**第一章**

JS不止可以在浏览器，也可在node.js

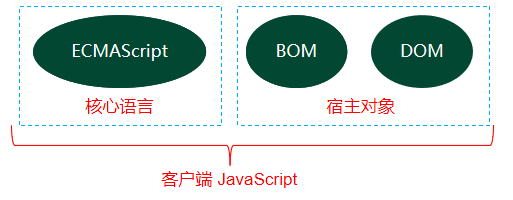
JS在浏览器中如何运行？

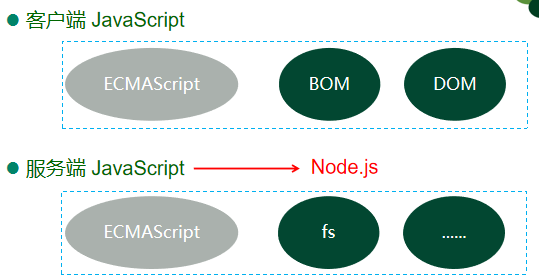
浏览器下载JS脚本文件后，由JS引擎解释执行

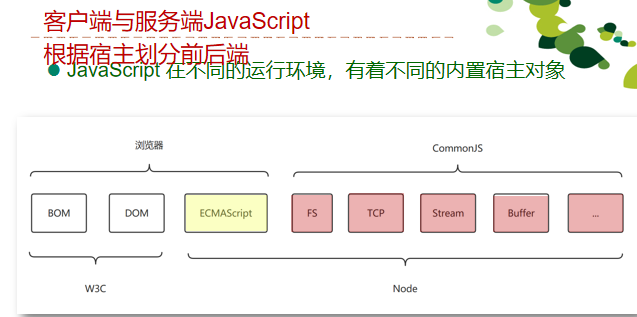
JS核心和DOM不是不可分割的，他们的标准相互独立。

DOM 对 JavaScript 来说，是宿主对象，是语言中可更换的部分

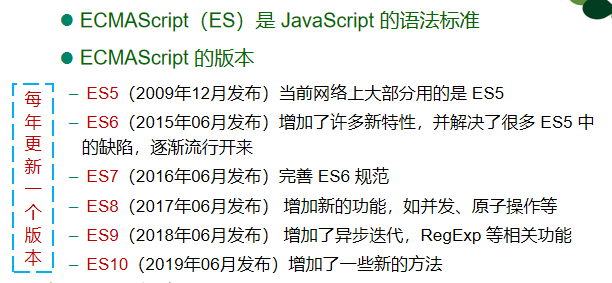
ECMAScript 对 JavaScript 来说，是核心语言，是不可被替代的功能







ECMA是JS的语法标准



JS是一种直译式脚本语言

在宿主（浏览器、node）中解释执行（非编译语言，不是在执行前提前编译可执行文件或字节码）

语言比较：c++,c的运行效率高于JS JS开发效率高于c

JS是一种弱类型、动态型语言

写程序时不用给变量指定特定的数据类型；可以动态的更改变量的类型

语言特点：ES5没有块作用域、函数式编程、闭包、基于原型链的继承方式、动态添加属性等；借鉴了Java的语法、Self原型继承、Python的正则等

**第二章**

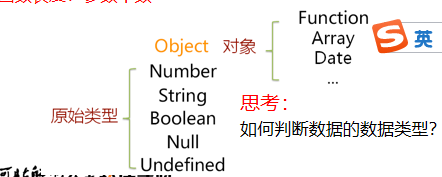
ES5数据类型（6种）及其划分（2类）

基本（原始）类型：Number,String,Boolean,Null,Undefined.

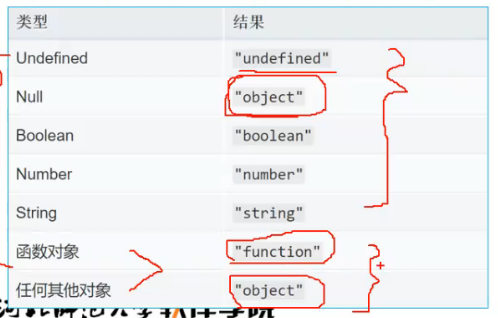
引用（对象）类型：Object(Array,Function,Date)

对象——一个单独拥有属性和方法的实体

函数长度：参数个数



typeof操作符，表示未经计算的操作数的类型



**数据、变量和类型**

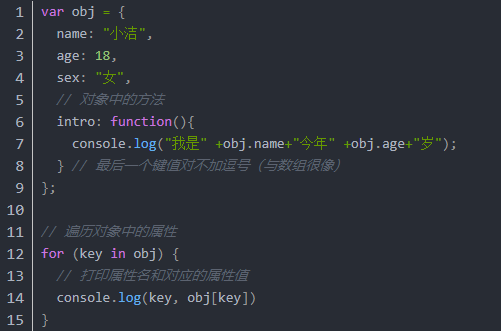
JavaScript 中的变量没有类型

JavaScript 中的数据值有类型

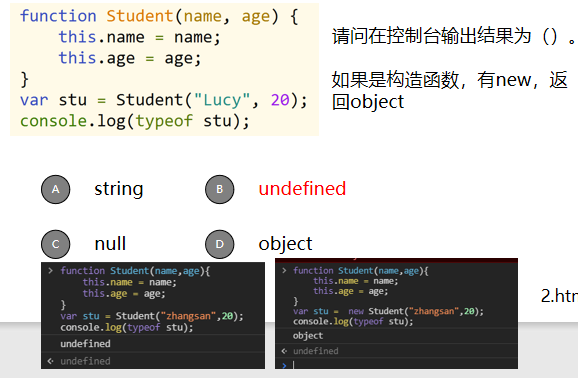
变量可以随时持有任何类型的值

对变量执行typeof操作，得到结果不是变量的类型而是变量所拥有的值的类型。

遍历一个对象的属性：for in







变量声明：

使用方便的标识符，用于引用计算机内存地址

变量声明指向一块内存空间，用于保存数据

变量赋值

向变量指向的内存空间存放数据

一般系统会划分出两种不同内存空间

栈内存；堆内存

**栈内存**：存储的值大小固定

由系统自动分配内存空间

空间小，运行效率高

**堆内存**：存储的值大小不定，可动态调整

由程序员通过代码进行分配

空间大，运行效率相对较低

**基本类型与引用类型的区别：**

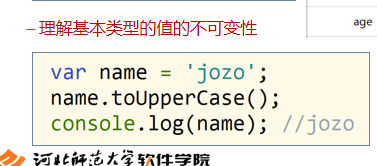
栈区与堆区、存值与存地址、影响变量的生命周期。

基本数据类型的临时变量分配在栈区；

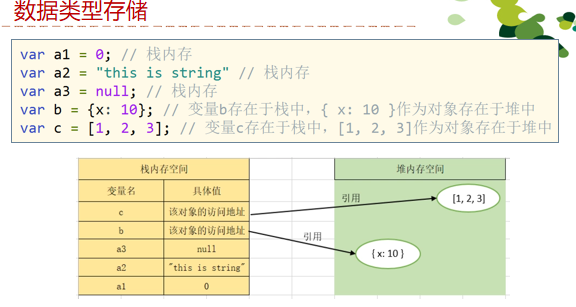
引用数据类型的变量的引用（地址）存储在栈区，被引用（指向）的对象在堆区

*栈区常用来存储函数局部临时变量，一般数据量较小*

*堆区常用来存储更为复杂的数据结构的对象*







**基本类型与引用类型的区别：**

*1）访问机制不同*

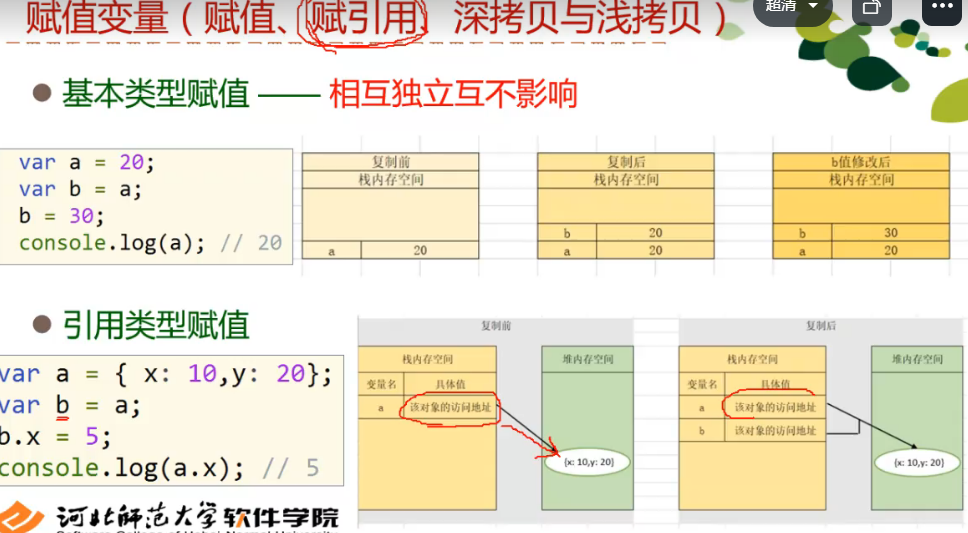
基本类型的值直接访问

引用类型的值通过引用访问，不能直接访问

首先，从栈中获取该对象的地址引用

其次，再从堆内存中取得我们需要的数据

*2）变量赋值不同*



*3）比较变量不同*

基本数据类型与引用类型的比较

值类型是判断变量的值是否相等（值比较）

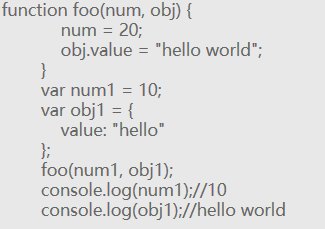
引用类型是判断所指向的内存空间（地址）是否相同（引用比较）

*4）参数传递不同*

ECMAScript 中所有函数的参数都是按值来传递的

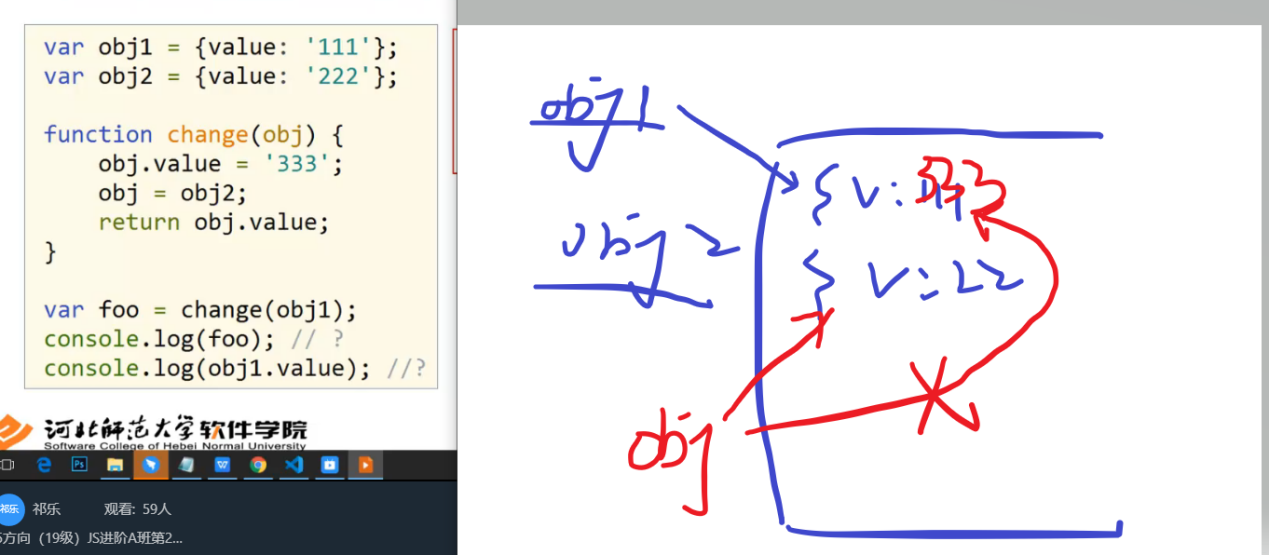
基本类型值：把变量里的数据值传递给参数，之后参数和变量互不影响。

引用类型值：把对象的引用（地址）值传递给参数，参数和对象都指向同一个对象，相互影响。

\*\*因为num1是基本类型，不改变值，所以还是全局变量下的10

PPT练习



222

333

（1）Obj1—>{value:’111’}

Obj2—>{value:’222’}

1. 经过change；obj1.value变成333；即obj1—>{value:333};

Obj—>{value:’222’};

1. 所以console.log(foo);是函数返回值obj.value=222；console.log(obj1.value)=333；

**类型转换 —**— 将值从一种类型转换为另一种类型

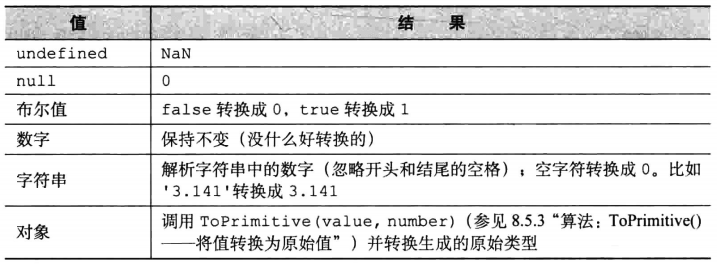
隐式类型转换：通常是某些操作的副作用，不易看出

显示类型转换：可以在代码中明显看出

**隐式类型转换**- 使用关系运算符时的转换（==、>、<、引用类型和基本类型比较时）  
- 使用算数运算符时的转换（'img'+ 3 + '.jpg'; “25”-0;）  
- 使用逻辑运算符时的转换（ !!0; ）  
- 执行流程语句时的转换（if(obj){...}）

**显式类型转换（使代码更清晰）**- Boolean（）、Number（）、String（）、Object（）  
- 数转为字符串（toString()、toFixed()、toPrecision()、toExponential()）  
- 字符串转为数字（parseInt()、parseFloat()）  
- 对象转换为原始值（toString()、valueOf()）

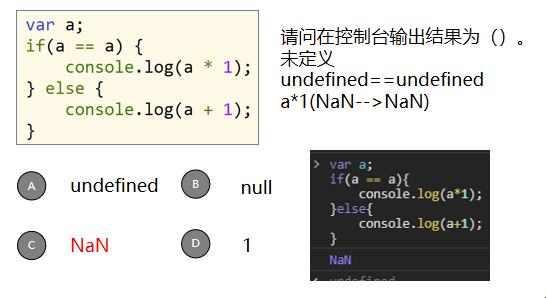
**转number**



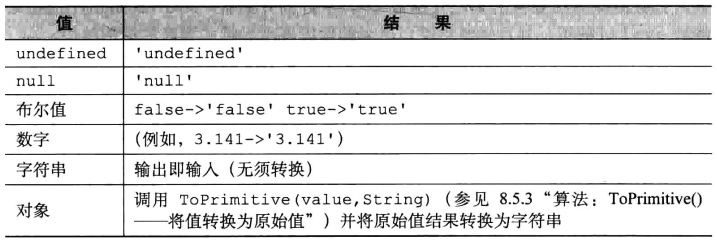
**NaN(Not a Number)**

**NaN != NaN**

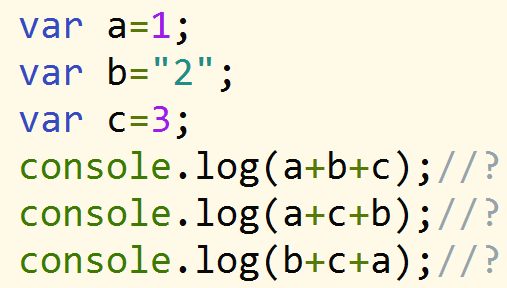
**isNaN();参数是NaN返回true，否则返回false**



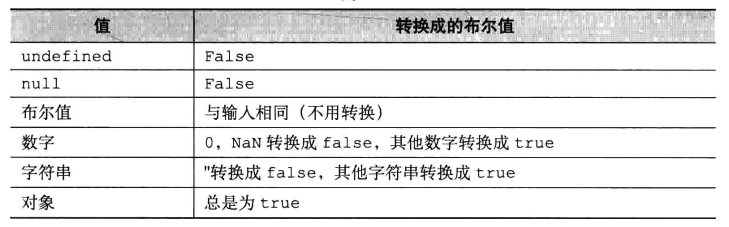
**转String**

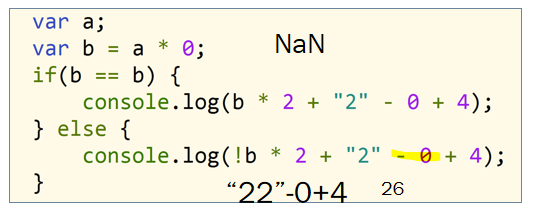


1. **”+”左右两侧有字符串时为拼接运算符**
2. **运算符等级相同时，从左往右计算**



**转Boolean**





\*\*a未定义；b=a\*0;所以b是NaN，走else,!b=1;所以”22”-0+4;

所以”22”-0变成数字22；所以+4=26

**第三章**

ES5数据类型（6种）及其划分（2类）

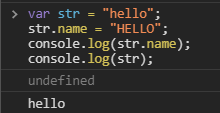
基本（原始）类型：Number,String,Boolean,Null,Undefined.

引用（对象）类型：Object(Array,Function,Date)

**包装对象（.length/.toUpperCase()）**

-存取字符串、数字或布尔值的属性时创建的临时对象称为包装对象

-用来处理属性的引用，一旦属性引用结束，包装对象就会销毁



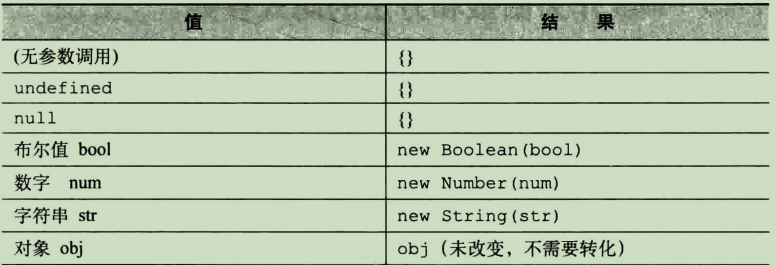
包装类型和引用类型的区别：

主要区别：对象的生存期；字符值和字符串对象；数字值和数值对象；布尔值和布尔对象

临时对象在使用之后立即释放

**数据类型转换（转换为Object类型）**

1. 对象转换为自身
2. Undefined和null转换为空对象{}
3. String/number/boolean转换为包装对象
4. 强制转换：Object()



**Object转Number**

先valueOf()；后toString()；最后Number()

Var a =[];——>NaN

Var a=[2];——>2

Var a=[2,3]——>NaN

**object转String**

ToString();

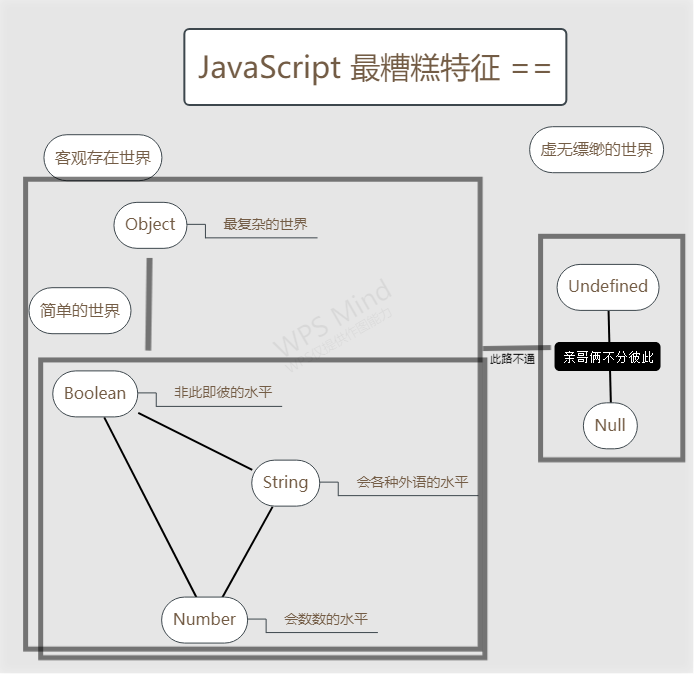
Var a={};——>”[object,object]”

Var a=[1,2,3]——>”1,2,3”

Var a = function(){}——>”function(){}”

**Object转boolean**

任意值转换为布尔类型均为true



**分析 console.log([] == []) 输出的值**

两个值都是对象 (引用值) 时，比较的是两个引用值在内存中是否是同一个对象。 虽然左操作数和右操作数同为空数组， 但此 [] 非彼 []，在内存中是两个互不相关的空数组， 所以结果为 false。

**分析 console.log([] == ![]) 输出的值**

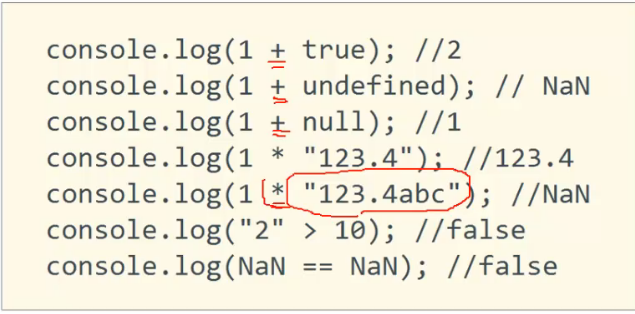
涉及到了 JavaScript 的运算符优先级 、宽松相等（即 ==）的判断过程以及类型转换

1. 等号右边有 ! ，优先级比 == 更高，优先计算右边的结果。 [] 为非假值，所以右边的运算结果为 false，即：![] ==> false

2. == 的两边分别是 object 和 boolean 类型的值把 object 转换成 number 类型，需要对 object 进行 ToNumber 操作，即 Number([].valueOf()) ==> 0

boolean 类型的值时先把这个值转换成 number 类型，右边转换成了 0，即Number(false) ==> 0





**第四章**

标识符是代码中用来标识变量、函数、或属性的字符序列

命名规则：

由字母、数字、下划线和$符号组成

不能以数字开头

大小写敏感（区分大小写）

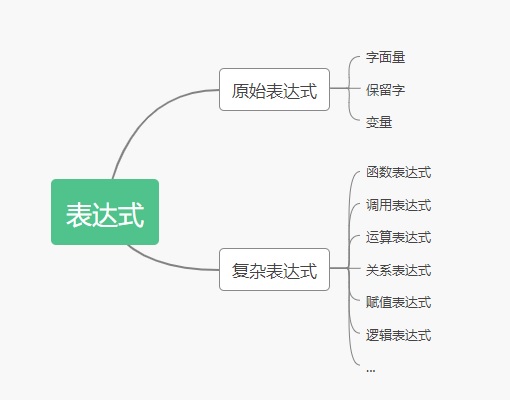
不与关键字同名

**访问对象属性方式：**

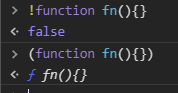
（.）点号后面要求是合法标识符，对于不合法的不可以使用

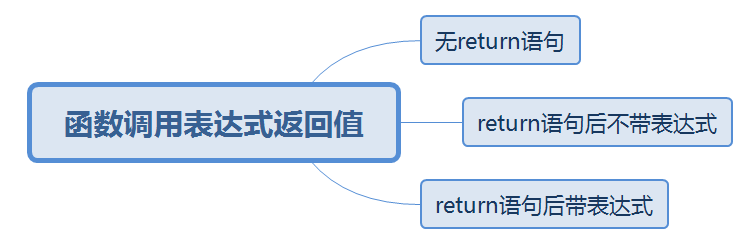
（[]）中括号要求是一个字符串即可，不必是合法标识符

字面量：表示自身的常量。



**Function前有任何其他元素，就会从函数声明转变为函数表达式**





**立即执行表达式 IIFE**

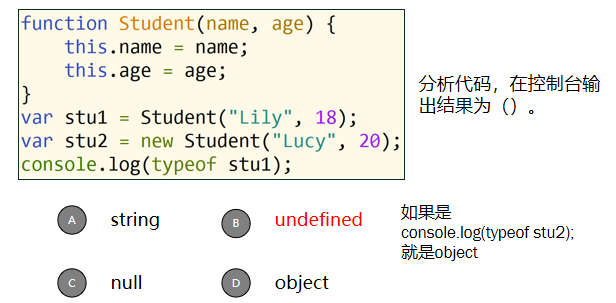
****在 function 前面加！、+、 - 甚至是逗号等到都可以起到识别为函数表达式的效果****

****在这些运算符中加括号是最安全的做法，因为它不会改变函数的返回值。****

****匿名函数为什么要加一对小括号呢？这样使用有什么好处么？****

1．在function前面加！、+、 - 甚至是逗号等到都可以起到函数定义后立即执行的效果，而（）、！、+、-、=等运算符，****都将函数声明转换成函数表达式，消除了javascript引擎识别函数表达式和函数声明的歧义****，告诉javascript引擎这是一个函数表达式，不是函数声明，可以在后面加括号，并立即执行函数的代码。

2．函数是有返回值的，返回我们指定的值或者undefined。函数的返回值我们可能需要在后续中使用。在函数前面加上不同的运算符，这些运算符还会和函数的返回值进行运算，有时造成不必要的麻烦。****在这些运算符中加括号是最安全的做法，因为它不会改变函数的返回值。****



**逻辑运算符两边的操作数都是布尔类型**

对于&&来说， 除了两侧都为真时为真，其他情况都为假

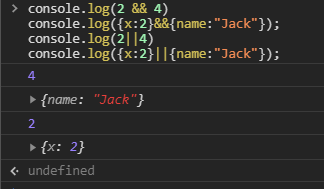
对于 | | 来说， 除了两侧都为假时为假，其他情况都为真

**当逻辑运算符 && 和 || 两侧的操作数不是布尔类型时**

1）首先将左操作数转换成布尔类型

2）对转换后的左操作数进行逻辑判断（true or false）

3）根据短路原则返回原始左操作数或原始右操作数



短路原则（忽略对右操作数的判断）

对于 &&，转换后的左操作数若为 true，则直接返回原始右操作数，若为 false 则直接返回原始左操作数

对于 | |，转换后的左操作数若为 true，则直接返回原始左操作数，若为 false 则直接返回原始右操作数

**比较运算符：**

严格相等运算符（===）

仅当两个操作数的类型相同且值相等为true

宽松相等运算符（==）

在进行比较之前，将两个操作数转换成相同的类型

**递增 (++)**

递增运算符为其操作数增加1，返回一个数值

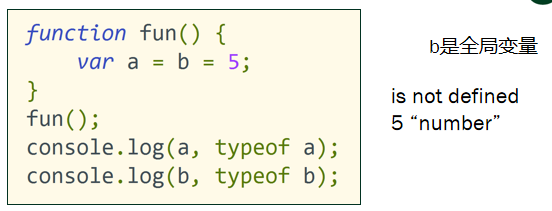
如果使用后置（postfix），即运算符位于操作数的后面（如 x++），那么将会在**递增前返回数值**

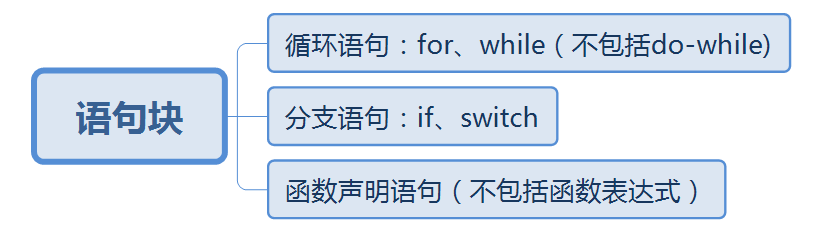
如果使用前置（prefix），即运算符位于操作数的前面（如 ++x），那么将会在递**增后返回数值**

递减（--）

递减运算符为其操作数减去1，返回一个数值

**赋值表达式的返回值为右操作数**





**第五章**

**函数定义和调用形式**

函数是可以通过外部代码调用的一个“子程序”。

一个函数由称为函数体的一系列语句组成。值可以传递给一个函数，函数将返回一个值。

函数定义方式：

通过函数声明形式定义

通过函数表达式形式定义

通过Function构造函数实例化的形式来定义

Function构造函数

可以传入任意数量的实参

最后一个实参为函数体

函数体中 javascript 语句之间分号隔开

Function 构造函数创建一个匿名函数

函数定义三要素：

函数名；函数参数；函数返回值

**匿名函数：**

单独的匿名函数无法运行；可以把匿名函数赋值给变量或立即执行

**具名函数：**

当遇到错误时，堆栈跟踪会显示函数名，容易寻找错误

**代理函数名：**

是可有可无的"代理"函数名

代理函数名的作用域是只能在函数的主体( FunctionBody )内部

**Name属性**

返回函数实例的名称

**Length属性**

指明函数定义的形参个数

Arguments对象

JavaScript函数在定义时有固定数目的命名参数，但当调用这个函数时，传递给他的参数数目是任意的。

形参(Parameters) 是函数定义时的形式参数，作为函数定义的一部分，是列出类的变量。

实参(Arguments) 是函数调用时的实际参数，是在函数被调用时传递给该函数的变量值。

**函数调用时，传递的参数大于函数定义时形参个数，则取前定义位数。**

**Arguments对象**

代表传入函数的实参

是函数中的局部变量

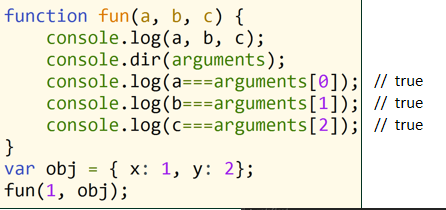
不能显式创建，只有函数调用时才可用

它是一个类数组对象

**类数组对象**

与数组一样具有 length 与 index 属性

**本质确是个 Object**



**Arguments双向绑定**

在调用时arguments对象与实际传递的参数变量发生双向绑定；

Arguments对象中的对应单元和命名参数建立关联

Arguments的length属性：

表示函数调用时传入**实参**的数量；

在调用时，实参个数确定，arguments.length确定，不会再发生改变。

**Call/apply/bind方法**

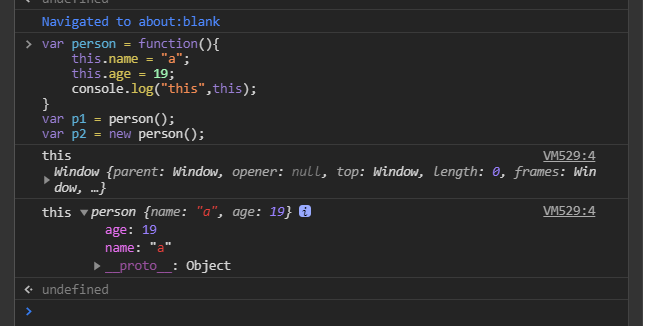
函数是对象，对象是一系列**属性和方法**的集合。

ToString方法：返回一个表示当前函数源代码的字符串

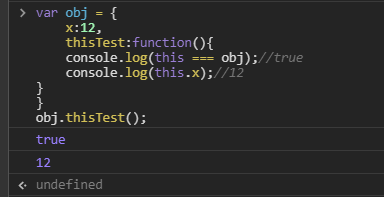
valueOf方法：返回函数本身

**This关键字**

在function内部被创建；指向调用时所在函数所绑定的对象；this不能被赋值，this的值取决于函数被调用的方式。



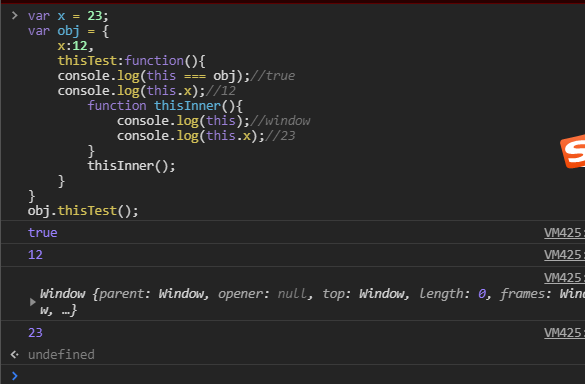
要有new



因为是obj点这个测试，所以函数中的内容都是obj下的。

***This指向问题***

1. 普通函数调用，指向window
2. 构造函数调用，指向实例对象
3. 对象方法调用，指向该方法所属对象
4. 定时器函数，指向window(setInterval)



**call方法**

语法： fn.call(thisObj，arg1，arg2，...)**//参数序列**

参数： arg1,arg2,...：被调用函数的实参

thisObj：将函数对象中的 this 指向 thisObj 对象

说明： 1. 如果 thisObj 未传递，this 指向全局对象 window

2. 如果传递为 undefined/null，this 指向 window 3. 如果传递为数字，字符串，布尔值，this 指向该原始值的包装对象

返回值：与 fn 普通调用相同

作用：调用函数，并改变函数执行的 this 指向

**Apply方法**

语法：fn.apply(thisObj，[arg1，arg2，...])**//参数数组**

参数：arg1,arg2,...：被调用函数的实参

thisObj：将函数对象中的 this 指向 thisObj 对象

说明：1. 如果 thisObj 未传递，this 指向全局对象 window

2. 如果传递为 undefined/null，this 指向 window 3. 如果传递为数字，字符串，布尔值，this 指向该原始值的包装对象

返回值：与 fn 普通调用相同

作用：调用函数，并改变函数执行的 this 指向

**bind（）方法**

语法：fn.bind(thisObj，arg1，arg2,...) **//参数序列**

参数：当绑定函数调用时，thisObj 参数作为原函数运行时的 this 指向。

arg1,arg2,... 当绑定函数被调用时，这些参数加上绑定函数本身的参数会按照顺序作为原函数运行时的参数。（预设参数）

返回值：返回一个原函数的拷贝（绑定函数），并拥有指定的 this 值和初始参数

**Bind不会调用函数，即不会执行原函数中的代码**

**apply，call，bind总结**

**apply，call，bind 三个方法第一个参数都是改变函数在调用时 this 指向的对象**

**apply，call，bind 第一个参数为空，null，undefined，this 指向的是 window**

**apply，call 两个方法只是参数形式有所不同，apply 参数是一个数组，call 参数则是列表序列**

**apply，call 都会立即调用函数执行，bind 不会立即调用函数**

**函数的调用：**

**作为函数直接调用**

定时器函数

立即执行函数

**作为对象方法调用**

事件函数

**作为构造函数调用**

**通过 call/apply 间接调用**

**第六章**

**函数定义和调用过程**

JavaScript 是一种具有函数优先的轻量级，解释型或即时编译型的编程语言

编译型语言写的程序执行之前，需要一个专门的编译过程，把所有程序编译成为机器语言的文件

解释型语言在运行程序的时候才解析，每执行一段代码就要翻译一段代码

两种方式编译的时机不同

**JavaScript 代码的执行过程**

JavaScript 引擎是一段一段地运行代码的

JavaScript 代码执行时，会为当前代码创建相应的运行环境

**JavaScript 运行环境**

全局环境：代码运行起来后会进入全局环境

函数环境：当函数被调用执行时，会进入当前函数中执行代码

**执行上下文(execution context)EC**

当前**代码的运行环境**

作用是用来保护当前代码运行所需的数据

在全局环境、函数环境中会创建执行上下文

**执行上下文栈(Execution context stack,ECS)**

执行上下文栈按照函数的调用顺序来管理执行上下文

栈底永远是全局上下文，栈顶是当前正在执行的函数

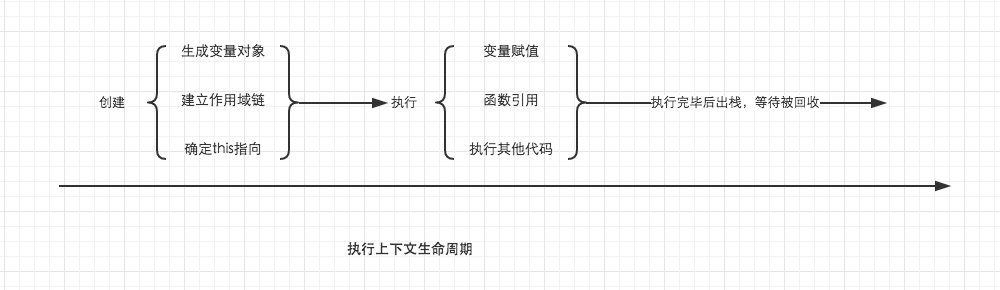
特点：先进后出

生命周期：

当一个函数调用时，一个新的执行上下文就会被创建。

执行上下文的生命周期：

创建阶段；执行阶段；等待回收阶段。



Return语句：销毁当前执行上下文，弹出执行上下文。

变量对象（Variable Object）

变量对象是与执行上下文相关的数据作用域(scope of data) 。

变量对象是与上下文关联的特殊对象，用于存储被定义在上下文中的变量(variables) 和函数声明(function declarations)

变量对象创建过程

建立 arguments 对象

检查当前上下文的函数声明

检查当前上下文中的变量声明

**JavaScript代码运行机制**

JavaScript 代码的执行分为两个阶段

代码解析阶段：将代码翻译成可执行代码

代码执行阶段：执行可执行代码

JavaScript 编译和执行过程

全局解析阶段（全局预解析）

全局顺序执行阶段（变量赋值、函数调用等操作）

当遇到函数调用时，进行函数范围内的预解析，再执行函数内代码，

当存在函数嵌套时，以此类推，会进行多次函数解析与执行

**解析和执行是一个不断交替的过程**

预解析工作之一——声明提升

所有的变量声明和函数声明提升到当前作用域的最前面

声明提升规则

规则1：函数声明整体提前

规则2：变量声明提前，赋值留在原地

规则3：函数会优先被提升，然后才是变量

规则4：函数声明有冲突，会覆盖；变量声明有冲突，会忽略

全局上下文中的变量对象

变量对象就是window对象；在页面关闭前一直存在。

**作用域**

是一套关于如何存储变量当中的值，并且能在之后对这个值进行访问和修改的规则。

作用域的作用

作用域指定变量与函数的可访问范围

作用域控制着变量与函数的可见性

**作用域类型：**

全局作用域：在全局作用域下声明的变量叫做全局变量；在全局下都可以使用；全局作用域中无法访问局部变量；

全局变量的创建方式：

在全局作用域下 var 声明的变量

在函数内部，没有使用 var关键字声明直接赋值的变量

使用 window 全局对象创建的属性和方法

函数作用域：在函数作用域下声明的变量叫做局部变量；局部变量只在当前函数内部中使用；**局部作用域中可以访问到全局作用域中的变量**

局部变量的创建方式

在函数内部通过 var 声明的变量； 函数的形参

块作用域：任何一对花括号 { } 中的语句集都属于一个块，在这之中定义的所有变量在代码块外都是不可见的，我们称之为块级作用域；ES5 没有块作用域，在 ES6 中添加了块作用域

**作用域共有两种主要的工作模型**

词法作用域（静态性）是由函数定义的**书写位置**决定的，与**调用**位置**无关；通过 new Function 创建的函数对象不遵从静态词法作用域；通过 new Function 创建的函数对象总是在全局作用域中执行**

动态作用域（动态性）由**调用位置决定**，不关心变量和函数的定义的书写位置

**Scope属性**

虚拟属性，无法访问和修改；函数**创建（定义）时生成**的属性，保存着这个函数所有父级执行上下文环境中的变量对象的集合

遇到函数（定义时）就有scope，嵌套函数，内层函数的scope还不存在，随需要去分配（解析与执行交替进行）

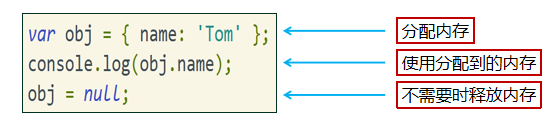
**作用域链（Scope Chain）**

由当前执行环境与所有父级执行环境的一系列变量对象组成；提供对变量和函数的权限和顺序的规则。

**变量函数访问规则**

延作用域链从里向外查找；查找会在找到第一个匹配的表示符时停止。

**内存空间管理**



**垃圾回收机制**

当一个值失去引用之后就会回收

一般的，当一个函数的执行上下文运行完毕之后，内部的所有内容就会失去引用，被垃圾回收机制回收

但是，一个函数的执行上下文运行完毕之后，内部的内容仍然被引用着，就不会被回收

**第七章：闭包和this**

立即执行函数表达式(Immediately-Invoked Function Expression)

**IIFE的写法：**

（1）小括号写法

(function foo( x,y){ ... }(2,3)); //2,3为传递的参数  
 (function foo(x,y){ ... })(2,3);

1. 与运算符结合的写法

var i = function( ){ return 10; }( ); //i为10  
通过IIFE对作用域的改变（限制变量作用域）  
- JS（ES5）中没有块作用域，js文件内和文件间的同名变量容易互相污染  
- 我们往往会通过IIFE引入一个新的作用域来限制变量的作用域

**闭包：**

函数与对其词法环境的引用共同构成闭包。

闭包是由函数以及创建该函数的词法环境组合而成。这个环境包闭包的作用

可通过闭包访问函数作用域内的局部变量

使函数中的变量被保存在内存中不被释放

闭包的缺点

由于闭包会使得函数中的变量都被保存在内存中，内存消耗大

闭包会在父函数外部，改变父函数内部变量的值

**This：**

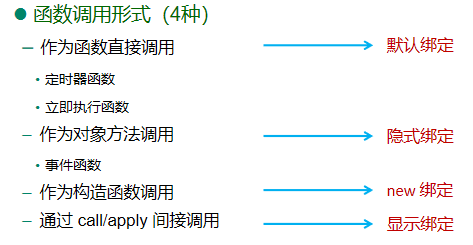
this 是被自动定义在所有函数的作用域中一个关键字

this 是在函数运行期间绑定，与函数定义无关

**this 作用**

复用代码，为函数自动引用合适的上下文对象

提供了一个更加优雅而简便的方式来隐式传递一个对象引用



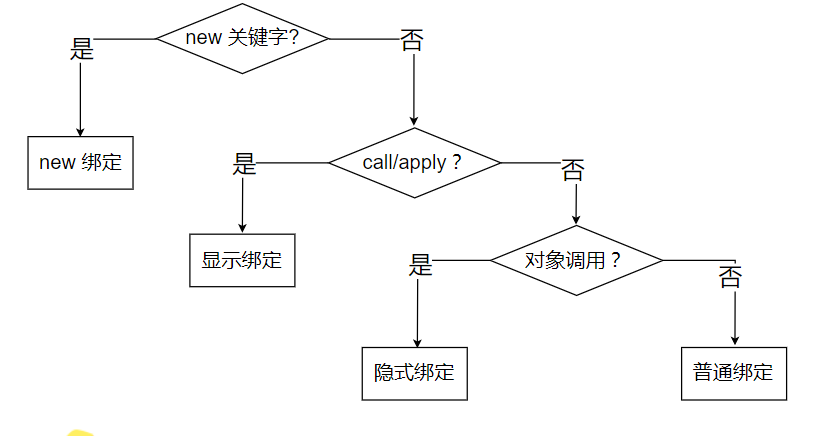
函数嵌套时，this不进行作用域传递；

隐式绑定调用位置是否有上下文对象：函数作为对象的方法调用时，函数上下文指向这个对象；数组中存放函数，被数组索引调用，this上下文指向这个数组

**绑定优先级**

**new 绑定 > 显式绑定 > 隐式绑定 > 默认绑定**

判断 this 的顺序



**严格模式：**

严格模式是采用具有限制性 JavaScript 变体的一种方式

严格模式可以应用到整个脚本或个别函数中

开启严格模式

整个脚本文件开启严格模式，需要在所有语句之前放一个特定语句 "use strict";

某个函数开启严格模式，得把 "use strict"; 声明放在函数体所有语句之前

**函数形式：**

纯函数；

***高阶函数；对其他函数进行操作的函数***

函数的参数为函数

函数的返回值为函数

在封装一个函数时，对于一个不确定的变量，我们使用传递参数的方式来指定。

在封装一个函数时，对于一个不确定的过程，我们向函数中传入另一个函数来指定

**回调函数：**

一个函数被作为参数传递给另一个函数（在这里我们把另一个函数叫做“otherFunction”），回调函数在 otherFunction 中被调用。

回调函数并不会马上被执行。它会在包含它的函数内的某个特定时间点被“回调”（比如定时器）

function foo(x){

return function(){

x.count = x.count? x.count+1:1;

}

}

var age = new Number(2);

var bar = foo(age);

bar(10);

bar(10);

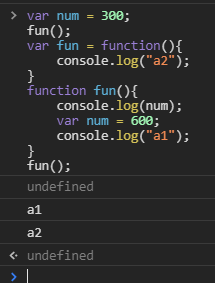
console.log(age.count);

x是undefined，所以undefined=undefined是unedfined

所以执行结果x.count = 1

1 = 1，所以x.count+1 = 2

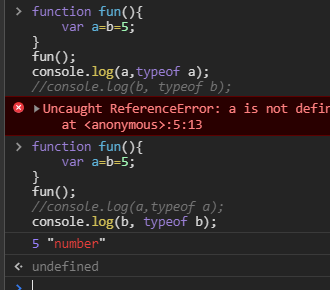
所以输出2

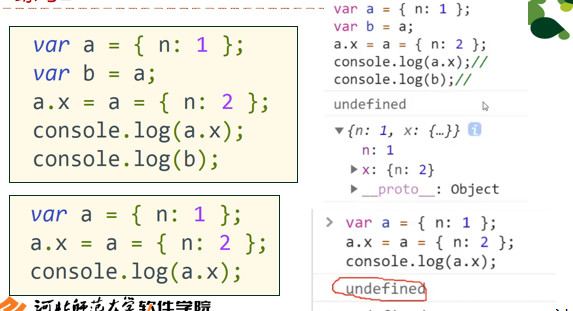


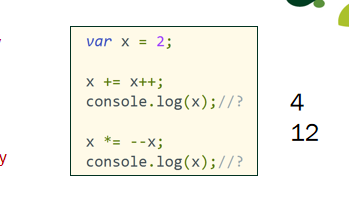
第九章

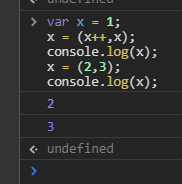
对象是一种复合值：将很多值复合在一起

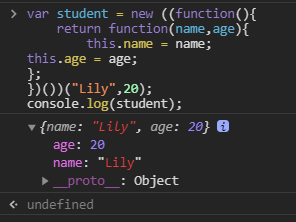
是若干无序属性的集合，可以直接





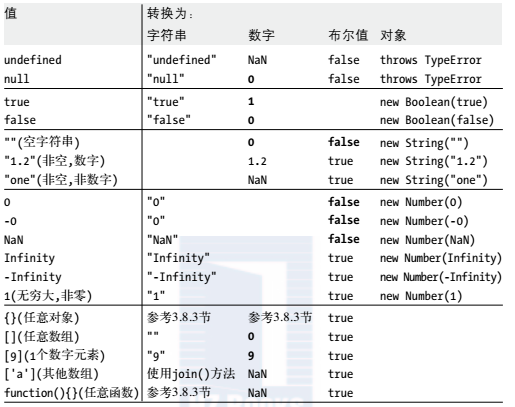


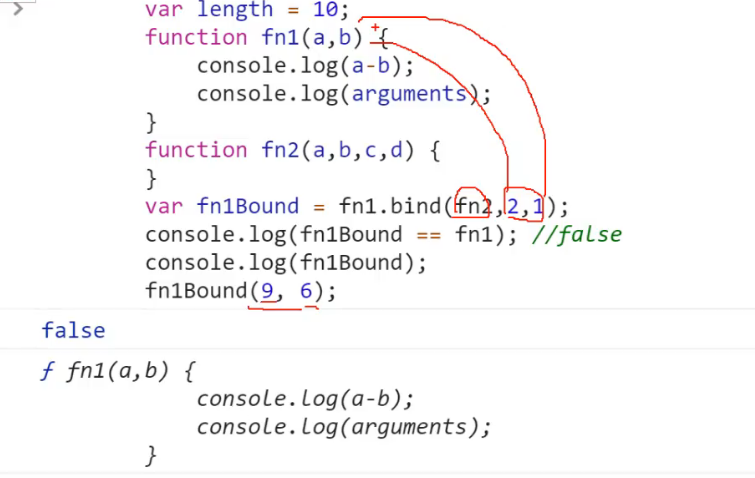




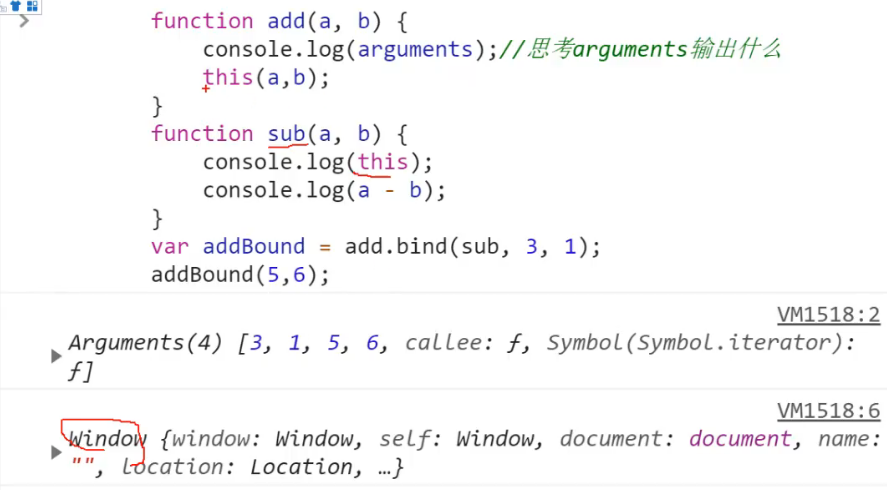
var a =123;  
function foo1(x){  
x = 345;  
}  
foo1(a);  
console.log(a);//123  
  
//  
var a ={y:123};  
function foo2(x){  
x.y = 345;  
}  
foo2(a);  
  
console.log(a.y);//345  
  
//  
var a ={y:123};  
function foo3(x){  
x.y = 345;  
x = {y:456};  
}  
foo3(a);  
console.log(a.y);//345  
  
//  
var a ={y:123};  
function foo4(x){  
x = {y:456};   
x.y = 345;   
}  
foo4(a);  
console.log(a.y);//345

传引用的才会变值，对象.属性才会改变属性值

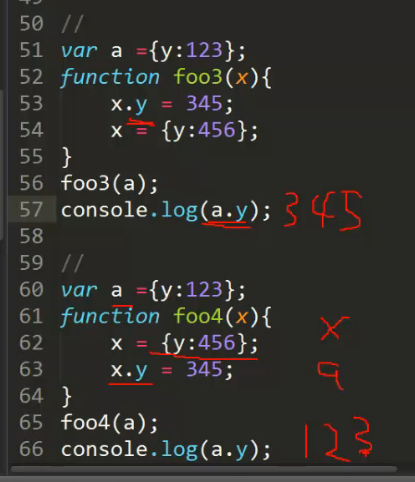




2196

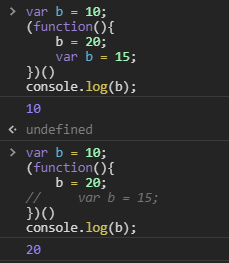






console.log(type of JSON)





预解析

复习：

Window和window：

Window==window；//false

Window.prototype==Window.\_\_proto\_\_;//true

