**Design Patterns: Observer Pattern in TypeScript**

Master the Implementation and Application Scenarios of Observer Pattern and Publish-Subscribe Pattern.

Welcome to the **Design Patterns in TypeScript** series, which introduces some useful design patterns in web development using TypeScript.

Previous articles are as follows:

* [Strategy Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-strategy-pattern-in-typescript-54eda9b40f09)
* [Chain of Responsibility Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-chain-of-responsibility-pattern-in-typescript-dba6bdffe456)
* [Observer Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-observer-pattern-in-typescript-f6589f1ce4fc)
* [Template Method Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-template-method-pattern-in-typescript-ce0c8b158985)
* [Adapter Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-adapter-pattern-in-typescript-4b7ad3c1c234)
* [Factory Method Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-factory-method-pattern-in-typescript-c4c3047a6289)
* [Abstract Factory Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-abstract-factory-pattern-in-typescript-84cd7b002964)

Design patterns are very important for web developers and we can write better code by mastering them. In this article, I will use **TypeScript** to introduce the **Observer Pattern and Publish–subscribe Pattern.**

**Observer Pattern**

Observer pattern is widely used in the web world, MutationObserver, IntersectionObserver, PerformanceObserver, ResizeObserver, ReportingObserver. These APIs can all see the observer pattern. In addition, this pattern is also used for event monitoring and data responsiveness (such as when data changes and the page is automatically updated).

Observer pattern defines a **one-to-many** relationship, allowing multiple observer objects to monitor a subject-object at the same time. When the state of the subject-object changes, all observer objects will be notified so that they can be automatically updated.

There are two main roles in the Observer pattern: **Subject and Observer**.

In the above diagram, the Subject is my article, and the observers are chris1993 and bytefish. since the observer pattern supports simple broadcast communication, all observers are automatically notified when a new article is published.

In order to better understand the following code, let’s first look at the corresponding UML class diagram:

In the above figure, we use interface to define the Observer and Subject interfaces respectively, which are used to describe the **Observer** and **Subject** objects.

**Observer interface**

interface Observer {  
 notify(article: Article): void;  
}

**Subject interface**

interface Subject {  
 observers: Observer[];  
 addObserver(observer: Observer): void;  
 deleteObserver(observer: Observer): void;  
 notifyObservers(article: Article): void;  
}

Then, we define the implementation classes of the above interfaces respectively: ConcreteObserver and ConcreteSubject :

**ConcreteObserver class**

class ConcreteObserver implements Observer {  
 constructor(private name: string) {} notify(article: Article) {  
 console.log(`"Article: ${article.title}" has been sent to ${this.name}.`);  
 }  
}

**ConcreteSubject class**

class ConcreteSubject implements Subject{  
 public observers: Observer[] = []; public addObserver(observer: Observer): void {  
 this.observers.push(observer);  
 } public deleteObserver(observer: Observer): void {  
 const n: number = this.observers.indexOf(observer);  
 n != -1 && this.observers.splice(n, 1);  
 } public notifyObservers(article: Article): void {  
 this.observers.forEach((observer) => observer.notify(article));  
 }  
}

Let’s verify the corresponding function:

const subject: Subject = new ConcreteSubject();  
const chris1993 = new ConcreteObserver("Chris1993");  
const bytefish = new ConcreteObserver("Bytefish");subject.addObserver(chris1993);  
subject.addObserver(bytefish);subject.notifyObservers({  
 author: "Bytefer",  
 title: "Observer Pattern in TypeScript",  
 url: "[https://medium.com](https://medium.com/***)/\*\*\*",  
});subject.deleteObserver(bytefish);  
subject.notifyObservers({  
 author: "Bytefer",  
 title: "Adapter Pattern in TypeScript",  
 url: "[https://medium.com](https://javascript.plainenglish.io/***.com)/\*\*\*",  
});

When the above code runs successfully, the terminal will output the following result:

"Article: Observer Pattern in TypeScript" has been sent to Chris1993.  
"Article: Observer Pattern in TypeScript" has been sent to Bytefish.  
"Article: Adapter Pattern in TypeScript" has been sent to Chris1993.

Currently, I write on two main topics, JavaScript and TypeScript, so if I want to publish a new article, only readers interested in JavaScript or TypeScript will be notified. If we use the Observer pattern, we need to create two different Subjects, and we can also use the **Publish–subscribe Pattern**.

**Publish–subscribe Pattern**

In software architecture, publish-subscribe is a messaging paradigm where senders of messages (called publishers) do not send messages directly to specific receivers (called subscribers). Instead, the published messages are grouped into different categories and sent to different subscribers. Likewise, subscribers can express interest in one or more categories and only receive messages of interest without knowing which publishers exist.

There are three main roles in the publish-subscribe model: Publishers, Channels, and Subscribers.

In the above figure, Publisher is Bytefer, Topic A and Topic B in Channels correspond to JavaScript topic and TypeScript topic respectively, and Subscriber is chris1993, bytefish, etc.

Let’s implement an EventEmitter based on the publish-subscribe pattern:

[Publish-subscribe pattern (github.com)](https://gist.github.com/semlinker/b63d748636ef27a0d57dc6c4c251f720)

<https://gist.github.com/semlinker/b63d748636ef27a0d57dc6c4c251f720>

type EventHandler = (...args: any[]) => any;

class EventEmitter {

private c = new Map<string, EventHandler[]>();

subscribe(topic: string, ...handlers: EventHandler[]) {

let topics = this.c.get(topic);

if (!topics) {

this.c.set(topic, (topics = []));

}

topics.push(...handlers);

}

unsubscribe(topic: string, handler?: EventHandler): boolean {

if (!handler) {

return this.c.delete(topic);

}

const topics = this.c.get(topic);

if (!topics) {

return false;

}

const index = topics.indexOf(handler);

if (index < 0) {

return false;

}

topics.splice(index, 1);

if (topics.length === 0) {

this.c.delete(topic);

}

return true;

}

publish(topic: string, ...args: any[]): any[] | null {

const topics = this.c.get(topic);

if (!topics) {

return null;

}

return topics.map((handler) => {

try {

return handler(...args);

} catch (e) {

console.error(e);

return null;

}

});

}

}

After defining the EventEmitter class, we can use it like this:

const eventEmitter = new EventEmitter();eventEmitter.subscribe("ts",   
 (msg) => console.log(`Received：${msg}`));eventEmitter.publish("ts", "Observer pattern");  
eventEmitter.unsubscribe("ts");  
eventEmitter.publish("ts", "Pub-Sub pattern");

When the above code runs successfully, the terminal will output the following result: Received: Observer pattern.

In an event-driven architecture, the publish-subscribe pattern plays an important role. The specific implementation of this pattern can be used as an event bus to realize message communication between different components or modules in the same system. For the widely used plug-in architecture, it can be used to implement message communication between different plug-ins.

After reading this article, I hope you have a certain understanding of the observer pattern and the publish-subscribe pattern. If you have any questions, please feel free to leave me a message. I will continue to introduce other patterns later, if you are interested, you can follow me on [Medium](https://medium.com/@bytefer) or [Twitter](https://twitter.com/Tbytefer).