# 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)

# 数据缓存

# 队列操作

## 概述

* + - 1. Queue队列是animate动画依赖的基础设施,整个jQuery中队列仅供给动画使用

## 源码分析

jQuery.extend( {  
 queue: function( elem, type, data ) {  
 var queue;  
  
 if ( elem ) {  
 type = ( type || "fx" ) + "queue"; // 专职供fx动画队列处理的  
 // queue对应的key默认为fxqueue,而且默认queue是数组  
 queue = dataPriv.get( elem, type );  
  
 // 如仅是查找，则快速出列  
 if ( data ) {  
 if ( !queue || Array.isArray( data ) ) {  
 // 因为queue返回的为数组，故需要将data转为数组，存在dataPrive的type+queue中  
 queue = dataPriv.access( elem, type, jQuery.makeArray( data ) ); // // 将类数组对象转换为数组对象  
 } else {  
 queue.push( data );  
 }  
 }  
 return queue || [];  
 }  
 },  
 /\*\*  
 \* $.queue(body, 'aa', function(){console.log('aa');});  
 \* $.dequeue(body,'aa');//输出aa  
 \* 出列就有点类似shift的操作，但是不同的是还会执行这个cb1与cb2  
 \* 注意：每调用一次仅出列一个  
 \* @param elem  
 \* @param type  
 \*/  
 dequeue: function( elem, type ) {  
 type = type || "fx";  
  
 var queue = jQuery.queue( elem, type ),// 获取elem上的数组  
 startLength = queue.length,  
 fn = queue.shift(), //拿到queue的头元素  
 hooks = jQuery.\_queueHooks( elem, type ),  
 next = function() {  
 jQuery.dequeue( elem, type );  
 };  
  
 // 第二次调用$.dequeue时，上面fn = queue.shift()会将fn赋值为inprogress，进入if后  
 // 将fn赋值为queue实际的函数  
 // ????????????????????????????????????这个操作有何目的？？？？  
 // 英文注释是避免自动出列  
 if ( fn === "inprogress" ) {  
 fn = queue.shift();  
 startLength--;  
 }  
  
 if ( fn ) {  
  
 // 为fx队列增加一个过程标尺，避免自动出列  
 if ( type === "fx" ) {  
 queue.unshift( "inprogress" ); // 向数组添加的第一个元素。  
 }  
  
 // jQuery.fn.delay 会为函数增加一个stop = function(){window.clearTimeout()}  
 // 删除这个属性  
 delete hooks.stop;  
 fn.call( elem, next, hooks );  
 }  
 // 当队列=0时，删除dataPriv的type+queue  
 if ( !startLength && hooks ) {  
 hooks.empty.fire();  
 }  
 },  
  
 // 非公共方法，产生一个queueHooks对象，或返回现在存在的一个  
 // 在当前元素elem的type + "queueHooks"，创建一个{empty：。。。}  
 \_queueHooks: function( elem, type ) {  
 var key = type + "queueHooks";  
 return dataPriv.get( elem, key ) || dataPriv.access( elem, key, {  
 empty: jQuery.Callbacks( "once memory" ).add( function() {  
 dataPriv.remove( elem, [ type + "queue", key ] );  
 } )  
 } );  
 }  
} );  
  
jQuery.fn.extend( {  
 queue: function( type, data ) {  
 var setter = 2;  
  
 if ( typeof type !== "string" ) {  
 data = type;  
 type = "fx";  
 setter--;  
 }  
 // 如$().queue只有一个参数并且是String，则直接调用$.queue  
 if ( arguments.length < setter ) {  
 return jQuery.queue( this[ 0 ], type );//获取匹配元素的一个,在匹配的第一个上面绑定队列  
 }  
  
 return data === undefined ?  
 this : // $().queue('aaa')情况,返回jquery对象  
 this.each( function() {  
 // 此处的this为jquery的匹配到的每个元素  
 var queue = jQuery.queue( this, type, data );  
  
 // 确保每个queue有一个hooks  
 jQuery.\_queueHooks( this, type );  
 // ?????????????????????????这个地方有问题，type应为函数，否则在dequeue中会有fn.call的调用  
 // 会报错  
 if ( type === "fx" && queue[ 0 ] !== "inprogress" ) {  
 jQuery.dequeue( this, type );  
 }  
 } );  
 },  
 dequeue: function( type ) {  
 return this.each( function() {  
 jQuery.dequeue( this, type );  
 } );  
 },  
 // 清除队列，就是将type类型的数据存储置[]  
 clearQueue: function( type ) {  
 return this.queue( type || "fx", [] );  
 },  
  
 // 获得一个promise resolved，当某个队列的type为空时，默认type为fx  
 promise: function( type, obj ) {  
 var tmp,  
 count = 1,  
 defer = jQuery.Deferred(),  
 elements = this,  
 i = this.length,  
 resolve = function() {  
 if ( !( --count ) ) {  
 defer.resolveWith( elements, [ elements ] );  
 }  
 };  
  
 if ( typeof type !== "string" ) {  
 obj = type;  
 type = undefined;  
 }  
 type = type || "fx";  
  
 while ( i-- ) {  
 tmp = dataPriv.get( elements[ i ], type + "queueHooks" );  
 if ( tmp && tmp.empty ) {  
 count++;  
 tmp.empty.add( resolve );  
 }  
 }  
 resolve();  
 return defer.promise( obj );  
 }  
} );