发布-订阅模式

1、定义

发布-订阅模式：发布—订阅模式又叫观察者模式，它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将得到通知。

2、作用

- 发布—订阅模式可以广泛应用于异步编程中， 这是一种替代传递回调函数的方案。无需过多关注对象在异步运行期间的内部状态，而只需要订阅感兴趣的事件发生点。
- 发布—订阅模式可以取代对象之间硬编码的通知机制，一个对象不用再显式地调用另外一个对象的某个接口。发布—订阅模式让两个对象松耦合地联系在一起，虽然不太清楚彼此的细节，但这不影响它们之间相互通信。

3、DOM事件（最常用的订阅发布模式）

只要我们曾经在 DOM 节点上面绑定过事件函数，那我们就曾经使用过发布—订阅模式，来看看下面这两句简单的代码发生了什么事情：

```
// 监听全局点击事件，（订阅全局点击事件）
document.body.addEventListener('click', function() {
    console.log('触发点击事件')
}, false)

document.body.click(); // 模拟用户点击 （发布点击事件）
```

4、自定义事件

需求介绍： 小明最近看上了一套房子，到了售楼处之后才被告知，该楼盘的房子早已售罄。好在售楼MM 告诉小明，不久后还有一些尾盘推出，开发商正在办理相关手续，手续办好后便可以购买。但到底是什么时候，目前还没有人能够知道。

于是小明记下了售楼处的电话，以后每天都会打电话过去询问是不是已经到了购买时间。除了小明，还有小红、小强、小龙也会每天向售楼处咨询这个问题。一个星期过后，售楼 MM 决定辞职，因为厌倦了每天回答 1000 个相同内容的电话。

当然现实中没有这么笨的销售公司，实际上故事是这样的：小明离开之前，把电话号码留在了售楼处。售楼 MM 答应他，新楼盘一推出就马上发信息通知小明。小红、小强和小龙也是一样，他们的电话号码都被记在售楼处的花名册上，新楼盘推出的时候，售楼 MM 会翻开花名册，遍历上面的电话号码，依次发送一条短信来通知他们。

实现发布—订阅模式思路：

- 首先要指定好谁充当发布者 （比如售楼处）
- 然后给发布者添加一个缓存列表，用于存放回调函数以便通知订阅者（售楼处的花名册）；

- 接着往回调函数里填入一些参数，订阅者可以接收这些参数 （短信里加上房子的单价、面积、容积率）
- 最后发布消息的时候，发布者会遍历这个缓存列表，依次触发里面存放的订阅者回调函数（遍历花名册，挨个发短信）。

代码如下：

```
// 定义售楼处
let event = {
    clientList: [], // 缓存列表，存放订阅者的回调函数
    subscribe: function() { //订阅事件
        this.clientList.push(fn);
    },
    publish: function() {
        for(var i = 0,fn; fn = this.clientList[i++];) {
            fn.apply(this, arguments); // arguments 是发布消息时带上的参数
        }
    }
}

// 下面简单测试

event.subscribe(function(price, squareMeter) {
    console.log('价格=' + price);
     console.log('平方米=' + squareMeter);
})

event.publish(200000, 88); // 输出： 200万，88平方米
event.trigger(3000000, 110); // 输出：300 万，110 平方米
```

至此，我们已经实现了一个最简单的发布—订阅模式，但这里还存在一些问题。我们看到订阅者接收到了发布者发布的每个消息，虽然小明只想买 88 平方米的房子，但是发布者把 110 平 方米的信息也推送给了小明，这对小明来说是不必要的困扰。所以我们有必要增加一个标示 key，让订阅者只订阅自己感兴趣的消息。改写后的代码如下：

```
let event = {
    clientList: [], // 缓存列表，存放订阅者的回调函数
    subscribe: function() { //订阅事件
      if ( !this.clientList[ key ] ){ // 如果还没有订阅过此类消息，给该类消息创建一个缓存列表
            this.clientList[ key ] = [];
        }
        this.clientList[ key ].push( fn ); // 订阅的消息添加进消息缓存列表
    },
    publish: function() {
       var key = Array.prototype.shift.call( arguments ), // 取出消息类型
        fns = this.clientList[ key ]; // 取出该消息对应的回调函数集合
        if ( !fns || fns.length === 0 ){ // 如果没有订阅该消息，则返回
            return false;
        }
```

```
            for( var i = 0, fn; fn = fns[ i++ ]; ){
                fn.apply( this, arguments ); // (2) // arguments 是发布消息时附送的参数
            }
        }
    }

    // 测试

    // 订阅事件
    event.subscribe('test0', function(data) {
            console.log('数据0: ' + data);
    });
    event.subscribe('test0', function(data) {
            console.log('数据1: ' + data);
    });
    event.subscribe('test1', function(data) {
            console.log(data);
    });

    setTimeout(function() {
        count++
        // 发布事件
        event.publish('test0', count)
        event.publish('test1', count)
    }, 1000)
```

5、取消订阅事件

有时候，我们也许需要取消订阅事件的功能。比如小明突然不想买房子了，为了避免继续接收到售楼处推送过来的短信，小明需要取消之前订阅的事件。现在我们给 event 对象增加unsubscribe方法：

```
    event.unsubscribe = function(key, fn) {
            const t = this.clientList[key];
        if (!t) { // 如果 key 对应的消息没有被人订阅，则直接返回
            return false;
        }
        if (!fn) {
            // 如果不指定处理方法，则取消该事件下所有的处理方法
            delete this.clientList[key];
            return true;
        }
        // 找到指定取消的处理方法的位置
        const i = t.indexOf(fn);
        if (i < 0) {
            return false;
        }
        t.splice(i, 1);
        // 如果事件下的处理方法为空则删除该事件
        if (!t.length) {
            delete this.clientList[key];
        }
```

```
    return true;
}
```

6、全局发布订阅事件（加强全局思想）

在程序中，发布—订阅模式可以用一个全局的 Event 对象来实现，订阅者不需要了解消息来自哪个发布者，发布者也不知道消息会推送给哪些订阅者，Event 作为一个类似"中介者"的角色，把订阅者和发布者联系起来。见如下代码：

```
var Event = (function(){
    var clientList = [],
        subscribe,
        publish,
        unsubscribe;
    subscribe = function() { //订阅事件
        if ( !this.clientList[ key ] ){ // 如果还没有订阅过此类消息，给该类消息创建一个缓存列表
                this.clientList[ key ] = [];
        }
        this.clientList[ key ].push( fn ); // 订阅的消息添加进消息缓存列表
    };
    publish = function() {
        var key = Array.prototype.shift.call( arguments ), // 取出消息类型
        fns = this.clientList[ key ]; // 取出该消息对应的回调函数集合
        if ( !fns || fns.length === 0 ){ // 如果没有订阅该消息，则返回
            return false;
        }
        for( var i = 0, fn; fn = fns[ i++ ]; ){
            fn.apply( this, arguments ); // (2) // arguments 是发布消息时附送的参数
        }
    };
    unsubscribe = function(key, fn) {
        const t = this.clientList[key];
        if (!t) { // 如果 key 对应的消息没有被人订阅，则直接返回
            return false;
        }
        if (!fn) {
            // 如果不指定处理方法，则取消该事件下所有的处理方法
            delete this.clientList[key];
            return true;
        }
        // 找到指定取消的处理方法的位置
        const i = t.indexOf(fn);
        if (i < 0) {
            return false;
        }
        t.splice(i, 1);
        // 如果事件下的处理方法为空则删除该事件
        if (!t.length) {
            delete this.clientList[key];
        }
```

```
        return true;
    };
    return {
        subscribe: listen,
        publish: trigger,
        unsubscribe: remove
    }

})()
```

7、利用class类实现订阅发布

第一次接触发布-订阅模式是前组长自定义全局Angular事件服务，用来在全应用中通过发布/订阅事件来进行通信，也曾阅读学习，受益良多，特此致谢！

代码如下： 源码链接

```
import { Injectable } from '@angular/core';

/**
 * @name EventsService
 * @description 自定义全局事件服务，用来在全应用中通过发布/订阅事件来进行通信
 * @usage
 * ```ts
 * import { EventsService } from '../services/events.service';
 *
 * constructor(public events: EventsService) {}
 *
 * // 订阅事件并打印信息
 * this.events.subscribe('test', (data: any) => {
 *              console.log(data); // '我是test事件发送来的信息！'
 * });
 *
 * // 发布事件
 * this.events.publish('test', '我是test事件发送来的信息！');
 *
 * // 取消订阅
 * this.events.unsubscribe('test');
 * ```
 */
@Injectable()
export class EventsService {
        private channels: any = [];

        /**
         * 通过事件主题订阅相应事件
         * @param {string} topic 订阅事件的主题
         * @param {function[]} handlers 事件处理方法
         */
        subscribe(topic: string, ...handlers: Function[]): void {
                if (!this.channels[topic]) {
```

```
                        this.channels[topic] = [];
                }
                handlers.forEach((handler) => {
                        this.channels[topic].push(handler);
                });
        }

        /**
         * 通过事件主题取消订阅相应事件
         * @param {string} topic 取消订阅事件的主题
         * @param {function} handler 指定取消该事件主下的处理方法
         * @returns {boolean} 取消成功返回true
         */
        unsubscribe(topic: string, handler: Function = null): boolean {
                const t = this.channels[topic];
                if (!t) {
                        return false;
                }
                if (!handler) {
                        // 如果不指定处理方法，则取消该事件下所有的处理方法
                        delete this.channels[topic];
                        return true;
                }
                // 找到指定取消的处理方法的位置
                const i = t.indexOf(handler);
                if (i < 0) {
                        return false;
                }
                t.splice(i, 1);
                // 如果事件下的处理方法为空则删除该事件
                if (!t.length) {
                        delete this.channels[topic];
                }
                return true;
        }

        /**
         * 通过事件主题发布相应事件
         * @param {string} topic 发布事件的主题
         * @param {any[]} args 通过事件发送的数据
         * @returns {any[]}
         */
        publish(topic: string, ...args: any[]): any[] {
                const t = this.channels[topic];
                if (!t) {
                        return null;
                }
                const responses: any[] = [];
                t.forEach((handler: any) => {
                        responses.push(handler(...args));
                });
                return responses;
        }
}
```

8、必须先订阅再发布吗（记录于此，加强一下辨证思维）

我们所了解到的发布—订阅模式，都是订阅者必须先订阅一个消息，随后才能接收到发布者发布的消息。如果把顺序反过来，发布者先发布一条消息，而在此之前并没有对象来订阅它，这条消息无疑将消失在宇宙中。

在某些情况下，我们需要先将这条消息保存下来，等到有对象来订阅它的时候，再重新把消息发布给订阅者。就如同 QQ 中的离线消息一样，离线消息被保存在服务器中，接收人下次登录上线之后，可以重新收到这条消息。

这种需求在实际项目中是存在的，比如在之前的商城网站中，获取到用户信息之后才能渲染用户导航模块，而获取用户信息的操作是一个 ajax 异步请求。当 ajax 请求成功返回之后会发布一个事件，在此之前订阅了此事件的用户导航模块可以接收到这些用户信息。

但是这只是理想的状况，因为异步的原因，我们不能保证 ajax 请求返回的时间，有时候它返回得比较快，而此时用户导航模块的代码还没有加载好（还没有订阅相应事件），特别是在用了一些模块化惰性加载的技术后，这是很可能发生的事情。也许我们还需要一个方案，使得我们的发布—订阅对象拥有先发布后订阅的能力。

为了满足这个需求，我们要建立一个存放离线事件的堆栈，当事件发布的时候，如果此时还没有订阅者来订阅这个事件，我们暂时把发布事件的动作包裹在一个函数里，这些包装函数将被存入堆栈中，等到终于有对象来订阅此事件的时候，我们将遍历堆栈并且依次执行这些包装函数，也就是重新发布里面的事件。当然离线事件的生命周期只有一次，就像 QQ 的未读消息只会被重新阅读一次，所以刚才的操作我们只能进行一次。

9、小结

发布—订阅模式的优点非常明显，一为时间上的解耦，二为对象之间的解耦。它的应用非常广泛，既可以用在异步编程中，也可以帮助我们完成更松耦合的代码编写。发布—订阅模式还可以用来帮助实现一些别的设计模式，比如中介者模式。从架构上来看，无论是 MVC 还是 MVVM，都少不了发布—订阅模式的参与，而且 JavaScript 本身也是一门基于事件驱动的语言。

当然，发布—订阅模式也不是完全没有缺点。创建订阅者本身要消耗一定的时间和内存，而且当你订阅一个消息后，也许此消息最后都未发生，但这个订阅者会始终存在于内存中。另外，发布—订阅模式虽然可以弱化对象之间的联系，但如果度使用的话，对象和对象之间的必要联系也将被深埋在背后，会导致程序难以跟踪维护和理解。特别是有多个发布者和订阅者嵌套到一起的时候，要跟踪一个 bug 不是件轻松的事情。