ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

装饰器

[说明] Decorator 提案经历了重大的语法变化，目前处于第三阶段，定案之前不知道是否还有变化。本章现在属于草稿阶段，凡是标注“新语法”的章节，都是基于当前的语法，不过没有详细整理，只是一些原始材料；未标注“新语法”的章节基于以前的语法，是过去遗留的稿子。之所以保留以前的内容，有两个原因，一是 TypeScript 装饰器会用到这些语法，二是里面包含不少有价值的内容。等到标准完全定案，本章将彻底重写：删去过时内容，补充材料，增加解释。（2022年6月）

---

1. 简介（新语法）

装饰器（Decorator）用来增强 JavaScript 类（class）的功能，许多面向对象的语言都有这种语法，目前有一个提案将其引入了 ECMAScript。

装饰器是一种函数，写成 `@ + 函数名`，可以用来装饰四种类型的值。

- 类
- 类的属性
- 类的方法
- 属性存取器（accessor）

下面的例子是装饰器放在类名和类方法名之前，大家可以感受一下写法。

```
@frozen class Foo {
  @configurable(false)
  @enumerable(true)
  method() {}

  @throttle(500)
  expensiveMethod() {}
}
```

上面代码一共使用了四个装饰器，一个用在类本身（@frozen），另外三个用在类方法（@configurable()、@enumerable()、@throttle()）。它们不仅增加了代码的可读性，清晰地表达了意图，而且提供一种方便的手段，增加或修改类的功能。

---

2. 装饰器 API（新语法）

装饰器是一个函数，API 的类型描述如下（TypeScript 写法）。

```
type Decorator = (value: Input, context: {
  kind: string;
  name: string | symbol;
  access: {
    get?(): unknown;
    set?(value: unknown): void;
  };
  private?: boolean;
  static?: boolean;
  addInitializer?(initializer: () => void): void;
}) => Output | void;
```

装饰器函数有两个参数。运行时，JavaScript 引擎会提供这两个参数。

- `value`：所要装饰的值，某些情况下可能是 `undefined`（装饰属性时）。
- `context`：上下文信息对象。

装饰器函数的返回值，是一个新版本的装饰对象，但也可以不返回任何值（void）。

`context` 对象有很多属性，其中 `kind` 属性表示属于哪一种装饰，其他属性的含义如下。

- `kind`：字符串，表示装饰类型，可能的取值有 `class`、`method`、`getter`、`setter`、`field`、`accessor`。
- `name`：被装饰的值的名称: The name of the value, or in the case of private elements the description of it (e.g. the readable name).
- `access`：对象，包含访问这个值的方法，即存值器和取值器。
- `static`：布尔值，该值是否为静态元素。
- `private`：布尔值，该值是否为私有元素。
- `addInitializer`：函数，允许用户增加初始化逻辑。

装饰器的执行步骤如下。

1.计算各个装饰器的值，按照从左到右，从上到下的顺序。
2.调用方法装饰器。
3.调用类装饰器。

---

3. 类的装饰

装饰器可以用来装饰整个类。

```
@testable
class MyTestableClass {
  // ...
}

function testable(target) {
  target.isTestable = true;
}

MyTestableClass.isTestable // true
```

上面代码中，`@testable` 就是一个装饰器。它修改了 `MyTestableClass` 这个类的行为，为它加上了静态属性 `isTestable`。`testable` 函数的参数 `target` 是 `MyTestableClass` 类本身。

基本上，装饰器的行为就是下面这样。

```
@decorator
class A {}

// 等同于

class A {}
A = decorator(A) || A;
```

也就是说，装饰器是一个对类进行处理的函数。装饰器函数的第一个参数，就是所要装饰的目标类。

```
function testable(target) {
  // ...
}
```

上面代码中， `testable` 函数的参数 `target` ，就是会被装饰的类。

如果觉得一个参数不够用，可以在装饰器外面再封装一层函数。

```
function testable(isTestable) {
  return function(target) {
    target.isTestable = isTestable;
  }
}

@testable(true)
class MyTestableClass {}
MyTestableClass.isTestable // true

@testable(false)
class MyClass {}
MyClass.isTestable // false
```

上面代码中，装饰器 `testable` 可以接受参数，这就等于可以修改装饰器的行为。

注意，装饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，装饰器能在编译阶段运行代码。也就是说，装饰器本质就是编译时执行的函数。

前面的例子是为类添加一个静态属性，如果想添加实例属性，可以通过目标类的 `prototype` 对象操作。

```
function testable(target) {
  target.prototype.isTestable = true;
}

@testable
class MyTestableClass {}

let obj = new MyTestableClass();
obj.isTestable // true
```

上面代码中，装饰器函数 `testable` 是在目标类的 `prototype` 对象上添加属性，因此就可以在实例上调用。

下面是另外一个例子。

```
// mixins.js
export function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list)
  }
```

```
}

// main.js
import { mixins } from './mixins.js'

const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();
obj.foo() // 'foo'
```

上面代码通过装饰器 `mixins`，把 `Foo` 对象的方法添加到了 `MyClass` 的实例上面。可以用 `Object.assign()` 模拟这个功能。

```
const Foo = {
  foo() { console.log('foo') }
};

class MyClass {}

Object.assign(MyClass.prototype, Foo);

let obj = new MyClass();
obj.foo() // 'foo'
```

实际开发中，React 与 Redux 库结合使用时，常常需要写成下面这样。

```
class MyReactComponent extends React.Component {}

export default connect(mapStateToProps, mapDispatchToProps)(MyReactComponent);
```

有了装饰器，就可以改写上面的代码。

```
@connect(mapStateToProps, mapDispatchToProps)
export default class MyReactComponent extends React.Component {}
```

相对来说，后一种写法看上去更容易理解。

---

4. 类装饰器（新语法）

类装饰器的类型描述如下。

```
type ClassDecorator = (value: Function, context: {
  kind: "class";
  name: string | undefined;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

类装饰器的第一个参数，就是被装饰的类。第二个参数是上下文对象，如果被装饰的类是一个匿名类，`name` 属性就为 `undefined`。

类装饰器可以返回一个新的类，取代原来的类，也可以不返回任何值。如果返回的不是构造函数，就会报错。

下面是一个例子。

```
function logged(value, { kind, name }) {
  if (kind === "class") {
    return class extends value {
      constructor(...args) {
        super(...args);
        console.log(`constructing an instance of ${name} with arguments ${args.join(", ")}`);
      }
    }
  }

  // ...
}

@logged
class C {}

new C(1);
// constructing an instance of C with arguments 1
```

如果不使用装饰器，类装饰器实际上执行的是下面的语法。

```
class C {}

C = logged(C, {
  kind: "class",
  name: "C",
}) ?? C;

new C(1);
```

---

5. 方法装饰器（新语法）

方法装饰器会修改类的方法。

```
class C {
  @trace
  toString() {
    return 'C';
  }
}

// 相当于
C.prototype.toString = trace(C.prototype.toString);
```

上面示例中，`@trace` 装饰 `toString()` 方法，就相当于修改了该方法。

方式装饰器使用 TypeScript 描述类型如下。

```
type ClassMethodDecorator = (value: Function, context: {
  kind: "method";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

方法装饰器的第一个参数 `value`，就是所要装饰的方法。

方法装饰器可以返回一个新函数，取代原来的方法，也可以不返回值，表示依然使用原来的方法。如果返回其他类型的值，就会报错。下面是一个例子。

```
function replaceMethod() {
  return function () {
    return `How are you, ${this.name}?`;
  }
}

class Person {
  constructor(name) {
    this.name = name;
  }
  @replaceMethod
  hello() {
    return `Hi ${this.name}!`;
  }
}

const robin = new Person('Robin');

robin.hello(), 'How are you, Robin?'
```

上面示例中，`@replaceMethod` 返回了一个新函数，取代了原来的 `hello()` 方法。

```
function logged(value, { kind, name }) {
  if (kind === "method") {
    return function (...args) {
      console.log(`starting ${name} with arguments ${args.join(", ")}`);
      const ret = value.call(this, ...args);
      console.log(`ending ${name}`);
      return ret;
    };
  }
}

class C {
  @logged
  m(arg) {}
}

new C().m(1);
// starting m with arguments 1
// ending m
```

上面示例中，装饰器 `@logged` 返回一个函数，代替原来的 `m()` 方法。

这里的装饰器实际上是一个语法糖，真正的操作是像下面这样，改掉原型链上面 `m()` 方法。

```
class C {
  m(arg) {}
}

C.prototype.m = logged(C.prototype.m, {
  kind: "method",
  name: "m",
  static: false,
  private: false,
}) ?? C.prototype.m;
```

6. 方法的装饰

装饰器不仅可以装饰类，还可以装饰类的属性。

```
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
```

上面代码中，装饰器 `readonly` 用来装饰“类”的 `name` 方法。

装饰器函数 `readonly` 一共可以接受三个参数。

```
function readonly(target, name, descriptor){
  // descriptor对象原来的值如下
  // {
  //   value: specifiedFunction,
  //   enumerable: false,
  //   configurable: true,
  //   writable: true
  // };
  descriptor.writable = false;
  return descriptor;
}

readonly(Person.prototype, 'name', descriptor);
// 类似于
Object.defineProperty(Person.prototype, 'name', descriptor);
```

装饰器第一个参数是类的原型对象，上例是 `Person.prototype` ，装饰器的本意是要“装饰”类的实例，但是这个时候实例还没生成，所以只能去装饰原型（这不同于类的装饰，那种情况时 `target` 参数指的是类本身）；第二个参数是所要装饰的属性名，第三个参数是该属性的描述对象。

另外，上面代码说明，装饰器（readonly）会修改属性的描述对象（descriptor），然后被修改的描述对象再用来定义属性。

下面是另一个例子，修改属性描述对象的 `enumerable` 属性，使得该属性不可遍历。

```
class Person {
  @nonenumerable
  get kidCount() { return this.children.length; }
}

function nonenumerable(target, name, descriptor) {
  descriptor.enumerable = false;
  return descriptor;
}
```

下面的 `@log` 装饰器，可以起到输出日志的作用。

```
class Math {
  @log
  add(a, b) {
    return a + b;
  }
}

function log(target, name, descriptor) {
  var oldValue = descriptor.value;

  descriptor.value = function() {
```

```
    console.log(`Calling ${name} with`, arguments);
    return oldValue.apply(this, arguments);
  };

  return descriptor;
}

const math = new Math();

// passed parameters should get logged now
math.add(2, 4);
```

上面代码中， `@log` 装饰器的作用就是在执行原始的操作之前，执行一次 `console.log` ，从而达到输出日志的目的。

装饰器有注释的作用。

```
@testable
class Person {
  @readonly
  @nonenumerable
  name() { return `${this.first} ${this.last}` }
}
```

从上面代码中，我们一眼就能看出， `Person` 类是可测试的，而 `name` 方法是只读和不可枚举的。

下面是使用 Decorator 写法的组件，看上去一目了然。

```
@Component({
  tag: 'my-component',
  styleUrl: 'my-component.scss'
})
export class MyComponent {
  @Prop() first: string;
  @Prop() last: string;
  @State() isVisible: boolean = true;

  render() {
    return (
      <p>Hello, my name is {this.first} {this.last}</p>
    );
  }
}
```

如果同一个方法有多个装饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

```
function dec(id){
  console.log('evaluated', id);
  return (target, property, descriptor) => console.log('executed', id);
}

class Example {
    @dec(1)
    @dec(2)
    method(){}
}
// evaluated 1
// evaluated 2
// executed 2
// executed 1
```

上面代码中，外层装饰器 `@dec(1)` 先进入，但是内层装饰器 `@dec(2)` 先执行。

除了注释，装饰器还能用来类型检查。所以，对于类来说，这项功能相当有用。从长期来看，它将是 JavaScript 代码静态分析的重要工具。

---

7. 为什么装饰器不能用于函数?

装饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。

```
var counter = 0;

var add = function () {
  counter++;
};

@add
function foo() {
}
```

上面的代码，意图是执行后 `counter` 等于 1，但是实际上结果是 `counter` 等于 0。因为函数提升，使得实际执行的代码是下面这样。

```
var counter;
var add;

@add
function foo() {
}

counter = 0;

add = function () {
  counter++;
};
```

下面是另一个例子。

```
var readOnly = require("some-decorator");

@readOnly
function foo() {
}
```

上面代码也有问题，因为实际执行是下面这样。

```
var readOnly;

@readOnly
function foo() {
}

readOnly = require("some-decorator");
```

总之，由于存在函数提升，使得装饰器不能用于函数。类是不会提升的，所以就没有这方面的问题。

另一方面，如果一定要装饰函数，可以采用高阶函数的形式直接执行。

```
function doSomething(name) {
  console.log('Hello, ' + name);
}
```

```
function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
    return result;
  }
}

const wrapped = loggingDecorator(doSomething);
```

---

8. 存取器装饰器（新语法）

存取器装饰器使用 TypeScript 描述的类型如下。

```
type ClassGetterDecorator = (value: Function, context: {
  kind: "getter";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;

type ClassSetterDecorator = (value: Function, context: {
  kind: "setter";
  name: string | symbol;
  access: { set(value: unknown): void };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

存取器装饰器的第一个参数就是原始的存值器（setter）和取值器（getter）。

存取器装饰器的返回值如果是一个函数，就会取代原来的存取器。本质上，就像方法装饰器一样，修改发生在类的原型对象上。它也可以不返回任何值，继续使用原来的存取器。如果返回其他类型的值，就会报错。

存取器装饰器对存值器（setter）和取值器（getter）是分开作用的。下面的例子里面，`@foo` 只装饰 `get x()`，不装饰 `set x()`。

```
class C {
  @foo
  get x() {
    // ...
  }

  set x(val) {
    // ...
  }
}
```

上一节的 `@logged` 装饰器稍加修改，就可以用在存取装饰器。

```
function logged(value, { kind, name }) {
  if (kind === "method" || kind === "getter" || kind === "setter") {
    return function (...args) {
      console.log(`starting ${name} with arguments ${args.join(", ")}`);
      const ret = value.call(this, ...args);
```

```
      console.log(`ending ${name}`);
      return ret;
    };
  }
}

class C {
  @logged
  set x(arg) {}
}

new C().x = 1
// starting x with arguments 1
// ending x
```

如果去掉语法糖，使用传统语法来写，就是改掉了类的原型链。

```
class C {
  set x(arg) {}
}

let { set } = Object.getOwnPropertyDescriptor(C.prototype, "x");
set = logged(set, {
  kind: "setter",
  name: "x",
  static: false,
  private: false,
}) ?? set;

Object.defineProperty(C.prototype, "x", { set });
```

---

9. 属性装饰器（新语法）

属性装饰器的类型描述如下。

```
type ClassFieldDecorator = (value: undefined, context: {
  kind: "field";
  name: string | symbol;
  access: { get(): unknown, set(value: unknown): void };
  static: boolean;
  private: boolean;
}) => (initialValue: unknown) => unknown | void;
```

属性装饰器的第一个参数是 `undefined`，即不输入值。用户可以选择让装饰器返回一个初始化函数，当该属性被赋值时，这个初始化函数会自动运行，它会收到属性的初始值，然后返回一个新的初始值。属性装饰器也可以不返回任何值。除了这两种情况，返回其他类型的值都会报错。

下面是一个例子。

```
function logged(value, { kind, name }) {
  if (kind === "field") {
    return function (initialValue) {
      console.log(`initializing ${name} with value ${initialValue}`);
      return initialValue;
    };
  }

  // ...
}
```

```
class C {
  @logged x = 1;
}

new C();
// initializing x with value 1
```

如果不使用装饰器语法，属性装饰器的实际作用如下。

```
let initializeX = logged(undefined, {
  kind: "field",
  name: "x",
  static: false,
  private: false,
}) ?? (initialValue) => initialValue;

class C {
  x = initializeX.call(this, 1);
}
```

---

10. accessor 命令（新语法）

类装饰器引入了一个新命令 `accessor`，用来属性的前缀。

```
class C {
  accessor x = 1;
}
```

它是一种简写形式，相当于声明属性 `x` 是私有属性 `#x` 的存取接口。上面的代码等同于下面的代码。

```
class C {
  #x = 1;

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

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

`accessor` 命令前面，还可以加上 `static` 命令和 `private` 命令。

```
class C {
  static accessor x = 1;
  accessor #y = 2;
}
```

`accessor` 命令前面还可以接受属性装饰器。

```
function logged(value, { kind, name }) {
  if (kind === "accessor") {
    let { get, set } = value;

    return {
```

```
      get() {
        console.log(`getting ${name}`);

        return get.call(this);
      },

      set(val) {
        console.log(`setting ${name} to ${val}`);

        return set.call(this, val);
      },

      init(initialValue) {
        console.log(`initializing ${name} with value ${initialValue}`);
        return initialValue;
      }
    };
  }

  // ...
}

class C {
  @logged accessor x = 1;
}

let c = new C();
// initializing x with value 1
c.x;
// getting x
c.x = 123;
// setting x to 123
```

上面的示例等同于使用 `@logged` 装饰器，改写 `accessor` 属性的 getter 和 setter 方法。

用于 `accessor` 的属性装饰器的类型描述如下。

```
type ClassAutoAccessorDecorator = (
  value: {
    get: () => unknown;
    set(value: unknown) => void;
  },
  context: {
    kind: "accessor";
    name: string | symbol;
    access: { get(): unknown, set(value: unknown): void };
    static: boolean;
    private: boolean;
    addInitializer(initializer: () => void): void;
  }
) => {
  get?: () => unknown;
  set?: (value: unknown) => void;
  initialize?: (initialValue: unknown) => unknown;
} | void;
```

`accessor` 命令的第一个参数接收到的是一个对象，包含了 `accessor` 命令定义的属性的存取器 get 和 set。属性装饰器可以返回一个新对象，其中包含了新的存取器，用来取代原来的，即相当于拦截了原来的存取器。此外，返回的对象还可以包括一个 `initialize` 函数，用来改变私有属性的初始值。装饰器也可以不返回值，如果返回的是其他类型的值，或者包含其他属性的对象，就会报错。

11. addInitializer() 方法（新语法）

除了属性装饰器，其他装饰器的上下文对象还包括一个 `addInitializer()` 方法，用来完成初始化操作。

它的运行时间如下。

- 类装饰器：在类被完全定义之后。
- 方法装饰器：在类构造期间运行，在属性初始化之前。
- 静态方法装饰器：在类定义期间运行，早于静态属性定义，但晚于类方法的定义。

下面是一个例子。

```
function customElement(name) {
  return (value, { addInitializer }) => {
    addInitializer(function() {
      customElements.define(name, this);
    });
  }
}

@customElement('my-element')
class MyElement extends HTMLElement {
  static get observedAttributes() {
    return ['some', 'attrs'];
  }
}
```

上面的代码等同于下面不使用装饰器的代码。

```
class MyElement {
  static get observedAttributes() {
    return ['some', 'attrs'];
  }
}

let initializersForMyElement = [];

MyElement = customElement('my-element')(MyElement, {
  kind: "class",
  name: "MyElement",
  addInitializer(fn) {
    initializersForMyElement.push(fn);
  },
}) ?? MyElement;

for (let initializer of initializersForMyElement) {
  initializer.call(MyElement);
}
```

下面是方法装饰器的例子。

```
function bound(value, { name, addInitializer }) {
  addInitializer(function () {
    this[name] = this[name].bind(this);
  });
}

class C {
  message = "hello!";
```

```
  @bound
  m() {
    console.log(this.message);
  }
}

let { m } = new C();

m(); // hello!
```

上面的代码等同于下面不使用装饰器的代码。

```
class C {
  constructor() {
    for (let initializer of initializersForM) {
      initializer.call(this);
    }

    this.message = "hello!";
  }

  m() {}
}

let initializersForM = []

C.prototype.m = bound(
  C.prototype.m,
  {
    kind: "method",
    name: "m",
    static: false,
    private: false,
    addInitializer(fn) {
      initializersForM.push(fn);
    },
  }
) ?? C.prototype.m;
```

---

12. core-decorators.js

core-decorators.js是一个第三方模块，提供了几个常见的装饰器，通过它可以更好地理解装饰器。

(1) @autobind

`autobind` 装饰器使得方法中的 `this` 对象，绑定原始对象。

```
import { autobind } from 'core-decorators';

class Person {
  @autobind
  getPerson() {
    return this;
  }
}

let person = new Person();
let getPerson = person.getPerson;
```

```
getPerson() === person;
// true
```

(2) @readonly

`readonly` 装饰器使得属性或方法不可写。

```
import { readonly } from 'core-decorators';

class Meal {
  @readonly
  entree = 'steak';
}

var dinner = new Meal();
dinner.entree = 'salmon';
// Cannot assign to read only property 'entree' of [object Object]
```

(3) @override

`override` 装饰器检查子类的方法，是否正确覆盖了父类的同名方法，如果不正确会报错。

```
import { override } from 'core-decorators';

class Parent {
  speak(first, second) {}
}

class Child extends Parent {
  @override
  speak() {}
  // SyntaxError: Child#speak() does not properly override Parent#speak(first, second)
}

// or

class Child extends Parent {
  @override
  speaks() {}
  // SyntaxError: No descriptor matching Child#speaks() was found on the prototype chain.
  //
  //   Did you mean "speak"?
}
```

(4) @deprecate (别名@deprecated)

`deprecate` 或 `deprecated` 装饰器在控制台显示一条警告，表示该方法将废除。

```
import { deprecate } from 'core-decorators';

class Person {
  @deprecate
  facepalm() {}

  @deprecate('We stopped facepalming')
  facepalmHard() {}

  @deprecate('We stopped facepalming', { url: 'http://knowyourmeme.com/memes/facepalm' })
  facepalmHarder() {}
}

let person = new Person();
```

```
person.facepalm();
// DEPRECATION Person#facepalm: This function will be removed in future versions.

person.facepalmHard();
// DEPRECATION Person#facepalmHard: We stopped facepalming

person.facepalmHarder();
// DEPRECATION Person#facepalmHarder: We stopped facepalming
//
//     See http://knowyourmeme.com/memes/facepalm for more details.
//
```

(5) @suppressWarnings

`suppressWarnings` 装饰器抑制 `deprecated` 装饰器导致的 `console.warn()` 调用。但是，异步代码发出的调用除外。

```
import { suppressWarnings } from 'core-decorators';

class Person {
  @deprecated
  facepalm() {}

  @suppressWarnings
  facepalmWithoutWarning() {
    this.facepalm();
  }
}

let person = new Person();

person.facepalmWithoutWarning();
// no warning is logged
```

---

13. 使用装饰器实现自动发布事件

我们可以使用装饰器，使得对象的方法被调用时，自动发出一个事件。

```
const postal = require("postal/lib/postal.lodash");

export default function publish(topic, channel) {
  const channelName = channel || '/';
  const msgChannel = postal.channel(channelName);
  msgChannel.subscribe(topic, v => {
    console.log('频道: ', channelName);
    console.log('事件: ', topic);
    console.log('数据: ', v);
  });

  return function(target, name, descriptor) {
    const fn = descriptor.value;

    descriptor.value = function() {
      let value = fn.apply(this, arguments);
      msgChannel.publish(topic, value);
    };
  };
}
```

上面代码定义了一个名为 `publish` 的装饰器，它通过改写 `descriptor.value` ，使得原方法被调用时，会自动发出一个事件。它使用的事件“发布/订阅”库是Postal.js。

它的用法如下。

```
// index.js
import publish from './publish';

class FooComponent {
  @publish('foo.some.message', 'component')
  someMethod() {
    return { my: 'data' };
  }
  @publish('foo.some.other')
  anotherMethod() {
    // ...
  }
}

let foo = new FooComponent();

foo.someMethod();
foo.anotherMethod();
```

以后，只要调用 `someMethod` 或者 `anotherMethod`，就会自动发出一个事件。

```
$ bash-node index.js
频道:  component
事件:  foo.some.message
数据:  { my: 'data' }

频道:  /
事件:  foo.some.other
数据:  undefined
```

---

14. Mixin

在装饰器的基础上，可以实现 `Mixin` 模式。所谓 `Mixin` 模式，就是对象继承的一种替代方案，中文译为“混入”（mix in），意为在一个对象之中混入另外一个对象的方法。

请看下面的例子。

```
const Foo = {
  foo() { console.log('foo') }
};

class MyClass {}

Object.assign(MyClass.prototype, Foo);

let obj = new MyClass();
obj.foo() // 'foo'
```

上面代码之中，对象 `Foo` 有一个 `foo` 方法，通过 `Object.assign` 方法，可以将 `foo` 方法“混入” `MyClass` 类，导致 `MyClass` 的实例 `obj` 对象都具有 `foo` 方法。这就是“混入”模式的一个简单实现。

下面，我们部署一个通用脚本 `mixins.js`，将 Mixin 写成一个装饰器。

```
export function mixins(...list) {
  return function (target) {
```

```
    Object.assign(target.prototype, ...list);
  };
}
```

然后，就可以使用上面这个装饰器，为类"混入"各种方法。

```
import { mixins } from './mixins.js';

const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();
obj.foo() // "foo"
```

通过 `mixins` 这个装饰器，实现了在 `MyClass` 类上面"混入" `Foo` 对象的 `foo` 方法。

不过，上面的方法会改写 `MyClass` 类的 `prototype` 对象，如果不喜欢这一点，也可以通过类的继承实现 Mixin。

```
class MyClass extends MyBaseClass {
  /* ... */
}
```

上面代码中， `MyClass` 继承了 `MyBaseClass` 。如果我们想在 `MyClass` 里面"混入"一个 `foo` 方法，一个办法是在 `MyClass` 和 `MyBaseClass` 之间插入一个混入类，这个类具有 `foo` 方法，并且继承了 `MyBaseClass` 的所有方法，然后 `MyClass` 再继承这个类。

```
let MyMixin = (superclass) => class extends superclass {
  foo() {
    console.log('foo from MyMixin');
  }
};
```

上面代码中， `MyMixin` 是一个混入类生成器，接受 `superclass` 作为参数，然后返回一个继承 `superclass` 的子类，该子类包含一个 `foo` 方法。

接着，目标类再去继承这个混入类，就达到了"混入" `foo` 方法的目的。

```
class MyClass extends MyMixin(MyBaseClass) {
  /* ... */
}

let c = new MyClass();
c.foo(); // "foo from MyMixin"
```

如果需要"混入"多个方法，就生成多个混入类。

```
class MyClass extends Mixin1(Mixin2(MyBaseClass)) {
  /* ... */
}
```

这种写法的一个好处，是可以调用 `super` ，因此可以避免在"混入"过程中覆盖父类的同名方法。

```
let Mixin1 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin1');
```

```
    if (super.foo) super.foo();
  }
};

let Mixin2 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin2');
    if (super.foo) super.foo();
  }
};

class S {
  foo() {
    console.log('foo from S');
  }
}

class C extends Mixin1(Mixin2(S)) {
  foo() {
    console.log('foo from C');
    super.foo();
  }
}
```

上面代码中，每一次 混入 发生时，都调用了父类的 `super.foo` 方法，导致父类的同名方法没有被覆盖，行为被保留了下来。

```
new C().foo()
// foo from C
// foo from Mixin1
// foo from Mixin2
// foo from S
```

---

15. Trait

Trait 也是一种装饰器，效果与 Mixin 类似，但是提供更多功能，比如防止同名方法的冲突、排除混入某些方法、为混入的方法起别名等等。

下面采用traits-decorator这个第三方模块作为例子。这个模块提供的 `traits` 装饰器，不仅可以接受对象，还可以接受 ES6 类作为参数。

```
import { traits } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') }
};

@traits(TFoo, TBar)
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
```

上面代码中，通过 `traits` 装饰器，在 `MyClass` 类上面“混入”了 `TFoo` 类的 `foo` 方法和 `TBar` 对象的 `bar` 方法。

Trait 不允许“混入”同名方法。

```
import { traits } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar)
class MyClass { }
// 报错
// throw new Error('Method named: ' + methodName + ' is defined twice.');
//       ^
// Error: Method named: foo is defined twice.
```

上面代码中，`TFoo` 和 `TBar` 都有 `foo` 方法，结果 `traits` 装饰器报错。

一种解决方法是排除 `TBar` 的 `foo` 方法。

```
import { traits, excludes } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar::excludes('foo'))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
```

上面代码使用绑定运算符（`::`）在 `TBar` 上排除 `foo` 方法，混入时就不会报错了。

另一种方法是为 `TBar` 的 `foo` 方法起一个别名。

```
import { traits, alias } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar::alias({foo: 'aliasFoo'}))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
```

```
obj.aliasFoo() // foo
obj.bar() // bar
```

上面代码为 `TBar` 的 `foo` 方法起了别名 `aliasFoo`，于是 `MyClass` 也可以混入 `TBar` 的 `foo` 方法了。

`alias` 和 `excludes` 方法，可以结合起来使用。

```
@traits(TExample::excludes('foo','bar')::alias({baz:'exampleBaz'}))
class MyClass {}
```

上面代码排除了 `TExample` 的 `foo` 方法和 `bar` 方法，为 `baz` 方法起了别名 `exampleBaz`。

`as` 方法则为上面的代码提供了另一种写法。

```
@traits(TExample::as({excludes:['foo', 'bar'], alias: {baz: 'exampleBaz'}}))
class MyClass {}
```

留言