目录

[函数作用域在定义时确定 2](#_Toc536741562)

[函数上下文环境在调用时确定 2](#_Toc536741563)

[函数中this变量在调用时确定 2](#_Toc536741564)

[作用域 2](#_Toc536741565)

[闭包 3](#_Toc536741566)

[使用场景 3](#_Toc536741567)

[实际使用 3](#_Toc536741568)

[使用 4](#_Toc536741569)

[保持变量不变 4](#_Toc536741570)

[this 4](#_Toc536741571)

[this 4](#_Toc536741572)

[call 5](#_Toc536741573)

[预解析 6](#_Toc536741574)

[变量提升与闭包 6](#_Toc536741575)

[词法分析和AO 链 7](#_Toc536741576)

[原型（prototype）-对象的引用 11](#_Toc536741577)

[new的过程 11](#_Toc536741578)

[继承 11](#_Toc536741579)

[原型 11](#_Toc536741580)

[原型规则 11](#_Toc536741581)

[proto和prototype对比 12](#_Toc536741582)

[不能new一个对象，只能new一个函数 12](#_Toc536741583)

[总结 13](#_Toc536741584)

[原型链与作用域链 16](#_Toc536741585)

[特殊实例 16](#_Toc536741586)

[new的顺序，new obj 与 new obj() 16](#_Toc536741587)

[constructor 17](#_Toc536741588)

[es6 18](#_Toc536741589)

[String 18](#_Toc536741590)

[String和toString 18](#_Toc536741591)

[运算 18](#_Toc536741592)

[运算 18](#_Toc536741593)

[[] + '' 18](#_Toc536741594)

[({}) + '' 18](#_Toc536741595)

[优先级 19](#_Toc536741596)

[逻辑判断 19](#_Toc536741597)

[格式化 19](#_Toc536741598)

[Number和parseInt 19](#_Toc536741599)

[技巧 19](#_Toc536741600)

[void 0 19](#_Toc536741601)

[===和== 19](#_Toc536741602)

[判断 20](#_Toc536741603)

[isArray 20](#_Toc536741604)

[Object.prototype.toString.call 21](#_Toc536741605)

[逻辑运算符 21](#_Toc536741606)

[falsy 22](#_Toc536741607)

[创建新变量 22](#_Toc536741608)

[变量类型 22](#_Toc536741609)

[for 22](#_Toc536741610)

[超高级 22](#_Toc536741611)

[指针 22](#_Toc536741612)

[在html中 23](#_Toc536741613)

[阻止默认事件 23](#_Toc536741614)

[调试 24](#_Toc536741615)

[js堆栈（值，址传递） 24](#_Toc536741616)

[内存相关 24](#_Toc536741617)

[内存释放 24](#_Toc536741618)

[回收计算规则 24](#_Toc536741619)

[gc 25](#_Toc536741620)

[内存泄漏 25](#_Toc536741621)

[循环引用 25](#_Toc536741622)

[dom泄漏 25](#_Toc536741623)

[纯js 25](#_Toc536741624)

[页面崩溃和浏览器崩溃 26](#_Toc536741625)

[事件循环 26](#_Toc536741626)

[html相关 26](#_Toc536741627)

[事件回调的this 26](#_Toc536741628)

[react等框架解决的问题 26](#_Toc536741629)

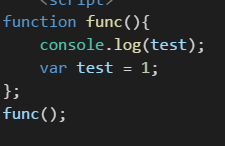
函数作用域在定义时确定 所以内部一般变量作用域在定义时已经确定

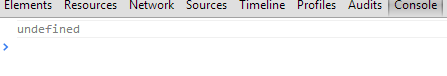
# 函数上下文环境在调用时确定

# 函数中this变量在调用时确定

# 作用域

函数作用域,全局作用域





不是not defined,进入函数就提前定义成undefined

## 闭包

### 使用场景

1函数作为返回值。

2函数作为参数传递。

### 实际使用

封装变量，收敛权限（1）

## 使用

### 保持变量不变



## this

### this

this的值在函数运行时确定。this也有作用域链（但this不考虑作用域链，因为每个作用域下都有this，所以不会向上层查找）

this不是代表这个函数，而是代表函数对象执行时**所在**作用域，比如f.b()中的this,因为b是f的属性，b执行时b所在的作用域为f的作用域，所以b中的this代表f。

var y = 500;

function Foo() {

this.name = 5;

var m = 50;

console.log(this.m); // undefined

console.log(m); // 50

console.log(this.y); // undefined

}

Foo.prototype.a = function () {

console.log(this.name);// 5

console.log(this)

// console.log(m); // 会报错，因为上层是window

}

var f = new Foo();

f.name = 1;

f.m = 10;

f.b = function () {

console.log(this.name); // 5，new的时候this被替换了

// console.log(m); // 会报错,因为定义时上层是window

}

console.log('Foo.prototype.a()');

Foo.prototype.a();// 这里的this代表prototype

// prototype有a,但是没有name属性，所以a为undefined

console.log('f.a()');

f.a();

f.b();var y = 500;

function Foo() {

this.name = 5;

var m = 50;

console.log(this.m); // undefined

console.log(m); // 50

console.log(this.y); // undefined new的时候this被替换了

}

Foo.prototype.a = function () {

console.log(this.name);// 1

console.log(this)

// console.log(m); // 会报错，因为上层是window

}

var f = new Foo();

f.name = 1;

f.m = 10;

f.b = function () {

console.log(this.name); // 1

// console.log(m); // 会报错,因为定义时上层是window

}

console.log('Foo.prototype.a()');

Foo.prototype.a();// 这里的this代表prototype

// prototype有a,但是没有name属性，所以a为undefined

console.log('f.a()');

f.a();// 1 {a,m,name}

f.b()//1

### call

bind和apply转换

function bindThis(f, oTarget) {

return f.bind(oTarget);

}

或

function bindThis(f, oTarget) {

return function(){

return f.apply(oTarget,arguments)

}

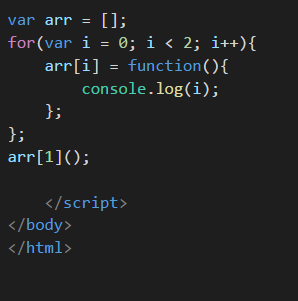
# 预解析

只储存变量名,不赋值, **但是如果是通过 function 进行函数声明的话则会将函数整体存储在执行环境中，所以函数执行可以写在函数声明之前**

alert(a); //undefined

var a=1;

## 变量提升与闭包



输出 2(区分arr[i]=function(i){})

**闭包**

**s作用域链**

简单来说，就是沿着作用域层级向上寻找变量（当前函数包括内部都不属于同一层级！！！除非调用了那些函数，进行数值覆盖！！！）

## 词法分析和AO 链

1、先分析参数

2、再分析变量的声明

3、分析函数说明

**具体的步骤：**

0：函数的在运行的瞬间，生成一个活动对象（Active Object)就是所谓的AO

1：分析参数

      1-1:函数接收参数，添加到AO的属性上面，值全部都是undefine,如AO.age=undefine

      1-2:接收实参，形成AO对应的属性值

2：分析变量声明，如var age,

     2-1：如果AO上还没有age属性，则添加AO 属性，值是undefine

     2-2：**如果AO 上面已经有了age属性,则不做任何操作**。

3：分析函数的声明，如果funcion foo(){},

3-1： 则把函数赋给AO.foo,如果**数据属性已经存在，则要被现在的新的值覆盖**

注意一个覆盖一个不覆盖,优先级3 >1> 2

**预解析过程不影响执行顺序！！！**

实例1

**function t(age) {**

**console.log(age);//1**

**var age = 99;**

**console.log(age);//2**

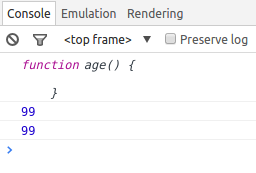
**function age() {**

**}**

**console.log(age);//3**

**}**

**t(5);**



**执行之前已经被预编译成函数,所以第一个log函数,然后age被赋值成99,,所以后面两个log 99**

实例2



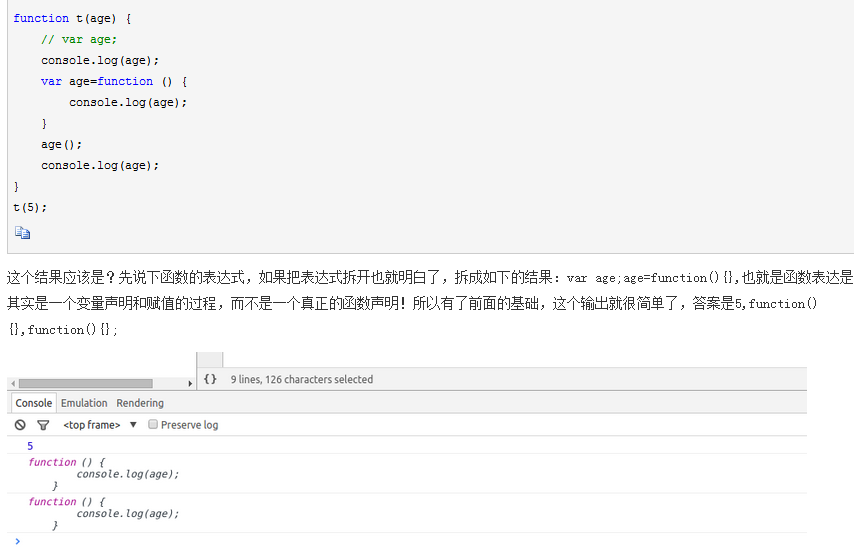
**实例3**

**var age=function () {**

**console.log(age);**

**}**

**赋值过程不是声明过程！！！**



**实例4**

**function t(age) {**

**// var age;**

**console.log(age);**

**var age=function age () {**

**console.log(age);**

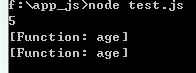
**}**

**age();**

**console.log(age);**

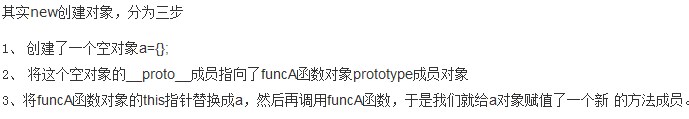
**}**

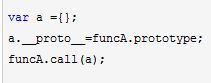
**t(5);**



# 原型（prototype）-对象的引用

## new的过程





而call过程是：

a.funcA=funcA;

a.funcA();

a.funcA=null;

通过改变this，这是简单的call，只是简单将this上的变量传递过去，实际call还要考虑参数（构造函数里没有参数）问题。

## 继承

所以写构造函数时一般将成员属性放在this里，成员方法放在prototype里。

## 原型

### 原型规则

1. 所有的引用类型( 数组 对象 函数 ) 都是 具有对象特性即自由拓展属性 （除了 null以外）

2. 所有的引用类型（数组 对象 函数 ） 都有一个 proto 属性。属性值是一个普通的对象

**3. 所有的函数 都有一个prototype 属性 ，属性值也是一个普通的对象(普通对象没有)**

4. 所有的引用类型（数组 对象 函数），proto 属性值指向它的构造函数 ”prototype“ 属性值

5. 当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性那么会去它的\_\_proto\_\_(即它的构造函数的prototype寻找)(不会找本身的prototype: Array.concat() 报错is not a function)

### proto和prototype对比

是\_\_proto\_\_不是\_proto\_(两根短线)

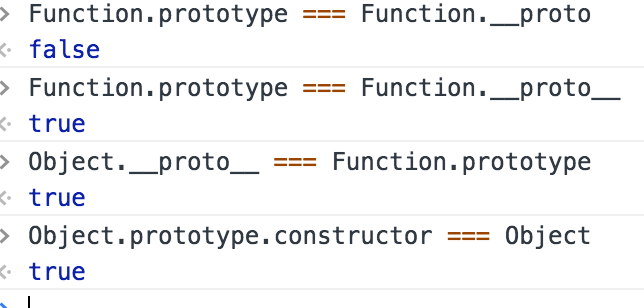
所有引用类型都有隐式proto,**指向构造函数的prototype(直接{} 指向Object的prototype)**

所有函数都有显式prototype,是普通的对象,prototype是创建的时候生成的（没有直接指向什么。），prototype的proto指向Object的prototype。

Function的proto指向自身的prototype。

Object的proto指向Funtion的prototype。

Object的prototype是独立创建的。





**默认对象的原型是Object(错) Object的proto是undifined（错）**

**当b=new a()时，b.prototype===a.\_proto\_;（错）正确应该是b.\_proto\_===a.prototype**

### 不能new一个对象，只能new一个函数

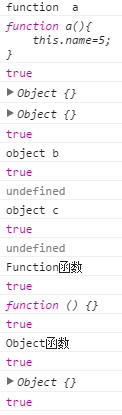
**function a(){}**

**b=new a();正确**

**d=new b();错误**

### 总结





函数由函数创建（比如普通函数由Function函数创建）函数的\_\_proto\_\_是Function.prototype(这个和函数继承没关系，函数继承是手动改变子函数prototype。), 函数的prototype是普通对象

对象由函数创建（比如普通对象由Object函数创建，对象的\_\_proto\_\_是创建它的函数的prototype，对象的prototype是null）

Function函数由自身创建，Function函数的原型是Funtion(Function.\_\_proto\_\_ === Function.prototype)。这里是根！！！

函数的prototype是一个普通对象，所以Func.prototype.\_\_proto\_\_ === Object.protype

Object函数由Function函数创建，Object函数的\_\_proto\_\_是Function.prototype

**一切都是指向，都是一个的\_\_proto\_\_指向另一个的prototype，可以把prototype理解成原型，\_\_proto\_\_理解成一个指向原型的指针，对象由指针，函数有原型**

**区分function与Function，object与Object**

**区分object和Object();**

**区分.prototype .\_\_proto\_\_ .prototype.\_\_proto\_\_**

**其他人理解：**

自定义函数Foo.\_\_proto\_\_指向Function.prototype，Object.\_\_proto\_\_指向Function.prototype，唉，怎么还有一个……Function.\_\_proto\_\_指向Function.prototype？这不成了循环引用了？

对！是一个环形结构。

其实稍微想一下就明白了。Function也是一个函数，函数是一种对象，也有\_\_proto\_\_属性。既然是函数，那么它一定是被Function创建。所以——Function是被自身创建的。所以它的\_\_proto\_\_指向了自身的Prototype。

所以继承是都是用挂在prototype上，子对象才能通过proto一层层访问到（当我们访问一个对象的属性 时，如果这个对象内部不存在这个属性，那么他就会去\_\_proto\_\_里找这个属性，这个\_\_proto\_\_又会有自己的\_\_proto\_\_，于是就这样 一直找下去，也就是我们平时所说的原型链的概念。）

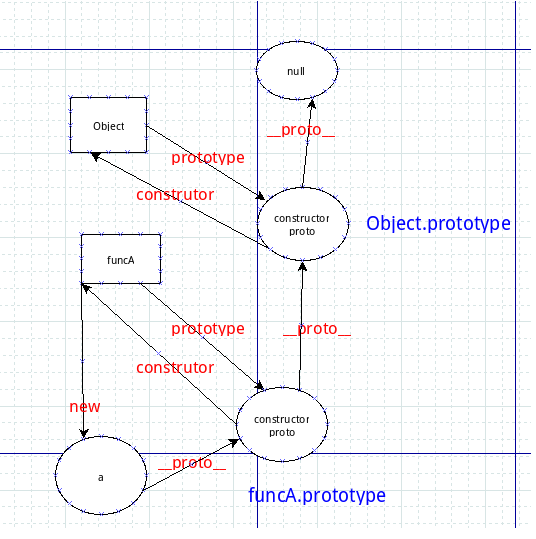
最后，其实prototype只是一个假象，他在实现原型链中只是起到了一个辅助作用，换句话说，他只是在new的时候有着一定的价值，而原型链的本质，其实在于\_\_proto\_\_！

Object

Object.prototype:匿名函数

Object.constructor:函数





可以看到对象和对象之间没有直接关系，是通过prototype和proto联系在一起的

## 原型链与作用域链

作用域链是判断一个属性是否存在，位于哪个作用域。

原型链是去查找一个属性具体的值

Javascript 解析引擎在读取一个Object的属性的值时，会沿着原型链向上寻找，如果最终没有找到，则该属性值为undefined；

## 特殊实例

### new的顺序，new obj 与 new obj()

new obj 等效于new obj()

先proto指向,在转移this,所以可以被覆盖

    function obj(name){

        this.name=name;

}

obj.prototype.name= "name2";

var a = obj("name1");

var b = new obj;

console.log(a);

console.log(b.name);

注意b



**注意不是new obj('name1'),第二个undefined是因为没有传参数。**

调用call函数时,this.name没有赋值，所以是undefined。

变化

    function obj(name){

        function one(){}

        one.prototype.name=name;

        if(name){

            return new one();

        }

}

obj.prototype.name= "name2";

var a = obj("name1");

var b = new obj;

console.log(a.name);

console.log(b.name);



## constructor

constructor在创建函数时创建，指向函数本身，挂在prototype上，所以函数创建的对象也能拿到，但对象上本身没有，是通过原型链拿到的。

Func.prototype.constructor: 指向函数本身。

Func.constructor: Function

Function.constructor Function 函数也是由函数创建的

# es6

var map = new Map(Object.entries(obj));

# String

## String和toString

toString不能转换null,undefined（用的话会报错）。可以写一个数字，代表进制，对应进制字符串。String可以转null,undefined(变成‘null’等)，不能写进制。

# 运算

## 运算

### [] + ''

[1,2,3] + ' '

"1,2,3 "

### ({}) + ''

({a: 'a'}) + ' '

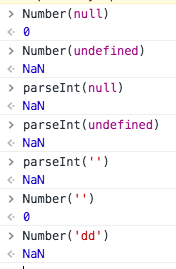
"[object Object] "

### 优先级

## 逻辑判断

## 格式化

### Number和parseInt



所以永远不要用Number。

# 技巧

## void 0

## ===和==

1、对于string,number等基础类型，==和===是有区别的

1）不同类型间比较，==之比较“转化成同一类型后的值”看“值”是否相等，===如果类型不同，其结果就是不等

2）同类型比较，直接进行“值”比较，两者结果一样

2、对于Array,Object等高级类型，==和===是没有区别的

进行“指针地址”比较

3、基础类型与高级类型，==和===是有区别的

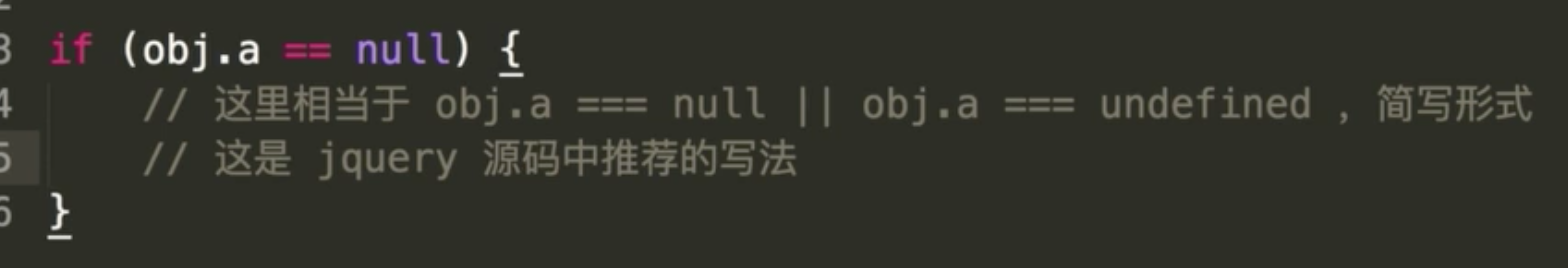
1）对于==，将高级转化为基础类型，进行“值”比较

2）因为类型不同，===结果为false

null undefined == true === false

ecma3就有了。

推荐除了下面这种其他都用===



## 判断

### isArray

function isArray(arr){

    return typeof arr == "object" && arr.constructor == Array;

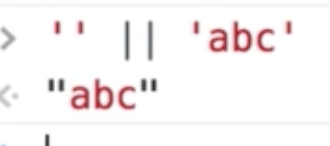
}

### Object.prototype.toString.call



## 逻辑运算符

返回并不是true,false，只是可以通过true,false判断。要返回true/false，加上!!





## falsy

0,NaN,false,'',null,undefined

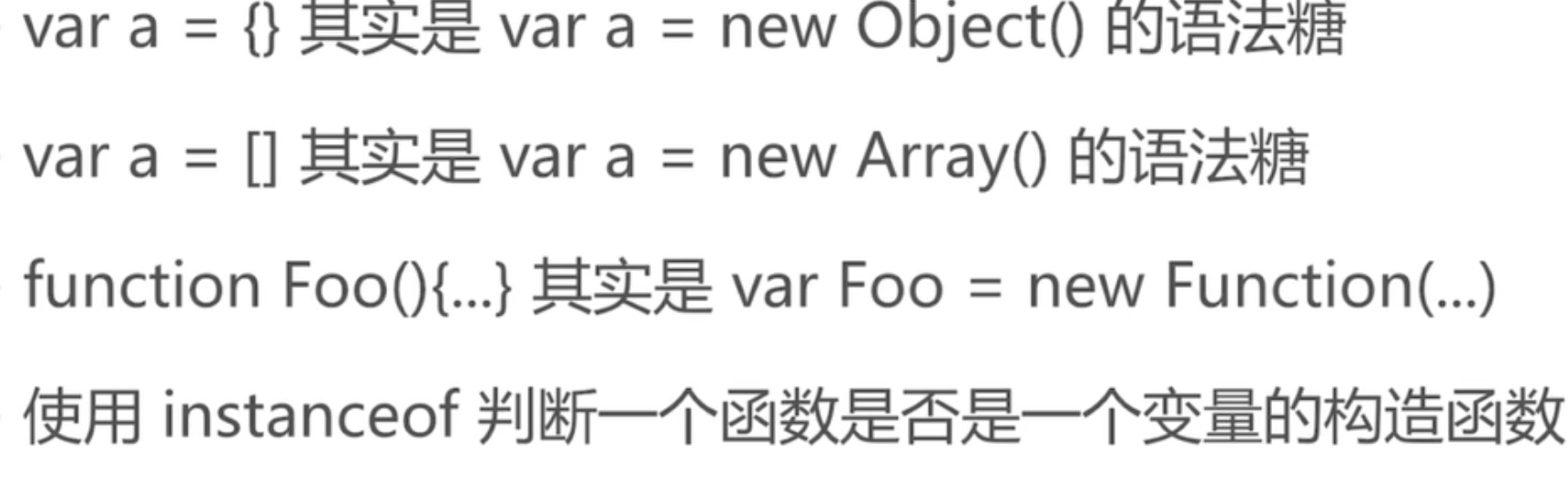
注意 {} [] 不是falsy

function a(){if(({})) return true} => a() => return true

0 false '' 可以互==,

null undefined可以 ==

## 创建新变量



## 变量类型

boolean number string null undefined

object

symbol

## for

对对象使用，遍历key值。

for (var i in temp) {

tempArr[len++] = i;

}

# 超高级

## 指针

function namespace(oNamespace, sPackage) {

var arr = sPackage.split('.');

var res = oNamespace; // 保留对原始对象的引用

for(var i = 0, len = arr.length; i < len; i++) {

if(arr[i] in oNamespace) { // 空间名在对象中

if(typeof oNamespace[arr[i]] !== "object") { // 为原始值

oNamespace[arr[i]] = {}; // 将此属性设为空对象

}

} else { // 空间名不在对象中，建立此空间名属性，赋值为空

oNamespace[arr[i]] = {};

}

oNamespace = oNamespace[arr[i]]; // 将指针指向下一个空间名属性。

}

return res;

}

# 在html中

## 阻止默认事件

例如:<a href="http:www.baidu.com;alert(11);**return false;**alert(22)">链接</a> <!-- 只会执行了alert(11);-->

**return false 就相当于终止符，return true 就相当于执行符。**

**在js中return false的作用一般是用来取消默认动作的。**

**return false的含义不是阻止事件继续向顶层元素传播，而是阻止浏览器对事件的默认处理。**

**return false写在function里面：**

如果将return false写在function里面，只是跳到函数调用的地方。

<form action="index.jsp" method="post" onsubmit="submitTest();"> <INPUT value="www"> <input type="submit" value="submit"></form>

<SCRIPT LANGUAGE="JavaScript"> function submitTest() { // 一些逻辑判断

return false; }

</SCRIPT>

以上代码实际情况是表单正常提交，若想它不提交，应该将

<form action="index.jsp" method="post" onsubmit="submitTest();">

改为

<form action="index.jsp" method="post" onsubmit="return submitTest();">（这样写function里一定要return false）

**returnvalue：**

JAVASCRIPT在事件中调用函数时用return返回值实际上是对window.event.returnvalue进行设置。

而该值决定了当前操作是否继续。

当返回的是true时，将继续操作。

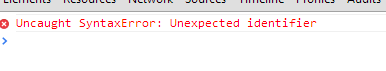
当返回是false时，将中断操作。

而直接执行时（不用return）。将不会对window.event.returnvalue进行设置

# 调试

**Let和const需要严格模式**

**Chome谷歌调试不支持es6**



# js堆栈（值，址传递）

变量存放在栈中。

简单类型栈中存放值。

引用类型栈中存放指针，实际值存放在堆中。

var a=[{x:1},2,3,4];

var b=a[0];

a[0]=5;

console.log(b);／／{ x: 1}

解释：a[0]是一个地址，b是指向同一个地址的不同指针，后面a[0]变成简单类型，a[0]中存放5，但b中依然存放的地址。所以没有影响堆里{x:1}这个变量。

如果改成a[0].x=5;打印出来{x:5}

# 内存相关

## 内存释放

垃圾回收器会按照固定的时间间隔周期性的执行。

只可能是局部变量，全局变量的生命周期直至浏览器卸载页面才会结束。局部变量只在函数的执行过程中存在。

### 回收计算规则

两种方式结合，引用计数是被动清除，标记是主动清除

引用计数

定义时次数为0，被引用一次加1，引用它的变量取得了另一个值，它的次数减1。

标记清除：

### gc

每次 GC 时，是根据 root 对象 (浏览器环境下的 window，Node.js 环境下的 global ) 依次梳理对象的引用，如果能从 root 的引用链到达访问，V8 就会将其标记为可到达对象，反之为不可到达对象。被标记为不可到达对象（即无引用的对象）后就会被 V8 回收。

## 内存泄漏

### 循环引用

window.onload=function outerFunction(){

var obj = document.getElementById("element");

obj.onclick=function innerFunction(){};

};

解决：

$(obj).click(function innerFunction(){});

jQuery绑定事件最终都没有直接绑定到DOM对象上，而是使用jQuery缓存来绑定的。

对于ECMAScript 对象而言，只要没有其他对象引用对象 a、b，也就是说它们只是相互之间的引用，那么仍然会被垃圾收集系统识别并处理。对于dom和其他对象，不清楚最新的浏览器情况。

### dom泄漏

当原有的DOM被移除时，子结点引用没有被移除则无法回收。

DOM 插入顺序导致内存泄漏（页面交叉）

dom清空或删除时，事件未清除导致的内存泄漏(jquery 的 empty和 remove会帮助开发者避免这个问题)

### 纯js

全局变量

闭包（结合全局变量）

事件监听：重复监听

## 页面崩溃和浏览器崩溃

# 事件循环

https://segmentfault.com/a/1190000010622146

任务和微任务

https://segmentfault.com/a/1190000013660033?utm\_source=channel-hottest

node的任务队列更多一些。node中：nextTick优先级比promise等microtask高。setTimeout和setInterval优先级比setImmediate高。

# html相关

## 事件回调的this

一般事件回调的this指向的是e.target。

# react等框架解决的问题

逻辑代码和页面渲染的问题。

react是生命周期结合jsx语法。

vue是模板字符串。

比如一个列表，点击的某一项给选中状态，先前选中的失去选中状态，保存选中的数据。原生写的就比较多，而有生命周期的话只需要更新那个数据既可以了，页面自己重新渲染。