关于Android架构，有这一篇就够了（干货总结）


DevCW (/u/bc06755b87c3) ✓已关注
1.6 2018.01.05 16:35* 字数 3170 阅读 5800 评论 6 喜欢 28
(/u/bc06755b87c3)

Android开发过程中，我们或许见过这样的现象：

1. 上帝类频频出现，有些类可能包含多个功能模块的代码

2. 臃肿类数不胜数， 很多类里面可能少则1000行，多则几千行的代码

3. 不同功能混杂，类中包含不同层次的功能，难以查询逻辑主线

4. 无节操的代码重叠

以上现象，在我们开发中，尤其是接手别人代码时，会显示痛不欲生；那么我们来看一下上述现象的弊端有哪些 ？

1. 代码查找 想一下找出需要的代码，难！哪怕ctrl + F查找，都要分析好久！！！

2. 逻辑交叉混乱 想改什么东西，实在鼓不起勇气！

3. 不同代码齐聚一堂 鬼才知道添加或修改一个功能都要改多少地方，何况在不了解需求的前提下

如果频频遇到以上现象，或许我们就应该想了，到底什么样的代码，才是所谓的好代码呢？ 有没有什么量化的标准 ？ 又如何来很好的避免上述的问题 ？

答案是**规范**与**标准设计**

首先谈一下规范，不详谈，不矫情，建议参考Alibaba推出的Java开发规范，请走这里：

阿里巴巴Java开发手册 (https://link.jianshu.com?t=https%3A%2F%2Fm.aliyun.com%2Fyunqi%2Farticles%2F69327)

这里主要详谈的是设计，首先来谈一下开发出高内聚，低耦合，层次分明，可读性高的代码，需要怎么做 ？

1. 分清代码职能

想要添加一段代码，首先必须明确这段代码是用来做什么， 这段代码的功能是什么类型的功能 ？ 这是一段高度反映业务的代码？或者是可复用型的独立业务功能代码？又或是与业务完全独立的常规辅助类型的代码？ 只有明晰了这段代码的功能与类型，才能映射出更好的设计。

2. 初步划分

模块划分，是系统源码合理划分的最有效的途径之一。但模块的划分不是必须的，项目越大，模块划分带来的益处越大；同时关注点的划分也越难。小规模的项目，完全可以越过这一步。

3. 初步层次划分

如果说，模块的划分可以把一个大的系统有效地划分为几个子系统；那么针对不同的子系统，同样包含有大量的代码；因此层次划分的重要性就不言而喻了；至于这个的使用，根据业务进行自我分析判断；一般而言，层次划分是必要的，如当前流行的MVC, MVP, MVVM等框架

4. 功能属性划分

和1类似，我们需要明白一点，我们的代码有几种类型？ 总体来区分，分为Common与Business;我感觉Business下还可以进行划分，有些Business是需要复用，不同模块都需要的， 我称之为**弱业务型**；有些Business代码没法重用，我称之为**强业务型**。因此，我的功能划分包括： **通用型**， **弱业务型**以及**强业务型**。

5. 单一职责原则

单一职责原则，作为面向对象的6大设计原则之一，极大的保证了不会产生逻辑交叉混乱与上帝类。也是日常开发中必须遵守的原则。在阿里巴巴的**《Java开发手册》**中，也提到了这点，概括来说，就是保证大到每一个子系统，每一个模块，每一个分包，小到每一个类，每一个方法都仅执行单一的功能；这个功能也就是关注点的划分，需要根据开发经验来区分，模块的划分与方法的划分必然是不同的，而且相去甚远。

6. 复用性与扩展性

代码复用性， 相信每一个人都对这个原则有深刻的理解；复用性可以极大的提高开发效率， 当然必须要配合扩展性，才能更好的达到目的。扩展性越强，复用范围就越广，该模块或方法就越具有原子性。

7. 封装性

在一个页面中，我们可能需要实现形形色色的需求；如果把这些需求都放在Activity中，那么可能最后Activity的规模就可想而知了；除了通过分层把不属于Activity的逻辑划分出去（如MVP中将逻辑放到了P）,余下一些的View实现，也需要大量的代码来构建；这时候我们可以再次划分不同的模块，如TitleBar我们可以根据业务定义自己的TitleBar, 只暴

露一些基本的配置，如setTitle（）等；然而有时TitleBar中也会包含大量的业务逻辑，这时我们可以在TitleBar UI层封装的基础上，加入一个TitleBarDao来处理TitleBar相关的实现，将这块逻辑继续分发到TitleBarDao中。

以上回答了如何更好的设计一段代码，才能更好的保证其逻辑层次的清晰， 解耦性以及可读性，都是个人愚见；代码设计是根据不同业务，不同规模进行合时宜的设计，万不可为了设计而造成过度设计。设计的目的是提高开发效率，提高维护与测试效率。说了这么多，如何来实现呢 ？ 以下是一些简单的实例。

1）添加一个View,View显示的宽度是屏幕的3/4

```
View targetView;

LayoutParams lp = targetView.getLayoutParams();

lp = 3 * ...获取屏幕宽度 / 4；

targetView.setLayoutParams(lp);
```

以上这段代码，很简单，相信很多人也写过；这段代码首先我们分析功能实现： View宽度的设计 和屏幕宽度获取; 这是一段通用型的代码，因此我们可以放在通用层中，方法内部包含屏幕宽度获取的实现，因此根据单一职责原则，我们可以将之抽取出来。

2）string, dimen, style文件

这些文件的使用，就是为了将文本, 尺寸, 样式等抽取出来，实现单一的目的。

3）组件化，插件化

这两种架构方式是大系统规模的代码广义层面的架构方式，将一个大系统分为若干子系统，子模块来管理；

4）MVC, MVP, MVVM

这些架构方式，则属于从小的层级进行代码分层划分，划分为不同的层次，每个层次实现单一职责。

说了这么多，代码架构的目的是什么？

1. 解耦

2. 复用

3. 可读性

4. 健壮性

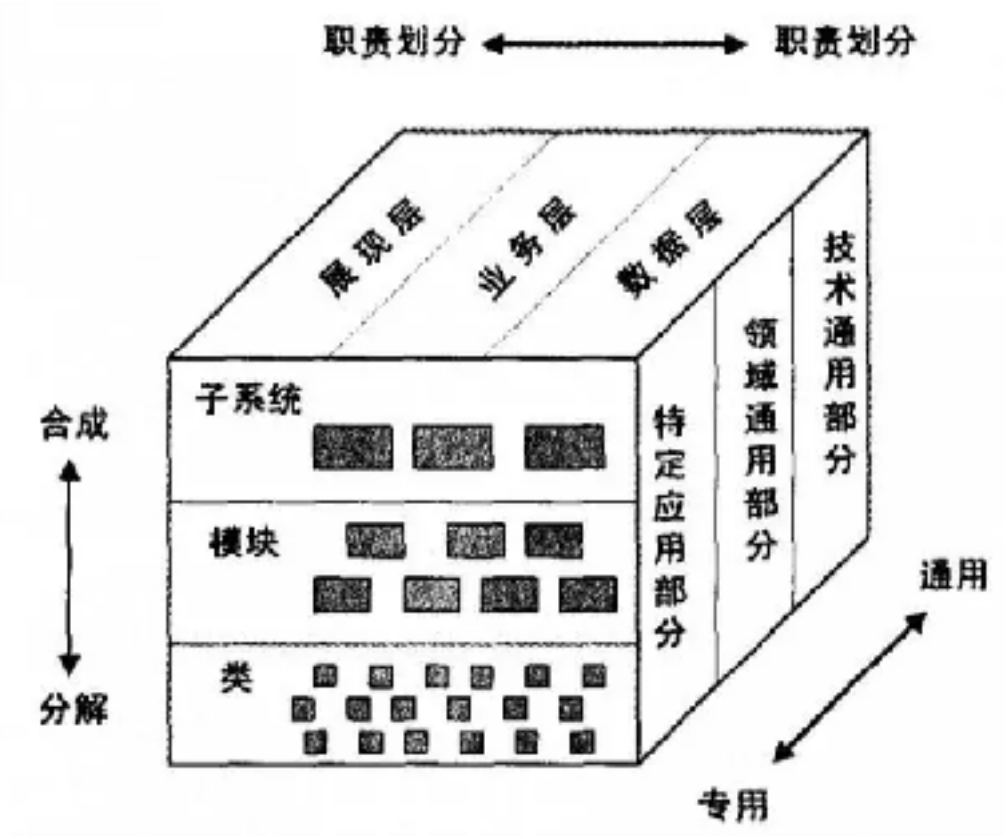
5. 提高并行开发效率

最近出了很多架构思想，无论是最初的MVC, 还是近几年风头极盛的MVP, Google推出的MVVM,还是系统层面的组件化，模块化，插件化；最终遵循的架构原则无非就是三点：

1. 横向划分模块

2. 纵向划分层次

3. 解耦通信



1. 横向分块

一个大的项目，代码量是极大的，甚至达到1G以上；这时必须先以大的层级进行划分，才能有效达到分离的目的。现在流行的模块化思想便有了用武之地；模块化思想主张将一个系统横向切分为不同的子系统或者可以称之为模块，根据业务与开发需要将工程划分为Common模块， Business-01模块， Business-02模块... 等，同时不同的模块之间从大的层级实现职责单一原则。

2. 模块之间解耦

一个系统划分为若干子系统，模块后，不同的模块之间存在交叉通信是必然的；那么如何实现模块之间解耦呢？

组件化的解耦**思路是实现一套注册路由机制，根据一套统一的注册路由系统来统一实现跨组件通信，当前比较流程的框架有ActivityRouter, ARouter等

插件化的解耦**则直接通过Android系统提供的Binder机制来进行

3. 纵向分层

从系统层级做了模块划分后，每个模块必然也需要自己的架构，这时候就需要使用分层的思想了，也就是要说的纵向分层策略。MVC, MVP, MVVM，Flutter这些设计则使用了纵向分层的思想；它们分别将一段代码或一个页面划分为3个层级：

Model层： 处理数据

View层： 处理UI显示

MVC的Controller, MVP的Presenter, Flutter的Store 则处理主要逻辑

经以上划分后，数据处理， Ui显示， 逻辑处理都放到了各自的层进行处理，每个层仅执行本层应该做的操作。分层后，代码已经做到了结构清晰，尽可能的解耦， 易于理解维护。不过，在开发中，Model返回的数据，不一定是我们想要的格式； 而且将异步放到Model层，则Model除了处理数据以外，还要进行不同的Async处理。在以上想法下，我们可以继续划分：

Model层： 处理数据（请求， 数据库， 缓存等）

Work层： 处理异步相关

Presenter层: 处理核心控制逻辑

Converter层: 数据转化层 将Model数据转化为UI需要的格式

4. 层次间解耦

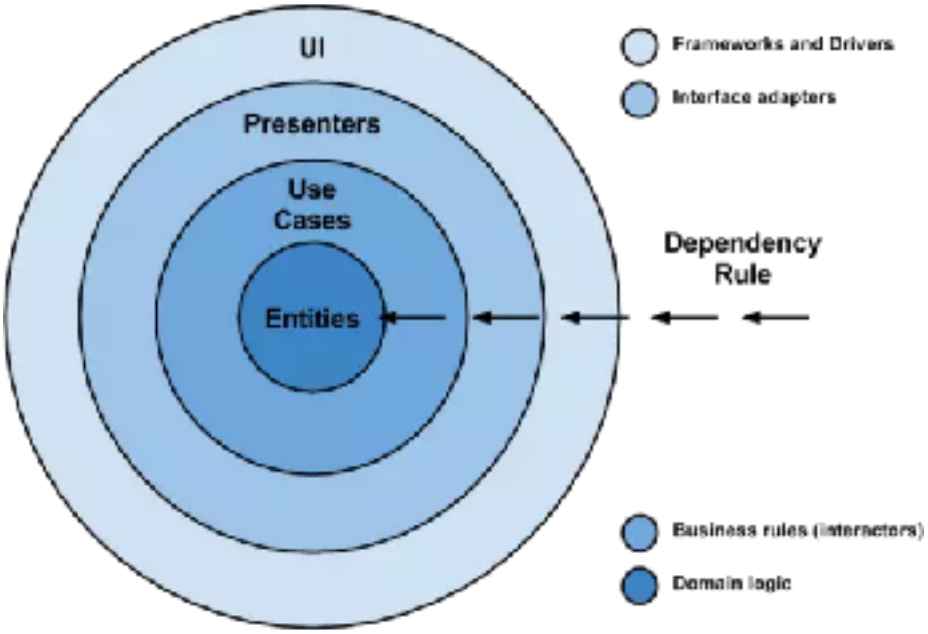
根据CleanArchitecture原则，采用圆葱形式的分层与解耦策略，因此在设计时注意两点：

1）明确的层次限定

2）禁止跨层次调用

3）从内到外单向调用

根据洋葱结构，由内到外我们可以得到Model -> Work -> Converter -> Presenter -> UI;
在调用时，某层只能调用向内的邻接层，如Presenter只能调用Work层（存在异步操作时）或Model层（无异步操作，甚至无异步操作也可以在Work中封装一层），而不能出现Model层调用Presenter层的现象；相同的，UI层只能调用Presenter层，而不能跨层调用Model层；Converter作为外围辅助层，不直接参与洋葱结构



层通信方式

层与层之间的通信，必然不能通过直接调用的方式（特别是View层与Presenter层），我们可以参考MVP模式中的**接口通信方式**或者Flutter模式中的**Event路由模式**来实现。

以登录功能为例，我们需要以下功能

参数校验

加载

登录请求

返回处理

取消加载

显示结果

参数校验， 参数的具体信息来自View层，但是其本身功能为逻辑处理，因此适合在Presenter层

加载， 取消加载 涉及到Activity Context来加载Dialog,因此适合在View层

登录请求， 毫无争议放在Model与Work层，如有参数转化，需要加入Converter层

返回处理， 毫无争议放在Presenter层

显示结果， 涉及到Toast显示以及提示，必然在View层

有些实现可以涉及Context且本身属于Presenter处理，不好区分；因此建议通过Context类型区分， 涉及必须使用Activity Context如Toast, Dialog, PopupWindow则在View层，AplicationContext亦可的则可以放在Presenter层。

本篇文章从理论方面讲解了Android架构一些常用知识与理念，且纯属个人愚见，有争议者可提出一并讨论；目的就是找到一个可以合理解决大部分问题且适合移动端的开发架构。

your like is my will~

赞赏支持