设计模式

## 6大设计原则

1. 单一职责原则（Single Responsibility Principle SRP）

There should never be more than one reason for a class to change.

好处：

* 类的复杂性降低，实现什么职责都有清晰明确的定义；
* 可读性提高；
* 可维护性提供；
* 变更引起的风险降低（修改不会造成对其他接口的影响，这对系统的扩展性、维护性都有非常大的帮助）。

1. 里氏替换原则（Liskov Substitution Principle LSP）

定义：

* 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1都代换成o2时，程序P的行为没有发生变化，那么类型S是类型T的子类型；
* 所有引用基类的地方必须能透明的使用其子类的对象（只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，但是反过来就不行）。

理解：

* 子类必须完全实现父类的方法（在类中调用其他类的时候要使用父类或接口类型，如果不能使用父类或接口，表示已经违背LSP原则）；

如果子类不能完整地实现父类方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系，采用依赖、聚集、组合等关系替代泛化（继承）；

* 子类可以有自己的个性；
* 覆盖或实现父类的方法时输入参数可以被放大（子类中方法的前置条件必须与超类中被覆写的方法的前置条件相同或更加宽松）；

例如：父类方法参数类型为HashMap，子类同名方法的参数类型Map，此时不是重写二是重载，用子类替换父类调用时，传入HashMap，依然调用父类方法，符合里氏替换原则；

* 覆写或实现父类的方法时输出结果可以被缩小。

1. 依赖倒置原则（Dependence Inversion Principle DIP）（面向接口编程 OOD）

含义：

* 高层模块不应该依赖底层模块，但都应该依赖其抽象；
* 抽象不应该依赖细节；
* 细节应该依赖抽象。

在Java中的表现为：

* 模块间的依赖通过抽象（接口或抽象类）发生，实现类之间不发生直接依赖；
* 接口或抽象类不依赖于实现类；
* 实现类依赖接口或抽象类。

好处：

* 减少类间耦合，提高系统稳定性，降低并行开发引起的风险，提高代码的可读性和可维护性。

最佳实践：

* 每个类尽量都有接口或抽象类，或者抽象类和接口两者都具备；
* 变量的表面类型尽量是接口或抽象类；
* 任何类都不应该从具体类派生；
* 尽量不要覆写基类的方法；
* 结合里氏替换原则使用。

1. 接口隔离原则（建立单一接口，不要建立臃肿庞大的接口）

含义：

* 客户端不应该依赖它不需要的接口；
* 类间的依赖关系应该建立在最小的接口上（要求接口细化、接口纯洁）。

与单一职责原则的区别：

角度不同，单一职责原则注重的是职责，是从业务逻辑上的划分，要求类和接口的职责单一；而接口隔离原则要求接口的方法尽量少。

保证接口的纯洁性：

* 接口要尽量小，但首先必须满足职责单一原则；
* 接口要高内聚（提高接口、类、模块的处理能力，减少对外的交互，具体到接口隔离原则，要求在接口中尽量少公布public方法，对外的承诺越少对系统开发越有利，变更的风险越少，有利于降低成本）；
* 定制服务（为不同的客户端（调用者）提供不同的接口，只提供访问者需要的方法）；
* 接口设计是有限度的（接口设计粒度越小系统月灵活，但会带来结构复杂化、开发难度增加、可维护性降低等问题，所以接口设计一定要适度）。

最佳实践：

* 一个接口只服务于一个子模块或业务逻辑；
* 通过业务逻辑压缩接口中的public方法；
* 已经被污染了的接口，尽量去修改，若变更风险太大，则采用适配器模式进行转化处理；
* 了解环境，拒绝盲从，环境不同，接口拆分标准就不同。

1. 迪米特法则（Law of Demeter LoD） 最少知识原则（Least Knowledge Principle LKP）

一个对象应该对其他对象有最少的了解（一个类应该对自己需要耦合或调用的类知道得最少，不关心其他类的具体实现，只需要按照约定调用public接口就可以了）。

含义：

* 只和朋友交流（例：老师叫体育委员清点女生人数，老师类不应该直接创建、调用女生类实例，老师只需创建、调用体育委员类实例）；
* 朋友间也是有距离的（对外不要公布太多public方法和非静态的public变量，尽量内敛，与接口隔离原则有类似的地方，可通过业务逻辑压缩public方法数量）；
* 是自己的就是自己的（当出现一个方法放在本类中也可以，放在其他类中也可以时，如果该方法放在本类中既不增加类间关系，也对本类不产生负面影响，那就放在本类中）；

最佳实践：

其核心观念就是类间解耦、弱耦合，只有弱耦合了以后才可以提高类的复用，但是其结果就是产生大量中转或跳转类，提高了系统复杂性，也为维护带来了难度，因此使用该法则需要反复权衡，既要做到结构清晰，又要做到高内聚低耦合；在实际运用中，如果一个类跳转两次以上才能访问到另一个类，就需要想办法重构了。

1. 开闭原则

Software entities like classes, modules and function should be open for extension（扩展） but closed for modifications（修改）.

精要：

“拥抱变化”，应尽量通过扩展的方式实现变化，而不是通过修改已有代码来实现变化，是对软件实体的未来事件而制定的对现行开发设计进行约束的原则；开闭原则是最基础的原则，是其他5大原则的精神领袖。

如何使用开闭原则：

* 抽象约束，1）通过接口或抽象类约束扩展，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；2）参数类型、引用对象尽量使用接口或抽象类，而不是实现类；3）抽象层尽量保持稳定，一旦确定即不允许修改；
* 元数据控制模块行为（配置优于编码）；
* 制定项目章程（约定优于配置）；
* 封装变化，1）将相同的变化封装到一个接口或抽象类；2）不同的变化封装到不同的接口或抽象类；封装变化，找出预计有变化或不稳的点，为这些变化点创建稳定的接口，以应对业务变化的实现，提高系统扩展性，通过扩展实现变化，提示系统稳定性和健壮性。

## 23种设计模式

1. 单例模式（Singleton Pattern）

Ensure a class has only one instance, and provide a global point of access to it.

缺点：

* 单例模式没有接口，扩展很困难，一般只能进行代码修改实现变化（特殊情况下，单例模式也可以实现接口或被继承，需根据实际情况判断）；
* 单例模式对测试是不利的，并行开发环境中，单例模式没有完成就不能进行测试，也无法mock；
* 单例模式与单一职责原则有冲突，因为单例模式有可能完成不知一个业务逻辑。

使用场景：

* 要求生产唯一序列号的环境；
* 在整个项目中需要一个共享访问点或共享数据；
* 创建一个对象需要消耗的资源过多，如IO和数据库访问；
* 需要定义大量的静态常量和静态方法（如工具类）的环境。

注意事项：

* 线程不安全的单例模式，

pivate static Singleton singleton = null;

public static Singleton getSingleton(){

if (singleton == null) // 如此，若出现判断，多个线程运行到这判断都为true，此时出现多个实例

singleton = new Singleton();

}

rturn sngleton;

}

* 对象复制的影响，若单例模式实现Cloneable接口，对象复制不需要调用构造函数，此时出现多个实例。

1. 工厂方法模式（Factory Pattern）

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclassea（工厂方法使一个类的实例化延迟到其子类）.

优点：

* 良好的封装性，代码结构清晰（创建一个对象是有条件的，调用者只需要知道产品的类名或约束字符串就可以了，不必关心创建过程）；
* 扩展性非常优秀（在增加产品类时，只需要适当修改具体的工厂类或扩展一个工厂类，就可以完成“拥抱变化”）；
* 屏蔽产品类（产品类的实现如何变化，调用者不需要关系，只需要关心产品的接口，只要接口不变，上传模块就不需要发生变化）；
* 是典型的解耦框架（高层模块只需要知道产品的抽象类，其他实现不需要关系；符合迪米特法则，不需要的就不进行交流；符合依赖倒置原则，只依赖产品的抽象类；符合里氏替换原则，产品子类替代产品父类没问题）。

使用场景：

* 替代new一个对象，但需要根据实际考虑是否有必要增加一个工厂，这样会使代码复杂度增加；
* 需要灵活、可扩展的框架时，如使用JDBC连接数据库；
* 工厂方法模式可以在异构系统中使用，如webservice，用工厂类管理从wsdl中产生的对象（作为产品类），减少与外围系统的耦合；
* 测试驱动开发时可以使用，如测试一个类A需要实例化关联的类B，可以使用工厂方法将B虚拟出来。

1. 抽象工厂模式（Abstract Factory Pattern）

Provide an interface for creating families of related or dependent objects without specifying their concrete classes（无需制定他们的具体类）.

优点：

* 封装性，高层不关心产品类具体实现，只要知道正确的工厂类就能实例化正确的产品；
* 产品族内的约束为非公开状态，如各工厂方法生产的产品使用一定比例进行约束。

使用场景：

一个对象族具有相同的约束，则可以使用抽象工厂模式，如通用代码，A、B产品，A1、A2产品……，他们具有相同或相似约束，甚至A、B抽象类可以有共同上级抽象类或接口。

注意事项：

* 抽象工厂模式最大的缺点就是产品族扩展非常困难，如通用代码，若增加C产品，将带来非常大的代码修改，AbstractCreator要增加一个创建C产品的抽象方法，然后两个实现类也需要修改，这严重违反了开闭原则；且，修改抽象类或接口，也违反了抽象类和接口的契约作用；
* 但是，扩展产品等级却是非常容易的，若增加一个级产品，只需要增加一个工厂类负责新产品的生产即可，也就是横向扩展容易，纵向扩展困难，从这一点看抽象工厂模式是符合开闭原则的。

1. 模板方法模式（Template Method Pattern）

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，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；其中，抽象模板中（一般是上层抽象类）一般包含两种方法，基本方法，由子类实现，模板方法，可以有一个或多个，且一般是具体方法，承担着框架或算法的角色，实现对基本方法的调度以完成特定的逻辑。

优点：

* 封装不变部分（模板方法），扩展可变部分（基本方法）；
* 提取公共部分代码，便于维护；
* 行为由父类控制，子类实现。

使用场景：

* 多个子类有公有方法，且逻辑基本相同；
* 重要、复杂的算法，可以把核心的算法设计成模板方法，周边各个细节则由子类实现；
* 重构时，把相同代码抽取到父类中，然后通过钩子（通常为一个设置父类标志属性的方法，在子类中进行重写以影响模板方法的规则）函数约束其行为。

注意事项：

* 为防止恶意操作，一般模板方法都加上final关键字，不允许被修改；
* 抽象模板中的基本方法尽量设计为protected类型，符合迪米特法则；不需要暴露的属性或方法尽量不要设置为protected，因为子类可以改变这些属性或方法逻辑；实现类若非必要，尽量不要扩大父类的访问权限。

1. 建造者模式（Builder Pattern）

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构建不同的表示。

4个角色：

* Product产品类，通常是实现了模板方法模式的，且建议这样做，组合使用使类结构、代码实现变得非常清晰（这也算是对建造者模式的扩展）；
* Builder抽象建造者，规范产品的组建；
* ConcreteBuilder具体建造者；
* Director导演类，负责安排已有模块的顺序，然后告诉Builder开始建造。

优点：

* 封装性，客户不需要知道产品的内部组成细节；
* 建造者独立，容易扩展，各个ConcreteBuilder是相互独立的，对系统扩展非常有利；
* 便于控制细节风险，由于建造者相互独立，因此可以对建造过程逐步细化，而不对其他模块产生任何影响。

使用场景：

* 相同的方法，不同的执行顺序，产生不同的事件结果；
* 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同；
* 产品类非常复杂，或者产品类中的方法调用顺序不同而产生了不同的结果；
* 在对象的创建过程中会使用到系统中的一些其他对象，这些对象在产品的创建过程中不易得到时（该种场景只是一种补偿方法，因为一个对象不以获取，而在计划阶段没有发觉，通过使用建造者模式柔化创建过程，本身已经违反设计的初衷）。

注意事项：

建造者模式关注的是零件类型和装配工艺（顺序），而工厂方法模式侧重点在创建上，创建零件是它的主要职责，两者注重点不一样。

1. 代理模式（Proxy Pattern）

为其他对象提供一种代理以控制这个对象的访问。

角色：

* Subject抽象主题，接口或抽象类，普通业务类型定义；
* RealSubject具体主题，业务逻辑的具体执行者；
* Proxy代理主题，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并在真实角色处理完毕前后做预处理和善后处理。

优点：

* 职责清晰，真实的角色实现实际的业务逻辑，不关心其他非本职责的事务，通过后期的代理完成一件事务，结果自然简洁清晰；
* 高扩展性，真实实现类的接口只要不变，具体业务不管如何变，代理类不需要任何修改；
* 智能化，如动态代理。

扩展：

* 普通代理：需要注意的是，为防止直接new真实主题类，一般采用项目组约束的方式，采用技术约束会对系统维护带来很多不利因素；
* 强制代理：通俗的讲就是，高层模块new了一个真实的主题类对象，返回的却是代理类实例，且代理类的实例化由真实主题类自己管理和实例化；
* 动态代理：在运行阶段才指定代理哪一个对象。

1. 原型模式（Prototype Pattern）（对象复制）

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。

优点：

* 性能优良，原型模式是在内存中进行二进制流的拷贝，比new一个对象性能好很多，特别是在循环内产生大量对象；
* 避免构造函数的约束，这也是其缺点，直接拷贝，不会执行构造函数，优缺点均是避免了构造函数的约束，只需要在实际使用中考虑是否使用原型模式。

使用场景：

* 资源优化，类的初始化需要消化非常多的资源，包括数据、硬件资源等；
* 性能和安全要求，通过new需要非常繁琐的数据准备或访问权限；
* 一个对象多个修改者。

注意：

* 原型模式一般与其他设计模式一起使用；
* 需要注意对象的浅拷贝和深拷贝问题；
* 类成员变量不能使用final关键字。

1. 中介者模式（Mediator Pattern）

用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示的相互作用，从而使其耦合松散，而且可以独立的改变他们之间的交换。

角色：

Mediator 抽象中介：定义各同事角色之间的通信接口；

Concrete Mediator 具体中介，实现抽象中介的业务逻辑；

Colleague 同事，每一个同事角色都需要知道中介者角色，与其他同事通信必须通过中介者；每个同事类都有自发行为（不依赖其他同事或中介）和依赖方法（必须依赖中介者才能完成）。

优点：减少类间依赖；

缺点：中介者会膨胀的很大，业务逻辑复杂；

使用场景：类间关系出现了蜘蛛网状，即：

N个对象之间产生了相互依赖（N>2）

多个对象依赖，但是依赖关系还不明确或又发生变化的可能；

产品开发，产品以稳定、高效、扩展为宗旨。

1. 命令模式（Command Pattern）

将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

角色：

Receive：接收者，执行业务逻辑的对象；

Command：命令，需要执行的所有命令抽象；

Invoker：调用者，接收命令并执行。

优点：

* 类间解耦，调用者与接收者没有直接依赖，调用者只需要调用Command的execute方法，不需要了解到底是哪个接收者执行；
* 可扩展性，Command子类可轻松扩展，Invoker与Client不产生严重代码耦合；
* 与其他模式结合，与责任链模式结合科实现命令族解析任务，与模板模式结合可减少Command子类膨胀的问题。

1. 责任链模式（chain\_of\_responsibility）

使多个对象都有机会处理请求，从而避免了请求的发送者和接受这之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理为止。

抽象处理者的三个职责：1)定义一个请求处理方法handleMessage，唯一对外开放（一般进行是否适合处理的判断并调用具体处理者的处理方法echo()）；2)定义一个链的编排方法setNext；3)定义具体处理者必须的两个方法，自己能够处理的级别getHandlerLevel和具体业务处理方法echo。

优点：将请求和处理分开，两者解耦，提高系统灵活性；

缺点：

* 性能问题，每个请求都要从链的头遍历到尾，当链很长时问题凸显（为了避免，一般是在Handler中设置一个最大节点数量，在setNext方法中判断是否已经超过其阈值，超过则不允许增加节点，避免无意识地破坏系统性能）；
* 调试不方便，当链比较长时，由于采用了类似递归的方式，调试时逻辑可能比较复杂。

1. 装饰模式（Decorator Pattern）

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更加灵活。

角色：

* Component：核心业务抽象类；
* ConcreteComponent：核心业务实现类；
* Decorator：装饰类，实现Component以代理实际业务处理对象，因此它里面必然有一个private变量指向Component；
* ConcreteDecoratorA、ConcreteDecoratorB……：具体装饰角色。

优点：

* 装饰类和被装饰类可独立发展，相互不耦合；
* 装饰模式是继承关系的一个替代方案，不管装饰多少层返回的对象依然是Component；
* 装饰模式可以动态的扩展一个类的功能。

缺点：

* 装饰层级过多时，系统变得复杂，想象一下剥洋葱，很有可能你要剥到最后一层才知道错误发生在哪里。

使用场景：

* 需要扩展一个类的功能，或给一个类增加附加功能；
* 需要动态的给一个对象增加功能，这些功能也可以动态的撤销；
* 需要为一批兄弟类进行改装或加装功能。

1. 策略模式（Strategy Pattern）

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。

角色：

* Context：封装角色（锦囊），承上启下，屏蔽高层模块对策略、算法的直接访问，封装可能存在的变化（是策略模式的重点，它借用了代理模式的思路，差别就是策略模式的封装角色和被封装的策略类不用是同一个接口）；
* Strategy：抽象策略对象，定义策略或算法必须具有的属性和方法；
* ConcreteStrategy：具体策略角色。

优点：

* 算法可以自由切换：这是策略模式本身定义的，只要实现了抽象策略，它就是策略家族的一员，通过封装角色即可使用；
* 避免使用多重条件判断：高层模块通过封装类决定使用哪一个策略，避免了多重条件判断；
* 扩展性良好：增加策略，只需要添加策略接口的实现，其他都不用修改，类似于一个可反复拆卸的插件。

缺点：

* 策略类数量增多：每一个策略都是一个类，复用的可能性小，类数量增多；
* 所有的策略类都需要对外暴露：上层模块必须要知道有哪些策略，然后才能决定使用哪一个，这与迪米特法则是相违背的，可使用工厂方法模式、代理模式或享元模式进行修正。（注意：如果一个系统中策略家族的具体策略数量超过4个，则需要考虑使用混合模式，解决策略类膨胀和对外暴露的问题。）

使用场景：

* 多个类只有在算法或行为上稍有不同；
* 算法需要自由切换；
* 需要屏蔽算法规则的场景。

策略模式的扩展：**策略枚举**，策略枚举是一个非常优秀和方便的模式，但是它受枚举类型的限制，每个枚举项都是public final static的，扩展性受到一定约束，因此在系统开发中，策略枚举一般担当不经常发生变化的角色。

1. 适配器模式（Adapter Pattern）

将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

角色：

* Target目标角色，定义把其他类转换为何种接口；
* Adaptee源角色，一般是已经存在、运行良好的类或对象；
* Adapter适配器角色，一般通过继承（类适配器）或关联（对象适配器）Adaptee并实现Target接口实现转换。

优点：

* 适配器模式可以让两个没有任何关系的类在一起运行；
* 增加了类的透明性，通过Target调用源角色的方法，高层模块也不需要了解源角色；
* 提高了类的复用性，源角色在原有系统中依然能用的同时又成为了Adapter中的新演员；
* 灵活性非常好，当不需要适配器时删掉即可，其他地方不需要进行代码修改。

使用场景：修改一个已经投产中的接口时。

注意：最好在详细设计阶段不要考虑使用，它不是为了在开发阶段解决问题的，而是解决正在服役的项目问题；另一点，项目一定要遵守依赖倒置和里氏替换原则，否则即使在适合使用适配器模式的场合下也会非常大的改造。

适配器模式的**扩展**：将继承关系改为关联关系，可实现适配多个源角色。

1. 迭代器模式（Iterator Pattern）（已很少使用）

它提供一种方法访问一个容器对象中的各个元素，而又不需暴露该对象的内部细节。

角色：

* Iterator抽象迭代器，负责定义访问和遍历元素的接口；
* ConcreteIterator具体迭代器；
* Aggregate抽象容器，负责提供创建具体迭代器角色的接口，一定会有一个类似createIterator()这样的方法；
* ConcreteAggregate具体容器。

使用场景：由于其在各种语言中已广泛使用融入了其API中，项目中一般不单独使用该模式。

1. 组合模式（Composite Pattern）

将对象组合成树形结构以表示“部分-整体”的层次关系，使得用户对单个对象和组合对象的使用具有一致性。

角色：

* Component抽象构件，定义参加组合对象的公共方法和属性，可以定义一些默认的行为或属性；
* Leaf叶子构件；
* Composite树枝构件，它的作用是组合树枝节点和树叶节点形成一个树形结构。

优点：

* 高层模块调用简单，树形结构中的所有节点都是Component，局部和整体对调用者来说没有任何区别，高层模块不必关系自己处理的是单个对象还是整个组合结构；
* 节点自由增加。

缺点：

组装树时使用到了树枝和树叶的实现类，在面向接口编程中是很不恰当的，也与依赖倒置原则冲突，由于它限制了接口的影响范围，在使用的时候需要考虑。

使用场景：

* 维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理；
* 从一个整体中能够独立出部分模块或功能的场合。

组合模式的**扩展**：透明模式（上面的叫安全模式），将组合（Composite 树枝）使用到的方法和属性都抽象到Component抽象构件中。

1. 观察者模式（Observer Pattern）

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

角色：

* Subject被观察者，定义被观察者必须实现的职责，必须能够动态增加、取消观察者，一般是抽象类或实现类，仅仅完成作为被观察者的职责：管理观察者并通知观察者；
* Observer观察者，接收到消息后对消息进行处理（update()）;
* ConcreteSubject具体的被观察者；
* ConcreteObserver具体的观察者，每个观察者对消息的处理都不同，各自实现自己的逻辑。

优点：

* 观察者和被观察者之间是抽象耦合，则两者的扩展是非常容易的；
* 建立一套触发机制，被观察者可以触发一系列职责单一（多个观察者）的动作。

使用场景：

* 关联行为场景，但关联行为是可以拆分的（被观察者管理观察者，且观察者可插拔），而不是“组合”关系；
* 事件多级触发的场景；
* 跨系统的消息交换场景，如消息队列处理机制。

注意事项：

* 开发效率和运行效率问题，一个被观察者，多个观察者，开发调试都比较复杂，且在Java中消息通知默认是顺序执行，一个观察者卡壳，会影响到整体的执行效率，此时，建议采用异步的方式；
* 广播链的问题，当一个观察者既是观察者又是被观察者且形成多级时，逻辑就变得复杂，可维护性非常差，此时，建议在一个观察者模式中最多出现一个对象既是观察者又是被观察者。

Java中已实现该模式，java.util.Observable/Observer，实际使用时被观察者继承Observable类即可调用对观察者的管理相关方法，观察者实现Observer接口中的update方法即可。

1. 门面模式（Façade Pattern）

要求一个子系统的外部和其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用（强调“统一对象”，子系统除了通过该接口（门面对象）进行访问不允许其他任何方式访问子系统）。

角色：

Façade门面角色，客户端通过该对象的方法进行调用子系统功能，该对象知晓子系统的所有功能和责任；

subsystem子系统角色，可以有一个或多个子系统。

优点：

* 减少系统的相互依赖，客户端不需要直接依赖子系统，门面对象就是子系统的委托，提升子系统维护的灵活性；
* 提供了灵活性，依赖减少了灵活性自然提高了，不管子系统如何变化，只要不影响到门面对象，其他地方无需修改；
* 提高安全性，想让你访问子系统的哪些业务就开通哪些逻辑，没有在门面对象中开通的方法是无法访问的。

缺点：

不符合开闭原则（对修改关闭，对扩展开放），门面对象在需求发生变化或发生问题时只能进行修改而无法进行扩展。

使用场景：

* 为一个复杂的模块或子系统提供一个外接访问的接口；
* 子系统相对独立，外接对子系统的操作只允许黑箱操作；
* 预防低水平人员带来风险扩散（只能在子系统中进行开发，然后提供门面）。

注意事项：

* 一个子系统可以有多个门面（门面已经很庞大、子系统可以提供不同的访问路径（逻辑））；
* 门面不参与子系统的业务。

1. 备忘录模式（Memento Pattern）（标准备忘录模式很难在项目中遇到，一般使用其变换处理方式）

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原先保存的状态。

角色：

Originator发起人角色，记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据；

Memento备忘录角色，负责存储Originator发起人对象的内部状态，在需要的时候提供发起人需要的内部状态；

Caretaker备忘录管理员角色，对备忘录进行管理、保存和提供备忘录。

注意事项：

* 备忘录的生命期，备忘录创建出来就要在“最近”的地方使用，要主动管理其生命周期，建立就要马上使用，不用就要立即删除其引用，等待垃圾回收器进行回收；
* 备忘录的性能，不要在频繁建立备份的场景中使用（比如for循环中），一是控制不了备忘录建立的对象数量，二是大对象的建立是需要消耗资源的，系统性能需要考虑。

备忘录模式的**扩展**：

* clone方式的备忘录，需要注意的是尽量不要与其他对象产生严重耦合（深克隆）；
* 多状态的备忘录模式，利用反射解析对象所有属性和值；如果要设计一个在运行期决定备份状态的框架，建议采用AOP框架来实现，避免采用动态代理无谓的增加程序逻辑的复杂性；
* 多备份的备忘录，（比如）使用时间戳作为key的Map来装多个备忘录，同时为了备份的安全，可将备忘录角色作为发起人角色的私有内部类，且备忘录角色实现一个空接口，对外暴露该接口即可。

1. 访问者模式（Visitor Pattern）

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新操作。

角色：

Visitor：抽象类或接口，声明访问者可以访问哪些元素，一般就是visit方法的参数；

ConcreteVisitor；

Element：抽象元素，接口或抽象类，声明接受哪一类访问者，一般是accept方法参数进行指定；

ConcreteElement：实现accept方法，通常是visitor.visit(this)；

ObjectStruture：结构对象，元素产生者，Java中一般使用集合框架接口作为容器；

优点：

* 符合单一职责原则：Visitor和Element各自分工明确；
* 优良的扩展性：同一访问者增加方法或提供多个访问者均可快速增加对数据的不同操作；
* 灵活性非常高：扩展好当然灵活。

缺点：

* 违反迪米特法则，Element对Visitor公开了细节；
* Element的变更比较困难，由于违反了迪米特法则，Visitor也需要修改；
* 违背了依赖倒置原则，Visitor依赖Element具体的实现，不依赖接口导致扩展困难。

使用场景：

业务规则要求遍历多个不同的对象，针对不同对象执行不同操作（这也是对迭代模式的扩充，迭代模式只能实现类型相同对象的遍历）。

1. 状态模式（State Pattern）

当一个对象内在状态改变时允许其改变行为，这个对象看起了像改变了其类（状态的改变引起了行为的改变，好像对象对应的类发生了变化）。

角色：

State：抽象状态，接口或抽象类，负责对象状态定义，封装环境角色以实现状态切换；

ConcreteState：具体状态角色，完成两个职责：本状态的行为管理和趋向状态管理（本状态下要做的事，如何过渡到其他状态）；

Context：环境角色，定义客户端调用接口，辅助状态切换。

优点：结构清晰（不需要大量判断）；遵循开闭、职责单一、迪米特法则；封装性好（通过Context调用，屏蔽了状态变换的细节）。

缺点：状态多了引发状态子类膨胀。

1. 解释器模式（Interpreter Pattern）（不常用，有其他工具或语言（Expression4J、MESP、Jep或shell、JRuby、Groovy））

给定一门语言，定义它的一种文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

角色：

AbstractExpression：抽象解释器；

TerminalExpression：终结符表达式（运算元素）；

NonterminalExpression：非终结符表达式（运算符）；

Context：环境角色（运算参数）。

缺点：

类膨胀，每个语法都要产生一个非终结符表达式；递归和循环的使用带来效率问题。

1. 享元模式（Flyweight Pattern）

使用共享对象可有效地支持大量的细粒度的对象。

**内部状态**：可共享信息，不随环境改变而改变（非共通属性）；

**外部状态**：对象得以依赖的标记（主键），随环境改变而改变、不可共享，是一批对象的统一标识，是唯一的一个索引值。

角色：

Flyweight：抽象享元角色，产品的抽象类，定义对象的外部、内部状态的接口或实现；

ConcreteFlyweight：具体享元角色，需要注意的是内部状态应该和环境无关，不应该出现一个操作改变了内部状态也改变了外部状态；

UnsharedConcreteFlyweight：不可共享的享元角色，不存在外部状态或者安全要求（如线程安全）不能够使用共享技术的对象，一般不出现在享元工厂中；

FlyweightFactory：享元工厂，构造一个池容器，并提供从池中获取对象的方法。

注意：

线程安全：使用的地方多考虑线程安全问题；尽可能提供足够多的共享对象；

性能平衡：尽量使用Java基本类型做为外部状态（key）（自定义对象作为key需要重写equals和hashCode方法，但对象比较毕竟没有基本类型比较速度快）。

1. 桥梁模式（Bridge Pattern）

将抽象和实现解耦，使得两者可以独立的变化（抽象角色的部分实现由实现角色完成）。

角色：

Abstraction：抽象化角色，主要职责是定义角色行为，同时保存一个对实现化角色的引用，一般是抽象类；

Implementor：实现化角色，普通接口或抽象类，定义角色的属性和行为；

RefinedAbstraction：修正抽象化角色，引用实现化角色对抽象化角色进行修正；

ConcreteImplementor：具体实现化角色。

使用场景：

不希望或不适用使用继承；

接口或抽象类不稳定；

重用性要求较高。