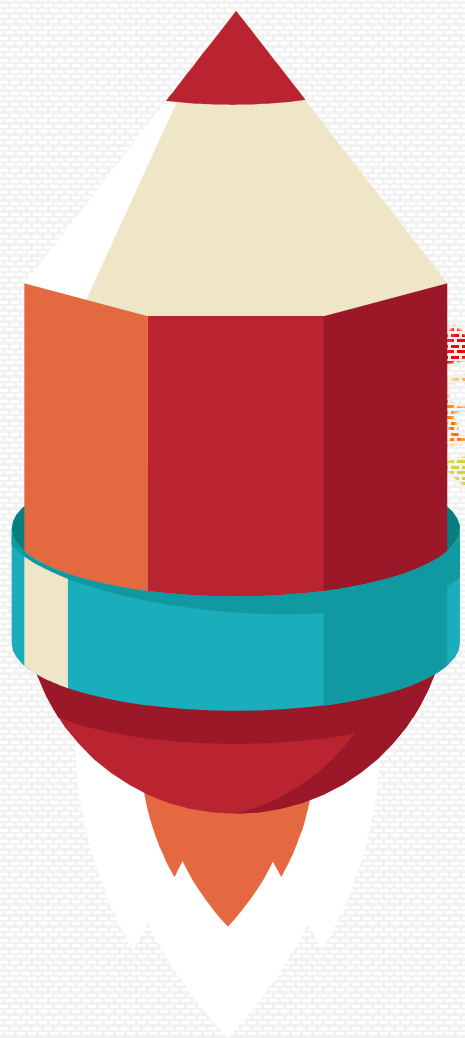




第34课：Class语法详解

主讲老师：万章



四知

Class的基本语法

Class的静态属性和方法

构造函数的新属性



Class的基本语法

Class的基本语法

JavaScript 语言中，生成实例对象的传统方法是通过构造函数和原型的组合模式。ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
Point.prototype.toString = function () {  
  return '(' + this.x + ', ' + this.y + ')';  
};  
  
var p = new Point(1, 2);
```

组合模式创造对象的方法

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  toString() {  
    return '(' + this.x + ', ' + this.y + ')';  
  }  
}
```

右边的代码用 ES6 的class改写，就是这样。

ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

“语法糖”：是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用

Class的基本语法

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
Point.prototype.toString = function () {  
  return '(' + this.x + ', ' + this.y + ')';  
};  
  
var p = new Point(1, 2);
```

ES5写法

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  toString() {  
    return '(' + this.x + ', ' + this.y + ')';  
  }  
}
```

ES6写法

ES6 的class与ES5写法的几个核心注意点:

- ES5 的构造函数Point, 对应 ES6 的Point类的构造方法。
- 类的所有方法都定义在类的prototype属性上面。
- 定义“类”的方法的时候, 前面不需要加上function这个关键字, 直接把函数定义放进去了就可以了
- 方法之间不需要逗号分隔, 加了会报错
- ES6的class使用方法与ES5的构造函数一模一样

Class的基本语法

```
class Point {  
  // ...  
}  
  
typeof Point // "function"  
Point === Point.prototype.constructor // true
```

- ES5 的构造函数Point, 对应 ES6 的Point类的构造方法。

上面代码表明, **类的数据类型就是函数, 类本身就指向构造函数。**

```
class Point {  
  constructor() {  
    // ...  
  }  
  
  toString() {  
    // ...  
  }  
  
  toValue() {  
    // ...  
  }  
}
```

等价于

```
Point.prototype = {  
  constructor() {},  
  toString() {},  
  toValue() {},  
};
```

- 类的所有方法都定义在类的prototype属性上面。

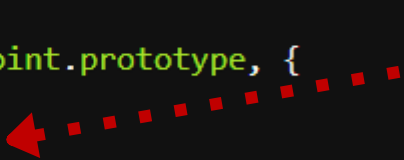
```
class B {}  
let b = new B();  
  
b.constructor === B.prototype.constructor // true
```

在类的实例上面调用方法, 其实就是调用原型上的方法。

Class的基本语法

由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。

```
class Point {  
  constructor(){  
    // ...  
  }  
}  
  
Object.assign(Point.prototype, {  
  toString(){},  
  toValue(){}  
});
```



Class直接定义的方法之间不需要逗号分隔，加了会报错。但是这里是Object.assign的方法格式，这里面需要往Point.prototype里面添加的方法就需要符合对象的默认格式

Class的基本语法

```
class Point {  
  constructor(x, y) {  
    // ...  
  }  
  
  toString() {  
    // ...  
  }  
}  
  
Object.keys(Point.prototype)  
// []  
Object.getOwnPropertyNames(Point.prototype)  
// ["constructor", "toString"]
```

类的内部所有定义的方法，都是不可枚举的
(non-enumerable)。

```
> class Point {  
  }  
  
Object.assign(Point.prototype, {  
  constructor(x, y) {  
    // ...  
  },  
  
  toString() {  
    // ...  
  }  
})  
< ▾ {toString: f, constructor: f} ⓘ  
  ▶ toString: f toString()  
  ▶ constructor: f constructor(x, y)  
  ▶ __proto__: Object  
  
> Object.keys(Point.prototype)  
< ▶ ["toString"]  
  
> Object.getOwnPropertyNames(Point.prototype)  
< ▶ (2) ["constructor", "toString"]  
  
> |
```

通过Object.assign方法往类的原型上添加的方法，
constructor不可枚举，其他的可以枚举

Class的基本语法之constructor

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

```
class Point {  
}  
  
// 等同于  
class Point {  
  constructor() {}  
}
```

constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象（得是在创建class时就定义设置的，在创建完class后，通过Object.assign的方式是没法改变构造函数的返回值的）

```
> class Point {  
  }  
< undefined  
> let o1= new Point()  
< undefined  
> o1 instanceof Point  
< true  
> Object.assign(Point.prototype, {  
  constructor() {  
    return Object.create(null);  
  }  
< ▶ {constructor: f}  
> let o2= new Point()  
< undefined  
> o2 instanceof Point  
< true  
> class Foo {  
  constructor() {  
    return Object.create(null);  
  }  
}  
  
new Foo() instanceof Foo  
// false  
< false  
> |
```

```
> class Point {  
  }  
< undefined  
> Point.constructor  
< f Function() { [native code] }  
> Object.assign(Point.prototype, {  
  constructor(x,y) {  
    this.x=x;  
    this.y=y;  
    return Object.create(null);  
  }  
< ▶ {constructor: f} ⓘ  
  ▶ constructor: f constructor(x,y)  
  ▶ __proto__: Object  
> Point.constructor  
< f Function() { [native code] }  
> let o1= new Point(10,10)  
< undefined  
> o1.x  
< undefined  
> o1.y  
< undefined  
> o1  
< ▼ Point {} ⓘ  
  ▼ __proto__:  
    ▶ constructor: f constructor(x,y)  
    ▶ __proto__: Object  
>
```

Class的基本语法之类的调用方式

类必须使用new调用，否则会报错。这是它跟普通构造函数（普通构造函数完全可以当做普通函数使用）的一个主要区别，后者不用new也可以执行。

```
class Foo {  
  constructor() {  
    return Object.create(null);  
  }  
}  
  
Foo()  
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

Class的基本语法之getter和setter

与 ES5 一样，在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

```
class MyClass {  
  constructor() {  
    // ...  
  }  
  get prop() {  
    return 'getter';  
  }  
  set prop(value) {  
    console.log('setter: ' + value);  
  }  
}  
  
let inst = new MyClass();  
  
inst.prop = 123;  
// setter: 123  
  
inst.prop  
// 'getter'
```

等价于



```
class MyClass {  
  constructor() {  
    // ...  
  }  
}  
  
Object.defineProperty(MyClass.prototype, "prop", {  
  enumerable: false,  
  configurable: true,  
  get: function() {  
    return 'getter';  
  },  
  set: function(value) {  
    console.log('setter: ' + value);  
  }  
})  
  
let inst = new MyClass();  
inst.prop = 123;  
// setter: 123  
inst.prop  
// 'getter'
```

Class的基本语法之类的属性名

类的属性名，可以采用表达式

```
let methodName = 'getArea';

class Square {
  constructor(length) {
    // ...
  }

  [methodName]() {
    // ...
  }
}
```

上面代码中，Square类的方法名getArea，是从表达式得到的。

Class的基本语法之class表达式

与函数一样，类也可以使用表达式的形式定义

```
let a=function fn(){  
  //  
}
```

```
const MyClass = class Me {  
  getClassName() {  
    return Me.name;  
  }  
};
```

```
let inst = new MyClass();  
inst.getClassName() // Me  
Me.name // ReferenceError: Me is not defined
```

上面代码使用表达式定义了一个类。需要注意的是，这个类的名字是Me，但是Me只在 Class 的内部可用，指代当前类。在 Class 外部，这个类只能用MyClass引用。

如果类的内部没用到的话，可以省略Me，也就是可以写成右侧这种形式。

```
const MyClass = class { /* ... */ };
```

Class的基本语法的特别注意点

(1) 严格模式

类和模块的内部，默认就是严格模式，所以不需要使用`use strict`指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。考虑到未来所有的代码，其实都是运行在模块之中，所以 ES6 实际上把整个语言升级到了严格模式。

(2) 不存在提升

```
new Foo(); // ReferenceError
class Foo {}
```

上面代码中，`Foo`类使用在前，定义在后，这样会报错，因为 ES6 不会把类的声明提升到代码头部。

(3) name 属性

```
class Point {}
Point.name // "Point"
```

由于本质上，ES6 的类只是 ES5 的构造函数的一层包装，所以函数的许多特性都被`Class`继承，包括`name`属性

Class的基本语法的特别注意点

(4) this 的指向

类的方法内部如果含有this，它默认指向类的实例。但是，必须非常小心，一旦单独使用该方法，很可能报错。

```
> class Logger {  
  printName(name = 'there') {  
    this.print(`Hello ${name}`);  
  }  
  
  print(text) {  
    console.log(text);  
  }  
}
```

```
<> undefined
```

```
> let o=new Logger()
```

```
<> undefined
```

```
> o.printName()
```

```
Hello there VM1586:7
```

```
<> undefined
```

```
> let { printName } = o; //解构赋值
```

```
<> undefined
```

```
> printName()
```

```
✖ ▶ Uncaught TypeError: Cannot read property 'print' of undefined VM1586:3  
    at printName (<anonymous>:3:10)  
    at <anonymous>:1:1
```

printName方法中的this，默认指向Logger类的实例。但是，如果将这个方法提取出来单独使用，this会指向该方法运行时所在的环境（由于 class 内部是严格模式，所以this实际指向的是undefined），从而导致找不到print方法而报错。

Class的基本语法的特别注意点

```
class Logger {  
  constructor() {  
    this.printName = this.printName.bind(this);  
  }  
  
  // ...  
}
```

一个比较简单的解决方法是，在构造方法中绑定this，这样就不会找不到print方法了。

bind之后返回的函数里面的this就永久锁死了

问题:这种写法有些反人类，所以弃用!!! !!!


```
class Obj {  
  constructor() {  
    this.getThis = () => this;  
  }  
}  
  
const myObj = new Obj();  
myObj.getThis() === myObj // true
```

另一种解决方法是使用箭头函数。

箭头函数位于构造函数内部，它的定义生效的时候，是在构造函数执行的时候。这时，箭头函数所在的运行环境，肯定是实例对象，所以this会总是指向实例对象。


Class的实例属性定义规范

```
class IncreasingCounter {  
  constructor() {  
    this._count = 0;  
  }  
  get value() {  
    console.log('Getting the current value!');  
    return this._count;  
  }  
  increment() {  
    this._count++;  
  }  
}
```



上面代码中，实例属性`this._count`定义在`constructor()`方法里面

```
class IncreasingCounter {  
  _count = 0;  
  get value() {  
    console.log('Getting the current value!');  
    return this._count;  
  }  
  increment() {  
    this._count++;  
  }  
}
```



另一种写法是，这个属性也可以定义在类的最顶层，其他都不变。这种新写法的好处是，所有实例对象自身的属性都定义在类的头部，看上去比较整齐，一眼就能看出这个类有哪些实例属性。



Class的静态属性和方法

Class的静态方法

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
```

```
class Foo {
  static bar() {
    this.baz();
  }
  static baz() {
    console.log('hello');
  }
  baz() {
    console.log('world');
  }
}

Foo.bar() // hello
```

注意，如果静态方法包含this关键字，这个this指的是类，而不是实例。静态方法可以与非静态方法重名。

Class的静态方法

```
class Foo {  
    static classMethod() {  
        return 'hello';  
    }  
}  
  
class Bar extends Foo {  
}  
  
Bar.classMethod() // 'hello'
```

父类的静态方法，可以被子类继承

```
class Foo {  
    static classMethod() {  
        return 'hello';  
    }  
}  
  
class Bar extends Foo {  
    static classMethod() {  
        return super.classMethod() + ', too';  
    }  
}  
  
Bar.classMethod() // "hello, too"
```

静态方法也是可以从super对象上调用的。

Class的静态属性

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。

```
class Foo {  
}  
  
Foo.prop = 1;  
Foo.prop // 1
```



目前，只有这种写法可行，因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性

```
> class MyClass {  
  static myStaticProp = 42;  
  
  constructor() {  
    console.log(MyClass.myStaticProp); // 42  
  }  
}  
< undefined  
> MyClass.myStaticProp  
< 42  
> |
```

现在有一个提案提供了类的静态属性，写法是在实例属性法的前面，加上static关键字。（最新版谷歌浏览器已兼容）

Class的私有方法和私有属性

私有方法和私有属性:是只能在类的内部访问的方法和属性，外部不能访问。

这是常见需求，有利于代码的封装，但 ES6 不提供，只能通过变通方法模拟实现。

```
class Widget {  
  
  // 公有方法  
  foo (baz) {  
    this._bar(baz);  
  }  
  
  // 私有方法  
  _bar(baz) {  
    return this.snaf = baz;  
  }  
  
  // ...  
}
```

一种做法是在命名上加以区别

_bar方法前面的下划线，表示这是一个只限于内部使用的私有方法。但是，这种命名是不保险的，在类的外部，还是可以调用到这个方法

```
const bar = Symbol('bar');  
const snaf = Symbol('snaf');  
  
export default class myClass{  
  
  // 公有方法  
  foo(baz) {  
    this[bar](baz);  
  }  
  
  // 私有方法  
  [bar](baz) {  
    return this[snaf] = baz;  
  }  
  
  // ...  
};
```

还有一种方法是利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值。

Class的私有方法和私有属性

目前，有一个提案，为class加了私有属性。方法是在属性名之前，使用#表示。

```
> class IncreasingCounter {
  #count = 0;
  get value() {
    console.log('Getting the current value!');
    return this.#count;
  }
  increment() {
    this.#count++;
  }
}
< undefined
> let o=new IncreasingCounter();
< undefined
> o.count
< undefined
> o.value
Getting the current value!
< 0
> |
```

```
const counter = new IncreasingCounter();
counter.#count // 报错
counter.#count = 42 // 报错
```

上面代码中，#count就是私有属性，只能在类的内部使用（this.#count）。如果在类的外部使用，就会报错。

Class的私有方法和私有属性

```
class Foo {
  #a;
  #b;
  constructor(a, b) {
    this.#a = a;
    this.#b = b;
  }
  #sum() {
    return #a + #b;
  }
  printSum() {
    console.log(this.#sum());
  }
}
```

这种写法不仅可以写私有属性，还可以用来写私有方法

```
class Counter {
  #xValue = 0;

  constructor() {
    super();
    // ...
  }

  get #x() { return #xValue; }
  set #x(value) {
    this.#xValue = value;
  }
}
```

私有属性也可以设置 getter 和 setter 方法。

私有属性不限于从this引用，只要是在类的内部，实例也可以引用私有属性。

Class的私有方法和私有属性

私有属性和私有方法前面，也可以加上static关键字，表示这是一个静态的私有属性或私有方法。

```
class FakeMath {  
    static PI = 22 / 7;  
    static #totallyRandomNumber = 4;  
  
    static #computeRandomNumber() {  
        return FakeMath.#totallyRandomNumber;  
    }  
  
    static random() {  
        console.log('I heard you like random numbers...')  
        return FakeMath.#computeRandomNumber();  
    }  
}  
  
FakeMath.PI // 3.142857142857143  
FakeMath.random()  
// I heard you like random numbers...  
// 4  
FakeMath.#totallyRandomNumber // 报错  
FakeMath.#computeRandomNumber() // 报错
```

上面代码中，#totallyRandomNumber是私有属性，#computeRandomNumber()是私有方法，只能在FakeMath这个类的内部调用，外部调用就会报错。



构造函数的新属性

构造函数的新属性

```
> function Person(name) {  
  if (new.target !== undefined) {  
    this.name = name;  
  } else {  
    console.log('必须使用 new 命令生成实例');  
  }  
}  
  
⏏ undefined  
  
> let o=new Person("万章");  
⏏ undefined  
  
> o.name  
⏏ "万章"  
  
> let o1=Person("银时")  
必须使用 new 命令生成实例  
⏏ undefined  
  
>
```

Class 内部调用new.target，返回当前 Class。

ES6 为new命令引入了一个new.target属性，该属性一般用在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

```
class Rectangle {  
  constructor(length, width) {  
    console.log(new.target === Rectangle);  
    this.length = length;  
    this.width = width;  
  }  
}  
  
var obj = new Rectangle(3, 4); // 输出 true
```