观察者模式

资料： C++观察者模式

01.简述

观察者模式（Observer Pattern），定义了对象间的一对多的依赖关系，让多个观察者对象同时监听某一个主题对象（被观察者）。当主题对象的状态发生更改时，会通知所有观察者，让它们能够自动更新。

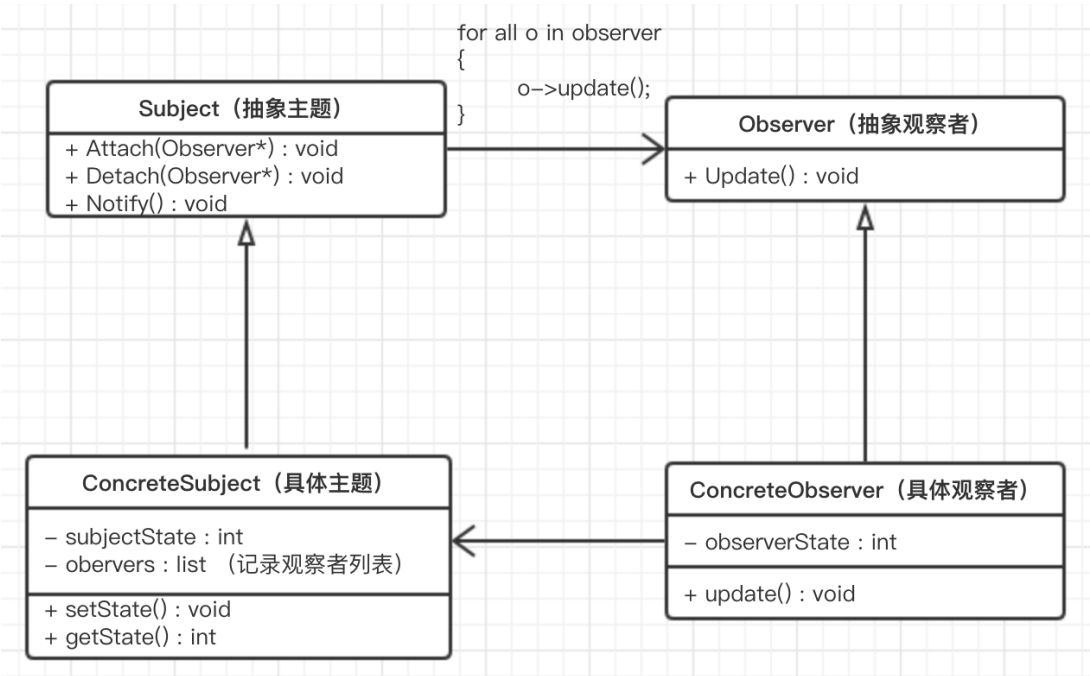
02.背景

很多时候，在应用程序的一部分发生改变时，需要同时更新应用程序的其他部分。有一种方法是：让接受者反复检查发送者来进行更新，但是这种方法存在两个主要问题：

- 占用大量CPU时间来检查新的状态
- 依赖于检测更新的时间间隔（一般采用定时器），可能并不会立即获得更新

对于这个问题，有一个简单的解决方案 - 观察者模式。

03.UML结构图



- Subject（抽象主题）：跟踪所有观察者，并提供添加和删除观察者的接口

- Observe（抽象观察者）：为所有观察者定义一个接口，在得到主题的通知时进行自我更新
- ConcreteSubject（具体主题）：将有关状态存入ConcreteObserver对象。当具体主题的状态发生任何改变是，通知所有观察者
- ConcreteObserve（具体观察者）：实线Observe所要求的更新接口，以便使本身的状态与主题的状态相协调

04.代码

```
//
//  T4_20190119.h
//  DesignPattern
//
//  Created by shadot on 2019/1/19.
//  Copyright © 2019 shadot. All rights reserved.
//

#ifndef T4_20190119_h
#define T4_20190119_h

#include <iostream>
#include <list>

using namespace std;

//观察者模式

class IObserver;

//抽象主题
class ISubject
{
public:
    virtual void Attach(IObserver*) = 0; //添加观察者
    virtual void Detach(IObserver*) = 0; //删除观察者
    virtual void Notify() = 0; //通知观察者
};

//抽象观察者
class IObserver
{
public:
    virtual void Update(int price) = 0; //更新价格
};

```

```
//具体主题
class ConcreteSubject:public ISubject
{
public:
    ConcreteSubject(){
        m_observers.clear();
        m_nprice = 10;
    }
    void SetPrice(int nprice){
        m_nprice = nprice;
    }
    void Attach(IObserver* observer){
        m_observers.push_back(observer);
    }
    void Detach(IObserver* observer){
        m_observers.remove(observer);
    }
    void Notify(){
        for (auto it : m_observers){
            it->Update(m_nprice);
        }
    }
private:
    list<IObserver*> m_observers; //观察者列表
    int m_nprice; //通知观察者更新价格
};

//具体观察者
class ConcreteObserver:public IObserver
{
public:
    ConcreteObserver(const string& name){
        m_strname = name;
    }
    void Update(int price){
        cout << m_strname << " - price: " << price << endl;
    }
private:
    string m_strname;
};

#endif /* T4_20190119_h */

//T4_20190119
    {
        // 创建主题、观察者
```

```
        ConcreteSubject* PSubject = new ConcreteSubject();
        IObserver* pObserver1 = new ConcreteObserver("Tom");
        IObserver* pObserver2 = new ConcreteObserver("Jik");

        //添加观察者
        PSubject->Attach(pObserver1);
        PSubject->Attach(pObserver2);
        PSubject->SetPrice(15);
        PSubject->Notify();

        PSubject->Detach(pObserver2);
        PSubject->SetPrice(20);
        PSubject->Notify();
    }
```

```
Tom - price: 15
Jik - price: 15
Tom - price: 20
```

一个实际案例:

```
//实现的功能类似于item1内部发生改变时通知item2也发生改变，item2内部发生改变时通知item也改变

class IDAS_ChildBaseItem
{
public:
    IDAS_ChildBaseItem(){};
    virtual void Attach(IDAS_ChildBaseItem*){};
    virtual void Detach(IDAS_ChildBaseItem*){};
    virtual void Notify(){};
    virtual void Update(int index){};
};

class IDAS_Child_1_Item:public IDAS_ChildBaseItem
{
public:
    IDAS_Child_1_Item(const string& name):IDAS_ChildBaseItem(){
        m_nIndex = 10;
        m_strName = name;
    };
    void SetIndex(int index){
        m_nIndex = index;
        Notify();
    }
    void Attach(IDAS_ChildBaseItem* item){
```

```
        m_listItem.push_back(item);
    }
    void Detach(IDAS_ChildBaseItem* item){
        m_listItem.remove(item);
    }
    void Notify(){
        for(auto item : m_listItem){
            item->Update(m_nIndex);
        }
    }
    void Update(int index){
        cout << m_strName << " index : " << index << endl;
    }
private:
    string m_strName;
    int m_nIndex;
    list<IDAS_ChildBaseItem*> m_listItem;
};

class IDAS_Child_2_Item:public IDAS_ChildBaseItem
{
public:
    //实现同IDAS_Child_1_Item
};

IDAS_Child_1_Item* item1 = new IDAS_Child_1_Item("item1");
IDAS_Child_2_Item* item2 = new IDAS_Child_2_Item("item2");

item1->Attach(item2);
item2->Attach(item1);

item1->SetIndex(5);

item2->SetIndex(20);

//输出结果
//item2 index : 5
//item1 index : 20
```

05.编译过程遇到的问题

此报错是因为实例化子类时，所有父类的纯虚函数必须全部重新实现，否则就会报错

```cpp
//抽象主题
class ISubject
{
public:
    virtual void Attach(IObserver*) = 0; //添加观察者
    virtual void Detach(IObserver*) = 0; //删除观察者
    virtual void Notify() = 0; //通知观察者
};

//具体主题
class ConcreteSubject:public ISubject
{
public:
    ConcreteSubject(){...}
    void SetPrice(int nprice){...}
    void Attach(IObserver* observer){...}
    void Detach(IObserver* observer){...}
    void Netify(){
        for (auto it : m_observers){
            it->Update(m_nprice);
        }
    }
private:
    list<IObserver*> m_observers; //观察者列表
    int m_nprice; //通知观察者更新价格
};
```