设计模式——行为型模式

设计模式(design pattern)是在长时间实践中，开发人员总结出的**优秀架构与解决方案。**学习设计模式将有助于经验不足的开发人员在实际开发中，灵活地运用面向对象特性，并能够快速构建不同场景下的程序框架，写出优质代码。

设计模式分类

- **行为型模式**

  关注对象行为功能上的抽象，从而提升对象在行为功能上的可拓展性，能以最少的代码变动完成功能的增减。

- **结构型模式**

  关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性，能在结构层面上尽可能的解耦合。

- **创建型模式**

  （本学期不涉及）

本章内容主要探讨设计模式中的行为型模式。

行为型模式分类

本章介绍三种行为型模式。

- 模版方法(Template Method)模式
- 策略(Strategy)模式
- 迭代器(Iterator)模式

模版方法Template Method

模版方法是一种针对接口编程的设计。

模版方法的思想是：基类是一个模板（也可以称作原型）。基类体现的是“抽象概念”，里面定义若干个纯虚函数，这些函数提供了这个类的“接口”。

比如，如果一个操作有operation1,operation2,两个步骤，我们可以定义一个基类：

```
class AbstractClass{
  public:
  virtual void Operation1() =0;
  virtual void Operation2() =0;
};
```

这个操作可能有很多个版本。每个版本的实现细节不同。比如，AbstractClass可能是“监视计算机节点的负载状态”的过程，Operation1是“得到总内存”，Operation2是“得到已占用内存”。而这两种操作在不同的环境下，比如Win32和Win64下可能是不同的。这些不同的细节则由子类负责实现。

在使用时，抽象类的算法骨架提供了大致方法，再由这个方法来根据需要调用具体类的实现细节。比如：

```
class ConcreteClassA:public AbstractClass{
  void Operation1(){
    cout<<"Do Operation 1 of A"<<endl;
  }
  void Operation2(){
    cout<<"Do Operation 2 of A"<<endl;
  }
};
```

这样当我们要拓展一种新的实现类时，重新对基类进行继承与实现即可，无需对已有的**实现类**进行修改。

由模板实现多态

现在我们有了很多继承自同一个抽象类的实现类，该怎么实现多态呢?

我们知道，模板多态是依赖于**指针和引用**的。编译器能够通过指针和引用判断实际指向的类型，并且调用实际类型里面override了的虚函数。所以在使用模板模式的时候，常常创建基类指针来调用实现类的函数。

例:

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

class AbstractClass{
  public:
  virtual void Operation1() =0;
  virtual void Operation2() =0;
};
class ConcreteClassA:public AbstractClass{
  public:
  void Operation1(){
    cout<<"Do Operation 1 of A"<<endl;
  }
  void Operation2(){
    cout<<"Do Operation 2 of A"<<endl;
  }
};
class ConcreteClassB:public AbstractClass{
  public:
  void Operation1(){
    cout<<"Do Operation 1 of B"<<endl;
  }
  void Operation2(){
    cout<<"Do Operation 2 of B"<<endl;
  }
};
int main(){
  AbstractClass *abstract;
  abstract=new ConcreteClassB();
  abstract->Operation1();
  abstract->Operation2();
  return 0;
}
```

策略模式Strategy Method

模板模式中，每个实现类里面可能有多个功能(比如Operation1,Operation2...)。可能有两个类，它们各有N个功能，其中只有一个的实现有区别。在模板模式下，我们却必须实现两个类。这就很麻烦。

为了避免这种情况，使用策略模式。

策略模式：定义一系列算法并加以封装，使得这些算法可以互相替换。这样，一种算法就不需要依附于某个实现类了，而是自成一类。

具体而言

- 每一种行为各自有方法虚基类A、B
- 每一个方法基类A有若干具体的方法A_1, A_2, A_3，每一种都会继承A
- 所有对象具有一个对象基类O，对象基类O含有所有的方法基类指针A^*、B^*，从而实现多态

  注意到对象基类如果仅含有方法基类指针，实际上没法调用方法基类的方法，故而还需要调用接口
- 每种对象是一个具体的对象类O_1, O_2, O_3，每种都会继承对象类O，同时让方法基类指针A^*、B^*具体指向方法派生类上A_x、B_y

所有对象

比如下面这个例子：有三种鸭子，MallarDuck（绿头鸭）,RubberDuck（橡皮鸭）,DecoyDuck（诱饵鸭子）

MallarDuck会“quack”地叫，会飞；RubberDuck会“queak”地叫，不会飞；DecoyDuck不会叫，不会飞。

发现所有鸭子都有飞、叫两种功能，但是各自的实现不太一样。我们不在每种鸭子里分别实现，而是在基类鸭子里储存这两种功能的基类指针，然后让鸭子的实现类指向对应的两种功能的实现类。

实现如下：

```
#include <iostream>
//飞行行为，用抽象类表示
class FlyBehavior{//fly方法基类
public:
   virtual ~FlyBehavior(){};
   virtual void fly() =0;
};
//叫声行为，用抽象类表示
class QuackBehavior{//Quack方法基类
public:
   virtual ~QuackBehavior(){};
   virtual void quack()= 0;
};
//鸭子基类，除了会变化的fly和quack外，还有不发生改变的display，performFly，performQuack，swim等方法
class Duck{//对象基类
    //私有数据在下方
public:
```

```
    Duck(FlyBehavior*p_FlyBehavior,QuackBehavior* p_QuackBehavior)//这是构造函数
    {
        pFlyBehavior= p_FlyBehavior;
        pQuackBehavior= p_QuackBehavior;

    }//构造函数
    virtual ~Duck(){};
    virtual void display(){};//display这个方法使用的是模板方法，其实不够优化
    void performFly()//perform是接口，因为指针是私有数据，故而设置了public接口
    {
        pFlyBehavior->fly();//根据传入的指针类型决定fly的具体方法
    }
    void performQuack()
    {
        pQuackBehavior->quack();
    }
private:
    FlyBehavior* pFlyBehavior;
    QuackBehavior* pQuackBehavior;//对象基类储存方法基类指针
};
//实现飞行行为的具体方法类
class FlyWithWings : public FlyBehavior{//每种每一个方法基类A有若干具体的方法，每一种都会继承方法基类
public:
    void fly(){
    std::cout<< ("I'm flying!!")<<std::endl;
    }
};

class FlyNoWay : public FlyBehavior{//只需要方法，不需要数据
public:
    void fly(){
    std::cout<< ("I can't fly")<<std::endl;
    }
};

//实现叫声行为的类
class Quack : public QuackBehavior{
public:
    void quack(){
    std::cout<< ("Quack") <<std::endl;
    }
};

class MuteQuack : public QuackBehavior{
public:
    void quack(){
    std::cout<< ("<< Slience >>")<< std::endl;
    }
};

class Squeak : public QuackBehavior{
public:
    void quack(){
    std::cout<< "Squeak"<<std::endl;
    }
};
```

```
74
75  //绿头鸭类
76  class MallardDuck : public Duck{//继承了对象基类，就已经有了方法基类指针
77  public:
78     MallardDuck(FlyBehavior*fly_behavior = new FlyWithWings(),//构造函数含有缺
    省值
79           QuackBehavior*quack_behavior = new Quack())//注意括号在这里才结尾，之后的
    冒号是赋值列表了
80           :Duck(fly_behavior,quack_behavior){}//绿头鸭的构造函数，飞行和叫声的基类指
    针分别指向了对应的实现类
81
82     void display()
83     {
84        std::cout<< "I'm a real Mallard duck"<< std::endl;
85     }
86  };
87
88  class RubberDuck : public Duck{
89  public:
90    RubberDuck(FlyBehavior*fly_behavior=new
    FlyNoWay(),QuackBehavior*quack_behavior = new
    Squeak()):Duck(fly_behavior,quack_behavior){}
91    void display(){
92      std::cout<<"I'm a Rubber duck"<<std::endl;
93    }
94  };
95
96  class DecoyDuck : public Duck{
97  public:
98    DecoyDuck(FlyBehavior*fly_behavior=new
    FlyNoWay(),QuackBehavior*quack_behavior = new
    MuteQuack()):Duck(fly_behavior,quack_behavior){}
99    void display(){
100     std::cout<<"I'm just a Decoy duck"<<std::endl;
101   }
102 };
103
104 int main()
105 {
106    Duck*mallard = new MallardDuck();//基类指针指向派生类对象
107    mallard->display();//由于display是虚函数，故而实现了多态
108    mallard->performFly();//调用逻辑，performfly是基类非虚函数，直接进入基类函数体。
109     //开始调用pFlyBehavior->fly();
110     //fly是基类虚函数，根据基类指针pFlyBehavior实际指向的对象类型来调用fly函数
111     //回顾MallardDuck类的基类指针pFlyBehavior实际指向的对象，
    FlyBehavior*fly_behavior = new FlyWithWings()，然后把fly_behavior传给了
    pFlyBehavior，故而pFlyBehavior指向fly_behavior，这是一个FlyWithWings方法类的对象
112     //综上所述，调用了FlyWithWings
113    mallard->performQuack();
114    Duck*rubber= new RubberDuck();
115    rubber->display();
116    rubber->performQuack();
117    return 0;
118 }
119 output:
120 I'm a real Mallard duck
121 I'm flying!!
122 Quack
```

```
I'm a Rubber duck
Squeak

```

从上面的例子可以看出，策略模式不过是更加多态化的模板模式罢了。在模板模式下，每个类（比如在上面这个例子里是每种鸭子）中的Fly,Quack函数都在这个鸭子类里实现。而这里则是先定义了Fly，Quack这两种动作的基类——QuackBehavior，FlyBehavior——并且在在基类鸭子里储存了这两种基类类型的指针。

然后，再通过继承关系实现了各种不同的Fly,Quack的实现类，并且在鸭子的实现类里让QuackBehavior，FlyBehavior的基类指针指向不同的Fly,Quack的实现类。