七大原则

开放封闭、单一职责、迪米特、依赖倒置、接口隔离、里氏替换、合成复用

1. **开放封闭原则（Open-Closed Principle）**
    - **简述**: 软件实体应该对扩展开放，对修改关闭。这意味着在不改变现有代码的情况下，可以通过增加新代码来扩展功能。
    - **例子**: 比如一个图形编辑器的程序，为了增加新的图形类型，你可以新增一个继承自图形基类的子类，而无需修改现有的图形类和使用它们的代码。
2. **单一职责原则（Single Responsibility Principle）**
    - **简述**: 一个类应该只有一个改变的理由，即一个类只负责一件事情。
    - **例子**: 如果有一个类负责两件事，如用户信息管理和用户权限验证，则应该将它分为两个类，每个类各自处理一个职责。
3. **依赖倒置原则（Dependency Inversion Principle）**
    - **简述**: 高层模块不应该依赖低层模块，它们都应该依赖于抽象；抽象不应该依赖于细节，细节应该依赖于抽象。
    - **例子**: 在数据库访问的应用中，高层的业务逻辑应该依赖于抽象的数据访问接口，而不是具体的数据库访问类。
4. **接口隔离原则（Interface Segregation Principle）**
    - **简述**: 不应该强迫客户依赖于它们不用的接口。
    - **例子**: 如果一个接口有过多的方法，并且某些实现类只用到了其中的一部分，则应该将这个接口拆分为几个更小的接口。
5. **里氏替换原则（Liskov Substitution Principle）**
    - **简述**: 子类应该能够替换它们的基类。
    - **例子**: 如果有一个函数接受一个基类对象作为参数，那么它在传入该基类的任何子类时也应该能正常工作。
6. **迪米特原则（Law of Demeter）**
    - **简述**: 一个对象应该对其他对象有最少的了解。
    - **例子**: 在设计类时，任何一个类都应该尽量减少对其他类的了解，只和朋友类通信。
7. **合成复用原则（Composite Reuse Principle）**
    - **简述**: 尽量使用对象组合(has-a)/聚合(contains-a)，而不是继承关系(is-a)来达到复用的目的。
    - **例子**: 如果需要增加一些新功能，可以通过包含一些对象的方式来实现，而不是通过继承一个包含这些功能的类。

设计模式分类

分为创建型、结构型、行为型

1. **创建型模式（Creational Patterns）**
    - 这类模式专注于处理对象创建机制，尝试以适合于特定情况的方式创建对象。创建型模式减少了程序与所需对象的直接依赖关系。
    - **包含的模式**:
        - **工厂方法模式（Factory Method）**：通过让子类决定应该实例化哪个类来创建对象。
        - **抽象工厂模式（Abstract Factory）**：允许创建一系列相关或相互依赖的对象，而无需指定它们的具体类。

        - **单例模式（Singleton）**：确保类只有一个实例，并提供对此实例的全局访问。
        - **建造者模式（Builder）**：允许创建复杂对象的逐步构造。
        - **原型模式（Prototype）**：通过复制现有的实例来创建新的实例。
2. **结构型模式（Structural Patterns）**
    - 这类模式关注类和对象的组合。结构型模式通过构造方式来实现新功能的组合。
    - **包含的模式**:
        - **适配器模式（Adapter）**：允许接口不兼容的对象能够相互合作。
        - **装饰器模式（Decorator）**：动态地给对象添加额外的职责。
        - **代理模式（Proxy）**：为另一个对象提供一个替身或占位符以控制对这个对象的访问。
        - **外观模式（Facade）**：提供了一个统一的接口，用来访问子系统中的一群接口。
        - **桥接模式（Bridge）**：将抽象部分与实现部分分离，使它们都可以独立地变化。
        - **组合模式（Composite）**：允许将对象组合成树形结构以表示“部分-整体”的层次结构。
        - **享元模式（Flyweight）**：减少对象数量以减少内存占用，通常用于处理大量小粒度的对象。
3. **行为型模式（Behavioral Patterns）**
    - 这类模式专注于对象之间的通信。
    - **包含的模式**:
        - **策略模式（Strategy）**：允许在运行时选择算法的行为。
        - **模板方法模式（Template Method）**：在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。
        - **观察者模式（Observer）**：当一个对象状态改变时，所有依赖于它的对象都会得到通知并被自动更新。
        - **迭代子模式（Iterator）**：提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。
        - **责任链模式（Chain of Responsibility）**：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。
        - **命令模式（Command）**：将一个请求封装成一个对象，从而使你可用不同的请求对客户进行参数化。
        - **备忘录模式（Memento）**：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。
        - **状态模式（State）**：允许一个对象在其内部状态改变时改变它的行为。
        - **访问者模式（Visitor）**：表示一个作用于某对象结构中的各元素的操作，它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。
        - **中介者模式（Mediator）**：封装一系列对象相互作用的方式，使得这些对象不需要显式地相互引用。
        - **解释器模式（Interpreter）**：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

三种工厂模式

简单工厂模式（Simple Factory）

- **简述**: 简单工厂模式并不是一个真正的设计模式，更像是一种编程习惯。在这个模式中，一个工厂类根据传入的参数决定创建出哪一种产品类的实例。简单工厂通常用一个静态方法来创建对象。
- **例子**: 假设有一个应用需要根据不同的文件格式（如XML、JSON、CSV）来解析文件，简单工厂可以根据传入的文件格式返回相应的解析器对象。

```
#include <iostream>
#include <memory>
```

```cpp
// 产品基类
class Product {
public:
    virtual void operation() = 0;
    virtual ~Product() {}
};

// 具体产品A
class ConcreteProductA : public Product {
public:
    void operation() override {
        std::cout << "Operation of ConcreteProductA\n";
    }
};

// 具体产品B
class ConcreteProductB : public Product {
public:
    void operation() override {
        std::cout << "Operation of ConcreteProductB\n";
    }
};

// 简单工厂
class SimpleFactory {
public:
    static std::unique_ptr<Product> createProduct(const std::string& type) {
        if (type == "A")
            return std::make_unique<ConcreteProductA>();
        else if (type == "B")
            return std::make_unique<ConcreteProductB>();
        else
            return nullptr;
    }
};

// 客户端代码
int main() {
    auto product = SimpleFactory::createProduct("A");
    if (product) {
        product->operation();
    }
    return 0;
}
```

工厂模式（Factory Method）

- **简述**: 工厂方法模式定义了一个创建对象的接口，但让实现这个接口的子类决定实例化哪个类。工厂方法模式让类的实例化推迟到其子类。
- **例子**: 考虑一个日志记录器的应用，可以有多种日志记录方式（如文件记录、数据库记录等）。使用工厂方法，可以为每种记录方式提供一个创建日志记录器的工厂。

```cpp
#include <iostream>
#include <memory>
```

```cpp
// 产品基类
class Product {
public:
    virtual void operation() = 0;
    virtual ~Product() {}
};

// 具体产品A
class ConcreteProductA : public Product {
public:
    void operation() override {
        std::cout << "Operation of ConcreteProductA\n";
    }
};

// 具体产品B
class ConcreteProductB : public Product {
public:
    void operation() override {
        std::cout << "Operation of ConcreteProductB\n";
    }
};

// 抽象工厂
class Factory {
public:
    virtual std::unique_ptr<Product> createProduct() = 0;
    virtual ~Factory() {}
};

// 具体工厂A
class ConcreteFactoryA : public Factory {
public:
    std::unique_ptr<Product> createProduct() override {
        return std::make_unique<ConcreteProductA>();
    }
};

// 具体工厂B
class ConcreteFactoryB : public Factory {
public:
    std::unique_ptr<Product> createProduct() override {
        return std::make_unique<ConcreteProductB>();
    }
};

// 客户端代码
int main() {
    std::unique_ptr<Factory> factory = std::make_unique<ConcreteFactoryA>();
    auto product = factory->createProduct();
    product->operation();

    factory = std::make_unique<ConcreteFactoryB>();
    product = factory->createProduct();
    product->operation();

    return 0;
```

```
}
```

抽象工厂模式（Abstract Factory）

- **简述**: 抽象工厂模式提供了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。抽象工厂通常包含多个工厂方法来创建不同类型的对象。
- **例子**: 如果你有一个跨平台的UI组件库，不同的操作系统（如Windows、Linux、macOS）需要不同的UI组件实现（如按钮、文本框）。抽象工厂可以为每个操作系统提供一个工厂，这些工厂分别创建与操作系统兼容的UI组件。

```cpp
#include <iostream>
#include <memory>

// 抽象产品A
class AbstractProductA {
public:
    virtual void operationA() = 0;
    virtual ~AbstractProductA() {}
};

// 抽象产品B
class AbstractProductB {
public:
    virtual void operationB() = 0;
    virtual ~AbstractProductB() {}
};

// 具体产品A1
class ProductA1 : public AbstractProductA {
public:
    void operationA() override {
        std::cout << "OperationA of ProductA1\n";
    }
};

// 具体产品B1
class ProductB1 : public AbstractProductB {
public:
    void operationB() override {
        std::cout << "OperationB of ProductB1\n";
    }
};

// 具体产品A2
class ProductA2 : public AbstractProductA {
public:
    void operationA() override {
        std::cout << "OperationA of ProductA2\n";
    }
};

// 具体产品B2
class ProductB2 : public AbstractProductB {
```

```
public:
    void operationB() override {
        std::cout << "OperationB of ProductB2\n";
    }
};

// 抽象工厂
class AbstractFactory {
public:
    virtual std::unique_ptr<AbstractProductA> createProductA() = 0;
    virtual std::unique_ptr<AbstractProductB> createProductB() = 0;
    virtual ~AbstractFactory() {}
};

// 具体工厂1
class ConcreteFactory1 : public AbstractFactory {
public:
    std::unique_ptr<AbstractProductA> createProductA() override {
        return std::make_unique<ProductA1>();
    }

    std::unique_ptr<AbstractProductB> createProductB() override {
        return std::make_unique<ProductB1>();
    }
};

// 具体工厂2
class ConcreteFactory2 : public AbstractFactory {
public:
    std::unique_ptr<AbstractProductA> createProductA() override {
        return std::make_unique<ProductA2>();
    }

    std::unique_ptr<AbstractProductB> createProductB() override {
        return std::make_unique<ProductB2>();
    }
};

// 客户端代码
int main() {
    std::unique_ptr<AbstractFactory> factory =
std::make_unique<ConcreteFactory1>();
    auto productA = factory->createProductA();
    auto productB = factory->createProductB();
    productA->operationA();
    productB->operationB();

    factory = std::make_unique<ConcreteFactory2>();
    productA = factory->createProductA();
    productB = factory->createProductB();
    productA->operationA();
    productB->operationB();

    return 0;
}
```

三者的区别

1. **目的和抽象级别**:
   - 简单工厂: 用一个类来创建所有类型的对象，是最简单的形式。
   - 工厂方法: 定义一个用于创建对象的接口，让子类决定实例化哪个类。每个生成的实例通常都有一个共同的父类或接口。
   - 抽象工厂: 创建一系列相关或相互依赖的对象。它包含多个创建对象的方法，用于创建一系列相关的对象。
2. **用途和应用场景**:
   - 简单工厂: 当创建逻辑不复杂，且工厂类可以覆盖创建所有实例时使用。
   - 工厂方法: 当有一个父类与多个子类，并且需要动态决定要使用哪个子类时。
   - 抽象工厂: 当需要创建一组相关或依赖的对象时。例如，不同的UI主题可能需要不同的组件集合。
3. **灵活性和扩展性**:
   - 简单工厂: 不太灵活，增加新的产品类需要修改工厂类。
   - 工厂方法: 比简单工厂灵活，新增产品类时无需修改现有工厂类。
   - 抽象工厂: 最为灵活，可以轻松添加新的工厂和产品族，但增加新的产品等级结构可能需要修改接口和所有工厂实现。

单例模式

单例模式是一种确保类只有一个实例，并提供一个全局访问点的创建型设计模式。在单例模式中，该类负责创建自己的对象，并确保只有单个对象被创建。这个类提供了一种方式来访问其唯一的对象，可以直接访问，不需要实例化该类的对象。

在C++中实现单例模式有几种方法，包括“饿汉式”、“懒汉式”和“函数内静态”方式。

1. 饿汉式实现

在饿汉式实现中，实例在程序加载时就被创建。

```
class Singleton {
private:
    static Singleton instance;

    // 私有构造函数，防止外部创建实例
    Singleton() {}

public:
    // 禁止拷贝和赋值
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    // 提供全局访问点
    static Singleton& getInstance() {
        return instance;
    }
};

// 在类外初始化静态成员
Singleton Singleton::instance;
```

2. 懒汉式实现

在懒汉式实现中，实例在第一次使用时被创建。为了保证线程安全，使用了双重检查锁定。

```
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mutex;

    // 私有构造函数
    Singleton() {}

public:
    // 禁止拷贝和赋值
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    // 提供全局访问点
    static Singleton* getInstance() {
        if (instance == nullptr) {
            std::lock_guard<std::mutex> lock(mutex);
            if (instance == nullptr) {
                instance = new Singleton();
            }
        }
        return instance;
    }
};

// 初始化静态成员
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mutex;
```

3. 函数内静态实现

这种方法利用了局部静态变量的特性，确保只在第一次访问时初始化，并且自动销毁。

```
class Singleton {
private:
    // 私有构造函数
    Singleton() {}

public:
    // 禁止拷贝和赋值
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    // 提供全局访问点
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }
};
```

总结：

- **单例模式**: 确保一个类只有一个实例，并提供全局访问点。
- **饿汉式**: 实例在程序加载时创建，线程安全但可能增加启动时间。
- **懒汉式**: 实例在首次使用时创建，通过双重检查锁定确保线程安全，但有性能损耗。
- **函数内静态**: 利用局部静态变量，线程安全，无锁，推荐使用。

代理模式（Proxy Pattern）

- **核心概念**: 为其他对象提供一种代理，以控制对这个对象的访问。
- **例子**: 购书代理服务。它代替你（客户端）与实际的书店（目标对象）进行交互，处理购买过程中的复杂问题。

适配器模式（Adapter Pattern）

- **核心概念**: 将一个类的接口转换成客户希望的另一个接口，使原本由于接口不兼容而不能一起工作的那些类可以一起工作。
- **例子**: USB到Type-C的适配器。它允许USB闪存盘（一个接口）与Type-C接口的笔记本电脑（另一个接口）连接和工作。

模板模式（Template Pattern）

- **核心概念**: 在一个方法中定义一个算法的框架，而将一些步骤的实现延迟到子类中。
- **例子**: 烹饪食物的过程。不同的食物（如披萨和汉堡）遵循相同的步骤（准备原料、烹饪、上菜），但具体细节各不相同。

装饰器模式（Decorator Pattern）

- **核心概念**: 允许向一个现有的对象添加新的功能，同时又不改变其结构。
- **例子**: 制作咖啡并添加配料（如糖、牛奶）。每次添加一个配料，都是在不改变咖啡本身的情况下增加了额外的功能。

观察者模式（Observer Pattern）

- **核心概念**: 对象之间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知并自动更新。
- **例子**: 订阅报纸服务。报社（被观察对象）有新报纸出版时，所有订阅者（观察者）会收到新报纸。