潜力无限的JavaScript

二、JavaScript进阶基础知识

周志明 @ 开发二部
2013年5月

潜力无限

- 越来越多的应用使用JavaScrip来完成
  - ✓ Window 8后支持原生程序开发。
  - ✓ HTML5下的程序与游戏体验。
  - ✓ Node.js支持下的服务端应用。
- 钱力无限
  - ✓ 不受OS限制，原生态无污染的天然跨平台语言。
  - ✓ 会JavaScript的人很多，精JavaScript的人极少。
  - ✓ Web时代的银弹，随着Web流行而发展。

不是《零基础学会JavaScript》

不是《21天精通JavaScript语言》

不是《论前端开发人员的修养》

只讨论一些JavaScript令人又爱又恨的特性

目录

1. 浏览器渲染与JavaScript执行原理
   - ✓ 浏览器渲染和执行引擎、阻塞、异步、定时器、事件流
2. JavaScript进阶基础知识
   - ✓ 执行上下文、变量对象、this指针、作用域链
3. 面向对象的JavaScript
   - ✓ 加载、执行、函数对象、原型继承、闭包、柯里化
4. 性能与JavaScript
   - ✓ 性能陷阱、“看起来快些”的技巧、HTTP头优化、问题排查工具

引言1：JavaScript难在哪里？

- 主观因素
  - ✓ 思维惯性，JS是做网页的东西，小打小闹木有前途。
  - ✓ JS没有技术含量，谁都会，没必要花精力研究。
- 客观因素
  - ✓ 标准库最少的常用语言，没有之一。
  - ✓ 各种反人类的概念和设计。Brendan Eich称设计的时间和对应用领域的估算不足，带来了很多愚蠢的理念。
  - ✓ 运行环境，实现版本差异大，历史包袱沉重。ECMAScript 4曾经尝试打破历史包袱，很不幸失败了。

引言1：JavaScript难在哪里？

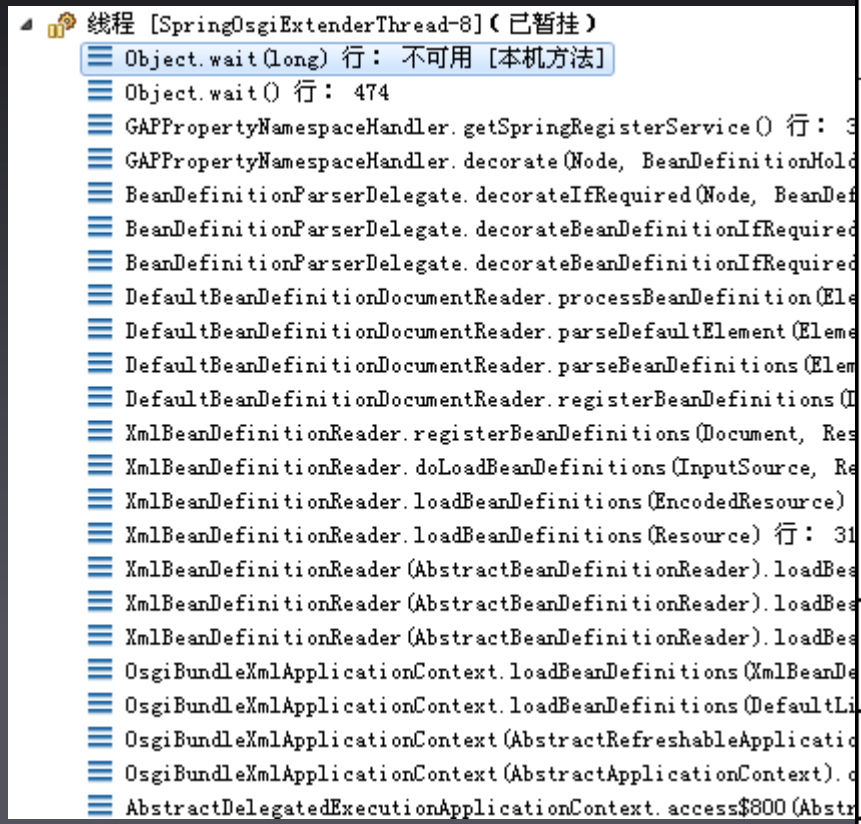
- 主观因素
  - ✓ 思维惯性，JS是做网页的东西，小打小闹木有前途。
  - ✓ JS没有技术含量，谁都会，没必要花精力研究。
- 客观因素
  - ✓ 标准库最少的常用语言，没有之一。
  - ✓ 各种反人类的概念和设计。Brendan Eich称设计的时间和对应用领域的估算不足，带来了很多愚蠢的理念。
  - ✓ 运行环境，实现版本差异大，历史包袱沉重。ECMAScript 4曾经尝试打破历史包袱，很不幸失败了。

今天我们要讲的主要内容。

引言2：程序执行环境

- 全静态执行环境：在程序执行期间内存中的数据都是被方法共享的、固定的、静态的，通过固定的地址来直接访问到这些变量。没有指针和动态内存分配的概念，也不支持函数递归调用。例子：FORTRAN77。
- 基于栈的执行环境：在程序执行期间，全局数据区中存放共享数据。每个方法都有自己的私有数据，以栈帧形式存放在执行栈中。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在栈中入栈到出栈的过程。例子： C、C++、Java、JavaScript

引言2：基于栈的执行环境

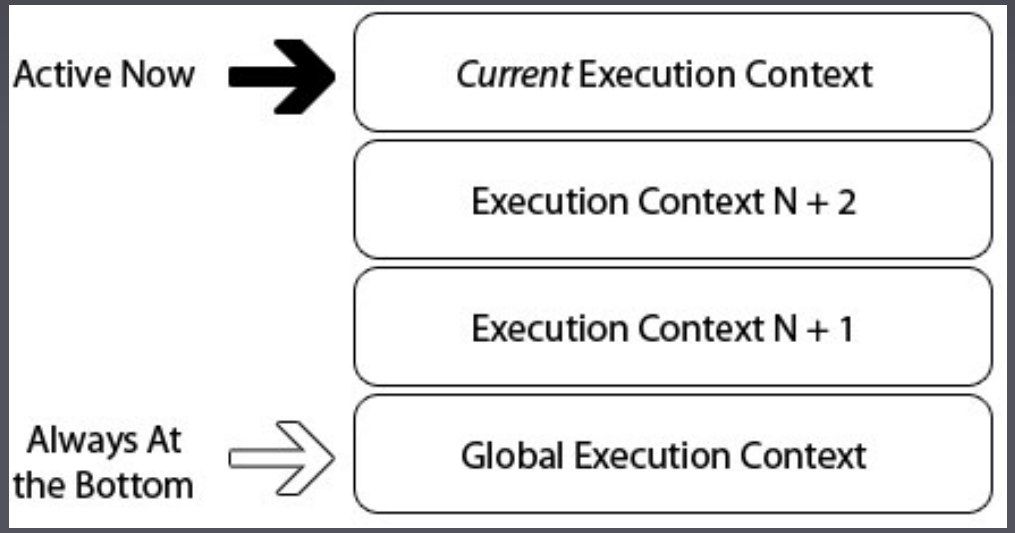


引言2：JavaScript中的执行环境

- ECMA-262-5定义：When control is transferred to ECMAScript executable code, control is entering an execution context. Active execution contexts logically form a stack. The top execution context on this logical stack is the running execution context.
- 说人话：JavaScript中的“栈帧”，函数执行时，用来存储执行状态信息（局部变量、this指针、作用域链、方法出口入口地址等）的地方。

执行上下文

- 什么时候产生执行上下文？
  - ✓ 每一个可执行代码在被调用时都会创建一个执行上下文，重复调用函数（包含递归调用的情形）会重新创建新的执行上下文，而后放置在堆栈中。堆栈会在程序运行时随着新的函数调用、函数return、未处理的异常抛出等情况动态变化，但堆栈的最顶部总是当前正运行的执行上下文，它的最底部总是全局执行上下文(Global Context)。



执行上下文

- 执行上下文中具体包含有什么内容？
  - ✓ 变量对象（ES3）
    - 词法环境（ES5）：存储在上下文中定义的变量、函数，以及函数中的形参（含arguments对象）
    - 变量环境（ES5）：在最初与语法环境是一致的，但变量环境在整个上下文中不会改变，而语法环境可以改变（如with语句）。
  - ✓ this指针
  - ✓ 作用域链
  - ✓ 对于具体引擎实例来说，可以将ECMA-262规范之外用户程序不可见的信息存储在执行上下文中
    - 方法入口、连接信息、返回地址等等

执行上下文

- 产生怎样的执行上下文？
  - ✓ 上下文类型取决于执行代码的类型
  - ✓ Global code（<Script>标签）— 全局上下文
  - ✓ Function code（函数调用）— 函数上下文
  - ✓ Eval code（eval()函数） — 执行位置所处的上下文
    - eval()别名函数 — 全局上下文

变量对象

- ECMA-262-3定义：A variable object is a scope of data related with the execution context. It’s a special object associated with the context and which stores variables、function declarations and arguments are being defined within the context.
- 说人话：在执行上下文中，负责存储局部变量的对象。

变量对象

- 有什么用？
  - ✓变量自己应该知道它的数据存储在哪里，并且知道如何访问。这种机制称为变量对象(variable object)。
- 变量对象存储着在上下文中声明的以下内容：
  1. 变量 (var, 变量声明);
  2. 函数声明 (Function Declaration);
  3. 函数的形参(arguments)

变量对象

- 示例：

```
<script type="text/javascript">
    var a = 10;

    function test(x) {
        var b = 20;
    };

    test(30);
</script>
```

→

```
// 全局上下文的变量对象
VO(globalContext) = {
    a: 10,
    test: <reference to function>
};

// test函数上下文的变量对象
VO(test functionContext) = {
    x: 30,
    b: 20
};
```

- 注意：只有全局上下文的变量对象允许通过VO的属性名称来间接访问(因为在全局上下文里，全局对象自身就是变量对象，稍后会详细介绍)，在其它上下文中是不能直接访问VO对象的，因为它只是内部机制的一个实现。

全局上下文中的变量对象

- 全局对象特征
  - ✓ 在进入任何执行上下文之前就已经创建了的对象，这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于页面退出那一刻。即global是全局的，单例的。
  - ✓ 全局对象初始创建阶段将Math、String、Date、parseInt作为自身属性，同样也可以有额外创建的其它对象作为属性。
  - ✓ 全局上下文中，this指针指向了全局对象，即global === this
  - ✓ 在基于浏览器的JavaScript中，全局对象的window属性引用了全局对象自身，即global === window

全局对象

- 非常有必要要理解全局对象的4个特征，基于这些特征，在全局上下文中声明的变量，我们才可以间接通过全局对象的属性来访问它。

```
<script type="text/javascript">
    String(10); // 就是global.String(10);

    // 带有前缀
    window.a = 10; // === global.window.a = 10 === global.a = 10;
    this.b = 20; // global.b = 20;

    var a = new String('test');

    alert(a); // 直接访问，在VO(globalContext)里找到："test"

    alert(window['a']); // 间接通过global访问：global === VO(globalContext): "test"
    alert(a === this.a); // true

    var aKey = 'a';
    alert(window[aKey]); // 间接通过动态属性名称访问："test"
</script>
```

函数上下文中的变量对象

- 活动对象的特征
  - ✓ 在进入函数时初始化，在函数退出，没有任何Root与之关联的之后（无论是函数结束还是出现异常）销毁。
  - ✓ 由形参名称和实参值组成的0至多个变量对象的属性被创建，如果仅定义了形参而没有传递对应参数的话，那么由名称和undefined值组成的一种变量对象的属性也将被创建。
  - ✓ 一个名为arguments，类型为Arguments的对象被初始化
    - callee — 指向当前函数的引用
    - length — 真正传递的参数个数

活动对象

- argument：

```
<script type="text/javascript">
    function foo(x, y, z) {
      // 声明的函数参数数量arguments (x, y, z)
      alert(foo.length); // 3
      // 真正传进来的参数个数(only x, y)
      alert(arguments.length); // 2
      // 参数的callee是函数自身
      alert(arguments.callee === foo); // true
      // 参数共享
      alert(x === arguments[0]); // true
      alert(x); // 10
      arguments[0] = 20;
      alert(x); // 20
      x = 30;
      alert(arguments[0]); // 30
      // 不过，没有传进来的参数z，和参数的第3个索引值是不共享的
      z = 40;
      alert(arguments[2]); // undefined
      arguments[2] = 50;
      alert(z); // 40
    }
    foo(10, 20);
</script>
```

上下文的初始化和执行阶段

- 初始化阶段
  1. 初始化函数的形参
     由名称和对应值组成的一个变量对象的属性被创建；没有传递对应参数的话，那么由名称和undefined值组成的一种变量对象的属性也将被创建。
  2. 初始化函数声明
     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. 初始化变量声明
     由名称和对应值（undefined）组成若干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。
- 执行阶段
  - ✓ 根据程序执行路径，填充、改变所声明变量值。

初始化阶段

- 示例：初始化阶段

```
<script type="text/javascript">
    function test(a, b) {
      var c = 10;
      function d() {}
      var e = function _e() {};
      (function x() {});
    }

    test(10); // call
</script>
```

→

```
<script type="text/javascript">
    AO(test) = {
      a: 10,
      b: undefined,
      c: undefined,
      d: <reference to FunctionDeclaration "d">
      e: undefined
    };
</script>
```

初始化阶段

- 示例：执行阶段

```
<script type="text/javascript">
    function test(a, b) {
      var c = 10;
      function d() {}
      var e = function _e() {};
      (function x() {});
    }

    test(10); // call
</script>
```

→

```
<script type="text/javascript">
    AO(test) = {
      a: 10,
      b: undefined,
      c: 10,
      d: <reference to FunctionDeclaration "d">
      e: <reference to FunctionDeclaration "e">
    };
</script>
```

定义变量

- 在初始化阶段中定义变量：var
- 一个观点：
  - ✓ 不管是否使用var关键字，都可以声明一个变量，区别仅是var关键字定义的变量在当前上下文中，而不使用var关键字，则在全局上下文中。

定义变量

- 一个观点：
  - ✓ 不管是否使用var关键字，都可以声明一个变量，区别仅是var关键字定义的变量在当前上下文中，而不使用var关键字，则在全局上下文中。

定义变量

- 声明变量必须使用var关键字。
- var关键字的特征：
  - ✓ 在上下文初始化阶段被解析。
  - ✓ var的解析过程不受程序执行路径的影响。
  - ✓ 运作方式为：首先检查活动上下文的变量对象是否已经具有var所声明的变量，如果有，跳过什么都不做。如果没有，则声明这个变量。

定义变量

- 测试1：

```
<script type="text/javascript">
    alert(a);       // 输出undefined
    alert(b);       // 抛出异常："b is not defined"

    b = 10;
    var a = 20;
</script>
```

- 结论：
  - ✓ b = 10等效与window.b = 10，与前面介绍的new String(10)等效于new window.String(10)是一个道理，这很明显是window对象b属性的赋值操作。

定义变量

- 测试2：

```
<script type="text/javascript">
    a = 10;
    alert(window.a); // 10
    alert(delete a); // true
    alert(window.a); // undefined

    var b = 20;
    alert(window.b); // 20
    alert(delete b); // false
    alert(window.b); // still 20
</script>
```

- 结论：
  - ✓ 属性和变量的一个重要区别是，变量具有{DontDelete}属性，标识不能被delete语句删除。
  - ✓ 例外：eval函数中定义的变量没有这个属性

问题1

- 3个alert()输出各是什么？为什么？

```
<script type="text/javascript">
    alert(x);
    var x = 10;
    alert(x);
    x = 20;
    function x() {};
    alert(x);
</script>
```

问题1

- 3个alert()输出各是什么？为什么？

```
<script type="text/javascript">
    alert(x);       // 输出function x(){}
    var x = 10;
    alert(x);       // 输出10
    x = 20;
    function x() {};
    alert(x);       // 输出20
</script>
```

- 输出function x(){}的原因：
  - ✓ 函数定义优先于变量定义，在解析var x时，x已经存在，根据前面介绍var的作用“如果已经声明，则跳过”，因此var x实际并没有执行。

问题2

- alert()输出是什么？为什么？

```
<script type="text/javascript">
    if (!("a" in window)) {
        var a = 1;
    }
    alert(a);
</script>
```

问题2

- alert()输出是什么？为什么？

```
<script type="text/javascript">
    if (!("a" in window)) {
        var a = 1;
    }
    alert(a);
</script>
```

- 答案：undefined
  - ✓ 原因很简单，if语句根本没进去。“a” in window返回true

问题3

- alert()输出是什么？为什么？

```
<script type="text/javascript">
    var a = 1,
    b = function a(x) {
        if (x > 0) {
            a(--x);
        } else {
            alert("stop");
        }
    };
    alert(a);
</script>
```

问题3

- alert(a)输出是什么？为什么？

```
<script type="text/javascript">
    var a = 1,
    b = function a(x) {
        if (x > 0) {
            a(--x);
        } else {
            alert("stop");
        }
    };
    alert(a);
</script>
```

- 答案：1
  - ✓ b为函数赋值，function a(x)的定义不会在解析阶段执行。

关于this

- 关键字this的特征
  - ✓ 与变量对象一样，this也是执行上下文的一个属性。
  - ✓ this值在进入上下文时确定，在进入之前可以改变，但是在上下文运行期间永久不变。

全局上下文中的this

- this就是全局上下文对象本身
  - ✓ 始终满足关系 global = window = this

```
<script type="text/javascript">
    // 显示定义全局对象的属性
    this.a = 10; // global.a = 10
    alert(a); // 10

    // 通过赋值给一个无标示符隐式
    b = 20;
    alert(this.b); // 20

    // 也是通过变量声明隐式声明的
    // 因为全局上下文的变量对象是全局对象自身
    var c = 30;
    alert(this.c); // 30
</script>
```

函数上下文中的this

- 在函数调用中，this对象是由方法执行方式所决定的
  - ✓ 示例：

```
<script type="text/javascript">
    var foo = {
        bar: function () {
            alert(this);
            alert(this === foo);
        }
    };

    foo.bar(); // foo, true
    var exampleFunc = foo.bar;
    alert(exampleFunc === foo.bar); // true
    // 再一次，同一个function的不同的调用表达式，this是不同的
    exampleFunc(); // global, false
    // 再来一次
    // 另外一种形式的调用表达式
    exampleFunc.prototype.constructor(); // foo.prototype,false
</script>
```

函数上下文中的this

- 在函数调用中，this对象是由方法执行方式所决定的
  - ✓ 如果是直接调用函数，this值为global对象（有一些极端情况的例外）。
  - ✓ 如果通过对象调用函数，this值为该对象。
  - ✓ 如果通过表达式调用函数（例子：(foo.bar, foo.bar)()），这时候需要判断返回结果是函数直接调用还是引用对象调用决定this的值。
  - ✓ 如果作为构造器调用函数（new bar()），this的值为构造器所创造的对象。
  - ✓ 如果通过call或apply方法间接函数，this值为call或apply的第一个参数
  - ✓ 如果通过bind方法对函数进行了绑定，那this的值即为已绑定的对象。

函数上下文中的this

```
<script type="text/javascript">
    var x = 1;
    function foo() { alert(this.x); }
    var bar = { x : 2 };
    bar.foo = foo;
</script>
```

```
<script type="text/javascript">
    // 直接调用
    foo();          // 输出1
    // 通过对象调用
    bar.foo();    // 输出2
    // 通过表达式调用
    (bar.foo = bar.foo)(); // 输出1
    (false || bar.foo)();  // 输出1
    (bar.foo, bar.foo)();  // 输出1
    // 在函数表达式中调用
    (function () {
        var x = 3;
        foo();    // 输出1；
    })();
    // 作为构造器调用
    var _foo = new foo();  // 输出undefined
    // 通过call或者apply调用
    foo.apply({x:3});       // 输出3
    // 通过bind调用
    foo = foo.bind({x:4});  // ECMAScript 5 Only
    foo();                  // 输出4
    foo().apply({x:3});     // 还是输出4
</script>
```

作用域链

- 什么是作用域链？

```
<script type="text/javascript">
    var x = 10;
    function foo() {
        var y = 20;
        function bar() {
            alert(x + y);
        }
        return bar;
    }
    foo()(); // 30
</script>
```

- ECMAScript 允许创建内部函数，我们甚至能从父函数中返回这些函数。作用域链正是内部上下文所有变量对象（包括父变量对象）的列表。此链用来变量解析查询。

作用域链

- 作用域链的特征
  - ✓ 是执行上下文的一个属性

```
activeExecutionContext = {
    VO: {...},
    this: thisValue,
    Scope: [ // Scope chain
      // 所有变量对象的列表
    ]
};
```

  - ✓ 逻辑上是一个数组，每个元素都是变量对象
  - ✓ 定义为：

    Scope = ActiveContext.VO + Function.[[Scope]]

  - ✓ 注意，[[Scope]]是函数的属性。

作用域链

- 理解函数的[[Scope]]属性
  - ✓ 此代码的执行结果？为什么？

```
<script type="text/javascript">
    var x = 10;

    function foo() {
        alert(x);
    }

    (function () {
        var x = 20;
        foo();
    })();
</script>
```

作用域链

- 理解函数的[[Scope]]属性
  - ✓ [[Scope]]是函数的私有属性，在函数被解析时建立，不会改变。
  - ✓ 此代码的执行结果？为什么？

```
<script type="text/javascript">
    var x = 10;

    function foo() {
        alert(x);
    }

    (function () {
        var x = 20;
        foo();      // 输出10，而不是20
    })();
</script>
```

  - ✓ 变量解析为10，而不是20。这个例子也清晰的表明，一个函数的[[Scope]]持续存在，即使是在函数创建的作用域已经完成之后。

作用域链

- 理解函数的[[Scope]]属性
  - ✓ [[Scope]]“通常”包含了父级函数的[[Scope]]属性，ECMAScript通过这个特性来实现闭包访问。“通常”意味着存在一些例外。
  - ✓ [[Scope]]是函数的属性，这点也决定了ECMAScript中没有Java那样的块级作用域。只有函数级作用域。

作用域链

- 理解函数的[[Scope]]属性
  - ✓ 观察以下3种函数构造方式的差异

```
<script type="text/javascript">
    var x = 10;

    function foo() {
        var y = 20;
        // 通过函数声明创建函数
        function barFD() { alert(x); alert(y); }
        // 通过函数表达式创建函数
        var barFE = function () { alert(x); alert(y); };
        // 通过函数构造器创建函数
        var barFn = Function('alert(x); alert(y);');

        barFD(); // 10, 20
        barFE(); // 10, 20
        barFn(); // 10, "y" is not defined
    }
</script>
```

作用域链

- 变量的二维链式查找
  - ✓ 变量的解析是通过作用域链来实现的。
  - ✓ 变量本质上是以变量对象的属性方式存在的，当变量对象与JavaScript中对象重叠时，它就天然地会受到原型链的影响。

作用域链

- 变量的二维链式查找

```
<script type="text/javascript">
    function foo() {
        alert(x);
    }

    Object.prototype.x = 10;

    foo(); // 输出10
</script>
```

✓ 原因：

- global === window，window是Object所派生的。
- 根据原型链的查找规则，实例中访问不到的属性和方法，将会在原型中查找。

作用域链

- 变量的二维链式查找

```
<script type="text/javascript">
    var x = 3
    function foo() {
        var x = 2;
        function bar() {
            var x = 1;
            alert(x);
        }
        bar();
    }
    Object.prototype.x = 4;
    foo();   // 输出1、2、3、4的优先级依次降低

</script>
```

```
Scope(bar) = VO(bar) + bar.[[Scope]]
           = VO(bar) + VO(foo) + foo.[[Scope]]
           = VO(bar) + VO(foo) + VO(global)
           = VO(bar) + VO(foo) + window
                                   |
                              Object.prototype
```

作用域链

- 全局代码和Eval代码中的作用域链
  - ✓ 全局上下文的作用域链仅包含全局对象。
  - ✓代码eval的上下文与当前的调用上下文（calling context）拥有同样的作用域链。
  - ✓ ECMAScript 5规定，如果对eval建立别名（非直接调用），这时作用域链仅包含全局对象。

作用域链

- 作用域链是可在运行时改变的
  - ✓在with和catch语句块中：
    Scope = withObject|catchObject + VO + [[Scope]]
  - ✓大多数情况下是不变的，但在with和catch语句块中，可以改变作用域链。这种技巧在些时候很有用，但大多数情况下应尽可能避免。
  - ✓ ECMAScript 5中，通过词法环境、词法环境记录的方式来描述这种变化。

作用域链

- 在with和catch语句块中的作用域链
  - ✓ 问题：以下4个alert()输出什么？为什么？

```
<script type="text/javascript">
    var x = 10, y = 10;

    with ({x: 20}) {
        var x = 30, y = 30;
        alert(x);
        alert(y);
    }

    alert(x);
    alert(y);
</script>
```

作用域链

- 在with和catch语句块中的作用域链
  - ✓ 问题：以下4个alert()输出什么？为什么？

```
<script type="text/javascript">
    var x = 10, y = 10;

    with ({x: 20}) {
        var x = 30, y = 30;
        alert(x);   // 输出30
        alert(y);   // 输出30
    }

    alert(x); // 输出10
    alert(y); // 输出30
</script>
```

作用域链

- 在with和catch语句块中的作用域链
  1. x = 10, y = 10;
  2. 对象{x:20}添加到作用域的前端;
  3. 在with内部，遇到了var声明，当然什么也没创建，因为在进入上下文时，所有变量已被解析添加;
  4. 在第二步中，仅修改变量“x”，实际上对象中的“x”现在被解析，并添加到作用域链的最前端，“x”为20，变为30;
  5. 同样也有变量对象“y”的修改，被解析后其值也相应的由10变为30;
  6. 此外，在with声明完成后，它的特定对象从作用域链中移除（已改变的变量“x” - - 30也从那个对象中移除），即作用域链的结构恢复到with得到加强以前的状态。
  7. 在最后两个alert中，当前变量对象的“x”保持相同，“y”的值现在等于30，在with声明运行中已发生改变。

作用域链

- 结合this指针一起
  - ✓ 直接调用函数，作用域为withObject

```
<script type="text/javascript">
    var x = 10;
    with ({
        foo: function () {
            alert(this.x);
        },
        x: 20
    }) {
        foo(); // 输出20
    }
</script>
```

总结

- 理解执行上下文，以及其中的变量对象、this、作用域链是理解JavaScript程序执行的基础。
- 尤其是this和作用域链，是下一次讲解面向对象JavaScript的必要基础。

References

- 《ECMAScript Language Specification》
  - ✓ http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf
- 《深入学习Javascript》
  - ✓ http://blog.goddyzhao.me/JavaScript-Internal
- 《深入了解JavaScript系列》
  - ✓ http://www.cnblogs.com/TomXu/archive/2011/12/15/2288411.html
- 《Inside-the-Browser》