## JS闭包

MDN(Mozilla Developer Network)的定义：

一个函数和对其周围状态（词法环境，静态的）的引用捆绑在一起（或者说一个函数被引用包围），这样的组合就是闭包；

另一个类似的说法（来自某FaceBook员工）：代码块（ECMAScript中是一个函数），和保存了所有父作用域的静态/词法环境的组合。

也就是说闭包让你可以在一个内层函数中访问到访问到外层函数的作用域。在JavaScript中，每当创建一个函数，闭包就会在函数创建的同时被创建出来。

function init() {

var name = 'Mozilla'; *// name是一个被init创建的局部变量*

function displayName() { *// displayName是一个内部函数，一个闭包*

alert(name); *// 使用了父函数中声明的变量*

}

displayName();

}

init();

--------------------------------分割线------------------------

function init() { // debounce、throttle也可用该原理实现

var name = 'Mozilla'; *// name是一个被init创建的局部变量*

function displayName() { *// displayName是一个内部函数，一个闭包*

alert(name); *// 使用了父函数中声明的变量*

}

return displayName;

}

var myFunc = init();

myFunc();

--------------------------------分割线------------------------

function makeAdder(x) { // 本质上是一个函数工厂

return function(y) {

return x + y;

}

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);

console.log(add5(3)); *// 8*

console.log(add10(5)); *// 15*

**实用的闭包**：

1.

*// js*

function makeSizer(size) {

return function() {

document.body.style.fontSize = size + 'px';

}

}

var size12 = makeSizer(12);

var size14 = makeSizer(14);

var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;

document.getElementById('size-14').onclick = size14;

document.getElementById('size-16').onclick = size16;

*// html*

<a href='#' id={'size-12'}>12</a>

<a href='#' id={'size-14'}>14</a>

<a href='#' id={'size-16'}>16</a>

2.用闭包模拟私有方法

var Counter = ( // 此处声明了一个匿名函数并且立即执行赋值给了Counter

function () {

var privateCounter = 0;

function changeBy(value) {

privateCounter += value;

};

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*

// 我们可以将其存储在makeCounter中，用以生成多个计数器。各自引用其词法作用域中的privateCounter

var makeCounter = function () {

var privateCounter = 0;

function changeBy(value) {

privateCounter += value;

};

return {

increment() {

changeBy(1);

},

decrement() {

changeBy(-1);

},

value() {

return privateCounter;

}

}

}

var counter1 = makeCounter();

var 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*

**常见错误（es2015即es6引入let关键字前）：**

1.在循环中创建闭包：以下代码原意想给arr的每一个字函数在执行时返回循环时对应的下标，但运行时会发现arr中每一个函数执行返回的都是10。这是由于var声明的i变量提升后具有函数作用域。每一次循环产生的闭包共享了同一个词法作用域。所以当它们被执行的时候i已经是10了；

function closureTest() {

var arr = [];

for (var i = 0; i < 10; ++i) {

arr[i] = function() {

return i;

}

}

return arr;

}

var arr = closureTest();

arr[0](); *// 10*

*// ...*

arr[5](); *// 10*

*// ...*

arr[9](); *// 10*

**解决方法**：

1.创建更多的闭包

function makeImmediate(index) {

return function() {

return index;

}

}

function closureTest() {

var arr = [];

for (var i = 0; i < 10; ++i) {

arr[i] = makeImmediate(i); // makeImmediate方法为每一个回调创建一个词法环境

}

return arr;

}

var arr = closureTest();

arr[0](); *// 0*

*// ...*

arr[5](); *// 5*

*// ...*

arr[9](); *// 9*

2.使用匿名闭包

function closureTest() {

var arr = [];

for (var i = 0; i < 10; ++i) {

arr[i] = (function() {

var index = i;

return function() {

return index;

}

})(); // 循环时立即将item项，和回调关联起来

}

return arr;

}

var arr = closureTest();

arr[0](); *// 0*

*// ...*

arr[5](); *// 5*

*// ...*

arr[9](); *// 9*

3.使用let（es2015引入）关键词，避免使用过多闭包

function closureTest() {

var arr = [];

for (let i = 0; i < 10; ++i) { // 使用了let关键词，每个闭包绑定了块作用域变量

arr[i] = function() {

return i;

}

}

return arr;

}

var arr = closureTest();

arr[0](); *// 0*

*// ...*

arr[5](); *// 5*

*// ...*

arr[9](); *// 9*

4.使用forEach

function closureTest() { // 个人感觉本质和使用更多闭包无差

var arr = ['value1','value2','value3'];

arr.forEach(function(value, index) {

arr[index] = function() {

return index;

}

})

return arr;

}

var arr = closureTest();

arr[0](); *// 0*

arr[1](); *// 1*

arr[2](); *// 2*

性能考量：如果特殊需求需要使用闭包，在其他函数中创建函数是不明智的。因为闭包在处理速度和内存消耗方面对脚本有负面影响。

例如：在创建新的对象或者类时，方法通常应该关联对象的原型而不是定义到对象的构造器中。因为这样的话每次调用构造器方法都会被重新赋值一遍（每个对象的创建方法都会被重新赋值）

我们看如下代码：

function MyObject(name, message) { // 此时我们并未利用到闭包的好处

this.name = name.toString();

this.message = message.toString();

this.getName = function() {

return this.name;

};

this.getMessage = function() {

return this.message;

}

}

优化后我们可以这样写：

function MyObject(name, message) {

this.name = name.toString();

this.message = message.toString();

}

MyObject.prototype = { // 直接重新定义原型是不太好的

getName: function() {

return this.name;

},

getMessage: function() {

return this.message;

}

}

最终我们应该这样写：

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;

}