元组类型

1. 介绍

元组是特有的数据类型，数组内的成员可以是各个类型的，但是每个成员要声明其类型。越界会报错

在方括号里面写各个成员的类型。

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

元组和数组最大的区别：元组的成员类型写在方括号里面，数组的成员类型写在方括号外部。

```
// 数组
let a:number[] = [1];

// 元组
let t:[number] = [1];
```

1. 不写成员类型，ts自动推断可能会出错

如果不给出成员的数据类型，ts会根据其值推断，可能会造成不同。

这里变量a本来是只读数组类型，结果没有显式声明，ts结果推断为了联合类型的数组。

```
// a 的类型被推断为 (number | boolean)[]
let a = [1, true];
```

2. 后缀问号(?)

元组成员的类型可以添加问号后缀（`?`），表示该成员是可选的。问号必须在数组的尾部

```
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。

3. 扩展运算符(...)

使用扩展运算符可以表示不限成员数量的元组

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

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

扩展运算符（`...`）用在元组的任意位置都可以，它的后面只能是一个数组或元组。

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

如果不确定元组成员的类型和数量，可以写成下面这样。

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

元组的成员可以添加成员名，这个成员名是说明性的，可以任意取名，没有实际作用。

```
type Color = [
  red: number,
  green: number,
  blue: number
];

const c:Color = [255, 255, 255];

```

上面示例中，类型`Color`是一个元组，它有三个成员。每个成员都有一个名字，写在具体类型的前面，使用冒号分隔。这几个名字可以随便取，没有实际作用，只是用来说明每个成员的含义。

元组可以通过方括号，读取成员类型。

```
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。

```
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。

2. 只读元组

元组也可以是只读的，不允许修改，有两种写法。

第一种：在类型前面添加readonly关键字。

第二种：泛型Readonly<T>

```
// 写法一
type t = readonly [number, string]
// 写法二
type t = Readonly<[number, string]>
```

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。

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

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

x = y; // 报错
```

3. 成员数量的推断

如果没有可选成员和扩展运算符，ts会自动推算成员的数量。

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

```

上面示例会报错，原因是 ts 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，ts 会推断出可能的成员数量。

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

```

上面示例会报错，原因是 ts 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，ts 就无法推断出成员数量。

```ts
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 ts 推断不出它的成员数量，因为它的类型用到了扩展运算符，ts 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，ts 内部就会把该元组当成数组处理。

4. 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 ts 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。**

```
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`，ts 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。

```
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。

如何解决

- **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，**

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

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

add(...arr) // 正确
```

- 另一种写法是使用`as const`断言。

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

```

上面这种写法也可以，因为 ts 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。