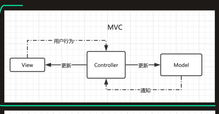
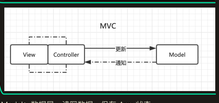
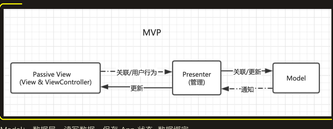
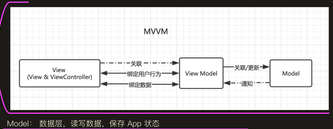
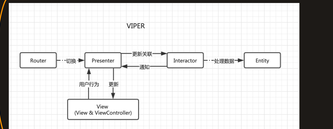
- iOS架构
  - 软件架构的七大原则：
    - 1.开闭原则
      - 开: 对模块扩展新功能
      - 闭: 尽量不改老接口
    - 2.依赖倒置原则
      - 面向接口编程,尽量依赖抽象接口，不依赖具体实现
    - 3.单一职责原则
      - 高层模块, 底层模块 ,细节功能 均互不依赖，但是都依赖抽象
      - 一个类、接口尽量变成相对单一的功能, 方便进行解耦和维护
    - 4.接口隔离原则
      - 类与类之间的接口依赖应该建立在最小接口(尽量适当小)上
      - 依赖单一的, 而不是功能繁多的总接口或者无关的接口
    - 5.迪米特法则
      - 最小知道原则：一个对象应该对其他对象保持最少的了解，尽量降低类与类之间的耦合
      - 功能模块之间不要存在相互依赖, 就算有也是通过他的友元类来转达
    - 6.里氏替换原则
      - 子类能够替换父类, 而不需要做适配
      - 子类可以增加方法, 不能覆盖
        - 重写/重载或实现抽象方法 要比父类方法的输入参数更宽松, 方法的输出/返回值 要比父类更严格
    - 7.合成/聚合复用原则
      - 尽量不通过继承关系达到软件复用的目
      - 可以通过协议代理, 委派等等达到复用,降低类与类之间的耦合度
  - 架构分层
    - 第一层：基础架构
      - 围绕MVC展开的MVP、MVVM、MVCS 、VIPER 等等
      - 主要是设计模式的改进
    - 第二层：增长期
      - 综合型应用架构：针对业务和功能模块进行架构
      - 一个发展迅速的企业APP, 必定对业务的快速响应要求极高, 以及功能模块的复用和解耦很重要
    - 第三层：快速迭代期
      - 如果项目更偏向于运营需求的(比如电商类) , 那么对迭代的要求变得更重要
      - 跨平台开发：多端一体开发, 兼容性,协调性等, 前后端高统一等等
    - 第四层：深度发展期
      - 当项目发展壮大,框架基本构造完成时, 就需要深入细分化项目
      - 组件化：组件或模块划分细分(粒度变小), 通用
      - SDK的深入：深度优化SDK相关个功能
        - 网络, 多线程, 数据存储 , FPS , 完全性
      - 自己造轮子：
        - 第三方或者开源框架产重不能满足需求时, 就需要自己造轮子
    - 第五层：成熟期
      - 当一个项目趋于成熟时, 其架构也基本保持稳定, 保持项目的高质量,高效率,高稳定 迭代最为重要
      - 规范化: 流程化：
        - 开发人员准大,水平不一
        - 模块演进, 流程规范
  - MVC
    - 关系图:
      - 
        
      - 
        
    - 职责：
      - Model：数据层，读写数据，保存 App 状态
      - View：页面层，和用户交互，向用户显示页面，反馈用户行为
      - Controller：逻辑层，更新数据，或者页面，处理业务逻辑
    - 说明：
      - Controller 会成为所有东西的代理或者数据源，数据绑定,通知更新, 网络等等
        - 没有区分业务逻辑和业务展示
        - 变得越发臃肿,单元测试难度加大
  - MVP
    - 关系图:
      - 
        
    - 职责：
      - Model： 数据层，读写数据，保存 App 状态, 数据绑定
      - View/ViewController：页面层，提供用户输入行为，并且显示输出状态
      - Presenter: 逻辑层，它将用户输入行为，转换成输出状态
    - 说明：
      - V触发用户行为给P, 由P去更新M, M处理完逻辑后告知P, 再由P去更新V
        - 方便单元测试，只要保证P层逻辑正常, 相关联的V和M也能正常
      - MVP更多的是使用代理模式, 但是一旦业务变得更复杂, 代理增多
        - 弹框, 消息, 加载提示等等增多, 代码量越大, 代理方法增加
        - 在View中需要P的代理方法去调用, P又关联着V, 耦合度高
  - MVVM
    - 关系图:
      - 
        
    - 职责：
      - Model： 数据层，读写数据，保存 App 状态
      - View：页面层，提供用户输入行为，并且显示输出状态
        - ViewController：主要负责数据绑定
      - ViewModel 逻辑层，它将用户输入行为，转换成输出状态 ,数据绑定
    - 说明：
      - VP直接返回VM的block, VM需要告知V的事情可以通过block回调通知
      - V的行为告知VM去更新M的数据, 同时VM监听着M的数据, 一旦M改变, VM就会去告知V
      - MVVM大量使用监听, 而RAC框架恰恰是MVVM最好的帮手, 这也促使其发展, 同时像弹框类似功能一段回调就搞定了, 无需代理去绑定
      - 优点：
        - 摘自网络:
        - 低耦合：View 可以独立于Model变化和修改，一个viewModel 可以绑定到不同的 View 上
        - 可重用性：可以把一些视图逻辑放在一个 viewModel里面，让很多 view 重用这段视图逻辑
        - 独立开发：开发人员可以专注于业务逻辑和数据的开发viewModel，设计人员可以专注于页面设计
        - 可测试：通常界面是比较难于测试的，而 MVVM 模式可以针对viewModel来进行测试
      - 缺点：
        - 数据绑定使得Bug 或调试 很难被分析,定位
          - BUG也很难快速定位到问题根源
          - 调试时通过对象原型查看数据内容不如直接通过NSDictionary/NSArray直观
        - 数据绑定和数据转化需要花费更多的内存（成本），大量block都是需要copy成本
        - 数组内容的转化成本较高：数组里面每项都要转化成Item对象, 转化之后的数据在大部分情况是不能直接被展示的, 为了能够被展示, 还需要第二次转化
  - VIPER
    - 关系图:
      - 
        
      - Presenter(展示器):包括 UI 相关（UIKit 之外）的一些业务逻辑，调用 Interactor 中的一些方法
    - 职责：
      - Interactor(交互器)：包括和数据相关的业务逻辑（Entities）或者网络请求，比如创建 entities 类的对象或者把它们从服务器中抓取出来。为了达到这些目的你会用到通常会被看做外部依赖而不被看做 VIPER 单元的一些服务（Services）和管理者（Managers）
      - Router(路由):负责 VIPER 模块之间的切换
      - Entities(实体):纯粹的数据对象，并非是数据访问层，数据访问是 Interactor 层的任务
      - View/ViewController：页面层，提供用户输入行为，并且显示输出状态
    - 说明：
      - 路由：
        - 项目存在大量的跳转逻辑, 刺离跳转和切换逻辑给路由去完成, 常见有url, 对url进行封装增强编译语法检查等
      - 实体：
        - 处理所有的业务逻辑
        - 也可以拆分出数据存储等等
  - 基础架构
    - 模块化：
      - 快速迭代初期,对功能进行模块化, 可以加快开发速度, 降低耦合度, 提高功能复用等等
    - 跨平台：
      - 对于类似电商需求的APP, 快速更新, 多端同步 更为迫切 , 配合H5共同, 分段开发是得尤为重要
        - iOS和Android采用 原生 + 跨平台 + H5
        - 稳定不变的功能使用原生
        - 经常变动和细节修改频繁的(如订单, 商品列表) 采用跨平台+H5混合开发
        - H5先上线, 再逐步开发原进迭代 (页面的切换,跳转均预留逻辑或者由后台返回统一支配)
      - 除了掌握：OC , Swift , 还需要：Web(vue , jquery 等至少一种) , 跨平台 ( Flutter (Google) 、React Native (Facebook) 、WEEX (阿里巴巴) 等至少一种)
        - 至少学习一门主流跨平台开发语言
    - 视觉(流程)化：
      - 随着项目需求的多样化, 组件化开发被越来越多的人讨论, 网上也有很多关于组件化的blog
  - 组件化开发
    - 参考：
      - 滴滴组件化架构
        - https://mp.weixin.qq.com/s?__biz=MzUxMzcxMzE5NQ==&mid=2247488503&idx=1&sn=2c9a82593ebb06833948417203c54550&source=41#wechat_redirect
      - 淘宝组件化架构
        - https://yq.aliyun.com/articles/129
      - 组件化架构漫谈 (详细对比多种方案):
        - https://www.jianshu.com/p/67a6004f6930
      - 组件化Demo：
        - https://github.com/DeveloperErenLiu/ComponentArchitecture
      - 蘑菇街 App 的组件化之路
        - https://limboy.me/tech/2016/03/10/mgj-components.html
      - iOS应用架构谈 组件化方案
        - https://casatwy.com/iOS-Modulization.html
      - 美团点评
        - https://www.jianshu.com/p/d9246be5ebc6
      - 简明的介绍,私有库配置:
        - https://www.jianshu.com/p/f472fa9f0616
      - 等等还有很多，但是组件化不是适合所有项目, 每个项目组件化的方案也是不一样的, 看需求
  - 组件化架构流程
    - APP组装(一切皆组件)
      - 原生组装
      - 服务器下发结构列表 - - - -> 解析列表 - - - -> 组装APP
    - 路由组件
      - 比如MGJRouter, 统一调度
    - pods
      - 资源文件Assets
        - 图片,txt 等等
        - sqlite
        - js、html
      - 业务层组件
        - 业务功能界面,UI等等
        - 注册地址
        - 基于MVC , MVVM , VIPER 等等
      - 核心组件(相互之间可能依赖, 也可能依赖基础组件)
        - 拆分出的业务核心功能功能
        - 安全性要求组件以 framework 等二进制形式
        - 基于MVC , MVVM , VIPER 等等
      - 基础层框架/组件 (相互之间不依赖, 也不依赖上层核心组件)
        - 宏定义, 公用方法 等等组件
        - 拆分出的基础功能组件
        - AF,SWWeb 等等