# 数据类型

## 1 六种数据类型

在JavaScript中一共有Number，String,Boolean,null,undefined五种原始类型和对象

函数，数组，日期等都是属于对象

## 2隐式转换

### 2.1 +/-

如果想把一个变量转换为字符串型，只需要num+””，如果要转换为数字，只需要

Num – 0

### 2.2 ==

“1.23” == 1.23

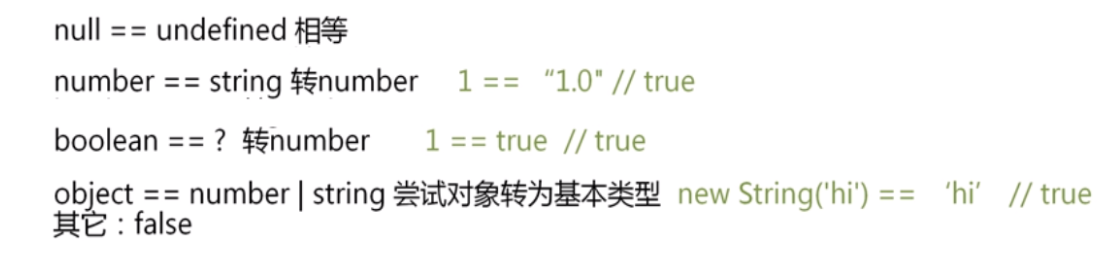
0 == false

使用==时，类型不同时，会尝试转换类型再比较

对象的比较是使用引用去比较，不是值去做比较，两个空对象不相等

New Object() != new Object()

[1,2]!=[1,2]



### 3.3 ===

如果类型不同，就直接返回false

类型相同：

Null === null

undefined === undefined

NaN!= NaN

## 3 包装对象

varstr = “string”;

str.length //6

str.t = 5;str.t //undefined

出现这种的情况的原因是因为，当对一个数据类型使用方法和属性的时候，JS会创建一个临时的对象，如上面将会创建varstrobj = new string(str);调用完后，马上删除，所以导致产生了上面的情况，能获取变量，设置属性

## 4 类型检测

### 4.1 typeof

返回一个表示类型的字符串

typeof 100 “number”

typeof true “boolean”

typeof function “function”

typeof undefined “undefined”

typeof new Object() “object”

typeof [1,2] “object”

typeofNaN “number”

typeof null “object”

### 4.2 instanceof

objinstanceof Object 判断obj的原型链是否有Object的构造函数，即obj是不是由Object构造而来，适合自定义类型

### 4.3 Object.prototype.toString

适合内置对象和基元类型，遇到null和undefined失效

### 4.4 constructor

### 4.5 duck type

## 5 编程练习

在index.html文件中编写arrysSimilar函数，实现判断传入的两个数是否相似

1. 数组中的成员类型相同，顺序可以不同，如[1,true]和[false,2]相似
2. 数组的长度一致
3. 类型的判断范围，需要区分：String,Boolean,Number,undefined,null,function,

日期，window

当以上全部满足，返回：“判定结果：通过”，否则”判断结果：不通过”

# 表达式和运算符

## 1表达式

表达式是指能计算出值的任何可用程序单元

### 原始表达式

常量，直接量：3.14,”test”

关键字：null ,this,true

变量：i,j,k

原始表达式可以通过运算符构成复合表达式：10\*20

### 数组，对象的初始化表达式

[1,2] new Array(1,2)

[1,,4] [1,undefined,undefined,4]

{x:1,y:2} var o = new Object();

o.x = 1;o.y=2;

### 函数表达式

varfe = function(){};

(function() {console.log(“hello world”);})();

### 属性访问表达式

var o = {x:1};

o.x

o[‘x’]

### 调用表达式

func();

### 对象创建表达式

newFunc(1,2);

new Object;

## 运算符

JS中的常规运算符如位运算符((|,&,^)，算术运算符(+,-,\*,/)，逻辑运算符(||,&&,!)，三目运算符(?:)都与C中相同，现就特殊运算符进行说明

### delete

deleteobj.x;

obj.x; //1

deleteobj.x;

obj.x; //undefined

varobj = {};

Object.defineProperty(obj,’x’,{

configurable:false,

value:1

})

deleteobj.x;

obj.x;

//这里设置了一个标签，当删除obj.x时会返回false

注：Object.defineProperty(obj,prop,descriptor)

参数：obj需要定义属性的对象

prop需定义或修改的属性的名字

descriptor 将被定义或修改的属性的描述符

返回值 返回传入函数的对象，即第一个参数

有以下可以设置的键值

configurable：当且仅当该属性的值为true时，该属性描述符才能被更改，也能够被删除，默认为false

enumerable：当且仅当该属性为true时，该属性才能出现在对象的枚举属性中，默认为false

### in

可以查看对象中是否有当前属性

window.x = 1

‘x’ in window;

### new

function Foo() {}

Foo.prototype.x = 1;

varobj = new Foo();

obj.x; //1

obj.hasOwnProperty(‘x’);

obj.\_proto\_.hasOwnProperty(‘x’);

### void

void 0 undefined

void (0) undefined

### 2.5 运算符优先级



# 语句、严格模式

## 1 块语句

JavaScript程序由语句组成.

一般可以用{}组合多个语句(块语句)，块没有作用域

{var x = 1;}

console.log(x); //1

function foo(){

var a = 1;

console.log(a); //1

}

foo();

console.log(typeof a);

## 2 try catch

try{

//执行代码

}catch(err)

{  
 //当try语句抛出了异常，才会执行这里处理错误

}finally{

//不论是否抛出异常，一定会执行的任务

}

## 3 for...in

for...in语句用于对数组或对象的属性进行循环操作

如：

var as = ["1",2,3,5,6,7];

for(var t in as)

{

console.log(t);

}

//输出：0 1 2 3 4 5

# 对象

对象中包含一系列属性，这些属性是无序的，每个属性都有一个字符串key和对应的value

var obj = {x:1,y:2};

## 1 对象key值

var obj = {};

obj[1]=1;

obj['1']=2;

obj// Object{1:2}

obj[{}] = true;

obj[{x : 1}] = true;

obj; // Object {1: 2, [object Object]: true}

对象的key必须是Js中正确定义的类型，数值，字符，对象，boolean等类型，而且数字和字符不加区别，相同的字符和数字会互相覆盖

## 2 原型对象

### 2.1 prototype和\_proto\_

prototype是函数才有的属性，\_proto\_是每个对象都有的属性

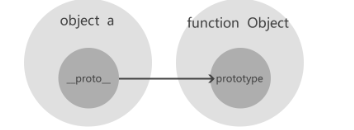
\_proto\_可以理解为"构造器的原型"

即 \_proto\_ === constructor.prototype

### 2.2 proto属性的指向

2.2.1 字面量方式

var a = {};

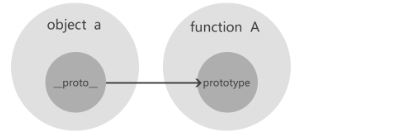


它指向的是function Object(){}中的的prototype

2.2.2 构造器方式

var A = function(){};

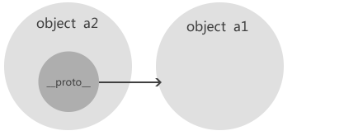
var a = new A();



2.2.3 Object.create方式

var a1 = {}

var a2 = Object.create(a1);

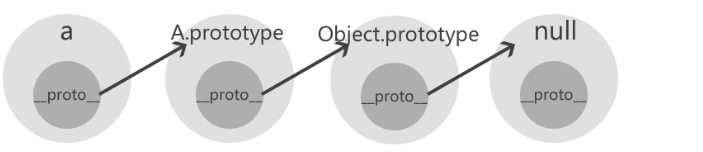


### 2.3 原型链

\_proto\_是任何对象都有的属性，所以可以形成一个原型链

var A = function(){};

var a = new A();



## 3 对象属性的特性

### 3.1 数据属性

数据属性包含一个数据值的位置，在这个位置可以读取和写入值

4个描述的行为特性

writeable 表示能否修改属性的值，默认为true

Enumerable 表示能否通过for in进行枚举

configurable 表示是否能将delete删除属性重新定义属性

value 包含这个属性的数据值，读取属性的时候，从这个位置读取

写入属性值的时候，把新值保存在这个位置。

例：

function Foo(){};

Foo.prototype.age = 22;

var obj = new Foo();

obj.name = "A";

Object.defineProperty(obj,"telephone",{value:12345678910,

writable:true,//可以修改

enumerable:true,//可以枚举

configurable:true//可以删除

});

如果将true修改为false将不能修改

对象上的原型也可以用这个方法修改定义

Object.defineProperty(Foo.prototype,"addr",{

value:"abc",

writable:true,

enumerable:true,

configurable:true,

});

其中configurable是一个顶层配置项，如果将其配置为false，其它两个特性就不能被重新定义

### 3.2 数据属性的操作

都是通过 . 或 []来访问设置元素，作为左值时，如果不存在对应的键值加添加，如果已经存在就修改，作为右值时就获取，还可以用delete来解除对象和属性的绑定

var a = {};

a.ext = true;//新建

a[ext] = false;//修改

var b =a[ext];//获取

delete a.ext;//删除

### 3.3 对象的特性

proto原型 即当前对象继承自哪一个对象，当前对象继承了原型中的哪些方法和 属性

class类 用来表示对象的类型

extensible可拓展

### 3.4 对象的结构

function Foo(){}; //声明函数

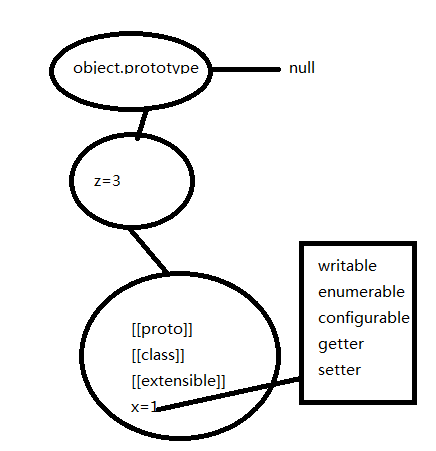
console.log(typeof Foo.prototype); //Foo.prototype是一个对象

Foo.prototype.z = 3;//设置了原型属性z为3

var obj = new Foo();//用构造函数来创建了obj对象

obj.x = 1;//x是obj对象上本身的属性

console.log(obj.toString());//toString是顶级object对象上的属性，先查看原型上有没有这个属性，没有再往原型链上顶级的object对象上查找



## 4 创建对象

### 4.1 字面量创建

var obj1 = {x:1,y:2};

var obj2 = {x:1,y:2,o:{z:3,n:4}};

### 4.2 new/原型链

function foo(){}

foo.prototype.z = 3;

var obj =new foo();

obj.y = 2;

obj.x = 1;

obj.x; // 1

obj.y; // 2

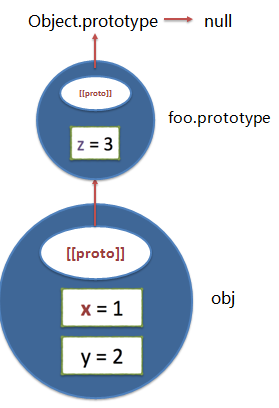
obj.z; // 3

typeof obj.toString; // ‘function'

'z' in obj; // true

obj.hasOwnProperty('z'); // false

内部结构：



### 4.3 对象创建 Object.create

var obj = Object.create({x:1});//obj就继承对象 {x:1};

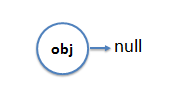
obj.x ;//1

typeof obj.toString;//"function"

obj.hasOwnProperty('x');//false;

var obj = Object.create(null);

obj.toString;//undefined



### 4.4 getter,setter()方法

var man={

name:'A',

weibo:'B',

get age(){

return new Date().getFullYear()-1988;

},

set age( val){

console.log('Age can\'t be set to'+val);

}

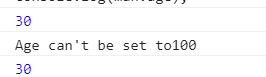
}

console.log(man.age);

man.age=100;

console.log(man.age);

/\*输出\*/



即使用getter,setter时，会自动调用get和set函数，函数名和属性名保持一致

## 5 对象标签

### 5.1 [[proto]]

### 5.2 [[class]]

### 5.3 [[extensible]]

## 6 序列化

## 7对象方法

# 数组

数组是值的有序集合，每个值叫做元素，每个元素在数组中有数字位置编号。JS中的数组可以包含不同类型的元素。数组元素可以是其它的对象或数组

## 1 创建数组

### 1.1 字面量创建数组

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

var student = [{name:'a',age:27},{name:'af',age:3}];

### 1.2 new Array

var arr = new Array();

var arrWithLength = new Array(100);

var arrLikesLi = new Array(true,false,null,1,2,"hi");

## 2 数组元素读写

var arr = [1,2,3,4,5]

arr[1];

arr.length;

arr[8]=6;

arr.length;//9

arr;// [1, 2, 3, 4, 5, 6, empty × 2, 6]

即数组可以直接访问任意大于0的位置，没有的就为空。

## 3 数组元素的增删

var arr = [];

arr[0] = 1;

arr[1] = 2;

arr.push(3);

arr; // [1, 2, 3]

arr[arr.length] = 4; // equal to arr.push(4);

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

arr.unshift(0);

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

delete arr[2];//将arr[2]置为未定义，但位置仍然保留

arr; // [0, 1, undefined, 3, 4]

arr.length; // 5

2 in arr; // false

arr.length -= 1;

arr; // [0, 1, undefined, 3, 4], 4 is removed

arr.pop(); // 3 returned by pop ，会移除数组末尾

arr; // [0, 1, undefined], 3 is removed

arr.shift(); // 0 returned by shift 从数组头部移除

arr; // [1, undefined]

## 4 数组迭代

可以使用for in和for循环进行迭代，和对象的处理方法类似

var i = 0, n = 10;

var arr = [1, 2, 3, 4, 5];

for (; i < n; i++) {

console.log(arr[i]); // 1, 2, 3, 4, 5

}

for(i in arr) {

console.log(arr[i]); // 1, 2, 3, 4, 5

}

Array.prototype.x = 'inherited';

for(i in arr) {

console.log(arr[i]); // 1, 2, 3, 4, 5, inherited

}

for(i in arr) {

if (arr.hasOwnProperty(i)) {

console.log(arr[i]); // 1, 2, 3, 4, 5

}

}

Array.prototype.x = 'inherited';

for(i in arr) {

console.log(arr[i]); // 1, 2, 3, 4, 5, inherited

}

for(i in arr) {

if (arr.hasOwnProperty(i)) {

console.log(arr[i]); // 1, 2, 3, 4, 5

}

}

## 5 二维数组，稀疏数组

### 5.1二维数组

和C中类似，

var arr = [[0, 1], [2, 3], [4, 5]];

var i = 0, j = 0;

var row;

for (; i < arr.length; i++) {

row = arr[i];

console.log('row ' + i);

for (j = 0; j < row.length; j++) {

console.log(row[j]);

}

}

### 5.2 稀疏数组

稀疏数组不含有从0开始的连续索引，一般length属性值比实际元素个数大

var arr1 = [undefined];

var arr2 = new Array(1);

0 in arr1; // true

0 in arr2; // false

arr1.length = 100;

arr1[99] = 123;

99 in arr1; // true

98 in arr1; // false

数组本质上只是一个对象，其索引不是真正的数字，而是一个键值。所以可以产生稀疏数组

## 6 数组方法

都是从Array.prototype中继承过来

### 6.1 将数组转为字符串

var arr = [1, 2, 3];

arr.join(); // "1,2,3"

arr.join("\_"); // "1\_2\_3"

function repeatString(str, n) {

return new Array(n + 1).join(str);

}

repeatString("a", 3); // "aaa"

repeatString("Hi", 5); // "HiHiHiHiHi"

### 6.2 将数组逆序

var arr = [1, 2, 3];

arr.reverse(); // [3, 2, 1]

arr; // [3, 2, 1]

### 6.3 排序

var arr = [13, 24, 51, 3];

arr.sort(); // [13, 24, 3, 51]

arr; // [13, 24, 3, 51]

arr.sort(function(a,b){return a-b;});//这时需要一个函数来确定元素顺序

arr = [{age : 25}, {age : 39}, {age : 99}];

arr.sort(function(a, b) {

return a.age - b.age;

});

arr.forEach(function(item) {

console.log('age', item.age);

});

### 6.4 数组合并

var arr = [1, 2, 3];

arr.concat(4, 5); // [1, 2, 3, 4, 5]

arr; // [1, 2, 3]

### 6.5 返回部分数组

var arr = [1, 2, 3, 4, 5];

arr.slice(1, 3); // [2, 3]

arr.slice(1); // [2, 3, 4, 5]

arr.slice(1, -1); // [2, 3, 4]

arr.slice(-4, -3); // [2]

//原数组没有被修改

### 6.6 数组拼接

var arr = [1, 2, 3, 4, 5];

arr.splice(2); // returns [3, 4, 5]

arr; // [1, 2]; //原数组被修改

### 6.7 数组遍历

var arr = [1, 2, 3, 4, 5];

arr.forEach(function(x, index, a){

console.log(x + '|' + index + '|' + (a === arr));

});

// 1|0|true

// 2|1|true

// 3|2|true

// 4|3|true

// 5|4|true

### 6.8 数组映射

var arr = [1, 2, 3];

arr.map(function(x) {

return x + 10;

}); // [11, 12, 13]

arr; // [1, 2, 3]

//映射就是将原数组映射为新数组，和遍历的操作类似，对每一个元素都执行相同逻辑的操作

### 6.9 数组过滤

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

arr.filter(function(x, index) {

return index % 3 === 0 || x >= 8;

}); // returns [1, 4, 7, 8, 9, 10]

arr; // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filter会返回一个新的符号逻辑条件的数组

### 6.10 数组判断

var arr = [1, 2, 3, 4, 5];

arr.every(function(x) {

return x < 10;

}); // true

arr.every(function(x) {

return x < 3;

}); // false

会为每个元素都执行一次every

var arr = [1, 2, 3, 4, 5];

arr.some(function(x) {

return x === 3;

}); // true

arr.some(function(x) {

return x === 100;

}); // false

会为每个元素都执行一次every,但找到符合条件的就返回不再执行

### 6.11 数组检索

var arr = [1, 2, 3, 2, 1];

arr.indexOf(2); // 1

arr.indexOf(99); // -1

arr.indexOf(1, 1); // 4

arr.indexOf(1, -3); // 4

arr.indexOf(2, -1); // -1

arr.lastIndexOf(2); // 3

arr.lastIndexOf(2, -2); // 3

arr.lastIndexOf(2, -3); // 1

//返回给定元素的具体位置，第一个参数为值，第二个为查找的起始位置

# 函数

函数是一块JS代码，被定义一次，但可执行和调用多次，JS中的函数也是对象。

## 1 函数声明

### 1.1函数声明及函数表达式

function add (a, b) {

a = +a;

b = +b;

if (isNaN(a) || isNaN(b)) {

return;

}

return a + b;

}

var add = function(a,b){

};

函数表达式

(function(){})();

return function(){};

var add = function foo(a,b){};

### 1.2 函数前置声明

func1(); // 输出：我是函数声明

func2(); // 报错

console.log(a); // 输出：undefined

function func1() {

console.log("我是函数声明");

}

var func2 = function() {

console.log("我是函数表达式");

}

var a = 10;

函数声明，变量声明都会被前置，但函数表达式只会将表达式的变量提前声明，但是未赋值，所以提前后为undefined.

### 1.3 函数构造器

var func = new Function('a','b','console.log(a+b);');

fun(1,2); //3

Function('var localVal = "local"; console.log(localVal);')(); console.log(typeof localVal);

//result:local,undefined

localVal仍为局部变量

## 2 this

### 2.1 一般函数的this

function f1(){ return this; }

f1() === window; // true

function f2(){

"use strict";//严格模式

return this; }

f2() === undefined; // true

### 2.2 对象方法的函数的this

var o = {

prop:37,

f:function(){

return this.prop;

}

}

console.log(o.f());//37

### 2.3 对象原型链上的this

var o = {f:function(){ return this.a + this.b; }};

var p = Object.create(o);

p.a = 1;

p.b = 4;

console.log(p.f()); // 5

### call/apply方法与this

function add(c, d) {

return this.a + this.b + c + d;

}

var o = { a: 1, b: 3 };

add.call(o, 5, 7);//1+3+5+7

add.apply(o, [10, 20]);//1+3+10+20

function bar() {

document.log(Object.prototype.toString.call(this));

}

bar.call(7);//”[object Number]”

call和apply都是在特定的作用域中调用函数，等于设置函数体内this对象的值，以扩充函数赖以运行的作用域

apply接受函数运行的作用域和参数数组。

call 第一个个也是函数运行的作用域，第二个是参数类表，必须一一列举

### bind方法与this

function f(){

return this.a;

}

var g = f.bind({a:”test”});

console.log(g());//test

var o = {a:37,f:f,g:g};

console.log(o.f(),o.g());//37,test

bind创建一个新的函数，当被调用时，将this关键字设置为提供的值，在调研新函数时，在任何提供之前提供一个给定的参数序列。

bind(this，….)第一个参数指定this，即作用域，后面指定参数列表中的参数

## 3 函数属性和参数

function foo(x,y,z){

arguments.length;

arguments[0];

arguments[0] = 10;//arguments[0]和x绑定，可以互相改变

x;

arguments[2] = 100;//未传入这个参数，不能和z绑定

z;

arguments.callee==foo;

}

foo.name 函数名

foo.length 形参个数

arguments.length 实参个数

arguments.callee是指向拥有这个arguments对象的函数

在严格模式下，arguments[0]只是x的一个副本，即不能修改x.

# 函数和作用域

## 1 闭包

闭包是指一个函数或函数的引用与一个引用环境绑定在一起，这个引用环境是一个存储该函数每个非局部变量的表

闭包不同于一般的函数，它允许一个函数在立即词法作用域外调用时，仍然可以访问非本地变量

即，闭包就相当于是能够读取其他局部变量的函数。

闭包可以读取函数内部的变量，另外一个就是让这些变量的值始终保持在内存中

如：

function f1(){

var n = 999;

nAdd = function(){n += 1;}

function f2(){

console.log(n);

}

return f2;

}

一般情况下当return之后，函数内部的变量将被释放，不能访问，但Js中会形成闭包，这个闭包包含了创建时能访问的所有局部变量

var Counter = (function() {

var privateCounter = 0;

function changeBy(val) {

privateCounter += val;

}

return {

increment: function() {

changeBy(1);

},

decrement: function() {

changeBy(-1);

},

value: function() {

return privateCounter;

}

}

})();

这里相当于创建了一个词法环境，为三个函数所共享，Counter.increment

Counter.decrement,Counter.value

## 2. 作用域

var a = 10;

(function() {

var b = 20;//函数局部变量

})();

console.log(a);//10

console.log(b);//会出错，不能在函数外部访问函数内部变量

for(var item in {a:1,b:2}){

console.log(item);

}

console.log(item);

eval("var a = 1;");//eval可以动态执行代码，它的作用域和它查找或新增的作用域完全一样

## 3 作用域链

因为内部函数可以访问外部函数变量这种机制，所以可以用链式查找决定哪些数据能被内部函数访问。