深入理解jQuery（2）——Callbacks

Alex Sun
2014-10-04

在jQuery中，一个很重要的辅助类便是Callbacks，用来管理回调函数。我们先来看一个例子：

```
function fn1() {
    console.log("fn1 is called");
}

function fn2() {
    console.log("fn2 is called");
}

var cb = $.Callbacks();
cb.add(fn1, fn2);
cb.fire();
```

当fire()调用后，fn1和fn2都会被执行。这是典型的观察者模式（发布订阅模式）。那么，如果让我们来设计，该如何实现呢？不难想象，我们需要维护一个数组，当执行add()的时候，就将回调函数添加到该数组中，当执行fire()的时候，就依次执行该数组中的回调函数。代码如下：

```
var myCallback = {
    list: [],
    add: function(fn) {
        this.list.push(fn);
        return this;
    },
    fire: function() {
        this.list.forEach(function(fn) {
            fn();
        });
        return this;
    }
}

myCallback.add(function() {
    console.log('fn1 is called');
}).add(function() {
    console.log('fn2 is called');
```

```
});

myCallback.fire();
```

当然，jQuery的Callbacks不是这么简单，jQuery的Callbacks支持如下选项：

- once：确保fire只执行一次，之后的fire不再执行
- memory：确保在fire之后add的回调函数也能够执行
- unique：当重复add某个回调函数时，确保回调函数列表中只存在该回调函数的一个副本
- stopOnFalse：如果某个回调函数在执行过程中返回false，那么之后的回调函数将不再执行

除此之外，Callbacks对象有如下方法：

- add：添加一个回调函数
- remove：删除一个回调函数
- has：判断一个回调函数是否在回调列表中
- empty：清空回调列表
- disable：禁用该回调对象
- disabled：获取禁用状态
- lock：锁定回调对象，即之后的fire不再执行
- locked：获取锁定状态
- fireWith：执行回调
- fire：执行回调
- fired：获取执行状态

下面我们将仿照jQuery实现一个简单的回调对象来实现上述功能，从而来理解jQuery的Callbacks，基本代码如下：

```
var MyCallbacks = function(options) {
    var defaultOptions = {
        once: false,
        unique: false
    };

    if (options) {
        for (var key in options) {
            defaultOptions[key] = options[key]
        };
    }
    options = defaultOptions;

    var list = [];
    var fired, firing;

    var self = {
        add: function(fn) {
            if (!options.unique || !this.has(fn)) {
```

```
                list.push(fn);
            }
            return this;
        },
        remove: function(fn) {
            var index = list.indexOf(fn);
            if (index > -1) {
                list.splice(index, 1);
            }
            return this;
        },
        has: function(fn) {
            return list.indexOf(fn) > -1;
        },
        empty: function() {
            list = [];
            return this;
        },
        fire: function() {
            if (!fired || !options.once) {
                firing = true;
                list.forEach(function(fn) {
                    fn();
                });
                firing = false;
            }
            fired = true;
        }
    }

    return self;
}



function fn1() {
    console.log('fn1 is called');
}

function fn2() {
    console.log('fn2 is called');
}

var cb1 = MyCallbacks({
    once: true
});
cb1.add(fn1);
cb1.fire();
cb1.fire();
```

```
var cb2 = MyCallbacks({
    unique: true
});
cb2.add(fn2);
cb2.add(fn1);
cb2.add(fn2)
cb2.fire();
```

这里我们仅仅实现了once和unique选项以及一部分方法，而且实现的功能很简单，没有考虑一些特殊情况。首先我们将传入的选项参数与默认的选项进行合并，得到配置后的选项，然后定义list（回调列表），fired（是否执行过回调），firing（是否正在执行回调）。接下来的方法都很容易理解，不做赘述。

其实在jQuery中，Callbacks的实现基本上也就是上面的思路，只不过更复杂些，考虑了各种特殊的情况，以及支持更多的选项配置和方法。下面我们将通过部分源码进行分析。

1. 配置选项

```
options = typeof options === "string" ?
    ( optionsCache[ options ] || createOptions( options ) ) :
    jQuery.extend( {}, options );
```

jQuery的Callbacks选项不仅支持对象形式，还支持字符串形式，例如”once memory unique”，当传入的是对象时，直接调用extend方法即可；如果传入的参数是字符串，那么首先查看optionsCache中是否有缓存，如果有，那么直接使用即可，如果没有，那么再将字符串解析为对象形式。其中optionsCache的格式为：

```
optionsCache = {
    'once unique': {
        once: true,
        unique: true
    }
}
```

即为字符串与对应的选项对象的对应。使用了cache之后，如果之前使用过某个字符串，那么之后如果再次遇到这个字符串就不必再进行解析了，直接使用即可。这也算是一个优化。

2. fire调用

通过分析源码，可以知道，当执行fire()的时候，调用顺序如下：

```
self.fire() --> self.fireWith() --> fire()
```

其中self.fire()基本上没做什么处理，就直接调用self.fireWith()，fireWith的源码如下：

```
fireWith: function(context, args) {
    if (list && (!fired || stack)) {
        args = args || [];
        args = [context, args.slice ? args.slice() : args];
        if (firing) {
            stack.push(args);
        } else {
            fire(args);
        }
    }
    return this;
}
```

其中stack = !options.once && []，因此当fired为true且once选项设置的时候，就不会进入if，也就是说，当设置了once且回调已经执行过的情况下，就不会再次执行，否则就会执行回调。但是我们发现，这里还要对firing进行一个判断，如果当前正在执行回调，那么执行stack.push(args)，否则去执行回调，这又是为什么呢？使用我们自定义的回调对象，考虑如下情况：

```
function fn1() {
    alert('fn1 is called');
    cb1.fire();

}

function fn2() {
    alert('fn2 is called');
}

var cb1 = MyCallbacks();
cb1.add(fn1);
cb1.add(fn2);
cb1.fire();
```

会发生什么情况呢，通过测试我们会发现，函数陷入了死循环，因为当我们执行fn1的时候，再次出发了回调，于是就会一直在执行fn1，这正是上面stack.push(args)所要解决的问题。在jQuery中，如果在执行回调的过程中再次触发了回调，那么再次触发的回调暂不执行，而是将其加入到一个队列中，等到当前回调执行结束后，再去执行它。下面，让我们来看看fire函数的代码：

```
    fire = function(data) {
        memory = options.memory && data;
        fired = true;
        firingIndex = firingStart || 0;
        firingStart = 0;
        firingLength = list.length;
        firing = true;
        for (; list && firingIndex < firingLength; firingIndex++) {
            if (list[firingIndex].apply(data[0], data[1]) === false &&
    options.stopOnFalse) {
                memory = false; // To prevent further calls using add
                break;
            }
        }
        firing = false;
        if (list) {
            if (stack) {
                if (stack.length) {
                    fire(stack.shift());
                }
            } else if (memory) {
                list = [];
            } else {
                self.disable();
            }
        }
    }
```

我们可以发现，首先是一个for循环执行回调，当执行完后，会判断stack的状态，如果队列中还有回调，那么会接着执行。