JAVASCRIPT继承详解

这一章我们将会重点介绍JavaScript中几个重要的属性（this、constructor、prototype），这些属性对于我们理解如何实现JavaScript中的类和继承起着至关重要的作用。

this

this表示当前对象，如果在全局作用范围内使用this，则指代当前页面对象window；如果在函数中使用this，则this指代什么是根据运行时此函数在什么对象上被调用。我们还可以使用apply和call两个全局方法来改变函数中this的具体指向。

先看一个在全局作用范围内使用this的例子：

函数中的this是在运行时决定的，而不是函数定义时，如下：

```
//  定义一个全局函数
        function foo() {
            console.log(this.fruit);
        }
        //  定义一个全局变量，等价于window.fruit = "apple";
        var fruit = "apple";
        //  此时函数foo中this指向window对象
        //  这种调用方式和window.foo();是完全等价的
        foo();  // "apple"

        //  自定义一个对象，并将此对象的属性foo指向全局函数foo
        var pack = {
            fruit: "orange",
            foo: foo
        };
        //  此时函数foo中this指向window.pack对象
        pack.foo(); // "orange"
```

全局函数apply和call可以用来改变函数中this的指向，如下：

```
//  定义一个全局函数
```

```
        function foo() {
            console.log(this.fruit);
        }

        // 定义一个全局变量
        var fruit = "apple";
        // 自定义一个对象
        var pack = {
            fruit: "orange"
        };

        // 等价于window.foo();
        foo.apply(window);  // "apple"
        // 此时foo中的this === pack
        foo.apply(pack);    // "orange"
```

注：apply和call两个函数的作用相同，唯一的区别是两个函数的参数定义不同。

因为在JavaScript中函数也是对象，所以我们可以看到如下有趣的例子：

```
// 定义一个全局函数
        function foo() {
            if (this === window) {
                console.log("this is window.");
            }
        }

        // 函数foo也是对象，所以可以定义foo的属性boo为一个函数
        foo.boo = function() {
            if (this === foo) {
                console.log("this is foo.");
            } else if (this === window) {
                console.log("this is window.");
            }
        };
        // 等价于window.foo();
        foo();  // this is window.

        // 可以看到函数中this的指向调用函数的对象
        foo.boo();  // this is foo.

        // 使用apply改变函数中this的指向
```

```
        foo.boo.apply(window);  // this is window.
```

prototype

我们已经在第一章中使用prototype模拟类和继承的实现。 prototype本质上还是一个JavaScript对象。并且每个函数都有一个默认的prototype属性。

如果这个函数被用在创建自定义对象的场景中，我们称这个函数为构造函数。比如下面一个简单的场景：

```
// 构造函数
        function Person(name) {
            this.name = name;
        }
        // 定义Person的原型，原型中的属性可以被自定义对象引用
        Person.prototype = {
            getName: function() {
                return this.name;
            }
        }
        var zhang = new Person("ZhangSan");
        console.log(zhang.getName());   // "ZhangSan"
```

作为类比，我们考虑下JavaScript中的数据类型 – 字符串（String）、数字（Number）、数组（Array）、对象（Object）、日期（Date）等。我们有理由相信，在JavaScript内部这些类型都是作为构造函数来实现的，比如：

```
// 定义数组的构造函数，作为JavaScript的一种预定义类型
        function Array() {
            // ...
        }

        // 初始化数组的实例
        var arr1 = new Array(1, 56, 34, 12);
        // 但是，我们更倾向于如下的语法定义：
        var arr2 = [1, 56, 34, 12];
```

同时对数组操作的很多方法（比如concat、join、push）应该也是在prototype属性中定义的。

实际上，JavaScript所有的固有数据类型都具有只读的prototype属性（这是可以理解的：因为如果修改了这些类型的prototype属性，则哪些预定义的方法就消失了），但是我们可以向其中添加自己的扩展方法。

```
// 向JavaScript固有类型Array扩展一个获取最小值的方法
        Array.prototype.min = function() {
```

```
            var min = this[0];
            for (var i = 1; i < this.length; i++) {
                if (this[i] < min) {
                    min = this[i];
                }
            }
            return min;
        };

        // 在任意Array的实例上调用min方法
        console.log([1, 56, 34, 12].min());  // 1
```

注意：这里有一个陷阱，向Array的原型中添加扩展方法后，当使用for-in循环数组时，这个扩展方法也会被循环出来。

下面的代码说明这一点（假设已经向Array的原型中扩展了min方法）：

```
var arr = [1, 56, 34, 12];
        var total = 0;
        for (var i in arr) {
            total += parseInt(arr[i], 10);
        }
        console.log(total);   // NaN
```

解决方法也很简单：

```
var arr = [1, 56, 34, 12];
        var total = 0;
        for (var i in arr) {
            if (arr.hasOwnProperty(i)) {
                total += parseInt(arr[i], 10);
            }
        }
        console.log(total);   // 103
```

constructor

constructor始终指向创建当前对象的构造函数。比如下面例子：

```
// 等价于 var foo = new Array(1, 56, 34, 12);
        var arr = [1, 56, 34, 12];
        console.log(arr.constructor === Array); // true
        // 等价于 var foo = new Function();
```

```
        var Foo = function() { };
        console.log(Foo.constructor === Function); //
true
        //  由构造函数实例化一个obj对象
        var obj = new Foo();
        console.log(obj.constructor === Foo); // true

        //  将上面两段代码合起来，就得到下面的结论
        console.log(obj.constructor.constructor ===
Function); // true
```

但是当constructor遇到prototype时，有趣的事情就发生了。

我们知道每个函数都有一个默认的属性prototype，而这个prototype的constructor默认指向这个函数。如下例所示：

```
function Person(name) {
            this.name = name;
        };
        Person.prototype.getName = function() {
            return this.name;
        };
        var p = new Person("ZhangSan");

        console.log(p.constructor === Person);  // true
        console.log(Person.prototype.constructor ===
Person); // true
        //  将上两行代码合并就得到如下结果
        console.log(p.constructor.prototype.constructor
=== Person); // true
```

当时当我们重新定义函数的prototype时（注意：和上例的区别，这里不是修改而是覆盖）， constructor的行为就有点奇怪了，如下示例：

```
function Person(name) {
            this.name = name;
        };
        Person.prototype = {
            getName: function() {
                return this.name;
            }
        };
        var p = new Person("ZhangSan");
```

```
        console.log(p.constructor === Person);  //
false
        console.log(Person.prototype.constructor ===
Person); // false
        console.log(p.constructor.prototype.constructor
=== Person); // false
```

为什么呢?

原来是因为覆盖Person.prototype时，等价于进行如下代码操作:

```
Person.prototype = new Object({
            getName: function() {
                return this.name;
            }
        });
```

而constructor始终指向创建自身的构造函数，所以此时Person.prototype.constructor === Object, 即是:

```
function Person(name) {
            this.name = name;
        };
        Person.prototype = {
            getName: function() {
                return this.name;
            }
        };
        var p = new Person("ZhangSan");
        console.log(p.constructor === Object);  // true
        console.log(Person.prototype.constructor ===
Object); // true
        console.log(p.constructor.prototype.constructor
=== Object); // true
```

怎么修正这种问题呢? 方法也很简单，重新覆盖Person.prototype.constructor即可:

```
function Person(name) {
            this.name = name;
        };
        Person.prototype = new Object({
            getName: function() {
```

```
                return this.name;
            }
        });
        Person.prototype.constructor = Person;
        var p = new Person("ZhangSan");
        console.log(p.constructor === Person);  // true
        console.log(Person.prototype.constructor ===
Person); // true
        console.log(p.constructor.prototype.constructor
=== Person); // true
```