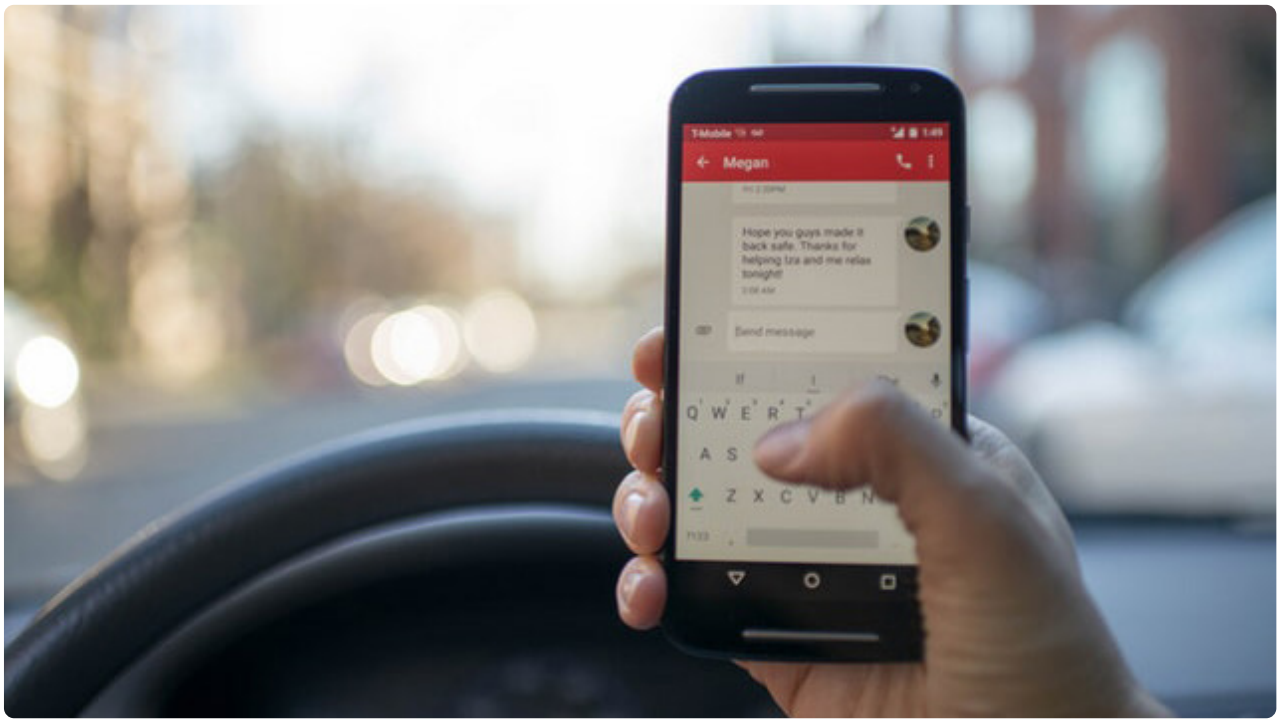
09 JS 更进一步：执行上下文与调用栈

更新时间：2020-03-20 18:04:53



> 困难只能吓倒懦夫懒汉，而胜利永远属于敢于等科学高峰的人。——茅以升

前面我们梳理了作用域、闭包、this 机制等一系列的实用知识点。不知道大家是否还记得，我们之前所说过的，“块状知识” 学习的最佳路径 —— 把知识放在完整的链路体系里去理解。

在 JS 核心知识这块，尤其是如此。

我们会发现，当我们想要深入理解闭包的时候，就不得不去探索作用域的世界，当我们试图去掌握 this 的时候，又不得不把它和词法作用域规则放在一起来看、以求一个更深入的理解。

总感觉这些知识点与点之间，好像有着说不清、道不明的联系。事实上，它们之间的关联确实不止于此。闭包也好、this 也罢，我们完全可以把它们放在一个完整的知识链路里来理解，那就是 **JS 的执行上下文**。

本节我们就以 JS 执行上下文的学习为契机，把前面所学的这些点串成串、寻得它们之间更深层次的内在联系，帮助大家更深入地把握 JS 这门语言的本质。

此外，站在面试官的角度来看，执行上下文及其相关的调用栈层面的知识，已经是 JS 语言类考题里面非常深入和底层的考察点。相信各位在结束了本节的学习后，面对任何类似的问题都能够言之有物、游刃有余。

为什么要有执行上下文

大家平时写项目的时候，肯定是一个文件一个文件去写；具体到每一个文件里，又会细分出不同的方法、模块 —— 我想应该不会有同学会把成千上万行的庞大的代码逻辑塞进一个文件里。当大家这样做的时候，其实已经在践行一种软件世界里非常重要的思想 —— **分治**。

分治是编写软件的一种策略，它意味着你会把一个庞大的问题拆分成若干个具体的小问题，然后逐个去解决它们，以此来化解问题的复杂度。表现在代码上，就是把庞大的逻辑拆分成独立的代码块。这些 “代码块” 根据粒度的不同，有着不同的名字，它可以是函数、模块、包等等等等。

如果说把代码逻辑划分成 “块”，是我们程序员在编写阶段的智慧。那么把庞大的执行任务划分成不同的执行上下文，就是 JS 引擎在执行阶段的智慧了。

我们可以把执行上下文理解为引擎在执行过程中对代码进行了又一次的 “划分”，这样做的目的，仍然是为了分解复杂度。

执行上下文是什么

执行上下文，从定义上理解，是 “执行代码的环境”—— 这是一个专业且抽象的定义。从学习的层面来说，我更推荐大家从执行上下文的分类、组成和生命周期等具体的维度去理解它。

执行上下文的分类

执行上下文主要分为三类：

- 全局上下文 —— 全局代码所处的环境，不在函数中的代码都在全局执行上下文中
- 函数上下文 —— 在函数调用时创建的上下文
- Eval 执行上下文 —— 运行 Eval 函数中的代码时所创建的环境，Eval 被前端诟病多年，时下对 Eval 感兴趣的人非常少了，面试官也普遍对它嗤之以鼻。大家答题时只需要说明 “我不用 Eval”，直接跳过这个东西就好了（还可以拉一波好感度，说明你是一个明辨是非的好孩子）。综上所述，Eval 执行上下文，不在我们本文的讨论范围内。

接下来我们就着重讲讲全局上下文和函数上下文。

全局上下文的创建和组成

当我们的 JS 脚本跑起来之后，第一个被创建的执行上下文就是全局上下文。

当我们的脚本里一行代码也没有的时候里，全局上下文里会比较干净，只有两个东西：

- 全局对象（浏览器里是 Window, Node 环境下是 Global）
- this 变量。这里的 this ，指向的还是全局变量

但只要你往里面写了点东西，这个世界就会热闹起来了，比如你要敢随手写这么几句：

```
var name = 'xiuyan'
var tel = '123456'

function getMe() {
  return {
    name: name,
    tel: tel
  }
}
```

全局上下文的组成就会立刻丰富成下面这个样子：

有同学就急了，我明明给 name 和 tel 都赋值了，它咋还是 undefined 呢？这里就要引出上下文的一个生命周期了，每一个执行上下文都会经历这样一个生命周期：

- **创建阶段 —— 执行上下文的初始化状态，此时一行代码都还没有执行，只是做了一些准备工作**
- **执行阶段 —— 逐行执行脚本里的代码**

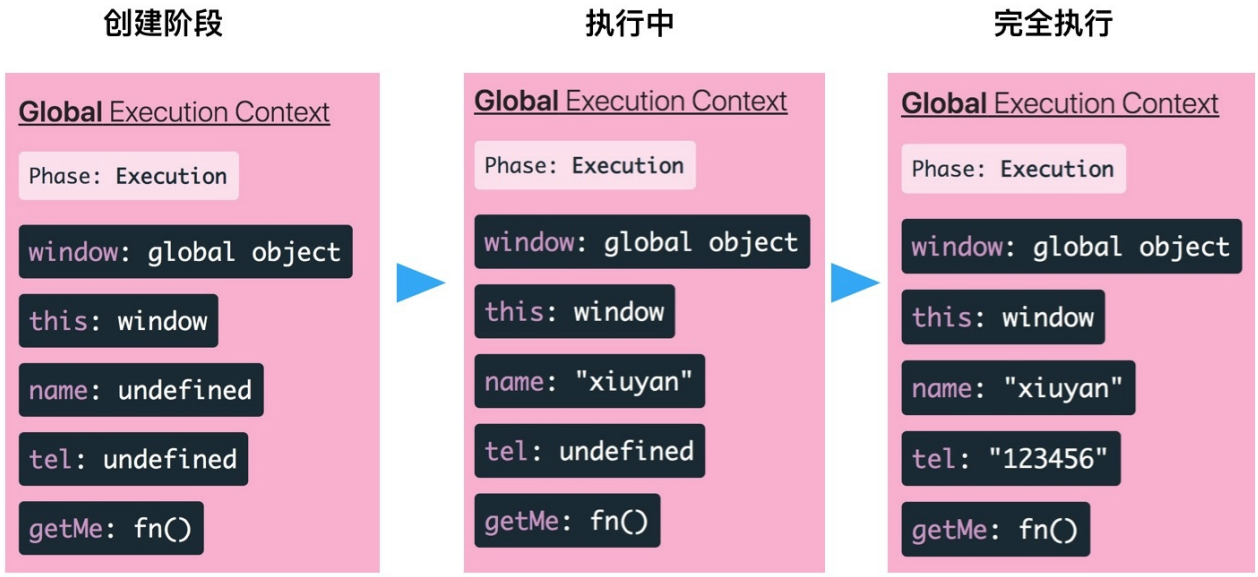
如图所示，各位看到的就是**创建阶段**的全局上下文概览。为啥这时候变量没有值呢？这是因为创建阶段里，JS 引擎不多不少只做这么几件事：

- 创建全局对象（Window 有了）
- 创建 this ，并让它指向全局对象
- 给变量和函数安排内存空间
- 默认给变量赋值为 undefined；将函数声明放入内存
- 创建作用域链

到这里为止， 真正的赋值动作都还没有执行。所以大家先别急，我们接下来赶紧瞅一眼**执行阶段**的全局上下文是什么光景：

这时候我们看到该有值的都有值了，这是因为 JS 引擎已经在一行一行执行代码、执行赋值操作了。

需要大家注意的是，**执行上下文在执行阶段里其实始终是处在一个动态**，比如说你执行完第一行没执行第二行的时候，这时候就只有 name 有值了， 而 tel 还是 undefined；再往下执行一行， tel 也有值了，就会看到执行上下文的内容又变了（从创建到执行完全的过程如下图）。



站在上下文角度，理解“变量提升”的本质

之前大家理解“变量提升”，或许更多的是靠记忆。

书本里会告诉你，在非严格模式下，当我们在没有声明一个变量就去调用它时，会出现这样的现象：

```
// 没有报错，而是输出 undefined
console.log(name)

var name = 'xiuyan'
```

JS 引擎不会抛出变量未声明的错误，而是会输出一个 undefined 值，表现得好像这个 name 变量早已被声明过一样。像这样的现象，我们叫它“变量提升”。

现在结合我们的上下文创建过程，你会知道，其实根本不存在任何的“提升”，变量一直在原地。所谓的“提升”，只是变量的创建过程（在上下文创建阶段完成）和真实赋值过程（在上下文执行阶段完成）的不同步带来的一种错觉。执行上下文在不同阶段完成的不同工作，才是“变量提升“的本质。

函数上下文的创建和组成

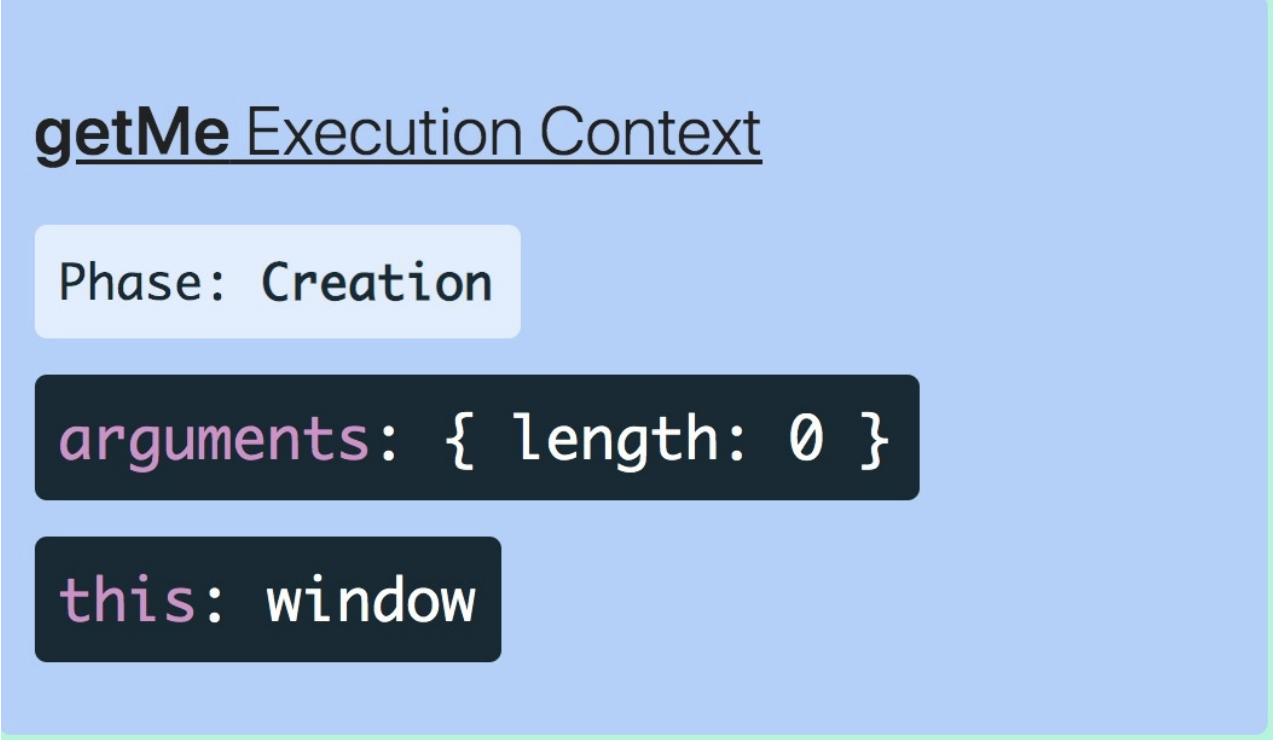
如果各位充分理解了前面全局上下文的工作流程，那么函数上下文对你来说就不再是什么难题。它在机制层面和全局上下文高度一致，各位只需要关注它与全局上下文之间的不同即可。**两者之间的不同主要体现在以下方面上：**

- 创建的时机 —— 全局上下文在进入脚本之初就被创建，而函数上下文则是在函数调用时被创建
- 创建的频率 —— 全局上下文仅在代码刚开始被解释的时候创建一次；而函数上下文由脚本里函数调用的多少决定，理论上可以创建无数次
- 创建阶段的工作内容不完全相同 —— 函数上下文不会创建全局对象（Window），而是创建参数对象（arguments）；创建出的 this 不再死死指向全局对象，而是取决于该函数是如何被调用的 —— 如果它被一个引用对象调用，那么 this 就指向这个对象；否则，this 的值会被设置为全局对象或者 undefined（在严格模式下）

除此之外，我们完全可以像理解全局上下文一样来理解函数上下文。我们仍然用一个简单的例子来看看函数上下文在不同阶段的表现：

```
var name = 'xiuyan'
var tel = '123456'

function getMe() {
    return {
        name: name,
        tel: tel
    }
}

// 增加了函数调用
getMe()
```

当引擎执行到 getMe () 调用这一行时，首先会进入函数上下文的创建阶段，在这个阶段里，函数上下文的内容如下：



接着进入执行阶段，逐行执行函数内部的代码。此处我们只有一行代码，在代码执行过程中，没有涉及到变量的修改，因此函数上下文的内容保持不变。执行完毕后，函数上下文的生命周期就结束了。

调用栈

我们看到函数执行完毕后，其对应的执行上下文也随之消失了。这个消失的过程，我们叫它”出栈“—— 没错，在 JS 代码的执行过程中，引擎会为我们创建”执行上下文栈“（也叫调用栈）。

因为函数上下文可以有许多个，我们不可能保留所有的上下文。当一个函数执行完毕，其对应的上下文必须让出之前所占用的资源。因此上下文的建立和销毁，就对应了一个” 入栈 “和” 出栈 “的操作。当我们调用一个函数的时候，就会把它的上下文推入调用栈里，执行完毕后出栈，随后再为新的函数进行入栈操作。

我们通过一个例子来理解一下这个过程：

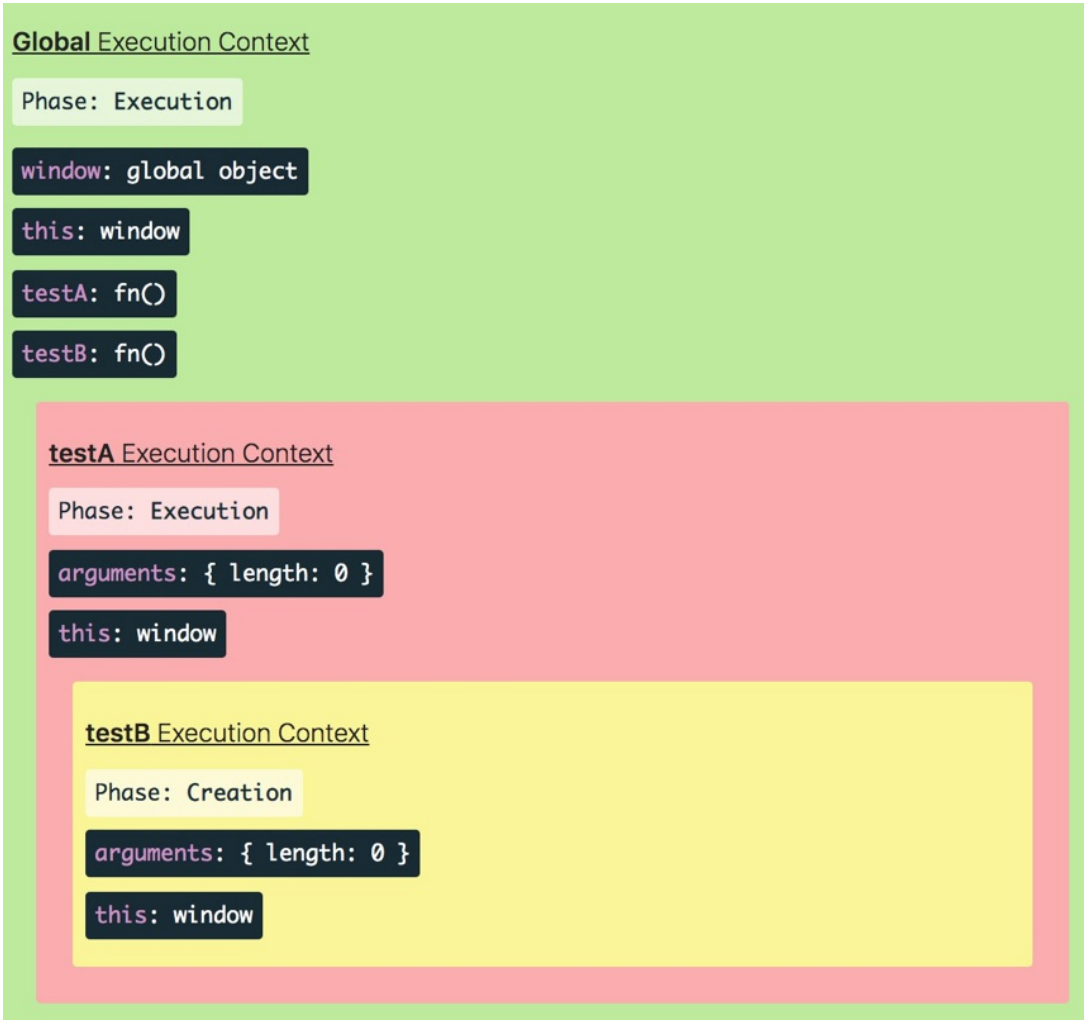
```
function testA() {
  console.log('执行第一个测试函数的逻辑');
  testB();
  console.log('再次执行第一个测试函数的逻辑');
}

function testB() {
  console.log('执行第二个测试函数的逻辑');
}

testA();
```

以上这个脚本的执行上下文栈，随着代码的执行，会经历一个这样的过程：

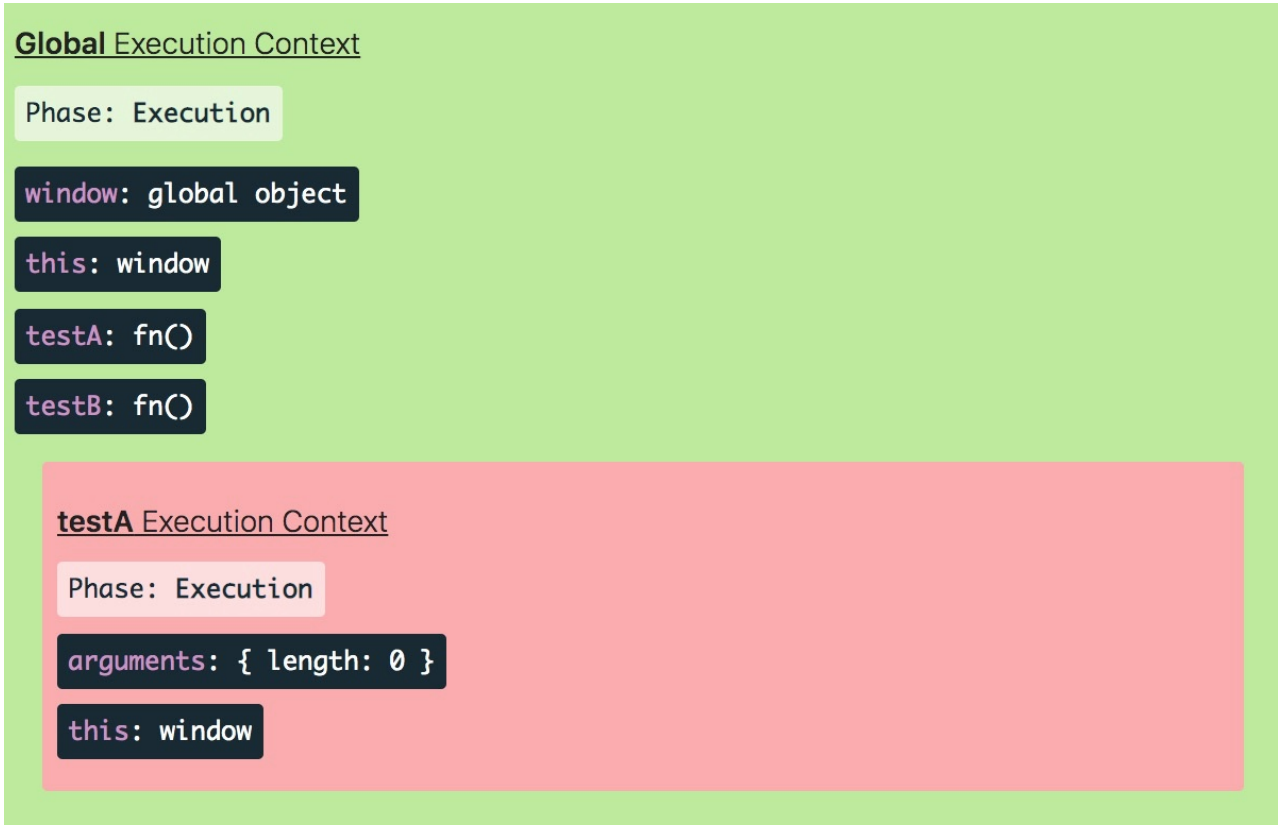
1. 执行之初，全局上下文创建：

2. 执行到 testA 调用处，testA 对应的函数上下文创建：

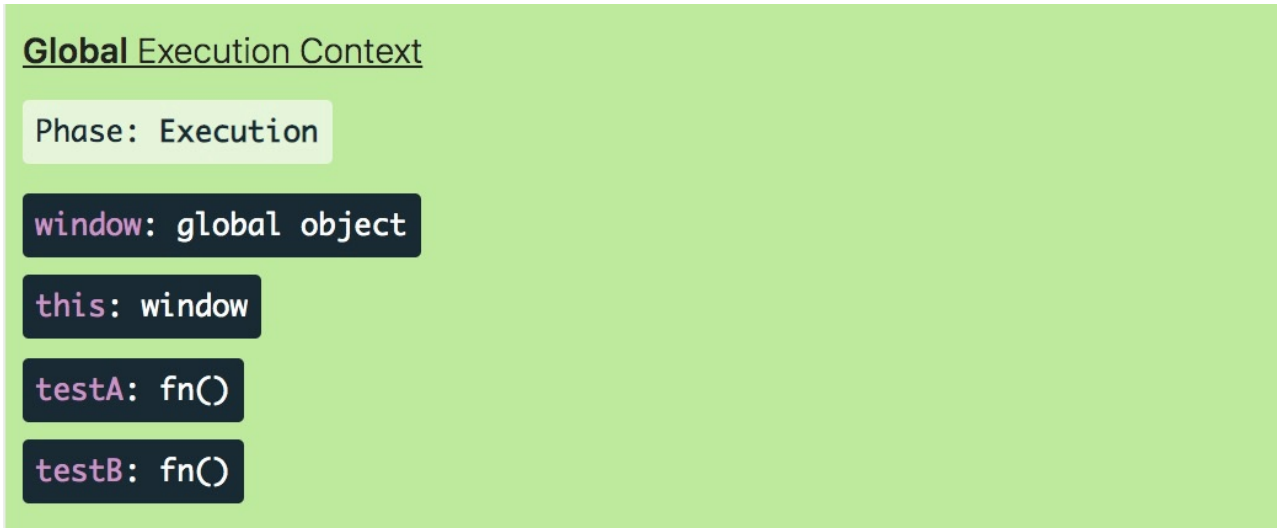
3. 执行到 testB 处，testB 对应的函数上下文创建：



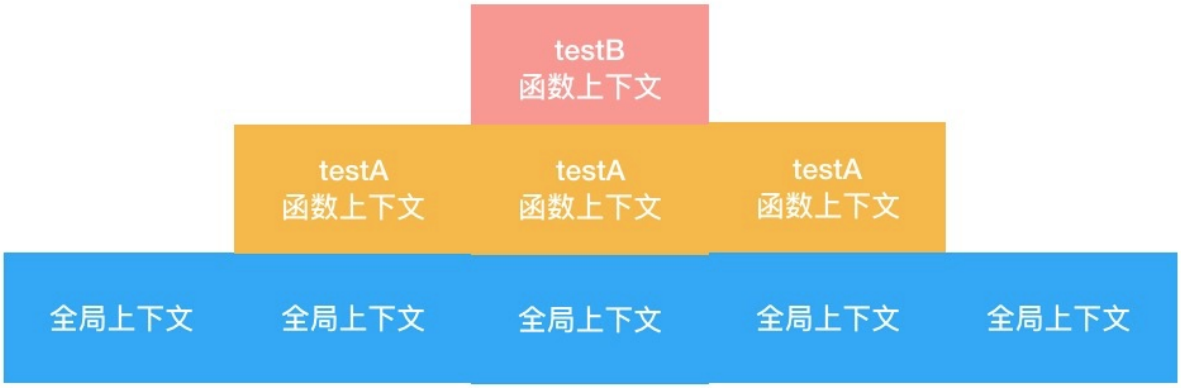
4. testB 执行完毕，对应上下文出栈，剩下 testA 和 全局上下文：



5. testA 执行完毕，对应执行上下文出栈，剩下全局上下文：



在这整个过程里，调用栈的变化示意如下：



站在调用栈的角度，理解作用域的本质

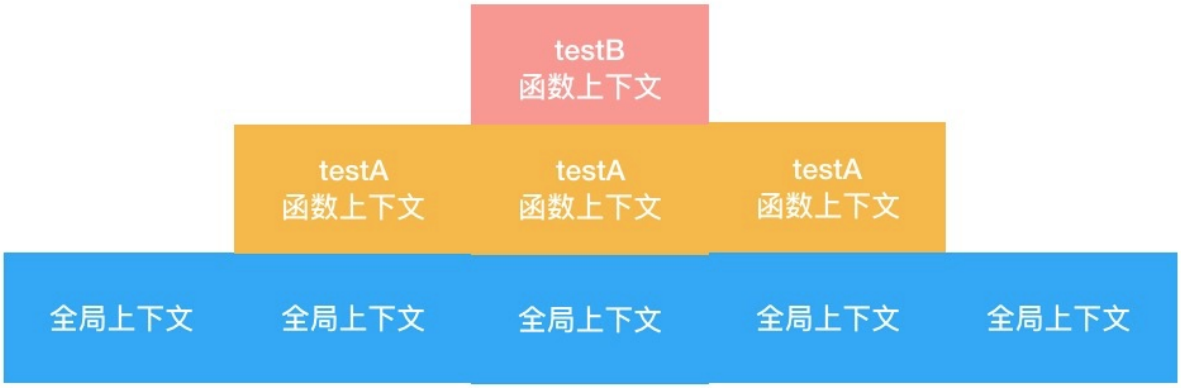
作用域是什么？之前，我们认为作用域是” 访问变量的一套规则 “。但现在我要告诉大家，作用域其实就是**当前所处的执行上下文**。我们基于执行上下文，来理解一下作用域的特征：

作用域对外隔离

我们仍然用这段代码来举例：

```
function testA() {
  console.log('执行第一个测试函数的逻辑');
  testB();
  console.log('再次执行第一个测试函数的逻辑');
}

function testB() {
  console.log('执行第二个测试函数的逻辑');
}

testA();
```

这里，全局作用域 相对于 testA 的函数作用域，它是外部作用域；全局作用域、testA 相对于 testB 的函数作用域，它们都是外部作用域。我们知道，作用域在嵌套的情况下，外部作用域是不能访问内部作用域的变量的。现在，结合调用栈的情况，相信你会更清楚这其中的原因：



以 testB 为例，我们看到，最初处于外部作用域（testA、全局上下文）时，testB 对应的上下文还没有被推入调用栈；而当 testB 执行结束、代码执行退回到外部作用域时，testB 早已从栈顶弹出。这意味着，每次位于外部作用域时，testB 的执行上下文都压根不存在于调用栈内。此时就算 testA 函数上下文和全局上下文无论如何也找不到任何关于 testB 的线索，自然访问不到它内部的变量啦！

闭包 —— 特殊的“弹出”

一般来说，函数出栈后，我们都没有办法再访问到函数内部的变量了。但闭包可不是这样：

```
function outer(a) {
  return function inner (b) {
    return a + b;
  };
}

var addA = outer(10);

addA(20)
```

在这个例子里，inner 函数引用了 outer 函数的自由变量 a 变量，形成了一个闭包。在 outer 函数执行完毕出栈后，实际上 inner 函数仍然可以访问到这个 a 变量 ——a 变量好像没用随着 outer 函数执行上下文的消失而消失，这是为什么呢？

大家别忘了，在执行上下文的创建阶段，跟着被创建的还有作用域链！这个作用域链在函数中以内部属性的形式存在，在函数定义时，其对应的父变量对象就会被记录到这个内部属性里。闭包正是通过这一层作用域链的关系，实现了对父作用域执行上下文信息的保留。

自由变量的查找 —— 作用域链与上下文变量的结合

前面咱们讲了外部作用域难以“触及”内部作用域的原因。但反过来看，站在函数作用域内部，却可以访问到外部作用域的变量，这又是为啥呢？我们稍微改一下第一小节的代码：

```
var name = 'xiuyan'

function testA() {
  console.log('执行第一个测试函数的逻辑');
  testB();
  console.log('再次执行第一个测试函数的逻辑');
}

function testB() {
  console.log(name);
}

testA();
```

仍然说回 testB。我们看到，当代码执行到 testB 这个位置时，它位于调用栈的栈顶，此时 testA 和全局上下文都稳稳地坐在调用栈底部 —— 这首先为 testB 查找到自由变量创造了可能性。

在执行阶段，如果像例子中的 testB 一样，在函数作用域内部找不到 name 这个变量，那么引擎会沿着作用域链向上找、定位到它对应的父级作用域的上下文、看有没有目标变量，如果还没有，那么就沿着作用域链继续往上定位、直到找到为止。

注意！这里是沿着作用域链找，可不是沿着调用栈一层一层往上找哦！调用栈是在执行的过程中形成的，而作用域链可是在书写阶段就决定了。因此，testB 里找不到的变量，绝不会去 testA 里找，而是去全局上下文变量里找！

}

08 改变 this 指向、深入理解 call/apply/bind 的原理

10 原型编程范式与面向对象