

# 目录

Introduction	1.1
--------------	-----

## 【 JS内功修炼 】

### 第一节

专业术语	3.1
执行上下文	3.2
作用域	3.3
闭包	3.4
this	3.5
相关面试题	3.6

# 爪哇教育-前端课程

- 讲师：袁鑫
- 2019-8

# JS内功修炼

## 专业术语

- 常量、变量、数据类型
- 形参、实参
- 匿名函数、具名函数、自执行函数
- 函数声明、函数表达式
- 堆、栈
- 同步、异步、进程、线程

## 执行上下文

当函数执行时，会创建一个称为执行上下文（execution context）的环境，分为创建和执行2个阶段

### 创建阶段

创建阶段，指函数被调用但还未执行任何代码时，此时创建了一个拥有3个属性的对象：

```
executionContext = {
  scopeChain: {}, // 创建作用域链 (scope chain)
  variableObject: {}, // 初始化变量、函数、形参
  this: {} // 指定this
}
```

### 代码执行阶段

代码执行阶段主要的工作是：1、分配变量、函数的引用，赋值。2、执行代码。

举个栗子

```
// 一段这样的代码
function demo(num) {
  var name = 'xiaowa';
  var getData = function getData() {};
  function c() {}
}
demo(100);

// 创建阶段大致这样，在这个阶段就出现了【变量提升(Hoisting)】
executionContext = {
  scopeChain: { ... },
  variableObject: {
```

```

    arguments: { // 创建了参数对象
        0: 100,
        length: 1
    },
    num: 100, // 创建形参名称，赋值/或创建引用拷贝
    c: pointer to function c() // 有内部函数声明的话，创建引用指向函数体
    name: undefined, // 有内部声明变量a，初始化为undefined
    getData: undefined // 有内部声明变量b，初始化为undefined
},
this: { ... }
}

// 代码执行阶段，在这个阶段主要是赋值并执行代码
executionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 100,
            length: 1
        },
        num: 100,
        c: pointer to function c()
        name: 'xiaowa', // 分配变量，赋值
        getData: pointer to function getData() // 分配函数的引用，赋值
    },
    this: { ... }
}

```

## 执行上下文栈

- 浏览器中的JS解释器是单线程的，相当于浏览器中同一时间只能做一个事情。
- 代码中只有一个全局执行上下文，和无数个函数执行上下文，这些组成了执行上下文栈（Execution Stack）。
- 一个函数的执行上下文，在函数执行完毕后，会被移出执行上下文栈。

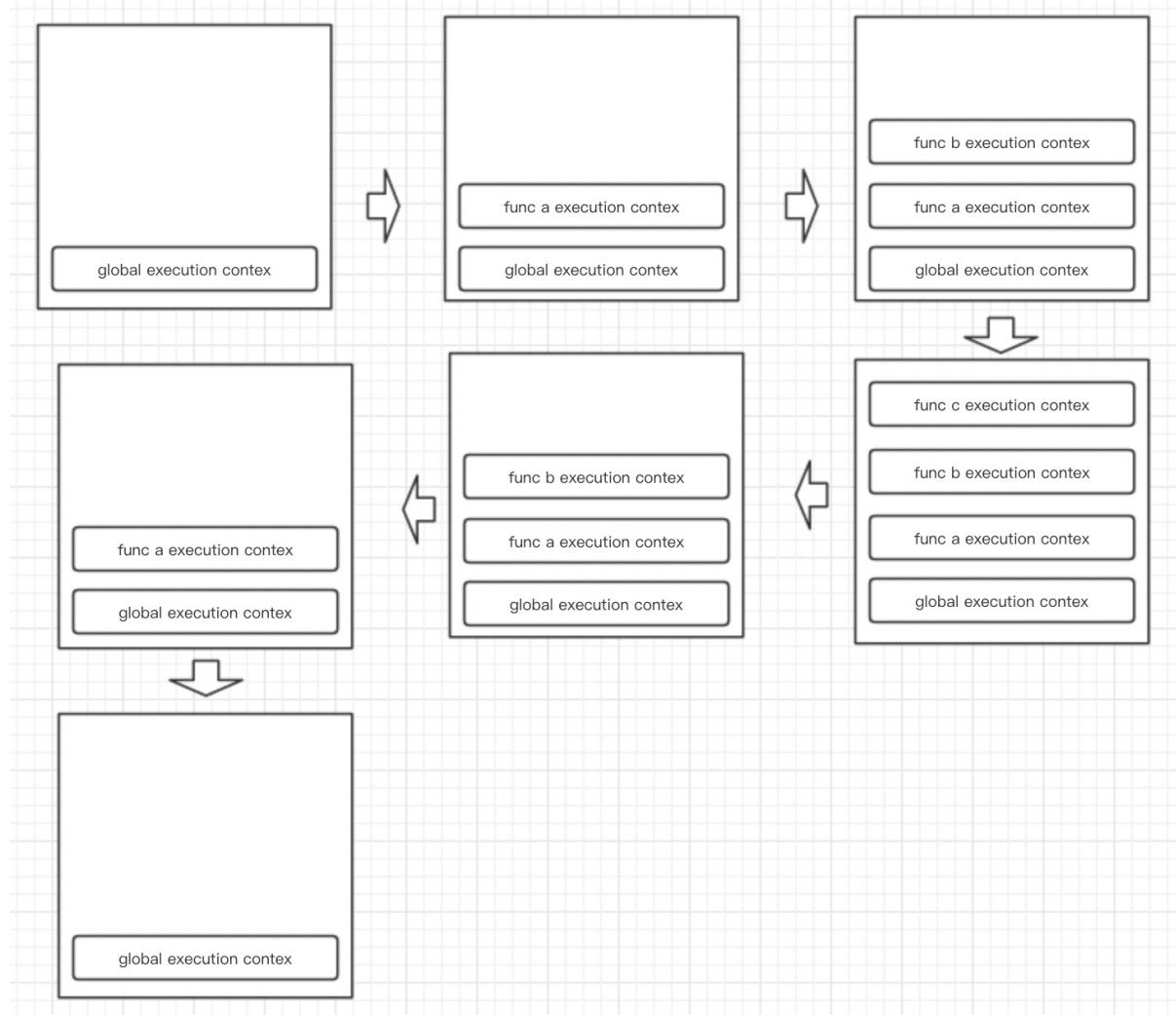
举个栗子

```

function c(){
    console.log('ok');
}
function a(){
    function b(){
        c();
    }
    b();
}
a();

```

这个栗子的执行上下文栈是这样的



## 作用域

js中有全局作用域、函数作用域，es6中又增加了块级作用域。作用域的最大用途就是隔离变量或函数，并控制他们的生命周期。作用域是在函数执行上下文创建时定义好的，不是函数执行时定义的。

举个栗子

```
// 不要看晕了哦~
function a () {
    return function b() {
        var myname = 'b';
        console.log(myname); // b
    }
}
function c() {
    var myname = 'c';
    b();
}
```

```

var b = a();
c();

// 去掉函数b中的myname声明后
function a () {
    return function b() {
        // var myname = 'b';
        console.log(myname); // 这里会报错
    }
}
function c() {
    var myname = 'c';
    b();
}
var b = a();
c();

```

## 作用域链

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。在当前函数中如果js引擎无法找到某个变量，就会往上一级嵌套的作用域中去寻找，直到找到该变量或抵达全局作用域，这样的链式关系就称为作用域链(Scope Chain)

## 闭包

高级程序设计三中:闭包是指有权访问另外一个函数作用域中的变量的函数.可以理解为(能够读取其他函数内部变量的函数)

wiki百科的解释: [https://en.wikipedia.org/wiki/Closure\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

In programming languages, a closure, also lexical closure or function closure, is a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment.[1] The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] Unlike a plain function, a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

```

function outer() {
    var top = xxxx;
    function inner() {
        xxx.innerHTML = top;
    }
}

```

平时用在哪儿?

## 1、封装私有变量(amd的框架等都使用)

```
// 普通的定义类的方式
function Person() {
    this._attackVolume = 100;
}

Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};

var person = new Person();
console.log(person._attackVolume);

// 工厂方法
function Person() {
    var _attackVolume = 100;
    return {
        attack() {
            body.bloodVolume -= this.attackVolume - body.defenseVolume;
        }
    };
}

var person = new Person();
console.log(person._attackVolume);
```

## 2、存储变量

```
// 封装的时候
function getListDataManager() {
    // 外层scope中定义一个变量
    let localData = null;

    return {
        getData() {
            // 里面的函数使用外层的变量，而且是反复使用
            if (localData) {
                return Promise.resolve(localData);
            }
            return fetch('xxxx')
                .then(data => localData = data.json());
        }
    };
}

// 用的时候
```

```
const listDataManager = getListDataManager();

button.onclick = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};

window.onscroll = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};
```

## this

一共有5种场景。

### 场景1：函数直接调用时

```
function myfunc() {
    console.log(this) // this是window
}
var a = 1;
myfunc();
```

### 场景2：函数被别人调用时

```
function myfunc() {
    console.log(this) // this是对象a
}
var a = {
    myfunc: myfunc
};
a.myfunc();
```

### 场景3：new一个实例时

```
function Person(name) {
    this.name = name;
    console.log(this); // this是指实例p
}
var p = new Person('zhaowa');
```

### 场景4：apply、call、bind时

```
function getColor(color) {
    this.color = color;
    console.log(this);
}

function Car(name, color){
    this.name = name;    // this指的是实例car
    getColor.call(this, color); // 这里的this从原本的getColor, 变成了car
}

var car = new Car('卡车', '绿色');
```

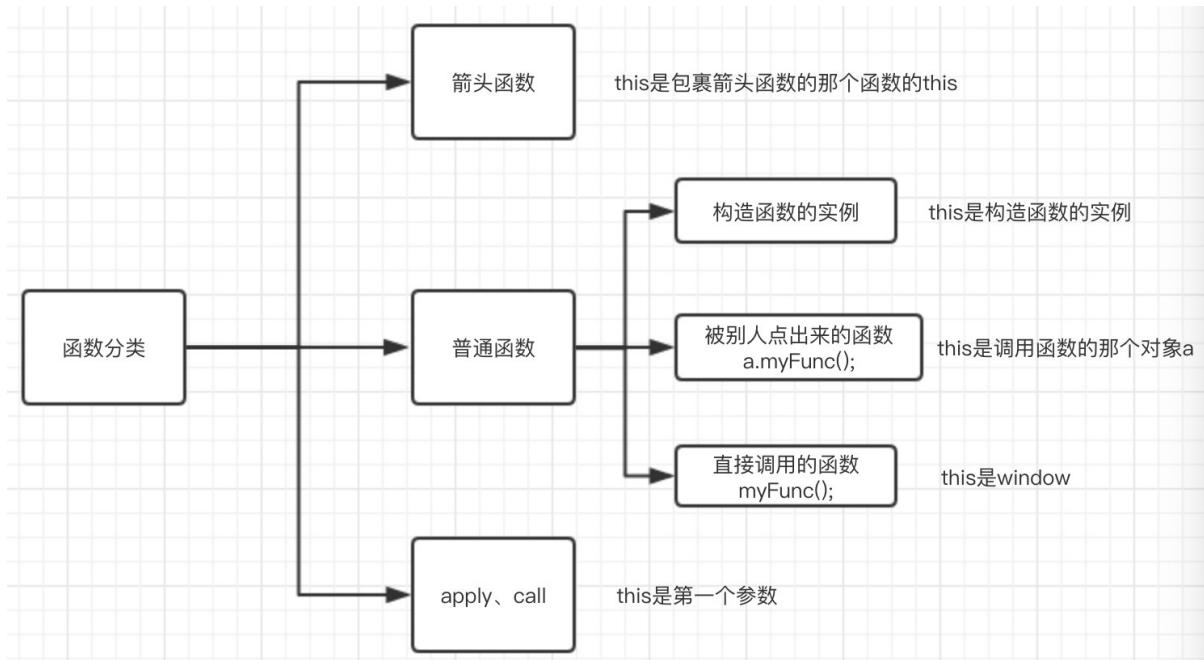
## 场景5: 箭头函数时

```
// 复习一下场景1
var a = {
    myfunc: function() {
        setTimeout(function(){
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();

// 稍微改变一下
var a = {
    myfunc: function() {
        var that = this;
        setTimeout(function(){
            console.log(that); // this是a
        }, 0)
    }
};
a.myfunc();

// 箭头函数
var a = {
    myfunc: function() {
        setTimeout(() => {
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();
```

总结一下



- 对于直接调用的函数来说，不管函数被放在了什么地方，this都是window
- 对于被别人调用的函数来说，被谁点出来的，this就是谁
- 在构造函数中，类中(函数体中)出现的this.xxx=xxx中的this是当前类的一个实例
- call、apply时，this是第一个参数。bind要优与call/apply哦，call参数多，apply参数少
- 箭头函数没有自己的this，需要看其外层的是否有函数，如果有，外层函数的this就是内部箭头函数的this，如果没有，则this是window

## 相关面试题

### 1. 考察this三板斧

#### 1.1

```

function show () {
  console.log('this:', this);
}
var obj = {
  show: show
};
obj.show();
  
```

```

function show () {
  console.log('this:', this);
}
  
```

```

var obj = {
  
```

```
show: function () {
    show();
}
};

obj.show();
```

## 1.2

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

(0, obj.show)();
```

## 1.3

```
var obj = {
    sub: {
        show: function () {
            console.log('this:', this);
        }
    }
};
```

## 1.4

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

var newobj = new obj.show();
```

## 1.5

```
var obj = {
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

var newobj = new (obj.show.bind(obj))();
```

## 1.6

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var newobj = new (obj.show.bind(obj))();
```

## 1.7

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var elem = document.getElementById('book-search-results');
elem.addEventListener('click', obj.show);
elem.addEventListener('click', obj.show.bind(obj));
elem.addEventListener('click', function () {
    obj.show();
});
```

## 2. 作用域

### 2.1

```
var person = 1;
function showPerson() {
    var person = 2;
    console.log(person);
}
showPerson();
```

### 2.2

```
var person = 1;
function showPerson() {
    console.log(person);
    var person = 2;
}
showPerson();
```

### 2.3

```
var person = 1;
function showPerson() {
    console.log(person);
```

```
var person = 2;
function person() {}
}
showPerson();
```

## 2.4

```
var person = 1;
function showPerson() {
    console.log(person);

    function person() {}
    var person = 2;
}
showPerson();
```

## 2.5

```
for(var i = 0; i < 10; i++) {
    console.log(i);
}

for(var i = 0; i < 10; i++) {
    setTimeout(function(){
        console.log(i);
    }, 0);
}

for(var i = 0; i < 10; i++) {
    (function(i){
        setTimeout(function(){
            console.log(i);
        }, 0)
    })(i);
}

for(let i = 0; i < 10; i++) {
    console.log(i);
}
```

## 3. 面向对象

### 3.1

```
function Person() {
    this.name = 1;
```

```
    return {};
}
var person = new Person();
console.log('name:', person.name);
```

### 3.2

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();
person.show();
```

### 3.3

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();

Person.prototype.show = function () {
    console.log('new show');
};

person.show();
```

### 3.4

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
}
```

```
};

var person = new Person();
var person2 = new Person();

person.show = function () {
    console.log('new show');
};

person2.show();
person.show();
```

#### 4 综合题

```
function Person() {
    this.name = 1;
}

Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};

Person.prototype.show();

(new Person()).show();
```