第 8 章

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。在 JavaScript 开发中，我们一般用事件模型来替代传统的发布–订阅模式。

8.1 现实中的发布–订阅模式

不论是在程序世界里还是现实生活中，发布–订阅模式的应用都非常之广泛。我们先看一个现实中的例子。

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。

8.2 发布–订阅模式的作用

在刚刚的例子中，发送短信通知就是一个典型的发布–订阅模式，小明、小红等购买者都是订阅者，他们订阅了房子开售的消息。售楼处作为发布者，会在合适的时候遍历花名册上的电话号码，依次给购房者发布消息。

可以发现，在这个例子中使用发布-订阅模式有着显而易见的优点。

- 购房者不用再天天给售楼处打电话咨询开售时间，在合适的时间点，售楼处作为发布者会通知这些消息订阅者。
- 购房者和售楼处之间不再强耦合在一起，当有新的购房者出现时，他只需把手机号码留在售楼处，售楼处不关心购房者的任何情况，不管购房者是男是女还是一只猴子。 而售楼处的任何变动也不会影响购买者，比如售楼 MM 离职，售楼处从一楼搬到二楼，这些改变都跟购房者无关，只要售楼处记得发短信这件事情。

第一点说明发布-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。比如，我们可以订阅 ajax 请求的 error、succ 等事件。或者如果想在动画的每一帧完成之后做一些事情，那我们可以订阅一个事件，然后在动画的每一帧完成之后发布这个事件。在异步编程中使用发布-订阅模式，我们就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。

第二点说明发布-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。当有新的订阅者出现时，发布者的代码不需要任何修改；同样发布者需要改变时，也不会影响到之前的订阅者。只要之前约定的事件名没有变化，就可以自由地改变它们。

8.3 DOM 事件

实际上，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：

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

document.body.click();    // 模拟用户点击
```

在这里需要监控用户点击 document.body 的动作，但是我们没办法预知用户将在什么时候点击。所以我们订阅 document.body 上的 click 事件，当 body 节点被点击时，body 节点便会向订阅者发布这个消息。这很像购房的例子，购房者不知道房子什么时候开售，于是他在订阅消息后等待售楼处发布消息。

当然我们还可以随意增加或者删除订阅者，增加任何订阅者都不会影响发布者代码的编写：

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

document.body.addEventListener( 'click', function(){
```

```
    alert(3);
}, false );

document.body.addEventListener( 'click', function(){
    alert(4);
}, false );

document.body.click();    // 模拟用户点击
```

注意，手动触发事件更好的做法是 IE 下用 fireEvent，标准浏览器下用 dispatchEvent 实现。

8.4　自定义事件

除了 DOM 事件，我们还会经常实现一些自定义的事件，这种依靠自定义事件完成的发布–订阅模式可以用于任何 JavaScript 代码中。

现在看看如何一步步实现发布–订阅模式。

- 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）；
- 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）。

另外，我们还可以往回调函数里填入一些参数，订阅者可以接收这些参数。这是很有必要的，比如售楼处可以在发给订阅者的短信里加上房子的单价、面积、容积率等信息，订阅者接收到这些信息之后可以进行各自的处理：

```
var salesOffices = {};    // 定义售楼处

salesOffices.clientList = [];    // 缓存列表，存放订阅者的回调函数

salesOffices.listen = function( fn ){        // 增加订阅者
    this.clientList.push( fn );    // 订阅的消息添加进缓存列表
};

salesOffices.trigger = function(){    // 发布消息
    for( var i = 0, fn; fn = this.clientList[ i++ ]; ){
        fn.apply( this, arguments );    // (2) // arguments 是发布消息时带上的参数
    }
};
```

下面我们来进行一些简单的测试：

```
salesOffices.listen( function( price, squareMeter ){    // 小明订阅消息
    console.log( '价格= ' + price );
    console.log( 'squareMeter= ' + squareMeter );
});

salesOffices.listen( function( price, squareMeter ){     // 小红订阅消息
    console.log( '价格= ' + price );
```

```
    console.log( 'squareMeter= ' + squareMeter );
});

salesOffices.trigger( 2000000, 88 );    // 输出：200 万，88 平方米
salesOffices.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：

```
var salesOffices = {};    // 定义售楼处

salesOffices.clientList = {};    // 缓存列表，存放订阅者的回调函数

salesOffices.listen = function( key, fn ){
    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
};

salesOffices.trigger = function(){    // 发布消息
    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
    }
};

salesOffices.listen( 'squareMeter88', function( price ){    // 小明订阅 88 平方米房子的消息
    console.log( '价格= ' + price );    // 输出： 2000000
});

salesOffices.listen( 'squareMeter110', function( price ){     // 小红订阅 110 平方米房子的消息
    console.log( '价格= ' + price );    // 输出： 3000000
});

salesOffices.trigger( 'squareMeter88', 2000000 );     // 发布 88 平方米房子的价格
salesOffices.trigger( 'squareMeter110', 3000000 );    // 发布 110 平方米房子的价格
```

很明显，现在订阅者可以只订阅自己感兴趣的事件了。

8.5 发布－订阅模式的通用实现

现在我们已经看到了如何让售楼处拥有接受订阅和发布事件的功能。假设现在小明又去另一

个售楼处买房子，那么这段代码是否必须在另一个售楼处对象上重写一次呢，有没有办法可以让所有对象都拥有发布–订阅功能呢？

答案显然是有的，JavaScript 作为一门解释执行的语言，给对象动态添加职责是理所当然的事情。

所以我们把发布–订阅的功能提取出来，放在一个单独的对象内：

```
var event = {
    clientList: [],
    listen: function( 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
    },
    trigger: function(){
        var key = Array.prototype.shift.call( arguments ),    // (1);
            fns = this.clientList[ key ];

        if ( !fns || fns.length === 0 ){    // 如果没有绑定对应的消息
            return false;
        }

        for( var i = 0, fn; fn = fns[ i++ ]; ){
            fn.apply( this, arguments );    // (2) // arguments 是 trigger 时带上的参数
        }
    }
};
```

再定义一个 installEvent 函数，这个函数可以给所有的对象都动态安装发布–订阅功能：

```
var installEvent = function( obj ){
    for ( var i in event ){
        obj[ i ] = event[ i ];
    }
};
```

再来测试一番，我们给售楼处对象 salesOffices 动态增加发布–订阅功能：

```
var salesOffices = {};
installEvent( salesOffices );

salesOffices.listen( 'squareMeter88', function( price ){    // 小明订阅消息
    console.log( '价格= ' + price );
});

salesOffices.listen( 'squareMeter100', function( price ){     // 小红订阅消息
    console.log( '价格= ' + price );
});

salesOffices.trigger( 'squareMeter88', 2000000 );    // 输出：2000000
salesOffices.trigger( 'squareMeter100', 3000000 );    // 输出：3000000
```

8.6 取消订阅的事件

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 remove 方法：

```
event.remove = function( key, fn ){
    var fns = this.clientList[ key ];

    if ( !fns ){    // 如果 key 对应的消息没有被人订阅，则直接返回
        return false;
    }
    if ( !fn ){    // 如果没有传入具体的回调函数，表示需要取消 key 对应消息的所有订阅
        fns && ( fns.length = 0 );
    }else{
        for ( var l = fns.length - 1; l >=0; l-- ){    // 反向遍历订阅的回调函数列表
            var _fn = fns[ l ];
            if ( _fn === fn ){
                fns.splice( l, 1 );    // 删除订阅者的回调函数
            }
        }
    }
};

var salesOffices = {};
var installEvent = function( obj ){
    for ( var i in event ){
        obj[ i ] = event[ i ];
    }
}

installEvent( salesOffices );

salesOffices.listen( 'squareMeter88', fn1 = function( price ){    // 小明订阅消息
    console.log( '价格= ' + price );
});

salesOffices.listen( 'squareMeter88', fn2 = function( price ){    // 小红订阅消息
    console.log( '价格= ' + price );
});

salesOffices.remove( 'squareMeter88', fn1 );    // 删除小明的订阅
salesOffices.trigger( 'squareMeter88', 2000000 );     // 输出：2000000
```

8.7 真实的例子——网站登录

通过售楼处的虚拟例子，我们对发布-订阅模式的概念和实现都已经熟悉了，那么现在就趁热打铁，看一个真实的项目。

假如我们正在开发一个商城网站，网站里有 header 头部、nav 导航、消息列表、购物车等模

块。这几个模块的渲染有一个共同的前提条件，就是必须先用 ajax 异步请求获取用户的登录信息。这是很正常的，比如用户的名字和头像要显示在 header 模块里，而这两个字段都来自用户登录后返回的信息。

至于 ajax 请求什么时候能成功返回用户信息，这点我们没有办法确定。现在的情节看起来像极了售楼处的例子，小明不知道什么时候开发商的售楼手续能够成功办下来。

但现在还不足以说服我们在此使用发布–订阅模式，因为异步的问题通常也可以用回调函数来解决。更重要的一点是，我们不知道除了 header 头部、nav 导航、消息列表、购物车之外，将来还有哪些模块需要使用这些用户信息。如果它们和用户信息模块产生了强耦合，比如下面这样的形式：

```
login.succ(function(data){
    header.setAvatar( data.avatar);   // 设置 header 模块的头像
    nav.setAvatar( data.avatar );     // 设置导航模块的头像
    message.refresh();                // 刷新消息列表
    cart.refresh();                   // 刷新购物车列表
});
```

现在登录模块是我们负责编写的，但我们还必须了解 header 模块里设置头像的方法叫 setAvatar、购物车模块里刷新的方法叫 refresh，这种耦合性会使程序变得僵硬，header 模块不能随意再改变 setAvatar 的方法名，它自身的名字也不能被改为 header1、header2。 这是针对具体实现编程的典型例子，针对具体实现编程是不被赞同的。

等到有一天，项目中又新增了一个收货地址管理的模块，这个模块本来是另一个同事所写的，而此时你正在马来西亚度假，但是他却不得不给你打电话：“Hi，登录之后麻烦刷新一下收货地址列表。”于是你又翻开你 3 个月前写的登录模块，在最后部分加上这行代码：

```
login.succ(function( data ){
    header.setAvatar( data.avatar);
    nav.setAvatar( data.avatar );
    message.refresh();
    cart.refresh();
    address.refresh();                // 增加这行代码
});
```

我们就会越来越疲于应付这些突如其来的业务要求，要么跳槽了事，要么必须来重构这些代码。

用发布–订阅模式重写之后，对用户信息感兴趣的业务模块将自行订阅登录成功的消息事件。当登录成功时，登录模块只需要发布登录成功的消息，而业务方接受到消息之后，就会开始进行各自的业务处理，登录模块并不关心业务方究竟要做什么，也不想去了解它们的内部细节。改善后的代码如下：

```
$.ajax( 'http:// xxx.com?login', function(data){    // 登录成功
    login.trigger( 'loginSucc', data);    // 发布登录成功的消息
});
```

各模块监听登录成功的消息：

```
var header = (function(){        // header 模块
    login.listen( 'loginSucc', function( data){
        header.setAvatar( data.avatar );
    });
    return {
        setAvatar: function( data ){
            console.log( '设置 header 模块的头像' );
        }
    }
})();

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

如上所述，我们随时可以把 setAvatar 的方法名改成 setTouxiang。如果有一天在登录完成之后，又增加一个刷新收货地址列表的行为，那么只要在收货地址模块里加上监听消息的方法即可，而这可以让开发该模块的同事自己完成，你作为登录模块的开发者，永远不用再关心这些行为了。代码如下：

```
var address = (function(){    // nav 模块
    login.listen( 'loginSucc', function( obj ){
        address.refresh( obj );
    });
    return {
        refresh: function( avatar ){
            console.log( '刷新收货地址列表' );
        }
    }
})();
```

8.8 全局的发布－订阅对象

回想下刚刚实现的发布-订阅模式，我们给售楼处对象和登录对象都添加了订阅和发布的功能，这里还存在两个小问题。

- 我们给每个发布者对象都添加了 listen 和 trigger 方法，以及一个缓存列表 clientList，这其实是一种资源浪费。
- 小明跟售楼处对象还是存在一定的耦合性，小明至少要知道售楼处对象的名字是 salesOffices，才能顺利的订阅到事件。见如下代码：

```
salesOffices.listen( 'squareMeter100', function( price ){     // 小明订阅消息
    console.log( '价格= ' + price );
});
```

如果小明还关心 300 平方米的房子，而这套房子的卖家是 salesOffices2，这意味着小明要开始订阅 salesOffices2 对象。见如下代码：

```
salesOffices2.listen( 'squareMeter300', function( price ){     // 小明订阅消息
    console.log( '价格= ' + price );
});
```

其实在现实中，买房子未必要亲自去售楼处，我们只要把订阅的请求交给中介公司，而各大房产公司也只需要通过中介公司来发布房子信息。这样一来，我们不用关心消息是来自哪个房产公司，我们在意的是能否顺利收到消息。当然，为了保证订阅者和发布者能顺利通信，订阅者和发布者都必须知道这个中介公司。

同样在程序中，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 = {},
        listen,
        trigger,
        remove;

    listen = function( key, fn ){
        if ( !clientList[ key ] ){
            clientList[ key ] = [];
        }
        clientList[ key ].push( fn );
    };

    trigger = function(){
        var key = Array.prototype.shift.call( arguments ),
            fns = clientList[ key ];
            if ( !fns || fns.length === 0 ){
                return false;
            }
            for( var i = 0, fn; fn = fns[ i++ ]; ){
                fn.apply( this, arguments );
}

    };

    remove = function( key, fn ){
        var fns = clientList[ key ];
        if ( !fns ){
            return false;
        }
```

```
        if ( !fn ){
            fns && ( fns.length = 0 );
        }else{
            for ( var l = fns.length - 1; l >=0; l-- ){
                var _fn = fns[ l ];
                if ( _fn === fn ){
                    fns.splice( l, 1 );
                }
            }
        }
    };

    return {
        listen: listen,
        trigger: trigger,
        remove: remove
    }

})();


Event.listen( 'squareMeter88', function( price ){     // 小红订阅消息
    console.log( '价格= ' + price );       // 输出：'价格=2000000'
});

Event.trigger( 'squareMeter88', 2000000 );    // 售楼处发布消息
```

8.9 模块间通信

上一节中实现的发布–订阅模式的实现，是基于一个全局的 Event 对象，我们利用它可以在两个封装良好的模块中进行通信，这两个模块可以完全不知道对方的存在。就如同有了中介公司之后，我们不再需要知道房子开售的消息来自哪个售楼处。

比如现在有两个模块，a 模块里面有一个按钮，每次点击按钮之后，b 模块里的 div 中会显示按钮的总点击次数，我们用全局发布–订阅模式完成下面的代码，使得 a 模块和 b 模块可以在保持封装性的前提下进行通信。

```
<!DOCTYPE html>
<html>

<body>
    <button id="count">点我</button>
    <div id="show"></div>
</body>

<script type="text/JavaScript">
var a = (function(){
    var count = 0;
    var button = document.getElementById( 'count' );
```

```
    button.onclick = function(){
        Event.trigger( 'add', count++ );
    }
})();

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

但在这里我们要留意另一个问题，模块之间如果用了太多的全局发布–订阅模式来通信，那么模块与模块之间的联系就被隐藏到了背后。我们最终会搞不清楚消息来自哪个模块，或者消息会流向哪些模块，这又会给我们的维护带来一些麻烦，也许某个模块的作用就是暴露一些接口给其他模块调用。

8.10　必须先订阅再发布吗

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。

8.11 全局事件的命名冲突

全局的发布-订阅对象里只有一个 clinetList 来存放消息名和回调函数，大家都通过它来订阅和发布各种消息，久而久之，难免会出现事件名冲突的情况，所以我们还可以给 Event 对象提供创建命名空间的功能。

在提供最终的代码之前，我们来感受一下怎么使用这两个新增的功能。

```
/************** 先发布后订阅 ********************/

Event.trigger( 'click', 1 );

Event.listen( 'click', function( a ){
    console.log( a );       // 输出：1
});

/************** 使用命名空间 ********************/

Event.create( 'namespace1' ).listen( 'click', function( a ){
    console.log( a );    // 输出：1
});

Event.create( 'namespace1' ).trigger( 'click', 1 );


Event.create( 'namespace2' ).listen( 'click', function( a ){
    console.log( a );     // 输出：2
});

Event.create( 'namespace2' ).trigger( 'click', 2 );
```

具体实现代码如下：

```
var Event = (function(){

    var global = this,
        Event,
        _default = 'default';

    Event = function(){
        var _listen,
            _trigger,
            _remove,
            _slice = Array.prototype.slice,
            _shift = Array.prototype.shift,
            _unshift = Array.prototype.unshift,
            namespaceCache = {},
            _create,
            find,
            each = function( ary, fn ){
                var ret;
                for ( var i = 0, l = ary.length; i < l; i++ ){
```

```
            var n = ary[i];
            ret = fn.call( n, i, n);
        }
        return ret;
    };

    _listen = function( key, fn, cache ){
        if ( !cache[ key ] ){
            cache[ key ] = [];
        }
       cache[key].push( fn );
    };

    _remove = function( key, cache ,fn){
        if ( cache[ key ] ){
            if( fn ){
                for( var i = cache[ key ].length; i >= 0; i-- ){
                    if( cache[ key ][i] === fn ){
                        cache[ key ].splice( i, 1 );
                    }
                }
            }else{
                cache[ key ] = [];
            }
        }
    };

    _trigger = function(){
        var cache = _shift.call(arguments),
            key = _shift.call(arguments),
            args = arguments,
            _self = this,
            ret,
            stack = cache[ key ];

        if ( !stack || !stack.length ){
            return;
        }

        return each( stack, function(){
            return this.apply( _self, args );
        });
    };

    _create = function( namespace ){
        var namespace = namespace || _default;
        var cache = {},
            offlineStack = [],    // 离线事件
            ret = {
                listen: function( key, fn, last ){
                    _listen( key, fn, cache );
                    if ( offlineStack === null ){
                        return;
                    }
                    if ( last === 'last' ){
```

```
                    offlineStack.length && offlineStack.pop()();
                }else{
                    each( offlineStack, function(){
                    this();
                });
            }

            offlineStack = null;
        },
        one: function( key, fn, last ){
            _remove( key, cache );
            this.listen( key, fn ,last );
        },
        remove: function( key, fn ){
            _remove( key, cache ,fn);
        },
        trigger: function(){
            var fn,
                args,
                _self = this;

            _unshift.call( arguments, cache );
            args = arguments;
            fn = function(){
                return _trigger.apply( _self, args );
            };

            if ( offlineStack ){
                return offlineStack.push( fn );
            }
            return fn();
        }
    };

    return namespace ?
        ( namespaceCache[ namespace ] ? namespaceCache[ namespace ] :
            namespaceCache[ namespace ] = ret )
                : ret;
};

return {
    create: _create,
    one: function( key,fn, last ){
        var event = this.create( );
            event.one( key,fn,last );
    },
    remove: function( key,fn ){
     var event = this.create( );
        event.remove( key,fn );
    },
    listen: function( key, fn, last ){
        var event = this.create( );
            event.listen( key, fn, last );
        },
    trigger: function(){
```

```
                var event = this.create( );
                event.trigger.apply( this, arguments );
            }
        };
    }();

    return Event;

})();
```

8.12　JavaScript 实现发布−订阅模式的便利性

这里要提出的是，我们一直讨论的发布−订阅模式，跟一些别的语言（比如 Java）中的实现还是有区别的。在 Java 中实现一个自己的发布−订阅模式，通常会把订阅者对象自身当成引用传入发布者对象中，同时订阅者对象还需提供一个名为诸如 update 的方法，供发布者对象在适合的时候调用。而在 JavaScript 中，我们用注册回调函数的形式来代替传统的发布−订阅模式，显得更加优雅和简单。

另外，在 JavaScript 中，我们无需去选择使用推模型还是拉模型。推模型是指在事件发生时，发布者一次性把所有更改的状态和数据都推送给订阅者。拉模型不同的地方是，发布者仅仅通知订阅者事件已经发生了，此外发布者要提供一些公开的接口供订阅者来主动拉取数据。拉模型的好处是可以让订阅者“按需获取”，但同时有可能让发布者变成一个“门户大开”的对象，同时增加了代码量和复杂度。

刚好在 JavaScript 中，`arguments` 可以很方便地表示参数列表，所以我们一般都会选择推模型，使用 `Function.prototype.apply` 方法把所有参数都推送给订阅者。

8.13　小结

本章我们学习了发布−订阅模式，也就是常说的观察者模式。发布−订阅模式在实际开发中非常有用。

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，但如果过度使用的话，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。