

cly

博客园 首页 新随笔 联系 订阅 管理 50 Posts :: 2 Stories :: 21 Comments :: 0 Trackbacks

公告

昵称: 戒色
园龄: 6年3个月
粉丝: 47
关注: 0
+加关注

搜索

找找看
 谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

设计模式(17)
算法 排序(4)
引用(1)
执行顺序(1)
算法(1)
算法 查找(1)
abstractFactory抽象工厂(1)
Adapter适配器模式(1)
Bridge桥接模式(1)
Builder建造者模式(1)
更多

随笔分类(55)

面向对象(29)
设计模式(19)
算法, 数据结构(7)

随笔档案(50)

2013年9月 (1)
2013年7月 (19)
2013年6月 (8)
2012年11月 (1)
2012年7月 (7)
2012年6月 (8)
2011年7月 (1)
2011年5月 (5)

最新评论

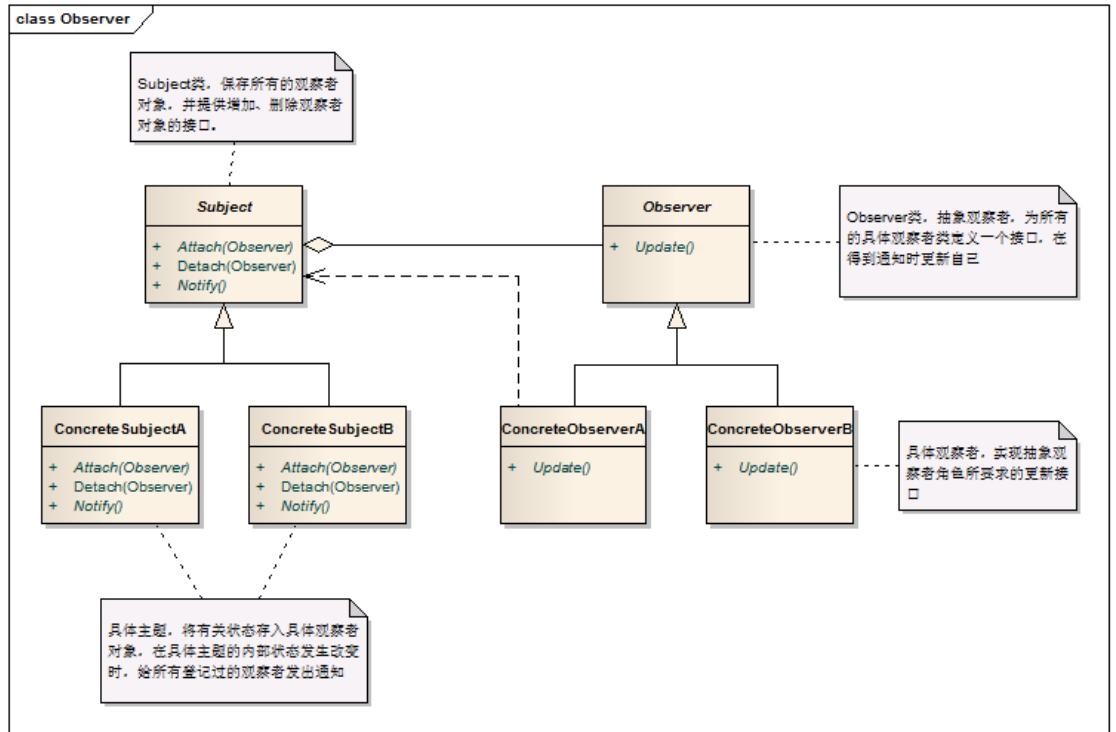
1. Re:C++设计模式-Flyweight享元模式
区分内外状态是实现享元模式的关键啊
--envoy
2. Re:C++设计模式-State状态模式
代码没有清晰的表达出状态模式的精髓, 感觉很空洞
--monalood
3. Re:C++设计模式-Observer观察者模式
蛮好的
--hust_hunter
4. Re:C++设计模式-Builder建造者模式
好, 很好
--奋斗牛牛
5. Re:C++设计模式-Strategy策略模式
你这main函数里面调用, new 实例化, 后面没有delete, 不会造成内存泄露吗
--一起经历

C++设计模式-Observer观察者模式

Observer观察者模式

作用: 观察者模式定义了一种一对多的依赖关系, 让多个观察者对象同时监听某一个主题对象, 这个主题对象在状态发生变化时, 会通知所有观察者对象, 使它们能够自动更新自己

UML图:



Subject类, 可翻译为主题或抽象通知者, 一般用一个抽象类或者一个接口实现。它把所有对观察者对象的引用保存在一个聚集里, 每个主题都可以有任何数量的观察者。抽象主题提供一个接口, 可以增加和删除观察者对象。

Observer类, 抽象观察者, 为所有的具体观察者定义一个接口, 在得到主题的通知时更新自己。这个接口叫做更新接口。抽象观察者一般用一个抽象类或者一个接口实现。更新接口通常包含一个Update()方法。

ConcreteSubject类, 叫做具体主题或具体通知者, 将有关状态存入具体通知者对象; 在具体主题的内部状态改变时, 给所有等级过的观察者发出通知。通常用一个具体子类实现。

ConcreteObserver类, 具体观察者, 实现抽象观察者角色所要求的更新接口, 以便使本身的状态与主题的状态相协调。具体观察者角色可以保存一个指向一个具体主题对象的引用。

特点: 将一个系统分割成一系列相互协作的类有一个很不好的副作用, 那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合, 这样会给维护、扩展和重用都带来不便。

何时使用:

当一个对象的改变需要同时改变其他对象的时候, 而且它不知道具体有多少对象有待改变时, 应该考虑使用观察者模式。

观察者模式所做的工作其实就是在解除耦合。让耦合的双方都依赖于抽象, 而不是依赖于具体。从而使得各自的变化都不会影响另一边的变化。

代码如下:

Observer.h

```
1 #ifndef _OBSERVER_H_
2 #define _OBSERVER_H_
3
4 #include <string>
5 #include <list>
6 using namespace std;
```

阅读排行榜	7
1. C++设计模式-Observer观察者模式(12911)	8 class Subject;
2. C++设计模式-Singleton(7935)	9
3. C++设计模式-Factory工厂模式(7238)	10 class Observer
4. C++设计模式-Adapter适配器模式(6724)	11 {
5. C++设计模式-Bridge桥接模式(5543)	12 public:
	13 ~Observer();
	14 virtual void Update (Subject*)=0;
	15 protected:
	16 Observer();
	17 private:
	18 };
	19
	20 class ConcreteObserverA : public Observer
	21 {
	22 public:
	23 ConcreteObserverA();
	24 ~ConcreteObserverA();
	25 virtual void Update (Subject*);
	26 protected:
	27 private:
	28 string m_state;
	29 };
	30
	31 class ConcreteObserverB : public Observer
	32 {
	33 public:
	34 ConcreteObserverB();
	35 ~ConcreteObserverB();
	36 virtual void Update (Subject*);
	37 protected:
	38 private:
	39 string m_state;
	40 };
	41
	42 class Subject
	43 {
	44 public:
	45 ~Subject();
	46 virtual void Notify();
	47 virtual void Attach (Observer*);
	48 virtual void Detach (Observer*);
	49 virtual string GetState();
	50 virtual void SetState(string state);
	51 protected:
	52 Subject();
	53 private:
	54 string m_state;
	55 list<Observer*> m_lst;
	56 };
	57
	58 class ConcreteSubjectA : public Subject
	59 {
	60 public:
	61 ConcreteSubjectA();
	62 ~ConcreteSubjectA();
	63 protected:
	64 private:
	65 };
	66
	67 class ConcreteSubjectB : public Subject
	68 {
	69 public:
	70 ConcreteSubjectB();
	71 ~ConcreteSubjectB();
	72 protected:
	73 private:
评论排行榜	
1. C++设计模式-Observer观察者模式(6)	
2. C++设计模式-Singleton(4)	
3. C++设计模式-Factory工厂模式(2)	
4. C++设计模式-Bridge桥接模式(2)	
5. C++设计模式-State状态模式(2)	
推荐排行榜	
1. C++设计模式-Bridge桥接模式(3)	
2. C++设计模式-Singleton(2)	
3. C++设计模式-Factory工厂模式(2)	
4. C++设计模式-Observer观察者模式(2)	
5. C++设计模式-Facade模式(2)	

```
74 };  
75  
76 #endif
```



Observer.cpp




```
1 #include "Observer.h"  
2 #include <iostream>  
3 #include <algorithm>  
4  
5 using namespace std;  
6  
7 Observer::Observer()  
8 {}  
9  
10 Observer::~Observer()  
11 {}  
12  
13 ConcreteObserverA::ConcreteObserverA()  
14 {}  
15  
16 ConcreteObserverA::~ConcreteObserverA()  
17 {}  
18  
19 void ConcreteObserverA::Update(Subject* pSubject)  
20 {  
21     this->m_state = pSubject->GetState();  
22     cout << "The ConcreteObserverA is " << m_state << std::endl;  
23 }  
24  
25 ConcreteObserverB::ConcreteObserverB()  
26 {}  
27  
28 ConcreteObserverB::~ConcreteObserverB()  
29 {}  
30  
31 void ConcreteObserverB::Update(Subject* pSubject)  
32 {  
33     this->m_state = pSubject->GetState();  
34     cout << "The ConcreteObserverB is " << m_state << std::endl;  
35 }  
36  
37 Subject::Subject()  
38 {}  
39  
40 Subject::~Subject()  
41 {}  
42  
43 void Subject::Attach(Observer* pObserver)  
44 {  
45     this->m_lst.push_back(pObserver);  
46     cout << "Attach an Observer\n";  
47 }  
48  
49 void Subject::Detach(Observer* pObserver)  
50 {  
51     list<Observer*>::iterator iter;  
52     iter = find(m_lst.begin(), m_lst.end(), pObserver);  
53     if(iter != m_lst.end())  
54     {  
55         m_lst.erase(iter);  
56     }  
57     cout << "Detach an Observer\n";  
58 }
```

```
59
60 void Subject::Notify()
61 {
62     list<Observer*>::iterator iter = this->m_lst.begin();
63     for(; iter != m_lst.end(); iter++)
64     {
65         (*iter)->Update(this);
66     }
67 }
68
69 string Subject::GetState()
70 {
71     return this->m_state;
72 }
73
74 void Subject::SetState(string state)
75 {
76     this->m_state = state;
77 }
78
79 ConcreteSubjectA::ConcreteSubjectA()
80 {}
81
82 ConcreteSubjectA::~ConcreteSubjectA()
83 {}
84
85 ConcreteSubjectB::ConcreteSubjectB()
86 {}
87
88 ConcreteSubjectB::~ConcreteSubjectB()
89 {}
```



main.cpp

```

#include "Observer.h"
#include <iostream>

using namespace std;

int main()
{
    Observer* p1 = new ConcreteObserverA();
    Observer* p2 = new ConcreteObserverB();
    Observer* p3 = new ConcreteObserverA();

    Subject* pSubject = new ConcreteSubjectA();
    pSubject->Attach(p1);
    pSubject->Attach(p2);
    pSubject->Attach(p3);

    pSubject->SetState("old");

    pSubject->Notify();

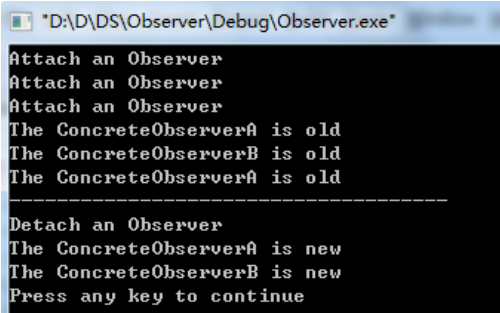
    cout << "-----" << endl;
    pSubject->SetState("new");

    pSubject->Detach(p3);
    pSubject->Notify();

    return 0;
}
```



结果如下：



分类: [面向对象](#), [设计模式](#)

标签: [设计模式](#), [Observer观察者模式](#)

好文要顶

关注我

收藏该文

戒色

关注 - 0

粉丝 - 47

2

0

+加关注

« 上一篇: [C++设计模式-State状态模式](#)
» 下一篇: [C++设计模式-Memento备忘录模式](#)

posted on 2013-07-11 11:37 戒色 阅读(12911) 评论(6) 编辑 收藏

Feedback

- #1楼 2014-09-19 14:40 木石

写的非常好！

支持(0) 反对(0)
- #2楼 2014-11-20 21:46 wenwenxiong

subject具有list<Observer*> m_lst;成员，在使用过程中attach和detach动态创建的Observer子类对象的指针，subject的析构函数应该负责释放list容器中指针所指的内存。detach在执行 m_lst.erase(iter);前应该delete *iter。subject的析构函数应该为

```
Subject::~~Subject()
{
    list<Observer*>::iterator iter = this->m_lst.begin();
    for(;iter != m_lst.end();iter++)
    {
        delete *iter;
    }
    m_lst.clear();
}
```

支持(0) 反对(1)
- #3楼 2015-01-20 11:59 汪飞

@ wenwenxiong

反对你这种做法，比如说你关注了某人的博客，如果别人的博客删除了，难道要将你的账号注销掉么

支持(0) 反对(0)

#4楼 2015-01-28 13:06 春夏

文中说的list中保存观察者的引用改为指针（引用不符合容器元素要求），二楼所说的让Subject负责释放Observer的指针不是太好，因为注册时Observer的指针时有用户创建而不是有Subject创建，所以Observer对象指针的创建和释放全部有用户来控制比较好。

支持(0) 反对(0)

#5楼 2015-01-28 13:06 春夏

总体写的不错~

支持(0) 反对(0)

#6楼 2015-07-07 19:38 hust_hunter

蛮好的

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#) 网站首页。

- 【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
- 【福利】Microsoft Azure给博客园的你专属双重好礼
- 【推荐】融云发布 App 社交化白皮书 IM 提升活跃超 8 倍
- 【直播】周三，微软全球黑带专家直播部署Web应用
- 【推荐】BPM免费下载



- 最新IT新闻：
- 优酷十周年：杨伟东说视频网站竞争将进入平台生态时代
 - 大赞！京东圣诞物流升级 多花1元准时送达
 - DeepMind计划在加州组建研发团队 增加与谷歌总部交流协同
 - 唐只是一方面 浅谈《马里奥Run》关卡设计
 - 招财宝再现逾期 保险公司称“索赔资料不全”
- » 更多新闻...



- 最新知识库文章：
- 写给未来的程序媛
 - 高质量的工程代码为什么难写
 - 循序渐进地代码重构
 - 技术的正宗与野路子
 - 陈皓：什么是工程师文化？
- » 更多知识库文章...

