Chapter 6. Closures

Definitions
定义

函数式参数 —— 是指**值为函数的参数**。

```
function exampleFunc(funArg) {
  funArg();
}
```

funArg 是函数式参数

接受**函数式参数的函数**称为高阶函数。

```
exampleFunc(function () {
  alert('funArg');
});
```

exampleFunc 是高阶函数

以**函数为返回值的函数**称为带函数值的函数。

```
(function functionValued() {
  return function () {
    alert('returned function is called');
  };
})()();
```

functionValued 是以函数为返回值的函数

接受**自己作为参数的函数**，称为自应用函数。

```
(function selfApplicative(funArg) {

  if (funArg && funArg === selfApplicative) {
    alert('self-applicative');
    return;
  }

  selfApplicative(selfApplicative);

})();
```

以**自己为返回值的函数**称为自复制函数。

```
(function selfReplicative() {
  return selfReplicative;
})();
```

在ECMAScript中，函数是可以封装在父函数中的，并**可以使用父函数上下文的变量**。这个特性会引发funarg问题。

Funarg problem
Funarg 问题

自由变量是指**在函数中使用**的，但既**不是函数参数**也**不是函数的局部变量**的变量

```
function testFn() {

  var localVar = 10;

  function innerFn(innerParam) {
    alert(innerParam + localVar);
  }

  return innerFn;
}

var someFn = testFn();
someFn(20); // 30
```

上述例子中，对于innerFn函数来说，localVar就属于自由变量。

两类funarg问题：
1. 取决于是否将**函数以返回值返回**（第一类问题）
2. 以及是否将**函数当函数参数使用**（第二类问题）。

为了解决上述问题，就引入了 闭包的概念。

Closure
闭包

> 闭包是代码块和创建该代码块的上下文中数据的结合。

我的理解：闭包是**函数**和**创建该函数的父级上下文的数据**的结合。

例子：

```
var x = 20;

function foo() {
  alert(x); // 自由变量"x" == 20
}

// 为foo闭包
fooClosure = {
  call: foo // 引用到function
  lexicalEnvironment: {x: 20} // 搜索上下文的上下文
};
```

ECMAScript closures implementation
ECMAScript闭包的实现

再次强调下：ECMAScript只使用**静态（词法）作用域**

根据函数创建的算法，我们看到 在ECMAScript中，**所有的函数都是闭包**，因为它们都是在**创建的时候**就**保存了上层上下文的作用域链**（除开异常的情况）（不管这个函数后续是否会激活 —— [[Scope]]在函数创建的时候就有了）：

```
var x = 10;

function foo() {
  alert(x);
}

// foo是闭包
foo: <FunctionObject> = {
  [[Call]]: <code block of foo>,
  [[Scope]]: [
    global: {
      x: 10
    }
  ],
  ... // 其它属性
};
```

这里只有一类函数除外，那就是通过**Function构造器创建的函数**，因为其[[Scope]]**只包含全局对象**。

Theory versions
理论版本

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
    1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
    2. 在代码中引用了自由变量

JavaScript 的静态作用域链与“动态”闭包链
https://developers.weixin.qq.com/community/develop/article/doc/00066a46588a682cc21ca8a335b013

深入理解JavaScript系列（16）：闭包（Closures）
https://www.cnblogs.com/TomXu/archive/2012/01/31/2330252.html