作用域 & 上下文

作用域链

- 面试题：

```
let a = 'global'

function course() {
    let b = 'zhaowa'

    session()
    function session() {
        let c = 'session'

        teacher()
        function teacher() {
            // let d = 'yy'
            console.log('ds', d)
            var d = 'yy'  // 变量提升 & 变量提升范围：当前作用域

            console.log('d', d); // 作用域生效
            console.log('b', b); // 作用域向上查找
        }
    }
}

course()

// ************************************
// 提升优先级的问题 => 函数会需要变量
// 变量 函数同时提升时
//  提升维度：变量优先
//  执行维度：函数先打印出来

function test() {}

// ************************************
// 结论：函数天然的隔离方案 => 模块化基础 => 模块化课程
if (true) {
    let e = 111
    var f = 222
}
console.log(e, f)
```

- i. 对于作用域链 我们可以直接通过创建态区定位链条中的某一环 => 后半节课
- ii. 手动取消链条环甚至全局作用域的时候，可以利用块级作用域区做性能优化 => 性能优化

this 上下文 context

- 我家门前有条河，门前河上有座桥，河里有群鸭
- 我家门前有条河，this有座桥，this有群鸭
- this是在执行时动态读取上下文所决定的

考察重点 - 各使用态中的指针指向

函数直接调用 - this指向是window

```
function foo() {
    console.log('函数内部this', this)
}

foo()
```

隐式绑定 - this的指向指向的是调用堆栈的上一级

```
function fn() {
    console.log('隐式绑定', this.a)
}

const obj = {
    a: 1,
    fn
}

obj.fn = fn
obj.fn()
```

面试题：

```
const foo = {
    bar: 10,
    fn: function() {
        console.log(this.bar)
        console.log(this)
    }
}

// 取出
let fn1 = foo.fn
fn1()

// 追问：
const o1 = {
    text: 'o1',
    fn: function() {
        // 直接使用上下文 => 传统分活
        console.log('o1fn_this', this)
        return this.text
    }
}

const o2 = {
    text: 'o2',
    fn: function() {
        // 求助领导 - 部门协作
        console.log('o2fn_this', this)
        return o1.fn()
    }
}

const o3 = {
    text: 'o3',
    fn: function() {
        // 直接借人
        console.log('o3fn_this', this)
        let fn = o1.fn
        return fn()
    }
}

console.log('o1', o1.fn())
```

```
console.log('o2', o2.fn())
console.log('o3', o3.fn())
```

- i. 在执行函数的时候，函数执行时调用方上一级 => 上下文
- ii. 公共函数 | 全局调用指向window

显式绑定（bind | apply | call）

- i. call < = > apply 传参不同 依次传入 / 数组传入
- ii. bind 返回值不同
- 面试：手写apply & bind

```
// 1. 需求：手写bind => bind位置 => Function.prototype => 原型
Function.prototype.newBind = function() {
    // 1.1 bind原理
    const _this = this
    const args = Array.prototype.slice.call(arguments) // 类数组
    const newThis = args.shift()

    // 1.2 返回值不执行 => 返回函数
    return function() {
        // 执行核心
        return _this.newApply(newThis, args)
    }
}

// 2. 内层实现
Function.prototype.newApply = function(context) {
    // 参数兜底
    context = context || window

    // 临时挂载执行函数
    context.fn = this

    let result = arguments[1]
        ? context.fn(...arguments[1])
        : context.fn()

    delete context.fn
    return result;
}
```

追问：如何突破作用域?

闭包

```
function mail() {
    let content = '信'

    return function() {
        return content
    }
}

const envelop = mail()
envelop()
```

=> 异步 + 闭包使用