8 发布-订阅模式

发布订阅模式又叫做观察者模式，用于定义对象之间的一对多依赖关系，再JS开发中我们一般用事件模型来代替传统的发布-订阅模式

现实中的发布-订阅模式

小明看房--房屋售罄--售楼人员记录小明信息--有房源之后通知小明（以及其他所有记录人员）

模式作用

- 购房者不用天天打电话通知销售人员，合适的人间点售楼处会作为消息的发布者将消息传递给所有订阅者
  - 该模式可以应用于异步编程之中，替代传统的回调函数方式
  - 例如requestAnimation'Frame就可以在每一帧收到通知做一些事情，只需要订阅感兴趣的事件即可
- 两者不再强耦合，购房者不需要关心售楼处或者销售人员的变更情况，只需要关心短信即可。售楼处也是如此
  - 取代对象之间硬编码的通知机制，不需要显式调用另一个对象的接口，松耦合关系

DOM事件

实际上只要我们在DOM节点上绑定过事件函数，那么我们就使用过发布-订阅模式

```
document.body.addEventListener('click', function(){
    alert(2)
}, false);

document.body.click();

// 我们还可以随意增加订阅者，不会影响发布者的代码
document.body.addEventListener('click', function(){
    alert(3)
}, false);
```

发布-订阅模式的通用实现

这里我们希望实现一个发布-订阅的通用实现：

- 让所有对象都可以拥有发布-订阅能力，不绑定任何一个售楼处对象
- 订阅者可以指订阅自己感兴趣的事件，不想收到其他杂信息

这里我们可以把发布-订阅的功能模块单独抽取出来，放在一个单独的对象中：

```
/**
 * 通用实现
 */
let commonEvent = {
    clientList: [],
    // 监听
    listen: (key, fn) => {
        if(!this.clientList[key]){
            this.clientList[key] = [];
        }
        this.clientList[key].push(fn); // 订阅的消息添加进缓存列表
    },
    // 触发广播
    trggier: () => {
        let key = Array.prototype.shift.call(arguments);
        let fns = this.clientList[key];
        if(!fns || fns.length === 0){
            return false;
        }
        for(let i = 0; i < fns.length; i++){
            const fn = fns[i];
            fn.apply(this, arguments);
        }
    },
    // 取消订阅
    remove: (key, fn) => {
        const fns = this.clientList[key];

        if(!fns){
            return false;
        }

        if(!fn){
            // 没有传入取消订阅的函数，默认取消所有
            fns && fns.length = 0;
        } else {
            for(let l = fns.length - 1; l >= 0; l--){
                const _fn = fns[l];
                if(_fn === fn){
```

```
                    // 删除此回调函数
                    fns.splice(l, 1);
                }
            }
        }
    }
}

// 配置一个install函数让对象可以动态安装发布-订阅功能
const installEvent = (obj) => {
    for(let i in obj){
        obj[i] = commonEvent[i];
    }
}

const salesOffices = {};
installEvent(salesOffices);

const callback = (price) => {
    console.log('价格='+price);
}

salesOffices.listen('square88', callback)

salesOffices.trggier('square88', 10000); // 输出10000

salesOffices.remove('square88', callback);

```

真实的例子-网站登录

这里我们再来看一个真实的例子来更好的理解这个模式带来的好处，假设我们在开发一个网站，有一个login功能，同时有其他许多功能模块，例如header，nav导航，购物车等等，这些模块需要先通过ajax获取用户的登录信息之后才能正常工作，例如获取用户名渲染header

这里异步问题是可以通过回调函数解决，但是更需要关注的是如何实现依赖的功能模块的逻辑和渲染，有一种方式很直观：

```
// 直接依赖具体实现来完成
login.success((data) => {
    header.setAvatar(data.avatar);
    navigator.setAvatar(data.avatar);
    message.refresh();
    // 新的地址模块也需要关注登录成功与否 -> 需要修改代码
    address.refresh();
```

```
})

```

但是我们发现这里的耦合很严重，非常僵硬。

我们通过重构之后，可以让感兴趣的业务模块自行订阅登录的消息，而登录模块只负责发布登录成功的消息即可，这样就不需要关心业务方具体要做什么。

将登录逻辑和业务逻辑解耦开来了：

```
// 重构之后
$.ajax('http://xxx.com?login', (data) => {
    login.trigger('loginSuccess', data);
})

const header = (function(){
    login.listen('loginSuccess', (data) => {
        header.setAvatar(data.avatar);
    })
    return {
        setAvatar: function(data){
            console.log('设置header模块的头像')
        }
    }
})();

const nav = (function(){
    login.listen('loginSuccess', (data) => {
        nav.setAvatar(data.avatar);
    })
    return {
        setAvatar: function(data){
            console.log('设置nav模块的头像')
        }
    }
})();
```

新的业务需要添加相关逻辑时候也不再需要登录模块的开发者介入了！

全局的发布-订阅对象

回到之前售楼处的实现模式，还有两个小问题：

- 我们给每一个发布者都install了listen和trigger与一个缓存列表，其实有点浪费

- 小明和售楼处还是存在一点耦合，小明必须知道售楼处对象的名字是salesOffices才能订阅
  - 如果小明关心另一个300的房子，卖家是salesOffices2，那么小明必须再次执行salesOffices2.listen才行
  - 现实中我们通常去找一家中介，委托中介帮我们找房子，然后我们就不需要关心消息具体来自哪里了，而售楼处也只需要和中介打交道

这里我们实现一个全局的Event对象作为真正的订阅者和发布者的联系人，然后小明和售楼处就只需要通过Event进行通讯即可：

```
/**
 * 全局Event
 */

let Event = (function(){
    var clientList = {},listen,trigger,remove;

    listen = (key, fn) => {
        if(!this.clientList[key]){
            this.clientList[key] = [];
        }
        this.clientList[key].push(fn); // 订阅的消息添加进缓存列表
    };

    // 触发广播
    trggier = () => {
        let key = Array.prototype.shift.call(arguments);
        let fns = this.clientList[key];
        if(!fns || fns.length === 0){
            return false;
        }
        for(let i = 0; i < fns.length; i++){
            const fn = fns[i];
            fn.apply(this, arguments);
        }
    };
    // 取消订阅
    remove = (key, fn) => {
        const fns = this.clientList[key];

        if(!fns){
            return false;
        }

```

```
        if(!fn){
            // 没有传入取消订阅的函数，默认取消所有
            fns && (fns.length = 0);
        } else {
            for(let l = fns.length - 1; l >= 0; l--){
                const _fn = fns[l];
                if(_fn === fn){
                    // 删除此回调函数
                    fns.splice(l, 1);
                }
            }
        }
    };

    return {
        listen,
        trigger,
        remove
    }
})();

Event.listen('squareMeter88', function(price){
    // 小红订阅消息
    console.log('价格=' + price);
}

Event.trigger('squareMeter88', 200000); // 售楼处发布消息
```

模块间通信

上一节实现的是一个全局的Event对象，通过它我们可以在两个封装良好的模块之间通信并且无需知道对方的存在，再来看一个a和b模块保持封装性的同时完成通信

```
/**
 * 模块通信
 */
var a = (function(){
    var count = 0;
    var button = document.getElementById('count');
    button.onclick = function(){
        Event.trggier('add',count++);
    }
})()

```

```
var b = (function(){
    var div = document.getElementById('show');
    Event.listen('add',function(count){
        div.innerHTML = count;
    });
})()
```

有一个问题必须留意：如果全局用了太多的发布-订阅模式来通信，那么模块与模块之间真实的联系其实被隐藏了，导致维护和调试会带来些麻烦，消息的流向不是那么直观了

必须先订阅再发布吗?

某些情况下有可能消息的到来比订阅事件要澡，如同QQ的离线消息一样，它被保存在服务器中，接收人下次登录上线之后会重新收到这些消息。这种需求场景是实际存在的。

因为ajax异步原因，不能保证返回的事件，如果消息返回的比较快，界面还使用了懒加载，很有可能此时模块代码还没有加载好 - 还没有订阅事件。

这里我们可以尝试建立一个存放离线事件的堆栈，如果没来得及订阅，我们可以先把发布事件的动作包裹在一个函数里，这些函数将被存入堆栈，然后等对象加载好启动订阅之后我们可以遍历堆栈依次执行这些包装函数，也就是重新发布里面的事件。这些离线事件的生命周期必须只有一次!

全局事件的命名冲突

有的时候作为全局对象，只有clientList难免会出现事件名冲突情况，这时候我们可以给Event对象扩展命名空间功能避免冲突

JS实现发布-订阅模式的便利性

Java中实现此模式需要把订阅者对象自身当成引用传入发布者对象中，同时订阅者自己需要提供update方法供发布者合适时机调用。

而在JS中我们通过注册回调函数方式来代替传统的发布-订阅模式，更加优雅和简单。

另外在JS中我们无需选择推模型或拉模型

- 推：事件发生时，发布者一次性将所有状态和数据推送给订阅者
    - JS中的arguments可以直接表示参数列表，一般都会选择推模式

- 拉：发布者仅仅通知订阅者事件发生了，然后订阅者通过发布者提供的一些公共接口去拉取数据
  - 发布者变成了一个门户大开的对象，但是订阅者可以按需获取

小结

优点和明显：

- 时间上的解耦
- 对象间的解耦

应用场景：

- 异步编程
- 帮助实现中介者等设计模式
- 帮助实现MVC与MVVM

缺点

- 额外消耗的时间和内存
- 过度使用难以维护和调试，不容易跟踪bug