你真的知道 JavaScript 中的作用域对性能的影响么

随着用户体验的日益重视，前端性能对用户体验的影响备受关注，但由于引起性能问题的原因相对复杂，我们很难但从某一方面或某几个方面来全面解决它，接下来用一系列文章来深层次探讨与梳理有关 Javascript 性能的方方面面，以填补并夯实大家的知识结构。

接下来我们从作用域的角度来聊一聊，看看作用域相关的性能提升有哪些。

我们先剪短回顾一下作用域的基本知识。

作用域

- 任何程序设计语言都有作用域的概念，简单的说，作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。
- 作用域(scope)是 JAVASCRIPT 编程中一个重要的运行机制，在 JAVASCRIPT 同步和异步编程以及 JAVASCRIPT 内存管理中起着至关重要的作用。
- 在 JAVASCRIPT 中，能形成作用域的有如下几点。
    - 函数的调用
    - with 语句
        - with 会创建自己的作用域，因此会增加其中执行代码的作用域的长度。
    - 全局作用域。
    - 块级作用域（ES6 中提出的）。

1. 全局作用域（Global Scope）

在代码中任何地方都能访问到的对象拥有全局作用域，一般来说以下几种情形拥有全局作用域：

1）最外层函数和在最外层函数外面定义的变量拥有全局作用域，例如：

```
var authorName="小渡";
function doSomething(){
```

```
    var name="渡一";
    function innerSay(){
        console.log(name);
    }
    innerSay();
}
doSomething(); //渡一
```

2）所有未定义直接赋值的变量自动声明为拥有全局作用域，例如：

```
function doSomething(){
    var myName = 'duyi';
        globalName = 'global';
        console.log(myName);
}
doSomething(); //duyi
console.log(myName); //global
```

变量 globalName 拥有全局作用域，myName 而在函数外部无法访问到。

3）所有 window 对象的属性拥有全局作用域

一般情况下，window 对象的内置属性都拥有全局作用域，例如 window.name、window.location、window.top 等等。

2. 局部作用域（Local Scope）

和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到，最常见的例如函数内部，所以有人把这种作用域也称为函数作用域，例如下列代码中的 blogName 和函数 innerSay 都只拥有局部作用域。

```
function doSomething(){
    var blogName="渡一";
    function innerSay(){
```

```
        alert(blogName);
    }
    innerSay();
}
```

3. 块级作用域（Block Scope）

在 ES6 中新提出的块级作用域。每个大括号中都会产生块级作用（对象的大括号不算）：

```
let globalName = 'global';
{
        let blockName = 'block';
}
console.log(globalName); // global
console.log(blockName); // error
```

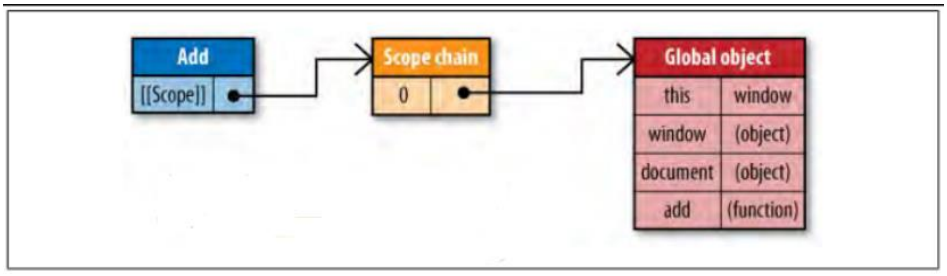
作用域链

在 JavaScript 中，函数也是对象，实际上，JavaScript 里一切都是对象。函数对象和其它对象一样，拥有可以通过代码访问的属性和一系列仅供 JavaScript 引擎访问的内部属性。其中一个内部属性是[[Scope]]，由 ECMA-262 标准第三版定义，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链，它决定了哪些数据能被函数访问。

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象填充。例如定义下面这样一个函数：

```
function add(num1,num2) {
    var sum = num1 + num2;
    return sum;
}
```

在函数 add 创建时，它的作用域链中会填入一个全局对象，该全局对象包含了所有全局变量，如下图所示（注意：图片只例举了全部变量中的一部分）：

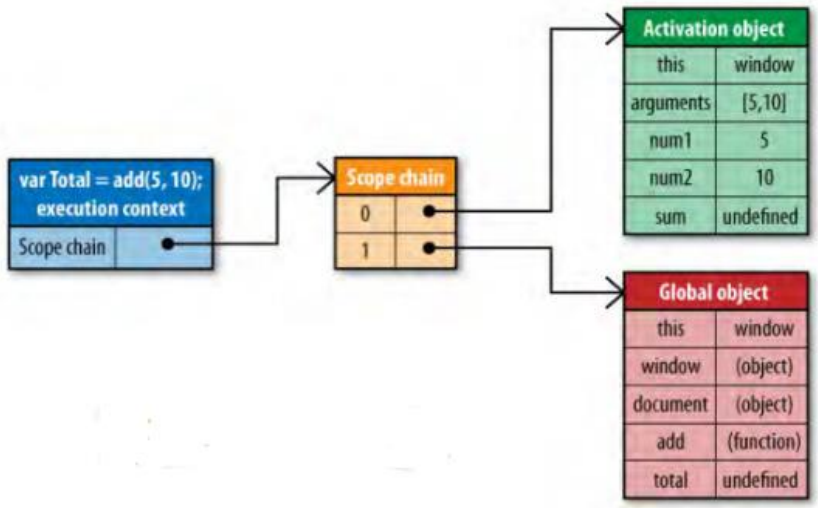


函数 add 的作用域将会在执行时用到。例如执行如下代码：

```
var  total = add(5,10);
```

执行此函数时会创建一个称为“运行期上下文(execution context)”的内部对象，运行期上下文定义了函数执行时的环境。每个运行期上下文都有自己的作用域链，用于标识符解析，当运行期上下文被创建时，而它的作用域链初始化为当前运行函数的[[Scope]]所包含的对象。

这些值按照它们出现在函数中的顺序被复制到运行期上下文的作用域链中。它们共同组成了一个新的对象，叫“活动对象(activation object)”，该对象包含了函数的所有局部变量、命名参数、参数集合以及 this，然后此对象会被推入作用域链的前端，当运行期上下文被销毁，活动对象也随之销毁。新的作用域链如下图所示：



在函数执行过程中，没遇到一个变量，都会经历一次标识符解析过程以决定从哪里获取和存储数据。该过程从作用域链头部，也就是从活动对象开始搜索，查找同名的标识符，如果找到了就使用这个标识符对应的变量，如果没找到继续搜索作用域链中的下一个对象，如果搜索完所有对象都未找到，则认为该标识符未定义。函数执行过程中，每个标识符都要经历这样的搜索过程。

改变作用域链带来的问题

函数每次执行时对应的运行期上下文都是独一无二的，所以多次调用同一个函数就会导致创建多个运行期上下文，当函数执行完毕，执行上下文会被销毁。每一个运行期上下文都和一个作用域链关联。一般情况下，在运行期上下文运行的过程中，其作用域链只会被 with 语句和 catch 语句影响。

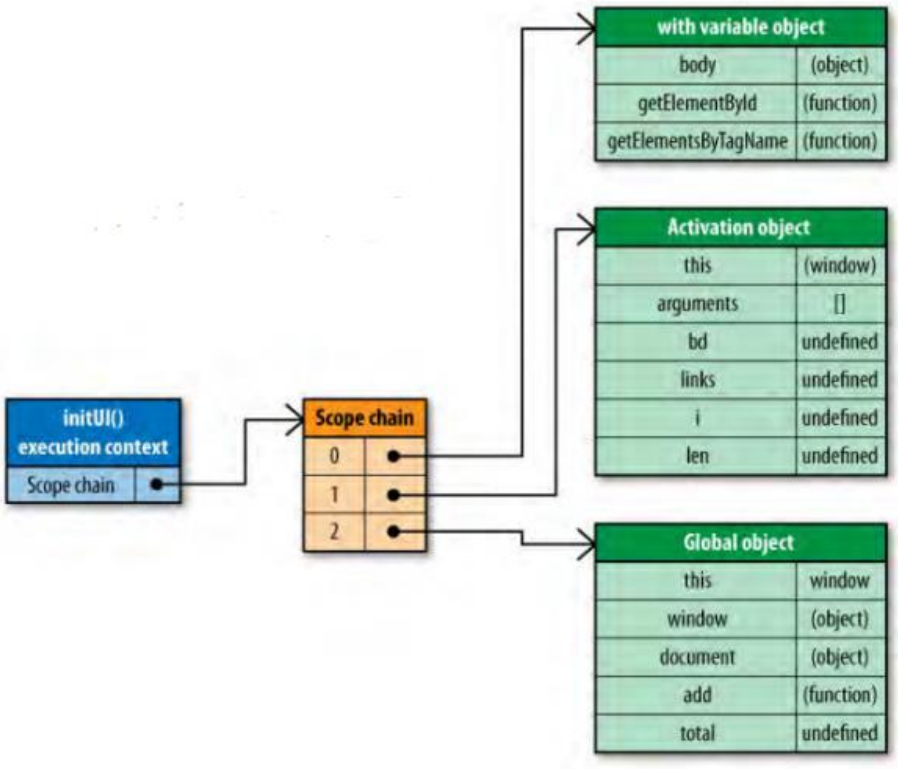
with 语句是对象的快捷应用方式，用来避免书写重复代码。例如：

```
function initUI(){
    with(document){
        var bd=body,
            links=getElementsByTagName("a"),
            i=0,
            len=links.length;
```

```
        while(i < len){
            update(links[i++]);
        }
        getElementById("btnInit").onclick=function(){
            doSomething();
        };
    }
}
```

这里使用 with 语句来避免多次书写 document，看上去更高效，实际上产生了性能问题。

当代码运行到 with 语句时，运行期上下文的作用域链临时被改变了。一个新的可变对象被创建，它包含了参数指定的对象的所有属性。这个对象将被推入作用域链的头部，这意味着函数的所有局部变量现在处于第二个作用域链对象中，因此访问代价更高了。如下图所示：



因此在程序中应避免使用 with 语句，在这个例子中，只要简单的把 document 存储在一个局部变量中就可以提升性能。

对于

另外一个会改变作用域链的是 try-catch 语句中的 catch 语句。当 try 代码块中发生错误时，执行过程会跳转到 catch 语句，然后把异常对象推入一个可变对象并置于作用域的头部。在 catch 代码块内部，函数的所有局部变量将会被放在第二个作用域链对象中。示例代码：

```
try{
    doSomething();
}catch(ex){
    alert(ex.message); //作用域链在此处改变
}
```

请注意，一旦 catch 语句执行完毕，作用域链机会返回到之前的状态。try-catch 语句在代码调试和异常处理中非常有用，因此不建议完全避免。你可以通过优化代码来减少 catch 语句对性能的影响。一个很好的模式是将错误委托给一个函数处理，例如：

```
try{
    doSomething();
}catch(ex){
    handleError(ex); //委托给处理器方法
}
```

优化后的代码，handleError 方法是 catch 子句中唯一执行的代码。该函数接收异常对象作为参数，这样你可以更加灵活和统一的处理错误。由于只执行一条语句，且没有局部变量的访问，作用域链的临时改变就不会影响代码性能了。

在这里多说一嘴，with 会影响作用域。面试的时候，面试官会问： 为什么不建议使用 with?

因为影响性能、减低代码的安全性、代码更加难于阅读。

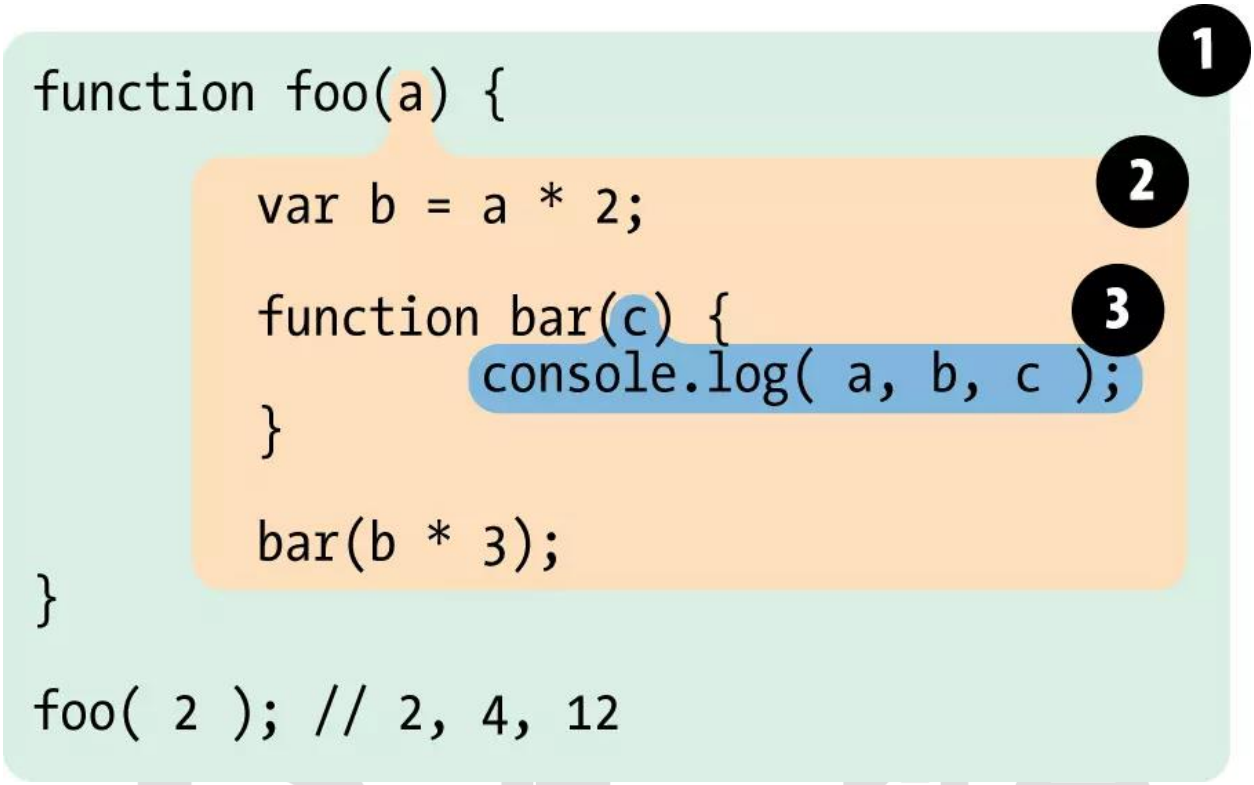
那为什么会影响性能？、为何会减低安全性？、怎么就让代码更加难于阅读？

这个些问题主要是考察 Js 开发人对词法作用域的理解。

先了解一下什么是词法作用域

简单地来说，词法作用域就是定义在词法阶段的作用域。词法作用域是由你在写代码时将变量和块作用域写在哪了来决定的，因此当词法分析器处理代码时会保存作用域不变

借助一张图来理解一下：



(1)、包含着整个全局作用域，其中只有一个标识符：foo
(2)、包含着 foo 所创建的作用域，其中有三个标识符： a、b、 bar
(3)、包含着 bar 所创建的作用域，其中只有一个标识符：c

从上面那张图片可以看到，作用域是逐级嵌套的。在执行 console.log(a, b, c)时，引擎会从最内部的作用域（3）开始查找，如果没有找到，就逐层往外找。作用域查找会在找到第一个匹配的标识符时停止。

为什么会影响性能呢？？？

这里就要提到“欺骗词法（代码在运行的时候"修改"词法作用域）”这个词， JavaScript 引擎在编译阶段有一些性能上的优化是依赖于能够根据代码的词法进行静态分析，并预先确定所有变量和函数的定义位置，才能在执行过程中快速的找到标识符。

但如果引擎在代码中发现了 with，它只能简单地假设关于标识符位置但判断都是无效的，因为无法在词法分析阶段明确的知道 eval 会接收到什么代码，这些代码是如何对作

用域进行修改，因此最简单的做法就是完全不做任何优化。如果代码中大量使用 with，导致引擎没有对这些进行优化。那么代码运行起来一定会变得更慢。

代码安全问题

with 它会在你不知的情况下把一些内部作用域声明的函数或者变量泄露到其他作用域，例如泄露到全局：

```
function a() {
        console.log('hello');
}
function foo(obj) {
        with(obj) {
                a = function() {
                        console.log("hi");
                };
 };
};
var obj = { b: 3};
a();
```


在 obj 创建了作用域没有找到 a，然后在 foo()作用域找，也没有找到、最后在全局作用域中找到了，然后就把它给改了。

代码更加难于阅读

这个就不难说明，如果传递给它们的参数都是不可预知的变量。那肯定难于阅读，因为你根本不知道代码下一步运行会不会改变这个变量。

总结

with 的本意是好，但它阻断了变量名的词法作用域绑定，导致的副作用远远大于它的使用价值，所以我们应该在开发中避免使用它。

闭包

闭包是 JAVASCRIPT 的高级特性，因为把带有内部变量引用的函数带出了函数外部，所以该作用域内的变量在函数执行完毕后的并不一定会被销毁，直到内部变量的引用被全部解除。所以闭包的应用很容易造成内存无法释放的情况。

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

这里所展示的让外层作用域访问内层作用域的技术便是闭包(Closure)。得益于高阶函数的应用，使 foo()函数的作用域得到`延伸`。foo()函数返回了一个匿名函数，该函数存在于 foo()函数的作用域内，所以可以访问到 foo()函数作用域内的 local 变量，并保存其引用。而因这个函数直接返回了 local 变量，所以在外层作用域中便可直接执行 bar()函数以获得 local 变量。

我们一定要做好闭包的管理

循环事件绑定、私有属性、含参回调等一定要使用闭包时，并谨慎对待其中的细节。

我们假设一个场景：有六个按钮，分别对应六种事件，当用户点击按钮时，在指定的地方输出相应的事件。

```
var btns = document.querySelectorAll('.btn'); // 6 elements
var output = document.querySelector('#output');
var events = [1, 2, 3, 4, 5, 6];
// Case 1
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = function(evt) {
    output.innerText += 'Clicked ' + events[i];
  };
}
/**这里第一个解决方案显然是典型的循环绑定事件错误。**/
// Case 2
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = (function(index) {
    return function(evt) {
      output.innerText += 'Clicked ' + events[index];
    };
  })(i);
}
```

/**第二个方案传入的参数是当前循环下标，而后者是直接传入相应的事件对象。事实上，后者更适合在大量数据应用的时候，因为在 JavaScript 的函数式编程中，函数调用时传入的参数是基本类型对象，那么在函数体内得到的形参会是一个复制值，这样这个值就被当作一个局部变量定义在函数体的作用域内，在完成事件绑定之后就可以对 events 变量进行手工解除引用，以减轻外层作用域中的内存占用了。而且当某个元素被删除时，相应的事件监听函数、事件对象、闭包函数也随之被销毁回收。**/

```
// Case 3
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = (function(event) {
    return function(evt) {
      output.innerText += 'Clicked ' + event;
    };
  })(events[i]);
}
```

闭包会带来内存泄露问题，对于 GC（Garbage Collection）垃圾回收机制来说也会产生性能问题。

我们编写大型的 JavaScript 程序时，没经验的程序员写出的程序肯定会出现大量的内存泄露，内存不能被回收，很容易导致程序崩溃。在 PC 上可能还表现的不太明显，但是在 ipad 等移动设备上面运行，也许你会发现，在程序运行过一段时间后，ipad 直接跳出浏览器，终止了你的程序。这种情况就说明你程序有严重的内存泄露，IOS 认为你的程序存在风险，终止了你的程序。

所以在做 Web 程序时，你必须关注你的内存使用情况，避免持续的增长。我们必须了解如果避免内存泄露，平时写程序的时候就应该有这个概念，不要等到项目快写完了，再来一次集中的性能优化，等到最后来做，那绝对会非常相当的痛苦.

内存泄露是指在 JavaScript 程序中的表现就是程序的内存得不到释放，持续的增长。释放内存的工作由 GC 来做，他的工作宗旨就是：“将不再使用的对象 KILL 掉，回收占有的内存”。这句话很简单，专业一点讲，就是将没有被任何变量引用的对象回收。GC 相当负责的工作着，无奈的是我们不配合 GC,随意的把对象赋给一些变量（不光光是外部变量，局部的变量也会犯罪，也会释放不掉！）。所以不是 GC 不聪明，是我们太傻太天真！因此，为了配合 GC 的工作，我们要坚决的将那些钉子户干掉，把对象将交给 GC，去除对象的所有引用就能让 GC 回收对象。所以使用闭包要慎重。