4. JS的运行机制

JS代码运行的环境

1. 浏览器 → webkit内核（V8）、Trident、Gecko、Blink...
2. Node → webkit内核
3. webview「Hybrid混合APP开发」 → webkit内核
4. ...

JS的运行机制:

1. 当浏览器(内核/引擎)渲染和解析JS代码的时候,会为JS代码提供一个可运行的环境,即“全局作用域(global/window scope)”
2. 代码自上而下执行(执行前存在变量提升阶段)

什么是栈与堆?

1. **栈内存(Stack)**:

   💡 执行环境栈 ECStack(Execution Context Stack):

   1.供原始值和变量存储

   2.供代码执行

   - 全局执行上下文 EC(G) (Execution Context Global):

     1.区分代码的执行环境

     2.全局下的代码会在全局上下文中执行

   - 变量对象VO(Varieble Object)/活跃对象AO(Active Object):

     1. 存放当前上下文中声明的变量

2. **堆内存(Heap)**:

   存放复杂数据类型, 并通过地址指向

JS代码在浏览器中运行过程

1. 等号=赋值操作过程

💡 原始数据类型的赋值操作全程在栈内存中执行

1.原始值: 在栈内存中单独找一个位置存储.
2.声明变量 Declare 把声明的变量存储到当前上下文的变量对象VO(Variable Object)
3.让创建的变量与值关联在一起,也就是定义(Defined)

💡 复杂数据类型的值会存储在堆内存中

```
1.对象：在堆内存中开辟一个新的空间存储.
2.产生一个供访问的16进制地址
3.依次将键值对存入对象
4.将空间地址存在栈中以供变量指向.
```

💡 全局对象GO (Global Object) : 就是一个存放在堆中的对象 存储了很多浏览器内置的API 如: JSON/ console / setTimeOut

2. 等号=赋值优先级问题

> 运算从右到左要注意运算优先级的问题

参考: 运算符优先级 - JavaScript | MDN

3. 执行上下文EC(G)

1. 词法解析(AST)

> 基于HTTP从服务器拉回来的代码其实是一串字符串, 浏览器会基于 ECMAscript 规则, 将这些代码转换为 C++能够识别的和解析的一套树结构对象

在词法解析阶段, 引擎就已经知道了在哪些地方存在哪些变量, 并且发现了已经被申明的变量基于let/const 重新声明, 则直接报错,不会执行代码

```
var x = 10;
var x = 20;
console.log(x);
let y = 10;
let y = 20; // 以上的代码全部不会执行, 直接报错
// Uncaught SyntaxError: Identifier 'y' has already been declaredconsole.log(y);
```

2. 产生全局上下文

变量提升(预解析): 在代码自顶向下执行之前, 浏览器会将`var function`的变量提前声明/定义

- 带var的变量先声明, 但不定义/赋值
- function则是声明 + 定义

3.闭包机制

1. **如何形成:** 当函数执行的时候,会产生一个私有上下文, 里面存储着该函数的私有变量, 当函数执行完, 其堆内存就会被出栈释放, 但是在某些情况下(如 return 出去的值引用了该上下文的私有变量), 就算函数执行完了, 它的私有变量也不会被释放, 这就形成了闭包.
2. **如何理解:** 闭包是一种机制, 函数形成私有的上下文, 一方面要保护上下文中的变量不被污染, 另一方面,如果不被释放, 私有变量以及它们的值又会被保存起来, 这种`保护`+`保存`的机制就成为闭包.

> 并不一定要 return function 才会产生闭包

```
function A(a){
    // 形参a: 1  => 该值在函数第二次执行的时候仍被保存着
    A= function(b) {
        console.log(a + b++)
    }
}
A(1)
A(2)
```

4. 声明变量var, let, const

var: varieble 声明的值是一个变量, 存在变量提升, 跟let有一定的差异

let: let声明的值是一个变量, 即可更改.

const: constant 声明的是一个常量, 不可改.

```
const obj = {
    name: 'link'
}
obj.name = 'chen'

console.log(obj.name) // `chen`
```

var, let 之间的差异

1. var存在变量提升
2. let不允许变量重复声明, 无论基于什么方式声明了一个变量, 再次用let/const声明都会报错.
3. 全局上下文中, var声明的变量会直接存放在**GO(window)**下, 而let声明的变量存放在**变量对象VO**中.
4. let存在暂时性死区 (在变量声明的代码行前不允许使用该变量)
5. let会产生块级作用域

```
var i = 10
let n = 20
if (1 === 1) {
    console.log(n)    // 暂时性死区, 提前使用n变量会导致抛错
    // Uncaught ReferenceError: Cannot access 'n' before initialization
    console.log(i) // => 10
    // 会直接使用全局上下文中的 i 变量
//-----------------------------------------------------------------------
    /*
        EC(BLOCK)
        VO => n : 200,
        没有this,类似箭头函数
        没有arguments对象
        变量提升: --[var 不存在块级作用域]
    */
    var i = 100
    let n = 200
     console.log(i, n) // 100, 200 }
console.log(i, n) // 100, 20 var不存在块级作用域, 所以它直接操作了全局中的i
```

块级作用域

> 总结: 在{}块内, 不应该声明函数

> 在ES6之前, JavaScript中只存在全局上下文和函数执行的私有上下文, 但是在有了let之后, 出现了第三种作用域,也就是块级作用域, 如果在大括号{}(除了 函数与对象)声明了let和const变

> 量/function, 则会在当前的{}中形成一个私有的块级执行上下文.

1. 如果**函数**出现在除了对象和函数的大括号中, 则在变量声明阶段,在全局EC(G)中的**VO** 函数**只声明不定义**.
2. 当执行流到达该**{}**时, 产生一个块级的执行上下文, 并且在此上下文正常进行变量提升, 函数**声明且定义**
3. 当执行流到达函数所在的代码行时, 会将该行数以上, 对函数的所有操作映射给全局的变量foo

```
/*
     1. EC(G)
     2. VO: foo
     4. foo: 0x001 [[scope]]: EC(block),EC(G)*/
{
  /*
      3. VO: foo : 0x001 [[scope]]: EC(block),EC(G)
      5. => 1
      */
  function foo() {}
  foo = 1
}
console.log(foo) // foo: 0x001 [[scope]]: EC(block),EC(G)
```

块级作用域中的性能优化

> 块级上下文的出现就会出现闭包, 闭包会额外占用多余的内存, 应该注意这方面的内存优化

循环事件绑定:

```
let buttons = document.querySelectorALl('button'); // 假设有5个按钮
for(var i = 0 ; i < buttons.length ; i++) {
    buttons[i].onclick = function () {
        alert(i)
    }
}
由于onclick是异步编程(此方法不执行, 后续被点击了才会执行),所以输出的是for循环最终的结果5
当发生点击事件的时候, 由于上下文中的i不是自己私有的而是全局的, 所以函数onclick会输出全局下的i =
5
```

解决方案1: 利用闭包的机制

```
let buttons = document.querySelectorALl('button'); // 假设有5个按钮
for(var i = 0 ; i < buttons.length ; i++) {
    buttons[i].onclick = (function () {
        return function ()  {
             alert(i)
        }
    })(i)
}


// let方案也是闭包方案, 只不过是浏览器自己的处理机制, 比我们手写的闭包性能更好
let buttons = document.querySelectorALl('button'); // 假设有5个按钮
for(let i = 0 ; i < buttons.length ; i++) {
    buttons[i].onclick = function () {
        alert(i)
    }
}
```

解决方案1: 事件委托 性能最优(40%~60%)

```
// @/index.html
    <button index="0">我是第1个按钮</button>
    <button index="1">我是第2个按钮</button>
    <button index="2">我是第3个按钮</button>
    <button index="3">我是第4个按钮</button>
    <button index="4">我是第5个按钮</button>
// 为每个按钮自定义一个index属性
// 基于事件冒泡
// 给父组件绑定事件
document.body.onclick = function (e) {
    let target = e.target
    if (target.tagName === 'BUTTON') {
        alert( target.getAttribute('index') )
    }
}
```

Js的垃圾回收机制

> 堆内存在被占用(地址被引用),是不会被回收的, 只有JS发现该堆不再被引用,浏览器会在空闲的时候释放他.

1. **可达性**: 即当前的所有变量与参数是否可以从全局对象从延神过去,如果不行则会回收该变量或参数.
2. **引用计数**: 当该内存被占用则增1, 每一次取消引用则减1, 当减到0时 则是释放内存

3. **引用检测**: `标记清除` 当被占用时,增加标记, 当移除引用就消除标记, 浏览器空闲时, 会将所有没有被标记的内存释放

基本数据类型与引用数据类型的赋值差异(执行差异)

1. **基本数据类型(值类型)**,为按值操作,即当每创建一个新值,都会为该值开辟一个新空间,存储在当前作用域下(栈内存)

   值类型: 数字 `Number` , 字符串 `String` , 布尔 `Boolean` , `null` , `undefined`

   ```
   数字 Number :
       NaN(not a number),指该值为一个数字类型,但非为有效数字,且与任何值都不相等
       isNaN,检测一个值是否非有效数字,会先使用Number()将值转换为数字再进行检测
   ```

   栈内存: 本身即是JS运行环境