深入理解jQuery（4）——Deferred

Alex Sun
2014-10-16

1. Deferred与Callbacks

在异步编程中，代码往往不是按照顺序由上到下执行的，例如下面的例子：

```
setTimeout(function() {
    console.log("hello");
});
console.log("Alex");
```

就会先打印出Alex再打印出hello，如果我们希望先打印hello再打印Alex，我们可以将后面的语句调整到回调函数中，即：

```
setTimeout(function() {
    console.log("hello");
    console.log("Alex");
});
```

但是这样就会产生另外一个问题，即第二个console语句的作用域发生了变化。在一些简单的例子中不会有什么影响，但是如果业务逻辑比较复杂的话，为了保证函数的作用域，可能需要做出更多的调整，增加代码的复杂度，降低了可读性。在很多情况下，我们可以通过Callbacks来解决，如下：

```
var cb = $.Callbacks();
setTimeout(function() {
    console.log("hello");
    cb.fire();
});
cb.add(function() {
    console.log("Alex");
});
```

Callbacks在之前的博客中讲到过，是jQuery的回调函数解决方案，不过，jQuery在此基础上，另外开发了一个功能模块，即Deferred，是延迟对象的解决方案。Callbacks是Deferred的实现基础，相比之下，Callbacks是更高层次的抽象，而Deferred则更加具体细化。jQuery的许多功能，例如Ajax等，都是基于Deferred进行开发的。如果我们使用Deferred来实现上面的功能，代码如下：

```
var dfd = $.Deferred();
setTimeout(function() {
    console.log("hello");
    dfd.resolve();
});
dfd.done(function() {
    console.log("Alex");
});
```

我们可以发现，Deferred的使用与Callbacks非常类似，只是方法名不同而已。事实上，fire与resolve，add与done，在本质上是等价的。下面我们将通过源码进行分析。

2. Deferred源码分析

整个Deferred模块的框架如下，可以发现，整个模块只是为jQuery添加了两个类方法：

```
jQuery.extend({
    Deferred: function( func ) {

    },

    // Deferred helper
    when: function( subordinate /* , ..., subordinateN */ ) {

    }
});
```

下面是Deferred方法的结构：

```
Deferred: function(func) {
    var tuples = [
            // action, add listener, listener list, final state
            ["resolve", "done", jQuery.Callbacks("once memory"),
"resolved"],
            ["reject", "fail", jQuery.Callbacks("once memory"),
"rejected"],
            ["notify", "progress", jQuery.Callbacks("memory")]
```

```
            ],
            state = "pending",
            promise = {
                state: function() {},
                always: function() {},
                then: function() {},
                promise: function() {}
            },
            deferred = {};

        // Keep pipe for back-compat
        promise.pipe = promise.then;

        // Add list-specific methods
        jQuery.each(tuples, function(i, tuple) {
            // do something
        });

        // Make the deferred a promise
        promise.promise(deferred);

        // Call given func if any
        if (func) {
            func.call(deferred, deferred);
        }

        // All done!
        return deferred;
    }
```

首先是tuples定义了三组状态，分别是成功，失败和进行中。然后有一个promise对象和一个deferred对象，经过之后的处理，最终返回deferred对象。下面我们先来分析jQuery.each：

```
    // Add list-specific methods
    jQuery.each(tuples, function(i, tuple) {
        // list事实上就是一个Callbacks对象
        var list = tuple[2],
            stateString = tuple[3];

        // promise[ done | fail | progress ] = list.add
        // done()，fail()和progress()方法其实都是Callbacks的add()方法
        promise[tuple[1]] = list.add;

        // Handle state
        // 当动作为resolve或者reject时，为list添加回调函数
        if (stateString) {
            list.add(function() {
```

```
                // state = [ resolved | rejected ]
                state = stateString;

                // [ reject_list | resolve_list ].disable;
    progress_list.lock
                // 即resolve和reject只执行一个，当某个执行时，另一个就disable掉
            }, tuples[i ^ 1][2].disable, tuples[2][2].lock);
        }

        // deferred[ resolve | reject | notify ]
        // resolve()，reject()和notify()对应的是Callbacks的fire()
        deferred[tuple[0]] = function() {
            deferred[tuple[0] + "With"](this === deferred ? promise : this,
    arguments);
            return this;
        };
        // resolveWith()，rejectWith()和notifyWith()对应的是Callbacks的
    fireWith()
        deferred[tuple[0] + "With"] = list.fireWith;
    });
```

到此为止，我们可以构建出promise和deferred这两个对象的大致轮廓了：

```
    promise={
        state(),
        always(),
        then(),
        promise(),
        pipe(),
        done(),
        fail(),
        progress()
    }

    deferred={
        resolve(),
        reject(),
        notify(),
        resolveWith(),
        rejectWith(),
        notifyWith()
    }
```

接下来执行的是promise.promise(deferred)，我们来看看promise对象的promise()方法：

```
    // Get a promise for this deferred
```

```
// If obj is provided, the promise aspect is added to the object
// 当obj不为null是，将promise对象扩展到obj上，否则返回promise对象
// 需要注意undefined!=null为false，而undefined!==null为true
promise: function(obj) {
    return obj != null ? jQuery.extend(obj, promise) : promise;
}
```

到此为止，我们将promise对象扩展到了deferred对象上。最后我们返回的是deferred对象。

3. promise源码分析

通过上面的分析我们知道了promise对象的结构如下：

```
promise={
    state(),
    always(),
    then(),
    promise(),
    pipe(),
    done(),
    fail(),
    progress()
}
```

下面我们队promise对象进行一个详细分析。

首先是state()方法，返回延迟对象的状态，有三种状态：resolved、rejected和pending。

always()就是说，无论成功还是失败都会执行，其源码如下：

```
always: function() {
    deferred.done(arguments).fail(arguments);
    return this;
},
```

promise()方法的源码在上面已经分析过了。然后done()、fail()和progress()事实上都是Callbacks的add()方法。那么就剩下then()和pipe()了，源码中有这样一句：

```
// Keep pipe for back-compat
promise.pipe = promise.then;
```

也就是说，then()和pipe()其实是同一个方法，而保留pipe()是为了兼容低版本。然而事实上，虽然then()和pipe()使用的是同一套代码，但是它们的功能是有区别的。首先我们先来了解then()方法的作用，then()方法可以接受三个参数，每个参数都是一个回调函数，分别对应resolve、reject和notify，例子如下：

```
var dfd = $.Deferred();
setTimeout(function() {
    dfd.resolve();
}, 1000);

dfd.then(function() {
    console.log("success");
}, function() {
    console.log("fail");
}, function() {
    console.log("progress");
});
```

下面我们来看pipe()的用法，pipe()的作用是返回一个新的延迟对象，类似于一个filter的作用，例子如下：

```
var dfd = $.Deferred();
setTimeout(function() {
    dfd.resolve("hello");
}, 1000);

var newDfd = dfd.pipe(function() {
    return arguments[0] + " Alex"
});

newDfd.done(function() {
    console.log(arguments[0]);
});
```

然后我们来看then()的源码，如下：

```
then: function( /* fnDone, fnFail, fnProgress */ ) {
    var fns = arguments;

    // return事实上与then无关，而是与pipe相关的
    // 返回一个新的延迟对象的promise
    return jQuery.Deferred(function(newDefer) {
        jQuery.each(tuples, function(i, tuple) {
            // tuple的顺序和fns的顺序是一致的，都是resolve、reject、notify
            // 如果fns[i]是函数，则fn=fns[i]，否则fn=false
```

```
                var action = tuple[0],
                    fn = jQuery.isFunction(fns[i]) && fns[i];

                // deferred[ done | fail | progress ] for forwarding actions
    to newDefer
                // 执行回调函数
                deferred[tuple[1]](function() {
                    var returned = fn && fn.apply(this, arguments);

                    // if-else 事实上与then无关，而是与pipe相关的
                    if (returned && jQuery.isFunction(returned.promise)) {
                        // 如果回调完成，且返回的是一个对象，则立即继续执行新的延迟对
    象的回调
                        // 此时returned是之前的回调对象
                        returned.promise()
                            .done(newDefer.resolve)
                            .fail(newDefer.reject)
                            .progress(newDefer.notify);
                    } else {
                        // 如果回调未完成，或者回调返回值不是一个对象，则执行新的回调
    对象的fireWith
                        newDefer[action + "With"](this === promise ?
    newDefer.promise() : this, fn ? [returned] : arguments);
                    }
                });
            });
            fns = null;
        }).promise();
    },
```

在实际使用中，常用的是then()，pipe()已经很少使用到了。

4. promise与deferred

在Derferred中，有一个promise对象和一个deferred对象，相比之下，deferred具有promise的全部功能，并且多了resolve()，reject()，notify()，resolveWith()，rejectWith()，notifyWith()这六个函数。在源码中，是通过promise.promise(deferred)将promise的属性扩展到deferred上的，在这之后，我们可以通过deferred.promise()来得到其promise对象。

那么，为什么要设计的这么复杂，直接使用一个deferred对象不可以吗？我们来看下面的例子：

```
function a() {
    var dfd = $.Deferred();
    setTimeout(function() {
```

```
        dfd.resolve();
    }, 1000);
    return dfd;
}

var dfd = a();
dfd.done(function() {
    console.log("done");
}).fail(function() {
    console.log("fail");
});
dfd.reject();
```

在执行函数a()的时候，我们设置1秒后出发resolve事件，然后在外面，我们触发了reject事件，最后打印出的是fail，这显然与我们的期望（即希望resolve）是不符合的。也就是说，在这种情况下，我们可以在外部随意更改延迟对象的状态。为了避免这种情况，我们修改代码如下：

```
function a() {
    var dfd = $.Deferred();
    setTimeout(function() {
        dfd.resolve();
    }, 1000);
    return dfd.promise();
}

var dfd = a();
dfd.done(function() {
    console.log("done");
}).fail(function() {
    console.log("fail");
});
dfd.reject(); // error
```

这里我们返回的不是deferred对象，而是其promise对象。前面已经提到，promise与deferred相比，少了修改状态的功能，因此我们可以再外面安全地使用它，如果我们在外面试图修改延迟对象的状态，则会报错。

5. 多个异步任务

之前的例子都是一个异步函数完成或者失败后调用回调，那么假如现在有多个异步任务，并且我们希望在它们都完成的时候才调用回调，该如何实现呢。最笨的方法就是异步函数的层层嵌套，化异步为同步，但是这样也就失去了异步的意义。

我们可以假设有一个计数器，初始化的时候，该计数器的值等于异步任务的个数，然后每当一个异步任务完成时就将计数器减1，当计数器为0的时候就调用回调函数。基于这样的一个思路，可以写出如下例子：

```
function a(callback) {
    setTimeout(function() {
        console.log("a");
        checkDone(callback);
    }, 1000);
}

function b(callback) {
    setTimeout(function() {
        console.log("b");
        checkDone(callback);
    }, 2000);
}

function c(callback) {
    setTimeout(function() {
        console.log("c");
        checkDone(callback);
    }, 3000);
}

function checkDone(callback) {
    count--;
    if (count == 0) {
        callback();
    }
}

function allDone() {
    console.log("All done!");
}

function foo(funcs, callback) {
    count = funcs.length;
    funcs.forEach(function(func) {
        func(callback);
    });
}

foo([a, b, c], allDone);
```

这里可以认为a，b，c为三个异步任务，分别需要1秒，2秒，3秒的时间才能完成。我们希望当这三个任务都完成的时候才打印出“All done!”，如果按照通常的思路，是比较难实现的，如果我们使

用异步的嵌套，如下所示：

```
setTimeout(function() {
    console.log("a");
    setTimeout(function() {
        console.log("b");
        setTimeout(function() {
            console.log("c");
            console.log("All done!")
        }, 3000);
    }, 2000);
}, 1000);
```

这样就一共需要6秒才能完成。而在上面的例子中，一共也就需三秒的时间。首先是foo()函数，接受两个参数，第一个参数是异步任务列表，第二个参数是回调函数。我们初始化计数器为异步任务的个数，然后开始执行异步任务。当每个异步任务执行结束的时候，我们都检测异步任务是否都执行结束，即checkDone()方法，如果发现异步任务都已经结束，即计数器为0，则执行回调函数。

jQuery中的when方法也就是基于这样的一个思路来实现的，下面将详细介绍。

6. when

在分析源码之前，我们先来了解when()方法的作用，如下例子：

```
function a() {
    var dfd = $.Deferred();
    dfd.resolve();
    return dfd;
}

function b() {
    var dfd = $.Deferred();
    dfd.reject();
    return dfd;
}

$.when(a(), b()).done(function() {
    console.log("done");
}).fail(function() {
    console.log("fail");
});
```

这里需要注意的是：

- 只有当全部resolve的时候才会触发done
- 只要有一个reject就会触发fail
- when的每个参数都应该为延迟对象，如果某个参数不是，则会忽略掉。例子中如果b()无返回值，那么就会触发resolve

下面我们来对when()的源码进行分析。

```
// Deferred helper
when: function( subordinate /* , ..., subordinateN */ ) {
    var i = 0,
        resolveValues = core_slice.call( arguments ),
        length = resolveValues.length,

        // the count of uncompleted subordinates
        // remaining就是一个计数器
        // 当只有一个参数且参数不为延迟对象的时候，remaining为0，否则remaining的值为参数个数
        remaining = length !== 1 || ( subordinate && jQuery.isFunction( subordinate.promise ) ) ? length : 0,

        // the master Deferred. If resolveValues consist of only a single Deferred, just use that.
        // 如果参数仅有一个且是延迟对象，那么就使用这个延迟对象，否则新创建一个延迟对象
        deferred = remaining === 1 ? subordinate : jQuery.Deferred(),

        // Update function for both resolve and progress values
        updateFunc = function( i, contexts, values ) {
            return function( value ) {
                contexts[ i ] = this;
                values[ i ] = arguments.length > 1 ? core_slice.call( arguments ) : value;
                if( values === progressValues ) {
                    deferred.notifyWith( contexts, values );
                } else if ( !( --remaining ) ) {
                    // 当某个延迟对象触发resolve的时候，将计数器减1
                    // 如果计数器减1后为0，则触发deferred的resolve
                    deferred.resolveWith( contexts, values );
                }
            };
        },

        progressValues, progressContexts, resolveContexts;

    // add listeners to Deferred subordinates; treat others as resolved
    if ( length > 1 ) {
```

```
        progressValues = new Array( length );
        progressContexts = new Array( length );
        resolveContexts = new Array( length );
        for ( ; i < length; i++ ) {
            // resolveValues为传入的参数，如果某个参数为延迟对象，则执行下面代码
            if ( resolveValues[ i ] && jQuery.isFunction( resolveValues[
i ].promise ) ) {
                // 如果某个延迟对象执行完成或进行中，则触发updateFunc
                // 一旦失败，则立即触发reject
                resolveValues[ i ].promise()
                    .done( updateFunc( i, resolveContexts, resolveValues
) )
                    .fail( deferred.reject )
                    .progress( updateFunc( i, progressContexts,
progressValues ) );
            } else {
                // 如果某个参数不是延迟对象，则执行--remaining，即将计数器减1
                --remaining;
            }
        }
    }

    // if we're not waiting on anything, resolve the master
    // 如果计数器为0，则直接触发resolve
    if ( !remaining ) {
        deferred.resolveWith( resolveContexts, resolveValues );
    }

    return deferred.promise();
}
```