周哥教 IT.设计模式精讲

视频课地址：https://ke.qq.com/course/3648042?tuin=a71606

目录

设计模式简介

设计模式（Design Pattern）是程序员前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。
1995 年，GoF（Gang of Four，四人组/四人帮）合作出版了《设计模式：可复用面向对象软件的基础》一书，共收录了 23 种设计模式，从此树立了软件设计模式领域的里程碑，人称「GoF 设计模式」。 这 23 种设计模式可以分为三大类：创建型模式（ Creational Patterns ）、结构型模式（ Structural Patterns ）和行为型模式（ Behavioral Patterns ）。

学习设计模式建议

设计模式对于提高代码设计质量很重要，但又非常难学，原因很多。一是设计模式比较多，而且很抽象，加上实际工作中应用较少，所以学习效果往往并不是很好。;二是目前市面上相关课程面面俱到，抓不住重点;三是小项目用不着设计模式，不会设计模式照样开发程序。
学习设计模式还是需要结合实际应用才会有更深的理解，而工作中用到各种各样的设计模式的场景毕竟不是很多，所以结合一些源码中对设计模式应用的例子来学习是一种折衷但不失为效果较好的方式，比如 java 等开源项目里面用到很多设计模式。
不用担心，不会设计模式，不影响编程写代码，不影响当程序员
没写过 5 万行代码谈设计模式都是在瞎扯，没有设计模式，软件照样开发，就是在大型软件系统开发及维护过程中就痛苦不堪，最后在不断重构后你会发现尼玛竟然用了好多设计模式。
问题，原理，实现，去繁从简，弃虚务实

课程大纲

创建型模式：5 个
单例模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式
结构型模式：7 个
适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式
行为型模式：11 个
策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式
讲解角度：问题与应用需求+原理分析+编码与实现+优缺点对比分析
创建型模式

什么是创建型模式？

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。这样可以降低系统的耦合度，使用者不需要关注对象的创建细节，对象的创建由相关的工厂来完成，就像我们去商场购买商品，不需要知道商品怎么生产出来的，因为它们由专门的厂商生产。
创建型模式分为以下几种。
单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点提供外部获取该实例，其拓展是有限多例模式。
原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
工厂方法（FactoryMethod）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。
抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。
以上 5 种创建型模式，除了工厂方法模式属于类创建型模式，其他的全部属于对象创建型模式

单例模式 1：问题与需求

Singleton Pattern
问题：为了某些应用场景，如何只生成类的一个对象实例？
A *p1 = new A;

A *p2 = new A;
A a;
单例模式的适用场景:
当某类需要频繁实例化，而创建的对象又频繁被销毁的时候。
创建对象耗时过多或者耗资源过多,但又经常用到的对象
频繁访问数据库或文件的对象
以及其他要求只有一个对象的场景
当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

单例模式 2：原理分析

保证系统中类只有一个实例,并提供一个访问它的全局访问点,该实例被所有程序模块共享。对于系统中的某些类来说,只有一个实例很重要,比如一个系统只能有一个窗口管理器或文件系统。
单例模式的要点有三个:
1. 单例类只能有一个实例
2. 它必须自行创建这个实例
3. 它必须自行向整个系统提供这个实例。
从具体实现角度来说就是以下三点:
1. 单例模式的类只提供私有的构造函数
2. 类定义中含有一个该类的静态私有对象
3. 该类提供了一个静态的公有的函数用于创建或获取它本身的静态私有对象
单例模式又分为饿汉式单例和懒汉式单例,饿汉式单例在单例类被加载时就实例化一个对象;而懒汉式在调用取得实例方法的时候才会实例化对象。
在 C++ 中一般都使用懒汉式单例。

```
//C++
class CSingleton{
private:
	CSingleton() {//构造函数是私有的
	}
	static CSingleton *m_pInstance;
public:
	static CSingleton * GetInstance(){
		if(m_pInstance == NULL) //判断是否第一次调用
			m_pInstance = new CSingleton();
		return m_pInstance;
	}
};
```

```
//java
class Singleton{
	//私有的构造函数，保证外类不能实例化本类
	private Singleton(){}
	//自己创建一个类的实例化
	private static Singleton singleton = new Singleton();
	//创建一个 get 方法，返回一个实例 s
	public static Singleton getInstance(){
		return singleton;
	}
}
```

java.lang.Runtime.getRuntime()
java.awt.Desktop.getDesktop()
java.lang.System.getSecurityManager()

单例模式 3：编码实现

单例模式 4：优点

单例模式的优点:
在内存中只有一个对象,节省内存空间
避免频繁的创建销毁对象,可以提高性能
避免对共享资源的多重占用
可以全局访问

工厂设计模式

简单工厂（不属于 23 种之一）

根据条件判断需要产生什么样子的类；新增产品需要新增产品类，修改工厂类。扩展的时候需要修改工厂类，不符合开闭原则（允许对代码进行扩展，但是不允许对原有的代码进行修改，别改代码，只需要添代码）
工厂方法
不同的产品使用不同的工厂实例；新增产品需要新增产品类和工厂类。符合开闭原则，系统易于扩展但是复杂
抽象工厂

组合生产产品。符合开闭原则，新增产品需要修改接口

简单工厂 1：问题与需求

simple factory
通过专门的工厂来生产各种产品（对象），实现生产与使用的分离。
一台咖啡机就可以理解为一个工厂模式，你只需要按下想喝的咖啡品类的按钮（摩卡或拿铁），它就会给你生产一杯相应的咖啡，你不需要管它内部的具体实现，只要告诉它你的需求即可。

简单工厂 2：原理分析

简单工厂模式又叫静态工厂方法模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。
根据条件判断需要产生什么样子的类；扩展的时候新增产品需要新增产品类，修改工厂类。不符合开闭原则

```
class Food {
}
//小饼干
class Cookie:Food{
}

//三明治
class Sandwich:Food{
}
class Factory {
public:
  Food* getFood(string foodName){
            Food *food=nullptr;
            if (foodName=="cookie"){
                        food = new Cookie();
              } else if (foodName=="sandwich"){
                        food= new Sandwich();
              } else {
              }
            return food;
      }
}
```

简单工厂 3：编码与实现

简单工厂 4：优缺点总结

工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象；
不易拓展，一旦添加新的产品类型，就不得不修改工厂的创建逻辑；
产品类型较多时，工厂的创建逻辑可能过于复杂，一旦出错可能造成所有产品的创建失败，不利于系统的维护

工厂方法 1：问题与需求

factory method
解决简单工厂中的问题。不同的产品使用不同的工厂实例；新增产品需要新增产品类和工厂类。符合开闭原则

工厂方法 2：原理分析

不同的产品使用不同的工厂实例；新增产品需要新增产品类和工厂类。符合开闭原则，系统易于扩展

```
//基础
 class Food {
 public:
 void sale(){
   }
}
//小饼干
class Cookie:Food{
}
//三明治
class Sandwich:Food{
}
//基础工厂
class Factory {
      virtual Food* createFood();
}
//小饼干的工厂
```

```
class CookieFactory: Factory{
  public:
  Food* createFood() {
            return new Cookie();
     }
}
//三明治的工厂
class SandwichFactory: Factory{
public:
  Food* createFood() {
            return new Sandwich();
      }
}
```

工厂方法 3：编码与实现

工厂方法 4：优缺点

符合开闭原则
系统易于扩展
但是复杂

抽象工厂 1：问题与需求

（Abstract Factory）
工厂方法模式，一个工厂只能生产一个类型的商品。而现实中，一个工厂，可能会生产多个系列产品（产品族），比如对于键盘和鼠标，按照品牌分为微软键盘、鼠标（微软产品族）以及华为的键盘、鼠标（华为产品族）。
如果用工厂方法模式，就需要创建 4 个工厂。
用抽象工厂模式，可以专门定义一个工厂，同时生产键盘和鼠标。
抽象工厂 2：原理分析

```
class KeyBoard{
public:
      virtual void show() = 0;
};
//  微软的键盘
class KeyBoardMicro : public KeyBoard{
public:
```

```
    void show(){
        std::cout << "微软的键盘" << std::endl;
    }
};
// 华为的键盘
class KeyBoardHuawei: public KeyBoard{
public:
    void show(){
        std::cout << "华为的键盘" << std::endl;
    }
};
// 鼠标
class Mouse{
public:
    virtual void show() = 0;
};

class MouseMicro : public Mouse{
public:
    void show(){
        std::cout << "微软的鼠标" << std::endl;
    }
};

class MouseHuawei : public Mouse{
public:
    void show(){
        std::cout << "华为的鼠标" << std::endl;
    }
};
// 工厂
class Factory{
public:
    virtual KeyBoard * createKeyBoard() = 0;
    virtual Mouse * createMouse() = 0;
};

// 微软的工厂
class FactoryMicro : public Factory{
public:
    KeyBoard * createKeyBoard(){
```

```
        return new KeyBoardMicro();
    }
    Mouse * createMouse(){
        return new MouseMicro();
    }
};

// 华为的工厂
class FactoryHuawei : public Factory{
public:
    KeyBoard * createKeyBoard(){
        return new KeyBoardHuawei();
    }
    Mouse * createMouse(){
        return new MouseHuawei();
    }
};
```

抽象工厂 3：编码与实现

抽象工厂 4：比较分析，优缺点

抽象工厂模式和工厂方法模式很相似。最大的区别就是抽象工厂模式不止一个产品族，并且每个工厂都不止生产一种产品。
在工厂方法模式中，不同的工厂可以生产不同的产品。比如键盘、鼠标。分别由键盘工厂和鼠标工厂，但是只能生产一种键盘和鼠标。但是在抽象工厂模式中，每个工厂可以生产键盘和鼠标。还可以生产不同牌子的键盘和鼠标。
优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。
缺点：产品族扩展非常困难，要增加一个系列的某一产品（鼠标，键盘，音箱），既要在抽象的 工厂里加代码，又要在具体的工厂里面加代码。

建造者模式 1：问题与需求

（Builder Pattern）
问题：对于一些复杂的对象的创建，这些对象内部的建造顺序通常是稳定的，但对象内部每一部分的构建面临着复杂的变化。这样的情况下，如何减少因为对构建过程的步骤缺失或者颠倒，最后出错呢？
需要将对象内部构建过程固定下来。

建造者模式的本质和建造楼房是一致的：即流程不变，但每个流程实现的具体细节则是经常变化的。建造者模式的好处就是保证了流程不会变化，流程既不会增加、也不会遗漏或者产生流程次序错误

建造者模式 2：原理分析

建造者模式（Builder），将一个复杂的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

通过建造者模式，使得建造过程通过 Director 类的 Construct 函数固定了，即建造过程不会变，“必须包括”。但是具体的头，身体，手脚这些身体的各个部分会变化，基类 Builder 中将各种 Build 函数定义为抽象方法，必须在子类中实现。这样不仅仅使得建造的过程不变，而且很利于系统的扩展，一旦出现其他种类的根本不需要改动之前的类，只需要新添加新的类。符合 OCP 原则。

区别：建造者模式注重零部件的组装过程和顺序，而工厂方法模式更注重整个产品的创建过程。

```
class Product{
	vector<string> parts;
public:
	void Add(const string part){
		parts.push_back(part);
	}
	void Show()const{
		for (int i = 0; i < parts.size(); i++){
			cout << parts[i] << endl;
		}
	}
};

XProduct
//抽象 builder 类
class Builder{
public:
	virtual void BuildHead() = 0;
	virtual void BuildBody() = 0;
	virtual void BuildHand() = 0;
	virtual void BuildFeet() = 0;
	virtual Product GetResult() = 0;
};

XBuilder
class Director{
```

```
public:
    void Construct(Builder &builder) {
        builder.BuildHead();
        builder.BuildBody();
        builder.BuildHand();
        builder.BuildFeet();
    }
};
```

建造者模式 3：编码与实现

原型模式 1：问题与需求

（Prototype Pattern）
问题：航空智能管理系统。飞机有名称和型号和厂商。厂商有名称，地址和负责人。负责人有姓名和年龄。系统中已经有一架飞机，现在要一架相同的飞机，如何快速创建这样的对象？
原型模式用于创建重复的对象，实现对象的拷贝。复制当前对象，实现对象数据的克隆。
应用：一个复杂的对象，包含多种数据和结构，层次较深时，适用于原型模式；对象之间相同或相似，即只是个别的几个属性不同的时候。对象的创建过程比较麻烦，但复制比较简单的时候

原型模式 2：原理分析

原型模式复制当前对象，实现对象数据的克隆。原型模式的克隆分为浅拷贝和深拷贝。孙悟空拔猴毛变多个孙悟空出来（一块石头吸收天地灵气日月精华才蹦出一个孙悟空，但是通过复制，也就是拔猴毛变孙悟空就快多了，这就是原型模式的意义）。一个孙悟空变成两个孙悟空，确实是两个不同的对象（一个从石头里蹦出来，一个是猴毛变的，产地不一样所以对比是 false），他们的外貌（就是基本数据类型的成员变量），浅拷贝可直接复制一份，所以两个孙悟空的外貌一模一样。但是他们的灵魂（就是引用数据类型的成员变量）还是同一个孙悟空的，也就是同一个打妖怪保护师傅的灵魂。如果你想把灵魂都复制一份，造出一个六耳猕猴，你就得用深拷贝进行复制，就可以得到一个和孙悟空完全不同的六耳猕猴的灵魂，但他们的外貌还是一模一样。
手机由名称,价格，生产工厂组成。生产工厂由工程师和名称组成。工程师由姓名这个基本属性。现在为了快速复制同一工厂的不同手机（华为和荣耀），解决这一实际应用场景。

```
class Prototype{
private:
    string str;
public:
    Prototype(string s){
```

```
        str = s;
    }
    Prototype(){
        str = "";
    }
    void show(){
        cout << str << endl;
    }
    virtual Prototype *clone() = 0;
};
```

原型模式 3：编码与实现

结构型模式

什么是结构型模式？
结构型模式（Structural Pattern）关注如何将现有类或对象组织在一起形成更加强大的结构。
可分为两种：
类结构型模式：关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系
对象结构型模式：关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。更符合“合成复用原则”
适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式

适配器模式 1：问题与需求

Adapter Pattern
接口转换器，电源适配器，USB 串口转换器等
问题：系统需要使用一些现有的类，而这些类的接口（如方法名）不符合系统的需要（不兼容），甚至没有这些类的源代码。想创建一个可以重复使用的类(adapter)，用于与一些彼此之间没有太大关联的一些类(target, adaptee)，包括一些可能在将来引进的类一起工作。

适配器模式 2：原理分析

把一个类的接口变成客户端所期待的另一种接口，从而使原本接口不匹配而无法一起工作的两个类能够在一起工作
(1)类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的两个类，实现新的接口即可。
(2)对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一

个包装类，持有原类的一个实例，在包装类的方法中，调用实例的方法就行。

```
// Targets
class Target{

public:
    virtual void Request(){
        cout<<"Target::Request"<<endl;
    }
};

// Adaptee

class Adaptee{
public:
    void SpecificRequest(){

        cout<<"Adaptee::SpecificRequest"<<endl;

    }

};

// Adapter
class Adapter : public Target, Adaptee{

public:
    void Request(){
        Adaptee::SpecificRequest();
    }
};
class Target{
public:
    Target(){}
    virtual ~Target(){}
    virtual void Request(){

        cout<<"Target::Request"<<endl;
    }
};
class Adaptee{
public:
```

```
    void SpecificRequest(){
        cout<<"Adaptee::SpecificRequest"<<endl;
    }
};
class Adapter : public Target{
public:
    Adapter() : m_Adaptee(new Adaptee) {}
    ~Adapter(){
        if (m_Adaptee != NULL){
            delete m_Adaptee;
            m_Adaptee = NULL;
        }
    }
    void Request(){
        m_Adaptee->SpecificRequest();
    }
private:
    Adaptee *m_Adaptee;

};
```

Java IO 的例子中,target 接口就对应了 Reader 抽象类, adaptee 对应到我们的 InputStream, InputStreamReader 就是适配器 Adapter

```
File file = new File("D:/dp/test/1.txt");
InputStreamReader reader = new InputStreamReader(new FileInputStream(file), "GBK");
```

适配器模式 3：编码与实现

装饰器模式 1：问题与需求

（Decorator Pattern）
问题：比如说一个 Person 类，该类的操作有能吃、能睡、能跑、但假如随着人类的进化，某一天 Person 能飞了，能在水里游了等等，按照一般的写法是修改 Person 这个类，给这个类添加上能飞，能游等操作，但是这样破坏了面向对象的开放-封闭原则,且随着人类的进化这个类就会变得越来越臃肿，越来越复杂，添加任何一个功能都必须对这个臃肿的类进行修改，出错的概率大大提升，且容易影响老功能，而装饰器模式可以解决这类问题，装饰器从外部给类添加职能（额外的功能），而不用去修改原始的类，拓展性好，可复用程度高。
装饰器的使用场景
用于拓展一个类的功能或者给一个类添加附加职责

装饰器模式 2：原理分析

装饰器模式允许向一个现有的类添加新的功能，同时又不改变其结构，它是作为现有的类的一个包装，创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下进行扩展，提供了额外的功能。

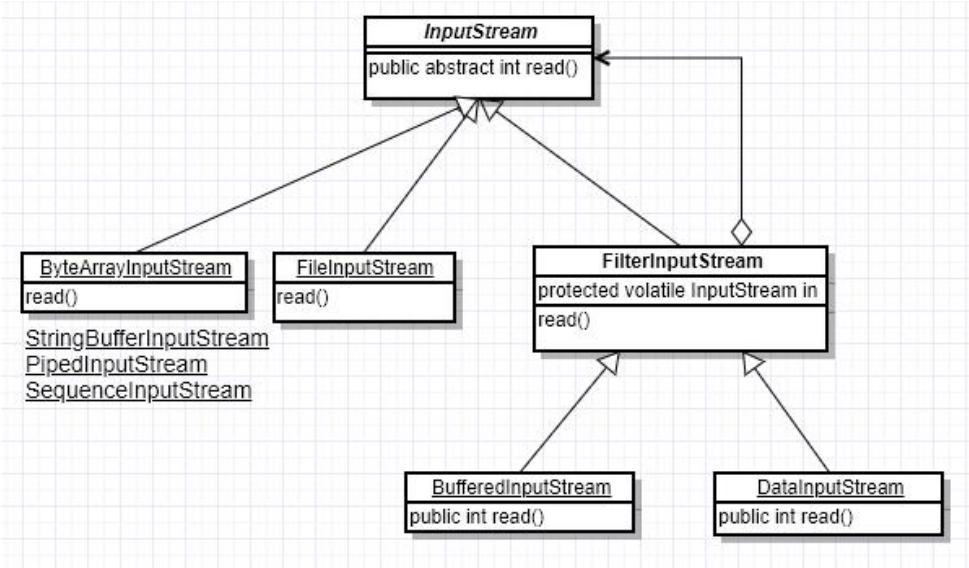
蛋糕店刚生产出来的蛋糕是最原始的，只是一个蛋糕原型，我们需要在这个蛋糕上加上奶油，加上巧克力，加上瓜子仁，写上字等等，将原始蛋糕作为一个 Cake 类，我们给这个 Cake 类的对象作装饰，为了不破坏开放-封闭原则，也为了更好的拓展，我们不能直接在 Cake 这个类里修改，而应该做一个装饰器，通过装饰器来给这个蛋糕做装饰。

设计一个几何形状库，基类是 Shape，提供绘制几何形状的虚拟 Draw 方法，圆(Circle)和长方形(Rectangel)继承于 Shape，实现各自的 Draw 功能。现在我们需要对几何形状进行装饰，在绘制的过程中，加上边框的颜色。

与适配器对比区别：适配器模式是在适配器中，重写旧接口的方法来调用新接口方法，来实现旧接口不改变，同时使用新接口的目的，新接口适配旧接口。而装饰模式，是装饰器和旧接口实现相同的接口，在调用新接口的方法中，会调用旧接口的方法，并对其进行扩展

```
Shape
->virtual Draw();

Circle:Shape
->Draw();

Rectangle:Shape
->Draw();


ShapeDecorator:Shape
->Draw()
->Shape *m_pShape

RedShapeDecorator:ShapeDecorator
->Draw(){//新接口与旧接口相同
      m_pShape->Draw();//旧接口
      SetBorder(red);//扩展
}
```

Java IO 中的装饰器模式



装饰器模式 3：编码与实现

代理模式 1：问题与需求

（Proxy Pattern）
在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。一个类代表另一个类的功能。在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。
何时使用：想在访问一个类时做一些控制

代理模式 2：原理分析

在访问此对象时加上一个对此对象的访问层。一个类代表另一个类的功能。在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。
1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。
2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。
auto_ptr 类就是一个代理，客户只需操作 auto_ptr 的对象，而不需要与被代理的指针 pointee 打交道

```
class Proxy:public Subject{
public:
    Proxy();
    Proxy(Subject* _sub);
```

```
    void Request(){//实现对委托者的委托任务执行与补偿
        bef();
        this->_sub->Request();
        end();
    }

    void bef(){}
    void end(){}
    ~Proxy();

private:
    Subject* _sub;

};
```

代理模式 3：编码与实现

外观模式 1：问题与需求

（Facade pattern）
问题：假设一台电脑 Computer，它包含了 CPU（处理器），Memory（内存），Disk（硬盘）这几个部件，若想要启动电脑，则先后必须启动 CPU、Memory、Disk。关闭也是如此。但是实际上我们在电脑开/关机时根本不需要去操作这些组件，因为电脑已经帮我们都处理好了，并隐藏了这些东西。这些组件好比子系统角色，而电脑就是一个外观角色。
适用场景：
当要为访问一系列复杂的子系统提供一个简单入口时可以使用外观模式。
客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性。

外观模式 2：原理分析

把不同类的不同接口，统一代理到一个类里面对外输出，使代码具有良好的封装性。引入一个外观角色（外观类）来简化客户端与子系统之间的交互，为复杂的子系统调用提供一个统一的入口，降低子系统与客户端的耦合度，且客户端调用非常方便。
外观模式包含如下两个角色：
Facade（外观角色）：在客户端可以调用它的方法，在外观角色中可以知道相关的（一个或者多个）子系统的功能和责任；在正常情况下，它将所有从客户端发来的请求委派到相应的子系统去，传递给相应的子系统对象处理。
SubSystem（子系统角色）：在软件系统中可以有一个或者多个子系统角色，每一个子系统可

以不是一个单独的类，而是一个类的集合，它实现子系统的功能；每一个子系统都可以被客户端直接调用，或者被外观角色调用，它处理由外观类传过来的请求；子系统并不知道外观的存在，对于子系统而言，外观角色仅仅是另外一个客户端而已。
外观模式的目的不是给予子系统添加新的功能接口，而是为了让外部减少与子系统内多个模块的交互，松散耦合，从而让外部能够更简单地使用子系统。外观模式的本质是：封装交互，简化调用。

```
class Facade{
public:
	void Compile(){
		CSyntaxParser syntaxParser;
		CGenMidCode genMidCode;
		CGenAssemblyCode genAssemblyCode;
		CLinkSystem linkSystem;

		syntaxParser.SyntaxParser();
		genMidCode.GenMidCode();
		genAssemblyCode.GenAssemblyCode();
		linkSystem.LinkSystem();
	}
};

// 客户端
int main(){
	Facade facade;
	facade.Compile();
}
```

外观模式 3：编码与实现

桥接模式 1：问题与需求

（Bridge pattern）
在项目开发中，我们会遇到这样的一种场景：某些类型由于自身的逻辑，往往具有两个或多个维度的变化，比如说手机，它有两个变化的维度：一是手机的品牌，可能有华为，小米、苹果等；二是手机上的软件，操作系统有可能是 ios（子系统），android（子系统），应用软件可能有 QQ、微信等。如何应对这种“多维度的变化”？怎样利用面向对象的技术来使得该类型能够轻松的沿着多个方向进行变化，而又不引入额外的复杂度？这就是桥接模式所要解决的问题。
同样，汽车在路上行驶的来说。既有小汽车又有公共汽车，它们都不但能在市区中的公路上行驶，也能在高速公路上行驶。这你会发现，对于交通工具（汽车）有不同的类型，然而它

们所行驶的环境（路）也在变化，在软件系统中就要适应两个方面的变化？怎样实现才能应对这种变化呢？

桥接模式 2：原理分析

一个类存在两个（或多个）独立变化的维度，且这两个（或多个）维度都需要独立进行扩展。对于那些不希望使用继承或因为多层继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。
“多个维度的实现部分分离，使它们都可以独立地变化”
手机品牌与手机软件为例，我们可以让手机既可以按照手机品牌来分类，也可以按手机软件来分类。由于实现的方式有多种，桥接模式的核心意图就是把这些实现独立出来，让它们各自地变化，这就使得每种实现的变化不会影响其他实现，从而达到应对变化的目的。
现在如果想要增加一个功能，比如音乐播放器，那么只有增加这个类就可以了，不会影响到其他任何类，类的个数增加也只是一个；如果是要增加 S 品牌，只需要增加一个品牌的子类就可以了，个数也是一个，不会影响到其他类。这显然符合开放-封闭原则。
而这里用到的合成/聚合复用原则是一个很有用处的原则，即优先使用对象的合成或聚合，而不是继承。究其原因是因为继承是一种强耦合的结构，父类变，子类就必须变。
适用于有两个维度以上的变化，并且都需要扩展时。

```
class OS{
public:
	virtual std::string GetOS() = 0;
};
class IOS : public OS{
public:
	virtual std::string GetOS() {
		return "IOS Operator System";
	}
};
class IOSSubSystem1 : public IOS{
public:
	virtual std::string GetOS() {
		return "IOS 5.1.1 Operator System";
	}
};
class IOSSubSystem2 : public IOS{
public:
	virtual std::string GetOS() {
		return "IOS 10.1.1 Operator System";
	}
};
```

```
class iPhone : public Phone{
public:
	iPhone(OS* os){
		m_pOS = os;
	}
	~iPhone(){
		DELETE_OBJECT(m_pOS);
	}
	virtual void SetOS(){
		cout << "Set The OS: " << m_pOS->GetOS().c_str() << endl;
	}
private:
	OS* m_pOS;//合成聚合
};

	OS* pIOS1 = new IOSSubSystem1(); //创建一个操作系统
	Phone* iPhone4 = new iPhone(pIOS1);//应用到该手机上
	iPhone4->SetOS();

	OS* pIOS2 = new IOSSubSystem2();//创建一个操作系统
	Phone* iPhone6 = new iPhone(pIOS2);//应用到该手机上
	iPhone6->SetOS();

	OS* pAndorid1 = new AndoridSubSystem1();//创建一个操作系统
	Phone* Xiaomi1 = new Xiaomi(pAndroid1);//应用到该手机上
	Xiaomi1->SetOS();

	OS* pAndroid2 = new AndoridSubSystem2();//创建一个操作系统
	Phone* Xiaomi2 = new Xiaomi(pAndroid2);//应用到该手机上
	Xiaomi1->SetOS();
```

桥接模式 3：编码与实现

组合模式 1：问题与需求

组合模式（Composite Pattern）

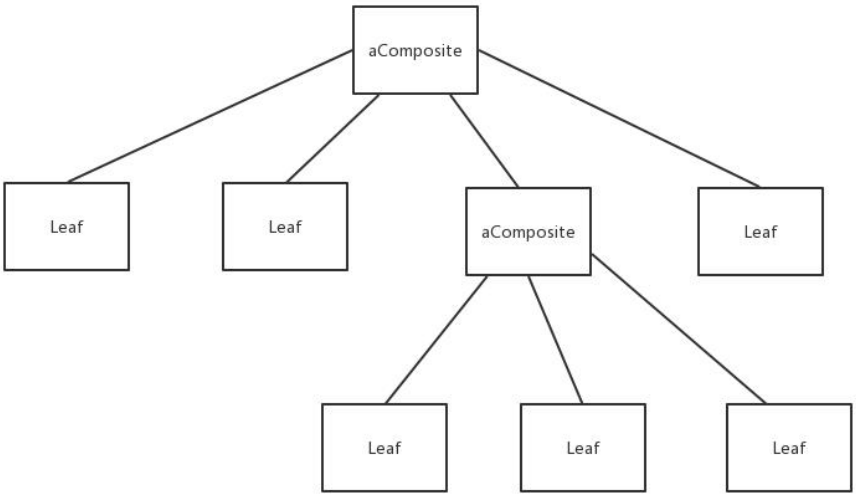
使用场景：部分、整体场景，如树形菜单，文件、文件夹的管理。
文件系统由文件和目录组成，每个文件里装有内容，而每个目录的内容可以有文件和目录，

目录就相当于是由单个对象或组合对象组合而成，如果你想要描述的是这样的数据结构，那么你就可以使用组合模式。

组合模式 2：原理分析

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。
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。
组成元素：
抽象构件角色(Composite)：是组合中对象声明接口，实现所有类共有接口的默认行为。
树叶构件角色(Leaf)：上述提到的单个对象，叶节点没有子节点。
树枝构件角色(Composite)：定义有子部件的组合部件行为，存储子部件，在 Component 接口中实现与子部件有关的操作。
客户端(Client)：使用 Component 部件的对象。

```
class AbstractFile {
public:
	virtual ~AbstractFile() = default;
	virtual void add(AbstractFile *file) = 0;
	virtual void remove(AbstractFile *file) = 0;
	virtual void killVirus() = 0;
protected:
	AbstractFile() = default;
};
class ImageFile : public AbstractFile...
class TextFile : public AbstractFile...

class Folder : public AbstractFile {
private:
	std::list<AbstractFile *> fileList;
	std::string name;
};
```



组合模式 3：编码与实现

享元模式 1：问题与需求

享元模式（Flyweight Pattern）（蝇量级;享元形式;轻量级）
问题：
1、系统中有大量对象
2、这些对象消耗大量内存，如果继续创建，有可能会造成内存溢出。

享元模式 2：原理与分析

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。运用共享技术有效地支持大量细粒度的对象。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。享元模式尝试重用（享）现有的同类对象，如果未找到匹配的对象，则创建新对象。
如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。
享元模式的工厂类维护了一个实例列表，这个列表中保存了所有的共享实例；当用户从享元模式的工厂类请求共享对象时，首先查询这个实例表，如果不存在对应实例，则创建一个；如果存在，则直接返回对应的实例。
关键代码：用 HashMap 存储这些对象。
应用实例：
1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。
2、数据库的数据池。

```
//享元类工厂类 DeviceFactory
class DeviceFactory{
public:
    DeviceFactory(){
        shared_ptr<NetworkDevice> nd1 = make_shared<Switch>("Ciso-WS-C2950-24");
        devices.insert(make_pair("cisco",nd1));
        shared_ptr<NetworkDevice> nd2 = make_shared<Hub>("TP-LINK-HF8M");
        devices.insert(make_pair("tp",nd2));
    }

    shared_ptr<NetworkDevice> getNetworkDevice(string type){
        if(devices.count(type)){
            totalTerminal++;
            return devices.find(type)->second;
        }else{
            return nullptr;
        }
    }

private:
    map<string,shared_ptr<NetworkDevice> > devices;
    int totalTerminal = 0;
};
```

享元模式 3：编码与实现

行为型模式

什么是行为型模式？
(Behavioral Pattern)，关注系统中对象之间的相互交互，研究运行时对象之间的相互通信和协作，明确对象职责
行为型模式包括：
策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式
策略模式 1：问题与需求
（Strategy Pattern）

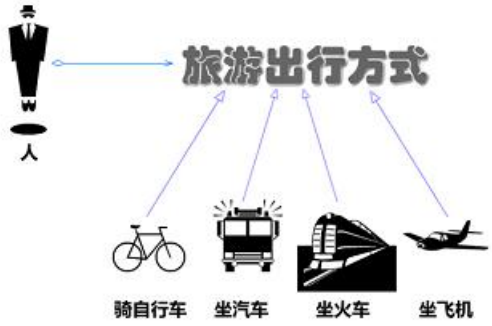
出门的时候会选择不同的出行方式，比如骑自行车、坐公交、坐火车、坐飞机、坐火箭等等，这些出行方式，每一种都是一个策略。

使用场景：

1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，如何动态地让一个对象在许多行为中选择一种行为？

2、一个系统需要动态地在几种算法中选择一种。

3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。



策略模式 2：原理分析

再比如我们去逛商场，商场现在正在搞活动，有打折的、有满减的、有返利的等等，其实不管商场如何进行促销，说到底都是一些算法，这些算法本身只是一种策略，并且这些算法是随时都可能互相替换的，比如针对同一件商品，今天打八折、明天满 100 减 30，这些策略间是可以互换的。

旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。

如我用策略模式，

```
person *per = new person(new  自行车);
per.action();
per = new person(new  汽车);
per.action();
class WindMode{
public:
    WindMode(IWind* wind): m_wind(wind){};
    ~WindMode(){free_ptr(m_wind);}
    void blowWind(){
        m_wind->blowWind();
    };
private:
    IWind* m_wind;
};
```

```
WindMode* warmWind = new WindMode(new WarmWind());
WindMode* coldWind = new WindMode(new ColdWind());
WindMode* noWind = new WindMode(new NoWind());

warmWind->BlowWind();
coldWind->BlowWind();
noWind->BlowWind();
```

策略模式 3：编码与实现

模板方法模式 1：问题与需求

（Template Method）
问题：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。
例如，去银行办理业务一般要经过以下 4 个流程：取号、排队、办理具体业务、对银行工作人员进行评分等，其中取号、排队和对银行工作人员进行评分的业务对每个客户是一样的，可以在父类中实现，但是办理具体业务却因人而异，它可能是存款、取款或者转账等，可以延迟到子类中实现。
又比如，一个人每天会起床、吃饭、做事、睡觉等，其中“做事”的内容每天可能不同。
解决方案：通过模板方法模式，把这些规定了流程或格式的实例定义成模板，允许使用者根据自己的需求去更新它。

模板方法模式 2：原理分析

模板方法：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中实现，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
它封装了不变部分，扩展可变部分，把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
它在父类中提取了公共的部分代码，便于代码复用。
部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则

```
class CAbstractClass{
public:
    CAbstractClass(){}
    virtual ~CAbstractClass(){}

    int Run(){
        ClusterInit();
```

```
        Stat();
        ClusterStat();
        return 0;
    }

protected:
    virtual int Stat() = 0;
    virtual int ClusterInit(){
        return 0;
    }
    virtual int ClusterStat(){
        return 0;
    }
};
```

模板方法模式 3：编码与实现

观察者模式 1：问题与需求

(Observer pattern)
问题：微信群专门用于通知消息，只要群里一有消息，我们就会知道。不管是我们订阅报纸的过程，还是接受群通知的过程，还是收听天气预报的过程，这是如何做到的呢？
多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

观察者模式 2：原理分析

观察者模式（又被称为发布-订阅（Publish/Subscribe）模式，属于行为型模式的一种，它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。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。
在观察者模式中有如下角色：
Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。
ConcreteSubject：具体主题（具体被观察者），该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。

Observer：抽象观察者，是观察者者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。

ConcreteObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

```
class ConcreteObserver : public Observer{
public:
    ConcreteObserver(Subject *pSubject) : m_pSubject(pSubject){}

    void Update(int value){
        cout << "ConcreteObserver get the update. New State:" << value << endl;
    }

private:
    Subject *m_pSubject;
};

class ConcreteSubject : public Subject{
public:
    void Attach(Observer *pObserver){m_ObserverList.push_back(pObserver);}
    void Detach(Observer *pObserver){m_ObserverList.remove(pObserver);}
    void Notify(){
        std::list<Observer *>::iterator it = m_ObserverList.begin();
        while (it != m_ObserverList.end()) {
            (*it)->Update(m_iState);
            ++it;
        }
    }
    void SetState(int state){
        m_iState = state;
    }
private:
    std::list<Observer *> m_ObserverList;
    int m_iState;
};
```

观察者模式 3：编码与实现

迭代子模式 1：问题与需求

（Iterator pattern）

STL 库或者 java 容器中常见 iterator

问题：在软件构建过程中，集合对象内部结构常常变化各异，但对于这些集合对象，我们希

望在不暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素；同时这种“透明遍历”也为同一种算法在多种集合对象上进行操作提供了可能。
使用面向对象技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方式。first/next/hasnext/current
迭代子(Iterator)模式又叫游标(Cursor)模式，常用的集合框架为 list，set，map 在实现的时候均可以支持迭代子模式
多个对象聚在一起形成的总体称之为聚集(Aggregate)，聚集对象是能够包容一组对象的容器对象。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部结构。
迭代子模式 2：原理分析
1、迭代子模式一般用于对集合框架的访问，常用的集合框架为 list，set，map 在实现的时候均可以支持迭代子模式
2、迭代子模式使用同一接口 Iterator 来完成对象的迭代，一般接口需要实现如下功能：first，next，isDone，currentItem 等遍历接口
3、常用的集合框架根据访问需要，可以使用数组，链表等数据结构进行实现，使用迭代子模式可以屏蔽掉数据结构遍历时候产生的差异

```
class ConcreteIterator : public Iterator{
	Aggregate *aggregate;
	size_t index;
public:
	ConcreteIterator(Aggregate *aggregate) : aggregate(aggregate), index(0) {}
	~ConcreteIterator() { cout << "~ConcreteIterator()" << endl; }
	string first() override {
		if (aggregate->Count() == 0)
			return string();
		return aggregate->pop(0);
	}
	string next() override {
		++index;
		if (index >= aggregate->Count())
			return string();
		return aggregate->pop(index);
	}
	string current() override {
		if (index >= aggregate->Count())
			return string();
		return aggregate->pop(index);
	}
	bool isEnd() override {
		return index >= aggregate->Count() || index < 0;
	}
};
```

```
class ConcreteAggregate : public Aggregate{
	vector<string> strs;
public:
	~ConcreteAggregate() { cout << "~ConcreteAggregate()" <<
		endl; }
	void push(const string& str) override { return
			strs.push_back(str); }
	string pop(size_t index) override {
		if (index >= strs.size())
			return string();
		return strs[index];
	}
	const size_t Count() const override { return strs.size(); }
	Iterator* createIterator() { return new
					ConcreteIterator(this); }
};

void TestIteratorPattern(){
	Aggregate *aggregate = new ConcreteAggregate();
	aggregate->push("str1");
	aggregate->push("str2");
	aggregate->push("str3");

	Iterator *iter = aggregate->createIterator();
	print(iter->first());
	while (!iter->isEnd()) {
		print(iter->next());
	}

	DELETE(iter);
	DELETE(aggregate);
}
```

迭代子模式 3：编码与实现

责任链模式 1：问题与需求

（Chain of Responsibility Pattern）

使用场景： 1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动

确定。

你要去给某公司借款 1 万元，当你来到柜台的时候向柜员发起 "借款 1 万元" 的请求时，柜员认为金额太多，处理不了这样的请求，他转交这个请求给他的组长，组长也处理不了这样的请求，那么他接着向经理转交这样的请求。

报销流程，项目经理<部门经理<总经理
其中项目经理报销额度不能超过 1000，部门经理报销额度不能超过 5000，超过 5000 的则需要总经理审核。

责任链模式 2：原理分析

责任链模式为请求创建一个接收者对象链，每个接收者都包含对另一个接收者的引用，如果一个对象不能处理该请求，那么它会把请求传给下一个接收者，依此类推
责任链模式避免了请求的发送者和接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

```
class Handler{
public:
	Handler() { m_pNextHandler = NULL; }
	virtual ~Handler() {}
	//设置下一个处理者
	void SetNextHandler(Handler *successor) { m_pNextHandler = successor; }
	//处理请求
	virtual void HandleRequest(int days) = 0;
protected:
	Handler *m_pNextHandler;   // 后继者
};
//具体处理者、主管
class Director :public Handler{
public:
	//处理请求
	virtual void HandleRequest(int days){
		if (days <= 1){
			cout << "我是主管，有权批准一天假，同意了！ " << endl;
		}else{
			m_pNextHandler->HandleRequest(days);
		}
	}
};
    Handler *director = new Director;
	Handler *manager = new Manager;
```

```
Handler *boss = new Boss;

//设置责任链
director->SetNextHandler(manager);
manager->SetNextHandler(boss);

director->HandleRequest(5);
```

责任链模式 3：编码与实现

命令模式 1：问题与需求

（Command Pattern）
拿订餐来说，客人需要向厨师发送请求，但是完全不知道这些厨师的名字和联系方式，也不知道厨师炒菜的方式和步骤。 命令模式把客人订餐的请求封装成 command 对象，也就是订餐中的订单对象。这个对象可以在程序中被四处传递，就像订单可以从服务员手中传到厨师的手中。这样一来，客人不需要知道厨师的名字，从而解开了请求调用者和请求接收者之间的耦合关系
客人的请求以命令的形式包裹在命令对象（command）中，并传给（waiter），waiter 对象寻找可以处理该命令的合适的对象（厨师，chef），并把该命令传给相应的对象，该对象执行命令。

命令模式 2：原理分析

命令模式的核心在于引入了命令类，通过命令类来降低发送者（客人）和接收者（厨师）的耦合度，请求发送者只需指定一个命令对象，再通过命令对象来调用请求接收者的处理方法。
三个角色对象：
客人生成命令(Command)
交给服务员(Waiter)
服务员通知厨师(Chef)做菜

```
class Command{
public:
    Command(Chef* chef);
    virtual ~Command();
    virtual void ExcuteCmd();

protected:
    Chef*     m_Chef;
```

```
};

class Chef{
public:
	Chef();
	void KungPaoChicken();
	void Mapodoufu();
	void BigPlateChicken();
};

class Waiter{
public:
	Waiter();
	void AddCmd(Command* cmd);
	void DelCmd(Command* cmd);
	void Nodify();
private:
	std::list<Command*>		m_CmdList;
};

std::shared_ptr<Chef> chef = std::make_shared<Chef>();
Waiter waiter;
std::shared_ptr<MapodoufuCmd> mpdf =
	std::make_shared<MapodoufuCmd>(chef.get());
waiter.AddCmd(mpdf.get());
waiter.Nodify();
```

命令模式 3：编码与实现

备忘录模式 1：问题与需求

（Memento pattern）
问题：需要保存和恢复数据的相关场景
游戏存档，我们玩游戏的时候肯定有存档功能，旨在下一次登录游戏时可以从上次退出的地方继续游戏，或者对复活点进行存档，如果挂掉了则可以读取复活点的存档信息重新开始。
与之相类似的就是数据库的事务回滚，或者重做日志 redo log 等。
提供一个可回滚的操作，如 ctrl+z、浏览器回退按钮、Backspace 键等
需要监控的副本场景
棋盘类游戏的悔棋

备忘录模式 2：原理分析

备忘录模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着这个状态。这样以后就可将该对象恢复到原先保存的状态。

```
class Memento {
public:
	Memento(int n, std::string s) {
		_s = s;
		_money = n;
	}
	void setMoney(int n) {
		_money = n;
	}

	int getMoney() {
		return _money;
	}

	void setState(std::string s) {
		_s = s;
	}

	std::string getState() {
		return _s;
	}

private:
	int _money;
	std::string _s;
};
class Worker {
public:
	Worker(std::string name, int money, std::string s) {
		_name = name;
		_money = money;
		_s = s;
	}

	void show() {
	}
```

```
    void setState(int money, std::string s) {
        _money = money;
        _s = s;
    }

    Memento* createMemento() {
        return new Memento(_money,_s);
    }

    void restoreMemento(Memento* m) {
        _money = m->getMoney();
        _s = m->getState();
    }
private:
    std::string _name;
    int _money;
    std::string _s;
};
```

备忘录模式 3：编码与实现

状态模式 1：问题与需求

（State Pattern）
在日常生活中，人在不同时间就有不同的状态，早上起来精神饱满，中午想睡觉，下午又渐渐恢复，晚上可能精神更旺也可能耗费体力只想睡觉，这一天中就对应着不同的状态。
电梯
状态：运行状态、开门状态、闭门状态、停止状态等
行为：开门，关门，运行，停止
行为随状态改变而改变的场景

状态模式 2：原理分析

状态模式，把状态声明为静态常量，有几个状态就声明几个状态常量。当一个对象的内在状态改变时允许改变其行为。对象行为处理本状态必须完成的任务，及决定是否可以过渡到其它状态。

与策略模式对比：

策略模式关注行为的变化,但归根结底只有一个行为,变化的只是行为的实现.客户不关注这些.当新增变化时对客户可以没有任何影响.
状态模式同样关注行为的变化,但这个变化是由状态来驱动，状态不同，行为就不同

```
class AbstractLift{
public:
	AbstractLift(void){
	}
	virtual ~AbstractLift(void) {
	}
	static const int OPENING_STATE = 1;
	static const int CLOSING_STATE = 2;
	static const int RUNNING_STATE = 3;
	static const int STOPPING_STATE = 4;
	virtual void SetState(int state) = 0;
	virtual void Open() = 0;
	virtual void Close() = 0;
	virtual void Run() = 0;
	virtual void Stop() = 0;
};
class CLift :public AbstractLift{
public:
	CLift(void);
	~CLift(void);
	void SetState(int state);
	void Open();
	void Close();
	void Run();
	void Stop();
private:
	int m_state;
};
void CLift::SetState(int state){
	this->m_state = state;
}
void CLift::Open(){
	switch(this->m_state){
	case OPENING_STATE:
		break;
	case CLOSING_STATE:
```

```
            cout<<”open”<<endl;
            this->SetState(OPENING_STATE);
            break;
        case RUNNING_STATE:
            break;
        case STOPPING_STATE:
            cout<<”open”<<endl;
            this->SetState(OPENING_STATE);
            break;
        }
}
void CLift::Run(){
        switch(this->m_state){
        case OPENING_STATE:
            break;
        case CLOSING_STATE:
            cout<<”run”<<endl;
            this->SetState(RUNNING_STATE);
            break;
        case RUNNING_STATE:
            break;
        case STOPPING_STATE:
            cout<<”run”<<endl;
            this->SetState(RUNNING_STATE);
            break;
        }
}
```

状态模式 3：编码与实现

访问者模式 1：问题与需求

（Visitor Pattern）
问题：
修电脑
医生看病
使用场景
一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。不能使用迭代器模式。
需要对一个对结构中的对象进行很多不同并且不相关的操作

访问者模式 2：原理分析

访问者
此处可为抽象类或接口，用于声明访问者可以访问哪些元素，具体到程序中就是 visit 方法的参数定义哪些对象是可以被访问的。影响访问者访问到一个类后该干什么、怎么干。
元素
声明接受哪一类访问者访问，程序上是通过 accpet 方法中的参数来定义的。实现 accept 方法，通常是 visitor.visit(this)。
结构对象
元素生产者，一般容纳在多个不同类、不同接口的容器，如 List、Set、Map 等，在项目中。

```
class Visitor {
public:
	Visitor(std::string name) {
		visitorName = name;
	}

	virtual void visitCPU( CPU* cpu ) {};
	virtual void visitVideoCard( VideoCard* videoCard ) {};
	virtual void visitMainBoard( MainBoard* mainBoard ) {}
;
	std::string getName() {
		return this->visitorName;
	};
private:
	std::string visitorName;
};

class FunctionVisitor : public Visitor {
public:
	FunctionVisitor(std::string name) : Visitor(name) {}
	virtual void visitCPU( CPU* cpu ) {
		cpu->getName() ;
	}
	void visitVideoCard( VideoCard* videoCard ) {
		videoCard->getName();
	}
	void visitMainBoard( MainBoard* mainboard ) {
		mainboard->getName();
	}
};
class Element {
```

```
public:
    Element( std::string name ) {
        eleName = name;
    }
    virtual void accept( Visitor* visitor ) {};
    virtual std::string getName() {
        return this->eleName;
    }
private:
    std::string eleName;
};
class CPU : public Element {
public:
    CPU(std::string name) : Element(name) {}
    void accept(Visitor* visitor) {
        visitor->visitCPU(this);
    }
};
class Computer {
public:
    Computer(CPU* cpu,VideoCard* videocard,
            MainBoard* mainboard) {
        elementList.push_back(cpu);
        elementList.push_back(videocard);
        elementList.push_back(mainboard);
    };
    void Accept(Visitor* visitor) {
        for( std::vector<Element*>::iterator i = elementList.begin(); i != elementList.end(); i++ ){
            (*i)->accept(visitor);
        }
    };
private:
    std::vector<Element*> elementList;
};
```

访问者模式 3：编码与实现

中介者模式 1：问题与需求

（Mediator pattern）

中介者使各个对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互
应用实例
联合国/WTO 作为中介者协调各个国家
房屋中介
MVC 框架，其中 C（Contorller 控制器）是 M（Model 模型）和 V（View 视图）的中介者

中介者模式 2：原理分析

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

```
class Mediator{
public:
    virtual void send(std::string    msg,
        Colleague * p) = 0;
};

class ConcreteMediator : public Mediator{
private:
    // 这里可是一个列表
    Colleague * m_p1;
    Colleague * m_p2;
public:
    void addColleague(Colleague * p1,
        Colleague * p2){
        m_p1 = p1;
        m_p2 = p2;
    }
    void send(std::string msg, Colleague * p){
        if (p == m_p1)
            m_p2->notify(msg);
        else
            m_p1->notify(msg);
    }
};
class Colleague{
protected:
    Mediator * m_mediator;
public:
```

```
    Colleague(Mediator * p){
        m_mediator = p;
    }
    virtual void send(std::string msg) = 0;
    virtual void notify(std::string msg) = 0;
};


class ConcreteColleague_0 : public Colleague{
public:
    ConcreteColleague_0(Mediator * p) : Colleague(p) {}
    void send(std::string msg){
        m_mediator->send(msg, this);
    }
    void notify(std::string msg){
        std::cout << "Colleague_0 收到了消息：" << msg << std::endl;
    }
};
```

中介者模式 3：编码与实现

解释器模式 1：问题与需求

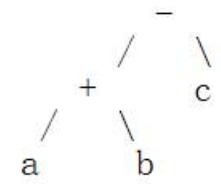
（Interpreter pattern）
给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。诸如此类的例子也有很多，比如编译器、正则表达式，四则运算等等。
应用实例
编译器
运算表达式计算、正则表达式
SQL 解析
符号处理引擎
解释器模式在实际的系统开发中使用的非常少，因为它会引起效率、性能以及维护方面的问题，并且难度较大，一般在一些大中型的框架型项目中能够找到它的身影。而现在又有很多的开源库提供了对实际需要的支持

解释器模式 2：原理分析

在解释器模式中可以通过一种称之为抽象语法树(Abstract Syntax Tree, AST)的图形方式来直观地表示语言的构成，每一棵抽象语法树对应一个语言实例

抽象表达式（AbstractExpression）：具体的解释任务由各个实现类完成。
终结符表达式（TerminalExpression）：实现与文法中的元素相关联的解释操作，通常一个解释器模式中只有一个终结表达式，但有多个实例，对应不同的终结符。
非终结符表达式（NonterminalExpression）：文法中的每条规则对应于一个非终结表达式，非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式

从左到向分析表达式（如:a+b-c），最终的语法树如下：




```
//抽象表达式类
class Expression{
public:
    //a+b-c, a=5, b=7, c=10
    virtual int interpreter(map<string, int>& var) = 0;
    virtual ~Expression(){};
};
//变量解析器（终结符表达式）
class VarExpression : public Expression{
    string key;
public:
    VarExpression(string key){
        this->key = key;
    }
    //从 map 中取出变量的值
    int interpreter(map<string, int>& var){
        return var[key];
    }
};
//抽象运算符号解析器
class SymbolExpression : public Expression{
protected:
    Expression* left;
    Expression* right;
public:
    SymbolExpression(Expression* left, Expression* right){
        this -> left = left;
        this -> right = right;
    }
```

```
};

//加法解析器
class AddExpression : public SymbolExpression{
public:
    AddExpression(Expression* left, Expression* right): SymbolExpression(left,right){
    }

    //把左右两个表达式运算的结果加起来
    int interpreter(map<string, int>& var){
        return left->interpreter(var) + right ->interpreter(var);
    }
    ~AddExpression(){
        cout << "~AddExpression()" << endl;
    }
};

class Calculator{
private:
    Expression* expression;
public:
    //构造函数传参，并解析表达式，构建语法树
    //a+b-c, expression 为语法树的根结点
    Calculator(string expStr){}
    int run(map<string, int>& var){
        return (expression == NULL) ? 0 : expression->interpreter(var);
    }

}
    string expStr = "a+b-c"; //为简化处理，这里必须是合法的表达式

    map<string, int> var;    //相当于 Interpreter 模式中的 Context
    var["a"] = 100;
    var["b"] = 20;
    var["c"] = 40;

    Calculator cal(expStr);

    cout <<"运算结果为：" << expStr << " = " << cal.run(var) << endl;
```

解释器模式 3：编码与实现