高等程序设计 - Qt/C++

第 5 章：设计模式

王培杰

长江大学地球物理与石油资源学院

2025 年 9 月 5 日

目录

目录

什么是设计模式

- 设计模式（Design Pattern）是前人总结出的、在软件开发中反复出现的、行之有效的代码结构和解决方案。
- GoF 在其经典著作《设计模式：可复用面向对象软件的基础》中，将设计模式分为三大类：创建型模式、结构型模式和行为型模式，共 **23 种经典模式**。
- 设计模式不是具体的代码实现，而是一种可复用的设计思想和最佳实践，帮助开发者解决常见的软件设计问题。
- 设计模式提高了代码的可维护性、可扩展性和复用性，促进团队协作和沟通。
- 学习和应用设计模式有助于理解面向对象设计原则，提升软件架构能力。
- 设计模式是软件工程的基石，是软件开发中不可或缺的重要知识。

23 种 GoF 设计模式分类

创建型模式 (5 种)

- 单例模式
- 工厂方法
- 抽象工厂
- 建造者模式
- 原型模式

结构型模式 (7 种)

- 适配器模式
- 桥接模式
- 组合模式
- 装饰器模式
- 外观模式
- 享元模式
- 代理模式

行为型模式 (11 种)

- 责任链模式
- 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
- 访问者模式

目录

什么是创建型模式

创建型模式其实就是帮我们”怎么创建对象更灵活”。如果我们直接用 new 去创建对象，代码会和具体的类绑死，后期想改、想扩展都很麻烦。
创建型模式就是用一些通用套路，把”怎么创建对象”这件事封装起来，让我们不用关心细节，想换对象、加功能都很方便。这样代码更容易维护，也更容易扩展。
这些模式可以让我们按需选择、组合、甚至复用对象，适应各种实际开发场景。
常见的创建型模式有：单例模式（全局唯一对象）、工厂方法模式（延迟决定创建哪种对象）、抽象工厂模式（创建一组相关对象）、建造者模式（一步步构建复杂对象）、原型模式（通过克隆快速生成新对象）等。每种模式都有自己的用武之地，比如只要一个实例、需要一组产品、对象很复杂、或者需要大量相似对象时，都能用上。

创建型模式

创建型模式（Creational Patterns）专注于如何高效、灵活、可控地创建对象。它们将对象的创建与使用解耦，隐藏了实例化的具体细节，便于系统在不影响已有代码的情况下引入新的对象类型。常见的创建型模式包括：

- **单例模式（Singleton Pattern）**：保证一个类只有一个实例，并提供全局访问点。常用于全局配置、日志管理等场景。
- **工厂方法模式（Factory Method Pattern）**：定义一个用于创建对象的接口，让子类决定实例化哪一个类。适用于需要将对象的创建延迟到子类的情况。
- **抽象工厂模式（Abstract Factory Pattern）**：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。适合产品族的创建。
- **建造者模式（Builder Pattern）**：将一个复杂对象的构建与它的表示分离，使同样的构建过程可以创建不同的表示。常用于复杂对象的分步构建。
- **原型模式（Prototype Pattern）**：通过复制已有实例来创建新对象，而不是通过 new 关键字。适合对象创建成本较高或需要大量相似对象的场景。

单例模式 (Singleton Pattern)

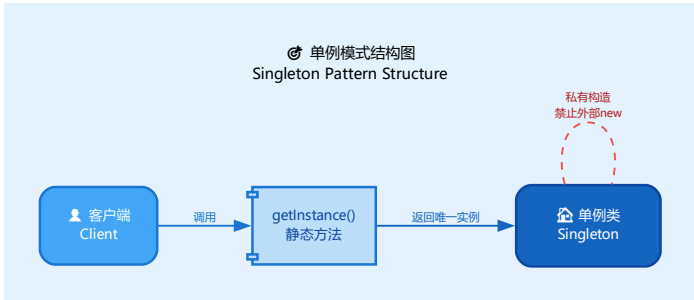
保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用场景

- 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
- 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

例如：全局配置

- 配置管理器
- 日志管理器
- 数据库管理器
- 线程池管理器
- 缓存管理器



单例模式举例说明

打印机管理系统——单例模式 假设我们要开发一个打印机管理系统，要求全局只存在一个打印队列管理器，所有用户和程序都通过它来提交打印任务，避免多台打印机同时混乱工作。**单例模式**的做法是：

- 首先定义一个打印队列管理器类（`PrinterManager`），用于统一管理所有的打印任务。
- 将 `PrinterManager` 设计为单例类，确保全局只有一个实例，并提供静态方法获取该实例。
- 程序的各个部分都通过该单例对象提交打印任务，保证打印任务的有序和统一管理。

单例模式类型

饿汉式单例模式：
在程序启动或类加载时立即创建唯一实例。
优点：实现简单，天然线程安全。
缺点：如果实例创建开销较大且未必用到，可能造成资源浪费。
适用场景：实例始终会被使用。

懒汉式单例模式：
在第一次使用时才创建实例。
优点：按需分配资源，避免不必要的资源浪费。
缺点：多线程下实现复杂，需加锁保证线程安全。
适用场景：实例可能不会被使用。

静态局部变量单例（C++11 推荐）：
利用函数内静态局部变量特性，首次调用时创建实例，C++11 后线程安全。
优点：懒加载，线程安全，代码简洁。
缺点：依赖编译器对 C++11 标准的支持。
适用场景：推荐 C++ 实现单例时使用。

单例模式示例——饿汉式

```
// 饿汉式单例
class PrinterManagerEager {
private:
    queue<string> printQueue;
    mutex mtx;
    PrinterManagerEager() { cout << "饿汉式打印机实例已创建" << endl; }
    PrinterManagerEager(const PrinterManagerEager&) = delete;
    PrinterManagerEager& operator=(const PrinterManagerEager&) = delete;
    static PrinterManagerEager instance; // 静态成员，类加载时创建
public:
    static PrinterManagerEager& getInstance() { return instance; }
    void addJob(const string& job) {
        lock_guard<mutex> lock(mtx);
        printQueue.push(job);
        cout << "[饿汉式] 已添加打印任务: " << job << endl;
    }
    void processJob() {
        lock_guard<mutex> lock(mtx);
        if (!printQueue.empty()) {
            cout << "[饿汉式] 正在打印: " << printQueue.front() << endl;
            printQueue.pop();
        } else {
            cout << "[饿汉式] 当前无待打印任务" << endl;
        }
    }
    int jobCount() {
        lock_guard<mutex> lock(mtx);
        return static_cast<int>(printQueue.size());
    }
};
// 静态成员初始化
PrinterManagerEager PrinterManagerEager::instance;
```

单例模式示例——懒汉式

```
40  // 懒汉式单例
41  class PrinterManagerLazy {
42  private:
43      queue<string> printQueue; mutex mtx;
44      static PrinterManagerLazy* instance;
45      static mutex instanceMtx;
46      PrinterManagerLazy() { cout << "懒汉式打印机实例已
        ↪  创建" << endl; }
47      PrinterManagerLazy(const PrinterManagerLazy&) =
        ↪  delete;
48      PrinterManagerLazy& operator=(const
        ↪  PrinterManagerLazy&) = delete;
49  public:
50      static PrinterManagerLazy* getInstance() {
51          if (instance == nullptr) {
52              lock_guard<mutex> lock(instanceMtx);
53              if (instance == nullptr) { instance = new
                ↪  PrinterManagerLazy(); }
54          }
55          return instance;
56      }
57      void addJob(const string& job) {
58          lock_guard<mutex> lock(mtx);
59          printQueue.push(job);
60          cout << "[懒汉式] 已添加打印任务: " << job <<
            ↪  endl;
61      }
62      void processJob() {
63          lock_guard<mutex> lock(mtx);
64          if (!printQueue.empty()) {
65              cout << "[懒汉式] 正在打印: " <<
                ↪  printQueue.front() << endl;
66              printQueue.pop();
67          } else {
68              cout << "[懒汉式] 当前无待打印任务" <<
                ↪  endl;
69          }
70      }
71      int jobCount() {
72          lock_guard<mutex> lock(mtx);
73          return static_cast<int>(printQueue.size());
74      }
75  };
76  // 静态成员初始化
77  PrinterManagerLazy* PrinterManagerLazy::instance =
    ↪  nullptr;
78  mutex PrinterManagerLazy::instanceMtx;
```

单例模式示例——静态内部类

```
80  // 静态局部变量单例
81  class PrinterManagerStatic {
82  private:
83      queue<string> printQueue; mutex mtx;
84      PrinterManagerStatic() {
85          cout << "静态局部变量式打印机实例已创建" <<
            ↪  endl;
86      }
87      PrinterManagerStatic(const PrinterManagerStatic&)
        ↪  = delete;
88      PrinterManagerStatic& operator=(const
        ↪  PrinterManagerStatic&) = delete;
89  public:
90      static PrinterManagerStatic& getInstance() {
91          static PrinterManagerStatic instance;
92          return instance;
93      }
94      void addJob(const string& job) {
95          lock_guard<mutex> lock(mtx);
96          printQueue.push(job);
97          cout << "[静态局部变量式] 已添加打印任务: " <<
            ↪  job << endl;
98      }
99      void processJob() {
100         lock_guard<mutex> lock(mtx);
101         if (!printQueue.empty()) {
102             cout << "[静态局部变量式] 正在打印: " <<
                ↪  printQueue.front() << endl;
103             printQueue.pop();
104         } else {
105             cout << "[静态局部变量式] 当前无待打印任务"
                ↪  << endl;
106         }
107     }
108     int jobCount() {
109         lock_guard<mutex> lock(mtx);
110         return static_cast<int>(printQueue.size());
111     }
112 };
```

单例模式使用

```
120     // 饿汉式
121     cout << "\n--- 饿汉式单例 ---" << endl;
122     PrinterManagerEager& eager1 =
        ↪  PrinterManagerEager::getInstance();
123     PrinterManagerEager& eager2 =
        ↪  PrinterManagerEager::getInstance();
124     cout << "eager1 和 eager2 是同一实例? " <<
        ↪  ((&eager1 == &eager2) ? "是" : "否") << endl;
125     eager1.addJob("Eager_ 文档 A.pdf");
126     eager2.addJob("Eager_ 图片 B.png");
127     cout << "当前队列任务数: " << eager1.jobCount() <<
        ↪  endl;
128     eager1.processJob();
129     eager2.processJob();
130     eager1.processJob();
131     // 懒汉式
132     cout << "\n--- 懒汉式单例 ---" << endl;
133     PrinterManagerLazy* lazy1 =
        ↪  PrinterManagerLazy::getInstance();
134     PrinterManagerLazy* lazy2 =
        ↪  PrinterManagerLazy::getInstance();
135     cout << "lazy1 和 lazy2 是同一实例? " << ((lazy1
        ↪  == lazy2) ? "是" : "否") << endl;
```

```
136     lazy1->addJob("Lazy_ 报告 C.docx");
137     lazy2->addJob("Lazy_ 表格 D.xlsx");
138     cout << "当前队列任务数: " << lazy1->jobCount() <<
        ↪  endl;
139     lazy1->processJob();
140     lazy2->processJob();
141     lazy1->processJob();
142     // 静态局部变量式
143     cout << "\n--- 静态局部变量式单例 ---" << endl;
144     PrinterManagerStatic& static1 =
        ↪  PrinterManagerStatic::getInstance();
145     PrinterManagerStatic& static2 =
        ↪  PrinterManagerStatic::getInstance();
146     cout << "static1 和 static2 是同一实例? " <<
        ↪  ((&static1 == &static2) ? "是" : "否") <<
        ↪  endl;
147     static1.addJob("Static_ 合同 E.pdf");
148     static2.addJob("Static_ 图片 F.png");
149     cout << "当前队列任务数: " << static1.jobCount()
        ↪  << endl;
150     static1.processJob();
151     static2.processJob();
152     static1.processJob();
```

单例模式优缺点

优点

- 全局唯一实例，节省内存，便于统一管理资源
- 避免频繁创建和销毁实例，减少系统开销
- 提高系统性能，适合需要频繁访问的共享资源
- 提供全局访问点，方便在系统各处获取实例
- 可以延迟实例化（懒汉式），按需分配资源

缺点

- 不利于单元测试，难以模拟和替换单例对象
- 可能隐藏依赖关系，增加代码耦合
- 多线程环境下实现复杂，容易引发线程安全问题
- 滥用单例会导致”伪全局变量”，影响系统扩展性
- 生命周期与程序一致，可能导致资源无法及时释放

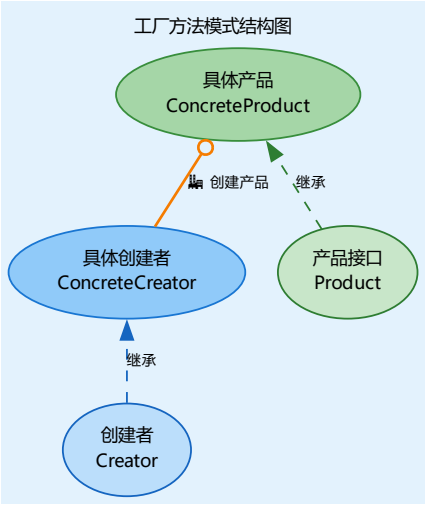
工厂方法模式 (Factory Method Pattern)

工厂方法模式（Factory Method Pattern）是一种创建型设计模式。它定义了一个用于创建对象的接口，但由子类决定要实例化的具体类。工厂方法使一个类的实例化延迟到其子类。

适用场景

- 当一个类不知道它所必须创建的对象的类时。
- 当一个类希望由它的子类来指定它所创建的对象时。
- 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化时。
- 需要灵活切换产品系列，或系统中产品类型经常变化时。

工厂方法模式结构



结构

工厂方法模式主要包含以下角色：

- **抽象产品（Product）**：定义产品的规范，描述产品的主要特性和功能。
- **具体产品（ConcreteProduct）**：实现抽象产品接口，定义具体的产品对象。
- **抽象工厂（Creator）**：声明工厂方法，用于返回一个产品对象。可以定义一个工厂方法的默认实现，返回默认的产品对象。
- **具体工厂（ConcreteCreator）**：重写工厂方法，返回具体的产品实例。

工厂方法模式举例

饮料工厂——工厂方法模式 假设我们要开发一个饮料工厂，能够根据不同的需求生产不同类型的饮料（如可乐、果汁、茶等）。**工厂方法模式**的做法是：

- 首先定义一个饮料产品的抽象类（Beverage），所有饮料都继承自它。
- 再定义一个工厂的抽象类（BeverageFactory），其中声明一个生产饮料的工厂方法（如 createBeverage()）。
- 每种具体饮料（如 Cola、Juice、Tea）都实现 Beverage 类。
- 每种具体饮料工厂（如 ColaFactory、JuiceFactory、TeaFactory）都继承自 BeverageFactory，并实现 createBeverage() 方法，负责生产对应的饮料对象。

优点：客户端只需与工厂接口和产品接口打交道，无需关心具体产品的创建细节，便于扩展和维护。

工厂方法模式示例（一）

```cpp
#include <iostream>
#include <memory>
#include <windows.h>

using namespace std;

class Beverage { // 抽象产品：饮料
public:
    virtual ~Beverage() = default;
    virtual void use() = 0;
};

class ConcreteCola : public Beverage { // 具体产品：可乐
public:
    void use() override { cout << "这是可乐" << endl; }
};

class ConcreteJuice : public Beverage { // 具体产品：果汁
public:
    void use() override { cout << "这是果汁" << endl; }
};

class ConcreteTea : public Beverage { // 具体产品：茶
public:
    void use() override { cout << "这是茶" << endl; }
};

class Creator { // 抽象工厂：饮料工厂
public:
    virtual ~Creator() = default;
    virtual unique_ptr<Beverage> createProduct() = 0;
    void someOperation() { auto product = createProduct(); }
};

class ConcreteCreatorA : public Creator { // 具体工厂：生产可乐的工厂
public:
    unique_ptr<Beverage> createProduct() override { return make_unique<ConcreteCola>(); }
};
```

工厂方法模式示例（二）

```
40  class ConcreteCreatorB : public Creator { // 具体工厂：生产果汁的工厂
41  public:
42      unique_ptr<Beverage> createProduct() override { return make_unique<ConcreteJuice>(); }
43  };
44
45  class ConcreteCreatorC : public Creator { // 具体工厂：生产茶的工厂
46  public:
47      unique_ptr<Beverage> createProduct() override { return make_unique<ConcreteTea>(); }
48  };
49
50  int main() {
51      SetConsoleOutputCP(65001);
52      SetConsoleCP(65001);
53
54      cout << "=== 工厂方法模式演示 ===" << endl;
55
```

```
56      cout << "\n1. 使用可乐工厂:" << endl;
57      unique_ptr<Creator> colaFactory = make_unique<ConcreteCreatorA>();
58      colaFactory->someOperation();
59
60      cout << "\n2. 使用果汁工厂:" << endl;
61      unique_ptr<Creator> juiceFactory = make_unique<ConcreteCreatorB>();
62      juiceFactory->someOperation();
63
64      cout << "\n3. 使用茶工厂:" << endl;
65      unique_ptr<Creator> teaFactory = make_unique<ConcreteCreatorC>();
66      teaFactory->someOperation();
67
68      cin.get();
69      return 0;
70  }
```

工厂方法模式优缺点

优点

- 工厂方法模式将产品创建的逻辑封装在具体工厂类中，客户端无需关心产品创建的具体过程，符合开闭原则。
- 通过工厂方法模式，可以灵活地添加新的产品类型，而无需修改现有的工厂类，符合开闭原则。
- 工厂方法模式可以更好地组织和管理产品创建的逻辑，使得代码更加清晰和易于维护。

缺点

- 工厂方法模式需要创建多个具体工厂类，增加了系统的复杂性。
- 工厂方法模式需要客户端知道具体工厂类的名称，增加了耦合度。
- 工厂方法模式需要客户端知道具体产品的名称，增加了耦合度。

抽象工厂模式 (Abstract Factory Pattern)

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。

适用场景

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 当要强调一系列相关的产品对象的设计以便进行联合使用时。
- 当提供一个产品类库，而只想显示它们的接口而不是实现时。

抽象工厂模式结构

结构

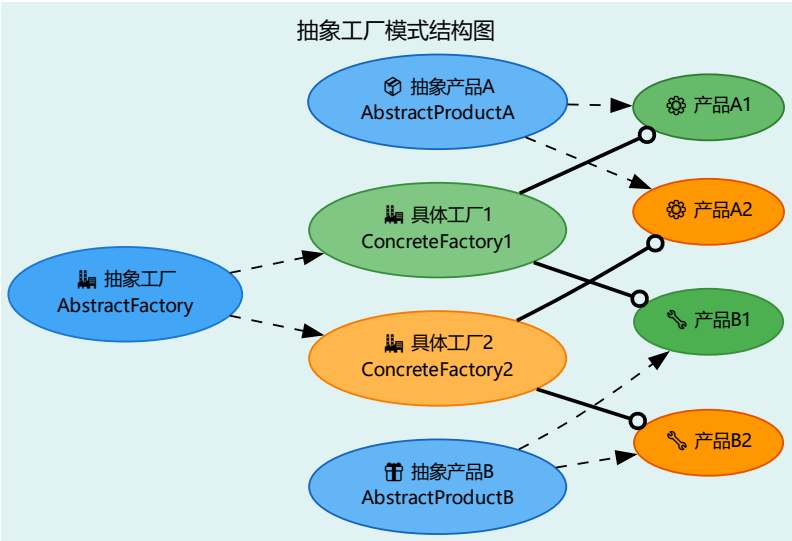
抽象工厂模式主要包含以下角色：

- **抽象产品**：为一类产品提供统一的接口或抽象类，规定产品应有的功能。
- **具体产品**：实现抽象产品接口，代表实际生产的具体产品对象。
- **抽象工厂**：声明创建一组相关产品的接口，通常包含多个创建不同产品的方法。
- **具体工厂**：实现抽象工厂接口，负责实例化一组具体产品。

与工厂方法模式的区别

- 工厂方法模式针对的是**一个产品等级结构**，而抽象工厂模式针对的是**多个产品等级结构**（即一组产品族）。
- 工厂方法模式每次只生产**一种产品**，而抽象工厂模式可以生产**一组相关的产品**。
- 工厂方法模式通常只有一个工厂接口，抽象工厂模式则包含多个产品的工厂方法。
- 抽象工厂模式强调产品族的约束，保证同一工厂生产的产品是相互兼容的。

抽象工厂模式结构


抽象工厂模式结构图
抽象产品A
AbstractProductA
产品A1
产品A2
具体工厂1
ConcreteFactory1
抽象工厂
AbstractFactory
具体工厂2
ConcreteFactory2
产品B1
产品B2
抽象产品B
AbstractProductB

抽象工厂模式举例说明

奶茶店系统——抽象工厂模式 假设我们要开发一个奶茶店系统，奶茶店提供多种不同口味的奶茶（如原味、甜味、巧克力味等），以及不同的饮品配料（如牛奶、糖、巧克力等）。**抽象工厂模式**的实现思路如下：

- 定义奶茶产品的抽象类（如 Tea），所有具体奶茶类（如原味奶茶、甜味奶茶、巧克力味奶茶）都继承自它。
- 定义配料产品的抽象类（如 Topping），所有具体配料类（如牛奶、糖、巧克力）都继承自它。
- 定义抽象工厂（如 TeaFactory），声明创建奶茶和配料的接口（如 createTea() 和 createTopping()）。
- 定义具体工厂（如原味 Factory、甜味 Factory、巧克力味 Factory），分别实现抽象工厂接口，生产一组特定口味的奶茶和配料。

优点：客户端只需依赖抽象工厂和产品接口，无需关心具体产品的创建细节，便于系统的扩展和维护。当需要增加新的产品族时，只需增加新的工厂类即可，无需修改已有代码。例如添加新的口味或配料，只需实现新的工厂类即可，无需修改已有代码。

抽象工厂模式示例（一）

```cpp
#include <iostream>
#include <memory>
#include <windows.h>

using namespace std;

// 抽象产品：奶茶
class Tea {
public:
    virtual ~Tea() = default;
    virtual void drink() = 0;
};

// 抽象产品：配料
class Topping {
public:
    virtual ~Topping() = default;
    virtual void add() = 0;
};
```

```cpp
// 具体产品：原味奶茶
class OriginalTea : public Tea {
public:
    void drink() override {
        cout << "喝一口原味奶茶" << endl;
    }
};
// 具体产品：甜味奶茶
class SweetTea : public Tea {
public:
    void drink() override {
        cout << "喝一口甜味奶茶" << endl;
    }
};
// 具体产品：咸味奶茶
class SaltyTea : public Tea {
public:
    void drink() override {
        cout << "喝一口咸味奶茶" << endl;
    }
};
```

抽象工厂模式示例（二）

```cpp
// 具体产品：牛奶配料
class MilkTopping : public Topping {
public:
    void add() override {
        cout << "加入牛奶配料" << endl;
    }
};
// 具体产品：糖配料
class SugarTopping : public Topping {
public:
    void add() override {
        cout << "加入糖配料" << endl;
    }
};
// 具体产品：巧克力配料
class ChocolateTopping : public Topping {
public:
    void add() override {
        cout << "加入巧克力配料" << endl;
    }
};
```

```cpp
// 抽象工厂：奶茶工厂
class TeaFactory {
public:
    virtual ~TeaFactory() = default;
    virtual unique_ptr<Tea> createTea() = 0;
    virtual unique_ptr<Topping> createTopping() = 0;
};
// 具体工厂：原味工厂
class OriginalFactory : public TeaFactory {
public:
    unique_ptr<Tea> createTea() override {
        return make_unique<OriginalTea>();
    }
    unique_ptr<Topping> createTopping() override {
        return make_unique<MilkTopping>();
    }
};
```

抽象工厂模式示例（三）

```
// 具体工厂：甜味工厂
class SweetFactory : public TeaFactory {
public:
    unique_ptr<Tea> createTea() override {
        return make_unique<SweetTea>();
    }
    unique_ptr<Topping> createTopping() override {
        return make_unique<SugarTopping>();
    }
};
// 具体工厂：巧克力味工厂
class ChocolateFactory : public TeaFactory {
public:
    unique_ptr<Tea> createTea() override {
        return make_unique<ChocolateTea>();
    }
    unique_ptr<Topping> createTopping() override {
        return make_unique<ChocolateTopping>();
    }
};
```

```
// 客户端代码
void clientCode(unique_ptr<TeaFactory> factory) {
    auto tea = factory->createTea();
    auto topping = factory->createTopping();
    cout << "制作奶茶..." << endl;
    topping->add();
    tea->drink();
}
int main() {
    SetConsoleOutputCP(65001);
    SetConsoleCP(65001);
    cout << "=== 抽象工厂模式演示（奶茶店） ===" <<
    ↪  endl;
    cout << "\n使用原味工厂:" << endl;
    clientCode(make_unique<OriginalFactory>());
    cout << "\n使用甜味工厂:" << endl;
    clientCode(make_unique<SweetFactory>());
    cout << "\n使用巧克力味工厂:" << endl;
    clientCode(make_unique<ChocolateFactory>());
    cin.get();
    return 0;
}
```

抽象工厂模式优缺点

优点

- 抽象工厂模式将产品族的创建与使用分离，符合开闭原则。
- 抽象工厂模式可以灵活地添加新的产品族，而无需修改现有的工厂类，符合开闭原则。
- 抽象工厂模式可以更好地组织和管理产品族的创建逻辑，使得代码更加清晰和易于维护。

缺点

- 抽象工厂模式需要创建多个具体工厂类，增加了系统的复杂性。
- 抽象工厂模式需要客户端知道具体工厂类的名称，增加了耦合度。
- 抽象工厂模式需要客户端知道具体产品的名称，增加了耦合度。

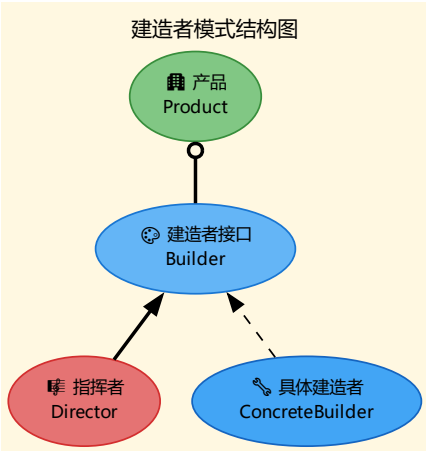
建造者模式 (Builder Pattern)

建造者模式 (Builder Pattern) 是一种创建型设计模式，它将一个复杂对象的构建过程与其最终的表示分离。通过这种方式，同样的构建过程可以创建不同的产品表示。

适用场景

- 当一个对象的构建过程复杂，且需要按照特定顺序分步骤创建时。
- 当需要创建的产品有不同的表示（即产品的组成部件相同，但装配方式或细节不同）时。
- 当构建过程独立于产品的组成部分和装配方式时。

建造者模式结构



角色

- **产品（Product）**：需要被构建的复杂对象，通常由多个部件组成。
- **抽象建造者（Builder）**：规定产品各个部件的创建接口。
- **具体建造者（ConcreteBuilder）**：实现 Builder 接口，负责具体部件的构建和装配。
- **指挥者（Director）**：调用建造者的接口完成产品的构建，负责控制产品的构建顺序。

建造者模式举例说明

游戏系统——角色创建 假设我们要开发一个游戏系统，需要创建不同类型的角色（如战士、法师、刺客），每个角色都有不同的属性（如生命值、攻击力、防御力），以及不同的技能（如战士的冲锋、法师的火球术、刺客的隐身术）。**建造者模式**的实现思路如下：

- 定义角色产品的抽象类（如 `Role`），所有具体角色类（如`战士`、`法师`、`刺客`）都继承自它。
- 定义角色属性的抽象类（如 `RoleAttribute`），所有具体角色属性类（如`生命值`、`攻击力`、`防御力`）都继承自它。
- 定义角色技能的抽象类（如 `RoleSkill`），所有具体角色技能类（如`冲锋`、`火球术`、`隐身术`）都继承自它。
- 定义抽象建造者（如 `RoleBuilder`），声明创建角色及其属性、技能的接口（如 `createRole()`、`createRoleAttribute()`、`createRoleSkill()`）。
- 定义具体建造者（如 `WarriorBuilder`、`MageBuilder`、`AssassinBuilder`），分别实现抽象建造者接口，负责具体角色及其属性、技能的构建和装配。
- 定义指挥者（如 `RoleDirector`），调用建造者的接口完成角色的构建，负责控制角色的构建顺序。

建造者模式示例（角色属性）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <windows.h>

using namespace std;

// 角色属性抽象类
class RoleAttribute {
public:
    virtual ~RoleAttribute() = default;
    virtual void show() const = 0;
};
```

```cpp
// 具体属性
class HPAttribute : public RoleAttribute {
public:
    void show() const override {
        cout << "生命值: 100" << endl;
    }
};
class AttackAttribute : public RoleAttribute {
public:
    void show() const override {
        cout << "攻击力: 30" << endl;
    }
};
class DefenseAttribute : public RoleAttribute {
public:
    void show() const override {
        cout << "防御力: 20" << endl;
    }
};
```

建造者模式示例（角色技能）

```
36  // 角色技能抽象类
37  class RoleSkill {
38  public:
39      virtual ~RoleSkill() = default;
40      virtual void use() const = 0;
41  };
42
43  // 具体技能
44  class ChargeSkill : public RoleSkill {
45  public:
46      void use() const override {
47          cout << "技能: 冲锋" << endl;
48      }
49  };
```

```
50  class FireballSkill : public RoleSkill {
51  public:
52      void use() const override {
53          cout << "技能: 火球术" << endl;
54      }
55  };
56  class StealthSkill : public RoleSkill {
57  public:
58      void use() const override {
59          cout << "技能: 隐身术" << endl;
60      }
61  };
```

建造者模式示例（角色）

```
63  // 角色产品抽象类
64  class Role {
65  public:
66      virtual ~Role() = default;
67      virtual void attack() const = 0;
68      virtual void defend() const = 0;
69      virtual void useSkill() const = 0;
70      virtual void show() const = 0;
71      virtual void
        ↪  addAttribute(unique_ptr<RoleAttribute> attr)
        ↪  = 0;
72      virtual void addSkill(unique_ptr<RoleSkill>
        ↪  skill) = 0;
73  protected:
74      vector<unique_ptr<RoleAttribute>> attributes;
75      vector<unique_ptr<RoleSkill>> skills;
76      void showAttributes() const {
77          for (const auto& attr : attributes)
            ↪  attr->show();
78      }
79      void showSkills() const {
80          for (const auto& skill : skills)
            ↪  skill->use();
81      }
82  };
```

```
84  // 具体角色
85  class Warrior : public Role {
86  public:
87      void addAttribute(unique_ptr<RoleAttribute> attr)
        ↪  override {
88          attributes.push_back(move(attr));
89      }
90      void addSkill(unique_ptr<RoleSkill> skill)
        ↪  override {
91          skills.push_back(move(skill));
92      }
93      void attack() const override { cout << "战士攻击"
        ↪  << endl; }
94      void defend() const override { cout << "战士防御"
        ↪  << endl; }
95      void useSkill() const override { cout << "战士技
        ↪  能" << endl;
96          showSkills(); }
97      void show() const override { cout << "角色: 战士"
        ↪  << endl;
98          showAttributes(); }
99  };
```

建造者模式示例（角色）

```
101  class Mage : public Role {
102  public:
103      void addAttribute(unique_ptr<RoleAttribute> attr)
     ↪  override {
104          attributes.push_back(move(attr));
105      }
106      void addSkill(unique_ptr<RoleSkill> skill)
     ↪  override {
107          skills.push_back(move(skill));
108      }
109      void attack() const override { cout << "法师攻击"
     ↪  << endl; }
110      void defend() const override { cout << "法师防御"
     ↪  << endl; }
111      void useSkill() const override { cout << "法师技
     ↪  能" << endl;
112          showSkills(); }
113      void show() const override { cout << "角色: 法师"
     ↪  << endl;
114          showAttributes(); }
115  };
```

```
117  class Assassin : public Role {
118  public:
119      void addAttribute(unique_ptr<RoleAttribute> attr)
     ↪  override {
120          attributes.push_back(move(attr));
121      }
122      void addSkill(unique_ptr<RoleSkill> skill)
     ↪  override {
123          skills.push_back(move(skill));
124      }
125      void attack() const override { cout << "刺客攻击"
     ↪  << endl; }
126      void defend() const override { cout << "刺客防御"
     ↪  << endl; }
127      void useSkill() const override { cout << "刺客技
     ↪  能" << endl;
128          showSkills(); }
129      void show() const override { cout << "角色: 刺客"
     ↪  << endl;
130          showAttributes(); }
131  };
```

建造者模式示例（建造者）

```cpp
// 抽象建造者
class RoleBuilder {
public:
    virtual ~RoleBuilder() = default;
    virtual void createRole() = 0;
    virtual void createRoleAttribute() = 0;
    virtual void createRoleSkill() = 0;
    virtual unique_ptr<Role> getResult() = 0;
protected:
    unique_ptr<Role> role;
};
```

```cpp
// 具体建造者 - 战士
class WarriorBuilder : public RoleBuilder {
public:
    void createRole() override {
        role = make_unique<Warrior>();
    }
    void createRoleAttribute() override {
        role->addAttribute(make_unique<HPAttribute>());
        role->addAttribute(make_unique<AttackAttribute>());
        role->addAttribute(make_unique<DefenseAttribute>());
    }
    void createRoleSkill() override {
        role->addSkill(make_unique<ChargeSkill>());
    }
    unique_ptr<Role> getResult() override {
        return move(role);
    }
};
```

建造者模式示例（建造者）

```
164 // 具体建造者 - 法师
165 class MageBuilder : public RoleBuilder {
166 public:
167     void createRole() override {
168         role = make_unique<Mage>();
169     }
170     void createRoleAttribute() override {
171         role->addAttribute(make_unique<HPAttribute>());
172         role->addAttribute(make_unique<AttackAttribute>());
173     }
174     void createRoleSkill() override {
175         role->addSkill(make_unique<FireballSkill>());
176     }
177     unique_ptr<Role> getResult() override {
178         return move(role);
179     }
180 };
```

```
182 // 具体建造者 - 刺客
183 class AssassinBuilder : public RoleBuilder {
184 public:
185     void createRole() override {
186         role = make_unique<Assassin>();
187     }
188     void createRoleAttribute() override {
189         role->addAttribute(make_unique<HPAttribute>());
190         role->addAttribute(make_unique<AttackAttribute>());
191         role->addAttribute(make_unique<DefenseAttribute>());
192     }
193     void createRoleSkill() override {
194         role->addSkill(make_unique<StealthSkill>());
195     }
196     unique_ptr<Role> getResult() override {
197         return move(role);
198     }
199 };
```

建造者模式示例（指挥者）

```
201  // 指挥者
202  class RoleDirector {
203  public:
204      void construct(RoleBuilder& builder) {
205          builder.createRole();
206          builder.createRoleAttribute();
207          builder.createRoleSkill();
208      }
209  };
210
211  int main() {
212      SetConsoleOutputCP(65001);
213      SetConsoleCP(65001);
214      cout << "=== 建造者模式演示（游戏角色） ===" <<
     ↪  endl;
215      RoleDirector director;
216      // 构建战士
217      WarriorBuilder warriorBuilder;
218      director.construct(warriorBuilder);
219      auto warrior = warriorBuilder.getResult();
```

```
220      cout << "\n构建的角色:" << endl;
221      warrior->show();
222      // 构建法师
223      MageBuilder mageBuilder;
224      director.construct(mageBuilder);
225      auto mage = mageBuilder.getResult();
226      cout << "\n构建的角色:" << endl;
227      mage->show();
228      // 构建刺客
229      AssassinBuilder assassinBuilder;
230      director.construct(assassinBuilder);
231      auto assassin = assassinBuilder.getResult();
232      cout << "\n构建的角色:" << endl;
233      assassin->show();
234      cin.get();
235      return 0;
236  }
```

建造者模式优缺点

优点

- 将产品的构建过程与产品的表示分离，支持相同构建过程生成不同产品。
- 细化产品的构建步骤，便于控制和复用构建流程。
- 有利于代码解耦，提高系统的灵活性和可扩展性。
- 便于增加新的产品类型和构建细节。

缺点

- 产品的组成部分如果变化较多，建造者类数量会增加，系统复杂度提升。
- 适用于产品结构较为复杂且构建过程稳定的场景，对于简单产品不适用。

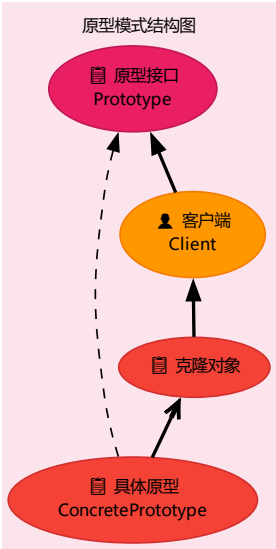
原型模式 (Prototype Pattern)

原型模式 (Prototype Pattern) 是一种创建型设计模式，它通过拷贝现有对象来创建新对象，而不是通过实例化新对象。原型模式允许在运行时动态创建对象，而无需依赖于具体的类。

适用场景

- 当一个系统应该独立于它的产品创建、构成和表示时。
- 当要实例化的类是在运行时刻指定时，例如通过动态装载。
- 当避免创建一个与产品类层次平行的工厂类层次时。
- 当一个类的实例只能有几个不同状态组合中的一种时。

原型模式结构



结构

- **原型接口 (Prototype)**：定义克隆自身的抽象接口，通常包含 `clone()` 方法。
- **具体原型 (ConcretePrototype)**：实现原型接口，完成对象自身的深拷贝或浅拷贝。
- **客户端 (Client)**：通过调用原型的 `clone()` 方法，复制已有对象并生成新实例，而无需依赖具体类。
- **克隆对象 (ConcretePrototype2)**：克隆出来的对象。

原型模式举例说明

游戏系统——道具批量生成 假设我们要开发一个游戏系统，需要批量生成不同类型的道具(如武器、防具、药水)，每种道具有不同的属性（如攻击力、防御力、恢复量）和特殊效果(如中毒、加速、回血)。这些道具结构复杂，初始化参数繁多，手动创建既繁琐又容易出错。**原型模式**的实现思路如下：

- 定义道具产品的抽象类（如 `Item`），所有具体道具类（如武器、防具、药水）都继承自它。
- 定义道具属性的抽象类（如 `ItemAttribute`），所有具体属性类（如攻击力、防御力、恢复量）都继承自它。
- 定义道具效果的抽象类（如 `ItemEffect`），所有具体效果类（如中毒、加速、回血）都继承自它。
- 在 `Item` 类中声明 `clone()` 方法，作为原型接口。
- 各具体道具类实现 `clone()` 方法，实现自身的深拷贝或浅拷贝。
- 先创建一个标准的道具对象作为原型，后续通过调用其 `clone()` 方法，快速生成大量相似但内容不同的新道具对象，并根据需要调整属性或效果。

原型模式示例（道具属性）

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <unordered_map>
#include <windows.h>

using namespace std;

// 道具属性抽象类

class ItemAttribute {
public:
    virtual ~ItemAttribute() = default;
    virtual unique_ptr<ItemAttribute> clone() const = 0;
    virtual void show() const = 0;
};
```

```cpp
// 具体属性：攻击力
class AttackAttribute : public ItemAttribute {
    int value;
public:
    AttackAttribute(int v) : value(v) {}
    unique_ptr<ItemAttribute> clone() const override {
        return make_unique<AttackAttribute>(*this);
    }
    void show() const override {
        cout << "攻击力: " << value << " ";
    }
};
```

原型模式示例（道具属性）

```cpp
// 具体属性：防御力
class DefenseAttribute : public ItemAttribute {
    int value;
public:
    DefenseAttribute(int v) : value(v) {}
    unique_ptr<ItemAttribute> clone() const override {
        return make_unique<DefenseAttribute>(*this);
    }
    void show() const override {
        cout << "防御力: " << value << " ";
    }
};
```

```cpp
// 具体属性：恢复量
class HealAttribute : public ItemAttribute {
    int value;
public:
    HealAttribute(int v) : value(v) {}
    unique_ptr<ItemAttribute> clone() const override {
        return make_unique<HealAttribute>(*this);
    }
    void show() const override {
        cout << "恢复量: " << value << " ";
    }
};
```

原型模式示例（道具效果）

```cpp
// 道具效果抽象类
class ItemEffect {
public:
    virtual ~ItemEffect() = default;
    virtual unique_ptr<ItemEffect> clone() const = 0;
    virtual void show() const = 0;
};

// 具体效果：中毒
class PoisonEffect : public ItemEffect {
public:
    unique_ptr<ItemEffect> clone() const override {
        return make_unique<PoisonEffect>(*this);
    }
    void show() const override {
        cout << "效果：中毒 ";
    }
};
```

```cpp
// 具体效果：加速
class SpeedEffect : public ItemEffect {
public:
    unique_ptr<ItemEffect> clone() const override {
        return make_unique<SpeedEffect>(*this);
    }
    void show() const override {
        cout << "效果：加速 ";
    }
};

// 具体效果：回血
class HealEffect : public ItemEffect {
public:
    unique_ptr<ItemEffect> clone() const override {
        return make_unique<HealEffect>(*this);
    }
    void show() const override {
        cout << "效果：回血 ";
    }
};
```

原型模式示例（道具）

```cpp
// 道具抽象类
class Item {
public:
    virtual ~Item() = default;
    virtual unique_ptr<Item> clone() const = 0;
    virtual void showInfo() const = 0;
};

// 具体道具：武器
class Weapon : public Item {
    string name;
    vector<unique_ptr<ItemAttribute>> attributes;
    vector<unique_ptr<ItemEffect>> effects;
public:
    Weapon(const string& n) : name(n) {}
    Weapon(const Weapon& other) : name(other.name) {
        for (const auto& attr : other.attributes)
            attributes.push_back(attr->clone());
        for (const auto& eff : other.effects)
            effects.push_back(eff->clone());
    }
    void addAttribute(unique_ptr<ItemAttribute> attr) {
        attributes.push_back(move(attr));
    }
    void addEffect(unique_ptr<ItemEffect> eff) {
        effects.push_back(move(eff));
    }
    unique_ptr<Item> clone() const override {
        return make_unique<Weapon>(*this);
    }
    void showInfo() const override {
        cout << "武器: " << name << " | ";
        for (const auto& attr : attributes)
            attr->show();
        for (const auto& eff : effects) eff->show();
        cout << endl;
    }
    void setName(const string& n) { name = n; }
};
```

原型模式示例（道具）

```cpp
// 具体道具：防具
class Armor : public Item {
    string name;
    vector<unique_ptr<ItemAttribute>> attributes;
    vector<unique_ptr<ItemEffect>> effects;
public:
    Armor(const string& n) : name(n) {}
    Armor(const Armor& other) : name(other.name) {
        for (const auto& attr : other.attributes)
            attributes.push_back(attr->clone());
        for (const auto& eff : other.effects)
            effects.push_back(eff->clone());
    }
    void addAttribute(unique_ptr<ItemAttribute> attr) {
        attributes.push_back(move(attr));
    }
    void addEffect(unique_ptr<ItemEffect> eff) {
        effects.push_back(move(eff));
    }
    unique_ptr<Item> clone() const override {
        return make_unique<Armor>(*this);
    }
    void showInfo() const override {
        cout << "防具: " << name << " | ";
        for (const auto& attr : attributes)
            attr->show();
        for (const auto& eff : effects) eff->show();
        cout << endl;
    }
    void setName(const string& n) { name = n; }
};
```

原型模式示例（道具）

```
};

// 具体道具：药水
class Potion : public Item {
    string name;
    vector<unique_ptr<ItemAttribute>> attributes;
    vector<unique_ptr<ItemEffect>> effects;
public:
    Potion(const string& n) : name(n) {}
    Potion(const Potion& other) : name(other.name) {
        for (const auto& attr : other.attributes)
            attributes.push_back(attr->clone());
        for (const auto& eff : other.effects)
            effects.push_back(eff->clone());
    }
    void addAttribute(unique_ptr<ItemAttribute> attr) {
        attributes.push_back(move(attr));
    }
    void addEffect(unique_ptr<ItemEffect> eff) {
        effects.push_back(move(eff));
    }
    unique_ptr<Item> clone() const override {
        return make_unique<Potion>(*this);
    }
    void showInfo() const override {
        cout << "药水: " << name << " | ";
        for (const auto& attr : attributes)
            attr->show();
        for (const auto& eff : effects) eff->show();
        cout << endl;
    }
    void setName(const string& n) { name = n; }
};
```

原型模式示例（原型管理）

```cpp
// 原型管理器
class ItemPrototypeManager {
    unordered_map<string, unique_ptr<Item>>
    ↪ prototypes;
public:
    void registerPrototype(const string& key,
    ↪ unique_ptr<Item> prototype) {
        prototypes[key] = move(prototype);
    }
    unique_ptr<Item> create(const string& key) {
        if (prototypes.count(key))
            return prototypes[key]->clone();
        return nullptr;
    }
};
```

```cpp
    ItemPrototypeManager manager;

    // 1. 注册原型
    cout << "\n1. 注册原型:" << endl;
    auto sword = make_unique<Weapon>("铁剑");
    sword->addAttribute(make_unique<AttackAttribute>(15));
    sword->addEffect(make_unique<PoisonEffect>());
    manager.registerPrototype("sword", move(sword));

    auto shield = make_unique<Armor>("木盾");
    shield->addAttribute(make_unique<DefenseAttribute>(10));
    shield->addEffect(make_unique<SpeedEffect>());
    manager.registerPrototype("shield",
    ↪ move(shield));

    auto potion = make_unique<Potion>("小型治疗药水");
    potion->addAttribute(make_unique<HealAttribute>(30));
    potion->addEffect(make_unique<HealEffect>());
    manager.registerPrototype("potion",
    ↪ move(potion));
```

原型模式示例（原型使用）

```
239     // 2. 克隆原型
240     cout << "\n2. 克隆原型:" << endl;
241
242     // 克隆武器
243     auto sword1 = manager.create("sword");
244     if (sword1) {
245
            ↪  dynamic_cast<Weapon*>(sword1.get())->setName("铁
            ↪  剑 +1");
246         cout << "克隆武器 1: ";
247         sword1->showInfo();
248     }
249     auto sword2 = manager.create("sword");
250     if (sword2) {
251
            ↪  dynamic_cast<Weapon*>(sword2.get())->setName("铁
            ↪  剑 +2");
252         cout << "克隆武器 2: ";
253         sword2->showInfo();
254     }
```

```
256     // 克隆防具
257     auto shield1 = manager.create("shield");
258     if (shield1) {
259
            ↪  dynamic_cast<Armor*>(shield1.get())->setName("木
            ↪  盾 • 强化");
260         cout << "克隆防具: ";
261         shield1->showInfo();
262     }
263
264     // 克隆药水
265     auto potion1 = manager.create("potion");
266     if (potion1) {
267
            ↪  dynamic_cast<Potion*>(potion1.get())->setName("大
            ↪  型治疗药水");
268         cout << "克隆药水: ";
269         potion1->showInfo();
270     }
```

原型模式优缺点

优点

- 可以在运行时动态创建对象，避免类创建时的复杂依赖。
- 创建新对象的效率高，适合需要大量相似对象的场景。
- 简化对象的创建过程，减少子类的数量。
- 可以方便地扩展和修改原型对象，灵活性高。

缺点

- 需要实现复杂对象的深拷贝，可能增加开发难度。
- 每个类都要配备克隆方法，代码维护成本较高。
- 对包含循环引用或复杂资源的对象克隆时容易出错。

目录

什么是结构型模式

结构型模式关注的是”如何将类和对象进行组合”，以构建更大、更灵活的系统结构。它们通过合理地组织类和对象之间的关系，使系统更易于扩展和维护。结构型模式强调”组合优于继承”，通过组合已有的类和对象来获得新的功能，而不是单纯依赖继承。
这些模式可以帮助我们简化系统结构、优化类之间的协作、提高代码复用性，并且让系统更容易适应变化。
常见的结构型模式有：适配器模式（接口转换）、桥接模式（抽象分离）、组合模式（树形结构）、装饰器模式（动态扩展）、外观模式（简化接口）、享元模式（对象共享）、代理模式（访问控制）等。
每种模式都有其独特的应用场景，比如需要兼容不同接口、动态添加功能、统一子系统入口、节省内存、控制访问等，都可以用结构型模式来解决。

结构型模式简介

结构型模式（Structural Patterns）主要关注如何将类和对象进行组合，从而形成更大、更灵活的结构。它们通过合理地组织类和对象之间的关系，使系统更易于扩展和维护。结构型模式的核心思想是” 组合优于继承”，强调通过组合已有的类和对象来获得新的功能，而不是通过继承来扩展行为。

- **适配器模式（Adapter Pattern）**：将一个类的接口转换成客户期望的另一个接口，使原本由于接口不兼容而不能一起工作的类可以协同工作。
- **桥接模式（Bridge Pattern）**：将抽象部分与实现部分分离，使它们可以独立变化，适用于系统需要在多个维度上扩展时。
- **组合模式（Composite Pattern）**：将对象组合成树形结构以表示” 部分-整体” 的层次结构，使客户端对单个对象和组合对象的使用具有一致性。
- **装饰器模式（Decorator Pattern）**：动态地给对象添加一些额外的职责，就增加功能来说，装饰器模式比生成子类更为灵活。
- **外观模式（Facade Pattern）**：为子系统中的一组接口提供一个统一的高层接口，使子系统更易使用。
- **享元模式（Flyweight Pattern）**：运用共享技术有效地支持大量细粒度对象的复用，节省内

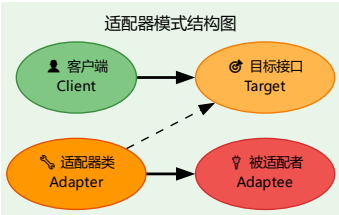
适配器模式 (Adapter Pattern)

适配器模式（Adapter Pattern）是一种重要的结构型设计模式。它的核心思想是：**将一个类的接口转换成客户期望的另一个接口**，使原本由于接口不兼容而不能一起工作的类可以协同工作。适配器模式通常用于系统集成、第三方库适配、老系统升级等场景，极大地提升了系统的灵活性和可扩展性。

适用场景

- 希望复用某个现有类，但其接口与当前系统不兼容。
- 需要创建一个可复用的类，使其能与其他接口各异的类协同工作。
- 不希望修改已有代码的前提下，适配第三方库或遗留系统。
- 系统需要使用一些现有的类，但这些类的接口不符合你的需求。
- 希望通过引入适配器，统一多个类的接口，简化客户端调用。

适配器模式结构



结构

- **目标接口（Target）**：客户端所期望的接口。
- **被适配者（Adaptee）**：现有的、需要适配的类，其接口与目标接口不兼容。
- **适配器（Adapter）**：通过包装一个被适配者对象，将其接口转换为目标接口，使其能够被客户端使用。

适配器模式举例说明

音频播放器系统——适配器模式 假设我们有一个音频播放器系统，原本只支持播放 MP3 格式的音频文件（`Mp3Player`），现在需要支持播放 MP4 和 VLC 格式的文件，但不希望修改原有的播放器代码。此时可以通过适配器模式来实现兼容。**适配器模式**的实现思路如下：

- 定义一个目标接口（如 `MediaPlayer`），声明统一的播放方法（如 `play(fileType, fileName)`）。
- 已有的 `Mp3Player` 类实现 `MediaPlayer` 接口，只能播放 MP3 文件。
- 新增 `AdvancedMediaPlayer` 接口，声明 `playMp4()` 和 `playVlc()` 等高级播放方法。
- 分别实现 `AdvancedMediaPlayer` 接口的 `Mp4Player` 和 `VlcPlayer` 类，分别支持 MP4 和 VLC 格式。
- 创建 `MediaAdapter` 适配器类，实现 `MediaPlayer` 接口，内部组合 `AdvancedMediaPlayer`，根据文件类型委托给对应的高级播放器。
- 客户端只依赖 `MediaPlayer` 接口，通过 `MediaAdapter` 即可无缝播放多种格式的音频文件，无需修改原有播放器代码。

适配器模式示例（目标接口）

```cpp
// 目标接口 - 媒体播放器
class MediaPlayer {
public:
    virtual ~MediaPlayer() = default;
    virtual void play(const string& audioType, const string& fileName) = 0;
};

// 高级媒体播放器接口
class AdvancedMediaPlayer {
public:
    virtual ~AdvancedMediaPlayer() = default;
    virtual void playVlc(const string& fileName) = 0;
    virtual void playMp4(const string& fileName) = 0;
};

// 具体实现类 - VLC 播放器
class VlcPlayer : public AdvancedMediaPlayer {
public:
    void playVlc(const string& fileName) override {
        cout << "使用 VLC 播放器播放: " << fileName << endl;
    }
    void playMp4(const string& fileName) override {
        // VLC 播放器不支持 MP4
    }
};

// 具体实现类 - MP4 播放器
class Mp4Player : public AdvancedMediaPlayer {
public:
    void playVlc(const string& fileName) override {
        // MP4 播放器不支持 VLC
    }
    void playMp4(const string& fileName) override {
        cout << "使用 MP4 播放器播放: " << fileName << endl;
    }
};
```

适配器模式示例（适配器）

```cpp
// 适配器类，实现 MediaPlayer 接口，内部组合 AdvancedMediaPlayer
class MediaAdapter : public MediaPlayer {
private:
    unique_ptr<AdvancedMediaPlayer> advancedPlayer;
public:
    MediaAdapter(const string& audioType) {
        if (audioType == "vlc") {
            advancedPlayer = make_unique<VlcPlayer>();
        } else if (audioType == "mp4") {
            advancedPlayer = make_unique<Mp4Player>();
        }
    }
    void play(const string& audioType, const string& fileName) override {
        if (audioType == "vlc") {
            advancedPlayer->playVlc(fileName);
        } else if (audioType == "mp4") {
            advancedPlayer->playMp4(fileName);
        }
    }
};

// 已有的 Mp3Player 类，实现 MediaPlayer 接口，只能播放 MP3
class Mp3Player : public MediaPlayer {
public:
    void play(const string& audioType, const string& fileName) override {
        if (audioType == "mp3") {
            cout << "使用内置播放器播放 MP3: " << fileName << endl;
        } else {
            cout << "Mp3Player 不支持的格式: " << audioType << endl;
        }
    }
};
```

适配器模式示例（客户端）

```
// 客户端只依赖 MediaPlayer 接口，通过 MediaAdapter 适配多种格式
class AudioPlayer : public MediaPlayer {
private:
    unique_ptr<MediaAdapter> mediaAdapter;
    unique_ptr<Mp3Player> mp3Player;
public:
    AudioPlayer() {
        mp3Player = make_unique<Mp3Player>();
    }
    void play(const string& audioType, const string& fileName) override {
        if (audioType == "mp3") {
            mp3Player->play(audioType, fileName);
        } else if (audioType == "vlc"  audioType == "mp4") {
            mediaAdapter = make_unique<MediaAdapter>(audioType);
            mediaAdapter->play(audioType, fileName);
        } else {
            cout << "不支持的音频格式: " << audioType << endl;
        }
    }
};
```

```
int main() {
    SetConsoleOutputCP(65001);
    SetConsoleCP(65001);
    cout << "=== 适配器模式演示 ===" << endl;
    auto audioPlayer = make_unique<AudioPlayer>();

    cout << "\n1. 播放 MP3 文件:" << endl;
    audioPlayer->play("mp3", "beyond_the_horizon.mp3");

    cout << "\n2. 播放 MP4 文件:" << endl;
    audioPlayer->play("mp4", "alone.mp4");

    cout << "\n3. 播放 VLC 文件:" << endl;
    audioPlayer->play("vlc", "far_far_away.vlc");

    cout << "\n4. 播放不支持的文件格式:" << endl;
    audioPlayer->play("avi", "mind_me.avi");

    cin.get();
    return 0;
}
```

适配器模式优缺点

优点

- 提高了类的复用性和系统的灵活性，无需修改原有代码即可兼容新接口。
- 遵循开闭原则，能够在不改变已有代码的基础上扩展系统功能。
- 客户端代码与具体实现解耦，便于后续维护和升级。

缺点

- 增加了系统的复杂度，过多使用适配器会导致系统结构混乱，难以维护。
- 适配器编写不当可能导致性能损耗或隐藏系统设计缺陷。

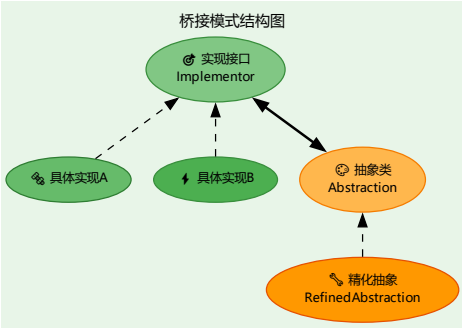
桥接模式（Bridge Pattern）

桥接模式（Bridge Pattern）是一种结构型设计模式，它将抽象部分与其实现部分分离，使它们可以独立变化。桥接模式通过组合的方式将抽象与实现解耦，避免了继承带来的类爆炸问题，提高了系统的可扩展性和灵活性。

适用场景

- 希望抽象和实现可以独立扩展，且系统不希望两者之间有固定的绑定关系。
- 抽象和实现都有可能通过子类化进行扩展，并希望通过组合而非继承来实现它们之间的关联。
- 需要对抽象的实现部分进行修改时，不影响客户端代码。
- 希望在多个对象间共享实现，但又不希望客户知道实现的细节。

桥接模式结构



结构

- **抽象类（Abstraction）**：定义抽象接口，包含对实现部分的引用。
- **实现接口（Implementor）**：定义实现接口，包含具体操作的实现。
- **具体实现（Concrete Implementor）**：实现实现接口，提供具体操作的实现。
- **精化抽象（Refined Abstraction）**：扩展抽象类，提供更具体的操作。

桥接模式举例说明

音频播放器系统——桥接模式 假设我们有一个音频播放器系统，支持多种操作系统（如 Windows、Linux、Mac），每种操作系统下都可以播放不同格式的音频文件（如 MP3、MP4、VLC）。如果直接用继承方式实现，每增加一种操作系统或音频格式都需要大量子类，系统会变得非常复杂。此时可以通过**桥接模式**来解耦”操作系统平台”与”音频格式”的变化。**桥接模式**的实现思路如下：

- 定义 `AudioFormat` 实现接口，声明如 `play(const std::string& fileName)` 等方法，具体实现如 `Mp3Format`、`Mp4Format`、`VlcFormat` 分别实现不同格式的播放。
- 定义 `Player` 抽象类，内部持有 `AudioFormat` 接口的指针或引用，声明如 `play(const std::string& fileName)` 等方法。
- 具体的 `Player` 子类如 `WindowsPlayer`、`LinuxPlayer`、`MacPlayer`，在 `play` 方法中调用 `AudioFormat` 的实现，实现平台与格式的解耦。
- 客户端可以灵活组合不同的 `Player` 和 `AudioFormat`，如”Windows+MP3”、”Linux+MP4”等，无需为每种组合创建新子类。

桥接模式示例（实现接口）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <windows.h>

using namespace std;

// 音频格式实现接口
class AudioFormat {
public:
    virtual ~AudioFormat() = default;
    virtual void play(const string& fileName) = 0;
};

// 具体实现：MP3 格式
class Mp3Format : public AudioFormat {
public:
    void play(const string& fileName) override {
        cout << "播放 MP3 音频文件: " << fileName <<
        ↪ endl;
    }
};

// 具体实现：MP4 格式
class Mp4Format : public AudioFormat {
public:
    void play(const string& fileName) override {
        cout << "播放 MP4 音频文件: " << fileName <<
        ↪ endl;
    }
};

// 具体实现：VLC 格式
class VlcFormat : public AudioFormat {
public:
    void play(const string& fileName) override {
        cout << "播放 VLC 音频文件: " << fileName <<
        ↪ endl;
    }
};
```

桥接模式示例（抽象类）

```cpp
// 抽象播放器
class Player {
protected:
    unique_ptr<AudioFormat> format;
public:
    Player(unique_ptr<AudioFormat> fmt) :
    ↪  format(move(fmt)) {}
    virtual ~Player() = default;
    virtual void play(const string& fileName) = 0;
};

// Windows 平台播放器
class WindowsPlayer : public Player {
public:
    WindowsPlayer(unique_ptr<AudioFormat> fmt) :
    ↪  Player(move(fmt)) {}
    void play(const string& fileName) override {
        cout << "[Windows 平台] ";
        format->play(fileName);
    }
};
```

```cpp
// Linux 平台播放器
class LinuxPlayer : public Player {
public:
    LinuxPlayer(unique_ptr<AudioFormat> fmt) :
    ↪  Player(move(fmt)) {}
    void play(const string& fileName) override {
        cout << "[Linux 平台] ";
        format->play(fileName);
    }
};

// Mac 平台播放器
class MacPlayer : public Player {
public:
    MacPlayer(unique_ptr<AudioFormat> fmt) :
    ↪  Player(move(fmt)) {}
    void play(const string& fileName) override {
        cout << "[Mac 平台] ";
        format->play(fileName);
    }
};
```

桥接模式示例（客户端）

```
int main() {
    // 设置控制台编码为 UTF-8，解决中文显示问题
    SetConsoleOutputCP(65001);
    SetConsoleCP(65001);

    cout << "=== 桥接模式演示 ===" << endl;

    cout << "\n1. Windows 平台播放 MP3:" << endl;
    auto winMp3 =
    ↪  make_unique<WindowsPlayer>(make_unique<Mp3Format>());
    winMp3->play("beyond_the_horizon.mp3");

    cout << "\n2. Linux 平台播放 MP4:" << endl;
    auto linuxMp4 =
    ↪  make_unique<LinuxPlayer>(make_unique<Mp4Format>());
    linuxMp4->play("alone.mp4");

    cout << "\n3. Mac 平台播放 VLC:" << endl;
    auto macVlc =
    ↪  make_unique<MacPlayer>(make_unique<VlcFormat>());
    macVlc->play("far_far_away.vlc");

    cout << "\n4. Windows 平台播放 VLC:" << endl;
    auto winVlc =
    ↪  make_unique<WindowsPlayer>(make_unique<VlcFormat>());
    winVlc->play("mind_me.vlc");

    cin.get();
    return 0;
}
```

桥接模式优缺点

优点

- 分离抽象与实现，使它们可以独立变化，降低耦合度。
- 扩展性好，新增抽象或实现部分都很方便，无需修改原有代码。
- 实现细节对客户透明，客户端只需关注抽象接口。
- 有助于系统分层。
- 可以动态组合不同的实现，灵活应对多维度变化。

缺点

- 增加系统设计的复杂度，理解和实现成本较高。
- 桥接模式引入了更多的抽象层，可能导致代码结构变得更加抽象。
- 对于简单场景，使用桥接模式可能会导致”杀鸡用牛刀”。

组合模式（Composite Pattern）

组合模式是一种结构型设计模式，它将对象组织成树形结构以表示” 部分-整体” 的层次关系。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，无需关心它们的具体类型。

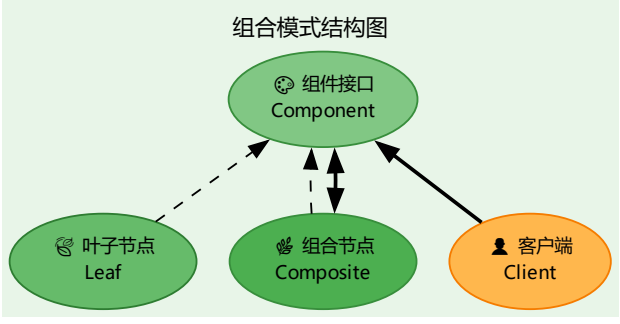
适用场景

- 需要表示对象的部分-整体层次结构，如树形菜单、文件系统等。
- 希望用户可以统一地处理单个对象和组合对象，无需区分它们。

组合模式结构

结构

- **组件接口（Component）**：定义组合对象和叶子对象的公共接口。
- **叶子节点（Leaf）**：表示组合中的叶节点，不包含子组件。
- **组合节点（Composite）**：表示组合中的非叶节点，包含子组件。



组合模式举例说明

文件系统——树形结构 假设我们要实现一个文件系统，文件系统中有文件和文件夹两种类型。文件夹可以包含子文件夹和文件，形成树形的层级结构。我们可以使用组合模式来表示这种文件系统结构。**组合模式**的实现思路如下：

- 定义 `Component` 组件接口，包含如 `add(Component)`、`remove(Component)` 等方法。
- 定义 `Leaf` 叶子节点类，实现 `Component` 接口，表示文件。
- 定义 `Composite` 组合节点类，实现 `Component` 接口，表示文件夹。

组合模式示例（组件）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <windows.h>
using namespace std;
// 组件基类
class Component {
public:
    virtual ~Component() = default;
    virtual void display(int depth = 0) const = 0;
    virtual void add(shared_ptr<Component> c) {}
};
// 文件（叶子节点）
class File : public Component {
    string name;
    int size;
public:
    File(const string& n, int s) : name(n), size(s)
    ↪ {}
    void display(int depth = 0) const override {
        cout << string(depth * 2, ' ') << "📄 " <<
        ↪ name << " (" << size << " bytes)" <<
        ↪ endl;
    }
};
// 文件夹（组合节点）
class Folder : public Component {
    string name;
    vector<shared_ptr<Component>> children;
public:
    Folder(const string& n) : name(n) {}
    void add(shared_ptr<Component> c) override {
        children.push_back(c);
    }
    void display(int depth = 0) const override {
        cout << string(depth * 2, ' ') << "📁 " <<
        ↪ name << "/" << endl;
        for (const auto& c : children)
        ↪ c->display(depth + 1);
    }
};
```

组合模式示例（组件使用）

```
42      // 构建一个复杂的树形结构
43      auto root = make_shared<Folder>("根目录");
44      auto docs = make_shared<Folder>("文档");
45      auto codes = make_shared<Folder>("代码");
46
47      // 文档文件夹下有两个文件和一个子文件夹
48      docs->add(make_shared<File>("说明.txt", 120));
49      docs->add(make_shared<File>("计划.docx", 300));
50      auto reports = make_shared<Folder>("报告");
51      reports->add(make_shared<File>("2023 年总结.pdf",
        ↪  500));
52      docs->add(reports);
53
54      // 代码文件夹下有一个文件和一个子文件夹
```

```
55      codes->add(make_shared<File>("main.cpp", 80));
56      auto utils = make_shared<Folder>("工具");
57      utils->add(make_shared<File>("util.h", 20));
58      utils->add(make_shared<File>("util.cpp", 60));
59      codes->add(utils);
60
61      // 根目录下添加三个文件夹
62      root->add(docs);
63      root->add(codes);
64
65      // 展示树形结构
66      root->display();
```

组合模式优缺点

优点

- 简化客户端代码，统一处理单个对象和组合对象。
- 简化系统结构，使得系统更易于扩展。
- 提高代码复用性，组合对象可以重用。

缺点

- 可能会导致系统设计变得复杂，难以理解。
- 组合模式可能会导致系统性能下降，因为需要遍历整个组合树。

装饰器模式（Decorator Pattern）

装饰器模式是一种结构型设计模式，允许在不修改原有对象结构的情况下，动态地为对象添加额外的功能。与通过继承生成子类相比，装饰器模式更加灵活，能够实现功能的按需组合和扩展 **核心思想**使用一系列装饰器类将原始对象”包裹”起来，每个装饰器都可以在保持原有接口的前提下，增加新的行为或责任。

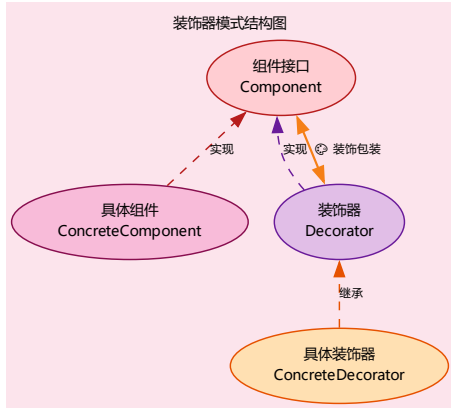
适用场景

- 需要在运行时为对象动态添加功能，且这些功能可以灵活组合。
- 希望通过组合而非继承的方式扩展对象的行为，避免类爆炸。
- 希望对核心类的功能进行增强，但又不希望影响到其他对象。
- 需要实现可撤销的职责或临时性的功能增强。

装饰器模式结构

结构

- **组件接口（Component）**：定义组件的公共接口。
- **具体组件（Concrete Component）**：表示具体的组件类，实现组件接口。
- **装饰器（Decorator）**：表示装饰器类，包含对组件的引用，实现组件接口。
- **具体装饰器（Concrete Decorator）**：表示具体的装饰器类，实现装饰器接口。



装饰器模式举例说明

音乐播放器——动态扩展功能 假设我们有一个音乐播放器，播放器本身可以播放音频文件。现在我们希望在不修改播放器核心代码的情况下，动态地为播放器增加新的功能，比如”均衡器”、”音量增强”、”歌词显示”等。此时可以使用装饰器模式来实现功能的灵活扩展。**装饰器模式**的实现思路如下：

- 定义 `Player` 组件接口，包含如 `play()` 等方法。
- 定义 `ConcretePlayer` 具体组件类，实现 `Player` 接口，负责基础的播放功能。
- 定义 `PlayerDecorator` 装饰器类，持有 `Player` 接口的引用，并实现 `Player` 接口。
- 定义 `ConcreteDecorator` 具体装饰器类，实现 `PlayerDecorator`，如”均衡器”、”音量增强”等，扩展播放器功能。

装饰器模式示例（客户端 + 装饰器）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;
// 播放器组件接口
class Player {
public:
    virtual ~Player() = default;
    virtual void play() = 0;
};
// 具体播放器
class ConcretePlayer : public Player {
public:
    void play() override {
        cout << "播放音频文件" << endl;
    }
};
```

```cpp
// 装饰器基类
class PlayerDecorator : public Player {
protected:
    unique_ptr<Player> player;
public:
    PlayerDecorator(unique_ptr<Player> p) :
    ↪  player(move(p)) {}
    virtual ~PlayerDecorator() = default;
};
// 具体装饰器：均衡器
class EqualizerDecorator : public PlayerDecorator {
public:
    EqualizerDecorator(unique_ptr<Player> p) :
    ↪  PlayerDecorator(move(p)) {}
    void play() override {
        player->play();
        cout << "均衡器功能：调整音效" << endl;
    }
};
```

装饰器模式示例（装饰器使用）

```
38  // 具体装饰器：音量增强
39  class VolumeBoostDecorator : public PlayerDecorator {
40  public:
41      VolumeBoostDecorator(unique_ptr<Player> p) :
    ↪   PlayerDecorator(move(p)) {}
42      void play() override {
43          player->play();
44          cout << "音量增强功能：提升音量" << endl;
45      }
46  };
47  // 具体装饰器：歌词显示
48  class LyricsDecorator : public PlayerDecorator {
49  public:
50      LyricsDecorator(unique_ptr<Player> p) :
    ↪   PlayerDecorator(move(p)) {}
51      void play() override {
52          player->play();
53          cout << "歌词显示功能：同步显示歌词" << endl;
54      }
55  };
```

```
67      player =
    ↪   make_unique<EqualizerDecorator>(move(player));
68      cout << "\n添加均衡器功能后：" << endl;
69      player->play();
70      // 增加音量增强功能
71      player =
    ↪   make_unique<VolumeBoostDecorator>(move(player));
72      cout << "\n再添加音量增强功能后：" << endl;
73      player->play();
74      // 增加歌词显示功能
75      player =
    ↪   make_unique<LyricsDecorator>(move(player));
76      cout << "\n再添加歌词显示功能后：" << endl;
77      player->play();
```

装饰器模式优缺点

优点

- 灵活性高，可以在运行时动态地为对象添加功能。
- 可以组合多个装饰器，实现复杂的功能。
- 可以实现可撤销的职责。

缺点

- 可能会导致系统设计变得复杂，难以理解。
- 装饰器模式可能会导致系统性能下降，因为需要遍历整个装饰器链。

外观模式（Facade Pattern）

外观模式是一种结构型设计模式，通过为子系统中的一组接口提供一个统一的高层接口，使得子系统更加易用。它将复杂子系统的内部实现细节对外部屏蔽，客户端只需与外观对象交互即可完成复杂操作，从而降低了系统的耦合度。

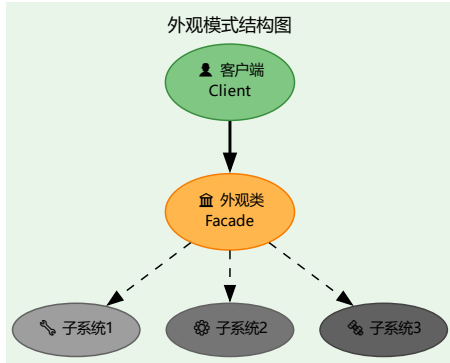
适用场景

- 希望为复杂子系统提供一个简单统一的入口。
- 客户端与多个子系统存在较强依赖，希望通过外观模式降低耦合。
- 需要分层设计时，可以使用外观模式作为每一层的入口，简化层与层之间的交互。

外观模式结构

结构

- **外观（Facade）**：提供简单接口，封装子系统复杂操作。
- **子系统（Subsystem）**：实现具体业务逻辑，被外观类调用。



外观模式举例说明

智能家居系统——回家模式 假设我们有一个智能家居系统，系统本身包含多个子系统：智能灯光、智能空调、安全监控等。现在我们希望用户只需按一个按钮即可完成”回家模式”这一系列操作，而无需分别操作每个子系统。此时可以使用外观模式来实现功能的简化和统一。**外观模式**的实现思路如下：

- 定义各个子系统的接口和实现类，如 `LightSystem`、`AirConditioner`、`SecuritySystem` 等，分别负责各自的功能。
- 定义 `SmartHomeFacade` 外观类，内部组合各个子系统对象，封装复杂的操作流程。
- 用户只需调用 `SmartHomeFacade` 的 `enterHomeMode()` 等简单方法，即可一键完成回家模式的所有操作，无需关心子系统的具体细节。

外观模式示例（子系统）

```cpp
#include <iostream>
#include <string>
#include <memory>
#include <windows.h>
using namespace std;

// 子系统：智能灯光
class LightSystem {
public:
    void on() { cout << "智能灯光已开启" << endl; }
    void off() { cout << "智能灯光已关闭" << endl; }
    void setBrightness(int level) {
        cout << "智能灯光亮度设置为：" << level << "%"
        << endl;
    }
};
```

```cpp
// 子系统：智能空调
class AirConditioner {
public:
    void on() { cout << "智能空调已开启" << endl; }
    void off() { cout << "智能空调已关闭" << endl; }
    void setTemperature(int temp) {
        cout << "智能空调温度设置为：" << temp << "℃"
        << endl;
    }
};

// 子系统：安全监控
class SecuritySystem {
public:
    void arm() { cout << "安全监控系统已布防" << endl;
    } 
    void disarm() { cout << "安全监控系统已撤防" <<
    endl; }
};
```

外观模式示例（外观类）

```
// 外观类：智能家居外观
class SmartHomeFacade {
private:
    unique_ptr<LightSystem> light;
    unique_ptr<AirConditioner> air;
    unique_ptr<SecuritySystem> security;
public:
    SmartHomeFacade() {
        light = make_unique<LightSystem>();
        air = make_unique<AirConditioner>();
        security = make_unique<SecuritySystem>();
    }

    // 回家模式
    void enterHomeMode() {
        cout << "🏠 [外观] 启动回家模式..." << endl;
        security->disarm();
        light->on();
        light->setBrightness(80);
        air->on();
        air->setTemperature(24);
        cout << "🏠 回家模式已开启，欢迎回家！" << endl;
    }

    // 离家模式
    void leaveHomeMode() {
        cout << "🏠 [外观] 启动离家模式..." << endl;
        light->off();
        air->off();
        security->arm();
        cout << "🏠 离家模式已开启，家中安全！" << endl;
    }
};
```

外观模式示例（客户端）

```
72      cout << "=== 外观模式示例：智能家居系统 ===" << endl;
73
74      SmartHomeFacade facade;
75
76      cout << "\n【场景 1】回家：" << endl;
77      facade.enterHomeMode();
78
79      cout << "\n【场景 2】离家：" << endl;
80      facade.leaveHomeMode();
81
82      cout << "\n演示结束，外观模式让复杂的子系统操作变得简单！" << endl;
```

外观模式优缺点

优点

- 简化客户端使用，降低耦合度。
- 客户端只需与外观类交互。
- 增强系统可扩展性，易于维护和修改。

缺点

- 可能引入额外复杂度，如果外观类过于复杂。
- 对原有子系统改动较大。

享元模式 (Flyweight Pattern)

享元模式的核心思想是：**通过共享技术，减少内存中对象的数量**，从而有效支持大量细粒度对象的高效复用。如果一个程序中有很多内容相似、重复的对象，我们可以把它们的”相同部分”提取出来共享，只为”不同部分”单独存储，这样可以大大节省内存。

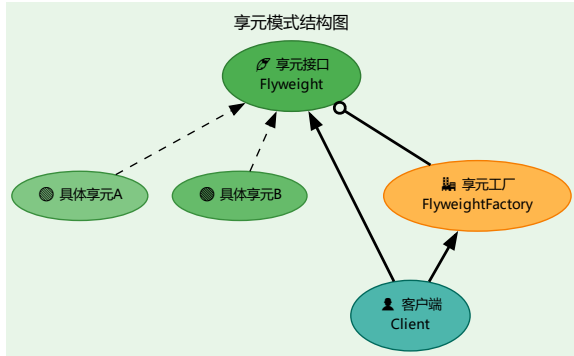
适用场景

- 程序中需要大量相似对象，导致内存占用大。
- 对象的大部分状态可以提取出来作为共享（内部状态），只有少部分信息需要单独保存（外部状态）。
- 可以通过共享对象来替代大量重复对象。
- 对象的身份（唯一性）不重要。

享元模式结构

结构

- **享元接口（Flyweight）**：共享对象，包含内部状态。
- **具体享元（Concrete Flyweight）**：实现享元接口，存储内部状态。
- **享元工厂（Flyweight Factory）**：管理享元对象，提供获取享元对象的方法。



享元模式举例说明

地图应用——树木图标共享 假设我们有一个地图应用，需要在地图上显示成千上万个树木图标。每棵树的种类有限，但位置和高度等信息各不相同。这时可以用享元模式来共享树的图标资源，只为每棵树单独保存位置等外部状态，从而大幅节省内存。**享元模式**的实现思路如下：

- 定义享元接口（Tree），包含树的图标等内部状态。
- 定义具体享元（ConcreteTree），实现享元接口，存储具体树种的图标等信息。
- 定义享元工厂（TreeFactory），管理和复用树种对象，提供获取享元对象的方法。
- 每棵树只保存位置、高度等外部状态，内部状态通过享元对象共享。

享元模式示例（享元接口与具体享元）

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <memory>
#include <windows.h>
using namespace std;

// 享元接口：树
class Tree {
public:
    virtual ~Tree() = default;
    virtual void display(int x, int y, int height)
    ↪  const = 0;
    virtual string getType() const = 0;
};
```

```cpp
// 具体享元：具体树种
class ConcreteTree : public Tree {
    string type;      // 树种类
    string texture;   // 树的贴图（内部状态，假设为字符
    ↪  串）
public:
    ConcreteTree(const string& t, const string& tex)
    ↪  : type(t), texture(tex) {}
    void display(int x, int y, int height) const
    ↪  override {
        cout << "在 (" << x << "," << y << ") 处显示
        ↪  [" << type << "] 树，高度" << height
                << "，贴图资源：" << texture << endl;
    }
    string getType() const override { return type; }
};
```

享元模式示例（享元工厂）

```
// 享元工厂：管理和复用树种对象
class TreeFactory {
    unordered_map<string, shared_ptr<Tree>> treeMap;
public:
    shared_ptr<Tree> getTree(const string& type) {
        auto it = treeMap.find(type);
        if (it != treeMap.end()) {
            //cout << " 重用已有树种：" << type << endl;
            return it->second;
        }
        // 假设贴图资源名与树种类一致
        auto tree = make_shared<ConcreteTree>(type, type + "_texture.png");
        treeMap[type] = tree;
        //cout << " 创建新树种：" << type << endl;
        return tree;
    }
    size_t getTreeTypeCount() const { return treeMap.size(); }
    void listTreeTypes() const {
        cout << "享元池中的树种：";
        for (const auto& p : treeMap) cout << p.first << " ";
        cout << endl;
    }
};

// 每棵树的外部状态
struct TreeInfo {
    int x, y;
    int height;
    string type;
    TreeInfo(int px, int py, int h, const string& t) : x(px), y(py), height(h), type(t) {}
};
```

享元模式示例（客户端）

```
// 地图应用：管理所有树的外部状态
class Forest {
    vector<TreeInfo> trees;
    unique_ptr<TreeFactory> factory;
public:
    Forest() { factory = make_unique<TreeFactory>(); }
    void plantTree(int x, int y, int height, const string& type) {
        trees.emplace_back(x, y, height, type);
    }
    void display() {
        cout << "\n🌲 地图上树木分布如下：" << endl;
        for (const auto& t : trees) {
            auto tree = factory->getTree(t.type);
            tree->display(t.x, t.y, t.height);
        }
    }
    void showStatistics() {
        cout << "\n🌲 享元模式统计：" << endl;
        cout << "总树木数：" << trees.size() << endl;
        cout << "树种享元对象数：" << factory->getTreeTypeCount() << endl;
        cout << "内存节省：" << trees.size() - factory->getTreeTypeCount() << " 个对象" << endl;
        cout << "节省比例：" << (1.0 - static_cast<double>(factory->getTreeTypeCount()) / trees.size()) * 100 << "%" << endl;
        factory->listTreeTypes();
    }
};
```

享元模式示例（应用）

```
94      Forest forest;
95      // 假设地图上有成千上万棵树，这里只演示部分
96      cout << "\n种植树木..." << endl;
97      for (int i = 0; i < 1000; i++) {
98          forest.plantTree(rand() % 100, rand() % 100, rand() % 100, "松树");
99      }
100     for (int i = 0; i < 1000; i++) {
101         forest.plantTree(rand() % 100, rand() % 100, rand() % 100, "杨树");
102     }
103     for (int i = 0; i < 1000; i++) {
104         forest.plantTree(rand() % 100, rand() % 100, rand() % 100, "枫树");
105     }
106     for (int i = 0; i < 1000; i++) {
107         forest.plantTree(rand() % 100, rand() % 100, rand() % 100, "桃树");
108     }
109     // 显示所有树
110     forest.display();
111     // 显示享元统计信息
112     forest.showStatistics();
```

享元模式优缺点

优点

- 大量对象共享内部状态，极大节省内存。
- 外部状态独立存储，灵活管理。
- 适合地图、图标、粒子等场景。

缺点

- 需要将内部状态和外部状态分离，这可能会增加系统复杂度。
- 外部状态的管理可能比较繁琐。

代理模式（Proxy Pattern）

为其他对象提供一种代理，以控制对该对象的访问。代理对象在客户端和目标对象之间起到中介作用，可以在不改变目标对象的前提下，扩展或控制其访问方式。

常见类型

- **远程代理**：为位于不同地址空间的对象提供本地代表，实现远程通信。
- **虚拟代理**：根据需要创建开销较大的对象，实现延迟加载或按需创建。
- **保护代理**：控制对目标对象的访问权限，进行安全控制。
- **智能引用**：在访问对象时附加额外操作，如引用计数、日志记录等。

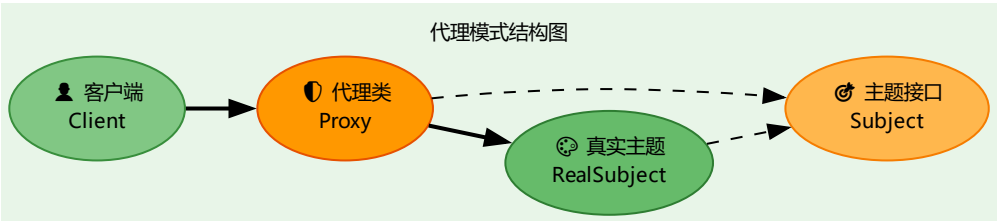
适用场景

- 需要对目标对象的访问进行控制或增强时。
- 希望在不修改目标对象代码的情况下，增加额外功能（如缓存、安全、延迟加载等）。
- 需要为资源消耗大或敏感的对象提供统一的访问接口。

代理模式结构

结构组成

- **Subject（抽象主题/接口）**：定义代理和真实对象的公共接口，确保两者可以互换。
- **RealSubject（真实主题）**：实现了 Subject 接口，提供实际的核心业务功能。
- **Proxy（代理类）**：同样实现 Subject 接口，内部持有 RealSubject 对象的引用，负责在客户端和真实对象之间进行控制和扩展（如权限校验、延迟加载、日志等）。



代理模式举例说明

虚拟代理：游戏资源的延迟加载 在大型游戏开发中，场景中通常包含大量高精度 3D 模型和贴图资源。这些资源文件体积庞大，加载耗时且占用大量内存。如果在游戏启动时一次性全部加载，容易导致启动缓慢、内存压力大。理想的做法是：**仅在玩家真正进入某个场景或靠近某个对象时，才加载对应资源。**此时，**虚拟代理**模式可以很好地解决这一问题。**实现思路如下：**

- 定义资源接口（`Subject`），如 `load()`、`render()` 等，统一资源的加载与显示操作。
- 实现真实资源类（`RealSubject`），负责真正从磁盘加载和渲染资源，构造时会读取大文件并初始化。
- 实现资源代理类（`Proxy`），同样实现资源接口，内部持有真实资源对象指针。只有在调用 `load()` 或 `render()` 时，才会真正创建和加载资源，实现延迟加载。

代理模式示例（主题类）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <windows.h>

using namespace std;

// 抽象主题（接口）
class GameResource {
public:
    virtual ~GameResource() = default;
    virtual void load() = 0;
    virtual void render() = 0;
};
```

```cpp
// 真实主题
class RealGameResource : public GameResource {
    string name;
    bool loaded = false;
public:
    RealGameResource(const string& n) : name(n) {}
    void load() override {
        if (!loaded) {
            cout << "[真实资源] 加载: " << name <<
            ↪  endl;
            loaded = true;
        }
    }
    void render() override {
        if (!loaded) {
            cout << "[真实资源] 未加载，无法渲染: " <<
            ↪  name << endl;
            return;
        }
        cout << "[真实资源] 渲染: " << name << endl;
    }
};
```

代理模式示例（代理类）

```
// 代理
class ResourceProxy : public GameResource {
    string name;
    unique_ptr<RealGameResource> realResource;
public:
    ResourceProxy(const string& n) : name(n) {
        cout << "[代理] 创建资源代理: " << name << endl;
    }
    void load() override {
        if (!realResource) {
            cout << "[代理] 延迟加载资源: " << name << endl;
            realResource = make_unique<RealGameResource>(name);
            realResource->load();
        }
    }
    void render() override {
        if (!realResource) load();
        realResource->render();
    }
};
```

```
    ResourceProxy tree("大树模型");
    ResourceProxy house("别墅模型");

    cout << "\n--- 渲染大树 ---" << endl;
    tree.render();

    cout << "\n--- 再次渲染大树 ---" << endl;
    tree.render();

    cout << "\n--- 渲染别墅 ---" << endl;
    house.render();

    cout << "\n--- 预加载别墅资源 ---" << endl;
    house.load();

    cout << "\n--- 再次渲染别墅 ---" << endl;
    house.render();

    cout << "\n演示完成！" << endl;
```

代理模式优缺点

优点

- 控制对真实对象的访问，保护真实对象。
- 延迟对象的创建，提高性能。
- 统一接口，方便扩展。

缺点

- 增加了系统的复杂度。
- 代理对象和真实对象可能不一致，需要额外处理。

目录

什么是行为型模式

行为型模式关注的是”对象之间如何协作与通信”，即如何合理分配职责、组织对象之间的交互，从而实现更灵活、可扩展的系统行为。行为型模式不仅仅关注类和对象的结构，还强调它们之间的动态交互和职责划分。

这些模式可以帮助我们降低对象之间的耦合度，优化职责分配，提高系统的灵活性和可维护性。

常见的行为型模式有：责任链模式（请求传递）、命令模式（请求封装）、解释器模式（语言解析）、迭代器模式（顺序访问）、中介者模式（对象交互）、备忘录模式（状态保存）、观察者模式（事件通知）、状态模式（状态变迁）、策略模式（算法选择）、模板方法模式(算法框架)、访问者模式（双重分派）等。

每种模式都有其独特的应用场景，比如请求链式处理、解耦发送者与接收者、动态切换算法、对象状态管理、事件通知等，都可以用行为型模式来解决。

行为型模式概述（上）

行为型模式（Behavioral Patterns）关注对象之间的通信和职责分配，共有 **11 种经典模式**：

- **责任链模式**：将请求沿着处理链传递，避免请求发送者和接收者之间的耦合
- **命令模式**：将请求封装为对象，支持请求的排队、撤销和重做操作
- **解释器模式**：定义语言的文法表示，并定义一个解释器来解释语言中的句子
- **迭代器模式**：提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示
- **中介者模式**：用一个中介对象来封装一系列的对象交互，使对象不需要显式地相互引用
- **备忘录模式**：在不破坏封装的前提下，捕获并保存一个对象的内部状态，以便稍后恢复

行为型模式概述（下）

- **观察者模式**：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知
- **状态模式**：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
- **策略模式**：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换
- **模板方法模式**：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中实现
- **访问者模式**：表示一个作用于某对象结构中的各元素的操作，使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作

责任链模式 (Chain of Responsibility Pattern)

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

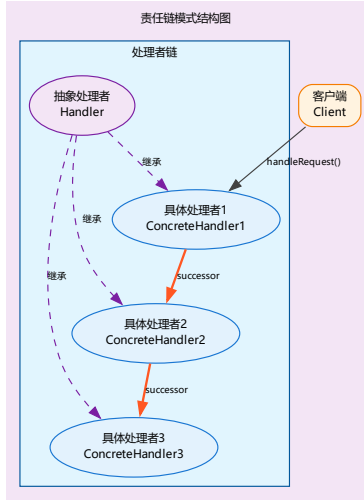
适用场景

- 有多个对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

责任链模式结构

结构组成

- **Handler（抽象处理者）**：定义一个处理请求的接口，并实现后继链(successor)。
- **ConcreteHandler（具体处理者）**：实现抽象处理者的处理方法，如果可以处理请求，则处理请求，否则将请求传递给后继者。
- **Client（客户端）**：创建处理链，并向链的第一个处理者发送请求，开始处理过程。



责任链模式举例说明

技能效果责任链：多环节处理技能请求 在角色扮演类游戏中，角色释放技能时，技能的最终效果往往需要经过多个环节的处理。例如，技能伤害会先经过装备加成、再叠加 Buff 增益，最后还可能被护盾吸收。每个环节都可以对技能效果进行修改或拦截。理想的做法是：**将这些处理环节串联成一条责任链，技能请求依次传递，直到最终生效或被拦截。实现思路如下：**

- 定义技能处理接口（`Handler`），如 `handleSkill()`，并实现后继链（successor），统一技能请求的处理流程。
- 实现具体处理者类（`ConcreteHandler`），如装备处理器、Buff 处理器、护盾处理器等，分别负责各自环节的处理逻辑。每个处理者判断是否处理当前请求，若不能处理则传递给下一个处理者。
- 客户端（`Client`）负责创建技能处理链，并将技能请求交给链的起点，技能效果依次经过各个处理环节，最终得到处理结果。

责任链模式示例（抽象处理者）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <windows.h>
using namespace std;

// 技能请求类
class SkillRequest {
private:
    string description;
    int power; // 技能威力
public:
    SkillRequest(const string& desc, int p) :
    ↪ description(desc), power(p) {}
    const string& getDescription() const { return
    ↪ description; }
    int getPower() const { return power; }
    void setPower(int p) { power = p; }
};
```

```cpp
// 抽象处理者
class SkillHandler {
protected:
    shared_ptr<SkillHandler> next;
    string name;

public:
    SkillHandler(const string& n) : name(n),
    ↪ next(nullptr) {}

    void setNext(shared_ptr<SkillHandler> handler) {
        next = handler;
    }

    virtual void handle(shared_ptr<SkillRequest>
    ↪ request) = 0;

    virtual ~SkillHandler() = default;
};
```

责任链模式示例（具体处理者）

```cpp
// 具体处理者 - 装备加成
class EquipmentHandler : public SkillHandler {
public:
    EquipmentHandler(const string& n) :
    ↪ SkillHandler(n) {}

    void handle(shared_ptr<SkillRequest> request)
    ↪ override {
        cout << name << " 装备加成：技能威力提升 20%"
        ↪ << endl;
        int newPower =
        ↪ static_cast<int>(request->getPower() *
        ↪ 1.2);
        request->setPower(newPower);
        if (next) next->handle(request);
    }
};
```

```cpp
// 具体处理者 - Buff 增益
class BuffHandler : public SkillHandler {
public:
    BuffHandler(const string& n) : SkillHandler(n) {}

    void handle(shared_ptr<SkillRequest> request)
    ↪ override {
        cout << name << " Buff 增益：技能威力提升 30%"
        ↪ << endl;
        int newPower =
        ↪ static_cast<int>(request->getPower() *
        ↪ 1.3);
        request->setPower(newPower);
        if (next) next->handle(request);
    }
};
```

责任链模式示例（具体处理者）

```cpp
// 具体处理者 - 护盾吸收
class ShieldHandler : public SkillHandler {
public:
    ShieldHandler(const string& n) : SkillHandler(n) {}

    void handle(shared_ptr<SkillRequest> request) override {
        cout << name << " 护盾吸收：技能威力减少 50 点" << endl;
        int newPower = request->getPower() - 50;
        if (newPower < 0) newPower = 0;
        request->setPower(newPower);
        if (next) next->handle(request);
    }
};
```

```cpp
// 具体处理者 - 最终结算
class FinalHandler : public SkillHandler {
public:
    FinalHandler(const string& n) : SkillHandler(n) {}

    void handle(shared_ptr<SkillRequest> request) override {
        cout << name << " 最终结算：技能 \"" << request->getDescription()
             << "\" 最终威力为 " << request->getPower() << endl;
    }
};
```

责任链模式示例（客户端）

```
96      // 创建处理者
97      auto equipment = make_shared<EquipmentHandler>("装备系统");
98      auto buff = make_shared<BuffHandler>("Buff 系统");
99      auto shield = make_shared<ShieldHandler>("护盾系统");
100     auto final = make_shared<FinalHandler>("结算系统");
101
102     // 构建责任链
103     equipment->setNext(buff);
104     buff->setNext(shield);
105     shield->setNext(final);
106
107     // 创建技能请求
108     auto skill1 = make_shared<SkillRequest>("火球术", 100);
109     auto skill2 = make_shared<SkillRequest>("雷霆一击", 200);
110
111     cout << "\n1. 释放技能：火球术" << endl;
112     equipment->handle(skill1);
113
114     cout << "\n2. 释放技能：雷霆一击" << endl;
115     equipment->handle(skill2);
```

责任链模式优缺点

优点

- 降低对象之间的耦合度，使请求的发送者与接收者解耦，提高系统灵活性和可维护性。
- 支持动态地添加、删除或重组处理者，便于扩展和修改。
- 责任链可以灵活组合，支持多种处理顺序和策略，增强系统的可配置性。
- 每个处理者只需关注自身职责，便于代码复用。

缺点

- 可能导致请求在链上传递过长，影响性能，调试和排查较为困难。
- 责任链过长或不合理，可能导致某些请求得不到及时处理或丢失。
- 处理者之间的顺序依赖需要谨慎管理，否则可能出现逻辑错误。

解释器模式（Interpreter Pattern）

给定一种语言，定义其文法的一种表示，并定义一个解释器，利用该表示来解释语言中的句子。解释器模式常用于实现自定义脚本、表达式求值、规则引擎等场景。

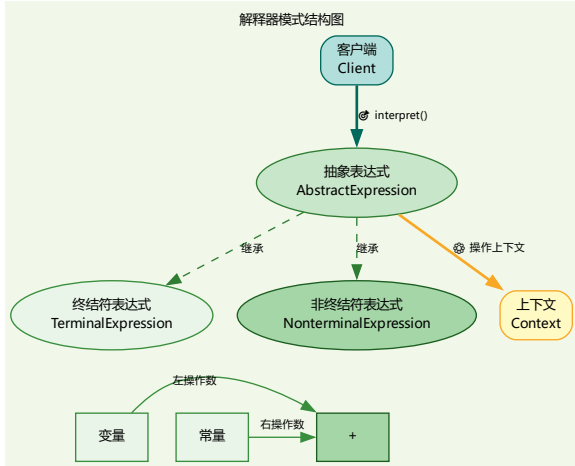
适用场景

- 需要解释执行的语言，并且可以将语言中的句子表示为抽象语法树（AST）。
- 语言的文法结构相对简单，变化不频繁。
- 对执行效率要求不高，主要关注灵活性和可扩展性。

解释器模式结构

结构组成

- **AbstractExpression（抽象表达式）**：定义解释器的接口，用于解释语法规则。
- **TerminalExpression（终结符表达式）**：实现抽象表达式的解释方法，处理最基本的语法单元。
- **NonTerminalExpression（非终结符表达式）**：实现抽象表达式的解释方法，处理复合语法单元。
- **Context（上下文）**：包含解释器需要的数据，传递给抽象表达式。



解释器模式举例说明

游戏中的解释器模式——属性公式解析 在一款角色扮演游戏中，角色的攻击力、伤害、暴击等属性往往需要根据策划配置的公式动态计算。例如” 攻击力 = 力量 * 2 + 武器加成”。为了让策划可以灵活调整公式，系统需要支持表达式的解析与计算。此时可以采用解释器模式，将公式解析和计算过程解耦，结构如下：

- 定义抽象表达式（AbstractExpression），声明解释接口（如 interpret(context)）。
- 实现终结符表达式（TerminalExpression），用于处理常量、变量（如” 力量”、” 武器加成”），直接返回对应值。
- 实现非终结符表达式（NonTerminalExpression），如加法、乘法等运算符，递归组合并解释子表达式。
- 定义上下文（Context），用于存储变量的取值环境（如角色当前属性表）。

通过解释器模式，游戏可以灵活支持各种属性公式的解析与计算，便于策划随时调整规则，无需修改底层代码。

解释器模式示例（抽象表达式）

```cpp
#include <iostream>
#include <map>
#include <memory>
#include <windows.h>

using namespace std;

// 抽象表达式
class Expression {
public:
    virtual int interpret(const map<string, int>&
    ↪  context) = 0;
    virtual ~Expression() = default;
};

// 终结符表达式 - 常量
class NumberExpression : public Expression {
    int value;
public:
    NumberExpression(int v) : value(v) {}
    int interpret(const map<string, int>& context)
    ↪  override {
        return value;
    }
};

// 终结符表达式 - 变量
class VariableExpression : public Expression {
    string name;
public:
    VariableExpression(const string& n) : name(n) {}
    int interpret(const map<string, int>& context)
    ↪  override {
        auto it = context.find(name);
        if (it != context.end()) {
            return it->second;
        }
        return 0;
    }
};
```

解释器模式示例（表达式）

```
39  // 非终结符表达式 - 加法
40  class AddExpression : public Expression {
41      shared_ptr<Expression> left;
42      shared_ptr<Expression> right;
43  public:
44      AddExpression(shared_ptr<Expression> l,
        ↪  shared_ptr<Expression> r)
45          : left(l), right(r) {}
46      int interpret(const map<string, int>& context)
        ↪  override {
47          return left->interpret(context) +
            ↪  right->interpret(context);
48      }
49  };
50
51  // 非终结符表达式 - 乘法
52  class MultiplyExpression : public Expression {
53      shared_ptr<Expression> left;
54      shared_ptr<Expression> right;
55  public:
56      MultiplyExpression(shared_ptr<Expression> l,
        ↪  shared_ptr<Expression> r)
57          : left(l), right(r) {}
58      int interpret(const map<string, int>& context)
        ↪  override {
59          return left->interpret(context) *
            ↪  right->interpret(context);
60      }
61  };
62
63  // 非终结符表达式 - 减法
64  class SubtractExpression : public Expression {
65      shared_ptr<Expression> left;
66      shared_ptr<Expression> right;
67  public:
68      SubtractExpression(shared_ptr<Expression> l,
        ↪  shared_ptr<Expression> r)
69          : left(l), right(r) {}
70      int interpret(const map<string, int>& context)
        ↪  override {
71          return left->interpret(context) -
            ↪  right->interpret(context);
72      }
73  };
```

解释器模式示例（客户端）

```
    cout << "=== 解释器模式示例：游戏属性公式解析 ===" << endl;

    // 创建角色属性上下文
    map<string, int> context;
    context["力量"] = 20;
    context["武器加成"] = 15;

    cout << "\n角色属性上下文:" << endl;
    cout << "力量 = " << context["力量"] << endl;
    cout << "武器加成 = " << context["武器加成"] << endl;

    // 构建表达式：攻击力 = 力量 * 2 + 武器加成
    // 即 AddExpression(MultiplyExpression(VariableExpression("力量"), NumberExpression(2)), VariableExpression("武器加成"))
```

```
    auto expr = make_shared<AddExpression>(
        make_shared<MultiplyExpression>(
            make_shared<VariableExpression>("力量"),
            make_shared<NumberExpression>(2)
        ),
        make_shared<VariableExpression>("武器加成")
    );

    cout << "\n表达式计算:" << endl;
    cout << "攻击力 = 力量 * 2 + 武器加成 = " << expr->interpret(context) << endl;
    cout << "计算过程:" << endl;
    int li = context["力量"];
    int wuqi = context["武器加成"];
    cout << "  力量 * 2 = " << li << " * 2 = " << li * 2 << endl;
    cout << "  (力量 * 2) + 武器加成 = " << (li * 2) << " + " << wuqi << " = " << expr->interpret(context) << endl;
```

解释器模式优缺点

优点

- 易于扩展新的运算符和语法规则，支持灵活定制和组合表达式，适合实现可配置的公式或脚本解析。
- 结构清晰，符合面向对象设计原则，每种语法规则对应独立类，便于管理和复用。

缺点

- 当语法规则较多时，类数量急剧增加，导致系统结构臃肿，维护成本上升。
- 解释执行效率较低，不适合对性能要求较高或高频计算的场景。
- 复杂表达式的解析和递归解释可能导致栈溢出或资源消耗过大。

迭代器模式 (Iterator Pattern)

提供一种方法顺序访问一个聚合对象中的各个元素，而无需暴露其内部实现细节。通过迭代器，客户端可以以一致的方式遍历不同类型的集合对象。

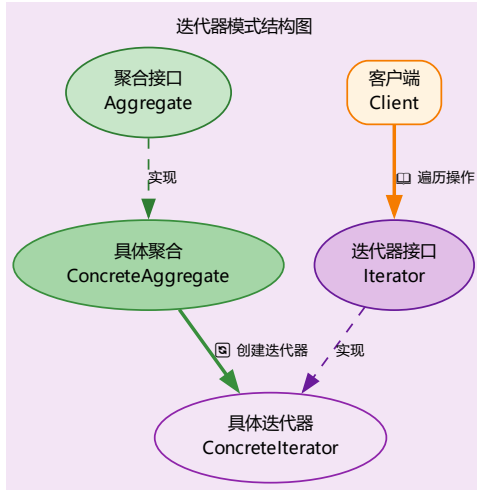
适用场景

- 需要访问一个聚合对象的内容，但不希望暴露其内部结构。
- 需要为不同的聚合结构（如数组、链表、树等）提供统一的遍历方式。
- 希望支持多种遍历方式（如正序、逆序、跳跃等）。
- 多个遍历操作需要在同一个聚合对象上并行进行。

迭代器模式结构

结构组成

- **Iterator（抽象迭代器）**：定义访问和遍历元素的接口。
- **ConcreteIterator（具体迭代器）**：实现抽象迭代器的具体操作，遍历聚合对象。
- **Aggregate（抽象聚合）**：定义创建迭代器的接口，返回一个抽象迭代器。
- **ConcreteAggregate（具体聚合）**：实现抽象聚合的接口，返回具体迭代器。



迭代器模式举例说明

游戏背包物品遍历——统一访问不同容器 在一款角色扮演游戏中，玩家的背包中可能包含多种类型的物品（如武器、药水、材料等），这些物品可能分别存储在不同的数据结构中(如数组、链表、集合等)。如果客户端希望以统一的方式遍历和访问所有物品，就可以引入**迭代器模式**。其实现思路如下：

- 定义**抽象迭代器（Iterator）**，规定遍历接口（如 `hasNext()`、`next()` 等）。
- 定义**具体迭代器（ConcreteIterator）**，实现抽象迭代器接口，负责遍历具体的容器结构。
- 定义**抽象聚合（Aggregate）**，声明创建迭代器的方法。
- 定义**具体聚合（ConcreteAggregate）**，实现抽象聚合接口，返回对应的具体迭代器。

通过迭代器模式，客户端无需关心物品的存储结构，只需通过统一的迭代器接口即可顺序访问所有物品，极大提升了代码的灵活性和可维护性。

迭代器模式示例（抽象迭代器与抽象聚合）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;

// 抽象迭代器
class Iterator {
public:
    virtual bool hasNext() const = 0;
    virtual const string& next() = 0;
    virtual ~Iterator() = default;
};
```

```cpp
// 抽象聚合
class Aggregate {
public:
    virtual shared_ptr<Iterator> createIterator()
    ↪ const = 0;
    virtual shared_ptr<Iterator>
    ↪ createReverseIterator() const = 0;
    virtual void addItem(const string& item) = 0;
    virtual ~Aggregate() = default;
};
```

迭代器模式示例（具体聚合）

```
// 具体聚合
class ConcreteAggregate : public Aggregate {
private:
    vector<string> items;
    // 正向迭代器
    class ConcreteIterator : public Iterator {
        const vector<string>& items;
        size_t index;
    public:
    ConcreteIterator(const vector<string>& items) : items(items), index(0) {}
    bool hasNext() const override { return index < items.size(); }
    const string& next() override {
    if (!hasNext()) throw out_of_range("没有更多元素");
            return items[index++];
        }
    };
    // 反向迭代器
    class ReverseIterator : public Iterator {
        const vector<string>& items;
        int index;
    public:
    ReverseIterator(const vector<string>& items) : items(items), index(static_cast<int>(items.size()) - 1) {}
    bool hasNext() const override { return index >= 0; }
    const string& next() override {
    if (!hasNext()) throw out_of_range("没有更多元素");
            return items[index--];
        }
    };
public:
    void addItem(const string& item) override { items.push_back(item); }
    shared_ptr<Iterator> createIterator() const override {
        return make_shared<ConcreteIterator>(items);
    }
    shared_ptr<Iterator> createReverseIterator() const override {
        return make_shared<ReverseIterator>(items);
    }
};
```

迭代器模式示例（客户端）

```
63  void printItems(const string& title,
    ↪  shared_ptr<Iterator> it) {
64      cout << "\n" << title << endl;
65      while (it->hasNext()) {
66          cout << "  " << it->next() << endl;
67      }
68  }
```

```
70  int main() {
71      SetConsoleOutputCP(65001);
72      SetConsoleCP(65001);
73
74      cout << "=== 迭代器模式示例：遍历游戏物品 ===" <<
        ↪  endl;
75      ConcreteAggregate aggregate;
76      aggregate.addItem("第一个元素");
77      aggregate.addItem("第二个元素");
78      aggregate.addItem("第三个元素");
79      aggregate.addItem("第四个元素");
80      aggregate.addItem("第五个元素");
81      printItems("1. 正向遍历：",
        ↪  aggregate.createIterator());
82      printItems("2. 反向遍历：",
        ↪  aggregate.createReverseIterator());
83
84      cin.get();
85      return 0;
86  }
```

迭代器模式优缺点

优点

- 提供统一的遍历接口，简化客户端使用，提高代码可读性和可维护性。
- 支持多种遍历方式（如正序、逆序、跳跃等），灵活性较高。
- 可以同时遍历多个聚合对象，便于并行操作。
- 符合开闭原则，易于扩展新的遍历方式。

缺点

- 当聚合对象结构复杂时，迭代器的实现可能变得复杂，性能下降。
- 遍历过程中需要维护状态，可能导致内存消耗增加。
- 不支持并发访问，在多线程环境下需要额外的同步措施。

中介者模式（Mediator Pattern）

通过引入一个中介对象，集中管理和协调一组对象之间的交互。这样，各对象之间不再直接引用彼此，降低了耦合度，使对象间的关系更清晰，交互逻辑更易于维护和扩展。

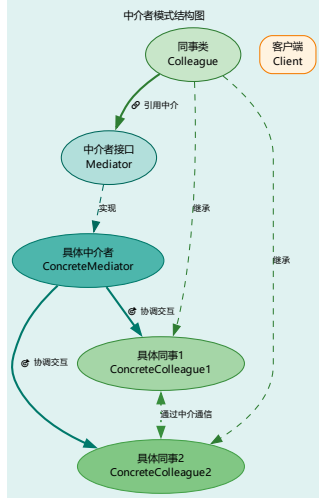
适用场景

- 系统中对象之间存在复杂的引用关系，导致依赖混乱、难以维护。
- 一个对象需要与许多其他对象进行通信，导致其难以复用。
- 希望将对象间的交互行为集中管理，避免在多个类中分散实现，减少子类数量。

中介者模式结构

结构组成

- **Mediator（抽象中介者）**：定义了与各个具体同事（ConcreteColleague）通信的接口。
- **ConcreteMediator（具体中介者）**：实现抽象中介者的方法，负责具体的协调工作，协调各个同事对象。
- **Colleague（抽象同事）**：定义了与中介者通信的接口，同时保持与中介者的引用。
- **ConcreteColleague（具体同事）**：实现抽象同事的接口，负责具体的业务逻辑，通过中介者与其他同事通信。



中介者模式举例说明

多人聊天室：房间中介者集中管理玩家交互 假设在一个多人在线游戏中，玩家可以在同一个房间内聊天、组队、交易等。如果每个玩家对象都直接与其他所有玩家通信，系统结构会非常复杂，难以维护和扩展。
为了解决这个问题，我们可以引入”房间”作为中介者。所有玩家只需与房间（中介者）通信，房间负责协调和转发消息、处理组队和交易请求等。这样，玩家之间无需直接引用彼此，极大降低了系统的耦合度。

结构说明如下：

- **抽象中介者（Mediator）**：定义房间与玩家通信的统一接口。
- **具体中介者（ConcreteMediator）**：如”房间”类，实现中介者接口，负责管理和协调所有玩家的交互。
- **抽象同事（Colleague）**：如”玩家”基类，定义与中介者通信的接口，并持有中介者的引用。
- **具体同事（ConcreteColleague）**：如具体的玩家类，实现抽象同事接口，通过中介者与其他玩家间接通信，处理自身业务逻辑。

总结：通过引入房间中介者，玩家之间的交互被集中管理，系统结构更加清晰，易于维护和扩展。

中介者模式示例（抽象中介者与抽象同事）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <windows.h>

using namespace std;

// 前向声明
class Player;

// 抽象中介者
class Mediator {
public:
    virtual void sendMessage(const string& message, Player* sender) = 0;
    virtual void addPlayer(Player* player) = 0;
    virtual ~Mediator() = default;
};
```

```cpp
// 抽象同事类（玩家基类）
class Player {
protected:
    Mediator* mediator;
    string name;

public:
    Player(const string& n, Mediator* med) : name(n), mediator(med) {
        if (mediator) {
            mediator->addPlayer(this);
        }
    }

    virtual void receiveMessage(const string& message, Player* sender) = 0;
    virtual void sendMessage(const string& message) = 0;

    const string& getName() const { return name; }

    virtual ~Player() = default;
};
```

中介者模式示例（具体中介者与具体同事）

```cpp
// 具体中介者 - 房间
class Room : public Mediator {
private:
    vector<Player*> players;

public:
    void addPlayer(Player* player) override {
        players.push_back(player);
    }

    void sendMessage(const string& message, Player* sender) override {
        cout << "房间广播消息: [" << sender->getName() << "] " << message << endl;
        // 向所有其他玩家发送消息
        for (auto player : players) {
            if (player != sender) {
                player->receiveMessage(message, sender);
            }
        }
    }
};
```

```cpp
// 具体同事类 - 普通玩家
class NormalPlayer : public Player {
public:
    NormalPlayer(const string& name, Mediator* mediator)
        : Player(name, mediator) {}

    void sendMessage(const string& message) override {
        cout << getName() << " 发送消息: " << message << endl;
        mediator->sendMessage(message, this);
    }

    void receiveMessage(const string& message, Player* sender) override {
        cout << getName() << " 收到来自 " << sender->getName()
             << " 的消息: " << message << endl;
    }
};
```

中介者模式示例（客户端）

```
86      // 创建房间（中介者）
87      auto room = make_shared<Room>();
88
89      // 创建玩家
90      auto p1 = make_shared<NormalPlayer>("玩家 A", room.get());
91      auto p2 = make_shared<NormalPlayer>("玩家 B", room.get());
92      auto p3 = make_shared<NormalPlayer>("玩家 C", room.get());
93
94      // 玩家发送消息
95      cout << "\n1. 玩家 A 发送消息: " << endl;
96      p1->sendMessage("大家好, 我是 A! ");
97      cout << endl;
98
99      cout << "2. 玩家 B 发送消息: " << endl;
100     p2->sendMessage("欢迎 A 加入房间! ");
101     cout << endl;
102
103     cout << "3. 玩家 C 发送消息: " << endl;
104     p3->sendMessage("我们一起组队吧! ");
105     cout << endl;
```

中介者模式优缺点

优点

- 集中管理对象间的交互，降低耦合度。
- 减少对象间的直接引用，提高系统的灵活性和可扩展性。
- 简化对象间的交互逻辑，使系统更易于维护。

缺点

- 中介者可能成为系统的中心化单点，增加系统的复杂性。
- 中介者需要处理所有对象的交互逻辑，可能导致中介者类过于庞大。
- 中介者模式可能引入新的依赖关系，使得系统更加复杂。

备忘录模式（Memento Pattern）

在不破坏封装性的前提下，捕获并保存一个对象的内部状态，以便在需要时将其恢复到先前的状态。

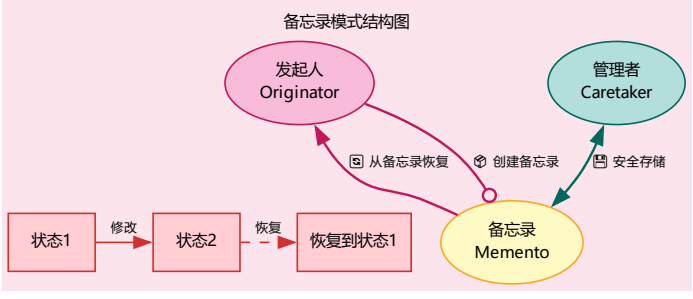
适用场景

- 需要保存和恢复对象历史状态（如撤销/回退操作）。
- 希望避免暴露对象的实现细节，保护其封装性。
- 对象的状态变化较为复杂，且需要在某些时刻进行快照保存。

备忘录模式结构

结构组成

- **Memento（备忘录）**：保存对象的内部状态，只能通过备忘录管理器访问。
- **Originator（发起者）**：创建并使用备忘录，负责创建和恢复备忘录。
- **Caretaker（管理者）**：负责管理备忘录，提供创建和恢复备忘录的接口。



备忘录模式举例说明

游戏进度存档——备忘录模式 假设我们在开发一款大型多人在线游戏，玩家可以随时保存当前的角色状态（如等级、生命、任务进度等），并在需要时恢复到某个历史存档。此时，备忘录模式可以帮助我们优雅地实现”游戏进度快照与回退”功能。其实现思路如下：

- 定义备忘录类（Memento），用于存储玩家的角色属性、任务进度等内部状态，外部无法直接访问其内容。
- 定义发起者类（Originator），即游戏角色本身，能够生成当前状态的备忘录，并可根据备忘录恢复到历史状态。
- 定义管理者类（Caretaker），负责保存和管理备忘录对象，实现多次撤销、重做等功能，但不会修改备忘录内容。

备忘录模式示例（备忘录）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;

// 备忘录（Memento）：负责存储玩家的角色属性、任务进度等内部状态数据
class Memento {
private:
    string state;
public:
    Memento(const string& s) : state(s) {}
    string getState() const { return state; }
};

// 发起者（Originator）：游戏角色本身，能够生成当前状态的备忘录，并可根据备忘录恢复到历史状态
class Originator {
private:
    string state;
public:
    void setState(const string& s) { state = s; }
    string getState() const { return state; }
    shared_ptr<Memento> saveStateToMemento() {
        return make_shared<Memento>(state);
    }
    void getStateFromMemento(shared_ptr<Memento> memento) {
        state = memento->getState();
    }
};
```

备忘录模式示例（发起者、管理者）

```
// 发起者（Originator）：游戏角色本身，能够生成当前状态的备忘录，并可根据备忘录恢复到历史状态
class Originator {
private:
    string state;
public:
    void setState(const string& s) { state = s; }
    string getState() const { return state; }
    shared_ptr<Memento> saveStateToMemento() {
        return make_shared<Memento>(state);
    }
    void getStateFromMemento(shared_ptr<Memento> memento) {
        state = memento->getState();
    }
};
```

```
// 管理者（Caretaker）：负责保存和管理备忘录对象
class Caretaker {
private:
    vector<shared_ptr<Memento>> mementoList;
public:
    void add(shared_ptr<Memento> memento) {
        mementoList.push_back(memento);
    }
    shared_ptr<Memento> get(int index) {
        if (index >= 0 && index < mementoList.size())
            return mementoList[index];
        return nullptr;
    }
    int size() const { return mementoList.size(); }
};
```

备忘录模式示例（客户端）

```
54      Originator player;
55      Caretaker caretaker;
56
57      // 初始状态
58      player.setState("等级 1，生命 100，任务进度 0%");
59      cout << "当前状态: " << player.getState() << endl;
60      caretaker.add(player.saveStateToMemento());
61
62      // 升级
63      player.setState("等级 2，生命 120，任务进度 20%");
64      cout << "当前状态: " << player.getState() << endl;
65      caretaker.add(player.saveStateToMemento());
66
67      // 受伤
68      player.setState("等级 2，生命 60，任务进度 20%");
69      cout << "当前状态: " << player.getState() << endl;
70      caretaker.add(player.saveStateToMemento());
```

```
72      // 完成任务
73      player.setState("等级 3，生命 80，任务进度 100%");
74      cout << "当前状态: " << player.getState() << endl;
75      caretaker.add(player.saveStateToMemento());
76
77      cout << "\n--- 恢复到之前的状态 ---" << endl;
78      // 恢复到受伤前
79      player.getStateFromMemento(caretaker.get(1));
80      cout << "恢复到状态 2: " << player.getState() <<
        ↪  endl;
81
82      // 恢复到初始状态
83      player.getStateFromMemento(caretaker.get(0));
84      cout << "恢复到状态 1: " << player.getState() <<
        ↪  endl;
```

备忘录模式优缺点

优点

- 保护了对象的封装性，避免了直接访问对象内部状态。
- 简化了发起者的实现，发起者只需管理备忘录的创建和恢复。
- 方便了管理者对备忘录的管理和维护。

缺点

- 需要额外的空间来存储备忘录，可能会占用较多的内存。
- 备忘录对象的创建和恢复可能会影响系统的性能和可维护性。

观察者模式（Observer Pattern）

观察者模式定义了一种一对多的依赖关系，使得当一个对象（被观察者/主题）的状态发生变化时，所有依赖于它的对象（观察者）都会自动收到通知并更新。

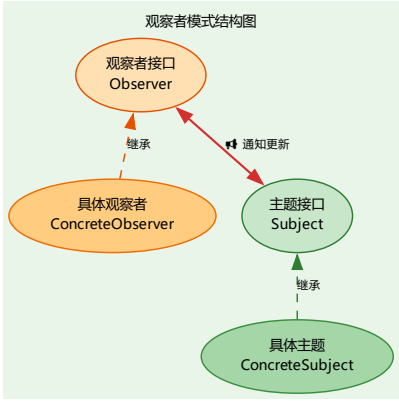
适用场景

- 当一个对象的改变需要同时通知并影响其他对象，且不希望这些对象之间紧密耦合时。
- 当系统中存在一对多的依赖关系，且被依赖方无需关心具体有多少依赖者时。
- 当希望对象能够动态地添加或移除依赖关系时。

观察者模式结构

结构组成

- **Subject（主题/被观察者）**：定义了添加、删除观察者的方法，以及通知观察者的方法。
- **ConcreteSubject（具体主题）**：实现了主题/被观察者的具体业务逻辑，当状态发生改变时，通知所有观察者。
- **Observer（观察者）**：定义了更新自己的方法。
- **ConcreteObserver（具体观察者）**：实现了观察者的具体业务逻辑，当收到通知时，更新自己的状态。



观察者模式举例说明

多人游戏任务进度通知——观察者模式 假设我们正在开发一款大型多人在线游戏，玩家在完成任务后需要实时通知队伍中的其他玩家当前的任务进度。此时，可以使用观察者模式来实现任务进度的自动推送和同步。其实现思路如下：

- 定义主题（Subject），负责维护观察者列表，并提供添加、移除和通知观察者的方法。
- 定义具体主题（TaskProgress），实现任务进度的具体业务逻辑，每当任务进度发生变化时，自动通知所有已注册的玩家。
- 定义观察者接口（Observer），声明接收通知的接口方法。
- 定义具体观察者（Player），实现接收任务进度通知的具体逻辑，玩家收到通知后可及时更新自己的界面或状态。

观察者模式示例（主题/被观察者）

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <algorithm>
#include <windows.h>

using namespace std;

// 观察者接口
class Observer {
public:
    virtual ~Observer() = default;
    virtual void update(const string& taskInfo) = 0;
    virtual string getName() const = 0;
};
```

```cpp
// 主题接口（被观察者）
class Subject {
public:
    virtual ~Subject() = default;
    virtual void addObserver(shared_ptr<Observer>
    ↪  observer) = 0;
    virtual void removeObserver(shared_ptr<Observer>
    ↪  observer) = 0;
    virtual void notifyObservers() = 0;
};
```

观察者模式示例（具体主题）

```cpp
// 具体主题：任务进度
class TaskProgress : public Subject {
private:
    vector<shared_ptr<Observer>> observers;
    string progressInfo;
public:
    void addObserver(shared_ptr<Observer> observer)
    ↪ override {
        observers.push_back(observer);
        cout << "已添加观察者: " <<
        ↪ observer->getName() << endl;
    }

    void removeObserver(shared_ptr<Observer>
    ↪ observer) override {
        auto it = find(observers.begin(),
        ↪ observers.end(), observer);
        if (it != observers.end()) {
            cout << "已移除观察者: " <<
            ↪ observer->getName() << endl;
            observers.erase(it);
        }
    }

    void notifyObservers() override {
        cout << "\n[系统] 通知所有玩家任务进度更新: "
        ↪ << progressInfo << endl;
        for (auto& obs : observers) {
            obs->update(progressInfo);
        }
    }

    void setProgress(const string& info) {
        progressInfo = info;
        notifyObservers();
    }
};
```

观察者模式示例（具体观察者）

```
59  // 具体观察者：玩家
60  class Player : public Observer, public
    ↪  enable_shared_from_this<Player> {
61  private:
62      string name;
63      string lastTaskInfo;
64      TaskProgress* subject;
65  public:
66      Player(const string& n, TaskProgress* s) :
        ↪  name(n), subject(s) {}
67
68      void join() {
69          subject->addObserver(shared_from_this());
70      }
```

```
72      void quit() {
73          subject->removeObserver(shared_from_this());
74      }
75
76      void update(const string& taskInfo) override {
77          lastTaskInfo = taskInfo;
78          cout << "玩家 [" << name << "] 收到任务进度通
            ↪  知: " << lastTaskInfo << endl;
79      }
80
81      string getName() const override {
82          return name;
83      }
84  };
```

观察者模式示例（客户端）

```
90      cout << "=== 观察者模式示例：多人游戏任务进度通知 ===" << endl;
91
92      // 创建任务进度主题
93      auto taskProgress = make_unique<TaskProgress>();
94
95      // 创建玩家（观察者）
96      auto player1 = make_shared<Player>("小明", taskProgress.get());
97      auto player2 = make_shared<Player>("小红", taskProgress.get());
98      auto player3 = make_shared<Player>("小刚", taskProgress.get());
99
100     // 玩家加入观察
101     player1->join();
102     player2->join();
103     player3->join();
```

```
105     // 第一次任务进度更新
106     cout << "\n--- 任务进度更新 1 ---" << endl;
107     taskProgress->setProgress("击败 10 只怪物（3/10）");
108
109     // 第二次任务进度更新
110     cout << "\n--- 任务进度更新 2 ---" << endl;
111     taskProgress->setProgress("击败 10 只怪物（7/10）");
112
113     // 小红退出观察
114     cout << "\n--- 小红退出队伍 ---" << endl;
115     player2->quit();
116
117     // 第三次任务进度更新
118     cout << "\n--- 任务进度更新 3 ---" << endl;
119     taskProgress->setProgress("击败 10 只怪物（10/10），任务完成！");
120
121     cout << "\n演示结束，观察者模式实现了任务进度的实时通知！" << endl;
```

观察者模式优缺点

优点

- 松耦合，观察者和被观察者之间解耦，互不依赖。
- 易于扩展，可以动态添加或移除观察者。

缺点

- 如果观察者数量很多，通知所有观察者可能会耗费大量时间。
- 如果观察者之间存在循环依赖，可能会导致死循环。

策略模式（Strategy Pattern）

定义一系列算法，将每一个算法封装起来，并使它们可以相互替换。策略模式让算法独立于使用它的客户端而变化。

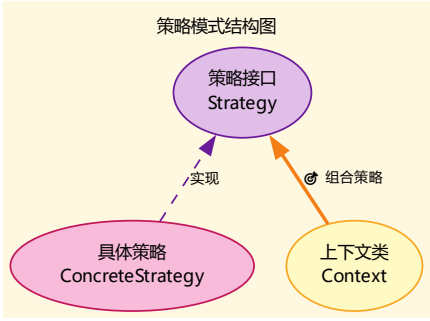
适用场景

- 系统中有许多仅在行为上有所不同的类，需要使用不同的算法或策略。
- 需要从多个算法中选择其一，并且可以灵活切换。
- 算法的实现细节对客户端隐藏，客户端无需了解具体实现。
- 一个类中有多种行为，并且这些行为在类的操作中以条件语句出现时，可用策略模式消除条件分支。

策略模式结构

结构组成

- **Strategy（策略）**：定义了算法的接口。
- **ConcreteStrategy（具体策略）**：实现了策略的接口。



策略模式举例说明

角色攻击方式的灵活切换 假设在一款游戏中，不同角色可以选择不同的攻击方式（如近战攻击、远程攻击、魔法攻击等）。每种攻击方式的实现不同，但角色可以在战斗中灵活切换攻击策略。此时可以使用**策略模式**来实现攻击方式的灵活切换，结构如下：

- 定义策略接口（`AttackStrategy`），声明攻击行为。
- 定义具体策略类（如 `MeleeAttack`、`RangedAttack`、`MagicAttack`），实现不同的攻击方式。
- 角色类中持有策略接口指针，根据需要动态切换不同的攻击策略对象。

这样，角色无需关心具体攻击方式的实现细节，只需选择合适的策略即可完成攻击行为，极大提升了系统的灵活性和可扩展性。

策略模式示例（策略）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;

// 攻击策略接口
class AttackStrategy {
public:
    virtual ~AttackStrategy() = default;
    virtual void attack() const = 0;
    virtual string getName() const = 0;
};
```

```cpp
// 具体策略：近战攻击
class MeleeAttack : public AttackStrategy {
public:
    void attack() const override {
        cout << "使用近战武器进行攻击！" << endl;
    }
    string getName() const override {
        return "近战攻击";
    }
};
```

策略模式示例（具体策略）

```cpp
// 具体策略：远程攻击
class RangedAttack : public AttackStrategy {
public:
    void attack() const override {
        cout << "使用远程武器进行攻击！" << endl;
    }
    string getName() const override {
        return "远程攻击";
    }
};

// 具体策略：魔法攻击
class MagicAttack : public AttackStrategy {
public:
    void attack() const override {
        cout << "释放魔法进行攻击！" << endl;
    }
    string getName() const override {
        return "魔法攻击";
    }
};
```

```cpp
// 角色类，持有攻击策略
class Character {
private:
    string name;
    unique_ptr<AttackStrategy> strategy;
public:
    Character(const string& n) : name(n) {}

    void setStrategy(unique_ptr<AttackStrategy> s) {
        strategy = move(s);
        cout << name << " 切换为 [" <<
        ↪  strategy->getName() << "] 策略。" << endl;
    }
    void attack() const {
        if (strategy) {
            cout << name << ": ";
            strategy->attack();
        } else {
            cout << name << " 还没有选择攻击方式！" <<
            ↪  endl;
        }
    }
};
```

策略模式示例（客户端）

```
74      cout << "=== 策略模式示例：角色攻击方式灵活切换 ===" << endl;
75
76      Character hero("勇者");
77
78      // 切换为近战攻击
79      hero.setStrategy(make_unique<MeleeAttack>());
80      hero.attack();
81
82      // 切换为远程攻击
83      hero.setStrategy(make_unique<RangedAttack>());
84      hero.attack();
85
86      // 切换为魔法攻击
87      hero.setStrategy(make_unique<MagicAttack>());
88      hero.attack();
89
90      cout << "策略模式让角色可以灵活切换攻击方式！" << endl;
```

策略模式优缺点

优点

- 灵活性，策略模式可以让算法独立于使用它的客户端而变化。
- 易于扩展，可以轻松添加新的策略。

缺点

- 需要为每个策略创建一个类，会增加类的数量。
- 客户端需要知道所有策略，并选择合适的策略。

模板方法模式（Template Method Pattern）

模板方法模式是一种行为型设计模式，它在父类中定义算法的整体结构（即”模板方法”），并将部分步骤的实现延迟到子类。这样，子类可以在不改变算法整体流程的情况下，重新定义某些特定步骤的实现。

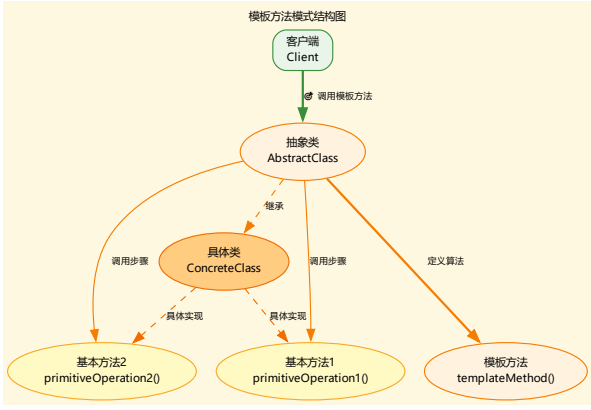
适用场景

- 需要复用算法的整体结构，但允许部分步骤有不同实现时。
- 希望将多个子类中的重复代码上移到父类，减少代码冗余。
- 需要对子类的扩展进行统一约束和控制，防止子类破坏算法结构。

模板方法模式结构

结构组成

- **抽象类（AbstractClass）**：定义算法的整体流程（模板方法），并声明若干抽象步骤。
- **模板方法（Template Method）**：在抽象类中实现，规定算法的执行顺序，调用若干基本操作（可为抽象或具体）。
- **具体类（ConcreteClass）**：继承抽象类，实现其中的抽象步骤，完成具体逻辑。



模板方法模式举例说明

游戏中的模板方法模式——角色升级流程 在一款游戏中，角色升级通常需要经历多个步骤(如打怪、完成任务、收集材料等)。虽然每个步骤的具体实现可能不同，但升级的整体流程是固定的。此时可以采用**模板方法模式**，将升级流程的骨架抽象出来，具体步骤由子类实现，其实现思路如下：

- 定义抽象类（`CharacterLevelUpProcess`），定义升级流程的模板方法，规定各步骤的执行顺序。
- 定义具体子类（`WarriorLevelUp`、`MageLevelUp`），实现抽象类中的具体步骤（如打怪、完成任务等）。
- 通过模板方法，保证升级流程统一，便于扩展和维护不同职业的升级细节。

模板方法模式示例（抽象类）

```cpp
// 抽象类：角色升级流程
class CharacterLevelUpProcess {
public:
    // 模板方法：规定升级流程
    void levelUp() {
        cout << "开始升级流程" << endl;
        fightMonsters();
        completeQuests();
        specialStep(); // 钩子方法，可选
        upgrade();
        cout << "升级流程结束" << endl;
    }
    virtual ~CharacterLevelUpProcess() = default;
protected:
    virtual void fightMonsters() = 0;
    virtual void completeQuests() = 0;
    virtual void upgrade() = 0;
    virtual void specialStep() {}
};
```

```cpp
// 具体子类：战士升级
class WarriorLevelUp : public CharacterLevelUpProcess {
protected:
    void fightMonsters() override {
        cout << "战士疯狂刷怪，提升经验值..." << endl;
    }
    void completeQuests() override {
        cout << "战士完成主线与支线任务..." << endl;
    }
    void upgrade() override {
        cout << "战士升级！力量和生命值大幅提升！" << endl;
    }
    void specialStep() override {
        cout << "战士进行武器锻造，提升攻击力..." << endl;
    }
};
```

模板方法模式示例（具体类）

```cpp
// 具体子类：法师升级
class MageLevelUp : public CharacterLevelUpProcess {
protected:
    void fightMonsters() override {
        cout << "法师远程施法击败怪物，积累经验..." << endl;
    }
    void completeQuests() override {
        cout << "法师完成魔法试炼任务..." << endl;
    }
    void upgrade() override {
        cout << "法师升级！魔法攻击和法力值大幅提升！" << endl;
    }
    void specialStep() override {
        cout << "法师学习新魔法技能..." << endl;
    }
};
```

```cpp
// 具体子类：游侠升级
class RangerLevelUp : public CharacterLevelUpProcess {
protected:
    void fightMonsters() override {
        cout << "游侠利用弓箭远程猎杀怪物..." << endl;
    }
    void completeQuests() override {
        cout << "游侠完成探索与追踪任务..." << endl;
    }
    void upgrade() override {
        cout << "游侠升级！敏捷和暴击率提升！" << endl;
    }
    // 不重写 specialStep，使用默认空实现
};
```

模板方法模式示例（客户端）

```
79      cout << "模板方法模式示例：角色升级流程" << endl;
80
81      unique_ptr<CharacterLevelUpProcess> warrior = make_unique<WarriorLevelUp>();
82      unique_ptr<CharacterLevelUpProcess> mage = make_unique<MageLevelUp>();
83      unique_ptr<CharacterLevelUpProcess> ranger = make_unique<RangerLevelUp>();
84
85      cout << "\n1. 战士升级流程" << endl;
86      warrior->levelUp();
87
88      cout << "\n2. 法师升级流程" << endl;
89      mage->levelUp();
90
91      cout << "\n3. 游侠升级流程" << endl;
92      ranger->levelUp();
93
94      cout << "\n模板方法模式保证了升级流程统一，细节可扩展" << endl;
```

模板方法模式优缺点

优点

- 提高代码复用性，模板方法模式将算法的通用部分抽象出来，避免重复代码。
- 提高代码可维护性，模板方法模式将算法的具体实现细节封装在子类中，易于维护和扩展。

缺点

- 需要为每个子类创建一个具体的实现类，增加了类的数量。
- 模板方法模式限制了子类的灵活性，子类必须遵守父类的模板方法。

访问者模式（Visitor Pattern）

访问者模式用于将作用于对象结构中各元素的操作与元素本身进行解耦。通过在不修改元素类的前提下，为其添加新的操作，增强系统的扩展性和灵活性。

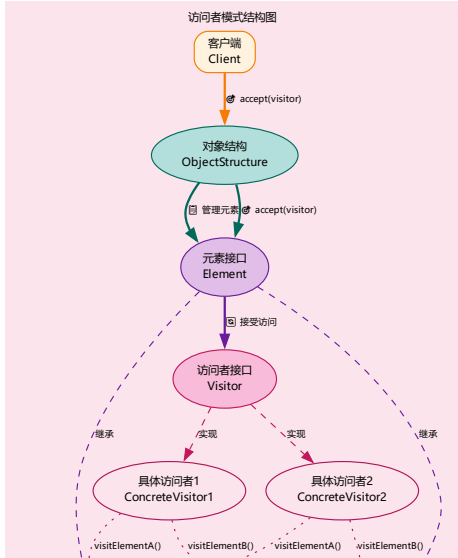
适用场景

- 对象结构包含多种类型的元素，需要对不同类型元素执行不同的操作。
- 需要在不修改元素类的情况下，为其添加新的操作，且这些操作经常变化。
- 希望将数据结构与操作解耦，避免操作”污染”元素类，便于维护和扩展。
- 元素类结构相对稳定，但对其操作需求经常变化。

访问者模式结构

结构组成

- **抽象访问者（Visitor）**：定义了对各个元素的操作接口。
- **具体访问者（ConcreteVisitor）**：实现抽象访问者接口，提供具体的操作实现。
- **抽象元素（Element）**：定义了接受访问者的接口。
- **具体元素（ConcreteElement）**：实现抽象元素接口，提供具体的操作实现。
- **对象结构**



访问者模式举例说明

RPG 游戏中的访问者模式应用 在 RPG 游戏开发中，常常需要对不同类型的角色（如战士、法师、游侠等）执行多种操作（如显示状态、计算战斗力、发放奖励等）。如果将这些操作直接写在角色类中，会导致角色类臃肿且难以扩展。此时，**访问者模式**可以很好地将操作与角色类型解耦，提升系统的可扩展性和灵活性。实现思路如下：

- 定义**抽象访问者**（`Visitor`），声明针对不同角色类型的访问接口。
- 定义**具体访问者**（如 `StatusVisitor`、`RewardVisitor`），实现具体的操作逻辑。
- 定义**抽象元素**（`GameCharacter`），声明接受访问者的接口（`accept` 方法）。
- 定义**具体元素**（`Warrior`、`Mage`、`Ranger`），实现 `accept` 方法，并在其中调用访问者的对应方法。
- 定义**对象结构**（`Party`），用于管理和遍历多个角色，统一接受访问者操作。

通过访问者模式，可以在不修改角色类的情况下，灵活地为其添加新的操作，符合开闭原则。

访问者模式示例（抽象访问者）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;

// 抽象访问者
class GameCharacter;
class Warrior;
class Mage;
class Ranger;

class Visitor {
public:
    virtual void visitWarrior(Warrior* w) = 0;
    virtual void visitMage(Mage* m) = 0;
    virtual void visitRanger(Ranger* r) = 0;
    virtual ~Visitor() = default;
};
```

```cpp
// 抽象元素
class GameCharacter {
public:
    virtual void accept(Visitor* visitor) = 0;
    virtual string getName() const = 0;
    virtual ~GameCharacter() = default;
};
```

访问者模式示例（具体访问者）

```cpp
// 具体元素：战士
class Warrior : public GameCharacter {
    string name;
    int hp;
    int attack;
public:
    Warrior(const string& n, int h, int a) : name(n), hp(h), attack(a) {}
    void accept(Visitor* visitor) override {
        visitor->visitWarrior(this);
    }
    string getName() const override { return name; }
    int getHP() const { return hp; }
    int getAttack() const { return attack; }
};
```

```cpp
// 具体元素：法师
class Mage : public GameCharacter {
    string name;
    int mp;
    int magicAttack;
public:
    Mage(const string& n, int m, int ma) : name(n), mp(m), magicAttack(ma) {}
    void accept(Visitor* visitor) override {
        visitor->visitMage(this);
    }
    string getName() const override { return name; }
    int getMP() const { return mp; }
    int getMagicAttack() const { return magicAttack; }
};
```

访问者模式示例（具体访问者）

```cpp
// 具体元素：游侠
class Ranger : public GameCharacter {
    string name;
    int agility;
    int crit;
public:
    Ranger(const string& n, int ag, int cr) :
    ↪  name(n), agility(ag), crit(cr) {}
    void accept(Visitor* visitor) override {
        visitor->visitRanger(this);
    }
    string getName() const override { return name; }
    int getAgility() const { return agility; }
    int getCrit() const { return crit; }
};
```

```cpp
// 对象结构：队伍
class Party {
    vector<shared_ptr<GameCharacter>> members;
public:
    void addMember(shared_ptr<GameCharacter> c) {
        members.push_back(c);
    }
    void accept(Visitor* visitor) {
        for (auto& m : members) {
            m->accept(visitor);
        }
    }
};
```

访问者模式示例（具体访问者）

```cpp
// 具体访问者：显示状态
class StatusVisitor : public Visitor {
public:
    void visitWarrior(Warrior* w) override {
        cout << "战士 [" << w->getName() << "] HP: "
             << w->getHP() << " 攻击: " <<
             w->getAttack() << endl;
    }
    void visitMage(Mage* m) override {
        cout << "法师 [" << m->getName() << "] MP: "
             << m->getMP() << " 魔法攻击: " <<
             m->getMagicAttack() << endl;
    }
    void visitRanger(Ranger* r) override {
        cout << "游侠 [" << r->getName() << "] 敏捷: "
             << r->getAgility() << " 暴击: " <<
             r->getCrit() << endl;
    }
};
```

```cpp
// 具体访问者：计算战斗力
class PowerVisitor : public Visitor {
public:
    void visitWarrior(Warrior* w) override {
        int power = w->getHP() + w->getAttack() * 2;
        cout << "战士 [" << w->getName() << "] 战斗力:
             " << power << endl;
    }
    void visitMage(Mage* m) override {
        int power = m->getMP() + m->getMagicAttack()
             * 3;
        cout << "法师 [" << m->getName() << "] 战斗力:
             " << power << endl;
    }
    void visitRanger(Ranger* r) override {
        int power = r->getAgility() * 2 +
             r->getCrit() * 4;
        cout << "游侠 [" << r->getName() << "] 战斗力:
             " << power << endl;
    }
};
```

访问者模式示例（客户端）

```
120 // 具体访问者：发放奖励
121 class RewardVisitor : public Visitor {
122 public:
123     void visitWarrior(Warrior* w) override {
124         cout << "战士 [" << w->getName() << "] 获得奖
        ↪  励：强化武器" << endl;
125     }
126     void visitMage(Mage* m) override {
127         cout << "法师 [" << m->getName() << "] 获得奖
        ↪  励：新魔法书" << endl;
128     }
129     void visitRanger(Ranger* r) override {
130         cout << "游侠 [" << r->getName() << "] 获得奖
        ↪  励：高级箭矢" << endl;
131     }
132 };
```

```
138     cout << "访问者模式示例：RPG 角色队伍" << endl;
139
140     Party party;
141     party.addMember(make_shared<Warrior>("亚瑟", 150,
        ↪  40));
142     party.addMember(make_shared<Mage>("梅林", 120,
        ↪  60));
143     party.addMember(make_shared<Ranger>("罗宾", 100,
        ↪  30));
144
145     cout << "\n1. 显示角色状态：" << endl;
146     StatusVisitor statusVisitor;
147     party.accept(&statusVisitor);
148
149     cout << "\n2. 计算战斗力：" << endl;
150     PowerVisitor powerVisitor;
151     party.accept(&powerVisitor);
152
153     cout << "\n3. 发放奖励：" << endl;
154     RewardVisitor rewardVisitor;
155     party.accept(&rewardVisitor);
156
157     cout << "\n 访问者模式展示完毕" << endl;
```

访问者模式优缺点

优点

- 将操作与数据结构分离，符合开闭原则。
- 可以灵活地添加新的操作，而不需要修改元素类。

缺点

- 需要为每个元素类型创建一个具体的访问者类，增加了系统的复杂性。
- 需要客户端知道具体的访问者类，增加了耦合度。

命令模式 (Command Pattern)

命令模式将请求封装为独立的对象，允许请求的参数化、排队和撤销，提高系统的灵活性和可扩展性。通过将请求对象化，可以更方便地实现撤销、重做、日志记录等功能，并且支持将请求与执行操作解耦，便于扩展新的命令类型。

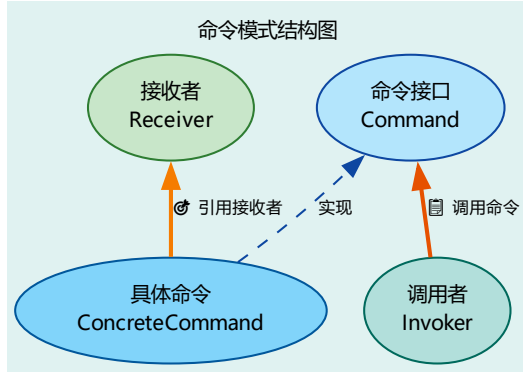
适用场景

- 需要将请求封装为独立的对象，以便进行参数化、排队、撤销和日志记录等操作。
- 需要支持撤销操作，如撤销重做、保存和恢复操作等。
- 需要支持修改日志，以便在系统崩溃时恢复修改。
- 需要将请求与执行操作解耦，便于扩展新的命令类型。

命令模式结构

结构组成

- **抽象命令（Command）**：定义了执行操作的接口。
- **具体命令（ConcreteCommand）**：实现抽象命令接口，定义具体的执行操作。
- **接收者（Receiver）**：包含具体的业务逻辑。
- **请求者（Invoker）**：负责调用命令对象执行请求。



命令模式举例说明

RPG 游戏中的命令模式应用 在 RPG 游戏开发中，命令模式可以用于封装玩家的各种操作(如移动、攻击、使用技能等)，实现操作的记录、撤销与重做、宏命令等功能。实现思路如下：

- 定义**抽象命令**（Command)，声明执行操作的接口。
- 定义**具体命令**（如 MoveCommand、AttackCommand、SkillCommand)，实现具体的操作逻辑。
- 定义**接收者**（Player)，包含实际的业务逻辑（如移动、攻击等方法)。
- 定义**请求者**（Invoker)，负责调用命令对象执行请求，并可实现命令的撤销、重做等功能。
- 支持**宏命令**（MacroCommand)，将一系列命令组合成一个复合命令，实现一键连招等功能。

通过命令模式，可以灵活地管理和扩展玩家操作，便于实现操作日志、回放、撤销/重做等高级功能，提升系统的可维护性和扩展性。

命令模式示例（抽象命令）

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <windows.h>
using namespace std;

// 抽象命令
class Command {
public:
    virtual ~Command() = default;
    virtual void execute() = 0;
    virtual void undo() = 0;
};

// 接收者：玩家
class Player {
private:
    string name;
    int x, y;
public:
    Player(const string& n) : name(n), x(0), y(0) {}

    void move(int dx, int dy) {
        x += dx;
        y += dy;
        cout << name << " 移动到位置 (" << x << ", "
        ↪ << y << ")" << endl;
    }

    void attack(const string& target) {
        cout << name << " 攻击了 " << target << "! "
        ↪ << endl;
    }

    void useSkill(const string& skill) {
        cout << name << " 使用了技能: " << skill <<
        ↪ endl;
    }

    void showPosition() {
        cout << name << " 当前坐标: (" << x << ", " <<
        ↪ y << ")" << endl;
    }
};
```

命令模式示例（具体命令）

```
42  // 具体命令：移动
43  class MoveCommand : public Command {
44  private:
45      Player* player;
46      int dx, dy;
47      int prevX, prevY;
48  public:
49      MoveCommand(Player* p, int dx, int dy) :
    ↪   player(p), dx(dx), dy(dy), prevX(0), prevY(0)
    ↪   {}
50
51      void execute() override {
52          // 记录上一次位置
53          player->showPosition();
54          prevX = dx;
55          prevY = dy;
56          player->move(dx, dy);
57      }
58
59      void undo() override {
60          player->move(-dx, -dy);
61          cout << "撤销移动操作" << endl;
62      }
63  };
64
65  // 具体命令：攻击
66  class AttackCommand : public Command {
67  private:
68      Player* player;
69      string target;
70  public:
71      AttackCommand(Player* p, const string& t) :
    ↪   player(p), target(t) {}
72
73      void execute() override {
74          player->attack(target);
75      }
76
77      void undo() override {
78          cout << "撤销攻击操作（无法真正撤销，仅做演示）"
    ↪   << endl;
79      }
80  };
```

命令模式示例（具体命令）

```
// 具体命令：使用技能
class SkillCommand : public Command {
private:
    Player* player;
    string skill;
public:
    SkillCommand(Player* p, const string& s) :
    ↪  player(p), skill(s) {}

    void execute() override {
        player->useSkill(skill);
    }

    void undo() override {
        cout << "撤销技能操作（无法真正撤销，仅做演示）"
        ↪  << endl;
    }
};

// 宏命令：组合多个命令
class MacroCommand : public Command {
private:
    vector<shared_ptr<Command>> commands;
public:
    void addCommand(shared_ptr<Command> cmd) {
        commands.push_back(cmd);
    }

    void execute() override {
        for (auto& cmd : commands) {
            cmd->execute();
        }
    }

    void undo() override {
        // 撤销顺序与执行相反
        for (auto it = commands.rbegin(); it !=
        ↪  commands.rend(); ++it) {
            (*it)->undo();
        }
    }
};
```

命令模式示例（请求者）

```
// 空命令
class NoCommand : public Command {
public:
    void execute() override {}
    void undo() override {}
};
```

```
// 请求者：操作记录与撤销
class Invoker {
private:
    vector<shared_ptr<Command>> history;
public:
    void executeCommand(shared_ptr<Command> cmd) {
        cmd->execute();
        history.push_back(cmd);
    }

    void undo() {
        if (!history.empty()) {
            history.back()->undo();
            history.pop_back();
        } else {
            cout << "没有可撤销的操作" << endl;
        }
    }
};
```

命令模式示例（客户端）

```
153     cout << "=== 命令模式 RPG 示例 ===" << endl;
154
155     // 创建玩家
156     Player player("勇者");
157
158     // 创建命令
159     auto move1 = make_shared<MoveCommand>(&player, 2,
        ↪  3);
160     auto attack1 =
        ↪  make_shared<AttackCommand>(&player, "史莱姆");
161     auto skill1 = make_shared<SkillCommand>(&player,
        ↪  "火球术");
162
163     // 宏命令：一键连招
164     auto macro = make_shared<MacroCommand>();
165     macro->addCommand(move1);
166     macro->addCommand(attack1);
167     macro->addCommand(skill1);
168
169     // 请求者
170     Invoker invoker;
171     cout << "\n1. 执行移动命令：" << endl;
172     invoker.executeCommand(move1);
```

```
173     cout << "\n2. 执行攻击命令：" << endl;
174     invoker.executeCommand(attack1);
175     cout << "\n3. 执行技能命令：" << endl;
176     invoker.executeCommand(skill1);
177     cout << "\n4. 执行宏命令（连招）：" << endl;
178     invoker.executeCommand(macro);
179     cout << "\n5. 撤销上一次操作（宏命令）：" << endl;
180     invoker.undo();
181     cout << "\n6. 撤销上一次操作（技能）：" << endl;
182     invoker.undo();
183     cout << "\n7. 撤销上一次操作（攻击）：" << endl;
184     invoker.undo();
185     cout << "\n8. 撤销上一次操作（移动）：" << endl;
186     invoker.undo();
187     cout << "\n9. 再次撤销（无操作）：" << endl;
188     invoker.undo();
```

命令模式优缺点

优点

- 将请求封装为独立的对象，方便参数化、排队、撤销和日志记录等操作。
- 支持撤销操作，如撤销重做、保存和恢复操作等。
- 支持修改日志，以便在系统崩溃时恢复修改。
- 将请求与执行操作解耦，便于扩展新的命令类型。

缺点

- 需要为每个命令类型创建一个具体的命令类，增加了系统的复杂性。
- 需要客户端知道具体的命令类，增加了耦合度。

状态模式（State Pattern）

状态模式允许一个对象在其内部状态发生变化时，自动切换其行为，使得看起来像是修改了其类。通过将与状态相关的行为封装到独立的状态类中，状态模式实现了状态与行为的解耦，提升了系统的灵活性和可维护性。

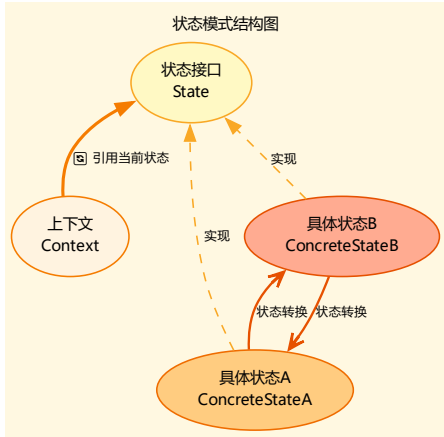
适用场景

- 对象的行为依赖于其内部状态，并且在运行时需要根据状态切换行为。
- 代码中存在大量基于状态的条件分支（如 if/else 或 switch），希望通过消除这些分支来优化结构。
- 需要将与状态相关的行为局部化到独立的状态类中，避免状态逻辑分散在主类中。

状态模式结构

结构组成

- **抽象状态（State）**：定义了状态的接口。
- **具体状态（ConcreteState）**：实现了抽象状态接口，定义了具体的状态行为。
- **上下文（Context）**：包含状态的引用，并负责状态的切换。



状态模式举例说明

RPG 游戏中的状态模式应用 在 RPG 游戏开发中，角色的行为通常取决于其当前状态（如正常、中毒、眩晕等）。如果将所有状态相关的逻辑直接堆叠在角色类中，会导致代码臃肿、难以维护和扩展。状态模式通过将每种状态的行为封装到独立的状态类中，实现了状态与行为的彻底解耦，大幅提升了系统的灵活性和可维护性。其核心实现思路如下：

- 定义**抽象状态**（`State`），统一声明所有状态应实现的接口。
- 定义**具体状态**（如 `NormalState`、`PoisonedState`、`StunnedState` 等），分别实现不同状态下的具体行为。
- 定义**上下文**（`CharacterContext`），持有当前状态的指针，并负责状态的切换与行为的委托。

状态模式示例（抽象状态）

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <windows.h>
using namespace std;

// 抽象状态
class State {
public:
    virtual ~State() = default;
    virtual void handle() = 0;
    virtual string getName() const = 0;
};
```

```cpp
// 具体状态：正常
class NormalState : public State {
public:
    void handle() override {
        cout << "角色处于【正常】状态，可以自由行动。"
        ↪  << endl;
    }
    string getName() const override {
        return "正常";
    }
};
```

状态模式示例（具体状态）

```cpp
// 具体状态：中毒
class PoisonedState : public State {
public:
    void handle() override {
        cout << "角色处于【中毒】状态，每回合损失生命值，行动受限！" << endl;
    }
    string getName() const override {
        return "中毒";
    }
};
```

```cpp
// 具体状态：眩晕
class StunnedState : public State {
public:
    void handle() override {
        cout << "角色处于【眩晕】状态，无法行动！" << endl;
    }
    string getName() const override {
        return "眩晕";
    }
};
```

状态模式示例（上下文）

```
48  // 上下文：角色
49  class CharacterContext {
50  private:
51      unique_ptr<State> state;
52  public:
53      CharacterContext() : state(nullptr) {}
54
55      void setState(unique_ptr<State> newState) {
56          if (state) {
57              cout << "角色状态由【" << state->getName()
                ↪  << "】切换为【" << newState->getName()
                ↪  << "】" << endl;
58          } else {
59              cout << "角色状态初始化为【" <<
                ↪  newState->getName() << "】" << endl;
60          }
61          state = move(newState);
62      }
```

```
64      void action() const {
65          if (state) {
66              state->handle();
67          } else {
68              cout << "角色无状态，无法行动。" << endl;
69          }
70      }
71
72      string getStateName() const {
73          return state ? state->getName() : "无状态";
74      }
75  };
```

状态模式示例（客户端）

```
82      cout << "状态模式示例：RPG 角色状态切换" << endl;
83      CharacterContext hero;
84      cout << "初始状态：" << hero.getStateName() << endl;
85      // 设置为正常状态
86      hero.setState(make_unique<NormalState>());
87      cout << "当前状态：" << hero.getStateName() << endl;
88      hero.action();
89      // 切换为中毒状态
90      hero.setState(make_unique<PoisonedState>());
91      cout << "当前状态：" << hero.getStateName() << endl;
92      hero.action();
93      // 切换为眩晕状态
94      hero.setState(make_unique<StunnedState>());
95      cout << "当前状态：" << hero.getStateName() << endl;
96      hero.action();
97      // 再切回正常状态
98      hero.setState(make_unique<NormalState>());
99      cout << "当前状态：" << hero.getStateName() << endl;
100     hero.action();
101     cout << "状态模式演示完毕。" << endl;
```

状态模式优缺点

优点

- 将每种状态的行为封装到独立的状态类中，便于扩展和维护。
- 状态与上下文对象解耦，状态切换灵活，代码结构清晰，易于理解和管理。
- 新增或修改状态时，无需修改上下文代码，降低了维护成本。

缺点

- 状态类数量随状态种类增加而增多，可能导致类爆炸，管理难度提升。
- 某些情况下，状态切换逻辑分散在各状态类中，可能导致切换流程不够集中，维护复杂度上升。
- 如果状态之间存在较多交互，可能引入额外的耦合和实现难度。

目录

SOLID 原则简介

什么是 SOLID 原则?

SOLID 是面向对象设计和编程的五个基本原则的首字母缩写:

- **S** - Single Responsibility Principle (单一职责原则)
- **O** - Open/Closed Principle (开闭原则)
- **L** - Liskov Substitution Principle (里氏替换原则)
- **I** - Interface Segregation Principle (接口隔离原则)
- **D** - Dependency Inversion Principle (依赖倒置原则)

SOLID 原则的重要性

- 提高代码的可读性和可维护性
- 降低代码的耦合度
- 增强代码的灵活性和扩展性
- 促进代码的重用

单一职责原则 (SRP)

原则定义

一个类应该只有一个引起它变化的原因。一个类只负责一项职责，即一个类只做一件事。如果一个类承担的职责过多，当某一职责发生变化时，可能会影响到其他职责的实现，导致代码难以维护和扩展。

SRP 的优势

- 提高代码的可维护性：每个类只负责一项职责，代码结构清晰，便于理解和维护。
- 降低变更带来的风险：当某一职责发生变化时，只需修改对应的类，不会影响到其他无关的功能。
- 便于代码复用：职责单一的类更容易被复用到其他场景中。
- 促进团队协作：不同开发人员可以专注于不同的类或模块，减少冲突，提高开发效率。
- 有助于单元测试：单一职责的类更容易编写针对性的测试用例，提高测试覆盖率和准确性。

单一职责原则（SRP）示例

设计模式案例：抽象工厂模式中的职责分离 以**抽象工厂模式（Abstract Factory Pattern）**为例，假设我们要开发一个奶茶店系统。系统中有奶茶（如 `Tea`）和配料（如 `Topping`）两大类产品。每个产品族（如原味、甜味、巧克力味）都需要生产一组相关的奶茶和配料。
 在抽象工厂模式中，`Tea` 和 `Topping` 分别作为独立的抽象产品类，各自只负责自身的属性和行为；`TeaFactory` 作为抽象工厂，只负责定义创建奶茶和配料的接口；每个具体工厂（如 `OriginalFactory`、`SweetFactory`、`ChocolateFactory`）只负责生产一组特定口味的奶茶和配料。
 这样，每个类都只承担单一职责：产品类只关注产品本身，工厂类只关注产品的创建，产品族的扩展和维护都非常清晰，充分体现了单一职责原则。

单一职责原则（SRP）示例

不遵循 SRP 的后果（以抽象工厂模式为例）

- **代码难以维护**：如果将所有奶茶和配料的创建逻辑都写在一个大工厂类或产品类中，添加或修改产品时需要频繁修改同一个类，容易引入 bug。
- **可读性和可测试性降低**：产品类既包含自身属性又包含其他产品或配料的逻辑，代码臃肿，难以理解和测试。
- **复用性差**：其他饮品或配料想要复用某种创建逻辑，必须复制粘贴相关代码，无法灵活组合。
- **团队协作受阻**：多人同时修改同一个大类时，容易产生冲突，影响开发效率。
- **扩展性受限**：新增产品族或变更产品逻辑时，必须修改已有类，系统演化困难。

SRP 在设计模式中的体现

- **策略模式（Strategy Pattern）**：将可变的算法行为抽象为独立的策略类，每个策略类只负责一种算法，便于扩展和维护，避免算法混杂在主业务类中。
- **命令模式（Command Pattern）**：将请求的每个操作封装为独立的命令对象，每个命令类只负责一种具体操作，调用者与执行者解耦，便于扩展新命令。
- **工厂模式（Factory Pattern）**：将对象的创建逻辑集中到工厂类中，产品类只关注自身职责，工厂类只负责对象的实例化，职责分明。
- **装饰器模式（Decorator Pattern）**：将附加功能封装到装饰器类中，核心组件类只关注核心职责，装饰器类只负责扩展功能，二者职责清晰。
- **观察者模式（Observer Pattern）**：主题对象只负责自身状态的维护，观察者对象只负责响应状态变化，二者各司其职，便于独立演化。
- **状态模式（State Pattern）**：每个状态类只负责一种状态下的行为，状态切换逻辑与具体行为分离，便于扩展新状态。

开闭原则 (OCP)

原则定义

软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。这意味着在添加新功能时，不应修改现有代码，而是通过扩展来实现。

OCP 的优势

- 提高系统的可扩展性：容易添加新功能而不影响现有代码。
- 降低风险：修改现有代码可能引入 bug，扩展则更安全。
- 促进代码复用：现有代码保持稳定，可被新扩展复用。
- 便于测试：新功能可独立测试，不干扰原有系统。
- 支持持续集成：新扩展可无缝融入现有系统。

开闭原则（OCP）示例

现实案例：策略模式下的攻击方式扩展 以**策略模式（Strategy Pattern）**为例，假设在一款游戏中，角色可以选择不同的攻击方式（如近战攻击、远程攻击、魔法攻击等）。
 如果采用传统实现，角色类中可能通过大量 `if-else` 或 `switch` 语句判断攻击类型，每增加一种新攻击方式都需要修改角色类的代码，容易引入 bug，违背开闭原则。
 使用策略模式后，每种攻击方式都实现统一的 `AttackStrategy` 接口，角色类只依赖该接口。
 当需要扩展新的攻击方式时，只需新增一个策略类实现 `AttackStrategy`，无需修改角色类的任何代码。
 这正体现了开闭原则：系统对新功能（新攻击方式）的扩展开放，对已有代码的修改关闭，保证了系统的稳定性和可维护性。

开闭原则（OCP）示例

不遵循 OCP 的后果（以策略模式为例）

- **代码脆弱性增加**：每次添加新攻击方式都要修改角色类，容易引入新 bug 或破坏原有功能。
- **维护成本上升**：修改核心角色类可能导致连锁反应，需要大量回归测试，消耗时间和资源。
- **可扩展性受限**：系统变得僵化，添加新攻击方式的风险高，导致开发效率低下。
- **版本控制复杂**：频繁修改角色类增加合并冲突的风险，影响团队协作。
- **系统稳定性下降**：积累的修改可能使代码难以理解和预测，增加出错概率。

OCP 在设计模式中的体现

- **策略模式（Strategy Pattern）**：通过定义算法族并封装每个算法，允许在运行时切换算法，实现对扩展开放。
- **装饰器模式（Decorator Pattern）**：动态地为对象添加新行为，而不修改其结构，支持功能扩展。
- **工厂模式（Factory Pattern）**：通过子类决定实例化哪个类，实现对新类型的扩展而不改动现有代码。
- **观察者模式（Observer Pattern）**：允许添加新观察者而不修改主题类，实现松耦合扩展。
- **模板方法模式（Template Method Pattern）**：在超类中定义算法骨架，子类重写具体步骤，实现扩展而不改动算法结构。
- **桥接模式（Bridge Pattern）**：将抽象与实现分离，允许两者独立变化，支持扩展。

里氏替换原则 (LSP)

原则定义

子类对象必须能够替换掉所有父类对象，而不影响程序的正确性。继承时，子类应扩展父类行为，而不应改变或限制它。

LSP 的优势

- 确保继承关系的正确性：子类可无缝替换父类，提高代码的可靠性。
- 促进多态使用：安全地使用多态特性，实现灵活的代码设计。
- 提高可维护性：修改子类不会意外影响依赖父类的代码。
- 便于扩展：允许通过继承添加新功能，而不破坏现有系统。
- 支持开闭原则：通过子类扩展实现 OCP。

里氏替换原则（LSP）示例

现实案例：模板方法模式中的角色升级 假设有一个抽象类 `CharacterLevelUpProcess`，定义了角色升级的整体流程（模板方法），并声明了若干抽象步骤（如打怪、完成任务、收集材料等）。
 子类如 `WarriorLevelUp` 和 `MageLevelUp` 分别实现这些步骤，体现不同职业的升级细节。
 如果某个子类在重写抽象步骤时，破坏了父类模板方法的整体流程（比如直接跳过某些关键步骤或改变流程顺序），那么用该子类对象替换父类对象时，升级流程就会出错，违反 LSP。
 正确做法是：子类只重写具体步骤的实现，不改变父类模板方法的结构，这样任何子类都能安全替换父类，保证升级流程的正确性。

里氏替换原则（LSP）示例

不遵循 LSP 的后果（以模板方法为例）

- **多态失效**：子类如果随意更改模板方法流程，父类引用指向子类对象时，整体算法可能被破坏，失去多态的意义。
- **代码不可靠**：依赖父类流程的代码在子类替换后可能出现逻辑错误，降低系统稳定性。
- **维护困难**：需要为每个子类单独检查流程正确性，增加维护和测试难度。
- **扩展受限**：继承关系不安全，开发者不敢放心扩展新子类，影响系统灵活性。
- **测试复杂**：需要为每个子类单独测试完整流程，增加测试负担。

LSP 在设计模式中的体现

- **模板方法模式（Template Method Pattern）**：子类重写具体步骤，但必须遵守父类的算法结构。
- **策略模式（Strategy Pattern）**：所有策略类实现同一接口，确保可替换性。
- **工厂方法模式（Factory Method Pattern）**：子工厂创建的产品必须符合产品接口。
- **组合模式（Composite Pattern）**：叶节点和复合节点统一接口，确保替换不影响树结构。
- **命令模式（Command Pattern）**：所有命令对象实现同一接口，支持替换。
- **状态模式（State Pattern）**：状态类实现同一接口，确保状态切换不违反 LSP。

接口隔离原则 (ISP)

原则定义

客户端不应被迫依赖于它们不使用的接口。接口应细粒度设计，只包含客户端需要的功能。

ISP 的优势

- 降低耦合：客户端只依赖所需接口，减少不必要依赖。
- 提高灵活性：易于修改和扩展接口而不影响无关客户端。
- 便于测试：小接口更容易 mock 和测试。
- 支持单一职责：接口职责单一，与 SRP 相辅相成。
- 改善可维护性：变化只影响相关客户端。

接口隔离原则（ISP）示例

现实案例：音频播放器适配器）假设有一个音频播放器系统，原本只支持播放 MP3 格式（`Mp3Player`），现在需要支持 MP4 和 VLC 格式。
 如果将所有播放方法（如 `playMp3`、`playMp4`、`playVlc`）都放在一个大接口 `MediaPlayer` 中，则即使只播放 MP3 的类也必须实现 MP4 和 VLC 方法（即使为空），这违反了接口隔离原则。
 正确做法：将不同格式的播放功能拆分为不同接口（如 `Mp3Playable`、`Mp4Playable`、`VlcPlayable`），播放器只实现自己需要的接口。
 适配器模式通过组合这些小接口，实现灵活扩展和最小依赖，完美体现了 ISP。

接口隔离原则（ISP）示例

不遵循 ISP 的后果（以适配器为例）

- **耦合度高**：只支持 MP3 的播放器也被迫实现 MP4、VLC 等无关方法，增加冗余代码。
- **灵活性差**：新增或修改某种格式的播放方法时，所有实现大接口的类都受影响。
- **代码臃肿**：播放器类中充斥大量无用或空实现的方法，降低可读性。
- **测试困难**：需要为所有方法编写测试，即使某些方法实际不会被用到。
- **维护成本高**：任何接口变动都可能影响所有播放器实现，容易引入 bug。

ISP 在设计模式中的体现

- **适配器模式（Adapter Pattern）**：通过适配器提供最小接口，隐藏复杂实现。
- **外观模式（Facade Pattern）**：提供简化接口，隔离子系统复杂性。
- **装饰器模式（Decorator Pattern）**：每个装饰器实现特定接口，支持细粒度扩展。
- **代理模式（Proxy Pattern）**：代理实现与主体相同的接口，但可添加控制。
- **桥接模式（Bridge Pattern）**：分离抽象和实现接口，支持独立变化。
- **组合模式（Composite Pattern）**：统一组件接口，但允许叶节点只实现必要方法。

依赖倒置原则 (DIP)

原则定义

高层模块不应依赖低层模块，二者都应依赖抽象。抽象不应依赖细节，细节应依赖抽象。

DIP 的优势

- 降低耦合：通过抽象解耦高层和低层模块。
- 提高灵活性：易于替换实现而不影响高层。
- 便于测试：可使用 mock 实现测试高层逻辑。
- 支持开闭原则：通过新实现扩展功能。
- 促进模块化：鼓励使用接口和抽象类。

依赖倒置原则（DIP）示例

现实案例：多人游戏任务进度通知（观察者模式）传统做法：任务进度类直接依赖具体玩家类，通知时需要逐个调用具体玩家的更新方法。如果玩家类型变化或增加新类型，任务进度类必须修改，导致高耦合，违背 DIP。
正确做法：定义 `Observer`（观察者）接口，所有玩家类实现该接口。任务进度类（主题）只依赖 `Observer` 接口，不关心具体玩家类型。
当有新玩家类型加入时，只需实现 `Observer` 接口，无需修改任务进度类，实现了高层（任务进度）依赖抽象，低层（玩家）依赖抽象，完美体现依赖倒置原则。

依赖倒置原则（DIP）示例

不遵循 DIP 的后果（以观察者模式为例）

- **高耦合**：任务进度类直接依赖具体玩家类，新增或修改玩家类型时需修改任务进度类。
- **灵活性低**：玩家类型变化会影响任务进度类，扩展新玩家不便。
- **测试困难**：任务进度类与具体玩家强耦合，难以对任务进度逻辑单独测试。
- **可维护性差**：代码臃肿，维护和扩展成本高。
- **扩展受限**：新增玩家类型或通知方式时，必须修改高层代码，违背 OCP。

DIP 在设计模式中的体现

- **工厂模式（Factory Pattern）**：通过工厂创建对象，高层依赖抽象而非具体类。
- **依赖注入（Dependency Injection）**：将依赖通过构造函数或 setter 注入，实现倒置。
- **策略模式（Strategy Pattern）**：高层依赖策略接口，低层实现具体策略。
- **观察者模式（Observer Pattern）**：主题依赖观察者接口，具体观察者实现接口。
- **桥接模式（Bridge Pattern）**：抽象层依赖实现接口，支持多种实现。
- **命令模式（Command Pattern）**：调用者依赖命令接口，具体命令实现细节。

SOLID 原则总结

SOLID 原则的相互关系

- **SRP** 为 OCP 提供基础，确保职责单一便于扩展
- **OCP** 依赖 LSP，通过继承实现扩展
- **LSP** 是 OCP 的基础，保证继承关系的正确性
- **ISP** 支持 DIP，通过接口隔离实现依赖倒置
- **DIP** 是 OCP 的高层实现，通过抽象实现解耦

SOLID 原则的应用实践

- 在设计类和接口时首先考虑单一职责
- 使用抽象和接口来实现开闭原则
- 确保继承关系符合里氏替换原则
- 将大接口拆分为小接口以遵循接口隔离
- 通过依赖注入实现依赖倒置

SOLID 原则与设计模式的关系

设计模式对 SOLID 原则的支持

SOLID 原则是设计模式的基础，而设计模式则是 SOLID 原则的具体实现方式：

- **策略模式** → OCP + DIP
- **工厂模式** → DIP + SRP
- **装饰器模式** → OCP + ISP
- **观察者模式** → OCP + DIP
- **适配器模式** → ISP + LSP
- **命令模式** → SRP + OCP
- **模板方法** → LSP + OCP
- **组合模式** → LSP + OCP
- **桥接模式** → OCP + DIP
- **外观模式** → ISP + DIP

学习建议

- 先掌握 SOLID 原则，再学习设计模式
- 在实际项目中结合使用原则和模式
- 通过代码重构实践 SOLID 原则
- 编写单元测试验证原则的正确性

目录

23 种 GoF 设计模式的完整总结

创建型模式 (5 种)

- 单例：唯一实例，全局访问
- 工厂方法：延迟创建，子类决定
- 抽象工厂：产品家族，系列创建
- 建造者：分步构建，复杂对象
- 原型：对象克隆，复制创建

结构型模式 (7 种)

- 适配器：接口转换，兼容性
- 桥接：抽象分离，解耦合
- 组合：树形结构，整体-部分
- 装饰器：动态扩展，功能增强
- 外观：简化接口，统一入口
- 享元：对象共享，内存优化
- 代理：访问控制，权限管理

设计模式总结 (续)

行为型模式 (11 种)

- 责任链：请求传递，多级处理
- 命令：请求封装，支持撤销
- 解释器：语言解析，文法表示
- 迭代器：顺序访问，遍历集合
- 中介者：对象交互，解耦通信
- 备忘录：状态保存，支持回退
- 观察者：事件通知，一对多依赖
- 状态：状态变迁，行为改变
- 策略：算法选择，可互换
- 模板方法：算法框架，子类扩展
- 访问者：双重分派，操作扩展

学习路径建议

入门级学习路径

1. **单例模式** - 理解对象唯一性概念
2. **工厂方法** - 掌握对象创建模式
3. **观察者模式** - 学习事件驱动编程
4. **策略模式** - 理解算法封装思想

进阶级学习路径

1. **装饰器模式** - 掌握对象功能扩展
2. **组合模式** - 理解树形数据结构
3. **命令模式** - 学习撤销重做机制
4. **模板方法** - 掌握算法框架设计

学习路径建议 (续)

高级学习路径

1. **访问者模式** - 双重分派技术
2. **解释器模式** - 语言处理基础
3. **桥接模式** - 抽象与实现分离
4. **中介者模式** - 复杂对象交互

实践建议

- 每个模式都要动手实现，加深理解
- 分析现有代码，识别可应用的设计模式
- 结合实际项目，灵活运用设计模式
- 持续学习，设计模式是软件开发的基石

设计模式的应用原则

SOLID 原则与设计模式

- **S - 单一职责原则**：每个类应该只有一个职责
- **O - 开闭原则**：对扩展开放，对修改关闭
- **L - 里氏替换原则**：子类可以替换父类
- **I - 接口隔离原则**：接口应该小而专一
- **D - 依赖倒置原则**：依赖抽象而不依赖具体

设计模式遵循的原则

- 大多数设计模式都遵循 SOLID 原则
- 设计模式是这些原则的具体体现
- 理解原则才能更好地应用模式
- 原则是基础，模式是应用

设计模式的优势与局限性

主要优势

- 提高代码的可复用性
- 增强代码的可维护性
- 促进代码的可扩展性
- 降低代码的耦合度
- 提高开发效率
- 便于团队协作
- 提升软件质量

可能的局限性

- 增加代码复杂度
- 可能影响性能
- 学习成本较高
- 过度使用可能适得其反

如何选择合适的设计模式

选择原则

- 根据具体问题选择最合适的设计模式
- 不要为了使用模式而使用模式
- 考虑团队成员的理解程度
- 权衡模式的复杂度和带来的收益
- 优先考虑简单有效的解决方案

实践建议

- 多读优秀开源代码，学习模式应用
- 参与代码审查，讨论设计决策
- 持续学习，跟踪设计模式的新发展
- 在实践中检验和改进自己的理解

参考资料与学习资源

经典著作

- **《Clean Code》**
- 《Agile Software Development, Principles, Patterns, and Practices》
- 《Refactoring: Improving the Design of Existing Code》
- 《Clean Architecture》
- 《设计模式：可复用面向对象软件的基础》
- 《Head First 设计模式》
- 《大话设计模式》
- 《设计模式之禅》

感谢学习！

设计模式的学习是一个持续的过程

通过实践和应用，才能真正掌握这些宝贵的经验

祝您编程愉快！