TypeScript 的 Enum 类型

目录

- 简介
- Enum 成员的值
- 同名 Enum 的合并
- 字符串 Enum
- keyof 运算符
- 反向映射

Enum 是 TypeScript 新增的一种数据结构和类型，中文译为 “枚举”。

简介

实际开发中，经常需要定义一组相关的常量。

```
const RED = 1;
const GREEN = 2;
const BLUE = 3;

let color = userInput();

if (color === RED) {/* */}
if (color === GREEN) {/* */}
if (color === BLUE) {/* */}

throw new Error('wrong color');
```

上面示例中，常量 `RED`、`GREEN`、`BLUE` 是相关的，意为变量 `color` 的三个可能的取值。它们具体等于什么值其实并不重要，只要不相等就可以了。

TypeScript 就设计了 Enum 结构，用来将相关常量放在一个容器里面，方便使用。

```
enum Color {
  Red,     // 0
  Green,   // 1
  Blue     // 2
}
```

上面示例声明了一个 Enum 结构 `Color`，里面包含三个成员 `Red`、`Green` 和 `Blue`。第一个成员的值默认为整数 `0`，第二个为 `1`，第三个为 `2`，以此类推。

使用时，调用 Enum 的某个成员，与调用对象属性的写法一样，可以使用点运算符，也可以使用方括号运算符。

```
let c = Color.Green; // 1
// 等同于
let c = Color['Green']; // 1
```

Enum 结构本身也是一种类型。比如，上例的变量 `c` 等于 `1`，它的类型可以是 Color，也可以是 `number`。

```
let c:Color = Color.Green; // 正确
let c:number = Color.Green; // 正确
```

上面示例中，变量 `c` 的类型写成 `Color` 或 `number` 都可以。但是，`Color` 类型的语义更好。

Enum 结构的特别之处在于，它既是一种类型，也是一个值。绝大多数 TypeScript 语法都是类型语法，编译后会全部去除，但是 Enum 结构是一个值，编译后会变成 JavaScript 对象，留在代码中。

```
// 编译前
enum Color {
  Red,     // 0
  Green,   // 1
  Blue     // 2
}

// 编译后
let Color = {
  Red: 0,
  Green: 1,
  Blue: 2
};
```

上面示例是 Enum 结构编译前后的对比。

由于 TypeScript 的定位是 JavaScript 语言的类型增强，所以官方建议谨慎使用 Enum 结构，因为它不仅仅是类型，还会为编译后的代码加入一个对象。

Enum 结构比较适合的场景是，成员的值不重要，名字更重要，从而增加代码的可读性和可维护性。

```
enum Operator {
  ADD,
  DIV,
  MUL,
  SUB
}

function compute(
  op:Operator,
  a:number,
  b:number
) {
  switch (op) {
    case Operator.ADD:
      return a + b;
    case Operator.DIV:
      return a / b;
    case Operator.MUL:
      return a * b;
    case Operator.SUB:
      return a - b;
    default:
      throw new Error('wrong operator');
  }
}

compute(Operator.ADD, 1, 3) // 4
```

上面示例中，Enum 结构 `Operator` 的四个成员表示四则运算“加减乘除”。代码根本不需要用到这四个成员的值，只用成员名就够了。

TypeScript 5.0 之前，Enum 有一个 Bug，就是 Enum 类型的变量可以赋值为任何数值。

```
enum Bool {
  No,
  Yes
```

```
}

function foo(noYes:Bool) {
  // ...
}

foo(33);  // TypeScript 5.0 之前不报错
```

上面示例中，函数 `foo` 的参数 `noYes` 是 Enum 类型，只有两个可用的值。但是，TypeScript 5.0 之前，任何数值作为函数 `foo` 的参数，编译都不会报错，TypeScript 5.0 纠正了这个问题。

另外，由于 Enum 结构编译后是一个对象，所以不能有与它同名的变量（包括对象、函数、类等）。

```
enum Color {
  Red,
  Green,
  Blue
}

const Color = 'red'; // 报错
```

上面示例，Enum 结构与变量同名，导致报错。

很大程度上，Enum 结构可以被对象的 `as const` 断言替代。

```
enum Foo {
  A,
  B,
  C,
}

const Bar = {
  A: 0,
  B: 1,
  C: 2,
} as const;

if (x === Foo.A) {}
// 等同于
if (x === Bar.A) {}
```

上面示例中，对象 `Bar` 使用了 `as const` 断言，作用就是使得它的属性无法修改。这样的话，`Foo` 和 `Bar` 的行为就很类似了，前者完全可以用后者替代，而且后者还是 JavaScript 的原生数据结构。

Enum 成员的值

Enum 成员默认不必赋值，系统会从零开始逐一递增，按照顺序为每个成员赋值，比如 0、1、2.....

但是，也可以为 Enum 成员显式赋值。

```
enum Color {
  Red,
  Green,
  Blue
}

// 等同于
enum Color {
  Red = 0,
  Green = 1,
  Blue = 2
}
```

上面示例中，Enum 每个成员的值都是显式赋值。

成员的值可以是任意数值，但不能是大整数（Bigint）。

```
enum Color {
  Red = 90,
  Green = 0.5,
  Blue = 7n // 报错
}
```

上面示例中，Enum 成员的值可以是小数，但不能是 Bigint。

成员的值甚至可以相同。

```
enum Color {
  Red = 0,
  Green = 0,
  Blue = 0
```

```
}
```

如果只设定第一个成员的值，后面成员的值就会从这个值开始递增。

```
enum Color {
  Red = 7,
  Green,  // 8
  Blue   // 9
}

// 或者
enum Color {
  Red, // 0
  Green = 7,
  Blue // 8
}
```

Enum 成员的值也可以使用计算式。

```
enum Permission {
  UserRead     = 1 << 8,
  UserWrite    = 1 << 7,
  UserExecute  = 1 << 6,
  GroupRead    = 1 << 5,
  GroupWrite   = 1 << 4,
  GroupExecute = 1 << 3,
  AllRead      = 1 << 2,
  AllWrite     = 1 << 1,
  AllExecute   = 1 << 0,
}

enum Bool {
  No = 123,
  Yes = Math.random(),
}
```

上面示例中， Enum 成员的值等于一个计算式，或者等于函数的返回值，都是正确的。

Enum 成员值都是只读的，不能重新赋值。

```
enum Color {
  Red,
  Green,
  Blue
}

Color.Red = 4; // 报错
```

上面示例中，重新为 Enum 成员赋值就会报错。

为了让这一点更醒目，通常会在 enum 关键字前面加上 `const` 修饰，表示这是常量，不能再次赋值。

```
const enum Color {
  Red,
  Green,
  Blue
}
```

加上 `const` 还有一个好处，就是编译为 JavaScript 代码后，代码中 Enum 成员会被替换成对应的值，这样能提高性能表现。

```
const enum Color {
  Red,
  Green,
  Blue
}

const x = Color.Red;
const y = Color.Green;
const z = Color.Blue;

// 编译后
const x = 0 /* Color.Red */;
const y = 1 /* Color.Green */;
const z = 2 /* Color.Blue */;
```

上面示例中，由于 Enum 结构前面加了 `const` 关键字，所以编译产物里面就没有生成对应的对象，而是把所有 Enum 成员出现的场合，都替换成对应的常量。

如果希望加上 `const` 关键词后，运行时还能访问 Enum 结构（即编译后依然将 Enum 转成对象），需要在编译时打开 `preserveConstEnums` 编译选项。

同名 Enum 的合并

多个同名的 Enum 结构会自动合并。

```
enum Foo {
  A,
}

enum Foo {
  B = 1,
}

enum Foo {
  C = 2,
}

// 等同于
enum Foo {
  A,
  B = 1,
  C = 2
}
```

上面示例中，`Foo` 分成三段定义，系统会自动把它们合并。

Enum 结构合并时，只允许其中一个的首成员省略初始值，否则报错。

```
enum Foo {
  A,
}

enum Foo {
  B, // 报错
}
```

上面示例中，`Foo` 的两段定义的第一个成员，都没有设置初始值，导致报错。

同名 Enum 合并时，不能有同名成员，否则报错。

```
enum Foo {
  A,
  B
}

enum Foo {
  B = 1, // 报错
  C
}
```

上面示例中，`Foo`的两段定义有一个同名成员`B`，导致报错。

同名 Enum 合并的另一个限制是，所有定义必须同为 const 枚举或者非 const 枚举，不允许混合使用。

```
// 正确
enum E {
  A,
}
enum E {
  B = 1,
}

// 正确
const enum E {
  A,
}
const enum E {
  B = 1,
}

// 报错
enum E {
  A,
}
const enum E {
  B = 1,
}
```

同名 Enum 的合并，最大用处就是补充外部定义的 Enum 结构。

字符串 Enum

Enum 成员的值除了设为数值，还可以设为字符串。也就是说， Enum 也可以用作一组相关字符串的集合。

```
enum Direction {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT',
}
```

上面示例中， `Direction` 就是字符串枚举，每个成员的值都是字符串。

注意，字符串枚举的所有成员值，都必须显式设置。如果没有设置，成员值默认为数值，且位置必须在字符串成员之前。

```
enum Foo {
  A, // 0
  B = 'hello',
  C // 报错
}
```

上面示例中， `A` 之前没有其他成员，所以可以不设置初始值，默认等于 `0`； `C` 之前有一个字符串成员，所以 `C` 必须有初始值，不赋值就报错了。

Enum 成员可以是字符串和数值混合赋值。

```
enum Enum {
  One = 'One',
  Two = 'Two',
  Three = 3,
  Four = 4,
}
```

除了数值和字符串， Enum 成员不允许使用其他值（比如 Symbol 值）。

变量类型如果是字符串 Enum，就不能再赋值为字符串，这跟数值 Enum 不一样。

```
enum MyEnum {
  One = 'One',
  Two = 'Two',
```

```
}

let s = MyEnum.One;
s = 'One'; // 报错
```

上面示例中，变量 `s` 的类型是 `MyEnum`，再赋值为字符串就报错。

由于这个原因，如果函数的参数类型是字符串 Enum，传参时就不能直接传入字符串，而要传入 Enum 成员。

```
enum MyEnum {
  One = 'One',
  Two = 'Two',
}

function f(arg:MyEnum) {
  return 'arg is ' + arg;
}

f('One') // 报错
```

上面示例中，参数类型是 `MyEnum`，直接传入字符串会报错。

所以，字符串 Enum 作为一种类型，有限定函数参数的作用。

前面说过，数值 Enum 的成员值往往不重要。但是有些场合，开发者可能希望 Enum 成员值可以保存一些有用的信息，所以 TypeScript 才设计了字符串 Enum。

```
const enum MediaTypes {
  JSON = 'application/json',
  XML = 'application/xml',
}

const url = 'localhost';

fetch(url, {
  headers: {
    Accept: MediaTypes.JSON,
  },
}).then(response => {
  // ...
});
```

上面示例中，函数 `fetch()` 的参数对象的属性 `Accept` ，只能接受一些指定的字符串。这时就很适合把字符串放进一个 Enum 结构，通过成员值来引用这些字符串。

字符串 Enum 可以使用联合类型（ union）代替。

```
function move(
  where:'Up'|'Down'|'Left'|'Right'
) {
  // ...
 }
```

上面示例中，函数参数 `where` 属于联合类型，效果跟指定为字符串 Enum 是一样的。

注意，字符串 Enum 的成员值，不能使用表达式赋值。

```
enum MyEnum {
  A = 'one',
  B = ['T', 'w', 'o'].join('') // 报错
}
```

上面示例中，成员 `B` 的值是一个字符串表达式，导致报错。

keyof 运算符

keyof 运算符可以取出 Enum 结构的所有成员名，作为联合类型返回。

```
enum MyEnum {
  A = 'a',
  B = 'b'
}

// 'A'|'B'
type Foo = keyof typeof MyEnum;
```

上面示例中，`keyof typeof MyEnum` 可以取出 `MyEnum` 的所有成员名，所以类型 `Foo` 等同于联合类型 `'A'|'B'` 。

注意，这里的 `typeof` 是必需的，否则 `keyof MyEnum` 相当于 `keyof string` 。

```
type Foo = keyof MyEnum;
// number | typeof Symbol.iterator | "toString" | "charAt" | "charCodeAt" | ...
```

上面示例中，类型 `Foo` 等于类型 `string` 的所有原生属性名组成的联合类型。这是 `MyEnum` 为字符串 Enum 的结果，如果 `MyEnum` 是数值 Enum，那么 `keyof MyEnum` 相当于 `keyof number`。

这是因为 Enum 作为类型，本质上属于 `number` 或 `string` 的一种变体，而 `typeof MyEnum` 会将 `MyEnum` 当作一个值处理，从而先将其转为对象类型，就可以再用 `keyof` 运算符返回该对象的所有属性名。

如果要返回 Enum 所有的成员值，可以使用 `in` 运算符。

```
enum MyEnum {
  A = 'a',
  B = 'b'
}

// { a: any, b: any }
type Foo = { [key in MyEnum]: any };
```

上面示例中，采用属性索引可以取出 `MyEnum` 的所有成员值。

反向映射

数值 Enum 存在反向映射，即可以通过成员值获得成员名。

```
enum Weekdays {
  Monday = 1,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

console.log(Weekdays[3]) // Wednesday
```

上面示例中，Enum 成员 `Wednesday` 的值等于 3，从而可以从成员值 `3` 取到对应的成员名 `Wednesday`，这就叫反向映射。

这是因为 TypeScript 会将上面的 Enum 结构，编译成下面的 JavaScript 代码。

```
var Weekdays;
(function (Weekdays) {
```

```
    Weekdays[Weekdays["Monday"] = 1] = "Monday";
    Weekdays[Weekdays["Tuesday"] = 2] = "Tuesday";
    Weekdays[Weekdays["Wednesday"] = 3] = "Wednesday";
    Weekdays[Weekdays["Thursday"] = 4] = "Thursday";
    Weekdays[Weekdays["Friday"] = 5] = "Friday";
    Weekdays[Weekdays["Saturday"] = 6] = "Saturday";
    Weekdays[Weekdays["Sunday"] = 7] = "Sunday";
})(Weekdays || (Weekdays = {}));
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
Weekdays[
  Weekdays["Monday"] = 1
] = "Monday";
```

□□□□□□□□□□□□□□□ = □□□□□□□□□□□□□□□□

```
Weekdays["Monday"] = 1;
Weekdays[1] = "Monday";
```

□□□□□□□□□□□□□□□ Enum□□□□□□□ Enum□□□□□□□□□□□□□□□□□□□□ Enum □□□□□□□□□□□

```
enum MyEnum {
  A = 'a',
  B = 'b'
}

// □□□
var MyEnum;
(function (MyEnum) {
    MyEnum["A"] = "a";
    MyEnum["B"] = "b";
})(MyEnum || (MyEnum = {}));
```

□□□ #2
□ □□□ □□□ 2023-07-09 12:07:22 UTC
□ □□□ □□□ 2025-08-30 09:59:09 UTC