泛型

有些时候，函数返回值的类型与参数类型是相关的。

```
function getFirst(arr) {
  return arr[0];
}
```

上例中，函数 `getFirst()` 总是返回参数数组的第一个成员。参数数组成员是什么类型，返回值就是什么类型。

为了解决这个问题，TypeScript 就引入了“泛型”（generics）。泛型的特点就是带有“类型参数”（type parameter）。

```
function getFirst<T>(arr:T[]):T {
  return arr[0];
}
```

上例中，函数 `getFirst()` 的函数名后面尖括号的部分，就是类型参数，参数要放在一对尖括号（<>）里面。本例只有一个类型参数T，可以将其理解为类型声明需要的变量，需要在调用时传入具体的参数类型。

上例的函数 `getFirst()` 的参数类型是 `T[]`，返回值类型是 `T`，就清楚地表示了两者之间的关系。比如，输入的参数类型是 `number[]`，那么 `T` 的值就是 `number`，因此返回值类型也是 `number`。

函数调用时，需要提供类型参数。

```
getFirst<number>([1, 2, 3])
```

上例中，调用函数 `getFirst()` 时，需要在函数名后面使用尖括号，给出类型参数 `T` 的值，本例是 `<number>`。

不过为了方便，函数调用时，往往省略不写类型参数的值，让 TypeScript 自己推断。

```
getFirst([1, 2, 3])
```

上例中，TypeScript 会从实际参数 `[1, 2, 3]`，推断出类型参数 `T` 的值为 `number`。

有些复杂的使用场景，TypeScript 可能推断不出类型参数的值，这时就必须显式给出了。

```
function comb<T>(arr1:T[], arr2:T[]):T[] {
  return arr1.concat(arr2);
```

```
}
```

上例中，两个参数 `arr1`、`arr2` 和返回值都是同一个类型。如果不给出类型参数的值，下面的调用会报错。

```
comb([1, 2], ['a', 'b']) // Type 'string' is not assignable to type 'number'.
```

上面示例会报错，TypeScript 认为两个参数不是同一个类型。但是，如果类型参数是一个联合类型，就不会报错。

```
comb<number|string>([1, 2], ['a', 'b']) // 正确
```

上例中，类型参数是一个联合类型，使得两个参数都符合类型参数，就不报错了。这种情况下，类型参数是不能省略不写的。

类型参数的名字，可以随便取，但是必须为合法的标识符。习惯上，类型参数的第一个字符往往采用大写字母。一般会使用T（type 的第一个字母）作为类型参数的名字。如果有多个类型参数，则使用 T 后面的 U、V 等字母命名，各个参数之间使用逗号（“,”）分隔。

```
function map<T, U>(arr:T[], f:(arg:T) => U):U[] {
  return arr.map(f);
}

// 用法实例
map<string, number>(['1', '2', '3'], (n) => parseInt(n)); // 返回 [1, 2, 3]
```

上例将数组的实例方法 `map()` 改写成全局函数，它有两个类型参数 `T` 和 `U`。含义是，原始数组的类型为 `T[]`，对该数组的每个成员执行一个处理函数 `f`，将类型 `T` 转成类型 `U`，那么就会得到一个类型为 `U[]` 的数组。

泛型可以理解成一段类型逻辑，需要类型参数来表达。有了类型参数以后，可以在输入类型与输出类型之间，建立一一对应关系。