设计模式

一、概述

设计模式是解决问题的方案，学习现有的设计模式可以做到经验复用。设计模式（Design Pattern）是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。

二、原则

1、单一职责原则

就一个类而言，应该仅有一个引起它变化的原因。如果存在多于一个动机去改变一个类，那么这个类就具有多于一个的职责，就应该把多余的职责分离出去，再去创建一些类来完成每一个职责。

举个例子：一个人身兼数职，而这些事情相关性不大，甚至有冲突，那他就无法很好的解决这些问题职责，应该分到不同的人身上去做。

单一职责原则是实现高内聚低耦合的最好方法，没有之一。

2、开放-封闭原则

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。目的就是保证程序的扩展性好，易于维护和升级。

开闭原则被称为面向对象设计的基石，实际上，其他原则都可以看作是实现开闭原则的工具和手段。意思就是：软件对扩展应该是开放的，对修改是封闭的，通俗来说就是，开发一个软件时，应该对其进行功能扩展，而在进行这些扩展时，不需要对原来的程序进行修改。

好处是：软件可用性非常灵活，扩展性强。需要新的功能时，可以增加新的模块来满足新需求。另外由于原来的模块没有修改，所以不用担心稳定性的问题。

3、依赖倒置原则

定义：

1. 高层模块不应该依赖低层模块，两者都要改依赖其抽象（模块间的依赖通过抽象产生，实现类不发生直接的依赖关系）
2. 抽象不应该依赖细节（接口或者抽象类不依赖实现类）
3. 细节可以依赖抽象（实现类依赖接口或者抽象类）

举例：存在一个Driver类，成员为一个Car对象，还有一个driver()方法，Car对象中有两个方法start()与stop()。显然Driver依赖Car，也就是说Driver类调用了Car类中的方法。但是当增加Driver类对于Bus类的支持时（司机有需要开公交车），

就必须更改Driver中的代码，就破坏了开放封闭原则。根本原因在于高层的的Driver类与底层的Car类仅仅的耦合在一起的。解决方法之一就是：对Car类和Bus类进行抽象，引入抽象类Automoble。而Car和Bus则是对Automobile的泛化。

面向接口编程：
接口负责定义 public 属性和方法，并且声明与其它对象的依赖关系，抽象类负责公共构造部分的实现，实现类准确实现业务逻辑，同时在适当的时候对父类进行细化。

4、里氏替换原则

子类必须能替换掉它们的父类型

继承的优点：

1. 代码共享，提高代码的重用性。
2. 提高代码的可扩展性。
3. 提高产品或者项目的开放性。

继承的缺点：

1. 继承是侵入式的，只要继承，就拥有了父类的属性和方法。
2. 降低代码灵活性，子类拥有了父类的属性和方法，多了一些约束。
3. 增强了耦合性。父类的常量、变量或方法改动时，必须还要考虑子类的修改，可能会有大段代码需要重构。

里氏替换原则四层含义：

1. 子类必须完全实现父类的方法

   在类中调用其他类时务必使用父类或接口，如若不能，则说明类的设计已经违背LSP原则。

   如果子类不能完整的实现父类的方法，或者父类的方法在子类中发生畸变，这建议断开父子继承关系，采用依赖、聚集、组合等方式代替继承。

2. 子类可以有自己的特性：即子类出现的地方父类未必可以出现。

3. 覆盖父类的方法时输入参数可以被放大：输入参数类型宽于父类的类型的覆盖范围，例如 hashmap -> map。

4. 覆盖父类的方法时输出参数可以被缩小

5、接口隔离原则

用于恰当的划分角色和接口，具有两种含义：1、用户不应该依赖它不需要的借口；2、类间的依赖关系应该建立在最小的的接口上。

将这两个定义概括为一句话：建立单一接口，代替庞大臃肿的接口。通俗来说就是：接口尽量细化，同时保证接口中的方法尽量的少。一个接口中包含太多的行为时，会导致它们与客户端的不正常依赖关系，要做的就是分离接口，从而实现解耦。

回到上述的单一职责原则，要求行为分离接口接口细化，感觉有些相同。但实际上，单一职责原则要求类与接口的职责单一，注重的是职责，没有要求接口尽量的少。

在接口隔离原则中，要求尽量使用多个专门的接口。专门的接口也就是提供给多个模块的接口。提供给几个模块就应该有几个接口，而不是建立一个臃肿庞大的接口，所有的模块都可以访问。

但是接口的设计是有限度的。接口的设计粒度越小系统越灵活，这是事实，但是接口太多这也就使得结构复杂，维护难度大。因此实际中，怎样把握就靠开发的经验和常识了。

6、迪米特原则（最少知道原则）

一个对象应该对其他对象有最少的了解。通俗来说就是，一个类对自己需要耦合或者调用的类知道的最少，你类内部怎么复杂，我不管，那是你的事，我只知道你有那么多公用的方法，我能调用。

迪米特原则不希望类与类之间建立直接的接触。如果真的需要有联系，那么就通过它们的友元类来传达。举例来说：你需要买房子了，现在存在三座合适的楼盘

A，B，C，但是你不必直接去楼盘买楼，而是在售楼处去了解情况。这样就减少了你（购房者）与楼盘两个类之间耦合。

但是应用迪米特原则很可能会造成一个后果：系统会存在大量的中介类，这些类（如上面的售楼处类）之所以存在是为了传递类之间的相互调用关系，这就一定会程度上增加了系统的复杂度。

迪米特原则核心观念就是：类间解耦，弱耦合。

三、创建型

1、单例模式（Singleton）

Intent

确保一个类只有一个实例，并提供该实例的全局访问点

Class Diagram

使用一个私有构造函数、一个私有静态变量和一个公有静态函数来实现。

私有构造函数保证了不能通过构造函数来创建对象示例，只能通过公有静态函数来返回唯一的私有静态变量。

懒汉式（线程不安全）

私有静态变量m_pInstance被延迟实例化，这样做的好处是，如果没有用到该类，那么就不会实例化m_pInstance，从而节约资源。

这个实现在多线程环境下是线程不安全的，如果多个线程能够同时进入if(m_pInstance == NULL)，并且此时m_pInstance为NULL，那么会有多个线程执行m_pInstance = new CSingleton();语句，这将导致实例化多次m_pInstance。

```
class CSingleton
{
private:
    static CSingleton *m_pInstance;
    CSingleton() //构造函数是私有的
```

```
    {
    }
public:
    static CSingleton * GetInstance()
    {
        if(m_pInstance == NULL) //判断是否第一次调用
            m_pInstance = new CSingleton();
        return m_pInstance;
    }
};
Singleton* Singleton::m_pInstance = NULL;
```

饿汉式（线程安全）

线程不安全问题主要是由于m_pInstance被实例化多次，采取直接实例化m_pInstance的方式就不会有线程不安全的问题。

但是直接实例化的方式也丢失了延迟实例化带来的节约资源的好处。

```
class Singleton
{
private:
    Singleton()
    {
        cout << "Singleton()" << endl;
    }
    static Singleton* m_pInstance;//这里是声明，需要在类的外面定义
public:
    static Singleton* GetSingleton()
    {
        return m_pInstance;
    }
    static Singleton* Destroy()
    {
        delete m_pInstance;
        m_pInstance = NULL;
    }
};
Singleton* Singleton::m_pInstance = new Singleton; //这里是定义，分配内存
```

懒汉式（线程安全）

只要对m_pInstance加锁，就可以保证在同一个时间点只能有一个线程实例化，从而避免了多次实例化m_pInstance。

但是当一个线程进入该方法后，其他试图进入该方法的线程都必须等待，即使m_pInstance已经被实例化了，这会让线程阻塞的时间过长，因此该方法有性能问题，不推荐。

```
std::mutex resource_mutex;//互斥量
class CSingleton
{
private:
    static CSingleton *m_pInstance;
    CSingleton() //构造函数是私有的
    {
    }
public:
    static CSingleton * GetInstance()
    {
        std::lock_guard<std::mutex> lg(resource_mutex);//自动加锁
        if(m_pInstance == NULL) //判断是否第一次调用
            m_pInstance = new CSingleton();
        return m_pInstance;
    }
};
Singleton* Singleton::m_pInstance = NULL;
```

懒汉式（双重校验锁）

m_pInstance只需要被实例化一次，之后就可以直接使用。加锁操作只需对实例化那部分代码进行，只有当m_pInstance没有被实例化时，才需要进行加锁。

双重校验锁先判断m_pInstance是否被实例化，如果没有被实例化，那么才对实例化语句进行加锁。

在这里，再加上一个释放单例对象的一种方法，类里面套一个类，静态对象的生命周期到程序退出的时候，当程序退出的时候，会调用这个对象的析构函数，从而达到释放单例对象的目的。

```
std::mutex resource_mutex;//互斥量
class CSingleton
{
private:
    static CSingleton *m_pInstance;
    CSingleton() //构造函数是私有的
    {
    }
public:
```

```
    static CSingleton * GetInstance()
    {
        if(m_pInstance == NULL)
        {
            std::lock_guard<std::mutex> lg(resource_mutex);//自动加锁
            if(m_pInstance == NULL) //判断是否第一次调用
            {
                m_pInstance = new CSingleton();
                static GC gc; //静态对象的生命周期到程序退出的时候，当程序退出的时候，会调用这个对象的析构函数
            }
        }
        return m_pInstance;
    }
    class GC {
        //类中套类，用来释放对象
    public:
        ~GC() {
            if (CSingleton::m_pInstance) {
                delete CSingleton::m_pInstance;
                CSingleton::m_pInstance = NULL;
            }
        }
    };
};
Singleton* Singleton::m_pInstance = NULL;
```

2、简单工厂模式

Intent

在创建一个对象时不向客户暴露内部的细节，并提供一个创建对象的通用接口。

Class Diagram

简单工厂模式的实例化操作单独放在一个类中，这个类就是简单工厂类，让简单工厂类来决定应该用哪个具体子类来实例化。

这个做能够把客户类和具体子类的实现解耦，客户类不再需要知道有哪些子类以及应当实例化哪个子类。客户类往往有多个，如果不使用简单工厂，那么所有客户类都要知道所有子类的细节。而且一旦子类发生改变，例如增加子类，那么所有的客户类都要进行改变。

比如一个要实现一个计算器功能，可使用如下方法。

```
//Operation运算类
class Operation
{
private:
    double numA = 0;
    double numB = 0;
public:
    double getNumA()
    {
        return numA;
    }
    void setNumA(double numA)
    {
        this.numA = numA;
    }
    double getNumB()
    {
        return NumB;
    }
    void setNumB(double NumB)
    {
        this.NumB = NumB;
    }
    virtual double getResult()
    {
        double result = 0;
        return result;
    }
};
```

```
//加减乘除类
class OperationAdd:public Operation
{
public:
    double getResult()
    {
        double result = 0;
        result = numA + numB;
        return result;
    }
}

class OperationSub:public Operation
{
public:
    double getResult()
    {
        double result = 0;
```

```
        result = numA - numB;
        return result;
    }
}

class OperationMul:public Operation
{
public:
    double getResult()
    {
        double result = 0;
        result = numA * numB;
        return result;
    }
}

class OperationDiv:public Operation
{
public:
    double getResult()
    {
        double result = 0;
        if(numB == 0)
        {
            break;
        }
        result = numA / numB;
        return result;
    }
}
```

```
//简单工厂类
class OperationFactory
{
public:
    static Operation createOpration(string operation)
    {
        Operation oper = NULL;
        switch(operation)
        {
            case "+":
                oper = new OperationAdd();
                break;
            case "-":
                oper = new OperationSub();
                break;
            case "*":
                oper = new OperationMul();
```

```
                break;
            case "/":
                oper = new OperationDiv();
                break;
        }
        return oper;
    }
};
```

```
//客户端代码
Operation oper;
oper = OperationFactory::createOpration("+");
oper.numA = 1;
oper.NumB = 2;
double result = oper.getResult();
```

3、工厂方法

Intent

定义了一个创建对象的接口，但由子类决定要实例化哪个类。工厂方法把实例化操作推迟到子类。

Class Diagram

在简单工厂中，创建对象的是另一个类，而在工厂方法中，是由子类来创建对象。

简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。如上面的计算器，让客户端不用管该用哪个类的实例，只需把“+”给工厂，工厂自动就给出了相应的实例，客户端只要去做运算就可以了，不同的实例会实现不同的运算。但是，如果要加一个‘求M的N次方’的功能，我们一定需要给运算工厂类的方法里加‘case’的分支条件，这就需要修改原有的类。违背了**开放-封闭原则**。

```
//工厂接口
class IFactory
{
public:
    virtual Operation createOperation();
};
```

```
class AddFactory : public IFactory
{
public:
    Operation createOperation()
    {
        return new OperationAdd();
    }
};
//其余运算工厂类似...
```

```
//客户端
IFactory *operFactory = new AddFactory();
Operation *oper = operFactory->createOperation();
oper.numA = 1;
oper.numB = 2;
double result = oper->getResult();
```

工厂方法模式实现时，客户端需要决定实例化哪一个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来进行。若要加功能，简单公共模式是要修改工厂类，而工厂模式是修改客户端。

4、抽象工厂模式

Intent

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。

Class Diagram

工厂方法模式 要求产品必须为同一类型，也就是说，BBA 只能生产汽车，要生产其他产品（例如：自行车）是不行的，这显然限制了产品的扩展。为了解决这个问题，抽象工厂模式出现了 - 将产品归类分组，然后将好几组产品构成一族。每个工厂负责生产一族产品，而工厂中的每个方法负责生产一种类型的产品。

这样，客户端只需要创建具体工厂的实例，然后调用工厂对象的工厂方法就可以得到所需要的产品对象。

需要有两个产品 - 汽车和自行车：

```
//创建抽象产品
// 汽车接口
class ICar
{
```

```
public:
    virtual string Name() = 0;  // 汽车名称
};

// 自行车接口
class IBike
{
public:
    virtual string Name() = 0;  // 自行车名称
};
```

```
//创建具体产品
/********** 汽车 **********/
// 奔驰
class BenzCar : public ICar
{
public:
    string Name() {
        return "Benz Car";
    }
};

// 宝马
class BmwCar : public ICar
{
public:
    string Name() {
        return "Bmw Car";
    }
};

// 奥迪
class AudiCar : public ICar
{
public:
    std::string Name() {
        return "Audi Car";
    }
};

/********** 自行车 **********/
// 奔驰
class BenzBike : public IBike
{
public:
    string Name() {
        return "Benz Bike";
    }
```

```
};

// 宝马
class BmwBike : public IBike
{
public:
    string Name() {
        return "Bmw Bike";
    }
};

// 奥迪
class AudiBike : public IBike
{
public:
    string Name() {
        return "Audi Bike";
    }
};
```

```
//创建抽象工厂
class AFactory
{
public:
    enum FACTORY_TYPE {
        BENZ_FACTORY,  // 奔驰工厂
        BMW_FACTORY,  // 宝马工厂
        AUDI_FACTORY  // 奥迪工厂
    };

    virtual ICar* CreateCar() = 0;  // 生产汽车
    virtual IBike* CreateBike() = 0;    // 生产自行车
    static AFactory* CreateFactory(FACTORY_TYPE factory);  // 创建工厂
};
```

```
// 创建工厂
AFactory* AFactory::CreateFactory(FACTORY_TYPE factory)
{
    AFactory *pFactory = NULL;
    switch (factory) {
    case FACTORY_TYPE::BENZ_FACTORY:  // 奔驰工厂
        pFactory = new BenzFactory();
        break;
    case FACTORY_TYPE::BMW_FACTORY:  // 宝马工厂
        pFactory = new BmwFactory();
        break;
    case FACTORY_TYPE::AUDI_FACTORY:  // 奥迪工厂
```

```
        pFactory = new AudiFactory();
        break;
    default:
        break;
    }
    return pFactory;
}
```

```
//创建具体工厂
// 奔驰工厂
class BenzFactory : public AFactory
{
public:
    ICar* CreateCar() {
        return new BenzCar();
    }

    IBike* CreateBike() {
        return new BenzBike();
    }
};

// 宝马工厂
class BmwFactory : public AFactory
{
public:
    ICar* CreateCar() {
        return new BmwCar();
    }

    IBike* CreateBike() {
        return new BmwBike();
    }
};

// 奥迪工厂
class AudiFactory : public AFactory
{
public:
    ICar* CreateCar() {
        return new AudiCar();
    }

    IBike* CreateBike() {
        return new AudiBike();
    }
};
```

```
//客户端
// 奔驰
AFactory *pFactory = AFactory::CreateFactory(AFactory::FACTORY_TYPE::
BENZ_FACTORY);
ICar *pCar = pFactory->CreateCar();
IBike *pBike = pFactory->CreateBike();

cout << "Benz factory - Car: " << pCar->Name() << endl;
cout << "Benz factory - Bike: " << pBike->Name() << endl;
```

四、行为型

1、观察者模式

Intent

多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

Class Diagram

观察者模式的主要角色如下：

1. 抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。
2. 具体主题（Concrete Subject）角色：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。
3. 抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。
4. 具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

```
//抽象观察者
class Observer
{
protected:
    string name;
    Subject *sub;
public:
    Observer(string name, Subject *sub)
    {
```

```
        this->name = name;
        this->sub = sub;
    }
    virtual void update() = 0;
};

//具体的观察者，看股票的
class StockObserver :public Observer
{
public:
    StockObserver(string name, Subject *sub) :Observer(name, sub)
    {
    }
    void update();
};
//具体的观察者，看NBA的
class NBAObserver :public Observer
{
public:
    NBAObserver(string name, Subject *sub) :Observer(name, sub)
    {
    }
    void update();
};

//抽象通知者
class Subject
{
protected:
    list<Observer*> observers;
public:
    string action;
    virtual void attach(Observer*) = 0;
    virtual void detach(Observer*) = 0;
    virtual void notify() = 0;
};

//具体通知者，秘书
class Secretary :public Subject
{
    void attach(Observer *observer)
    {
        observers.push_back(observer);
    }
    void detach(Observer *observer)
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
```

```
        {
            if ((*iter) == observer)
            {
                observers.erase(iter);
            }
            ++iter;
        }
    }
    void notify()
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            (*iter)->update();
            ++iter;
        }
    }
};

void StockObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭股票，装做很认真工作的样子！" << endl;
    }
}

void NBAObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭NBA，装做很认真工作的样子！" << endl;
    }
}

int main()
{
    Subject *dwq = new Secretary(); //创建观察者<br>        //被观察的对
象
    Observer *xs = new NBAObserver("xiaoshuai", dwq);
    Observer *zy = new NBAObserver("zouyue", dwq);
    Observer *lm = new StockObserver("limin", dwq);
       //加入观察队列
    dwq->attach(xs);
    dwq->attach(zy);
    dwq->attach(lm);
```

```
        //事件
    dwq->action = "去吃饭了！";<br>        //通知
    dwq->notify();
    cout << endl;
    dwq->action = "梁所长来了！";
    dwq->notify();
    return 0;
}
```

五、结构性

1、装饰模式

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

```
/*
测试装饰模式实现方法
*/
#include <iostream>
#include <string>
using namespace std;

/*
基类接口
*/
class Component
{
public:
    virtual void operation(const string& msg) = 0;
};

/*
核心业务
*/
class ConcreteComponent : public Component
{
public:
    void operation(const string& msg)
    {
        cout << "I am contrete work." << endl;
        return;
    }
```

```
};

/*
装饰器基类
*/
class Decorator : public Component
{
private:
    Component* com;
public:
    Decorator(Component* c):com(c){}

public:
    void operation(const string& msg)
    {
        com->operation(msg);
    }
};


/*
实际装饰类
*/
class Checker : public Decorator
{
public:
    Checker(Component* c): Decorator(c){}

public:
    void operation(const string& msg)
    {
        cout << "do check operations..." << endl;
        Decorator::operation(msg);
    }
};

/*
实际装饰类
*/
class Logger : public Decorator
{
public:
    Logger(Component* c) : Decorator(c){}

public:
    void operation(const string & msg)
    {
        cout << "do loggs..." << endl;
```

```
        Decorator::operation(msg);
    }
};


int main(int argc, char** argv)
{
    /*
    将不同的装饰类灵活的组合在一起，形成不同的功能
    check-log-operation组合
    */
    Component* op = new Checker(new Logger(new ConcreteComponent()));
    op->operation("This is a test!");

    cout << endl;

    /*
    log-check-operation组合
    */
    Component* op1 = new Logger(new Checker(new ConcreteComponent
()));
    op1->operation("This is a test 2!");
    return 0;
}
```

2、外观模型

Intent

提供了一个统一的接口，用来访问子系统中的一群接口，从而让子系统更容易使用

说白了就是：我们仅仅需调用高层的函数接口。而不用关心高层内部调用是怎样组合底层方法的。更不用关心底层函数是怎样实现的。

设计原则

最少知识原则：只和你的密友谈话。也就是说客户对象所需要交互的对象应当尽可能少。

```
#include<iostream>
using namespace std;
class Scanner
{
public:
    void Scan() { cout<<"词法分析"<<endl; }
```

```
};
class Parser
{
public:
    void Parse() { cout<<"语法分析"<<endl; }
};
class GenMidCode
{
public:
    void GenCode() { cout<<"产生中间代码"<<endl; }
};
class GenMachineCode
{
public:
    void GenCode() { cout<<"产生机器码"<<endl;}
};
//高层接口  Fecade
class Compiler
{
public:
    void Run()
    {
        Scanner scanner;
        Parser parser;
        GenMidCode genMidCode;
        GenMachineCode genMacCode;
        scanner.Scan();
        parser.Parse();
        genMidCode.GenCode();
        genMacCode.GenCode();
    }
};

//client
int main()
{
    Compiler compiler;
    compiler.Run();
    return 0;
}
```