Dmitry Soshnikov in ECMAScript | 2010-03-21

ECMA-262-3 in detail. Chapter 4. Scope chain.

Read this article in: Russian, Chinese (version 1, version 2), Korean, French.

Introduction

As we already know from the second chapter concerning the *variable object*, the data of an execution context (variables, function declarations, and formal parameters of functions) are stored as properties of the variables object.

Also, we know that the variable object is created and filled with initial values every time on entering the context, and that its updating occurs at code execution phase.

This chapter is devoted one more detail directly related with execution contexts; this time, we will mention a topic of a *scope chain*.

Definition

If to describe briefly and showing the main point, a scope chain is mostly related with inner functions.

As we know, ECMAScript allows creation of inner functions and we can even return these functions from parent functions.

```
var x = 10;

function foo() {

  var y = 20;

  function bar() {
    alert(x + y);
  }

  return bar;

}

foo()(); // 30
```

Thus, is known that *every context has its own variables object*: for the global context it is *global object* itself, for functions it is the *activation object*.

And the scope chain is exactly this *list of all (parent) variable objects* for the inner contexts. This chain is used for variables lookup. I.e. in the example above, scope chain of “bar” context includes AO(bar), AO(foo) and VO(global).

But, let’s examine this topic in detail.

Let’s begin with the definition and further will discuss deeper on examples.

> *Scope chain* is related with an execution context a *chain of variable objects* which is used for variables lookup at *identifier resolution*.

The scope chain of a function context is created at function *call* and consists of the *activation object* and the internal *[[Scope]]* property of this function. We will discuss the [[Scope]] property of a function in detail below.

Schematically in the context:

```
activeExecutionContext = {
    VO: {...}, // or AO
    this: thisValue,
    Scope: [ // Scope chain
      // list of all variable objects
```

```
        // for identifiers lookup
    ]
};
```

where *Scope* by definition is:

```
Scope = AO + [[Scope]]
```

For our examples we can represent *Scope*, and *[[Scope]]* as normal ECMAScript arrays:

```
var Scope = [VO1, VO2, ..., VOn]; // scope chain
```

The alternative structure view can be represented as a hierarchical object chain with the reference to the parent scope (to the parent variable object) on every link of the chain. For this view corresponds *__parent__* concept of some implementations which we discussed in the second chapter devoted variable object:

```
var VO1 = {__parent__: null, ... other data}; -->
var VO2 = {__parent__: VO1, ... other data}; -->
// etc.
```

But to represent a scope chain using an array is more convenient, so we will use this approach. Besides, the specification statements abstractly itself (see 10.1.4) that “a scope chain is a *list* of objects”, regardless that on the implementation level can be used the approach with the hierarchical chain involving the *__parent__* feature. And the array abstract representation is a good candidate for the *list* concept.

The combination AO + [[Scope]] and also process of *identifier resolution*, which we will discuss below, are related with the life cycle of functions.

Function life cycle

Function life cycle is divided into a stage of *creation* and a stage of *activation (call)*. Let’s consider them in detail.

Function creation

As is known, function declarations are put into variable/activation object (VO/AO) on entering the context stage. Let's see on the example a variable and a function declaration in the global context (where variable object is the global object itself, we remember, yes?):

```
var x = 10;

function foo() {
  var y = 20;
  alert(x + y);
}

foo(); // 30
```

At function activation, we see correct (and expected) result – 30. However, there is one very important feature.

Before this moment we spoke only about variable object of the current context. Here we see that "y" variable is defined in function "foo" (which means it is in the AO of "foo" context), but variable "x" is not defined in context of "foo" and accordingly is not added into the AO of "foo". At first glance "x" variable does not exist at all for function "foo"; but as we will see below — only "at first glance". We see that the activation object of "foo" context contains only one property — property "y":

```
fooContext.AO = {
  y: undefined // undefined – on entering the context, 20 – at activation
};
```

How does function "foo" have access to "x" variable? It is logical to assume that function should have access to the variable object of a higher context. In effect, it is exactly so and, physically this mechanism is implemented via the internal *[[Scope]]* property of a function.

> *[[Scope]]* is a hierarchical chain of all *parent* variable objects, which are *above* the current function context; the chain is saved to the function at its *creation*.

Notice the important point — [[Scope]] is saved at function *creation* — statically (invariably), once and forever — until function destruction. I.e. function *can be never called*, but [[Scope]] property is *already written and stored* in function object.

Another moment which should be considered is that *[[Scope]]* in contrast with *Scope (Scope chain)* is the property of a *function* instead of a *context*. Considering the above example, [[Scope]] of the “foo” function is the following:

```
foo.[[Scope]] = [
  globalContext.VO // === Global
];
```

And further, by a function call as we know, there is an entering a function context where the *activation object* is created and *this* value and *Scope (Scope chain)* are determined. Let us consider this moment in detail.

Function activation

As it has been said in definition, on entering the context and after creation of AO/VO, *Scope* property of the context (which is a scope chain for variables lookup) is defined as follows:

```
Scope = AO|VO + [[Scope]]
```

High light here is that the activation object is the *first* element of the *Scope* array, i.e. added to the *front of scope chain*:

```
Scope = [AO].concat([[Scope]]);
```

This feature is very important for the process of *identifier resolution*.

> *Identifier resolution* is a process of determination to which variable object in scope chain the variable (or the function declaration) belongs.

On return from this algorithm we have always a value of type *Reference*, which *base* component is the corresponding variable object (or *null* if variable is not found), and a *property name* component is the name of the looked up (resolved) identifier. In detail *Reference* type is discussed in the Chapter 3. This.

Process of identifier resolution includes lookup of the property corresponding to the name of the variable, i.e. there is a consecutive examination of variable objects in the scope chain, starting from the deepest context and up to the top of the scope chain.

Thus, local variables of a context at lookup have higher priority than variables from parent contexts, and in case of two variables with the same name but from different contexts, the first is found the variable of deeper context.

Let's a little complicate an example described above and add additional inner level:

```
var x = 10;

function foo() {

  var y = 20;

  function bar() {
    var z = 30;
    alert(x +  y + z);
  }

  bar();
}

foo(); // 60
```

For which we have the following *variable/activation objects*, *[[Scope]]* properties of functions and *scope chains* of contexts:

Variable object of the global context is:

```
globalContext.VO === Global = {
  x: 10
  foo: <reference to function>
};
```

At `foo` creation, the `[[Scope]]` property of `foo` is:

```
foo.[[Scope]] = [
  globalContext.VO
];
```

At `foo` function call, the *activation object* of `foo` context is:

```
fooContext.AO = {
  y: 20,
  bar: <reference to function>
};
```

And the *scope chain* of `foo` context is:

```
fooContext.Scope = fooContext.AO + foo.[[Scope]] // i.e.:

fooContext.Scope = [
  fooContext.AO,
  globalContext.VO
```

```
];
```

At creation of inner `bar` function its `[[Scope]]` is:

```
bar.[[Scope]] = [
  fooContext.AO,
  globalContext.VO
];
```

At `bar` function call, the *activation object* of `bar` context is:

```
barContext.AO = {
  z: 30
};
```

And the *scope chain* of `bar` context is:

```
barContext.Scope = barContext.AO + bar.[[Scope]] // i.e.:

barContext.Scope = [
  barContext.AO,
  fooContext.AO,
  globalContext.VO
];
```

Identifier resolution for `x`, `y` and `z` names:

```
- "x"
-- barContext.AO // not found
-- fooContext.AO // not found
-- globalContext.VO // found - 10
```

```
- "y"
-- barContext.AO // not found
-- fooContext.AO // found - 20
```

```
- "z"
-- barContext.AO // found - 30
```

Scope features

Let’s consider some important features related with Scope chain and `[[Scope]]` property of functions.

Closures

Closures in ECMAScript are directly related with the `[[Scope]]` property of functions. As it has been noted, `[[Scope]]` is saved at function creation and exists until the function object is destroyed. Actually, a *closure* is exactly a *combination of a function code and its* `[[Scope]]` *property*. Thus, `[[Scope]]` contains that *lexical environment* (the parent variable object) in which function is *created*. Variables from higher contexts at the further function activation will be searched in this lexical (statically saved at creation) chain of variable objects.

Examples:

```
var x = 10;

function foo() {
  alert(x);
}

(function () {
  var x = 20;
  foo(); // 10, but not 20
})();
```

We see that `x` variable is found in the `[[Scope]]` of `foo` function, i.e. for variables lookup the lexical *(closured)* chain defined at the moment of function *creation*, but not the *dynamic* chain of the *call* (at which value of `x` variable would be resolved to `20`) is used.

Another (classical) example of closure:

```
function foo() {

  var x = 10;
  var y = 20;

  return function () {
    alert([x, y]);
  };

}

var x = 30;

var bar = foo(); // anonymous function is returned

bar(); // [10, 20]
```

Again we see that for the identifier resolution the lexical scope chain defined at function creation is used — the variable `x` is resolved to `10`, but not to `30`. Moreover, this example clearly shows that `[[Scope]]` of a function (in this case of the anonymous function returned from function `foo`) continues to exist *even after the context in which a function is created is already finished*.

In more details about the theory of closures and their implementation in ECMAScript read in the Chapter 6. Closures.

[[Scope]] of functions created via Function constructor

In the examples above we see that function at creation gets the `[[Scope]]` property and via this property it accesses variables of all parent contexts. However, in this rule there is one important exception, and it concerns functions created via the *Function* constructor.

```
1  var x = 10;
2
3  function foo() {
4
5    var y = 20;
6
7    function barFD() { // FunctionDeclaration
8      alert(x);
9      alert(y);
10   }
11
12   var barFE = function () { // FunctionExpression
13     alert(x);
14     alert(y);
15   };
16
17   var barFn = Function('alert(x); alert(y);');
18
19   barFD(); // 10, 20
20   barFE(); // 10, 20
21   barFn(); // 10, "y" is not defined
22
23 }
24
25 foo();
```

As we see, for `barFn` function which is created via the `Function` constructor the variable `y` is not accessible. But it does not mean that function `barFn` has no internal `[[Scope]]` property (else it would not have access to the variable `x`). And the matter is that `[[Scope]]` property of functions created via the `Function` constructor contains *always only the global object*. Consider it since, for example, to create closure of upper contexts, except global, via such function is not possible.

Two-dimensional Scope chain lookup

Also, an important point at lookup in scope chain is that prototypes (if they are) of variable objects can be also considered — because of prototypical nature of ECMAScript: if property is not found directly in the object, its lookup proceeds in

the *prototype chain*. I.e. some kind of 2D-lookup of the chain: (1) on scope chain links, (2) and on every of scope chain link — deep into on prototype chain links. We can observe this effect if define property in `Object.prototype`:

```
function foo() {
  alert(x);
}

Object.prototype.x = 10;

foo(); // 10
```

Activation objects do not have prototypes what we can see in the following example:

```
function foo() {

  var x = 20;

  function bar() {
    alert(x);
  }

  bar();
}

Object.prototype.x = 10;

foo(); // 20
```

If activation object of `bar` function context would have a prototype, then property `x` should be resolved in `Object.prototype` because it is not resolved directly in AO. But in the first example above, traversing the scope chain in identifier resolution, we reach the global object which (in some implementation but not in all) is inherited from `Object.prototype` and, accordingly, `x` is resolved to `10`.

The similar situation can be observed in some versions of SpiderMokey with *named function expressions (abbreviated form is NFE)*, where special object which stores the optional name of function-expression is inherited from `Object.prototype`, and also in some versions of *Blackberry* implementation where *activation objects* are inherited from *Object.prototype*. But more detailed this features are discussed in Chapter 5. Functions.

Scope chain of the global and eval contexts

Here is not so much interesting, but it is necessary to note. The scope chain of the global context contains *only global object*. The context with code type “eval” has

the same scope chain as a *calling context*.

```
globalContext.Scope = [
  Global
];

evalContext.Scope === callingContext.Scope;
```

Affecting on Scope chain during code execution

In ECMAScript there are two statements which can modify scope chain at runtime code execution phase. These are *with* statement and *catch* clause. Both of them add to the front of scope chain the object required for lookup identifiers appearing within these statements. I.e., if one of these case takes place, scope chain is schematically modified as follows:

```
Scope = withObject|catchObject + AO|VO + [[Scope]]
```

The statement *with* in this case adds the object which is its parameter (and thus properties of this object become accessible without prefix):

```
var foo = {x: 10, y: 20};

with (foo) {
  alert(x); // 10
  alert(y); // 20
}
```

Scope chain modification:

```
Scope = foo + AO|VO + [[Scope]]
```

Let us show once again that the identifier is resolved in the object added by the *with* statement to the front of scope chain:

```
var x = 10, y = 10;

with ({x: 20}) {

  var x = 30, y = 30;

  alert(x); // 30
  alert(y); // 30
}

alert(x); // 10
alert(y); // 30
```

What happened here? On entering the context phase, “x” and “y” identifiers have been added into the variable object. Further, already at runtime code executions stage, following modifications have been made:

- x = 10, y = 10;
- the object *{x: 20}* is added to the front of scope chain;
- the met *var* statement inside *with*, of course, created nothing, because all variables have been parsed and added on entering the context stage;
- there is only modification of “x” value, and exactly that “x” which is resolved now in the object added to the front of scope chain at second step; value of this “x” was 20, and became 30;
- also there is modification of “y” which is resolved in variable object above; accordingly, was 10, became 30;
- further, after *with* statement is finished, its special objects is removed from the scope chain (and the changed value “x” – 30 is removed also with that object), i.e. scope chain structure is restored to the previous state which was before *with* statement augmentation;
- as we see in last two alerts: the value of “x” in current variable object remains the same and the value of “y” is equal now to 30 and has been changed at *with* statement work.

Also, a `catch` clause in order to have access to the parameter-exception creates an intermediate scope object with the only property — exception parameter name, and places this object in front of the scope chain. Schematically it looks so:

```
try {
  ...
} catch (ex) {
  alert(ex);
}
```

Scope chain modification:

```
var catchObject = {
  ex: <exception object>
};

Scope = catchObject + AO|VO + [[Scope]]
```

After the work of *catch* clause is finished, scope chain is also restored to the previous state.

Conclusion

At this stage, we have considerate almost all general concepts concerning execution contexts and related with them details. Further, according to plan, — detailed analysis of function objects: types of functions (FunctionDeclaration, FunctionExpression) and *closures*. By the way, closures are directly related with the [[Scope]] property discussed in this article, but about it is in appropriate chapter. I will be glad to answer your questions in comments.

Additional literature

- 8.6.2 – [[Scope]]
- 10.1.4 – Scope Chain and Identifier Resolution

Translated by: Dmitry A. Soshnikov.
Published on: 2010-03-21

Originally written by: Dmitry A. Soshnikov [ru, read »]
Originally published on: 2009-07-01

Dmitry Soshnikov

Software engineer interested in learning and education. Sometimes blog on topics of programming languages theory, compilers, and ECMAScript.

Published
2010-03-21

Write a Comment

69 COMMENTS

← Older Comments

Dmitry Soshnikov
2016-11-01

@**Oleg**, it is a reference.

In the recent version of the spec VO/AO concepts are combined by the single concept of environments.

```
1 [[Scope]] -> parent Env -> grandparent Env -> ... -> null
```

Oleg
2016-11-02

It means that if the outer function returns an inner function and does not exist anymore, its part, namely VO, is alive. Isn't it?

Dmitry Soshnikov
2016-11-02

@**Oleg**, yes, that's correct. That's because the VO (or function's activation environment/object) is still referenced from the `[[Scope]]` property of the returning to the outside function, so garbage collector doesn't remove it, even if the outer function itself is finished.

Oleg
2016-11-03

Thank you, Dmitry.

Joiner
2016-11-21

Dmitry ,This is a very good article.

> *“by a function call as we know, there is an entering a function context where the activation object is created and this value and Scope (Scope chain) are determined.”*

and

> *” function can be never called, but [[Scope]] property is already written and stored in function object.”*

I have a question:
if a function do not be called, so its AO will not be created.
The inner function’s [[Scope]] = AO of parent + AO of grandpa + …
but here his parent function do not be called, in other words, parent’s AO
has not yet been created, there is no parent’s AO exist.
How [[Scope]] property of its inner function may look like?

Dmitry Soshnikov
2016-11-21

@**Joiner**

> *but here his parent function do not be called, in other words, parent’s AO has not yet been created, there is no parent’s AO exist.*
> *How [[Scope]] property of its inner function may look like?*

So if an outer function is not called (its AO/environment is not created), so the inner function *is not created* as well.

The `[[Scope]]` is recorded *only* when a function *is created*, i.e. the outer function should be called.

```
var x = 10;

// "foo" is created, its [[Scope]] contains
// global object {x: 10, foo: function}
function foo() {
  var y = 20;
  function bar() {}
}

// "foo" is called, its AO/environment
// is created: {y: 20, bar: function}
// At this point, bar's [[Scope]] is
// created and is AO(foo) -> global
foo();
```

You see that the `[[Scope]]` of `bar` is created when `foo` is called. But then `bar` is not called itself.

Joiner

2016-11-22

I get it.
Thank you very much Dmitry!
The Additional literature link looks like no content, any suggestion?

max

2016-12-25

can you explain the difference in behavior (value of ‘i’ in the `setTimeout` `console.log` between this

```
for (let i = 0; i > 10; i++) {
      console.log(i);
      setTimeout(function () {
          console.log(`The number is ${i}`);
      }, 1000);
  }
```

and this

```
for (var i = 0; i > 10; i++) {
      console.log(i);
      setTimeout(function () {
          console.log(`The number is ${i}`);
      }, 1000);
```

```
6     }
```

Dmitry Soshnikov

2016-12-27

@**max**, `let` creates *block-scoped* variables, and `var` creates *function-scoped* variables (i.e. such variables are hoisted to the nearest enclosing scope — function or global).

The second example is the same as:

```
var i;

for (i = 0; i > 10; i++) { ... }
```

So `i` is shared between all functions, and has last assigned value at execution.

Bruno Lesieur

2017-03-31

Hi Dmitry,

You said this :

> Variables from higher contexts at the further function activation will be searched in this lexical (statically saved at creation) chain of variable objects.

But, lexical chain (that is scope chain, right?) is not a part of variable objects because you have said this:

```
executionContext = {
    VO: {}, // <= this is the variable object
    Scope: [] //  Variables from higher contexts at the further function a
};
```

This is true or what I have not correctly understood?

Thx by advance!

Dmitry Soshnikov
2017-04-04

@Bruno Lesieur

When a context is create, its `Scope` property is set to `VO + [[Scope]]`, that is variables are searched in the own VO, and then in the captured `[[Scope]]`.

Since ES5 AO/VO are combined to one model of lexical environments. When a function is executed, a new environment is created (AO), and parent environment is set to the captured one (still it's stored in the `[[Scope]]`).

Ron Villalon
2017-07-26

Hello Dmitry,

Hats off to you on all the great articles you've published on your website.

Regarding this:

> In ECMAScript there are two statements which can modify scope chain at runtime code execution phase.

…

> Scope = withObject|catchObject + AO|VO + [[Scope]]

Am I wrong to question your statement that "with" and "catch" modify the scope chain? My understanding is that both perform a temporary modification and/or augmentation of AO when it executes. Upon completion, AO is brought back to its original state. So, even though the effect is the same between temporary modification/augmentation and modification of scope chain as you described, technically, the former is the accurate implementation specification of the standard.

Per ECMA-262 standard "the with statement adds an object environment record for a computed object to the lexical environment of the current execution context. It then

executes a statement using this augmented lexical environment. Finally, it restores the original lexical environment."

Here's an example:

```
var a = 0, b = 0;
with ({b, c: 100}) { // take snapshot of current AO (LexicalEnvironment)
    a = 100;
    b = 100;
    console.log(a); // 100
    console.log(b); // 100
    console.log(c); // 100
} // reset current AO to original state (b becomes 0 again and c is delet
console.log(a); // 100
console.log(b); // 0
```

Ron Villalon

2017-07-26

To be clear: "with" and "catch" doesn't modify the scope chain but rather modifies the VO of the current execution context then reverts back to the original state. There is no additional linking going on–only modifying the current link.

Dmitry Soshnikov

2017-07-28

@**Ron Villalon**, good points, although this article used terminology, and descriptions from the ES3 spec, which says:

> *The with statement adds a computed object to the front of the scope chain of the current execution context*

and the same with `catch` clause.

Starting ES5 (and currently), the lexical environments terminology is used, and also with creates a new environment, setting the with object, and passing the old environment as the parent.

You can find details about ES5+ lexical environments in the ES5.3.2 chapter.

Ron Villalon
2017-07-31

I read through the rest of the articles and realized you were talking about ES3. Thanks, Dmitry!

Lee
2017-08-08

hello! Article translated in Korean URL is changed to “http://huns.me/development/334”. 🙂 Thank you!

Dmitry Soshnikov
2017-08-10

@**Lee**, thanks, updated.

Srikanth
2018-02-26

Hello Dimitry, Thank you so much
You have leveled up my understanding of how JavaScript works.

Dmitry Soshnikov
2018-03-01

@**Srikanth**, thanks, glad it’s useful!

← Older Comments

Write a Comment

RELATED CONTENT BY TAG [[SCOPE]] ECMA-262-3 ECMASCRIPT SCOPE CHAIN