ECMAScript 6 入门

作者：阮一峰

目录

Class

1. Class基本语法

概述

JavaScript语言的传统方法是通过构造函数，定义并生成新对象。下面是一个例子。

```
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);
```

上面这种写法跟传统的面向对象语言（比如C++和Java）差异很大，很容易让新学习这门语言的程序员感到困惑。

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。基本上，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。上面的代码用ES6的"类"改写，就是下面这样。

```
//定义类
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

上面代码定义了一个"类"，可以看到里面有一个`constructor`方法，这就是构造方法，而`this`关键字则代表实例对象。也就是说，ES5的构造函数`Point`，对应ES6的`Point`类的构造方法。

`Point`类除了构造方法，还定义了一个`toString`方法。注意，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。另外，方法之间不需要逗号分隔，加了会报错。

ES6的类，完全可以看作构造函数的另一种写法。

```
class Point {
  // ...
}

typeof Point // "function"
Point === Point.prototype.constructor // true
```

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。

使用的时候，也是直接对类使用`new`命令，跟构造函数的用法完全一致。

```
class Bar {
  doStuff() {
    console.log('stuff');
  }
}

var b = new Bar();
b.doStuff() // "stuff"
```

构造函数的`prototype`属性，在ES6的"类"上面继续存在。事实上，类的所有方法都定义在类的`prototype`属性上面。

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

  toString(){
    // ...
  }

  toValue(){
    // ...
  }
}

// 等同于

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

在类的实例上面调用方法，其实就是调用原型上的方法。

```
class B {}
let b = new B();

b.constructor === B.prototype.constructor // true
```

上面代码中，`b`是B类的实例，它的`constructor`方法就是B类原型的`constructor`方法。

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(){}
});
```

`prototype`对象的`constructor`属性，直接指向"类"的本身，这与ES5的行为是一致的。

```
Point.prototype.constructor === Point // true
```

另外，类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

```
class Point {
  constructor(x, y) {
    // ...
  }

  toString() {
    // ...
  }
}

Object.keys(Point.prototype)
```

```
// []
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```

上面代码中，`toString`方法是`Point`类内部定义的方法，它是不可枚举的。这一点与ES5的行为不一致。

```
var Point = function (x, y) {
  // ...
};

Point.prototype.toString = function() {
  // ...
};

Object.keys(Point.prototype)
// ["toString"]
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```

上面代码采用ES5的写法，`toString`方法就是可枚举的。

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。

---

constructor方法

`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。

```
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。

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}

new Foo() instanceof Foo
// false
```

上面代码中，`constructor`函数返回一个全新的对象，结果导致实例对象不是`Foo`类的实例。

类的构造函数，不使用`new`是没法调用的，会报错。这是它跟普通构造函数的一个主要区别，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'
```

类的实例对象

生成类的实例对象的写法，与ES5完全一样，也是使用`new`命令。如果忘记加上`new`，像函数那样调用`Class`，将会报错。

```
// 报错
var point = Point(2, 3);

// 正确
var point = new Point(2, 3);
```

与ES5一样，实例的属性除非显式定义在其本身（即定义在`this`对象上），否则都是定义在原型上（即定义在`class`上）。

```
//定义类
```

```
class Point {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }

}

var point = new Point(2, 3);

point.toString() // (2, 3)

point.hasOwnProperty('x') // true
point.hasOwnProperty('y') // true
point.hasOwnProperty('toString') // false
point.__proto__.hasOwnProperty('toString') // true
```

上面代码中，`x`和`y`都是实例对象`point`自身的属性（因为定义在`this`变量上），所以`hasOwnProperty`方法返回`true`，而`toString`是原型对象的属性（因为定义在`Point`类上），所以`hasOwnProperty`方法返回`false`。这些都与ES5的行为保持一致。

与ES5一样，类的所有实例共享一个原型对象。

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);
```

```
p1.__proto__ === p2.__proto__
//true
```

上面代码中，`p1`和`p2`都是Point的实例，它们的原型都是Point，所以`__proto__`属性是相等的。

这也意味着，可以通过实例的`__proto__`属性为Class添加方法。

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);

p1.__proto__.printName = function () { return 'Oops' };

p1.printName() // "Oops"
p2.printName() // "Oops"

var p3 = new Point(4,2);
p3.printName() // "Oops"
```

上面代码在`p1`的原型上添加了一个`printName`方法，由于`p1`的原型就是`p2`的原型，因此`p2`也可以调用这个方法。而且，此后新建的实例`p3`也可以调用这个方法。这意味着，使用实例的`__proto__`属性改写原型，必须相当谨慎，不推荐使用，因为这会改变Class的原始定义，影响到所有实例。

不存在变量提升

Class不存在变量提升（hoist），这一点与ES5完全不同。

```
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。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

```
{
  let Foo = class {};
  class Bar extends Foo {
  }
}
```

上面的代码不会报错，因为`class`继承`Foo`的时候，`Foo`已经有定义了。但是，如果存在`class`的提升，上面代码就会报错，因为`class`会被提升到代码头部，而`let`命令是不提升的，所以导致`class`继承`Foo`的时候，`Foo`还没有定义。

Class表达式

与函数一样，类也可以使用表达式的形式定义。

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

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`MyClass`而不是`Me`，`Me`只在Class的内部代码可用，指代当前类。

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

上面代码表示，`Me`只在Class内部有定义。

如果类的内部没用到的话，可以省略`Me`，也就是可以写成下面的形式。

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

采用Class表达式，可以写出立即执行的Class。

```
let person = new class {
  constructor(name) {
    this.name = name;
  }

  sayName() {
    console.log(this.name);
  }
}('张三');

person.sayName(); // "张三"
```

上面代码中，`person`是一个立即执行的类的实例。

私有方法

私有方法是常见需求，但ES6不提供，只能通过变通方法模拟实现。

一种做法是在命名上加以区别。

```
class Widget {

  // 公有方法
  foo (baz) {
    this._bar(baz);
  }

  // 私有方法
  _bar(baz) {
    return this.snaf = baz;
  }

  // ...
}
```

上面代码中，`_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。

另一种方法就是索性将私有方法移出模块，因为模块内部的所有方法都是对外可见的。

```
class Widget {
  foo (baz) {
    bar.call(this, baz);
  }
```

```
  // ...
}

function bar(baz) {
  return this.snaf = baz;
}
```

上面代码中，`foo`是公有方法，内部调用了`bar.call(this, baz)`。这使得`bar`实际上成为了当前模块的私有方法。

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。

```
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;
  }

  // ...
};
```

上面代码中，`bar`和`snaf`都是`Symbol`值，导致第三方无法获取到它们，因此达到了私

有方法和私有属性的效果。

---

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);
  }
}

const logger = new Logger();
const { printName } = logger;
printName(); // TypeError: Cannot read property 'print' of undefined
```

上面代码中，`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，因为找不到`print`方法而导致报错。

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。

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

  // ...
}
```

另一种解决方法是使用箭头函数。

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

  // ...
}
```

还有一种解决方法是使用`Proxy`，获取方法的时候，自动绑定`this`。

```
function selfish (target) {
  const cache = new WeakMap();
  const handler = {
    get (target, key) {
      const value = Reflect.get(target, key);
      if (typeof value !== 'function') {
        return value;
      }
      if (!cache.has(value)) {
        cache.set(value, value.bind(target));
```

```
      }
      return cache.get(value);
    }
  };
  const proxy = new Proxy(target, handler);
  return proxy;
}

const logger = selfish(new Logger());
```

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。

---

name属性

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

`name`属性总是返回紧跟在`class`关键字后面的类名。

---

2. Class的继承

基本用法

Class之间可以通过`extends`关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

```
class ColorPoint extends Point {}
```

上面代码定义了一个`ColorPoint`类，该类通过`extends`关键字，继承了`Point`类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个`Point`类。下面，我们在`ColorPoint`内部加上代码。

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // 调用父类的constructor(x, y)
    this.color = color;
  }

  toString() {
    return this.color + ' ' + super.toString(); // 调用父类的toString()
  }
```

```
}
```

上面代码中，`constructor`方法和`toString`方法之中，都出现了`super`关键字，它在这里表示父类的构造函数，用来新建父类的`this`对象。

子类必须在`constructor`方法中调用`super`方法，否则新建实例时会报错。这是因为子类没有自己的`this`对象，而是继承父类的`this`对象，然后对其进行加工。如果不调用`super`方法，子类就得不到`this`对象。

```
class Point { /* ... */ }

class ColorPoint extends Point {
  constructor() {
  }
}

let cp = new ColorPoint(); // ReferenceError
```

上面代码中，`ColorPoint`继承了父类`Point`，但是它的构造函数没有调用`super`方法，导致新建实例时报错。

ES5的继承，实质是先创造子类的实例对象`this`，然后再将父类的方法添加到`this`上面（`Parent.apply(this)`）。ES6的继承机制完全不同，实质是先创造父类的实例对象`this`（所以必须先调用`super`方法），然后再用子类的构造函数修改`this`。

如果子类没有定义`constructor`方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有`constructor`方法。

```
constructor(...args) {
  super(...args);
}
```

另一个需要注意的地方是，在子类的构造函数中，只有调用`super`之后，才可以使用`this`关键字，否则会报错。这是因为子类实例的构建，是基于对父类实例加工，只有`super`方法才能返回父类实例。

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    this.color = color; // ReferenceError
    super(x, y);
    this.color = color; // 正确
  }
}
```

上面代码中，子类的`constructor`方法没有调用`super`之前，就使用`this`关键字，结果报错，而放在`super`方法之后就是正确的。

下面是生成子类实例的代码。

```
let cp = new ColorPoint(25, 8, 'green');

cp instanceof ColorPoint // true
```

```
cp instanceof Point // true
```

上面代码中，实例对象`cp`同时是`ColorPoint`和`Point`两个类的实例，这与ES5的行为完全一致。

类的prototype属性和__proto__属性

大多数浏览器的ES5实现之中，每一个对象都有`__proto__`属性，指向对应的构造函数的prototype属性。Class作为构造函数的语法糖，同时有prototype属性和`__proto__`属性，因此同时存在两条继承链。

（1）子类的`__proto__`属性，表示构造函数的继承，总是指向父类。

（2）子类`prototype`属性的`__proto__`属性，表示方法的继承，总是指向父类的`prototype`属性。

```
class A {
}

class B extends A {
}

B.__proto__ === A // true
B.prototype.__proto__ === A.prototype // true
```

上面代码中，子类`B`的`__proto__`属性指向父类`A`，子类`B`的`prototype`属性的`__proto__`属性指向父类`A`的`prototype`属性。

这样的结果是因为，类的继承是按照下面的模式实现的。

```
class A {
}

class B {
}

// B的实例继承A的实例
Object.setPrototypeOf(B.prototype, A.prototype);

// B继承A的静态属性
Object.setPrototypeOf(B, A);
```

《对象的扩展》一章给出过`Object.setPrototypeOf`方法的实现。

```
Object.setPrototypeOf = function (obj, proto) {
  obj.__proto__ = proto;
  return obj;
}
```

因此，就得到了上面的结果。

```
Object.setPrototypeOf(B.prototype, A.prototype);
// 等同于
B.prototype.__proto__ = A.prototype;

Object.setPrototypeOf(B, A);
// 等同于
B.__proto__ = A;
```

这两条继承链，可以这样理解：作为一个对象，子类（`B`）的原型（`__proto__`属性）是父类（`A`）；作为一个构造函数，子类（`B`）的原型（`prototype`属性）是父类的实例。

```
Object.create(A.prototype);
// 等同于
B.prototype.__proto__ = A.prototype;
```

Extends 的继承目标

`extends`关键字后面可以跟多种类型的值。

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

上面代码的`A`，只要是一个有`prototype`属性的函数，就能被`B`继承。由于函数都有`prototype`属性（除了`Function.prototype`函数），因此`A`可以是任意函数。

下面，讨论三种特殊情况。

第一种特殊情况，子类继承Object类。

```
class A extends Object {
}

A.__proto__ === Object // true
```

```
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，`A`其实就是构造函数`Object`的复制，`A`的实例就是`Object`的实例。

第二种特殊情况，不存在任何继承。

```
class A {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，A作为一个基类（即不存在任何继承），就是一个普通函数，所以直接继承`Funciton.prototype`。但是，`A`调用后返回一个空对象（即`Object`实例），所以`A.prototype.__proto__`指向构造函数（`Object`）的`prototype`属性。

第三种特殊情况，子类继承`null`。

```
class A extends null {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === undefined // true
```

这种情况与第二种情况非常像。`A`也是一个普通函数，所以直接继承`Funciton.prototype`。但是，A调用后返回的对象不继承任何方法，所以它的`__proto__`指向`Function.prototype`，即实质上执行了下面的代码。

```
class C extends null {
```

```
  constructor() { return Object.create(null); }
}
```

Object.getPrototypeOf()

`Object.getPrototypeOf`方法可以用来从子类上获取父类。

```
Object.getPrototypeOf(ColorPoint) === Point
// true
```

因此，可以使用这个方法判断，一个类是否继承了另一个类。

---

super关键字

`super`这个关键字，有两种用法，含义不同。

（1）作为函数调用时（即`super(...args)`），`super`代表父类的构造函数。

（2）作为对象调用时（即`super.prop`或`super.method()`），`super`代表父类。注意，此时`super`即可以引用父类实例的属性和方法，也可以引用父类的静态方法。

```
class B extends A {
  get m() {
    return this._p * super._p;
```

```
  }
  set m() {
    throw new Error('该属性只读');
  }
}
```

上面代码中，子类通过`super`关键字，调用父类实例的`_p`属性。

由于，对象总是继承其他对象的，所以可以在任意一个对象中，使用`super`关键字。

```
var obj = {
  toString() {
    return "MyObject: " + super.toString();
  }
};

obj.toString(); // MyObject: [object Object]
```

实例的__proto__属性

子类实例的__proto__属性的__proto__属性，指向父类实例的__proto__属性。也就是说，子类的原型的原型，是父类的原型。

```
var p1 = new Point(2, 3);
var p2 = new ColorPoint(2, 3, 'red');

p2.__proto__ === p1.__proto__ // false
p2.__proto__.__proto__ === p1.__proto__ // true
```

上面代码中，`ColorPoint`继承了`Point`，导致前者原型的原型是后者的原型。

因此，通过子类实例的`__proto__.__proto__`属性，可以修改父类实例的行为。

```
p2.__proto__.__proto__.printName = function () {
  console.log('Ha');
};

p1.printName() // "Ha"
```

上面代码在`ColorPoint`的实例`p2`上向`Point`类添加方法，结果影响到了`Point`的实例`p1`。

---

3. 原生构造函数的继承

原生构造函数是指语言内置的构造函数，通常用来生成数据结构。ECMAScript的原生构造函数大致有下面这些。

- Boolean()
- Number()
- String()
- Array()
- Date()
- Function()
- RegExp()

- Error()
- Object()

以前，这些原生构造函数是无法继承的，比如，不能自己定义一个`Array`的子类。

```
function MyArray() {
  Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
  constructor: {
    value: MyArray,
    writable: true,
    configurable: true,
    enumerable: true
  }
});
```

上面代码定义了一个继承Array的`MyArray`类。但是，这个类的行为与`Array`完全不一致。

```
var colors = new MyArray();
colors[0] = "red";
colors.length  // 0

colors.length = 0;
colors[0]  // "red"
```

之所以会发生这种情况，是因为子类无法获得原生构造函数的内部属性，通过`Array.apply()`或者分配给原型对象都不行。原生构造函数会忽略`apply`方法传入

的`this`，也就是说，原生构造函数的`this`无法绑定，导致拿不到内部属性。

ES5是先新建子类的实例对象`this`，再将父类的属性添加到子类上，由于父类的内部属性无法获取，导致无法继承原生的构造函数。比如，Array构造函数有一个内部属性`[[DefineOwnProperty]]`，用来定义新属性时，更新`length`属性，这个内部属性无法在子类获取，导致子类的`length`属性行为不正常。

下面的例子中，我们想让一个普通对象继承`Error`对象。

```
var e = {};

Object.getOwnPropertyNames(Error.call(e))
// [ 'stack' ]

Object.getOwnPropertyNames(e)
// []
```

上面代码中，我们想通过`Error.call(e)`这种写法，让普通对象`e`具有`Error`对象的实例属性。但是，`Error.call()`完全忽略传入的第一个参数，而是返回一个新对象，`e`本身没有任何变化。这证明了`Error.call(e)`这种写法，无法继承原生构造函数。

ES6允许继承原生构造函数定义子类，因为ES6是先新建父类的实例对象`this`，然后再用子类的构造函数修饰`this`，使得父类的所有行为都可以继承。下面是一个继承`Array`的例子。

```
class MyArray extends Array {
  constructor(...args) {
```

```
    super(...args);
  }
}

var arr = new MyArray();
arr[0] = 12;
arr.length // 1

arr.length = 0;
arr[0] // undefined
```

上面代码定义了一个`MyArray`类，继承了`Array`构造函数，因此就可以从`MyArray`生成数组的实例。这意味着，ES6可以自定义原生数据结构（比如Array、String等）的子类，这是ES5无法做到的。

上面这个例子也说明，`extends`关键字不仅可以用来继承类，还可以用来继承原生的构造函数。因此可以在原生数据结构的基础上，定义自己的数据结构。下面就是定义了一个带版本功能的数组。

```
class VersionedArray extends Array {
  constructor() {
    super();
    this.history = [[]];
  }
  commit() {
    this.history.push(this.slice());
  }
  revert() {
    this.splice(0, this.length, ...this.history[this.history.length - 1
  }
}
```

```
var x = new VersionedArray();

x.push(1);
x.push(2);
x // [1, 2]
x.history // [[]]

x.commit();
x.history // [[], [1, 2]]
x.push(3);
x // [1, 2, 3]

x.revert();
x // [1, 2]
```

上面代码中，`VersionedArray`结构会通过`commit`方法，将自己的当前状态存入`history`属性，然后通过`revert`方法，可以撤销当前版本，回到上一个版本。除此之外，`VersionedArray`依然是一个数组，所有原生的数组方法都可以在它上面调用。

下面是一个自定义`Error`子类的例子。

```
class ExtendableError extends Error {
  constructor(message) {
    super();
    this.message = message;
    this.stack = (new Error()).stack;
    this.name = this.constructor.name;
  }
}

class MyError extends ExtendableError {
  constructor(m) {
```

```
    super(m);
  }
}

var myerror = new MyError('ll');
myerror.message // "ll"
myerror instanceof Error // true
myerror.name // "MyError"
myerror.stack
// Error
//     at MyError.ExtendableError
//     ...
```

注意，继承`Object`的子类，有一个行为差异。

```
class NewObj extends Object{
  constructor(){
    super(...arguments);
  }
}
var o = new NewObj({attr: true});
console.log(o.attr === true);  // false
```

上面代码中，`NewObj`继承了`Object`，但是无法通过`super`方法向父类`Object`传参。这是因为ES6改变了`Object`构造函数的行为，一旦发现`Object`方法不是通过`new Object()`这种形式调用，ES6规定`Object`构造函数会忽略参数。

---

4. Class的取值函数（getter）和存值函数（setter）

与ES5一样，在Class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'
```

上面代码中，`prop`属性有对应的存值函数和取值函数，因此赋值和读取行为都被自定义了。

存值函数和取值函数是设置在属性的descriptor对象上的。

```
class CustomHTMLElement {
  constructor(element) {
    this.element = element;
  }
```

```
  get html() {
    return this.element.innerHTML;
  }

  set html(value) {
    this.element.innerHTML = value;
  }
}

var descriptor = Object.getOwnPropertyDescriptor(
  CustomHTMLElement.prototype, "html");
"get" in descriptor  // true
"set" in descriptor  // true
```

上面代码中，存值函数和取值函数是定义在`html`属性的描述对象上面，这与ES5完全一致。

---

5. Class的Generator方法

如果某个方法之前加上星号（`*`），就表示该方法是一个Generator函数。

```
class Foo {
  constructor(...args) {
    this.args = args;
  }
  * [Symbol.iterator]() {
    for (let arg of this.args) {
      yield arg;
    }
```

```
  }
}

for (let x of new Foo('hello', 'world')) {
  console.log(x);
}
// hello
// world
```

上面代码中，Foo类的Symbol.iterator方法前有一个星号，表示该方法是一个Generator函数。Symbol.iterator方法返回一个Foo类的默认遍历器，for...of循环会自动调用这个遍历器。

---

6. 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
```

上面代码中，`Foo`类的`classMethod`方法前有`static`关键字，表明该方法是一个静态方法，可以直接在`Foo`类上调用（`Foo.classMethod()`），而不是在`Foo`类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。

父类的静态方法，可以被子类继承。

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

class Bar extends Foo {
}

Bar.classMethod(); // 'hello'
```

上面代码中，父类`Foo`有一个静态方法，子类`Bar`可以调用这个方法。

静态方法也是可以从`super`对象上调用的。

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

class Bar extends Foo {
  static classMethod() {
```

```
    return super.classMethod() + ', too';
  }
}

Bar.classMethod();
```

---

7. Class的静态属性和实例属性

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
```

上面的写法为`Foo`类定义了一个静态属性`prop`。

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。

```
// 以下两种写法都无效
class Foo {
  // 写法一
  prop: 2

  // 写法二
```

```
  static prop: 2
}

Foo.prop // undefined
```

ES7有一个静态属性的提案，目前Babel转码器支持。

这个提案对实例属性和静态属性，都规定了新的写法。

（1）类的实例属性

类的实例属性可以用等式，写入类的定义之中。

```
class MyClass {
  myProp = 42;

  constructor() {
    console.log(this.myProp); // 42
  }
}
```

上面代码中，`myProp`就是`MyClass`的实例属性。在`MyClass`的实例上，可以读取这个属性。

以前，我们定义实例属性，只能写在类的`constructor`方法里面。

```
class ReactCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
```

```
    };
  }
}
```

上面代码中，构造方法`constructor`里面，定义了`this.state`属性。

有了新的写法以后，可以不在`constructor`方法里面定义。

```
class ReactCounter extends React.Component {
  state = {
    count: 0
  };
}
```

这种写法比以前更清晰。

为了可读性的目的，对于那些在`constructor`里面已经定义的实例属性，新写法允许直接列出。

```
class ReactCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  state;
}
```

（2）类的静态属性

类的静态属性只要在上面的实例属性写法前面，加上`static`关键字就可以了。

```
class MyClass {
  static myStaticProp = 42;

  constructor() {
    console.log(MyClass.myProp); // 42
  }
}
```

同样的，这个新写法大大方便了静态属性的表达。

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

// 新写法
class Foo {
  static prop = 1;
}
```

上面代码中，老写法的静态属性定义在类的外部。整个类生成以后，再生成静态属性。这样让人很容易忽略这个静态属性，也不符合相关代码应该放在一起的代码组织原则。另外，新写法是显式声明（declarative），而不是赋值处理，语义更好。

---

8. new.target属性

new是从构造函数生成实例的命令。ES6为new命令引入了一个new.target属性，（在构造函数中）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。

```
function Person(name) {
  if (new.target !== undefined) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

// 另一种写法
function Person(name) {
  if (new.target === Person) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三');  // 报错
```

上面代码确保构造函数只能通过new命令调用。

Class内部调用new.target，返回当前Class。

```
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
```

需要注意的是，子类继承父类时，`new.target`会返回子类。

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    // ...
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length);
  }
}

var obj = new Square(3); // 输出 false
```

上面代码中，`new.target`会返回子类。

利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

```
class Shape {
  constructor() {
    if (new.target === Shape) {
```

```
      throw new Error('本类不能实例化');
    }
  }
}

class Rectangle extends Shape {
  constructor(length, width) {
    super();
    // ...
  }
}

var x = new Shape();  // 报错
var y = new Rectangle(3, 4);  // 正确
```

上面代码中，`Shape`类不能被实例化，只能用于继承。

注意，在函数外部，使用`new.target`会报错。

---

9. Mixin模式的实现

Mixin模式指的是，将多个类的接口"混入"（mix in）另一个类。它在ES6的实现如下。

```
function mix(...mixins) {
  class Mix {}

  for (let mixin of mixins) {
    copyProperties(Mix, mixin);
    copyProperties(Mix.prototype, mixin.prototype);
  }
```

```
  return Mix;
}

function copyProperties(target, source) {
  for (let key of Reflect.ownKeys(source)) {
    if ( key !== "constructor"
      && key !== "prototype"
      && key !== "name"
    ) {
      let desc = Object.getOwnPropertyDescriptor(source, key);
      Object.defineProperty(target, key, desc);
    }
  }
}
```

上面代码的`mix`函数，可以将多个对象合成为一个类。使用的时候，只要继承这个类即可。

```
class DistributedEdit extends mix(Loggable, Serializable) {
  // ...
}
```

留言