函数

函数的实际参数个数，可以少于类型指定的参数个数，但是不能多于，即 TypeScript 允许省略参数。

```
let myFunc: (a:number, b:number) => number;
myFunc = (a:number) => a; // 正确
myFunc = (a:number, b:number, c:number) => a + b + c; // 报错
```

上例中，变量 `myFunc` 的类型只能接受两个参数，如果被赋值为只有一个参数的函数，并不报错。但是，被赋值为有三个参数的函数，就会报错。

这是因为 JavaScript 函数在声明时往往有多余的参数，实际使用时可以只传入一部分参数。比如，数组的 `forEach()` 方法的参数是一个函数，该函数默认有三个参数 `(item, index, array) => void`，实际上往往只使用第一个参数 `(item) => void`。因此，TypeScript 允许函数传入的参数不足。

```
let x = (a:number) => 0;
let y = (b:number, s:string) => 0;

y = x; // 正确
x = y; // Target signature provides too few arguments. Expected 2 or more, but got
1.
```

```
interface myFn {
  (a:number, b:number): number;
}
var add:myFn = (a, b) => a + b;
```

上例中，`interface` 命令定义了接口 `myFn`，这个接口的类型就是一个用对象表示的函数。

1. Function 类型

TypeScript 提供 `Function` 类型表示函数，任何函数都属于这个类型。

```
function doSomething(f:Function) {
  return f(1, 2, 3);
}
```

上例中，参数 `f` 的类型就是 `Function`，代表这是一个函数。`Function` 类型的值都可以直接执行。`Function` 类型的函数可以接受任意数量的参数，每个参数的类型都是 `any`，返回值的类型也是 `any`，代表没有任何约束，**所以不建议使用 `Function` 这个类型，给出函数详细的类型声明会更好。**

```
type Person = { name: string };
const people = ['alice', 'bob', 'jan'].map((name):Person => ({name}));
```

上例中，Person 是一个类型别名，代表一个对象，该对象有属性 name。变量 people 是数组的 map() 方法的返回值。map() 方法的参数是一个箭头函数 (name):Person => ({name})，该箭头函数的参数 name 的类型省略了，因为可以从 map() 的类型定义推断出来，箭头函数的返回值类型为 Person。相应地，变量 people 的类型是 Person[]。

2. 可选参数

如果函数的某个参数可以省略，则在参数名后面加问号表示。

```
function f(x?:number) { /* */ }
f();   // OK
f(10); // OK
```

参数名带有问号，表示该参数的类型实际上是 原始类型|undefined，它有可能为 undefined 。比如，上例的 x 虽然类型声明为 number，但是实际上是 number|undefined。

```
function f(x?:number) {
  return x;
}
f(undefined); // 正确
```

上例中，参数 x 是可选的，等同于说 x 可以赋值为 undefined。但是，反过来就不成立，类型显式设为 undefined 的参数，就不能省略。

```
function f(x:number|undefined) {
  return x;
}
f(); // 报错
```

上例中，参数 x 的类型是 number|undefined，表示要么传入一个数值，要么传入 undefined，如果省略这个参数，就会报错。

函数的可选参数只能在参数列表的尾部，跟在必选参数的后面。

```
let myFunc: (a?:number, b:number) => number; // A required parameter cannot follow
an optional parameter. 必选参数不能跟在可选参数后面。
```

如果前部参数有可能为空，这时只能显式注明该参数类型可能为 undefined。

```
let myFunc:
  (
    a:number|undefined,
    b:number
  ) => number;
```

上例中，参数 a 有可能为空，就只能显式注明类型包括 undefined，传参时也要显式传入 undefined。函数体内部用到可选参数时，需要判断该参数是否为 undefined。

```
let myFunc: (a:number, b?:number) => number;

myFunc = function (x, y) {
  if (y === undefined) {
    return x;
  }
  return x + y;
}
```

上例中，由于函数的第二个参数为可选参数，所以函数体内部需要判断一下，该参数是否为空。

3. 参数默认值

设置了默认值的参数，就是可选的。如果不传入该参数，它就会等于默认值。

```
function createPoint(
  x:number = 0,
  y:number = 0
):[number, number] {
  return [x, y];
}
createPoint() // [0, 0]
```

上例中，参数 x 和 y 的默认值都是 0，调用 createPoint() 时，这两个参数都是可以省略的。这里其实可以省略 x 和 y 的类型声明，因为可以从默认值推断出来。

```
function createPoint(
  x = 0, y = 0
) {
  return [x, y];
}
```

可选参数与默认值不能同时使用。

```
function f(x?: number = 0) { // Parameter cannot have question mark and
initializer. 形参不能有问号和初始化值
  // ...
}
```

设有默认值的参数，如果传入 `undefined`，也会触发默认值。

```
function f(x = 456) {
  return x;
}
f(undefined) // 456
```

具有默认值的参数如果不位于参数列表的末尾，调用时不能省略，如果要触发默认值，必须显式传入 undefined。

```
function add(
  x:number = 0,
  y:number
) {
  return x + y;
}
add(1);            // 报错
add(undefined, 1); // 正确
```

4. readonly 只读参数

如果函数内部不能修改某个参数，可以在函数定义时，在参数类型前面加上 `readonly` 关键字，表示这是只读参数。

```
function arraySum(
  arr:readonly number[]
) {
  // ...
  arr[0] = 0; // 报错
}
```

上例中，参数 `arr` 的类型是 `readonly number[]`，表示为只读参数。如果函数体内部修改这个数组，就会报错。

5. void 类型

void 类型表示函数没有返回值。

```
function f():void {
  console.log('hello');
}
```

上例中，函数 f 没有返回值，类型就要写成 void。如果返回其他值，就会报错。

```
function f():void {
  return 123; // Type 'number' is not assignable to type 'void'.
}
```

void 类型允许返回 undefined。

```
function f():void {
  return undefined; // 正确
}
function f():void {
  return null; // Type 'null' is not assignable to type 'void'.
}
```

如果变量、对象方法、函数参数是一个返回值为 void 类型的函数，那么并不代表不能赋值为有返回值的函数。恰恰相反，该变量、对象方法和函数参数可以接受返回任意值的函数，这时并不会报错。

```
type voidFunc = () => void;
const f:voidFunc = () => {
  return 123;
};
```

上例中，变量 f 的类型是 voidFunc，是一个没有返回值的函数。但是实际上，f 的值可以是一个有返回值的函数（返回 123），编译时不会报错。

这是因为，这时 TypeScript 认为，这里的 void 类型只是表示该函数的返回值没有利用价值，或者说不应该使用该函数的返回值。只要不用到这里的返回值，就不会报错。

这样设计是有现实意义的。举例来说，数组方法 Array.prototype.forEach(fn) 的参数 fn 是一个函数，而且这个函数应该没有返回值，即返回值类型是 void。

但是，实际应用中，很多时候传入的函数是有返回值，但是它的返回值不重要，或者不产生作用。

```
const src = [1, 2, 3];
const ret = [];
src.forEach(el => ret.push(el));
```

上例中，`push()` 有返回值，表示新的数组长度。但是，对于 `forEach()` 方法来说，这个返回值是没有作用的，根本用不到，所以 TypeScript 不会报错。

如果后面使用了这个函数的返回值，就违反了约定，则会报错。

```
type voidFunc = () => void;
const f:voidFunc = () => {
  return 123;
};
f() * 2; // The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type. 算术运算的左边必须是“any”、“number”、“bigint”或枚举类型。
```

上例中，最后一行报错了，因为根据类型声明，**f()没有返回值，但是却用到了它的返回值，因此报错了。**

注意，这种情况仅限于变量、对象方法和函数参数，函数字面量如果声明了返回值是 void 类型，还是不能有返回值。

```
function f():void {
  return true; // 报错
}
const f3 = function ():void {
  return true; // 报错
};
```

上例中，函数字面量声明了返回 `void` 类型，这时只要有返回值（除了 `undefined` 和 `null`）就会报错。

函数的运行结果如果是抛错，也允许将返回值写成 `void`。

```
function throwErr():void {
  throw new Error('something wrong');
}
```

上例中，函数 `throwErr()` 会抛错，返回值类型写成 `void` 是允许的。

6. 局部类型

函数内部允许声明其他类型，该类型只在函数内部有效，称为局部类型。

```
function hello(txt:string) {
  type message = string;
  let newTxt:message = 'hello ' + txt;
  return newTxt;
}
const newTxt:message = hello('world'); // Cannot find name 'message'. Exported
```

```
variable 'newTxt' has or is using private name 'message'. 找不到名称“message”。 导出变量'newTxt'已经或正在使用私有名称'message'。
```

上例中，类型 `message` 是在函数 `hello()` 内部定义的，只能在函数内部使用。在函数外部使用，就会报错。

7. 高阶函数

一个函数的返回值还是一个函数，那么前一个函数就称为高阶函数（higher-order function）。

下面箭头函数返回的还是一个箭头函数：

```
(someValue: number) => (multiplier: number) => someValue * multiplier;
```

8. 函数重载

有些函数可以接受不同类型或不同个数的参数，并且根据参数的不同，会有不同的函数行为。这种根据参数类型不同，执行不同逻辑的行为，称为函数重载（function overload）。

```
reverse('abc');     // 'cba'
reverse([1, 2, 3]); // [3, 2, 1]
```

上例中，函数 `reverse()` 可以将参数颠倒输出。参数可以是字符串，也可以是数组。这意味着，该函数内部有处理字符串和数组的两套逻辑，**根据参数类型的不同，分别执行对应的逻辑，这就叫“函数重载”。**

```
function reverse(str:string):string;
function reverse(arr:any[]):any[];
```

上例中，分别对函数 `reverse()` 的两种参数情况，给予了类型声明。但是，到这里还没有结束，后面还必须对函数 `reverse()` 给予完整的类型声明。

```
function reverse(str:string):string;
function reverse(arr:any[]):any[];
function reverse(
  stringOrArray:string|any[]
):string|any[] {
  if (typeof stringOrArray === 'string')
    return stringOrArray.split('').reverse().join('');
  else
    return stringOrArray.slice().reverse();
}
```

上例中，前两行类型声明列举了重载的各种情况。第三行是函数本身的类型声明，它必须与前面已有的重载声明兼容。

有一些编程语言允许不同的函数参数，对应不同的函数实现。但是，JavaScript 函数只能有一个实现，必须在这个实现当中，处理不同的参数。因此，函数体内部就需要判断参数的类型及个数，并根据判断结果执行不同的操作。

```
function add(
  x:number,
  y:number
):number;
function add(
  x:any[],
  y:any[]
):any[];
function add(
  x:number|any[],
  y:number|any[]
):number|any[] {
  if (typeof x === 'number' && typeof y === 'number') {
    return x + y;
  } else if (Array.isArray(x) && Array.isArray(y)) {
    return [...x, ...y];
  }

  throw new Error('wrong parameters');
}
```

上例中，函数 `add()` 内部使用 `if` 代码块，分别处理参数的两种情况。 **重载的各个类型描述与函数的具体实现之间，不能有其他代码，否则报错。**

另外，虽然函数的具体实现里面，有完整的类型声明。但是，函数实际调用的类型，以前面的类型声明为准。比如，上例的函数实现，参数类型和返回值类型都是 `number|any[]` ，但不意味着参数类型为 `number` 时返回值类型为 `any[]`。

函数重载的每个类型声明之间，以及类型声明与函数实现的类型之间，不能有冲突。

```
// This overload signature is not compatible with its implementation signature.
此重载签名与其实现签名不兼容。
function fn(x:boolean):void;
function fn(x:string):void;
function fn(x:number|string) {
  console.log(x);
}
```

重载声明的排序很重要，因为 TypeScript 是按照顺序进行检查的，一旦发现符合某个类型声明，就不再往下检查了， **类型最宽的声明应该放在最后面，防止覆盖其他类型声明。**

```
function f(x:any):number;
function f(x:string): 0|1;
```

```
function f(x:any):any { /* */ }
const a:0|1 = f('hi'); // Type 'number' is not assignable to type '0 | 1'. 'x' is
declared but its value is never read. 类型'number'不能赋值给类型'0 | 1'。 声明
了'x'，但永远不会读取它的值。
```

上面声明中，第一行类型声明 `x:any` 范围最宽，导致函数 `f()` 的调用都会匹配这行声明，无法匹配第二行类型声明，所以最后一行调用就报错了，因为等号两侧类型不匹配，左侧类型是0|1，右侧类型是number。这个函数重载的正确顺序是，第二行类型声明放到第一行的位置。

对象的方法也可以使用重载。

```
class StringBuilder {
  #data = '';

  add(num:number): this;
  add(bool:boolean): this;
  add(str:string): this;
  add(value:any): this {
    this.#data += String(value);
    return this;
  }

  toString() {
    return this.#data;
  }
}
```

上例中，方法 `add()` 也使用了函数重载。

函数重载也可以用来精确描述函数参数与返回值之间的对应关系。

```
function createElement(
  tag:'a'
):HTMLAnchorElement;
function createElement(
  tag:'canvas'
):HTMLCanvasElement;
function createElement(
  tag:'table'
):HTMLTableElement;
function createElement(
  tag:string
):HTMLElement {
  // ...
}
```

上例中，函数重载精确描述了参数 `tag` 的三个值，所对应的不同的函数返回值。

这个示例的函数重载，也可以用对象表示。

```
type CreateElement = {
  (tag:'a'): HTMLAnchorElement;
  (tag:'canvas'): HTMLCanvasElement;
  (tag:'table'): HTMLTableElement;
  (tag:string): HTMLElement;
}
```

由于重载是一种比较复杂的类型声明方法，为了降低复杂性，一般来说，如果可以的话，应该优先使用联合类型替代函数重载。

```
// 写法一：  函数重载
function len(s:string):number;
function len(arr:any[]):number;
function len(x:any):number {
  return x.length;
}

// 写法二：联合类型
function len(x:any[]|string):number {
  return x.length;
}
```

上例中，写法二使用联合类型，要比写法一的函数重载简单很多。

9., 构造函数

JavaScript 语言使用构造函数，生成对象的实例。**构造函数的最大特点，就是必须使用 `new` 命令调用。**

```
const d = new Date();
```

上例中，`Date()` 就是一个构造函数，使用 `new` 命令调用，返回 `Date` 对象的实例。

构造函数的类型写法，就是在参数列表前面加上 `new` 命令。

```
class Animal {
  numLegs:number = 4;
}
type AnimalConstructor = new () => Animal;
function create(c:AnimalConstructor):Animal {
  return new c();
}
const a = create(Animal);
```

上面示例中，类型AnimalConstructor就是一个构造函数，而函数create()需要传入一个构造函数。 **在JavaScript 中，类（class）本质上是构造函数**，所以Animal这个类可以传入create()。

构造函数还有另一种类型写法，就是采用对象形式。

```
type F = { new (s:string): object; };
```

上例中，类型 `F` 就是一个构造函数。类型写成一个可执行对象的形式，并且在参数列表前面要加上 `new` 命令。

某些函数既是构造函数，又可以当作普通函数使用，比如 `Date()`。这时，类型声明可以写成下面这样。

```
type F = {
  new (s:string): object;
  (n?:number): number;
}
```

上例中，`F` 既可以当作普通函数执行，也可以当作构造函数使用。