元组

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。

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

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`。

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]`）。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。

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

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。

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

上例中，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。

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

上例中，元组 `a` 的第二个成员是可选的，可以省略。

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。

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

上例中，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。

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

x[2] = 'c'; // Tuple type '[string, string]' of length '2' has no element at index
'2'.
```

上例中，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。

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];
```

上例中，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。

```
// 上例中，扩展运算符分别在元组的尾部、中部和头部。
type t1 = [string, number, ...boolean[]];
type t2 = [string, ...boolean[], number];
type t3 = [...boolean[], string, number];

const t1: t1 = ['1', 1, true, false, true, true];
const t2: t2 = ['1', true, false, false, 1];
const t3: t3 = [false, true, '1', 1];
```

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[]];
```

上例中，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。

```
type Tuple = [string, number];
type Age = Tuple[1];  // number
type Name = Tuple[0]; // string
let age: Age = 18;
let nickName: Name = 'lisi';
```

上例中，`Tuple[1]` 返回 `1` 号位置的成员类型 `number`，`Tuple[0]` 返回 `0` 号位置的成员类型 `string`。

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

let a1: TupleEl = 'a';
let a2: TupleEl = 1;
let a3: TupleEl = new Date();
```

上例中，`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。

```
// 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。

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

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

x = y; // The type 't1' is 'readonly' and cannot be assigned to the mutable type
't2'.
```

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。

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

let point = [3, 4] as const;

distanceFromOrigin(point); // 报错
```

上例中，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。

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。

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

3. 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）。

```
function f(point: [number, number]) {
  if (point.length === 3) {  // This comparison appears to be unintentional
because the types '2' and '3' have no overlap.
    // ...
  }
}
```

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。

```
function f(
  point:[number, number?, number?]
) {
  if (point.length === 4) {  //  This comparison appears to be unintentional
because the types '1 | 2 | 3' and '4' have no overlap.这种比较似乎是无意的，因为类
型'1 | 2 | 3'和'4'没有重叠。
    // ...
  }
}
```

如果使用了扩展运算符，TypeScript 就无法推断出成员数量。

```
const myTuple:[...string[]] = ['a', 'b', 'c'];
if (myTuple.length === 4) { // 正确
  // ...
}
```

上例中，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。

4. 扩展运算符与成员数量

扩展运算符（...）将数组（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。

```
const arr = [1, 2];
function add(x:number, y:number){
  // ...
}
add(...arr) // A spread argument must either have a tuple type or be passed to a
rest parameter.扩展参数必须具有元组类型，或者传递给rest形参。
```

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。

需要改成下面这样：

```
const t: [number, number] = [1, 2];
function add(x: number, y: number) {
  // ...
}
add(...t); // 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 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。