javascript

1.语法

1.1 基本语法

```
// 1.定义变量  变量类型 变量名 = 变量值;
var num = 1;
var name = "yusiquweierwang";
var score = 66;
// 2.条件控制
if (2 > 1) {
  alert("female");
} else if (score > 40) {
  alert("fe");
}
//javaScript严格区分大小写

//console.log()在浏览器控制台打印变量
```

```
//javascript 不区分小数和整数，number,Number
123; //整数123
123.3; //浮点数123.3
1.2e2 - //科学计数法
  99; //复数
NaN; //not a number
Infinity; //无穷大

//    字符串'abc' "bad"
//布尔值
//逻辑运算
//比较运算符
//==:类型不一样，值一样，true
//===:类型一样，值一样，true
//NaN===NaN:与所有数值都不相等，isNaN来判断

//浮点数问题：console.log(1/3) false  -->因为存在精度问题
console.log(1 / 3) === console.log(1 - 2 / 3);
console.log(Math.abs(1 / 3 - (1 - 2 / 3)) < 0.000001);

//undefined 未定义
// null空

var arr = [1, 2, 3, 4, 5, "hello", null, true];
//java中必须列相同类型的对象，javascipt中不需要
//取数组下标，如果越界了，会:-->undefined

//    对象：对象是大括号，数组是中括号
var person = {
  name: "yusiquweierwang",
  age: 3,
  tags: ["js", "java", "web", "..."],
};
```

严格检查模式：'use strict';

模板字符串

可包含特定语法：(${expression})的占位符。

```
var a=5;
var b=20;
console.log(`twenty-five is ${a+b} and
not ${2*a+b}.`);
//"twenty-five is 25 and not 30."
```

1.2 标识符(identifier)

指用来识别各种值的合法名称，常见标识符就是变量名，及函数名。JS 对标识符大小写敏感。

JS 中有一些不能作标识符的保留字： arguments、break、case、catch、class、const、continue、debugger、default、delete、do、else、enum、 eval、export、extends、false、finally、for、function、if、implements、import、in、instanceof、interface、let、 new、null、package、private、protected、public、return、static、super、switch、this、throw、true、try、typeof、 var、void、while、with、yield。

1.3 条件语句

JS 提供 `if`结构和 `switch`结构。

```
if(bool){
    语句1;
}
```

```
switch(fruit){
    case:'banana';
    //...
    break;
    case:'apple';
    //...
    break;//不写break代码块会继续往下执行，而非跳出switch结构
    default:
    //...
}
```

三元运算符： (条件)?表达式 1:表达式 2;

eg:

```
var 判断奇偶 = "math" + n + "is" + (n % 2 === 0 ? "even" : "odd");
```

1.5 循环

1.5.1 标签 `label`定位符，用于跳转到程序的任意位置

```
label: 语句;
```

- break 与标签配合使用

```
top: for (i = 0; i < 3; i++) {
  for (j = 0; j < 3; j++) {
    if (i === 1 && j == 1) break top;
    {
      console.log("i=" + i + "j=" + j);
    }
  }
}
// i=0j=0
// i=0j=1
// i=0j=2
// i=1j=0
```

上述如果 break 后面不加标签，则只能跳出内层循环，进入下一次外层循环。

- continue 与标签配合使用 不加，则进入下一个内层循环，加了跳出此内层循环。

for of

专门用于可迭代对象（Array,Set,Map,String,Argument）

若只想迭代键或值，则用keys()或values()方法。

for in 可迭代对象的属性

Object.entries()

返回一个给定对象自身可枚举属性的键值对数组，（for ..in 循环还会枚举原型链中的属性）。

```
const object={
    a:'something',
    b:43
};
for(const[key,value] of Object.entries(object)){
    console.log(`${key}:${value}`);
}

//"a:something"
//"b:43"
```

用处：把对象转换为真正的map 结构。

Map() Set()

ES6内置的set(集合)和map(映射)类是可迭代的，

Set:

集合：

1. 集合没有索引或顺序，也不允许重复：一个值要么是集合的成员，要么不是；此值只在集合中出现一次。

创建：

```
let s=new Set();
```

属性：

size(类似数组length)

方法：

1.add()

2.delete()

1. set类可迭代，可以使用for/of循环枚举集合的所有元素：

```
let oneDigitPrimes=new Set([2,3,5,6]);
let sum=0;
for(let p of oneDigitPrimes){
    sum+=p;
}
sum;//16
```

下面代码打印文本字符串中唯一单词：

```
let text='Na na na na batman';
```

```
let wordSet=new Set(text.split(' '));
//String.prototype.split()方法通过设定分隔符来分割字符串。。
let unique=[];
for(let word of wordSet){
    unique.push(word);
}
//Na na batman
```

```
        let text='i will survive i will survive';
        let iSplit=text.split('i');
        for(i of iSplit){
            console.log(i);
        }
        let iSplitSet=new Set(text.split('i'));
        for(i of iSplitSet){
            console.log(i);
        }

//  w
// ll surv
// ve
//  w
// ll surv
// ve

//  w
// ll surv
// ve
// ve
```

Map类

map对象表示一组被称为键的值，其中每个键都关联（或映射）另一个值。

方法：

get()和键：查询关联的值；

set():添加新的键/值对。

```
//map()
var map = new Map([
  ["Tom", 100],
  ["Jack", 90],
  ["Smith", 74],
]);
var Tom_name = map.get("Tom"); // 通过key 获得value
map.set("admin", 23);
```

```
console.log(Tom_name); //输出100
var admin = map.get("admin");
console.log(admin); //输出23
//set:无序不重复的集合
var set = new Set([3, 1, 1, 1, 1]); // set 可以去重
set.add("more", 2);
```

遍历 map

```
var map = new Map([
  ["tom", 20],
  ["jack", 30],
  ["mark", 90],
]);
for (let [key, value] of map.entries()) {
  console.log(key + " " + value);
}
```

将Object转换为Map

new Map() 构造函数接受一个可迭代的entries ， 借助Object.entries方法可容易的将Object转为Map:

```
var obj={foo:'bar',baz:32};
var map=new Map(Object.entries(obj));
console.log(map);
```

实现可迭代对象

es6中引入了迭代器可可迭代对象及其支持，如for of 循环，Map(iteable)构造器。 迭代器原理类似指针，指向数据集合中的某个元素，也可移动以获取其他元素。

JS中每次获取到的迭代器总是初始指向第一个元素，且迭代器只有next()一种行为。 所以，**迭代器任务是按某种次序遍历数据集中的元素。 允许调用next()方法，每次调用都返回一个对象 `{value,done}`,当返回done值为true,value为每次获取的元素。** 具有迭代器的对象：Array,Set,Map,类数组对象：Arguments,NodeList...

- 方法一：Symbol()
- 方法二：封装一个迭代器Generator()函数

```
```

```
let obj={
  "0":"tom","1":"jerry","2":"terry"
}
//封装一个keys迭代器
function* keys(){
  for(let key in this){
    let value=this[key];
    yield value;
    //TODO:看看这里的this是怎么起作用的
  }
}
//封装一个value迭代器
function* values(){
  for(let key in this){
    let value=this[key];
    yield
  }
}
//封装一个entries()迭代器
function* entries(){
  for(let key in this){
    let value=this[key];
    yield [key,value]
  }
}

//调用
obj[Symbol.iterator]=entries;
let entry=obj[Symbol.iterator]();
entry.next() //{value:["0","tom"],done:false}
entry.next() //{value:["1","jerry"],done:false}
entry.next() //{value:["2","terry"],done:true}
```

1.1.6 for/await 和异步迭代

```
async function printStream(stream){
    for await(let chunk of stream){
        console.log(chunk);
    }
}
```

1.6 跳转语句

continue

break

return

yield

yield 只能用在ES6新增的生成器函数中，以回送生成的值序列中的下一个值，又不会真正返回。

```
function * foo(x){
    while(true){
        x=x*2;
        yield x;
    }
}

//当调用foo时，会得到一个具有next 方法的Generator对象。
var g=foo(2);
g.next();//4
g.next();//8
```

即yield 类似return ，但return 返回值x,而yield 返回一个函数，该函数提供一个迭代下一个值的方法。

yield关键字只是帮助异步地随时暂停和恢复功能。

另外，它有助于从生成器函数返回值。

throw

抛出异常表明发生了错误或意外情形，catch（捕获）异常则要处理它。

try/catch/finally

debugger

包含debugger的程序在运行时，可执行某些调试操作。类似一个断点

1.2.数据类型

7 种数据类型

- string
- number
- undefined
- null
- boolean
- object
- Symbol

javascript 确定数据类型方法

- typeof 用于判断语句

```
if(b){
  //...
}
Reference:b is not defined
if(typeof b==='undefined'){
  //...
}
```

- instanceof

用于检测构造函数的prototype属性是否出现在某个实例对象的原型链上。

```
function Car(make,model,year){
    this.make=make;
    this.model=model;
    this.year=year;
}
const auto=new Car('honda','accord',1887);

console.log(auto instanceof Car);

//true
```

- Object.prototype.toString

返回一个表示该对象的字符串。

1.3 数值

解构

数值转换

parseInt 将数值转换为整数 parseFloat 将数值转换为浮点数 · ###进制转换

```
parseInt("1000", 2); //parseInt('数值',进制);
```

判断数值

isNaN()

1.4 对象

即一组键值对的集合，一组无序复合数据集合、

相关操作 创建，设置，查询，删除，测试，枚举它们的值

1.如果属性的值还是一个对象，就形成了一个链式引用。

```
var o1 = {};
var o2 = {
  pa: function (x) {
    return x * x;
  },
};
o1.foo = o2;
o1.foo.pa();
```

2.对象的引用 如果不同变量名都指向一个对象，则它们都是这个对象名的引用，都指向同一个内存地址。修改其中一个，会影响其他所有变量

```
var p1={
  p11=3;
```

```
}
var o1=p1;
var o2=p1;
o1.p11=4;
```

3.表达式和语义 {foo:33}

- 表示包含 foo 属性的对象
- 表示一个代码区块，里面有一个 foo标签，指向表达式 33.

为避免歧义，一般默认{}为前者意思。

例：eval 语句。

4.对象的属性删除 delete

5.for.. in 循环遍历一个对象的全部属性。

1.4.2 读取属性

- 点运算符 pa.ji;
- 方括号运算符 pa['ji'];

1.4.3 查看对象所有属性

Object.key()方法；

```
var p1 = {
  //....
};
Object.key(p1);
```

1.4.4 with 语句

扩展一个语句的作用域链

将某个对象添加到作用域的顶部，如在statement中有某个未使用命名空间的变量，跟作用域链中的某个属性同名，则此变量将指向此属性值。如没有同名属性，则抛出referenceError异常。

不建议使用

```
with(expression){
    statement
}
```

1.5 数组

JavaScript 定义（创建或者声明）数组的方法有两种：构造数组和数组直接量。

- 构造数组

```
var array = new Array(); //创建一个空数组。

//传入参数
let a=new Array(20);//这样会传入一个指定长度为20 的数组;
```

Array.of

可以使用其参数值（无论多少个）作为数组元素来创建并返回新数组。

```
Array.of(1,2,3)

//return [1,2,3]
```

- 数组直接量

```
var array = [];
```

###1.5.1 for..in 遍历数组

for.. in 不仅可以遍历数组所有数字键，还能遍历非数字键。

```
var a = [3, 45, 643];
a.ee = true;
for (var i in a) {
  console.log(a[i]);
  /*
```

```
        return 3,45,643,true
        */
}
```

1.5.2 forEach()方法

对数组的每个元素执行给定一次callback函数。

同时对已删除或未初始化的项将被跳过（如稀疏数组）。

可依次向callback 函数传入3个参数：

1.数组当前项的值。

2.数组当前项的索引。

3.数组对象本身。

```
array.forEach(element=>console.log(element));
```

map 和 foreach 的区别：

foreach不会返回执行结果，而是undefined 。

即foreach修改原来的数组，而map 得到一个新数组返回。

filter() & find()

find()在断言函数找到第一个元素时停止迭代。

every()＆some()

every()类似全称量词，

some()类似存在量词。

reduce()

使用指定函数归并数组元素，最终产生一个值。

```
let a=[1,2,3,4,5];
a.reduce((x,y)=>x+y,0)//15
a.reduce((x,y)=>x*y,1)//120
```

第一个参数是执行的函数，第二个参数是初始值。

flat() 和 flatMap() 打平数组

按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并到一个新数组并返回。

concat() 添加数组

创建并返回一个新数组

```
let a=[1,2,3];
a.concat(4,5);
//[1,2,3,4,5]
```

slice() splice() fill()

slice():

```
let a=[1,2,3,4];
a.slice(0,2);//return [1,2];
a.slice(-)
```

1.5.3 类似数组的对象

如果一个对象的所有键名都是正整数或零，并且有 length 属性，那么这个对象就很像数组，语法上称为“类似数组的对象”（array-like object）。

```
var obj = {
  0: "a",
  1: "b",
  2: "c",
  length: 3,
};

obj[0]; // 'a'
obj[1]; // 'b'
obj.length; // 3
obj.push("d"); // TypeError: obj.push is not a function
// 1.类对象不是数组，因此使用数组的push 方法会报错。
// 2.类对象根本特征就是具length属性，但length不会动态改变
```

典型类数组对象:

- 函数 arguments 对象
- DOM 元素集
- 字符串

转为真正的数组 `slice` 方法可将类数组对象变为真正的数组

```
var arr = Array.prototype.slice.call(arrayLike);
```

类数组对象可以使用数组 通过 call()方法把数组方法放到对象上面.

```
function print(value, index) {
  console.log(index + ":" + value);
}
Array.prototype.forEach.call(类数组对象);
```

字符串应用 call()遍历：

```
Array.prototype.forEach.call("yusiquweierwang", function (chr) {
  console.log(chr);
});
y;
//  u
//  s
//  i
//  q
//  u
//  w
//  e
//  i
```

```
// e
// r
// w
// a
// n
// g
```

1.5.4 多维数组

javascript 中不能直接定义多维数组

方法1：

```
let arr=new Array();
for(let i=0;i<2;i++){
    arr[i]=new Array();
    for(let j=0;j<3;j++){
        arr[i][j]=i*j;
    }
}
```

定形数组

定形数组类型：

构造函数	数组类型
Int8Array()	有符号字节
Uint8Array()	无符号字节
Uint8ClamperArray()	无符号字节（上溢不归零）

2 函数

1. 函数既是函数又是对象；

在控制台console.dir发现函数既有prototype又有_proto_

prototype是构造函数的；

_proto_是实例对象的；

因为每个函数都是一个Function对象；

```
var fn=new Function('a','b','c');
```

this

this指向执行时的环境：再全局函数中，this=window;

当函数被作为某个对象调用时，this等于那个对象。

（匿名函数this指window)

scene1.全局环境的this:

函数在浏览器全局环境中被简单调用，非严格模式下this指向window,'use strict'模式下undefined:

```
function f1(){
    console.log(this);
}
function f2(){
    'use strict'
    console.log(this)
}
f1()  //window
f2()  //undefined
```

```
const foo={
    bar:10,
    fn:function(){
        console.log(this)
        console.log(this.bar)
    }
}
var fn1=foo.fn
fn1()
```

此处this仍指向window,但赋值fn1后，fn1执行在window全局环境中

```
```

```
const foo={
    bar:10,
    fn:function(){
console.log(this);
    console.log(this.bar);
    }

}
foo.fn();

//输出
{bar:10,fn:f}
10
```

以上，在执行函数时，如果函数中**this** 是被上一级对象所调用，则**this**指向的是上一级的对象；否则指向全局对象。

scene2:

上下文对象调用中的this

```
const person={
    name:'Lucas',
    brother:{
        name:"Mike",
        fn:function(){
            return this.name;
        }
    }
}
console.log(person.brother.fn());
```

more complex?

```
const o1={
    text:'o1',
    fn:function(){
                return this.text;
    }
}
const o2={
    text:'o2',
    fn:function(){
        return o1.fn()
    }
}
const o3={
    text:'o3',
    fn:function(){
```

```
        var fn=o1.fn
        return fn()
    }
}
console.log(o1.fn())
console.log(o2.fn())
console.log(o3.fn())
//答案：o1,o1,undefined
```

面试时，若需要让

```
console.log(o2.fn())
```

输出o2,

1. 使用bind,call,apply对this指向进行干预

```
const foo={
    name:'Lucas',
    logName:function(){
        console.log(this.name)
    }
}
const bar={
    name:'mike'
}
console.log(foo.logName.call(bar))

//输出mike
```

1. this指向最后调用它的对象：

```
const o1={
text:'o1',
fn:function(){
    return this.text;
}}
const o2={
    text:'o2',
    fn:o1.fn;
}
console.log(o2.fn())
```

scene4.构造函数和this

```
function Foo(){
        this.bar='Lucas',
}
const instance=new Foo();
console.log(instance.bar);
```

问题：new 操作符调用构造函数，具体做了什么？

1. 创建一个新的对象。
2. 将构造函数的this 指向此新对象。
3. 为此对象添加属性和方法。
4. 最终返回新对象。

上过程如下：

```
var obj={}
obj._prototype_=Foo.prototype;
Foo.call(obj);
```

在构造函数中若出现了显式return情况，两种场景：

```
function Foo(){
    this.user='Lucas',
    const o={},
    return o;
}
const instance=new Foo();
console.log(instance.user);

//将会输出undefined,此时instance 返回的是空对象o
```

```
function Foo(){
    this.user='Lucas'
    return 1;
}
const instance=new Foo();
console.log(instance.user);

//将会输出Lucas,即此时instance 返回目标对象实例this
```

scene5.箭头函数的this 指向

1. 箭头函数中**this**指向的是定义时的**this,**而不是执行时的**this.**

2. **根据外层作用域决定。**0

```
const foo={
    fn:function(){
        setTimeout(function(){
            console.log(this)
        })
    }
}
console.log(foo.fn())

//此处this出现在setTimeout()中的匿名函数里，而setTimeout()是window对象的方法，因此this指向window对象
```

```
const foo={
        fn:function(){
        setTimeout(()=>{
            console.log(this)
        })
    }
}
console.log(foo.fn())

//{fn:f}
```

scene6.this优先级相关

显示绑定：call,apply,bind,new 对 this 的绑定；

隐式绑定：根据调用关系确定的this。

优先级：

2.1.定义函数

2.1.1 function 命令

```
function abs(x) {
  //不存在参数时的规避问题：抛出异常；
  if (typeof x !== "number") {
    throw "Not a Number";
  }
```

```
  //arguments代表传递进来的所有参数是一个数组
  console.log("x=>" + x);
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }

  if (x > 0) {
    return x;
  } else {
    return -x;
  }
}
```

2.1.2 函数表达式

```
var print=function('333'){
  console.log('333');
}
```

此方法将一匿名函数赋值给变量，此匿名函数称 function expression>

```
var x=function x(){
  表达式;
}
```

此写法:

- 利于在函数体内部调用自身
- 方便除错

2.1.3 function 构造函数

```
var add=new function(
  'a',
  'b',
  'return a+b',
)
//equals to
var add=funtion(a,b){
  return a+b;
}
```

2.1.4 name 属性

用处：获取参数的名字。

```
var myFunction;
var myFuntion = function add() {};
function test(f) {
  console.log(f.name);
}
test(myFunction); //return add
```

2.1.5 length 属性

返回函数定义之中的参数个数

```
function s(a, b) {
  //表达式;
}
console.log(s.length);
//return 2
```

2.1.6 toString();

返回函数的源码

2.1.7 call()和apply()

可用来重新定义函数的执行环境。即this的指向。

call()

调用一个对象的方法，用另一个对象替换当前的对象，可继承另一个对象的属性

语法：

```
function.call(obj[,param1[,param2[,[,...[paramN]]]]]);
```

apply()

```
Function.apply(obj[,argArray]);
```

- obj:此对象将代替Function类里的this对象。
- argArray:这是个数组，他将作为参数传给Function.

bind()

创建一个新的函数，在bind()被调用时，此新函数的this被指定为bind()的第一个参数；

函数柯里化

利用闭包，可形成一个不销毁的私有作用域，把预先处理的内容都存在此不销毁的作用域里面，且返回一个函数，以后要执行的就是此函数。

2.2 函数作用域 scope

1.函数作为命名空间：

```
function chunkNamespace(){
    //要复用的代码放于此
    //在此定义的任何变量都是函数的局部变量
    //不会污染全局命名空间
}

chunkNamespace();
```

立即调用函数表达式：

```
(function(){

}());//函数定义结束后立即调用它;
```

作用域链

自由变量：

```
var a=100;
function fn(){
    var b=200;
    console.log(a);//a在这里就是一个自由变量
    console.log(b);
}
fn();
```

作用域和执行上下文

javascript执行分为：解释和执行两个阶段；

解释阶段：

1.词法分析；

2.作用域规则确定；

执行阶段：

1.创建执行上下文；

2.执行函数代码；

3.垃圾回收；

区别：执行上下文在运行时确定，随时可能改变；作用域在定义时就确定，并且不会改变。

1.函数内部的变量提升

2.函数体本身作用域提升

函数执行时定义域是定义时作用域而不是调用时的作用域。|||/ 例：

```
var add = function () {
  console.log(a);
};
```

```
function bdd(f) {
  var a = 2;
  f();
}
bdd(add);
```

上述报错，因为函数 add 是在函数 bdd 体外生命的，作用域绑定外层，因此找不到函数 bdd 内部变量 a，报错。

2.3 参数

2.3.1 参数省略

```
function f(a,b){
  return a;
}
f(,1);//syntaxError:Unexpected token
f(undefined,1);//bingo
```

2.3.2 传递方式

- 函数参数若是原始类型的值 number,string,boolean ,则传递方式为按值传递(passer by value),即在函数体内修改值，不会影响函数外部。

```
var a = 3;
function f(a) {
  var a = 5;
}
console.log(a);
//return 3;
```

- 按地址传递 函数参数是复合类型的值 数组，对象，其他函数，则传递方式为按地址传递，即传入函数的原始值的地址。即函数内部修改参数，会影响到原始值

```
var a={
  p:1;
}
function o(arg){
  arg.p=3;
```

```
}
o(a);
a.p;//return 3;
```

有同名参数，则取后面的参数的值

2.4 arguments 对象

1.arguments 对象包含了函数运行时的所有参数 2.正常下 arguments 对象运行时可修改。

```
var f = function (a, b) {
  arguments[0] = 3;
  argumens[1] = 5;
  return a + b;
};
f(1, 1); //8
```

严格模式下不能修改

```
var f = function (a, b) {
  "use strict";
  arguments[0] = 3;
  argumens[1] = 5;
  return a + b;
};
f(1, 1); //8
```

3.arguments.length 查看函数有几个参数；

2.4.1 让 arguments 对象使用数组的方法：

```
var args = Array.prototype.slice.call(arguments);
//or
var args = [];
for (var i = 0; i < arguments.length; i++) {
  args.push(arguments[i]);
}
```

2.4.2 callee 属性返回 arguments 对象所对应的原函数。

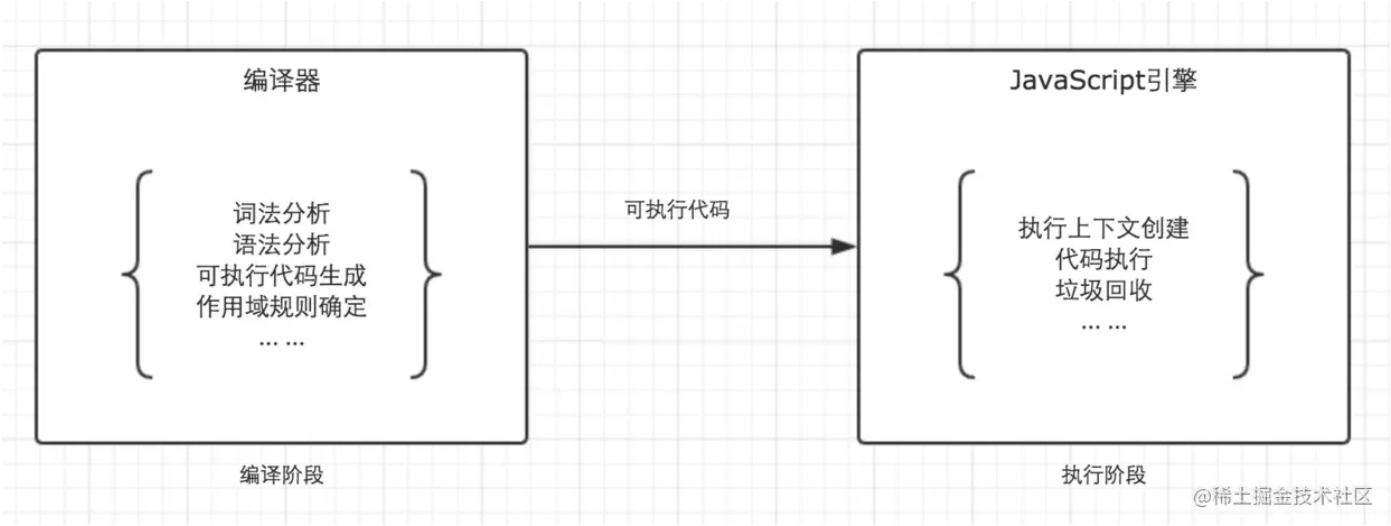
2.5 闭包

javascript使用词法作用域（lexical scoping),规则：函数执行时使用的是定义函数时生效的变量作用域，而不是调用函数时生效的的变量作用域。

闭包是指有权访问另外一个函数作用域中的变量的函数。

为实现词法作用域，javascript函数对象内部状态不仅要包括函数代码，还要包括对函数定义所在作用域的引用。

此类函数对象和作用域（即一组变量绑定）组合起来解析函数变量的机制，称为闭包（closure）。



"定义在一个函数内部的函数"

\

本质：当前环境中存在指向父级作用域的引用。

应用场景：

柯里化bind

模块

作用：

- 读取外层函数内部的变量
- 让这些变量始终保存在内存中，使内部变量记得上一次调用时的运算结果。因此闭包可看作函数内部作用域的一个接口。

```
function createIncrementor(start) {
  return function () {
    return start++;
  };
}
var inc = createIncrementor(5);
inc(); // 5
inc(); // 6
inc(); // 7
```

闭包（上例 inc()）能返回外层函数的内部变量的原因： 闭包（上例 inc()）用到了外层变量 start，导致 `createIncrementor`不能释放内存

- 封装对象的私有属性和私有方法

```
function Person(name) {
  var _age;
  function setAge(n) {
    _age = n;
  }
  function getAge() {
    return _age;
  }

  return {
    name: name,
    getAge: getAge,
    setAge: setAge,
  };
}

var p1 = Person("张三");
p1.setAge(25);
p1.getAge(); // 25
```

```
function fn1(){
    let n=124;
    ruturn function(){
        console.log(n)
    }
}

//这个被ruturn 返回的匿名函数就是闭包
```

```
let _f1=fn1();
_f1();//此时_f1里保存的是匿名函数在堆内存中的引用，在函数之外通过调用引用，而匿名函数依然还是在fn1()里面执行。
```

实例：

```
var str='i m you ';
var _obj={
    str:'what',
    getData:function(){
        return this.str;
    }
}
console.log(_obj.getData()())
```

2.6 立即调用函数 IIFE

function既可以做语句，也可以做表达式，因此会产生歧义。

```
// 语句
function f() {}

// 表达式
var f = function f() {};
```

```
(function(){}());
```

2.7 eval 命令

eval 命令接受一个字符串作为参数，并将此字符串当作语句执行。

```
eval("var a=0;");
a; //return 0;
```

2.8 回调函数

3 错误处理机制

3.1 Error 实例对象

JS 原生提供 `Error` 构造函数，所有抛出错误都是此函数的实例。

```
var err = new Error("wrong!");
err.message; //wrong!
```

- **Error 实例对象必须有 message 属性**
- **name:错误名称（非标准属性）**
- **stack:错误堆栈（非标准属性）**

message 和 name 属性对错误进行大概了解，stack 属性用来查看错误发生时的堆栈。

```
function throwit() {
  throw new Error("");
}

function catchit() {
  try {
    throwit();
  } catch (e) {
    console.log(e.stack); // print stack trace
  }
}

catchit();
// Error
//     at throwit (~/examples/throwcatch.js:9:11)
//     at catchit (~/examples/throwcatch.js:3:9)
//     at repl:1:5
```

上面代码中，错误堆栈的最内层是 throwit 函数，然后是 catchit 函数，最后是函数的运行环境。

3.2 原生错误类型

- SyntaxError 对象 解析代码时发生的语法错误。

```
var 1d;
//Invalid or unexpected token;
console.log'ffe');
//unexpected string
```

- Reference Error 对象 引用一个不存在变量时发生的错误
- Range Error 对象
    - 数组长度为负数
    - `Number`对象方法参数超过了范围，以及函数堆栈超过最大值
- TypeError 对象 对象是变量或参数不是预期参数时发生的错误。
- URLError 对象

3.3 自定义错误

3.4 throw 语句

`throw` 语句作用：手动中断程序执行，抛出一个错误。 `throw`可自定义抛出错误：

```
function userError(message) {
  this.message = message;
  this.name = "userError";
}
throw new userError("wrong");
//Uncaught userError{message:'wrong';name:'userError};
```

3.5 try...catch 结构

JS 提供 `try...catch`结构，选择是否往下执行程序。

```
try{
  throw new Error('wrong!');
} catch(n){
  console.log(n.name+':'+n.message);
  console.log(n.stack);
```

```
}
//Error:wrong!;
//at <anonymous>:4:9
//...
```

加入判断语句可捕获何类型的错误：

```
try {
  foo.bar();
} catch (e) {
  if (e instanceof RangeError) {
    console.log(e.name + e.message);
  } else if (e instanceof EvalError) {
    console.log(e.name + e.message);
  }
  //...
}
```

3.6 finally 代码块

下面代码因为不含 catch 代码块，一旦发生错误，会停止执行，中断执行前，会先执行 Finally 代码块，再向用户提示报错信息。

```
function clean() {
  try {
    throw new Error("wrong");
    console.log("此行不会执行");
  } finally {
    console.log("Finally");
  }
}
clean();
//Finally;
//wrong;
```

4 console 对象和控制台

console.log 方法支持以下占位符，不同类型的数据必须使用对应的占位符。

%s 字符串 %d 整数 %i 整数 %f 浮点数 %o 对象的链接 %c CSS 格式字符串

eg

```
console.log(" %s + %s = %s", 1, 1, 2);
//  1 + 1 = 2

var number = 11 * 9;
var color = "red";
console.log("%d %s balloons", number, color);
// 99 red balloons
```

3.1 console 对象的静态方法

3.1.1 console.table()

将某些复合类型数据转为表格显示。

```
var languages = [
  { name: "JavaScript", fileExtension: ".js" },
  { name: "TypeScript", fileExtension: ".ts" },
  { name: "CoffeeScript", fileExtension: ".coffee" },
];

console.table(languages);
```

输出为

(index)	name	fileExtension
0	"JavaScript"	".js"
1	"TypeScript"	".ts"
2	"CoffeeScript"	".coffee"

3.1.2 console.count()

用于计数，输出它被调用了多少次。

```
function punch(person) {
  console.count();
```

```
  return "punch" + person;
}
punch("Jack");
//1
//punch Jack;
punch("Bob");
//2
//punch Bob;
```

3.1.3 console.dir() & console.dirxml()

console.dir 用来对一个对象进行检查，并以用于阅读和打印的格式显示。 通常用于输出 DOM 对象，因为会显示 DOM 对象的所有属性。

3.1.4 console.assert() 1.用于程序执行出错时，汇报出错状态，但不中断程序运行。 2.接受两个参数，第一个参数是表达式，第二个参数是字符串。

只有当第一个参数为 false，才会提示有错误，在控制台输出第二个参数，否则不会有任何结果。

```
console.assert(list.childNodes.length < 500, "节点个数大于等于500");
```

3.1.5 console.time() & consoel.timrEnd()

```
console.time("array initialize");
var array = new Array(1000);
for (var i = array.length; i > 0; i--) {
  array[i] = new Object();
}
console.timeEnd();
```

3.1.6 console.group()

3.1.7 **console.trace() **

显示当前执行的代码在堆栈中的调用路径

4.对象

1.JS 原生提供 Object 对象，其他对象都继承自 Object 对象，即是 Object 对象的实例。 2.Object 对象有两种方法：Object 本身的方法和 Object 的实例方法。 3.Object 实例方法即定义在 Object 原型对象 Object.prototyp 上的方法，凡是定义在 Object.prototype 对象上面的属性和方法，将被所有实例对象共享。

对象扩展 Object.assign()

Object.assign()方法用于将所有可枚举属性的值从一个或多个源对象分配到目标对象，它将返回目标对象。

```
const target ={a:1,b:2};
const source={b:4,c:5};

const returnedTarget=Object.assign(target,source);

console.log(target);
console.log(returnedTarget);

//上面两个都返回{a:1,b:4,c:5}
```

```
//Object本身的方法
Object.print = function (o) {
  console.log(o);
};

//Object实例方法
Object.prototype.print = function () {
  console.log(this);
};

var obj = new Object();
obj.print(); // Object
```

(1)对象表示法：

- 点表示法：{}

```
var person={
    name:['Bob','Smith'];
    age:32;
    gender:'female';
    greeting:function(){
        alert('Hi!i\'m'+this.name);
    }
}
//改成
name:{
    first:'Bob',
    last:'Smith'
},
//调用时:
person.name.first;
```

- 括号表示法：

```
person['age']
person['name']['first']
```

4.0 构造函数

ES6之前，没有class类，所以产生了构造函数。

与普通函数区别：

1.名字：构造函数首字母建议大写

2.内容：

构造函数建议用this关键字，普通函数内部不建议用this，因为此时this指向的是window全局对象，无意会为window添加一些全局变量或函数。

构造函数默认不用return返回值，普通函数一般有return返回值。

3.调用：

构造函数使用new关键字调用

```
function createNewPerson(name){
    var obj={};
    obj.name=name;
    obj.greeting=function(){
                alert('Hi!I\'m'+this.name);
    }
}
```

可以通过调用此函数创建一个新的人。

```
var salva=createNewPerson('salva');
salva.name;
salva.greeting();
```

更好的办法：将上述代码改为：

```
function Person(name){
    this.name=name;
    this.greeting=function(){
        alert('Hi!i\'m'+this.name);
    }
}
```

注：一个构造函数通常用大写字母开头以区分构造函数和普通函数

(3)创建最终的构造函数

上代码替换成一下代码：

```
function Person(first,last,age,gender,inrterests){
    this.name={
        'first':first,
        'last':last,
    };
    this.age=age;
    this.gender=gender;
    this.interests=interests;
    this.bio=function({
        alert(this.name.first+' '+this.name.last+'is'+this.age+'years'
    };
    this.greeting=function(){
        alert('hi');

    }
```

```
                    )
}
```

加此行创建一个对象：

```
var person1=new Person('Bob','Smith',32,'male',['music','skiing'])
```

4.1 创建对象的其他方式

Object()

Object 本身是一个函数，可当作工具方法使用，将任意值转为对象。 用于保证某个值一定是对象。

- 若参数是原始类型的值，则 Object 方法将其转为对应的包装对象的实例。
- 若参数是一个对象，则总返回该对象，不用转换。

此点可写一个判断变量是否为对象的函数。

```
function isObject() {
  return value === Object(value);
}
isObject([]); //true
isObject(true); //false
```

使用create()方法

JS中有内嵌方法 `create()`,允许基于现有对象创建新的对象。

以上，对象创建三种方式：对象字面量，**new** 关键字，**Object.create()**

4.1.2 Object 构造函数

前面添加 new 命令当作构造函数使用。

```
var obj = new Object();
//equals to
var obj = {};
```

4.1.3 Object.keys() & Object.getOwnPropertyNames()

1.两者参数都是一个对象，返回一个数组，包含该对象所有属性名

2.用法：可用于计算对象属性个数：Object.keys(Objs).length;

3.Object.getOwnPropertyNames()可返回对象自身全部属性，不管它可不可以遍历。

```
var obj = new Object();
obj = {
  name: "ee",
  p: "fe",
};
Object.getOwnPropertyNames(obj); //  ['name','p']
```

4.1.4 其他 Object 静态方法

（1）对象属性模型的相关方法 Object.getOwnPropertyDescriptor()：获取某个属性的描述对象。 Object.defineProperty()：通过描述对象，定义某个属性。 Object.defineProperties()：通过描述对象，定义多个属性。

（2）控制对象状态的方法

Object.preventExtensions()：防止对象扩展。 Object.isExtensible()：判断对象是否可扩展。 Object.seal()：禁止对象配置。 Object.isSealed()：判断一个对象是否可配置。 Object.freeze()：冻结一个对象。 Object.isFrozen()：判断一个对象是否被冻结。

（3）原型链相关方法

Object.create()：该方法可以指定原型对象和属性，返回一个新的对象。 Object.getPrototypeOf()：获取对象的 Prototype 对象。

4.1.5 Object 实例方法

六种

- Object.prototype.valueOf()：返回当前对象对应的值。
- Object.prototype.toString()：返回当前对象对应的字符串形式。
- Object.prototype.toLocaleString()：返回当前对象对应的本地字符串形式。
- Object.prototype.hasOwnProperty()：判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。
- Object.prototype.isPrototypeOf()：判断当前对象是否为另一个对象的原型。
- Object.prototype.propertyIsEnumerable()：判断某个属性是否可枚举。

4.2 属性描述对象

JS 提供一个属性描述对象，描述对象属性，控制它的行为，如该属性是否可写，是否可遍历

```
{
  value: 123,
  writable: false,
  enumerable: true,//是否可遍历
  configurable: false,
  get: undefined,
  set: undefined
}
```

4.2.1 Object.getOwnPropertyDescriptor()

```
Object.getOwnPropertyDescriptor(目标对象，'字符串，对应目标对象某属性')`
```

只能用于对象自身的属性，不能用于继承的属性。

4.2.2 Object.defineProperty()，Object.defineProperties()

允许定义和修改一个属性，然后返回修改后的对象。

```
Object.defineProperty(object, propertyName, attributesObject);
```

如果一次性定义或修改多个属性，可以使用 Object.defineProperties()方法。

```
var obj = Object.defineProperties(
  {},
  {
    p1: { value: 123, enumerable: true },
    p2: { value: "abc", enumerable: true },
    p3: {
      get: function () {
        return this.p1 + this.p2;
      },
      enumerable: true,
      configurable: true,
    },
  }
);

obj.p1; // 123
obj.p2; // "abc"
obj.p3; // "123abc"
```

一旦定义了取值函数 get（或存值函数 set），就不能将 writable 属性设为 true，或者同时定义 value 属性，否则会报错。

```
var obj = {};

Object.defineProperty(obj, "p", {
  value: 123,
  get: function () {
    return 456;
  },
});
// TypeError: Invalid property.
// A property cannot both have accessors and be writable or have a value

Object.defineProperty(obj, "p", {
  writable: true,
  get: function () {
    return 456;
  },
});
// TypeError: Invalid property descriptor.
// Cannot both specify accessors and a value or writable attribute
```

4.2.3 元属性

控制属性的属性

enumerable 属性 可用来设置秘密属性

4.2.4 存取器

除了直接定义以外，属性还可以用存取器（accessor）定义。其中，存值函数称为 setter，使用属性描述对象的 set 属性；取值函数称为 getter，使用属性描述对象的 get 属性。

一旦对目标属性定义了存取器，那么存取的时候，都将执行对应的函数。利用这个功能，可以实现许多高级特性，比如定制属性的读取和赋值行为。

```
var obj = Object.defineProperty({}, "p", {
  get: function () {
    return "getter";
  },
  set: function (value) {
    console.log("setter: " + value);
  },
});

obj.p; // "getter"
obj.p = 123; // "setter: 123"

// 写法二
var obj = {
  get p() {
    return "getter";
  },
  set p(value) {
    console.log("setter: " + value);
  },
};
```

存取器往往用于属性的值依赖对象内部数据的场合。

4.2.5 对象的拷贝

Object.defineProperty 方法拷贝属性

```
var extend=function (to,from){
  for(var property in from){
    if(!from.hasOwnProperty(property)
    Object.defineProperty(
      to,property,
      Object.getOwnPropertyDescription(from,property)
    );
  }
}

return to;

extend({},{get a(){return 1}})
```

4.2.6 控制对象状态

`Object.preventExtensions`使一个对象无法再增加新的特性。 `Object.seal()`使对象无法增加新特性，也无法删除旧特性。

`Object.freeze()`使对象无法添加，删除，修改属性，即使变量实际上变成一个常量。

局限：

- 可以通过改变原型对象，为对象增添新的属性。

  ```
  var obj = new Object();
  Object.preventExtensions(obj);

  var proto = Object.getPrototypeOf(obj);
  Object.preventExtensions(proto);
  ```

- 如果属性值是对象，则上述方法只能冻结属性指向的对象，而不能冻结对象本身的内容。

```
var obj=
{
  foo:1;
  bar:['fe','mn']
}
Object.freeze(obj);
obj.bar.push('cn');
obj.bar;//['fe','mn','cn']
```

4.3 Array 对象

4.3.1 构造函数

Array 是 JS 的原生对象，同时也是一个构造函数，可用来生成新的数组。

```
var arr = new Array(2);
//equals to var arr=Array(2)
arr.length; //2
arr; //[empty*2]
```

4.3.2 shift(),unshift()

`shift`方法用于删除数组的第一个元素，并返回该元素。 `shift`方法可以遍历并清空一个数组。

```
var list = [1, 2, 3, 4, 5];
var item;

while ((item = list.shift())) {
  console.log(item);
}
list; //[]
```

`push`和 `shift`结合可构成“先进先出”的队列结构。（queue)

unshift 方法用于在数组第一个位置添加元素，并返回添加新元素后的数组长度。

```
var arr = ["x", "y"];
arr.unshift("e", "ef"); //4
arr; //['x','y','e','ef']
```

4.3.3 join()

```
var a = [1, 2, 3, 4];
a.join("   "); //'1   2   3   4'
```

默认会用逗号分隔

4.3.4 concat()

用于多个数组的合并。 将新数组成员添加到原数组成员的后部，返回一个新数组，原数组不变。

```
["hello"].concat(["world"], ["!"]);
//['hello','world','!']
```

如果成员包括对象,concat 方法返回当前数组的一个浅拷贝。（浅拷贝指新数组拷贝的是对象的引用）

```
var obj = { a: 1 };
var oldArray = [obj];

var newArray = oldArray.concat();
obj.a = 2;
newArray[0].a; //2
newArray[0]; //{a:2}
```

4.3.5 reverse()

用于颠倒排列数组元素，返回改变后的数组。

4.3.6 slice()

用于提取目标数组的一部分，返回一个新数组，原数组不变。 slice(start,end-1);

4.3.7 splice()

用于删除原数组的一部分成员，并在删除位置添加新的数组成员，返回被删除元素。

```
arr.splice(start,count,element1,element2...)

//eg
var a=['a','b','c','d','e','f'];
a.splice(4,2,'t','v');//['e','f']
a//['a','b','c','d','t','v']
```

4.3.8 sort()

对数组成员排序，默认按字典顺序。

```
[10111,1101,111].sort(function(a,b){
  return a-b;
}
//
```

4.3.9 map()

将数组的所有成员依次传递进参数函数，然后把每一次执行结果组成一个新数组返回。

```
var numbers = [1, 3, 4];
numbers.map(function (n) {
  return n + 1;
});
//[2,4,5]
numbers; //[1,3,4]
```

map()方法接受一个函数作为参数，该函数调用时，map 方法向他传入三个参数：当前成员，当前位置，数组本身。

```
[1, 2, 3].map(function (elem, index, arr) {
  return elem * index;
});
//[0,2,6]
```

4.3.10 forEach()

forEach()方法也是对数组所有成员依次执行参数函数，但 forEach 方法不返回值，只用来操作数据。 也是三个参数：当前成员，当前位置，数组本身。

forEach 方法也可接受第二个参数，绑定参数函数的 this 变量。

```
var out = [];

[1, 3, 5].forEach(function (elem) {
  this.push(elem * elem);
}, out);

out; //[1,9,25]
```

上例空数组 out 是 forEach 函数第二个参数，结果回调函数内部的 this 关键字即指向 out。

4.3.11 filter()

filter 方法用于过滤组成员，满足条件的组成员形成一个新数组返回。

4.3.12 some() & every()

类似“断言”，返回一个布尔值，表示判断数组成员是否符合某种条件。 some方法只要一个成员返回值为 true,则整个 some方法返回值就是 true,否则返回 false.

```
var arr = [1, 2, 3, 4, 5];
arr.some(function (elem, index, arr) {
  return elem > 3;
});
//true;
```

every所有才返回 true.

reduce() & reduceRight()

依次处理数组每个成员，最终累计为一个值。 reduce()从左往右处理，reduceRight()从右往左处理。

```
[1, 2, 3, 4, 5].reduce(function (a, b) {
  console.log(a, b);
  return a + b;
});
// 1 2
// 3 3
// 6 4
// 10 5
// 最终结果: 15
```

4.3.13 indexOf() lastIndexOf()

1.indexOf 返回给定元素在数组中第一次出现位置，没有出现则返回-1。 2.接受第二个参数表示搜索的开始位置。

```
var a = [1, 2, 3];
a.indexOf(1, 1);
//-1
```

3.lastIndexOf()放回给定元素在数组中最后一次出现的位置，没有出现则返回-1。

4.3.14 链式使用

```
var users = [
  { name: "Jack", email: "325@qq.com" },
  { name: "Tom", email: "2243@qq.com" },
];
users
  .map(function (user) {
    return user.email;
  })
  .filter(function (email) {
    return /^2/.test(email);
  })
  .forEach(console.log);
//2243@qq.com
```

4.4 包装对象

“包装对象”即分别与数值，字符串，布尔值相对应的 Number，string，Boolean 三个原生对象，此三原生对象可将原始类型的值变成包装对象。

目的：1.使得 JS 对象涵盖所有的值。2.使得原始类型的值可以方便调用某种方法。

4.4.2 实例方法

包装对象可使用 Object 对象提供的原生方法。

- valueOf()方法
- toString()方法。

4.4.3 原始类型与实例对象自动转换

原始类型的值可自动当作包装对象使用，即调用各种包装对象的属性和方法，如字符串可调用 length 属性，返回字符串长度。

```
"abc".length; //3
```

4.4.4 自定义对象

三种包装对象除提供很多原生实力方法，还可在原型上添加自定义方法和属性，供原始类型的值直接调用。

如可新增一个 double 方法，使字符串和数字翻倍。

```
String.prototype.double = function () {
  return this.valueOf() * 2;
};

"abc".double();
//abcabc

Number.prototype.double = (function () {
  return this.valueOf() * 2;
})(352).double();
//352352
```

4.4.5 Boolean 对象

4.5 Number 对象

Number 对象是数值对应的包装对象。

```
var n = new Number(1);
typeof n; //'object'
```

4.5.1 Number 对象属性

- Number.POSITIVE_INFINITY：正的无限，指向 Infinity。
- Number.NEGATIVE_INFINITY：负的无限，指向-Infinity。
- Number.NaN：表示非数值，指向 NaN。
- Number.MIN_VALUE：表示最小的正数（即最接近 0 的正数，在 64 位浮点数体系中为 5e-324），相应的，最接近 0 的负数为-Number.MIN_VALUE。
- Number.MAX_SAFE_INTEGER：表示能够精确表示的最大整数，即 9007199254740991。
- Number.MIN_SAFE_INTEGER：表示能够精确表示的最小整数，即-9007199254740991。

4.5.2 实例方法

Number 对象有 4 个实例方法。

(1)Number.prototype.toString() 将一个数值转换为字符串形式。

```
(10)
  .toString()
  (
    //'10'

    10
  )
  .toString(2)(
    //1010
    10
```

```
  )
  .toString(16); //'a'
```

(2)Number.prototype.toFixed() 将一个数转为指定位数的小数，然后返回这个小数对应的字符串。

```
(10).prototype.toFixed(2); //10.00
```

(3)Number.prototype.toExponential() 用于将一个数转为科学计数法形式。

(4)Number.prototype.toPrecision() 用于将一个数转为指定位数有效数字。

4.5.3 自定义方法

```
Number.prototype.add = (function (x) {
  return this + x;
})(3).add(2); //5
```

4.6 String 对象

4.6.1 静态方法

String.fromCharCode() 参数是一个或多个数值，代表 Unicode 码点，返回值是这些码点组成的字符串。

4.6.2 实例属性

String.prototype.length

4.6.3 实例方法

(1)String.prototype.charAt() 返回指定位置的字符，参数是从 0 到开始编号的位置。

```
```

```
var s = new String("abc");

s.charAt(1); //'b'
s.charAt(s.length - 1); //'c'
```

(2)String.prototype.fromCharCodeAt()

(3)String.prototype.concat()

(4)String.prototype.slice()

(5)String.prototype.substring()

(6)String.prototype.substr() 从原字符串取出子字符串并返回，不改变原字符串

(7)String.prototype.indexOf()

(8)String.prototype.trim() 用于去除字符串两端的空格，返回一个新字符串。

(9)String.toLowerCase() 全转换为小写

(10)String.match() 用于确定源字符串是否匹配某子字符串。 有返回一个数组，成员为匹配的第一个字符串，没有返回 NULL.

```
"cat", "bat", "sat", "fat".match("at"); //['at']
```

还可用于使用正则表达式作为参数。

(11)String.protorype.search() 等同于 match()

(12)String.prototype.split() 按照给定规则分隔字符串。

(13)String.prototype.localeCompare() 用于比较两个字符。

4.7 Math 对象

4.7.1 Math 静态属性

Math.E：常数 e。 Math.LN2：2 的自然对数。 Math.LN10：10 的自然对数。
Math.LOG2E：以 2 为底的 e 的对数。 Math.LOG10E：以 10 为底的 e 的对数。

Math.PI：常数 π。 Math.SQRT1_2：0.5 的平方根。 Math.SQRT2：2 的平方根。

4.7.2 Math 静态方法

。。。

4.7.3 Math

4.8 Date 对象

4.8.1 Date 普通函数

Date()用于获取当前时间。

4.8.2 Date 构造函数用法

使用 new 命令，返回一个 Date 对象的实例。

```
var today = new Date();
```

4.8.3 DATe 静态方法

(1)Date.now()

(2)Date.parse()

(3)Date.UTC()

4.8.4 DATe 实例方法

- to 类：从 Date 对象返回一个子 u 穿，表示指定的时间。
- get 类：获取 Date 对象的日期和时间。
- set 类：设置 Date 对象的日期和时间。

4.9 RegExp 对象

regular expression 是一种表达文本模式（字符串匹配结构）的方法。

4.9.1 RegExp 创建方法

- 使用字面量,以斜杠表示开始和结束。

```
var regex = /xyz/;
```

- 使用 RegExp构造函数。

```
var regex = new Regexp("xyz");
```

区别：第一种在引擎编译代码时，就会新建正则表达式，第二种方法在运行时新建正则表达式。 实际工作用前者。

4.9.2 RegExp 实例属性

> RegExp.prototype.ignoreCase:返回一个布尔值，表示是否设置了 i 修饰符。>

> RegExp.prototype.global:返回一个布尔值，表示是否设置了 g 修饰符。>

> RegExp.prototype.multiline:返回一个布尔值，表示是否设置了 m 修饰符。>

4.9.3 RegExp 实例方法

(1)RegExp.prototype.test() 返回一个布尔值，表示当前模式是否能匹配参数字符串。

```
/car/.test('car and tree')//true;

var r=/x/g;
var s='_x_x';

r.lastIndex//0
r.test(s)//true
```

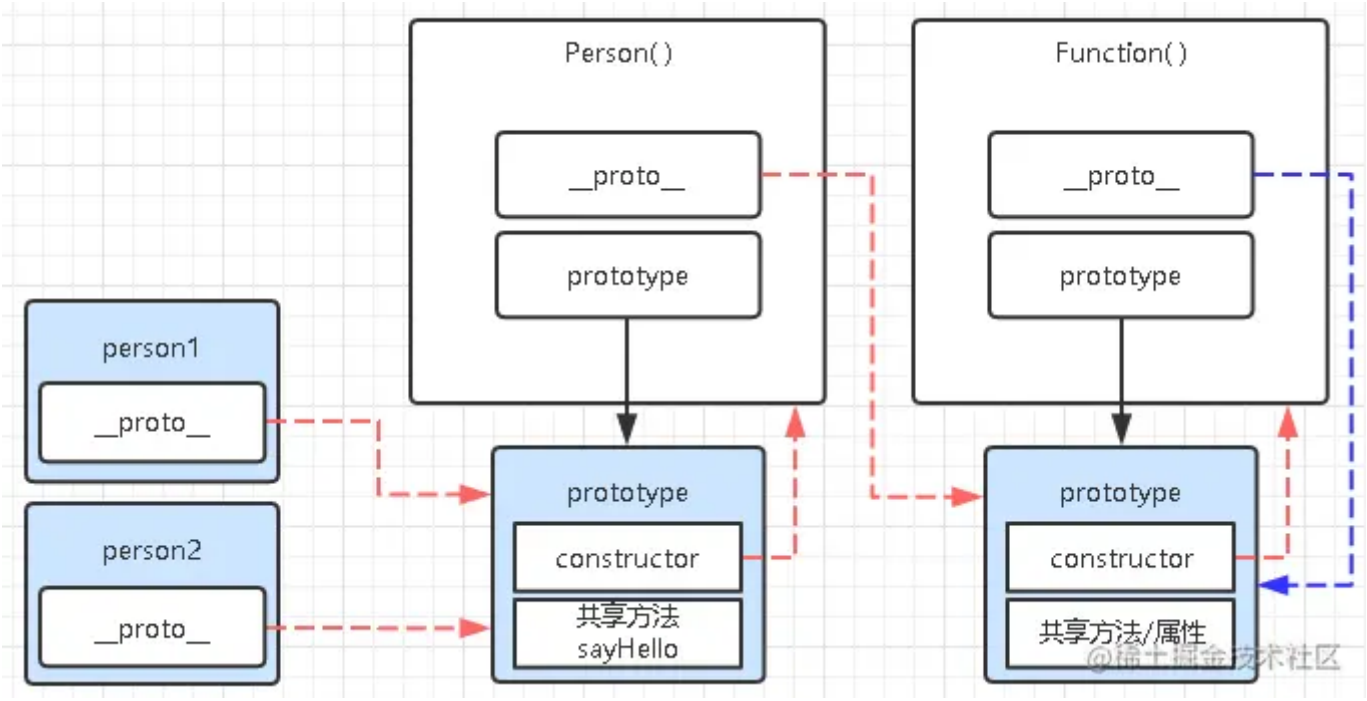
```
r.lastIndex//2
r.test(s)//true

r.lastIndex//4
r.test(s)//false
```

4.10 JSON 对象

JSON可作为一个对象或字符串存在，前者用于解读JSON中的数据，后者用于通过网络传输JSON数据。

5 对象原型



5.1 JS中的继承

**原型式继承-prototype inheritance

class关键字的类

```
class Range{
    constructor(from,to){
```

```
        //保存新范围对象的起点和重点（状态）
        //这些属性不是继承的，而是当前对象独有的
        this.from=from ;
        this.to=to;
    }

    //如果x在范围内则返回true,否则返回false;
    includes(x){return this.from<=x&& x<=this.to;}

    //此生成器函数让此类的实例可迭代
    //下面只使用数值范围:
    *[Symbol.iterator](){
        for(let x=Math.ceil(this.from);x<this.to;x++) yield x;
    }
}

//下面生成Range类的实例
let r=new Range(2,3);
r.includes(2);//true
[...r]        //[2,3] 通过迭代器转换为数组
```

static

类通过static关键字定义静态方法。不能在类的实例上调用静态方法，而应通过类本身调用。

调用静态方法

静态方法调用同一类中其他静态方法，可用this关键字。

```
class StaticMethodCall{
    static staticMethod(){
                return 'fefefe';
    };
    static anotherStaticMethod(){
        return this.staticMethod()+' from another static method';
    }
}
StaticMethodCall.anotherStaticMethod();
//'fefefe from another static'
```

类私有域

类属性默认公有，增加哈希前缀 #定义私有字段；

私有字段包括私有实例字段和私有静态字段。

私有实例字段

从作用域外引用#名称，内部在为声明情况下引用私有字段，或尝试使用delete 移除声明字段都会抛出语法错误。

```
class PrivateField{
    #privateField;
    //用# 声明私有字段
    constructor(){
        this.#privateField=43;
        delete this.#privateField;//语法错误
    }
}

//类似公有字段，私有字段在构造(constructor)基类或调用子类的super()方法时被添加到类实例中
class Subclass extends PrivateField{
    #subPrivateField;
    constructor(){
        this.#subPrivateField=33;
    }
}

new SubClass();
//Subclass{#privateField:43,#subPrivateField:33}
```

私有静态字段

在解析类结构时被添加到类的构造方法中（constructor)，且静态变量只能被静态方法调用的限制仍成立。

私有实例方法

访问权限

类内部一般可相互访问；但类外部通过类的对象，只能访问public 属性的成员，不能访问protected ， private 属性的成员。

C++中：

访问	该类中的函数	子类函数	友元函数	该类的对象
public	可	可	可	可
protected	可	可	可	不可

访问	该类中的函数	子类函数	友元函数	该类的对象
private	可	不可	可	不可

通过extends 和 super 创建子类

about super()

1. 如果使用extends 关键字定义了一个类，则此类的构造函数必须使用super()调用父类构造函数。
2. 如果没有在子类中定义构造函数，解释器会自动创建一个。此隐式定义的构造函数会取得传给它的值，然后把这些值传给super().
3. 通过super()调用父类构造器前，不能在构造函数中使用this关键字。

when to use delegation instead of inheritation ?

当想“复制”/公开基类的API时，使用继承；

当只想复制功能时，使用委托。

抽象类

使用abstract 关键字修饰方法，此方法就成了抽象方法。

抽象方法只包含一个方法名，而没有方法体。

1. 抽象类：将抽象的部分和相似的部分抽取到一个父类当中（共性内容向上抽取）。
2. 抽象类和普通类的区别：抽象类可以定义抽象方法。
3. 当将共性的行为（方法）抽取到父类后，该行为在父类描述不清了。但此行为还是子类（强制重写）必须要做的行为。就可定义为抽象类；

exp:

```
动物类:
eat();//将共性的eat方法抽取到父类后，发现该行为描述不清了。
猫类:
eat(){吃鱼};
狗类:
eat(){吃肉};
```

6 异步Javascript

计算密集型

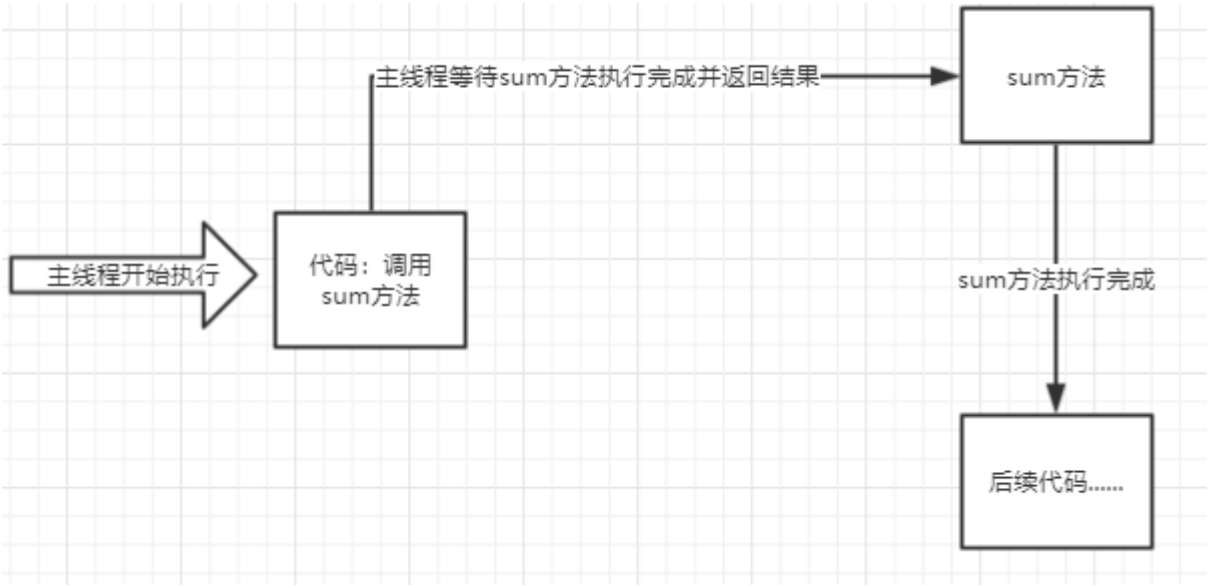
computationally intensive)

程序大部分在做计算，逻辑判断，循环导致cpu占用率很高的情况，

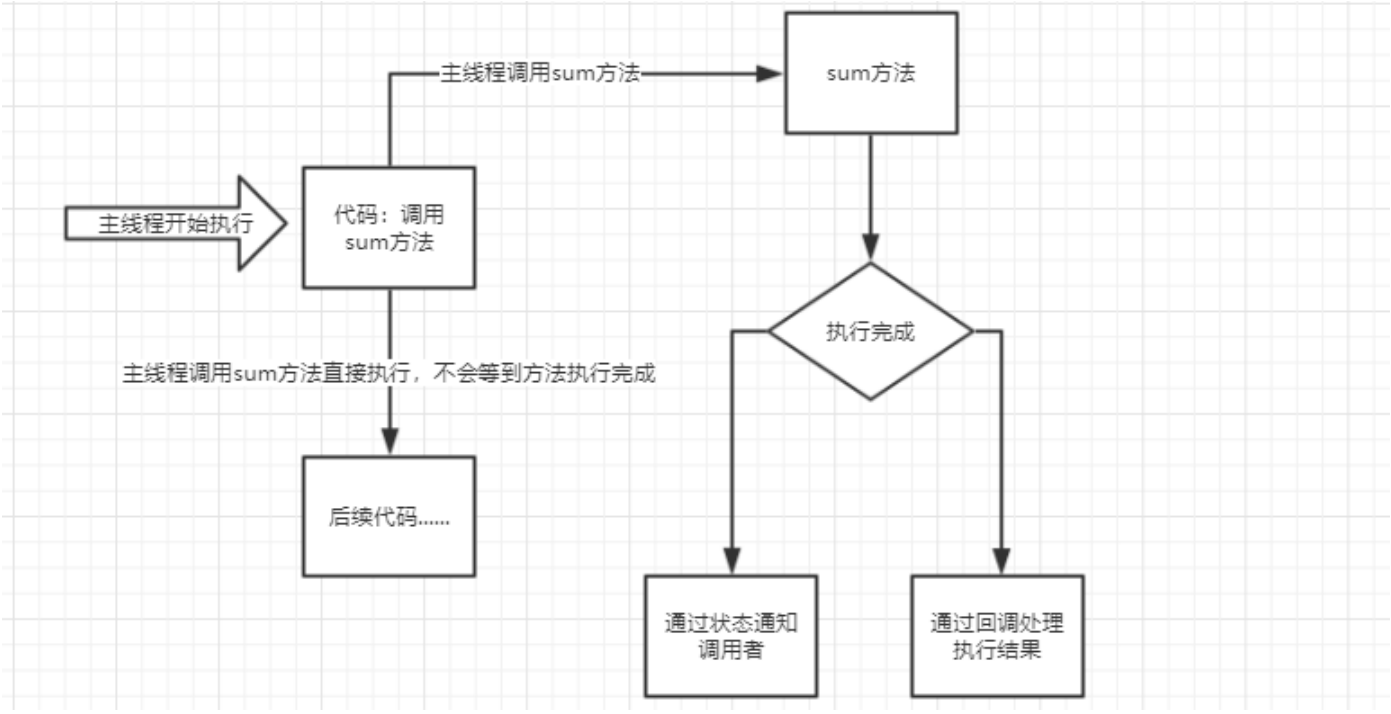
I/O密集型：

频繁网络传输，读取硬盘，及其他io设备称之为io密集型

同步和异步



同步执行当调用方法执行完成后并返回结果，才执行后续代码



异步编程允许在执行一个长时间任务时，不等待，而是继续执行之后的代码。

回调

回调

回调

事件

javascript几乎全是事件驱动的。

事件驱动的javascript程序在特定上下文中为特定类型的事件注册回调函数，而浏览器在指定的事件发生时调用这些函数。这些回调函数叫做事件处理程序或事件监听器，通过addEventListener()注册：

事件流

event

是网页元素接受事件的顺序。

事件捕获阶段

处于目标阶段

事件冒泡阶段

网络事件

fetch()

Node中的回调事件

Node.js服务器端javascript环境底层就是异步的，定义了很多使用回调和事件的API.

学习前，

并发，并行，异步，同步

并发 concurrency

代表计算机能执行多项任务。

并行 parallelism

对于多核处理器，可以在不同核心上并行执行任务，而不采用分配时间片的情况。

javascript本身没有多线程概念，但通过**函数回调**机制，仍能做到单线程的“并发”。

如通过fetch()函数同时访问多个网络资源。

在Promise.all()接受一个Promise的数组作为输入，返回一个Promise.

```
//先定义一个URL数组
const urls=[/*零或多个url*/];
//将它转为一个Promise对象的数组
promise=urls.map(url=>fetch(url).then(r.text()));
//现在用一个Promise来并行数组中的所有Promise
Promise.all(promise)
.then(bodies=>{/*处理得到的字符串数组*/})
.catch(e=>console.log(e));
```

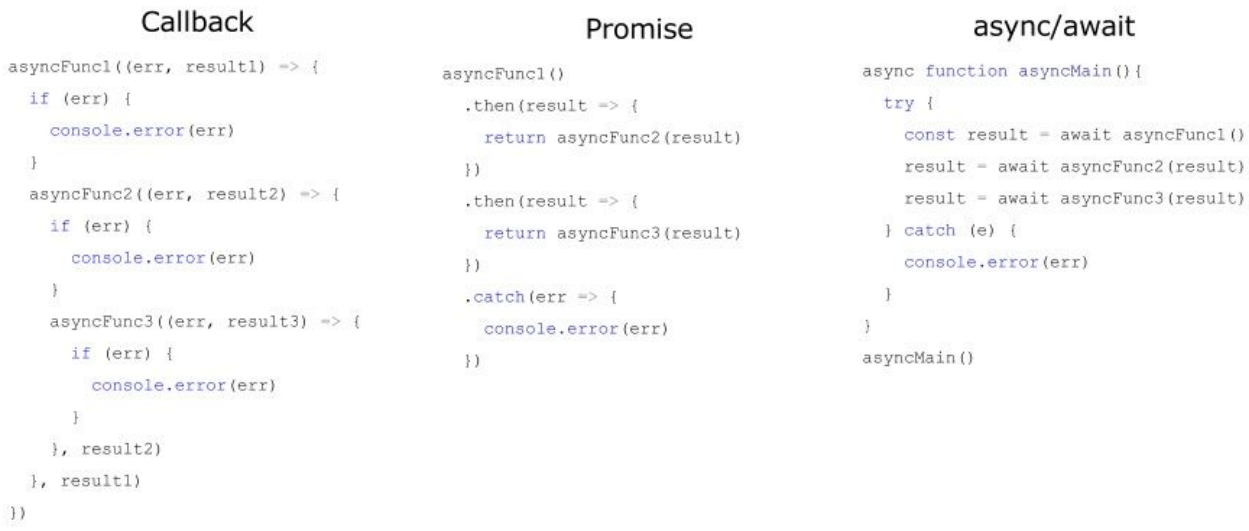
与阻塞和非阻塞

阻塞和非阻塞

强调程序在等待调用结果（消息，返回值）时的状态，阻塞调用是指调用结果返回前，当前线程会被挂起。

同步和异步

强调消息通信机制（synchronous communication/asynchronous communication)。



同步**Javascript**演示

```
const btn=document.querySelector('button');
btn.addEventListener('click',()=>{
    alert('you click me');
    let pElem=document.createElement('p');
    pElem.textContent='This is a newly-added paragraph.';
    document.body.appendChild(pElem);//把此段落放入网页
});
```

以上代码一行一行顺序执行：

1.先取得在DOM里面的 <button>引用

2.点击按钮，添加一个click事件监听器：

- alert()消息出现。
- 一旦alert结束，创建一个 <p>元素。
- 给他的文本内容赋值。
- 最后，把这个段落放入网页。

异步Javascript

两种：callbacks和promise

callbacks对比promise

1. promise可使用多个then()操作将多个异步操作链接在一起，并将其中一个操作的结果作为输入传递到下一个操作。这对于回调会产生”回调地狱“。
2. promise总是严格按照它们放置在事件队列中的顺序调用。
3. 错误处理好——所有错误都有快末尾的一个.then()块处理，而非在”金字塔“每一层单独处理。

callbacks

```
const btn=document.querySelector('button');
btn.addEventListener('click',()=>{
    alert('you click me');

    let pElem=document.createElement('p');
    pElem.textContent='This is a newly-added paragraph.';
    document.body.appendChild(pElem);//把此段落放入网页
});
```

第一个参数是侦听的事件类型，第二个是事件发生时调用的回调函数。

6.1 Promise

Promise是一个对象，表示异步操作的结果

```
console.log('Starting');
let image;

fetch('coffee.jpg').then((response)=>{
    console.log('It works')
```

```
    return response.blob();
}).then((myBlob)=>{
    let objectURL=URL.createObjectURL(myBlob);
    image=document.createElement('img');
    image.src=objectURL;
    document.body.appendChild(image);
}).catch((error) => {
  console.log('There has been a problem with your fetch operation: ' +
error.message);
});

console.log ('All done!');
```

1.console.log(输出Starting),创建image变量

2.执行fetch()块，但因为fetch()是异步执行的，没有阻塞，所以在promise 相关代码之后继续执行，到达最后的console.log()语句(All done)将其输出到控制台。

3.当fetch()块完成运行返回结果给.then() ，才能看到第二个console.log()消息(it worked)

消息出现顺序
1.starting
2.all done
3.it worked