

# 目錄

Introduction	1.1
基础认知	1.2
基础类型	1.3
接口与类	1.4
函数	1.5
泛型	1.6
枚举与类型	1.7
Symbol 与模块	1.8
装饰器与混合对象	1.9
声明合并	1.10
tsconfig.json	1.11

# TypeScript 精通指南

10W+字长文帮你深入精通 TypeScript 语言。

让我们带着问题，去寻找答案。

我们的目标是玩烂 `ts`。

需要一定的 JS 基础，或者后端面向对象语言（JAVA、PHP、C#等都行）的基础

## ts 安装与更新

首先我们安装我们的 `typescript` 命令行工具，同样你也可以使用该命令更新版本。

```
npm install -g typescript
```

它会给我们安装 `tsc` 这个命令，它是 `typescript compile` 的缩写

## 婴儿的第一声啼哭

切换到你的工作目录，创建一个 `nodelover.me` 的文件夹，再在文件夹里面创建 `baby.ts` 文件

```
class Baby {  
    constructor() {  
        console.log('小宝贝正在哭泣，哇哇哇哇哇~~~')  
    }  
}  
  
let baby = new Baby();
```

这段代码非常的简单，声明了一个叫 `Baby` 的类，每次通过 `new` 实例化的时候，会自动调用 `constructor` 方法，也就是小孩的第一声啼哭。

编译 `ts` 文件，你会得到一个 `js` 文件

```
tsc baby.ts
```

你需要记住的是，`ts` 文件是不能直接运行的，能运行的是 `js` 文件，无论你的 `js` 文件是运行在浏览器中，或者是 `node` 环境中，也就是说 `ts` 的编译器并不关心你编译之后的 `js` 文件运行在哪。

运行的的 `baby.js` 文件

```
node baby.js
```

你会得到这样的显示

```
小宝贝正在哭泣，哇哇哇哇哇~~~
```

## 哪儿来的 `typescript`

看看我们的 `js` 文件

```
var Baby = (function () {
    function Baby() {
        console.log('小宝贝正在哭泣，哇哇哇哇哇~~~');
    }
    return Baby;
}());
var baby = new Baby();
```

对于很多后端语言的开发者来说，`function Baby(){}`  就是一个类，这非常令人难以理解，简直就是破坏宇宙的和平。

在 `typescript` 之前，还有一种语言叫 `coffeescript` 。

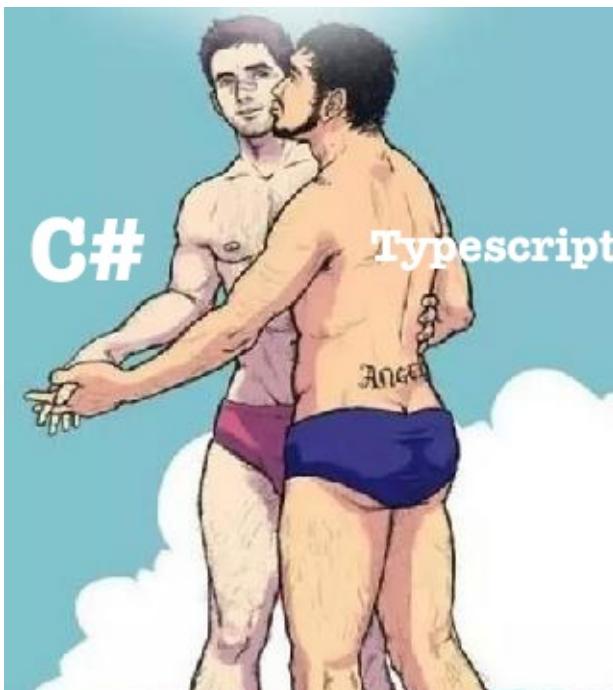


于是微软公司的员工就想，能不能我也搞一个像 `coffeescript` 这样的，把自己的语言文件编译成 `js` 再运行呢？所以微软为了维护宇宙的正义与和平，创造出了 `typescript` 语言。

# TypeScript

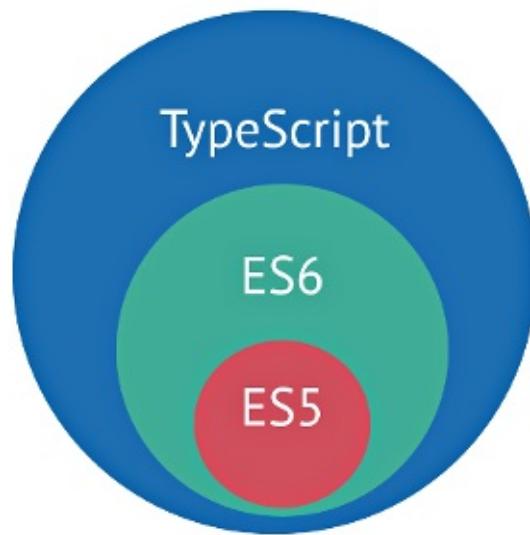
JavaScript for tools

因为微软主推的是 C# 语言，所以 typescript 的语法，基本上与 C# 的差异不大，简直海尔兄弟。



慢慢的 js 升级换代了，出了个 es6 ，微软波澜不惊的抽了根烟，淡淡的说：“好吧，你升级换代了我也不怕，我针对你的 es6 ，不仅吸收你的新特性，我还可以编译成 es6 的样子，分分钟秒杀你。”

所以说在 typescript 里面直接写 es6 语法的 js 基本是没有什么问题的。



于是微软公司的员工又开始想了，写 `js` 没有代码提示，开发效率简直就是战斗力为五的渣渣，所以 `typescript` 就沿用了 C# 那套代码提示，于是问题来了。

假如我要引入的代码是 `js` 文件而不是 `ts` 文件，那代码提示怎么办？

## 代码提示的秘密 `d.ts`

给我们的 `baby.ts` 添加一些代码

```
export class Baby{
    private _name: string;
    constructor(name: string){
        this._name = name;
        console.log('小宝贝正在哭泣，哇哇哇哇哇~~~')
    }

    static smile(){
        console.log('O(∩_∩)O哈！')
    }

    public getBabyName(): string{
        return this._name;
    }
}

export let baby = new Baby('Nico');
```

代码非常的简单，导出了一个 `Baby` 类，和一个叫 `baby` 的 `Baby` 实例。

`Baby` 包含一个私有的字段 `_name`，静态的方法 `smile`，公开的方法 `getBabyName`，在通过 `new` 调用 `constructor` 的时候，会初始化我们的 `_name`，而 `getBabyName` 就是拿到我们私有的 `_name`，之所以需要 `getBabyName`，是因为通过 `private` 关键字指定的私有字段和方法，在实例中是无法访问的。

再次编译 `ts` 文件，这次我们加上 `-d` 选项

```
tsc baby.ts -d
```

你会发现我们多出了一个 `baby.d.ts` 文件

大多数 `ts` 初学者会这样问：请问一下各位，如何在 `ts` 文件里面，引入已经写好的 `js` 文件呢？

答案就在这里，`d.ts` 文件，

```
export declare class Baby {
    private _name;
    constructor(name: string);
    static smile(): void;
    getBabyName(): string;
}
export declare let baby: Baby;
```

我们发现 `baby.ts` 里面所有的方法声明都被导入到了 `d.ts` 文件里面，而我们的 `typescript` 恰恰就是通过这个 `d.ts` 文件进行代码提示的。

因为我们的 `ts` 文件可以直接通过 `-d` 进行生成，而其他的 `js` 库呢？只有我们自己去看着 `API` 去写 `d.ts` 文件了。

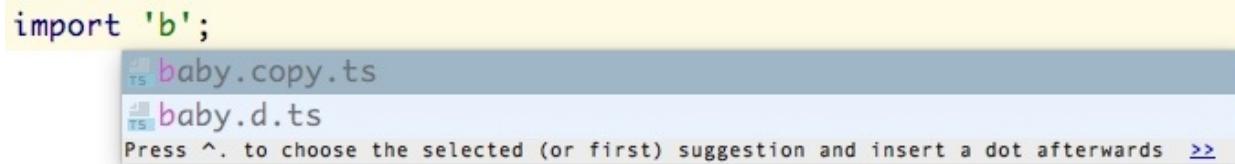
## 实验

- 重命名一下我们的 `baby.ts`，把它改成 `baby.copy.ts`

```
mv baby.ts baby.copy.ts
```

此时我们只剩下 `baby.js` , `baby.d.ts` 文件了。

- 新建 `main.ts` 文件当使用 `import` 导入的时候, `webstorm` 会自动提示你的 `baby.d.ts` ;

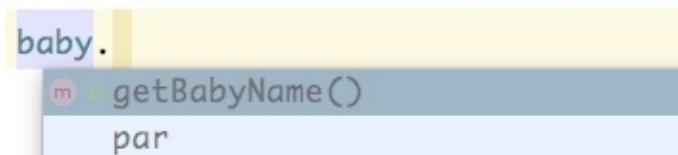


- 输入 `baby` ,他会提示你要导入哪一个文件里面的，这里我们选择 `baby.d.ts` ;



此时我们立刻就看到了 `getBabyName` 方法的提示

```
import {baby} from "./baby";
```



完整的代码应该像这样

```
import {baby} from "./baby";  
  
console.log(baby.getBabyName())
```

- 编译 `main.ts`

```
tsc main.ts
```

- 查看并运行 `main.js`

```
"use strict";
var baby_1 = require("./baby");
console.log(baby_1.baby.getBabyName());
```

```
node main.js
```

小宝贝正在哭泣，哇哇哇哇哇~~~

Nico

## 总结

这一节我们了解了

## **typescript** 的特点

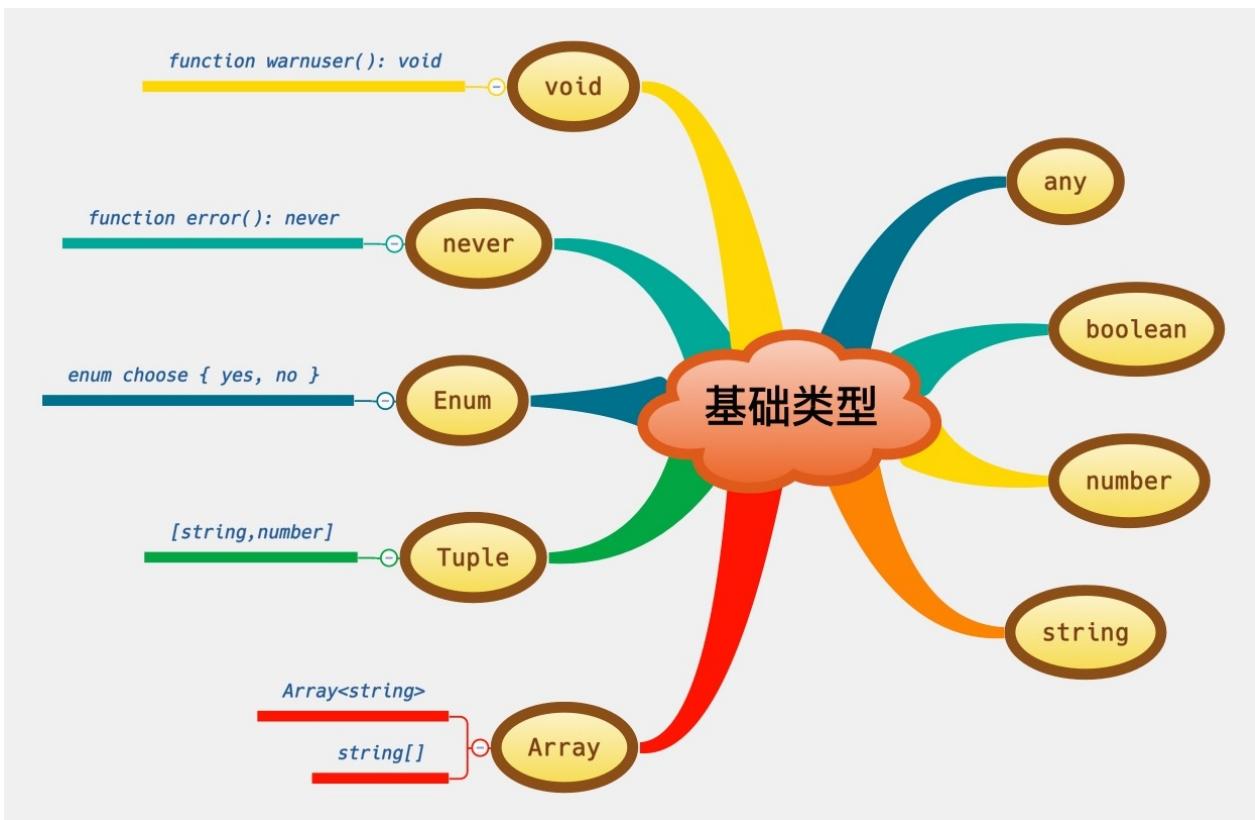
跟好兄弟 C# 类似，是静态类型的，拥有代码提示，ts 并不能运行，只有编译成 js 文件后，运行 js 文件

## **typescript** 的起源

因为后端人员写 js，简直反人类，所以微软搞出了 ts

## **typescript** 代码提示

秘密在于 d.ts 文件



新建我们的 `person.ts`，我们通过这一个文件，使用所有的基础类型。

## 实验

### 实验一

#### 代码

```
enum Choose { Wife = 1, Mother = 2} // 选择 妻子 还是 妈妈

function question(choose: Choose) : void{
    console.log('你老婆和你妈妈同时掉进水里你先救哪个?');
    console.log('你的选择是： ' + choose);
}

question(Choose.Mother)
```

#### 解释

首先用关键字 `enum` 声明了一个 `Choose` 的枚举，可以选择的是救落水的妈妈还是妻子。

在枚举 `enum` 中，我们可以对里面的选项赋值，假如不赋值的话，默认从0开始，一直自动递增。

## 结果

编译运行之后，我们发现 `choose.Mother` 的值其实是 2

```
你老婆和你妈妈同时掉进水里你先救哪个?  
你的选择是： 2
```

## 实验二

### 代码

```
class Person{  
    name: string; // 名字  
    age: number; // 年龄  
    labels: Array<string>; // 标签组  
    wives: string[]; // 妻子们  
    contact: [string, number]; // 元组 联系[联系地址，联系电话]  
    other: any; // 备注  
    saveMonth: boolean = true; // 是否救落水的妈妈  
    constructor(){}
    answer() : Choose{  
        if (this.saveMonth === false){  
            return Choose.Wife;  
        }  
        return Choose.Mother;  
    }
    hello() : void{  
        console.log('hello~ i'm ' + this.name);  
        // return undefined;  
        // return null;  
    }
    empty(){}
}
```

```

down() : never {
    while (true){}
    // throw new Error('error')
}

let zhangsan = new Person();

zhangsan.name = "张三";
zhangsan.age = 28;
zhangsan.labels = ["颜值逆天", "年轻多金"];
zhangsan.wives = ["一号", "二号", "三号"];
zhangsan.contact = ["北京xxxxxxxx别墅", 2];
zhangsan.saveMonth = false;
zhangsan.other = '不好不坏的人';

let len = (<string>zhangsan.other).length;

console.log(len);
question(zhangsan.answer());

zhangsan.hello();

console.log(zhangsan.empty());

```

## 解释

这里我们声明了一个 `Person` 的类，包含了一下属性。

- `name` 名称
- `age` 年龄
- `labels` 标签数组，每一个人都可以打上一些标签，代表一个人的特点
- `wives` 妻子数组，可以是一个，也可以是很多个，取决于个人的道德
- `contact` 联系方式，这是一个元组，语法跟数组类似，只不过提前规定了数组里面每一项的类型，所以说我们只能按照规定传入。
- `saveMother` 是否救妈妈
- `other` 备注，这是一个 `any` 类型表示任意类型

和以下方法

- `answer` 根据 `saveMother` 这个属性去回答之前的问题。
- `hello` 打招呼，我们限定了返回值是 `void`，你可以解开注释，你会发现 `void` 可以接受 `null` 和 `undefined` 的返回值。
- `empty` 这是一个空的方法，用来查看函数的默认返回值
- `down` 方法返回 `never`，表示不会结束，或者出错。（非常不常用）

这里我们构造了一个虚拟的年少多金的张三，给他赋值了一些属性。

为什么需要 `any` 类型？

假如我们需要用户输入一段值，它可以是数字还可以是字符串，请问我们该把它定义成什么类型呢

当我们去访问 `any` 类型的 `other` 的时候，`ts` 并不知道此时的 `other` 是 `string` 类型，所以它不会提示 `string` 的相关方法。

```
zhangsan.other = '不好不坏的人';  
zhangsan.other.|
```

当我们已经知道 `any` 的变量，具体为什么类型的时候怎么办？答案是，强制转换。

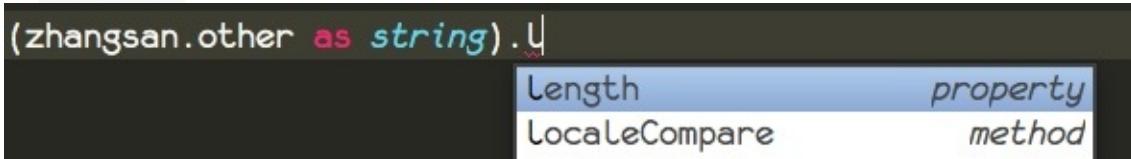
第一种方法

```
<string>zhangsan.other
```

第二种方法

```
zhangsan.other as string
```

此时我们随便选一种方法进行测试，我们发现，代码编辑器立刻就显示了 `string` 上面可以操作的一些方法。



结果

编译之后。我们可以得到一下结果

```
6  
你老婆和你妈妈同时掉进水里你先救哪个?  
你的选择是： 1  
hello~ i'm 张三  
undefined
```

说明 `empty` 默认方法返回的是 `undefined`

## 定义变量

```
let zhangsan = new Person();
```

这里我们使用 `let` 定义变量，而不是 `var`，因为 `var` 存在一些缺陷，所以被废弃掉了。

## 定义常量

```
const PI = '3.1415926'
```

不变的值就用 `const`，声明变量之前，请先问自己一句，这个值需要修改吗？

## 声明类型

```
let 变量名称 : 变量类型 = 变量的值  
let word : string = 'how'
```

：冒号这个标点符号，在语义中的作用就是，对冒号之前的东西进行解释和阐述。

比如

```
let 我： 宇宙超级无敌大帅哥 = DevOps
```

同样我们可以这样来读

```
宇宙超级无敌大帅哥的我是一个 DevOps
```

同理

```
string 类型的 word 是 'how'
```

变量类型可以是

- number
- string
- 某类型数组 -> 某类型[] Array<某类型>
- boolean
- 元组
- any

同样对于方法来说

```
hello() : void
```

在编程的世界里面 () 常常跟函数，方法一起出现，所以我们这里把它读作方法

同时 : 对方法和函数的解释就是告诉你它的返回值

```
hello 方法 返回 空
```

```
hello -> hello  
() -> 方法  
: -> 返回值  
void -> 空
```

读起来就是，hello方法的返回值是空

## 类型转换

```
<string>zhangsan.other
```

<string> 我把它看做为 `zhangsan.other` 打上一个标签。

我给你打上什么类型的标签，你就是什么类型。

就像这幅图一样，宝宝在尝了一口这个奶瓶里面的奶之后，立马给它打了一个 `<Bad>` 标签，表示不好喝，同时再给宝宝喂的时候，宝宝极力的晃脑袋。



```
zhangsan.other as string
```

而这种方式则比较简单， `as` 的中文意思就是作为，如同什么一样。

我让你作为是 `string` 就是 `string`，就是要让你像 `string` 一样。

```
zhangsan.other 如同 string 一样
```

同理我们来改一下套马杆的 飞驰的骏马像疾风一样

```
飞驰的骏马 as 疾风
```

这是不是很程序员！！

## 变量解构

小小的解构里面大学问，很有讲究。

相亲讲究个门当户对，解构讲究个对称。

### 增加你的代码

```
let [ a, b, c] = [2, 8 , 16]  
  
console.log(a, b , c)
```

结果

```
2 8 16
```

= 就像镜子一样，`let` 的这一边放什么变量，就可以得到右边的值。

### 栗子一

这一次我们以非诚勿扰为例，公开约~，刺激奔放。

此时我们的男嘉宾 `Alex` 上场了，他是来自.....，

此时我们跳过所有无关情节，最终`16`号为他留灯。

此时 `Alex` 就像这样

```
let [ , , Alex ] = [ 2, 8, 16 ]
```

结果 `Alex` 成功把`16`号带走了，回家生猴子去了。

### 修改你的代码

```
let [ Alex, ...X ] = [ 5, 7, 12, 18 ];  
  
console.log(Alex)  
console.log(X)
```

```
5  
[ 7, 12, 18 ]
```

## 栗子二

男嘉宾 Alex 因为房钱谁给的原因，跟女嘉宾打起来了，立刻就闹掰了，或许这是预谋好的，谁又知道呢？Alex 继续联系导演，导演决定让年少多金的 Alex 再次上场。

....

5号，7号，12号，18号为 Alex 留灯了。

Alex 直接走向了 5 号，对于其他的看都没看一眼，成功带走。

此时神秘嘉宾 X 先生发动大招（...）操作符，把剩余亮灯的女嘉宾全带走了。

记住 ... 后面要跟上变量的名字，这种好用的流氓大招，你应该要记住。

## 修改你的代码

这样的定律对于对象是同样适用的，记得是修改你的代码，要不然重复定义变量会报错的。

```
let { a, ...b } = { a:'string a', b: 'string b',c: 'string c' };  
  
console.log(a)  
console.log(b)
```

```
string a  
{ b: 'string b', c: 'string c' }
```

假如我们想换一个名字呢？

```
let { a, ...b } = { a:'string a', b: 'string b',c: 'string c' };

let x = a;
```

假如像上面这样，就感觉特别麻烦，所以我们可以像下面这样解构。

```
let { a: x, ...b } = { a:'string a', b: 'string b',c: 'string c'
};

console.log(x)
console.log(b)
```

当然我们还可以放到函数中去，当做函数的默认值。

```
function somefunc({ a: x, ...b } = { a:'string a', b: 'string b'
,c: 'string c' }) {
    console.log(x)
    console.log(b)
}
```

还可以这样，在调用的时候再传

```
function somefunc({ a: x, ...b } = {
    console.log(x)
    console.log(b)
})

somefunc({ a:'string a', b: 'string b',c: 'string c' })
```

其实原理还是对称，理他就是这个理，话操理不操。

## 小实验

代码

```

function saySomeThing1({ x , y } = {x : 0, y: 0}) {
    console.log(x, y)
}

function saySomeThing2({ x = 0 , y = 0 }) {
    console.log(x, y)
}

function saySomeThing3({ x = 0 , y = 0 } = {x:2,y : 2}) {
    console.log(x, y)
}

saySomeThing1()
saySomeThing1({ x: 3 , y: 3 })

saySomeThing2({})
saySomeThing2({ x: 3 , y: 3 })

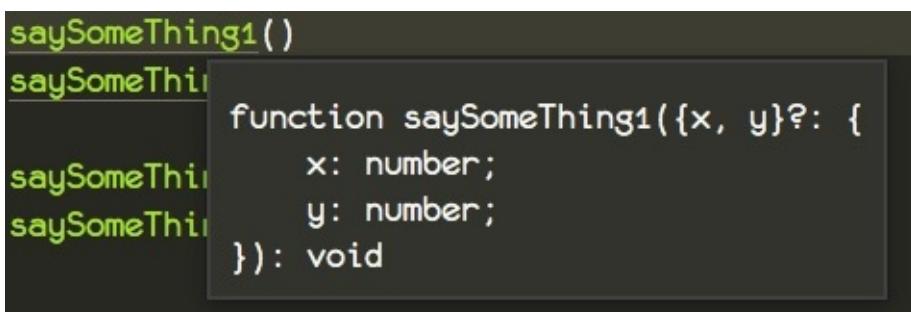
saySomeThing3()
saySomeThing3({})

```

## 解释

你可以看到这3种是不一样的，`saySomeThing2` 要求我们必须传递参数，其他的则不会。

把鼠标移动到 `saySomeThing1` 和 `saySomeThing2` 、 `saySomeThing3` 上面，它会出现这个函数的定义。



The screenshot shows two code snippets in a TypeScript editor. The first snippet defines a function `saySomeThing2` that takes an optional object parameter `{x, y}`. The second snippet defines a function `saySomeThing3` that also takes an optional object parameter `{x, y}`. Both functions return `void`. The code is as follows:

```
saySomeThing2({})  
saySomeThing3()  
  
function saySomeThing2({x, y}: {  
    x?: number;  
    y?: number;  
}): void  
  
function saySomeThing3({x, y}?: {  
    x?: number;  
    y?: number;  
}): void
```

这里我们只做简单的说明，更多信息放到后面说，`?` 代表的意思就是可选的意思。

因为第一个函数是一个默认对象，所以导致对象可以不传，第二个函数是有默认的属性，所以导致属性可传可不传。第三个就是前俩个的合体。

## 结果

```
0 0  
3 3  
0 0  
3 3  
2 2  
0 0
```

## 总结

这节完成之后，你应该明白了 `TypeScript` 的基本类型，和如何声明变量，以及变量的解构。关于 `...` 的更多用法后面会再次讲解。

## 为什么需要接口？

我们来看一下这个代码，对于眼神不好使的人来说简直就是遭罪，当然我这里只是简单的给了几个属性，假如有20个属性呢？20个使用这种结构的函数呢？

```
function somefunc1({ x = 0, y = 0 }: { x: number, y: number }) {  
    // ...  
}  
  
function somefunc2({ x = 0, y = 0, z = 0 }: { x: number, y: number, z: number }) {  
    // ...  
}
```

一切需要复制粘贴的代码，都可以通过代码去解决。

于是我们有了接口，就像神说，要有光一样亮趟。

```
function somefunc1({ x = 0, y = 0 }: pointer2d) {  
    // ...  
}  
  
function somefunc2({ x = 0, y = 0, z = 0 }: pointer3d) {  
    // ...  
}  
  
interface pointer2d {  
    x: number;  
    y: number;  
}  
  
interface pointer3d extends pointer2d {  
    z: number;  
}
```

接口的作用就是去描述结构的形态。

我们可以把 `interface` 做为语文里面的总结。

```
interface pointer2d {  
    x: number;  
    y: number;  
}
```

总结一下，其中二维坐标系点需要2个属性，一个是 `number` 类型的 `x`，一个是 `number` 类型的 `y`

```
function somefunc1({ x = 0, y = 0 }: pointer2d) {  
    // ...  
}
```

`somefunc1` 需要传入一个像二维坐标系点一样的对象。

连起来完整的就是，`somefunc1` 需要传入一个像二维坐标系点一样的对象，二维坐标系点需要2个属性，一个是 `number` 类型的 `x`，一个是 `number` 类型的 `y`

而 `extends` 就是总结的总结了。

```
interface pointer3d extends pointer2d {  
    z: number;  
}
```

读作，总结一下，`pointer3d` 首先要像 `pointer2d` 一样，需要2个属性，一个是 `number` 类型的 `x`，一个是 `number` 类型的 `y`。同时还需要一个新的属性 `z`

也可以理解为在阐述的基础上继续阐述，`pointer2d` 是论点一，比如说，吃蔬菜的好处 在这个论点给你说清楚了之后，继续升入讲 `pointer3d` 也就是，什么样的蔬菜包含什么样的维生素，就像这样由浅入深的阐述你的变量的结构形态。

## 描述类

上面的例子描述了函数传入对象，必须拥有的字段，这一次我们来正经的描述一下对象。

```
interface Db{
    host: string;
    port: number;
}
```

这个Db接口描述了，必须要有2个属性，一个是 string 的 host 和 number 的 port

此时我们把接口理解为合同，而 implements 就是履行。

```
interface -> 合同
Db      -> 合同名称
implements -> 履行
```

因为 MySQL 类履行了 Db 合同，是不是要执行里面的条款呀？

这里我们没有执行我们合同里面的条款，所以这里报错了，提示你的 host 条款上哪去了？

```
class MySQL implements Db{
}
Class 'MySQL' incorrectly implements interface 'Db'.
Property 'host' is missing in type 'MySQL'.
class MySQL
```

当我们做好 host 之后，编译器检测到你还有条框没有执行，所以又报错了，这里它又发问了，你的 port 上哪去了？会不会写程序！！自己写的规定都没实现。

```
class MySQL implements Db{
    host
}
Class 'MySQL' incorrectly implements interface 'Db'.
Property 'port' is missing in type 'MySQL'.
class MySQL
```

完善一下我们的代码

```
interface Db {  
    host: string;  
    port: number;  
}  
  
class MySQL implements Db {  
    host: string;  
    port: number;  
  
    constructor(host: string, port: number) {  
        this.host = host;  
        this.port = port;  
        console.log('正在连接 ' + this.host + ":" + this.port + "  
        的数据库....")  
    }  
}  
  
let mysql = new MySQL('localhost', 3306);
```

结果

```
正在连接 localhost:3306 的数据库....
```

## 属性修饰符

修饰符就想形容词一样，表示对属性的一些修饰，比如好看的，和难看的，以及夹在中间好难看的。

### **readonly**

当我们去描述一个类的时候，我们想要让某一个字段，只能被读取，而不能被修改，就像宪法一样，可看不可改。

同样你也可以把它理解为属性常量。

```
interface Person{
    readonly IdCard: string; // 身份证号
}

class Person implements Person{
    readonly IdCard: string = "42xxxxxxxxxxxxxx";
    constructor(){}
}
```

像只读属性，我们初始化的时候就必须给它赋值。

从下面编译好的 js 代码里面可以看出， interface 并不会产生任何实际代码

```
var Person = (function () {
    function Person() {
        this.IdCard = "42xxxxxxxxxxxxxx";
    }
    return Person;
}());
```

当然我们不仅可以用 interface 去描述带有构造器的 class ，我们还可以直接描述通过字面量 {} 构造的类变量；

```
interface Person{
    readonly IdCard: string; // 身份证号
}

let person: Person = { IdCard:'43xxxxxxxx' }
```

而生成的代码依旧不含有任何与 interface 相关的东西。

```
var person = { IdCard: '43xxxxxxxx' };
```

## private 与 protected

`private` 表示私有的变量，不能被其他任何访问，只归自己管。

```
class Dad{  
    protected surname; // 姓氏  
    private private_money; // 私房钱  
    constructor(){}
}  
  
class Son extends Dad {  
  
    constructor() {  
        super()  
        this.  
    }  
}
```

surname                      property

从这里可以看到，父亲的私房钱，只归自己管，哪怕儿子继承了父亲也不行，在儿子的构造器里面，仅仅可以取得到 `surname` 姓氏。

同时我们可以看到，在 `Son` 中可以访问得到 `protected` 的 `surname`，也就是说被 `protected` 修饰的是可以被继承的。

## public 和 默认的

修改一下我们的代码，增加一个 `public` 的属性，和一个没有任何修饰的属性。

```
class Dad{  
    protected surname; // 姓氏  
    private private_money; // 私房钱  
    public public_something;  
    default_something;  
    constructor(){}  
}  
  
class Son extends Dad {  
  
    constructor() {  
        super()  
        this.  
    }  
}
```

default_something	property
public_something	property
surname	property

从结果可以看出，继承，可以继承除了 `private` 的所有。

而对于通过 `new` 创建的实例来说。

```
class Dad{  
    protected surname; // 姓氏  
    private private_money; // 私房钱  
    public public_something;  
    default_default_something;  
    constructor(){}  
}  
  
class Son extends Dad {  
  
    constructor() {  
        super()  
    }  
}  
  
let d = new Dad()  
d.public_something = 'some_thing';  
d.default_something = 'default_thing'  
  
let s = new Son()  
s.  
    default_something    property  
    public_something     property
```

我们可以看到，实例只能访问 `public` 和默认的属性，同时也说明了默认就是 `public`，所以 `public` 可以省略不写。

完整的代码如下

```
class Dad{  
    protected surname; // 姓氏  
    private private_money; // 私房钱  
    public public_something;  
    default_something;  
    constructor(){}
}  
  
class Son extends Dad {  
    constructor() {  
        super()  
    }
}  
  
let d = new Dad()  
d.public_something = 'some_thing';  
d.default_something = 'default_thing'  
  
let s = new Son()  
s.public_something = 'some_thing';  
s.default_something = 'default_thing'
```

记得继承 `Dad` 的 `Son` 必须要先调用 `super()`，`super()` 表示父类的构造器，先有父亲，后有儿子。

对于属性修饰符你可以这么理解。

`class` 代表着一个家族成员，`extends` 表示血缘关系，就像上面的父亲与儿子。

`private` 是属于家族成员的私有物品，私人空间，别人是不能看到的，除非自己告诉别人，通过方法返回。

`protected` 表示家族资产，比如姓氏，某一宝物，古董。

`public` 表示共有资产，谁想拿，去问 `class` 的示例拿就好了。

## 可选属性

这个比较简单，就是一个 `?` 就行，表示可传，可不传。

它的语义就是强制需要传递这个参数吗？不强制需求。

```
interface Person{
    readonly IdCard: string; // 身份证号
    name?: string;
}

let person: Person = { IdCard: '43xxxxxxxxx' }
```

name	property
ng2-pipe	Angular 2 Pipe Snippet
ng2-import	Angular 2 Import Snippet
ng2-service	Angular 2 Service Snippet
ng2-component	Angular 2 Component Snippet
ng2-subscribe	Angular 2 Subscribe Snippet
ng2-http-get	Angular 2 HTTP Get Snippet

这里我们定义了一个可选的 `name` 属性，当我们在 `IdCard` 后面的属性，继续加的时候，编辑器自动提示了有一个 `name` 的 `property`（属性）。

```
interface Person{
    readonly IdCard: string; // 身份证号
    name?: string;
}

let person: Person = { IdCard: '43xxxxxxxxx' }
```

哪怕我们不传也是不会报错的。

这些修饰符，既然可以修饰类，那必然可以修饰接口上面的属性。

假如我们需要一个可以添加属性的 **interface** 怎么办呢？

```
interface Person{
    readonly IdCard: string; // 身份证号
    name?: string;
}

let person: Person = { IdCard: '43xxxxxxxxx' }

function getPerson(p: Person) {
    console.log(p);
}

getPerson({ IdCard: 's', b: 2 })
```

`getPerson` 这个函数要求我们传入的对象需要符合 `Person` 合同，当我们添加一些其他没有在合同里面定义的条款的时候，就会报错。

所以我们需要修改一下我们的合同，给它定义一下对于新增的条款，有些什么限制。

```
interface Person{
    readonly IdCard: string; // 身份证号
    name?: string;
    [propName : string]: any;
}

let person: Person = { IdCard:'43xxxxxxxx' }

function getPerson(p: Person) {
    console.log(p);
}

getPerson({ IdCard: 's', b : 2 })
```

我们在 `js` 语言中，访问一个对象的属性可以通过 `.` 去访问，还可以通过 `['属性名']` 这样的形式去访问。

```
let some2 = { a: 1, b: 2 }

some2.|  
  a          property  
some2|b          property
```

```
let some2 = { a: 1, b: 2 }

some2.a

some2[ '|'  
  interface.js  
  interface.ts  
  person.js  
  person.js.map  
  person.ts  
  a          property  
  b          property
```

我们可以看到，这2种都是有代码提示的，说明都是可行的。再看一看我们 `interface` 里面的 `[propName : string]: any;`

他们之间存在着 `[]` 的联系，放心这不是卡巴斯基与巴基斯坦的巴基联系，而是很大的联系。

`[propName : string]: any;`

`[]` 里面就限定了属性名的类型，而后面的 `any` 就限定了属性值的类型。

这个 `propName` 是可以随意修改的。

接下来我们看看不正经的代码。

```
interface Person {
    readonly IdCard: string; // 身份证号
    name?: string;
    [propName: number]: any;
}

let person: Person = { IdCard: '43xxxxxxxx' }

function getPerson(p: Person) {
    console.log(p);
}

getPerson({ IdCard: 's', 0: 2 })
```

这是不是没有问题？这个还算正常吧，限定了为 `number`，也没报错。

但是得到的结果是 `'0'`，不要大惊小怪这是正常的。

```
{ '0': 2, IdCard: 's' }
```

我们在 Chrome 里面测试一下 JS

```
> var a = { 0 : 1 }
< undefined
> a
< > Object {0: 1}
> a.0
✖ Uncaught SyntaxError: Unexpected number
> a['0']
< 1
>
```

其实 js 对象是不允许你设置属性名为 `number` 的，他会自动转换为字符串。

此时我们再添加一个属性，这报错了。非常正常是吧。

`interface` 仅仅只是在 `ts` 层面起了作用。

```
getPerson({ IdCard: 's', 0: 2, some: '22' })
```

接下来我们看看这个

```
interface Person {
    readonly IdCard: string; // 身份证号
    name?: string;
    [propName: string]: any;
}

let person: Person = { IdCard: '43xxxxxxxx' }

function getPerson(p: Person) {
    console.log(p);
}

getPerson({ IdCard: 's', o: 2, some: '22' })
```

诶，这就很奇怪了，我明明给的是 number 为什么不报错呢？

并且装换出来的 js 代码还是原样。

```
function getPerson(p) {
    console.log(p);
}
getPerson({ IdCard: 's', o: 2, some: '22' });
```

其实答案就是隐身转换。数字可以转换成字符串，而字符串不一定能转换成数字，比如 'xx1' 怎么转成数字，请问 x 该转成几？

也就是说，你所给的类型只要可以转换成合同里面的类型，我就认为你是对的，我比较明智，是个老司机，我懂你各种隐含意思。

还有一点就是，对于你声明的属性，假如你不用字符串的 '' 去包裹起来，js 编译器会帮你去做，从下面的代码你可以看出来。

```
> var p1 = { name: 'p1' }
< undefined
> var p2 = { 'name': 'p1' }
< undefined
> p1
< Object {name: "p1"}
> p2
< Object {name: "p1"}
> p1 == p2
< false
> p1 === p2
< false
> p1.name === p2.name
< true
```

## 描述函数

描述函数，我们只能使用一个变量接受匿名函数的引用，而不能通过 `function` 去创建一个具体有名称的函数。

```
interface Db {
    host: string;
    port: number;
}

interface InitFunc{
    (options: Db) : string;
}

let myfunc : InitFunc = function (opts: Db) {
    return '';
}
```

有了前面的经验，理解这个应该不难。

我们之前提到过 `()` 代表函数调用，对于这个 `interface`，我们这样读。

`InitFunc` 接口规定，它的函数调用需要传递一个 `Db` 合同、约束的 `options` 对象，返回一个 `string` 类型的值。

同时你也可以看到 `interface` 只是约束类型并不约束你的变量名。

## 描述可实例化

正常的理所当然的，我们会认为下面的代码是正确的。

这是初学者经常会犯的错误。

```
interface MyDate {
    new (year: string, month: string, day: string): void
}

class DateClass implements MyDate {
    constructor // Class 'DateClass' incorrectly implements interface 'MyDate'.
    Type 'DateClass' provides no match for the signature 'new (year: string, month: string, day: string): void'
    class DateClass
}
```

错误的代码我就不放上来了，免得误导大家，现在把正确的代码放在这里。

```

interface MyDateInit {
    new (year: string, month: string, day: string) : MyDate;
}

interface MyDate {
    year: string;
    month: string;
    day: string;
}

class DateClass implements MyDate {
    year: string;
    month: string;
    day: string;
    constructor(year: string, month: string, day: string) {
        this.year = year;
        this.month = month;
        this.day = day;
        return this;
    }
}

function getDate(Class: MyDateInit, { year, month, day } ) : MyDate{
    return new Class(year, month, day);
}

getDate(DateClass, { year: '2017', month: '12', day: '1' });

```

通常描述一个类的构造器和字段方法是分开来的。

`MyDateInit` 描述的就是我们的构造器

`MyDate` 描述的就是我们的类

就像前面的描述函数一样，没法用 `function somefunc(){}` 去接受实现合同一样。同样我们没办法在 `class someClass` 上面去实现构造器的合同。

```
let SuccessExample : MyDateInit = DateClass
```

具体原因是，`interface` 描述的是实例化后的类，也就是`{name:'some', age:22}` 这样的类型。而`constructor` 方法是属于静态方法。

你可以理解`new Class()` 其实就是去调用了`class.constructor()` 方法，是的没错，是方法。

重点是方法2个字，`new (year: string, month: string, day: string) : MyDate;` 在这段代码里面是不是有`()` 这个符号，这个代表着方法，而`new` 则代表着可实例化的意思。

合起来就是，`MyDateInit`描述了一个可以实例化的对象，并且它的构造器需要一个`string` 类型的`year`、`month`、`day` 参数，返回一个实现了`MyDate` 接口的类。

从js源码上可以看出，这个`new()` 真正意义上面的描述的是以下代码，描述的是一个函数方法。

```
var DateClass = (function () {
    function DateClass(year, month, day) {
        this.year = year;
        this.month = month;
        this.day = day;
        return this;
    }
    return DateClass;
}());
```

同时我也尝试这么写，通过`function` 去构造，我发现约束类型不对。

```
131
132 let errorExample : MyDateInit = function(year: string, month: string, day: string) : MyDate{
133     this.year = year
134     this.month = month
135     this.day = day
136     return this as MyDate;
137 };
138
139
/Users/Yugo/Desktop/nodeLover.me/interface.ts [4 errors]
(132, 6) Type '(year: string, month: string, day: string) => MyDate' is not assignable to type 'MyDateInit'.
Type '(year: string, month: string, day: string) => MyDate' provides no match for the signature 'new (year: string, month: string, day: string): MyDate'
```

我写出来的类型签名是下面这个，而`=>` 代表着函数的返回值。

```
(year: string, month: string, day: string) => MyDate
```

相当于编译器告诉我，我描述的是类的构造器，你把一个函数给我是怎么个意思？是不是想打架！来啊，心平气和的干一架~ 二营长，把他x的意大利炮拉出来！！

经过这样修改之后，就可以正常运行了。也就是去掉 new

```
interface MyDateInit2 {  
    (year: string, month: string, day: string) : MyDate  
}  
  
let ExamplePlus : MyDateInit2 = function(year: string, month: string, day: string) : MyDate{  
    this.year = year  
    this.month = month  
    this.day = day  
    return this as MyDate;  
};
```

哪怕你想通过限定 constructor 的参数，来限制构造器依旧是不行的。

```
interface test{
    constructor(year: string, month: string, day: string);
}

// 错误例子
// class a1 implements test{
//     constructor(year: string, month: string, day: string){
//     }
// }

// 错误例子
// let a2 : test = class test{
//     constructor(year: string, month: string, day: string){
//     }
// }

let a3 : test = {
    constructor(year: string, month: string, day: string){

    }
}
```

ts 虽然支持直接 new function 但是， function 必须是返回值为 void 的函数。

```
interface test{
    constructor(year: string, month: string, day: string): void;
}

let a3 : test = {
    constructor(year: string, month: string, day: string){

    }
}

let cc = new a3.constructor('', '', '')
```

其实关于 new() 最贴切与最精简的例子还行下面的代码。

```

let some : MyDateInit = class SomeDate implements MyDate {
    year: string;
    month: string;
    day: string;
    constructor(year: string, month: string, day: string) {

    }
}

```

## 描述混合类型

混合类型通常出现在第三方js库的d.ts文件上面，在我们写d.ts文件的时候可能需要。

```

interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) {console.log
('start is ' + start)};
    counter.interval = 123;
    counter.reset = function () {console.log('do you want reset
counter?')};
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;

console.dir(c)

```

我们把官网的代码拿下来，小小的修改了一下。

`Counter` 描述的是一个函数，并且它有静态的 `interval` 属性，和静态的 `reset` 方法。

`getCounter` 就是一个工厂函数，每次访问他，都可以得到一个被 `Counter` 接口修饰的函数。

并且我们通过 `tsc -d` 生成一下我们的 `d.ts` 文件。

```
interface Counter {  
    (start: number): string;  
    interval: number;  
    reset(): void;  
}  
declare function getCounter(): Counter;  
declare let c: Counter;
```

假如我们想要定义实例方法呢？

让 `counter` 函数返回一个我们定好的接口即可

```
interface couterInstance{
    start: number;
}

interface Counter {
    (start: number): couterInstance;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) {
        console.log('start is ' + start)
        this.start = start;
    };
    counter.interval = 123;
    counter.reset = function () {console.log('do you want reset
counter?')};
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;

console.dir(c)
```

编译之后，新建一个 `index.html` 文件

```
<meta charset="utf-8">
<script src=".//interface.js"></script>
```

用浏览器打开它，打开控制台，把鼠标移动到函数的上面，右键，选择 `store as global variable`

```
正在连接 localhost:3306 的数据库....  
▶ Object {0: 2, IdCard: "s", some: "22"}  
start is 10  
do you want reset counter?  
▶ function cou
```

查询“counter”

Store as Global Variable  
Show Function Definition

复制

语音  
服务

通过 `new` 去实例化这个函数，我们就可以看到有一个 `start` 属性

```
▶ function counter(start)  
> temp1  
< function (start) {  
    console.log('start is ' + start);  
    this.start = start;  
}  
> let a = new temp1(5)  
start is 5  
< undefined  
> a  
◀ ▶ counter {start: 5}  
▶ |
```

## 总结

我希望这个总结你来写，写与不写选择都在于你。

就像有的人会选择最坎坷的走向优秀的道路，而有的人不会。

最好把故事、场景都自己再阐述一篇，代码都读一篇。

故事：`public` 就是公共资产，`private` 就是私人物品，比如爸爸的私房钱，`protected` 就是家族资产。

读代码：

```
new (year: string, month: string, day: string) : MyDate
```

表示可以实例化的对象，并且它的构造器需要一个 `string` 类型的 `year` 、 `month` 、 `day` 参数，返回一个实现了 `MyDate` 接口的类。

## 函数声明

首先新建你的 `func.ts` 文件

函数声明有俩种方式，一种是拥有具体函数名字的，一种是没有名字的。

```
let func1 = function () {
    console.log('func1')
}

function func2() {
    console.log('func2')
}

let func3 = function func4() {
    console.log('func3 && func4')
}

console.log(func3 === func4)
```

Cannot find name 'func4'.  
any

从这段例子我们可以看到。

- `func1` 是一个匿名函数，因为 `function` 关键字后面没有跟着名字，而是使用一个变量来保存这个匿名函数的引用。
- `func2` 是一个拥有具体名称的函数
- `func3` 和 `func4` 是俩种的混合，这里报了一个错误，说没有找到 `func4`，说明匿名函数的优先级要比具体名称的函数高。

## 函数修饰与函数类型

对于函数来说，我们修饰它们什么东西呢？

参数类型，返回值类型。它们2个合起来就修饰了函数的类型。

```
let add1 = function(x: number, y: number) : number {
    return x + y;
}

function add2(x: number, y: number) : number {
    return x + y;
}
```

以上就是我们函数的俩种写法，修饰参数，在参数名后面加`:`修饰类型即可，而修饰返回值`() :`则在方法`()`的后面加`:`即可，因为这个值需要在方法调用之后才会有，所以要写在后面。

具体怎么读，就不再赘述，根据前面的经验，你应该知道。

我们知道，尽管我们没有描述`a`的类型，`let a = 2`这样的代码是不会报错的，因为编译器已经推断了`a`的类型。

```
let add1 = function(x: number, y: number) : number {
    return x + y;
}
```

```
function add2(x: number, y: number) : number {
    return x + y;
}
```

把你的鼠标放在`add1`和`add2`上面，就可以看到函数的类型，我们发现匿名函数与具名函数的类型是有一定差异的。

在`ES6`中`() => {}`代表着匿名函数，这是一种新语法，`ts`同样兼容它。

```
let add3 = (x, y) => x + y

let add4 = (x, y) => { return x + y }

let add5 = (x, y) => { x + y }
```

```
let add3 = (x, y) => x + y
```

```
Let add3: (x: any, y: any) => any
```

```
let add4 = (x, y) => { return x + y }
```

```
Let add4: (x: any, y: any) => any
```

```
let add5 = (x, y) => { x + y }
```

```
Let add5: (x: any, y: any) => void
```

我们可以看到 ts 都帮我们把函数的类型给推导了出来，同时从上面的例子我们可以看出有没有 {} 的区别，没有 {} 的时候，会把这一条语句的值作为函数的返回值。

所有我们完整的，拥有函数类型的推导一个像这样。

```
let add1 : (a: number, b: number) => number = function(x: number, y: number) : number {
    return x + y;
}
```

而具名函数我们是没法通过这种语法修饰的，我们可以看到上面的代码，我们用了 a 、 b ，这里只是一个名字而已， ts 仅仅只会去匹配参数的类型，而不会限定你参数名具体叫什么。

假如我们强行修饰的话！我只能说强行装逼会进医院的。

```
let add6 : (a: number, b: number) : number = function add2(x: number, y: number) : number {
    return x + y;
} (Local function) add2(x: number, y: number): number
```

就会像这样，尽管 add2 自动推断出来的类型是 (x: number, y: number) : number ，当我们使用 = 接受这个类型的时候，还是个匿名函数，所以我们这里 () : 会报错，这样做会忽略掉 add2 推断出来的类型，这种做法没有任何意义。

## 函数

```
32 let add6 : (a: number, b: number) : number = function add2(x: number, y: number) : number {
33     return x + y;
34 }
35
/Users/Yugo/Desktop/nodeLover.me/func.ts [1 errors]
(32, 35) '=>' expected.
```

错误提示我们应该把 : 改成 =>

这里我们把 => 读作流出，他就像流程图里面的流程。

```
(x: number , y: number) => number
```

读作，当我们访问需要传递一个 number 的 x 和 y 参数的匿名函数时候，会流出一个一个 number 的返回值。

哎，我邪恶了，想到了不好的东西。流鼻涕好邪恶。

假如你真的需要修饰语描述具名函数，你应该用 interface

```
interface Add{
    (x: number, y: number) : number
}

let add6 : Add

add6 = function add2(x: number, y: number) : number {
    return x + y;
}
```

对于函数的修饰来说，我们可以写在左边，也可以写在右边

```
let add7 : (x: number, y : number) => number = function (x, y) {
    return x + y;
}
```

```
let add8  = function (x: number, y : number) : number {
    return x + y;
}
```

```
let add7 : (x: number, y : number) => number = function (x, y) {
}
  let add7: (x: number, y: number) => number
```

```
let add8 = function (x: number, y : number) : number {
}
  let add8: (x: number, y: number) => number
```

类型都是可以正常推断出来的。

## 参数的可选与默认值

可选的参数是不能放在第一个的，除非你所有参数都是可选的。

```
let add9 = function(x?: number , y: number) : number{
  return 250;
}
  A required parameter cannot follow an optional parameter.
  (parameter) y: number
```

给参数添加默认值，直接写 `=` 即可，同时因为直接就赋值了，所以 `ts` 是可以推断出类型的，`number` 是可以省略的。

```
let add9 = function(x: number = 10 , y: number) : number {
  return 250;
}
```

## 剩余参数

你应该记得我们的 `...` 大招，一波全部带走。

系统默认是给我们推断成限制最小的 `any[]`

```
let add10 = function (x: number, ...y) {
  console.log(x)
  console.log(y);
}
  (parameter) y: any[]
```

贴上代码

```
let add10 = function (x: number, ...y) {  
    console.log(x)  
    console.log(y);  
}  
  
add10(1, 2, 3, 5, 9, 6, 8)
```

打印出结果

```
1  
[ 2, 3, 5, 9, 6, 8 ]
```

这是没有问题的吧，我们再来学点关于 `...` 的新内容，循序渐进。

`...` 还可以展开数组和展开对象，这样我们就可以实现拷贝数组和对象

```
const arr = [1, 2, 3]  
  
const copyArr1 = [...arr, 4, 5]  
const copyArr2 = [4, 5, ...arr]  
  
const obj = { a: 1, b: 2 };  
  
const copyObj1 = { c: 3, a: 5, ...obj }  
const copyObj2 = { c: 3, ...obj, a: 5, d: 20 }  
  
console.log(copyArr1)  
console.log(copyArr2)  
  
console.log(copyObj1)  
console.log(copyObj2)
```

```
...arr = 1, 2, 3  
...obj = a: 1, b: 2
```

```
[ 1, 2, 3, 4, 5 ]  
[ 4, 5, 1, 2, 3 ]  
{ c: 3, a: 1, b: 2 }  
{ c: 3, a: 5, b: 2, d: 20 }
```

当 `...` 面对的是一个数组的时候，把 `...arr` 写在前面，它的值就位于数组的前面，而写在后面，它的值也就位于数组的后面。

当 `...` 面对的是一个对象的时候，把 `...obj` 写在后面，假如存在相同的属性名称，`obj` 就会覆盖前面的属性。

属性覆盖的原理非常的简单，后面覆盖前面请看下图。

```
> var a = { a: 1, b: 2, a : 4}  
< undefined  
> a  
< > Object {a: 4, b: 2}  
>
```

## 作用域

我们知道 `TS` 的使用者大多数都是后端语言开发者，所以通常会对 `this` 产生困惑，而 `this` 理解不理解意味着你的 `js` 有没有入门。

在将 `this` 之前，我们先看一看作用域。

通常我们所知的作用域有全局作用域，也就是我们的文件，你可以把文件看成一个被 `{}` 包裹的代码。块作用域，有 `function(){} if(){} etc` 等等包含 `{}` 的，其中每一个 `{}` 都是一个作用域。

接下来我们写下这么一段代码，它会报错，因为在外层的作用域里面找不到 `b` 变量。

```
/Users/Yugo/Desktop/nodeLover.me/func.ts [1 errors]  
(127, 10) Cannot find name 'b'.
```

```
{  
    // 我  
    let a = 1;  
    {  
        // 敌人  
        let b = a;  
    }  
  
    let c = b;  
}
```

故暗我明，对待这样的敌人，我们需要特别的谨慎。

敌人在暗处，可以看到我们所有的变量，但是我们却不知道敌人是怎么个情况。

## This

this 语义就是这，在编程中表示自己。

在 js 中，你只需要记住谁调用的， this 就指向谁，也就是 . 前面的对象，假如没有 . ，默认补一个 window. ;

对于 ()=>{} 的匿名函数来说， this 会提前绑定，看此刻是谁调用的它。而不是等调用的时候，去看谁调用的它，在去指定 this

讲道理 ()=>{} 里面是没有 this 的，它的 this 是从上一个作用域（父作用域）里面拿到的。

## 小实验

代码

```

let someObj = {
  a() {
    console.log(this);
  },
  b: () => {
    console.log(this);
  }
}

let w1 = someObj.a
let w2 = someObj.b

w1()
w2()

someObj.a()
someObj.b()

let someObj2 = {
  name: 'yugo',
  a: () => { },
  b: () => { },
}

someObj2.a = someObj.a
someObj2.a()

someObj2.b = someObj.b
someObj2.b()

```

把编译好的代码，嵌入到 `html` 文件里面去，打开控制台，这样便于观察。

结果如下

```

▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo
...}
▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo
...}
▶ Object {}
▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo
...}
▶ Object {name: "yugo"}
▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo
...}

```

- w1() 打印出了 window

这其实非常的正常 w1 拿到的是 a 函数的引用，根据我们的没有 . 加上 window. 。此时就是 window.w1() ，也就是 window 调用的 w1 所以理所应当也是打印出 window

- w2() 打印出 window

你以为原理和 w1 一样是吗？其实并不是，对于 `()=>{}` ，我们来看此时，是谁调用的 b , 相当于，抛弃本身的作用域，查找 b 的父级作用域。

此时情况变成了

```
windos.someObj = {  
    // 找 this  
}
```

someObj 的 this 是谁？此时 this 理所应当的指向了 window

- someObj.a() 打印了对象自己, 也就是 someObj ，根据上面的，这是理所当然的。
- someObj.b() 打印了 window ，因为 `()=>{}` 在声明的时候就绑定到了 this ，而这个声明时候的 this 指向的是 window

对于 `()=>{}` 的 this 指向，我们看一看编译出来的 js 就瞬间明白了。

```
var _this = this;  
  
....  
  
var someObj = {  
    a: function () {  
        console.log(this);  
    },  
    b: function () {  
        console.log(_this);  
    }  
};
```

其实 `ts` 是声明了一个临时变量去保存这个 `this`。

而没有 `.` 的默认添加 `window.`, 所以默认 `this` 就是 `window`

- `someObj2.a()`

打印 `someObj2`，这个应该咩问题吧！

- `someObj2.b()`

无论 `b` 怎么赋值，`b` 函数里面的 `this` 都被替换了 `_this` 变量，  
而 `_this` 都指向 `window`

## 函数重载

当我们一个函数可以接受多种类型的参数的时候怎么办？这个时候就要用到我们的函数重载了。

简单的来说就是，把每一种特定的函数声明写出来，最后实现一个能够处理所有参数类型的函数。这样有点拗口。

那就通俗来说，实现一个可以煮很多东西的电器，然后在使用说明书上面说明，支持煮哪些东西。

代码如下

```
// 我可以处理传入 string 类型的 food
function cookingSomeThing(food: string) : void;

// 我可以处理传入 string 数组类型的 foods
function cookingSomeThing(foods: string[]): void ;

// 我是如何处理的
function cookingSomeThing(food) : void{
    if(typeof food === 'string') {
        console.log('food is string');
    }

    if(Array.isArray(food)) {
        console.log('food is array');
    }
}

cookingSomeThing('chicken')
cookingSomeThing(['chicken', 'beef'])
```

我们需要实现一个万能的函数，可以处理所有状况。

自己写一个煮菜函数吧，对于传入 class Chicken 该怎么处理，而对于传入 class Beef 怎么处理你自己决定，还可以添加一些你想要的功能。

试一试这样的写法，联合类型

```
function cooking( food: Chichen | Beef ) : any {  
}
```

这样是不是避免了很多声明代码，更多细节以后再谈。

答案在这里

```
class Chicken{}  
class Beef{}  
  
function cooking(c : Chicken) : any ;  
function cooking(b : Beef) : any ;  
  
function cooking(food) : any{  
    if(food instanceof Chicken) {  
        console.log("第一步： 杀鸡取卵");  
        console.log("煮鸡肉呀~ 我最喜欢吃~");  
    }  
  
    if(food instanceof Beef) {  
        console.log("牛肉不能煮太久，要不然不好吃了")  
        console.log("哎呀~ 这肉被我煮黑了。");  
    }  
}  
  
let c = new Chicken()  
let b = new Beef()  
  
cooking(c)  
cooking(b)
```

这是 pro plus 版本

```
class Chicken{}  
class Beef{}  
  
function cooking(food : Chicken | Beef ) : any {  
    if(food instanceof Chicken) {  
        console.log("第一步： 杀鸡取卵");  
        console.log("煮鸡肉呀~ 我最喜欢吃~");  
    }  
  
    if(food instanceof Beef) {  
        console.log("牛肉不能煮太久，要不然不好吃了")  
        console.log("哎呀~ 这肉被我煮黑了。");  
    }  
}  
  
let c = new Chicken()  
let b = new Beef()  
  
cooking(c)  
cooking(b)
```

你还可以尝试，在两个 class 里面添加一些属性，看看在 cooking 里面的 food 的代码提示是怎么样的。

答案就是， food instanceof Chicken 里面的会提示 chicken 的属性，外面则不会，同样 food instanceof Beef 里面的会提示 beef 的属性，

## 自我总结

接下来看你的了。

讲道理，我已经都会了，还每一段代码我都会自己写出来，我一点都没有偷懒，你们都还没懂其中的门道，那就更需要练习了，连我都没偷懒，你们就更不应当偷懒了。

新建 `t.ts` 文件

## 我们为什么需要泛型？

在前端框架发展的过程中，你一定听过组件化这种思维，大公司的程序员哥哥们，为了早日回家吃夜宵（通常下班都10点了），所以他们通常会想尽一切办法减少工作量（拿刀砍需求），或者提高工作效率（效率工具、可复用的组件）。

所以可以减少重复性的代码就叫组件，一直提取提取，提取到最后，就成了框架。

有一点我们非常清楚，`ts` 是静态类型，对于 `Array` 类型的，代码编辑器会自动提示它可以访问的一些属性的方法，而 `js` 则不同，所有方法都需要你自己去记忆。

当我们需要写一个传入什么类型就得到什么类型的函数。

于是聪明的你可能给出这样的“万能”函数

```
function one(a: any) : any{
    return a;
}
```

好像你这个函数没有提示的功能，且我给你 `number`，但是你给的是 `any`，跟写原声的 `js` 没有任何区别呀~

你说等会，让我改改，于是你可能又给出这样的函数。

其实 `a as number` 有点多余，去掉这个你会发现代码提示依旧会有，上一节最后的小练习里面其实有提到过。

```
function one(a: any) : any{
    if(typeof a === 'number') {
        let ret = (a as number)
        return ret ;
    }
    return a;
}
```

```

function one(a: any) : any{
    if(typeof a === 'number') {
        let ret = (a as number).
            toExponential();
        return ret;
    }
    return a;
}

let a = one(2);

```

toExponential method  
toFixed method  
toLocaleString method  
toPrecision method  
toString method  
valueOf method

这还算不错的，尽管没有完全解决我们的问题，我们来想象一下最糟糕的情况，也就是下下策。

那就是为每一种类型都写一个方法，这样每一个方法的输入输出都是可以预测的，传入是啥类型，输出就是什么类型，而不是 `any`。

可能有的人思维又发散一点，通过重载去解决这个问题。

我只能说你们都非常棒，思维都很活跃，世界需要你们这样的人才。

## 那什么才是最优的做法呢？

或者说有没有一种方法在调用的时候再指定类型呢？

你肯定已经猜到了，那就是本节的主题泛型，代码的组件化，

其实代码也非常的简单，比起你写很多声明要好很多，不过因为我们这个方法是运行的时候才指定类型，而且 `<T>` 是可以多种类型，所以还是需要你自己去适配，通过 `typeof` 去检测的。

```

function one<T>(a: T) : T{
    return a;
}

let a1 = one<number>(1)

let a2 = one(520)

```

当你去查看他们的函数类型的时候，你会发现比较有意思的事情。

这个你可能你不会觉得奇怪。

```
let a1 = one<number>(1)
let a2 = one<number>(1)
```

那么看看这个，当时我就惊呆了。

```
let a2 = one(520)
function one<520>(a: 520): 520
```

也就是说描述  $T$  是什么类型的时候，你可以在  $<number>$  描述它是一个  $number$  类型，同样也可以这样描述描述  $(a: T)$  对应  $(520)$ ， $T$  就是  $520$  的类型。

当你想要这样去写函数的时候，编辑器霸道的拒绝了你，说了一句，这是我的方言，不要以为你会说方言就可以跟我装老乡，对不起，我不认。

```
function two<520>(a: number = 520): 520{
    return a;
}

interface funcI {
    <520>(a: 520): 520
}
```

编译器其实非常的智能，能不让你写的东西，绝对不然绝对不让你多写，尽管编译器把你的类型翻译成了他的方言了。

能更具体的，它就更具体，相比较  $number$  和  $520$  来说，哪个更具体？

## 泛型数组

有的时候，我们需要传入一个某个类型  $T$  的数组。

对于描述数组有2种写法，所以这里也有2种写法

```
function two<T>(a: T[]) : T{
    return a[0];
}

function three<T>(a: Array<T>) : T{
    return a[0];
}
```

对于函数，是不是还有匿名函数用变量保存的形式呀？

```
let two2 : <T>(a : T[]) => T = function (a) {
    return a[0];
}
```

只需要把 `<T>` 写 `()` 前面就好了，其他的跟匿名函数的描述基本一致。

同样我们用接口来描述某个泛型方法

```
interface funcI {
    <T>(a: T[]) : T
}

let two3 : funcI = two2

let two4 : { <T>(a: T[]) : T } = two2
```

等于，是一个非常有意思的名词，他的符号叫 `=`，在数学意义上，就是两者具有相同的辩证关系，而在现实世界中代表着类似事物，或者同一事物。

在学习编程的时候，老师有的时候会叫你忘掉 `=`，把它认为成赋值。

我觉得你理解为复制还差不多，从某种意义上来说它还是相等，你也可以理解为，赋值之后，他们是同一事物，所以相等。

其实实际意义的相等与编程里面的 `=` 的差异就是，编程里面的 `=` 不支持前后替换，也就是 `a = b` 不能 `b = a`。

因为

```
a = b
```

```
c = a
```

所以

```
b = c
```

而对于上面的例子

funcI 就等于

```
{  
    <T>(a: T[]) : T  
}
```

所以

```
let two3 : funcI
```

就可以替换为

```
let two3 : { <T>(a: T[]) : T }
```

## 接口泛型

同时我们还可以这样玩，在声明的时候指定接口泛型里面的类型

```
interface someI<T>{  
    (a : T) : T  
}  
  
let b : someI<number>
```

此时 b 一个就是一个 `(a: number) => number` 的匿名函数

## 类泛型

我们知道，泛型的作用就是在调用的时候，再限定某些值的类型。

```
class Person<T, U>{
    other: T
    age: U
}

let p = new Person<string, number>()
p.other = "good men"
p.age = 12
```

这样我们就可以在实例化的时候再指定 `other` 和 `age` 的具体类型。

之前没有提到如何声明多个泛型，其实非常简单，用逗号隔开就行了。

| 假如我们明确指定泛型有些什么字段怎么办？

像这样通过 `extends` 关键字进行继承即可，这里是可以继承 `interface` 和 `class`，从某种意义上来说，就实现了我们前面提到的代码提示功能。

```
interface hasLength {
    length: number;
}

function funcTest<T extends hasLength>(arg: T): T {
    arg.|_
    return length          property|
}
```

```
interface hasLength {
    length: number;
}

function funcTest<T extends hasLength>(arg: T): T {
    return arg;
}
```

泛型由于比较特殊，你可以把它理解为特殊的接口类型，所以可以继承接口，而 class 是不能继承接口的

亦或者说，通过接口来描述泛型。

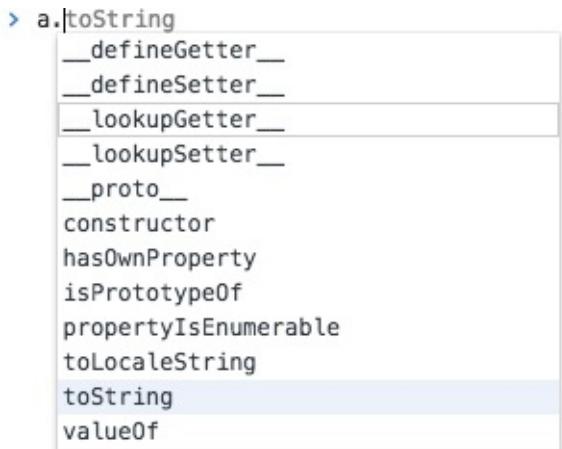
```
class clTest extends hasLength{
    length: number
}
```

## 原型

继承必须提到原型，对于后端开发者可能对原型不太了解，这里我尽量把原型说的通俗易懂，且内容也足够深入。因为原型是 js 核心内容，反正这个原型被开发者各种玩，每个老司机对于原型的理解绝对不一般。

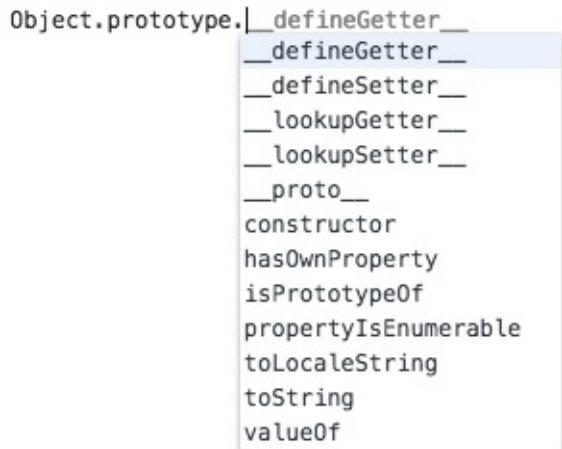
```
var a = []
undefined
a
▼ Object
  ▶ __proto__: Object
```

这里我们定义了一个 {}，其实我们看到这个 a 是一个空的对象，上面什么实际属性都没有，但是我们可以看到一个 \_\_proto\_\_ 属性，其实它就是我们说的原型。



当我们通过`.`去调用它的方法的时候，我们可以发现明明我们没有定义这些方法，但是我们却可以访问这些方法。

同时我们输入`Object.prototype.`就可以出现代码提示。



我们发现他们提示的方法都是一样的。

我不知道智慧的你有没有注意到`__proto__: Object` 其实这个`__proto__` 其实就是`Object`。

而`js`里面，当一个对象，找不到某些东西的时候，它会往它的`__proto__`对象上面去找，当然假如`__proto__`对象上面还有`__proto__`，它会一直找下去，所以就有了原型链这种说法。所以通过特性就实现继承。

这里我们深入一点。

`Object.prototype`与`a`提示的方法一样，那么它们是否存在某种不可为外人所知的秘密呢？

你可能先想到它们是否相等呢？

```
Object.prototype == a
```

```
false
```

```
a.__proto__ == Object
```

```
false
```

我想你是不是忘了什么，提示你一下 `__proto__`，找不到的属性和方法怎么办？

```
Object.prototype == a.__proto__
```

```
true
```

我们再来看看下面的

```
> var a = {}
< undefined
> a
< ▼ Object { }
  ► __proto__: Object
> var b = new Object()
< undefined
> b
< ▷ Object {}
```

通过 `{}` 与 `new Object` 出来的对象是一样的。

其实通过 `{}` 新建的对象，叫做字面量方法创建对象。

它会被编译器转换为 `new Object` 的。

那么 `new` 究竟发生了什么事情呢？

之前我们是不是说过 `new` 会调用 `constructor`

```
> Object.prototype.constructor()
```

```
< ▷ Object {}
```

你是不是发现了什么！

其实还有一件事编译器做了的，那就是新创建的这个对象的 `__proto__` 等于 `Object.prototype`。

我们梳理一下思路，`new A()`，首先通过 `prototype.constructor` 构造一个新的对象，然后把 `A` 对象的 `prototype` 赋值给新对象的 `__proto__`。

同样也就是说 `prototype` 上面的属性和方法就是实例方法和变量。

## 深入的理解继承

先写我们的 ts 代码

```
class a {}

class b extends a{}
```

编译出来之后，这里我做了一些修改。

把以下代码写到一个 `html` 文档里面，我们通过 `chrome` 调试运行。

```
var __extends = (this && this.__extends) || function (d, b) {
    console.log(d)
    console.log(b)
    for (var p in b) {
        if (b.hasOwnProperty(p)) {
            console.log(p)
            d[p] = b[p]
        }
    }
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (___.prototype
= b.prototype, new __());
};

var a = (function () {
    function a() {
    }
    a.sname = '1';
    return a;
}());
var b = (function (_super) {
    __extends(b, _super);
    function b() {
        return _super.apply(this, arguments) || this;
    }
    return b;
}(a));
```

这里我们一个一个的来说，慢慢来。

## 了解 `Object.create`

首先我们创建一个变量 `a`，给它一个 `name` 属性，当做一个标识

```
> var a = {}
< undefined
> a
< Object {__proto__: Object}
> a.name = 2
< 2
> a
< Object {name: 2}
    > __proto__: Object
```

此时我们通过 `create` 来创建 `b`

```
> var b = Object.create(a)
< undefined
> b
< Object {__proto__: Object}
    > name: 2
        > __proto__: Object
```

我们发现 `__proto__` 指向了我们的 `a`

相当于，我们新建了一个对象，并且把 `__proto__` 指向了我们的 `a`。

```
> var c = {}
< undefined
> c.__proto__ = a
< Object {name: 2}
> c
< Object {__proto__: Object}
    > name: 2
        > __proto__: Object
```

当我们传入 `null` 的时候。

```
> a = Object.create(null)
< Object {No Properties}
```

它便是一个真正意义上的空对象。

当然我们也可以把我们的 `__proto__` 赋值为 `null`，从下面的例子我们可以看出使用 `delete` 是没有任何作用的。

```
> var b = {}
< undefined
> delete b.__proto__
< true
> b
< ▾ Object [1]
  ▶ __proto__: Object
> b.__proto__ = null
< null
> b
< ▾ Object [1]
  No Properties
```

首先我们看 `var __extends = (this && this.__extends) || function (d, b) {}`

```
(this && this.__extends)
```

```
> this
< ▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo ...}
```

在控制台里面输入 `this`，之前我们也有说过 `this` 默认是 `window`，在浏览器环境中。

接下来我们看看 `&&` 运算符

```
> 1 && 2
< 2
> 0 && 2
< 0
> 1 && 0
< 0
> 1 && 0 && 2
< 0
```

我们把 `&&` 想象成电线，`0` 或者其他否定的值，比如 `undefined`、`null` 等，代表着断开，也就是断路了，假如断开了，就返回断开的时候的否定值。

假如全线通过，一路绿灯，那就返回最后一个。

```
> var some = '123'
< undefined
> window.some
< "123"
> this.some
< "123"
>
```

而我们在全局作用域下面通过 `var` 声明的变量会自动挂载到 `window` 下面，这样容易形成全局代码命名污染。

因为这里定义一个 `a`，那里又定义一个 `a`，js 解释器就不知道你要使用的是哪个 `a`。

```
> let some2 = '123'
< undefined
> window.some2
< undefined
> |
```

而 `let` 不会，这就是为什么我推荐你使用 `let` 的原因之一。

`(this && this.__extends)` 这段先判断是否有 `this` 有再继续判断 `this` 上面是否已经有定义过了 `__extends`，假如没有此时返回一个否定值，也就是断开了，这个否定值是 `undefined`，假如定义过了，就直接返回 `this.__extends`，于是此时 `var __extends` 就拿到了全局的 `__extends` 当得到否定值的时候

```
undefined || function (d, b) {}
```

```
undefined || 'aaa'
"aaa"
undefined || null
null
'aaa' || undefined
"aaa"
```

当遇到第一个正确的时候，就立刻返回当前的值，假如都不正确，那就放回最后一个否定值。

此时 `var __extends` 就是后面这个函数了。

此时我们进入函数 `function (d, b) {}` 内部逻辑，用 `ts` 的代码来说就是 `class d extends b`。

```
for (var p in b) {
    if (b.hasOwnProperty(p)) {
        console.log(p)
        d[p] = b[p]
    }
}
```

这一段代码，会遍历 `b`，所有属性，此时的属性是静态属性，你可以看到上面完整的代码里面的 `a` 类有 `sname` 静态属性，这里又有一个 `console.log(p)` ，它会打印到我们的控制，在控制台里面我们可以看到一个 `sname` 。

`b.hasOwnProperty` 会判断 `p` 是不是 `b` 自己的属性，它只会拷贝自己的属性，而不会拷贝原型链上面的。

```
function __() { this.constructor = d; }
```

这一段代码就是创建一个 `__` 函数，在原生 js，我们是通过函数来模拟类，而这个方法里面的 `this` 就等于 `___.prototype`，挂载在上面的变量就是实例变量。

而当我们去 `new __()` 的时候，会去调用 `___.prototype.constructor`，也就是 `this.constructor` 同样也就是 `d`。

```
d.prototype = b === null ? Object.create(b) : (__.prototype = b.
prototype, new __());
```

这个首先判断 `b === null?`，假如是空的话，`Object.create(null)` 就直接是一个空对象，也就是 `d.prototype` 指向一个空对象。

而假如 `b` 不为空的话，那么他就是一个函数。

```
___.prototype = b.prototype, new __()
```

对于 `,`，你可以这么理解，首先 `a` 被复制为 1。之后 `,` 会返回最后一个表达式的值。也就是 `b = a`

```
> var b = a = 1 ,a
< undefined
> b
< 1
```

把 `b.prototype` 复制给 `___.prototype`，之后再 `new __()`

`new __()` 会去调用 `d` 函数，也就是 `class d extends b` 中的 `d` 自己的逻辑。之后再返回一个对象 `{__proto__: b.prototype}`。这样 `d.prototype` 就等于返回的这个对象。

当我们 `new d()` 的时候新对象的 `__proto__` 指向 `d.prototype`，这样就形成了原型链，当找不到方法的时候，会去找 `__proto__` 的 `__proto__`，而这个 `__proto__` 就是我们 `class d extends b` 的 `b.prototype`，这样我们就可以访问到了 `b` 的实例方法。

```
function b() {
    return _super.apply(this, arguments) || this;
}
```

而这个 `_super` 就是 `a`，这里就是调用 `a` 的函数，然后绑定一下 `this`，假如没有 `_super.apply` 就直接放回 `this`。

说实话，我自己都晕乎，我非常担心你们看不懂，已经讲的最细了，js 的原型指来指去的，连老司机都要翻，这就是为什么会出现 `ts` 与 `es` 的 `class` 关键字。

这不是写程序，这是在走迷宫。

我们来梳理一下思路，保持有条理。

```
class d extends b
```

- 首先拷贝静态属性
- 构造一个新的`()`函数，当`new()`的时候，会先执行被继承函数 `b`，这个函数通常会初始化一些变量比如 `this.x = 'some'`之类的。
- 把 `__` 函数的原型指向 `b` 的原型，这样 `new` 出来的对象就可以访问 `b` 的实例方法

这样是不是就条理清晰了。思路很简单，但是代码解释起来就不敢恭维了。

关于 `prototype` 与 `__proto__` 的区别，请向上找到 `new` 调用的过程，`new` 其实是一种特殊的函数调用方式。

## 泛型继承类

```

class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function findKeeper<A extends Animal, K> (a: {new(): A;
prototype: {keeper: K}}): K {

    return a.prototype.keeper;
}

findKeeper(Lion).nametag;

```

这里最核心的就是 `a: {new(): A;prototype: {keeper: K}}`

我们慢慢来，分解开读，`new(): A` 表示可以 `new` ，也就是实例化，并且它的返回值是 `A` 泛型

`prototype: {keeper: K}` 描述原型，刚刚我们也提到了，这里指原型上面有一个 `keeper` 属性并且类型是 `K` 泛型。这里的原型上面的属性，其实就是实例属性。

```
function findKeeper<A extends Animal, K> (a: {new(): A;  
prototype: {keeper: K}}): K {  
  
    return a.prototype.keeper;  
}
```

整体的来读一下这个函数。

`findKeeper` 函数规定有2个泛型，泛型 `A` 集成 `Animal` 拥有 `Animal` 所有属性和方法，还有一个泛型 `K`，传入一个参数，规定这个参数是可以 `new` 关键字调用的，也就是说传入的应该是 `Class` 而不是实例，并且它必须有一个实例属性 `keeper`，此时把 `keeper` 的类型标记为泛型 `K`，并且把泛型 `K` 的类型作为返回值的类型，函数内容就是返回原型链上的 `keeper` 属性。

其实上面这段代码我们是没法运行，尽管没报错，因为 `keeper` 是实例属性，都没初始化，哪来的值。看看你能不能想什么办法让上面代码运行。

我这里给一份答案，把 `keeper` 改成静态的，而且讲道理不应该是每一个狮子都住一个狮子园，应该是所有共同的 `keeper`，所以说是所有实例狮子共享的住的地府，所以它更应该是静态的。

```
class BeeKeeper {  
    hasMask: boolean;  
}  
  
class ZooKeeper {  
    constructor(public nametag: string){  
    }  
}  
  
class Animal {  
    numLegs: number;  
}  
  
class Bee extends Animal {  
    static keeper: BeeKeeper = new BeeKeeper();  
}  
  
class Lion extends Animal {  
    static keeper: ZooKeeper = new ZooKeeper('zookeeper');  
}  
  
function findKeeper<A extends Animal, K> (a: {new(): A;  
    keeper: K }): K {  
    return a.keeper;  
}  
  
let a2 = findKeeper(Lion).nametag  
  
console.log(a2);
```

你可能对

```
class ZooKeeper {  
    constructor(public nametag: string){  
    }  
}
```

有困惑，其实它等于

```
class ZooKeeper {  
    public nametag  
    constructor(nametag: string){  
        this.nametag = nametag  
    }  
}
```

对于 `{new(): A; keeper: K}` 我们可以把它理解为 `css` 的内联样式，我们修改一下，改成接口泛型，接口名称 `aInterface` 就相当于我们 `css` 的 `class` 名称

```
interface aInterface<A, K>{  
    new(): A;  
    keeper: K;  
}  
  
function findKeeper<A extends Animal, K> (a: aInterface<A, K>): K  
{  
    return a.keeper;  
}
```

这样看起来精简多了，但是初学者看起来会马良的，因为这里面包含的信息量太大，包含了如何毁灭地球的计划。

## 总结

泛型可以看做特殊类型的接口，所以泛型可以继承接口，而类不行。

泛型拥有与接口相似的功劳，那就是描述类型，不过泛型可以延迟描述，等需要调用的时候再描述。

你自己还有什么要总结的呢？都自己添到自己的笔记本上吧。

枚举是为了让程序可读性更好，比如用来描述用户的角色，普通的会员、付费的会员等，同时也限定了用户角色的种类，保证安全性，不会出现上帝角色这种乱入的东西。

## 枚举的类别与写法

默认值从0开始，依次递增，这个你应该还记得。

### 普通的枚举

```
let str = 'something'

enum test{
    test01,
}

enum FileAccess {
    None,
    Read     = 1 << 1,
    Write    = 1 << 2,
    ReadWrite = Read | Write,
    Test = test.test01,
    0 = str.length
}
```

对于前面这几种，是常理属性，因为它会直接计算出来，只要是使用了一元运算符号，或者使用了其他枚举的内容，都会计算出来。

```
None,
Read     = 1 << 1,
Write    = 1 << 2,
ReadWrite = Read | Write,
Test = test.test01,
```

而

```
0 = str.length
```

则不会，编译出来的 JS 代码会是这个样子。

```
var str = 'something';
var test;
(function (test) {
    test[test["test01"] = 0] = "test01";
})(test || (test = {}));
var FileAccess;
(function (FileAccess) {
    FileAccess[FileAccess["None"] = 0] = "None";
    FileAccess[FileAccess["Read"] = 2] = "Read";
    FileAccess[FileAccess["Write"] = 4] = "Write";
    FileAccess[FileAccess["ReadWrite"] = 6] = "ReadWrite";
    FileAccess[FileAccess["Test01"] = 0] = "Test01";
    FileAccess[FileAccess["0"] = str.length] = "0";
})(FileAccess || (FileAccess = {}));
```

它这个写法你可能看起来比较复杂。

你分开来看就行了。`FileAccess` 其实是一个对象。

```
FileAccess["None"] = 0
```

此时 `FileAccess.None` 就为 0 了

```
> var a = {}
< undefined
> a.name = 2
< 2
```

我们可以看到给属性赋值的时候，返回的是这个值。

接着就是

```
FileAccess[0] = "None"
```

把编译出来的代码加一个 `console.log` 然后放到 `html` 中执行。

所以最后这个对象可能会是这样。

```
▼ Object
  0: "Test01"
  2: "Read"
  4: "Write"
  6: "ReadWrite"
  9: "0"
  20: "Test02"
  None: 0
  0: 9
  Read: 2
  ReadWrite: 6
  Test01: 0
  Test02: 20
  Write: 4
▶ __proto__: Object
```

## 常量枚举

这个枚举只会存在 `ts` 文件中，编译成 `js` 不会生成任何代码，全部都用它的值代替。

既然是常量枚举，那就在他前面加个 `const` 即可

```
const enum Enum {
  A = 1,
  B = A * 2
}

var arr = [Enum.A, Enum.B]
```

它编译出来之后就这么一句。

```
var arr = [1 /* A */, 2 /* B */];
```

直接把它的值给编译了出来，只不过加了一个注释而已。

## 自动推导

往兼容性最好，描述最准确（且不会报错）的地方推

```
let a2 = 1;
      Let a2: number .L
```

当我们给 `a2` 赋值为 1 的时候，编译器自动推导出为 `number`

```
let arr2 = [1, 23, null]  
let n | let arr2: number[]
```

这个被推导成了数字数组，因为 `number` 可以接受为 `null`

```
let num: number = null
```

```
let arr3 = [1, 23, {}]  
let o | let arr3: {}[]
```

这个被推导成了对象数组，因为把 `2` 付给对象类型是不会报错的，从某种意义上说 `2` 也是 `Number` 对象的实例

```
let obj: {} = 2
```

而当我们给 `{}`，添加属性的时候，

```
let arr3 = [1, 23, {a: 2}]  
let ob | let arr3: (number | {  
    a: number;  
})[]
```

它会添加一个 `|` 表示或者的关系。

```
let a2 = 1;  
  
let arr2 = [1, 23, null]  
  
let num: number = null  
  
let arr3 = [1, 23, {a: 2}]  
  
let obj: {} = 2
```

而对于类

```
class o {
    name: string = 'default';
}

class a extends o{
    // a: string;
}
class b extends o{
    // b: string;
}
class c extends o{
    // c: string;
}

var arr4 = [new a(), new b(), new c()];
    var arr4: a[]
// let ss = arr4[0] as a
```

你会发现 `arr4` 被推导成了 `a[]` 类型，为什么呢？

因为 `ts` 判断一个类是否相等是会去看它的类型，而这里的三个类的类型都是 `{}` 空对象。所以编译器认为3个都是相等的，所以就用了第一个的名字，也就是 `a`。

```

class a extends o{
    a: string;
}
class b extends o{
    // b: string;
}
class c extends o{
    // c: string;
}

var arr4 = [new a(), new b(), new c()];
    var arr4: b[]

```

当我们打开 `a` 的属性的时候，我们发现 `arr4` 变成了 `b[]` 类型。

为什么呢？

`a` 的类型是什么呢？我们用内联的接口描述一下，`{a:string}` 而 `b` 的类型是 `{}`，假如我们 `arr4` 的类型是 `a` 的类型的数组的话，`b` 就不符合规范。

而 `arr4` 的类型是 `b` 的类型数组，那就全部合适，不会报错。

其实此时我们是已经丢失了类型的。

```

let ss = arr4[0]

ss.
    name      property

```

此时的 `a`，根本就拿不到任何它自己的属性。

```

var arr4: o[] = [new a(), new b(), new c()];

let ss = arr4[0]

```

跟使用它们的父类去接受值是一样的，丢失了自己的属性。

当然我们可以这样处理，强制转换。

```
var arr4: o[] = [new a(), new b(), new c()];
if(arr4[0] instanceof a) {
    let ss = arr4[0] as a
    ss.
```

a	property
name	property

我们把其他的注释也解开

```
class o {
    name: string = 'default';
}

class a extends o{
    a: string;
}

class b extends o{
    b: string;
}

class c extends o{
    c: string;
}

var arr4 = [new a(), new b(), new c()];
let a1 var arr4: (a | b | c)[]
```

此时的 `arr4` 是 `(a|b|c)[]`，表示一个可以放 `a / b / c` 实例化的数组。

```
let a1 = arr4[0]
let a1: a | b | c
```

所以我们取到的类型是

依旧还是需要强制转换，那就没有什么其他的办法保证数组里面每一个类的特征吗？

有，利用元组，不过这样就不能把它看做数组，往里面添加新的东西了。

```
var arr4: [a,b,c] = [new a(), new b(), new c()];
let a1 = arr4[0]
a1.
```

a	property
name	property

函数参数的自动推导

```
window.onkeyup = (ev) => {
    ev.
```

AT_TARGET	property
BUBBLING_PHASE	property
CAPTURING_PHASE	property

我们可以看到此时的 `ev` 有代码提示，说明已经自动推导出了类型。

选择 `onkeyup` 按 `f12`，就可以看到以下声明，这是系统自带的类型。

```
onkeyup: (this: Window, ev: KeyboardEvent) => any;
```

第一个参数是指定 `this`。

```
class some {
    name: 'some';
    clickHandle: (this: some, s: string) => any = (s) => {
        this.
    }
}
```

clickHandle	property
name	property

这个 `this` 就像这样，可以提供代码提示。

这种绑定 `this` 的做法只能在 `Class` 和 `Interface` 里面使用。

而 `ev` 就是我们函数的 `ev` 也就是 `KeyboardEvent` 类型，所以才有代码提示。

其实也就是从左边类型，推导出右边的类型。

## 类型之间的兼容性

之前我们提到过 `ts` 自动推导类型的时候，会推导出限制最低的类型。

而判断类型之间是否兼容，会通过类型的比较得出。

```
interface Named {
    name: string;
}

class Person{
    name: string;
}

let p: Named;
p = new Person();
```

这一段代码是正确没有报错的，我们用内联形式描述一下 `Person {name:string}`，恰好与接口 `Named` 所描述的是一致的。

也就是说，其实 `Person` 隐含的意义实现了 `Named` 接口。

当我们给 `Person` 添加一个属性的时候依旧没报错。

```
class Person{
    name: string;
    age: number;
}
```

但是此时的属性 `age` 丢失了，因为 `Named` 上面并没有 `age`，



尽管我们知道，它一定有一个 `age` 属性。

而此时为什么可以赋值，其实跟我们之前的数组保证类的特征类似。

此时 `Person` 的约束为 `{name:string;age:number}` 而 `Named` 的约束为 `{name:string}`。

你可以看成解构 `{name} = {name, age}` 所以此时的 `age` 就丢失了，尽管在 js 层面是一定存在 `age` 这个属性的，但是在 ts 层面来说，因为你抛弃了 `age`，所以它是不记得有 `age` 这个东西的。

当然假如反过来 `{name, age} = {name}` 就会报错，因为此时 `age` 拿不到任何值。

也就是说，约束少的可以得到约束多的一部分，也就是俩人都有的属性。

就像类型装换，`number` 可以转成 `string`，而并不是所有的 `string` 都可以转成 `number`，只有存在一点不安全的东西，ts 就不允许你这么做。

接下来我们看看函数之间的兼容。

```
let x = (a: number) => 1;

let y = (b: number, s: string) => 2;

y = x

console.log(y());
    y(b: number, s: string): number
      b:
        1/1
```

我们可以看到把 `x` 赋值给 `y` 并没有报错，而当我们再次调用 `y` 的时候，依旧需要我们传递 2 个参数。但是在 js 层面来说，假如我们打印一下 `y` 的话。

```
function (a) { return 1; }
```

这里为什么可以把一个少参数的函数赋给一个多的参数函数呢？你可能会这样问。

那么我问你，请问函数的参数越多是否意味着约束越强？

答案肯定是的，相比较 `x` 来说，`y` 的约束性更强，当把 `x` 复制给 `y` 的时候，首先会去判断 `x` 的参数类型与 `y` 的参数类型匹配不匹配。

这里的匹配是有顺序的。

```
let x = (a: number, c: string) => 1;

let y = (b: number, s: string) => 2;

y = x
```

当我们增加一个参数，依旧正常。

当我们修改一下位置，立马就报错了。

```
let x = (a: string, c: number) => 1;

let y = (b: number, s: string) => 2;

y = x

Type '(a: string, c: number) => number' is not assignable to type '(b: number, s: string) => number'.
  Types of parameters 'a' and 'b' are incompatible.
    Type 'number' is not assignable to type 'string'.
let y: (b: number, s: string) => number
```

约束强的可以兼容约束弱的。

通过接口的兼容性来实现来实现 js 的一些特殊的回调方法。

```
enum mEventType { Mouse, Keyboard }

interface mEvent { timestamp: number; }
interface mM MouseEvent extends mEvent { x: number; y: number }
interface mKeyEvent extends mEvent { keyCode: number }

function listenEvent(eventType: mEventType, handler: (n: mEvent)
=> void) {}

listenEvent(mEventType.Mouse, (e: mM MouseEvent) => console.log(e.
x + ',' + e.y));

listenEvent(mEventType.Mouse, (e: mEvent) => console.log((<mMouse
eEvent>e).x + ',' + (<mMouseEvent>e).y));
listenEvent(mEventType.Keyboard, <(e: mEvent) => void>((e: mKeyE
vent) => console.log(e.keyCode)));
```

对于 `mMouseEvent` 和 `mKeyEvent` 来说，`mEvent` 相当于父类。

而 `listenEvent(eventType: mEventType, handler: (n: mEvent) => void)`  
`{}` 要求我们第二个参数传入的回调里面的一个参数是 `mEvent` 类型 因为  
`mMouseEvent` 和 `mKeyEvent` 都继承了 `mEvent`，所有传入 `mMouseEvent` 和  
`mKeyEvent` 都是可以的。

当然上面的代码也用到了多次强制转换，一个是强制转换回调的类型，一个是强制  
 转换参数的类型。

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = new Animal('123', 1);

s = new Size(2);

a = s;
```

而对于类来说，只会比较他们之间的实例变量。

当泛型并没有实际约束任何属性的时候，他们是兼容的。

```
interface Empty<T> {}

let x: Empty<number>;
let y: Empty<string>;

x = y;
```

而

```
interface Empty<T> {
    name: T;
}
```

这样就不行了，因为 ts 会去检测属性是否匹配。

## 类型之间的逻辑

在之前，我们描述一类具有多个特征，可能要通过 `implement` 来实现多个接口，而现在我们可以 `&` 与 `|` 来实现泛型、类型之间的逻辑。

比如 `Person & Serializable & Loggable` 意味着同时是 `Person` 和 `Serializable` 和 `Loggable`

```
interface a{
    name: string;
}

interface b{
    age: number;
}

function person(a: a & b) {
    a.|
```

age	property
name	property

此时我们的 `a` 变量就具有了 `a` 接口和 `b` 接口的所以特征。

当然我们也可以等调用的时候再传

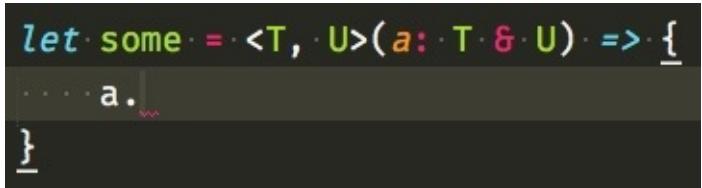
```
interface a {
    name: string;
}

interface b {
    age: number;
}

let some = <T, U>(a: T & U) => {

}

some<a, b>({ name: '123', age: 28 })
```



```
let some = <T, U>(a: T & U) => {
    a.
}
```

不过这样就不会有任何的代码提示，除非你强制转换。

假如我们的函数想要传递数字或者是字符串，你可能会用 `any` 不过这样对其他程序员不友好，看函数的类型，并不能理解得到需要具体传啥。

```
function some2(a : string | number) : any {
    if(typeof a === 'number') {
        a.toExponential()
    }
}
```

所以我们可以 `|` 来实现或者逻辑。

而对于接口的逻辑 `|` 来说。

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    return { swim(){},layEggs(){} };
}

let pet = getSmallPet();

if((pet as Fish).swim) {
    (pet as Fish).swim()
}
```

```
let pet = getSmallPet();
if((pet as Fish).swim) {
    (pet as Fish).swim()
}
```

这时候我们类型还是 `Fish | Bird`。

```
pet.
    layEggs
```

当我们去调用方法的时候，只能访问到他们共同的方法。

而想要判断是否某一类型，还是需要强转之后判断属性是否存在。

当然我们还可以写一个方法。

```

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

if(checkFish(pet)) {
    pet.
}

```

layEggs                    method  
swim                      method

这里的代码提示是通过 `pet is Fish` 实现的，当然我们可以改成 `boolean`，尽管不会报错，但是这样我们就不会有代码提示了。

## 类型别名

你可以认为这种 `{new(name:string):Person;hello():void}` 为内联类型描述，而 `interface`，认为 `class` 而类型别名，就好像把多个 `class` 或者内联的组合在一起。

当然类型别名是不再支持继承的，当你没法用接口描述类型的时候，你就应该尝试这种方式了。

```

interface a {
    name: string;
}

interface b {
    age: number;
}

type aAndB = a & b & {sayHello(name: string)};

function some3(a : aAndB) {
    a.age;
    a.name;
    a.sayHello('bob');
}

```

当然，你也可以给出具体的值，这样就有点类似枚举，也就是说，传入的值，必须是其中的某一项。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

而对于类型来说，还有一种特殊的类型 `this`，这是为方便实现链式调用。

```
class BasicCalculator {
    public constructor(protected value: number = 0) { }
    public currentValue(): number {
        return this.value;
    }
    public add(operand: number): this {
        this.value += operand;
        return this;
    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
    .multiply(5)
    .add(1)
    .currentValue();
```

就像这个 `add` 方法。

```
public add(operand: number): this {
    this.value += operand;
    return this;
}
```

实现链式调用的密码就是返回 `this`



Symbol 就像一个箱子，哪怕里面装的东西是一样的，也是没法相等的，这是因为每一个箱子都是唯一的。

Symbol 产生主要跟 `for of` 循环有关，而 `for of` 与 `for in` 的区别就是，`in` 遍历的是对象的 `key`，而 `of` 则是遍历 `value`。

只要实现了 `Symbol.iterator` 这个接口，就可以通过 `for of` 遍历。

最简单的例子就是

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

## 模块导入与导出

我们知道，基本上任何语言的都有相应管理代码导包的机制，不如 java 的 `package`，php 的 `include` 和 `require` 等。

而我们的 js 当可以运行在后端的时候，所以必须要有一种模块加载机制，于是就有了 `cjs` 规范。

而对于前端浏览器环境来说，我们引入 `script` 是通过标签进行引入的，而我们加载好的库，通常会在全局变量上面挂载某一个变量，比如我们熟知的 `$`，然而就是应该这样的方式容易导致模块的命名冲突，假如 `a.js` 和 `b.js` 都想往 `window` 上面挂载一个 `$` 对象，这样我们就没法确定这个 `$` 从哪来的。

所以说我们不得不重视模块的重要性。

在 `ts` 中，我们想要让别人使用的方法就 `export` 导出。想要使用别人的方法就用 `import` 导入。

创建我们的 `a.ts` 文件。

```

export function whatsYourName(name: string) {
    console.log(name);
}

export interface StringValidator {
    isAcceptable(s: string): boolean;
}

export const numberRegexp = /^[0-9]+$/;

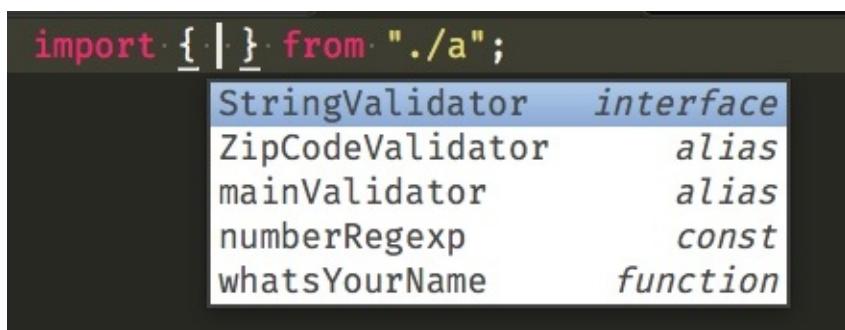
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };

```

其实 ts 跟 es6 的导出基本是一致的。

假如我们像上面这样导出，我们新建一个 `b.ts` 来导入 `a` 导出的东西。



这里就提示了我们所有导出的方法。我们可以看到我用了一个 `{}`，其实这就是解构。

我们可以把导出了看做一个对象 `{}`，`export function whatsYourName`，其实就是在导出的对象上面挂载一个 `whatsYourName` 引用。

当然我们的 `export` 可以直接写在声明的前面，或者使用先声明后导出的模式。

`export { ZipCodeValidator };` 这种先声明后导出，都需要用 `{}` 包裹起来。

`ZipCodeValidator as mainValidator` 这里的 `as` 并不是类型转换，而且给导出的取一个别名。

```
export default {  
    name: 'a'  
}
```

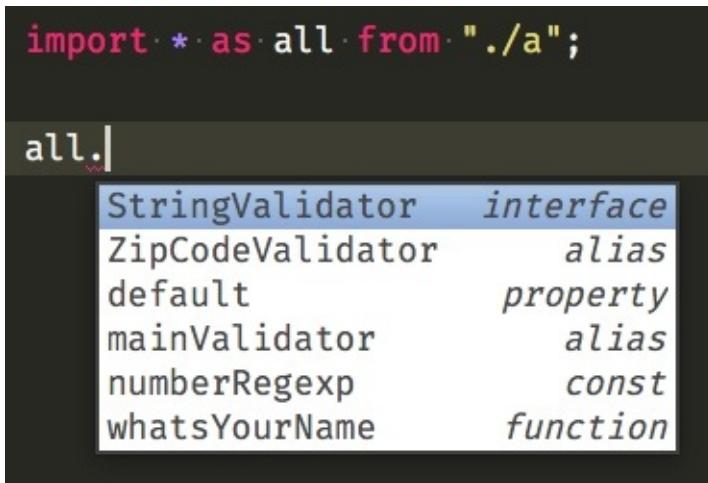
当我们使用这样的 `default` 关键字的好处就是，我并不需要具体知道你类里面有哪些方法，我直接拿，就是拿到的一个默认导出，也就是 `default` 后面的东西，导入的时候不用加 `{}`。

`export` 可以多次导出，不过取的时候要用 `{}` 和它具体的名称，因为只有对应了才能解构。

而 `export default` 则不需要 `{}`，而且你可以给它取任何名称，并且一个模块只能有一个默认导出。

```
import a from "./a";
```

比如这里拿到的 `a` 就是 `{name:'a'}`



当然我们也可以使用 `* as` 这样的语法，把 `a` 模块里面所有 `export` 导出的都挂载到 `all` 这个变量上面去。

此时我们再修改一下我们 `b.ts`

```
export * from './a';

export const name = 'b.ts';
```

`export * from './a';` 就是把 `a` 文件里面所有 `export` 的，再导出一次。

再次新建一个 `c.ts` 文件

同样可以得到提示。

```
import { } from "./b";
StringValidator      interface
ZipCodeValidator    alias
mainValidator        alias
name                const
numberRegexp         const
whatsYourName       function
```

这样是不包含默认导出的。

```
import b from "./b";
|           Module './Users/Yugo/Desktop/nodelover.me/b' has no default export.
import b
```

假如想导出 `a` 的默认导出。

```
import a from './a';

export { a };
```

我们只能这样导出，没人任何其他的简写形式。

我们通过命令

```
tsc --module commonjs a.ts
```

编译得到 `commonjs` 规范的 `js` 文件

```

exports.mainValidator = ZipCodeValidator;
exports.__esModule = true;
exports["default"] = {
    name: 'a'
};

```

多有的 `export` 导出的变成了 `exports` 上面的属性。而默认导出的挂载到了 `default` 属性上面。

```
tsc --module amd a.ts
```

而 `amd`

```

define(["require", "exports"], function (require, exports) {
    "use strict";
    function whatsYourName(name) {
        console.log(name);
    }
    exports.whatsYourName = whatsYourName;
    exports.numberRegexp = /^[0-9]+$/;
    var ZipCodeValidator = (function () {
        function ZipCodeValidator() {
        }
        ZipCodeValidator.prototype.isAcceptable = function (s) {
            return s.length === 5 && exports.numberRegexp.test(s);
        };
        return ZipCodeValidator;
    })();
    exports.ZipCodeValidator = ZipCodeValidator;
    exports.mainValidator = ZipCodeValidator;
    exports.__esModule = true;
    exports["default"] = {
        name: 'a'
    };
});

```

就是在外面加了一个大的 `define` 函数而已，这个函数来自 `require.js`，前端的一个异步加载 js 解决方案。

假如你通过 `node` 运行会报错，因为此时没有 `define` 函数以及 `umd`，这是一种兼容 `amd` 与 `commonjs` 的做法。通过 `node` 依然可以运行。

不信你可以自己打印点东西看下。

```
tsc --module amd a.ts
```

```

(function (dependencies, factory) {
    if (typeof module === 'object' && typeof module.exports ===
'object') {
        var v = factory(require, exports); if (v !== undefined)
module.exports = v;
    }
    else if (typeof define === 'function' && define.amd) {
        define(dependencies, factory);
    }
})(["require", "exports"], function (require, exports) {
    "use strict";
    function whatsYourName(name) {
        console.log(name);
    }
    exports.whatsYourName = whatsYourName;
    exports.numberRegexp = /^[0-9]+$/;
    var ZipCodeValidator = (function () {
        function ZipCodeValidator() {
        }
        ZipCodeValidator.prototype.isAcceptable = function (s) {
            return s.length === 5 && exports.numberRegexp.test(s
);
        };
        return ZipCodeValidator;
    })();
    exports.ZipCodeValidator = ZipCodeValidator;
    exports.mainValidator = ZipCodeValidator;
    exports.__esModule = true;
    exports["default"] = {
        name: 'a'
    };
});

```

还有 System

```
tsc --module system a.ts
```

```

System.register([], function (exports_1, context_1) {
    "use strict";
    var __moduleName = context_1 && context_1.id;
    function whatsYourName(name) {
        console.log(name);
    }
    exports_1("whatsYourName", whatsYourName);
    var numberRegexp, ZipCodeValidator;
    return {
        setters: [],
        execute: function () {
            exports_1("numberRegexp", numberRegexp = /^[0-9]+$/)
        ;
            ZipCodeValidator = (function () {
                function ZipCodeValidator() {
                }
                ZipCodeValidator.prototype.isAcceptable = function (s) {
                    return s.length === 5 && numberRegexp.test(s);
                };
                return ZipCodeValidator;
            })();
            exports_1("ZipCodeValidator", ZipCodeValidator);
            exports_1("mainValidator", ZipCodeValidator);
            exports_1("default", {
                name: 'a'
            });
        }
    };
});

```

而这个 `System` 的导出是通过 `exports_1("ZipCodeValidator", ZipCodeValidator);` 导出的。

第一个参数是导出名称，第二个参数是导出的引用。

现在，你应该都清楚，每一种模式转换之后的 js 都是如何导出的了。

其实这些导出，都是挂载到某一个变量下面，集中管理，或许是对象，或许是数组，等需要用到的时候，再去根据名字拿就是了。

## 外部模块与内部模块

外部模块，顾名思义，外，表示不属于内部的，对于 ts 语言来说，内部就是 ts 文件，此时的外代表着 js 文件。

我们知道引用 JS 文件，需要为它写 d.ts 文件，此时拥有 d.ts 的文件，我们可以把它看做 js + d.ts = .ts。

而引用 js 或者 ts 文件需要 import，我们把所有需要 import 的都叫做引用外部模块。因为模块是基于文件的导入导出的，需要导入的就是来自外部的。

而内部模块就代表着 ts 内部的，同时它有一个别名叫做命名空间。命名空间的作用就是把一份代码分割到多个文件。

## 模块的寻找

其实跟 node 寻找模块一样，这里简单的说一下。

它会根据你写的相对路径去找文件。ts 因为需要代码提示，所以它通常会找 d.ts 和 .ts 文件。

假如你写的是绝对路径，比如 import \$ from 'jquery' 这种，我们知道 jquery 并没有 .ts 版本的，但是有人提供了 jquery 的 d.ts 文件。

在 ts2.0 版本之后，我们直接通过 npm install @types/xxx 安装 d.ts 文件。

当我们通过 npm install @types/jquery 安装之后，会在我们的 node\_modules 里面有个 @types 文件夹，里面存放着我们的 d.ts 文件。

当 ts 找不到的时候会来这个目录找，再找不到就报错了，当然你可以定一些它需要寻找的特定目录，比如说你创建一个专门存放 d.ts 的文件夹，然后在 tsconfig.json 配置一下。

tsconfig.json 就是我们编译 ts 文件的编译选项。

## 小实验

首先创建一个文件夹 `ts-modules`

通过 `tsc --init` 生成 `tsconfig.json`

通过 `npm init -y` 生成 `package.json`

再安装 `jquery`

```
mkdir ts-modules

cd ts-modules

tsc --init

npm init -y

npm install jquery @types/jquery -S
```

创建 `src` 目录，进入再创建我们的 `main.ts`

```
mkdir src

cd src

touch main.ts
```

你一定要清楚 `@types/jquery` 是 `d.ts` 而 `jquery` 是 `js`，只有这俩样合起来才能被正常使用。

还有一点你必须要明白，`tsc` 并不会打包代码。

在你的 `main.ts` 里面输入

```
import * as $ from "jquery";

$(document).html("Hello World!");
```

来到 `tsconfig.json`，添加一行。

```
"outDir": "./dist"
```

在你的终端里面

```
tsc
```

我们可以看到 `dist/main.js`

```
"use strict";
var $ = require("jquery");
$(document).html("Hello World!");
```

此时的 `jquery` 并没有打包到该文件里面。

此时新建我们的 `modules.ts` 和 `namespace.ts`

```
// modules.ts
export const modules_a = 123;
```

```
// namespace.ts
namespace OwnSpace{
    export let var_a = 'own_space';
}
```

```
main.ts          modules.ts          namespace.ts
1 import * as $ from "jquery";
2
3 $(document).html("Hello World!");
4
5 OwnSpace.
       var_a           let
```

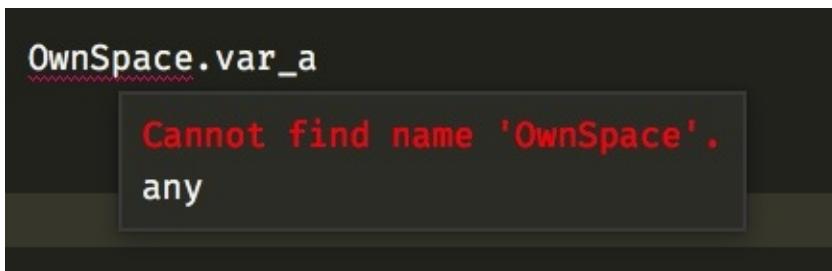
当我们在 `main.ts` 里面输入 `OwnSpace` 的时候，我们发现出现了代码提示。

而 `modules.ts` 里面导出的 `modules_a` 则不会，这是模块与命名空间的一个小区别。

当我们给 `namespace.ts` 添加一个导出的时候，立刻就报错了。

```
namespace OwnSpace{
    export let var_a = 'own_space';
}

export let out_a = 20
```



所有我们知道，包含 `namespace` 的外层不应该有 `export`，要不然就变成模块了。

当然 `namespace` 里面的变量只有导出才能被访问。

`tsc` 编译一下

来看看我们编译出来的命名空间

```
var OwnSpace;
(function (OwnSpace) {
    OwnSpace.var_a = 'own_space';
})(OwnSpace || (OwnSpace = {}));
```

其实他就是自执行函数，然后给他挂载一些变量而已。

```
> var ob
< undefined
> window.ob === ob
< true
> ob = { name : 'test'}
< ▶ Object {name: "test"}
> var ob
< undefined
> ob
< ▶ Object {name: "test"}
```

其实就跟这里演示的一样，通过 `var` 其实就是挂载到了 `window` 变量上面。

而多次通过 `var` 声明并不会报错。

修改一下 `namespace.ts`

```
namespace OwnSpace{
    export let var_a = 'own_space_b';
}

namespace OwnSpace{
    export let var_b = 'own_space_b';
    let inner_b = '123';
}
```

编译出来的会像这样。

```
var OwnSpace;
(function (OwnSpace) {
    OwnSpace.var_a = 'own_space_b';
})(OwnSpace || (OwnSpace = {}));

(function (OwnSpace) {
    OwnSpace.var_b = 'own_space_b';
    var inner_b = '123';
})(OwnSpace || (OwnSpace = {}));
```

只有 `export` 出来的变量才会挂载到 `OwnSpace` 变量上面。

此时我们再新建一个 `name2.ts`

```
namespace OwnSpace{
    export let var_c = 'own_space_c';
}
```

编译之后，在 `dist` 目录新建一个 `index.html`

```
<meta charset="utf-8">

<script src='namespace.js'></script>
<script src='name2.js'></script>

<script>
    console.log(OwnSpace);
</script>
```

用浏览器打开它，并打开控制台

```
▼ Object
  var_a: "own_space_b"
  var_b: "own_space_b"
  var_c: "own_space_c"
  ► __proto__: Object
```

`namespace.js` 挂载了 `var_a` 和 `var_b` `name2.js` 挂载了 `var_c`

我们看到所有的命名空间下面的东西都是通过 `.` 来访问的，我们可以认为命名空间就是一个对象。

所以 **namespace** 有什么作用呢？

其实非常明显他就是往 `window` 上面挂载某些变量，就像 `jquery`。

当你在 `html` 中引入 `<script src='xxxxxxxx/jquery.js'></script>` 它就会往 `window` 上面挂载一个 `$`

其实这样非常鸡肋，容易造成全局命名空间污染，而且现在都有打包工具了，没必要这么干，之所以会有 `namespace` 可能是为了方便为已有的 `js` 库写 `d.ts`。

所以大多数时候你是用不到 `namespace` 的。

## 配置 `d.ts`

typescript 的代码核心就是 `d.ts`，所以说只要 `d.ts` 写的好，走遍天下都不怕。

## 继续小实验

在之前的小实验里面，再新建俩个文件，`some.d.ts`、`some.js`

```
// some.d.ts
declare const name : string;
export default name;
```

```
// some.js
export defalt const name = 'hello world';
```

在 `main.ts` 文件里面

```
import name from './some';
```

这里我们使用的是相对路径。这样是正确无误的。

创建 `test.d.ts`，定义一个外部模块。外部模块是不是需要 `import` 的啊？

我们发现这里有一个双引号

```
declare module "lodash"{
  export let version : string;
  let _ : any;

  export default _;
}
```

在 `main.ts` 中

```
import { version } from "lodash";
```

这里的双引号就和之前的对应，你会发现这里没有相对路径，而是直接 "lodash"。

这里之所以能找到是因为它们在同一个目录。

而这寻找，他会遍历你项目里面的文件，你不信可以把这个文件移动到项目根目录下。

其实编译器会在 `main.ts` 文件的头部会自动添加一个这样的编译指令，尽管现在你是看不到它的。

这个指令会告诉编译器去哪寻找 `d.ts` 文件，表示这个文件使用了 `d.ts` 里面声明的名字；并且，这个包要在编译阶段与声明文件一起被包含进来

```
/// <reference path="../test.d.ts" />
```

在你的 `tsconfig.json` 里面配置这一项。

```
"listFiles": true
```

```
# Yugo @ Tractor in ~/Desktop/ts-modules [14:05:24] C:2
$ tsc -d
/usr/Local/Lib/node_modules/typescript/Lib/Lib.d.ts
/Users/Yugo/Desktop/ts-modules/test.d.ts
/Users/Yugo/Desktop/ts-modules/node_modules/@types/jquery/index.d.ts
/Users/Yugo/Desktop/ts-modules/src/some.d.ts
/Users/Yugo/Desktop/ts-modules/src/main.ts
/Users/Yugo/Desktop/ts-modules/src/modules.ts
/Users/Yugo/Desktop/ts-modules/src/name2.ts
/Users/Yugo/Desktop/ts-modules/src/namespace.ts
```

而且还可以配置去哪找，分别是 `typeRoots` 和 `types` 选项。

所有的选项都在这

```
https://www.tslang.cn/docs/handbook/compiler-options.html
```

而关于编译器怎么遍历 `.d.ts` 目录。

在你的 `tsconfig.json` 里面配置这一项。

```
"traceResolution": true
```

再次编译你就可以看到打印出了寻找路径。

```
Yugo@Tractor: ~/Desktop/ts-modules
File '/Users/node_modules/Lodash/index.d.ts' does not exist.
File '/Users/node_modules/@types/Lodash.d.ts' does not exist.
File '/Users/node_modules/@types/Lodash/package.json' does not exist.
File '/Users/node_modules/@types/Lodash/index.d.ts' does not exist.
File '/node_modules/Lodash/index.d.ts' does not exist.
File '/node_modules/Lodash.ts' does not exist.
File '/node_modules/Lodash.d.ts' does not exist.
File '/node_modules/Lodash/package.json' does not exist.
File '/node_modules/Lodash/index.ts' does not exist.
File '/node_modules/Lodash/index.tsx' does not exist.
File '/node_modules/Lodash/index.d.ts' does not exist.
File '/node_modules/@types/Lodash.d.ts' does not exist.
File '/node_modules/@types/Lodash/package.json' does not exist.
File '/node_modules/Lodash/index.js' does not exist.
Loading module 'Lodash' from 'node_modules' folder.
File '/Users/Yugo/Desktop/ts-modules/src/node_modules/Lodash.js' does not exist.
File '/Users/Yugo/Desktop/ts-modules/src/node_modules/Lodash.jsx' does not exist.
File '/Users/Yugo/Desktop/ts-modules/src/node_modules/Lodash/package.json' does not exist.
File '/Users/Yugo/Desktop/ts-modules/src/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/ts-modules/src/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/ts-modules/node_modules/Lodash.js' does not exist.
File '/Users/Yugo/Desktop/ts-modules/node_modules/Lodash.jsx' does not exist.
File '/Users/Yugo/Desktop/ts-modules/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/ts-modules/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.js' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
File '/Users/Yugo/Desktop/node_modules/Lodash/index.jsx' does not exist.
==== Module name 'Lodash' was not resolved. ====
==== Resolving type reference directive 'jquery', containing file '/Users/Yugo/Desktop/ts-modules/_inferred type names...ts', root directory '/Users/Yugo/Desktop/ts-modules/node_modules/@types'. ====
Resolving with primary search path '/Users/Yugo/Desktop/ts-modules/node_modules/@types'
Found 'package.json' at '/Users/Yugo/Desktop/ts-modules/node_modules/@types/jquery/package.json'.
'package.json' does not have a 'types' or 'main' field.
File '/Users/Yugo/Desktop/ts-modules/node_modules/@types/jquery/index.d.ts' exist - use it as a name resolution result.
```

里面还有非常多的配置选项，自己去尝试一下，算是留给大家的课后作业。

学习的最好办法就是尝试。

看过 angular2 的人基本都知道，它大量使用了装饰器，而装饰器还属于 es6 的征集意愿的第一阶段。

使用装饰器，需要在 tsconfig.json 里面开启支持选项 experimentalDecorators

```
// tsconfig.json
"experimentalDecorators": true
```

装饰器基本可以对所有变量起作用，它的语法是 `@装饰器`，这个装饰器必须是函数。

## 对类使用装饰器

代码 main.ts

```
function addAge(ctor: Function) {
    ctor.prototype.age = 18;
}

@addAge
class Hello{
    name: string;
    age: number;
    constructor() {
        console.log('hello');
        this.name = 'yugo';
    }
}

let hello = new Hello();

console.log(hello.age);
```

编译之后运行，得到的结果

```
$ node dist/main.js
hello
18
```

当装饰器作为修饰类的时候，会把构造器传递进去。

`constructor.prototype.age` 就是在每一个实例化对象上面添加一个 `age` 值

这里我们的 `addAge` 就添加了一个 `age` 值。

## 对方法使用装饰器

```
function addAge(constructor: Function) {
    constructor.prototype.age = 18;
}

function method(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log(target);
    console.log("prop " + propertyKey);
    console.log("desc " + JSON.stringify(descriptor) + "\n\n");
};

@addAge
class Hello{
    name: string;
    age: number;
    constructor() {
        console.log('hello');
        this.name = 'yugo';
    }

    @method
    hello(){
        return 'instance method';
    }

    @method
    static shello(){
        return 'static method';
    }
}
```

我们得到的结果是

```
$ node dist/main.js
Hello { hello: [Function] }
prop hello
desc {"writable":true,"enumerable":true,"configurable":true}

{ [Function: Hello] shello: [Function] }
prop shello
desc {"writable":true,"enumerable":true,"configurable":true}
```

假如我们修饰的是 `hello` 这个实例方法，第一个参数将是原型对象，也就是 `Hello.prototype`。

假如是 `shello` 这个静态方法，则第一个参数是构造器 `constructor`。

第二个参数分别是属性名，第三个参数是属性修饰对象。

## setter 和 getter 使用装饰器

```
function addAge(constructor: Function) {
    constructor.prototype.age = 18;
}

function method(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log(target);
    console.log(Hello.prototype);
    console.log("prop " + propertyKey);
    console.log("desc " + JSON.stringify(descriptor) + "\n\n");
};

function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        descriptor.configurable = value;
    };
}
```

```
@addAge
class Hello{
    name: string;
    age: number;
    constructor() {
        console.log('hello');
        this.name = 'yugo';
    }

    @method
    hello(){
        return 'instance method';
    }

    @method
    static shello(){
        return 'static method';
    }

    @configurable(false)
    get own_name(){
        return this.name;
    }
}
```

这里的作用就是让这个函数不可再配置

我们的装饰器就像这样 `@configurable(false)` 使用，我们发现这里多了一个`()`并且写的方法跟之前的也不一样的。

我们分开来看，`()` 代表函数调用，先进行函数调用 `configurable(false)`，然后再把它的返回值作为装饰器调用。

```
@configurable(false)
get own_name() {
    return function configurable(value: boolean): (target: any, propertyKey: string, descriptor: PropertyDescriptor) => void
```

可以看到它的返回值函数与我们的之前说的装饰器的函数参数类型一致。

一个函数返回一个函数叫做函数柯里化，也有的人喜欢叫它工厂函数。

有的时候，我们需要很多函数，但是这些函数就是里面的某一个值不一样而已，例如加三函数、加二十函数，不一样的地方就是一个要加的数，假如我们都写一次，那势必会很麻烦。

所以我们就用一个函数返回一个函数，内层作用域可以访问外层作用域的原理（闭包），创建了人们常说的工厂函数。而需要传递进去的参数可以理解为规格。

在现实生活中就是，根据订单的规格要求工厂生成特定规格的产品。

而内层作用域可以访问外层作用域，当领导来视察了，地方的员工就会收到通知。所以地方员工是可以察觉外面的领导的。

而外层作用域不能访问内层作用域（闭包），也就是天高皇帝远，猴子称大王。总部哪管得到我们这小地方。

## 属性装饰器

跟上面使用一样，但是没有第三个参数，只有前面两个。因为，此刻的属性还没有初始化，哪来的配置项。

## 装饰器运行的顺序

```
class C {  
    @f()  
    @g()  
    method() {}  
}
```

执行顺序为

- f()
- g()
- @g
- @f

## 其他高级用法

这里暂不进行赘述，因为只有开发库才可能用到。

更多详情请了解 `reflect-metadata` 这个库。

## 混合对象

大家叫他 `Mixins`，其实原理非常简单，`implements` 只会继承属性的类型，而不会继承实际的逻辑。

之所以需要声明属性，是因为在 `ts` 中只有声明过的属性，才可以被赋值。要不然会报错。

```
class a {
    namea: string;

}

class b {
    nameb: string;
    changeb() {
        this.nameb = 'changed';
    }
}

class c implements a, b {
    namea: string;
    nameb: string;

    changeb(): void { }

}

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}

applyMixins(c, [a, b]);
```

这里的核心就是 `applyMixins`，第一个参数是要混合的主体，第二个参数是要混入的对象数组，主要逻辑就是把原型链上面的方法拷贝到要混合的主体上面。

```

class c implements a {
    namea: string;
    // nameb: string;

    changeb(): void { }
}

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}

applyMixins(c, [a, b]);

let c_instance = new c();

c_instance.changeb();
c_instance.nameb

```

假如我们把 `implements` 的 `b` 去掉，保留在 `applyMixins` 中的 `b`。

此时尽管 js 运行起来能找到 `nameb`，但是在 ts 编译这个阶段是报错的。

有非常多的人这样跟我抱怨 ts，说我写 ts 总是报各种错误，让人烦躁。其实我想告诉你的是，计算机是不会骗人的，报错，说明你的逻辑不对，不符合规范，当然跟你没有认真阅读文档有一定的关系，当你真正理解了 ts 之后，你会发现，ts 就是后端转前端开发人员的天助神器。

TS 玩的顺溜不顺溜，就看你的 `d.ts` 文件写的溜不溜。

在学如何书写声明文件之前，我们先来看看声明相关的一些东西。

## 接口合并

当我们多次使用 `interface` 定义的时候，会合并接口

```
interface A {  
  name: string;  
}  
  
interface A {  
  age: number;  
}  
  
let obj: A;  
  
obj = { name: "123" }  
  
Type '{ name: string; }' is not assignable to type 'A'.  
  Property 'age' is missing in type '{ name: string; }'.  
let obj: A
```

这里报错的原因是，我们并没有完全的实现 `A` 接口。

错误提示告诉我们，还有一个 `age` 属性没有。

```
type t = keyof A;  
  
type t = "name" | "age"
```

假如你使用的 2.1 版本的 ts，那么你可以用 `keyof` 关键字拿到 `A` 的所有属性值类型。

## 命名空间的合并

此时必须要导出，不导出哪怕合并了，外面也访问不到。

真相只有一个，如下图。

```
namespace B{
    export function inner_one(){}
}

namespace B{
    export function inner_two(){}
}

B.|  
  inner_one      function  
  inner_two      function
```

命名空间与其他的合并。

命名空间与类，实现内部类。

```
class Album {
    label: Album.AlbumLabel;
}

namespace Album {
    export class AlbumLabel {
        static id = 'inner';
    }
}
```

此时有一个有意思的小问题

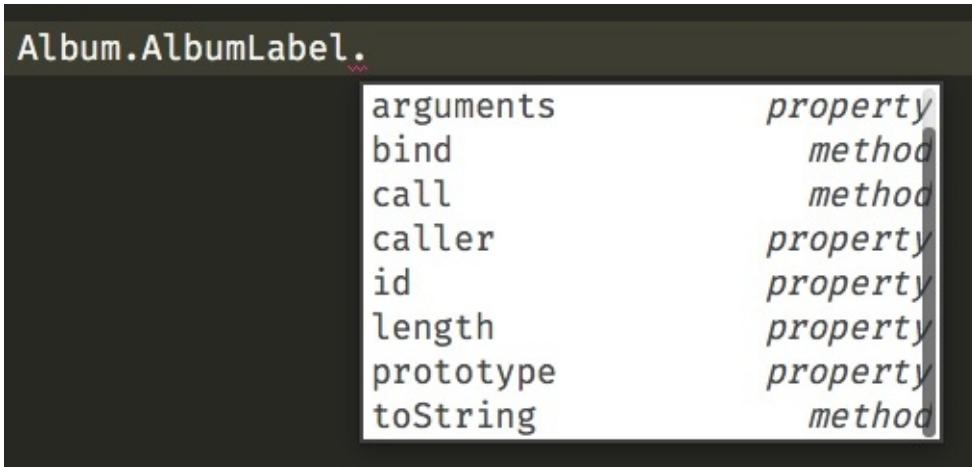
```
let album = new Album()

let label = album.label;

let id = label.
```

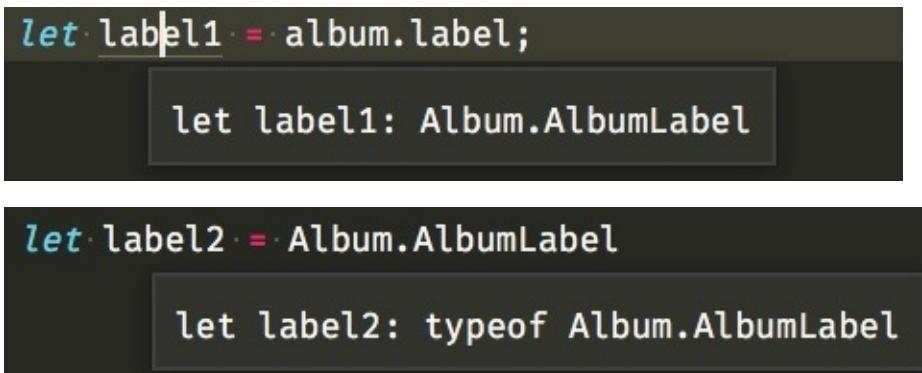
你会发现实例化后的 `label` 并不能访问它的 `id` 属性。

而直接访问却可以。



so? 哪出了问题? 是不是骗我?

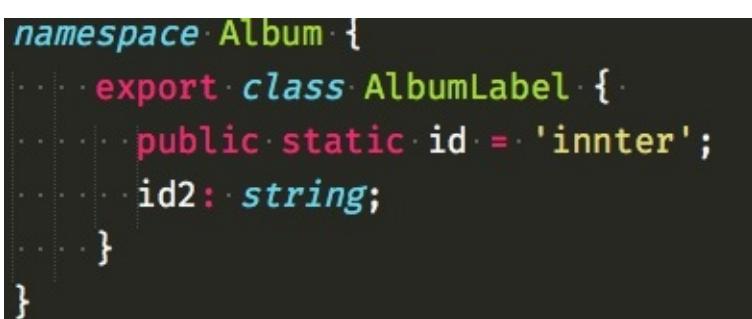
别着急，我们来看一看类型。



假如你认为 `typeof` 出问题了，于是你会花大量的时间去搜索关于它的资料。

其然不是，而是它们本身就不一样。

`label: Album.AlbumLabel;` 表示的是 `Album.AlbumLabel` 的实例，我们此时没有声明任何实例属性。



此时我们加一个示例属性。

```
let label1 = album.label;
```

```
label1.
```

```
    id2           property
```

是不是就可以正常访问了呢。

从上面的例子我们可以看出，假如想让它指向的是一个带有构造器的类，而不是实例，我们可以用 `typeof`。

```
class Album {  
  ... label: typeof Album.AlbumLabel;  
}
```

同样还可以给方法上面添加一些东西，比如静态属性。

```
function func(name: string) {  
  ...  
}  
  
namespace func {  
  ... export let id: string = "123";  
}
```

```
func.i  
  id           let
```

还可以跟枚举配合

```

enum Color{
  R = 1,
  G,
  B,
}

namespace Color {
  export function mixColor(colorName: string) {
    if (colorName === "yellow") {
      return Color.R + Color.G;
    }
    else if (colorName === "white") {
      return Color.R + Color.G + Color.B;
    }
    else if (colorName === "magenta") {
      return Color.R + Color.B;
    }
    else if (colorName === "cyan") {
      return Color.G + Color.B;
    }
  }
}

let yellow = Color.mixColor('yellow')

```

这里有一个缺陷就是得到的 yellow 是一个 number，得出来之后，就无法知道具体是由哪些颜色混合而成的了。

## 扩展一些类

新建 ob.ts 文件

```

export class Observable<T> {
}

```

## 新建 map.ts 文件

```

import { Observable } from "./ob";

declare module "./ob" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}

Observable.prototype.map = function (f) {
    let rets = f();
    return new Observable<typeof rets>();
}

let o: Observable<number> = new Observable();
let newValue = o.map(x => x.toFixed());

```

```

import { Observable } from "./ob";

declare module "./ob" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}

Observable.prototype.map = function (f) {
    let rets = f();
    return new Observable<typeof rets>();
}

let o: Observable<number> = new Observable();
let newValue = o.map(x => x.toFixed());

newValue.

```

map                  method

原理就是往原型上面挂载一些方法，其他细节，之前内容都有提到过，不懂的需要复习一下前面的知识。

温故而知新。

假如你想往全局上面扩展方法，你可以这样做。

把你想要扩展的写到 `declare global` 里面。

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

## 识别全局变量

就像 JQuery 那样，在浏览器中全局就可以访问的对象。通常我们会使用 `namespace`，好处就是防止命名冲突。

通常全局变量在源码中会有如下特性

- 顶级的`var`语句或`function`声明
- 挂载变量到 `window` 上

声明可能会像这样

```
declare namespace jQuery{
    export let $ : { version: string };
}

let $ = jQuery.$;

declare global{
    interface Window {
        $;
    }
}

$.|
```

version      property

当然可能跟官方不一样，咱先暂时这样。

## 识别模块化库

所有需要 import require 的都是模块库。

假如依赖是的是全局库

```
/// <reference types="yourLib" />
```

一种方式是通过指令包含，假如我们每一个文件都写一个这个，这样会非常的烦，所以你可以去 tsconfig 里面去配置，分别是 types 指定文件， typeRoots 指定目录，选择一样即可。

## 定义 declare

所有 d.ts 文件里面声明，都需要加上 declare ，d.ts 只能声明，不能有任何默认值。

```
declare let version: 2.20;
```

```
/Users/Yugo/Desktop/ts-modules/src/some.d.ts [1 errors]
(4, 21) Initializers are not allowed in ambient contexts.
```

编译器告诉我们，该上下文不允许初始化。

定义一个全局变量 `version` 如下

```
declare let version: number;
```

定义一个全局函数

```
declare function func(name: string): number;
```

定义一个全局对象下面具有某些属性

之前在说 `namespace` 的时候，有提到过，其实 `namespace` 就是对象。

```
declare namespace someObj{
    export let name: string;
    export let age: number;
    export function sayHello(): void;
}
```

```
declare namespace someObj{
    export let name: string;
    export let age: number;
    export function sayHello(): void;
}
```

`someObj.`

age	let
name	let
sayHello	function

### 使用接口和函数重载

在定义文件里面已经可以使用接口，因为接口也是描述类型的一种。

```
declare namespace someObj{
    export let name: string;
    export let age: number;
    export function sayHello(): void;
}

interface GreetingSettings {
    greeting: string;
    duration?: number;
    color?: string;
}

declare function greet(setting: GreetingSettings | {}): void;
declare function greet(setting: string): void;
```

### 定义类

```
declare class Greeter {
    constructor(greeting: string);

    greeting: string;
    showGreeting(): void;
}
```

其实这些都不难，就是加上 `declare`，且只写描述类型不写具体实现而已。

## 回调的可选参数

对于定义回调的可选参数，有一点你必须要注意。

```

interface Fetcher {
  getObject(done: (data: any, elapsedTime: number) => void): void;
}

let f : Fetcher = {
  getObject(done){
    done("123", 22)
  }
}

f.getObject((data) => data)
  
```

getObject(done: (data: any, elapsedTime: number) => void): void  
done:  
1/1

对于传入的回调 `done` 我们可以选择不要第二个参数，因为回调允许抛弃一些参数，这样是不会报错的，而假如我们在 `getObject` 方法里面少传一个参数就会报错。

```

let f : Fetcher = {
  getObject(done){
    done("123", )
  }
}
  
```

done(data: any, elapsedTime: number): void  
elapsedTime:  
1/1

(18, 5) Supplied parameters do not match any signature of call target.

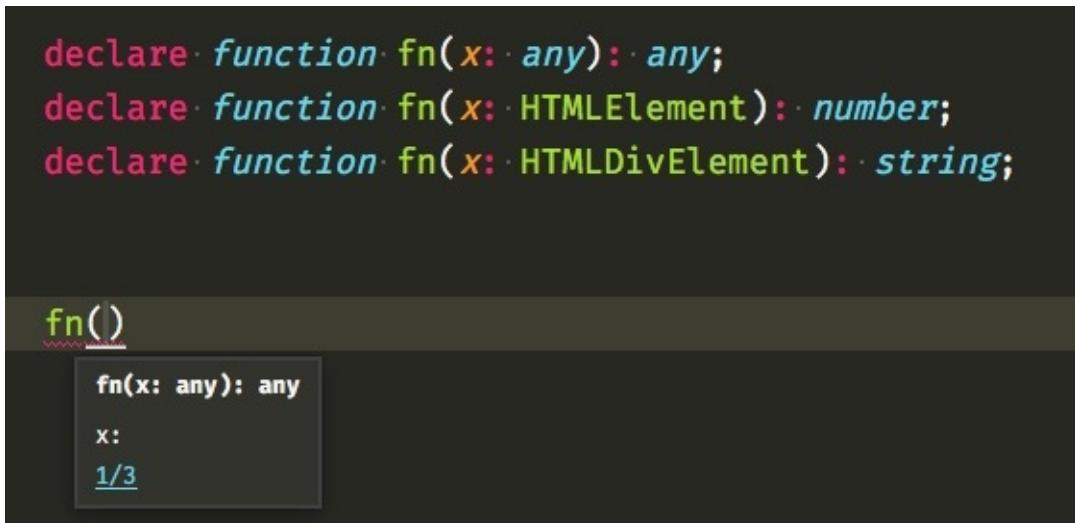
编译器告诉我们类型不匹配。

## 定义参数重载的注意事项

应该从小范围到大范围，如下。

```

declare function fn(x: HTMLDivElement): string;
declare function fn(x: HTMLElement): number;
declare function fn(x: any): any;
  
```



假如你这样写，对其他人来说，开发体验不友好，刚开始就搞那么大的新闻，而且 ts 匹配到第一个就直接调用了，也就是说这样写，基本上全调用的 (x:any)。

## 类型合并的声明

我们之前是不是说过类型合并，我们可以用到这里来。

```

export var Bar: { a: Bar };
export interface Bar {
    count: number;
}

```

这里的 Bar 就合并成了 var Bar = { a: Bar, count: number }，

你可以认为 interface Bar 为该对象添加了一个 count 属性。

此时 Bar 可以当做类型，也就是接口所描述的那样，有一个 count

也可以当做对象。



此时只有只有 count 属性，是因为它的类型是被 interface Bar 描述的。

讲道理可以当类型，又可以当值，这种开发姿势非常魔性，不懂它的人只能吐血了，或许当你遇到什么变态的需求的时候，可能需要它。

## 关于导出

我不管你从哪知道的 `export = something` 语法，别用，你会发现我从来没有提到这个东西，是因为它已经快被淘汰了，尽可能的使用 `export default something` 。

为了更好的学习如何写好 d.ts 大家可以阅读这里的示例文件 [地址](#)

```
tsc --init
```

更多配置项请见[这里](#)

```
https://www.tslang.cn/docs/handbook/compiler-options.html
```

## 添加自己的 **typings** 文件夹

添加以下配置项。就可以加载自己的 **d.ts** 文件了

```
{
  "compilerOptions": {
    "typeRoots": [
      "typings"
    ]
  }
}
```

## 下载已经写好的 **d.ts** 文件

**xxx** 就是你的库的名字。

```
npm install @types/xxx -D
```

这个有点像 **lodash**，因为这只是安装一个小部件。

```
npm install lodash/assign -S
```

**@types** 这个包存放了所有别人已经写好的 **d.ts** 文件。可以自己先到 **npmjs** 去搜索看看有没有。

## 假如没有库的 **d.ts** 文件

有的库会报错，而有的不会，那怎么解决这个问题呢？

配置好自己 typings 目录，在该目录下新建一个 xxx.d.ts ，这个 xxx 你可以随意写。

```
declare module "koa" {  
    interface Context {  
        render(filename: string, ...args: any[]) : any;  
        session: any;  
        i18n: any;  
        csrf: any;  
        flash: any;  
    }  
}
```

这里注意了 "koa" 就是你报错库的名称，记得别忘记双引号。

这里我就是给 koa 库添加一些属性，防止代码编辑器报错。

还有一点你需要注意的是，报错一定是因为该包主目录下没有一个 index.js ，它可能放到 lib 目录下面了，所以会报错。新版本的 ts 只要你安装了库，并且它的下面有 index.js 可以加载到，ts 是不会报错的，只是会让你导入的库是 any 类型的。

## 如何发布 **d.ts** 文件

第一种方式就是在你的库下面的 package.json 里面配置。

这里最好写上相对路径

```
"types": "./lib/main.d.ts"  
  
// or  
"typings": "./lib/main.d.ts"
```

第二种方式是给这个地址提交 PR

<https://github.com/DefinitelyTyped/DefinitelyTyped.git>

假如把 **@types** 里面的东西 **download** 到自己的私有源里面

克隆下来跑一次就可以了

```
https://github.com/Microsoft/types-publisher
```

我一万个建议你把 DefinitelyTyped 仓库下载下来阅读。