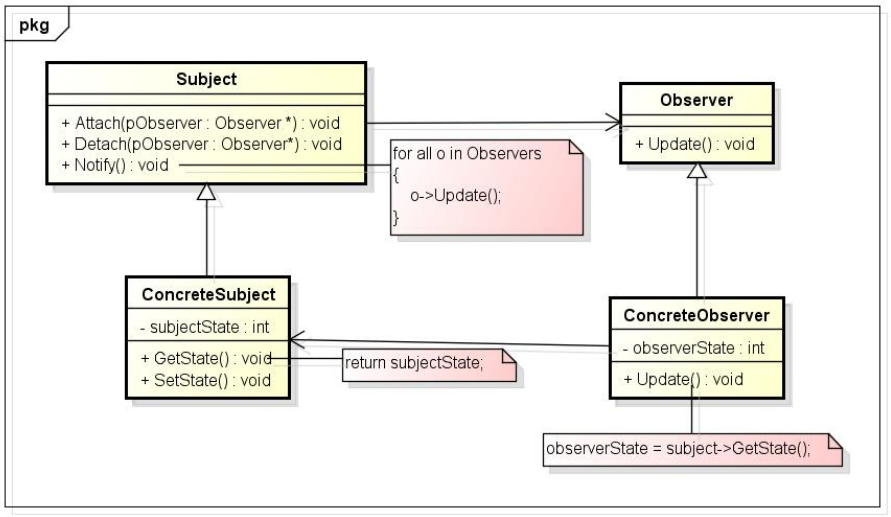
设计模式之观察者模式（c++）

Observer 模式应该可以说是应用最多、影响最广的模式之一，因为 Observer 的一个实例 Model/View/Control（ MVC ） 结构在系统开发架构设计中有着很重要的地位和意义， MVC实现了业务逻辑和表示层的解耦。在 MFC 中， Doc/View（文档视图结构）提供了实现 MVC 的框架结构。在 Java 阵容中， Struts 则提供和 MFC 中 Doc/View 结构类似的实现 MVC 的框架。另外 Java 语言本身就提供了 Observer 模式的实现接口。当然， MVC 只是 Observer 模式的一个实例。 Observer 模式要解决的问题为： 建立一个一（ Subject ）对多（ Observer ） 的依赖关系， 并且做到当"一" 变化的时候， 依赖这个"一"的多也能够同步改变。

在GOF的《设计模式:可复用面向对象软件的基础》一书中对观察者模式是这样说的：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。当一个对象发生了变化，关注它的对象就会得到通知；这种交互也称为发布-订阅(publish-subscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者。

最常见的一个例子就是： 对同一组数据进行统计分析时候， 我们希望能够提供多种形式的表示 （例如以表格进行统计显示、柱状图统计显示、百分比统计显示等）。这些表示都依赖于同一组数据， 我们当然需要当数据改变的时候， 所有的统计的显示都能够同时改变。 Observer 模式就是解决了这一个问题。

UML类图：



Subject（目标）
——目标知道它的观察者。可以有任意多个观察者观察同一个目标；
——提供注册和删除观察者对象的接口。

Observer（观察者）
——为那些在目标发生改变时需获得通知的对象定义一个更新接口。

ConcreteSubject（具体目标）
——将有关状态存入各ConcreteObserver对象；
——当它的状态发生改变时，向它的各个观察者发出通知。

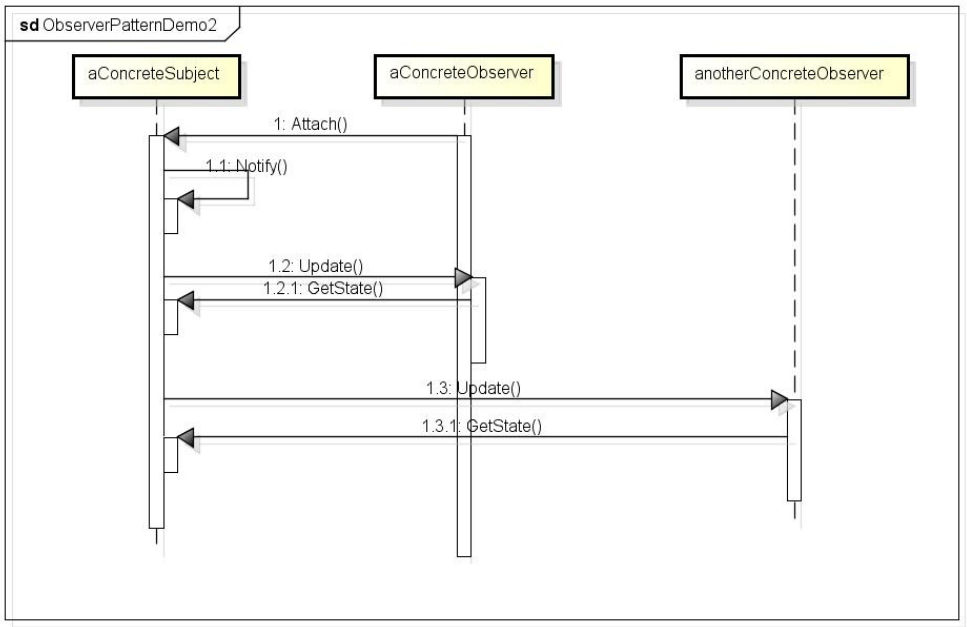
ConcreteObserver（具体观察者）
——维护一个指向ConcreteSubject对象的引用；

——存储有关状态，这些状态应与目标的状态保持一致；
——实现Observer的更新接口以使自身状态与目标的状态保持一致。

观察者模式按照以下方式进行协作：

1. 当ConcreteSubject发生任何可能导致其观察者与其本身状态不一致的改变时，它将通知它的各个观察者；
2. 在得到一个具体目标的改变通知后，ConcreteObserver对象可向目标对象查询信息。ConcreteObserver使用这些信息以使它的状态与目标对象的状态一致。

以下是调用时序图：



适用场合

在以下任一情况下都可以使用观察者模式：

1. 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立的改变和复用；
2. 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变；
3. 当一个对象必须通知其它对象，而它又不能假定其它对象是谁；也就是说，你不希望这些对象是紧密耦合的。

代码实现：

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

class Observer
{
public:
    virtual void Update(int) = 0;
};

class Subject
{
public:
    virtual void Attach(Observer *) = 0;
    virtual void Detach(Observer *) = 0;
    virtual void Notify() = 0;
};

class ConcreteObserver : public Observer
{
public:
    ConcreteObserver(Subject *pSubject) : m_pSubject(pSubject){}

    void Update(int value)
    {
        cout << "ConcreteObserver get the update. New State:" << value << endl;
    }

private:
    Subject *m_pSubject;
```

牛客网学习笔记(8)
数据结构与算法(8)
更多

随笔分类
C# & .NET(40)
C/C++(31)
Java(6)
leetcode刷题(49)
linux之路(15)
NodeJs(1)
php(2)
python(2)
QT(3)
REST(1)
WebService(3)
版本控制工具(3)
笔试&面试&就业(20)
读书笔记&心得(6)
机器学习相关(2)
计算机的基本修养(7)
计算机网络(7)
设计模式（C++/C#篇
数据结构与算法(6)
数据可视化(5)
数据库相关&建模(17)

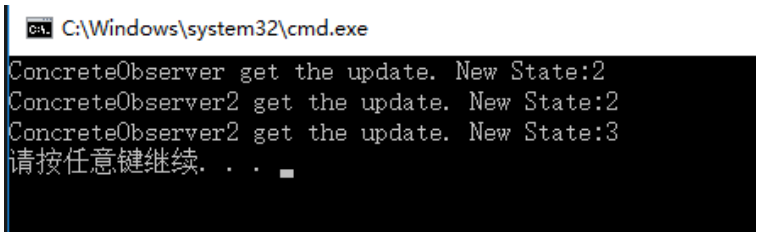
随笔档案

```
};

class ConcreteObserver2 : public Observer
{
public:
    ConcreteObserver2(Subject *pSubject) : m_pSubject(pSubject){}

    void Update(int value)
    {
        cout << "ConcreteObserver2 get the update. New State:" << value << endl;
    }

private:
    Subject *m_pSubject;
};

class ConcreteSubject : public Subject
{
public:
    void Attach(Observer *pObserver);
    void Detach(Observer *pObserver);
    void Notify();

    void SetState(int state)
    {
        m_iState = state;
    }

private:
    std::list<Observer *> m_ObserverList;
    int m_iState;
};

void ConcreteSubject::Attach(Observer *pObserver)
{
    m_ObserverList.push_back(pObserver);
}

void ConcreteSubject::Detach(Observer *pObserver)
{
    m_ObserverList.remove(pObserver);
}

void ConcreteSubject::Notify()
{
    std::list<Observer *>::iterator it = m_ObserverList.begin();
    while (it != m_ObserverList.end())
    {
        (*it)->Update(m_iState);
        ++it;
    }
}

int main()
{
    // Create Subject
    ConcreteSubject *pSubject = new ConcreteSubject();

    // Create Observer
    Observer *pObserver = new ConcreteObserver(pSubject);
    Observer *pObserver2 = new ConcreteObserver2(pSubject);

    // Change the state
    pSubject->SetState(2);

    // Register the observer
    pSubject->Attach(pObserver);
    pSubject->Attach(pObserver2);

    pSubject->Notify();

    // Unregister the observer
    pSubject->Detach(pObserver);

```

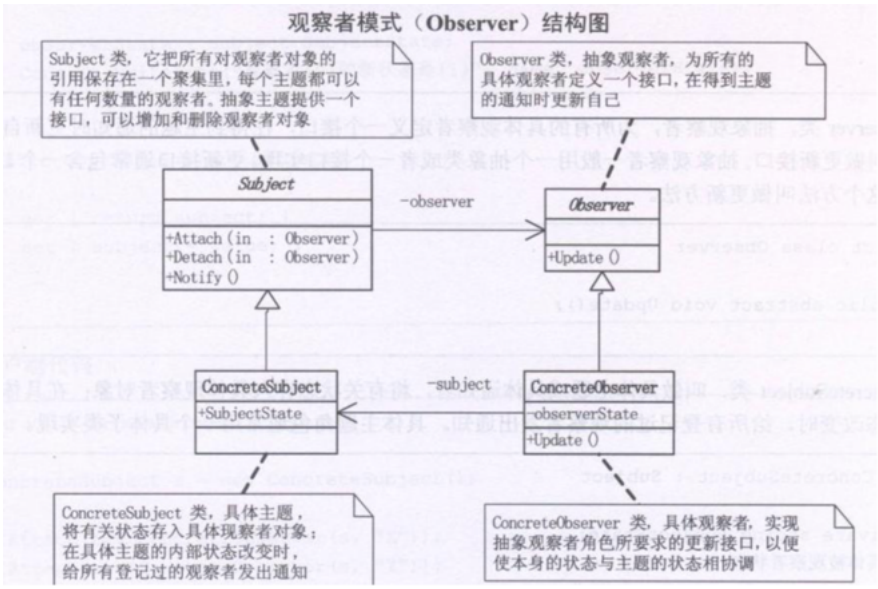
2018年5月 (2)
2018年3月 (2)
2018年2月 (2)
2017年12月 (5)
2017年11月 (6)
2017年10月 (1)
2017年9月 (5)
2017年8月 (16)
2017年7月 (5)
2017年6月 (2)
2017年5月 (8)
2016年10月 (3)
2016年9月 (13)
2016年8月 (11)
2016年6月 (6)
2016年5月 (26)
2016年4月 (10)
2016年3月 (21)
2016年2月 (14)
2016年1月 (14)
2015年12月 (5)
2015年10月 (11)
2015年9月 (3)
2015年8月 (4)
2015年7月 (29)
2015年6月 (23)

```
    pSubject->SetState(3);
    pSubject->Notify();

    delete pObserver;
    delete pObserver2;
    delete pSubject;
}
```

vs2013运行结果：



示例2代码实现：



这里的目标 Subject 提供依赖于它的观察者 Observer 的注册（ Attach ）和注销（ Detach ）操作，并且提供了使得依赖于它的所有观察者同步的操作（ Notify ）。 观察者 Observer 则提供一个 Update 操作， 注意这里的 Observer 的 Update 操作并不在 Observer 改变了 Subject 目标状态的时候就对自己进行更新， 这个更新操作要延迟到 Subject 对象发出 Notify 通知所有Observer 进行修改（调用 Update ）。

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

class Subject;
//抽象观察者
class Observer
{
protected:
    string name;
    Subject *sub;
public:
    Observer(string name, Subject *sub)
    {
        this->name = name;
        this->sub = sub;
    }
    virtual void update() = 0;
};
//具体的观察者，看股票的
class StockObserver :public Observer
{
public:
    StockObserver(string name, Subject *sub) :Observer(name, sub)
```

```
    {
    }
    void update();
};
//具体的观察者，看NBA的
class NBAObserver :public Observer
{
public:
    NBAObserver(string name, Subject *sub) :Observer(name, sub)
    {
    }
    void update();
};
//抽象通知者
class Subject
{
protected:
    list<Observer*> observers;
public:
    string action;
    virtual void attach(Observer*) = 0;
    virtual void detach(Observer*) = 0;
    virtual void notify() = 0;
};
//具体通知者，秘书
class Secretary :public Subject
{
    void attach(Observer *observer)
    {
        observers.push_back(observer);
    }
    void detach(Observer *observer)
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            if ((*iter) == observer)
            {
                observers.erase(iter);
            }
            ++iter;
        }
    }
    void notify()
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            (*iter)->update();
            ++iter;
        }
    }
};

void StockObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭股票，装做很认真工作的样子！" << endl;
    }
}

void NBAObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭NBA，装做很认真工作的样子！" << endl;
    }
}

int main()
{
```

阅读排行榜

1. 使用IntelliJ IDEA开
13)
2. C++中String类的字
8228)
3. UML实践详细经典教
4. 设计模式之观察者模
41)
5. C++中四种类型转换

评论排行榜

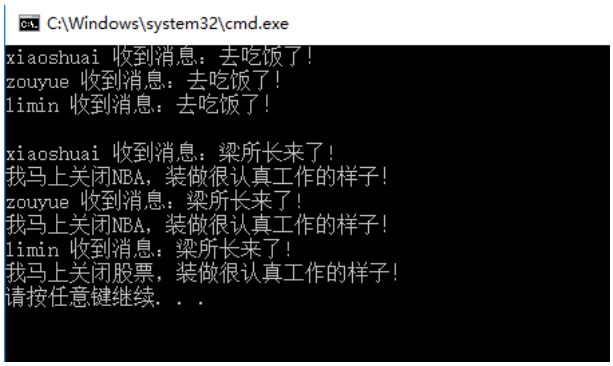
1. 腾讯2017暑期实习生
2. 使用IntelliJ IDEA开
3. UML实践详细经典教
4. leetcode资料整理(2
5. 设计模式之观察者模

推荐排行榜

1. 使用IntelliJ IDEA开
2. UML实践详细经典教
3. C++ main()函数及
4. extern “C”的作用详
5. C++中四种类型转换

```
    Subject *dwq = new Secretary(); //创建观察者<br>       //被观察的对象
    Observer *xs = new NBAObserver("xiaoshuai", dwq);
    Observer *zy = new NBAObserver("zouyue", dwq);
    Observer *lm = new StockObserver("limin", dwq);
       //加入观察队列
    dwq->attach(xs);
    dwq->attach(zy);
    dwq->attach(lm);
       //事件
    dwq->action = "去吃饭了！";<br>       //通知
    dwq->notify();
    cout << endl;
    dwq->action = "梁所长来了!";
    dwq->notify();
    return 0;
}
```

运行结果：



参考文献：

《大话设计模式 C++》

《C++设计模式》

C++设计模式——观察者模式

"I find that the harder i work , the more luck i seem to have ! "

分类： 设计模式（C++/C#篇）

标签： C/C++

#小金乌会发光&
关注 - 23
粉丝 - 32
+加关注

0 0

« 上一篇：mysql理论结合实际篇（一）
» 下一篇：设计模式之工厂模式（c++）

posted @ 2016-08-14 14:56 #小金乌会发光& 阅读(10041) 评论(2) 编辑 收藏

评论列表

#1楼 2017-07-19 11:16 飘雪的冬夜

文中说："这种交互也称为发布-订阅(publish-subscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者。"，而下面又说："——目标知道它的观察者。可以有任意多个观察者观察同一个目标；"是不是前者说错了呢？

支持(0) 反对(0)

#2楼[楼主] 2017-07-19 16:00 #小金乌会发光&

@ 飘雪的冬夜
前者并没有说错，"目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者。"指的是目标不需要知道观察者的身份，和数量，这完全符合发布-订阅模式的理念。"目标知道它的观察者"指的是在subject类中保存了观察者对象的引用，与前者并不矛盾。
这两者字面上看起来好像矛盾，实际上还是需要从内涵和应用上来理解。谢谢你的问题！

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。