第八课 执行环境与作用域

学习目录

- **js 预解析**
- **执行环境**
- **什么是作用域**
- **作用域链**

一．js 预解析

在 js 代码执行过程中，在正式执行代码之前，会有一个预解析的过程，这个过程我们也称为预编译或者预处理，在预解析完成之后才是代码的执行阶段。

第一种情况：变量声明和函数表达式

```
console.log(a);//undefined

console.log(b);//Uncaught ReferenceError: b is not defined

var a = 100;
```

从上面这个小例子可以看出，预解析首先对声明的变量 a 进行了默认赋值 undefined，因为 js 在执行 console.log(a)时，已经发现 a 这个变量被声明了，这个声明的过程不是在 js 代码执行的时候，而是在预解析的时候。

所以这里有两个过程：

1. **var a;这个过程发生在 js 预解析的时候**
2. **a = 10;这个过程发生在 js 代码执行的时候**

在预解析的时候 js 知道 a 是 undefined，但不知道 a 是 10，这个过程就是 js 预解析，

这里的变量声明被赋值 undefined 就是 js 预解析中的一种体现。

跟变量声明类似的是函数表达式，注意这里说的是函数表达式，函数表达式的预解析跟变量声明预解析的过程一样。

```
console.log(fn1);//undefined
var fn1 = function(){};
console.log(fn1);//ƒ (){}
```

第二种情况：this 赋值

当我们在 js 代码中的任何位置去输出 this，都会得到一个值，在全局是 window，在函数中可能是某个调用函数的对象，这里的 this 被赋值的情况也发生在 js 预解析的时候。

```
console.log(this);//Window
```

第三种情况：函数声明

在 js 预解析遇到函数声明时，会把函数全部赋值。

```
console.log(fn2);//ƒ fn2(){}
function fn2(){};
console.log(fn2);//ƒ fn2(){}
```

这个预解析的过程主要包括以下操作内容：

1.声明变量、声明函数表达式，默认赋值为 undefined，js 会首先把这些东西预先存储到内存中的某个地方。

2.对 this 赋值，无论在任何情况下，this 都是有值的。

3.对函数声明赋值，预解析阶段对于函数声明是直接赋值，函数中的参数也会被赋值。

这上面说的三种预解析要做的操作，都会生成执行环境，这里默认是全局的执行环境，在函数体内部还有函数体内部的准备工作。

函数体内部的预解析过程

之前提到的都是全局环境下的 js 预解析过程，在函数体内部还有对应的预解析过程，在函数体的代码执行之前，arguments 变量集合和函数的具体参数都已经被赋值。

```
var d = 9;
function fn1(a,b){
    console.log(c);//undefined
    console.log(fn2);//fn2 函数声明赋值
    console.log(fn3);//undefined
    console.log(d);//9
    console.log(this);//window //{x: 1, y: 2}
    var c = 3;
    var fn3 = function(){
        console.log(fn3);
    }
    function fn2(){
        console.log('fn2');
    }
    console.log(arguments);
```

```
    console.log(a + b);
}
fn1(1,2);//Arguments(2) [1, 2] //3
fn1(100,200);//Arguments(2) [100, 200] //300
fn1.call({x:1,y:2});//{x: 1, y: 2}
```

从上面的函数可以看出来，当一个函数被声明到被执行之前，内部预解析的过程主要分为以下几个方面：

1. **先会对 arguments 变量集合和函数的具体参数进行复制**
2. **假如这个函数引用了外部的一个变量，这个变量也会被赋值，这个外部变量一般称为自由变量，是通过作用域链来取到的。**
3. **如果在函数体内部定义了一个新的变量或者函数表达式，这个新的变量和函数表达式也会先被赋值成 undefined，如果是定义了一个新的函数声明，这个函数声明也会被直接赋值，这种函数体内部新定义变量、函数表达式、函数声明的预解析过程与全局预解析过程类似。**
4. **另外 this 在函数定义的时候不会被赋值，因为 this 的取值是在函数被调用的时候根据调用对象的不同才能确定的。**

函数被调用一次，由于传递的参数不同，就会产生一次不同的执行环境，因此函数的执行环境是动态的。

二．执行环境

在 js 的执行环境中可以理解成分为全局执行环境和函数执行环境。

1. 执行全局环境下的代码时，会先产生一个全局的执行环境。当在全局代码中调用了

一个函数 fn1，那么执行的上下文就会进入这个函数 fn1 中，这个函数 fn1 就会创建一个新的执行环境，同时这个新的执行环境会被添加到一个叫执行栈的东西的顶部。

2.当这个函数 fn1 调用完成之后，执行栈就会把这个执行环境弹出，也就是说这个执行环境会被销毁，保存在这个函数 fn1 其中的变量、其他函数等也会被销毁。然后程序会重新回到全局执行环境下继续执行。

3.假如在这个执行函数 fn1 中调用了其他函数 fn2 ,执行上下文也会进入函数 fn2 内部，并在 fn2 中创建一个新的执行环境，把 fn2 这个函数也添加到执行栈的顶部执行。

4.因此 js 在执行的时候始终只有一个执行环境在执行，只不过在函数调用的时候会把执行上下文给到执行栈顶部对应的函数中，从而执行当前栈顶的执行环境对应的函数。

5.当所有的函数调用产生的执行环境都执行完毕之后，他们的执行环境都会被销毁，执行权力重新给到全局执行环境直到整个程序执行完毕退出。

执行环境

三．什么是作用域

作用域是可以从来访问变量的集合，简单理解就是定义一套规则，这套规则专门用来解释如何 js 程序中如何存储和使用变量。变量可以是值类型，也可以是引用类型，比如函数与对象。

作用域可以理解为全局作用域和局部作用域。

全局作用域

1. 只要变量在函数外定义，这个变量就是全局变量。
2. 全局变量存在于全局作用域中，全局作用域专门用来提供全局变量的存储和访问。
3. 全局作用域中的全局变量在整个 js 脚本中都可以被访问使用。
4. 全局变量在全局作用域被关闭时，也就是页面关闭后被销毁。

局部作用域

在 js 中只有函数可以创建的作用域，js 中的作用域和其他语言有所不同，花括号不能创建独立的作用域，也就是说 js 中没用块级作用域 的概念。

ps：目前我们学习的 ECMAScript262 和 ECMAScript5 中没有块级作用域的概念，花括号中的代码块不是独立的作用域，只有函数可以创新独立的作用域。

```
if(true){
    //这里不是作用域
}else{
    //这里不是作用域
}
```

1. 我们在函数体内声明的变量就是这个函数的局部变量 ,这个函数就是一个局部作用域了。
2. 局部作用域存在一种上下级关系 ,局部作用域除了提供当前局部作用域的变量的存储和访问，还能够提供上一级作用域中变量的访问。
3. 局部作用域中上下级关系是由这个函数是在哪一级作用域下创建来决定的。

4. 局部作用域中的变量名称可以重复，他们之间不有冲突。在各自的作用域下，用各自的同名局部变量。

5. 作用域最大的用处就是隔离变量，每个函数中都有自己独立的变量，同时又可以使用上一级的变量，这里需要作用域链的概念，我们之后详细介绍。

```
function fn1(){
    var a = 100;
    function fn2(){
        var a = 1000;
        var b = 200;
        function fn3(){
            var a = 10000;
            var b = 2000;
            var c = 300;
        }
    }
}
```

四．作用域链

函数在一开始创建之初，局部作用域就是已经确定得了，在不同的作用域中访问某个变量的时候，通常会有一个取值的过程。

1. 一般情况下，局部作用域中的变量取值会先在创建这个变量的函数局部作用域中来取值。

2. 如果在当前函数的局部作用域中没有找到对应的变量值 ,就会向上一级局部作用域去查找，如果查找到了就拿来使用。

3. 如果上一级局部作用域中没有对应的变量，就会向更上一级的局部作用域中去查找，直到查找到全局作用域下，如果有就拿来使用，如果没有就会报错。

```
var c = 300;
function fn1(){
    var a = 100;
    function fn2(){
        var b = 200;
        console.log(a + b + c);//600
        console.log(d);//Uncaught ReferenceError: d is not defined
    }
    fn2();
}
fn1();
```

这个取值的过程就是通过作用域链来完成的 ,作用域链的上下级关系也是在函数一开始创建的时候就确定的。

执行环境与作用域的关系

某个函数的执行环境是在这个函数被调用的时候才创建的。

某个函数只要创建了，也就等于创建了一个局部作用域。

1. 对于一个函数而言，他创建的这个局部作用域有可能存在过多个执行环境。比如这个函数在不同的作用域下被不同的对象所调用。

```
function fn1(x,y){
    console.log(x + y);
    console.log('被' + this + '调用');
}
fn1(100,200);//被[object Window]调用

var obj = {};
fn1.call(obj,300,400);//被[object Object]调用

function fn2(){
    var a = 1;
    var b = 2;
    fn1(a,b);//被[object Window]调用
}
fn2();
```

2. 当这个函数从来没有被调用过的时候，这个函数形成的局部作用域也就不会有执行环境。

3. 当函数执行完毕之后，这个调用函数的执行环境就被销毁了，因此这个函数产生的作用域下面的执行环境也只能说有过，但是执行过，要等待下一次执行。

4. 还有可能一个函数的作用域中有多个执行环境，这个情况就是 js 中闭包的概念。

```
function fn1(a){
    var b = 200;
    return function(){
        console.log(a + b);
    }
}
var f1 = fn1(100);//这里执行完毕之后 fn1(100)产生的执行环境在执行完毕之后不会被销毁，因为 fn1 内部有闭包的引用
var f2 = fn1(300);//当执行到 fn1(300)的时候，fn1(300)也会产生执行环境，此时 fn1 这个函数形成的作用就会包含两个执行环境
f1();//300
f2();//500
```

谢谢观看！

我是星星课堂老师：周小周