

观察者模式

资料：C++观察者模式

观察者模式

- 01.简述
- 02.背景
- 03.UML结构图
- 04.代码
- 05.编译过程遇到的问题

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。

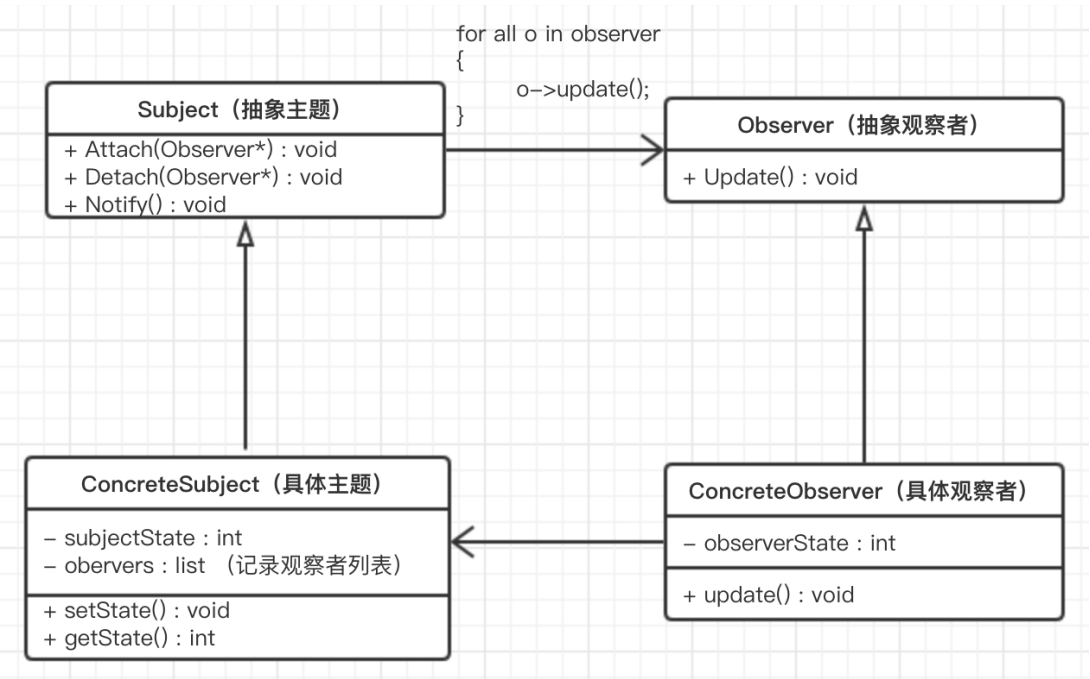
02.背景

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）：实现**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

```
1 //
2 //  T4_20190119.h
3 //  DesignPattern
4 //
5 //  Created by shadot on 2019/1/19.
6 //  Copyright © 2019 shadot. All rights reserved.
7 //
8
9 #ifndef T4_20190119_h
10 #define T4_20190119_h
11
12 #include <iostream>
13 #include <list>
14
15 using namespace std;
16
17 //观察者模式
18
19 class IObserver;
20
21 //抽象主题
22 class ISubject
23 {
24 public:
25     virtual void Attach(IObserver*) = 0; //添加观察者
26     virtual void Detach(IObserver*) = 0; //删除观察者
27     virtual void Notify() = 0; //通知观察者
28 };
29
30 //抽象观察者
31 class IObserver
32 {
33 public:
34     virtual void Update(int price) = 0; //更新价格
35 };
36
```

```

37 //具体主题
38 class ConcreteSubject:public ISubject
39 {
40 public:
41     ConcreteSubject(){
42         m_observers.clear();
43         m_nprice = 10;
44     }
45     void SetPrice(int nprice){
46         m_nprice = nprice;
47     }
48     void Attach(IObserver* observer){
49         m_observers.push_back(observer);
50     }
51     void Detach(IObserver* observer){
52         m_observers.remove(observer);
53     }
54     void Notify(){
55         for (auto it : m_observers){
56             it->Update(m_nprice);
57         }
58     }
59 private:
60     list<IObserver*> m_observers; //观察者列表
61     int m_nprice; //通知观察者更新价格
62 };
63
64 //具体观察者
65 class ConcreteObserver:public IObserver
66 {
67 public:
68     ConcreteObserver(const string& name){
69         m_strname = name;
70     }
71     void Update(int price){
72         cout << m_strname << " - price: " << price << endl;
73     }
74 private:
75     string m_strname;
76 };
77
78 #endif /* T4_20190119_h */
79
80 //T4_20190119
81 {
82     // 创建主题、观察者

```

```

83         ConcreteSubject* PSubject = new ConcreteSubject();
84         IObserver* pobserver1 = new ConcreteObserver("Tom");
85         IObserver* pobserver2 = new ConcreteObserver("Jik");
86
87         //添加观察者
88         PSubject->Attach(pobserver1);
89         PSubject->Attach(pobserver2);
90         PSubject->SetPrice(15);
91         PSubject->Notify();
92
93         PSubject->Detach(pobserver2);
94         PSubject->SetPrice(20);
95         PSubject->Notify();
96     }

```

```

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

```

一个实际案例：

```

1  //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
2
3  class IDAS_ChildBaseItem
4  {
5  public:
6      IDAS_ChildBaseItem(){};
7      virtual void Attach(IDAS_ChildBaseItem*){};
8      virtual void Detach(IDAS_ChildBaseItem*){};
9      virtual void Notify(){};
10     virtual void Update(int index){};
11 };
12
13 class IDAS_Child_1_Item:public IDAS_ChildBaseItem
14 {
15 public:
16     IDAS_Child_1_Item(const string& name):IDAS_ChildBaseItem(){
17         m_nIndex = 10;
18         m_strName = name;
19     };
20     void SetIndex(int index){
21         m_nIndex = index;
22         Notify();
23     }
24     void Attach(IDAS_ChildBaseItem* item){

```

```

25         m_listItem.push_back(item);
26     }
27     void Detach(IDAS_ChildBaseItem* item){
28         m_listItem.remove(item);
29     }
30     void Notify(){
31         for(auto item : m_listItem){
32             item->Update(m_nIndex);
33         }
34     }
35     void Update(int index){
36         cout << m_strName << " index : " << index << endl;
37     }
38 private:
39     string m_strName;
40     int m_nIndex;
41     list<IDAS_ChildBaseItem*> m_listItem;
42 };
43
44 class IDAS_Child_2_Item:public IDAS_ChildBaseItem
45 {
46 public:
47     //实现同IDAS_Child_1_Item
48 };
49
50 IDAS_Child_1_Item* item1 = new IDAS_Child_1_Item("item1");
51 IDAS_Child_2_Item* item2 = new IDAS_Child_2_Item("item2");
52
53 item1->Attach(item2);
54 item2->Attach(item1);
55
56 item1->SetIndex(5);
57
58 item2->SetIndex(20);
59
60 //输出结果
61 //item2 index : 5
62 //item1 index : 20

```

05.编译过程遇到的问题

```

{
    ConcreteSubject* PSubject = new ConcreteSubject();
}

```

! Allocating an object of abstract class type 'ConcreteSubject'

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

```
21 //抽象主题
22 class ISubject
23 {
24 public:
25     virtual void Attach(IObserver*) = 0; //添加观察者
26     virtual void Detach(IObserver*) = 0; //删除观察者
27     virtual void Notify() = 0; //通知观察者
28 };
29 |
30 //具体主题
31 class ConcreteSubject:public ISubject
32 {
33 public:
34     ConcreteSubject(){ ... }
38     void SetPrice(int nprice){ ... }
41     void Attach(IObserver* observer){ ... }
44     void Detach(IObserver* observer){ ... }
47     void Netify(){
48         for (auto it : m_observers){
49             it->Update(m_nprice);
50         }
51     }
52 private:
53     list<IObserver*> m_observers; //观察者列表
54     int m_nprice; //通知观察者更新价格
55 };
```

子类函数名写错

