**原生JS部分基础思维导图：**

**JS:** 专门编写网页交互的语言

**交互：** 1，用户执行操作或者输入数据

2，程序接受用户输入的数据，相应的操作

3，程序返回的结果

**包含：** 1，ECMAScript

2，DOM：专门操作网页的内容API

3，BOM：专门操作浏览器窗口的API

**特点：** 1，运行在脚本脚本解释引擎中----多数情况运行在客户端浏览器----减少JS文件大小，提高下载速度

2，解释执行

3，弱类型：（1）声明变量时不需要提前制定变量的数据类型

（2）同一变量，先后可以保存不同的数据类型

（3）JS在运行时，根据自身需要，隐式转换数据类型

**变量：**内存中存储一个数据的存储空间，再起一个名字

何时使用：如果一个数据要反复使用，就将他保存在变量中

如何使用：声明、赋值、简写、取值

主流浏览器： 内核

IE trident

Chrome webkit/ blink

Firefox Gecko

Opera presto

Safari webkit

JS： 解释型语言(动态语言、脚本语言)

单线程

**JS基本语法**

变量： var 声明、赋值

原始值(栈数据)： Number、Boolen、String、undefined（未定义）、null

引用值(堆数据)： array、object、function、·· …date、RegExp

NaN不等于任何东西

逻辑运算符 || && ！

// undefined、null、NaN、‘ ’、0、false(转换成boolen) 🡺 false

&&： var a = 1 && 2 // 2

Var a = 0 && 2 // 0

Object：var obj = {}

typeof: number string boolean undefined **function** object(**null**、array ....)  
typeof 返回字符串类型 typeof(typeof(a)) ---> String   
显示类型转换，Number() **null**--0 undefined--nan  
parseInt(string, radix(基底) radix： 2-36  
parseFloat(string)  
.toString(radix) radix:以10进制为基底转换成目标进制  
String()  
Boolean()  
隐式类型转换  
isNaN() : 先将里面的数Number显式转换 ---> Number()  
++/--: ---> Number()  
+/-(正/负)：---> Number()  
+(加号)：--->String()  
-/\*% ---> Number()  
==/!= ---> Number()  
不发生类型转换： ===/!==

/\*  
高内聚，低耦合  
函数  
\*/  
function sum(a, b, c, d) {  
 // arguments -- [1, 2] 实参列表  
 a = 2; // arguments[0] = 2  
 arguments[0] = 3; // a = 3  
 // arguments与实参一起变化,映射关系，  
 // 但是实参列表少于arguments时，每对应的arguments[sum.length]没有映射关系  
 // c = 2; --> arguments[2] = undefined  
}  
sum(1, 2)

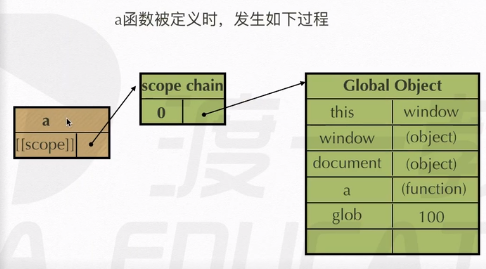
// 作用域：全局变量，局部变量  
// JS三部曲：语义分析、预编译、解释执行  
// 预编译: 函数声明整体提升、变量 声明提升  
// 1，暗示全局变量(imply global)：即任何变量，如果变量未经声明就赋值，  
// 此变量就为全局对象所有  
// 2，一切声明的全局变量，全是iwindow的属性  
// 预编译: 预编译发生在函数执行的前一刻  
// 函数体预编译4步：  
// 1，创建AO对象（activation object，执行期上下文）// GO对象  
// 2，找形参和变量声明，将形参和变量明作为AO对象的属性名，值为undefined  
// 3，将实参值和形参统一  
// 4，在函数体里面找函数声明，值赋予函数体  
function fn(a) {  
 console.log(a); // function a() {}  
 var a =123;  
 console.log(a); // 123  
 function a() {}  
 console.log(a); // 123  
 var b = function () {} // 函数表达式  
 console.log(b); // function () {}  
 function d() {}  
}  
fn(1);  
// AO {  
// a: undefined -->1 --> function a() {}  
// b: undefined -->function () {}  
// d: undefined --> function d() {}  
// }  
console.log(test); // function test() {...}  
function test(test) {  
 console.log(test); // function test() { }  
 var test = 234;  
 console.log(test); // 234  
 function test() { }  
}  
test(1);  
var test = 123;  
  
/\*GO{  
 a: undefined ---> 10,  
 c: 234  
}\*/  
function test() {  
 console.log(b); // undefined  
 if(a) {  
 var b = 100;  
 }  
 console.log(b); // undefined  
 c = 234;  
 console.log(c); // 234  
}  
var a;  
test();  
/\*AO{  
 b: undefined  
}\*/  
a = 10;  
console.log(c); // 234

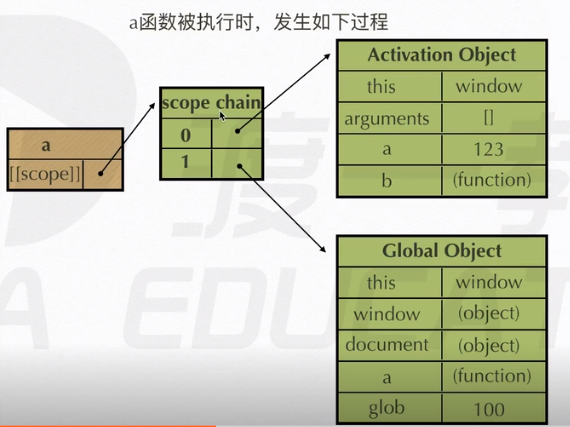
例题：

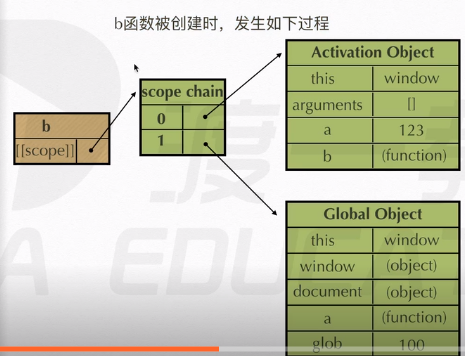
var str = false + 1;  
console.log(str); //1  
var demo = false == 1;  
console.log(demo); // false  
if(typeof(a)&&-true + (+undefined) + '') {  
 console.log('基础不错') // '基础不错'  
}  
if(11 + '11' \* 2 == 33) {  
 console.log('基础不错') //'基础不错'  
}  
!!'' + !!'' - !!false || console.log("can 打印吗") // "can 打印吗"

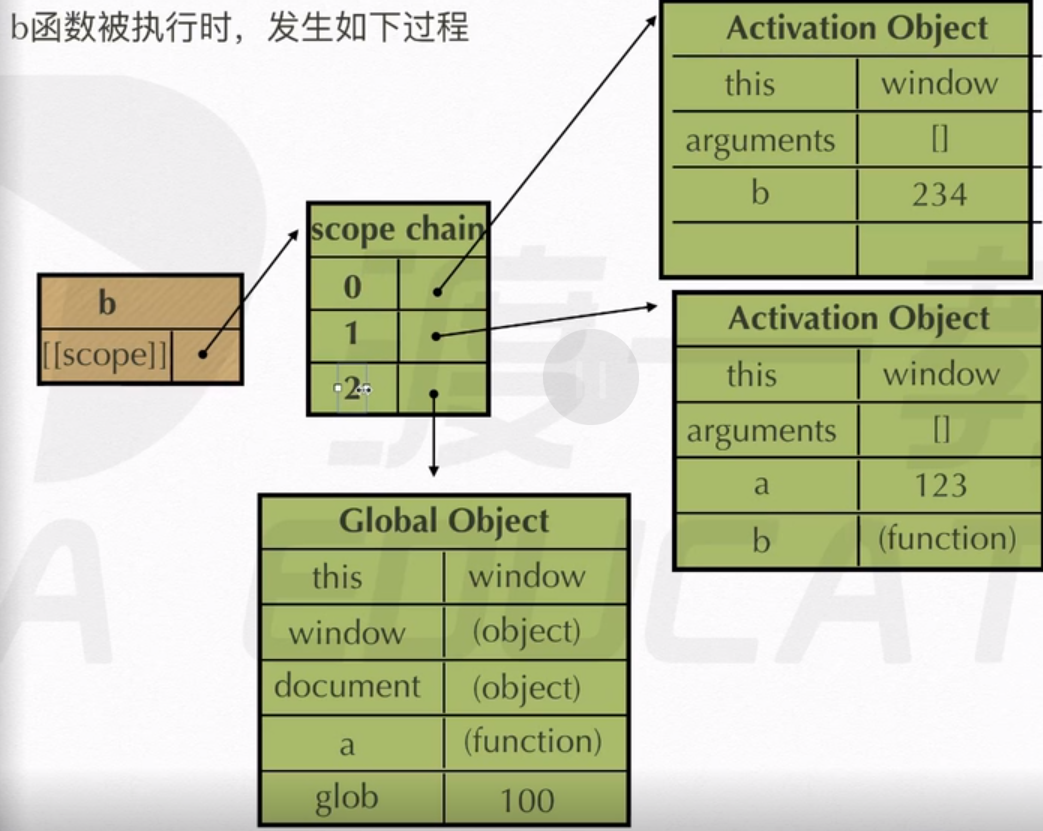
**作用域，作用域精讲：**

// [[scope]]: 每个JavaScript函数都是一个对象，对象中有些属性我们可以访问，但是有些不可以，不可以的属性仅供JavaScript引擎存取，[[scope]]就是其中一个。  
// [[scope]]：指的是我们所说的作用域，其中存储了运行期上下文的集合。  
// 执行期上下文（AO,GO）：当函数执行时，会创建一个称为执行期上下文的内部对象。  
// 作用域链：[[scope]]中所存储的执行器上下文对象的集合，这个集合呈链式连接---即作用域链。  
// 查找变量：从作用域链的顶端依此向下查找。  
function a() {  
 function b() {  
 var b = 234;  
 }  
 var a = 123;  
 b();  
}  
var glob = 100;  
a();









**立即执行函数 闭包**

function a() {  
 var num = 100;  
 function b() {  
 num++;  
 console.log(num);  
 }  
 function c() {  
 num--;  
 console.log(num);  
 }  
 return [b, c];  
}  
var demo = a();  
demo[1](); // 99  
demo[0](); // 100

**闭包：**当内部的函数被保存到外部，就会生成闭包。闭包会导致原有的作用域链不释放，造成内存泄漏。

**闭包作用：**

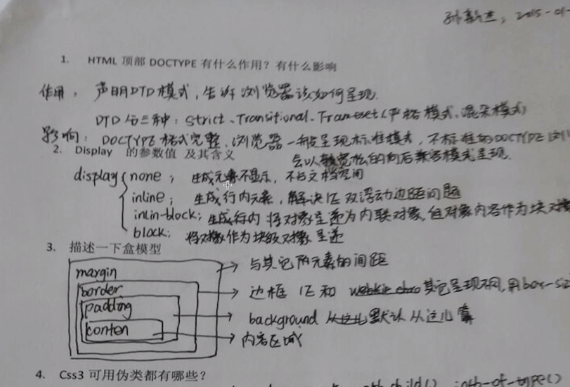
**立即执行函数：此类函数没有声明，再一次执行后即释放。适合做初始化工作**

// 写法  
(function () {}()) //建议第一种  
(function () {})()  
// 只有表达式才能被执行符号()执行  
// 被执行符号执行的表达式放弃表达式的名字

(function adc(a, b, c) {  
 var a = 123;  
 console.log(a + b + c);  
}(1, 2, 3)) ; // 6

function adc(a, b, c) {  
 var a = 123;  
 console.log(a + b + c);  
}(); // 报错

function adc(a, b, c) {  
 var a = 123;  
 console.log(a + b + c);  
}(1, 2, 3); // 不报错也不输出



**对象与包装类**

对象的创建方法：  
1， var obj = {} 对象字面量/对象直接量  
2， 构造函数：

（1）系统自带的构造函数 new Object() var obj = new Object()  
（2）自定义 function Person() {} var per = new Person();

构造函数内部原理：  
1，在函数体最前面隐式的加上this={}  
2，执行this.xxx = xxx;  
3，隐式的返回this  
包装类: new Number(), new String(), new Boolean()  
// var str = 'abcd';  
// str.length = 2;  
// console.log(str) // abcd  
var str = 'abc';  
str+=1;  
var test = typeof(str); // "String"  
if(test.length == 6){  
 test.sign = 'typeof的返回结果可能为String';  
 // new String(test).sign = 'typeof的返回结果可能为String' delete   
}  
// new String(test).sign  
console.log(test.sign); // undefined

// 原型 prototype  
// 1，定义：原型是function对象的一个属性，它定义了构造函数制造出的对象的公共祖先。  
// 通过该构造函数产生的对象，可以继承改原型的对象和方法。原型也是对象  
// 2，利用原型的特点和概念，可以提取共有属性  
// 3，对象如何查看原型 --> 隐式属性 \_\_proto\_\_  
// 4，对象如何查看对象的构造函数 --> constructor  
// 原型链  
// 1, 绝大多数对象的最终都会继承自 Object.prototype

特例：Object.create(null)

// 2, Object.create(原型)  
var obj = {name: 'sunny'};  
var obj1 = Object.create(obj);  
// a.sayName() sayName里面的this指向谁，谁调用的这个方法，this就是指向谁  
Person.prototype = {  
 name: 'a',  
 sayName: function () {  
 console.log(this.name)  
 }  
}  
function Person() {  
 this.name = 'b'  
}  
var person = new Person(); // b

call/apply  
作用：改变this的指向  
区别: call: 需要把实参按照形参的个数传进去  
 apply： 需要传一个arguments

// **继承模式**  
// 1，传统模式 ---> 原型链： 过多继承了没用的属性  
// 例：  
function Grand() {}  
var grand = new Grand();  
fathor.prototype = grand;  
function Father() {}  
var father = new Father();  
sun.prototype = father;  
function Son() {}  
var son = new Son();  
  
// 2，借用构造函数： (1) 不能继承借用构造函数的原型 （2）每次构造函数都要多走一个函数  
// 例：  
function Person(name, age, sex) {  
 this.name = name;  
 this.age = age;  
 this.sex = sex;  
}  
function Student(name, age, sex, grade) {  
 Person.call(this, name, age, sex); // Person.apply(this, [name, age, sex])  
 this.grand = grade;  
}  
var student = new Student();  
  
// 3，共享原型：不能随便改动自己的原型  
// 例：  
Father.prototype.lastName = "li";  
function Father() {}  
function Son() {}  
Son.prototype = Father.prototype;  
// 4，圣杯模式  
// 例：  
function inherit(Target, Origin) {  
 function F() {}  
 F.prototype = Origin.prototype;  
 Target.prototype = new F();  
 Target.prototype.constructor = Target; //  
 Target.prototype.uber = Origin.prototype; //超类  
}  
Father.prototype.lastName = "li";  
function Father() {}  
function Son() {}  
inherit(Son, Father);  
var son = new Son();  
var father = new Father();  
// son.\_\_proto\_\_ --> new F().\_\_proto\_\_ --> Father.prototype

命名空间：管理变量，防止污染全局，适用于模块化开发。  
对象的枚举：1，for in 2，hasOwnProperty 3，in 4，instanceof: 看A的原型链上有没有B的原型

/\*this  
 1，函数预编译过程this ---> window  
 2，全局作用域里面this ---> window  
 3，call、apply可以改变函数运行时this的指向  
 4，obj.func(); func()里面的this指向obj\*/  
 **var** name = '222';  
 **var** a = {  
 name: '111',  
 say: **function** () {  
 console.log(**this**.name)  
 }  
 }  
 **var** fun = a.say; // 此时 fun = function () {console.log(this.name)}  
 fun(); // 222  
 a.say(); // 111  
  
 **var** b = {  
 name: '333',  
 say: **function** (fun) {  
 fun();  
 }  
 }  
 b.say(a.say); // 222  
 b.say = a.say;  
 b.say(); // 333  
  
// arguments  
// 1, arguments.callee : 指函数本身  
// 2, fun.caller  
  
 **var** a = 5;  
 **function** test() {  
 a = 0;  
 console.log(a);  
 console.log(**this**.a);  
 **var** a;  
 console.log(a);  
 }  
 test(); // 0 5 0  
 **new** test(); // 0 undefined 0 此时this = { \_\_proto\_\_: test.prototype}

// 克隆 : 浅、深克隆  
 **var** obj = {  
 name: 'abc',  
 age: 123,  
 card: ['visa', 'master'],  
 wife: {  
 name: 'bcd',  
 son: {  
 name: 'aaa'  
 }  
 }  
 }  
 /\*遍历对象 for(var prop in obj)  
 1, 判断是不是原始值 typeof(Object)  
 2, 判断是数组还是对象 instanceof toString constructor  
 3, 建立相关的数组或者对象  
 4, 递归\*/  
 **function** deepClone(origin, target) {  
 **var** target = target || {},  
 toStr = Object.prototype.toString,  
 arrStr = ['object Array'];  
  
 **for** (**var** prop in origin){  
 **if**(origin.hasOwnProperty(prop)){  
 **if**(origin[prop] !== 'null' && typeof(origin[prop]) == 'object') {  
 **if**(toStr.call(origin[prop]) == arrStr) {  
 target[prop] = [];  
 }**else** {  
 target[prop] = {};  
 }  
 deepClone(origin[prop], target[prop]);  
 }**else** {  
 target[prop] = origin[prop];  
 }  
 }  
 }  
 **return** target;  
 }

/\*数组常用的方法  
改变数组； push,pop,shift,unshift,sort,reverse,splice  
不改变原始数组：concat,join（数组） --> split（字符串）(互逆),toString,slice  
\*/  
Array.prototype.push = **function** () {  
 **for**(**var** i = 0; i < arguments.length; i++) {  
 **this**[**this**.length] = arguments[i]  
 }  
 **return this**.length;  
}  
**var** arr = [1, 1, 2, 3, 2, 10];  
// arr.splice(从第几位开始，截取多少长度，在切口处添加新的数据)  
arr.sort(**function** (a, b) {  
 **return** a-b; // 升序  
 // return b-a; // 降序  
})  
// 给一个有序的数组，乱序  
**var** array = [1,2,3,4,5,6,7];  
arr.sort(**function** (a,b) {  
 **return** Math.random()-0.5 // 返回随机数  
})  
  
// 类数组: 属性要为索引（数字）属性，必须有length属性，最好加上push方法  
**var** obj = {  
 '2': 'a',  
 '3': 'b',  
 'length': 2,  
 'push': Array.prototype.push  
}  
obj.push('c');  
obj.push('d');  
console.log(obj) // {'2': 'c', '3': 'd', 'length': 4}  
  
/\*typeof 函数\*/  
**function** MyType(target) {  
 **var** template = {  
 '[object Array]': 'array',  
 '[object Object]': 'object',  
 '[object Number]': 'number-object',  
 '[object Boolean]': 'boolean-object',  
 '[object String]': 'string-object',  
 };  
 **if**(target === **null**){  
 **return null**;  
 }  
 **if**(typeof(target) === 'object') {  
 // 数组、对象、包装类（Object.prototype.toString）  
 **var** str = Object.prototype.toString.call(target);  
 **return** template[str];  
 }**else** {  
 **return** typeof(target)  
 }  
}  
  
/\*数组去重\*/  
Array.prototype.unique = **function** () {  
 **var** tem = {}, arr = [];  
 **var** len = **this**.length;  
 **for**(**var** i = 0; i < len; i++){  
 **if**(!tem[**this**[i]]){  
 tem[**this**[i]] = "abc";  
 arr.push(**this**[i]);  
 }  
 }  
 **return** arr;  
}

// 在try里面的发生错误，不会执行错误以后的try里面的代码  
try{}catch(e){} // error error.message error.name  
// error.name对应的信息：  
// 1，EvalError: eval()的使用与定义不一致  
// 2，RangeError：数组越界  
// 3，ReferenceError：非法或者不能识别的引用数值  
// 4，SyntaxError：语法解析错误  
// 5，TypeError：操作数类型错误  
// 6，URLError：URL处理函数使用不当  
  
/\*  
es5严格模式的启用 "use strict"  
浏览器基于es3.0 + es5.0的新增方法使用的  
es3.0 和es5.0产生冲突的部分  
es5.0严格模式,那么es3.0和es5.0产生冲突的部分就用es5.0，否则会有es3.0  
\*/

// DOM操作  
// 封装函数insertAfter();功能类似insertBefore(a, b)  
Element.prototype.innerAfter = function (targetNode, afterNode) {  
 var beforeNode = afterNode.nextElementSibling;  
 if (beforeNode === null){  
 this.appendChild(targetNode);  
 }else{  
 this.insertBefore(targetNode, beforeNode);  
 }  
}  
//定时器  
var timer1 = setInterval(function () {}, 1000); // 循环器  
clearInterval(timer1);  
var timer2 = setTimeout(function () {}, 100) // 运行一次 ==>  
clearTimeout(timer2)

事件  
1，ele.onxxx = function (event) {}  
2, obj.addEventListener(type, fun, false) ==》 removeEventListener()  
 false----冒泡 true-----捕获  
3，obj.attachEvent('on'+type, fun) // IE独有  
  
事件处理模型  
冒泡： 结构上嵌套关系的元素，子元素冒泡向父元素（自底向上）  
捕获：结构上嵌套关系，父元素捕获至子元素（自顶向下）。  
IE上面没有捕获事件  
顺序：先捕获，后冒泡

时间对象 e  
取消冒泡：（1）W3C标准：event.stopPropagation()  
 （2）IE独有：event.cancelBubble  
封装取消冒泡的函数 stopBubble(event)  
function stopBubble(event) {  
 event = event || window.event;  
 event.stopPropagation ? event.stopPropagation() : event.cancelBubble = true;  
}  
阻止默认事件：默认事件 -----> 表单提交，a标签跳转等  
(1)return false: 以对象属性的方式注册的事件才生效  
(2)event.preventDefault(): W3C标注  
(3)event.returnValue = false: 兼容IE  
封装阻止默认事件的函数cancelHandler(event)  
function cancelHandler(event) {  
 event = event || window.event;  
 if(event.preventDefault){  
 event.preventDefault();  
 }else{  
 event.returnValue = false;  
 }  
}  
事件委托：利用事件冒泡和事件源对象进行处理  
优点：（1）性能不需要循环所有的元素一个个绑定事件  
 （2）灵活 当有新的子元素时不需要重新绑定事件

/\*  
JSON：是一种数据传输的格式 ---> 对象：{}, 属性名加引号  
JSON.parse(); string ---> json  
JSON.stringify(); json ---> string  
  
重排（reflow），重绘（repaint）: 重排必定会引发重绘，但重绘不一定会引发重排。  
\*/  
  
异步加载JS  
JS异步加载的三种方案：  
1，defer 异步加载，但是需要等到dom文档全部解析完才会被执行。只有IE能用，也可以将代码写到内部。

<script type="text/javascript" src="js/demo.js" defer="defer"></script>  
2，async 异步加载，加载完就执行，async只能加载外部脚本，不能把js写在script标签里面  
 <script type="text/javascript" src="js/demo.js" async="async"></script>  
1，2执行时不阻塞页面  
3，创建script，插入到DOM中，加载完毕后callBack ---可实现异步加载和按需加载

// 3 封装异步加载JS

function loadScript(url, callBack) {  
 var script = document.createElement('script');  
 script.type = "text/javascript";  
 if(script.readyState){  
 script.onreadystatechange = function () {  
 // IE的script只有onreadystatechange（），别的标签都有onload  
 if(script.readyState === 'complete' || script.readyState === 'loaded') {  
 callBack();  
 }  
 }  
 }else {  
 script.onload = function(){  
 // Safari、chrome、firefox、opera的script标签才有onload  
 callBack();  
 }  
 }  
 script.src = url; // js/demo.js  
 document.head.appendChild(script);  
}  
loadScript("js/demo.js", function () {  
 test();  
})