1 创建型模式

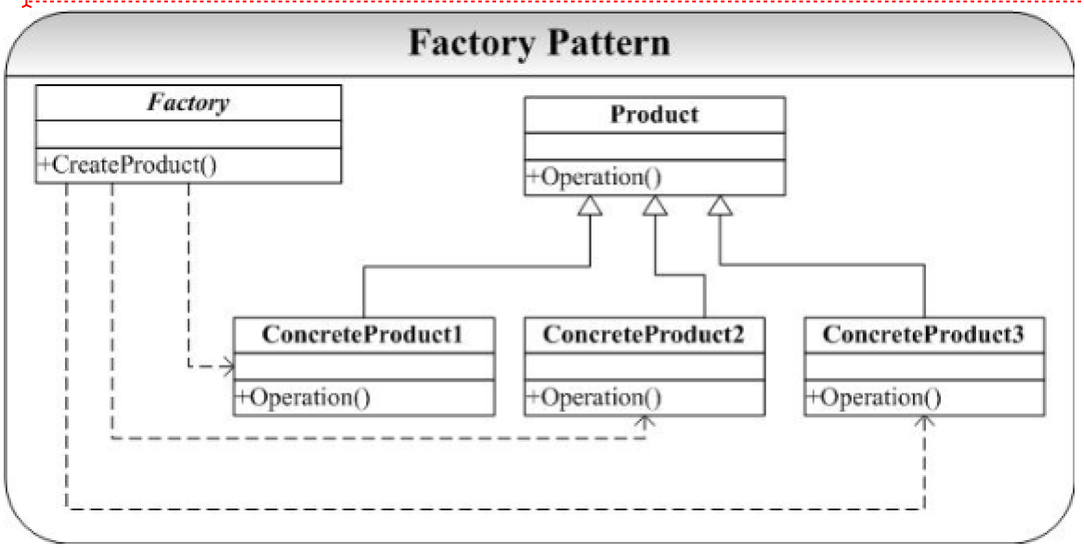
1.1 Factory 模式

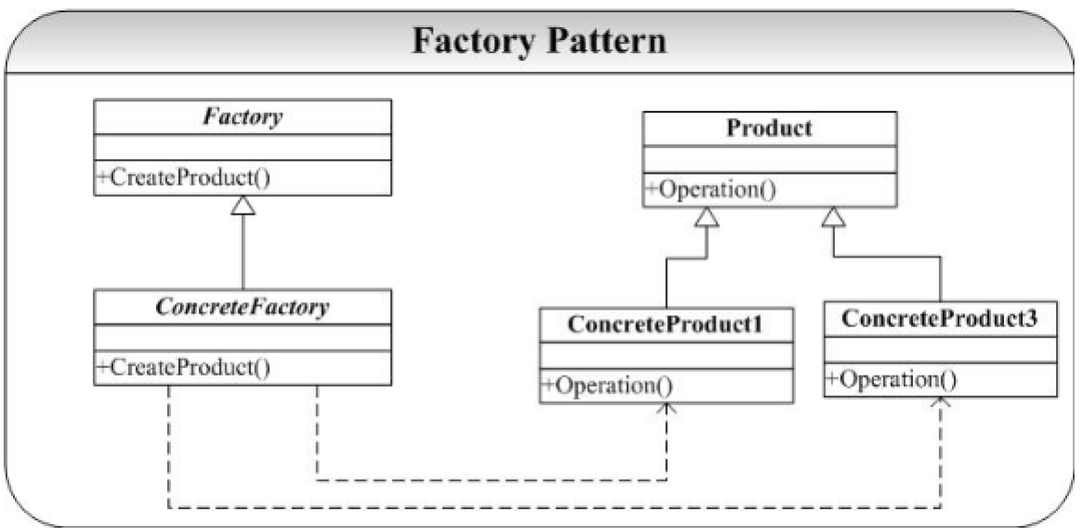
1.1.1、概述

定义一个用于创建对象的接口，让子类决定实例化哪一个类。FactoryMethod 使一个类的实例化延迟到其子类。当一个类不知道它所必须创建的对象的类的时候。当一个类希望由它的子类来指定它所创建的对象的时候。当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

1.3.2、类图

批注[MU1]：好多工厂都生产同一种产品，我们只是面对工厂，由工厂负责给我们提供一个产品类对象。我们只知道得到一个产品类对象，但是不知道这个对象是那个具体类提供的。





1.1.3、模拟实现

Product.h

```
#include<iostream>
using namespace std;
class Product
{
public:
        virtual ~Product()=0;
protected:
        Product();
};

class ConcreteProduct:public Product
{
public:
        ~ConcreteProduct();
        ConcreteProduct();
protected:
private:
};
```

Product.cpp

```
#include "Product.h"

Product::Product(){}
Product::~Product(){}
ConcreteProduct::ConcreteProduct()
{cout<<"产品类对象"<<endl;}

ConcreteProduct::~ConcreteProduct(){}
```

Factory.h

```
//#include "Product.h"
class Product;

class Factory
{
public:
        virtual ~Factory() = 0;
        virtual Product* CreateProduct()
= 0;
protected:
        Factory();
};

class ConcreteFactory1:public Factory
{
public:
        ~ConcreteFactory1();
        ConcreteFactory1();
        Product* CreateProduct();
protected:
private:
};

class ConcreteFactory2:public Factory
{
public:
        ~ConcreteFactory2();
        ConcreteFactory2();
        Product* CreateProduct();
protected:
private:
};
```

Factory.cpp

```
#include "Factory.h"
#include "Product.h"

Factory::Factory(){}
Factory::~Factory(){}

ConcreteFactory1::ConcreteFactory1()
{cout<<"工厂一开动"<<endl;}

ConcreteFactory1::~ConcreteFactory1(){
}

Product*
ConcreteFactory1::CreateProduct()
{return new ConcreteProduct();}

ConcreteFactory2::ConcreteFactory2()
{cout<<"工厂二开动"<<endl;}

ConcreteFactory2::~ConcreteFactory2(){
}
```

```
Product*
ConcreteFactory2::CreateProduct()
{return new ConcreteProduct();}
------
主函数
#include "stdafx.h"
#include "Factory.h"
#include "Product.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
        Factory* fac = new
ConcreteFactory1();
        Product* p =
fac->CreateProduct();
        printf("Hello World!\n");
        return 0;
}
```

1.1.4、讨论

示例代码中给出的是 Factory 模式解决父类中并不知道具体要实例化哪一个具体的子类的问题.知道这个工厂和这个工厂的一系列产品。我们可以用工厂的基类指针去指向工厂类的对象（我们已经明确要那个工厂为我们服务，当然此时也可以不用指针而直接使用这个工厂的的指针或对象来操作。但使用指针可能是可以便于复用和其他复杂的控制吧。）然后通过这个工厂类对象来获得一个产品类对象（想象一下这个工厂有很多个产品类都可以创建同样的对象，我们（类的使用者）不必去判断那个产品类可用，那个正忙，反正我需要一个这种产品的对象，我不用去管这个对象是那个类产生的，但作为类的使用者，一定已经知道这个产品类的功能即行为）。可以通过一个例子来消化一下，我需要一台海尔电脑（产品），很多地方（工厂）对供应（生产）他，我知道三好街（一个具体工厂）供应。好我需要得到一个三好街的对象（销售商），这个销售商可能对应多个店面（产品类），我不必去管这个销售商具体是在那个店（通过那个类）供应我电脑（产品对象），那时他（工厂类对象）的是。但是我的确能够通过他得到一个电脑（特别明确的对象），并且我会使用电脑（明确对象行为）

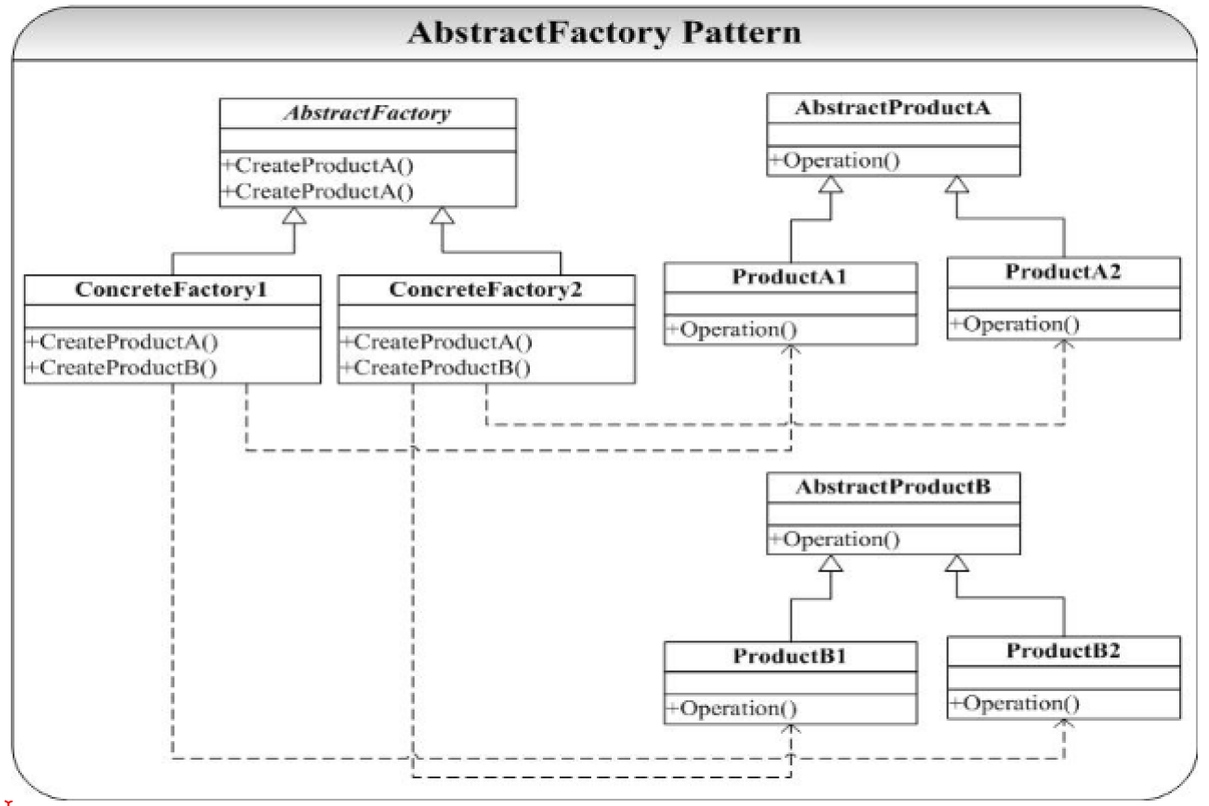
可以通过参数化工厂方法，即给工厂中创建产品对象的方法传递一个参数用以决定是创建具体哪一个具体的Product。

1.2 AbstactFactory 模式

1.2.1、概述

要创建一组相关或者相互依赖的对象，而无需指定它们具体的类。这样当我们要创建一组对象（ProductA1，ProductA2）的时候我们只用维护一个创建对象（ConcreteFactory1），大大简化了维护的成本和工作。

1.2.2、类图



批注[MU2]：有很多的产品，他们属于不同的产品类(抽象基类)。有很多工厂，他们都是一个抽象工厂的派生类，每个工厂都能成产产品中的若干钟。类的用户只需要知道那个厂子产什么产品，然后根据需要做出选择。至于生产的过程，那时工厂的事，工厂负责维护，我们只需要一个指向生产对象的指针，不必去管这个产品是那个厂子产的。当然作为产品的使用者，可以不知道产品是怎么来的，可以不负责产品的出产和维护，但一定要知道产品的使用

1.2.3、模拟实现

Product.h

```
#include <iostream>

using namespace std;


class AbstractProductA
{
public:
    virtual ~AbstractProductA();
    virtual void fun()=0;
protected:
    AbstractProductA();
private:
};
class AbstractProductB
{
public:
```

```
    virtual ~AbstractProductB();
    virtual void fun()=0;
protected:
    AbstractProductB();
private:
};
class ProductA1:public AbstractProductA
{
public:
    ProductA1();
    void fun();
    ~ProductA1();
protected:
private:
};

class ProductA2:public AbstractProductA
{
public:
    ProductA2();
    void fun();
    ~ProductA2();
protected:
private:
};

class ProductB1:public AbstractProductB
{
public:
    ProductB1();
    void fun();
    ~ProductB1();
protected:
private:
};

class ProductB2:public AbstractProductB
{
public:
    ProductB2();
    void fun();
    ~ProductB2();
protected:
private:
```

```
};
-------------------------------------
------
Product.cpp
#include "stdafx.h"
#include "Product.h"

AbstractProductA::AbstractProductA(){}
AbstractProductA::~AbstractProductA(){}
AbstractProductB::AbstractProductB(){}
AbstractProductB::~AbstractProductB(){}

ProductA1::ProductA1()
{
    cout<<"一厂产品 A..."<<endl;
}

ProductA1::~ProductA1(){}

ProductA2::ProductA2()
{
    cout<<"二厂产品 A..."<<endl;
}

void ProductA1::fun()
{
    cout<<"一厂产品 A 的行为"<<endl;
}

void ProductA2::fun()
{
    cout<<"二厂产品 A 的行为"<<endl;
}

ProductA2::~ProductA2(){}

ProductB1::ProductB1()
{
    cout<<"一厂产品 B..."<<endl;
}

ProductB1::~ProductB1(){}

ProductB2::ProductB2()
```

```
{
    cout<<"二厂产品 B..."<<endl;
}

void ProductB1::fun()
{
    cout<<"一厂产品 B 的行为"<<endl;
}

void ProductB2::fun()
{
    cout<<"二厂产品 B 的行为"<<endl;
}

ProductB2::~ProductB2(){}
```

AbstractFactory.h

```
class AbstractProductA;
class AbstractProductB;

class AbstractFactory
{
public:
    virtual ~AbstractFactory();
    virtual AbstractProductA*
CreateProductA()=0;
    virtual AbstractProductB*
CreateProductB()=0;
protected:
    AbstractFactory();
private:
};

class ConcreteFactory1:public
AbstractFactory
{
public:
    ConcreteFactory1();
    ~ConcreteFactory1();
    AbstractProductA* CreateProductA();
    AbstractProductB* CreateProductB();
protected:
private:
```

```
};

                                                AbstractFactory::AbstractFactory(){}
//*                                             AbstractFactory::~AbstractFactory(){}
class                ConcreteFactory2:public    ConcreteFactory1::ConcreteFactory1(){}
AbstractFactory                                 ConcreteFactory1::~ConcreteFactory1(){}
{                                               AbstractProductA*
public:                                         ConcreteFactory1::CreateProductA()
    ConcreteFactory2();                         {return new ProductA1();}
    ~ConcreteFactory2();
    AbstractProductA* CreateProductA();         AbstractProductB*
    AbstractProductB* CreateProductB();         ConcreteFactory1::CreateProductB()
protected:                                      {return new ProductB1();}
private:
};                                              ConcreteFactory2::ConcreteFactory2(){}
--------------------------                      ConcreteFactory2::~ConcreteFactory2(){}
AbstractFactory.cpp
#include "stdafx.h"                             AbstractProductA*
#include "AbstractFactory.h"                    ConcreteFactory2::CreateProductA()
#include "AbstractFactory.h"                    {return new ProductA2();}
#include "Product.h"
#include <iostream>                             AbstractProductB*
using namespace std;                            ConcreteFactory2::CreateProductB()
```

```
{return new ProductB2();}

------------------------

主函数

#include "stdafx.h"

#include "AbstractFactory.h"

#include "Product.h"

#include <iostream>

using namespace std;


int main(int argc, char* argv[])

{

    AbstractProductA* proA;

    AbstractProductB* proB;


    //生成工厂一对象

    AbstractFactory*  cf1  =  new
ConcreteFactory1();

    //通过工厂一的方法 创建产品对象 A

    proA=cf1->CreateProductA();

    //对象 A 的方法

    proA->fun();

    //(new ProductA1)->fun();


    proB=cf1->CreateProductB();

    proB->fun();


    AbstractFactory*  cf2  =  new
ConcreteFactory2();

    proA=cf2->CreateProductA();

    proA->fun();

    proB=cf2->CreateProductB();

    proB->fun();

    printf("Hello World!\n");

    return 0;

}
```

1.2.4、讨论

AbstractFactory 模式是为创建一组（有多类）相关或依赖的对象提供创建接口，而 Factory 模式是为一类对象提供创建接口或延迟对象的创建到子类中实现。AbstractFactory 模式通常都是使用 Factory 模式实现。

工厂模式图一：一个工厂，生产同一个抽象类别的相同或不同的产品。如果产品过多，工厂维护相对复杂。工厂模式图二，对产品再分类，不同小类的产品由不同工厂生产。减少维护成本。

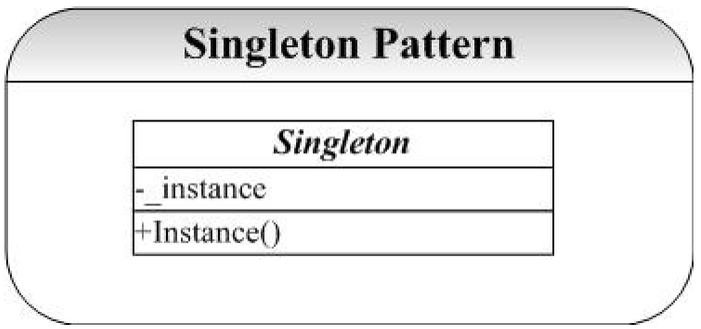
抽象工厂：工厂和产品进行 m*n 组合。更方便调用

1.3、Singleton 模式

1.3.1、概述

保证一个类仅有一个实例，并提供一个访问它的全局访问点。当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

1.3.2、类图



批注[MU3]：关键在于设法保证两点：一、外部无法直接实例化本类。二、在类中生成自己的对象并保证只有一个，这个对象可以通过类直接得到

1.3.3、模拟实现

Singleton.h

```
#include <iostream>

using namespace std;



class Singleton

{

private:

    static Singleton* _instance;//静态私有的本类指针

    Singleton();//私有的构造方法，也可定义为protected:这样保证此类不能被实例化

public:

    static Singleton* Instance();//静态函数，可以被类调用

    virtual ~Singleton();

};

-------------------------------------
```

```
-------

Singleton.cpp

#include "Singleton.h"

Singleton* Singleton::_instance = 0;


Singleton::Singleton()

{

    cout<<"Singleton...."<<endl;

}

Singleton* Singleton::Instance()

{

    if (_instance == 0)

    {

        _instance = new Singleton();

    }

    return _instance;

}

-------------------------------------

--------------------

Main

#include "Singleton.h"

int main(int argc, char* argv[])

{

    //Singleton sgn1;//实例化错误

    Singleton* sgn = Singleton::Instance();//
通过类名调用函数得到一个对象

    printf("Hello World!\n");

    return 0;

}
```

1.3.4、讨论

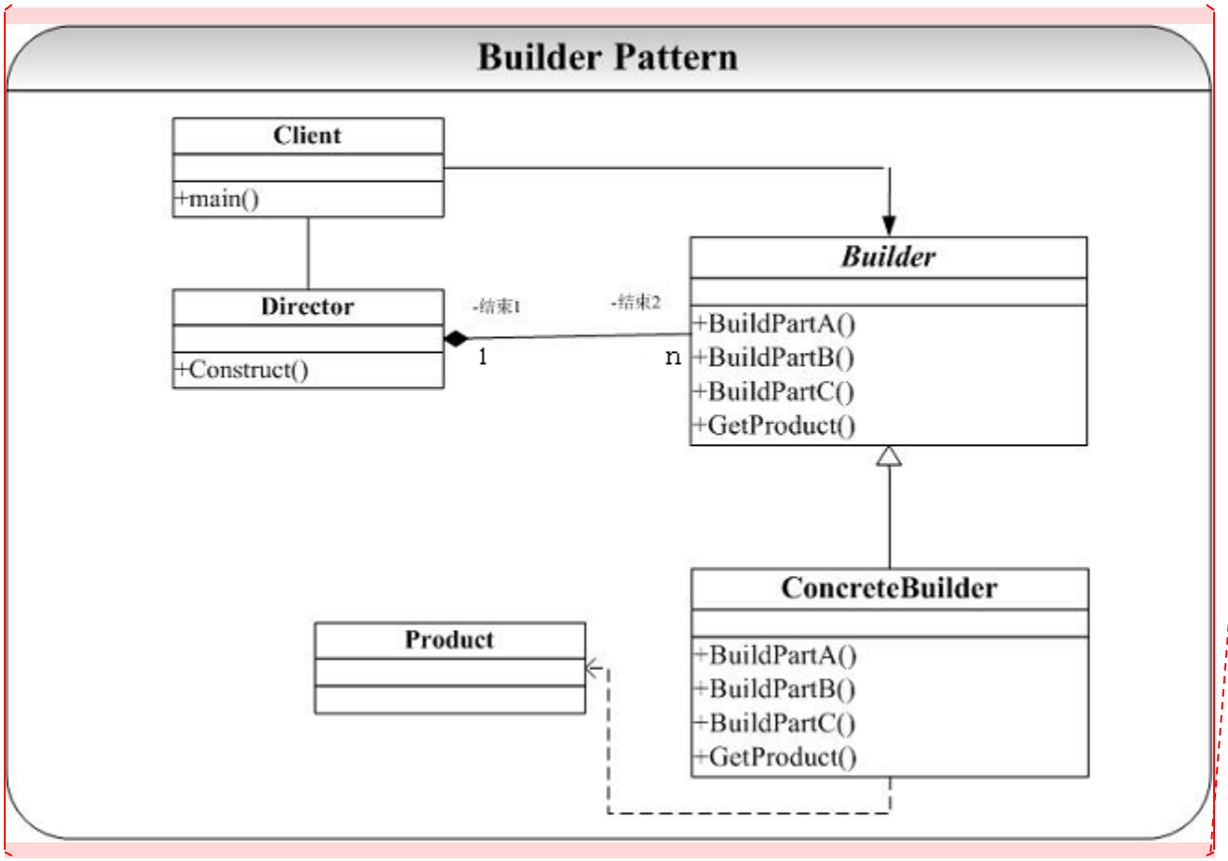
定义一个 Instance 操作，允许客户访问它的唯一实例。Instance 是一个类操作可能负责创建它自己的唯一实例。当一个类能且仅能创建一个实例时，可用此模式

1.4 Builder 模式

1.4.1、概述

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。.当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。当构造过程必须允许被构造的对象有不同的表示时。

1.4.2、类图



批注[MU4]：想象一下，有很多的ConcreteBuilder，他们有着基本相同的创建过程，但每个过程的具体实现都有所不同。用户知道自己需要什么样的产品，不必知道这个产品是有那个Product类创建。只需给指挥者（Director）一个合适的创建工厂参数（ConcreteBuilder）参数，就可以得到

1.4.3、模拟实现

Product.h

```
#include "string"
#include <iostream>
using namespace std;

class Product
{
private:
    string part1,part2,part3;
public:
    Product();
    ~Product();
    void setPart1(string pt1){part1=pt1;}
    void setPart2(string pt2){part1=pt2;}
    void setPart3(string pt3){part1=pt3;}
    void ProducePart();
    void fun();
protected:
```

```
};
----------------------------
Product.cpp
#include "Product.h"
#include <iostream>
using namespace std;

Product::Product()
{//ProducePart();}

Product::~Product(){}
void Product::ProducePart()
{cout<<"build part of product.."<<endl;}
void Product::fun()
{cout<<"产品功能"<<endl;}
-------------------------------------
-------
Builder.h
#include "Product.h"
#include <string>
using namespace std;

class Builder
{
public:
    virtual ~Builder();
virtual void BuildPartA(const string&
buildPara)=0;
virtual void BuildPartB(const string&
buildPara) = 0;
virtual void BuildPartC(const string&
buildPara) = 0;
virtual Product* GetProduct() = 0;
protected:
    Builder();
private:
};

class ConcreteBuilder:public Builder
{
public:
    ConcreteBuilder();
    ~ConcreteBuilder();
    void BuildPartA(const string& buildPara);
```

```
    void BuildPartB(const string& buildPara);
    void BuildPartC(const string& buildPara);
    Product* GetProduct();
protected:
private:
    Product pt;
};
-------------------------------------
-----
Builder.cpp
#include "Builder.h"
#include "Product.h"
Builder::Builder(){}
Builder::~Builder(){}
ConcreteBuilder::ConcreteBuilder(){}
ConcreteBuilder::~ConcreteBuilder(){}

void ConcreteBuilder::BuildPartA(const
string& buildPara)
{cout<<"产 品 创 建 第 一 步 ...
"<<buildPara<<endl;
    pt.setPart1(buildPara);}
void ConcreteBuilder::BuildPartB(const
string& buildPara)
{cout<<"产 品 创 建 第 二 步 ...
"<<buildPara<<endl;
    pt.setPart2(buildPara);}
void ConcreteBuilder::BuildPartC(const
string& buildPara)
{cout<<"产 品 创 建 第 三 步 ...
"<<buildPara<<endl;
    pt.setPart3(buildPara);}

Product* ConcreteBuilder::GetProduct()
{BuildPartA("第一步参数表");
    BuildPartB("第二步参数表");
    BuildPartC("第三步参数表");
    cout<<"成功得到一个产品"<<endl;
    return new Product();}
-------------------------------------
------
Director.h
#include "Product.h"
```

```
class Builder;
class Director
{
public:
    Director(Builder* bld);
    ~Director();
    Product* Construct();
protected:
private:
    Builder* _bld;
};
```

Director.cpp

```
#include "Director.h"
#include "Builder.h"

Director::Director(Builder* bld)
{cout<<"得到相关定制参数，正在生成创建工厂,";
    _bld = bld;}
Director::~Director(){}

Product* Director::Construct()
{cout<<"创建工厂已经生成，开始逐步构建产品"
<<endl;
    return _bld->GetProduct();}
```

Main

```
#include "Builder.h"
#include "Product.h"
#include "Director.h"

int main(int argc, char* argv[])
{
    Product* pro;//接收产品的指针
    //将一个具体的构建者付给指挥者
    Director* d = new Director(new
ConcreteBuilder());
    pro=d->Construct();//调用产生产品
    pro->fun();
    printf("Hello World!\n");
    return 0;
}
```

1.4.4、讨论

生活中有着很多的 Builder 的例子，大学生活就是一个 Builder 模式的最好体验：要完成大学教育，一般将大学教育过程分成 4 个学期进行，因此每个学期可以看作是构建完整大学教育的一个部分构建过程，每个人经过这 4 年的（4 个阶段）构建过程得到的最后的结果不一样，因为可能在四个阶段的构建中引入了很多的参数（每个人的机会和际遇不完全相同）。

GoF 在《设计模式》一书中给出的关于 Builder 模式的意图是非常容易理解、间接的：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示（在示例代码中可以通过传入不同的参数实现这一点）。

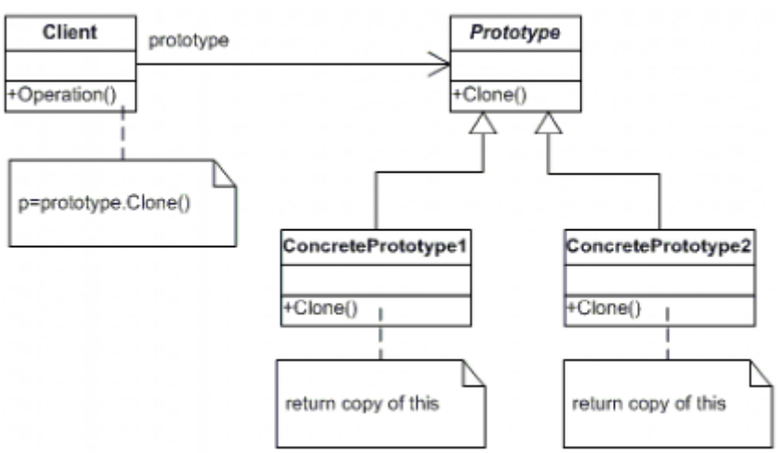
Builder 模式和 AbstractFactory 模式在功能上很相似，因为都是用来创建大的复杂的对象，它们的区别是：Builder 模式强调的是一步步创建对象，并通过相同的创建过程可以获得不同的结果对象，一般来说 Builder 模式中对象不是直接返回的。而在 AbstractFactory 模式中对象是直接返回的，AbstractFactory 模式强调的是为创建多个相互依赖的对象提供一个同一的接口。

1.5 Prototype 模式

1.5.1、概述

通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。当一个系统应该独立于它的产品创建、构成和表示时。当要实例化的类是在运行时刻指定时，例如，通过动态装载。为了避免创建一个与产品类层次平行的工厂类层次时。当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。原型模式其实就是从一个对象再创建另外一个可定制的对象，而且不需知道任何创建的细节。用原型实例指定创建对象的种类，并且通过拷贝这些原型对象创建新的对象。

4.5.2、类图



1.5.3、模拟实现

Prototype.h

```
#include "string"
#include "iostream"
using namespace std;

class Prototype
{
public:
    virtual ~Prototype(){}
    virtual Prototype* Clone()const=0;
    virtual void show()=0;
    void setId(int id){this->id=id;}
    void setName(string name){this->name=name;}
    void setAge(int age){this->age=age;}
    int getId(){return id;}
    string getName(){return name;}
    int getAge(){return age;}
protected:
    //构造函数，为快速构造对象而设
    Prototype(int id1,string name1,int age1):id(id1),name(name1),age(age1){}
    Prototype(){}
    int id;
```

```
    string name;
    int age;
};

class ConcretePrototype:public Prototype
{
public:
    //一个空构造函数
    ConcretePrototype(){}
    //构造函数，为快速构造对象而设
    ConcretePrototype(int id1,string name1,int age1):Prototype(id1,name1,age1){}
    ~ConcretePrototype(){}
    Prototype* Clone() const;

    void show()
    {

    cout<<"id:"<<id<<",name:"<<name<<",age:"<<age<<endl;
    }
protected:
private:
};
```

```
#include "Prototype.h"
Prototype* ConcretePrototype::Clone() const
{cout<<"在此完成克隆操作"<<endl;
return new ConcretePrototype(*this);}
```

Main

```
#include "Prototype.h"

int main(int argc, char* argv[])
{
    /*以下区块为一个测试，测试不用原型模式克隆对象，直接生成对象赋值，
    个人感觉和原型方法区别不是很大。唯一的区别需要知道具体类。抽象基类不能实例化，就算基类不是抽象的，
    仅凭基类以及子类的一个实例（不知道具体子类名）想得到一个与子类对象一样的实例是很困难的*/
    /*
    ConcretePrototype cp1(101,"wang",23);
    //ConcretePrototype cp2;
    //cp2=cp1;
```

```
    //上两句等效于
    ConcretePrototype cp2(cp1);
    cp1.setName("zhang");
    cp1.show(); cp2.show();
    //*/
    Prototype* p = new ConcretePrototype(101,"wang",23);
    /*new ConcretePrototype(101,"wang",23);
```

可以是从其他渠道传递过来的

也就是说，我们无需知道具体的类，只有拥有他的抽象接口，和一个具体的实例对象，我们就可以得到一个与其同样的对象，加以应用。这样，无论在系统运行时需要操作那个具体类，都可以动态制定

```
    */
    Prototype* p1 = p->Clone();
    p->show(); p1->show();
    p->setName("li");
    p->show(); p1->show();
    printf("Hello World!\n");
    return 0;
}
```

1.5.4、讨论

按我的理解，原型引入的根本原因就是在于它可以利用一个原型对象（在这，我指的是实例，而非类），快速地生成一批和原型对象一样的实例。举个例子来说，你有一个类 A 的实例 a (A a=new A()),现在你想生成一个和 a 一样的实例 b,那么，按照原型的定义，你应该可以这样做 b=a.clone()。这样，你就可以得到一个和 a 一模一样的实例 b(即 a 和部 b 的数据成员的值完全一样）。

上面是原型的一个简单说明，那么引入原型有什么好处呢？按我的理解，就是在于：你如果要生成一大批很相像的类的实例时，省得每次去做重复的赋值工作。再举个例子，如果你有一个类 A,它有十个成员变量，现在你打算生成 100 个 A 的实例，而这些实例的变量值大部分相同（比如说七个相同），只有一小部分不一样（比如说三个），那么如果没有 Prototype,那么你就得每次 New 一个 A 的对像，然后赋值，这样，你要重复 100 次同样的七个变量的赋值工作，显然，这样很麻烦。现在你有了原型，那么问题就简单了，你只要生成一个 A 的实例，再通过 clone 来生成其它的实例，然后再一一修改其它实例不同的地方。

2.结构型模式

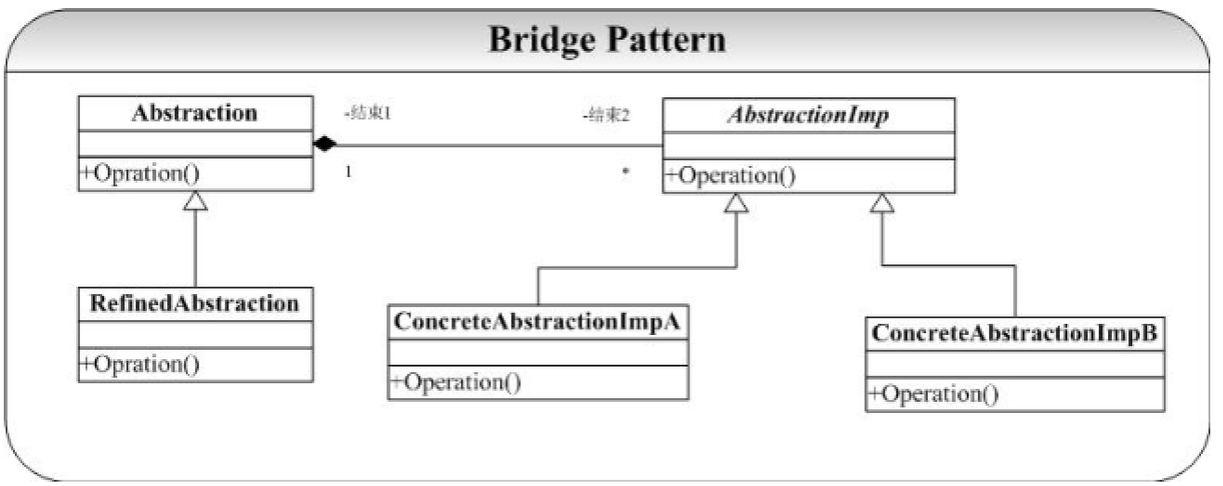
2.1 Bridge 模式

2.1.1、概述

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。例如,一杯咖啡为例,子类实现类为四个：中杯加奶、大杯加奶、 中杯不加奶、大杯不加奶。但是，我们注意到：上面四个子类中有概念重叠，可从另外一个角度进行考虑，这四个类实际是两个角色的组合：抽象 和行为，其中抽象为：中杯和大杯；行为为：加奶 不加奶（如加橙汁 加苹果汁）。

实现四个子类在抽象和行为之间发生了固定的绑定关系，如果以后动态增加加葡萄汁的行为，就必须再增加两个类：中杯加葡萄汁和大杯加葡萄汁。显然混乱,扩展性极差。那我们从分离抽象和行为的角度，使用 Bridge 模式来实现。

2.1.2、类图



2.1.3、模拟实现

Abstraction.h

```
#include "string"
#include "AbstractionImp.h"
#include "Bridge.h"
```

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

class AbstractionImp;
```

```
//活动 比如游玩、学习等
class Abstraction
{
public:
    virtual ~Abstraction(){}
    virtual void Operation()=0;
protected:
    Abstraction(){}
private:
};

/*一个具体的活动.
 注意此活动的实现需要借助于一个具体的方案（比如有 m 种可选方案）作为构造参数。具体方案的实施并不在这里。
 这样做的好处是一旦产生一个新的活动，我们只需要些一个类，而不是 m 个。
*/
class RefinedAbstraction:public Abstraction
{
public:
    //构造函数
    /*根据需要，可以在此通过 Bridge 获得 AbstractionImp 实例指针_imp。
        */
    RefinedAbstraction(){}
    ~RefinedAbstraction(){}
    //方案的实施调用
    void Operation()
    {
        _imp=Bridge::getActionImp();
        cout<<"活动一采用";
        _imp->Operation();
    }
protected:
private:
    Bridge* bridge;
    AbstractionImp* _imp;
};
-----------------------
AbstractionImp.h
#include "iostream"
#include "string"
using namespace std;
```

```
class AbstractionImp
{
public:
    virtual ~AbstractionImp(){}
    virtual void Operation()=0;
protected:
    AbstractionImp(){}
private:
};

class ConcreteAbstractionImp:public AbstractionImp
{
public:
    ConcreteAbstractionImp(){}
    ~ConcreteAbstractionImp(){}
    virtual void Operation()
    {
        cout<<"默认方案...."<<endl;
    }
protected:
private:
};

class ConcreteAbstractionImpA:public AbstractionImp
{
public:
    ConcreteAbstractionImpA(){}
    ~ConcreteAbstractionImpA(){}
    virtual void Operation()
    {
        cout<<"方案 A...."<<endl;
    }
protected:
private:
};

class ConcreteAbstractionImpB:public AbstractionImp
{
public:
    ConcreteAbstractionImpB(){}
```

```
    ~ConcreteAbstractionImpB(){}
    virtual void Operation()
    {
        cout<<"方案 B...."<<endl;
    }
protected:
private:
};
------------------------------
Bridge.h
#include "AbstractionImp.h"

class Bridge
{
public:
    virtual ~Bridge(){};
    static void setActionImp(AbstractionImp*
actionImp1)
    {
        actionImp=actionImp1;
    }
    static AbstractionImp* getActionImp()
    {
        if(actionImp==0)
            actionImp = new
ConcreteAbstractionImp();
        return actionImp;
    }
    static Bridge* getBridge()
    {
        if(bridge==0)
        {
            bridge= new Bridge();
        }
        return bridge;
    }
private:
    Bridge()
    {
        actionImp = new
ConcreteAbstractionImp();
    }
    static AbstractionImp* actionImp;
    static Bridge* bridge;
```

```
};

//对于单态模式此句必不可少。如果有对应 cpp 文件，此句必须写在 cpp 文件中

Bridge* Bridge::bridge=0;

AbstractionImp* Bridge::actionImp=0;

---------------------------------

Main

#include "stdafx.h"

#include "Abstraction.h"

#include "AbstractionImp.h"

#include "Bridge.h"

#include <iostream>

using namespace std;


int main(int argc, char* argv[])

{

    /*Bridge 采用单态模式，里面所以方法都为静态方法。这样无论是否生成 Bridge 类对象指针。都可以使用

    Bridge* bridge   = Bridge::getBridge();

                    bridge->setActionImp(new ConcreteAbstractionImpA());

                Abstraction*   action   =   new RefinedAbstraction();

    action->Operation();*/

    //设置一个新的活动

            Abstraction*   action   =   new RefinedAbstraction();

    //不设置方案则采用默认方案

    action->Operation();

    //通过桥设置方案

    Bridge::setActionImp(new ConcreteAbstractionImpA());

    action->Operation();

    printf("Hello World!\n");

    return 0;

}
```

2.1.4、讨论

这里说的意思不是让抽象基类与具体类分离，而是现实系统可能有多角度分类，每一种分类都有可能变化，那么把这种多角度分离出来让它们独立变化，减少它们之间的耦合性，即如果继承不能实现“开放－封闭原则”的话，就应该考虑用桥接模式。比如我们的例子，活动可能有很多，方案也可能有很多。那么不同的活动不同的方案交叉起来就有很多的可能。如果活动与方案再动态变化，那么情况就十分复杂。所以一定要实现活动和方案的分离。

在用户类中（例子中主函数），给桥（Bridge）设置一个方案。每个活动都要到桥去取方案。这样，每个活动和方案的定义与更改都是相互独立的。我们只需要知道任意一对活动、方案，借助桥就可以将他们联系起来

2.2 Adapter 模式

2.2.1、概述

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。Adapter 模式解决的问题在生活中经常会遇到：比如我们有一个 Team 为外界提供 S 类服务，但是我们 Team 里面没有能够完成此项人物的 member，然后我们得知有 B 可以完成这项服务（他把这项人物重新取了个名字叫 S'，并且他不对外公布他的具体实现）。为了保证我们对外的服务类别的一致性（提供 S 服务），我们有以下两种方式解决这个问题：

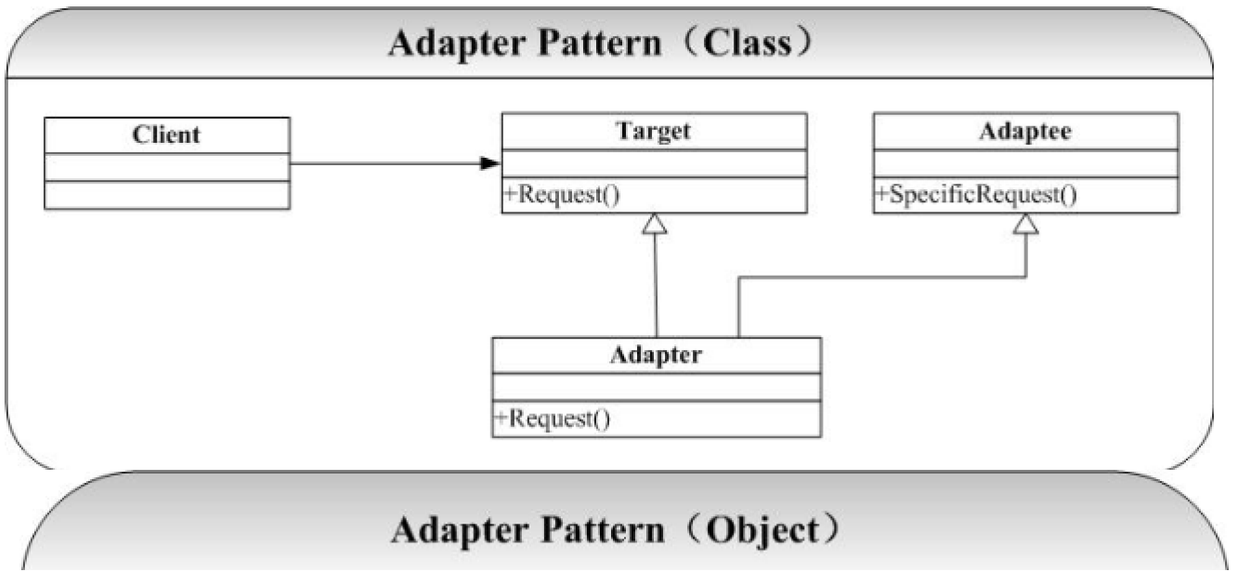
1）把 B 君直接招安到我们 Team 为我们工作，提供 S 服务的时候让 B 君去办就是了；

2）B 君可能在别的地方有工作，并且不准备接受我们的招安，于是我们 Team 可以想这样一种方式解决问题：我们安排 C 君去完成这项任务，并做好工作（Money:））让 C 君工作的时候可以向 B 君请教，因此 C 君就是一个复合体（提供 S 服务，但是是 B 君的继承弟子）。

实际上在软件系统设计和开发中，这种问题也会经常遇到：我们为了完成某项工作购买了一个第三方的库来加快开发。这就带来了一个问题：我们在应用程序中已经设计好了接口，与这个第三方提供的接口不一致，为了使得这些接口不兼容的类（不能在一起工作）可以在一起工作了，Adapter 模式提供了将一个类（第三方库）的接口转化为客户（购买使用者）希望的接口。

在上面生活中问题的解决方式也就正好对应了 Adapter 模式的两种类别：类模式和对象模式。

2.2.2、类图



2.2.3、模拟实现

2.2.3.1、类适配

Adapter.h

```
#include <iostream>

using namespace std;


//模拟原始目标类，通过 Request 实现功能。但是我们需要一个附加功能第三方类已经实现，但其名称为 SpecificRequest

class Target

{

public:

    Target(){}

    virtual ~Target(){}

    virtual void Request()

    {

        cout<<"我们的目标类方法，方法名称为 Request"<<endl;

    };

protected:

private:

};

/*模拟第三方类，里面 SpecificRequest 是我们需要的。我们期望给用户提供统一接口，都通过 Request 方法*/

class ThirdParty

{

public:

    ThirdParty(){}

    ~ThirdParty(){}

    void SpecificRequest()

    {

        cout<<"我们需要的第三方类方法，方法名称为 SpecificRequest" <<endl;

    };

protected:

private:

};


//继承目标类，则其指针可以指向适配器对象，继承第三方类则适配器可以调用第三方类方法

class Adapter:public Target,private ThirdParty

{
```

```
public:
    Adapter(){}
    ~Adapter(){}
    void Request()
    {
        this->SpecificRequest();
    };
protected:
private:
};
```

Main

```
#include "Adapter.h"

int main(int argc, char* argv[])
{
    //目标类指针可以指向其子类（适配器）对象
    Target* adt = new Adapter();
    //通过适配器调用第三方方法。
    adt->Request();
    //目标类指针当然可以指向本类对象
    adt=new Target();
    //直接调用本类方法
    adt->Request();
    printf("Hello World!\n");
    return 0;
}
```

2.2.3.2、对象适配模式

Adapter.h

```
#include <iostream>
using namespace std;
/*模拟原始目标类，通过 Request 实现功能。但是我们需要一个附加功
能第三方类已经实现，但其名称为 SpecificRequest
*/
class Target
{
public:
    Target(){}
    virtual ~Target(){}
    virtual void Request()
    {
        cout<<"我们的目标类方法，方法名称为
```

```
Request"<<endl;
    };
protected:
private:
};
/*模拟第三方类接口，里面抽象方法 SpecificRequest 是我们需要的。他的不同实现可能具有不同功能*/
class ThirdPartyInterface
{
public:
    ThirdPartyInterface(){}
    ~ThirdPartyInterface(){}
    virtual void SpecificRequest()=0;
protected:
private:
};
/*模拟第三方类，里面 SpecificRequest 是我们需要的。我们期望给用户提供统一接口，都通过 Request 方法*/
class ThirdPartyOne:public ThirdPartyInterface
{
public:
    ThirdPartyOne(){}
    ~ThirdPartyOne(){}
    void SpecificRequest()
    {
        cout<<"我们需要的第三方类方法，方法名称为 SpecificRequest" <<endl;
    }
};

//继承目标类，则其指针可以指向适配器对象
class Adapter:public Target
{
public:
    Adapter(ThirdPartyInterface* thirdParty)//:Target(),_thirdParty(thirdParty){}
    {
        this->_thirdParty = thirdParty;
    }
    ~Adapter(){}
    void Request()
    {
```

```
        _thirdParty->SpecificRequest();
    }
protected:
private:
    ThirdPartyInterface* _thirdParty;
};
----------------------------
Main
#include "Adapter.h"


int main(int argc, char* argv[])
{
    //定义第三方类的一个接口
    ThirdPartyInterface* thirdPartyInterface;
    //第三方类接口指向第三方类的一个实例，此实例可换
    thirdPartyInterface = new ThirdPartyOne;
    //目标类指针指向适配器则调用第三方类方法
    Target* tgt= new Adapter(thirdPartyInterface);
    //第三方类方法，注意我们调用时用的是我们熟悉的目标类方法 Request
    tgt->Request();
    //目标类指针指向本类对象，则调用本类方法
    tgt = new Target();
    //调用本类方法
    tgt->Request();
    printf("Hello World!\n");
    return 0;
}
```

2.2.4、讨论

类适配模式：我们需要知道源目标类（我们肯定知道的），适配器类（我们必须知道的）。适配器类继承目标类和第三方类。当目标类指针指向适配器类对象时，可以调用第三方方法；当目标类指针指向目标类对象时，调用本类方法。这种适配器是死的，和固定的第三方类捆绑在一起。如果这个第三方方法不合适，或有新的解决方法，必须修改适配器类。

对象适配器模式：我们需要知道源目标类（我们肯定知道的），适配器类（我们必须知道的），第三方类接口以及我们要应用的第三方类。适配器继承目标类，并内置第三方接口对象。这样可以通过适配器集成不同的第三发类。方便我们扩充修改。

附网络言语：在 Adapter 模式的两种模式中，有一个很重要的概念就是接口继承和实现继承的区别和联系。接口继承和实现继承是面向对象领域的两个重要的概念，接口继承指的是通过继承，子类获得了父类的接口，而实现继承指的是通过继承子类获得了父类的实现（并不统共接口）。在 C++中的 public 继承

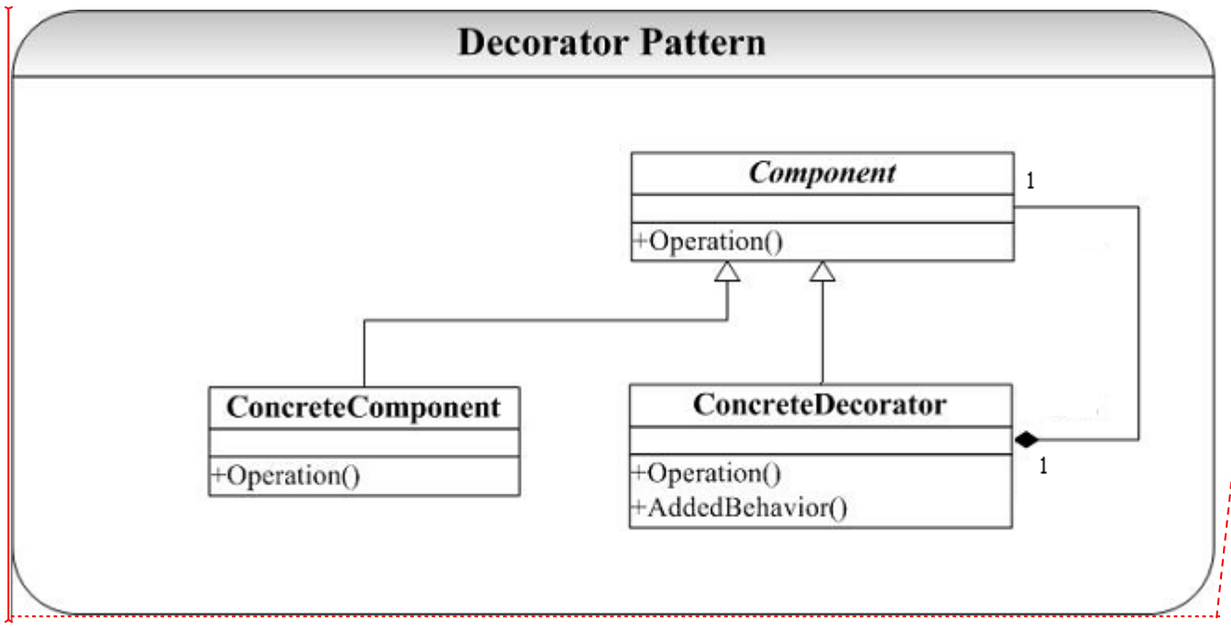
既是接口继承又是实现继承，因为子类在继承了父类后既可以对外提供父类中的接口操作，又可以获得父类的接口实现。当然我们可以通过一定的方式和技术模拟单独的接口继承和实现继承，例如我们可以通过 private 继承获得实现继承的效果（private 继承后，父类中的接口都变为 private，当然只能是实现继承了。），通过纯抽象基类模拟接口继承的效果，但是在 C++中 pure virtual function 也可以提供默认实现，因此这是不纯正的接口继承，但是在 Java 中我们可以 interface 来获得真正的接口继承了。

2.3 Decorator 模式

2.3.1、概述

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

2.3.2、类图



批注[MU5]：原对象类和装饰类都基于一个共同接口。装饰类里维护一个接口（用于指向不同源对象）。装饰类的方法里，我们可以根据需要添加适当信息，然后调用原对象方法。

2.3.3、模拟实现

Decorator.h

```
#include <iostream>

using namespace std;


//公共接口
class Component
{
public:
    virtual ~Component();
    virtual void Operation();
```

```
protected:
    Component();
private:
};

//原始操作
class ConcreteComponent:public Component
{
public:
    ConcreteComponent();
    ~ConcreteComponent();
    void Operation()
    {
        cout<<"原操作功能..."<<endl;
    }

protected:
private:
};

//附加 A 操作
class DecoratorA:public Component
{
public:
    DecoratorA(Component* com)
    {
        this->_com = com;
    }
    virtual ~DecoratorA()
    {
        delete _com;
    }
    void Operation()
    {
        this->AddedBehavior();
        _com->Operation();
    }
    void AddedBehavior()
    {
        cout<<"附加操作 A...."<<endl;
    }
protected:
    Component* _com;
private:
```

```
};

//附加 B 操作
class DecoratorB:public Component
{
public:
    DecoratorB(Component* com)
    {
        this->_com = com;
    }
    virtual ~DecoratorB()
    {
        delete _com;
    }
    void Operation()
    {
        _com->Operation();
        this->AddedBehavior();
    }
    void AddedBehavior()
    {
        cout<<"附加操作 B...."<<endl;
    }
protected:
    Component* _com;
private:
};
----------------------
Decorator.cpp
#include "Decorator.h"
Component::Component(){}
Component::~Component() {}
void Component::Operation() {}
ConcreteComponent::ConcreteComponent()
{}
ConcreteComponent::~ConcreteComponent()
{}
--------------------------------
Main
#include "Decorator.h"
int main(int argc, char* argv[])
{
    //一个公共接口
    Component* com;
```

```
    //指向原始操作
    com= new ConcreteComponent();
    com->Operation();
    //指向 A 装饰划线重新打印
    cout<<"--------------------------
-----------"<<endl;
    com = new DecoratorA(com);
    com->Operation();
    //指向 B 装饰划线重新打印
    cout<<"--------------------------
-----------"<<endl;
    com = new DecoratorB(com);
    com->Operation();
    return 0;
}
```

2.3.4、讨论

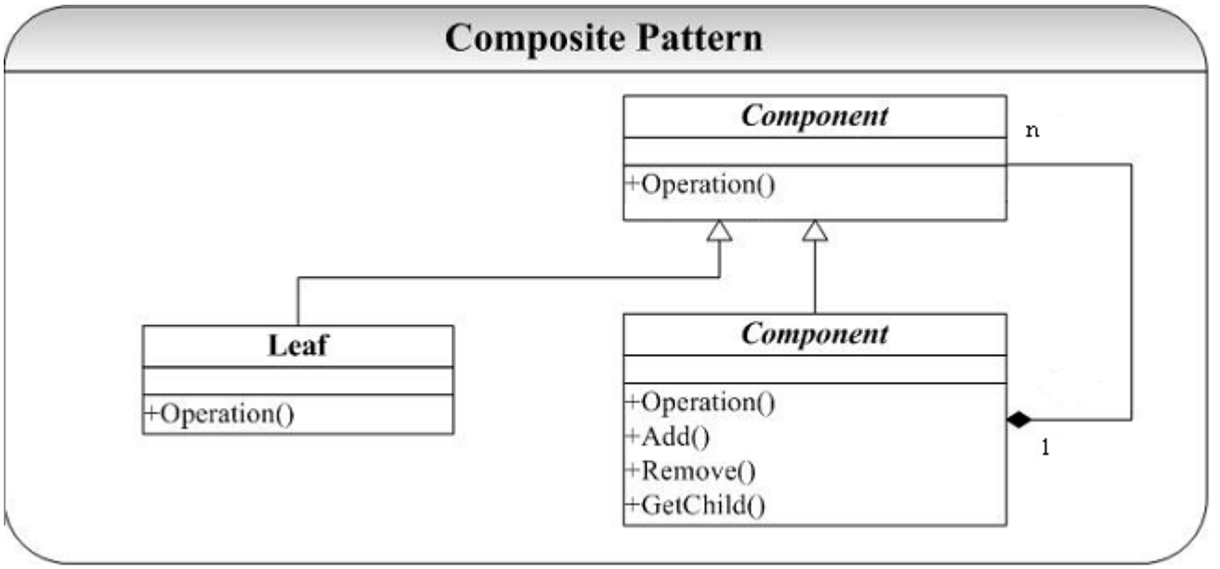
ConcreteComponent 和 Decorator 需要有同样的接口，因此 ConcreteComponent 和 Decorator 有着一个共同的父类。这里有人会问，让 Decorator 直接维护一个指向 ConcreteComponent 引用（指针）不就可以达到同样的效果，答案是肯定并且是否定的。肯定的是你可以通过这种方式实现，否定的是你不要用这种方式实现，因为通过这种方式你就只能为这个特定的 ConcreteComponent 提供修饰操作了，当有了一个新的 ConcreteComponent 你又要去新建一个 Decorator 来实现。但是通过结构图中的 ConcreteComponent 和 Decorator 有一个公共基类，就可以利用 OO 中多态的思想来实现只要是 Component 型别的对象都可以提供修饰操作的类，这种情况下你就算新建了 100 个 Component 型别的类 ConcreteComponent，也都可以由 Decorator 一个类搞定。这也正是 Decorator 模式的关键和威力所在了。当然如果你只用给 Component 型别类添加一种修饰，则 Decorator 这个基类就不是很必要了。

2.4 Composite 模式

2.4.1、概述

将对象组合成树形结构以表示"部分-整体"的层次结构。"Composite 使得用户对单个对象和组合对象的使用具有一致性。" .你想表示对象的部分-整体层次结构。你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

2.4.2、类图



2.4.3、模拟实现

Component.h

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

Composite.h

```
#include "Component.h"
#include <vector>
using namespace std;

class Composite:public Component
{
public:
    Composite(){}//vector<Component*>::iterator itend = comVec.begin();
    ~Composite(){}
public:
    //调出所有操作
    void Operation()
```

```
    {
        vector<Component*>::iterator comIter = comVec.begin();
        for(;comIter != comVec.end();comIter++)
        {
            (*comIter)->Operation();
        }
    }
//添加组件
    void Add(Component* com)
    {
        comVec.push_back(com);
    }
    //移除组件
    void Remove(Component* com)
    {
        comVec.erase(&com);
    }
    //得到组件
    Component* GetChild(int index)
    {
        return comVec[index];
    }
protected:
private:
    vector<Component*> comVec;
};
```

Leaf.h

```
#include "Component.h"
#include <iostream>
using namespace std;

class LeafA:public Component
{
public:
    LeafA(){}
    ~LeafA(){}
    void Operation()
    {
        cout<<"组成操作 A....."<<endl;
    }
protected:
```

```
private:
};

class LeafB:public Component
{
public:
    LeafB(){}
    ~LeafB(){}
    void Operation()
    {
        cout<<"组成操作 B....."<<endl;
    }
protected:
private:
};
------------------------------
```

main

```
#include "Component.h"
#include "Composite.h"
#include "Leaf.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    Composite* com=new Composite();
    com->Add(new LeafA());
    com->Add(new LeafB());
    com->Operation();
    Component* ll = com->GetChild(0);
    ll->Operation();
    //printf("Hello World!\n");
    return 0;
}
```

2.4.4、讨论

Composite 模式和 Decorator 模式有着类似的结构图，但是 Composite 模式旨在构造类，而 Decorator 模式重在不生成子类即可给对象添加职责。Decorator 模式重在修饰，而 Composite 模式重在表示。如果你想要创建层次结构，并可以在其中以相同的方式对待所有元素，那么组合模式就是最理想的选择。本章使用了一个文件系统的例子来举例说明了组合模式的用途。在这个例子中，文件和目录都执行相同的接口，这是组合模式的关键。通过执行相同的接口，你就可以用相同的方式对待文件和目录，从而实现将文件或者目录储存为目录的子级元素。

2.5 Flyweight 模式

2.5.1、概述

运用共享技术有效地支持大量细粒度的对象。当都具备下列情况时，使用 Flyweight 模式：

1.一个应用程序使用了大量的对象。

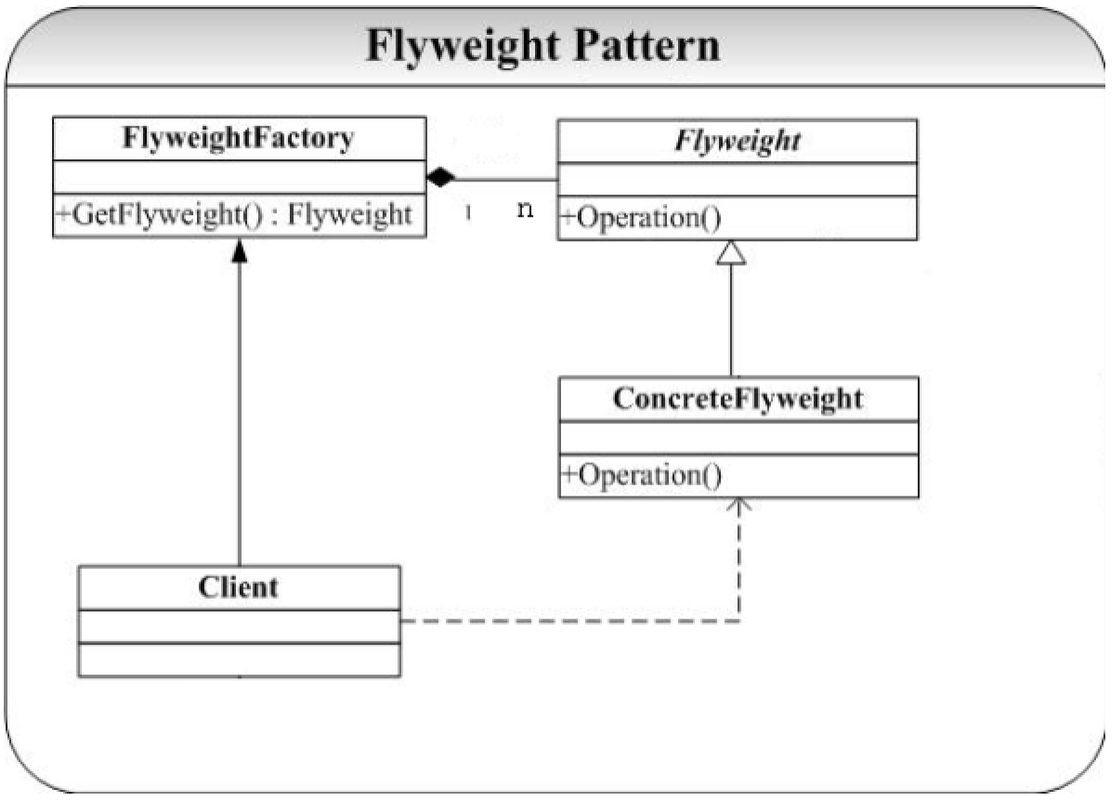
2.完全由于使用大量的对象，造成很大的存储开销。

3.对象的大多数状态都可变为外部状态。

4.如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

5.应用程序不依赖于对象标识。由于 Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

2.5.2、类图



2.5.3、模拟实现

Flyweight.h

```
#include <iostream>
```

```
using namespace std;
#include "string"
using namespace std;

class Flyweight
{
public:
    virtual ~Flyweight(){}
    virtual void Operation(){}
    //返回本类对象的描述
    string GetIntrinsicState()
    {
        return this->_intrinsicState;
    }
protected:
    //本类对象的构造器，构造一个对象并设置描述
    Flyweight(string intrinsicState)
    {
        this->_intrinsicState = intrinsicState;
    }
private:
    string _intrinsicState;//描述语句
};

class ConcreteFlyweight:public Flyweight
{
public:
    ConcreteFlyweight(string intrinsicState):Flyweight(intrinsicState)
    {
        cout<<"创 建 了 "<<intrinsicState<<"元素"<<endl;
    }
    ~ConcreteFlyweight(){}
    void Operation()
    {
        cout<<this->GetIntrinsicState()<<"元素的相关操作，可以定义"<<endl;
    }
protected:
private:
};
```

FlyweightFactory.h

```
#include "Flyweight.h"
#include <iostream>
//#include <cassert>
#include <string>
#include <vector>
using namespace std;

/*cassert 为原 c 语言 cassert 头文件，
编写代码时，我们总是会做出一些假设，断言就是用于在代码中捕捉这些假设，
可以将断言看作是异常处理的一种高级形式。断言表示为一些布尔表达式，
程序员相信在程序中的某个特定点该表达式值为真。可以在任何时候启用和禁用断言验证，
因此可以在测试时启用断言，而在部署时禁用断言。同样，程序投入运行后，
最终用户在遇到问题时可以重新起用断言。
*/

class FlyweightFactory
{
public:
    FlyweightFactory(){}
    ~FlyweightFactory(){}
    Flyweight* GetFlyweight(const string& key)
    {
        vector<Flyweight*>::iterator it = _fly.begin();
        for (; it != _fly.end();it++)
        { //找到了，就一起用，^_^
            if ((*it)->GetIntrinsicState() == key)
            {
                cout<<"在 库 中 检 索 到 "<<key<<"元素"<<endl;
                return *it;
            }
        }
        //没有找到则创建
        Flyweight* fn = new ConcreteFlyweight(key);
        //放入库中
        _fly.push_back(fn);
        return fn;
    }
protected:
private:
    vector<Flyweight*> _fly;
```

```
};                                              Flyweight*          fw1          =
------------------------------          fc->GetFlyweight("hello");
Main                                            Flyweight*          fw2          =
#include "Flyweight.h"                  fc->GetFlyweight("world!");
#include "FlyweightFactory.h"                   Flyweight*          fw3          =
                                                fc->GetFlyweight("hello");
int main(int argc, char* argv[])                //printf("Hello World!\n");
{                                               return 0;
    FlyweightFactory*       fc      =       new     }
FlyweightFactory();
```

2.5.4、讨论

在面向对象系统的设计何实现中，创建对象是最为常见的操作。这里面就有一个问题：如果一个应用程序使用了太多的对象，就会造成很大的存储开销。特别是对于大量轻量级（细粒度）的对象，比如在文档编辑器的设计过程中，我们如果为没有字母创建一个对象的话，系统可能会因为大量的对象而造成存储开销的浪费。例如一个字母“a”在文档中出现了 100000 次，而实际上我们可以让这一万个字母“a”共享一个对象，当然因为在不同的位置可能字母“a”有不同的显示效果（例如字体和大小等设置不同），在这种情况我们可以为将对象的状态分为“外部状态”和“内部状态”，将可以被共享（不会变化）的状态作为内部状态存储在对象中，而外部对象（例如上面提到的字体、大小等）我们可以在适当的时候将外部对象最为参数传递给对象（例如在显示的时候，将字体、大小等信息传递给对象）。

2.6 Facade 模式

2.6.1、概述

为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

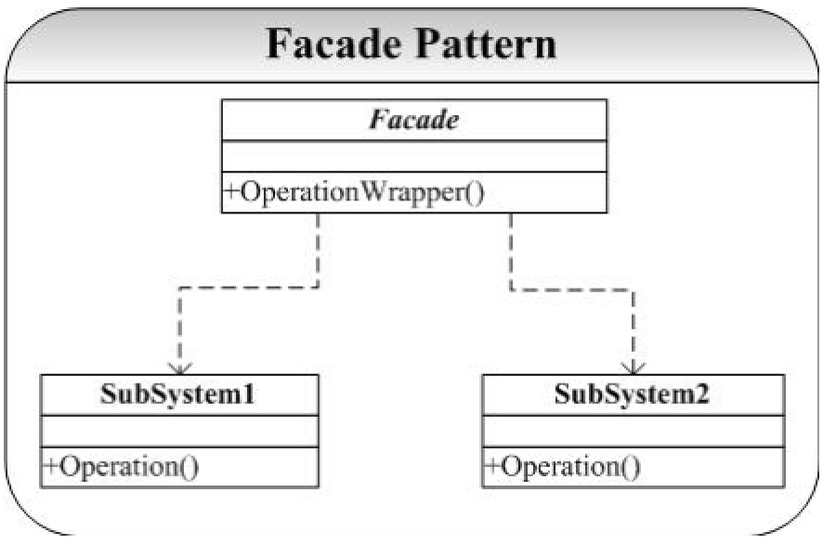
1.当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade 可以提供一个简单的缺省视图，这一视图

对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过 facade 层。

2.客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

3.当你需要构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过 facade 进行通讯，从而简化了它们之间的依赖关系。

2.6.2、类图



2.6.3、模拟实现

Façade.h

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

class Subsystem1
{
public:
    Subsystem1(){}
    ~Subsystem1(){}
    void Operation()
    {
        cout<<"对象一的行为.."<<endl;
    }
protected:
private:
};

class Subsystem2
```

```
{
public:
    Subsystem2(){}
    ~Subsystem2(){}
    void Operation()
    {
        cout<<"对象二的行为.."<<endl;
    }
protected:
private:
};

class Facade
{
public:
    Facade()
    {
        cout<<"在构造的过程中生成两个小对象，当然可以根据需要设计不同的 Facade "<<endl;
        this->_subs1 = new Subsystem1();
        this->_subs2 = new Subsystem2();
    }
    ~Facade()
    {
        delete _subs1;
        delete _subs2;
    }
    void OperationWrapper()
    {
        this->_subs1->Operation();
        this->_subs2->Operation();
    }
protected:
private:
    Subsystem1* _subs1;
    Subsystem2* _subs2;
};
--------------------------------
Main
int main(int argc, char* argv[])
{
    Facade* f = new Facade();
    f->OperationWrapper();
```

```
    printf("Hello World!\n");                    }

    return 0;
```

2.6.4、讨论

举一个生活中的小例子，大凡开过学或者毕过业的都会体会到这样一种郁闷：你要去 n 个地方办理 n 个手续（现在大学合并后就更加麻烦，因为可能那 n 个地方都隔的比较远）。但是实际上我们需要的就是一个最后一道手续的证明而已，对于前面的手续是怎么办的、到什么地方去办理我们都不感兴趣。

实际上在软件系统开发中也经常回会遇到这样的情况，可能你实现了一些接口（模块），而这些接口（模块）都分布在几个类中（比如 A 和 B、C、D）：A 中实现了一些接口，B 中实现一些接口（或者 A 代表一个独立模块，B、C、D 代表另一些独立模块）。然后你的客户程序员（使用你设计的开发人员）只有很少的要知道你的不同接口到底是在那个类中实现的，绝大多数只是想简单的组合你的 A－D 的类的接口，他并不想知道这些接口在哪里实现的。

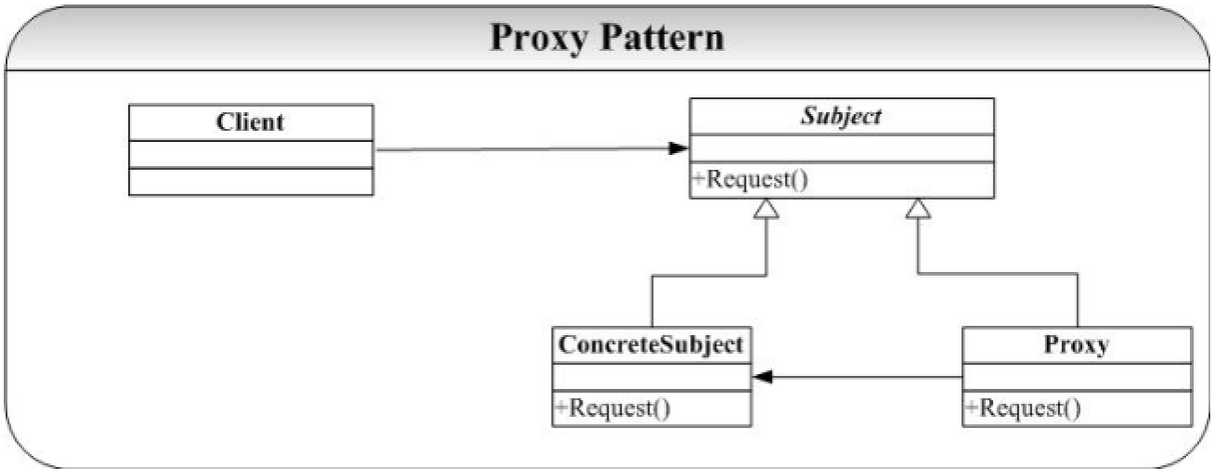
这里的客户程序员就是上面生活中想办理手续的郁闷的人！在现实生活中我们可能可以很快想到找一个人代理所有的事情就可以解决你的问题（你只要维护和他的简单的一个接口而已了！），在软件系统设计开发中我们可以通过一个叫做 Façade 的模式来解决上面的问题。

2.7 Proxy 模式

2.7.1、概述

为其他对象提供一种代理以控制对这个对象的访问。1.远程代理（RemoteProxy）为一个对象在不同的地址空间提供局部代表。2.虚代理（VirtualProxy）根据需要创建开销很大的对象。3.保护代理（ProtectionProxy）控制对原始对象的访问。4.智能指引（SmartReference）取代了简单的指针，它在访问对象时执行一些附加操作。

2.7.2、类图



2.7.3、模拟实现

Proxy.h

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

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

```
class ConcreteSubject:public Subject
{
public:
    ConcreteSubject(){}
    ~ConcreteSubject(){}
    void Request()
    {

    cout<<"ConcreteSubject.....request    ....
"<<endl;
    }
protected:
private:
};
```

```
class Proxy
{
public:
    Proxy(){}
    Proxy(Subject* sub)
    {
        _sub = sub;
    }
    ~Proxy()
    {
        delete _sub;
    }
    void Request()
    {
        cout<<"Proxy request...."<<endl;
        _sub->Request();
    }
protected:
private:
    Subject* _sub;
};
```

Main

```
#include "Proxy.h"
int main(int argc, char* argv[])
{
    Subject* sub = new ConcreteSubject();
    Proxy* p = new Proxy(sub);
    p->Request();
    //printf("Hello World!\n");
    return 0;
}
```

2.7.4、讨论

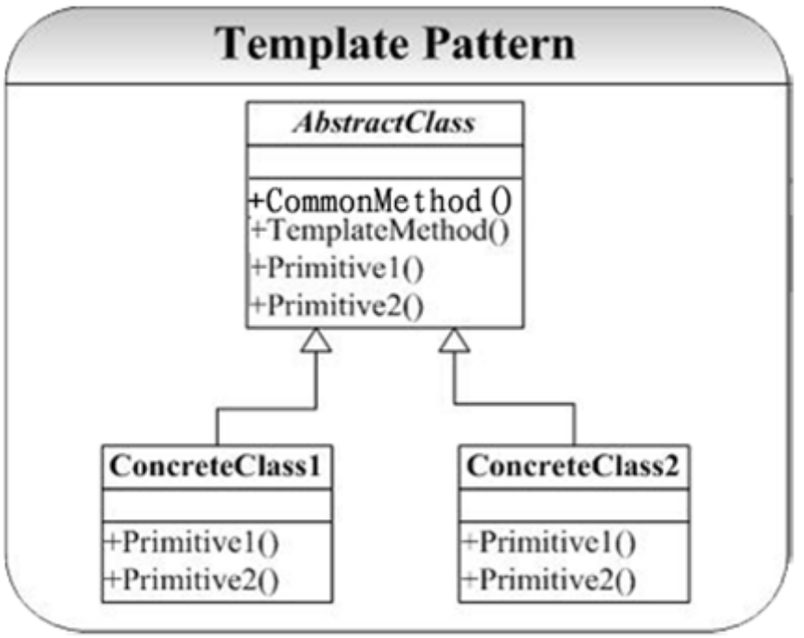
什么时候要使用代理模式呢？在对已有的方法进行使用的时候出现需要对原有方法进行改进或者修改，这时候有两种改进选择：修改原有方法来适应现在的使用方式，或者使用一个“第三者”方法来调用原有的方法并且对方法产生的结果进行一定的控制。第一种方法是明显违背了“对扩展开放、对修改关闭”（开闭原则），而且在原来方法中作修改可能使得原来类的功能变得模糊和多元化（就像现在企业多元化一样），而使用第二种方式可以将功能划分的更加清晰，有助于后面的维护。所以在一定程度上第二种方式是一个比较好的选择！

3.行为模式

3.1 Template 模式

3.1.1、概述

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。1.一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。2.各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。3.控制子类扩展。



3.1.2、类图

3.1.3、模拟实现

Template.h

```
#include <iostream>
```

```
using namespace std;

class AbstractClass
{
public:
    virtual ~AbstractClass(){}
    void CommonMethod()
    {
        cout<<"这是本类族公共方法"<<endl;
    }
    void TemplateMethod()
    {
        this->PrimitiveOperation1();
        this->PrimitiveOperation2();
    }
protected:
    virtual void PrimitiveOperation1() = 0;
    virtual void PrimitiveOperation2() = 0;
    AbstractClass(){}
private:
};

class ConcreteClass1:public AbstractClass
{
public:
    ConcreteClass1(){}
    ~ConcreteClass1(){}
protected:
    void PrimitiveOperation1()
    {
        cout<<"子类一的方法一"<<endl;
    }
    void PrimitiveOperation2()
    {
        cout<<"子类一的方法二"<<endl;
    }
private:
};

class ConcreteClass2:public AbstractClass
{
public:
    ConcreteClass2(){}
    ~ConcreteClass2(){}
```

```
protected:
    void PrimitiveOperation1()
    {
        cout<<"子类二的方法一"<<endl;
    }
    void PrimitiveOperation2()
    {
        cout<<"子类二的方法二"<<endl;
    }
private:
};
```

Main

```
#include "Template.h"


int main(int argc, char* argv[])
{
    //模板模式，本类族公共方法在抽象类
    //分别生成不同子类对象各一个
    AbstractClass* p1 = new ConcreteClass1();
    AbstractClass* p2 = new ConcreteClass2();
    //调用公共方法测试
    cout<<"调用公共方法测试:....."<<endl;
    p1->CommonMethod();p2->CommonMethod();
    cout<<"调用子类方法测试:....."<<endl;
    //p1->PrimitiveOperation1();p2->PrimitiveOperation2();
    p1->TemplateMethod();
    p2->TemplateMethod();
    //printf("Hello World!\n");
    return 0;
}
```

3.1.4、讨论

Template 模式获得一种反向控制结构效果，这也是面向对象系统的分析和设计中一个原则 DIP（依赖倒置：Dependency Inversion Principles）。其含义就是父类调用子类的操作（高层模块调用低层模块的操作），低层模块实现高层模块声明的接口。这样控制权在父类（高层模块），低层模块反而要依赖高层模块。

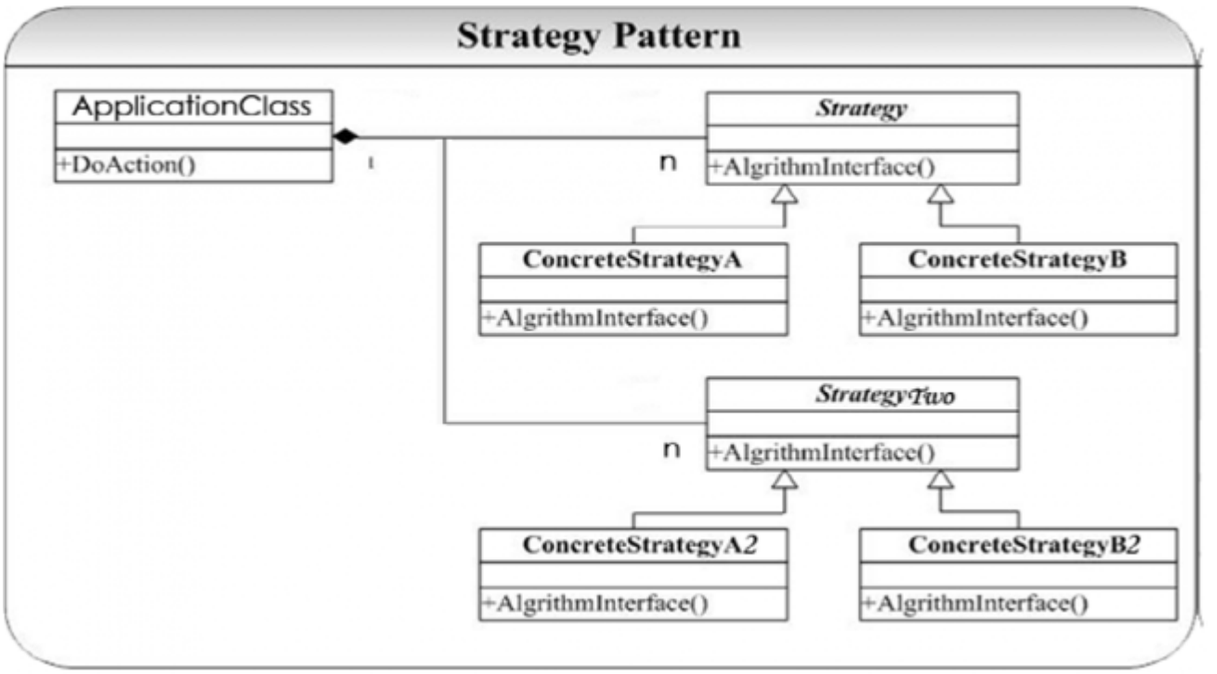
继承的强制性约束关系也让 Template 模式有不足的地方，我们可以看到对于 ConcreteClass 类中的实现的原语方法 Primitive1()，是不能被别的类复用。假设我们要创建一个 AbstractClass 的变体 AnotherAbstractClass，并且两者只是通用算法不一样，其原语操作想复用 AbstractClass 的子类的实现。但是这是不可能实现的，因为 ConcreteClass 继承自 AbstractClass，也就继承了 AbstractClass 的通用算法，AnotherAbstractClass 是复用不了 ConcreteClass 的实现，因为后者不是继承自前者。Template 模式暴露的问题也正是继承所固有的问题，Strategy 模式则通过组合（委托）来达到和 Template 模式类似的效果，其代价就是空间和时间上的代价，关于 Strategy 模式的详细讨论请参考 Strategy 模式解析。

3.2 Strategy 模式

3.2.1、概述

定义一系列的算法,把它们一个个封装起来,并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。1.许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。2.需要使用一个算法的不同变体。3.算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。4.一个类定义了多种行为,并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的 Strategy 类中以代替这些条件语句。

3.2.2、类图



3.2.3、模拟实现

Streategy.h

```
#include <iostream>

using namespace std;

//行为一抽象接口

class BehaviorOne

{

public:

    virtual void BehaviorOneMethod() = 0;

protected:

private:

};

//行为二抽象接口

class BehaviorTwo

{

public:

    virtual void BehaviorTwoMethod() = 0;

protected:

private:

};

//行为一的 A 方法

class BehaviorOneA:public BehaviorOne

{
```

```
public:
    void BehaviorOneMethod()
    {
        cout<<"行为一的 A 方法"<<endl;
    }
protected:
private:
};

//行为一的 B 方法
class BehaviorOneB:public BehaviorOne
{
public:
    void BehaviorOneMethod()
    {
        cout<<"行为一的 B 方法"<<endl;
    }
protected:
private:
};
//行为二的 A 方法
class BehaviorTwoA:public BehaviorTwo
{
public:
    void BehaviorTwoMethod()
    {
        cout<<"行为二的 A 方法"<<endl;
    }
protected:
private:
};

//行为二的 B 方法
class BehaviorTwoB:public BehaviorTwo
{
public:
    void BehaviorTwoMethod()
    {
        cout<<"行为二的 B 方法"<<endl;
    }
protected:
private:
};
----------------------
```

Application.h

```
#include "strategy.h"

/*有一个现实的应用类族，这族类里有一些属性、
行为是共同的。还有一些行为是可变的。
在这里我们假设有两种行为是可变的，将他们抽出
来。
*/

//应用抽象类
class Application
{
public:
    Application(){}
    virtual ~Application(){}
    //这是所有子类都有的功能
    void CommonMethod()
    {
        cout<<"这是我们都有的一样的功能
"<<endl;
    }
    //所有子类都有但不同例如描述信息
    virtual void disply()=0;

    void setBehaviorOne(BehaviorOne*
behaviorOne)
    {
        this->behaviorOne= behaviorOne;
    }

    void setBehaviorTwo(BehaviorTwo*
behaviorTwo)
    {
        this->behaviorTwo= behaviorTwo;
    }

    void perForm()
    {
        behaviorOne->BehaviorOneMethod();
        behaviorTwo->BehaviorTwoMethod();
    }
protected:
    BehaviorOne* behaviorOne;
    BehaviorTwo* behaviorTwo;
};

//插件式定制具体应用类
class ApplicationOne:public Application
```

```
{
public:
    //在构造器里面分别给行为设置默认插件。
    ApplicationOne()
    {
        this->behaviorOne=new
BehaviorOneA();
        this->behaviorTwo=new
BehaviorTwoA();
    }
    //子类特有的功能
    void disply()
    {
        cout<<"你好，我是具体应用类，这是我
的描述信息"<<endl;
    }
};
/*
如果需要生成新的应用类，向上面一样，继承
Application 类，并重写描述方法就可以。
两个参数的默认值也可以在 Application 里面设
置。但缺少灵活性。
如果不这样，每个子类都去写自己的行为（这里是
两个），那么当子类很多而行为用不固定，甚至某种
行为
下属还会不断化时。给修改很维护带来很多的麻烦。
将每种变化都抽出来，在具体应用类里面预留接口。
实行
插件式编程，带来很大的方便
*/
------------------------------
Main
#include "stdafx.h"
#include "Strategy.h"
#include "Application.h"

int main(int argc, char* argv[])
{
    //生成一个应用类对象
    Application *app = new ApplicationOne();
    //测试行为
    app->disply();
    app->CommonMethod();
    app->perForm();
    //设置新行为
    app->setBehaviorOne(new
BehaviorOneB());
    app->setBehaviorTwo(new
BehaviorTwoB());
```

```
app->perForm();                                    return 0;
//printf("Hello World!\n");                        }
```

3.2.4、讨论

可以看到 Strategy 模式和 Template 模式解决了类似的问题，Strategy 模式和 Template 模式实际是实现一个抽象接口的两种方式：继承和组合之间的区别。要实现一个抽象接口，继承是一种方式：我们将抽象接口声明在基类中，将具体的实现放在具体子类中。组合（委托）是另外一种方式：我们将接口的实现放在被组合对象中，将抽象接口放在组合类中。这两种方式各有优缺点，先列出来：

1） 继承优点:易于修改和扩展那些被复用的实现。 缺点:①破坏了封装性，继承中父类的实现细节暴露给子类了；②当父类的实现更改时，其所有子类将不得不随之改变.③从父类继承而来的实现在运行期间不能改变（编译期间就已经确定了）。

2） 组合优点:①封装性好包含对象的内部细节对外是不可见的；②实现和抽象的依赖性很小（组合对象和被组合对象之间的依赖性小）；③可以在运行期间动态定义实现（通过一个指向相同类型的指针，典型的是抽象基类的指针）。 缺点:①系统中对象过多。

从上面对比中我们可以看出，组合相比继承可以取得更好的效果，因此在面向对象的设计中的有一条很重要的原则就是：优先使用（对象）组合，而非（类）继承（Favor Composition Over Inheritance）。实际上，继承是一种强制性很强的方式，因此也使得基类和具体子类之间的耦合性很强。例如在 Template 模式中在 ConcreteClass1 中定义的原语操作别的类是不能够直接复用。而组合（委托）的方式则有很小的耦合性，实现（具体实现）和接口（抽象接口）之间的依赖性很小，例如在本实现中，ConcreteStrategyA 的具体实现操作很容易被别的类复用，例如我们要定义另一个 Context 类 AnotherContext，只要组合一个指向 Strategy 的指针就可以很容易地复用 ConcreteStrategyA 的实现了。另外 Strategy 模式很 State 模式也有相似之处，但是 State 模式注重的对象在不同的状态下不同的操作。两者之间的区别就是 State 模式中具体实现类中有一个指向 Context 的引用，而 Strategy 模式则没有。

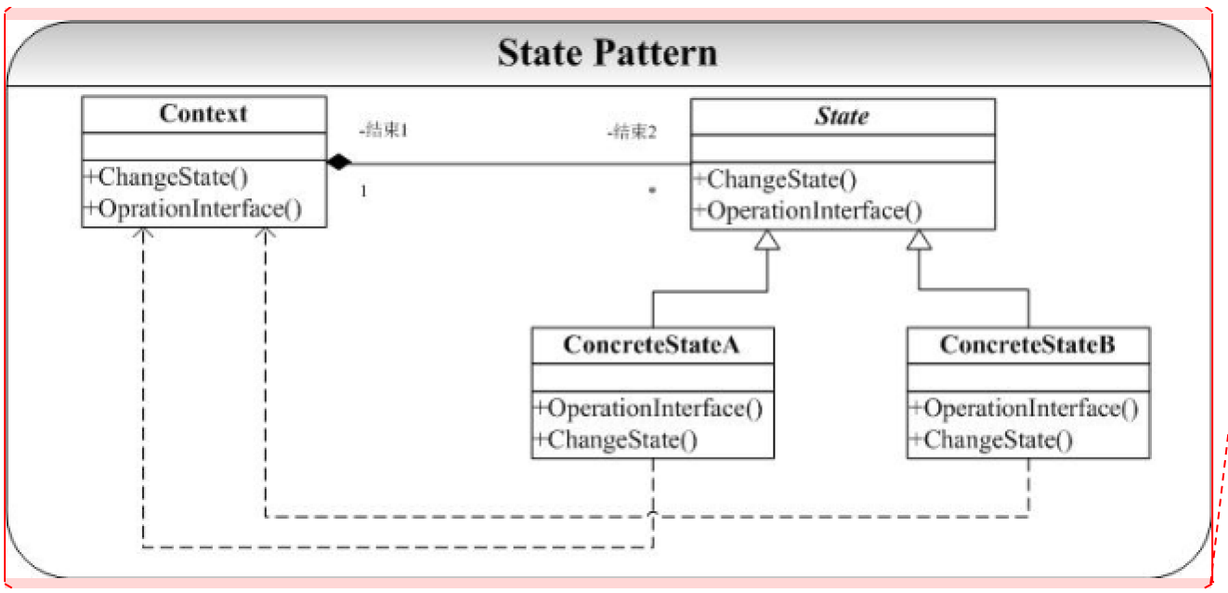
总之，策略模式的目的就是将产品类不变的行为沉淀到基类，变化的行为每个独立出来分别设置和基类平行的抽象类。这样每个实体产品类都基于基类（即成了不变的行为）。在产品基类中，设置行为基类的指针，在实体产品中给指针赋值并调用相关功能。这个每个产品类都是一个**组合。**这样当某个行为发生变化，我们需要修改的只是**一个**行为类，而不是 n 多个 产品类。便于代码的**复用**

3.3、State 模式

3.3.1、概述

定义对象间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。1.一个对象的行为取决于它的状态,并且它必须在运行时刻根据状态改变它的行为。2.一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常,有多个操作包含这一相同的条件结构。State 模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

3.3.2、类图



批注[MU6]：任何一种状态族都要基础 state，这样一个 context 类的不同对象可以管理不同的 state 类族。Context 和 state 类相互注册目的在于解偶合，不会增强依赖，如果将状态的切换代码直接写在 context 里面，那么我们不仅需要为每个 state 类族写一个特定的 context，更重要的是不利于 state 类族的更新。Context 类只有有限的几个动作：调用状态行为不切换，调用状态行为并分别按某种规则切换。行为、切换规则都在 state 类自身。相互没有依赖。要体会这种组合的思想。

3.3.3、模拟实现

代码功能及执行：

state 类族，一个基类对应一种状态

virtual void OperationInterface()方法

实现此状态功能

bool ChangeState(Context* con,State* st)方法

在上下文(con)中装注册新状态(st)

virtual void OperationChangeStateLarge(Context*)方法**********

上线文（context）类同名方法调用此方法

调用上边 OperationInterface 方法实现状态功能

调用上边 ChangeState 注册本状态的下一状态（大）

virtual void OperationChangeStateSmall(Context*)方法**********

上线文（context）类同名方法调用此方法

调用上边 OperationInterface 方法实现状态功能

调用上边 ChangeState 注册本状态的下一状态（小）

上下文 Context 类，在这里注册 状态

Context(State* state);构造方法

用一个状态构造一个新上下文对象

void OprationInterface();

调用状态类同名方法，实现状态功能，但不切换

void OperationChangStateLarge();方法

此方法调用状态类同名方法。

void OperationChangStateSmall();方法

此方法调用状态类同名方法。

bool ChangeState(State* state);

用于设置状态当对象空构造，或需要改变状态时调用

程序执行

1、先生成一个状态类对象并用指针 st 指向

2、用 st 构造一个上下问对象 con（也可空构造，然后用 ChangeState 设置改变对象状态）

3、调用 con 的 OprationInterface 方法，执行状态功能，但不切换

调用 con 的 OperationChangStateLarge 方法，执行状态功能，向大方向切换

调用 con 的 OperationChangStateLarge 方法，执行状态功能，向小方向切换

4、可以应用 ChangeState 方法设置 状态

一点说明

因为 Context 类中 ChangeState 方法为私有方法，所以不能由其他类对象中的 public 方法调用。固借助状态类 protected 方法调用。

State.h

```
class Context; //前置声明

class State
{
public:
    State(){}
    virtual ~State(){}
    virtual void OperationInterface() = 0;
    virtual void OperationChangeStateLarge(Context*) = 0;
    virtual void OperationChangeStateSmall(Context*) = 0;
protected:
    bool ChangeState(Context* con,State* st);
private:
    //bool ChangeState(Context* con,State* st);
};

class ConcreteStateA:public State
{
public:
    ConcreteStateA(){}
    ~ConcreteStateA(){}
    void OperationInterface( );
    void OperationChangeStateLarge(Context*);
```

```
    void
OperationChangeStateSmall(Context*);
protected:
private:
};

class ConcreteStateB:public State
{
public:
    ConcreteStateB(){}
    ~ConcreteStateB(){}
    void OperationInterface();
    void
OperationChangeStateLarge(Context*);
    void
OperationChangeStateSmall(Context*);
protected:
private:
};

class ConcreteStateC:public State
{
public:
    ConcreteStateC(){}
    ~ConcreteStateC(){}
    void OperationInterface();
    void
OperationChangeStateLarge(Context*);
    void
OperationChangeStateSmall(Context*);
protected:
private:
};
----------------------
State.cpp
#include "State.h"

#include "Context.h"
#include <iostream>
using namespace std;

bool  State::ChangeState(Context*  con,State*
st)
{
```

```
    con->ChangeState(st);
    return true;
}

void ConcreteStateA::OperationInterface ()
{
    cout<<"状态 A 的功能接口"<<endl;
}

void
ConcreteStateA::OperationChangeStateLarge(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateB());
    //con->ChangeState(new
ConcreteStateB());
}

void
ConcreteStateA::OperationChangeStateSmall(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateC());
    //con->ChangeState(new
ConcreteStateC());
}

void ConcreteStateB::OperationInterface ()
{
    cout<<"状态 B 的功能接口....."<<endl;
}

void
ConcreteStateB::OperationChangeStateLarge(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateC());
```

```
    //con->ChangeState(new
ConcreteStateC());
}

void
ConcreteStateB::OperationChangeStateSmall(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateA());
    //con->ChangeState(new
ConcreteStateA());
}

void ConcreteStateC::OperationInterface ()
{
    cout<<"状态 C 的功能接口....."<<endl;
}

void
ConcreteStateC::OperationChangeStateLarge(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateA());
    //con->ChangeState(new
ConcreteStateA());
}

void
ConcreteStateC::OperationChangeStateSmall(
Context* con)
{
    OperationInterface();
    this->ChangeState(con,new
ConcreteStateB());
    //con->ChangeState(new
ConcreteStateB());
}
```

Context.h

```
class State; /** * **/
```

```
class Context
{
public:
    Context();
    Context(State* state);
    ~Context();
    //此方法调用状态功能但不切换下一个
    void OprationInterface();
    //此方法调用状态功能切换下一个（大）
    void OperationChangStateLarge();
    //此方法调用状态功能切换下一个（小）
    void OperationChangStateSmall();
protected:
private:
    friend class State; //表明在 State 类中可以访问 Context 类的 private 字段
    bool ChangeState(State* state);//用于设置状态
private:
    State* _state;
};
```

Context.cpp

```
#include "stdafx.h"
#include "Context.h"

#include "State.h"
Context::Context(){}

Context::Context(State* state)
{
    this->_state = state;
}
Context::~Context(){delete _state;}

void Context::OprationInterface()
{
    _state->OperationInterface();
}

bool Context::ChangeState(State* state)
{ ///_state->ChangeState(this,state);
    this->_state = state;
    return true;
```

```
}

void Context::OperationChangStateLarge()
{
    _state->OperationChangeStateLarge(this)
;
}

void Context::OperationChangStateSmall()
{
    _state->OperationChangeStateSmall(this);
}
-------------------------------------
-
Main
```

```
#include "Context.h"
#include "State.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    State* st = new ConcreteStateA();
    Context* con = new Context(st);

    con->OperationChangStateLarge();
    con->OperationChangStateLarge();
    con->OperationChangStateLarge();

    return 0;
}
```

3.3.4、讨论

每个人、事物在不同的状态下会有不同表现（动作），而一个状态又会在不同的表现下转移到下一个不同的状态（State）。最简单的一个生活中的例子就是：地铁入口处，如果你放入正确的地铁票，门就会打开让你通过。在出口处也是验票，如果正确你就可以 ok，否则就不让你通过（如果你动作野蛮，或许会有报警（Alarm），：））。

在应用系统中对输入有不同的响应（状态转移）。通常我们在实现这类系统会使用到很多的 Switch/Case 语句，Case 某种状态，发生什么动作，Case 另外一种状态，则发生另外一种状态。但是这种实现方式至少有以下两个问题：

1）当状态数目不是很多的时候，Switch/Case 可能可以搞定。但是当状态数目很多的时候（实际系统中也正是如此），维护一大组的 Switch/Case 语句将是一件异常困难并且容易出错的事情。

2）状态逻辑和动作实现没有分离。在很多的系统实现中，动作的实现代码直接写在状态的逻辑当中。这带来的后果就是系统的扩展性和维护得不到保证。State 模式就是被用来解决上面列出的两个问题的，在 State 模式中我们将状态逻辑和动作实现进行分离。当一个操作中要维护大量的 case 分支语句，并且这些分支依赖于对象的状态。State 模式将每一个分支都封装到独立的类中。State 模式在实现中，有两个关键点：

1）将 State 声明为 Context 的友元类（friend class），其作用是让 State 模式访问 Context 的 protected 接口 ChangeSate（）。

2）State 及其子类中的操作都将 Context*传入作为参数，其主要目的是 State 类可以通过这个指针调用 Context 中的方法（在本示例代码中没有体现）。这也是 State 模式和 Strategy 模式的最大区别所在。

运行了示例代码后可以获得以下的结果：连续 3 次调用了 Context 的 OprationInterface（）因为每次调用后状态都会改变（A－B－A），因此该动作随着 Context 的状态的转变而获得了不同的结果。

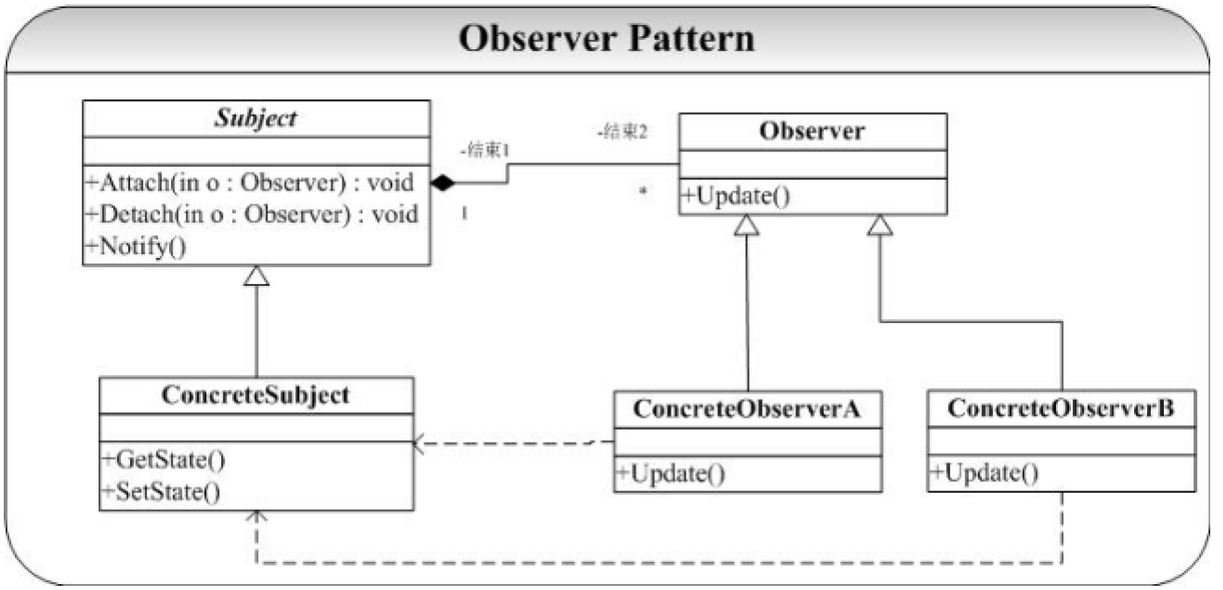
3.4 Observer 模式

3.4.1、概述

定义对象间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。1.当一个抽象模型有两个方面,其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。2.当对一个对象的改变需要同时改变其它对象,而不知道具体有多少对象有待改变。3.当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

最常见的一个例子就是：对同一组数据进行统计分析时候，我们希望能够提供多种形式的表示（例如以表格进行统计显示、柱状图统计显示、百分比统计显示等）。这些表示都依赖于同一组数据，我们当然需要当数据改变的时候，所有的统计的显示都能够同时改变。Observer 模式就是解决了这一个问题。

3.4.2、类图



3.4.3、模拟实现

方法功能说明：

subject 类

virtual ~Subject();

virtual void Attach(Observer* obv);方法

向观察者列表添加观察成员

virtual void Detach(Observer* obv);方法

从观察者列表移除指定的观察者

virtual void Notify();

逐个遍历观察者列表，执行观察成员的 Update 方法

virtual void SetState(const State& st) = 0; ???

设置科目当前状态

virtual State GetState() = 0;

得到科目当前状态

Subject();构造方法

创建具体科目对象时会自动调用此构造方法（父类），创建一个观察者列表

ConcreteSubject();构造方法

创建科目对象，并将状态复原

Observer 类

virtual ~Observer();

virtual void Update(Subject* sub) = 0;方法

获得状态信息，调用反应方法

virtual void PrintInfo() = 0;方法

根据状态信息做出反应

Observer();构造方法

构造一个观察者对象，并置状态变量为 0

ConcreteObserverA(Subject* sub)；构造方法。

根据科目创建一个具体观察对象，调用科目方法将自己在科目对象内注册

~ConcreteObserverA()，析构方法

从科目对象的观察者列表中删除自己（执行科目对象的 Detach 方法）

程序执行
有很多的课题，每种课题都包含一些状态，每个状态的变化都会引起一个或多个反应。而一种反应也可能因多种状态的改变引起。每种反应都要时刻观察课题中状态的变化。每种状态一旦变化都要通知相应的反应。
本例中
每一个科目对象都维护一观察者列表。我们（用户）可以根据需要生成观察者对象，并将观察者对象添加

到科目对象的观察者类表中（本例中由于在观察者类维护一个科目对象的指针，所以可以自动将自己添加如列表）。当科目中状态发生改变时，执行状态对象的通知方法，通知方法会遍历列表，执行列表中每一个观察者对象的相应方法。
以科目为核心，一个观察者对应一个科目为一个对象。这些对象按科目分类，并在科目中注册

Subject.h

```
#include <list>
#include <string>
using namespace std;
typedef string State;
class Observer;

class Subject
{
public:
    virtual ~Subject();
    void Attach(Observer* obv);
    void Detach(Observer* obv);
    void Notify();
    virtual void SetState(const State& st) = 0;
    virtual State GetState() = 0;
protected:
    Subject();
private:
    list<Observer* >* _obvs;
};

class ConcreteSubject:public Subject
{
public:
    ConcreteSubject();
    ~ConcreteSubject();
    State GetState();
    void SetState(const State& st);
protected:
private:
    State _st;
};
```

Subject.cpp

```
#include "Subject.h"
#include "Observer.h"
#include <iostream>
#include <list>
```

```
using namespace std;
typedef string state;

Subject::Subject()
{ //****在模板的使用之前一定要 new，创建
    _obvs = new list<Observer*>;
    cout<<"已经为科目对象创建一个观察者列表,
";
}
Subject::~Subject(){}

void Subject::Attach(Observer* obv)
{
    _obvs->push_front(obv);
}

void Subject::Detach(Observer* obv)
{
    if (obv != NULL) _obvs->remove(obv);
}

void Subject::Notify()
{
    list<Observer*>::iterator it;
    cout<<(_obvs->empty());
    it = _obvs->begin();
    for (;it != _obvs->end();it++)
    {//关于模板和 iterator 的用法
        (*it)->Update(this);
    }
}

ConcreteSubject::ConcreteSubject()
{
    cout<<"已经生成一个科目对象."<<endl;;
    _st = '\0';
}

ConcreteSubject::~ConcreteSubject(){ }
State ConcreteSubject::GetState(){ return _st;}
void  ConcreteSubject::SetState(const  State&
st){ _st = st;}
-------------------------------------
-------
```

Observer.h

```
#include "Subject.h"
#include <string>
using namespace std;
typedef string State;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* sub) = 0;
    virtual void PrintInfo() = 0;
protected:
    Observer();
    State _st;
private:
};
class ConcreteObserverA:public Observer
{
public:
    virtual Subject* GetSubject();
    ConcreteObserverA(Subject* sub);
    virtual ~ConcreteObserverA();
    //传入 Subject 作为参数，这样可以让一个 View
属于多个的 Subject。
    void Update(Subject* sub);
    void PrintInfo();
protected:
private:
    Subject* _sub;
};
class ConcreteObserverB:public Observer
{
public:
    virtual Subject* GetSubject();
    ConcreteObserverB(Subject* sub);
    virtual ~ConcreteObserverB();
    //传入 Subject 作为参数，这样可以让一个
View 属于多个的 Subject。
    void Update(Subject* sub);
    void PrintInfo();
protected:
private:
    Subject* _sub;
};
```

```
------------------------------
Observer.cpp
#include "Observer.h"

#include "Subject.h"
#include <iostream>
#include <string>
using namespace std;
Observer::Observer(){ _st = '\0';}

Observer::~Observer(){}

ConcreteObserverA::ConcreteObserverA(Subj
ect* sub)
{
    _sub = sub;
    _sub->Attach(this);
    cout<<"观察者 A 以启动，已经在科目对象中
注册"<<endl;
}

ConcreteObserverA::~ConcreteObserverA()
{
    _sub->Detach(this);
    if (_sub != 0){delete _sub;}
}

Subject*    ConcreteObserverA::GetSubject(){
return _sub; }

void ConcreteObserverA::PrintInfo()
{
    cout<<"观察者 A 观察到科目状态变为"
<<_sub->GetState()<<"已经做出相应的反应
"<<endl;
}

void    ConcreteObserverA::Update(Subject*
sub)
{
    _st = sub->GetState();
    PrintInfo();
}
```

```
ConcreteObserverB::ConcreteObserverB(Subje
ct* sub)
{
    _sub = sub;
    _sub->Attach(this);
    cout<<"观察者 B 以启动，已经在科目对象中
注册"<<endl;
}

ConcreteObserverB::~ConcreteObserverB()
{
    _sub->Detach(this);
    if (_sub != 0){ delete _sub;}
}

Subject*
ConcreteObserverB::GetSubject(){return
_sub;}

void ConcreteObserverB::PrintInfo()
{
    cout<<"观察者 B 观察到科目状态变为"
<<_sub->GetState()<<"已经做出相应的反应
"<<endl;
}

void ConcreteObserverB::Update(Subject* sub)
{
    _st = sub->GetState();
    PrintInfo();
}
```

Main

```
#include "Subject.h"
#include "Observer.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    ConcreteSubject*    sub    =    new
ConcreteSubject();
        Observer*    o1    =    new
ConcreteObserverA(sub);
```

```
    Observer*        o2        =        new        sub->Notify();
ConcreteObserverB(sub);                              cout<<endl<<endl;
    cout<<endl<<endl;
                                                     sub->Detach(o1);/*从科目对象的观察者列
    sub->SetState("old");                        表中删除一个观察者 o1->~Observer();//观察者
    sub->Notify();                               对象将自己从科目列表中删除*/
    cout<<endl<<endl;                                sub->Notify();
                                                     //printf("Hello World!\n");
    sub->SetState("new");   //也  可  以  由        return 0;
Observer 调用                                    }
```

3.4.4、讨论

在 Observer 模式的实现中，Subject 维护一个 list 作为存储其所有观察者的容器。每当调用 Notify 操作就遍历 list 中的 Observer 对象，并广播通知改变状态（调用 Observer 的 Update 操作）。目标的状态 state 可以由 Subject 自己改变(示例)，也可以由 Observer 的某个操作引起 state 的改变(可调用 Subject 的 SetState 操作)。Notify 操作可以由 Subject 目标主动广播（示例），也可以由 Observer 观察者来调用（因为 Observer 维护一个指向 Subject 的指针）。

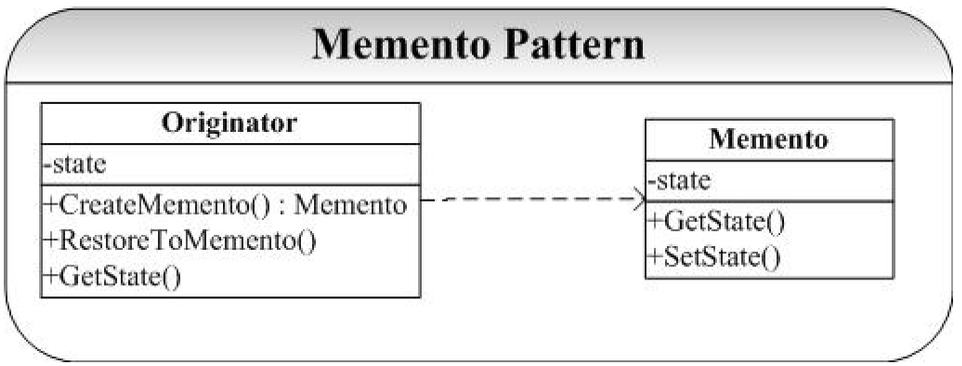
运行示例程序，可以看到当 Subject 处于状态“old”时候，依赖于它的两个观察者都显示“old”，当目标状态改变为“new”的时候，依赖于它的两个观察者也都改变为“new” Observer 模式也称为发布一订阅（publish-subscribe），目标就是通知的发布者，观察者则是通知的订阅者（接受通知）。

3.5 Memento 模式

3.5.1、概述

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。这样以后就可将该对象恢复到原先保存的状态。 适用性 1.必须保存一个对象在某一个时刻的(部分)状态,这样以后需要时它才能恢复到先前的状态。2.如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

3.5.2、类图



3.5.3、模拟实现

方法功能说明：

Originator 类

Originator();构造方法

构造操作对象，并设置需要保存的状态初值为空，保存器指针清零

Originator(const State& sdt);构造方法

构造操作对象，并设置需要保存的状态初值为指定值，保存器指针清零

Memento* CreateMemento();方法

生成一个保存器，将当前状态值置入。返回保存期

void SetMemento(Memento* men);方法

空方法

void RestoreToMemento(Memento* mt);方法

设置当前状态为保存器中读取值

State GetState();方法

得到当前状态值

void SetState(const State& sdt);方法

设置当前状态值

void PrintState();方法

打印当前状态值//检验保存效果

protected:

private:

State _sdt;

Memento* _mt;

};

Memento 类

private:friend class Originator;

//这是最关键的地方，将 Originator 为 friend 类，可以访问内部信息，但是其他类不能访问，便于保密

typedef string State;

可以将 state 设置为需要的类别

private:Memento();构造方法

空构造方法

private:Memento(const State& sdt);构造方法

构造保存器对象，报设置值

private:void SetState(const State& sdt);

设置状态值

private:State GetState();

得到状态值

};

Memento.h

#include <string>

using namespace std;

class Memento;

class Originator

{

public:

typedef string State;

Originator();

Originator(const State& sdt);

~Originator();

Memento* CreateMemento();

void SetMemento(Memento* men);

void RestoreToMemento(Memento* mt);

State GetState();

void SetState(const State& sdt);

void PrintState();

```
protected:
private:
    State _sdt;
    Memento* _mt;
};

class Memento
{
public:
protected:
private:
    //这是最关键的地方，将 Originator 为 friend 类，可以访问内部信息，但是其他类不能访问
    friend class Originator;
    typedef string State;
    Memento();
    Memento(const State& sdt);
    ~Memento();
    void SetState(const State& sdt);
    State GetState();
private:
    State _sdt;
};
```

Memento.cpp

```
#include "Memento.h"
#include <iostream>
using namespace std;

typedef string State;
Originator::Originator()
{
    _sdt = "";_mt = 0;
}

Originator::Originator(const State& sdt)
{
    _sdt = sdt;_mt = 0;
}

Originator::~Originator() {}

Memento* Originator::CreateMemento()
{
```

```
    cout<<"保存器创造成功，并将"<< _
sdt<<"值保存"<<endl;
    return new Memento(_sdt);
}
State Originator::GetState() { return _sdt; }

void Originator::SetState(const State& sdt)
{
    cout<<"状态值以由"<< _sdt<<"设置为
"<<sdt<<endl;
    _sdt = sdt;
}

void Originator::PrintState()
{   cout<<"当 前 状 态 值 为 ： "
<<this->_sdt<<endl;}

void Originator::SetMemento(Memento* men)
{}

void
Originator::RestoreToMemento(Memento* mt)
{
    this->_sdt = mt->GetState();
    cout<<"状态值恢复为"<< _sdt<<endl;
}

Memento::Memento(){ }
Memento::Memento(const State& sdt){ _sdt =
sdt; }
State Memento::GetState() { return _sdt; }
void  Memento::SetState(const  State&  sdt){
_sdt = sdt; }
----------------------
Main
#include "stdafx.h"
#include "Memento.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    Originator* o = new Originator();
```

```
    o->SetState("old"); //备忘前状态
    o->PrintState();//显示当前值
    Memento* m1 = o->CreateMemento();
//将当前状态备忘
    cout<<endl;


    o->SetState("newA"); //修改状态
    o->PrintState();
    cout<<endl;


    o->SetState("newB"); //修改状态
    o->PrintState();
    cout<<endl;


    o->SetState("newC"); //修改状态
    o->PrintState();
     Memento* m2 = o->CreateMemento();
//将当前状态备忘
    cout<<endl;


    o->RestoreToMemento(m1); //恢复修改
前状态
    o->PrintState();
    cout<<endl;


    o->RestoreToMemento(m2); //恢复修改
前状态
    o->PrintState();
//  printf("Hello World!\n");
    return 0;
}
```

3.5.4、讨论

没有人想犯错误，但是没有人能够不犯错误。犯了错误一般只能改过，却很难改正（恢复）。世界上没有后悔药，但是我们在进行软件系统的设计时候是要给用户后悔的权利（实际上可能也是用户要求的权利:))，我们对一些关键性的操作肯定需要提供诸如撤销（Undo）

Memento 模式的关键就是要在不破坏封装行的前提下，捕获并保存一个类的内部状态，这样就可以利用该保存的状态实施恢复操作。

3.6 Mediator 模式

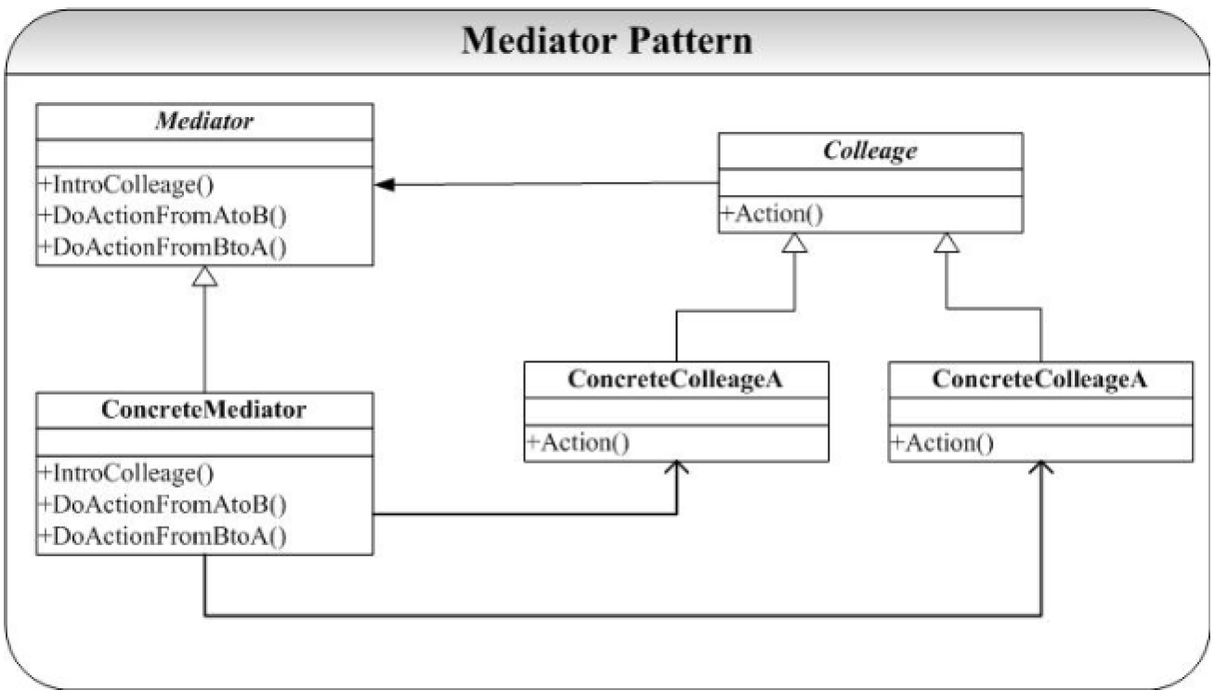
3.6.1、概述

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。 适用性 1.一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。2.一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。3.想定制一个分布在多个类中的行为，而又不想生成太多的子类。

在面向对象系统的设计和开发过程中，对象之间的交互和通信是最为常见的情况，因为对象间的交互本身就是一种通信。在系统比较小的时候，可能对象间的通信不是很多、对象也比较少，我们可以直接硬编码到各个对象的方法中。但是当系统规模变大，对象的量变引起系统复杂度的急剧增加，对象间的通信也变得越来越复杂，这时候我们就要提供一个专门处理对象间交互和通信的类，这个中介者就是 Mediator 模式。Mediator 模式提供将对象间的交互和通讯封装在一个类中，各个对象间的通信不必显势去声明和引用，大大降低了系统的复杂性能（了解一个对象总比深入熟悉 n 个对象要好）。另外 Mediator 模式还带来了系统对象间的松耦合，这些将在讨论中详细给出。

3.6.2、类图



3.6.3、模拟实现

方法功能说明：

Mediator 类

ConcreteMediator(Colleage* clgA,Colleage* clgB);构造方法

构造对象，并设置两个关联参数 clgA clgB

void SetConcreteColleageA(Colleage* clgA);方法

设置 clgA 为给定对象

void SetConcreteColleageB(Colleage* clgB);方法

设置 clgB 为给定对象

Colleage* GetConcreteColleageA();方法

返回 clgA

Colleage* GetConcreteColleageB();方法

返回 clgB

void IntroColleage(Colleage* clgA,Colleage* clgB);

重新设置 两个关联参数 clgA clgB

void DoActionFromAtoB();方法

得到 A 对象状态值并设置给 B 对象

void DoActionFromBtoA();

得到 A 对象状态值并设置给 B 对象

Colleage 类

Colleage();空构造方法

Colleage(Mediator* mdt);构造方法

构造对象并设置值

ConcreteColleageA(Mediator* mdt);构造方法

构造对象并设置值

void Aciton();方法

中介对象的 do 方法

打印信息

void SetState(const string& sdt);方法

设置状态值

string GetState();方法

返回状态值

protected:

private:

string _sdt;

};

程序执行

1、两种开始方法，总之需要通信的同事都要在中介中注册，中介要在每个同事中注册。

开始一：生成同事一，生成同事二，用同事 1、2 生成中介对象，将中介对象在同事中注册。

开始二：生成中介对象，用中介对象生成同事一、二，同事一、二在中介中注册。

2、当某个同事状态信息发生改变时，可以执行自己的 action 方法。action 调用中介方法（因为同事 action 方法需要调用 中介方法，所以中介

需要在 同事中注册）

3、中介中为每个通信的同事写有专门的通信方法。

当调用此方法时，按照前期设定，读取发送的信息，

并为需要的同事设置信息（中介需要读

取、设置同时信息，所以每个同事都要在中介中注册）

4、完成一个信息传递

代码：

Mediator.h

```
class Colleage;

class Mediator
{
public:
    virtual ~Mediator(){}
    virtual void DoActionFromAtoB() = 0;
    virtual void DoActionFromBtoA() = 0;
protected:
    Mediator(){}
private:
};

class ConcreteMediator:public Mediator
{
public:
    ConcreteMediator(){}
    ConcreteMediator(Colleage*
clgA,Colleage* clgB);
    ~ConcreteMediator(){}
    void SetConcreteColleageA(Colleage*
clgA);
    void SetConcreteColleageB(Colleage*
clgB);
    Colleage* GetConcreteColleageA();
    Colleage* GetConcreteColleageB();
    void IntroColleage(Colleage*
clgA,Colleage* clgB);
    void DoActionFromAtoB();
    void DoActionFromBtoA();
protected:
private:
    Colleage* _clgA;
```

```
    Colleage* _clgB;
};
-----------------------------
Mediator.cpp
#include "Mediator.h"
#include "Colleage.h"
ConcreteMediator::ConcreteMediator(Colleag
e* clgA,Colleage* clgB)
{
    this->_clgA = clgA;
    this->_clgB = clgB;
}

void ConcreteMediator::DoActionFromAtoB()
{
    _clgB->SetState(_clgA->GetState());
}
void
ConcreteMediator::SetConcreteColleageA(Coll
eage* clgA)
{
    this->_clgA = clgA;
}
void
ConcreteMediator::SetConcreteColleageB(Coll
eage* clgB)
{
    this->_clgB = clgB;
}
Colleage*
ConcreteMediator::GetConcreteColleageA()
{
    return _clgA;
}
Colleage*
ConcreteMediator::GetConcreteColleageB()
{
    return _clgB;
}
void
ConcreteMediator::IntroColleage(Colleage*
clgA,Colleage* clgB)
{
    this->_clgA = clgA;
```

```
    this->_clgB = clgB;
}
void ConcreteMediator::DoActionFromBtoA()
{
    _clgA->SetState(_clgB->GetState());
}
```

Colleage.h

```
#include <string>
using namespace std;

class Mediator;

class Colleage
{
public:
    virtual ~Colleage();
    virtual void Aciton() = 0;
    virtual void SetState(const string& sdt) =
0;
    virtual string GetState() = 0;
protected:
    Colleage();
    Colleage(Mediator* mdt);
    Mediator* _mdt;
private:
};

class ConcreteColleageA:public Colleage
{
public:
    ConcreteColleageA();
    ConcreteColleageA(Mediator* mdt);
    ~ConcreteColleageA();
    void Aciton();
    void SetState(const string& sdt);
    string GetState();
protected:
private:
    string _sdt;
};

class ConcreteColleageB:public Colleage
{
```

```
public:
    ConcreteColleageB();
    ConcreteColleageB(Mediator* mdt);
    ~ConcreteColleageB();
    void Aciton();
    void SetState(const string& sdt);
    string GetState();
protected:
private:
    string _sdt;
};
-----------------------------------
Colleage.cpp
#include "Colleage.h"
#include "Mediator.h"
#include <iostream>
using namespace std;

Colleage::Colleage()
{ //_sdt = " ";
}
Colleage::Colleage(Mediator* mdt)
{
    this->_mdt = mdt;//_sdt = " ";
}
Colleage::~Colleage(){}

ConcreteColleageA::ConcreteColleageA(){}
ConcreteColleageA::~ConcreteColleageA() {}
ConcreteColleageA::ConcreteColleageA(Medi
ator* mdt):Colleage(mdt){}
string    ConcreteColleageA::GetState(){return
_sdt;}
void         ConcreteColleageA::SetState(const
string& sdt){ _sdt = sdt; }
void ConcreteColleageA::Aciton()
{
    _mdt->DoActionFromAtoB();
    cout<<"同 事 一 当 前 值 为 "<<"
"<<this->GetState()<<"已经通过中介方法将信
息传递给目标对象"<<endl;
}

ConcreteColleageB::ConcreteColleageB() {}
```

```
ConcreteColleageB::~ConcreteColleageB() {}
ConcreteColleageB::ConcreteColleageB(Media
tor* mdt):Colleage(mdt){}

void ConcreteColleageB::Aciton()
{
    _mdt->DoActionFromBtoA();
    cout<<"同 事 二 当 前 值 为 "<<"
"<<this->GetState()<<"已经通过中介方法将信
息传递给目标对象"<<endl;
}

string ConcreteColleageB::GetState(){ return
_sdt; }
void ConcreteColleageB::SetState(const
string& sdt){ _sdt = sdt;}
--------------------------------
Main
#include "Mediator.h"
#include "Colleage.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    ConcreteMediator* m = new
ConcreteMediator();//中介对象
    ConcreteColleageA* c1 = new
ConcreteColleageA(m);//生成同事一
    ConcreteColleageB* c2 = new
ConcreteColleageB(m);//生成同事二
    m->IntroColleage(c1,c2);//需要通信的同
事设置入中介，我想如果很多可以用列表吧

    c1->SetState("111111111111"); //同事一
设置信息
        //cout<<"同 事 一 当 前 值 ： "
<<c1->GetState()<<endl;//读出同事一的信息
    c1->Aciton();/*同事一的活动,在中介中为每
个同事都写一个专门的活动，
                当然如果某同事活动种类
很多，可以为他写很多个，供其开关调用*/
    cout<<"同 事 二 当 前 值 :
"<<c2->GetState()<<endl;/*活动一已经通过中
介将信息传递给同事二,读出同事二的值*/
```

```
                                                    c2->Aciton();
    c2->SetState("22222222");/*/同事二收到
信息，做出反应设置自己的信息*/                          //printf("Hello World!\n");*/
    //cout<<"同 事 二 当 前 值 ： "                    return 0;
<<c2->GetState();//读出同事二的信息                  }
```

3.6.4、讨论

Mediator 模式是一种很有用并且很常用的模式，它通过将对象间的通信封装到一个类中，将多对多的通信转化为一对多的通信，降低了系统的复杂性。Mediator 还获得系统解耦的特性，通过 Mediator，各个 Colleague 就不必维护各自通信的对象和通信协议，降低了系统的耦合性，Mediator 和各个 Colleague 就可以相互独立地修改了。Mediator 模式还有一个很显著额特点就是将控制集中，集中的优点就是便于管理，也正式符合了 OO 设计中的每个类的职责要单一和集中的原则。

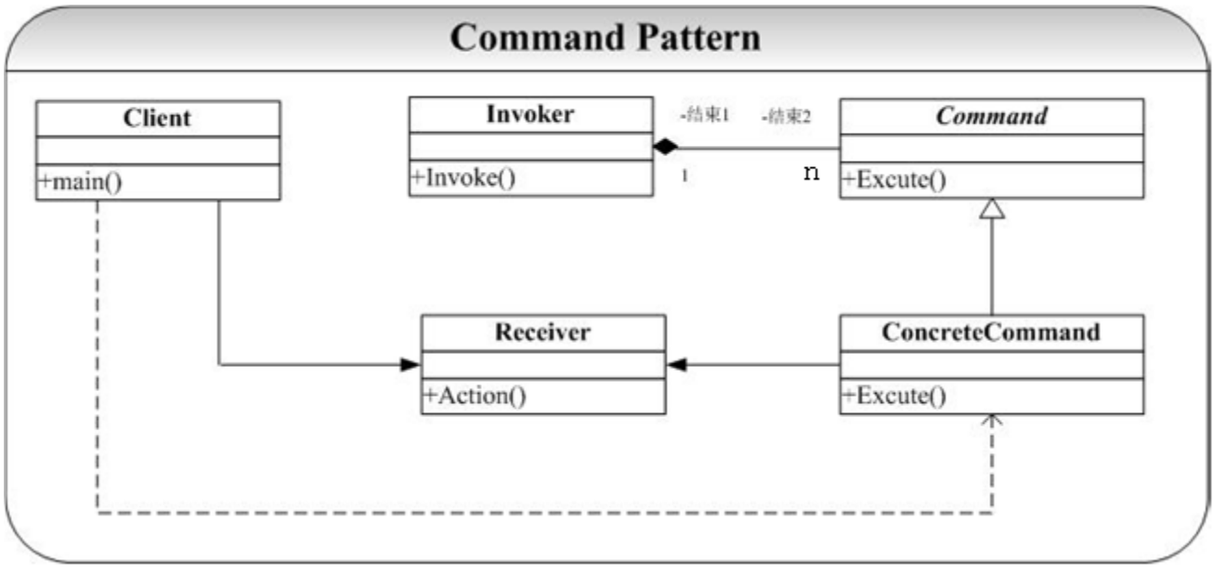
对象不直接交流而是通过中介，用一个中介者对象来封装一系列的对象交互。中介者使各对象不需要显式的相互引用，从而使其耦合松散，而且可以独立的改变他们之间的交互中介者模式很容易在系统中应用，也很容易在系统中误用。Mediator 的出现减少了各个 Colleague 的耦合，使得可以独立地改变和复用各个 Colleague 类和 Mediator；由于把对象如何协作进行了抽象，将中介作为一个独立的概念并将其封装在一个对象中，这样关注的对象就从对象各自本身的行为转移到它们之间的交互上来，也就是站在一个更宏观的角度去看待系统。由于 ConcreteMediator 控制了集中化，于是就把交互复杂性变为了中介者的复杂性，这就使得中介者会变得比任何一个 ConcreteColleague 都复杂。

3.7 Command 模式

3.7.1、概述

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。适用性 1.抽象出待执行的动作以参数化某对象。2.在不同的时刻指定、排列和执行请求。3.支持取消操作。4.支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。5.用构建在原语操作上的高层操作构造一个系统。

3.7.2、类图



3.7.3、模拟实现

方法功能说明：

Reciever 类

void Action();方法

处理对象的处理活动

Command 类

ConcreteCommand(Reciever* rev);构造方法

生成命令类对象，设置内含处置对象

void Excute();方法

调用处理类对象的处理活动

Invoker 类

Invoker(Command* cmd);构造方法

生成激活对象，设置内含命令对象

void Invoke();方法

调用命令的 excute 方法

程序执行

一个请求（invoker)可能激活（包含）多个命令，每个命令都对应一个行为（处理）。我们将需要的每个处理对象都包含在命令对象中，在将这些命令对象在 invoker 对象中进行注册。这样一个激活操作可以调用所有需要的命令，产生需要的结果。

Reciever.h

```
#include <iostream>
class Reciever
{
public:
    Reciever(){}
    ~Reciever(){}
    void Action()
```

Command.h

```
#include "Reciever.h"
#include <iostream>

class Reciever;

class Command
{
public:
    virtual ~Command(){}
    virtual void Excute() = 0;
protected: Command(){}
private:
```

```
    {
        std::cout<<"处 理 对 象 的 处 理 活 动
"<<std::endl;
    }
protected:
private:
};
```

```
};

class ConcreteCommand:public Command
{
public:
    ConcreteCommand(Reciever* rev)
    { this->_rev = rev;}
    ~ConcreteCommand()
    { delete this->_rev; }
    void Excute()
    {
        std::cout<<"命 令 已 经 被 唤 醒 ...
"<<std::endl;
```

```
        _rev->Action();
    }
protected:
private:
    Reciever* _rev;
};
-------------------------------------
-
Invoker.h
#include "Command.h"
#include <iostream>
```

```
class Command;
class Invoker
{
public:
    Invoker(Command* cmd){ _cmd = cmd; }
    ~Invoker(){ delete _cmd; }
    void Invoke(){ _cmd->Excute(); }
protected:
private:
    Command* _cmd;
};
```

3.7.4、讨论

Command 模式在实现的实现和思想都很简单，其关键就是将一个请求封装到一个类中（Command），再提供处理对象（Receiver），最后 Command 命令由 Invoker 激活。Command 模式在有些设计中起到了很好的效果。主要体现在：1） Command 模式将调用操作的对象和知道如何实现该操作的对象解耦。在上面 Command 的结构图中，Invoker 对象根本就不知道具体的是那个对象在处理 Excute 操作（当然要知道是 Command 类别的对象，也仅此而已）。2） 在 Command 要增加新的处理操作对象很容易，我们可以通过创建新的继承自 Command 的子类来实现这一点。3） Command 模式可以和 Memento 模式结合起来，支持取消的操作。

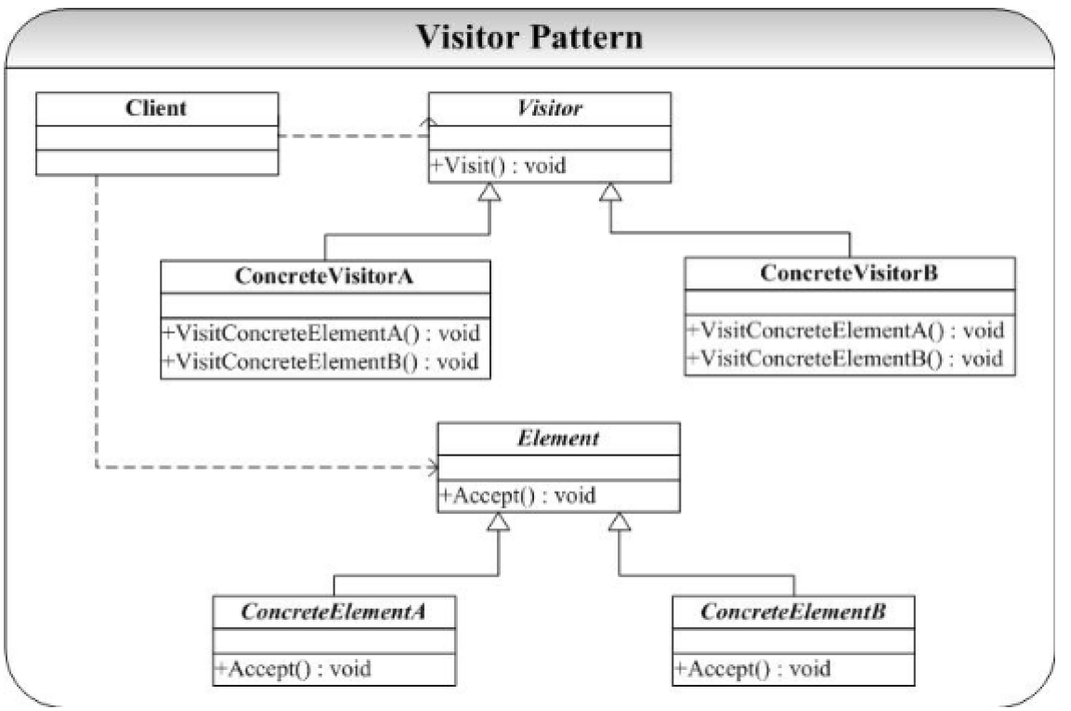
3.8 Visitor 模式

3.8.1、概述

访问者模式：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提

下定义作用于这些元素的新操作。 适用性：1.一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。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，而你想避免让这些操作“污染”这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用 Visitor 模式让每个应用仅包含需要用到的操作。3.定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。 如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

在面向对象系统的开发和设计过程，经常会遇到一种情况就是需求变更（Requirement Changing），经常我们做好的一个设计、实现了一个系统原型，咱们的客户又会有了新的需求。我们又因此不得不去修改已有的设计，最常见就是解决方案就是给已经设计、实现好的类添加新的方法去实现客户新的需求，这样就陷入了设计变更的梦魇：不停地打补丁，其带来的后果就是设计根本就不可能封闭、编译永远都是整个系统代码 。



Visitor 模式则提供了一种解决方案：将更新(变更)封装到一个类中（访问操作），并由待更改类提供一个接收接口，则可达到效果。

3.8.2 、类图

3.8.3、模拟实现

Visitor 类（访问实现即行为，在这里每个元素都有配套的行为）

virtual void VisitConcreteElementA(Element* elm)方法

某元素的 A 行为

virtual void VisitConcreteElementB(Element* elm);方法

某元素的 B 行为

Element 类（具体元素）

void Accept(Visitor* vis);方法

vis->调用行为方法

白话文理解：
表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。也就是说 在具体的元素类（Element）中和行为想关的只有一个 accept 方法，而在每个具体行为类（visitor）中都为每个元素专门单列方法。当某个元素需要某个行为时，我们要做的只是：1、元素* ement = 新具体元素（）； 2、行为* visitor= 新具体行为（）；

3、通过元素类的 accept 方法（emement->Accept(visitor);）即可调用某具体行为类中专为为此元素写的方法（visitor->某元素方法）。如果需要增加新的行为，只需继承行为抽象类，并在其中为每个元素写上方法。然后在上面 2 中形成此类对象即可，和元素无关。但元素更新很不方便。

Visitor.h

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

class ConcreteElementA;
class ConcreteElementB;
class Element;

class Visitor
{
public:
    virtual ~Visitor(){}
    virtual void VisitConcreteElementA(Element* elm) = 0;
    virtual void VisitConcreteElementB(Element* elm) = 0;
protected:
    Visitor(){}
private:
};

class ConcreteVisitorA:public Visitor
{
public:
    ConcreteVisitorA()
    {
        cout<<"你好，这里是 A 行为"<<endl;
    }
    virtual ~ConcreteVisitorA(){}
    virtual void VisitConcreteElementA(Element* elm);
    virtual void
```

```
VisitConcreteElementB(Element* elm);
protected:
private:
};

class ConcreteVisitorB:public Visitor
{
public:
    ConcreteVisitorB(){cout<<"你好，这里是B行为"<<endl;}
    virtual ~ConcreteVisitorB(){}
    virtual void VisitConcreteElementA(Element* elm);
    virtual void VisitConcreteElementB(Element* elm);
protected:
private:
};
```

Visitor.cpp

```
#include "Visitor.h"
#include "Element.h"
#include <iostream>
using namespace std;

void ConcreteVisitorA::VisitConcreteElementA(Element* elm)
{
    cout<<"A行为中专为A元素写的方法"<<endl;
}

void ConcreteVisitorA::VisitConcreteElementB(Element* elm)
{
    cout<<"A行为中专为B元素写的方法"<<endl;
}

void ConcreteVisitorB::VisitConcreteElementA(Ele
```

```
ment* elm)
{
    cout<<"B 行 为 中 专 为 A 元 素 写 的 方 法 "
<<endl;
}
void
ConcreteVisitorB::VisitConcreteElementB(Ele
ment* elm)
{
    cout<<"B 行 为 中 专 为 B 元 素 写 的 方 法 "
<<endl;
}
-------------------------------------
-----------
Element.h
#include <iostream>
using namespace std;


class Visitor;

class Element
{
public:
    virtual ~Element(){}
    virtual void Accept(Visitor* vis) = 0;
protected:
    Element(){}
private:
};


class ConcreteElementA:public Element
{
public:
    ConcreteElementA(){cout<<"你好，这里是
元素 A"<<endl;}
    ~ConcreteElementA(){}
    void Accept(Visitor* vis);
protected:
private:
};

class ConcreteElementB:public Element
{
```

```
public:
    ConcreteElementB(){cout<<"你好，这里是元素 B"<<endl;}
    ~ConcreteElementB(){}
    void Accept(Visitor* vis);
protected:
private:
};
```

Element.cpp

```
#include "Element.h"
#include "Visitor.h"


void ConcreteElementA::Accept(Visitor* vis)
{
    cout<<"即将调用元素 A 的方法，";
    vis->VisitConcreteElementA(this);
}
void ConcreteElementB::Accept(Visitor* vis)
{
    cout<<"即将调用元素 B 的方法，";
    vis->VisitConcreteElementB(this);
}
```

Main

```
#include "Element.h"
#include "Visitor.h"


int main(int argc, char* argv[])
{
    Visitor* visA = new ConcreteVisitorA();//生成一个行为对象
    Visitor* visB = new ConcreteVisitorB();//生成一个行为对象
    Element* elmA = new ConcreteElementA();//生成一个元素
    Element* elmB = new ConcreteElementB();//生成一个元素
    cout<<endl;
    elmA->Accept(visA);//元素调用方法
    elmA->Accept(visB);//元素调用方法
    elmB->Accept(visA);//元素调用方法
    elmB->Accept(visB);//元素调用方法
```

return 0; }

3.8.4、讨论

Visitor 模式的实现过程中有以下的地方要注意：

1）Visitor 类中的 Visit（）操作的实现。这里我们可以向 Element 类仅仅提供一个接口 Visit（），而在 Accept（）实现中具体调用哪一个 Visit（）操作则通过函数重载（overload）的方式实现：我们提供 Visit（）的两个重载版本 a）Visit（ConcreteElementA* elmA），b）Visit（ConcreteElementB* elmB）。Visitor 模式可以使得 Element 在不修改自己的同时增加新的操作，但是这也带来了至少以下的两个显著问题：

1） 破坏了封装性。Visitor 模式要求 Visitor 可以从外部修改 Element 对象的状态，这一般通过两个方式来实现：a）Element 提供足够的 public 接口，使得 Visitor 可以通过调用这些接口达到修改 Element 状态的目的；b）Element 暴露更多的细节给 Visitor，或者让 Element 提供 public 的实现给 Visitor（当然也给了系统中其他的对象），或者将 Visitor 声明为 Element 的 friend 类，仅将细节暴露给 Visitor。但是无论那种情况，特别是后者都将是破坏了封装性原则

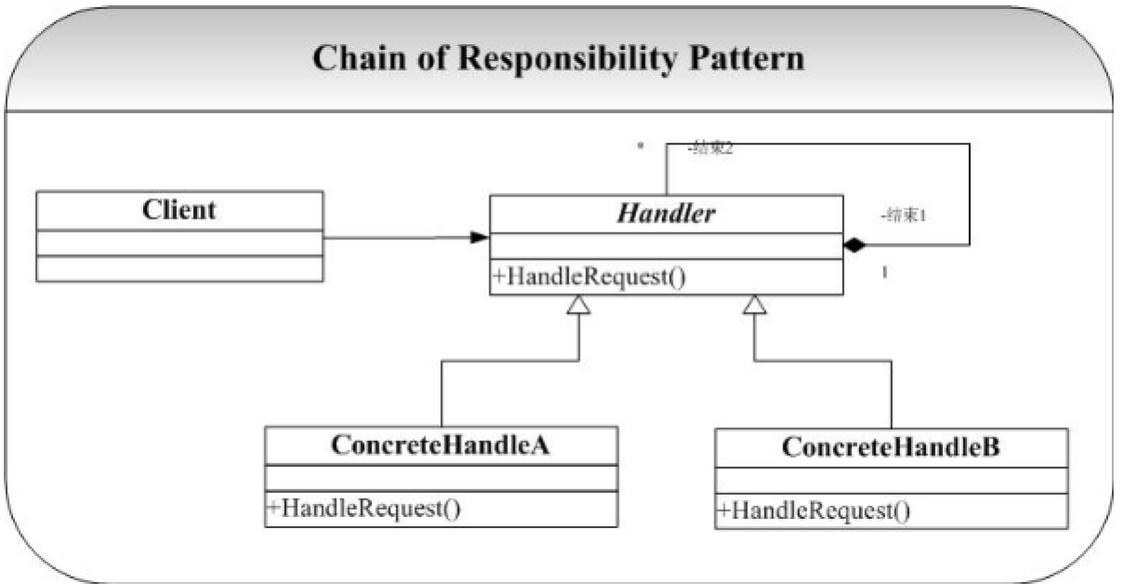
2） ConcreteElement 的扩展很困难：每增加一个 Element 的子类，就要修改 Visitor 的接口，使得可以提供给这个新增加的子类的访问机制。从上面我们可以看到，或者增加一个用于处理新增类的 Visit（）接口，或者重载一个处理新增类的 Visit（）操作，或者要修改 RTTI 方式实现的 Visit（）实现。无论那种方式都给扩展新的 Element 子类带来了困难。

3.9 Chain of Responsibility 模式

3.9.1、概述

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，并沿着这条链传递该请求，直到有一个对象处理它为止。这一模式的想法是，给多个对象处理一个请求的机会，从而解耦发送者和接受者. 适用性 1.有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。2.你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。3.可处理一个请求的对象集合应被动态指定。

3.9.2、类图



3.9.3、模拟实现

Request 类

Request();构造方法

构造问题对象，应该加上参数 名称和级别

void setRequestLeverl(int requestLeverl);方法

设置问题级别

int getRequestLeverl();方法

得到问题级别

void setRequestName(string requestName);方法

设置问题名称

string getRequestName();方法

得到问题名称

HandleChain 类

HandleChain();构造方法

生成责任链对象，并设置第一个责任人。

void setRequest(Request* Request);方法

调用方法将问题交给第一责任人执行

Handle 类

Handle(string handleName,int handleLevel,Handle* succ);

给定责任人名字、处理问题级别、后继者，构造责任人

Handle(string handleName,int handleLevel);

给定责任人名字、处理问题级别，构造责任人

void SetSuccessor(Handle* succ);//设置后继

设置后继责任人

Handle* GetSuccessor();//得到后继

得到后继责任人

string getHandleName();//得到责任人名字

int getHandleLevel();//得到应处理问题级别

ConcreteHandleA(string handleName,int handleLevel,Handle* succ);

构造方法

ConcreteHandleA(string handleName,int handleLevel);

构造方法

void HandleRequest(Request* request);

如果责任人级别和问题级别相符则处理，否则后面有人后传，后面无人提示

Request.h

#include "iostream"

#include "string"

using namespace std;

class Handle;

class ConcreteHandleA;

class ConcreteHandleB;

class Request

{

public:

Request();

virtual ~Request();

void setRequestLeverl(int requestLeverl);

int getRequestLeverl();

void setRequestName(string requestName);

string getRequestName();

private:

int requestLeverl;

string requestName;

};

```
class HandleChain
{
public:
    HandleChain();
    ~HandleChain(){}
    void setRequest(Request* Request);
private:
    Handle* FirstHandle;
};
```

```
#include "Request.h"
#include "Handle.h"

Request::Request(){}
Request::~Request(){}
void Request::setRequestLeverl(int
requestLeverl)
{
    this->requestLeverl = requestLeverl;
}
int Request::getRequestLeverl()
{
    return requestLeverl;
}
void Request::setRequestName(string
requestName)
{
    this->requestName = requestName;
}

string Request::getRequestName()
{
    return requestName;
}

HandleChain::HandleChain()
{
    Handle* h1 = new
ConcreteHandleB("B4",1);//生成一个处理者
    h1->SetSuccessor(0);
                Handle* h2 = new
ConcreteHandleB("B3",2,h1);//生成一个处理者
```

```
    Handle*  h3  =  new
ConcreteHandleB("B2",3,h2);//生成一个处理者
    Handle*  h4  =  new
ConcreteHandleB("B1",4,h3);//生成一个处理者
    Handle*  h5  =  new
ConcreteHandleA("A5",5,h4);//生成一个处理者
    Handle*  h6  =  new
ConcreteHandleA("A4",6,h5);//生成一个处理者
    Handle*  h7  =  new
ConcreteHandleA("A3",7,h6);//生成一个处理者
    Handle*  h8  =  new
ConcreteHandleA("A2",8,h7);//生成一个处理者
    Handle*  h9  =  new
ConcreteHandleA("A1",9,h8);//生成一个处理者
    this->FirstHandle=h9;
    cout<<"已经生成一个处理者链条，可以根据
问题级别处理，这只是一个测试"<<endl;
}

void  HandleChain::setRequest(Request*
Request)
{
    this->FirstHandle->HandleRequest(Requ
est);
}
```

Handle.h

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

class Request;
class Handle
{public:
virtual ~Handle(){}
    virtual  void  HandleRequest(Request*
request) = 0;//处理问题的方法
    void SetSuccessor(Handle* succ);//设置后
继
    Handle* GetSuccessor();//得到后继
    std::string getHandleName();//得到处理者
名字
    int getHandleLevel();//得到处理级别
protected:
    Handle(string  handleName,int
```

```
handleLevel,Handle* succ);
    Handle(string handleName,int handleLevel);
private:
    Handle* _succ;
    int _handleLevel;
    string _handleName;
};

//假设有两类处理问题的人员。每类有很多个成员
class ConcreteHandleA:public Handle
{
public:
    ConcreteHandleA(string handleName,int handleLevel,Handle* succ);
    ConcreteHandleA(string handleName,int handleLevel);
    ~ConcreteHandleA();
    void HandleRequest(Request* request);
protected:
private:
};
class ConcreteHandleB:public Handle
{
public:
    ConcreteHandleB(string handleName,int handleLevel,Handle* succ);
    ConcreteHandleB(string handleName,int handleLevel);
    ~ConcreteHandleB();
    void HandleRequest(Request* request);
protected:
private:
};
-------------------------------
Handle.cpp
#include "Handle.h"
#include "Request.h"

void Handle::SetSuccessor(Handle* succ){_succ = succ; }//设置后继
Handle* Handle::GetSuccessor(){return _succ;}//得到后继
```

```
string Handle::getHandleName(){return _handleName;}//得到处理者名字
int Handle::getHandleLevel(){return _handleLevel;}//得到处理级别
Handle::Handle(string handleName,int handleLevel,Handle* succ)
:_handleName(handleName),_handleLevel(handleLevel),_succ(succ){}
Handle::Handle(string handleName,int handleLevel)
:_handleName(handleName),_handleLevel(handleLevel){}

ConcreteHandleA::ConcreteHandleA(string handleName,int handleLevel,Handle* succ)
    :Handle(handleName,handleLevel,succ){}
ConcreteHandleA::ConcreteHandleA(string handleName,int handleLevel)
    :Handle(handleName,handleLevel){}
ConcreteHandleA::~ConcreteHandleA(){}

void ConcreteHandleA::HandleRequest(Request* request)
{
    if((this->getHandleLevel())==(request->getRequestLeverl()))
    {
        cout<<this->getHandleName()<<"处理问题:"<<request->getRequestName()<<endl;
    }else if(this->GetSuccessor() != 0)
    {
        cout<<"我是"<<this->getHandleName()<<",问题("<<request->getRequestName()<<")不归我处理,已经后传"<<endl;

        this->GetSuccessor()->HandleRequest(request);
    }
    else
    {
```

```
        cout<<"问                    题
("<<request->getRequestName()<<")没 有 找
到合适的处理者，暂时无法处理，请与管理员联系"
<<endl;
    }
}


ConcreteHandleB::ConcreteHandleB(string
handleName,int handleLevel,Handle* succ)
    :Handle(handleName,handleLevel,succ){}
ConcreteHandleB::ConcreteHandleB(string
handleName,int handleLevel)
    :Handle(handleName,handleLevel){}
ConcreteHandleB::~ConcreteHandleB(){}

void
ConcreteHandleB::HandleRequest(Request*
request)
{
    if((this->getHandleLevel())==(request->
getRequestLeverl()))
    {
        cout<<this->getHandleName()<<"
处      理      问      题      :      "
<<request->getRequestName()<<endl;
    }else if(this->GetSuccessor() != 0)
    {
        cout<<"我                    是
"<<this->getHandleName()<<",问          题
("<<request->getRequestName()<<")不 归 我
处理,已经后传"<<endl;

    this->GetSuccessor()->HandleRequest(re
quest);
    }
    else
    {
        cout<<"我 是 责 任 链 最 后 一 人 :
"<<this->getHandleName()<<",问          题
("<<request->getRequestName()<<")没 有 找
到合适的处理者，暂时无法处理，请与管理员联系"
<<endl;
```

```
    }
}
---------------------------
Main
#include "Request.h"
#include "Handle.h"

int main(int argc, char* argv[])
{
    HandleChain hc;//生成一个链条，在应用时可以根据需要定义链条
    Request* que= new Request();//生成一个问题
    que->setRequestName("打豆豆");//当然可以在构造问题时提供，本例没有这样做
    que->setRequestLeverl(2);//当然可以在构造问题时提供，本例没有这样做*/
    hc.setRequest(que);//问题送入链条

    que->setRequestLeverl(15);//更改问题级别
    hc.setRequest(que);

    //printf("Hello World!\n");
    return 0;
}
```

3.9.4、讨论

Chain of Responsibility 模式的最大的一个有点就是给系统降低了耦合性，请求的发送者完全不必知道该请求会被哪个应答对象处理，极大地降低了系统的耦合性。

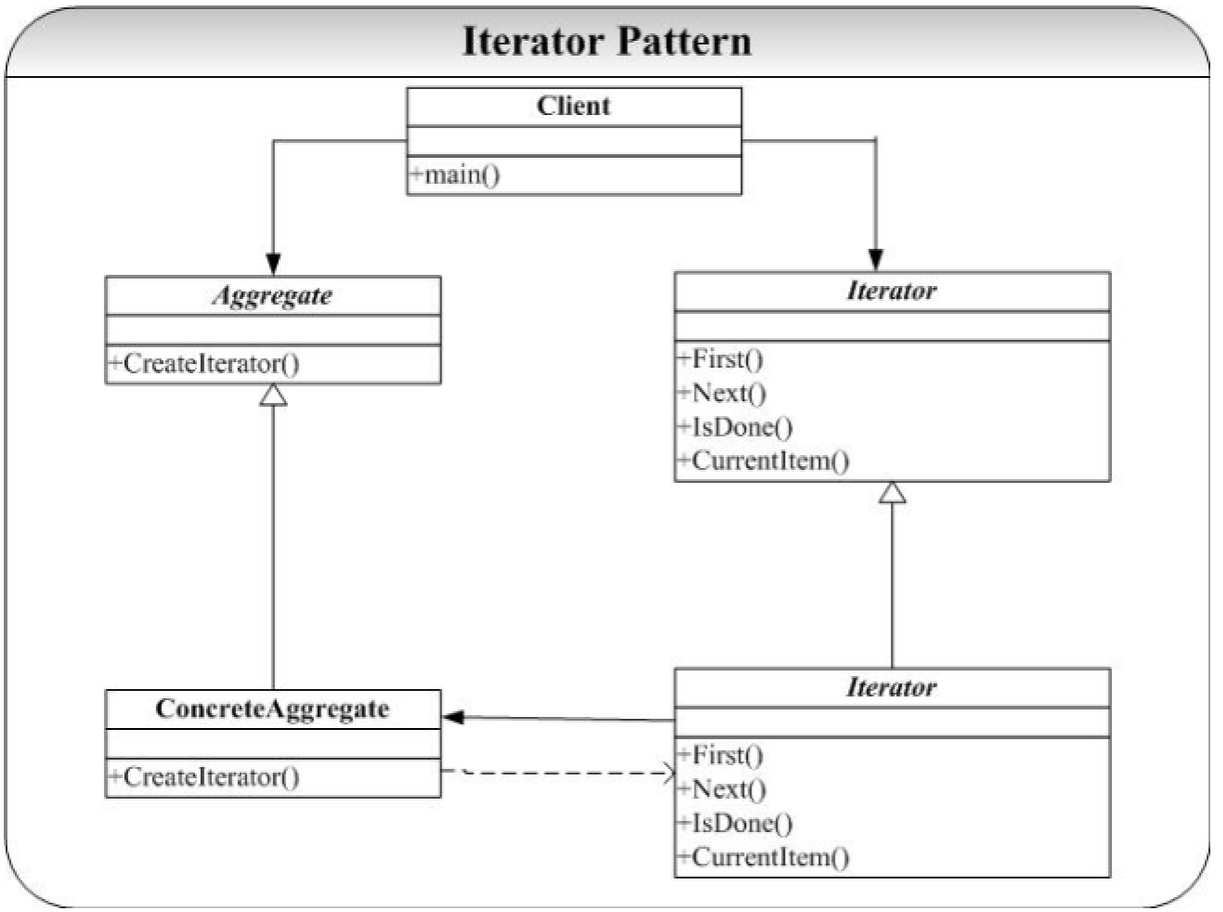
3.10 Iterator 模式

3.10.1、概述

迭代器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。适用性 1.访问一个聚合对象的内容而无需暴露它的内部表示。2.支持对聚合对象的多种遍历。3.为遍历不同的聚合结构提供一个统一的接口(即,支持多态迭代)。

Iterator 模式也正是用来解决对一个聚合对象的遍历问题，将对聚合的遍历封装到一个类中进行，这样就避免了暴露这个聚合对象的内部表示的可能。

3.10.2、类图



3.10.3、模拟实现

程序方法说明：

Aggregate 类

enum{SIZE = 3};

ConcreteAggregate();构造方法

构造集合对象，并为内置数组设置值

Iterator* CreateIterator();方法

用本集合构造一个迭代器，并返回

Object GetItem(int idx);方法

得到集合中某项值

int GetSize();

得到集合大小

Iterator 方法

ConcreteIterator(Aggregate* ag , int idx = 0)

构造方法

构造一个具体迭代器，并设置集合参数 当前序号 idx 默认为 0.

void First();方法

当前序号置零

void Next();方法

当前序号+1

bool IsDone();

是否到尾。

Object CurrentItem();方法

返回集合中当前序号指定值

protected:

private:

Aggregate* _ag;

int _idx;

};

代码:

Aggregate.h

```
class Iterator;
typedef int Object;

class Interator;
class Aggregate
{
public:
    virtual ~Aggregate();
    virtual Iterator* CreateIterator() = 0;
    virtual Object GetItem(int idx) = 0;
    virtual int GetSize() = 0;
protected:
    Aggregate();
private:
};

class ConcreteAggregate:public Aggregate
{
public:
    enum{SIZE = 3};
    ConcreteAggregate();//构造集合
    ~ConcreteAggregate();
    Iterator* CreateIterator();//创建迭代器
```

```
    Object GetItem(int idx);//得到集合中指定
序号值
    int GetSize();//得到集合大小
protected:
private:
    Object _objs[SIZE];
};
```

Aggregate.cpp

```
#include "Aggregate.h"
#include "Iterator.h"
#include <iostream>
using namespace std;

Aggregate::Aggregate(){}
Aggregate::~Aggregate(){}

ConcreteAggregate::ConcreteAggregate()
{
    for (int i = 0; i < SIZE; i++) _objs[i] = i;
}
ConcreteAggregate::~ConcreteAggregate(){}

Iterator* ConcreteAggregate::CreateIterator()
{
    return new ConcreteIterator(this);
}

Object ConcreteAggregate::GetItem(int idx)
{
    if (idx < this->GetSize())
        return _objs[idx];
    else
        return -1;
}

int ConcreteAggregate::GetSize()
{
    return SIZE;
}
```

Iterator.h

```
class Aggregate;
typedef int Object;
class Iterator
{
public:
    virtual ~Iterator();
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() = 0;
    virtual Object CurrentItem() = 0;
protected:
    Iterator();
private:
};

class ConcreteIterator:public Iterator
{
public:
    ConcreteIterator(Aggregate* ag , int idx
= 0);
    ~ConcreteIterator();
    void First();//当前序号置零
    void Next();//当前序号+1
    bool IsDone();//是否到表尾
    Object CurrentItem();//解析并返回当前值
protected:
private:
    Aggregate* _ag;
    int _idx;
};
```

-

Iterator.cpp

```
#include "Iterator.h"
#include "Aggregate.h"
#include <iostream>
using namespace std;

Iterator::Iterator(){}
Iterator::~Iterator(){}

ConcreteIterator::ConcreteIterator(Aggregate
* ag , int idx)
```

```
{
    this->_ag = ag;
    this->_idx = idx;
}
ConcreteIterator::~ConcreteIterator() {}

Object ConcreteIterator::CurrentItem()
{
    return _ag->GetItem(_idx);
}
void ConcreteIterator::First()
{_idx = 0;}

void ConcreteIterator::Next()
{
    if (_idx < _ag->GetSize()) _idx++;
}

bool ConcreteIterator::IsDone()
{
    return (_idx == _ag->GetSize());
}
```

Main

```
#include "Iterator.h"
#include "Aggregate.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    Aggregate* ag = new
ConcreteAggregate();
    Iterator* it = new ConcreteIterator(ag);
    for (; !(it->IsDone());it->Next())
    {
        cout<<it->CurrentItem()<<endl;
    }
//  printf("Hello World!\n");
    return 0;
}
```

3.10.4 、讨论

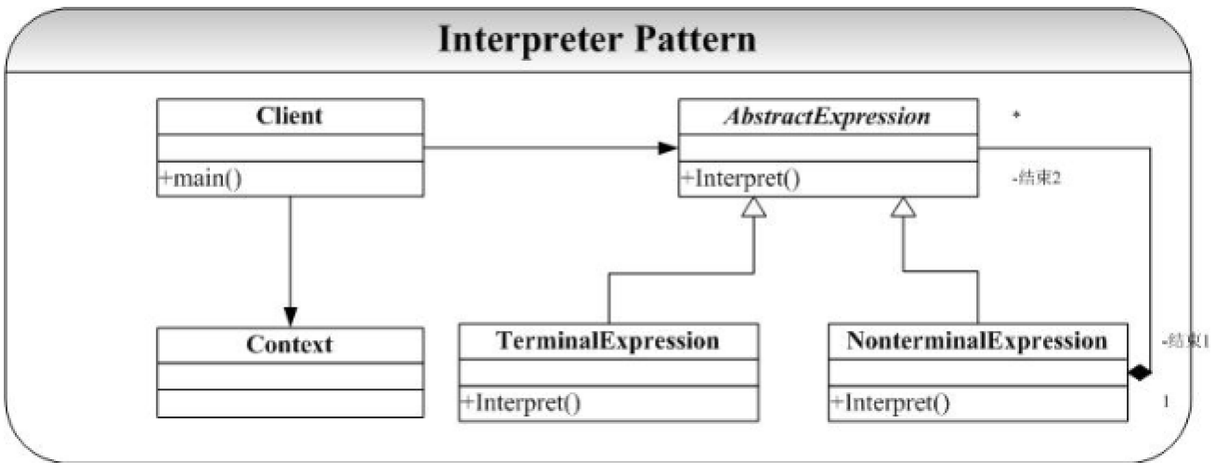
为了更好地保护 Aggregate 的状态，我们可以尽量减小 Aggregate 的 public 接口，而通过将 Iterator 对象声明位 Aggregate 的友元来给予 Iterator 一些特权，获得访问 Aggregate 私有数据和方法的机会。

3.11 Interpreter 模式

3.11.1、概述

解释器模式:给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。 适用性,当有一个语言需要解释执行,并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：1.该文法简单对于复杂的文法,文法的类层次变得庞大而无法管理。2.效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的,而是首先将它们转换成另一种形式。

3.11.2、类图



3.11.3、模拟实现

Context 类：

Context 类：

typedef std::pair<std::string, bool> str_bool;

自定类型

bool Lookup(const char* name) const 方法

返回映射中指定名称对应值

void Assign(VariableExp* ve, bool b);方法

构造一个 字符---布尔映射

AbstractExpression 抽象类

VariableExp:public AbstractExpression 类　变量

VariableExp(const char* name)构造方法

　　构造对象 设置 n_name 字符串变量

bool Interpret(Context& aContext)方法

　　调用 (Context 类 Lookup 方法 得到 n_name 对应值

AndExp：public AbstractExpression 类 对变量进行与运算

AndExp(BooleanExp* op1，BooleanExp* op2) 构造方法

　　构造对象并设置两个变量 m_operand1 m_operand2

bool Interpret(Context& aContext) 方法

　　返回两个变量的与运算

OrExp：public AbstractExpression 类　或运算

基本同上

--+---

NotExp：public AbstractExpression 类　对变量进行非运算

---

class Constant：public AbstractExpression 不变量

程序执行

这个程序的目的是学习解析器模式。方法是构造三个非终端表达式类（与：AndExp，或：OrExp ，非：NotExp ），两个终端表达式类（常量：Constant，单变量：VariableExp）。终端可以直接得到 真、假值，非终端表达式需要进行递归运算。

表达式中的子表达式靠多态自动调用相应类对象的解析方法。常量对象可以直接得到值（真假）。变量对象需要查 Context 表。每个变量分内部名字（Context 表中映射关键字）和外部名字（供用户使用）。

Expression.h

```
class Context;

//抽象表达式类
class AbstractExpression
{
public:
    AbstractExpression(){}
```

```
    virtual ~AbstractExpression(){}
    virtual bool Interpret(Context&)=0;
protected:
private:
};

//终端表达式（常数）
class ConstantExp: public AbstractExpression
{
public:
    //构造常数
    ConstantExp(bool b);
    virtual ~ConstantExp(){}
    //常数直接返回值
    bool Interpret(Context& aContext);
private:
    bool m_bConstant;
};

//定义一个变量对象（终端解析式），并指定他的内部名称
class VariableExp:public AbstractExpression
{
public:
    VariableExp(const char* name);
    //按内部名称在 解析器中查值
    bool Interpret(Context& aContext);
    char* m_name;
};

//非终端表达式： 与运算
class AndExp : public AbstractExpression
{
public:
    AndExp(AbstractExpression*          op1,
AbstractExpression* op2);
    virtual ~AndExp(){}
    bool Interpret(Context& aContext);
private:
    AbstractExpression* m_operand1;
    AbstractExpression* m_operand2;
};
//非终端表达式： 或运算
class OrExp : public AbstractExpression
{
```

```
public:
    OrExp(AbstractExpression*          op1,
AbstractExpression* op2);
    virtual ~OrExp(){}
    bool Interpret(Context& aContext);
private:
    AbstractExpression* m_operand1;
    AbstractExpression* m_operand2;
};
//非终端表达式：  非运算
class NotExp:public AbstractExpression
{
public:
    NotExp(AbstractExpression* op);
    virtual ~NotExp(){}
    bool Interpret(Context& aContext);
private:
    AbstractExpression* m_operand;
};
----------------------------
Expression.cpp
#include "Expression.h"
#include "string"
#include "Context.h"

//终端表达式（常数）
ConstantExp::ConstantExp(bool b)
{
    m_bConstant = b;
}
     //常数直接返回值
bool          ConstantExp::Interpret(Context&
aContext)
{
    return m_bConstant;
}

//定义一个变量对象（终端解析式），并指定他的内
部名称
VariableExp::VariableExp(const char* name)
{
    m_name = strdup(name);
}
     //按内部名称在 解析器中查值
bool          VariableExp::Interpret(Context&
```

```
aContext)
{
    return aContext.Lookup(m_name);
}

//非终端表达式： 与运算
AndExp::AndExp(AbstractExpression*    op1,
AbstractExpression* op2)
{
    m_operand1 = op1;
    m_operand2 = op2;
}
   //两个表达式分别计算，结果相与
bool AndExp::Interpret(Context& aContext)
{
    return m_operand1->Interpret(aContext)
&& m_operand2->Interpret(aContext);
}

//非终端表达式： 或运算
OrExp::OrExp(AbstractExpression*       op1,
AbstractExpression* op2)
{
    m_operand1 = op1;
    m_operand2 = op2;
}
   //两个表达式分别计算，结果相或
bool OrExp::Interpret(Context& aContext)
{
    return m_operand1->Interpret(aContext)
|| m_operand2->Interpret(aContext);
}

//非终端表达式： 非运算
NotExp::NotExp(AbstractExpression* op)
{
    this->m_operand = op;
}
   //两个表达式分别计算，结果相非
bool NotExp::Interpret(Context& aContext)
{
    return !m_operand->Interpret(aContext);
}
-------------------------------------
```

Context.h

```
#include "string"
#include <map>
using namespace std;

class VariableExp;

class Context
{
public:
    typedef std::pair<std::string, bool> str_bool;
    Context();
    virtual ~Context();
    bool Lookup(const char* name) const;
    //形成变量内部名称和值的映射
    void Assign(VariableExp* ve, bool b);
private:
    std::map<std::string, bool> m_varMap;
};
```

Context.cpp

```
#include "Context.h"
#include "Expression.h"
Context::Context(){}
Context::~Context(){}

bool Context::Lookup(const char* name)
const
{
    return m_varMap.find(name)->second;
}
//形成变量内部名称和值的映射
void Context::Assign(VariableExp* ve,bool b)
{
    m_varMap.insert(str_bool(ve->m_name,
b));
}
```

Main

```
#include "Context.h"
```

```
#include "Expression.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    //生成表达式
    AbstractExpression* expression;
    //解析器对象
    Context context;
    //两个变量
    VariableExp* varX = new VariableExp("keyX");
    VariableExp* varY = new VariableExp("keyY");
    /*一个复杂表达式 或（与（常量，变量），与（变量，非（变量）））*/
    expression = new OrExp(
        new AndExp(new ConstantExp(true), varX),
        new AndExp(varY, new NotExp(varX)));
    //在解析器中建立外部名字和值的关联
    context.Assign(varX, false);
    context.Assign(varY, true);
    //递归运算表达式求值
    bool result = expression->Interpret(context);
    cout<<"result: "<<result<<endl;
    printf("Hello World!\n");
    return 0;
}
```

3.11.4、讨论