数组

TypeScript 数组有一个根本特征：所有成员的类型必须相同，但是成员数量是不确定的，可以是无限数量的成员，也可以是零成员。

数组类型有两种写法：

```
let arr1:number[] = [1, 2, 3];
let arr2:(number|string)[] = [1, 2, 3];

let arr3: Array<number> = [1, 2, 3];
let arr4: Array<number|string> = [1, 2, 3];
```

由于成员数量可以动态变化，所以 TypeScript 不会对数组边界进行检查，越界访问数组并不会报错。

```
let arr:number[] = [1, 2, 3];
let foo = arr[3]; // 正确
```

TypeScript 允许使用方括号读取数组成员的类型。

```
type Names = string[];
type Name = Names[0]; // string
```

上例中，类型 `Names` 是字符串数组，那么 `Names[0]` 返回的类型就是 `string`。

由于数组成员的索引类型都是 `number`，所以读取成员类型也可以写成下面这样。

```
type Names = string[];
type Name = Names[number]; // string
```

上例中，`Names[number]` 表示数组 Names 所有数值索引的成员类型，所以返回 `string`。

1. 数组类型推断

如果数组变量没有声明类型，TypeScript 就会推断数组成员的类型。这时，推断行为会因为值的不同，而有所不同。

如果变量的初始值是空数组，那么 TypeScript 会推断数组类型是 `any[]`。

```
// 推断为 any[]
const arr = [];
```

后面，为这个数组赋值时，TypeScript 会自动更新类型推断。

```
const arr = [];
arr; // 推断为 any[]

arr.push(123);
arr; // 推断类型为 number[]

arr.push('abc');
arr; // 推断类型为 (string|number)[]
```

上例中，数组变量 `arr` 的初始值是空数组，然后随着新成员的加入，TypeScript 会自动修改推断的数组类型。

类型推断的自动更新只发生初始值为空数组的情况。

如果初始值不是空数组，类型推断就不会更新。

```
// 推断类型为 number[]
const arr = [123];

arr.push('abc'); // Argument of type 'string' is not assignable to parameter of
type 'number'.
```

上例中，数组变量 `arr` 的初始值是 `[123]`，TypeScript 就推断成员类型为 `number`。新成员如果不是这个类型，TypeScript 就会报错，而不会更新类型推断。

2. 只读数组

TypeScript 允许声明只读数组，方法是在数组类型前面加上 `readonly` 关键字。

```
const arr:readonly number[] = [0, 1];

arr[1] = 2;    // Index signature in type 'readonly number[]' only permits
reading.
arr.push(3);   // Property 'push' does not exist on type 'readonly number[]'.
delete arr[0]; // Index signature in type 'readonly number[]' only permits
reading.
```

上例中，`arr` 是一个只读数组，删除、修改、新增数组成员都会报错。

TypeScript 将 `readonly number[]` 与 `number[]` 视为两种不一样的类型，后者是前者的子类型。这是因为只读数组没有 `pop()`、`push()` 之类会改变原数组的方法，所以 `number[]` 的方法数量要多于 `readonly number[]`，这意味着 `number[]` 其实是 `readonly number[]` 的子类型。

子类型继承了父类型的所有特征，并加上了自己的特征，所以子类型 `number[]` 可以用于所有使用父类型的场合，反过来就不行。

```
let a1:number[] = [0, 1];
let a2:readonly number[] = a1; // 正确
a1 = a2; // The type 'readonly number[]' is 'readonly' and cannot be assigned to
the mutable type 'number[]'.
```

上例中，子类型 `number[]` 可以赋值给父类型 `readonly number[]`，但是反过来就会报错。只读类型不可分配给可变类型。

由于只读数组是数组的父类型，所以它不能代替数组。这一点很容易产生令人困惑的报错。

```
function getSum(s:number[]) { /* */ }
const arr:readonly number[] = [1, 2, 3];
getSum(arr) // The type 'readonly number[]' is 'readonly' and cannot be assigned
to the mutable type 'number[]'.
```

`readonly` 关键字不能与数组的泛型写法一起使用。

```
const arr:readonly Array<number> = [0, 1]; // 'readonly' type modifier is only
permitted on array and tuple literal types.
```

TypeScript 提供了两个专门的泛型，用来生成只读数组的类型。

```
const a1:ReadonlyArray<number> = [0, 1];
const a2:Readonly<number[]> = [0, 1];
```

泛型 `ReadonlyArray<T>` 和 `Readonly<T[]>` 都可以用来生成只读数组类型。两者尖括号里面的写法不一样，`Readonly<T[]>` 的尖括号里面是整个数组（`number[]`），而 `ReadonlyArray<T>` 的尖括号里面是数组成员（`number`）。

3. const 断言

只读数组还有一种声明方法，就是使用“const 断言”。

```
const arr = [0, 1] as const;
arr[0] = [2]; // Cannot assign to '0' because it is a read-only property.
```

4. 多维数组

TypeScript 使用 `T[][]` 的形式，表示二维数组，`T` 是最底层数组成员的类型。

```
let multi1:number[][] = [[1,2,3], [23,24,25]];
let multi2:Array<Array<number>> = [[1,2,3], [23,24,25]];
```

上例中，变量 multi 的类型是 number[][]，表示它是一个二维数组，最底层的数组成员类型是 number。