## 2 知识补充

### 2.1 代码块

代码块就是 {}

如果将代码块赋值给了一个变量，或者参与了运算，就变成了 字面量。

{

var a = 1;

console.log(a);

}

// 需要通过运算，把{}转换为字面量，才能调用方法

console.log(({}).toString()); // '[object Object]'

// 错误，代码块不能调用方法

// {}.toString();

### 2.2 原型补充

/\*

\* Object.prototype.toString

\* 作用：根据内部的this返回一个类似于这样的字符串'[object constructorName]'

\* 这个方法有个缺点，不能获取用户自定义对象的具体类型，只能获取内置对象的类型。

\* \*/

// 测试

console.log(Object.prototype.toString.call([])); //[object Array]

console.log(Object.prototype.toString.call( new Date() ));

console.log(Object.prototype.toString.call( Array )); //[object Function]

console.log(Object.prototype.toString.call( Date ));

// 简写 ==> 因为{}对象直接继承Object.prototype，

// 所以通过这个对象得到的toString，一定来自Object.prototype。

console.log(({}).toString.call([]));

console.log(({}).toString.call( new Date() ));

console.log(({}).toString.call( Array ));

console.log(({}).toString.call( Date ));

// 用这个toString获取自定义对象的类型

function Person(){}

var xiaofang = new Person();

console.log(({}).toString.call( xiaofang ));

### 2.3 toString

// 字符串 ==> String.prototype.toString

var a="ddd";

console.log(a.toString());

var e= new String();

console.log(e.toString());

var e= new String('string');

console.log(e.toString());

// 数组 ==> Array.prototype.toString

var b=[1,2,3,4,5,6];

console.log(b.toString());

var b=[];

console.log(b.toString());

var d=new Array();

console.log(d.toString());

// 对象 ==> Object.prototype.toString

var c={};

console.log(c.toString());

// 函数 ==> Function.prototype.toString

console.log(Function.toString());

console.log(Array.toString());

console.log(RegExp.toString());

function fn() {

console.log('我写的');

}

console.log(fn.toString());

### 2.4 iframe

一个iframe就是一个页面，可以获得页面数量：

console.log( window.length );

### 2.5 window

// window对象有一个特点，就是window有一个window属性，指向自己

// 如果一个数据的window属性是自己，那么就认为它是window。

console.log( obj.window === obj );

### 2.6 伪数组

/\*

\* 伪数组的特点：

\* 1、必须要有length属性

\* 2、如果length属性值为0，那么这个对象有没有元素无所谓

\* 3、如果length属性值不为0，那么这个对象一定有 (length - 1) 为下标的属性值。

\* \*/

// 是伪装组

var obj = { length: 0 };

var obj = { length: 4, 3: 'a' }; // 没有0，1，2可以认为0，1，2存的是undefined

var obj = { length: 2, 0: 'a', 1: 'b' };

// 不是伪数组，因为没有 9(length - 1) 这个属性

var obj = { length: 10, 0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f' };

### 2.7 数组的分类

\* 数组的分类：

\*

\* 1、真数组

\* new Array 创建的数组，或者字面量形式 [] 创建的数组，都是真数组。

\* 1.1 稀疏数组

\* 数组中，某些下标不存在，即为稀疏数组。

\* 1.2 非稀疏数组

\* 数组中没有缺失任何值。

\*

\* 2、伪数组

\* 必须要有length属性，

\* 要么length为0，要么有length - 1为下标的属性。

### 2.8 数组方法借用

// 借用数组的push方法给obj按照下标的方式添加属性值，并且会自动维护lenghth属性

var obj = {};

[].push.apply( obj, [ 1, 2, 3, 4 ] );

console.log( obj );

// 借用数组的pop方法删除obj最后一个属性值

[].pop.call( obj );

console.log( obj );

### 2.9 trim方法

/\*

\* trim方法是ES5提供的字符串方法，用来去除字符串首尾空白字符，但是IE9+才能兼容。

\* \*/

function trim( 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, '' );

}

### 2.10 函数分类

\* 在js中，函数是一等公民，最强大。

\*

\* 普通函数：

\* 直接定义或者声明的函数。

\* var fn = function() {};

\* function fn() {}

\*

\* 方法：

\* 如果一个函数有了宿主，那么这个函数称作宿主的方法。

\* var obj = {

\* fn: function() {}

\* }

\*

\* 构造函数：

\* 如果一个函数，被通过new用来创建对象，

\* 那么这个函数又有了一个新的称呼，叫构造函数。

\*

\* 回调函数：

\* 如果一个函数，被作为参数传递给另外一个函数，

\* 那么这个函数(指的是被传递的函数)就可以称之为回调函数。

### 2.11 实例方法与静态方法

/\*

\* 实例成员：

\* 提供给实例使用的方法和属性，就称之为实例成员。

\*

\* 实例方法：

\* 提供给实例使用的方法，就称之为实例方法。

\*

\* 静态成员（类成员）：

\* 提供给函数使用的方法和属性，就称之为静态成员。

\*

\* 静态方法（类方法）：

\* 提供给函数使用的方法，就称之为静态方法。

\*

\* 类：

\* 在ES6之前，通常我们会把构造函数当做是类。

\* \*/

//

function Person( name, age, MAX\_AGE ) {

/\*

\* 这里通过this添加的实例成员，

\* 都是添加到了实例自身上。

\* \*/

// 这是实例成员，因为这是供实例使用的。

this.name = name;

// 这是实例成员，因为这是供实例使用的。

this.age = age;

// 这是实例成员，因为这是供实例使用的。

// 实例成员的作用面比较窄，

// 这里代表某个人最大的年龄

this.MAX\_AGE = MAX\_AGE;

}

// 这是静态成员，因为这是添加给函数使用的。

// 静态成员的作用面更广，代表的含义也更广。

// 我们这里代表整个人类最大的年龄

Person.MAX\_AGE = 300;

Person.prototype = {

constructor: Person,

// 这是实例成员，因为这是供实例使用的。

// 但是这里的属性和方法，并没有添加到实例自身。

run: function( ) {

// 为了减少冗余，复用静态方法

Person.run( this.age );

}

}

// 如果一些方法，通用性已经很强了，

// 那么就可以考虑把他加到函数中作为静态方法使用。

Person.run = function( age ) {

if ( age < 5 ) {

console.log( '趴着跑' );

}else if ( age >= 5 && age <= 16 ) {

console.log( '跳着跑' );

}else if ( age > 16 && age < 30 ) {

console.log( '跑酷' );

}else {

console.log( '拄着拐杖跑' );

}

}

/\*

\* 为什么都是实例方法复用静态方法：

\* 因为。静态方法服务的面更广，即可以服务实例，

\* 也可以服务其他的地方，所以实例方法可以复用静态方法( 即实例方法中可以使用静态方法解决问题 )。

\*

\* 而实例方法只服务于实例，没法服务与更多的地方，

\* 所以静态方法无法复用实例方法( 即静态方法中无法使用实例方法解决问题 )。

\*

\* \*/

### 2.12 实例方法与静态方法的使用

function Person( name, MAX\_AGE ) {

// 这是实例成员

this.name = name;

// 这是实例成员

this.MAX\_AGE = MAX\_AGE;

}

// 这是静态成员

Person.MAX\_AGE = 300;

Person.prototype = {

// 这是实例成员

run: function( ) {

Person.run( this.age );

}

}

// 这是静态成员

Person.say = function() {

console.log( '说' );

}

// 这是静态成员

Person.run = function( age ) {

if ( age < 5 ) {

console.log( '趴着跑' );

}else if ( age >= 5 && age <= 16 ) {

console.log( '跳着跑' );

}else if ( age > 16 && age < 30 ) {

console.log( '跑酷' );

}else {

console.log( '拄着拐杖跑' );

}

}

// 创建一个实例

var bingjie = new Person( '冰冰姐', 100 );

console.log( bingjie.MAX\_AGE ); // 100

// 实例不能直接使用静态方法。

bingjie.say();

// 因为实例不能直接使用静态方法，

// 所以通常会根据静态方法再实现一个实例方法，供实例使用，

// 在实例方法的实现中，实际上是利用静态方法实现的。

bingjie.run();

// 找到的自身上的MAX\_AGE，所以为300

console.log(Person.MAX\_AGE); // 300

person.say(); // 静态方法必须要通过函数来使用

/\*

\* 实例不能使用静态方法，

\* 构造函数不能使用实例方法。

\* \*/

### 2.13 显示原型和隐式原型

/\*

\* 在js中，有两个属性和原型对象相关：

\* 这两个属性，都指定原型对象，

\* 为了却分，所以给这两个属性起了不同的名字。

\* 1、prototype ( 显示原型、原型属性 )

\* 2、\_\_proto\_\_ ( 隐式原型、原型对象 )

\* \*/

// 原型对象：就是实例继承的那些对象。

### 2.14 extend

function Person() {

}

// 这里是extend 即时静态方法，又是实例方法

Person.extend = Person.prototype.extend = function( obj ) {

for ( var key in obj ) {

this[ key ] = obj[ key ];

}

};

/\*

\* 预期，谁调用extend，就给谁混入内容；

\* 那么这就和this相关了，内置的实现就要使用this。

\* \*/

// 预期给Person自己添加静态成员

Person.extend({

});

// 预期给Person的实例自己添加实例成员

Person.prototype.extend({

});

### 2.15 数组方法与借用

/\*

\* slice为什么单独做了处理：

\* slice的返回值是一个数组，jQ预期我们需要继续对这个数组进行进一步操作，

\* 所以把这个数组又包装成新的实例对象了（这样可以调用Jq提供的那些方法）。

\* 1、因为JQ处理了

\* 2、处理的好处是可以链式编程

\* \*/

$('span').slice(0, 2).css({

backgroundColor: 'blue',

display: 'block',

width: '50px',

height: '10px'

});

/\*

\* push没有做处理：

\* push的返回值是一个新的长度，jQ可能为了不增加我们的负担，

\* 所以沿用了push的返回值。

\* 1、因为JQ没有处理

\* \*/

console.log( 'push之后新的长度为:' + $('span').push(document.body) );

/\*

\* sort没有做处理：

\* sort谁调用返回谁，如果是实例调用，那么返回的还是这个实例，

\* 本身就不用做额外的处理，所以沿用了sort的返回值。

\* 1、因为JQ没有处理

\* \*/

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

// sort返回值是一个实例，不用做额外处理

$('span').sort().attr( 'sort', 'OK' );

/\*

\* splice没有做处理：

\* splice的返回值是被删除掉的元素，可能JQ认为这个被删除的元素，

\* 已经没有价值了，包装也没啥用，所以沿用了splice的返回值。

\* 1、因为JQ没有处理

\* \*/

console.log($('span').splice( 0, 3 ));

### 2.16 each

<div>

<span>1111</span>

<span>2222</span>

<span>3333</span>

<span>4444</span>

<span>5555</span>

</div>

<script>

var str = '';

$('span').each( function() {

// 这里面的this指向遍历到的每一个span

// 如果遇到内容4444，那么就不加了

if ( this.innerHTML === '4444' ) {

return false;

}

str += this.innerHTML;

});

console.log( str );

</script>

eache实现

<div>

<span>1111</span>

<span>2222</span>

<span>3333</span>

<span>4444</span>

<span>5555</span>

</div>

<script>

// 实现each方法

function each( obj, fn ) {

// 如果是数组或伪数组，按照顺序遍历

if ( 'length' in obj ) {

for ( var i = 0, len = obj.length; i < len; i++ ) {

// 改变回调执行时内部的this为val，

// 通过根据回调的返回值看看要不要中断遍历。

if ( fn.call( obj[ i ], i, obj[ i ] ) === false ) {

break;

}

}

}else {

for ( var key in obj ) {

// 改变回调执行时内部的this为val，

// 通过根据回调的返回值看看要不要中断遍历。

if ( fn.call( obj[ key ], key, obj[ key ] ) === false ) {

break;

}

}

}

}

var arr = [1,2,3];

each( arr, function( i, val ) {

console.log( i, val );

} );

</script>

### 2.17 异常捕获

/\*

\* 异常捕获

\* \*/

try {

console.log(a);

}catch( e ) {

console.log(8);

}finally {

console.log('我一定要执行');

}

console.log(111);

/\*

\* 异常抛出

\* \*/

function fn( obj ) {

if ( !obj ) {

throw '参数错误！';

}

}

fn();

### 2.18 extend给类扩展方法

function jQuery(){

}

//传入的obj，谁调用就给谁混入方法

jQuery.extend = jQuery.prototype.extend = function(obj){

for (var key in obj){

this[key] = obj[key];

}

}

//添加静态方法

jQuery.extend({

isFunction: function(fn){ //假设加入一个方法

return typeof fn === 'function';

}

});

//添加实例方法

jQuery.prototype.extend({

alert: function(msg){

alert(msg);

}

});

//应用

console.log(jQuery.isFunction([]));

(new jQuery()).alert("test");

### 2.19 IE8下的apply

/\*

\* apply方法可以该改变this指向，同时可以把数组或伪数组平铺传入给函数，

\* 但是IE8，apply只能平铺真数组或者内置的伪数组，我们自定义的伪数组会报错。

\* \*/

var obj = { 0: 'abc', 1: 'cba', length: 2 };

var o = {};

// IE8中需要先把自定义伪数组转换为真数组，才能借用apply

/\*var arr = [];

for ( var i = 0, len = obj.length; i < len; i++ ) {

arr.push( obj[ i ] );

}

[].push.apply( o, arr );\*/

/\*-------------------------------简化版本--------------------------------\*/

// 借用数组的slice方法，通过一个伪数组得到一个真数组

// console.log( [].slice.call( obj ) );

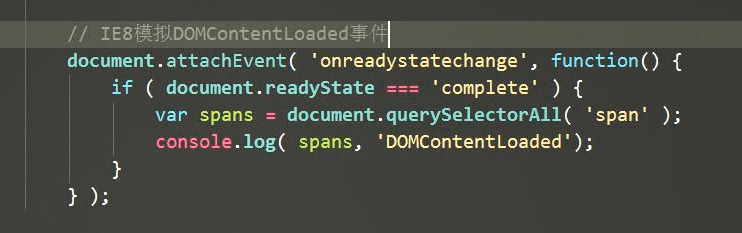
// 简写

[].push.apply( o, [].slice.call( obj ) );

### 2.20 jQuery调用机制以及DOMContentLoaded事件







### 2.21 jQuery原型核心方法简介

|  |
| --- |
| <script>  /\*  \* jQ原型上的核心方法和属性：  \* 1、jquery 获取版本号  \* 2、selector 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  \* 3、length 代表所有实例默认的长度  \* 4、toArray 把实例转换为数组返回  \* 5、get 获取指定下标的元素，获取的是原生DOM  \* 6、each 遍历实例，把遍历到的数据分别传给回调使用  \* 7、map 遍历实例，把遍历到的数据分别传给回调使用，然后把回调的返回值收集起来组成一个数组返回  \* 8、slice 截取实例的部分元素，构成一个新的jQuery实例返回。  \* 9、first 获取实例中的第一个元素，是jQuery类型的实例对象。  \* 10、last 获取实例中的最后一个元素，是jQuery类型的实例对象。  \* 11、eq 获取指定下标的元素，获取的是jQuery类型的实例对象。  \* 12、push 给实例添加新元素  \* 13、sort 对实例中的元素进行排序  \* 14、splice 按照指定下标指定数量删除元素，也可以替换删除的元素。  \* \*/  console.log($().jquery);  console.log($().selector);  console.log($().length);  console.log($('li').toArray());  // get 不传相当于toArray，  // 传参获取指定下标获取元素，可以传负数，获取到的是原生DOM  console.log($('li').get());  console.log($('li').get(0));  console.log($('li').get(-1));  // 使用方式和数组的slice一样，但是得到是一个新的实例  console.log($('li').slice( 0, 2 ));  // get 不传相当于得到一个新实例，  // 传参获取指定下标获取元素，可以传负数，获取到的元素是新的实例  console.log($('li').eq());  console.log($('li').eq(0));  console.log($('li').eq(-2));  // 获取第一个和最后一个元素，得到的是新的实例  console.log($('li').first());  console.log($('li').last());  // 给实例添加一个新元素  var $$ = $();  $$.push( document.body );  console.log( $$ );  // 给实例排序  /\*var $lis = $('li');  $lis.sort(function( li1, li2 ) {  return li1.innerHTML > li2.innerHTML;  });  console.log( $lis );  $lis.appendTo( document.body );\*/  // 删除或替换  var $lis = $('li');  $lis.splice( 1, 1, document.body );  console.log( $lis );  </script> |

### 2.22 jQuery的each

1 简介

<script>

var obj = { a: 111, b: 222, c: 333, d: 444 };

var arr = [ 'aaa', 'bbb', 'ccc', 'dddd' ];

/\*

\* jQ的each方法，变量对象或者数组时，

\* 下标和值传入回调的顺序和数组的forEach相反。

\* \*/

/\*$.each( obj, function( key, val ) {

console.log( key, val );

} );

$.each( arr, function( index, val ) {

console.log( index, val );

} );\*/

// 这是数组的forEach

/\*arr.forEach( function( val, index ) {

console.log( val, index );

});\*/

function each( obj, fn ) {

/\*

\* 1、先判断obj是不是数组或者伪数组，

\* 2、如果是，则通过i的方式遍历这个对象

\* 3、如果不是，则通过for in的方式遍历这个对象

\* 4、在遍历的过程中，把每一次遍历到key和val分别传给回调。

\* \*/

if( 'length' in obj ) {

for ( var i = 0, len = obj.length; i < len; i++ ) {

fn( i, obj[ i ] );

}

}else {

for ( var key in obj ) {

fn( key, obj[ key ] );

}

}

}

each( obj, function( key, val ) {

console.log( key, val );

} );

each( arr, function( index, val ) {

console.log( index, val );

} );

</script>

2 each中的this

<script>

var obj = { a: {val:1}, b: 222, c: 333, d: 444 };

/\*$.each( obj, function( key, val ) {

console.log( key, val );

console.log( this ); // this指向val

} );\*/

function each( obj, fn ) {

/\*

\* 1、先判断obj是不是数组或者伪数组，

\* 2、如果是，则通过i的方式遍历这个对象

\* 3、如果不是，则通过for in的方式遍历这个对象

\* 4、在遍历的过程中，把每一次遍历到key和val分别传给回调。

\* \*/

if( 'length' in obj ) {

for ( var i = 0, len = obj.length; i < len; i++ ) {

// 指定fn执行时，this指向val

fn.call( obj[ i ], i, obj[ i ] );

}

}else {

for ( var key in obj ) {

// 指定fn执行时，this指向val

fn.call( obj[ key ], key, obj[ key ] );

}

}

}

each( obj, function( key, val ) {

console.log( key, val, this );

} );

</script>

3 each遍历中断

<script>

var obj = { a: {val:1}, b: 222, c: 333, d: 444 };

// 中断遍历

/\*$.each( obj, function( key, val ) {

// 当得到val为222这个的属性时，就不用再遍历之后的属性了

if ( val == 333 ) {

return false;

}

console.log( key, val );

} );\*/

function each( obj, fn ) {

/\*

\* 1、先判断obj是不是数组或者伪数组，

\* 2、如果是，则通过i的方式遍历这个对象

\* 3、如果不是，则通过for in的方式遍历这个对象

\* 4、在遍历的过程中，把每一次遍历到key和val分别传给回调。

\* \*/

if( 'length' in obj ) {

for ( var i = 0, len = obj.length; i < len; i++ ) {

// 指定fn执行时，this指向val

if ( fn.call( obj[ i ], i, obj[ i ] ) === false ){

break;

}

}

}else {

for ( var key in obj ) {

// 如果fn的返回值为false，那么就break，跳出循环

if ( fn.call( obj[ key ], key, obj[ key ] ) === false ) {

break;

}

}

}

}

each( obj, function( key, val ) {

// 当得到val为222这个的属性时，就不用再遍历之后的属性了

if ( val == 333 ) {

return false;

}

console.log( key, val );

} );

</script>

4 each实现

<script>

var obj = { a: {val:1}, b: 222, c: 333, d: 444 };

function each( obj, fn ) {

var i, len, key;

// likeArray

if ( 'length' in obj ) {

for ( i = 0, len = obj.length; i < len; i++ ) {

if ( fn.call( obj[ i ], i, obj[ i ] ) === false ) {

break;

}

}

}else {

for ( key in obj ) {

if ( fn.call( obj[ key ], key, obj[ key ] ) === false ) {

break;

}

}

}

}

</script>

### 2.23 jQuery的map

var obj = { a: 111, b: 222, c: 333, d: 444 };

var arr = [ 'aaa', 'bbb', 'ccc', 'dddd' ];

/\*

\* jQ的map方法，用来遍历对象或者数组，

\* 然后把遍历到了val和key值传给回调，

\* 最后收集回调的返回值，构成新数组返回。

\* \*/

/\*console.log($.map(obj, function (val, key) {

console.log(val, key, this);

return val;

}));\*/

// map实现

function map( obj, fn ) {

/\*

\* 1、先判断obj是不是数组或者伪数组，

\* 2、如果是，则通过i的方式遍历这个对象

\* 3、如果不是，则通过for in的方式遍历这个对象

\* 4、在遍历的过程中，把每一次遍历到key和val分别传给回调。

\* 5、在给回调传参的时候，需要收集回调的返回值，最后把所有的返回值构成新数组返回。

\* \*/

var i, len, key, result = [];

if( 'length' in obj ) {

for ( i = 0, len = obj.length; i < len; i++ ) {

result.push( fn.call( obj[ i ], obj[ i ], i ) );

}

}else {

for ( key in obj ) {

result.push( fn.call( obj[ key ], obj[ key ], key ) );

}

}

return result;

}

console.log(map(obj, function (val, key) {

console.log(val, key, this);

// return val / 100;

}));