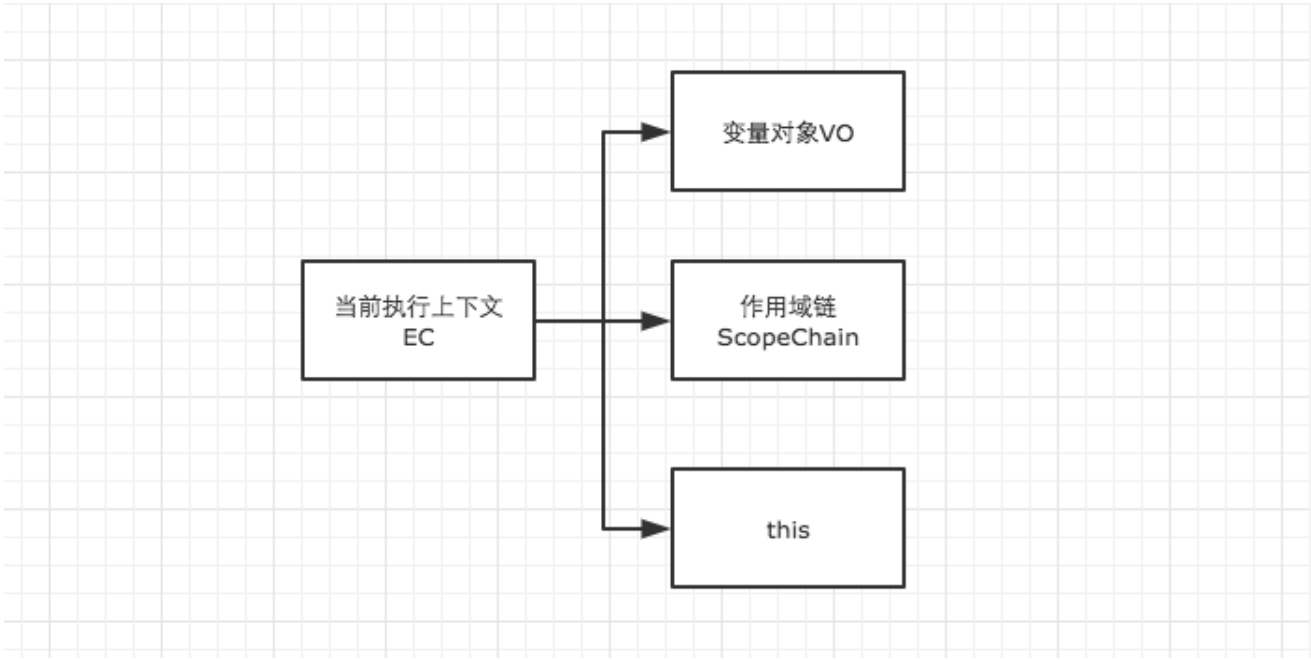
变量对象详解

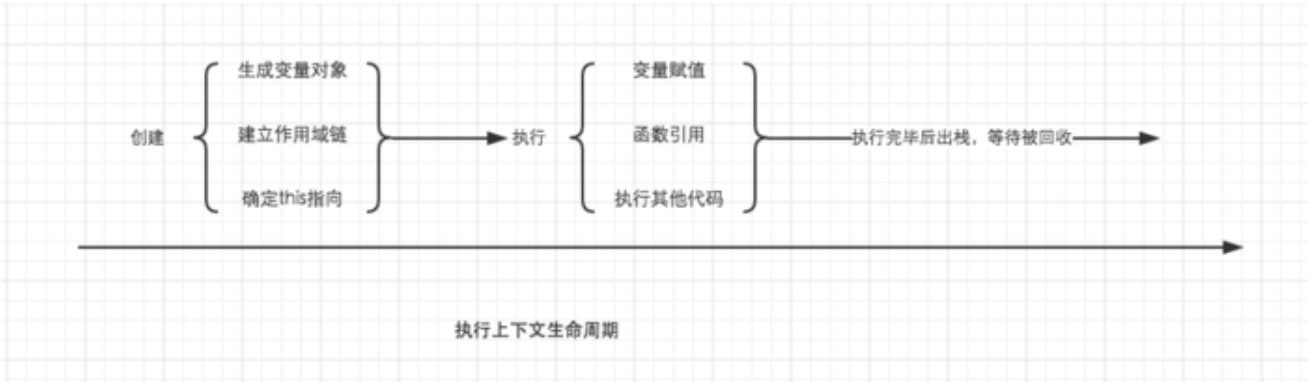


在`javaScript`中，我们肯定不可避免的需要声明变量和函数，可是`JS`解析器是如何找到这些变量的呢？我们还得对执行上下文有一个进一步的了解。

我们已经知道，当调用一个函数时（激活），一个新的执行上下文就会被创建。而一个执行上下文的生命周期可以分为两个阶段。

- **创建阶段[当函数被调用, 但内部的代码还没开始执行]**

  在这个阶段中，执行上下文会分别创建变量对象，建立作用域链，以及确定`this`的指向。
- **代码执行阶段** 创建完成之后，就会开始执行代码，这个时候，会完成变量赋值，函数引用，以及执行其他代码。



这里我们就可以看出详细了解执行上下文极为重要，因为其中涉及到了变量对象，作用域链，`this`等极为重要的概念。

变量对象（Variable Object）

变量对象的创建，依次创建以下属性**（只适用于变量对象创建过程）**。

1. 函数的形参

- 由名称和对应值组成的一个变量对象的属性被创建
- 没有实参，属性值设为 undefined

2. 函数声明
   - 由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建
   - 如果变量对象已经存在相同名称的属性，则完全替换这个属性
3. 变量声明
   - 由名称和对应值（undefined）组成一个变量对象的属性被创建；
   - 如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性

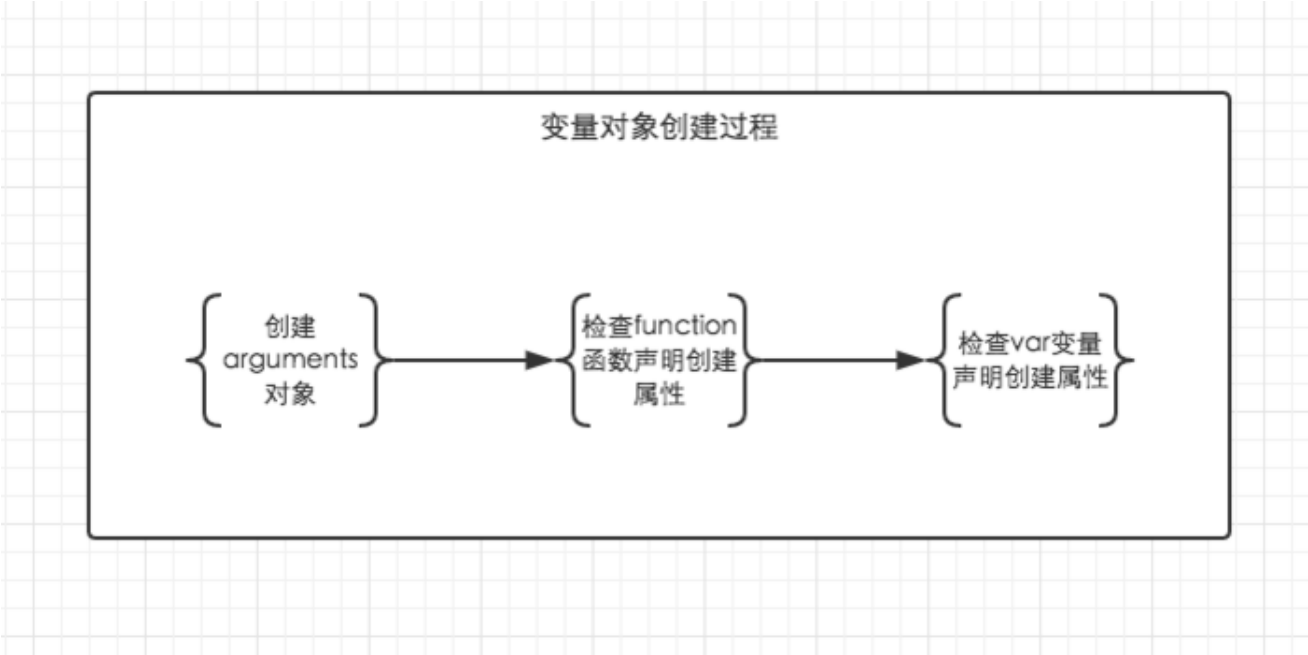
阅读第三点的时候会因为下面的这样场景对于 **不会干扰** 一词产生疑问。既然变量声明的`foo` **不会干扰** 函数声明的`foo`，可是为什么最后`foo`的输出结果仍然是被覆盖了？

```
function foo() { console.log('function foo') }
var foo = 20;
console.log(foo); // 20
```

因为上面的三条规则仅仅适用于变量对象的创建过程。也就是执行上下文的创建过程。而`foo = 20`是在执行上下文的执行过程中运行的，输出结果自然会是20。对比下例。

```
console.log(foo); // function foo
function foo() { console.log('function foo') }
var foo = 20;
```

```
// 上栗的执行顺序为
// 首先将所有函数声明放入变量对象中
function foo() { console.log('function foo') }
// 其次将所有变量声明放入变量对象中，但是因为foo已经存在同名函数，因此此时会跳过undefined的赋值
// var foo = undefined;
// 然后开始执行阶段代码的执行
console.log(foo); // function foo
foo = 20;
```



根据这个规则，理解变量提升就变得十分简单了。

```
function test() {
    console.log(a);
    console.log(foo());

    var a = 1;
    function foo() {
        return 2;
    }
}
test();
```

从`test()`的执行上下文开始理解。全局作用域中运行`test()`时，`test()`的执行上下文开始创建。为了便于理解，用如下的形式来表示 。

```
// 创建过程
testEC = {
    VO:{// 变量对象
        arguments: {...},
        foo: <foo reference>  // 表示foo的地址引用
        a: undefined
    },
    scopeChain: {},
    this:Window
}
// 因为本文暂时不详细解释作用域链，所以把变量对象专门提出来说明
```

未进入执行阶段之前，变量对象中的属性都不能访问。但是进入执行阶段之后，变量对象转变为了活动对象，里面的属性都能被访问了，然后开始进行执行阶段的操作。 变量对象和活动对象有什么区别 ？他们其实都是同一个对象，只是处于执行上下文的不同生命周期。不过只有处于函数调用栈栈顶的执行上下文中的变量对象，才会变成活动对象。

```
// 执行阶段
VO ->  AO   // Active Object

testEC = {
    AO:{// 变量对象
        arguments: {...},
        foo: <foo reference>,
        a: 1
    },
    scopeChain: {},
    this:Window
}
```

因此，上面的例子，执行顺序就变成了这样

```
function test() {
    function foo() {
        return 2;
    }
    var a;
    console.log(a);
    console.log(foo());
    a = 1;
}
test();
```

再来一个例子，巩固一下我们的理解。

```
function test() {
    console.log(foo);
    console.log(bar);

    var foo = 'Hello';
    console.log(foo);
    var bar = function () {
        return 'world';
    }

    function foo() {
        return 'hello';
    }
}

test();
```

```
// 创建阶段
testEC = {
    VO:{
        arguments: {...},
        foo: <foo reference>,
        bar: undefined,
        this:Window
    },
    scopeChain: {},
    this:Window
}
// 这里有一个需要注意的地方，因为var声明的变量当遇到同名的属性时，会跳过而不会覆盖
```

```
// 执行阶段
VO -> AO
testEC = {
    AO:{
        arguments: {...},
        foo: 'Hello',
        bar: <bar reference>,
        this: Window
    }
    scopeChain: {},
    this:Window
}
```

全局上下文的变量对象

以浏览器中为例，全局对象为`window`。 全局上下文有一个特殊的地方，它的变量对象，就是`window`对象。而这个特殊，在`this`指向上也同样适用，`this`也是指向`window`。

```
// 全局上下文
windowEC = {
    VO: Window,
    scopeChain: {},
    this: Window
}
```

全局上下文的生命周期，与程序的生命周期一致，只要程序运行不结束，比如关掉浏览器窗口，全局上下文就会一直存在。其他所有的上下文环境，都能直接访问全局上下文的属性 。

关于javaScript中的形参与实参

```
function one(a,b,c) {
    return one.length;
}
function two(a,b,c,d,e,f,g){
    return arguments.length;
}
console.log(one()); //3 形参
console.log(two()); //0 实参 实参是调用的时候判断是否传参才能确定的
```

这里我们在`one`函数里面返回了`one.length`，在`two`函数里面返回`arguments.length`。可能你已经发现了，输出的`one()`返回了`one.length`就是**形参**的数量，而`argument.length`就是**实参**的数量。