**Es5规范：**

**严格模式**

在JS文件或是函数的顶部添加"use strict"即可启用严格模式。因为"use strict"就是个字符串，因此其会被旧版浏览器安全地忽视。[strict](strict.html)

function strict(){

"use strict";

//...

}

**JSON方法**

JSON.parse(text [, reviver])

>> var result = JSON.parse('{"a": 1, "b": "2"}');

Object

>> result.b

"2"

如果我们想确保解析的值是个整数，我们可以使用reviver方法。返回undefined就删除这一项。

var result = JSON.parse('{"a": 1, "b": "2"}', function(key, value){

if (typeof value == 'string'){

return parseInt(value);

} else {//一定要有这个else，最后一次的value是解析后的json对象，不返回则result被赋值为undefined。

return value;

}

})

>> result.b

2

JSON.stringify(value [, replacer [, space]])

如果我们需要改变值字符串化的方式，或是对我们选择的提供过滤，我们可以将其传给replacer函数。

var nums = {

"first": 7,

"second": 14,

"third": 13

}

var luckyNums = JSON.stringify(nums, function(key, value){

if (value == 13) {

return undefined;

} else {

return value;

}

});

>> luckyNums

'{"first": 7, "second": 14}'

space参数可以是个数字，表明了作缩进的JSON字符串或字符串每个水平上缩进的空格数。如果参数是个超过10的数值，或是超过10个字符的字符串，将导致取数值10或是截取前10个字符。

var luckyNums = JSON.stringify(nums, function(key, value) {

if (value == 13) {

return undefined;

} else {

return value;

}

}, 2);

>> luckyNums

'{

"first":7,

"second":14

}'

**新增Array方法**

Array在ES5新增的方法中，参数都是function类型，默认有传参，这些参数分别是？见下面：

1. forEach

[1, 2 ,3, 4].forEach(console.log);

// 结果：

// 1, 0, [1, 2, 3, 4]

// 2, 1, [1, 2, 3, 4]

// 3, 2, [1, 2, 3, 4]

// 4, 3, [1, 2, 3, 4]

[].forEach(function(value, index, array) {

// ...

});

对比jQuery中的$.each方法：第1个和第2个参数正好是相反的，大家要注意了，不要记错了。后面类似的方法，例如$.map也是如此。

再下面，更进一步，forEach除了接受一个必须的回调函数参数，还可以接受一个可选的上下文参数（改变回调函数里面的this指向）（第2个参数）。[Dome1](dome1.html)

如果这第2个可选参数不指定，则使用全局对象代替（在浏览器是为window），严格模式下甚至是undefined.

综上全部规则，我们就可以对IE6-IE8进行仿真扩展了，如下代码：

// 对于古董浏览器，如IE6-IE8

if (typeof Array.prototype.forEach != "function") {

Array.prototype.forEach = function (fn, context) {

for (var k = 0, length = this.length; k < length; k++) {

if (typeof fn === "function" && Object.prototype.hasOwnProperty.call(this, k)) {

fn.call(context, this[k], k, this);

}

}

};

}

1. map

这里的map不是“地图”的意思，而是指“映射”。[].map(); 基本用法跟forEach方法类似：

array.map(callback,[ thisObject]);

[].map(function(value, index, array) {

// ...

});

map方法的作用不难理解，“映射”嘛，也就是原数组被“映射”成对应新数组。

var data = [1, 2, 3, 4];

var arrayOfSquares = data.map(function (item) {

return item \* item;

});

alert(arrayOfSquares); // 1, 4, 9, 16

callback需要有return值，如果没有，全部变成undefined

Array.prototype扩展可以让IE6-IE8浏览器也支持map方法：

if (typeof Array.prototype.map != "function") {

Array.prototype.map = function (fn, context) {

var arr = [];

if (typeof fn === "function") {

for (var k = 0, length = this.length; k < length; k++) {

arr.push(fn.call(context, this[k], k, this));

}

}

return arr;

};

}

1. filter

filter为“过滤”、“筛选”之意。指数组filter后，返回过滤后的新数组。用法跟map极为相似：

array.filter(callback,[ thisObject]);

filter的callback函数需要返回布尔值true或false. 如果为true则表示通过。如果为false抛弃。返回值只要是弱等于== true/false就可以了，而非非得返回 === true/false.

古老浏览器：

if (typeof Array.prototype.filter != "function") {

Array.prototype.filter = function (fn, context) {

var arr = [];

if (typeof fn === "function") {

for (var k = 0, length = this.length; k < length; k++) {

fn.call(context, this[k], k, this) && arr.push(this[k]);

}

}

return arr;

};

}

1. some

意指“某些”，指是否“某些项”合乎条件。

var scores = [5, 8, 3, 10];

var current = 7;

function higherThanCurrent(score) {

return score > current;

}

if (scores.some(higherThanCurrent)) {

alert("朕准了！");

}

我们自然可以使用forEach进行判断，不过，相比some, 不足在于，some只有有true即返回不再执行了。

1. every

跟some类似，同样是返回Boolean值，不过，every需要每一个都满足，否则false

if (scores.every(higherThanCurrent)) {

console.log("朕准了！");

} else {

console.log("来人，拖出去斩了！");

}

1. indexOf

数组这里的indexOf方法与string的类似。

array.indexOf(searchElement[, fromIndex]) var data = [2, 5, 7, 3, 5];

console.log(data.indexOf(5, "x")); // 1 ("x"被忽略)

console.log(data.indexOf(5, "3")); // 4 (从3号位开始搜索)

console.log(data.indexOf(4)); // -1 (未找到)

console.log(data.indexOf("5")); // -1 (未找到，因为5 !== "5")

兼容处理如下：

if (typeof Array.prototype.indexOf != "function") {

Array.prototype.indexOf = function (searchElement, fromIndex) {

var index = -1;

fromIndex = fromIndex \* 1 || 0;

for (var k = 0, length = this.length; k < length; k++) {

if (k >= fromIndex && this[k] === searchElement) {

index = k;

break;

}

}

return index;

};

}

1. lastIndexOf
2. reduce

“迭代”、“递归(recursion)”， 此方法相比上面的方法都复杂，用法如下：

array.reduce(callback[, initialValue])

var sum = [1, 2, 3, 4].reduce(function (previous, current, index, array) {

return previous + current;

});

console.log(sum); // 10 // 初始设置

//以下是循环过程

previous = initialValue = 1, current = 2

// 第一次迭代

previous = (1 + 2) = 3, current = 3

// 第二次迭代

previous = (3 + 3) = 6, current = 4

// 第三次迭代

previous = (6 + 4) = 10, current = undefined (退出)

有了reduce，我们可以轻松实现二维数组的扁平化：

var matrix = [

[1, 2],

[3, 4],

[5, 6]

];

// 二维数组扁平化

var flatten = matrix.reduce(function (previous, current) {

return previous.concat(current);

});

console.log(flatten); // [1, 2, 3, 4, 5, 6]

IE6-8

if (typeof Array.prototype.reduce != "function") {

Array.prototype.reduce = function (callback, initialValue ) {

var previous = initialValue, k = 0, length = this.length;

if (typeof initialValue === "undefined") {

previous = this[0];

k = 1;

}

if (typeof callback === "function") {

for (k; k < length; k++) {

this.hasOwnProperty(k) && (previous = callback(previous, this[k], k, this));

}

}

return previous;

};

}

1. reduceRight

实现上差异在于reduceRight是从数组的末尾开始实现。看下面这个例子：

var data = [1, 2, 3, 4];

var specialDiff = data.reduceRight(function (previous, current, index) {

if (index == 0) {

return previous + current;

}

return previous - current;

});

console.log(specialDiff); // 0

1. 扩展

我们使用forEach遍历DOM元素。

var eleDivs = document.getElementsByTagName("div");

Array.prototype.forEach.call(eleDivs, function(div) {

console.log("该div类名是：" + (div.className || "空"));

});

为IE6-8做的兼容存在<es5-array.js>中

**Object**

这些新增的好处之一是对象的属性有了更多控制，例如哪些是允许被修改的，哪些是可以枚举的，哪些是可以删除的等。http://www.cnblogs.com/blog-zwei1989/archive/2012/10/25/2737282.html

Object.getPrototypeOf(o)：获取给定对象的prototype对象。

Object.getPrototypeOf([])

[]

Object.getPrototypeOf({})

Object {}

Object.getPrototypeOf(1)

Number {[[PrimitiveValue]]: 0}

Object.getPrototypeOf('a')

String {length: 0, [[PrimitiveValue]]: ""}

Object.getOwnPropertyDescriptor(o,p): 获取对象描述。和Object.defineProperty的相关方法。

描述：

Value：与属性（数据仅描述符）相关联的值。

Writable：true 当且仅当与属性相关联的值可以改变（仅数据描述符）。

Get：其用作属性的getter的函数，或者undefined如果没有getter（仅访问描述符）。

Set：它作为对属性的setter的函数，或者undefined如果没有setter方法（仅访问描述符）。

Configurable：true 当且仅当该属性描述符的类型可被改变，并且如果该属性可以由相应的对象被删除。

Enumerable：true 当且仅当相应的对象的属性的枚举期间此属性显示出来。

var o, d;

o = { get foo() { return 17; } };

d = Object.getOwnPropertyDescriptor(o, 'foo');

// d is { configurable: true, enumerable: true, get: /\*这个函数\*/, set: undefined }

o = { bar: 42 };

d = Object.getOwnPropertyDescriptor(o, 'bar');

// d is { configurable: true, enumerable: true, value: 42, writable: true }

o = {};

Object.defineProperty(o, 'baz', { value: 8675309, writable: false, enumerable: false });

d = Object.getOwnPropertyDescriptor(o, 'baz');

// d is { value: 8675309, writable: false, enumerable: false, configurable: false }

Object.getOwnPropertyNames(o)：获取自有属性名列表。结果列表将不包含原型链上的属性。

// 遍历json或者用for in循环

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

console.log(Object.getOwnPropertyNames(obj).sort());

// logs ["0", "1", "2"]

// Logging property names and values using Array.forEach

Object.getOwnPropertyNames(obj).forEach(function(val, idx, array) {

console.log(val + ' -> ' + obj[val]);

});

// logs

// 0 -> a

// 1 -> b

// 2 -> c

**Bind**

function() {}.bind(thisArg [, arg1 [, arg2, …]]);

函数.bind(this上下文参数, 普通参数1, 普通参数2, ...);[dome1](dome1.html)

bind方法尤其在对象字面量编程风格下尤为受用。

var OOO = {

color: "#cd0000",

element: $("#text"),

events: function() {

$("input[type='button']").addEventListener("click", function(e) {

this.element.style.color = this.color;

}.bind(this));

return this;

},

init: function() {

this.events();

}

};

IE6-8不支持bind方法，需要做兼容处理。

if (!function() {}.bind) {

Function.prototype.bind = function(context) {

var self = this

, args = Array.prototype.slice.call(arguments);

return function() {

return self.apply(context, args.slice(1));

}

};

}