Js的六种主要类型:string number boolen null undefined object

null是js的bug 但是一直沿用至今，对null执行typeof()会返回object

Js的内置对象有:String Number Array Function Object Boolean Date RegExp Error

typeof()可以判断对象类型

Instanceof判断前者是否为后者的实例化

对内置对象的判断

console.log(typeof(String));

console.log(typeof(Number));

console.log(typeof(Boolean));

console.log(typeof(Object));

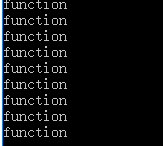
console.log(typeof(Function));

console.log(typeof(Array));

console.log(typeof(Date));

console.log(typeof(RegExp));

console.log(typeof(Error));



对js对象的判断

console.log(typeof(string));

console.log(typeof(number));

console.log(typeof(boolean));

console.log(typeof(null));

console.log(typeof(undefined));

console.log(typeof(object));



内置对象是对象的子类型

欺骗语法

function foo(str, a) {

eval( str ); // 欺骗！

console.log( a, b );

}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3

JavaScript 中的 eval(..) 函数可以接受一个字符串为参数，并将其中的内容视为好像在书

写时就存在于程序中这个位置的代码。

不加eval()函数的话输出就是1 和2

# JavaScript严格模式与非严格模式的区别

严格模式通过****抛出错误****来消除一些原有的****静默错误****。  
其次，严格模式修复了一些导致JavaScript引擎难以执行优化的缺陷：有时候，相同的代码，严格模式可以比非严格模式下****运行的更快****。  
第三，严格模式****禁用了****在ECMAScript的未来版本中可能会定义的一些语法。

使用严格模式

* 函数中使用

funtion () {

'use strict'; *//或者是"use strict";*}

* 1
* 2
* 3
* 整个脚本中使用：在这个JavaScript文件开头写'use strict';(或者是"use strict";)

with可以封装某个对象，减少某个对象的调用，var a = {};

a.aa = 1;

a.bb = 2;

a.cc = 3;

with(a) {

console.log(aa);

console.log(bb);

console.log(cc);

}

输出1 2 3

如果代码中大量使用 eval(..) 或 with，那么运行起来一定会变得非常慢。

let可以使变量隐藏在特定作用域内

for(let i=0;i<10;i++)

{

console.log(i);

}

console.log(i);

ReferenceError: i is not defined

for(var i=0;i<10;i++)

{

console.log(i);

}

console.log(i);

console.log(i);console.log(i);

改成var后，后面控制台就还能打印出i

const b=2;

b=3;

TypeError: Assignment to constant variable.

const声明的变量在块作用域内被改变值，会报这个错误

var foo = true;

if (foo) {

var a = 2;

const b = 3; // 包含在 if 中的块作用域常量

a = 3; // 正常 !

b = 4; // 错误 !

}

console.log( a ); // 3

console.log( b ); // ReferenceError!

闭包的概念：

function foo(){

var a =2;

function bar(){

console.log(a);

}

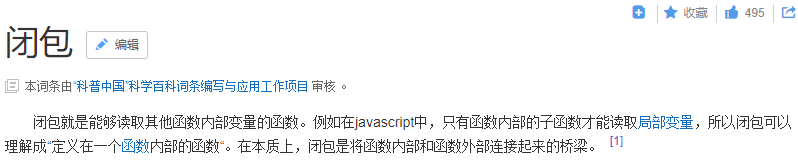
return bar;

}

var baz = foo();

baz();

原本foo()函数的内存使用完后会被回收，但是因为闭包可以阻止回收的发生



this的特点:

function foo() {

var a = 2;

console.log( this.a );

}

foo();

控制台显示Undefined，因为这里this.a是指向全局变量的

this会根据调用位置不同，而发生规则的变化

对象

var str="qwe qweret";

console.log(str.charAt(4));

控制台打印q

charAt()会返回字符串在该位置的字符，空格会自动忽略

# try{}finally{}

finally 中抛出异常（无论是有意还是无意），函数就会在此处终止。如果此前 try 中

已经有 return 设置了返回值，则该值会被丢弃

原型

var myObj = {

a:2,

b:234

};

console.log(myObj.a);

console.log(myObj.b);

如果对象中a和b，就调用get操作打印，如果无法再对象本身找到a和b，就要调用对象的prototype链，如下

var anotherObj={

a:2

};

var myObj =Object.create(anotherObj);

Console.log(myObj.a);

用过prototype链，两个对象建立了联系（Object.create()函数可以关联其他对象）

所有的prototype链最终都指向Object.prototype

在nodejs中，异步的IO是通过回调函数来实现的，

function readFileCallBack(err, data) {

if (err) {

console.error(err);

} else {

console.log(data);

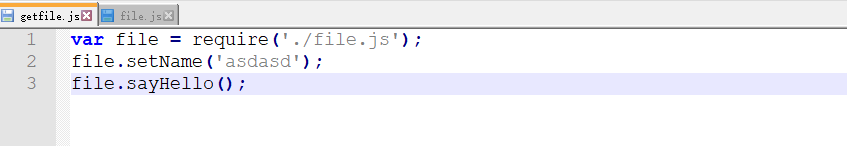
}

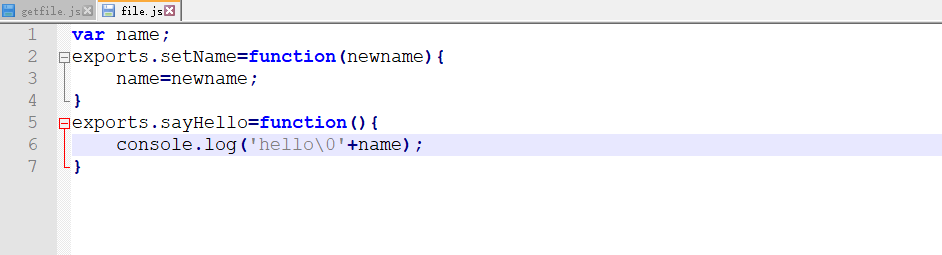
}

var fs = require('fs');

fs.readFile('file.txt', 'utf-8', readFileCallBack);

console.log('end.');





文件就是一个模块，exports可以让接口公有

require引用的对象是单次加载的，不会重复加载

var file1 = require('./file.js');

file1.setName('1111');

var file2 = require('./file.js');

file2.setName('22222');

file1.sayHello();

最后打印是2222

file2把file1覆盖了