行为型模式是对在不同的对象之间划分责任和算法的抽象化。

行为型模式

- 可以更加清晰地划分类与对象地职责
- 研究系统在运行时实例对象之间地交互。
  在系统运行时，对象可以通过相互通信与协作完成一些复杂地功能，一个对象在运行时也将影响到其他对象地运行。

行为型模式地分类：

- 类行为型模式：使用继承关系在几个类之间分配行为，主要通过多态等方式来分配父类与子类地职责
- 对象行为模式：使用对象的聚合关联关系来分配行为，主要通过对象关联等方式来分配两个或多个类的职责。

行为模式包含的模式：

- 职责链模式
- 命令模式
- 解释器模式
- 迭代器模式
- 中介者模式
- 备忘录模式
- 观察者模式
- 状态模式
- 策略模式
- 模板方法模式
- 访问者模式

主要学习了观察者模式

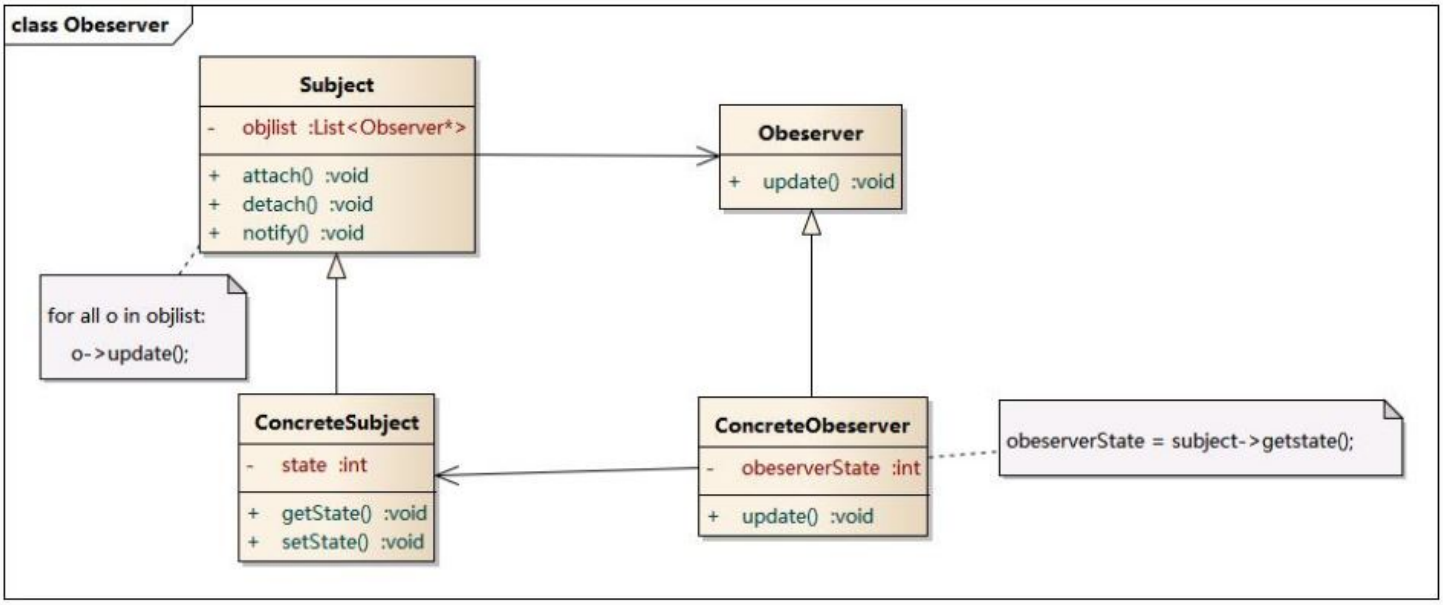
1 观察者模式

定义对象间的一种一对多依赖关系，使得每当一个对象状态发生变化时，其相关依赖对象皆得到通知并被自动跟新。

1.1 模式结构：

- 目标
- 具体目标
- 观察者
- 具体观察者

UML图


class Obeserver
Subject
- objlist :List<Observer*>
+ attach() :void
+ detach() :void
+ notify() :void
Obeserver
+ update() :void
for all o in objlist:
o->update();
ConcreteSubject
- state :int
+ getState() :void
+ setState() :void
ConcreteObeserver
- obeserverState :int
+ update() :void
obeserverState = subject->getstate();

代码

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

class Obeserver
{
public:
        Obeserver(){};
        virtual ~Obeserver(){};
        virtual void update(const int state) = 0;
};

class Subject
{
public:
    static int state;
    Subject(){};
    virtual ~Subject(){};
    //Obeserver *m_Oberserver;

    void attach(Obeserver *pObeserver);
    void detach(Obeserver * pObeserver);
    void notify();
    //virtual int getState()=0;
    virtual void setState()=0;
private:
    vector<Obeserver *> m_vtObj;
};

int Subject::state = 0;
void Subject::attach(Obeserver *pObeserver){
    m_vtObj.push_back(pObeserver);
};

void Subject::detach(Obeserver *pObeserver)
{
    for (vector<Obeserver *>::iterator it = m_vtObj.begin(); it != m_vtObj.end(); it++){
        if((*it) == pObeserver){
            m_vtObj.erase(it);
            return;
        }
    }
};

void Subject::notify()
{
    for (vector<Obeserver *>::iterator it = m_vtObj.begin(); it != m_vtObj.end(); it++){
        (*it)->update(this->state);
    }
}
```

```
class ConcreteSubject: public Subject
{
public:
    ConcreteSubject(){};
    virtual ~ConcreteSubject(){};

    //virtual int getState() { return state; };
    virtual void setState() { state++; };
};

class ConcreteObeserver : public Obeserver
{
public:
    ConcreteObeserver() {
        m_obeserverState = 0;
        m_objName = "";
    };
    ConcreteObeserver(string name);
        virtual ~ConcreteObeserver();
        virtual void update(const int state);

private:
        string m_objName;
        int m_obeserverState;
};

ConcreteObeserver::ConcreteObeserver(string name){
        m_objName = name;
    m_obeserverState = 0;
}

ConcreteObeserver::~ConcreteObeserver(){

}

void ConcreteObeserver::update(const int state){
        m_obeserverState = state;
        cout << "update oberserver[" << m_objName << "] state:" << m_obeserverState << endl;
}

int main()
{
    Obeserver *A = new ConcreteObeserver("A");
    Obeserver *B = new ConcreteObeserver("B");
    Subject *S = new ConcreteSubject();
    S->attach(A);
    S->attach(B);
    S->setState();
    S->notify();
    S->detach(B);
    S->setState();
```

```
    S->notify();
    return 0;
}
```

主要优点:
可以实现表示层和数据逻辑层的分离，并在观察目标和观察者之间建立一个抽象的耦合，支持广播通信;
主要缺点:
在于如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间，而且如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。