

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

Q

目录

- 0. 前言
- 1. ECMAScript 6简介
- 2. let和const命令
- 3. 变量的解构赋值
- 4. 字符串的扩展
- 5. 正则的扩展
- 6. 数值的扩展
- 7. 数组的扩展
- 8. 函数的扩展
- 9. 对象的扩展
- 10. Symbol
- 11. Proxy和Reflect
- 12. 二进制数组
- 13. Set和Map数据结构
- 14. Iterator和for...of循环
- 15. Generator函数
- 16. Promise对象
- 17. 异步操作和Async函数
- 18. Class
- 19. Decorator
- 20. Module

修饰器

- 1. 类的修饰
- 2. 方法的修饰
- 3. 为什么修饰器不能用于函数？
- 4. core-decorators.js
- 5. 使用修饰器实现自动发布事件
- 6. Mixin
- 7. Trait
- 8. Babel转码器的支持

1. 类的修饰

修饰器（Decorator）是一个函数，用来修改类的行为。这是ES7的一个提案，目前Babel转码器已经支持。

修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，修饰器能在编译阶段运行代码。

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

@testable
```

- 21. 编程风格
- 22. 读懂规格
- 23. 参考链接

其他

- 源码
- 修订历史
- 反馈意见

```
class MyTestableClass {}

console.log(MyTestableClass.isTestable) // true
```

上面代码中，`@testable` 就是一个修饰器。它修改了 `MyTestableClass` 这个类的行为，为它加上了静态属性 `isTestable`。

基本上，修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，修饰器本质就是编译时执行的函数。

修饰器函数的第一个参数，就是所要修饰的目标类。

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

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

如果觉得一个参数不够用，可以在修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，修饰器 `testable` 可以接受参数，这就等于可以修改修饰器的行为。

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，修饰器函数 `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'

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

@mixins(Foo)
class MyClass {}

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

上面代码通过修饰器 `mixins`，把 `Foo` 类的方法添加到了 `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'
```

2. 方法的修饰

修饰器不仅可以修饰类，还可以修饰类的属性。

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

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

此时，修饰器函数一共可以接受三个参数，第一个参数是所要修饰的目标对象，第二个参数是所要修饰的属性名，第三个参数是该属性的描述对象。

```
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);
```

上面代码说明，修饰器（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` 修饰器，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(null, arguments);
    };

    return descriptor;
}

const math = new Math();

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

```

上面代码中，`@log` 修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。

修饰器有注释的作用。

```

@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。

如果同一个方法有多个修饰器，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，外层修饰器 `@dec(1)` 先进入，但是内层修饰器 `@dec(2)` 先执行。

除了注释，修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。

3. 为什么修饰器不能用于函数？

修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，使得修饰器不能用于函数。类是不会提升的，所以就没有这方面的问题。

4. core-decorators.js

[core-decorators.js](#)是一个第三方模块，提供了几个常见的修饰器，通过它可以更好地理解修饰器。

(1) @autobind

`autobind`修饰器使得方法中的 `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` 修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` 修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
}

// or

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

(4) @deprecated (别名 @deprecated)

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

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

class Person {
  @deprecated
  facepalm() {}

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

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

let person = new Person();

person.facepalm();
```

```
// DEPRECATION Person#facepalm: This function will be removed in future

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` 修饰器抑制 `decorated` 修饰器导致的 `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
```

5. 使用修饰器实现自动发布事件

我们可以使用修饰器，使得对象的方法被调用时，自动发出一个事件。

```
import postal from "postal/lib/postal.lodash";

export default function publish(topic, channel) {
  return function(target, name, descriptor) {
    const fn = descriptor.value;

    descriptor.value = function() {
      let value = fn.apply(this, arguments);
      postal.channel(channel || target.channel || "/").publish(topic, value);
    };
  };
}
```

上面代码定义了一个名为 `publish` 的修饰器，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。

```
import publish from "path/to/decorators/publish";

class FooComponent {
  @publish("foo.some.message", "component")
  someMethod() {
    return {
      my: "data"
    };
  }
}
```

```

    };
  }
  @publish("foo.some.other")
  anotherMethod() {
    // ...
  }
}

```

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

```

let foo = new FooComponent();

foo.someMethod() // 在"component"频道发布"foo.some.message"事件，附带的数据为
foo.anotherMethod() // 在"/"频道发布"foo.some.other"事件，不附带数据

```

6. Mixin

在修饰器的基础上，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` 写成一个修饰器。

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

然后，就可以使用上面这个修饰器，为类“混入”各种方法。

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

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

@mixin(Foo)
class MyClass {}

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


通过mixins这个修饰器，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
```

7. Trait

Trait也是一种修饰器，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` 修饰器，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修饰器，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修饰器报错。

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 {}

```

8. Babel转码器的支持

目前，Babel转码器已经支持Decorator。

首先，安装 `babel-core` 和 `babel-plugin-transform-decorators`。由于后者包括在 `babel-preset-stage-0` 之中，所以改为安装 `babel-preset-stage-0` 亦可。

```
$ npm install babel-core babel-plugin-transform-decorators
```

然后，设置配置文件 `.babelrc`。

```
{
  "plugins": ["transform-decorators"]
}
```

这时，Babel就可以对Decorator转码了。

脚本中打开的命令如下。

```
babel.transform("code", {plugins: ["transform-decorators"]})
```

Babel的官方网站提供一个[在线转码器](#)，只要勾选Experimental，就能支持Decorator的在线转码。

留言

上一章

下一章