理解JavaScript中的Scope

2018年04月19日 18:42:11 九九八八 阅读数 174 更多

介绍

JavaScript有一个特征叫做作用域(Scope)。对于很多新手开发者来讲，作用域的概念不是那么容易理解，这里我尽我可能最简单地向你解释它是什么。会使你的代码出色，减少错误，而且通过它你能够做出强大的设计模式。

什么是作用域 Scope？

作用域 Scope 是你代码中的变量(variable)，函数(function)和对象(object)在运行时(runtime)的可访问性(accessibility)。换句话讲，作用域了在你的代码中的特定区域内，变量和其他资源是否可见。

为什么要使用作用域 Scope：最少访问原则

代码里面为什么要限制变量的可见性，而不是让所有东西在任意位置都可用呢？一个好处是作用域 Scope 为你的代码提供了一定级别的安全。计算机安见原则是每次操作中用户应该只访问他这次访问所需要的东西。

JavaScript 中的作用域 Scope

JavaScript语言中，有两种类型的作用域 Scope ：

- 全局作用域 (Global Scope)
- 本地作用域 (Local Scope)

函数内定义的变量在本地作用域中，而函数外定义的变量处于全局作用域中。每个函数的每次调用都会创建一个新的作用域。

全局作用域 Global Scope

当你打开一个文档(document)开始写JavaScript代码时，你已经在全局作用域 Global Scope 里面了。在整个JavaScript文档里面只有唯一一个全局作用 Scope 。如果一个变量定义在任何一个函数外面，那么它就属于全局作用域 Global Scope 。如下例子所示：

```
// 这是一个JavaScipt文档的内容，而不是某个JavaScript函数定义的内容
// 一个JavaScript文档的缺省作用域是全局作用域(Global Scope)
var name = 'Hammad';
```

全局作用域 Global Scope 里面定义的变量可以在任何一个其他的作用域内被访问或者修改。

```
// 全局作用域变量定义
var name = '全局变量';

console.log(name); // 控制台输出'全局变量'，全局作用域中访问全局作用域变量

function logName() {
    console.log(name); // 函数本地作用域中访问全局作用域变量
}

logName(); // 控制台输出 '全局变量'
```

本地作用域 Local Scope

函数内定义的变量属于本地作用域 Local Scope ，并且每次函数调用这些变量的本地作用域 Local Scope 都不同。这意味着同样名字的变量可以用在中(译注:或者说，不同的函数内部可以定义名字一样的变量)。这是因为，这些变量是绑定到他们各自所属的函数上的，每个都有不同的作用域，并且从访问不到。

```
// 全局作用域 Global Scope

function someFunction() {
    // 本地作用域 Local Scope #1
    function someOtherFunction() {
        // 本地作用域 Local Scope #2
    }
```

```
}

// 全局作用域 Global Scope

function anotherFunction() {
    // 本地作用域 Local Scope #3
}
// 全局作用域 Global Scope
```

语句块 Block Statements

在ES6之前，像 `if` / `switch` 条件，或者'for'/ `while` 循环这样的语句块 `Block Statements`，跟函数不同，他们并不会产生新的作用域。语句块内定义的
在该语句块所在的作用域中。

```
if (true) {
    // 该语句块不产生新的作用域
    var name = '局部变量'; // 这里通过 var定义的变量name的作用域和当前所在if语句块所属的作用域相同
}

console.log(name); // '局部变量'
```

而从ES6开始，引入了关键字 `let` 和 `const` 。这些关键字也是用来定义变量/常量的，可以用来替代 `var` 关键字。**但这两个关键字跟 `var` 不同，它们支持
`Block Statements` 内声明本地作用域。**例子：

```
if (true) {
    // 变量 name 通过 var 定义，
    // 所以属于当前if语句块所从属的作用域
    var name = 'var变量';//该变量的作用域和当前if语句所属作用域一样
    // likes 通过 let 定义，
    // 它属于当前if语句块内新建的一个块级作用域，和当前if语句块所属的作用域不同
    let likes = '变量，属于当前语句块内的块级作用域';
    // skills 通过 const 定义，
    // 它属于当前if语句块内新建的一个块级作用域，和当前if语句块所属的作用域不同
    const skills = '常量，和上面的变量likes的作用域一样，和上面name的作用域不同';
}

console.log(name); // 'var变量'
console.log(likes); // ReferenceError: likes is not defined
console.log(skills); // ReferenceError: skills is not defined
```

> 只要你的应用还处于存活状态，全局作用域就一直存在。
> 只要你的函数还正在被调用和执行，其本地作用域就还存在。

上下文 Context

许多开发人员会经常弄混作用域 `Scope` 和上下文 `Context`，好像二者说的是同一概念。但实际上不是这样的。作用域是上面我们所讲的概念，而上下文
中某些特定的地方表示 `this` 所指向的值。我们可以通过函数的方法改变上下文，这一部分我们稍后会讲。在全局作用域中，上下文总是当前 `Window` 对
里假设使用了浏览器的JavaScript环境）。

```
// 这里会输出当前Window，因为此时this指向当前Window
console.log(this);

// 定义一个函数输出this
function logFunction() {
    console.log(this);
}

// 这个函数在这里的调用还是会输出当前Window，
// 因为此时该函数不属于任何一个对象
logFunction();
```

如果作用域是在一个对象的某个方法里面，上下文就是方法所属的对象：

```
class User {
  logName() {
```

```
        console.log(this);
    }
}

(new User).logName(); // 输出当前所新建的User对象 : User {}
```

这里需要注意的一点是如果你使用 new 关键字调用你的函数，上下文会跟上面所讲的有所不同。这时上下文会被设置成所调用的函数的实例。重新考虑面的例子，这次使用 new 关键字调用函数：

```
function logFunction() {
    console.log(this);
}
// 输出 logFunction {},
// 注意：这里不再输出 Window 对象了，原因：使用了 new 关键字
new logFunction();
```

> 当一个函数在**Strict Mode**下被调用时，上下文缺省为 undefined 。

执行上下文 Execution Context

为了消除上面我们研究内容引起的所有困惑，先说明一点：**执行上下文**中**上下文**一词指的是作用域而不是上下文。这个命名很奇怪，不过JavaScript规范的，我们也只能这么用了。

JavaScript是一个单线程语言，所以同一时间只有一个任务在执行。其他的任务会在“执行上下文”中排队等待执行。上面我已经说了，当JavaScript解释你的代码的时候，上下文(作用域)缺省设置为全局。这个全局的上下文被追加到你的执行上下文，这个上下文实际上是启动执行上下文的第一个上下文。然后，每个函数调用会追加他自己的上下文到执行上下文。当另一个函数在该函数内部或者其他地方被调用时会发生同样的事情。

> 每个函数会创建他自己的执行上下文。

一旦浏览器完成了上下文中的代码，该上下文会被从执行上下文中弹出，然后执行上下文中的当前上下文状态会转到双亲上下文。浏览器总是执行执行上下文(实际上，永远是你的代码中最内层的作用域)。

> 总是只会有一个全局上下文，和任意个函数上下文。

执行上下文分两个阶段：创建和代码执行。

创建阶段

执行上下文的第一个阶段是创建阶段，此阶段出现在函数被调用但是其代码尚未被执行。这一阶段发生的主要事情是:
- 创建变量对象 Variable(Activation) Object
- 创建作用域链 Scope chain
- 设置上下文指针 this

变量对象 Variable Object

变量对象，也可以叫做激活对象，包含了执行上下文某个特定分支里面定义的所有变量，函数或者其他东西。当一个函数被调用时，解释器会扫描所有函数参数，变量定义和其他声明。所有这些东西，被打包成了一个对象，就变成了”变量对象”。

```
'variableObject': {
    // 包括函数参数，内部定义的变量和函数声明
}
```

作用域链 Scope Chain

在执行上下文的创建阶段，作用域链在变量对象创建之后创建。作用域链自身包含了变量对象。作用域链被用于解决(resolve)变量。当被要求解决一个JavaScript总是从代码嵌套的最内层开始，逐层跳到双亲作用域直到找到目标变量或者资源。作用域链可以简单地定义成这样一个对象，它包含了自己的变量对象，同时也包含了所有其他的双亲执行上下文，这是象拥有一堆其他对象的一个对象。

```
'scopeChain': {
    // contains its own variable object
    // and other variable objects of the parent execution contexts
}
```

执行上下文对象

执行上下文可以抽象地表示成如下对象:

```
executionContextObject = {
    // contains its own variableObject
    // and other variableObject of the parent execution contexts
    'scopeChain': {},

    // contains function arguments, inner variable and function declarations
    'variableObject': {},

    'this': valueOfThis
}
```

执行阶段

执行上下文的第二个阶段是代码执行阶段，这里函数体内的代码会最终被执行。

词法作用域 Lexical Scope

词法作用域指的是在一组嵌套的函数中，位于内部的函数能够访问它们双亲作用域中的变量和其他资源。这意味着子函数词法上绑定到了双亲的执行上作用域有时也被叫做静态作用域。看一个例子：

```
function grandfather() {
    var name = 'Hammad';
    // 这里不能访问 likes
    function parent() {
        // 这里可以访问 name
        // 这里不能访问 likes
        function child() {
            // 作用域链的最内层
            // 这里可以访问 name
            var likes = 'Coding';
        }
    }
}
```

这里你会注意到，关于词法作用域，他是向前工作的，也就是说，变量 `name` 可以被它的子执行上下文访问。但是它并不向后工作到它的双亲上，也就是 `likes` 不能被它的双亲上下文访问。这一点也告诉我们，不同执行上下文中名字相同的变量的优先级顺序是从执行栈的栈顶到栈底。**一个变量，如果跟量有同样的名字(译注:在不同的函数中定义)，最内层的函数(执行栈栈顶的上下文)中的那个会拥有最高优先级。**

闭包 Closure

闭包的概念跟词法作用域紧密相连，上面我们已经讲过词法作用域了。当里层函数试图访问其外层函数作用域链时，也就是直接词法作用域之外的变量闭包被创建。闭包有自己的作用域链，其双亲作用域链和全局作用域。

> 闭包不仅能访问定义在其外层函数中的变量，而且可以访问其外层函数的参数。

一个闭包可以访问其外层函数的变量哪怕是函数已经返回(`return`)。这允许返回的函数能够继续访问外层函数的所有资源。

当你从一个函数中返回一个内部定义的函数时，如果你要调用这个函数，它所返回的内部函数并不会被调用。你必须首先保存对外部函数的调用到一个这个变量作为一个函数调用，才能调用到内部定义的那个函数。看一下这个例子：

```
function greet() {
    name = 'Hammad';
    return function () {
        console.log('Hi ' + name);
    }
}

greet(); // 什么都不会发生，控制台也不输出任何东西

// greet() 被执行，它执行返回的函数被记录到了变量 greetLetter
greetLetter = greet();

```

```
// 现在 greetLetter 就是 greet() 函数内部定义并返回的那个函数,
// 将 greetLetter 作为函数调用将会输出 'Hi Hammad'
greetLetter(); // 输出 'Hi Hammad'
```

这里需要注意的关键点是,这个 `greetLetter` 函数可以访问函数 `greet` 的变量，而此时函数 `greet` 已经返回了。

还有一种方法可以不用参数赋值然后调用，而是直接调用 `greet` 返回的函数的方法，那就是使用括号 `()` 两次，就像这样:

```
function greet(){
    name = 'Hammad';
    return function(){
        console.log('Hi ' + name);
    }
}

greet()(); // 输出 'Hi Hammad'
```

公开和私有作用域 Public and Private Scope

在很多其他语言中，可以使用 `public` , `protected` , `private` 等作用域来设置类属性的可见性。思考一下下面这个PHP例子：

```
// Public Scope
public $property;
public function method() {
  // ...
}

// Private Sccpe
private $property;
private function method() {
  // ...
}

// Protected Scope
protected $property;
protected function method() {
  // ...
}
```

将公开或者全局作用域的函数进行封装可以使它们免受攻击。但是在JavaScript中，没有类似 `public` , `private` 这样的作用域。然而，我们可以通过闭
征。为了将所有的东西从全局作用域分开，我们首先要封装我们的函数到这样一个函数中去：

```
(function(){
  // private scope,模拟了一个私有作用域
})();
```

上面例子中函数末尾的括号 `()` 告诉解释器读取到该函数后立即执行。我们可以在该函数中添加变量或者函数并且它们在外面是访问不到的。但是如果我
访问它们该怎么办呢？ 也就是说，我们需要将它们一部分设置为公开 `public` ， 另外一部分设置为私有 `private` 。我们可以使用一种叫做模块模式(`Mod`
`Pattern`)的闭包类型来做到这一点：在一个对象中即可以有 `public` 也可以有 `private` 可见性。

模块模式 Module Pattern

模块模式看起来是这样的：

```
var Module = (function(){
    // 私有方法
    function privateMethod(){
        // do something
    }

    return {
        // 外部可访问方法
        publicMethod: function(){
            // can call privateMethod();
        }
    };
})();
```

模块 `Module` 的返回语句包含了我们的公开函数。私有函数就是那些没有被返回的内部定义的函数。不返回某个函数就是让该函数从模块 `Module` 的命名
失，变得不可访问。但是我们的公开函数还是可以访问这些私有函数，这些私有函数可能是一些让公开函数变得方便的辅助函数，AJAX调用函数或者其

```
Module.publicMethod(); // 正常可工作
Module.privateMethod(); // ReferenceError: privateMethod is not defined
```

一个约定是私有函数使用以下划线开头的函数名称，而包含公开方法的对象以匿名方式返回。这样在一个比较长的对象中管理会变得容易一些。类似这

```
var Module = (function (){
    function _privateMethod(){
        // do something
    }
    function publicMethod(){
        // do something
    }
    return {
        publicMethod: publicMethod,
    }
})();
```

立即调用函数表达式 IIFE

另外一种闭包类型是立即调用函数表达式(Immediately-Invoked Function Expression) `IIFE` 。这是一种将window作为上下文的匿名自调用函数，也就是
里面， `this` 被设置成了 `window` 。这样就暴露了唯一一个全局接口用来交互。它是这么工作的：

```
// 一个匿名自调用函数，形式参数是 window,
// 在JavaScript文档最外层执行这段代码时，实际参数 this 其实就是当前的 window 对象
(function(window){
    // do anything
})(this);
```

使用 .call(),.apply() 和 .bind()改变上下文

函数 `call` 和 `apply` 被用于调用一个函数时改变其上下文。这一点提供了不可思议的编程能力。要使用 `call` 和 `apply` ,你需要在目标函数上调用它们(并
数必须是要使用的上下文)，而不是使用 `()` 方式直接调用目标函数。就像这个样子：

```
function hello(){
    // do something...
}

hello(); // 通常的函数调用方式
hello.call(context); // 现在 hello 函数内部的 this 是这里的 context
hello.apply(context); // 现在 hello 函数内部的 this 是这里的 context
```

`.call()` 和 `.apply()` 的不同在于，除了第一个上下文参数之外， `.call()` 的其他参数都需要一个一个写出来，而 `.apply()` 的其他参数是通过一个数组
例子：

```
function introduce(name, interest){
    console.log('Hi! I\'m '+ name +' and I like '+ interest +'.');
    console.log('The value of this is '+ this +'.')
}

// 通常的函数调用方式
introduce('Hammad', 'Coding');
// 列出每个参数
introduce.call(window, 'Batman', 'to save Gotham');
// 上下文之外的参数使用数组方式传递(这个例子里面上下文传递了字符串 Hi)
introduce.apply('Hi', ['Bruce Wayne', 'businesses']);

// Output:
// Hi! I'm Hammad and I like Coding.
// The value of this is [object Window].
// Hi! I'm Batman and I like to save Gotham.
// The value of this is [object Window].
// Hi! I'm Bruce Wayne and I like businesses.
// The value of this is Hi.
```

> 性能上讲， `call` 比 `apply` 略快。

下面看一个例子，该例子中，一个匿名函数通过变化上下文方式作用到文档中的一组 `<li>` 元素上，并将它们一个个输出到控制台：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Things to learn</title>
</head>
<body>
    <h1>Things to Learn to Rule the World</h1>
    <ul>
        <li>Learn PHP</li>
        <li>Learn Laravel</li>
        <li>Learn JavaScript</li>
        <li>Learn VueJS</li>
        <li>Learn CLI</li>
        <li>Learn Git</li>
        <li>Learn Astral Projection</li>
    </ul>
    <script>
    // Saves a NodeList of all list items on the page in listItems
    var listItems = document.querySelectorAll('ul li');
    // Loops through each of the Node in the listItems NodeList and logs its content
    for (var i = 0; i < listItems.length; i++) {
      (function () {
        console.log(this.innerHTML);
      }).call(listItems[i]);
    }

    // Output logs:
    // Learn PHP
    // Learn Laravel
    // Learn JavaScript
    // Learn VueJS
    // Learn CLI
    // Learn Git
    // Learn Astral Projection
  </script>
</body>
</html>
```

这个HTML文档只包含了一组无序 `<li>` 。JavaScript将它们全部从 `DOM` 拿到，然后循环遍历列表中的每个元素。循环内部，我们将每个 `<li>` 元素的内容
台上。

输出语句被封装到了一个匿名函数中然后对其调用了 `call` 方法，传递的上下文是当前的 `<li>` 元素，所以该函数内部的 `this` 也就是当前的 `<li>` 元素，
输出语句就能正确地输出每个 `<li>` 元素的 `innerHTML` 内容了。

对象可以有方法，类似地，函数也是对象，也可以有方法。事实上，一个JavaScript函数带有四个内置方法：

- Function.prototype.apply()
- Function.prototype.bind() (Introduced in ECMAScript 5 (ES5))
- Function.prototype.call()
- Function.prototype.toString()

> Function.prototype.toString() 返回一个字符串，表示该函数的源代码。

截止到目前，我们讨论了 `.call()` , `.apply()` , 和 `toString()` 。跟 `.call()` 和 `.apply()` 不同，另外一个函数 `.bind()` 自己并不调用函数，而只是用
用前绑定函数的上下文值和其他一些参数。下面是一个使用 `.bind()`的例子：

```
// 该例子都采用浏览器JavaScript环境，如果是nodejs环境，将window换成global
(function introduce(name, interest) {
    console.log('Hi! I\'m '+ name +' and I like '+ interest +'.');
    console.log('The value of this is '+ this +'.')
```

```
5 }).bind(window, 'Hammad', 'Cosmology')();
6
7 // 控制台输出:
8 // Hi! I'm Hammad and I like Cosmology.
9 // The value of this is [object Window].
```

`bind` 类似 `call` ,上下文之外的参数的传递需要一个一个列出来而不是像 `apply` 那样需要传递数组。

参考资料

英文原文 : Understanding Scope in JavaScript
相关另外一篇 : Understanding Scope and Context in JavaScript