TypeScript 的元组类型

简介

元组（tuple）是 TypeScript 特有的数据类型，JavaScript 没有单独区分这种类型。它表示成员类型可以自由设置的数组，即数组的各个成员的类型可以不同。

元组必须明确声明每个成员的类型。

typescript

```
const s: [string, string, boolean] = ["a", "b", true];
```

上面示例中，元组 `s` 的前两个成员的类型是 `string` ，最后一个成员的类型是 `boolean` 。

元组类型的写法，与上一章的数组有一个重大差异。数组的成员类型写在方括号外面（ `number[]` ），元组的成员类型是写在方括号里面（ `[number]` ）。

TypeScript 的区分方法是，成员类型写在方括号里面的就是元组，写在外面的就是数组。

typescript

```
let a: [number] = [1];
```

上面示例中，变量 `a` 是一个元组，只有一个成员，类型是 `number` 。

使用元组时，必须明确给出类型声明（上例的 `[number]` ），不能省略，否则 TypeScript 会把一个值自动推断为数组。

typescript

```
// a 的类型为 (number | boolean)[]
let a = [1, true];
```

上面示例中，变量 `a` 的值其实是一个元组，但是 TypeScript 会将其推断为一个联合类型的数组，即 `a` 的类型为 `(number | boolean)[]` 。

元组成员的类型可以添加问号后缀（ ? ），表示该成员是可选的。

```typescript
let a: [number, number?] = [1];
```

上面示例中，元组 a 的第二个成员是可选的，可以省略。

注意，问号只能用于元组的尾部成员，也就是说，所有可选成员必须在必选成员之后。

```typescript
type myTuple = [number, number, number?, string?];
```

上面示例中，元组 myTuple 的最后两个成员是可选的。也就是说，它的成员数量可能有两个、三个和四个。

由于需要声明每个成员的类型，所以大多数情况下，元组的成员数量是有限的，从类型声明就可以明确知道，元组包含多少个成员，越界的成员会报错。

```typescript
let x: [string, string] = ["a", "b"];

x[2] = "c"; // 报错
```

上面示例中，变量 x 是一个只有两个成员的元组，如果对第三个成员赋值就报错了。

但是，使用扩展运算符（ ... ），可以表示不限成员数量的元组。

```typescript
type NamedNums = [string, ...number[]];

const a: NamedNums = ["A", 1, 2];
const b: NamedNums = ["B", 1, 2, 3];
```

上面示例中，元组类型 NamedNums 的第一个成员是字符串，后面的成员使用扩展运算符来展开一个数组，从而实现了不定数量的成员。

扩展运算符用在元组的任意位置都可以，但是它后面只能是数组或元组。

```typescript
type t1 = [string, number, ...boolean[]];
type t2 = [string, ...boolean[], number];
type t3 = [...boolean[], string, number];
```

上面示例中，扩展运算符分别在元组的尾部、中部和头部。

如果不确定元组成员的类型和数量，可以写成下面这样。

typescript

```typescript
type Tuple = [...any[]];
```

上面示例中，元组 `Tuple` 可以放置任意数量和类型的成员。但是这样写，也就失去了使用元组和 TypeScript 的意义。

元组可以通过方括号，读取成员类型。

typescript

```typescript
type Tuple = [string, number];
type Age = Tuple[1]; // number
```

上面示例中， `Tuple[1]` 返回 1 号位置的成员类型。

由于元组的成员都是数值索引，即索引类型都是 `number` ，所以可以像下面这样读取。

typescript

```typescript
type Tuple = [string, number, Date];
type TupleEl = Tuple[number]; // string|number|Date
```

上面示例中， `Tuple[number]` 表示元组 `Tuple` 的所有数值索引的成员类型，所以返回 `string|number|Date` ，即这个类型是三种值的联合类型。

只读元组

元组也可以是只读的，不允许修改，有两种写法。

typescript

```typescript
// 写法一
type t = readonly [number, string];

// 写法二
type t = Readonly<[number, string]>;
```

上面示例中，两种写法都可以得到只读元组，其中写法二是一个泛型，用到了工具类型 `Readonly<T>` 。

跟数组一样，只读元组是元组的父类型。所以，元组可以替代只读元组，而只读元组不能替代元组。

```typescript
type t1 = readonly [number, number];
type t2 = [number, number];

let x: t2 = [1, 2];
let y: t1 = x; // 正确

x = y; // 报错
```

上面示例中，类型 `t1` 是只读元组，类型 `t2` 是普通元组。 `t2` 类型可以赋值给 `t1` 类型，反过来就会报错。

由于只读元组不能替代元组，所以会产生一些令人困惑的报错。

```typescript
function distanceFromOrigin([x, y]: [number, number]) {
  return Math.sqrt(x ** 2 + y ** 2);
}

let point = [3, 4] as const;

distanceFromOrigin(point); // 报错
```

上面示例中，函数 `distanceFromOrigin()` 的参数是一个元组，传入只读元组就会报错，因为只读元组不能替代元组。

读者可能注意到了，上例中 `[3, 4] as const` 的写法，在上一章讲到，生成的是只读数组，其实生成的同时也是只读元组。因为它生成的实际上是一个只读的“值类型” `readonly [3, 4]` ，把它解读成只读数组或只读元组都可以。

上面示例报错的解决方法，就是使用类型断言，在最后一行将传入的参数断言为普通元组，详见《类型断言》一章。

```typescript
distanceFromOrigin(point as [number, number]);
```

成员数量的推断

如果没有可选成员和扩展运算符，TypeScript 会推断出元组的成员数量（即元组长度）。

typescript

```
function f(point: [number, number]) {
  if (point.length === 3) {
    // 报错
    // ...
  }
}
```

上面示例会报错，原因是 TypeScript 发现元组 `point` 的长度是 `2` ，不可能等于 `3` ，这个判断无意义。

如果包含了可选成员，TypeScript 会推断出可能的成员数量。

typescript

```
function f(point: [number, number?, number?]) {
  if (point.length === 4) {
    // 报错
    // ...
  }
}
```

上面示例会报错，原因是 TypeScript 发现 `point.length` 的类型是 `1|2|3` ，不可能等于 `4` 。

如果使用了扩展运算符，TypeScript 就无法推断出成员数量。

typescript

```
const myTuple: [...string[]] = ["a", "b", "c"];

if (myTuple.length === 4) {
  // 正确
  // ...
}
```

上面示例中， `myTuple` 只有三个成员，但是 TypeScript 推断不出它的成员数量，因为它的类型用到了扩展运算符，TypeScript 把 `myTuple` 当成数组看待，而数组的成员数量是不确定的。

一旦扩展运算符使得元组的成员数量无法推断，TypeScript 内部就会把该元组当成数组处理。

扩展运算符与成员数量

扩展运算符（ `...` ）将数组（注意，不是元组）转换成一个逗号分隔的序列，这时 TypeScript 会认为这个序列的成员数量是不确定的，因为数组的成员数量是不确定的。

这导致如果函数调用时，使用扩展运算符传入函数参数，可能发生参数数量与数组长度不匹配的报错。

typescript

```typescript
const arr = [1, 2];

function add(x: number, y: number) {
  // ...
}

add(...arr); // 报错
```

上面示例会报错，原因是函数 `add()` 只能接受两个参数，但是传入的是 `...arr` ，TypeScript 认为转换后的参数个数是不确定的。

有些函数可以接受任意数量的参数，这时使用扩展运算符就不会报错。

typescript

```typescript
const arr = [1, 2, 3];
console.log(...arr); // 正确
```

上面示例中， `console.log()` 可以接受任意数量的参数，所以传入 `...arr` 就不会报错。

解决这个问题的一个方法，就是把成员数量不确定的数组，写成成员数量确定的元组，再使用扩展运算符。

typescript

```typescript
const arr: [number, number] = [1, 2];

function add(x: number, y: number) {
  // ...
}

add(...arr); // 正确
```

上面示例中， `arr` 是一个拥有两个成员的元组，所以 TypeScript 能够确定 `...arr` 可以匹配函数 `add()` 的参数数量，就不会报错了。

另一种写法是使用 `as const` 断言。

typescript

```
const arr = [1, 2] as const;
```

上面这种写法也可以，因为 TypeScript 会认为 `arr` 的类型是 `readonly [1, 2]` ，这是一个只读的值类型，可以当作数组，也可以当作元组。

最后更新: 2023/8/13 15:25

Previous page
数组

Next page
symbol 类型