Class

1. 类的由来

JavaScript 语言中，生成实例对象的传统方法是通过构造函数：

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

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。

上例用 ES6 的 `Class` 改写，就是下面这样：

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

上面 `constructor()` 方法，这就是构造方法，而 `this` 关键字则代表实例对象。这种新的 `Class` 写法，本质上与 ES5 的构造函数 `Point` 是一致的。

`Point` 类除了构造方法，还定义了一个 `toString()` 方法。**定义 `toString()` 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。方法与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
```

类的数据类型就是函数，类本身就指向构造函数。

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 = {
  constructor() {},
  toString() {},
  toValue() {},
};
```

上例中，constructor()、toString()、toValue() 这三个方法，其实都是定义在 Point.prototype 上面。

因此，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 {}
const b = new B();
b.constructor === B.prototype.constructor; // true
```

上例中，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
```

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"]
```

上例中，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。

```
let Point = function (x, y) { /* */ };
Point.prototype.toString = function () { /* */ };
Object.keys(Point.prototype);                // ["toString"]
Object.getOwnPropertyNames(Point.prototype); // ["constructor","toString"]
```

上例采用 ES5 的写法，`toString()` 方法就是可枚举的。

2. 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。

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

上例中，定义了一个空的类 `Point`，JavaScript 引擎会自动为它添加一个空的 `constructor()` 方法。

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
```

上例中，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。

类必须使用 new 调用，否则会报错。 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'
```

3. 类的实例

类的属性和方法，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）。

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

let 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
```

上例中，`x` 和 `y` 都是实例对象 `point` 自身的属性（因为定义在 `this` 对象上），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`。

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
p1.__proto__ === Point.prototype; // true
```

4. 实例属性的新写法

实例属性现在除了可以定义在 `constructor()` 方法里面的 `this` 上面，也可以定义在类内部的最顶层。

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

上例中，实例属性 `_count` 定义在 `constructor()` 方法里面的 `this` 上面。

现在的新写法是，这个属性也可以定义在类的最顶层，其他都不变。

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

上例中，实例属性 `_count` 与取值函数 `value()` 和 `increment()` 方法，处于同一个层级。这时，不需要在实例属性前面加上 `this`。

新写法定义的属性是实例对象自身的属性，而不是定义在实例对象的原型上面。

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。

5. 取值函数（getter）和存值函数（setter）

```
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'
```

6. 属性表达式

```
let methodName = 'getArea';
class Square {
  constructor(length) { /* */ }
  [methodName]() { /* */ }
}
```

`Square` 类的方法名 `getArea`，是从表达式得到的。

7. Class 表达式

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"
```

8. 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'
let foo = new Foo();
foo.classMethod(); // TypeError: foo.classMethod is not a function
```

如果静态方法包含this关键字，这个 `this` 指的是类，而不是实例。

```
class Foo {
  static bar() {
    this.baz();
  }
  static baz() {
    console.log('hello');
  }
  baz() {
    console.log('world');
  }
}
Foo.bar(); // hello
let f = new Foo();
f.baz();   // world
```

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'
```

9. 静态属性

在实例属性的前面，加上 `static` 关键字。

```
class Foo {
  static prop = 1;
}
Foo.prop; // 1
let f = new Foo();
f.prop;   // undefined
```

10. 私有属性

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++;
  }
}
```

#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。

```
const counter = new IncreasingCounter();
counter.#count // 报错，从 Chrome 111 开始，开发者工具里面可以读写私有属性，不会报错，
原因是 Chrome 团队认为这样方便调试。其他浏览器和 Chrome 111 以下会报错。
counter.#count = 42 // 报错
```

上例中，在类的外部，读取或写入私有属性 #count，都会报错。

11. in 运算符

in 运算符，使它也可以用来判断私有属性。也可以跟this一起配合使用。

```
class A {
  #foo = 0;
  m() {
    console.log(#foo in this);
  }
}
let a = new A();
a.m(); // true
```

判断私有属性时，in 只能用在类的内部。另外，判断所针对的私有属性，一定要先声明，否则会报错。

```
class A {
  m() {
    console.log(#foo in this); // Private field '#foo' must be declared in an
enclosing class。私有字段“#foo”必须在封闭类中声明
  }
}
```