前端基础进阶（九）：详解面向对象、构造函数、原型与原型链

.

如果要我总结一下学习前端以来我遇到了哪些瓶颈，那么面向对象一定是第一个毫不犹豫想到的。尽管我现在对于面向对象有了一些的了解，但是当初的那种似懂非懂的痛苦，依然历历在目。

为了帮助大家能够更加直观的学习和了解面向对象，我会用尽量简单易懂的描述来展示面向对象的相关知识。并且也准备了一些实用的例子帮助大家更加快速的掌握面向对象的真谛。

- jQuery的面向对象实现
- 封装拖拽
- 简易版运动框架封装

这可能会花一点时间，但是却值得期待。所以如果有兴趣的朋友可以来简书和公众号关注我。

而这篇文章主要来聊一聊关于面向对象的一些重要的基本功。

一、对象的定义

在ECMAScript-262中，对象被定义为**“无序属性的集合，其属性可以包含基本值，对象或者函数”**。

也就是说，在JavaScript中，对象无非就是由一些列无序的`key-value`对组成。其中value可以是基本值，对象或者函数。

```
// 这里的person就是一个对象
var person = {
    name: 'Tom',
    age: 18,
    getName: function() {},
    parent: {}
}
```

创建对象

我们可以通过new的方式创建一个对象。

```
var obj = new Object();
```

也可以通过对象字面量的形式创建一个简单的对象。

```
var obj = {};
```

当我们想要给我们创建的简单对象添加方法时，可以这样表示。

```
// 可以这样
```

```
var person = {};
person.name = "TOM";
person.getName = function() {
    return this.name;
}

// 也可以这样
var person = {
    name: "TOM",
    getName: function() {
        return this.name;
    }
}
```

访问对象的属性和方法

假如我们有一个简单的对象如下：

```
var person = {
    name: 'TOM',
    age: '20',
    getName: function() {
        return this.name
    }
}
```

当我们想要访问他的name属性时，可以用如下两种方式访问。

```
person.name

// 或者
person['name']
```

如果我们想要访问的属性名是一个变量时，常常会使用第二种方式。例如我们要同时访问person的name与age，可以这样写：

```
['name', 'age'].forEach(function(item) {
    console.log(person[item]);
```

```
})
```

> 这种方式一定要重视，记住它以后在我们处理复杂数据的时候会有很大的帮助。

二、工厂模式

使用上面的方式创建对象很简单，但是在很多时候并不能满足我们的需求。就以person对象为例。假如我们在实际开发中，不仅仅需要一个名字叫做TOM的person对象，同时还需要另外一个名为Jake的person对象，虽然他们有很多相似之处，但是我们不得不重复写两次。

```
var perTom = {
    name: 'TOM',
    age: 20,
    getName: function() {
        return this.name
    }
};

var perJake = {
    name: 'Jake',
    age: 22,
    getName: function() {
        return this.name
    }
}
```

很显然这并不是合理的方式，当相似对象太多时，大家都会崩溃掉。

我们可以使用工厂模式的方式解决这个问题。顾名思义，工厂模式就是我们提供一个模子，然后通过这个模子复制出我们需要的对象。我们需要多少个，就复制多少个。

```
var createPerson = function(name, age) {

    // 声明一个中间对象，该对象就是工厂模式的模子
    var o = new Object();

    // 依次添加我们需要的属性与方法
    o.name = name;
    o.age = age;
    o.getName = function() {
        return this.name;
    }

    return o;
}

// 创建两个实例
var perTom = createPerson('TOM', 20);
var PerJake = createPerson('Jake', 22);
```

相信上面的代码并不难理解，也不用把工厂模式看得太过高大上。很显然，工厂模式帮助我们解决了重复代码上的麻烦，让我们可以写很少的代码，就能够创建很多个person对象。但是这里还有两个麻烦，需要我们注意。

第一个麻烦就是这样处理，我们没有办法识别对象实例的类型。使用instanceof可以识别对象的类型，如下例子：

```
var obj = {};
var foo = function() {}

console.log(obj instanceof Object);  // true
console.log(foo instanceof Function); // true
```

因此在工厂模式的基础上，我们需要使用构造函数的方式来解决这个麻烦。

三、构造函数

在JavaScript中，new关键字可以让一个函数变得与众不同。通过下面的例子，我们来一探new关键字的神奇之处。

```
function demo() {
    console.log(this);
}

demo();  // window
new demo();  // demo
```

为了能够直观的感受他们不同，建议大家动手实践观察一下。很显然，使用new之后，函数内部发生了一些变化，让this指向改变。那么new关键字到底做了什么事情呢。嗯，其实我之前在文章里用文字大概表达了一下new到底干了什么，但是一些同学好奇心很足，总期望用代码实现一下，我就大概以我的理解来表达一下吧。

```
// 先一本正经的创建一个构造函数，其实该函数与普通函数并无区别
var Person = function(name, age) {
    this.name = name;
    this.age = age;
    this.getName = function() {
        return this.name;
    }
}

// 将构造函数以参数形式传入
function New(func) {

    // 声明一个中间对象，该对象为最终返回的实例
    var res = {};
    if (func.prototype !== null) {

        // 将实例的原型指向构造函数的原型
        res.__proto__ = func.prototype;
    }

    // ret为构造函数执行的结果，这里通过apply，将构造函数内部的this指向修改为指
    var ret = func.apply(res, Array.prototype.slice.call(arguments, 1)
```

```
    // 当我们在构造函数中明确指定了返回对象时，那么new的执行结果就是该返回对象
    if ((typeof ret === "object" || typeof ret === "function") && ret
        return ret;
    }

    // 如果没有明确指定返回对象，则默认返回res，这个res就是实例对象
    return res;
}

// 通过new声明创建实例，这里的p1，实际接收的正是new中返回的res
var p1 = New(Person, 'tom', 20);
console.log(p1.getName());

// 当然，这里也可以判断出实例的类型了
console.log(p1 instanceof Person); // true
```

> JavaScript内部再通过其他的一些特殊处理，将`var p1 = New(Person, 'tom', 20);`等效于`var p1 = new Person('tom', 20);`。就是我们认识的new关键字了。具体怎么处理的，我也不知道，别刨根问底了，一直回答下去太难 - -!

老实讲，你可能很难在其他地方看到有如此明确的告诉你new关键字到底对构造函数干了什么的文章了。理解了这段代码，你对JavaScript的理解又比别人深刻了一分，所以，一本正经厚颜无耻求个赞可好?

当然，很多朋友由于对于前面几篇文章的知识理解不够到位，会对new的实现表示非常困惑。但是老实讲，如果你读了我的前面几篇文章，一定会对这里new的实现有似曾相识的感觉。而且我这里已经尽力做了详细的注解，剩下的只能靠你自己了。

但是只要你花点时间，理解了他的原理，那么困扰了无数人的构造函数中this到底指向谁就变得非常简单了。

所以，为了能够判断实例与对象的关系，我们就使用构造函数来搞定。

```
var Person = function(name, age) {
    this.name = name;
    this.age = age;
    this.getName = function() {
        return this.name;
    }
}

var p1 = new Person('Ness', 20);
console.log(p1.getName());  // Ness

console.log(p1 instanceof Person); // true
```

关于构造函数，如果你暂时不能够理解new的具体实现，就先记住下面这几个结论吧。

- 与普通函数相比，构造函数并没有任何特别的地方，首字母大写只是我们约定的小规定，用于区分普通函数；
- new关键字让构造函数具有了与普通函数不同的许多特点，而new的过程中，执行了如下过程：
    1. 声明一个中间对象；
    2. 将该中间对象的原型指向构造函数的原型；
    3. 将构造函数的this，指向该中间对象；
    4. 返回该中间对象，即返回实例对象。

四、原型

虽然构造函数解决了判断实例类型的问题，但是，说到底，还是一个对象的复制过程。跟工厂模式颇有相似之处。也就是说，当我们声明了100个person对象，那么就有100个getName方法被重新生成。

这里的每一个getName方法实现的功能其实是一模一样的，但是由于分别属于不同的实例，就不得不一直不停的为getName分配空间。这就是工厂模式存在的第二个麻烦。

显然这是不合理的。我们期望的是，既然都是实现同一个功能，那么能不能就让每一个实例对象都访问同一个方法?

当然能，这就是原型对象要帮我们解决的问题了。

我们创建的每一个函数，都可以有一个prototype属性，该属性指向一个对象。这个对象，就是我们这里说的原型。

当我们在创建对象时，可以根据自己的需求，选择性的将一些属性和方法通过prototype属性，挂载在原型对象上。而每一个new出来的实例，都有一个__proto__属性，该属性指向构造函数的原型对象，通过这个属性，让实例对象也能够访问原型对象上的方法。因此，当所有的实例都能够通过__proto__访问到原型对象时，原型对象的方法与属性就变成了共有方法与属性。

我们通过一个简单的例子与图示，来了解构造函数，实例与原型三者之间的关系。

> 由于每个函数都可以是构造函数，每个对象都可以是原型对象，因此如果在理解原型之初就想的太多太复杂的话，反而会阻碍你的理解，这里我们要学会先简化它们。就单纯的剖析这三者的关系。

```
// 声明构造函数
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// 通过prototye属性，将方法挂载到原型对象上
Person.prototype.getName = function() {
    return this.name;
}

var p1 = new Person('tim', 10);
var p2 = new Person('jak', 22);
```

```
console.log(p1.getName === p2.getName); // true
```

通过图示我们可以看出，构造函数的prototype与所有实例对象的__proto__都指向原型对象。而原型对象的constructor指向构造函数。

除此之外，还可以从图中看出，实例对象实际上对前面我们所说的中间对象的复制，而中间对象中的属性与方法都在构造函数中添加。于是根据构造函数与原型的特性，我们就可以将在构造函数中，通过this声明的属性与方法称为私有变量与方法，它们被当前被某一个实例对象所独有。而通过原型声明的属性与方法，我们可以称之为共有属性与方法，它们可以被所有的实例对象访问。

当我们访问实例对象中的属性或者方法时，会优先访问实例对象自身的属性和方法。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.getName = function() {
        console.log('this is constructor.');
    }
}

Person.prototype.getName = function() {
    return this.name;
}

var p1 = new Person('tim', 10);

p1.getName(); // this is constructor.
```

在这个例子中，我们同时在原型与构造函数中都声明了一个getName函数，运行代码的结果表示原型中的访问并没有被访问。

我们还可以通过in来判断，一个对象是否拥有某一个属性/方法，无

论是该属性/方法存在与实例对象还是原型对象。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.getName = function() {
    return this.name;
}

var p1 = new Person('tim', 10);

console.log('name' in p1); // true
```

in的这种特性最常用的场景之一，就是判断当前页面是否在移动端打开。

```
isMobile = 'ontouchstart' in document;

// 很多人喜欢用浏览器UA的方式来判断，但并不是很好的方式
```

** 更简单的原型写法 **

根据前面例子的写法，如果我们要在原型上添加更多的方法，可以这样写：

```
function Person() {}

Person.prototype.getName = function() {}
Person.prototype.getAge = function() {}
Person.prototype.sayHello = function() {}
... ...
```

除此之外，我还可以使用更为简单的写法。

```
function Person() {}

Person.prototype = {
    constructor: Person,
    getName: function() {},
    getAge: function() {},
    sayHello: function() {}
}
```

这种字面量的写法看上去简单很多，但是有一个需要特别注意的地方。`Person.prototype = {}`实际上是重新创建了一个`{}`对象并赋值给Person.prototype，这里的`{}`并不是最初的那个原型对象。因此它里面并不包含`constructor`属性。为了保证正确性，我们必须在新创建的`{}`对象中显示的设置`constructor`的指向。即上面的`constructor: Person`。

五、原型链

原型对象其实也是普通的对象。几乎所有的对象都可能是原型对象，也可能是实例对象，而且还可以同时是原型对象与实例对象。这样的一个对象，正是构成原型链的一个节点。因此理解了原型，那么原型链并不是一个多么复杂的概念。

我们知道所有的函数都有一个叫做toString的方法。那么这个方法到底是在哪里的呢?

先随意声明一个函数：

```
function add() {}
```

那么我们可以用如下的图来表示这个函数的原型链。

其中add是Function对象的实例。而Function的原型对象同时又是Object原型的实例。这样就构成了一条原型链。原型链的访问，其实

跟作用域链有很大的相似之处，他们都是一次单向的查找过程。因此实例对象能够通过原型链，访问到处于原型链上对象的所有属性与方法。这也是foo最终能够访问到处于Object原型对象上的toString方法的原因。

基于原型链的特性，我们可以很轻松的实现**继承**。

六、继承

我们常常结合构造函数与原型来创建一个对象。因为构造函数与原型的不同特性，分别解决了我们不同的困扰。因此当我们想要实现继承时，就必须得根据构造函数与原型的不同而采取不同的策略。

我们声明一个Person对象，该对象将作为父级，而子级cPerson将要继承Person的所有属性与方法。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

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

首先我们来看构造函数的继承。在上面我们已经理解了构造函数的本质，它其实是在new内部实现的一个复制过程。而我们在继承时想要的，就是想父级构造函数中的操作在子级的构造函数中重现一遍即可。我们可以通过call方法来达到目的。

```
// 构造函数的继承
function cPerson(name, age, job) {
    Person.call(this, name, age);
    this.job = job;
}
```

而原型的继承，则只需要将子级的原型对象设置为父级的一个实例，加入到原型链中即可。

```
// 继承原型
cPerson.prototype = new Person(name, age);

// 添加更多方法
cPerson.prototype.getLive = function() {}
```

当然关于继承还有更好的方式。

七、更好的继承

假设原型链的终点`Object.prototype`为原型链的E(end)端，原型链的起点为S(start)端。

通过前面原型链的学习我们知道，处于S端的对象，可以通过S -> E的单向查找，访问到原型链上的所有方法与属性。因此这给继承提供了理论基础。我们只需要在S端添加新的对象，那么新对象就能够通过原型链访问到父级的方法与属性。因此想要实现继承，是一件非常简单的事情。

因为封装一个对象由构造函数与原型共同组成，因此继承也会分别有构造函数的继承与原型的继承。

假设我们已经封装好了一个父类对象Person。如下。

```
var Person = function(name, age) {
    this.name = name;
    this.age = age;
}

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

```
Person.prototype.getAge = function() {
    return this.age;
}
```

构造函数的继承比较简单，我们可以借助call/apply来实现。假设我们要通过继承封装一个Student的子类对象。那么构造函数可以如下实现。

```
var Student = function(name, age, grade) {
    // 通过call方法还原Person构造函数中的所有处理逻辑
    Student.call(Person, name, age);
    this.grade = grade;
}


// 等价于
var Student = function(name, age, grade) {
    this.name = name;
    this.age = age;
    this.grade = grade;
}
```

原型的继承则稍微需要一点思考。首先我们应该考虑，如何将子类对象的原型加入到原型链中？我们只需要让子类对象的原型，成为父类对象的一个实例，然后通过__proto__就可以访问父类对象的原型。这样就继承了父类原型中的方法与属性了。

因此我们可以先封装一个方法，该方法根据父类对象的原型创建一个实例，该实例将会作为子类对象的原型。

```
function create(proto, options) {
    // 创建一个空对象
    var tmp = {};

    // 让这个新的空对象成为父类对象的实例
    tmp.__proto__ = proto;
```

```
    // 传入的方法都挂载到新对象上，新的对象将作为子类对象的原型
    Object.defineProperties(tmp, options);
    return tmp;
}
```

简单封装了create对象之后，我们就可以使用该方法来实现原型的继承了。

```
Student.prototype = create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
    getGrade: {
        value: function() {
            return this.grade
        }
    }
})
```

那么我们来验证一下我们这里实现的继承是否正确。

```
var s1 = new Student('ming', 22, 5);

console.log(s1.getName());  // ming
console.log(s1.getAge());   // 22
console.log(s1.getGrade()); // 5
```

全部都能正常访问，没问题。在ECMAScript5中直接提供了一个Object.create方法来完成我们上面自己封装的create的功能。因此我们可以直接使用Object.create.

```
Student.prototype = create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
```

```
    getGrade: {
        value: function() {
            return this.grade
        }
    }
})
```

完整代码如下：

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.getName = function() {
    return this.name
}
Person.prototype.getAge = function() {
    return this.age;
}

function Student(name, age, grade) {
    // 构造函数继承
    Person.call(this, name, age);
    this.grade = grade;
}

// 原型继承
Student.prototype = Object.create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
    getGrade: {
        value: function() {
            return this.grade
        }
    }
})


var s1 = new Student('ming', 22, 5);
```

```
console.log(s1.getName());  // ming
console.log(s1.getAge());   // 22
console.log(s1.getGrade()); // 5
```

八、属性类型

在上面的继承实现中，使用了一个大家可能不太熟悉的方法defineProperties。并且在定义getGrade时使用了一个很奇怪的方式。

```
getGrade: {
    value: function() {
        return this.grade
    }
}
```

这其实是对象中的属性类型。在我们平常的使用中，给对象添加一个属性时，直接使用object.param的方式就可以了，或者直接在对象中挂载。

```
var person = {
    name: 'TOM'
}
```

在ECMAScript5中，对每个属性都添加了几个属性类型，来描述这些属性的特点。他们分别是

- **configurable**: 表示该属性是否能被delete删除。当其值为false时，其他的特性也不能被改变。默认值为true
- **enumerable**: 是否能枚举。也就是是否能被for-in遍历。默认值为true
- **writable**: 是否能修改值。默认为true
- **value**: 该属性的具体值是多少。默认为undefined

- **get**: 当我们通过person.name访问name的值时，get将被调用。该方法可以自定义返回的具体值时多少。get默认值为undefined
- **set**: 当我们通过person.name = 'Jake'设置name的值时，set方法将被调用。该方法可以自定义设置值的具体方式。set默认值为undefined

> 需要注意的是，不能同时设置value、writable 与 get、set的值。

我们可以通过Object.defineProperty方法来修改这些属性类型。

下面我们用一些简单的例子来演示一下这些属性类型的具体表现。

configurable

```
// 用普通的方式给person对象添加一个name属性，值为TOM
var person = {
    name: 'TOM'
}

// 使用delete删除该属性
delete person.name;  // 返回true 表示删除成功

// 通过Object.defineProperty重新添加name属性
// 并设置name的属性类型的configurable为false，表示不能再用delete删除
Object.defineProperty(person, 'name', {
    configurable: false,
    value: 'Jake'  // 设置name属性的值
})

// 再次delete，已经不能删除了
delete person.name   // false

console.log(person.name)    // 值为Jake

// 试图改变value
person.name = "alex";
console.log(person.name) // Jake 改变失败
```

enumerable

```
var person = {
    name: 'TOM',
    age: 20
}

// 使用for-in枚举person的属性
var params = [];

for(var key in person) {
    params.push(key);
}

// 查看枚举结果
console.log(params);  // ['name', 'age']

// 重新设置name属性的类型，让其不可被枚举
Object.defineProperty(person, 'name', {
    enumerable: false
})

var params_ = [];
for(var key in person) {
    params_.push(key)
}

// 再次查看枚举结果
console.log(params_); // ['age']
```

writable

```
var person = {
    name: 'TOM'
}

// 修改name的值
person.name = 'Jake';

// 查看修改结果
console.log(person.name); // Jake 修改成功
```

```
// 设置name的值不能被修改
Object.defineProperty(person, 'name', {
    writable: false
})

// 再次试图修改name的值
person.name = 'alex';

console.log(person.name); // Jake 修改失败
```

value

```
var person = {}

// 添加一个name属性
Object.defineProperty(person, 'name', {
    value: 'TOM'
})

console.log(person.name)  // TOM
```

get/set

```
var person = {}

// 通过get与set自定义访问与设置name属性的方式
Object.defineProperty(person, 'name', {
    get: function() {
        // 一直返回TOM
        return 'TOM'
    },
    set: function(value) {
        // 设置name属性时，返回该字符串，value为新值
        console.log(value + ' in set');
    }
})

// 第一次访问name，调用get
console.log(person.name)   // TOM
```

```
// 尝试修改name值，此时set方法被调用
person.name = 'alex'   // alex in set

// 第二次访问name，还是调用get
console.log(person.name) // TOM
```

> 请尽量同时设置get、set。如果仅仅只设置了get，那么我们将无法设置该属性值。如果仅仅只设置了set，我们也无法读取该属性的值。

Object.defineProperty只能设置一个属性的属性特性。当我们想要同时设置多个属性的特性时，需要使用我们之前提到过的Object.defineProperties

```
var person = {}

Object.defineProperties(person, {
    name: {
        value: 'Jake',
        configurable: true
    },
    age: {
        get: function() {
            return this.value || 22
        },
        set: function(value) {
            this.value = value
        }
    }
})

person.name   // Jake
person.age    // 22
```

读取属性的特性值

我们可以使用Object.getOwnPropertyDescriptor方法读取某一个属

性的特性值。

```
var person = {}

Object.defineProperty(person, 'name', {
    value: 'alex',
    writable: false,
    configurable: false
})

var descripter = Object.getOwnPropertyDescriptor(person, 'name');

console.log(descripter);  // 返回结果如下

descripter = {
    configurable: false,
    enumerable: false,
    value: 'alex',
    writable: false
}
```

九、总结

关于面向对象的基础知识大概就是这些了。我从最简单的创建一个对象开始，解释了为什么我们需要构造函数与原型，理解了这其中的细节，有助于我们在实际开发中灵活的组织自己的对象。因为我们并不是所有的场景都会使用构造函数或者原型来创建对象，也许我们需要的对象并不会声明多个实例，或者不用区分对象的类型，那么我们就可以选择更简单的方式。

我们还需要关注构造函数与原型的各自特性，有助于我们在创建对象时准确的判断我们的属性与方法到底是放在构造函数中还是放在原型中。如果没有理解清楚，这会给我们在实际开发中造成非常大的困扰。

最后接下来的几篇文章，我会挑几个面向对象的例子，继续帮助大家掌握面向对象的实际运用。