**作用域**

概念“{}”中间的部分都是块级作用域ex：for while if ，js中没有块级作用域，但是可以用闭包实现类似功能。

<script type="text/javascript">

var i=1;

if(i==1){

var b=2;

}

alert(b);//输出2

for(var j=0;j<=2;j++){

var c=3; //c为函数作用范围内的全局变量

}

alert(c);//输出3

</script>

**Try catch**

catch语句会形成一个词法作用域

(function(){

e="default"; // e为全局变量 跟catch中的e无关

try{

throw "test";

}catch(e){ //catch捕获的e是throw 返回的e或错误事件，可以看出是一个错误对象集合

var e,x=123; // 这里定义变量e 作用域function顶层 跟catch无关

console.log(e); //test //这里的e取到的是catch 返回的 e对象

e=456;

console.log(e); //456

};

console.log(x); //123

console.log(e); //default

console.log(window.e); //undefined

})();

1. 首先谨记js只有函数作用域，语句块只是函数的一部分，try-catch也不例外。所以catch中的内容和外边function是在一个作用域中的，也因此，因为catch中有var e；所以function里的e也变成了局部变量。 2.catch确实会对作用域链做一个小动作，这个小动作就是把catch(e)中的e单独放入一个作用域，然后把这个作用域加入作用域链的最前端。所以catch花括号中的var e是在function的作用域中进行，而e=456时的这个e实际上是catch(e)中的e。这里之所以让你迷惑是因为你犯了个大错：javascript里delete不能用来删除变量，只能用来删除属性。你可以在delete e那一行之后再console.log(e)一下，你会发现还是输出test，这个变量没有被删除。

<https://www.web-tinker.com/article/21242.html>

事件冒泡和捕获，元素的冒泡和捕获支持情况

**Dispatchenent 自定义事件触发器**

var fireEvent = function(element,event){

if (document.createEventObject){

// IE浏览器支持fireEvent方法

var evt = document.createEventObject();

return element.fireEvent('on'+event,evt)

}

else{

// 其他标准浏览器使用dispatchEvent方法

var evt = document.createEvent( 'HTMLEvents' );

evt.initEvent(event, true, true);

return !element.dispatchEvent(evt);

}

};

**JS 函数变量提升**

console.log(foo);

function foo(){

console.log("函数声明");

}

var foo = "变量";

等同于

function foo(){

console.log("函数声明");

}

var foo;

console.log(foo);

foo = "变量";

**JavaScript 提升**

<http://blog.csdn.net/ligang2585116/article/details/46271699>

**JavaScript中的this**

<http://blog.csdn.net/ligang2585116/article/details/47059289>

**JavaScript 函数、对象**

<http://blog.csdn.net/ligang2585116/article/details/47379317>

**JavaScript作用域闭包**

<http://blog.csdn.net/ligang2585116/article/details/46850399>

**JavaScript 词法作用域**

<http://blog.csdn.net/ligang2585116/article/details/46367565>

**Arguments 、callee、 caller**

**function** foo(a) {

// var a; //定义a，不赋值并不影响参数a的传值

**var** b = a;

**return** a + b;

}

**var** c = foo( 2 ); //4

function foo(a) {

var a; //定义了变量，但是跟参数a同名，导致变量a被参数赋值

console.log(typeof(a))

console.log(arguments[0])

console.log(a)

var b = a;

a = 10; //对a的赋值才会覆盖arguments 参数赋值

return a + b;

}

var c = foo( 2 );

c

**JS作用域**

作者：通哥

链接：https://zhuanlan.zhihu.com/p/25327657

来源：知乎

著作权归作者所有，转载请联系作者获得授权。

在JavaScript语言中无块级作用域

function Main(){

if(1==1){

var name = 'seven';

}

console.log(name);

}

// 输出： seven

补充：标题之所以添加双引号是因为JavaScript6中新引入了 let 关键字，用于指定变量属于块级作用域。

#二、JavaScript采用函数作用域

在JavaScript中每个函数作为一个作用域，在外部无法访问内部作用域中的变量。

function Main(){

var innerValue = 'seven';

}

Main();

console.log(innerValue);

// 报错：Uncaught ReferenceError: innerValue is not defined

#三、JavaScript的作用域链

由于JavaScript中的每个函数作为一个作用域，如果出现函数嵌套函数，则就会出现作用域链。

xo = 'alex';

function Func(){

var xo = "seven";

function inner(){

var xo = 'alvin';

console.log(xo);

}

inner();

}

Func();

如上述代码则出现三个作用域组成的作用域链，如果出现作用域链后，那么寻找变量时候就会出现顺序，对于上述实例：

当执行console.log(xo)时，其寻找顺序为根据作用域链从内到外的优先级寻找，如果内层没有就逐步向上找，直到没找到抛出异常。

#四、JavaScript的作用域链执行前已创建

JavaScript的作用域在被执行之前已经创建，日后再去执行时只需要按照作用域链去寻找即可。

示例一：

xo = 'alex';

function Func(){

var xo = "seven";

function inner(){

console.log(xo);

}

return inner;

}

var ret = Func();

ret();

// 输出结果： seven

上述代码，在函数被调用之前作用域链已经存在：

全局作用域 -> Func函数作用域 -> inner函数作用域

当执行【ret();】时，由于其代指的是inner函数，此函数的作用域链在执行之前已经被定义为：全局作用域 -> Func函数作用域 -> inner函数作用域，所以，在执行【ret();】时，会根据已经存在的作用域链去寻找变量。

示例二：

xo = 'alex';

function Func(){

var xo = "eirc";

function inner(){

console.log(xo);

}

xo = 'seven';

return inner;

}

var ret = Func();

ret();

// 输出结果： seven

上述代码和示例一的目的相同，也是强调在函数被调用之前作用域链已经存在：

全局作用域 -> Func函数作用域 -> inner函数作用域

不同的时，在执行【var ret = Func();】时，Func作用域中的xo变量的值已经由 “eric” 被重置为 “seven”，所以之后再执行【ret();】时，就只能找到“seven”。

示例三：

xo = 'alex';<br>

function Bar(){

console.log(xo);

}

function Func(){

var xo = "seven";

return Bar;

}

var ret = Func();

ret();

// 输出结果： alex

上述代码，在函数被执行之前已经创建了两条作用域链：

全局作用域 -> Bar函数作用域

全局作用域 -> Func函数作用域

当执行【ret();】时，ret代指的Bar函数，而Bar函数的作用域链已经存在：全局作用域 -> Bar函数作用域，所以，执行时会根据已经存在的作用域链去寻找。

#五、声明提前

在JavaScript中如果不创建变量，直接去使用，则报错：

console.log(xxoo);

// 报错：Uncaught ReferenceError: xxoo is not defined

JavaScript中如果创建值而不赋值，则该值为 undefined，如：

var xxoo;

console.log(xxoo);

// 输出：undefined

在函数内如果这么写：

function Foo(){

console.log(xo);

var xo = 'seven';

}

Foo();

// 输出：undefined

上述代码，不报错而是输出 undefined，其原因是：JavaScript的函数在被执行之前，会将其中的变量全部声明，而不赋值。所以，相当于上述实例中，函数在“预编译”时，已经执行了var xo；所以上述代码中输出的是undefined。

作者：Liril

链接：https://zhuanlan.zhihu.com/p/25407758

来源：知乎

著作权归作者所有，转载请联系作者获得授权。

金三银四搞事季，前端这个近年的热门领域，搞事气氛特别强烈，我朋友小伟最近就在疯狂面试，遇到了许多有趣的面试官，有趣的面试题，我来帮这个搞事 boy 转述一下。

以下是我一个朋友的故事，真的不是我。

for (var i = 0; i < 5; i++) {

console.log(i);

}

“小伟，你说说这几行代码会输出什么？”

当面试官在 Sublime 打出这几行代码时，我竟有点蒙蔽。蛤？这不是最简单的一个循环吗？是不是有陷阱啊，我思索一下，这好像和我看的那个闭包的题很像啊，这面试官是不是没写完啊？有毒啊。

“应该是直接输出 0 到 4 吧...”，我弱弱的说到。

“是啊，别紧张，这题没啥陷阱，我就是随便写一下。”

（Excuse me？面试官你是来搞笑的吗，吓死老子了！）

“那你在看看这几行代码会输出什么？”

for (var i = 0; i < 5; i++) {

setTimeout(function() {

console.log(i);

}, 1000 \* i);

}

额，什么鬼，怎么还不是我背了那么多遍的那道闭包题，让我想想。 setTimeout 会延迟执行，那么执行到 console.log 的时候，其实 i 已经变成 5 了，对，就是这样，这么简单怎么可能难到老子。

“应该是开始输出一个 5，然后每隔一秒再输出一个 5，一共 5 个 5。”

“对，那应该怎么改才能输出 0 到 4 呢？”

终于到我熟悉的了，加个闭包就解决了，稳！

for (var i = 0; i < 5; i++) {

(function(i) {

setTimeout(function() {

console.log(i);

}, i \* 1000);

})(i);

}

“很好，那你能说一下，我删掉这个 i 会发生什么吗？”

for (var i = 0; i < 5; i++) {

(function() {

setTimeout(function() {

console.log(i);

}, i \* 1000);

})(i);

}

“这样子的话，内部其实没有对 i 保持引用，其实会变成输出 5。”

“很好，那我给你改一下，你看看会输出什么？”

for (var i = 0; i < 5; i++) {

setTimeout((function(i) {

console.log(i);

})(i), i \* 1000);

}

蛤？什么鬼，这是什么情况，让我想想。这里给 setTimeout 传递了一个立即执行函数。额，setTimeout 可以接受函数或者字符串作为参数，那么这里立即执行函数是个啥呢，应该是个 undefined ，也就是说等价于：

setTimeout(undefined, ...);

而立即执行函数会立即执行，那么应该是立马输出的。

“应该是立马输出 0 到 4 吧。”

“哎哟，不错哦，最后一题，你对 Promise 了解吧？”

“还可以吧...”

“OK，那你试试这道题。”

setTimeout(function() {

console.log(1)

}, 0);

new Promise(function executor(resolve) {

console.log(2);

for( var i=0 ; i<10000 ; i++ ) {

i == 9999 && resolve();

}

console.log(3);

}).then(function() {

console.log(4);

});

console.log(5);

WTF！！！！我想静静！

这道题应该考察我 JavaScript 的运行机制的，让我理一下思路。

首先先碰到一个 setTimeout，于是会先设置一个定时，在定时结束后将传递这个函数放到任务队列里面，因此开始肯定不会输出 1 。

然后是一个 Promise，里面的函数是直接执行的，因此应该直接输出 2 3 。

然后，Promise 的 then 应当会放到当前 tick 的最后，但是还是在当前 tick 中。

因此，应当先输出 5，然后再输出 4 。

最后在到下一个 tick，就是 1 。

“2 3 5 4 1”

“好滴，等待下一轮面试吧。”

So easy！妈妈再也不用担心我的面试了。