对象的继承

面向对象编程很重要的一个方面，就是对象的继承。A 对象通过继承 B 对象，就能直接拥有 B 对象的所有属性和方法。这对于代码的复用是非常有用的。

大部分面向对象的编程语言，都是通过“类”（class）实现对象的继承。传统上，JavaScript 语言的继承不通过 class，而是通过“原型对象”（prototype）实现，本章介绍 JavaScript 的原型链继承。

ES6 引入了 class 语法，基于 class 的继承不在这个教程介绍，请参阅《ES6 标准入门》一书的相关章节。

原型对象概述

构造函数的缺点

JavaScript 通过构造函数生成新对象，因此构造函数可以视为对象的模板。实例对象的属性和方法，可以定义在构造函数内部。

```javascript
function Cat (name, color) {
  this.name = name;
  this.color = color;
}

var cat1 = new Cat('大毛', '白色');

cat1.name // '大毛'
cat1.color // '白色'
```

上面代码中，`Cat`函数是一个构造函数，函数内部定义了`name`属性和`color`属性，所有实例对象（上例是`cat1`）都会生成这两个属性，即这两个属性会定义在实例对象上面。

通过构造函数为实例对象定义属性，虽然很方便，但是有一个缺点。同一个构造函数的多个实例之间，无法共享属性，从而造成对系统资源的浪费。

```javascript
function Cat(name, color) {
  this.name = name;
  this.color = color;
  this.meow = function () {
    console.log('喵喵');
  };
}
```

```
var cat1 = new Cat('大毛', '白色');
var cat2 = new Cat('二毛', '黑色');

cat1.meow === cat2.meow
// false
```

上面代码中，`cat1`和`cat2`是同一个构造函数的两个实例，它们都具有`meow`方法。由于`meow`方法是生成在每个实例对象上面，所以两个实例就生成了两次。也就是说，每新建一个实例，就会新建一个`meow`方法。这既没有必要，又浪费系统资源，因为所有`meow`方法都是同样的行为，完全应该共享。

这个问题的解决方法，就是 JavaScript 的原型对象（prototype）。

prototype 属性的作用

JavaScript 继承机制的设计思想就是，原型对象的所有属性和方法，都能被实例对象共享。也就是说，如果属性和方法定义在原型上，那么所有实例对象就能共享，不仅节省了内存，还体现了实例对象之间的联系。

下面，先看怎么为对象指定原型。JavaScript 规定，每个函数都有一个`prototype`属性，指向一个对象。

```javascript
function f() {}
typeof f.prototype // "object"
```

上面代码中，函数`f`默认具有`prototype`属性，指向一个对象。

对于普通函数来说，该属性基本无用。但是，对于构造函数来说，生成实例的时候，该属性会自动成为实例对象的原型。

```javascript
function Animal(name) {
  this.name = name;
}
Animal.prototype.color = 'white';

var cat1 = new Animal('大毛');
var cat2 = new Animal('二毛');

cat1.color // 'white'
cat2.color // 'white'
```

上面代码中，构造函数`Animal`的`prototype`属性，就是实例对象`cat1`和`cat2`的原型对象。原型对象上添加一个`color`属性，结果，实例对象都共享了该属性。

原型对象的属性不是实例对象自身的属性。只要修改原型对象，变动就立刻会体现在**所有**实例对象上。

```javascript
Animal.prototype.color = 'yellow';

cat1.color // "yellow"
cat2.color // "yellow"
```

上面代码中，原型对象的`color`属性的值变为`yellow`，两个实例对象的`color`属性立刻跟着变了。这是因为实例对象其实没有`color`属性，都是读取原型对象的`color`属性。也就是说，当实例对象本身没有某个属性或方法的时候，它会到原型对象去寻找该属性或方法。这就是原型对象的特殊之处。

如果实例对象自身就有某个属性或方法，它就不会再去原型对象寻找这个属性或方法。

```javascript
cat1.color = 'black';

cat1.color // 'black'
cat2.color // 'yellow'
Animal.prototype.color // 'yellow';
```

上面代码中，实例对象`cat1`的`color`属性改为`black`，就使得它不再去原型对象读取`color`属性，后者的值依然为`yellow`。

总结一下，原型对象的作用，就是定义所有实例对象共享的属性和方法。这也是它被称为原型对象的原因，而实例对象可以视作从原型对象衍生出来的子对象。

```javascript
Animal.prototype.walk = function () {
  console.log(this.name + ' is walking');
};
```

上面代码中，`Animal.prototype`对象上面定义了一个`walk`方法，这个方法将可以在所有`Animal`实例对象上面调用。

原型链

JavaScript 规定，所有对象都有自己的原型对象（prototype）。一方面，任何一个对象，都可以充当其他对象的原型；另一方面，由于原型对象也是对象，所以它也有自己的原型。因此，就会形成一个“原型链”（prototype chain）：对象到原型，再到原型的原型……

如果一层层地上溯，所有对象的原型最终都可以上溯到`Object.prototype`，即`Object`构造函数的`prototype`属性。也就是说，所有对象都继承了`Object.prototype`的属性。这就是所有对象都有`valueOf`和`toString`方法的原因，因为这是从`Object.prototype`继承的。

那么，`Object.prototype`对象有没有它的原型呢？回答是`Object.prototype`的原型是`null`。`null`没有任何属性和方法，也没有自己的原型。因此，原型链的尽头就是`null`。

```javascript
Object.getPrototypeOf(Object.prototype)
// null
```

上面代码表示，`Object.prototype`对象的原型是`null`，由于`null`没有任何属性，所以原型链到此为止。`Object.getPrototypeOf`方法返回参数对象的原型，具体介绍请看后文。

读取对象的某个属性时，JavaScript 引擎先寻找对象本身的属性，如果找不到，就到它的原型去找，如果还是找不到，就到原型的原型去找。如果直到最顶层的`Object.prototype`还是找不到，则返回`undefined`。如果对象自身和它的原型，都定义了一个同名属性，那么优先读取对象自身的属性，这叫做“覆盖”（overriding）。

注意，一级级向上，在整个原型链上寻找某个属性，对性能是有影响的。所寻找的属性在越上层的原型对象，对性能的影响越大。如果寻找某个不存在的属性，将会遍历整个原型链。

举例来说，如果让构造函数的`prototype`属性指向一个数组，就意味着实例对象可以调用数组方法。

```javascript
var MyArray = function () {};

MyArray.prototype = new Array();
MyArray.prototype.constructor = MyArray;

var mine = new MyArray();
mine.push(1, 2, 3);
mine.length // 3
mine instanceof Array // true
```

上面代码中，`mine`是构造函数`MyArray`的实例对象，由于`MyArray.prototype`指向一个数组实例，使得`mine`可以调用数组方法（这些方法定义在数组实例的`prototype`对象上面）。最后那行`instanceof`表达式，用来比较一个对象是否为某个构造函数的实例，结果就是证明`mine`为`Array`的实例，`instanceof`运算符的详细解释详见后文。

上面代码还出现了原型对象的`constructor`属性，这个属性的含义下一节就来解释。

constructor 属性

`prototype`对象有一个`constructor`属性，默认指向`prototype`对象所在的构造函数。

```javascript
function P() {}
P.prototype.constructor === P // true
```

由于`constructor`属性定义在`prototype`对象上面，意味着可以被所有实例对象继承。

```javascript
function P() {}
var p = new P();

p.constructor === P // true
p.constructor === P.prototype.constructor // true
p.hasOwnProperty('constructor') // false
```

上面代码中，`p`是构造函数`P`的实例对象，但是`p`自身没有`constructor`属性，该属性其实是读取原型链上面的`P.prototype.constructor`属性。

`constructor`属性的作用是，可以得知某个实例对象，到底是哪一个构造函数产生的。

```javascript
function F() {};
var f = new F();

f.constructor === F // true
f.constructor === RegExp // false
```

上面代码中，`constructor`属性确定了实例对象`f`的构造函数是`F`，而不是`RegExp`。

另一方面，有了`constructor`属性，就可以从一个实例对象新建另一个实例。

```javascript
function Constr() {}
var x = new Constr();

var y = new x.constructor();
y instanceof Constr // true
```

上面代码中，`x`是构造函数`Constr`的实例，可以从`x.constructor`间接调用构造函数。这使得在实例方法中，调用自身的构造函数成为可能。

```javascript
Constr.prototype.createCopy = function () {
  return new this.constructor();
};
```

上面代码中，`createCopy`方法调用构造函数，新建另一个实例。

`constructor`属性表示原型对象与构造函数之间的关联关系，如果修改了原型对象，一般会同时修改`constructor`属性，防止引用的时候出错。

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

Person.prototype.constructor === Person // true

Person.prototype = {
  method: function () {}
};

Person.prototype.constructor === Person // false
Person.prototype.constructor === Object // true
```

上面代码中，构造函数`Person`的原型对象改掉了，但是没有修改`constructor`属性，导致这个属性不再指向`Person`。由于`Person`的新原型是一个普通对象，而普通对象的`constructor`属性指向`Object`构造函数，导致`Person.prototype.constructor`变成了`Object`。

所以，修改原型对象时，一般要同时修改`constructor`属性的指向。

```javascript
// 坏的写法
C.prototype = {
  method1: function (...) { ... },
  // ...
};

// 好的写法
C.prototype = {
  constructor: C,
  method1: function (...) { ... },
  // ...
};

// 更好的写法
C.prototype.method1 = function (...) { ... };
```

上面代码中，要么将`constructor`属性重新指向原来的构造函数，要么只在原型对象上添加方法，这样可以保证`instanceof`运算符不会失真。

如果不能确定`constructor`属性是什么函数，还有一个办法：通过`name`属性，从实例得到构造函数的名称。

```javascript
function Foo() {}
var f = new Foo();
f.constructor.name // "Foo"
```

instanceof 运算符

`instanceof`运算符返回一个布尔值，表示对象是否为某个构造函数的实例。

```javascript
var v = new Vehicle();
v instanceof Vehicle // true
```

上面代码中，对象`v`是构造函数`Vehicle`的实例，所以返回`true`。

`instanceof`运算符的左边是实例对象，右边是构造函数。它会检查右边构建函数的原型对象（prototype），是否在左边对象的原型链上。因此，下面两种写法是等价的。

```javascript
v instanceof Vehicle
// 等同于
Vehicle.prototype.isPrototypeOf(v)
```

上面代码中，`Object.prototype.isPrototypeOf`的详细解释见后文。

由于`instanceof`检查整个原型链，因此同一个实例对象，可能会对多个构造函数都返回`true`。

```javascript
var d = new Date();
d instanceof Date // true
d instanceof Object // true
```

上面代码中，`d`同时是`Date`和`Object`的实例，因此对这两个构造函数都返回`true`。

由于任意对象（除了`null`）都是`Object`的实例，所以`instanceof`运算符可以判断一个值是否为非`null`的对象。

```javascript
var obj = { foo: 123 };
obj instanceof Object // true

null instanceof Object // false
```

上面代码中，除了`null`，其他对象的`instanceOf Object`的运算结果都是`true`。

`instanceof`的原理是检查右边构造函数的`prototype`属性，是否在左边对象的原型链上。有一种特殊情况，就是左边对象的原型链上，只有`null`对象。这时，`instanceof`判断会失真。

```javascript
var obj = Object.create(null);
typeof obj // "object"
Object.create(null) instanceof Object // false
```

上面代码中，`Object.create(null)`返回一个新对象`obj`，它的原型是`null`（`Object.create`的详细介绍见后文）。右边的构造函数`Object`的`prototype`属性，不在左边的原型链上，因此`instanceof`就认为`obj`不是`Object`的实例。但是，只要一个对象的原型不是`null`，`instanceof`运算符的判断就不会失真。

`instanceof`运算符的一个用处，是判断值的类型。

```javascript
var x = [1, 2, 3];
var y = {};
x instanceof Array // true
y instanceof Object // true
```

上面代码中，`instanceof`运算符判断，变量`x`是数组，变量`y`是对象。

注意，`instanceof`运算符只能用于对象，不适用原始类型的值。

```javascript
var s = 'hello';
s instanceof String // false
```

上面代码中，字符串不是`String`对象的实例（因为字符串不是对象），所以返回`false`。

此外，对于`undefined`和`null`，`instanceof`运算符总是返回`false`。

```javascript
undefined instanceof Object // false
null instanceof Object // false
```

利用`instanceof`运算符，还可以巧妙地解决，调用构造函数时，忘了加`new`命令的问题。

```javascript
function Fubar (foo, bar) {
  if (this instanceof Fubar) {
    this._foo = foo;
    this._bar = bar;
  } else {
    return new Fubar(foo, bar);
  }
}
```

上面代码使用`instanceof`运算符，在函数体内部判断`this`关键字是否为构造函数`Fubar`的实例。如果不是，就表明忘了加`new`命令。

构造函数的继承

让一个构造函数继承另一个构造函数，是非常常见的需求。这可以分成两步实现。第一步是在子类的构造函数中，调用父类的构造函数。

```javascript
function Sub(value) {
  Super.call(this);
  this.prop = value;
}
```

上面代码中，`Sub`是子类的构造函数，`this`是子类的实例。在实例上调用父类的构造函数`Super`，就会让子类实例具有父类实例的属性。

第二步，是让子类的原型指向父类的原型，这样子类就可以继承父类原型。

```javascript
Sub.prototype = Object.create(Super.prototype);
Sub.prototype.constructor = Sub;
Sub.prototype.method = '...';
```

上面代码中，`Sub.prototype`是子类的原型，要将它赋值为`Object.create(Super.prototype)`，而不是直接等于`Super.prototype`。否则后面两行对`Sub.prototype`的操作，会连父类的原型`Super.prototype`一起修改掉。

另外一种写法是`Sub.prototype`等于一个父类实例。

```javascript
Sub.prototype = new Super();
```

上面这种写法也有继承的效果，但是子类会具有父类实例的方法。有时，这可能不是我们需要的，所以不推荐使用这种写法。

举例来说，下面是一个`Shape`构造函数。

```javascript
function Shape() {
  this.x = 0;
  this.y = 0;
}

Shape.prototype.move = function (x, y) {
  this.x += x;
  this.y += y;
  console.info('Shape moved.');
};
```

我们需要让`Rectangle`构造函数继承`Shape`。

```javascript
// 第一步，子类继承父类的实例
function Rectangle() {
  Shape.call(this); // 调用父类构造函数
}
// 另一种写法
function Rectangle() {
  this.base = Shape;
  this.base();
}

// 第二步，子类继承父类的原型
Rectangle.prototype = Object.create(Shape.prototype);
Rectangle.prototype.constructor = Rectangle;
```

采用这样的写法以后，`instanceof`运算符会对子类和父类的构造函数，都返回`true`。

```javascript
var rect = new Rectangle();

rect instanceof Rectangle  // true
rect instanceof Shape  // true
```

上面代码中，子类是整体继承父类。有时只需要单个方法的继承，这时可以采用下面的写法。

```javascript
ClassB.prototype.print = function() {
```

```
  ClassA.prototype.print.call(this);
  // some code
}
```

上面代码中，子类`B`的`print`方法先调用父类`A`的`print`方法，再部署自己的代码。这就等于继承了父类`A`的`print`方法。

多重继承

JavaScript 不提供多重继承功能，即不允许一个对象同时继承多个对象。但是，可以通过变通方法，实现这个功能。

```javascript
function M1() {
  this.hello = 'hello';
}

function M2() {
  this.world = 'world';
}

function S() {
  M1.call(this);
  M2.call(this);
}

// 继承 M1
S.prototype = Object.create(M1.prototype);
// 继承链上加入 M2
Object.assign(S.prototype, M2.prototype);

// 指定构造函数
S.prototype.constructor = S;

var s = new S();
s.hello // 'hello'
s.world // 'world'
```

上面代码中，子类`S`同时继承了父类`M1`和`M2`。这种模式又称为 Mixin（混入）。

模块

随着网站逐渐变成“互联网应用程序”，嵌入网页的 JavaScript 代码越来越庞大，越来越复杂。网页越来越像桌面程序，需要一个团队分工协作、进度管理、单元测试等等……开发者必须使用软件工程的方法，管理网页的业务逻辑。

JavaScript 模块化编程，已经成为一个迫切的需求。理想情况下，开发者只需要实现核心的业务逻辑，其他都可以加载别人已经写好的模块。

但是，JavaScript 不是一种模块化编程语言，ES6 才开始支持“类”和“模块”。下面介绍传统的做法，如何利用对象实现模块的效果。

基本的实现方法

模块是实现特定功能的一组属性和方法的封装。

简单的做法是把模块写成一个对象，所有的模块成员都放到这个对象里面。

```javascript
var module1 = new Object({
  _count : 0,
  m1 : function (){
     //...
  },
  m2 : function (){
    //...
  }
});
```

上面的函数`m1`和`m2`，都封装在`module1`对象里。使用的时候，就是调用这个对象的属性。

```javascript
module1.m1();
```

但是，这样的写法会暴露所有模块成员，内部状态可以被外部改写。比如，外部代码可以直接改变内部计数器的值。

```javascript
module1._count = 5;
```

封装私有变量：构造函数的写法

我们可以利用构造函数，封装私有变量。

```javascript
function StringBuilder() {
  var buffer = [];

  this.add = function (str) {
     buffer.push(str);
  };
```

```
  this.toString = function () {
    return buffer.join('');
  };

}
```

上面代码中，`buffer`是模块的私有变量。一旦生成实例对象，外部是无法直接访问`buffer`的。但是，这种方法将私有变量封装在构造函数中，导致构造函数与实例对象是一体的，总是存在于内存之中，无法在使用完成后清除。这意味着，构造函数有双重作用，既用来塑造实例对象，又用来保存实例对象的数据，违背了构造函数与实例对象在数据上相分离的原则（即实例对象的数据，不应该保存在实例对象以外）。同时，非常耗费内存。

```javascript
function StringBuilder() {
  this._buffer = [];
}

StringBuilder.prototype = {
  constructor: StringBuilder,
  add: function (str) {
    this._buffer.push(str);
  },
  toString: function () {
    return this._buffer.join('');
  }
};
```

这种方法将私有变量放入实例对象中，好处是看上去更自然，但是它的私有变量可以从外部读写，不是很安全。

封装私有变量：立即执行函数的写法

另一种做法是使用“立即执行函数”（Immediately-Invoked Function Expression，IIFE），将相关的属性和方法封装在一个函数作用域里面，可以达到不暴露私有成员的目的。

```javascript
var module1 = (function () {
    var _count = 0;
    var m1 = function () {
      //...
    };
    var m2 = function () {
        //...
    };
    return {
        m1 : m1,
        m2 : m2
    };
```

})();
```

使用上面的写法，外部代码无法读取内部的`_count`变量。

```javascript
console.info(module1._count); //undefined
```

上面的`module1`就是 JavaScript 模块的基本写法。下面，再对这种写法进行加工。

模块的放大模式

如果一个模块很大，必须分成几个部分，或者一个模块需要继承另一个模块，这时就有必要采用“放大模式”（augmentation）。

```javascript
var module1 = (function (mod){
  mod.m3 = function () {
    //...
  };
  return mod;
})(module1);
```

上面的代码为`module1`模块添加了一个新方法`m3()`，然后返回新的`module1`模块。

在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用上面的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用"宽放大模式"（Loose augmentation）。

```javascript
var module1 = (function (mod) {
  //...
  return mod;
})(window.module1 || {});
```

与"放大模式"相比，“宽放大模式”就是“立即执行函数”的参数可以是空对象。

输入全局变量

独立性是模块的重要特点，模块内部最好不与程序的其他部分直接交互。

为了在模块内部调用全局变量，必须显式地将其他变量输入模块。

```javascript
var module1 = (function ($, YAHOO) {
  //...
```
```

```
})(jQuery, YAHOO);
```

上面的`module1`模块需要使用 jQuery 库和 YUI 库，就把这两个库（其实是两个模块）当作参数输入`module1`。这样做除了保证模块的独立性，还使得模块之间的依赖关系变得明显。

立即执行函数还可以起到命名空间的作用。

```javascript
(function($, window, document) {

  function go(num) {
  }

  function handleEvents() {
  }

  function initialize() {
  }

  function dieCarouselDie() {
  }

  //attach to the global scope
  window.finalCarousel = {
    init : initialize,
    destroy : dieCarouselDie
  }

})( jQuery, window, document );
```

上面代码中，`finalCarousel`对象输出到全局，对外暴露`init`和`destroy`接口，内部方法`go`、`handleEvents`、`initialize`、`dieCarouselDie`都是外部无法调用的。

参考链接

- [JavaScript Modules: A Beginner's Guide](https://medium.freecodecamp.com/javascript-modules-a-beginner-s-guide-783f7d7a5fcc), by Preethi Kasireddy