闭包

在本章中，我们将会讨论关于JavaScript被讨论最多的话题——闭包，事实上，这个话题在这之前已经被讨论很多次了，我们将从一个理论的角度去讨论理解它，并且聚焦于在ECMAScript中是如何实现的。

先前的两章作用域链和变量对象的推荐先阅读一遍，因为本章将会使用到之前讨论过的理念。

在直接讨论ECMAScript的闭包之前有必要说明一下在函数式编程中的一般理论的定义。众所周知，在函数式语言中（ECMAScript支持这种范式和文体），函数是数据，比如，他们可以被赋值给变量，作为参数传递给其他函数，以及从其他函数中返回等等，这种函数拥有特殊的名字和结构体。

定义

A *functional argument (“Funarg”)* — is an argument which value is a function.（此句作为定义不做翻译）。

例如：

function exampleFunc(funArg) {

  funArg();

}

exampleFunc(function () {

  console.log('funArg');

});

在这个例子中，实参funarg被同步传给exampleFunc函数

In turn, a function which *receives* another function as the argument is called a *higher-order function (HOF)*.

一个函数接收其他函数作为参数，被称为高阶函数

HOF的另一个名字是“函数式”，或者更接近于数学上的“操作符”的概念，在上面这个例子中，exampleFunc函数就是一个高阶函数。

正如所注意到的那样，一个函数不仅仅能被作为参数传递，也可以从其他函数中作为参数返回。

A function which returns another function is called a function with a functional value (or a function-valued function).

一个返回另外函数的函数被称为有函数返回值的函数。

(function functionValued() {

  return function () {

    console.log('returned function is called');

  };

})()();

函数可以表现为普通数据，例如作为参数传递，接收函数参数或者返回函数类型的值，被称为第一类函数

在ECMAScript中，所有的函数都是第一类函数，一个函数可以接收他本身作为参数，叫做自应用函数。

(function selfApplicative(funArg) {

  if (funArg && funArg === selfApplicative) {

    console.log('self-applicative');

    return;

  }

  selfApplicative(selfApplicative);

})();

返回自身的函数被称为自重复函数，有时候，在文献中使用self-reproducing这个词汇：

(function selfReplicative() {

  return selfReplicative;

})();

自重复函数有一种比较有意思的形式，就是用集合中的单个参数的形式而不是接受集合本身：

// 接收集合的形式

function registerModes(modes) {

  modes.forEach(registerMode, modes);

}

// 调用

registerModes(['roster', 'accounts', 'groups']);

// 使用自重复函数的形式

function modes(mode) {

  registerMode(mode); // 记录一个mode

  return modes; // and return the function itself

}

// 调用：我们只是声明了modes

modes('roster')('accounts')('groups')

但是在实际工作中，使用集合自身会更加高效和直观

定义在传递的函数形式的参数中的本地变量当然会被这个函数的激活对象获取到，因为变量对象在每次进入上下文时被创建用来存储上下文的对象：

function testFn(funArg) {

  // funarg的激活对象是可以获得localVar属性的

  funArg(10); // 20

  funArg(20); // 30

}

testFn(function (arg) {

  let localVar = 10;

  console.log(arg + localVar);

});

但是，正如我们在作用域链中所知道的那样，在ECMAScript中，函数是可以和父函数一起封装的，并且能够使用父上下文的变量对象，这个特性被叫做函数参数问题（funarg problem）。

Funarg问题

在面向堆栈的编程语言中，函数的本地变量被存储在一个堆栈中每当函数被调用时，将这些变量和函数参数push进来。

在函数返回的时候，这些变量被从堆栈中pop，这个模型对于使用函数作为函数值是一个很大的限制（比如，从父函数中返回他们）。大多数情况下，这个问题会出现在使用自由变量的时候。

自由变量就是一个被函数使用的变量，但它既不是一个参数，也不是函数的本地变量。

例子：

function testFn() {

  let localVar = 10;

  function innerFn(innerParam) {

    console.log(innerParam + localVar);

  }

  return innerFn;

}

let someFn = testFn();

someFn(20); // 30

在这个例子中，localVar变量对于innerFn函数来讲是自由变量。

如果这个系统使用面向堆栈的模型来存储本地变量的话，它意味着当从testFn函数返回时，它的所有本地变量将会从堆栈中移除，这将会导致一个在外部激活innerFn函数时会有一个错误。

此外，在这个特殊的例子中，在面向堆栈的实现中，innerFn函数的返回将是不可能的，因为innerFn也是testFn本地的，将会在从testFn返回的时候被移除。

另外一个函数对象的问题和在动态作用域的系统中将函数作为参数传递相关。

例子，伪代码：

let z = 10;

function foo() {

  console.log(z);

}

foo(); // 10 – 在动态和静态作用域中都是10

(function () {

  let z = 20;

  // 备注: 在JS中一直是10

  foo(); // 10 – 静态作用域中, 20 – 动态作用域中

})();

// 作为参数传递时，情况是一样的

(function (funArg) {

  let z = 30;

  funArg(); // 10 – 静态作用域中, 30 – 动态作用域中

})(foo);

在动态作用域系统中，变量解析由动态变量栈管理。因此自由变量在当前活动的动态链中寻找——调用函数的地方，但是不在保存在函数创建时的静态作用域中。

并且这将导致歧义，因此，即使z存在（和之前那个本地变量将会从栈中移除的例子相反），也会存在一个问题：在不同调用foo函数的时候使用哪个z的值（比如哪个上下文中的，哪个作用域中的）？

描述的事件是两种类型的funarg问题——取决于我们处理从函数中返回的函数值，还是将函数型的参数传递给函数。

为了解决这个问题，将会阐述闭包的概念。

闭包

闭包是一个上下文（创建代码块的地方）的代码块和数据的组合，

A closure is a combination of a code block and data of a context in which this code block is created.

我们看一个伪代码的例子：

let x = 20;

function foo() {

  console.log(x); // 变量 "x" = 20

}

// foo函数的闭包

let fooClosure = {

  code: foo // 指向函数引用

  environment: {x: 20}, // 查询自由变量的上下文

};

在上面这个例子中，fooClosure是伪代码，在ECMAScript中，foo函数在创建时就捕获了他上下文的词法环境。

“词法”一词通常被隐藏掉或者省略——在这种情况下，我们说，闭包将其父变量保存在源代码的词法位置，即定义函数的位置。 在下一次激活该函数时，将在此保存的（封闭的）上下文中搜索自由变量，我们可以在上面的示例中看到，其中在ECMAScript中始终应将变量z解析为10。

在定义中，我们使用了一个一般的概念——“代码块”，但是，通常情况下我们都是用“函数”这个词。但是，不是所有的闭包实现通过函数，在Ruby编程语言中，闭包的表现形式可以是一个过程对象，一个lambda表达式或者一个代码块。

至于实现层面，想要在上下文销毁之后保存本地变量，基于栈机构的方式不再适用了（以为它和基于栈结构的定义相违背）。因此，在这种情况下，捕获到的词法环境被存储在动态内存中（在堆heap上，基于heap的实现），使用垃圾回收（GC）。这种系统和基于栈的系统相比速度上很低效，但是实现层面可以做不同的优化，比如，如果数据不是闭包的话，就不把它分配到堆内存。

ECMAScript闭包的实现

理论已经讨论完毕，我们最后直接来看ECMAScript的闭包。在这里我们应该注意到，ECMAScript只使用静态词法作用域（而在一些其他语言中，比如Perl，可以在静态或者动态作用域中声明变量）

let x = 10;

function foo() {

  console.log(x);

}

(function (funArg) {

  let x = 20;

  // funArg的变量x在创建它的（词法）上下文被静态存储

  // from the (lexical) context, in which it was created

  // therefore:

  funArg(); // 10, 而不是20

})(foo);

从技术层面上讲，一个函数的父级环境变量保存在它内部的[[Scope]]属性中，所以，如果我们完全理解了[[Scope]]和作用域链的文章，我们在之前的章节中有过讨论，理解ECMAScript闭包中的问题自己就会消失掉了。

根据函数创建的算法，我们发现在ECMAScript中，所有的函数都是闭包，因为所有的函数在创建时都保存父级上下文的作用域链。最重要的一点就在这，不管函数稍后是否激活，父级的作用域已经在它创建阶段被捕获了：

let x = 10;

function foo() {

  console.log(x);

}

// foo is a closure

foo: <FunctionObject> = {

  [[Call]]: <foo函数代码块>,

  [[Scope]]: [

    global: {

      x: 10

    }

  ],

  ... // 其他属性

};

正如我们提到的那样为了达到优化的目的，当一个函数不使用自由变量的时候，具体实现则不会保存父级作用域链。但是，在ECMAScript定义没有要求这个，所以，正式的情况（并通过技术算法实现），所有的函数在创建阶段都会在[[Scope]]属性中保存作用域链。

一些实现允许直接访问闭包的作用域，比如在Rhino中，函数的[[Scope]]属性对应我们之前在变量对象一章中讨论的非标准的属性\_\_parent\_\_:

var global = this;

var x = 10;

var foo = (function () {

  var y = 20;

  return function () {

    console.log(y);

  };

})();

foo(); // 20

console.log(foo.\_\_parent\_\_.y); // 20

foo.\_\_parent\_\_.y = 30;

foo(); // 30

// 我们可以更进一步到达作用域链的顶部

console.log(foo.\_\_parent\_\_.\_\_parent\_\_ === global); // true

console.log(foo.\_\_parent\_\_.\_\_parent\_\_.x); // 10

所有人共享同一个[[Scope]]

有必要注意一下，在同一个父级上下文中创建的几个内部函数的封装的[[Scope]]在ECMAScript中是相同的，这也就意味着从一个闭包中改变闭包的变量会影响到其他闭包中的变量。

也就是，所有内部函数共享同一个父级环境。

let firstClosure;

let secondClosure;

function foo() {

  let x = 1;

  firstClosure = function () { return ++x; };

  secondClosure = function () { return --x; };

  x = 2; // 影响AO["x"], 这个AO在这两个闭包上

  console.log(firstClosure()); // 3, 通过firstClosure.[[Scope]]

}

foo();

console.log(firstClosure()); // 4

console.log(secondClosure()); // 3

关于这个特性，通常会造成一个很广泛的错误。通常程序员在一个循环中创建函数并尝试每个函数绑定循环的计数变量，期望每个函数保存它自己需要的数值时，会得到预测之外的错误。

var data = [];

for (var k = 0; k < 3; k++) {

  data[k] = function () {

    console.log(k);

  };

}

data[0](); // 3, but not 0

data[1](); // 3, but not 1

data[2](); // 3, but not 2

先前的例子解释这个行为——创建这三个函数的函数，它的上下文作用域对于这三个函数来讲是同一个。每个函数通过[[Scope]]属性引用它，在这个父级作用域中的变量k可被随意的改变。

示例：

activeContext.Scope = [

  ... // 更深层的变量对象

  {data: [...], k: 3} // 激活对象

];

data[0].[[Scope]] === Scope;

data[1].[[Scope]] === Scope;

data[2].[[Scope]] === Scope;

因此，在函数激活的时候，k变量最终被赋值为3。

事实上就是所有变量在代码执行之前即在进入上下文时被创建，这种行为也被认为是“托管”。

创建额外的封装上下文可以帮助来解决这个问题：

var data = [];

for (var k = 0; k < 3; k++) {

  data[k] = (function \_helper(x) {

    return function () {

      console.log(x);

    };

  })(k); // 传递“k”值

}

// 现在是正确的了

data[0](); // 0

data[1](); // 1

data[2](); // 2

让我们来看一下这个例子中发生了什么。

首先，函数\_helper创建后立即使用k作为参数执行。

然后，\_helper函数的返回值也是一个函数，并且刚好是保存了data数组相应的元素。

这个技术做出了下面的行为：被激活时，这个\_helper函数每一次都创建了一个新的有参数x的激活对象，并且这个参数的值是传过来的k变量的值。

因此，被返回函数的作用域是下面这样子的：

data[0].[[Scope]] === [

  ... // higher variable objects

  AO of the parent context: {data: [...], k: 3},

  AO of the \_helper context: {x: 0}

];

data[1].[[Scope]] === [

  ... // higher variable objects

  AO of the parent context: {data: [...], k: 3},

  AO of the \_helper context: {x: 1}

];

data[2].[[Scope]] === [

  ... // higher variable objects

  AO of the parent context: {data: [...], k: 3},

  AO of the \_helper context: {x: 2}

];

我们看到现在函数的[[Scope]]属性有所需要值的引用——通过额外创建的作用域捕获到的x变量。

注意，从被返回的函数中我们引用k变量的话，所有函数值都相同是3是正确的。

通常javascript闭包

通常情况下JavaScript闭包问题不会完全通过上面那种模式来减少——通过创建额外的函数来捕获需要的值。通过实践的角度，这种模式确实是已知的，但是，我们注意到从理论的角度上，ECMAScript中所有的函数都是闭包。

上面讲到的模式不是一个唯一的方式，获取到k变量的值也是可能的，比如用下面这种方法：

var data = [];

for (var k = 0; k < 3; k++) {

  (data[k] = function () {

    console.log(arguments.callee.x);

  }).x = k; // 将k作为函数的属性保存下来

}

// 下面的结果都是正确的

data[0](); // 0

data[1](); // 1

data[2](); // 2

注意：在ES6通过使用let或const关键字声明变量标准化了块级作用域。上面的例子可以简单重写为：

let data = [];

for (let k = 0; k < 3; k++) {

  data[k] = function () {

    console.log(k);

  };

}

// 下面也是正确的输出

data[0](); // 0

data[1](); // 1

data[2](); // 2

Funarg和return

另一个特性就是从函数中返回。在ECMAScript中，一个从闭包中的return语句，返回一个控制流来调用上下文（caller）。在其他语言中，例如在Ruby中，有多种形式的闭包，处理return语句的方式也不同：可能返回给调用者，或者从活动上下文中完全退出。

ECMAScript标准化return行为：

function getElement() {

  [1, 2, 3].forEach(element => {

    if (element % 2 == 0) {

      // 返回到forEach函数而不是从getElement中返回

      console.log('found: ' + element); // found: 2

      return element;

    }

  });

  return null;

}

console.log(getElement()); // null, 而不是 2

但是，在这种情况下，抛出或捕获一些特殊的“break”异常是可以起到作用的：

const $break = {};

function getElement() {

  try {

    [1, 2, 3].forEach(element => {

      if (element % 2 == 0) {

        // 从getElement中返回

        console.log('found: ' + element); // found: 2

        $break.data = element;

        throw $break;

      }

    });

  } catch (e) {

    if (e == $break) {

      return $break.data;

    }

  }

  return null;

}

console.log(getElement()); // 2

理论版本

正如我们提到的那样，通常开发者没有完全理解，例如他们认为闭包只是从父级上下文中返回的内部函数。

让我们再来讲一遍，所有的函数，不管他们是什么类型的：匿名的、命名的、函数表达式或者函数声明，由于作用域链的机制，都是闭包。

对于这个规则有一个例外，就是通过构造函数创建的函数，因为他的[[Scope]]只包含全局对象。

并且，为了正确的阐述这个问题，关于ECMAScript，让我们来提供闭包的两种正确版本：

ECMAScript中的闭包是：

从理论角度：所有函数都是闭包，因为他们在他们在创建阶段就保存了父级上下文的变量。及时一个简单的全局函数，指向一个全局变量引用左右变量，因为使用的是一般作用域链的机制。

从实际角度：这些函数主要特征是：  
当父级上下文结束时仍然存在，比如从父级函数中返回的内部函数。

使用自由变量

闭包的实际应用

在实际情况中，闭包可以创建出很优雅的设计，允许通过funarg（参数）自定义多样的计算方式，其中一个例子就是数组的sort方法，接收一个排序条件的函数作为参数：

[1, 2, 3].sort((a, b) => {

  ... // 排序条件

});

或者，比如像数组的map方法的这类映射功能，根据函数类型参数的条件映射一个新数组：

[1, 2, 3].map(element => {

  return element \* 2;

}); // [2, 4, 6]

通常，定义一个几乎没有限制的查找条件的函数类型参数来实现查找函数是很方便的：

someCollection.find(element => {

  return element.someProperty == 'searchCondition';

});

并且，我们提到一些函数调用，比如forEach方法在数组元素上调用函数：

[1, 2, 3].forEach(element => {

  if (element % 2 != 0) {

    console.log(element);

  }

}); // 1, 3

顺便说一下，函数对象的apply和call方法，也起源于函数式编程的函数调用。我们在之前this值一章中已经讨论了这些方法，在这里，我们看到了它们在应用函数中的作用-函数应用于参数（在应用中应用于参数列表（在应用中以及在调用中定位到参数））：

(function (...args) {

  console.log(args);

}).apply(this, [1, 2, 3])

闭包的另外一个重要应用就是延时调用：

let a = 10;

setTimeout(() => {

  console.log(a); // 10, 一秒后

}, 1000);

还有回调函数：

...

let x = 10;

// only for example

xmlHttpRequestObject.onreadystatechange = function () {

// 回调，在后续会被调用，当数据已经就绪，不管在什么上下文中，变量// x都是可获取到的，他在创建时就已经决定了。

  console.log(x); // 10

};

..

还有类似为了隐藏实现细节，创建封装模块作用域的情况：

// 初始化

const M = (function () {

  // 初始数据

  let x = 10;

  // API.

  return {

    getX() {

      return x;

    },

  };

})();

console.log(M.getX()); // 得到闭包的x – 10

结论

在这篇文章中，我们尝试从一般理论的角度来更多的讨论闭包，即“Funarg problem”，我希望使ECMAScript的闭包更容易被理解。