作用域与作用域链

讲解作用域前先理解以下概念

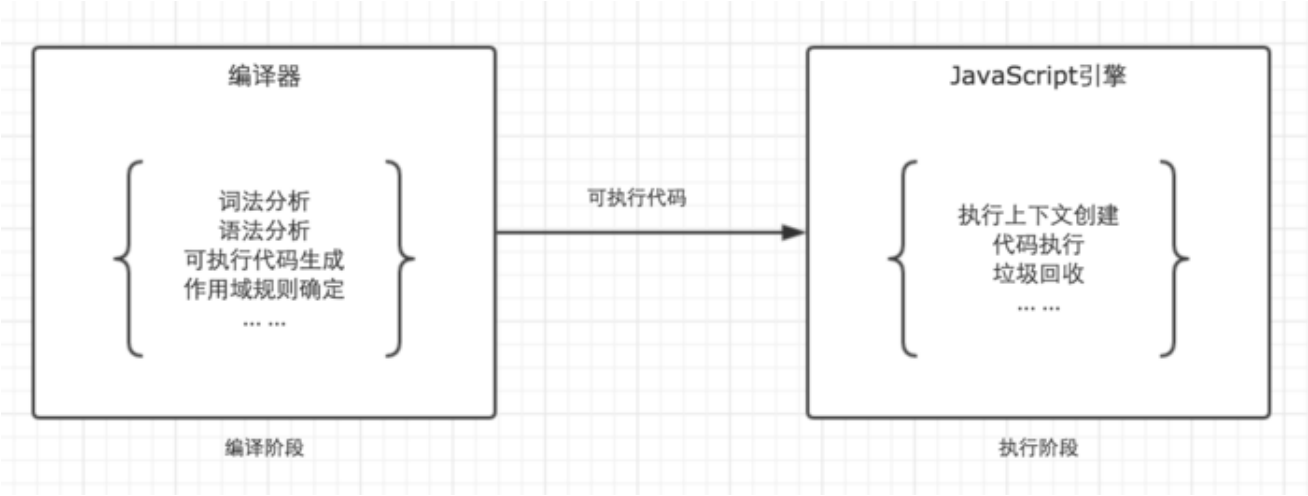
- 基础数据类型与引用数据类型
- 内存空间
- 垃圾回收机制
- 执行上下文
- 变量对象与活动对象

作用域

- 作用域是指程序源代码中定义变量的区域。
- 作用域规定了如何查找变量，也就是确定当前执行代码对变量的访问权限。
- `JavaScript` 采用词法作用域(`lexical scoping`)，也就是静态作用域。
- `JavaScript`中只有全局作用域与函数作用域(因为`eval`我们平时开发中几乎不会用到它，这里不讨论)。
- 作用域与执行上下文是完全不同的两个概念。

作用域与执行上下文产生的阶段

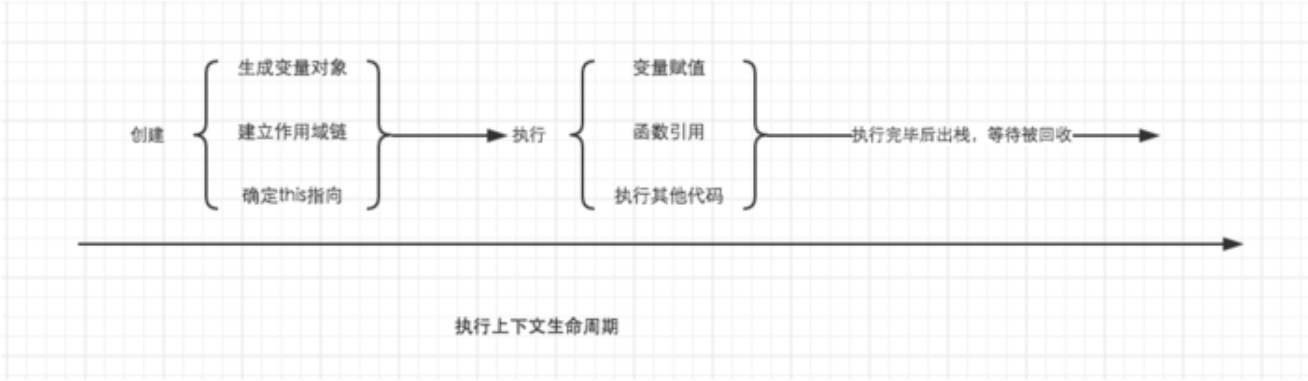
- `JavaScript`代码的整个执行过程，分为两个阶段，代码编译阶段与代码执行阶段。
- 编译阶段由编译器完成，将代码翻译成可执行代码，这个阶段**作用域规则**会确定。
- 执行阶段由引擎完成，主要任务是执行可执行代码，执行上下文在这个阶段创建。



区别：执行上下文在运行时才确定的，根据调用条件随时可能改变；作用域在定义时就确定了，永远不会改变。

作用域链

先看下执行上下文的生命周期



函数在调用激活时，会开始创建对应的执行上下文，在执行上下文生成的过程中，变量对象，作用域链，以及`this`的值会分别被确定。

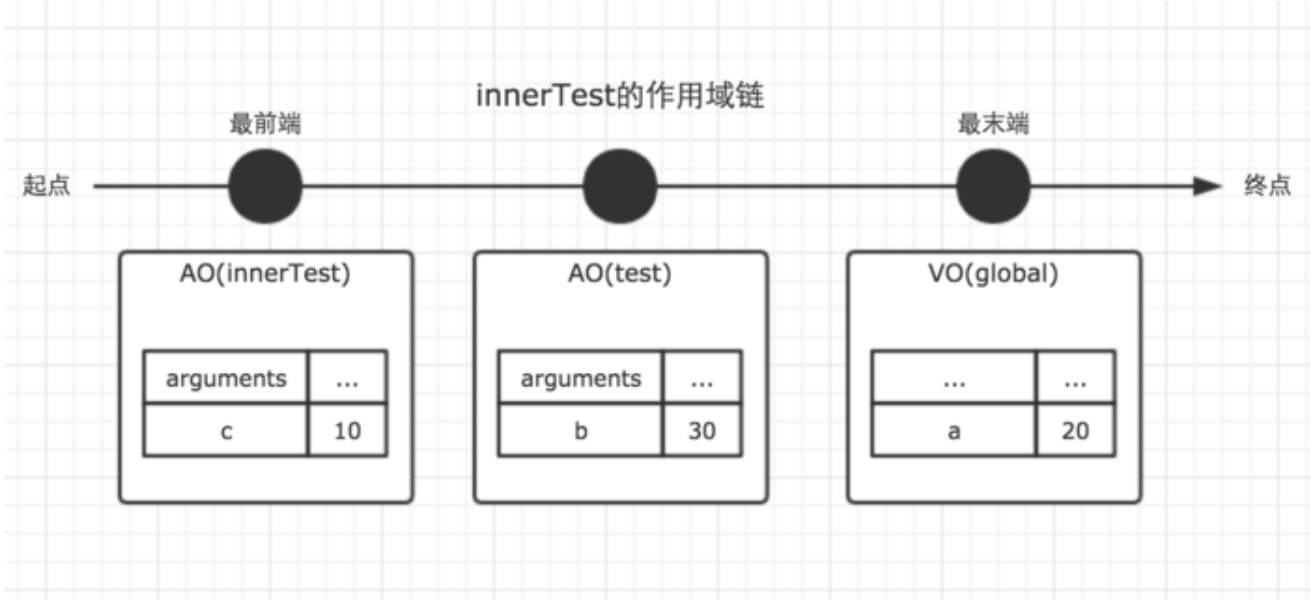
```
var a = 20;

function test() {
    var b = a + 10;

    function innerTest() {
        var c = 10;
        return b + c;
    }

    return innerTest();
}

test();
```

上面的例子中，全局，函数`test`，函数`innerTest`的执行上下文先后创建。我们设定他们的**变量对象**分别为`VO(global)`，`VO(test)`，`VO(innerTest)`。而`innerTest`的作用域链，则同时包含了这三个**变量对象**，所以`innerTest`的执行上下文可如下表示。

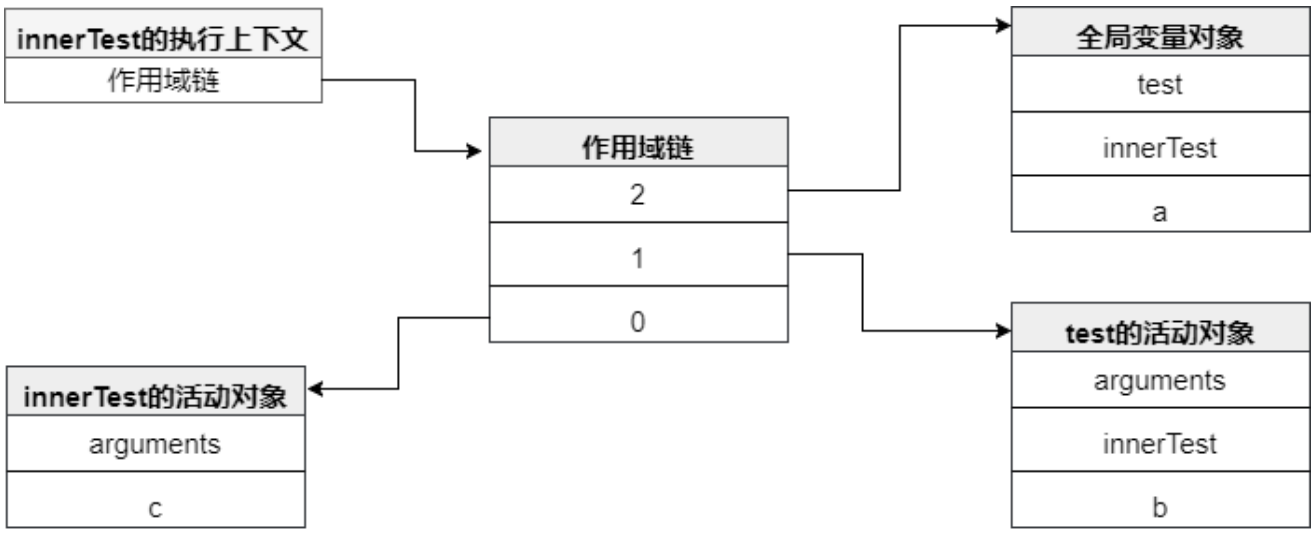
```
innerTestEC = {
    VO: {...},  // 变量对象
    scopeChain: [VO(innerTest), VO(test), VO(global)], // 作用域链
    this:Window
}
```

当函数调用激活时，进入函数上下文，创建`VO/AO`后，就会将活动对象添加到作用链的前端。

我们可以直接用一个数组来表示作用域链，数组的第一项`scopeChain[0]`为作用域链的最前端，而数组的最后一项，为作用域链的最末端，所有的最末端都为全局变量对象。



因为变量对象在执行上下文进入执行阶段时，就变成了活动对象，因此图中使用了`AO（Active Object ）`来表示。**作用域链就是由一系列变量对象组成的**，我们可以在这个单向通道中，查询变量对象中的标识符，这样就可以访问到上一层作用域中的变量了。（变量对象和活动对象是处于不同时期的同一个对象）



闭包

- **闭包是一种特殊的对象。它由两部分组成。执行上下文(代号A)，以及在该执行上下文中创建的函数（代号B）。当B执行时，如果访问了A中变量对象中的值，那么闭包就会产生。**
- **在大多数理解中，包括许多著名的书籍，文章里都以函数B的名字代指这里生成的闭包。而在chrome中，则以执行上下文A的函数名代指闭包。**

> MDN:闭包是由函数以及创建该函数的词法环境组合而成。

因此我们只需要知道，一个闭包对象，由A、B共同组成 。

```
function foo() {
    var a = 20;
    var b = 30;

    function baz() {
        return a + b;
    }

    return baz;
}

var bar = foo();
bar();
```

上面的例子，首先有执行上下文`foo`，在`foo`中定义了函数`baz`，而通过对外返回`baz`的方式让`baz`得以执行。当`bar`执行时，访问了`foo`内部的变量`a`，`b`。因此这个时候闭包产生。

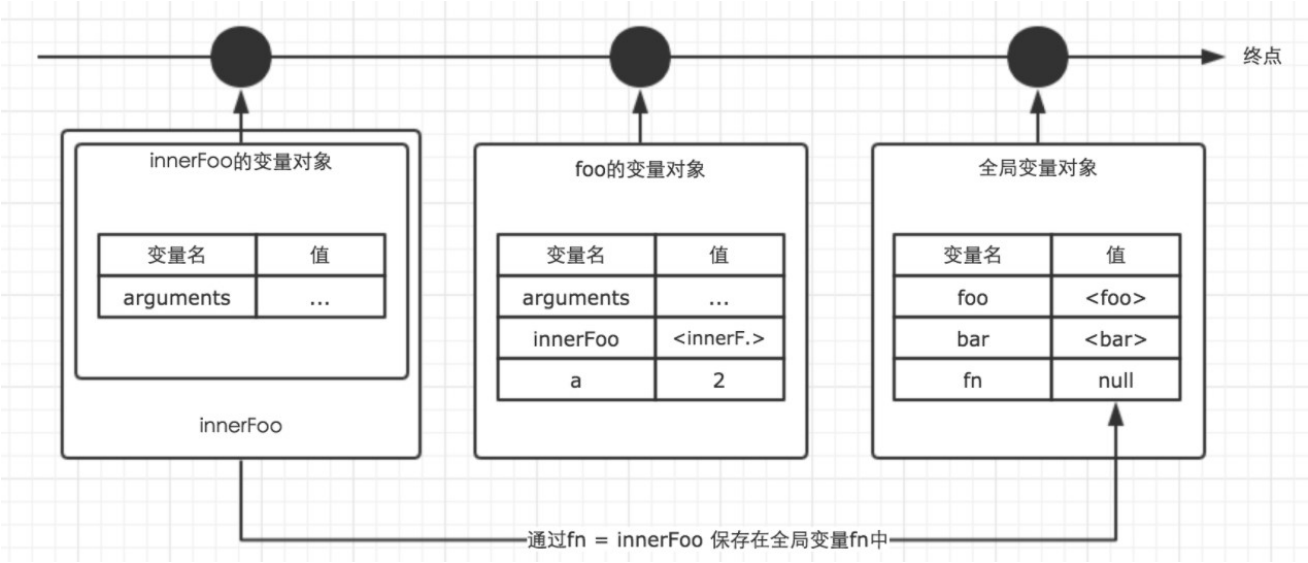
`JavaScript`拥有自动的垃圾回收机制，关于垃圾回收机制，有一个重要的行为，那就是，当一个值，在内存中失去引用时，垃圾回收机制会根据特殊的算法找到它，并将其回收，释放内存。

我们知道，函数的执行上下文，在执行完毕之后，生命周期结束，那么该函数的执行上下文就会失去引用。其占用的内存空间很快就会被垃圾回收器释放。可是闭包的存在，会阻止这一过程。

```
var fn = null;
function foo() {
    var a = 2;
    function innnerFoo() {
        console.log(a);
    }
    fn = innnerFoo; // 将 innnerFoo的引用，赋值给全局变量中的fn
}

function bar() {
    fn(); // 此处的保留的innerFoo的引用
}

foo();
bar(); // 2
```

`foo()`执行完毕之后，按照常理，其执行环境生命周期会结束，所占内存被垃圾收集器释放。但是通过`fn = innerFoo`，函数`innerFoo`的引用被保留了下来，复制给了全局变量`fn`。这个行为，导致了`foo`的变量对象，也被保留了下来。于是，函数`fn`在函数`bar`内部执行时，依然可以访问这个被保留下来的变量对象。所以此刻仍然能够访问到变量`a`的值。

这样，我们就可以称`foo`为闭包。下面是**闭包`foo`的作用域链**



所以，通过闭包，我们可以在其他的执行上下文中，访问到函数的内部变量。 比如在上面的例子中，我们在函数 `bar` 的执行环境中访问到了函数 `foo` 的 `a` 变量。