# 概述

设计模式的基础是：多态。

学习设计模式将有助于更加深入地理解面向对象思想，让你知道：

1. 如何将代码分散在几个不同的类中？
2. 为什么要有“接口”？
3. 何谓针对抽象编程？
4. 何时不应该使用继承？
5. 如何不修改源代码增加新功能？
6. 更好地阅读和理解现有类库与其他系统中的源代码。

参考书籍：《设计模式：可复用面向对象软件的基础》

# 分类

1、从目的来看：

创建型模式

结构型模式

行为型模式

2、从范围来看：

类模式处理类与子类的静态关系

对象模式处理对象间的动态关系。

3、从封装角度来看：

组件协作：Template Method，Strategy，Observer/Event

单一职责：Decorator，Bridge

对象创建：Factory Method，Abstract Factory，Prototype，Builder

对象性能：Singleton，Flyweight

接口隔离：Façade，Proxy，Mediator，Adaptor

状态变化：Memento，State

数据结构：Composite，Iterator，Chain of Responsibility

行为变化：Command，Visitor

领域问题：Interpreter

GoF提出的设计模型有23个，包括：

创建型模式

结构型模式

行为型模式

有一个“简单工厂模式”不属于GoF23种设计模式，因此设计模式目前分类：

GoF的23种 + 简单工厂模式 = 24种

## 创建型

创建型（Creational）模式：如何创建对象。

## 结构型

结构型（Structual）模式：如何实现类或对象的组合。

## 行为型

行为型（Behaviioral）模式：类或对象怎样交互以及怎样分配职责。

# 面向对象

**理解面向对象：**

1. 向下：面向对象三大机制

封装：隐藏内部实现

继承：复用现有代码

多态：改写对象行为

1. 向上：深刻把握面向对象机制所带来的的抽象意义，理解如何使用这些机制来表达现实世界，掌握什么是“好的面向对象设计”。

**如何解决复杂性？**

1. 分解

人们面对复杂性有一个常见的做法：即分而治之，将大问题分解为多个小问题，将复杂问题分解为多个简单问题。

1. 抽象

更高层次来讲，人们处理复杂性有一个通用的技术，即抽象。由于不能掌握全部的复杂对象，我们选择忽视它的非本质细节，而去处理泛化和理想化的对象模型。

# 封装分类

## 对象创建模式

### 工厂方法模式Factory Method

### 抽象工厂Abstract Factory

### 原型模式Prototype

### 建造者模式Builder

## 对象性能模式

### 单例模式Singleton

### 享元模式Flyweight

## 接口隔离模式

### 外观模式Facader

### 代理模式Proxy

### 适配器模式Adapter

### 中介者模式Mediator

## 状态变化模式

### 状态模式State

### 备忘录模式Memento

## 单一职责模式

### 装饰模式Decorator

### 桥接模式Bridge

## 动作协作模式

### 模板方法Template Method

### 策略模式Strategy

### 观察者模式Observer/Event

## 数据结构模式

### 组合模式Composite

### 迭代器模式Iterator

### 职责链模式Chain of Responsibility

## 行为变化模式

### 命令模式Command

### 访问器Visitor

## 领域规则模式

### 解析器Interpret

# 面向对象设计原则

对于面向对象软件系统的设计而言，在支持可维护性的同时，提高系统的可复用性是一个至关重要的问题。如何同时提高一个软件系统的可维护性和可复用性是面向对象设计需要解决的核心问题之一。在面向对象设计中，可维护性的复用是以设计原则为基础的。每一个原则都蕴含一些面向对象设计的思想，可以从不同的角度提升一个软件结构的设计水平。

面向对象设计原则为支持可维护性复用而诞生，这些原则蕴含在很多设计模式中，它们是从许多设计方案中总结出的指导性原则。面向对象设计原则也是我们用来评价一个设计模式的使用效果的重要指标之一。

**原则的目的：高内聚，低耦合**

## 单一职责原则SRP

单一职责原则（Single Responsibility Priciple，SRP）:

类的职责单一，对外只提供一种功能，引起类变化的原因应该只有一个。

变化的方向隐含着类的责任。

## 开放封闭原则OCP

### 原理

开闭原则（Open-Closed Priciple，OCP）:

对扩展开放，对更改封闭，即类的改动是通过增加代码进行的，而不是修改源代码。

类模块应该是可扩展的，但是不可修改。

### 代码

#### 旧方案

不合理的方案：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

// 开闭原则对扩展开放，对修改关闭，增加功能是通过增加代码来实现，而不是去修改源代码

//语算器类

class Caculaor{

private:

int m\_a;

int m\_b;

string m\_operator;

int m\_ret;

public:

Caculaor(int a, int b, string moperator){

this->m\_a = a;

this->m\_b = b;

this->m\_operator = moperator;

}

int getResult(){

if (m\_operator.compare("+") == 0){

return m\_a + m\_b;

}

else if (m\_operator.compare("-") == 0){

return m\_a - m\_b;

}

else if (m\_operator.compare("\*") == 0){

return m\_a\*m\_b;

}

else if (m\_operator.compare("/") == 0) {

return m\_a / m\_b;

}

}

};

/\*

缺点：如果增加取模的运算需要修改getResult成员方法，如果增加新功能的情况下要修改调代码，那么就会有修改出错的可能性。

我们应该在增加新的功能时候，不能影响其他已经完成的功能。

这就是对修改关闭，对扩展开放，叫做开闭原则

\*/

void test01(){

Caculaor\* caculator = new Caculaor(10, 20, "+");

cout << caculator->getResult() << endl;

}

int main(){

test01();

return 0;

}

#### 新方案

修改后的方案：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

// 开闭原则对扩展开放，对修改关闭，增加功能是通过增加代码来实现，而不是去修改源代码

//写一个抽象类

class AbstractCaculator{

public:

virtual int getResult() = 0;//纯虚函数

virtual void setOperatorNumber(int a, int b)=0;

};

//加法计算器类

class PlusCaculator :public AbstractCaculator{

public:

int mA;

int mB;

public:

virtual void setOperatorNumber(int a, int b){

this->mA = a;

this->mB = b;

}

virtual int getResult(){

return mA + mB;

}

};

//减法计算器类

class MinuteCaculator :public AbstractCaculator{

public:

int mA;

int mB;

public:

virtual void setOperatorNumber(int a, int b){

this->mA = a;

this->mB = b;

}

virtual int getResult(){

return mA - mB;

}

};

//乘法计算器类

class MultiplyCaculator :public AbstractCaculator{

public:

int mA;

int mB;

public:

virtual void setOperatorNumber(int a, int b){

this->mA = a;

this->mB = b;

}

virtual int getResult(){

return mA \* mB;

}

};

//取模计算器类，通过增加代码，而不是修改原来的

class QumoCaculator :public AbstractCaculator{

public:

int mA;

int mB;

public:

virtual void setOperatorNumber(int a, int b){

this->mA = a;

this->mB = b;

}

virtual int getResult(){

return mA % mB;

}

};

void test01() {

AbstractCaculator \* caculator = new PlusCaculator;

caculator->setOperatorNumber(10, 20);

cout << "ret:" << caculator->getResult() << endl;

caculator = new MinuteCaculator;

caculator->setOperatorNumber(10, 20);

cout << "ret:" << caculator->getResult() << endl;

}

int main(){

test01();

return 0;

}

## Liskov替换原则LSP

里氏代换原则（Liskov Substitution Priciple，LSP）:

子类必须能够替换它们的基本（is-a），即任何抽象类出现的地方都可以用他的实现类进行替换，实际上就是虚拟机制，语言级别实现面向对象功能。

继承表达类型抽象。

## 优先使用对象组合，而不是类继承

类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。

继承在某种程度上破坏了封装性，子类父类耦合度高。

而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。

## 依赖倒置原则DIP

### 原理

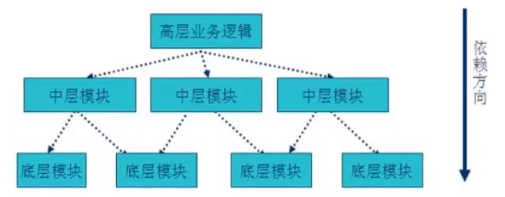
依赖倒转原则（Dependence Inversion Priciple，DIP）：

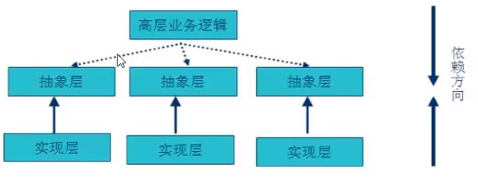
高层模块（稳定）不应该依赖于低层模块（变化），二者都应该依赖于抽象（稳定）。

抽象（稳定）不应该依赖于实现细节（变化），实现细节应该依赖于抽象（稳定）。

即，依赖于抽象（接口），不要依赖具体的实现（类），也就是**针对接口编程**。

注：之所以叫依赖倒置，是相对于原来的高层依赖底层，抽象依赖具体的“倒置”。





### 代码

#### 旧方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//银行工作人员

class BankWorker{

public:

void saveService(){

cout << "办理存款业务..." << endl;

}

void payService(){

cout << "办理支付业务.." << endl;

}

void tranferService(){

cout << "办理转账业务.." << endl;

}

};

//中层模块

void doSaveBussiness(BankWorker\* worker) {

worker->saveService();

}

void doPayBussiness(BankWorker\* worker){

worker->payService();

}

void doTransferBussiness(BankWorker\* worker) {

worker->tranferService();

}

void test01(){

BankWorker\*worker = new BankWorker;

doSaveBussiness(worker);//办理存款业务

doPayBussiness(worker);//办理支付业务

doTransferBussiness(worker);//办理转账业务

}

int main(){

test01();

return 0;

}

#### 新方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//银行工作人员

class AbstractWorker{

public:

virtual void doBusiness() = 0;

};

//专门负责办理存款业务的工作人员

class SaveBankWorer :public AbstractWorker{

public:

virtual void doBusiness(){

cout << "办理存款业务.." << endl;

}

};

//专门办理支付业务的工作人员

class PayBankWorker :public AbstractWorker {

public:

virtual void doBusiness(){

cout << "办理支付业务.." << endl;

}

};

//专门办理转账业务的工作人员

class TransferBankWorker :public AbstractWorker{

public:

virtual void doBusiness(){

cout << "办理转账业务..." << endl;

}

};

//中层业务

void doNewBusiness(AbstractWorker\*worker){

worker->doBusiness();

}

void test02(){

AbstractWorker\* transfer = new TransferBankWorker;

doNewBusiness(transfer);

}

int main(){

test02();

return 0;

}

## 接口隔离原则ISP

接口隔离原则（Interface Segregation Priciple，ISP）：

不应该强迫客户程序依赖它们不用的方法，一个接口应该只提供一种对外功能，不应该把所有操作都封装到一个接口中去。

接口应该小而完备。

注：单一职责原则说的是类，接口隔离原则说的是方法（接口）。

## 合成复用原则CRP

### 原理

合成复用原则（Composite Reuse Priciple，CRP）:

类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。

如果使用继承，会导致父类的任何交换都可能影响到子类的行为。继承在某种程度上破坏了封装性，子类父类耦合度高。

而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。

注：优先使用对象组合，而不是类继承

### 代码

#### 旧方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//抽象车

class AbstractCar{

public:

virtual void run() = 0;

};

//大众车

class Dazhong :public AbstractCar{

public:

virtual void run(){

cout << "大众车启动..." << endl;

}

};

// 拖拉机

class Tuolaji :public AbstractCar{

public:

virtual void run(){

cout << "拖拉机启动.." << endl;

}

};

class Person1 : public Tuolaji{

public:

void Doufeng(){//兜风

run();

}

};

class Person2 : public Dazhong{

public:

void Doufeng(){//兜风

run();

}

};

int main(){

return 0;

}

#### 新方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//抽象车

class AbstractCar{

public:

virtual void run() = 0;

};

//大众车

class Dazhong :public AbstractCar{

public:

virtual void run(){

cout << "大众车启动..." << endl;

}

};

// 拖拉机

class Tuolaji :public AbstractCar{

public:

virtual void run(){

cout << "拖拉机启动.." << endl;

}

};

/\*

class Person1 : public Tuolaji{

public:

void Doufeng(){//兜风

run();

}

};

class Person2 : public Dazhong{

public:

void Doufeng(){//兜风

run();

}

};

\*/

//可以使用组合

class Person{

public:

AbstractCar\*car;

public:

void setCar(AbstractCar\*car){

this->car = car;

}

void Doufeng(){

this->car->run();

}

~Person(){

if (this->car != NULL) {

delete this->car;

}

}

};

void test01()

{

Person\* p = new Person;

p->setCar(new Dazhong);

p->Doufeng();

p->setCar(new Tuolaji);

p->Doufeng();

delete p;

}

int main(){

test01();

return 0;

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//抽象车

class AbstractCar{

public:

virtual void run() = 0;

};

//大众车

class Dazhong :public AbstractCar{

public:

virtual void run(){

cout << "大众车启动..." << endl;

}

};

// 拖拉机

class Tuolaji :public AbstractCar{

public:

virtual void run(){

cout << "拖拉机启动.." << endl;

}

};

/\*

class Person1 : public Tuolaji{

public:

void Doufeng(){//兜风

run();

}

};

class Person2 : public Dazhong{

public:

void Doufeng(){//兜风

run();

}

};

\*/

//可以使用组合

class Person{

public:

AbstractCar\*car;

public:

void setCar(AbstractCar\*car){

this->car = car;

}

void Doufeng(){

this->car->run();

if (this->car != NULL) {

delete this->car;

}

this->car = NULL;

}

};

void test01()

{

Person\* p = new Person;

p->setCar(new Dazhong);

p->Doufeng();

p->setCar(new Tuolaji);

p->Doufeng();

delete p;

}

int main(){

test01();

return 0;

}

## 封装变化点

使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。

## 迪米特法则LoD

### 原理

迪米特法则（Law of Demeter，LoD）/最少知识原则:

一个对象应当对其他对象尽可能少的了解，从而降低各个对象之间的耦合，提高系统的可维护性。例如在一个程序中，各个模块之间相互调用时，通常会提供一个统一的接口来实现。这样其他模块不需要了解另外一个模块的内部实现细节，这样当一个模块内部的实现发生改变时，不会影响其他模块的使用（黑盒原理）。

不将变量类型声明为某个特定的具体类，而是声明为某个接口。

客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。

减少系统中各部分的依赖关系，从而实现“高内聚、松耦合”的类型设计方案。

注：针对接口编程，而不是针对实现编程

### 代码

#### 旧方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <string>

using namespace std;

//迪米特法则又叫最少知识原则

//写一个抽象类

class AbstractBuilding{

public:

virtual void sale() = 0;

};

//楼盘A

class BuildingA :public AbstractBuilding{

public:

string mQulity;

public:

BuildingA(){

mQulity = "高品质";

}

virtual void sale(){

cout << "楼盘A" << mQulity << "被售卖!" << endl;

}

};

//楼盘A

class BuildingB :public AbstractBuilding{

public:

string mQulity;

public:

BuildingB(){

mQulity = "低品质";

}

virtual void sale(){

cout << "楼盘A" << mQulity << "被售卖!" << endl;

}

};

void test01() {

BuildingA\*ba = new BuildingA;

if (ba->mQulity == "低品质"){

ba->sale();

}

BuildingB\*bb = new BuildingB;

if (bb->mQulity == "低品质"){

bb->sale();

}

}

int main(){

test01();

return 0;

}

#### 新方案

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <string>

#include <vector>

using namespace std;

//迪米特法则又叫最少知识原则

//写一个抽象类

class AbstractBuilding{

public:

string mQulity;

public:

virtual void sale() = 0;

virtual string getQuality() = 0;

};

//楼盘A

class BuildingA :public AbstractBuilding{

public:

BuildingA(){

mQulity = "高品质";

}

virtual void sale(){

cout << "楼盘A" << mQulity << "被售卖!" << endl;

}

virtual string getQuality(){

return mQulity;

};

};

//楼盘B

class BuildingB :public AbstractBuilding{

public:

BuildingB(){

mQulity = "低品质";

}

virtual void sale(){

cout << "楼盘A" << mQulity << "被售卖!" << endl;

}

virtual string getQuality(){

return mQulity;

};

};

void test01() {

BuildingA\*ba = new BuildingA;

if (ba->mQulity == "低品质"){

ba->sale();

}

BuildingB\*bb = new BuildingB;

if (bb->mQulity == "低品质"){

bb->sale();

}

}

//中介类

class Mediator{

public:

vector<AbstractBuilding\*> vBuilding;

public:

Mediator(){

AbstractBuilding\* building = new BuildingA;

vBuilding.push\_back(building);

building = new BuildingB;

vBuilding.push\_back(building);

}

~Mediator(){

for (

vector<AbstractBuilding\*>::iterator it = vBuilding.begin();

it != vBuilding.end();

it++

){

if (\*it != NULL) delete \*it;

}

}

//对外提供接口

AbstractBuilding\* findMyBuilding(string quality){

for (

vector<AbstractBuilding\*>::iterator it = vBuilding.begin();

it != vBuilding.end();

it++

){

//两个if任选一个

//if ((\*it)->getQuality() == quality) return \*it;

if ((\*it)->mQulity == quality) return \*it;

}

return NULL;

}

};

//客户端

void test02(){

Mediator\*mediator = new Mediator;

AbstractBuilding\*building = mediator->findMyBuilding("高品质");

if (building != NULL){

building->sale();

}

else{

cout << "没有符合您条件的楼盘！" << endl;

}

building = mediator->findMyBuilding("低品质");

if (building != NULL){

building->sale();

}

else{

cout << "没有符合您条件的楼盘！" << endl;

}

}

int main(){

test02();

return 0;

}

# 使用

什么时候不用设计模式？

1. 代码可读性很差时
2. 需求理解还很浅时
3. 变化没有显现时
4. 不是系统的关键依赖点
5. 项目没有复用价值时
6. 项目将要发布时