在运行的进程的函数调

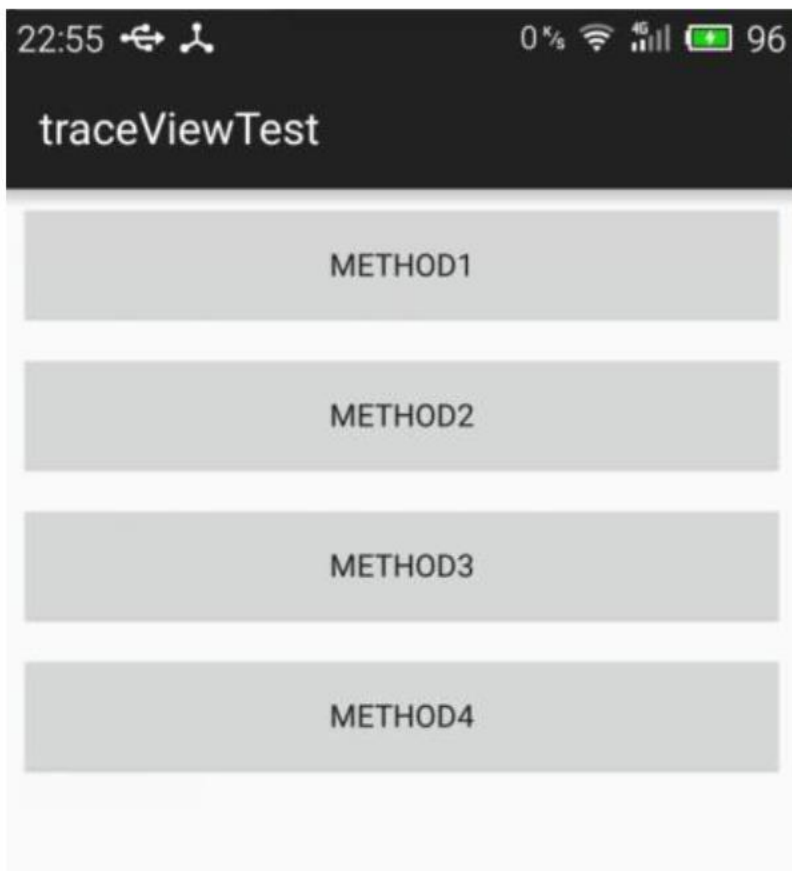
用信息。对开发者而言，此方法适用于没有目标应用源代码的情况。DDMS工具中 Traceview 的使用如下图所示。



点击上图中所示按钮即可以采集目标进程的数据。当停止采集时，DDMS 会自动触发 Traceview 工具来浏览采集数据。

下面，我们通过一个示例程序介绍 Traceview 的使用。

实例程序如下图所示：界面有 4 个按钮，对应四个方法。



点击不同的方法会进行不同的耗时操作。

```
public class MainActivity extends ActionBarActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
  
    public void method1(View view) {  
        int result = jisuan();  
        System.out.println(result);  
    }  
}
```

```
private int jisuan() {
    for (int i = 0; i < 10000; i++) {
        System.out.println(i);
    }
    return 1;
}

public void method2(View view) {
    SystemClock.sleep(2000);
}

public void method3(View view) {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
        sum += i;
    }
    System.out.println("sum=" + sum);
}

public void method4(View view) {
    Toast.makeText(this, "" + new Date(), 0).show();
}
}
```

我们分别点击按钮一次，要求找出最耗时的方法。点击前通过 DDMS 启动 Start Method Profiling 按钮。

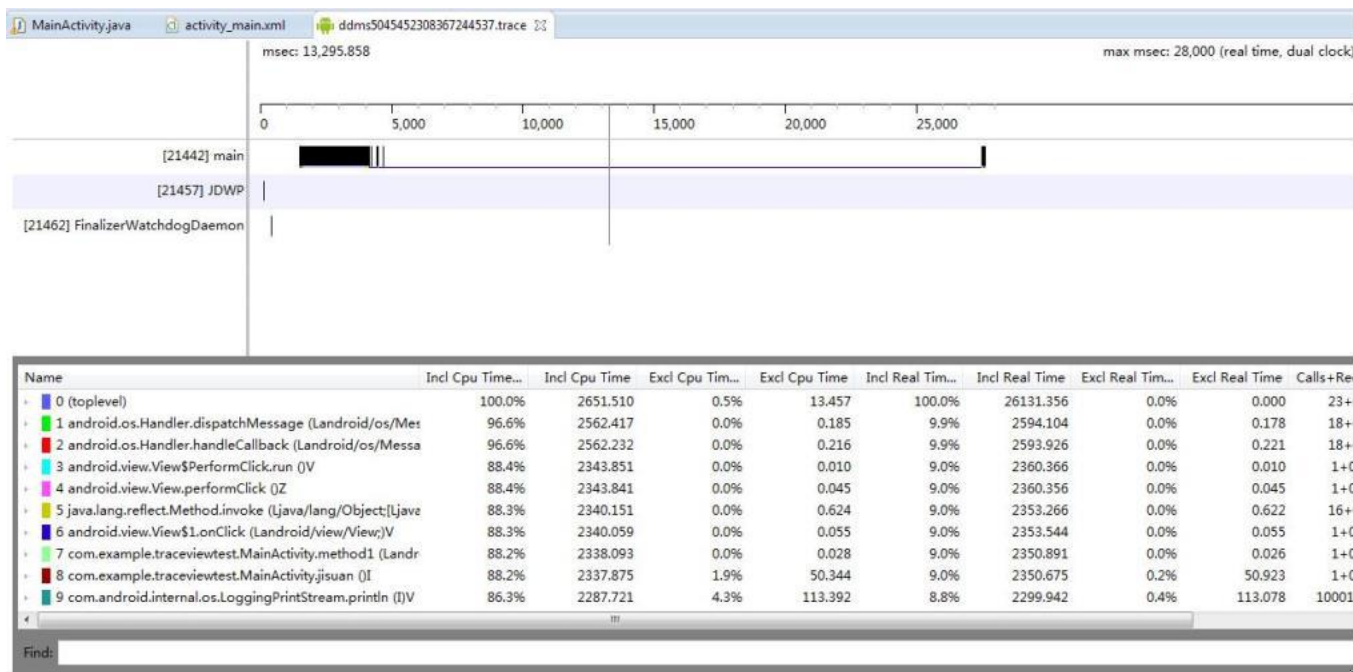


然后依次点击 4 个按钮，都执行后再次点击上图中红框中按钮，停止收集数据。

接下来我们开始对数据进行分析。

当我们停止收集数据的时候会出现如下分析图表。该图表分为 2 大部分，上面分不同的行，每一行代表一个线程的执行耗时情况。main 线程对应行的内容非常丰富，而其他

线程在这段时间内干得工作则要少得多。图表的下半部分是具体的每个方法执行的时间情况。显示方法执行情况的前提是先选中某个线程。



我们主要是分析 **main** 线程。

上面方法指标参数所代表的意思如下：

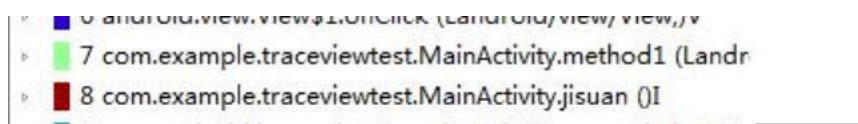
列名	描述
Name	该线程运行过程中所调用的函数名
Incl Cpu Time	某函数占用的 CPU 时间，包含内部调用其它函数的 CPU 时间
Excl Cpu Time	某函数占用的 CPU 时间，但不含内部调用其它函数所占用的 CPU 时间
Incl Real Time	某函数运行的真实时间（以毫秒为单位），内含调用其它函数所占用的真实时间
Excl Real Time	某函数运行的真实时间（以毫秒为单位），不含调用其它函数所占用的真实时间
Call+Recur Calls/Total	某函数被调用次数以及递归调用占总调用次数的百分比

Cpu Time/Call	某函数调用 CPU 时间与调用次数的比。相当于该函数平均执行时间
Real Time/Call	同 CPU Time/Call 类似，只不过统计单位换成了真实时间

我们为了找到最耗时的操作，那么可以通过点击 **Incl Cpu Time**，让其按照时间的倒序排列。我点击后效果如下图：

Name	Incl Cpu Time...	Incl Cpu ...	Excl Cpu Tim...	Excl Cpu Time	Incl Real Tim...	Incl Real Time	Excl Real Tim...	Excl Real Time	Cal
0 (top level)	100.0%	2651.510	0.5%	13.457	100.0%	26131.356	0.0%	0.000	
1 android.os.Handler.dispatchMessage (Landroid/os/Me...	96.6%	2562.417	0.0%	0.185	9.9%	2594.104	0.0%	0.178	
2 android.os.Handler.handleCallback (Landroid/os/Messa...	96.6%	2562.232	0.0%	0.216	9.9%	2593.926	0.0%	0.221	
3 android.view.View\$PerformClick.run (V	88.4%	2343.851	0.0%	0.010	9.0%	2360.366	0.0%	0.010	
4 android.view.View.performClick (IZ	88.4%	2343.841	0.0%	0.045	9.0%	2360.356	0.0%	0.045	
5 java.lang.reflect.Method.invoke (Ljava/lang/Object;[Ljava...	88.3%	2340.151	0.0%	0.624	9.0%	2353.266	0.0%	0.622	
6 android.view.View\$1.onClick (Landroid/view/View;V	88.3%	2340.059	0.0%	0.055	9.0%	2353.544	0.0%	0.055	
7 com.example.traceviewtest.MainActivity.method1 (Landr...	88.2%	2338.093	0.0%	0.028	9.0%	2350.891	0.0%	0.026	
8 com.example.traceviewtest.MainActivity.jisuan ()I	88.2%	2337.875	1.9%	50.344	9.0%	2350.675	0.2%	50.923	
9 com.android.internal.os.LoggingPrintStream.println (I)V	86.3%	2287.721	4.3%	113.392	8.8%	2299.942	0.4%	113.078	
10 com.android.internal.os.LoggingPrintStream.flush (Z)V	52.1%	1382.477	8.9%	234.993	5.3%	1388.300	0.9%	235.907	
11 java.lang.StringBuilder.append (Ljava/lang/StringBuilc...	29.9%	791.852	2.6%	69.036	3.1%	798.386	0.3%	69.077	
12 java.lang.IntegralToString.appendInt (Ljava/lang/Abstr...	27.3%	722.816	2.6%	67.797	2.8%	729.309	0.3%	67.760	
13 java.lang.IntegralToString.convertInt (Ljava/lang/Abstra...	24.7%	655.084	5.5%	146.132	2.5%	659.603	0.6%	147.898	
14 java.lang.StringBuilder.substring (I)Ljava/lang/String;	13.7%	363.810	2.6%	70.150	1.4%	365.069	0.3%	69.746	
15 java.lang.ThreadLocal.get ()Ljava/lang/Object;	11.6%	308.482	7.6%	201.578	1.2%	311.870	0.8%	203.362	

通过分析发现：**method1** 最耗时，耗时 2338 毫秒。



那么有了上面的信息我们可以进入我们的 **method1** 方法查看分析我们的代码了。

heap

(二) heap 简介

heap 工具可以帮助我们检查代码中是否存在会造成内存泄漏的地方。

用 heap 监测应用进程使用内存情况的步骤如下：

1. 启动 eclipse 后，切换到 DDMS 透视图，并确认 Devices 视图、Heap 视图都是打开的；
2. 点击选中想要监测的进程，比如 system_process 进程；
3. 点击选中 Devices 视图界面中最上方一排图标中的“Update Heap”图标；

4. 点击 Heap 视图中的 “Cause GC” 按钮；

5. 此时在 Heap 视图中就会看到当前选中的进程的内存使用量的详细情况。

说明：

- a. 点击 “Cause GC” 按钮相当于向虚拟机请求了一次 gc 操作；
- b. 当内存使用信息第一次显示以后，无须再不断的点击 “Cause GC”，Heap 视图界面会定时刷新，在对应用的不断的操作过程中就可以看到内存使用的变化；
- c. 内存使用信息的各项参数根据名称即可知道其意思，在此不再赘述。

如何才能知道我们的程序是否有内存泄漏的可能性呢？

- 这里需要注意一个值：Heap 视图中部有一个 Type 叫做 data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在 data object 一行中有一列是 “Total Size”，其值就是当前进程中所有 Java 数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：
 - 不断的操作当前应用，同时注意观察 data object 的 Total Size 值；
 - 正常情况下 Total Size 值都会稳定在一个有限的范围内，也就是说由于程序中的的代码良好，没有造成对象不被垃圾回收的情况，所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行 GC 的过程中，这些对象都被回收了，内存占用量会落到一个稳定的水平；
 - 反之如果代码中存在没有释放对象引用的情况，则 data object 的 Total Size 值在每次 GC 后不会有明显的回落，随着操作次数的增多 Total Size 的值会越来越大，直到到达一个上限后导致进程被 kill 掉。
 - 此处以 system_process 进程为例，在我的测试环境中 system_process 进程所占用的内存的 data object 的 Total Size 正常情况下会稳定在 2.2~2.8 之间，而当其值超过

3.55 后进程就会被 kill。

总之，使用 DDMS 的 Heap 视图工具可以很方便地确认我们的程序是否存在内存泄漏的可能性。

allocation tracker

（三）allocation tracker 简介

allocation tracker 是内存分配跟踪工具

步骤：

运行 DDMS，只需简单的选择应用进程并单击 Allocation tracker 标签，就会打开一个新的窗口，单击 “Start Tracing” 按钮；

然后，让应用运行你想分析的代码。运行完毕后，单击 “Get Allocations” 按钮，一个已分配对象的列表就会出现第一个表格中。

单击第一个表格中的任何一项，在表格二中就会出现导致该内存分配的栈跟踪信息。通过 allocation tracker，不仅知道分配了哪类对象，还可以知道在哪个线程、哪个类、哪个文件的哪一行。

2、什么情况下会导致内存泄露

Android 的虚拟机是基于寄存器的 Dalvik，它的最大堆大小一般是 16M，有的机器为 24M。因此我们所能利用的内存空间是有限的。如果我们的内存占用超过了一定的水平就会出现 OutOfMemory 的错误。

内存溢出的几点原因：

1. 资源释放问题

程序代码的问题，长期保持某些资源，如 **Context**、**Cursor**、**IO** 流的引用，资源得不到释放造成内存泄露。

2. 对象内存过大问题

保存了多个耗用内存过大的对象（如 **Bitmap**、**XML** 文件），造成内存超出限制。

3.

static 关键字的使用问题

static 是 **Java** 中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用 **static** 修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（**Context** 的情况最多），这时就要谨慎对待了。

```
public class ClassName {  
  
    private static Context mContext;  
  
    //省略  
  
}
```

以上的代码是很危险的，如果将 **Activity** 赋值到 **mContext** 的话。那么即使该 **Activity** 已经 **onDestroy**，但是由于仍有对象保存它的引用，因此该 **Activity** 依然不会被释放。

我们举 **Android** 文档中的一个例子。


```
private static Drawable sBackground;
@Override
protected void onCreate(Bundle state) {
    super.onCreate(state);
    TextView label = new TextView(this);    //getApplicationContext
    label.setText("Leaks are bad");
    if (sBackground == null) {
        sBackground = getDrawable(R.drawable.large_bitmap);
    }
    label.setBackgroundDrawable(sBackground);
    setContentView(label);
}
```

sBackground 是一个静态的变量，但是我们发现，我们并没有显式的保存 Context 的引用，但是，当 Drawable 与 View 连接之后，Drawable 就将 View 设置为一个回调，由于 View 中是包含 Context 的引用的，所以，实际上我们依然保存了 Context 的引用。这个引用链如下：

Drawable->TextView->Context

所以，最终该 Context 也没有得到释放，发生了内存泄露。

◆ 针对 static 的解决方案

- 1) 应该尽量避免 static 成员变量引用资源耗费过多的实例，比如 Context。
- 2) Context 尽量使用 ApplicationContext，因为 Application 的 Context 的生命周期比较长，引用它不会出现内存泄露的问题。
- 3) 使用 WeakReference 代替强引用。比如可以使用

WeakReference<Context> mContextRef;

4. 线程导致内存溢出

线程产生内存泄露的主要原因在于线程生命周期的不可控。我们来考虑下面一段代码。

```
public class MyActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        new MyThread().start();
    }
    private class MyThread extends Thread{
    @Override
        public void run()
        {
            super.run();
            //do something while(true)
        }
    }
}
```

这段代码很平常也很简单，是我们经常使用的形式。我们思考一个问题：假设 **MyThread** 的 **run** 函数是一个很费时的操作，当我们开启该线程后，将设备的横屏变为了竖屏，一般情况下当屏幕转换时会重新创建 **Activity**，按照我们的想法，老的 **Activity** 应该会被销毁才对，然而事实上并非如此。

由于我们的线程是 **Activity** 的内部类，所以 **MyThread** 中保存了 **Activity** 的一个引用，当 **MyThread** 的 **run** 函数没有结束时，**MyThread** 是不会被销毁的，因此它所引用的老的 **Activity** 也不会被销毁，因此就出现了内存泄露的问题。

有些人喜欢用 **Android** 提供的 **AsyncTask**，但事实上 **AsyncTask** 的问题更加严重，**Thread** 只有在 **run** 函数不结束时才出现这种内存泄露问题，然而 **AsyncTask** 内部的实现机制是运用了 **ThreadPoolExecutor**，该类产生的 **Thread** 对象的生命周期是不确定的，是应用程序无法控制的，因此如果 **AsyncTask** 作为 **Activity** 的内部类，就更容易出现内存泄露的问题。

针对这种线程导致的内存泄露问题的解决方案：

(一) 将线程的内部类，改为静态内部类（因为非静态内部类拥有外部类对象的强引用，而静态类则不拥有）。

(二) 在线程内部采用弱引用保存 Context 引用。

3、如何避免 OOM 异常

OOM 内存溢出，想要避免 OOM 异常首先我们要知道什么情况下会导致 OOM 异常。

1、图片过大导致 OOM

Android 中用 bitmap 时很容易内存溢出，比如报如下错误：

Java.lang.OutOfMemoryError : bitmap size exceeds VM budget.

解决方法：

方法 1： 等比例缩小图片

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 2;
//Options只保存图片尺寸大小，不保存图片到内存
BitmapFactory.Options opts = new BitmapFactory.Options();
opts.inSampleSize = 2;
Bitmap bmp =
    null;
bmp = BitmapFactory.decodeResource(getResources(),
    mImageIds[position],opts);
//回收
bmp.recycle();//
```

以上代码可以优化内存溢出，但它只是改变图片大小，并不能彻底解决内存溢出。

方法 2：对图片采用软引用，及时地进行 recycle()操作

```
SoftReference<Bitmap> bitmap = new SoftReference<Bitmap>(pBitmap);  
if(bitmap != null){  
    if(bitmap.get() != null && !bitmap.get().isRecycled()){  
        bitmap.get().recycle();  
        bitmap = null;  
    }  
}
```

方法 3：使用加载图片框架处理图片，如专业处理加载图片的 **ImageLoader** 图片加载框架。

还有我们学的 **XUtils** 的 **BitMapUtils** 来做处理。

2、界面切换导致 OOM

一般情况下，开发中都会禁止横屏的。因为如果是来回切换话，**activity** 的生命周期会重新销毁然后创建。

有时候我们会发现这样的问题，横竖屏切换 **N** 次后 **OOM** 了。

这种问题没有固定的解决方法，但是我们可以从以下几个方面下手分析。

1、看看页面布局当中有没有大的图片，比如背景图之类的。

去除 **xml** 中相关设置，改在程序中设置背景图（放在 **onCreate()** 方法中）：

```
Drawable drawable = getResources().getDrawable(R.drawable.id);  
ImageView imageView = new ImageView(this);  
imageView.setBackgroundDrawable(drawable);
```

在 **Activity destory** 时注意，**drawable.setCallback(null)**；防止 **Activity** 得不到及时的释放。

2、跟上面方法相似，直接把 **xml** 配置文件加载成 **view** 再放到一个容器里，然后直接调用 **this.setContentView(View view)**；方法，避免 **xml** 的重复加载。

3、在页面切换时尽可能少地重复使用一些代码

比如：重复调用数据库，反复使用某些对象等等.....

3、查询数据库没有关闭游标

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 **Cursor** 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在长时间大量操作的情况下才会出现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

4、构造 **Adapter** 时，没有使用缓存的 **convertView**

在使用 **ListView** 的时候通常会使用 **Adapter**，那么我们应该尽可能的使用 **convertView**。

为什么要使用 **convertView**?

当 **convertView** 为空时，用 **setTag()** 方法为每个 **View** 绑定一个存放控件的 **ViewHolder** 对象。当 **convertView** 不为空，重复利用已经创建的 **view** 的时候，使用 **getTag()** 方法获取绑定的 **ViewHolder** 对象，这样就避免了 **findViewById** 对控件的层层查询，而是快速定位到控件。

5、**Bitmap** 对象不再使用时调用 **recycle()** 释放内存

有时我们会手工的操作 **Bitmap** 对象，如果一个 **Bitmap** 对象比较占内存，当它不再被使用的时候，可以调用 **Bitmap.recycle()** 方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。

6、其他

Android 应用程序中最典型的需要注意释放资源的情况是在 Activity 的生命周期中，在 onPause()、onStop()、onDestroy()方法中需要适当的释放资源的情况。使用广播没有注销也会产生 OOM。

4、Android 中如何捕获未捕获的异常

（一）UncaughtExceptionHandler

1、自定义一个 Application，比如叫 MyApplication 继承 Application 实现 UncaughtExceptionHandler。

2、覆写 UncaughtExceptionHandler 的 onCreate 和 uncaughtException 方法。

```
@Override
public void onCreate() {
    super.onCreate();
    Thread.setDefaultUncaughtExceptionHandler(this);
}

@Override
public void uncaughtException(final Thread thread, final Throwable
    new Thread(new Runnable() {

        @Override
        public void run()
        {
            Looper.prepare();
            System.out.println(Thread.currentThread());
            Toast.makeText(getApplicationContext(), "thread="+thread.getId()
ex="+ex.toString(), 1).show();
            Looper.loop();
        }
    }).start();
    SystemClock.sleep(3000);
    android.os.Process.killProcess(android.os.Process.myPid());
}
}
```

注意：上面的代码只是简单的将异常打印出来。

在 `onCreate` 方法中我们给 `Thread` 类设置默认异常处理 `handler`，如果这句代码不执行则一切都是白搭。

在 `uncaughtException` 方法中我们必须新开辟个线程进行我们异常的收集工作，然后将系统给杀死。

3、在 `AndroidManifest` 中配置该 `Application`

```
<application
    android:name="com.example.uncatchexception.MyApplication"
```

（二）Bug 收集工具 Crashlytics

Crashlytics 是专门为移动应用开发者提供的保存和分析应用崩溃的工具。国内主要使用的是友盟做数据统计。

Crashlytics 的好处：

1. Crashlytics 不会漏掉任何应用崩溃信息。
2. Crashlytics 可以象 Bug 管理工具那样，管理这些崩溃日志。
3. Crashlytics 可以每天和每周将崩溃信息汇总发到你的邮箱，所有信息一目了然。

使用步骤：

- 1.注册需要审核通过才能使用，国内同类产品顶多发个邮箱激活链接；
- 2.支持 Eclipse、IntelliJ IDEA 和 Android Studio 等三大 IDE；
- 3.Eclipse 插件是 iOS 主题风格 UI，跟其他 plugin 在一起简直是鹤立鸡群；
- 4.只要登录帐号并选择项目，会自动导入 jar 包并生成一个序列号，然后在 AndroidManifest.xml 和启动 Activity 的入口添加初始化代码，可以说是一键式操作，当然要使用除错误统计外的其他功能还是得自己添加代码；
- 5.不像友盟等国内同类产品，将固定的序列号直接写入 xml 文件，而是动态自动生成的；当然这个存放序列号的 xml 文件也是不能修改和提交到版本控制系统的；
- 6.后台可以设置邮件提醒，当然这个最好不要开启，Android 开发那数量惊人、千奇百怪的错误信息你懂的。
- 7.不仅能统计到 UncaughtException 这种未捕获的 Crash 异常信息，只要在 try/catch 代码块的 catch 中添加一行代码就能统计到任何异常；


```
try{ myMethodThatThrows(); }catch(Exception e){ Crashlytics.logException(e);  
  
//handle your exception here! }
```

8.相当详细的错误信息，不仅仅是简单的打印 **StackTrace** 信息；并且能看到最近一次 **crash** 的机器可用内存等信息，而不仅仅是简单统计机型和版本号。

使用连接：<http://blog.csdn.net/smking/article/details/39320695>

5、 **ANR** 是什么？怎样避免和解决 **ANR**（重要）

在 **Android** 上，如果你的应用程序有一段时间响应不够灵敏，系统会向用户显示一个对话框，这个对话框称作应用程序无响应（**ANR: Application Not Responding**）对话框。用户可以选择让程序继续运行，但是，他们在使用你的应用程序时，并不希望每次都要处理这个对话框。因此，在程序里对响应性能的设计很重要，这样，系统不会显示 **ANR** 给用户。

Activity 5 秒 broadcast10 秒

耗时的操作 worker thread 里面完成, handler message...AsyncTask , intentservice.
等...

ANR:Application Not Responding，即应用无响应

ANR 一般有三种类型：

1: **KeyDispatchTimeout(5 seconds)** --主要类型

按键或触摸事件在特定时间内无响应

2: **BroadcastTimeout(10 seconds)**

BroadcastReceiver 在特定时间内无法处理完成

3: ServiceTimeout(20 seconds) --小概率类型

Service 在特定的时间内无法处理完成

超时的原因一般有两种：

(1)当前的事件没有机会得到处理（UI 线程正在处理前一个事件没有及时完成或者 looper 被某种原因阻塞住）

(2)当前的事件正在处理，但没有及时完成

UI 线程尽量只做跟 UI 相关的工作，耗时的工作（数据库操作，I/O，连接网络或者其他可能阻碍 UI 线程的操作）放入单独的线程处理，尽量用 Handler 来处理 UI thread 和 thread 之间的交互。

UI 线程主要包括如下：

Activity: onCreate(), onResume(), onDestroy(), onKeyDown(), onClick()

AsyncTask: onPreExecute(), onProgressUpdate(), onPostExecute(),
onCancel()

Mainthread handler: handleMessage(), post(runnable r)

查找 ANR 的方式：1. 导出/data/data/anr/traces.txt，找出函数和调用过程，分析代码 2.
通过性能 LOG 人肉查找

6、Android 线程间通信有哪几种方式（重要）

- 共享内存（变量）；
- 文件，数据库；
- Handler；
- Java 里的 wait(), notify(), notifyAll()

7、Devik 进程，linux 进程，线程的区别

Dalvik 虚拟机运行在 Linux 操作系统之上。Linux 操作系统并没有纯粹的线程概念，只要两个进程共享一个地址空间，那么就可以认为它们是同一个进程的两个线程。Linux 系统提供了两个 fork 和 clone 调用，其中，前者是用来创建进程的，而后者是用来创建线程的。

一般来说，虚拟机的进程和线程都是和目标机器本地操作系统的进程和线程一一对应的，这样的好处是可以使本地操作系统来调度进程和线程。

每个 Android 应用程序进程都有一个 Dalvik 虚拟机实例。这样做得好处是 Android 应用程序进程之间不会互相影响，也就是说，一个 Android 应用程序进程的意外终止，不会影响到其他的应用程序进程的正常运行。

- 每个 Android 应用程序进程都是由一种称为 Zygote 的进程 fork 出来的。Zygote 进程是由 init 进程启动起来的，也就是在系统启动的时候启动的。Zygote 进程在启动的时候，会创建一个虚拟机实例，并且在这个虚拟机实例将所有的 Java 核心库都加载起来。每当 Zygote 进程需要创建一个 Android 应用程序进程的时候，它就通过复制自身来实现，也就是通过 fork 系统调用来实现。这些被 fork 出来的 Android 应用

程序进程，一方面是复制了 **Zygote** 进程中的虚拟机实例，另外一方面是与 **Zygote** 进程共享了同一套 **Java** 核心库。这样不仅 **Android** 程序进程的创建很快，而且所有的应用程序都共享同一套 **Java** 核心库而节省了内存空间。

8、描述一下 **android** 的系统架构？

1. **android** 系统架构分从下往上为 **linux** 内核层、运行库、应用程序框架层、和应用程序层。
2. **linuxkernel**: 负责硬件的驱动程序、网络、电源、系统安全以及内存管理等功能。
3. **libraries** 和 **androidruntime**: **libraries**: 即 **c/c++** 函数库部分，大多数都是开放源代码的函数库，例如 **webkit**，该函数库负责 **android** 网页浏览器的运行，例如标准的 **c** 函数库 **libc**、**openssl**、**sqlite** 等，当然也包括支持游戏开发 **2dsgl** 和 **3dopengles**，在多媒体方面有 **mediaframework** 框架来支持各种影音和图形文件的播放与显示，例如 **mpeg4**、**h.264**、**mp3**、**aac**、**amr**、**jpg** 和 **png** 等众多多媒体文件格式。**android** 的 **runtime** 负责解释和执行生成的 **dalvik** 格式的字节码。
4. **applicationframework**（应用软件架构），**java** 应用程序开发人员主要是使用该层封装好的 **api** 进行快速开发。
5. **applications**: 该层是 **java** 的应用程序层，**android** 内置的 **googlemaps**、**e-mail**、即时通信工具、浏览器、**mp3** 播放器等处于该层，**java** 开发人员开发的程序也处于该层，而且和内置的应用程序具有平等的位置，可以调用内置的应用程序，也可以替换内置的应用程序。

9、android 应用对内存是如何限制的?我们应该如何合理使用内存? (2016.01.24)

如何限制的?

Android 应用的开发语言为 Java，每个应用最大可使用的堆内存受到 Android 系统的限制

- Android 每一个应用的堆内存大小有限
- 通常的情况为 16M-48M
- 通过 ActivityManager 的 getMemoryClass()来查询可用堆内存限制
- 3.0(HoneyComb)以上的版本可以通过 largeHeap= "true" 来申请更多的堆内存
- NexusS(4.2.1):normal 192, largeHeap 512
- 如果试图申请的内存大于当前余下的堆内存就会引发 OutOfMemoryError()
- 应用程序由于各方面的限制，需要注意减少内存占用，避免出现内存泄漏。

获取这个代码:

```
[java] mActivityManager = (ActivityManager) this.getSystemService(Context.ACTIVITY_SERVICE);  
mMaxMemory = mActivityManager.getMemoryClass();
```

如何合理使用内存?

- 1、注意资源回收，像数据库，输入输出流，定位操作这样的对象，要在使用完及时关闭流。
- 2、少使用静态变量，因为系统将静态变量的优先级设定的很高，会最后回收。所以可能因为静态变量导致该回收的没有回收。而回收了不该回收的内存。
- 3、注意大图片的缩放，如果载入的图片很大，要先经过自己程序的处理，降低分辨率等。最好设置多种分辨率格式的图片，以减少内存消耗。

4、动态注册监听，把一些只有显示的时候才使用到的监听放进程序内部，而不是放在 `manifest` 中去。

5、减少使用动画，或者适当减少动画的帧数。

6、注意自己的程序逻辑，在该关闭自己程序的控件的时候，主动关闭，不要交给系统去决定。（这个要自己把握好，也不是说都自己搞定，只有那些自己确定需要关闭的对象，自己将其关闭。）

10、简述 `android` 应用程序结构是哪些？（2016.01.24）

`Android` 应用程序结构也就是讲我们的工程结构：



src 目录是源代码目录，所有允许用户修改的 **java** 文件和用户自己添加的 **java** 文件都保存在这个目录中

gen 目录是 1.5 版本新增的目录，用来保存 ADT 自动生成的 **java** 文件，例如 **R.java** 或 **AIDL** 文件

注意：R.java 文件（非常重要）

a) **R.java** 文件是 ADT 自动生成的文件，包含对 **drawable**、**layout** 和 **values** 目录内的资源的引用指针，**Android** 程序能够直接通过 **R** 类引用目录中的资源

b) **R.java** 文件不能手工修改，如果向资源目录中增加或删除了资源文件，则需要在工程名称上右击，选择 **Refresh** 来更新 **R.java** 文件中的代码

c) **R** 类包含的几个内部类，分别与资源类型相对应，资源 ID 便保存在这些内部类中，例如子类 **drawable** 表示图像资源，内部的静态变量 **icon** 表示资源名称，其资源 ID 为 **0x7f020000**。一般情况下，资源名称与资源文件名相同

android.jar 文件是 **Android** 程序所能引用的函数库文件，**Android** 通过平台所支持 **API** 都包含在这个文件中

assets 目录用来存放原始格式的文件，例如音频文件、视频文件等二进制格式文件。此目录中的资源不能被 **R.java** 文件索引，所以只能以资源流的形式读取。一般情况下为空

layout 目录用来存放我们为每个界面写的布局文件

Strings.xml 文件是程序中的一些字符串的引用

AndroidManifest.xml 是 XML 格式的 **Android** 程序声明文件，包含了 **Android** 系统运行 **Android** 程序前所必须掌握的重要信息，这些信息包含应用程序名称、图标、包名称、模块组成、授权和 **SDK** 最低版本等，而且每个 **Android** 程序必须在根目录下包含一个 **AndroidManifest.xml** 文件

注：AndroidMainfest.xml 文件：

- 1) AndroidManifest.xml 文件的根元素是 **manifest**，包含了 **xmlns:android**、**package**、**android:versionCode** 和 **android:versionName** 共 4 个属性
 - 2) **xmlns:android** 定义了 **Android** 的命名空间，值为 **http://schemas.android.com/apk/res/android**
 - 3) **package** 定义了应用程序的包名称
 - 4) **android:versionCode** 定义了应用程序的版本号，是一个整数值，数值越大说明版本越新，但仅在程序内部使用，并不提供给应用程序的使用者
 - 5) **android:versionName** 定义了应用程序的版本名称，是一个字符串，仅限于为用户提供一个版本标识
 - 6) **manifest** 元素仅能包含一个 **application** 元素，**application** 元素中能够声明 **Android** 程序中最重要四个组成部分，包括 **Activity**、**Service**、**BroadcastReceiver** 和 **ContentProvider**，所定义的属性将影响所有组成部分
 - 7) **android:icon** 定义了 **Android** 应用程序的图标，其中 **@drawable/icon** 是一种资源引用方式，表示资源类型是图像，资源名称为 **icon**，对应的资源文件为 **res/drawable** 目录下的 **icon.png**
 - 8) **android:label** 则定义了 **Android** 应用程序的标签名称
- default.properties** 文件记录 **Android** 工程的相关设置，该文件不能手动修改，需右键单击工程名称，选择 “**Properties**” 进行修改

11、请解释下 **Android** 程序运行时权限与文件系统权限的区别？（2016.01.24）

apk 程序是运行在虚拟机上的,对应的是 **Android** 独特的权限机制，只有体现到文件系统上时才使用 **linux** 的权限设置。

（一）linux 文件系统上的权限

```
-rwxr-x--x system    system    4156 2010-04-30 16:13 test.apk
```

代表的是相应的用户/用户组及其他人对此文件的访问权限，与此文件运行起来具有的权限完全不相关。比如上面的例子只能说明 **system** 用户拥有对此文件的读写执行权限；**system** 组的用户对此文件拥有读、执行权限；其他人对此文件只具有执行权限。而 **test.apk** 运行起来后可以干哪些事情，跟这个就不相关了。千万不要看 **apk** 文件系统上属于 **system/system** 用户及用户组，或者 **root/root** 用户及用户组，就认为 **apk** 具有 **system** 或 **root** 权限

（二）Android 的权限规则

- （1）Android 中的 **apk** 必须签名
- （2）基于 **UserID** 的进程级别的安全机制
- （3）默认 **apk** 生成的数据对外是不可见的
- （4）**AndroidManifest.xml** 中的显式权限声明

12、**Framework** 工作方式及原理，**Activity** 是如何生成一个 **view** 的，机制是什么？

（2016.01.24）

所有的框架都是基于反射 和 配置文件（**manifest**）的。

普通的情况：

Activity 创建一个 **view** 是通过 **ondraw** 画出来的，画这个 **view** 之前呢，还会调用 **onmeasure** 方法来计算显示的大小。

特殊情况：

Surfaceview 是直接操作硬件的，因为 或者视频播放对帧数有要求，**onDraw** 效率太

低，不够使，Surfaceview 直接把数据写到显存。

13、多线程间通信和多进程之间通信有什么不同，分别怎么实现？（2016.01.24）

一、进程间的通信方式

管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

有名管道 (namedpipe) ：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

信号量(semaphore) ：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。

因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

消息队列(messagequeue)：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

信号 (sinal) ：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

共享内存(shared memory) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

套接字(socket) ：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

二、线程间的通信方式

锁机制：包括互斥锁、条件变量、读写锁

*互斥锁提供了以排他方式防止数据结构被并发修改的方法。

*读写锁允许多个线程同时读共享数据，而对写操作是互斥的。

*条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量

信号机制(Signal)：类似进程间的信号处理


线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

二、Android 屏幕适配


1、屏幕适配方式都有哪些


1.1 适配方式之 dp

名词解释：

 分辨率：eg：480*800,1280*720。表示物理屏幕区域内像素点的总和。(切记：跟屏幕适配没有任何关系)

因为我们既可以把 1280*720 的分辨率做到 4.0 的手机上面。我也可以把 1280*720 的分辨率做到 5.0 英寸的手机上面，如果分辨率相同，手机屏幕越小清晰。

 px(pix)：像素，就是屏幕中最小的一个显示单元

 **dpi**（像素密度）：即每英寸屏幕所拥有的像素数，像素密度越大，显示画面细节就越丰富。

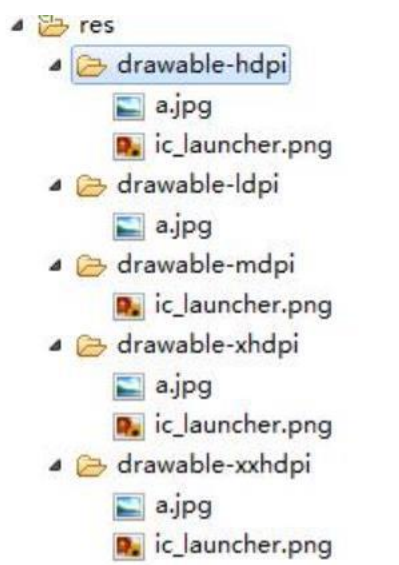
计算公式：像素密度= $\sqrt{\text{长度像素数}^2 + \text{宽度像素数}^2}$ / 屏幕尺寸

注：屏幕尺寸单位为英寸 例：分辨率为 1280*720 屏幕宽度为 6 英寸 计算所得像素密度约等于 245，屏幕尺寸指屏幕对角线的长度。

在 Android 手机中 dpi 分类：

ldpi	Resources for low-density (ldpi) screens (~120dpi).
mdpi	Resources for medium-density (mdpi) screens (~160dpi). (This is the baseline density.)
hdpi	Resources for high-density (hdpi) screens (~240dpi).
xhdpi	Resources for extra high-density (xhdpi) screens (~320dpi).





在我们的 Android 工程目录中有如下 **drawable-*dpi** 目录，这些目录是用来适配不同分辨率手机的。



Android 应用在查找图片资源时会根据其分辨率自动从不同的文件目录下查找（这本身就是 Android 系统的适配策略），如果在低分辨的文件目录中比如 **drawable-mdpi** 中

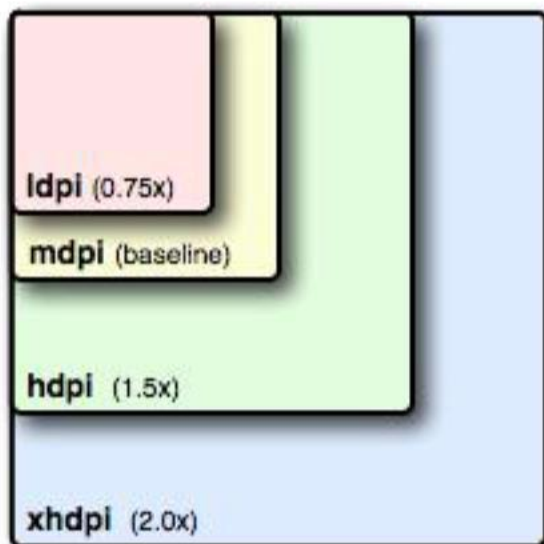
没有图片资源，其他目录中都有，当我们将该应用部署到 **mdpi** 分辨率的手机上时，那么该应用会查找分辨率较高目录下的资源文件，如果较高分辨率目录下也没有资源则只好找较低目录中的资源了。

常见手机屏幕像素及对应分辨率级别：

-  ldpi 320*240
-  mdpi 480*320
-  hdpi 800*480
-  xhdpi 1280*720
-  xxhdpi 1920*1080

dp 和 px 之间的简单换算关系：

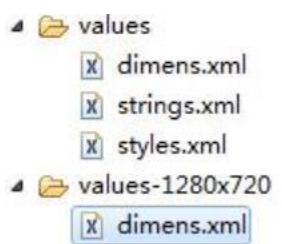
-  ldpi 的手机 1dp=0.75px
-  mdpi 的手机 1dp=1.0px
-  hdpi 的手机 1dp=1.5px
-  xhdpi 的手机 1dp=2.0px
-  xxhdpi 的手机 1dp=3.0px



Tips：根据上面的描述我们得出如下结论，对于 **mdpi** 的手机，我们的布局通过 **dp** 单位可以达到适配效果。

1.2 适配方式之 **dimens**

跟 **drawable** 目录类似的，在 **Android** 工程的 **res** 目录下有 **values** 目录，这个是默认的目录，同时为了适配不同尺寸手机我们可以创建一个 **values-1280x720** 的文件夹，同时将 **dimens.xml** 文件拷贝到该目录下。



在 **dimens.xml** 中定义一个尺寸，如下图所示。

```
dimens.xml
1 <resources>
2   <dimen name="width">160dp</dimen>
3 </resources>
```

在 **values-1280x720** 目录中的 **dimens.xml** 中定义同样的尺寸名称，但是使用不同的尺寸，

如下图所示。

```
<resources>
    <dimen name="width">180dp</dimen>
</resources>
```

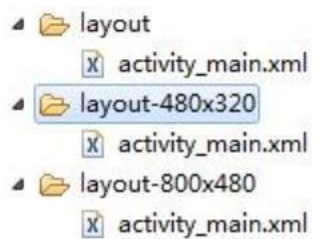
当我们在布局文件中使用长或者宽度单位时，比如下图所示，应该使用@dimen/width 来灵活的定义宽度。

```
<TextView
    android:background="#000000"
    android:layout_width="@dimen/width"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
<LinearLayout
```

Tips: 在 values-1280x720 中，中间的是大写字母 X 的小写形式 x，而不是加减乘除的乘号。如果我们在 values-1280x720 中放置了 **dimens** 常量，一定记得也将该常量的对应值在 values 目录下的 **dimens.xml** 中放一份，因为该文件是默认配置，当用户的手机不是 1280*720 的情况下系统应用使用的是默认 values 目录中的 **dimens.xml**。

1.3 适配方式之 layout

跟 values 一样，在 Android 工程目录中 layout 目录也支持类似 values 目录一样的适配，在 layout 中我们可以针对不同手机的分辨率制定不同的布局，如下图所示。



1.4 适配方式之 java 代码适配

为了演示用 java 代码控制适配的效果，因此假设有这样的需求，让一个 **TextView** 控件的宽和高分别为屏幕的宽和高的一半。

我们新创建一个 Android 工程，修改 main_activity.xml，布局文件清单如下：

```
<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"
    tools:context=".MainActivity" >
```

```
<!--当前控件宽高为屏幕宽度的各 50% -->
<TextView
    android:id="@+id/tv"
    android:background="#000000"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
</RelativeLayout>
```

在 MainActivity.java 类中完成用 java 代码控制 TextView 的布局效果，其代码清单如下：

```
public class MainActivity extends Activity {

    private static final String tag = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //去掉 title
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);
        //获取 TextView控件
        TextView tv = (TextView) findViewById(R.id.tv);
        //找到当前控件的父控件(父控件上给当前的子控件去设定一个规则)
        DisplayMetrics metrics = new DisplayMetrics();
        //给当前 metrics去设置当前屏幕信息(宽(像素)高(像素))
        getWindowManager().getDefaultDisplay().getMetrics(metrics);
        //获取屏幕的高度和宽度
        Constant.srceenHeight = metrics.heightPixels;
        Constant.srceenWidth = metrics.widthPixels;
        //日志输出屏幕的高度和宽度
        Log.i(tag, "Constant.srceenHeight = "+Constant.srceenHeight);
        Log.i(tag, "Constant.srceenWidth = "+Constant.srceenWidth);
        //宽高各 50%
        RelativeLayout.LayoutParams layoutParams = new RelativeLayout.Layo
            //数学角度上四舍五入
            (int)(Constant.srceenWidth*0.5+0.5),
            (int)(Constant.srceenHeight*0.5+0.5));
        //给 tv控件设置布局参数
        tv.setLayoutParams(layoutParams);
    }
}
```

其中 Constant 类是一个常量类，很简单，只有两个常量用来记录屏幕的宽和高，其代码清单如下：

```
public class Constant {
    public static int srceenHeight;
    public static int srceenWidth;
}
```

1.5 适配方式之 weight 权重适配

在控件中使用属性 `android:layout_weight="1"`可以起到适配效果，但是该属性的使用有如下规则：

只能用在线性控件中，比如 `LinearLayout`。

竖直方向上使用权重的控件高度必须为 `0dp`（Google 官方的推荐用法）

水平方向上使用权重的控件宽度必须为 `0dp`（Google 官方的推荐用法）

2、屏幕适配的处理技巧都有哪些

手机自适应主要分为两种情况：横屏和竖屏的切换，以及分辨率大小的不同。

2.1 横屏和竖屏的切换

1、Android 应用程序支持横竖屏幕的切换，Android 中每次屏幕的切换都会重启 Activity，所以应该在 Activity 销毁（执行 `onPause()`方法和 `onDestroy()`方法）前保存当前活动的状态；在 Activity 再次创建的时候载入配置，那样，进行中的游戏就不会自动重启了！有的程序适合从竖屏切换到横屏，或者反过来，这个时候怎么办呢？可以在配置 Activity 的地方进行如下的配置 `android:screenOrientation="portrait"`（`landscape` 是横向，`portrait` 是纵向）。这样就可以保证是竖屏总是竖屏了。

2、而有的程序是适合横竖屏切换的。如何处理呢？首先要在配置 Activity 的时候进行如下的配置：

`android:configChanges="keyboardHidden|orientation"`，另外 需要 重写 Activity 的 `onConfigurationChanged` 方法。实现方式如下：

```
@Override
public void onConfigurationChanged(Configuration newConfig){
    super.onConfigurationChanged(newConfig);
    if(this.getResources().getConfiguration().orientation==Configuration.ORIENTATION_LANDSCAPE){
        //TODO
    }else
    if(this.getResources().getConfiguration().orientation==Configuration.ORIENTATION_PORTRAIT){
        //TODO
    }
}
```

2.2 分辨率大小不同

对于分辨率问题，官方给的解决办法是创建不同的 **layout** 文件夹，这就需要对每种分辨率的手机都要写一个布局文件，虽然看似解决了分辨率的问题，但是如果其中一处或多处有修改了，就要每个布局文件都要做出修改，这样就造成很大的麻烦。那么可以通过以下几种方式解决：

一）使用 **layout_weight**

目前最为推荐的 **Android** 多屏幕自适应解决方案。

该属性的作用是决定控件在其父布局中的显示权重，一般用于线性布局中。其值越小，则对应的 **layout_width** 或 **layout_height** 的优先级就越高（一般到 100 作用就不太明显了）；一般横向布局中，决定的是 **layout_width** 的优先级；纵向布局中，决定的是 **layout_height** 的优先级。

传统的 **layout_weight** 使用方法是将当前控件的 **layout_width** 和 **layout_height** 都设置成 **fill_parent**，这样就可以把控件的显示比例完全交给 **layout_weight**；这样使用的话，就出现了 **layout_weight** 越小，显示比例越大的情况（即权重越大，显示所占的效果越小）。不过对于 2 个控件还好，如果控件过多，且显示比例也不相同的时候，控制起来

就比较麻烦了，毕竟反比不是那么好确定的。于是就有了现在最为流行的 0px 设值法。看似让人难以理解的 `layout_height=0px` 的写法，结合 `layout_weight`，却可以使控件成正比显示，轻松解决了当前 Android 开发最为头疼的碎片化问题之一。

二) 清单文件配置：【不建议使用这种方式，需要对不同的界面写不同的布局】

需要在 `AndroidManifest.xml` 文件的 `<manifest>` 元素如下添加子元素

```
<supports-screensandroid:largeScreens="true"
```

```
android:normalScreens="true"
```

```
android:anyDensity="true"
```

```
android:smallScreens="true"
```

```
android:xlargeScreens="true">
```

```
</supports-screens>
```

以上是为我们屏幕设置多分辨率支持（更准确的说是适配大、中、小三种密度）。

`Android:anyDensity="true"`，这一句对整个的屏幕都起着十分重要的作用，值为 `true`，我们的应用程序当安装在不同密度的手机上时，程序会分别加载 `hdpi,mdpi,ldpi` 文件夹中的资源。相反，如果值设置为 `false`，即使我们在 `hdpi,mdpi,ldpi, xdpi` 文件夹下拥有同一种资源，那么应用也不会自动地去相应文件夹下寻找资源。而是会在大密度和小密度手机上加载中密度 `mdpi` 文件中的资源。

有时候会根据需要在代码中动态地设置某个值，可以在代码中为这几种密度分别设置偏移量,但是这种方法最好不要使用，最好的方式是在 `xml` 文件中不同密度的手机进行分别设置。这里地图的偏移量可以在 `values-xpdi, values-hpdi,values-mdpi,values-ldpi` 四种文件夹中的 `dimens.xml` 文件进行设置。

三)、其他:

说明：

在不同分辨率的手机模拟器下，控件显示的位置会稍有不同

通过在 **layout** 中定义的布局设置的参数，使用 **dp (dip)**，会根据不同的屏幕分辨率进行适配

但是在代码中的各个参数值，都是使用的像素 (**px**) 为单位的

技巧：

1、尽量使用线性布局，相对布局，如果屏幕放不下了，可以使用 **ScrollView**（可以上下拖动）

ScrowView 使用的注意：

在不同的屏幕上显示内容不同的情况，其实这个问题我们往往是用滚动视图来解决的，也就是 **ScrowView**；需要注意的是 **ScrowView** 中使用 **layout_weight** 是无效的，既然使用 **ScrowView** 了，就把它里面的控件的大小都设成固定的吧。

2、指定宽高的时候，采用 **dip** 的单位，**dp** 单位动态匹配

3、由于 **android** 代码中写的单位都是像素，所有需要通过工具类进行转化

4、尽量使用 **9-patch** 图，可以自动的依据图片上面显示的内容被拉伸和收缩。其中在编辑的时候，灰色区域是被拉伸的，上下两个点控制水平方向的拉伸，左右两点控制垂直方向的拉伸

3、**dp** 和 **px** 之间的关系

dp：是 **dip** 的简写，指密度无关的像素。

指一个抽象意义上的像素，程序用它来定义界面元素。一个与密度无关的，在逻辑尺寸上，与一个位于像素密度为 **160dpi** 的屏幕上的像素是一致的。要把密度无关像素转换

为屏幕像素，可以用这样一个简单的公式： $\text{pixels} = \text{dips} * (\text{density} / 160)$ 。举个例子，在 DPI 为 240 的屏幕上，1 个 DIP 等于 1.5 个物理像素。

布局时最好使用 **dp** 来定义我们程序的界面，因为这样可以保证我们的 UI 在各种分辨率的屏幕上都可以正常显示。

```
/**
 *根据手机的分辨率从    px(像素)的单位转成为    dp
 */
public static int px2dip(Context context, float pxValue) {
    final float scale = context.getResources().getDisplayMetrics().density;
    return (int) (pxValue / scale + 0.5f);
}
/**
 *根据手机的分辨率从    dip的单位转成为    px(像素)
 */
public static int dip2px(Context context, float dpValue) {
    final float scale = context.getResources().getDisplayMetrics().density;
    return (int) (dpValue * scale + 0.5f);
}
```

三、AIDL

1、什么是 AIDL 以及如何使用

①aidl 是 Android interface definition Language 的英文缩写，意思 Android 接口定义语言。

②使用 aidl 可以帮助我们发布以及调用远程服务，实现跨进程通信。

③将服务的 aidl 放到对应的 src 目录，工程的 gen 目录会生成相应的接口类

我们通过 `bindService (Intent, ServiceConnect, int)` 方法绑定远程服务，在 `bindService` 中有一个 `ServiceConnec` 接口，我们需要覆写该类的

onServiceConnected(ComponentName,IBinder)方法，这个方法的第二个参数 IBinder 对象其实就是已经在 aidl 中定义的接口，因此我们可以将 IBinder 对象强制转换为 aidl 中的接口类。

我们通过 IBinder 获取到的对象（也就是 aidl 文件生成的接口）其实是系统产生的代理对象，该代理对象既可以跟我们的进程通信，又可以跟远程进程通信，作为一个中间的角色实现了进程间通信。

2、AIDL 的全称是什么?如何工作?能处理哪些类型的数据?

AIDL 全称 Android Interface Definition Language（Android 接口描述语言）是一种接口描述语言；编译器可以通过 aidl 文件生成一段代码，通过预先定义的接口达到两个进程内部通信进程跨界对象访问的目的。需要完成 2 件事情：1. 引入 AIDL 的相关类；2. 调用 aidl 产生的 class。理论上，参数可以传递基本数据类型和 String，还有就是 Bundle 的派生类，不过在 Eclipse 中，目前的 ADT 不支持 Bundle 做为参数。

四、Android 中的事件处理

1、Handler 机制

Android 中主线程也叫 UI 线程，那么从名字上我们也知道主线程主要是用来创建、更新 UI 的，而其他耗时操作，比如网络访问，或者文件处理，多媒体处理等都需要在子线程中操作，之所以在子线程中操作是为了保证 UI 的流畅程度，手机显示的刷新频率是 60Hz，也就是一秒钟刷新 60 次，每 16.67 毫秒刷新一次，为了不丢帧，那么主线程处理代码最好

不要超过 16 毫秒。当子线程处理完数据后，为了防止 UI 处理逻辑的混乱，Android 只允许主线程修改 UI，那么这时候就需要 Handler 来充当子线程和主线程之间的桥梁了。

我们通常将 Handler 声明在 Activity 中，然后覆写 Handler 中的 handleMessage 方法，当子线程调用 handler.sendMessage() 方法后 handleMessage 方法就会在主线程中执行。

这里面除了 Handler、Message 外还有隐藏的 Looper 和 MessageQueue 对象。

在主线程中 Android 默认已经调用了 Looper.preper() 方法，调用该方法的目的是在 Looper 中创建 MessageQueue 成员变量并把 Looper 对象绑定到当前线程中。当调用 Handler 的 sendMessage (对象) 方法的时候就将 Message 对象添加到了 Looper 创建的 MessageQueue 队列中，同时给 Message 指定了 target 对象，其实这个 target 对象就是 Handler 对象。主线程默认执行了 Looper.looper () 方法，该方法从 Looper 的成员变量 MessageQueue 中取出 Message，然后调用 Message 的 target 对象的 handleMessage() 方法。这样就完成了整个消息机制。

2、事件分发机制

2.1 事件分发中的 onTouch 和 onTouchEvent 有什么区别，又该如何使用？

这两个方法都是在 View 的 dispatchTouchEvent 中调用的，onTouch 优先于 onTouchEvent 执行。如果在 onTouch 方法中通过返回 true 将事件消费掉，onTouchEvent 将不会再执行。

另外需要注意的是，onTouch 能够得到执行需要两个前提条件，第一 mOnTouchListener 的值不能为空，第二当前点击的控件必须是 enable 的。因此如果你有一个控件是非 enable 的，那么给它注册 onTouch 事件将永远得不到执行。对于这一类控

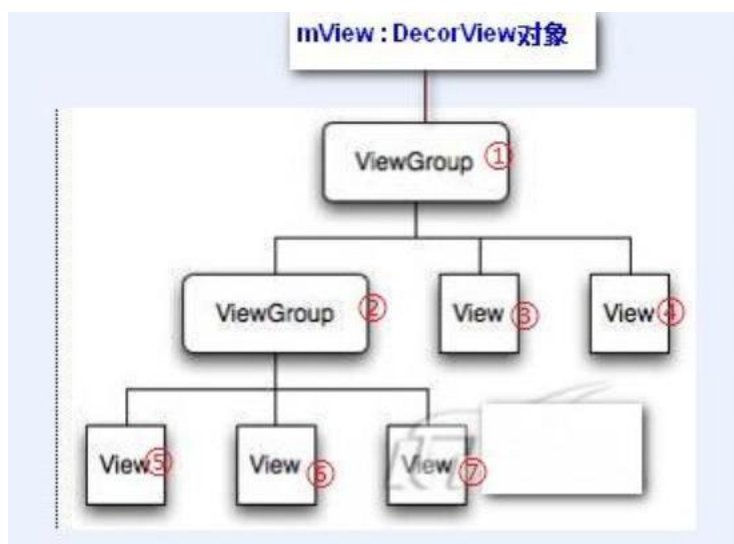
件，如果我们想要监听它的 `touch` 事件，就必须通过在该控件中重写 `onTouchEvent` 方法来实现。

2.2 请描述一下 Android 的事件分发机制

Android 的事件分发机制主要是 `Touch` 事件分发，有两个主角：`ViewGroup` 和 `View`。
`Activity` 的 `Touch` 事件事实上是调用它内部的 `ViewGroup` 的 `Touch` 事件，可以直接当成 `ViewGroup` 处理。

`View` 在 `ViewGroup` 内，`ViewGroup` 也可以在其他 `ViewGroup` 内，这时候把内部的 `ViewGroup` 当成 `View` 来分析。

先分析 `ViewGroup` 的处理流程：首先得有个结构模型概念：`ViewGroup` 和 `View` 成了一棵树形结构，最顶层为 `Activity` 的 `ViewGroup`，下面有若干的 `ViewGroup` 节点，每个节点之下又有若干的 `ViewGroup` 节点或者 `View` 节点，依次类推。如图：



当一个 `Touch` 事件(触摸事件为例)到达根节点，即 `Activity` 的 `ViewGroup` 时，它会依次下发，下发的过程是调用子 `View(ViewGroup)` 的 `dispatchTouchEvent` 方法实现的。简单来说，就是 `ViewGroup` 遍历它包含着的子 `View`，调用每个 `View` 的 `dispatchTouchEvent` 方法，而当子 `View` 为 `ViewGroup` 时，又会通过调用 `ViewGroup` 的 `dispatchTouchEvent`

方法继续调用其内部的 `View` 的 `dispatchTouchEvent` 方法。上述例子中的消息下发顺序是这样的：①-②-⑤-⑥-⑦-③-④。`dispatchTouchEvent` 方法只负责事件的分发，它拥有 `boolean` 类型的返回值，当返回为 `true` 时，顺序下发会中断。在上述例子中如果⑤的 `dispatchTouchEvent` 返回结果为 `true`，那么⑥-⑦-③-④将都接收不到本次 `Touch` 事件。

1.Touch 事件分发中只有两个主角：`ViewGroup` 和 `View`。`ViewGroup` 包含 `onInterceptTouchEvent`、`dispatchTouchEvent`、`onTouchEvent` 三个相关事件。`View` 包含 `dispatchTouchEvent`、`onTouchEvent` 两个相关事件。其中 `ViewGroup` 又继承于 `View`。

2.`ViewGroup` 和 `View` 组成了一个树状结构，根节点为 `Activity` 内部包含的一个 `ViwGroup`。

3.触摸事件由 `Action_Down`、`Action_Move`、`Aciton_UP` 组成，其中一次完整的触摸事件中，`Down` 和 `Up` 都只有一个，`Move` 有若干个，可以为 0 个。

4.当 `Acitivty` 接收到 `Touch` 事件时，将遍历子 `View` 进行 `Down` 事件的分发。`ViewGroup` 的遍历可以看成是递归的。分发的目的是为了找到真正要处理本次完整触摸事件的 `View`，这个 `View` 会在 `onTouchuEvent` 结果返回 `true`。

5.当某个子 `View` 返回 `true` 时，会中止 `Down` 事件的分发，同时在 `ViewGroup` 中记录该子 `View`。接下去的 `Move` 和 `Up` 事件将由该子 `View` 直接进行处理。由于子 `View` 是保存在 `ViewGroup` 中的，多层 `ViewGroup` 的节点结构时，上级 `ViewGroup` 保存的会是真实处理事件的 `View` 所在的 `ViewGroup` 对象：如 `ViewGroup0-ViewGroup1-TextView` 的结构中，`TextView` 返回了 `true`，它将被保存在 `ViewGroup1` 中，而 `ViewGroup1` 也会返回 `true`，被保存在 `ViewGroup0` 中。当 `Move` 和 `UP` 事件来时，会先从 `ViewGroup0` 传递至 `ViewGroup1`，再由 `ViewGroup1` 传递至 `TextView`。

6.当 `ViewGroup` 中所有子 `View` 都不捕获 `Down` 事件时，将触发 `ViewGroup` 自身的

onTouchEvent 事件。触发的方式是调用 `super.dispatchTouchEvent` 函数，即父类 `View` 的 `dispatchTouchEvent` 方法。在所有子 `View` 都不处理的情况下，触发 `Activity` 的 `onTouchEvent` 方法。

7.onInterceptTouchEvent 有两个作用：1.拦截 Down 事件的分发。2.中止 Up 和 Move 事件向目标 `View` 传递，使得目标 `View` 所在的 `ViewGroup` 捕获 Up 和 Move 事件。

3、子线程发消息到主线程进行更新 **UI**，除了 **handler** 和 **AsyncTask**，还有什么？

1、用 `Activity` 对象的 `runOnUiThread` 方法更新

在子线程中通过 `runOnUiThread()`方法更新 UI:

```
1. new Thread() {
2.     public void run() {
3.         //这儿是耗时操作，完成之后更新UI；
4.         runOnUiThread(new Runnable(){
5.
6.             @Override
7.             public void run() {
8.                 //更新UI
9.                 imageView.setImageBitmap(bitmap);
10.            }
11.
12.        });
13.    }
14. }.start();
```

如果在非上下文类中（`Activity`），可以通过传递上下文实现调用；

```
1. Activity activity = (Activity) imageView.getContext();
2.     activity.runOnUiThread(new Runnable() {
3.
4.         @Override
5.         public void run() {
6.             imageView.setImageBitmap(bitmap);
7.         }
8.     });
```

2、用 `View.post(Runnable r)`方法更新 UI

```
1. imageView.post(new Runnable(){
2.
3.     @Override
4.     public void run() {
5.         imageView.setImageBitmap(bitmap);
6.     }
7.
8. });
```

4、子线程中能不能 **new handler**? 为什么?

不能,如果在子线程中直接 `new Handler()`会抛出异常 `java.lang.RuntimeException`:

Can't create handler inside thread that has not called

在没有调用 `Looper.prepare()`的时候不能创建 `Handler`,因为在创建 `Handler` 的源

码中做了如下操作

`Handler` 的构造方法中

```
public static Looper myLooper() {
    return sThreadLocal.get();
}
```

```
mLooper = Looper.myLooper();
```

```
if (mLooper == null) {
```

```
    throw new RuntimeException(
```

```
        "Can't create handler inside thread that has not called
```

```
Looper.prepare()");
```

```
}
```

五、Android 中的动画

1、Android 中的动画有哪几类，它们的特点和区别是什么


Android 中动画分为两种，一种是 **Tween** 动画、还有一种是 **Frame** 动画。

Tween 动画，这种实现方式可以使视图组件移动、放大、缩小以及产生透明度的变化；

Frame 动画，传统的动画方法，通过顺序的播放排列好的图片来实现，类似电影。

2、如何修改 Activity 进入和退出动画

可以通过两种方式，一是通过定义 **Activity** 的主题，二是通过覆写 **Activity** 的 **overridePendingTransition** 方法。

 通过设置主题样式


在 **styles.xml** 中编辑如下代码：

```
<style name="AnimationActivity" parent="@android:style/Animation.Activity">
    <item name="android:activityOpenEnterAnimation">@anim/slide_in_left</item>
    <item name="android:activityOpenExitAnimation">@anim/slide_out_left</item>
    <item name="android:activityCloseEnterAnimation">@anim/slide_in_right</item>
    <item name="android:activityCloseExitAnimation">@anim/slide_out_right</item>
</style>
```

添加 **themes.xml** 文件：

```
<style name="ThemeActivity">
    <item name="android:windowAnimationStyle">@style/AnimationActivity</item>
    <item name="android:windowNoTitle">>true</item>
</style>
```

在 **AndroidManifest.xml** 中给指定的 **Activity** 指定 **theme**。

 覆写 `overridePendingTransition` 方法

```
overridePendingTransition(R.anim.fade, R.anim.hold);
```

3、属性动画，例如一个 **button** 从 **A** 移动到 **B** 点，**B** 点还是可以响应点击事件，这个原理是什么？

补间动画只是显示的位置变动，**View** 的实际位置未改变，表现为 **View** 移动到其他地方，点击事件仍在原处才能响应。而属性动画控件移动后事件相应就在控件移动后本身进行处理

六、ContentObserver 内容观察者作用及特点

一、ContentObserver 目的是观察(捕捉)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。触发器分为表触发器、行触发器，相应地 ContentObserver 也分为“表 ContentObserver”、“行” ContentObserver，当然这是与它所监听的 **Uri MIME Type** 有关的。

1) 注册 ContentObserver 方法

```
public final void registerContentObserver(Uri uri, boolean  
notifyForDescendents, ContentObserver observer)
```

功能：为指定的 **Uri** 注册一个 ContentObserver 派生类实例，当给定的 **Uri** 发生改变

时，回调该实例对象去处理。

参数: uri 表示需要观察的 Uri

notifyForDescendents 为 false 表示精确匹配，即只匹配该 Uri。 为 true 表示可以同时匹配其派生的 Uri。

取消注册 ContentObserver 方法

```
public final void unregisterContentObserver(ContentObserver observer)
```

功能：取消对给定 Uri 的观察

参数： observer ContentObserver 的派生类实例

ContentObserver 类介绍

构造方法 ContentObserver(Handler h)

void onChange(boolean selfChange) 功能：当观察到的 Uri 发生变化时，回调该方法去处理。所有 ContentObserver 的派生类都需要重载该方法去处理逻辑。

3.观察特定 Uri 的步骤如下：

- 1、创建我们特定的 ContentObserver 派生类，必须重载父类构造方法，必须重载 onChange()方法去处理回调后的功能实现
- 2、利用 context.getContentResolver() 获 ContentResolver 对象，接着调用 registerContentObserver()方法去注册内容观察者
- 3、在不需要时，需要手动的调用 unregisterContentObserver()去取消注册。

例子：监听短信内容变化

在 Activity 中：


```
public class Day0108_contentobserverActivity extends Activity {

    private Handler handler = new Handler(){

        public void handleMessage(android.os.Message msg) {

            switch (msg.what) {

                case 100:

                    String body = (String) msg.obj;

                    TextView tv = (TextView) findViewById(R.id.tv);

                    tv.setText(body);

                    break;

            }

        }

    };

    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        ContentResolver cr = getContentResolver();

        ContentObserver smsObserver = new

SmsContentObserver(this,handler);

        //第二个参数,true 表示观察所有有关短信的

        cr.registerContentObserver(Uri.parse("content://sms"), true,
```

```
smsObserver);

        //content://sms/inbox //收件箱

        //content://sms/sent //已发送

        //content://sms/draft //草稿箱

        //content://sms/outbox //发件箱

        //content://sms/failed //失败短信

        //content://sms/queued //待发队列

    }

}

//SmsContentObserver 代码如下：

public class SmsContentObserver extends ContentObserver {

    private Handler handler;

    private Context context;

    public SmsContentObserver(Context context,Handler handler) {

        super(handler);

        this.handler = handler;

        this.context = context;

    }

    @Override

    public void onChange(boolean selfChange) {

        ContentResolver cr = context.getContentResolver();
```

```
        Cursor c = cr.query(Uri.parse("content://sms/inbox"), null, "0",
null, "date desc");

        StringBuilder sb = new StringBuilder();
        while(c.moveToNext()){

            //发件人手机号码

            String sendNumber = c.getString(
c.getColumnIndex("address"));

            //信息内容

            String body = c.getString(c.getColumnIndex("body"));

            //readType 表示是否已经读

            int hasRead = c.getInt(c.getColumnIndex("read"));
            if(hasRead == 0){//表示短信未读

                System.out.println("短信未读"+sendNumber);

            }

            sb.append(sendNumber+": "+body+"\n");

        }

        handler.obtainMessage(100,sb.toString()).sendToTarget();

    }

}
```

项目框架的使用 (★★★)

一、 自我介绍

从姓名，工作多长时间，及项目开发的周期来说。

(1) 姓名

(2) 工作时间

(3) 开发周期

例如， 你好，我是 XXX，到现在为止我工作了一年半，期间有做过三款 app。最近的一个项目开发了两个月。（这时候可以把你准备好的手机里的 app 打开给面试官看）

二、 开发中都使用过哪些框架、平台

1. EventBus（事件处理）

2. xUtils（网络、图片、ORM）

xUtils 分为四大模块：

① DbUtils 模块：Android 中的 orm 框架（对象关系映射，它的作用是在关系型数据库和业务实体对象之间作为一个映射），一行代码就可以进行增删改查。（Logo 新闻内容缓存到数据库 当没有网络的时候）

② ViewUtils 模块：android 中的 ioc 框架(生命周期由框架控制)，完全注解的方式就可以进行对 UI 绑定和事件的绑定。

③ HttpUtils 模块：（请求服务器 客户端 传过去标示 head=" md5" ）

- a. 支持同步，异步方式的请求。
- b. 支持大文件上传，上传大文件不会 oom(内存溢出)。
- c. 支持 GET,POST,DELETE 请求。

④ BitmapUtil 模块：

可以先说下三级缓存的原理：

- 1. 从缓存中加载。
- 2. 从本地文件中加载（数据库，SD）
- 3. 从网络加载。

a.加载 bitmap 的时候无需考虑 bitmap 加载过程中出现的 oom(内存溢出)和 android 容器快速滑动的时候出现的图片错位等现象。（16M）

b. 支持加载网络图片和本地图片。

c. 内存管理使用的 lru 算法（移除里面是有频率最少的对象），更好的管理 bitmap 的内存。

d.可配置线程加载的数量，缓存的大小，缓存的路径，加载显示的动画等。

清除缓存是怎么做的？

- (1) 清除内存的缓存。
- (2) 数据库，SD。

注：需要添加一下权限<uses-permission

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

```
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

3. JPush（推送平台）

推送的好处：

- a. 及时主动性。（这是推送服务最基本的特点，即当有新的信息需要提交时，依据传送信息的类型和重要性不同，推送软件会主动提醒用户接收新信息。从而提高了用户获取信息的及时性。）
- b. 针对目的性。（推送服务提供的信息是根据用户的特定需求定的，这充分体现了用户的个性化需求。这种个性化的服务还是动态的，用户只需在定制之初描述信息需求，推送软件就会自动跟踪用户的使用倾向，实时地完成特定信息的推送。）
- c. 便捷高效性。（用户只需输入一次信息请求，就可获得连续的信息服务。推送服务还采用信息代理机制，可以自动跟踪用户的信息需求。这样的推送服务既节省了用户主动拉取的时间，又减少了冗余信息的传递提高了信息的匹配度，从而大大方便了用户，提高了效率。）

我们在项目中主要使用的是极光推送，在极光的官网里（<https://www.jpush.cn/>）下载 android 的 demo，将 demo 中的 aapid 换成自己申请的，测试推送，然后集成到自己的项目中去。

4. 友盟（统计平台）
5. 有米（优米）（广告平台）
6. 百度地图

1) 下载百度地图移动版 API(Android)开发包

要在 Android 应用中使用百度地图 API，就需要在工程中引用百度地图 API 开发包，这个开发包包含两个文件：baidumapapi.jar 和 libBMapApiEngine.so。下载地址：

http://dev.baidu.com/wiki/static/imap/files/BaiduMapApi_Lib_Android_1.0.zip

2) 申请 API Key

和使用 Google map api 一样，在使用百度地图 API 之前也需要获取相应的 API Key。百度地图 API Key 与你的百度账户相关联，因此您必须先有百度帐户，才能获得 API Key；并且，该 Key 与您引用 API 的程序名称有关。

百度 API Key 的申请要比 Google 的简单多了，其实只要你有百度帐号，应该不超过 30 秒就能完成 API Key 的申请。申请地址：<http://dev.baidu.com/wiki/static/imap/key/>

3) 创建一个 Android 工程

这里需要强调一点：百度地图移动版 api 支持 Android 1.5 及以上系统，因此我们创建的工程应基于 Android SDK 1.5 及以上。

工程创建完成后，将 baidumapapi.jar 和 libBMapApiEngine.so 分别拷贝到工程的根目录及 libs/armeabi 目录下，并在工程属性->Java Build Path->Libraries 中选择 “Add JARs”，选定 baidumapapi.jar，这样就可以在应用中使用百度地图 API 了。

7. bmob（服务器平台、短信验证、邮箱验证、第三方支付）

8. 阿里云 OSS（云存储）

9. ShareSDK（分享平台、第三方登录）

SDK 简介：ShareSDK 是为 iOS 的 APP 提供社会化功能的一个组件，开发者只需 10 分钟即可集成到自己的 APP 中，它不仅支持如 QQ、微信、新浪微博、腾讯微博、开心网、人人网、豆瓣、网易微博、搜狐微博、facebook、twitter、google+ 等国内外主流社交平台，还有强大的统计分析管理后台，可以实时了解用户、信息流、回流率、传播效应等数据，有效的指导日常运营与推广，同时为 APP 引入更多的社会化流量。

主要功能：

- a. 支持分享到主流的各大平台上。（国内主要的分享平台：QQ，微信，新浪微博，腾讯微博 国外的：facebook twitter google+）
- b. 支持获取授权用户资料及其他用户资料，可以通过 sdk 制作使用新浪微博登录，QQ 登录等。
- c. 支持关注官方微博，支持@好友，插入话题，图片。
- d. 支持一键分享，用户可以一次性将内容分享至全部的社交平台。

使用：

- （1）获取 SharedSDK。（SharedSDK 官网：<http://wiki.mob.com/>）
- （2）将 SharedSDK 集成（导入）到项目的 libs 目录下。
- （3）配置 AndroidManifest.xml 权限

```
1<uses-permission android:name="android.permission.READ_CONTACTS" />
2<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```



```
3<uses-permission
4android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
5<uses-permission
6android:name="android.permission.ACCESS_NETWORK_STATE" />
7<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
8<uses-permission android:name="android.permission.INTERNET" />
9<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.GET_TASKS" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
/>
```

(4) 添加代码，启动 SDK。`SMSSDK.initSDK(this, "<您的 appkey>", "<您的 appsecret>");`

10. Gson（解析 json 数据框架）

根据服务器返回的 Gson 数据来设计类的模型，让 Gson 解析字符串为对应的对象模型。简单来讲就是 根据 json 的数据结构定义出相应的 javabean ---> "new" 出

Gson 的实例 gson----> `gson.fromJson(jsonString, JavaBean.class)` 即可。

面试概要：

可以先说下 Gson 的作用，然后在向后拓展下。（Gson 呢，是 google 提供的一个快速解析 json 数据的开源框架,原来我们解析数据的时候都是 `jsonObject jsonArray` 一层层解析，我发现这样层层解析很浪费时间，于是我在业余时间研究了 Gson，Gson 满足了我们

快速开发的特性，只要从服务器拿到 json 数据用 Gson 解析，Gson 就会返回一个数据对象，我们就可以直接对数据进行操作了。原来解析可能需要十几分钟的事，现在两三分钟就搞定了)

补充：为什么数据要以 json 形式传输？

- ① 易读性
- ② 高效率

11. imageLoader （图片处理框架）

12. zxing （二维码扫描）

三、 都使用过哪些自定义控件

- 1. pull2RefreshListView
- 2. LazyViewPager
- 3. SlidingMenu
- 4. SmoothProgressBar
- 5. 自定义组合控件
- 6. ToggleButton
- 7. 自定义吐司（Toast）

四、 自定义控件：绘制圆环的实现过程

使用自定义控件绘制一个圆环,需要重现的方法是 `OnDraw()` 实现对 `view` 的绘制，从而输出符合自己需求的 `view` 控件

观察圆环的组成部分：

外层圆+中间百分比文字+不断变化进度的弧形圈

--->分析:每一个组成部分需要的属性,构成几个关键的自定义属性

1:外层圆的颜色

2:弧形进度圈的颜色

3:中间百分比文字的颜色

4:中间百分比文字的大小

5:圆环的宽度(作为进度弧形圈的宽度)

6:*首页当中也有一个圆环进度，为了兼容使用首页的圆环进度,增加一个自定义属

性，绘制进度弧形圈的风格(实心[Fill]，空心[Stroke])

分析完毕-->绘制步骤：

OKHTTP 的实际开发中的用法

1:构造方法当中初始化画笔对象，获取自定义的属性值.

2:重写 `OnDraw` 方法

---2.1:绘制最外层的圆

-关键方法 `canvas.drawCircle(center, center, radius, paint);` //画出圆环

*:计算半径、中心点坐标、画笔设置

```
paint.setColor(roundColor); //设置圆环的颜色
```

```
paint.setStyle(Paint.Style.STROKE); //设置空心
```

```
paint.setStrokeWidth(roundWidth); //设置圆环的宽度---这个宽度也是提供给
```

进度弧形圈绘制的时候覆盖的宽度

```
paint.setAntiAlias(true); //消除锯齿
```

中心点坐标

```
int center = getWidth() / 2; //获取圆心的 x 坐标
```

半径:

```
int radius = (int) (center - roundWidth/2) ---画图说明最容易理解
```

---2.2:绘制中间的百分比文字

```
--关键方法: canvas.drawText(percent + "%", center - textWidth / 2, center +  
textSize / 2, paint); //画出进度百分比
```

测量画笔上的文本宽度

```
float textWidth = paint.measureText(percent + "%");
```

画笔设置

```
paint.setStrokeWidth(0);
```

```
paint.setColor(textColor);

paint.setTextSize(textSize);

paint.setTypeface(Typeface.DEFAULT_BOLD); //设置字体
```

绘制的文字的位置,由参数 2,3 的 X,Y 坐标值决定--圆环的中心点位置显示

X:表示从哪开始绘制,如果你直接中心点开始绘制-->画图说明最容易理解

-->正确的 $X = \text{center} - \text{textWidth} / 2$; $Y = \text{center} + \text{textSize} / 2$ -- (因为 android 坐标系与数学坐标系 Y 轴值是相反的,也可以画图说明,这里的 textsize 就可以代表高度,paint.measureText 测量方法执行之后,默认的文字高度就是根据文字大小计算的,相当于 wrap_content, 所以 textSize 就是本身文字所占的高度值)

*:绘制的进度 要转换 为百分比形式 : $\text{int percent} = (\text{int}) (((\text{float}) \text{progress} / (\text{float}) \text{max}) * 100);$

---2.3:绘制进度弧形圈

---关键方法: `canvas.drawArc(oval, 0, 360 * progress / max, false, paint);` //

根据进度画圆弧

参数解释:

oval: 绘制的弧形的范围轮廓

0:从多少角度开始绘制

$360 * \text{progress} / \text{max}$:绘制弧形扫过的角度对应的区域

false:不包含圆心, 如果是 true, 表示包含圆心

paint:绘制使用的画笔

画笔设置

```
paint.setStrokeWidth(roundWidth); //设置进度弧形圈的宽度，必须保持和外层
```

圆的 StrokeWidth 一致，确保弧形圈绘制的时候覆盖的范围就是外层圆的宽度

```
paint.setColor(roundProgressColor); //设置进度的颜色
```

弧形范围计算

```
RectF oval = new RectF(center - radius, center - radius, center  
    + radius, center + radius); --->画图说明最容易理解
```

```
left:center - radius
```

```
top:center-radius
```

```
right:center+radius
```

```
bottom:center+radius
```

*:注意，因为 progress 是相对于 100 当中占比多少，而弧形总共是按照角度分成 360 分的，所以绘制弧形圈指定参数扫过的区域角度需要计算转换一下

```
=360 * progress / max (max=100)
```

*:对应自定义属性中的最后一个属性是实心还是空心，绘制的时候需要判断一下，兼容首页的圆环进度处理

```
switch (style) {  
    case STROKE:{  
        paint.setStyle(Paint.Style.STROKE);
```

```
        canvas.drawArc(oval, 0, 360 * progress / max, false,
paint); //根据进度画圆弧

        break;

    }

    case FILL:{

        paint.setStyle(Paint.Style.FILL_AND_STROKE);

        if(progress !=0)

            canvas.drawArc(oval, 0, 360 * progress / max,
true, paint); //根据进度画圆弧

        break;

    }

}
```

最后提供一个设置进度，根据进度重新绘制圆环的方法

```
public void setProgress(int progress) {

    if(progress < 0){

        throw new IllegalArgumentException("progress not less
than 0");

    }

    if(progress > max){

        progress = max;

    }

}
```

```
        if(progress <= max){  
            this.progress = progress;  
            postInvalidate();  
        }  
    }  
}
```

五、 自定义控件：摩天轮的实现过程

- ① 摩天轮控件是可以通过触摸旋转的，但旋转的过程中保持子 **View** 的方向。
- ② 摩天轮控件中触摸旋转，惯性转动的基础是自控件的摆放，使用了三角函数来

确定子 **view** 的中心点位置，注意在分析阶段我们把摩天轮的中心点作为参考点(0,0)，在写代码的时候，记得偏移 to 控件左上方。

③ 在触摸旋转和惯性旋转时，我们需要改变所有的孩子的位置，其实也就是孩子中心点与圆心连线的角度，但因为角度间隔相等，所以只需要改变第一个孩子的角度，然后其他孩子与第一个孩子保持角度间隔即可，然后调用 **requestLayout** 重新摆放孩子即可。

④ 触摸旋转和惯性转动，都使用了 **GestureDector** 帮助我们判断触摸事件的类型，以及提供给我们所需要的参数。

⑤ 在触摸旋转中，我们通过计算当前点的角度和上一次点的角度，再算出两点的差，就可以知道需要旋转多少角度了。

- ⑥ 比较难的地方就是把惯性滑动的像素速度转化为角速度，对于没有学过微积分

的同学可能想不到。然后我们使用了值动画不断地改变角度来模拟转动，又使用了减速插值器来模拟减速。

⑦ 因为摩天轮的触摸事件处理都是写在了 `onTouchEvent` 方法中，如果子控件消费了触摸事件，会导致摩天轮的 `onTouchEvent` 方法没有被调用，所以我们需要在 `onInterceptTouchEvent` 方法中返回 `true`，然后再检测单击事件发送给被点击的孩子。

六、 自定义控件：可拖拽排序的 `GridLayout` 的实现过程

⑧ 为什么没有使用 `GridView`，而是使用了 `GridLayout`？

因为 `GridView` 必然需要一个 `Adapter`，而 `Adapter` 需要数据集，如果在拖动中想要处理好三者的关系，在编码实现上具有一定的难度。

`GridLayout` 是安卓 4.0 出现的一个新的布局，不需要适配器即可让每个子 `View` 占据一定的格子，需要注意的是 `GridLayout` 的孩子可以横向、纵向地合并单元格。

⑨ 布局动画是什么？

可详见 <http://www.cnblogs.com/mengdd/p/3305973.html>

布局动画可帮助我们非常简单地处理 `ViewGroup` 的子控件的出现、消失、移动，是安卓 3.0 后出现的一个类。

⑩ `Drag Drop` 框架是怎么回事？

`DragDrop` 框架是安卓 3.0 后出现的，用户可以通过此框架以控件的形式移动“数据”，其重要的方法有两个，都是 `view` 上的：`startDrag` 和 `setOnDragListener`。

`startDrag` 方法要传递 4 个参数，但最重要的就是第二个 `ShadowBuilder`，用于构建跟随手指移动的影子，一般可在创建 `ShadowBuilder` 对象时，传入一个 `view` 作为影子的原形即可；

当产生影子后，安卓系统会发送 `DragEvent` 给当前应用中所有可见的 `View`，如果想要接收 `DragEvent`，可通过 `View` 的 `setOnDragListener` 方法，设置一个拖拽监听；

`DragEvent` 有 6 种类型，分别是开始、结束、进入、退出、移动、放下。需要注意的是，如果影子的中心点在设置了监听的控件之外移动或放下，是不会收到相应事件的。

在不同的事件类型中，还可以获得不同的数据，如图：

<code>getAction()</code> value	<code>getClipDescription()</code> value	<code>getLocalState()</code> value	<code>getX()</code> value	<code>getY()</code> value	<code>getClipData()</code> value	<code>getResult()</code> value
<code>ACTION_DRAG_STARTED</code>	X	X	X	X		
<code>ACTION_DRAG_ENTERED</code>	X	X	X	X		
<code>ACTION_DRAG_LOCATION</code>	X	X	X	X		
<code>ACTION_DRAG_EXITED</code>	X	X				
<code>ACTION_DROP</code>	X	X	X	X	X	
<code>ACTION_DRAG_ENDED</code>	X	X				X

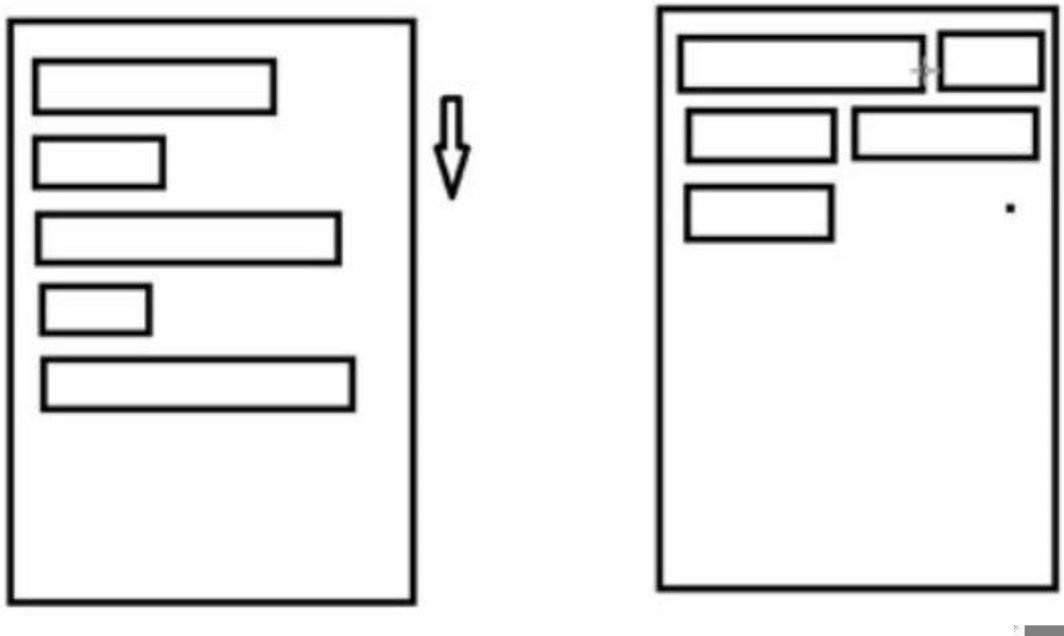
其中，`x` 和 `y` 是对于对设置监听的控件的左上角的位置，而 `Clip` 数据是 `startDrag` 的第一个参数的数据，`LocalState` 是 `startDrag` 的第三个数据，`Result` 是 `Drop` 事件的返回值。

七、 流式布局的实现过程

流式布局的特点以及应用场景

特点：当上面一行的空间不够容纳新的 `TextView` 时候，才开辟下一行的空间

原理图：



场景：主要用于关键词搜索或者热门标签等场景

2.自定义 ViewGroup,重点重写下面两个方法

1) onMeasure:测量子 view 的宽高，设置自己的宽和高

2) onLayout:设置子 view 的位置

onMeasure:根据子 view 的布局文件中属性，来为子 view 设置测量模式和测量值

测量=测量模式+测量值；

测量模式有 3 种：

EXACTLY：表示设置了精确的值，一般当 childView 设置其宽、高为精确值、match_parent 时，ViewGroup 会将其设置为 EXACTLY；

AT_MOST：表示子布局被限制在一个最大值内，一般当 childView 设置其宽、高为 wrap_content 时，ViewGroup 会将其设置为 AT_MOST；

UNSPECIFIED: 表示子布局想要多大就多大，一般出现在 `AdapterView` 的 item 的 `heightMode` 中、`ScrollView` 的 `childView` 的 `heightMode` 中；此种模式比较少见

3) LayoutParams

`ViewGroup.LayoutParams`: 每个 `ViewGroup` 对应一个 `LayoutParams`; 即

`ViewGroup` -> `LayoutParams`

`getLayoutParams` 不知道转为哪个对应的 `LayoutParams`, 其实很简单, 就是如下:

子 `View`.`getLayoutParams` 得到的 `LayoutParams` 对应的就是 子 `View` 所在的父控件的 `LayoutParams`;

例如, `LinearLayout` 里面的子 `view`.`getLayoutParams`

-> `LinearLayout.LayoutParams`

所以 咱们的 `FlowLayout` 也需要一个 `LayoutParams`, 由于上面的效果图是子 `View` 的 `margin`,

所以应该使用 `MarginLayoutParams`。即 `FlowLayout`->`MarginLayoutParams`

八、 项目的流程

这个在项目中不一定问, 但是大家要知道项目开发的流程, 作为有经验的程序员程序的流程是一定知道的。

按时间轴来排列:

立项: 确定项目、负责人、开发的周期、成本、人力、物力

需求: 文档、原型图

开发: 编码

测试：测试人员

上线：产品部门

维护：修复新的 bug

升级:改版、添加新的功能

九、 项目中常见的问题（11.9 更新）

1. 新闻详情页 **WebView** 是怎么使用的？

WebView 是 **View** 的子类，可以让你在 **activity** 中显示网页。

- 1) 添加网络访问权限。
- 2) 调用 **webview** 的 **loadUrl** 加载网络地址。

2. 如何去记录用户是否看过这条新闻？

- 1) 在新闻详情页用 **sharePrefence** 存入当前新闻的 **id**。
- 2) 在新闻列表的 **getView** 方法中判断当前 **id** 在数据中是否存在，存在的话将字体设置为灰色。

3. 在项目中，你是如何缓存数据的？

- 1) 我们可以说下数据是存在数据库中，并延伸到 DBUtil 的使用中去。
- 2) 将数据缓存到 SD 中。

4. 如何清除 **webview** 的缓存

我们可以看下我们下面写的demo运行后的文件结构，打开DDMS的File Explorer：



- 1) **LOAD_CACHE_ONLY**: 不使用网络，只读取本地缓存数据
- 2) **LOAD_DEFAULT**: 根据 **cache-control** 决定是否从网络上取数据。
- 3) **LOAD_CACHE_NORMAL**: API level 17 中已经废弃， 从 API level 11 开始作用同 **LOAD_DEFAULT** 模式
- 4) **LOAD_NO_CACHE**: 不使用缓存，只从网络获取数据.
- 5) **LOAD_CACHE_ELSE_NETWORK**，只要本地有，无论是否过期，或者 no-cache，都使用缓存中的数据。
- 6) 总结：根据以上两种模式，建议缓存策略为，判断是否有网络，有的话，使用 **LOAD_DEFAULT**,

无网络时，使用 `LOAD_CACHE_ELSE_NETWORK`。

```
public class MainActivity extends AppCompatActivity { private WebView wView;
private Button btn_clear_cache;

private Button btn_refresh; private static final String APP_CACHE_DIRNAME =
"/webcache"; // web 缓存目录

private static final String URL = "http://blog.csdn.net/coder_pig";

@Override

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main); wView = (WebView)
    findViewById(R.id.wView);

    btn_clear_cache = (Button) findViewById(R.id.btn_clear_cache);
    btn_refresh = (Button) findViewById(R.id.btn_refresh);

    wView.loadUrl(URL); wView.setWebViewClient(new WebViewClient()
    { //设置在 webView 点击打开的新网页在当前界面显示,而不跳转到新的浏览器中

@Override

    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url); return true;
    }
}
```

```
}

});

WebSettings settings = wView.getSettings(); settings.setJavaScriptEnabled(true); //
设置缓存模式

settings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK); // 开启 DOM
storage API 功能 settings.setDomStorageEnabled(true); // 开启 database storage
API 功能 settings.setDatabaseEnabled(true); String cacheDirPath =
getFilesDir().getAbsolutePath() + APP_CACHE_DIRNAME; Log.i("cachePath",
cacheDirPath); // 设置数据库缓存路径 settings.setAppCachePath(cacheDirPath);
settings.setAppCacheEnabled(true);

Log.i("databasepath", settings.getDatabasePath());

btn_clear_cache.setOnClickListener(new View.OnClickListener() {

    @Override

    public void onClick(View v) {

wView.clearCache(true);

    }

});

btn_refresh.setOnClickListener(new View.OnClickListener() { @Override public void
```



```
onClick(View v) {  
  
    wView.reload();  
  
}  
  
}  
  
);  
  
} //重写回退按钮的点击事件 @Override public void onBackPressed()  
{ if(wView.canGoBack()){ wView.goBack(); }else{ super.onBackPressed();  
  
}  
  
}  
  
}
```

上面的示例，我们通过调用WebView的clearCache(true)方法，已经实现了对缓存的删除！

除了这种方法外，还有下述方法：

```
setting.setCacheMode(WebSettings.LOAD_NO_CACHE);
```

```
deleteDatabase( "WebView.db" );和 deleteDatabase( "WebViewCache.db" );
```

```
webView.clearHistory();
```

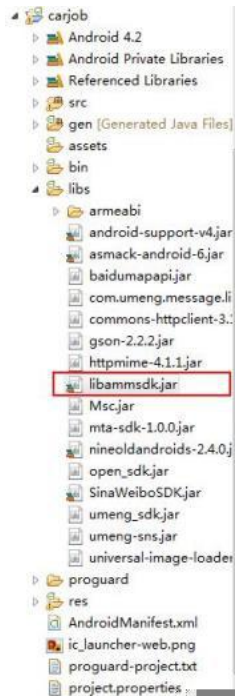
```
webView.clearFormData();
```

```
getCacheDir().delete();
```

手动写 delete 方法，循环迭代删除缓存文件夹！

5. 如何实现第三方登录(以微信为例子)

1) 下载微信官方最新的 sdk，集成到 app 中。如下图：



2) 注册到微信

可以在app的某个Activity的onCreate方法中注册，这里也可以在app的application的 onCreate()方法中注册，这样，在整个 app 的中都可以使用。例如：

```
public class CarApplication extends Application{

.....

public static IWXAPI api;

public void onCreate() {

//注册微信
```

```
api = WXAPIFactory.createWXAPI(this, " 你的应用在微信上申请的  
app_id" , true);
```

```
api.registerApp( "你的应用在微信上申请的 app_id" );
```

在 **app** 的包名目录下新建一个 **wxapi** 目录，然后在此目录下新建 **WXEntryActivity.java** 文件，如下，**app** 的包名为 **carjob.com.cn**，新建的 **wxapi** 目录如下。（注意：一定是包名目录下新建，不要在其他目录新建，否则 **WXEntryActivity.java** 里的 **public void onResp(BaseResp resp)**方法不会被调用）



WXEntryActivity 继承 **Activity**，实现 **IWXAPIEventHandler**，并重写 **protected void onNewIntent(Intent intent)**、**public void onReq(BaseReq arg0)**、**public void onResp(BaseResp resp)**方法。**WXEntryActivity.java** 文件可以见最后附录。

3) 发送微信登录的请求

app 中点击某一个 **view**，发送微信登录的请求如下：

```
final SendAuth.Req req = new SendAuth.Req();
```

```
req.scope = "snsapi_userinfo";
```

```
req.state = "carjob_wx_login";
```

```
CarApplication.api.sendReq(req);
```

其中，**CarApplication.api** 就是第 2 步中注册的 **IWXAPI** 对象。

请求成功后，可拉起微信的授权登录页面，如下。用户点击“确认登录”后，SDK 通过

SendAuth 的 **Resp** 返回数据给调用方（即 **app**），此时 **WXEntryActivity** 中的

`public void onResp(BaseResp resp)`方法被调用（微信、朋友圈分享成功后，此方法同样会被调用），微信登录的返回值说明如下。这里 `app` 可以做相关的处理，见

`WXEntryActivity.java` 文件中的处理，取 `code` 为下一步获取 `access_token` 和 `openid` 等信息做准备。

返回值	说明
ErrCode	ERR_OK = 0(用户同意) ERR_AUTH_DENIED = -4 (用户拒绝授权) ERR_USER_CANCEL = -2 (用户取消)
code	用户换取access_token的code，仅在ErrCode为0时有效
state	第三程序发送时用来标识其请求的唯一性的标志，由第三程序调用sendReq时传入，由微信终端回传，state字符串长度不能超过1K
lang	微信客户端当前语言
country	微信用户当前国家信息

6. 如何实现文件断点上传

在 **Android** 中上传文件可以采用 **HTTP** 方式，也可以采用 **Socket** 方式，但是 **HTTP** 方式不能上传大文件，这里介绍一种通过 **Socket** 方式来进行断点续传的方式，服务端会记录下文件的上传进度，当某一次上传过程意外终止后，下一次可以继续上传，这里用到的其实还是 **J2SE** 里的知识。

这个上传程序的原理是：客户端第一次上传时向服务端发送

“`Content-Length=35;filename=WinRAR_3.90_SC.exe;sourceid=`”这种格式的字符串，服务端收到后会查找该文件是否有上传记录，如果有就返回已经上传的位置，否则返回新生成的 `sourceid` 以及 `position` 为 0，类似 `sourceid=2324838389;position=0` “这样的字

字符串，客户端收到返回后的字符串后再从指定的位置开始上传文件。

7. 什么是 **socket**?

所谓 **Socket** 通常也称作“套接字”，用于描述 **IP** 地址和端口，是一个通信连的句柄，应用程序通常通过“套接字”向网络发送请求或者应答网络请求，它就是网络通信过程中端点的抽象表示。它主要包括以下两个协议：

TCP (Transmission Control Protocol 传输控制协议)：传输控制协议,提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个 **TCP** 连接，之后才能传输数据。**TCP** 提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

UDP (User Datagram Protocol 用户数据报协议)：用户数据报协议，是一个简单的面向数据报的运输层协议。**UDP** 不提供可靠性，它只是把应用程序传给 **IP** 层的数据报发送出去，但是并不能保证它们能到达目的地。由于 **UDP** 在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

8. 谈谈你在工作中是怎样解决一个 **bug**

思路大体和上面差不多，

- 1) 异常附近多打印 **log** 信息；
- 2) 分析 **log** 日志，实在不行的话进行断点调试；
- 3) 调试不出结果，上 **Stack Overflow** 贴上异常信息，请教大牛

- 4) 再多看看代码，或者从源代码中查找相关信息
- 5) 实在不行就 GG 了，找师傅来解决！

9. 怎样对 **android** 进行优化？

- 1) 对 **listview** 的优化。
- 2) 对图片的优化。
- 3) 对内存的优化。
 - 尽量不要使用过多的静态类 **static**
 - 数据库使用完成后要记得关闭 **cursor**
 - 广播使用完之后要注销

10. 谈谈你对 **Bitmap** 的理解, 什么时候应该手动调用 **bitmap.recycle()** (重要)

Bitmap 是 **android** 中经常使用的一个类，它代表了一个图片资源。**Bitmap** 消耗内存很严重，如果不注意优化代码，经常会出现 **OOM** 问题，优化方式通常有这么几种： 1. 使用缓存； 2. 压缩图片； 3. 及时回收；

至于什么时候需要手动调用 **recycle**，这就看具体场景了，原则是当我们不再使用 **Bitmap** 时，需要回收之。另外，我们需要注意，2.3 之前 **Bitmap** 对象与像素数据是分开存放的，

Bitmap 对象存在 java Heap 中而像素数据存放在 Native Memory 中，这时很有必要调用 recycle 回收内存。但是 2.3 之后，Bitmap 对象和像素数据都是存在 Heap 中，GC 可以回收其内存。

11. 请介绍下 AsyncTask 的内部实现，适用的场景是

AsyncTask 内部也是 Handler 机制来完成的，只不过 Android 提供了执行框架来提供线程池来执行相应地任务，因为线程池的大小问题，所以 AsyncTask 只应该用来执行耗时间较短的任务，比如 HTTP 请求，大规模的下载和数据库的更改不适用于 AsyncTask，因为会导致线程池堵塞，没有线程来执行其他的任务，导致的情形是会发生 AsyncTask 根本执行不了的问题。

12. Android 系统中 GC 什么情况下会出现内存泄露呢？ 视频编解码/内存泄露

导致内存泄漏主要的原因是，先前申请了内存空间而忘记了释放。如果程序中存在对无用对象的引用，那么这些对象就会驻留内存，消耗内存，因为无法让垃圾回收器 GC 验证这些对象是否不再需要。如果存在对象的引用，这个对象就被定义为"有效的活动"，同时不会被释放。要确定对象所占内存将被回收，我们就要务必确认该对象不再会被使用。典型的做法就是把对象数据成员设为 null 或者从集合中移除该对象。但当局部变量不需要时，不需明显的设为 null，因为一个方法执行完毕时，这些引用会自动被清理。

Java 带垃圾回收的机制,为什么还会内存泄露呢?

```
Vector v = new Vector(10);
```

```
for (int i = 1; i < 100; i++)    {  
  
    Object o = new Object();  
  
    v.add(o);  
  
    o = null;  
  
} //此时，所有的 Object 对象都没有被释放，因为变量 v 引用这些对象。  
  
Java 内存泄露的根本原因就是 保存了不可能再被访问的变量类型的引用  
  
检测内存工具 heap
```

13. 说说 mvc 模式的原理，它在 android 中的运用。（重要）

MVC 英文即 Model-View-Controller，即把一个应用的输入、处理、输出流程按照 Model、View、Controller 的方式进行分离，这样一个应用被分成三个层——模型层、视图层、控制层。

Android 中界面部分也采用了当前比较流行的 MVC 框架，在 Android 中 M 就是应用程序中二进制的的数据，V 就是用户的界面。Android 的界面直接采用 XML 文件保存的，界面开发变的很方便。在 Android 中 C 也是很简单的，一个 Activity 可以有多个界面，只需要将视图的 ID 传递到 setContentView()，就指定了以哪个视图模型显示数据。

在 Android SDK 中的数据绑定，也都是采用了与 MVC 框架类似的方法来显示数据。在控制层上将数据按照视图模型的要求（也就是 Android SDK 中的 Adapter）封装就可以直接在视图模型上显示了，从而实现了数据绑定。比如显示 Cursor 中所有数据的 ListActivity，其视图层就是一个 ListView，将数据封装为 ListAdapter，并传递给 ListView，

数据就在 `ListView` 中显示。

另一种方式解答：

a.模型（`model`）对象：是应用程序的主体部分，所有的业务逻辑都应该写在该层。

b. 视图（`view`）对象：是应用程序中负责生成用户界面的部分。也是在整个 `mvc` 架构中用户唯一可以看到的一层，接收用户的输入，显示处理结果。

c.控制器（`control`）对象：是根据用户的输入，控制用户界面数据显示及更新 `model` 对象状态的部分

View: 自定义 `View` 或 `ViewGroup`，负责将用户的请求通知 `Controller`，并根据 `model` 更新界面；

Controller: `Activity` 或者 `Fragment`，接收用户请求并更新 `model`；

Model: 数据模型，负责数据处理相关的逻辑，封装应用程序状态，响应状态查询，通知 `View` 改变，对应 `Android` 中的 `datebase`、`SharePreference` 等。

14. 你一般在开发项目中都使用什么设计模式？如何来重构，优化你的代码？

较为常用的就是单例设计模式和工厂设计模式以及观察者设计模式,一般需要保证对象在内存中的唯一性时就是用单例模式,例如对数据库操作的 `SqliteOpenHelper` 的对象。工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。观察者模式定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

15. Android 应用中验证码登陆都有哪些实现方案

验证码应该只有两种获取方式：从服务器端获取图片，通过短信服务，将验证码发送给客户端这两种。

16. 定位项目中，如何选取定位方案，如何平衡耗电与实时位置的精度？

始定位，Application 持有一个全局的公共位置对象，然后隔一定时间自动刷新位置，每次刷新成功都把新的位置信息赋值到全局的位置对象，然后每个需要使用位置请求的地方都使用全局的位置信息进行请求。该方案好处：请求的时候无需再反复定位，每次请求都使用全局的位置对象，节省时间。该方案弊端：耗电，每隔一定时间自动刷新位置，对电量的消耗比较大。

方案 2：按需定位，每次请求前都进行定位。这样做的好处是比较省电，而且节省资源，但是请求时间会变得相对较长。

17. android 应用第二次登录实现自动登录

我来说一个实际案例的流程吧。前置条件是所有用户相关接口都走 https，非用户相关列表类数据走 http。1. 第一次登陆 getUserInfo 里带有一个长效 token，该长效 token 用来判断用户是否登陆和换取短 token 2. 把长效 token 保存到 SharedPreferences 3. 接口请求用长效 token 换取短 token，短 token 服务端可以根据你的接口最后一次请求作为标示，超时时间为一天。4. 所有接口都用短效 token 5. 如果返回短效 token 失效，执行 3 再直接当前接口 6. 如果长效 token 失效（用户换设备或超过两周），提示用户登录。

18. 说说 LruCache 底层原理

LruCache 使用一个 LinkedHashMap 简单的实现内存的缓存，没有软引用，都是强引用。

如果添加的数据大于设置的最大值，就删除最先缓存的数据来调整内存。

maxSize 是通过构造方法初始化的值，他表示这个缓存能缓存的最大值是多少。

size 在添加和移除缓存都被更新值，他通过 safeSizeOf 这个方法更新值。safeSizeOf 默认返回 1，但一般我们会根据 maxSize 重写这个方法，比如认为 maxSize 代表是 KB 的话，那么就以 KB 为单位返回该项所占的内存大小。

除异常外首先会判断 size 是否超过 maxSize，如果超过了就取出最先插入的缓存，如果不为空就删掉，并把 size 减去该项所占的大小。这个操作将一直循环下去，直到 size 比 maxSize 小或者缓存为空。

19. jni 的调用过程?

1. 安装和下载 Cygwin，下载 Android NDK。
2. 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。

20. 手机 APP 安全登录的几种方式

一、登录过程的用户认证，常见的手段有密码加密传输、动态密码、验证码等。

1、密码加密。

目前互联网行业的移动 APP 有不少在使用最简单的做法：根据密码生成一个散列值，把散列值发送给服务器。服务器计算库中用户密码的散列值，然后和客户端传来的散列值比较，一致的话，登录成功。

如果安全性要求更高一些的话，常见的做法就是公钥加密。具体做法是这样，登录前先向服务器请求一个公钥密钥，用公钥密钥加密一串根据密码生成的散列值，然后发送给服务器。服务器使用私钥密钥解密，然后与根据数据库中的用户密码计算出来的散列值进行比较，一致的话，登录成功。当然，还可以做的更优化一些，就是控制公钥密钥的有效期来增强安全性，比如公钥 10 秒钟失效、只能使用一次等。关于公钥加密，可以参考这篇文章：

http://www.360doc.com/content/11/0406/17/4146412_107621805.shtml

2、动态密码。

关于动态密码，其本质就是选择另外一种可以识别用户身份唯一性的方式来和用户的静态密码一起做用户认证。具体常见的几种实现手段，可以参考这篇文章：

<http://baike.soso.com/v5973952.htm>

目前市面上适合 App 使用的最常见的方式是利用手机短信进行动态密码认证。即用户常规登录时，如果服务器发现有异常，可以向用户手机发送一条包含动态密码的短信，用户在有效期（常见的是 30 秒到 1 分钟）内把用户名、用户密码、动态密码一起发送给服务器进行验证。这个对用户来说，操作门槛比较低，也很方便。

3、验证码。

服务器一旦发现登录有异常，如 IP 变化、短时间内登录次数过多等，会向 App 下发一个图片，用户把图片中要求输入的数据和用户名、用户密码一起提交给服务器。

为了降低用户登录过程的复杂性，通常情况下，用户只需要输入用户名和密码，只有服务器发现异常情况才会启用验证码、动态密码等机制。

二、减少用户输入次数的自动登录。

App 登录成功后，服务器会告诉 App 一个 session，后续交流都使用 session。但通常为了安全起见 session 都是要设置有效期的，从 1 星期到 20 天都见过。那么，为了不让用户在 session 失效后重新登录，减少用户的手动输入用户名和用户密码的次数，引入了“自动登录”概念。

流程如下：

登录成功后，服务器给App下发session的同时，还下发一个认证token，客户端把token做为应用程序的私有数据存储起来。以后，每当 session 过期后，就把 token 发送给服务器获取新的 session。整个过程都是对用户透明的，对用户来说，输入一次用户名和密码后，就再也没有登录这个事情了。

当然，这种自动登录的前提，是能保证 token 的安全性远大于 session。我们知道，由于手机 OS 的安全机制，token 做为应用程序的私有数据，对其它应用是不可见的，可以保证 token 的安全性。我们还可以再上一个锁，把 token 和用户使用 App 的那个设备做绑定。可供选择的绑定数据有 imsi,mac 等。这样的话，只要用户手机不丢，就没事。

没有一种安全机制是绝对安全的，我们需要在实际应用过程中综合运用 App 使用场景、具体业务类型、用户习惯等各种方式来平衡安全性、用户体验还有商业应用中很重要的成本。

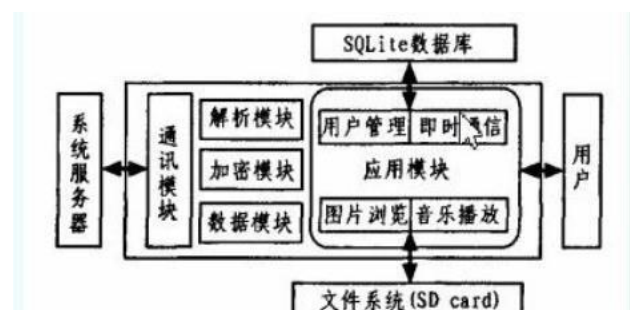
十、 即时通讯是是怎么做的？

答: 1.我在之前的项目中使用了 **asmark** 开源框架实现的即时通讯功能.该框架基于开源的 **XMPP** 即时通信协议，采用 **C / S** 体系结构，通过 **GPRS** 无线网络用 **TCP** 协议连接到服务器，以架设开源的 **Openfn'e** 服务器作为即时通讯平台。

2. 客户端基于 **Android** 平台进行开发。负责初始化通信过程，进行即时通信时，由客户端负责向服务器发起创建连接请求。系统通过 **GPRS** 无线网络与 **Internet** 网络建立连接，通过服务器实现与 **Android** 客户端的即时通信脚。

3.服务器端则采用 **Openfire** 作为服务器。允许多个客户端同时登录并且并发的连接到一个服务器上。服务器对每个客户端的连接进行认证，对认证通过的客户端创建会话，客户端与服务器端之间的通信就在该会话的上下文中进行。

客户端的补充内容:



a. 客户端分为五大模块开发:

通讯模块负责与服务器建立通讯旧。通过创建 3 个线程来进行处理。分别负责消息的发送、接收和系统信息的发送。

解析模块主要用来解析 XML 数据流根据解析元素不同类型封装成不同的数据对象。

数据模块定义整个客户端中大部分的数据类型和对象；

应用模块包括即时通信、图片浏览和音乐播放。是客户端和用户交流的接口。

加密模块对发送和接收的消息进行加解密。以确保通讯数据的安全。

b. xmpp 协议补充内容:

1.XMPP（可扩展消息处理现场协议）是基于可扩展标记语言（XML）的协议，它用于即时消息（IM）以及在线现场探测。它在促进服务器之间的准即时操作。这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息，即使其操作系统和浏览器不同。

2.XMPP 协议的优势

与 HTTP 相比，XMPP 具有如下的优势：

- （1）能够“推送”数据，而不是“拉”。
- （2）防火墙友好(???)
- （3）牢固的身份验证和安全机制
- （4）为许多不同的问题提供大量即开即用的工具

3.XMPP 协议的不足

每种协议都有各自的优缺点，在许多场合中 XMPP 并不是完成任务的最佳工具或受到某种限制。

- (1) 有状态协议
- (2) 社区和部署不及 HTTP 广泛
- (3) 对于简单的请求，其开销比 HTTP 大
- (4) 仍然需要专门的实现

4.XMPP 协议的数据格式

在 XMPP 中，各项工作都是通过在一个 XMPP 流上发送和接收 XMPP 节来完成的。核心 XMPP 工具集由三种基本节组成，这三种节分别为<presence>、出席<message>、<iq>。

XMPP 流由两份 XML 文档组成，通信的每个方向均有一份文档。这份文档有一个根元素<stream:stream>，这个根元素的子元素由可路由的节以及与流相关的顶级子元素构成。

下面给出一段简短的 XMPP 会话：roster 花名册

拉取联系人

```
<stream:stream>

  <iq type='get'>

    <query xmlns='jabber:iq:roster'/>
```



```
</iq>

<presence type=' available' >

<message from='laowang@itcast.cn' />

</stream:stream>
```

简单解释下：

- (1) laowang 登陆后，将自己的第一节（一个<iq>元素）发送出去，这个<iq>元素请求 laowang 的联系人列表。
- (2) 接下来，他使用<presence>节通知服务器他在线并且可以访问。
- (3) 当 laowang 注意到 cui 也在线时，他发送了一条<message>节给 cui。
- (4) 最后，laowang 发送了一个<presence>节告诉服务器自己不可访问并关闭<stream:stream>元素，然后结束会话。

所有三种节都支持一组通用的属性：

- (1) from
- (2) to
- (3) type
- (4) id

5.1 presence 节

<presence>节控制并报告实体 Jid 的可访问性，

Mode: 在线 chat、离线 away、离开 xa、请勿打扰 dnd

Type:available, unavailable,subscribe,subscribed,unsubscribe,

Unsubscribed,error

此外，这个节还用来建立和终止向其他实体发布出席订阅。

普通 presence 节

普通<presence>节不含 type 属性，或者 type 属性值为 unavailable 或 error。type 属性没有 available 值，因为可以通过缺少 type 属性来指出这种情况。

用户通过发送不带 to 属性，直接发往服务器的<presence>节来操纵自己的出席状态。

```
<presence/>

<presence type='unavailable'/>

<presence>

    <show>away</show>

    <status>at the ball</status>

<presence/>

<presence>
```

```
<status>touring the countryside</status>

<priority>10</priority>

</presence>

<presence>

    <priority>10</priority>

</presence>
```

简单解释：

(1) `<show>` 元素用来传达用户的可访问性，只能出现在 `<presence>` 节中。该元素的值可以为：`away`、`chat`、`dnd` 和 `xa`，分别表示离开、有意聊天、不希望被打扰和长期离开。

(2) `<status>` 元素是一个人类可读的字符串，在接收者的聊天客户端中，这个字符串一般会紧挨着联系人名字显示。

(3) `<priority>` 元素用来指明连接资源的优先级，介于 `-128~127`，默认值为 `0`。

扩展 `presence` 节

开发人员希望能够扩展 `<presence>` 节以包含更详细的信息，比如用户当前听的歌或个人的情绪。

因为 `<presence>` 节会广播给所有联系人，并且在 `XMPP` 网络流量中占据了很大的份额，所以不鼓励这种做法。这类扩展应该交给额外信息传送协议来处理。

出席订阅

用户的服务器会自动地将出席信息广播给那些订阅该用户出席信息的联系人。类似的，用户从所有他已经出席订阅的联系人那里接收到出席更新信息。

与一些社交网络和 IM 系统不同，在 XMPP 中，出席订阅是有方向的。我订阅了你的出席信息，但是你并不一定订阅了我的出席信息。

可以通过设置 **type** 的值来识别出席订阅节：**subscribe**(请求加好友)、**unsubscribe**（删除好友）、**subscribed**（答应加好友的请求）、**unsubscribed**（拒绝加好友的请求）。

```
public enum Type {  
  
    /**  
     * The user is available to receive messages (default).  
     */  
    available,  
  
    /**  
     * The user is unavailable to receive messages.  
     */  
    unavailable,  
  
    /**
```

* Request subscription to recipient's presence.

*/

subscribe,

/**

* Grant subscription to sender's presence.

*/

subscribed,

/**

* Request removal of subscription to sender's presence.

*/

unsubscribe,

/**

* Grant removal of subscription to sender's presence.

*/

unsubscribed,

```
/**  
  
 * The presence packet contains an error message.  
  
 */  
  
error  
  
}
```

定向出席

定向出席是一种直接发给另一个用户或其他实体的普通<presence>节，这种节用来向那些没有进行出席订阅（通常因为只是临时需要出席信息）的实体传达出席状态信息。

message 节

<message>节用来从一个实体向另一个实体发送消息。这些可以是简单的聊天信息，也可以是任何结构化信息。例如，绘制指令、游戏状态和新游戏变动状况。

<message>属于发送后不管的类型，没有内在的可靠性，就像电子邮件一样。在有些情况下（比如向不存在的服务器发送信息），发送者可能会收到一个错误提示节，从中了解出现的问题。可以通过在应用程序协议中增加确认机制来实现可靠传送。

消息类型

<message> 节有几种不同的类型，这些类型由 **type** 属性指出，可取的值有：**chat**、**error**、**normal**、**groupchat** 和 **headline**。该属性是可选的，如果没有指定 **type** 值，默认为 **normal**。

normal a normal text message used in email like interface

groupchat Chat message sent to a groupchat server for group chats.

chat Chat message sent to a groupchat server for group chats

消息内容

尽管<message> 节可以包含任意扩展元素，但<body>和<thread>元素是为向消息中添加内容提供的正常机制。这两种子元素均是可选的。

Sets the thread id of the message, which is a unique identifier for a sequence of "chat" messages. thread the thread id of the message.

IQ 节

<iq>节表示的是 Info/Query，它为 XMPP 通信提供请求和响应机制。它与 HTTP 协议的基本工作原理非常相似，允许获取和设置查询，与 HTTP 的 GET 和 POST 动作类似。

<iq>节有四种，通过 **type** 属性区分，其中两种请求 **get** 和 **set**，两种响应 **result** 和 **error**。

每一个 IQ-get 或 IQ-set 节都必须接收响应的 IQ-result 和 IQ-error 节。此外，每一对<iq>必须匹配 **id** 属性。

Get: 获取当前域值

Set: 设置或替换 `get` 查询的值

Result: 说明成功的响应了先前的查询

Error: 查询和响应中出现的错误

The base IQ (Info/Query) packet. IQ packets are used to get and set information on the server, including authentication, roster operations, and creating accounts. Each IQ packet has a specific type that indicates what type of action is being taken: "get", "set", "result", or "error".

IQ packets can contain a single child element that exists in a specific XML namespace. The combination of the element name and namespace determines what type of IQ packet it is.

```
query xmlns="jabber:iq:auth"
```

十一、 设计模式六大原则

1.单一职责原则:不要存在多于一个导致类变更的原因。

通俗的说:即一个类只负责一项职责。

2.里氏替换原则:所有引用基类的地方必须能透明地使用其子类的对象。

通俗的说:当使用继承时。类 B 继承类 A 时，除添加新的方法完成新增功能 P2 外，尽量不要重写父类 A

的方法，也尽量不要重载父类 A 的方法。如果子类对这些非抽象方法任意修改，就会对整个继承体系造

成破坏。子类可以扩展父类的功能，但不能改变父类原有的功能。

3.依赖倒置原则:高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应

该依赖抽象。

通俗的说:在 java 中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的

是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。依赖

倒置原则的核心思想是面向接口编程。

4.接口隔离原则:客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

通俗的说:建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，

我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

5. **迪米特法则**: 一个对象应该对其他对象保持最少的了解

通俗的说: 尽量降低类与类之间的耦合。

6. **开闭原则**: 一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

通俗的说: 用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件

架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根

据需求重新派生一个实现类来扩展就可以了。

十二、 第三方登陆

参考资料: <http://blog.csdn.net/infsafe/article/details/10209401>



- 1、你们需要支持用户注册
- 2、你们需要在应用登录的时候提供第三方平台的图标
- 3、用户点击第三方平台图标以后，你们尝试判断用户是否已经授权
- 4、如果用户授权，获取他的唯一识别符，比方说 WeiboDb 里面的 weiboId 这个字段
- 5、如果用户没有授权，引导用户授权，授权成功后也可以获取 weibo Id

6、然后用这个唯一识别符登录你们的系统，如果用户已经注册，则应该让用户登录到你们的系统，流程结束

7、如果你们的系统发现用户没有注册，引导用户进入你们应用的注册页面，并通过 **share sdk** 的 **showuser** 方法获取用户资料，自动帮助用户完成注册资料的填写，然后等待用户确认

8、如果用户确认了注册信息，你们的应用就根据他的信息完成这注册操作，如果操作成功，则应该让用户登录到你们的系统，流程结束

十三、 第三方支付

申请流程

注册支付宝账号??进行实名认证??提交审核资料??审核通过

备注：申请通过后会获得：合作者身份 ID（PID），该 ID 在项目配置中需要用到

开发流程：

第一步：

下载 **API** 开发文档后，即可获取官方 **Demo**，该 **Demo** 中需要将审核通过后获取的 **PID** 替换，并且输入支付宝收款账户即可。这里非常简单，就不过多叙述。

第二步：

官方 **Api** 开发文档中，存在一个 **openssl** 的文件夹，该文件夹主要是用于生成支付宝所需要用到的公钥以及私钥。打开该文件夹可以看到详细的生成方式，根据提示生成公钥及私钥，请注意，密钥需要经过 **pkcs8** 二次加密。

第三步：

将生成的公钥和私钥配置到 Demo 中。

第四步（可省略）：

为了方便后期维护，建议将支付宝相关的方法及配置项抽取出来做为单独的一个类，后期需要使用直接调用即可。

十四、 常见框架分析

1.Otto 与 EventBus

这两个框架的实现原理差不多，在开发中常用 Otto.

我们假设这样一种业务场景，现在在做一款及时聊天应用，我们在聊天页面进行收发信息，同时也要实时更新前一页面的聊天记录，这时我们该如何去实现？可以使用的是广播接收器 **BroadcastReceiver**，在接收和发送消息的时候就不停去发送广播，然后在需要实时更新的地方进行接收更新。实现的思想比较简单，也不存在代码上的耦合问题，但是有个弊端。弊端就是需要去在很多地方实现 **BroadcastReceiver**，代码虽不算冗余，但比较多，看起来很是不爽。使用 **Otto** 能解决代码体积的问题。**Otto** 是一款目前比较流行事件总线框架，旨在保持应用各页面和模块之间通信高效的前提下，对应用进行解耦。**Otto** 是基于观察者设计模式，简单来说，如果你想订阅某个消息，使用 **@Subscribe** 注解即可进行接收，同时使用

Bus.post(Object obj)进行消息的发布，这样的设计达到了完全的解耦。

Otto 使用过程:

一、Bus 实例化

Bus 这个类是整个框架的灵魂，它负责消息的发布和接收，整个流程都是经过这个 **Bus** 去实现的。**Bus** 的实例化推荐使用单例，就是说整个应用内只实例化一个 **Bus** 对象，所有的

消息的处理都是经过这单一的实例去实现。因为要实现消息的接受者接收到发布的消息，一定要经过同一个 **Bus** 对象的处理。**Bus** 的构造器可以接收 **ThreadEnforcer** 类型的参数，**ThreadEnforcer** 其实是一个接口，它自身有两个实现，分别表示 **Bus** 运行在 **Main Thread** 中还是异步线程中。

```
public final class BusProvider {  
    private static final Bus BUS = new Bus(ThreadEnforcer.MAIN);  
  
    public static Bus getInstance() {  
        return BUS;  
    }  
  
    private BusProvider() {  
        // No instances.  
    }  
}
```

二、注册和解绑 Bus

根据具体的业务需求进行 **Bus** 的注册和解绑，对于 **android** 中的组件，一般是基于生命周期方法中去实现；同时如果是任意你自定义的类中都可以进行。下面展示的是在 **Activity** 和 **Fragment** 里面实现。

```
@Override public void onResume() {  
    super.onResume();  
    BusProvider.getInstance().register(this);  
}  
  
@Override public void onPause() {  
    super.onPause();  
    BusProvider.getInstance().unregister(this);  
}
```

三、消息的发布

发布消息是整个框架中最重要的部分，它允许你告诉所有的订阅者一个事件已经触发。任何一个类的实例对象都可以通过总线 **Bus** 去发布，同时也只能被订阅这种对象的接受者所接收。下面展示的是通过 **Bus** 去发布一个消息，消息的内容是 **LocationChangeEvent**，所以

LocationChangeEvent 的接受者都能接收到此发布的消息。注意的是，发布消息只能一个 Object 对象。

```
// Post new location event
lastLatitude = DEFAULT_LAT;
lastLongitude = DEFAULT_LON;
LocationChangeEvent event = new LocationChangeEvent(
    lastLatitude, lastLongitude);
BusProvider.getInstance().post(event);
```

四、消息的订阅

消息的订阅和发布之前都要在当前的类中进行 Bus 的注册。订阅是对消息发布的补充，当消息发布的事件调用之后，对应的消息订阅者就能立即接收到此消息。实现订阅功能是通过自定义方法实现的，方法的名称可以随意，同时还得需要满足三个条件。

```
@Subscribe public void onLocationChanged(LocationChangeEvent event) {
    locationEvents.add(0, event.toString());
    if (adapter != null) {
        adapter.notifyDataSetChanged();
    }
}
```

1、方法前使用@Subscribe 注解

2、访问修饰符为 public

3、单一参数，根据你想订阅的消息进行设置

注：使用之前，记得进行注册；使用完毕，记得释放。

```
@Subscribe public void onLocationChanged(LocationChangeEvent event) {
    locationEvents.add(0, event.toString());
    if (adapter != null) {
        adapter.notifyDataSetChanged();
    }
}
```

五、消息的 produce

当订阅者注册完毕，针对特定的消息，通常也需要获取当前已知的值。这个时候，就需要用到 produce。同样的使用 produce 的方法名称可以随意，同时有三点需要注意。

- 1、方法前使用@produce 注解
- 2、访问修饰符为 public
- 3、无参，返回值是基于订阅者参数类型

```
@Produce public LocationChangedEvent produceLocationEvent() {  
    // Provide an initial value for location based on the last known position.  
    return new LocationChangedEvent(lastLatitude, lastLongitude);  
}
```

当然 Otto 的缺点也是有的，要实现上述订阅/发布模型的功能，付出的代价就是对各个注册 Bus 的类进行反射。如果大量的使用的情况下，对应用的性能多少有点副影响。

实现原理：

这个框架所实现的功能本质是在一个对象中调用任意的另一个或多个对象中的方法，而不需要将这个或者这些对象传入调用者。

如果把它们都传入调用者去调用它们的方法，那么调用者就依赖这些对象，程序的耦合度就高了，所以说这个框架是用来解耦的。

无论 EventBus 还是 Otto 都有一个注册(register)的方法，方法参数是需要订阅事件的对象。register 方法会拿到参数的 class 文件，

Otto 通过反射获取类中有@Subscribe 注解的方法，

将该 Method 对象和参数放入 Otto 内部的一个集合中，

在发布事件时调用 post 方法，post 方法会根据参数类型，在这个集合中找到 register 时放入的相对应 Method 对象，

调用这个 Method 所需要的对象是 register 时传入的参数，所需的参数是 post 时传入的参数，这些都已经有了，直接将它 invoke

EventBus 与 Otto 不同仅在于 Otto 是通过注解来确定哪些方法是需要接收事件的方法，而 EventBus 是通过固定的方法名来确定的，所以在项目上线，代码混淆时，使用了 EventBus

的类都不能混淆，因此在项目中使用 **Otto** 会更多一点。

2.ImageLoader

一个强大的图片加载框架，很好的处理了图片加载的多线程，缓存，内存溢出等问题。

1. 多线程异步加载和显示图片（图片来源于网络、sd 卡、assets 文件夹，drawable 文件夹（不能加载 9patch） 可以加载视频缩略图）

Uri 参数:

`http:// site.com/image.png`

`file:/// mnt/sdcard/image.png`

`file:/// mnt/sdcard/video.mp4`

`content:// media/external/images/media/13`

`content://media/external/video/media/13`

`assets:// image.png`

`drawable://R.drawable.img`

可以这么使用

`assets: Scheme.ASSETS.wrap(path)`

`drawable: Scheme.DRAWABLE.wrap(path)`

`file: Scheme.FILE.wrap(path)`

`content: Scheme.CONTENT.wrap(path)`

`http: Scheme.HTTP.wrap(path)`

`https: Scheme.HTTPS.wrap(path)`

- 2、支持监视加载的过程，可以暂停加载图片，在经常使用的 **ListView**、**GridView** 中，可

以设置滑动时暂停加载，停止滑动时加载图片

3、高度可定制化（可以根据自己的需求进行各种配置，如：线程池，图片下载器，内存缓存策略等）

4、支持图片的内存缓存，SD 卡（文件）缓存

5、在网络速度较慢时，还可以对图片进行加载并设置下载监听

6、减少加载大图片 OOM 的情况

7、避免同一 Uri 加载过程中重复开启任务加载

框架中三个核心类:

ImageLoader

ImageLoaderConfiguration

DisplayImageOptions

ImageLoaderConfiguration 是给 ImageLoader 做配置的, DisplayImageOptions 是显示图片的一些配置. 这两个类都使用了构建器模式.

下面是使用的例子, 代码中的注释写的很详细:



MyApplication.java

最简单的使用范例:

```
public class MyApplication extends Application {  
    @Override  
    public void onCreate() {  
        super.onCreate();  
  
        File cacheDir = StorageUtils.getCacheDirectory(this);  
  
        ImageLoaderConfiguration config = new ImageLoaderConfiguration
```

```
.Builder(this)

.denyCacheImageMultipleSizesInMemory()

.diskCache(new UnlimitedDiskCache(cacheDir))

.diskCacheExtraOptions(480, 800, null)

.build();

ImageLoader.getInstance().init(config);

}

}
```

在 `AndroidManifest` 文件的 `application` 节点下配置:

```
android:name="cn.itcast.demo.app.MyApplication"
```

使用:

```
ImageLoader.getInstance().displayImage(url);
```

`ImageLoader` 显示图片一般使用 `displayImage` 方法, 它的执行流程图:



可以看到就是讲过的三级缓存。

3.BufferKnife 和 AndroidAnnotations

这是两个轻量级的 IOC 框架, 类似 J2ee 的 Spring. 安卓类似的框架还有很多, 这两个用的比较多, 也较具有代表性, 这两个框架的实现原理不是很重要, 但是它的思想必须要了解。

IOC 是面向对象编程中非常重要的一个原则, IOC 的意思是控制反转: Inversion of Control,

控制反转分为依赖注入和依赖查找, 这两个框架是依赖注入。

控制反转指的是创建对象的功能转移了, 可以把它看做是工厂模式的升华, 也是为了解耦

ButterKnife 是通过注解+反射来实现的，作用在于解耦和减少代码书写量。缺点在于使用反射实现，会影响程序运行速度。

注解：其实注解没有什么其它的用途，它只是一个标志，当你需要给某个类，某个方法，或某个属性打一个特定的标记，就可以使用注解。

ButterKnife 的使用比较简单，举个栗子，如果你想自动 `findViewById`，那么需要两步

1. 给属性加上 `@InjectView(R.id.xx)` 括号中的参数是这个控件的 `id`
2. 在 `Activity` 的 `setContentView` 之后调用 `ButterKnife.inject(this);`

实现过程：

`inject` 方法会拿到传入的 `Activity` 的 `class` 对象，

获取类中所有的 `Field`，

判断该属性是否被 `@InjectView` 注解，

如果被注解，获取注解的值

调用 `findViewById()` 传入注解的值，获取 `view` 对象

将 `view` 对象设置给这个 `field`

AndroidAnnotations 框架是另一种实现模式，它在编译时会给注解的类生成一个子类，在这个子类做了初始化和绑定事件等操作，在使用时应该使用这个子类，比如注解 `Activity` 时，在 `AndroidManifest` 文件配置时是配置生成的这个子类而不是原类，子类的类名是原类类名+`_` 比如原类名是 `MainActivity`，生成的子类类名就是 `MainActivity_`。

相比其它 **IOC** 框架它的优点在于没有使用反射运行更快，缺点在于会生成大量类文件，增加 `apk` 体积。

ButterKnife 使用示例：

```
public class MainActivity extends Activity {
```

```
//相当于 findViewById(R.id.tv)

@InjectView(R.id.tv)

TextView tv;

@Override

protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);

    ButterKnife.inject(this);//将对象注入,就能对它用 ButterKnife 的注解

}

//给 id=R.id.tv 的控件设置点击事件

@OnClick(R.id.tv)

public void show(View v){

    Toast.makeText(this,"butterKnif!", Toast.LENGTH_LONG).show();

}

}
```

4.Picasso, ImageLoader, Fresco, Glide 优劣

首先看 **Fresco**, 它的优点是其他几个框架没有的, 或者说是其他几个框架的短板.

Fresco:

优点:

1. 图片存储在安卓系统的匿名共享内存, 而不是虚拟机的堆内存中, 图片的中间缓冲

数据也存放在本地堆内存，所以，应用程序有更多的内存使用，不会因为图片加载而导致 oom，同时也减少垃圾回收器频繁调用回收 Bitmap 导致的界面卡顿，性能更高。

2. 渐进式加载 JPEG 图片，支持图片从模糊到清晰加载
3. 图片可以以任意的中心点显示在 ImageView，而不仅仅是图片的中心。
4. JPEG 图片改变大小也是在 native 进行的，不是在虚拟机的堆内存，同样减少 OOM
5. 很好的支持 GIF 图片的显示

缺点：

1. 框架较大，影响 Apk 体积
2. 使用较繁琐

ImageLoader, Picasso, Glide: 这三者实现机制都差不多

ImageLoader:

比较老的框架，稳定，加载速度适中，缺点在于不支持 GIF 图片加载，使用稍微繁琐，并且缓存机制没有和 http 的缓存很好的结合，完全是自己的一套缓存机制。

Picasso:

使用方便，一行代码完成加载图片并显示，框架体积小，

缺点在于不支持 GIF，并且它可能是想让服务器去处理图片的缩放，它缓存的图片是未缩放的，并且默认使用 ARGB_8888 格式缓存图片，缓存体积大。

Glide:

可以说是 Picasso 的升级版，有 Picasso 的优点，并且支持 GIF 图片加载显示，图片缓存也会自动缩放，默认使用 RGB_565 格式缓存图片，是 Picasso 缓存体积的一半。

5.Volley 和 Ok-http,android-async-http,retrofit

volley 是一个简单的异步 **http** 库，仅此而已。比较适合小的而频繁的 **Http** 请求

缺点是不支持同步，这点会限制开发模式；

不能 **post** 大数据，所以不适合用来上传文件。

android-async-http 与 **volley** 一样是异步网络库，使用了 **nio** 的方式实现，不过 **nio** 更适合大量连接的情况，对于移动平台有点杀鸡用牛刀的味道。**volley** 是封装的 **URLConnection**，**android-async-http** 封装的 **httpClient**，而 **android** 平台不推荐用 **HttpClient** 了，所以这个库已经不适合 **android** 平台了。

okhttp 是高性能的 **http** 库，支持同步、异步，而且实现了 **spdy**、**http2**、**websocket** 协议，**api** 很简洁易用，和 **volley** 一样实现了 **http** 协议的缓存。**picasso** 就是利用 **okhttp** 的缓存机制实现其文件缓存，实现的很优雅，很正确，反例就是 **UIL**（**universal image loader**），自己做的文件缓存，而且不遵守 **http** 缓存机制。

retrofit 与 **picasso** 一样都是在 **okhttp** 基础之上做的封装，项目中可以直接用了。

picasso、**uil** 都不支持 **inbitmap**，项目中有用到 **picasso** 的富图片应用需要注意这点。

okhttp 和 **async http** 是一个基础的通信库，都很强大，但需要自己封装使用才更方便。

另外 **okhttp** 已经被谷歌官方用在 **android** 源码中了。**retrofit** 和 **volley** 是属于比较高级点的封装库了，其中 **retrofit** 是默认使用 **okhttp**，**volley** 也支持 **okhttp** 作为其底层通信的部件。**retrofit** 的特点是使用清晰简单的接口，非常方便，而 **volley** 在使用的时候也还简单，不过要使用高级一点的功能需要自己自定义很多东西。

推荐使用 **retrofit+okhttp**。最新的 **retorfit** 已经支持 **nio** 了，而且 **retorfit** 内置了 **RxJava**。

6. RxJava

RxJava 是一个异步操作库，一个在 **Java VM** 上使用可观测的序列来组成异步的、基于事

件的程序的库.它是仿照.net 的 **Reactive Extensions (Rx)** 类库设计的, 可以帮助我们更容易的进行相应式编程. **RxJava** 的优势在于简洁.

这个简洁不是指代码书写量少, 而是代码逻辑与结构简洁. 异步操作很关键的一点是程序的简洁性, 因为在调度过程比较复杂的情况下, 异步代码经常会既难写也难被读懂. **Android** 创造的 **AsyncTask** 和 **Handler** , 其实都是为了让异步代码更加简洁.

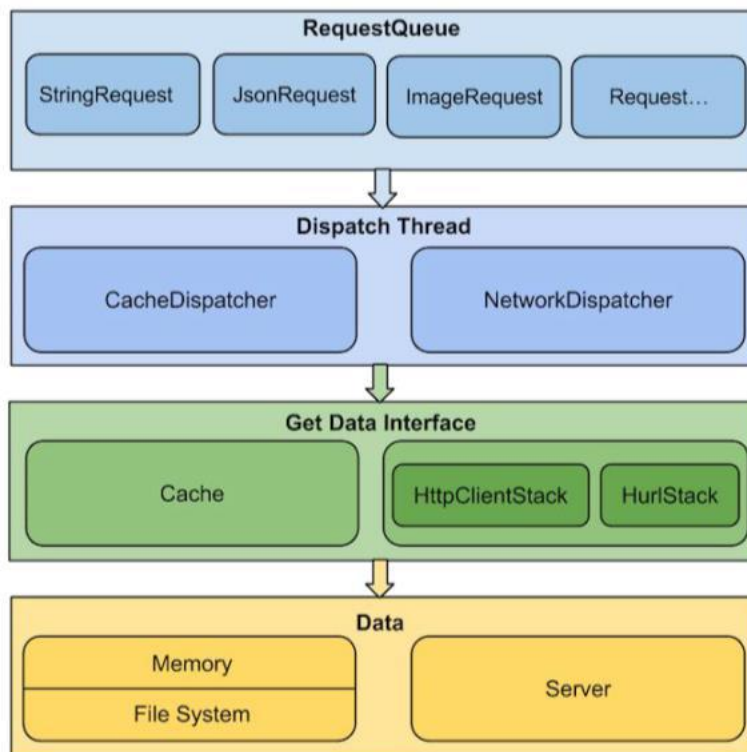
RxJava 的简洁的与众不同之处在于, 随着程序逻辑变得越来越复杂, 它依然能够保持简洁, 能够很大程度提升代码的阅读性, 易于后期升级维护. 他通过一种扩展的观察者模式来实现的, **RxJava** 有四个基本概念: **Observable** (可观察者, 即被观察者)、**Observer** (观察者)、**subscribe** (订阅事件). **Observable** 和 **Observer** 通过 **subscribe()** 方法实现订阅关系, 从而 **Observable** 可以在需要的时候发出事件来通知 **Observer**.

它的缺点在于有一定的学习成本.

7. Volley

Volley 的主要特点

- (1). 扩展性强。**Volley** 中大多是基于接口的设计, 可配置性强。
- (2). 一定程度符合 **Http** 规范, 包括返回 **ResponseCode(2xx、3xx、4xx、5xx)** 的处理, 请求头的处理, 缓存机制的支持等。并支持重试及优先级定义。
- (3). 默认 **Android2.3** 及以上基于 **HttpURLConnection**, 2.3 以下基于 **HttpClient** 实现
- (4). 提供简便的图片加载工具。



上面是 Volley 的总体设计图，主要是通过两种 Dispatch Thread 不断从 RequestQueue 中取出请求，根据是否已缓存调用 Cache 或 Network 这两类数据获取接口之一，从内存缓存或是服务器取得请求的数据，然后交由 ResponseDelivery 去做结果分发及回调处理。也是三级缓存。

Volley 的优缺点：

优点: Volley 的优点在于请求队列的管理，适合小而频繁的请求，如果 app 比较小，网络请求要求不高的情况下可以使用 volley，通常情况下是要结合其他框架一起来使用，比如 volley+okhttp。

缺点: 下载大文件性能不佳, 不支持大文件上传

Volley 简单使用示范:

注意要添加联网权限。

```
public class MainActivity extends Activity {

    @InjectView(R.id.tv)
    TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        ButterKnife.inject(this);
    }

    @OnClick(R.id.tv)
    public void show(View v){

        String url =
        "http://news.xinhuanet.com/photo/2016-01/29/128684788_14540614045291n.jpg";

        RequestQueue requestQueue = Volley.newRequestQueue(this);
        MyResponseListener responseListener = new MyResponseListener();

        ImageRequest imageRequest = new ImageRequest(
            url, //图片的 url

            new MyResponseListener(), //响应回调接口
```

```
        720, //图片宽

        1280, //图片高

        ImageView.ScaleType.FIT_XY,

        Bitmap.Config.RGB_565,

        new MyErrorListener()); //请求错误的回调接口

requestQueue.add(imageRequest); //添加到请求队列

    }

    private class MyErrorListener implements Response.ErrorListener{

        @Override

        public void onErrorResponse(VolleyError error) {

            //Do Something...

        }

    }

    private class MyResponseListener implements Response.Listener<Bitmap>{

        @Override

        public void onResponse(Bitmap response) {

            Drawable drawable = (Drawable) new

            BitmapDrawable(getResources(), response);

            tv.setBackground(drawable);

        }

    }
```

```
}  
  
}
```

Ok-http 使用简单范例

```
public class MainActivity extends Activity {  
  
    @InjectView(R.id.tv)  
    TextView tv;  
  
    private OkHttpClient okHttpClient;  
  
    private Call call;  
  
    private ResponseCallback responseCallBack;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        ButterKnife.inject(this);  
        okHttpClient = new OkHttpClient(); //创建 Client  
        String url =  
        "http://news.xinhuanet.com/photo/2016-01/29/128684788_14540614045291n.jpg"  
        "  
        Request request = new Request.Builder() //创建请求  
            .url(url)
```

```
        .build();

        call = okHttpClient.newCall(request); //创建调用对象

        responseCallBack = new ResponseCallBack(); //创建回调接口

        call.enqueue(responseCallBack); // 调用

    }

    private class ResponseCallBack implements Callback{

        @Override

        public void onFailure(Call call, IOException e) {}

        @Override

        public void onResponse(Call call, Response response) throws IOException {

            InputStream in = response.body().byteStream();

            Bitmap bitmap = BitmapFactory.decodeStream(in);

            final Drawable drawable = (Drawable) new

BitmapDrawable(getResources(), bitmap);

            runOnUiThread(new Runnable() {

                @Override

                public void run() {

                    tv.setBackground(drawable);

                }

            });

        }

    }

};
```

```
    }  
  
    }  
  
}
```

Java 面试题（10.23 更新）（★★）

一、Java 基础（★★）

1、Java 中引用类型都有哪些

（1）在虚拟机内存不足的情况下，也不会回收强引用对象。如果我们将（强引用）对象置为 `null`，会大大加大垃圾回收执行频率。几乎只要我们给出建议（GC），jvm 就会回收。

强引用，例如下面代码：

```
Object o=new Object();  
  
Object o1=o;
```

（2）对于软引用，如果不显式的置为 `null` 的话，和强引用差不多，垃圾回收不会执行。只会等到内存不足的时候才会调用。

（3）对于弱引用，就算你不显式的把他置为 `null`，垃圾回收也会立即执行。

（4）虚引用，相当于 `null`。

2、什么是重载，什么是重写，有什么区别？

重载(Overloading):

(1) **Overloading** 是一个类中多态性的一种表现，让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。

(2) 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同，也可以不相同。无法以返回型别作为重载函数的区分标准。

重写 (Overriding):

(1) 父类与子类之间的多态性，对父类的函数进行重新定义。即在子类中定义某方法与其父类有相同的名称和参数。

(2) 若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。如需父类中原有的方法，可使用 **super** 关键字，该关键字引用了当前类的父类。

3、String、StringBuffer 和 StringBuilder 的区别

JAVA 早期平台提供了两个类：**String** 和 **StringBuffer**，它们可以储存和操作字符串，即包含多个字符的字符数据。这个 **String** 类提供了数值不可改变的字符串。而这个 **StringBuffer** 类提供的字符串 可以进行修 改。当你 知道字符数 据要改变 的时候你就 可以使用 **StringBuffer**。。**String** 类是不可变类，任何对 **String** 的改变都会引发新的 **String** 对象的生成；而 **StringBuffer** 则是可变类，任何对它所指代的字符串的改变都不会产生新的对象。**StringBuilder** 是后面引入的，它与 **StringBuffer** 类的区别在于，新引入的 **StringBuilder** 类不是线程安全的，但其在单线程中的性能比 **StringBuffer** 高。（有兴趣的可以去读下

《Think in Java》描述 `HashTable` 和 `HashMap` 区别的那部分章节比较熟悉的话，就是支持线程同步保证线程安全而导致性能下降的问题）

典型地，你可以使用 `StringBuffers` 来动态构造字符数据。另外，`String` 实现了 `equals` 方法，`new String("abc").equals(newString("abc"))` 的结果为 `true`，而 `StringBuffer` 没有实现 `equals` 方法，所以，`new StringBuffer("abc").equals(newStringBuffer("abc"))` 的结果为 `false`。

4、关键字 `final` 和 `static` 是怎么使用的

`final` 有着“终态的”“这是无法改变的”含义，阻止了多态和继承。

具体使用有：

`final` 类不能被继承，没有子类，`final` 类中的方法默认是 `final` 的。

`final` 方法不能被子类的方法覆盖，但可以被继承。

`final` 成员变量表示常量，只能被赋值一次，赋值后值不再改变。

`final` 不能用于修饰构造方法。

注意：父类的 `private` 成员方法是不能被子类方法覆盖的，因此 `private` 类型的方法默认是 `final` 类型的。

`static` 表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态 `static` 代码块，但是 `Java` 语言中没有全局变量的概念。

被 `static` 修饰的成员变量和成员方法独立于该类的任何对象，`static` 对象可以在它的任何对象创建之前访问，无需引用任何对象。

`static` 前面也可用 `public` 或 `private` 来修饰，其中 `private` 是访问权限限定，`static` 表示不

要实例化就可以使用。

主要用于静态变量，静态方法，**static** 代码块

静态变量：对于静态变量在内存中只有一个拷贝（节省内存），**JVM** 只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

静态方法：静态方法可以直接通过类名调用，任何的实例也都可以调用，因此静态方法中不能用 **this** 和 **super** 关键字，不能直接访问所属类的实例变量和实例方法(就是不带 **static** 的成员变量和成员成员方法)，只能访问所属类的静态成员变量和成员方法。因为实例成员与特定的对象关联！

static 代码块：**atic** 代码块也叫静态代码块，是在类中独立于类成员的 **static** 语句块，可以有多个，位置可以随便放，它不在任何的方法体内，**JVM** 加载类时会执行这些静态的代码块，如果 **static** 代码块有多个，**JVM** 将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次

static 和 **final** 一起使用：

static final 用来修饰成员变量和成员方法，可简单理解为“全局常量”！

对于变量，表示一旦给值就不可修改，并且通过类名可以访问。

对于方法，表示不可覆盖，并且可以通过类名直接访问。

特别要注意一个问题：

对于被 **static** 和 **final** 修饰过的实例常量，实例本身不能再改变了，但对于一些容器类型（比如，**ArrayList**、**HashMap**）的实例变量，不可以改变容器变量本身，但可以修改容器中存放的对象，这一点在编程中用到很多。

5、TCP/IP 协议簇分哪几层？TCP、IP、XMPP、HTTP、分别属于哪一层？（2016.01.24）

通讯协议采用了 4 层的层级结构，每一层都呼叫下一层所提供的网络来完成自己的需求。这 4 层分别为：

应用层：应用程序间沟通的层，如简单电子邮件传输（SMTP）、文件传输协议（FTP）、网络远程访问协议（Telnet）、超文本传输协议(HTTP)、可扩展通讯和表示协议（XMPP）等。

传输层：在此层中，它提供了节点间的数据传送服务，如传输控制协议（TCP）、用户数据报协议（UDP）等，TCP 和 UDP 给数据包加入传输数据并把它传输到下一层中，这一层负责传送数据，并且确定数据已被送达并接收。

互连网络层：负责提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机（但不检查是否被正确接收），如网际协议（IP）。

网络接口层：对实际的网络媒体的管理，定义如何使用实际网络（如 Ethernet、SerialLine 等）来传送数据。

二、Java 中的设计模式

1、你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：


创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2、单例设计模式

最好理解的一种设计模式，分为懒汉式和饿汉式。

 饿汉式：

```
public class Singleton {  
    //直接创建对象  
    public static Singleton instance = new Singleton();  
  
    //私有化构造函数  
    private Singleton() {  
    }  
  
    //返回对象实例  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

 懒汉式：

```
public class Singleton {  
    //声明变量  
    private static volatile Singleton singleton2 = null;  
  
    //私有构造函数  
    private Singleton2() {  
    }  
  
    //提供对外方法  
    public static Singleton2 getInstance() {  
        if (singleton2 == null) {  
            synchronized (Singleton2.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
}
```

3、工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。

工厂方法模式

工厂方法模式分为三种：普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

普通工厂模式

```
public interface Sender {
    public void Send();
}
public class MailSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is mail sender!");
    }
}
public class SmsSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
public class SendFactory {
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}
```

◆ 多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
public class SendFactory {
    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}

public class FactoryTest {
    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.send();
    }
}
```

◆ 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {
    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}

public class FactoryTest {
    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.send();
    }
}
```

◆ 抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

```
public interface Provider {
    public Sender produce();
}

-----

public interface Sender {
    public void send();
}

-----

public class MailSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is mail sender!");
    }
}

-----

public class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is sms sender!");
    }
}

-----

public class SendSmsFactory implements Provider {

    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
```



```
public class SendMailFactory implements Provider {

    @Override
    public Sender produce() {
        return new MailSender();
    }
}

-----

public class Test
{
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.send();
    }
}
```

4、建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 **Test** 结合起来得到的。

```
public class Builder {  
    private List<Sender> list = new ArrayList<Sender>();  
  
    public void produceMailSender(int count) {  
        for (int i = 0; i < count; i++) list.add(new MailSender());  
    }  
  
    public void produceSmsSender(int count) {  
        for (int i = 0; i < count; i++) list.add(new SmsSender());  
    }  
}
```

```
public class TestBuilder {  
    public static void main(String[] args) {  
        Builder builder = new Builder();  
        builder.produceMailSender(10);  
    }  
}
```

5、适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。



类的适配器模式

```
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}

-----

public interface Targetable {
    /*与原类中的方法相同 */
    public void method1();
    /*新类的方法 */
    public void method2();
}

public class Adapter extends Source implements Targetable {
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}

public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```



对象的适配器模式

基本思路和类的适配器模式相同，只是将 **Adapter** 类作修改，这次不继承 **Source** 类，而是持有 **Source** 类的实例，以达到解决兼容性的问题。

```
public class Wrapper implements Targetable {
    private Source source;

    public Wrapper(Source source) {
        super();
        this.source = source;
    }

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}

-----

public class AdapterTest {

    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

6、装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

```
public interface Sourceable {
    public void method();
}

-----

public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}

-----

public class Decorator implements Sourceable {
    private Sourceable source;
    public Decorator(Sourceable source) {
        super();
        this.source = source;
    }

    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}

-----

public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

7、策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

```
public interface ICalculator {
    public int calculate(String exp);
}

-----

public class Minus extends AbstractCalculator implements ICalculator

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "-");
        return arrayInt[0] - arrayInt[1];
    }
}

-----

public class Plus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "\\+");
        return arrayInt[0] + arrayInt[1];
    }
}

-----

public class AbstractCalculator {
    public int[] split(String exp, String opt) {
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}

-----

public class StrategyTest {
    public static void main(String[] args) {
        String exp = "2+8";
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}
```

8、观察者模式 (Observer)

观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

```
public interface Observer {
    public void update();
}

public class Observer1 implements Observer {
    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}

public class Observer2 implements Observer {
    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }
}

public interface Subject {
    /*增加观察者*/
    public void add(Observer observer);

    /*删除观察者*/
    public void del(Observer observer);
}
```



```
        /*通知所有的观察者*/
        public void notifyObservers();

        /*自身的操作*/
        public void operation();
    }

    public abstract class AbstractSubject implements Subject {

        private Vector<Observer> vector = new Vector<Observer>();

        @Override
        public void add(Observer observer) {
            vector.add(observer);
        }

        @Override
        public void del(Observer observer) {
            vector.remove(observer);
        }

        @Override
        public void notifyObservers() {
            Enumeration<Observer> enumo = vector.elements();
            while (enumo.hasMoreElements()) {
                enumo.nextElement().update();
            }
        }
    }

    public class MySubject extends AbstractSubject {

        @Override
        public void operation() {
            System.out.println("update self!");
            notifyObservers();
        }
    }

    public class ObserverTest {
        public static void main(String[] args) {
            Subject sub = new MySubject();
            sub.add(new Observer1());
        }
    }
}
```

```
sub.add(new Observer2());  
sub.operation();  
}  
}
```