闭包

闭包是由捆绑起来（封闭的）的函数和函数周围状态（**词法环境**）的引用组合而成。换言之，闭包让函数能访问它的外部作用域。在 JavaScript 中，闭包会随着函数的创建而同时创建。

词法作用域

注意下面的示例代码：

JS

```
function init() {
  var name = "Mozilla"; // name 是 init 创建的局部变量
  function displayName() {
    // displayName() 是内部函数，它创建了一个闭包
    console.log(name); // 使用在父函数中声明的变量
  }
  displayName();
}
init();
```

`init()` 创建了一个名为 `name` 的局部变量和一个名为 `displayName()` 的函数。`displayName()` 是在 `init()` 内定义的内部函数，并且仅在 `init()` 函数的函数体内可用。请注意，`displayName()` 没有自己的局部变量。然而，因为内部函数能访问外部作用域的变量，所以 `displayName()` 能访问在 `init()` 父函数中声明的 `name` 变量。

使用这个 JSFiddle 链接运行该代码后发现，`displayName()` 函数内的 `console.log()` 成功显示了在其父函数中声明的 `name` 变量的值。这是一个*词法作用域*的示例，它描述了解析器在函数嵌套时如何解析变量名。*词法*一词是指词法作用域使用源代码内变量声明的位置决定变量可用的位置。嵌套函数能访问在其外部作用域中声明的变量。

let 和 const 的作用域

一直以来（ES 6 之前），JavaScript 变量仅有两种类型的作用域：*函数作用域*和*全局作用域*。用 `var` 声明的变量要么属于函数作用域要么属于全局作用域，这取决于变量是在函数内声明的还是在函数外声明的。花括号块不为 `var` 创建作用域让人有点难以捉摸：

JS

```
if (Math.random() > 0.5) {
  var x = 1;
} else {
  var x = 2;
}
console.log(x);
```

对学习过块创建作用域的语言（如：C、Java）的开发者而言，上面的代码应该在 `console.log` 这一行抛出一个错误，因为我们在任意一个块的 `x` 作用域的外边。然而，因为块不会为 `var` 创建作用域，所以这里的 `var` 语句实际上创建的是全局变量。下面也介绍了一个实际的例子，解释了和闭包结合时，这个特性如何导致实际问题。

在 ES 6 中，JavaScript 引入了 `let` 和 `const` 声明，这些声明围绕在诸如暂时性死区的其他东西之中，会创建块级作用域的变量。

JS

```
if (Math.random() > 0.5) {
  const x = 1;
} else {
  const x = 2;
}
console.log(x); // ReferenceError: x is not defined
```

从本质上说，在 ES 6 中仅当使用 `let` 或 `const` 声明变量时，块才会认为是作用域。此外，ES 6 引入了模块，模块引入了另一种作用域。闭包能够捕获所有这些作用域中的变量，稍后我们会介绍这些情形。

闭包

注意下面的代码示例：

JS

```
function makeFunc() {
  const name = "Mozilla";
  function displayName() {
    console.log(name);
  }
  return displayName;
}

const myFunc = makeFunc();
myFunc();
```

运行这段代码的效果和之前 `init()` 函数的示例完全一样。其中不同的地方（也是有意思的地方）在于 `displayName()` 内部函数*在执行前*，从外部函数返回。

第一眼看上去，也许不能直观地看出这段代码能够正常运行。在一些编程语言中，函数内的局部变量仅存在于函数的执行期间。一旦 `makeFunc()` 执行完毕，你可能会认为 `name` 变量将不能再被访问。然而，因为代码仍按预期运行，所以在 JavaScript 中情况显然与此不同。

原因在于，JavaScript 中的函数创建了闭包。*闭包*是由函数以及函数声明所在的词法环境组合而成的。该环境包含了这个闭包创建时作用域内的任何局部变量。在本例中，`myFunc` 是执行 `makeFunc` 时创建的 `displayName` 函数的实例引用。`displayName` 的实例有一个它的词法环境的引用，而 `name` 变量位于这个词法环境中。因此，当 `myFunc` 被调用时，`name` 变量仍然可用，其值 `Mozilla` 就被传递到 `console.log` 中。

下面是一个稍微更有意思的示例——一个 `makeAdder` 函数：

JS

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}
```

```
const add5 = makeAdder(5);
const add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

在这个示例中，我们定义了 `makeAdder(x)` 函数，它接受一个 `x` 参数，并返回一个新函数。返回的函数接受一个 `y` 参数，并返回 `x` 和 `y` 的和。

从本质上讲，`makeAdder` 是一个函数工厂。它创建了将指定的值和它的参数相加求和的函数。在上面的示例中，函数工厂创建了两个新函数——一个将其参数和 5 求和，另一个将其参数和 10 求和。

`add5` 和 `add10` 都创建了闭包。它们共享相同的函数体定义，但是保存了不同的词法环境。在 `add5` 的词法环境中，`x` 为 5，而在 `add10` 的词法环境中，`x` 则为 10。

实用的闭包

闭包很有用，因为它能将数据（词法环境）与运算数据的函数关联起来。这显然类似于面向对象编程。在面向对象编程中，对象能将数据（对象的属性）与一个或者多个方法关联起来。

因此，在你通常使用只有一个方法的对象的地方，都可以使用闭包。

在 Web 中，你想要这样做的情况特别常见。在前端 JavaScript 中书写的大部分代码都是基于事件的。你定义某种行为，然后将其添加到由用户触发的事件上（比如点击或者按键）。代码添加为回调（在事件响应中执行的一个函数）。

例如，假设我们想在页面上添加一些可以调整字号的按钮。一种方法是指定 `body` 元素的 font-size（像素单位），然后使用相对的 `em` 单位设置页面上其他元素（例如 header）的字号：

CSS

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}

h2 {
```

```
  font-size: 1.2em;
}
```

这个字号调整按钮能修改 `body` 元素的 `font-size` 属性，并且因为使用的是相对单位，页面上的其他元素也会相应地调整。

以下是 JavaScript：

JS

```
function makeSizer(size) {
  return function () {
    document.body.style.fontSize = `${size}px`;
  };
}

const size12 = makeSizer(12);
const size14 = makeSizer(14);
const size16 = makeSizer(16);
```

`size12`、`size14` 和 `size16` 三个函数将分别把 `body` 的字号调整为 12、14、16 像素。你可以将它们添加到按钮上，就像下面的代码示例那样。

JS

```
document.getElementById("size-12").onclick = size12;
document.getElementById("size-14").onclick = size14;
document.getElementById("size-16").onclick = size16;
```

HTML

```
<button id="size-12">12</button>
<button id="size-14">14</button>
<button id="size-16">16</button>
```

用 JSFiddle 运行这段代码。

用闭包模拟私有方法

像 Java 这样的编程语言支持将方法声明为私有的，即它们只能被同一个类中的其他方法调用。

没有类之前的 JavaScript 没有声明私有方法的原生方式，但使用闭包模拟私有方法是可能的。私有方法不仅有利于限制代码访问，还为管理全局命名空间提供强大能力。

下面的代码展示了如何使用闭包定义能访问私有函数和私有变量的公共函数。注意，这些闭包遵循模块设计模式。

JS

```
const counter = (function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
})();

console.log(counter.value()); // 0

counter.increment();
```

```
counter.increment();
console.log(counter.value()); // 2

counter.decrement();
console.log(counter.value()); // 1
```

在之前的示例中，每个闭包都有它自己的词法环境。而这次，只创建了一个由三个函数共享的词法环境：`Counter.increment`、`Counter.decrement` 和 `Counter.value`。

该共享的词法环境在*立即执行*（也称作 IIFE）的匿名函数体中创建。这个词法环境包含两个私有项：一个名为 `privateCounter` 的变量和一个名为 `changeBy` 的函数。你不能从这个匿名函数外部访问私有成员。相反，你间接地通过匿名函数返回的三个公共函数进行访问。

这三个公共函数创建了共享相同词法环境的闭包。多亏 JavaScript 的词法作用域，它们每个都能访问 `privateCounter` 变量和 `changeBy` 函数。

JS

```
const makeCounter = function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
};

const counter1 = makeCounter();
const counter2 = makeCounter();
```

```
console.log(counter1.value()); // 0.

counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2.

counter1.decrement();
console.log(counter1.value()); // 1.
console.log(counter2.value()); // 0.
```

请注意两个 counter 是如何维护它们各自的独立性的。每个闭包通过它自己的闭包引用不同版本的 `privateCounter` 变量。每次调用其中一个 counter 时，改变这个变量的值会改变它的词法环境。然而对一个闭包中的变量进行修改，不会影响到另外一个闭包中的变量值。

> **备注：**以这种方式使用闭包得到了通常和面向对象编程相关联的好处。特别是*数据隐藏*和*封装*。

闭包作用域链

嵌套函数能访问的外部函数作用域包括外部函数包围的作用域——高效地创建一条函数作用域链。为了解释这一点，注意下面的示例代码。

JS

```
// 全局作用域
const e = 10;
function sum(a) {
  return function (b) {
    return function (c) {
      // 外部函数作用域
      return function (d) {
        // 局部作用域
        return a + b + c + d + e;
      };
    };
  };
```

```
}

console.log(sum(1)(2)(3)(4)); // 20
```

你也可以不用匿名函数：

JS

```
// 全局作用域
const e = 10;
function sum(a) {
  return function sum2(b) {
    return function sum3(c) {
      // 外部函数作用域
      return function sum4(d) {
        // 局部作用域
        return a + b + c + d + e;
      };
    };
  };
}

const sum2 = sum(1);
const sum3 = sum2(2);
const sum4 = sum3(3);
const result = sum4(4);
console.log(result); // 20
```

在上面的示例中，有一系列的嵌套函数，所有的嵌套函数都能访问外部函数的作用域。在这个上下文中，我们说闭包能访问*全部*的外部作用域。

闭包也能捕获块作用域和模块作用域中的变量。例如，下面的示例创建了块级作用域变量 `y` 的闭包：

JS

```
function outer() {
  let getY;
  {
    const y = 6;
    getY = () => y;
  }
  console.log(typeof y); // undefined
  console.log(getY()); // 6
}

outer();
```

模块作用域的闭包更有趣。

JS

```
// myModule.js
let x = 5;
export const getX = () => x;
export const setX = (val) => {
  x = val;
};
```

这里，模块导出一对 getter-setter 函数，它们在模块作用域变量 `x` 上创建了闭包。即便在其他模块中不能直接访问 `x` 的情况下，也能通过函数对 `x` 进行读写。

JS

```
import { getX, setX } from "./myModule.js";

console.log(getX()); // 5
setX(6);
console.log(getX()); // 6
```

也能在导入的值上创建闭包，这认为是*实时绑定*，因为当原始值变化时，导入值也相应地变化。

JS

```
// myModule.js
export let x = 1;
export const setX = (val) => {
  x = val;
};
```

JS

```
// closureCreator.js
import { x } from "./myModule.js";

export const getX = () => x; // 在导入值上创建一个实时绑定
```

JS

```
import { getX } from "./closureCreator.js";
import { setX } from "./myModule.js";

console.log(getX()); // 1
setX(2);
console.log(getX()); // 2
```

在循环中创建闭包：一个常见错误

在引入 `let` 关键字之前，当你在循环中创建闭包时，会发生一个常见的闭包问题，注意下面的代码示例：

HTML

```html
<p id="help">Helpful notes will appear here</p>
<p>Email: <input type="text" id="email" name="email" /></p>
<p>Name: <input type="text" id="name" name="name" /></p>
<p>Age: <input type="text" id="age" name="age" /></p>
```

JS

```js
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    // 罪魁祸首是在这一行使用的 `var`
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function () {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

试着在 JSFiddle 中运行该代码。

`helpText` 数组中定义了三个有用的提示信息，每个都与文档中 input 字段的 ID 关联。循环遍历这些定义，将 `onfocus` 事件与显示帮助信息的方法进行关联。

如果你试着运行这段代码，你会发现它没有达到预期的效果。无论你聚焦在那个字段上，显示的都是关于年龄的信息。

原因是赋值给 `onfocus` 的函数创建了闭包。这些闭包是由函数定义和从 `setupHelp` 函数作用域中捕获的环境所组成的。这三个闭包在循环中创建，但每个都共享同一个词法环境，这个环境有一个不断改变值的变量（`item`）。这是因为 `item` 变量用 `var` 声明，并由于声明提升，因此拥有函数作用域。而 `item.help` 的值是在 `onfocus` 回调执行时决定。因为循环在事件触发之前早已执行完毕，所以 `item` 变量对象（由三个闭包共享）已经指向了 `helpText` 的最后一项。

这个例子的一个解决方案就是使用更多的闭包：特别是使用前面所述的函数工厂：

JS

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function makeHelpCallback(help) {
  return function () {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

使用这个 JSFiddle 链接运行该代码。

这次符合预期。回调不再都共享同一个词法环境，`makeHelpCallback` 函数为每一个回调创建了*一个新的词法环境*，在每个新的词法环境中，`help` 指向 `helpText` 数组中对应的字符串。

另一种方法是使用匿名闭包：

JS

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    (function () {
      var item = helpText[i];
      document.getElementById(item.id).onfocus = function () {
        showHelp(item.help);
      };
    })(); // 立即将事件监听器附着到当前的 item 值（保留到每次迭代）。
  }
}

setupHelp();
```

如果你不想使用过多的闭包，你可以使用 `let` 或 `const` 关键词：

JS

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
```

```
}

function setupHelp() {
  const helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (let i = 0; i < helpText.length; i++) {
    const item = helpText[i];
    document.getElementById(item.id).onfocus = () => {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

这个示例使用 `const` 而不是 `var`，因此每个闭包绑定的是块作用域变量，这意味着不再需要额外的闭包。

另一个可选方案是使用 `forEach()` 遍历 `helpText` 数组并给每一个 `<input>` 添加一个监听器，如下所示：

JS

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];
```

```
  helpText.forEach(function (text) {
    document.getElementById(text.id).onfocus = function () {
      showHelp(text.help);
    };
  });
}

setupHelp();
```

性能考量

正如前面提及的，每个函数实例管理着它自己的作用域和闭包。因此，如果特定的任务不需要使用闭包，在其他函数内不必要地创建函数是不明智的，因为在处理速度和内存消耗两方面将对脚本性能产生负面影响。

例如，在创建一个新对象或类时，方法通常应该关联到对象的原型上，而不是定义到对象的构造函数中。理由是每次调用构造函数时，方法都会重新赋值（也就是每次创建对象时）。

注意下面的例子：

JS

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function () {
    return this.name;
  };

  this.getMessage = function () {
    return this.message;
  };
}
```

因为在上面的代码中并没有利用到在这个特殊的实例中使用闭包的好处，我们应该重写避免使用闭包，修改如下：

JS

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName() {
    return this.name;
  },
  getMessage() {
    return this.message;
  },
};
```

然而，重新定义原型也不推荐。下面的示例是追加到已存在的原型上：

JS

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function () {
  return this.name;
};
MyObject.prototype.getMessage = function () {
  return this.message;
};
```

在前面两个示例中，继承的原型由所有的对象共享，因此方法定义不需要出现在对象创建中。参见继承与原型链了解更多。

Your blueprint for a better internet.