<https://blog.csdn.net/zhangda89/article/details/82710580>

# [http://es6.ruanyifeng.com/#docs/module#export](http://es6.ruanyifeng.com/#docs/module)

# [ECMAScript 6 入门](http://es6.ruanyifeng.com/#docs/destructuring)

## <http://es6.ruanyifeng.com/#docs/destructuring>

## **1、TypeScript是干什么的 ?**

* TypeScript是由微软Anders Hejlsberg(安德斯·海尔斯伯格，也是本人的偶像)领衔开发的。  
  (安德斯·海尔斯伯格是Delphi 和 C#之父，Turbo Pascal 编译器的主要作者，.NET 概念发起人之一，同时也是TypeScript开源项目的重要领导人。他于1996年加入微软，目前是 C# 语言的首席架构师和 TypeScript 的核心开发者，获微软卓越工程师 Distinguished Engineer 和微软技术院士 Technical Fellow 称号。)

TypeScript 是微软开发的 JavaScript 的超集，TypeScript兼容JavaScript，可以载入JavaScript代码然后运行。TypeScript与JavaScript相比进步的地方 包括：加入注释，让编译器理解所支持的对象和函数，编译器会移除注释，不会增加开销；增加一个完整的类结构，使之更象传统的面向对象语言。

## **2、为什么选择TypeScript ?**

* JavaScript 只是一个脚本语言，并非设计用于开发大型 Web 应用，JavaScript 没有提供类和模块的概念，而 TypeScript 扩展了 JavaScript 实现了这些特性。
* TypeScript 主要特点包括：  
  TypeScript 是微软推出的开源语言，使用 Apache 授权协议，  
  TypeScript 是 JavaScript 的超集。  
  TypeScript 增加了可选类型、类和模块  
  TypeScript 可编译成可读的、标准的 JavaScript  
  TypeScript 支持开发大规模 JavaScript 应用  
  TypeScript 设计用于开发大型应用，并保证编译后的 JavaScript 代码兼容性  
  TypeScript 扩展了 JavaScript 的语法，因此已有的 JavaScript 代码可直接与 TypeScript 一起运行无需更改  
  TypeScript 文件扩展名是 ts，而 TypeScript 编译器会编译成 js 文件  
  TypeScript 语法与 JScript .NET 相同  
    
  TypeScript 易学易于理解
* 网上流传一句话，Angular 2选择了TypeScript作为主语言。如果你是个C#程序员，一定会对它的语法感觉似曾相识。没错，TypeScript和C#、Delphi有同一个“爹” —— 传奇人物Anders Hejlsberg。即使是Java程序员，也不会觉得陌生：强类型、类、接口、注解等等，无一不是后端程序员们耳熟能详的概念。说到底，并没有什么前端语言和后端语言，在语言领域耕耘多年的Anders太熟悉优秀语言的共性了，他所做的取舍值得你信赖。

## **3、TypeScript语法特性**

* 兼容 ECMAScript 2015（ES6）规范，可选择编译成ES6或ES5规范的JavaScript代码（ECMAScript 3及以上版本）；
* 面向对象，并拥有一些函数式特性；
* 类型语言；
* 实现了注解、泛型等特性；
* 适配大型App构建。
* 类 Classes
* 接口 Interfaces
* 模块 Modules
* 类型注解 Type annotations
* 编译时类型检查 Compile time type checking
* Arrow 函数 (类似 C# 的 Lambda 表达式)

## **4、TypeScrip与JavaScript 的区别**

* TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法，因此现有的 JavaScript 代码可与 TypeScript 一起工作无需任何修改，TypeScript 通过类型注解提供编译时的静态类型检查。TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译。
* 通过编译器把TypeScript程序代码编译生成JavaScript代码。

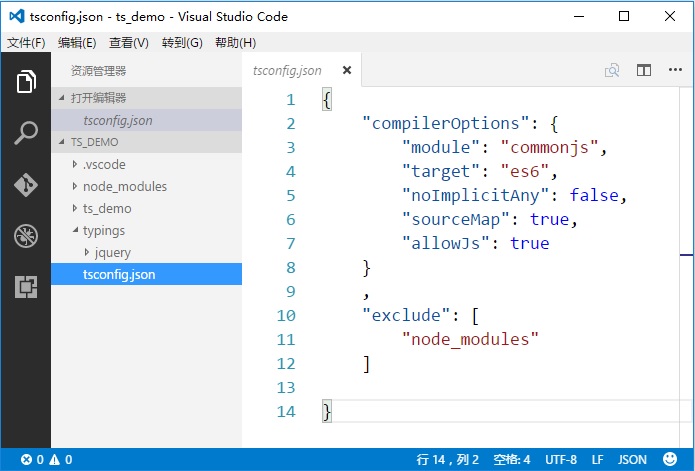
## **12、创建 tsconfig.json**

* (1)、启动VS Code
* (2)、选择菜单 文件/打开文件夹，选择刚刚创建的ts\_demo文件夹
* (3)、双击tsconfig.json打开如下图：



## **13、修改tsconfig.json**

* 把tsconfig.json修改为：



{

"compilerOptions": {

"module": "commonjs",

"target": "es6",

"noImplicitAny": false,

"sourceMap": true,

"allowJs": true

}

,

"exclude": [

"node\_modules"

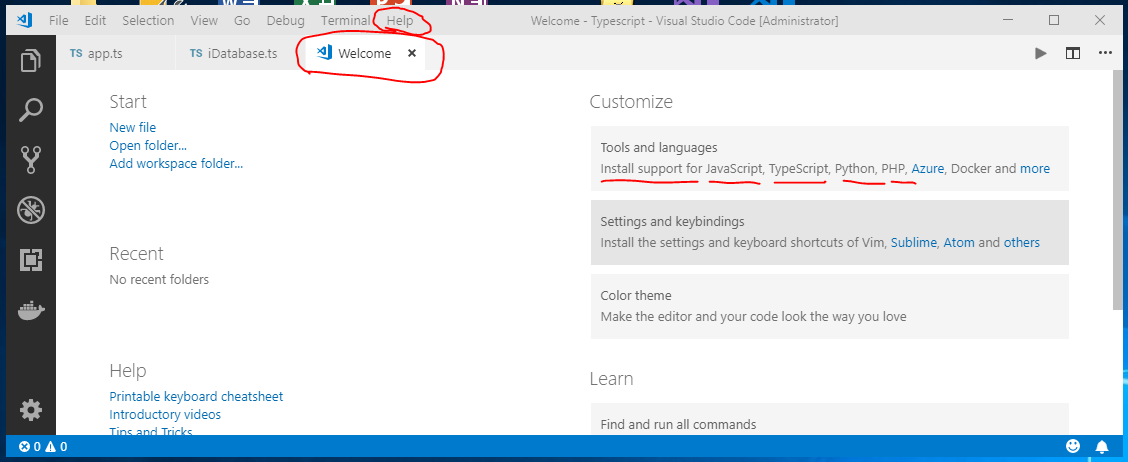
]

}

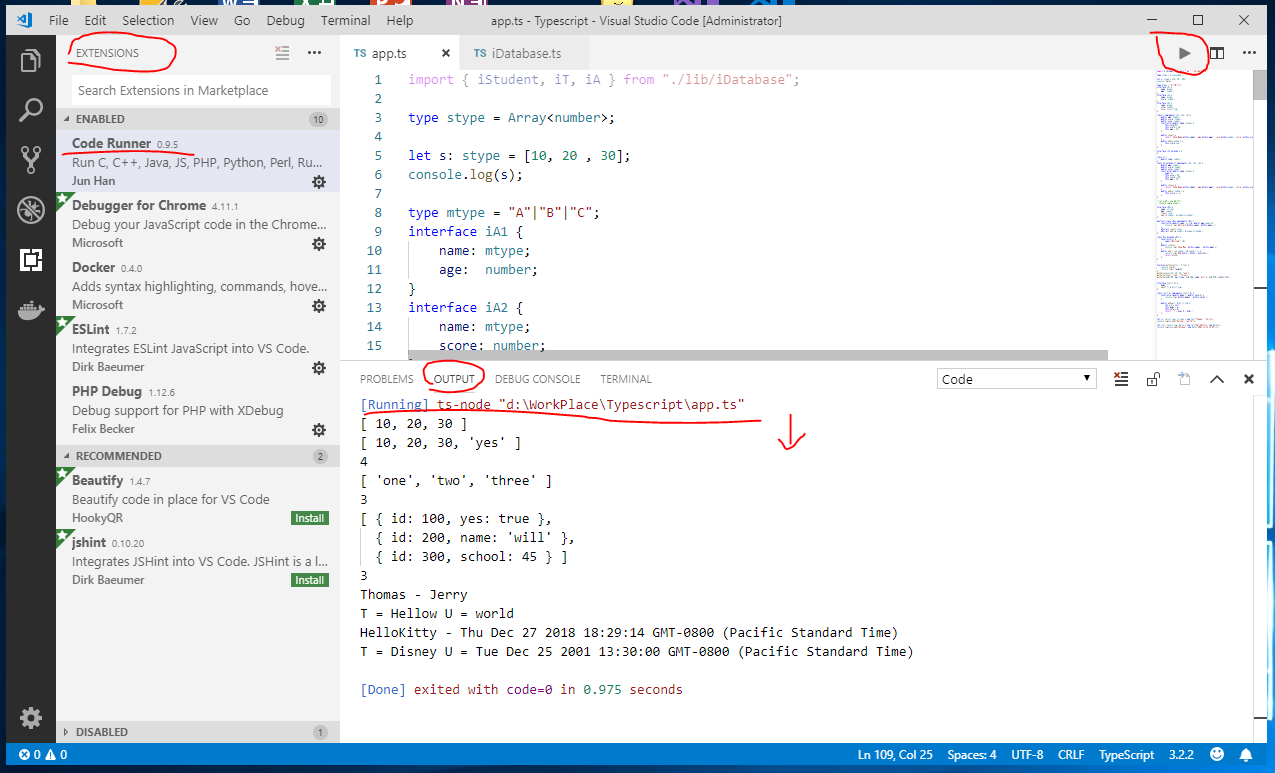
* target：编译之后生成的JavaScript文件需要遵循的标准。有三个候选项：es3、es5、es2015。
* noImplicitAny：为false时，如果编译器无法根据变量的使用来判断类型时，将用any类型代替。为true时，将进行强类型检查，无法推断类型时，提示错误。
* module：遵循的JavaScript模块规范。主要的候选项有：commonjs、AMD和es6。
* removeComments：编译生成的JavaScript文件是否移除注释。
* sourceMap：编译时是否生成对应的source map文件。这个文件主要用于前端调试。当前端js文件被压缩引用后，出错时可借助同名的source map文件查找源文件中错误位置。
* outDir：编译输出JavaScript文件存放的文件夹。
* include、exclude：编译时需要包含/剔除的文件夹。

**Visual Studio Code 的环境配置：**

1. 可以去到欢迎页面，把该支持的脚本语言安装上：



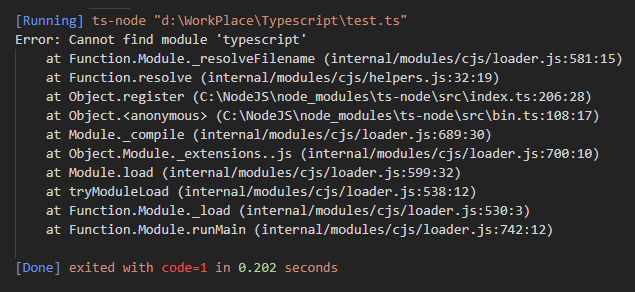
1. 把 “run code”直接运行按钮安装上，这样运行结果可以直接在 output 窗口输出：



TypeScript in Visual Studio Code Environment:

1. npm init - create package.json
2. tsc --init - create tsconfig.json
3. npm install ts-node -g
4. VSCode 安装 Code Runner 可以直接运行各种脚本：包括 typescript, javascript, python 等

如果出现如下错误：



请执行：

在程序目录里执行：npm install typescript -save

**TypeScript基本数据类型**

原创 2017年04月20日 18:34:13

TypeScript基本数据类型

* boolean （布尔）

var value : boolean = true;//声明一个布尔类型变量

* number （数值）

var value = 1; //number类型

var value; //any类型

var value : number = 1; //number类型

* String （字符串）

var name : string = "jack"//声明一个字符串变量

* Array （数组）

var arrayValue : string[] = ["jack","rose","nick"];//声明一个字符串数组

var arrayValue2 : Array<string> = ["jack","rose","nick"];//利用泛型声明一个字符串数组

* ReadonlyArray （数组）

let ra:ReadonlyArray<number> =[1,2,3,4,5];

ra = [4,5,6]; - 可以整体重新赋值， 但是不能有其他删除，添加的操作

console.log(ra);

* Tuple（元祖）

var arrayValue : [string,number] = ["jack",2,3,4,5,"rose"];//声明一个元祖类型

var arrayValue : [string,number] = [2,3,4,5,"rose"];//会报错

* enum（枚举）

enum Color {Red,Green,Blue};//从左往右值为0，1，2

var c = Color.Green; //c值为1

enum color {Red=30, Yellow, Pink=40, Blue};

let n:color = color.Yellow;

let m:color = 31;

console.log(m);

if( n as color == m as color)

    console.log("==");

else

    console.log("<>");

注意：

1. enum 赋值时可以赋予整型数值
2. enum 变量的必须通过转化才可以否则出错

if( n as color == color.Pink)

另外一个有趣的问题：js enum 不象 c# 的enum , 可以不一定是数值类型，可以是字符串或其他，

enum color {Red="Rd", Yellow="Ye", Pink="Pk", Blue="Blu"};

console.log(color);

编译后

{ Red: 'Rd', Yellow: 'Ye', Pink: 'Pk', Blue: 'Blu' }

enum color {Red="Rd", Yellow="Ye", Pink="Pk", Blue="Blu"};

console.log(color);

let n:string = color["Yellow"];

console.log(n);

if(n as color == color.Yellow)

    console.log("it is yellow");

else

    console.log("it is not");

输出结果是：

{ Red: 'Rd', Yellow: 'Ye', Pink: 'Pk', Blue: 'Blu' }

Ye

it is yellow

let n:string = color["Ye"];

console.log(n); - undefined

enum category { Food="Fd", Car="Cr", Boat="Bt" };

let n:category = category.Food;

let m:category = category.Boat;

console.log(<string>n);

enum category { Food=100, Car=Food \* 3, Boat = Car \* 3 };

let n:category = category.Food;

let m:category = category.Boat;

console.log(<number>m); -- 900

* any

var value : any = 1; //类似java的Object类型

* void （与any相对）

function warnUser(): void {

alert("This is my warning message");

}

Never （never类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型）

function error(message: string): never {

throw new Error(message); //抛出异常

}

TypeScript允许声明联合类型

var arrayValue : string[]|number;

arrayValue = ["jack","rose","nick"];

arrayValue = 1;

### 联合类型

TS允许声明联合类型：

var myType:string[]|string|number|boolean;

//myType类型可以是字符串数组、字符串、数字、布尔值

myType="type";

myType=["a","b","c"];

myType=100;

myType=false;

//不可以是其他类型

myType=function(){};//报错

type mtype = "A"|"B"|"C"; - 有点相当于enum, 但是稍微有些区别

let m: mtype = "A";

m = "B";

if(m=="B") { - m==“C” 则编译出错

console.log("m == B");

}

let n: 200|300|400 = 300;

if(<200|300|400>n==300)

    console.log(300);

else

    console.log("NaN");

比较时，需要强制转换类型

TypeScript允许使用type关键字声明类型别名（与c语言类似）

type MyNumber = number;

TS允许使用type关键字声明类型别名：

type PrimitiveArray=Array<string|number|boolean>

type MyNumber=number;

type Callback=()=>void;

type ms = string;

let ss:ms = "Hello World";

let ss1: string = "Hello World";

if(ss==ss1)

    console.log("Peaceful World");

else

    console.log("Not Good");

type ms = string[];

let ss:ms = ["Hello World"];

let ss1: Array<string> = ["Hello World"];

if(ss[0]==ss1[0])

    console.log("Peaceful World");

else

    console.log("Not Good");

type ArrStr = Array<string>|Array<boolean>|Array<Date>;

let aa:ArrStr = [new Date()];

let bb:ArrStr = new Array("Good", "Day");

if( <ArrStr>aa===<ArrStr>bb ) {

    console.log("dkdkd");

}

### 类型守护

类型守护是一种错误提示机制。

JavaScript 一个常用的方式就是使用 typeof 或者 instanceof 来在运行时检查一个表达式的类型。TypeScript 现在可在 if 区域块中理解这种情况。

例如：

var x:any={};

if(typeof x==="string"){

console.log(x.splice(3,1));

}

let s1:any = 2333;

if( typeof(s1) == "string" ) {

console.log(s1.slice(3, 10));

}

if(typeof(s1)=="number") {

console.log(s1-100);

}

判断: object instanceof class

class A1 implements iA1 {

constructor(public name:mtype, public age:number, public memo:string) {

}

}

class A2 implements iA2 {

constructor(public name:mtype, public score:number) {

}

}

let a1 = new A1("A", 36, "Good Man");

console.log(a1 instanceof A1); - true

console.log(a1 instanceof A2); - false

**Basic Types**

**Introduction**

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type thrown in to help things along.

**Boolean**

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

let isDone: boolean = false;

**Number**

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

let decimal: number = 6;

let hex: number = 0xf00d;

let binary: number = 0b1010;

let octal: number = 0o744;

**String**

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

let color: string = "blue";

color = 'red';

You can also use *template strings*, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

let fullName: string = `Bob Bobbington`;

let age: number = 37;

let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;

This is equivalent to declaring sentence like so:

let sentence: string = "Hello, my name is " + fullName + ".\n\n" +

"I'll be " + (age + 1) + " years old next month.";

**Array**

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

let list: number[] = [1, 2, 3];

The second way uses a generic array type, Array<elemType>:

let list: Array<number> = [1, 2, 3];

**Tuple**

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

// Declare a tuple type

let x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

When accessing an element with a known index, the correct type is retrieved:

console.log(x[0].substr(1)); // OK

console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

When accessing an element outside the set of known indices, a union type is used instead:

x[3] = "world"; // OK, 'string' can be assigned to 'string | number'

console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'

Union types are an advanced topic that we’ll cover in a later chapter.

**Enum**

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

enum Color {Red, Green, Blue}

let c: Color = Color.Green;

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

enum Color {Red = 1, Green, Blue}

let c: Color = Color.Green;

Or, even manually set all the values in the enum:

enum Color {Red = 1, Green = 2, Blue = 4}

let c: Color = Color.Green;

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren’t sure what that mapped to in the Color enum above, we could look up the corresponding name:

enum Color {Red = 1, Green, Blue}

let colorName: string = Color[2];

alert(colorName); // Displays 'Green' as its value is 2 above

**Any**

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type:

let notSure: any = 4;

notSure = "maybe a string instead";

notSure = false; // okay, definitely a boolean

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them - you can’t call arbitrary methods on them, even ones that actually exist:

let notSure: any = 4;

notSure.ifItExists(); // okay, ifItExists might exist at runtime

notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;

prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

let list: any[] = [1, true, "free"];

list[1] = 100;

**Void**

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

function warnUser(): void {

alert("This is my warning message");

}

Declaring variables of type void is not useful because you can only assign undefined or null to them:

let unusable: void = undefined;

只能赋予： null 或者 undefined

**Null and Undefined**

In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they’re not extremely useful on their own:

// Not much else we can assign to these variables!

let u: undefined = undefined;

let n: null = null;

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

However, when using the --strictNullChecks flag, null and undefined are only assignable to void and their respective types. This helps avoid *many* common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type string | null | undefined. Once again, more on union types later on.

As a note: we encourage the use of --strictNullChecks when possible, but for the purposes of this handbook, we will assume it is turned off.

undefined, null 都可以当作类型来使用，它们可以通用， 比较时 = = 是true, = = = 是 false

let a: void = null;

let b: undefined = null;

let c: null = undefined;

console.log(a);

console.log(b);

console.log(c);

**Never**

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, *no* type is a subtype of, or assignable to, never (except never itself). Even any isn’t assignable to never.

Some examples of functions returning never:

// Function returning never must have unreachable end point

function error(message: string): never {

throw new Error(message);

}

// Inferred return type is never

function fail() {

return error("Something failed");

}

// Function returning never must have unreachable end point

function infiniteLoop(): never {

while (true) {

}

}

**Type assertions**

Sometimes you’ll end up in a situation where you’ll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

*Type assertions* are a way to tell the compiler “trust me, I know what I’m doing.” A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the “angle-bracket” syntax:

let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;

And the other is the as-syntax:

let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only as-style assertions are allowed.

**A note about let**

You may’ve noticed that so far, we’ve been using the let keyword instead of JavaScript’s var keyword which you might be more familiar with. The let keyword is actually a newer JavaScript construct that TypeScript makes available. We’ll discuss the details later, but many common problems in JavaScript are alleviated by using let, so you should use it instead of var whenever possible.

# <http://www.cnblogs.com/winfred/p/6368744.html>

# [TypeScript笔记 1--环境搭建](http://www.cnblogs.com/winfred/p/6368744.html)

### TypeScript是什么

Typescript是JavaScript的超集，支持ES6特性并且提供了类型系统，可以编译成Javascript。是微软开发且已经在github上开源。  
ES6（ES2015，ES2016等）虽然已发布，但是很多浏览器厂商对其并不是完全支持,当前我们要想使用最新的ES6特性，必须使用转换工具（如babel），才能最终运行在浏览器上。而TypeScript对ES6新特性的支持使得我们可以直接使用开发，并且不用依赖第三方工具（不是绝对的），再加上静态类型检查等，使得我们可以像写java一样爽。

### 环境搭建

有两种主要的方式获取TypeScript工具。通过npm（Node.js包管理器）和安装TypeScript的Visual Studio插件。作为前端开发人员在这里使用npm这种简单高效方式安装。

* 1 npm安装

npm install -g typescript

* 2 是否安装成功,在命令行中输入以下命令，如果出现版本号表示安装成功

tsc -v



### 入门示例

* 创建文件Person.ts,并输入如下内容

class Person {

name:string;

age:number;

constructor(name:string, age:number) {

this.name = name;

this.age = age;

}

msg() {

return `${this.name} is ${this.age} years old`;

}

}

let user = new Person('Jack', 20);

document.body.innerHTML = user.msg();

* 编译后会在同目录下生成Person.js文件，编译命令如下：

tsc Person.ts

* 编译后的文件Person.js内容如下，其实就是javascript源码。
* var Person = (function () {
* function Person(name, age) {
* this.name = name;
* this.age = age;
* }
* Person.prototype.msg = function () {
* return this.name + " is " + this.age + " years old";
* };
* return Person;
* }());
* var user = new Person('Jack', 20);

document.body.innerHTML = user.msg();

* 新建hello.html文件，输入以下内容：
* <!DOCTYPE html>
* <html>
* <head><title>TypeScript Hello World</title></head>
* <body>
* <script src="Person.js"></script>
* </body>

</html>

* 用浏览器打开hello.html即可看到效果  
  

### 工欲善其事必先利其器--VS Code

VS Code也是微软开发的编辑器，其本身也是使用TypeScript开发的，代码已在[github](https://github.com/Microsoft/vscode)上开源，基于[Electron](http://electron.atom.io/)框架开发，相对于Atom速度快，体验较好，是TypeScript IDE的首选。

其源码编辑器[monaco-editor](https://microsoft.github.io/monaco-editor/)联想功能比较强大,已经单独开源。微信小程序开发工具的中源码编辑器就是使用了monaco-editor。另外其插件系统使得我们可以丰富其功能。

/ / 这段代码很诡异

function Person(age) {

this.age = age

this.growOld = function() {

this.age++;

}

}

var person = new Person(1);

setTimeout(person.growOld,1000);

# setTimeout(function() { console.log(person.age); },2000);

# // 输出的是 1, 而不是想象的 2 should have been 2

interface iA4 {

(x: number, y: number):string;

show: ()=>void;

total: number;

}

let func: iA4 = <iA4>function(a: number, b: number) {

console.log(`Init Total = ${a+b}`);

return `Init Total = ${a+b}`;

}

func.total = 515;

func.show = ()=>{

console.log(`Show Total: ${this.total}`);

}

func(233, 155);

func.show();

console.log(func.total);

//

Init Total = 388

Show Total: undefined - 输出的不是想象中的 515

515

分析： func 并不是一个对象，并不能作为 show 方法的context , 真正的context 是 window

function UselessDelay()

{

this.message='Why was I delayed?';

}

UselessDelay.prototype.start=function()

{

setTimeout(this.show,1000);

}

UselessDelay.prototype.show=function()

{

console.log(this.message);

};

var ud=new UselessDelay();

ud.start();//Actual sshow undefined, Expected : Will show 'Why was I delayed?' after 1 second

如何解决上面的问题： 关键是 setTimeout 的context 是javascript window 对象

对方法使用 bind 来绑定对象

UselessDelay.prototype.start=function()

{

setTimeout(this.show.bind(this),1000);

}

这段代码就可以输出正确

function Person(age) {

this.age = age

this.growOld = function() {

this.age++;

}

}

var person = new Person(1);

setTimeout(person.growOld.bind(person),1000);

setTimeout(function() { console.log(person.age); },2000);

很好理解： setTimeout( method, 1000) 而 person.growOld 只是作为方法传入， 并没有涉及到person 对象并把它作为 context , 所以必须要对方法绑定一个context 对象。

# <http://www.cnblogs.com/winfred/p/6380188.html>

# [TypeScript笔记 2--代码调试](http://www.cnblogs.com/winfred/p/6380188.html)

代码调试（debug）是日常开发中必不可少的手段之一，无法进行代码调试会让我们痛苦不已，本文主要介绍如何调试TypeScript代码

很多刚接触TypeScript的人或许有疑问，我们编写的TypeScript编译成JavaScript，在html中引用的是编译后的JavaScript，这样代码报错的地方就和实际开发ts对应不上，这怎么调试？

我们可以采用Source Map

### Source Map是什么

如今前端技术飞速发展，一不留神就落后了。JS变得越来越复杂（压缩，文件合并，其它语言编译成Js），导致实际运行的代码和开发态的代码不一致，调试困难，于是Source Map出现了。

简单地说Source Map就是一个文件，里面是转换代码的代码位置和转换前的代码位置的映射信息，这样当前程序报错时，我们可以找到开发态代码位置。

### 如何生成Source Map

TypeScript在编译时可以同附带参数生成,命令如下：

tsc --sourcemap Person.ts

执行后，会在同目录下生成名为Person.js.map文件，这就是Source Map文件,内容如下：

{

"version": 3,

"file": "Person.js",

"sourceRoot": "",

"sources": [

"Person.ts"

],

"names": [],

"mappings": "AAAA;IAGI,gBAAY,IAAW,EAAE,GAAU;QAC/B,IAAI,CAAC,IAAI,GAAG,IAAI,CAAC;QACjB,IAAI,CAAC,GAAG,GAAG,GAAG,CAAC;IACnB,CAAC;IAED,oBAAG,GAAH;QACI,MAAM,CAAI,IAAI,CAAC,IAAI,YAAO,IAAI,CAAC,GAAG,eAAY,CAAC;IACnD,CAAC;IACL,aAAC;AAAD,CAAC,AAXD,IAWC;AAED,IAAI,IAAI,GAAG,IAAI,MAAM,CAAC,MAAM,EAAE,EAAE,CAAC,CAAC;AAElC,QAAQ,CAAC,IAAI,CAAC,SAAS,GAAG,IAAI,CAAC,GAAG,EAAE,CAAC"

}

该文件是一个json格式，分别记录编译前文件和编译后文件信息，至于文件内容究竟是何含义，我们就不深究了，有兴趣的可以阅读阮一峰老师的文章[JavaScript Source Map 详解](http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html)

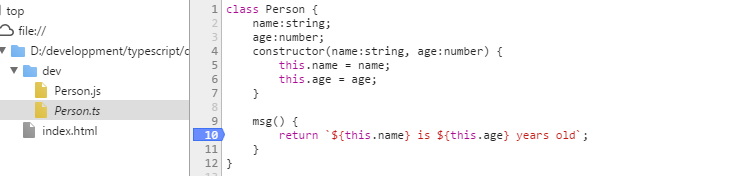
### Source Map如何使用

目前，暂时只有Chrome浏览器支持这个功能。在Developer Tools的Setting设置中，确认选中"Enable source maps"。

同时需要在编译的js代码加上以后信息(编译时已自动加上了)

//# sourceMappingURL=Person.js.map

注意Person.js.map是一个路径，Source Map文件可以是本地文件，也可以是网络文件

这样可以了像调试js代码一样调试TypesSript了  


### 参考文章

[JavaScript Source Map 详解](http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html)-阮一峰

<http://www.cnblogs.com/winfred/p/6388386.html>

# [TypeScript笔记 3--基础类型](http://www.cnblogs.com/winfred/p/6388386.html)

强类型是TypeScript值得称赞的特性，对于很多后端开发者来说是绝对的福利，加上静态检查使得我们可以在开发态就可以发现很多问题。

TypeScript中的类型和JavaScript差不多，下面我们一一介绍。

### 1 布尔(boolean)

let isMe:boolean = true

### 2 数值(number)

同JavaScript中一样，数值类型都是浮点数，十进制，十六进制，八进制和二进制都支持。

let count:number = 10;

let hexLiteral: number = 0xf00d;

let binaryLiteral: number = 0b1010;

let octalLiteral: number = 0o744;

### 3 字符串(string)

字符串可以使用双引号和单引号

let msg: string = "Hello World!";

let warning: string = 'alert';

另外，我们还可以使用字符模板，我们可以直接定义多行字符串，还可以使用变量。在JavaScript中要想使用多行必须使用+连接起来，现在字符模板简单多了

let name:string = `Winfred`;

let log:string = `Hello, My name is ${name}

I am 20 years old`

### 4 数组(Array)

有两种方式可以定义数组

let numbers:number[] = [1,2,3]

let numbers:Array<number> = [1,2,3]

### 5 元组(Tuple)

元组是已知的元素数量和类型的数组，每个元素类型不一定要相同。

let t:[string, number] = ['msg', 10]

console.log(t[0]) // msg

console.log(t[1]) // 10

### 6 枚举(enum)

枚举在JavaScript，对其补充，不过在C++，Java语言中都是有的

enum Color {Red, Green, Blue};

let c1: Color = Color.Green;

let c2: Color = Color[0];

// 默认从0开始编号，我们可以改变其默认编号，下面两种都是可以的

enum Color {Red = 1, Green, Blue};

enum Color {Red = 1, Green = 2, Blue = 4};

### 7 任意值(any)

any是指不知道的类型，直到给其赋值后才知道其类型,而且可以改变其类型

let msg:any = 10;

msg = "string";

msg = true;

// 编辑器不会报错，可以正常使用

### 8 空值,null和undefined

在TypeScript中void,null,undefined都是一种类型。

let t1:void = null;

let t2:void = undefined;

let t3:null = null;

let t4:undefined = undefined;

声明一个空值只能给其赋值null或者undefined

### 9 never

never就是永远达不到的类型，很少用，一般用于异常等

<http://www.cnblogs.com/winfred/p/6395699.html>

# [TypeScript笔记 4--变量声明](http://www.cnblogs.com/winfred/p/6395699.html)

在[上一篇：基础变量](http://www.cnblogs.com/winfred/p/6388386.html)中我们在声明变量时使用了关键字let，这和JS中的var有点类似。

### 语法

基本语法：let 变量名:类型。当然类型不是必须的。

let x:number;

let y:string = 'xx';

let z;

z = 'xxx';

### var

var和let最大区别就是作用域不同,let有严格的块作用域。

* 1 我们都知道，js中的作用域不是很严格。比如下面代码：

flag = true;

if (flag) {

var msg = 'I am OK';

}

console.log(msg); // I am OK

在js中变量msg仍然可以输出I am OK，甚至变量flag不适用关键字也可声明。

* 再举一个关于闭包的例子,如下：

for (var i = 0; i < 10; i++) {

setTimeout(function() { console.log(i); }, 100 \* i);

}

输出结果则是：10个10。

* 3 重复定义和变量提升

var x = 10;

var x = 20;

在js中连续声明同一个变量是不会报错的，我们只会得到最后声明的。

* 4 变量提升

a++;

var a';

变量提升在js中也很常见，解释器解析时会先遍历变量声明，然后将其提升到函数最上面，所以上面的写法完全没有问题

以上这些问题经常在review时漏洞，引发无穷的麻烦。很多团队中也禁止出现这种代码，就是防止潜在的风险。现在有了let我们再也不用担心这方面的问题了。

### let

和var相比，let是块作用域。 块作用域变量在包含它们的块或for循环之外是不能访问的。

for (let i = 0; i < 10; i++) {

setTimeout(function() { console.log(i); }, 100 \* i);

}

输出结果：从0到9

flag = true;

if (flag) {

let msg = 'I am OK';

}

console.log(msg); // 编译报错

var x = 10;

var x = 20; // 编译报错

a++;

var a'; // 编译报错

var和let区别

* 作用不同，let是有严格的块作用域
* 在块中，不能重复声明，要先声明后使用，这和java这种强类型语言一样

### const声明

const是对let的一个增强，它能阻止对一个变量再次赋值。类似java中的final.

const msg = "This is a message";

msg = "xx" // 编译报错

const person = {

name: "Aurora",

age: 20

}

person = {}; // 编译报错

person.name = "tom" // OK

person.age = 10 // OK

对于对象，我们可以改变对象内部属性的值，但是不能改变对象引用（不确切地说是对象地址）

<http://www.cnblogs.com/winfred/p/6399639.html>

# [TypeScript笔记 5--变量声明(解构和展开)](http://www.cnblogs.com/winfred/p/6399639.html)

## 解构是什么

解构(destructuring assignment)是一种表达式，将数组或者对象中的数据赋给另一变量。

在开发过程中，我们经常遇到这样问题，需要将对象某个属性的值赋给其它两个变量。代码如下:

var person = {

name:'xxxx',

age: 20,

hobby:['football', 'run']

}

var a = person.name;

var hobby = person.bobby;

有了解构，我们可以精简代码了

var person = {

name:'xxxx',

age: 20,

hobby:['football', 'run']

}

let {a, hobby} = person;

let [first] = person.hobby;

### 数组解构

let nums = [1,2,3]

let [f,s] = nums;

console.log(f); // 输出： 1

console.log(s); // 输出： 2

将数组中前两项的值分别赋给f和s

**变量值交换**

[f, s] = [s, f]; //不需要 let [f, s] = [s, f] 因为 f, s 都是已经定义好的变量。只是互换值操作

console.log(f); // 输出： 2

console.log(s); // 输出： 1

**使用...创建剩余变量**

let nums= [1,2,3,4]

let [f, ...rest] = nums;

console.log(f); // 输出： 1

console.log(rest);// 输出： [2,3,4]

**忽略数组中某些元素**

let [first] = [1, 2, 3, 4];

console.log(first); // 输出： 1

let [, second, , fourth] = [1, 2, 3, 4];

### 对象解构

let o = {

a: "foo",

b: 12,

c: "bar"

}

let { a, b } = o;

console.log(a); // 输出： foo

console.log(b);// 输出： 12

将对象o.a赋值给a,0.b赋值给b，这里的a,b都是对象属性名且必须一致，而属性c则会忽略

**重命名**  
上述例子中声明的变量a和b必须和对象中属性一致，如果想换一个名字，写法如下：

let { a: aa, b: bb } = o;

**默认值**

let o = {

a: "foo",

b: undefined,

c: "bar"

}

let {a, b=1}= o;

console.log(a); // 输出： foo

console.log(b);// 输出： 1

当属性b的值为undefined时，解构表达式会使用默认值，对于null, 则会仍然赋值null

## 展开

展开和解构正好相反，将一个数组展开为另一个数组，或将一个对象展开为另一个对象。  
**数组展开**

let first = [1, 2];

let second = [3, 4];

let bothPlus = [0, ...first, ...second, 5];

console.log(bothPlus);// 输出：0,1,2,3,4,5

展开操作会对数组first和second浅拷贝，如果fisrt值修改，则bothPlus值不会改变。但是如果first中是对象数组，改变对象内部的值，bothPlus对应的值会改变。

**对象展开**

let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

let search = { ...defaults, food: "rich" };

console.log(search); //输出：{food: "rich", price: "$$", ambiance: "noisy"}

展开是从左到右处理，意味着对象后面的属性会覆盖前面的。如下：

let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

let search = {food: "rich", ...defaults };

console.log(search); //输出：{food: "spicy", price: "$$", ambiance: "noisy"}

var person = {

name: "William",

age: 45,

company: "D3 Security",

entry: new Date("2018-05-01"),

score: undefined

}

var flower = {

season: "Four",

name: "Rose",

score: 89

}

let creature = {flower, person};

console.log(creature);

{ flower:

{ season: 'Four', name: 'Rose', score: 89 },

person:

{ name: 'William',

age: 45,

company: 'D3 Security',

entry: 2018-05-01T00:00:00.000Z,

score: undefined

}

}

var person = {

name: "William",

age: 45,

company: "D3 Security",

entry: new Date("2018-05-01"),

score: undefined,

list: {id: 100, title:"CEO"}

}

var flower = {

season: "Four",

name: "Rose",

score: 89,

list: {subject:"English", when: "Today"}

}

let creature = {...flower, ...person};

console.log(creature);

{ season: 'Four', - 所有属性合并，相同的则后面的覆盖前面的

name: 'William',

score: undefined,

list: { id: 100, title: 'CEO' }, - 后面的覆盖前面的， 而不是合并。

age: 45,

company: 'D3 Security',

entry: 2018-05-01T00:00:00.000Z

}

注意：使用...创建展开，[请使用typescript@2.1.x版本](mailto:%E8%AF%B7%E4%BD%BF%E7%94%A8typescript@2.1.x版本)<http://www.cnblogs.com/winfred/p/6582719.html>

# [TypeScript笔记 6--接口](http://www.cnblogs.com/winfred/p/6582719.html)

### 接口定义

接口和Java语言一样，都是通过关键字interface定义的，如下例子：

interface People {

name: string;

age: number;

hobby?: string; // 可选属性

readonly sex: string; // 只读属性

run: (steps: number) => void; // 箭头函数

play():void; // 普通函数

}

### 接口使用

function test(people:People) {

// content

people.sex = 'xxx'; // 编译器报错

}

let p = {

age: 1, name: 'z', sex: 'x',

run: (s: number) => {

return 1;

},

play:function() {

}，

height:180

}

test(p);

如上例子，当前对象p作为参数传入函数test中时，编译器只会检查那些必需的属性(函数也是一种属性)是否存在，类型是否匹配，如果缺少必要的属性，编译器会报错。至于多出的属性height，编译器则不会检查，这点我们可以这样理解：对象p是People的实现，也就是其子类。height是子类中的属性，这样的话很容易理解。

注意以下的区别：

function showObj(obj: {name:string, age:number}) {

    console.log(obj);

    return {name: obj.name+ ' Mr', age: obj.age + 1000};

}

let myinfo = {name: "William", age: 47, title: "Senior Developer"};

console.log( showObj(myinfo));  //  Okey

console.log( showObj( {name: "William", age: 47, title: "Senior Developer"} ));  // Error

### 可选属性

接口里的属性不全都是必需的。属性名后面加上?表示是一个可选属性。

有些是只在某些条件下存在，或者根本不存在.如上述例子中hobby属性，我们在构造p对象时就算没有传入hobby属性，编译器不会报错。

### 只读属性

只读属性用readonly修饰,一旦该属性赋值后，就不能再修改了，否则编译器报错。

前几章中学习了常量const，它们没有什么差别，最简单判断该用readonly还是const的方法是看要把它做为变量使用还是做为一个属性。 做为变量使用的话用 const，若做为属性则使用readonly。

### 函数

关于函数的声明有两种，箭头函数和普通方式。在上面的例子中，我们声明了两个函数run和play，虽然生命方法不一样，但是使用方法是一样的，只是作用域不一样，究竟有什么不一样，以后再讲。

### 接口实现和继承

类可以实现接口，接口也可以继承接口并且只能描述公有属性，这些都是和Java是一样的。

class Player implements People {

name: string;

age: number;

readonly sex: string; // 只读属性

constructor() {

}

run= (steps: number) => {

} // 箭头函数

play():number {

return 100;

}

}

interface Player extends People {

item:string;

}

### 接口继承类，多继承

在Java中接口只能继承接口，而且只能继承一个接口，但是ts中的接口可以继承类，同时支持多继承。接口继承只能继承类中的属性和函数（不继承其实现）。父类中的私有属性和受保护属性也能继承，但是如果这样的继承的话，这个接口类型只能被这个类或其子类所实现。

class Animal {

name: string;

private age: string;

run(): void {

console.log('running...');

}

}

interface Person extends Animal {}

function eat(p: Person) {}

let p = { name: '', age: 'x', sayHello: function () { return '' } };

eat(p); // 编译报错

上述代码会报错，因为age是私有属性，而对象p中age是公有的，这样会产生冲突。所以接口Person必须是Animal的子类所实现。

要想编译不报错，有两种方法：第一是将age属性改为公有，第二种是新增一个类，该类是Animal子类， 如下：

class Animal {

name: string;

private age: string;

run(): void {

console.log('running...');

}

}

interface Person extends Animal {

say(): void;

}

class Child extends Animal {

say(): void {

}

}

function eat(p: Person) { }

let p = new Child();

eat(p);// 这样不会报错

分类: [typescript](http://www.cnblogs.com/winfred/category/943284.html)

# [TypeScript基础入门 - 接口 - 函数类型](https://www.gowhich.com/blog/879)

interface SomeInterface {

  (arg1: string, arg2: string): boolean; - 没有名字，已有约定的参数和结果类型

}

let someFunc: SomeInterface

someFunc = function (arg1: string, arg2: string) {

    const res = arg1.search(arg2)

    return res > -1;

}

console.log(someFunc('weast','east'));

# [TypeScript基础入门 - 接口 – 混合类型](https://www.gowhich.com/blog/879)

interface iA4 {

(x: number, y: number):string;

show: ()=>void;

total: number;

}

let func: iA4 = <iA4>function(a: number, b: number) {

console.log(`Init Total = ${a+b}`);

return `Init Total = ${a+b}`;

}

func.total = 515;

func.show = ()=>{ - 注意和下面的区别： this 的指向不同

console.log(`Show Total: ${this.total}`);

}

func(233, 155);

func.show();

console.log(func.total);

Init Total = 388 - 此处输出

Show Total: undefined - 注意此处输出并非是515，而是 undefineds

515 - 输出正确

---------------------------------------------------------------------------------------------

interface iA4 {

(x: number, y: number):string;

show():void;

total: number;

}

let func: iA4 = <iA4>function(a: number, b: number) {

console.log(`Init Total = ${a+b}`);

return `Init Total = ${a+b}`;

}

func.total = 515;

show:()=>void; 或 show() 都可以

func.show = function(){ - 注意和上面的区别：this 的指向不同

console.log(`Show Total: ${this.total}`);

}

func(233, 155);

func.show();

console.log(func.total);

Init Total = 388

Show Total: 515

515

### **继承接口**

和类一样，接口也可以相互继承。 这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。如下实例演示

interface Shape {

    color: string;

}

interface Square extends Shape {

    sideLength: number;

}

let square = <Square> {};

square.color = 'red'

square.sideLength = 10;

interface Shape {

    color: string;

}

interface PenStroke {

    penWidth: number;

}

interface Square extends Shape, PenStroke {

    sideLength: number;

}

let square = <Square> {};

square.color = 'red'

square.sideLength = 10;

square.penWidth = 10;

# TypeScript学习笔记之 接口（Interface）

2017年01月12日 15:42:04 [朽木\_不折](https://me.csdn.net/yuzhiqiang_1993) 阅读数：10066更多

个人分类： [web前端](https://blog.csdn.net/yuzhiqiang_1993/article/category/6536371)

所属专栏： [从零开始，学习web前端](https://blog.csdn.net/column/details/15081.html)

版权声明：本文为博主原创文章，转载请注明地址。如果文中有什么纰漏或错误的话,请留言指正，我会及时更正。如果您觉得本文还不错的话，记得点个赞呦，希望能帮到你，谢谢。 https://blog.csdn.net/yuzhiqiang\_1993/article/details/54378607

在java中，接口是用来定义一些规范，使用这些接口，就必须实现接口中的方法，而且接口中的属性必须是常量。   
javascript中是没有接口的概念的。所以TypeScript在编译成 JavaScript 的时候，所有的接口都会被擦除掉。   
而TypeScript的核心之一就是类型检查。 在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约并让代码看起来更好理解。

**接口的简单使用**   
如下代码所示，我们需要给add函数中传入一个对象，如果不使用接口，我们传入的对象可以有其他属性。

**function** **add**(num:{x:number,y:number}){

console.log(num.x+num.y);

}

**let** **n**={x:1,y:2,z:3};//这里我们的对象只要包含必要属相且值类型正确即可，允许存在其他属性

**add**(n);//不报错

// add({x:1,y:2,z:2})//报错,会提示z不存在

**add**({x:1,y:2});//不报错

用接口来表示，可以看到，我们先定义个接口，然后把符合接口定义的对象传进去，这样能提高代码可读性。   
需要注意的是：   
如果我们不给传进去的对象指定是接口类型的数据，那么传入的对象参数可以包含其他属性，编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。   
如果我们给对象指定是接口类型的数据，那么，对象的属性必须和定义好的接口的必要属性一致。必要属性不能多也不能少。

/\*\*

\* Created by yzq on 2017/1/12.

\*/

interface num{

x:number;

y:number;

}

function add(n:num){

console.log(n.x+n.y);

}

let n={x:1,y:2,z:3};

let n1:num={x:1,y:2};//如果我们将n1指定为num类型的数据，那么，该对象所包含的属性必须跟定义好的接口完全一致。不能多也不能少。

// let n2:num={x:1};//错误，缺少y属性

// let n3:num={x:1,y:2,z:3};//错误，没有找到z属性，当我们将对象字面量赋值给变量或作为参数传递的时候。TypeScript会进行额外属性检查。如果一个对象字面量存在任何“目标类型”不包含的属性时，就会报错。

add(n);//不报错

add(n1);//不报错

add(n2);

add(n3);

// add({x:1,y:2,z:2})//报错,会提示z不存在，额外的属性检查

add({x:1,y:2});//不报错

**接口的可选属性**   
在实际应用中，接口里的属性不全都是必需的。 有些是只在某些条件下存在，或者根本不存在。在这种个情况下可以是用接口的可选属性去定义。   
如下代码所示，name和age被定义为可选属性，那么在传对象的时候name和age就可有可无。

interface Person{

name?:string;

age?:number

}

function getInfo(p:Person){

console.log(p.name);

console.log(p.age);

}

let student={}

// let student={name:"yzq"}

// let student={name:"yzq",age:23}

getInfo(student);

**接口的只读属性**   
如果我们希望对象属性只能在对象刚刚创建的时候修改其值。 我们可以在属性名前用 readonly来指定只读属性。

**interface** **Point**{

readonly x:number;

readonly y:number;

}

**function** **getPoint**(p:Point){

console.log(p.x);

console.log(p.y);

}

let point:Point={x:1,y:2};

// point.x=2;//错误 这里不能再子修改值

getPoint(point);

**定义只读数组**   
只读数组也一样，一旦定义后不能再修改数组

**let** a:ReadonlyArray<number> =[1,2,3,4,5];

// a[0]=2;//不能再修改该数组

// a.length=20;

**接口的函数类型**   
接口可以描述javascript的任何对象，不仅能描述对象的属性类型，当然也能描述对象的函数类型。   
如下代码所示，接口描述了这个函数的参数类型和返回值类型

**interface** **getStr** {

/\*在这里我们描述了这个接口有个函数 这个函数传进去2个number类型的参数 然后返回一个string类型的数据\*/

(x: number, y: number): string;

}

let myStr : getStr;

myStr = **function** (gradeNum: number, classNum: number) {

**return** `${gradeNum}年级${classNum}班`;

}

console.log(myStr(2,8));//2年级8班

**接口的数组类型（可索引的类型）**   
跟接口描述函数类型差不多，我们也可以描述那些能够“通过索引得到”的类型，比如通过下标获取数组中的值 a[2];需要注意的是，索引器的类型只能为 number 或者 string。

**interface** stringArr{

/\*描述的一个数组 这个数组里面的元素是string类型 并且只能通过number类型来索引 [index:number]是索引器 string是该数组元素类型\*/

[index:number]:**string**;

// age:number;//需要注意的是，当我们将这个接口是数组类型时，那么，接口中定义的其它属性的类型都必须是该数组的元素类型。 这里的number类型是报错的

}

**let** strArr:stringArr;

strArr=["1","2"];

// strArr.name="3";

**let** str:**string**=strArr[0];

console.log(str);//打印1

**接口的类类型**   
所谓类类型，就是一个类去实现接口，而不是直接把接口拿来用，这更符合我们的使用习惯。

**interface** **IClock**{

/\*定义了一个接口 这个接口中有一个属性和一个方法\*/

currentTime:Date;

getTime(d:Date);

}

/\*Time类实现IClock接口\*/

**class** **Time** **implements** **IClock**{

currentTime:Date;

getTime(d:Date){

**this**.currentTime=d;

}

}

**扩展接口**   
在TypeScript中，接口跟类一样是可以相互继承的， 这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。

/\*接口可以继承接口 并且可以多继承\*/

**interface** **shape**{

color:string;

}

**interface** **pen** **extends** **shape**{

width:number;

}

let circle=<pen>{};//注意这里的写法，创建一个对象并指定泛型

circle.color="red";//这里可以获取color属性

circle.width=2;//有width属性

一个接口可以继承多个接口，创建出多个接口的合成接口。

/\*接口可以继承接口 并且可以多继承\*/

**interface** **shape**{

color:string;

}

**interface** **pen** **extends** **shape**{

width:number;

}

**interface** **Circle** **extends** **shape**,**pen**{

point:number[];

}

let c=<Circle>{};//注意这里的写法，创建一个对象并指定泛型

c.point=[1,2];

c.color="red";

c.width=1;

**混合类型**   
所谓的混合类型就是在一个接口中定义多种类型，比如属性，函数，数组等。

**interface** **Counter** {

(start: number): string;

interval: number;

reset(): **void**;

}

**function** **getCounter**()**: Counter** {

let counter = <Counter>**function** (start: number) { };

counter.interval = 123;

counter.reset = **function** () { };

**return** counter;

}

let c = getCounter();

c(10);

c.reset();

c.interval = 5.0;

# <http://www.cnblogs.com/winfred/p/6959128.html>

# [webpack打包非模块化js](http://www.cnblogs.com/winfred/p/6959128.html)

本文主要记录了非模块化js如何使用webpack打包

### 模块化打包实现方式

webpack是模块打包工具，通过入口文件递归遍历其依赖图谱，绝对是打包神器。

bar.js

export default function bar() {

//

}

foo.js

import bar from './bar';

bar();

通过如下，webpack配置很快实现打包。通过插件我们还可以实现文件压缩，开发态我们还可以配置sourceMap进行代码调试(chrome浏览器支持sourcemap调试)。

module.exports = {

entry: './foo.js',

output: {

filename: 'bundle.js'

},

devtool: "source-map",

plugins: [

// compress js

new webpack.optimize.UglifyJsPlugin({

sourceMap: true

})

]

}

### 非模块化文件打包压缩

这里我们可以使用webpack可以配置多入口文件及ExtractTextPlugin 插件将非模块文件压缩到一个文件中。

m1.js

functon a() {

console.log('m1 file')

}

m2.js

functon b() {

console.log('m2 file')

}

webpack配置文件

var webpack = require('webpack')

var path = require('path')

module.exports = {

entry: {

'app': [

'./src/a.js',

'./src/b.js'

]

},

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "[name].js"

}

}

打包后，发现我去不能运行？？原因是webpack打包会将每个文件内容放入闭包函数中，我们去调用闭包中的函数，当然不行啦。

/\*\*\*\*\*\*/ (function(modules) { // webpackBootstrap

/\*\*\*\*\*\*/ // The module cache

/\*\*\*\*\*\*/ var installedModules = {};

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // The require function

/\*\*\*\*\*\*/ function \_\_webpack\_require\_\_(moduleId) {

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Check if module is in cache

/\*\*\*\*\*\*/ if(installedModules[moduleId]) {

/\*\*\*\*\*\*/ return installedModules[moduleId].exports;

/\*\*\*\*\*\*/ }

/\*\*\*\*\*\*/ // Create a new module (and put it into the cache)

/\*\*\*\*\*\*/ var module = installedModules[moduleId] = {

/\*\*\*\*\*\*/ i: moduleId,

/\*\*\*\*\*\*/ l: false,

/\*\*\*\*\*\*/ exports: {}

/\*\*\*\*\*\*/ };

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Execute the module function

/\*\*\*\*\*\*/ modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Flag the module as loaded

/\*\*\*\*\*\*/ module.l = true;

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Return the exports of the module

/\*\*\*\*\*\*/ return module.exports;

/\*\*\*\*\*\*/ }

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // expose the modules object (\_\_webpack\_modules\_\_)

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.m = modules;

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // expose the module cache

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.c = installedModules;

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // define getter function for harmony exports

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.d = function(exports, name, getter) {

/\*\*\*\*\*\*/ if(!\_\_webpack\_require\_\_.o(exports, name)) {

/\*\*\*\*\*\*/ Object.defineProperty(exports, name, {

/\*\*\*\*\*\*/ configurable: false,

/\*\*\*\*\*\*/ enumerable: true,

/\*\*\*\*\*\*/ get: getter

/\*\*\*\*\*\*/ });

/\*\*\*\*\*\*/ }

/\*\*\*\*\*\*/ };

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // getDefaultExport function for compatibility with non-harmony modules

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.n = function(module) {

/\*\*\*\*\*\*/ var getter = module && module.\_\_esModule ?

/\*\*\*\*\*\*/ function getDefault() { return module['default']; } :

/\*\*\*\*\*\*/ function getModuleExports() { return module; };

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.d(getter, 'a', getter);

/\*\*\*\*\*\*/ return getter;

/\*\*\*\*\*\*/ };

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Object.prototype.hasOwnProperty.call

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.o = function(object, property) { return Object.prototype.hasOwnProperty.call(object, property); };

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // \_\_webpack\_public\_path\_\_

/\*\*\*\*\*\*/ \_\_webpack\_require\_\_.p = "";

/\*\*\*\*\*\*/

/\*\*\*\*\*\*/ // Load entry module and return exports

/\*\*\*\*\*\*/ return \_\_webpack\_require\_\_(\_\_webpack\_require\_\_.s = 0);

/\*\*\*\*\*\*/ })

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*/ ([

/\* 0 \*/

/\*\*\*/ (function(module, exports, \_\_webpack\_require\_\_) {

\_\_webpack\_require\_\_(1);

module.exports = \_\_webpack\_require\_\_(2);

/\*\*\*/ }),

/\* 1 \*/

/\*\*\*/ (function(module, exports) {

/\*\*\*/ }),

/\* 2 \*/

/\*\*\*/ (function(module, exports) {

function b() {

console.log('b file')

}

/\*\*\*/ })

/\*\*\*\*\*\*/ ]);

//# sourceMappingURL=app.js.map

怎么办呢？我们可以对我们当前代码进行修改，让所有函数或属性都能通过window对象调用即可。

(function(Demo) {

Demo.module1 = {

msg:function() {

return 'Hello World';

}

}

})(window.Demo = window.Demo || {})

所以我们对于上面闭包形式且所有对象都挂在window对象这种类型代码，不会出现函数调用不到现象。通过webpack压缩后一样正常运行

<http://www.cnblogs.com/winfred/p/8179815.html>

# [TypeScript和Node模块解析策略](http://www.cnblogs.com/winfred/p/8179815.html)

一般我们在模块化编码时，总会导入其它模块，通常我们使用如下语法：

import { A } from './a'; // ES6语法

import { A } from 'a';

var A = require('./a'); // commonjs规范

不论使用哪种语法，导入的文件一般有两种：内部文件(自己开发的)和外部（node\_modules）中两种，其中导入内部模块称之为相对导入，导入node\_modules中，称之为非相对导入，它们在语法上的区别就是导入的路径是否是相对的

接下来我们看看typescript和node中它们是如何解析模块的

### Typescript模块解析策略

### 相对导入

假如b.ts路径是：/root/src/b.ts

import { A } from './a';

typescript编译器在查找a模块时会依次按照如下顺序查找，如果仍然找不到则会模块找不到的错误。

/root/src/a.ts

/root/src/a.tsx

/root/src/a.d.ts

/root/src/a/package.json (如果指定了"types"属性,则使用types中)

/root/src/a/index.ts

/root/src/a/index.tsx

/root/src/a/index.d.ts

### 非相对导入

假如b.ts路径是：/root/src/b.ts

import { A } from 'a';

typescript编译器在查找a模块时会按照如下顺序查找：

/root/src/node\_modules/a.ts

/root/src/node\_modules/a.tsx

/root/src/node\_modules/a.d.ts

/root/src/node\_modules/a/package.json

/root/src/node\_modules/a/index.ts

/root/src/node\_modules/a/index.tsx

/root/src/node\_modules/a/index.d.ts

/root/node\_modules/a.ts

/root/node\_modules/a.tsx

/root/node\_modules/a.d.ts

/root/node\_modules/a/package.json

/root/node\_modules/a/index.ts

/root/node\_modules/a/index.tsx

/root/node\_modules/a/index.d.ts

/node\_modules/a.ts

/node\_modules/a.tsx

/node\_modules/a.d.ts

/node\_modules/a/package.json

/node\_modules/a/index.ts

/node\_modules/a/index.tsx

/node\_modules/a/index.d.ts

其中在上面两处空白行处，编译器会跳到上一级目录查找，直到到工程根目录

注意：有时候我们在导入外部模块（没有ts文件，只有），编译器会报模块找不到，但是我们node\_modules确实有，这种方式不是编译器bug而需要我们在配置文件tsconfig.json中修改模块解析策略：

"moduleResolution": "node"

说到这里我们看看Nodejs时如何解析模块的，NodeJs使用了commonjs模块规范,typescript编译和其大同小异。

### Nodejs相对导入

假如b.ts路径是：/root/src/b.js

var A = require('./a')

typescript编译器在查找a模块时会按照如下顺序查找：

/root/src/a

/root/src/a.js

/root/src/a.json

/root/src/a/package.json (如果指定了"main"属性,则使用main中的)

/root/src/a/index.js

/root/src/a/index.json

上述第二步中，假如main:"./core/main.js",则最终模块路径：

/root/src/a/core/main.js

### Nodejs非相对导入

var A = require('a')

typescript编译器在查找a模块时会按照如下顺序查找：

/root/src/node\_modules/a.js

/root/src/node\_modules/a/package.json (如果指定了"main"属性,则使用main中的)

/root/src/node\_modules/a/index.js

/root/node\_modules/a.js

/root/node\_modules/a/package.json (如果指定了"main"属性,则使用main中的)

/root/node\_modules/a/index.js

/node\_modules/a.js

/node\_modules/a/package.json (如果指定了"main"属性,则使用main中的)

/node\_modules/a/index.js

分类: [typescript](http://www.cnblogs.com/winfred/category/943284.html)

[TypeScript （Generator）](http://www.cnblogs.com/winfred/p/8216650.html) function\* Generator() – yield

The generator object just follows the iterator interface (next, throw, return functions).

function\* Gens() {

let a = [10, 20, 30, 40];

try {

for(let b of a) {

if(b>30) throw Error("Over 40");

yield b;

}

}

catch(err) {

console.log("Error Occurred: " + err);

}

}

let g = Gens();

let a1 = g.next();

if(a1.done==false) console.log(a1.value);

console.log(g.next());

console.log(g.next());

console.log(g.throw("External Error"));

console.log(g.next());

console.log(g.next());

10

{ value: 20, done: false }

{ value: 30, done: false }

Error Occurred: External Error - 从这里抛出错误以后，就没有 yield 了

{ value: undefined, done: true }

{ value: undefined, done: true }

{ value: undefined, done: true }

function\* Gens() {

let a = [10, 20, 30, 40];

try {

for(let b of a) {

if(b>30) throw Error("Over 40");

yield b;

}

}

catch(err) {

console.log("Error Occurred: " + err);

}

return 100;

}

let g = Gens();

let a1 = g.next();

if(a1.done==false) console.log(a1.value);

console.log(g.next());

console.log(g.next());

console.log(g.throw("External Error"));

console.log(g.next());

console.log(g.next());

10

{ value: 20, done: false }

{ value: 30, done: false }

Error Occurred: External Error

{ value: 100, done: true } - 还可以再next 一次，返回最后的return, 遇到return就代表结束

{ value: undefined, done: true }

{ value: undefined, done: true }

# <http://www.cnblogs.com/winfred/p/8216650.html>

# [TypeScript装饰器（decorators）](http://www.cnblogs.com/winfred/p/8216650.html)

装饰器是一种特殊类型的声明，它能够被附加到类声明，方法， 访问符，属性或参数上，可以修改类的行为。 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。

例：

@Path('/hello')

class HelloService {}

在TypeScript中装饰器还属于实验性语法，所以要想使用必须在配置文件中tsconfig.json编译选项中开启：

{

"compilerOptions": {

"experimentalDecorators": true

}

}

### 如何定义装饰器

装饰器本身其实就是一个函数，理论上忽略参数的话，任何函数都可以当做装饰器使用。例：

demo.ts

function Path(target:any) {

console.log("I am decorator.")

}

@Path

class HelloService {}

使用tsc编译后,执行命令node demo.js，输出结果如下：

I am decorator.

### 装饰器执行时机

修饰器对类的行为的改变，是代码编译时发生的（不是TypeScript编译，而是js在执行机中编译阶段），而不是在运行时。这意味着，修饰器能在编译阶段运行代码。也就是说，修饰器本质就是编译时执行的函数。  
在Node.js环境中模块一加载时就会执行

### 函数柯里化解决参数问题

但是实际场景中，有时希望向装饰器传入一些参数, 如下：

@Path("/hello", "world")

class HelloService {}

此时上面装饰器方法就不满足了（VSCode编译报错），这是我们可以借助JavaScript中函数柯里化特性

function Path(p1: string, p2: string) {

return function (target) { // 这才是真正装饰器

// do something

}

}

### 五种装饰器

在TypeScript中装饰器可以修饰四种语句：类，属性，访问器，方法以及方法参数。

##### 1 类装饰器

应用于类构造函数，其参数是类的构造函数。  
注意class并不是像Java那种强类型语言中的类，而是JavaScript构造函数的语法糖。

function Path(path: string) {

return function (target: Function) {

!target.prototype.$Meta && (target.prototype.$Meta = {})

target.prototype.$Meta.baseUrl = path;

};

}

@Path('/hello')

class HelloService {

constructor() {}

}

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

let hello = new HelloService();

console.log(hello.$Meta) // 输出：{ baseUrl: '/hello' }

类装饰器：

function path(url:string) {

// 函数只有一个参数，并且参数代表了类Function

return function(t: Function) { - 只有一个参数代表类，类是Function

t.prototype.myUrl = url;

}

}

@path("http://www.sohu.com/")

class Teacher{

public myUrl:string; - 要想类的实例能访问和继承同名的值，必须要定义类的属性来继承访问

constructor(){}

}

let h: Teacher = new Teacher();

console.log(Teacher.prototype.myUrl); - 注意这是类原型访问属性

console.log(h.myUrl); - 注意这是类的实例访问属性，类必须要定义同名属性才可以继承

访问自原型

输出：

http://www.sohu.com/

http://www.sohu.com/

我们来看看： 原型prototype 与实例的同名属性的关系

let h: Teacher = new Teacher();

console.log("value: " + h.title);

console.log(Teacher.prototype.myUrl);

console.log(h.myUrl);

h.myUrl = "google";

Teacher.prototype.myUrl = "www.yahoo.com";

console.log(Teacher.prototype.myUrl);

console.log(h.myUrl);

输出：

http://www.sohu.com/

http://www.sohu.com/

www.yahoo.com

google - 注意实例属性一旦改变， 和原型的继承关系就打断了，这符合javascript 的原理

let h: Teacher = new Teacher();

console.log("value: " + h.title);

console.log(Teacher.prototype.myUrl);

console.log(h.myUrl);

//h.myUrl = "google";

Teacher.prototype.myUrl = "www.yahoo.com";

console.log(Teacher.prototype.myUrl);

console.log(h.myUrl);

输出：

http://www.sohu.com/

http://www.sohu.com/

www.yahoo.com

[www.yahoo.com](http://www.yahoo.com) - 注意实例的属性和原型属性的值一样

##### 2 方法装饰器

它会被应用到方法的 属性描述符上，可以用来监视，修改或者替换方法定义。  
方法装饰会在运行时传入下列3个参数：

* 1、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。
* 2、成员的名字。
* 3、成员的属性描述符。

function GET(url: string) {

return function (target, methodName: string, descriptor: PropertyDescriptor) {

!target.$Meta && (target.$Meta = {});

target.$Meta[methodName] = url;

}

}

class HelloService {

constructor() { }

@GET("xx")

getUser() { }

}

console.log((<any>HelloService).$Meta);

注意：在vscode编辑时有时会报作为表达式调用时，无法解析方法修饰器的签名。错误，此时需要在tsconfig.json中增加target配置项：

{

"compilerOptions": {

"target": "es6",

"experimentalDecorators": true,

}

}

function muser(x:number, y:number) {

// 带有三个参数：目标，方法名，方法描述

return function(target:any, methodName:string, descriptor:PropertyDescriptor) {

// 可以临时改变方法的定义， 但是最后不会替代实际的方法定义

target[methodName] = function(x:number, y:number):void {

console.log("Muser: X=" + x + " - " + y);

}

if(typeof(target[methodName]) == "function") target[methodName](x, y);

此时对方法的执行： 输出是： Muser: X=300 - 433

console.log(target[methodName]);

console.log(methodName);

console.log(descriptor);

}

}

class Teacher{

public myUrl:string;

constructor(){}

@muser(300, 433)

GetUser(x:number, y:number):void {

console.log("GetUser: " + x + " : " + y); - 最终方法是不会被修饰器修改覆盖的。

}

}

let h: Teacher = new Teacher();

h.GetUser(100, 200);

h.GetUser(233, 166);

输出：

Muser: X=300 – 433 - 方法被临时改变

[Function] - console.log(target[methodName]);

GetUser - console.log(methodName);

{ value: [Function], - console.log(descriptor);

writable: true,

enumerable: true,

configurable: true }

GetUser: 100 : 200 - h.GetUser(100, 200);

GetUser: 233 : 166 - h.GetUser(233, 166);

##### 3 方法参数装饰器

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：

* 1、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。
* 2、参数的名字。
* 3、参数在函数参数列表中的索引。

function PathParam(paramName: string) {

return function (target, methodName: string, paramIndex: number) {

!target.$Meta && (target.$Meta = {});

target.$Meta[paramIndex] = paramName;

}

}

class HelloService {

constructor() { }

getUser( @PathParam("userId") userId: string) { }

}

console.log((<any>HelloService).prototype.$Meta); // {'0':'userId'}

-----------------------------------------------------------

function param(x:string) {

return function(target:any, methodName:string, pIdx:number) {

target[pIdx]=x; - pIdx可以自动告诉你这是第几个参数。 其他的用途基本是只读性质的

target[pIdx]=x; - 这个赋值动作，其实没有什么用。

console.log("type: " + typeof(target[pIdx]));

console.log(target[pIdx]);

target[methodName](111, 222);

console.log(pIdx);

}

}

//@path("http://www.sohu.com/")

class Teacher{

public myUrl:string;

constructor(){}

@muser(300, 433)

GetUser(@param("x") x:number,@param("y") y:number):void {

console.log("GetUser: " + x + " : " + y);

}

}

输出：

[Running] ts-node "d:\WorkPlace\Typescript\path.ts"

type: string

y

GetUser: 111 : 222

1

type: string

x

GetUser: 111 : 222

0

Muser: X=300 - 433

##### 4 属性装饰器

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：

* 1、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。
* 2、成员的名字。

function DefaultValue(value: string) {

return function (target: any, propertyName: string) {

target[propertyName] = value;

}

}

class Hello {

@DefaultValue("world") greeting: string;

}

console.log(new Hello().greeting);// 输出: world

function ptitle(def:string) {

return function(target:any, pName:string) {

target[pName] = def; - 设置默认值得

console.log(target);

console.log(pName);

}

}

class Teacher{

public myUrl:string;

constructor(){}

@ptitle("hello world")

public title:string;

}

let h: Teacher = new Teacher();

console.log("value: " + h.title);

输出：

[Running] ts-node "d:\WorkPlace\Typescript\path.ts"

Teacher { title: 'hello world' }

title

value: hello world

### 装饰器加载顺序

function ClassDecorator() {

return function (target) {

console.log("I am class decorator");

}

}

function MethodDecorator() {

return function (target, methodName: string, descriptor: PropertyDescriptor) {

console.log("I am method decorator");

}

}

function Param1Decorator() {

return function (target, methodName: string, paramIndex: number) {

console.log("I am parameter1 decorator");

}

}

function Param2Decorator() {

return function (target, methodName: string, paramIndex: number) {

console.log("I am parameter2 decorator");

}

}

function PropertyDecorator() {

return function (target, propertyName: string) {

console.log("I am property decorator");

}

}

@ClassDecorator()

class Hello {

@PropertyDecorator()

greeting: string;

@MethodDecorator()

greet( @Param1Decorator() p1: string, @Param2Decorator() p2: string) { }

}

输出结果：

I am property decorator

I am parameter2 decorator

I am parameter1 decorator

I am method decorator

I am class decorator

从上述例子得出如下结论：

1、有多个参数装饰器时：从最后一个参数依次向前执行

2、方法和方法参数中参数装饰器先执行。

3、类装饰器总是最后执行。

4、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，所以参数会一直紧紧挨着方法执行。上述例子中属性和方法调换位置，输出如下结果：

I am parameter2 decorator

I am parameter1 decorator

I am method decorator

I am property decorator

I am class decorator

分类: [typescript](http://www.cnblogs.com/winfred/category/943284.html)

<http://www.cnblogs.com/winfred/p/8260885.html>

# [基于TypeScript装饰器定义Express RESTful 服务](http://www.cnblogs.com/winfred/p/8260885.html)

### 前言

本文主要讲解如何使用TypeScript装饰器定义Express路由。文中出现的代码经过简化不能直接运行，完整代码的请戳：<https://github.com/WinfredWang/express-decorator>

### 1 为什么使用装饰器

当我们在使用Express时，经常要暴露RESTful服务，代码如下：

var express = require('express');

var app = express();

app.get('/users', function(req, res) {

res.send([{name:'xx'}]);

});

// 路由模块化写法

var router = express.Router();

app.get('/users', function(req, res) {

res.send([{name:'xx'}]);

});

熟悉Java WEB童鞋知道[jax-rs](https://baike.baidu.com/item/JAX-RS/10914743?fr=aladdin)可以使用标注(annotation)声明服务。例：

@Path("/myResource")

public class SomeResource {

@GET

public String doGetAsPlainText() {

...

}

@GET

public String doGetAsHtml() {

...

}

}

使用这种方式声明的服务非常简洁方便，免去了写一坨重复代码之苦，而且看起来更加清晰，那我们看看在Node.js中如何做。

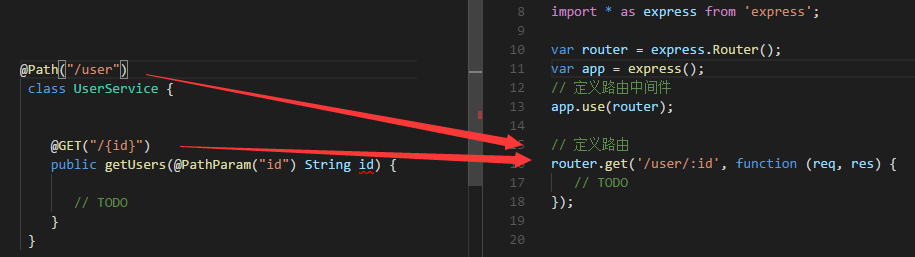
### 2 需求

参照[jax-rs](https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html)规范，我们列出如下需求：

* 使用@Path声明RESTful服务路由
* 使用@GET/@POST/@DELETE/@PUT声明子路由
* 使用@PathParam，@QueryParam，@HeaderParam，@CookieParam，@FormParam,来接受服务参数

### 3 实现思路

在ES6和TypeScript中有新特性:装饰器(Decorator)，正好我们可以借助它实现我们的需求。至于装饰器用法，可以参考我的[上一篇文章](http://www.cnblogs.com/winfred/p/8216650.html)。



上图中左边是Java中定义RESTful代码，右边是Express代码，其实他们本质上是一一对应的。我们只要在装饰器的定义中实现Express 路由即可。

继续思考，我们Express 路由到底是放到那个注解中实现呢？  
我们知道不同装饰器(类/方法/参数)执行顺序不同：

参数装饰器先执行，然后方法最后类装饰器

根据这个特性我们应该将核心实现放到类装饰器Path中执行是不是就可以了呢？

其实不是，我们看如下代码，我们在user-service.ts中定义了UserService服务。

@Path("/user")

class UserService {

@GET("/{id}")

public getUsers(@PathParam("id") id： string) {

// TODO

}

}

我们定义好了服务，然后想让Node.js模块加载，我们必须在工程入口模块(main.ts)中导入上述文件  
main.ts代码：

import { HelloService } from './hello-service'

// TODO

上述服务代码会执行吗？也就是说  
如果仅仅导入模块，而没有使用该模块的话，Node.js是否会加载这个模块呢，换句话说这个模块会执行吗？答案是NO。  
为啥呀？因为Node.js对其做了优化，只有一个模块被真正用到才会加载。

上有政策，下有对策。我们就在模块引用一下。

import { HelloService } from './hello-service'

HelloService; // 就是为了让Node加载它

这样好吗，当然不好。谁知道这是干嘛的。

所以我们应该换了思路，将Express 注册路由代码拿到装饰器外部，额外提供注册服务的入口，通过该注册服务入口，用户可以显式看到有哪些服务。

import { HelloService } from './hello-service';

import {RegisterService } from 'xxx';

RegisterService([HelloService]);//注册服务

### 4 装饰器核心代码

基于上面的思考，我们在装饰器的实现中只是单纯地存储RESTful url以及参数即可，剩下服务注册工作交给RegisterService去做。

##### Path装饰器实现

function Path(baseUrl: string) {

return function (target) {

target.prototype.$Meta = {

baseUrl: baseUrl

}

}

}

这里我们将RESTful路由存储到类的原型中，以便服务实例化时能获取到。

##### GET/POST/DELETE/PUT

function GET (url: string) => {

return (target, methodName: string, descriptor: PropertyDescriptor) => {

let meta = getMethod(target, methodName);

meta.subUrl = url;

meta.httpMethod = httpMehod;

}

}

##### QueryParam/PathParam等实现

function PahtParam(paramType: string) {

return function (target, methodName: string, paramIndex: number) {

let meta = getMethod(target, methodName);

meta.params.push({

name: paramName ? paramName : paramType,

index: paramIndex,

type: paramType

});

}

}

上述就装饰自身代码，本质上就是讲路由、http请求方法和参数存储到类的原型对象中，以便后续可以去到。

### 5 注册服务核心代码

##### 路由实现

经过上面的分析，我们可知注册服务主要将Express中注册路由交由我们框架处理,核心代码如下：

function RegisterService(app, service) {

let router = Router();

// 1. 获取存储在原型对象中的http请求信息()

let meta = getClazz(service.prototype);

// 2. 实例化服务类

let serviceInstance = new service();

let routes = meta.routes;

for (const methodName in routes) {

let methodMeta = routes[methodName];

let httpMethod = methodMeta.httpMethod;

// 3. 回调函数

let fn = (req, res, next) => {

let result = service.prototype[methodName].apply(serviceInstance, params);

res.send(result);

};

// 4. 注册路由

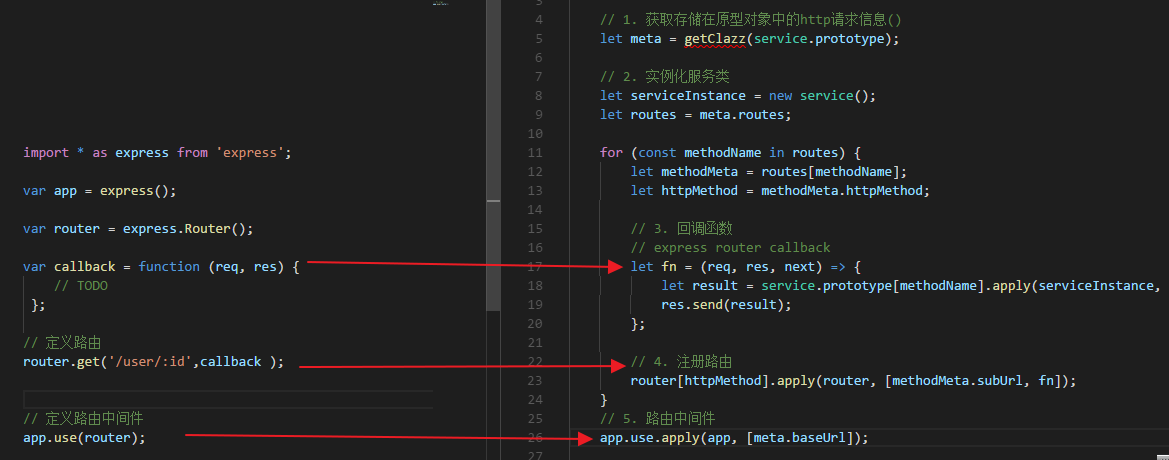
router[httpMethod].apply(router, methodMeta.subUrl);

}

// 5. 路由中间件

app.use.apply(app, [meta.baseUrl]);

}



##### http请求参数处理

@GET('/:id', [ testMidware1 ])

list( @PathParam('id') id: string, @QueryParam('name') name: string) {

return {name:"tom", age: 10}

}

用户编码时我们期望回调函数中的参数框架自动注入，而不是让用户自己从request中取，所以在注册服务代码中第3处，框架需要出更加参数装饰器中信息，从request中取值后注入回调函数中

// 3. 回调函数

let params = extractParameters(req, res, methodMeta['params']);

let fn = (req, res, next) => {

let result = service.prototype[methodName].apply(serviceInstance, params);

res.send(result);

};

// 根据参数类型，从request取出对应的值

function extractParameters(req, paramMeta) {

let paramHandlerTpe = {

'query': (paramName: string) => req.query[paramName],

'path': (paramName: string) => req.params[paramName],

'form': (paramName: string) => req.body[paramName],

'cookie': (paramName: string) => req.cookies && req.cookies[paramName],

'header': (paramName) => req.get(paramName),

'request': () => req, // 获取request/response对象，做一些特别操作

'response': () => res,

}

let args = [];

params.forEach(param => {

args.push(paramHandlerTpe[param.type](param.name))

})

return args;

}

##### response处理

javascript @GET('/:id', [ testMidware1 ]) list( @PathParam('id') id: string, @QueryParam('name') name: string) { return {name:"tom", age: 10} }  
一个服务处理完成后，总是要向浏览器返回值的，在回调函数中直接使用return语句，而不是自己调用response.send方法， 如下代码：

// 3. 回调函数

let fn = (req, res, next) => {

let result = service.prototype[methodName].apply(serviceInstance, params);

// 支持promise处理

if (result instanceof Promise) {

result.then(value => {

!res.headersSent && res.send(value);

}).catch(err => {

next(err);

});

} else if (result !== undefined) {

!res.headersSent && res.send(result);

}

};

### 6 总结

以上就是我们框架处理核心代码，核心实现主要有两步：

* 装饰器本身用来存在路由信息
* 注册机制实现express路由注册（回调函数参数处理，返回值处理等）

分类: [typescript](http://www.cnblogs.com/winfred/category/943284.html)

<https://www.2cto.com/kf/201604/497764.html>

TypeScript 面向对象基础知识

# TypeScript笔记[2]类.继承.多态

和C++，C#那些可以面向对象的语言那样，TS也可以面向对象。

与JS相比，TS更像传统的面向对象语言（如Java，C#）。所以，TS有类。

**一、类**

定义一个类，要用关键字【class】。

class Animai {

name: string

eat(): void {

alert(this.name + '吃');

}

sleep(): void {

alert(this.name + '睡觉')

}

breath(): void {

alert(this.name + '呼吸');

}

constructor(name: string) {

this.name = name;

}

}

这里定义了一个类Animal，constructor是它的构造函数。与C++不同，TS的构造函数的名字必须是constructor，而不是与类名相同。由于TS有GC，所以TS的类没有析构函数。

Animal中有个成员变量name，还有三个方法：eat、sleep、breath。一个类的对象调用类的方法也是通过点号（.），例如：

var ani: Animal = new Animal('杰瑞');

ani.eat();

类中的成员变量和函数默认是公有的（public），如果需要可以将其设为私有（private）。

class X {

public a: number; // 公有

private s: string; // 私有

}

**二、继承**

TS通过关键字extends实现继承。下面的代码，有一个类Fish继承了Animal，并重写了breath方法。

class Fish extends Animai {

constructor(name: string) {

super(name);

}

breath(): void {

alert(this.name + '吐泡');

}

}

在TS的继承中，派生类的构造函数必须通过关键字【super】来调用一下基类的构造函数。

**三、多态**

如果没有多态，继承就会黯然失色。

下面的函数实现了Animal的多态。

function Breath(p: Animai): void {

p.breath();

}

如果p是一个Animal，则会显示“呼吸”，如果p是一个Fish，则会显示“吐泡”。

多态的用法和C++的多态极其类似。不妨比较一下：

class X

{

public:

virtual ~X(){}

virtual void Func()

{

std::cout << "X::Func" << std::endl;

}

};

class Y : public X

{

public:

virtual ~Y(){}

void Func() override

{

std::cout << "Y::Func" << std::endl;

}

};

void Test(X &x)

{

x.Func();

}

void main()

{

X;

Y;

Test(x); // X::Func

Test(y); // Y::Func

}

type mtype = "A"|"B"|"C";

interface iA1 {

name: mtype;

age: number;

}

interface iA2 {

name: mtype;

score: number;

}

interface iA3 {

name: mtype;

size: number;

show: ()=>string;

}

class A implements iA1, iA2, iA3 {

public age: number;

public score: number;

public size: number;

constructor(public name: mtype) {

this.size=99;

this.score = 88;

this.age = 46;

}

public show() {

return `Show Name:${this.name} - age:${this.age} - size:${this.size} - score: ${this.score}`;

}

public add(x:number) {

this.score +=x;

}

}

interface iA5 extends A {

}

class A7 {

public zoom: number;

}

class A6 extends A7 implements iA5, iA1, iA2 {

public age: number;

public score: number;

public size: number;

constructor(public name: mtype) {

super(); - 对于继承必须的

this.size = 99;

this.score = 88;

this.age = 46;

}

public show() {

return `Show Name:${this.name} - age:${this.age} - size:${this.size} - score: ${this.score}`;

}

public add(x: number) {

this.score += x;

}

}

let a:A6 = new A6("A");

console.log(a.show());

**keyof**

interface person {

name:string;

age: number;

}

function Get<T extends keyof person>(t: T) {

console.log(`t=${t} type=${typeof(t)}`);

}

Get("name");

输出： t=name type=string

**相当于：**

function Get(t: “name” | “age”) {

console.log(`t=${t} type=${typeof(t)}`);

}

**in**

type yy = 'AA'|'BB';

type man = { - in 只能使用 type, 不能使用 interface

[x in yy]: string;

}

let ar: man = {"AA":"Hello", "BB": "World"}

console.log(ar);

输出：

{ AA: 'Hello', BB: 'World' }

### 类型遍历

当你已知某个类型范围的时候，可以使用 in 和 keyof 来遍历类型，例如上面的 ChinaMobile 例子，我们可以使用 in 来约束属性名必须为三家运营商之一：

type ChinaMobilePhones = '10086' | '10010' | '10000'

interface ChinaMobile {

name: string;

website: string;

}

// 只能 type 使用， interface 无法使用

type ChinaMobileList = {

// 遍历属性

[phone in ChinaMobilePhones]: ChinaMobile

}

我们也可以用 keyof 来约定方法的参数

export type keys = {

name: string;

appId: number;

config: object;

}

class Application {

// 参数和值约束范围

set<T extends keyof keys>(key: T, val: keys[T])

get<T extends keyof keys>(key: T): keys[T]

}

**四、与HTML配合**

前端编程语言还是要和HTML配合的。下面给出一个例子，其中用到了继承与多态。

HTML部分：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>TypeScript</title>

<script src="app.js"></script>

</head>

<body>

<h1>TypeScript</h1>

<p><input id="btn1" type="button" value="动物呼吸" /></p>

<p><input id="btn2" type="button" value="鱼吐泡" /></p>

</body>

</html>

TypeScript部分：

class Animai {

name: string

eat(): void {

alert(this.name + '吃');

}

sleep(): void {

alert(this.name + '睡觉')

}

breath(): void {

alert(this.name + '呼吸');

}

constructor(name: string) {

this.name = name;

}

}

class Fish extends Animai {

constructor(name: string) {

super(name);

}

breath(): void {

alert(this.name + '吐泡');

}

}

function Breath(p: Animai): void {

p.breath();

}

window.onload = () => {

var a: Animai = new Animai('杰瑞');

var f: Fish = new Fish('迈克');

var btn1 = document.getElementById('btn1');

var btn2 = document.getElementById('btn2');

btn1.onclick = () => {

Breath(a);

}

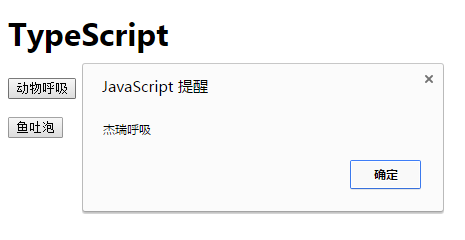
btn2.onclick = () => {

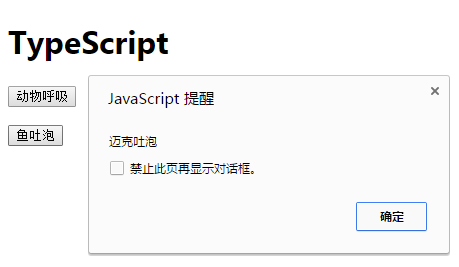
Breath(f);

}

};

效果如下：

[](http://image.codeweblog.com/upload/c/b6/cb66a564f59b7deb.png)

[](http://image.codeweblog.com/upload/f/28/f2843ba705b18929.png)

分类：[TypeScript](http://www.codeweblog.com/category/typescript/) 时间：2015-04-15 人气：**1218**

本文关键词： [typescript](http://www.codeweblog.com/tag/typescript/) [JavaScript](http://www.codeweblog.com/tag/javascript/) [类](http://www.codeweblog.com/tag/%e7%b1%bb/) [继承](http://www.codeweblog.com/tag/%e7%bb%a7%e6%89%bf/) [多态](http://www.codeweblog.com/tag/%e5%a4%9a%e6%80%81/)

# <http://blog.csdn.net/maomaolaoshi/article/details/78374506?locationNum=6&fps=1>

# <https://www.2cto.com/kf/201604/497764.html>

# <http://www.codeweblog.com/typescript%e7%ac%94%e8%ae%b0-2-%e7%b1%bb-%e7%bb%a7%e6%89%bf-%e5%a4%9a%e6%80%81/>

# TypeScript类、接口、继承

原创 2017年11月01日 17:20:38

* 标签：
* [typescript](http://so.csdn.net/so/search/s.do?q=typescript&t=blog) /
* [类](http://so.csdn.net/so/search/s.do?q=类&t=blog) /
* [接口](http://so.csdn.net/so/search/s.do?q=接口&t=blog) /
* [继承](http://so.csdn.net/so/search/s.do?q=继承&t=blog) /
* [类的修饰符](http://so.csdn.net/so/search/s.do?q=类的修饰符&t=blog)

       TS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，TS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。

### 类

       定义一个类

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

toString() {

return '(' + this.x + ', ' + this.y + ')';

}

}

使用这个类

let p=new Point(1,2);

需要注意的地方有以下几点：   
       ①类和模块内部默认采用严格模式，不需要使用 use strict 指定运行模式。

       ② 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，这一点与Java的类一致。

       ③必须使用 new 命令来调用 class ，否则将会报错。

       ④类不存在变量提升，只有先声明类，才能使用类。

       ⑤类的方法内部如果含有 this ，它默认指向类的实例。但是如果我们单独将其方法提取出来， this 值可能会指向当前运行的环境。为了防止这种事情的发生，我们可以使用箭头函数（箭头函数的 this 值指向初始化的函数）。

#### **public、private、protected和readonly**

       public、private、protected和readonly都是类的成员（属性）修饰符   
**public**   
       在TS里，成员都默认为public。被public修饰的属性，我们在类的内外都可以自由访问到这些被定义的属性。

class Animal {

public name: string;

public constructor(theName: string) { this.name = theName; }

}

new Animal("Cat").name;//Cat

**private**   
       当成员被标记成private时，它就不能在声明它的类的外部访问。

class Animal {

private name: string;

constructor(theName: string) { this.name = theName; }

}

new Animal("Cat").name;//Error!: Property 'name' is private and only accessible within class 'Animal'.

       TS使用的是结构性类型系统。 当我们比较两种不同的类型时，并不在乎它们从何处而来，如果所有成员的类型都是兼容的，我们就认为它们的类型是兼容的。

       这里的比较并非我们说得 == 或者 === 的比较，而是对期望值（结构）的比较。

class Animal1 {

name: string;

constructor(theName: string) { this.name = theName; }

}

class Animal2 {

name: string;

constructor(theName: string) { this.name = theName; }

}

//这样的写法是不会出错的

let a:Animal1=new Animal2("cat");

       但是被 private 或 protected 修饰的成员类型不一样。如果其中一个类型里包含一个private (或protected)成员，那么当另外一个类型中也存在这样一个private (或protected)成员， 并且它们都是来自同一处声明时，那么这两个类型是兼容的，否则是不兼容的。

class Animal1 {

private name: string;

constructor(theName: string) { this.name = theName; }

}

class Animal2 extends Animal1{

constructor(theName: string) {super(name); }

}

class Animal3 {

private name: string;

constructor(theName: string) { this.name = theName; }

}

let animal1: Animal1 = new Animal2("cat");//没问题。Animal1和Animal2的private修饰的成员变量name都来自于Animal1（都是来自同一处声明）。

let animal3: Animal1 = new Animal3("cat");//ERROR:Type 'Animal3' is not assignable to type 'Animal3'.

**protected**   
       protected 修饰符与 private 修饰符的行为很相似，但有一点不同，protected 成员在派生类中仍然可以访问。   
使用 private 修饰的父类成员，派生类无法访问。

class Person {

private name: string;

constructor(name: string) { this.name = name; }

}

class Employee extends Person {

constructor(name: string) { super(name)}

public sayName() {

return this.name;//ERROR!: Property 'name' is private and only accessible within class 'Person'.

}

}

let xiaoming = new Employee("xiaoming");

console.log(xiaoming.sayName());

       使用protected 修饰的父类成员，在派生类中仍然可以访问

class Person {

protected name: string;

constructor(name: string) { this.name = name; }

}

class Employee extends Person {

constructor(name: string) { super(name)}

public sayName() {

return this.name;

}

}

//派生类中仍能继续使用

let xiaoming = new Employee("xiaoming");

console.log(xiaoming.sayName());

**readonly修饰符**   
        readonly 关键字与 public 、 private 和 protected 不一样，它修饰的不是成员的访问权限，而是成员的再赋值权限。   
        使用readonly 关键字将属性设置为只读的。 **只读属性必须在声明时或构造函数里被初始化。**

class Octopus {

readonly name: string;

readonly numberOfLegs: number = 8;

constructor (theName: string) {

this.name = theName;

}

}

let dad = new Octopus("Man with the 8 strong legs");

dad.name = "Man with the 3-piece suit"; // 错误! name 是只读的.

#### **抽象类**

       抽象类做为其它派生类的基类使用。 它们一般不会直接被实例化。 不同于接口，抽象类可以包含成员的实现细节。   
       abstract关键字是用于定义抽象类和在抽象类内部定义抽象方法。

abstract class Animal {

abstract makeSound(): void;// 必须在派生类中实现

move(): void {

console.log('roaming the earch...');

}

}

**注意：**   
       ①抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。   
       ②抽象方法必须包含abstract关键字并且可以包含访问修饰符。

### 接口

       在传统的面向对象概念中，一个类可以扩展另一个类，也可以实现一个或多个接口。一个接口可以实现一个或多个接口但是不能扩展另一个类或接口。wiki百科中对 OOP 中接口的定义是：

在面向对象的语言中，术语 interface 经常被用来定义一个不包含数据和逻辑代码但是用函数签名定义了行为的抽象类型。

       但是对于TS来说，接口更重要的意义是对值所具有的 结构 进行类型检查。   
接口根据属性划分，可以划分成三类，一种是**必选属性**，另一种是**可选属性**，还有一种就是**只读属性**。

#### **必选属性**

       必选属性就是函数必须要有的属性。

interface PersonVaule{

name:string;

age:number;

}

function Person(person:PersonVaule){

this.name=person.name;

this.age=person.age;

}

//创建实例

var xiaoming=new Person({name:"xiaoming",age:18})

       类型检查器并不会检查属性的顺序，但是必须要必选属性。

var xiaoming2=new Person({age:18,name:"xiaoming"})//没有问题

var xiaoming3=new Person({name:"xiaoming"})//提示属性缺失:Property 'age' is missing in type '{ name: string; }'.

#### **可选属性**

       接口里的属性不全都是必需的。 有些是只在某些条件下存在，或者根本不存在。 可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

       带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个?符号。

interface AnimalVaule{

name?:string;

eat:string;

lifestyle?:string;

}

function Animal(animal:AnimalVaule){

this.name=animal.name;

this.eat=animal.eat;

this.lifestyle=animal.lifestyle;

}

let cat=new Animal({eat:"食肉动物",lifestyle:"昼伏夜出"});

可选属性好处有二：

       1. 可以对可能存在的属性进行预定义   
       2. 可以捕获引用了不存在的属性时的错误。

下面这个例子就出现了错误提示：

let dog=new Animal({eat:"适应性的肉食类动物",lifestle:"昼行夜伏"})//'lifestle' does not exist in type 'AnimalVaule'.

#### **只读属性**

       一些对象属性只能在对象刚刚创建的时候修改其值。 你可以在属性名前用readonly来指定只读属性:

interface Point {

readonly x: number;

readonly y: number;

}

       你可以通过赋值一个对象字面量来构造一个Point。 赋值后，x和y再也不能被改变了。

let p1: Point = { x: 10, y: 20 };

p1.x = 5; // error!

**readonly 和 const**   
       readonly 和 const 声明的变量或属性都不允许二次修改。这两个属性的使用区别在于是作为变量还是属性：   
做为变量使用的话用const，   
做为属性则使用readonly。

       接口不仅仅能描述对象的属性，还能描述**函数类型**，**可索引类型**和**类类型**。

#### **函数类型**

       为了使用接口表示函数类型，我们需要给接口定义一个调用签名。 它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

interface SearchFunc {

(source: string, subString: string): boolean;

}

let mySearch:SearchFunc=function(src,sub){

let result = src.search(sub);

return result > -1;

}

**注意**：   
       函数的参数会逐个进行检查，要求对应接口的位置上的参数类型是兼容的，无需名称一致。

#### **可索引类型**

       与使用接口描述函数类型差不多，我们也可以描述那些能够“通过索引得到”的类型，比如a[10]或ageMap["daniel"]。可索引类型具有一个 索引签名 ，它描述了对象索引的类型，还有相应的索引返回值类型。

**索引签名共有两种形式：字符串和数字。**

**数字索引签名**：

interface NN {[index: number]: number;}

let nn: NN = [1, 2];

interface NS {[index: number]: string;}

let ns: NS = ["1", "2"];

       上面例子里，我们定义了 NN 接口和 NS 接口，它们具有索引签名。 这个索引签名表示了当用 number 去索引 NN 或NS 接口 时会得到 number 类型或 string 的返回值。

**字符串索引签名**：   
       字符串索引签名能够很好的描述 dictionary 模式，并且它们也会确保所有属性与其返回值类型相匹配。

interface SS {[index:string]:string}

let ss: SS = {"A":"a", "B":"b"};

interface SN {[index: string]: number;}

let sn: SN = {"A":1, "B":2};

       你可以将索引签名设置为只读，这样就防止了给索引赋值：

interface ReadonlyStringArray {

readonly [index: number]: string;

}

let myArray: ReadonlyStringArray = ["Alice", "Bob"];

myArray[2] = "Mallory"; // error!

**索引的返回值可以不只一个，但是必须是同一个类型。**

interface NN {

[index: number]: number;

length:number;

name: string // 错误，`name`的类型与索引类型返回值的类型不匹配

}

let nn: NN = [1, 2];

**注意：** 如果有多个返回值，那么数字索引的返回值必须是字符串索引返回值类型的子类型。   
对于上述的解释，TS原话是这样的：

这是因为当使用number来索引时，JavaScript会将它转换成 string 然后再去索引对象。 也就是说用100（一个 number）去索引等同于使用”100”（一个 string）去索引，因此两者需要保持一致。

       虽然字面上的解释不明所以，但是我们通过例子可以去理解其含义。

class Animal {

name: string;

}

class Dog extends Animal {

breed: string;

}

//ERROR!: Numeric index type 'Animal' is not assignable to string index type 'Dog'.

interface NotOkay {

[x: number]: Animal;

[x: string]: Dog;

}

       对于上述例子， number 索引的返回值是父类Animal，而 string 索引的返回是子类 Dog。所以TS报错。

       如果修改成 number 索引的返回值是子类Dog，string 索引的返回值是父类 Animal，则毫无问题。

class Animal {

name: string;

}

class Dog extends Animal {

breed: string;

}

interface Okay {

[x: number]: Dog;

[x: string]: Animal;

}

#### **类类型**

       与C#或Java里接口的基本作用一样，TS也能够用它来明确的强制一个类去符合某种契约。

interface ClockInterface {

currentTime: Date;

}

class Clock implements ClockInterface {

currentTime: Date;

constructor(h: number, m: number) { }

}

**注意**：接口描述了类的公共部分，而不是公共和私有两部分。 它不会帮你检查类是否具有某些私有成员。

### 继承

TS允许我们通过extends关键字来 创建子类（实现继承）。   
下面这个例子，Dog 类继承自 Animal 类，在Dog 类中我们可以访问父类 Animal 的属性和方法。

class Animal {

name: string;

constructor(theName: string) { this.name = theName; }

}

class Dog extends Animal {

breed: string;

}

new Dog("mydog").name;//mydog

**注意：**包含构造函数的派生类必须调用super()，它会执行基类的构造方法。

抽象类：

interface iMan {

name: string;

age: number;

show():void;

add:(x:number, y:number)=>number;

}

abstract class aMan implements iMan {

constructor(public name: string, public age:number){

console.log(`Abstract(${this.name}, ${this.age})`);

}

abstract show():void;

abstract add:(a:number, b:number)=>number;

}

class Man extends aMan {

constructor() {

super("William", 46);

}

public show(){

console.log(`Show Man: ${this.name} - ${this.age}`);

}

public add = (x1:number, x2:number) => {

console.log(`Add ${x1} + ${x2} = ${x1+x2}`);

return x1+x2;

}

}

let m:Man = new Man();

m.show();

m.add(33, 44);

# typescript(九)--ts中泛型、泛型方法、泛型类、泛型接口

如题，本篇我们介绍写ts中的泛型。

什么是泛型呢？我们可以理解为泛型就是在编译期间不确定方法的类型(广泛之意思)，在方法调用时，由程序员指定泛型具体指向什么类型。泛型在传统面向对象编程语言中是极为常见的，ts中当然也执行泛型，如果你理解c#或java中的泛型，相信本篇理解起来会很容易。

泛型函数、泛型类、泛型接口。

function getValue<T>(...t:T[]) {

console.log(t);

console.log(t.length);

}

getValue<any>(10, 20, 30, "yes");

getValue("one", "two", "three");

getValue({id:100, yes:true}, {id:200, name:'will'}, {id:300, school:45});

[ 10, 20, 30, 'yes' ]

4

[ 'one', 'two', 'three' ]

3

[ { id: 100, yes: true },

{ id: 200, name: 'will' },

{ id: 300, school: 45 }

]

3

/\*

\* ts中泛型

\* 泛型就是在编译期间不确定方法的类型(广泛之意思)，在方法调用时，由程序员指定泛型具体指向什么类型

\*/

//1 泛型函数

/\*\*

\* 获取数组中最小值 （数字）

\* @param {number[]} arr

\* @returns {number}

\*/

function getMinNumber(arr:number[]):number{

var min=arr[0];

arr.forEach((value)=>{

if(value<min){

min=value;

}

});

return min;

}

/\*\*

\* 获取数组中最小值 （字符串）

\* @param {number[]} arr

\* @returns {number}

\*/

function getMinStr(arr:string[]):string{

var min=arr[0];

arr.forEach((value)=>{

if(value<min){

min=value;

}

});

return min;

}

console.log(getMinNumber([1, 3, 5, 7, 8]));//1

console.log(getMinStr(["tom","jerry","jack","sunny"]));//jack

/\*\*

\* 获取数组中最小值 (T泛型通用)

\* @param {T[]} arr

\* @returns {T}

\*/

function getMin<T>(arr:T[]):T{

var min=arr[0];

arr.forEach((value)=>{

if(value<min){

min=value;

}

});

return min;

}

console.log(getMin([1, 3, 5, 7, 8]));

console.log(getMin(["tom","jerry","jack","sunny"]));

//2 泛型类

class GetMin<T>{

arr:T[]=[];

add(ele:T){

this.arr.push(ele);

}

min():T{

var min=this.arr[0];

this.arr.forEach(function (value) {

if(value<min){

min=value;

}

});

return min;

}

}

var gm1= new GetMin<number>();

gm1.add(5);

gm1.add(3);

gm1.add(2);

gm1.add(9);

console.log(gm1.min());

var gm2= new GetMin<string>();

gm2.add("tom");

gm2.add("jerry");

gm2.add("jack");

gm2.add("sunny");

console.log(gm2.min());

/\*\*

\* 3 泛型函数接口

\*/

interface ConfigFn{

<T>(value:T):T;

}

var getData:ConfigFn=function<T>(value:T):T{

return value;

}

getData<string>('张三');

// getData<string>(1243); //错误

// 类似 Map<String,Object> Param 接口

interface Param{

[index:string]:any

}

//4 泛型类接口

/\*\*

\* page分页对象

\*/

class Page{

private currentPage:number=1; //当前页码 默认1

private pageSize:number=10;//每页条数 默认为10

private sortName:string; //排序字段

private sortOrder:string="asc"; // 排序规则 asc | desc 默认为asc正序

constructor(param:Param){

if(param["currentPage"]){

this.currentPage=param["currentPage"];

}

if(param["pageSize"]){

this.pageSize=param["pageSize"];

}

if(param["sortName"]){

this.sortName=param["sortName"];

}

if(param["sortOrder"]){

this.sortOrder=param["sortOrder"];

}

}

public getStartNum():number{

return (this.currentPage-1)\*this.pageSize;

}

}

class User{

id:number;//id主键自增

name:string;//姓名

sex:number;//性别 1男 2女

age:number;//年龄

city:string;//城市

describe:string;//描述

}

//泛型接口

interface BaseDao<T> {

findById(id:number):T;//根据主键id查询一个实体

findPageList(param:Param,page:Page):T[];//查询分页列表

findPageCount(param:Param):number;//查询分页count

save(o:T):void;//保存一个实体

update(o:T):void;//更新一个实体

deleteById(id:number);//删除一个实体

}

/\*\*

\* 接口实现类

\*/

class UserDao<User> implements BaseDao<User>{

findById(id:number):User{

return null;

}

findPageList(param:Param,page:Page):User[]{

return [];

}

findPageCount(param:Param):number{

return 0;

}

save(o:User):void{

}

update(o:User):void{

}

deleteById(id:number){

}

}

--

interface iCat<T,U> {

memo: T;

add(t:T, u:U):string;

}

class Cat<T,U> implements iCat<T,U> {

constructor(public memo:T, public note:U) {

console.log(`${this.memo} - ${this.note}`);

}

public add(a:T, b:U):string {

let u:U = null;

this.memo = a;

this.note = b;

return `T = ${a} U = ${b}`;

}

}

let cc: Cat<string, string> = new Cat("Thomas", "Jerry");

console.log(cc.add("Hellow", "world"));

let cccc: Cat<string, Date> = new Cat("HelloKitty", new Date());

console.log(cccc.add("Disney", new Date("2001-12-25 13:30")));

Thomas - Jerry

T = Hellow U = world

HelloKitty - Wed Dec 26 2018 23:39:56 GMT-0800 (Pacific Standard Time)

T = Disney U = Tue Dec 25 2001 13:30:00 GMT-0800 (Pacific Standard Time)

## 类型保护与区分类型

interface Fish {

swim():void;

}

interface Bird {

fly():void;

}

function Animal(pet: Fish|Bird) {

if( (<Fish>pet).swim ) { - 注意不能使用: typeof(pet)==’Fish’, typeof只对内建的类型有效

(pet as Fish).swim();

}

if( (<Bird>pet).fly ) {

(<Bird>pet).fly();

}

}

let mypet: Fish = {

swim: function() {

console.log("Fish can swim");

}

}

Animal(mypet);

输出： Fish can swim

let hispet:Bird = {

fly: function() {

console.log("Bird can fly");

}

}

Animal(hispet);

输出：Bird can fly

### 用户自定义的类型保护

interface iA<T> {

add(t: T):T;

IsString(t:any):t is string; - 其实 t is string 相当于 Boolean,直接使用boolean效果一样

IsNumber(t:any):t is number;

IsType(t:any): t is object;

}

class A<T> implements iA<T> {

constructor(public name:T){}

public add(t:T):T {

return this.name;

}

public IsString(t:any): t is string {

return typeof(t) === "string";

}

public IsNumber(t:any): t is number {

return typeof t === "number";

}

public IsType(t:any): t is object {

return t instanceof Object;

}

}

function show(x: number|string): x is number {

return typeof(x) === "number";

}

let aa : A<string> = new A("William");

console.log(aa);

console.log("aa IsType: " + aa.IsType(aa));

console.log("aa IsNumber: " + aa.IsNumber("100"));

console.log("aa IsNumber: " + aa.IsNumber(235));

console.log("Show : good " + show("good"));

console.log("Show : 1000 " + show(1000));

输出:

A { name: 'William' }

aa IsType: true

aa IsNumber: false

aa IsNumber: true

Show : good false

Show : 1000 true

---------------------------------------------------------------------------------------

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

}

---------------------------------------------------------------------------------------

interface hasLength {

length: number;

}

function loggingIdentity<T extends hasLength>(arg: T): T {

console.log(arg.length); // 现在我们知道它含有.length属性，并且不报错

return arg;

}

# [Typescript Mixins(混合)](https://www.cnblogs.com/ys-ys/p/5250594.html)

除了惯例的面对对象的思想，另一种较流行的通过可复用组件创建类的方法是将简单的类混合到一起。你可能对这种混合的方式比较熟悉或对Scala语言的特性有理解，这种模式在JavaScript社区也有一定的人气。

**混合案例**

在下面的代码中，我们展示了如何在TypeScript中混合模型，看完代码之后，再分析它的执行。

[复制代码](javascript:void(0);)

// Disposable Mixin

class Disposable {

isDisposed: boolean;

dispose() {

this.isDisposed = true;

}

}

// Activatable Mixin

class Activatable {

isActive: boolean;

activate() {

this.isActive = true;

}

deactivate() {

this.isActive = false;

}

}

class SmartObject implements Disposable, Activatable {

constructor() {

setInterval(() => console.log(this.isActive + " : " + this.isDisposed), 500);

}

interact() {

this.activate();

}

// Disposable

isDisposed: boolean = false;

dispose: () => void;

// Activatable

isActive: boolean = false;

activate: () => void;

deactivate: () => void;

}

applyMixins(SmartObject, [Disposable, Activatable])

var smartObj = new SmartObject();

setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////

// 在你代码的某处

////////////////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {

baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {

derivedCtor.prototype[name] = baseCtor.prototype[name];

})

});

}

### Typescript : async await

public async getValue(t:number):Promise<number> {

return new Promise<number>(

(res, rej)=>{

setTimeout(

()=> {

if(t>100)

res(t = t + 3000);

else

rej("Error Happen");

}, 2000);

}

);

}

可以返回 Promise

aa.getValue(66).then(a=>console.log("then: " + a), err=>console.log(err));

aa.getValue(66).then(a=>console.log("then: " + a)).catch(err=>console.log(err));

public async getValue(t:number):Promise<number> {

if(t>100) throw Error("Error Happen");

return t+3000;

}

也可以直接返回类型， 系统自动会Promise.resolve( return\_value )

# TypeScript namespace

export namespace myspace {

export var hello:string = "Lib Hello World";

export class Frog {

constructor(public name:string, public age:number){}

public show() {

console.log(`${this.name} - ${this.age}`);

}

}

}

export namespace hispace {

export let hello: Date = new Date();

}

import { myspace as Fg } from "./lib";

let aaa = new Fg.Frog("Tom", 45);

aaa.show();

console.log(Fg.hello);

console.log(Hg.hello);

输出：

Tom - 45

Lib Hello World

2018-12-28T01:18:14.612Z

# TypeScript 高级类型整理

2017年06月29日 15:55:32 [天马3798](https://me.csdn.net/u011127019) 阅读数：1045 标签： [TypeScript 高级类型整理](http://so.csdn.net/so/search/s.do?q=TypeScript%20%E9%AB%98%E7%BA%A7%E7%B1%BB%E5%9E%8B%E6%95%B4%E7%90%86&t=blog) 更多

个人分类： [TypeScript](https://blog.csdn.net/u011127019/article/category/6977261)

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/u011127019/article/details/73897087

**一、交叉类型（Intersection Types）**

交叉类型是奖多个类型合并为一个类型。这往我们可以把多个现有的多种类型叠加成为一种类型，它包含了所需的所有类型的特性。主要用于多个对象的混合。

例如， Person & Serializable & Loggable同时是Person和Serializable和Loggable。 就是说这个类型的对象同时拥有了这三种类型的成员。

将多个对象混合成一个交叉类型，这种操作用的情况应该比较少

1. function extend<T, U>(first: T, second: U): T & U {
2. let result = <T & U>{};
3. for (let id in first) {
4. (<any>result)[id] = (<any>first)[id];
5. }
6. for (let id in second) {
7. if (!result.hasOwnProperty(id)) {
8. (<any>result)[id] = (<any>second)[id];
9. }
10. }
11. return result;
12. }
13. class Person {
14. constructor(public name: string) { }
15. }
16. interface Loggable {
17. log(): void;
18. }
19. class ConsoleLogger implements Loggable {
20. log() {
21. // ...
22. }
23. }
24. var jim = extend(new Person("Jim"), new ConsoleLogger());
25. var n = jim.name;
26. jim.log();

**二、联合类型（UnionTypes）**

联合类型与交叉类型很有关联，但是使用上却完全不同。

联合类型，将类型使用 | 关联在一起。

联合类型，可以将一个变量指定为已知的默写类型，可以将函数的返回值指定为已知的默写类型。

当前一个函数需要传入string类型或者number类型，可以使用any，同样更推荐使用联合类型。

1. function padLeft(value: string, padding: string | number) {
2. *// ...*
3. }
5. let indentedString = padLeft("Hello world", true); *// errors during compilation*

返回类型中，使用联合类型

1. interface Bird {
2. fly();
3. layEggs();
4. }
6. interface Fish {
7. swim();
8. layEggs();
9. }
11. function getSmallPet(): Fish | Bird {
12. *// ...*
13. }
15. let pet = getSmallPet();
16. pet.layEggs(); *// okay*
17. pet.swim(); *// errors*

对于类型的私有部分

1. let pet=getSmallPet();
2. pet.layEggs();
3. *//对于两种类型的个字的特有的方法，编译异常*
4. *// pet.fly();*
5. *// pet.swim();*

**1.类型保护与区分类型，使用类型断言处理<>或者as**

1. *//使用类型断言，判断类型的理由部分*
2. *//使用类型断言，编写代码的时候有代码提示*
3. if((<Bird>pet).fly){
4. (pet as Bird).fly();
5. }else{
6. (<Fish>pet).swim();
7. }

**2.用户自定义类型区分，使用 is 谓词**

类型判断方法

1. function isFish(pet: Fish | Bird): pet is Fish {
2. return (<Fish>pet).swim !== undefined;
3. }

 谓词为 parameterName is Type这种形式，parameterName必须是来自于当前函数签名里的一个参数名。  
**3.typeof 类型保护，主要用于判断基础类型**

只有两种形式能被识别：typeof v === "typename"和typeof v !== "typename"，"typename"必须是"number"，"string"，"boolean"或"symbol"。 但是TypeScript并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

1. function padLeft(value: string, padding: string | number) {
2. if (typeof padding === "number") {
3. return Array(padding + 1).join(" ") + value;
4. }
5. if (typeof padding === "string") {
6. return padding + value;
7. }
8. throw new Error(`Expected string or number, got '${padding}'.`);
9. }

**4.instanceof 类型保护**

instanceof类型保护是通过构造函数来细化类型的一种方式。  
1.instanceof要求判断类型有构造函数，也就是对类类型可判断，如果是接口类型则不能使用

2.编译器会识别instanceof，在编写代码识别的时候，会有自动提示

1. interface Padder {
2. getPaddingString(): string;
3. }
4. class SpaceRepeatingPadder implements Padder {
5. repeat: string = '张三丰';
6. getPaddingString() {
7. return 'space';
8. }
9. }
10. class StringPadder implements Padder {
11. age: number = 15,
12. getPaddingString() {
13. return 'string';
14. }
15. }
16. function getRandomPadder(): Padder {
17. return Math.random() < 0.5 ?
18. new SpaceRepeatingPadder() :
19. new StringPadder();
20. }
21. let padder: Padder = getRandomPadder();
22. *//使用 instanceof 类型保护*
23. *//编译器自动识别，写代码时自动根据类型提示*
24. if (padder instanceof SpaceRepeatingPadder) {
25. console.log(padder.repeat);
26. }
27. if (padder instanceof StringPadder) {
28. console.log(padder.age);
29. }

**5.可以为null的类型，可选参数、可选属性**

TypeScript具有两种特殊的类型，null和undefined，它们分别具有值null和undefined. 我们在[基础类型](https://www.tslang.cn/docs/handbook/basic-types.html)一节里已经做过简要说明。 默认情况下，类型检查器认为 null与undefined可以赋值给任何类型。  
--strictNullChecks标记可以解决此错误：当你声明一个变量时，它不会自动地包含null或undefined。

默认情况下这个属性不启用。

 你可以使用联合类型明确的包含它们：

1. let s = "foo";
2. s = null; *// 错误, 'null'不能赋值给'string'*
3. let sn: string | null = "bar";
4. sn = null; *// 可以*
6. sn = undefined; *// error, 'undefined'不能赋值给'string | null'*

使用了--strictNullChecks，可选参数会被自动地加上| undefined:

默认情况下这个属性是启用的。

1. function f(x: number, y?: number) {
2. return x + (y || 0);
3. }
4. f(1, 2);
5. f(1);
6. f(1, undefined);
7. f(1, null); *// error, 'null' is not assignable to 'number | undefined'*

**三、类型别名 type**

类型别名会给一个类型启一个新的名字。

器别名不会新建一个类型。

类型别名可以使用泛型。

不能被extends和implements扩展。

1. type Name = string;
2. type NameResolver = () => string;
3. type NameOrResolver = Name | NameResolver;
4. function getName(n: NameOrResolver): Name {
5. if (typeof n === 'string') {
6. return n;
7. }
8. else {
9. return n();
10. }
11. }

**四、字符串字面量类型**

字符串字面量类型允许你指定字符串必须的固定值。在实际应用中，字符串字面量类型可以与联合类型，类型保护和类型别名很好的配合。

通过结合使用这些特性，你可以实现类似枚举类型的字符串

1. type Easing = "ease-in" | "ease-out" | "ease-in-out";
2. class UIElement {
3. animate(dx: number, dy: number, easing: Easing) {
4. if (easing === "ease-in") {
5. *// ...*
6. }
7. else if (easing === "ease-out") {
8. }
9. else if (easing === "ease-in-out") {
10. }
11. else {
12. *// error! should not pass null or undefined.*
13. }
14. }
15. }
17. let button = new UIElement();
18. button.animate(0, 0, "ease-in");
19. button.animate(0, 0, "uneasy"); *// error: "uneasy" is not allowed here*

五、可辨识联合（Discriminated Unions）

感觉不太好用。

**多态的this类型**

多态的this类型表示的是某个包含类或接口的子类型。 这被称做 F-bounded多态性。 它能很容易的表现连贯接口间的继承，比如。 在计算器的例子里，在每个操作之后都返回 this类型：

1. class BasicCalculator {
2. public constructor(protected value: number = 0) { }
3. public currentValue(): number {
4. return this.value;
5. }
6. public add(operand: number): this {
7. this.value += operand;
8. return this;
9. }
10. public multiply(operand: number): this {
11. this.value \*= operand;
12. return this;
13. }
14. *// ... other operations go here ...*
15. }
17. let v = new BasicCalculator(2)
18. .multiply(5)
19. .add(1)
20. .currentValue();