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！