观察者模式

题目链接

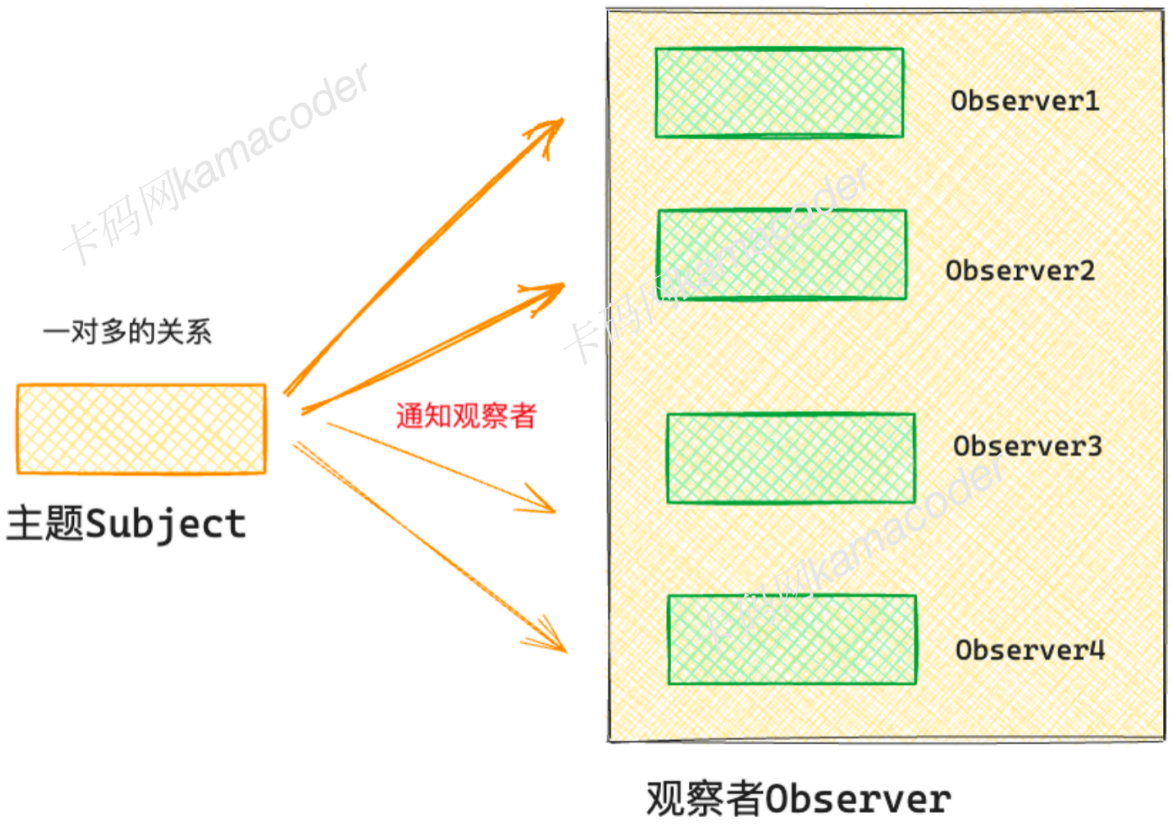
观察者模式-时间观察者

什么是观察者模式

观察者模式（发布-订阅模式）属于行为型模式，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一个主题对象，当主题对象的状态发生变化时，所有依赖于它的观察者都得到通知并被自动更新。

观察者模式依赖两个模块：

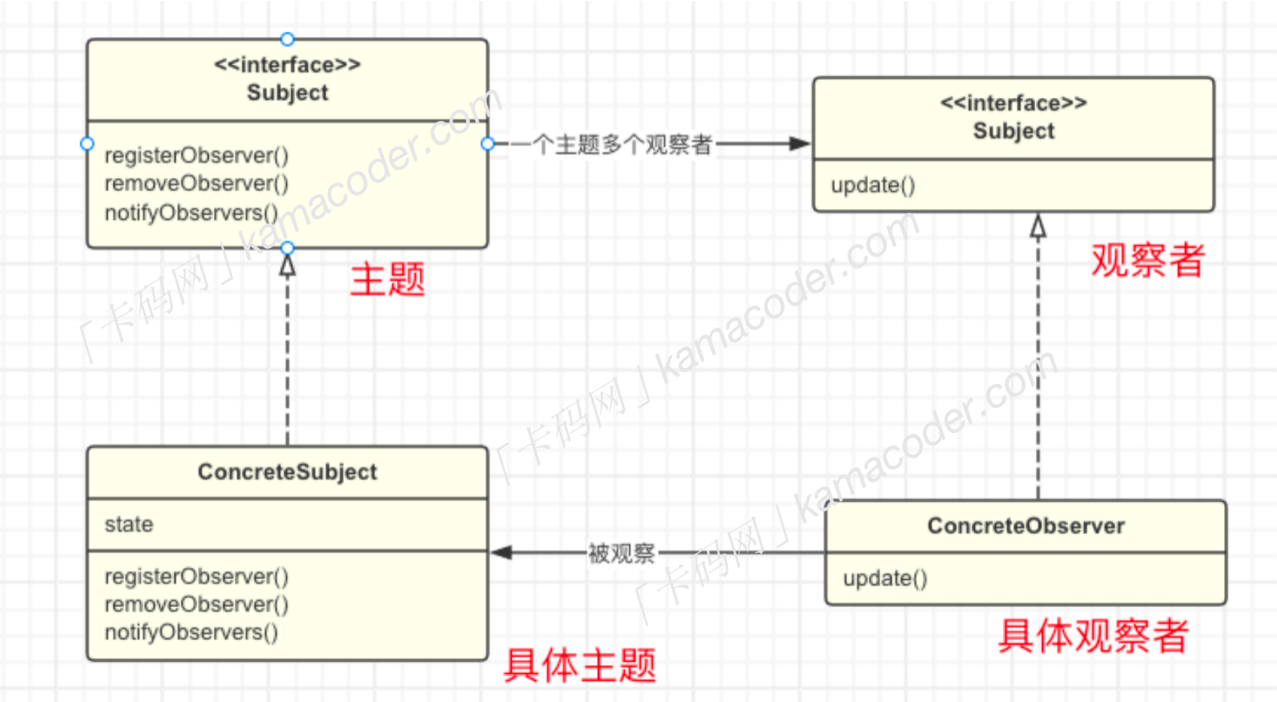
- `Subject`(主题)：也就是被观察的对象，它可以维护一组观察者，当主题本身发生改变时就会通知观察者。
- `Observer`(观察者)：观察主题的对象，当“被观察”的主题发生变化时，观察者就会得到通知并执行相应的处理。



使用观察者模式有很多好处，比如说观察者模式将主题和观察者之间的关系解耦，主题只需要关注自己的状态变化，而观察者只需要关注在主题状态变化时需要执行的操作，两者互不干扰，并且由于观察者和主题是相互独立的，可以轻松的增加和删除观察者，这样实现的系统更容易扩展和维护。

观察者模式的结构

观察者模式依赖主题和观察者，但是一般有4个组成部分：



- 主题Subject，一般会定义成一个接口，提供方法用于**注册、删除和通知观察者**，通常也包含一个状态，当状态发生改变时，通知所有的观察者。
- 观察者Observer：观察者也需要实现一个接口，包含一个更新方法，在接收主题通知时执行对应的操作。
- 具体主题ConcreteSubject：主题的具体实现，维护一个观察者列表，包含了观察者的注册、删除和通知方法。
- 具体观察者ConcreteObserver：观察者接口的具体实现，每个具体观察者都注册到具体主题中，当主题状态变化并通知到具体观察者，具体观察者进行处理。

观察者模式的基本实现

根据上面的类图，我们可以写出观察者模式的基本实现

```
// 主题接口 （主题）
interface Subject {
    // 注册观察者
```

```
    void registerObserver(Observer observer);
    // 移除观察者
    void removeObserver(Observer observer);
    // 通知观察者
    void notifyObservers();
}
// 观察者接口 (观察者)
interface Observer {
    // 更新方法
    void update(String message);
}

// 具体主题实现
class ConcreteSubject implements Subject {
    // 观察者列表
    private List<Observer> observers = new ArrayList<>();
    // 状态
    private String state;

    // 注册观察者
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    // 移除观察者
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    // 通知观察者
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            // 观察者根据传递的信息进行处理
            observer.update(state);
        }
    }
    // 更新状态
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}

// 具体观察者实现
class ConcreteObserver implements Observer {
```

```
    // 更新方法
    @Override
    public void update(String message) {
    }
}
```

什么时候使用观察者模式

观察者模式特别适用于**一个对象的状态变化会影响到其他对象，并且希望这些对象在状态变化时能够自动更新的情况。** 比如说在图形用户界面中，按钮、滑动条等组件的状态变化可能需要通知其他组件更新，这使得观察者模式被广泛应用于GUI框架，比如Java的Swing框架。

此外，观察者模式在前端开发和分布式系统中也有应用，比较典型的例子是前端框架Vue，当数据发生变化时，视图会自动更新。而在分布式系统中，观察者模式可以用于实现节点之间的消息通知机制，节点的状态变化将通知其他相关节点。

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 观察者接口
interface Observer {
    void update(int hour);
}

// 主题接口
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// 具体主题实现
class Clock implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int hour = 0;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
```

```
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(hour);
        }
    }

    public void tick() {
        hour = (hour + 1) % 24; // 模拟时间的推移
        notifyObservers();
    }
}

// 具体观察者实现
class Student implements Observer {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    @Override
    public void update(int hour) {
        System.out.println(name + " " + hour);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取学生数量
        int N = scanner.nextInt();

        // 创建时钟
        Clock clock = new Clock();

        // 注册学生观察者
        for (int i = 0; i < N; i++) {
            String studentName = scanner.next();
            clock.registerObserver(new Student(studentName));
        }
```

```
        // 读取时钟更新次数
        int updates = scanner.nextInt();

        // 模拟时钟每隔一个小时更新一次
        for (int i = 0; i < updates; i++) {
            clock.tick();
        }
    }
}
```

其他语言版本

C++

```
#include <iostream>
#include <vector>
#include <algorithm>

// 观察者接口
class Observer {
public:
    virtual void update(int hour) = 0;
    virtual ~Observer() = default; // 添加虚析构函数
};

// 主题接口
class Subject {
public:
    virtual void registerObserver(Observer* observer) = 0;
    virtual void removeObserver(Observer* observer) = 0;
    virtual void notifyObservers() = 0;
    virtual ~Subject() = default; // 添加虚析构函数
};

// 具体主题实现
class Clock : public Subject {
private:
    std::vector<Observer*> observers;
    int hour;

public:
    Clock() : hour(0) {}

    void registerObserver(Observer* observer) override {
```

```
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) override {
        auto it = std::find(observers.begin(), observers.end(), observer);
        if (it != observers.end()) {
            observers.erase(it);
        }
    }

    void notifyObservers() override {
        for (Observer* observer : observers) {
            observer->update(hour);
        }
    }

    // 添加获取观察者的函数
    const std::vector<Observer*>& getObservers() const {
        return observers;
    }

    void tick() {
        hour = (hour + 1) % 24; // 模拟时间的推移
        notifyObservers();
    }
};

// 具体观察者实现
class Student : public Observer {
private:
    std::string name;

public:
    Student(const std::string& name) : name(name) {}

    void update(int hour) override {
        std::cout << name << " " << hour << std::endl;
    }
};

int main() {
    // 读取学生数量
    int N;
    std::cin >> N;

    // 创建时钟
```

```
    Clock clock;

    // 注册学生观察者
    for (int i = 0; i < N; i++) {
        std::string studentName;
        std::cin >> studentName;
        clock.registerObserver(new Student(studentName));
    }

    // 读取时钟更新次数
    int updates;
    std::cin >> updates;

    // 模拟时钟每隔一个小时更新一次
    for (int i = 0; i < updates; i++) {
        clock.tick();
    }

    // 释放动态分配的观察者对象
    for (Observer* observer : clock.getObservers()) {
        delete observer;
    }

    return 0;
}
```

Python

```
from typing import List

# 观察者接口
class Observer:
    def update(self, hour: int):
        pass

# 主题接口
class Subject:
    def register_observer(self, observer: Observer):
        pass

    def remove_observer(self, observer: Observer):
        pass

    def notify_observers(self):
        pass
```

```python
# 具体主题实现
class Clock(Subject):
    def __init__(self):
        self.observers: List[Observer] = []
        self.hour = 0

    def register_observer(self, observer: Observer):
        self.observers.append(observer)

    def remove_observer(self, observer: Observer):
        self.observers.remove(observer)

    def notify_observers(self):
        for observer in self.observers:
            observer.update(self.hour)

    def tick(self):
        self.hour = (self.hour + 1) % 24  # 模拟时间的推移
        self.notify_observers()

# 具体观察者实现
class Student(Observer):
    def __init__(self, name: str):
        self.name = name

    def update(self, hour: int):
        print(f"{self.name} {hour}")

if __name__ == "__main__":
    # 读取学生数量
    N = int(input())

    # 创建时钟
    clock = Clock()

    # 注册学生观察者
    for _ in range(N):
        student_name = input()
        clock.register_observer(Student(student_name))

    # 读取时钟更新次数
    updates = int(input())

    # 模拟时钟每隔一个小时更新一次
    for _ in range(updates):
        clock.tick()
```

GO

```go
package main

import (
    "fmt"
)

// 观察者接口
type Observer interface {
    Update(hour int)
}

// 主题接口
type Subject interface {
    RegisterObserver(observer Observer)
    RemoveObserver(observer Observer)
    NotifyObservers()
}

// 具体主题实现
type Clock struct {
    observers []Observer
    hour      int
}

func (c *Clock) RegisterObserver(observer Observer) {
    c.observers = append(c.observers, observer)
}

func (c *Clock) RemoveObserver(observer Observer) {
    for i, obs := range c.observers {
        if obs == observer {
            c.observers = append(c.observers[:i], c.observers[i+1:]...)
            break
        }
    }
}

func (c *Clock) NotifyObservers() {
    for _, observer := range c.observers {
        observer.Update(c.hour)
    }
}

func (c *Clock) Tick() {
```

```
    c.hour = (c.hour + 1) % 24 // 模拟时间的推移
    c.NotifyObservers()
}

// 具体观察者实现
type Student struct {
    name string
}

func NewStudent(name string) *Student {
    return &Student{name: name}
}

func (s *Student) Update(hour int) {
    fmt.Println(s.name, hour)
}

func main() {
    // 读取学生数量
    var N int
    fmt.Scan(&N)

    // 创建时钟
    clock := &Clock{}

    // 注册学生观察者
    for i := 0; i < N; i++ {
        var studentName string
        fmt.Scan(&studentName)
        clock.RegisterObserver(NewStudent(studentName))
    }

    // 读取时钟更新次数
    var updates int
    fmt.Scan(&updates)

    // 模拟时钟每隔一个小时更新一次
    for i := 0; i < updates; i++ {
        clock.Tick()
    }
}
```

Typescript

```typescript
interface ISubject {
  add(observer: IObserver): void;
  remove(observer: IObserver): void;
  notify(msg: string): void;
}

interface IObserver {
  update(msg: string): void;
}

class Clock implements ISubject {
  private observers: IObserver[]

  constructor() {
    this.observers = [];
  }

  add(observer: IObserver) {
    this.observers.push(observer);
  }

  remove(observer: IObserver) {
    this.observers = this.observers.filter((item) => item!== observer);
  }

  notify(msg: string) {
    this.observers.forEach((item) => {
      item.update(msg);
    });
  }
}

class Student implements IObserver {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  update(msg: string) {
    console.log(`${this.name} ${msg}`);
  }
}

// @ts-ignore
entry(2, (...args) => {
```

```
  return (time: number) => {
    const timeSubject = new Clock();
    args.forEach((item) => {
      timeSubject.add(new Student(item));
    });

    for (let i = 1; i <= time; i++) {
      timeSubject.notify(i.toString());
    }
  };
})("Alice")("Bob")(3);

function entry(count: number, fn: (...args: any) => void) {
  function dfs(...args) {
    if (args.length < count) {
      return (arg) => dfs(...args, arg);
    }
    return fn(...args);
  }
  return dfs;
}
```