类

ECMAScript 2015 中引入的 JavaScript 类主要是 JavaScript 现有的基于原型的继承的语法糖。 类语法不是向JavaScript引入一个新的面向对象的继承模型。JJavaScript类提供了一个更简单和更清晰的语法来创建对象并处理继承。

定义类

类实际上是个"特殊的函数"，而正如可以定义函数表达式和函数声明一样，类语法有两个组件：类表达式和类声明。

类声明

定义一个类的一种方法是使用一个**类声明**。要声明一个类，你可以使用带有class关键字的类名（这里是"Rectangle"）。

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

提升

函数声明和类声明之间的一个重要区别是函数声明是hoisted，类声明不会。你首先需要声明你的类，然后访问它，否则像下面的代码会抛出一个ReferenceError：

```
let p = new Rectangle();

// ReferenceError

class Rectangle {}
```

类表达式

一个**类表达式**是定义一个类的另一种方式。类表达式可以是被命名的或匿名的。赋予一个命名类表达式的名称是类的主体的本地名称。

```
/* 匿名类 */
let Rectangle = class {
  constructor(height, width) {
```

```
    this.height = height;
    this.width = width;
  }
};

/* 命名的类 */
let Rectangle = class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

注意: 类**表达式**也受到类**声明**中提到的同样提升问题的困扰。

类体和方法定义

一个类的类体是一对花括号/大括号 {} 中的部分。这是你定义类成员的位置，如方法或构造函数。

严格模式

类声明和类表达式的主体都执行在严格模式下。

构造函数

构造函数方法是一个特殊的方法，其用于创建和初始化使用一个类创建的一个对象。一个类只能拥有一个名为 “constructor”的特殊方法。如果类包含多个构造函数的方法，则将抛出 一个`SyntaxError` 。

一个构造函数可以使用 **super** 关键字来调用一个父类的构造函数。

原型方法

参见方法定义。

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  get area() {
    return this.calcArea()
  }
  calcArea() {
    return this.height * this.width;
  }
}
const square = new Rectangle(10, 10);
```

```
15 console.log(square.area);
16 // 100
```

静态方法

`static` 关键字用来定义一个类的一个静态方法。调用静态方法而不实例化其类，不能通过一个类实例调用静态方法。静态方法通常用于为一个应用程序创建工具函数。

```
1  class Point {
2      constructor(x, y) {
3          this.x = x;
4          this.y = y;
5      }
6
7      static distance(a, b) {
8          const dx = a.x - b.x;
9          const dy = a.y - b.y;
10
11         return Math.sqrt(dx*dx + dy*dy);
12     }
13 }
14
15 const p1 = new Point(5, 5);
16 const p2 = new Point(10, 10);
17
18 console.log(Point.distance(p1, p2));
```

用原型和静态方法装箱

当调用静态或原型方法而没有值为“this”的对象时，(或“this”作为布尔，字符串，数字，未定义或null)，，那么“this”值将在被调用的函数内部**未定义**。 不会发生自动装箱。即使我们以非严格模式编写代码，行为也是一样的。

```
1  class Animal {
2    speak() {
3      return this;
4    }
5    static eat() {
6      return this;
7    }
8  }
9
10 let obj = new Animal();
11 let speak = obj.speak;
12 speak(); // undefined
13
14 let eat = Animal.eat;
15 eat(); // undefined
```

```
eat(); // undefined
```

如果我们使用传统的基于函数的类来编写上述代码，那么基于调用该函数的“this”值将发生自动装箱。

```
function Animal() { }

Animal.prototype.speak = function() {
  return this;
}

Animal.eat = function() {
  return this;
}

let obj = new Animal();
let speak = obj.speak;
speak(); // global object

let eat = Animal.eat;
eat(); // global object
```

使用 extends 创建子类

`extends` 关键字在类声明或类表达式中用于创建一个类作为另一个类的一个子类。

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

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}


var d = new Dog('Mitzie');
// 'Mitzie barks.'
d.speak();
```

如果子类中存在构造函数，则需要在使用“this”之前首先调用super（ ）。

也可以扩展传统的基于函数的"类"：

```
function Animal (name) {
  this.name = name;
}
Animal.prototype.speak = function () {
  console.log(this.name + ' makes a noise.');
}

class Dog extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
}

var d = new Dog('Mitzie');
d.speak();
```

请注意，类不能扩展常规（不可构造/非构造的）对象。如果要继承常规对象，可以改用 `Object.setPrototypeOf()`:

```
var Animal = {
  speak() {
    console.log(this.name + ' makes a noise.');
  }
};

class Dog {
  constructor(name) {
    this.name = name;
  }
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
}
Object.setPrototypeOf(Dog.prototype, Animal);


var d = new Dog('Mitzie');
d.speak();
```

Species

你可能希望在派生数组类 *MyArray* 中返回 `Array`对象。这种**类/种类**模式允许你覆盖默认的构造函数。

例如，当使用像`map()`返回默认构造函数的方法时，您希望这些方法返回一个父Array对象，而不是MyArray对象。`Symbol.species` 符号可以让你这样做：

```
class MyArray extends Array {
  // Overwrite species to the parent Array constructor
  static get [Symbol.species]() { return Array; }
}
var a = new MyArray(1,2,3);
var mapped = a.map(x => x * x);

console.log(mapped instanceof MyArray);
// false
console.log(mapped instanceof Array);
// true
```

使用 super 调用超类

`super` 关键字用于调用对象的父对象上的函数。

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

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Lion extends Cat {
  speak() {
    super.speak();
    console.log(this.name + ' roars.');
  }
}
```

Mix-ins 混合

抽象子类或者 *mix-ins* 是类的模板。 一个 ECMAScript 类只能有一个单超类，所以想要从工具类来多重继承的行为是不可能的。子类继承的只能是父类提供的功能性。因此，例如，从工具类的多重继承是不可能的。该功能必须由超类提供。

一个以超类作为输入的函数和一个继承该超类的子类作为输出可以用于在ECMAScript中实现混合：

```
var calculatorMixin = Base => class extends Base {
  calc() { }
};

var randomizerMixin = Base => class extends Base {
  randomize() { }
};
```

使用 mix-ins 的类可以像下面这样写：

```
class Foo { }
class Bar extends calculatorMixin(randomizerMixin(Foo)) { }
```

规范

Specification	Status	Comment
ECMAScript 2015 (6th Edition, ECMA-262) Class definitions	ST Standard	Initial definition.
ECMAScript Latest Draft (ECMA-262) Class definitions	D Draft	

浏览器兼容性

Desktop Mobile

Feature	Chrome	Firefox (Gecko)	Edge	Internet Explorer	Opera	Safari
Basic support	42.0[1] 49.0	45	13	未实现	未实现	9.0

[1] 要求使用严格模式。非严格模式需要勾选启用实验性的 JavaScript（Enable Experimental JavaScript），其默认不勾选。

相关链接

- 函数
- 类声明
- 类表达式
- `super`
- Blog post: "ES6 In Depth: Classes"
- https://en.wikipedia.org/wiki/Mixin

这篇文章有帮助吗？