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。

上面的代码用 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() {},
};
```

上面代码中，`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
```

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。

```
let 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。

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() {}
}
```

上面代码中，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` 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
```