软件系统设计 具体设计模式

By 张洪胤 未经本人同意请勿转发

模式名称	英文名/别名	模式分类	设计模式简述	设计模式角色	相关设计原则	设计模式适用场景	设计模式实例	设计模式优点	设计模式缺点	模式扩展
策略模式	*Strategy Pattern* *Policy Pattern*	对象行为型	**定义：**策略模式定义了一系列算法，将每个算法封装在一起，并使它们可替换，策略使得算法独立于使用该算法的客户端而变化(变化在客户使用时才会出现，必须将细节暴露给用户)			1. 许多相关类仅在**行为**上有所不同：使用配置类配置 2. 需要**算法的不同变体**：定义不同具体的算法 3. 避免**暴露复杂的、特定于算法的数据结构** 4. 一个类定义多种行为并显示为多个**条件语句**	1. Java 的加密方法 2. Java 的时间显示算法	1. 消除条件语句 2. 提供多种实现方式供客户选择	1. 客户在选择合适的策略之前必须先了解策略的不同 2. 策略和上下文之间通信开销 3. 对象数量增加	
简单工厂模式	*Simple Factory Pattern* *Static Factory Pattern*	类创建型	**定义：**简单工厂模式定义一个专门的类负责创建其他类的实例(这些类通常都有相同的父类)，可以根据参数的不同返回不同类的实例	1. **工厂角色：**提供**静态**工厂方法，接受参数(配置文件)传入 **2. 抽象产品角色(父类)** **3. 具体产品角色**	**满足单一职责原则：**将对象创建和业务逻辑使用分离 **违背开闭原则：**对开闭原则支持不好	1. 工厂类负责创建的对象比较少(不会导致判断逻辑太复杂) 2. 客户端只需要知道传入工厂的参数，对于如何创建对象不关心	1. 权限管理 2. Java 的 DateFormat 3. Java 的加密技术(对称非对称加密)	1. 实现了责任切割，提供专门的工厂类用于创建对象 2. 客户端无需知道所处构建的具体产品类类名，而只需要知道参数，减少记忆量 3. 通过引入配置文件，可以在不修改客户端代码的情况下更换和增加新的具体产品类，提高灵活性	**1. 工厂集中了的职责过重，增加产品违背开闭原则(需要修改判断逻辑代码)** 2. 系统扩展困难，添加新产品就需要修改工厂逻辑 3. 不利于系统的扩展和维护 4. 工厂角色无法形成基于集成的等级结构。 5. 增加系统中类的个数，增加复杂性和理解难度	简单工厂模式的简化：工厂类由抽象产品角色对象扮演(适用于产品本身只提供 1-2 个方法时)
工厂方法模式	*Factory Method Pattern* 虚拟构造 *Virtual Constructor* *多态工厂* *Polymorphic Factory*	类创建型	生成**一类产品** **定义：**将产品类的实例化操作推迟到工厂子类中完成。 **变化的部分：**产品的服务 **不变的部分：**对象的使用	1. **工厂父类：**定义创建产品对象的公共接口 2. **工厂子类：**定义生成具体的产品对象(推迟实例化) **3. 抽象产品对象** **4. 具体产品对象**	**符合开闭原则：**允许系统在不修改工厂角色的情况下引进新产品	1. 一个类不知道它所需的对象的类(只需要知道对应工厂) 2. 一个类通过其子类来指定创建哪个对象(结合多态性和里氏代换原则) 3. 将创建对象的任务委托给多个工厂子类中的某一个，客户端使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定(可以用配置文件)	1. 日志记录器(文件记录、数据库记录)	1. 用户只需要关心所需产品对应的工厂，无须关注创建细节，甚至无须指导具体产品的类名。 2. 工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节完全封装在具体工厂内部。 3. 系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改具体工厂和具体产品，只需要添加一个具体工厂和具体产品。	1. 编写新的具体产品类时，还需要提供与之对应的具体工厂类，导致类个数成对增加，增加系统的复杂度和编译开销 2. 抽象层引入增加了系统的抽象性和理解难度，且实现中会使用到 DOM 和反射等技术，实现难度大。	**模式退化：**抽象工厂和具体工厂合并，且创建对象的工厂方法被设计为静态方法时退化成具体简单工厂模式
抽象工厂模式	*Abstract Factory Pattern* *Kit 模式*	对象创建型	**定义：**生成**多个位于不同产品等级结构中属于不同类型的具体产品。** 提供一个创建一系列相关或相互依赖对象的接口，而无须指定他们具体的类。	**1. 抽象工厂** **2. 具体工厂** **3. 抽象产品** 4. **具体产品**	**违反开闭原则：**需要编写修改克隆方法。	1. 一个系统不应当依赖于产品实例如何被创建、组合和表达的细节(适用于所有工厂) 2. 系统中有多于一个的产品族，而每次只使用其中某一产品族 3. 属于同一个产品族的产品将在一起使用(**系统设计约束**) 4. 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使得客户端不依赖于具体实现。	1. 电器工厂(生产电视和空调等多个产品族) 2. 数据库操作工厂(生成 Connection 和 Statement)	1. 隔离了具体类的生成，改变具体工厂的实例就可以某种程度上改变整个软件系统的行为，实现高内聚低耦合。 2. 当产品族多个对象被设计成一起工作时，能保证客户端只使用同一个产品族中的对象。 3. 增加新的具体工厂和产品族很方便，无须修改已有系统	1. 添加新的产品对象时，难以扩展抽象工厂来生成新种类的产品(开闭原则的倾斜性，增加新的工厂和产品族容易，增加新的产品等级结构麻烦)	**开闭原则的倾斜性：**增加新的产品族只需要增加新的具体工厂，而增加新的产品等级结构，则需要修改所有工厂对象和抽象工厂。 **模式退化：**只存在一个产品等级结构时，抽象工厂退化为工厂方法模式。 **产品等级结构：**产品等级即产品的集成结构。比如，抽象类是电视机，子类是海尔电视机等。 **产品族：**由同一个工厂生成的，位于不同产品等级结构中的一组产品。比如，海尔电视机在电视机的产品族中
建造者模式	*Builder Pattern*	对象创建型	生成**一个组装好的完整产品。** **定义：**将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示(一步一步创建一个复杂的对象，用户不需要知道构建的细节，只需要指定复杂对象的类型和内容即可) **变化的部分：**产品的细节 **不变的部分：**建造的过程	1. **抽象建造者：**为创建一个产品对象的各个部件指定抽象接口。 2. **具体建造者：**实现抽象创建早接口，实现每个部件的构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象。 3. **指挥者：**隔离客户与生成的过程；负责控制产品的生成流程(针对抽象建造者编程，客户只需要知道具体建造者的类型) 4. **产品角色：**被构建复杂对象		1. 产品对象具有复杂的内部结构(包含多个成员属性) 2. 产品对象属性相互依赖，需要执行生成顺序 3. 产品创建过程独立于创建对象的类。(引入指挥者类，对创建过程进行封装) 4. 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。	1. KFC 套餐 2. Java Mail 3. 地图和任务	1. 客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的对象。 2. 每个建造者相对独立，用户使用不同的具体建造者即可得到不同的产品对象。 3. 可以跟家精细地控制产品的创建过程(步骤分解) 4. 增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者编程，系统扩展方便，符合开闭原则	1. 要求创建的产品具有较多的相同点，其组成部分相似，使用范围受限。 2. 如果产品内部变化复杂，则需要很多的具体建造者，导致系统庞大	**省略抽象建造者角色：**系统内只需要一个具体建造者。 **省略指挥家角色：**如果抽象建造者已经被省略，那么可以让具体建造者同时扮演指挥家和建造者。 **与抽象工厂模式的区别：**更侧重于创建一个完整的对象，并且可以通过指挥者类来完成创建。可以将抽象工厂模式比作汽车配件生产工厂，创建者模式是汽车组装工厂。
原型模式	*Prototype Pattern*	对象创建型	**定义：**用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。(无须知道任何创建的细节) 基本原理是通过将一个原型对象传给要发动创建的对象，要发动创建的对象通过请求原型对象拷贝自己来实现创建过程。	1. **抽象原型类：**具有克隆自己的方法的接口。 2. **具体原型类：**实现具体得到克隆方法，返回自己的克隆对象。 3. **客户类：**直接实例化或通过工厂方法创建一个对象后，就可以通过克隆方法进行复制。	**违反开闭原则：**需要编写修改克隆方法。	1. 创建新对象的成本较大。使用复制可以降低成本。 2. 如果系统需要**保存对象状态**时，如果对象状态变化小或占用内存较少，可以使用**原型模式配合备忘录模式**。如果对象状态变化较大，则使用**状态模式**。 3. 需要通过避免使用分层次的工厂类来创建分层次的对象，且类的实例只有一个或较少状态时，使用复制比使用构造更好。	1. Java Object 的 clone 方法(需要实现接口 Cloneable)复制一份对象并返回(满足克隆对象和原对象不是同一个对象，类型相同) 2. 邮件复制(浅克隆，不复制附件) 3. 软件中的 Ctrl+C、Ctrl+V 4. Strut2 中为保证线程安全性，创建 Action，避免其中定义的变量加锁进行同步。 5. Spring 中原型模式创造新的 Bean 后对其进行修改(不影响原实例)	1. 原型模式可以快速创建很多相同或相似的对象，简化对象的创建过程(对于复杂对象)，提供了简化的创建结构 2. 可以动态增加或减少产品类 3. 可以使用深克隆保存对象的状态	1. 每个类都需要提供克隆方法，必须修改类的源代码，违反开闭原则 2. 实现深克隆时需要编写较为复杂的代码	**原型模式分类：**深克隆(递归拷贝成员变量)和浅克隆 **带原型管理器的原型模式**(PrototypeManager)

状态模式	*State Pattern* 状态对象 *Objects of States*	对象行为型	**定义：**允许一个对象在其内部状态改变时改变它的行为。 状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。 **变化的部分：**增加新的状态，面向新的状态的行为	1. **环境类(Context)：**拥有状态的对象(可以充当状态管理器完成状态切换)，存放尽可能多的数据，减少状态中的信息，需要针对抽象状态类(持有一个抽象状态类的实例)进行编程。 2. **抽象状态类：**专门表示对象的状态的抽象类(或接口) 3. **具体状态类：**实现了不同状态的行为，包括各状态之间的转换。会被设置到环境类中。	**违反开闭原则：**为追求对用户的透明(避免用户设置状态变更)。	1. 对象行为依赖于它的状态(属性)并且可以根据状态改变而改变行为。 2. 代码中包含大量与对象状态相关的条件语句，导致代码的低可维护性、灵活性和高耦合，不能方便地删除和增加状态。	1. 论坛不同等级用户 2. 政府 OA 系统中批文状态 3. RPG 游戏中游戏角色控制(游戏活动、游戏角色等级)	1. 封装了转换规则 2. 枚举了可能的状态(需要先确认状态的种类) 3. 方便增加状态(状态的行为集中) 4. 允许状态转换逻辑与状态对象合成一体(而不是复杂条件语句) 5. 允许多个环境对象共享一个状态对象，减少对象的个数	1. 增加类和对象的个数 2. 结构和实现复杂，实现不当容易导致程序结构和代码的混乱 3. 对开闭原则的支持不好，增加状态类需要修改状态切换代码，修改行为也需要修改对应状态类的代码	**共享模式：**多个状态需要共享同一个状态，那么将状态对象定义为环境的静态成员变量。 **简单状态模式：**所有的状态彼此独立无需转换，可以在客户端实例化状态类后将状态对象设置到环境类(符合开闭原则) **可切换状态的状态模式：**大部分状态模式的情况，切换时调用 Context 的 setState()方法进行状态切换。
命令模式	*Command Pattern* 动作模式 *Action Pattern* 事务模式 *Transaction Pattern*	对象行为型	**本质：**对命令进行封装，将发出命令的责任和执行命令的责任分离开(使得请求方不知道请求的处理细节) **关键：**引入抽象命令接口，发送者针对抽象命令接口编程，具体命令与接收者关联。 **定义：**将一个请求封装为一个对象，从而使得可用不同的请求对客户进行参数化，从而对请求排队或记录日志，以及支持可撤销的操作。	1. **抽象命令类：**声明执行请求的 execute()方法，通过这些方法调用请求接收者的相关操作。 2. **具体命令类：**实现抽象命令类中声明的方法，对应具体的接收者，将接收者对象的行为进行绑定。 3. **调用者(Invoker)：**请求的发送者，通过命令对象执行请求。 4. **接收者(Receiver)：**执行与请求相关的操作，具体实现对请求的业务处理。 **5. 客户类(Client)**		1. 系统需要将请求调用者和请求接收者解耦(调用者和接收者不直接交互) 2. 系统需要在不同时间指定请求、将请求排队和执行请求 3. 系统需要支持命令的撤销和重做操作。 4. 系统需要将一组操作组合在一起(支持宏命令)	1. 电视机遥控器：打开、关闭电视和切换频道 2. 功能键设置 3. Java 实现 AWT/Swing GUI 的委派事件模型(DEM)：界面组件是请求发送者，AWT 提供事件监听器接口和时间适配器是抽象命令接口，用户可以实现具体命令类来完成相应操作。 4. Unix 平台下的 Shell 编程提供宏命令	1. 降低系统的耦合性 2. 新的命令可以更容易的加入系统 3. 可以更容易设计命令队列和宏命令(组合命令) 4. 可以方便的实现对请求的 undo 和 redo	1. 导致系统有过多的命令类(每一个命令对应一个命令类)	**宏命令：**命令模式和组合模式联用，调用宏命令时，宏命令将递归调用每个成员命令的 execute 方法
观察者模式	*Observer Pattern* 发布-订阅 *Publish/Subscribe* 模型-视图 *Model/View* 源-监听器 *Source/Listener* 从属者 *Dependents*	对象行为型	**定义：**定义对象间的**一种一对多依赖关系**，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。 **关键对象：**观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，目标的状态发生改变，所有的观察者都将得到通知。	1. **目标(Subject)：**又称主题，被观察的对象 2. **具体目标：**发出通知时不需要知道谁是他的观察者。 3. **观察者(Observer)：**根据 4. **具体观察者：**维护一个纸箱具体目标对象的引用，存储具体观察者的有关状态。		1. 抽象模型具有两方面，一个方面依赖于另一个方面，需要将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。 2. 一个对象的改变将导致其他一个或多个独享也发生变化，而不知道有多少对象变化(降低对象间的耦合) 3. 一个对象必须通知其他对象，而并不知道这些对象是谁。 4. 需要在系统中创建一个触发链(A 触发 B，B 触发 C 等等)	1. 猫、狗与老鼠：老鼠和狗观察猫 2. 自定义登录控件：事件处理模型(Java Swing/AWT 控件) 3. Java 实现 AWT/Swing GUI 的委派事件模型(DEM)：事件发布者是事件源，订阅者是事件监听器，可以通过事件对象来传递必要的信息(此时事件监听对象又可以被称为事件处理对象) 4. Java 的 SAX2 和 Servlet 技术的事件处理机制都基于 DEM 5. 电子商务网站发送给多个用户商品打折信息 6. 团队战斗游戏全体广播	1. 实现表示层和数据逻辑层的分离，定义了稳定的消息传递机制，抽象更新接口。 2. 在观察目标和观察者之间建立一个抽象的耦合。 3. 支持广播通信。 4. 符合开闭原则要求。	1. 如果直接和间接观察者过多，则将所有的观察者都通知到需要花费很多的时间。 2. 如果观察者和观察目标之间存在循环依赖则容易导致系统崩溃。 3. 观察者模式没有相应的机制让观察者了解被观察目标对象是如何变化的。	**Java 对观察者模式支持：**Java 提供了 Observable 类以及 Observer 接口。 **MVC 模式：**包含模型(Model，观察目标)、视图(View，观察者)和控制器(Controller，中介者)三个角色 TODO 推模型、拉模型 **与策略模式不同：**观察者模式需要由子类完成状态切换，而策略模式是直接设置上下文来完成策略选择。
中介者模式	*Mediator Pattern* 调停者模式	对象行为型	**定义：**用一个中介对象封装一系列的对象交互，中介者使对象不需要显式地互相引用，从而使其耦合松散，并且可以独立改变他们之间的交互。	1. **抽象中介者(Mediator)：**定义用户各通知对象之间通信的接口。 2. **具体中介者：**协调各个同事对象来实现协作行为，了解并维护它的各个同事对象的引用。 3. **抽象同事类(Colleague)：**定义了各同时的公共方法。 4. **具体同事类：**通过具体中介者类间接完成和其他同时类的通信。	**符合迪米特法则：**中介者对象减少了有关对象所引用的其他对象的数量减少到最少。	1. 对象之间存在复杂的引用关系，产生的依赖关系结构混乱且难以理解。 2. 一个对象应用了很多其他对象并直接通信，导致难以复用该对象。 3. 想通过一个中间类来封装多个类的行为，而又不想生成太多子类。	1. 虚拟聊天室：不同会员 2. 事件驱动类软件中多见(GUI 程序的设计) 3. MVC 模式中的 Controller 作为中介者完成 View 和 Model 的交互 4. Struts 中，Action 作为 JSP 页面与业务对象之间的中介者。	1. 简化了对象之间的交互 2. 将各个同事对象解耦 3. 减少了子类的生成 4. 简化了各同事类的设计和实现	1. 在具体中介者类中包含了同时的交互细节，可能导致具体中介者类非常复杂难以维护。	**中介者职责：** 1. 中转作用(结构性)：通过中介者提供的中转作用，避免各个同事对象显式的引用其他同事。 2. 协调作用(行为性)：中介者可以进一步对同事的关系进行封装，保证同事们一致的和中介者交互(由中介者根据封装在自身内部的协调逻辑来进行处理，从而将同事成员之间的关系行为进行分离和封装) **GUI 设计：**方便地应用在 GUI 开发中，将交互的组件作为具体同事类，将之间的引用和控制关系交给中介者。
模板方法模式	*Template Method Pattern*	类行为型	**定义：**定义了一个操作中算法的股价，将一些步骤延迟到子类中，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。 **方法**包括模板方法和基本方法，基本方法又可以分为抽象方法、具体方法和钩子方法。	**1. 抽象类：**定义一系列基本操作(具体或抽象)，实现一个模板方法定义算法骨架。 **2. 具体子类：**实现父类定义的抽象基本操作。 **3. 模板方法：**定义在抽象类中的、把所有操作方法组合在一起形成一个总算法或一个总行为的方法。 **4. 基本方法：**实现算法各个步骤的方法。包括抽象方法、具体方法和钩子方法	**符合开闭原则：**通过对子类扩展来增加新的行为。 **符合单一职责原则：**类内聚性提高	1. 一次性定义算法的不变部分，将可变的行为留给子类来实现 2. 各子类中公共的额行为应当被抽取出来并集中到一个公共父类以避免代码重复 3. 对复杂的算法进行切割，将算法中固定不变的部分设计为末模板方法或具体方法。而可变的部分交给子类。 4. 控制子类的扩展	1. 银行业务办理流程：取号、排队、办理业务、评分。 2. 数据库操作模板：连接、打开、使用、管理等步骤 3. 广泛使用与框架设计确保父类控制处理流程的逻辑：Spring、Struts、Junit	1. 在一个类中抽象地定义算法，而由它的子类实现细节的处理。 2. 是一种代码复用的基本技术。 3. 导致一种反向的控制结构，通过父类调用子类操作，使用对子类的扩展增加新的行为，符合开闭原则。	1. 每个不同的实现都需要定义一个子类，导致类数量增加，系统庞大、设计更抽象。	**继承的讨论：**鼓励恰当的使用集成，将可复用的一般性行为移动到父类中，给开发带来便捷。 **好莱坞原则**：不要给我们打电话，我们会给你打电话。具体体现为子类不需要调用父类，而通过父类来调用子类，将某些步骤的实现写在子类中，由父类来控制整个流程。 **钩子方法：**使子类可以控制父类的行为，钩子方法可能是空方法或默认实现，复杂一些可能是对其他方法进行约束(返回布尔值判断是否执行)

适配器模式	*Adapter Pattern* 包装器 *Wrapper*	类结构型/对象结构型	**定义：**将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的类可以一起工作。	1. **目标抽象类：**定义客户要用的特定领域的接口(必须) 2. **适配器类(Adapter)：**可以调用另一个接口，将适配者和抽象目标类进行适配，是适配器模式的和兴。 3. **适配者类(Adaptee)：**被适配的橘色，定义了一个已经存在需要适配的接口。 4. **客户类(Client)：**对目标抽象类编程，调用目标抽象类中定义的方法。	**符合开闭原则：**可以在不修改代码的基础上添加新的适配器。	1. 系统需要使用现有的类，但是接口不符合系统的需要 2. 系统想要一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括未来可能引进的类一起工作。	1. 仿生机器人：适配行为 2. 加密适配器：重用第三方加密方法 3. JDBC 给出了客户端的通用的抽象接口，而数据引擎的 JDBC 驱动软件都是介于 JDBC 接口和数据库引擎接口之间的适配器软件 4. Java 的 InputStreamAdapter 类，用来包装 ImageInputStream 及其子类	1. 将目标类和适配者类解耦：引入适配器类来重用现有的适配者类。 2. 增加类的透明性和复用性：将具体实现封装在适配者类中，对客户端透明，提高复用性。 3. 灵活性和扩展性都非常好：通过配置文件可以方便的更换适配器，或者不修改代码的基础上添加适配器，符合开闭原则。 4. (类适配器)适配器是适配者的子类，因此可以在适配器类中替换一些适配者的方法，使得适配器的灵活性更强。 5. (对象适配器)对象适配器可以将多个不同的适配者适配到同一个目标接口。	1. (类适配器)不支持多继承的语言，一次最多只能适配一个适配者类，且目标抽象类必须为抽象类(不可以是具体类)，具有一定的局限性。 2. (对象适配器)相比于类适配器模式，置换适配者类的方法就不容易(需要用适配者的子类置换对应方法后，将适配者的子类作为真正的适配者)	**默认适配器模式(缺省适配器模式，单接口适配器模式)：**不需要全部事先接口提供的方法时，可以设计一个抽象类实现接口，为接口中每个方法提供默认实现(空方法)，适用于接口不想使用其所有的方法时。 **双向适配器模式：**适配器同时包含目标类和适配者类的引用，可以调用目标类和适配者类中的方法。
组合模式	*Composite Pattern* 整体-部分模式 *Part-Whole Pattern*	对象结构型	**定义：**组合数个对象形成树形结构以表示“整体-部分”的结构层次。组合模式对单个对象(即叶子对象)和组合队形(即容器对象)的使用具有一致性 **关键：**定义了抽象构件类(代表叶子或容器)	1. **抽象构件(Component)：**是叶子构件和容器构件的对象声明接口，包含所有子类共有行为的声明和实现。 2. **叶子节点(Leaf)：**叶子节点没有子节点。 3. **容器(Composite)：**和抽象构件之间建立聚合关联关系(使用集合用于存储子节点)，容器对象可以包含叶子或容器，形成树形结构。 4. **客户类(Client)：**针对抽象构建进行编程		1. 需要表示一个对象整体或部分层次，在这种层次结构中，希望通过一种方式忽略整体和部分的差异来一致对待 2. 客户端希望忽略不同对象层次的变化，针对抽象构建编程，无须关心对象层次结构的细节。 3. 对象的结构是动态且复杂程度不同，但客户希望一致地处理它们。	1. 水果盘：遍历盘子吃 2. 操作系统的目录结构：文件树结构 3. XML 文档解析 4. Java 的 AWT/Swing 的组件之间	1. 清晰地定义分层次的复杂对象，表示对象的全部层次，易于添加新构件。 2. 客户端调用简单，客户端可以一致的使用组合结构或其中单个对象 3. 定义包含叶子对象和容器对象的类层次结构，叶子可以被组合为容器，容器也可以组合成更复杂的容器，递归组成复杂的树结构。 4. 更容易在组合体内加入对象构建，避免加入新对象构件而修改代码。	1. 设计变得抽象，业务逻辑比较复杂时实现组合模式有很大挑战(不是所有的方法都与叶子对象子类有关联) 2. 增加新构建时可能产生一些问题，很难对容器中的构建类型进行限制。	**更复杂的组合模式：**更深的层次结构 **透明组合模式：**叶子节点继承实现了全部方法(TODO) **安全组合模式：**违反里氏代换原则(删除叶子节点不可能使用的方法)
桥接模式	*Bridger Pattern* 柄体模式 *Handle and Body* 接口模式 *Interface Pattern*	对象结构型	**定义：**将抽象部分与它的实现部分分离，使他们都可以独立地变化。 **关键：**在软件系统的抽象化和实现化之间使用关联关系而不是继承关系 **核心：**将自身的机制沿着多个维度进行变化(与装饰者模式不同)	1. **抽象类(Abstraction)：**定义一个实现类接口类型的对象并维护该对象 2. **扩充抽象类(RefinedAbstraction)：**扩充由抽象类定义的接口，实现了在抽象类中定义的抽象业务方法 3. **实现类接口(Implementor)**仅提供基本操作，而抽象类定义的接口可能会做更多更复杂的操作 4. **具体实现类：**实现了实现类接口并具体实现它，在不同的具体实现类中提供基本操作的不同实现，在程序运行时，具体实现类对象将替换其父类对象，提供给客户端具体业务操作方法。		1. 系统需要在构建的抽象化角色和具体化角色之间增加更多的灵活性，避免两个层次建立静态的继承关系(桥接模式建立关联关系) 2. 抽象化角色和实现化角色可以以集成的方式独立扩展而互不影响 3. 一个类存在两个独立变化的维度，并且两个维度都需要扩展，且两个角色都需要独立的管理 4. 不希望使用集成或因为多层次集成导致系统类的个数急剧增加的系统	1. 模拟毛笔：抽象颜色、型号 2. 跨平台视频播放器：不同操作系统、不同视频格式 3. Java 的 JVM 的平台无关性 4. Java 的 AWT 中 Peer 架构 5. Java 的 JDBC 的驱动程序(抽象角色)，数据库(实现角色)：驱动程序将一个特定类型的数据库与一个 Java 应用程序绑定在一起，实现抽象角色与实现角色的动态耦合。	1. 分离抽象接口及其实现部分。 2. 桥接模式是比多继承方案更好的解决方案(避免违背单一职责原则、类的个数庞大等弊端) 3. 提高了系统的可扩充性，在两个变化维度中任意扩展一个维度不修改原系统。 4. 实现细节对客户透明，可以对客户隐藏实现细节。	1. 增加系统的理解与设计难度(聚合关联关系在抽象层，要求开发者针对抽象设计与编程) 2. 桥接模式要求正确识别出系统中两个变化的维度，使用范围具有一定的局限性。	**抽象化：**将对象的共同特性抽取出来形成类的过程。 **实现化：**针对抽象化给出具体的实现。 **脱耦：**将抽象化和实现化之间的耦合解脱开，或者将他们之间的强关联转换为弱关联，继承关系装换为关联关系。 **适配器模式与桥接模式的联用：**桥接模式用于系统的初步设计，初步设计完后如果发现某些类无法协作，则采用适配器模式(尤其设计大量第三方接口时)
装饰者模式	*Decorator Pattern* 包装器 *Wrapper* 油漆工模式	对象结构型	**定义：**动态地给一个对象增加一些额外的职责。就增加对象功能而言，装饰模式比生成子类实现更为灵活。 **核心：**增加新的职责(与桥接模式不同)	1. **抽象构件(Component)：**定义了对象的接口，可以为对象动态的增加职责。 2. **具体构件：**实现抽象构建中声明的方法，装饰器给他增加额外的职责。 3. **抽象装饰(Decorator)：**抽象构建类的子类，用于给具体构建增加职责 4. **具体装饰：**负责向。构建添加新的职责。	**符合开闭原则：**增加新的具体构件类和具体装饰类时仅需要组合。	1. 不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。 2. 需要动态地给一个对象增加功能，也要动态地取消这些功能。 3. 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能继承的情况分两类：其一是大系统存在大量独立的扩展，为支持每一种组合将产生大量子类，呈现爆炸式增长；第二是类定义为不能继承(final)	1. 变形金刚 2. 多重加密系统 3. Java 的 Swing 包中可以通过装饰模式动态给一些构建增加新的行为或改善其外观显示。 4. Java IO： a. 抽象构建类 InputStream b. 具体构建类 FileInputStream、ByteArrayInputStream c. 抽象装饰类 FileInputStream d. 具体装饰类 BufferedInputStream、DataInputStream 等	1. 可以提供比继承更多的灵活性。 2. 可以通过动态的方式来扩展一个对象的功能，通过配置文件在运行时选择不同装饰器。 3. 通过使用不同的具体装饰类和这些装饰类的排列组合，可以创造出很多不同行为的组合(多个装饰者装饰一个对象) 4. 具体构件类与具体装饰类可以独立变化，增加新的具体构件类和具体装饰类使用时仅需要组合，符合“开闭原则”	1. 产生很多的小对象，他们之间的区别仅仅是连接方式不同，同时还产生很多具体装饰类，这些具体装饰类和小对象会增加系统的复杂度，加大学习与理解的难度。 2. 比继承更灵活机动的特性，意味着装饰模式比继承更容易出错且难以排查(需要逐级排查)，较为繁琐	**关联与继承：**关联不破坏类的封装性，而继承是耦合度较大的静态关系，无法运行时动态扩展。关联不减少编码量，但松耦合让系统更容易维护，不过要比继承创建更多的对象 **装饰模式简化：**装饰类接口和被装饰类接口保持一致；不要将太多的逻辑和状态放在具体构建类中，通过装饰类进行扩展；如果只有一个具体构件类没有抽象构建类，那么抽象装饰类可以作为具体构建类的直接子类。 **透明装饰模式(多重加密系统)：**客户端完全针对抽象编程，客户端程序全声明为抽象构建类型。 **半透明装饰模式(变形金刚)：**允许客户端声明具体装饰者类型的对象，调用其在具体装饰者中新增的方法。
外观模式	*Facade Pattern* 门面模式	对象结构型	**定义：**外观与一个子系统的通信必须通过一个统一的外观对象进行，为子系统的一组接口提供一个统一的界面，外观模式定义了一个高层接口，这个接口系统更容易使用。 **目的：**降低系统的复杂程	1. **外观对象(Façade)：**客户端直接调用的对象，在外观角色中可以知道相关的子系统功能和责任，它将所有从客户端的请求委托给对应的子系统处理。 2. **子系统角色(SubSystem)**	**符合单一职责原则：**引入外观对象，为子系统的访问提供一个简单而统一的接口，让子系统的通信和互相依赖关系达到最小。 **符合迪米特法则：**	1. 当要为一个复杂子系统提供一个简单接口时可以使用外观模式。 2. 客户程序与多个子系统之间存在很大的依赖性(外观类可以将子系统与客户以及其他子系统解耦) 3. 在层次化结构中，可以用外观	1. 电源总开关：管理灯、电视等等 2. 文件加密：使用加密外观类完成文件读取、加密和保存三个类的功能 3. 外观模式应用在 JDBC(Update 等操作) 4. Session 外观模式	1. 对客户屏蔽子系统组件，减少客户处理的对象数目使得子系统使用起来更加容易。 2. 实现了子系统与客户之间的松耦合关系，使得子系统组件变化不影响客户类 3. 降低大型软件系统中的编译依赖性，并简化了系统在不同平台之间	1. 不能很好地限制客户使用子系统类(过多限制降低可变性和灵活性) 2. 在不引入抽象外观类的情况下，增加新的子系统需要修改外观类和客户端代码，违反开闭原则。	**多个外观类：**通常只需要一个外观类(单例类)，节约系统资源，但系统中可以设计多个外观类负责和不同的子系统交互。 **抽象外观类：**解决增加新的子系统或移除子系统时违反开闭原则的问题。

			度 **不要通过外观类为子系统增加新的行为。**		引入外观对象降低原系统的复杂性，降低客户类与子系统类的耦合度。 **违反开闭原则：**没有引入抽象外观类时新增子系统需要修改外观类和客户端源代码。	模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。		的移植过程(编译一个子系统不需要编译所有其他的子系统) 4. 只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。		
享元模式	*FlyWeight Pattern* 轻量级模式	对象结构型	**定义：**运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。 **内部状态：**可以共享的相同内容，是享元对象内部且不会随环境改变而改变的状态，可共享。 **外部状态：**需要外部环境来设置的不能共享的内容，随环境改变而改变，不可共享。 **享元工厂：**通常享元模式会和工厂模式共用，维护享元池(Flyweight Pool)用来存储具有相同内部状态的享元对象。 **享元对象(细粒度对象)：**一般设计为较小的对象，所包含的内部状态较少	1. **抽象享元类(FlyWeight)：**声明接口来接受并作用于外部状态。 2. **具体享元类：**实现抽象享元接口，其实例为享元对象。 3. **非共享享元类：**不可以被共享的抽象享元类对象 4. **享元工厂类：**提供一个用于存放享元对象的享元池。用户访问时限查询享元池，如果享元池中存在则直接返回，否则创建一个存入享元池后返回。		1. 系统有大量相同或相似对象，导致大量内存开销 2. 对象的大部分状态都可以外部化，传入对象 3. 需要维护存储享元对象的享元池，比较消耗资源，所以需要在多次重复使用享元对象时才使用。	1. 共享网络设备(无外部状态) 2. 共享网络设备(有外部状态)：分配外部端口 3. 编辑器(图片对象) 4. JDK 中的 String	1. 极大地减少内存中对象的数量，使得系统在内存中只保存一份。 2. 外部状态相对独立，且不会影响内部状态，从而使得享元对象可以在不同环境中被共享。	1. 使得系统更加复杂，需要分离内部状态和外部状态，复杂化程序逻辑。 2. 为共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。	**单纯享元模式：**所有的享元对象都可以共享(无非共享具体享元类) **符合享元模式：**将单纯享元使用组合模式加以组合，可以形成复合享元对象，这样的复合享元对象本身不可分享，但是分解后可以分享。 **与工厂模式：**通常享元工厂提供一个静态工厂方法返回享元对象(简单工厂) **与单例模式：**一个系统中通常只有一个享元工厂，因此可以使用单例模式 **与组合模式：**构成符合享元模式，统一设置外部状态
代理模式	*Proxy Pattern* *Surrogate Pattern*	对象结构型	**定义：**给某个对象提供一个代理，并由代理对象控制对原对象的引用	1. **抽象主题角色(Subject)：**声明了真实主题和代理主题的共同接口。 2. **代理主题对象(Proxy)：**包含了对真实主题的引用 3. **真实主题对象 RealSubject** 实现真实的业务操作。	**符合迪米特法则**	1. 远程代理：为一个位于不同的地址空间对象提供一个本地代理对象，远程代理被称为大使(Ambassador) 2. 虚拟代理：先创建消耗小的对象来代理一个创建消耗大的对象，真实对象只有被使用到时才被创建 3. Copy-on-Write 代理：虚拟代理的一种，将复制和克隆推迟到客户端真正需要时才执行(尤其是深克隆) 4. 保护(Protect of Access)代理：控制一个对象的访问，可以给不同的用户提供不同级别的使用权限 5. 缓冲(Cache)代理：为某一个操作结果提供临时存储空间，多个客户端共享结果。 6. 防火墙(Firewall)代理：保护目标不被恶意访问 7. 智能引用代理(Smart Reference)：当对象被引用时，提供一些额外的操作(比如记录被访问次数)	1. 论坛权限控制代理(代理保护) 2. 数学运算代理(远程调用) 3. Java RMI 远程方法调用 4. EJB、Web Service 等分布式技术	1. 协调调用者和被调用者，一定程度上降低了系统的耦合性 2. 远程代理使得客户端可以访问在远程机器上的对象 3. 虚拟代理可以使用一个小对象代理一个大对象，减少修通资源消耗，优化系统，提高运行速度 4. 保护代理可以控制对真实对象的使用权限	1. 有些代理模式将导致请求的处理速度变慢 2. 实现代理模式需要额外的工作，有些代理模式的实现非常复杂。	
单例模式	*Singleton Pattern*	类创建型	**定义：**某个类只能有一个实例，提供一个全局的访问点。	1.		1.	1.	1. 节约系统资源，提高系统效率	1. 单例的职责过重，难以扩展	
迭代器模式	*Iterator Pattern*	对象行为型	**定义：**一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。	1. 迭代器角色(Iterator)：定义遍历元素所需的方法，一般包含 next()、hasNext()和 remove()三个方法 2. 具体迭代器角色：定义迭代器接口中定义的方法，完成集合的迭代。 3. 容器对象(Aggregate)：一般是一个接口，提供 Iterator()方法，如 Java 的 Collection、List、Set 接口等。 4. 具体容器对象：抽象容器的具体实现类，如 ArrayList		1.	1. JDK 的 ArrayList、LinkList 等等	1. 简化遍历方式：方便用户对基本类型和对象类型进行遍历。 2. 提供多种遍历方式：正序遍历或倒序遍历 3. 封装性良好：用户不必关心遍历算法。	1. 简单遍历的时候，使用迭代器的方式进行遍历比较繁琐。	
责任链	*Chain of*	对象行为型	**定义：**将请求的发送者和	1. **抽象处理者角色(Handle)：**		1. 系统的审批需要通过多个对象	1.	1. 降低了请求的发送者和接收者之	1. 在找到正确的处理对象之前，	

模式	*Responsibility Pattern*		接收者解耦，使得多个接收者都有处理这个请求的机会。	定义处理请求的接口，这个接口通常由接口或抽象类实现。 2. **具体处理者角色：**具体处理者接受请求后可以选择自己处理，也可以传递给下一个处理者，因此每个处理者都保存了下一个处理者的引用。		才能完成处理的情况，例如请假系统。 2. 代码中存在多个 if-else 语句时		间的耦合。 2. 将多个条件判断分散到各个处理类中，使得代码更加清晰，责任更加明确。	所有的条件判断都要执行一遍，可能导致性能问题。 2. 可能导致某个请求不被处理。	
备忘录模式	*Memento Pattern*	对象行为型	**定义：**在不破坏封装的前提下，保持对象内部的状态。 **核心：**保存了对象的部分信息（快照），这与原型模式不同	1. **发起人(Originator)**:负责创建备忘录(Memento)，用以记录当前时刻自身的内部状态，并可使用备忘录恢复内部状态。发起人可以需要决定备忘录存储自己的哪些状态。 2. **备忘录(Memento)**：负责存储发起人对象的内部状态，并防止发起人之外的其他对象来访问备忘录，包含两个接口，管理者(Caretaker)只能看到备忘录的窄接口，可以将备忘录传递给其他对象，而发起人可以看到备忘录的宽接口，允许它访问返回到先前状态所需的全部数据。 3. **管理者(Caretaker)**：负责管理备忘录，不能访问和操作备忘录的内容。		1. 状态保存和回滚	1. 游戏保存状态 2. 保存副本	1. 可以维护和记录比较复杂的属性历史记录。	1.	
解释器模式	*Interpreter Pattern*	对象行为型	**定义：**给定一个语言，定义它的文法的一种表示，并确定一个解释器来解释该语言中的句子。	1. **抽象表达式(Expression)：**声明所有的具体表达式都需要实现的抽象接口，这个接口包含 interpret()方法，被称为解释操作。 2. **终结符表达式(Terminal Expression)：**实现了抽象表达式所要求的接口，文法中每一个终结符都有一个具体中介表达式与之对应。 3. **非终结符表达式(Non-Terminal Expression)：**文法中的每一条规则都需要有一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字。 4. **环境(Context)：**一般存放文法中各个终结符锁对英国的具体值，比如 R1=100		1.	1.	1. 可扩展性好、灵活。 2. 增加了细腻的解释表达式的方式 3. 容易实现文法	1. 虽然可扩展性强，但是如果语法规则的数目过大则会导致该模式非常复杂。 2. 执行效率比较低，可利用场景比较少 3. 对于复杂的文法比较难以维护	
访问者模式	*Visitor Pattern*	对象行为型	**定义：**表示一个作用于某个对象结构中的各个元素的操作，再不改变元素的前提下定义以下操作：对象结构中存储多种不同类型的对象信息、对同一对象结构中元素的操作方式并不唯一，可能要提供多种不同的处理方式、可能要增加新的处理方式、为每种对象提供一个解决方案。	1. **抽象访问者(Visitor)：**接口或抽象类，定义了对于每一个元素的访问行为，参数是可访问的元素，其方法数原则上与元素的个数一致。 2. **具体访问者：**给出每一个元素类访问时所产生的具体行为。 3. **抽象元素：**元素接口或抽象类，定义了一个接受访问者的方法，意义是每一个元素都要可以被访问者访问。 4. **具体元素：**具体的元素类，提供了接受访问方法的具体实现，通常情况下这个方法是使用访问者提供的访问该元素类的方法。 5. **对象接口：**定义对象结构，内部管理了元素集合，并且可以迭代这些元素供访问者访问。		1. 适用于元素的类稳定不会频繁修改，不然会导致 Visitor 反复修改。	1.	1.	1.	