作用域

下面会提到几个概念：

* 函数对象的[[scope]]属性
* Scope Chain（作用域链）
* Execution Context（运行期上下文）
* Activation Object（激活对象）

函数对象的[[scope]]属性

JavaScript 中每个函数都都表示为一个函数对象（函数实例），既然是对象，就有相关的属性和方法。除了正常的属性，函数对象具有仅供 JavaScript 引擎内部使用，但不能通过代码访问的一系列内部属性。这些属性中，其中一个就是 [[scope]] 属性。

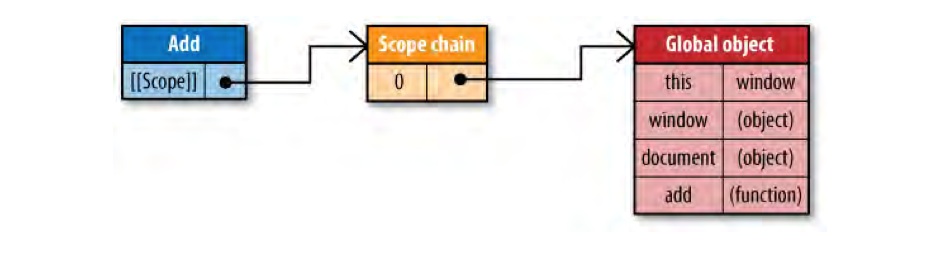
Scope Chain（作用域链）

内部的 [[scope]] 属性包含了该函数在被创建时作用域中的所有对象集合。该集合称为函数的作用域链（scope chain）。当创建一个函数时，其作用域链中保存的对象，就是在创建该函数时作用域中所有可访问的数据。例如，考虑以下全局函数：

JavaScript 代码:

1. function add(num1, num2) {
2. var sum = num1 + num2;
3. return sum;
4. }

当定义 add 函数后，其作用域链就创建了。函数所在的全局作用域的全局对象被放置到 add 函数作用域链（[[scope]] 属性）中。我们可以从下图中看到作用域链的第一个对象保存的是全局对象，全局对象中保存了诸如 this , window , document 以及全局对象中的 add 函数，也就是他自己。这也就是我们可以在全局作用域下的函数中访问 window(this)，访问全局变量，访问函数自身的原因。作用域链在稍后的执行函数时使用。当然还有函数作用域不是全局的情况，等会儿我们再讨论。



Execution Context（执行期上下文）

假设我们运行以下代码：

JavaScript 代码:

1. var total = add(5, 10);

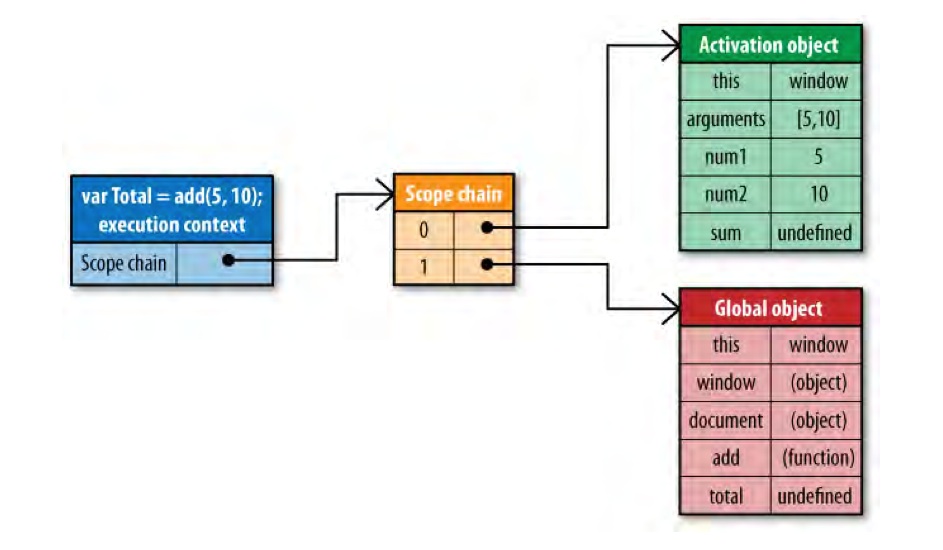
执行该函数创建一个内部对象，称为 Execution Context（执行期上下文）。执行期上下文定义了一个函数正在执行时的作用域环境。特别注意，执行期上下文和我们平常说的上下文不同，执行期上下文指的是作用域。平常说的上下文是this的取值指向。执行期上下文和函数创建时的作用域链对象 [[scope]] 区分，这是两个不同的作用域链对象。分开的原因很简单，函数定义时的作用域链对象 [[scope]] 是固定的，而 执行期上下文 会根据不同的运行时环境变化。而且该函数每执行一次，都会创建单独的 执行期上下文，因此对同一函数调用多次，会导致创建多个执行期上下文。一旦函数执行完成，执行期上下文将被销毁。

执行期上下文对象有自己的作用域链，当创建执期行上下文时，其作用域链将使用执行函数[[scope]]属性所包含的对象（即，函数定义时的作用域链对象）进行初始化。这些值按照它们在函数中出现的顺序复制到执行期上下文作用域链中。

Activation Object（激活对象）

随后，在执行其上下文中创建一个名为 Activation Object（激活对象）的新对象。 这个激活对象保存了函数中的所有形参，实参，局部变量，this 指针等函数执行时函数内部的数据情况。然后将这个激活对象推送到执行其上下文作用域链的顶部。

激活对象是一个可变对象，里面的数据随着函数执行时的数据的变化而变化，当函数执行结束之后，执行期上下文将被销毁。也就会销毁Execution Context的作用域链，激活对象也同样被销毁。但如果存在闭包，激活对象就会以另外一种方式存在，这也是闭包产生的真正原因，具体的我们稍后讨论。下图显示了执行上下文及其作用域链：



从左往右看，第一部分是函数执行时创建的执行期上下文，它有自己的作用域链，第二部分是作用域链中的对象，索引为1的对象是从[[scope]]作用域链中复制过来的，索引为0的对象是在函数执行时创建的激活对象，第三部分是作用域链中的对象的内容Activation Object(激活对象)和Global Object(全局对象)。

函数在执行时，每遇到一个变量，都会去执行期上下文的作用域链的顶部，执行函数的激活对象开始向下搜索，如果在第一个作用域链（即，Activation Object 激活对象）中找到了，那么就返回这个变量。如果没有找到，那么继续向下查找，直到找到为止。如果在整个执行期上下文中都没有找到这个变量，在这种情况下，该变量被认为是未定义的。这也就是为什么函数可以访问全局变量，当局部变量和全局变量同名时，会使用局部变量而不使用全局变量，以及 JavaScript 中各种看似怪异的、有趣的作用域问题的答案。

闭包（Closure）

闭包（Closure）是 JavaScript 最强大的特性之一。闭包在日常编码工作中非常常见。但是，它会对性能造成影响。了解闭包我们使用以下示例代码：

JavaScript 代码:

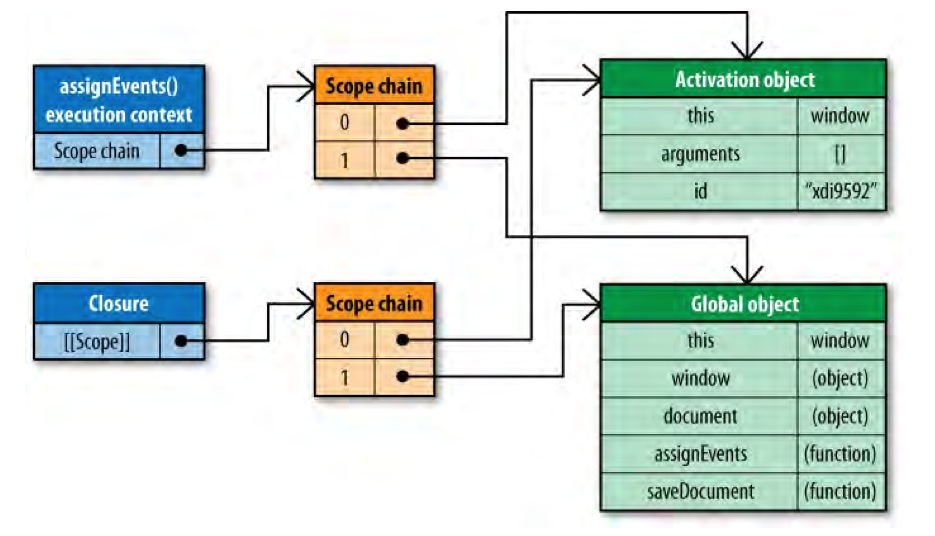
1. function assignEvents(){
2. var id = "xdi9592";
3. document.getElementById("save-btn").onclick = function(event) {
4. saveDocument(id);
5. };
6. }

assignEvents 函数为DOM元素分配一个事件处理程序。这个处理函数就是一个闭包。为了使该闭包访问id变量，必须创建一个特定的作用域链。

我们一起来从作用域的角度分析一下闭包的形成过程：

assignEvents 函数创建并且词法解析后，函数对象assignEvents的[[scope]]属性被初始化，作用域链形成，作用域链中包含了全局对象的所有属性和方法（注意，此时因为 assignEvents 函数还未被执行，所以闭包函数并没有被解析）。

assignEvents 开始执行时，创建 Execution Context（执行期上下文），在执行期上下文的作用域链中创建 Activation Object(激活对象)，并将 Activation Object(激活对象) 推送到作用域链顶部，在其中保存了函数执行时所有可访问函数内部的数据。激活对象包含 id 变量。



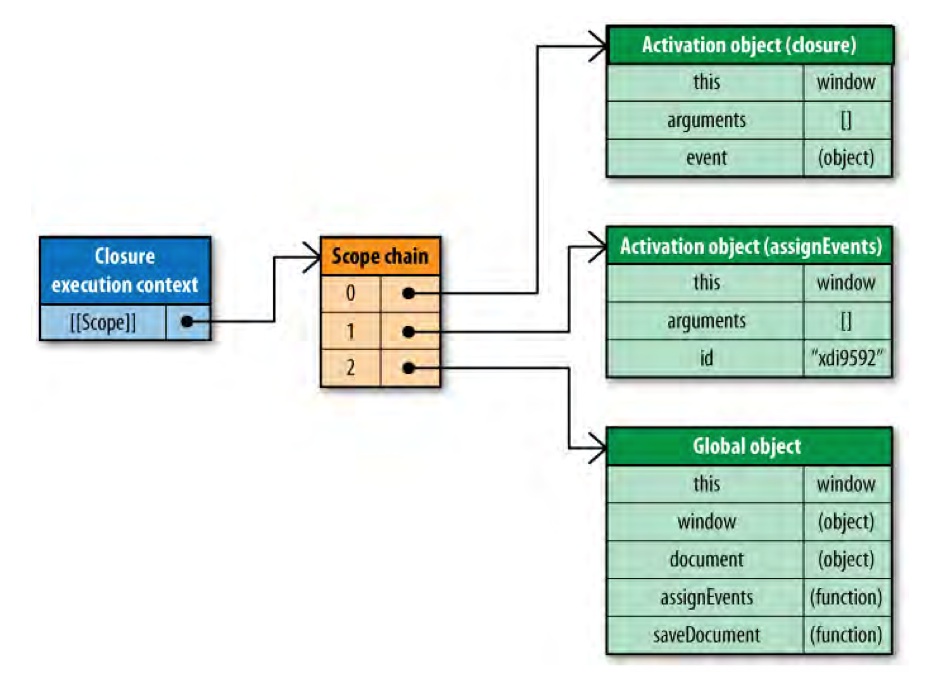
当执行到闭包时，JavaScript 引擎发现了闭包函数的存在，按照通常的手法，将闭包函数解析，为闭包函数对象创建 [[scope]] 属性，初始化作用域链。特别注意的是，这个时候，闭包函数对象的作用域链中有两个对象，一个是 assignEvents 函数执行时的 Activation Object(激活对象) ，还有一个是全局对象，如上图。

我们看到图中闭包函数对象的作用域链和 assignEvents 函数的执行期上下文的作用域链是相同的。为什么相同呢？我们来分析一下，闭包函数是在 assignEvents 函数执行的过程中被定义并且解析的，而函数执行时的作用域是 Activation Object(激活对象) ，闭包函数被解析的时候它的作用域正是 assignEvents 作用域链中的第一个作用域对象 Activation Object(激活对象) ，当然，由于作用域链的关系，全局对象作用域也被引入到闭包函数的作用域链中。

在词法分析的时候闭包函数的 [[scope]] 属性 就已经在作用域链中保存了对 assignEvents 函数的 Activation Object(激活对象) 的引用，所以当 assignEvents 函数执行完毕之后，闭包函数虽然还没有开始执行，但依然可以访问 assignEvents 的局部数据，并不是因为闭包函数要访问 assignEvents的局部变量id，所以当 assignEvents 函数执行完毕之后依然保持了对局部变量id的引用。而是不管是否存在变量引用，都会保存对 assignEvents 的 Activation Object(激活对象)作用域对象的引用。因为在词法分析时，闭包函数没有执行，函数内部根本就不知道是否要对 assignEvents 的局部变量进行访问和操作，所以只能先把 assignEvents 的 Activation Object(激活对象) 作用域对象保存起来，当闭包函数执行时，如果需要访问 assignEvents 的局部变量，那么再去作用域链中查找。

也正是因为这种引用，造成了一个副作用。通常，当执行期上下文被销毁时，函数的激活对象也就被销毁了。当有闭包引用时，激活对象就不会被销毁，因为他仍然被引用。这意味着闭包比非隔离的函数需要更多的内存。

闭包函数执行时创建了自己的 Execution Context（执行期上下文），其作用域链使用了 [[scope]]属性，其引用了 assignEvents 函数的 Activation Object(激活对象) 和 全局对象。然后为闭包本身创建一个新的 Activation Object(激活对象)。 所以在闭包函数的执行期上下文的作用域链中保存了自己的 Activation Object(激活对象)，外层函数 assignEvents Execution Context（执行期上下文）的 Activation Object(激活对象)，以及 Global Object(全局对象)，如图：



结论

JavaScript 引擎使用的内部hook(钩子)跟踪函数定义和执行期上下文的作用域链。 在函数执行时，变量标识符按照从上到下的顺序通过作用域链解析。 如果在最后没有找到相同的变量标识符，则抛出一个 undefined(未定义) 的错误。 闭包的开销是其的作用域链保持了对其执行期上下文的激活对象的引用，从而防止激活对象被正常地销毁。 因此，闭包函数代码通常比非闭包函数需要更多的内存。

 正文完。下面还有一个推广让最好的人才遇见更好的机会！

[](https://cn.100offer.com/how-it-works/?utm_source=css88&utm_medium=display&utm_campaign=css88_20170310&utm_content=find_new_job&campaign_code=css88)

互联网行业的年轻人，他们面对着怎样的职业瓶颈、困惑与未来选择？过去，这鲜有人关心。资深的职场人，也多半优先选择熟人去推荐机会。

100offer致力于改变现状，帮互联网行业最好的人才发现更好的机会。使用 [100offer.com](https://cn.100offer.com/how-it-works/?utm_source=css88&utm_medium=display&utm_campaign=css88_20170310&utm_content=find_new_job&campaign_code=css88) 或 100offer App ，可以一周内获得中国、美国等数千家优质企业的工作机会。