设计原则

设计原则是设计模式的评判标准。在面向对象设计中，有以下几种主要的设计原则[1][2][3][4]：

1. **单一职责原则（Single Responsibility Principle）**：一个类应该只有一个引起它变化的原因[1]。也就是说，一个类或者模块只负责一项职责[2]。
2. **开闭原则（Open Closed Principle）**：软件实体（类、模块、函数等等）应该可以扩展，但是不可修改[1]。也就是说，对扩展开放，对修改封闭[2]。
3. **里氏替换原则（Liskov Substitution Principle）**：所有引用基类的地方必须能透明地使用其子类的对象[1]。也就是说，子类可以扩展父类的功能，但不能改变父类原有的功能[2]。
4. **依赖倒置原则（Dependence Inversion Principle）**：高层模块不应该依赖低层模块，二者都应该依赖其抽象[1]。也就是说，要面向接口编程，不要面向实现编程[2]。
5. **接口隔离原则（Interface Segregation Principle）**：使用多个专门的接口，而不使用单一的总接口[1]。也就是说，客户端不应该依赖它不需要的接口[2]。
6. **迪米特法则（Law of Demeter）**：一个对象应该对其他对象保持最少的了解[1]。也叫作"最少知道原则"[2]。
7. **合成复用原则（Composite Reuse Principle）**：尽量使用对象组合/聚合，而不是继承[1]。也就是说，尽量使用组合/聚合，以便在系统中使用封装变化的概念[2]。

(1) 六大设计原则超详细介绍（再不理解你打我） - 知乎专栏. https://zhuanlan.zhihu.com/p/110130347.
(2) 一篇搞定设计原则和 23 种设计模式 - 知乎. https://zhuanlan.zhihu.com/p/457307426.
(3) 设计原则|极客教程. https://geek-docs.com/design-pattern/design-principle/design-principle-index.html.
(4) 深入理解设计模式！六大设计原则的分析与介绍-阿里 .... https://developer.aliyun.com/article/812062.
(5) undefined. https://github.com/CatchZeng/typescript-design-patterns设计.

设计模式

1. **享元模式（Flyweight Pattern）**：这种模式主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了一种减少对象数量从而改善应用所需的对象结构的方式。
2. **代理模式（Proxy Pattern）**：这种模式涉及到一个类，该类表示另一个类的功能。这种类型的设计模式属于结构型模式。在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。
3. **责任链模式（Chain of Responsibility Pattern）**：这种模式创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。
4. **命令模式（Command Pattern）**：数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。
5. **迭代器模式（Iterator Pattern）**：这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。迭代器模式属于行为型模式。
6. **观察者模式（Observer Pattern）**：当对象间存在一对多关系时使用的设计模式。例如，当一个对象被修改时，则会自动通知依赖它的对象。观察者模式属于行为型模式。
7. **状态模式（State Pattern）**：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像改变了其类。状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。

8. **策略模式（Strategy Pattern）**：它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。策略模式属于行为型模式。
9. **模板模式（Template Pattern）**：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。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。模板模式属于行为型模式。
10. **访问者模式（Visitor Pattern）**：主要将数据结构与数据操作分离。访问者模式属于行为型模式。
11. **单例模式（Singleton Pattern）**：这种模式涉及到一个类，该类创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。
12. **建造者模式（Builder Pattern）**：使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。
13. **原型模式（Prototype Pattern）**：这种模式涉及到实现一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。
14. **适配器模式（Adapter Pattern）**：作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。
15. **装饰器模式（Decorator Pattern）**：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。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。
16. **外观模式（Facade Pattern）**：隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。
17. **桥接模式（Bridge Pattern）**：用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。
18. **组合模式（Composite Pattern）**：用于需要表示整体与部分的层次结构的场景。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。这种模式创建了一个包含自己对象组的类，该类提供了修改相同对象组的方式。
19. **中介者模式（Mediator Pattern）**：中介者模式提供了一种方式，使得对象之间不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。在这个模式中，通常会有一个中介者对象和多个同事对象。同事对象做自己的事情，但是与其他同事对象的通信都通过中介者对象进行。
20. **解释器模式（Interpreter Pattern）**：解释器模式提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式涉及到实现一个表达式接口，该接口解释一个特定的上下文。此模式用于 SQL 解析、符号处理引擎等。
21. **备忘录模式（Memento Pattern）**：备忘录模式保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于行为型模式。
22. **工厂模式（Factory Pattern）**：这种模式涉及到一个类，该类负责创建其他类的实例。在工厂模式中，我们不会直接使用 `new` 操作符来创建对象，而是使用工厂类的方法来创建。这种设计模式属于创建型模式，它提供了一种创建对象的最佳方式。在工厂模式中，当创建对象时，我们不会向客户端显示创建逻辑，并且总是使用一个共同的接口来创建新的对象。这样，当用户需要扩展新的类型时，只需要修改工厂类，而不需要修改用户的代码，从而降低了系统的耦合度。

这22个设计模式可以按照它们的类型进行分类：

创建型模式：

1. 单例模式（Singleton Pattern）
2. 建造者模式（Builder Pattern）
3. 原型模式（Prototype Pattern）
4. 工厂模式（Factory Pattern）

结构型模式：

1. 享元模式（Flyweight Pattern）
2. 代理模式（Proxy Pattern）
3. 适配器模式（Adapter Pattern）
4. 装饰器模式（Decorator Pattern）
5. 外观模式（Facade Pattern）
6. 桥接模式（Bridge Pattern）
7. 组合模式（Composite Pattern）

行为型模式：

1. 责任链模式（Chain of Responsibility Pattern）
2. 命令模式（Command Pattern）
3. 迭代器模式（Iterator Pattern）
4. 观察者模式（Observer Pattern）
5. 状态模式（State Pattern）
6. 策略模式（Strategy Pattern）
7. 模板模式（Template Pattern）
8. 访问者模式（Visitor Pattern）
9. 中介者模式（Mediator Pattern）
10. 解释器模式（Interpreter Pattern）
11. 备忘录模式（Memento Pattern）

1.享元模式（Flyweight Pattern）

享元模式（Flyweight Pattern）是一种结构型设计模式，主要用于减少创建对象的数量，以减少内存占用和提高性能。这种模式的基本思想是通过共享一些通用的对象，来减少需要创建的对象数量。

具体来说，享元模式将对象分为两类：内部状态和外部状态。内部状态是对象的共享部分，外部状态则是每个对象的不同部分。内部状态是指不随着环境的改变而改变的状态，而外部状态则是随着环境的改变而改变的状态。

例如，假设我们有一个`Circle`对象，它有一个`color`属性。我们可以创建一个`Circle`的工厂，当我们需要一个特定颜色的`Circle`时，我们首先检查工厂中是否已经有了这个颜色的`Circle`，如果有，我们就直接使用这个已经存在的`Circle`，否则我们再创建一个新的`Circle`。这样，对于同一种颜色的`Circle`，我们只需要创建一次，然后就可以多次使用，这就大大减少了需要创建的对象的数量。

享元模式在实际应用中，主要应用于缓存，即客户端如果重复请求某些对象，不必每次查询数据库或者读取文件，而是直接返回内存中缓存的数据。例如，Java中的`String`对象和数据库的连接池就使用了享元模式。

总的来说，享元模式就是通过工厂方法创建对象，在工厂方法内部，很可能返回缓存的实例，而不是新创建实例，从而实现不可变实例的复用。这种模式可以大大提高程序的性能，但是也会增加程序的复杂性[1]。在使用享元模式时，需要注意划分外部状态和内部状态，否则可能会引起线程安全问题。
(1) 享元模式 | 菜鸟教程. https://www.runoob.com/design-pattern/flyweight-pattern.html.
(2) 享元 - 廖雪峰的官方网站. https://www.liaoxuefeng.com/wiki/1252599548343744/1281319417937953.
(3) 大幅提升程序性能的秘密武器——C++享元模式 - 知乎. https://zhuanlan.zhihu.com/p/630983060.
(4) 秒懂 Java 的享元模式 - 知乎. https://zhuanlan.zhihu.com/p/608779317.
(5) 大幅提升程序性能的秘密武器——C++享元模式 - 知乎. https://bing.com/search?q=%e4%ba%ab%e5%85%83%e6%a8%a1%e5%bc%8f.

```
class Flyweight {
```

```
public:
    virtual void Operation(int extrinsicstate) = 0;
};

class ConcreteFlyweight : public Flyweight {
public:
    void Operation(int extrinsicstate) override {
        std::cout << "ConcreteFlyweight: " << extrinsicstate << std::endl;
    }
};

class UnsharedConcreteFlyweight : public Flyweight {
public:
    void Operation(int extrinsicstate) override {
        std::cout << "UnsharedConcreteFlyweight: " << extrinsicstate <<
std::endl;
    }
};

class FlyweightFactory {
private:
    std::unordered_map<std::string, Flyweight*> flyweights;
public:
    Flyweight* GetFlyweight(const std::string& key) {
        if (flyweights.find(key) != flyweights.end()) {
            return flyweights[key];
        } else {
            ConcreteFlyweight* fw = new ConcreteFlyweight();
            flyweights.insert(std::make_pair(key, fw));
            return fw;
        }
    }
};
```

这段代码是一个享元模式的C++实现。在这个例子中，`Flyweight`是一个抽象类，它定义了一个`Operation`方法。`ConcreteFlyweight`和`UnsharedConcreteFlyweight`是`Flyweight`的具体实现。

`FlyweightFactory`是一个工厂类，它维护了一个`flyweights`的哈希表，用于存储已经创建的`Flyweight`对象。当调用`GetFlyweight`方法时，它首先检查`flyweights`哈希表中是否已经存在对应的`Flyweight`对象，如果存在，就直接返回这个对象；如果不存在，就创建一个新的`ConcreteFlyweight`对象，存入`flyweights`哈希表，然后返回这个新创建的对象。

这样，对于同一个`key`，我们只需要创建一次`ConcreteFlyweight`对象，然后就可以多次使用，这就大大减少了需要创建的对象的数量，从而节省了内存，提高了性能，这就体现了享元模式的思想。

需要注意的是，`ConcreteFlyweight`对象的状态应该是内部状态，即不会因为环境的改变而改变的状态；而传入`Operation`方法的`extrinsicstate`参数应该是外部状态，即会因为环境的改变而改变的状态。这样，我们可以共享同一个`ConcreteFlyweight`对象，但是通过改变`extrinsicstate`参数来改变它的行为。这也是享元模式的一个重要特点。

2.代理模式（Proxy Pattern）：

代理模式（Proxy Pattern）是一种结构型设计模式，它通过创建一个代理对象来控制对原始对象的访问[1234]。代理对象充当了客户端和目标对象之间的中介，可以在访问目标对象前后进行一些额外的操作[3]。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口[1][2]。这种类型的设计模式属于结构型模式[1][2]。

例如，假设我们有一个`Image`对象，它有一个`load`方法用于从磁盘加载图像。如果我们直接在每次需要显示图像时都调用`load`方法，可能会导致程序的性能下降。为了解决这个问题，我们可以创建一个`ImageProxy`对象，它也有一个`load`方法，但是在这个方法中，它首先检查图像是否已经被加载，如果已经被加载，就直接返回图像，否则才调用`Image`的`load`方法来加载图像。这样，我们就可以通过`ImageProxy`来控制对`Image`的访问，从而提高程序的性能[1][2]。

代理模式的优点包括职责清晰、高扩展性、智能化[1]。但是，由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢[1]。此外，实现代理模式需要额外的工作，有些代理模式的实现非常复杂[1]。

(1) 设计模式（四）——搞懂什么是代理模式 - 知乎. https://zhuanlan.zhihu.com/p/72644638.
(2) 代理模式 | 菜鸟教程. https://www.runoob.com/design-pattern/proxy-pattern.html.
(3) 【设计模式】代理模式 (简介 | 适用场景 | 优缺点 ....
https://developer.aliyun.com/article/864585.
(4) 代理模式_百度百科. https://baike.baidu.com/item/%E4%BB%A3%E7%90%86%E6%A8%A1%E5%BC%8F/8374046.

```
class Subject {
public:
    virtual void Request() = 0;
};

class RealSubject : public Subject {
public:
    void Request() override {
        std::cout << "RealSubject Request" << std::endl;
    }
};

class Proxy : public Subject {
private:
    RealSubject* realSubject;
public:
    void Request() override {
        if (realSubject == nullptr) {
            realSubject = new RealSubject();
        }
        realSubject->Request();
    }
};
```

这段代码是一个代理模式的C++实现。在这个例子中，`Subject`是一个抽象类，它定义了一个`Request`方法。`RealSubject`是`Subject`的具体实现，它实现了`Request`方法。

`Proxy`也是`Subject`的一个实现，但是它内部有一个`RealSubject`的指针。在`Proxy`的`Request`方法中，它首先检查`RealSubject`是否已经被创建，如果没有被创建，就创建一个新的`RealSubject`，然后调用`RealSubject`的`Request`方法。

这样，`Proxy`就充当了客户端和`RealSubject`之间的中介，可以在访问`RealSubject`前后进行一些额外的操作。这就体现了代理模式的思想。

需要注意的是，这个例子中的`Proxy`并没有释放它创建的`RealSubject`对象，这可能会导致内存泄漏。在实际使用时，我们应该在适当的地方释放已经创建的对象。

3.责任链模式（Chain of Responsibility Pattern）：

责任链模式（Chain of Responsibility Pattern）是一种行为型设计模式，它创建了一个接收者对象的链[1][2][4]。这种模式的主要目标是将请求的发送者和接收者解耦[1][2][4]。在这种模式中，通常每个接收者都包含对另一个接收者的引用[1][2][4]。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推[1][2][4]。

例如，假设我们有一个请假审批的场景，员工请假小于或等于2天，班主任可以批准；小于或等于7天，系主任可以批准；小于或等于10天，院长可以批准；其他情况不予批准[2]。这个场景就适合使用责任链模式实现。首先，定义一个处理者类（Handler），它是抽象处理者，包含了一个指向下一位处理者的指针next和一个处理请求的抽象处理方法handleRequest。然后，定义班主任类（ClassAdviser）、系主任类（DepartmentHead）和院长类（Dean），它们是抽象处理者的子类，是具体处理者，必须根据自己的权力去实现父类的handleRequest方法，如果无权处理就将请求交给下一个处理者，直到最后[2]。

责任链模式的优点包括降低了对象之间的耦合度，增强了系统的可扩展性，增强了给对象指派职责的灵活性，简化了对象之间的连接[1][2][4]。但是，它不能保证每个请求一定被接收，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[1][2][4]。
(1) 责任链模式 | 菜鸟教程. https://www.runoob.com/design-pattern/chain-of-responsibility-pattern.html.
(2) 责任链模式（职责链模式）详解 - 知乎. https://zhuanlan.zhihu.com/p/94660491.
(3) 33 设计模式——责任链模式 详解 - 知乎. https://zhuanlan.zhihu.com/p/446544507.
(4) 实战：设计模式之责任链设计模式深度解析 - 知乎. https://bing.com/search?q=%e8%b4%a3%e4%bb%bb%e9%93%be%e6%a8%a1%e5%bc%8f.
(5) 责任链模式 - 维基百科，自由的百科全书. https://zh.wikipedia.org/wiki/%E8%B4%A3%E4%BB%BB%E9%93%BE%E6%A8%A1%E5%BC%8F.
(6) 设计模式之责任链模式（Chain of Responsibility）详解及 .... https://www.cnblogs.com/jing99/p/12610095.html.

```
class Handler {
protected:
    Handler* successor;
public:
    void SetSuccessor(Handler* successor) {
        this->successor = successor;
    }
    virtual void HandleRequest(int request) = 0;
};

class ConcreteHandler1 : public Handler {
public:
    void HandleRequest(int request) override {
        if (request >= 0 && request < 10) {
            std::cout << "ConcreteHandler1 handled request " << request <<
std::endl;
        } else if (successor != nullptr) {
            successor->HandleRequest(request);
        }
    }
};
```

```
class ConcreteHandler2 : public Handler {
public:
    void HandleRequest(int request) override {
        if (request >= 10 && request < 20) {
            std::cout << "ConcreteHandler2 handled request " << request <<
std::endl;
        } else if (successor != nullptr) {
            successor->HandleRequest(request);
        }
    }
};
```

这段代码是一个责任链模式的C++实现。在这个例子中，`Handler`是一个抽象类，它定义了一个`HandleRequest`方法和一个`SetSuccessor`方法。`HandleRequest`方法是用于处理请求的，`SetSuccessor`方法是用于设置下一个处理者的。

`ConcreteHandler1`和`ConcreteHandler2`是`Handler`的具体实现，它们都实现了`HandleRequest`方法。在`HandleRequest`方法中，它们首先检查是否可以处理当前的请求，如果可以处理，就处理请求并输出一条消息；如果不能处理，就将请求传递给下一个处理者。

这样，当一个请求来到时，它会被一系列的处理者按照顺序处理，每个处理者都有机会处理这个请求，直到找到一个可以处理它的处理者为止。这就体现了责任链模式的思想。

需要注意的是，这个例子中的责任链是静态的，也就是说，责任链的结构在程序运行时是不会改变的。在实际使用时，我们可能需要一个动态的责任链，可以在程序运行时添加或删除处理者。

```
#include <iostream>
#include <vector>

class Handler {
public:
    virtual bool handle(int request) = 0;
};

class ConcreteHandler1 : public Handler {
public:
    bool handle(int request) override {
        if (request >= 0 && request < 10) {
            std::cout << "ConcreteHandler1 handled request " << request <<
std::endl;
            return true;
        }
        return false;
    }
};

class ConcreteHandler2 : public Handler {
public:
    bool handle(int request) override {
        if (request >= 10 && request < 20) {
            std::cout << "ConcreteHandler2 handled request " << request <<
std::endl;
            return true;
        }
        return false;
```

```
    }
};

class DynamicChain {
private:
    std::vector<Handler*> handlers;
public:
    void addHandler(Handler* handler) {
        handlers.push_back(handler);
    }

    void handle(int request) {
        for (auto handler : handlers) {
            if (handler->handle(request)) {
                break;
            }
        }
    }
};

int main() {
    DynamicChain chain;
    chain.addHandler(new ConcreteHandler1());
    chain.addHandler(new ConcreteHandler2());

    chain.handle(5);
    chain.handle(15);

    return 0;
}
```

在这个例子中，DynamicChain类维护了一个处理者的列表。当一个请求来到时，它会按照处理者的添加顺序依次调用处理者的handle方法，直到找到一个可以处理该请求的处理者为止。这就体现了责任链模式的思想。

需要注意的是，这个例子中的责任链是动态的，也就是说，我们可以在程序运行时动态地添加或删除处理者。这增加了责任链的灵活性，使得我们可以根据需要来调整责任链的结构

4.命令模式（Command Pattern）：

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式[124]。在命令模式中，请求以命令的形式包裹在对象中，并传给调用对象[124]。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令[124]。

命令模式的主要优点是降低了系统的耦合度[124]。它将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化[124]。这样，调用者和接收者之间的耦合关系就被解开了[124]。

例如，假设我们有一个遥控器（调用者），它可以控制电视（接收者）。我们可以为遥控器创建一个“打开电视”的命令和一个“关闭电视”的命令。当用户按下遥控器的按钮时，遥控器就会执行相应的命令，从而控制电视[124]。

命令模式的一个重要应用是实现“撤销”操作[124]。我们可以将已经执行的命令存储在一个历史记录中。当用户选择“撤销”操作时，我们就从历史记录中取出最后一个命令，并执行它的“撤销”方法[124]。
(1) 命令模式 | 菜鸟教程. https://www.runoob.com/design-pattern/command-pattern.html.
(2) 深入理解设计模式（十）： 命令模式 - 一指流砂 .... https://www.cnblogs.com/xuwendong/p/9814421.html.

(3) 重学设计模式（三、设计模式-命令模式） - 知乎. https://zhuanlan.zhihu.com/p/530746979.
(4) 命令模式（Command模式）详解-CSDN博客. https://bing.com/search?q=%e5%91%bd%e4%bb%a4%e6%a8%a1%e5%bc%8f.
(5) undefined. http://www.cnblogs.com/xuwendong/.

```
class Command {
public:
    virtual void Execute() = 0;
};

class Receiver {
public:
    void Action() {
        std::cout << "Receiver action" << std::endl;
    }
};

class ConcreteCommand : public Command {
private:
    Receiver* receiver;
public:
    ConcreteCommand(Receiver* receiver) {
        this->receiver = receiver;
    }
    void Execute() override {
        receiver->Action();
    }
};

class Invoker {
private:
    Command* command;
public:
    void SetCommand(Command* command) {
        this->command = command;
    }
    void ExecuteCommand() {
        command->Execute();
    }
};
```

这段代码是一个命令模式的C++实现。在这个例子中，`Command`是一个抽象类，它定义了一个`Execute`方法。`ConcreteCommand`是`Command`的具体实现，它实现了`Execute`方法。

`Receiver`是一个类，它有一个`Action`方法。在`ConcreteCommand`的`Execute`方法中，它调用了`Receiver`的`Action`方法。

`Invoker`是一个类，它有一个`Command`的指针和一个`ExecuteCommand`方法。在`ExecuteCommand`方法中，它调用了`Command`的`Execute`方法。

这样，当我们调用`Invoker`的`ExecuteCommand`方法时，它就会执行`Command`的`Execute`方法，从而触发`Receiver`的`Action`方法。这就体现了命令模式的思想。

需要注意的是，这个例子中的`Invoker`并没有释放它持有的`Command`对象，这可能会导致内存泄漏。在实际使用时，我们应该在适当的地方释放已经创建的对象。

如何撤销

撤销操作在编程中通常是通过命令模式来实现的[3]。命令模式可以将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化[3]。

在实现撤销操作时，通常会有一个历史记录来存储已经执行的命令[1][2]。每个命令都有一个执行和撤销的方法[1][2]。当执行一个命令时，将它添加到历史记录中[1][2]。当执行撤销操作时，从历史记录中取出最后一个命令，并调用它的撤销方法[1][2]。

以下是一个简单的撤销操作的C++代码示例：

```
#include <iostream>
#include <stack>

class Command {
public:
    virtual void execute() = 0;
    virtual void unexecute() = 0;
};

class History {
private:
    std::stack<Command*> history;
public:
    void push(Command* cmd) {
        cmd->execute();
        history.push(cmd);
    }

    void undo() {
        if (!history.empty()) {
            Command* cmd = history.top();
            cmd->unexecute();
            history.pop();
        }
    }
};

class ConcreteCommand : public Command {
private:
    int oldValue, newValue;
    int* receiver;
public:
    ConcreteCommand(int* receiver, int newValue) : receiver(receiver),
newValue(newValue), oldValue(*receiver) {}

    void execute() override {
        *receiver = newValue;
    }

    void unexecute() override {
        *receiver = oldValue;
    }
};

int main() {
```

```
    int value = 0;
    History history;

    history.push(new ConcreteCommand(&value, 1));
    std::cout << "value: " << value << std::endl;  // Output: value: 1

    history.push(new ConcreteCommand(&value, 2));
    std::cout << "value: " << value << std::endl;  // Output: value: 2

    history.undo();
    std::cout << "value: " << value << std::endl;  // Output: value: 1

    return 0;
}
```

在这个例子中，`Command`是一个抽象类，它定义了`execute`和`unexecute`两个方法。`ConcreteCommand`是`Command`的一个具体实现，它在`execute`方法中执行命令，在`unexecute`方法中撤销命令。

`History`类维护了一个命令的历史记录。当执行一个命令时，将它添加到历史记录中。当执行撤销操作时，从历史记录中取出最后一个命令，并调用它的`unexecute`方法。
(1) 各类软件的『撤销上一步』功能，在编程中是如何 .... https://www.zhihu.com/question/340978281.
(2) 撤消、恢复或重复操作 - Microsoft 支持. https://support.microsoft.com/zh-cn/office/%E6%92%A4%E6%B6%88-%E6%81%A2%E5%A4%8D%E6%88%96%E9%87%8D%E5%A4%8D%E6%93%8D%E4%BD%9C-84bdb9bc-4e23-4f06-ba78-f7b893eb2d28.
(3) 如何优雅的实现撤销和回退功能 - CSDN博客. https://blog.csdn.net/minminaya/article/details/123678029.

区别

命令模式和责任链模式都是行为型设计模式，但它们的目标和使用场景有所不同。

命令模式的主要目标是将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。在命令模式中，请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。命令模式的一个重要应用是实现"撤销"操作。

责任链模式的主要目标是将请求的发送者和接收者解耦。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

总的来说，命令模式主要用于封装请求，使得发送者不需要知道接收者的任何信息，只需要知道如何发出请求即可。而责任链模式则主要用于在多个对象之间分配请求，使得发送者不需要知道哪个对象会处理该请求，只需要将请求发送到责任链上即可。

5.迭代器模式（Iterator Pattern）：

迭代器模式（Iterator Pattern）是一种设计模式，它在 Java 和 .Net 编程环境中非常常用[1][2][4]。这种模式用于顺序访问集合对象的元素，而不需要知道集合对象的底层表示[1][2][4]。

在迭代器模式中，通常会有一个迭代器接口（Iterator），它定义了用于遍历元素的方法，如`hasNext`（判断是否还有下一个元素）和`next`（获取下一个元素）[2]。然后，会有一个具体的迭代器类（Concrete Iterator）实现这个接口，它包含了遍历集合所必要的信息，如当前元素的位置[2]。

此外，还会有一个聚合接口（Aggregate），它定义了获取迭代器的方法[2]。具体的聚合类（Concrete Aggregate）会实现这个接口，它包含了一组元素，并提供一个方法来获取一个可以遍历这些元素的迭代器[2]。

迭代器模式的优点包括：

1. 它支持以不同的方式遍历一个聚合对象[1]。
2. 迭代器简化了聚合类[1]。
3. 在同一个聚合上可以有多个遍历[1]。
4. 在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码[1]。

缺点是由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性[1]。

(1) 迭代器模式 | 菜鸟教程. https://www.runoob.com/design-pattern/iterator-pattern.html.
(2) 一次性搞懂设计模式--迭代器模式 - 知乎. https://zhuanlan.zhihu.com/p/537080924.
(3) 迭代器模式（Iterator Pattern）总结 - 知乎. https://zhuanlan.zhihu.com/p/86408779.
(4) 迭代器模式（Iterator Pattern）总结 - 知乎. https://bing.com/search?q=%e8%bf%ad%e4%bb%a3%e5%99%a8%e6%a8%a1%e5%bc%8f.

```
class Iterator {
public:
    virtual bool HasNext() = 0;
    virtual int Next() = 0;
};

class ConcreteIterator : public Iterator {
private:
    std::vector<int> data;
    int index;
public:
    ConcreteIterator(const std::vector<int>& data) : data(data), index(0) {}
    bool HasNext() override {
        return index < data.size();
    }
    int Next() override {
        return data[index++];
    }
};

class Aggregate {
public:
    virtual Iterator* CreateIterator() = 0;
};

class ConcreteAggregate : public Aggregate {
private:
    std::vector<int> data;
public:
    ConcreteAggregate(const std::vector<int>& data) : data(data) {}
    Iterator* CreateIterator() override {
        return new ConcreteIterator(data);
    }
};
```

这段代码是一个迭代器模式的C++实现。在这个例子中，`Iterator`是一个抽象类，它定义了`HasNext`和`Next`两个方法。`ConcreteIterator`是`Iterator`的一个具体实现，它实现了`HasNext`和`Next`方法。

`Aggregate`是一个抽象类，它定义了`CreateIterator`方法。`ConcreteAggregate`是`Aggregate`的一个具体实现，它实现了`CreateIterator`方法。

在这个例子中，`ConcreteIterator`内部有一个`data`向量和一个`index`索引。`HasNext`方法用于判断是否还有下一个元素，`Next`方法用于获取下一个元素并将索引向后移动。

`ConcreteAggregate`内部有一个`data`向量，`CreateIterator`方法用于创建一个可以遍历`data`的`ConcreteIterator`。

这样，我们就可以通过`Aggregate`的`CreateIterator`方法来获取一个迭代器，然后使用这个迭代器来遍历`Aggregate`中的元素。这就体现了迭代器模式的思想。

需要注意的是，这个例子中的`ConcreteAggregate`并没有释放它创建的`ConcreteIterator`对象，这可能会导致内存泄漏。在实际使用时，我们应该在适当的地方释放已经创建的对象。

增加新的迭代器和聚合

在迭代器模式中，如果你想增加新的迭代器和聚合，你可以按照以下步骤进行：

1. **创建新的迭代器**：首先，你需要创建一个新的迭代器类，这个类需要实现`Iterator`接口，并实现`HasNext`和`Next`两个方法。这两个方法的具体实现取决于你的需求，例如，你可能需要遍历一个数组、一个链表，或者其他的数据结构。

```
class NewIterator : public Iterator {
private:
    // Your data structure here
public:
    bool HasNext() override {
        // Your implementation here
    }
    int Next() override {
        // Your implementation here
    }
};
```

1. **创建新的聚合**：然后，你需要创建一个新的聚合类，这个类需要实现`Aggregate`接口，并实现`CreateIterator`方法。在`CreateIterator`方法中，你需要返回一个新创建的迭代器对象。

```
class NewAggregate : public Aggregate {
private:
    // Your data structure here
public:
    Iterator* CreateIterator() override {
        return new NewIterator(/* Your data here */);
    }
};
```

6.观察者模式（Observer Pattern）：

观察者模式（Observer Pattern）是一种设计模式，它定义了一种一对多的依赖关系，当一个对象的状态发生改变时，其所有依赖者都会收到通知并自动更新[1][2][4]。这种模式通常被用来实现事件处理系统[4]。

在观察者模式中，有一个称作“主题”的对象和若干个称作“观察者”的对象[5]。“主题”和“观察者”间是一种一对多的依赖关系，当“主题”的状态发生变化时，所有“观察者”都得到通知[5]。

例如，假设我们有一个新闻网站（主题），用户（观察者）可以在网站上订阅他们感兴趣的新闻类别。当网站发布新的新闻时，所有订阅了相关类别的用户都会收到通知[1][2][4]。

观察者模式的优点包括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[1][2][4]。它还可以建立一套触发机制[1][2][4]。

但是，如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间[1][2][4]。此外，如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃[1][2][4]。

(1) 观察者模式 | 菜鸟教程. https://www.runoob.com/design-pattern/observer-pattern.html.
(2) 观察者模式（Observer模式）详解 - 知乎. https://zhuanlan.zhihu.com/p/77275289.
(3) 观察者模式_百度百科. https://baike.baidu.com/item/%E8%A7%82%E5%AF%9F%E8%80%85%E6%A8%A1%E5%BC%8F/5881786.
(4) 深入理解设计模式（八）： 观察者模式 - 一指流砂 .... https://www.cnblogs.com/xuwendong/p/9814417.html.
(5) 【简易设计模式05】观察者模式（监听模式） - 知乎专栏. https://bing.com/search?q=%e8%a7%82%e5%af%9f%e8%80%85%e6%a8%a1%e5%bc%8f.

```
class Observer {
public:
    virtual void Update(int value) = 0;
};

class ConcreteObserver : public Observer {
public:
    void Update(int value) override {
        std::cout << "ConcreteObserver: " << value << std::endl;
    }
};

class Subject {
private:
    std::list<Observer*> observers;
    int state;
public:
    void Attach(Observer* observer) {
        observers.push_back(observer);
    }
    void Detach(Observer* observer) {
        observers.remove(observer);
    }
    void Notify() {
        for (Observer* observer : observers) {
            observer->Update(state);
        }
    }
    void SetState(int state) {
        this->state = state;
        Notify();
    }
};
```

7.状态模式（State Pattern）：

状态模式（State Pattern）是一种设计模式，它主要用来解决对象在多种状态转换时，需要对外输出不同的行为的问题[3]。状态和行为是一一对应的，状态之间可以相互转换[3]。当一个对象的内在状态改变时，允许改变其行为，这个对象看起来像是改变了其类[1][2][4]。

例如，假设我们有一个订单系统，订单有待付款，已付款待发货，待收货，待评价, 已完成等状态[3]。在不同的状态下，订单的行为是不同的。例如，在待付款状态下，用户可以取消订单，但在已付款待发货状态下，用户就不能取消订单了[3]。

在状态模式中，通常会有一个状态接口（State），它定义了所有的状态和行为[1][2][4]。然后，会有一些具体的状态类（Concrete State），它们实现了状态接口，并定义了在该状态下的具体行为[1][2][4]。此外，还会有一个上下文类（Context），它维护了当前的状态，并提供了改变状态的方法[1][2][4]。

状态模式的优点包括增强了程序的可扩展性，因为我们很容易添加一个新的状态[1][2][4]。它还增强了程序的封装性，每个状态的操作都被封装到了一个状态类中[1][2][4]。但是，状态模式的使用必然会增加系统类和对象的个数[1][2][4]。

(1) 23种设计模式(七)-状态设计模式-腾讯云开发者社区 .... https://cloud.tencent.com/developer/article/1846042.
(2) 秒懂设计模式之状态模式（State Pattern） - 知乎专栏. https://zhuanlan.zhihu.com/p/369732910.
(3) 状态模式 | 菜鸟教程. https://www.runoob.com/design-pattern/state-pattern.html.
(4) 34 设计模式——状态模式 详解 - 知乎. https://zhuanlan.zhihu.com/p/446938172.
(5) 设计模式之状态模式（State）详解及代码示例 - kosamino .... https://www.cnblogs.com/jing99/p/12610105.html.

```
class State {
public:
    virtual void Handle() = 0;
};

class ConcreteStateA : public State {
public:
    void Handle() override {
        std::cout << "ConcreteStateA" << std::endl;
    }
};

class ConcreteStateB : public State {
public:
    void Handle() override {
        std::cout << "ConcreteStateB" << std::endl;
    }
};

class Context {
private:
    State* state;
public:
    Context(State* state) : state(state) {}
    void SetState(State* state) {
        this->state = state;
    }
    void Request() {
        state->Handle();
    }
```

```
};
```

8.策略模式（Strategy Pattern）：

策略模式（Strategy Pattern）是一种行为型设计模式，它定义了一系列的算法，并将每个算法封装在独立的类中，使得它们可以互相替换[1][3][4]。这种模式的主要目标是将算法的使用和算法的实现分离开来[1][3][4]。

在策略模式中，通常会有一个策略接口，这个接口定义了所有策略类需要实现的方法[1][3][4]。然后，会有一些具体的策略类，它们实现了策略接口，并提供了具体的算法实现[1][3][4]。

此外，还会有一个上下文类，它维护了一个对策略对象的引用，负责将客户端请求委派给具体的策略对象执行[1][3][4]。上下文类可以通过依赖注入、简单工厂等方式来获取具体策略对象[1][3][4]。

策略模式的优点包括算法可以自由切换，避免使用多重条件判断，扩展性良好[1][3][4]。缺点是策略类会增多，所有策略类都需要对外暴露[1][3][4]。
(1) 策略模式 | 菜鸟教程. https://www.runoob.com/design-pattern/strategy-pattern.html.
(2) 策略模式详解 - 知乎. https://bing.com/search?q=%e7%ad%96%e7%95%a5%e6%a8%a1%e5%bc%8f.
(3) Java 设计模式：策略模式 (Strategy) - 知乎. https://zhuanlan.zhihu.com/p/638802128.
(4) 【设计模式系列】策略模式看这篇就够了 - 掘金. https://juejin.cn/post/7030976391596212255.
(5) 【精选】一文搞懂策略模式(优化策略模式完全消除if .... https://blog.csdn.net/bookssea/article/details/117043820.

状态模式和策略模式的区别

策略模式和状态模式在结构上非常相似，但它们的意图和使用场景有所不同[1][2]。

策略模式封装了一组相关算法，它允许客户端在运行时使用可互换的行为[1][2]。策略模式定义了对象"怎么做"的部分。例如，排序对象怎么对数据排序[2]。策略模式的改变由客户端完成[2]。

状态模式帮助一个类在不同的状态显示不同的行为[1][2]。状态模式封装了对象的状态[1][2]。状态模式定义了对象"是什么"和"什么时候做"的部分。例如，对象处于什么状态，什么时候处在某个特定的状态[2]。状态模式中很好的定义了状态转移的次序；而策略模式并无此需要：客户端可以自由的选择任何策略[2]。状态模式的改变，由上下文或状态自己完成[2]。
(1) 策略模式 VS 状态模式 | 菜鸟教程. https://www.runoob.com/w3cnote/state-vs-strategy.html.
(2) 设计模式之：状态模式和策略模式的区别 - 知乎. https://zhuanlan.zhihu.com/p/142056182.
(3) 聊一聊策略模式和状态模式 - 知乎. https://zhuanlan.zhihu.com/p/138267848.

```
class Strategy {
public:
    virtual void AlgorithmInterface() = 0;
};

class ConcreteStrategyA : public Strategy {
public:
    void AlgorithmInterface() override {
        std::cout << "ConcreteStrategyA" << std::endl;
    }
};

class ConcreteStrategyB : public Strategy {
public:
    void AlgorithmInterface() override {
        std::cout << "ConcreteStrategyB" << std::endl;
    }
};
```

```
class Context {
private:
    Strategy* strategy;
public:
    Context(Strategy* strategy) : strategy(strategy) {}
    void SetStrategy(Strategy* strategy) {
        this->strategy = strategy;
    }
    void ContextInterface() {
        strategy->AlgorithmInterface();
    }
};
```

9.模板模式（Template Pattern）：

模板模式（Template Pattern）是一种设计模式，它在一个抽象类公开定义了执行它的方法的方式/模板[1][2][4]。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行[1][2][4]。这种类型的设计模式属于行为型模式[1][2][4]。

模板模式定义了一个操作中的算法的骨架，而将一些步骤延迟到子类中[1][2][4]。模板方法使得子类可以在不改变一个算法的结构的情况下，重新定义该算法的某些特定步骤[1][2][4]。

例如，假设我们有一个保险系统，保险分为很多种类，比如：产险、寿险、养老险、健康险、车险等[2]。我们作为后台开发者在设计开发这些保险产品上线的时候，首先要分析下这几个险种购买流程的共同点[2]。这些保险在产品购买时的流程如下（举个例子，实际情况有所差异）[2]：

1. 健康险、产险、养老险、寿险、车险：
2. 保费试算
3. 健康告知（车险没有，是检验车牌）
4. 核保
5. 支付 (统一的)
6. 承保

可以看到，核心的购买流程是非常相似的，但是不同的险种具体实现是不一样的，比如健康险和产险他们健康告知的内容就是不一样的[2]。如果直接搞定几个对应的类实现，肯定会存在很多重复的代码，如果让你来设计，肯定第一个想法就是定义一个超类，将共同的方法放在其中，然后购买 createOrder（）方法由于在子类（车险和产寿养健）中的流程调用方法是不一样的，所以定义成抽象，然后例子中的健康告知等方法是不公用的所以放到具体的子类中[2]。

模板模式的优点包括封装不变部分，扩展可变部分[1][2][4]。提取公共代码，便于维护[1][2][4]。行为由父类控制，子类实现[1][2][4]。缺点是每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大[1][2][4]。

(1) 模板模式 | 菜鸟教程. https://www.runoob.com/design-pattern/template-pattern.html.
(2) 最有用的设计模式之一-模板方法 - 知乎. https://zhuanlan.zhihu.com/p/163692932.
(3) 设计模式最佳套路4 —— 愉快地使用模板模式-阿里云 .... https://developer.aliyun.com/article/786380.
(4) Java设计模式——模板方法模式【Template Method Pattern .... https://bing.com/search?q=%e6%a8%a1%e6%9d%bf%e6%a8%a1%e5%bc%8f.
(5) java设计模式之模板模式 - 知乎. https://zhuanlan.zhihu.com/p/64477290.

```
class AbstractClass {
public:
    void TemplateMethod() {
```

```
        PrimitiveOperation1();
        PrimitiveOperation2();
    }
    virtual void PrimitiveOperation1() = 0;
    virtual void PrimitiveOperation2() = 0;
};

class ConcreteClass : public AbstractClass {
public:
    void PrimitiveOperation1() override {
        std::cout << "ConcreteClass PrimitiveOperation1" << std::endl;
    }
    void PrimitiveOperation2() override {
        std::cout << "ConcreteClass PrimitiveOperation2" << std::endl;
    }
};
```

10.访问者模式（Visitor Pattern）：

访问者模式（Visitor Pattern）是一种将算法与对象结构分离的设计模式[1][2][4]。这种模式的基本想法是：首先我们拥有一个由许多对象构成的对象结构，这些对象的类都拥有一个`accept`方法用来接受访问者对象[1][2][4]。访问者是一个接口，它拥有一个`visit`方法，这个方法对访问到的对象结构中不同类型的元素作出不同的反应[1][2][4]。在对象结构的一次访问过程中，我们遍历整个对象结构，对每一个元素都实施`accept`方法，在每一个元素的`accept`方法中调用访问者的`visit`方法，从而使访问者得以处理对象结构的每一个元素[1][2][4]。

访问者模式的优点包括：

1. 符合单一职责原则[1][2][4]。
2. 优秀的扩展性[1][2][4]。
3. 灵活性[1][2][4]。

缺点包括：

1. 具体元素对访问者公布细节，违反了迪米特原则[1][2][4]。
2. 具体元素变更比较困难[1][2][4]。
3. 违反了依赖倒置原则，依赖了具体类，没有依赖抽象[1][2][4]。

使用场景包括：

1. 对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作[1][2][4]。
2. 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类[1][2][4]。

(1) 访问者模式 | 菜鸟教程. https://www.runoob.com/design-pattern/visitor-pattern.html.
(2) 秒懂设计模式之访问者模式（Visitor Pattern） - 知乎专栏. https://zhuanlan.zhihu.com/p/380161731.
(3) java设计模式之访问者模式 - 知乎. https://zhuanlan.zhihu.com/p/65652456.
(4) 【设计模式】详解访问者（Visitor）模式-读完这篇你就 .... https://bing.com/search?q=%e8%ae%bf%e9%97%ae%e8%80%85%e6%a8%a1%e5%bc%8f.
(5) 访问者模式 - 维基百科，自由的百科全书. https://zh.wikipedia.org/wiki/%E8%AE%BF%E9%97%AE%E8%80%85%E6%A8%A1%E5%BC%8F.

```
class Element {
public:
```

```
    virtual void Accept(class Visitor* visitor) = 0;
};

class ConcreteElementA : public Element {
public:
    void Accept(Visitor* visitor) override;
};

class ConcreteElementB : public Element {
public:
    void Accept(Visitor* visitor) override;
};

class Visitor {
public:
    virtual void VisitConcreteElementA(ConcreteElementA* concreteElementA) = 0;
    virtual void VisitConcreteElementB(ConcreteElementB* concreteElementB) = 0;
};

class ConcreteVisitor1 : public Visitor {
public:
    void VisitConcreteElementA(ConcreteElementA* concreteElementA) override {
        std::cout << "ConcreteVisitor1 visited ConcreteElementA" << std::endl;
    }
    void VisitConcreteElementB(ConcreteElementB* concreteElementB) override {
        std::cout << "ConcreteVisitor1 visited ConcreteElementB" << std::endl;
    }
};

class ConcreteVisitor2 : public Visitor {
public:
    void VisitConcreteElementA(ConcreteElementA* concreteElementA) override {
        std::cout << "ConcreteVisitor2 visited ConcreteElementA" << std::endl;
    }
    void VisitConcreteElementB(ConcreteElementB* concreteElementB) override {
        std::cout << "ConcreteVisitor2 visited ConcreteElementB" << std::endl;
    }
};

void ConcreteElementA::Accept(Visitor* visitor) {
    visitor->VisitConcreteElementA(this);
}

void ConcreteElementB::Accept(Visitor* visitor) {
    visitor->VisitConcreteElementB(this);
}
```

11.单例模式（Singleton Pattern）：

单例模式是一种常用的软件设计模式，属于创建型模式的一种。在应用这个模式时，单例对象的类必须保证只有一个实例存在[3]。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象[2]。

在开发过程中，很多时候一个类我们希望它只创建一个对象，比如：线程池、缓存、网络请求等。当这类对象有多个实例时，程序就可能会出现异常，比如：程序出现异常行为、得到的结果不一致等。这时候就应该使用单例模式。单例主要有这两个优点：

1. 提供了对唯一实例的受控访问。
2. 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能[1]。

实现单例模式主要有以下几个关键点：

- 在类加载的期间，所以，instance 实例的创建是线程安全的。
- 不过，这样的实现方式懒汉式相对于饿汉式的优势是但它的缺点也很明显，getInstance 使用了实现线程同步，导致这个方法的并发很低，每次调用都会频繁的枷锁、释放锁，会导致性能瓶颈。
- 饿汉式不能延时加载，懒汉式有性能问题，而双重检测方式既支持延迟加载、又支持高并发的单例实现方式[1]。
  (1) 单例模式 - 维基百科，自由的百科全书. https://zh.wikipedia.org/wiki/%E5%8D%95%E4%BE%8B%E6%A8%A1%E5%BC%8F.
  (2) 单例模式 | 菜鸟教程. https://www.runoob.com/design-pattern/singleton-pattern.html.
  (3) 单例模式详细解析 - 知乎. https://zhuanlan.zhihu.com/p/160835278.
  (4) 单例模式解释_持久化对象使用单例模式-CSDN博客. https://blog.csdn.net/sanjay_f/article/details/39353173.

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}
public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};
Singleton* Singleton::instance = nullptr;
```

12.建造者模式（Builder Pattern）：

建造者模式是一种设计模式，它将一个复杂对象的构建过程与它的表示分离，使得同样的构建过程可以创建不同的表示[14]。这种模式涉及到一个 Builder 类，该类会一步一步构造最终的对象[1]。这种设计模式属于创建型模式，它提供了一种创建对象的最佳方式[1]。

建造者模式的主要优点是：

1. 分离构建过程和表示，使得构建过程更加灵活，可以构建不同的表示[1]。
2. 可以更好地控制构建过程，隐藏具体构建细节[1]。
3. 代码复用性高，可以在不同的构建过程中重复使用相同的建造者[1]。

建造者模式的主要缺点是：

1. 如果产品的属性较少，建造者模式可能会导致代码冗余[1]。
2. 建造者模式增加了系统的类和对象数量[1]。

建造者模式在创建复杂对象时非常有用，特别是当对象的构建过程涉及多个步骤或参数时[1]。它可以提供更好的灵活性和可维护性，同时使得代码更加清晰可读[1]。

(1) 建造者模式 | 菜鸟教程. https://www.runoob.com/design-pattern/builder-pattern.html.
(2) 建造者模式*百度百科*. *https://baike.baidu.com/item/%E5%BB%BA%E9%80%A0%E8%80%85%E6%A8%A1%E5%BC%8F/3229729*.
(3) 建造者模式——结合案例，深入浅出 - 知乎. *https://zhuanlan.zhihu.com/p/143101326*.
*(4) 建筑者模式【图解，清晰思路】代码演示实现*建造者 .... https://blog.csdn.net/qq_40965479/article/details/107881162.
(5) 建造者模式详解附有代码案例分析（包含建造者模式 .... https://blog.csdn.net/hyyyya/article/details/108778482.

```
class Product {
public:
    std::vector<std::string> parts;
    void Add(std::string part) {
        parts.push_back(part);
    }
};

class Builder {
public:
    virtual void BuildPartA() = 0;
    virtual void BuildPartB() = 0;
    virtual Product* GetResult() = 0;
};

class ConcreteBuilder : public Builder {
private:
    Product* product;
public:
    ConcreteBuilder() {
        product = new Product();
    }
    void BuildPartA() override {
        product->Add("PartA");
    }
    void BuildPartB() override {
        product->Add("PartB");
    }
    Product* GetResult() override {
        return product;
    }
};

class Director {
public:
    void Construct(Builder* builder) {
        builder->BuildPartA();
        builder->BuildPartB();
    }
};
```

13.原型模式（Prototype Pattern）：

原型模式是一种创建型设计模式，用于通过复制现有对象来创建新对象[3]。在创建对象成本高昂或复杂时，复制现有对象并修改它以满足我们的需要会更有效[3]。

原型模式的主要优点是：

1. 性能提高[2]。
2. 逃避构造函数的约束[2]。

原型模式的主要缺点是：

1. 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候[2]。
2. 必须实现 Cloneable 接口[2]。

原型模式的使用场景包括：

1. 资源优化场景[2]。
2. 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等[2]。
3. 性能和安全要求的场景[2]。
4. 通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式[2]。
5. 一个对象多个修改者的场景[2]。
6. 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用[2]。

(1) Python原型模式简述 - 知乎. https://zhuanlan.zhihu.com/p/591683145.
(2) 原型模式 | 菜鸟教程. https://www.runoob.com/design-pattern/prototype-pattern.html.
(3) 设计模式之什么是原型模式 - 知乎. https://zhuanlan.zhihu.com/p/343917614.
(4) C++ 设计模式（五）：原型 - 知乎. https://zhuanlan.zhihu.com/p/365222623.
(5) 原型模型概述及说明情况_原型说明书百科-CSDN博客. https://blog.csdn.net/aerors321/article/details/122004966.

```
class Prototype {
public:
    virtual Prototype* Clone() = 0;
};

class ConcretePrototype : public Prototype {
public:
    Prototype* Clone() override {
        return new ConcretePrototype(*this);
    }
};
```

14.适配器模式（Adapter Pattern）：

适配器模式是一种结构型设计模式，它的主要目标是使原本接口不兼容的对象可以一起工作[1][2][4]。适配器模式有时也被称为包装器模式[1][4]。

适配器模式的主要思想是将一个类的接口转换成客户希望的另一个接口[1][2][3][4]。这个模式可以使得原本由于接口不兼容而不能一起工作的类可以一起工作[1][2][3][4]。适配器模式的做法是将类自己的接口包裹在一个已存在的类中[1]。

适配器模式有两种类型：类适配器模式和对象适配器模式[1][3]。

- 类适配器模式：基于继承。让适配器实现目标接口，并且继承适配者，这样适配器就具备目标接口和适配者的特性，就可以将两者进行转化[3][5]。

- 对象适配器模式：基于组合。让适配器实现目标接口，然后内部持有适配者实例，然后在目标接口规定的方法内转换适配者[3][5]。

适配器模式的应用场景包括：

- 系统需要使用现有的类，而此类的接口不符合系统的需要，即接口不兼容[3]。

(1) 适配器模式 - 维基百科，自由的百科全书. https://zh.wikipedia.org/wiki/%E9%80%82%E9%85%8D%E5%99%A8%E6%A8%A1%E5%BC%8F.
(2) 八、适配器模式与桥接模式详解 - 知乎. https://zhuanlan.zhihu.com/p/112497289.
(3) 精读《设计模式 - Adapter 适配器模式》 - 知乎. https://zhuanlan.zhihu.com/p/280733597.
(4) 适配器模式（Adapter模式）详解 - 简书. https://bing.com/search?q=%e9%80%82%e9%85%8d%e5%99%a8%e6%a8%a1%e5%bc%8f%e8%a7%a3%e9%87%8a.
(5) C#设计模式笔记之适配器模式 (Adapter Pattern) - 知乎. https://zhuanlan.zhihu.com/p/346509882.

```
class Target {
public:
    virtual void Request() = 0;
};

class Adaptee {
public:
    void SpecificRequest() {
        std::cout << "Adaptee SpecificRequest" << std::endl;
    }
};

class Adapter : public Target {
private:
    Adaptee* adaptee;
public:
    Adapter(Adaptee* adaptee) : adaptee(adaptee) {}
    void Request() override {
        adaptee->SpecificRequest();
    }
};
```

15.装饰器模式（Decorator Pattern）：

装饰器模式是一种设计模式，它允许你在不改变现有对象结构的情况下，动态地给该对象添加新的功能[1]。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装[1]。

装饰器模式的主要优点有：

1. 装饰器是继承的有力补充，比继承灵活，在不改变原有对象的情况下，动态的给一个对象扩展功能，即插即用[3]。
2. 通过使用不同的装饰类及这些装饰类的排列组合，可以实现不同效果[3]。
3. 装饰器模式完全遵守开闭原则[3]。

装饰器模式的主要缺点是：装饰器模式会增加许多子类，过度使用会增加程序的复杂性[2]。

装饰器模式的使用场景包括：

1. 扩展一个类的功能[2]。
2. 动态增加功能，动态撤销[2]。

(1) 装饰器模式 | 菜鸟教程. https://www.runoob.com/design-pattern/decorator-pattern.html.
(2) 24 设计模式——装饰器模式（装饰设计模式）详解 - 知乎. https://bing.com/search?q=%e8%a3%85%e9%a5%b0%e5%99%a8%e6%a8%a1%e5%bc%8f%e8%a7%a3%e9%87%8a.
(3) 24 设计模式——装饰器模式（装饰设计模式）详解 - 知乎. https://zhuanlan.zhihu.com/p/444298983.

```
class Component {
public:
    virtual void Operation() = 0;
};

class ConcreteComponent : public Component {
public:
    void Operation() override {
        std::cout << "ConcreteComponent Operation" << std::endl;
    }
};

class Decorator : public Component {
protected:
    Component* component;
public:
    Decorator(Component* component) : component(component) {}
    void Operation() override {
        component->Operation();
    }
};

class ConcreteDecorator : public Decorator {
public:
    ConcreteDecorator(Component* component) : Decorator(component) {}
    void Operation() override {
        Decorator::Operation();
        AddedBehavior();
    }
    void AddedBehavior() {
        std::cout << "ConcreteDecorator AddedBehavior" << std::endl;
    }
};
```

这段代码是一个装饰器模式的实现。这里有四个类：`Component`、`ConcreteComponent`、`Decorator` 和 `ConcreteDecorator`。

- `Component` 是一个抽象基类，定义了一个名为 `Operation` 的纯虚函数。
- `ConcreteComponent` 是 `Component` 的一个具体实现，它重写了 `Operation` 函数，打印出 "ConcreteComponent Operation"。
- `Decorator` 是一个装饰器基类，它也是 `Component` 的子类，并且包含一个 `Component` 类型的指针 `component`。它的 `Operation` 函数调用了 `component` 的 `Operation` 函数，这样就可以在不改变 `component` 的情况下添加新的行为。
- `ConcreteDecorator` 是 `Decorator` 的一个具体实现，它重写了 `Operation` 函数，在调用 `Decorator` 的 `Operation` 函数后，还调用了一个名为 `AddedBehavior` 的函数，打印出 "ConcreteDecorator AddedBehavior"，这就是添加的新行为。

这样，我们就可以创建一个 `ConcreteComponent` 对象，然后用 `ConcreteDecorator` 来装饰它，添加新的行为，而不需要修改 `ConcreteComponent` 的代码。

为什么不直接在concreteComponent继承

直接在 `ConcreteComponent` 上进行继承确实是一种可能的方式来添加新的行为，但这种方式有一些缺点。

首先，如果我们有很多不同的行为要添加，那么我们需要为每一种行为创建一个新的子类，这会导致类的数量急剧增加，使得代码变得复杂和难以维护。

其次，如果我们想要动态地添加或删除行为，那么使用继承就会变得非常困难，因为继承是在编译时确定的，而不是在运行时。

相比之下，装饰器模式可以在运行时动态地添加或删除行为，而不需要改变原有对象的代码。这使得我们可以在不增加类的数量的情况下，灵活地扩展对象的功能。

总的来说，虽然继承是一种可能的方式来添加新的行为，但装饰器模式在许多情况下会更加灵活和强大。

动态添加或删除行为

使用装饰器模式确实会产生新的子类，但这些子类是装饰器，它们的目的是为了增强或修改原有对象的行为。

当我们说"动态地添加或删除行为"时，我们是指在运行时决定是否以及如何装饰对象，而不是在编译时。这意味着我们可以根据需要来选择是否使用装饰器，以及使用哪种装饰器，而不需要改变原有对象的代码。

例如，假设我们有一个 `Component` 对象，我们可以像这样来装饰它：

```
Component* component = new ConcreteComponent();
component = new ConcreteDecoratorA(component);
component = new ConcreteDecoratorB(component);
```

在这个例子中，我们首先创建了一个 `ConcreteComponent` 对象，然后我们用 `ConcreteDecoratorA` 来装饰它，然后又用 `ConcreteDecoratorB` 来装饰它。这样，我们就动态地给 `component` 对象添加了 `ConcreteDecoratorA` 和 `ConcreteDecoratorB` 的行为。

如果我们想要删除某个行为，我们可以简单地不使用对应的装饰器。例如，如果我们不想要 `ConcreteDecoratorA` 的行为，我们可以像这样来创建和装饰 `component` 对象：

```
Component* component = new ConcreteComponent();
component = new ConcreteDecoratorB(component);
```

这样，`component` 对象就只有 `ConcreteDecoratorB` 的行为了。

总的来说，虽然装饰器模式会产生新的子类，但这些子类提供了一种灵活的方式来动态地添加或删除行为，而不需要改变原有对象的代码。

适配器和装饰器模式的区别

适配器模式和装饰器模式都是常见的设计模式，它们都可以在不修改原有类的代码的情况下，增加或改变类的行为。但是，它们的目标和使用方式有所不同[1][2][3][4][5]。

适配器模式的主要目标是将一个类的接口转换成另一个接口，以满足客户端的需求[1][2][3][4][5]。例如，如果你有一个老的类，它的方法和属性不能满足你的新需求，你就可以创建一个适配器，这个适配器实现了你的新接口，并且内部持有一个老的类的实例，然后在新接口的方法中调用老的类的方法[1][2][3][4][5]。

装饰器模式的主要目标是动态地给一个对象添加新的行为[1][2][3][4][5]。装饰器模式涉及到一个装饰器类，这个类和原有的类实现了同样的接口，然后在接口的方法中调用原有类的方法，然后再添加一些新的行为[1][2][3][4][5]。

总的来说，适配器模式和装饰器模式虽然都是用于类的扩展和变换，但适配器模式更注重接口的转换和功能的实现；而装饰器模式则更注重对象的动态性，能够在运行时动态地为对象添加新的职责或行为[1][2][3][4][5]。

(1) 【模式对比】适配器 VS 装饰器 - 知乎. https://zhuanlan.zhihu.com/p/628505159.
(2) 适配器模式，装饰模式，代理模式异同，别再说看着 .... https://zhuanlan.zhihu.com/p/296319765.
(3) 适配器模式与装饰器模式的区别-腾讯云开发者社区 .... https://cloud.tencent.com/developer/article/1706888.
(4) 适配器模式和装饰模式的区别 - 电源设计应用 - 电子 .... https://www.elecfans.com/article/83/116/2018/20180115615839.html.
(5) 适配器模式和装饰模式 - 掘金. https://juejin.cn/post/6974421304963235877.

游戏开发中使用装饰器模式

以下是一个使用装饰器模式来设计游戏角色的 C++ 代码示例：

```
// 抽象化角色：角色
class Character {
public:
    virtual void Operation() = 0;
};

// 扩展抽象化角色：具体角色
class ConcreteCharacter : public Character {
public:
    void Operation() override {
        std::cout << "ConcreteCharacter Operation" << std::endl;
    }
};

// 实现化角色：属性
class CharacterAttribute : public Character {
protected:
    Character* character;
public:
    CharacterAttribute(Character* character) : character(character) {}
    void Operation() override {
        character->Operation();
    }
};

// 具体实现化角色：具体属性
class AttackPower : public CharacterAttribute {
public:
    AttackPower(Character* character) : CharacterAttribute(character) {}
    void Operation() override {
        std::cout << "AttackPower Operation" << std::endl;
        CharacterAttribute::Operation();
```

```
    }
};

class DefensePower : public CharacterAttribute {
public:
    DefensePower(Character* character) : CharacterAttribute(character) {}
    void Operation() override {
        std::cout << "DefensePower Operation" << std::endl;
        CharacterAttribute::Operation();
    }
};

// 实现化角色：行为
class CharacterBehavior : public Character {
protected:
    Character* character;
public:
    CharacterBehavior(Character* character) : character(character) {}
    void Operation() override {
        character->Operation();
    }
};

// 具体实现化角色：具体行为
class Jump : public CharacterBehavior {
public:
    Jump(Character* character) : CharacterBehavior(character) {}
    void Operation() override {
        std::cout << "Jump Operation" << std::endl;
        CharacterBehavior::Operation();
    }
};

class Dash : public CharacterBehavior {
public:
    Dash(Character* character) : CharacterBehavior(character) {}
    void Operation() override {
        std::cout << "Dash Operation" << std::endl;
        CharacterBehavior::Operation();
    }
};
```

在这个例子中，`Character` 是抽象化角色，`ConcreteCharacter` 是扩展抽象化角色，`CharacterAttribute` 和 `CharacterBehavior` 是装饰器，`AttackPower`、`DefensePower`、`Jump` 和 `Dash` 是具体的装饰器。

16.外观模式（Facade Pattern）：

外观模式是一种设计模式，它提供了一个统一的接口，用来访问子系统中的一群接口[1][2]。这种模式定义了一个高层接口，使得子系统更容易使用[1][2]。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性[1][2]。

外观模式的主要优点是：

1. 降低了访问复杂系统的内部子系统时的复杂度，简化了客户端之间的接口[1][2]。
2. 提高了灵活性[1][2]。

3. 提高了安全性[1][2]。

外观模式的主要缺点是不符合开闭原则，如果要改变东西很麻烦，继承重写都不合适[1][2]。

外观模式的使用场景包括：

1. 为复杂的模块或子系统提供外界访问的模块[1][2]。
2. 子系统相对独立[1][2]。
3. 预防低水平人员带来的风险[1][2]。

(1) 外观模式 | 菜鸟教程. https://www.runoob.com/design-pattern/facade-pattern.html.
(2) 精读《设计模式 - Facade 外观模式》 - 知乎. https://zhuanlan.zhihu.com/p/333225431.
(3) C#设计模式(11)——外观模式（Facade Pattern） - Learning .... https://bing.com/search?q=%e5%a4%96%e8%a7%82%e6%a8%a1%e5%bc%8f%e8%a7%a3%e9%87%8a.

```
class SubSystemA {
public:
    void OperationA() {
        std::cout << "SubSystemA OperationA" << std::endl;
    }
};

class SubSystemB {
public:
    void OperationB() {
        std::cout << "SubSystemB OperationB" << std::endl;
    }
};

class Facade {
private:
    SubSystemA* subSystemA;
    SubSystemB* subSystemB;
public:
    Facade() {
        subSystemA = new SubSystemA();
        subSystemB = new SubSystemB();
    }
    void Operation() {
        subSystemA->OperationA();
        subSystemB->OperationB();
    }
};
```

这段代码是一个外观模式的实现。这里有三个类：`SubSystemA`、`SubSystemB`和`Facade`。

- `SubSystemA`和`SubSystemB`是两个子系统，它们各自有一个操作，分别是`OperationA`和`OperationB`，这两个操作在调用时会打印出 "SubSystemA OperationA" 和 "SubSystemB OperationB"。
- `Facade`是一个外观类，它包含了`SubSystemA`和`SubSystemB`的实例。它有一个`Operation`方法，这个方法会调用`SubSystemA`的`OperationA`方法和`SubSystemB`的`OperationB`方法。

这样，客户端只需要与`Facade`类交互，而不需要直接与`SubSystemA`和`SubSystemB`交互。这就简化了客户端的操作，同时也隐藏了子系统的复杂性。

总的来说，这段代码是一个很好的外观模式的例子，它展示了如何使用外观模式来简化对子系统的操作。

外观模式和模板模式的区别

外观模式和模板方法模式是两种不同类型的设计模式，它们各自有不同的目标和使用场景[1][2][3]。

- **外观模式**属于结构型设计模式，主要目标是简化接口和减少系统间的依赖[1][2][3]。当一个系统的组件或子系统过于复杂时，外观模式可以提供一个简化的接口，使得客户端更容易使用这些功能[1][2][3]。外观模式将多个复杂的子系统封装起来，并暴露出一个简单的统一API，使得客户端可以更容易地使用这些功能而不需要了解其内部实现[2]。
- **模板方法模式**属于行为型设计模式，主要目标是定义一个算法的骨架，而将特定步骤的实现延迟到子类中[1][2][3]。这种模式使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤[1][2][3]。模板方法模式通常用于编写框架或库的核心代码，它提供了一个结构性框架，以便针对具体情况进行扩展[2]。

因此，虽然外观模式和模板方法模式都是设计模式，但它们的应用范围、实现方法和目标等方面有所不同[1][2][3]。

(1) 设计模式解析四 模板方法模式和外观模式 - 简书. https://www.jianshu.com/p/a55872e81bcd.
(2) 外观模式 和 模板方法模式的区别 - CSDN博客. https://blog.csdn.net/zjj_flower/article/details/131570355.
(3) 外观模式与模板方法模式的异同_java外观模式和模板 .... https://bing.com/search?q=%e5%a4%96%e8%a7%82%e6%a8%a1%e5%bc%8f%e5%92%8c%e6%a8%a1%e6%9d%bf%e6%a8%a1%e5%bc%8f%e7%9a%84%e5%8c%ba%e5%88%ab.

17.桥接模式（Bridge Pattern）：

桥接模式是一种结构型设计模式，用于将抽象化与实现化解耦，使得二者可以独立变化[1][2]。这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类，这两种类型的类可被结构化改变而互不影响[1][2]。

桥接模式的主要优点是：

1. 分离抽象接口及其实现部分[1][2]。
2. 提供了系统的可扩展性[1][2]。
3. 实现细节对客户透明[1][2]。

桥接模式的主要缺点是：

1. 桥接模式的引入会增加系统的理解与设计难度[1][2]。

桥接模式的使用场景包括：

1. 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系[1][2]。
2. 对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用[1][2]。

(1) 桥接模式 | 菜鸟教程. https://www.runoob.com/design-pattern/bridge-pattern.html.
(2) 精读《设计模式 - Bridge 桥接模式》 - 知乎. https://zhuanlan.zhihu.com/p/296254090.
(3) 网络桥接模式是什么？ 网络桥接模式和路由模式的 .... https://zhuanlan.zhihu.com/p/642805630.

```
class Implementor {
public:
    virtual void OperationImpl() = 0;
};
```

```
class ConcreteImplementorA : public Implementor {
public:
    void OperationImpl() override {
        std::cout << "ConcreteImplementorA OperationImpl" << std::endl;
    }
};

class ConcreteImplementorB : public Implementor {
public:
    void OperationImpl() override {
        std::cout << "ConcreteImplementorB OperationImpl" << std::endl;
    }
};

class Abstraction {
protected:
    Implementor* implementor;
public:
    Abstraction(Implementor* implementor) : implementor(implementor) {}
    virtual void Operation() {
        implementor->OperationImpl();
    }
};

class RefinedAbstraction : public Abstraction {
public:
    RefinedAbstraction(Implementor* implementor) : Abstraction(implementor) {}
    void Operation() override {
        std::cout << "RefinedAbstraction Operation" << std::endl;
        implementor->OperationImpl();
    }
};
```

这段代码是一个桥接模式的实现。这里有五个类：`Implementor`、`ConcreteImplementorA`、`ConcreteImplementorB`、`Abstraction`和`RefinedAbstraction`。

- `Implementor`是一个抽象基类，定义了一个名为`OperationImpl`的纯虚函数。
- `ConcreteImplementorA`和`ConcreteImplementorB`是`Implementor`的两个具体实现，它们重写了`OperationImpl`函数，打印出 "ConcreteImplementorA OperationImpl" 和 "ConcreteImplementorB OperationImpl"。
- `Abstraction`是一个抽象类，它包含了一个`Implementor`类型的指针`implementor`。它有一个`Operation`方法，这个方法会调用`implementor`的`OperationImpl`方法。
- `RefinedAbstraction`是`Abstraction`的一个具体实现，它重写了`Operation`方法，在打印出 "RefinedAbstraction Operation" 后，调用`implementor`的`OperationImpl`方法。

这样，我们就可以创建一个`Implementor`对象，然后用`Abstraction`或`RefinedAbstraction`来封装它，使得`Abstraction`和`Implementor`可以独立变化。这就是桥接模式的主要优点。

应用场景

桥接模式在实际应用中有很多场景，以下是一些例子：

1. **游戏开发**：在游戏开发中，可以使用桥接模式来处理游戏角色的不同属性，角色可以作为抽象部分，而属性可以作为实现部分，通过桥接模式可以灵活地增加或修改角色的属性[3]。

2. **电子设备的遥控器**：在电子设备的遥控器中，可以使用桥接模式将遥控器的功能和不同的电子设备分离开来，使得遥控器可以适用于不同的电子设备[3]。
3. **跨平台的视频播放器**：例如，开发跨平台的视频播放器，平台有 Android，iOS，Windows，Linux，Mac，播放器支持的格式有 MP4，AVI，RMVB，FLV 格式。这种情况下，适合使用桥接模式[1]。

总的来说，桥接模式可以用于任何需要将抽象和实现分离，并且使它们可以独立变化的场景。它提供了一种灵活和可扩展的设计解决方案，可以减少类之间的耦合性，并且方便地进行功能扩展和修改[3]。
(1) JAVA设计模式9：桥接模式，将抽象部分与实现部分分离 .... https://cloud.tencent.com/developer/article/2331578.
(2) 【精选】【设计模式】桥接模式 (简介 | 适用场景 .... https://blog.csdn.net/shulianghan/article/details/119842918.
(3) 设计模式-桥接模式及应用 - 掘金. https://juejin.cn/post/7025937532042870798.
(4) 详解设计模式：桥接模式-腾讯云开发者社区-腾讯云. https://cloud.tencent.com/developer/article/2183989.

游戏开发中使用桥接模式

当然可以。在游戏开发中，我们经常需要处理各种各样的游戏角色，这些角色可能有各种不同的属性，比如攻击力、防御力、移动速度等。同时，我们可能还需要为这些角色添加各种特殊的技能或行为，比如跳跃、冲刺、施放魔法等。

如果我们直接在角色类中添加所有这些属性和行为，那么角色类将会变得非常复杂，而且难以维护。此外，如果我们想要添加一个新的属性或行为，我们可能需要修改角色类的代码，这可能会引入错误，并且破坏已有的功能。

为了解决这个问题，我们可以使用桥接模式。我们可以创建一个 `Character` 类，这个类包含了所有角色共有的属性和行为。然后，我们可以创建一个 `CharacterAttribute` 类，这个类是一个抽象类，定义了所有属性的接口。我们可以为每种属性创建一个具体的 `CharacterAttribute` 子类，比如 `AttackPower`、`DefensePower` 和 `MoveSpeed`。

同样，我们也可以创建一个 `CharacterBehavior` 类，这个类是一个抽象类，定义了所有行为的接口。我们可以为每种行为创建一个具体的 `CharacterBehavior` 子类，比如 `Jump`、`Dash` 和 `CastMagic`。

然后，我们可以在 `Character` 类中包含一个 `CharacterAttribute` 对象和一个 `CharacterBehavior` 对象。这样，我们就可以在不修改 `Character` 类的情况下，动态地为角色添加新的属性和行为。例如，我们可以创建一个 `Character` 对象，然后用 `AttackPower` 和 `Jump` 来装饰它，这样我们就得到了一个可以跳跃并且有攻击力的角色。

总的来说，桥接模式可以帮助我们将角色的属性和行为分离开来，使得我们可以灵活地为角色添加新的属性和行为，而不需要修改角色类的代码。

```
// 抽象化角色：角色
class Character {
protected:
    CharacterAttribute* attribute;
    CharacterBehavior* behavior;
public:
    Character(CharacterAttribute* attribute, CharacterBehavior* behavior)
        : attribute(attribute), behavior(behavior) {}
    virtual void Operation() {
        attribute->OperationImpl();
        behavior->OperationImpl();
    }
};
```

```
// 扩展抽象化角色：具体角色
class ConcreteCharacter : public Character {
public:
    ConcreteCharacter(CharacterAttribute* attribute, CharacterBehavior*
behavior)
        : Character(attribute, behavior) {}
    void Operation() override {
        std::cout << "ConcreteCharacter Operation" << std::endl;
        Character::Operation();
    }
};

// 实现化角色：属性
class CharacterAttribute {
public:
    virtual void OperationImpl() = 0;
};

// 具体实现化角色：具体属性
class AttackPower : public CharacterAttribute {
public:
    void OperationImpl() override {
        std::cout << "AttackPower OperationImpl" << std::endl;
    }
};

class DefensePower : public CharacterAttribute {
public:
    void OperationImpl() override {
        std::cout << "DefensePower OperationImpl" << std::endl;
    }
};

// 实现化角色：行为
class CharacterBehavior {
public:
    virtual void OperationImpl() = 0;
};

// 具体实现化角色：具体行为
class Jump : public CharacterBehavior {
public:
    void OperationImpl() override {
        std::cout << "Jump OperationImpl" << std::endl;
    }
};

class Dash : public CharacterBehavior {
public:
    void OperationImpl() override {
        std::cout << "Dash OperationImpl" << std::endl;
    }
};
```

桥接模式和外观模式的区别

桥接模式和外观模式都是设计模式，但它们的目标和使用方式有所不同[1][2][3][4][5]。

- **桥接模式**的主要目标是将抽象化与实现化解耦，使得二者可以独立变化[1][2][3][4][5]。这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类，这两种类型的类可被结构化改变而互不影响[1][2][3][4][5]。桥接模式通常用于系统设计的前期，即在设计类时将类规划为逻辑和实现两个大类，使他们可以分别精心演化[4]。
- **外观模式**的主要目标是提供一个统一的接口，用来访问子系统中的一群接口[1][2][3][4][5]。这种模式定义了一个高层接口，使得子系统更易于使用[1][2][3][4][5]。外观模式通常用于封装复杂的子系统，并提供一个简化的接口[1][2][3][4][5]。

总的来说，虽然桥接模式和外观模式都是设计模式，但它们的应用范围、实现方法和目标等方面有所不同[1][2][3][4][5]。

(1) 设计模式之：外观模式和桥接模式 - 知乎. https://zhuanlan.zhihu.com/p/82702337.
(2) 简单总结下外观、桥接模式_桥接模式和外观模式区别 .... https://blog.csdn.net/ajsliu1233/article/details/106125360.
(3) 【模式对比】装饰器模式 VS 桥接模式 - 知乎. https://zhuanlan.zhihu.com/p/627915806.
(4) 装饰者模式---装饰者模式和桥接模式的区别 - CSDN博客. https://blog.csdn.net/qq_45167987/article/details/121175029.
(5) 设计模式：外观模式(Fecade)与桥接模式(Bridge)详解 .... https://blog.csdn.net/shenpibaipao/article/details/77234084.

18.组合模式（Composite Pattern）：

组合模式，也被称为部分整体模式，是一种将对象组合成树状的层次结构的模式，用来表示“整体-部分”的关系，使用户对单个对象和组合对象具有一致的访问性[1][2][3]。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构[1][2][3]。

组合模式的主要优点是：

1. 组合模式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[1]。
2. 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”[1]。

其主要缺点是：

1. 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[1]。
2. 不容易限制容器中的构件[1]。
3. 不容易用继承的方法来增加构件的新功能[1]。

组合模式的使用场景包括：

1. 当你的程序结构有类似树一样的层级关系时，例如文件系统，视图树，公司组织架构等等[2]。
2. 当你要以统一的方式操作单个对象和由这些对象组成的组合对象的时候[2]。

(1) 27 设计模式——组合模式（详解版） - 知乎. https://zhuanlan.zhihu.com/p/444784138.
(2) 秒懂设计模式之组合模式（Composite Pattern） - 知乎. https://zhuanlan.zhihu.com/p/369731677.
(3) 组合模式 | 菜鸟教程. https://www.runoob.com/design-pattern/composite-pattern.html.

```
class Component {
public:
    virtual void Operation() = 0;
    virtual void Add(Component* component) {}
```

```
    virtual void Remove(Component* component) {}
    virtual Component* GetChild(int index) { return nullptr; }
};

class Leaf : public Component {
public:
    void Operation() override {
        std::cout << "Leaf Operation" << std::endl;
    }
};

class Composite : public Component {
private:
    std::vector<Component*> children;
public:
    void Operation() override {
        for (Component* child : children) {
            child->Operation();
        }
    }
    void Add(Component* component) override {
        children.push_back(component);
    }
    void Remove(Component* component) override {
        children.erase(std::remove(children.begin(), children.end(), component),
children.end());
    }
    Component* GetChild(int index) override {
        if (index >= 0 && index < children.size()) {
            return children[index];
        }
        return nullptr;
    }
};
```

19.**中介者模式（Mediator Pattern）**

中介者模式（Mediator Pattern）是一种设计模式，用于降低多个对象和类之间的通信复杂性[1][2][3]。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护[1][2][3]。中介者模式属于行为型模式[1][2][3]。

中介者模式的主要优点是：

1. 降低了类的复杂度，将一对多转化成了一对一[1]。
2. 各个类之间的解耦[1]。
3. 符合迪米特原则[1]。

其主要缺点是：

1. 中介者会庞大，变得复杂难以维护[1]。

中介者模式的使用场景包括：

1. 系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象[1]。
2. 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类[1]。

(1) 中介者模式 | 菜鸟教程. https://www.runoob.com/design-pattern/mediator-pattern.html.
(2) 36 设计模式——中介者模式 详解 - 知乎. https://zhuanlan.zhihu.com/p/447208807.
(3) 【设计模式系列17】中介者模式原理及其在JDK源码中的 .... https://bing.com/search?q=%e4%b8%ad%e4%bb%8b%e8%80%85%e6%a8%a1%e5%bc%8f%e8%a7%a3%e9%87%8a.
(4) 【精选】中介者模式与解释器模式_解释器模式和中介 .... https://blog.csdn.net/qq_43170213/article/details/120388913.

```
class Mediator;

class Colleague {
protected:
    Mediator* mediator;
public:
    virtual void Send(std::string message) = 0;
    virtual void Notify(std::string message) = 0;
};

class Mediator {
public:
    virtual void Send(std::string message, Colleague* colleague) = 0;
};

class ConcreteColleague1 : public Colleague {
public:
    void Send(std::string message) override {
        mediator->Send(message, this);
    }
    void Notify(std::string message) override {
        std::cout << "Colleague1 gets message: " << message << std::endl;
    }
};

class ConcreteColleague2 : public Colleague {
public:
    void Send(std::string message) override {
        mediator->Send(message, this);
    }
    void Notify(std::string message) override {
        std::cout << "Colleague2 gets message: " << message << std::endl;
    }
};

class ConcreteMediator : public Mediator {
private:
    ConcreteColleague1* colleague1;
    ConcreteColleague2* colleague2;
public:
    void Send(std::string message, Colleague* colleague) override {
        if (colleague == colleague1) {
            colleague2->Notify(message);
        } else {
            colleague1->Notify(message);
        }
    }
};
```

这段代码是一个中介者模式的实现。这里有四个类：`Colleague`、`Mediator`、`ConcreteColleague1`和`ConcreteColleague2`。

- `Colleague`是一个抽象基类，定义了两个纯虚函数`Send`和`Notify`。它还包含了一个指向`Mediator`的指针`mediator`。
- `ConcreteColleague1`和`ConcreteColleague2`是`Colleague`的两个具体实现，它们重写了`Send`和`Notify`函数。在`Send`函数中，它们调用`mediator`的`Send`方法来发送消息。在`Notify`函数中，它们打印出收到的消息。
- `Mediator`是一个抽象基类，定义了一个纯虚函数`Send`。
- `ConcreteMediator`是`Mediator`的一个具体实现，它重写了`Send`函数。在`Send`函数中，它根据消息的发送者来决定将消息转发给哪个`Colleague`。

这样，`Colleague`对象可以通过`Mediator`对象来发送和接收消息，而不需要直接与其他`Colleague`对象交互。这就是中介者模式的主要优点。

20.**解释器模式（Interpreter Pattern）**

解释器模式（Interpreter Pattern）是一种设计模式，它提供了评估语言的语法或表达式的方式，属于行为型模式[2][3]。这种模式实现了一个表达式接口，该接口解释一个特定的上下文[2][3]。这种模式被用在 SQL 解析、符号处理引擎等[2][3]。

解释器模式的主要角色有[1]：

- **抽象表达式（Abstract Expression）**：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()。
- **终结符表达式（Terminal Expression）**：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。
- **非终结符表达式（Nonterminal Expression）**：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。
- **上下文（Context）**：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[1]。

解释器模式的优点包括扩展性好，利用继承机制扩展文法[1]。但是，它的缺点是效率低，调用麻烦，容易引起类膨胀，这个模式很少能用到[1]。

(1) 解释器模式 | 菜鸟教程. https://www.runoob.com/design-pattern/interpreter-pattern.html.
(2) 设计模式之 解释器模式 Java实例代码演示 - CSDN博客. https://blog.csdn.net/qq_41688840/article/details/107727182.
(3) 看完就明白的解释器模式 - 知乎. https://zhuanlan.zhihu.com/p/394063958.
(4) 精读《设计模式 - Interpreter 解释器模式》 - 知乎. https://zhuanlan.zhihu.com/p/345061737.

```
class AbstractExpression {
public:
    virtual void Interpret(std::string context) = 0;
};

class TerminalExpression : public AbstractExpression {
private:
    std::string data;
public:
    TerminalExpression(std::string data) : data(data) {}
    void Interpret(std::string context) override {
        if (context.find(data) != std::string::npos) {
            std::cout << data << " is found!" << std::endl;
```

```
        }
    }
};

class OrExpression : public AbstractExpression {
private:
    AbstractExpression* expr1;
    AbstractExpression* expr2;
public:
    OrExpression(AbstractExpression* expr1, AbstractExpression* expr2) :
expr1(expr1), expr2(expr2) {}
    void Interpret(std::string context) override {
        expr1->Interpret(context);
        expr2->Interpret(context);
    }
};
```

21.**备忘录模式（Memento Pattern）**

备忘录模式（Memento Pattern）也被称为快照模式或者令牌模式，是一种设计模式，它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[1][2]。这样，我们就可以在需要的时候将该对象恢复到原先保存的状态[1][2]。备忘录模式属于行为型模式[1][2]。

备忘录模式的主要优点是：

1. 提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态[1][2]。
2. 实现了信息的封装，使得用户不需要关心状态的保存细节[1][2]。

其主要缺点是：

1. 消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存[1][2]。

备忘录模式的使用场景包括：

1. 需要保存/恢复数据的相关状态场景[1][2]。
2. 提供一个可回滚的操作[1][2]。

(1) 【设计模式系列21】备忘录模式:我简单到已经让人 .... https://blog.csdn.net/zwx900102/article/details/109294498.
(2) 备忘录模式 | 菜鸟教程. https://www.runoob.com/design-pattern/memento-pattern.html.

```
class Memento {
private:
    std::string state;
public:
    Memento(std::string state) : state(state) {}
    std::string GetState() { return state; }
};

class Originator {
private:
    std::string state;
public:
    void SetState(std::string state) { this->state = state; }
    std::string GetState() { return state; }
    Memento* SaveStateToMemento() { return new Memento(state); }
```

```
    void GetStateFromMemento(Memento* memento) { state = memento->GetState(); }
};

class CareTaker {
private:
    std::vector<Memento*> mementoList;
public:
    void Add(Memento* state) { mementoList.push_back(state); }
    Memento* Get(int index) { return mementoList[index]; }
};
```

这是一个实现了备忘录模式的C++代码。备忘录模式是一种行为设计模式，允许在不破坏对象的封装性的情况下保存和恢复对象之前的状态。

这个代码中有三个类：`Memento`，`Originator`和`CareTaker`。

- `Memento`类是用来存储`Originator`对象的状态的。它有一个私有的`state`字符串和一个公有的获取状态的方法`GetState()`。
- `Originator`类是我们希望保存状态的对象。它有一个私有的`state`字符串，一个设置状态的方法`SetState()`，一个获取状态的方法`GetState()`，一个保存状态到备忘录的方法`SaveStateToMemento()`，和一个从备忘录获取状态的方法`GetStateFromMemento()`。
- `CareTaker`类是用来保存和恢复`Memento`对象的。它有一个私有的`mementoList`向量，一个添加状态的方法`Add()`，和一个获取状态的方法`Get()`。

这个设计模式在需要保存一个对象在某一时刻的状态，以便在未来需要时恢复的情况下非常有用。例如，你可以在一个文本编辑器中使用它来实现撤销和重做操作。

使用备忘录模式

备忘录模式的使用通常涉及以下步骤：

1. 创建一个`Originator`对象，并设置其初始状态。

```
Originator* originator = new Originator();
originator->SetState("State1");
```

2. 使用`Originator`对象的`SaveStateToMemento()`方法保存当前状态到一个`Memento`对象中。

```
Memento* memento = originator->SaveStateToMemento();
```

3. 创建一个`CareTaker`对象，并使用其`Add()`方法将`Memento`对象添加到`CareTaker`对象中。

```
CareTaker* careTaker = new CareTaker();
careTaker->Add(memento);
```

4. 更改`Originator`对象的状态。

```
originator->SetState("State2");
```

5. 当需要恢复到之前的状态时，使用`CareTaker`对象的`Get()`方法获取相应的`Memento`对象，然后使用`Originator`对象的`GetStateFromMemento()`方法从`Memento`对象中恢复状态。

```
originator->GetStateFromMemento(careTaker->Get(0));
```

以上就是备忘录模式的基本使用方法。这种模式在需要保存和恢复对象状态的场景中非常有用，例如实现撤销/重做操作、保存游戏进度等。

为什么不将CareTaker集成到Originator中

将`CareTaker`类集成到`Originator`类中，确实可以使代码更简洁，但这样做可能会违反**单一职责原则**。这个原则是指一个类应该只有一个引起它变化的原因。在这个例子中，`Originator`类的职责是管理其自身的状态，而`CareTaker`类的职责是管理`Memento`对象的集合。如果我们将`CareTaker`的功能集成到`Originator`中，那么`Originator`就需要同时管理自身的状态和`Memento`对象的集合，这就增加了`Originator`类的复杂性。

另外，将`CareTaker`集成到`Originator`中可能会破坏封装性。在备忘录模式中，`Memento`对象通常应该只能被`Originator`对象访问。如果`CareTaker`的功能被集成到`Originator`中，那么`Originator`就能直接访问和修改`Memento`对象，这可能会导致`Memento`对象的状态被错误地修改。

所以，虽然将`CareTaker`集成到`Originator`中可以使代码更简洁，但这样做可能会带来上述的一些问题。在设计模式的选择和使用中，我们需要根据具体的需求和上下文来做出决策。

22.工厂模式（Factory Pattern）

（1）简单工厂

```
class Product {
public:
    virtual void show() = 0;
};

class ConcreteProductA : public Product {
public:
    void show() override {
        std::cout << "I'm product A\n";
    }
};

class ConcreteProductB : public Product {
public:
    void show() override {
        std::cout << "I'm product B\n";
    }
};

class Factory {
public:
    std::unique_ptr<Product> createProduct(char type) {
        switch (type) {
            case 'A':
                return std::make_unique<ConcreteProductA>();
            case 'B':
                return std::make_unique<ConcreteProductB>();
            default:
                return nullptr;
        }
    }
};
```

(2) 工厂方法

```
class Product {
public:
    virtual void show() = 0;
};

class ConcreteProductA : public Product {
public:
    void show() override {
        std::cout << "I'm product A\n";
    }
};

class ConcreteProductB : public Product {
public:
    void show() override {
        std::cout << "I'm product B\n";
    }
};

class Factory {
public:
    virtual std::unique_ptr<Product> createProduct() = 0;
};

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

class ConcreteFactoryB : public Factory {
public:
    std::unique_ptr<Product> createProduct() override {
        return std::make_unique<ConcreteProductB>();
    }
};
```

(3)抽象工厂

```
class Product1 {
public:
    virtual void show() = 0;
};

class Product2 {
public:
    virtual void show() = 0;
};

class ConcreteProduct1A : public Product1 {
public:
    void show() override {
```

```
        std::cout << "I'm product 1A\n";
    }
};

class ConcreteProduct2A : public Product2 {
public:
    void show() override {
        std::cout << "I'm product 2A\n";
    }
};

class ConcreteProduct1B : public Product1 {
public:
    void show() override {
        std::cout << "I'm product 1B\n";
    }
};

class ConcreteProduct2B : public Product2 {
public:
    void show() override {
        std::cout << "I'm product 2B\n";
    }
};

class AbstractFactory {
public:
    virtual std::unique_ptr<Product1> createProduct1() = 0;
    virtual std::unique_ptr<Product2> createProduct2() = 0;
};

class ConcreteFactoryA : public AbstractFactory {
public:
    std::unique_ptr<Product1> createProduct1() override {
        return std::make_unique<ConcreteProduct1A>();
    }

    std::unique_ptr<Product2> createProduct2() override {
        return std::make_unique<ConcreteProduct2A>();
    }
};

class ConcreteFactoryB : public AbstractFactory {
public:
    std::unique_ptr<Product1> createProduct1() override {
        return std::make_unique<ConcreteProduct1B>();
    }

    std::unique_ptr<Product2> createProduct2() override {
        return std::make_unique<ConcreteProduct2B>();
    }
};
```