对象的继承

1. 原型对象概述

1.1. 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。

```
function Cat (name, color) {
  this.name = name;
  this.color = color;
}
let cat1 = new Cat('大毛', '白色');
cat1.name;  // '大毛'
cat1.color; // '白色'
```

上例中，`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。**

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

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

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

上例中，`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）。

1.2. 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。**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。**

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

上例中，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。

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

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

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

上例中，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。

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

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

上例中，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。**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。

```
cat1.color = 'black';

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

上例中，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`。

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。

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

上例中，`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。

1.3. 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。`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。**原型链的尽头就是 `null`。**

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

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）。

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。

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。

```
let MyArray = function () {};

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

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

上例中，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。

1.4. 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。

```
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。

```
function P() {}
let p = new P();

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

上例中，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。

```
function F() {};
let f = new F();

f.constructor === F;      // true f 的构造函数是 F
f.constructor === RegExp; // false
```

上例中，constructor 属性确定了实例对象 f 的构造函数是 F，而不是 RegExp。

有了constructor属性，就可以从一个实例对象新建另一个实例。

```
function Constr() {}
let x = new Constr();

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

上例中，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。

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

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。

```
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
```

上例中，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。

```
// bad
C.prototype = {
  method1: function (...) { ... },
  // ...
};

// good
C.prototype = {
  constructor: C,
  method1: function (...) { ... },
  // ...
};

// best
C.prototype.method1 = function (...) { ... };
```

上例中，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。

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

2. 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。

```
let v = new Vehicle();
v instanceof Vehicle; // true
```

instanceof 运算符的左边是实例对象，右边是构造函数。它会检查右边构造函数的原型对象（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。

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

上例中，Vehicle 是对象 v 的构造函数，它的原型对象是 Vehicle.prototype，isPrototypeOf() 方法是 JavaScript 提供的原生方法，用于检查某个对象是否为另一个对象的原型。

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。

```
let d = new Date();
d instanceof Date;   // true
d instanceof Object; // 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。

```
let obj = { foo: 123 };
obj instanceof Object;  // true
null instanceof Object; // false
```

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。

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

上例中，`Object.create(null)` 返回一个新对象 `obj`，它的原型是 `null` 。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。**这是唯一的 `instanceof` 运算符判断会失真的情况（一个对象的原型是null）。**

`instanceof` 运算符的一个用处，是判断值的类型。

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

上例中，`instanceof` 运算符判断，变量 `x` 是数组，变量 `y` 是对象。

`instanceof` 运算符只能用于对象，不适用原始类型的值。

```
let s = 'hello';
s instanceof String; // false
```

上例中，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`。

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

```
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。

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

上例使用 `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。

3. 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。

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

上例中，`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。

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

上例中，`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。

一个 `Shape` 构造函数：

```
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`。

```
// 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`。

```
let rect = new Rectangle();

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