

# 类型系统

TypeScript 包含 number、string、boolean、bigint、symbol、object、undefined、null 八种类型。

## 1. 基本类型

### 1.1. bigint 类型

bigint 类型包含所有的大整数。

```
const x:bigint = 123n;  
const y:bigint = 0xfffffn;
```

bigint 与 number 类型不兼容。

```
const x:bigint = 123; // Type 'number' is not assignable to type 'bigint'.  
const y:bigint = 3.14; // Type 'number' is not assignable to type 'bigint'.
```

### 1.2. object 类型

object 类型包含了所有对象、数组和函数。

```
const x:object = { foo: 123 };  
const y:object = [1, 2, 3];  
const z:object = (n:number) => n + 1;
```

### 1.3. undefined 类型, null 类型

undefined 和 null 是两种独立类型，它们各自都只有一个值。

**undefined 类型只包含一个值 undefined，设置成其他值都会报错。** 表示未定义（即还未给出定义，以后可能会有定义）。

```
let x:undefined = undefined; // 正确  
  
let x:undefined = '1'; // Type '"1"' is not assignable to type 'undefined'.  
let x:undefined = 1; // Type '1' is not assignable to type 'undefined'.  
let x:undefined = true; // Type 'true' is not assignable to type 'undefined'.  
let x:null = Symbol(); // Type 'symbol' is not assignable to type 'undefined'.
```

上例中，变量 x 就属于 undefined 类型。两个 undefined 里面，第一个是类型，第二个是值。

**null** 类型也只包含一个值 **null**，设置成其他值都会报错。表示为空（即此处没有值）。

```
const x:null = null; // 正确

let x:null = 1; // Type '1' is not assignable to type 'null'.
let x:null = '1'; // Type '"1"' is not assignable to type 'null'.
let x:null = true; // Type 'true' is not assignable to type 'null'.
let x:null = Symbol(); // Type 'symbol' is not assignable to type 'null'.
```

上例中，变量 **x** 就属于 **null** 类型。两个 **null** 里面，第一个是类型，第二个是值。

如果没有声明类型的变量，被赋值为 **undefined** 或 **null**，它们的类型会被推断为 **any**。

```
let a = undefined; // any
const b = undefined; // any

a = ''; // 可以被赋值为其他类型的值，证明 a 是 any 类型。b 不能被赋值为其他类型，因为 b 是常量。

let c = null; // any
const d = null; // any

c = true; // 可以被赋值为其他类型的值，证明 c 是 any 类型。d 不能被赋值为其他类型，因为 d 是常量。
```

## 2. 包装对象类型

JavaScript 的 8 种类型之中，**undefined** 和 **null** 其实是两个特殊值，**object** 属于复合类型，剩下的五种属于原始类型（primitive value），代表最基本的、不可再分的值。

- boolean
- string
- number
- bigint
- symbol

上面这五种原始类型的值，都有对应的包装对象（wrapper object）。所谓“包装对象”，指的是这些值在需要时，会自动产生的对象。

```
'hello'.charAt(1) // 'e'
```

上例中，字符串 **hello** 执行了 **charAt()** 方法。但是，在 JavaScript 语言中，只有对象才有方法，原始类型的值本身没有方法。这行代码之所以可以运行，**就是在调用方法时，字符串会自动转为包装对象**，**charAt()** 方法其实是定义在包装对象上。这样的设计大大方便了字符串处理，省去了将原始类型的值手动转成对象实例。

五种包装对象之中，symbol 类型和 bigint 类型无法直接获取它们的包装对象（即 `Symbol()` 和 `BigInt()` 不能作为构造函数使用），但是剩下三种可以。

- `Boolean()`
- `String()`
- `Number()`

这三个构造函数，执行后可以直接获取某个原始类型值的包装对象。

```
const s = new String('hello');
typeof s // 'object'
s.charAt(1) // 'e'

const s1 = String(1);
s1; // "1"
typeof s1; // "string"
```

上例中，`s` 就是字符串 `hello` 的包装对象，`typeof` 运算符返回 `object`，不是 `string`，但是本质上它还是字符串，可以使用所有的字符串方法。

`String()` 只有当作构造函数使用时（即带有 `new` 命令调用），才会返回包装对象。如果当作普通函数使用（不带有 `new` 命令），返回就是一个普通字符串。其他两个构造函数 `Number()` 和 `Boolean()` 也是如此。

```
let n0 = new Number('1');
n0; // 1
typeof n0; // "number"

let n1 = Number('1');
n1; // 1
typeof n1; // "number"

let b1 = Number(1);
b1; // true
typeof b1; // "boolean"

let b2 = Number(0);
b2; // false
typeof b2; // "boolean"
```

## 2.1. 包装对象类型和字面量类型

由于包装对象的存在，导致每一个原始类型的值都有包装对象和字面量两种情况。

```
'hello' // 字面量
new String('hello') // 包装对象
```

为了区分这两种情况，TypeScript 对五种原始类型分别提供了大写和小写两种类型。

- Boolean 和 boolean
- String 和 string
- Number 和 number
- BigInt 和 bigint
- Symbol 和 symbol

其中，大写类型同时包含包装对象和字面量两种情况，小写类型只包含字面量，不包含包装对象。

```
const s1:String = 'hello'; // 正确
const s2:String = new String('hello'); // 正确

const s3:string = 'hello'; // 正确
const s4:string = new String('hello'); // Type 'String' is not assignable to type 'string'. 'string' is a primitive, but 'String' is a wrapper object. Prefer using 'string' when possible.
```

上例中，`String` 类型可以赋值为字符串的字面量，也可以赋值为包装对象。但是，`string` 类型只能赋值为字面量，赋值为包装对象就会报错。

**建议只使用小写类型，不使用大写类型。** 因为绝大部分使用原始类型的场合，都是使用字面量，不使用包装对象。而且，TypeScript 把很多内置方法的参数，定义成小写类型，使用大写类型会报错。

```
const n1:number = 1;
const n2:Number = 1;

Math.abs(n1) // 1
Math.abs(n2) // Argument of type 'Number' is not assignable to parameter of type 'number'. 'number' is a primitive, but 'Number' is a wrapper object. Prefer using 'number' when possible.
```

上例中，`Math.abs()` 方法的参数类型被定义成小写的 `number`，传入大写的 `Number` 类型就会报错。

## 3. Object 类型与 object 类型

TypeScript 的对象类型也有大写 `Object` 和小写 `object` 两种。

### 3.1. Object 类型

大写的 `Object` 类型代表 JavaScript 语言里面的广义对象。所有可以转成对象的值，都是 `Object` 类型，这囊括了几乎所有的值。**原始类型值、对象、数组、函数都是合法的 `Object` 类型。**

```
let obj:Object;

obj = true;
obj = 'hi';
```

```
obj = 1;
obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;
```

除了 `undefined` 和 `null` 这两个值不能转为对象，其他任何值都可以赋值给 `Object` 类型。

```
let obj:Object;

obj = undefined; // Type 'undefined' is not assignable to type 'Object'.
obj = null; // Type 'null' is not assignable to type 'Object'.
```

空对象 `{}` 是 `Object` 类型的简写形式，所以使用 `Object` 时常常用空对象代替。

```
let obj:{};

obj = true;
obj = 'hi';
obj = 1;
obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;

obj = undefined; // Type 'undefined' is not assignable to type 'Object'.
obj = null; // Type 'null' is not assignable to type 'Object'.
```

### 3.2. object 类型

小写的 `object` 类型代表 JavaScript 里面的狭义对象，即可以用字面量表示的对象，只包含对象、数组和函数，不包括原始类型的值。

```
let obj:object;

obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;
obj = true; // Type 'boolean' is not assignable to type 'object'.
obj = 'hi'; // Type 'string' is not assignable to type 'object'.
obj = 1; // Type 'number' is not assignable to type 'object'.
```

大多数时候，我们使用对象类型，只希望包含真正的对象，不希望包含原始类型。所以，**建议总是使用小写类型 `object`，不使用大写类型 `Object`。**

无论是大写的 `Object` 类型，还是小写的 `object` 类型，都只包含 JavaScript 内置对象原生的属性和方法，用户自定义的属性和方法都不存在于这两个类型之中。

```
const o1:Object = { foo: 0 };
const o2:object = { foo: 0 };

o1.toString() // 正确
o1.foo // Property 'foo' does not exist on type 'Object'.

o2.toString() // 正确
o2.foo // Property 'foo' does not exist on type 'Object'.
```

对象类型需要明确定义属性：

```
const o1:{foo: number} = { foo: 0 };
const o2:{foo: number} = { foo: 0 };

o1.toString() // 正确
o1.foo // 正确

o2.toString() // 正确
o2.foo // 正确
```

## 4. 值类型

TypeScript 规定，单个值也是一种类型，称为“值类型”。

```
var c = 'world'; // var c: string
let a = 'hello'; // let a: string
const b = 'hello'; // const b: "hello"
```

上面例子中，没有显示定义类型，TypeScript 将使用类型推断。使用 `var` 和 `let` 定义的变量，类型推断为 `string`，当使用 `const` 定义的变量，类型推断为值类型 `hello`。

```
let x:'hello';

x = 'hello'; // 正确
x = 'world'; // Type '"hello2"' is not assignable to type '"hello"'.
```

上例中，变量 `x` 的类型是字符串 `hello`，导致它只能赋值为这个字符串，赋值为其他字符串就会报错。

TypeScript 推断类型时，遇到 `const` 命令声明的变量，如果代码里面没有注明类型，就会推断该变量是值类型。

```
// x 的类型是 "https"
const x = 'https';
```

```
// y 的类型是 string
const y:string = 'https';
```

上例中，变量x是const命令声明的，TypeScript 就会推断它的类型是值 `https`，而不是 `string` 类型。这样推断是合理的，因为 `const` 命令声明的变量，一旦声明就不能改变，相当于常量。值类型就意味着不能赋为其他值。

**const命令声明的变量，如果赋值为对象，并不会推断为值类型。**

```
// x 的类型是 { foo: number }
const x = { foo: 1 };
```

上例中，变量x没有被推断为值类型，而是推断属性 `foo` 的类型是 `number`。这是因为 JavaScript 里面，`const` 变量赋值为对象时，属性值是可以改变的。

值类型可能会出现一些很奇怪的报错。

```
const x:5 = 4 + 1; // Type 'number' is not assignable to type '5'.
```

上例中，等号左侧的类型是数值 `5`，等号右侧 `4 + 1` 的类型为 `number`。由于 `5` 是 `number` 的子类型，父类型不能赋值给子类型。反过来是可以的，子类型可以赋值给父类型。

```
let x:5 = 5; // 值类型 '5'
let y:number = 4 + 1; // number 类型

x = y; // Type 'number' is not assignable to type '5'.
y = x; // 正确
```

如果一定要让子类型可以赋值为父类型的值，就要用到类型断言。

```
const x:5 = (4 + 1) as 5; // 正确
```

上例中，在 `4 + 1` 后面加上 `as 5`，实现了类型断言，告诉编译器可以把 `4 + 1` 的类型视为值类型 `5`，这样就不会报错了。

## 5. 联合类型

联合类型 (union types) 指的是多个类型组成的一个新类型，使用符号 `|` 表示。联合类型 `A|B` 表示，任何一个类型只要属于 A或B，就属于联合类型 `A|B`。

```
let x:string|number;
```

```
x = 123; // 正确
x = 'abc'; // 正确
```

联合类型可以与值类型相结合，表示一个变量的值有若干种可能。

```
let setting:true|false;
let gender:'male' | 'female';
let rainbowColor:'赤' | '橙' | '黄' | '绿' | '青' | '蓝' | '紫';

rainbowColor = '1'; // Type '"1"' is not assignable to type '"赤" | "橙" | "黄" | "绿" | "青" | "蓝" | "紫"'.
```

上面的示例都是由值类型组成的联合类型，非常清晰地表达了变量的取值范围。其中，`true|false` 其实就是布尔类型 `boolean`。

前面提到，打开编译选项`strictNullChecks`后，其他类型的变量不能赋值为`undefined`或`null`。这时，如果某个变量确实可能包含空值，就可以采用联合类型的写法。

```
let name1:string|null;

name1 = 'John';
name1 = null;
```

联合类型的第一个成员前面，也可以加上竖杠`|`，这样便于多行书写。

```
let x:
  | 'one'
  | 'two'
  | 'three'
  | 'four';
```

如果一个变量有多种类型，读取该变量时，往往需要进行“类型缩小”（type narrowing），区分该值到底属于哪一种类型，然后再进一步处理。

```
function printId(
  id:number|string
) {
  console.log(id.toUpperCase()); // Property 'toUpperCase' does not exist on
  type 'number'.
}
```

`toUpperCase()` 方法只存在于字符串，不存在于数值。解决方法就是对参数 `id` 做一下类型缩小，确定它的类型以后再进行处理。



```
function printId(  
  id:number|string  
) {  
  if (typeof id === 'string') {  
    console.log(id.toUpperCase());  
  } else {  
    console.log(id);  
  }  
}
```

**“类型缩小”是 TypeScript 处理联合类型的标准方法，凡是遇到可能为多种类型的场合，都需要先缩小类型，再进行处理。** 实际上，联合类型本身可以看成是一种“类型放大”（type widening），处理时就需要“类型缩小”（type narrowing）。

下面是“类型缩小”的另一个例子。

```
function getPort(  
  scheme: 'http' | 'https'  
) {  
  switch (scheme) {  
    case 'http':  
      return 80;  
    case 'https':  
      return 443;  
  }  
}
```

上例中，函数体内部对参数变量 `scheme` 进行类型缩小，根据不同的值类型，返回不同的结果。

## 6. 交叉类型

交叉类型（intersection types）指的多个类型组成的一个新类型，使用符号 `&` 表示。交叉类型 `A&B` 表示，任何一个类型必须同时属于A和B，才属于交叉类型 `A&B`，即交叉类型同时满足A和B的特征。交叉类型的主要用途是表示对象的合成。

```
let obj:  
  { foo: string } &  
  { bar: string };  
  
obj = {  
  foo: 'hello',  
  bar: 'world'  
};
```

变量 `obj` 同时具有属性 `foo` 和属性 `bar`。

交叉类型常常用来为对象类型添加新属性。下面为类型 B 增加了属性 bar。

```
type A = { foo: number };
type B = A & { bar: number };
const bbb:B = {foo: 1, bar: 2};
```

## 7. type 命令

type命令用来定义一个类型的别名。

```
type Age = number;
let age:Age = 55;
```

type 命令为 number 类型定义了一个别名 Age。这样就能像使用 number 一样，使用 Age 作为类型。

别名可以让类型的名字变得更有意义，也能增加代码的可读性，还可以使复杂类型用起来更方便，便于以后修改变量的类型。别名不允许重名。

```
type Color = 'red';
type Color = 'blue'; // Duplicate identifier 'Color'.
```

别名的作用域是块级作用域。这意味着，代码块内部定义的别名，影响不到外部。

```
type Color = 'red';

if (Math.random() < 0.5) {
  type Color = 'blue'; // if代码块内部的类型别名Color，跟外部的Color是不一样的。
}
```

别名支持使用表达式，也可以在定义一个别名时，使用另一个别名，即别名允许嵌套。

```
type World = "world";
type Greeting = `hello ${World}`;
```

上例中，别名 Greeting 使用了模板字符串，读取另一个别名 World。

type 命令属于类型相关的代码，编译成 JavaScript 的时候，会被全部删除。

## 8. typeof 运算符

JavaScript 语言中，typeof 运算符是一个一元运算符，返回一个字符串，代表操作数的类型。这时 typeof 的操作数是一个值。

```
typeof 'foo'; // 'string'
```

JavaScript 里面，typeof运算符只可能返回八种结果，而且都是字符串。

```
typeof undefined; // "undefined"
typeof true; // "boolean"
typeof 1337; // "number"
typeof "foo"; // "string"
typeof {}; // "object"
typeof parseInt; // "function"
typeof Symbol(); // "symbol"
typeof 127n // "bigint"
```

TypeScript 将 `typeof` 运算符移植到了类型运算，它的操作数依然是一个值，但是返回的不是字符串，而是该值的 TypeScript 类型。

```
const a = { x: 0 };
type T0 = typeof a; // { x: number }
type T1 = typeof a.x; // number
```

`typeof a` 表示返回变量 `a` 的 TypeScript 类型 (`{ x: number }`)。同理，`typeof a.x` 返回的是属性 `x` 的类型 (`number`)。

这种用法的 `typeof` 返回的是 TypeScript 类型，所以只能用在类型运算之中（即跟类型相关的代码之中），不能用在值运算。

也就是说，同一段代码可能存在两种 `typeof` 运算符，一种用在值相关的 JavaScript 代码部分，另一种用在类型相关的 TypeScript 代码部分。

```
let a = 1;
let b:typeof a;

if (typeof a === 'number') {
  b = a;
}
```

上例中，用到了两个`typeof`，第一个是类型运算，第二个是值运算。

JavaScript 的 `typeof` 遵守 JavaScript 规则，TypeScript 的 `typeof` 遵守 TypeScript 规则。它们的一个重要区别在于，编译后，前者会保留，后者会被全部删除。

`typeof` 的参数只能是标识符，不能是需要运算的表达式。

```
type T = typeof Date(); // Expression expected.  
  
type T = typeof Date;
```

`typeof` 的参数不能是一个值的运算式，而 `Date()` 需要运算才知道结果。

另外，`typeof` 命令的参数不能是类型。

```
type Age = number;  
type MyAge = typeof Age; // 'Age' only refers to a type, but is being used as a  
value here.  
//Exported type alias 'MyAge' has or is using private name 'Age'.
```

`typeof` 是一个很重要的 TypeScript 运算符，有些场合不知道某个变量 `foo` 的类型，这时使用 `typeof foo` 就可以获得它的类型。

## 9. 块级类型声明

TypeScript 支持块级类型声明，即类型可以声明在代码块（用大括号表示）里面，并且只在当前代码块有效。

```
if (true) {  
  type T = number;  
  let v:T = 5;  
} else {  
  type T = string;  
  let v:T = 'hello';  
}
```

上例中，存在两个代码块，其中分别有一个类型 `T` 的声明。这两个声明都只在自己的代码块内部有效，在代码块外部无效。

## 10. 类型的兼容

TypeScript 的类型存在兼容关系，某些类型可以兼容其他类型。

```
type T = number|string;  
  
let a:number = 1;  
let b:T = a;
```

变量 `a` 赋值给变量 `b` 并不会报错，`b` 的类型兼容 `a` 的类型。

如果类型 `A` 的值可以赋值给类型 `B`，那么类型 `A` 就称为类型 `B` 的子类型（subtype）。在上例中，类型 `number` 就是类型 `number|string` 的子类型。

TypeScript 的一个规则是，凡是可以使用父类型的地方，都可以使用子类型，但是反过来不行。

```
let a:'hi' = 'hi';  
let b:string = 'hello';  
  
b = a; // 正确  
a = b; // Type 'string' is not assignable to type '"hi"'. 类型 'string' 不能赋值给  
类型 'hi'
```

`hi` 类型是 `string` 类型的子类型，`string` 是 `hi` 的父类型。所以，变量 `a` 可以赋值给变量 `b`，但是反过来就会报错。

之所以有这样的规则，是因为子类型继承了父类型的所有特征，所以可以用在父类型的场合。但是，子类型还可能有一些父类型没有的特征，所以父类型不能用在子类型的场合。