设计模式(道)

第一篇：

设计问题没有没有标准答案，不要轻信一家之言，主动思考，积极讨论。

第二篇：

设计模式对于复杂功能系统的设计开发具有重要作用；让读源码、学框架事半功倍；为你的职场发展做铺垫

第三篇：

好代码：

1. 可维护性：
2. 可读性：符合编码规范，命名达意，注释详尽，函数长短合适、模块划分清晰、高内聚低耦合
3. 可扩展性：
4. 灵活性
5. 简洁性：思从深而行从简，真正的高手能云淡风轻的用最简单的方法解决最复杂的问题
6. 可复用性
7. 可测试性：能否比较容易写出单元测试

第四篇：

抽象类：不能实例化，只能继承；抽象类可以包含属性和方法，不包含代码实现的方法叫做抽象方法；子类继承抽象类，必须实现抽象类中的所有抽象方法。

Java中的接口：接口不能包含属性；接口只能声明方法，方法不能包含代码实现；类实现接口的时候，必须实现接口中声明的所有方法。

他们之间的区别：抽象类是一种特殊的类，这种类不能被实例化对象，只能被子类继承。继承关系是一种is-a的关系；而接口表示一种has-a的关系，表示具有某些功能，对于接口有一个更加形象的叫法，那就是协议。

第五篇：

基于接口而非实现编程，是一种非常高效的提高代码质量的手段。

原因： 可以将接口和实现分离，封装不稳定的实现，暴露稳定的接口。越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能够在不破坏原有代码设计的情况下灵活应对。而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。

（音频数据管理中的接口其实可以改成 接口类的实现，这样就会更加灵活）

第六篇：

组合优于继承。

继承层次过深、过复杂会影响到代码的可维护性。

有一些特殊的场景必须使用继承，如果你不能改变一个函数的入参类型，而入参又非接口，为了支持多态，只能采用继承实现，重写这个类中的方法。

结合具体场景使用。

接口通常和组合关系配合使用

第七篇：

贫血模式 和 充血模式

什么是贫血模式？什么是充血模式？

贫血模式: 业务与数据不在一个类中即业务与数据相分离，破坏了面向对象的封装特性，是一种典型的面向过程的编程风格。

充血模型（Rich Domain Model）：正好相反，数据和对应的业务逻辑被封装到同一个类中。因此满足面向对象的封装特性。

领域驱动设计即DDD,主要用来指导如何解耦业务系统，划分业务模块，定义业务领域模型及其交互。属于充血模型

为什么说贫血模式的传统开发模式违反OOP?

基于贫血模式的传统开发模式既然违反OOP，那又为什么如此流行？

什么情况下我们应该考虑使用基于充血模型的DDD开发模式？ 复杂系统，业务划分建模

第八篇：

面向对象设计：

1. 划分职责进而识别出有那些类
2. 定义类属性和方法
3. 定义类与类之间的交互关系
4. 将类组装起来并提供执行入口

第九篇：

设计原则：SOLID KISS YAGNI DRY LOD

SOLID:

S: 单一职责原则

即一个类或者模块只负责完成一个职责（或者功能）；但是评价一个类职责是否单一，我们没有一个非常明确的可量化的标准。实际上在软件开发中，我们没必要过于未雨绸缪，过度设计，我们可以先写一个粗粒度的类，满足业务需求。随着业务的发展，如果粗粒度的类越来越庞大，代码越来越多，这个时候我们可以再做 划分重构。

几条参考判断原则：

1. 类中的代码行数、函数属性过多，会影响代码可读性和可维护性，此时就需要对类进行拆分
2. 类依赖的其他类过多
3. 私有方法过多，考虑是否将私有方法独立到新的类中设置为public方法
4. 比较难给类起一个合适的名字
5. 类中的大量方法都是集中操作类中的某几个属性

高内聚低耦合，不必拆的过分细

O 开闭原则

对扩展开放，对修改关闭的原则

最重要的原则，我们要时刻具备扩展意识、抽象意识、封装意识，在写代码的时候，多花点时间思考一下如何更易于以后的扩展。很多设计原则设计模式都是以提高代码的扩展性为最终目的，最常用来提高扩展性的方法有：多态、依赖注入、基于接口而非实现的编程，及大部分设计模式

L 里氏替换原则

子类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏

多态与里氏替换原则教相似，但他们关注的角度是不一样的，多态是面向对象的一大特性及语法，而里氏替换是一种原则，指导继承关系中的子类该如何设计

I 接口隔离原则

客户端不应该被强迫依赖它不需要的接口。其中的客户端，可以理解为接口的调用者或使用者。

与单一职责原则的区别：单一职责原则针对的是模块、类、接口的设计。接口隔离原则更侧重于接口的设计，通过调用者判断接口职责是否单一

D 依赖反转原则(依赖倒置)

**控制反转IOC** : 一般用来指导框架层面的设计，没使用框架之前，程序员自己控制程序的执行，使用框架之后，框架来控制整个程序的执行

**依赖注入DI**: 是一种具体的编码技巧，不通过new的方式在类内部创建依赖类的对象，而是将依赖的类的对象在外部创建好后，通过构造函数、函数参数等方式传递(或注入)

**依赖注入框架**：通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间的依赖关系，就可以实现框架自动创建对象、管理对象生命周期、依赖注入等 原本需要程序员来做的事情

**依赖反转**：和控制反转有点相似，用来指导框架层面的设计，高层模块不依赖低层模块，他们 共同依赖同一个抽象。抽象不要依赖具体实现细节，具体细节依赖抽象

第十篇：

迪米特法则

不该有直接直接依赖关系的类之间，不要有依赖；有依赖关系的类之间尽量只依赖必要的接口。减少类之间的耦合，让类越独立越好，每个类都应该少了解系统其他部分。

第十一篇：

重构

对于项目而言重构可以保持代码质量处于一个可控状态；建立持续重构的意识，把重构作为开发必不可少的一部分，融入到日常开发中。

大规模重构需要有组织、有计划的进行，分阶段小步快跑，时刻让代码处于可运行的状态。

单元测试的编写：

针对代码设计各种测试用例，以覆盖各种输入、异常、边界情况，并将其翻译成代码。

建立正确的认知：

1. 编写单元测试尽管繁琐但不太耗时
2. 我们可以稍微放低对单元测试代码质量的要求
3. 覆盖率作为衡量单元测试的唯一标准是不合理的
4. 单元测试不要依赖被测代码的具体实现逻辑
5. 单元测试框架无法测试，主要是因为代码的可测试性不好

依赖注入的方式编写单元测试

第十一篇：创建型

单例模式

定义： 一个类只允许创建一个对象或实例，那这个类就叫单例类，这种设计模式就叫单例模式。 避免资源访问冲突，业务概念上的全局唯一类。

饿汉式：提前创建好单例，不支持延迟加载

懒汉式：调用的时候再创建单例支持延迟加载需要做好线程安全 通过双重检测的方式即先判断实例是否存在，如果不存在再加锁

单例对于继承和多态特性支持不好，并且违背面向接口而非实现编程的原则。

单例会隐藏类之间的依赖关系，不方便代码阅读

单例对于代码的可测试性不友好，主要由于全局对象导致不同测试用例都可修改

单例模式的替代方案：

1. 为了保证全局唯一，除了使用单例，还可以使用静态方法来实现，可以解决单例隐藏类之间依赖关系的问题。不过，对于单例存在的其他问题，比如对 OOP 特性、扩展性、可测性不友好等问题，还是无法解决。
2. 进程间的单例模式：参考计费批价进程的单例模式(一个通道只能启动一次)，或者采用外部文件加锁的方式

工厂模式

简单工厂、工厂方法原理比较简单也较常用

抽象工厂较复杂，实际中相对不常用

工厂模式使用时的参考标准：

1. 封装变化：创建逻辑有可能变化，封装成工厂后，创建逻辑对调用者透明
2. 代码复用：创建代码抽离到独立的工厂类之后可以复用
3. 隔离复杂性：封装复杂的创建逻辑，调用者无需了解如何创建对象

建造者模式

使用场景：

1. 我们把类的必填属性放到构造函数中，强制创建对象的时候就设置。如果必填的属性有很多，把这些必填属性都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填属性通过 set() 方法设置，那校验这些必填属性是否已经填写的逻辑就无处安放了。
2. 如果类的属性之间有一定的依赖关系或者约束条件，我们继续使用构造函数配合 set() 方法的设计思路，那这些依赖关系或约束条件的校验逻辑就无处安放了。
3. 如果我们希望创建不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值，要实现这个功能，我们就不能在类中暴露 set() 方法。构造函数配合 set() 方法来设置属性值的方式就不适用了。

原型模式：

对于熟悉javascript语言的前端程序员来说，原型模式是一种比较常用的开发模式。

定义：如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作**原型设计模式**

第十二篇：结构型

结构型模式主要总结了一些类或对象组合在一起的经典结构，这些经典结构可以解决特定应用场景的问题。包括：代理模式、桥接模式、装饰器模式、适配器模式、门面模式、组合模式、享元模式

代理模式

在不改变原始类(或叫被代理类)代码的情况下，通过引入代理类给原始类附加功能。一般情况下，我们让代理类和原始类实现同样的接口。但是，如果原始类并没有定义接口，并且原始类代码并不是我们开发维护的。在这种情况下，我们可以通过让代理类继承原始类的方法来实现代理模式。

桥接模式

定义：Decouple an abstraction from its implementation so that the two can vary independently

将抽象和实现解耦，让它们可以独立变化。

组合由于继承

装饰器模式

与代理模式很像但又不同，代理模式中，代理类附加的是和原始类无关的功能，而在装饰器模式中，装饰器类附加的是跟原始类相关的增强功能。

适配器模式：

较常用，将不兼容的接口转换为可兼容的接口

两种实现方式：类适配器和对象适配器，其中类适配器使用继承关系实现，对象适配器使用组合关系实现

适用场景：

1. 封装有缺陷的接口设计：假设依赖的外部系统在设计上有缺陷，引入之后会影响到自身代码的可测试下，为了隔离设计上的缺陷，我们希望对外部系统提供的接口进行二次封装，抽象出更好的接口设计。
2. 统一多个类的接口设计
3. 替换依赖的外部系统
4. 兼容老版本接口
5. 适配不同格式的数据

适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。

门面模式(外观模式Façade Design Pattern)：

门面模式为子系统提供一组统一的接口，定义一组高层接口让子系统更易用。

我们知道，类、模块、系统之间的“通信”，一般都是通过接口调用来完成的。接口设计的好坏，直接影响到类、模块、系统是否好用。所以，我们要多花点心思在接口设计上。我经常说，**完成接口设计，就相当于完成了一半的开发任务**。只要接口设计得好，那代码就差不到哪里去。

组合模式：

主要用来处理树形结构数据

将一组对象组织（Compose）成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（在很多设计模式书籍中，“客户端”代指代码的使用者。）可以统一单个对象和组合对象的处理逻辑

组合模式，将一组对象组织成树形结构，将单个对象和组合对象都看做树中的节点，以统一处理逻辑，并且它利用树形结构的特点，递归地处理每个子树，依次简化代码实现。使用组合模式的前提在于，你的业务场景必须能够表示成树形结构。所以，组合模式的应用场景也比较局限，它并不是一种很常用的设计模式。(批价树)

第十二篇：行为型模式

23中经典的设计模式分为三类：创建型、结构型、行为型

创建型设计模式主要解决“对象创建的问题”；

结构型设计模式主要解决“类或对象组合或组装”的问题

行为型设计模式主要解决“类或对象之间交互”的问题

行为型模式比较多分别是：观察者模式、模板模式、策略模式、职责链模式、状态模式、迭代器模式、访问者模式、备忘录模式、命令模式、解释器模式

观察者模式（发布订阅模式）：

使用较多,流程一般都是先注册然后再轮询notify各个注册者

应用场景： 发布订阅， RSS Feeds 等

模板模式

较常用，

主要解决复用和扩展两个问题。

定义：模板方法模式在一个方法中定义一个算法骨架，并将某些步骤延迟到子类中实现。模板方法模式让子类在不改变算法整体结构的情况下，重新定义算法中特定步骤。

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

例子 poco notification

复用指的是，所有的子类可以复用父类中提供的模板方法的代码。扩展指的是，框架通过模板模式提供功能扩展点，让框架用户可以在不修改框架源码的情况下，基于扩展点定制化框架的功能

策略模式

较常用，

常见应用场景：避免冗长的if-else 或switch分支判断，还可以像模板模式那样，提供框架的扩展点。

定义：定义一族算法类，将每个算法分别封装起来，让他们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it。

Map映射类，条件与处理建立映射，通过工厂模式创建

设计原则和思想其实比设计模式更加普适和重要，掌握了代码的设计原则和思想，我们甚至可以自己创造出来新的设计模式。

职责链模式：

较常用，

定义：将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，直到某个链上的某个接收对象能处理它为止

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

应用场景举例：

对于支持 UGC（User Generated Content，用户生成内容）的应用（比如论坛）来说，用户生成的内容（比如，在论坛中发表的帖子）可能会包含一些敏感词（比如涉黄、广告、反动等词汇）。针对这个应用场景，我们就可以利用职责链模式来过滤这些敏感词。

计费系统的分拣，上述的敏感词过滤也是！

实际上，更具体点来说，职责链模式最常用来开发框架的过滤器和拦截器。如：Servlet Filter、Spring Interceptor 这两个 Java 开发中常用的组件

状态模式：

状态模式一般用来实现状态机，而状态机常用在游戏、工作流引擎等系统开发中。

有限状态机：Finite State Machine 缩写为FSM,简称状态机。状态机有3个部分：状态(state)、事件(event)、动作(action)；其中事件也成为转移条件（Transition Condition）,事件触发状态的转移及动作的执行。不过动作不是必须的，也可能只转移状态，不执行动作。

状态机实现方式一：分支逻辑法

If-else switch-case分支判断 不推荐

状态机实现方式二：查表法

将状态与事件，事件与动作的对应关系存放到两个二维数组中(transiontable actionTable)，构成两张二维表

状态机实现方式三：状态机模式

在查表法的代码实现中，事件触发的动作只是简单的积分加减，所以，我们用一个 int 类型的二维数组 actionTable 就能表示，二维数组中的值表示积分的加减值。但是，如果要执行的动作并非这么简单，而是一系列复杂的逻辑操作（比如加减积分、写数据库，还有可能发送消息通知等等），我们就没法用如此简单的二维数组来表示了。

状态模式通过**将事件触发的状态转移和动作执行，拆分到不同的状态类中，来避免分支判断逻辑**。对于状态并不多、状态转移也比较简单，但事件触发执行的动作包含的业务逻辑可能比较复杂的状态机来说，我们首选这种实现方式

迭代器模式

也叫游标模式

对于复杂数据结构的遍历，利用迭代器可以将复杂的遍历操作放到迭代器类中

设计”快照”功能的迭代器

访问者模式

允许一个或者多个操作应用到一组对象上，解耦操作和对象本身。

增加中间类visitor 函数重载

静态绑定 运行时动态绑定

备忘录模式

在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态

全量备份 增量备份

Captures and externalizes an object’s internal state so that it can be restored later, all without violating encapsulation.

命令模式：