目录

[一、工厂模式 4](#_Toc200450409)

[**适用场景** 4](#_Toc200450410)

[**工厂模式分类** 4](#_Toc200450411)

[1. 简单工厂模式（Simple Factory） 4](#_Toc200450412)

[**思路：** 4](#_Toc200450413)

[**示例：创建不同类型的动物** 4](#_Toc200450414)

[**优点：** 5](#_Toc200450415)

[**缺点：** 5](#_Toc200450416)

[2. 工厂方法模式（Factory Method） 5](#_Toc200450417)

[**思路：** 5](#_Toc200450418)

[**示例：每个工厂只生产一种动物** 5](#_Toc200450419)

[**优点：** 7](#_Toc200450420)

[**缺点：** 7](#_Toc200450421)

[3. 抽象工厂模式（Abstract Factory） 7](#_Toc200450422)

[**思路：** 7](#_Toc200450423)

[**示例：不同风格的动物和植物一起创建** 7](#_Toc200450424)

[**优点：** 9](#_Toc200450425)

[**缺点：** 9](#_Toc200450426)

[二、单例模式 9](#_Toc200450427)

[**核心要点：** 9](#_Toc200450428)

[**示例** 9](#_Toc200450429)

[**输出结果：** 10](#_Toc200450430)

[**优点** 10](#_Toc200450431)

[**缺点** 11](#_Toc200450432)

[三、适配器模式 11](#_Toc200450433)

[**使用场景** 11](#_Toc200450434)

[**现实类比：电源适配器** 11](#_Toc200450435)

[**角色说明** 11](#_Toc200450436)

[四、外观模式 11](#_Toc200450437)

[**外观模式是什么？** 11](#_Toc200450438)

[**类比** 12](#_Toc200450439)

[**外观模式结构图（UML 简化）** 12](#_Toc200450440)

[**优缺点分析** 12](#_Toc200450441)

[**优点：** 12](#_Toc200450442)

[**缺点：** 12](#_Toc200450443)

[**使用场景** 12](#_Toc200450444)

[五、责任链模式 13](#_Toc200450445)

[**适用场景** 13](#_Toc200450446)

[**类比** 13](#_Toc200450447)

[**类图结构（UML）** 13](#_Toc200450448)

[**优点** 13](#_Toc200450449)

[**缺点** 13](#_Toc200450450)

[**总结：** 14](#_Toc200450451)

[六、观察者模式 14](#_Toc200450452)

[**定义** 14](#_Toc200450453)

[**角色组成** 14](#_Toc200450454)

[**类比** 14](#_Toc200450455)

[**优点** 14](#_Toc200450456)

[**缺点** 14](#_Toc200450457)

[七、策略模式 15](#_Toc200450458)

[**结构组成** 15](#_Toc200450459)

[**示例：实现一个计算器，支持加、减、乘三种策略** 15](#_Toc200450460)

[**优点** 16](#_Toc200450461)

[**缺点** 16](#_Toc200450462)

[八、代理模式 17](#_Toc200450463)

[**结构组成** 17](#_Toc200450464)

[**示例：基于角色控制文档访问** 17](#_Toc200450465)

[**优点** 18](#_Toc200450466)

[**缺点** 19](#_Toc200450467)

# 一、工厂模式

工厂设计模式（Factory Pattern）是一种 **创建型设计模式**，用于封装对象的创建过程。其主要目标是 **将对象的创建与使用解耦**，使得代码更具可扩展性和可维护性。

将创建对象的工作交给特定的类，即工厂类

**适用场景**

* 程序需要处理大量具有相同接口的对象。
* 你不希望在客户端代码中暴露具体类的创建逻辑。
* 创建逻辑较复杂，集中封装更清晰。

**工厂模式分类**

1. **简单工厂模式（Simple Factory）**
2. **工厂方法模式（Factory Method）**
3. **抽象工厂模式（Abstract Factory）**

## 1. 简单工厂模式（Simple Factory）

**思路：**

一个工厂类根据参数决定创建哪一个类的实例。

**示例：创建不同类型的动物**

#include <iostream>

#include <memory>

#include <string>

class Animal {

public:

virtual void speak() = 0;

virtual ~Animal() {}

};

class Dog : public Animal {

public:

void speak() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void speak() override {

std::cout << "Meow!" << std::endl;

}

};

// 简单工厂

class AnimalFactory {

public:

static std::unique\_ptr<Animal> createAnimal(const std::string& type) {

if (type == "dog") {

return std::make\_unique<Dog>();

} else if (type == "cat") {

return std::make\_unique<Cat>();

} else {

return nullptr;

}

}

};

int main() {

auto animal = AnimalFactory::createAnimal("dog");

if (animal) animal->speak(); // 输出：Woof!

return 0;

}

**优点：**

* 客户端只依赖接口，不需要了解对象创建细节。

**缺点：**

* 每新增一个类型，就要修改工厂类，不符合**开闭原则**。

## 2. 工厂方法模式（Factory Method）

**思路：**

每个工厂负责创建一种产品。

**示例：每个工厂只生产一种动物**

#include <iostream>

#include <memory>

// 抽象产品

class Animal {

public:

virtual void speak() = 0;

virtual ~Animal() {}

};

// 具体产品

class Dog : public Animal {

public:

void speak() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void speak() override {

std::cout << "Meow!" << std::endl;

}

};

// 工厂接口

class AnimalFactory {

public:

virtual std::unique\_ptr<Animal> createAnimal() = 0;

virtual ~AnimalFactory() {}

};

// 具体工厂

class DogFactory : public AnimalFactory {

public:

std::unique\_ptr<Animal> createAnimal() override {

return std::make\_unique<Dog>();

}

};

class CatFactory : public AnimalFactory {

public:

std::unique\_ptr<Animal> createAnimal() override {

return std::make\_unique<Cat>();

}

};

int main() {

std::unique\_ptr<AnimalFactory> factory = std::make\_unique<DogFactory>();

auto animal = factory->createAnimal();

animal->speak(); // 输出：Woof!

return 0;

}

**优点：**

* 新增产品只需要新增工厂类，符合**开闭原则**。
* 将创建与使用完全解耦。

**缺点：**

* 类的数量增多，维护成本高。

## 3. 抽象工厂模式（Abstract Factory）

**思路：**

创建一组相关对象，而不是一个对象。每个工厂都创建多个产品。

**示例：不同风格的动物和植物一起创建**

#include <iostream>

#include <memory>

// 抽象产品A：动物

class Animal {

public:

virtual void speak() = 0;

virtual ~Animal() {}

};

// 抽象产品B：植物

class Plant {

public:

virtual void grow() = 0;

virtual ~Plant() {}

};

// 产品族1：森林

class ForestDog : public Animal {

public:

void speak() override {

std::cout << "Forest dog howls" << std::endl;

}

};

class ForestTree : public Plant {

public:

void grow() override {

std::cout << "Forest tree grows tall" << std::endl;

}

};

// 产品族2：沙漠

class DesertDog : public Animal {

public:

void speak() override {

std::cout << "Desert dog barks" << std::endl;

}

};

class DesertCactus : public Plant {

public:

void grow() override {

std::cout << "Cactus grows slowly" << std::endl;

}

};

// 抽象工厂

class EcosystemFactory {

public:

virtual std::unique\_ptr<Animal> createAnimal() = 0;

virtual std::unique\_ptr<Plant> createPlant() = 0;

virtual ~EcosystemFactory() {}

};

// 具体工厂：森林

class ForestFactory : public EcosystemFactory {

public:

std::unique\_ptr<Animal> createAnimal() override {

return std::make\_unique<ForestDog>();

}

std::unique\_ptr<Plant> createPlant() override {

return std::make\_unique<ForestTree>();

}

};

// 具体工厂：沙漠

class DesertFactory : public EcosystemFactory {

public:

std::unique\_ptr<Animal> createAnimal() override {

return std::make\_unique<DesertDog>();

}

std::unique\_ptr<Plant> createPlant() override {

return std::make\_unique<DesertCactus>();

}

};

int main() {

std::unique\_ptr<EcosystemFactory> factory = std::make\_unique<ForestFactory>();

auto animal = factory->createAnimal();

auto plant = factory->createPlant();

animal->speak(); // 输出：Forest dog howls

plant->grow(); // 输出：Forest tree grows tall

return 0;

}

**优点：**

* 相关对象的一致性。
* 增加新的产品族容易。

**缺点：**

* 不容易扩展新的“产品种类”（如再加一个“气候”接口），需修改所有工厂类。

# 二、单例模式

**单例模式**是一种创建型设计模式，**保证一个类只有一个实例，并提供一个全局访问点**。

**核心要点：**

1. 私有的构造函数
2. 类定义中包含一个该类的私有静态对象
3. 提供一个公有的静态方法以初始化实例对象

**示例**

#include <iostream>

#include <mutex>

class Singleton {

private:

// 构造函数私有化，防止外部构造

Singleton() {

std::cout << "Constructor called." << std::endl;

}

// 单例指针

static Singleton\* instance;

static std::mutex mtx;

public:

// 获取实例的静态方法

static Singleton\* getInstance() {

if (instance == nullptr) {

std::lock\_guard<std::mutex> lock(mtx); // 加锁保证线程安全

if (instance == nullptr) {

instance = new Singleton();

}

}

return instance;

}

void doSomething() {

std::cout << "Doing something with Singleton." << std::endl;

}

};

// 初始化静态成员

Singleton\* Singleton::instance = nullptr;

std::mutex Singleton::mtx;

// 测试代码

int main() {

Singleton\* s1 = Singleton::getInstance();

s1->doSomething();

Singleton\* s2 = Singleton::getInstance();

std::cout << "Are s1 and s2 the same? " << (s1 == s2 ? "Yes" : "No") << std::endl;

return 0;

}

**输出结果：**

Constructor called.

Doing something with Singleton.

Are s1 and s2 the same? Yes

**优点**

* **减少了内存的开销**：内存中只有一个实例
* **提供了对唯一对象的受控访问，避免对资源的多重访问**

**缺点**

* **一定程度上违背了单一职责原则**：单例类将对象的创建和对象的功能融合在一起
* **没有接口，不能继承**，因此单例类的扩展很困难

# 三、适配器模式

**适配器模式** 是一种结构型设计模式，它的目的是：

**将一个接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起工作的类可以协同工作。**

**使用场景**

适配器模式适用于：

* 已有的类不能修改（比如来自第三方库、老代码等）；
* 新系统期望统一的接口格式；
* 老代码和新接口不一致，需兼容使用。

**现实类比：电源适配器**

* 我们有一个 **中国插头（两扁头）**，但我们在 **欧洲（圆孔插座）**，不能直接用。
* 所以我们需要一个 **适配器（插头转换器）**，让中国插头能插进欧洲插座。

**角色说明**

* **Target（目标接口）**：客户端期望的接口；
* **Adaptee（已有的类）**：已有接口，不能改；
* **Adapter（适配器）**：将 Adaptee 转换为 Target；

# 四、外观模式

**外观模式是什么？**

**外观模式（Facade Pattern）** 是一种结构型设计模式，用于：

为一个复杂子系统提供一个统一的简化接口，让客户端更容易使用这些子系统，而无需直接和底层细节打交道。

本来客户类需要和多个功能类进行交互。引入外观模式后，客户类只需要和服务类交互，由服务类负责与多个功能类交互，实现了客户类和功能类的解耦。

**类比**

想象去电影院看电影：

* 实际操作包括：买票、检票、找座位、调灯光、播放影片等；
* 但你只跟前台说：“我要看《电影A》”；
* 前台（Facade）自动协调多个子系统完成这个流程。

**外观类就像前台服务员**，帮你屏蔽复杂细节，提供简单调用方式。

**外观模式结构图（UML 简化）**

Client

│

▼

Facade ─────► SubsystemA

─────► SubsystemB

─────► SubsystemC

客户端只与 Facade 接触，Facade 与多个子系统进行协作。

**优缺点分析**

**优点：**

|  |  |
| --- | --- |
| 优点 | 说明 |
| 简化接口 | 客户端只需关注高层接口，减少学习和依赖 |
| 降低耦合 | 子系统的变化不会影响客户端 |
| 更好组织代码 | 有助于分层系统架构，例如 UI 与逻辑分离 |

**缺点：**

|  |  |
| --- | --- |
| 缺点 | 说明 |
| 可能封装过度 | 把所有逻辑堆进外观类可能让其变得臃肿 |
| 子系统接口仍需暴露 | 有时客户端还是得访问底层子系统以获取完整控制权 |

**使用场景**

* 提供简单接口给复杂库（如 OpenGL、FFmpeg、Boost）；
* 控制大型子系统启动与关闭（如系统初始化、游戏引擎）；
* 构建三层架构（Controller、Service、DAO）时，为 Controller 提供统一接口。

# 五、责任链模式

**责任链模式** 是一种行为型设计模式，它允许多个对象都有机会处理请求，将这些对象连接成一条链，并沿着这条链传递请求，直到某个对象处理它为止。

**适用场景**

* 有多个对象可以处理同一个请求，且处理者在运行时动态确定；
* 不希望客户端指定具体哪个对象处理；
* 想要将请求的发送者和接收者解耦。

**类比**

假设你有一套公司请假流程：

* 小于 1 天：部门经理审批；
* 小于 3 天：总监审批；
* 更长：总经理审批。

你提交一个请假申请，不需要指定审批人，系统自动按规则一级一级传递处理。

**类图结构（UML）**

Client → Handler → Handler → Handler

↑

ConcreteHandler

每个处理者持有一个指向“下一个处理者”的指针，从而形成一条链。

**优点**

|  |  |
| --- | --- |
| 优点 | 说明 |
| 解耦 | 请求发送者无需知道由谁处理 |
| 灵活性高 | 可动态调整处理链顺序 |
| 符合开闭原则 | 添加新的处理器无需修改其他类 |

**缺点**

|  |  |
| --- | --- |
| 缺点 | 说明 |
| 调试复杂 | 请求可能一直传递不被处理，难以定位问题 |
| 性能可能受影响 | 如果链很长，请求要遍历多个节点 |

**总结：**

责任链模式的核心是 **“谁能处理就处理，不能就丢给下一个”**，它将请求处理过程灵活、解耦地组织成一条链式结构。

# 六、观察者模式

观察者模式（Observer Pattern）是一种**行为型设计模式**，用于在对象之间定义一种**一对多的依赖关系**，使得当一个对象状态发生改变时，其所有依赖者（观察者）都会自动收到通知并更新。

**定义**

观察者模式定义了对象之间的一种**发布-订阅关系**。一个\*\*被观察者（Subject）\*\*维护一个观察者列表，在自身状态发生变化时通知所有观察者。

**角色组成**

1. **Subject（被观察者）**
   * 也叫发布者，提供注册、移除和通知观察者的方法。
   * 状态发生变化时，通知所有观察者。
2. **Observer（观察者）**
   * 定义一个更新接口，用于接收通知。
3. **ConcreteSubject（具体的被观察者）**
   * 维护具体状态，并在状态变化时通知观察者。
4. **ConcreteObserver（具体的观察者）**
   * 实现更新接口，接收到通知后进行相应的处理。

**类比**

你关注了一个名为“科技头条”的微信公众号。每当公众号发布新内容时，你的微信就会收到一条推送通知。这时你就是一个“观察者”，而公众号是“被观察者”。

**优点**

* **低耦合性**：观察者和被观察者之间是松耦合。
* **广播通信机制**：一个状态改变会通知所有观察者。
* **符合开闭原则**：新增观察者不影响已有代码。

**缺点**

* 如果观察者过多，可能会导致**性能问题**。
* 被观察者如果频繁更新，通知观察者的代价较高。
* 观察者之间可能产生**不一致的状态更新顺序问题**。

# 七、策略模式

策略模式（**Strategy Pattern**）是一种行为型设计模式，它定义了一系列算法（策略），将每一个算法封装起来，并使它们可以互相替换。策略模式让算法独立于使用它的客户端独立变化。

**结构组成**

1. **Strategy（策略接口）**：支持所有算法的公共接口
2. **ConcreteStrategy（具体策略类）**：各种算法
3. **Context（上下文）**：使用一个 Strategy 对象来调用算法

**示例：实现一个计算器，支持加、减、乘三种策略**

#include <iostream>

#include <memory>

// 抽象策略类

class Strategy {

public:

virtual int doOperation(int a, int b) const = 0;

virtual ~Strategy() = default;

};

// 具体策略类：加法

class AddStrategy : public Strategy {

public:

int doOperation(int a, int b) const override {

return a + b;

}

};

// 具体策略类：减法

class SubtractStrategy : public Strategy {

public:

int doOperation(int a, int b) const override {

return a - b;

}

};

// 具体策略类：乘法

class MultiplyStrategy : public Strategy {

public:

int doOperation(int a, int b) const override {

return a \* b;

}

};

// 上下文类

class Context {

private:

std::shared\_ptr<Strategy> strategy;

public:

Context(std::shared\_ptr<Strategy> strategy) : strategy(std::move(strategy)) {}

void setStrategy(std::shared\_ptr<Strategy> newStrategy) {

strategy = std::move(newStrategy);

}

int executeStrategy(int a, int b) const {

return strategy->doOperation(a, b);

}

};

// 测试

int main() {

Context context(std::make\_shared<AddStrategy>());

std::cout << "10 + 5 = " << context.executeStrategy(10, 5) << std::endl;

context.setStrategy(std::make\_shared<SubtractStrategy>());

std::cout << "10 - 5 = " << context.executeStrategy(10, 5) << std::endl;

context.setStrategy(std::make\_shared<MultiplyStrategy>());

std::cout << "10 \* 5 = " << context.executeStrategy(10, 5) << std::endl;

return 0;

}

**优点**

* 算法可以自由切换，不影响使用者。
* 避免使用大量的 if-else 或 switch-case。
* 更符合开闭原则（对扩展开放，对修改关闭）。

**缺点**

* 增加了类的数量。
* 使用者必须了解不同策略类的区别才能正确选择。

# 八、代理模式

代理模式是一种**结构型设计模式**，为其他对象提供一个代理以控制对这个对象的访问。**代理对象在客户端和目标对象之间起到中介作用**，可以在不改变原始对象的前提下，添加额外功能，如访问控制、懒加载、日志记录等。

代理类把目标类再次封装，可以在代理类中添加额外功能，客户类使用的是代理类而不是目标类。

**结构组成**

1. **Subject（抽象主题接口）**：定义真实对象与代理对象的公共接口。
2. **RealSubject（真实主题）**：实现真实业务逻辑的类。
3. **Proxy（代理类）**：实现与 RealSubject 相同的接口，并控制对其的访问。

**示例：基于角色控制文档访问**

#include <iostream>

#include <string>

#include <memory>

// 抽象接口

class Document {

public:

virtual void view() const = 0;

virtual void edit() const = 0;

virtual ~Document() = default;

};

// 真实主题：实际的文档

class RealDocument : public Document {

private:

std::string content;

public:

RealDocument(const std::string& text) : content(text) {}

void view() const override {

std::cout << "Viewing document: " << content << std::endl;

}

void edit() const override {

std::cout << "Editing document: " << content << std::endl;

}

};

// 代理类：添加权限控制

class DocumentProxy : public Document {

private:

std::shared\_ptr<RealDocument> realDocument;

std::string userRole;

public:

DocumentProxy(const std::string& text, const std::string& role)

: realDocument(std::make\_shared<RealDocument>(text)), userRole(role) {}

void view() const override {

std::cout << "[Proxy] Checking permission to view...\n";

realDocument->view(); // 所有人都能查看

}

void edit() const override {

std::cout << "[Proxy] Checking permission to edit...\n";

if (userRole == "admin") {

realDocument->edit(); // 只有管理员能编辑

} else {

std::cout << "Access denied: only admin can edit this document.\n";

}

}

};

// 测试

int main() {

DocumentProxy adminDoc("Secret Plan", "admin");

DocumentProxy userDoc("Secret Plan", "user");

std::cout << "\n[Admin Access]\n";

adminDoc.view();

adminDoc.edit();

std::cout << "\n[User Access]\n";

userDoc.view();

userDoc.edit(); // 被拒绝

return 0;

}

**优点**

* **控制访问**：可限制对真实对象的访问。
* **延迟加载**：如虚拟代理在真正需要时才创建对象，节省资源。
* **额外功能**：可添加缓存、权限控制、日志记录等。
* **符合开闭原则**：增强功能无需修改真实类。

**缺点**

* 增加系统复杂性。
* 对于频繁调用的系统，可能带来性能开销（如频繁代理检查）。