类型断言

TypeScript 提供了“类型断言”这样一种手段，允许开发者在代码中“断言”某个值的类型，告诉编译器此处的值是什么类型。TypeScript 一旦发现存在类型断言，就不再对该值进行类型推断，而是直接采用断言给出的类型。

类型断言有两种语法。

```
// 语法一：<类型>值
<Type>value

// 语法二：值 as 类型
value as Type
```

上面两种语法是等价的，`value` 表示值，`Type` 表示类型。早期只有语法一，后来因为 TypeScript 开始支持 React 的 JSX 语法（尖括号表示 HTML 元素），为了避免两者冲突，就引入了语法二。目前，推荐使用语法二。

```
type T = 'a'|'b'|'c';
let foo = 'a';

// 语法一
let bar:T = <T>foo; // JSX element 'T' has no corresponding closing tag.

// 语法二
let bar:T = foo as T;
```

上面示例是两种类型断言的语法，其中的语法一因为跟 JSX 语法冲突，使用时必须关闭 TypeScript 的 React 支持，否则会无法识别。由于这个原因，现在一般都使用语法二。

```
const username = document.getElementById('username'); // const username:
HTMLElement | null
(username as HTMLInputElement).value; // 正确
HTMLInputElement.value; // Property 'value' does not exist on type '{ new ():
HTMLInputElement; prototype: HTMLInputElement; }'.
```

上例中，变量 `username` 的类型是 `HTMLElement | null`，`null` 类型是没有 `value` 属性的。如果 `username` 是一个输入框，那么就可以通过类型断言，将它的类型改成 `HTMLInputElement`，就可以读取 `value` 属性。

类型断言不应滥用，因为它改变了 TypeScript 的类型检查，很可能埋下错误的隐患。

```
const data:object = {
  a: 1,
  b: 2,
```

```
  c: 3
};
data.length; // Property 'length' does not exist on type 'object'.
(data as Array<string>).length; // 正确
```

变量 data 是一个对象，没有 length 属性。但是通过类型断言，可以将它的类型断言为数组，这样使用 length 属性就能通过类型检查。但是，编译后的代码在运行时依然会报错，所以类型断言可以让错误的代码通过编译。

类型断言可以指定 unknown 类型的变量的具体类型。

```
const value:unknown = 'Hello World';
const s1:string = value; // Type 'unknown' is not assignable to type 'string'.
const s2:string = value as string; // 正确
```

类型断言也适合指定联合类型的值的具体类型。

```
const s1:number|string = 123;
const s2:number = s1 as number; // 断言 s1 类型为 number，便可以给类型为 number 的变量 s2 赋值
```

上例中，变量 s1 是联合类型，可以断言其为联合类型里面的一种具体类型，再将其赋值给变量 s2。

1. 类型断言的条件

类型断言并不意味着，可以把某个值断言为任意类型。

```
const n = 1;
const m:string = n as string; // Conversion of type 'number' to type 'string' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.
```

上例中，变量 n 是数值类型 '1'，是 number 类型的子类型，无法把它断言成字符串，TypeScript 会报错。

类型断言的使用前提是，值的实际类型与断言的类型必须满足一个条件。expr as T，expr 是实际的值，T 是类型断言，它们必须满足下面的条件：expr 是 T 的子类型，或者 T 是 expr 的子类型。

也就是说，类型断言要求实际的类型与断言的类型兼容，实际类型可以断言为一个更加宽泛的类型（父类型），也可以断言为一个更加精确的类型（子类型），但不能断言为一个完全无关的类型。

```
const n1 = 1; // const n1: 1, 数值类型 1
const m1:number = n1 as number; // 断言为父类型 number

const n2:number = 1; // const n2: number, number 类型
const m2:number = n2 as 1; // 断言为数值类型 1
```

但是，如果真的要断言成一个完全无关的类型，也是可以做到的。那就是连续进行两次类型断言，**先断言成 unknown 类型或 any 类型，然后再断言为目标类型。**因为any类型和unknown类型是所有其他类型的父类型，所以可以作为两种完全无关的类型的中介。

```
// 或者写成 <T><unknown>expr
expr as unknown as T
```

上面代码中，`expr` 连续进行了两次类型断言，第一次断言为 `unknown` 类型，第二次断言为 `T` 类型。这样的话，`expr` 就可以断言成任意类型 `T`，而不报错。

```
const n = 1;
const m:string = n as unknown as string; // 正确
```

通过两次类型断言，变量 `n` 的类型就从数值，变成了完全无关的字符串，从而赋值时不会报错。

2. as const 断言

如果没有声明变量类型，let 命令声明的变量，会被类型推断为 TypeScript 内置的基本类型之一；const 命令声明的变量，则被推断为值类型常量。

```
// 类型推断为基本类型 string
let s1 = 'JavaScript';

// 类型推断为数值类型 “JavaScript”
const s2 = 'JavaScript';
```

上例中，变量 `s1` 的类型被推断为 `string`，变量 `s2` 的类型推断为值类型 `JavaScript` 。**后者是前者的子类型，相当于 `const` 命令有更强的限定作用，可以缩小变量的类型范围。**

有些时候，let 变量会出现一些意想不到的报错，变更成 const 变量就能消除报错。

```
let s = 'JavaScript';
type Lang =
  |'JavaScript'
  |'TypeScript'
  |'Python';
function setLang(language:Lang) {
  /* ... */
}
setLang(s); // Argument of type 'string' is not assignable to parameter of type
'Lang'.
```

函数 `setLang()` 的参数 `language` 类型是 `Lang`，这是一个联合类型。传入的字符串 `s` 的类型被推断为 `string`，属于 `Lang` 的父类型。父类型不能替代子类型，导致报错。

一种解决方法就是把 let 命令改成 const 命令。

```
const s = 'JavaScript';
```

这样的话，变量 `s` 的类型就是值类型 `"JavaScript"`，它是联合类型 `Lang` 的子类型，传入函数 `setLang()` 就不会报错。

另一种解决方法是使用类型断言。TypeScript 提供了一种特殊的类型断言 `as const`，用于告诉编译器，推断类型时，可以将这个值推断为常量，即把 `let` 变量断言为 `const` 变量，从而把内置的基本类型变更为值类型。

```
let s = 'JavaScript' as const;
setLang(s);  // 正确
```

变量 `s` 虽然是用 `let` 命令声明的，但是使用了 `as const` 断言以后，就等同于是用 `const` 命令声明的，变量s的类型会被推断为*值类型* *`"JavaScript"`*。使用了 `as const` 断言以后，`let` 变量就不能再改变值了。

```
let s = 'JavaScript' as const; // let s: "JavaScript"
s = 'Python'; // Type '"Python"' is not assignable to type '"JavaScript"'.
```

`let` 命令声明的变量 `s`，使用 `as const` 断言以后，就不能改变值了，否则报错。

`as const` 断言只能用于字面量，不能用于变量。

```
let s1 = 'JavaScript';
let s2 = s1 as const; // A 'const' assertions can only be applied to references to
enum members, or string, number, boolean, array, or object literals.'const'断言只
能应用于对枚举成员或字符串、数字、布尔值、数组或对象字面值的引用。
```

`as const`也不能用于表达式。

```
let s = ('Java' + 'Script') as const; //  // A 'const' assertions can only be
applied to references to enum members, or string, number, boolean, array, or
object literals.'const'断言只能应用于对枚举成员或字符串、数字、布尔值、数组或对象字面值
的引用。
```

`as const`断言可以用于整个对象，也可以用于对象的单个属性，这时它的类型缩小效果是不一样的。

```
const v1 = {
  x: 1,
  y: 2,
}; // const v1: { x: number; y: number; }

const v2 = {
  x: 1 as const, // 对属性x缩小类型
  y: 2,
}; // const v2: { x: 1; y: number; }

// 对整个对象缩小类型
const v3 = {
  x: 1,
  y: 2,
} as const; // const v3: { readonly x: 1; readonly y: 2; }
```

`as const` 会将字面量的类型断言为不可变类型，缩小成 TypeScript 允许的最小类型。

```
const a1 = [1, 2, 3]; // const a1: number[]
let a2 = [1, 2, 3]; // let a2: number[]

const a3 = [1, 2, 3] as const; // const a3: readonly [1, 2, 3]
let a4 = [1, 2, 3] as const; // let a4: readonly [1, 2, 3]
```

数组字面量使用 `as const` 断言后，类型推断就变成了只读元组。由于 `as const` 会将数组变成只读元组，所以很适合用于函数的 `rest` 参数。

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

const numbers = [1, 2]; // const numbers: number[]
const total = add(...numbers); // A spread argument must either have a tuple type
or be passed to a rest parameter. 扩展参数必须具有元组类型，或者传递给rest形参。
```

变量 `numbers` 的类型推断为 `number[]`，导致使用扩展运算符...传入函数 `add()` 会报错，因为 `add()` 只能接受两个参数，而 `...numbers` 并不能保证参数的个数。

事实上，对于固定参数个数的函数，如果传入的参数包含扩展运算符，那么扩展运算符只能用于元组。只有当函数定义使用了 rest 参数，扩展运算符才能用于数组。

解决方法就是使用 `as const` 断言，将数组变成元组。

```
const numbers = [1, 2] as const; // const numbers: readonly [1, 2]
const total = add(...numbers);
```

上例中，使用 `as const` 断言后，变量 `numbers` 的类型会被推断为 `readonly [1, 2]`，使用扩展运算符展开后，正好符合函数 `add()` 的参数类型。

Enum 成员也可以使用 `as const` 断言。

```
enum Foo {
  X,
  Y,
}
let e1 = Foo.X; // let e1: Foo
let e2 = Foo.X as const; // let e2: Foo.X
```

上例中，如果不使用 `as const` 断言，变量 `e1` 的类型被推断为整个 `Enum` 类型；使用了 `as const` 断言以后，变量 `e2` 的类型被推断为 Enum 的某个成员，这意味着它不能变更为其他成员。

3. 非空断言

对于那些可能为空的变量（即可能等于 `undefined` 或 `null`），TypeScript 提供了非空断言，保证这些变量不会为空，写法是在变量名后面加上感叹号 `!`。

```
const root = document.getElementById('root');

// 'root' is possibly 'null'.
root.addEventListener('click', e => {
  /* ... */
});
```

如果可以确认root元素肯定会在网页中存在，这时就可以使用非空断言。

```
const root = document.getElementById('root')!;
root!.addEventListener('click', e => {
  /* ... */
});
```

`getElementById()` 方法加上后缀 `!`，或者 `root` 加上非空断言 `!` 表示这个方法肯定返回非空结果。

比较保险的做法还是手动检查一下是否为空。

```
const root = document.getElementById('root');
if (root === null) {
  throw new Error('Unable to find DOM element #root');
}
root.addEventListener('click', e => {
  /* ... */
});
```

4. 断言函数

断言函数是一种特殊函数，用于保证函数参数符合某种类型。如果函数参数达不到要求，就会抛出错误，中断程序执行；如果达到要求，就不进行任何操作，让代码按照正常流程运行。

```
function isString(value:unknown):asserts value is string {
  if (typeof value !== 'string')
    throw new Error('Not a string');
}
```

上例中，函数 `isString()` 的返回值类型写成 `asserts value is string`，其中 `asserts` 和 `is` 都是关键词，`value` 是函数的参数名，`string` 是函数参数的预期类型。它的意思是，该函数用来断言参数 `value` 的类型是 `string` ，如果达不到要求，程序就会在这里中断。

断言函数的asserts语句等同于void类型，所以如果返回除了undefined和null以外的值，都会报错。

```
function isString(value:unknown):asserts value is string {
  if (typeof value !== 'string')
    throw new Error('Not a string');
  return true; // 报错
}
```

上例中，断言函数返回了true，导致报错。

```
type AccessLevel = 'r' | 'w' | 'rw';

function allowsReadAccess(level:AccessLevel):asserts level is 'r' | 'rw' {
  if (!level.includes('r'))
    throw new Error('Read not allowed');
}
```

函数 `allowsReadAccess()` 用来断言参数 `level` 一定等于 `r` 或 `rw`。

使用断言函数的同时又想定义函数返回类型，可以使用箭头函数：

```
type AssertIsNumber = (value:unknown) => asserts value is number;

const assertIsNumber:AssertIsNumber = (value): string => {
    if (typeof value !== 'number')
        throw Error('Not a number');
    return "value is " + value;
};
```