为什么需要设计模式?

世界越来越复杂，市场和需求变化越来越快。我们希望我们设计出来的软件更有生命力，能够自演化，更容易维护，扩展，复用，有更高的代码质量。

1、提高维护性

需求变化 -> 需要修改已有的功能

只需要最小限度的改动，只改动其中相关的一个子功能，不影响系统其他的部分，其他子功能不用动，甚至不用公开。这样影响最小。

例子：内存坏了，修内存。

活字印刷 vs 刻板印刷

2、提高扩展性

需求变化 -> 需要增加一个新的功能

只需要增加类和方法，不需要修改已有的类和方法。

注：设计模式是从代码设计的角度考虑需求变更。另外，还有架构设计的角度，项目管理的角度（比如敏捷实践）

不好的维护性或扩展性，在面对需求变更时，只能重做。已有的功能，不能再被其他系统调用。

3、提高复用性

已有功能，可以最大限度地重复使用。

4、提高灵活性

需求不同 -> 根据需求选择不同的子功能。

如何实现?

高内聚，低耦合

面向对象三大特性：封装，继承，多态。

解耦

比如前端和后端解耦，页面逻辑和业务逻辑解耦。

设计原则

《设计模式》，GOF DP

《设计模式解析》，Alan Shalloway，James R. Trott DPE

《敏捷软件开发：原则、模式与实践》Robert C.Martin ASD

《重构-改善既有代码的设计》Martin Fowler

《重构与模式》Joshua Kerievsky

《Java 与模式》阎宏

选择某种模式，用抽象来封装变化。用继承来复用不变。

单一职责原则 SRP

就一个类而言，应该仅有一个引起它变化的原因。ASD

如果一个类承担的职责太多，就等于把这些职责耦合在一起。一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受意想不到的破话。ASD

软件设计真正要做的工作之一，就是发现职责并把这些职责相互分离。ASD

开放-封闭原则 OCP

软件实体（类、模块、函数等）应该可以扩展，但是不可修改。ASD

面对（未来）需求的变化，对于扩展是开放的 **Open for extension**，对于更改是封闭的 **Closed for modification**。避免面对新的需求、新的市场变化，需要推倒重来。尽量保持稳定的架构和版本，在此基础上迭代以满足新的需求。

对于目标的修改是关闭的，对于达成目标的方法的扩展是开放的。

绝对的封闭不可能，设计人员必须对他设计的模块应该对哪种变化封闭做出选择。必须预计最有可能发生的变化种类，然后构造抽象来隔离哪些变化。

先知先觉的人善于预见变化，特别是很可能发生的，可能频繁发生的变化，并提前做出应对。对于后知后觉的人，有时候预计变化是困难的，可以等到变化发生时立即采取（重构）行动，创建抽象来隔离以后发生的同类变化。ASD

知道可能发生变化的需求的等待时间越长，意味着已经完成的开发量越大，历史包袱越大，重构的成本越高。

最小知识原则 LKR

迪米特法则 LoD：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。如果一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。DP
在类的结构设计上，每一个类都应当尽量降低成员的访问权限。
根本思想是强调类之间的松耦合。
类之间的耦合越弱，越有利于复用。一个处于弱耦合的类被修改，不会对有关系的类造成波及。

依赖倒转原则

高层模块不应该直接依赖低层模块。两个都应该依赖抽象。

抽象不应该依赖细节。细节应该依赖抽象。

高层和低层约定好接口，低层实现了接口，高层通过接口来调用低层。

接口只负责抽象，具体逻辑由具体实现类完成。

里氏代换原则 LSP：子类型必须能够替换掉它们的父类型。ASD

正是因为里氏代换原则，才使得开发封闭成为可能。才使得子类型的替换成为可能。

只有当**子类**可以替换掉**父类**，软件单位的功能不受影响，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为。

正是由于这种可替换性（父类可以被子类替换，子类之间可以互相替换），才使得使用父类的模块在无需修改的情况下，就可以扩展。

依赖 VS 依赖倒转

依赖：高层模块和**低层模块**之间直接调用，直接依赖，甚至某个品牌的高层模块只能使用某个品牌的低层模块，高度耦合。

比如把访问数据库的代码写成函数库，然后调用这些函数库。

当数据库变了，访问数据库的函数库也要变，调用函数库的高层代码也要变。高层代码和函数库都无法复用。

依赖倒转：高层和低层约定好接口，高层调用接口，低层实现接口。不同数据库访问的函数库都实现同一套接口，不同的函数库就可以随便切换，不会影响高层的代码和复用。

合成/聚合复用原则 CARP

优先使用合成/聚合，而不是类继承。DP

优先使用合成/聚合将有助于保持每个类被封装，并被集中在单个任务上，这样类和类继承层次会保持较小规模，并且不大可能增长为不可控制的庞然大物。DP

类之间的关系

关联：B 类是 A 类的一个（普通）成员，或者是 A 类一个方法中的成员。

聚合：B 类是 A 类的一个数组成员，特殊的关联关系。

组合：B 类是 A 类的一个成员，而且是在 A 类的构造函数中实例化 B 类，A 和 B 类的生命周期相同，特殊的关联关系。

依赖：B 类是 A 类的一个成员方法的参数，特殊的关联关系。

耦合的强弱程度

关联 < 接口实现 < 聚合 < 组合 < 继承

关联 association：类 A 知道类 B，类 B 是类 A 的一个成员变量。

聚合 aggregation：类 A **弱拥有**类 B，类 B 是类 A 的一个数组成员变量。

组合 compostion：类 A **强拥有**类 B，类 B 是类 A 的一个成员变量，而且出现在构造函数，即生命周期相同。体现了严格的部分和整体的关系。

依赖 dependency：类 A 依赖类 B，类 B 出现在类 A 成员方法的参数。

对象的继承关系在编译时就定义好了，所以无法在运行时改变从父类继承的实现。子类的实现与其父类有非常紧密的依赖关系，以至于父类实现中的任何变化必然导致子类发生变化。当你需要复用子类时，如果继承下来的实现不适合解决新的问题，则父类必须重写或者被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。DP

继承是一种强耦合。

设计模式的分类

创建型模式

单例模式	保证一个类仅有一个实例	Spring 的注解@Component 默认是单例

工厂模式	工厂模式 = 产品接口 + 具体产品类 + 工厂接口 + 具体工厂类	具体产品类集成相同接口，且通过 spring 管理; 在具体工厂类，通过@Autowired 自动注入具体产品类; 在具体工厂类, 增加 map<产品类名, 具体产品类>;
抽象工厂模式		
建造者模式	建造者模式 = **指挥者类 Director**（不变的装配配件的顺序）+ **建造者接口 Builder**（不变的装配配件的方法）+ **具体建造者类**（变化的具体的配件的实现）	
原型模式	Clone	

结构型模式

适配器模式	需要把类 A 或接口 a 统一入现有的架构 B 中。 **翻译**	
装饰模式	**区分核心职责和装饰职责**。系统 A 完成的核心职责，而 Bn 负责满足特定情况下才会执行的 n 个特殊行为，每一个 Bn 只负责其独立的特殊职责。**由调用者根据需要有选择地、按顺序地使用装饰职责层层增强核心职责。** **女生和衣服**	
桥接模式	设计类结构时，发现实现方式有 2 种（或更多）	
组合模式	树结构	

享元模式	共享对象	可以创建一个@Clone **通过注解，为需要克隆的类自动生成并实现克隆方法 clone().**
代理模式	真实对象和代理对象	
外观模式	聚合接口 **男生和衣服**	

行为型模式

解释器模式		
中介者模式	**用一个中介对象来封装一系列的对象交互**。	
访问者模式	**把数据结构和作用在结构之上的操作解耦，从而使操作集合可以相对自由地演化。**	
策略模式	**工厂模式实例化出来的对象的总数是有限的，而策略模式的算法对象可能是无限的。** **工厂模式返回的是具体的实现对象，而策略模式返回的是（拥有具体实现对象的）上下文对象。**	
备份模式	**游戏存档** 在对象之外保存对象的状态 被备份的对象 + 备份类 + 管理备份的类	**可以创建一个注解@Memento，** 通过注解，为需要备份的类自动生成备份方法 createMemento(). 默认备份该类所有属性，可以在注解中选择需要备份的属性。或者在需要备份的属性在增加注解**@MementoField**

迭代器模式		
观察者模式	**观察者模式的关键对象是主题类 Subject 和观察者 Observer。** 一个 Subject 可以有任意数目的依赖它的 Observer，一旦 Subject 的状态发生了改变，所有的 Observer 都会得到通知。 主题类不需要知道谁是它的观察者。任何一个观察者不需要知道其他观察者的存在。 **使用场景：一个对象的改变需要同时改变其他对象的时候。**	**Spring ApplicationEvent**
模板方法模式	**模板方法模式把不变的行为抽象到超类，从而去除了子类中的重复代码，实现了代码复用。** **模板类 + 模板方法 + 虚方法**	
命令模式	命令模式把**请求一个操作的对象 Invoker** 和知道怎么**执行一个操作的对象 Receiver** 分离（解耦）。 **操作请求类 + 操作执行类 + 命令接口 + 具体命令类**	
状态模式	**使用场景：if-else 过长。** 主要解决当控制一个对象状态转换的条件表达式过于复杂的情况。**把状态的判断逻辑转移到表示不同状态的一系列的类当中**，可以把复杂的判断逻辑简化。	

	好处是将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。DP 意思就是如果有 10 个 if-else，就分成 10 个独立的**状态类 State**，每个类只处理自己的逻辑，非自己的职责转交给下一个状态类处理。 在**上下文类 Context** 中，存放状态切换的条件和当前状态类，并负责调用当前状态类的方法，该方法在状态类实现的接口中定义。	
职责链模式	**将这些对象连成一条链条**，并沿着这条链传递该请求，直到有一个对象处理它为止。	

简单工厂模式和策略模式 **Strategy** 的区别

工厂模式实例化出来的对象的总数是有限的，而策略模式的算法对象可能是无限的。

工厂模式是根据场景的不同，比如运算类型加减乘除，请求头传递过来的客户端类型 web/mobile/API 等，实例化出对应的对象并返回。这些对象实现了相同的接口及接口方法。

策略模式是在策略上下文类 Context 聚合了策略类，根据场景的不同，实例化出相应的策略类对象，返回策略上下文对象。调用者通过策略上下文对象调用不同的策略类对象的方法。

所以，**工厂模式返回的是具体的实现对象，而策略模式返回的是（拥有具体实现对象的）上下文对象。**

装饰模式 Decorator

使用的场景一：已经完成的系统或模块 A 不可改动，新需求需要对系统 A 增加新的功能。新的功能可以分为 B1，B2，Bn，**需要可选的、不分顺序的加在系统 A 上**。

场景二：**区分核心职责和装饰职责**。系统 A 完成的核心职责，而 Bn 负责满足特定情况下才会执行的 n 个特殊行为，每一个 Bn 只负责其独立的特殊职责。**由调用者根据需要有选择地、按顺序地使用装饰职责层层增强核心职责。**

如果新的功能只有一个 B，即没有变化，则是常见的在简单的在类 B 中注入类 A，在类 B 的方法 b 中调用类 A 的方法 a，在方法 b 中增加更多的逻辑。

装饰模式就像是人和衣服，人是核心职责，衣服是装饰职责，可以有很多，可以有选择地，按顺序地给人穿上不同的衣服。

工厂模式 Factory

Factory Method：定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使得一个类的实例化延迟到其子类。DP

工厂模式和简单工厂模式的区别

简单工厂模式的优点在工厂类中包含了必要的逻辑判断，根据调用者传递的条件，动态地实例化相关产品类。对于调用者而言，去除了对具体产品类的依赖。

问题在于，当增加一个新的产品类时，需要更改工厂类，违背了开闭原则。

从工厂类抽象出一个接口，接口方法返回抽象产品类，让所有的具体工厂类实现该接口。把简单工厂模式中的工厂类，变成了一个抽象工厂接口和多个具体工厂类。

当增加一个新的产品类时，不需要更改抽象工厂接口，只需要增加一个新的产品类和对应的具体工厂类。

简单工厂模式 = 产品接口 + 具体产品类 + 工厂类

工厂模式 = 产品接口 + 具体产品类 + 工厂接口 + 具体工厂类

抽象工厂模式 = 产品 A 接口 + 具体产品 A 类 + 产品 B 接口 + 具体产品 B 类 + 工厂接口 + 具体工厂类

抽象工厂模式 Abstract Factory

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。DP

缺点：当需要增加一个新的产品时，需要增加和修改多个类。

抽象工厂模式和工厂模式的区别

如果只有一类产品，比如运算类（加减乘除），工厂模式即可。

如果有多类产品，可使用抽象工厂模式。

代理模式 Proxy

Proxy，为其他对象提供一种代理，以控制对这个对象的访问。DP

使用场景一：远程代理，为一个对象在不同的地址空间提供局部代表，用于隐藏该对象存在于不同地址空间的事实。DP，引用 webservice

二、虚拟代理，根据需要创建开销很大的对象，通过它存放实例化需要很长时间的真实对象。DP，用于优化性能，比如在打开网页时，用虚拟代理代替真实图片，这样可以更快打开网页，虚拟代理只保留真实图片的路径和尺寸。

三、安全代理，用来控制真实对象访问时的权限。DP

四、智能指引，当调用真实对象时，代理处理另外一些事情。DP

比如前端调用后端服务，通过 bff 将多个服务的接口整合为一个接口。如果其中一个服务的接口开销很大，可能导致用户长时间的等待。可以使用代理接口返回 mock 数据，用户不用

等待既可以看到当前页面的 80%的真实数据，前端还是只调用一个接口不变，等到开销大的那个接口返回真实数据再返回给前端，代替代理接口的 mock 数据，实现二次响应。

一个接口调用如何实现二次响应？

原型模式 Prototype

Prototype：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。DP

只需要通过 new 构造函数实例化一次对象，之后就可以通过 **clone** 来创建对象，从而隐藏对象创建的细节，又提高了性能。

注意有浅复制和深复制的问题。

https://blog.csdn.net/river66/article/details/87859605

模板方法模式 Template

定义一个操作中的算法的骨架，而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新定义该算法的某些特定步骤。DP

模板方法模式把不变的行为抽象到超类，从而去除了子类中的重复代码，实现了代码复用。

模板类 + 虚方法

外观模式 Facade

Facade：为子系统中的一组接口提供一个一致的界面（接口），此模式定义了一个高层接口，该接口使得这一系统更加容易使用。DP

类似 BFF 的聚合功能，提供一个聚合了后端多个服务的接口供前端调用，避免前端直接面对后端多个服务。

使用场景一、设计初期阶段，有意识将不同的层分离，比如表示层、业务逻辑层和数据访问层。在层之间建立 Facade，降低耦合。

二、在开发阶段，子系统因为不断演化重构变得越来越复杂，增加 Façade 可以提供一个简单的接口，减少依赖。

三、维护一个遗留的大型系统，非常难以维护和扩展，而新的需求必须要依赖于它。可以为新系统开发一个外观 Facade，提供遗留代码的比较清晰的简单接口。新系统与 Facade 交互，Facade 与遗留代码交互所有复杂工作。R2P

外观模式和装饰模式的区别

装饰模式，像是**女生和衣服**，女生的需求是要有选择衣服的自由。

外观模式，则是**男生和衣服**，男生的需求是易用。

原来男生需要面对 10 件衣服的管理，中间增加了 Facade 负责管理 10 件衣服，男生只需要面对 Facade。

在代码层面上，装饰模式有一个抽象类，不同的衣服继承该抽象类。

外观模式有一个 Facade 类，直接注入各衣服类。

建造者模式 Builder

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。DP

用户只需要指定需要建造的类型就可以得到它们，而不需要知道具体建造的细节。

当构建一个复杂对象时，构建（装配）的过程可以分为 ABC，装配的顺序是不变的。

把装配的顺序抽象到**指挥者类 Director** 完成。

装配的步骤是相同的，把步骤抽象到**建造者接口 Builder** 的方法（或者是抽象类）。

每个步骤的具体实现不同，由具体的建造者类 **ConcreteBuilder** 实现。

最后完成的**产品类 Product**，可以放在建造者类 **ConcreteBuilder** 返回。

建造者模式 = 指挥者类（不变的装配顺序）+ 建造者接口（装配步骤）+ 具体建造者类（具体装配步骤实现）

观察者模式 Publish/Subscribe

也叫发布订阅模式，定义了一种一对多的依赖关系，让多个**观察者对象**同时监听某一个**主题对象**。这个主题对象在其状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。DP

将一个系统分割成一系列相互协作的类有一个很不好的副作用，需要维护相关对象间的一致性。我们不希望为了维护一致性而使得各类紧密耦合，这样会给维护、扩展和重用都都带来不便。DP

观察者模式的关键对象是主题类 Subject 和观察者 Observer。

一个 Subject 可以有任意数目的依赖它的 Observer，一旦 Subject 的状态发生了改变，所有的 Observer 都会得到通知。

主题类不需要知道谁是它的观察者。任何一个观察者不需要知道其他观察者的存在。

使用场景：一个对象的改变需要同时改变其他对象的时候。

该模式是在解除耦合。让耦合的双方都依赖于抽象，而不是依赖于具体，从而使各自的变化都不会影响另一方的变化。

通过事件委托，去除了抽象观察者接口，从而进一步解除具体观察者对抽象观察者的依赖。

JAVA 如何实现？

不同的服务之间，可以通过消息队列实现观察者模式。

同一个服务内部，如何实现观察者模式？

状态模式 **State**

当一个对象的内在状态改变时，允许改变其行为，这个对象看起来像是改变了其类。DP

使用场景：if-else 过长。

主要解决当控制一个对象状态转换的条件表达式过于复杂的情况。**把状态的判断逻辑转移到表示不同状态的一系列的类当中**，可以把复杂的判断逻辑简化。

好处是将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。DP

意思就是如果有连续 10 个 if-else，就分成 10 个独立的**状态类 State（解耦）。每个具体 State 类只处理自己的逻辑，非自己的职责转交给下一个状态类处理。**

注：意味着每个具体状态类还需要知道自己的下一个状态类。

注：每个具体 State 类在设置下一个状态类时可能出现死循环，比如 StateA 设置下一个状态类是 StateB，而 StateB 设置下一个状态类的 StateA。

在**上下文类 Context** 中，存放**当前状态类**、状态切换和完成的条件，并负责调用当前状态类的方法，该方法在状态类实现的接口中定义。

调用者只需要知道上下文类 Context 即可。

状态模式和职责链模式的区别

一、两者的类关系图不同，职责链模式是一种自己聚合自己（具体实现子类中聚合了相同的父类），而状态模式是聚合+接口：**上下文类 Context** 聚合**状态处理接口 State**，**具体状态处理子类 StateA** 等实现接口 State。

另外，这种自聚合关系还出现在装饰模式中。

二、职责链模式的调用者需要实例化并设置职责链的各环节，意味着职责链的各环节可以在运行中动态组成。

状态模式的各处理环节是在开发环节就已经固定了。

职责链模式 Chain of Responsibility

使得多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。**将这些对象连成一条链条**，并沿着这条链传递该请求，直到有一个对象处理它为止。DP

发送请求的客户端并不知道最终是哪个对象处理它发出的请求，这样系统的更改可以在不影响客户端的情况下动态地重新组织和分配责任。

当客户端提交一个请求时，请求是沿着链条传递，直到有一个 ConcreteHandler 类对象负责处理它。DP
客户端定义链条的结构，定义每一个处理环节的后续继任者。客户端可以随时增加或修改一个链条的结构，增强了给对象指派职责的灵活性。DP
注意：一个请求可能到了链条末端也得不到处理，或者因为没有正确配置而得不到处理。也可能出现死循环。

链条中的对象也不知道其他对象，更不知道整条链条的结构全貌。

适配器模式 Adaptor

将一个类的接口转换成客户希望的另外一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。DP
包括类适配器模式和对象适配器模式。GoF

使用场景：现有的类 A 或接口 a 无法改动，又需要复用。
而现有的架构 B 已经实现，无法改动，比如已经设计了统一的对外接口 b。
需要把类 A 或接口 a 统一入现有的架构 B 中。

比如，翻译

适配器模式和代理模式的区别

一是**代理模式**要求代理对象和真实对象实现自同一个接口，适配器模式不需要。
二是适配器模式的场景往往是运维阶段，需要将无法改动的系统或接口统一到现已存在的接口架构中；而代理模式的场景是设计阶段，设计真实对象的代理对象。

备份模式 Memento

在不破坏封装性的前提下，捕捉一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。DP

游戏存档

Originator 发起人类：需要被备份的类。负责**创建**备份类，并可使用备份类**恢复**内部状态。
Memento 备份类：负责存储发起人类的（部分或全部的）内部状态。防止发起人类以外的其他对象访问。
Caretaker 管理者类：负责保存备份类，不能对备份类的内容进行操作或者检查。

组合模式 Composite

将对象组合成**树形结构**以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。DP

迭代器模式 Iterator

提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。DP

该模式分离了集合对象的遍历行为，抽象出一个迭代器类/接口来负责，从而实现即不暴露集合的内部结构，又可以让外部代码透明地访问集合内部的数据。

单例模式 Singleton

保证一个类仅有一个实例，并提供一个访问它的全局访问点。DP

饿汉式单例 vs 懒汉式单例

类加载时实例化 vs 第一次被引用时实例化

应用场景：

类的实例化复杂且时间长。

注意：

类的成员变量和静态成员变量线程不安全。

考虑加锁或使用 ThreadLocal。

桥接模式 Bridge

将抽象部分与它的实现部分分离，使得它们都可以独立地变化。DP

如果你设计类结构时，发现实现方式有 2 种（或更多），桥接模式的核心就是把这些实现独立出来，互不影响，让它们各自地变化。每种实现的变化不会影响其他实现方式。

实现系统可以有多角度的分类，每一种分类都有可能变化，那么就把这种多角度变化的每一个角度都分离出来，让它们各自变化，减少它们之间的耦合。

比如手机有品牌 A 和品牌 B，手机软件有通讯录和游戏，每一种分类都有可能变化，比如出现品牌 C 等等，软件可以增加音乐播放器等。

手机品牌 手机软件	品牌 A	品牌 B

通讯录		
软件		

在横轴和纵轴上都有可能发生变化，比如增加更多的品牌，更多的软件。

桥接模式和访问者模式的区别

一、桥接模式是在两个方向上变化，访问者模式在其中一个方向上是固定的。

二、因为访问者模式在一个方向上是固定的，比如只有 2 种类别，所以在另一个方向的实现上，可以直接列出这两种类别对应的实现。

三、访问者模式把固定的那个方向视为稳定的数据结构，把变化的那个方向视为操作。

通过访问者模式的设计，把数据和操作分离解耦，增加新的操作就增加一个新的具体访问者类 Visitor。

人类 场景	男人	女人
成功		
失败		
结婚		

四、访问者模式因为有固定的数据结构，所以具体的 Element 类是固定不变的，而抽象访问者类 Visitor 可以有稳定的抽象方法，分别对应固定的具体 Element 类，而具体访问者类负责抽象方法的具体实现。

五、访问者模式进一步定义了数据结构对象 ObjectStructure，提供一个接口以允许一个具体的访问者（遍历）访问它的元素。

命令模式 Command

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。DP

命令模式通过增加**命令接口 ICommand + 具体命令类 Command，**把**请求一个操作的对象 Invoker** 和知道怎么**执行一个操作的对象 Receiver** 分离（解耦）。

通过增加 **Invoker** 类，把调用者 **Client** 和具体执行者 **Receiver** 解耦。

使用场景：需要不固定地调用某个类的几个执行方法，可能反复调用。需要记录执行日志，调用可以被撤销和恢复。

如果没有命令模式，意味着 Client 需要直接调用 Receiver 类（强耦合）。
通过增加 Invoker 类和 Command 类，可以实现 Client 和 Receiver 类的解耦。Client 甚至可以不知道 Receiver 类的存在。Client 只需要通过 Invoker 类的接口，就可以实现新增（撤销、恢复）命令，以及执行命令。

这种解耦是通过增加 Invoker 类和 Command 类，以及增加 Invoker 类和 Command 类之间，Command 类和子类，Command 类和 Receiver 类之间的聚合关系实现的。
简言之，**通过增加类、接口、聚合实现解耦**。

命令模式的优点：
一、较容易地设计一个命令队列；
二、在需要的情况下，可以较容易地将命令记入日志；
三、允许接收请求的一方 Invoker 决定是否要否决请求。
四、较容易地实现对请求的撤销和重做；
注：以上均是通过 Invoker 类实现。
五、增加新的命令类不影响其他的类，因此增加新的具体命令类很容易；

类图 Invoker 和 Receiver 是不是搞反了？不是

操作请求类 Invoker + 操作执行类 Receiver + 命令接口 ICommand + 具体命令类 Command

操作请求类 Invoker：负责设置需要的命令对象
具体命令类 Command：负责通过调用操作执行类 Receiver 执行
操作执行类 Receiver：负责具体命令的具体执行

中介者模式 **Mediator**

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显示地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

优点是通过 Mediator 类减少了各 Colleague 类之间的耦合，使其可以独立地改变和复用各个 Colleague 类和 Mediator 类。
二是由于把对象如何协作进行了抽象，把中介作为一个独立的概念并封装在一个对象中，从而使中介对象关注交互行为，而 Colleague 对象关注本身的业务。

缺点是 ConcreteMediator 类可能因为 ConcreteColleague 类越来越多，而变得非常复杂，反而不容易维护。
把交互的复杂性转移到中介类的复杂性。

中介者模式容易在系统中应用，也容易误用。如果系统中出现了多对多交互复杂的对象群时，不要急于使用中介者模式，而要先反思系统在设计上是不是合理。

享元模式 **Flyweight**

运用共享技术有效地支持大量细粒度的对象。DP

内部状态：在享元对象内部并且不会随环境变化而改变的共享部分。
外部状态：随环境改变而改变，不可以共享的状态就是外部状态。

使用场景：如果一个应用使用了大量的对象，而这些大量的对象造成了很大的存储开销时，就应该考虑使用享元模式；
二、对象的大多数状态可以外部状态，如果删除对象的外部状态，那么就可以用相对较少的共享对象取代很多组对象，此时可以考虑使用享元模式。

使用享元模式可以大大减少实例总数。
比如 String 类就使用了 Flyweight 模式。

比如棋类游戏的棋子对象，颜色是内部状态，位置是外部状态。如果不使用享元模式，围棋可能产生最多 361 个棋子对象，现在可以减少到 2 个实例。

UnsharedFlyweight 类解决那些不需要共享对象的问题。

解释器模式 Interpretor

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。DP

如果一种特定类型的问题发生频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。DP

比如字符串匹配。判断 email，手机号码。一种方案是为每一个特定需求写一个算法函数。另一种是使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符串的集合。DP

比如浏览器，解释 HTML 语法。

访问者模式 Visitor

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。DP

访问者模式适用于数据结构相对稳定的系统，**把数据结构和作用在结构之上的操作解耦，从而使操作集合可以相对自由地演化。**

访问者模式的目的是把处理从数据结构分离出来。把算法和数据结构分开，这样有比较稳定的数据结构，又有易于变化的算法。

缺点是增加新的数据结构变得困难。