**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)

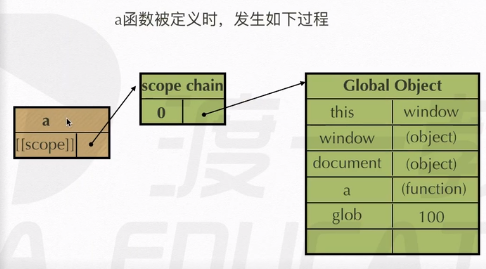
// 作用域：全局变量，局部变量  
// JS三部曲：语义分析、预编译、解释执行  
// 预编译: 函数声明整体提升、变量 声明提升  
// 1，暗示全局变量(imply global)：即任何变量，如果变量未经声明就赋值，  
// 此变量就为全局对象所有  
// 2，一切声明的全局变量，全是iwindow的属性  
// 预编译: 预编译发生在函数执行的前一刻  
// 函数体预编译4步：  
// 1，创建AO对象（activation object，执行期上下文）// GO对象  
// 2，找形参和变量声明，将形参和变量明作为AO对象的属性名，值为undefined  
// 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

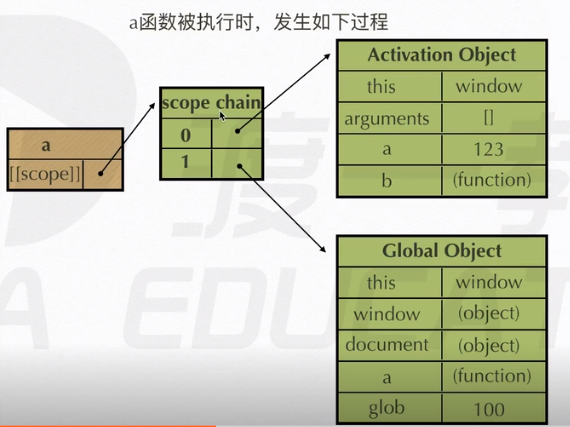
例题：

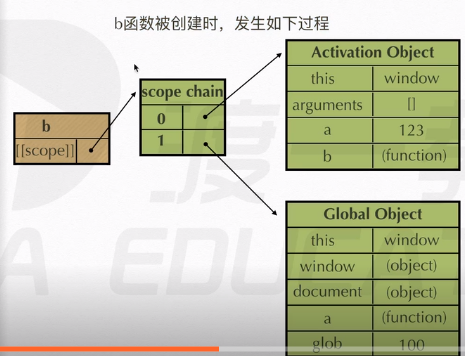
var str = false + 1;  
console.log(str); //1  
var demo = false == 1;  
console.log(demo); // false  
if(typeof(a)&&-true + (+undefined) + '') {  
 console.log('基础不错') // '基础不错'  
}  
if(11 + '11' \* 2 == 33) {  
 console.log('基础不错') //'基础不错'  
}  
!!'' + !!'' - !!false || console.log("can 打印吗") // "can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

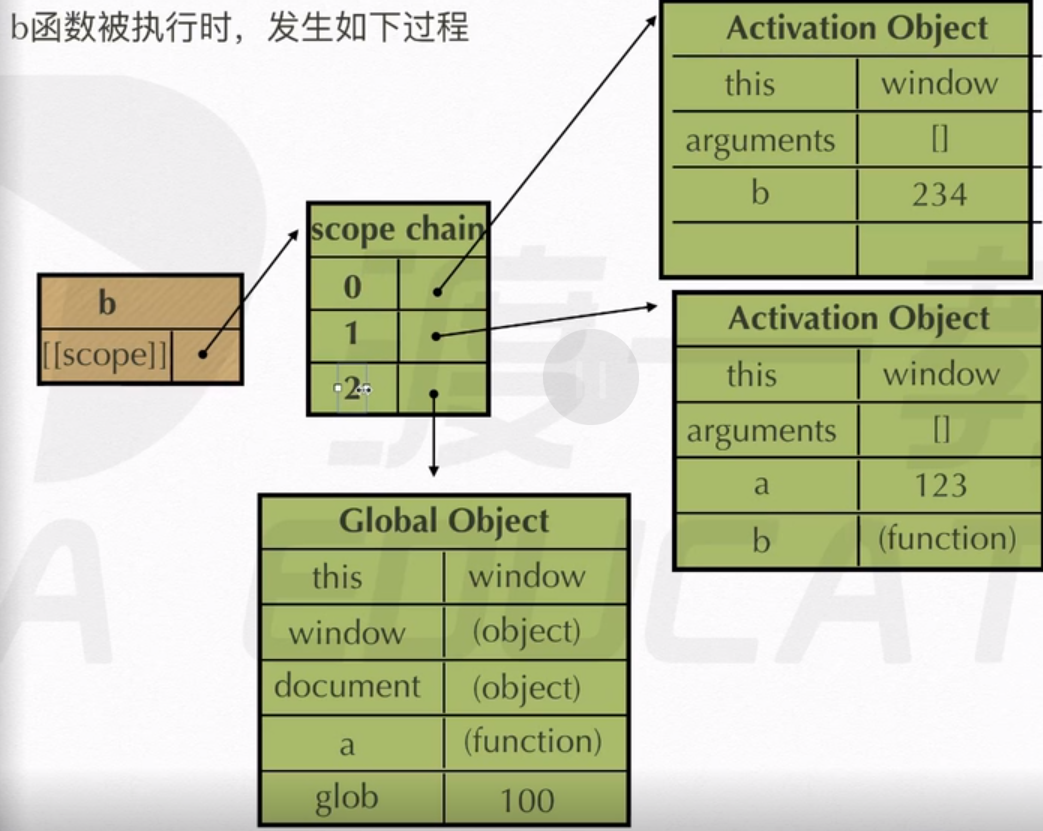
**作用域，作用域精讲：**

// [[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：**

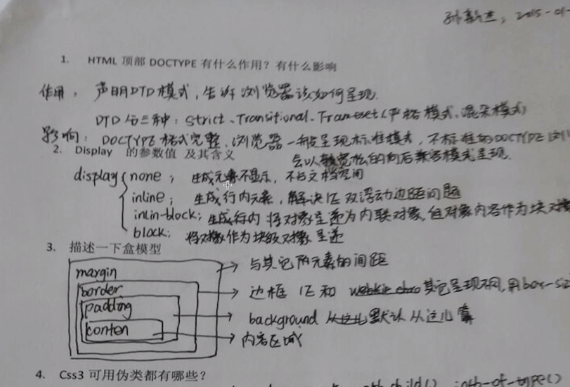
**立即执行函数：此类函数没有声明，再一次执行后即释放。适合做初始化工作**

// 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