闭包

JavaScript 是一种非常面向函数的语言。它给我们很大的发挥空间。函数创建后，可以赋值给其他变量或作为参数传递给另一个函数，并在完全不同的位置进行调用。

我们知道函数可以访问其外部变量；这是一个常用的特性。

但是当外部变量变化时会发生什么呢？函数获得的是最新的值还是创建时的值呢？

另外，函数移动到其他位置调用会如何呢——它是否可以访问新位置处的外部变量呢？

不同的语言在这里行为不同，在本章中，我们将介绍 JavaScript 的行为。

几个问题

一开始让我们考虑两种情况，然后逐步学习其内部机制，这样你就可以回答下列问题和未来更难的问题。

1. 函数 `sayHi` 用到 `name` 这个外部变量 。当函数执行时，它会使用哪一个值呢？

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

*!*
sayHi(); // 它会显示 "John" 还是 "Pete" 呢？
*/!*
```

   这种情况在浏览器端和服务端的开发中都很常见。函数可能会在创建后一段时间才调度执行，比如在用户操作或网络请求之后。

   所以，问题是：它是否接收到的是最新的值呢？

1. 函数 `makeWorker` 生成并返回另一个函数。这个新的函数可以在其他位置进行调用。它访问的是创建位置还是调用位置的外部变量呢，还是都可以？

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
```

```
}

let name = "John";

// 创建一个函数
let work = makeWorker();

// call it
*!*
work(); // 它显示的是什么呢？"Pete"（创建位置的 name）还是"John"（调用位置的 name）呢？
*/!*
```

词法环境

要了解究竟发生了什么，首先我们要来讨论一下『变量』究竟是什么?

在 JavaScript 中，每个运行的函数、代码块或整个程序，都有一个称为**词法环境（Lexical Environment）**的关联对象。

词法环境对象由两部分组成：

1. **环境记录（Environment Record）**—— 一个把所有局部变量作为其属性（包括一些额外信息，比如 `this` 值）的对象。
2. **外部词法环境（outer lexical environment）**的引用 —— 通常是嵌套当前代码（当前花括号之外）之外代码的词法环境。

所以，『变量』只是环境记录这个特殊内部对象的属性。『访问或修改变量』意味着『访问或改变词法环境的一个属性』。

举个例子，这段简单的代码中只有一个词法环境：

这是一个所谓的与整个程序关联的全局词法环境。在浏览器中，所有的 `<script>` 标签都同享一个全局（词法）环境。

在上图中，矩形表示环境记录（存放变量），箭头表示外部（词法环境）引用。全局词法环境没有外部（词法环境）引用，所以它指向了 `null`。

这是关于 `let` 变量（如何变化）的全程展示：

右侧的矩形演示在执行期间全局词法环境是如何变化的：

1. 执行开始时，词法环境为空。

2. `let phrase` 定义出现。它没有被赋值，所以存为 `undefined`。
3. `phrase` 被赋予了一个值。
4. `phrase` 引用了一个新值。

现在一切看起来都相当简单易懂，是吧？

总结一下：

- 变量是特定内部对象的属性，与当前执行的（代码）块/函数/脚本有关。
- 操作变量实际上操作的是该对象的属性。

函数声明

函数声明不像 `let` （声明的）变量，代码执行到它们时，它们并不会执行，而是在词法环境创建完成后才会执行。对于全局词法环境来说，它意味着脚本启动的那一刻。

这就是为什么可以在（函数声明）的定义之前调用函数声明。

下面的代码的词法环境一开始并不为空。因为 `say` 是一个函数声明，所以它里面有 `say`。后面它获得了用 `let` 定义的 `phrase` （属性）。

内部和外部的词法环境

在调用中，`say()` 用到了一个外部变量，那么让我们了解一下其中的细节。

首先，当函数运行时，会自动创建一个新的函数词法环境。这是一条对于所有函数通用的规则。这个词法环境用于存储调用的局部变量和参数。

下面是 `say("John")` 内执行时词汇环境的图示，线上标有一个箭头：

在这个函数的执行中，有两个词法环境：内部一个（用于函数调用）和外部一个（全局）：

- 内部词法环境对应于 `say` 的当前执行。它有一个单独的变量：`name`，它是一个函数参数。我们执行 `say("John")`，那么 `name` 的值为 `"John"`。
- 它的外部词法环境就是全局词法环境。

它的内部词法环境有**外部**引用（指向）外部的那个。

当代码试图访问一个变量时 —— 它首先会在内部词法环境中进行搜索，然后是外部环境，然后是更外部的环境，直到（词法环境）链的末尾。

在严格模式下，变量未定义会导致错误。在非严格模式下，为了向后兼容，给未定义的变量赋值会创建一个全局变量。

让我们看看例子中的查找是如何进行的：

- 当 `say` 中的 `alert` 试图访问 `name` 时，它立即在函数词法环境中被找到。
- 当它试图访问 `phrase` 时，然而内部没有 `phrase` ，所以它追踪**外部**引用并在全局中找到它。

现在我们可以回答本章开头第一个问题了。

函数访问外部变量；它使用的是最新的值。

这是因为上述的机制。旧的变量不会被存储。当函数需要它们时，它会从外部词法环境中或自身（内部词法环境）中获得当前值。

所以第一个问题的答案是 `Pete`：

```js run let name = "John";

function sayHi() { alert("Hi, " + name); }

name = "Pete"; // (*)

*!* sayHi(); // Pete *!*
```

上述代码的执行流程：

1. 全局词法环境中有 `name: "John"`。
2. 在 `(*)` 那一行，全局变量已经变化，现在它为 `name: "Pete"`。
3. 当函数 `say()` 执行时，它从外部获得 `name`。它取自全局词法环境，它已经变为 `"Pete"` 了。

```smart header="一次调用 —— 一个词法环境"
请记住，每次函数运行会都会创建一个新的函数词法环境。

如果一个函数被调用多次，那么每次调用也都会此创建一个拥有指定局部变量和参数的词法环境。
```

```smart header="词法环境是一个规范对象" 『词法环境』是一个规范对象。我们不能在代码中获取或直接操作该对象。但 JavaScript 引擎同样可以优化它，比如清除未被使用的变量以节省内存和执行其他内部技巧等，但显性行为应该是和上述的无差。
```

```
## 嵌套函数

当在函数中创建函数时，这就是所谓的『嵌套』。

在 JavaScript 中是很容易实现的。

我们可以使用嵌套来组织代码，比如这样：

```js
function sayHiBye(firstName, lastName) {

  // 辅助嵌套函数如下
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

这里创建的**嵌套**函数 `getFullName()` 是为了方便说明。它可以访问外部变量，因此可以返回全名。

更有意思的是，可以返回一个嵌套函数：把它作为一个新对象的属性（如果外部函数创建一个有方法的对象）或是将其直接作为结果返回。其后可以在别处调用它。不论在哪里进行调用到它，它仍可以访问同样的外部变量。

一个构造函数的例子（请参考 ）：

```js run // 构造函数返回一个新对象 function User(name) {

// 这个对象方法为一个嵌套函数 this.sayHi = function() { alert(name); }; }

let user = new User("John"); user.sayHi(); // 该方法访问外部变量 "name"
```

一个返回函数的例子：

```js run
function makeCounter() {
  let count = 0;

  return function() {
    return count++; // has access to the outer counter
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```
```

让我们继续来看 `makeCounter` 这个例子。它返回一个函数，（返回的）该函数每次调用都会返回下一个数字。尽管它的代码很简单，但稍加变型就会有实际的用途，比如，作一个 伪随机数生成器 等等。所以这个例子并不像看起来那么造作。

计数器内部的工作是怎样的呢?

当内部函数运行时， `count++` 会由内到外搜索该变量。在上面的例子中，步骤应该是：

1. 内部函数的本地。
2. 外部函数的变量。
3. 以此类推直到到达全局变量。

在这个例子中， `count` 在第二步中被找到。当外部变量被修改时，在找到它的地方被修改。因此 `count++` 找到该外部变量并在它从属的词法环境中进行修改。好像我们有 `let count = 1` 一样。

这里有两个要考虑的问题：

1. 我们可以用某种方式在 `makeCounter` 以外的代码中改写 `counter` 吗? 比如，在上例中的 `alert` 调用后。
2. 如果我们多次调用 `makeCounter()` —— 它会返回多个 `counter` 函数。它们的 `count` 是独立的还是共享的同一个呢?

在你继续读下去之前，请先回答这些问题。

...

想清楚了吗?

好吧，我们来重复一下答案。

1. 这是不可能做到的。 `counter` 是一个局部函数的变量，我们不能从外部访问它。
2. `makeCounter()` 的每次调用都会创建一个拥有独立 `counter` 的新词法环境。因此得到的 `counter` 是独立的。

下面是一个例子：

```js run function makeCounter() { let count = 0; return function() { return count++; }; }

let counter1 = makeCounter(); let counter2 = makeCounter();

alert( counter1() ); // 0 alert( counter1() ); // 1

alert( counter2() ); // 0 （独立的)
```

希望现在你对外部变量的情况相当清楚了。但对于更复杂的情况，可能需要更深入的理解。所以让我们更加深入吧。

环境详情

现在你大概已经了解闭包的工作原理了，我们可以探讨一下实际问题了。

以下是 `makeCounter` 的执行过程，遵循它确保你理解所有的内容。请注意我们还没有介绍另外的 `[[Environment]]` 属性。

1. 在脚本开始时，只存在全局词法环境：

   在开始时里面只有一个 `makeCounter` 函数，因为它是一个函数声明。它还没有执行。

   所有的函数在『诞生』时都会根据创建它的词法环境获得隐藏的 `[[Environment]]` 属性。我们还没有讨论到它，但这是函

   在这里，`makeCounter` 创建于全局词法环境，那么 `[[Environment]]` 中保留了它的一个引用。

   换句话说，函数会被创建处的词法环境引用『盖章』。隐藏函数属性 `[[Environment]]` 中有这个引用。

2. 代码执行，`makeCounter()` 被执行。下图是当 `makeCounter()` 内执行第一行瞬间的快照：

   在 `makeCounter()` 执行时，包含其变量和参数的词法环境被创建。

   词法环境中存储着两个东西：
   1. 一个是环境记录，它保存着局部变量。在我们的例子中 `count` 是唯一的局部变量（当执行到 `let count` 这一行时出
   2. 另外一个是外部词法环境的引用，它被设置为函数的 `[[Environment]]` 属性。这里 `makeCounter` 的 `[[Environ

   所以，现在我们有了两个词法环境：第一个是全局环境，第二个是 `makeCounter` 的词法环境，它拥有指向全局环境的引用

3. 在 `makeCounter()` 的执行中，创建了一个小的嵌套函数。

   不管是使用函数声明或是函数表达式创建的函数都没关系，所有的函数都有 `[[Environment]]` 属性，该属性引用着所创建

   我们新的嵌套函数的 `[[Environment]]` 的值就是 `makeCounter()` 的当前词法环境（创建的位置）。

   请注意在这一步中，内部函数并没有被立即调用。 `function() { return count++; }` 内的代码还没有执行，我们要返回

4. 随着执行的进行，`makeCounter()` 调用完成，并且将结果（该嵌套函数）赋值给全局变量 `counter`。

   这个函数中只有 `return count++` 这一行代码，当我们运行它时它会被执行。

5. 当 `counter()` 执行时，它会创建一个『空』的词法环境。它本身没有局部变量，但是 `counter` 有 `[[Environment]]`

   如果它要访问一个变量，它首先会搜索它自身的词法环境（空），然后是前面创建的 `makeCounter()` 函数的词法环境，然

   当它搜索 `count` ，它会在最近的外部词法环境 `makeCounter` 的变量中找到它。

   请注意这里的内存管理工作机制。虽然 `makeCounter()` 执行已经结束，但它的词法环境仍保存在内存中，因为这里仍然有

```
    通常，只要有一个函数会用到该词法环境对象，它就不会被清理。并且只有没有（函数）会用到时，才会被清除。

6. `counter()` 函数不仅会返回 `count` 的值，也会增加它。注意修改是『就地』完成的。准确地说是在找到 `count` 值的地

    ![](lexenv-nested-makecounter-6.png)

    因此，上一步只有一处不同 — `count` 的新值。下面的调用也是同理。

7. 下面 `counter()` 的调用也是同理。

本章开头问题的答案应该已经是显而易见了。

下面代码中的 `work()` 函数通过外部词法环境引用使用到来自原来位置的 `name`。

![](lexenv-nested-work.png)

所以，这里的结果是 `"Pete"`。

但如果 `makeWorker()` 中没有 `let name` 的话，那么搜索会进行到外部，直到到达链未的全局环境。在这个例子，它应该会

```smart header="闭包"
开发者应该有听过『闭包』这个编程术语。

函数保存其外部的变量并且能够访问它们称之为[闭包](https://en.wikipedia.org/wiki/Closure_(computer_programming)

也就是说，他们会通过隐藏的 `[[Environment]]` 属性记住创建它们的位置，所以它们都可以访问外部变量。

在面试时，前端通常都会被问到『什么是闭包』，正确的答案应该是闭包的定义并且解释 JavaScript 中所有函数都是闭包，以及
```

代码块和循环、IIFE

上面的例子都集中于函数。但对于 `{...}` 代码块，词法环境同样也是存在的

当代码块中包含块级局部变量并运行时，会创建词法环境。这里有几个例子。

If

在上面的例子中，当代码执行入 `if` 块，新的 "if-only" 词法环境就会为此而创建：

新的词法环境是封闭的作为其外部引用，所以找不到 `phrase`。但在 `if` 内声明的变量和函数表达式都保留在该词法环境中，从外部是无法被访问到的。

例如，在 `if` 结束后，下面的 `alert` 是访问不到 `user` 的，因为会发生错误。

For, while

对于循环而言，每次迭代都有独立的词法环境。如果在 `for` 循环中声明变量，那么它在词法环境中是局部的：
```

```js run for (let i = 0; i < 10; i++) { // 每次循环都有其自身的词法环境 // {i: value} }

alert(i); // 错误，没有该变量

这实际上是个意外，因为 `let i` 看起来好像是在 `{...}` 外。但事实上，每一次循环执行都有自己的词法环境，其中包含着

在循环结束后，`i` 是访问不到的。

### 代码块

我们也可以使用『空』的代码块将变量隔离到『局部作用域』中。

比如，在 Web 浏览器中，所有脚本都共享同一个全局环境。如果我们在一个脚本中创建一个全局变量，对于其他脚本来说它也是可

如果变量名是一个被广泛使用的词，并且脚本作者可能彼此也不知道。

如果我们要避免这个，我们可以使用代码块来隔离整个脚本或其中一部分：

```js run
{
  // 用局部变量完成一些不应该被外面访问的工作

  let message = "Hello";

  alert(message); // Hello
}

alert(message); // 错误：message 未定义
```

这是因为代码块有其自身的词法环境，块之外（或另一个脚本内）的代码访问不到代码块内的变量。

IIFE

在旧的脚本中，我们可以找到一个所谓的『立即调用函数表达式』（简称为 IIFE）用于此目的。

它们看起来像这样：

```js run (function() {

let message = "Hello";

alert(message); // Hello

})();

这里创建了一个函数表达式并立即调用。因此代码拥有自己的私有变量并立即执行。

函数表达式被括号 `(function {...})` 包裹起来，因为在 JavaScript 中，当代码流碰到 `"function"` 时，它会把它当成

```js run
// Error: Unexpected token (
function() { // <-- JavaScript 找不到函数名，遇到 ( 导致错误

  let message = "Hello";

  alert(message); // Hello

}();
```

我们可以说『好吧，让其变成函数声明，让我们增加一个名称』，但它是没有效果的。JavaScript 不允许立即调用函数声明。

```js run // syntax error because of brackets below function go() {

}(); // <-- can't call Function Declaration immediately

因此，需要使用圆括号告诉给 JavaScript，这个函数是在另一个表达式的上下文中创建的，因此它是一个表达式。它不需要函数名

在 JavaScript 中还有其他方式来定义函数表达式：

```js run
// 创建 IIFE 的方法

(function() {
  alert("Brackets around the function");
}*!*)*/!*();

(function() {
  alert("Brackets around the whole thing");
}()*!*)*/!*;

*!*!*/!**function() {
  alert("Bitwise NOT operator starts the expression");
}();

*!*+*/!**function() {
  alert("Unary plus starts the expression");
}();
```

在上面的例子中，我们声明一个函数表达式并立即调用：

垃圾收集

我们所讨论的词法环境和常规值都遵循同样的内存管理规则。

- 通常，在函数运行后词法环境会被清理。举个例子：
```

```
function f() {
  let value1 = 123;
  let value2 = 456;
}

f();
```

这里的两个值都是词法环境的属性。但是在 `f()` 执行完后，该词法环境变成不可达，因此它在内存中已被清理。

- ...但是如果有一个嵌套函数在 `f` 结束后仍可达，那么它的 `[[Environment]]` 引用会继续保持着外部词法环境存在：

```
function f() {
  let value = 123;

  function g() { alert(value); }

*!*
  return g;
*/!*
}

let g = f(); // g 是可达的，并且将其外部词法环境保持在内存中
```

- 请注意如果多次调用 `f()`，返回的函数被保存，那么其对应的词法对象同样也会保留在内存中。下面代码中有三个这样的函数：

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 数组中的三个函数，每个都有词法环境相关联。
// 来自对应的 f()
//         LE   LE   LE
let arr = [f(), f(), f()];
```

- 词法环境对象在变成不可达时会被清理：当没有嵌套函数引用（它）时。在下面的代码中，在 `g` 变得不可达后， `value` 同样会被从内存中清除；

```
function f() {
  let value = 123;

  function g() { alert(value); }

  return g;
}
```

```
let g = f(); // 当 g 存在时
// 对应的词法环境可达

g = null; // ...在内存中被清理
```

实际的优化

正如我们所了解的，理论上当函数可达时，它外部的所有变量都将存在。

但实际上，JavaScript 引擎会试图优化它。它们会分析变量的使用情况，如果有变量没被使用的话它也会被清除。

V8（Chrome、Opera）的一个重要副作用是这样的变量在调试是无法访问的。

打开 Chrome 浏览器的开发者工具运行下面的代码。

当它暂停时，在控制台输入 `alert(value)`。

```js run function f() { let value = Math.random();

function g() { debugger; // 在控制台中输入 alert( value );没有该值！ }

return g; }

let g = f(); g();
```

正如你所见的 —— 没有该值！理论上，它应该是可以访问的，但引擎对此进行了优化。

这可能会导致有趣的调试问题。其中之一 — 我们可以看到的是一个同名的外部变量，而不是预期的变量：

```js run global
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // 在控制台中：输入 alert( value )；Surprise!
  }

  return g;
}

let g = f();
g();
```

```warn header="再见！" V8 的这个特性了解一下也不错。如果用 Chrome/Opera 调试的话，迟早你会遇到。

这并不是调试器的 bug ， 而是 V8 的一个特别的特性。或许以后会进行修改。 你可以经常运行本页的代码来进行检查（这个特性）。 ```
```