第4章 面向对象编程

课程介绍

- 闭包详解
- 面向对象编程（OOP）
- this变化
- call和apply

4.1 闭包详解

4.1.1 闭包是什么

在 JavaScript 中，闭包是一个让人很难弄懂的概念。ECMAScript 中给闭包的定义是：闭包，指的是词法表示包括不被计算的变量的函数，也就是说，函数可以使用函数之外定义的变量。

只是看官方定义的话，并不是很容易理解他的具体意义，我们来分析一下：

- 闭包是一个函数
- 闭包可以使用在它外面定义的变量
- 闭包存在定义该变量的作用域中

好像有点清晰了，但是使用在它外面定义的变量是什么意思，我们先来看看变量作用域。

4.1.2 变量作用域

变量可分为全局变量和局部变量。全局变量的作用域就是全局性的，在 js 的任何地方都可以使用全局变量。在函数中使用 var 关键字声明变量，这时的变量即是局部变量，它的作用域只在声明该变量的函数内，在函数外面是访问不到该变量的。

实例：

```
<script type="text/javascript">
    var city = '广州';            //全局变量
    var func = function(){
        var company = '前端开发';  //局部变量
        console.log(company);     // 前端开发
        console.log(city);       // 广州
    }
    func();
    console.log(city);             // 广州
    console.log(company);        // Uncaught ReferenceError: company is not defined
</script>
```

4.1.3 变量生存周期

全局变量，生命周期是永久的。局部变量，当定义该变量的函数调用结束时，该变量就会被垃圾回收机制回收而销毁。再次调用该函数时又会重新定义了一个新变量。

实例：

```
<script type="text/javascript">
    var func = function(){
        var a = 'hqjy';
        console.log(a);
    }
    func();
</script>
```

a 为局部变量，在 func 调用完之后，a 就会被销毁了。

实例：

```
<script type="text/javascript">
    var func = function(){
        var a = 'hqjy';
        var func1 = function(){
            a += ' a';
            console.log(a);
        }
        return func1;
    }

    var func2 = func();
    func2();          // hqjy a
    func2();          // hqjy a a
    func2();          // hqjy a a a
</script>
```

可以看出，在第一次调用完 func2 之后，func 中的变量 a 变成 ' hqjy a'，而没有被销毁。因为此时 func1 形成了一个闭包，导致了 a 的生命周期延续了。

这下子闭包就比较明朗了。

- 闭包是一个函数，比如上面的 func1 函数
- 闭包使用其他函数定义的变量，使其不被销毁。比如上面 func1 调用了变量 a
- 闭包存在定义该变量的作用域中，变量 a 存在 func 的作用域中，那么 func1 也必然存在这个作用域中。

现在可以说，满足这三个条件的就是闭包了。

下面我们通过一个简单而又经典的例子来进一步熟悉闭包。

实例：

```
<script type="text/javascript">
    for (var i = 0; i < 4; i++) {
        setTimeout(function () {
            console.log(i)
        }, 0)
    }
</script>
```

我们可能会简单的以为控制台会打印出 0 1 2 3，可事实却打印出了 4 4 4 4，这又是为什么呢？我们发现，setTimeout 函数是异步的，等到函数执行时，for循环已经结束了，此时的 i 的值为 4，所以 function() { console.log(i) } 去找变量 i，只能拿到 4。

我们想起上一个例子中，闭包使 a 变量的值被保存起来了，那么这里我们也可以用闭包把 0 1 2 3 保存起来。

实例：

```
<script type="text/javascript">
    for (var i = 0; i < 4; i++) {
        (function (i) {
            setTimeout(function () {
                console.log(i)
            }, 0)
        })(i)
    }
</script>
```

当 i=0 时，把 0 作为参数传进匿名函数中，此时 function(i){} 此匿名函数中的 i 的值为 0，等到 setTimeout 执行时顺着外层去找 i，这时就能拿到 0。如此循环，就能拿到想要的 0 1 2 3。

4.1.4 内存管理

在闭包中调用局部变量，会导致这个局部变量无法及时被销毁，相当于全局变量一样会一直占用着内存。如果需要回收这些变量占用的内存，可以手动将变量设置为null。

然而在使用闭包的过程中，比较容易形成 JavaScript 对象和 DOM 对象的循环引用，就有可能造成内存泄露。这是因为浏览器的垃圾回收机制中，如果两个对象之间形成了循环引用，那么它们都无法被回收。

实例：

```
<script type="text/javascript">
    function func() {
        var test = document.getElementById('test');
        test.onclick = function () {
            console.log('hello world');
        }
    }
</script>
```

在上面例子中，func 函数中用匿名函数创建了一个闭包。变量 test 是 JavaScript 对象，引用了 id 为 test 的 DOM 对象，DOM 对象的 onclick 属性又引用了闭包，而闭包又可以调用 test ，因而形成了循环引用，导致两个对象都无法被回收。要解决这个问题，只需要把循环引用中的变量设为 null 即可。

实例：

```
<script type="text/javascript">
    function func() {
        var test = document.getElementById('test');
        test.onclick = function () {
            console.log('hello world');
        }
        test = null;
    }
</script>
```

如果在 func 函数中不使用匿名函数创建闭包，而是通过引用一个外部函数，也不会出现循环引用的问题。

实例：

```
<script type="text/javascript">
    function func() {
        var test = document.getElementById('test');
        test.onclick = funcTest;
    }
    function funcTest(){
        console.log('hello world');
    }
</script>
```

4.2 面向对象编程（OOP）

4.2.1 面向对象的定义

面向对象编程（ObjectOrientedProgramming，OOP，面向对象程序设计）是一种计算机编程架构。OOP的一条基本原则是计算机程序是由单个能够起到子程序作用的单元或对象组合而成。OOP达到了软件工程的三个主要目标：重用性、灵活性和扩展性。为了实现整体运算，每个对象都能够接收信息、处理数据和向其它对象发送信息。

4.2.2 与面向过程的区分

- 面向对象是注重于对象自身（谁去做的）
- 面向过程是注重于函数过程（做了什么）

面向过程的解决方法

在面向过程的编程方式中实现“把大象放冰箱”这个问题答案是耳熟能详的，一共分三步：

1. 开门（冰箱）；
2. 装进（冰箱，大象）；
3. 关门（冰箱）。

面向对象的解决方法

1. 冰箱.开门（ ）；
2. 冰箱.装进（大象）；

3. 冰箱.关门（ ）。

4.2.3 JavaScript对象

Javascript是一种基于对象（object-based）的语言，你遇到的所有东西几乎都是对象。但是，它又不是一种真正的面向对象编程（OOP）语言， 因为它的语法中没有class（类）。那么，如果我们要把"属性"（property）和"方法"（method），封装成一个对象，甚至要从原型对象生成一个实例对象，我们应该怎么做呢？

4.2.3.1 构造函数方式

为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。所谓"构造函数"，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

实例：

```
<script type="text/javascript">
    function Cat(name,color){
        this.name=name;
        this.color=color;
    }

    var cat1 = new Cat("大毛","黄色");
    var cat2 = new Cat("二毛","黑色");
    console.log(cat1.name); // 大毛
    console.log(cat1.color); // 黄色
    console.log(cat1.constructor == Cat); //true
    console.log(cat2.constructor == Cat); //true
    console.log(cat1 instanceof Cat); //true
    console.log(cat2 instanceof Cat); //true
</script>
```

比如，猫的原型对象现在可以这样写，我们现在就可以生成实例对象了。这时cat1和cat2会自动含有一个constructor属性，指向它们的构造函数。Javascript还提供了一个instanceof运算符，验证原型对象与实例对象之间的关系。

构造函数的问题

构造函数方法很好用，但是存在一个浪费内存的问题。请看，我们现在为Cat对象添加一个不变的属性type（种类），再添加一个方法eat（吃）。那么，原型对象Cat就变成了下面这样：

实例：

```
<script type="text/javascript">
    function Cat(name,color){
        this.name = name;
        this.color = color;
        this.type = "猫科动物";
        this.eat = function(){console.log("吃老鼠");};
    }

    var cat1 = new Cat("大毛","黄色");
    var cat2 = new Cat ("二毛","黑色");
```

```
    console.log(cat1.type); // 猫科动物
    cat1.eat(); // 吃老鼠
    console.log(cat1.eat == cat2.eat); //false
</script>
```

表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。那就是对于每一个实例对象，type属性和eat()方法都是一模一样的内容，每一次生成一个实例，都必须为重复的内容，多占用一些内存。这样既不环保，也缺乏效率。能不能让type属性和eat()方法在内存中只生成一次，然后所有实例都指向那个内存地址呢？答案是可以的。

4.2.3.2 原型模式

prototype原型介绍

“prototype”字面翻译是“原型”，是javascript实现继承的主要手段。

简单来说就是：prototype是javascript中的函数(function)的一个保留属性，并且它的值是一个对象（我们可以称这个对象为"prototype对象"）。通过以此函数作为构造函数构造出来的对象都自动的拥有构造函数的prototype对象的实例属性和实例方法。

其中的要点是：

prototype是函数(function)的一个必备的保留属性(只要是function,就一定有一个prototype属性)prototype的值是一个对象可以任意修改函数的prototype属性的值。一个对象会自动拥有这个对象的构造函数的prototype的成员属性和方法。

创建对象的流程分析：

```
var now = new Date();
```

1. javascript解析引擎遇到new后，在内存中开辟一块空间并创建了一个空对象，并且将“this”指向这个空对象；
2. javascript解析引擎将这个空对象的*proto*指向后面紧跟着的构造函数默认的prototype对象（一指向到prototype对象后，解析引擎就知道了“噢，这个对象要拥有这个prototype对象的属性和方法了”）；
3. javascript解析引擎执行构造函数体内的代码，也就正式开始对这个空对象进行构造(初始化)的过程(this.name="xxx",this.info=function(){...}等等)；
4. 对象被构造好后，将构造好的对象赋值给= 号左边的变量now。

Prototype模式

Javascript规定，每一个构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。

实例：

```
<script type="text/javascript">
    function Cat(name,color){
        this.name = name;
        this.color = color;
    }
    Cat.prototype.type = "猫科动物";
    Cat.prototype.eat = function(){console.log("吃老鼠")};

    var cat1 = new Cat("大毛","黄色");
```

```
    var cat2 = new Cat("二毛","黑色");
    console.log(cat1.type); // 猫科动物
    cat1.eat(); // 吃老鼠
    console.log(cat1.eat == cat2.eat); //true
</script>
```

这时所有实例的type属性和eat()方法，其实都是同一个内存地址， 指向prototype对象，因此就提高了运行效率。

4.2.3.4.1 利用原型属性自定义内置函数

自定义数组内置方法

自定义两个方法：1、删除指定下标的元素，2、根据内容删除元素。

实例：

```
<script type="text/javascript">
    //删除指定下标的元素
    Array.prototype.removeAt = function(index){
        if(isNaN(index)||index>this.length){
            return;//index输入不合理，直接结束对象
        }
        this.splice(index,1);
    }
    //根据内容删除元素
    Array.prototype.remove = function(obj){
        if(typeof obj != "object"){
            for (var i=0;i<this.length;i++) {
                if(this[i] === obj){
                    this.splice(i,1);
                    --i;
                }
            }
        }
    }

    var arr = [1,2,3,4,5,6];
    arr.removeAt(3);
    console.log(arr);//[1, 2, 3, 5, 6]
    arr.remove(2);
    console.log(arr);//[1, 3, 5, 6]
</script>
```

自定义数字内置方法

自定义一个方法：给数字前面补零，参数为数字的最终位数。

实例：

```
<script type="text/javascript">
    //给数字前面补零，n为数字的最终位数
    Number.prototype.prefixZero = function(n){
        if(n <= this.toString().length){
            return;
        }
        var zero = Array(n).join(0);
        zero += this
        return zero.slice(-n);
    }
    console.log((1).prefixZero(3));//001
</script>
```

自定义数学内置方法

自定义一个方法：传入两个参数，生成一个值在他们之间的随机数。

实例：

```
<script type="text/javascript">
    Math.rndNum = function(min,max){
        if(min > max){
            min = max,max=min;
        }
        return this.floor(this.random()*(max - min + 1) + min);
    }
    console.log(Math.rndNum(20,100));
</script>
```

注意：Math对象和数组对象他们不同，其并不是构造函数创建的对象，所以不具有原型（prototype）

4.2.3.3 混合模式

混合模式：原型模式 + 构造函数模式。混合模式中构造函数模式用于定义实例属性，而原型模式用于定义方法和共享属性。每个实例都会有自己的一份实例属性，但同时又共享着方法，最大限度的节省了内存。另外这种模式还支持传递初始参数。优点甚多。这种模式在ECMAScript中是使用最广泛、认同度最高的一种创建自定义对象的方法。

实例：

```
<script type="text/javascript">
    //混合模式
    function Createperson(name,qq){//构造函数，构造了对象出来（属性）
        this.name=name;
        this.qq=qq;
    };
    //原型（在类上加方法，就是css中的class）
    Createperson.prototype.showName=function(){
        return ("名字是："+this.name);
    };
    Createperson.prototype.showQQ=function(){
        return ("qq是："+this.qq);
```

```
    };
    var obj1=new Createperson('6688jingtian','2405878063');
    var obj2=new Createperson('小明','456465644');

    console.log(obj1.showName==obj2.showName);// true
    console.log(obj2.showQQ())//qq是：456465644
</script>
```

4.2.3.4 基本模式（JSON语法格式）

语法：

```
var ObjectName = {
    field :value,
    field2:value2,
    ....
    fieldN:valueN,
    method1:function(arguments){…},
        …
}
```

field是对象的属性。

value则是对象的值,值可以是任意数据类型，包括函数 。

实例：

```
var Person = {
    v: 1,
    add: function() {
        return this.v++;
    }
};
```

这就是最简单的对象封装了，把两个属性或方法封装在一个对象里面。但是，这样的写法有两个缺点，一是如果多生成几个实例，写起来就非常麻烦；二是实例与原型之间，没有任何办法，可以看出有什么联系。

4.2.3.5 工厂模式

原始模式的改进，上面的基本模式或者说定于语法模式我们可以写一个函数，解决代码重复的问题。

实例：

```
<script type="text/javascript">
    function Cat(name,color) {
        return {
            name:name,
            color:color
        }
    }
</script>
```

然后生成实例对象，就等于是在调用函数：

```
var cat1 = Cat("大毛","黄色");
var cat2 = Cat("二毛","黑色");
```

这种方法的问题依然是，cat1和cat2之间没有内在的联系，不能反映出它们是同一个原型对象的实例。

所以，综合以上，混合模式在ECMAScript中是使用最广泛、认同度最高的一种创建自定义对象的方法。

4.3 this变化

函数的调用方式和调用中的this变化

函数调用方式不同，this 含义也跟着不同。

1. 调用者.函数名(参数列表); // 调用者可以省略
2. 函数引用.call(调用者 , 参数列表);
3. 函数引用.apply(调用者, arguments);

call与apply的本质一样的，只不过apply可以通过arguments来访问当前函数的参数。

```
调用者.xxx(p1, p2 , p3) = xxx.call(调用者, p1 , p2 , p3) = xxx.apply(调用者, arguments)
```

4.3.1 全局方式调用函数

调用全局函数，全局变量或函数都相当于window对象的属性，这里 this绑定到 window对象。下面2个代码等价：

实例：

```
<script type="text/javascript">
    var show = function() {
        console.log("1");
        console.log(this === window);
    }
    show();

    window.show = function() {
        console.log("2");
        console.log(this === window);
    }

    window.show();
</script>
```

4.3.2 全局函数调用内部函数

全局函数调用内部函数，不论有多少级函数嵌套，只要是以函数的方式调用，this 都是绑定 window 对象。

实例：

```
<script type="text/javascript">
    var out = function() {
        console.log(this === window); //true
        var inner = function() {
            console.log(this === window);//true
        }
        inner();
    }
    out();
</script>
```

4.3.3 对象方法方式调用函数

对象方法方式调用函数，以对象方法方式调用函数，this则绑定宿主对象。

实例：

```
<script type="text/javascript">
    var obj = {
        show:function(v) {
            console.log(v);
            console.log(this === obj);//true
        }
    }
    obj.show("hello");
</script>
```

4.3.4 对象方法调用内部函数

分析实例，out()中的this绑定到obj，因为out()是以对象方法方式调用的，this都绑定的是宿主对象。inner()中的this绑定到window,因为inner()是以函数方式调用的。不论有多少级函数嵌套，只要是以函数的方式调用，this都是绑定 window 对象。

实例：

```
<script type="text/javascript">
    var obj = {};
    obj.out = function() {
        console.log(this === obj);
        //私有方法
        var inner = function() {
            console.log("1 this===obj="+(this === obj));//false
            console.log("2 this===window="+(this === window));//true
        }
        inner();
        //对象方法
        this.inner = function() {
            console.log("3 this===obj="+(this === obj)); //true
            console.log("4 this===window="+(this === window));//flase
        }
        this.inner();
    }
```

```
    obj.out();
</script>
```

4.3.5 对象构造函数调用

如果通过new用某个函数，那么这个函数就是一个对象,就会创建一个连接到该函数prototype属性的新对象，再把this绑定到该对象，然后执行该函数，最后返回该对象。

如果返回值，且返回值为[字符串|数字|布尔值]等简单数据类型，该返回值会被丢弃。

如果返回值，且返回值为[对象|函数|数组]等复合类型对象，该函数则会返回该返回值，将丢弃this绑定的对象。

实例：

```
<script type="text/javascript">
    function Person(name, age) {
        this.name = name;
        this.age = age;
        //return new Date();
    }
    var p = new Person('hanfeili', 18);
    //JSON.stringify()简单说就是将js对象转换成json格式的字符串
    var jsonObj = JSON.stringify(p);
    console.log(jsonObj);
</script>
```

4.3.6 bind保持this上下文

在对象里面，所有的 this 都指向对象本身，而在全局作用域定义的变量的 this 指向 Window。

实例：

```
<script type="text/javascript">
    var obj = {
        getThis: function() {
            console.log(this);
        }
    };
    obj.getThis();  // obj
    var getThisCopy = obj.getThis;
    getThisCopy();   // window
</script>
```

通过上述例子我们会发现，虽然是 getThisCopy 是复制了 obj 的 getThis 方法，但是他们所属的对象却不一样。

bind方法

```
fun.bind(thisArgument, argument1, argument2, ...)
```

参数说明：

- thisArgument：在 fun 函数运行时指定的 this 值，如果绑定函数时使用 new 运算符构造的，则该值将被忽略。
- argument1, argument2, ...：指定的参数列表。

返回值：具有指定 this 值和初始参数的给定函数的副本。

实例：

```
<script type="text/javascript">
    var obj = {
        getThis: function(){
            console.log(this);
        }
    }

    var getThisCopy = obj.getThis;
    getThisCopy.bind(obj)();

    var obj = {
        num: 100,
        numFun: function(){
            console.log(this.num);
        }
    }

    var numFunCopy = obj.numFun;
    numFunCopy.bind(obj)();
</script>
```

4.4 call和apply

call和apply函数调用方式不同，this 含义也跟着不同。

- 调用者.函数名(参数列表); // 调用者可以省略
- 函数引用.call(调用者 , 参数列表);
- 函数引用.apply(调用者, arguments)。

call与apply的本质一样的，只不过apply可以通过arguments来访问当前函数的参数。

```
调用者.xxx(p1, p2 , p3) = xxx.call(调用者, p1 , p2 , p3) = xxx.apply(调用者, arguments)
```

4.4.1 apply

```
函数引用.apply(调用者, [数组|arguments]);
```

语法：

```
函数引用.apply([thisObj[,argArray]])
```

如果argArray 不是一个有效的数组或者不是arguments对象，那么将导致一个 TypeError。如果没有提供thisObj和argArray任何一个参数，那么全局对象(window)被用作默认的【第一个参数】，并且无法被传递任何参数。

实例：

```
<script type="text/javascript">
    var joins = function(p1,p2,p3) {
        console.log(this);
        return this+"-"+p1+"-"+p2+"-"+p3;
    }

    var params = ['hello', 'hi', '你好']
    console.log(joins.apply('妹子',params));
</script>
```

用对象的apply方式调用方法的apply/call方法调用方法时，this则被绑定到apply/call方法的第一个参数。apply第2个参数必须是数组否则报异常。

4.4.2 call

```
函数引用.call(调用者,参数列表);
```

语法：

```
函数引用.call([thisObj[,arg1[, arg2[, [,.argN]]]]])
```

call()方法可以用来代替另一个对象调用一个方法。

call()方法可将一个函数的对象上下文从初始的上下文改变为由【thisObj第一个参数】 指定的新对象。如果没有提供【第一个参数】参数，那么全局对象(window)被用作默认的【第一个参数】。说明白一点其实就是更改对象的内部指针，即改变对象的this指向的内容。这在面向对象的JavaScript编程过程中有时是很有用的。

实例：

```
<script type="text/javascript">
    var joins = function(p1,p2,p3) {
        console.log(this);
        return this+"-"+p1+"-"+p2+"-"+p3;
    }

    console.log(joins.call('妹子', 'hello', 'hi', '你好'));
    console.log(joins.call(null, 'hello', 'hi', '你好'));
</script>
```

用对象的call方式调用方法的apply/call方法调用方法时，this则被绑定到apply/call方法的第一个参数。call()函数可以有任意个参数。

4.4.3 call和apply的区别

apply()要求第二个参数必须是数组。否则就会报：

现在异常：Uncaught TypeError:Function.prototype.apply:Arguments list has wrong type。

以前异常：Uncaught TypeError:second argument to Function.prototype.apply must be an array。

call()第二个参数可以是任意类型，并且可以有N个参数。

4.5 课程总结

1.闭包详解

1）闭包是什么

2）变量作用域

3）变量生存周期

4）内存管理

2.面向对象编程（OOP）

1）面向对象的定义

2）与面向过程的区分

3）JavaScript对象

a.基本模式

b.工厂模式

c.构造模式

d.原型模式

e.混合模式

3.this变化

1）全局方式调用函数

2）全局函数调用内部函数

3）对象方法方式调用函数

4）对象方法调用内部函数

5）对象构造函数调用

6）bind保持this上下文

\4. call和apply

1）apply

2）call

3）call和apply的区别

4.6 实训

1.呆呆是一条可爱的小狗(Dog)，它的叫声很好听(wow)，每次看到主人的时候就会乖乖叫一声(yelp)。请写出从这段描述可以得到的对象。

2.有以下题目：

```
function Foo(){
    getName = function(){console.log(1);};
    return this;
}

Foo.getName = function(){console.log(2);};
Foo.prototype.getName = function(){console.log(3);};

var getName = function(){console.log(4);};

function getName(){console.log(5);}
```

不使用浏览器调试的方法，计算以下函数会输出什么？

```
Foo.getName();
getName();
Foo().getName();
getName();
new Foo.getName();
new Foo().getName();
new new Foo().getName();
```