█预处理

●

经典问题

<script>

var a = 1

function xx () {

console.log(a)

var a = 2

}

xx() //undefined

</script>

问题详解

<script>

        console.log(a) //undefined(全局预处理后a的值是undefined)

        var a = 1

        console.log(a) //1 (正常执行a的赋值语句后，a被赋值为1)

        function xx () {

            console.log(a) //函数内部预处理后，此时a的值是undefined

            var a = 2

            console.log(a) //执行a的复制语句后，此时a的值是2

        }

        xx()

        // 执行xx后输出如下

        //undefined

        //2

    </script>

●JS的解析与执行过程

全局预处理阶段

全局执行阶段

函数预处理阶段

函数执行阶段

◎重要说明：JS执行一段代码，首先是全局预处理，然后一行一行全局执行代码。当执行到函数调用时，进入该函数，进行该函数预处理，然后一行一行执行该函数的代码。该函数代码执行完毕后跳出该函数，回到全局代码继续执行。

●全局预处理和执行

◎全局预处理阶段，扫描全局代码的以下内容：

1. 用声明的方式创建的函数
2. 用var定义的变量

◎讲扫描到的函数和方法保存到全局作用域的变量对象中：

1. 用声明的方式创建的函数的值是对该函数的引用
2. 用var定义的变量的值是undefined

例如：

全局代码中有如下内容：

var a = 2;

function b (){alert(1);}

那么预处理扫描后，会把以下内容加到全局变量对象中：

window{

a:undefined,

b:对b函数的引用

}

<script>

alert(g===undefined); //弹出true，因为g是表达式方式声明的函数，在预处理阶段只被认为是var声明的变量，其值被保存为undefined

f(); //会输出1，因为f是用声明的方式创建的函数，被预处理保存在全局变量对象window中了

g(); //会报错，"Uncaught TypeError: undefined is not a function"。因为g是用表达式的方式声明的函数，只是被认为是var声明的变量，所以值被保存为undefined

function f (){

console.log(1);

}

var g = function(){

console.log(2);

}

</script>

◎如果标识符有冲突的处理：

全局预处理会先扫描函数声明，然后再扫描var变量。

处理函数声明有冲突，会覆盖

处理变量有冲突，会忽略（因为值都是undefined）

处理函数和变量的冲突，会以函数为重

<script>

alert(a); //会打印出第二个函数a的代码，因为函数声明冲突会采用覆盖

alert(b); //会打印undefined，因为var变量冲突会忽略，因为值都是undefined

alert(c); //会打印函数c的代码，因为函数声明和变量冲突时，会以函数为重

function a (){

console.log(1);

}

function a (){

console.log(2);

}

var b=3;

var b=5;

function c (){

console.log(3);

}

var c=7;

</script>

◎预处理后的执行

预处理后，开始一行一行执行代码，代码的执行过程中，不断更新全局变量对象中的内容

<script>

//经过预处理,全局变量对象是这样的: window{a:undefined, f:引用, g:undefined}

            alert(a); //打印undefined

        //alert(b); //报错，因为window中没有b

            alert(f); //打印函数f的代码

            alert(g); //打印undefined

            var a =5;

            b = 6;

        //预处理完成后,开始一行一行执行代码,执行到这里时,用执行后的结果更新了window{a:5, f:引用, b:6，g：undefined}

            alert(b); //打印6

            function f(){

                console.log(1);

            }

            var g =function(){

                console.log(2);

            }

        //代码执行到这里后,window中又更新了g的值window{a:5, f:引用, b:6，g：引用}

            alert(g); //打印函数g的引用

</script>

●函数预处理和执行

◎函数预处理阶段

函数每调用一次，就产生一个活动对象，函数执行完毕后活动对象被销毁（闭包除外）

活动对象中包括：

1. 函数的参数（值是undefined，执行时如果传入具体值，则用具体值来更新undefined）
2. 函数内部声明式函数
3. 函数内部var变量
4. arguments

处理冲突与全局处理方式一样

<script>

function f (a, b, c) {

            console.log(a) //1 因为实参是1，并且var冲突忽略，所以忽略了var a = 100, 所以预处理a的值是1

            console.log(b) //函数引用 虽然实参是2， 但是预处理阶段会被函数声明b所覆盖，所以预处理b的值是函数b的引用

            console.log(c) //undefined 调用函数f时没有传第三个参数，所以c的值是undefined，并且var冲突忽略，所以忽略了var c = 200,所以预处理c的值是undefined

            var a = 100

            function b () {

                console.log('b')

            }

            var c = 200

        }

        f(1, 2)

</script>

◎执行函数时，如果函数中不用var定义变量，例如b=2;则执行该行代码后，b被加到全局变量对象中。（没执行时预处理不会添加）

<script>

function f1 () {

            function f2 () {

                a = 99

            }

            f2()

        }

        f1()

        console.log(window.a) //99 在f2函数内声明的a，没有var，则会添加到最外部的变量对象中，即window对象里

</script>

█作用域链及闭包

●每个执行环境（execution context）都有一个与之关联的变量对象（variable object），该执行环境中定义的所有变量和函数都保存在这个变量对象中。

●执行环境有全局执行环境和局部执行环境：

◎全局执行环境被认为是window对象，因此全局变量和函数都是作为window的属性和方法创建的。全局环境直到应用程序退出时才被销毁。

◎局部执行环境中的所有代码执行完毕后，该环境被销毁，里面的所有变量和函数也随之被销毁。

●函数：

◎每个函数都有自己的执行环境。

◎当创建一个函数时（也可以理解成当预处理程序读到一个函数时），其作用域链被保存在内部属性[[Scope]]中。（Scope中保存的是创建该函数时的变量对象。比如在全局作用域下创建一个函数a，Scope保存的就是全局变量对象，即window对象。然后在a里又创建一个函数b，那么b的Scope保存的就是a的Scope加上a的活动对象）

◎当调用这个函数时，首先会创建一个当前函数的执行环境及活动对象（activation object）（我理解活动对象就是当前函数的变量对象）。然后从[[Scope]]中读取作用域链，并将新创建的活动对象加到作用域链的最前端。组成目前执行环境的完整的作用域链。

◎当这个函数执行完毕后，局部活动对象被销毁，内存中只保留原本它的[[Scope]]

案例

<script type="text/javascript">

            var q=1;

            function a (num) {

                for (var i=0; i<10; i++) {

                }

                return function b (nn){

                    return nn + num +i; //在闭包b中可以访问其父级a函数的活动对象,如num和i,注意,访问的是最终结果,比如for循环的i是最终循环完的值10

                };

            }

            var n1=a(3); //a执行完毕后，其活动对象不会被销毁，因为b的作用于链仍然再引用这个活动对象。

            var n2=n1(2);

            alert(n2); //返回15 nn(2)+num(3)+i(10)

            n1=null; //解除对b的引用，已释放内存，销毁a的活动对象。

        </script>

●例子详解：

●作为包裹闭包函数b的a函数，发生了以下事情：

◎当创建a函数时，全局执行环境的全局变量对象被保存在a函数的内部属性[[Scope]]中。

◎调用a时，发生了以下事情：

1. 创建a的执行环境及活动对象（活动对象理解成新创建的执行环境的变量对象）。
2. 复制[[Scope]]中的对象，然后加上新创建的活动对象，构成当前函数的作用域链。

◎a执行完毕后，发生了以下事情：

本应该销毁a的活动对象，内存中仅保留全局作用域[[Scope]]，但是，由于a里面的b函数的作用域链仍然在引用a的这个活动对象，所以a执行完毕后它的活动对象也不会被销毁。

●作为闭包的b函数，发生了以下事情：

◎当创建b函数时，全局执行环境的全局变量对象加上a的活动对象被保存在b函数的内部属性[[Scope]]中。

◎调用b时，发生了以下事情：

1. 创建b的执行环境及活动对象。
2. 复制[[Scope]]中的对象（即，全局变量对象，加上执行a后保留下来的a的活动对象）再加上新创建的b的活动对象，组成完整的b的作用域链。

b执行完毕后，b的活动对象会被销毁。（因为b里面没有引用b的闭包函数了）。

●最后，a,b都执行完了，可是内存里还保留着a的活动对象，所以用n1=null来清除

后记：比如在b里要找一个变量q 那么首先会在b的活动对象中找，如果找不到就顺着作用域链找其父级的词法环境，也就时b的scope中保存的对象。如果b的scope中也没有，那么就报错了。

全局执行环境

全局变量对象

a的执行环境

a的活动对象

b的活动对象

b的执行环境

◎全局变量对象（window）包括：

变量q

函数a

◎a的活动对象包括：

变量q

函数a

参数num

变量i

函数b

◎b的活动对象包括：

变量q

函数a

参数num

变量i

函数b

参数nn

●注意：[[scope]]保存的是引用而不是复制

<script type="text/javascript">

            function f(){

                var num=1;

                //创建g时会将全局变量对象加f的活动对象保存到g的[[scope]]中,但保存的是引用,而不是复制

                function g(){

                    alert(num);

                }

                num++;

                g(); //执行g时，因为f的活动对象num的值发生了变化，而g得scope中是保存的对f活动对象的引用，因此也会相应变化

            }

            f(); //输出2

        </script>

●●闭包总结

●关于闭包的两个问题

◎闭包是在什么时候被创建的？

答案：当一个函数内部预处理执行时，扫描到其内部包含的闭包函数时，则此时就创建闭包了

<script type="text/javascript">

            function f1 () {

                var a = 1

                // 当f1的预处理扫描到f2时，闭包被创建

                function f2 () {

                    console.log(a)

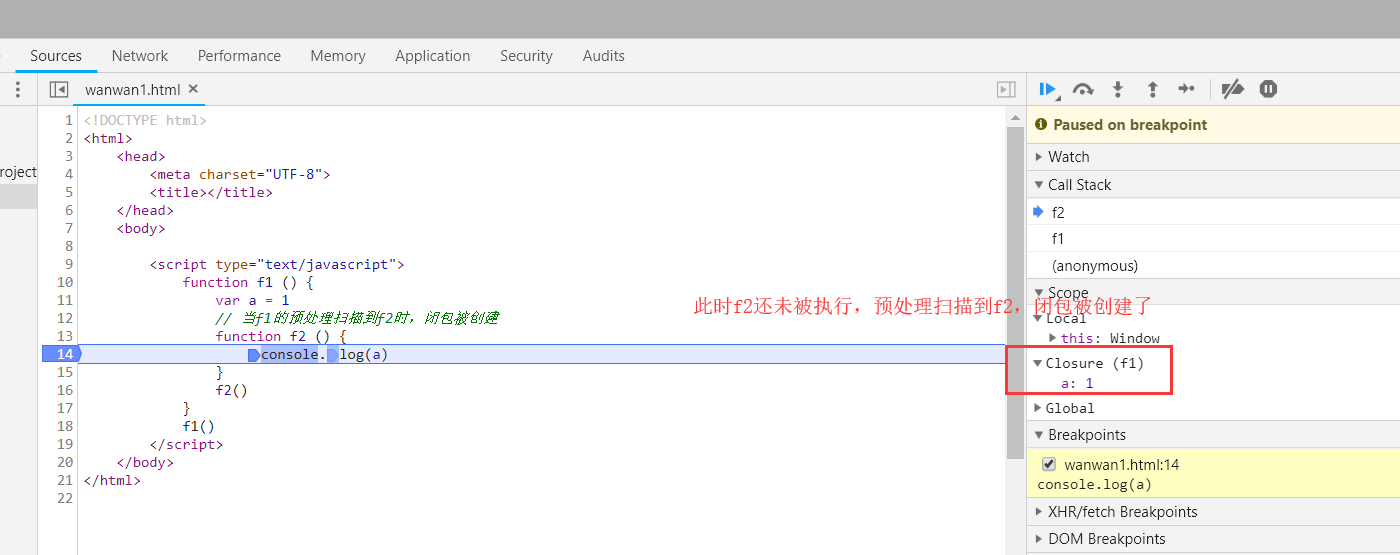
                }

                f2()

            }

            f1()

        </script>



◎问题2 如果一个函数内部的函数，不引用外部函数的变量，那会创建闭包吗？

答案：不会

<script type="text/javascript">

            function f1 () {

                var a = 1

                // f2函数内没有对f1变量对象的引用，不会创建闭包

                function f2 () {

                    console.log('abc')

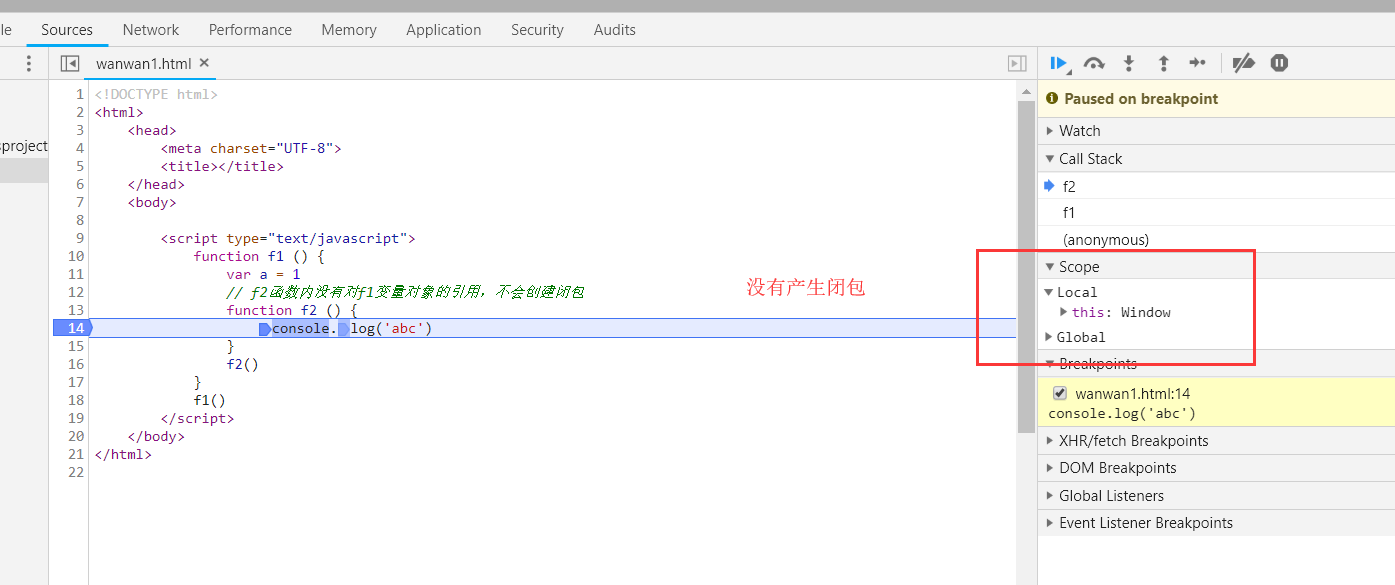
                }

                f2()

            }

            f1()

        </script>



●闭包的作用

减少全局变量

减少传递给函数的参数数量

封装

◎减少全局变量的例子

<script type="text/javascript">

            var a = 0

            function add () {

                a++

                console.log(a)

            }

            add() //1

            add() //2

        </script>

以上的例子声明了一个全局变量a, 如想避免使用全局变量，则可改成下面的例子，使用闭包来避免使用全局变量

<script type="text/javascript">

            // 利用闭包来避免使用全局变量，把a放到闭包的变量对象中

            function f () {

                var a = 0

                return function () {

                    a++

                    console.log(a)

                }

            }

            var result = f()

            result() //1

            result() //2

        </script>

◎减少传递给函数的参数数量的例子

<script>

            // 实现效果，输入一个基数base，在输入一个最大数max

            // 然后要输出：从1累加直到最大数，最后再加上基数

            // 例如基数2 最大数3 输出： 2 + （1+2+3）=8

            function calFactory (base) {

                return function (max) {

                    var total = 0

                    for (var i = 1; i <= max; i++) {

                        total += i

                    }

                    return total + base

                }

            }

            var adder = calFactory(2)

            console.log(adder(3)) //8

            console.log(adder(4)) //12

            // 每执行一次calFactory函数，就会生成一个新的闭包环境

            var adder2 = calFactory(1)

            console.log(adder2(3)) //7

            console.log(adder2(4)) //11

        </script>

◎封装的例子

<script>

            // 建立一个自执行函数，然后将其内部的闭包getM和setM绑到window对象中

            (function () {

                var m = 0

                function getM () {

                    return m

                }

                function setM (val) {

                    m = val

                }

                window.g = getM

                window.f = setM

            })()

            // 在外面就可以直接使用g和f方法了

            f(12)

            console.log(g()) //12

        </script>

●闭包的注意事项

对捕获的变量只是引用，不是复制

父函数每调用一次，会产生不同的闭包

循环中的问题

◎对捕获的变量只是引用，不是复制

<script type="text/javascript">

            function f(){

                var num=1;

                //创建g时会将全局变量对象加f的活动对象保存到g的[[scope]]中,但保存的是引用,而不是复制

                function g(){

                    alert(num);

                }

                num++;

                g(); //执行g时，因为f的活动对象num的值发生了变化，而g得scope中是保存的对f活动对象的引用，因此也会相应变化

            }

            f(); //输出2

        </script>

◎父函数每调用一次，会产生不同的闭包

<script>

            function f () {

                var num = 1

                return function () {

                    num++

                    console.log(num)

                }

            }

            var result1 = f()

            result1() //2

            result1() //3

            // 第二次调用f()会产生新的闭包，与第一次的互不影响

            var result2 = f()

            result2() //2

            result2() //3

        </script>

◎循环中的问题

案例，希望点击不同的div，输出不同的数字1，2，3

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <div id="1">1</div>

    <div id="2">2</div>

    <div id="3">3</div>

    <script>

        // js没有块级作用域，所以这里的i是一个全局变量

        // 当点击div事件调用函数时，i的值是for循环的结果，4

        // 所以无论点击哪个div都会打印同样的4

        for (var i = 1; i <= 3; i++) {

            var ele = document.getElementById(i)

            ele.onclick = function () {

                console.log(i)

            }

        }

    </script>

</body>

</html>

以上代码点击div都只能输出4

改成以下代码，就会实现正常的效果

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <div id="1">1</div>

    <div id="2">2</div>

    <div id="3">3</div>

    <script>

        for (var i = 1; i <= 3; i++) {

            var ele = document.getElementById(i)

            // 每循环一次，由于下面函数是立即被调用的，所以i传进去后，生成闭包，每次循环的i的值就存在了不同的闭包中了

            ele.onclick = (function (id) { //这里id就是传进来的i，写成i也一样

                return function () {

                    console.log(id)

                }

            })(i)

        }

    </script>

</body>

</html>