2019 一线互联网 Android 面试题解析大全

1. 自定义 View

1 良好的自定义 View

易用，标准，开放。

一个设计良好的自定义 view 和其他设计良好的类很像。封装了某个具有易用性接口的功能组合，这些功能能够有效地使用 CPU 和内存，并且十分开放的。但是，除了开始一个设计良好的类之外，一个自定义 view 应该：

l 符合安卓标准

l 提供能够在 Android XML 布局中工作的自定义样式属性

l 发送可访问的事件

l 与多个 Android 平台兼容。

Android 框架提供了一套基本的类和 XML 标签来帮您创建一个新的，满足这些要求的 view。忘记提供属性和事件是很容易的，尤其是当您是这个自定义 view 的唯一用户时。请花一些时间来仔细的定义您 view 的接口以减少未来维护时所耗费的时间。一个应该遵从的准则是：暴露您 view 中所有影响可见外观的属性或者行为。

2 创建自定义 View (步骤)

2.1 继承 View 完全自定义或继承 View 的派生子类

必须提供一个能够获取 Context 和作为属性的 AttributeSet 对象的构造函数，获取属性，当 view 从 XML 布局中创建了之后，XML 标签中所有的属性都从资源包中读取出来并作为一个 AttributeSet 传递给 view 的构造函数。

View 派生出来的直接或间接子类：ImageView, Button, CheckBox, SurfaceView, TextView, ViewGroup, AbsListView

ViewGourp 派生出来的直接或间接子类：AbsoluteLayout, FrameLayout, RelativeLayout, LinearLayout

所有基类、派生类都是 Android framework 层集成的标准系统类，可直接引用 SDK 中这些系统类及其 API

2.2 定义自定义属性

l 在资源元素<declare-styleable>中为您的 view 定义自定义属性。

在项目组添加<declare-styleable>资源。这些资源通常是放在 res/values/attrs.xm 文件里。如下是 attrs.xml 文件的一个例子：

```
<resources>;
   <declare-styleable name="PieChart">
      <attr name="showText" format="boolean" />
      <attr name="labelPosition" format="enum">
         <enum name="left" value="0"/>
         <enum name="right" value="1"/>
      </attr>
   </declare-styleable>
</resources>
```

l 在您的 XML 布局中使用指定属性的值。

布局 XML 文件中可以像内建属性一样使用它们。唯一不同是自定义属性属于不同的命名空间。

http://schemas.android.com/apk/res/[你的自定义 View 所在的包路径]

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3  xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
4  <com.example.customviews.charting.PieChart
5  custom:showText="true"
6  custom:labelPosition="left" />
7  </LinearLayout>
```

l 在运行时取得属性值。

l 将取回的属性值应用到您的 view 中。

2.3 获取自定义属性

当 view 从 XML 布局中创建了之后，XML 标签中所有的属性都从资源包中读取出来并作为一个 AttributeSet 传递给 view 的构造函数。尽管从 AttributeSet 中直接读取值是可以的，但是这样做有一些缺点：

l 带有值的资源引用没有进行处理

l 样式并没有得到允许

取而代之的是，将 AttributeSet 传递给 obtainStyledAttributes()方法。这个方法传回了一个 TypedArray 数组，包含了已经解除引用和样式化的值。

为了时能能够更容易的调用 obtainStyledAttributes()方法，Android 资源编译器做了大量的工作。res 文件夹 中的每个<declare-styleable>资源，生成的 R.java 都定义了一个属性 ID 的数组以及一套定义了指向数组中的每一个属性 的常量。您可以使用预定义的常量从 TypedArry 中读取属性。下例是 PieChart 类是如何读取这些属性的：

```
1
2   public PieChart(Context ctx, AttributeSet attrs) {
3   super(ctx, attrs);
4   TypedArray a = context.getTheme().obtainStyledAttributes(
5   attrs,
6   R.styleable.PieChart,
7   0, 0);
8   try {
9   mShowText = a.getBoolean(R.styleable.PieChart_showText, false);
10  mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);
11  } finally {
12  a.recycle();
13  }
14  }
```

注意,TypedArry 对象是一个共享的资源，使用完毕必须回收它。

2.4 添加属性和事件

属性是控制 view 的行为和外观的强有力的方式，但是只有 view 在初始化后这些属性才可读。为了提供动态的行为，需要暴露每个自定义属性的一对 getter 和 setter。下面的代码片段显示 PieChart 是如何提供 showText 属性的。

```
1
2   public boolean isShowText() {
3   return mShowText;
4   }
5   public void setShowText(boolean showText) {
6   mShowText = showText;
7   invalidate();
8   requestLayout();
9   }
```

注意，setShowText 调用了 invalidate()和 requestLayout()。这些调用关键是为了保证 view 行为是可靠的。你必须在改变这个可能改变外观的属性后废除这个 view，这样系统才知道需要重绘。同样，如果属性的变化可能影响尺寸或者 view 的形状，您需要请求一个新的布局。忘记调用这些方法可能导致难以寻找的 bug。

自定义 view 同样需要支持和重要事件交流的事件监听器。

2.5 自定义绘制（实施）

绘制自定义视图里最重要的一步是重写 onDraw()方法. onDraw()的参数是视图可以用来绘制自己的 Canvas 对象. Canvas 定义用来绘制文本、线条、位图和其他图像单元. 你可以在 onDraw()里使用这些方法创建你的自定义用户界面(UI).

android.graphics 框架把绘图分成了两部分:

l 画什么, 由 Canvas 处理

l 怎么画, 由 Paint 处理

例如, Canvas 提供画线条的方法，而 Paint 提供定义线条颜色的方法. Canvas 提供画矩形的方法，而 Paint 定义是否用颜色填充矩形或让它为空. 简而言之, Canvas 定义你可以在屏幕上画的形状，而 Paint 为你画的每个形状定义颜色、样式、字体等等.

onDraw()不提供 3d 图形 api 的支持。如果你需要 3d 图形支持 ,必须继承 SurfaceView 而不是 View，并且通过单独的线程画图。

3 优化

3.1 降低刷新频率

为了提高 view 的运行速度，减少来自于频繁调用的程序的不必要的代码。从 onDraw()方法开始调用，这会给你带来最好的回报。特别地，在 onDraw()方法中你应该减少冗余代码，冗余代码会带来使你 view 不连贯的垃圾回收。初始化的冗余对象，或者动画之间的，在动画运行时，永远都不会有所贡献。

加之为了使 onDraw()方法更有依赖性，你应该尽可能的不要频繁的调用它。大部分时候调用 onDraw()方法就是调用 invalidate()的结果，所以减少不必要的调用 invalidate()方法。有可能的，调用四种参数不同类型的 invalidate()，而不是调用无参的版本。无参变量需要刷新整个 view ,而四种参数类型的变量只需刷新指定部分的 view. 这种高效的调用更加接近需求，也能减少落在矩形屏幕外的不必 要刷新的页面。

3.2 使用硬件加速

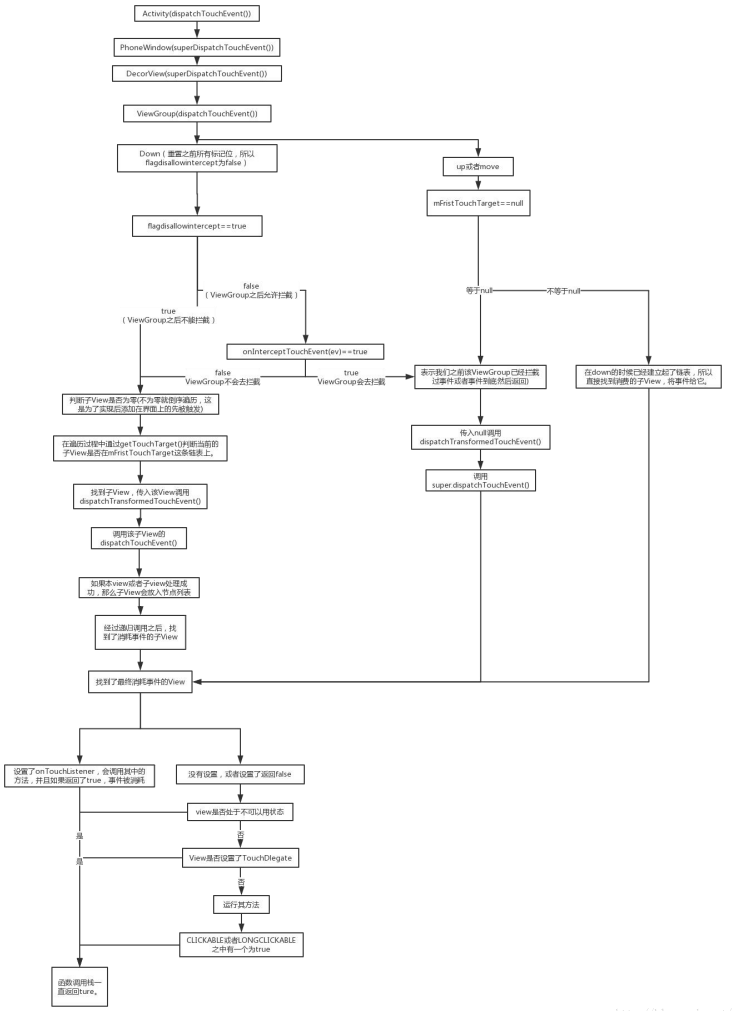
作为 Android3.0，Android2D 图表系统可以通过大部分新的 Android 装置自带 GPU（图表处理单元）来增加，对于许多应用程序 来说，GPU 硬件加速度能带来巨大的性

能增加，但是对于每一个应用来讲，并不都是正确的选择。Android 框架层更好地为你提供了控制应用程序部分硬件 是否增加的能力。

怎样在你的应用，活动，或者窗体级别中使用加速度类，请查阅 Android 开发者指南中的 Hardware Acceleration 类。注意到在开发者指南中的附加说明，你必须在你的 AndroidManifest.xml 文件中的<uses-sdk android:targetSdkVersion="11"/>中将应用目标 API 设置到 11 或者更高的级别。

一旦你使用硬件加速度类 ,你可能没有看到性能的增长 ,手机 GPUs 非常擅长某些任务，例如测量，翻转，和平移位图类的图片。特别地，他们不擅长其他的任务，例如画直线和曲线。为了利用 GPU 加速度类，你应该增加 GPU 擅长的操作数量，和减少 GPU 不擅长的操作数量。

1. 事件拦截分发



2. 解决过的一些性能问题，在项目中的实际运用

3. 性能优化工具

1. 性能优化分析工具学习

在开始代码优化之前，先得学会使用性能分析工具。以下三个工具都是谷歌官方推出的，可以帮助我们定位分析问题，从而优化我们的 APP。

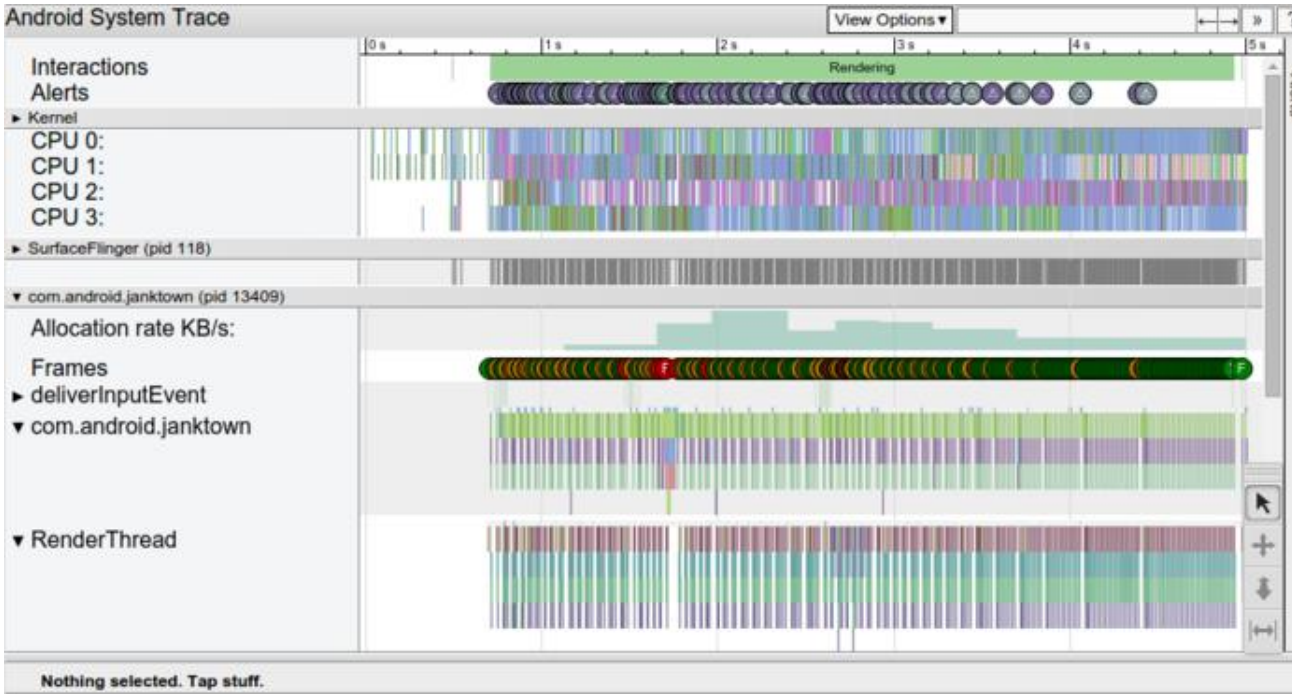
- **System Trace**

Systrace 是一个收集和检测时间信息的工具，它能显示 CPU 和时间被消耗在哪儿了，每个进程和线程都在其 CPU 时间片内做了

什么事儿. 而且会指示哪个地方出了问题，以及给出 Fix 建议。给出的结果 trace 文件是以 html 形式打开的，直接用浏览器打开

查看十分方便。打开方法：打开 DDMS 后，连接手机，点击手机上方一排按钮中的 SysTrace 按钮。

打开的效果如下图：



在代码中打点方式如下

```
Trace.beginSection("name");
//要检测运行时间的代码
Trace.endSection();
```

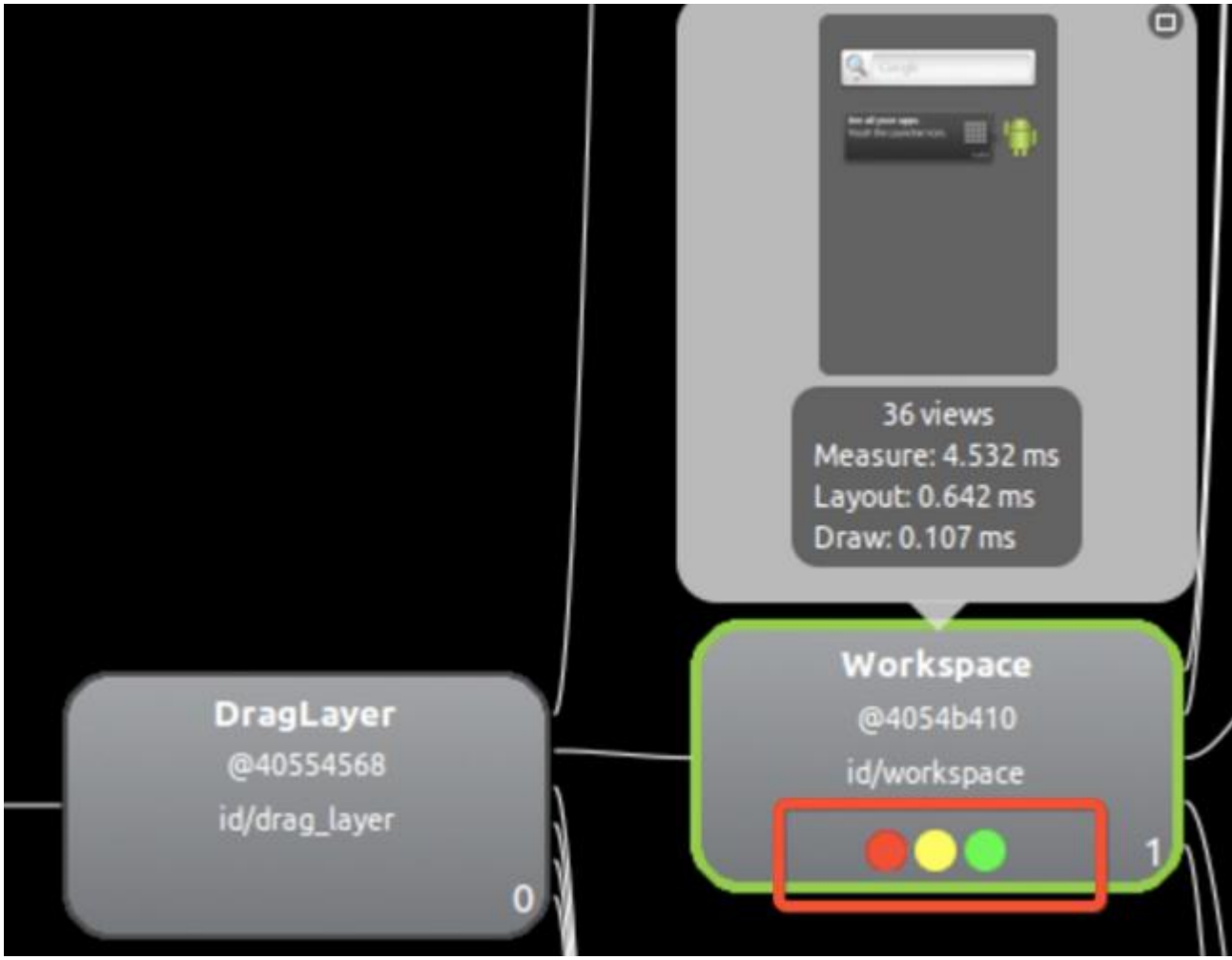
- **Hierarchy Viewer**

Hierarchy Viewer 提供了一个可视化的界面来观测布局的层级，让我们可以优化布局层级，删除多余的不必要的 View

层级，提升布局速度。另外，开发者模式中调试 GPU 过度绘制选项也可以进行视图层级调试。在 SDK-> tools 目录下

打开 hierarchyviewer.bat 即可。

效果如下图：



- **TraceView**

一个图形化的工具，用来展示和分析方法的执行时间。也是一款性能优化的神器。可以通过像打 log 一样的方式去定位代码的执行时

间，从而可以准确定位是哪一段代码的执行消耗了太多时间。相比 SysTrace，功能更强大，使用起来也更复杂。

2. 布局优化

布局优化相对比较容易，优化可以先从布局来展开。使用 Hierarchy Viewer 和开发者模式中关于布局绘制的选项，可以查到一些问题然后进行修改。

- **布局嵌套过深** 有的时候为了赶进度，布局设计的不是很好。层级嵌套过深的话，深度遍历各个节点会非常消耗时间，这也是布局优化余地最大的一个点了。很多过深的层级是不必要的。如果布局真的很复杂，不深度嵌套没法实现想要的效果。试试最新的约束布局 Constraintlayout 吧。没有使用过的话，下面这篇官方文档可以帮助你：Constraintlayout 官方介绍文档

  使用合适的布局 三种常见的 ViewGroup 的绘制速度：FrameLayout > LinerLayout > RelativeLayout。当然，如果用 RelativeLayout 可以避免布局嵌套的话是值得的。可以根据这些去决定选用什么样的布局。

  列表控件优化 不论是 ListView 还是 RecycleView 都有优化点，一个是 convertView 的复用，一个是 ViewHolder 的使用避免重复遍历节点。当然这些都是基础中的基础了。如果发现项目中的代码 ListView 或者 RecycleView 的使用不规范的话，赶紧进行修改吧。

  使用 include 标签 在布局文件中，<include>标签可以指定插入一段布局文件到当前布局。这样的话既提高了布局复用，也减少了我们的代码书写。另外，<merge>标签可以和<include>的标签一起使用从而减少布局层级。

  ViewStub 延时加载 有些布局，比如网络出错的布局，没必要在所有时候都加载出来。使用 ViewStub 可以实现按需加载。ViewStub 本身没有宽高，加载起来几乎不

消耗什么资源。当对他 setVisibility(View.VISIBLE)的时候会调用它引用的真实布局填充到当前位置，从而实现了延时加载，节省了正常加载的时间。

移除 Activity 默认背景 只要我们不需要 Activity 的默认背景，就可以移除掉，以减少 Activity 启动时的渲染时间，提升启动效率。移动方法见下：

```
<style name="MyStyle" parent="AppTheme">
        <item name="android:windowNoTitle">true</item>
        <item name="android:windowBackground">@null</item>
</style>
```

3. 线程优化

线程的创建和销毁会带来比较大的性能开销。因此线程优化也很有必要。查看项目中是否存在随意 new thread，线程缺乏管理的情况。使用 AsyncTask 或者线程池对线程进行管理，可以提升 APP 的性能。另外，我比较推荐使用 Rxjava 来实现异步操作，既方便又优雅。

推荐一篇 Rxjava 的入门文章

4. 内存泄露

内存泄露会导致 APP 占用内存过高，影响效率，严重的话会导致 OOM。因此如果项目存在内存泄露的话要优先解决。查找内存泄露可以用 LeakCanary 等工具，具体怎么解决，有哪些泄露点，以后有时间也写篇总结。

5. 算法优化

毋庸置疑，使用合适的算法处理事务可以大幅提升 APP 的性能。当然算法不是我的强项，也只能给出一些大致的点：查询考虑二分查找节省时间，尽量不要使用耗时的递归算法。必要的时候可以空间换时间来提高 APP 运行效率。

6. 其他优化点

异步处理耗时任务 在 Activity、Fragemnt 的 onCreate 等初始化方法中，如果执行了太耗时的操作（例如读取各种数据），会影响页面的加载速度，让用户觉得 APP 太慢。这时候可以异步处理这些耗时任务，减小应用启动的时候的负担。

替换矢量图 尽管矢量图有诸多优点，但矢量图的绘制是消耗性能的。在应用初始化加载等比较影响用户体验的地方，还是建议使用 Bitmap 来代替矢量图，提高 APP 开启效率。

正则表达式 经小伙伴用 TraceView 不断的打点发现，正则表达式非常消耗时间。因此尽管正则表达式非常优雅，涉及到性能问题的时候，可以改为其他判断方式来提高 APP 性能。

浮点类型 在 Java 中浮点类型的运算大概比整型数据慢两倍，因此整型数据能解决的问题尽量用整型。

减少冗余 log 开发的时候用于调试的 log，在项目上线的时候没用的要及时删除。当然有用的 log 还是要留下，以便以后分析问题。

删除无用资源 没用用的资源会增大 APK 大小，既然没有用了，上线的时候当然要及时删除。

Lint 代码检查 使用 Lint 等静态代码检查工具可以帮助我们发现很多隐藏的问题。Lint 检查出来的问题越少，说明代码越规范，越不容易出现各种问题，APP 性能自然也会提升。

滥用全局广播 全局广播也是十分消耗性能的一个点。对于应用内的通讯，使用接口回调，EventBus 等手段比起广播是更好地选择。动态注册广播的时候，也不要忘了广播的注销。

4. 性能优化 （讲讲你自己项目中做过的性能优化）

6. Http[s]请求慢的解决办法（DNS、携带数据、直接访问 IP）

7. 缓存自己如何实现（LRUCache 原理）

LruCache 中 Lru 算法的实现就是通过 LinkedHashMap 来实现的。LinkedHashMap 继承于 HashMap，它使用了一个双向链表来存储 Map 中的 Entry 顺序关系，对于 get、put、remove 等操作，LinkedHashMap 除了要做 HashMap 做的事情，还做些调整 Entry 顺序链表的工作。

LruCache 中将 LinkedHashMap 的顺序设置为 LRU 顺序来实现 LRU 缓存，每次调用 get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。

调用 put 插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

8. 图形图像相关：OpenGL ES 管线流程、EGL 的认识、Shader 相关

9. SurfaceView、TextureView、GLSurfaceView 区别及使用场景

10. 动画、差值器、估值器（Android 中的 View 动画和属性动画 - 简书、Android 动画 介绍与使用）

11. MVC、MVP、MVVM

- 1.mvc:数据、View、Activity，View 将操作反馈给 Activity，Activitiy 去获取数据，数据通过观察者模式刷新给 View。循环依赖
    - 1.Activity 重，很难单元测试
    - 2.View 和 Model 耦合严重
- 2.mvp:数据、View、Presenter，View 将操作给 Presenter，Presenter 去获取数据，数据获取好了返回给 Presenter，Presenter 去刷新 View。PV，PM 双向依赖
    - 1.接口爆炸
    - 2.Presenter 很重
- 3.mvvm:数据、View、ViewModel，View 将操作给 ViewModel，ViewModel 去获取数据，数据和界面绑定了，数据更新界面更新。
    - 1.viewModel 的业务逻辑可以单独拿来测试
    - 2.一个 view 对应一个 viewModel 业务逻辑可以分离，不会出现全能类
    - 3.数据和界面绑定了，不用写垃圾代码，但是复用起来不舒服

12. Handler、ThreadLocal、AsyncTask、IntentService 原理及应用

- 1.MessageQueue：读取会自动删除消息，单链表维护，在插入和删除上有优势。在其 next()中会无限循环，不断判断是否有消息，有就返回这条消息并移除。
- 2.Looper：Looper 创建的时候会创建一个 MessageQueue，调用 loop()方法的时候消息循环开始，loop()也是一个死循环，会不断调用 messageQueue 的 next()，当有消息就处理，否则阻塞在 messageQueue 的 next()中。当 Looper 的 quit()被调用的时候会调用 messageQueue 的 quit(),此时 next()会返回 null，然后 loop()方法也跟着退出。
- 3.Handler：在主线程构造一个 Handler，然后在其他线程调用 sendMessage(),此时主线程的 MessageQueue 中会插入一条 message，然后被 Looper 使用。
- 4.系统的主线程在 ActivityThread 的 main()为入口开启主线程，其中定义了内部类 Activity.H 定义了一系列消息类型，包含四大组件的启动停止。
- 5.MessageQueue 和 Looper 是一对一关系，Handler 和 Looper 是多对一

13. Gradle（Groovy 语法、Gradle 插件开发基础）

为何要学 **Groovy**

Gradle 是目前 Android 主流的构建工具，不管你是通过命令行还是通过 AndroidStudio 来 build，最终都是通过 Gradle 来实现的。所以学习 Gradle 非常重要。

目前国内对 Android 领域的探索已经越来越深，不少技术领域如插件化、热修复、构建系统等都对 Gradle 有迫切的需求，不懂 Gradle 将无法完成上述事情。所以 Gradle 必须要学习。

Gradle 不单单是一个配置脚本，它的背后是几门语言：

- Groovy Language
- Gradle DSL
- Android DSL

  DSL 的全称是 Domain Specific Language，即领域特定语言，或者直接翻译成"特定领域的语言"，再直接点，其实就是这个语言不通用，只能用于特定的某个领域，俗称"小语言"。因此 DSL 也是语言。

实际上，Gradle 脚本大多都是使用 groovy 语言编写的。

Groovy 是一门 jvm 语言，功能比较强大，细节也很多，全部学习的话比较耗时，对我们来说收益较小，并且玩转 Gradle 并不需要学习 Groovy 的全部细节，所以其实我们只需要学一些 Groovy 基础语法与 API 即可。

为何要使用 Groovy

Groovy 是一种基于 JVM 的敏捷开发语言，它结合了众多脚本语言的强大的特性，由于同时又能与 Java 代码很好的结合。一句话：既有面向对象的特性又有纯粹的脚本语言的特性。

由于 Groovy 运行在 JVM 上，因此也可以使用 Java 语言编写的组建。

简单来说，Groovy 提供了更加灵活简单的语法，大量的语法糖以及闭包特性可以让你用更少的代码来实现和 Java 同样的功能。

如何编译运行 Groovy

Groovy 是一门 jvm 语言，它最终是要编译成 class 文件然后在 jvm 上执行，所以 Java 语言的特性 Groovy 都支持，Groovy 支持 99%的 java 语法，我们完全可以在 Groovy 代码中直接粘贴 java 代码。

可以安装 Groovy sdk 来编译和运行。但是我并不想搞那么麻烦，毕竟我们的最终目的只是学习 Gradle。

推荐大家通过这种方式来编译和运行 Groovy。

在当面目录下创建 build.gradle 文件，在里面创建一个 task，然后在 task 中编写 Groovy 代码即可，如下所示：

```
task(testGroovy).doLast {
    println "开始运行自定义 task"
    test()
}

def test() {
    println "执行 Groovy 语法的代码"
    System.out.println("执行 Java 语法的代码!");
}
```

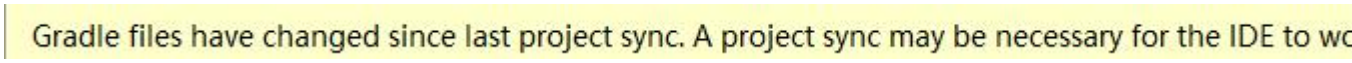
然后在命令行终端中执行如下命令即可：

```
gradle testGroovy
> Configure project :app

> Task :app:testGroovy
开始运行自定义 task
执行 Groovy 语法的代码
执行 Java 语法的代码!

BUILD SUCCESSFUL in 3s1 actionable task: 1 executed
```

我们知道，在 Android 项目中，我们只要更改 `build.gradle` 文件一点内容，AS 就会提示我们同步：



但是在我们测试 Groovy 时中，我们更改 `build.gradle` 文件后可以不必去同步，执行命令时会自动执行你修改后的最新逻辑。

最基本的语法

Groovy 中的类和方法默认都是 public 权限的，所以我们可以省略 public 关键字，除非我们想使用 private。
Groovy 中的类型是弱化的，所有的类型都可以动态推断，但是 Groovy 仍然是强类型的语言，类型不匹配仍然会报错。
Groovy 中通过 `def` 关键字来声明变量和方法。
Groovy 中很多东西都是可以省略的，比如

- 语句后面的分号是可以省略的
- def 和 变量的类型 中的其中之一是可以省略的(不能全部省略)
- def 和 方法的返回值类型 中的其中之一是可以省略的(不能全部省略)
- 方法调用时的圆括号是可以省略的
- 方法声明中的参数类型是可以省略的
- 方法的最后一句表达式可作为返回值返回，而不需要 return 关键字

    > 省略 return 关键字并不是一个好的习惯，就如同 if else while 后面只有一行语句时可以省略大括号一样，以后如果添加了其他语句，很有可能会导致逻辑错误

-

```
def int a = 1; //如果 def 和 类型同时存在，IDE 会提示你"def 是不需要的(is unnecessary)"def String b = "hello world" //省略分号，存在分号时也会提示你 unnecessarydef c = 1 //省略类型
def hello() {  //省略方法声明中的返回值类型
   println ("hello world");
   println "hello groovy" //省略方法调用时的圆括号
   return 1;
}
def hello(String msg) {
   println "hello" + msg //省略方法调用时的圆括号
   1;                  //省略 return
}

int hello(msg) { //省略方法声明中的参数类型
   println msg
   return 1 // 这个 return 不能省略
```

```
    println "done" //这一行代码是执行不到的，IDE 会提示你 Unreachable statement，但语法没错
}
```

支持的数据类型

在 Groovy 中，数据类型有：

- Java 中的基本数据类型
- Java 中的对象
- Closure（闭包）
- 加强的 List、Map 等集合类型
- 加强的 File、Stream 等 IO 类型

类型可以显示声明，也可以用 def 来声明，用 def 声明的类型 Groovy 将会进行类型推断。
基本数据类型和对象和 Java 中的一致，只不过在 Gradle 中，对象默认的修饰符为 public。

String

String 的特色在于字符串的拼接，比如

```
def a = 1def b = "hello"def c = "a=${a}, b=${b}"
println c //a=1, b=hello
```

闭包

Groovy 中有一种特殊的类型，叫做 Closure，翻译过来就是闭包，这是一种类似于 C 语言中函数指针的东西。
闭包用起来非常方便，在 Groovy 中，闭包作为一种特殊的数据类型而存在，闭包可以作为方法的参数和返回值，也可以作为一个变量而存在。
闭包可以有返回值和参数，当然也可以没有。下面是几个具体的例子：

```
def test() {
    def closure = { String parameters -> //闭包的基本格式
        println parameters
    }
    def closure2 = { a, b -> // 省略了闭包的参数类型
        println "a=${a}, b=${b}"
    }
    def closure3 = { a ->
        a + 1 //省略了 return
    }
    def closure4 = { // 省略了闭包的参数声明
        println "参数为 ${it}" //如果闭包不指定参数，那么它会有一个隐含的参数 it
    }
```

```
    closure("包青天") //包青天
    closure2 10086, "包青天" //a=10086, b=包青天
    println closure3(1) //2
    //println closure3 2 //不允许省略圆括号，会提示：Cannot get
property '1' on null object
    closure4() //参数为 null
    closure4 //不允许省略圆括号，但是并不会报错
    closure4 10086 //参数为 10086
}
```

闭包的一个难题是如何确定闭包的参数(包括参数的个数、参数的类型、参数的意义)，尤其当我们调用 Groovy 的 API 时，这个时候没有其他办法，只有查询 Groovy 的文档才能知道。

List 和 Map

Groovy 加强了 Java 中的集合类，比如 List、Map、Set 等。

基本使用如下：

```
def emptyList = []
def list = [10086, "hello", true]list[1] = "world"
assert list[1] == "world"
println list[0] //10086list << 5 //相当于 add()
assert 5 in list // 调用包含方法
println list //[10086, world, true, 5]
def range = 1..5
assert 2 in rangeprintln range //1..5println range.size() //5
def emptyMap = [:]
def map = ["id": 1, "name": "包青天"]map << [age: 29] //添加元素 map["id"]
= 10086 //访问元素方式一 map.name = "哈哈" //访问元素方式二，这种方式
最简单 println map //{id=10086, name=哈哈, age=29}
```

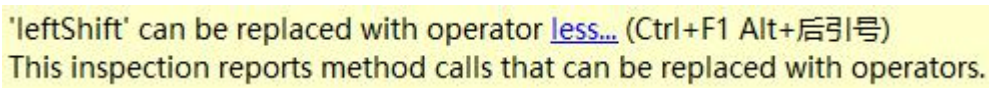
可以看到，通过 Groovy 来操作 List 和 Map 显然比 Java 简单的多。

上面有一个看起来很奇怪的操作符<<，其实这并没有什么大不了，<<表示向 List 中添加新元素的意思，这一点从 List 文档 当也能查到。

```
public List leftShift(Object value)
```

- Overloads the left shift operator to provide an easy way to append objects to a List. 重载左移位运算符，以提供将对象 append 到 List 的简单方法。
- Parameters: value - an Object to be added to the List.
- Returns: same List, after the value was added to it.

实际上，这个运算符是大量使用的，并且当你用 leftShift 方法时 IDE 也会提示你让你使用左移位运算符<<替换：



```
def list = [1, 2]list.leftShift 3
assert list == [1, 2, 3]list << 4
println list //[1, 2, 3, 4]
```

闭包的参数

这里借助 Map 再讲述下如何确定闭包的参数。比如我们想遍历一个 Map，我们想采用 Groovy 的方式，通过查看文档，发现它有如下两个方法，看起来和遍历有关：

- `Map each(Closure closure)`：Allows a Map to be iterated through using a closure.
- `Map eachWithIndex(Closure closure)`: Allows a Map to be iterated through using a closure.

可以发现，这两个 each 方法的参数都是一个闭包，那么我们如何知道闭包的参数呢？当然不能靠猜，还是要查文档。

`public Map each(Closure closure)`

- Allows a Map to be iterated through using a closure. If the closure takes `one parameter` then it will be passed the `Map.Entry` otherwise if the closure takes `two parameters` then it will be passed `the key and the value`.
- In general, the order in which the map contents are processed cannot be guaranteed(通常无法保证处理元素的顺序). In practise(在实践中), specialized forms of Map(特殊形式的 Map), e.g. a `TreeMap` will have its contents processed according to the natural ordering(自然顺序) of the map.

```
def result = ""
[a:1, b:3].each { key, value -> result += "$key$value" } //两个参数 assert result == "a1b3"
def result = ""
[a:1, b:3].each { entry -> result += entry } //一个参数 assert result == "a=1b=3"

[a: 1, b: 3].each { println "[${it.key} : ${it.value}]" } //一个隐含的参数 it，key 和 value 是属性名
```

试想一下，如果你不知道查文档，你又怎么知道 each 方法如何使用呢？光靠从网上搜，API 文档中那么多接口，搜的过来吗？记得住吗？

加强的 IO

在 Groovy 中，文件访问要比 Java 简单的多，不管是普通文件还是 xml 文件。怎么使用呢？查来 File 文档。

`public Object eachLine(Closure closure)`

- Iterates through this file line by line. Each line is passed to the given 1 or 2 arg closure. The file is read using a reader which is closed before this method returns.
- Parameters: closure - a closure (arg 1 is line, optional arg 2 is line number starting at line 1)

- Returns: the last value returned by the closure

可以看到，eachLine 方法也是支持 1 个或 2 个参数的，这两个参数分别是什么意思，就需要我们学会读文档了，一味地从网上搜例子，多累啊，而且很难彻底掌握：

```
def file = new File("a.txt")
file.eachLine { line, lineNo ->
    println "${lineNo} ${line}" //行号，内容
}

file.eachLine { line ->
    println "${line}" //内容
}
```

除了 eachLine，File 还提供了很多 Java 所没有的方法，大家需要浏览下大概有哪些方法，然后需要用的时候再去查就行了，这就是学习 Groovy 的正道。

访问 xml 文件

Groovy 访问 xml 有两个类：XmlParser 和 XmlSlurper，二者几乎一样，在性能上有细微的差别，不过这对于本文不重要。
groovy.util.XmlParser 的 API 文档

文档中的案例：

```
def xml = '<root><one a1="uno!"/><two>Some text!</two></root>'//或者 def xml = new XmlParser().parse(new File("filePath.xml"))def rootNode = new XmlParser().parseText(xml) //根节点 assert rootNode.name() == 'root' //根节点的名称 assert rootNode.one[0].@a1 == 'uno!' //根节点中的子节点 one 的 a1 属性的值 assert rootNode.two.text() == 'Some text!' //根节点中的子节点 two 的内容
rootNode.children().each { assert it.name() in ['one','two'] }
```

更多的细节查文档即可。

其他的一些语法特性

Getter 和 Setter

当你在 Groovy 中创建一个 beans 的时候，通常我们称为 POGOS(Plain Old Groovy Objects)，Groovy 会自动帮我们创建 getter/setter 方法。
当你对 getter/setter 方法有特殊要求，你尽可提供自己的方法，Groovy 默认的 getter/setter 方法会被替换。

构造器

有一个 bean

```
class Server {
    String name
    Cluster cluster
```

```
}
```

初始化一个实例的时候你可能会这样写：

```
def server = new Server()server.name = "Obelix"server.cluster = aCluster
```

其实你可以用带命名的参数的默认构造器，会大大减少代码量：

```
def server = new Server(name: "Obelix", cluster: aCluster)
```

Class 类型

在 Groovy 中 Class 类型的.class 后缀不是必须的，比如：

```
def func(Class clazz) {
    println clazz
}func(File.class) //class java.io.Filefunc(File) //class
java.io.File
```

使用 with()操作符

当更新一个实例的时候，你可以使用 with()来省略相同的前缀，比如：

```
Book book = new Book()
book.with {
   id = 1 //等价于 book.id = 1
   name = "包青天"
   start(10086)
   stop("包青天")
}
```

判断是否为真

所有类型都能转成布尔值，比如 null 和 void 相当于 0 或者相当于 false，其他则相当于 true，所以：

```
if (name) {}//等价于 if (name != null && name.length > 0) {}
```

在 Groovy 中可以在类中添加 asBoolean()方法来自定义是否为真。

简洁的三元表达式

在 Groovy 中，三元表达式可以更加简洁，比如：

```
def result = name ?: ""//等价于 def result = name != null ? name : ""
```

捕获任何异常

如果你实在不想关心 try 块里抛出何种异常，你可以简单的捕获所有异常，并且可以省略异常类型：

```
try {
    // ...
} catch (any) { //可以省略异常类型
    // something bad happens
```

```
}
```

这里的 any 并不包括 Throwable，如果你真想捕获 everything，你必须明确的标明你想捕获 Throwable

简洁的非空判断

在 java 中，你要获取某个对象的值必须要检查是否为 null，这就造成了大量的 if 语句；在 Groovy 中，非空判断可以用?.表达式，比如：

```
println order?.customer?.address//等价于 if (order != null) {
   if (order.getCustomer() != null) {
      if (order.getCustomer().getAddress() != null) {
         System.out.println(order.getCustomer().getAddress());
      }
   }
}
```

使用断言

在 Groovy 中，可以使用 assert 来设置断言，当断言的条件为 false 时，程序将会抛出异常：

```
def check(String name) {
   assert name // 检查方法传入的参数是否为空，name non-null and non-empty
according to Groovy Truth
   assert name?.size() > 3
}
```

==和 equals

Groovy 里的 is()方法等同于 Java 里的==。
Groovy 中的==是更智能的 equals()，比较两个类的时候，你应该使用 a.is(b)而不是==。
Groovy 中的==可以自动避免 NullPointerException 异常

```
status == "包青天"
//等价于 Java 中的 status != null && status.equals("包青天")
```

switch 方法

在 Groovy 中，switch 方法变得更加灵活，可以同时支持更多的参数类型：

```
def x = nulldef result = ""switch (x) {
    case "foo": result = "found foo" //没有 break 时会继续向下判断
    case "bar": result += "bar"
       break
    case [4, 5, 6]: result = "list" //匹配集合中的元素
       break
    case 12..30: result = "range" //匹配某个范围内的元素
       break
    case Integer: result = "integer" //匹配 Integer 类型
```

```
        break
    case { it > 3 }: result = "number > 3" //匹配表达式
        break
    case Number: result = "number" //匹配 Number 类型
        break    default: result = "default"
}
println result
```

字符串分行

Java 中，字符串过长需要换行时我们一般会这样写：

```
throw new PluginException("Failed to execute command list-applications:" +
    " The group with name " +
    parameterMap.groupname[0] +
    " is not compatible group of type " +
    SERVER_TYPE_NAME)
```

Groovy 中你可以用 \ 字符，而不需要添加一堆的双引号：

```
throw new PluginException("Failed to execute command list-applications: \
The group with name ${parameterMap.groupname[0]} \is not compatible group
of type ${SERVER_TYPE_NAME}")
```

或者使用多行字符串"""：

```
throw new PluginException("""Failed to execute command list-applications:
    The group with name ${parameterMap.groupname[0]}
    is not compatible group of type ${SERVER_TYPE_NAME)}""")
```

Groovy 中，单引号引起来的字符串是 java 字符串，不能使用占位符来替换变量，双引号引起的字符串则是 java 字符串或者 Groovy 字符串。

Import 别名

在 java 中使用两个类名相同但包名不同的两个类，像 java.util.List 和 java.wt.List，你必须使用完整的包名才能区分。Groovy 中则可以使用 import 别名：

```
import java.util.List as jurist //使用别名 import java.awt.List as
aListimport java.awt.WindowConstants as WCimport static pkg.SomeClass.foo
//静态引入方法
```

14. 热修复、插件化

插件化框架描述：dynamicLoadApk 为例子

- 1.可以通过 DexClassLoader 来对 apk 中的 dex 包进行加载访问

- 2.如何加载资源是个很大的问题，因为宿主程序中并没有 apk 中的资源，所以调用 R 资源会报错，所以这里使用了 Activity 中的实现 ContextImpl 的 getAssets() 和 getResources()再加上反射来实现。
- 3.由于系统启动 Activity 有很多初始化动作要做，而我们手动反射很难完成，所以可以采用接口机制，将 Activity 的大部分生命周期提取成接口，然后通过代理 Activity 去调用插件 Activity 的生命周期。同时如果像增加一个新生命周期方法的时候，只需要在接口中和代理中声明一下就行。
- 4.缺点：
    - 1.慎用 this，因为在 apk 中使用 this 并不代表宿主中的 activity，当然如果 this 只是表示自己的接口还是可以的。除此之外可以使用 that 代替 this。
    - 2.不支持 Service 和静态注册的 Broadcast
    - 3.不支持 LaunchMode 和 Apk 中 Activity 的隐式调用。

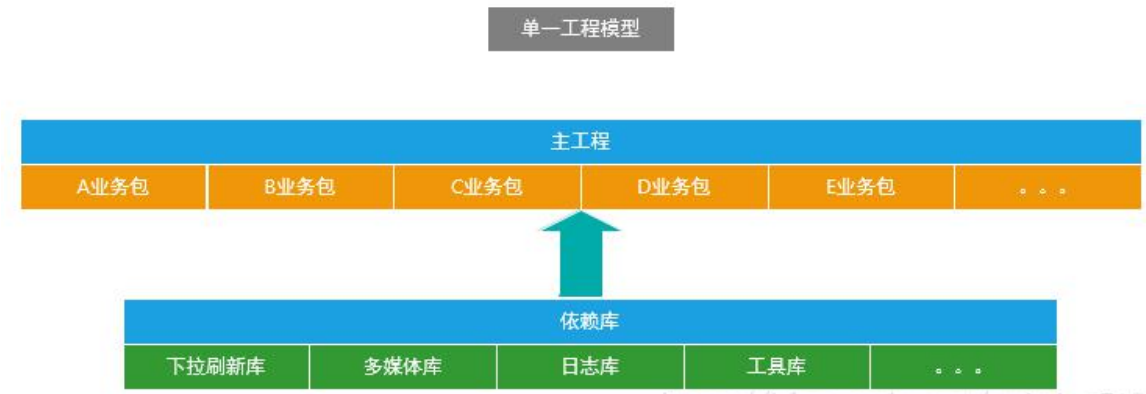
热修复：Andfix 为例子

- 1.大致原理：apkpatch 将两个 apk 做一次对比，然后找出不同的部分。可以看到生成的 apatch 了文件，后缀改成 zip 再解压开，里面有一个 dex 文件。通过 jadx 查看一下源码，里面就是被修复的代码所在的类文件,这些更改过的类都加上了一个_CF 的后缀，并且变动的方法都被加上了一个叫@MethodReplace 的 annotation，通过 clazz 和 method 指定了需要替换的方法。然后客户端 sdk 得到补丁文件后就会根据 annotation 来寻找需要替换的方法。最后由 JNI 层完成方法的替换。
- 2.无法添加新类和新的字段、补丁文件很容易被反编译、加固平台可能会使热补丁功能失效

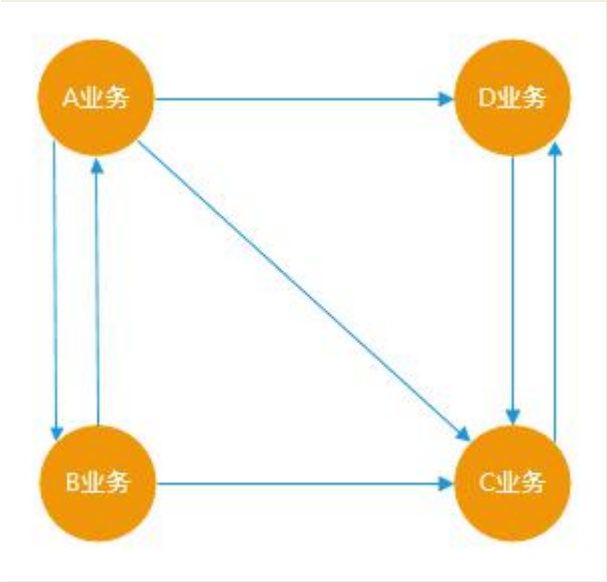
15. 组件化架构思路

1、为什么要项目组件化

随着 APP 版本不断的迭代，新功能的不断增加，业务也会变的越来越复杂，APP 业务模块的数量有可能还会继续增加，而且每个模块的代码也变的越来越多，这样发展下去单一工程下的 APP 架构势必会影响开发效率，增加项目的维护成本，每个工程师都要熟悉如此之多的代码，将很难进行多人协作开发，而且 Android 项目在编译代码的时候电脑会非常卡，又因为单一工程下代码耦合严重，每修改一处代码后都要重新编译打包测试，导致非常耗时，最重要的是这样的代码想要做单元测试根本无从下手，所以必须要有更灵活的架构代替过去单一的工程架构。



上图是目前比较普遍使用的 Android APP 技术架构，往往是在一个界面中存在大量的业务逻辑，而业务逻辑中充斥着各种网络请求、数据操作等行为，整个项目中也没有模块的概念，只有简单的以业务逻辑划分的文件夹，并且业务之间也是直接相互调用、高度耦合在一起的；

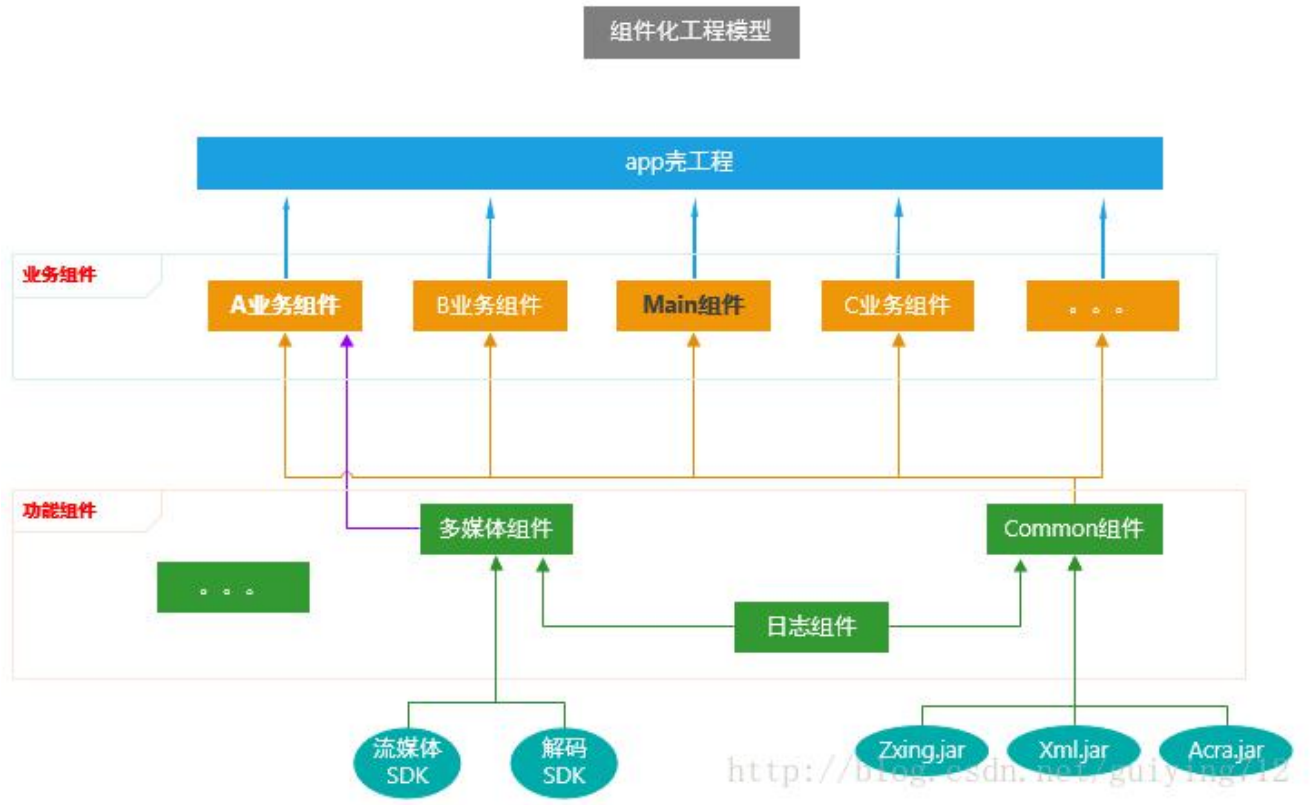


上图单一工程模型下的业务关系，总的来说就是：你中有我，我中有你，相互依赖，无法分离。
然而随着产品的迭代，业务越来越复杂，随之带来的是项目结构复杂度的极度增加，此时我们会面临如下几个问题：

1、实际业务变化非常快，但是单一工程的业务模块耦合度太高，牵一发而动全身；
2、对工程所做的任何修改都必须要编译整个工程；
3、功能测试和系统测试每次都要进行；
4、团队协同开发存在较多的冲突.不得不花费更多的时间去沟通和协调，并且在开发过程中，任何一位成员没办法专注于自己的功能点，影响开发效率；
5、不能灵活的对业务模块进行配置和组装；

为了满足各个业务模块的迭代而彼此不受影响，更好的解决上面这种让人头疼的依赖关系，就需要整改 App 的架构。

2、如何组件化



上图是组件化工程模型，为了方便理解这张架构图，下面会列举一些组件化工程中用到的名词的含义：

名词	含义
集成模式	所有的业务组件被“app 壳工程”依赖，组成一个完整的 APP；
组件模式	可以独立开发业务组件，每一个业务组件就是一个 APP；
app 壳工程	负责管理各个业务组件，和打包 apk，没有具体的业务功能；
业务组件	根据公司具体业务而独立形成一个的工程；
功能组件	提供开发 APP 的某些基础功能，例如打印日志、树状图等；
Main 组件	属于业务组件，指定 APP 启动页面、主界面；
Common 组件	属于功能组件，支撑业务组件的基础，提供多数业务组件需要的功能，例如提供网络请求功能；

**

Android APP 组件化架构的目标是告别结构臃肿，让各个业务变得相对独立，业务组件在组

件模式下可以独立开发，而在集成模式下又可以变为 arr 包集成到“app 壳工程”中，组成一个完整功能的 APP；
从组件化工程模型中可以看到，**业务组件之间是独立的，没有关联的**，这些业务组件在集成模式下是一个个 library，被 app 壳工程所依赖，组成一个具有完整业务功能的 APP 应用，但是在组件开发模式下，业务组件又变成了一个个 application，它们可以独立开发和调试，由于在组件开发模式下，业务组件们的代码量相比于完整的项目差了很远，因此在运行时可以显著减少编译时间。

这是组件化工程模型下的业务关系，业务之间将不再直接引用和依赖，而是通过“路由”这样一个中转站间接产生联系，而 Android 中的路由实际就是对 URL Scheme 的封装；
如此规模大的架构整改需要付出更高的成本，还会涉及一些潜在的风险，但是整改后的架构能够带来很多好处：

1、加快业务迭代速度，各个业务模块组件更加独立，不再出现业务耦合情况；
2、稳定的公共模块采用依赖库方式，提供给各个业务线使用，减少重复开发和维护工作量；
3、迭代频繁的业务模块采用组件方式，各业务研发可以互不干扰、提升协作效率，并控制产品质量；
4、为新业务随时集成提供了基础，所有业务可上可下，灵活多变；
5、降低团队成员熟悉项目的成本，降低项目的维护难度；
6、加快编译速度，提高开发效率；
7、控制代码权限，将代码的权限细分到更小的粒度；

3、组件化实施流程

1）组件模式和集成模式的转换

Android Studio 中的 Module 主要有两种属性，分别为：

1、**application 属性**，可以独立运行的 Android 程序，也就是我们的 APP；

```
apply plugin: ‘com.android.application’
```

2、**library 属性**，不可以独立运行，一般是 Android 程序依赖的库文件；

```
apply plugin: ‘com.android.library’
```

Module 的属性是在每个组件的 **build.gradle** 文件中配置的，当我们在组件模式开发时，业务组件应处于 application 属性，这时的业务组件就是一个 Android App，可以独立开发和调试；而当我们转换到集成模式开发时，业务组件应该处于 library 属性，这样才能被我们的“app 壳工程”所依赖，组成一个具有完整功能的 APP；

但是我们如何让组件在这两种模式之间自动转换呢？总不能每次需要转换模式的时候去每个业务组件的 Gralde 文件中去手动把 Application 改成 library 吧？如果我们的项目只有两三个组件那么这个办法肯定是可行的，手动去改一遍也用不了多久，但是在大型项目中我们可能会有十几个业务组件，再去手动改一遍必定费时费力，这时候就需要程序员发挥下懒的本质了。

试想，我们经常在写代码的时候定义静态常量，那么定义静态常量的目的什么呢？当一个常量需要被好几处代码引用的时候，把这个常量定义为静态常量的好处是当这个常量的值需要改变时我们只需要改变静态常量的值，其他引用了这个静态常量的地方都会被改变，**做到了一次改变，到处生效**；根据这个思想，那么我们就可以在我们的代码中的某处定义一个决定业务组件属性的常量，然后让所有业务组件的 build.gradle 都引用这个常量，这样当我们改变了常量值的时候，所有引用了这个常量值的业务组件就会根据值的变化改变自己的属性；可是问题来了？静态常量是用 Java 代码定义的，而改变组件属性是需要在 Gradle 中定义的，Gradle 能做到吗？

Gradle 自动构建工具有一个重要属性，可以帮助我们完成这个事情。每当我们用 AndroidStudio 创建一个 Android 项目后，就会在项目的根目录中生成一个文件 **gradle.properties**，我们将使用这个文件的一个重要属性：**在 Android 项目中的任何一个 build.gradle 文件中都可以把 gradle.properties 中的常量读取出来**；那么我们在上面提到解决办法就有了实际行动的方法，首先我们在 gradle.properties 中定义一个常量值 **isModule（是否是组件开发模式，true 为是，false 为否）**：

```
# 每次更改“isModule”的值后，需要点击 "Sync Project" 按钮
isModule=false
```

然后我们在业务组件的 build.gradle 中读取 **isModule**，但是 gradle.properties 还有一个重要属性： **gradle.properties 中的数据类型都是 String 类型，使用其他数据类型需要自行转换**；也就是说我们读到 isModule 是个 String 类型的值，而我们需要的是 Boolean 值，代码如下：

```
if (isModule.toBoolean()) {
    apply plugin: 'com.android.application'
```

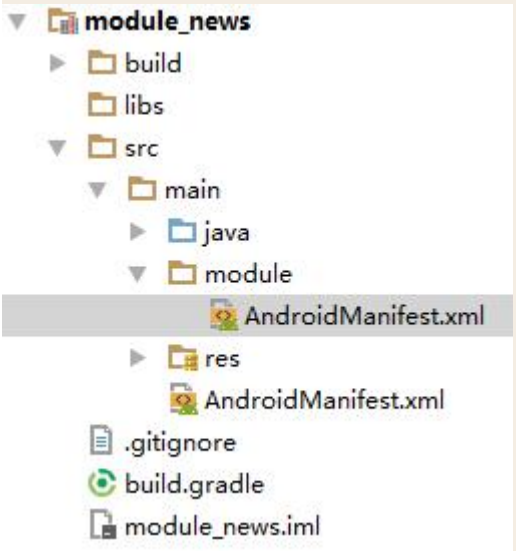
```
} else {
    apply plugin: 'com.android.library'
}
```

这样我们第一个问题就解决了，当然了 **每次改变 isModule 的值后，都要同步项目才能生效；**

2）组件之间 AndroidManifest 合并问题

在 AndroidStudio 中每一个组件都会有对应的 AndroidManifest.xml，用于声明需要的权限、Application、Activity、Service、Broadcast 等，当项目处于组件模式时，业务组件的 AndroidManifest.xml 应该具有一个 Android APP 所具有的的所有属性，尤其是声明 Application 和要 launch 的 Activity，但是当项目处于集成模式的时候，每一个业务组件的 AndroidManifest.xml 都要合并到“app 壳工程”中，要是每一个业务组件都有自己的 Application 和 launch 的 Activity，那么合并的时候肯定会冲突，试想一个 APP 怎么可能会有多个 Application 和 launch 的 Activity 呢？

但是大家应该注意到这个问题是在组件开发模式和集成开发模式之间转换引起的问题，而在上一节中我们已经解决了组件模式和集成模式转换的问题，另外大家应该都经历过将 Android 项目从 Eclipse 切换到 AndroidStudio 的过程，由于 Android 项目在 Eclipse 和 AndroidStudio 开发时 AndroidManifest.xml 文件的位置是不一样的，我们需要在 **build.gradle 中指定下 AndroidManifest.xml 的位置**，AndroidStudio 才能读取到 AndroidManifest.xml，这样解决办法也就有了，我们可以**为组件开发模式下的业务组件再创建一个 AndroidManifest.xml，然后根据 isModule 指定 AndroidManifest.xml 的文件路径，让业务组件在集成模式和组件模式下使用不同的 AndroidManifest.xml**，这样表单冲突的问题就可以规避了。



上图是组件化项目中一个标准的业务组件目录结构，首先我们在 main 文件夹下创建一个 module 文件夹用于存放组件开发模式下业务组件的 AndroidManifest.xml，而

AndroidStudio 生成的 AndroidManifest.xml 则依然保留，并用于集成开发模式下业务组件的表单；然后我们需要在业务组件的 build.gradle 中指定表单的路径，代码如下：

```
  sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
            }
        }
    }
```

这样在不同的开发模式下就会读取到不同的 AndroidManifest.xml ，然后我们需要修改这两个表单的内容以为我们不同的开发模式服务。

首先是集成开发模式下的 AndroidManifest.xml，前面我们说过集成模式下，业务组件的表单是绝对不能拥有自己的 Application 和 launch 的 Activity 的，也不能声明 APP 名称、图标等属性，总之 app 壳工程有的属性，业务组件都不能有，下面是一份**标准的集成开发模式**下业务组件的 AndroidManifest.xml：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.guiying.girls">

    <application android:theme="@style/AppTheme">
        <activity
            android:name=".main.GirlsActivity"
            android:screenOrientation="portrait" />
        <activity
            android:name=".girl.GirlActivity"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar" />
    </application>
</manifest>
```

我在这个表单中只声明了应用的主题，而且这个主题还是跟 app 壳工程中的主题是一致的，都引用了 common 组件中的资源文件，在这里声明主题是为了方便这个业务组件中有使用默认主题的 Activity 时就不用再给 Activity 单独声明 theme 了。

然后是组件开发模式下的表单文件：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.guiying.girls">

    <application
        android:name="debug.GirlsApplication"
        android:allowBackup="true"
```

```
        android:icon="@mipmap/ic_launcher"
        android:label="@string/girls_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".main.GirlsActivity"
            android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".girl.GirlActivity"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar" />
    </application>
</manifest>
```

组件模式下的业务组件表单就是一个 Android 项目普通的 AndroidManifest.xml，这里就不在过多介绍了。

3）全局 Context 的获取及组件数据初始化

当 Android 程序启动时，Android 系统会为每个程序创建一个 Application 类的对象，并且只创建一个，application 对象的生命周期是整个程序中最长的，它的生命周期就等于这个程序的生命周期。在默认情况下应用系统会自动生成 Application 对象，但是如果我们自定义了 Application，那就需要在 AndroidManifest.xml 中声明告知系统，实例化的时候，是实例化我们自定义的，而非默认的。

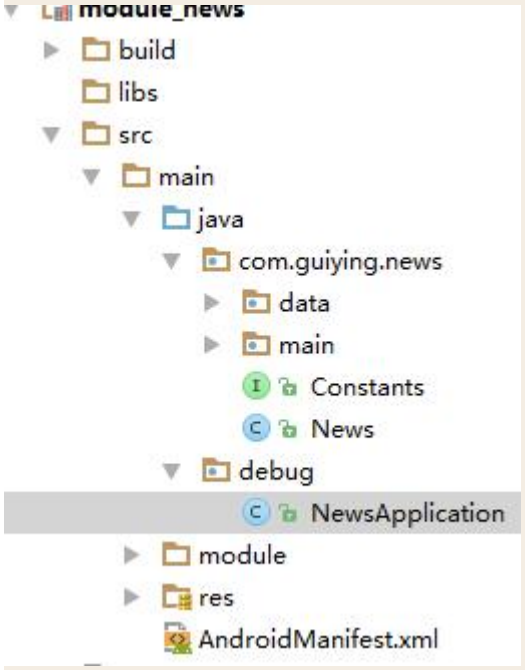
但是我们在组件化开发的时候，可能为了数据的问题每一个组件都会自定义一个 Application 类，如果我们在自己的组件中开发时需要获取 **全局的 Context**，一般都会直接获取 application 对象，但是当所有组件要打包合并在一起的时候就会出现问题，因为最后程序只有一个 Application，我们组件中自己定义的 Application 肯定是没法使用的，因此我们需要想办法再任何一个业务组件中都能获取到全局的 Context，而且这个 Context 不管是在组件开发模式还是在集成开发模式都是生效的。

在 组件化工程模型图中，功能组件集合中有一个 **Common 组件**， Common 有公共、公用、共同的意思，所以这个组件中主要封装了项目中需要的基础功能，并且每一个业务组件都要依赖 Common 组件，Common 组件就像是万丈高楼的地基，而业务组件就是在 Common 组件这个地基上搭建起来我们的 APP 的，Common 组件会专门在一个章节中讲解，这里只讲 Common 组件中的一个功能，在 Common 组件中我们封装了项目中用到的各种 Base 类，这些基类中就有 **BaseApplication 类**。

BaseApplication 主要用于各个业务组件和 app 壳工程中声明的 Application 类继承用的，只要各个业务组件和 app 壳工程中声明的 Application 类继承了 BaseApplication，当应用启动时 BaseApplication 就会被动实例化，这样从 BaseApplication 获取的 Context 就会生效，也就从根本上解决了我们不能直接从各个组件获取全局 Context 的问题；

这时候大家肯定都会有个疑问？不是说了业务组件不能有自己的 Application 吗，怎么还让他们继承 BaseApplication 呢？其实我前面说的是业务组件不能在集成模式下拥有自己的 Application，但是这不代表业务组件也不能在组件开发模式下拥有自己的 Application，其实业务组件在组件开发模式下必须要有自己的 Application 类，一方面是为了让 BaseApplication 被实例化从而获取 Context，还有一个作用是，**业务组件自己的 Application 可以在组件开发模式下初始化一些数据**，例如在组件开发模式下，A 组件没有登录页面也没法登录，因此就无法获取到 Token，这样请求网络就无法成功，因此我们需要在 A 组件这个 APP 启动后就应该已经登录了，这时候组件自己的 Application 类就有了用武之地，我们在组件的 Application 的 onCreate 方法中模拟一个登陆接口，在登陆成功后将数据保存到本地，这样就可以处理 A 组件中的数据业务了；另外我们也**可以在组件 Application 中初始化一些第三方库**。

但是，实际上业务组件中的 Application 在最终的集成项目中是没有什么实际作用的，组件自己的 Application 仅限于在组件模式下发挥功能，因此我们需要在将项目从组件模式转换到集成模式后将组件自己的 Application 剔除出我们的项目；在 AndroidManifest 合并问题小节中介绍了如何在不同开发模式下让 Gradle 识别组件表单的路径，这个方法也同样适用于 Java 代码；



我们在 Java 文件夹下创建一个 debug 文件夹，用于存放不会在业务组件中引用的类，例如上图中的 NewsApplication ，你甚至**可以在 debug 文件夹中创建一个 Activity，然后组件表单中声明启动这个 Activity，在这个 Activity 中不用 setContentView，只需要在启动你的目标 Activity 的时候传递参数就行，这样就就可以解决组件模式下某些 Activity 需要 getIntent 数据而没有办法拿到的情况**，代码如下；

```
public class LauncherActivity extends AppCompatActivity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        request();
        Intent intent = new Intent(this, TargetActivity.class);
        intent.putExtra("name", "avcd");
        intent.putExtra("syscode", "023e2e12ed");
        startActivity(intent);
        finish();
    }

    //申请读写权限
    private void request() {
        AndPermission.with(this)
                .requestCode(110)
                .permission(Manifest.permission.WRITE_EXTERNAL_STORAGE,
                        Manifest.permission.CAMERA,
Manifest.permission.READ_PHONE_STATE)
                .callback(this)
                .start();
    }

}
```

接下来在业务组件的 build.gradle 中，根据 isModule 是否是集成模式将 debug 这个 Java 代码文件夹排除：

```
    sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
                //集成开发模式下排除 debug 文件夹中的所有 Java 文件
                java {
                    exclude 'debug/**'
                }
            }
        }
    }
```

4）library 依赖问题

在介绍这一节的时候，先说一个问题，在**组件化工程模型图**中，多媒体组件和 Common 组件都依赖了日志组件，而 A 业务组件有同时依赖了多媒体组件和 Common 组件，这时候就会有人问，你这样搞岂不是日志组件要被重复依赖了，而且 Common 组件也被每一个业务组件依赖了，这样不出问题吗？

其实大家完全没有必要担心这个问题，如果真有重复依赖的问题，在你编译打包的时候就会报错，如果你还是不相信的话可以反编译下最后打包出来的 APP，看看里面的代码你就知道了。组件只是我们在代码开发阶段中为了方便叫的一个术语，在组件被打包进 APP 的时候是没有这个概念的，这些组件最后都会被打包成 arr 包，然后被 app 壳工程所依赖，在构建 APP 的过程中 Gradle 会自动将重复的 arr 包排除，APP 中也就不会存在相同的代码了；

但是虽然组件是不会重复了，但是我们还是要考虑另一个情况，我们在 build.gradle 中 compile 的第三方库，例如 AndroidSupport 库经常会被一些开源的控件所依赖，而我们自己一定也会 compile AndroidSupport 库 ，这就会造成第三方包和我们自己的包存在重复加载，解决办法就是找出那个多出来的库，并将多出来的库给排除掉，而且 Gradle 也是支持这样做的，分别有两种方式：**根据组件名排除或者根据包名排除**，下面以排除 support-v4 库为例：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile("com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion") {
        exclude module: 'support-v4'//根据组件名排除
        exclude group: 'android.support.v4'//根据包名排除
    }
}
```

library 重复依赖的问题算是都解决了，但是我们在开发项目的时候会依赖很多开源库，而这些库每个组件都需要用到，要是每个组件都去依赖一遍也是很麻烦的，尤其是给这些库升级的时候，为了方便我们统一管理第三方库，我们将给给整个工程提供统一的依赖第三方库的入口，前面介绍的 Common 库的作用之一就是统一依赖开源库，因为其他业务组件都依赖了 Common 库，所以这些业务组件也就间接依赖了 Common 所依赖的开源库。

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //Android Support
    compile
"com.android.support:appcompat-v7:$rootProject.supportLibraryVersion"
    compile "com.android.support:design:$rootProject.supportLibraryVersion"
    compile "com.android.support:percent:$rootProject.supportLibraryVersion"
    //网络请求相关
    compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
    compile
"com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
```

```
    compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
    //稳定的
    compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
    compile "com.orhanobut:logger:$rootProject.loggerVersion"
    compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile "com.google.code.gson:gson:$rootProject.gsonVersion"
    compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"

    compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
    compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"

    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

5）组件之间调用和通信

在组件化开发的时候，组件之间是没有依赖关系，我们不能在使用显示调用来跳转页面了，因为我们组件化的目的之一就是解决模块间的强依赖问题，假如现在要从 A 业务组件跳转到业务 B 组件，并且要携带参数跳转，这时候怎么办呢？而且组件这么多怎么管理也是个问题，这时候就需要引入“路由”的概念了，由本文开始的组件化模型下的业务关系图可知路由就是起到一个转发的作用。

这里我将介绍开源库的**“ActivityRouter”**，有兴趣的同学情直接去 ActivityRouter 的 Github 主页学习：ActivityRouter，ActivityRouter 支持给 Activity 定义 URL，这样就可以通过 URL 跳转到 Activity，并且支持从浏览器以及 APP 中跳入我们的 Activity，而且还支持通过 url 调用方法。下面将介绍如何将 ActivityRouter 集成到组件化项目中以实现组件之间的调用：

1、首先我们需要在 Common 组件中的 build.gradle 将 ActivityRouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

2、这一步我们需要先了解 **APT** 这个概念，**APT(Annotation Processing Tool)是一种处理注解的工具，它对源代码文件进行检测找出其中的 Annotation，使用 Annotation 进行额外的处理。 Annotation 处理器在处理 Annotation 时可以根据源文件中的 Annotation 生成额外的源文件和其它的文件(文件具体内容由 Annotation 处理器的编写者决定)，APT 还会编译生成的源文件和原来的源文件，将它们一起生成 class 文件**。在这里我们将在每一个**业务组件**的 build.gradle 都引入 ActivityRouter 的 Annotation 处理器，我们将会在声明组件和 Url 的时候使用，annotationProcessor 是 Android 官方提供的 Annotation 处理器插件，代码如下：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
}
```

3、接下来需要在 **app 壳工程**的 AndroidManifest.xml 配置，到这里 ActivityRouter 配置就算完成了：

```
<!--声明整个应用程序的路由协议-->
        <activity
android:name="com.github.mzule.activityrouter.router.RouterActivity"
            android:theme="@android:style/Theme.NoDisplay">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />

                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />

                <data android:scheme="@string/global_scheme" /> <!-- 改成自己的
scheme -->
            </intent-filter>
        </activity>
        <!--发送崩溃日志界面-->
```

4、接下来我们将声明项目中的业务组件，声明方法如下：

```
@Module("girls")public class Girls {
}
```

在每一个业务组件的 java 文件的根目录下创建一个类，用 **注解@Module** 声明这个业务组件；

然后在“**app 壳工程**”的 **应用 Application** 中使用 **注解@Modules** 管理我们声明的所有业务组件，方法如下：

```
@Modules({"main", "girls", "news"})public class MyApplication extends
BaseApplication {
}
```

到这里组件化项目中的所有业务组件就被声明和管理起来了，组件之间的也就可以互相调用了，当然前提是要给业务组件中的 Activity 定义 URL。

5、例如我们给 Girls 组件 中的 GirlsActivity 使用 **注解@Router** 定义一个 URL：“news”，方法如下：

```
@Router("girls")public class GirlsActivity extends BaseActionBarActivity {

    private GirlsView mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new GirlsView(this);
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}
```

然后我们就可以**在项目中的任何一个地方通过 URL 地址 ： module://girls，调用 GirlsActivity**，方法如下：

```
    Routers.open(MainActivity.this, "module://girls");
```

- 1

组件之间的调用解决后，另外需要解决的就是组件之间的通信，例如 A 业务组件中有消息列表，而用户在 B 组件中操作某个事件后会产生一条新消息，需要通知 A 组件刷新消息列表，这样业务场景需求可以使用 Android 广播来解决，也可以使用第三方的事件总线来实现，比如 EventBus。

6）组件之间资源名冲突

因为我们拆分出了很多业务组件和功能组件，在把这些组件合并到**“app 壳工程”**时候就有可能会出现资源名冲突问题，例如 **A 组件和 B 组件都有一张叫做“ic_back”的图标**，这时候在集成模式下打包 APP 就会编译出错，解决这个问题最简单的办法就是**在项目中约定资源文件命名规约，比如强制使每个资源文件的名称以组件名开始，这个可以根据实际情况和开发人员制定规则**。当然了万能的 Gradle 构建工具也提供了解决方法，通过在在组件的 build.gradle 中添加如下的代码：

```
    //设置了 resourcePrefix 值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
    //但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。
    resourcePrefix "girls_"
```

但是设置了这个属性后有个问题，所有的资源名必须以指定的字符串做前缀，否则会报错，而且 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名；所以我并不推荐使用这种方法来解决资源名冲突。

4、组件化项目的工程类型

在组件化工程模型中主要有：**app 壳工程、业务组件和功能组件 3 种类型，而业务组件中的 Main 组件和功能组件中的 Common 组件比较特殊**，下面将分别介绍。

1）app 壳工程

app 壳工程是从名称来解释就是一个空壳工程，没有任何的业务代码，也不能有 Activity，但它又必须被单独划分成一个组件，而不能融合到其他组件中，是因为它有如下几点重要功能：

1、**app 壳工程中声明了我们 Android 应用的 Application**，这个 Application 必须继承自 Common 组件中的 BaseApplication（如果你无需实现自己的 Application 可以直接在表单声明 BaseApplication），因为只有这样，在打包应用后才能让 BaseApplication 中的 Context 生效，当然你还可以在这个 Application 中初始化我们工程中使用到的库文件，还可以在这里解决 Android 引用方法数不能超过 65535 的限制，对崩溃事件的捕获和发送也可以在这里声明。

2、**app 壳工程的 AndroidManifest.xml 是我 Android 应用的根表单**，应用的名称、图标以及是否支持备份等等属性都是在这份表单中配置的，其他组件中的表单最终在集成开发模式下都被合并到这份 AndroidManifest.xml 中。

3、**app 壳工程的 build.gradle 是比较特殊的**，app 壳不管是在集成开发模式还是组件开发模式，它的属性始终都是：com.android.application，因为最终其他的组件都要被 app 壳工程所依赖，被打包进 app 壳工程中，这一点从组件化工程模型图中就能体现出来，所以 app 壳工程是不需要单独调试单独开发的。另外 Android 应用的打包签名，以及 buildTypes 和 defaultConfig 都需要在这里配置，而它的 dependencies 则需要根据 isModule 的值分别依赖不同的组件，在组件开发模式下 app 壳工程只需要依赖 Common 组件，或者为了防止报错也可以根据实际情况依赖其他功能组件，而在集成模式下 app 壳工程必须依赖所有在应用 Application 中声明的业务组件，并且不需要再依赖任何功能组件。

下面是一份 app 壳工程 的 build.gradle 文件：

```
apply plugin: 'com.android.application'
static def buildTime() {
    return new Date().format("yyyyMMdd");
}

android {
    signingConfigs {
        release {
            keyAlias 'guiying712'
            keyPassword 'guiying712'
            storeFile file('/mykey.jks')
            storePassword 'guiying712'
        }
    }

    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion
    defaultConfig {
        applicationId "com.guiying.androidmodulepattern"
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
        multiDexEnabled false
        //打包时间
        resValue "string", "build_time", buildTime()
    }

    buildTypes {
        release {
            //更改 AndroidManifest.xml 中预先定义好占位符信息
            //manifestPlaceholders = [app_icon: "@drawable/icon"]
            // 不显示 Log
            buildConfigField "boolean", "LEO_DEBUG", "false"
```

```
            //是否 zip 对齐
            zipAlignEnabled true
            // 缩减 resource 文件
            shrinkResources true
            //Proguard
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
            //签名
            signingConfig signingConfigs.release
        }

        debug {
            //给 applicationId 添加后缀".debug"
            applicationIdSuffix ".debug"
            //manifestPlaceholders = [app_icon: "@drawable/launch_beta"]
            buildConfigField "boolean", "LOG_DEBUG", "true"
            zipAlignEnabled false
            shrinkResources false
            minifyEnabled false
            debuggable true
        }
    }


}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
    if (isModule.toBoolean()) {
        compile project(':lib_common')
    } else {
        compile project(':module_main')
        compile project(':module_girls')
        compile project(':module_news')
    }
}
```

2）功能组件和 Common 组件

功能组件是为了支撑业务组件的某些功能而独立划分出来的组件，功能实质上跟项目中引入的第三方库是一样的，功能组件的特征如下：

1、功能组件的 AndroidManifest.xml 是一张空表，这张表中只有功能组件的包名；

2、功能组件不管是在集成开发模式下还是组件开发模式下属性始终是：com.android.library，所以功能组件是不需要读取 gradle.properties 中的 isModule 值的；另外功能组件的 build.gradle 也无需设置 buildTypes ，只需要 dependencies 这个功能组件需要的 jar 包和开源库。

下面是一份 普通 的功能组件的 build.gradle 文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }

}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Common 组件除了有功能组件的普遍属性外，还具有其他功能：

1、Common 组件的 AndroidManifest.xml 不是一张空表，这张表中声明了我们 Android 应用用到的所有使用权限 uses-permission 和 uses-feature，放到这里是因为在组件开发模式下，所有业务组件就无需在自己的 AndroidManifest.xm 声明自己要用到的权限了。

2、Common 组件的 build.gradle 需要统一依赖业务组件中用到的 第三方依赖库和 jar 包，例如我们用到的 ActivityRouter、Okhttp 等等。

3、Common 组件中封装了 Android 应用的 Base 类和网络请求工具、图片加载工具等等，公用的 widget 控件也应该放在 Common 组件中；业务组件中都用到的数据也应放于 Common 组件中，例如保存到 SharedPreferences 和 DataBase 中的登陆数据；

4、Common 组件的资源文件中需要放置项目公用的 Drawable、layout、sting、dimen、color 和 style 等等，另外项目中的 Activity 主题必须定义在 Common 中，方便和 BaseActivity 配合保持整个 Android 应用的界面风格统一。

下面是一份 Common 功能组件的 build.gradle 文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }

}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //Android Support
    compile
"com.android.support:appcompat-v7:$rootProject.supportLibraryVersion"
    compile "com.android.support:design:$rootProject.supportLibraryVersion"
    compile "com.android.support:percent:$rootProject.supportLibraryVersion"
    //网络请求相关
    compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
    compile
"com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
    compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
    //稳定的
    compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
    compile "com.orhanobut:logger:$rootProject.loggerVersion"
    compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile "com.google.code.gson:gson:$rootProject.gsonVersion"
    compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"

    compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
    compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"

    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

2）业务组件和 Main 组件

业务组件就是根据业务逻辑的不同拆分出来的组件，业务组件的特征如下：

1、业务组件中要有两张 AndroidManifest.xml，分别对应组件开发模式和集成开发模式，这两张表的区别请查看 **组件之间 AndroidManifest 合并问题** 小节。

2、业务组件在集成模式下是不能有自己的 Application 的，但在组件开发模式下又必须实现自己的 Application 并且要继承自 Common 组件的 BaseApplication，并且这个 Application 不能被业务组件中的代码引用，因为它的功能就是为了使业务组件从 BaseApplication 中获取的全局 Context 生效，还有初始化数据之用。

3、业务组件有 debug 文件夹，这个文件夹在集成模式下会从业务组件的代码中排除掉，所以 debug 文件夹中的类不能被业务组件强引用，例如组件模式下的 Application 就是置于这个文件夹中，还有组件模式下开发给目标 Activity 传递参数的用的 launch Activity 也应该置于 debug 文件夹中；

4、业务组件必须在自己的 Java 文件夹中创建业务组件声明类，以使 **app 壳工程 中的 应用 Application 能够引用**，实现组件跳转，具体请查看 **组件之间调用和通信** 小节；

5、**业务组件必须在自己的 build.gradle 中根据 isModule 值的不同改变自己的属性**，在组件模式下是：com.android.application，而在集成模式下 com.android.library；同时还需要在 build.gradle 配置资源文件，如 指定不同开发模式下的 AndroidManifest.xml 文件路径，排除 debug 文件夹等；业务组件还必须在 dependencies 中依赖 Common 组件，并且引入 ActivityRouter 的注解处理器 annotationProcessor，以及依赖其他用到的功能组件。

下面是一份普通业务组件的 build.gradle 文件：

```
if (isModule.toBoolean()) {
    apply plugin: 'com.android.application'
} else {
    apply plugin: 'com.android.library'
}

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }
```

```
    sourceSets {
        main {
            if (isModule.toBoolean()) {
                manifest.srcFile 'src/main/module/AndroidManifest.xml'
            } else {
                manifest.srcFile 'src/main/AndroidManifest.xml'
                //集成开发模式下排除 debug 文件夹中的所有 Java 文件
                java {
                    exclude 'debug/**'
                }
            }
        }
    }

    //设置了 resourcePrefix 值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
    //但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。
    //resourcePrefix "girls_"


}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor "com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
    compile project(':lib_common')
}
```

Main 组件除了有业务组件的普遍属性外，还有一项重要功能：

1、Main 组件集成模式下的 AndroidManifest.xml 是跟其他业务组件不一样的，Main 组件的表单中声明了我们整个 Android 应用的 launch Activity，这就是 Main 组件的独特之处；所以我建议 SplashActivity、登陆 Activity 以及主界面都应属于 Main 组件，也就是说 Android 应用启动后要调用的页面应置于 Main 组件。

```
        <activity
            android:name=".splash.SplashActivity"
            android:launchMode="singleTop"
            android:screenOrientation="portrait"
            android:theme="@style/SplashTheme">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

5、组件化项目的混淆方案

组件化项目的 Java 代码混淆方案采用在集成模式下集中在 app 壳工程中混淆，各个业务组件不配置混淆文件。集成开发模式下在 app 壳工程中 build.gradle 文件的 release 构建类型中开启混淆属性，其他 buildTypes 配置方案跟普通项目保持一致，Java 混淆配置文件也放置在 app 壳工程中，各个业务组件的混淆配置规则都应该在 app 壳工程中的混淆配置文件中添加和修改。

之所以不采用在每个业务组件中开启混淆的方案，是因为 **组件在集成模式下都被 Gradle 构建成了 release 类型的 arr 包**，一旦业务组件的代码被混淆，而这时候代码中又出现了 bug，将很难根据日志找出导致 bug 的原因；另外每个业务组件中都保留一份混淆配置文件非常不便于修改和管理，这也是不推荐在业务组件的 build.gradle 文件中配置 buildTypes（构建类型）的原因。

6、工程的 build.gradle 和 gradle.properties 文件

1）组件化工程的 build.gradle 文件

在组件化项目中因为每个组件的 build.gradle 都需要配置 compileSdkVersion、buildToolsVersion 和 defaultConfig 等的版本号，而且每个组件都需要用到 annotationProcessor，**为了能够使组件化项目中的所有组件的 build.gradle 中的这些配置都能保持统一，并且也是为了方便修改版本号**，我们统一在 Android 工程根目录下的 build.gradle 中定义这些版本号，当然为了方便管理 Common 组件中的第三方开源库的版本号，最好也在这里定义这些开源库的版本号，然后在各个组件的 build.gradle 中引用 Android 工程根目录下的 build.gradle 定义的版本号，组件化工程的 build.gradle 文件代码如下：

```
buildscript {
    repositories {
        jcenter()
        mavenCentral()
    }

    dependencies {
        //classpath "com.android.tools.build:gradle:$localGradlePluginVersion"
```

```
        //$localGradlePluginVersion 是 gradle.properties 中的数据
        classpath "com.android.tools.build:gradle:$localGradlePluginVersion"
    }
}

allprojects {
    repositories {
        jcenter()
        mavenCentral()
        //Add the JitPack repository
        maven { url "https://jitpack.io" }
        //支持 arr 包
        flatDir {
            dirs 'libs'
        }
    }
}
task clean(type: Delete) {
    delete rootProject.buildDir
}
// Define versions in a single place//时间：2017.2.13；每次修改版本号都要添加修改时间
ext {
    // Sdk and tools
    //localBuildToolsVersion 是 gradle.properties 中的数据
    buildToolsVersion = localBuildToolsVersion
    compileSdkVersion = 25
    minSdkVersion = 16
    targetSdkVersion = 25
    versionCode = 1
    versionName = "1.0"
    javaVersion = JavaVersion.VERSION_1_8

    // App dependencies version
    supportLibraryVersion = "25.3.1"
    retrofitVersion = "2.1.0"
    glideVersion = "3.7.0"
    loggerVersion = "1.15"
    eventbusVersion = "3.0.0"
    gsonVersion = "2.8.0"
    photoViewVersion = "2.0.0"

    //需检查升级版本
    annotationProcessor = "1.1.7"
```

```
    routerVersion = "1.2.2"
    easyRecyclerVersion = "4.4.0"
    cookieVersion = "v1.0.1"
    toastyVersion = "1.1.3"
}
```

2）组件化工程的 gradle.properties 文件

在组件化实施流程中我们了解到 gradle.properties 有两个属性对我们非常有用：

1、在 Android 项目中的任何一个 build.gradle 文件中都可以把 gradle.properties 中的常量读取出来，不管这个 build.gradle 是组件的还是整个项目工程的 build.gradle；

2、gradle.properties 中的数据类型都是 String 类型，使用其他数据类型需要自行转换；

利用 gradle.properties 的属性不仅可以解决集成开发模式和组件开发模式的转换，而且还可以解决在多人协同开发 Android 项目的时候，因为开发团队成员的 Android 开发环境（开发环境指 Android SDK 和 AndroidStudio）不一致而导致频繁改变线上项目的 build.gradle 配置。

在每个 Android 组件的 build.gradle 中有一个属性：**buildToolsVersion**，表示构建工具的版本号，这个属性值对应 AndroidSDK 中的 **Android SDK Build-tools**，正常情况下 build.gradle 中的 buildToolsVersion 跟你电脑中 Android SDK Build-tools 的最新版本是一致的，比如现在 Android SDK Build-tools 的最新的版本是：25.0.3，那么我的 Android 项目中 build.gradle 中的 buildToolsVersion 版本号也是 25.0.3，但是一旦一个 Android 项目是由好几个人同时开发，总会出现每个人的开发环境 Android SDK Build-tools 是都是不一样的，并不是所有人都会经常升级更新 Android SDK，而且代码是保存到线上环境的（例如使用 SVN/Git 等工具），某个开发人员提交代码后线上 Android 项目中 build.gradle 中的 buildToolsVersion 也会被不断地改变。

另外一个原因是因为 Android 工程的根目录下的 build.gradle 声明了 Android Gradle 构建工具，而这个工具也是有版本号的，而且 **Gradle Build Tools** 的版本号跟 AndroidStudio 版本号一致的，但是有些开发人员基本很久都不会升级自己的 AndroidStudio 版本，导致团队中每个开发人员的 Gradle Build Tools 的版本号也不一致。

如果每次同步代码后这两个工具的版本号被改变了，开发人员可以自己手动改回来，并且不要把改动工具版本号的代码提交到线上环境，这样还可以勉强继续开发；但是很多公司都会使用持续集成工具（例如 Jenkins）用于持续的软件版本发布，而 Android 出包是需要 Android SDK Build-tools 和 Gradle Build Tools 配合的，一旦提交到线上的版本跟持续集成工具所依赖的 Android 环境构建工具版本号不一致就会导致 Android 打包失败。

为了解决上面问题就必须将 Android 项目中 build.gradle 中的 buildToolsVersion 和 GradleBuildTools 版本号从线上代码隔离出来，保证线上代码的 buildToolsVersion 和 Gradle Build Tools 版本号不会被人为改变。

具体的实施流程大家可以查看我的这篇博文： AndroidStudio 本地化配置 gradle 的 buildToolsVersion 和 gradleBuildTools

---

7、组件化项目 Router 的其他方案-ARouter

在组件化项目中使用到的跨组件跳转库 ActivityRouter 可以使用阿里巴巴的开源路由项目：阿里巴巴 ARouter；

ActivityRouter 和 ARouter 的接入组件化项目的方式是一样的，ActivityRouter 提供的功能目前 ARouter 也全部支持，但是 ARouter 还支持依赖注入解耦，页面、拦截器、服务等组件均会自动注册到框架。对于大家来说，没有最好的只有最适合的，大家可以根据自己的项目选择合适的 Router。

下面将介绍 ARouter 的基础使用方法，更多功能还需大家去 Github 自己学习；

1、首先 ARouter 这个框架是需要初始化 SDK 的，所以你需要在“app 壳工程”中的应用 Application 中加入下面的代码，**注意：在 debug 模式下一定要 openDebug**：

```
    if (BuildConfig.DEBUG) {
            //一定要在 ARouter.init 之前调用 openDebug
            ARouter.openDebug();
            ARouter.openLog();
        }
        ARouter.init(this);
```

2、首先我们依然需要在 Common 组件中的 build.gradle 将 ARouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //router
    compile 'com.alibaba:arouter-api:1.2.1.1'
}
```

3、然后在每一个**业务组件**的 build.gradle 都引入 ARouter 的 Annotation 处理器，代码如下：

```
android {
    defaultConfig {
    ...
    javaCompileOptions {
        annotationProcessorOptions {
```

```
        arguments = [ moduleName : project.getName() ]
        }
    }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor 'com.alibaba:arouter-compiler:1.0.3'
}
```

4、由于 ARouter 支持自动注册到框架，所以我们不用像 ActivityRouter 那样在各个组件中声明组件，当然更不需要在 Application 中管理组件了。 我们给 Girls 组件 中的 GirlsActivity 添加注解：**@Route(path = "/girls/list")**，需要注意的是**这里的路径至少需要有两级，/xx/xx**，之所以这样是因为 **ARouter 使用了路径中第一段字符串(/*/)作为分组**，比如像上面的"girls"，而分组这个概念就有点类似于 ActivityRouter 中的组件声明 @Module ，代码如下：

```
@Route(path = "/girls/list")public class GirlsActivity extends
BaseActionBarActivity {

    private GirlsView mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new GirlsView(this);
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}
```

然后我们就可以**在项目中的任何一个地方通过 URL 地址 ： /girls/list，调用 GirlsActivity**，方法如下：

```
    ARouter.getInstance().build("/girls/list").navigation();
```

16. 系统打包流程

一、添加 android 平台

添加之后，在项目目录的 platforms 下会生成一个 android 文件夹。ionic cordova platform add android

二、cordova 编译应用

使用 build 命令编译应用的发布版本，这个过程需要你的 android sdk 和环境变量、java jdk 和环境变量、android 的 gradle 配置没有错误。ionic cordova build android --prod --release

编译成功之后，在项目路径 platforms/android/build/outputs/apk/android-release-unsigned.apk 未签名文件，这个时候的 apk 还不能被安装到手机上。

三、生成签名文件

keytool -genkey -v -keystore jhy-release-key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias alias_jhy

输入的密码要记住，其他姓名地区等信息随便填吧，最好还是记住，成功之后在主目录下就生成了 jhy-release-key.keystore 文件，命令中 jhy-release-key.keystore 是生成文件的名字，alias_jhy 是别名，随便起但是要记住，一会签名要用到，其他信息如加密、有效日期等就不说了，无需改动。

生成后会提示:

JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore jhy-release-key.jks -destkeystore jhy-release-key.jks -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。执行命令:keytool -importkeystore -srckeystore jhy-release-key.jks -destkeystore jhy-release-key.jks -deststoretype pkcs12

执行结果:Warning: 已将 "jhy-release-key.jks" 迁移到 Non JKS/JCEKS。将 JKS 密钥库作为 "jhy-release-key.jks.old" 进行了备份。

四、签名应用文件

把在第二步生成的 android-release-unsigned.apk 拷贝到与生成的 jhy-release-key.jks 同一目录下，也就是项目的主目录下，执行命令:

jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore jhy-release-key.jks android-release-unsigned.apk alias_jhy

输入签名文件的密码，成功之后主目录下的 android-release-unsigned.apk 就被签名成功了，会比原来未被签名的 apk 文件大一点，能够安装到手机或 android 虚拟机上了。

签名完成后会提示没有时间戳，忽略即可

检测是否签名成功:

apksigner verify android-release-unsigned.apk

也可用以下命令签名并生成新 apk 文件

```
jarsigner -verbose -keystore jhy-release-key.jks -signedjar
notepad.apk android-release-unsigned.apk alias_jhy
```

数据存储在开发中是使用最频繁的，根据不同的情况选择不同的存储数据方式对于提高开发效率很有帮助。下面等台中实现数据存储的 5 种方式。

使用 SharedPreferences 存储数据

SharedPreferences 是 Android 平台上一个轻量级的存储类，主要是保存一些常用的配置比如窗口状态，一般在 onSaveInstance State 保存一般使用 SharedPreferences 完成，它提供了 Android 平台常规的 Long 长 整形、的保存。

它是什么样的处理方式呢? SharedPreferences 类似过去 Windows 系统上的 ini 配置文件，但是它分为多种权限 android123 提示最终是以 xml 方式来保存，整体效率来看不是特别的高，对于常规的轻量级而言比 SQLite 要好可以考虑自己定义文件格式。xml 处理时 Dalvik 会通过自带底层的本地 XML Parser 解析，比如 XMLpull 方式，较好。

它的本质是基于 XML 文件存储 key-value 键值对数据，通常用来存储一些简单的配置信息。

其存储位置在/data/data/<包名>/shared_prefs 目录下。

SharedPreferences 对象本身只能获取数据而不支持存储和修改，存储修改是通过 Editor 对象实现。

实现 SharedPreferences 存储的步骤如下：

1.根据 Context 获取 SharedPreferences 对象

2.利用 edit()方法获取 Editor 对象。

3.通过 Editor 对象存储 key-value 键值对数据。

4.通过 commit()方法提交数据。

下面是示例代码：

```
1  public class MainActivity extends Activity {
2   @Override
3       public void onCreate(Bundle savedInstanceState) {
4           super.onCreate(savedInstanceState);
5           setContentView(R.layout.main);
6
7           //获取 SharedPreferences 对象
8           Context ctx = MainActivity.this;
9           SharedPreferences sp = ctx.getSharedPreferences("SP", MODE_PRIVATE);
10          //存入数据
11          Editor editor = sp.edit();
12          editor.putString("STRING_KEY", "string");
13          editor.putInt("INT_KEY", 0);
```

```
14        editor.putBoolean("BOOLEAN_KEY", true);
15        editor.commit();
16
17        //返回 STRING_KEY 的值
18        Log.d("SP", sp.getString("STRING_KEY", "none"));
19        //如果 NOT_EXIST 不存在，则返回值为"none"
20        Log.d("SP", sp.getString("NOT_EXIST", "none"));
21    }
22}
```

这段代码执行过后，即在/data/data/com.test/shared_prefs 目录下生成了一个 SP.xml 文件，一个应用可以创建

SharedPreferences 对象与 SQLite 数据库相比，免去了创建数据库，创建表，写 SQL 语句等诸多操作，相对而言

SharedPreferences 也有其自身缺陷，比如其职能存储 boolean，int，float，long 和 String 五种简单的数据类型

询等。所以不论 SharedPreferences 的数据存储操作是如何简单，它也只能是存储方式的一种补充，而无法完全替

的其他数据存储方式。

文件存储数据

关于文件存储，Activity 提供了 openFileOutput()方法可以用于把数据输出到文件中，具体的实现过程与在 J2SE 环

一样的。

文件可用来存放大量数据，如文本、图片、音频等。

默认位置：/data/data/<包>/files/***.***。

代码示例：

```
1  public void save()
2   {
3         try {
4             FileOutputStream outStream=this.openFileOutput("a.txt",Context.MODE_WORLD_READABLE);
5             outStream.write(text.getText().toString().getBytes());
6             outStream.close();
7             Toast.makeText(MyActivity.this,"Saved",Toast.LENGTH_LONG).show();
8         } catch (FileNotFoundException e) {
9             return;
10        }
11        catch (IOException e){
12            return ;
13        }
14 }
```

openFileOutput()方法的第一参数用于指定文件名称，不能包含路径分隔符“/” ，如果文件不存在，Android

创建的文件保存在/data/data//files 目录，如： /data/data/cn.itcast.action/files/itcast.txt ，通过点击 Eclipse

View”-“Other” ，在对话窗口中展开 android 文件夹，选择下面的 File Explorer 视图，然后在 File Explorer 视

目录就可以看到该文件。

openFileOutput()方法的第二参数用于指定操作模式，有四种模式，分别为：

Context.MODE_PRIVATE = 0

Context.MODE_APPEND = 32768

Context.MODE_WORLD_READABLE = 1

Context.MODE_WORLD_WRITEABLE = 2

Context.MODE_PRIVATE：为默认操作模式，代表该文件是私有数据，只能被应用本身访问，在该模式下，写入的

如果想把新写入的内容追加到原文件中。可以使用 Context.MODE_APPEND

Context.MODE_APPEND：模式会检查文件是否存在，存在就往文件追加内容，否则就创建新文件。

Context.MODE_WORLD_READABLE 和 Context.MODE_WORLD_WRITEABLE 用来控制其他应用是否有权限读

MODE_WORLD_READABLE：表示当前文件可以被其他应用读取;

MODE_WORLD_WRITEABLE：表示当前文件可以被其他应用写入。

如果希望文件被其他应用读和写，可以传入： openFileOutput("itcast.txt", Context.MODE_WORLD_READAB

Context.MODE_WORLD_WRITEABLE); android 有一套自己的安全模型，当应用程序(.apk)在安装时系统就会分

应用要去访问其他资源比如文件的时候，就需要 userid 匹配。默认情况下，任何应用创建的文件，sharedpreferen

有的(位于/data/data//files)，其他程序无法访问。

除非在创建时指定了 Context.MODE_WORLD_READABLE 或者 Context.MODE_WORLD_WRITEABLE ,只有这

读取文件示例：

```
1  public void load()
2  {
3      try {
4          FileInputStream inStream=this.openFileInput("a.txt");
5          ByteArrayOutputStream stream=new ByteArrayOutputStream();
6          byte[] buffer=new byte[1024];
7          int length=-1;
8  while((length=inStream.read(buffer))!=-1)   {
9              stream.write(buffer,0,length);
10         }
11         stream.close();
12         inStream.close();
13         text.setText(stream.toString());
14         Toast.makeText(MyActivity.this,"Loaded",Toast.LENGTH_LONG).show();
15     } catch (FileNotFoundException e) {
16         e.printStackTrace();
17     }
18     catch (IOException e){
19         return ;
20     }
21}
```

对于私有文件只能被创建该文件的应用访问，如果希望文件能被其他应用读和写，可以在创建文件时，指定 Context.MODE_WORLD_READABLE 和 Context.MODE_WORLD_WRITEABLE 权限。

Activity 还提供了 getCacheDir()和 getFilesDir()方法： getCacheDir()方法用于获取/data/data//cache 目录 g /data/data//files 目录。

把文件存入 SDCard：

使用 Activity 的 openFileOutput()方法保存文件，文件是存放在手机空间上，一般手机的存储空间不是很大，存 放像视频这样的大文件，是不可行的。对于像视频这样的大文件，我们可以把它存放在 SDCard。

SDCard 是干什么的?你可以把它看作是移动硬盘或 U 盘。 在模拟器中使用 SDCard，你需要先创建一张 SDCard 只是镜像文件)。

创建 SDCard 可以在 Eclipse 创建模拟器时随同创建，也可以使用 DOS 命令进行创建，如下： 在 Dos 窗口中进入 tools 目录，输入以下命令创建一张容量为 2G 的 SDCard，文件后缀可以随便取，建议使用.img： mksdcard 2 D:\AndroidTool\sdcard.img 在程序中访问 SDCard，你需要申请访问 SDCard 的权限。

在 AndroidManifest.xml 中加入访问 SDCard 的权限如下:

```
1 <!-- 在 SDCard 中创建与删除文件权限 -->
2    <uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
3
4    <!-- 往 SDCard 写入数据权限 -->
5    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

要往 SDCard 存放文件，程序必须先判断手机是否装有 SDCard，并且可以进行读写。

注意：访问 SDCard 必须在 AndroidManifest.xml 中加入访问 SDCard 的权限。

```
1if(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)){
2File sdCardDir = Environment.getExternalStorageDirectory();//获取 SDCard 目录
3File saveFile = new File(sdCardDir, “a.txt”);
4FileOutputStream outStream = new FileOutputStream(saveFile);
5outStream.write("test".getBytes());
6outStream.close();
7}
```

Environment.getExternalStorageState()方法用于获取 SDCard 的状态，如果手机装有 SDCard，并且可以进行
等于 Environment.MEDIA_MOUNTED。

Environment.getExternalStorageDirectory()方法用于获取 SDCard 的目录，当然要获取 SDCard 的目录，你也

```
File sdCardDir = new File("/sdcard"); //获取 SDCard 目录
File saveFile = new File(sdCardDir, "itcast.txt");
//上面两句代码可以合成一句：
File saveFile = new File("/sdcard/a.txt");
FileOutputStream outStream = new FileOutputStream(saveFile);
```

outStream.write("test".getBytes());

outStream.close();

SQLite 数据库存储数据

SQLite 是轻量级嵌入式数据库引擎，它支持 SQL 语言，并且只利用很少的内存就有很好的性能。此外它还是开源
许多开源项目((Mozilla, PHP, Python)都使用了 SQLite.SQLite 由以下几个组件组成：SQL 编译器、内核、后端
用虚拟机和虚拟数据库引擎(VDBE)，使调试、修改和扩展 SQLite 的内核变得更加方便。

特点：

面向资源有限的设备，

没有服务器进程,

所有数据存放在同一文件中跨平台，

可自由复制。

SQLite 内部结构：

SQLite 基本上符合 SQL-92 标准 ,和其他的主要 SQL 数据库没什么区别。它的优点就是高效 ,Android 运行时环

SQLite 和其他数据库最大的不同就是对数据类型的支持，创建一个表时，可以在 CREATE TABLE 语句中指定某
把任何数据类型放入任何列中。当某个值插入数据库时，SQLite 将检查它的类型。如果该类型与关联的列不匹配

转换成该列的类型。如果不能转换，则该值将作为其本身具有的类型存储。比如可以把一个字符串(String)放入 IN

为“弱类型”(manifest typing.)。 此外，SQLite 不支持一些标准的 SQL 功能，特别是外键约束(FOREIGN KE

transcaction 和 RIGHT OUTER JOIN 和 FULL OUTER JOIN, 还有一些 ALTER TABLE 功能。 除了上述功能外

SQL 系统，拥有完整的触发器，交易等等。

Android 集成了 SQLite 数据库 Android 在运行时(run-time)集成了 SQLite，所以每个 Android 应用程序都

对于熟悉 SQL 的开发人员来时，在 Android 开发中使用 SQLite 相当简单。但是，由于 JDBC 会消耗太多的系

手机这种内存受限设备来说并不合适。因此，Android 提供了一些新的 API 来使用 SQLite 数据库，Android 推

这些 API。

数据库存储在 data/< 项目文件夹 >/databases/ 下。Android 开发中使用 SQLite 数据库 Activites 可以通过

Service 访问一个数据库。

下面会详细讲解如果创建数据库 ,添加数据和查询数据库。 创建数据库 Android 不自动提供数据库。在 Android

必须自己创建数据库，然后创建表、索引，填充数据。

Android 提供了 SQLiteOpenHelper 帮助你创建一个数据库，你只要继承 SQLiteOpenHelper 类，就可以轻松

SQLiteOpenHelper 类根据开发应用程序的需要，封装了创建和更新数据库使用的逻辑。

SQLiteOpenHelper 的子类，至少需要实现三个方法：

1 构造函数，调用父类 SQLiteOpenHelper 的构造函数。这个方法需要四个参数：上下文环境(例如，一个 Acti

选的游标工厂(通常是 Null)，一个代表你正在使用的数据库模型版本的整数。

2 onCreate()方法，它需要一个 SQLiteDatabase 对象作为参数，根据需要对这个对象填充表和初始化数据。

3 onUpgrage() 方法，它需要三个参数，一个 SQLiteDatabase 对象，一个旧的版本号和一个新的版本号，这样

数据库从旧的模型转变到新的模型。

下面示例代码展示了如何继承 SQLiteOpenHelper 创建数据库：

```
1  public class DatabaseHelper extends SQLiteOpenHelper {
2    DatabaseHelper(Context context, String name, CursorFactory cursorFactory, int version)
3   {
4      super(context, name, cursorFactory, version);
5       }
6
7       @Override
8       public void onCreate(SQLiteDatabase db) {
9           // TODO 创建数据库后，对数据库的操作
10     }
11
12     @Override
13 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
14          // TODO 更改数据库版本的操作
15     }
16
17 @Override
18 public void onOpen(SQLiteDatabase db) {
```

```
19          super.onOpen(db);
20          // TODO 每次成功打开数据库后首先被执行
21      }
22 }
```

接下来讨论具体如何创建表、插入数据、删除表等等。调用 getReadableDatabase() 或 getWriteableDatabas
SQLiteDatabase 实例，具体调用那个方法，取决于你是否需要改变数据库的内容：

```
1db=(new DatabaseHelper(getContext())).getWritableDatabase();
2       return (db == null) ? false : true;
```

上面这段代码会返回一个 SQLiteDatabase 类的实例，使用这个对象，你就可以查询或者修改数据库。 当你完成
的 Activity 已经关闭)，需要调用 SQLiteDatabase 的 Close() 方法来释放掉数据库连接。 创建表和索引 为了
SQLiteDatabase 的 execSQL() 方法来执行 DDL 语句。如果没有异常，这个方法没有返回值。

例如，你可以执行如下代码：

```
1db.execSQL("CREATE TABLE mytable (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, value R
```

这条语句会创建一个名为 mytable 的表，表有一个列名为 _id，并且是主键，这列的值是会自动增长的整数(例如
会给这列自动赋值)，另外还有两列：title(字符) 和 value(浮点数)。 SQLite 会自动为主键列创建索引。 通
库时创建了表和索引。

如果你不需要改变表的 schema ,不需要删除表和索引 . 删除表和索引 ,需要使用 execSQL() 方法调用 DROP IN
句。 给表添加数据 上面的代码，已经创建了数据库和表，现在需要给表添加数据。有两种方法可以给表添加数据

像上面创建表一样，你可以使用 execSQL() 方法执行 INSERT, UPDATE, DELETE 等语句来更新表的数据。exec

返回结果的 SQL 语句。

例如： db.execSQL("INSERT INTO widgets (name, inventory)"+ "VALUES ('Sprocket', 5)");

另一种方法是使用 SQLiteDatabase 对象的 insert(), update(), delete() 方法。这些方法把 SQL 语句的一部分

示例如下：

```
ContentValues cv=new ContentValues();

cv.put(Constants.TITLE, "example title");

cv.put(Constants.VALUE, SensorManager.GRAVITY_DEATH_STAR_I);

db.insert("mytable", getNullColumnHack(), cv);
```

update()方法有四个参数，分别是表名，表示列名和值的 ContentValues 对象，可选的 WHERE 条件和可选的

串，这些字符串会替换 WHERE 条件中的“?”标记。

update() 根据条件，更新指定列的值，所以用 execSQL() 方法可以达到同样的目的。 WHERE 条件和其参数和

似。

例如：

```
String[] parms=new String[] {"this is a string"};

db.update("widgets", replacements, "name=?", parms);
```

delete() 方法的使用和 update() 类似 ,使用表名 ,可选的 WHERE 条件和相应的填充 WHERE 条件的字符串。

UPDATE, DELETE，有两种方法使用 SELECT 从 SQLite 数据库检索数据。

1 .使用 rawQuery() 直接调用 SELECT 语句; 使用 query() 方法构建一个查询。

Raw Queries 正如 API 名字，rawQuery() 是最简单的解决方法。通过这个方法你就可以调用 SQL SELECT 语

例如： Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND name='my

在上面例子中，我们查询 SQLite 系统表(sqlite_master)检查 table 表是否存在。返回值是一个 cursor 对象，

询结果。 如果查询是动态的，使用这个方法就会非常复杂。

例如，当你需要查询的列在程序编译的时候不能确定，这时候使用 query() 方法会方便很多。

Regular Queries query() 方法用 SELECT 语句段构建查询。SELECT 语句内容作为 query() 方法的参数，比如

字段名，WHERE 条件，包含可选的位置参数，去替代 WHERE 条件中位置参数的值，GROUP BY 条件，HAV

他参数可以是 null。所以，以前的代码段可以可写成：

```
1String[] columns={"ID", "inventory"};
2 String[] parms={"snicklefritz"};
3 Cursor result=db.query("widgets", columns, "name=?",parms, null, null, null);
```

使用游标

不管你如何执行查询，都会返回一个 Cursor，这是 Android 的 SQLite 数据库游标，

使用游标，你可以：

通过使用 getCount() 方法得到结果集中有多少记录;

通过 moveToFirst(), moveToNext(), 和 isAfterLast() 方法遍历所有记录;

通过 getColumnNames() 得到字段名;

通过 getColumnIndex() 转换成字段号;

通过 getString()，getInt() 等方法得到给定字段当前记录的值;

通过 requery() 方法重新执行查询得到游标;

通过 close() 方法释放游标资源;

例如，下面代码遍历 mytable 表：

```
1  Cursor result=db.rawQuery("SELECT ID, name, inventory FROM mytable");
2  result.moveToFirst();
3      while (!result.isAfterLast()) {
4          int id=result.getInt(0);
5          String name=result.getString(1);
6          int inventory=result.getInt(2);
7          // do something useful with these
8          result.moveToNext();
9       }
10 result.close();
```

在 Android 中使用 SQLite 数据库管理工具 在其他数据库上作开发，一般都使用工具来检查和处理数据库的内
的 API。

使用 Android 模拟器，有两种可供选择的方法来管理数据库。

首先，模拟器绑定了 sqlite3 控制台程序，可以使用 adb shell 命令来调用他。只要你进入了模拟器的 shell，在
命令就可以了。

数据库文件一般存放在： /data/data/your.app.package/databases/your-db-name 如果你喜欢使用更友好的
到你的开发机上，使用 SQLite-aware 客户端来操作它。这样的话，你在一个数据库的拷贝上操作，如果你想要
你需要把数据库备份回去。

把数据库从设备上考出来，你可以使用 adb pull 命令(或者在 IDE 上做相应操作)。

存储一个修改过的数据库到设备上，使用 adb push 命令。 一个最方便的 SQLite 客户端是 FireFox SQLite M
有平台使用。

下图是 SQLite Manager 工具：

如果你想要开发 Android 应用程序，一定需要在 Android 上存储数据，使用 SQLite 数据库是一种非常好的选

使用 ContentProvider 存储数据

Android 这个系统和其他的操作系统还不太一样，我们需要记住的是，数据在 Android 当中是私有的，当然这些
数据以及一些其他类型的数据。那这个时候有读者就会提出问题，难道两个程序之间就没有办法对于数据进行交换

不会让这种情况发生的。解决这个问题主要靠 ContentProvider。一个 Content Provider 类实现了一组标准的方
应用保存或读取此 Content Provider 的各种数据类型。也就是说，一个程序可以通过实现一个 Content Provide
暴露出去。外界根本看不到，也不用看到这个应用暴露的数据在应用当中是如何存储的，或者是用数据库存储还是
上获得，这些一切都不重要，重要的是外界可以通过这一套标准及统一的接口和程序里的数据打交道，可以读取程
序的数据，当然，中间也会涉及一些权限的问题。

一个程序可以通过实现一个 ContentProvider 的抽象接口将自己的数据完全暴露出去，而且 ContentProviders 是
将数据暴露，也就是说 ContentProvider 就像一个“数据库”。那么外界获取其提供的数据，也就应该与从数据
样，只不过是采用 URI 来表示外界需要访问的“数据库”。

Content Provider 提供了一种多应用间数据共享的方式，比如：联系人信息可以被多个应用程序访问。

Content Provider 是个实现了一组用于提供其他应用程序存取数据的标准方法的类。应用程序可以在 Content Pr
询数据 修改数据 添加数据 删除数据

标准的 Content Provider: Android 提供了一些已经在系统中实现的标准 Content Provider，比如联系人信息，
Content Provider 来访问设备上存储的联系人信息，图片等等。

查询记录:

在 Content Provider 中使用的查询字符串有别于标准的 SQL 查询。很多诸如 select, add, delete, modify 等操作
来进行，这种 URI 由 3 个部分组成， “content://”，代表数据的路径，和一个可选的标识数据的 ID。

以下是一些示例 URI:

content://media/internal/images 这个 URI 将返回设备上存储的所有图片

content://contacts/people/ 这个 URI 将返回设备上的所有联系人信息

content://contacts/people/45 这个 URI 返回单个结果(联系人信息中 ID 为 45 的联系人记录)

尽管这种查询字符串格式很常见，但是它看起来还是有点令人迷惑。为此，Android 提供一系列的帮助类(在 and

包含了很多以类变量形式给出的查询字符串，这种方式更容易让我们理解一点，参见下例:

MediaStore.Images.Media.INTERNAL_CONTENT_URI Contacts.People.CONTENT_URI

因此，如上面 content://contacts/people/45 这个 URI 就可以写成如下形式：

Uri person = ContentUris.withAppendedId(People.CONTENT_URI, 45);

然后执行数据查询: Cursor cur = managedQuery(person, null, null, null);

这个查询返回一个包含所有数据字段的游标，我们可以通过迭代这个游标来获取所有的数据：

```
1  package com.wissen.testApp;
2  public class ContentProviderDemo extends Activity {
3      @Override
4      public void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          setContentView(R.layout.main);
7         displayRecords();
8      }
9      private void displayRecords() {
```

```
10      //该数组中包含了所有要返回的字段
11    String columns[] = new String[] { People.NAME, People.NUMBER };
12      Uri mContacts = People.CONTENT_URI;
13      Cursor cur = managedQuery(
14          mContacts,
15          columns,  // 要返回的数据字段
16      null,          // WHERE 子句
17      null,          // WHERE 子句的参数
18      null         // Order-by 子句
19    );
20      if (cur.moveToFirst()) {
21          String name = null;
22          String phoneNo = null;
23          do {
24            // 获取字段的值
25        name = cur.getString(cur.getColumnIndex(People.NAME));
26            phoneNo = cur.getString(cur.getColumnIndex(People.NUMBER));
27            Toast.makeText(this, name + "  " + phoneNo, Toast.LENGTH_LONG).show();
28        } while (cur.moveToNext());
29      }
30    }
31}
```

上例示范了一个如何依次读取联系人信息表中的指定数据列 name 和 number。

修改记录:

我们可以使用 ContentResolver.update()方法来修改数据，我们来写一个修改数据的方法:

```
1private void updateRecord(int recNo, String name) {
2        Uri uri = ContentUris.withAppendedId(People.CONTENT_URI, recNo);
3        ContentValues values = new ContentValues();
4        values.put(People.NAME, name);
5        getContentResolver().update(uri, values, null, null);
6    }
```

现在你可以调用上面的方法来更新指定记录： updateRecord(10,“ XYZ”); //更改第 10 条记录的 name 字段值

添加记录:

要增加记录，我们可以调用 ContentResolver.insert()方法，该方法接受一个要增加的记录的目标 URI，以及一个
象，调用后的返回值是新记录的 URI，包含记录号。

上面的例子中我们都是基于联系人信息簿这个标准的 Content Provider ,现在我们继续来创建一个 insertRecord(
进行数据的添加：

```
1   private void insertRecords(String name, String phoneNo) {
2      ContentValues values = new ContentValues();
3      values.put(People.NAME, name);
```

```
4      Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
5      Log.d(“ ANDROID”, uri.toString());
6      Uri numberUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);
7      values.clear();
8      values.put(Contacts.Phones.TYPE, People.Phones.TYPE_MOBILE);
9      values.put(People.NUMBER, phoneNo);
10     getContentResolver().insert(numberUri, values);
11}
```

这样我们就可以调用 insertRecords(name, phoneNo)的方式来向联系人信息簿中添加联系人姓名和电话号码。

删除记录:

Content Provider 中的 getContextResolver.delete()方法可以用来删除记录。

下面的记录用来删除设备上所有的联系人信息：

```
1private void deleteRecords() {
2Uri uri = People.CONTENT_URI;
3getContentResolver().delete(uri, null, null);
4}
```

你也可以指定 WHERE 条件语句来删除特定的记录：

getContentResolver().delete(uri, “NAME=” + “ ‘XYZ XYZ’ ”，null);

这将会删除 name 为 ‘XYZ XYZ’ 的记录。

创建 Content Provider:

至此我们已经知道如何使用 Content Provider 了，现在让我们来看下如何自己创建一个 Content Provider。

要创建我们自己的 Content Provider 的话，我们需要遵循以下几步：

1. 创建一个继承了 ContentProvider 父类的类

2. 定义一个名为 CONTENT_URI，并且是 public static final 的 Uri 类型的类变量，你必须为其指定一个唯一的字
类的全名称，

如: public static final Uri CONTENT_URI = Uri.parse(“content://com.google.android.MyContentProvide

3. 创建你的数据存储系统。大多数 Content Provider 使用 Android 文件系统或 SQLite 数据库来保持数据，但是
方式来存储。

4. 定义你要返回给客户端的数据列名。如果你正在使用 Android 数据库，则数据列的使用方式就和你以往所熟悉
你必须为其定义一个叫_id 的列，它用来表示每条记录的唯一性。

5. 如果你要存储字节型数据，比如位图文件等，那保存该数据的数据列其实是一个表示实际保存文件的 URI 字符
应的文件数据，处理这种数据类型的 Content Provider 需要实现一个名为_data 的字段，_data 字段列出了该文件
精确路径。这个字段不仅是供客户端使用，而且也可以供 ContentResolver 使用。客户端可以调用 ContentReso
方法来处理该 URI 指向的文件资源，如果是 ContentResolver 本身的话，由于其持有的权限比客户端要高，所以

6. 声明 public static String 型的变量，用于指定要从游标处返回的数据列。

7. 查询返回一个 Cursor 类型的对象。所有执行写操作的方法如 insert(), update() 以及 delete()都将被监听。我

ContentResover().notifyChange()方法来通知监听器关于数据更新的信息。

8. 在 AndroidMenifest.xml 中使用标签来设置 Content Provider。

9. 如果你要处理的数据类型是一种比较新的类型，你就必须先定义一个新的 MIME 类型，以供 ContentProvide

MIME 类型有两种形式:

一种是为指定的单个记录的，还有一种是为多条记录的。这里给出一种常用的格式：

vnd.android.cursor.item/vnd.yourcompanyname.contenttype (单个记录的 MIME 类型) 比如，一个请求列车

content://com.example.transportationprovider/trains/122 可能就会返回 typevnd.android.cursor.item/vn

MIME 类型。

vnd.android.cursor.dir/vnd.yourcompanyname.contenttype (多个记录的 MIME 类型) 比如，一个请求所有列

content://com.example.transportationprovider/trains 可能就会返回 vnd.android.cursor.dir/vnd.example.

下列代码将创建一个 Content Provider，它仅仅是存储用户名称并显示所有的用户名称(使用 SQLLite 数据库存

```
1  package com.wissen.testApp;
2  public class MyUsers {
3      public static final String AUTHORITY  =  “com.wissen.MyContentProvider”;
4      // BaseColumn 类中已经包含了 _id 字段
5     public static final class User implements BaseColumns {
```

```
6         public static final Uri CONTENT_URI  = Uri.parse(“ content://com.wissen.MyContentProvider”);
7         // 表数据列
8       public static final String  USER_NAME  =  “USER_NAME”;
9      }
10}
```

上面的类中定义了 Content Provider 的 CONTENT_URI，以及数据列。下面我们将定义基于上面的类来定义实际

```
1  package com.wissen.testApp.android;
2  public class MyContentProvider extends ContentProvider {
3      private SQLiteDatabase     sqlDB;
4      private DatabaseHelper    dbHelper;
5      private static final String  DATABASE_NAME     =  “Users.db”;
6      private static final int        DATABASE_VERSION         = 1;
7      private static final String TABLE_NAME   =  “User”;
8      private static final String TAG =  “MyContentProvider”;
9      private static class DatabaseHelper extends SQLiteOpenHelper {
10        DatabaseHelper(Context context) {
11            super(context, DATABASE_NAME, null, DATABASE_VERSION);
12        }
13        @Override
14        public void onCreate(SQLiteDatabase db) {
15            //创建用于存储数据的表
```

```
16      db.execSQL(" Create table "  + TABLE_NAME +  "( _id INTEGER PRIMARY KEY AUTOINCREMEN
17      }
18      @Override
19      public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
20          db.execSQL(" DROP TABLE IF EXISTS "  + TABLE_NAME);
21          onCreate(db);
22      }
23  }
24  @Override
25  public int delete(Uri uri, String s, String[] as) {
26      return 0;
27  }
28  @Override
29  public String getType(Uri uri) {
30      return null;
31  }
32  @Override
33  public Uri insert(Uri uri, ContentValues contentvalues) {
34      sqlDB = dbHelper.getWritableDatabase();
35      long rowId = sqlDB.insert(TABLE_NAME,  "" , contentvalues);
36      if (rowId > 0) {
37          Uri rowUri = ContentUris.appendId(MyUsers.User.CONTENT_URI.buildUpon(), rowId).build();
```

```
38          getContext().getContentResolver().notifyChange(rowUri, null);
39          return rowUri;
40      }
41      throw new SQLException(" Failed to insert row into "  + uri);
42  }
43  @Override
44  public boolean onCreate() {
45      dbHelper = new DatabaseHelper(getContext());
46      return (dbHelper == null) ? false : true;
47  }
48  @Override
49  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortO
50      SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
51      SQLiteDatabase db = dbHelper.getReadableDatabase();
52      qb.setTables(TABLE_NAME);
53      Cursor c = qb.query(db, projection, selection, null, null, null, sortOrder);
54      c.setNotificationUri(getContext().getContentResolver(), uri);
55      return c;
56  }
57  @Override
58  public int update(Uri uri, ContentValues contentvalues, String s, String[] as) {
59      return 0;
```

```
60    }
61}
```

一个名为 MyContentProvider 的 Content Provider 创建完成了，它用于从 Sqlite 数据库中添加和读取记录。

Content Provider 的入口需要在 AndroidManifest.xml 中配置:

之后，让我们来使用这个定义好的 Content Provider:

```
1  package com.wissen.testApp;
2  public class MyContentDemo extends Activity {
3      @Override
4      protected void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          insertRecord(“ MyUser”);
7          displayRecords();
8      }
9
10     private void insertRecord(String userName) {
11         ContentValues values = new ContentValues();
12         values.put(MyUsers.User.USER_NAME, userName);
13         getContentResolver().insert(MyUsers.User.CONTENT_URI, values);
14     }
15     private void displayRecords() {
```

```
16        String columns[] = new String[] { MyUsers.User._ID, MyUsers.User.USER_NAME };
17        Uri myUri = MyUsers.User.CONTENT_URI;
18        Cursor cur = managedQuery(myUri, columns,null, null, null );
19        if (cur.moveToFirst()) {
20            String id = null;
21            String userName = null;
22            do {
23                id = cur.getString(cur.getColumnIndex(MyUsers.User._ID));
24                userName = cur.getString(cur.getColumnIndex(MyUsers.User.USER_NAME));
25                Toast.makeText(this, id + " " + userName, Toast.LENGTH_LONG).show();
26            } while (cur.moveToNext());
27        }
28    }
29}
```

上面的类将先向数据库中添加一条用户数据，然后显示数据库中所有的用户数据。

网络存储数据

前面介绍的几种存储都是将数据存储在本地设备上，除此之外，还有一种存储(获取)数据的方式，通过网络来实现

我们可以调用 WebService 返回的数据或是解析 HTTP 协议实现网络数据交互。

具体需要熟悉 java.net.*，Android.net.*这两个包的内容，在这就不赘述了，请大家参阅相关文档。

下面是一个通过地区名称查询该地区的天气预报 ,以 POST 发送的方式发送请求到 webservicex.net 站点 ,访问 We

站点上提供查询天气预报的服务。

代码如下：

```
1  package com.android.weather;
2  import java.util.ArrayList;
3  import java.util.List;
4  import org.apache.http.HttpResponse;
5  import org.apache.http.NameValuePair;
6  import org.apache.http.client.entity.UrlEncodedFormEntity;
7  import org.apache.http.client.methods.HttpPost;
8  import org.apache.http.impl.client.DefaultHttpClient;
9  import org.apache.http.message.BasicNameValuePair;
10import org.apache.http.protocol.HTTP;
11import org.apache.http.util.EntityUtils;
12import android.app.Activity;
13import android.os.Bundle;
14public class MyAndroidWeatherActivity extends Activity {
15    //定义需要获取的内容来源地址
16    private static final String SERVER_URL =
17        "http://www.webservicex.net/WeatherForecast.asmx/GetWeatherByPlaceName";
18
```

```
19
20  /** Called when the activity is first created. */
21  @Override
22  public void onCreate(Bundle savedInstanceState) {
23      super.onCreate(savedInstanceState);
24      setContentView(R.layout.main);
25
26      HttpPost request = new HttpPost(SERVER_URL); //根据内容来源地址创建一个 Http 请求
27      // 添加一个变量
28      List<NameValuePair> params = new ArrayList<NameValuePair>();
29      // 设置一个地区名称
30      params.add(new BasicNameValuePair("PlaceName", "NewYork"));  //添加必须的参数
31
32
33      try {
34          //设置参数的编码
35          request.setEntity(new UrlEncodedFormEntity(params, HTTP.UTF_8));
36          //发送请求并获取反馈
37          HttpResponse httpResponse = new DefaultHttpClient().execute(request);
38
39          // 解析返回的内容
40          if(httpResponse.getStatusLine().getStatusCode() != 404){
```

```
41              String result = EntityUtils.toString(httpResponse.getEntity());
42              System.out.println(result);
43          }
44       } catch (Exception e) {
45          e.printStackTrace();
46      }
47   }
48}
```

别忘记了在配置文件中设置访问网络权限：

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

17. Android 有哪些存储数据的方式。

18. SharedPrefrence 源码和问题点；

- 1.储存于硬盘上的 xml 键值对，数据多了会有性能问题
- 2.ContextImpl 记录着 SharedPreferences 的重要数据，文件路径和实例的键值对
- 3.在 xml 文件全部内加载到内存中之前，读取操作是阻塞的，在 xml 文件全部内加载到内存中之后，是直接读取内存中的数据
- 4.apply 因为是异步的没有返回值, commit 是同步的有返回值能知道修改是否提交成功
- 5.多并发的提交 commit 时，需等待正在处理的 commit 数据更新到磁盘文件后才会继续往下执行，从而降低效率; 而 apply 只是原子更新到内存，后调用 apply 函数会直接覆盖前面内存数据，从一定程度上提高很多效率。 3.edit()每次都是创建新的 EditorImpl 对象.
- 6.博客推荐：全面剖析 SharedPreferences

18. sqlite 相关

一.SQLite 的介绍

1.SQLite 简介

SQLite 是一款轻型的数据库，是遵守 ACID 的关联式数据库管理系统，它的设计目标是嵌入 式的，而且目前已经在很多嵌入式产品中使用了它，它占用资源非常的低，在嵌入式设备中，可能只需要几百 K 的内存就够了。它能够支持 Windows/Linux/Unix 等等主流的操作系统，同时能够跟很多程序语言相结合，比如 Tcl、PHP、Java、C++、.Net 等，还有 ODBC 接口，同样比起 Mysql、PostgreSQL 这两款开源世界著名的数据库管理系统来讲，它的处理速度比他们都快。

2.SQLite 的特点：

- 轻量级

SQLite 和 C/S 模式的数据库软件不同，它是进程内的数据库引擎，因此不存在数据库的客户端和服务器。使用 SQLite 一般只需要带上它的一个动态 库，就可以享受它的全部功能。而且那个动态库的尺寸也挺小，以版本 3.6.11 为例，Windows 下 487KB、Linux 下 347KB。

- 不需要"安装"

SQLite 的核心引擎本身不依赖第三方的软件，使用它也不需要"安装"。有点类似那种绿色软件。

- 单一文件

数据库中所有的信息（比如表、视图等）都包含在一个文件内。这个文件可以自由复制到其它目录或其它机器上。

- 跨平台/可移植性

除了主流操作系统 windows，linux 之后，SQLite 还支持其它一些不常用的操作系统。

- 弱类型的字段

同一列中的数据可以是不同类型

- 开源

这个相信大家都懂的！！！！！！！！！！！

3.SQLite 数据类型

一般数据采用的固定的静态数据类型，而 SQLite 采用的是动态数据类型，会根据存入值自动判断。SQLite 具有以下五种常用的数据类型：

NULL：这个值为空值

VARCHAR(n)：长度不固定且其最大长度为 n 的字串，n 不能超过 4000。

CHAR(n)：长度固定为 n 的字串，n 不能超过 254。

INTEGER：值被标识为整数,依据值的大小可以依次被存储为 1,2,3,4,5,6,7,8.

REAL：所有值都是浮动的数值,被存储为 8 字节的 IEEE 浮动标记序号.

TEXT：值为文本字符串,使用数据库编码存储(TUTF-8, UTF-16BE or UTF-16-LE).

BLOB：值是 BLOB 数据块，以输入的数据格式进行存储。如何输入就如何存储,不改 变格式。

DATA ：包含了 年份、月份、日期。

TIME： 包含了 小时、分钟、秒。

相信学过数据库的童鞋对这些数据类型都不陌生的!!!!!!!!!!

二.SQLiteDatabase 的介绍

Android 提供了创建和是用 SQLite 数据库的 API。SQLiteDatabase 代表一个数据库对象，提供了操作数据库的一些方法。在 Android 的 SDK 目录下有 sqlite3 工具，我们可以利用它创建数据库、创建表和执行一些 SQL 语句。下面是 SQLiteDatabase 的常用方法。

SQLiteDatabase 的常用方法

方法名称	方法表示含义
openOrCreateDatabase(String path,SQLiteDatabase.CursorFactory factory)	打开或创建数据库
insert(String table,String nullColumnHack,ContentValues values)	插入一条记录
delete(String table,String whereClause,String[] whereArgs)	删除一条记录
query(String table,String[] columns,String selection,String[] selectionArgs,String groupBy,String having,String orderBy)	查询一条记录
update(String table,ContentValues values,String whereClause,String[] whereArgs)	修改记录
execSQL(String sql)	执行一条 SQL 语句
close()	关闭数据库

Google 公司命名这些方法的名称都是非常形象的。例如 openOrCreateDatabase,我们从字面英文含义就能看出这是个打开或创建数据库的方法。

1、打开或者创建数据库

在 Android 中使用 SQLiteDatabase 的静态方法 openOrCreateDatabase(String path,SQLiteDatabae.CursorFactory factory)打开或者创建一个数据库。它会自动去检测是否存在这个数据库，如果存在则打开，不存在则创建一个数据库；创建成功则返回一个 SQLiteDatabase 对象，否则抛出异常 FileNotFoundException。

下面是创建名为"stu.db"数据库的代码：
openOrCreateDatabase(String path,SQLiteDatabae.CursorFactory factory)
参数 1 数据库创建的路径

参数 2 一般设置为 null 就可以了

```
db=SQLiteDatabase.openOrCreateDatabase("/data/data/com.lingdududu.db/databases
```

```
/stu.db",null);
```

2、创建表

创建一张表的步骤很简单：

- 编写创建表的 SQL 语句
- 调用 SQLiteDatabase 的 execSQL()方法来执行 SQL 语句

下面的代码创建了一张用户表，属性列为：id（主键并且自动增加）、sname（学生姓名）、snumber（学号）

```
private void createTable(SQLiteDatabase
db){
//创建表 SQL 语句
String stu_table="create table
usertable(_id integer primary key
autoincrement,sname text,snumber text)";
//执行 SQL 语句
db.execSQL(stu_table);
}
```

3、插入数据

插入数据有两种方法：

①SQLiteDatabase 的 insert(String table,String nullColumnHack,ContentValues values)方法，

参数 1 表名称，

参数 2 空列的默认值

参数 3 ContentValues 类型的一个封装了列名称和列值的 Map；

②编写插入数据的 SQL 语句，直接调用 SQLiteDatabase 的 execSQL()方法来执行

第一种方法的代码：

```
private void insert(SQLiteDatabase
db){
//实例化常量值
ContentValues cValue
= new ContentValues();
//添加用户名
cValue.put("sname","xiaoming");
```

```
//添加密码
cValue.put("snumber","01005");
//调用 insert()方法插入数据
db.insert("stu_table",null,cValue);
}
```

第二种方法的代码：

```
private void insert(SQLiteDatabase
db){
//插入数据 SQL 语句
String stu_sql="insert into
stu_table(sname,snumber)
values('xiaoming','01005')";
//执行 SQL 语句
db.execSQL(sql);
}
```

4、删除数据

删除数据也有两种方法：

①调用 SQLiteDatabase 的 delete(String table,String whereClause,String[] whereArgs)方法
参数 1 表名称
参数 2 删除条件
参数 3 删除条件值数组

②编写删除 SQL 语句，调用 SQLiteDatabase 的 execSQL()方法来执行删除。

第一种方法的代码：

```
private void delete(SQLiteDatabase db) {
//删除条件
String whereClause = "id=?";
//删除条件参数
String[] whereArgs = {String.valueOf(2)};
//执行删除
db.delete("stu_table",whereClause,whereArgs);
```

```
}
```

第二种方法的代码：

```
private void delete(SQLiteDatabase
db) {
//删除 SQL 语句
String sql = "delete from stu_table
where _id = 6";
//执行 SQL 语句
db.execSQL(sql);
}
```

5、修改数据

修改数据有两种方法：

①调用 SQLiteDatabase 的 update(String table,ContentValues values,String whereClause, String[] whereArgs)方法
参数 1 表名称
参数 2 跟行列 ContentValues 类型的键值对 Key-Value
参数 3 更新条件（where 字句）
参数 4 更新条件数组

②编写更新的 SQL 语句，调用 SQLiteDatabase 的 execSQL 执行更新。

第一种方法的代码：

```
private void update(SQLiteDatabase db) {
//实例化内容值 ContentValues values = new
ContentValues();
//在 values 中添加内容
values.put("snumber","101003");
//修改条件
String whereClause = "id=?";
//修改添加参数
String[] whereArgs={String.valueOf(1)};
//修改
db.update("usertable",values,whereClause,whereArgs);
```

```
}
```

第二种方法的代码：

```
private void update(SQLiteDatabase
db){
//修改 SQL 语句
String sql = "update stu_table set
snumber = 654321 where id = 1";
//执行 SQL
db.execSQL(sql);
}
```

6、查询数据

在 Android 中查询数据是通过 Cursor 类来实现的，当我们使用 SQLiteDatabase.query() 方法时，会得到一个 Cursor 对象，Cursor 指向的就是每一条数据。它提供了很多有关查询的方法，具体方法如下：

public Cursor query(String table,String[] columns,String selection,String[] selectionArgs,String groupBy,String having,String orderBy,String limit);

各个参数的意义说明：

参数 table:表名称

参数 columns:列名称数组

参数 selection:条件字句，相当于 where

参数 selectionArgs:条件字句，参数数组

参数 groupBy:分组列

参数 having:分组条件

参数 orderBy:排序列

参数 limit:分页查询限制

参数 Cursor:返回值，相当于结果集 ResultSet

Cursor 是一个游标接口，提供了遍历查询结果的方法，如移动指针方法 move()，获得列值方法 getString()等.

Cursor 游标常用方法

方法名称	方法描述
getCount()	获得总的数据项数
isFirst()	判断是否第一条记录
isLast()	判断是否最后一条记录
moveToFirst()	移动到第一条记录
moveToLast()	移动到最后一条记录
move(int offset)	移动到指定记录
moveToNext()	移动到下一条记录
moveToPrevious()	移动到上一条记录
getColumnIndexOrThrow(String columnName)	根据列名称获得列索引
getInt(int columnIndex)	获得指定列索引的 int 类型值
getString(int columnIndex)	获得指定列缩影的 String 类型值

下面就是用 Cursor 来查询数据库中的数据，具体代码如下：

```
private void query(SQLiteDatabase db) {
//查询获得游标
Cursor cursor = db.query
("usertable",null,null,null,null,null,null);

//判断游标是否为空
if(cursor.moveToFirst() {
//遍历游标
for(int i=0;i<cursor.getCount();i++){
cursor.move(i);
```

```
//获得 ID
int id = cursor.getInt(0);
//获得用户名
String username=cursor.getString(1);
//获得密码
String password=cursor.getString(2);
//输出用户信息
System.out.println(id+":"+sname+":"+snumber);
}
}
}
```

7、删除指定表

编写插入数据的 SQL 语句，直接调用 SQLiteDatabase 的 execSQL()方法来执行

```
private void drop(SQLiteDatabase
db){
//删除表的 SQL 语句
String sql ="DROP TABLE
stu_table";
//执行 SQL
db.execSQL(sql);
}
```

三. SQLiteOpenHelper

该类是 SQLiteDatabase 一个辅助类。这个类主要生成一 个数据库，并对数据库的版本进行管理。当在程序当中调用这个类的方法 getWritableDatabase()或者 getReadableDatabase()方法的时候，如果当时没有数据，那么 Android 系统就会自动生成一个数据库。 SQLiteOpenHelper 是一个抽象类，我们通常需要继承它，并且实现里面的 3 个函数：

1.onCreate（SQLiteDatabase）

在数据库第一次生成的时候会调用这个方法，也就是说，只有在创建数据库的时候才会调用，当然也有一些其它的情况，一般我们在这个方法里边生成数据库表。

2. onUpgrade（SQLiteDatabase，int，int）

当数据库需要升级的时候，Android 系统会主动的调用这个方法。一般我们在这个方法里边删除数据表，并建立新的数据表，当然是否还需要做其他的操作，完全取决于应用的需求。

3. onOpen（SQLiteDatabase）：

这是当打开数据库时的回调函数，一般在程序中不是很常使用。

写了这么多，改用用实际例子来说明上面的内容了。下面这个操作数据库的实例实现了创建数据库，创建表以及数据库的增删改查的操作。
该实例有两个类：
com.lingdududu.testSQLite 调试类
com.lingdududu.testSQLiteDb 数据库辅助类

SQLiteActivity.java

```
1  package com.lingdududu.testSQLite;
2
3  import com.lingdududu.testSQLiteDb.StuDBHelper;
4
5  import android.app.Activity;
6  import android.content.ContentValues;
7  import android.database.Cursor;
8  import android.database.sqlite.SQLiteDatabase;
9  import android.os.Bundle;
10 import android.view.View;
11 import android.view.View.OnClickListener;
12 import android.widget.Button;
13 /*
14 * @author lingdududu
15 */
16 public class SQLiteActivity extends Activity {
17 /** Called when the activity is first created. */
18 //声明各个按钮
19 private Button createBtn;
20 private Button insertBtn;
21 private Button updateBtn;
22 private Button queryBtn;
23 private Button deleteBtn;
24 private Button ModifyBtn;
25 @Override
26 public void onCreate(Bundle savedInstanceState) {
```

```
27  super.onCreate(savedInstanceState);
28  setContentView(R.layout.main);
29
30  //调用 creatView 方法
31  creatView();
32  //setListener 方法
33  setListener();
34  }
35
36  //通过 findViewById 获得 Button 对象的方法
37  private void creatView(){
38  createBtn = (Button)findViewById(R.id.createDatabase);
39  updateBtn = (Button)findViewById(R.id.updateDatabase);
40  insertBtn = (Button)findViewById(R.id.insert);
41  ModifyBtn = (Button)findViewById(R.id.update);
42  queryBtn = (Button)findViewById(R.id.query);
43  deleteBtn = (Button)findViewById(R.id.delete);
44  }
45
46  //为按钮注册监听的方法
47  private void setListener(){
48  createBtn.setOnClickListener(new CreateListener());
49  updateBtn.setOnClickListener(new UpdateListener());
50  insertBtn.setOnClickListener(new InsertListener());
51  ModifyBtn.setOnClickListener(new ModifyListener());
52  queryBtn.setOnClickListener(new QueryListener());
53  deleteBtn.setOnClickListener(new DeleteListener());
54  }
55
56  //创建数据库的方法
57  class CreateListener implements OnClickListener{
58
59  @Override
60  public void onClick(View v) {
61  //创建 StuDBHelper 对象
62  StuDBHelper dbHelper
```

```
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可读的 SQLiteDatabase 对象
SQLiteDatabase db =dbHelper.getReadableDatabase();
}
}

//更新数据库的方法
class UpdateListener implements OnClickListener{

@Override
public void onClick(View v) {
// 数据库版本的更新,由原来的 1 变为 2
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,2);
SQLiteDatabase db =dbHelper.getReadableDatabase();
}
}

//插入数据的方法
class InsertListener implements OnClickListener{

@Override
public void onClick(View v) {

StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getWritableDatabase();

//生成 ContentValues 对象 //key:列名，value:想插入的值
ContentValues cv = new ContentValues();
//往 ContentValues 对象存放数据，键-值对模式
cv.put("id", 1);
cv.put("sname", "xiaoming");
cv.put("sage", 21);
cv.put("ssex", "male");
```

```
//调用 insert 方法，将数据插入数据库
db.insert("stu_table", null, cv);
//关闭数据库
db.close();
}
}

//查询数据的方法
class QueryListener implements OnClickListener{

@Override
public void onClick(View v) {

StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getReadableDatabase();
//参数 1：表名
//参数 2：要想显示的列
//参数 3：where 子句
//参数 4：where 子句对应的条件值
//参数 5：分组方式
//参数 6：having 条件
//参数 7：排序方式
Cursor cursor =
db.query("stu_table", new String[]{"id","sname","sage","ssex"}, "id=?
", new String[]{"1"}, null, null, null);
while(cursor.moveToNext()){
String name = cursor.getString(cursor.getColumnIndex("sname"));
String age = cursor.getString(cursor.getColumnIndex("sage"));
String sex = cursor.getString(cursor.getColumnIndex("ssex"));
System.out.println("query------->" + "姓名:"+name+" "+"年龄:"+age+" "+"
性别: "+sex);
}
//关闭数据库
db.close();
```

```
}
}

//修改数据的方法
class ModifyListener implements OnClickListener{

@Override
public void onClick(View v) {

StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getWritableDatabase();
ContentValues cv = new ContentValues();
cv.put("sage", "23");
//where 子句 "?"是占位符号，对应后面的"1",
String whereClause="id=?";
String [] whereArgs = {String.valueOf(1)};
//参数 1 是要更新的表名
//参数 2 是一个 ContentValeus 对象
//参数 3 是 where 子句
db.update("stu_table", cv, whereClause, whereArgs);
}
}

//删除数据的方法
class DeleteListener implements OnClickListener{

@Override
public void onClick(View v) {

StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getReadableDatabase();
String whereClauses = "id=?";
```

```
String [] whereArgs = {String.valueOf(2)};
//调用 delete 方法，删除数据
db.delete("stu_table", whereClauses, whereArgs);
}
}
}
```

StuDBHelper.java

```
package com.lingdududu.testSQLiteDb;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class StuDBHelper extends SQLiteOpenHelper {

private static final String TAG = "TestSQLite";
public static final int VERSION = 1;

//必须要有构造函数
public StuDBHelper(Context context, String name, CursorFactory
factory,
int version) {
super(context, name, factory, version);
}

// 当第一次创建数据库的时候，调用该方法
public void onCreate(SQLiteDatabase db) {
String sql = "create table stu_table(id int,sname
varchar(20),sage int,ssex varchar(10))";
//输出创建数据库的日志信息
Log.i(TAG, "create Database------------->");
//execSQL 函数用于执行 SQL 语句
```

```
28  db.execSQL(sql);
29  }
30
31  //当更新数据库的时候执行该方法
32  public void onUpgrade(SQLiteDatabase
33  db, int oldVersion, int newVersion) {
34  //输出更新数据库的日志信息
    Log.i(TAG, "update Database------------->");
    }
    }
```

main.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout
3   xmlns:android="http://schemas.android.com/apk/res/android"
4   android:orientation="vertical"
5   android:layout_width="fill_parent"
6   android:layout_height="fill_parent"
7   >
8   <TextView
9   android:layout_width="fill_parent"
10  android:layout_height="wrap_content"
11  android:text="@string/hello"
12  />
13  <Button
14  android:id="@+id/createDatabase"
15  android:layout_width="fill_parent"
16  android:layout_height="wrap_content"
17  android:text="创建数据库"
18  />
19  <Button
20  android:id="@+id/updateDatabase"
21  android:layout_width="fill_parent"
22  android:layout_height="wrap_content"
23  android:text="更新数据库"
```

```
/>
<Button
android:id="@+id/insert"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="插入数据"
/>
<Button
android:id="@+id/update"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="更新数据"
/>
<Button
android:id="@+id/query"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="查询数据"
/>
<Button
android:id="@+id/delete"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="删除数据"
/>
</LinearLayout>
```

程序运行的效果图：

使用 adb 命令查看数据库：

1.在命令行窗口输入 adb shell 回车，就进入了 Linux 命令行，现在就可以使用 Linux 的命令了。

2.ls 回车，显示所有的东西，其中有个 data。

3.cd data 回车，再 ls 回车，cd data 回车，ls 回车后就会看到很多的 com.....，那就是系统上的应用程序包名，找到你数据库程序的包名，然后进入。

4.进去后在查看所有，会看到有 databases,进入 databases，显示所有就会发现你的数据库名字，这里使用的是"stu_db"。

5.sqlite3 stu_db 回车就进入了你的数据库了，然后".schema"就会看到该应用程序的所有表及建表语句。

6.之后就可以使用标准的 SQL 语句查看刚才生成的数据库及对数据执行增删改查了。
注：ls,cd 等命令都是 linux 的基本命令，不了解的同学可以看看有关这方面的资料。

下面介绍几个在 SQLite 中常用到的 adb 命令：

查看
.database 显示数据库信息；
.tables 显示表名称；
.schema 命令可以查看创建数据表时的 SQL 命令；
.schema table_name 查看创建表 table_name 时的 SQL 的命令；

插入记录
insert into table_name values (field1, field2, field3...);

查询
select * from table_name;查看 table_name 表中所有记录；
select * from table_name where field1='xxxxx'; 查询符合指定条件的记录；

删除
drop table_name;　　删除表；
drop index_name;　　删除索引；
----------查询，插入，删除等操作数据库的语句记得不要漏了;----------

sqlite3 stu_db
sqlite3 stu_db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
.schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE stu_table(id int,sname varchar(20),sage int,ssex varchar(10)); --->创建的表
sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|21|male
sqlite>

插入数据

sqlite> insert into stu_table values(2,'xiaohong',20,'female');

插入的数据记得要和表中的属性一一对应
insert into stu_table values(2,'xiaohong',20,'female');
sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|21|male
2|xiaohong|20|female ----------> 插入的数据
sqlite>

当点击修改数据的按钮时候

sqlite> select * from stu_table;
select * from stu_table;

1|xiaoming|23|male ---------->年龄被修改为 23
2|xiaohong|20|female
sqlite>

当点击删除数据的按钮

sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|23|male　　id=2 的数据已经被删除

20. 如何判断一个 APP 在前台还是后台？

六种方法的区别

方法	判断原理	需要权限	可以判断其他应用位于前台	特点
方法一	RunningTask	否	Android4.0 系列可以,5.0 以上机器不行	5.0 此方法被废弃
方法二	RunningProcess	否	当 App 存在后台常驻的 Service 时失效	无
方法三	ActivityLifecycleCallbacks	否	否	简单有效,代码最少
方法四	UsageStatsManager	是	是	需要用户手动授权
方法五	通过 Android 无障碍功能实现	否	是	需要用户手动授权
方法六	读取/proc 目录下的信息	否	是	当 proc 目录下文件夹过多时,过多的 IO 操作会引起耗时

方法一：通过 RunningTask

原理

当一个 App 处于前台的时候，会处于 RunningTask 的这个栈的栈顶，所以我们可以取出 RunningTask 的栈顶的任务进程，看他与我们的想要判断的 App 的包名是否相同，来达到效果

缺点

getRunningTask 方法在 Android5.0 以上已经被废弃，只会返回自己和系统的一些不敏感的 task，不再返回其他应用的 task，用此方法来判断自身 App 是否处于后台，仍然是有效的，但是无法判断其他应用是否位于前台，因为不再能获取信息

方法二：通过 RunningProcess

原理

通过 runningProcess 获取到一个当前正在运行的进程的 List，我们遍历这个 List 中的每一个进程，判断这个进程的一个 importance 属性是否是前台进程，并且包名是否与我们判断的 APP 的包名一样，如果这两个条件都符合，那么这个 App 就处于前台

缺点：

在聊天类型的 App 中，常常需要常驻后台来不间断的获取服务器的消息，这就需要我们把 Service 设置成 START_STICKY，kill 后会被重启（等待 5 秒左右）来保证 Service 常驻后台。如果 Service 设置了这个属性，这个 App 的进程就会被判断是前台，代码上的表现就是 appProcess.importance 的值永远是 ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND，这样就永远无法判断出到底哪个是前台了。

方法三：通过 ActivityLifecycleCallbacks

原理

AndroidSDK14 在 Application 类里增加了 ActivityLifecycleCallbacks，我们可以通过这个 Callback 拿到 App 所有 Activity 的生命周期回调。

```
public interface ActivityLifecycleCallbacks {
```

```
        void onActivityCreated(Activity activity, Bundle
savedInstanceState);
        void onActivityStarted(Activity activity);
        void onActivityResumed(Activity activity);
        void onActivityPaused(Activity activity);
        void onActivityStopped(Activity activity);
        void onActivitySaveInstanceState(Activity activity, Bundle
outState);
        void onActivityDestroyed(Activity activity);
    }
```

知道这些信息，我们就可以用更官方的办法来解决问题，当然还是利用方案二里的 Activity 生命周期的特性，我们只需要在 Application 的 onCreate()里去注册上述接口，然后由 Activity 回调回来运行状态即可。

可能还有人在纠结，我用 back 键切到后台和用 Home 键切到后台，一样吗？以上方法适用吗？在 Android 应用开发中一般认为 back 键是可以捕获的，而 Home 键是不能捕获的（除非修改 framework）,但是上述方法从 Activity 生命周期着手解决问题，虽然这两种方式的 Activity 生命周期并不相同，但是二者都会执行 onStop（）；所以并不关心到底是触发了哪个键切入后台的。另外,Application 是否被销毁,都不会影响判断的正确性

方法四:通过使用 UsageStatsManager 获取

原理

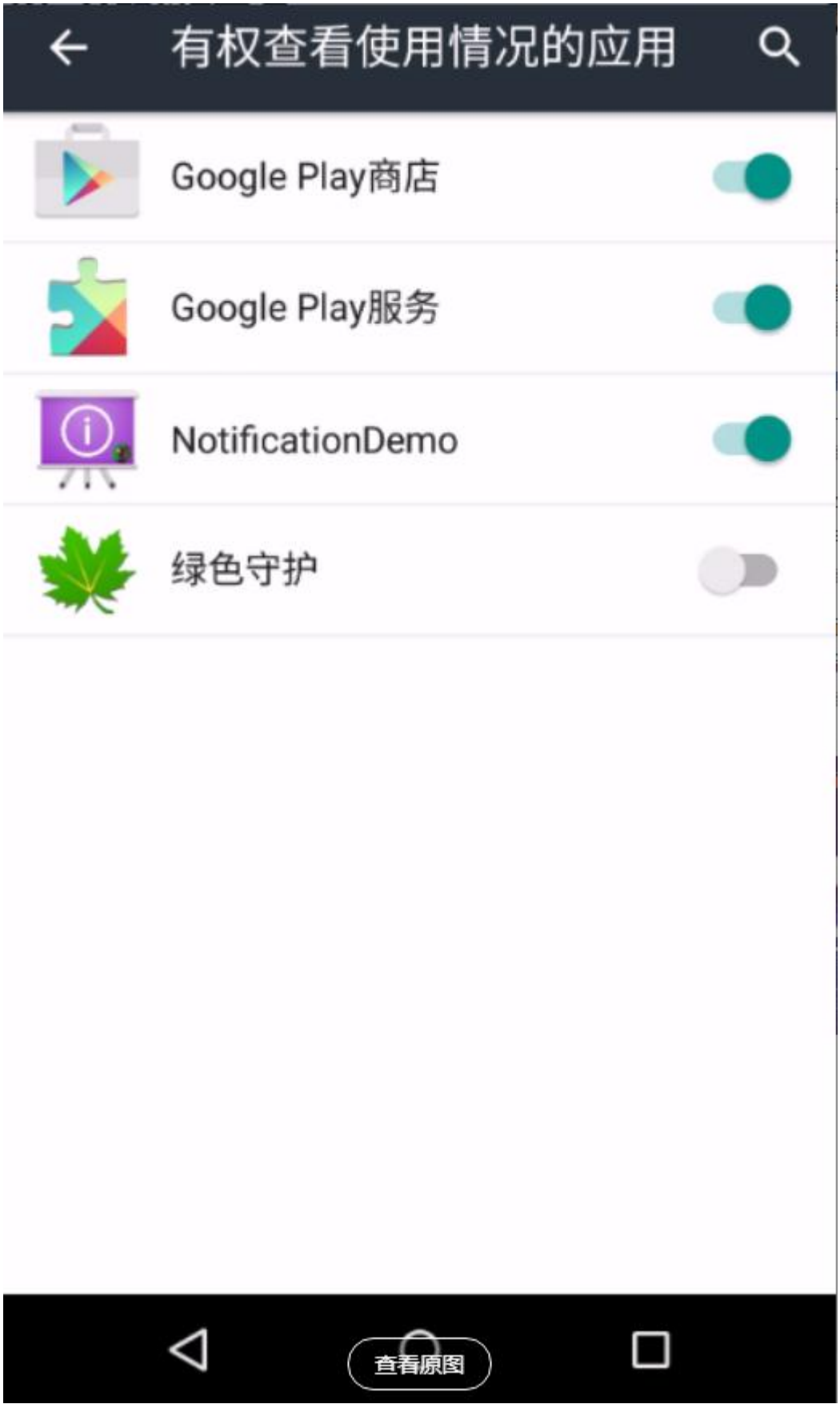
> 通过使用 UsageStatsManager 获取，此方法是 Android5.0 之后提供的新 API，可以获取一个时间段内的应用统计信息，但是必须满足一下要求

使用前提

1.此方法只在 android5.0 以上有效
2.AndroidManifest 中加入此权限

```
<uses-permission xmlns:tools="http://schemas.android.com/tools"
 android:name="android.permission.PACKAGE_USAGE_STATS"
  tools:ignore="ProtectedPermissions" />
```

3.打开手机设置，点击安全-高级，在有权查看使用情况的应用中，为这个 App 打上勾


有权查看使用情况的应用
Google Play商店
Google Play服务
NotificationDemo
绿色守护
查看原图

enter image description here

方法五：通过 Android 自带的无障碍功能

非常感谢@EffectiveMatrix 大神带来的新的判断前后台的方法

此方法属于他原创，具体的博文参照这里
http://effmx.com/articles/tong-guo-android-fu-zhu-gong-neng-accessibility-service-jian-ce-ren-yi-qian-tai-jie-mian/

此方法无法直观的通过下拉通知视图来进行前后台的观察，请到 LogCat 中进行观察即可，以下是 LogCat 中打印的信息

enter image description here

原理

> Android 辅助功能(AccessibilityService) 为我们提供了一系列的事件回调，帮助我们指示一些用户界面的状态变化。 我们可以派生辅助功能类，进而对不同的 AccessibilityEvent 进行处理。同样的，这个服务就可以用来判断当前的前台应用

优势

AccessibilityService 有非常广泛的 ROM 覆盖，特别是非国产手机，从 Android API Level 8(Android 2.2) 到 Android Api Level 23(Android 6.0)

AccessibilityService 不再需要轮询的判断当前的应用是不是在前台，系统会在窗口状态发生变化的时候主动回调，耗时和资源消耗都极小

1. 不需要权限请求
2. 它是一个稳定的方法，与 “方法 6”读取 /proc 目录不同，它并非利用 Android 一些设计上的漏洞，可以长期使用的可能很大
3. 可以用来判断任意应用甚至 Activity, PopupWindow, Dialog 对象是否处于前台

劣势

需要要用户开启辅助功能

辅助功能会伴随应用被“强行停止”而剥夺

方法六：读取 Linux 系统内核保存在/proc 目录下的 process 进程信息

此方法并非我原创，原作者是国外的大神，GitHub 项目在这里，也一并加入到工程中，供大家做全面的参考选择

原理

> 无意中看到乌云上有人提的一个漏洞，Linux 系统内核会把 process 进程信息保存在/proc 目录下，Shell 命令去获取的他，再根据进程的属性判断是否为前台

优点

不需要任何权限

可以判断任意一个应用是否在前台，而不局限在自身应用

缺点

当/proc 下文件夹过多时,此方法是耗时操作

用法
获取一系列正在运行的 App 的进程

```
List<AndroidAppProcess> processes = ProcessManager.getRunningAppProcesses();
```

获取任一正在运行的 App 进程的详细信息

```
AndroidAppProcess process = processes.get(location);String processName = process.name;
```

```
Stat stat = process.stat();int pid = stat.getPid();int parentProcessId
= stat.ppid();long startTime = stat.stime();int policy =
stat.policy();char state = stat.state();
Statm statm = process.statm();long totalSizeOfProcess =
statm.getSize();long residentSetSize = statm.getResidentSetSize();
PackageInfo packageInfo = process.getPackageInfo(context, 0);String
appName = packageInfo.applicationInfo.loadLabel(pm).toString();
```

判断是否在前台

```
if (ProcessManager.isMyProcessInTheForeground()) {
  // do stuff}
```

获取一系列正在运行的 App 进程的详细信息

```
List<ActivityManager.RunningAppProcessInfo> processes =
ProcessManager.getRunningAppProcessInfo(c
```

21. 混合开发

Flutter 如何与 Android iOS 通信?

Flutter 通过 PlatformChannel 与原生进行交互，其中 PlatformChannel 分为三种：

1. BasicMessageChannel：用于传递字符串和半结构化的信息。
2. MethodChannel：用于传递方法调用。Flutter 主动调用 Native 的方法，并获取相应的返回值。
3. EventChannel：用于数据流（event streams）的通信。