目录

[第一财经APP项目重构建议 1](#_Toc438499277)

[现在第一财经APP存在的问题（从技术的角度看） 1](#_Toc438499278)

[怎样的APP才是一款好的APP 2](#_Toc438499279)

[从不同的角度判断一款好的APP 2](#_Toc438499280)

[项目架构 2](#_Toc438499281)

[接口层 3](#_Toc438499282)

[核心层 6](#_Toc438499283)

[界面层 7](#_Toc438499284)

[模型层 7](#_Toc438499285)

[界面篇 9](#_Toc438499286)

[规范性 9](#_Toc438499287)

[单一性 10](#_Toc438499288)

[简洁性 12](#_Toc438499289)

# 第一财经APP项目重构建议

## 现在第一财经APP存在的问题（从技术的角度看）

1. 整个项目结构不明朗；
2. 整个项目没有按照各个模块独立出来；
3. 整个项目对于常量的管理很混乱；
4. 整个APP运行起来的时候内存没有优化好；
5. 整个项目命名不规范；
6. 在布局文件中直接写数字、字符串、color没有写到values中去；
7. 动画卡顿，跳转衔接不流畅；
8. 对于style的运用不到位；
9. APP中有些图标在高分辨率的手机上运行起来很模糊；
10. 有些界面在低分辨率手机上显示不全；
11. APP布局重用不够，重复布局代码太多；
12. 布局代码冗长过度重绘严重。

## 怎样的APP才是一款好的APP

### 从不同的角度判断一款好的APP

#### 从用户的角度来说

* 能够解决用户实际问题；
* 产品结构简单，信息架构清晰，便捷容易上手；
* 交互优雅，有整体的风格；
* 核心功能突出，次要功能含蓄 ；
* **运行足够流畅，不会经常崩溃和访问速度慢；**
* **有合理的提示以及过度动画；**
* **产品有趣味性，互动性。**

#### 开发的角度来说

* 运行足够流畅，不会经常崩溃和访问速度慢（内存优化）；
* 有合理的提示以及过度动画（优秀的衔接动画）；
* 减少用户没必要的操作，替用户做某些事儿（保存用户的痕迹）。

## 项目架构

项目分为了四个层级：

1. 模型层、

模型层定义了所有的模型；

1. 接口层、

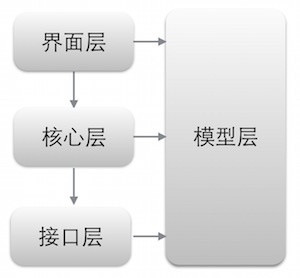
接口层封装了服务器提供的API；

1. 核心层、

核心层处理所有业务逻辑；

1. 界面层。

界面层就处理界面的展示。

几个层级之间的关系如下图所示：  


### 接口层

接口层封装了网络底层的API，并提供给核心层调用。比如：

1. PostEngine，请求引擎类，对请求的发送和响应结果进行处理；
2. Response，响应类，封装了Http请求返回的数据结构；
3. Api，接口类，定义了所有接口方法；
4. ApiImpl，接口实现类，实现所有接口方法。

PostEngine将请求封装好发送到服务器，并对响应结果的json数据转化为Response对象返回。Response其实就是响应结果的json数据实体类，json数据是有固定结构的，分为三类，如下：

{**"event"**: "0", **"msg"**: "success"}

{**"event"**: "0", **"msg"**: "success", **"obj"**:{...}}

{**"event"**: "0", **"msg"**: "success", **"objList"**:[{...}, {...}], **"currentPage"**: 1, **"pageSize"**: 20, **"maxCount"**: 2, **"maxPage"**: 1}

event为返回码，0表示成功，msg则是返回的信息，obj是返回的单个数据对象，objList是返回的数据对象数组，currentPage表示当前页，pageSize则表示当前页最多对象数量，maxCount表示对象数据总量，maxPage表示总共有多少页。根据此结构，Response基本的定义如下：

**public** **class** **Response**<T> {

**private** String event;

**private** String msg;

**private** T obj;

**private** T objList;

**private** int currentPage;

**private** int pageSize;

**private** int maxCount;

**private** int maxPage;

*//getter和setter方法*

...

}

每个属性名称都要与json数据对应的名称相一致，否则无法转化。obj和objList用泛型则可以转化为相应的具体对象了。

Api接口类定义了所有的接口方法，方法定义类似如下：

**public** Response<Void> login(String loginName, String password);

**public** Response<VersionInfo> getLastVersion();

**public** Response<List<Coupon>> listNewCoupon(int currentPage, int pageSize);

ApiImpl则实现所有Api接口了，实现代码类似如下：

**@Override**

**public** Response<Void> login(String loginName, String password) {

**try** {

String method = Api.LOGIN;

List<NameValuePair> params = **new** ArrayList<NameValuePair>();

params.add(**new** BasicNameValuePair("loginName", loginName));

params.add(**new** BasicNameValuePair("password", EncryptUtil.makeMD5(password)));

TypeToken<Response<Void>> typeToken = **new** TypeToken<Response<Void>>(){};

**return** postEngine.specialHandle(method, params, typeToken);

} **catch** (Exception e) {

*//异常处理*

}

}

实现中将请求参数和返回的类型定义好，调用PostEngine对象进行处理。  
接口层的核心基本上就是这些了。

### 核心层

核心层介于接口层和界面层之间，主要处理业务逻辑，集中做数据处理。向上，给界面层提供数据处理的接口，称为Action；向下，调用接口层向服务器请求数据。向上的Action中定义的方法类似如下：

**public** void getCustomer(String loginName, CallbackListener<Customer> callbackListener);

这是一个获取用户信息的方法，因为需要向接口层请求服务器Api数据，所以添加了callback监听器，在callback里对返回的数据结果进行操作。CallbackListener就定义了一个成功和一个失败的方法，代码如下：

**public** **interface** **CallbackListener**<T> {

*/\*\**

*\* 请求的响应结果为成功时调用*

*\* @param data 返回的数据*

*\*/*

**public** void onSuccess(T data);

*/\*\**

*\* 请求的响应结果为失败时调用*

*\* @param errorEvent 错误码*

*\* @param message 错误信息*

*\*/*

**public** void onFailure(String errorEvent, String message);

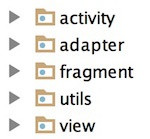
}

接口的实现基本分为两步：

1. 参数检查，检查参数的合法性，包括非空检查、边界检查、有效性检查等；
2. 使用异步任务调用接口层的Api，返回响应结果。

需要注意的是，Action是面向界面的，界面上的数据可能需要根据不同情况调用不同的Api。  
后续扩展可以在这里添加缓存，但也要视不同情况而定，比如有些变化太快的数据，添加缓存就不太适合了。

### 界面层

界面层处于最上层，其核心就是负责界面的展示。  
因为公司有为不同商户定制不同app的需求，因此，这里就需要建立多个app的界面，这是一个很麻烦的事情，还好，Android Studio提供了很方便的方法可以大大减少工作量，主要通过设置Gradle，不同app可以添加不同的productFlavors。  
界面层package的定义可以根据不同类型划分，主要分为以下几个包：  
  
其中，activity、adapter、fragment各自都有一个基类，做统一的处理，比如定义了一些共用的常量、对象和方法等。  
界面层是最复杂，最容易变得混乱不堪，最容易出问题的层级。所以，从架构到代码，很多东西都需要设计好，以及规范好，才能保证程序易维护、易扩展。后续的文章里将会详细分享下我在这方面的经验。

### 模型层

模型层横跨所有层级，封装了所有数据实体类，基本上也是跟json的obj数据一致的，在接口层会将obj转化为相应的实体类，再通过Action传到界面层。另外，模型层还定义了一些常量，比如用户状态、支付状态等。在Api里返回的是用1、2、3这样定义的，而我则用枚举类定义了这些状态。用枚举类定义，就可以避免了边界的检查，同时也更明了，谁会记得那么多1、2、3都代表什么状态呢。然而用枚举类定义的话，就必须能将1、2、3转化为相应的枚举常量。这里，我提供两种实现方式：

1. 使用gson的@SerializedName标签，比如0为FALSE，1为TRUE，则可以如下定义：

**public** **enum** BooleanType {

**@SerializedName**("0")

FALSE,

**@SerializedName**("1")

TRUE

}

1. 通过定义一个value，如下：

**public** **enum** BooleanType {

FALSE("0"),

TRUE("1");

**private** String value;

BooleanType(String value) {

**this**.value = value;

}

**public** String getValue() {

**return** value;

}

}

通过gson的方式，直接访问TRUE或FALSE就会自动序列化为1或0；如果通过第二种方式，因为没有序列化，则需要通过getValue方式获取1或0。

## 界面篇

将项目分为了四个层级：模型层、接口层、核心层、界面层。其中，最上层的界面，是变化最频繁的一个层面，也是最复杂最容易出问题的一个层面，如果规划不好，很容易做着做着，又乱成一团了。  
要规划好界面层，至少应该遵循几条基本的原则：

1. 保持规范性：定义好开发规范，包括书写规范、命名规范、注释规范等，并按照规范严格执行；
2. 保持单一性：布局就只做布局，内容就只做内容，各自分离好；每个方法、每个类，也只做一件事情；
3. 保持简洁性：保持代码和结构的简洁，每个方法，每个类，每个包，每个文件，都不要塞太多代码或资源，感觉多了就应该拆分。

### 规范性

每个人的编码习惯和风格都不同，不说那些缺乏良好编码习惯的开发人员，就连那些已经养成良好编码习惯的人员，很多方面都会不同。比如缩进，有的喜欢4个空格，有的喜欢两个空格；比如变量名，有的喜欢m开头，例如mValue，有的喜欢直接就命名为value。如果不设定好规范，让每个人都按照自己的习惯和风格去编码，久了肯定乱，尤其当团队中存在还没养成良好编码习惯的人员时，更容易乱。所谓无规矩不成方圆，若无规范，久必乱。定义好规范，才能统一风格，才可提高代码可读性，同时也提高了维护性，还减低了引入bug的机会。  
开发规范并没有统一的标准，在这里，我只是根据自己的经验对一些点提供一点建议，仅供参考。

#### 缩进

很多人都习惯用Tab缩进，不管是规范4个空格还是2个空格，统一设置好Tab缩进的size就好了，这样就不用让每个人都去敲空格。

#### 命名

一个好的命名，一眼就可以从名字中看到它是干嘛的，做什么用，什么类型等等。举个id命名的例子，看到有些团队喜欢将一些控件缩写，比如TextView缩写为tv，ListView缩写为lv，这种缩写倒是挺简洁的，但是并不能一眼就能看出它是什么，对于不熟悉的人来说，谁知道tv和lv是什么啊，还不如用text和list更明确些。我喜欢的id命名结构为：控件\_范围\_功能，例如：edit\_login\_password，这是一个登录页的密码输入框。

#### 单位

文字大小的单位应该统一用sp，其他元素用dp。因为这两个单位是与设备分辨率无关的，能够解决在不同分辨率设备上显示效果不同的问题。另外，SDK里面，对文字大小系统默认是用sp单位的，但其他元素单位默认却不是dp，而是px的，同时也没有提供dp的设置接口，所以，自己写两个dp和px转换的方法是很有必要的。

最重要的并不在于规范怎么定义，而是在于规范的严格执行。如果规范定义好了，但却不遵守，那规范就等于形同虚设，因此，规范一旦设定，就要严格执行。

### 单一性

我们都知道，面向对象设计中，有一个基本原则就是单一职责原则，它规定一个类应该只有一个发生变化的原因。而这里说的单一性，不只是规定类，也规定了方法、包，甚至到最大层面的分层架构。保持单一性是减低耦合度的关键标准，其目的就是各方面的解耦。架构上的分层就是最大层面的解耦，而方法上的单一就是最小层面的解耦了。

#### 界面的单一

界面上的单一，首先是界面的布局和界面的数据应该分离。这一点，Android已经用layout和Activity做好解耦了，我们只要确保用layout文件排好布局，在Activity展示数据就好了。另外，界面数据的获取和展示也应该分离。很多开发团队习惯将数据的获取和展示都放在Activity或Fragment里完成的，[架构篇](http://keeganlee.me/post/android/20150605)的读者里也有人反映了这个情况，请求接口、获取数据、检查数据、显示数据更新UI，全都在界面上完成的。这样子的话，当数据的获取发生改变时，比如要添加缓存，这时候界面就需要改动了，当数据的展示也需要修改时，比如某个控件要展示其他数据，界面也一样需要改动，也就是说，界面上已经有两个发生变化的原因，这就违反了单一职责原则。  
界面上的单一，就是要保持界面上每个维度都做好分离，从界面的布局，到数据的获取，数据的检查，数据的展示。

#### 包和类的单一

定义包之前，需要先想好它的职责是什么，明确定义并确保它只有一个职责。例如，com.keegan.activity，就是activity类的包，不会有其他组件；com.keegan.adapter，就是存放各种适配器的包；com.keegan.util就是工具包了。同样，类的定义，也是需要明确它的单一职责。有些人习惯将adapter写在Activity里，因为觉得这个adapter只在这个Activity里用到，没必要再把它独立出来。以前的我也是这么干的，这么做了一段时间之后，觉得实在糟糕透了，重复的代码无法复用，界面上的一点小需求调整时，很多代码需要跟着调整。后来，进行了一番重构，将所有adapter独立了出来，并抽象出了一个adapter的基类，自此，当需要再添加adapter时，编写的代码量大大减少了，当界面需求调整时，修改的地方也大大减少了。所以，不要让一个类做太多事情，要分离好各种元素，每个元素只做一件简单的事。

#### 方法的单一

方法的单一，表现为一个方法是对一个行为的封装。然而，一个行为又可以拆分为多个步骤，每个步骤其实也是一个更细的行为，又可以封装成一个新的方法。因此，方法嵌套方法是一种常态。那么，保持方法的单一性，关键并不在于怎么定义这个方法的行为，而在于这个行为要怎么拆分成更细的行为。举个例子，通常在Activity的onCreate方法，做数据的初始化，细分出来就分为了：控件的初始化、逻辑变量的初始化、数据的加载和展示。数据的加载和展示可以再细分：从缓存加载数据、从网络加载数据、展示数据。每个细化的行为都应该封装为一个独立的方法，这样，才真正符合方法的单一性。

#### 资源文件的单一

Android提供了各种资源文件，strings.xml用来存储字符串，arrays.xml用来存储字符串数组，colors.xml用来存储颜色值，dimens.xml用来存储尺寸值，等等。资源文件的单一，是说所有相关的资源信息要在资源文件里定义并引用到代码或布局文件里，而不是在代码或布局文件里直接定义。很多开发人员，为了图方便，应用界面中出现的字符串经常在代码或布局文件里直接定义的，尺寸值也是，这样造成的结果就是，当某些字符串需要修改时，比如要支持国际化，或一些尺寸值需要修改时，通常是很多地方都要修改。因此，就必须规范好，应用界面中的字符串统一在strings.xml中定义，颜色值统一在colors.xml中定义，尺寸值统一在dimens.xml中定义，代码或布局里需要用到的都去引用资源文件相应的字段。

要保持单一性，必定伴随着重构。需求总会变动，代码总会扩展，扩展了慢慢就会破坏原有的单一性，因此就需要重构，再次保持单一性。不断扩展，不断重构，这样才能不断保持良好的单一性。

### 简洁性

代码最怕的就是臃肿，臃肿的代码可读性差，维护麻烦，扩展更不用说了。没有人会喜欢看臃肿的代码，去维护更痛苦。我看到臃肿的代码，都恨不得即刻进行重构。让代码保持简洁，会让人看得舒服，一目了然，维护和扩展起来也都非常方便。简洁的代码，甚至不需要写注释，只从代码就能让人一眼看懂其做了什么。简洁也并不只表现在代码上，类、包、资源文件等的命名和组织结构等也同样需要保持简洁。  
如何保持简洁？这个问题并没有一个标准的答案，但有一个判断是否简洁的简单标准，那就是：直接阅读代码就能够理解代码的意图，如果意图不够明显，那就说明这段代码还不够简洁。类、包、资源文件等等，也是同样的评判标准。下面是我觉得对保持简洁有一定作用的一些操作方法。

#### 包的组织

按照组件类型来分包，而不是按业务模块来分包。业务有可能会变，但组件类型是基本不变的。另外，新加入的开发人员，对业务不熟悉，但对组件是很清楚的，理解快，入手也快。

#### 类和接口的命名

组件类的命名添加该组件的后缀，例如：Activity类命名添加Activity后缀，Fragment类命令添加Fragment后缀，适配器添加Adapter后缀，等等。实体类则可添加BO的后缀名称，工具类添加util后缀，接口的实现类添加Impl的后缀。接口的命名也一样，比如，我的项目中，接口层的接口后缀都带上了Api，核心层的接口后缀都带Action。

#### 资源文件的分类

strings.xml文件用来存储应用中的所有字符串，包括页面标题，按钮文字，标签文字，提示文字等等，应该做好分类并统一存放。下面是我推荐的分类方法，如果某个分类的字符串数量太多了，还可以拆分出来放到一个独立的文件，比如页面标题，可以拆分到strings\_title.xml文件里，其他资源文件也可以用类似的方式进行处理：

* + 页面标题，命名格式为：title\_{页面}
  + 按钮文字，命名格式为：btn\_{按钮事件}
  + 标签文字，命名格式为：label\_{标签文字}
  + 选项卡文字，命名格式为：tab\_{选项卡文字}
  + 消息框文字，命名格式为：toast\_{消息}
  + 编辑框的提示文字，命名格式为：hint\_{提示信息}
  + 图片的描述文字，命名格式为：desc\_{图片文字}
  + 对话框的文字，命名格式为：dialog\_{文字}