- 设计模式
  - 30种
    - 设计原则
      - 说明： 为了代码可重用性、让代码更容易被他人理解、保证代码可靠性等等, 设计模式 首先要满足以下7点设计原则
      - 单一职责原则
        - 一个类只承担一个职责, 尽量不要身兼数职(功能)
      - 开闭原则
        - 开：类、模块、函数， 可以去扩展(增加方法和属性),
        - 闭：但原有方法尽量不直接修改， 尽量用继承或组合的方式来扩展类的功能
      - 里氏替换原则
        - 子类可以直接替换父类,而不用做适配(子类可以扩展父类的方法， 但不应该复写父类的方法)
      - 接口隔离原则
        - 对象不应被强迫依赖它不使用的方法(不要把所有功能都集中到一个接口里)
        - 强迫所有动物都有飞 的功能就是不合理的, 应根据动物种类不同设计相符的接口
      - 依赖倒置原则
        - 高层模块不应该直接依赖低层模块， 二者都应该依赖其抽象； 抽象不应该依赖细节； 细节应该依赖抽象
        - 加载图片直接使用sd_weblmage方法,一旦换框架或者方法名改变, 高层需要大量修改(依赖低层)
        - 在它们之间建抽象层返回xx_weblmage方法, 让抽象层其调用具体实现方法
      - 迪米特法则
        - 一个对象应该对其他对象保持最少的了解， 实现低耦合、高内聚
        - 视频播放关系播放方法即可, 不要去关心视频卡顿, 如何渲染等, 让别人自己处理
      - 组合/聚合复用原则
        - 少用继承， 多用合成关系来拥有已存在的功能
          - 一个模块不要通过继承另一个模块方式来扩展功能,一个模块一旦修改, 它的继承者也必须修改, 耦合太强, 无法复用
        - 人可以是老师,学生,校长, 继承只能选一种, 无法满足即是校长又是老师的多重身份, 可以赋予一个角色对象, 让角色去配置相应的职责
    - 创建型模式
      - 说明: 使用合适的方式创建对象 不同需求使用不同的创建方式
      - 抽象工厂模式
        - 通过一个类的不同接口,直接返回不同的子对象, 无需知道如何创建的
          - 苹果工厂同时生产苹果鼠标和苹果键盘, 想要苹果鼠标：苹果工厂.创建苹果鼠标(); 即可
        - 缺点：需要一个对象就必须把其工厂一起继承过来
          - 想要苹果鼠标 必须继承 苹果工厂
      - 建造者模式
        - 通过创建多个不同子对象, 来组合成一个复杂的对象(而不是直接创建, 存在多种不同的组合情况)
          - 优点： 1、建造者独立， 易扩展。 2、便于控制细节风险。
          - 缺点： 1、产品必须有共同点， 范围有限制。 2、如内部变化复杂， 会有很多的建造类。
        - 比如套餐A: (汉堡+薯条+可乐) , 套餐B: (汉堡+鸡翅) 等等不同组合
      - 工厂方法模式
        - 定义一个创建对象的接口， 让其子类自己决定实例化哪一个工厂类， 工厂模式使其创建过程延迟到子类进行
          - 汽车工厂有不同接口使得生产出不同配置的汽车, 比如红色,蓝色, 或者有天窗和无天窗的
          - 生产汽车是汽车工厂类, 生产飞机是飞机工厂类, 不是在一个类里
      - 原型模式
        - 通过拷贝这些原型创建新的对象
          - 通过复制的方式创建一个对象(区别深复制和浅复制)
      - 单例模式
        - 一个类仅有一个实例， 并提供一个访问它的全局访问点 , 只能被实例化一次(iOS中dispatch_once修饰) , 全局访问, 运行过程不销毁, (适应一个全局使用的类频繁地创建与销毁时, 节省系统资源)
    - 结构型模式
      - 适配器模式
        - 将一个类的接口(无法直接使用,不兼容)转换成客户能直接用的另外一个接口(兼容的)
        - 不同类型的内存卡都不能直接插在电脑端口上使用, 但是中间接上读卡器(适配器)就可以了
          - 就是通过接口兼容适配,格式转换等产生新的可使用的接口
      - 桥接模式
        - 当对象是由多个维度不断变化组合产生时, 如果直接继承就会导致子类种数爆炸(三维排列组合：n*m*k)
        - 比如画一个有颜色的形状(二维:颜色+形状), 不同形状和颜色组合和种类繁多, 那么桥接如何使用 ？
          - 形状和颜色由单独的类(Color 和 Shape)去完成, 而对象能够接受这两个类对象(红色:Color.red , 圆形: Shape.circle(...)):
          - 1.需求画一个红色的长方形(宽20,高30)
            ```
            obj.setColor(Color.red);
            obj.draw(Shape.rectangle(w:20,h:30));
            ```
          - 2.需求画一个蓝色的圆形(半径30,x)
            ```
            obj.setColor(Color.blue);
            obj.draw(Shape.circle(r: 10, x: 100, y: 100));
            ```
      - 组合模式
        - 树形结构：就是在一个对象中包含其他对象， 这些被包含的对象可能是终点对象（不再包含别的对象） ， 也有可能是非终点对象（其内部还包含其他对象， 或叫组对象） ， 我们将对象比作节点 , 那么整体结构就是树形结构了
        - 所以组合模式的使用场景就是出现树形结构的地方。比如： 文件目录显示， 多及目录呈现等树形结构数据的操作
      - 装饰器模式
        - 在不必改变原类文件和使用继承的情况下， 动态地扩展一个对象的功能
          - iOS使用 Category 分类添加方法本身就是一种装饰器模式
        - 王者荣耀等类Dota游戏中， 英雄升级一样。每次英雄升级都会附加一个额外技能点学习技能, 那么只要对技能类动态添加或者删除方法(技能) , 英雄类(hero)直接使用新方法(技能)即可
      - 外观模式
        - 黄牛模式：完成一个类功能需要经历多个步骤(方法)才能完成时, 如果有一个抽象类(代理人)来完成这些步骤, 就好了(把任务交给代理人去完成复杂的流程, 让代理人去关联相关类), 懒得麻烦 , (比如代上牌)
          - 对外暴露方法（启动电脑， 关闭电脑） 。
          - 启动电脑（按一下电源键） ： 启动CPU、启动内存、启动硬盘
          - 关闭电脑（按一下电源键） ： 关闭硬盘、关闭内存、关闭CPU
      - 享元模式
        - 租赁(共享)模式：在有大量对象时， 有可能会造成内存溢出， 我们把其中共同的部分抽象出来， 如果有相同的业务请求， 直接返回在内存(比如内存, 线程池, 栈 等中寻找)中已有的对象， 避免重新创建
          - 比如共享单车, 不需要每次需要单车都去购买一辆, 直接租赁一辆即可(发现共享单车(已创建的对象), 并且闲置就可以使用, 自行车库就好比内存,线程池,栈等)
      - 代理模式
        - 专业(授权)机构模式：一些任务很难或者没有权限直接去处理时,就需要一个代理机构去(专业, 有授权等等)完成, 帮自己解决问题(比如第三方认证机构)
          - 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理
    - 行为型模式
      - 职责链模式
        - 场景1: 不知道接收者是谁, 将请求交给能处理的处理机构, 让机构去分配责任人处理, 并给到接收者
        - 避免请求发送者与接收者耦合在一起：[请求发送者A –> 请求处理者(责任人)B (接单人B1 –接单人B2 – 接单人B3 ...) –> 接收者C] , 接单人形成一条链, B1不能处理就询问B2,再询问B3...直到谁能处理就处理.
          - 优点：1. 请求者和接收者解耦, 2.请求者不需要知道处理者是怎么处理的, 只管发送 3. 处理链可独立扩展,增删处理逻辑
          - 缺点：1、不能保证请求一定被接收。 2、系统性能将受到一定影响， 而且在进行代码调试时不太方便， 可能会造成循环调用。 3、可能不容易观察运行时的特征， 有碍于除错。
      - 命令模式
        - 根据命令操作：在软件系统中， 行为请求者与行为实现者通常是一种紧耦合的关系， 但某些场合， 比如需要对行为进行记录、撤销或重做、事务等处理时， 这种无法抵御变化的紧耦合的设计就不太合适,比如事务处理(命令), git操作
          ```
          Stock abcStock = new Stock();
          BuyStock buyStockOrder = new BuyStock(abcStock);
          SellStock sellStockOrder = new SellStock(abcStock);
          Broker broker = new Broker();
          broker.takeOrder(buyStockOrder);//根据传入命令处理
          broker.takeOrder(sellStockOrder);//根据传入命令处理
          broker.placeOrders();
          ```
      - 解释器模式
        - 这种模式实现了一个表达式接口， 该接口解释一个特定的上下文。传入一个对象, 返回其解释或者算法结果,或者编译结果
          - 编译器、运算表达式计算, 树形型结果递归算法 等等
      - 迭代器模式
        - 遍历一个聚合对象：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。
          - 把在元素之间游走的责任交给迭代器， 而不是聚合对象, 比如链表hasNext, next
      - 中介者模式
        - 多个对象之间都通过中介者来处理事务, 对象之间解耦
          - 优点: 相互解耦, 独立升级
          - 缺点：中介者会变得膨胀
      - 备忘录模式
        - 临时存储,备份, 方便撤销或者回滚等等
          - 编辑过程存储部分编辑内容临时信息, 可以进行撤销等操作, 操作事务同样支持回滚等
      - 观察者模式
        - 通知功能：一对多依赖关系, 一个对象发生改变, 其他依赖立马收到通知
          - iOS的通知, KVO 等等
      - 状态模式
        - 状态切换：类的行为是基于它的状态改变的 , 扩展时遵循开闭原则 , 且状态的切换必须在运行过程中
          ```
          // 初始为开始状态
          context = context.setState(new StartState());
          // 切换为关闭状态
          context.close();
          // 切换为开始状态
          context.start();
          ```
      - 策略模式
        - 解决问题的多样性：封装很多方法类, 在运行过程中可以选择不同的方式继续解决问题 , 它们可相互替换
          - 旅行的出游方式， 选择骑自行车、坐汽车， 每一种旅行方式都是一个策略
      - 模板方法模式
        - 父类已经定义好了主要的操作流程和规则, 由其子类进行实现, 可以根据自身需求在主题架构上进行细节多样化
          - 商品房一栋楼的架构上下层都是一样的, 但是装修可以自己定义(建筑结构不能改, 装饰可以改)
      - 访问者模式
        - 对类增加 与 架构和其他固定结构都无关的一些功能, 我们增加一个访问类, 专门封装这些事务
          - 房屋的骨架和装修已经固定, 但是 来房间的打扫阿姨, 或者 客人都是访问者, 将他们与房屋(类)本身区分开
    - 其他
      - 软编码: 耦合度比较低
      - 硬编码: 耦合度比较高