# 一 jQuery结构

## 1 jQuery结构分析

jQuery对外暴露了两个方法：jQuery 与$，他们实际上是同一个方法，是用来构造jQuery实例对象的工厂函数。

jQuery实例对象是一个伪数组，使用字面量对象来验证：

var div = $('<div></div>');

console.log(({}).toString.call(div)); //输出[object Object]

jQuery的整体结构：

(  
 function (w,j) {  
 *//jQuery源码部分* j(); *//2.0新增参数，用以支持模块化* }(window,function () { return jQuery})  
);

jQuery整体上是一个自调函数，

参数一：window，

参数二：返回值为jQuery的函数（2.0新增参数，用以支持模块化）

我们要创建的my-jQuery框架自己调用即可，可以采用jQuery1.7版本简化为：

(function (w) {  
 function jQuery() {  
  
 }  
 w.jQuery = w.$ = jQuery;  
}(window));

## 2 jQuery初步结构

(function (w) {  
  
 *//暴露的工厂函数* function jQuery() {  
 return new jQuery.fn.init();  
 }  
  
 *//重写原型，fn是为了书写简单* jQuery.fn = jQuery.prototype = function(){};  
  
 *//init是真正的构造函数* var init = jQuery.fn.init = function(){};  
  
 *// 替换构造函数的原型 为 jQuery工厂的原型  
 // 这样可直接使用工厂原型添加方法，以支持插件* init.prototype = jQuery.fn;  
  
 w.jQuery = w.$ = jQuery;  
}(window));

插件测试：

// 实现一个jQ弹出框插件

jQuery.fn.alert = function( msg ) { //直接给工厂原型添加方法，实现弹出插件

alert( msg );

};

// 测试插件

var $$ = $(); //创建jQuery实例

$$.alert('弹出框插件'); //实例可以直接调用原型方法

# 二 jQuery功能分析

## 1 jQuery入口函数结构

通过$工厂，最终达到init函数，这个函数即是入口函数，所有的初始化实例过程都是在入口函数内实现的。

依据不同的参数，入口的返回值也不同。

### 1.1传入null、undefined、0、Nan、’’

都会返回一个空实例，如：

console.log($());

console.log($(null));

### 1.2 传入选择器

会获取到元素，并把下表存储，同时添加一个length属性代表数量

// { 0:div, 1:div, length:2 }

console.log($('div'));

### 1.3 传入html片段

会根据html片段创建DOM对象，然后按照下标进行存储，添加一个length属性。

// { 0:span, 1:span, 2,span, length:3 }

console.log($('<span></span>'));

// 测试JQ传入html片段，得到是Dom对象

document.body.appendChild( $('<span>span</span>')[0] );

### 1.4 传入数组

会把数组中的每一项分别添加到实例身上，同时添加length

console.log($(['abc', 'cba']));

console.log($(['1', '2', '3']));

### 1.5 传入伪数组

会把数组中的每一项分别添加到到实例身上，同时添加length

var as = document.querySelectorAll('a');

console.log($(as));

### 1.6 传入对象

会直接把这个对象按照下表添加到实例身上，同时添加length

console.log($( new Date() ));

console.log($( { val: 0000 } ));

## 2 jQuery参数处理规律

\* jq入口对不同参数处理的规律：

\* 1、传入null、undefined、0、NaN、''返回空对象( 即空实例 )

\* 2、传入字符串，那么需要判断是html片段 还是 其它，

\* 如果是片段，则创建对应的DOM，然后添加到实例身上；

\* 否则按照选择器获取页面中的DOM，然后把获取到的DOM添加到实例身上。

\* 3、如果是数组或许伪数组，那么把每一项分别添加到实例身上。

\* 4、除了上面的数据类型，剩余的，统一添加到实例身上。

## 3 jQuery入口实现思路

function init( selector ) {

// 传入null、undefined、0、NaN、''返回空对象( 即空实例 )

if ( !selector ) {

return this;

}

// 传入字符串，那么需要判断是html片段 还是 其它

else if ( typeof selector == 'string' ) {

// 如果是片段，则创建对应的DOM，然后添加到实例身上，

// 如果第一个字母是<,最后一个字母是>,并且length >= 3,则是html片段。

if ( ) {

/\*

\* 创建的思路：

\* 1、先创建一个临时的div容器

\* 2、设置这个div的innerHTML为html片段，

\* 这些片段就自动转变为了div的子元素，

\* 3、然后遍历div的子元素分别添加this身上，记住给实例补充length属性值，

\* 可以使用数组的push来给实例添加，同时可以使用apply简化遍历过程。

\* \*/

}

// 否则按照选择器获取页面中的DOM，然后把获取到的DOM添加到实例身上

else {

/\*

\* 实现的思路：

\* 1、使用querySelectorAll获取页面中的元素

\* 2、然后遍历获取到所有元素分别添加this身上，记住给实例补充length属性值，

\* 可以使用数组的push来给实例添加，同时可以使用apply简化遍历过程。

\* \*/

}

}

/\*

\* 判断是不是真假数组的思路：

\* 1、先把函数和window排除掉，

\* 2、然后通过toString来判断是不是真数组

\* 3、否则再判断是不是伪数组

\* 备注：下面的判断只用来判断是不是伪数组，不要用下面的判断条件判断真数组，

\* 例如：[ 0:1, , , ] 使用下面的判断，就会得到false，造成真数组的误判。

\* 3.1、 先看看这个对象有没有length属性，

\* 3.2、 如果有，看看length的值是不是为0，如果为0，OK是伪数组，

\* 3.3、 如果length的值不为0，看看这个数据有没有 length - 1这个属性，如果有，OK是伪数组。

\* 建议把这个是否是真假数组的判断封装为一个函数，在这里调用。

\* \*/

else if ( ) {

/\*

\* 实现的思路：

\* 把真或伪数组中的每一项分别添加到实例身上，记住给实例补充length属性值，

\* 可以使用数组的push来给实例添加，同时可以使用apply简化遍历过程。

\* \*/

}

else {

/\*

\* 实现的思路：

\* 把这个参数直接添加到实例身子，length为1即可。

\* \*/

}

}

console.log(new init(null));;

# 三 jQuery入口函数实现

### 3.1 字符串、HTML片段判断

// 判断是不是html片段

init.isHTML = function (html) {

// 如果是空类型的，直接返回false

if (!html) {

return false;

}

// 如果字符串的第一个字母是<,最后一个字母是>,且length >= 3,则是html片段

if (html.charAt(0) == '<' && html.charAt(html.length - 1) == '>' && html.length >= 3) {

return true;

}

return false;

};

// 去掉字符串首尾空白字符

init.trim = function (str) {

// 不是字符串就不处理了

if (typeof str !== 'string') {

return str;

}

// 优先使用内置的trim方法

if (str.trim) {

return str.trim();

}

// 把首尾空白字符替换为空，然后返回

return str.replace(/^\s+|\s+$/g, '');

};

### 3.2 函数、window判断

// 判断是不是函数

init.isFunction = function (fn) {

if (typeof fn == 'function') {

return true;

}

return false;

},

// 判断是不是window

init.isWindow = function (w) {

if (w.window == w) {

return true;

}

return false;

},

### 3.3 伪数组、数组判断

// 判断是不是伪数组或真数组

init.isLikeArray = function( arr ) {

// 过滤函数和window，以及非对象的其他数据

if ( init.isFunction( arr ) || init.isWindow( arr ) || typeof arr !== 'object' ) {

return false;

}

// 判断是不是真数组

if ( ({}).toString.call( arr ) == '[object Array]' ) {

return true;

}

// 判断是不是伪数组：有length，且要么length为0，要么有length - 1这个属性值

if ( ('length' in arr) && ( (arr.length === 0) || (arr.length - 1 in arr) ) ) {

return true;

}

return false;

};

### 3.4 使用extend

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**> <**script**>  (**function**( w ) {  **function** *jQuery*( selector ) {  **return new** *jQuery*.**fn**.init( selector );  }  *jQuery*.**fn** = *jQuery*.**prototype** = { *// 替换原型 + 原型简称* constructor: *jQuery* }   *// 给jQuery和原型分别添加extend方法  jQuery*.extend = *jQuery*.**fn**.extend = **function**( obj ) {  **for** ( **var** key in obj ) {  **this**[ key ] = obj[ key ];  }  }  *jQuery*.extend({ *// 给jQuery添加一些静态方法* trim: **function**( str ) { *// 去掉首尾空白字符* **if** ( !str ) {  **return** str;  }  **if** ( str.trim ) {  **return** str.trim();  }  **return** str.replace( /^\s+|\s+$/g, **''**);   },   isHTML: **function**( html ) { *// 判断是不是html片段* **if** ( !html ) {  **return false**;  }  **if**( html.charAt(0) === **'<'** &&  html.charAt( html.**length** - 1 ) === **'>'** &&  html.**length** >= 3 ) {  **return true**;  }  **return false**;  },   \_isHTML: **function**( html ) { *// 判断是不是html片段* **return** !!html &&  html.charAt(0) === **'<'** &&  html.charAt( html.**length** - 1 ) === **'>'** &&  html.**length** >= 3;  },   isFunction: **function**( fn ) { *// 判断是不是函数* **if** ( typeof fn === **'function'** ) {  **return true**;  }  **return false**;  },   \_isFunction: **function**( fn ) { *// 判断是不是函数* **return** typeof fn === **'function'**;  },   isWindow: **function**( w ) { *// 判断是不是window* **if**( !w ) {  **return false**;  }  **if**( w.**window** === w ) {  **return true**;  }  **return false**;  },   isObject: **function**( obj ) { *// 判断是不是对象* **if** ( obj === **null** ) { *// 防止typeof对null的误判* **return false**;  }  *// 如果是object或function，那就是对象* **if** ( typeof obj === **'object'** || typeof obj === **'function'** ) {  **return true**;  }  **return false**;  },    \_isWindow: **function**( w ) { *// 判断是不是window* **return** !!w && w.**window** === w;  },    isString: **function**( str ) { *// 判断是不是字符串* **if** ( typeof str === **'string'** ) {  **return true**;  }  **return false**;  },    \_isString: **function**( str ) { *// 判断是不是字符串* **return** typeof str === **'string'**;  },    isLikeArray: **function**( arr ) { *// 判断是不是真数组或伪数组  // Function、window、!Object* **if** ( *jQuery*.isFunction( arr ) || *jQuery*.isWindow( arr ) || !*jQuery*.isObject( arr ) ) {  **return false**;  }  *// 判断是不是真数组* **if** ( ({}).toString.call( arr ) === **'[object Array]'** ) {  **return true**;  }  *// 判断是不是伪数组* **if** ( **'length'** in arr && ( arr.**length** === 0 || arr.**length** - 1 in arr ) ) {  **return true**;  }  **return false**;  }  });    **var** init = *jQuery*.**fn**.init = **function**( selector ) {    **if** ( !selector ) {  **return this**;  }   *// string ==> ( html || selector )* **if**( *jQuery*.isString( selector ) ) {    *// 为了用户友好体验，先去掉首尾空白字符* selector = *jQuery*.trim( selector );   *// html* **if**( *jQuery*.isHTML( selector ) ) {   *// 利用一个临时的div来创建DOM，  // 然后把创建好的DOM依次push给实例。* **var** tempDiv = **document**.createElement( **'div'** );  tempDiv.**innerHTML** = selector;  [].push.apply( **this**, tempDiv.**childNodes** );   }   *// selector* **else** {   **try** {  [].push.apply( **this**, **document**.querySelectorAll( selector ) );  }**catch**(e) {  *// 如果报错了，那么手动补一个length属性，代表没有获取到任何元素* **this**.**length** = 0;  }  }  }   *// array || likeArray* **else if**( *jQuery*.isLikeArray( selector ) ) {  [].push.apply( **this**, selector );  }   *// 其它* **else** {  **this**[0] = selector;  **this**.**length** = 1;  }  };   *// 替换init的原型为工厂的原型，这样外界就可以通过工厂给实例扩展方法* init.**prototype** = *jQuery*.**fn**;   *// 暴露工厂和工厂的简称* w.**jQuery** = w.$ = *jQuery*;   }( **window** ));   *// 静态方法测试* **console**.log(*$*.\_isFunction([]));  **console**.log(*$*.isWindow( **window** ));  **console**.log(*$*.isLikeArray( { **length**: 5 } ));  **console**.log(*$*.isLikeArray( { **length**: 0 } ));   *// 入口测试* **console**.log(*$*([1, 2]));  **console**.log(*$*(**' script'**));  **console**.log(*$*(**' <span>111</span><span>222</span> '**));  **console**.log(*$*({ 0:**document**.**body**, **length**:1 })); </**script**> </**body**> </**html**> |

### 3.5 入口函数参数处理理论

<script>

// 为了防止获取不到页面的元素，所以要把函数传入到jQuery，

// 然后在这个回调函数中编写代码。

// 这个回调函数会在页面DOM解析完毕之后执行。

$(function() {

// 可以获取到

console.log($('span'));

});

// 获取不到span，因为这个时候span还不存在呢

console.log($('span'));

</script>

<div>

<span>123</span>

<span>456</span>

</div>

### 3.6 jQuery入口函数参数处理

|  |
| --- |
| ready: **function**( fn ) {   *// 先统一判断DOMContentloaded有没有触发，  // 通过document.readyState === 'complete'判断  // 如果为true，fn可以直接调用。   // 如果为false，那么判断支不支持addEventListener，  // 如果支持，绑定DOMContentLoaded事件   // 如果不支持，使用attchEvent绑定onreadystatechang事件,  // 注意，需要在里面判断document.readyState === 'complete'才执行fn。  // 防止fn多次执行。   // DOM已经构造完毕，fn可以直接执行* **if** ( **document**.**readyState** === **'complete'** ) {  fn();  }   *// 如果DOM没有构造完毕，那么判断addEventListener是否兼容* **else if**( **document**.addEventListener ) {  **document**.addEventListener( **'DOMContentLoaded'**, fn );  }   *// 如果不兼容addEventListener，那么采取attachEvent的方式，  // 同时事件变为了onreadystatechange，为了防止这个事件多次触发造成的fn多次执行，  // 所以需要一个包装函数来进行过滤。* **else** {  **document**.attachEvent( **'onreadystatechange'**, **function**() {  **if**( **document**.**readyState** === **'complete'** ) {  fn();  }  } );  } } |

## 3 原型方法

|  |
| --- |
| constructor: *jQuery*,  *// 获取版本号* **jquery**: version,  *// 代表所有实例默认的选择器，也代表实例是一个jQuery类型的对象* **selector**: **''**,  *// 代表所有实例默认的长度* **length**: 0,  *// 把实例转换为数组返回* toArray: **function**() {  **return** [].slice.call( **this** ); },  *// 获取指定下标的元素，获取的是原生DOM* get: **function**( i ) {  */\*  \* 1、如果传入null或undefined，那么转换为数组返回  \* 2、如果传入的是正数，按照指定的下标获取元素返回  \* 3、如果传入的是负数，按照下标倒着( this.length + 负数 )获取元素返回  \* \*/   // null、undeinfed* **if** ( i == **null** ) {  **return this**.toArray();  }   *// 其他* **if** ( i >= 0 ) {  **return this**[ i ];  }**else** {  **return this**[ **this**.**length** + i ];  } },  \_get: **function**( i ) {  **return** i == **null**?  **this**.toArray() :  ( i >= 0? **this**[ i ] : **this**[ **this**.**length** + i ] ); },  each: **function**() {  return jQuery.each(this,fn); }, map: **function**() {  return jQuery.map(this,fn); },  *// 截取实例的部分元素，构成一个新的jQuery实例返回* slice: **function**() {  */\*  \* 1、通过数组的slice截取部分元素(slice返回的是数组)，  \* 2、把截取到的元素转换为实例对象返回。  \* \*/   // 因为slice的参数会有变化，所以需要是arguments，  // 我们要把arguments中的每一项传给数组的slice，所以需要借用apply平铺传递过去，  // 最后把slice返回数组，通过jQuery工厂保证成实例返回。* **var** nodes = [].slice.apply( **this**, arguments );  **return** *jQuery*( nodes ); },  \_slice: **function**() {  *// 因为slice的参数会有变化，所以需要是arguments，  // 我们要把arguments中的每一项传给数组的slice，所以需要借用apply平铺传递过去，  // 最后把slice返回数组，通过jQuery工厂保证成实例返回。* **return** *jQuery*( [].slice.apply( **this**, arguments ) ); },  *// 获取指定下标的元素，获取的是jQuery类型的实例对象。* eq: **function**( i ) {  */\*  \* 1、如果传入null或undefined，返回一个新实例，  \* 2、如果传入的是正数，按照指定的下标获取元素，再包装成新实例返回  \* 3、如果传入的是负数，按照下标倒着( this.length + 负数 )获取元素，再包装成新实例返回  \* \*/   // null、undefined得到新实例* **if**( i == **null** ) {  **return** *jQuery*();  }   **if**( i >= 0 ) {  **return** *jQuery*( **this**[ i ] );  }**else** {  **return** *jQuery*( **this**[ **this**.**length** + i ] );  } },  \_eq: **function**( i ) {  **return** i == **null**? *jQuery*() : *jQuery*( **this**.get( i ) ); },  *// 获取实例中的第一个元素，是jQuery类型的实例对象。* first: **function**() {  **return this**.eq( 0 ); },  *// 获取实例中的最后一个元素，是jQuery类型的实例对象。* last: **function**() {  **return this**.eq( -1 ); },  *// 原型上的方法供实例调用， // 即实例.push，在调用过程中，push内的this就指向了实例， // 所以这里不需要通过call和apply改变this指向即可借用数组的方法* **push**: [].push, **sort**: [].sort, **splice**: [].splice |

### 3.1

### 3.2