# TypeScript 的类型断言

## 简介

对于没有类型声明的值，TypeScript 会进行类型推断，很多时候得到的结果，未必是开发者想要的。

```typescript
type T = "a" | "b" | "c";
let foo = "a";

let bar: T = foo; // 报错
```

上面示例中，最后一行报错，原因是 TypeScript 推断变量 `foo` 的类型是 `string` ，而变量 `bar` 的类型是 `'a'|'b'|'c'` ，前者是后者的父类型。父类型不能赋值给子类型，所以就报错了。

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。

这种做法的实质是，允许开发者在某个位置“绕过”编译器的类型推断，让本来通不过类型检查的代码能够通过，避免编译器报错。这样虽然削弱了 TypeScript 类型系统的严格性，但是为开发者带来了方便，毕竟开发者比编译器更了解自己的代码。

回到上面的例子，解决方法就是进行类型断言，在赋值时断言变量 `foo` 的类型。

```typescript
type T = "a" | "b" | "c";

let foo = "a";
let bar: T = foo as T; // 正确
```

上面示例中，最后一行的 `foo as T` 表示告诉编译器，变量 `foo` 的类型断言为 `T` ，所以这一行不再需要类型推断了，编译器直接把 `foo` 的类型当作 `T` ，就不会报错了。

总之，类型断言并不是真的改变一个值的类型，而是提示编译器，应该如何处理这个值。

类型断言有两种语法。

typescript

```
// 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。

typescript

```
// 语法一
let bar: T = <T>foo;

// 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。

下面看一个例子。《对象》一章提到过，对象类型有严格字面量检查，如果存在额外的属性会报错。

typescript

```
// 报错
const p: { x: number } = { x: 0, y: 0 };
```

上面示例中，等号右侧是一个对象字面量，多出了属性 `y` ，导致报错。解决方法就是使用类型断言，可以用两种不同的断言。

typescript

```
// 正确
const p0: { x: number } = { x: 0, y: 0 } as { x: number };
```

```
// 正确
const p1: { x: number } = { x: 0, y: 0 } as { x: number; y: number };
```

上面示例中，两种类型断言都是正确的。第一种断言将类型改成与等号左边一致，第二种断言使得等号右边的类型是左边类型的子类型，子类型可以赋值给父类型，同时因为存在类型断言，就没有严格字面量检查了，所以不报错。

下面是一个网页编程的实际例子。

typescript

```
const username = document.getElementById("username");

if (username) {
  (username as HTMLInputElement).value; // 正确
}
```

上面示例中，变量 `username` 的类型是 `HTMLElement | null` ，排除了 `null` 的情况以后，HTMLElement 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。

注意，上例的类型断言的圆括号是必需的，否则 `username` 会被断言成 `HTMLInputElement.value` ，从而报错。

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。

typescript

```
const data: object = {
  a: 1,
  b: 2,
  c: 3,
};

data.length; // 报错

(data as Array<string>).length; // 正确
```

上面示例中，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。

类型断言的一大用处是，指定 unknown 类型的变量的具体类型。

typescript

```typescript
const value: unknown = "Hello World";

const s1: string = value; // 报错
const s2: string = value as string; // 正确
```

上面示例中，unknown 类型的变量 `value` 不能直接赋值给其他类型的变量，但是可以将它断言为其他类型，这样就可以赋值给别的变量了。

另外，类型断言也适合指定联合类型的值的具体类型。

typescript

```typescript
const s1: number | string = "hello";
const s2: number = s1 as number;
```

上面示例中，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` 。

---

## 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。

typescript

```typescript
const n = 1;
const m: string = n as string; // 报错
```

上面示例中，变量 `n` 是数值，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。

typescript

```typescript
expr as T;
```

上面代码中， `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。

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。

typescript

```
// 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。

下面是本小节开头那个例子的改写。

typescript

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

上面示例中，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。

---

## 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。

typescript

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

// 类型推断为字符串 “JavaScript”
const s2 = "JavaScript";
```

上面示例中，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。

typescript

```
let s = "JavaScript";
```

```
type Lang = "JavaScript" | "TypeScript" | "Python";

function setLang(language: Lang) {
  /* ... */
}

setLang(s); // 报错
```

上面示例中，最后一行报错，原因是函数 `setLang()` 的参数 `language` 类型是 `Lang` ，这是一个联合类型。但是，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。

typescript

```
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。

typescript

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

上面示例中，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。

typescript

```
let s = "JavaScript" as const;
s = "Python"; // 报错
```

上面示例中，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。

注意， `as const` 断言只能用于字面量，不能用于变量。

typescript

```
let s = "JavaScript";
setLang(s as const); // 报错
```

上面示例中， `as const` 断言用于变量 `s` ，就报错了。下面的写法可以更清晰地看出这一点。

typescript

```
let s1 = "JavaScript";
let s2 = s1 as const; // 报错
```

另外， `as const` 也不能用于表达式。

typescript

```
let s = ("Java" + "Script") as const; // 报错
```

上面示例中， `as const` 用于表达式，导致报错。

`as const` 也可以写成前置的形式。

typescript

```
// 后置形式
expr as const

// 前置形式
<const>expr
```

`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。

typescript

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

const v2 = {
  x: 1 as const,
  y: 2,
}; // 类型是 { x: 1; y: number; }

const v3 = {
  x: 1,
```

```
  y: 2,
} as const; // 类型是 { readonly x: 1; readonly y: 2; }
```

上面示例中，第二种写法是对属性 `x` 缩小类型，第三种写法是对整个对象缩小类型。

总之， `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。

下面是数组的例子。

typescript

```
// a1 的类型推断为 number[]
const a1 = [1, 2, 3];

// a2 的类型推断为 readonly [1, 2, 3]
const a2 = [1, 2, 3] as const;
```

上面示例中，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。

typescript

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

const nums = [1, 2];
const total = add(...nums); // 报错
```

上面示例中，变量 `nums` 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 `...nums` 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。

typescript

```
const nums = [1, 2] as const;
const total = add(...nums); // 正确
```

上面示例中，使用 `as const` 断言后，变量 `nums` 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。

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

上面示例中，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。

---

## 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 `!` 。

```typescript
function f(x?: number | null) {
  validateNumber(x); // 自定义函数，确保 x 是数值
  console.log(x!.toFixed());
}

function validateNumber(e?: number | null) {
  if (typeof e !== "number") throw new Error("Not a number");
}
```

上面示例中，函数 `f()` 的参数 `x` 的类型是 `number|null` ，即可能为空。如果为空，就不存在 `x.toFixed()` 方法，这样写会报错。但是，开发者可以确认，经过 `validateNumber()` 的前置检验，变量 `x` 肯定不会为空，这时就可以使用非空断言，为函数体内部的变量 `x` 加上后缀 `!` ，`x!.toFixed()` 编译就不会报错了。

非空断言在实际编程中很有用，有时可以省去一些额外的判断。

```typescript
const root = document.getElementById("root");
```

```
// 报错
root.addEventListener("click", (e) => {
  /* ... */
});
```

上面示例中， `getElementById()` 有可能返回空值 `null` ，即变量 `root` 可能为空，这时对它调用 `addEventListener()` 方法就会报错，通不过编译。但是，开发者如果可以确认 `root` 元素肯定会在网页中存在，这时就可以使用非空断言。

typescript

```
const root = document.getElementById("root")!;
```

上面示例中， `getElementById()` 方法加上后缀 `!` ，表示这个方法肯定返回非空结果。

不过，非空断言会造成安全隐患，只有在确定一个表达式的值不为空时才能使用。比较保险的做法还是手动检查一下是否为空。

typescript

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

if (root === null) {
  throw new Error("Unable to find DOM element #root");
}

root.addEventListener("click", (e) => {
  /* ... */
});
```

上面示例中，如果 `root` 为空会抛错，比非空断言更保险一点。

非空断言还可以用于赋值断言。TypeScript 有一个编译设置，要求类的属性必须初始化（即有初始值），如果不对属性赋值就会报错。

typescript

```
class Point {
  x: number; // 报错
  y: number; // 报错

  constructor(x: number, y: number) {
    // ...
```

```
  }
}
```

上面示例中，属性 `x` 和 `y` 会报错，因为 TypeScript 认为它们没有初始化。

这时就可以使用非空断言，表示这两个属性肯定会有值，这样就不会报错了。

typescript

```typescript
class Point {
  x!: number; // 正确
  y!: number; // 正确

  constructor(x: number, y: number) {
    // ...
  }
}
```

另外，非空断言只有在打开编译选项 `strictNullChecks` 时才有意义。如果不打开这个选项，编译器就不会检查某个变量是否可能为 `undefined` 或 `null` 。

---

## 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。

typescript

```typescript
function isString(value) {
  if (typeof value !== "string") throw new Error("Not a string");
}
```

上面示例中，函数 `isString()` 就是一个断言函数，用来保证参数 `value` 是一个字符串。

下面是它的用法。

typescript

```typescript
const aValue: string | number = "Hello";
isString(aValue);
```

上面示例中，变量 `aValue` 可能是字符串，也可能是数组。但是，通过调用 `isString()` ，后面的代码就可以确定，变量 `aValue` 一定是字符串。

断言函数的类型可以写成下面这样。

typescript

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

上面代码中，函数参数 `value` 的类型是 `unknown` ，返回值类型是 `void` ，即没有返回值。可以看到，单单从这样的类型声明，很难看出 `isString()` 是一个断言函数。

为了更清晰地表达断言函数，TypeScript 3.7 引入了新的类型写法。

typescript

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

上面示例中，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。

使用了断言函数的新写法以后，TypeScript 就会自动识别，只要执行了该函数，对应的变量都为断言的类型。

注意，函数返回值的断言写法，只是用来更清晰地表达函数意图，真正的检查是需要开发者自己部署的。而且，如果内部的检查与断言不一致，TypeScript 也不会报错。

typescript

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

上面示例中，函数的断言是参数 `value` 类型为字符串，但是实际上，内部检查的却是它是否为数值，如果不是就抛错。这段代码能够正常通过编译，表示 TypeScript 并不会检查断言与实际的类型检查是否一致。

另外，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。

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

上面示例中，断言函数返回了 `true` ，导致报错。

下面是另一个例子。

```typescript
type AccessLevel = "r" | "w" | "rw";

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

上面示例中，函数 `allowsReadAccess()` 用来断言参数 `level` 一定等于 `r` 或 `rw` 。

如果要断言参数非空，可以使用工具类型 `NonNullable<T>` 。

```typescript
function assertIsDefined<T>(value: T): asserts value is NonNullable<T> {
  if (value === undefined || value === null) {
    throw new Error(`${value} is not defined`);
  }
}
```

上面示例中，工具类型 `NonNullable<T>` 对应类型 `T` 去除空类型后的剩余类型。

如果要将断言函数用于函数表达式，可以采用下面的写法。

```typescript
// 写法一
const assertIsNumber = (value: unknown): asserts value is number => {
  if (typeof value !== "number") throw Error("Not a number");
};

// 写法二
type AssertIsNumber = (value: unknown) => asserts value is number;

const assertIsNumber: AssertIsNumber = (value) => {
```

```
  if (typeof value !== "number") throw Error("Not a number");
};
```

注意，断言函数与类型保护函数（type guard）是两种不同的函数。它们的区别是，断言函数不返回值，而类型保护函数总是返回一个布尔值。

typescript

```
function isString(value: unknown): value is string {
  return typeof value === "string";
}
```

上面示例就是一个类型保护函数 `isString()` ，作用是检查参数 `value` 是否为字符串。如果是的，返回 `true` ，否则返回 `false` 。该函数的返回值类型是 `value is string` ，其中的 `is` 是一个类型运算符，如果左侧的值符合右侧的类型，则返回 `true` ，否则返回 `false` 。

如果要断言某个参数保证为真（即不等于 `false` 、 `undefined` 和 `null` ），TypeScript 提供了断言函数的一种简写形式。

typescript

```
function assert(x: unknown): asserts x {
  // ...
}
```

上面示例中，函数 `assert()` 的断言部分， `asserts x` 省略了谓语和宾语，表示参数 `x` 保证为真（ `true` ）。

同样的，参数为真的实际检查需要开发者自己实现。

typescript

```
function assert(x: unknown): asserts x {
  if (!x) {
    throw new Error(`${x} should be a truthy value.`);
  }
}
```

这种断言函数的简写形式，通常用来检查某个操作是否成功。

typescript

```
type Person = {
  name: string;
  email?: string;
};
```

```ts
function loadPerson(): Person | null {
  return null;
}

let person = loadPerson();

function assert(condition: unknown, message: string): asserts condition {
  if (!condition) throw new Error(message);
}

// Error: Person is not defined
assert(person, "Person is not defined");
console.log(person.name);
```

上面示例中，只有 `loadPerson()` 返回结果为真（即操作成功）， `assert()` 才不会报错。

---

## 参考链接

- Const Assertions in Literal Expressions in TypeScript, Marius Schulz
- Assertion Functions in TypeScript, Marius Schulz
- Assertion functions in TypeScript, Matteo Di Pirro



最后更新: 2023/8/13 15:25

---


Previous page
Enum 类型

Next page
模块