TypeScript 的类型系统

本章是 TypeScript 类型系统的总体介绍。

TypeScript 继承了 JavaScript 的类型，在这个基础上，定义了一套自己的类型系统。

基本类型

概述

JavaScript 语言（注意，不是 TypeScript）将值分成 8 种类型。

- boolean
- string
- number
- bigint
- symbol
- object
- undefined
- null

TypeScript 继承了 JavaScript 的类型设计，以上 8 种类型可以看作 TypeScript 的基本类型。

注意，上面所有类型的名称都是小写字母，首字母大写的 `Number` 、 `String` 、 `Boolean` 等在 JavaScript 语言中都是内置对象，而不是类型名称。

另外，undefined 和 null 既可以作为值，也可以作为类型，取决于在哪里使用它们。

这 8 种基本类型是 TypeScript 类型系统的基础，复杂类型由它们组合而成。

以下是它们的简单介绍。

boolean 类型

`boolean` 类型只包含 `true` 和 `false` 两个布尔值。

typescript

```typescript
const x: boolean = true;
const y: boolean = false;
```

上面示例中，变量 `x` 和 `y` 就属于 boolean 类型。

string 类型

`string` 类型包含所有字符串。

typescript

```typescript
const x: string = "hello";
const y: string = `${x} world`;
```

上面示例中，普通字符串和模板字符串都属于 string 类型。

number 类型

`number` 类型包含所有整数和浮点数。

typescript

```typescript
const x: number = 123;
const y: number = 3.14;
const z: number = 0xffff;
```

上面示例中，整数、浮点数和非十进制数都属于 number 类型。

bigint 类型

bigint 类型包含所有的大整数。

typescript

```typescript
const x: bigint = 123n;
const y: bigint = 0xffffn;
```

上面示例中，变量 `x` 和 `y` 就属于 bigint 类型。

bigint 与 number 类型不兼容。

```typescript
const x: bigint = 123; // 报错
const y: bigint = 3.14; // 报错
```

上面示例中， `bigint` 类型赋值为整数和小数，都会报错。

注意，bigint 类型是 ES2020 标准引入的。如果使用这个类型，TypeScript 编译的目标 JavaScript 版本不能低于 ES2020（即编译参数 `target` 不低于 `es2020` ）。

symbol 类型

symbol 类型包含所有的 Symbol 值。

```typescript
const x: symbol = Symbol();
```

上面示例中， `Symbol()` 函数的返回值就是 symbol 类型。

symbol 类型的详细介绍，参见《Symbol》一章。

object 类型

根据 JavaScript 的设计，object 类型包含了所有对象、数组和函数。

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

上面示例中，对象、数组、函数都属于 object 类型。

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` ，表示未定义（即还未给出定义，以后可能会有定义）。

typescript

```
let x: undefined = undefined;
```

上面示例中，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 ，表示为空（即此处没有值）。

typescript

```
const x: null = null;
```

上面示例中，变量 x 就属于 null 类型。

注意，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 。

typescript

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

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

如果希望避免这种情况，则需要打开编译选项 strictNullChecks 。

typescript

```
// 打开编译设置 strictNullChecks
let a = undefined; // undefined
const b = undefined; // undefined

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

上面示例中，打开编译设置 strictNullChecks 以后，赋值为 undefined 的变量会被推断为 undefined 类型，赋值为 null 的变量会被推断为 null 类型。

包装对象类型

包装对象的概念

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。

```javascript
"hello".charAt(1); // 'e'
```

上面示例中，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，就是因为在调用方法时，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的麻烦。

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()`

以上三个构造函数，执行后可以直接获取某个原始类型值的包装对象。

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

上面示例中， `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。

注意， `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。

包装对象类型与字面量类型

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。

```javascript
"hello"; // 字面量
new String("hello"); // 包装对象
```

上面示例中，第一行是字面量，第二行是包装对象，它们都是字符串。

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。

```typescript
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"); // 报错
```

上面示例中， `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。

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

```
Math.abs(n1); // 1
Math.abs(n2); // 报错
```

上面示例中， `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。

上一小节说过， `Symbol()` 和 `BigInt()` 这两个函数不能当作构造函数使用，所以没有办法直接获得 symbol 类型和 bigint 类型的包装对象，因此 `Symbol` 和 `BigInt` 这两个类型虽然存在，但是完全没有使用的理由。

Object 类型与 object 类型

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

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。

typescript

```
let obj: Object;

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

上面示例中，原始类型值、对象、数组、函数都是合法的 `Object` 类型。

事实上，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。

typescript

```
let obj: Object;
```

```
obj = undefined; // 报错
obj = null; // 报错
```

上面示例中， `undefined` 和 `null` 赋值给 `Object` 类型，就会报错。

另外，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。

typescript

```
let obj: {};

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

上面示例中，变量 `obj` 的类型是空对象 `{}` ，就代表 `Object` 类型。

显然，无所不包的 `Object` 类型既不符合直觉，也不方便使用。

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。

typescript

```
let obj: object;

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

上面示例中， `object` 类型不包含原始类型值，只包含对象、数组和函数。

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` 。

注意，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。

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

o1.toString(); // 正确
o1.foo; // 报错

o2.toString(); // 正确
o2.foo; // 报错
```

上面示例中， `toString()` 是对象的原生方法，可以正确访问。 `foo` 是自定义属性，访问就会报错。如何描述对象的自定义属性，详见《对象类型》一章。

---

undefined 和 null 的特殊性

`undefined` 和 `null` 既是值，又是类型。

作为值，它们有一个特殊的地方：任何其他类型的变量都可以赋值为 `undefined` 或 `null` 。

```typescript
let age: number = 24;

age = null; // 正确
age = undefined; // 正确
```

上面代码中，变量 `age` 的类型是 `number` ，但是赋值为 `null` 或 `undefined` 并不报错。

这并不是因为 `undefined` 和 `null` 包含在 `number` 类型里面，而是故意这样设计，任何类型的变量都可以赋值为 `undefined` 和 `null` ，以便跟 JavaScript 的行为保持一致。

JavaScript 的行为是，变量如果等于 `undefined` 就表示还没有赋值，如果等于 `null` 就表示值为空。所以，TypeScript 就允许了任何类型的变量都可以赋值为这两个值。

但是有时候，这并不是开发者想要的行为，也不利于发挥类型系统的优势。

typescript

```
const obj: object = undefined;
obj.toString(); // 编译不报错，运行就报错
```

上面示例中，变量 `obj` 等于 `undefined` ，编译不会报错。但是，实际执行时，调用 `obj.toString()` 就报错了，因为 `undefined` 不是对象，没有这个方法。

为了避免这种情况，及早发现错误，TypeScript 提供了一个编译选项 `strictNullChecks` 。只要打开这个选项， `undefined` 和 `null` 就不能赋值给其他类型的变量（除了 `any` 类型和 `unknown` 类型）。

下面是 tsc 命令打开这个编译选项的例子。

typescript

```
// tsc --strictNullChecks app.ts

let age: number = 24;

age = null; // 报错
age = undefined; // 报错
```

上面示例中，打开 `--strictNullChecks` 以后， `number` 类型的变量 `age` 就不能赋值为 `undefined` 和 `null` 。

这个选项在配置文件 `tsconfig.json` 的写法如下。

json

```
{
  "compilerOptions": {
    "strictNullChecks": true
    // ...
  }
}
```

打开 `strictNullChecks` 以后， `undefined` 和 `null` 这两种值也不能互相赋值了。

typescript

```
// 打开 strictNullChecks

let x: undefined = null; // 报错
let y: null = undefined; // 报错
```

上面示例中， `undefined` 类型的变量赋值为 `null` ，或者 `null` 类型的变量赋值为 `undefind` ，都会报错。

总之，打开 `strictNullChecks` 以后， `undefined` 和 `null` 只能赋值给自身，或者 `any` 类型和 `unknown` 类型的变量。

typescript

```
let x: any = undefined;
let y: unknown = null;
```

值类型

TypeScript 规定，单个值也是一种类型，称为“值类型”。

typescript

```
let x: "hello";

x = "hello"; // 正确
x = "world"; // 报错
```

上面示例中，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。

typescript

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

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

上面示例中，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。

注意， `const` 命令声明的变量，如果赋值为对象，并不会推断为值类型。

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

上面示例中，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。

```typescript
const x: 5 = 4 + 1; // 报错
```

上面示例中，等号左侧的类型是数值 `5` ，等号右侧 `4 + 1` 的类型，TypeScript 推测为 `number` 。由于 `5` 是 `number` 的子类型， `number` 是 `5` 的父类型，父类型不能赋值给子类型，所以报错了（详见本章后文）。

但是，反过来是可以的，子类型可以赋值给父类型。

```typescript
let x: 5 = 5;
let y: number = 4 + 1;

x = y; // 报错
y = x; // 正确
```

上面示例中，变量 `x` 属于子类型，变量 `y` 属于父类型。 `y` 不能赋值为子类型 `x` ，但是反过来是可以的。

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（详见《类型断言》一章）。

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

上面示例中，在 `4 + 1` 后面加上 `as 5` ，就是告诉编译器，可以把 `4 + 1` 的类型视为值类型 `5` ，这样就不会报错了。

只包含单个值的值类型，用处不大。实际开发中，往往将多个值结合，作为联合类型使用。

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` 。

typescript

```
let x: string | number;

x = 123; // 正确
x = "abc"; // 正确
```

上面示例中，变量 `x` 就是联合类型 `string|number` ，表示它的值既可以是字符串，也可以是数值。

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。

typescript

```
let setting: true | false;

let gender: "male" | "female";

let rainbowColor: "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。

typescript

```
let name: string | null;

name = "John";
name = null;
```

上面示例中，变量 `name` 的值可以是字符串，也可以是 `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。

typescript

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

上面示例中，联合类型的第一个成员 `one` 前面，加上了竖杠。

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。

typescript

```
function printId(id: number | string) {
  console.log(id.toUpperCase()); // 报错
}
```

上面示例中，参数变量 `id` 可能是数值，也可能是字符串，这时直接对这个变量调用 `toUpperCase()` 方法会报错，因为这个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。

typescript

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

上面示例中，函数体内部会判断一下变量 `id` 的类型，如果是字符串，就对其执行 `toUpperCase()` 方法。

“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。

typescript

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

```
      return 443;
    }
  }
```

上面示例中，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。

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。

typescript

```
let x: number & string;
```

上面示例中，变量 `x` 同时是数值和字符串，这当然是不可能的，所以 TypeScript 会认为 `x` 的类型实际是 `never` 。

交叉类型的主要用途是表示对象的合成。

typescript

```
let obj: { foo: string } & { bar: string };

obj = {
  foo: "hello",
  bar: "world",
};
```

上面示例中，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。

typescript

```
type A = { foo: number };

type B = A & { bar: number };
```

上面示例中，类型 `B` 是一个交叉类型，用来在 `A` 的基础上增加了属性 `bar` 。

---

type 命令

`type` 命令用来定义一个类型的别名。

typescript
```
type Age = number;

let age: Age = 55;
```

上面示例中， `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。

typescript
```
type Color = "red";
type Color = "blue"; // 报错
```

上面示例中，同一个别名 `Color` 声明了两次，就报错了。

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。

typescript
```
type Color = "red";

if (Math.random() < 0.5) {
  type Color = "blue";
}
```

上面示例中， `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。

typescript
```
type World = "world";
```

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

上面示例中，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。

---

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。

javascript

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

上面示例中， `typeof` 运算符返回字符串 `foo` 的类型是 `string` 。

注意，这时 typeof 的操作数是一个值。

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

javascript

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

上面示例是 `typeof` 运算符在 JavaScript 语言里面，可能返回的八种结果。

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。

typescript

```
const a = { x: 0 };

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

上面示例中， `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。

typescript

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

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

上面示例中，用到了两个 `typeof` ，第一个是类型运算，第二个是值运算。它们是不一样的，不要混淆。

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。

上例的代码编译结果如下。

typescript

```
let a = 1;
let b;
if (typeof a === "number") {
  b = a;
}
```

上面示例中，只保留了原始代码的第二个 typeof，删除了第一个 typeof。

由于编译时不会进行 JavaScript 的值运算，所以 TypeScript 规定，typeof 的参数只能是标识符，不能是需要运算的表达式。

typescript

```
type T = typeof Date(); // 报错
```

上面示例会报错，原因是 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。

```typescript
type Age = number;
type MyAge = typeof Age; // 报错
```

上面示例中， `Age` 是一个类型别名，用作 `typeof` 命令的参数就会报错。

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。

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。

```typescript
if (true) {
  type T = number;
  let v: T = 5;
} else {
```



```
}
```

上面示例中，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。

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。

```typescript
type T = number | string;

let a: number = 1;
let b: T = a;
```

上面示例中，变量 `a` 和 `b` 的类型是不一样的，但是变量 `a` 赋值给变量 `b` 并不会报错。这时，我们就认为， `b` 的类型兼容 `a` 的类型。

TypeScript 为这种情况定义了一个专门术语。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; // 报错
```

上面示例中， `hi` 是 `string` 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。

最后更新: 2023/8/13 15:25

Previous page
any 类型

Next page
数组