第5章

集合类型

集合(collection)是用于存储多个元素的数据结构。TypeScript 中的数组、元组(tuple)、集合(set,一种特殊的 collectim)、映射(map)等类型都可被视作集合类型。


视频讲解

5.1 数组

2.1.4 节已经简单介绍过数组。数组是一种有序的集合,可以存储相同类型的多个元素。它可以通过索引访问元素,并且可以动态调整大小。

5.1.1 创建数组对象

有两种方式创建数组对象,一种是用符号[]直接加元素创建,另一种使用泛型类 Array 来创建。

【例 5-1】 用两种方式创建数组对象

```
1.    let fabs:number[] = [1,1,2,3,5]
2.    console.log(fabs)
3.    let nums:Array<number> = new Array(1,1,2,3,5)
4.    nums[7] = 21                //TypeScript 中数组是自动增长的,没有下标越界问题
5.    console.log(nums)
```

第 1 行和第 3 行,分别用两种方式创建数组对象。

第 4 行,设置下标 7 对应的元素值。注意,nums 数组只有 5 个元素,下标 7 显然超过了下标[0,4]的范围值,但由于在 TypeScript 中数组是自动增长的,因此没有产生下标越界问题。

执行结果为:

```
[ 1, 1, 2, 3, 5 ]
[ 1, 1, 2, 3, 5, <2 empty items>, 21 ]
```

5.1.2 Array 类常用函数和属性

Array 类的常用函数如表 5-1 所示。

表 5-1 Array 类的常用函数和属性

函　数	描　述
concat()	拼接数组,并返回结果
join()	将元素值合并为一个字符串。默认用逗号连接,可指定连接符
every()	通过回调函数判断是否每个元素都满足条件
some()	通过回调函数判断是否存在满足条件的元素
filter()	通过回调函数遍历每个元素,返回满足条件的元素
map()	通过回调函数处理每个元素,返回处理后的数组
reduce()	通过回调函数遍历每个元素,并计算为一个值。注：reduce 具有缩小归纳之意
forEach()	通过回调函数遍历每个元素,对每个元素都执行一次
indexOf()	正向搜索元素首次出现的位置
lastIndexOf()	反向搜索元素首次出现的位置
push()	在数组尾部追加一个或多个元素,返回为数组长度
pop()	弹出数组尾部元素,并返回该元素
shift()	弹出数组头部元素,并返回该元素
unshift()	在数组头部追加若干元素,并返回长度
splice()	删除指定下标开始的若干元素。可选操作：在删除时可插入若干元素
slice()	获取起始下标到结束下标之间的元素。注意,不会获取结束下标处元素
reverse()	将数组的元素进行倒序排列
sort()	将数组的元素进行正序排序

【例 5-2】 连接操作：数组拼接成新数组时用 concat(),元素连接成字符串时用 join()

```
1.    let ary = [1,1,2]
2.    let fabs = ary.concat([3,5,8])        //拼接数组,并返回结果
3.    console.log(fabs)                     //[ 1, 1, 2, 3, 5, 8 ]
4.    let str = fabs.join()                 //元素值合并为一个字符串,默认连接符为逗号
5.    console.log(str)                      //1,1,2,3,5,8
6.    str = fabs.join('/')
7.    console.log(str)                      //1/1/2/3/5/8
```

第 2 行,用 concat()函数,将 ary 中元素和[3,5,8]中元素拼接为一个新的数组。
第 4 行,用 join()函数,将元素值合并为一个字符串。注意,默认连接符为逗号“,”。
第 6 行,用 join()函数,指定斜杠符/作为连接符,将数组中元素合并为一个字符串。
执行结果为：

```
[ 1, 1, 2, 3, 5, 8 ]
1,1,2,3,5,8
1/1/2/3/5/8
```

通常使用 every()和 some()函数来判断元素是否满足条件。都满足用 every(),部分满

足用 some()。

【例 5-3】 判断元素条件：都满足用 every()函数、部分满足用 some()函数

```
1.    let fabs = [ 1, 1, 2, 3, 5, 8 ]
2.    let everyOdd = fabs.every(                // 每个元素是否都满足条件
3.        (value,index,array) => value % 2 == 1
4.    )
5.    console.log(everyOdd)                     //false
6.    let someOdd = fabs.some(                  // 是否存在元素满足条件
7.        (value,index,array) => value % 2 == 1
8.    )
9.    console.log(someOdd)                      //true
```

第 2～4 行，在 every()函数中使用回调函数(此处为箭头函数)来判断：fab 数组的每个元素是否都为奇数。

第 6～8 行，在 some()函数中使用回调函数(此处为箭头函数)来判断：fab 数组中是否存在为奇数的元素。

执行结果为：

```
false
true
```

另有 filter()、map()、reduce()、forEach()等函数也使用回调函数，可在遍历元素过程中实现一些特殊功能。

【例 5-4】 filter()、map()、reduce()、forEach()函数使用

```
1.    let allOdd = [1,1,2,3,5,8].filter(        //对每个元素进行判断,返回满足条件的元素
2.        (value,index,array) => value % 2 == 1
3.    )
4.    console.log(allOdd)                       //[ 1, 1, 3, 5 ]
5.
6.    let scoresChg = [36,49,64].map(Math.sqrt) //[ 6, 7, 8 ] // 遍历元素,映射为新值
7.    console.log(scoresChg)
8.
9.    let sum = [1,1,2,3].reduce (              //遍历元素,并计算为一个值
10.       (total,e) => total + e
11.   )
12.   console.log('total = ',sum);
13.
14.   [1,1,2,3].forEach( //遍历元素,每个元素都执行一次
15.       (value,index,array) => {
16.           console.log(index + ': ',value)
17.   })
```

第 1～3 行，在 filter()函数中通过回调函数遍历每个元素，当元素满足条件时，将其放入返回数组。

第 6 行，在 map()函数中通过回调函数遍历每个元素，用指定函数映射处理每个元素，

并将结果放入返回数组。

第 9～11 行，在 reduce()函数中通过回调函数遍历每个元素，并计算为一个值。注意，回调函数有两个参数：第 1 个参数存放计算值，第 2 个参数为遍历的元素。

第 14～17 行，在 forEach()函数中通过回调函数遍历每个元素，每个元素都被执行一次。

执行结果为：

```
[ 1, 1, 3, 5 ]
[ 6, 7, 8 ]
total = 7
0: 1
1: 1
2: 2
3: 3
```

搜索元素出现的位置有两个函数：正向搜索用 indexOf()，反向搜索用 lastIndexOf()。

【例 5-5】 搜索元素出现的位置

```
1.    let fabs = [ 1, 1, 2, 3, 5, 8 ]
2.    console.log(fabs.indexOf(1))              //0 正向搜索元素首次出现的位置
3.    console.log(fabs.lastIndexOf(1))          //1 反向搜索元素首次出现的位置
```

第 2 行，使用 indexOf()函数正向搜索元素 1 首次出现的位置。

第 3 行，使用 lastIndexOf()函数反向搜索元素 1 首次出现的位置。

执行结果为：

```
0
1
```

元素追加、弹出、在指定位置进行删除或添加、获取指定区间元素等操作，也有相应的函数。

【例 5-6】 元素追加、弹出、在指定位置进行删除或添加、获取指定区间元素操作

```
1.    let ary = [0,1,2]
2.    let len = ary.push(3)     //在尾部追加一个或多个元素,返回数组长度
3.    console.log(ary, len)     //[ 0, 1, 2, 3 ] 4
4.    let el = ary.pop()        //在尾部弹出元素,并返回该元素
5.    console.log(ary, el)      //[ 0, 1, 2 ] 3
6.    el = ary.shift()          //在头部弹出元素,并返回该元素
7.    console.log(ary, el)      //[ 1, 2 ] 0
8.    len = ary.unshift( - 1, 0) //在头部追加一个或多个元素,并返回数组长度
9.    console.log(ary, len)     //[ -1, 0, 1, 2 ] 4
10.
11.   ary = [0,1,2,3,4,5]
12.   ary.splice(1,2, 22,33)  //从下标 1 处开始删除,删除元素个数 2,在删除位置插入元素 22、33
13.   console.log(ary)          // 0, 22, 33, 3, 4, 5 ]
```

```
14.    ary.splice(1,2)          //删除操作对应的起始下标,删除几个元素
15.    console.log(ary)         //[ 0, 3, 4, 5 ]
16.
17.    console.log( [1,1,2,3,5].slice(2,4) )   //按下标[开始,结束)选取数组的一部分并返回
```

第 2 行,用 push()函数在数组 ary 尾部追加元素 3,并返回数组长度。

第 4 行,用 pop()函数弹出数组 ary 尾部元素,并返回该元素。

第 6 行,用 shift()函数弹出数组 ary 头部元素,并返回该元素。

第 8 行,用 unshift()函数在数组 ary 头部追加两个元素－1 和 0,并返回长度。

第 12 行,用 splice()函数删除从下标 1 处开始的两个元素,并插入两个元素 22 和 33。

第 14 行,用 splice()函数删除从下标 1 处开始的两个元素。

第 17 行,用 slice()函数获取下标 2 到下标 4 之间的元素,注意下标范围半闭半开,因此不获取下标为 4 的元素,实际获取下标 2、3 所对应的元素。

执行结果为:

```
[ 0, 1, 2, 3 ] 4
[ 0, 1, 2 ] 3
[ 1, 2 ] 0
[ -1, 0, 1, 2 ] 4
[ 0, 22, 33, 3, 4, 5 ]
[ 0, 3, 4, 5 ]
[ 2, 3 ]
```

使用 reverse()和 sort()函数,可轻松实现对数组元素的倒序和排序功能。

【例 5-7】 数组元素的倒序和排序

```
1.    let arrRev = [2, 1, 0, 3].reverse()
2.    console.log(arrRev)                  //[ 3, 0, 1, 2 ]
3.    console.log( arrRev.sort() )         //[ 0, 1, 2, 3 ]
```

第 1 行,使用 reverse()函数将数组元素进行倒序,返回倒序的数组。

第 3 行,使用 sort()函数将数组元素进行由小到大正向排序,返回正向排序的数组。

执行结果为:

```
[ 3, 0, 1, 2 ]
[ 0, 1, 2, 3 ]
```


视频讲解

5.2　元组

2.1.4 节已经简单介绍过元组。元组也是一种有序的集合,但不同于数组,可用于存储不同类型的元素。此外,元组的长度是固定的,不可动态调整。

5.2.1 定义元组和赋值

在 TypeScript 中，对元组类型变量进行初始化或赋值时，需要匹配元组类型中指定项的个数和类型。

【例 5-8】 定义元组变量并赋值

```
1.    let person : [string, number, string]
2.    person = ['Ada',19,'F']
3.    //person = [19,'Ada','F']                    //类型不匹配
4.    //person = ['Ada',19]                        //数量不匹配
```

第 1 行，定义元组类型变量 person 它包含 3 个类型元素，依次为 string、number 和 string 元素。

第 2 行，赋值没有问题，元素数量和类型都符合定义的要求。

第 3 行，赋值与定义的类型不匹配，会报错：

```
Type 'number' is not assignable to type 'string'
```

第 4 行，赋值与定义的数量不匹配，会报错：

```
Type '[string, number]' is not assignable to type '[string, number, string]'.    Source has 2
element(s) but target requires 3.
```

定义元组的元素项类型时，可使用可选元素和剩余元素。

【例 5-9】 定义元组时使用可选元素

```
1.    type Point = [x:number, y?:number, z?:number]
2.    let p:Point = [1]
3.    let p2D:Point = [1,2]
4.    let p3D:Point = [1,2,3]
```

第 1 行，定义元组类型并赋别名 Point，注意后 2 个元素用问号？设置为可选元素。

第 2～4 行，声明 3 个 Point 类型的变量，并分别赋 1 个值、2 个值和 3 个值。因为 Point 定义了可选元素，所以没有语法问题。

【例 5-10】 定义元组时使用剩余元素

```
1.    type Team = [leaderName:string, ...otherNames:string[]]
2.    let team : Team
3.    team = ['张珊']
4.    team = ['张珊','李思']
5.    team = ['张珊','李思','王武']
```

第 1 行，定义元组类型别名 Team。其中 otherNames 参数定义时前面有 3 个点，为剩余元素。

第 3～5 行，分别赋予元组变量一个值和多个值，因为 otherNames 为剩余元素，所以没有语法问题。

5.2.2　元组常用操作

元组可被视作特殊的数组，因此元组的各种常见操作和数组类似，如通过下标访问元素，拼接元素，连接元组，遍历元素，搜索元素，获取区间元素等。

【例 5-11】 通过下标访问元组的元素

```
1.    let person:[string,number,string]
2.    person = ['Ada',19,'F']
3.    person[0] = 'Amanda'
4.    console.log(person[0])                    //Amanda
```

第 3 行，通过下标修改元组相应元素的值。

第 4 行，通过下标获取元组相应元素的值。

执行结果为：

```
Amanda
```

【例 5-12】 元组连接操作：join()的拼接结果为字符串，concat()用于拼接元素

```
1.    type Emp = [string, number, string]
2.    let ada : Emp = ['Ada', 18, 'F']
3.    console.log(ada.join())                   //Ada,18,F 默认用逗号连接
4.    let bob:Emp = ['Bob',19,'M']
5.    let p2 = ada.concat(bob)                  //拼接两个元组的元素
6.    console.log(p2)                           //[ 'Ada', 18, 'F', 'Bob', 19, 'M' ]
```

第 3 行，join()函数用于将指定元组中的元素拼接为字符串。默认用逗号进行连接，也可以指定连接的符号，比如，ada.join('/')，就是用斜杠符拼接元组中的元素。

第 5 行，concat()函数用于拼接多个元组的元素，并且以新元组形式返回。

执行结果为：

```
Ada,18,F
[ 'Ada', 18, 'F', 'Bob', 19, 'M' ]
```

【例 5-13】 用回调函数遍历元组中的元素

```
1.    type Emp = [string, number, string]
2.    let cindy:Emp = ['Cindy', 18, 'F']
3.    cindy.forEach(
4.         (value,index) => { console.log(`${index}: ${value}`) }
5.    )
```

执行结果为：

```
0: Cindy
1: 18
2: F
```

搜索元素出现的位置，可使用 indexOf()和 lastIndexOf()函数。其中，正向搜索用 indexOf()函数，反向搜索用 lastIndexOf()函数。

【例 5-14】 搜索元素出现的位置

```
1.    type Emp = [string,number,string]
2.    let cindy:Emp = ['Cindy',18,18,'F']
3.    console.log(cindy.indexOf(18))                    //1 正向搜索元素首次出现的位置
4.    console.log(cindy.lastIndexOf(18))                //2 反向搜索元素首次出现的位置
```

执行结果为：

```
1
2
```

【例 5-15】 获取元组区间元素和获得元组长度

```
1.    type Emp = [string, number, string]
2.    let cindy:Emp = ['Cindy', 18, 'F']
3.    let vals = cindy.slice(1, cindy.length)                    //[ 18, 'F' ]
4.    console.log(vals, vals.length);
```

第 3 行，用 slice()函数获取下标范围对应区间中的元素。注意，区间是半闭半开的，即不获取结束位置的元素。另外，length 是长度属性，用于获取元组中元素的个数。

执行结果为：

```
[ 18, 'F' ] 2
```

5.3 集合

Set 结构是一种存储唯一值的集合。它的元素是无序的，不能通过索引访问，但可以添加、删除和判断元素是否存在。

5.3.1 创建 Set 对象

TypeScript 使用 new 关键字加构造函数来创建 Set 类型对象，并可通过构造函数的参数初始化元素。

【例 5-16】 创建 Set 对象

```
1.    let set1 = new Set()
2.    let set2 = new Set([1,2,3])
```

第 1 行，用无参构造函数创建 Set 对象，后续可通过 add()函数向 Set 对象中添加元素，如下所示：

```
set1.add(1)
```

第 2 行，用带参构造函数创建 Set 对象。其中，参数值为数组，实际上也可使用其他可迭代类型数据，如元组、Set、字符串和 Map。

5.3.2　Set 类常用操作

Set 类常用的函数和属性有 add()、has()、delete()、clear()、size 等，如表 5-2 所示。

表 5-2　Set 类的常用函数和属性

函数和属性	描　　述
add()	向 Set 对象中添加元素
has()	判断 Set 对象中是否存在指定元素
delete()	删除 Set 对象中的元素
clear()	清空 Set 对象中元素
size	返回 Set 对象中元素个数

【例 5-17】 向 Set 对象中添加元素，程序会自动剥离重复值

```
1.    let nameSet = new Set()
2.    nameSet.add('Ada').add('Bob')
3.    nameSet.add('Cindy')
4.    nameSet.add('Cindy')                    //重复插入元素
5.    console.log(nameSet)                    //Set(3) { 'Ada', 'Bob', 'Cindy' }
```

第 2～4 行，向 Set 对象 nameSet 中添加元素。注意，add()函数可以链式调用。

在第 3 行和第 4 行添加了相同元素，Set 类型变量不允许重复值存在，因此重复值会自动被剥离，以确保集合中每个值都是唯一的。

执行结果如下：

```
Set(3) { 'Ada', 'Bob', 'Cindy' }
```

【例 5-18】 判断 Set 对象中是否存在指定元素

```
1.    let nameSet = new Set()
2.    nameSet.add('Ada').add('Bob')
3.    console.log(nameSet.has('Ada'))              //true
4.    console.log(nameSet.has('Bobbie'))           //false
```

第 3～4 行，分别判断 Set 对象 nameSet 中是否存在'Ada'和'Bobbie'元素。

执行结果如下：

```
true
false
```

【例 5-19】 删除、清空和返回 Set 的元素个数

```
1.    let nameSet = new Set(['Ada','Bob','Cindy'])
2.    nameSet.delete('Ada')
3.    console.log(nameSet.size)
4.    nameSet.clear()
5.    console.log(nameSet.size)
```

第 2 行，用 delete()函数删除 Set 对象中的'Ada'元素。
第 3 行，用 size 属性返回 Set 对象中的元素个数。
第 4 行，用 clear()函数清空 Set 对象中的元素。
执行结果如下：

```
2
0
```

【例 5-20】 遍历 Set 中的元素

```
1.    let scores = new Set([81,66,72])
2.    for(let value of scores){
3.         console.log(value)
4.    }
5.    scores.forEach(
6.         element => { console.log(element)
7.    })
```

第 2～4 行，用 for…of 语句遍历 Set 变量 scores 中的元素。
第 5～7 行，用 forEach 语句遍历 Set 变量 scores 中的元素。
执行结果如下：

```
81
66
72
81
66
72
```


视频讲解

5.4 映射

Map 结构是一种键值对(Key-Value)的集合。每个键都是唯一的，并且键与一个值相关联。可以通过键来访问对应的值，也可以添加、删除和判断键是否存在。注意，在 TypeScript

中，任何类型值都可以作为键或值。

5.4.1　创建 Map 对象

TypeScript 使用构造函数来创建 Map 类型对象，并可通过构造参数初始化键值对数据。

【例 5-21】 创建 Map 对象，并直接初始化键值对数据

```
1.    let books = new Map([
2.        ['9787302614616','软件工程导论与项目案例教程'],
3.        ['9787302615590','React 全栈式实战开发入门'],
4.        ['9787302610274','C#程序设计与编程案例'],
5.        ['9787302610274','大数据分析——预测建模与评价机制'],
6.    ])
7.    console.log(books)
```

7. console.log(books)

注意，当输入相同键时，会保留后面的键值对数据。第 4 行和第 5 行中的键相同，都为'9787302610274'，因此，第 4 行数据将被第 5 行数据替代。

执行结果如下：

```
Map(3) {
  '9787302614616' => '软件工程导论与项目案例教程',
  '9787302615590' => 'React 全栈式实战开发入门',
  '9787302610274' => '大数据分析——预测建模与评价机制'
}
```

5.4.2　Map 类的常用函数和属性

Map 类的常用函数和属性如表 5-3 所示。

表 5-3　Map 类的常用函数和属性

函数和属性	描　　述
set()	设置键值对数据，并返回该 Map 对象
get()	返回键对应的值，若不存在则返回 undefined
has()	判断 Map 中是否包含键对应的值，返回布尔值
delete()	删除 Map 中的键值对，返回布尔值
clear()	移除 Map 对象的所有键值对数据
size	返回 Map 对象中键值对的个数
keys()	返回 Iterator 对象，用于迭代 Map 中每个元素的键
values()	返回 Iterator 对象，用于迭代 Map 中每个元素的值
entries()	返回 Iterator 对象，用于迭代 Map 中每个键值对元素

【例 5-22】 Map 类的常见操作

```
1.    let emps = new Map([
2.         [1, {name:'ada', sex:'F', salary:3000}],
3.         [2, {name:'bob', sex:'M', salary:3500}],
4.    ])
5.    emps.set(2,{name:'Bobbie',sex:'F',salary:3300})        //设置 Map 键值对数据
6.    emps.set(3,{name:'Candy',sex:'F',salary:3500})
7.    console.log('emps:',emps)
8.    console.log(emps.get(2))                               //获取键对应值
9.    console.log(emps.has(3), emps.has(4))                  //判断是否包含键值
10.    emps.delete(1)                                        //删除键对应的元素
11.    console.log(emps.size)                                //元素个数
12.    emps.clear()                                          //清除所有元素
13.    console.log(emps.size)
```

第 1～4 行，用构造函数创建 Map 对象 emps，并初始化两个键值对元素。

第 5 行，用 set()函数设置键值对数据，因为键“2”存在，所以起到了替代作用。

第 6 行，用 set()函数设置键值对数据，因为键“3”不存在，所以起到了添加作用。

第 8 行，用 get()函数获取键“2”对应的值。

第 9 行，用 has()函数分别判断键“3”和键“4”对应的值是否存在。

第 11 行，用 size 属性获得键值对个数。

第 12 行，清除所有键值对数据。

执行结果为：

```
emps: Map(3) {
  1 => { name: 'ada', sex: 'F', salary: 3000 },
  2 => { name: 'Bobbie', sex: 'F', salary: 3300 },
  3 => { name: 'Candy', sex: 'F', salary: 3500 }
}
{ name: 'Bobbie', sex: 'F', salary: 3300 }
true false
2
0
```

【例 5-23】 遍历 Map 中的元素

```
1.    let emps = new Map([
2.         [1, {name:'ada', sex:'F', salary:3000}],
3.         [2, {name:'bob', sex:'M', salary:3500}],
4.    ])
5.    for(let key of emps.keys()){
6.         console.log(key)
7.    }
8.    for(let value of emps.values()){
```

```
9.          console.log(value)
10.    }
11.    for(let kv of emps.entries()){
12.          console.log(kv[0], kv[1])
13.    }
14.    for(let [key,value] of emps){
15.          console.log(key, value)
16.    }
17.    emps.forEach((value, key, map) =>{
18.          console.log(key,value, `共 ${map.size} 个元素`)
19.    })
```

第 1～4 行，创建 Map 对象 emps，并初始化两个键值对元素。其中键为数值类型，值为对象类型。

第 5～7 行，用 keys()函数返回 Iterator 对象，并用 for…of 语法迭代显示每个元素的键。

第 8～10 行，用 values()函数返回 Iterator 对象，并用 for…of 语法迭代显示每个元素的值。

第 11～13 行，用 entries()函数返回 Iterator 对象，并用 for…of 语法迭代显示每个元素。其中元素的 0 下标和 1 下标位置分别代表键和值。

第 14～16 行，使用对象解构方式获得 Map 的所有键和值，并用 for…of 语法迭代显示每个元素的键和值。

第 17～19 行，在 forEach()函数中使用回调函数，迭代显示 Map 中元素的键、值以及元素个数。

执行结果为：

```
{ name: 'ada', sex: 'F', salary: 3000 }
{ name: 'bob', sex: 'M', salary: 3500 }
1 { name: 'ada', sex: 'F', salary: 3000 }
2 { name: 'bob', sex: 'M', salary: 3500 }
1 { name: 'ada', sex: 'F', salary: 3000 }
2 { name: 'bob', sex: 'M', salary: 3500 }
1 { name: 'ada', sex: 'F', salary: 3000 } 共 2 个元素
2 { name: 'bob', sex: 'M', salary: 3500 } 共 2 个元素
```

注意，若编译时出现如下报错：

```
Cannot find name 'Map'.    Do you need to change your target library? Try changing the 'lib'
compiler option to 'es2015' or later.
```

可指定 ES 版本进行编译，如下所示：

```
tsc -t es2015 test.ts
```

5.5 不同集合类型间的转换

在 TypeScript 项目开发中,需掌握数组、Set 和 Map 三种类型之间的转换。

【例 5-24】 实现数组、Set 和 Map 三种类型之间的转换

```
1.    let arr : number[] = [1,2,3]
2.    let set = new Set<number>(arr)                    //数组转换为 Set
3.    arr = Array.from(set)                             //Set 转换为数组方式 1
4.    arr = [...set]                                    //Set 转换为数组方式 2
5.    let map : Map<number,string> = new Map([
6.         [1,'Ada'],[2,'Bob'],[3,'Cindy']
7.    ])
8.    arr = []
9.    for(let key of map.keys()){                       //Map 中的键(key)转换为数组
10.        arr.push(key)
11.   }
12.   let arrVal = []
13.   for(let val of map.values()){                     //Map 中的值(value)转换为数组
14.        arrVal.push(val)
15.   }
```

第 2 行,用 Set 构造函数将数组变量转换为 Set 类型变量。

第 3 行,用 Array 的 from()函数将 Set 类型变量转换为数组类型变量。

第 4 行,用[…Set 变量]语法将 Set 类型变量转换为数组类型变量。

第 9～11 行,将 Map 类型变量中的键(key)内容转换为数组类型变量。

第 13～15 行,将 Map 类型变量中的值(value)内容转换为数组类型变量。

5.6 实战闯关——集合

不同类型集合的结构特点各异,可分别应用于不同场景中。建议对数组、Set 和 Map 三种集合类型分别进行巩固练习。

【实战 5-1】 数组练习。

实践步骤:

(1) 将 72、66、81、99、66 这 5 个成绩数值保存到一个数组变量 scores 中。

(2) 将数组[65,71,80,98]与 scores 数组合并,并保存到数组变量 scores 中。

(3) 利用 Set 元素不重复的特点,剔除数组变量 scores 中的重复数据,并保存到数组变量 scores 中。

(4) 对数组变量 scores 中的数值元素进行由小到大排序,并保存到数组变量 scores 中。

(5) 对数组变量 scores 中的数值元素进行倒序处理,并保存到数组变量 scores 中。

【实战 5-2】 Set 练习

实践步骤:

(1) 将 72、54、81、100、66 这 5 个成绩数值保存到一个 Set 变量 scores 中。

（2）判断 Set 变量 scores 中是否含有满分值（即值为 100 的元素）。

（3）剔除低于 60 分的元素。

【实战 5-3】 Map 练习

针对通讯录中的好友信息：Ada，女，13701930685；Bob，男，13641949728；Cindy，女，13601632677。实践如下操作：

（1）将好友信息放入 Map 变量 friends，其中“键”存放姓名，“值”存放姓名、性别、联系手机号。

（2）在 Map 变量 friends 中添加新好友信息：Danny，男，15779366596。

（3）获取 Map 变量 friends 中 Bob 的“值”信息。

（4）删除 Map 变量 friends 中 Bob 的信息。

（5）显示 Map 变量 friends 中的所有“值”信息。