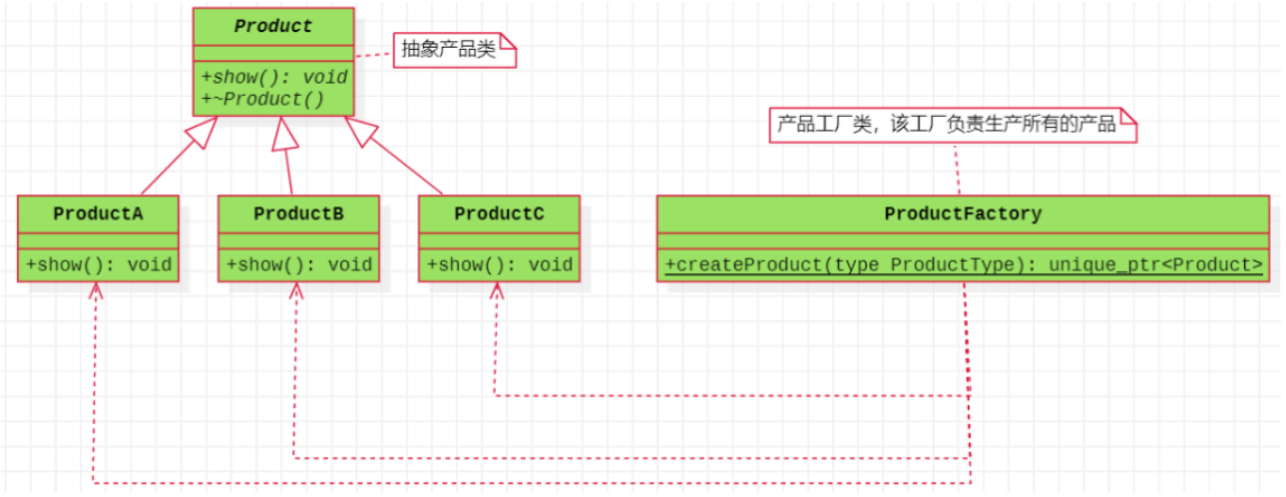
浅谈设计模式

简单工厂模式

概述

简单工厂模式又叫静态工厂方法模式，提供一个工厂类，在工厂类中做判断，根据传入的类型创造相应的产品，当增加新的产品时，就需要修改工厂类。简单工厂模式提供了专门的工厂类用于创建对象，将对象的创建和对象的使用分离开，它作为一种最简单的工厂模式在软件开发中得到了较为广泛的应用。

类图



具体实现

```
#include <iostream>
#include <memory>
using namespace std;

/*
    简单工厂模式（静态工厂方法模式）
*/

enum ProductType
```

```
{
    TypeA,
    TypeB,
    TypeC
};

// 抽象产品类
class Product
{
public:
    virtual void show() = 0;
    virtual ~Product() {}
};

// 各个产品
class ProductA : public Product
{
public:
    ProductA()
    {
        cout << "ProductA()" << endl;
    }

    void show() override
    {
        cout << "void ProductA::show()" << endl;
    }

    ~ProductA()
    {
        cout << "~ProductA()" << endl;
    }
};

class ProductB : public Product
{
public:
    ProductB()
    {
        cout << "ProductB()" << endl;
    }

```

```
52      void show() override
53      {
54          cout << "void ProductB::show()" << endl;
55      }
56
57      ~ProductB()
58      {
59          cout << "~ProductB()" << endl;
60      }
61  };
62
63  class ProductC : public Product
64  {
65  public:
66      ProductC()
67      {
68          cout << "ProductC()" << endl;
69      }
70
71      void show() override
72      {
73          cout << "void ProductC::show()" << endl;
74      }
75
76      ~ProductC()
77      {
78          cout << "~ProductC()" << endl;
79      }
80  };
81
82  // 产品工厂类
83  class ProductFactory
84  {
85  public:
86      // 基类指针指向派生类对象，构成多态的条件
87      static unique_ptr<Product> createProduct(ProductType type)
88      {
89          switch(type){
90              case TypeA:
91                  return unique_ptr<Product>(new ProductA());
92              case TypeB:
93                  return unique_ptr<Product>(new ProductB());
```

```
94              case TypeC:
95                  return unique_ptr<Product>(new ProductC());
96              default:
97                  cout << "This type cannot be recognized." <<
    endl;
98                  return unique_ptr<Product>(nullptr);
99          }
100     }
101 };
102
103 void test()
104 {
105     // 创建产品对象
106     unique_ptr<Product> pa =
    ProductFactory::createProduct(TypeA);
107     unique_ptr<Product> pb =
    ProductFactory::createProduct(TypeB);
108     unique_ptr<Product> pc =
    ProductFactory::createProduct(TypeC);
109
110     // 产品对象的使用（触发多态）
111     pa->show();
112     pb->show();
113     pc->show();
114 }
115
116 int main()
117 {
118     test();
119     return 0;
120 }
```

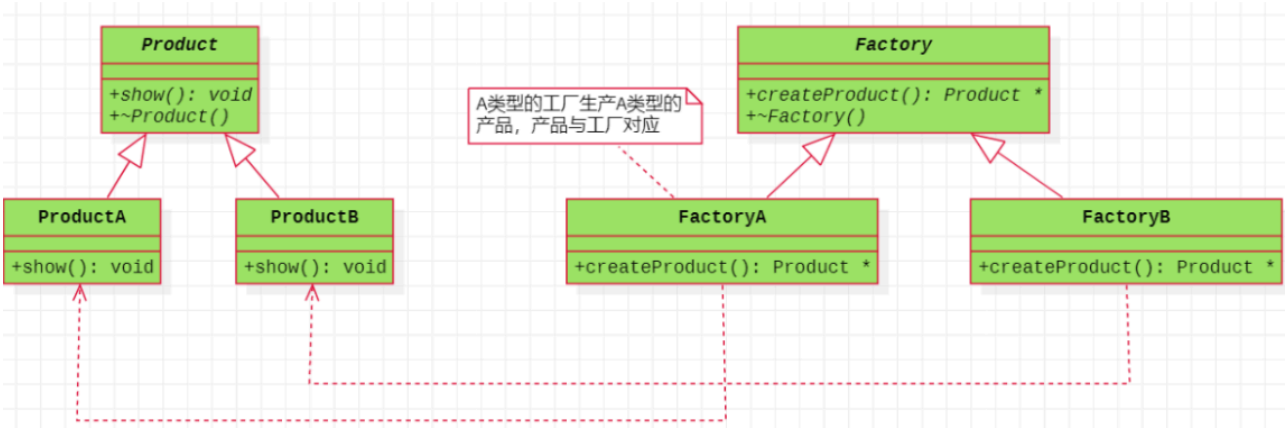
普通工厂模式

概述

在软件开发及运行过程中，经常需要创建对象，但常出现由于需求的变更，需要创建的对象的具体类型也要经常变化。工厂方法通过采取虚函数的方法，实现了使用者和具体类型之间的解耦，可以用来解决这个问题。工厂方法模式对简单工厂模式中的工厂类进一步抽象。核心工厂类不再负责产品的创建，而是演变为一个抽象工厂角色，仅负责定义具体工厂子类必须实现的接口，同时，针对不同的产品提供不同的工厂，即每个产品都有一个与

之对应的工厂。这样，系统在增加新产品时就不会修改工厂类逻辑而是添加新的工厂子类，从而弥补简单工厂模式对修改开放的缺陷。定义一个创建对象的接口，让子类决定实例化哪个类，该模式使类对象的创建延迟到子类。

类图



具体实现

```cpp
#include <iostream>
#include <memory>
using namespace std;

// 抽象产品类
class Product
{
public:
    virtual void show() = 0;
    virtual ~Product() {}
};

// 各个产品
class ProductA : public Product
{
public:
    ProductA()
    {
        cout << "ProductA()" << endl;
    }

    void show() override
    {
```

```
24          cout << "void ProductA::show()" << endl;
25      }
26
27      ~ProductA()
28      {
29          cout << "~ProductA()" << endl;
30      }
31  };
32
33  class ProductB : public Product
34  {
35  public:
36      ProductB()
37      {
38          cout << "ProductB()" << endl;
39      }
40
41      void show() override
42      {
43          cout << "void ProductB::show()" << endl;
44      }
45
46      ~ProductB()
47      {
48          cout << "~ProductB()" << endl;
49      }
50  };
51
52  class Factory
53  {
54  public:
55      virtual Product *createProduct() = 0;
56      virtual ~Factory() {}
57  };
58
59  class FactoryA : public Factory
60  {
61  public:
62      FactoryA()
63      {
64          cout << "FactoryA()" << endl;
65      }
```

```
66
67      Product *createProduct() override
68      {
69          return new ProductA();
70      }
71
72      ~FactoryA()
73      {
74          cout << "~FactoryA()" << endl;
75      }
76  };
77
78  class FactoryB : public Factory
79  {
80  public:
81      FactoryB()
82      {
83          cout << "FactoryB()" << endl;
84      }
85
86      Product *createProduct() override
87      {
88          return new ProductB();
89      }
90
91      ~FactoryB()
92      {
93          cout << "~FactoryB()" << endl;
94      }
95  };
96
97  void test()
98  {
99      // 创建生产产品A的工厂
100     unique_ptr<Factory> factoryA(new FactoryA());
101     // 生产产品A
102     unique_ptr<Product> productA(factoryA->createProduct());
103     // 使用产品A（触发多态）
104     productA->show();
105
106     // 创建生产产品B的工厂
107     unique_ptr<Factory> factoryB(new FactoryB());
```

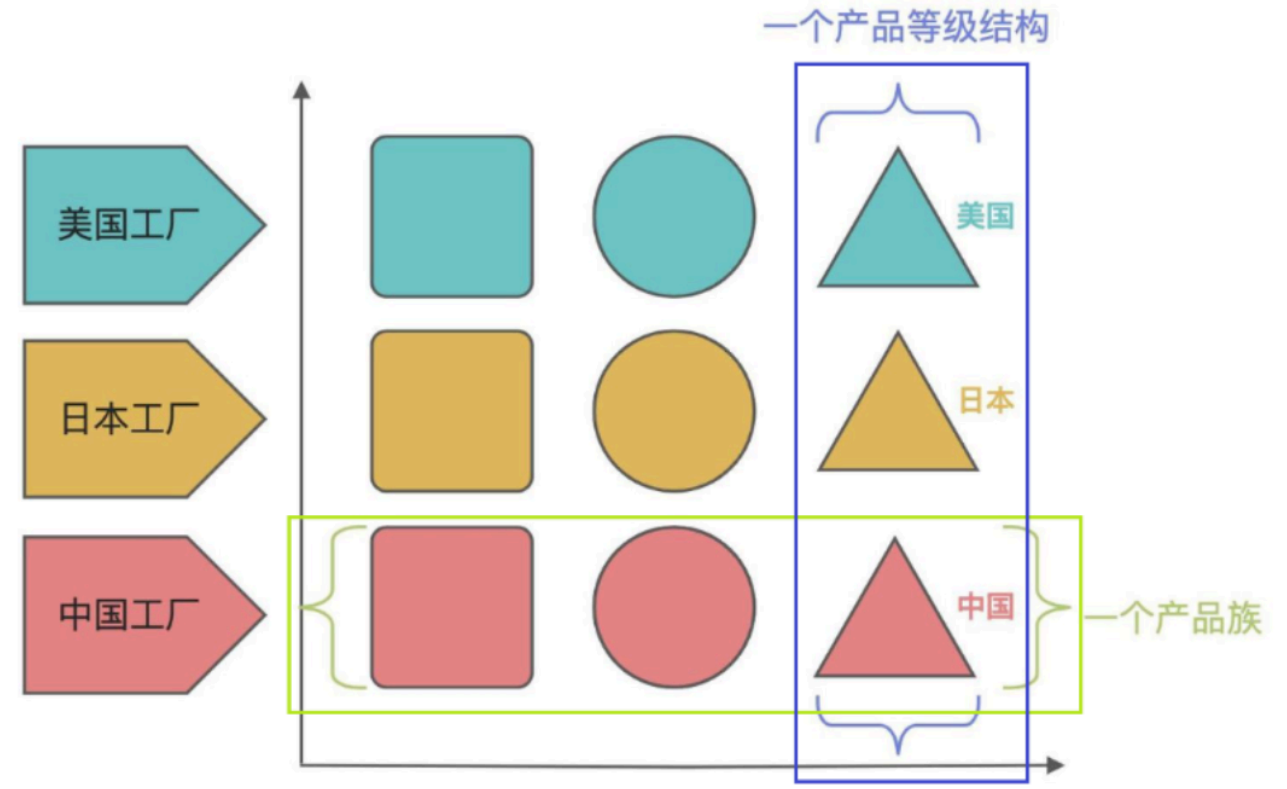
```
    // 生产产品B
    unique_ptr<Product> productB(factoryB->createProduct());
    // 使用产品B（触发多态）
    productB->show();
}

int main()
{
    test();
    return 0;
}
```

抽象工厂模式

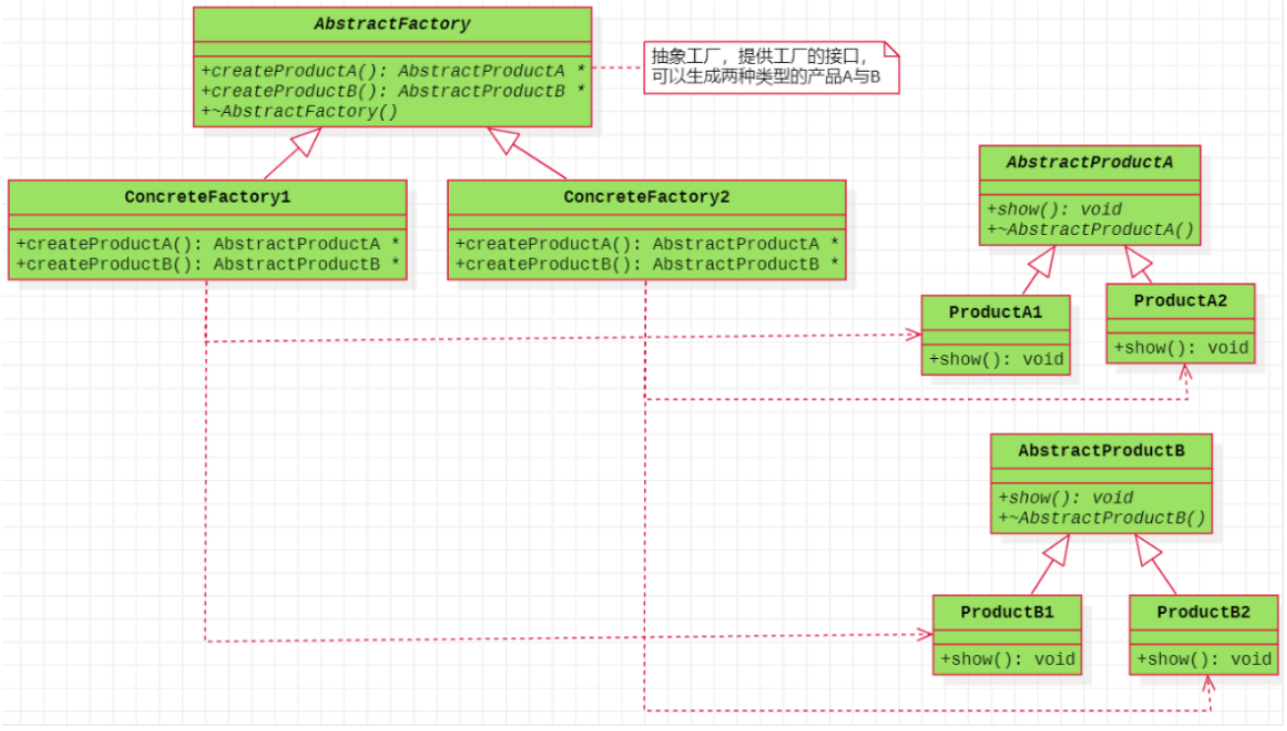
概述

在软件开发及运行过程中，经常面临着"一系列相互依赖的对象"的创建工作；而由于需求的变化，常常存在更多系列对象的创建问题。定义提供一个接口，该接口负责创建一系列"相关或者相互依赖的对象"，无需指定它们具体的类。这里有两个概念，产品等级结构与产品族的概念，对于理解抽象工厂很重要。

- 产品等级结构：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。（都是同一类产品，都是电视机）。
- 产品族：指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中，海尔电视机、海尔电冰箱构成了一个产品族。（不同类的产品，但是属于同一家公司）



类图



具体实现

```
#include <iostream>
#include <memory>
using namespace std;

```

```
5  class AbstractProductA
6  {
7  public:
8      virtual void show() = 0;
9      virtual ~AbstractProductA() {}
10 };
11
12 class ProductA1 : public AbstractProductA
13 {
14 public:
15     void show() override
16     {
17         cout << "void ProductA1::show()" << endl;
18     }
19 };
20
21 class ProductA2 : public AbstractProductA
22 {
23 public:
24     void show() override
25     {
26         cout << "void ProductA2::show()" << endl;
27     }
28 };
29
30 class AbstractProductB
31 {
32 public:
33     virtual void show() = 0;
34     virtual ~AbstractProductB() {}
35 };
36
37 class ProductB1 : public AbstractProductB
38 {
39 public:
40     void show() override
41     {
42         cout << "void ProductB1::show()" << endl;
43     }
44 };
45
46 class ProductB2 : public AbstractProductB
```

```
47  {
48  public:
49      void show() override
50      {
51          cout << "void ProductB2::show()" << endl;
52      }
53  };
54
55  class AbstractFactory
56  {
57  public:
58      virtual AbstractProductA *createProductA() = 0;
59      virtual AbstractProductB *createProductB() = 0;
60      virtual ~AbstractFactory() {}
61  };
62
63  class ConcreteFactory1 : public AbstractFactory
64  {
65  public:
66      AbstractProductA *createProductA() override
67      {
68          cout << "ConcreteFactory1::createProductA()" << endl;
69          return new ProductA1();
70      }
71
72      AbstractProductB *createProductB() override
73      {
74          cout << "ConcreteFactory1::createProductB()" << endl;
75          return new ProductB1();
76      }
77  };
78
79  class ConcreteFactory2 : public AbstractFactory
80  {
81  public:
82      AbstractProductA *createProductA() override
83      {
84          cout << "ConcreteFactory2::createProductA()" << endl;
85          return new ProductA2();
86      }
87
88      AbstractProductB *createProductB() override
```

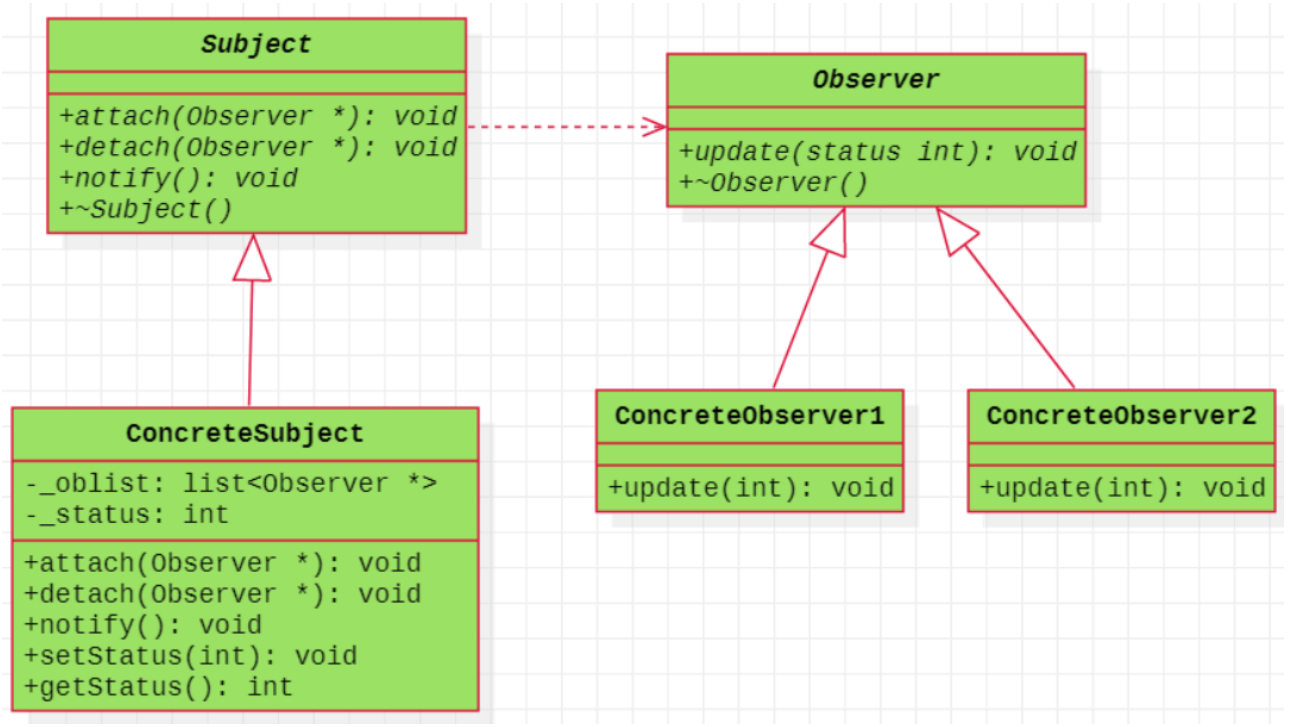
```
    {
        cout << "ConcreteFactory2::createProductB()" << endl;
        return new ProductB2();
    }
};

void test(){
    // 创建工厂
    unique_ptr<AbstractFactory> factory1(new ConcreteFactory1());
    unique_ptr<AbstractFactory> factory2(new ConcreteFactory2());

    // 生产产品
    unique_ptr<AbstractProductA> productA1(factory1->createProductA());
    unique_ptr<AbstractProductB> productB1(factory1->createProductB());
    productA1->show();
    productB1->show();

    cout << "--------------------------------------------------------" << endl;

    unique_ptr<AbstractProductA> productA2(factory2->createProductA());
    unique_ptr<AbstractProductB> productB2(factory2->createProductB());
    productA2->show();
    productB2->show();

}

int main()
{
    test();
    return 0;
}
```

观察者模式

概述

在GOF的《设计模式：可复用面向对象软件的基础》一书中对观察者模式是这样定义的：定义对象的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。当一个对象发生了变化，关注它的对象就会得到通知；这种交互也成为发布-订阅（publish-subscribe）。

类图



具体实现

```
#include <iostream>
#include <memory>
#include <string>
#include <list>
using namespace std;

class Observer;
class Subject
{
public:
    virtual void attach(Observer *pObserver) = 0;
    virtual void detach(Observer *pObserver) = 0;
```

```
13      virtual void notify() = 0;
14      virtual ~Subject() {}
15  };
16
17  class Observer
18  {
19  public:
20      virtual void update(int) = 0;
21      virtual ~Observer() {}
22  };
23
24  class ConcreteSubject : public Subject
25  {
26  public:
27      void attach(Observer *pObserver) override
28      {
29          if (pObserver)
30          {
31              _oblist.push_back(pObserver);
32          }
33      }
34
35      void detach(Observer *pObserver) override
36      {
37          if (pObserver)
38          {
39              _oblist.remove(pObserver);
40          }
41      }
42
43      void notify() override
44      {
45          for (auto & ob : _oblist)
46          {
47              ob->update(_status);
48          }
49      }
50
51      void setStatus(int status)
52      {
53          _status = status;
54      }
```

```
55
56      int getStatus()
57      {
58          return _status;
59      }
60
61  private:
62      list<Observer *> _oblist;
63      int _status;
64  };
65
66
67  class ConcreteObserver1 : public Observer
68  {
69  public:
70      ConcreteObserver1(const string &name)
71      :_name(name)
72      {}
73
74      void update(int value)
75      {
76          cout << "ConcreteObserver1 " << _name << ", value = "
    << value << endl;
77      }
78
79  private:
80      string _name;
81  };
82
83  class ConcreteObserver2 : public Observer
84  {
85  public:
86      ConcreteObserver2(const string & name)
87      :_name(name)
88      {
89      }
90
91      void update(int value)
92      {
93          cout << "ConcreteObserver2 " << _name << ", value = "
    << value << endl;
94      }
```

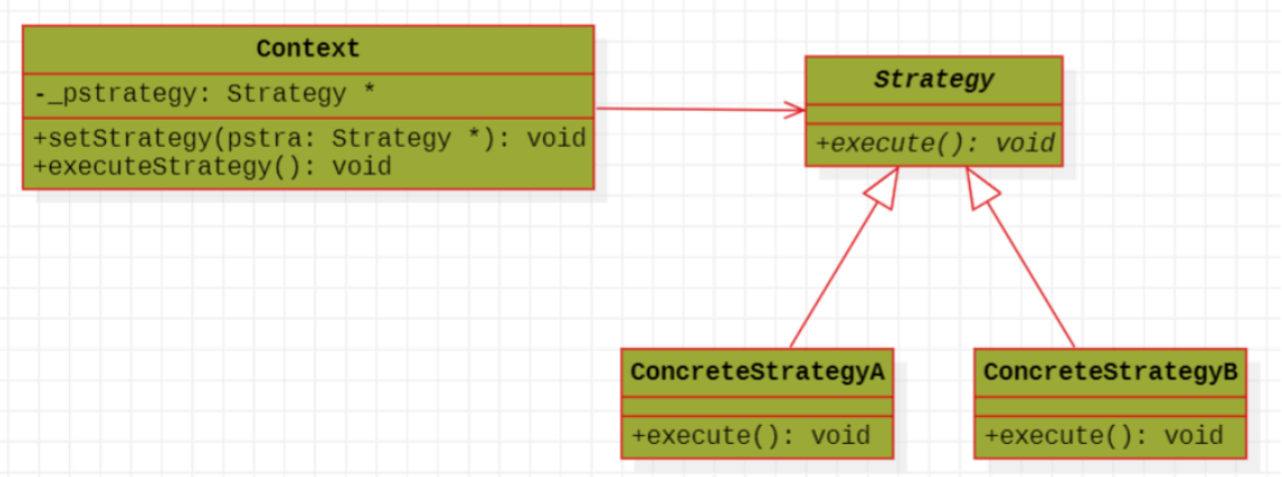
```
95  private:
96      string _name;
97  };
98
99  void test(){
100     // 创建ConcreteSubject对象
101     unique_ptr<ConcreteSubject> subject(new ConcreteSubject());
102     // 创建Observer对象
103     unique_ptr<Observer> pObserver1(new
    ConcreteObserver1("Linus"));
104     unique_ptr<Observer> pObserver2(new
    ConcreteObserver2("Wiles"));
105     subject->setStatus(2);
106
107     subject->attach(pObserver1.get());
108     subject->attach(pObserver2.get());
109
110     subject->notify();
111
112     subject->detach(pObserver2.get());
113     subject->setStatus(10);
114     subject->notify();
115 }
116
117 int main()
118 {
119     test();
120     return 0;
121 }
```

策略模式

概述

策略模式是一种行为型设计模式，策略模式在软件开发场景中定义了一系列的算法，并将每个算法单独封装在可替换的对象中，使应用程序在运行时可以根据具体的上下文来动态地选择和切换算法，同时保持原有的代码架构不被修改。策略模式的设计使得**算法的实现与调用被分离**，让算法可以独立于外部客户端进行开发和改动，使用独立的类来封装特定的算法，也避免了不同算法策略之间的互相影响。

类图



具体实现

```
#include <iostream>
#include <memory>
using namespace std;

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

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

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

class Context{
public:

    Context(Strategy *pstra)
```

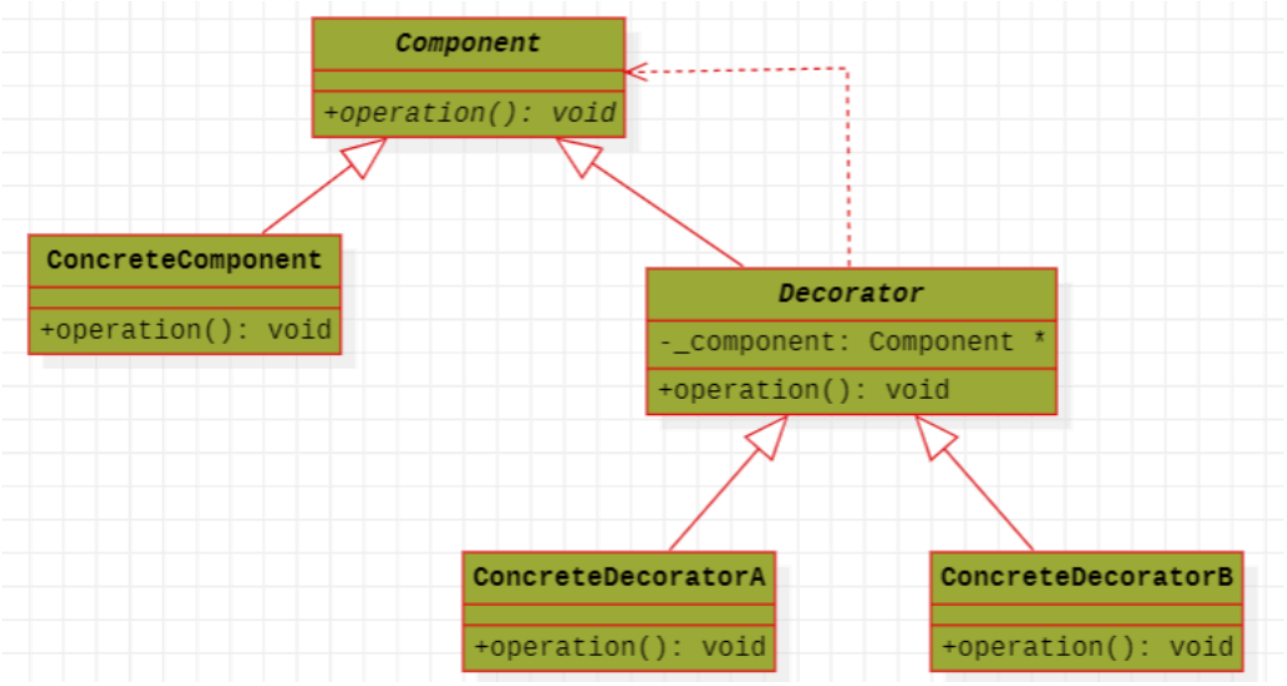
```
28      :_pStrategy(pstra)
29      {}
30
31      void setStrategy(Strategy * ps){
32          _pStrategy = ps;
33      }
34
35      void executeStrategy(){
36          if(_pStrategy){
37              _pStrategy->execute();
38          }
39      }
40
41      ~Context(){}
42
43  private:
44      Strategy *_pStrategy;
45  };
46
47  void test(){
48      // 创建策略
49      unique_ptr<Strategy> pa(new ConcreteStrategyA());
50      unique_ptr<Strategy> pb(new ConcreteStrategyB());
51
52      // 创建内容
53      Context context(pa.get());
54      context.executeStrategy();
55
56      // 更换策略
57      context.setStrategy(pb.get());
58      context.executeStrategy();
59  }
60
61  int main(){
62      test();
63      return 0;
64  }
```

装饰器模式

概述

装饰器模式是一种结构型设计模式，它允许你在不改变已有对象代码的情况下，动态地给对象添加新的行为和责任。这种模式通过创建一系列装饰器类，每个装饰器类都包装了一个具体组件对象，并提供了与组件相同的接口，以便能够递归地将多个装饰器叠加到一个对象上，从而达到动态扩展对象功能的目的，既保持了类的封装性，又提高了代码的灵活性。

类图



具体实现

```cpp
#include <iostream>
#include <memory>
using namespace std;

// 抽象类作为接口
class Component{
public:
    Component(){
        cout << "Component()" << endl;
    }

    ~Component(){
```

```
13          cout << "~Component()" << endl;
14      }
15
16      virtual void operation() = 0;
17  };
18
19  // 具体组件
20  class ConcreteComponent : public Component{
21  public:
22
23      ConcreteComponent(){
24          cout << "ConcreteComponent()" << endl;
25      }
26
27      virtual void operation () override{
28          cout << "具体对象的操作" << endl;
29      }
30
31      ~ConcreteComponent(){
32          cout << "~ConcreteComponent()" << endl;
33      }
34  };
35
36  // 装饰器
37  class Decorator : public Component{
38  public:
39      Decorator(){
40          cout << "Decorator()" << endl;
41      }
42
43      ~Decorator(){
44          cout << "~Decorator()" << endl;
45      }
46
47      void setComponent(Component * component){
48          _component = component;
49      }
50
51      virtual void operation() override{
52          if(_component != nullptr){
53              _component->operation();
54          }
```

```
    }

private:
    Component *_component;
};

// 具体装饰器A
class ConcreteDecoratorA : public Decorator{
public:
    ConcreteDecoratorA(){
        cout << "ConcreteComponentA()" << endl;
    }

    virtual void operation() override{
        Decorator::operation();
        addBehavior();
    }

    // 添加新功能
    void addBehavior(){
        cout << "为具体构件A扩展的操作" << endl;
    }

    ~ConcreteDecoratorA(){
        cout << "~ConcreteDecoratorA()" << endl;
    }
};

class ConcreteDecoratorB : public Decorator{
public:
    ConcreteDecoratorB(){
        cout << "ConcreteDecoratorB()" << endl;
    }

    virtual void operation() override{
        Decorator::operation();
        addBehavior();
    }

    // 添加新功能
    void addBehavior(){
        cout << "为具体构件B扩展的操作" << endl;
```

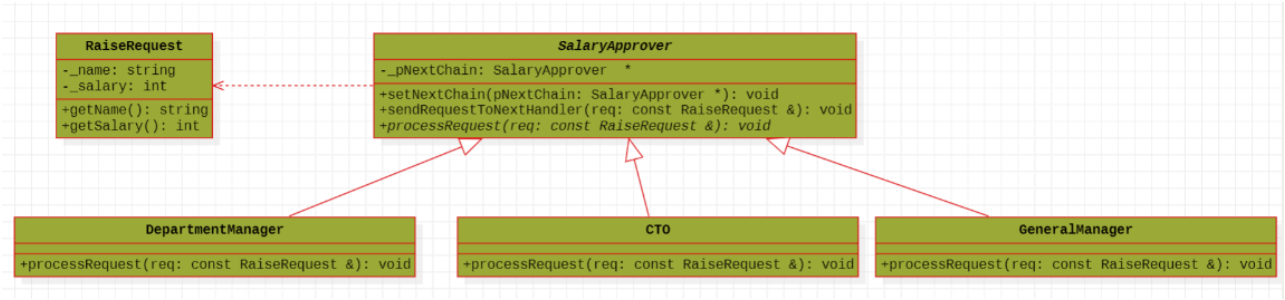
```
97      }
98
99      ~ConcreteDecoratorB(){
100         cout << "~ConcreteDecoratorB()" << endl;
101     }
102 };
103
104 void test(){
105     unique_ptr<Component> pcom(new ConcreteComponent());
106     unique_ptr<Decorator> pda(new ConcreteDecoratorA());
107     unique_ptr<Decorator> pdb(new ConcreteDecoratorB());
108
109     pda->setComponent(pcom.get());
110     pdb->setComponent(pda.get());
111     pdb->operation();
112 }
113
114 int main(){
115     test();
116     return 0;
117 }
```

职责链模式

概述

职责链(Chain Of Responsibility)模式也叫责任链模式，是一种行为型模式，用于将一个请求传递给一个链中的若干对象，哪个对象适合处理这个请求就让哪个对象来处理。职责链看起来与传统数据结构中的"链表"非常类似。引人职责链设计模式的定义(实现意图)：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链(构成对象链)，并沿着这条链传递该请求，直到有一个对象处理它为止。

类图



具体实现

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

class RaiseRequest{
public:

    RaiseRequest(string name, int salary)
    :_name(name)
    ,_salary(salary)
    {
    }

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

    int getSalary() const{
        return _salary;
    }

    ~RaiseRequest(){
    }
private:
    string _name;
    int _salary;
};

class SalaryApprover{
public:
```

```
32
33      SalaryApprover()
34      :_pNextChain(nullptr)
35      {}
36
37      virtual ~SalaryApprover(){}
38
39      void setNextChain(SalaryApprover *pNextChain){
40          _pNextChain = pNextChain;
41      }
42
43      void sendRequestToNextHandler(const RaiseRequest & req){
44          if(_pNextChain != nullptr){
45              _pNextChain->processRequest(req);
46          }else{
47              cout << req.getName() << "的加薪要求为：" <<
    req.getSalary() << "元，但无人能够审批！" << endl;
48          }
49      }
50
51      virtual void processRequest(const RaiseRequest &req) = 0;
52
53  private:
54      SalaryApprover *_pNextChain;
55  };
56
57  class DepartmentManager : public SalaryApprover{
58  public:
59      DepartmentManager(){}
60
61      ~DepartmentManager(){}
62
63      virtual void processRequest(const RaiseRequest &req){
64          int salary = req.getSalary();
65          if(salary <= 1000){
66              cout << req.getName() << "的加薪要求为："
67                   << salary << "元，部门经理审批通过！" << endl;
68          }else{
69              sendRequestToNextHandler(req);
70          }
71      }
72  };
```

```
73
74  class CTO : public SalaryApprover{
75  public:
76      CTO(){}
77      ~CTO(){}
78
79      virtual void processRequest(const RaiseRequest &req){
80          int salary = req.getSalary();
81          if(salary > 1000 && salary <= 5000){
82              cout << req.getName() << "的加薪要求为："
83                   << salary << "元，技术总监审批通过！" << endl;
84          }else{
85              sendRequestToNextHandler(req);
86          }
87      }
88  };
89
90  class GeneralManager : public SalaryApprover{
91  public:
92      GeneralManager(){}
93      ~GeneralManager(){}
94
95      virtual void processRequest(const RaiseRequest &req){
96          int salary = req.getSalary();
97          if(salary > 5000){
98              cout << req.getName() << "的加薪要求为："
99                   << salary << "元，总经理审批通过！" << endl;
100         }else{
101             sendRequestToNextHandler(req);
102         }
103     }
104 };
105
106 void test(){
107     unique_ptr<SalaryApprover> pdm(new DepartmentManager());
108     unique_ptr<SalaryApprover> pcto(new CTO());
109     unique_ptr<SalaryApprover> pgm(new GeneralManager());
110
111     pdm->setNextChain(pcto.get());
112     pcto->setNextChain(pgm.get());
113     pgm->setNextChain(nullptr);
114
```

```
115     RaiseRequest zhangsan("张三", 15000); //张三要求加薪15000
116     RaiseRequest lisi("李四", 3500);      //李四要求加薪3500
117     RaiseRequest wangwu("王五", 800);     //王五要求加薪800
118 
119     // 体现了多态
120     pdm->processRequest(zhangsan);
121     pdm->processRequest(lisi);
122     pdm->processRequest(wangwu);
123 }
124 
125 int main(){
126     test();
127     return 0;
128 }
```