Table of Contents

Introduction

作用域

编程语言最基本功能之一：储存变量当中的值，并且能在之后对这个值进行访问和修改

状态：储存和访问变量的值的能力

需要一套规则来维护状态，即储存变量，且能方便地找到这些变量，这套规则称为**【作用域】**.

> js引擎不会有大量的时间来进行优化，因为js大部分情况下编译发生在代码执行前的几微秒时间内

查找规则

为何会区分LHS查询和RHS查询呢?

> 因为RHS失败会抛出一个referenceError异常，而LHS查询失败，在非严格模式下，会创建一个全局作用域中具有该名称的变量，并将其返还给引擎

LHS查找

指代赋值语句左侧查找，比如👇：

```
var a = 1
```

这里引擎会为变量a进行LHS查询

RHS查找

指代非左侧查询，比如下面两个例子都是RHS查询

```
var a = b
console.log(b)
```

这里执行了两次b的RHS查询

小结

LHS和RHS的根本

LHS查询：Left Hand Side 对变量进行赋值
RHS查询：Right Hand Side 获取变量的值

LHS和RHS的共同点和不同点

LHS和RHS都是从当前作用域开始查找，直到全局作用域。全局作用域也没有的话，RHS查询会抛出referenceError错误，LHS会隐式在全局作用域中创建一个变量

代码解析过程

```
var a = 2
```

上面代码会经历如下几个步骤：

1. var a在其作用域中声明新变量
2. a = 2进行LHS查询变量a并对其进行赋值

作用域的原理

一套用来管理引擎如何在当前作用域以及嵌套的子作用域中根据标识符名称进行变量查找的规则

词法作用域

词法作用域是由你在写代码时将变量和块级作用域写在哪决定的

动态作用域

js中的作用域就是词法作用域，不具备动态作用域

词法作用域： 在写代码或者定义时确定
动态作用域： 在运行时确定

欺骗词法作用域

欺骗词法作用域有两种方法：

eval

eval会把字符串当作代码块执行

eval的副作用

影响原作用域

eval执行的代码类似本插入在此的代码，其中的声明和操作会影响当前作用域，比如👇：

```
function foo (code) {
    eval(code)
    console.log(b)
}

var b = 2;
foo('var b = 3;')
```

这段代码会输出3，说明eval在foo的词法作用域中声明了变量b为3

如何解决?
1.严格模式

```
function foo (code) {
    "use strict"
    eval(code)
    console.log(b)
}

var b = 2;
foo('var b = 3;')
```

2.new Function代替eval

```
function foo (code) {
    new Function(code)
    console.log(b)
}

var b = 2;
foo('var b = 3;')
```

其他执行字符串的函数

1. setTimeout
2. setInterval

这些功能已经过时，不要使用它们！！！

with

with可以把一个对象处理为词法作用域

变量泄漏

在with中定义的变量，会泄漏到父级作用域中

```
function foo () {
    var obj = {
        name: 'hahaha'
    }
    with (obj) {
        console.log(name);
        var b = 11
    }
    console.log(b)
}
foo()
```

b会输出11。说明with中定义的变量泄漏到了foo作用域中

小结

1. eval和with会在运行修改或创建新的作用域
2. js引擎会在编译时根据代码的词法进行静态分析，预先确定变量和函数的定义位置，才能在执行过程中快速定位标识符
3. 因为无法在词法分析阶段明确知道eval会接收到什么代码，这些代码会如何对作用域进行修改，也无法确定with用来创建词法作用域的对象的内容到底是什么，所以会导致js引擎无法做优化
4. 如果代码中大量使用eval和with，会导致运行速度非常慢

函数作用域

最小暴露原则

在软件设计中，应该最小限度地暴露必要内容，而将其他内容都隐藏起来

1. 控制访问权限
2. 规避冲突，如命名冲突

匿名函数

匿名函数地缺点：

1. 在栈追踪中不会显示出有意义地函数名，使得调试困难
2. 函数无法引用自身，只能通过arguments.callee
3. 一个描述性地名称可以让代码不言自明，可读性变差

始终给函数表达式命名是一个最佳实践

IIFE

自执行函数

```
(function IIFE() {})()

=> 很多人更下面的形式👇

(function IIFE() {}())
```

IIFE解决undefined的问题

undefined有什么问题？ => undefined并不是一个关键字，可能会被覆盖，就像下面这样👇

```
undefined = true
```

解决方案：

1. 使用void(0)作为undefined
2. 使用IIFE

```
(function IIFE(undefined){})()
```

提升

函数声明会提升，函数表达式不会提升

```
foo() // 这里会报typeError，因为foo提升是undefined，并不是个函数
var foo = function () {
  // ...
}
```

换个具名来看看

```
foo(); // TypeError
bar(); // ReferenceError
var foo = function bar() {
  // ...
};
```

foo，bar都会报错，是因为这段代码提升之后变成了下面这样👇：

```
var foo;
foo(); // TypeError
bar(); // ReferenceError
foo = function() {
  var bar = ...self...
  // ...
}
```

函数优先

意指：函数首先被提升，然后才是变量
=> 重复声明会被忽略

```
foo(); // 1
var foo;
function foo() {
    console.log( 1 );
}
foo = function() {
    console.log( 2 );
};
```

=> 重复的函数声明会被覆盖

闭包

函数嵌套，js引擎变量查找的规则引起的一种现象

1. 函数能记住并访问所在的词法作用域
2. 即使函数是在当前词法作用域之外执行

现象

1. 内部函数可以访问外部作用域的变量
2. 外部作用域的变量会一直存在内存中

应用

1. 函数柯理化
2. 块级作用域变量泄漏

```
<!-- 函数柯理化 -->
function foo (context) {
  return function (...args) {
    return Array.prototype.push.apply(context, args)
  }
}
```

```
<!-- 闭包解决for+setTimeout问题 -->
for (var i = 0; i < 10; i++) {
  (function (i) {
    setTimeout(() => {
      console.log(i)
    }, i * 1000)
  })(i)
}
```

this

this是在运行时进行绑定的，取决于函数的调用方式

箭头函数 =>

用一个例子来说明下除了书写方便之外，之前的function声明有何问题?

```
var obj = {
  name: 'leo',
  call: function () {
    console.log(this.name)
  }
}
let name = 'global'
obj.call()
setTimeout(obj.call, 0)

<!-- 输出 -->
leo
global
```

setTimeout那里的obj.call丢失了同this之间的绑定

词法作用域作为this

把上面的demo，function修改成箭头函数 =>，发现this都变成了global

箭头函数是把当前的词法作用域作为this

绑定规则

默认绑定

独立函数调用，最常用的函数调用类型
如果函数是严格模式下，那么默认绑定不会绑到全局对象

```
function foo () {
    "use strict";
    console.log(this.name)
}
var a = 2;
foo(); // TypeError: this is undefined
```

隐式绑定

场景

函数调用时：用obj上下文来引用函数

```
obj.foo()
```

隐式丢失

```
var baz = obj.foo;
haz();
```

显式绑定

call，apply，bind

这些函数如果传入的是null，那么还是按照默认规则执行

new绑定

自定义bind

```
function foo () {
    console.log(this.name)
}
var name = 'i am global'
var obj = {
    name: 'i am foo'
}
Function.prototype._bind = function (context) {
    let fn = this
    return function (...args) {
        fn.apply(context, args)
    }
}
foo._bind(obj)()
```

```
<!-- MDN实现 -->
if (!Function.prototype.bind) { Function.prototype.bind = f
if (typeof this !== "function") { // 与 ECMAScript 5 最接近的
throw new TypeError(
                    "Function.prototype.bind - what is try
                    "to be bound is not callable"
 bar(3) 并没有像我们预计的那样把 obj1.a
 this全面解析 | 93
); }
var aArgs = Array.prototype.slice.call( arguments, 1 ), fTo
fNOP = function(){}, fBound = function(){
return fToBind.apply( (
this instanceof fNOP &&
oThis ? this : oThis ),
                        aArgs.concat(
                           Array.prototype.slice.call( arg
); }
;
fNOP.prototype = this.prototype;
fBound.prototype = new fNOP(); return fBound;
}; }
```

DMZ对象

> DMZ：非军事区

```
<!-- ø 表示：我希望this是空 -->
var ø = Object.create(null)
```