七个结构型模式：适配器模式（Adapter Pattern），桥接模式（Bridge），组合模型（Composite）
装饰模式（Decorator），外观模式（Facade），享元模式（Flyweight），代理模式（Proxy）

①适配器模式（Adapter）：为了协调不兼容结构。就比如：耳机的插头是圆柱形，而手机口是矩形，这时需要一个适配器协调。

定义：将一个不匹配的接口转换成客户端希望的另一个接口，使接口不兼容的类可以一起工作。

三类结构：客户端需要的接口（target）［它希望调用此接口可以直接操作另一不兼容接口］

Adapter：适配器类（实现target，并维持被适配者类的引用，通过这些引用调用方法，放在target声明的方法中）。

Adaptee：适配者类（有自己的方法，这些正是客户端需要的）

以上是 对象适配器模式：在适配器中维持一个对象的引用。（关联关系）通过引用调用方法与客户端接口相适应。

类适配器模式：实现target和继承被适配者类。就可以用这种方式了

```
public void getSort(){    → 客户端接口方法。
    sort();               → 适配者方法
}
```

但这种模式耦合度大，而且需要多继承，假如target不是接口，则无用了。（对于Java这种单继承来说）。

双适配器模式：在适配器中维持两个引用，一个target，一个Adaptee。实现两个接口，对它们进行适配。

思考：为什么不直接在客户端中维持Adaptee引用或者new Adaptee()呢？然后调用方法。

①假如Adaptee发生改变，每个客户端的Adaptee都要修改。耦合度高。而用Adapter，只需要改这一个类就可以了。

②假如客户端需要多个Adaptee，客户端一个个new不太实际。可直接在Adapter中维持多个引用。

③增加了类的透明性和复用性。灵活性和扩展性非常好。

②桥接模式（Bridge）：处理多维度变化。例如：一支笔有不同粗细，颜色和型号。若笔有5种颜色，3种型号，需要开发所有种类对象：

两种维度耦合：5×3=15个 对象

两种维度分开：5+3=8个 对象

定义：将抽象部分和实现部分进行分离，它们可以独立变化。

需要给每个维度的抽象。比如：A抽象，B抽象

A抽象的实体类（说明这实体实现了一种性质），包有一个B抽象的引用。通过setB(B b)增加另一个维

this.b=b;

增加后这个A的实现类就是我们需要的类。

③组合模式：用于树形结构的处理。比如文件系统中，一个文件可能是文件，也可能是文件夹，怎样才能做到对它们统一编程？

就比如对文件夹扫描，怎么才能不区别对待文件和文件夹。

定义：由于容器对象和叶子对象在功能上有区别，所以我们希望能统一使用。

结构：抽象构件类：它是所有容器类和叶子类的父类（核心）（将所有类共有的方法声明和定义在抽象类）

Composite 容器构件类：它继承自抽象构件，实现了管理叶子构件的方法（如：add、remove等），和递归遍历子节点进行特定操作的方法。

Leaf 叶子构件类：它也继承抽象构件，实现了特定操作。但对于管理方法（继承来的），它只能通过异常等方式处理。（因为它没有）

以上的为透明组合模式：抽象构件类声明了所有类的方法。包括 add()、remove()。

好处：使所有构件类拥有相同的接口，便于统一处理。缺点：叶子构件类对于管理方法必须抛出异常，否则出问题。

安全组合模式：抽象构件类，不声明容器构件类的管理方法。这样的叶子构件类也不需要声明那些管理方法，很安全。缺点：容器构件类需要单独声明方法，不利于统一编程。

④装饰模式：在系统原有基础上增加新功能。（传统思维，进行继承，但这样会增加很多类）。

定义：它是一种用于替代继承的技术，它通过一种无须定义子类的方式来给对象动态的增加职责。（使用关联关系取代类的继承关系）。

(Component)结构：抽象构件类：所有构件都实现这个抽象操作方法。operation();

具体构件类：实现抽象构件，重写操作方法。operation();

(Decorator) 抽象装饰类：实现抽象构件，并维持一个抽象构件引用。重写操作方法，但根据里面是通过维持的引用调用方法。

具体装饰类：在抽象类的基上，增加装饰方法。这样就在原有基础上增加了新的方法。

如：public add(){ → 增加方法
  ......
}

public operation(){
  super.operation();
  add();
}

如：public void operation(){
  component.operation(); ← 维持的引用。
}

以上的为透明装饰模式：将所有新增的方法都加入到operation，那么具体构件类可以调用operation方法，得到装饰，满足需求。

好处：1. 无需关注装饰前和装饰后对象的区别，对用户是透明的。

2. 可以进行多次修饰。

半透明装饰：具体装饰类的add()方法，并未加入到operation()，那么调用add方法，必须声明具体装饰类的引用调用，而不像上面只需抽象构件引用就可以调用operation中的add()方法。

⑤外观模式引入了外观类充当了软件系统中的"服务员"，它为多个业务类的调用提供了统一的接口。
如果没有外观类，那么每个客户类需要和多个子系统进行交互，耦合度很大。
定义：为子系统中的一组接口提供统一的入口，这个接口使得这一子系统更加容易使用。
结构：Facade(外观类)：客户端通过调用它完成一系列调用子系统的方法的封装。
维护了各个子系统的引用。
subsystem(子系统类)：完成特定的功能，可以有多个子系统。(在外观类有它们的引用)
使用：比如3个子系统A,B,C(有a(),b(),c())方法。Facade类写一方法调用完成某种功能：method(){ A.a(); C.c(); B.b(); } 可自定义调用顺序完成一系列操作。
[redacted]如果只定义外观类，明显违背了开闭原则。如果要修改里面子系统引用，这是不允许的。
为此：将外观类抽象出来，让具体继承抽象类。这样如果改需求，新建一个类就行了。
优点：①屏蔽了子系统组件，减少了客户端所需处理的对象，简单使用。
②子系统的改变不会影响到调用子系统。

⑥享元模式：实现对象的复用。例如：五子棋如果每一个棋子都new一个对象，那么无疑很耗费内存的。
定义：运用共享技术有效的支持大量细粒度对象的复用。
能做到共享的原因：它区分了内部状态和外部状态。
内部状态：是存储在享元对象内部并且不会随着环境改变而改变。如：棋
外部状态：随着环境会发生变化不可以共享的状态。如：五子棋坐标，
享元池：可以将相同内部状态的对象存储在享元池中，享元池中的对象是可以共享的。
使用：通常搭配工厂类使用。
①享元工厂类(享元池)：通过一个Hash table充当享元池，如果hashtable中有对象，则直接取出，
如果没有则存入。

```
getInstance(key){
    if(hashtable.contains(key))
        return hashtable.get(key);
```

②通过工厂类来得到享元对象[redacted](棋)存在服务端
(所有使用)
③那要记入坐标怎么办？只需将参数或者对象作为参数传入就可以了。
但是这些参数并不会保存在这个对象中，这个对象也不会存传入对象的引用。
(如果存了，那就不一样了吗)
作用：实现对象的复用。

⑦代理模式(Proxy)：当无法直接访问某个对象时，可以通过一个代理对象来间接访问，它们实现相同接口。

定义：为某一个对象提供一个代理，并由代理对象对原对象的访问。

结构：①抽象对象接口，有实现类的实现类target类。

②实现了接口的代理类，并且维持了target类的的引用。

③重写的方法中，使用target的方法，并可在该方法前后加上处理。

以上是典型的静态代理。

动态代理：1、JDK动态代理：通过调用Proxy类的静态方法，newProxyInstance返回代理对象。

```
Proxy = Proxy.newProxyInstance() 需要传入三个参数。
    ① ClassLoader loader 类加载器 真实对象.getClass().getClassLoader();
    ② Class<?>[] interfaces 目标对象实现的接口：对象.getClass().getInterfaces()
    ③ InvocationHandler h: 需要传入一个实现类，再调用的内部类
        new InVocationHandler(){
            public object invoke(Object proxy, Method method, Object[] args){
                system.out.println("开始");
                object target = method.invoke(target, args);
                system.out.println("结束");
                return target;
            }
        }
```

Proxy.方法();

2、Cglib代理（需要实现代理对象接口）

十一种行为模式：职责链模式，命令模式，解释器模式，迭代器模式，中介者模式，备忘录模式，观察者模式，状态模式，策略模式，模板方法模式，访问者模式

①职责链模式：引入：当一个请求任务不满足时，选择进入下一次判断，满足选择执行该方法，不满足继续判断。
(chain of Responsibility) 这样有缺点 ①要太多if.else语句，多个方法都聚集在同一个类中。
②不能随意更改if.else的顺序。

核心：引入了一个抽象处理者。所有处理者都继承抽象类，并都持有一个处理者的引用。

例如：new BufferOutputStream(new FileOutputStream());
传入一个处理者的引用。调用方法时，如果满足条件，就自己处理，不满足传给下一个引用处理。

1.纯的责任链模式：遇到请求要么处理，要么不处理直接发下一个。不存在处理到一半就转发的。

2.不纯的责任链模式：处理一部分，然后传给下一个。

②命令模式：核心：引入命令类，通过命令类来降低发送者和接收者的耦合度，请求发送者只需指定一个命令对象
(command) 再通过命令对象来调用来请求接收者的处理方法。

结构：抽象命令类，具体命令类（必持有一个接收者的引用，执行处理发送者的请求）
发送者类（必持有一个命令类的引用，通过注入的命令类和接收者交互。
接收者类（完成某个处理请求，被命令类调用执行方法）

思考：当一个请求有多个接收类怎么处理？① 在命令类中持有一个接收者List的引用并执行，当收到请求，遍历执行接收者方法。
作用：将请求发送者与接收者解耦。很容易的增加新的命令。

③解释器模式：定义一个语言的文法，并且建立一个解释器来解释该语言中的句子。
(Interpreter) （相当于一个语言解释器）

④迭代器模式：聚合对象：里面包含许多对象。比如：List。但是遍历的职责不应该加到它的身上。
(Iterator) 因此专门用一个迭代器进行遍历。

定义：[struck] 由迭代器来提供遍历聚合对象，而不用暴露这个。通过游标达成目的。

基本结构：提供基本的遍历方法和cursor（游标，确定位置）

一般的集合类内部都实现了Iterator。因为Collection extends Iterator。

⑤ Mediator Pattern：可以使对象之间的关系数量减少，通过引入中介者对象，可以将网状结构改造成以中介者为中心的中心结构。

定义：用一个中介对象来封装一系列的交互，使得各对象不需要显式地相互引用，从而模式松耦合状态。

两个作用：① 中转作用：各个对象不需要显示调用其它对象。② 协调作用：对各对象间关系进一步封装。（封装多对对象的行为）

缺点：因为把封装了对象的行为，那这个中介者会变得非常复杂。

结构：① 抽象中介者类。② 在具体中介者中维持各个同事类的引用，并在方法中完成各个同事类之间的交互。

③ 同事类：即那些普通的对象。（需要进行交互的对象）

⑥ Memento Pattern：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后可以将对象恢复到原先保存的状态。

结构：① 原发器类：它是一个普通类，可以创建一个备忘录，但是并不维持备忘录引用。（只保留有价值的属性）（例如：棋子的X,Y属性）

② 备忘录类：存储原发器类的状态。（以类似一样描述的属性）。

③ 负责人：将原发器类产生的备忘录注入，并维持备忘录类引用。（若想要恢复多步，维持一个LIS集合）

⑦ observer Pattern：定义对象之间的一种一对多的依赖关系，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。

结构：① 目标：它包含经常发生改变的数据，当它状态改变时，向他的各个观察者发送通知（增加和删除通知观察者的方法）

② 观察者：维护一个指向具体目标对象的引用，当目标对象出现了变化，可以通过目标对象中的方法，对其它观察者作出通知。

注意：MVC也采用了观察者模式。当model发生改变，view也会发生改变。

⑧ State Pattern：定义：~~允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。~~

允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。

结构：环在类：维持一个抽象状态类的引用，和引发状态变化的属性。可以通过调用这个属性，调用 setState() 向环在类设置不同的状态对象。（实际环在类才是我们真正使用的对象，状态类只是辅助）

状态类：每个子类实现一个与环在类的一个状态相关的行为。

主要还是环在类，通过调用环在类的状态来调用方法。