

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。TypeScript 提供最新的和不断发展的 JavaScript 特性，包括那些来自 2015 年的 ECMAScript 和未来的提案中的特性，比如异步功能和 Decorators，以帮助建立健壮的组件。

阿宝哥第一次使用 TypeScript 是在 Angular 2.x 项目中，那时候 TypeScript 还没有进入大众的视野。然而现在学习 TypeScript 的小伙伴越来越多了，本文阿宝哥将分享这些年在学习 TypeScript 过程中，曾被困扰过的一些 TS 问题，希望本文对学习 TypeScript 的小伙伴能有一些帮助。

好的，下面我们来开始介绍第一个问题 —— 如何在 window 对象上显式设置属性。

## 一、如何在 window 对象上显式设置属性

对于使用过 JavaScript 的开发者来说，对于 `window.MyNamespace = window.MyNamespace || {};` 这行代码并不会陌生。为了避免开发过程中出现冲突，我们一般会为某些功能设置独立的命名空间。

然而，在 TS 中对于 `window.MyNamespace = window.MyNamespace || {};` 这行代码，TS 编译器会提示以下异常信息：

```
Property 'MyNamespace' does not exist on type 'Window & typeof globalThis'.
```

以上异常信息是说在 `Window & typeof globalThis` 交叉类型上不存在 `MyNamespace` 属性。那么如何解决这个问题呢？最简单的方式就是使用类型断言：

```
(window as any).MyNamespace = {};
```

虽然使用 `any` 大法可以解决上述问题，但更好的方式是扩展 `lib.dom.d.ts` 文件中的 `Window` 接口来解决上述问题，具体方式如下：

```
declare interface Window {  
  MyNamespace: any;  
}  
  
window.MyNamespace = window.MyNamespace || {};
```

下面我们再来看一下 `lib.dom.d.ts` 文件中声明的 `Window` 接口：

```
/**  
 * A window containing a DOM document; the document property  
 * points to the DOM document loaded in that window.  
 */  
interface Window extends EventTarget, AnimationFrameProvider, GlobalEventHandlers, WindowEventHandlers, WindowLocalStorage, WindowOrWorkerGlobalScope, WindowS  
  // 已省略大部分内容  
  readonly devicePixelRatio: number;  
  readonly document: Document;  
  readonly top: Window;  
  readonly window: Window & typeof globalThis;  
  addEventListener(type: string, listener: EventListenerOrEventListenerObjc
```

```
options?: boolean | AddEventListenerOptions): void;
removeEventListener<K extends keyof WindowEventMap>(type: K,
  listener: (this: Window, ev: WindowEventMap[K]) => any,
  options?: boolean | EventListenerOptions): void;
[index: number]: Window;
}
```

在上面我们声明了两个相同名称的 `Window` 接口，这时并不会造成冲突。TypeScript 会自动进行接口合并，即把双方的成员放到一个同名的接口中。

## 二、如何为对象动态分配属性

在 JavaScript 中，我们可以很容易地为对象动态分配属性，比如：

```
let developer = {};
developer.name = "semlinker";
```

以上代码在 JavaScript 中可以正常运行，但在 TypeScript 中，编译器会提示以下异常信息：

```
Property 'name' does not exist on type '{}'.(2339)
```

`{}` 类型表示一个没有包含成员的对象，所以该类型没有包含 `name` 属性。为了解决这个问题，我们可以声明一个 `LooseObject` 类型：

```
interface LooseObject {
  [key: string]: any
}
```

该类型使用 **索引签名** 的形式描述 `LooseObject` 类型可以接受 `key` 类型是字符串，值的类型是 `any` 类型的字段。有了 `LooseObject` 类型之后，我们就可以通过以下方式来解决上述问题：

```
interface LooseObject {
  [key: string]: any
}

let developer: LooseObject = {};
developer.name = "semlinker";
```

对于 `LooseObject` 类型来说，它的约束是很宽松的。在一些应用场景中，我们除了希望能支持动态的属性之外，也希望能够声明一些必选和可选的属性。

比如对于一个表示开发者的 `Developer` 接口来说，我们希望它的 `name` 属性是必填，而 `age` 属性是可选的，此外还支持动态地设置字符串类型的属性。针对这个需求我们可以这样做：

```
interface Developer {
  name: string;
  age?: number;
  [key: string]: any
}

let developer: Developer = { name: "semlinker" };
developer.age = 30;
developer.city = "XiaMen";
```

其实除了使用 **索引签名** 之外，我们也可以使用 TypeScript 内置的工具类型 `Record` 来定义 `Developer` 接口：

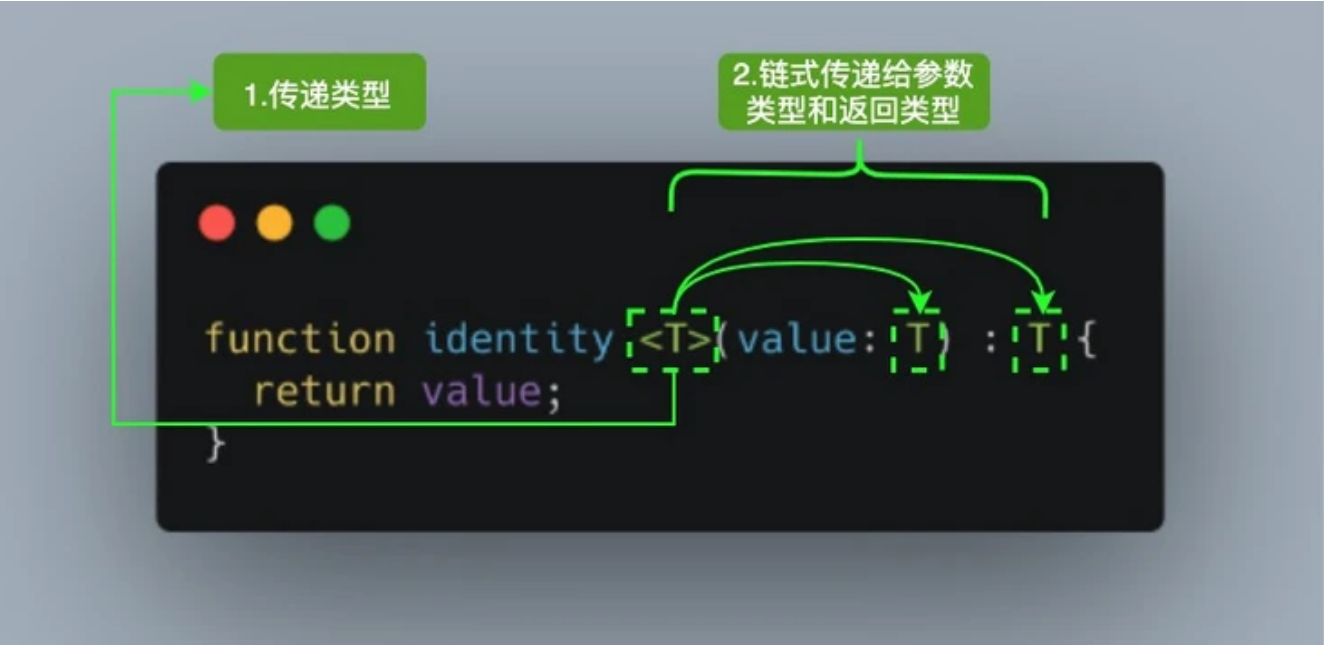
```
// type Record<K extends string | number | symbol, T> = { [P in K]: T; }
interface Developer extends Record<string, any> {
  name: string;
```

```
age?: number;
}

let developer: Developer = { name: "semlinker" };
developer.age = 30;
developer.city = "XiaMen";
```

三、如何理解泛型中的 <T>

对于刚接触 TypeScript 泛型的读者来说，首次看到 <T> 语法会感到陌生。其实它没有什么特别，就像传递参数一样，我们传递了我们想要用于特定函数调用的类型。



参考上面的图片，当我们调用 identity<Number>(1)，Number 类型就像参数 1 一样，它将在出现 T 的任何位置填充该类型。图中 <T> 内部的 T 被称为类型变量，它是我们希望传递给 identity 函数的类型占位符，同时它被分配给 value 参数用来代替它的类型：此时 T 充当的是类型，而不是特定的 Number 类型。

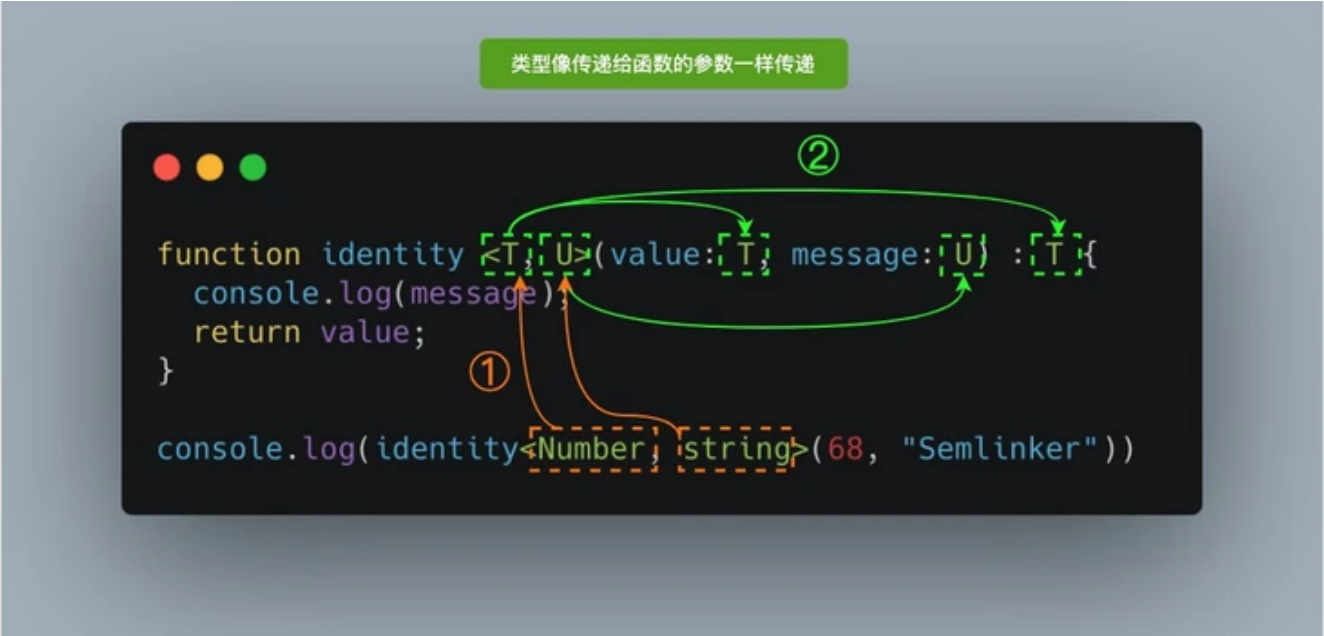
其中 T 代表 **Type**，在定义泛型时通常用作第一个类型变量名称。但实际上 T 可以用任何有效名称代替。除了 T 之外，以下是常见泛型变量代表的意思：

- K (Key)：表示对象中的键类型；
- V (Value)：表示对象中的值类型；
- E (Element)：表示元素类型。

其实并不是只能定义一个类型变量，我们可以引入希望定义的任何数量的类型变量。比如我们引入一个新的类型变量 U，用于扩展我们定义的 identity 函数：

```
function identity <T, U>(value: T, message: U) : T {
  console.log(message);
  return value;
}

console.log(identity<Number, string>(68, "Semlinker"));
```



除了为类型变量显式设定值之外，一种更常见的做法是使编译器自动选择这些类型，从而使代码更简洁。我们可以完全省略尖括号，比如：

```
function identity <T, U>(value: T, message: U) : T {
  console.log(message);
  return value;
}

console.log(identity(68, "Semlinker"));
```

对于上述代码，编译器足够聪明，能够知道我们的参数类型，并将它们赋值给 T 和 U，而不需要开发人员显式指定它们。

## 四、如何理解装饰器的作用

在 TypeScript 中装饰器分为类装饰器、属性装饰器、方法装饰器和参数装饰器四大类。装饰器的本质是一个函数，通过装饰器我们可以方便地定义与对象相关的元数据。

比如在 `ionic-native` 项目中，它使用 `Plugin` 装饰器来定义 IonicNative 中 Device 插件的相关信息：

```
@Plugin({
  pluginName: 'Device',
  plugin: 'cordova-plugin-device',
  pluginRef: 'device',
  repo: 'https://github.com/apache/cordova-plugin-device',
  platforms: ['Android', 'Browser', 'iOS', 'macOS', 'Windows'],
})
@Injectable()
export class Device extends IonicNativePlugin {}
```

在以上代码中 Plugin 函数被称为装饰器工厂，调用该函数之后会返回类装饰器，用于装饰 Device 类。Plugin 工厂函数的定义如下：

```
// https://github.com/ionic-team/ionic-native/blob/v3.x/src/%40ionic-native/c
export function Plugin(config: PluginConfig): ClassDecorator {
  return function(cls: any) {
    // 把config对象中属性，作为静态属性添加到cls类上
    for (let prop in config) {
      cls[prop] = config[prop];
    }

    cls['installed'] = function(printWarning?: boolean) {
      return !!getPlugin(config.pluginRef);
    };
    // 省略其他内容
    return cls;
  };
}
```

通过观察 Plugin 工厂函数的方法签名，我们可以知道调用该函数之后会返回 `ClassDecorator` 类型的对象，其中 `ClassDecorator` 类型的声明如下所示：

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction)
=> TFunction | void;
```

类装饰器顾名思义，就是用来装饰类的。它接收一个参数 —— `target: TFunction`，表示被装饰器的类。介绍完上述内容之后，我们来看另一个问题 `@Plugin({...})` 中的 `@` 符号有什么用？

其实 `@Plugin({...})` 中的 `@` 符号只是语法糖，为什么说是语法糖呢？这里我们来看一下编译生成的 ES5 代码：

```
var __decorate = (this && this.__decorate) || function (decorators, target, k,
var c = arguments.length, r = c < 3 ? target : desc === null ? desc = Object
if (typeof Reflect === "object" && typeof Reflect.decorate === "function")
else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i])
return c > 3 && r && Object.defineProperty(target, key, r), r;
};

var Device = /** @class */ (function (_super) {
__extends(Device, _super);
function Device() {
return _super !== null && _super.apply(this, arguments) || this;
}
Device = __decorate([
Plugin({
pluginName: 'Device',
plugin: 'cordova-plugin-device',
pluginRef: 'device',
repo: 'https://github.com/apache/cordova-plugin-device',
platforms: ['Android', 'Browser', 'iOS', 'macOS', 'Windows'],
}),
Injectable()
], Device);
return Device;
})(IonicNativePlugin));
```

通过生成的代码可知，`@Plugin({...})` 和 `@Injectable()` 最终会被转换成普通的方法调用，它们的调用结果最终会以数组的形式作为参数传递给 `__decorate` 函数，而在 `__decorate` 函数内部会以 `Device` 类作为参数调用各自的类型装饰器，从而扩展对应的功能。

此外，如果你有使用过 Angular，相信你对以下的代码并不会陌生。

```
const API_URL = new InjectionToken('apiUrl');

@Injectable()
export class HttpService {
constructor(
private httpClient: HttpClient,
@Inject(API_URL) private apiUrl: string
) {}
}
```

在 `Injectable` 类装饰器修饰的 `HttpService` 类中，我们通过构造注入的方式注入了用于处理 HTTP 请求的 `HttpClient` 依赖对象。而通过 `Inject` 参数装饰器注入了 `API_URL` 对应的对象，这种方式我们称之为依赖注入（Dependency Injection）。

关于什么是依赖注入，在 TS 中如何实现依赖注入功能，出于篇幅考虑，这里阿宝哥就不继续展开了。感兴趣的小伙伴可以阅读“了不起的 IoC 与 DI”这篇文章。

## 五、如何理解函数重载的作用

### 5.1 可爱又可恨的联合类型

由于 JavaScript 是一个动态语言，我们通常会使用不同类型的参数来调用同一个函数，该函数会根据不同的参数而返回不同的类型的调用结果：

```
function add(x, y) {
return x + y;
}

add(1, 2); // 3
add("1", "2"); //"12"
```



由于 TypeScript 是 JavaScript 的超集，因此以上的代码可以直接在 TypeScript 中使用，但当 TypeScript 编译器开启 `noImplicitAny` 的配置项时，以上代码会提示以下错误信息：

```
Parameter 'x' implicitly has an 'any' type.
Parameter 'y' implicitly has an 'any' type.
```

该信息告诉我们参数 `x` 和参数 `y` 隐式具有 `any` 类型。为了解决这个问题，我们可以为参数设置一个类型。因为我们希望 `add` 函数同时支持 `string` 和 `number` 类型，因此我们可以定义一个 `string | number` 联合类型，同时我们为该联合类型取个别名：

```
type Combinable = string | number;
```

在定义完 `Combinable` 联合类型后，我们来更新一下 `add` 函数：

```
function add(a: Combinable, b: Combinable) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}
```

为 `add` 函数的参数显式设置类型之后，之前错误的提示消息就消失了。那么此时的 `add` 函数就完美了么，我们来实际测试一下：

```
const result = add('semlinker', 'kakuqo');
result.split('');
```

在上面代码中，我们分别使用 `'semlinker'` 和 `'kakuqo'` 这两个字符串作为参数调用 `add` 函数，并把调用结果保存到一个名为 `result` 的变量上，这时候我们想当然的认为此时 `result` 的变量的类型为 `string`，所以我们可以正常调用字符串对象上的 `split` 方法。但这时 TypeScript 编译器又出现以下错误信息了：

```
Property 'split' does not exist on type 'Combinable'.
Property 'split' does not exist on type 'number'.
```

很明显 `Combinable` 和 `number` 类型的对象上并不存在 `split` 属性。问题又来了，那如何解决呢？这时我们就可以利用 TypeScript 提供的函数重载。

## 5.2 函数重载

函数重载或方法重载是使用相同名称和不同参数数量或类型创建多个方法的一种能力。

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;
function add(a: Combinable, b: Combinable) {
  // type Combinable = string | number;
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}
```

在以上代码中，我们为 `add` 函数提供了多个函数类型定义，从而实现函数的重载。在 TypeScript 中除了可以重载普通函数之外，我们还可以重载类中的成员方法。

方法重载是指在同一个人类中方法同名，参数不同（参数类型不同、参数个数不同或参数个数相同时参数的先后顺序不同），调用时根据实参的形式，选择与它匹配的方法执行操作的一种技

术。所以类中成员方法满足重载的条件是：在同一个类中，方法名相同且参数列表不同。下面我们来举一个成员方法重载的例子：

```
class Calculator {
  add(a: number, b: number): number;
  add(a: string, b: string): string;
  add(a: string, b: number): string;
  add(a: number, b: string): string;
  add(a: Combinable, b: Combinable) {
    if (typeof a === 'string' || typeof b === 'string') {
      return a.toString() + b.toString();
    }
    return a + b;
  }
}

const calculator = new Calculator();
const result = calculator.add('Semlinker', ' Kakuqo');
```

这里需要注意的是，当 TypeScript 编译器处理函数重载时，它会查找重载列表，尝试使用第一个重载定义。如果匹配的话就使用这个。因此，在定义重载的时候，一定要把最精确的定义放在最前面。另外在 Calculator 类中，`add(a: Combinable, b: Combinable){ }`并不是重载列表的一部分，因此对于 add 成员方法来说，我们只定义了四个重载方法。

## 六、interfaces 与 type 之间有什么区别

### 6.1 Objects/Functions

接口和类型别名都可以用来描述对象的形状或函数签名：

接口

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

interface SetPoint {
  (x: number, y: number): void;
}
```

类型别名

```
type Point = {
  x: number;
  y: number;
};

type SetPoint = (x: number, y: number) => void;
```

### 6.2 Other Types

与接口类型不一样，类型别名可以用于一些其他类型，比如原始类型、联合类型和元组：

```
// primitive
type Name = string;

// object
type PartialPointX = { x: number; };
type PartialPointY = { y: number; };

// union
type PartialPoint = PartialPointX | PartialPointY;
```

```
// tuple
type Data = [number, string];
```

## 6.3 Extend

接口和类型别名都能够被扩展，但语法有所不同。此外，接口和类型别名不是互斥的。接口可以扩展类型别名，而反过来是不行的。

### Interface extends interface

```
interface PartialPointX { x: number; }
interface Point extends PartialPointX {
  y: number;
}
```

### Type alias extends type alias

```
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };
```

### Interface extends type alias

```
type PartialPointX = { x: number; };
interface Point extends PartialPointX { y: number; }
```

### Type alias extends interface

```
interface PartialPointX { x: number; }
type Point = PartialPointX & { y: number; };
```

## 6.4 Implements

类可以以相同的方式实现接口或类型别名，但类不能实现使用类型别名定义的联合类型：

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

class SomePoint implements Point {
  x = 1;
  y = 2;
}

type Point2 = {
  x: number;
  y: number;
};

class SomePoint2 implements Point2 {
  x = 1;
  y = 2;
}

type PartialPoint = { x: number; } | { y: number; };

// A class can only implement an object type or
// intersection of object types with statically known members.
class SomePartialPoint implements PartialPoint { // Error
  x = 1;
```



## 6.5 Declaration merging

与类型别名不同，接口可以定义多次，会被自动合并为单个接口。

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

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

## 七、object, Object 和 {} 之间有什么区别

### 7.1 object 类型

object 类型是：TypeScript 2.2 引入的新类型，它用于表示非原始类型。

```
// node_modules/typescript/lib/lib.es5.d.ts
interface ObjectConstructor {
  create(o: object | null): any;
  // ...
}

const proto = {};

Object.create(proto);    // OK
Object.create(null);     // OK
Object.create(undefined); // Error
Object.create(1337);     // Error
Object.create(true);     // Error
Object.create("oops");   // Error
```

### 7.2 Object 类型

Object 类型：它是所有 Object 类的实例的类型，它由以下两个接口来定义：

- Object 接口定义了 Object.prototype 原型对象上的属性；

```
// node_modules/typescript/lib/lib.es5.d.ts
interface Object {
  constructor: Function;
  toString(): string;
  toLocaleString(): string;
  valueOf(): Object;
  hasOwnProperty(v: PropertyKey): boolean;
  isPrototypeOf(v: Object): boolean;
  propertyIsEnumerable(v: PropertyKey): boolean;
}
```

- ObjectConstructor 接口定义了 Object 类的属性。

```
// node_modules/typescript/lib/lib.es5.d.ts
interface ObjectConstructor {
  /** Invocation via `new` */
  new(value?: any): Object;
  /** Invocation via function calls */
  (value?: any): any;
  readonly prototype: Object;
  getPrototypeOf(o: any): any;
  // ...
}

declare var Object: ObjectConstructor;
```

Object 类的所有实例都继承了 Object 接口中的所有属性。

### 7.3 {} 类型

{} 类型描述了一个没有成员的对象。当你试图访问这样一个对象的任意属性时，TypeScript 会产生一个编译时错误。

```
// Type {}
const obj = {};
```

```
// Error: Property 'prop' does not exist on type '{}'.
obj.prop = "semlinker";
```

但是，你仍然可以使用在 Object 类型上定义的所有属性和方法，这些属性和方法可通过 JavaScript 的原型链隐式地使用：

```
// Type {}
const obj = {};
```

```
// "[object Object]"
obj.toString();
```

## 八、数字枚举与字符串枚举之间有什么区别

### 8.1 数字枚举

在 JavaScript 中布尔类型的变量含有有限范围的值，即 `true` 和 `false`。而在 TypeScript 中利用枚举，你也可以自定义相似的类型：

```
enum NoYes {
  No,
  Yes,
}
```

`No` 和 `Yes` 被称为枚举 `NoYes` 的成员。每个枚举成员都有一个 `name` 和一个 `value`。数字枚举成员值的默认类型是 `number` 类型。也就是说，每个成员的值都是一个数字：

```
enum NoYes {
  No,
  Yes,
}
```

```
assert.equal(NoYes.No, 0);
assert.equal(NoYes.Yes, 1);
```

除了让 TypeScript 为我们指定枚举成员的值之外，我们还可以手动赋值：

```
enum NoYes {
  No = 0,
  Yes = 1,
}
```

这种通过等号的显式赋值称为 `initializer`。如果枚举中某个成员的值使用显式方式赋值，但后续成员未显示赋值，TypeScript 会基于当前成员的值加 1 作为后续成员的值。

### 8.2 字符串枚举

除了数字枚举，我们还可以使用字符串作为枚举成员值：

```
enum NoYes {
  No = 'No',
  Yes = 'Yes',
}
```

```
}

assert.equal(NoYes.No, 'No');
assert.equal(NoYes.Yes, 'Yes');
```

### 8.3 数字枚举 vs 字符串枚举

数字枚举与字符串枚举有什么区别呢？这里我们来分别看一下数字枚举和字符串枚举编译的结果：

#### 数字枚举编译结果

```
"use strict";
var NoYes;
(function (NoYes) {
    NoYes[NoYes["No"] = 0] = "No";
    NoYes[NoYes["Yes"] = 1] = "Yes";
})(NoYes || (NoYes = {}));
```

#### 字符串枚举编译结果

```
"use strict";
var NoYes;
(function (NoYes) {
    NoYes["No"] = "No";
    NoYes["Yes"] = "Yes";
})(NoYes || (NoYes = {}));
```

通过观察以上结果，我们知道数值枚举除了支持 **从成员名称到成员值** 的普通映射之外，它还支持 **从成员值到成员名称** 的反向映射。另外，对于纯字符串枚举，我们不能省略任何初始化程序。而数字枚举如果没有显式设置值时，则会使用默认值进行初始化。

### 8.4 为数字枚举分配越界值

讲到数字枚举，这里我们再来看个问题：

```
const enum Fonum {
    a = 1,
    b = 2
}

let value: Fonum = 12; // Ok
```

相信很多读者看到 `let value: Fonum = 12;` 这一行，TS 编译器并未提示任何错误会感到惊讶。很明显数字 12 并不是 Fonum 枚举的成员。为什么会这样呢？我们来看一下 [TypeScript issues 26362](#) 中 **DanielRosenwasser** 大佬的回答：

The behavior is motivated by bitwise operations. There are times when `SomeFlag.Foo | SomeFlag.Bar` is intended to produce another `SomeFlag`. Instead you end up with number, and you don't want to have to cast back to `SomeFlag`.  
该行为是由按位运算引起的。有时 `SomeFlag.Foo | SomeFlag.Bar` 用于生成另一个 `SomeFlag`。相反，你最终得到的是数字，并且你不想强制回退到 `SomeFlag`。

了解完上述内容，我们再来看一下 `let value: Fonum = 12;` 这个语句，该语句 TS 编译器不会报错，是因为数字 12 是可以通过 Fonum 已有的枚举成员计算而得。

```
let value: Fonum =
    Fonum.a << Fonum.b << Fonum.a |  Fonum.a << Fonum.b; // 12
```

## 九、使用 # 定义的私有字段与 private 修饰符定义字段有什么区别

在 TypeScript 3.8 版本就开始支持 **ECMAScript** 私有字段，使用方式如下：

```
class Person {
  #name: string;

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

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}

let semlinker = new Person("Semlinker");

semlinker.#name;
//      ~~~~~
// Property '#name' is not accessible outside class 'Person'
// because it has a private identifier.
```

与常规属性（甚至使用 **private** 修饰符声明的属性）不同，私有字段要牢记以下规则：

- 私有字段以 # 字符开头，有时我们称之为私有名称；
- 每个私有字段名称都唯一地限定于其包含的类；
- 不能在私有字段上使用 TypeScript 可访问性修饰符（如 public 或 private）；
- 私有字段不能在包含的类之外访问，甚至不能被检测到。

说到这里使用 # 定义的私有字段与 **private** 修饰符定义字段有什么区别呢？现在我们先来看一个 **private** 的示例：

```
class Person {
  constructor(private name: string){}
}

let person = new Person("Semlinker");
console.log(person.name);
```

在上面代码中，我们创建了一个 Person 类，该类中使用 **private** 修饰符定义了一个私有属性 **name**，接着使用该类创建一个 **person** 对象，然后通过 **person.name** 来访问 **person** 对象的私有属性，这时 TypeScript 编译器会提示以下异常：

```
Property 'name' is private and only accessible within class 'Person'.(2341)
```

那如何解决这个异常呢？当然你可以使用类型断言把 **person** 转为 **any** 类型：

```
console.log((person as any).name);
```

通过这种方式虽然解决了 TypeScript 编译器的异常提示，但是在运行时我们还是可以访问到 **Person** 类内部的私有属性，为什么会这样呢？我们来看一下编译生成的 ES5 代码，也许你就知道答案了：

```
var Person = /** @class */ (function () {
  function Person(name) {
    this.name = name;
  }
  return Person;
})();
```

```
var person = new Person("Semlinker");
console.log(person.name);
```

这时相信有些小伙伴会好奇，在 TypeScript 3.8 以上版本通过 # 号定义的私有字段编译后会生成什么代码：

```
class Person {
  #name: string;

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

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}
```

以上代码目标设置为 ES2015，会编译生成以下代码：

```
"use strict";
var __classPrivateFieldSet = (this && this.__classPrivateFieldSet)
  || function (receiver, privateMap, value) {
    if (!privateMap.has(receiver)) {
      throw new TypeError("attempted to set private field on non-instance");
    }
    privateMap.set(receiver, value);
    return value;
  };

var __classPrivateFieldGet = (this && this.__classPrivateFieldGet)
  || function (receiver, privateMap) {
    if (!privateMap.has(receiver)) {
      throw new TypeError("attempted to get private field on non-instance");
    }
    return privateMap.get(receiver);
  };

var _name;
class Person {
  constructor(name) {
    _name.set(this, void 0);
    __classPrivateFieldSet(this, _name, name);
  }
  greet() {
    console.log(`Hello, my name is ${__classPrivateFieldGet(this, _name)}!`);
  }
}
```

通过观察上述代码，使用 # 号定义的 ECMAScript 私有字段，会通过 WeakMap 对象来存储，同时编译器会生成 \_\_classPrivateFieldSet 和 \_\_classPrivateFieldGet 这两个方法用于设置值和获取值。

以上提到的这些问题，相信一些小伙伴们在学习 TS 过程中也遇到了。如果有表述不清楚的地方，欢迎你们给我留言或直接与我交流。之后，阿宝哥还会继续补充和完善这一方面的内容，感兴趣的小伙伴可以一起参与哟。

## 十、参考资源

- [how-do-you-explicitly-set-a-new-property-on-window-in-typescript](#)
- [how-do-i-dynamically-assign-properties-to-an-object-in-typescript](#)
- [typescript-interfaces-vs-types](#)

## 十一、推荐阅读


- [了不起的 TypeScript 入门教程](#)
- [一文读懂 TypeScript 泛型及应用](#)





- 你不知道的 WebSocket
- 你不知道的 Blob
- 你不知道的 WeakMap

前端 typescript

阅读 9.3k · 发布于 2020-09-02

 赞 45

 收藏 22

 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



全栈修仙之路

聚焦全栈，专注分享 Angular、TypeScript、Node.js/Java 、...

关注专栏



阿宝哥

[链接]

15.1k 声望    9.6k 粉丝

关注作者

2 条评论

得票数    最新




撰写评论 ...




提交评论



**lzy2014love**: 好文  
 · 回复 · 2020-09-09



**羽天一**: 写的真好  
 · 回复 · 5 月 14 日

你知道吗？

修正程序错误的第一步是要复现这个错误。

注册登录

继续阅读

「1.8W字」一份不可多得的 TS 学习指南

阿宝哥第一次使用 TypeScript 是在 Angular 2.x 项目中，那时候 TypeScript 还没有进入大众...

阿宝哥 · 阅读 8.9k · 152 赞 · 8 评论

## 细数 TS 中那些奇怪的符号

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且...

阿宝哥 · 阅读 12.6k · 62 赞 · 3 评论

## React + TS

为每个文件创建一个类组件。在一个文件内请尽量只写一个类组件，其他的请使用函数组件...

NikolaChen · 阅读 28.6k · 59 赞 · 6 评论

## 一文让你彻底掌握 TS 枚举

创建了一个 “重学TypeScript” 的微信群，想加群的小伙伴，加我微信"semliker"，备注重学...

阿宝哥 · 阅读 3.1k · 6 赞 · 2 评论

## TS基础应用 & Hook中的TS

{代码...} 若您还不熟悉 TS，那本文可帮助您完成 TS 应用部分的学习，伴随众多 Demo 例来...

袋鼠云数栈前端 · 阅读 572 · 4 赞

## 在 TS 中如何实现类型保护？类型谓词了解一下

在 TypeScript 中，一个变量不会被限制为单一的类型。如果你希望一个变量的值，可以有多...

阿宝哥 · 阅读 1.9k · 4 赞 · 2 评论

## TypeScript 为何如此重要？

每次你声明变量或函数参数时，必须先明确标明它们的类型，然后再使用。这个概念背后的...

方旭 · 阅读 528

## TypeScript 类型编程

到目前为止，用TS也写了不少业务了，大部分时候其实只要有 类 以及 类型 的概念，在用TS...

用户bPbopXz · 阅读 293

### 产品

热门问答

热门专栏

热门课程

最新活动

技术圈

酷工作

### 课程

Java 开发课程

PHP 开发课程

Python 开发课程

前端开发课程

移动开发课程

### 资源

每周精选

用户排行榜

帮助中心

建议反馈

### 合作

关于我们

广告投放

职位发布

讲师招募

联系我们

合作伙伴

### 关注

产品技术日志

社区运营日志

市场运营日志

团队日志

社区访谈

### 条款

服务协议

隐私政策

下载 App

Copyright © 2011-2021 SegmentFault. 当前呈现版本 21.11.11

浙ICP备15005796号-2 浙公网安备33010602002000号 ICP 经营许可  
浙B2-20201554

杭州堆栈科技有限公司版权所有

