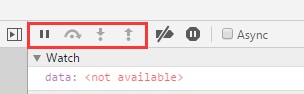
**（1）调试**

使用Chrome，按F12。

进入Sources，右方位置红框处



功能依次为：

《1》继续执行按钮；

《2》单步执行按钮；

《3》进入函数按钮；

《4》退出函数按钮；

**添加监视的变量：**



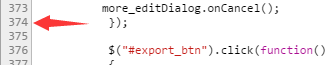
《1》在watch处点击加号，添加要监视到变量；

**通过console面板调试：**



在Source面板的下方，有console面板，于是可以通过console面板来调试代码。

**加断点：**



点击需要加断点的代码左边的行号即可。

**查找js文件：**

Ctrl + O键，输入文件名然后可以查找（似乎支持模糊查找）

**查找某个函数：**

Ctrl + Shift + O键，然后输入函数名查找（注意，只在js文件里有效，对html文件是无效的）

断点后就可以单步执行、添加监视的变量、或者进行其他操作了。

**Console面板的其他用处：**

可以在里面修改变量的值（例如把一个变量值是1的改为100）

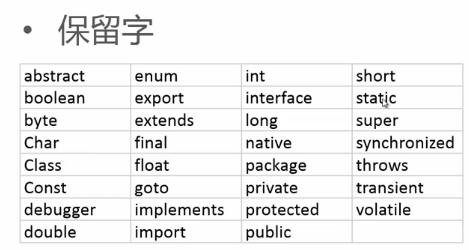
**（2）变量命名**

①除了字母、下划线之外，还可以以 **$** 开头

②不可以用某些关键字作为变量名



③不可以以保留字作为变量名



④对大小写是敏感的。

**（3）类型**

①number

普通的；

八进制：以0开头，如var num = 070，换算成十进制就是56

十六进制：以0x开头，如var num = 0xa 换算成十进制就是10

例如：

$(document).ready(function(){

$("#a").text(**0xa**);

});

会在<div id="a"></div>的位置显示10，而若是017则显示15

浮点数：1.1e5表示1.1x10的5次方=110000

**NAN：**不是一个数字（not a number）

**infinity：**无限、无穷大（如1/0）；

②字符串；

③bool；

④Ojbect：一组无序的名值对的集合

例如：

var cat = {

name:"kitty",

age:2,

speak: function(){

alert("喵");

}

}

⑤null 赋值了，对象不存在，空的对象，可以赋值为null

⑥undefined 已声明的，但没有赋值的变量；对象中不存在的属性。

⑦类型识别：typeof 变量名，可以返回变量类型

⑧引用类型：

对象、数组、函数是引用类型。

简单来说，假如给一个对象（num1）赋值（num1={a:1}），将一个对象（num1）赋值给另一个变量（num2），那么通过那个变量（num2）修改对象的某个值（num2.a=10），那么此时第一个对象的值（num1.a）也将变为num2.a的值（即num1.a=10）；

如代码；

$(document).ready(function(){

var num1 = {a:1};

var num2 = num1;

num2.a = 10;

if(num1.a == 10)

{

$("#a").text(num1.a);

}

});

显示结果是10（num1.a的值）

**（4）操作符**

①=== 全等

!== 不全等

②! 取非操作符

!0 true

![] false

!"" true

!![] true

③&& 和，左右两个都要成立，其值为：

第一个是true，值为第二个的值；——由第二个决定

第一个为false的话，值为第一个的值；——被第一个决定

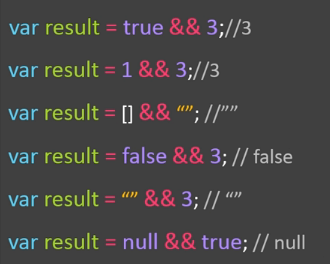
假如第一个为false的话，第二个不会执行，

如：num = 0;

var a = false && num++;

alert(num);

值为0



④|| 或，成立一个即可，其值为：

左边为true，值是左边；——被第一个决定

左边为false，其值是右边——被第二个决定

假如左边是true，那么右边的不会被执行

⑤new 获得对象的实例

⑥delete 删除对象的属性

例如

$(document).ready(function(){

var num1 = {a:1};

var num2 = num1;

num2.a = 10;

delete num2.a;

$("#a").text(typeof num1.a);

});

值是undefined

⑦.和[]类似，获取对象的属性

例如

var dog = {age:1};

dog.age;

dog['age'];

后两行的返回值都是1。

⑧instanceof 判断变量是否是某个类型的实例

例如：

var num1 = {a:1};

if(num1 instanceof Object)

{

$("#a").text(num1.a);

}

会输出1。

但注意，num1.a并不是Number的实例（即值为false）

⑨in 判断某个属性是否在对象中

var num1 = {a:1};

if('a' in num1)

{

$("#a").text(num1.a);

}

输出1。

注意，a需要加引号（单引号双引号都可以）

⑩位操作符

~

**&** 将前后两个数字转为二进制，假如在某一位上，两个二进制数都是1，那么返回true，否则返回false。例如7=111; 4=100; 7和4在第三位上都是1，于是返回true，而8=1000; 没有任何一位和7在同位置上都是1，于是返回false

**<<** 左移一位，可以理解为将值x2（正负都一样），忽视小数部分。具体来说是，将当前数字每一位都左移一位。

>> 应该是右移一位了。值除二，有小数部分则省略。

|

^

>>>

据说用的比较少

**（5）语句**

①if else、switch、while、do while语句、用法都同C++的

如：

a = 8;

switch(a)

{

case 8:$("#a").text(1);

break;

default:$("#a").text(2);

}

②for (属性名 in 对象) { 语句 }

例如：

for (var i in obj)

console.log(i);

将输出obj的所有属性名

③with (表达式) {语句}

例如：

$(document).ready(function(){

obj = {

name:"john",

age:10

}

with(obj){

$("#a").text(name + "　age　" + age);

}

});

其name属性指的就是obj对象的。

但注意，以下这样写是不行的

~~obj.name="john";~~

~~obj.age=10;~~

但若在这两行之前加上：

obj = new Object; 那么是可以的

④异常捕获语句

**try{**

**语句**

**} catch(exception){**

**语句**

**} finally{**

**语句**

**}**

其中try是要执行的语句，假如出错，执行catch里的语句，最后无论如何（错误与否），会执行finally里的语句。

**（6）数值**

注意，Math的M需要大写

①绝对值：Math.abc(x) 获得x的绝对值（之后的意思都类似，非特殊不再说明）

②求四舍五入后结果：

Math.round(1.9) = 2;

Math.round(1.1)=1;

③向上取整：

Math.ceil(1.1)=2;

Math.ceil(-1.1)=-1;

④向下取整：

Math.floor(1.9)=1;

Math.floor(-1.9)=-2;

⑤求最大值：

Math.max(1,2,3,4) = 4;

⑥求最小值

Math.min(-1,-2,-3,-4) = -4;

⑦随机数：

0 <= Math.random() < 1

⑧



第二个是e的多少次方，第三个是log，第四个是对x进行开方，第五个是x是y次方。

e指的是自然对数（以常数e为底数的对数叫做自然对数记作ln N(N>0).）

第一个是指弧度，弧度=2\*π/360\*角度；

⑨对字符串进行处理变为数字：

**变为整数：**

parseInt(string, radix); //第二个参数表示输出几进制的数字可略

从第一个数字来开始识别，识别到不能识别为止。

例如：（注意，实际运用中，应该加引号，如 parseInt("1a11.1")这样）

1 1

1.1 1

1.1a 1

1a11.1 1

a1.1 NaN

**变为浮点数：**

同理上面

parseFloat("1.2a11.1") 1.2

**转为数值：**

Number(" 字符串 ")

值：

1.1 1.1

1.1a NaN

**保留指定位小数：**

num.toFixed(digits); 输出的是 **字符串**

例如：

(100.111).toFixed(2); 输出是100.11

(100.1).toFixed(2) 输出的是100.10

(0).toFixed(2) 输出的是0.00

**（7）字符串**

假设字符串为str，例如str="aaa";

**①str.length** 返回字符串的长度（字符数）汉字是1个字符

**②得到输入框（input）的长度：var str = input.value;**

得到输入框：document.getElementById("#input"); ——这里假设输入框ID为input

**③str.charAt(index)** 得到字符串索引值（index）位置的字符

示例："abc".charAt(0) 的值 "a";

最后一个字符是str.charAt(str.length-1);

**④str.indexOf( 字符串, 从哪个index开始找)** //第二个参数可以省略

用处返回字符串第一个字符所在的位置（如果没有，则为 -1 ——注意，-1不是false）；

可以用于判断字符串是否在str里。

**⑤str.search(regexp)** 参数是正则表达式

返回字符串里是否有符合要求的，如果有，返回位置，如果没有，返回-1

例如：str.search(/[0-9]/) 查找字符串里是否有0~9的数字

var str="abc1";

$("#a").text(str.search(/[0-9]/));

输出为3

var str="abc";

$("#a").text(str.search(/[0-9]/));

输出为-1

**⑥str.match(regexp)** 参数是正则表达式

返回匹配到的正则表达式的值，返回值是一个数组

例如：

var str="abc1332d";

$("#a").text(str.match(/[0-9]/));

返回值是1

例如：（加/g返回全部的）

var str="abc1332d";

$("#a").text(str.match(/[0-9]/g));

返回值是：1,3,3,2

如果找不到，返回值是null

**⑦str.replace(正则表达式|被替换字符串, 替换的字符串|函数)**

例如：

var str="abc21";

$("#a").text(str.replace(/[0-9]/g, 'q'));

返回值是：abcqq（两个数字都被替换为了q）

若不加g，那么则返回的是abcq1（第二个数字没有被替换）

注意，参数一被替换为参数二

**⑧str.substring(索引1, 索引2)**  返回索引1~索引2之间的字符串

前包含后不包含，例如：

var str="abc21";

$("#a").text(str.substring(1,2));

输出的是b

假如索引2被省略，那么一直输出到字符串的结尾

例如：

var str="abc21";

$("#a").text(str.substring(1));

输出：bc21

**⑨str.slice(开始索引, 结束索引)** 和上面功能一样，额外功能是，结束索引可以是负数。

假如是负数，那么从后往前数第n个，同样不包含结尾的。例如：

var str="abc21";

$("#a").text(str.slice(1,-1));

输出bc2（从下标1输出到下标倒数第1个但不包含倒数第一个）

var str="abc21";

$("#a").text(str.slice(-1));

输出1（因为从最后一个开始，到结尾，因此只有一个）

**⑩str.substr(开始索引, 长度)** 从开始输出，输出指定长度的

var str="abc21";

$("#a").text(str.substr(1,2));

输出c（从第二个c开始取1个）

**《11》str.split( 分隔符, 输出几个)**  以分隔符为分割，输出数组

例如：

var str="a1b1c1d";

$("#a").text(str.split("1"));

输出：数组a,b,c,d

分隔符也可以是**正则表达式**：

var str="a1b1c1d";

$("#a").text(str.split(/[0-9]/));

输出不变

**《12》str.toLowerCase()**  将字符串变为小写

var str="ABC";

$("#a").text(str.toLowerCase());

返回值是abc

**《13》str.toUpperCase()**  将字符串变大写

var str="aaaAs";

$("#a").text(str.toUpperCase());

返回值是：AAAAS

**《14》String(被转对象)**  转为字符串

例如：

var str = 111;

str = String(str);

$("#a").text(typeof str);

返回值是：string

又例如：

var str = true;

str = String(str);

$("#a").text(str);

返回值是：

true

这里是true是字符串true，而不是bool类型

又例如：

var str = function(){alert("1")};

str = String(str);

$("#a").text(str);

输出：

function (){alert("1")}

**《15》转义字符**

例如：

var str = "a\"a";

$("#a").text(str);

输出：a"a

转义后 转义前

\" "

\\ \

\t tab按键

其他略

**（8）对象**

①创建对象：

**var obj = new Ojbect{};**

或者是

**var obj = {};**

②增加对象的方法：

**ojb.name = "John";**

就是给加一个name属性，其值为John，同理可以加函数。

③删除对象方法：

**delete obj.name;**

就是删除obj的name属性

④查看某个对象的类型：

**obj.constructor;**

例如：

var str ={};

str.name="John";

str.age=10;

if(str.constructor == Object)

$("#a").text("111");

是能输出111的

**⑤obj.toString()** 将对象字符串（但并不是所有对象都可以）

例如：

var str ={};

str.name="John";

str.age=10;

var p = new Number(123);

$("#a").text(p);

如果是p就可以输出123；如果是str就会输出Object

**⑥obj.valueOf()**  获取某个对象的原始值

function abc(){this.a=1};

给其加一个属性abc.prototype.b=2;

此时输入abc.valueOf();

显示：function abc(){this.a=1};

**⑦obj.hasOwnProperty( 参数 )** 返回参数是否存在于对象的属性中。

可以用于验证属性是否从继承而来。如果是继承，返回依然是false。

例如：

var str ={};

str.name="John";

str.age=10;

$("#a").text(str.hasOwnProperty("name"));

输出true

**（9）数组**

**①创建数组：**

var array = new Array();

或者是：

**var array = [];**

例：var array = [1, 3, 5];

数组里可以是数字、字符串、对象、数组、bool值

以下假设数组为arr

**②arr.length 数组的长度（空为0）**

**③arr[x] 数组下标为x的成员**

**④修改数组元素，可以通过③中的直接修改。**

**⑤arr.indexOf(元素, 从哪里索引开始)**

第二个参数一般不用。返回值是索引值（和之前的indexOf用法一样）；

**⑥arr.forEach(回调函数, 参数2)**  遍历数组，回调函数遍历到每个成员都会执行。参数2可省略（貌似是什么this替代的什么的，不懂）

**注意：IE8不支持**

回调函数的参数有三个：第一个是遍历到的元素，第二个是该元素的索引，第三个是这个数组本身。

例如：

$(document).ready(function(){

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

var result = new String();

arr.forEach(function(item, index, array){

item += 1;

item = item.toString();

result += item +"的索引是" + index + "　";

})

$("#a").text(result);

});

输出是：

2的索引是0　4的索引是1　6的索引是2　8的索引是3

**⑦arr.reverse()**  将数组的元素顺序颠倒一次

上代码即可：

$(document).ready(function(){

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

var result = new String();

arr.reverse();

arr.forEach(function(item, index, array){

item = item.toString();

result += item +"的索引是" + index + "　";

})

$("#a").text(result);

});

输出：

7的索引是0　5的索引是1　3的索引是2　1的索引是3

**⑧arr.sort( 回调函数 )**  排序

不传回调函数的话，默认升序。

对字符串，按照Unicode码从第一个字符开始逐字符比较，然后排列。

例子：

$(document).ready(function(){

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

**arr.sort(function(a,b){**

**return b - a; //b-a为降序，a-b为升序**

**});**

$("#a").text(arr);

});

输出8,7,6,5,4,3,2,1

**⑨arr.push( 元素1, 元素2, .....)**  给已有数组加元素，加到最后位置。

**arr.unshift( 元素1, 元素2, ...)** 给已有数组添加，加到最前面

**⑩arr.shift()** 获取数组第一个元素，并且从数组第一个元素中移除他。

如代码：

$(document).ready(function(){

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

alert(arr.shift()); //输出1

$("#a").text(arr); //从3开始输出

});

**arr.pop()** 类似上面，但从最后一个拿

**《11》arr.splice( 从哪个索引开始, 删除多少, 插入的新元素);**  可以用于添加、删除、替换等

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

arr.splice(1,0,4); //位置1开始删除0个再添加4

$("#a").text(arr);

输出：1,4,3,5,7

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

arr.splice(1,1); //位置1删除1个于是剩下1,5,7

**《12》arr.slice(开始索引, 结束索引)**  返回值是数组的拷贝，开始位置包含结束不包含，注意这里没有**p**

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

var b = arr.slice(1,2); //从3开始5结束（不包含5）

$("#a").text(b);

输出3

第二个参数没有就表示到结束。

arr.slice(0); 效果是拷贝所有

**《13》arr.concat( 值1, 值2, .....)**  可以把几个数组连在一起，arr数组是第一个，值1是第二个。（注意，参数不一定是数组，也可以是其他类型）

例如：

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

var arr1 = [2,4,6];

var str = "aa";

var b = arr.concat(arr1, str);

$("#a").text(b);

输出：1,3,5,7,2,4,6,aa

**《14》arr.join( 分隔符 );**

将数组以分隔符链接起来，输出字符串（前面有将字符串拆成数组）；

如：

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

arr = arr.join(" + "); //注意，这步的返回值才是我们想要的

$("#a").text(arr);

输出：

1 + 3 + 5 + 7

如果没有参数，则默认是逗号，如果参数是""，那么无分隔符

**《15》arr.map( 回调函数 )**  遍历，并返回一个数组作为返回值

例如：

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

var arr1 = arr.map(function(item, index, array){

return item+1;

});

$("#a").text(arr1);

输出：2,4,6,8

**《16》arr.reduce( 回调函数, 传入的初始值begin )** 具体要看回调函数

function( 上一次函数的返回值last, item, index, array){

retunr last + item;

}

last是上一次调用回调函数时的返回值，而第一次调用回调函数时，其值是begin。

例如：

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

var arr1 = arr.reduce(function(last, item, index, array){

console.log(last);

return last + item;

}, 100);

$("#a").text(arr1);

输出值是116。原理是100 + 1 + 3 + 5 + 7

console结果是100, 101， 104，109

**注意：潜在的问题，没有执行第一次回调函数时的情况**

假如没有给第二个参数begin，那么console.log只有三个结果（即1，4，9），说明第一次对于1，并没有执行回调函数。

如代码：

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

var arr1 = arr.reduce(function(last, item, index, array){

item += 1;

return last + item;

});

$("#a").text(arr1);

预想输出是20（2+4+6+8），但实际输出是19（1+4+6+8）；

另外需要注意，以上这次函数时，并没有对arr本身造成影响。

**（10）函数**

①**函数的形参少于／多于实参：**

var fun = function(a,b){return a+b}

fun(1)返回NaN——形参多

fun(2,3,4)返回5（因为2+3）——实参多

②**所有的参数，被放置在arguments这个变量里，以数组形式保存。**

例如：

var arr = function(){

var total=0;

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

{

total += **arguments[i]**;

}

return total;

};

console.log(arr(1,2,3));

输出结果是6

③**函数的参数，**

原始类型——按值传递（函数内不影响函数外）；

对象Object——按引用传递（函数内被修改，函数外也会被改）；

如代码：

dog = {age:1}; //对象

var arr = function(obj){ //函数

obj.age++;

return obj;

};

**cat = arr(dog); //dog传递给函数，返回值给cat**

console.log(dog.age); //dog被修改

**cat.age++; //cat是dog对象的引用，修改属性**

console.log(dog.age); //dog再次被修改

说明：对象作为实参，是按引用传递，返回值若是对象，也是引用。可以理解为C++的引用变量，只要是引用变量，修改的便是同一个初始的变量（cat，dog还有函数里的obj，指向同一个对象dog）；

④**构造函数：**

**function 函数名( 形参 )**

**{**

**this.形参1 = 某个值;**

**this.函数内自定义变量名 = 某个值;**

**this.函数名 = function( 参数 )**

**{**

**函数内容**

**}**

**//这里可以理解为，隐含了一个return this**

**}**

这就是一个**构造函数**。

例如代码：

function fun( num ){ this.value = num;}

var obj1 = new fun(1);

console.log(obj1.value);

输出结果是1

⑤**构造函数的原型：**

是加了prototype在对象名和属性名之间，例如：

obj.**prototype**.add = function(){ this.value ++;}

示例代码：

function fun( num ){

this.value = num;

}

fun.prototype.num=10;

fun.prototype.plus = function(){

this.value += this.num;

}

var obj1 = new fun(1);

var obj2 = new fun(2);

console.log(obj2.num); //输出10（说明和obj1互不干扰）

obj1.plus();

console.log(obj1.value); //输出11

console.log(obj1.num); //输出10

obj1.num++; //10-》11

obj1.plus(); //11+11=22

console.log(obj1.value); //输出22

console.log(obj1.num); //输出11

**（11）Date日期**

**①当前时间：new Date();**

**②指定年月日：new Date(年, 月-1, 日, 时, 分, 秒);**  ——参数是数字，

注意：月是从0开始的（0是1月）

如代码：

console.log(new Date(2016,4,4,1,2,3))

输出Wed **May** 04 2016 01:02:03 GMT+0800 (中国标准时间)

**③获取某个具体的：（注意，date是在之前获取到的时间）**

获取年 date.getFullYear();

获取月 date.getMonth();

获取日 date.getDate();

获取时 date.getHours();

获取分 date.getMinutes();

获取秒 date.getSeconds();

**④时间格式化：**

如代码：

function dateFormat(date){

return date.getFullYear() + "-" + padding(date.getMonth() + 1) + "-" + padding(date.getDate()) + " " + padding(date.getHours()) + ":" + padding(date.getMinutes()) + ":" + padding(date.getSeconds());

}

function padding(num){

return num<10 ? "0" + num : num;

}

var date=new Date();

console.log(dateFormat(date));

输出：

2016-05-10 20:50:58

**⑤设置date时间：**

date.setFullYear(2010) 设置年份为2010

其他同理

date.setDate(40) 会在当前月份基础上，查看第40天是哪天（下个月的哪天），即会自动递增。

**⑥求天数**

newDate(年份, 10, 0)由于10代表11月，11月没有第0天，因此会显示10月份的最后一天。

同理，假如10输入-1，年份输入2001年，那么会显示2000年12月。

也就是说，月份、日期，时分秒，都可以 **输入负数**。

其真正时间，是上一级单位减去对应的数字后的时间。

⑦Date——》Number

对一个Date对象使用getTime()方法，可以获得其距离1970-1-1 00:00:00的毫秒数。

Number——》Date

使用newDate(毫秒数)可以转为日期

也可以使用setTime(毫秒数)来设置日期

**（12）正则表达式**

①描述字符串规则的表达式

使用  **/规则/**  来表示

②正则表达式.test(字符串)

用于测试正则表达式与字符串是否匹配

返回值是true或者false

注：只需要字符串里包含正则表达式，即返回true，也就是说，假如正则表达式是/123/，字符串是a1234b，那么依然返回true

**③锚点：**

^ 起始位置（shift+6），/^http:/匹配以https开头的

$ 结束为止，/\.jpg$/ 表示以.jpg结尾

\b 单词边界，\bTom\b表示Tom需要是一个独立的单词

例：

/^123/.test('ab123') 返回false

/^123/.test('123ab') 返回true

/\.jpg$/.test("123.jpg") 返回true

/\bTom\b/.test("name is Tom Peel") 返回true

/\bTom\b/.test("nameisTomPeel") 返回false

/^12$/.test("12") 返回true（表示12开头和结尾，1212是false）

/^12$/.test("a12") 返回false

**④字符类：（表示一个字符）**

[abc] 表示a或b或c

[a-z] 表示a~z中的一个

[0-9] 表示0~9中的一个

[^0-9] 表示不是0-9（不是数字）

. 表示任意字符（除换行符）（是一个英文句号）

例子：

/./.test("1233")

true（因为有通配符）

/[0-9]/.test("1233")

true（因为有数字）

/[^0-9]/.test("1233")

false（因为没有非数字的）

/[^0-9]/.test("1233a")

true（因为有了非数字a）

/[^0-9][^0-9]/.test("1233a")

false（因为只有一个非数字字符）

/^[0-9]/.test("a1233a")

false（因为不是以0-9的某个数字为开头的）

**⑤元字符**

\d 表示[0-9]

\D 表示[^\d]，即非0-9

\s 空白符

\S [^\s]非空白符

\w [A-Za-z0-9\_]字母、数字、下划线

例子：

/\d/.test("2");

true（因为有数字）

/\D/.test("2");

false（因为没有非数字）

/\w/.test("2");

true（因为有字母、数字、下划线的一个）

/\s/.test("2");

false（因为没有空白符）

/\s/.test("2 2");

true（因为有空白符）

/^\s/.test("2 2")

false（虽然有空白符，但不在开头）

/\S/.test("2 2");

true（因为有非空白符2）

/\S/.test(" ");

false（因为没有非空白符空格）

**⑥量词**

{m,n} 出现m~n次

\* {0,}表示0~无穷次

? {0,1}出现0次或者1次

+ {1,}表示出现至少1次

例子：

/\d+/.test("1");

true（数字至少出现了一次）

/\d+/.test("a");

false（数字一次都没出现）

/\d{2,}/.test("1");

false（数字至少出现2次但只有1次）

/\d+:/.test("1");

false（虽然有数字，但是数字后面没有跟冒号字符

/\d+:/.test("1:");

true（这个1后面有冒号所以是true）

/1\d{1}/.test("1a1");

false（因为第二个字符应该也是一个数字，并且出现一次）

/1\d{2}/.test("12a1");

false（第2个字符开始，2个字符需要是数字，不符合）

**⑦正则表达式的两个模式：**

简述：

贪婪模式：正则表达式在匹配成功的前提下，尽可能多的去匹配，匹配次数少，效率高；

惰性模式：正则表达式在匹配成功的前提下，尽可能少的去匹配，匹配次数多，效率低。

详述：

1.从语法角度看贪婪与非贪婪

被匹配优先量词修饰的子表达式，使用的是贪婪模式；被忽略优先量词修饰的子表达式，使用的是非贪婪模式。

匹配优先量词包括：“{m,n}”、“{m,}”、“?”、“\*”和“+”。

忽略优先量词包括：“{m,n}?”、“{m,}?”、“??”、“\*?”和“+?”。

2.从应用角度看贪婪与非贪婪

贪婪与非贪婪模式影响的是被量词修饰的子表达式的匹配行为，贪婪模式在整个表达式匹配成功的前提下，尽可能多的匹配；而非贪婪模式在整个表达式匹配成功的前提下，尽可能少的匹配。非贪婪模式只被部分NFA引擎所支持。

3.从匹配原理角度看贪婪与非贪婪

能达到同样匹配结果的贪婪与非贪婪模式，通常是贪婪模式的匹配效率较高。

所有的非贪婪模式，都可以通过修改量词修饰的子表达式，转换为贪婪模式。

贪婪模式可以与固化分组结合，提升匹配效率，而非贪婪模式却不可以。

引用：http://blog.csdn.net/lxcnn/article/details/4756030

**⑧转义符：**

|  |  |
| --- | --- |
| 元字符 | 转义后的字符 |
| / | \/ |
| **.** | **\.** |
| **?** | **\?** |

例子：

/^https?:\/\/[a-z]+\.[a-z]+\.[a-z]+/.test("https://www.baidu.com")

true（表示http开头，然后可能有s，下来是冒号外加2个/，之后至少有一个字母（小写的），下来有一个英文句号，之后至少有一个字母（小写的），下来有一个英文句号，然后以至少一个英文字母为结尾）

**⑨多选分支**

| 表示或

例子：

/^(a|b)/.test("a1");

true（表示a或者b开头的，等价于[ab]）

/^(a|b)/.test("c1");

false（因为是c开头）

/\.(jpg|bmp|png|jpeg|gif)$/.test(".jpg1");

false（因为结尾多了一个1，可以用来匹配图片的文件名后缀）

**⑩正则表达式的捕获：**

() 保存匹配到的字符串，之后可以用

(?:匹配条件) 不保存匹配到的字符串

var p = str.match(regexp) 获取匹配到的字符串，返回值是p

捕获到的变量存储在$1，$2之类的变量中

例子：

(https?:)\/\/ 表示http，可能有s，后面是一个冒号，再后面跟着//

([^/\]+) 表示不是\的字符，有至少1个

(\/[^/?]\*)? 表示先有一个/，然后非?字符有0~多个（并且这一段可能有可能没有）

(\?[^#]\*)? 表示?开头，然后非#字符有0~多个（并且这一段也可能有可能没有）

(#.\*)? 表示#开头，任意字符有0~多个。并且这段可能有可能没有

var str = "http://www.baidu.com".match(/(https?:)\/\/([w]+)\.([a-z]+)\.([a-z]+)/)

str 这里是输出str（即match的返回值，注意，str是一个字符串数组）

["http://www.baidu.com", "http:", "www", "baidu", "com"]

**⑪替换一个子串**

str.replace(regexp/substr, replacement) 表示字符串用正则表达式匹配后，替换为参数2。注意，参数2可以是**函数，函数的参数是要被替换的内容**

例子：

"abc".replace(/(c)/,"$1.00");

"abc.00"（c被匹配到了，$1表示捕获到的c，替换为c.00）

"price is 5".replace(/(\d)/g,"$$$1.00");

"price is $5.00"（使用两个$可以表示一个$）

**⑫表示全部**

在正则表达式后面加g

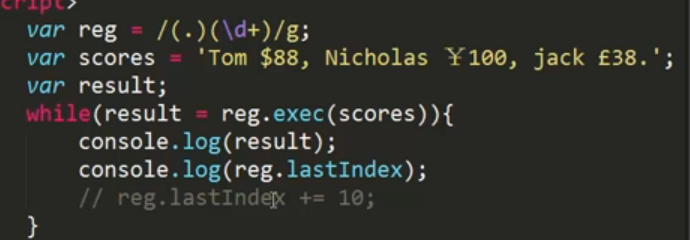
/a/g

例子：

abca".replace(/(a)/g,"$1.00")

"a.00bca.00"（所有a都被替换为a.00）

⑬var 字符串 = reg.**exec**(字符串)

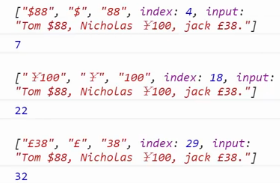
result是exec方法的返回值，其结果包含三部分：

第一部分是一个数组，是匹配到的内容（如$88、￥100等）；

第二部分是索引（第一次的索引是$88出现的位置4，第二次是￥100出现的索引18）

第三部分是整个输入内容，即scores这个字符串。

而reg.lastIndex是上一次匹配到的索引，如图：



分别是4，18和29（即三个字符串出现的位置）

⑭正则表达式的结尾：

/i (忽略大小写)  
/g (全文查找出现的所有匹配字符)  
/m (多行查找)  
/gi(全文查找、忽略大小写)  
/ig(全文查找、忽略大小写)

**（13）JSON**

①JSON是JavaScript Ojbect Notaion

②json中，某一项最后一个属性的最后，不能有逗号，最后一项的花括号外面也不能有逗号

例如：  
[{"name":"Petter","age":10}]这样正确的

[{"name":"Petter","age":10**,**}**,**]这样是错误的，特别是IE是无法识别出来的

③var obj = JSON.parse(json对象, 回调函数)

返回值obj是一个js的对象。

回调函数可以省略，其形式为：

function(key, value){return val;}

即传递两个参数，第一个是key值，第二个是value值（会遍历整个对象，对每一项执行），返回值是当前属性的value值。

注：IE6、IE7不支持

解决办法：

if(!window.JSON){ //如果不支持JSON方法

window.JSON = {

parse: function(sJSON){

return eval('(' + sJSON + ')');

},

  stringify: function(obj){

            var result = "";

            for(var key in obj){

                if(typeof obj[key] == "string"){

                    // 如果属性值是String类型，属性值需要加上双引号

                    result += "\"" + key + "\":\"" + obj[key] + "\",";

                }else if(obj[key] instanceof RegExp){

                    // 如果属性是正则表达式，属性值只保留一对空大括号{}

                    result += "\"" + key + "\":{},";

                }else if(typeof obj[key] == "undefined" || obj[key] instanceof Function){

                    // 如果属性值是undefined, 该属性被忽略。忽略方法。

                }else if(obj[key] instanceof Array){

                    // 如果属性值是数组

                    result += "\"" + key + "\":[";

                    var arr = obj[key];

                    for(var item in arr){

                        if(typeof arr[item] == "string"){

                            // 如果数组项是String类型，需要加上双引号

                            result += "\"" + arr[item] + "\",";

                        }else if(arr[item] instanceof RegExp){

                            // 如果属数组项是正则表达式，只保留一对空大括号{}

                            result += "{},";

                        }else if(typeof arr[item] == "undefined" || arr[item] instanceof Function){

                            // 如果数组项是undefined, 则显示null。如果是函数，则显示null?。

                            result += null +",";

                        }else if(arr[item] instanceof Object){

                            //如果数组项是对象(非正则，非函数，非null)，调用本函数处理

                            result += this.stringify(arr[item]) +",";

                        }else{

                            result += arr[item] + ",";

                        }

                    }

                    result = result.slice(0,-1)+"],"

                }else if(obj[key] instanceof Object){

                    // 如果属性值是对象(非null，非函数，非正则)，调用本函数处理

                    result += "\"" + key + "\":" + this.stringify(obj[key]) + ",";

                }else{

                    result += "\"" + key + "\":" + obj[key] + ",";

                }

            }

            // 去除最后一个逗号,两边加{}

            return "{" + result.slice(0,-1) + "}";

        }

}:

}

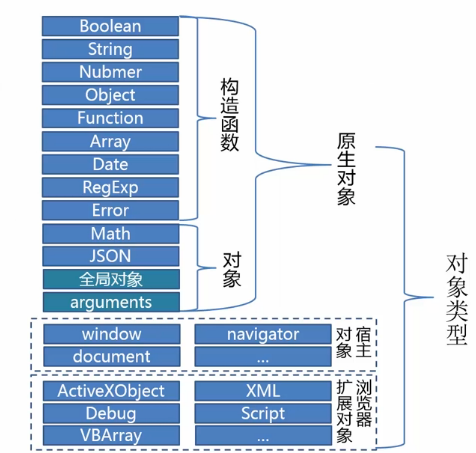
自行定义一个这样的方法

④JSON.stringify(js的对象, 回调函数)

将js的对象，转化为一个JSON对象，返回值是JSON对象

**（14）类型进阶**

**①对象类型（引自网易云课堂）：**



**②原始类型和对象类型的区别：**

原始对象的值存储在栈内存之中；

对象类型的值存储在堆内存，在栈内存之中只存储了堆内存的地址。因此，对象的赋值是引用传递（赋值的是地址）

**③隐式类型转换的情况**

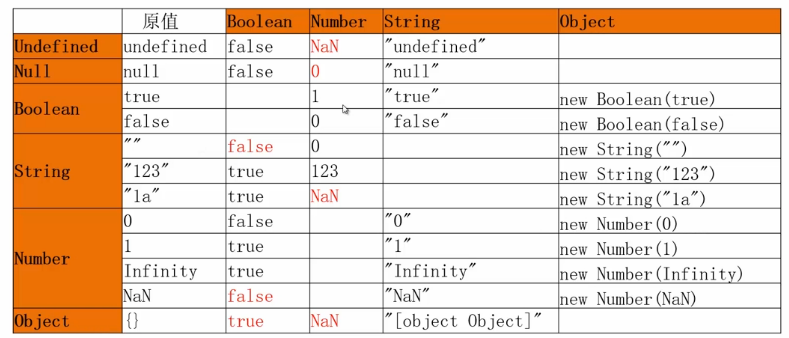
数字运算符+-\*/

英文句号

if语句

== 比较运算符

**④隐式类型转换的结果：**

**⑤显示转换**

Number(参数)

String(参数)

Boolean(参数)

parseInt(参数)

parseFloat(参数)

! 或者!! 取布尔值

**(15)类型识别**

①注：str、num等是一个变量名

typeof(str) == 'string' 是否是字符串，string要小写

typeof(num) == 'number' 是否是数字，number要小写

arr instanceof Array 是否是数组（A要大写）

typeof(obj) == 'ojbect' 是否是对象，object小写

注意：对象数组（[{},{}]）也是对象object。（并且数组也是对象）

②类型识别的方法：

typeof

instanceof

Object. prototype.toString.call

constructor

**③typeof 操作符**

可以写typeof("123")；

也可以写typeof 123

返回都是number

typeof识别为对象object的有：

null；

[]数组；

Date日期；

/[1-2]/正则表达式；

function abc(){}; typeof new abc()构造函数的实例

另：

typeof function(){}将被识别为函数function

**可以识别标准类型(Null除外）**

**不能识别具体对象类型（function()）除外**

**④instanceof**

[] instanceof Array 能识别数组

/[0-9]/ instanceof RegExp 能识别正则表达式

**可以识别内置对象类型（比如数组）**

**但不能识别原始类型（比如number、string等）**

**也可以判定自定义对象类型（识别所有对象类型）；**

假如有函数对象function abc(){}，有另一函数对象function ABC(){}

其中，ABC继承abc，ABC.prototype = new abc();

有一对象p，是ABC的一个实例：var p = new ABC()

那么，p instanceof ABC返回true；

p instanceof abc返回true（因为ABC继承于abc）

但注意：假如在var p = new ABC()声明于ABC.prototype = new abc()之前，那么p instanceof abc将返回false（即使之后ABC继承于abc也一样）。原因在于，声明p的时候，已经创建一个对象了。此时ABC再继承abc将无法改变已经创建的对象。

**⑤Object.prototype.toString.call(参数)**

返回一个字符串，这个字符串里说明了该参数的类型。

例如：

Object.prototype.toString.call(12)

"[object Number]"（可以识别数字）

Object.prototype.toString.call("12")

"[object String]"（可以识别字符串）

Object.prototype.toString.call([])

"[object Array]"（可以识别数组）

Object.prototype.toString.call({})

"[object Object]"（可以识别对象）

Object.prototype.toString.call(/\d/)

"[object RegExp]"（可以识别正则表达式）

typeof Object.prototype.toString.call({})

"string"（返回内容是一个字符串）

还可以识别null、undefined等

进行函数封装：

function type(obj){

return Object.prototype.toString.call(obj).slice(8,-1)

}

将返回一个字符串，是该类型。

例如字符串则返回String等

type("123")

"String"

type(/[0]/)

"RegExp"

由于返回是字符串，因此可以进行字符串的比较。

**该方法可以识别标准对象类型，以及所有内置对象类型。**

**但不能识别自定义对象类型。**

**⑥constructor 显示构造函数**

例如：

(123).constructor的返回值是一个数字的构造函数，但可以全等于Number

("123").constructor的返回值是字符串，可以全等于String

(123).constructor === Number

true

("123").constructor === String

true

**总结：可以用来判断标准类型（如Number，除undefined和null）**

**也可以用来判断内置对象类型。**

**可以判断自定义对象类型**

封装：（旧）

function getConstructorName(obj){

return obj && obj.constructor && obj.constructor.toString().match(/function\s\*([^(]\*)/)[1];

}

obj表示是存在；

obj.constructor表示其构造函数存在；

obj.constructor.toString()表示将其构造函数转为字符串类型（就是那个函数，转为字符串）

然后对转换后的字符串进行正则表达式的匹配，如果匹配成功则返回对应的值。

**正则表达式表示：**function开头，\s表示是一个空格符，\*表示这个空格符出现0~无数次，([^(]\*)/)是我们要获取的字符串。

[^(]表示一个字符，这个字符不是(，后面加了\*表示是取所有在(字符之前的字符，也就是函数名（比如是function Number()中的Number），然后捕获这些（因为有()），

接下来的/表示正则表达式结束了，)表示match这个方法结束了，[1]表示匹配的第二个，原因在于会匹配成功2次，因为获取的第一个是(之前的所有字符，第二个才是我们想要的。具体见例子：

"abc".constructor.toString()

"function String() { [native code] }"

"abc".constructor.toString().match(/function\s\*([^(]\*)/)

["function String", "String"]

"abc".constructor.toString().match(/function\s\*([^(]\*)/)[1]

"String"

由于&&的特点是，如果左边为true，则返回右边的，因此return的最终结果是返回我们想要的类型名，也就是构造函数名。

由于传入 0, false, "", NaN 这些值时，得到错误的返回结果，新封装如下：

function getConstructorName(obj){

    return (obj===undefined||obj===null)?obj:(obj.constructor&&obj.constructor.toString().match(/function\s\*([^(]\*)/)[1]);

}

**（16）函数进阶**

①函数定义的方法：

函数声明：function(a){return a;}

函数表达式：var fun = function(a){return a;}

函数实例化：var fun = new Function("a","return a");

函数实例化的方法不推荐；

②区别：

**函数声明定义函数的特点：**

可以在定义前被调用（表达式和实例化不行）

原因在于：js会在执行代码之前，预解析变量定义（先），函数声明（后）等。

也因此，当重复声明一个函数时（function a(){return a;}和function a(){return a+1;}），执行时，只会执行最后一个函数。

例子：

console.log(a(1));

function a(b){return b;}

function a(b){return b+1;}

console.log(a(1));

输出的都是2

总结：

**函数定义被前置，重复定义时最后一次有效**

**函数表达式：**

后一个定义的函数，不影响前一个定义的函数。

function a(b){return b;}  
console.log("1:"+a(1));  
var a = function(b){return b+100;}  
console.log("2:"+a(1));  
function a(b){return b+1;}  
console.log("3:"+a(1));

输出结果是：

1:2

2:101

3:101

推断原因是：由于函数声明会被预解析，因此第一次执行的是预解析的第三个函数（因此结果是1+1=2）；

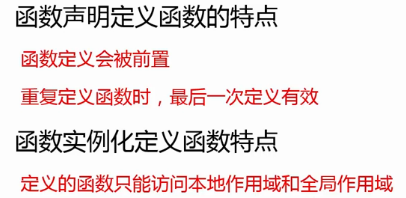
之后，由于函数表达式，因此a实际是第二个函数（参数+100），因此结果是101；当第三次输出时，由于函数预解析，因此执行的实际是函数表达式，因此也是101

**③函数作用域：**

函数表达式、函数声明：其变量作用域，都是逐级向上的。

即先找同级的，没有的话找上一级的，然后逐级向上找。

函数实例化：其变量作用域不遵循逐级向上，如果当前没有，那么直接访问全局变量。



**(17)函数调用**

①四种方法

②第一种：函数调用模式

函数名加括号，比如add(2,3);

③第二种，方法调用模式：

函数作为一个对象的对象方法（函数），使用：对象名.方法名() 这样的办法；

如：var plus = {  
 add:function(m,n){return m+n;}  
}  
plus.add(2,3);

返回5

④第三种：构造函数调用模式

使用new来声明一个构造函数实例，例如：

function plus(m){  
 this.**a**=m;  
**}**var p=new plus(1);

//话说下面这种方式也算构造函数调用么？

**console**.log(new **Function**("a","return a")(5));

⑤第四种：apply(call)调用模式

apply是函数对象（function Function(){}）的prototype方法中的一个方法

可以通过**console.dir(Function)**来打印全部方法

又因为所有函数都继承于Function函数，因此，所有函数都继承了这个方法。

apply的功能是，函数借用。将函数借用给一个对象，帮助他实现函数所定义的逻辑的功能。

如例子：

function ab(x, y) {  
 this.**x** = x;  
 this.**y** = y;  
}  
ab.**prototype**.move = function (x, y) {  
 this.**x** += x;  
 this.**y** += y  
}  
var c = new ab(1, 1);  
**console**.log(c);  
c.move(2, 2)  
**console**.log(c);  
**p** = {**x**: 0, **y**: 0, **z**: 0}  
**console**.log(**p**);  
c.move.apply(**p**, [5, 5]) //在这步，ab的实例c将move方法提供给了p，参数是数组 **console**.log(**p**);

输出结果为：

ab {x: 1, y: 1}

ab {x: 3, y: 3}

Object {x: 0, y: 0, z: 0}

Object {x: 5, y: 5, z: 0}

注：如果c中没有y，那么在执行后，会显示y的值为NaN

**⑥this的指向：**

函数调用模式（上面方法一）：

**this指向的是全局对象**，（包含嵌套函数），所以在函数里假如声明变量时不加var，其变量将为全局变量。

function ab() {  
 var a=10;  
 **console**.log(this);  
 function c(){  
 **console**.log(this);  
 }  
 c();  
 }  
**console**.log(a);

注意：此时a会显示undefined

方法调用模式（上面方法二）：

**this指向的是当前对象，即调用者**

var plus = {  
 add: function (m, n) {  
 **console**.log(this);  
 return m + n;  
 }  
}

特殊情况：在方法调用里套用函数调用，由于函数调用的this是window，而不是调用者，因此若对变量操作，实际操作的是window的变量。如代码：

var plus = {  
 **m**:0,  
 add: function (m) {  
 var p = function(){  
 plus.**m**++; //必须加了plus.m，才能对m进行操作  
 **console**.log(this); //这里的this是window  
 **console**.log(m); //这里的m是undefined  
 }  
 p();  
 }  
}  
plus.add();

构造函数调用模式（上面方法三）：

this指向被构造的对象。

function plus(m) {  
 this.**a** = m;  
 **console**.log(this);  
}  
var p = new plus(1);

输出的是plus {a: 1}

apply(call)调用模式（上面方法四）：

this指向apply的第一个参数（即对其生效者）；

function ab(x, y) {  
 this.**x** = x;  
 this.**y** = y;  
}  
ab.**prototype**.move = function (x, y) {  
 this.**x** += x;  
 this.**y** += y  
 **console**.log(this);  
}  
var c = new ab(1, 1);  
**p** = {**x**: 0, **y**: 0, **z**: 0};  
c.move.apply(**p**, [5, 5]);

输出的是Object {x: 5, y: 5, z: 0}，说明是调用者p

⑦arguments 是一个对象，是函数的实参数组

arguments[i] 获取第i+1个参数

arguments.length 参数数量

⑧函数的传参：

原始类型：按值传递

对象类型：按共享传递，其与按引用传递的区别在于：

《1》直接修改对象内部的值的时候，实参内部的变量的值同样也会改变；（**共同点**）

《2》将另一个对象赋值给形参时，形参获得新对象的地址（而不是旧对象的地址），因此对其进行修改，修改的是新对象的值，而不是旧的。（**不同点**）

例子：（修改内部值，共同点）

var p={**x**:0};  
function ab(obj) {  
 obj.**x** = 1;  
}  
ab(p);  
**console**.log(p);

输出：Object {x: 1}

给形参赋值对象（不同点）：

var p={**x**:0};  
function ab(obj) {  
 obj.**x** = 10;  
 obj={**x**:0};  
 obj.**x** = 1;  
 return obj;  
}  
**console**.log(ab(p));  
**console**.log(p);

两次输出结果为：

Object {x: 1}

Object {x: 10}

发现，第一次确实被修改了，但是第二次修改的是函数内部被赋值的对象（{x:0}）。

**（18）闭包**

①函数内部的子函数，用到了父函数的变量，就叫做闭包。

②可以为函数保存其执行状态，

③其原理大概是：

首先，假如B函数在A函数的内部，则A为外部函数，B为内部函数，B可以访问A的变量（变量的作用域规定的）；

然后，假如在函数内部，如果有一个return，那么在return结束前，这个函数中的变量，其值是维持不变的；

另外需要注意，不能让A函数（外部函数）执行完毕，否则状态无法保留（因为闭包的前提是A函数没有执行完毕），因此不能直接调用A函数，而是需要生成A函数return部分的一个实例。

因为有了这两个前提，于是两个办法：

**方法一：**我们可以把B函数放在A函数的return之中。return的内容是一个对象{}，然后通过new生成这样一个对象，B函数成为这个对象的一个方法；

如代码：

var func = function () {  
 var i = 0;  
 return {  
 getI: function () {  
 return i;  
 },  
 add: function () {  
 i++;  
 }  
 }  
}  
var p = new func();  
p.add();  
**console**.log(p.getI());

此时，p.add()会导致i的值增加，此时i的值变为了1。

**方法二：**让A函数的return部分，成为一个函数，

但不能直接操纵A函数，因为调用A函数必然会导致A函数的return执行完毕，（因为函数会执行到return结束）

因此，需要将A函数的return部分赋值给另外一个变量C，此时C是A函数return部分（要记得，这个return部分是一个函数）的一个实例（相当于一个构造函数生成一个实例）。

而B函数是这个return部分的构造函数的一个方法，因此这个生成的实例也拥有了B函数这个方法。

如代码：

var func = function () {  
 var i = 0;  
 return function(){  
 i++;  
 return i;  
 }  
}  
var p = new func();  
**console**.log(p());  
**console**.log(p());

这时，p的调用会导致i加一，因此调用p，输出的值分别为1和2。

ps：无论是哪种方法，假如我们再new一个实例

var q = new func();

其并不会导致i的值在p和q之间共享。

④优点：

按照网易云课堂的说法，闭包可以减少内存使用，提高效率。（个人经过实测，认为没有说服性）

但是经过我实测，大部分方法和闭包差不多，比闭包慢的一个方法我后面列出。

如：

var func = function () {  
 var i = 0;  
 return function () {  
 i++;  
 return i;  
 }  
}  
var starttime = new Date();  
var p = new func();  
for(var i=0;i<1000000;i++){  
 p();  
}  
**console**.log(p())  
**console**.log(new Date()-starttime);

计算一百万次，消耗时间大概10毫秒左右。

这是我看网易云课堂给的一个示例（我自己略有修改）；

function sum(i) {  
 var add = function (i) {  
 i++;  
 return i;  
 }  
 return add(i);  
}  
var starttime = new Date();  
for (var i = 0; i < 1000000; i++) {  
 sum(1);  
}  
**console**.log(sum(1));  
**console**.log(new Date() - starttime);

其时间耗时为30毫秒左右。

但这个例子事实上返回了2次，所以我觉得不能证明闭包更快。例如，常规写法：

function sum(i) {  
 i++;  
 return i;  
}

其耗时就和闭包差不多。我把计算量增加10倍（到一千万次），其时间和上面那个所说的比较慢的方法是差不多的（30ms左右）。

因此个人认为，应该是双重return所导致了耗时增多。

**⑤闭包的利用：**

假设我们有N个按钮，想点击第x按钮时，输出x。

var m = function () {  
 function cli(i){  
 return function(){  
 **console**.log(i);  
 }  
 }  
 for (var i = 1; i < 3; i++) {  
 **document**.getElementById("test" + i).**onclick** = cli(i);  
 }  
}  
m();

初始时，不输出任何变量。点击id="test1"，输出1，点击id="test2"，输出2。

假如逐个绑定，那么是不能达成这种目的的。例如：

for (var i = 1; i < 3; i++) {  
 $("#test" + i).click(function () {  
 **console**.log(i)  
 });  
}

并不能如我们所愿那样，点击第一个按钮便输出1，事实上，无论点哪个，输出都是3。

**（19）bind()绑定事件**

①简单来说，bind是函数的一个方法（继承来的）；

其使用方法是：

函数名.bind(调用函数的对象, 调用函数时使用的参数);

例如：

var i = 0;  
function getValue(m) {  
 i = m;  
}  
var o = getValue.bind(i, 5);  
**console**.log(i);  
o();  
**console**.log(i);

第一次输出0，第二次输出5。

②bind的返回值是一个函数的引用。

调用它的时候，就用返回值加括号()来调用。

③有点类似用jquery方法绑定dom结点和事件；

事件是函数，可能有参数，然后dom结点是返回值；

点击/或者其他操作对这个dom结点，便是对这个返回值后面加了()来执行这个bind函数。

**（20）函数的原型**

**①Object.create(obj)**

根据原型obj创建一个对象；

例如：

var obj = {  
 **a**: 1  
}  
var o1 = Object.create(obj);  
**console**.log(obj);  
**console**.log(o1);

输出为：

Object {a: 1}

Object {}

②使用构造函数方法：

使用构造函数，通过new来创建一个实例（之前有，这里略）

③

**①和②之间的区别：**

通过②的继承：

var obj = {  
 **a**: 1  
}  
var o1 = Object.create(obj);  
var o2 = Object.create(o1);  
o1.**b** = 5;  
o2.**c** = 2;  
**console**.log(o1.**b**);  
**console**.log(o2.**b**);

o1和o2都有属性b，并且值为5；

但是，通过继承来的，则不一样。

var obj = function () {  
 **a**: 1  
}  
var o1 = function(){}  
o1.**prototype** = new obj();  
var o2 = function(){}  
o2.**prototype** = new o1();  
o1.**b** = 5;  
o2.**c** = 2;  
**console**.log(o1.**b**);  
**console**.log(o2.**b**);

输出是5和undefined

通过查看console属性可以发现，Ojbect.create，其原型链是直接链接到创建时的原型对象的。

例如假设o2没有属性b，那么其值则为o1的属性b的值，假如o1.b的值被更改，那么o2.b的值也会被更改。

如代码：

var obj = {  
 **a**: 1  
}  
var o1 = Object.create(obj);  
var o2 = Object.create(o1);  
o1.**b** = 3;  
**console**.log(o2.**b**);  
o1.**b** = 5;  
**console**.log(o2.**b**);

其输出的值分别是3和5

④hasOwnProperty()

通过Object.create()方法声明的变量，其原型的属性，在自身调用 hasOwnProperty()会提示false

var obj = {  
 **a**: 1  
}  
var o1 = Object.create(obj);  
var o2 = Object.create(o1);  
o1.**b** = 5;  
**console**.log(o2.hasOwnProperty("b"));  
o2.**b** = 2;  
**console**.log(o2.hasOwnProperty("b"));

例如，第一个是true，第二个是false

但需要注意，这种方法无法通过

o2.**prototype**.**b** = 5;

来为之设置属性

而另一种方法和其的区别在于，可以通过o2.prototype.b = 5;来设置属性，以及继承。

**（21）函数的作用域**

①静态作用域：（JS使用的是这种）

var x = 1;  
function a() {  
 **console**.log(x);  
}  
function b() {  
 var x = 2;  
 a();  
}  
b();

输出结果是1，而不是2。

因为a()定义在window里，而不是在b()里，所以使用的是window中的x，而不是b中的x，如以下代码输出结果便为2：

var x = 1;  
function b() {  
 var x = 2;  
 function a() {  
 **console**.log(x);  
 }  
 a();  
}  
b();

②JS变量的作用域：全局作用域、函数作用域。没有块级作用域（即{}之间声明的变量的作用域）；

③with方法的临时作用域：

var x = 1;  
with ({**x**: 2}) {  
 function a() {  
 **console**.log(**x**);  
 }  
 (function () {  
 **console**.log(" " + **x**); //输出2  
 a() //输出1  
 })();  
}  
**console**.log(**x**); //输出1

之所以，是因为匿名函数可以理解为在函数内部创建并执行的（类似一个表达式），因此其内部变量的作用域是在with环境创建的临时作用域。（这个作用域，是x=2的作用域）

而a函数是声明的，因此其作用域是全局的（因为没有块级作用域，只有全局和函数内部的）。（函数被预解析时，知道其作用域是全局的）

由于with环境作用域是临时的，并且声明的是一个对象，因此不会对全局变量x进行修改。

**（22）面向对象**

**①全局变量**

定义方式：

var a = 1; //方法一：常规  
function m() {  
 b = 1; //方法二：函数内，没加var声明一个变量  
 **window**.**c** = 1; //方法三：window下声明一个变量  
}  
m();  
**console**.log(a);  
**console**.log(b);  
**console**.log(c);

这三种方法是常规的声明。

以下方法，n是全局变量，m是函数内变量

function a() {  
 var m = **n** = 1;  
}  
a();  
**console**.log(m);  
**console**.log(**n**);

delete 全局变量：

var a = 1; //方法一：常规  
function m() {  
 b = 1; //方法二：函数内，没加var声明一个变量  
 **window**.**c** = 1; //方法三：window下声明一个变量  
}  
m();  
**console**.log(a);  
**console**.log(b);  
**console**.log(c);  
delete a;  
delete b;  
delete c;  
**console**.log(a);  
**console**.log(b);  
**console**.log(c);

结果：

a未被delete掉；

b和c在delete掉后会报错：

Uncaught ReferenceError: c is not defined

**②信息隐藏**

方式：封装

以下这种方法是无法封装的：

function A() {  
 this.**a** = 1;  
 this.**b** = 1;  
}  
var m = new A();  
**console**.log(m.**a**);

最后一步可以输出1

而以下这种方式是可以封装的

function A() {  
 var a = 1;  
 this.getA = function () {  
 return a;  
 }  
}  
var m = new A();

**console**.log(m.**a**); //无输出，提示出错（必须注释掉）  
**console**.log(m.getA()); //输出1