|  |
| --- |
| [键入公司名称] |
| 简单的文档说明 |
| Javascript及其库的使用 |
|  |
|  |
|  |

|  |
| --- |
| [在此处键入文档摘要。摘要通常为文档内容的简短概括。在此处键入文档摘要。摘要通常为文档内容的简短概括。] |

目录

[Ajax 的使用 1](#_Toc509167298)

[Json的使用 （在js） 2](#_Toc509167299)

[对于javascript的使用 3](#_Toc509167300)

[对undefined的认识 3](#_Toc509167301)

[操作字符串 4](#_Toc509167302)

[数组 4](#_Toc509167303)

[IndexOf的使用 5](#_Toc509167304)

[slice的使用 5](#_Toc509167305)

[push和pop 5](#_Toc509167306)

[unshift和shift 5](#_Toc509167307)

[splice()方法 6](#_Toc509167308)

[对象 6](#_Toc509167309)

[循环 7](#_Toc509167310)

[Map 和set 7](#_Toc509167311)

[Iterable方法 7](#_Toc509167312)

[函数 8](#_Toc509167313)

[变量作用域与解构赋值 10](#_Toc509167314)

[变量提升 10](#_Toc509167315)

[解构赋值 10](#_Toc509167316)

[方法 11](#_Toc509167317)

[装饰器 12](#_Toc509167318)

[高阶函数 12](#_Toc509167319)

[Reduce（） 13](#_Toc509167320)

[Filter（） 13](#_Toc509167321)

[回调函数 13](#_Toc509167322)

[Sort 高阶函数 14](#_Toc509167323)

[闭包 14](#_Toc509167324)

[箭头函数 15](#_Toc509167325)

[Generator 自动生成器 16](#_Toc509167326)

[标准对象 17](#_Toc509167327)

[包装对象 17](#_Toc509167328)

[Date 用来表示日期和时间 18](#_Toc509167329)

[Js中的继承 20](#_Toc509167330)

[一种是类式继承 （构造函数） 20](#_Toc509167331)

[对于原型继承 20](#_Toc509167332)

[操作dom对象 document object model 21](#_Toc509167333)

[插入dom 22](#_Toc509167334)

[删除DOM 23](#_Toc509167335)

[操作表单 24](#_Toc509167336)

[操作文件 26](#_Toc509167337)

[Js的回调函数的理解 27](#_Toc509167338)

[JQuery 的使用 28](#_Toc509167339)

[层级选择器 28](#_Toc509167340)

[子选择器（Child Selector） 28](#_Toc509167341)

[查找和过滤 29](#_Toc509167342)

[过滤 29](#_Toc509167343)

[显示和隐藏DOM 29](#_Toc509167344)

# Ajax 的使用

AJAX不是JavaScript的规范，它只是一个哥们“发明”的缩写：Asynchronous JavaScript and XML，意思就是用JavaScript执行异步网络请求

现代浏览器上写AJAX主要依靠XMLHttpRequest对象：

var xhr = new XMLHttpResquset();

对于低版本的IE，需要换一个ActiveXObject对象

Var activeObj = new ActiveXObject(“Micsoft.XMLHTTP ”)

一般为了进行兼容性较好的支持

xmlHttpResquset 对象的创建 /// 为了适应不同的浏览器

var xhr = false;

try{

xhr = new XMLHttpResquset();

} catch( e ) {

try {

xhr = new ActiveXobject ("Microsoft.XMLHTTP")

} // 老版本的浏览器

catch (e1){

xhr = false;

}

}

接下来就是对ajax’一些也是对xmlHttpResquset （是ajax的核心） 的简单介绍

XMLhttpResquest 对象的属性

readState 表示异步请求的各种状态

onreadstatechange 触发事件的时间处理程序

responseText 服务器返回的数据的字符形式

responseXML 返回的XML文档数据对象

status 服务器返回的响应状码

statusText 状态码的字符串信息

XMLHttpResponse 的方法

Void open ( method ,url, asynch ,usename , password)

建立对服务器的调用

Asynch 两个值 true和false 表示是否为异步

Void send （content） 发出的请求 附加内容content

Void setResqusetHeader (header value)

一个 ajax 应用实例 一般要经过一下过程

1 在页面进行 Ajax 请求的触发事件

2 创建XMLHttpResquest 的对象

3确定请求地址和请求参数

4调用open() 方法进行建立对服务器的调用

5通过readystatechange 属性指定响应事件处理函数

6在函数中根据响应状态进行数据获取和数据处理工作

7通关send()方法向服务器发出请求

# Json的使用 （在js）

1. 将json 数据赋值给变量

Var usersArray =

{ “users” : [

{ “firstName”: “Bob”,”lastName”:”Liu”,”email”:”Liu@gamil.com”}

{ “firstName”: “Bob”,”lastName”:”Liu”,”email”:”Liu@gamil.com”}

{ “firstName”: “Bob”,”lastName”:”Liu”,”email”:”Liu@gamil.com”}

]}：

2获取 json 数据中对象信息

userArray.users[0].firstName

1. 对json 进行修改

usersArray.users[0].email = “”

1. 将json文本转换成js对象

//定义json文本变量

Var text =”{\”province\”:\”city\”:\”华北\”}”

//将json文本转换js

Var jsonObj = eval ( “(“+ text + “)”);

// 获取对象

Alert (jsonObj.city)

# 对于javascript的使用

## 对undefined的认识

undefined 表示一个未声明的变量，或已声明但没有赋值的变量，或一个并不存在的对象属性，函数没有返回值时，默认返回undefined

一种是相等的比较运算符 不是== 而是 ===

NaN这个特殊的number 他和自己都不相等 唯一能判断的方法是isNaN

NaN === NaN （false） isNaN（NaN） （true

最后要注意浮点数的相等比较：

1 / 3 === (1 - 2 / 3); // false

这不是JavaScript的设计缺陷。浮点数在运算过程中会产生误差，因为计算机无法精确表示无限循环小数。要比较两个浮点数是否相等，只能计算它们之差的绝对值，看是否小于某个阈值：

Math.abs(1 / 3 - (1 - 2 / 3)) < 0.0000001; // true

数组

Js 中数组可以包括任意数据类型

[ 1,2,3’hello’, null ,true]

数组的定义也可以采用Array() new Array(1,3,2);

数组的元素可以通过索引来访问

var arr = [1, 2, 3.14, 'Hello', null, true];

arr[0]; // 返回索引为0的元素，即1

arr[5]; // 返回索引为5的元素，即true

arr[6]; // 索引超出了范围，返回undefined

对象 js的对象是一组由键-值组成的无序集合

var person = {

name: 'Bob',

age: 20,

zipcode: null

};

获取属性的方式和java类似（都是用对象.属性）

Eg: person.name // ‘Bob’

字符串一般都是 ‘’ and “”

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如

'I\'m \"OK\"!';

表示的字符串内容是：I'm "OK"!

var message = '你好, ' + name + ', 你今年' + age + '岁了!';

var message = `你好, ${name}, 你今年${age}岁了!`; （前提当然是var一些变量已经定义好了 相当于一个模板的作用）

## 操作字符串

1. 索引取值 (字符串是不可变的)
2. 一些常见的方法并不会改变原来内容而是返回一个新的内容

toUpperCase()把一个字符串全部变为大写 ( var s =’Hello’ ;s.toUpperCase())

toLowerCase()把一个字符串全部变为小写

indexOf()会搜索指定字符串出现的位置

var s = 'hello, world';

s.indexOf('world'); // 返回7

s.indexOf('World'); // 没有找到指定的子串，返回-1

substring()返回指定索引区间的子串

var s = 'hello, world'

s.substring(0, 5); // 从索引0开始到5（不包括5），返回'hello'

s.substring(7); // 从索引7开始到结束，返回'world'

# 数组

请注意，直接给Array的length赋一个新的值会导致Array大小的变化：

var arr = [1, 2, 3];

arr.length; // 3

arr.length = 6;

arr; // arr变为[1, 2, 3, undefined, undefined, undefined]

arr.length = 2;

arr; // arr变为[1, 2]

Array可以通过索引把对应的元素修改为新的值，因此，对Array的索引进行赋值会直接修改这个Array：

var arr = ['A', 'B', 'C'];

arr[1] = 99;

arr; // arr现在变为['A', 99, 'C']

请注意，如果通过索引赋值时，索引超过了范围，同样会引起Array大小的变化：

var arr = [1, 2, 3];

arr[5] = 'x';

arr; // arr变为[1, 2, 3, undefined, undefined, 'x']

大多数其他编程语言不允许直接改变数组的大小，越界访问索引会报错。然而，JavaScript的Array却不会有任何错误。在编写代码时，不建议直接修改Array的大小，访问索引时要确保索引不会越界

## IndexOf的使用

与String类似，Array也可以通过indexOf()来搜索一个指定的元素的位置：

var arr = [10, 20, '30', 'xyz'];

arr.indexOf(10); // 元素10的索引为0

arr.indexOf(20); // 元素20的索引为1

arr.indexOf(30); // 元素30没有找到，返回-1

arr.indexOf('30'); // 元素'30'的索引为2

## slice的使用

slice()就是对应String的substring()版本，它截取Array的部分元素，然后返回一个新的Array：

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

arr.slice(0, 3); // 从索引0开始，到索引3结束，但不包括索引3: ['A', 'B', 'C']

arr.slice(3); // 从索引3开始到结束: ['D', 'E', 'F', 'G']

注意到slice()的起止参数包括开始索引，不包括结束索引。

如果不给slice()传递任何参数，它就会从头到尾截取所有元素。利用这一点，我们可以很容易地复制一个Array：

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

var aCopy = arr.slice();

aCopy; // ['A', 'B', 'C', 'D', 'E', 'F', 'G']

aCopy === arr; // false

## push和pop

push()向Array的末尾添加若干元素，pop()则把Array的最后一个元素删除掉：

var arr = [1, 2];

arr.push('A', 'B'); // 返回Array新的长度: 4

arr; // [1, 2, 'A', 'B']

arr.pop(); // pop()返回'B'

arr; // [1, 2, 'A']

arr.pop(); arr.pop(); arr.pop(); // 连续pop 3次

arr; // []

arr.pop(); // 空数组继续pop不会报错，而是返回undefined

arr; // []

## unshift和shift

如果要往Array的头部添加若干元素，使用unshift()方法，shift()方法则把Array的第一个元素删掉

sort

sort()可以对当前Array进行排序，它会直接修改当前Array的元素位置，直接调用时，按照默认顺序排

reverse

reverse()把整个Array的元素给掉个个，也就是反转

var arr = ['one', 'two', 'three'];

arr.reverse();

arr; // ['three', 'two', 'one'] 一般的实现方法都是这样的 显示调用 然后是变量名输出显示）

## splice()方法

splice()方法是修改Array的“万能方法”，它可以从指定的索引开始删除若干元素，然后再从该位置添加若干元素：

var arr = ['Microsoft', 'Apple', 'Yahoo', 'AOL', 'Excite', 'Oracle'];

// 从索引2开始删除3个元素,然后再添加两个元素:

arr.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素 ['Yahoo', 'AOL', 'Excite']

arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']

// 只删除,不添加:

arr.splice(2, 2); // ['Google', 'Facebook']

arr; // ['Microsoft', 'Apple', 'Oracle']

// 只添加,不删除:

arr.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素

arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']

# 对象

var xiaoming = {

name: '小明'

};

xiaoming.age; // undefined

xiaoming.age = 18; // 新增一个age属性

xiaoming.age; // 18

delete xiaoming.age; // 删除age属性

xiaoming.age; // undefined

delete xiaoming['name']; // 删除name属性

xiaoming.name; // undefined

delete xiaoming.school; // 删除一个不存在的school属性也不会报错

注意 要是检验一个属性是否存在的话采用的是 in 操作符

'toString' in xiaoming; // true

## 循环

For ……in 可以把对象的属性一一遍历出来

要过滤掉对象继承的属性，用hasOwnProperty()来实现：

var o = {

name: 'Jack',

age: 20,

city: 'Beijing'

};

for (var key in o) {

if (o.hasOwnProperty(key)) {

console.log(key); // 'name', 'age', 'city'

}

}

请注意，for ... in对Array的循环得到的是String而不是Number。

## Map 和set

初始化Map需要一个二维数组，或者直接初始化一个空Map。Map具有以下方法：

var m = new Map(); // 空Map

m.set('Adam', 67); // 添加新的key-value

m.set('Bob', 59);

m.has('Adam'); // 是否存在key 'Adam': true

m.get('Adam'); // 67

m.delete('Adam'); // 删除key 'Adam'

m.get('Adam'); // undefined

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

Set也是数组 但是不存在重复的键值

## Iterable方法

遍历Array可以采用下标循环，遍历Map和Set就无法使用下标。为了统一集合类型，ES6标准引入了新的iterable类型，Array、Map和Set都属于iterable类型。

具有iterable类型的集合可以通过新的for ... of循环来遍历

用for ... of循环遍历集合，用法如下：

var a = ['A', 'B', 'C'];

var s = new Set(['A', 'B', 'C']);

var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);

for (var x of a) { // 遍历Array

console.log(x);

}

for (var x of s) { // 遍历Set

console.log(x);

}

for (var x of m) { // 遍历Map

console.log(x[0] + '=' + x[1]);

}

你可能会有疑问，for ... of循环和for ... in循环有何区别？

for ... in循环由于历史遗留问题，它遍历的实际上是对象的属性名称。一个Array数组实际上也是一个对象，它的每个元素的索引被视为一个属性。

当我们手动给Array对象添加了额外的属性后，for ... in循环将带来意想不到的意外效果：

var a = ['A', 'B', 'C'];

a.name = 'Hello';

for (var x in a) {

console.log(x); // '0', '1', '2', 'name'

}

for ... in循环将把name包括在内，但Array的length属性却不包括在内。

for ... of循环则完全修复了这些问题，它只循环集合本身的元素：

var a = ['A', 'B', 'C'];

a.name = 'Hello';

for (var x of a) {

console.log(x); // 'A', 'B', 'C'

}

这就是为什么要引入新的for ... of循环

# 函数

两种不同的方式，只是稍有不同

Javascript 中有一个关键字 arguments 只在函数中起作用，并且永远指向当前函数的调用者传入的所有参数

function foo(x) {

console.log('x = ' + x); // 10

for (var i=0; i<arguments.length; i++) {

console.log('arg ' + i + ' = ' + arguments[i]); // 10, 20, 30

}

}

foo(10, 20, 30); 最后打印的是 一个循环的输出

利用arguments，你可以获得调用者传入的所有参数。也就是说，即使函数不定义任何参数，还是可以拿到参数的值

function abs() {

if (arguments.length === 0) {

return 0;

}

var x = arguments[0];

return x >= 0 ? x : -x;

}

abs(); // 0

abs(10); // 10

abs(-9); // 9

rest 参数 用来接收 多余的参数 （js 中是允许接受任意个参数的）

function foo(a, b) {

var i, rest = [];

if (arguments.length > 2) {

for (i = 2; i<arguments.length; i++) {

rest.push(arguments[i]);

}

}

console.log('a = ' + a);

console.log('b = ' + b);

console.log(rest);

} rest 是为了获取多余的参数

ES6 标准 引入了rest参数

function foo(a, b, ...rest) {

console.log('a = ' + a);

console.log('b = ' + b);

console.log(rest);

}

注意一下争正确的格式 js是有自动添加分好的机制 的但是 return只是在本句本行而已

function foo() {

return { // 这里不会自动加分号，因为{表示语句尚未结束

name: 'foo'

};

}

注意 js 的max函数 正确使用应该是 Math.max

# 变量作用域与解构赋值

1. 如果一个变量在函数体内部申明，则该变量的作用域为整个函数体，在函数体外不可引用该变量
2. 如果两个不同的函数各自申明了同一个变量，那么该变量只在各自的函数体内起作用。换句话说，不同函数内部的同名变量互相独立，互不影响
3. 由于JavaScript的函数可以嵌套，此时，内部函数可以访问外部函数定义的变量，反过来则不行
4. 如果内部函数和外部函数的变量名重名怎么办？这说明JavaScript的函数在查找变量时从自身函数定义开始，从“内”向“外”查找。如果内部函数定义了与外部函数重名的变量，则内部函数的变量将“屏蔽”外部函数的变量

## 变量提升

JavaScript的函数定义有个特点，它会先扫描整个函数体的语句，把所有申明的变量“提升”到函数顶部

'use strict';

function foo() {

var x = 'Hello, ' + y;

console.log(x);

var y = 'Bob';

}

foo();

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象window，全局作用域的变量实际上被绑定到window的一个属性：

## 解构赋值

就是 同时对一个数组的值赋予变量

// 如果浏览器支持解构赋值就不会报错:

var [x, y, z] = ['hello', 'JavaScript', 'ES6'];

传统的赋值时逐个进行赋值

解构赋值可以对对象进行赋值

var person = {

name: '小明',

age: 20,

gender: 'male',

passport: 'G-12345678',

school: 'No.4 middle school'

};

// 把passport属性赋值给变量id:

let {name, passport:id} = person;

name; // '小明'

id; // 'G-12345678'

// 注意: passport不是变量，而是为了让变量id获得passport属性:

passport; // Uncaught ReferenceError: passport is not defined

解构赋值的使用场景

例如，交换两个变量x和y的值，可以这么写，不再需要临时变量：

var x=1, y=2;

[x, y] = [y, x]

## 方法

可以在对象中绑定方法

var xiaoming = {

name: '小明',

birth: 1990,

age: function () {

var y = new Date().getFullYear();

return y - this.birth;

}

};

需要的注意的是 在内部方法中this的指向问题 ，需要视情况而定可能是指向对象 也可能指向的是window （这个全局对象）

采用var that =this ; 就可以在方法内部定义其他函数

var xiaoming = {

name: '小明',

birth: 1990,

age: function () {

var that = this; // 在方法内部一开始就捕获this

function getAgeFromBirth() {

var y = new Date().getFullYear();

return y - that.birth; // 用that而不是this

}

return getAgeFromBirth();

}

};

xiaoming.age(); // 25

但是想要进行对this指向改变的话 可以使用apply方法 （接受两个参数一个是this一个是Array）

Function get Age() {

Var y = new Date().getFullYear()’

Return y – this.birth;

}

Var xiaoming ={

Name : ‘小明’;

Birth : 1990;

Age : getage

};

Xiaoming.age(); //25

getAge.apply(xiaoming ,[]) // 25 this 指向xiaoming 参数weikong

## 装饰器

利用apply() 我们还可以动态改变函数的行为

Js中所有的对象都是动态的 即使是内置的函数也可以进行指向的改变

假定想统计parseInt() 的使用次数

var count = 0;

var oldParseInt = parseInt; // 保存原函数

window.parseInt = function () {

count += 1;

return oldParseInt.apply(null, arguments); // 调用原函数

};

## 高阶函数

JavaScript的函数其实都指向某个变量。既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数

function add(x, y, f) {

return f(x) + f(y);

}

Map 的使用 把一个函数作用在一个数组上

Map()方法是定义在javascript的Array 的map()方法中,然后传入自己的函数

Function pow(x){

Return x \*x;

}

Var arr = [1,2,3,4];

Var result = arr.map(pow);

Console.log (result); 传入的正式函数对象本身

### Reduce（）

再看reduce的用法。Array的reduce()把一个函数作用在这个Array的[x1, x2, x3...]上，这个函数必须接收两个参数，reduce()把结果继续和序列的下一个元素做累积计算，

var arr = [1, 3, 5, 7, 9];

arr.reduce(function (x, y) {

return x + y;

}); // 25

### Filter（）

filter也是一个常用的操作，它用于把Array的某些元素过滤掉，然后返回剩下的元素。

和map()类似，Array的filter()也接收一个函数。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是true还是false决定保留还是丢弃该元素

例如，在一个Array中，删掉偶数，只保留奇数，可以这么写：

var arr = [1, 2, 4, 5, 6, 9, 10, 15];

var r = arr.filter(function (x) {

return x % 2 !== 0;

});

r; // [1, 5, 9, 15]

Filter() 这个高阶函数 ，关键在于实行删选

## 回调函数

filter()接收的回调函数，其实可以有多个参数。通常我们仅使用第一个参数，表示Array的某个元素。回调函数还可以接收另外两个参数，表示元素的位置和数组本身

var arr = ['A', 'B', 'C'];

var r = arr.filter(function (element, index, self) {

console.log(element); // 依次打印'A', 'B', 'C'

console.log(index); // 依次打印0, 1, 2

console.log(self); // self就是变量arr

return true;

});

利用filter，可以巧妙地去除Array的重复元素

r = arr.filter(function (element, index, self) {

return self.indexOf(element) === index;

});

去除重复元素依靠的是indexOf总是返回第一个元素的位置，后续的重复元素位置与indexOf返回的位置不相等，因此被filter滤掉了

Sort 高阶函数 （一个排序方式并不是按照数字的大小进行排序的）

但是可以在其中进行接受一个比较函数进行自定义的排序

Array[] 的sort（）是吧所有的元素先转换成string 在进行排序

arr.sort(function (x, y) {

if (x < y) {

return -1;

}

if (x > y) {

return 1;

}

return 0;相应的排列，

}); 便可实现

字符串的比较

var arr = ['Google', 'apple', 'Microsoft'];

arr.sort(function (s1, s2) {

x1 = s1.toUpperCase();

x2 = s2.toUpperCase();

if (x1 < x2) {

return -1;

}

if (x1 > x2) {

return 1;

}

return 0;

});

## 闭包

function lazy\_sum(arr) {

var sum = function () {

return arr.reduce(function (x, y) {

return x + y;

});

}

return sum; // 注意是在函数体内 闭包

}

们在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力

闭包可以简单的理解成“定义在一个函数内部的函数”所以本质闭包就是将函数内部函数和外部函数的链接起来的一座桥梁

使用闭包时需要注意的是 闭包会使函数中的变量都保存在内存中 内存消耗的非常大

闭包的使用

function create\_counter(initial) {

var x = initial || 0;

return {

inc: function () {

x += 1;

return x;

}

}

返回的对象中 闭包携带了 局部变量x 就相当于java或者c中Private（封装），外界无法访问

还有就是可以把多参数函数变成单参数函数

function make\_pow(n) {

return function (x) {

return Math.pow(x, n);

}

} 当然具体的Math.pow() 自己看

## 箭头函数

箭头函数相当于匿名函数，并且简化了函数的定义一种像上面的，只包含一个表达式，连{ ... }和return都省略掉了。还有一种可以包含多条语句，这时候就不能省略{ ... }和return：

// 两个参数:

(x, y) => x \* x + y \* y

// 无参数:

() => 3.14

// 可变参数:

(x, y, ...rest) => {

var i, sum = x + y;

for (i=0; i<rest.length; i++) {

sum += rest[i];

}

return sum;

}

注意的是其中的**this 关键字，**其中最大的区别是指定的是词法中的作用域，但是之前函数中的指向是不确定的，可能是window或者是undefined

之前的是需要 进行 var that = this; 进行指向

由于this在箭头函数中已经按照词法作用域绑定了，所以，用call()或者apply()调用箭头函数时，无法对this进行绑定，即传入的第一个参数被忽略

# Generator 自动生成器

generator和函数不同的是，generator由function\*定义（注意多出的\*号），并且，除了return语句，还可以用yield返回多次

function\* fib(max) {

var

t,

a = 0,

b = 1,

n = 0;

while (n < max) {

yield a;

[a, b] = [b, a + b];

n ++;

}

return;

}

直接调用试试：

fib(5); // fib {[[GeneratorStatus]]: "suspended", [[GeneratorReceiver]]: Window}

直接调用一个generator和调用函数不一样，fib(5)仅仅是创建了一个generator对象，还没有去执行它。

调用generator对象有两个方法，一是不断地调用generator对象的next()方法：

var f = fib(5);

f.next(); // {value: 0, done: false}

f.next(); // {value: 1, done: false}

f.next(); // {value: 1, done: false}

f.next(); // {value: 2, done: false}

f.next(); // {value: 3, done: false}

f.next(); // {value: undefined, done: true}

next()方法会执行generator的代码，然后，每次遇到yield x;就返回一个对象{value: x, done: true/false}，然后“暂停”。返回的value就是yield的返回值，done表示这个generator是否已经执行结束了。如果done为true，则value就是return的返回值。

当执行到done为true时，这个generator对象就已经全部执行完毕，不要再继续调用next()了。

第二个方法是直接用for ... of循环迭代generator对象，这种方式不需要我们自己判断done：

**用generator的好处**

1因为generator可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数，利用这一点，写一个generator就可以实现需要用面向对象才能实现的功能

2 generator还有另一个巨大的好处，就是把异步回调代码变成“同步”代码。这个好处要等到后面学了AJAX以后才能体会到

## 标准对象

typeof操作符获取对象的类型，它总是返回一个字符串：

typeof 123; // 'number'

typeof NaN; // 'number'

typeof 'str'; // 'string'

## 包装对象

比如说 （int和integer的关系）int 是基本类型，直接存数值，进行初始化时int类的变量初始为0。

integer是对象，用一个引用指向这个对象，Integer的变量则初始化为null

具体的详细地址在 <http://blog.csdn.net/chenliguan/article/details/53888018>

原始类型: boolean，char，byte，short，int，long，float，double

封装类类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

自动装箱：将基本数据类型重新转化为对象

自动拆箱：将对象重新转化为基本数据类型

本身对对象是不能直接进行数值赋值于对象 但是jdk 1.5 之后就可以了

在使用包装对象是需要注意的是

不要使用new Number()、new Boolean()、new String()创建包装对象；

用parseInt()或parseFloat()来转换任意类型到number；

用String()来转换任意类型到string，或者直接调用某个对象的toString()方法；

通常不必把任意类型转换为boolean再判断，因为可以直接写if (myVar) {...}

typeof操作符可以判断出number、boolean、string、function和undefined；

判断Array要使用Array.isArray(arr)；

判断null请使用myVar === null；

判断某个全局变量是否存在用typeof window.myVar === 'undefined'；

函数内部判断某个变量是否存在用typeof myVar === 'undefined'。

最后有细心的同学指出，任何对象都有toString()方法吗？null和undefined就没有！确实如此，这两个特殊值要除外，虽然null还伪装成了object类型。

更细心的同学指出，number对象调用toString()报SyntaxError

## Date 用来表示日期和时间

Var now = new Date();

Now.getDay();

需要注意的是 js中月份是0~11

使用Date.parse()时传入的字符串使用实际月份01~12，转换为Date对象后getMonth()获取的月份值为0~11

时区

Date对象表示的时间总是按浏览器所在时区显示的，不过我们既可以显示本地时间，也可以显示调整后的UTC时间：

var d = new Date(1435146562875);

d.toLocaleString(); // '2015/6/24 下午7:49:22'，本地时间（北京时区+8:00），显示的字符串与操作系统设定的格式有关

d.toUTCString(); // 'Wed, 24 Jun 2015 11:49:22 GMT'，UTC时间，与本地时间相差8小时

对于时间戳的认识 时间戳是一个自增的整数，它表示从1970年1月1日零时整的GMT时区开始的那一刻，到现在的毫秒数。假设浏览器所在电脑的时间是准确的，那么世界上无论哪个时区的电脑，它们此刻产生的时间戳数字都是一样的，所以，时间戳可以精确地表示一个时刻，并且与时区无关

RegExp对象表示正则表达式,它是对字符串执行模式匹配的强大工具

而正则表达式对象的构造函数，如 new RegExp('ab+c') 提供了正则表达式运行时编译。

var expression = /pattern/ flags;

pattern 部分可以是任何简单或复杂的正则表达示

console.log(pattern5.global); //false 是否设置了g标志

console.log(pattern5.ignoreCase); //true 是否设置了i标志

console.log(pattern5.multiline); //false 是否设置了m标志

console.log(pattern5.lastIndex); //0 开始搜索下一个匹配项的起始位置

console.log(pattern5.source); //[bc]at 正则表达式的字符串表示

具体的详细地址 <http://www.jb51.net/article/95905.htm>

正则表达式第二个作用就是分组 （提取子串）

如果正则表达式中定义了组，就可以在RegExp对象上用exec()方法提取出子串来。

exec()方法在匹配成功后，会返回一个Array，第一个元素是正则表达式匹配到的整个字符串，后面的字符串表示匹配成功的子串。

exec()方法在匹配失败时返回null。

关于贪婪匹配

var re = /^(\d+)(0\*)$/;

re.exec('102300'); // ['102300', '102300', '']

由于\d+采用贪婪匹配，直接把后面的0全部匹配了，结果0\*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的0匹配出来，加个?就可以让\d+采用非贪婪匹配

JavaScript的正则表达式还有几个特殊的标志，最常用的是g，表示全局匹配：

var r1 = /test/g;

// 等价于:

var r2 = new RegExp('test', 'g')

对于构造函数

function Person( name){

       this.name =name;

     }

      var p1=new Person('John');

等同于

function person(name ){

     Object obj =new Object();

     obj.name =name;

      return obj;

   }

    var p1= person("John");

这就是区别的地方,new出来的是构造函数并且 没有return返回直接直接使用简单的object.perpority

函数也是一个对象 ,它的原型链是

foo ----> Function.prototype ----> Object.prototype ----> null

function Student(props) {

this.name = props.name || '匿名'; // 默认值为'匿名'

this.grade = props.grade || 1; // 默认值为1

}

Student.prototype.hello = function () {

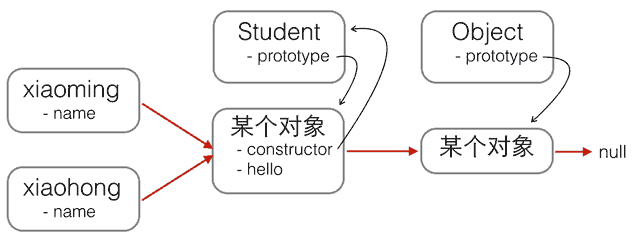
alert('Hello, ' + this.name + '!');

};

function createStudent(props) {

return new Student(props || {})

}



要让创建的对象共享一个hello函数，根据对象的属性查找原则，我们只要把hello函数移动到xiaoming、xiaohong这些对象共同的原型上就可以了，也就是Student.prototype （共享一个函数可以进行节省内存）

如果是忘记写new 函数的话，可以编写一个create()函数 在内部进行new 的封装

# Js中的继承

## 一种是类式继承 （构造函数）

Js 中是没有类的概念的 之所以这样说也是模拟的概念 ，其中call和apply为继承提供了支持，并且通过改变this

的作用环境，使子类具有父类的各种属性

var father = function() {

this.age = 52;

this.say = function() {

alert('hello i am '+ this.name ' and i am '+this.age + 'years old');

}

}

var child = function() {

this.name = 'bill';

father.call(this);

}

var man = new child();

man.say();

## 对于原型继承

原型继承在开发中经常用到。它有别于类继承是因为继承不在对象本身，而在对象的原型上（prototype）。每一个对象都有原型，在浏览器中它体现在一个隐藏的\_\_proto\_\_属性上

var

father=function(){}

father.prototype.a=function(){} // 原型继承

var

child=function(){}

//开始继承

child.prototype=new 真正的开始继承

father();

var

man=new

child();

man.a();

# 操作dom对象 document object model

由于HTML文档被浏览器解析后就是一棵DOM树，要改变HTML的结构，就需要通过JavaScript来操作DOM。

始终记住DOM是一个树形结构。操作一个DOM节点实际上就是这么几个操作：

更新：更新该DOM节点的内容，相当于更新了该DOM节点表示的HTML的内容；

遍历：遍历该DOM节点下的子节点，以便进行进一步操作；

添加：在该DOM节点下新增一个子节点，相当于动态增加了一个HTML节点；

删除：将该节点从HTML中删除，相当于删掉了该DOM节点的内容以及它包含的所有子节点。

在操作一个DOM节点前，我们需要通过各种方式先拿到这个DOM节点。最常用的方法是document.getElementById()和document.getElementsByTagName()，以及CSS选择器document.getElementsByClassName()。

由于ID在HTML文档中是唯一的，所以document.getElementById()可以直接定位唯一的一个DOM节点。document.getElementsByTagName()和document.getElementsByClassName()总是返回一组DOM节点。要精确地选择DOM，可以先定位父节点，再从父节点开始选择，以缩小范围

// 返回ID为'test'的节点：

var test = document.getElementById('test');

// 先定位ID为'test-table'的节点，再返回其内部所有tr节点：

var trs = document.getElementById('test-table').getElementsByTagName('tr');

// 先定位ID为'test-div'的节点，再返回其内部所有class包含red的节点：

var reds = document.getElementById('test-div').getElementsByClassName('red');

// 获取节点test下的所有直属子节点:

var cs = test.children;

// 获取节点test下第一个、最后一个子节点：

var first = test.firstElementChild;

var last = test.lastElementChild;

第二种方法是使用querySelector()和querySelectorAll()，需要了解selector语法，然后使用条件来获取节点，更加方便：

// 通过querySelector获取ID为q1的节点：

var q1 = document.querySelector('#q1');

// 通过querySelectorAll获取q1节点内的符合条件的所有节点：

var ps = q1.querySelectorAll('div.highlighted > p');

更新dom的操作方法

一种是修改innerHTML属性，这个方式非常强大，不但可以修改一个DOM节点的文本内容，还可以直接通过HTML片段修改DOM节点内部的子树：

// 获取<p id="p-id">...</p>

var p = document.getElementById('p-id');

// 设置文本为abc:

p.innerHTML = 'ABC'; // <p id="p-id">ABC</p>

// 设置HTML:

p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';

// <p>...</p>的内部结构已修改

## 插入dom

如果这个DOM节点是空的，例如，<div></div>，那么，直接使用innerHTML = '<span>child</span>'就可以修改DOM节点的内容，相当于“插入”了新的DOM节点。

如果这个DOM节点不是空的，那就不能这么做，因为innerHTML会直接替换掉原来的所有子节点

有两个办法可以插入新的节点。一个是使用appendChild，把一个子节点添加到父节点的最后一个子节点

<!-- HTML结构 -->

<p id="js">JavaScript</p>

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

</div>

把<p id="js">JavaScript</p>添加到<div id="list">的最后一项：

var

js = document.getElementById('js'),

list = document.getElementById('list');

list.appendChild(js);

insertBefore

如果我们要把子节点插入到指定的位置怎么办？可以使用parentElement.insertBefore(newElement, referenceElement);，子节点会插入到referenceElement之前。

还是以上面的HTML为例，

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

</div>

可以这么写：

var

list = document.getElementById('list'),

ref = document.getElementById('python'),

haskell = document.createElement('p');

haskell.id = 'haskell';

haskell.innerText = 'Haskell';

list.insertBefore(haskell, ref);

## 删除DOM

删除一个DOM节点就比插入要容易得多。

要删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的removeChild把自己删掉：

// 拿到待删除节点:

var self = document.getElementById('to-be-removed');

// 拿到父节点:

var parent = self.parentElement;

// 删除:

var removed = parent.removeChild(self);

removed === self; // true

注意到删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置

\

# 操作表单

如果我们获得了一个<input>节点的引用，就可以直接调用value获得对应的用户输入值：

// <input type="text" id="email">

var input = document.getElementById('email');

input.value; // '用户输入的值'

进行相应的赋值直接在结尾 input.vaule = number；

但是对于选择框和复选框 永远返回的都是HTML 设置的值需要进行checked 的检验

Eg ： 如上所例 input.checked 会返回的值是false或者是true

提交表单 （一般在页面会使用进行数据的获取）

**一种是根据在button的一个click事件**

<form id="test-form">

<input type="text" name="test">

<button type="button" onclick="doSubmitForm()">Submit</button>

</form>

<script>

function doSubmitForm() {

var form = document.getElementById('test-form');

// 可以在此修改form的input...

// 提交form:

form.submit();

}

</script>

另一种是二种方式是响应<form>本身的onsubmit事件，在提交form时作修改：

form id="test-form" onsubmit="return checkForm()">

<input type="text" name="test">

<button type="submit">Submit</button>

</form>

<script>

function checkForm() {

var form = document.getElementById('test-form');

// 可以在此修改form的input...

// 继续下一步:

return true;

}

</script>

很多登录表单希望用户输入用户名和口令，但是，安全考虑，提交表单时不传输明文口令，而是口令的MD5

<!-- HTML -->

<form id="login-form" method="post" onsubmit="return checkForm()">

<input type="text" id="username" name="username">

<input type="password" id="input-password">

<input type="hidden" id="md5-password" name="password">

<button type="submit">Submit</button>

</form>

<script>

function checkForm() {

var input\_pwd = document.getElementById('input-password');

var md5\_pwd = document.getElementById('md5-password');

// 把用户输入的明文变为MD5:

md5\_pwd.value = toMD5(input\_pwd.value);

// 继续下一步:

return true;

}

</script>

# 操作文件

在HTML表单中，可以上传文件的唯一控件就是<input type="file"> js 对待上传的文件是不能进行操作的但是课可以进行对文件类型的筛选

var f = document.getElementById('test-file-upload');

var filename = f.value; // 'C:\fakepath\test.png'

if (!filename || !(filename.endsWith('.jpg') || filename.endsWith('.png') || filename.endsWith('.gif'))) {

alert('Can only upload image file.');

return false;

}

但是HTML5的File API提供了File和FileReader两个主要对象，可以获得文件信息并读取文件

var

fileInput = document.getElementById('test-image-file'),

info = document.getElementById('test-file-info'),

preview = document.getElementById('test-image-preview');

// 监听change事件:

fileInput.addEventListener('change', function () {

// 清除背景图片:

preview.style.backgroundImage = '';

// 检查文件是否选择:

if (!fileInput.value) {

info.innerHTML = '没有选择文件';

return;

}

// 获取File引用:

var file = fileInput.files[0];

// 获取File信息:

info.innerHTML = '文件: ' + file.name + '<br>' +

'大小: ' + file.size + '<br>' +

'修改: ' + file.lastModifiedDate;

if (file.type !== 'image/jpeg' && file.type !== 'image/png' && file.type !== 'image/gif') {

alert('不是有效的图片文件!');

return;

}

// 读取文件:

var reader = new FileReader();

reader.onload = function(e) {

var

data = e.target.result; // 'data:image/jpeg;base64,/9j/4AAQSk...(base64编码)...'

preview.style.backgroundImage = 'url(' + data + ')';

};

// 以DataURL的形式读取文件:

reader.readAsDataURL(file);

});

在最后一句中 read.readAsDataURL(File);

reader.onload = function(e) {

// 当文件读取完成后，自动调用此函数:

}; js 在实际操纵的时候 其实是异步的

# Js的回调函数的理解

Js的函数作为一个变量传给函数

function a(callbackFunction){

alert("这是parent函数a");

var m =1;

var n=3;

return callbackFunction(m,n);

}

function b(m,n){

alert("这是回调函数B");

return m+n;

}

$(function(){

var result = a(b);

alert("result = "+ result);

});

执行顺序为：

这是parent函数a

这是回调函数B

result = 4

函数首先执行了主题函数a，之后调用了回调函数b，最后返回函数a的返回值。

# JQuery 的使用

什么是jQuery对象？jQuery对象类似数组，它的每个元素都是一个引用了DOM节点的对象

如果id为abc的<div>不存在，返回的jQuery对象如下：

[]

总之jQuery的选择器不会返回undefined或者null，这样的好处是你不必在下一行判断if (div === undefined)。

选择器

## 层级选择器

两个DOM元素具有层级关系，就可以用$('ancestor descendant')

<div class="testing">

<ul class="lang">

<li class="lang-javascript">JavaScript</li>

<li class="lang-python">Python</li>

<li class="lang-lua">Lua</li>

</ul>

</div>

ul.lang li.lang-javascript或者是 div.testing.li.lang-javascript

## 子选择器（Child Selector）

子选择器$('parent>child')类似层级选择器，但是限定了层级关系必须是父子关系，就是<child>节点必须是<parent>节点的直属子节点

过滤器（Filter）

一般不是单独使用 附加在选择器上帮助我们进行更加精准的定位元素

## 查找和过滤

通常情况下选择器可以直接定位到我们想要的元素，但是，当我们拿到一个jQuery对象后，还可以以这个对象为基准，进行查找和过滤。

最常见的查找是在某个节点的所有子节点中查找，使用find()方法

var ul = $('ul.lang'); // 获得<ul>

var dy = ul.find('.dy'); // 获得JavaScript, Python, Scheme

如果要从当前节点开始向上查找，使用parent()方法，对于位于同一层级的节点，可以通过next()和prev()方法

## 过滤

var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Scheme和Haskell

var a = langs.filter('.dy'); // 拿到JavaScript, Python, Schem

操作dom （修改html文件 ，css文件）

<ul id="test-ul">

<li class="js">JavaScript</li>

<li name="book">Java &amp; JavaScript</li>

</ul>

分别获取文本和HTML：

$('#test-ul li[name=book]').text(); // 'Java & JavaScript'

$('#test-ul li[name=book]').html(); // 'Java &amp; JavaScript'

要高亮显示动态语言，调用jQuery对象的css('name', 'value')方法

## 显示和隐藏DOM

要隐藏一个DOM，我们可以设置CSS的display属性为none，利用css()方法就可以实现。不过，要显示这个DOM就需要恢复原有的display属性，这就得先记下来原有的display属性到底是block还是inline还是别的值。

考虑到显示和隐藏DOM元素使用非常普遍，jQuery直接提供show()和hide()方法，我们不用关心它是如何修改display属性的，总之它能正常工作

var a = $('a[target=\_blank]');

a.hide(); // 隐藏

a.show(); // 显示

注意，隐藏DOM节点并未改变DOM树的结构，它只影响DOM节点的显示。这和删除DOM节点是不同的

attr()和removeAttr()方法用于操作DOM节点的属性

// <div id="test-div" name="Test" start="1">...</div>

var div = $('#test-div');

div.attr('data'); // undefined, 属性不存在

div.attr('name'); // 'Test'

div.attr('name', 'Hello'); // div的name属性变为'Hello'

div.removeAttr('name'); // 删除name属性

div.attr('name'); // undefined

操作表单

对于表单元素 jquery 对象同意提供val() 方法获取个设置value属性

input.val(); // 'test'

input.val('abc@example.com') // input 是之前获得属性