# 整体架构-jQuery3.2.1

# 创建jQuery对象----(L94)

# jQuery原型添加方法与属性

# 核心函数

# 选择器sizzle引擎

# 回调对象

# Deferred对象(异步队列)

## 概述

### Deferreds对象

jQuery1.5中新增的Deferreds对象，可以将任务完成的处理方式与任务本身解耦合。举一个简单例子，因为异步导致流程并不是输出1,2,3，Deferreds 对这个问题提供了一个更好的解决方案，它是非阻塞的，并且与代码完全解耦 。

Promise/A只是一种规范（通过一组API规范异步化操作，让异步操作流程控制更加容易），Deferred可以看作这种规范的具体实现;

**相当于异步统一管理**

主要作用：1、解决时序以及动态添加执行函数的问题；

alert(1)  
setTimeout(function(){alert(2)},0)  
alert(3)  
//alert(1)   
//alert(3)  
//alert(2)

2、解决参数传递的问题，比如所有函数需要相同的参数

* + - 1. 每次调用runList时，都需要传递list与val两个参数，这种重复劳动没有任何意义。

var list = [],  
 val = 0;  
list[list.length] = a;  
list[list.length] = b;  
function runList(listArray,value){  
 for(var i = 0; i < listArray.length; i++){  
 listArray[i](value);  
 }  
 listArray.length = 0;  
}  
val = 5;  
runList(list,val );//执行a = 5;b = 5;

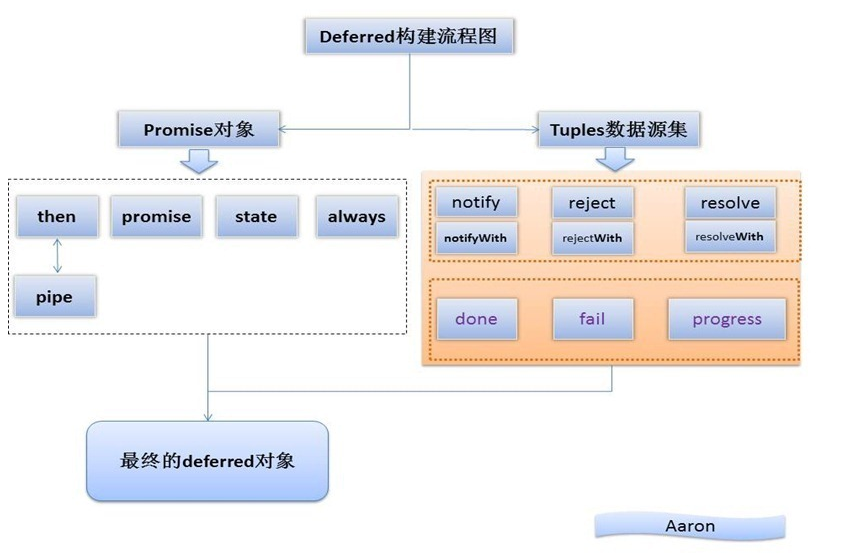
list[list.length] = function c(value){alert("c = " + value)};  
runList(list,val);//执行结果c = 5

### Promises

大多情况下，promise作为一个模型，提供了一个在软件工程中描述延时（或将来）概念的解决方案。

背后的思想是：不是执行一个方法然后阻塞应用程序等待结果返回，而是返回一个promise对象来满足未来值。

### 构建Deffered对象流程图



### jQuery涉及到promise的相关实现都不很理解

## Deferred对象方法简介

### $.Deferred() :生成一个deferred对象。

### Done、fail、then:

成功时回调与失败的回调、准确说应该是添加成功或失败回调；

then将成功与失败回调整合在一起

### Resolve():手动改变deferred对象运行状态为“已完成”，从而触发done()方法

### Reject():手动改变deferred对象运行状态为“已失效”，从而触发fail()方法

### Always()：无论调用resolve()还是reject()，总是执行

## Deferred对象的基本用法

* + - 1. 输出顺序，aaa,111,bbb

function runAsync(){  
 var def = $.Deferred();  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 def.resolve('bbb');  
 }, 0);  
 return def;  
}  
runAsync().then(function(data){  
 console.log('1111');  
 console.log(data)  
});

* + - 1. ES6的promise实现方式

function runAsync(){  
 var p = new Promise(function(resolve, reject){  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 resolve('bbb');  
 }, 0);  
 });  
 return p;  
}  
runAsync().then(function (data) {  
 console.log('1111');  
 console.log(data)  
});

* + - * 1. 与jQuery的Deferred重要区别，deferred上具有resolve方法，可以在外部改变def状态
      1. 外部修改deferred状态
         1. 输出结果：aaaa,111, 在外部结束

function runAsync(){  
 var def = $.Deferred();  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 def.resolve('bbb');  
 }, 0);  
 return def;  
}  
var d = runAsync();  
d.then(function(data){  
 console.log('1111');  
 console.log(data)  
});  
d.resolve('在外部结束');

* + - 1. 通过def.promise()避免这样的问题出现
         1. 在(1)中的runAsync中，返回def.promise,这样就不能从外部修改状态，调用d.resolve会报错

## 源码分析

### 总览

Callback被剥离之后，deffered非常精简，精简之后为

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

var tuples = [];

promise = {  
 state: function(){};

always: function(){};

………………

}

// 合并promise到deferred

promise.promise( deferred );

}

when: function( singleValue ){}

} );

* + - * 1. Deferred是个工厂类，返回的是内部构建的deferred对象；
        2. tuples 创建三个$.Callbacks对象，分别表示成功，失败，处理中三种状态；
        3. 创建了一个promise对象，具有state、always、then、primise方法
        4. 扩展primise对象生成最终的Deferred对象，返回该对象

### 源码（L3469）

* + - 1. 实现并不理解，仅了解大概，认为是promiseA规范下的回调队列

function Identity( v ) {  
 return v;  
}  
function Thrower( ex ) {  
 throw ex;  
}  
// jQuery.when使用的回调工厂  
function adoptValue( value, resolve, reject, noValue ) {  
 var method;  
  
 try {  
  
 // 检查value值是否为promise，Check for promise aspect first to privilege synchronous behavior  
 if ( value && jQuery.isFunction( ( method = value.promise ) ) ) {  
 method.call( value ).done( resolve ).fail( reject );  
  
 // 其他 thenables值类型  
 } else if ( value && jQuery.isFunction( ( method = value.then ) ) ) {  
 method.call( value, resolve, reject );  
  
 // 非thenables值类型  
 } else {  
  
 // 通过数组控制 `resolve`参数个数  
 // slice函数会将boolean的noValue值转换为integer  
 // \* false: [ value ].slice( 0 ) => resolve( value )  
 // \* true: [ value ].slice( 1 ) => resolve()  
 resolve.apply( undefined, [ value ].slice( noValue ) );  
 }  
  
 // 对于Promises/A+, 将exceptions转换为rejections  
 // 因为jQuery.when并不会展开thenables，我们可以跳过额外的检查，然后有条件的拒绝  
 } catch ( value ) {  
  
 // Support: Android 4.0 only  
 // 严格模式下，函数调用.call/.apply获取不到全局对象上下文  
 reject.apply( undefined, [ value ] );  
 }  
}  
  
jQuery.extend( {  
  
 Deferred: function( func ) {  
 // 创建三个$.Callbacks对象，分别表示成功，失败，处理中三种状态  
 // 创建过程在下面的jQuery.each，此数组只是抽取公共部分，方便处理  
 var tuples = [  
  
 // action, add listener, callbacks,  
 // ... .then handlers, argument index, [final state]  
 // "memomry"：保存之前的值，当新函数add到list时，无需fire，自动执行  
 // "once" : 确保回调列表只能fired一次（如Deferred）  
 [ "notify", "progress", jQuery.Callbacks( "memory" ),  
 jQuery.Callbacks( "memory" ), 2 ],  
 [ "resolve", "done", jQuery.Callbacks( "once memory" ),  
 jQuery.Callbacks( "once memory" ), 0, "resolved" ],  
 [ "reject", "fail", jQuery.Callbacks( "once memory" ),  
 jQuery.Callbacks( "once memory" ), 1, "rejected" ]  
 ],  
 state = "pending", // deferred默认状态  
 // promise是deferred的一种简化形式（去掉了改变状态的接口）  
 promise = {  
 state: function() {  
 return state;  
 },  
 // 函数列表总是执行  
 always: function() {  
 deferred.done( arguments ).fail( arguments );  
 return this;  
 },  
 "catch": function( fn ) {  
 return promise.then( null, fn );  
 },  
  
 // 保留 pipe 函数以备用  
 pipe: function( /\* fnDone, fnFail, fnProgress \*/ ) {  
 var fns = arguments;  
 // newDefer可以看作是newDefer = $.Deferred();  
 return jQuery.Deferred( function( newDefer ) {  
 jQuery.each( tuples, function( i, tuple ) {  
  
 // Map tuples (progress, done, fail) to arguments (done, fail, progress)  
 var fn = jQuery.isFunction( fns[ tuple[ 4 ] ] ) && fns[ tuple[ 4 ] ];  
 // 给父deferred对象的[ done | fail | progress ]方法都增加一个过滤函数的方法  
 // deferred.progress(function() { bind to newDefer or newDefer.notify })  
 // deferred.done(function() { bind to newDefer or newDefer.resolve })  
 // deferred.fail(function() { bind to newDefer or newDefer.reject })  
 deferred[ tuple[ 1 ] ]( function() {  
 var returned = fn && fn.apply( this, arguments );  
 if ( returned && jQuery.isFunction( returned.promise ) ) {  
 returned.promise()  
 .progress( newDefer.notify )  
 .done( newDefer.resolve )  
 .fail( newDefer.reject );  
 } else {  
 newDefer[ tuple[ 0 ] + "With" ](  
 this,  
 fn ? [ returned ] : arguments  
 );  
 }  
 } );  
 } );  
 fns = null;  
 } ).promise();  
 },  
 /\*\*  
 \* ?????????????????????对实现并不理解，关键是其中resolve函数是实现promiseA规范的  
 \* 理解的则是deferred是具有promise特性的callback回调序列  
 \* @param onFulfilled resolve时调用的回调函数  
 \* @param onRejected reject时调用的回调函数  
 \* @param onProgress progress()方法调用的回调函数。  
 \* @return {\*} deferred.promise()即deferred的简易版本  
 \*/  
 then: function( onFulfilled, onRejected, onProgress ) {  
 var maxDepth = 0;  
 // 在then的return中调用了resolve函数，以第一个举例  
 // depth:0,deferred=newDefer，handler：onProgress,special:newDefer.notifyWith  
 function resolve( depth, deferred, handler, special ) {  
 return function() {  
 var that = this,  
 args = arguments,  
 mightThrow = function() {  
 var returned, then;  
  
 // Support: Promises/A+ section 2.3.3.3.3  
 // https://promisesaplus.com/#point-59  
 // 忽略多次解决尝试  
 if ( depth < maxDepth ) {  
 return;  
 }  
  
 returned = handler.apply( that, args );  
  
 // Support: Promises/A+ section 2.3.1  
 // https://promisesaplus.com/#point-48  
 // If promise and x refer to the same object, reject promise with a TypeError as the reason.  
 if ( returned === deferred.promise() ) {  
 throw new TypeError( "Thenable self-resolution" );  
 }  
  
 // Support: Promises/A+ sections 2.3.3.1, 3.5  
 // https://promisesaplus.com/#point-54 Let then be x.then  
 // https://promisesaplus.com/#point-75  
 // 仅检索 `then` 一次  
 then = returned &&  
  
 // Support: Promises/A+ section 2.3.4  
 // https://promisesaplus.com/#point-64  
 // If x is not an object or function, fulfill promise with x.  
 ( typeof returned === "object" ||  
 typeof returned === "function" ) &&  
 returned.then;  
  
 // Handle a returned thenable  
 if ( jQuery.isFunction( then ) ) {  
  
 // Special processors (notify) just wait for resolution  
 if ( special ) {  
 then.call(  
 returned,  
 resolve( maxDepth, deferred, Identity, special ),  
 resolve( maxDepth, deferred, Thrower, special )  
 );  
  
 // Normal processors (resolve) also hook into progress  
 } else {  
  
 // ...and disregard older resolution values  
 maxDepth++;  
  
 then.call(  
 returned,  
 resolve( maxDepth, deferred, Identity, special ),  
 resolve( maxDepth, deferred, Thrower, special ),  
 resolve( maxDepth, deferred, Identity,  
 deferred.notifyWith )  
 );  
 }  
  
 // Handle all other returned values  
 } else {  
  
 // Only substitute handlers pass on context  
 // and multiple values (non-spec behavior)  
 if ( handler !== Identity ) {  
 that = undefined;  
 args = [ returned ];  
 }  
  
 // Process the value(s)  
 // Default process is resolve  
 ( special || deferred.resolveWith )( that, args );  
 }  
 },  
  
 // Only normal processors (resolve) catch and reject exceptions  
 process = special ?  
 mightThrow :  
 function() {  
 try {  
 mightThrow();  
 } catch ( e ) {  
  
 if ( jQuery.Deferred.exceptionHook ) {  
 jQuery.Deferred.exceptionHook( e,  
 process.stackTrace );  
 }  
  
 // Support: Promises/A+ section 2.3.3.3.4.1  
 // https://promisesaplus.com/#point-61  
 // Ignore post-resolution exceptions  
 if ( depth + 1 >= maxDepth ) {  
  
 // Only substitute handlers pass on context  
 // and multiple values (non-spec behavior)  
 if ( handler !== Thrower ) {  
 that = undefined;  
 args = [ e ];  
 }  
  
 deferred.rejectWith( that, args );  
 }  
 }  
 };  
  
 // Support: Promises/A+ section 2.3.3.3.1  
 // https://promisesaplus.com/#point-57  
 // 立即重新承诺，避免错误的拒绝。  
 if ( depth ) {  
 process();  
 } else {  
  
 // Call an optional hook to record the stack, in case of exception  
 // since it's otherwise lost when execution goes async  
 if ( jQuery.Deferred.getStackHook ) {  
 process.stackTrace = jQuery.Deferred.getStackHook();  
 }  
 window.setTimeout( process );  
 }  
 };  
 }  
 // 最终返回的是$.Deferred(func).promise();  
 // 根据deferred对象构建中，func.call(deferred,deferred);  
 // 因此newDefer指向构建好的jQuery.Deferred对象  
 return jQuery.Deferred( function( newDefer ) {  
  
 // progress\_handlers.add( ... )  
 tuples[ 0 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onProgress ) ?  
 onProgress :  
 Identity,// 返回参数  
 newDefer.notifyWith  
 )  
 );  
  
 // fulfilled\_handlers.add( ... )  
 tuples[ 1 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onFulfilled ) ?  
 onFulfilled :  
 Identity  
 )  
 );  
  
 // rejected\_handlers.add( ... )  
 tuples[ 2 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onRejected ) ?  
 onRejected :  
 Thrower  
 )  
 );  
 } ).promise();// 返回新的promise对象  
 },  
  
 // 为deferred对象获得一个promise  
 // 如obj设置了，则将deferred对象添加到这个对象上  
 // 未传入参数则直接返回promise  
 promise: function( obj ) {  
 return obj != null ? jQuery.extend( obj, promise ) : promise;  
 }  
 },  
 deferred = {};  
  
 // 针对tuples 元素集  
 // 把相同有共同特性的代码的给合并成一种结构，然后通过一次处理  
 // 分别创建a.done,a.progress,a.fail(var a = $.Deferred();)  
 jQuery.each( tuples, function( i, tuple ) {  
 // i= 0时，为[ "notify", "progress", jQuery.Callbacks( "memory" ),jQuery.Callbacks( "memory" ),2 ]  
 // list = tuple[ 2 ]，相当于$.Callbacks("once memory");这一列  
 // callbacks数组,创建不同的Callback对象  
 // 假设三个列表分别为,progressCallback（进度回调列表）,doneCallback（成功回调列表）,failCallback（失败回调列表）  
 var list = tuple[ 2 ],  
 stateString = tuple[ 5 ];// 最终状态，只有i=1,i=2存在  
  
 // promise.progress = progressCallback.add  
 // promise.done = doneCallback.add  
 // promise.fail = failCallback.add  
 // 就是为promise赋jQuery.Callbacks  
 promise[ tuple[ 1 ] ] = list.add;// $.Callback()回调对象的add方法  
  
 // 处理状态，为resolved和rejected两种状态的列表添加预设回调函数  
 if ( stateString ) {  
 // 将3个函数添加到list中  
 list.add(  
 function() {// 为list添加当前状态标识函数  
  
 // state = "resolved" (i.e., fulfilled)  
 // state = "rejected"  
 state = stateString;  
 },  
  
 // 为doneCallback，添加failCallback.disable  
 // 为failCallback，添加为doneCallback.disable  
 // 主要目的是禁止修改状态，如已经成功(i=1时)，则需要将reject列表禁用，不能再执行失败函数  
 tuples[ 3 - i ][ 2 ].disable,// 禁用 .fire and .add  
  
 // progress\_callbacks.lock  
 // 锁定notify列表，即notify列表fire方法不能被调用，但因为callback的options为memory  
 // 故还可以调用add方法  
 tuples[ 0 ][ 2 ].lock // 禁用 .fire  
 );  
 }  
  
 // progress\_handlers.fire  
 // fulfilled\_handlers.fire  
 // rejected\_handlers.fire  
 // 这之后，相当于为list添加了4个函数（对于有stateString状态的）  
 list.add( tuple[ 3 ].fire );  
  
 // deferred.notify = function() { deferred.notifyWith(...) }  
 // deferred.resolve = function() { deferred.resolveWith(...) }  
 // deferred.reject = function() { deferred.rejectWith(...) }  
 // 执行deferred.notifyWith(...)并返回this  
 deferred[ tuple[ 0 ] ] = function() {  
 deferred[ tuple[ 0 ] + "With" ]( this === deferred ? undefined : this, arguments );  
 return this;  
 };  
  
 // deferred.notifyWith = list.fireWith  
 // deferred.resolveWith = list.fireWith  
 // deferred.rejectWith = list.fireWith  
 // fireWith:访问给定的上下文和参数列表中的所有回调  
 // resolveWith/rejectWith/notifyWith 是 callbacks.fireWith 队列方法引用  
 deferred[ tuple[ 0 ] + "With" ] = list.fireWith;  
 } );  
 // 上面foreach整个循环完，deferred包含notify,notifyWith,reject,rejectWith,resolve,resolveWith,6个函数  
 // promise对象增加了progress，done，fail三个函数(其实就是list.add函数)，done,fail因为是Callback，里面list有4个函数  
  
 // 将3个tuples全部处理完后的promise合并到deferred  
 promise.promise( deferred );  
  
 // 如传入参数则调用，then方法就传入了func  
 // 则将构建好的deferred作为执行对象和参数，在then方法中就有调用  
 if ( func ) {  
 func.call( deferred, deferred );  
 }  
  
 // 返回deferred对象  
 return deferred;  
 //小结，  
 // 因此deferred.done,deferred.fail,deferred.progress其实就是不同$.callbacks().add()方法  
 // deferred.resolve,deferred.reject,deferred.notify，其实是调用的$.callback().fireWith()  
 // promise对象是deferred的简易版，没有上述6个函数，故无法改变状态  
 },  
  
 /\*\*  
 \* 提供一种方法来执行一个或多个对象的回调函数，返回这些对象的延时（Deferred）对象。  
 \* 1、无参数， jQuery.when()将返回一个resolved（解决）状态的promise对象。  
 \*  
 \*/  
 when: function( singleValue ) {  
 var  
  
 // 参数个数  
 remaining = arguments.length,  
  
 // 未处理的参数个数  
 i = remaining,  
  
 resolveContexts = Array( i ),  
 resolveValues = slice.call( arguments ),// 将参数类数组转换为数组  
  
 // 主Deferred对象  
 master = jQuery.Deferred(),  
  
 // 次要回调工厂  
 updateFunc = function( i ) {  
 return function( value ) {  
 resolveContexts[ i ] = this;  
 resolveValues[ i ] = arguments.length > 1 ? slice.call( arguments ) : value;  
 if ( !( --remaining ) ) {  
 master.resolveWith( resolveContexts, resolveValues );//相当于fire  
 }  
 };  
 };  
  
 // when无参数或单个参数，采用Promise.resolve  
 // 如无参数，则等价于：master.done( updateFunc( i ) ).resolve()  
 if ( remaining <= 1 ) { // i = remaining  
 adoptValue( singleValue, master.done( updateFunc( i ) ).resolve, master.reject,  
 !remaining );//如无参数则！remaining为true  
  
 // 使用 .then() to 展开thenables (cf. gh-3000)  
 if ( master.state() === "pending" ||  
 jQuery.isFunction( resolveValues[ i ] && resolveValues[ i ].then ) ) {  
  
 return master.then();  
 }  
 }  
  
 // 多参数形式，就如同Promise.all  
 while ( i-- ) {  
 adoptValue( resolveValues[ i ], updateFunc( i ), master.reject );  
 }  
  
 return master.promise();  
 }  
} );

## jQuery.ready实现

* + - 1. jQuery.ready是基于Deferred实现的

// deferred用于DOM ready  
var readyList = jQuery.Deferred();  
  
jQuery.fn.ready = function( fn ) {  
  
 readyList  
 .then( fn ) // 如果fn出错则用jQuery默认的ready异常处理方式  
  
 // 将 jQuery.readyException 包在函数内；为了错误处理时查找错误，而不是用回调方式  
 .catch( function( error ) {  
 jQuery.readyException( error );  
 } );  
  
 return this;  
};  
  
jQuery.extend( {  
  
 // 是否DOM可以被使用，仅设置true一次  
 isReady: false,  
  
 // 计数器，用于计量ready事件发生前需要等待多少项  
 // #6781  
 readyWait: 1,  
  
 ready: function( wait ) {  
  
 // 如wait  
 if ( wait === true ? --jQuery.readyWait : jQuery.isReady ) {  
 return;  
 }  
  
 // 设置DOM已经准备好的标志 　  
 jQuery.isReady = true;  
  
 // 一直等到readyWait为0  
 if ( wait !== true && --jQuery.readyWait > 0 ) {  
 return;  
 }  
  
 // 执行绑定的延时事件 　　  
 readyList.resolveWith( document, [ jQuery ] );  
 }  
} );  
  
jQuery.ready.then = readyList.then;  
  
// completed方法，删除事件监听，调用ready方法  
function completed() {  
 document.removeEventListener( "DOMContentLoaded", completed );  
 window.removeEventListener( "load", completed );  
 jQuery.ready();  
}  
  
// 处理$(document).ready()在浏览器事件触发之后调用的情况  
// Support: IE <=9 - 10 only  
// 某些老的ie浏览器可能会出现  
if ( document.readyState === "complete" ||  
 ( document.readyState !== "loading" && !document.documentElement.doScroll ) ) {  
  
 // 异步处理，使脚本有机会延迟  
 window.setTimeout( jQuery.ready );  
  
} else {  
  
 // 标准浏览器DOM加载完后会触发“DOMContentLoaded”事件  
 document.addEventListener( "DOMContentLoaded", completed );  
  
 // 如没有DOMContentLoaded事件，则回退监听load事件，这个事件会在图片等都加载完响应  
 window.addEventListener( "load", completed );  
}