目录

[Js数据类型 1](#_Toc43131028)

[js交换两个变量的值 1](#_Toc43131029)

[instanceof 和 typeof 的实现原理 1](#_Toc43131030)

[ type of: 1](#_Toc43131031)

[ Instanceof: 1](#_Toc43131032)

[闭包 2](#_Toc43131033)

[总结原生JS实现JSONP的步骤 5](#_Toc43131034)

[JS中继承实现的几种方式 6](#_Toc43131035)

[JavaScript异步编程史：回调函数到Promise到Async/Await 6](#_Toc43131036)

[函数调用 6](#_Toc43131037)

[setTimeout和 setInterval 7](#_Toc43131038)

[Generator函数的Trunk输出的是什么 7](#_Toc43131039)

[页面优化之滚动优化 7](#_Toc43131040)

# Js数据类型

6种基本：string number Boolean undefined null symbol

Object类型：array function date regExp正则

Typeof判断基本类型: 1.对于基本类型，除 null 以外，均可以返回正确的结果。  
2.对于引用类型，除 function 以外，一律返回 object 类型。  
3.对于 null ，返回 object 类型。

# js交换两个变量的值

https://blog.csdn.net/weixin\_44827421/article/details/93717370

# instanceof 和 typeof 的实现原理

## type of:

typeof null == 'object'

原理是这样的， 不同的对象在底层都表示为二进制， 在 JavaScript 中二进制前三位都为 0 的话会被判断为 object 类型， null 的二进制表示是全 0， 自然前三位也是 0， 所以执行 typeof 时会返回“object”。

其实这个是一个历史遗留的bug，在 javascript 的最初版本中，使用的 32 位系统，为了性能考虑使用低位存储了变量的类型信息：

* 000：对象
* 1：整数
* 010：浮点数
* 100：字符串
* 110：布尔

有 2 个值比较特殊：

* undefined：用 - （−2^30）表示。
* null：对应机器码的 NULL 指针，一般是全零。

所以当你使用typeof null时返回就是object了

## Instanceof:

<https://juejin.im/post/5b0b9b9051882515773ae714>

**instanceof 主要的作用就是判断一个实例是否属于某种类型**

**也可以判断一个实例是否是其父类型或者祖先类型的实例。**

**instanceof 主要的实现原理就是只要右边变量的 prototype 在左边变量的原型链上。因此，instanceof 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 prototype，如果查找失败，返回 false，左边变量并非是右边变量的实例。**

# 闭包

<https://segmentfault.com/a/1190000021725949>

闭包指 **有权访问另一个函数作用域中的变量的函数** 。

闭包：一般情况就是在一个函数中包含另一个函数。  
函数作用域是独立的、封闭的，外部的执行环境是访问不了的，但是闭包具有这个能力和权限。  
那闭包是怎样的一个表现形式呢？  
第一，闭包是一个函数，而且存在于另一个函数当中  
第二，闭包可以访问到父级函数的变量，且该变量不会销毁

作用1：隐藏变量，避免全局污染  
作用2：可以读取函数内部的变量  
同时闭包使用不当，优点就变成了缺点：  
缺点1：导致变量不会被垃圾回收机制回收，造成内存消耗  
缺点2：不恰当的使用闭包可能会造成内存泄漏的问题  
这里简单说一下，为什么使用闭包时变量不会被垃圾回收机制收销毁呢，这里需要了解一下JS垃圾回收机制；  
JS规定在一个函数作用域内，程序执行完以后变量就会被销毁，这样可节省内存；使用闭包时，按照作用域链的特点，闭包（函数）外面的变量不会被销毁，因为函数会一直被调用，所以一直存在，如果闭包使用过多会造成内存销毁。

举例;

Founction age(){

Var age=1;

Var a(){

age++;

console.log(age);

}

Return a;

}

直接调用age，每次都是1

而 var personAge=age(); 相当于a(),绕过初始值

personAge();

**严格模式的改变**

#### ****1、普通变量****

//严格模式下对不可写属性赋值，将报错。

var demo2 = {};

Object.defineProperty(demo2, "x", { value: 42, writable: false });

demo2.x = 9; //报错

//严格模式下对只读属性赋值，将报错。

var demo3 = { get x() { return 17; } };

demo3.x = 5; //报错

//严格模式下对禁止扩展的对象添加新属性，将报错。

var demo4 = {};

Object.preventExtensions(demo4);

demo4.newProp = "ohai"; //报错

//严格模式下删除一个不可删除的属性，将报错。

delete Object.prototype; //报错

//严格模式下无法删除变量。只有configurable设置为true的对象属性，才能被删除。

var x;

delete x; // 报错

var demo5 = Object.create(null, {'x': {

value: 1,

configurable: true

}});

delete demo5.x; // 删除成功

#### ****3、禁止this关键字指向全局对象****

严格模式下，全局作用域的函数中的this不再指向全局而是undefined。  
如果使用构造函数时，如果忘了加new，this不再指向全局对象，而是undefined报错。

function demo7\_1(){

console.log(this);

}

function demo7\_2(){

function demo7\_3(){

console.log(this);

}

demo7\_3();

}

demo7\_1(); //undefined

demo7\_2(); //undefined

function demo7\_4(){

&emsp; this.a = 1;

};

demo7\_4();// 报错，使用构造函数时，如果忘了加new，this不再指向全局对象，而是undefined.a。

复制代码

#### ****4、静态绑定****

* 禁止使用with语句
* eval语句本身就是一个作用域，它所生成的变量只能用于eval内部。

//严格模式下禁用with

var demo8 = 1;

with (o){ // 报错

&emsp;&emsp;demo8 = 2;

}

//正常模式下，eval语句的作用域，取决于它处于全局作用域，还是处于函数作用域。

//严格模式下，eval语句本身就是一个作用域，不再能够生成全局变量了，它所生成的变量只能用于eval内部。

//严格模式下，eval语句内传入的字符串也是按照严格模式执行。

function demo9() {

var x = 2;

eval("var y = 1; console.log(y); "); //1

eval("var x = 12");

console.log(x); //2

console.log(y); //报错：y is not defined

}

demo9();

#### ****5、arguments对象的限制****

//不允许对arguments赋值

arguments++; //报错

//arguments不再追踪参数的变化

//在非严格模式中,修改arguments对象中某个索引属性的值,和这个属性对应的形参变量的值也会同时变化,反之亦然。

//在严格模式中arguments 对象会以形参变量的拷贝的形式被创建和初始化，因此arguments对象的改变不会影响形参。

function demo10\_1(a) {

a = 2;

return [a, arguments[0]];

}

console.log(demo10\_1(1)); // 正常模式为[2,2]

function demo10\_2(a) {

"use strict"

a = 2;

return [a, arguments[0]];

}

console.log(demo10\_2(1)); // 严格模式为[2,1]

//禁止使用arguments.callee

var demo11 = function() { return arguments.callee; };

demo11(); // 报错

复制代码

#### ****6、禁止在函数内部遍历调用栈****

function demo12(){

demo12.caller; // 报错

demo12.arguments; // 报错

}

demo12();

#### ****7、保留字****

使用未来保留字(也许会在ECMAScript 6中使用):implements, interface, let, package, private, protected, public, static,和yield作为变量名或函数名会报错。

# 总结原生JS实现JSONP的步骤

<https://www.jianshu.com/p/e1e2920dac95>

**客户端**

1. 定义获取数据后调用的回调函数
2. 动态生成对服务端JS进行引用的代码
   * 设置url为提供jsonp服务的url地址，并在该url中设置相关callback参数
   * 创建script标签，并设置其src属性
   * 把script标签加入head，此时调用开始。

**服务端**

将客户端发送的callback参数作为函数名来包裹住JSON数据，返回数据至客户端。

|  |
| --- |
|  |

# 函数调用

函数调用的方法一共有 4 种

1. 作为一个函数调用

a(); 这样一个最简单的函数，不属于任何一个对象，就是一个函数，这样的情况this在 JavaScript 的在浏览器中的非严格模式默认是属于全局对象 window 的，在严格模式，就是 undefined。

1. 函数作为方法调用

**this 永远指向最后调用它的那个对象**”

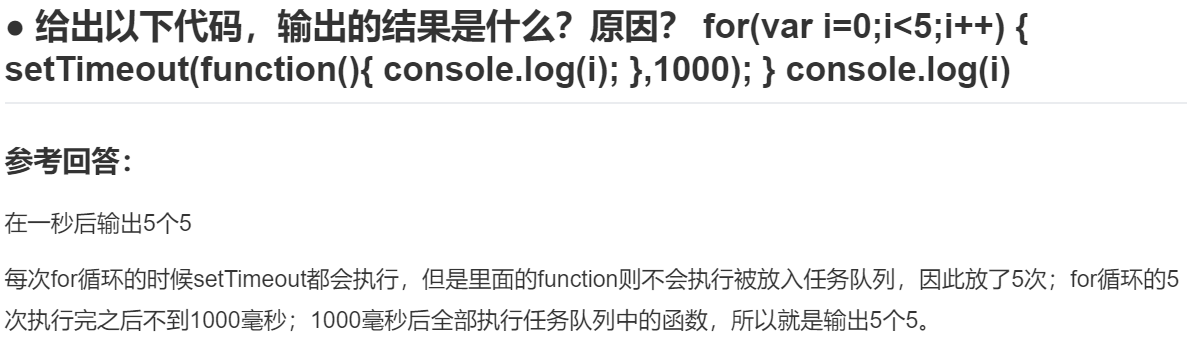
1. 使用构造函数调用函数

如果函数调用前使用了 new 关键字, 则是调用了构造函数。  
这看起来就像创建了新的函数，但实际上 JavaScript 函数是重新创建的对象

1. 作为函数方法调用函数（call、apply）

# setTimeout和 setInterval

用setTimeout实现计时器效果（setInterval）



# Generator函数的Trunk输出的是什么

<http://www.ruanyifeng.com/blog/2015/05/thunk.html>

# 页面优化之滚动优化

<https://www.cnblogs.com/coco1s/p/5499469.html>

防抖和节流