ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

Class 的基本语法

---

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。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 `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。这种新的 Class 写法，本质上与本章开头的 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。注意，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。另外，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
```

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');
  }
}

const 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 = {
  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
```

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。

---

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

生成类的实例的写法，与 ES5 完全一样，也是使用 `new` 命令。前面说过，如果忘记加上 `new`，像函数那样调用 `Class()`，将会报错。

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

// 报错
var point = Point(2, 3);

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

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 + ')';
  }
}

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` 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`。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.prototype`，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“类”添加方法。

> `__proto__` 并不是语言本身的特性，这是各大厂商具体实现时添加的私有属性，虽然目前很多现代浏览器的 JS 引擎中都提供了这个私有属性，但依旧不建议在生产中使用该属性，避免对环境产生依赖。生产环境中，我们可以使用 `Object.getPrototypeOf()` 方法来获取实例对象的原型，然后再来为原型添加方法/属性。

```
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“类”的原始定义，影响到所有实例。

---

4. 实例属性的新写法

ES2022 为类的实例属性，又规定了一种新写法。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++;
```

```
  }
}
```

上面示例中，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++;
  }
}
```

上面代码中，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` 。

注意，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。

```
class foo {
  bar = 'hello';
  baz = 'world';

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

上面的代码，一眼就能看出， `foo` 类有两个实例属性，一目了然。另外，写起来也比较简洁。

---

5. 取值函数（getter）和存值函数（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'
```

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。

---

6. 属性表达式

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。

---

7. 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 `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。

```
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。

---

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'

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。

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。

```
class Foo {
  static bar() {
    this.baz();
  }
```

```
  static baz() {
    console.log('hello');
  }
  baz() {
    console.log('world');
  }
}

Foo.bar() // hello
```

上面代码中，静态方法 `bar` 调用了 `this.baz` ，这里的 `this` 指的是 `Foo` 类，而不是 `Foo` 的实例，等同于调用 `Foo.baz` 。另外，从这个例子还可以看出，静态方法可以与非静态方法重名。

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() // "hello, too"
```

---

9. 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
```

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。现在有一个提案提供了类的静态属性，写法是在实例属性的前面，加上 `static` 关键字。

```
class MyClass {
  static myStaticProp = 42;

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

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。

---

10. 私有方法和私有属性

早期解决方案

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，但早期的 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类，因为类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` 是公开方法，内部调用了 `bar.call(this, baz)` 。这使得 `bar()` 实际上成为了当前类的私有方法。

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，一般情况下无法获取到它们，因此达到了私有方法和私有属性的效果。但是也不是绝对不行， `Reflect.ownKeys()` 依然可以拿到它们。

```
const inst = new myClass();

Reflect.ownKeys(myClass.prototype)
// [ 'constructor', 'foo', Symbol(bar) ]
```

上面代码中，Symbol 值的属性名依然可以从类的外部拿到。

私有属性的正式写法

ES2022正式为 `class` 添加了私有属性，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++;
```

```
  }
}
```

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。

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

上面示例中，在类的外部，读取或写入私有属性 `#count`，都会报错。

另外，不管在类的内部或外部，读取一个不存在的私有属性，也都会报错。这跟公开属性的行为完全不同，如果读取一个不存在的公开属性，不会报错，只会返回 `undefined`。

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

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

上面示例中，`#myCount` 是一个不存在的私有属性，不管在函数内部或外部，读取该属性都会导致报错。

注意，私有属性的属性名必须包括 `#`，如果不带 `#`，会被当作另一个属性。

```
class Point {
  #x;

  constructor(x = 0) {
    this.#x = +x;
  }

  get x() {
    return this.#x;
  }

  set x(value) {
    this.#x = +value;
  }
}
```

上面代码中，`#x` 就是私有属性，在 `Point` 类之外是读取不到这个属性的。由于井号 `#` 是属性名的一部分，使用时必须带有 `#` 一起使用，所以 `#x` 和 `x` 是两个不同的属性。

这种写法不仅可以写私有属性，还可以用来写私有方法。

```
class Foo {
  #a;
  #b;
  constructor(a, b) {
    this.#a = a;
    this.#b = b;
  }
  #sum() {
```

```
    return this.#a + this.#b;
  }
  printSum() {
    console.log(this.#sum());
  }
}
```

上面示例中，`#sum()` 就是一个私有方法。

另外，私有属性也可以设置 getter 和 setter 方法。

```
class Counter {
  #xValue = 0;

  constructor() {
    console.log(this.#x);
  }

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

上面代码中，`#x` 是一个私有属性，它的读写都通过 `get #x()` 和 `set #x()` 操作另一个私有属性 `#xValue` 来完成。

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 Foo {
  #privateValue = 42;
  static getPrivateValue(foo) {
    return foo.#privateValue;
  }
}

Foo.getPrivateValue(new Foo()); // 42
```

上面代码允许从实例 `foo` 上面引用私有属性。

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。

---

in 运算符

前面说过，直接访问某个类不存在的私有属性会报错，但是访问不存在的公开属性不会报错。这个特性可以用来判断，某个对象是否为类的实例。

```
class C {
  #brand;

  static isC(obj) {
    try {
      obj.#brand;
      return true;
    } catch {
      return false;
    }
  }
}
```

上面示例中，类 `C` 的静态方法 `isC()` 就用来判断，某个对象是否为 `C` 的实例。它采用的方法就是，访问该对象的私有属性 `#brand`。如果不报错，就会返回 `true`；如果报错，就说明该对象不是当前类的实例，从而 `catch` 部分返回 `false`。

因此，`try...catch` 结构可以用来判断某个私有属性是否存在。但是，这样的写法很麻烦，代码可读性很差，ES2022 改进了 `in` 运算符，使它也可以用来判断私有属性。

```
class C {
  #brand;

  static isC(obj) {
    if (#brand in obj) {
      // 私有属性 #brand 存在
      return true;
    } else {
      // 私有属性 #foo 不存在
      return false;
    }
  }
}
```

上面示例中，`in` 运算符判断某个对象是否有私有属性 `#foo`。它不会报错，而是返回一个布尔值。

这种用法的 `in`，也可以跟 `this` 一起配合使用。

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

注意，判断私有属性时，`in` 只能用在类的内部。

子类从父类继承的私有属性，也可以使用 `in` 运算符来判断

```
class A {
  #foo = 0;
  static test(obj) {
    console.log(#foo in obj);
  }
}

class SubA extends A {};

A.test(new SubA()) // true
```

上面示例中， `SubA` 从父类继承了私有属性 `#foo` ， `in` 运算符也有效。

注意， `in` 运算符对于 `Object.create()` 、 `Object.setPrototypeOf` 形成的继承，是无效的，因为这种继承不会传递私有属性。

```
class A {
  #foo = 0;
  static test(obj) {
    console.log(#foo in obj);
  }
}
const a = new A();

const o1 = Object.create(a);
A.test(o1) // false
A.test(o1.__proto__) // true

const o2 = {};
Object.setPrototypeOf(o2, a);
A.test(o2) // false
A.test(o2.__proto__) // true
```

上面示例中，对于修改原型链形成的继承，子类都取不到父类的私有属性，所以 `in` 运算符无效。

---

11. 静态块

静态属性的一个问题是，如果它有初始化逻辑，这个逻辑要么写在类的外部，要么写在 `constructor()` 方法里面。

```
class C {
  static x = 234;
  static y;
  static z;
}

try {
  const obj = doSomethingWith(C.x);
  C.y = obj.y
  C.z = obj.z;
} catch {
  C.y = ...;
  C.z = ...;
}
```

上面示例中，静态属性 `y` 和 `z` 的值依赖于静态属性 `x` 的运算结果，这段初始化逻辑写在类的外部（上例的 `try...catch` 代码块）。另一种方法是写到类的 `constructor()` 方法里面。这两种方法都不是很理想，前者是将类的内部逻辑写到了外部，后者则是每次新建实例都会运行一次。

为了解决这个问题，ES2022 引入了静态块（static block），允许在类的内部设置一个代码块，在类生成时运行且只运行一次，主要作用是对静态属性进行初始化。以后，新建类的实例时，这个块就不运行了。

```
class C {
  static x = ...;
  static y;
  static z;

  static {
    try {
      const obj = doSomethingWith(this.x);
      this.y = obj.y;
      this.z = obj.z;
    }
    catch {
      this.y = ...;
      this.z = ...;
    }
  }
}
```

上面代码中，类的内部有一个 static 代码块，这就是静态块。它的好处是将静态属性 `y` 和 `z` 的初始化逻辑，写入了类的内部，而且只运行一次。

每个类允许有多个静态块，每个静态块中只能访问之前声明的静态属性。另外，静态块的内部不能有 `return` 语句。

静态块内部可以使用类名或 `this` ，指代当前类。

```
class C {
  static x = 1;
  static {
    this.x; // 1
    // 或者
    C.x; // 1
  }
}
```

上面示例中， `this.x` 和 `C.x` 都能获取静态属性 `x` 。

除了静态属性的初始化，静态块还有一个作用，就是将私有属性与类的外部代码分享。

```
let getX;

export class C {
  #x = 1;
  static {
    getX = obj => obj.#x;
  }
}

console.log(getX(new C())); // 1
```

上面示例中， `#x` 是类的私有属性，如果类外部的 `getX()` 方法希望获取这个属性，以前是要写在类的 `constructor()` 方法里面，这样的话，每次新建实例都会定义一次 `getX()` 方法。现在可以写在静态块里面，这样的话，只在类生成时定义一次。

---

12. 类的注意点

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。

不存在提升

类不存在变量提升（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 `Bar` 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 `Bar` 继承 `Foo` 的时候， `Foo` 还没有定义。

---

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。

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。

---

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（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。

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 Obj {
  constructor() {
    this.getThis = () => this;
  }
}

const myObj = new Obj();
myObj.getThis() === myObj // true
```

箭头函数内部的 `this` 总是指向定义时所在的对象。上面代码中，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。

还有一种解决方法是使用 `Proxy` ，获取方法的时候，自动绑定

```
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());
```

---

13. new.target 属性

new 是从构造函数生成实例对象的命令。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 命令或 Reflect.construct() 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 命令生成实例');
  }
}

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

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, width) {
    super(length, width);
  }
}

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。

留言