# Enum 类型

Enum（枚举）是 TypeScript 新增的一种数据结构和类型，用来将相关常量放在一个容器里面。

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

console.log(Color.Red);      // 0
console.log(Color['Green']); // 1
console.log(Color['Blue']);  // 2
```

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。调用 Enum 的某一个成员，与调用对象属性的写法一样，可以使用点运算符，也可以使用方扩号运算符。

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

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

编译前:

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

编译后:

```
"use strict";
var Color;
(function (Color) {
    Color[Color["Red"] = 0] = "Red";
    Color[Color["Green"] = 1] = "Green";
    Color[Color["Blue"] = 2] = "Blue";
})(Color || (Color = {}));
```

**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
compute(Operator.DIV, 6, 3); // 2
compute(Operator.ADD, 1, 3); // 3
compute(Operator.ADD, 3, 1); // 2
```

**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'; // Enum declarations can only merge with namespace or other
enum declarations. 枚举声明只能与命名空间或其他枚举声明合并。
```

上例中，Enum 结构与变量同名，导致报错。

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;

let x = 2;
if (x === Foo.A) {}
// 等同于
if (x === Bar.A) {}
```

上例中，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。

## 1. 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
}
```

**成员的值可以是任意数值，但不能是大整数（Bigint）。**

```
// Enum 成员的值可以是小数，但不能是 Bigint。
enum Color {
  Red = 90,
  Green = 0.5,
  Blue = 7n // Type 'bigint' is not assignable to type 'number' as required for
computed enum member values. 类型'bigint'不能赋值给类型'number'作为计算枚举成员值的要
求。
}
```

成员的值甚至可以相同。

```
enum Color {
  Red = 0,
  Green = 0,
  Blue = 0
}
console.log(Color3.Red);      // 0
console.log(Color3['Green']); // 0
console.log(Color3['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 Color4 {
  Red = 7,
  Green,
  Blue
}
console.log(Color4.Red);      // 7
console.log(Color4['Green']); // 8
console.log(Color4['Blue']);  // 9

enum Color5 {
  Red,
  Green = 7,
  Blue
}
console.log(Color5.Red);      // 0
console.log(Color5['Green']); // 7
console.log(Color5['Blue']);  // 8
```

Enum 成员的值也可以使用计算式，和函数的返回值。

```
enum MyEnum1 {
  A = 123,
  B = 1 + 2
}

enum MyEnum2 {
  A = 123,
  B = Math.random(),
}
```

**Enum 成员值都是只读的，不能重新赋值。**

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

Color.Red = 4; // Cannot assign to 'Red' because it is a read-only property. 不能赋值给'Red', 因为它是一个只读属性。
```

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
"use strict";
const x = 0 /* Color.Red */;
const y = 1 /* Color.Green */;
const z = 2 /* Color.Blue */;
```

**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。**

## 2. 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
}
```

**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, // In an enum with multiple declarations, only one declaration can omit an initializer for its first enum element. 在具有多个声明的枚举中，只有一个声明可以省略其第一个枚举元素的初始化式。
}
```

**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, //  Duplicate identifier 'B'. 重复标识符'B'。
  C
}
```

**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 E1 {
  A,
}
enum E1 {
  B = 1,
}
```

```
// 正确
const enum E2 {
  A,
}
const enum E2 {
  B = 1,
}

// Enum declarations can only merge with namespace or other enum declarations. 枚举声明只能与命名空间或其他枚举声明合并。
enum E3 {
  A,
}
const enum E3 {
  B = 1,
}
```

**同名 Enum 的合并，最大用处就是补充外部定义的 Enum 结构。**

## 3. 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',
}
```

`Direction` 就是字符串枚举，每个成员的值都是字符串。

**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 Foo1 {
  A, // 0,  第一个成员不设初始值默认为 0
  B = 'hello',
  C //  Enum member must have initializer. 枚举成员必须有初始化式。
}

// 成员的类型可以是 number 和 string，两者可以混合在一个 Enum 中
enum Foo2 {
  A = 1,
  B = 'hello',
  C = 0
}
```

```
enum Foo3 {
  A, // 0
  B = 'hello',
  C = 'world'
}
```

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、Boolean）。

```
enum Foo {
  A = true // Type 'boolean' is not assignable to type 'number' as required for computed enum member values. 类型“boolean”不能赋值给类型“number”，因为计算枚举成员值需要赋值。
}
enum Foo2 {
  A = true // Type 'symbol' is not assignable to type 'number' as required for computed enum member values. 类型“symbol”不能赋值给类型“number”，因为计算枚举成员值需要赋值。
}
```

**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'; // Type '"One"' is not assignable to type 'MyEnum'.
```

上例中，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(MyEnum['One']); // 正确
f(MyEnum.One); // 正确

f('One'); //  Argument of type '"One"' is not assignable to parameter of type
'MyEnum'.
```

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。** 效果跟指定为字符串 Enum 是一样的

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

**字符串 Enum 的成员值，不能使用字符串表达式赋值。可以使用数值表达式、函数返回值，和可以转换为数值的计算值。**

```
enum MyEnum1 {
  A = 'one',
  B = ['1', '2', '3'].join('') // Type 'string' is not assignable to type 'number'
as required for computed enum member values. 类型“string”不能按计算枚举成员值的要求赋值给类型“number”。
}
```

```
enum MyEnum2 {
  A = 'one',
  B = String(1) // Type 'string' is not assignable to type 'number' as required
for computed enum member values.
}
enum MyEnum3 {
  A = 123,
  B = ['1', '2'][0] // Type 'string' is not assignable to type 'number' as
required for computed enum member values.
}

enum MyEnum10 {
  A = 'one',
  B = Number(['1', '2', '3'].join('')), // 正确，数值表达式
}
enum MyEnum11 {
  A = 'one',
  B = Number('1') // 正确
}
enum MyEnum12 {
  A = 123,
  B = Number(['1', '2'][0]) // 正确
}
enum MyEnum13 {
  A = 123,
  B = [1, 2][0] // 正确
}
enum MyEnum14 {
  A = 123,
  B = Math.random(), // 正确，函数返回值
}
enum MyEnum15 {
  A = 123,
  B = 1 + 2, // 正确，数值计算值
}
enum MyEnum16 {
  A = 123,
  B = '1' + '2', // 正确
}
type e = {[key in MyEnum16]: number};  // type e = { 123: number; 12: number; }
```

成员 `B` 的值是一个字符串表达式，导致报错。

## 4. 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'
}
```

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

`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'`。

**这里的 `typeof` 是必需的，否则 `keyof MyEnum` 相当于 `keyof number` 。**

```
type Foo = keyof MyEnum;
// "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf" |
"toLocaleString"
```

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 MyEnum13 {
  A = 'a',
  B = 'b'
}
type Foo13 = { [key in MyEnum]: number }; // type Foo13 = { a: number; b: number;
}
```

## 5. 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。** 字符串 Enum 不存在反向隐射。

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

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

上例中，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`，这是反向映射。

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

```
"use strict";
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 = {}));
```

上例中，实际进行了两组赋值，以第一个成员为例。

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

上面代码有两个赋值运算符（=），实际上等同于下面的代码。

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

**这种情况只发生在数值 Enum，对于字符串 Enum，不存在反向映射。这是因为字符串 Enum 编译后只有一组赋值。**

编译前

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

编译后：

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