第二章：TypeScript 进阶

学习目标

- 掌握迭代器与生成器的应用
- 理解模块与命名空间的概念，并熟练应用
- 掌握声明合并的方法
- 了解 JSX 的原理及简单应用
- 理解装修器的作用，并能熟练应用

一、迭代器与生成器

1.迭代器与生成器介绍

- 迭代器是一种特殊对象，有一个next方法，每次调用next方法，都会返回一个对象，该对象包含两个属性，一个是value, 表示返回的值。另一个是done，是一个布尔值，用来表示该迭代器是否还有数据可以返回。
- 当一个对象实现了 Symbol.iterator 方法时，我们认为它是可迭代的。
- 一些内置的类型，如 Array，Map，Set，String，Int32Array，Uint32Array等都已经实现了各自的 Symbol.iterator。
- 对象上的 Symbol.iterator 函数负责返回供迭代的值，即迭代器。
- 生成器是一种特殊的函数，Symbol.iterator 函数就是一个生成器函数，用于创建迭代器。

2.迭代器与生成器实例

for..of 语句：

```
let someArray: [number, string, boolean] = [1, "string", false];
for (let entry of someArray) {
    console.log(entry); // =>  1, "string", false
}
```

for..of 与for..in 语句：

```
let list = [7, 8, 9];
for (let i in list) {
    console.log(i); // "0", "1", "2",
}
for (let i of list) {
    console.log(i); // "7", "8", "9"
}
```

for..of 与 for..in 对比实例：

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";
// for..in 可用于查看任何对象的属性
for (let pet in pets) {
    console.log(pet); // "species"
}
// for..of 关注对象的键对应的值
for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

生成代码实例：生成 ES3 或 ES5 时

```
// ts 代码
let numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}
// 生成的js代码
var numbers1 = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
    var num = numbers[_i];
    console.log(num);
}
```

生成器函数实例：

```
// 生成器函数
function * foo(x) {
    while(x >= 0){
        yield x;
        x--;
    }
    return;
}
var it=foo(3);
console.log(it.next());//{ value: 3, done: false }
console.log(it.next());//{ value: 2, done: false }
console.log(it.next());//{ value: 1, done: false }
console.log(it.next());//{ value: 0, done: false }
console.log(it.next());//{ value: undefined, done: true }
```

自定义可迭代类型实例：

```
// 自定义迭代器实例
class SortedArray {
    *[Symbol.iterator]() {
        yield 11;
        yield 12;
        yield 13;
    }
}
const testingIterables = new SortedArray();
for(let item of testingIterables) {
    console.log(item); // 11, 12, 13
}
```

```
tsc -t es2015  3.2_iterator.ts  // 编译时要指明目标版本
```

二、模块

1.模块的概念与特点

- TypeScript 模块的设计理念是可以更换的组织代码；
- 模块是在其自身的作用域里执行，模块里面的变量、函数和类等在模块外部是不可见的，除非明确地使用 export 导出它们；
- 必须通过 import 导入其他模块导出的变量、函数、类，才能使用；
  两个模块之间的关系是通过在文件级别上使用 import 和 export 建立的；
- 在运行时，模块加载器的作用是在执行前去查找并执行这个模块的所有依赖

2.模块的导出与导入

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加 export 关键字来导出。

```
// 模块的导出
// 导出接口
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
// 导出变量
export const numberRegexp = /^[0-9]+$/;
// 导出函数
export const isEmpty = (str)=> reutrn str.length > 0;
// 导出类
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

可以使用以下 import 形式之一来导入其它模块中的导出内容。

```
// 模块的导入
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator = new ZipCodeValidator();
// 可以对导入内容重命名
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
// 将整个模块导入到一个变量，并通过它来访问模块的导出部分
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
// 具有副作用的导入模块，全局性的
import "./my-module.js";
```

使用 export default 默认导出，可简化使用。

```
// 默认导出类
// ZipCodeValidator.ts
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
// Test.ts
import validator from "./ZipCodeValidator";
let myValidator = new validator();
```

```
// 默认导出函数
// StaticZipCodeValidator.ts
const numberRegexp = /^[0-9]+$/;
export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
// Test.ts
import validate from "./StaticZipCodeValidator";
let strings = ["Hello", "98052", "101"];
// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

```
// 默认导出值
// OneTwoThree.ts
export default "123";
// Log.ts
import num from "./OneTwoThree";
console.log(num); // "123"
```

3.模块导出与导入的特殊语法

为了支持 CommonJS 和 AMD 的 exports, TypeScript 提供了export =语法。
export = 和 import = require() 要对应使用。

```
// export = 语法实例
// ZipCodeValidator.ts
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
// Test.ts
import zip = require("./ZipCodeValidator");
let validator = new zip();
// other code
```

4.编译参数

为了编译，我们必须要在命令行上指定一个模块目标。

对于 Node.js 来说，使用--module commonjs； 对于 Require.js来说，使用 --module amd；

```
// modules 可以是 none,commonjs,amd,system,umd,es2015,ESNext

tsc --module commonjs Test.ts
```

5.创建模块结构指导

- 尽可能地在顶层导出

  （1）如果仅导出单个 class 或 function，使用 export default

  （2）如果要导出多个对象，把它们放在顶层里导出

  （3）明确地列出导出的名字
- 当你要导出大量内容的时候使用命名空间导入模式
- 使用重新导出进行扩展
- 模块里不要使用命名空间

当要导出大量内容时，使用命名空间导入模式：

```
// MyLargeModule.ts
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
// Consumer.ts
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

当要对某个对象进行扩展时，推荐的方案是不要去改变原来的对象，而是导出一个新的实体来提供新的功能。

```
// 使用重新导出进行扩展
import { Calculator } from "./Calculator";
class ProgrammerCalculator extends Calculator {...}
// 导出新实体
export { ProgrammerCalculator as Calculator };
export { test } from "./Calculator";
// 导入使用
import { Calculator, test } from "./ProgrammerCalculator";
let c = new Calculator(2);
test(c, "001+010=");
```

三、命名空间

1.命名空间的概念和作用

命名空间是 TypeScript 中组织代码的一种方式。最明确的目的就是解决重名问题。
命名空间定义了标识符的可见范围，一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是不冲突的，是互不相干的。

```
// 命名空间
namespace MyNameSpace {
    export interface Imyinterface {      }
    export class MyClass {      }
}
// 在同一文件内使用，可直接引用
MyNameSpace.MyClass
// 在其它命名空间中使用，要先引入定义文件再使用
/// <reference path = "MyNameSpace.ts" />
```

2.命名空间应用

可以在多个文件中定义同一个命名空间，定义与使用举例：

```
// 命名空间实例
// A.ts
namespace Helper {
    export class A {
        public help() { console.log("A.help()"); }
    }
}
// B.ts
namespace Helper {
    export class B {
        public help() { console.log("B.help()"); }
    }
}
// Test.ts
/// <reference path = "A.ts" />
/// <reference path = "B.ts" />
new Helper.A().help();
new Helper.B().help();
```

3.嵌套命名空间

命名空间内部是可以嵌套另一个命名空间的，这样可以体现层次关系，定义与使用举例：

```
// 嵌套命名空间
namespace com {
    export namespace app {
       export class Order {
          public getDiscount(price: number) {
             return price * .40;
          }
       }
    }
}
// InvoiceTest.ts 文件代码:
/// <reference path = "com.ts" />
var invoice = new com.app.Order();
console.log(invoice.getDiscount(500));
```

4.命名空间编译

第一种方式：把所有的输入文件编译为一个输出文件，需要使用 -- outFile 标记。

```
tsc --outFile sample.js Test.ts
// 编译器会根据源码里的引用标签自动地对输出进行排序
// 也可以单独地指定每个文件
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts Test.ts
```

第二种方式：可以编译每一个文件（默认方式），每个源文件对应生成一个JavaScript文件；然后把所有生成的JavaScript文件按正确的顺序引进来。

5.命名空间与模块

- 像命名空间一样，模块可以包含代码和声明。
- 不同的是模块可以声明它的依赖。
- 模块也提供了更好的代码重用，更强的封闭性以及更好的使用工具进行优化。
- 从 ES6 开始，模块成为了语言内置的部分，应该会被所有正常的解释引擎所支持。 因此，对于新项目来说推荐使用模块做为组织代码的方式。

四、声明合并

1.声明合并的概念

- “声明合并”是指编译器对同名的多个独立声明合并为单一声明。
- 合并后的声明同时拥有原先多个声明的特性。
- 任何数量的声明都可被合并，没有限制。
- TypeScript 中可以进行声明合并的对象有接口、命名空间、类、函数、枚举等。

2.声明合并的规则

最简单也最常见的声明合并类型是接口合并，合并的机制是把双方的成员放到一个同名的接口里。

```
// 声明合并之接口合并
interface Box {
    height: number;
    width: number;
}
interface Box {
    scale: number;
    width: string;  // 报错，相同名称类型必须相同
}
let box: Box = {height: 5, width: 6, scale: 10};
```

对于接口的函数成员，每个同名函数声明都会被当成这个函数的一个重载。
同时需要注意：当接口 A与后来的接口 A合并时，后面的接口具有更高的优先级

```
// 接口函数成员合并
interface Cloner {
    clone(animal: Animal): Animal;
}
interface Cloner {
    clone(animal: Sheep): Sheep;
}
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
// 合并后结果
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
// 注意每组接口里的声明顺序保持不变，但各组接口之间的顺序是后来的接口重载出现在靠前位置。
```

如果函数声明里有一个参数的类型是单一的字符串字面量（不是字符串字面量的联合类型），那么它将会被提升到重载列表的最顶端。

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
// 合并后的 Document将会像下面这样：
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement; // 特殊
    createElement(tagName: "div"): HTMLDivElement;  // 特殊
    createElement(tagName: "span"): HTMLSpanElement; // 特殊
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

命名空间的声明合并举例：

```
// 声明合并命名空间
namespace Animals {
    export class Elephant { }
}
namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Mouse { }
}
// 合并之后
namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Elephant { }
    export class Mouse { }
}
```

非导出成员仅在其原有的（合并前的）命名空间内可见

```
从其它命名空间合并进来的成员无法访问非导出成员。如下：
namespace Animal {
    let haveMuscles = true;
    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}
namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // Error, 因为 haveMuscles 不能被访问
    }
}
```

声明合并命名空间与类

```
// 声明合并命名空间和类
// 声明合并命名空间和类
class A {
    label: string;
}
namespace A {
    export class B {
        name: string;
        constructor(name: string = 'unknow'){
            this.name = name;
        }
    }
}
console.log(A); // [Function: A] { B: [Function: B] }
let b = new A.B();
console.log(b); // B { name: 'unknow' }
let a = new A();
a.label = "aaaa";
console.log(a); // A { label: 'aaaa' }
```

声明合并命名空间与函数

```
// 合并命名空间和函数
// 函数
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}
// 命名空间
namespace buildLabel {
    export let suffix = " ~";
    export let prefix = "Hello, ";
}
console.log(buildLabel); // [Function: buildLabel] { suffix: ' ~', prefix: 'Hell
o, ' }
console.log(buildLabel.prefix); // Hello,
console.log(buildLabel.suffix); //  ~
console.log(buildLabel("Michael Json")); // Hello, Michael Json ~
```

五、JSX

1.JSX 是什么？

JSX 是一个类 XML 的 JavaScript 语法扩展。

优点：

- JSX 执行更快，因为它在编译为 JavaScript 代码后进行了优化。
- 它是类型安全的，在编译过程中就能发现错误。
- 使用 JSX 编写模板更加简单快速。

JSX是一种嵌入式的类似XML的语法，可以被转换成合法的JavaScript；尽管转换的语义是依据不同的实现而定。

JSX因React框架而流行，但也存在其它的实现；TypeScript 支持内嵌，类型检查以及将 JSX 直接编译为

JavaScript。

```
// jsx 实例
const myStyle = {
  fontSize: 100,
  color: '#FF0000'
};
const element = <h1 style={myStyle}> Hello, world! </h1>;
```

2.JSX 应用

JSX必须做两件事：

- 给文件一个.tsx 扩展名
- 启用 --jsx 选项，TypeScript具有三种JSX模式：preserve，react和react-native

模式	输入	输出	输出文件扩展名
preserve			.jsx
react		React.createElement("div")	.js
react-native			.js

通过在命令行里使用--jsx标记或tsconfig.json里的选项来指定模式；
编译命令： tsc --jsx preserve test.tsx

3.as 操作符

类型断言有两种形式；

一个是“尖括号”语法：

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

另一个是 as 语法：

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

4.类型检查

类型检查，首先要理解固有元素与基于值的元素之间的区别。

- TypeScript使用与React相同的规范来区别它们。
- 固有元素总是以一个小写字母开头，基于值的元素总是以一个大写字母开头。
- 固有元素是环境自带的，比如在DOM环境里的div或span, 而基于值的元素是我们自定义的。

  两者的区别有如下两点：

  （1）对于React，固有元素会生成字符串（React.createElement("div")），然而由你自定义的组件却不会生成（React.createElement(MyComponent)）。

  （2）传入JSX元素里的属性类型的查找方式不同；固有元素属性本身就支持，然而自定义的组件会自己去指定它们具有哪个属性。

固有元素举例：

```
// 固有元素使用特殊的接口JSX.IntrinsicElements来查找。
// 默认地，如果这个接口没有指定，会全部通过，不对固有元素进行类型检查。
declare namespace JSX {
  interface IntrinsicElements {
      foo: any
  }
}
<foo />; // 正确
<bar />; // 错误，未找到
```

基于值的元素举例：

```
// 基于值的元素会简单的在它所在的作用域里按标识符查找。
import MyComponent2 from "./myComponent";

<MyComponent2 />; // 正确
<SomeOtherComponent />; // 错误
```

有两种方式可以定义基于值的元素：

- 无状态函数组件 (SFC)
- 类组件

无状态函数组件举例：

```
// 无状态函数组件
// 无状态函数组件
interface FooProp {
  className: string;
  X: number;
  Y: number;
}
function ComponentFoo(prop: FooProp): JSX.Element {
    return <div className = {prop.className}/>;
}
let mytag = <ComponentFoo className={"title"} X={1} Y={2} />;
```

类组件举例：

```
// 类组件
declare namespace JSX {
  interface ElementClass {
  render: any;
  }
}
class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}
<MyComponent />; // 正确
<MyFactoryFunction />; // 正确
class NotAValidComponent {}  // 缺少 render
function NotAValidFactoryFunction() {
  return {};                 // 缺少 render
}
<NotAValidComponent />; // 错误
<NotAValidFactoryFunction />; // 错误
```

固有元素属性类型检查，需要是 JSX.IntrinsicElements 中定义的属性类型，举例：

```
declare namespace JSX {
    interface IntrinsicElements {
       foo2: { requiredProp: string; optionalProp?: number }
    }
}
<foo2 requiredProp="bar" />; // 正确
<foo2 requiredProp="bar" optionalProp={0} />; // 正确
<foo2 />; // 错误, 缺少 requiredProp
<foo2 requiredProp={0} />; // 错误, requiredProp 应该是字符串
<foo2 requiredProp="bar" unknownProp />; // 错误, unknownProp 不存在
<foo2 requiredProp="bar" some-unknown-prop />; // 正确, `some-unknown-prop`不是个合法的标识符
```

基于值元素属性类型检查，举例：

```
// 基于值的元素，首先检查 JSX.ElementAttributesProperty 中是否有指定，
// 如果未指定则使用元素类型上指定属性
declare namespace JSX {
  interface ElementAttributesProperty {
  props; // 指定用来使用的属性名
  }
}
class MyComponent1 {
  // 在元素实例类型上指定属性
  props: {
    foo?: string;
  }
}
// `MyComponent`的元素属性类型为`{foo?: string}`
<MyComponent1 foo="bar" />
```

六、装饰器

- 装饰器是一种特殊类型的声明，它能够被附加到类的声明，方法， 访问符，属性或参数上；
- 装饰器使用 @expression 这种形式，expression求值后必须为一个函数；
- 它会在运行时被调用，被装饰的声明信息做为参数传入。
- 装饰器执行时机是在代码编译时发生的（不是 TypeScript 编译，而是在 JavaScript 执行编译阶段）

1.装饰器的概念

启用装饰器特性，必须在命令行或 tsconfig.json 里启用 experimentalDecorators 编译器选项：

```
tsc --target ES5 --experimentalDecorators
// tsconfig.json 中开启:
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

2.装饰器语法

装饰器定义与应用举例：

```
// 装饰器定义
function sealed(target) {
    // do something with "target" ...
}
// 或 使用装饰器工厂函数
function color(value: string) { // 这是一个装饰器工厂
    return function (target) { //  这是装饰器
        // do something with "target" and "value"...
    }
}
// 应用到一个声明上，例如：
@sealed @color let x: string
// 或
@sealed
@color
let x: string
```

在 TypeScript 里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：

- 由上至下依次对装饰器表达式求值。
- 求值的结果会被当作函数，由下至上依次调用。

多个装饰器应用在一个声明，举例：

```
// 多个装饰器应用在一个声明
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string) {
        console.log("f(): called");
    }
}
function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string) {
        console.log("g(): called");
    }
}
class C {
    @f() @g() method() {}
}
// f(): evaluated
// g(): evaluated
// g(): called
// f(): called
```

3.类装饰器

类中不同声明上的装饰器将按以下规定的顺序应用：

- 有多个参数装饰器时：从最后一个参数依次向前执行。
- 方法和方法参数中,参数装饰器先执行。
- 方法和属性装饰器，谁在前面谁先执行。因为参数属于方法一部分，所以参数会挨着方法执行。

- 类装饰器总是最后执行。

类装饰器在类声明之前被声明（紧靠着类声明）。

类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。

```
// 类装饰符实例
function Path(path: string) {
    return function (target: Function) {
        !target.prototype.$Meta && (target.prototype.$Meta = {})
        target.prototype.$Meta.baseUrl = path;
    };
}

@Path('/hello')
class HelloService {
    [x: string]: any;
    constructor() {}
}

console.log(HelloService.prototype.$Meta);// 输出：{ baseUrl: '/hello' }
let hello = new HelloService();
console.log(hello.$Meta) // 输出：{ baseUrl: '/hello' }
```

4.方法装饰器

方法装饰器声明在一个方法的声明之前，可以用来监视，修改或者替换方法定义。

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

- 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象；
- 成员的名字；
- 成员的属性描述符；

  注意：如果代码输出目标版本小于ES5，属性描述符将会是 undefined，返回值会被忽略。
  如果方法装饰器返回一个值，它会被用作方法的属性描述符。

方法装饰器，举例：

```
// 方法装饰符实例
function GET(alias: string) {
    return function (target, methodName: string, descriptor: PropertyDescriptor)
 {
        target._alias = alias;
    }
}

class HelloService {
    constructor() { }
    @GET("getMyName")
    getUser() { }
}
let hello = new HelloService();
console.log(hello._alias);         // gtMyName
```

5.访问装饰器

访问器装饰器声明在一个访问器的声明之前，可以用来监视，修改或替换一个访问器的定义。

访问装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

- 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象；
- 成员的名字；
- 成员的属性描述符；

  注意：如果代码输出目标版本小于ES5，属性描述符将会是 undefined，返回值会被忽略。
  如果访问装饰器返回一个值，它会被用作方法的属性描述符。

  TypeScript 不允许同时装饰一个成员的get和set访问器；

  一个成员的所有装饰的必须应用在文档顺序的第一个访问器上；因为在装饰器应用于一个属性描述符时，它联

  合了get和set访问器，而不是分开声明的。

访问装饰器，举例：

```
// 访问装饰器实例
function access(value: string) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescr
iptor) {
        !target.$Meta && (target.$Meta = {})
        target.$Meta[propertyKey] =  `It's a ${value} method`
    };
}
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }
    @access('get')
    get x() { return this._x; }
    @access('set')
    set y(y: number) { this._y = y; }
}
let point = new Point(1,2);
console.log(point['$Meta']); //{ x: "It's a get method", y: "It's a set method"
}
```

6.属性装饰器

属性装饰器声明在一个属性声明之前。

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

- 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
- 成员的名字。

  注意：

  属性描述符不会做为参数传入属性装饰器，因为目前无法在定义一个原型对象的成员时描述一个实例属性，且

  无法监视或修改一个属性的初始化方法，因此，属性描述符只能用来监视类中是否声明了某个名字的属性。

属性装饰器，举例：

```
// 属性装饰符实例
function DefaultValue(value: string) {
    return function (target: any, propertyName: string) {
        target[propertyName] = value;
    }
}

class Hello {
    @DefaultValue("world") greeting: string;
}

console.log(new Hello().greeting); // world
```

7.参数装饰器

参数装饰器声明在一个参数声明之前，参数装饰器应用于类构造函数或方法声明。

参数装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

- 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象；
- 成员的名字；
- 参数在函数参数列表中的索引；

  注意：

  参数装饰器只能用来监视一个方法的参数是否被传入。
  参数装饰器的返回值会被忽略。

参数装饰器，举例：

```
// 参数装饰器实例
function PathParam(paramName: string) {
    return function (target, methodName: string, paramIndex: number) {
        !target.$Meta && (target.$Meta = {});
        target.$Meta[paramIndex] = paramName;
    }
}

class HelloService {
    constructor() { }
    getUser( @PathParam("the user's id") userId: string) { }
}

console.log((<any>HelloService).prototype.$Meta); // { '0': "the user's id" }
```

章总结

重难点

- 模块【重点】
- 声明合并【重点】【易错点】
- 迭代器与生成器【重点】【易错点】
- JSX【重点】【易错点】

- 修饰器【难点】【易错点】

拓展延伸

思考：如何应用本章中学习的 TypeScript 高级概念和特性优化程序设计，使其更规范和更严谨？

延伸资料：

- TypeScript 官网文档 http://www.typescriptlang.org/docs/home.html
- TypeScript 中文文档 https://www.tslang.cn/docs/home.html