23种经典设计模式(附c++实现代码)

- **概述**

  设计模式，**是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结**。

  描述了在软件设计过程中的**一些不断重复发生的问题**，以及该问题的解决方案。

  是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。

  其目的是**为了提高代码的可重用性、代码的可读性和代码的可靠性**。

- **作用**

  - 提高思维能力、编程能力和设计能力。
  - 使程序**设计更加标准化**、代码编制**更加工程化**，使软件开发效率大大提高，从而**缩短软件的开发周期**。
  - 使设计的代码**可重用性高、可读性强、可靠性高、灵活性好、可维护性强**。
- 部分**转载**的原文链接

  - **软件设计模式概述**
  - CS-Notes **设计模式**
- 本文**示例代码**

  design-mode

7种开发原则

开闭原则

- 定义

  - `Open Closed Principle，OCP`

  - 软件实体应当**对扩展开放，对修改关闭**。即, 当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

- 作用
    - 对**软件测试**的影响

      测试时**只需要对扩展的代码进行测试**就可以了，因为原有的测试代码仍然能够正常运行。

    - 提高代码的**可复用**性

      粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。

    - 提高软件的**可维护**性

      稳定性高和延续性强，从而易于扩展和维护。

里氏替换原则

- 定义
    - `Liskov Substitution Principle，LSP`
    - **继承必须确保基类所拥有的性质在子类中仍然成立**。即，子类可以扩展父类的功能，但不能改变父类原有的功能。
- 作用
    - 是实现开闭原则的重要方式之一。
    - 克服了继承中重写父类造成的可复用性变差的缺点。
    - 类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

依赖倒置原则

- 定义
    - `Dependence Inversion Principle，DIP`
    - **要面向接口编程，不要面向实现编程。**即，高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。
- 作用
    - 可以**降低类间的耦合性**。
    - 可以减少并行开发引起的风险。
    - 可以**提高代码的可读性**和可维护性。

单一职责原则

- 定义
  - `Single Responsibility Principle，SRP`
  - 单一职责原则规定**一个类应该有且仅有一个引起它变化的原因**，否则类应该被拆分。
  - 如果一个对象承担了太多的职责，至少存在以下两个缺点
    - 一个职责的变化可能会削弱或者抑制这个类实现其他职责的能力；
    - **当客户端需要该对象的某一个职责时，不得不将其他不需要的职责全都包含进来，从而造成冗余代码**或代码的浪费。
- 作用
  - 降低类的复杂度。一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多。
  - **提高类的可读性**。复杂性降低，自然其可读性会提高。
  - 提高系统的可维护性。可读性提高，那自然更容易维护了。
  - 变更引起的风险降低。变更是必然的，如果单一职责原则遵守得好，当修改一个功能时，可以显著降低对其他功能的影响。

接口隔离原则

- 定义
  - `Interface Segregation Principle，ISP`
  - **客户端不应该被迫依赖于它不使用的方法**
  - **一个类对另一个类的依赖应该建立在最小的接口上**
  - 与单一职责原则的区别
    - 单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离。
    - 单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。
- 作用
  - **将臃肿庞大的接口分解为多个粒度小的接口**，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
  - 接口隔离提高了系统的内聚性，**减少了对外交互**，降低了系统的耦合性。
  - 使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。
  - 能**减少项目工程中的代码冗余**。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

迪米特法则

- 定义
  - `Law of Demeter，LoD`，又叫作**最少知识原则**（`Least Knowledge Principle，LKP`）
  - **如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用**，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

- 作用
  - 降低了类之间的耦合度，提高了模块的相对独立性。
  - 由于亲合度降低，从而提高了类的可复用率和系统的扩展性。

合成复用原则

- 定义
  - `Composite Reuse Principle，CRP`，又叫**组合/聚合复用原则**（`Composition/Aggregate Reuse Principle，CARP`）。
  - 在软件复用时，要尽量**先使用组合或者聚合等关联关系**来实现，其次才考虑使用继承关系来实现。
- 作用
  - 通常类的复用分为**继承复用**和**合成复用**两种，继承复用虽然有简单和易实现的优点，但它也存在以下缺点:
    - **继承复用破坏了类的封装性**。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。
    - **子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化**，这不利于类的扩展与维护。
    - 它**限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。**
  - 采用组合或聚合复用时，可以**将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能**，它有以下优点:
    - 它维持了类的封装性。因为**成分对象的内部细节是新对象看不见的**，所以这种复用又称为“黑箱”复用。
    - 新旧类之间的耦合度低。这种复用所需的依赖较少，**新对象存取成分对象的唯一方法是通过成分对象的接口**。
    - 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

23种设计模式

> 设计模式有两种分类方法，即根据模式的目的来分和根据模式的作用的范围来分。

- **根据目的来分**
    - `创建型模式`：用于描述“怎样创建对象”，它的主要特点是“**将对象的创建与使用分离**”。 如, 单例、原型、工厂方法、抽象工厂、建造者等 5 种。
    - `结构型模式`：用于描述如何将类或对象**按某种布局组成更大的结构**。 如, 代理、适配器、桥接、装饰、外观、享元、组合等 7 种。
    - `行为型模式`：用于描述**类或对象之间怎样相互协作**共同完成单个对象都无法单独完成的任务，以及怎样分配职责。如, 模板方法、策略、命令、责任链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种。
- **根据作用范围来分**
    - `类模式`：用于处理类与子类之间的关系，这些关系**通过继承来建立**，是静态的，在**编译时刻便确定**下来了。如, 工厂方法、（类）适配器、模板方法、解释器属于该模式。
    - `对象模式`：用于处理对象之间的关系，这些关系可以**通过组合或聚合来实现**，在**运行时刻是可以变化**的，更具动态性。除了以上 4 种，其他的都是对象模式。

创建型 - 单例

- 含义

    `Singleton` 确保**一个类只有一个实例**，并提供该实例的全局访问点。

- 实现方式
    - 使用**一个私有构造函数**、**一个私有静态变量**以及**一个公有静态函数**来实现。
    - 私有构造函数保证了不能通过构造函数来创建对象实例，只能通过公有静态函数返回唯一的私有静态变量。
- 应用场景
    - 某类**只要求生成一个对象**的时候，如一个班中的班长、每个人的身份证号等。
    - **当对象需要被共享**的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。
    - **当某类需要频繁实例化**，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

- 示例
  - **懒汉式**

```
#pragma once

#include <mutex>

namespace singleton {

/// @brief 懒汉式单例. 类加载时没有生成单例, 第一次调用 getInstance 方法时才去创建
class LazySingleton {
public:
    LazySingleton* getInstance() {
        if (!instance_) {
            std::unique_lock<std::mutex> lock(mutex_);
            if (!instance_) instance_ = new LazySingleton();    // 可能出现两个线程都执行了第一次 if 语句
        }
        return instance_;
    }

private:
    LazySingleton() {}
    static LazySingleton* instance_;
    static std::mutex mutex_;
};

LazySingleton* LazySingleton::instance_ = nullptr;

} // namespace singleton
```

- **饿汉式**

```
#pragma once

namespace singleton {

/// @brief 饿汉式单例. 类一旦加载就创建一个单例
/// 优点: 线程安全
/// 缺点: 丢失延迟实例化带来的节约资源的好处
class HungrySingleton {
public:
    HungrySingleton* getInstance() { return instance_; }

private:
    HungrySingleton() {}
    static HungrySingleton* instance_;
};

HungrySingleton* HungrySingleton::instance_ = new
HungrySingleton();

} // namespace singleton
```

- 扩展

  可扩展为**有限的多例（`Multitcm`）模式**，这种模式可生成有限个实例并保存在ArmyList中，客户需要时可随机获取

创建型 - 原型

- 含义

  `Prototype` 将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。

  复制一个对象分为浅拷贝和深拷贝:

  - **浅拷贝**: 给对象中的每个成员变量进行复制，就是把A1类中的变量直接赋给A2类中变量，属于**值传递**，但是涉及到有new之类内存分配的地方，他们却是**共享内存**的。[默认]
  - **深拷贝**: 不仅使用值传递，而是要每个变量都有自己一份独立的内存空间，互不干扰。

- 应用场景
    - 对象之间相同或相似，即只是个别的几个属性不同的时候。
    - 对象的创建过程比较麻烦，但复制比较简单的时候。

- 示例

```
#pragma once

#include <stdio.h>

namespace prototype {

/// @brief 需要从A的实例得到一份与A内容相同，但是又互不干扰的实例

/// @brief 抽象原型类
class AbstractPrototype {
public:
    AbstractPrototype() = default;
        virtual ~AbstractPrototype() {}

        virtual AbstractPrototype* clone() = 0;
};

/// @brief 具体原型类
class ConcretePrototype : public AbstractPrototype {
public:
        ConcretePrototype() = default;
        ~ConcretePrototype() {}

        AbstractPrototype* clone() {
        return new ConcretePrototype(*this);
    }

private:
        ConcretePrototype(const ConcretePrototype& other) {
        fprintf(stderr, "ConcretePrototype copy construct!\n");
    }
};

} // namespace prototype

void Test_02_prototype_impl_1() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    prototype::AbstractPrototype* ptr_a = new
prototype::ConcretePrototype();
    prototype::AbstractPrototype* ptr_b = ptr_a->clone();

    delete ptr_a;
    delete ptr_b;
}

// -------------------- Test_02_prototype_impl_1 --------------------
// ConcretePrototype copy construct!
```

创建型 - 工厂方法

- 含义

  `Factory Method` 定义一个用于创建产品的接口，由子类决定生产什么产品。

  - 优点:
    - 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
    - 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，**满足开闭原则**；
  - 缺点:

    每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。
- 应用场景

  - 客户只知道创建产品的工厂名，而不知道具体的产品名。
  - 创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口。
  - 客户不关心创建产品的细节，只关心产品的品牌。
- 扩展

  当需要生成的产品不多且不会增加，一个具体工厂类就可以完成任务时，可删除抽象工厂类。这时工厂方法模式将退化到**简单工厂模式**。

- 示例

```cpp
#pragma once

#include <stdio.h>

namespace factory {

/// @brief 抽象产品 - 定义了产品的规范，描述了产品的主要特性和功能
class AbstractProduct {
public:
    virtual ~AbstractProduct() {}
    virtual void Show() = 0;
};

/// @brief 具体产品 - 实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应
class ConcreteProduct : public AbstractProduct {
public:
    virtual void Show() {
        printf("ConcreteProduct Show ...\n");
    }
};

/// @brief 抽象工厂 - 提供了创建产品的接口
class AbstractFactory {
public:
    virtual ~AbstractFactory() {}
    virtual AbstractProduct* CreateProduct() = 0;
};

/// @brief 具体工厂 - 实现抽象工厂中的抽象方法，完成具体产品的创建
class ConcreteFactory : public AbstractFactory {
public:
    virtual AbstractProduct* CreateProduct() {
        AbstractProduct* ptr = new ConcreteProduct();
        return ptr;
    }
};

} // namespace factory

void Test_03_factory_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    factory::AbstractFactory* ptr_factory = new factory::ConcreteFactory();
    factory::AbstractProduct* ptr_product = ptr_factory->CreateProduct();
    ptr_product->Show();
    delete ptr_factory;
    delete ptr_product;
}

// -------------------- Test_03_factory_impl_1 --------------------
// ConcreteProduct Show ...
```

创建型 - 抽象工厂

- 含义

  `AbstractFactory` 为访问类提供一个创建一组相关或相互依赖对象的接口，且**访问类无须指定所要产品的具体类**就能得到同族的不同等级的产品的模式结构。

  抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

  - 优点
    - 可以在类的内部对产品族中相关联的**多等级产品共同管理**，而不必专门引入多个新的类来进行管理。
    - 当**增加一个新的产品族时不需要修改原代码，满足开闭原则**。
  - 缺点

    当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

- 示例

```
#pragma once

#include <stdio.h>

namespace abstract_factory {

/// @brief 抽象产品 - 定义了产品的规范，描述了产品的主要特性和功能
class AbstractProductA {
public:
    virtual ~AbstractProductA() {}
    virtual void Show() = 0;
};
class AbstractProductB {
public:
    virtual ~AbstractProductB() {}
    virtual void Show() = 0;
};

/// @brief 具体产品 - 实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应
class ConcreteProductA1 : public AbstractProductA {
public:
    virtual void Show() {
        printf("ConcreteProductA1 Show ...\n");
    }
};
class ConcreteProductA2 : public AbstractProductA {
public:
    virtual void Show() {
        printf("ConcreteProductA2 Show ...\n");
    }
};
class ConcreteProductB1 : public AbstractProductB {
public:
    virtual void Show() {
        printf("ConcreteProductB1 Show ...\n");
    }
};
class ConcreteProductB2 : public AbstractProductB {
public:
    virtual void Show() {
        printf("ConcreteProductB2 Show ...\n");
    }
};

/// @brief 抽象工厂 - 提供了创建产品的接口
class AbstractFactory {
public:
    virtual ~AbstractFactory() {}
    virtual AbstractProductA* CreateProductA() = 0;
    virtual AbstractProductB* CreateProductB() = 0;
};

/// @brief 具体工厂 - 实现抽象工厂中的抽象方法，完成具体产品的创建
class ConcreteFactory1 : public AbstractFactory {
public:
    virtual AbstractProductA* CreateProductA() {
        AbstractProductA* ptr = new ConcreteProductA1();
        return ptr;
    }
```

```
    virtual AbstractProductB* CreateProductB() {
        AbstractProductB* ptr = new ConcreteProductB1();
        return ptr;
    }
};
class ConcreteFactory2 : public AbstractFactory {
public:
    virtual AbstractProductA* CreateProductA() {
        AbstractProductA* ptr = new ConcreteProductA2();
        return ptr;
    }
    virtual AbstractProductB* CreateProductB() {
        AbstractProductB* ptr = new ConcreteProductB2();
        return ptr;
    }
};

} // namespace abstract_factory

void Test_04_abstract_factory_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    abstract_factory::AbstractFactory* ptr_factory_1 = new
abstract_factory::ConcreteFactory1();
    abstract_factory::AbstractProductA* ptr_product_A1 = ptr_factory_1-
>CreateProductA();
    ptr_product_A1->Show();
    delete ptr_factory_1;
    delete ptr_product_A1;
}

// -------------------- Test_04_abstract_factory_impl_1 --------------------
// ConcreteProductA1 Show ...
```

创建型 - 建造者

- 含义

  `Builder` 将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

  - 优点
    - **各个具体的建造者相互独立**，有利于系统的扩展。
    - **客户端不必知道产品内部组成的细节**，便于控制细节风险。
  - 缺点
    - **产品的组成部分必须相同**，这限制了其使用范围。
    - 如果产品的内部变化复杂，该模式会增加很多的建造者类。

  建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程。

- 应用场景

  - **创建的对象较复杂**，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。
  - **创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式**，即产品的构建过程和最终的表示是独立的。

- 示例

```
#pragma once

#include <stdio.h>
#include <string>

namespace builder {

/// @brief 产品 - 包含多个组成部件的复杂对象。
class Product {
public:
    void Show() {
        printf("Show Product ...\n");
    }

private:
    std::string partA_;
    std::string partB_;
    std::string partC_;
};

/// @brief 抽象建造者 - 包含创建产品各个子部件的抽象方法
class AbstractBuilder {
public:
    AbstractBuilder() {
        product_.reset(new Product());
    }
    Product* GetResult() {
        return product_.get();
    }

    virtual void BuilderPartA() = 0;
    virtual void BuilderPartB() = 0;
    virtual void BuilderPartC() = 0;

protected:
    std::shared_ptr<Product> product_;
};

/// @brief 具体建造者 - 实现了抽象建造者接口
class ConcreteBuilder : public AbstractBuilder {
public:
    virtual void BuilderPartA() {
        printf("Builder PartA...\n");
    }
    virtual void BuilderPartB() {
        printf("Builder PartB...\n");
    }
    virtual void BuilderPartC() {
        printf("Builder PartC...\n");
    }
};

/// @brief 指挥者 - 调用建造者中的方法完成复杂对象的创建
class Director {
public:
    Director(AbstractBuilder* builder) : builder_(builder) {}
    Product* ConstructProduct() {
        builder_->BuilderPartA();
        builder_->BuilderPartB();
        builder_->BuilderPartC();
```

```
        return builder_->GetResult();
    }

private:
    AbstractBuilder* builder_;
};

} // namespace builder

void Test_05_builder_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<builder::AbstractBuilder> builder(new
builder::ConcreteBuilder());
    std::shared_ptr<builder::Director> director(new
builder::Director(builder.get()));

    builder::Product* ptr = director->ConstructProduct();
    ptr->Show();
}

// -------------------- Test_05_builder_impl_1 --------------------
// Builder PartA...
// Builder PartB...
// Builder PartC...
// Show Product ...
```

结构型 - 代理

- 含义

  `Proxy` 为某对象提供一种代理以控制对该对象的访问。即**控制对其它对象的访问。**

  - 优点
    - 在客户端与目标对象之间**起到一个中介作用和保护目标对象的作用**；
    - **可以扩展目标对象的功能**；
    - 能将客户端与目标对象分离，在一定程度上**降低了系统的耦合度**；
  - 缺点
    - 在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
    - 增加了系统的复杂度；

- 应用场景
    - 在有些情况下，一个客户不能或者不想直接访问另一个对象，这时需要找一个中介帮忙完成某项任务，这个中介就是代理对象。例如，购买火车票不一定要去火车站买，可以通过 12306 网站或者去火车票代售点买。又如找女朋友、找保姆、找工作等都可以通过找中介完成。
    - 在软件设计中，使用代理模式的例子也很多，例如，要访问的远程对象比较大（如视频或大图像等），其下载要花很多时间。还有因为安全原因需要屏蔽客户端直接访问真实对象，如某单位的内部数据库等。

- 示例

```cpp
#pragma once

#include <stdio.h>

namespace proxy {

/// @brief 抽象主题类, 用于声明真实主题和代理对象实现的业务方法
class AbstractSubject {
public:
    virtual ~AbstractSubject() {}
    virtual void Request() = 0;
};

/// @breif 真实主题类, 实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象
class RealSubject : public AbstractSubject {
public:
    virtual void Request() {
        printf("this is RealSubject::Request!\n");
    }
};

// @breif 代理类, 提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能
class SubjectProxy : public AbstractSubject {
public:
    virtual ~SubjectProxy() {
        if (real_) delete real_;
    }
    virtual void Request() {
        if (!real_) real_ = new RealSubject();
        PreRequest();
        real_->Request();
        PostRequest();
    }
    virtual void PreRequest() {
        printf("this is SubjectProxy::PreRequest!\n");
    }
    virtual void PostRequest() {
        printf("this is SubjectProxy::PostRequest!\n");
    }

private:
    RealSubject* real_;
};

} // namespace proxy

void Test_06_proxy_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    proxy::AbstractSubject* ptr = new proxy::SubjectProxy();
    ptr->Request();
    delete ptr;
}

// -------------------- Test_06_proxy_impl_1 --------------------
// this is SubjectProxy::PreRequest!
// this is RealSubject::Request!
// this is SubjectProxy::PostRequest!
```

结构型 - 适配器

- 含义

  `Adapter` **将一个类的接口转换成客户希望的另外一个接口**，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

  - 优点
    - 客户端通过适配器可以透明地调用目标接口。
    - 复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类。
    - 将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。
  - 缺点

    对类适配器来说，更换适配器的实现过程比较复杂。
- 应用场景

  - 在现实生活中，**经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配**。例如，讲中文的人同讲英文的人对话时需要一个翻译，用直流电的笔记本电脑接交流电源时需要一个电源适配器，用计算机访问照相机的 SD 内存卡时需要一个读卡器等。
  - 在软件设计中也可能出现：**需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高**，这时用适配器模式能很好地解决这些问题。

- 示例

```
#pragma once

#include <stdio.h>

namespace adapter {

/// @brief 目标接口, 即当前系统业务所期待的接口
class AbstractTarget {
public:
    virtual ~AbstractTarget() {}
    virtual void Request() = 0;
};

/// @brief 被适配者
class Adaptee {
public:
    virtual void SpecificRequest() {
        printf("this is Adaptee::SpecificRequest!\n");
    }
};

/// @brief 适配器, 把适被配者接口转换成目标接口，让客户按目标接口的格式访问被适配
者
class Adapter : public Adaptee, public AbstractTarget {
public:
    virtual void Request() {
        SpecificRequest();
    }
};

} // namespace adapter

void Test_07_adapter_impl_1() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    adapter::AbstractTarget* ptr = new adapter::Adapter();
    ptr->Request();
    delete ptr;
}

// -------------------- Test_07_adapter_impl_1 --------------------
// this is Adaptee::SpecificRequest!
```

结构型 - 桥接

- 含义

  `Bridge` **将抽象与实现分离，使它们可以独立变化**。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

  - 优点
    - 由于抽象与实现分离，所以**扩展能力强**；
    - 其**实现细节对客户透明**。
  - 缺点

    要求开发者针对抽象化进行设计与编程
- 应用场景

  - 在现实生活中，**某些类具有两个或多个维度的变化**，如图形既可按形状分，又可按颜色分。如何设计类似于 Photoshop 这样的软件，能画不同形状和不同颜色的图形呢？如果用继承方式，m 种形状和 n 种颜色的图形就有 m×n 种，不但对应的子类很多，而且扩展困难。
  - 这样的例子还有很多，如不同颜色和字体的文字、不同品牌和功率的汽车、不同性别和职业的男女、支持不同平台和不同文件格式的媒体播放器等。

- 示例

- **简单的抽象&实现分离**

```
  #pragma once

  #include <stdio.h>

  namespace bridge {

  /// @brief 抽象类接口
  class AbstractInterface {
  public:
      AbstractInterface();
      ~AbstractInterface();

      void Request();

  private:
      class Impl;
      Impl* impl_;
  };

  /// @brief 实现类接口
  class AbstractInterface::Impl {
  public:
      void Request() {
          printf("this is AbstractInterface::Impl::Request!\n");
      }
  };

  /// 抽象类接口的实现 - 调用实现类接口
  AbstractInterface::AbstractInterface() {
      impl_ = new Impl();
  }

  AbstractInterface::~AbstractInterface() {
      delete impl_;
  }

  void AbstractInterface::Request() {
      impl_->Request();
  }

  } // namespace bridge

  void Test_08_bridge_impl_1() {
      printf("-------------------- %s --------------------\n",
__FUNCTION__);
      bridge::AbstractInterface* ptr = new bridge::AbstractInterface();
    ptr->Request();
      delete ptr;
}

  // -------------------- Test_08_bridge_impl_1 --------------------
  // this is AbstractInterface::Impl::Request!
```

- **复制的抽象&实现分离**

```
#pragma once

#include <stdio.h>

namespace bridge {

/// @brief 实现化角色, 定义实现化角色的接口
class AbstractImplementor {
public:
    virtual ~AbstractImplementor() {}
    virtual void Request() = 0;
};

/// @brief 具体实现化角色, 给出实现化角色接口的具体实现
class ConcreteImplementorA : public AbstractImplementor {
public:
    virtual void Request() {
        printf("this ConcreteImplementorA::Request!\n");
    }
};
class ConcreteImplementorB : public AbstractImplementor {
public:
    virtual void Request() {
        printf("this ConcreteImplementorB::Request!\n");
    }
};

/// @brief 抽象类角色, 对实现化对象的引用
class AbstractRole {
public:
    AbstractRole(AbstractImplementor* impl) : impl_(impl) {}
    virtual ~AbstractRole() {}

    virtual void Request() = 0;

protected:
    AbstractImplementor* impl_;
};

/// @brief 扩展抽象化角色, 实现父类中的业务方法
class ExtendRole : public AbstractRole {
public:
    ExtendRole(AbstractImplementor* impl) : AbstractRole(impl) {}

    virtual void Request() {
        impl_->Request();
    }
};

} // namespace bridge

void Test_08_bridge_impl_2() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    bridge::AbstractImplementor* ptr_impl = new
bridge::ConcreteImplementorA();
    bridge::AbstractRole* ptr_role = new bridge::ExtendRole(ptr_impl);
    ptr_role->Request();
    delete ptr_role;
    delete ptr_impl;
```

```
}

// -------------------- Test_08_bridge_impl_2 --------------------
// this ConcreteImplementorA::Request!
```

结构型 - 装饰

- 含义

  `Decorator` **动态的给对象增加一些职责**，即增加其额外的功能。

  通常情况下，扩展一个类的功能会使用**继承**方式来实现。但**继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀**。如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。

  - 优点
    - 采用装饰模式扩展对象的功能**比采用继承方式更加灵活**。
    - 可以设计出多个不同的具体装饰类，**创造出多个不同行为的组合**。
  - 缺点

    增加了许多子类，如果过度使用会使程序变得很复杂。
- 应用场景

  - 在现实生活中，常常需要**对现有产品增加新的功能或美化其外观**，如房子装修、相片加相框等。
  - 在软件开发过程中，有时想用一些现存的组件。这些组件可能只是完成了一些核心功能。但**在不改变其结构的情况下，可以动态地扩展其功能**。所有这些都可以采用装饰模式来实现。

- 示例

```
#pragma once

#include <stdio.h>

namespace decorator {

/// @brief 抽象组件 - 被装饰的接口基类
class AbstractComponent {
public:
    virtual ~AbstractComponent() {}
    virtual void Request() = 0;
};

/// @brief 具体组件 - 被装饰的接口派生类
class ConcreteComponent : public AbstractComponent {
public:
    virtual void Request() {
        printf("this is ConcreteComponent::Request!\n");
    }
};

/// @brief 抽象装饰
class AbstractDecorator : public AbstractComponent {
public:
    AbstractDecorator(AbstractComponent* comp) : comp_(comp) {}

    virtual void Request() {
        if (comp_) comp_->Request();
    }

private:
    AbstractComponent* comp_;
};

/// @brief 具体装饰
class ConcreteDecoratorA : public AbstractDecorator {
public:
    ConcreteDecoratorA(AbstractComponent* comp) : AbstractDecorator(comp) {}

    virtual void Request() {
        PreRequest();
        AbstractDecorator::Request();
        PostRequest();
    }

protected:
    void PreRequest() {
        printf("ConcreteDecoratorA::PreRequest!\n");
    }
    void PostRequest() {
        printf("ConcreteDecoratorA::PostRequest!\n");
    }
};

} // namespace decorator

void Test_09_decorator_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    decorator::AbstractComponent* ptr_comp = new
decorator::ConcreteComponent();
```

```
    ptr_comp->Request();    // 原有组件该有的功能不变

    // 对现有组件的功能进行扩展
    decorator::AbstractDecorator* ptr_decorator = new
decorator::ConcreteDecoratorA(ptr_comp);
    ptr_decorator->Request();

    delete ptr_decorator;
    delete ptr_comp;
}

// -------------------- Test_09_decorator_impl_1 --------------------
// this is ConcreteComponent::Request!
// ConcreteDecoratorA::PreRequest!
// this is ConcreteComponent::Request!
// ConcreteDecoratorA::PostRequest!
```

结构型 - 外观

- 含义

  `Facade` 为多个复杂的子系统**提供一个一致的接口**，使这些子系统更加容易被访问。

  - 优点
    - 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。
    - 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。
    - 降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编译一个子系统不会影响其他的子系统，也不会影响外观对象。
  - 缺点
    - 不能很好地限制客户使用子系统类。
    - **增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”**。
- 应用场景

  - 在现实生活中，常常存在办事较复杂的例子，如办房产证或注册一家公司，有时要同多个部门联系，这时要是有一个综合部门能解决一切手续问题就好了。
  - 软件设计也是这样，当一个系统的功能越来越强，子系统会越来越多，客户对系统的访问也变得越来越复杂。这时如果系统内部发生改变，客户端也要跟着改变，这**违背了“开闭原则”，也违背了“迪米特法则”**，

  所以有必要**为多个子系统提供一个统一的接口，从而降低系统的耦合度**，这就是外观模式的目标。

- 示例

```
#pragma once

#include <iostream>

namespace facade {

/// @brief 具体实现子模块
class SubModuleImplementorA {
public:
    void PreRequest () {
        printf("SubModuleImplementorA::PreRequest!\n");
    }
};
class SubModuleImplementorB {
public:
    void RealRequest () {
        printf("SubModuleImplementorB::RealRequest!\n");
    }
};
class SubModuleImplementorC {
public:
    void PostRequest () {
        printf("SubModuleImplementorC::PostRequest!\n");
    }
};

/// @brief 外观角色
class Facade {
public:
    Facade() {
        implA_.reset(new SubModuleImplementorA());
        implB_.reset(new SubModuleImplementorB());
        implC_.reset(new SubModuleImplementorC());
    }

    void Request() {
        implA_->PreRequest();
        implB_->RealRequest();
        implC_->PostRequest();
    }

private:
    std::shared_ptr<SubModuleImplementorA> implA_;
    std::shared_ptr<SubModuleImplementorB> implB_;
    std::shared_ptr<SubModuleImplementorC> implC_;
};

} // namespace facade

void Test_10_facade_impl_1() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    std::shared_ptr<facade::Facade> ptr(new facade::Facade());
    ptr->Request();
}

// -------------------- Test_10_facade_impl_1 --------------------
// SubModuleImplementorA::PreRequest!
// SubModuleImplementorB::RealRequest!
// SubModuleImplementorC::PostRequest!
```

结构型 - 享元

- 含义

  `Flyweight` 运用共享技术来有效地支持大量细粒度对象的复用。

  - 优点

    相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。
  - 缺点
    - **为了使对象可以共享，需要将一些不能共享的状态外部化**，这将增加程序的复杂性。
    - 读取享元模式的外部状态会**使得运行时间稍微变长**。
- 应用场景

  - 在面向对象程序设计过程中，**有时会面临要创建大量相同或相似对象实例的问题**。创建那么多的对象将会耗费很多的系统资源，它是系统性能提高的一个瓶颈。
  - 例如，围棋和五子棋中的黑白棋子，图像中的坐标点或颜色，局域网中的路由器、交换机和集线器，教室里的桌子和凳子等。
  - 这些对象有很多相似的地方，**如果能把它们相同的部分提取出来共享，则能节省大量的系统资源**，这就是享元模式的产生背景。

- 示例

```
#pragma once

#include <iostream>
#include <map>
#include <string>

namespace flyweight {

/// @brief 非享元类, 以参数的形式注入具体享元的相关方法中
class UnsharableConcreteFlyweight {
public:
    UnsharableConcreteFlyweight(const std::string& content) :
content_(content) {}
    const std::string& GetContent() const { return content_; }

private:
    std::string content_;
};

/// @brief 抽象享元类, 为具体享元规范需要实现的公共接口
class AbstractFlyweight {
public:
    virtual ~AbstractFlyweight() {}
    virtual void Run(UnsharableConcreteFlyweight* unsharable_content_ptr) =
0;
};

/// @brief 具体享元类, 实现抽象享元角色中所规定的接口
class ConcreteFlyweightA : public AbstractFlyweight {
public:
    virtual void Run(UnsharableConcreteFlyweight* unsharable_content_ptr) {
        std::cout << "ConcreteFlyweightA::Run!" << std::endl;
        std::cout << unsharable_content_ptr->GetContent() << std::endl;
    }
};
class ConcreteFlyweightB : public AbstractFlyweight {
public:
    virtual void Run(UnsharableConcreteFlyweight* unsharable_content_ptr) {
        std::cout << "ConcreteFlyweightB::Run!" << std::endl;
        std::cout << unsharable_content_ptr->GetContent() << std::endl;
    }
};

/// @brief 享元工厂类
class FlyweightFactory {
public:
    enum FlyweightType {
        ConcreteA = 0,
        ConcreteB,
    };

    ~FlyweightFactory() {
        for (auto it = fly_weights_.begin(); it != fly_weights_.end(); it++)
{
            if (it->second) delete it->second;
        }
        fly_weights_.clear();
    }

    AbstractFlyweight* GetFlyweight(const std::string& key, const
```

```
FlyweightType& type = ConcreteA) {
        auto it = fly_weights_.find(key);
        if (fly_weights_.end() == it) {
            AbstractFlyweight* ptr = nullptr;
            if (ConcreteA == type) {
                ptr = new ConcreteFlyweightA();
            }
            else if (ConcreteB == type) {
                ptr = new ConcreteFlyweightB();
            }
            fly_weights_[key] = ptr;
        }
        return fly_weights_[key];
    }

private:
    std::map<std::string, AbstractFlyweight*> fly_weights_;
};

} // namespace flyweight

void Test_11_flyweight_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<flyweight::UnsharableConcreteFlyweight>
unsharable_content_ptr_1(new
flyweight::UnsharableConcreteFlyweight("unsharable_content_ptr_1"));
    std::shared_ptr<flyweight::UnsharableConcreteFlyweight>
unsharable_content_ptr_2(new
flyweight::UnsharableConcreteFlyweight("unsharable_content_ptr_2"));

    std::shared_ptr<flyweight::FlyweightFactory> ptr(new
flyweight::FlyweightFactory());
    flyweight::AbstractFlyweight* flyweight_ptr_A_red = ptr-
>GetFlyweight("Red");
    flyweight::AbstractFlyweight* flyweight_ptr_A_point = ptr-
>GetFlyweight("Point");

    flyweight_ptr_A_red->Run(unsharable_content_ptr_1.get());
    flyweight_ptr_A_point->Run(unsharable_content_ptr_2.get());
}

// -------------------- Test_11_flyweight_impl_1 --------------------
// ConcreteFlyweightA::Run!
// unsharable_content_ptr_1
// ConcreteFlyweightA::Run!
// unsharable_content_ptr_2
```

结构型 - 组合

- 含义

  `Composite` 将对象**组合成树状层次结构**，使用户**对单个对象和组合对象具有一致的访问性**。

  - 优点
    - **使得客户端代码可以一致地处理单个对象和组合对象**，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码；
    - 更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，**满足“开闭原则”**；
  - 缺点
    - 设计较复杂，客户端需要花更多时间理清类之间的层次关系；
    - 不容易限制容器中的构件；
    - 不容易用继承的方法来增加构件的新功能；
- 应用场景

  - 在现实生活中，存在很多“部分-整体”的关系，例如，大学中的部门与学院、总公司中的部门与分公司、学习用品中的书与书包、生活用品中的衣月艮与衣柜以及厨房中的锅碗瓢盆等。
  - 在软件开发中也是这样，例如，文件系统中的文件与文件夹、窗体程序中的简单控件与容器控件等。对这些简单对象与复合对象的处理，如果用组合模式来实现会很方便。

- 示例

  组合模式分为 `透明式` 的组合模式和 `安全式` 的组合模式。

- **透明方式**
    - 由于**抽象构件声明了所有子类中的全部方法**，所以客户端无须区别树叶对象和树枝对象，对客户端来说是透明的。
    - 但其缺点是：树叶构件本来没有 Add()、Remove() 及 GetChild() 方法，却要实现它们（空实现或抛异常），这样会带来一些安全性问题。

```
  #pragma once

  #include <iostream>
  #include <string>
  #include <vector>

  namespace composite {

  /// @brief 抽象构件, 为树叶构件和树枝构件声明公共接口，并实现它们的默认行为
  ///        - 透明式: 声明访问和管理子类的接口
  ///        - 安全式: 不声明访问和管理子类的接口，管理工作由树枝构件完成
  class AbstractTransparentComponent {
  public:
      virtual ~AbstractTransparentComponent() {}

      // 透明式下, 需声明所有子类中的全部方法
      virtual void Add(AbstractTransparentComponent* comp) = 0;
      virtual void Remove(AbstractTransparentComponent* comp) = 0;
      virtual AbstractTransparentComponent* GetChild(const int id) = 0;

      virtual void Operation() = 0;
  };

  /// @brief 树叶构件, 组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中声
明的公共接口
  class LeafTransparentComponent : public AbstractTransparentComponent
{
  public:
      LeafTransparentComponent(const std::string& name) : name_(name)
{}

      // 透明式下, 树叶构件需要实现的不必要接口 (可空实现或跑出异常)
      virtual void Add(AbstractTransparentComponent* comp) {
          printf("Error! Leaf can't realize Add!\n");
      }
      virtual void Remove(AbstractTransparentComponent* comp) {
          printf("Error! Leaf can't realize Remove!\n");
      }
      virtual AbstractTransparentComponent* GetChild(const int id) {
          printf("Error! Leaf can't realize GetChild!\n");
          return nullptr;
      }

      // 树叶构件切实需要实现的接口
      virtual void Operation() {
          printf("this(%s) is LeafTransparentComponent::Operation!\n",
name_.c_str());
      }

  private:
      std::string name_;
  };

  /// @brief 树枝构件, 组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声
明的接口，主要作用是存储和管理子部件
  class CompositeTransparentComponent : public
AbstractTransparentComponent {
  public:
      virtual void Add(AbstractTransparentComponent* comp) {
          children_.emplace_back(comp);
```

```
      }
      virtual void Remove(AbstractTransparentComponent* comp) {
          for (auto it = children_.begin(); it != children_.end();
it++) {
              if (comp == (*it)) {
                  children_.erase(it++);
                  break;
              }
          }
      }
      virtual AbstractTransparentComponent* GetChild(const int id) {
          if (id >= 0 && id < children_.size()) return children_[id];
          return nullptr;
      }
      virtual void Operation() {
          for (auto* child : children_) {
              child->Operation();
          }
      }

  private:
      std::vector<AbstractTransparentComponent*> children_;
  };

  } // namespace composite

  void Test_12_composite_impl_1() {
      printf("-------------------- %s --------------------\n",
__FUNCTION__);
      std::shared_ptr<composite::AbstractTransparentComponent>
composite_ptr(new composite::CompositeTransparentComponent());

      std::shared_ptr<composite::AbstractTransparentComponent>
leaf_ptr_A(new composite::LeafTransparentComponent("leaf A"));
      std::shared_ptr<composite::AbstractTransparentComponent>
leaf_ptr_B(new composite::LeafTransparentComponent("leaf B"));
      std::shared_ptr<composite::AbstractTransparentComponent>
leaf_ptr_C(new composite::LeafTransparentComponent("leaf C"));

      composite_ptr->Add(leaf_ptr_A.get());
      composite_ptr->Add(leaf_ptr_B.get());
      composite_ptr->Add(leaf_ptr_C.get());

      composite::AbstractTransparentComponent* tmp_ptr = composite_ptr-
>GetChild(1);
      tmp_ptr->Operation();
      composite_ptr->Remove(leaf_ptr_B.get());

      composite_ptr->Operation();
}

  // -------------------- Test_12_composite_impl_1 --------------------
// this(leaf B) is LeafTransparentComponent::Operation!
  // this(leaf A) is LeafTransparentComponent::Operation!
  // this(leaf C) is LeafTransparentComponent::Operation!
```

- **安全方式**
    - 将管理子构件的方法移到树枝构件中，抽象构件和树叶构件没有对子对象的管理方法，这样就避免了上一种方式的安全性问题，
    - 但由于叶子和分支有不同的接口，**客户端在调用时要知道树叶对象和树枝对象的存在**，所以失去了透明性。

```
#pragma once

#include <iostream>
#include <string>
#include <vector>

namespace composite {

/// @brief 抽象构件, 为树叶构件和树枝构件声明公共接口，并实现它们的默认行为
///        - 透明式: 声明访问和管理子类的接口
///        - 安全式: 不声明访问和管理子类的接口，管理工作由树枝构件完成
class AbstractSafeComponent {
public:
    virtual ~AbstractSafeComponent() {}

    virtual void Operation() = 0;
};

/// @brief 树叶构件, 组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中声明
的公共接口
class LeafSafeComponent : public AbstractSafeComponent {
public:
    LeafSafeComponent(const std::string& name) : name_(name) {}

    // 树叶构件切实需要实现的接口
    virtual void Operation() {
        printf("this(%s) is LeafSafeComponent::Operation!\n",
name_.c_str());
    }

private:
    std::string name_;
};

/// @brief 树枝构件, 组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明
的接口，主要作用是存储和管理子部件
class CompositeSafeComponent : public AbstractSafeComponent {
public:
    // 安全式下, 由树枝构件自己实现对树叶构件的管理
    virtual void Add(AbstractSafeComponent* comp) {
        children_.emplace_back(comp);
    }
    virtual void Remove(AbstractSafeComponent* comp) {
        for (auto it = children_.begin(); it != children_.end(); it++)
{
            if (comp == (*it)) {
                children_.erase(it++);
                break;
            }
        }
    }
    virtual AbstractSafeComponent* GetChild(const int id) {
        if (id >= 0 && id < children_.size()) return children_[id];
        return nullptr;
    }
    virtual void Operation() {
        for (auto* child : children_) {
            child->Operation();
        }
    }
```

```
private:
    std::vector<AbstractSafeComponent*> children_;
};

} // namespace composite

void Test_12_composite_impl_2() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    composite::AbstractSafeComponent* composite_ptr = new
composite::CompositeSafeComponent();

    composite::AbstractSafeComponent* leaf_ptr_A = new
composite::LeafSafeComponent("leaf A");
    composite::AbstractSafeComponent* leaf_ptr_B = new
composite::LeafSafeComponent("leaf B");
    composite::AbstractSafeComponent* leaf_ptr_C = new
composite::LeafSafeComponent("leaf C");

    // 对树叶构件的操作
    {
        composite::CompositeSafeComponent* composite_son_ptr =
dynamic_cast<composite::CompositeSafeComponent*>(composite_ptr);

        composite_son_ptr->Add(leaf_ptr_A);
        composite_son_ptr->Add(leaf_ptr_B);
        composite_son_ptr->Add(leaf_ptr_C);

        composite::AbstractSafeComponent* tmp_ptr = composite_son_ptr-
>GetChild(1);
        tmp_ptr->Operation();
        composite_son_ptr->Remove(leaf_ptr_B);
    }

    composite_ptr->Operation();

    delete composite_ptr;
    delete leaf_ptr_A;
    delete leaf_ptr_B;
    delete leaf_ptr_C;
}

// -------------------- Test_12_composite_impl_2 --------------------
// this(leaf B) is LeafSafeComponent::Operation!
// this(leaf A) is LeafSafeComponent::Operation!
// this(leaf C) is LeafSafeComponent::Operation!
```

行为型 - 模板方法

- 含义

  `TemplateMethod` **定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中**，使得**子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤**。

  - 优点
    - 它**封装了不变部分，扩展可变部分**。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
    - 它在父类中提取了公共的部分代码，便于代码复用。
    - 部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，**符合开闭原则**。
  - 缺点
    - 对每个不同的实现都需要定义一个子类，这会导致**类的个数增加**，系统更加庞大，设计也更加抽象。
    - 父类中的抽象方法由子类实现，**子类执行的结果会影响父类的结果**，这导致一种反向的控制结构，它提高了代码阅读的难度。
- 应用场景

  - 在面向对象程序设计过程中，程序员常常会遇到这种情况：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。
  - 生活中的例子: **一个人每天会起床、吃饭、做事、睡觉等，其中“做事”的内容每天可能不同**。
  - 我们把这些规定了流程或格式的实例定义成模板，允许使用者根据自己的需求去更新它，例如，简历模板、论文模板、Word 中模板文件等。
- 实现方式

  抽象类负责给出一个算法的轮廓和骨架, 即定义两种类型的方法

  - **模板方法**: 定义了算法的骨架，按某种顺序调用其包含的基本方法
  - **基本方法**:
    - **抽象方法**：在抽象类中申明，由具体子类实现
    - **具体方法**：在抽象类中已经实现，在具体子类中可以继承或重写它
    - **钩子方法**：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种

- 示例

```
#pragma once

#include <iostream>

namespace template_method {

/// @brief 抽象类, 给出一个算法的轮廓和骨架
class AbstractClass {
public:
    virtual ~AbstractClass() {}

    /// @brief 模板方法, 定义了算法的骨架，按某种顺序调用其包含的基本方法
    virtual void TemplateMethod() {
        SpecificMethodA();
        if (HookMethodB()) {
            HookMethodA();
        }
        AbstractMethod();
        SpecificMethodB();
    }

protected:
    /// @brief 基本方法, 可以有三种
    ///             - 抽象方法：在抽象类中申明，由具体子类实现
    ///             - 具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它
    ///             - 钩子方法：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重
写的空方法两种
    virtual void AbstractMethod() = 0;
    virtual void SpecificMethodA() {
        printf("this is AbstractClass::SpecificMethodA!\n");
    }
    virtual void SpecificMethodB() {
        printf("this is AbstractClass::SpecificMethodB!\n");
    }
    virtual void HookMethodA() {    // 通常为空实现
        printf("this is AbstractClass::HookMethodA!\n");
    }
    virtual bool HookMethodB() { return false; }
};

/// @brief 具体类, 实现抽象类中所定义的抽象方法和钩子方法
class ConcreteClass : public AbstractClass {
public:
    virtual void AbstractMethod() {
        printf("this is ConcreteClass::AbstractMethod!\n");
    }

protected:
    virtual void SpecificMethodB() {
        printf("this is ConcreteClass::SpecificMethodB!\n");
    }
    virtual void HookMethodA() {    // 通常为空实现
        printf("this is ConcreteClass::HookMethodA!\n");
    }
    virtual bool HookMethodB() { return true; }
};

} // namespace template_method

void Test_13_composite_impl_1() {
```

```
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    template_method::AbstractClass* ptr = new
template_method::ConcreteClass();
    ptr->TemplateMethod();

    delete ptr;
}

// -------------------- Test_13_template_method_impl_1 --------------------
// this is AbstractClass::SpecificMethodA!
// this is ConcreteClass::HookMethodA!
// this is ConcreteClass::AbstractMethod!
// this is ConcreteClass::SpecificMethodB!
```

行为型 - 策略

- 含义

  `Strategy` 定义了一系列算法，并**将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户**。

  - 优点
    - 多重条件语句不易维护，而使用策略模式可以**避免使用多重条件语句**。
    - 策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而**避免重复的代码**。
    - 策略模式**可以提供相同行为的不同实现**，客户可以根据不同时间或空间要求选择不同的。
    - 策略模式提供了**对开闭原则的完美支持**，可以在不修改原代码的情况下，灵活增加新算法。
    - 策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。
  - 缺点
    - **客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类**。
    - 策略模式造成很多的策略类。
- 应用场景

  - 在现实生活中常常遇到**实现某种目标存在多种策略可供选择的情况**，例如，出行旅游可以乘坐飞机、乘坐火车、骑自行车或自己开私家车等，超市促销可以采用打折、送商品、送积分等方法。
  - 在软件开发中也常常遇到类似的情况，当**实现某一个功能存在多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能**，如数据排序策略有冒泡排序、选择排序、插入排序、二叉树排序等。

- 示例

```
#pragma once

#include <stdio.h>

namespace strategy {

/// @brief 抽象策略类, 定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法
class AbstractStrategy {
public:
    virtual ~AbstractStrategy() {}
    virtual void StrategyMethod() = 0;
};

/// @brief 具体策略类, 实现了抽象策略定义的接口，提供具体的算法实现
class ConcreteStrategyA : public AbstractStrategy {
public:
    virtual void StrategyMethod() {
        printf("this is ConcreteStrategyA::StrategyMethod!\n");
    }
};
class ConcreteStrategyB : public AbstractStrategy {
public:
    virtual void StrategyMethod() {
        printf("this is ConcreteStrategyB::StrategyMethod!\n");
    }
};

/// @brief 环境类, 持有一个策略类的引用，最终给客户端调用
class Context {
public:
    void SetStrategy(AbstractStrategy* strategy) {
        strategy_ = strategy;
    }
    void StrategyMethod() {
        if (strategy_) strategy_->StrategyMethod();
    }

private:
    AbstractStrategy* strategy_ = nullptr;
};

} // namespace strategy

void Test_14_strategy_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    strategy::AbstractStrategy* ptr_strategy_A = new
strategy::ConcreteStrategyA();
    strategy::AbstractStrategy* ptr_strategy_B = new
strategy::ConcreteStrategyB();

    strategy::Context env_;
    env_.SetStrategy(ptr_strategy_A);
    env_.StrategyMethod();

    env_.SetStrategy(ptr_strategy_B);
    env_.StrategyMethod();

    delete ptr_strategy_A;
    delete ptr_strategy_B;
```

```
}

// -------------------- Test_14_strategy_impl_1 --------------------
// this is ConcreteStrategyA::StrategyMethod!
// this is ConcreteStrategyB::StrategyMethod!
```

行为型 - 命令

- 含义

  `Command` **将一个请求封装为一个对象**，使发出请求的责任和执行请求的责任分割开。

  - 优点
    - **降低系统的耦合度**。命令模式能将调用操作的对象与实现该操作的对象解耦。
    - **增加或删除命令非常方便**。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活。
    - 可以实现宏命令。命令模式**可以与** `组合模式` **结合，将多个命令装配成一个组合命令，即宏命令**。
    - **方便实现 Undo 和 Redo 操作**。命令模式可以与 `备忘录模式` 结合，实现命令的撤销与恢复。
  - 缺点

    可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。
- 应用场景

  - 在软件开发系统中，**常常出现“方法的请求者”与“方法的实现者”之间存在紧密的耦合关系**。这不利于软件功能的扩展与维护。例如，想对行为进行“撤销、重做、记录”等处理都很不方便。
  - 在现实生活中，这样的例子也很多，例如，电视机遥控器（命令发送者）通过按钮（具体命令）来遥控电视机（命令接收者），还有计算机键盘上的“功能键”等。

- 示例

- 简单调用

```
  #pragma once

  #include <iostream>

  namespace command {

  /// @brief 抽象命令类, 声明执行命令的接口
  class AbstractCommand {
  public:
      virtual ~AbstractCommand() {}
      virtual void Execute() = 0;
  };

  /// @brief 命令接收者, 执行命令功能的相关操作，是具体命令对象业务的真正实现者
  class Receiver {
  public:
      void Action() {
          printf("this is Receiver::Action!\n");
      }
  };

  /// @brief 具体命令类, 它拥有接收者对象，并通过调用接收者的功能来完成命令要执行
的操作
  class ConcreteCommand : public AbstractCommand {
  public:
      ConcreteCommand() {
          receiver_.reset(new Receiver());
      }
      virtual void Execute() {
          receiver_->Action();
      }

  private:
      std::shared_ptr<Receiver> receiver_;
  };

  /// @brief 命令请求者, 请求的发送者，它通常拥有很多的命令对象，并通过访问命令对
象来执行相关请求，它不直接访问接收者
  class Invoker{
  public:
      Invoker(AbstractCommand* cmd) : cmd_(cmd) {}
      void SetCommand(AbstractCommand* cmd) {
          cmd_ = cmd;
      }
      void Request() {
          cmd_->Execute();
      }

  private:
      AbstractCommand* cmd_;
  };

  } // namespace command

  void Test_15_command_impl_1() {
      printf("-------------------- %s --------------------\n",
__FUNCTION__);
      std::shared_ptr<command::AbstractCommand> cmd(new
command::ConcreteCommand());
```

```
    command::Invoker client(cmd.get());
      client.Request();
}

  // -------------------- Test_15_command_impl_1 --------------------
  // this is Receiver::Action!
```

- 与 `组合模式` 结合, 实现**宏命令**

```
#pragma once

#include <iostream>
#include <vector>

namespace command {

/// 命令模式 + 组合模式 = 宏命令
/// 宏命令包含了一组命令，它充当了具体命令与调用者的双重角色，执行它时将递归调用它所包含的所有命令

/// @brief 抽象命令类, 声明执行命令的接口
class AbstractMacroCommand {
public:
    virtual ~AbstractMacroCommand() {}
    virtual void Execute() = 0;
};

/// @brief 命令接收者, 执行命令功能的相关操作，是具体命令对象业务的真正实现者
class CompositeReceiver {
public:
    void ActionA() {
        printf("this is CompositeReceiver::ActionA!\n");
    }
    void ActionB() {
        printf("this is CompositeReceiver::ActionB!\n");
    }
};

/// @brief 具体命令类, 它拥有接收者对象，并通过调用接收者的功能来完成命令要执行的操作
class ConcreteCommandA : public AbstractMacroCommand {
public:
    ConcreteCommandA() {
        receiver_.reset(new CompositeReceiver());
    }
    virtual void Execute() {
        receiver_->ActionA();
    }

private:
    std::shared_ptr<CompositeReceiver> receiver_;
};
class ConcreteCommandB : public AbstractMacroCommand {
public:
    ConcreteCommandB() {
        receiver_.reset(new CompositeReceiver());
    }
    virtual void Execute() {
        receiver_->ActionB();
    }

private:
    std::shared_ptr<CompositeReceiver> receiver_;
};

/// @brief 命令请求者, 请求的发送者，它通常拥有很多的命令对象，并通过访问命令对象来执行相关请求，它不直接访问接收者
class CompositeInvoker{
public:
```

```
    void AddCommand(AbstractMacroCommand* cmd) {
        cmds_.emplace_back(cmd);
    }
    void RemoveCommand(AbstractMacroCommand* cmd) {
        for (auto it = cmds_.begin(); it != cmds_.end(); it++) {
            if (cmd == (*it)) {
                cmds_.erase(it++);
                break;
            }
        }
    }
    void Request() {
        for (auto* cmd : cmds_) {
            cmd->Execute();
        }
    }

private:
    std::vector<AbstractMacroCommand*> cmds_;
};

} // namespace command

void Test_15_command_impl_2() {
  printf("-------------------- %s --------------------\n",
__FUNCTION__);
    std::shared_ptr<command::AbstractMacroCommand> cmd_A(new
command::ConcreteCommandA());
    std::shared_ptr<command::AbstractMacroCommand> cmd_B(new
command::ConcreteCommandB());

    command::CompositeInvoker client;
    client.AddCommand(cmd_A.get());
    client.AddCommand(cmd_B.get());

    client.Request();
}

// -------------------- Test_15_command_impl_2 --------------------
// this is CompositeReceiver::ActionA!
// this is CompositeReceiver::ActionB!
```

- 与 `备忘录模式` 结合, 实现**命令的撤销与恢复**

行为型 - 责任链

- 含义

  `Chain of Responsibility` **把请求从链中的一个对象传到下一个对象，直到请求被响应为止**。通过这种方式去除对象之间的耦合。

  注: 责任链模式也叫**职责链模式**。

  - 优点
    - **降低了对象之间的耦合度**。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
    - 增强了系统的**可扩展性**。可以根据需要增加新的请求处理类，**满足开闭原则**。
    - 增强了给对象指派职责的**灵活性**。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。
    - 责任链简化了对象之间的连接。**每个对象只需保持一个指向其后继者的引用**，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。
    - 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，**符合类的单一职责原则**。
  - 缺点
    - **不能保证每个请求一定被处理**。由于一个请求没有明确的接收者，该请求可能一直传到链的末端都得不到处理。
    - 对比较长的职责链，请求的处理可能涉及多个处理对象，**系统性能将受到一定影响**。
    - 职责链建立的**合理性要靠客户端来保证**，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。
- 应用场景

  - 在现实生活中，常常会出现这样的事例：**一个请求有多个对象可以处理，但每个对象的处理条件或权限不同**。例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同，员工必须根据自己要请假的天数去找不同的领导签名，也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。
  - 在计算机软硬件中也有相关例子，如总线网中数据报传送，每台计算机根据目标地址是否同自己的地址相同来决定是否接收；还有异常处理中，处理程序根据异常的类型决定自己是否处理该异常等。

- 示例

```cpp
#pragma once

#include <iostream>
#include <string>

namespace responsibility {

/// @brief 抽象处理者, 定义一个处理请求的接口，包含抽象处理方法和一个后继连接
class AbstractHandler {
public:
    virtual ~AbstractHandler() {}
    virtual void SetNext(AbstractHandler* handler) {
        next_ = handler;
    }
    virtual AbstractHandler* GetNext() {
        return next_;
    }

    virtual void Request(const std::string& state) = 0;

private:
    AbstractHandler* next_ = nullptr;
};

/// @brief 具体处理者, 实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者
class ConcreteHandlerA : public AbstractHandler {
public:
    virtual void Request(const std::string& state) {
        if (0 == state.compare("A")) {
            printf("this is ConcreteHandlerA::Request!\n");
        }
        else {
            if (GetNext()) {
                GetNext()->Request(state);
            }
            else {
                printf("From A, Nobody handle this request!\n");
            }
        }
    }
};
class ConcreteHandlerB : public AbstractHandler {
public:
    virtual void Request(const std::string& state) {
        if (0 == state.compare("B")) {
            printf("this is ConcreteHandlerB::Request!\n");
        }
        else {
            if (!GetNext()) {
                GetNext()->Request(state);
            }
            else {
                printf("From B, Nobody handle this request!\n");
            }
        }
    }
};

} // namespace responsibility
```

```
void Test_16_chain_of_responsibility_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<responsibility::AbstractHandler> ptr_A(new
responsibility::ConcreteHandlerA());
    std::shared_ptr<responsibility::AbstractHandler> ptr_B(new
responsibility::ConcreteHandlerB());

    ptr_A->SetNext(ptr_B.get());
    ptr_A->Request("B");
}

// -------------------- Test_16_chain_of_responsibility_impl_1 -------------
-------
// this is ConcreteHandlerB::Request!
```

行为型 - 状态

- 含义

  `State` **允许一个对象在其内部状态发生改变时改变其行为能力**。

  在软件开发过程中，**应用程序中的有些对象可能会根据不同的情况做出不同的行为**，我们把这种对象称为有状态的对象，而把影响对象行为的一个或多个动态变化的属性称为状态。当有状态的对象与外部事件产生互动时，其内部状态会发生改变，从而使得其行为也随之发生改变。如人的情绪有高兴的时候和伤心的时候，不同的情绪有不同的行为，当然外界也会影响其情绪变化。

  对这种有状态的对象编程，**传统的解决方案是：将这些所有可能发生的情况全都考虑到，然后使用 if-else 语句来做状态判断，再进行不同情况的处理**。但当对象的状态很多时，程序会变得很复杂。而且增加新的状态要添加新的 if-else 语句，这违背了“开闭原则”，不利于程序的扩展。

  - 优点
    - 状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，**满足“单一职责原则”**。
    - **减少对象间的相互依赖**。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。
    - 有**利于程序的扩展**。通过定义新的子类很容易地增加新的状态和转换。
  - 缺点

    状态模式的使用必然会增加系统的类与对象的个数
- 应用场景

  - 当一个对象的行为取决于它的状态，并且它**必须在运行时根据状态改变它的行为时**。
  - 一个操作中含有庞大的分支结构，并且这些分支决定于对象的状态时。

- 示例

开关状态

```cpp
#pragma once

#include <iostream>

namespace state {

/// @brief 环境类, 定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理
class AbstractState;
class EnvContext {
public:
    EnvContext();
    void SetState(std::shared_ptr<AbstractState>& state) {
        state_ = state;
    }
    std::shared_ptr<AbstractState>& GetState() {
        return state_;
    }
    void Operation();

private:
    std::shared_ptr<AbstractState> state_ = nullptr;
};

/// @brief 抽象状态类, 封装环境对象中的特定状态所对应的行为
class AbstractState {
public:
    virtual ~AbstractState() {}
    virtual void Operation(EnvContext* context) = 0;
};

/// @brief 具体状态类, 实现抽象状态所对应的行为
class ConcreteStateA : public AbstractState {
public:
    virtual void Operation(EnvContext* context);
};
class ConcreteStateB : public AbstractState {
public:
    virtual void Operation(EnvContext* context);
};

EnvContext::EnvContext() {
    state_.reset(new ConcreteStateA()); // 给定初始状态
}

void EnvContext::Operation() {
    state_->Operation(this);
}

void ConcreteStateA::Operation(EnvContext* context) {
    printf("Now Your State is A!\n");
    // 同时改变状态
    std::shared_ptr<AbstractState> new_state(new ConcreteStateB());
    context->SetState(new_state);
}

void ConcreteStateB::Operation(EnvContext* context) {
    printf("Now Your State is B!\n");
    // 同时改变状态
    std::shared_ptr<AbstractState> new_state(new ConcreteStateA());
```

```
    context->SetState(new_state);
}

} // namespace state

void Test_17_state_impl_1() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    std::shared_ptr<state::EnvContext> ptr(new state::EnvContext());
    ptr->Operation();   // A
    ptr->Operation();   // B
    ptr->Operation();   // A
    ptr->Operation();   // B
}

// -------------------- Test_17_state_impl_1 --------------------
// Now Your State is A!
// Now Your State is B!
// Now Your State is A!
// Now Your State is B!
```

行为型 - 观察者

- 含义

  `Observer` **多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为**。

  在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。例如，某种商品的物价上涨时会导致部分商家高兴，而消费者伤心；还有，当我们开车到交叉路口时，遇到红灯会停，遇到绿灯会行。这样的例子还有很多，例如，股票价格与股民、微信公众号与微信用户、气象局的天气预报与听众、小偷与警察等。

  在软件世界也是这样，例如，Excel 中的数据与折线图、饼状图、柱状图之间的关系；MVC 模式中的模型与视图的关系；事件模型中的事件源与事件处理者。

  - 优点
    - 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
    - 目标与观察者之间建立了一套触发机制。
  - 缺点
    - 目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。
    - 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

- 应用场景
    - 对象间**存在一对多关系**，**一个对象的状态发生改变会影响其他对象**。
    - 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

- 示例

```
#pragma once

#include <vector>

namespace observer {

/// @brief 抽象观察者, 包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。
class AbstractObserver {
public:
    ~AbstractObserver() {}

    // 被告知时做出响应
    virtual void Response() = 0;
};

/// @brief 具体观察者, 实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。
class ConcreteObserverA : public AbstractObserver {
public:
    virtual void Response() {
        printf("this is ConcreteObserverA::Response!\n");
    }
};
class ConcreteObserverB : public AbstractObserver {
public:
    virtual void Response() {
        printf("this is ConcreteObserverB::Response!\n");
    }
};

/// @brief 抽象主题类, 它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法
class AbstractSubject {
public:
    virtual ~AbstractSubject() {}

    virtual void NotifyObserver() = 0;

    void AddObserver(AbstractObserver* observer) {
        observers_.emplace_back(observer);
    }
    void RemoveObserver(AbstractObserver* observer) {
        for (auto it = observers_.begin(); it != observers_.end(); it++) {
            if ((*it) == observer) {
                observers_.erase(it++);
                break;
            }
        }
    }

protected:
    std::vector<AbstractObserver*> observers_;
};

/// @brief 具体主题类, 它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。
class ConcreteSubject : public AbstractSubject {
public:
    virtual void NotifyObserver() {
        for (auto* observer : observers_) {
```

```
            observer->Response();
        }
    }
};

} // namespace observer

void Test_18_observer_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<observer::AbstractSubject> ptr_subject(new
observer::ConcreteSubject());
    std::shared_ptr<observer::AbstractObserver> ptr_observer_A(new
observer::ConcreteObserverA());
    std::shared_ptr<observer::AbstractObserver> ptr_observer_B(new
observer::ConcreteObserverB());

    ptr_subject->AddObserver(ptr_observer_A.get());
    ptr_subject->AddObserver(ptr_observer_B.get());

    ptr_subject->NotifyObserver();
}

// -------------------- Test_18_observer_impl_1 --------------------
// this is ConcreteObserverA::Response!
// this is ConcreteObserverB::Response!
```

行为型 - 中介者

- 含义

  `Mediator` 定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

  - 优点
    - 降低了对象之间的耦合性，使得对象易于独立地被复用。
    - 将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。
  - 缺点

    当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。
- 应用场景

  - 当对象之间存在**复杂的网状结构关系**而导致依赖关系混乱且难以复用时。
  - 当**想创建一个运行于多个类之间的对象**，又不想生成新的子类时。

- 示例

```cpp
#pragma once

#include <list>

namespace mediator {

/// @brief 抽象同事类, 定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能
class AbstractMediator;
class AbstractColleague {
public:
    virtual ~AbstractColleague() {}

    /// @brief 抽象接口
    virtual void Receive() = 0;
    virtual void Send() = 0;

    /// @brief 设置中介对象
    void SetMedium(AbstractMediator* mediator) {
        mediator_ = mediator;
    }

protected:
    AbstractMediator* mediator_ = nullptr;
};

/// @brief 具体同事类, 抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互
class ConcreteColleagueA : public AbstractColleague {
public:
    virtual void Receive();
    virtual void Send();
};
class ConcreteColleagueB : public AbstractColleague {
public:
    virtual void Receive();
    virtual void Send();
};

/// @brief 抽象中介类, 中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法
class AbstractMediator {
public:
    virtual ~AbstractMediator() {}

    /// @brief 注册
    virtual void Register(AbstractColleague* colleague) = 0;
    /// @brief 转发
    virtual void Relay(AbstractColleague* colleague) = 0;
};

/// @brief 具体中介类, 实现中介者接口. 管理同事对象，协调各个同事角色之间的交互关系，依赖于同事角色
class ConcreteMediator : public AbstractMediator {
public:
    virtual void Register(AbstractColleague* colleague) {
        for (auto* c : colleagues_) {
            if (c == colleague) return;
        }
        colleagues_.emplace_back(colleague);
```

```
        colleague->SetMedium(this);
    }

    virtual void Relay(AbstractColleague* colleague) {
        for (auto* c : colleagues_) {
            // 转发给除自己外的同事
            if (c != colleague) c->Receive();
        }
    }

private:
    std::list<AbstractColleague*> colleagues_;
};

////////////////////////////////////////////////////// 同事类实现
void ConcreteColleagueA::Receive() {
    printf("this is ConcreteColleagueA::Receive!\n");
}
void ConcreteColleagueA::Send() {
    printf("this is ConcreteColleagueA::Send!\n");
    // 请中介者转发
    mediator_->Relay(this);
}
void ConcreteColleagueB::Receive() {
    printf("this is ConcreteColleagueB::Receive!\n");
}
void ConcreteColleagueB::Send() {
    printf("this is ConcreteColleagueB::Send!\n");
    // 请中介者转发
    mediator_->Relay(this);
}

} // namespace mediator

void Test_19_mediator_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<mediator::AbstractMediator> ptr_md(new
mediator::ConcreteMediator());

    std::shared_ptr<mediator::AbstractColleague> ptr_c1(new
mediator::ConcreteColleagueA());
    std::shared_ptr<mediator::AbstractColleague> ptr_c2(new
mediator::ConcreteColleagueB());

    ptr_md->Register(ptr_c1.get());
    ptr_md->Register(ptr_c2.get());

    ptr_c1->Send();

    printf("-----------\n");

    ptr_c2->Send();
}

// -------------------- Test_19_mediator_impl_1 --------------------
// this is ConcreteColleagueA::Send!
// this is ConcreteColleagueB::Receive!
// -----------
// this is ConcreteColleagueB::Send!
// this is ConcreteColleagueA::Receive!
```

行为型 - 迭代器

- 含义

  `Iterator` 提供一种方法来**顺序访问聚合对象中的一系列数据**，而**不暴露聚合对象的内部表示**。

  - 优点
    - 访问一个聚合对象的内容而无须暴露它的内部表示。
    - 遍历任务交由迭代器完成，这简化了聚合类。
    - 它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。
    - 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
    - 封装性良好，为遍历不同的聚合结构提供一个统一的接口。
  - 缺点

    增加了类的个数，这在一定程度上增加了系统的复杂性。
- 应用场景
- 示例

```
// 直接使用 STL
即可.
```

行为型 - 访问者

- 含义

  `Visitor` 在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

  - 优点
    - 扩展性好。**能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。**
    - 复用性好。**可以通过访问者来定义整个对象结构通用的功能**，从而提高系统的复用程度。
    - 灵活性好。访问者模式**将数据结构与作用于结构上的操作解耦**，使得操作集合可相对自由地演化而不影响系统的数据结构。
    - **符合单一职责原则**。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。
  - 缺点
    - 增加新的元素类很困难。在访问者模式中，**每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”**。
    - 破坏封装。访问者模式中**具体元素对访问者公布细节，这破坏了对象的封装性**。
    - 违反了依赖倒置原则。访问者模式**依赖了具体类，而没有依赖抽象类**。

- 应用场景
    - 对象结构相对稳定，但其**操作算法经常变化**的程序。
    - **对象结构中的对象需要提供多种不同且不相关的操作**，而且要避免让这些操作的变化影响对象的结构。
    - 对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作。

- 示例

```
#pragma once

#include <iostream>
#include <list>

namespace visitor {

/// @brief 抽象访问者, 定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作，该操作中的参数类型标识了被访问的具体元素。
class ConcreteElementA;
class ConcreteElementB;
class AbstractVisitor {
public:
    virtual ~AbstractVisitor() {}

    /// @brief 访问操作, 依赖于具体元素类
    virtual void Visit(ConcreteElementA* element) = 0;
    virtual void Visit(ConcreteElementB* element) = 0;
};

/// @brief 具体访问者, 实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。
class ConcreteVisitorA : public AbstractVisitor {
public:
    virtual void Visit(ConcreteElementA* element);
    virtual void Visit(ConcreteElementB* element);
};
class ConcreteVisitorB : public AbstractVisitor {
public:
    virtual void Visit(ConcreteElementA* element);
    virtual void Visit(ConcreteElementB* element);
};

/// @brief 抽象元素, 声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数
class AbstractElement {
public:
    virtual ~AbstractElement() {}

    virtual void Accept(AbstractVisitor* visitor) = 0;
};

/// @brief 具体元素角色：实现抽象元素角色提供的 accept() 操作，另外具体元素中可能还包含本身业务逻辑的相关操作。
class ConcreteElementA : public AbstractElement {
public:
    virtual void Accept(AbstractVisitor* visitor) {
        visitor->Visit(this);
    }

    void OperatorA() {
        printf("this is ConcreteElementA::OperatorA!\n");
    }
};
class ConcreteElementB : public AbstractElement {
public:
    virtual void Accept(AbstractVisitor* visitor) {
        visitor->Visit(this);
    }
```

```
    void OperatorB() {
        printf("this is ConcreteElementB::OperatorB!\n");
    }
};

void ConcreteVisitorA::Visit(ConcreteElementA* element) {
    printf("this is ConcreteVisitorA::Visit!\n");
    element->OperatorA();
}
void ConcreteVisitorA::Visit(ConcreteElementB* element) {
    printf("this is ConcreteVisitorA::Visit!\n");
    element->OperatorB();
}
void ConcreteVisitorB::Visit(ConcreteElementA* element) {
    printf("this is ConcreteVisitorB::Visit!\n");
    element->OperatorA();
}
void ConcreteVisitorB::Visit(ConcreteElementB* element) {
    printf("this is ConcreteVisitorB::Visit!\n");
    element->OperatorB();
}

} // namespace visitor

void Test_20_visitor_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::list<visitor::AbstractElement*> elements;
    elements.emplace_back(new visitor::ConcreteElementA());
    elements.emplace_back(new visitor::ConcreteElementB());

    // 为所有元素设置访问者A
    std::shared_ptr<visitor::AbstractVisitor> ptr_visitor_a(new
visitor::ConcreteVisitorA());
    for (auto* e : elements) {
        e->Accept(ptr_visitor_a.get());
    }

    printf("----------------------\n");

    // 为所有元素设置访问者B
    std::shared_ptr<visitor::AbstractVisitor> ptr_visitor_b(new
visitor::ConcreteVisitorB());
    for (auto* e : elements) {
        e->Accept(ptr_visitor_b.get());
    }

    for (auto* e : elements) {
        if (e) delete e;
    }
}

// -------------------- Test_20_visitor_impl_1 --------------------
// this is ConcreteVisitorA::Visit!
// this is ConcreteElementA::OperatorA!
// this is ConcreteVisitorA::Visit!
// this is ConcreteElementB::OperatorB!
// ----------------------
// this is ConcreteVisitorB::Visit!
// this is ConcreteElementA::OperatorA!
// this is ConcreteVisitorB::Visit!
// this is ConcreteElementB::OperatorB!
```

行为型 - 备忘录

- 含义

  `Memento` 在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

  - 优点
    - 提供了一种可以恢复状态的机制。**当用户需要时能够比较方便地将数据恢复到某个历史的状态**。
    - 实现了内部状态的封装。**除了创建它的发起人之外，其他对象都不能够访问这些状态信息**。
    - 简化了发起人类。**发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合** `单一职责原则` 。
  - 缺点

    资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。
- 应用场景

  很多应用软件都提供了这项功能，如 Word、记事本、Photoshop、Eclipse 等软件在编辑时按 Ctrl+Z 组合键时能撤销当前操作，使文档恢复到之前的状态；还有在 IE 中的后退键、数据库事务管理中的回滚操作、玩游戏时的中间结果存档功能、数据库与操作系统的备份操作、棋类游戏中的悔棋功能等都属于这类。

- 示例

```
#pragma once

namespace memento {

/// @brief 需求: 利用备忘录模式设计相亲游戏
///     假如有西施、王昭君、貂蝉、杨玉环四大美女同你相亲，你可以选择其中一位作为你的爱人，游戏提供后悔机制，不爱了可以换人

#include <iostream>
#include <string>
#include <vector>

/// @brief 备忘录, 提供了获取和存储美女信息的功能
class Memento_Girl {
public:
    Memento_Girl(const std::string& name) : name_(name) {}
    const std::string& GetName() const {
        return name_;
    }

private:
    std::string name_;
};

/// @brief 发起人, 记录当前时刻的内部状态信息（临时妻子的姓名）
class Originator_Man {
public:
    void SetGirlName(const std::string& girl) {
        girl_name_ = girl;
    }
    const std::string& GetGirlName() const {
        return girl_name_;
    }

    /// 提供创建备忘录和恢复备忘录数据的功能
    Memento_Girl* CreateMemento_Gril() {
        return new Memento_Girl(girl_name_);
    }
    void RestoreMemento(Memento_Girl* m) {
        SetGirlName(m->GetName());
    }

private:
    std::string girl_name_;
};

/// @brief 管理者, 对备忘录进行管理，用于保存相亲者（Man）前面选过的美女信息
class Caretaker_GirlStack {
public:
    ~Caretaker_GirlStack() {
        for (auto* girl : girls_) {
            delete girl;
        }
    }
    void Push(Memento_Girl* girl) {
        girls_.emplace_back(girl);
    }
    Memento_Girl* Pop() {
        Memento_Girl* girl = girls_.back();
        girls_.pop_back();
```

```
        return girl;
    }

private:
    std::vector<Memento_Girl*> girls_;
};

} // namespace memento

void Test_22_memento_impl_1() {
    printf("-------------------- %s --------------------\n", __FUNCTION__);
    std::shared_ptr<memento::Originator_Man> man_ptr(new
memento::Originator_Man());
    std::shared_ptr<memento::Caretaker_GirlStack> girl_stack_ptr(new
memento::Caretaker_GirlStack());

    man_ptr->SetGirlName("XiShi");
    girl_stack_ptr->Push(man_ptr->CreateMemento_Gril());
    printf("Now, You Select: %s\n", man_ptr->GetGirlName().c_str());

    man_ptr->SetGirlName("WangZhaoJun");
    printf("Now, You Reselect: %s\n", man_ptr->GetGirlName().c_str());

    man_ptr->RestoreMemento(girl_stack_ptr->Pop());
    printf("Now, You Restore To Select: %s\n", man_ptr-
>GetGirlName().c_str());
}

// -------------------- Test_22_memento_impl_1 --------------------
// Now, You Select: XiShi
// Now, You Reselect: WangZhaoJun
// Now, You Restore To Select: XiShi
```

行为型 - 解释器

- 含义

  `Interpreter`：为语言创建解释器，通常由语言的语法和语法分析来定义。

  - 优点
    - 扩展性好。由于在解释器模式中使用类来表示语言的文法规则，因此可以**通过继承等机制来改变或扩展文法**。
    - 容易实现。在语法树中的**每个表达式节点类都是相似的**，所以实现其文法较为容易。
  - 缺点
    - 执行效率较低。解释器模式中**通常使用大量的循环和递归调用**，当要解释的句子较复杂时，其运行速度很慢，且代码的调试过程也比较麻烦。
    - 会引起类膨胀。解释器模式中的**每条规则至少需要定义一个类**，当包含的文法规则很多时，类的个数将急剧增加，导致系统难以管理与维护。
    - 可应用的场景比较少。**在软件开发中，需要定义语言文法的应用实例非常少**，所以这种模式很少被使用到。
- 应用场景

  - 当语言的文法较为简单，且执行效率不是关键问题时。
  - 当问题重复出现，且可以用一种简单的语言来进行表达时。
  - 当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如 XML 文档解释。

  注意：解释器模式在实际的软件开发中使用比较少，因为**它会引起效率、性能以及维护等问题**。

- 示例

用解释器模式设计一个"厦漳泉"公交车卡的读卡器程序。

```
#pragma once

#include <iostream>
#include <string>
#include <vector>

/*
    用解释器模式设计一个"厦漳泉"公交车卡的读卡器程序。

    说明：假如"厦漳泉"公交车读卡器可以判断乘客的身份，如果是这三座城市的"老人""
"妇女" "儿童"就可以免费乘车，其他人员乘车一次扣 2 元。
*/
/*
    文法规则
    <expression> ::= <city>的<person>
    <city> ::= 厦门|泉州|漳州
    <person> ::= 老人|妇女|儿童
*/

namespace interpreter {

/// @brief 抽象表达式, 定义解释器的接口，约定解释器的解释操作，主要包含解释方法
interpret()。
class AbstractExpression {
public:
    virtual ~AbstractExpression() {}

    /// @brief 解释方法
    virtual bool Interpret(const std::string& info) = 0;
};

/// @brief 终结符表达式, 是抽象表达式的子类，用来实现文法中与终结符相关的操作，文
法中的每一个终结符都有一个具体终结表达式与之相对应。
class TerminalExpression : public AbstractExpression {
public:
    TerminalExpression(const std::vector<std::string>& content) :
contents_(content) {}

    virtual bool Interpret(const std::string& info) {
        for (auto& content : contents_) {
            if (info == content) return true;
            std::size_t len = info.size();
            for (int i = 0; i < content.size(); i++) {
                if (i + len  <= content.size() && content.substr(i,
i+len).compare(info) == 0) {
                    return true;
                }
            }
        }
        return false;
    }

private:
    std::vector<std::string> contents_;
};

/// @brief 非终结符表达式, 也是抽象表达式的子类，用来实现文法中与非终结符相关的操
作，文法中的每条规则都对应于一个非终结符表达式。
class NonterminalExpression : public AbstractExpression {
```

```
public:
    NonterminalExpression(std::shared_ptr<AbstractExpression>&
city_eps, std::shared_ptr<AbstractExpression>& person_eps) :
city_eps_(city_eps), person_eps_(person_eps) {}

    virtual bool Interpret(const std::string& info) {
        std::vector<std::string> split_vec;
        if (2 != Split(info, '|', split_vec)) {
            return false;
        }
        return city_eps_->Interpret(split_vec[0]) && person_eps_-
>Interpret(split_vec[1]);
    }

protected:
    size_t Split(const std::string& src, const char ch,
std::vector<std::string>& vec) {
        size_t nPos = 0;
        std::string tmp;
        for (size_t i = 0; i < src.size(); i++) {
            if (src[i] == ch) {
                tmp = src.substr(nPos, i - nPos);
                if (!tmp.empty() && tmp != "\n")
                    vec.push_back(tmp);
                nPos = i + 1;
            }
        }
        if (nPos <= src.size()) {
            tmp = src.substr(nPos, src.size() - nPos);
            if (!tmp.empty() && tmp != "\n")
                vec.push_back(tmp);
        }
        return vec.size();
    }

private:
    std::shared_ptr<AbstractExpression> city_eps_ = nullptr;
    std::shared_ptr<AbstractExpression> person_eps_ = nullptr;
};

/// @brief 环境类，通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有
解释器共享的数据，后面的解释器可以从这里获取这些值。
class Context {
public:
    Context() {
        std::vector<std::string> citys;
        std::vector<std::string> persons;

        citys.emplace_back("厦门");
        citys.emplace_back("漳州");
        citys.emplace_back("泉州");

        persons.emplace_back("老人");
        persons.emplace_back("妇女");
        persons.emplace_back("儿童");

        std::shared_ptr<AbstractExpression> city_eps(new
TerminalExpression(citys));
        std::shared_ptr<AbstractExpression> person_eps(new
TerminalExpression(persons));
```

```
        eps_.reset(new NonterminalExpression(city_eps, person_eps));
    }
    void FreeRide(const std::string& info) {
        if (eps_ && eps_->Interpret(info)) {
            printf("身份识别: %s, 您本次乘车免费！\n", info.c_str());
        }
        else {
            printf("身份识别: %s, 本次乘车扣费2元！\n", info.c_str());
        }
    }

protected:
    std::shared_ptr<AbstractExpression> eps_;
};

} // namespace interpreter

void Test_23_visitor_impl_1() {
    printf("-------------------- %s --------------------\n",
__FUNCTION__);
    std::shared_ptr<interpreter::Context> context(new
interpreter::Context());
    context->FreeRide("厦门|老人");
    context->FreeRide("漳州|妇女");
    context->FreeRide("泉州|儿童");
    context->FreeRide("上海|老人");
    context->FreeRide("厦门|青年");
}

// -------------------- Test_23_visitor_impl_1 --------------------
// 身份识别: 厦门|老人, 您本次乘车免费！
// 身份识别: 漳州|妇女, 您本次乘车免费！
// 身份识别: 泉州|儿童, 您本次乘车免费！
// 身份识别: 上海|老人, 本次乘车扣费2元！
// 身份识别: 厦门|青年, 本次乘车扣费2元！
```