链滴

JavaScript 面向对象编程

作者：limanting
原文链接：https://ld246.com/article/1629877667089
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

JavaScript不区分类和实例的概念，而是通过**原型prototype**来实现面向对象编程。

JavaScript的原型链和Java的Class区别就在，它没有“Class”的概念，所有对象都是实例，所谓继关系不过是把一个对象的原型指向另一个对象而已。

```
var Student = {
    name: 'Robot',
    height: 1.2,
    run: function () {
        console.log(this.name + ' is running...');
    }
};

var xiaoming = {
    name: '小明'
};

xiaoming.__proto__ = Student; // 把xiaoming的原型指向了对象Student，看上去xiaoming仿佛从Student继承下来的
```

Object.create()方法可以传入一个原型对象，并创建一个基于该原型的新对象，但是新对象什么属性没有。

创建对象

JavaScript对每个创建的对象都会设置一个原型，指向它的原型对象。

当我们用 obj.xxx访问一个对象的属性时，JavaScript引擎先在当前对象上查找该属性，如果没有找到就到其原型对象上找，如果还没有找到，就一直上溯到 Object.prototype对象，最后，如果还没有到，就只能返回 undefined。

```
var arr = [1, 2, 3];
arr ----> Array.prototype ----> Object.prototype ----> null

function foo() {
    return 0;
}
foo ----> Function.prototype ----> Object.prototype ----> null
```

如果原型链很长，那么访问一个对象的属性就会因为花更多的时间查找而变得更慢，因此要注意不要原型链搞得太长。

构造函数

除了直接用 {}创建一个对象外，JavaScript还可以用一种构造函数的方法来创建对象。它的用法是，定义一个构造函数,用关键字 new来调用这个函数，并返回一个对象。如果不写 new，这就是一个普函数，它返回 undefined。但是，如果写了 new，它就变成了一个构造函数，它绑定的 this指向新建的对象，并默认返回 this，也就是说，不需要在最后写 return this

```
var xiaoming = new Student('小明');
xiaoming.name; // '小明'
xiaoming.hello(); // Hello, 小明!
```

原文链接：JavaScript 面向对象编程

```
xiaoming ----> Student.prototype ----> Object.prototype ----> null
```

xiaoming的原型指向函数Student的原型。如果你又创建了xiaohong、xiaojun，那么这些对象的原与xiaoming是一样的

用 new Student()创建的对象还从原型上获得了一个 constructor属性，它指向函数 Student本身。

```
xiaoming.constructor === Student.prototype.constructor; // true
Student.prototype.constructor === Student; // true

Object.getPrototypeOf(xiaoming) === Student.prototype; // true

xiaoming instanceof Student; // true
```

xiaoming、xiaohong这些对象可没有 prototype这个属性，不过可以用 __proto__这个非标准用法查看。

如果我们通过 new Student()创建了很多对象，这些对象的 hello函数实际上只需要共享同一个函数可以了，这样可以节省很多内存。

要让创建的对象共享一个 hello函数，根据对象的属性查找原则，我们只要把 hello函数移动到 xiaomig、xiaohong这些对象共同的原型上就可以了，也就是 Student.prototype。

```
function Student(name) {
    this.name = name;
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
};
```

忘记写new

如果一个函数被定义为用于创建对象的构造函数，但是调用时忘记了写 new怎么办?

在strict模式下，this.name = name将报错，因为 this绑定为 undefined

在非strict模式下，this.name = name不报错，因为 this绑定为 window，于是无意间创建了全局变量ame，并且返回 undefined，这个结果更糟糕。

构造函数首字母应当大写，而普通函数首字母应当小写，这样，一些语法检查工具如jslint将可以帮你测到漏写的 new

在内部封装所有的 new操作。一个常用的编程模式像这样：

```
function Student(props) {
    this.name = props.name || '匿名'; // 默认值为'匿名'
    this.grade = props.grade || 1; // 默认值为1
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
};

function createStudent(props) {
```

```
    return new Student(props || {})
}
```

这个 createStudent()函数有几个巨大的优点：一是不需要 new来调用，二是参数非常灵活，可以不
，也可以这么传：

```
var xiaoming = createStudent({
    name: '小明'
});

xiaoming.grade; // 1
```

原型继承

在传统的基于Class的语言如Java、C++中，继承的本质是扩展一个已有的Class，并生成新的Subclas
。

但是，JavaScript由于采用原型继承，我们无法直接扩展一个Class，因为根本不存在Class这种类型。

基于 Student扩展出 PrimaryStudent，可以先定义出 PrimaryStudent：

```
function PrimaryStudent(props) {
    // 调用Student构造函数，绑定this变量:
    Student.call(this, props);
    this.grade = props.grade || 1;
}
```

调用了 Student构造函数不等于继承了 Student，PrimaryStudent创建的对象的原型是：

```
new PrimaryStudent() ----> PrimaryStudent.prototype ----> Object.prototype ----> null
```

借助一个中间对象来实现正确的原型链，这个中间对象的原型要指向 Student.prototype 。 中间对
可以用一个空函数 F来实现。

```
// PrimaryStudent构造函数:
function PrimaryStudent(props) {
    Student.call(this, props);
    this.grade = props.grade || 1;
}

// 空函数F:
function F() {
}

// 把F的原型指向Student.prototype:
F.prototype = Student.prototype;

// 把PrimaryStudent的原型指向一个新的F对象，F对象的原型正好指向Student.prototype:
PrimaryStudent.prototype = new F();

// 把PrimaryStudent原型的构造函数修复为PrimaryStudent:
PrimaryStudent.prototype.constructor = PrimaryStudent;

// 继续在PrimaryStudent原型（就是new F()对象）上定义方法：
```

```
PrimaryStudent.prototype.getGrade = function () {
    return this.grade;
};

// 创建xiaoming:
var xiaoming = new PrimaryStudent({
    name: '小明',
    grade: 2
});
xiaoming.name; // '小明'
xiaoming.grade; // 2

// 验证原型:
xiaoming.__proto__ === PrimaryStudent.prototype; // true
xiaoming.__proto__.__proto__ === Student.prototype; // true

// 验证继承关系:
xiaoming instanceof PrimaryStudent; // true
xiaoming instanceof Student; // true
```

函数 F仅用于桥接，我们仅创建了一个 new F()实例，而且，没有改变原有的 Student定义的原型链.

instanceof 运算符 用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上。

JavaScript的原型继承实现方式就是：

1. 定义新的构造函数，并在内部用 call()调用希望“继承”的构造函数，并绑定this；
2. 借助中间函数 F实现原型链继承，最好通过封装的inherits函数完成；
3. 继续在新的构造函数的原型上定义新方法。

class继承

基于原型继承的特点是简单，缺点是理解起来比传统的类 - 实例模型要困难，最大的缺点是继承的实需要编写大量代码，并且需要正确实现原型链。

class的目的就是让定义类更简单。

```
class Student {
    constructor(name) {
        this.name = name;
    }

    hello() {
        alert('Hello, ' + this.name + '!');
    }
}

var xiaoming = new Student('小明');
```

比较一下就可以发现，class的定义包含了构造函数 constructor和定义在原型对象上的函数 hello()（意没有 function关键字），这样就避免了 Student.prototype.hello = function () {...}这样分散的代。

原文链接：JavaScript 面向对象编程

用 class定义对象的另一个巨大的好处是继承更方便了，原型继承的中间对象，原型对象的构造函数等都不需要考虑了，直接通过 extends来实现

```
class PrimaryStudent extends Student {
    constructor(name, grade) {
        super(name); // 记得用super调用父类的构造方法!
        this.grade = grade;
    }

    myGrade() {
        alert('I am at grade ' + this.grade);
    }
}
```

注意 PrimaryStudent的定义也是class关键字实现的，而 extends则表示原型链对象来自 Student。类的构造函数可能会与父类不太相同，例如，PrimaryStudent需要 name和 grade两个参数，并且要通过 super(name)来调用父类的构造函数，否则父类的 name属性无法正常初始化。

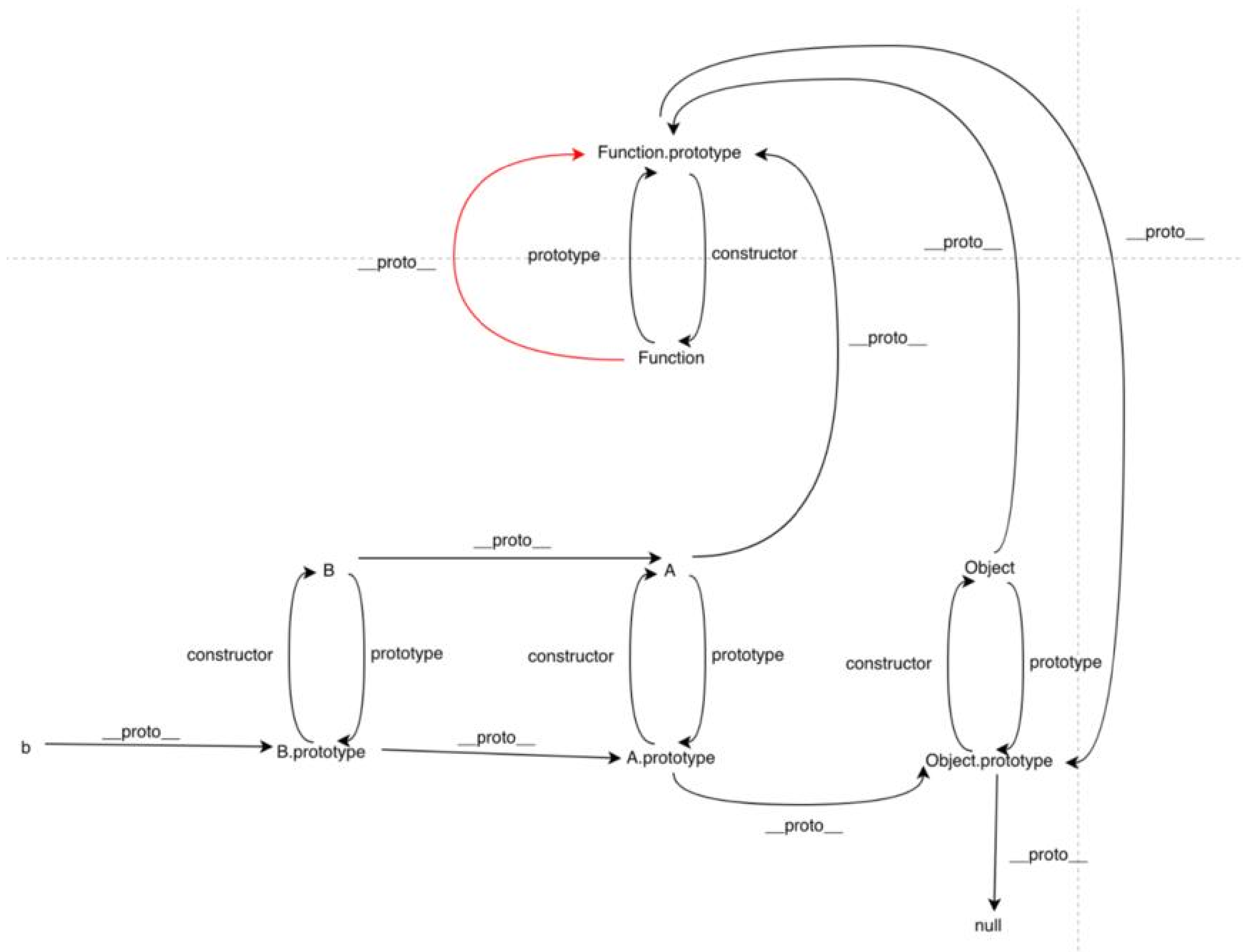
ES6引入的 class和原有的JavaScript原型继承有什么区别呢？实际上它们没有任何区别，class的作用是让JavaScript引擎去实现原来需要我们自己编写的原型链代码。简而言之，用 class的好处就是极大简化了原型链代码。

Tip 画原型图理解原型关系

参考https://yanhaijing.com/javascript/2021/03/13/javascript-prototype-chain/

```
class A {}
class B extends A {}

const b = new B();
```



下面的写法和上面写法的有什么区别？该如何弥补？

```
function A() {}
function B() {}

B.prototype = Object.create(A.prototype);

const b = new B();

// 下面两行语句的结果是，为什么
Function instanceof Object
Object instanceof Function
```

原文链接：JavaScript 面向对象编程