# 江 苏 科 技 大 学

# 本 科 毕 业 设 计（论文）

学　　院 计算机学院

专　　业 软件工程

学生姓名 甘 尧

班级学号 1341904209

指导教师 夏永锋

2017 年 6 月

江苏科技大学本科毕业论文

# 基于 MVP 的 Android App 架构设计与案例分析

Android Architecture Based On MVP And Case Analysis

江 苏 科 技 大 学

# 毕业设计（论文）任务书

学院名称：计算机学院　专　业：软件工程

学生姓名：甘 尧　学　号：1341904209

指导教师：夏永锋　职　称：讲 师

**毕业设计（论文）题目：**

基于 MVP 的 Android App 架构设计与案例分析

一、 毕业设计（论文）内容及要求（包括原始数据、技术要求、达到的指标和应做的实验等）

1 内容:

Android 是一种基于 Linux 的自由及开放源代码的操作系统，分析 Android 架构以及主流框架实现，设计完成基于 MVP 分层架构的 Android App 架构。并且基于所设计的架构进行实例开发，对架构进行验证。

2 要求:

(1) 分析 Android 系统 App 设计架构；

(2) 分析 Android 主流应用层框架源码及复现部分框架;

(3) 分析几个主流架构，比较各优缺点；

(4) 根据实例需求设计 Android App 架构；

(5) 设计几个场景，验证架构优越性，如需求添加，需求更改，增加产品线等。

二、 完成后应交的作业（包括各种说明书、图纸等）

1. 毕业设计论文一份（不少于 1.5 万字）；

2. 外文译文一篇（不少于 5000 英文单词）；

3. 软件产品及设计源程序；

4. 其它(根据课题性质、类型确定)。

三、 完成日期及进度

2017 年 3 月 13 日至 2017 年 6 月 10 日，共 13 周。

进度安排：

1：3.13-3.26，调查、收集资料、完成开题报告，准备好课题设计；

2：3.27-4.09，系统分析、初步设计、拿出设计方案；

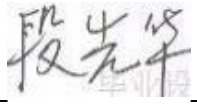
3：4.10-5.07， 详细的设计、编程、调试，系统成型；

4：5.08-5.21， 论文初稿完成并进行查重；

5：5.22-6.10， 修改论文并定稿、答辩。

[1] ReactiveX. Reactive Extensions for the JVM
https://github.com/ReactiveX/RxJava [DB/OL]. Github, 2016
[2] Wequick. Small 开源组件化框架
https://github.com/wequick/Small [DB/OL].Github, 2014
[3] Alibaba. Atlas 阿里巴巴开源组件化框架
https://github.com/alibaba/atlas [DB/OL]. Github, 2016
[4] Retrofit. Square 异步 Http 请求框架
https://github.com/square/retrofit [DB/OL]. Github, 2016
[5] Greenrobot. EventBus 事件驱动框架
https://github.com/greenrobot/EventBus [DB/OL]. Github, 2016
[6] IBM. AspectJ AOP 框架
https://github.com/eclipse/org.aspectj [DB/OL]. Github, 2016
[7] Wiki.SoftwareArchitecture
https://en.wikipedia.org/wiki/Software_architecture [DB/OL]. Github, 2016
[8] Steve McConnel.“代码大全第二版”（Riel 1996）[M]. 电子工业出版社,2006.
[9] 甘尧. 打造超越 EventBus 的事件管理框架
http://blog.csdn.net/ganyao939543405/article/details/52847648[DB/OL].CSDN, 2016
[10] 甘尧. Android MVP 的实现
http://blog.csdn.net/ganyao939543405/article/details/52963144[DB/OL].CSDN, 2016
[11] Google. Android 架构蓝图
https://github.com/googlesamples/android-architecture [DB/OL]. Github, 2016.
[12] Alibaba. Atlas 官网 http://atlas.taobao.org/ [DB/OL]. Taobao, 2017.

系(教研室)主任：  （签章） 2017 年 3 月 15 日

学院主管领导：  （签章） 2017 年 3 月 15 日

# 摘　要

随着 Android 客户端 App 的飞速发展，简单的 MVC 分层架构早已不符合当下开发的需求，现在的 App 具有开发上线周期短，需求变化多，功能体量多跨度大，更新周期短，适配机型多，合作开发人员多等特点。

那么，对于现在体积庞大的 App，本文设计了一个立体的架构。本架构的设计从横向，纵向和延伸性上考虑。纵向主要分解流程；横向主要分解业务，和开发期间的功能复用；延伸性上本架构充分考虑了架构的升级和发展潜力，降低架构演进成本。

纵向上，本架构实现了合理分层。每个组件进行分层例如 MVP；App 整体被分为驱动层，lib 层，架构层，App 层，保证了底层架构对于上层 App 的控制力。

横向上，本架构采用组件化和模块化设计。组件化将每个部分进行细粒化的分离，对每个具体业务进行切割隔离，并且实现组件之间的树形依赖管理，使底层组件可以最高效的被上层业务组件复用；赋予组件生命周期管理，并且组件层级之间使用 DI 进行依赖拼装。在业务量大的情况下，组件提供了模块化进一步切割业务及功能，辅以页面路由器进行模块间跳转，对于大型 App 来说，组件化使各个团队在开发期间的隔离分工；App 落地运行期间子模块按需加载，即功能组件的懒加载，为用户节省资源；除此之外利于动态增加功能模块。

延伸性上，本架构将架构的底层设计高度抽象化，严格遵循面向接口，保证了架构的演进升级。对于上层 App，现在 App 对于更新的需求日益增长，架构集成了增量更新组件，Bugfix 上，架构也集成了动态热补丁。

同时本架构也经过了实际开发案例的检验，最后给出了已经上线的实际开发案例检验了本架构优越性。

**关键词：** Android；架构设计；App 开发；组件化；模块化；分层；MVP.

# Abstract

With the development of the application of the android client,the mvc witch a simple layered architecture can not meet the current complex needs.Nowadays, app has many traits such as short circle of the development,frequent change of requirements,complex in function,frequently updated, many types of adaptation and multi person cooperation.

So, for the now bulky App, this article designed a three-dimensional architecture. The design of this architecture is considered in terms of landscape, longitudinal and extensibility. The main decomposition process; the main decomposition of the main business, and the development of the functional reuse; extension of the framework of the architecture to take full account of the upgrade and development potential, reduce the evolution of the cost of architecture.

For vertical, the architecture to achieve a reasonable layer. Each component is layered, such as MVP; App overall is divided into the driver layer, lib layer, architecture layer, App layer, to ensure that the underlying architecture for the upper App control.

For horizontal, the architecture uses modular and modular design. The component separates each part by fine-grained separation, isolates each specific business, and implements tree-dependent management between components so that the underlying components can be most efficiently reused by the upper business components; Cycle management, and the use of DI between the component level to rely on the assembly. In the case of large traffic, the components provide a modular further cutting business and functions, supplemented by page routers to jump between modules, for large App, the components of the various teams during the development of the division of labor; During the sub-module on-demand loading, that is, lazy loading of functional components for users to save resources; in addition to the dynamic function of the module.

For extensibility, the architecture of the architecture of the underlying design highly abstract, strictly follow the interface-oriented, to ensure the evolution of the architecture upgrade. For the upper App, App is now growing demand for updates, the architecture integrates incremental update components, Bugfix, the structure also integrates dynamic hot patch.

At the same time the structure has also been the actual development of the case test.

**Keywords:**Android;Architecture;Design;ApplicationDevelopment;Component-based;Bundle-based;Layered architecture;MVP.

# 目 录

# 第一章 绪论

## 1.1 研究背景

时代背景:互联网热潮

现在正处于互联网最好的时代，任何人任何想法都想在这片互联网热潮中寻找一丝机会，因此，各种 App 不断涌现。迫于市场竞争的压力，现在的 App 必然面临着开发时间紧迫，功能迭代迅速等必要需求，如何在前期能快速开发并且兼顾在后期顺利的迭代维护，这就需要一个优秀的架构进行支撑。

关于 Android 应用层框架国内外使用教程较多，但是将诸多主流框架综合分析，论述的文章还存在空缺。对于 Android 本身设计来说是基于 MVC 三层架构，但是随着 App 体量的极速膨胀，MVC 显然已经不能满足现在的需要。

## 1.2 Android 架构的现状和发展

Android 的架构发展基本是从最原始给予官方的 Activity，xml 的设计开始，过度到开始分层的阶段即 MVC，随后是 MVVM，接着 MVP，从分层角度来说 MVP 已经是作为客户端应用较为理想的结构了。但是随着业务量的增多，从横向上来看，接口越来越多，模块之间还是存在很多耦合，包括四大组件的通讯交互等等，所以 EventBus ，Otto 这些消息驱动框架便出现了。Model 层随着业务的增多，也需要一款类似 SpringMVC 或者 Struct 的 Model 业务代理框架，这便是 Retrofit，他的出现使得 Api 层无需再关注各种 Http 协议的处理，开发者仅仅需要关注自己的业务接口即可。

总的来说随着编程理念的不断革新，各种新的技术不断涌现，亦或是其他已有的技术被第一次引入到 Android，例如 Rx，RN，DI，AOP，Bundle 化等等。

### 1.2.1 Android 架构的初期状态

Android 应用初期架构是基于 Android 官方在设计系统时给定的 View 和 Repositry，即 Activity 为 Repositry 数据的提供者，布局 xml 为 View。简而言之就是直接使用原生的 api,大量 Handler，Thread，HttpClient 等充斥着 Activity，使 Activity 越来越臃肿。问题在于 Activity 不仅需要充当 数据源的角色，而且大量 View 特效动画等等控制也需要在 Activity 中完成，当然还有生命周期的回调。

### 1.2.2 Android 4.0 之后，Fragment 的引入

后来，为了解决 Activity 中的耦合问题，Android 官方在 4.0 之后推出了

Fragment。Fragment 虽然从 Activity 中抽走了不少 View 控制代码。但是较为繁琐的使用和麻烦的生命周期控制无疑又增加了开发复杂度。有些时候增加 Fragment 甚至是得不偿失的。

### 1.2.3 MVP & MVVM

传统的 MVC 越来越无法适应越来越多的需求，所以隔离性更强的 MVP 出现了，他将 Model 和 View 完全隔离开来，View 层即 Activity 仅仅处理 UI 逻辑，数据业务的处理被放在了 Presenter 中，Activity 迎来了解放。

而 MVVM 则脱胎于数据绑定 DataBinding，有很多开发者在微软平台上吃到了 DataBing 的甜头，数据绑定确实是一个方便高效的开发模式。数据 ←→ UI 双向绑定的模式为开发者省去了大量代码，等于 IDE 或者说是 lib 帮助开发者实现了一个较为通用的数据 Adapter，但是相比于他的优点，MVVM 的缺点也同样明显。问题在于 MVVM 对数据的控制力太小了，所以不建议在复杂的业务场景中使用。

### 1.2.4 事件驱动编程

随着业务的增多，组件之间，层级之间接口回调迅速膨胀，为了解耦组件，层级之间的通信，EventBus 这种事件总线框架出现了，它的出现使得 Android 层级组件之间的通讯可以以类似广播的形式实现，而不是较为耦合的接口调用。

这样的话，接不接受调用取决于你这个模块是否订阅相关事件，如果发布和订阅互相不知道对方的情况。发布者不知道谁会订阅该事件，订阅方也不知道发布者会在何时何地发布事件，这样通讯就被解藕了。

### 1.2.5 RxJava 带来的革命

RxJava 从编程范式，异步处理方法，数据处理流程等方面给 Android Code 带来了革新。函数式的链式调用；和 Retrofit 结合所带来的 Http 请求和业务数据处理；事件订阅和发布等。

RxJava 是一个强大的数据流水线，以数据为基础，提供了一系列任务调度，操作变换，事件处理等。

# 第二章 架构需求分析

## 2.1 需求目标

### 2.1.1 目标产品

本架构的目标产品是标准的 C/S 软件系统中的 Android 客户端产品，经过调查市面上百分之八十以上的 App 都属于此类型。

所以本架构需要分离出这类 App 共同的特征，并且为这些共同特征提供解决方案。

此外在部分非本类型的 App，通过架构裁剪和简单调整，本架构依然可以适用于这些 App。比如文件管理器类型的 App，可能并不需要联网通讯，则可以酌情裁剪掉网络请求组件。

### 2.1.2 目标用户(开发者)

本架构的目标开发者是中小型或者创业类型的团队。这部分用户在互联网创业热潮的今天占了开发者很大一部分比重。

这类团队人员往往较少，并且团队成员的经验多有不足。这些团队在接手项目的时候往往不知道怎么架构一个项目，转而去借鉴互联网上的一些开源项目，而这些开源项目有可能已经老旧过时。这样的话项目建设初期就没有打下一个良好的基础。

## 2.2 开发需求

### 2.2.1 开发期的功能需求

1. 开发者希望架构能够提供其开发所需要的各种公共组件，这样的话开发者可以将精力放在业务本身的开发上。

2. 开发者希望架构能使他们开发的业务流程更加清晰，这就需要我们对主体业务进行架构分层。而在大多数 C/S 架构的 Android App 中，主要的业务流程无非就是用户操作 -> 请求服务器 -> 处理返回数据 -> 显示给用户。这个是一类非常统一类似的业务场景，我们需要做的就是怎么让这个流程更加清晰规范。

3. 开发者不希望写大量的重复代码，在客户端开发中，开发者往往身陷于各种重复的状态检测，和业务调用前的准备工作:例如检查网络，检查权限，检查用户登陆状态，检查点击事件防止双击等等一系列的类似工作。所以开发者希望架构能帮助其解决这样问题。在本架构中我们使用 AOP 为开发者解决了这一难题。

4. 开发者往往要求平滑更加平滑的业务添加，在现今开发中，产品和设计为了提

高产品的竞争力往往高频率的提出各种新的需求，一个不成熟的架构往往会拖慢这一进度。小至一个界面的需求添加，大到一个完整的大型业务模块。这就要求我们对架构进行科学的业务切割，架构需要把握细粒度定义业务划分边界并管理他们之间的依赖和生命周期，这就要用到组件化了。在开发期间的团队协作中，代码的管理和团队人员的任务的管理也让人头疼，本架构引入模块让各个团队各司其职，工程开发期间互相不受影响。

### 2.2.2 开发期的质量需求

1.开发者希望其开发出来的代码具有高可读性，易理解，因为那些小团队往往人员变动频繁，新加入的成员如果能尽快的熟悉项目结构和代码，尽早的加入到正式开发之中对于中小型团队也会节省一笔不小的人员开支。这就要求架构思路清晰，容易被开发者接受，也要求架构具有高控制力，约束代码结构。当然也少不了详尽的说明文档，我觉得本文足以担此重任。

2.可扩展性和可维护性:开发初期，项目代码必然会面临着大量修改和调整；开发后期，项目会加入一些新的需求，为了保证修改和添加不会影响原有代码其他部分的正常开发，架构需要保证程序各个部分之间的松散耦合，架构可以使用事件驱动模式(发布/订阅模式)实现各个部分的通讯，而不是大量可能会变动的接口。

3.可重用性与可移植性:不难理解一个软件最核心的部分就是其业务逻辑，而其他的都是可变环境，包括平台环境和其他控制变量。为了保证业务的代码的独立即可重用可移植，必然需要对平台服务 api 做合理的抽象封装，为业务逻辑提供可靠的基础组件接口。最完美的状态就是在 App 层中看不到一点与 Android 平台或者其他运行环境 的 Api 代码。

4.可测试性:为保证程序的鲁棒性，除了 QA 需要对程序做自动化测试和人工测试，开发者自己也需要对程序做严格单元测试，而单元测试需要使用测试单元替代正式的业务单元做控制变量，检查出有 Bug 的错误单元。为了减少测试对业务代码的污染和侵入，架构对于单元之间的依赖除了要用接口做隔离，也需要使用 IOC 对依赖做统一拼装。这样就可以方便的替换测试单元。

## 2.3 用户级需求

### 2.3.1 用户级功能需求

本来用户功能需求只有在具体的系统设计时才会被描述的。但是作为 C/S 结构

App 客户端的通用架构，架构希望对用户的需求有所预测，本身当今的 App 就存在许多同质化的需求，相同的业务需求也可以帮助开发者节省开发成本。

比如用户登陆这样的基本需求，架构就为用户管理设计了 UerManager 组件。又比如编译环境在 Android 6.0 之上，向用户动态申请系统权限也是每个 App 必做的事情之一，架构同样也提供了 ospermisssion 组件。

### 2.3.2 运行期质量需求

1.性能需求:在 Android 平台上，软件性能主要表现在对主线程的保护上，原则上禁止任何在主线程上的耗时操作，除此之外也要注意优化 View 这些这些必须在主线程操作的任务的性能。为了减轻主线程的压力，架构必须让开发者方便的处理异步任务，这样开发者才更愿意将工作从主线程转移到子线程中。除此之外架构有必要为开发者提供性能监控和收集的功能，以便于开发者能够高效的为性能调优。

2.鲁棒性，即稳定性:除了架构本身的健壮性以外，架构必须提供一套高效统一的异常处理机制。除此之外需要提供对用户操作及输入的检查，避免非法的输入影响软件的稳定性。系统服务本身的状态也不容忽视，比如网络状态的检查，必须及时向 App 层告知系统资源的状态。

3.可伸缩性:如果一个 App 包含大量的功能，那么其中必然有很多功能是一部分用户所用不到的，那么这部分功能资源就对用户的设备资源造成了浪费。那么，架构就需要对功能组件实现按需加载，当用户需要用到该功能时再从网络上下载该功能模块并且动态加载。

## 2.4 需求总结

综上所述，本架构主要具备以下技术细节:

1.架构分层

2.组件化

3.模块化，容器化

4.依赖注入

5. AOP 面向切面

6.页面路由

7.事件驱动

下面我将根据架构的技术细节逐一具体分析。

# 第三章 架构整体设计

## 3.1 架构逻辑设计

### 3.1.1 架构图

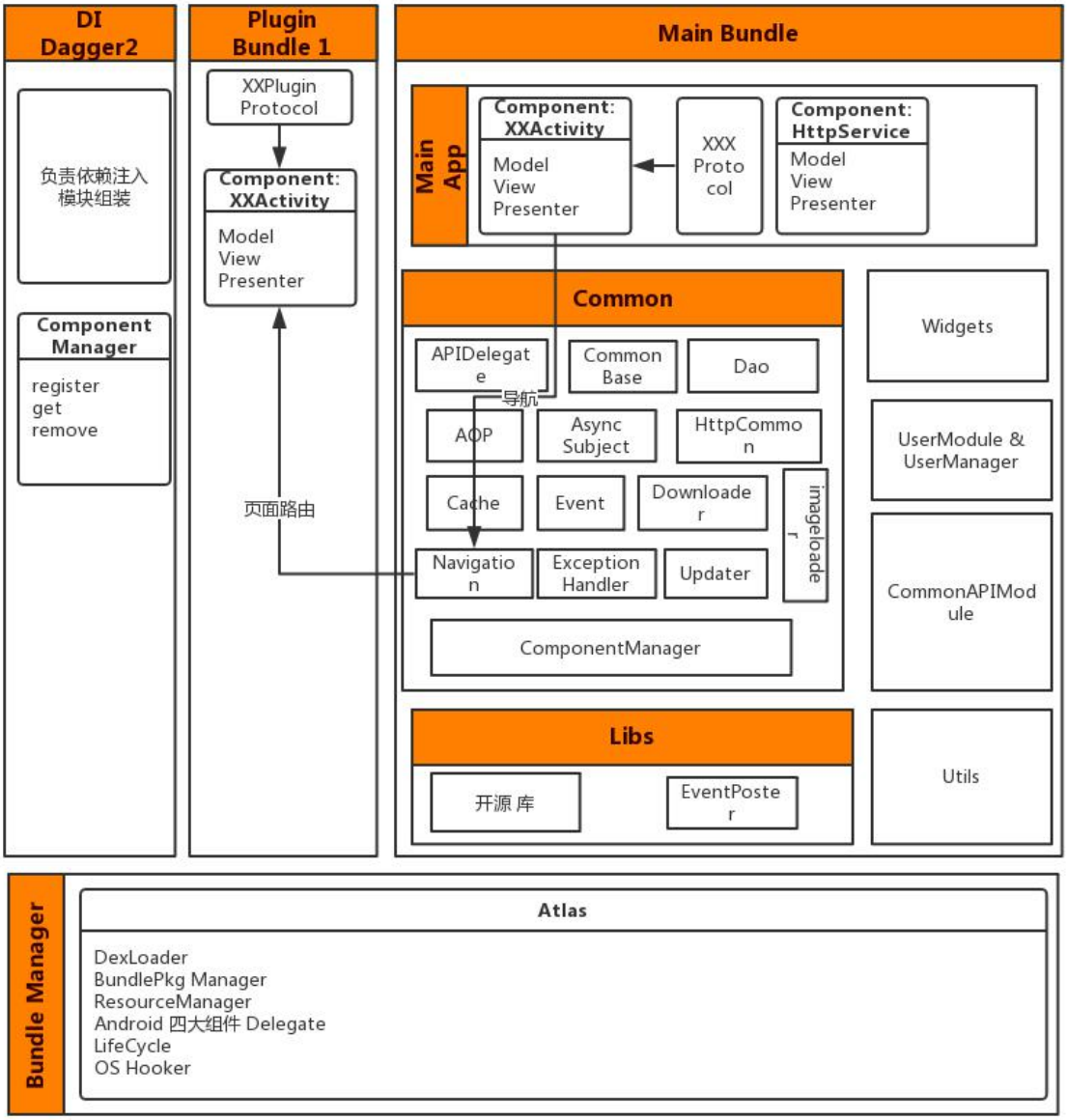



图 3-1 架构图

### 3.1.2 架构结构划分

如上面架构图所示:

1.BundleManager: 整个架构的底层是驱动层，主要负责 Bundle 组建管理。BundleManager 负责组件的动态加载，需要注意的是 BundleManager 本身是位于主模块的，随 App 首先被安装到用户设备上。BundleManager 提供了组件的动态按需加载。

2.主模块 MainModule: 主模块分为两部分，一是架构层，主要有模块化的底层驱动，框架层 Common 和一些其他的公共模块；二是主 App，里面包含了 App 的主体业务。

3.Common 包含了各种程序所需要的各种公共组件，和上层开发所需要继承的公共基类。需要注意的是 ComponentManager 组件管理器，上层可以通过组件管理器来寻找组件。

4.MainApp 是主 App 程序，在安装时被直接加载。

5.PlugBindle 是组件组件程序，将在用户需要的时候动态按需加载。

6.EventPoster 是事件驱动组件，可以处理自定义事件，View 事件，Activity 生命周期事件，以及 BroadCastReceiver 事件等，并且支持扩展。

### 3.1.3 架构特点

1.模块化，整个项目分为主 Bundle 和一些子功能模块，主模块包含了主 App 和公共库和组件。

2.组件化，包括公共组件，业务组件，和 MVP 组件。

3.AOP 编程，使用 AOP 进行用户管理，权限管理，性能监控，异步控制等。

4.使用页面导航进行页面跳转。

5.Java8，使用 Java8 lambda，函数指针，default 函数等新特性开发。

6.MVP 使用 MVP 分层架构。

7.依赖注入，使用依赖注入进行对象拼装。

8.页面组件路由。

9.事件驱动。

### 3.1.4 架构核心 Common

Common 模块是整个架构的核心，参考上面的架构图 1-3-1，Common 提供了架构必须的各种公共组件包括组件管理器。

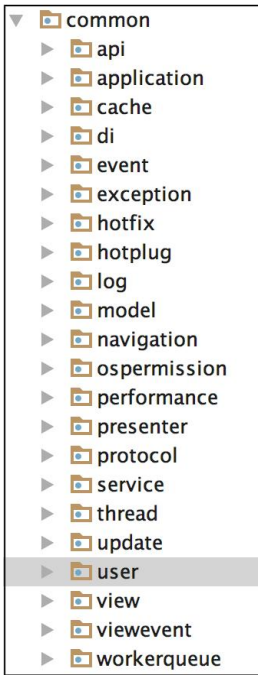



图 3-2 Common 包结构

1.api: 提供了 http api 的实现代理，并提供了 Retrofit 的默认实现。

2.cache: 缓存，包括 KV 缓存公共接口，Lru 缓存实现，和 ACache 的实现封装。

3.di: 依赖注入: 包含 App 全局组件的依赖接口和默认配置，组件管理器，组件生命周期 Scope，除此之外还有 Base 的 API 服务组件。

4.event: 事件驱动抽象接口，默认使用 EventPoster 实现。

6.exception: 异常处理，对异常的捕获，归类，和转换。包含自定义的业务异常，和异常处理引擎。

7.hotfix: 热更新，热修复，默认使用 Atlas 实现。

8.model: 模型层，基础的模型层 BaseModel，封装了异步和回调处理，Http 异常处理，缓存，公共返回值转换器，IamgeLoader 抽象接口等。

9.navigation: 页面导航封装参数栈。

10.ospermission: 动态权限检测切面。

11.performance: 性能监控切面。

12.presenter: presenter 层基础类。

13.protocol: 基础协议层，主要封装 MVP 三层之间的公共协议接口。

14.thread: 异步装饰器，修饰异步方法。

15.update: 程序更新组件。

16.view: View 层基础类，包括 Fragment，Activity，数据 adapter 和一些公共接口。

17.workqueue: 异步工作队列，实现消费生产模型。

## 3.2 项目开发期间架构设计

开发架构设计主要描述了工程开发期间的开发模式，和规范约束。

### 3.2.1 工程结构

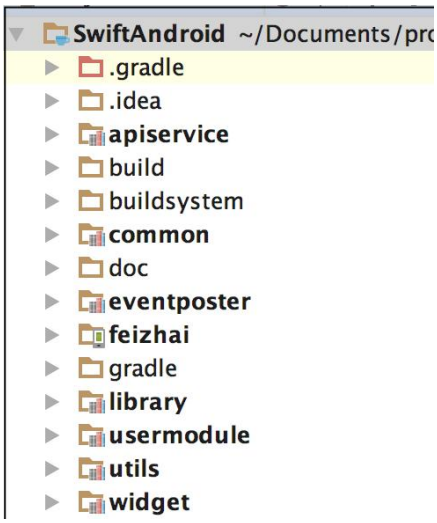



图 3-3 工程结构

1.common: 架构核心，提供各种公共组件和封装。

2.buildsystem: 编译脚本，记录了库依赖，编译规则，混淆规则。

3.eventposter: 自己实现的事件驱动框架。

4.library: 框架依赖集合。

5.utils: 公共工具类集合，如图像工具，手机信息工具，单位转换工具等。

6.apimodule: api 服务模块，含有公共 api，和 api 管理模块。

7.usermodule: 用户管理公共模块。

8.feizhai: 主 App 模块，app 的主体。

### 3.2.2 使用本架构构建 App

拷贝整个工程到工作目录，新建自己的 App 模块，依赖 Common 等公共模块。在 Application 中初始化 BaseApplication，或者直接继承 BaseApplication。

实例 App 结构:

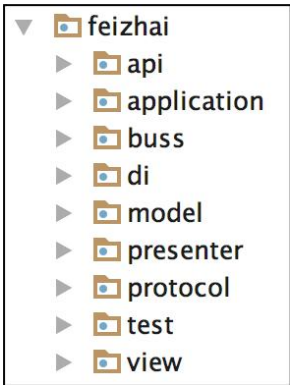



图 3-4 实例 App

1.使用 ComponentManager 获取包括 AppComponent 的公共组件，通过公共组件调用各种公共服务，包括 Cache，Downloader，Thread 等。（具体参考 3.6/4.4.4）

2.如果创建业务 API，先写业务接口，然后在 di 中创建 APIComponent，Module 继承于 BaseAPIModule，将提供 API 接口解析实现，Session 管理，Cache 等服务。（具体参考 3.5）

3.如果新增页面/业务，则先增加协议，编写 MVP 三层业务接口并且增加 MVP 三层，分别继承自 BaseM/V/P，最后在 di 中新增 MVP Component。（具体参考 3.4）

4.新增子模块 Bundle: 参考 Atlas/Small 使用。

### 3.2.3 编译期间依赖关系

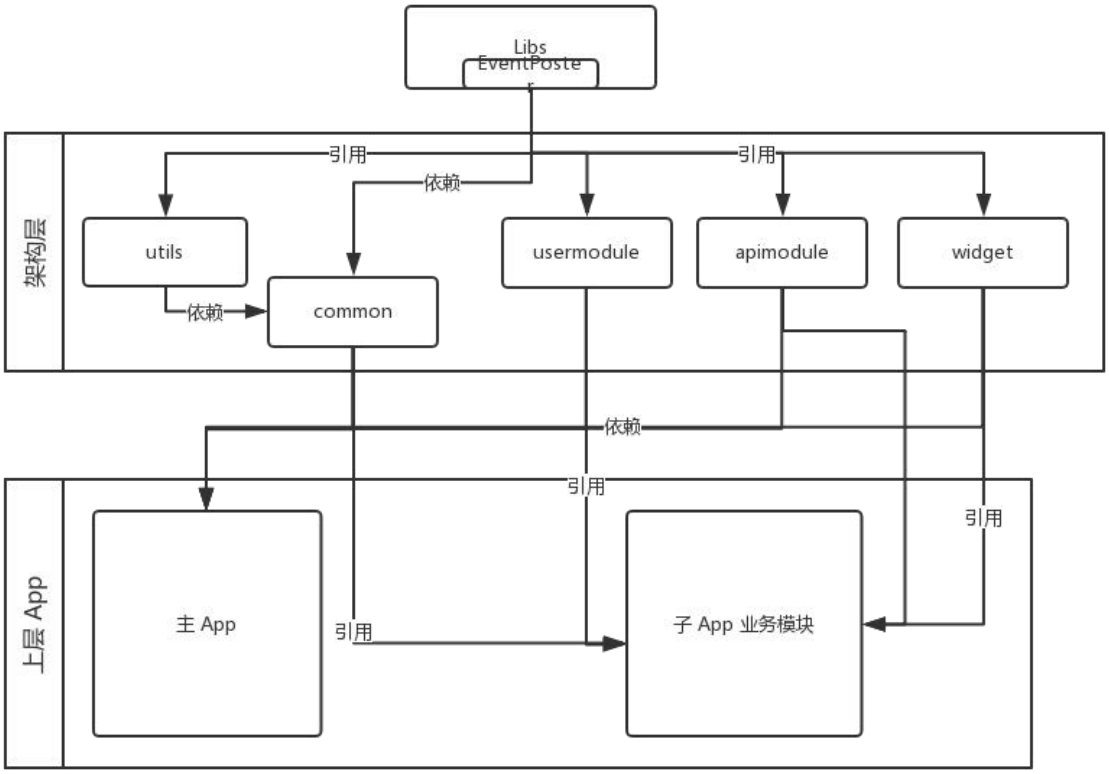



图 3-5 编译期间依赖关系

如上图所示，编译期间 Libs 和架构层以及主 App 模块都被编译进主模块中，生成主 Apk。

子业务 App 模块则只是引用 Libs 以及架构层，编译的子模块并不讲他们编译进 Bundle 包。

### 3.2.4 模块隔离开发

在开发期间，各个团队负责一个模块的开发维护，当完成一个版本之后，将模块单独打包 Publish 到公共仓库(Maven,JCenter)，其他团队如果需要依赖该模块，则在 build.gradle 中引用该模块。

上级模块引用下级模块层模块，例如 App 层引用架构层可以直接使用强类型引用，必须严格遵循面向接口。

同级 App 层之间引用:1. 可以使用暴露的业务接口。2. 使用页面路由或者组件别名进行“软引用”。

## 3.3 运行期间架构

### 3.3.1 运行期间的模块加载升级

Apk 安装之后首次打开，主模块被系统加载，主模块包含了主 App 和 架构层以及诸多类库。

当用户需要打开子功能模块的内容时，主模块请求云端，下载最新的子业务模块，BundleManager 加载该下载完成的子模块。使用页面路由跳转到子功能模块。

模块可以随时动态升级，当升级管理器从云端发现新版本的模块后，管理器将从云端下载最新的模块以动态替换掉老旧模块。

### 3.3.2 运行期组件依赖

组件从逻辑和运行期间都是树形依赖的，在运行期间首先加载的是 AppComponent 即 App 全局组件，在 Application 的 OnCreate 中初始化。

底层业务组件在一个业务集合开始时被初始化，比如说 UserApiComponent 就是在用户登陆时被初始化的，以供上层需要用到 User 状态和信息的 MVP 组件调用。

上层组件初始化时，需要传入所需依赖已经初始化好的底层组件，底层组件的引用会被该上层组件持有直到该上层组件生命周期结束。在该上层组件作用域期间，其可以调用他所依赖的所有底层组件的资源，而且这个过程是传递的，即树的叶子节点可以使用其所有父节点的资源。

### 3.3.3 页面导航和业务服务导航

编译期间，所有打上“服务/页面”注解的类型引用及和 TAG 的映射关系都会被注解处理器收集起来，在程序初始化时注册到服务注册表中。

当通过页面路由跳转页面时，路由器根据 TAG 在注册表中找到该页面跳转。同样的，在寻找某个服务类型时，服务管理器根据 TAG 或者类型找到该服务的类型引用，并初始化返回服务的实例。

# 第四章 分层架构

## 4.1 分层概述

分层架构是架构设计中的重要组成部分，分层架构是所有架构的鼻祖，分层的作用就是隔离，使程序的结构更为立体。

分层架构的好处:

1.它一定程度上分离了关注点，将业务流程分层，利于各层逻辑解耦和的重用；

2.它规范化了层间的调用关系，可以降低层与层之间的依赖；

3.如果层间接口设计合理，则用新的实现来替换原有层次的实现也不是什么难事。

常见的分层架构有这几种：MVC, MVVM, MVP。

## 4.2 MVC 分层架构

### 4.2.1 MVC 简介

Model-View-Controller: Model：逻辑模型，View：视图模型，Controlelr：控制器。

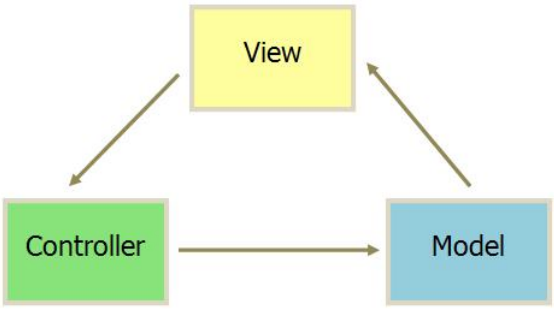



图 4-1 MVC 结构

如图 4-1 所示在 MVC 的模式下，程序流程的层次被划分为 3 层：模型（Model）、视图（View）、控制器（Controller）。Model 负责建立数据结构以及相应的行为操作处理。View 对象负责在视图上渲染出相应的图形文字信息展示给用户看。Controller 对象负责接收用户的按键和触摸等操作事件，协调 Model 层和 View 层。

用户与 View 层 交互，View 层接收并反馈用户的操作；View 把用户的操作传给相应的 Controller，由 Controller 决定调用哪个 Model，然后由 Model 调用相应的业务逻辑对用户请求并加工处理，如果需要返回数据，Model 会把相应的数据返回给 Controller，由 Controller 调用相应的 View，最终由 View 格式化和渲染返回的数据。

### 3.2.2 Model 模型层

对于 Model 层来说，还可以分为两个部分，一个是描述业务数据结构的 Bean，即各种业务的实体类，二是对数据获取的封装:

1.从 Http 请求来说，Model 一般封装了 Http 请求的 url 填充，请求方式设置，参数拼装，请求头拼装(包括 Session，Cookies 的处理)。对于请求的返回，如果返回正常数据，即 2XX 状态，则需要在 Model 里面处理返回的结构话数据：这个数据格式可能是 JSON 或者是 XML，可以使用 ORM 映射 lib 或者手动将 JSON 转换填充到 Model 传给上层处理。而当请求返回失败时(可能是网络错误，也可能是业务上的错误)，就应该抛出对应的异常交给上层处理。

2.从 DB 数据库来说，也叫 DAO 层，Model 需要封装 SQL 语句的拼装，当然也可以使用 ORM 数据库关系映射来组装 SQL 例如 OrmLite，GreenDao 等，或者使用一些 NOSQL 数据库，在移动端一般是 REALM 为代表的新型数据库。

当子线程请求到数据之后，Model 层就可以控制 View 去显示数据了。

Model 的代码:

1.参数拼装:

```
public GetAvlMeterModel() {
    super();
    Map<String,String> par = new HashMap<String,String>();
    par.put("start","0");
    par.put("limit","9999");
    par.put(Config.Par.TOKEN_ID,Global.user.getSid());
    httpService.setParams(par);
}
```

代码段 4-1 参数拼装

2. 返回值处理

```
@Override
public Serializable dealReturn(String result) throws HttpServiceException {
    List<AvlMeterBean> res = new ArrayList<AvlMeterBean>();
    try {
        JSONObject object = new JSONObject(result);
        if (object == null)
            throw new JSONException("返回错误");
        JSONArray dataArrary = object.getJSONArray(Config.JSONConfig.detailBase.DATA_KEY);
        if (dataArrary == null || dataArrary.length() == 0)
            throw new JSONException("返回错误");
        for (int i=0;i < dataArrary.length();i++){
            JSONObject obj = dataArrary.getJSONObject(i);
            AvlMeterBean bean = new AvlMeterBean();
            bean.setMeterid(obj.getString("yb_id"));
            bean.setName(obj.getString("yb_name"));
            res.add(bean);
        }
    }catch (JSONException e) {
        e.printStackTrace();
        throw new HttpServiceException(e.toString());
    }
    return (Serializable) res;
}
```

代码段 4-3 返回值处理

### 4.2.3 Controller 控制器

Controller 负责调用 Model 请求数据，并且接收 View 的事件回调。

在 Android 中，Controller 一般就代指 Activity。Activity 需要处理大量的回调，扮演着活动发起者的角色，大部分的事件都是在 Activity 中触发的。对于 Actiity 一般的项目会有一个 BaseActivity 的封装，在没有异步请求框架的时代，BaseActivity 除了要封装例如 initView，initData，showXXX，dismissXXX，这类公共函数之外，还需要 封装一套 Handler 以处理异步 Http 请求。

### 4.2.4 View 层

对于 Android 平台来说，View 层就是布局文件 xml，xml 作为 Dom 树形描述语言，在 Android 主要用来描述 Android 视图的层级结构，类似 Html 一样。这样 xml 作为 View 层的话，控制力实在太弱了。所以仍然有很多对 View 的处理，如动画等是放在 Activity 内用 Java 代码实现的。

### 4.2.5 MVC 所存在的问题

这个问题需要从 Android 架构设计本身说起。

对于 Activity 的这个角色，他似乎介于 View 和 Controller 之间。所以内部难免会有大量逻辑耦合。其实这也和 Android 系统本身的设计有关，在 Android 中，每个活动片段被设计为一个 Context，即一个上下文/场景/交互。

Activity 是 Context 的实现类，Activity 直接继承于 ContextThemeWrapper，不同点

在于 Activity 包含了大量与 UI 操作相关的逻辑，这就是为什么 Activity 总是被当作上帝类的原因了。Activity 实在太重要了，但是他也太笨重了，Activity 包含了程序运行所需要的的各种驱动事件，Activity 的生命周期，按键的监听，这些都是逻辑运行的起点。

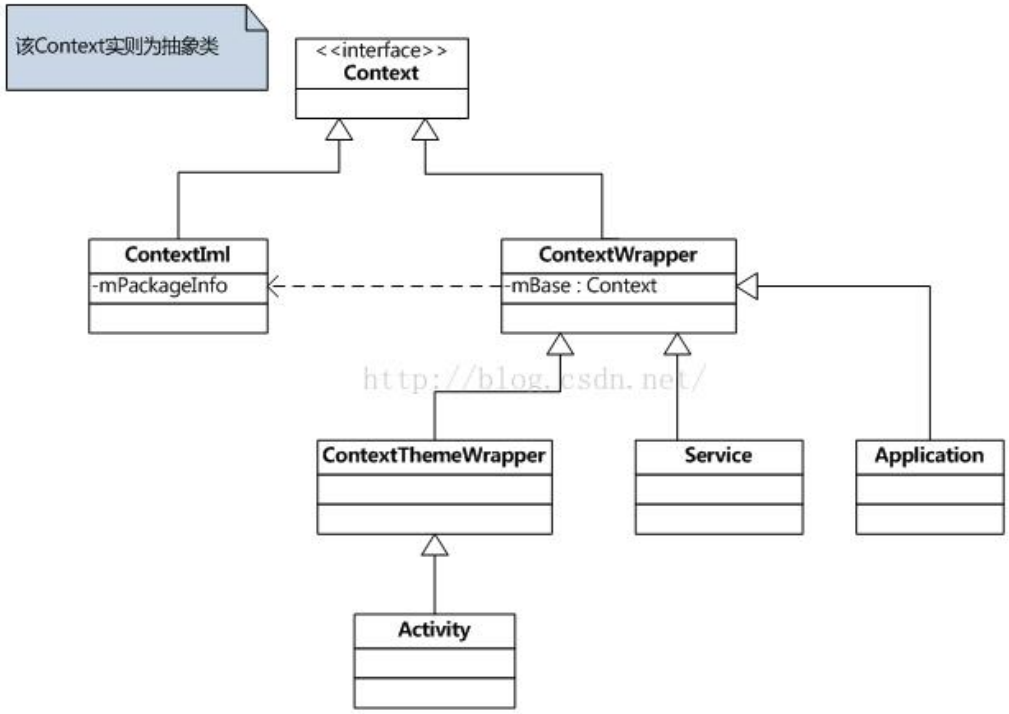



图 4-2 Context 继承关系

所以说，作为 Android 四大组件中最重要的一环，Activity 的地位十分重要，其本身系统架构所赋予的功能就很多了，所谓功能越多任务越重，Activity 的代码难免会大量膨胀。

不难想象，大量代码在一个类中膨胀会造成什么后果，不知不觉，Activity 就会称为一个上帝类。

程序员要避免创建神类。避免创建无所不知，无所不能的上帝类。如果一个类要花费大量代码从其他类中通过 Get() 或者 Set() 方法做数据交互（也就是说，需要深入挖掘业务并且告诉它们如何去做这些事情），所以应该把这些功能函数分离到其它类而不是都塞到上帝类中。

上帝类具有很高维护成本，人们很难理解上帝类中正在进行的复杂操作，并且难以进行测试或者扩展，这就是为什么要避免创建上帝类的原因。

然而，在实际的开发中，如果你不使用科学的架构分层，Activity 类必然会越来越

臃肿不堪。因为，在 Android 中，允许 View 和其它线程共存于 Activity 类内。Activity 中同时存在业务逻辑和 UI 操作逻辑， 这就是问题产生的原因。这会增加测试或者维护的成本。

## 4.3 MVVM 分层架构

### 4.3.1 MVVM 简介

Model-View-ViewModel:

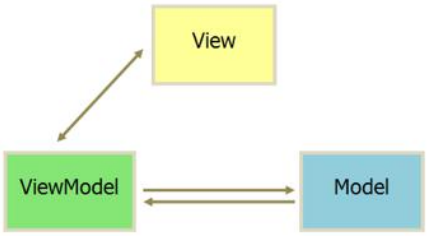



图 4-3 MVVM 结构

相对于拥有的悠久的历史的 MVC 架构，MVVM 架构可谓是一个相当年轻的架构，MVVM 最早在 2005 年由微软 WPF 和 Silverlight 的架构师 John Gossman 提出，并且大量应用在微软的软件开发中。当时 MVC 架构已经被提出了 20 多年了，可见两者出现的时间差有多大。

MVVM 在使用当中，通常还会用到双向绑定技术，这项技术使得在 Model 变化的同时，ViewModel 也会自动保持同步，而在 ViewModel 变化时，View 也会自动变化。所以，MVVM 模式又被称作：model-view-binder 模式。

具体在 iOS 中，可以使用 KVO 或 Notification 技术达到这种效果。在 Android 中，MVVM 是由 Android 官方的 DataBinding 框架实现的。

### 4.3.2 MVVM 各层

Model: 负责数据实现和逻辑处理，类似 MVC 中的 Model 。

View: 对应于 Activity 和 XML，负责 View 的绘制以及与用户交互，类似 MVC 中的 View，或者是 JSP 代码。

ViewModel: ViewModel 负责描述 View 与 Model 之间的关联，将 model 和 view 绑定起来。这样的话，model 层 数据的变化时，数据绑定框架会按照 viewmodel 的逻辑反馈给 view。view 的 xml 布局文件，经过特定的编写，编译工具处理后，生成的代

码会接收 viewmodel 的数据通知消息，自动刷新界面。

### 4.3.3 DataBinding 原理简介

Android 中的 DataBinding 的原理类似 JSP 代码。我们先来看一段使用了 DataBinding 的布局 XML 文件。

```
activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="user"
            type="mvvm.wangjing.com.mvvm.User.User" />
    </data>
    <RelativeLayout
        android:id="@+id/activity_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="mvvm.wangjing.com.mvvm.MainActivity">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:onClick="@{user.onItemClick}"
            android:text="@{`My name is `+  user.name+`  I'm   `+user.a
    </RelativeLayout>
</layout>
```

代码段 4-3 Databinding 例子

和 JSP 一样，在 XML 语言中插入了类似 Java 的脚本代码段。这些代码段在编译期间会被对应的 APT(编译处理器) 读取处理，APT 会自动生成相应的 Binding 类，其中就包含了写在 XML 中的逻辑代码。

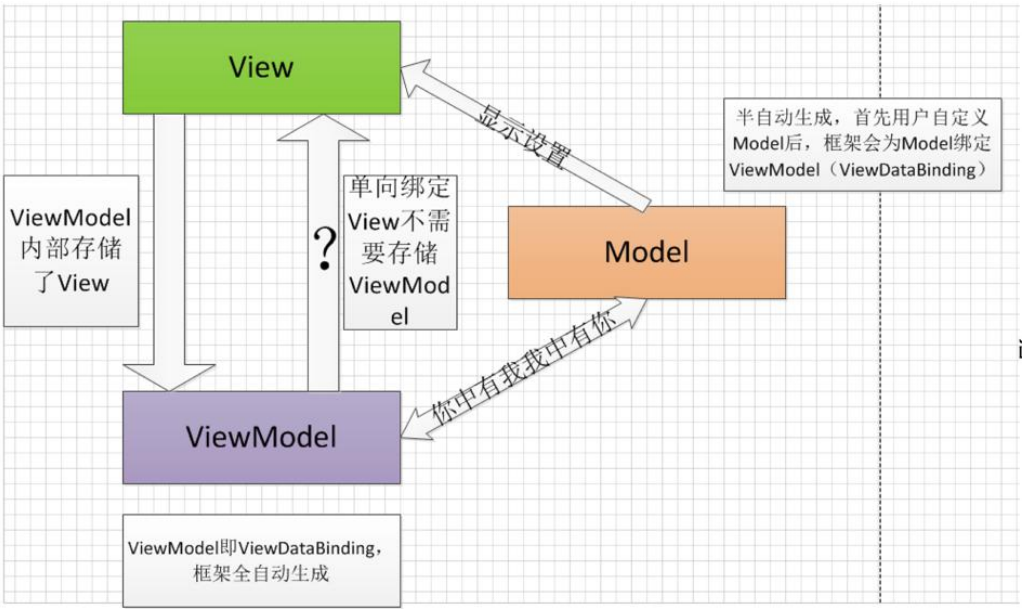



图 4-4 MVVM 结构图

简单看一下自动生成的代码:

```
ViewDataBinding文件是框架根据layout文件和model自动生成的，无法修改。TuanOrderCommonI
public class TuanOrderCommonItem2Binding extends android.databinding.ViewData
this.check = (android.widget.CheckBox) bindings[1];//存储Layout中的控件
private com.dianping.tuan.Model.OrderListItemModel mItemMode; //存储Model数据
// read checked~.~ itemMode ~
  if ( itemMode != null)
    {
       checkedItemMode = itemMode.isChecked();      //获取ItemMode中的数据
    }
   if ((dirtyFlags & 0b11L) != 0)
    {
        this.check.setChecked(checkedItemViewMode);        //将数据赋值给控件
    }
    ……
```

代码段 4-4 自动生成代码

### 4.3.4 MVVM 缺点

1.目前只支持单向绑定。

2.ViewModel 与 View 一一对应。

3.使用起来灵活性比较低。

4.Model 属性发生变化时，ViewDatabinding 采用异步更新数据，对于现实大量数据的 ListView，会有一定延迟，在实践测试中发现，Databing 效率较低，对于负责的界面不太适用。

5.编译器会自动生成大量代码，其中包括各种属性，字段：ViewDataBinding 实现类 DataBinderMapper 等。

6.最后也是最重要一点，DataBinding 需要在 XML 插入 Java 逻辑代码，这样不仅会污染 XML，也会让 Java 逻辑孤立。除此之外，这个模式的实现完全依赖框架本身提供的功能，其控制力不足，只适用于基本的 数据 –> View 的场景，不适合用于业务复杂的场景。

## 4.4 MVP 分层架构

### 4.4.1 MVP 简介

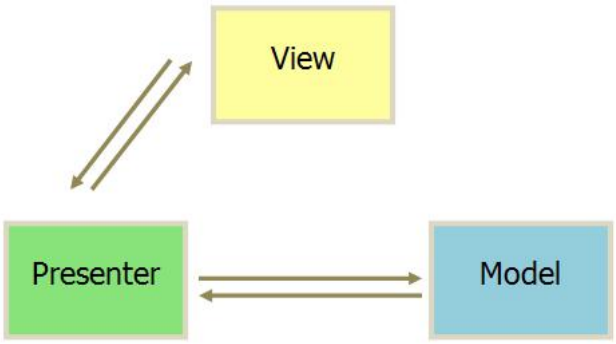



图 4-5 MVP 架构

MVP 是 Mike Potel 在 1996 年的时候提出来的，在一篇论文中，他在 MVC 变体 1（那个时候，控件系统早已成熟）的基础上，提出了 MVP 架构，与 MVC 变体 1 最大的区别在于：Presenter 是一个总控制系统，取代了众多的的 Controller 们。其原话：we refer to this kind of controller as a presenter.Presenter 是一个特殊的 Controller，它与经典版 MVC 中的 Controller 差异更大：

1.它是总控系统。

2.它接收来自 View 层的消息。

### 4.4.2 Model 层设计

MVP 的 Model 类似 MVC 的 Model，其不同的地方在于，MVC 的 Model 在请求完成之后会把数据直接抛给 View 层，而 MVP 则是送给 Presenter 层加工后再送到 View 层直接显示。

Model 和 MVC 不同，他不会持有 View 层，但是 Model 会持有来自 Presenter 调用时候传过来的回调。

### 4.4.3 View 层设计

MVP 的 View 层将专职于 View 的显示，Activity 中只保留 UI 相关的代码，基本只会保留 get，set 接口,所有的业务逻辑都会被抽取到 Presenter 中。和 MVC 一样，MVP 架构的 App 一般也会有一个 BaseActivity，除了子类需要的公共函数外，对 Presenter 的生命周期管理也在其中。

View 会持有 Presenter，在 Activity OnCreate 时对 Presenter 初始化，调用 Presenter 的初始化接口，并且在 OnDestroy 的时候 调用 Presenter 的 destroy 接口以销毁 Presenter。

### 4.4.3 Presenter 层设计

Presenter 取代了原本 MVC 中 Activity 的位置。他同时持有 Model 和 View，从 Model 请求数据，然后对数据处理，根据结果调用 View 显示界面。

Presenter 层连接了 Model 层和 View 层，Presenter 层主要负责对业务数据进行处理。在 MVP 架构中 Model 层和 View 层无法直接交互。Presenter 层会持有 Model 它会从 Model 层发起数据请求，当数据获取完毕时通过 Callback 返回 Presenter，Presenter 再去处理数据并通过 View 层显示。这样的话因为 Presenter 层将 View 层与 Model 层隔离，使得 View 层和 Model 层之间不存在耦合，Presenter 成为了一个业务逻辑主体的角色，与此同时也将业务逻辑从 Activity 中抽离了。

### 4.4.4 MVP 的统筹 协议层设计

在 MVP 架构中，协议将这三层分别抽象到各自的接口中,这个接口集合包含三个内部接口分别为 IView，IModel 和 IPresenter。通过接口将各层之间隔离，而 Presenter 对 View 和 Model 之间相互依赖也就等于依赖了各自的接口。这方面符合了接口隔离原则，也符合面向接口原则。

在 Presenter 中层中包含了一个 IView 接口和一个 IModel 接口，Presenter 同时持有协议中 M/V 两部分，从而将 Model 层与 View 层联系起来。而对于 View 层则会持有一个 IPresenter 接口并且只保留对 Presenter 层接口的调用，具体业务逻辑全部放倒 Presenter 层接口实现类中处理。

以下就是一个简单的协议。

```
public interface LoginProtocol {
    interface View extends BaseProtocol.View, BaseProtocol.ProgressView {
        void showLoginSuccess();
        void showLoginFailure(String msg);
    }
    interface Model {
        IAsyncSubject login(String name, String pass, AsyncCallback<UserInfo> callback);
    }

    interface Presenter {
        void login(String name, String pass);
    }
}
```

代码段 4-5 协议类

这个是一个登陆的协议。

注意继承架构层中的基础协议:

```
public class BaseProtocol {
    public interface ProgressView {
        void showProgress();
        void dismissProgress();
    }
    public interface View {
        void showMessage(String string);
        void lockUI();
        void unLockUI();
    }
    public interface Presenter extends IAsyncSubjectsQueue {
        void onViewInited();
        void onViewDestroyed();
    }
    public interface Model {
        void destroyRequests();
    }
}
```

代码段 4-6 基础协议

基础协议中:

1.基础 ProgressView 描述了带进度显示的 View。

2.基础 View 描述了，消息显示，锁定/解锁 UI。

3.基础 Presenter 主要暴露了 View 的初始化和销毁，以便于在 Presenter 层中收到业务页面的生命周期回掉，以初始化/销毁资源。IasyncSubjectQueue 封装了异步工作队列的抽象，包括添加异步任务，销毁异步任务。

### 4.4.5 MVP 的优缺点

优点:

1.降低耦合度。

2.模块职责划分明显。

3.利于测试驱动开发。

4.代码复用。

5.隐藏数据。

6.代码灵活性。

缺点:

1.接口膨胀，接口细粒度不好把握。

2.代码量增加，对于业务简单的场景有点得不偿失。

### 4.4.6 MVP 与 MVC 的不同。

MVP 与 MVC 不同点在于，Presenter 把 View 和 Model 完全隔离开来，Model

不可以持有 View，View 不可以持有 Model，View 与 Model 之间的交互完全由 Presenter 控制。

# 第五章 组件化

## 5.1 组件化介绍

软件的组件化由来已久，组件，又叫 Component，一般是某个功能所在的容器。组件化就是以重用软件单元的目的，将一个的软件系统按分离关注点的形式，拆分成多个独立的单元，减少耦合。

组件可以是一个逻辑页面 (Activity/Fragment) 组成的 MVP 单元，也可以是一些基础的服务单元例如 Cache，也可以是一些特殊的业务，例如某一组 API 的实现。

## 5.2 本架构中组件的实现

在我的架构中，Component 是比较细粒化的，为了和后面的 Bundle 模块，也是容器区分开来，他们有不同的职责，Component 将较为频繁的在横向上即业务上分解项目的结构。

在 Android 中:

一个 MVP，即一个页面(上面 MVP 中所说的一个协议)包括其衍生的三层架构是一个 Component。

一个业务功能块，如用户管理，这个组件可能被其他多个组件所依赖，所以他是一个全局的公共 Component。

一组业务 API，这组 API 也可能被众多其他业务所使用，所以这是个公共的业务组件。

AppComponent App 全局公共组件，里面可以包含各种包括 Cache，Downloader，ImgLoader，ThreadPool 等等公用的工具。当然，AppComponent 也可以被进一步更具功能拆分，但是应为其生命周期都是全局单例，也没有拆分的必要。

## 5.3 Component 的依赖

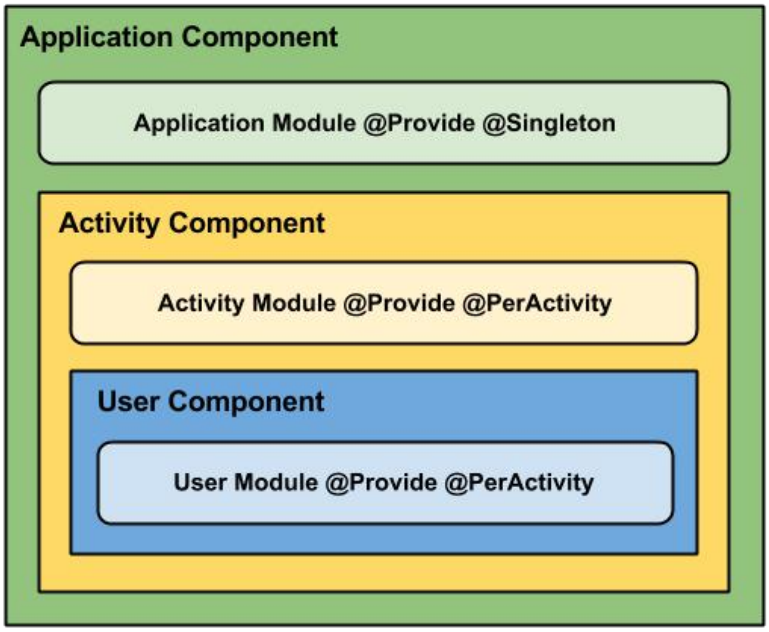



图 5-1 组件依赖关系

Component 之间必然是存在依赖的，比如说登陆组件(MVP)需要依赖 API 组件，API 组件需要依赖组件中的 Cache，和 API Factory。

组件之间的依赖必须是单向的，如果双向以来就会出现循环依赖，这样就会破坏依赖的树形结构，生命周期也将不好管理。

所谓组件间的依赖关系有点类似类的继承结构。

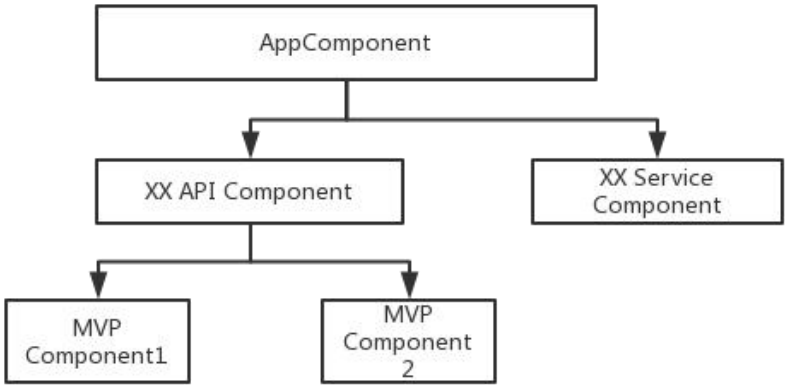



图 5-2 本架构中组件依赖关系

## 5.4 Component 的生命周期

既然组件之间存在属性依赖关系，那么组件的作用域也是不同的，比如说

AppComponent 的作用域就是全局单例，任何其他组件都可以依赖和引用 AppComponent。API Component 的作用域一般是某个业务场景下的子组件，比如在登陆之后，和 USER 相关的 API 组件就开始了自己的生命周期，USER 组件可以被多个和用户相关的页面(MVP 组件)依赖或引用，但是在退出登陆之后，USER 组件的生命周期也就结束了。

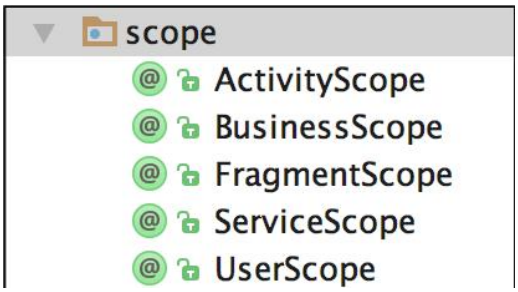



图 5-3 本架构定义的生命周期

在本架构中一共定义了 6 种生命周期:

1.Singleton: 全局单例，这个是默认生命周期。

2.ActivityScope: 生命周期跟随 Activity，一般下面所说的 MVP Component 使用这个。

3.BusinessScope: 业务的生命周期，由自定义的业务组件管理，对应下文的 API Component。

4.FragmentScope: 和 ActivityScope 类似，对应 Fragment 作为 View 层时所使用的 Tag。

5.ServiceScope: 跟随 Android 的 Service 组件。

6.UserScope: 跟随用户的生命周期，当某用户登陆是开始，登陆失效或者登出时结束。

## 5.5 MVP Component

为什么要将一个 MVP 单独作为一个 Component，首先一个 MVP 三层结构在业务上是较为独立的，在程序中是一个单独的作用域。MVP 内部三层之间相互调用。

在生命周期上，Activity 的周期由系统框架管理，Activity 开始的时候就会带动组建内其他两层 Model 和 Presenter 启动。也就是说每个 MVP Component 的生命周期与作用域是由其 View 也就是 Activity(也可以是 Fragment)来管理的。

```
public class LoginModule {

    private LoginProtocol.View view;
    private LoginProtocol.Presenter presenter;
    private LoginProtocol.Model model;

    public LoginModule(LoginProtocol.View view) {
        this.view = view;
        this.model = new LoginModel();
        this.presenter = new LoginPresenter(model, view);
    }
    @Provides
    @ActivityScope
    public LoginProtocol.View provideView() {
        return view;
    }
    @Provides
    @ActivityScope
    public LoginProtocol.Presenter providePresenter() {
        return presenter;
    }
    @Provides
    @ActivityScope
    public LoginProtocol.Model provideModel() {
        return model;
    }
}
```

代码段 5-1 依赖拼装图

以上是登陆界面 MVP Component 三层的拼装，其涉及的 Dagger2 将在下面分析。

## 5.6 API Component

在 App 中，一些业务页面经常需要依赖一组特定的 Http API，在一个作用域中，API 只需要一次实现，一个 HOST，一个 Session，一个公共返回值处理器，一种缓存策略等。

```
@Component(modules = FeiZhaiAPIModule.class, dependencies = AppComponent.class)
@ServiceScope
public interface FeiZhaiAPIComponent {
    void inject(BaseHttpModel model);
    void inject(BasePresenter presenter);
    IFeiZhaiAPI getAPI();
    Session getSession();
}
@Module
public class FeiZhaiAPIModule extends BaseAPIModule {
    public FeiZhaiAPIModule() {
        init(Constant.FEIZHAI_URL);
    }
    @Provides
    @ServiceScope
    public IFeiZhaiAPI provideFeizhaiAPI() {
        return apiGenerator.getAPI(IFeiZhaiAPI.class);
```

```
    }
    @Provides
    @ServiceScope
    public Session provideSession() {
        return generateSession();
    }
}
```

代码段 5-2 API Component

如图 3-5-1 其他组件通过这个 APIComponent 可以拿到这个 API 的实现与 Session。

## 5.7 AppComponent

### 5.7.1 AppComponent 含义

全局 AppComponent 是一个 App 运行的必要组成部分，一个 App 必然需要 Cache 组件，图片异步加载组件等一些公共组件，这类组件都是全局单例的。

Android 中，这些组件在 Application 的 OnCreate 时初始化并且作用于 App 全域，服务于其他所有的组件。

我们把公共组件抽象成接口，更具配置对接口进行依赖拼装，其他组件调用公共组件应该通过接口而不是直接通过实现类，应为大多数情况下，公共组件都是用了第三方库或者 SDK，在 App 开发维护漫长的生命周期中，第三方库是有可能更换的，所以有必要在中间抽象一层。

```
/**
 * App 全局依赖的组件
 * Created by ganyao on 2016/10/26.
 */
@Singleton
@Component(modules = {AppModule.class})
public interface AppComponent {
    void inject(Object object);
    void inject(BaseHttpModel model);
    void inject(BasePresenter presenter);

    Context globalContext();

    ConcurrentHashMap globalData();

    IAPIGenerator apiGenerator();

    IKVDiskCache kvDiskCache();

    ACache aCache();

    Handler mainHandler();
```

```
    IAsyncSubjectsQueue generateSubscriber();

    IBaseHttpModel getBaseHttpModel();

    IImageLoader imageLoader();

    IEventHub eventHub();

}
```

代码段 5-3 AppComponent

这样的话，在 Application 初始化时，传入具体的公共组件实现。

### 5.7.2 各个基础组件

1.依赖注入组件

- 架构中各个组件使用 DI 进行依赖的组装以解耦各个组件之间的引用依赖。

- MVP 三层间使用 DI 进行生命周期以及依赖拼装。

- 各个组件间使用接口进行抽象，并且实现使用 DI 进行组装，实现解耦。

- 一些依赖使用 DI 进行组装管理，并自动注入到高层组件需要的上下文中。本架构中，依赖主要有以下 3 种：

- 业务依赖：包括 Http 接口，Dao 接口等。生命周期大多为全局单例，所以存储于 AppComponent 中的全局容器，依赖 ServiceModule。

- App 全局依赖：包括工作的线程池的实现，全局的 Context，以及 Session，User 相关的存储等，生命周期是全局单例，存储于 AppComponent 全局容器中，依赖 AppModule。

- 与各个 Activity/Fragment 作用域绑定的依赖。生命周期与各个 Activity 一致，每个 Activity/Fragment 和它对应的 Presenter, Model,Fragment 绑定在一起。

2.数据流水线 RxJava

- 使用 RxJava 将网络请求打包发布到 IO 线程池中，再于主线程中订阅网络请求的处理结果。

- 使用 RxJava 的 CompositeSubscription 对订阅进行集中管理。在 View 销毁时取消 Presenter 层 Callback 对网络请求结果的订阅，以避免造成内存泄漏。

- 使用 RxJava 进行异步任务调度，减少线程与 Handler 的相关代码。

- 使用 RxJava 的 map 操作符对公共返回数据进行处理，统一抛出业务异常到上层进行处理显示。

3.网络请求 Retrofit2 + OKHttp3

- 基于 Retrofit2 框架，使用 GSON 解析器解析数据模型。

- 使用 OkHttp3 拦截器对需要统一处理的请求进行拦截，加入公共请求参数。

4.组建间通讯组件 RxBus 或者 EventBus

- 使用 EventBus 进行组件间或者层级间通讯。

- 考虑到 EventBus 需求可能较少，大多数事件的订阅/发布工作可以用 RxJava 代替，如果遇到相对分散的事件，则可以用 EventBus 代替。

5.数据持久化组件

- 使用 Realm 数据库，支持持久化数据到对象的双向转换，代替 sqlite 和 SharePreference 存储一些类似应用配置或者用户信息的持久化数据。

- Realm 对比于 Sqlite 具有更高的性能，接口上复合 Nosql 到特征，具有完全面向对象的接口。

- Realm 对于异步的影响：因为 Realm 使用了懒加载的技术，所以说其内部已经实现即用即取的功能，所以在使用 Realm 方案下其实是不需要异步加载数据的。

- 对于 Realm 对上层的高入侵性处理：直接将业务 Model 继承 Realm 的对象是高耦合的，不仅 Model 必须继承 RealmObject，而且 Model 层中的集合都需要换成 RealmArray 对象；除此之外，在取数据的时候，Realm 希望我们使用 RealmResults 这个 List 实现类来作为查询结果，这样对于 View 层来说也是严重的耦合。

- 解决方法：对于 Model，我们使用数据适配器(Adapter)模式，将 Model 转换为 Realm 缓存数据结构再进行缓存，保持业务 Model 的独立性。对于查询返回的 RealmArray 数据结构我们使用 List 接口代理，这样无论 MVP 哪一层，接口都是完全独立的。

- Realm 的优势：对于异常高度封装，结合 RxJava 可以方便的对其进行错误处理；具备极高的性能，查询性能约是 GreenDao 的 100 倍，再加上使用了懒加载技术，可以省略异步和分页的逻辑处理；除此之外，他保证了同一线程内全局的数据一致性，这样的话在主线程发生操作时时，在主线程查询出的数据也会同步得到更新，免去了同步更新数据的逻辑处理。

6.View 和 View 事件绑定组件

- 使用 ButterKnife 进行 View 的绑定以及 View 事件的绑定，以减少不必要的重复代码，这样使代码简洁明了，提高了代码的可读性。

7.图片异步加载与缓存组件

- 使用 Picasso 进行网络图片的异步加载显示和多级缓存。

8.Http 缓存组件

- 如果 URL 固定，业务简单：缓存使用 Retrofit2 自带即可。本地缓存则可以使用 OkHttp 自带的缓存。

- 如果 URL 不固定，缓存的策略不固定：本地缓存库建议 AsimpleCache 或者 DiskLruCache，当的缓存场景比较少时，选取这类轻量级的框架。

- 与 RxJava 结合：把缓存操作包装为 Observable ；首先去缓存中读取数据，如果没有缓存则使用再网络请求的 Observable ；并且使用 defer 操作来包装上述数据源选择的逻辑为新的 Observable。这样就保证了 Model 层，Presenter 层的逻辑一致，这样就不会因为加入缓存逻辑而对整个层次造成影响。

9.异常处理组件

- 异常的路径：异常通常产生于 API 和 Model 层，现在需要反应在 View 层呈现给用户。当然除此之外，也需要在其他层统一截获某些异常做一些收集上报之类的操作。整体上异常是在 Model 层的 RxJava 中 onError 捕获的，最终传递到 View 层变成用户可读异常呈现给用户。

- 异常的转换与加工：我们需要一个统一的异常处理引擎，对各种异常进行处理，并转换成用户可以阅读的异常信息。利用 RxJava 的 onErrorResumeNext 回调插入对异常的转换合加工逻辑。这样的话在 onError 的回调中的到的就已经是用户可读的异常了。

- 异常的统一记录：为了方便对异常进行统一的记录，可以利用 RxJava 的 doOnError，插入对异常的记录上报服务器逻辑。

## 5.8 组件管理器 ComponentManager

有些全局 Component 需要暴露在全局的作用域之下，ComponentManager 提供了注册 Component 和获取 Component 的方法。以让底层 Component 注册进入，提供给上层组件使用。

当某个组件被注册时或者注销时，ComponentManager 会向事件总线 EventHub 发送相应的事件通知，上层 App 可以订阅这个通知做相应的处理。

```
public class ComponentManager {

    private static Map<Class, Wrapper> wrappers = new ConcurrentHashMap<>();
    /**
     * lifecycle: static
     */
    private static Map<Class,Object> components = new ConcurrentHashMap<>();

    /**
     * 不影响 Dagger2 生命周期管理
     */
    private static Map<Class,WeakReference> weakComponents = new ConcurrentHashMap<>();
    static {
        wrappers.put(UserManagerComponent.class, new Wrapper(UserManagerComponent.class,
UserManager::init));
    }

    public static <T> void registerStaticComponent(Class type, T instance) {
    components.put(type, instance);
    AppComponent appComponent = getStaticComponent(AppComponent.class);
    if (appComponent != null) {
        appComponent.eventHub()
                .post(new ComponentEvent(ComponentEvent.Type.Register, type));
    }
}

public static <T> T getStaticComponent(Class<T> type) {
    synchronized (type) {
        T t = (T) components.get(type);
        if (t == null && wrappers.containsKey(type)) {
            t = (T) wrappers.get(type).initer.init();
            components.put(type, t);
        }
        return t;
    }
}

    public static void removeStaticComponent(Class type) {
        components.remove(type);
    }
    public static <T> void registerWeakComponent(Class type, T instance) {
        components.put(type, new WeakReference(instance));
    }
    public static <T> T getWeakComponent(Class<T> type) {
        T t = (T) components.get(type);
        if (t == null)
            removeWeakComponent(type);
        return t;
    }
    public static void removeWeakComponent(Class type) {
        components.remove(type);
    }
}
```

代码段 5-4 组件管理器

# 第六章 DI 依赖注入

## 6.1 依赖注入定义

依赖注入(DI)是控制反转(IOC)的一种，最早在 2004 年，Martin Fowler 就提出了“哪些方面的控制被反转了？” 的观点。

依赖注入简单来说就是“高层类应该依赖底层基础设施来提供必要的服务”。高层类使用底层基础的接口，而这些接口实现可能由多种，而依赖注入正是管理多种实现并且依据配置选定实现并且注入到接口上的工具。

## 6.2 依赖注入意义

如今我们说面向接口编程，利用接口对系统进行解耦，接口描述了“做什么”，而实现类则是“怎么做”，但是当别的类去使用这个类时，它仅仅需要知道它能“做什么”，接口就是定义（规范，约束）与实现的分离，即名实分离的原则。

当我们按照面向接口编程的原则开发程序之后，我们仍然需要为接口赋值，而这个操作在依赖注入之前都是 new 某个实现类，也有方法是使用工厂模式获取想要的实现类。但是无论如何，一个接口都有可能有多个实现类，所以 new 就带来了耦合。

对象的依赖是一张图，我们可以使用图来描述对象之间的依赖关系，这样我们就可以把对象的 new 从逻辑代码中抽取出来放到一个统一的图里面。这张图可以是 Spring 早期使用的 XML，也可以是 Java 代码上的注解。

首先我们应该知道，人们在很长的一段时间里都是利用控制反转原则规定：应用程序的流程取决于在程序运行时对象图的建立。通过抽象类和接口的实现对象交互可以实现这样的动态流程。而使用依赖注入技术或者服务定位器也可以完成运行时对象绑定。

使用依赖注入可以带来以下好处:

依赖的注入和配置独立于组件之外。

因为对象是在一个独立、不耦合的地方初始化，所以当注入抽象方法的时候，我们只需要修改对象的实现方法，而不用大改代码库。

依赖可以注入到一个组件中：我们可以注入这些依赖的模拟实现，这样使得测试更加简单。

可以看到，能够管理创建实例的范围是一件非常棒的事情。按我的观点，你 app 中的所有对象或者协作者都不应该知道有关实例创建和生命周期的任何事情，这些都应该由我们的依赖注入框架管理的。

可以简单想象一下以下场景:

在单元测试中，你需要把 MVP 中的 Model 更换成测试数据源，这时候你就需要更换 Model 的实现类，而直接在业务代码中更换是不健壮的。

在公共组件中，你需要更换 Cache 的实现，由 LRU 改为 FIFO。

## 6.3 Android 中的依赖注入 Dagger2

### 6.3.1 简介

Dagger2 是 Dagger 的升级版，是由 Google 接手维护的 DI 依赖注入框架，从某个方面来说，Dagger2 也是 Android 官方的框架了。从这点来说还是比较建议大家研究使用的。

对比 Dagger1:

1.没有使用反射：图的验证、配置和预先设置都在编译期间执行。

2.更加容易调试和可跟踪：提供完全具体地调用提供和创建的堆栈。

3.更好的性能：谷歌声称他们提高了 13% 的性能。

4.代码混淆：使用派遣方法，就如同自己写的代码一样。

### 6.3.2 Dagger2 中的重要概念

1.Inject 注解: 我们通常在需要依赖注入的地方使用这个注解。也就是说，你用它告诉 Dagger2 这个类或者字段需要一个实现对象。这样，Dagger 就会搜索依赖类型并构造一个这个类的实例来满足他们的依赖。

2.Module 注解: Modules 类里面的方法专门提供依赖图，所以我们定义这个类，用@Module 注解，这样 Dagger 在构造类的实例的时候，就知道从哪去找到所需要的依赖类型。Modules 的一个重要特征是它们被设计为可分区得并可以组合在一起（在我们的 app 中可以有多个组成在一起的 modules）。

3.Provide 注解: 在 modules 中，我们定义的方法是使用这个注解，以此来标记我们想要构造对象并提供这些依赖。

4.Component 注解: Components 就是一个注入器，也可以说是连接@Inject 和@Module 的桥梁，它的主要作用就是关联这两个部分。 Components 可以提供所有定义了的类型的实例方法，我们必须用 @Component 注解一个接口然后写出所有的@Modules 组成该组件，如果缺失了任何一块都会在编译的时候抛出异常。所有的组件都可以通过它的 modules 知道它所依赖的范围。编译时，APT 会帮我们实现

Component 接口，其实所有依赖就被保存在 Component 的实现类中，所以我们必须在作用域的发起者中持有 Component 对象。换句话说，Component 除了是个注入器之外，还是保存组件组件内部所有依赖的容器。

5.Scope 注解: Scopes 是非常的有用，Dagger 可以通过自定义注解限定其作用域。这是一个非常强大的特点，因为就如前面说的一样，没有必要让每一个对象都去了解如何管理他们的实例。我们用自定义的 @PerActivity 注解一个类，所以这个对象的生命周期就和 activity 一样。简单来说就是我们可以自己定义所有范围的粒度(@PerFragment, @PerUser, 等等)。其实 Scope 只会在 Module 和 Component 之间起作用，某种程度上 Scope 只是一个标记。

6.Qualifier 注解: 当类的类型不足以鉴别一个依赖的时候，我们就可以使用这个注解为 Component 起一个别名。在 Android 中，我们会需要不同类型的 context，所以我们就可以定义 qualifier 注解“@ForApplication”和“@ForActivity”，这样当注入一个 context 的时候，我们就可以告诉 Dagger 我们想要哪种类型的 context。

注入方法:

1.构造方法注入：在每个类的构造方法上面注解 @Inject。

2.成员变量注入：在类的成员变量（非私有）前面注解 @Inject。

3.函数方法注入：在函数上面注释@Inject。

## 6.4 本架构中的依赖注入

### 6.4.1 MVP 三层依赖组装

在 MVP 中，如代码段 5-1 所示，MVP 三层通过协议接口互相引用，这样的话我们使用 Dagger2 完成 MVP 三层依赖的拼装。

拼装过程如 代码段 4-5 所示，Model，View，Presenter 的实例都保存在了 Module 中，在 ProvideXXXX 中，就是真正 new 的过程，Module 相当于 Spring 中的 XML。

而对于 Component，其依赖所需的 API Component，而 API Component 依赖于 App Component，Component 的作用域来自 Activity/Fragment 所以其 Scope 为 ActivityScope/FragmentScope。

```
@Component(modules = LoginModule.class, dependencies = FeiZhaiAPIComponent.class)
@ActivityScope
public interface LoginComponent {

    void inject(LoginProtocol.Presenter presenter);
    void inject(LoginProtocol.View view);
    void inject(LoginProtocol.Model view);

    LoginProtocol.Presenter presenter();
}
```

代码段 6-1 Component 例子

如上面代码段所示，Component 提供了 MVP 三层的注入接口，便于 Component 所依赖的其他依赖的注入。

从上面知道，Component 的实现类是保存所有依赖的容器，所以在 MVP Component 中我们需要让 Activity/Fragment 这个活动发起方持有 Component 的强引用，这样 Component 的生命周期等于是委托给 Activity/Fragment 管理了。

所以在 BaseActivity 应该插入 Component 初始化的抽象方法，交由子类实现:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ....
    component = setupActivityComponent();
    basePresenter = setPresenter();
    initView();
    if (basePresenter != null) {
        basePresenter.onViewInited();
    }
    ....
}
protected abstract T setupActivityComponent();

protected abstract BasePresenter setPresenter();
```

代码段 6-2 BaseActivity

### 6.4.2 App 全局组件依赖

前面也说过，App 需要用刀许多全局的公共组件，这些组件都使用接口抽象了一层进行解藕。

而依赖的注入拼装依然使用 Dagger2，来看一下依赖图 Module，框架层使用默认实现，而应用层可以在 App 初始化时指定自定义的依赖配置。

```
/**
 * App 基础组建依赖组装
 * Created by ganyao on 2017/3/15.
 */

public interface IAppModule {

    IKVDiskCache provideKVDiskCache();

    IAsyncSubjectsQueue provideSubscriber();

    IAPIGenerator provideAPIGenerator();

    IBaseHttpModel provideBaseHttpModel();

    IImageLoader provideImageLoader();

    IEventHub provideEventHub();
}
```

代码段 6-3 App 全局依赖拼装

### 6.4.3 业务组件依赖拼装

在 App 整个生命周期中，有许多业务组件被多个例如 MVP Component 的高层组件依赖，例如 LoginComponent 和 UserInfoComponent 都依赖 UserApiComponent，登陆组件，和用户信息组件都依赖用户 HTTP API 组件。

下面这个类时 HTTP API 业务组件的基类，提供 API 接口实现的代理，Session 管理等。

当此 API Component 生命周期开始时，对应的 Session 也将开始，直到此会话结束，API Component 的生命周期也将结束。

```
public abstract class BaseAPIModule {

    public IAPIGenerator apiGenerator;

    protected String BASE_URL;
    private Map<String,String> headers, pars;
    private APIServiceConfigs configs;

    protected Session session;
    protected void init(String url) {
        BASE_URL = url;
        apiGenerator = BaseApplication.getAppComponent().apiGenerator();
        apiGenerator.setUrl(url);
    }
    public Session generateSession() {
        session = new Session();
        setHeaders(session.getHeaders());
        setPars(session.getPars());
        return session;
    }
    public void setPars(Map<String, String> pars) {
        this.pars = pars;
        apiGenerator.setPars(pars);
    }
    public void setHeaders(Map<String, String> headers) {
        this.headers = headers;
        apiGenerator.setHeaders(headers);
    }
    public void setConfigs(APIServiceConfigs configs) {
        this.configs = configs;
        apiGenerator.setConfigs(configs);
    }
    protected <T> T getAPIService(Class<T> tClass) {
        return apiGenerator.getAPI(tClass);
    }
    protected Binder getBinder() {
        return new Binder();
    }
    public class Binder {
        public BaseAPIModule getModule() {
            return BaseAPIModule.this;
        }
    }
}
```

代码段 6-4 API Module

1. 业务组件公共返回值处理：一般 JSON 格式数据都是类似{status:0,msg:”success”,data:{“xxxxx”}}这种格式，真正的业务 Model 会被一层公共的数据包裹一层，也就是说这里可以视为一个切面，这个切面主要就是两个作用，其一解包出真正的 Model，其二检查状态码，有错则抛出业务异常，并带上里面的异常信息，抛到上层进行处理。

```
public interface IResponseAdapter<I,O> {
    O adapter(I i);
}
```

代码段 6-5 公共返回值适配器

上面的接口抽象了公共返回值到具体业务 Model 的过程。

```
@Override
public T adapter(BaseResponse<T> baseResponse) {
    int status = baseResponse.getStatus();
    String msg = baseResponse.getMessage();
    T data = baseResponse.getData();
    if (status != 1) {
        throw new HttpServiceException(msg);
    }
    return data;
}
```

代码段 6-6 默认 adapter

Http 异步请求封装:

```
/**
 * @O = 观察者实体类型
 * @A = API 接口返回类型
 * Created by ganyao on 2017/3/9.
 */
public interface IBaseHttpModel<O,A> {
    void setBaseResponse(IResponseAdapter baseResponse);
    <M> IAsyncSubject<O> getAsyncSubject(A api, AsyncCallback<M> callback);
    <M> IAsyncSubject<O> getAsyncSubjectWithCache(A api, IGetFromCache<M> getFromCache,
ISaveToCache<M> saveToCache, AsyncCallback<M> callback);
    <M> IAsyncSubject<O> mergeSubjects(AsyncCallback<M> callback, A... subjects);
    <M> IAsyncSubject<O> concatSubjects(AsyncCallback<M> callback, A... subjects);
    interface IGetFromCache<T> {
        T fromCache();
    }
    interface ISaveToCache<T> {
        void toCache(T t);
    }
}
```

代码段 6-7 异步任务封装

1.getAsyncSubject 获得异步普通任务，需要传入 Callback 和 API。

2.getAsyncSubjectWithCache 获得带 Cache 的异步任务，需要传入读写 Cache 的实现。

3.mergeSubjects 合并异步任务，将多个异步请求合并成一个并行执行，很多情况下，我们需要并行请求多个业务接口再触发 Callback 显示结果。这样的话，需要等到所有请求都完成的时候再处理结果。

4.concatSubjects 链接异步任务，将多个异步任务串联，解决了一些情况下需要先做什么再做什么的问题，即异步任务之间执行顺序条件的问题。

# 第七章 AOP 面向切面编程

## 7.1 AOP 的定义

面向切面编程（AOP，Aspect-oriented programming）需要把程序逻辑分解成关注点（concerns，功能的内聚区域）。也就是说，在 AOP 中，我们不需要显式的修改切面发生处的代码就可以向原有的逻辑中插入新的的逻辑块。这种编程范式定义了横切关注点（cross-cutting concerns，多处代码中需要的类似逻辑，但没有一个单独的类来实现）应该是单例的，且能够多次注入到很多需要该逻辑的地方，这些地方组成我们所说的切面。

代码注入是 AOP 中的重要组成部分：它在处理上述提及的横切整个程序的关注点时非常重要，例如日志打印和性能监控。这种场景，并不如你所想的应用甚少，相反的，每个程序员都可以有使用这种注入代码能力的机会，这样可以避免很多痛苦的重复性劳动。

AOP 是已经存在了多年的编程范式。把它应用到 Android 开发中也很非常用。经过调研之后，我认为把 AOP 用在我们的架构中可以获得很多好处。

## 7.2 在 Android 上使用 Aspectj 实现 AOP

在开始之前，我们先看一下需要了解的概念：

1.Cross-cutting concerns（横切关注点）：尽管面向对象模型中大多数类会实现单一特定的功能，但通常也会开放一些通用的附属功能给其他类。例如，我们希望监控某些方法的执行性能，在 Android 中，主线程的执行效率是非常敏感的。尽管每个类都有一个区别于其他类的主要功能，但在代码里，仍然经常需要添加一些相同的流程。

2.Advice（通知）：注入到类文件中的代码。典型的 Advice 类型有 before、after 和 around，分别表示在目标方法执行之前、执行后和完全替代目标方法执行的代码。 除了在方法中注入代码，也可能会对代码做其他修改，比如在一个类中增加字段或者接口。

3.Joint point（连接点）：程序中可能作为代码注入目标的特定的点，例如一个方法调用或者方法入口，调用 jointpoint.proccess 执行切面方法。

4.Pointcut（切入点）：告诉代码注入工具，在何处注入一段特定代码的表达式。例如，在哪些 joint points 标记了一个特定的 Advice。切入点可以选择唯一一个，比如执行某一个方法，也可以有多个选择，比如，在方法描述上标记了一个 @DebugLog 的自定义注解的所有方法。

5.Aspect（切面）：我们可以把 Pointcut 和 Advice 的组合看做切面。例如，我们

在应用中通过定义一个 pointcut 并且给他指定合适的 advice，例如添加一个权限检查切面。

6.Weaving（织入）：简单来说就是描述了注入代码（advices）到目标位置（joint points）的过程。

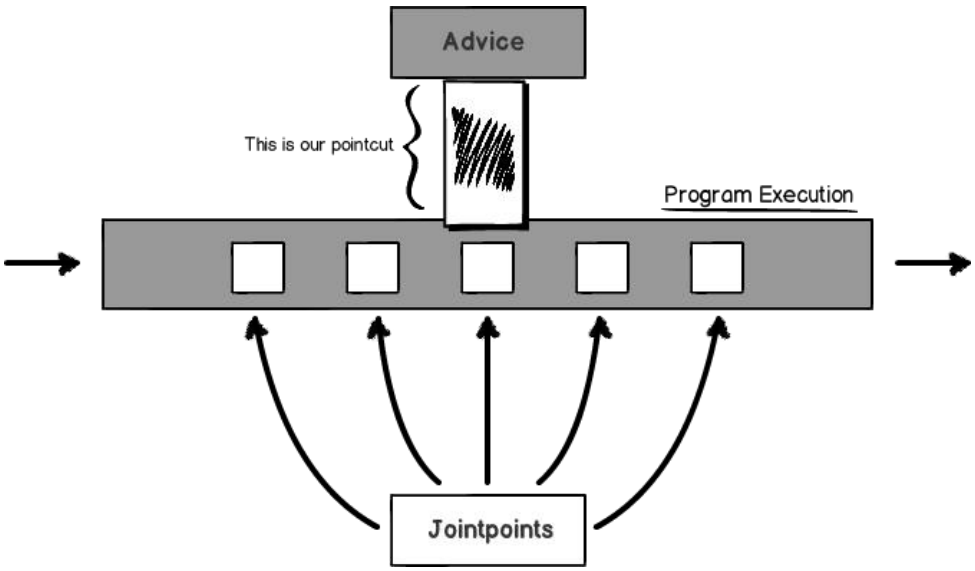



图 7-1 AspectJ 织入过程

## 7.3 动态织入实现原理

一般来说实现切面织入需要类似方法 Hook 的能力，在 Java 中实现类似功能的在 Runtime 一般有两种方式:

动态代理，动态代理是 JDK 官方支持的方法 Hook 方式，动态代理实际是 JDK 帮你生成了目标接口的实现类，而这个实现方法会调用 Proxy 方法。所以动态代理的局限在于，只能代理从接口派生的方法。

cglib，cglib 是第三方动态代理库，其原理完全不同于第一种，CGLIB 包的底层是通过使用一个小而快的字节码处理框架 ASM(Java 字节码操控框架)，来转换字节码并生成新的类，这种方式不需要方法是继承自接口的。

在 JavaEE 中 Spring AOP 可以选择织入的实现方式，cglib 也在可选之列，但是在 Android 中，由于平台特殊性(Android 字节码是 Dex 非 .class)，Aspectj 选择编译期间的织入方式。

切入点方法将在编译期间被 APT “替换”成代理方法，当然事实上原方法并没有改变，只是 APT 会新增加一个方法去装饰原方法，对原方法的调用被重定向到新方法上，新方法再调用原方法。

这样的话保障了在 Android 客户端上运行的稳定性。除此之外，犹豫并没有大量使用反射，所以也保障了性能，当然也损失了一些动态性。

## 7.4 AOP 在本架构中的使用

### 7.4.1 动态权限检测

众所周知，Android 自从 6.0 之后引入了动态权限检测，开发者在使用众多系统资源的之前，都需要询问用户是否授权。

最为主要的是这个动作是大量重复的，除了重复代码，多重的分支判断也会污染真正的逻辑代码。

那么很明显的我们可以认为权限检测是一个切面，我们可以用 AspectJ 实现的装饰器装饰需要用到权限的方法，当调用到该方法时，进行统一的权限检测，如果用户同意授权则继续调用该函数，如果拒绝则放弃执行，Toast 告知用户拒绝了该权限。其实可以类似 python 中的装饰品语法糖。

```
@Around("pointcutOnPermissionCheckMethod(permissionCheck)")
public Object permissionCheckAndExecute(ProceedingJoinPoint joinPoint, PermissionCheck permissionCheck) throws
Throwable {
    RxPermissions.getInstance(BaseApplication.getApplication())
            .request(permissionCheck.value())
            .subscribe(granted -> {
                if (granted) {
                    try {
                        joinPoint.proceed();
                    } catch (Throwable throwable) {
                        LOG.e("permission check invoke error!");
                    }
                } else {
                    BaseApplication.getAppComponent()
                            .eventHub()
                            .post(new PermissionDenyEvent(permissionCheck.value(), joinPoint.getSignature()));
                }
            });
    return null;
}
```

代码段 7-1 权限检测切面

```
@PermissionCheck(Manifest.permission.WRITE_EXTERNAL_STORAGE)
protected void initData() {
.....
```

代码段 7-2 权限检测的使用

至于如何使用：只需要在需要申请权限的函数上面加上相应的装饰器 Annotation 即可，APT 自然会编译时代理该函数。

### 7.4.2 登陆检测

和动态权限检测类似，检测用户是否登陆也是 App 中常常出现的切面之一，在 App 中当用户点击按钮想进入某个页面时，经常需要判断用户是否已经登陆，如果没有登陆就跳转到未登陆的 Callback，一般直接跳转到登陆界面让用户登陆，可以通过 Event 总线框架向 App 上层发送未登陆的 Event。

```
public Object loginAndExecute(ProceedingJoinPoint joinPoint) throws Throwable {
    UserManager userManager =
ComponentManager.getStaticComponent(UserManagerComponent.class).userManager();
    BaseApplication.getAppComponent()
            .eventHub()
            .post(new LoginChecked(userManager.isLogin()
                    ? LoginChecked.LoginCheckedResult.Login
                    : LoginChecked.LoginCheckedResult.UnLogin));
    if (userManager.isLogin()) {
        return joinPoint.proceed();
    } else {
        Object[] pars = joinPoint.getArgs();
        if (pars != null && pars.length > 0) {
            for (Object par:pars) {
                if (par != null && par instanceof UnLoginCallback) {
                    UnLoginCallback callback = (UnLoginCallback) par;
                    callback.onUnLogined();
                    break;
                }
            }
        }
        SwiftLog.LOGV("LoginManager", "please login first");
        return null;
    }
}
```

代码段 7-3 登陆检测切面

其使用方式可以参考动态权限检测的使用方式。

### 7.4.3 性能检测 & Log

在 Android 中我们需要监控某个函数的执行消耗时间是非常常见的场景，尤其是在主线程的高耗时作业，主线程对性能要求是高度名感的，一旦主线程进入到高耗时的函数，给用户的直观感受就是卡顿。

但是，很多情况下高耗时操作又是难以避免的，例如绘制一个复杂的 View，还有一些看似简单的 IO 操作也可能在资源竞争激烈的时候发生阻塞；除此之外调用某些系统接口，尤其是 IPC(进程间通讯) 操作例如请求某个 Service RPC 调用时也会有意料之外的耗时。

同上面一样，我们希望在敏感的方法上面加上注解，这样该方法就会被引入性能监控的切面。

```
@Around("method() || constructor()")
public Object logAndExecute(ProceedingJoinPoint joinPoint) throws Throwable {
    enterMethod(joinPoint);

    long startNanos = System.nanoTime();
    Object result = joinPoint.proceed();
    long stopNanos = System.nanoTime();
    long lengthMillis = TimeUnit.NANOSECONDS.toMillis(stopNanos - startNanos);

    exitMethod(joinPoint, result, lengthMillis);

    return result;
}
```

代码段 7-4 性能监控切面

### 7.4.4 异步与主线程

我们可以利用 AspectJ 实现类似 C# 中 async 和 wait 关键字效果的语法糖，以简化 Android 中复杂的异步操作。

首先我们利用 Android 自身的官方注解 WorkThread 和 UiThread，这两个注解本身只是一个标记(TAG)，以便于 IDE link 对代码进行静态审查。我们可以利用这两个注解做一个异步与同步的切面。

WorkThread 在切面中，我们将原方法包裹在子线程体中调用，这样一来，返回值就不能正常返回，我们可以在原方法的参数中传入 Callback 在切面中遍历方法参数找到 Callback 则调用。

UiThread 则较为简单，判断方法是否是在主线程中调用，如果是则直接调用，不是则 Post 到 MainLooper 中。

```
@Around("method() || constructor()")
public Object asyncAndExecute(ProceedingJoinPoint joinPoint) throws Throwable {
    new Thread( () -> {
        Object[] pars = joinPoint.getArgs();
        AsyncSuccessCallback successCallback = null;
        AsyncFailureCallback failureCallback = null;
        if (pars != null) {
            for (Object par : pars) {
                if (par instanceof AsyncSuccessCallback) {
                    successCallback = (AsyncSuccessCallback) par;
                } else if (par instanceof AsyncFailureCallback) {
                    failureCallback = (AsyncFailureCallback) par;
                }
            }
        }
        try {
            Object res = joinPoint.proceed();
            if (successCallback != null) {
                successCallback.success(res);
            }
        } catch (Throwable throwable) {
            if (failureCallback != null) {
                failureCallback.failure(throwable);
            }
        }
    }).start();
    return null;
}
@Around("method() || constructor()")
public Object uiThreadAndExecute(ProceedingJoinPoint joinPoint) throws Throwable {
    if (Looper.myLooper() == Looper.getMainLooper()) {
        return joinPoint.proceed();
    } else {
        new Handler(Looper.getMainLooper()).post(() -> {
            try {
                joinPoint.proceed();
            } catch (Throwable throwable) {
            }
        });
        return null;
    }
}
```

代码段 7-5 异步和 UiThread 切面

# 第八章 Bundle 模块化容器化

## 8.1 定义

模块在业务上是重量级的业务划分；在工程结构中就是一个单独的子工程；在 APK 中就是一个单独的 DEX 加资源包；在开发中可能就是一个单独的子项目。

Bundle 与 组件 Component 的区别在于，Component 是轻量级的功能部分，而 Bundle 是重量级的，一个 Bundle 可能包含很多个 Component。

## 8.2 意义

产品开发迭代期间:模块化解决了多团队协作的问题，在工程期，实现工程独立开发，调试的功能，工程模块可以独立，各个团队可以独立工作。这样避免了繁琐并且容易出问题的代码 merge，冲突解决等问题，大大增加了开发迭代的效率。

在产品运行部署期间:实现了按需加载，用户用不着的功能模块可以先不下载，节省了流量和存储。当用户需要用到某个功能模块时再从网络上下载对应模块动态加载。

产品运维期间:提供了快速增量的更新修复能力，快速升级，修复 Bug。

## 8.3 Atlas & Small

### 8.3.1 简介

Atlas 和 Small 是现在 Android 比较成熟的 Bundle 框架。

Bundle: 类似 OSGI 规范里面 bundle（组件）的概念，每个 bundle 都有自己的 classloader，并且与其他 bundle 相隔离，同时 Atlas 框架下 bundle 都有自身的资源段，Atlas 帮助每个 Bundle 分配 PackageID，打包时 AAPT 指定；另外与原有 OSGI 所定义的 service 格式不同之处是 Atlas 里面 Bundle 透出所有定义在 Manifest 里面的 component，随着 service，activity 的触发执行 bundle 的安装，运行。

Atlas 是伴随着手机淘宝的不断发展而衍生出来的一个运行于 Android 系统上的一个模块化容器框架，我们也叫动态组件化(Dynamic Bundle)框架。它主要提供了解耦、组件、动态性的支持。在工程师的工程编码期、Apk 运行期以及后续运维期间都发挥了重要作用。

Atlas 是工程期和运行期共同起作用的框架，我们尽量将一些工作放到工程期，这样保证运行期更简单，更稳定。

相比 multidex，atlas 在解决了方法数限制的同时以 OSGI 为参考，明确了业务开发

的边界，使得业务在满足并行迭代，快速开发的同时，能够进行灵活发布，动态更新以及提供了线上故障快速修复的能力。

与外界某些插件框架不同的是，atlas 是一个组件框架，atlas 不是一个多进程的框架，他主要完成的就是在运行环境中按需地去完成各个 bundle 的安装，加载类和资源。

### 8.3.2 原理

无论是 Atlas 还是 Small 其原理都大同小异。

有以下几个部件组成：

1.构建期间，Atlas 实现了自己的构建脚本程序以及资源打包 AAPT，以及 diff 具。

2.Dex 动态加载，资源动态加载，依然利用的是 dexElements。

3.组件注册，Atlas 不同于插件化框架，Atlas 的不存在动态的组件注册，子模块的组件会在编译期间被 merge 进主模块的 menifest。所以当主模块被安装时子模块中的组件已经被提前注册了。

4.Bundle 生命周期，Bundle 可以动态的加载和卸载，那么就必然存在生命周期的管理。

运行期:

bundle 的运行期加载策略:

由之前的容器的运行机制可以得知，bundle 动态部署后对容器来说没有任何变化，还是按需的 load bundle，只是在需要去 load 前版本选择上会使用最高的可用版本

主 apk 动态部署后的加载策略:

了解主 Apk 的加载策略之前，先分别介绍下主 Apk 动态部署得以成功的技术基础再阐述生效的方式：

class|so:

普通 apk（也就是主 Apk）class，so 库加载入口--PathClassLoader 前面容器的技术原理里面介绍了 PathClassLoader 的父子关系，PathClassLoader 自身负责主 Apk 的类和 c 库的查找路口；其 parent BootClassloader 负责 framework sdk 的内容的查找。

PathClassLoader 本身也是一个 class，继承了 BaseDexClassLoader（同 DexClassLoader），里面查找过程在 DexPathList 里面实现（如下图）:

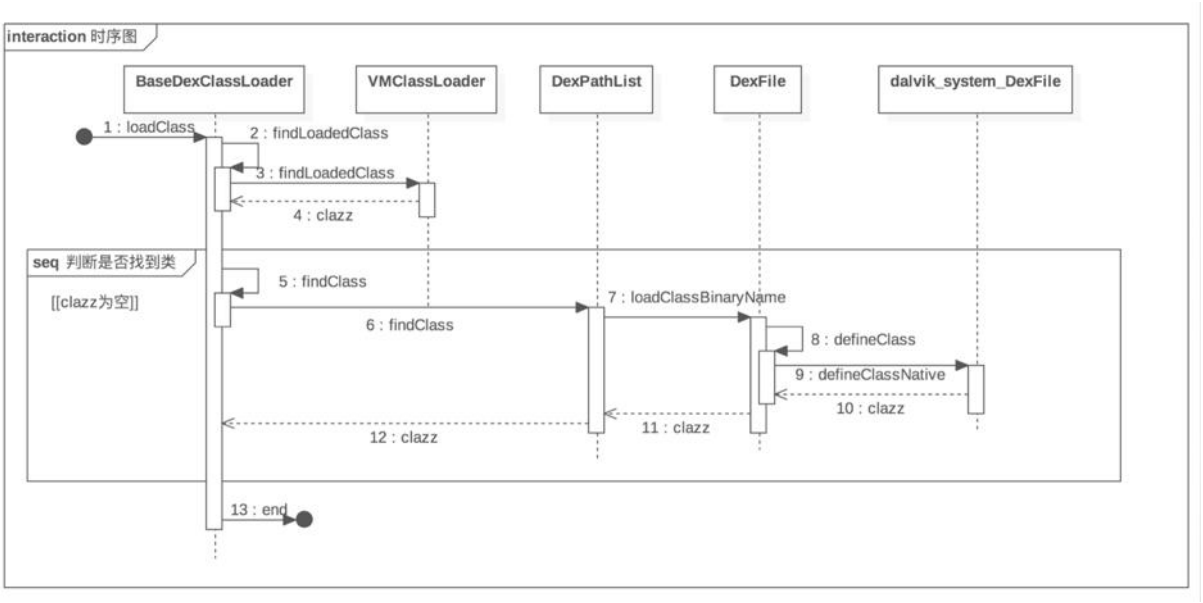



图 8-1 Dexloader

DexPathList 最终通过 DexFile 去 loadClass，DexPathList 可以理解为持有者 DexFile 以及 nativeLibrary 目录，再查找的时候遍历这些对象，直到找到需要的类或者 c 库，那么动态部署的方式就是把新修改的内容添加到这些对象的最前面，从而使得查找的过程中新修改的内容能够提前找到从而替换原有的（如下图）

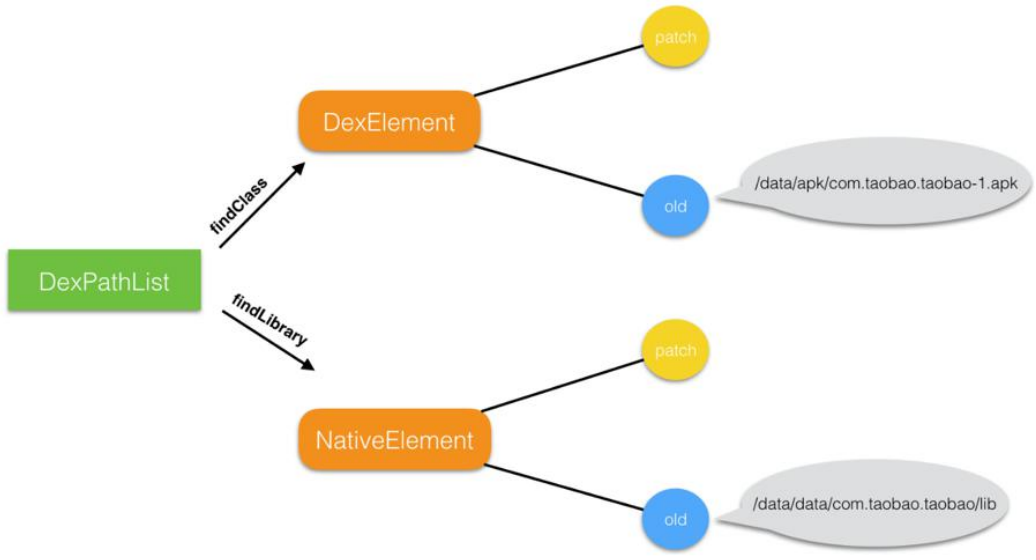



图 8-2 dexElements

## 8.4 本架构中的模块化

如图 2-1 所示，本架构模块大致分为 1+N,即一个主模块 + N 个子功能模块，主模块(主容器)又包括了如下几个小模块:

1.第三方 lib 包装 library 模块，对第三方库进行统一管理。

2.EventPoster 事件处理模块，框架层在特定时候会向 App 应用层发送状态消息，如用户拒绝了某次动态授权时。这种场景往往发生在 App 层不方便向框架层注册 Callback 的时候。

3.Common 架构主体，包含了各种 Base，约束了上层 App 的实现结构。

4.Bundle 底层驱动，由 Atlas 实现，负责了模块的加载，生命周期管理等等。

5.各种公共模块，User，Utils，API，控件等等。

6.在 APK 结构上，主模块将在第一次安装时被加载到用户设备上。

其他的子功能模块都可以引用主模块中的各种模块和组件，但是在打包子模块时，这些组件将不会重复打包进子模块，在运行期间，子模块被加载然后共享主模块中的底层组件。

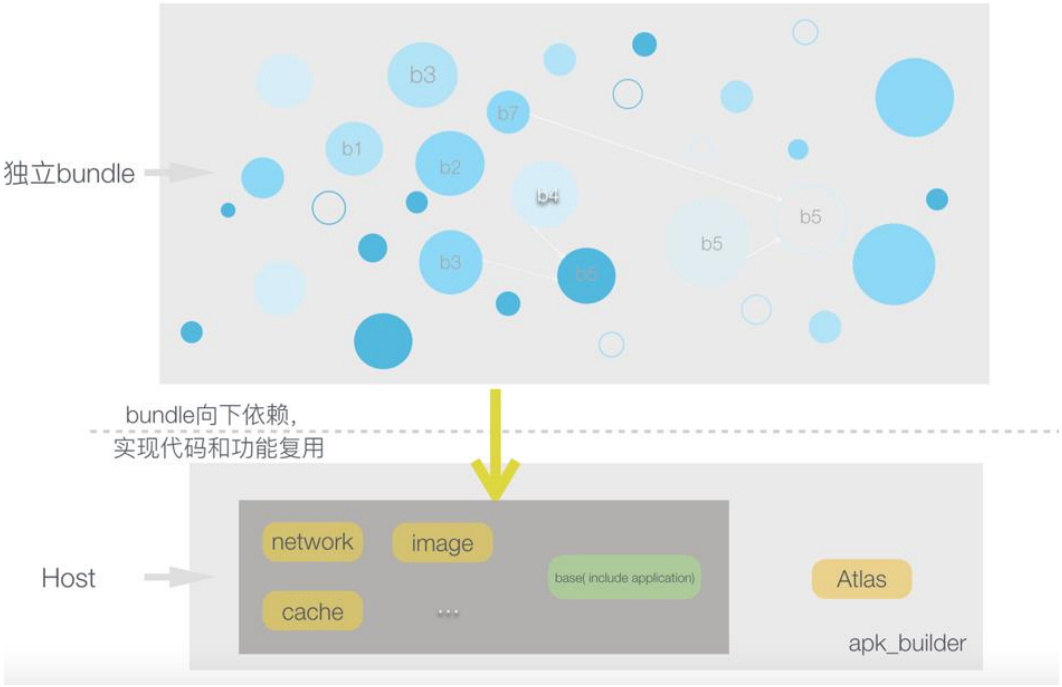



图 8-3 Bundle 结构

# 第九章 Router 页面路由

## 9.1 背景

现在的 App 可能包含众多的页面，页面的导航是一张图，页面的层级是一棵树。页面打开往往是有顺序的，一个界面会依赖其前面的界面。也就是说你在打开某个界面之前需要先打开某些界面。

现在的 App 出于各种需要，业务上页面路径被设计成 RESTFul 风格的路径，不仅可以和网页端保持一致，也有利于业务设计和分析。

我们经常发现可以在很多其他 App 内部或者 Web 页面中点击超链接来跳转到我们自己 App 的某个页面，这样的场景越来越多。

当我们的 App 架构模块化时，跨模块间的页面跳转因为不知道类的具体引用，从而让传统的使用 Intent 跳转变得十分复杂。

页面跳转时所附加的参数越来越多。

## 9.2 Intent 导航遇到的问题

1.集中式的 URL 管理：谈到集中式的管理，总是比较蛋疼，多人协同开发的时候，大家都去 AndroidManifest.xml 中定义各种 IntentFilter，使用隐式 Intent,最终发现 AndroidManifest.xml 中充斥着各种 Schame，各种 Path，需要经常解决 Path 重叠覆盖、过多的 Activity 被导出，引发安全风险等问题

2.可配置性较差：Manifest 限制于 xml 格式，书写麻烦，配置复杂，可以自定义的东西也较少

3.跳转过程中无法插手：直接通过 Intent 的方式跳转，跳转过程开发者无法干预，一些面向切面的事情难以实施，比方说登录、埋点这种非常通用的逻辑，在每个子页面中判断又很不合理，毕竟 activity 已经实例化了

4.跨模块无法显式依赖：在 App 小有规模的时候，我们会对 App 做水平拆分，按照业务拆分成多个子模块，之间完全解耦，通过打包流程控制 App 功能，这样方便应对大团队多人协作，互相逻辑不干扰，这时候只能依赖隐式 Intent 跳转，书写麻烦，成功与否难以控制。

## 9.3 意义

为了解决上述的问题，页面路由诞生了，首先页面之间不再通过类的 Class 对象硬

链接，而是通过业务名称软链接。而软路径往往是 RESTful 风格的路径，包含了一定的语意，并且和其他平台诸如 IOS，Web 统一。

1.解决了模块之间弱引用的问题，是跨模块页面间的跳转变的简单高效。

2.解决了页面跳转之间规则的问题，诸如依赖解决，页面的先后顺序。

3.解决了页面间大数据传递的问题。

4.解决了第三方外部请求页面显示的问题。

## 9.4 页面路由的使用

页面路径声明：

```
@Route(path = "/test/activity")
public class Test1Activity extends Activity {
    @Autowired
    public String name;
    @Autowired
    int age;
    @Autowired(name = "girl") //  通过 name 来映射 URL 中的不同参数
    boolean boy;
    @Autowired
    TestObj obj;      //  支持解析自定义对象，URL 中使用 json 传递

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ARouter.getInstance().inject(this);

        // ARouter 会自动对字段进行赋值，无需主动获取
        Log.d("param", name + age + boy);
    }
}
```

代码段 9-1 页面路径定义

页面跳转：

```
// 1. 应用内简单的跳转(通过 URL 跳转在'进阶用法'中)
ARouter.getInstance().build("/test/activity").navigation();

// 2. 跳转并携带参数
ARouter.getInstance().build("/test/1")
                     .withLong("key1", 666L)
                     .withString("key3", "888")
                     .withObject("key4", new Test("Jack", "Rose"))
                     .navigation();
```

代码段 9-2 页面跳转

## 9.5 页面路由的原理及实现

### 9.5.1 页面发现

页面路由中，页面的类型，及 Activity 的类型和页面路径是一一对应的，为了加速运行时页面搜索的速度。最佳的方案就是在初始化的时候就将页面路径和页面的映射关系存放在哈希表中，这样在导航时的页面搜索将会是 O(1) 时间复杂度，并且避免了反射。

页面发现也有两种策略，和 Event 驱动一样，一种是运行时期在 Application 初始化时对类进行反射扫描，但是这种太过耗时，显然在 App 类越来越对的现在是不太可行的。

二就是使用注解处理器:编译期间，APT 注解处理器会收集所有带了 Route 注解的页面给页面导航的注解处理器部分，这时候 APT 会自动生成一个初始化类，初始化类中会将这些页面的映射关系添加到 Hashmap 中，这个 Hash 表我们称为路由表。

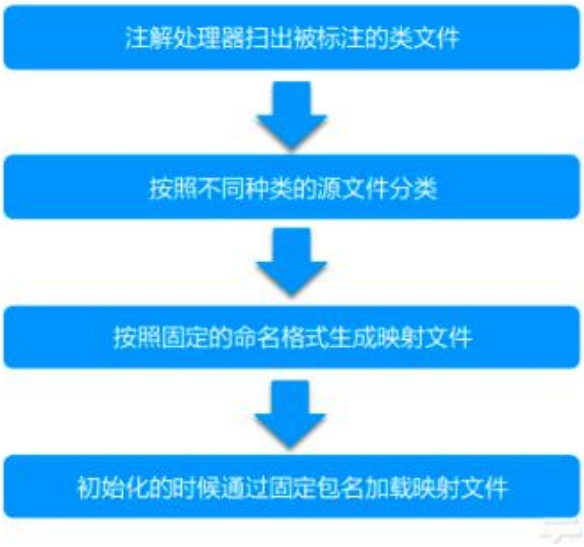



图 9-1 页面路由初始化过程

初始化时候，页面路由会调用在编译期间生成的的初始化方法，将页面路径和页面类型的映射关系保存到缓存中，然后通过 URL 跳转目标页面时候，会通过映射文件的规则，进行跳转，拦截，操作服务中方法等动作，从而达到模块间解耦。

### 9.5.2 页面分发

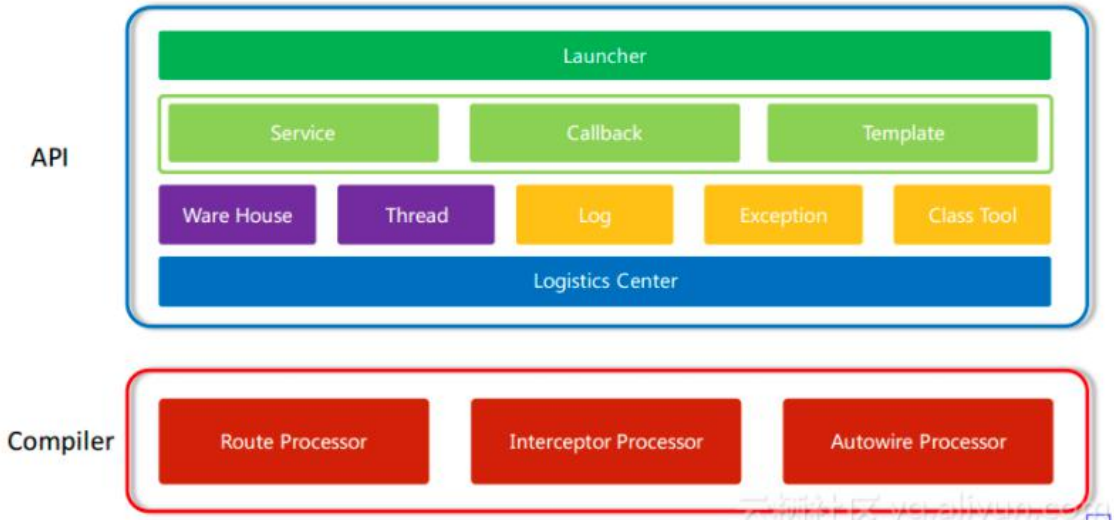



图 9-2 页面路由架构

当 Application 初始化完毕的时候页面路径和页面类型的映射关系已经保存在缓存中了，整个路由器已经准备就绪。当当前页面要求导航到另外一个页面的时候，传入的 RestFul Url 将会被送到分发器进行路径解析。

这时通过页面路径找到已经缓存好的页面类型，这时还是需要一定的反射操作的，但是很快就会被保存到缓存中。除此之外，页面路由器将导航所要求的数据和页面动画封装，交由系统的 Api Intent 进行导航，整个路由的过程也就完成了。

当然，导航过程中经常会需要先打开某个页面，再打开某个页面，也就是说路由器需要实现类似拦截器的功能，在打开某个页面之前，我们拦截这个路由请求，先打开某个依赖页面再继续导航。

# 第十章 事件驱动

## 10.1 事件驱动定义

事件驱动也被称为 EDA，是指以事件发布/订阅为模型核心的一种层级间或者组件间的通讯方式。

事件驱动主要分为订阅者，发布者，以及事件分发者(事件框架本身):订阅者何发布者分别对应传统接口通讯方式中的被调用者何调用方。

## 10.2 事件驱动意义

当我们的架构在横纵方向上有了组件化和架构分层之后，并且我们还严格遵循了面向接口原则，我们的架构确实再一定程度上解耦了。但是有个问题是不容忽视的，接口本身也是存在耦合的，确实接口让定义和实现分离了，但是当层级间或者组件间的通讯接口需要发生变化时，通讯双方的实现必然要发生改变，这个时候耦合就发生了。

事件驱动模型提供了一种松散的通讯方式，相比于接口通讯这种硬连接的方式。事件驱动更像是一种软链接，如同 Android 本身自带的 BroadCastReceiver 一样，事件的接收方和发送方彼此互不关心。发送方不知道有谁订阅了他的事件，订阅方也不知道谁何时何地向他发送事件。

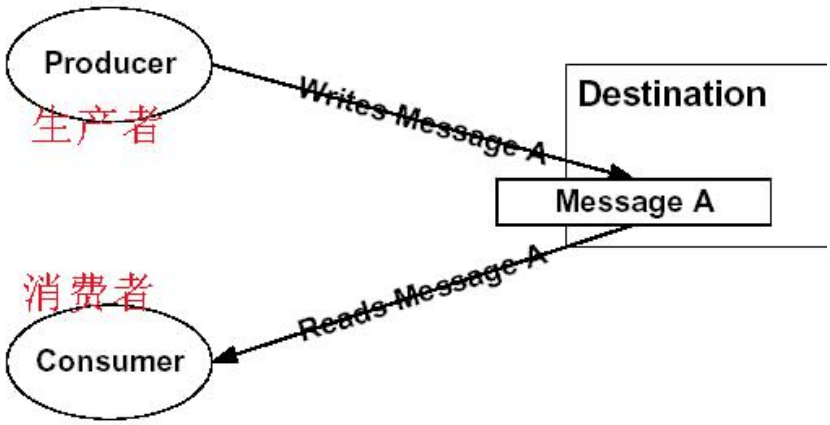



图 10-1 事件驱动模型

## 10.3 事件驱动框架原理

EventBus 2.0 和 3.0 两个版本的实现方式是有所不同的，但是这种不同点主要体现在 Event 的订阅上即对 Class 中 Method 解析上。

EventBus 作为事件订阅发布总线，其核心无非订阅和发布两种，熟练掌握反射或者注解处理器的人确实也是挺容易实现的。应为本质就是方法的全局存储 + 分类 + 全局动态调用。

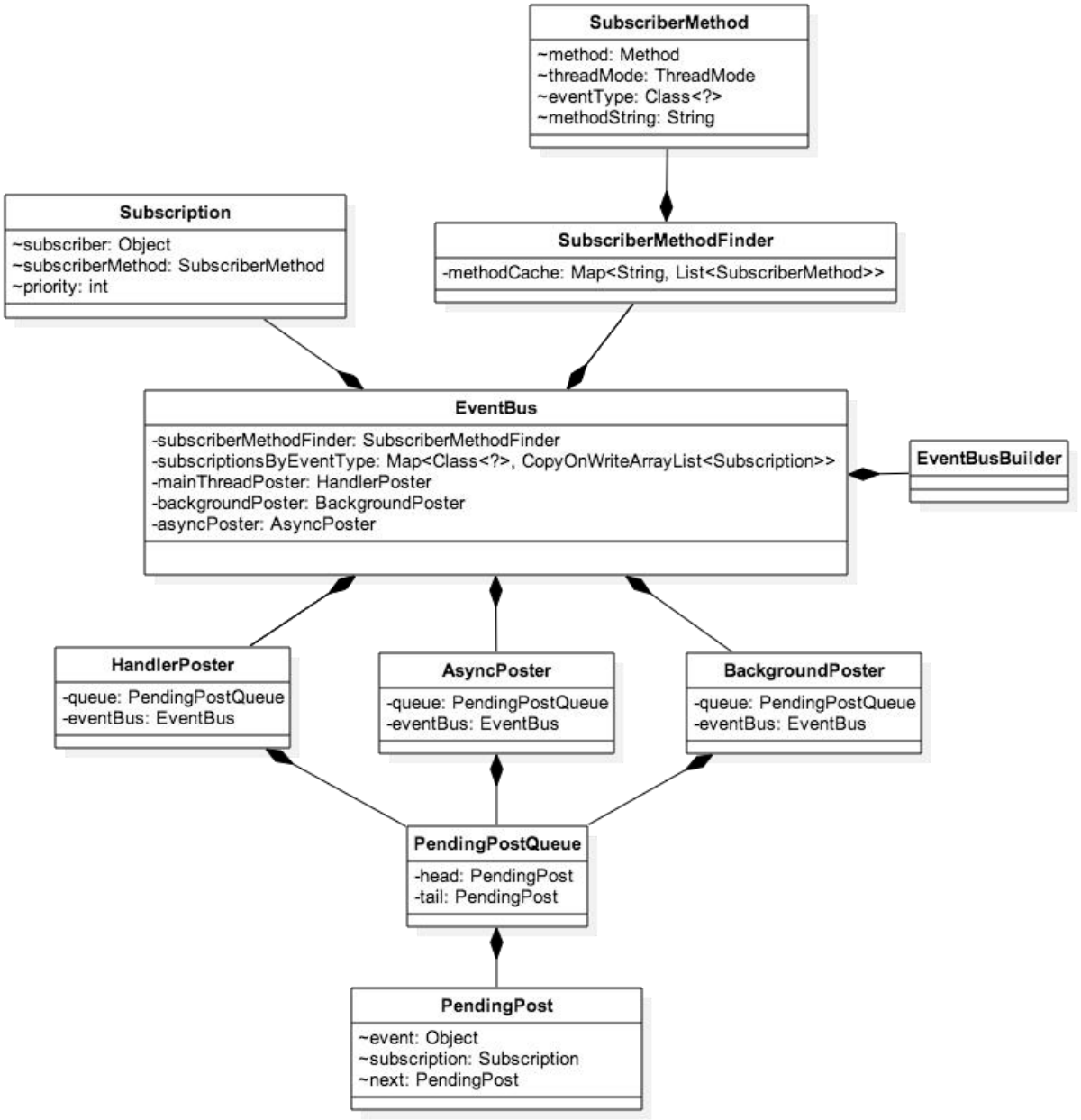



图 10-4 EventBus 核心类 UML

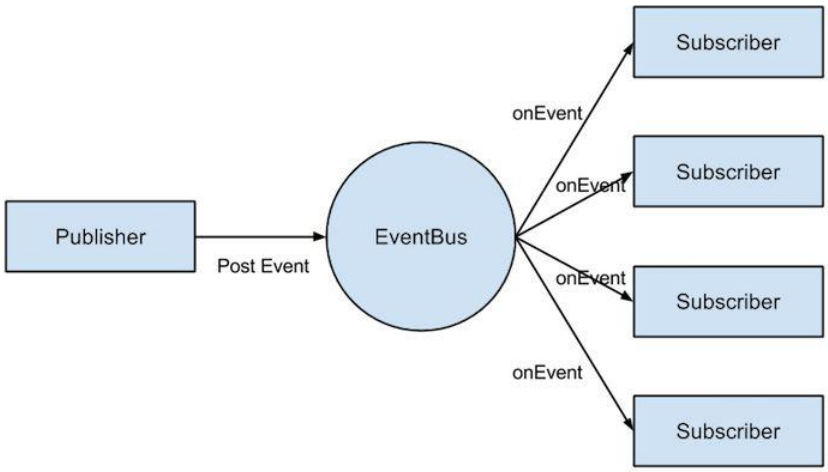



图 10-5 事件总线结构

事件订阅:

1.反射:事件订阅本质就是扫描注册的对象对应类型下面的 Method，如果发现某个 Method 上面附带了 Event 注解(2.0 之前 EventBus 使用方法名 + event 进行匹配，原因是老版本 DVM 对注解处理的效率过慢)，便会开始解析此方法的参数类型，方法的参数类型指定了接收事件的类型，使用全局 HashMap 将同一种 Event 类型的订阅保存起来，订阅实体中保存了方法，目标类型，目标对象等。

2.注解处理器:注解处理器会在编译期间解析带 Event 注解的方法，并生成代码显式调用该方法，而不是通过反射。

事件发布:程序在全局任何地方发布某种类型的 Event，EventBus 遍历调用同一种类型的事件订阅。

## 10.4 事件框架的实现 EventPoster

### 10.4.1 EventPoster 优势简述

相较于已有的 EventBus 和 Otto 这类事件框架。EventPoster 具有一下主要的优势:

1.模块化，可扩展。

2.高效的缓存管理。

3.预加载和预解析的支持。

4.统一的订阅者的对象管理，防止内存泄漏。

5.已有多种 Event 支持。

### 10.4.2 模块 Handler 以及分发

EventPoster 是模块化可扩展的，其实现这一特性的核心在于 EventPoster 核心本身不会直接参与到具体的某一种类型的 Event 的处理之中。只要是 EventPoster 所标记的 Event 类型，EventPoster 都会将它们收集起来，再通过这些 Event 类型所指派的 Handler 分发给对应的 Event 处理器。

EventPoster 管理了一整套事件注册和管理生命周期的流程，但是 EventPoster 本身并不 Care Event 的具体处理，当然 EventPoster 会缓存 Event 的原始 Entity 以供各个模块使用。

```
public interface IHandler<T extends IEventEntity> {
    void init(Object... objects);
    void destroy(Object... objects);
    T parse(EventAnnoInfo annoInfo);
    void load(T eventEntity, Object invoker);
    void unload(T eventEntity, Object invoker);
    void inject(Object object);
    void remove(Object object);
}
```

代码段 10-1 IHandler 抽象接口

Activity 生命周期事件的注解，可以看到注解上指定了对应的 Handler:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@EventBase(ActivityLifeHandler.class)
public @interface ActivityLife {
    ActivityLifeType lifeType();
    Class<? extends Activity>[] activity();
}
```

代码段 10-2 ActivityLifeEvent 的 Handler

从上面不难看出，想要扩展 EventPoster 实现自己的 Event 模块，只要实现 IHandler 接口，并且实现一个对应 Event 的注解就可以了，不要忘了在注解上指定你的 Handler。

### 10.4.3 性能优化

靠反射工作的事件框架其性能瓶颈的确在于反射本身之上，但是结合实际情况，现今的 Android 设备大多已经发展到 8 核甚至 10 核。事件总是在主线程注册的，传统的反射工作的事件框架，对类反射解析的工作是在主线程进行的。这无疑会影响 App 运行的性能。

那么不难看出优化的思路在于异步，我们可以在事件注册之前(一般在 Activity onCreate 中注册)之前提前对目标类型进行异步的反射解析。除此之外，解析后的结构化数据被统一保存在缓存中，由 EventPoster 统一管理。

当事件被注册时，各个模块就可以从缓存中直接读取反射完毕的事件信息，当时间被发布是，唯一的反射操作仅有 Invoke 了。

### 10.4.4 防止 Leak

在事件框架的使用中，注册和注销必须是成对的，但是忘记注销的情况在实际开发中也是时有发生的。这时候内存泄漏就发生了，EventPoster 统一使用弱引用管理注册队列。

当事件发布时，各个模块向 EventPoster 请求同类型的订阅者的实例，只有那些还在生命周期中的对象才会被推送目标时间。

# 第十一章 框架原理及实践

## 11.1 自己动手实现 JSON ORM 框架

Http 开发中我们经常使用 JSON 来传递数据，在出现 ORM 关系映射之前，需要我们手动遍历 JSON 树，费时费力。

事实上，JSON 的树形结构已经通过代码表达在了业务 Mdoel 中，我们完全可以解析业务 Model 来得到 JSON 树形结构，然后通过遍历树来遍历 JSON 结构。

那么实现一个 JSON ORM 框架重点如下:

我们选择先根据业务 Model 构建 JSON 结构树，然后将树缓存，以备用。如何根据 Model 类生成树？我们用反射遍历 Model 的 Field，如果遇到非基本类型的 Field，我们则继续递归便利该类型。

当有 JSON 字符串送来，要求转换成 Model 实例时，我们从缓存中取出 JSON 树，一边遍历树，一边遍历 JSON 字符串生成 Model 对象，给对象里的基本类型赋值。

缓存:每次解析类结构之前先去缓存读取，如果已经解析则直接读取缓存，避免重复反射解析，毕竟反射是高耗时操作，下面看一下 JSON 的树形结构:

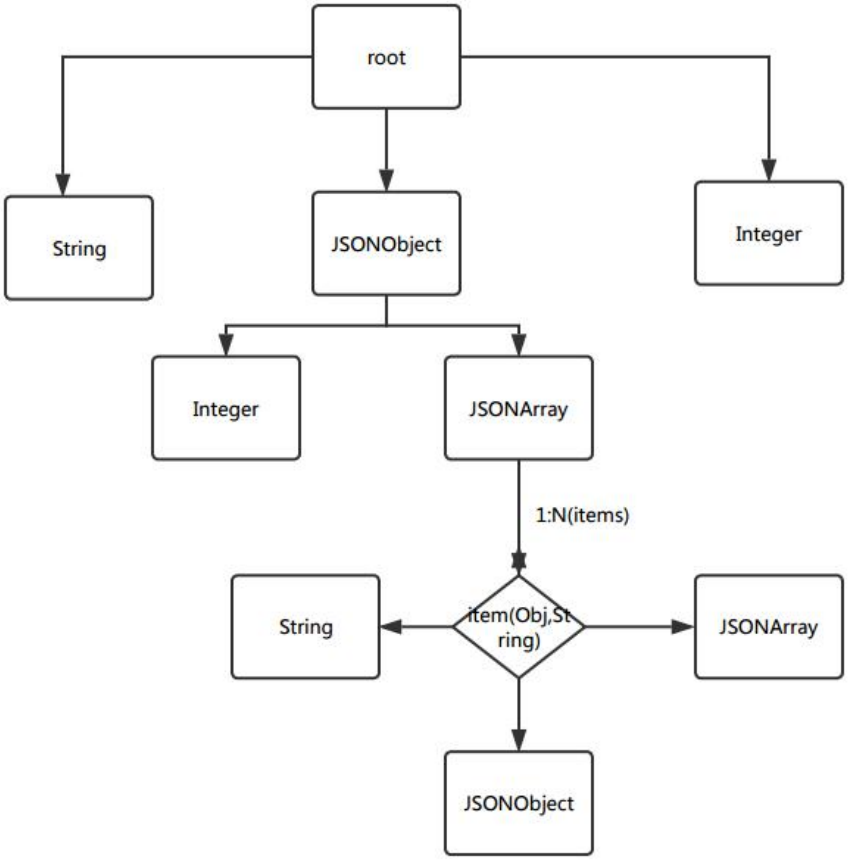


图 11-1 JSON 树形结构

实例代码,关键部分:

```
private static void establish(JsonMem mem){
    Field[] fields = mem.getType().getDeclaredFields();
    List<JsonMem> childs = new ArrayList<>();
    //遍历所有域
    for (Field field:fields){
        Annotation[] annotations = field.getAnnotations();
        if (annotations == null||annotations.length == 0)
            continue;
        field.setAccessible(true);
        //便利所有注解
        for (Annotation annotation:annotations){
            Class<? extends Annotation> type = annotation.annotationType();
            //如果元素是 JSONArray
            if (type == JsonSet.class){
                JsonSet set = (JsonSet) annotation;
                String key = set.name();
                if (key.equals(""))
                    key = field.getName();
                JsonMem jsonMem = new JsonMem();
                jsonMem.setKey(key);
                jsonMem.setJsontype(JsonObjType.ARRAY);
                jsonMem.setField(field);
                jsonMem.setType(set.clazz());
                //判断 JSONArray 的 item 是否是普通元素还是 JSONObject, 这里普通元素暂时偷懒用 String 代替
                //如果不是普通类型则需要递归解析
                if (jsonMem.getType()!=String.class)
                    establish(jsonMem);
                childs.add(jsonMem);
                //如果元素是 JsonObject
            }else if (type == JsonObject.class){
                JsonObject jsonObject = (JsonObject) annotation;
                String key = jsonObject.value();
                if (key.equals(""))
                    key = field.getName();
                JsonMem jsonMem = new JsonMem();
                jsonMem.setKey(key);
                jsonMem.setJsontype(JsonObjType.OBJECT);
                jsonMem.setField(field);
                jsonMem.setType(field.getType());
                //递归解析
                establish(jsonMem);
                childs.add(jsonMem);
            }else{
                //只是普通元素
                JsonBase jbase = annotation.annotationType().getAnnotation(JsonBase.class);
                if (jbase == null)
                    continue;
                String key =  getItem(annotation);
                if (key == null||key.equals(""))
                    key = field.getName();
                Class valueType = jbase.value();
                JsonMem jsonMem = new JsonMem();
                jsonMem.setKey(key);
                jsonMem.setJsontype(JsonObjType.ELEMENT);
```

```
                    jsonMem.setField(field);
                    jsonMem.setType(field.getType());
                    childs.add(jsonMem);
                }
            }
        }
        if (childs.size()!=0)
            mem.setChilds(childs);
}
```

代码段 11-1 JSON 树构建

```
public static String getJson(JsonMem mem,Object o) throws Exception{
    String jsonStr = null;
    Field field = mem.getField();
    Class type = mem.getType();
    List<JsonMem> childs = mem.getChilds();
    //依旧是判断元素类型
    switch (mem.getJsontype()){
        //如果是根节点
        case HEAD:
            JSONObject headobj = new JSONObject();
            if (childs!=null){
                //遍历子元素
                for (JsonMem child:childs){
                    Object co = mem.getField().get(o);
                    headobj.put(mem.getKey(),getJson(child,o));
                }
                jsonStr =   headobj.toString();
            }
            break;
        case ELEMENT:
            //普通元素直接赋值
            jsonStr =   mem.getField().get(o).toString();
            break;
        //JSONObject 需要递归其子元素
        case OBJECT:
            JSONObject jsonObject = new JSONObject();
            if (childs!=null){
                for (JsonMem child:childs){
                    Object co = mem.getField().get(o);
                    jsonObject.put(mem.getKey(),getJson(child,o));
                }
            }
            jsonStr =   jsonObject.toString();
            break;
        //如果是 JSONArrary
        case ARRAY:
            JSONArray array = new JSONArray();
            List list = (List) o;
            if (array == null||array.length() == 0)
                return null;
            //如果 item 是普通元素则直接赋值
            if (type == String.class){
                for (int i = 0;i < list.size();i++){
                    array.put(list.get(i).toString());
                }
            }else {
```

```
                    //否则递归
                    for (int i = 0;i < list.size();i++){
                        JSONObject subjobj = new JSONObject();
                        if (childs!=null){
                            for (JsonMem child:childs){
                                subjobj.put(child.getKey(),getJson(child,list.get(i)));
                            }
                        }
                        array.put(subjobj);
                    }
                }
                jsonStr = array.toString();
                break;
        }
        return jsonStr;
    }

    private static void getValue(JsonMem mem,String json,Object object) throws Exception {
        Field field = mem.getField();
        Class type = mem.getType();
        List<JsonMem> childs = mem.getChilds();
        //依旧是判断元素类型
        switch (mem.getJsontype()){
            //如果是根节点
            case HEAD:
                JSONObject headobj = new JSONObject(json);
                if (childs!=null){
                    //遍历子元素
                    for (JsonMem child:childs){
                        String str = headobj.getString(child.getKey());
                        getValue(child,str,object);
                    }
                }
                break;
            case ELEMENT:
                //普通元素直接赋值
                field.set(object,json);
                break;
            //JSONObject 需要递归其子元素
            case OBJECT:
                JSONObject jsonObject = new JSONObject(json);
                Object valueobj = type.newInstance();
                if (childs!=null){
                    for (JsonMem child:childs){
                        String str = jsonObject.getString(child.getKey());
                        getValue(child,str,valueobj);
                    }
                }
                field.set(object,valueobj);
                break;
            //如果是 JSONArrary
            case ARRAY:
                JSONArray array = new JSONArray(json);
                if (array == null||array.length() == 0)
                    break;
                List list = new ArrayList();
                //如果 item 是普通元素则直接赋值
```

```
                if (type == String.class){
                    for (int i = 0;i < array.length();i++){
                        list.add(array.getString(i));
                    }
                }else {
                    //否则递归
                    for (int i = 0;i < array.length();i++){
                        Object arritem = type.newInstance();
                        JSONObject subobj = array.getJSONObject(i);
                        if (childs!=null){
                            for (JsonMem child:childs){
                                String str = subobj.getString(child.getKey());
                                getValue(child,str,arritem);
                            }
                        }
                        list.add(arritem);
                    }
                }
                field.set(object,list);
                break;
            }
        }
```

代码段 11-2 根据 JSON 树构建 Model 对象

## 11.2 图像异步加载框架原理及实现

图像异步加载是 Android 开发中必要的基本组件，为了避免主线程阻塞，图像的异步加载是必要的，github 上有很多优秀的实现，不过整体架构都大同小异。

图像异步加载框架重点在于异步，加载，缓存，处理(压缩)，显示这几部分。大概步骤即是用户调用传入 URL 和 ImageView -> 以 URL 为 Key 到缓存中搜索 -> 没有搜到则包装一个异步下载任务 -> 把下载任务扔到线程池中 -> 下载完毕保存到 Cache 显示图像。

### 11.2.1 Cache

在图像异步加载中，必然需要双重 Cache，即 RamCache 和 DiskCache。基本逻辑是先从 RamCache 中搜索，如果没有则去 DiskCache 搜素，再没有则去网络拉取，然后保存到两个 Cache 中。

缓存策略方面，无论 RamCache 还是 DiskCache 都适合 LRU 策略，即最早未被使用的 Cache 会被最先淘汰。

在 RamCache 中，使用 LinkedHashMap 保存 Cache 实体，因为 LinkedHashMap 的特性:最新 get 的元素会被放置到链表头部，那么最后未被 get 的元素就会在链表尾部，除此之外，因为所保存的对象 Bitmap 单个体积很大，一个普通图片的 Bitmap 对象的大小也不容乐观，可能占用超过 10M 的 Ram，而 Bitmap 对象的大小是可计算

的:= 每像素大小 * 高 * 宽。所以 RamCache 还需要对 Ram 大小加以控制。

```
long getSizeInBytes(Bitmap bitmap) {
    if (bitmap == null)
        return 0;
    return bitmap.getRowBytes() * bitmap.getHeight();
}
```

代码段 11-3 计算 Bitmap 大小

```
private void checkSize() {
    if (size > limit) {
        Iterator<Entry<String, Bitmap>> iter = cache.entrySet().iterator();
        while (iter.hasNext()) {
            Entry<String, Bitmap> entry = iter.next();
            size -= getSizeInBytes(entry.getValue());
            iter.remove();
            if (size <= limit)
                break;
        }
    }
}
```

代码段 11-4 检查缓存池大小，超过阈值就释放

### 11.2.2 线程池

异步加载是一个并发数量很高的操作，一个界面可能同时加载十几张图片，但是处理机的数量是有限的，并且 new 一个 Thread 是一个高消耗的操作，引入 ThreadPool 可以解决 Thread 复用的问题，减少增加新的异步任务时的消耗，减少了垃圾收集器(GC)的压力。

如何实现一个线程池:

```
ThreadFactory factory = new WorkThreadFactory("thread-pool",
        android.os.Process.THREAD_PRIORITY_BACKGROUND);
BlockingQueue<Runnable> workQueue = new LinkedBlockingDeque<Runnable>();
mExecutor = new ThreadPoolExecutor(configs.getPOOL_SIZE(), configs.getMAX_POOL_SIZE(),
        configs.getKEEP_ALIVE_TIME(), TimeUnit.SECONDS, workQueue, factory);
```

代码段 11-5 简单线程池

### 11.2.3 图片压缩

很多时候，下载的图片大小可能远远超过 ImageView 的尺寸，因为手机的屏幕大小是有限的，而这样就大大浪费了内存空间，所以根据 ImageView 的最大尺寸压缩 Bitmap 是非常有必要的。这样可以极大的减少内存开支。

### 11.2.4 其他优化

Context 单实例优化，根据 ImageLoader 的使用场景，一个 Context 对象只需要一个 ImageLoader 实例，根据 Context 缓存 ImageLoader 对象也是可以优化内存的消耗的。

堆外直接内存，为了避免 OOM，我们可以使用 JVM 堆外的原生内存存储 Bitmap 对象，这样内存就不受 JVM 管理，从而减轻 JVM GC 压力。

## 11.3 Http 请求框架 Retrofit

Retrofit 所做的事情简单来说就是将你的 Http 业务 API --> Http 请求实现，类似于 Spring MVC 中的 Controller，它的主要任务是解析你的业务接口，从接口上获取你的 Http 接口协议，然后组装 Http 请求，进行异步 Request。

Retrofit 整合了多个组件，包括 JSON/XML 的 ORM 映射，用于解析返回值；Http 请求驱动用语发送 Http 请求，Retrofit 默认使用的是 Okhttp；异步处理框架，包括 Observable。

### 11.3.1 原理

如何生成一个接口的实现类，除了手动实现该接口之外，还有另外一种选择，那就是动态代理；Retrofit 的核心正是如此。

如同 SpringMVC 的 Controller 一样，在方法描述上(方法描述包括 Modifiers，参数列表，返回值类型，异常列表)，再加上参数列表上的注解，方法体上的注解，通过一个方法的描述就可以基本确定一个 Http 接口的所有协议了。

1. 请求目标地址和请求方法：通过解析方法体上的注解。类似这种 @POST("/mobile/login") 代表请 POST 方法，地址“Domain/mobile/login”。

2.请求参数:通过解析参数列表和参数列表上的注解，如(@Query("loginname")String loginName, @Query("pass")String pass) 代表其有两个参数 loginName，pass。

3. 请求返回值，通过解析方法的返回值类型。例如 Observable<BaseResponse<UserInfo>> Observable 是异步可观测数据 RxJava 的包裹类，真正的业务 Model 类型在范型里面。

### 11.3.2 源码分析

先来看创建 API 代理的入口方法 Retrofit.create(Class class):

```
public <T> T create(final Class<T> service) {
  Utils.validateServiceInterface(service);
  if (validateEagerly) {
    eagerlyValidateMethods(service);
  }
  return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
      new InvocationHandler() {
        private final Platform platform = Platform.get();
```

```
        @Override public Object invoke(Object proxy, Method method, Object... args)
            throws Throwable {
          // If the method is a method from Object then defer to normal invocation.
          if (method.getDeclaringClass() == Object.class) {
            return method.invoke(this, args);
          }
          if (platform.isDefaultMethod(method)) {
            return platform.invokeDefaultMethod(method, service, proxy, args);
          }
          ServiceMethod serviceMethod = loadServiceMethod(method);
          OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
          return serviceMethod.callAdapter.adapt(okHttpCall);
        }
      });
}
```

代码段 11-6 Retrofit 入口方法

1.Utils.validateServiceInterface(service):

检查传进来的 Class 对象是否合法，主要检查两个方面一是 Class 是否是接口，二是接口一定不能继承其他接口。

2.eagerlyValidateMethods(service):

validateEagerly 这个参数控制了是否是在 create 是提前解析业务接口并且进行缓存，否则的话将需要在业务方法调用时解析。

3.Platform:

如字面所描述平台类，Retrofit 专门为操作系统运行环境抽象了一层，事实上，Retrofit 可以运行在 Android，IOS，Java8 三个平台上。获取比较诡异的事为什么 Java 代码可以在 IOS 平台运行，事实上是借助了一个叫 robovm 的第三方 JVM，

Class.forName("org.robovm.apple.foundation.NSOperationQueue"); 这段代码就是获取 robovm 的线程池。Platform 主要提供了异步所需要的线程池，异步同步控制，而这个是一个抽象的接口，在 Retrofit 初始化时是可以指定其实现的。CallAdapter 就是上述接口。

4.invokeDefaultMethod:

这个对应了 Java8 中的 default 方法，暂时没有实现，会抛出 UnSupportOperationException，也就是说你不可以调用接口中的 default 方法。

5.loadServiceMethod(method):

加载 API 方法的实现，即加载动态代理，前面说过了，如果在 create 时已经解析过接口了的话，代理对象直接在缓存中找到。如果没有则需要在这个时候解析并且存储到缓存中，下次调用同样的接口就不需要重复解析了，毕竟业务解析大量用到了反射这种耗时操作。

6.ServiceMethod.Build():

解析业务接口方法并且生成对应的动态代理。

解析返回值类型: Utils.getRawType 从返回值的范型中扣出真正的业务类型，应为返回值一般不会直接写业务 Model，而是对异步任务的包裹，对 Callback 的封装。

createResponseConverter 和 parseMethodAnnotation 开始创建 Http 请求，这时候需要解析方法上的 Annotation，这个 Annotation 包含了 URL，和请求方法。获取完了这些信息之后，这个方法就开始根据这些参数拼装完整的 URL。

parseParameter 解析方法参数，获取请求参数。这个方法主要就是调用了 parseParameterAnnotation 解析参数列表上的注解，注解中包含了参数类型，参数名，Header，Path 等信息。参数类型也可以是 Form，Part 等复杂类型。

createCallAdapter 上面说过了，Retrofit 只是一个业务解析器，组装器，诸如返回值的 JSON/XML 解析，Http 请求底层等等都是交给其他框架处理的，Adapter 就是抽象接口，createCallAdapter 就是根据配置进行工具组装，组装完毕，请求到解析一整套流程就可以顺利执行了。

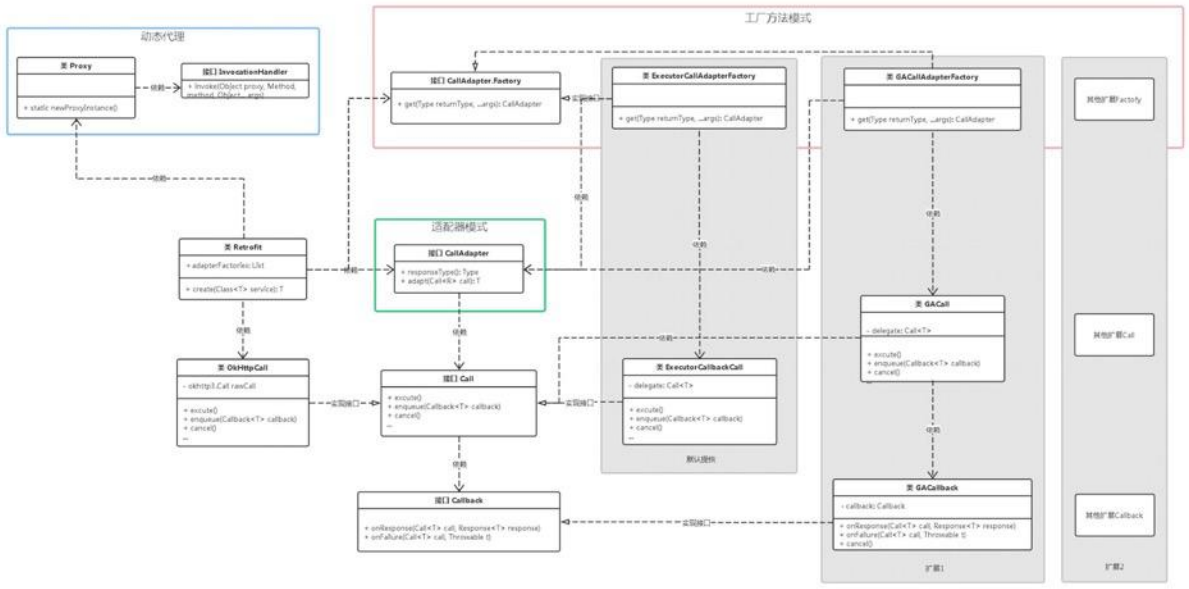



图 11-2 Retrofit 核心类 UML

# 第十二章 案例实践

## 12.1 案例简述-数读

数读是我在广州爱范儿公司重构的新闻 App 客户端。其核心在于突出数字为新闻的重点，特点在于界面简单，动画美观大方，再加上支持手表客户端，包含一个可以看新闻和天气的表盘和一个手表端的小 App。

由于数读是商业项目，所以无法贴代码以及具体细节。

## 12.2 需求简述-数读

老版本的数读由于是外包给其他公司做的，而且最新版本完成于 2012 年，架构老旧，最新版本代码也无法找到，老版本代码亦不具备可读性和可维护性。

老版本 Bug 很多，包括 View 滑动特效很容易复现类似卡死，重叠等 Bug，对网络的异常处理不完善，网络不稳定时容易出现 Bug。

需要对老版本功能进行大量升级，包括增加收藏，分享，以及增加手表表盘以及手表 App 客户端。

所以公司决定对数读进行完全重构。

## 12.3 重构之前的程序结构-数读

老版本数读程序在重构之前整体处于 Android 架构的第一阶段，具体可以参考绪论的 1.2.1。

异步和回调处理:使用 Android 系统提供的 Handler，Looper。Activity 中包含了大量的自定义 Hadnler，并且需要使用 WeakReference 处理内存泄漏问题。

网络请求:还在使用过时的 HttpClient，这个类已经在 Android5.0 的时候被 Google 从 Android 官方类库中移除了，除此之外每一次网络请求都需要重复的做异步处理，并且需要手动解析 JSON 数据。没有对网络超时做异常处理，导致网络超时时首页 Spalsh 不跳转到主界面。

Cache 与持久化存储:依然是使用了系统自带的 SharePreference，也没有对其做任何封装。

总的来说原有的架构混乱不堪，纵向结构过于扁平，所有流程全部挤压在 Activity 内部，没有做合理的分层，导致 Activity 混乱不堪。异常处理分散让异常处理有很多疏漏，导致程序不够健壮，业务上逻辑混杂，没有做业务上的明确划分。封装不够导致，

存在大量重复代码。

## 12.4 重构后的程序结构-数读

### 12.4.1 架构裁剪-数读

根据项目实际对架构进行裁剪:由于数读业务相对简单，只是一个较为简单的资讯阅读类 App。所以我们需要对上面所设计的重量级架构做适当的裁剪。

1.Bundle 容器:模块化在数读这种业务简单的 App 中是没有必要的，因为其本身也就包含了 1-2 个模块。即手机新闻客户端模块+手表端模块，且手表端模块根据 Google 要求其实是一个单独的 App，他和手机端之间是通过蓝牙进行通讯。并不是运行在同一空间中。

2.页面路由:因为不存在模块间的导航，并且业务页面也不多，系统自带的 Intent 相对够用，所以页面路由在本项目中是不需要的。

## 12.4.2 UML 类图-数读

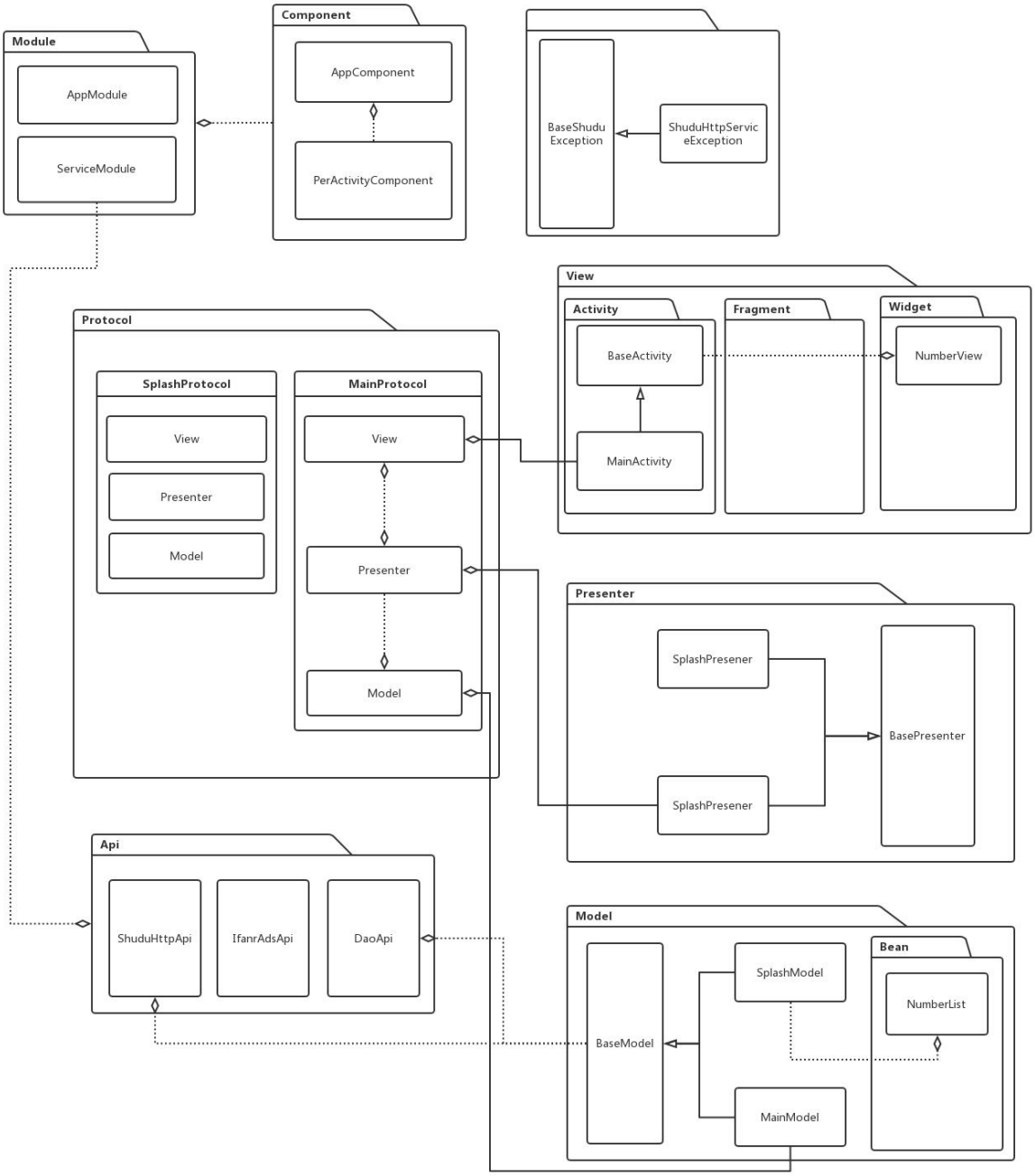



图 12-1 新数读大致 UML 类图

## 12.4.3 主要功能需求-数读

1.主页左右上下滑动效果，上下滑动收放菜单，显示收缩咨询内容。


图 12-2 数读主页功能

2.收藏以及内容回顾


图 12-3 文章收藏和回顾页面

3.手表端表盘以及简单阅读 App:


图 12-4 手表表盘


图 12-4 手表阅读 App

### 12.4.4 重构带来的改变

首先整个架构无论是从业务单元的切割，还是业务流程的分层来说，都变得无比清晰，代码的可读性变的非常高。

可读性上，刚接手这个项目，先去看整体架构说明文档，了解具体业务的话，直接找到协议层，整个业务流程可以通过各层接口看的一清二楚。

可测试性，合理的分层和组件化，再加上 DI，让单元测试变的轻松有条理，我们直接找到 DI 的依赖图换成测试依赖，而不必在业务代码中东翻西找。

可扩展性上，高可重用用的组件化架构使得添加业务仅仅只需要实现几个接口和业务流程。

鲁棒性上，由于健壮的异常处理，数读重构之后几乎没有发生未捕获的异常造成的 App Crash 现象。统一的异常收集帮助产品更好的监控 App 的运行健康。

总的来说，重构之后的数读 App 拥有清晰健壮并且可伸缩的架构，无论是各个方面都远远优于原来的版本。

# 结　论

架构设计日新月异，本文是开源技术在这一个阶段的总结，但是技术是在不断进步的，每一次发现问题都是进步的动力，我们在平时的编码中要知其所以然。编码好坏的标准永远不是能不能实现，而是做的够不够好。本文提出的立体架构能在一定程度上为 App 开发打一个好的地基，无论是在项目的工程期，App 的运行期，后续的 App 维护阶段都发挥了一定作用。并且经过了上线 App 的实际检验。但是对于小型 App 来说有些设计还是没有必要的，这样的架构反而略显笨重了。

# 致 谢

感谢无私的开源精神给予了软件开发行业蓬勃发展的今天，无论是 Android 还是其他平台软件的开发，一路走来诞生了无数优秀的开源框架和作品，让我们有机会一睹优秀作品的风采。最后感谢老师的耐心指导和修改建议。

# 参 考 文 献


[1] ReactiveX. Reactive Extensions for the JVM https://github.com/ReactiveX/RxJava [DB/OL]. Github, 2016

[2] Wequick. Small 开源组件化框架 https://github.com/wequick/Small [DB/OL].Github, 2014

[3] Alibaba. Atlas 阿里巴巴开源组件化框架 https://github.com/alibaba/atlas [DB/OL]. Github, 2016

[4] Retrofit. Square 异步 Http 请求框架 https://github.com/square/retrofit [DB/OL]. Github, 2016

[5] Greenrobot. EventBus 事件驱动框架 https://github.com/greenrobot/EventBus [DB/OL]. Github, 2016

[6] IBM. AspectJ AOP 框架 https://github.com/eclipse/org.aspectj [DB/OL]. Github, 2016

[7] Wiki.SoftwareArchitecture https://en.wikipedia.org/wiki/Software_architecture [DB/OL]. Github, 2016

[8] Steve McConnel.“代码大全第二版”（Riel 1996）[M]. 电子工业出版社,2006.

[9] 甘尧. 打造超越 EventBus 的事件管理框架 http://blog.csdn.net/ganyao939543405/article/details/52847648 [DB/OL]. CSDN, 2016

[10] 甘尧. Android MVP 的实现 http://blog.csdn.net/ganyao939543405/article/details/52963144 [DB/OL]. CSDN, 2016

[11] Google. Android 架构蓝图 https://github.com/googlesamples/android-architecture [DB/OL]. Github, 2016.

[12] Alibaba. Atlas 官网 http://atlas.taobao.org/ [DB/OL]. Taobao, 2017.

[13] Square. Picasso 加载框架 http://square.github.io/picasso/ [DB/OL]. Square, 2014.

[14] Google. Android 开发文档 http://developer.android.com/ [DB/OL]. Google, 2007.

[15] 迷死特兔. 对软件架构设计的一些总结和理解 http://blog.csdn.net/cooldragon/article/details/48241965 [DB/OL]. CSDN, 2015.