# MVP in Android



Model： 封装的各种数据的实体类，以及提供对数据操作的接口，比如本地查询数据或者网上加载数据等操作。完全独立，只涉及数据层，不依赖。

View：xml布局文件以及Activity或Fragment

Xml布局文件编写静态ui界面，Activity或Fragment提供可交互的ui操作，比如弹窗或点击事件等。

Activity或Fragment里通常拥有Presenter的引用，各种业务操作可先定义为接口，由Presenter来实现，然后通过Presenter的引用来调用。

Presenter：业务层操作

通常拥有Activity或Fragment的引用，也拥有Model层的引用，先通过Model层的引用获取数据后再通过Activity的引用来通知Activity该做出的反应。

**代码示例**

# RxJava

抛物线： <http://gank.io/post/560e15be2dca930e00da1083>

Github： <https://github.com/ReactiveX/RxJava>    
<https://github.com/ReactiveX/RxAndroid>

RxJava是什么？------异步，一个在JAVA VM上使用可观测的序列来组成异步、基于事件的程序的库

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。

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()。



使用方式：（被观察者.订阅（观察者），虽然与现实相反，但这样支持链式调用）

必须包括三步：

1. 创建被观察者
2. 创建观察者
3. 关联观察者和被观察者的联系

代码示例：



# 观察者模式（发布/订阅模式）

**概念**

现实概念上的观察者模式：A对象（观察者）需要时时关注B对象（被观察者）的某种状态是否发生改变，在发生变化的一瞬间做出反应。

程序里的观察者模式：B对象（被观察者）的某种状态发生变化时，需要主动通知A对象（观察者），让A对象（观察者）做出反应。所以也称作发布/订阅模式，A对象（观察者）订阅B对象（被观察者）的某一状态，当B对象的被订阅的状态发生变化时，就发布通知，告知所有订阅这一状态的观察者需要做出相应的反应了。

两者的区别有点类似于《计算机组成原理》里的程序查询方式和中断方式，一种是主动查询，一种是被动等待通知。

**Android源码里的例子**

OnClickListener()

-按钮的点击事件

Button-----被观察者 onClickListener-----观察者（观察Button的onClick()状态）

两者通过Button.setOnClickListener(onClickListener)来建立订阅关系



**代码示例**

# VerticalGridView

# PropertyValuesHolder

用途：可以用来实现view的抖动

使用方法：

第一步：生成Keyframe对象；  
第二步：利用PropertyValuesHolder.ofKeyframe()生成PropertyValuesHolder对象  
第三步：ObjectAnimator.ofPropertyValuesHolder()生成对应的Animator

示例代码：

Keyframe frame0 = Keyframe.ofFloat(0f, 0);

Keyframe frame1 = Keyframe.ofFloat(0.1f, -20f);

Keyframe frame2 = Keyframe.ofFloat(1, 0);

PropertyValuesHolder frameHolder = PropertyValuesHolder.ofKeyframe("rotation",frame0,frame1,frame2);

 Animator animator = ObjectAnimator.ofPropertyValuesHolder(mImage,frameHolder);

animator.setDuration(1000);

animator.start();

Keyframe.ofFloat(0, 0)表示动画进度为0时，动画所在的数值位置为0；Keyframe.ofFloat(0.25f, -20f)表示动画进度为25%时，动画所在的数值位置为-20；Keyframe.ofFloat(1f,0)表示动画结束时，动画所在的数值位置为0；

# ViewStub

在开发应用程序的时候，经常会遇到这样的情况，会在运行时动态根据条件来决定显示哪个View或某个布局。那么最通常的想法就是把可能用到的View都写在上面，先把它们的可见性都设为View.GONE，然后在代码中动态的更改它的可见性。这样的做法的优点是逻辑简单而且控制起来比较灵活。但是它的缺点就是，耗费资源。虽然把View的初始可见View.GONE但是在Inflate布局的时候View仍然会被Inflate，也就是说仍然会创建对象，会被实例化，会被设置属性。也就是说，会耗费内存等资源。

      推荐的做法是使用***[android](http://lib.csdn.net/base/15" \t "_blank" \o "Android知识库)***.view.ViewStub，ViewStub 是一个轻量级的View，它一个看不见的，不占布局位置，占用资源非常小的控件。可以为ViewStub指定一个布局，在Inflate布局的时候，只有 ViewStub会被初始化，然后当ViewStub被设置为可见的时候，或是调用了ViewStub.inflate()的时候，ViewStub所向 的布局就会被Inflate和实例化，然后ViewStub的布局属性都会传给它所指向的布局。这样，就可以使用ViewStub来方便的在运行时，要还 是不要显示某个布局。

      但ViewStub也不是万能的，下面总结下ViewStub能做的事儿和什么时候该用ViewStub，什么时候该用可见性的控制。

     首先来说说ViewStub的一些特点：

         1. ViewStub只能Inflate一次，之后ViewStub对象会被置为空。按句话说，某个被ViewStub指定的布局被Inflate后，就不会够再通过ViewStub来控制它了。

         2. ViewStub只能用来Inflate一个布局文件，而不是某个具体的View，当然也可以把View写在某个布局文件中。

     基于以上的特点，那么可以考虑使用ViewStub的情况有：

         1. 在程序的运行期间，某个布局在Inflate后，就不会有变化，除非重新启动。

              因为ViewStub只能Inflate一次，之后会被置空，所以无法指望后面接着使用ViewStub来控制布局。所以当需要在运行时不止一次的显示和 隐藏某个布局，那么ViewStub是做不到的。这时就只能使用View的可见性来控制了。

         2. 想要控制显示与隐藏的是一个布局文件，而非某个View。

              因为设置给ViewStub的只能是某个布局文件的Id，所以无法让它来控制某个View。

# android:drawableRight

**TextView的drawable属性**

TextView有一些属性可以在Text的四周设置一个drawable对象，图片,shape等合法的drawable都可以用。

* drawableStart API 14才有
* drawableLeft
* drawableTop
* drawableBottom
* drawableRight
* drawableEnd API 14才有
* drawablePadding 用以设置drawable与text之间的空间

**TextView的drawable的一些注意事项**

1. . TextView的padding作用在drawable之外
2. . TextView的高度或宽度为wrap\_content时将是文字和drawable中较大的那一个，再加上padding和margin
3. . gravity只对文字起作用，对drawable不起作用
4. . drawable会在其所在的维度居中显示，比如drawableLeft是上下垂直居中的，以此类推

**局限性**

有些时候它也有一些局限性而没有办法用它： 1. 当drawable本身没有高度时（比如shape），这个drawable高度就会依赖于文字，因为padding是加在drawable之外，所以只会依赖于文字的高度。有些时候这不是想要的结果。 2. 当Icon需要与文字分开单独控制时，很显示这要分成二个View。 3. 当需要对Icon进行特殊的个性化时，比如添加背景，特效等。 4. 其他一些造成无法使用的。 5. 除上述情况外，就要考虑使用drawable了。

# android:nextFocusRight

控制下一个焦点

# TabHost、TabWidget

# android:descendantFocusability——父容器和子控件的焦点获取问题

通过android:descendantFocusability这个属性可以解决上面的问题。这个属性定义了当一个焦点要传递给父容器或者子控件时，父容器和子控件之间获得焦点的关系。具体值如下：

beforeDescendants：父容器会比其子控件率先获得焦点。

afterDescendants：如果没有任何子控件要获得焦点的话，那么父容器才会获得焦点。

blocksDescendants：父容器会阻止其子控件获得焦点（也就是说焦点会由父容器获得）。

# android:duplicateParentState

指的是当前控件是否跟随父控件的(点击、焦点等)状态

# textview.setMovementMethod(ScrollingMovementMethod.getInstance());

# findFocus()、hasFocus()、isFocused()、getFocusChild()

View的焦点状态

# 删除系统应用

1、 mount -o rw,remount /system 卸载系统应用时先运行这句

2、 后把 /system/app 和/data/data 下的相关文件删掉

3、 reboot重启盒子

4、 安装debug应用

添加一下、system目录的权限，就能删了

# FileLock、FileChannel多线程读写文件

解决对同一文件同时进行读写操作时的逻辑错误

# Loaders、LoadManager

子线程加载数据、源数据发送变化时回调

# TextView测量文字宽度

TextPaint textPaint = textView.getPaint();

float textPaintWidth = textPaint.measureText(text);

# TextView源码学习

getCompoundPadding\*\*\*\*

getCompoundDrawablePadding()

getExtendedPaddingBottom/Top

getMinWidth/Height

getTotalPadding\*\*\*\*

·Drawable区域可以放置在TextView的Left, Top, Right, Bottom四个方向

·如果没有Drawable区域的话，CompoundPadding\*\*\* = Padding\*\*\* （\*\*\*表示left等值），如果有Drawable区域的话CompoundPadding\*\*\* = Padding\*\*\* + Drawable长/宽 + DrawablePadding

·如果没有设置maxLines属性，或文字全部显示出来时:

ExtendedPaddingTop/Bottom = CompoundPaddingTop/Bottom

·如果有设置maxLines属性且文字有部分未显示出来时：

ExtendedPaddingTop/Bottom = TotalPaddingTop/Bottom(应该是)

·TotalPaddingLeft/Right = CompounPaddingLeft/Right

·TotalPaddingTop/Bottom = 文字的首尾行分别到TextView上下端边界的距离

DrawablePadding

CompoundPaddingLeft

PaddingLeft

TextView

TotalPaddingBottom

PaddingTop

TotalPaddingTop

PaddingRight

PaddingBottom

**显示出来的文字占据的区域**

**文字区域**

**Drawable区域**

# SharePreference

registerOnSharedPreferenceChangeListener(listener)  
这个方法才是我要说的重点，因为之前有些需求就是更改了SharedPreferences之后，要通知相应的组件做出改变，我以前的处理方式是通过事件订阅实现的，发一个event出去，然后目标收到event再做出反应，当时觉得特别蛋疼，两边都要做些操作，显的特别啰嗦，当时就在想可不可以在SharedPreferences上设置一个观察者，一旦有什么风吹草动，就自动通知目标，不曾想，人家早已经实现了，只是我愚昧无知，今天去看了下源码发现了这个方法，相见恨晚。



# Dp和px

* dip        ： Density independent pixels ，设备无关像素。
* dp        ：就是dip
* px        ： 像素
* **dpi**：dots per inch ， 直接来说就是一英寸多少个像素点。常见取值 120，160，240。我一般称作像素密度，简称密度
* **density**： 直接翻译的话貌似叫 密度。常见取值 1.5 ， 1.0 。和标准dpi的比例（160px/inc）
* 分辨率   ： 横纵2个方向的像素点的数量，常见取值 480X800 ，320X480
* 屏幕尺寸： 屏幕对角线的长度。电脑电视同理。
* 屏幕比例的问题。因为只确定了对角线长，2边长度还不一定。所以有了4：3、16：9这种，这样就可以算出屏幕边长了。

换算公式如下：

dp = density px

PX = density \* DP

代码



# Gradle

**概念：**

projects 和 tasks是 Gradle 中最重要的两个概念。

任何一个 Gradle 构建都是由一个或多个 projects 组成。

每个 project 都由多个 tasks 组成。每个 task 都代表了构建执行过程中的一个原子性操作。如编译，打包，生成 javadoc，发布到某个仓库等操作。

**在两个任务之间指明依赖关系 dependsOn**

task hello << {

println 'Hello world!'

}

task intro(dependsOn: hello) << {

println "I'm Gradle"

}

//或者

task0.dependsOn task2, task3

**为任务增加自定义属性**

task myTask {

ext.myProperty = "myValue"

}

task printTaskProperties << {

println myTask.myProperty

}

**依赖任务的不同输出**

task distribution << {

println "We build the zip with version=$version"

}

task release(dependsOn: 'distribution') << {

println 'We release now'

}

gradle.taskGraph.whenReady {taskGraph ->

if (taskGraph.hasTask(release)) {

version = '1.0'

} else {

version = '1.0-SNAPSHOT'

}

}

gradle -q distribution 的输出结果

Output of gradle -q distribution

\> gradle -q distribution

We build the zip with version=1.0-SNAPSHOT

gradle -q release 的输出结果

Output of gradle -q release

\> gradle -q release

We build the zip with version=1.0

We release now

whenReady 会在已发布的任务之前影响到已发布任务的执行。即使已发布的任务不是主要任务(也就是说，即使这个任务不是通过命令行直接调用)



<http://stormzhang.com/android/2016/03/13/gradle-config/>

# Groovy

Groovy是一种基于JVM的动态语言，说简单点，就是可以再Java虚拟机上运行的脚步语音。它和Java类似（算是Java的升级版，但是又具备脚本语言的特点），都在Java虚拟机中运行。当运行Groovy脚本时它会先被编译成Java类字节码，然后通过JVM虚拟机执行这个Java字节码类。

Task是Groovy的核心所在，Task是完成Groovy任务的最小执行单元。添加一个Task很简单，指定 task [task name]即可。

task hello {

println ‘hello world’

}

Hello.enable = false 关闭或开启一个task

复制文件task

Task testCopy(type: Copy){

From ‘src/man/res/layout’

Into ‘src/man/new

}

一个task编译时打出UP-TO-DATE表示该Task之前已经编译过了，且没有任何修改，所以跳过该task的执行，也就是Gradle的增量编译。

字符串的定义：单引号，双引号，斜杠，三引号

> apply plugin: 'com.android.application'

常见的还有另一个就是 apply plugin: 'library'，作用也就是设置当前model是主程序还是库。

作用清楚了，下面来理解一下，为什么要这样设置。gradle是基于groovy语言的，

参考文献：

在AS中自定义Gradle插件 [http://www.jianshu.com/p/d53399cd507b#](http://www.jianshu.com/p/d53399cd507b)

构建神器 <http://jiajixin.cn/2015/08/07/gradle-android/>

Gradle tip <http://blog.csdn.net/lzyzsd/article/category/2795779>

Groovy探索GString <http://blog.csdn.net/hivon/article/details/2271000>

Diff with Java <http://www.groovy-lang.org/differences.html>

Gradle for Android <https://segmentfault.com/a/1190000004229002>

Gradle Plugin User Guide <http://tools.android.com/tech-docs/new-build-system/user-guide>

Groovy语言规范-语法 <http://ifeve.com/groovy-syntax/>

学习Gradle：

1.了解groovy基本语法

2.粗度 Gradle User Guide 和 Gradle Plugin User Guide

<https://docs.gradle.org/current/userguide/userguide.html>

<http://tools.android.com/tech-docs/new-build-system/user-guide>

3.实战

Groovy学习：

官方教程 <http://www.groovy-lang.org/differences.html>

IMB 精通Groovy <http://www.ibm.com/developerworks/cn/education/java/j-groovy/j-groovy.html>

//获取所有的task, 过滤掉某些task的执行

tasks.whenTaskAdded{task ->

if (task.name.contains('AndroidTest')) {

task.enabled = false;

}

}

gradle执行分三个阶段，初始化阶段（setting.gradle)，配置阶段(build.gradle生成Project和Gradle对象）

执行阶段(执行各个task)。

//afterEvaluate 会在gradle的配置阶段结束时执行

aftertEvaluate {

android.applicationVariants.each { variant ->

def dx = tasks.findByName("dex$variant.name.capitalize()}")

def hello = "hello$variant.name.capitalize()}"

task(hello) << {

println "hello"

}

tasks.findByName(hello).dependsOn dx.taskDependencies.getDependencies(dx)

dx.dependsOn tasks.findByName(hello)

}

}

**Groovy里的方法**

Groovy方法的括号可以省略





Groovy支持闭包作为方法参数，如上图第三行代码

Groovy规定，如果方法的最后一个参数是闭包，那么闭包可以放在括号外面即方法后面，如上图第7行

Groovy支持省略掉方法的括号，因此最终如上图第.2行代码



这在build.gradle里很常见，其实都是一个个的方法，参数为闭包，都省略掉方法的括号，所以最终写出来都如是 方法名{} 格式。

**属性和get,set方法**





# build.gradle

查看android插件的各个标签作用

<http://google.github.io/android-gradle-dsl/current/index.html>

**读取local.properties里的变量时**

Properties properties = new Properties();  
properties.load(project.rootProject.file('local.properties').newDataInputStream())  
println properties.getProperty('KEYPASS')

也可以 properties.load(new FileInputStream(project.rootProject.file(‘local.properties’)

应用于一些敏感数据时，可以用上述方法，比如签名配置，各种app key

**多渠道打包配置**

1.配置AndroidManifest.xml  
以友盟渠道为例，渠道信息一般都是写在 AndroidManifest.xml文件中：

<meta-data android:name="UMENG\_CHANNEL" android:value="xiaomi" />

如果不使用多渠道打包方法，那就需要我们手动一个一个去修改value中的值，xiaomi，360，qq，wandoujia等等。使用多渠道打包的方式，就需要把上面的value配置成下面的方式：

<meta-data android:name="UMENG\_CHANNEL" android:value="${UMENG\_CHANNEL\_VALUE}" />

其中${UMENG\_CHANNEL\_VALUE}中的值就是你在gradle中自定义配置的值。

2.在build.gradle设置productFlavors  
写法如下：

productFlavors {

wandoujia {

manifestPlaceholders = [UMENG\_CHANNEL\_VALUE: "wandoujia"]

}

xiaomi {

manifestPlaceholders = [UMENG\_CHANNEL\_VALUE: "xiaomi"]

}

qq {

manifestPlaceholders = [UMENG\_CHANNEL\_VALUE: "qq"]

}

360 {

manifestPlaceholders = [UMENG\_CHANNEL\_VALUE: "360"]

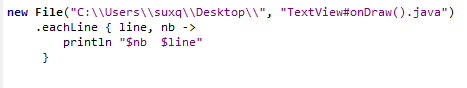
}

}

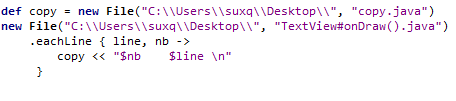
其中[UMENG\_CHANNEL\_VALUE: "wandoujia"]就是对应  
${UMENG\_CHANNEL\_VALUE}的值。

# Groovy 文件操作

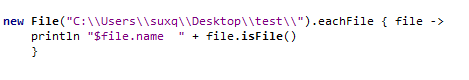
读文件：



写文件



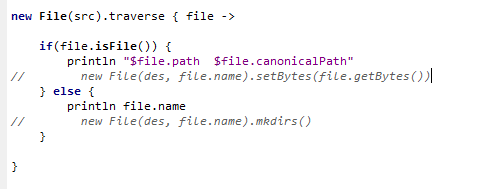
遍历文件夹，包括文件和子文件夹



创建文件夹



遍历文件夹



# Chrome插件

<http://stormzhang.com/devtools/2016/01/15/google-chrome-extension/>

Su