日常开发与设计模式的那点事

前言

“很多程序员不知道怎么组织代码、怎么提升效率、怎么提高代码的可维护性、可重用性、可扩展性、灵活性，写出来的代码一团糟，但这样一团糟的代码居然能正常运行。”

这样的代码经历，你是否也似曾相识？

身边好多程序员都会有这样的一个经历，过个一年半载再去查看曾经写下的代码时，很吃惊的在想，这么糟糕的代码，真的是我以前写的吗？我居然能写出这么糟糕的代码！而对于还在维护的代码，此时，会萌生一种去重构的想法，或者会有一种更好的方式去实现。此时，你与代码的爱恨情仇已经开始了…

本片主要从六大基本原则说起，作为设计模式的引子，叙述六大基本原则和设计模式的关系，后续会一篇一个设计模式，详细介绍设计模式与日常开发。

基本的规范和约束

对于基本的规范和约束，我相信每个合格的团队都会有一套自己的玩意，一方面统一标准，增加可读性和可维护性，另一方面也方便离职后出现bug，后来者也能更快的去定位并解决问题。杂乱无章的代码实现一个大功能，对于后来者去维护，无疑会亲切的问候各路祖宗。一个好的编码习惯，属于一个合格程序员的自我修养，于己于人，百利而无一害。

对于开发中的规范和约束，首当其冲要说的就是命名，这五年多的工作，和形形色色的人合作过，记得最多的时候，我曾同时期开发和维护五个项目，业余时间，也曾和各路英雄好汉互相合作、互相学习和共同进步，在这个过程中，最让我觉得头疼的就是一些命名的不规范，不规范的样式有很多，各种奇怪的命名都有。我曾看到过这样一串命名，其中有两个功能，一个叫做专栏详情，一个叫做专栏留言，命名却是“ZhuanLaiDetalActivity”和“ZhuanLaiLiuYanActivity”，看的我很懵。

这样的命名，就问你怕不怕！对于这样的命名都怕了，那真没见过世面，这个至少还能看出个大概，之前看到过一些汉语拼音的缩写，比如动检证命名为djz，这个看起来才更懵了。

对于拼音命名，这里说一点不知道会不会被喷，遇到过一些朋友总是喜欢拼音命名，汉语拼音是中华民族推动汉文化的伟大创举，但是在编程的时候用拼音，真的觉得好low，即使再牛逼的技术作支撑，写出来的代码也像小学生的作品，这里没有看不起汉语拼音的意思，只是发表下内心的一些想法。

建议：大驼峰、小驼峰或者下划线命名都可以，如果没有一个统一的标准，可以参考《阿里巴巴Java开发手册》，对于刚入行的朋友，更应该从命名抓起，对于以后的成长有很大的帮助。

开发手册下载地址详见这里。

再者要强调的就是注释，很多人也许觉得注释是越多越好，之前也在书上看到过提倡多加注释，我觉得不然，有些时候注释给我们增加了很多负担和误解，在上次review的时候发现，一些同事copy我的一些代码的时候，其实是想做另一个功能，只是想把一些代码拷贝过去然后大修改（我不喜欢重复造轮子，对于相似的一些功能，最好做的灵活一点，提高代码的可重用性和灵活性），其实可以重用的地方很少，搞不明白为啥不自己写那么几行代码，这都不是事，让我很懵逼的是他们把我的备注和作者也拷贝过去了，当我进入那个类的时候，发现作者是我，去git查看历史提交，完全没我啥事，而且功能描述和此类完全不相关…

对于注释，还有一点要说的就是一些多余的注释，这个叫需要和命名相结合，好的命名规范，可以省略好多不必要注释，比如login、register，再加上登录、注册的注释，完全没必要，良好的习惯，可以给我们开发带来很多便捷，但有些喜欢textview1、textview2命名的，这些就算加了注释，等下文用到的时候，看了也是一群羊驼在奔跑，上下奔腾的那种。

```
for (int i = 0; i < j; i++) {
    // TODO
}
```

对于这样的代码，可能很多人会觉得很正常，也会有部分人会把责任归咎于谭浩强老师，是的，谭老的书中问题确实很多，但这不是写这种代码的理由，日常开发中，还有平时维护别人代码的同时，总会去调试for语句，难道不觉得这样的代码很糟糕，看得有点懵吗？就算加了注释，还是一坨一坨的。

因为每个团队有自己的规范和约束，大的公司，会有一套自己的规范，统一于各个团队，不同的语言也有不同的约束，如果日后有时间，会专门写一篇详细的约束与规范的blog赠送。对于这块，想写的东西真的好多好多，比如case后面的乱用，1、2、3总是让人费解，比如必要的常量替代变量，比如线程池取代线程，比如必要的地方使用单例，东西真的好多好多，不再比如下去了，今天就先说明两条重点，命名和注释。

建议：合理使用注释，对于新手在学习期间，在陌生的代码和不清晰的逻辑上，尽可能多一点注释，便于理解，对于老鸟，尽可能规范的命名，通过命名达到注释的效果，但是对于逻辑复杂或者操作状态太多的时候，必要的注释还是很重要的，减小维护成本。

一些应该熟知的编程思想

“一个程序员用在写程序上的时间大概占他的工作时间的10-20%，大部分的程序员每天大约能写出10-12行的能进入最终的产品的代码——不管他的技术水平有多高。好的程序员花去90%的时间在思考、研究和实验，来找出最优方案。差的程序员花去90%的时间在调试问题程序、盲目的修改程序，期望某种写法能可行。”

对于一个优秀的程序员来说，逻辑才是最重要的，他们愿意花更多的时间做思考，这样做的同时，就是更少bug会出现，甚至可以把bug率降到很低。我并不是很优秀的开发者，但这些年依然有这么一个习惯，对于复杂或者多样的功能，总会先理清思路，先列举出会有哪些操作，哪些地方是bug的雷区应该多注意，我也经常会和队友提起，一图胜千言，理清思路再下手，事半功倍。

不管业务逻辑是否复杂，上去就是干，发现有何不妥的再去修改，发现漏掉的再去添加，这样导致代码总是一坨一坨的堆在那里，经过多次的修改，已经面目全非，对于维护的人来说，更是苦不堪言，在此，笔者也建议读者朋友，不妨试一试先绘图在动手，把一个模块继续拆解成一个个接口，通过实现接口去实现这个模块，做到面向接口编程，这样可维护性会提升好多……

对于模块与模块之间的通信，不应该是类与类之间的关联，而是通过抽象去实现交互，抽象不应该依赖于细节，细节应该依赖于抽象，这话比较绕口，说白了，就是面向接口编程，而不是面向实现编程。

这样做的好处就是，将来你要把这个被调用的类换成一个别的实现类时，你就不用去把调用过它的类一个个改掉了，因为它们调的是接口，接口没变，在配置里把接口的实现类换成新的类，就全部都替换掉了。类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类引起波及。

建议：理清思路再下手，事半功倍。开发过程中，不妨先定义好接口，通过实现接口去完成模块的开发，尽可能的减小bug率，写出更加优质的代码。技术能力的提高，从代码上的体现主要在于“高内聚、低耦合”，因为这些思想衍生出许多开发模式，比如现在比较流行的MVC、MVP、MVVM等。

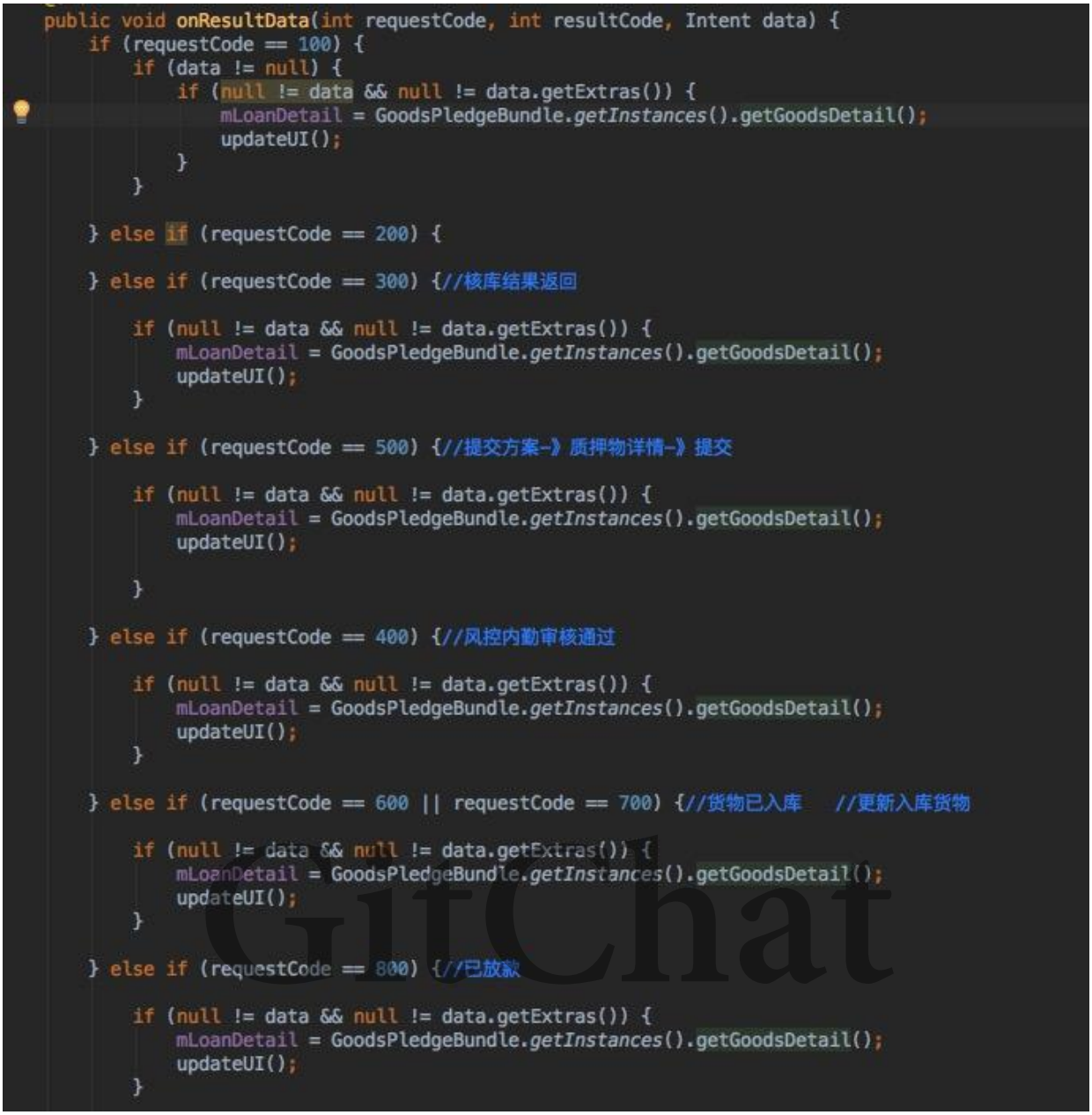
版本迭代与重构

“我们在做任何系统的时候，都不要指望系统一开始时需求确定，就再也不会变化，这是不现实也不科学的想法，而既然需求是一定会变化的，那么如何在面对需求的变化时，设计软件的可以相对容易修改，不至于说，新需求一来，就要把整个程序推倒重来。”

相信很多朋友都遇到过，原本一个很普通的需求，在经历过N次迭代和修改后，已经形成一个庞大的功能，随着版本的不断迭代，维护起来的成本也随着越来越大，这样就形成了一种恶性循环，重构代码即将登上历史舞台。

不可否认，从维护成本上看，重构确实是一个很不错的方案，重构的成本比原基础维护的成本更小，也更方便以后的维护。有些公司甚至在多次版本迭代后，直接把整个项目推到重构，这样的事情不仅仅发生在小公司，在一些大公司，也是会发生多次。

从技术上来说，重构复杂代码时要做三件事：理解旧代码、分解旧代码、构建新代码。而待重构的旧代码往往难以理解，尤其是在多次迭代且多人经手的模块；模块之间过度耦合导致牵一发而动全身，不易控制影响范围；旧代码不易测试导致无法保证新代码的正确性，尤其是在产品文档不全的时候。



```
public void onResultData(int requestCode, int resultCode, Intent data) {
    if (requestCode == 100) {
        if (data != null) {
            if (null != data && null != data.getExtras()) {
                mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
                updateUI();
            }
        }

    } else if (requestCode == 200) {

    } else if (requestCode == 300) {//核库结果返回

        if (null != data && null != data.getExtras()) {
            mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
            updateUI();
        }

    } else if (requestCode == 500) {//提交方案->质押物详情->提交

        if (null != data && null != data.getExtras()) {
            mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
            updateUI();

        }

    } else if (requestCode == 400) {//风控内勤审核通过

        if (null != data && null != data.getExtras()) {
            mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
            updateUI();
        }

    } else if (requestCode == 600 || requestCode == 700) {//货物已入库    //更新入库货物

        if (null != data && null != data.getExtras()) {
            mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
            updateUI();
        }

    } else if (requestCode == 800) {//已放款

        if (null != data && null != data.getExtras()) {
            mLoanDetail = GoodsPledgeBundle.getInstances().getGoodsDetail();
            updateUI();
        }
```

这是上次review时候发现的一段代码，先不说常量使用的不规范，这是经过一次次产品迭代后的结果，但这不是借口，造了这么多次轮子，真不应该。做为代码可重用性的反面教材，此处体现的淋漓尽致，如果有更多的状态，此处必然还是会重复多次…

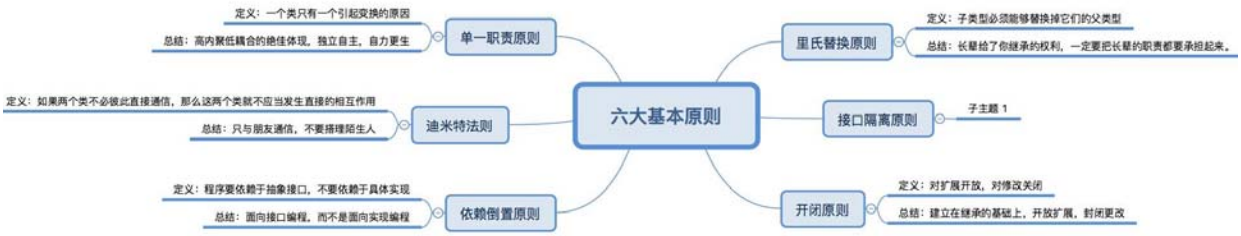
建议：重构，并不是万能的，重构后的代码，当再次经历后续几个版本修改后，代码又显的杂乱无章，那一坨坨代码总是在不断的重演。既然无法确定需求日后是否会修改，那我们只能通过提高代码质量来应对，以不变应万变，合理设计接口，每次更改需求时多思考，对于多次使用的代码进行封装提取，尽可能少的改动既有的逻辑。

设计模式的重要性

“会建筑设计的是建筑工程师，不会建筑设计的是搬砖的。”

前面已经说了很多，现在直接说一下设计模式的重要性，提到设计模式，就必须要提到六大基本原则和架构设计，提到架构设计，设计模式的重要性便可想而知。首先，六大基本原则还是有点争议的，我之前看到的书籍中，一般都是单一职责原则、迪米特法则、里氏替换原则、开闭原则、依赖倒置原则和接口隔离原则，但最近在一些帖子上看

到，有一种说法是没有接口隔离原则，而是合成/聚合复用原则，为了不影响之前的准备，合成/聚合复用原则会单独拿出来说一下。



六大基本原则，它是整个架构设计的灵魂，是架构设计的一种指导思想，而设计模式是架构设计的一种具体设计技巧，是架构设计的具体实践。

先从架构设计说起，对于架构设计，主要体现在抽象能力，抽象能力又依赖于架构者编码的阅历、功能的拆解和理解、逻辑的严密性。做架构设计应该尽可能且更全面的考虑问题，尽可能做好代码的包容性，海纳百川，有容乃大之势，这是架构设计者应该具备的基本条件。考虑的问题越周全，包容性越强，则工作难度越大，给自己造成的障碍也越多。合理的将这些细节问题进行抽象，并提出解决方案，抽象程度越高，解决方案越合理，这才是架构者的价值所在。从具体的需求，到代码实现，再到具体的产品。架构设计的目的无外乎系统的复用性、扩展性与稳定性，具体的东西是无法很好地体现这些特性的，只有抽象的事物才能最好的体现。

在架构设计的过程中，单一职责原则告诉我们应该更好的体现高内聚、低耦合，这个类是用来数据请求的，就别放一些解析json的方法，如果这个类是用来图片加载的，view的注解请隔离开，做到一个类只负责一个职责，只有一个引起变化的原因，如果一个类承担的职责越多，就等于把这些职责耦合到一起，会带来一些不必要的维护成本。从大的角度来说，MVP和MVC模式都是单一职责原则的体现，model-视图-控制相隔离，各司其职。

迪米特法则指导我们如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法时，可以通过第三者转发这个调用。类之间的耦合越弱，就越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。主要是强调了类之间的松耦合。

对于里氏替换原则，或许很多人没听过这个名词，但是在实际开发过程中却无时无刻不在使用，其实很简单，子类型必须能够替换掉它们的父类型。举个简单的例子，“List< String> list = new ArrayList<>();”，这么做的好处其实很简单，比如有一天ArrayList满足不了需求，需要改用LinkedList，只需要把ArrayList替换成LinkedList，而不是把全局的list对象都改一遍，提高了可维护性。

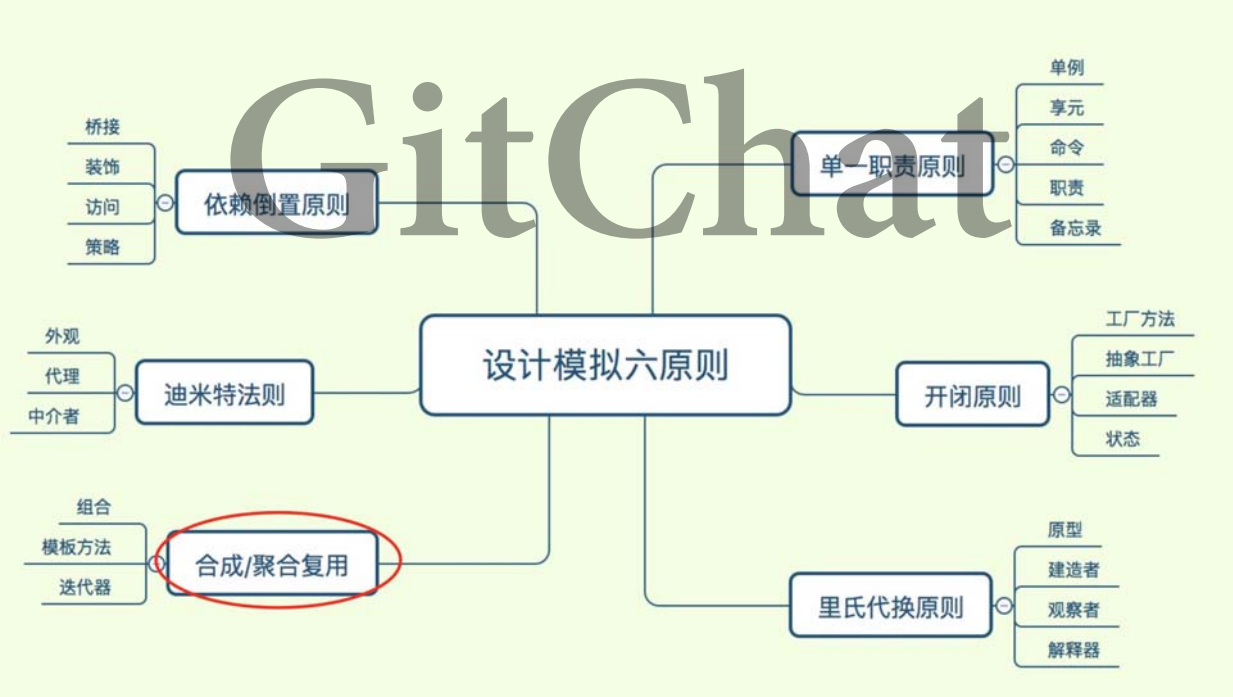
开闭原则是面向对象原则的核心，有两部分组成，对扩展开放，对修改关闭。软件需求总是会变化的，对软件设计人员来说，必须在不需要对原有系统进行修改的情况下，实现灵活的系统扩展。

对扩展开放，就是对抽象编程，而不是具体编程，因为抽象相对稳定，通过接口或者抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法。让类依赖于固定的抽象，所以，对修改是关闭的。这是建立在继承和多态的基础上，可以实现对抽象类的继承，通过覆盖其方法来扩展方法。

依赖倒置原则指的是依赖于抽象而不是依赖于具体实现，这一块在上述已经说过，其实就是面向接口编程而不是面向实现编程，这样做的好处就是解决耦合。一般情况下抽象的变化概率很小，让用户程序依赖于抽象，实现的细节也依赖于抽象。即使实现细节不断变动，只要抽象不变，客户程序就不需要变化。这大大降低了客户程序与实现细节的耦合度。

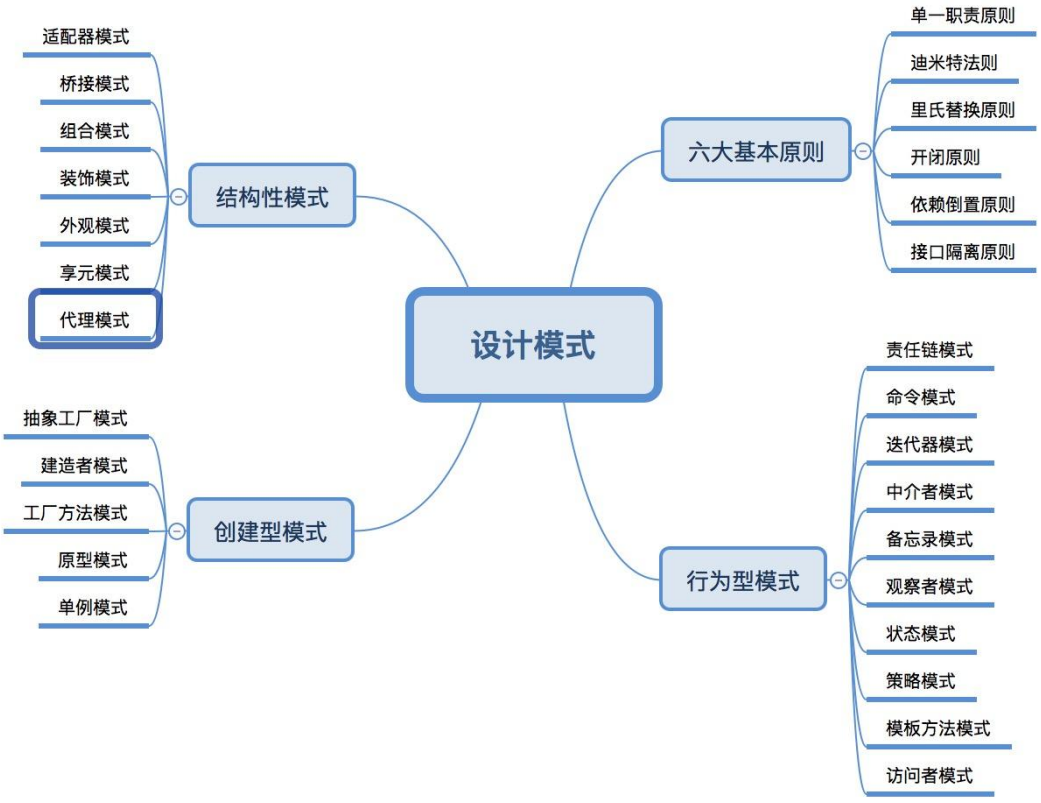
接口隔离原则认为，“使用多个专门的接口总比使用单一的接口要好”。一个模块应该依赖它需要的接口，需要什么接口就提供什么接口，把不需要的接口剔除掉，同时也应该遵循单一职责原则，这样避免臃肿的接口带来的污染，将没有关系的接口合并在一起，形成臃肿的大接口，就是对接口的一种污染。

接口的粒度也不能太小，太小会导致接口额数量剧增，对开发人员不友好；接口额粒度太大，灵活性降低，无法提供定制服务，给整体项目带来无法预估的风险，合理的设计接口，也是一门艺术。



对于这张图，一定存在很多的争议，因为很多设计模式都用到了多个基本原则，上图只是对设计模式的一个比较粗糙的总结，强调六大基本原则(含有合成/聚合复用原则)在设计模式中的具体体现，同时也说明了六大基本原则和23种设计模式是相辅相成的，六大基本原则作为设计模式的基石和模板，设计模式是六大基本原则运用的灵活体现。

合成/聚合复用原则这个是存在一定争议的，目前有的书中还是保留了合成/聚合复用原则去掉了接口隔离原则，合成/聚合复用原则指的是少用继承，多用合成关系来实现，合成和聚合都是对象建模关联关系的一种，聚合表示一种弱的拥有关系，整体由部分组成，部分可以脱离整体作为一个独立的个体存在，合成则是一种强的拥有关系，体现了严格的部分和整体的关系，部分和整体的生命周期一致，部分不能脱离整体。



总结：六大基本原则是面向对象思想的体现，单一职责原则与接口隔离原则体现了封装的思想，开闭原则体现了对象的封装与多态，而里氏替换原则是对对象继承的规范，至于依赖倒置原则，则是多态与抽象思想的体现。在充分理解面向对象的基础上，掌握基本的设计原则，并且能够在项目设计中灵活运用，就能够改善我们的代码质量和结构设计，尤其能够保证可重用性、可维护性、可扩展性和灵活性，这也是理解和掌握设计模式必备的知识。

补充

对于六大基本原则，这是我们开发都应该熟记于心并且灵活运用的，对于设计模式在日常开发中的运用，有一点还是要强调的，适合自己的才是最好的。如果此时一个模块是很轻量级，仅仅为了使用设计模式而用设计模式，这无疑也会显得不伦不类，使项目变的臃肿，同时也带来一些不必要的维护成本（虽然维护成本很低）。

最近开始整理资料，准备写设计模式专题，主要是MVP爬坑与迪米特法则、framework与开闭原则、单例与爬坑、换肤与观察者模式、加载列表与模板方法模式、构造函数与建造者的对比、多个第三方登录与命令模式，后续还会继续完善，敬请期待。