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。

本文将分享这些年在学习 TypeScript 过程中，遇到的 **10** 大 “奇怪” 的符号。其中有一些符号，笔者第一次见的时候也觉得 “一脸懵逼”，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符号 —— **！非空断言操作符**。

一、！非空断言操作符

在上下文中当类型检查器无法断定类型时，一个新的后缀表达式操作符 `！` 可以用于断言操作对象是非 null 和非 undefined 类型。**具体而言，x! 将从 x 值域中排除 null 和 undefined 。**

那么非空断言操作符到底有什么用呢？下面我们先来看一下非空断言操作符的一些使用场景。

1.1 忽略 undefined 和 null 类型

```
function myFunc(maybeString: string | undefined | null) {
  // Type 'string | null | undefined' is not assignable to type 'string'.
  // Type 'undefined' is not assignable to type 'string'.
  const onlyString: string = maybeString; // Error
  const ignoreUndefinedAndNull: string = maybeString!; // Ok
}
```

1.2 调用函数时忽略 undefined 类型

```
type NumGenerator = () => number;

function myFunc(numGenerator: NumGenerator | undefined) {
  // Object is possibly 'undefined'.(2532)
  // Cannot invoke an object which is possibly 'undefined'.(2722)
  const num1 = numGenerator(); // Error
  const num2 = numGenerator!(); //OK
}
```

因为 `！` 非空断言操作符会从编译生成的 JavaScript 代码中移除，所以在实际使用的过程中，要特别注意。比如下面这个例子：

```
const a: number | undefined = undefined;
const b: number = a!;
console.log(b);
```

以上 TS 代码会编译生成以下 ES5 代码：

```
"use strict";
const a = undefined;
const b = a;
console.log(b);
```

虽然在 TS 代码中，我们使用了非空断言，使得 `const b: number = a!;` 语句可以通过 TypeScript 类型检查器的检查。但在生成的 ES5 代码中， `!` 非空断言操作符被移除了，所以在浏览器中执行以上代码，在控制台会输出 `undefined` 。

👉 继续阅读*：*介绍了 ?. 和 ?? 运算符，再来个 ! 非空断言操作符

二、?. 运算符

TypeScript 3.7 实现了呼声最高的 ECMAScript 功能之一：可选链（Optional Chaining）。有了可选链后，我们编写代码时如果遇到 `null` 或 `undefined` 就可以立即停止某些表达式的运行。可选链的核心是新的 `?.` 运算符，它支持以下语法：

> ```
> obj?.prop
> obj?.[expr]
> arr?.[index]
> func?.(args)
> ```

这里我们来举一个可选的属性访问的例子：

```
const val = a?.b;
```

为了更好的理解可选链，我们来看一下该 `const val = a?.b` 语句编译生成的 ES5 代码：

```
var val = a === null || a === void 0 ? void 0 : a.b;
```

上述的代码会自动检查对象 a 是否为 `null` 或 `undefined` ，如果是的话就立即返回 `undefined` ，这样就可以立即停止某些表达式的运行。你可能已经想到可以使用 `?.` 来替代很多使用 `&&` 执行空检查的代码：

```
if(a && a.b) { }

if(a?.b){ }
/**
* if(a?.b){ } 编译后的ES5代码
*
* if(
*  a === null || a === void 0
*  ? void 0 : a.b) {
* }
*/
```

但需要注意的是，`?.` 与 `&&` 运算符行为略有不同，`&&` 专门用于检测 `false` 值，比如空字符串、`0`、`NaN`、`null` 和 `false` 等。而 `?.` 只会验证对象是否为 null 或 undefined`，对于 0 或空字符串来说，并不会出现 “短路”。

2.1 可选元素访问

可选链除了支持可选属性的访问之外，它还支持可选元素的访问，它的行为类似于可选属性的访问，只是可选元素的访问允许我们访问非标识符的属性，比如任意字符串、数字索引和 Symbol：

```
function tryGetArrayElement<T>(arr?: T[], index: number = 0) {
  return arr?.[index];
}
```

以上代码经过编译后会生成以下 ES5 代码：

```
"use strict";
function tryGetArrayElement(arr, index) {
    if (index === void 0) { index = 0; }
    return arr === null || arr === void 0 ? void 0 : arr[index];
}
```

通过观察生成的 ES5 代码，很明显在 `tryGetArrayElement` 方法中会自动检测输入参数 arr 的值是否为 `null` 或 `undefined`，从而保证了我们代码的健壮性。

2.2 可选链与函数调用

当尝试调用一个可能不存在的方法时也可以使用可选链。在实际开发过程中，这是很有用的。系统中某个方法不可用，有可能是由于版本不一致或者用户设备兼容性问题导致的。函数调用时如果被调用的方法不存在，使用可选链可以使表达式自动返回 `undefined` 而不是抛出一个异常。

可选调用使用起来也很简单，比如：

```
let result = obj.customMethod?.();
```

该 TypeScript 代码编译生成的 ES5 代码如下：

```
var result = (_a = obj.customMethod) === null
  || _a === void 0 ? void 0 : _a.call(obj);
```

另外在使用可选调用的时候，我们要注意以下两个注意事项：

- 如果存在一个属性名且该属性名对应的值不是函数类型，使用 `?.` 仍然会产生一个 `TypeError` 异常。
- 可选链的运算行为被局限在属性的访问、调用以及元素的访问 —— 它不会沿伸到后续的表达式中，也就是说可选调用不会阻止 `a?.b / someMethod()` 表达式中的除法运算或 `someMethod` 的方法调用。

👉 继续阅读**：**遇到访问对象深层次属性怎么办？可选链了解一下！

三、?? 空值合并运算符

在 TypeScript 3.7 版本中除了引入了前面介绍的可选链 `?.` 之外，也引入了一个新的逻辑运算符 —— 空值合并运算符 `??` 。**当左侧操作数为 null 或 undefined 时，其返回右侧的操作数，否则返回左侧的操作数。**

与逻辑或 `||` 运算符不同，逻辑或会在左操作数为 false 值时返回右侧操作数。也就是说，如果你使用 || 来为某些变量设置默认的值时，你可能会遇到意料之外的行为。比如为 falsy 值（''、NaN 或 0）时。

这里来看一个具体的例子：

```
const foo = null ?? 'default string';
console.log(foo); // 输出: "default string"

const baz = 0 ?? 42;
console.log(baz); // 输出: 0
```

以上 TS 代码经过编译后，会生成以下 ES5 代码：

```
"use strict";
var _a, _b;
var foo = (_a = null) !== null && _a !== void 0 ? _a : 'default string';
console.log(foo); // 输出: "default string"

var baz = (_b = 0) !== null && _b !== void 0 ? _b : 42;
console.log(baz); // 输出: 0
```

通过观察以上代码，我们更加直观的了解到，空值合并运算符是如何解决前面 `||` 运算符存在的潜在问题。下面我们来介绍空值合并运算符的特性和使用时的一些注意事项。

3.1 短路

当空值合并运算符的左表达式不为 `null` 或 `undefined` 时，不会对右表达式进行求值。

```
function A() { console.log('A was called'); return undefined;}
function B() { console.log('B was called'); return false;}
function C() { console.log('C was called'); return "foo";}

console.log(A() ?? C());
console.log(B() ?? C());
```

上述代码运行后，控制台会输出以下结果：

```
A was called
C was called
foo
B was called
false
```

3.2 不能与 && 或 || 操作符共用

若空值合并运算符 `??` 直接与 AND（&&）和 OR（||）操作符组合使用 ?? 是不行的。这种情况下会抛出 SyntaxError。

```
// '||' and '??' operations cannot be mixed without parentheses.(5076)
null || undefined ?? "foo"; // raises a SyntaxError

// '&&' and '??' operations cannot be mixed without parentheses.(5076)
true && undefined ?? "foo"; // raises a SyntaxError
```

但当使用括号来显式表明优先级时是可行的，比如：

```
(null || undefined ) ?? "foo"; // 返回 "foo"
```

3.3 与可选链操作符 ?. 的关系

空值合并运算符针对 undefined 与 null 这两个值，可选链式操作符 `?.` 也是如此。可选链式操作符，对于访问属性可能为 undefined 与 null 的对象时非常有用。

```
interface Customer {
  name: string;
  city?: string;
}

let customer: Customer = {
  name: "Semlinker"
};

let customerCity = customer?.city ?? "Unknown city";
console.log(customerCity); // 输出：Unknown city
```

前面我们已经介绍了空值合并运算符的应用场景和使用时的一些注意事项，该运算符不仅可以在 TypeScript 3.7 以上版本中使用。当然你也可以在 JavaScript 的环境中使用它，但你需要借助 Babel，在 Babel 7.8.0 版本也开始支持空值合并运算符。

*👉 *继续阅读**：*****在 TS 中你踩过默认值的坑么？给 ?? 运算符把坑填了！*

四、?: 可选属性

在面向对象语言中，接口是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类去实现。**TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。**

在 TypeScript 中使用 `interface` 关键字就可以声明一个接口：

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

let semlinker: Person = {
  name: "semlinker",
  age: 33,
};
```

在以上代码中，我们声明了 `Person` 接口，它包含了两个必填的属性 `name` 和 `age` 。在初始化 Person 类型变量时，如果缺少某个属性，TypeScript 编译器就会提示相应的错误信息，比如：

```
// Property 'age' is missing in type '{ name: string; }' but required in type
'Person'.(2741)
let lolo: Person  = { // Error
  name: "lolo"
}
```

为了解决上述的问题，我们可以把某个属性声明为可选的：

```
interface Person {
  name: string;
  age?: number;
}

let lolo: Person  = {
  name: "lolo"
}
```

4.1 工具类型

4.1.1 `Partial<T>`

在实际项目开发过程中，为了提高代码复用率，我们可以利用 TypeScript 内置的工具类型 `Partial<T>` 来快速把某个接口类型中定义的属性变成可选的：

```
interface PullDownRefreshConfig {
  threshold: number;
  stop: number;
}

/**
 * type PullDownRefreshOptions = {
 *   threshold?: number | undefined;
 *   stop?: number | undefined;
 * }
 */
type PullDownRefreshOptions = Partial<PullDownRefreshConfig>
```

是不是觉得 `Partial<T>` 很方便，下面让我们来看一下它是如何实现的：

```ts
/**
 * Make all properties in T optional
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

4.1.2 `Required<T>`

既然可以快速地把某个接口中定义的属性全部声明为可选，那能不能把所有的可选的属性变成必选的呢？答案是可以的，针对这个需求，我们可以使用 `Required<T>` 工具类型，具体的使用方式如下：

```ts
interface PullDownRefreshConfig {
  threshold: number;
  stop: number;
}

type PullDownRefreshOptions = Partial<PullDownRefreshConfig>

/**
 * type PullDownRefresh = {
 *   threshold: number;
 *   stop: number;
 * }
 */
type PullDownRefresh = Required<Partial<PullDownRefreshConfig>>
```

同样，我们来看一下 `Required<T>` 工具类型是如何实现的：

```ts
/**
 * Make all properties in T required
 */
type Required<T> = {
  [P in keyof T]-?: T[P];
};
```

原来在 `Required<T>` 工具类型内部，通过 `-?` 移除了可选属性中的 `?`，使得属性从可选变为必选的。

*👉 *继续阅读*：*掌握 TS 这些工具类型，让你开发事半功倍*

五、& 运算符

在 TypeScript 中交叉类型是将多个类型合并为一个类型。通过 `&` 运算符可以将现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。

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

let point: Point = {
  x: 1,
  y: 1
}
```

在上面代码中我们先定义了 `PartialPointX` 类型，接着使用 `&` 运算符创建一个新的 `Point` 类型，表示一个含有 x 和 y 坐标的点，然后定义了一个 `Point` 类型的变量并初始化。

5.1 同名基础类型属性的合并

那么现在问题来了，假设在合并多个类型的过程中，刚好出现某些类型存在相同的成员，但对应的类型又不一致，比如：

```
interface X {
  c: string;
  d: string;
}

interface Y {
  c: number;
  e: string
}

type XY = X & Y;
type YX = Y & X;

let p: XY;
let q: YX;
```

在上面的代码中，接口 X 和接口 Y 都含有一个相同的成员 c，但它们的类型不一致。对于这种情况，此时 XY 类型或 YX 类型中成员 c 的类型是不是可以是 `string` 或 `number` 类型呢？比如下面的例子：

```
p = { c: 6, d: "d", e: "e" };
```

```
q = { c: "c", d: "d", e: "e" };
```

为什么接口 X 和接口 Y 混入后，成员 c 的类型会变成 `never` 呢？这是因为混入后成员 c 的类型为 `string & number`，即成员 c 的类型既可以是 `string` 类型又可以是 `number` 类型。很明显这种类型是不存在的，所以混入后成员 c 的类型为 `never` 。

5.2 同名非基础类型属性的合并

在上面示例中，刚好接口 X 和接口 Y 中内部成员 c 的类型都是基本数据类型，那么如果是非基本数据类型的话，又会是什么情形。我们来看个具体的例子：

```
interface D { d: boolean; }
interface E { e: string; }
interface F { f: number; }

interface A { x: D; }
interface B { x: E; }
interface C { x: F; }

type ABC = A & B & C;

let abc: ABC = {
  x: {
    d: true,
    e: 'semlinker',
    f: 666
  }
};

console.log('abc:', abc);
```

由上，在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，那么是可以成功合并。

*👉 *继续阅读**：***TypeScript 交叉类型**

六、| 分隔符

在 TypeScript 中联合类型（Union Types）表示取值可以为多种类型中的一种，联合类型使用 `|` 分隔每个类型。联合类型通常与 `null` 或 `undefined` 一起使用：

```
const sayHello = (name: string | undefined) => { /* ... */ };
```

以上示例中 `name` 的类型是 `string | undefined` 意味着可以将 `string` 或 `undefined` 的值传递给 `sayHello` 函数。

```
sayHello("semlinker");
sayHello(undefined);
```

此外，对于联合类型来说，你可能会遇到以下的用法：

```
let num: 1 | 2 = 1;
type EventNames = 'click' | 'scroll' | 'mousemove';
```

示例中的 `1` 、 `2` 或 `'click'` 被称为字面量类型，用来约束取值只能是某几个值中的一个。

6.1 类型保护

当使用联合类型时，我们必须尽量把当前值的类型收窄为当前值的实际类型，而类型保护就是实现类型收窄的一种手段。

类型保护是可执行运行时检查的一种表达式，用于确保该类型在一定的范围内。换句话说，类型保护可以保证一个字符串是一个字符串，尽管它的值也可以是一个数字。类型保护与特性检测并不是完全不同，其主要思想是尝试检测属性、方法或原型，以确定如何处理值。

目前主要有四种的方式来实现类型保护：

6.1.1 in 关键字

```
interface Admin {
  name: string;
  privileges: string[];
}

interface Employee {
  name: string;
  startDate: Date;
}

type UnknownEmployee = Employee | Admin;

function printEmployeeInformation(emp: UnknownEmployee) {
  console.log("Name: " + emp.name);
  if ("privileges" in emp) {
    console.log("Privileges: " + emp.privileges);
  }
```

```
  if ("startDate" in emp) {
    console.log("Start Date: " + emp.startDate);
  }
}
```

6.1.2 typeof 关键字

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

`typeof` 类型保护只支持两种形式：`typeof v === "typename"` 和 `typeof v !== typename`，`"typename"` 必须是 `"number"`，`"string"`，`"boolean"` 或 `"symbol"`。但是 TypeScript 并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

6.1.3 instanceof 关键字

```
interface Padder {
  getPaddingString(): string;
}

class SpaceRepeatingPadder implements Padder {
  constructor(private numSpaces: number) {}
  getPaddingString() {
    return Array(this.numSpaces + 1).join(" ");
  }
}

class StringPadder implements Padder {
  constructor(private value: string) {}
  getPaddingString() {
    return this.value;
  }
}

let padder: Padder = new SpaceRepeatingPadder(6);
```

```
if (padder instanceof SpaceRepeatingPadder) {
  // padder的类型收窄为 'SpaceRepeatingPadder'
}
```

6.1.4 自定义类型保护的类型谓词（type predicate）

```
function isNumber(x: any): x is number {
  return typeof x === "number";
}

function isString(x: any): x is string {
  return typeof x === "string";
}
```

👉 **继续阅读*：*读懂 TS 中联合类型和交叉类型的含义**

七、_ 数字分隔符

TypeScript 2.7 带来了对数字分隔符的支持，正如数值分隔符 ECMAScript 提案中所概述的那样。对于一个数字字面量，你现在可以通过把一个下划线作为它们之间的分隔符来分组数字：

```
const inhabitantsOfMunich = 1_464_301;
const distanceEarthSunInKm = 149_600_000;
const fileSystemPermission = 0b111_111_000;
const bytes = 0b1111_10101011_11110000_00001101;
```

分隔符不会改变数值字面量的值，但逻辑分组使人们更容易一眼就能读懂数字。以上 TS 代码经过编译后，会生成以下 ES5 代码：

```
"use strict";
var inhabitantsOfMunich = 1464301;
var distanceEarthSunInKm = 149600000;
var fileSystemPermission = 504;
var bytes = 262926349;
```

7.1 使用限制

虽然数字分隔符看起来很简单，但在使用时还是有一些限制。比如你只能在两个数字之间添加 `_` 分隔符。以下的使用方式是非法的：

```
// Numeric separators are not allowed here.(6188)
```

```
3_.141592 // Error
3._141592 // Error

// Numeric separators are not allowed here.(6188)
1_e10 // Error
1e_10 // Error

// Cannot find name '_126301'.(2304)
_126301  // Error
// Numeric separators are not allowed here.(6188)
126301_ // Error

// Cannot find name 'b111111000'.(2304)
// An identifier or keyword cannot immediately follow a numeric literal.(1351)
0_b111111000 // Error

// Numeric separators are not allowed here.(6188)
0b_111111000 // Error
```

当然你也不能连续使用多个 `_` 分隔符，比如：

```
// Multiple consecutive numeric separators are not permitted.(6189)
123__456 // Error
```

7.2 解析分隔符

此外，需要注意的是以下用于解析数字的函数是不支持分隔符：

- `Number()`
- `parseInt()`
- `parseFloat()`

这里我们来看一下实际的例子：

```
Number('123_456')
NaN
parseInt('123_456')
123
parseFloat('123_456')
123
```

很明显对于以上的结果不是我们所期望的，所以在处理分隔符时要特别注意。当然要解决上述问题，也很简单只需要非数字的字符删掉即可。这里我们来定义一个 `removeNonDigits` 的函数：

```
const RE_NON_DIGIT = /[^0-9]/gu;

function removeNonDigits(str) {
  str = str.replace(RE_NON_DIGIT, '');
  return Number(str);
}
```

该函数通过调用字符串的 `replace` 方法来移除非数字的字符，具体的使用方式如下：

```
removeNonDigits('123_456')
123456
removeNonDigits('149,600,000')
149600000
removeNonDigits('1,407,836')
1407836
```

八、 `<Type>` 语法

8.1 TypeScript 断言

有时候你会遇到这样的情况，你会比 TypeScript 更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。类型断言好比其他语言里的类型转换，但是不进行特殊的数据检查和解构。它没有运行时的影响，只是在编译阶段起作用。

类型断言有两种形式：

8.1.1 “尖括号” 语法

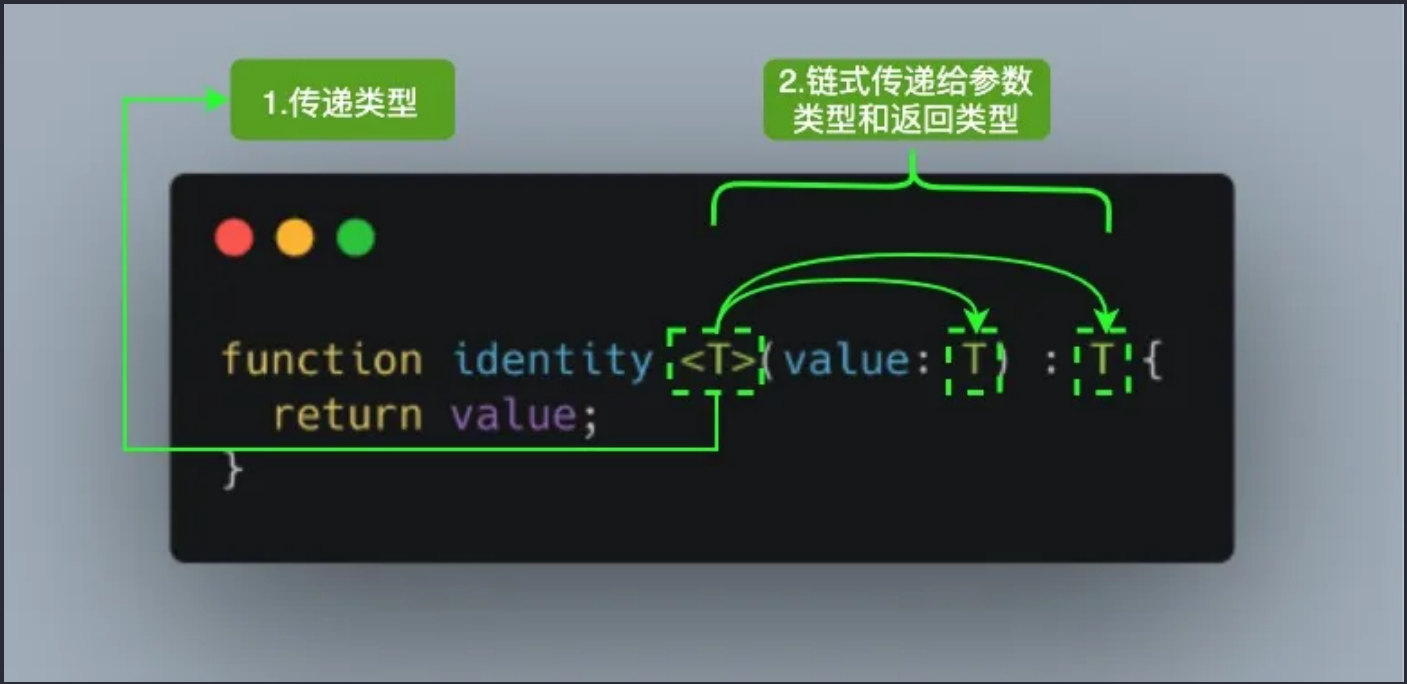
```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

8.1.2 as 语法

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

8.2 TypeScript 泛型

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。

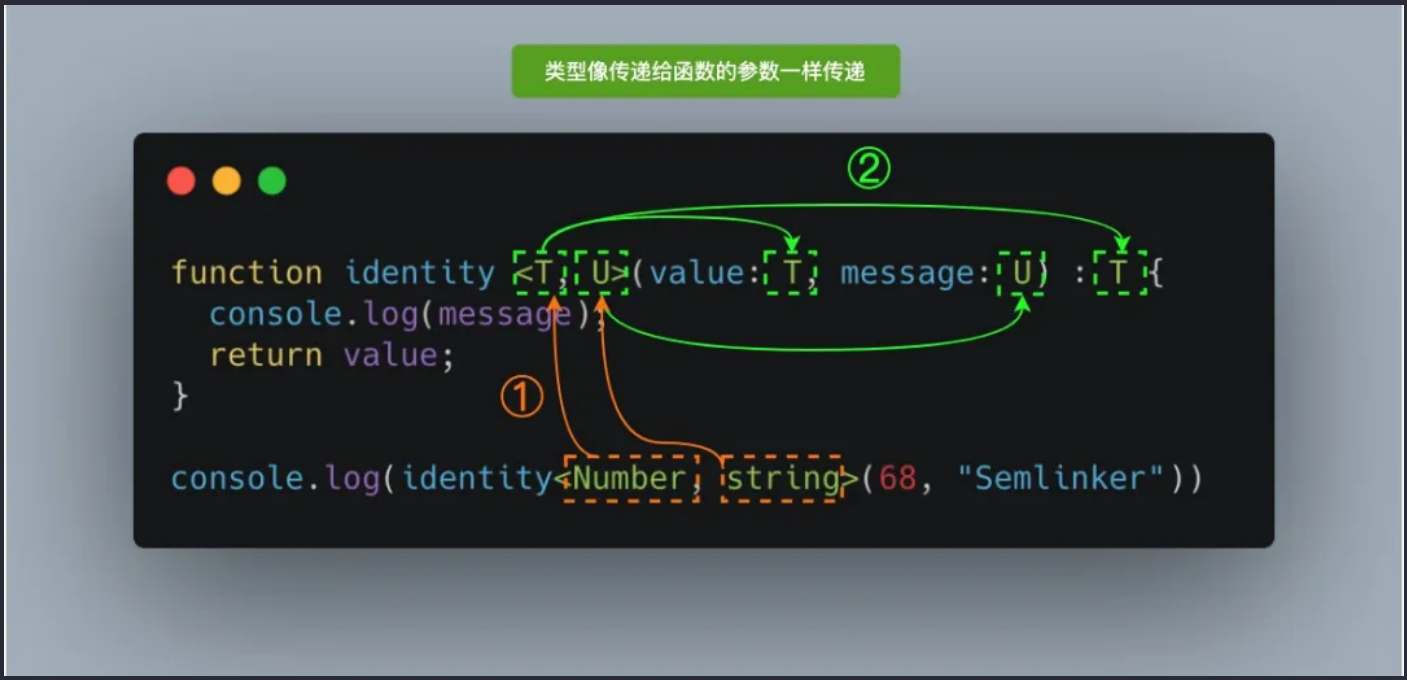
其中 `T` 代表 **Type**，在定义泛型时通常用作第一个类型变量名称。但实际上 `T` 可以用任何有效名称代替。除了 `T` 之外，以下是常见泛型变量代表的意思：

- 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。

👉 继续阅读*：*一文读懂 TypeScript 泛型及应用

九、@XXX 装饰器

9.1 装饰器语法

对于一些刚接触 TypeScript 的小伙伴来说，在第一次看到 `@Plugin({...})` 这种语法可能会觉得很惊讶。其实这是装饰器的语法，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。

```
@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，我们通过装饰器来保存 ionic-native 插件的相关元信息，而 `@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, key,
desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc =
Object.getOwnPropertyDescriptor(target, key) : desc, d;
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r
= Reflect.decorate(decorators, target, key, desc);
    else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i])
r = (c < 3 ? d(r) : c > 3 ? d(target, key, r) : d(target, key)) || r;
    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。

9.2 装饰器的分类

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。

9.2.1 类装饰器

类装饰器声明：

```
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 - 被装饰的类

看完第一眼后，是不是感觉都不好了。没事，我们马上来个例子：

```
function Greeter(target: Function): void {
  target.prototype.greet = function (): void {
    console.log("Hello Semlinker!");
  };
}

@Greeter
class Greeting {
  constructor() {
    // 内部实现
  }
}

let myGreeting = new Greeting();
myGreeting.greet(); // console output: 'Hello Semlinker!';
```

上面的例子中，我们定义了 `Greeter` 类装饰器，同时我们使用了 `@Greeter` 语法糖，来使用装饰器。

> 友情提示：读者可以直接复制上面的代码，在 TypeScript Playground 中运行查看结果。

9.2.2 属性装饰器

属性装饰器声明：

```
declare type PropertyDecorator = (target:Object,
  propertyKey: string | symbol ) => void;
```

属性装饰器顾名思义，用来装饰类的属性。它接收两个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 被装饰类的属性名

趁热打铁，马上来个例子热热身：

```
function logProperty(target: any, key: string) {
  delete target[key];

  const backingField = "_" + key;

  Object.defineProperty(target, backingField, {
    writable: true,
    enumerable: true,
    configurable: true
  });

  // property getter
  const getter = function (this: any) {
    const currVal = this[backingField];
    console.log(`Get: ${key} => ${currVal}`);
    return currVal;
  };

  // property setter
  const setter = function (this: any, newVal: any) {
    console.log(`Set: ${key} => ${newVal}`);
    this[backingField] = newVal;
  };

  // Create new property with getter and setter
  Object.defineProperty(target, key, {
    get: getter,
    set: setter,
    enumerable: true,
```

```
    configurable: true
  });
}

class Person {
  @logProperty
  public name: string;

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

const p1 = new Person("semlinker");
p1.name = "kakuqo";
```

以上代码我们定义了一个 `logProperty` 函数，来跟踪用户对属性的操作，当代码成功运行后，在控制台会输出以下结果：

```
Set: name => semlinker
Set: name => kakuqo
```

9.2.3 方法装饰器

方法装饰器声明：

```
declare type MethodDecorator = <T>(target:Object, propertyKey: string | symbol,

  descriptor: TypePropertyDescript<T>) => TypedPropertyDescriptor<T> | void;
```

方法装饰器顾名思义，用来装饰类的方法。它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- descriptor: TypePropertyDescript - 属性描述符

废话不多说，直接上例子：

```
function LogOutput(tarage: Function, key: string, descriptor: any) {
  let originalMethod = descriptor.value;
  let newMethod = function(...args: any[]): any {
    let result: any = originalMethod.apply(this, args);
```

```
      if(!this.loggedOutput) {
        this.loggedOutput = new Array<any>();
      }
      this.loggedOutput.push({
        method: key,
        parameters: args,
        output: result,
        timestamp: new Date()
      });
      return result;
    };
    descriptor.value = newMethod;
  }

  class Calculator {
    @LogOutput
    double (num: number): number {
      return num * 2;
    }
  }

  let calc = new Calculator();
  calc.double(11);
  // console ouput: [{method: "double", output: 22, ...}]
  console.log(calc.loggedOutput);
```

9.2.4 参数装饰器

参数装饰器声明：

```
  declare type ParameterDecorator = (target: Object, propertyKey: string | symbol,
    parameterIndex: number ) => void
```

参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- parameterIndex: number - 方法中参数的索引值

```
  function Log(target: Function, key: string, parameterIndex: number) {
    let functionLogged = key || target.prototype.constructor.name;
    console.log(`The parameter in position ${parameterIndex} at ${functionLogged}
  has
```

```
    been decorated`);
  }

  class Greeter {
    greeting: string;
    constructor(@Log phrase: string) {
   this.greeting = phrase;
    }
  }

  // console output: The parameter in position 0
  // at Greeter has been decorated
```

👉 **继续阅读：觉得装饰器有点难？那就进来看看呗**

十、#XXX 私有字段

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。

10.1 私有字段与 private 的区别

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)}!`);
    }
}
_name = new WeakMap();
```

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。

👉 **继续阅读：你不知道的 WeakMap**

以上提到的这些 “奇怪” 的符号，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。

十一、参考资源

- ES proposal: numeric separators
- typescriptlang.org