1. 函数类型介绍

函数类型声明，需要在声明函数时，给出参数的类型和返回值的类型。

参数类型直接在参数后接冒号+ 类型；返回值类型在参数圆括号后接冒号+类型。

这里定义的参数类型是string，返回值的类型是void

```
function hello(txt:string):void {
  console.log('hello ' + txt);
}
```

如果不指定参数类型，ts会自动推断，如果推断不出来会推断为any类型。

返回值通常可以不写，ts会自己推断，但是有时候出于代码文档目的，返回值类型写上比较好。

1. 函数赋值为一个变量

如果变量被赋值为一个函数，变量的类型有两种写法。

第一种：根据函数类型推断出变量的类型。

第二种：使用箭头函数形式为变量指定类型，参数类型在箭头左侧，返回值类型在箭头右侧。

第二种写法需要注意的是：

- 函数的参数一定要写在圆括号内，不然会报错。
- 参数名必须写，不能只写类型，否则会把类型当做参数名，any类型

```
// 写法一
const hello = function (txt:string) {
  console.log('hello ' + txt);
}

// 写法二
const hello: (txt:string) => void = function (txt) {
  console.log('hello ' + txt);
};
```

给变量定义类型时，函数类型里面的参数名与实际参数名，可以不一致。

如果一个函数类型定义很长，可以使用type命令定义一个别名。

```
type MyFunc = (txt:string) => void;

const hello:MyFunc = function (txt) {
  console.log('hello ' + txt);
};
```

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，也就是ts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

```

如果一个变量要嵌套另一个函数类型，有个技巧就是使用typeof运算符。

这是一个很有用的技巧，任何需要类型的地方，都可以使用`typeof`运算符从一个值获取类型。

```
function add(
  x:number,
  y:number
) {
  return x + y;
}

const myAdd:typeof add = function (x, y) {
  return x + y;
}
```

函数类型还可以采用对象的写法。

这种写法的函数参数与返回值之间，间隔符是冒号`:`，而不是正常写法的箭头`=>`，因为这里采用的是对象类型的写法，对象的属性名与属性值之间使用的是冒号。

这种写法平时很少用，但是非常合适用在一个场合：**函数本身存在属性。**

```
let add:{
  (x:number, y:number):number
};

add = function (x, y) {
  return x + y;
};


function f(x:number) {
  console.log(x);
}
f.version = '1.0';
// 函数f()本身还有一个属性version。这时，f完全就是一个对象，类型就要使用对象的写法。

let foo: {
  (x:number): void;
  version: string
} = f;
```

函数类型也可以使用 Interface 来声明，这种写法就是对象写法的翻版

```ts
interface myfn {
  (a:number, b:number): number;
}

var add:myfn = (a, b) => a + b;
```

2. Function类型

ts中提供Function类型表示函数，任何函数属于这个类型。

这里参数`f`的类型就是`Function`，代表这是一个函数。

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

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，所以不建议使用这个类型，给出函数详细的类型声明会更好。

3. 箭头函数

首先箭头函数是普通函数的一种简化写法，它的类型写法和普通写法类似。**参数类型写在参数名的后面，返回值类型写在参数列表的圆括号后面。**

```ts
const repeat = (
  str:string,
  times:number
):string => str.repeat(times);
```

当一个函数A的一个参数是函数B，给参数B定义类型，使用的箭头函数形式定义该参数B，**此时参数列表外的圆括号外的类型是函数A的返回值类型，B的返回值类型在参数圆括号内存。**

```ts
function greet(
  fn:(a:string) => void  // void是参数fn的返回值类型
):void { // void是函数greet返回值类型
  fn('world');
}
```

4. 可选参数

如果函数的某个参数后加上问号，代表该参数可以省略。

```
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`。

但是如果显式定义为undefined则**调用时**就不能省略了。

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

f() // 报错   这里必须显式写上undefined
```

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，否则会报错。

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

如果说参数在前面但是有可能为空，那就得显式定义为undefined，并且为空时，也要传入该参数为undefined才行。

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;
}
```

5. 参数默认值

如果设置默认值，就默认是可选参数，不传该参数，就会等于默认值。只有当参数为undefined时才会触发默认值。

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

createPoint() // [0, 0]
```

可选参数与默认值不能同时使用。

```
// 报错
function f(x?: number = 0) {
  // ...
}
```

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
```

6. 参数解构

函数参数如果存在变量解构，类型写法如下。

```
function f(
  [x, y]: [number, number]
) {
  // ...
}

function sum(
  { a, b, c }: {
     a: number;
     b: number;
     c: number
  }
) {
  console.log(a + b + c);
}

```

参数解构可以结合类型别名（type 命令）一起使用，代码会看起来简洁一些。

```
type ABC = { a:number; b:number; c:number };

function sum({ a, b, c }:ABC) {
  console.log(a + b + c);
}
```

7. rest剩余参数

rest参数表示函数剩余所有的参数，它可以是数组（剩余参数类型相同），也可以是元组（剩余参数类型不同）

```
// rest 参数为数组
function joinNumbers(...nums:number[]) {
  // ...
}

// rest 参数为元组
function f(...args:[boolean, number]) {
  // ...
}

```

元组需要声明每一个剩余参数的类型。如果元组里面的参数是可选的，则要使用可选参数。

```
function f(
  ...args: [boolean, string?]
) {}
```

rest 参数甚至可以嵌套。

```
function f(...args:[boolean, ...string[]]) {
  // ...
}
```

rest 参数可以与变量解构结合使用。

```
function repeat(
  ...[str, times]: [string, number]
):string {
  return str.repeat(times);
}

// 等同于
function repeat(
  str: string,
  times: number
):string {
  return str.repeat(times);
}
```

8. 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，在参数类型加上readonly关键字，表示这个参数是只读的。

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

9. void类型

void类型表示函数没有返回值，如果是函数字面量有返回值就会报错。

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

void 类型允许返回 `undefined` 或 `null`

如果打开了 `strictNullChecks` 编译选项，那么 void 类型只允许返回 `undefined`。如果返回 `null`，就会报错。这是因为 JavaScript 规定，如果函数没有返回值，就等同于返回 `undefined`。

```
// 打开编译选项 strictNullChecks

function f():void {
  return undefined; // 正确
}

function f():void {
  return null; // 报错
}
```

问题

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，它可以接受返回任意值的函数，只要是你不对结果再有任何处理。**

解释：因为有时候传入的函数是有返回值，但是不会产生作用就没有事情，也就不会报错

例子：

push方法有返回值，返回插入新元素后的数组长度，但是他没有任何作用

```
const src = [1, 2, 3];
const ret = [];

src.forEach(el => ret.push(el));
```

如果一个函数的运行结果如果是抛出错误，可以将返回值写成 `void`。

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

10. never类型

never类型表示肯定不会出现的值，它用在函数的返回值，表示某个函数肯定不会返回值，即函数不会正常执行结束。函数没有执行结果，不可能有返回值

应用

- 抛出错误的函数

只有抛出错误才是never类型，如果是return一个Error对象，则返回值是Error类型。

抛出错误的情况属于never、void类型，所以从返回值类型中不知道，抛出的是哪一种错误。

```
function fail(msg:string):never {
  throw new Error(msg);
}
```

- 无限执行的函数

这里while语句，判断条件一直是true，程序会一直循环，不会停止。

```
const sing = function():never {
  while (true) {
    console.log('sing');
  }
};
```

如果一个函数抛出了异常或者陷入死循环，那么该函数就无法正常返回一个值，这个函数的返回类型就是never。

而如果程序调用了一个返回值类型为never的函数，就意味着程序会在该函数的调用位置终止，永远不会执行后续的代码。

一个函数如果某些条件下有正常返回值，另一些条件下抛出错误，这时它的返回值类型可以省略`never`。

never类型和void类型区别：never类型表示函数没有执行结束，不可能返回值；void类型表示函数正常执行结束，但是不返回值或者说返回undefined。

11. 局部类型

函数内部声明其他类型，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'); // 报错
```

12. 高阶函数

一个函数的返回值还是一个函数，该函数就被称为高阶函数。

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

13. 函数重载

有的函数可以接受不同类型或者不同个数的参数，可能参数类型的不同，会有不同的逻辑，就称为函数重载。对象的方法也能重载。

ts对于声明**函数重载**类型，要逐一定义每一种情况的类型，然后再定义完整的类型声明。

函数reverse有两种参数情况，都声明了，并且还有一个完整的类型声明，在完整的类型声明里面根据类型的不同处理不同逻辑的代码。

```
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();
}
```

虽然函数的具体实现里面，有一个完整的类型声明，但是在函数实际调用类型时，要以最前面的类型声明为准。

重载声明的排序很重要，因为ts是按照顺序进行检查的，只要发现符合某个类型，就不再往下检查了，所以应该将类型最宽的声明放在最后面，防止覆盖其他类型声明。

在一一定义每一种类型声明时，可以用对象表示，这样更简洁

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

14. 构造函数

第一种：就是在参数列表前加上new命令。

```
type AnimalConstructor = new () => Animal;

```

第二种：采用对象形式

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

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;
}

```

上面示例中，F 既可以当作普通函数执行，也可以当作构造函数使用。