1. 类型断言简介

就是对于没有类型声明的值，有时候ts类型推断时，推断的结果并不正确，所以ts提供了一种**类型断言**的方法，告诉编译器此处的值是什么类型的，而ts一旦发现存在类型断言，就不再对该值进行类型推断直接给出断言的类型。

类型断言的实质：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验的代码能够通过，避免编译器报错，这样虽然削弱了ts类型系统的严格性，但也给开发者带来了方便。

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。

写法：

```
// 语法一：<类型>值
<Type>value

// 语法二：值 as 类型
value as Type
```

应用

- 对象类型会严格检查字面量，如果存在额外的属性会报错，此时可以通过类型断言解决

写法一：断言将右边类型改为于左边类型一致

写法二：右边类型变为左边类型的子类型

```
// 报错
const p:{ x: number } = { x: 0, y: 0 };
// 正确 写法一
const p0:{ x: number } =
  { x: 0, y: 0 } as { x: number };

// 正确 写法二
const p1:{ x: number } =
  { x: 0, y: 0 } as { x: number; y: number };
```

- 有时候不能滥用，否则会在运行时报错

比如对象本身没有length属性，而使用类型断言强行设置，虽然能通过编译时的检测但等到运行时也会报错。

- 指定unknown类型的变量具体类型，因为unknown类型的变量不能赋值给其他类型，但是通过类型断言就可以。
- 也可以指定联合类型的值的具体类型,本来一个变量是联合类型的值，但是通过类型断言可以指定其具体的类型。

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

2. 类型断言的条件

类型断言并不能将某个值断言为任意类型，而是必须某种条件，就是**指定类型是实际值的子类型或者实际值是指定类型的子类型**才可以。

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子类型，但不能断言为一个完全无关的类型。

如果真要断言成一个无关类型的，可以进行两次类型断言，先断言成any或unknown类型，再断言成目标类型，因为any、unknown类型是所有其他类型的父类型，可以作为两种完全无关类型的中介。

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

3. as const断言

如果没有声明变量的类型，对于let命名的变量，会被ts推断为内置的基本类型，而const声明的变量，会被推断为值类型。

`as const`本质：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。

问题

有时候let声明的变量，可能会报错，可以使用const解决。

比如函数传参的时候传的是let声明的字符串，而参数类型是一个联合类型，这时候就会报错，因为字符串类型是联合类型的父类。

解决方案：

- 把let命令改为const命令，变成值类型，这样就不会报错。
- 使用类型断言`as const`,用来告诉编译器，推断类型时，把这个类型推断为常量，即把let变量断言为const变量，也就从一个字符串类型变成了值类型。此时let已经变为const，所以说此时变量就不能再该值了。

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

as const使用时的注意点

- 只能用在字面量，不能用在变量，否则会报错
- 不能用于表达式

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

用于对象

可以用于整个对象，也可以用于对象的单个属性，它的类型缩小效果是不一样的。

```
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; }
```

对于数组使用as const 断言后，类型推断为只读元组。此时很适合用于函数的rest参数。

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

Enum成员

Enum成员也可以使用as const断言，使用后就不能进行变更。

4. 非空断言

就是对于可能为空的变量(值等于undefined或努力),ts提供了非空断言，当ts推断时，给推断说该变量不为空。写法直接在变量名后面加上感叹号!。

对于过多使用非空断言会造成安全隐患，一定要确保某些变量一定不为空才可以。

非空断言可以用于赋值断言，因为ts有个编译选项，要求类的属性必须初始化，如果不初始化会报错，此时就可以使用非空断言，表示这两个属性肯定会有值，就不会报错了。

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

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

5. 断言函数

断言函数是一种特殊函数，用来保证函数参数符合某种类型。如果函数参数达不到要求就抛出错误，中断程序执行，要是达到要求，就正常执行代码。

ts3.7之前的断言函数

用来保证参数value是一个字符串 ，否则就抛出错误，中断执行。

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

缺点：参数类型是unknown，返回值类型是void，因为从这样声明类型，不是很清晰的表达断言函数

ts3.7引入了新类型写法

返回值类型写成`asserts value is string`，其中asserts和is是关键字，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，如果达不到要求就报错。

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

注意点：函数返回值的断言写法，只是为了表达函数意图，真正的校验需要开发者自己部署。而且如果内部坚持与断言不一致，ts不会报错

函数的断言是参数value类型为字符串，但内部检查为数值，也不会抛错误。

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

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

如果要断言参数为非，可以使用工具类型`NoNullable<T>`

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

用在函数表达式

```
// 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');
};
```

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。

如果要断言某个参数保证为真（即不等于`false`、`undefined`和`null`），可以使用一种简写形式。直接使用`asserts x`。这种简写形式，通常来检查某个操作是否成功。

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