设计模式

## 一．创建型模式

#### ****1.单例模式（Singleton）****

* 1. **定义**：确保一个类只有一个实例，并提供一个全局访问点。

class Singleton {public:

static Singleton& getInstance() {

static Singleton instance;

return instance;

}private:

Singleton() {}

Singleton(const Singleton&) = delete;

Singleton& operator=(const Singleton&) = delete;

};

* · **使用场景**：当一个类只能有一个实例并且需要全局访问点时。例如，日志记录器、线程池、配置管理类。

#### ****2.工厂模式（Factory Method）****

* 1. **定义**：定义一个创建对象的接口，但由子类决定要实例化的类是哪一个。

class Product {public:

virtual void use() = 0;

};

class ConcreteProductA : public Product {public:

void use() override { /\*...\*/ }

};

class Creator {public:

virtual Product\* factoryMethod() = 0;

};

class ConcreteCreator : public Creator {public:

Product\* factoryMethod() override {

return new ConcreteProductA();

}

};

**使用场景**：当一个类无法预料需要创建哪种类的实例时，或创建逻辑在子类中实现。例如，跨平台UI库中创建不同平台的控件。

#### ****3.抽象工厂模式（Abstract Factory）****

* 1. **定义**：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

class AbstractProductA {public:

virtual void use() = 0;

};

class AbstractProductB {public:

virtual void eat() = 0;

};

class ConcreteProductA1 : public AbstractProductA {public:

void use() override { /\*...\*/ }

};

class ConcreteProductB1 : public AbstractProductB {public:

void eat() override { /\*...\*/ }

};

class AbstractFactory {public:

virtual AbstractProductA\* createProductA() = 0;

virtual AbstractProductB\* createProductB() = 0;

};

class ConcreteFactory1 : public AbstractFactory {public:

AbstractProductA\* createProductA() override {

return new ConcreteProductA1();

}

AbstractProductB\* createProductB() override {

return new ConcreteProductB1();

}

};

**使用场景**：当需要创建一系列相关或互相依赖的对象，并且不希望指定它们的具体类时。例如，创建不同主题的一组UI组件。

#### ****4建造者模式（Builder）****

* 1. **定义**：将一个复杂对象的构建与其表示分离，使得同样的构建过程可以创建不同的表示。

class Product {public:

void setPartA(const std::string& partA) { /\*...\*/ }

void setPartB(const std::string& partB) { /\*...\*/ }

};

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->setPartA("A"); }

void buildPartB() override { product->setPartB("B"); }

Product\* getResult() override { return product; }

};

class Director {private:

Builder\* builder;public:

void setBuilder(Builder\* b) { builder = b; }

void construct() {

builder->buildPartA();

builder->buildPartB();

}

};

**使用场景**：当创建一个复杂对象需要多个步骤或配置时，并且希望代码能一步一步构造对象时。例如，创建复杂的文档对象或复杂的对象图。

#### ****5,原型模式（Prototype）****

* 1. **定义**：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
  2. **实**

class Prototype {public:

virtual Prototype\* clone() const = 0;

};

class ConcretePrototype : public Prototype {public:

Prototype\* clone() const override {

return new ConcretePrototype(\*this);

}

};

**使用场景**：当一个系统需要独立于其对象创建过程时，或希望避免创建对象时的性能开销时。例如，克隆对象（如大型对象的复制）。

### 二、结构型模式

#### ****1.适配器模式（Adapter）****

* 1. **定义**：将一个类的接口转换成客户希望的另一个接口，使得原本接口不兼容的类可以一起工作。

class Target {public:

virtual void request() = 0;

};

class Adaptee {public:

void specificRequest() { /\*...\*/ }

};

class Adapter : public Target {private:

Adaptee\* adaptee;public:

Adapter(Adaptee\* a) : adaptee(a) {}

void request() override {

adaptee->specificRequest();

}

};

**使用场景**：当需要将一个类的接口转换成客户期望的另一个接口时。例如，连接新旧系统，或不同接口规范的系统之间的适配。

#### ****2.装饰模式（Decorator）****

* 1. **定义**：动态地给对象添加一些职责。

class Component {public:

virtual void operation() = 0;

};

class ConcreteComponent : public Component {public:

void operation() override { /\*...\*/ }

};

class Decorator : public Component {protected:

Component\* component;public:

Decorator(Component\* c) : component(c) {}

void operation() override {

component->operation();

}

};

class ConcreteDecorator : public Decorator {public:

ConcreteDecorator(Component\* c) : Decorator(c) {}

void operation() override {

Decorator::operation();

additionalOperation();

}private:

void additionalOperation() { /\*...\*/ }

};

**使用场景**：当需要动态地给一个对象添加额外的职责时，而不希望影响其他对象时。例如，动态扩展对象的功能（如窗口系统中的滚动条）。

#### ****3代理模式（Proxy）****

* 1. **定义**：为其他对象提供一种代理以控制对这个对象的访问。
  2. **实**

class Subject {public:

virtual void request() = 0;

};

class RealSubject : public Subject {public:

void request() override { /\*...\*/ }

};

class Proxy : public Subject {private:

RealSubject\* realSubject;public:

void request() override {

if (!realSubject) {

realSubject = new RealSubject();

}

realSubject->request();

}

};

**使用场景**：当需要控制对一个对象的访问时。例如，远程代理（控制对远程对象的访问）、虚拟代理（控制开销较大的对象的访问）、保护代理（控制对对象的访问权限）。

#### ****4.桥接模式（Bridge）****

* 1. **定义**：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

class Implementor {public:

virtual void operationImpl() = 0;

};

class ConcreteImplementorA : public Implementor {public:

void operationImpl() override { /\*...\*/ }

};

class Abstraction {protected:

Implementor\* implementor;public:

Abstraction(Implementor\* impl) : implementor(impl) {}

virtual void operation() {

implementor->operationImpl();

}

};

class RefinedAbstraction : public Abstraction {public:

RefinedAbstraction(Implementor\* impl) : Abstraction(impl) {}

void operation() override {

implementor->operationImpl();

}

};

**使用场景**：当需要将抽象部分与实现部分分离，使它们都可以独立地变化时。例如，不同操作系统上的图形处理。

#### ****5组合模式（Composite）****

* 1. **定义**：将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

class Component {public:

virtual void operation() = 0;

virtual void add(Component\* c) { }

virtual void remove(Component\* c) { }

virtual Component\* getChild(int index) { return nullptr; }

};

class Leaf : public Component {public:

void operation() override { /\*...\*/ }

};

class Composite : public Component {private:

std::vector<Component\*> children;public:

void operation() override {

for (auto child : children) {

child->operation();

}

}

void add(Component\* c) override {

children.push\_back(c);

}

void remove(Component\* c) override {

children.erase(std::remove(children.begin(), children.end(), c), children.end());

}

Component\* getChild(int index) override {

return children.at(index); }};

**使用场景**：当需要表示对象的部分-整体层次结构时，并且希望用户对单个对象和组合对象的使用具有一致性时。例如，文件系统中的文件和文件夹结构

## 三．行为型模式

#### ****1.观察者模式（Observer）****

* 1. **定义**：定义对象间的一对多依赖关系，当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。

class Observer {public:

virtual void update() = 0;

};

class Subject {private:

std::vector<Observer\*> observers;public:

void attach(Observer\* o) {

observers.push\_back(o);

}

void detach(Observer\* o) {

observers.erase(std::remove(observers.begin(), observers.end(), o), observers.end());

}

void notify() {

for (auto observer : observers) {

observer->update();

}

}

};

class ConcreteObserver : public Observer {public:

void update() override {

// 更新逻辑

}

};

**使用场景**：当一个对象的改变需要通知其他对象时，并且不希望这些对象之间是紧密耦合的时。例如，GUI框架中的事件处理机制，或者数据模型的变化通知视图。

#### ****2.策略模式（Strategy）****

* 1. **定义**：定义一系列算法，把它们一个个封装起来，并且使它们可以相互替换。本模式使得算法可独立于使用它的客户而变化。
  2. **实现**：

class Strategy {public:

virtual void execute() = 0;

};

class ConcreteStrategyA : public Strategy {public:

void execute() override { /\*...\*/ }

};

class ConcreteStrategyB : public Strategy {public:

void execute() override { /\*...\*/ }

};

class Context {private:

Strategy\* strategy;public:

void setStrategy(Strategy\* s) {

strategy = s;

}

void executeStrategy() {

strategy->execute();

}

};

**使用场景**：当需要在运行时动态选择算法时。例如，不同的排序算法或不同的路径查找算法。

#### ****3.状态模式（State）****

* 1. **定义**：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

class State {public:

virtual void handle() = 0;

};

class ConcreteStateA : public State {public:

void handle() override { /\*...\*/ }

};

class ConcreteStateB : public State {public:

void handle() override { /\*...\*/ }

};

class Context {private:

State\* state;public:

void setState(State\* s) {

state = s;

}

void request() {

state->handle();

}

};

**使用场景**：当一个对象的行为取决于其状态，并且需要在运行时改变其行为时。例如，有限状态机（如用户界面中的状态变化）。

#### ****4.命令模式（Command）****

* 1. **定义**：将请求封装成对象，从而使你可以用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

class Command {public:

virtual void execute() = 0;

};

class ConcreteCommand : public Command {private:

Receiver\* receiver;public:

ConcreteCommand(Receiver\* r) : receiver(r) {}

void execute() override {

receiver->action();

}

};

class Receiver {public:

void action() { /\*...\*/ }

};

class Invoker {private:

Command\* command;public:

void setCommand(Command\* c) {

command = c;

}

void invoke() {

command->execute();

}

};

**使用场景**：当需要将请求封装成对象，使得不同的请求、队列请求或日志请求能够使用相同的方式处理时。例如，菜单操作、撤销操作的实现。

#### ****5.责任链模式（Chain of Responsibility）****

* 1. **定义**：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受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。
  2. **实现**

class Handler {protected:

Handler\* next;public:

void setNext(Handler\* n) {

next = n;

}

virtual void handleRequest(int request) {

if (next) {

next->handleRequest(request);

}

}

};

class ConcreteHandler1 : public Handler {public:

void handleRequest(int request) override {

if (request == 1) {

// 处理请求

} else if (next) {

next->handleRequest(request);

}

}

};

class ConcreteHandler2 : public Handler {public:

void handleRequest(int request) override {

if (request == 2) {

// 处理请求

} else if (next) {

next->handleRequest(request);

}

}

};

**使用场景**：当多个对象都有机会处理请求，但具体哪个对象处理请求是在运行时确定时。例如，事件的处理链（如GUI事件处理），或者过滤器链。

#### ****6.中介者模式（Mediator）****

* 1. **定义**：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
  2. **实**

class Mediator;

class Colleague {protected:

Mediator\* mediator;public:

Colleague(Mediator\* m) : mediator(m) {}

};

class ConcreteColleague1 : public Colleague {public:

ConcreteColleague1(Mediator\* m) : Colleague(m) {}

void send(const std::string& message);

void receive(const std::string& message);

};

class ConcreteColleague2 : public Colleague {public:

ConcreteColleague2(Mediator\* m) : Colleague(m) {}

void send(const std::string& message);

void receive(const std::string& message);

};

class Mediator {public:

virtual void send(const std::string& message, Colleague\* colleague) = 0;

};

class ConcreteMediator : public Mediator {private:

ConcreteColleague1\* colleague1;

ConcreteColleague2\* colleague2;public:

void setColleague1(ConcreteColleague1\* c1) {

colleague1 = c1;

}

void setColleague2(ConcreteColleague2\* c2) {

colleague2 = c2;

}

void send(const std::string& message, Colleague\* colleague) override {

if (colleague == colleague1) {

colleague2->receive(message);

} else if (colleague == colleague2) {

colleague1->receive(message);

}

}

};

void ConcreteColleague1::send(const std::string& message) {

mediator->send(message, this);

}

void ConcreteColleague1::receive(const std::string& message) {

// 处理消息

}

void ConcreteColleague2::send(const std::string& message) {

mediator->send(message, this);

}

void ConcreteColleague2::receive(const std::string& message) {

// 处理消息

}

**使用场景**：当多个对象之间有复杂的交互关系，需要通过一个中介对象来减少对象之间的耦合时。例如，聊天系统中的消息传递或组件之间的通信。

#### · 7****迭代器模式（Iterator）****

* · **定义**：提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。

template <typename T>class Iterator {public:

virtual ~Iterator() {}

virtual bool hasNext() = 0;

virtual T next() = 0;

};

template <typename T>class Aggregate {public:

virtual ~Aggregate() {}

virtual Iterator<T>\* createIterator() = 0;

};

template <typename T>class ConcreteIterator : public Iterator<T> {private:

std::vector<T> items;

int index;public:

ConcreteIterator(const std::vector<T>& items) : items(items), index(0) {}

bool hasNext() override {

return index < items.size();

}

T next() override {

return items[index++];

}

};

template <typename T>class ConcreteAggregate : public Aggregate<T> {private:

std::vector<T> items;public:

Iterator<T>\* createIterator() override {

return new ConcreteIterator<T>(items);

}

void addItem(const T& item) {

items.push\_back(item);

}

};

**使用场景**：当需要顺序访问一个聚合对象中的各个元素，而不暴露其内部实现时。例如，容器类（如数组、列表）的遍历。

#### 8 ****模板方法模式（Template Method）****

* · **定义**：定义一个操作中的算法骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。

class AbstractClass {public:

void templateMethod() {

primitiveOperation1();

primitiveOperation2();

}protected:

virtual void primitiveOperation1() = 0;

virtual void primitiveOperation2() = 0;

};

class ConcreteClass : public AbstractClass {protected:

void primitiveOperation1() override {

// 实现细节

}

void primitiveOperation2() override {

// 实现细节

}

};

**使用场景**：当需要定义一个算法的骨架，并允许子类实现某些具体步骤时。例如，处理不同类型的数据的抽象处理框架。

#### · 9****备忘录模式（Memento）****

* · **定义**：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后恢复对象到原先保存的状态。

class Memento {private:

int state;public:

Memento(int s) : state(s) {}

int getState() {

return state;

}

};

class Originator {private:

int state;public:

void setState(int s) {

state = s;

}

int 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];

}

};

**使用场景**：当需要保存一个对象的状态以便以后恢复时，并且不希望破坏对象的封装性时。例如，撤销操作的实现。

每种设计模式都有其特定的使用场景，选择适当的设计模式可以提高代码的可维护性、可扩展性和重用性。在实际开发中，理解和应用这些设计模式可以有效解决常见的设计问题。