那些高级/资深的前端是如何回答JavaScript面试题的 （一）

Question 1: JS闭包，你了解多少？`

应该有面试官问过你：

1. 什么是闭包？
2. 闭包有哪些实际运用场景？
3. 闭包是如何产生的？
4. 闭包产生的变量如何被回收？

这些问题其实都可以被看作是同一个问题，那就是面试官在问你： 你对JS闭包了解多少？

来总结一下我听到过的答案，尽量完全复原候选人面试的时候说的原话。

答案1: 就是一个 function 里面 return 了一个子函数，子函数访问了外面那个函数的变量。

答案2: for循环里面可以用闭包来解决问题。

js 复制代码

```
for(var i = 0; i < 10; i++){
    setTimeout(()=>console.log(i),0)
}
// 控制台输出10遍10.
for(var i = 0; i < 10; i++){
    (function(a){
    setTimeout(()=>console.log(a),0)
    })(i)
}
 // 控制台输出0-9
```

答案3： 当前作用域产产生了对父作用域的引用。

答案4： 不知道。是跟浏览器的垃圾回收机制有关吗？

开杠了。请问，小伙伴的答案和以上的内容有多少相似程度？

其实，拿着这些问题好好想想，你就会发现这些问题都只是为了最终那一个问题。

闭包的底层实现原理

1. JS执行上下文

我们都知道，我们手写的js代码是要经过浏览器V8进行预编译后才能真正的被执行。例如变量提升、函数提升。举个栗子。

```
// 栗子：
var d = 'abc';
function a(){
    console.log("函数a");
};
console.log(a);   // ƒ a(){ console.log("函数a"); }
a();                  // '函数a'
var a = "变量a";
console.log(a);   // '变量a'
a();                      // a is not a function
var c = 123;

// 输出结果及顺序：
// ƒ a(){ console.log("函数a"); }
// '函数a'
// '变量a'
// a is not a function

// 栗子预编后相当于：
function a(){
    console.log("函数a");
};
var d;
console.log(a);  // ƒ a(){ console.log("函数a"); }
a();                     // '函数a'

a = "变量a";     // 此时变量a赋值，函数声明被覆盖

console.log(a); // "变量a"
a();            // a is not a function
```

那么问题来了。 **请问是谁来执行预编译操作的？那这个谁又是在哪里进行预编译的？**

是的，你的疑惑没有错。js代码运行需要一个运行环境，那这个环境就是**执行上下文**。 是的，js运行前的预编译也是在这个环境中进行。

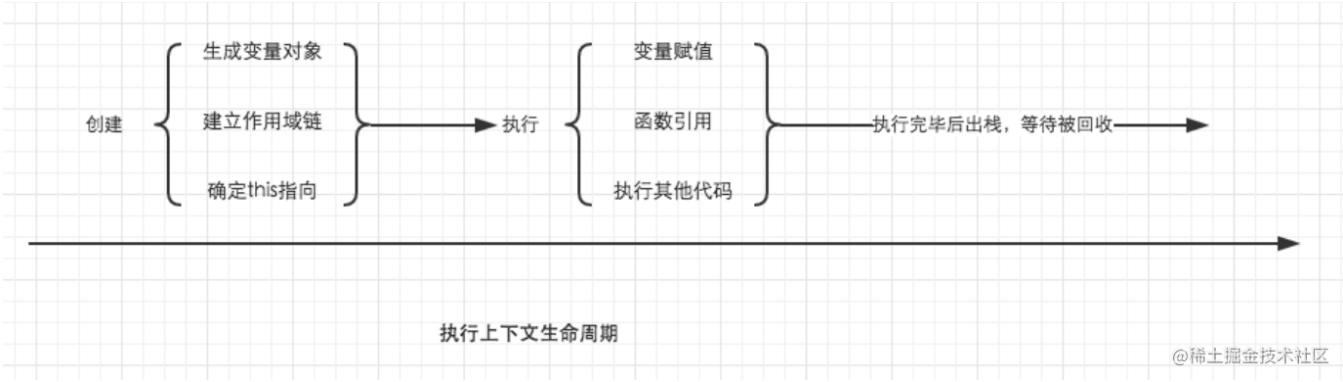
js执行上下文分三种：

- 全局执行上下文： 代码开始执行时首先进入的环境。
- 函数执行上下文： 函数调用时，会开始执行函数中的代码。
- eval执行上下文： 不建议使用，可忽略。

那么，执行上下文的周期，分为两个阶段：

- 创建阶段
  - 创建词法环境
  - 生成变量对象(VO)，建立**作用域链**、**作用域链**、**作用域链**（重要的事说三遍）
  - 确认 this 指向，并绑定 this
- 执行阶段 。这个阶段进行变量赋值，函数引用及执行代码。

看图理解一下。



你现在猜猜看，预编译是发生在什么时候?

噢，我忘记说了，其实与编译还有另一个称呼： 执行期上下文 。

预编译发生在函数执行之前。预编译四部曲为：

1. 创建 AO 对象
2. 找形参和变量声明，将变量和形参作为AO属性名，值为 undefined
3. 将实参和形参相统一
4. 在函数体里找到函数声明，值赋予函数体。最后程序输出变量值的时候，就是从 AO 对象中拿。

所以，预编译真正的结果是：

js 复制代码

```
var AO = {
    a = function a(){console.log("函数a");};
```

```
    d = 'abc'
}
```

我们重新来。

1. 什么叫变量对象?

变量对象是 `js` 代码在进入执行上下文时， `js` 引擎在内存中建立的一个对象，用来存放当前执行环境中的变量。

2. 变量对象(VO)的创建过程

变量对象的创建，是在执行上下文创建阶段，依次经过以下三个过程：

- 创建 `arguments` 对象。

  对于函数执行环境，首先查询是否有传入的实参，如果有，则会将参数名是实参值组成的键值对放入 `arguments` 对象中。否则，将参数名和 `undefined` 组成的键值对放入 `arguments` 对象中。

```
//举个栗子
function bar(a, b, c) {
    console.log(arguments);  // [1, 2]
    console.log(arguments[2]); // undefined
}
bar(1,2)
```

- 当遇到同名的函数时，后面的会覆盖前面的。

```
console.log(a); // function a() {console.log('Is a ?') }
function a() {
    console.log('Is a');
}
function a() {
  console.log('Is a ?')
}

/**
ps: 在执行第一行代码之前，函数声明已经创建完成.
后面的对之前的声明进行了覆盖。
**/
```

- 检查当前环境中的变量声明并赋值为 `undefined` 。当遇到同名的函数声明，**为了避免函数被赋值为 `undefined` ,会忽略此声明**

```js 复制代码
console.log(a); // function a() {console.log('Is a ?') }
console.log(b); // undefined
function a() {
  console.log('Is a ');
}
function a() {
console.log('Is a ?');
}
var b = 'Is b';
var a = 10086;

/**
这段代码执行一下，你会发现 a 打印结果仍旧是一个函数，而 b 则是 undefined。
**/
```

根据以上三个步骤，对于变量提升也就知道是怎么回事了。

3. 变量对象变为活动对象

执行上下文的第二个阶段，称为执行阶段，在此时，会进行变量赋值，函数引用并执行其他代码，此时，变量对象变为活动对象。

我们还是举上面的例子：

```js 复制代码
console.log(a); // function a() {console.log('fjdsfs') }
console.log(b); // undefined
function a() {
   console.log('Is a');
}
function a() {
 console.log('Is a?');
}
var b = 'Is b';
console.log(b); // 'Is b'
var a = 10086;
console.log(a);  // 10086
var b = 'Is b?';
console.log(b); // 'Is b?'
```

在上面的代码中，代码真正开始执行是从第一行 console.log() 开始的，自这之前，执行上下文是这样的：

```
// 创建过程
EC= {
  VO:  {}; // 创建变量对象
  scopeChain: {}; // 作用域链
}
VO = {
  argument: {...}; // 当前为全局上下文，所以这个属性值是空的
  a: <a reference> // 函数 a  的引用地址
  b: undefiend  // 见上文创建变量对象的第三步
}
```

词法作用域（Lexical scope）

这里想说明，我们在函数执行上下文中有变量，在全局执行上下文中有变量。 JavaScript 的一个复杂之处在于它如何查找变量，如果在函数执行上下文中找不到变量，它将在调用上下文中寻找它，如果在它的调用上下文中没有找到，就一直往上一级，直到它在全局执行上下文中查找为止。(如果最后找不到，它就是 undefined)。

再来举个栗子：

```
1: let top = 0; //
2: function createWarp() {
3:   function add(a, b) {
4:     let ret = a + b
5:     return ret
6:   }
7:   return add
8: }
9: let sum = createWarp()
10: let result = sum(top, 8)
11: console.log('result:',result)
```

分析过程如下：

- 在全局上下文中声明变量 top 并赋值为0.

- 2 - 8行。在全局执行上下文中声明了一个名为 `createWarp` 的变量，并为其分配了一个函数定义。其中第3-7行描述了其函数定义，并将函数定义存储到那个变量(`createWarp`)中。
- 第9行。我们在全局执行上下文中声明了一个名为 `sum` 的新变量，暂时，值为 `undefined` 。
- 第9行。遇到 `()` ，表明需要执行或调用一个函数。**那么查找全局执行上下文的内存并查找名为 `createWarp` 的变量。** 明显，已经在步骤2中创建完毕。接着，调用它。
- 调用函数时，回到第2行。创建一个新的 `createWarp` 执行上下文。我们可以在 `createWarp` 的执行上下文中创建自有变量。 `js` 引擎 `createWarp` 的上下文添加到调用堆栈（ `call stack` ）。因为这个函数没有参数，直接跳到它的主体部分.
- 3 - 6 行。我们有一个新的函数声明，**在 `createWarp` 执行上下文中创建一个变量 `add` 。 `add` 只存在于 `createWarp` 执行上下文中, 其函数定义存储在名为 `add` 的自有变量中。**
- 第7行，我们返回变量 `add` 的内容。js引擎查找一个名为 `add` 的变量并找到它. 第4行和第5行括号之间的内容构成该函数定义。
- **`createWarp` 调用完毕， `createWarp` 执行上下文将被销毁。add 变量也跟着被销毁。** 但 `add` 函数定义仍然存在，因为它返回并赋值给了 `sum` 变量。 *(ps: 这才是闭包产生的变量存于内存当中的真相)*
- 接下来就是简单的执行过程，不再赘述。。
- ……
- 代码执行完毕，全局执行上下文被销毁。sum 和 result 也跟着被销毁。

小结一下

现在，如果再让你回答什么是闭包，你能答出多少?

其实，大家说的都对。不管是函数返回一个函数，还是产生了外部作用域的引用，都是有道理的。

所以，什么是闭包?

- 解释一下作用域链是如何产生的。
- 解释一下js执行上下文的创建、执行过程。
- 解释一下闭包所产生的变量放在哪了。
- 最后请把以上3点结合起来说给面试官听。

> 另外，假如被问到 `Event loop、执行栈（EC Stack）、调用栈（Call Stack）` ，**请你一定要明确一件事情，执行栈和调用栈它们不是一个东西。** 很多文章都不写清楚两者的区别，或者干脆就说他们就是一个东西。后面有时间，为会专门文它们写一篇文章为大家解惑

祝，君无往不利。