第一章：TypeScript 基础

学习目标

- 理解 TypeScript 的语言特点
- 熟练掌握 TypeScript的安装与调试方法
- 掌握 TypeScript 的基础类型、接口、 类、函数T、泛型和高级类型

一、语言简介

1.TypeScript 是什么？

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open source.

- ypeScript 是 JavaScript 的类型的超集，它可以编译成纯 JavaScript。
- 编译出来的 JavaScript 可以运行在任何浏览器上。
- TypeScript 编译工具可以运行在任何服务器和任何系统上。
- TypeScript 是开源的。

2.为什么是TypeScript ？

TypeScript 增加了代码的可读性和可维护性

- 类型系统实际上是最好的文档，大部分的函数看看类型的定义就可以知道如何使用；
- 可以在编译阶段就发现大部分错误，这总比在运行时候出错好；
- 增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等；

TypeScript 包容性

- .js 文件可以直接重命名为 .ts 文件
- 自动类型推论
- 可以定义从简单到复杂的几乎一切类型
- 即使 TypeScript 编译报错，也可以生成 JavaScript 文件
- 兼容JavaScript第三方库

TypeScript 拥有活跃的社区

- 大部分第三方库都有提供给 TypeScript 的类型定义文件
- Google Angular2 使用 TypeScript 编写
- Vue.js、React 也很好的支持 TypeScript
- TypeScript 拥抱了 ES6 规范，支持部分 ESNext 草案规范
- TypeScript 开发维护团队有微软、谷歌等大厂支持

3.TypeScript 注意事项

- 需要理解接口（Interfaces）、泛型（Generics）、类（Classes）、枚举类型（Enums）等前端工程师可不熟悉的概念；
- 学习成本较高，但会大大降低维护成本；
- 集成到构建流程需要一些工作量；
- 可能和一些库结合的不是很完美；

4.学习资源

- 官方网站： http://www.typescriptlang.org/
- 中文网站： https://www.tslang.cn/
- TypeScript 中文手册 https://typescript.bootcss.com
- 《TypeScript 入门教程》： https://ts.xcatliu.com/

二、安装与调试

1.TypeScript 官网介绍

- 官网 http://www.typescriptlang.org/
- 中文非官网 https://www.tslang.cn/

2.TypeScript 环境安装

- 安装依赖环境 Node.js : https://nodejs.org/en/download/
- 安装 TypeScript: npm install -g typescript
- 安装集成开发环境 IDE: Visual Studio Code , Sublime Text , WebStorm

3.TypeScript 代码调试

- 命令行工具 tsc
- VS Code 中调试代码
- tsconfig.json 项目配置文件

三、基础语法

1.变量

```
var [变量名] : [类型] = 值;
var [变量名] = 值;
var [变量名] : [类型];
```

```
const hello : string   = "Hello World!";
const score : number   = 100;
let    arr   : string[]  = ['abc', 'def', 'ghi'];
let   name             = 'Jack';
let   alias : string;
```

2.模块

一个文件就是一个模块，模块可以导出供其它模块，也可以导入其它模块来使用。

```
// person.ts
export class Person {
    name: string;
    constructor( name: string){
        this.name = name;
    }
    sayhi():void{
        console.log(`Hi, I'am ${this.name}`);
    }
}
```

```
// test.ts
import mod = require('./person');
const person = new mod.Person("Jack");
person.sayhi(); // => Hi, I'am Jack
```

3.函数

箭头定义函数与 function 定义的区别：

- 简化语法
- 自动绑定定义时的 this 上下文环境。

```
// 普通函数
const sayhi = function(name: string): string{
    return `hi, ${name}`;
}
// 箭头函数
const sayhi1 = (name: string): string => {
    return `hi, ${name}`;
}
// 箭头函数简写
const sayhi2 = (name: string): string => `hi, ${name}`
const result = sayhi2('Jack');
console.log(result); // => hi, Jack
```

4.语句和表达式

每一条语句以分号结束，一行中只有一条语句时分号可省略；

表达式有返回值，单值表达式返回自身，复杂表达式返回运算结果。

```
console.log("Baidu")
console.log("Google");
console.log("Baidu");console.log("Google");
// 表达式
let age : number = 18;
age++;
age = age < 20 ? 20 : age;
if(age > 0){
    console.log(`I'am ${age} years old.`)
}
```

5.注释

注释语法分为单行注释和多行注释

```
// 单行注释
/**
 * 多行
 * 注释
 * ...
 */
```

四、基础类型应用

1.基础类型

- **布尔类型（Boolean）, 可以在声明时直接初始化，也可以接受表达式的结果；**

```
// 布尔类型： true/false
let isDone: boolean = false;
let isRight = 1 < 2;
let allRight = 1 > 0 && 3 < 2;
```

- **数值类型（Number）, 支持多种进制数值**；

```
// 数值类型
let decLiteral: number = 6;        // 十进制数
let hexLiteral: number = 0xf00d;  // 十六进制数
let binaryLiteral: number = 0b1010; // 二进制数
let octalLiteral: number = 0o744;   // 八进制数
```

- **字符串类型（String）**

```
// 字符串类型
let myname: string = "Jack";
let years: number = 5;
let words: string = `Hi, I am ${myname}, ${years} years old.`;
```

- **Any类型, 可以被赋予任何类型的值**；

```
// Any类型，可被赋予任何类型的值
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false;
```

- **Null 与 Undefined 类型， 它们是其他任何类型（包括 void）的子类型，可以赋值给其它类型。**

  Null类型表示空值， Undefined类型，表示未被赋值。

```
// Null类型，表示空值
let n: null = null;
let x: string = 'xxxx';
x = n;   // OK
// Undefined类型，表示未被赋值
let u: undefined = undefined;
x = u;  // OK
```

- **Void类型, 表示无类型；**

```
// Void类型，表示无类型
function warnUser(): void {
    console.log("This is my warning message");
}
```

- **Never类型： never类型表示的是那些永不存在的值的类型，是任何类型的子类型。**

（1）可以赋值给任何类型；

（2）没有类型是never的子类型或可以赋值给never类型（除了never本身之外）；

（3）即使 any也不可以赋值给never；

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
    throw new Error(message);
}
// 推断的返回值类型为never
function fail() {
    return error("Something failed");
}
```

2.变量声明

变量命名规则

- 可以包含数字和字母；
- 除了下划线 _ 和美元 $ 符号外，不能包含其他特殊字符，包括空格；
- 变量名不能以数字开头；

```
var [变量名] : [类型] = 值; // 声明变量指定类型并赋值
var [变量名] : [类型];      // 声明变量指定类型, 值为undefined
var [变量名] = 值;         // 声明变量并赋值, 编译器自动类型推断
var [变量名];           // 声明变量, 值为undefined, 未指定类型即为Any

var name : string = 'Jack';
var name : string;
var name = 'Jack';
var name;
```

声明关键字 var 与 let/const

- var 关键字存在如变量提升、可多次重复声明、for循环中变量绑定异常等问题。
- let 关键字支持块级作用域可避免 var 的这些问题。
- 使用最小特权原则，所有变量除了你计划去修改的都应该使用 const。

```
// const 实例
const  a: string = "aaaaa";
a = "AAAAA";  // Error, 不可改变
const o = {name:'Jack'};
o = {name:"Tom"}; // Error, 不可改变
o.name = "Tom";  // OK, 可以改变
```

for 循环中变量绑定异常问题

```
// 同级作用域内，变量被覆盖
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 1000);
}
// 解决方法：使用立即执行的函数表达式
for (var i = 0; i < 10; i++) {
    (function(i) {
        setTimeout(function() { console.log(i); }, 1000);
       })(i);
}
// let块级作用域，不存在问题
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() {console.log(i); }, 1000);
}
```

变量作用域，变量有多种级别的作用域：

- 全局作用域
- 类作用域
- 函数作用域
- 块级作用域

```
let global_num = 12          // 全局作用域变量
class Numbers {
   num_val = 13;             // 类实例变量
   static sval = 10;         // 类静态变量
   storeNum(num:number):void {
      let local_num = 14;    // 函数作用域变量
      if(num > 0){
        let xx = 99;         // 块级作用域变量
      }
   }
}
```

3.运算符

```
算术运算符：+  -  *  /  %  ++  --
关系运算符：==  !=  >  <  >=  <=
逻辑运算符：&&  ||  !
赋值运算符：=  +=  -=  *=  /=
位运算符：    &  |  ~  ^  <<  >>  >>>
三元运算符：Test ? expr1 : expr2
类型运算符：typeof  instanceof
负号运算符：-
字符串运算符：+
```

```
// 赋值运算符
var a: number = 12
var b: number = 10
a = b   // 直接赋值
a += b  // 先加再赋值
a -= b  // 先减再赋值
a *= b  // 先乘再赋值
// 取模运行符
var res = 10 % 3; // 1
// 短路运算符
var a = 10
var result = ( a<10 && a>5) // false
var result2 = ( a>5 || a<10) // true
// 三元运算符
var num:number = -2
var result3 = num > 0 ? "> 0" : "< 0，or = 0" // < 0，or = 0
// typeof 运算符
var num = 12
console.log(typeof num);   //输出结果: number
// 负号运算符
var x:number = 4
var y = -x; // y = -4
```

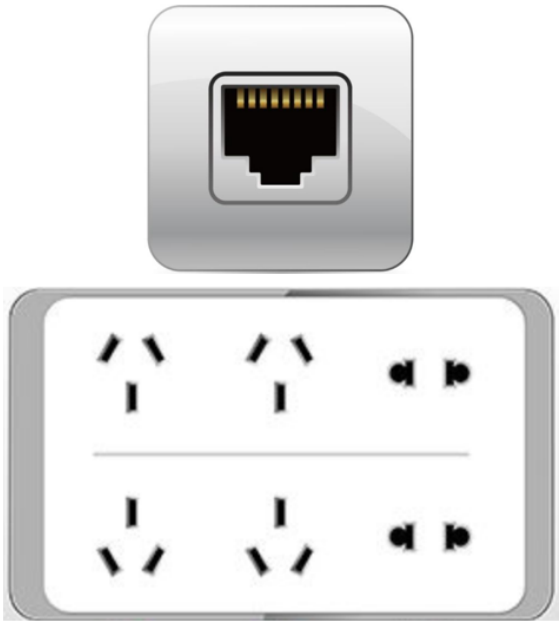
4.循环

TypeScript 中for...in、for...of、forEach、while、do...while等多种循环遍历实例代码：

```
for ( let i=0; i< 10; i++ ){...}
//for...in 语句用于一组值的集合或列表进行迭代输出

let list = [1,2,3];
for ( let i in list ){
    let item = list[i];
    ...
}

let someArray = [1, "string", false];
for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}

// forEach
let someArray = [1, "string", false];
someArray.forEach((val, index, array)=>{...})
// while
let a = 0;
while(a < 10){
    ...
    a++;
}
// do...while
let a = 0;
do{
    ...
    a++;
}while(a < 10)
```

五、接口

1.接口是什么？



2.为什么要使用接口？

TypeScript 的核心原则之一是对值所具有的结构进行类型检查。

接口的作用：为这些类型命名和为你的代码或第三方代码定义契约。

简单地说：

（1）接口的作用就是去描述结构的形态；

（2）对指定接口类型的值进行约束；

（3）方便对同一接口类型的值进行统一处理

（4）通过继承简化代码结构，提高代码复用性

3.接口的使用方法

接口通过关键字 interface 定义，接口中的属性可以是可选的、只读的、带其它任何属性的，接口也可以是可索

引和函数类型的；

```
// 接口定义
interface IPerson2 {
    firstName:string,
    lastName?:string,       // 可选属性
    readonly age: number,  // 只读属性
    [propName: string]: any // 其它任意属性
    sayHi: ()=>string
}
interface StringArray {     // 可索引接口
    [index: number]: string;
}
interface SearchFunc {      // 函数类型接口
    (source: string, subString: string): boolean;
```

```
}
```

接口可作为一种类型应用于变量、参数和函数返回值的类型；

```
// 接口定义
interface IPerson {
    firstName:string,
    lastName:string,
    sayHi: ()=>string
}
// 接口实例
var customer:IPerson = {
    firstName:"Tom",
    lastName:"Hanks",
    sayHi: ():string =>{return "Hi there"}
}
```

接口为可索引类型与函数类型实例；

```
// 可索引类型接口
interface StringArray1 {
    [index: number]: string; // index 支持 string 和 number 类型
}
let myArray1: StringArray1;
myArray1 = ["Bob", "Fred"];
let myStr1: string = myArray[0];

// 函数类型接口
interface SearchFunc {
  (source: string, subString: string): boolean;
}
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    return result > -1;
}
```

定义为某一接口类型的变量、参数或返回值，在编译时会进行类型合法性检查；

```
// 接口定义
interface IPerson {
    firstName:string,
    lastName:string,
    sayHi: ()=>string
}
// 接口实例
var customer:IPerson = {
    //firstName:"Tom", //注释此行，会报错提示缺少 firstName 属性
    lastName:"Hanks",
    sayHi: ():string =>{return "Hi there"}
}
console.log("Customer 对象 ")
console.log(customer.firstName)
console.log(customer.lastName)
```

4.接口的继承

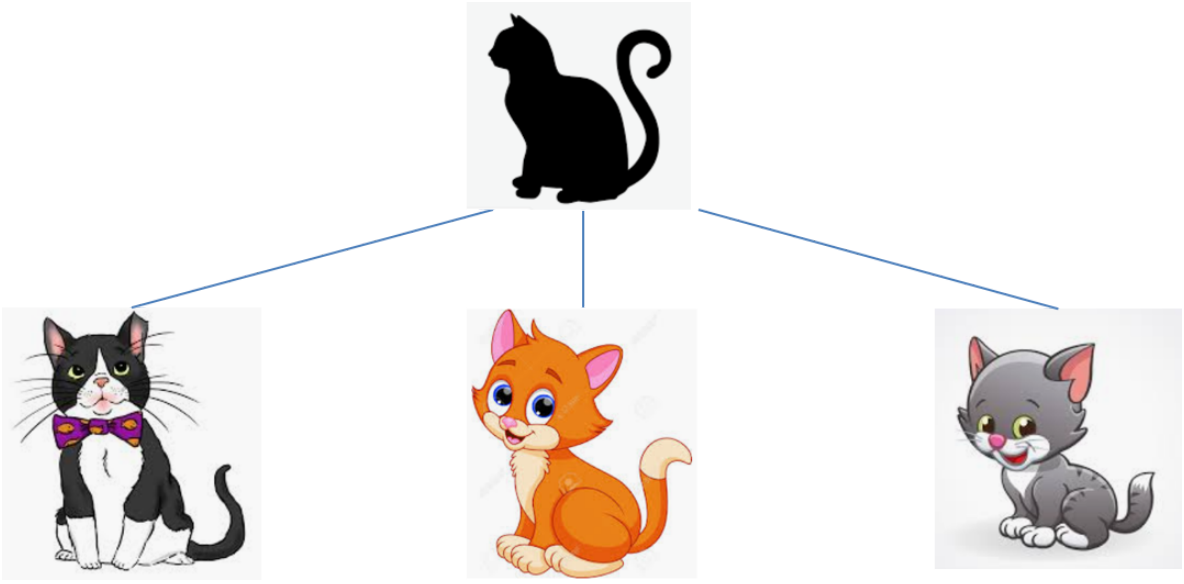
TypeScript 中接口可以继承其它接口，且可以同时继承多个接口

```
// 定义接口Person
interface Person {
    age:number
}
// 定义接口Musician，继承自 Person
interface Musician extends Person {
    instrument:string
}
// 应用接口
var drummer = <Musician>{};
drummer.age = 27
drummer.instrument = "Drums"
```

```
// 接口多继承
interface Shape {
    color: string;
}
interface PenStroke {
    penWidth: number;
}
interface Square extends Shape, PenStroke {
    sideLength: number;
}
let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

六、类

1.类的概念



类是现实世界或思维世界中的实体在计算机中的反映，它将数据以及这些数据上的操作封装在一起。

（1）类和对象是面向对象编程技术中的最基本的概念；

（2）类是一种抽象的数据类型；

（3）对象是对客观事物的抽象，类是对对象的抽象；

（4）类是对象的模板，对象是类的实例；

（5）类描述了所创建的对象共同的属性和方法。

2.类的定义

类通过关键字 class 定义，可以有构造函数、属性和方法；

```
// 类的定义
class Car {
    // 属性
    name:string;
    // 构造函数
    constructor(name:string) {
        this.name = name
    }
    // 方法
    desc():void {
        console.log(`I am a ${this.name} car.`)
    }
}
```

3.类的应用

通过关键字 new 创建类的实例（对象），每个类的实例拥有独立的属性和方法；

```
// 创建Car类的实例
let suv = new Car('SUV')
console.log(suv.name);
// => SUV
console.log(suv.desc())
// => I am a SUV car.

let sedan = new Car('Sedan')
console.log(sedan.name);
// => Sedan
console.log(sedan.desc())
// => I am a Sedan car.
```

一个类可以通过关键字 extends 来继承另一个类，被继承的类称为父类，继承的类称为子类；

子类就会拥有父类的属性和方法；实现不同层次的抽象，以达到简化设计和代码复用的目的。

```
// 类的继承
class Animal1 {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}
class Dog extends Animal1 {
    bark() {
        console.log('Woof! Woof!');
    }
```

```
}
const dog = new Dog();
dog.bark();
dog.move(10);
```

修饰符的作用：指明属性或方法的访问级别，public（默认）为公有，可以在任何地方被访问。

```
// public （默认）修饰符
class Animal {
    public name: string
    public constructor(theName: string) {
  this.name = theName;
    }
    public getName() {
        return this.name;
    }
}
let fish = new Animal('fish');
fish.name;     // 通点操作符直接访问
fish.getName(); // 通过点操作符直接访问
```

Private 修饰符表明为私有，只能在其自身内部被访问。

```
// private 修饰符
class Animal {
    private name: string
    public constructor(theName: string) { this.name = theName; }
    public getName() {
        return this.name;
    }
}
let fish = new Animal('fish');
fish.name;     // 异常：不能在外部被访问
fish.getName(); // 通过点操作符可访问
```

Protected 修饰符表明为受保护的，可以被其自身内部以及其子类内部访问。

```
// protected 修饰符
class Animal {
    protected name: string
    public constructor(theName: string) { this.name = theName; }
    public getName() { return this.name; }
}
class Fish extends Animal {
    public getName(){ return this.name; }
}

let fish1 = new Animal('fish');
let fish2 = new Fish('fish');
fish1.name;     // 异常：不能在外部访问
fish2.getName(); // 子类内部可访问
```

readonly 修饰符表明为只读的，只读属性必须在声明时或构造函数里被初始化。

```
// readonly 修饰符
class Animal {
    readonly name: string
    public constructor(theName: string) { this.name = theName; }
    public getName() {
        return this.name;
    }
}

let fish = new Animal('fish');
fish.name = 'xxx';     // 异常：只读属性不能被修改
```

static 修饰符表明为静态成员（属性和方法），只能通过类点操作符直接访问。

```
// static 修饰符
class Animal2 {
    static category: string = 'animal';
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    static getCategory() {
        return 'I am an ' + this.category;
    }
}
Animal2.category;       // 静态属性，通过类的点操作符直接访问
Animal2.getCategory(); // 静态方法，通过类的点操作符直接访问
```

Abstract 修饰符表明为抽象的类，其内部有抽象方法，必须由其子类（派生类）实现抽象的方法。

```
// 抽象类定义
abstract class Animal3 {
    abstract makeSound(): void;
    move(): void {
        console.log('move...');
    }
}
// 继承抽象类，必须要实现其抽象方法
class Cat2 extends Animal3 {
    makeSound(): void {
        console.log('miao...');
    }
}
let cat = new Cat2();
cat.move();
cat.makeSound();
```

4.类与接口

类可以通过关键字 implements 来实现某个接口，类的定义必须要满足接口的约束。

```
// 接口与类
interface ILoan {
    interest:number
}
class HouseLoan implements ILoan {
    interest:number
```

```
    rebate:number
    constructor(interest:number,rebate:number) {
        this.interest = interest
        this.rebate = rebate
    }
}
let loan1: ILoan = new HouseLoan(10,1);
console.log(loan1.interest);
let loan2: HouseLoan = new HouseLoan(10,2);
console.log(loan2.interest);
```

七、函数

1.函数定义与调用

函数主要是用来定义行为，TypeScript 为 JavaScript 函数添加了额外的功能，让我们可以更容易地使用，定义

函数的两种方法：

```
// 命名函数
function add(x, y) {
    return x + y;
}
// 匿名函数
let myAdd = function(x, y) {
    return x + y;
}
let sum1 = add(1,2);
let sum2 = myAdd(1,3);
```

2.函数类型

在 TypeScript 中可以指定函数的参数与返回值的类型来约束函数的使用；

```
// 命名函数
function add(x: number, y: number): number {
    return x + y;
}
// 匿名函数
let myAdd = function(x: number, y: number): number {
    return x + y;
};
let sum1 = add(1,2);
let sum2 = myAdd(1,3);
console.log(sum1);
console.log(sum2);
```

在 TypeScript 中函数定义的完整写法和使用类型推断的简化写法对比；

```
// 完整写法
let myAdd1: (baseValue: number, increment: number) => number = function(x: numbe
r, y: number): number {
    return x + y;
}
// 类型推断
let myAdd2 = function(x: number, y: number): number {
    return x + y;
}
// 类型推断
let myAdd3: (baseValue: number, increment: number) => number = function(x, y) {
    return x + y;
}
```

3.函数参数

在 TypeScript 中函数定义的参数默认情况下是必须提供的，缺少或过多都会编译异常；

```
// 函数参数
function getFullName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}
let result1 = getFullName(“Jack”);          // error, 参数太少
let result2 = getFullName(“Jack”, "Bill", "Sr.");  // error, 参数太多
let result3 = getFullName(“Jack”, “Bill”);  // 正确
```

在 TypeScript 中函数定义的参数使用 ？ 可定义为可选，可选参数必须跟在必须参数后面。

```
// 函数可选参数
function getFullName(firstName: string, lastName?: string) {
    return firstName + " " + (lastName || '');
}
let result1 = getFullName("Jack"); // 正确,lastName为可选参数
let result2 = getFullName(“Jack”, “Bill”, “Sr.”);   // error, 参数太多
let result3 = getFullName("Jack", "Bill"); // 正确
```

在TypeScript 中函数定义的参数可以设置默认值，带默认值的参数不需要放在必须参数的后面；

带默认值的参数出现在必须参数前面，则必须明确的传入 undefined 值来获得默认值。

```
// 函数参数默认值
function getFullName(firstName: string, lastName: string = 'Smith') {
    return firstName + " " + lastName;
}
let result1 = getFullName("Jack");             // 正确，lastName 有默认值
let result3 = getFullName("Jack", "Bill");     // 正确
```

```
// 如果有默认值的参数在必须参数前
function getFullName(firstName: string = 'Smith', lastName: string ) {
    return firstName + " " + lastName;
}
let result1 = getFullName("Jack");           // 错误，缺少 lastName 参数
let result2 = getFullName(undefined,"Jack"); // 正确， "Smith Jack"
```

在 TypeScript 中函数定义的参数中，对于数量不确定的参数可以使用省略号（... 变量名 ）的方式，来接收剩余

的参数集合。

```
// 函数剩余其它参数
// 扩展运算符
[...'hello'] // [ "h", "e", "l", "l", "o" ]
var arr1 = ['a', 'b'];
var arr2 = ['c'];
[...arr1, ...arr2] // [ 'a', 'b', 'c']
// 取代 apply 方法
Math.max.apply(null, [14, 3, 77]) // ES5 的写法
Math.max(...[14, 3, 77]) // ES6 的写法

// 函数剩余其它参数
function getFullName(firstName: string, lastName: string, ...otherNames: string[
]) {
    return firstName + " " + lastName + " " + otherNames.join(" ");
}
let result1 = getFullName("Jack", "Bill", "Smith", "Mick"); // 正确，lastName 有默
认值
let result2 = getFullName("Jack", "Bill");                  // 正确
let result3 = getFullName(...["Jack", "Bill"]);             // 正确
```

4.箭头函数

Lambda 函数也称之为箭头函数；箭头函数表达式的语法比函数表达式更短。

函数只有一行语句时，可省略 {} 和 return；单个参数时 () 是可选的。

```
// 箭头函数，Lambda 函数
var foo = x => 10 + x
console.log(foo(100))      //输出结果为 110

var foo1 = (x:number)=> {
    return 10 + x
}
console.log(foo1(100))      //输出结果为 110
```

箭头函数与普通函数最大的区别：

它能绑定函数创建时的 this 值，而普通函数绑定的是调用时的 this。

```
// 箭头函数，Lambda 函数
var person = {
    name: 'Jack',
    greet: function(){
        return (function(){
            return `Hi, I am ${this.name}`;  // 输出 "Hi, I am undefined"
        })()
    }
}
var person1 = {
    name: 'Jack',
```

```
    greet: function(){
        return (()=>{
            return `Hi, I am ${this.name}`;  // 输出 "Hi, I am Jack"
        })()
    }
}
```

5.函数重载

JavaScript 中可以根据不同的参数类型返回不同类型的值，在 TypeScript 中可以定义多种不同参数类型组合的函

数声明；在定义重载的时候，一定要把最精确的定义放在最前面。

```
// 函数重载
function add1 (a: number, b: number): number;
function add1 (a: string, b: string): string;
function add1 (a: any, b: any): any{
    if(typeof a == 'number'){
        return a + b;
    }
    if(typeof a == 'string'){
        return '' + (parseInt(a) + parseInt(b));
    }
};
console.log(add1(1,2));
console.log(add1('2', '3'));
console.log(add1('2'));       // 异常，没有匹配的参数列表
```

八、泛型

1.泛型概念

泛型代表的是可能出现的任何一种类型；可使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。

这样就可以以自己的数据类型来使用组件。

举例：

```
// 泛型
function identity(arg: number): number {
    return arg;
}
function identity(arg: any): any {
    return arg;
}
function identity<T>(arg: T): T {
    return arg;
}
```

2.泛型变量

我们把泛型变量T当做类型的一部分使用，而不是整个类型，增加了灵活性。

```
// 泛型变量
function outputIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error T 没有属性length
    return arg;
}
function outputIdentity1<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array 有length属性
    return arg;
}
function outputIdentity2<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array 有length属性
    return arg;
}
let output = outputIdentity1<string>(["id111","id222"]);
let output2 = outputIdentity2<string>(["id333","id444"]);
let output3 = outputIdentity2(["id555","id666"]);  //使用类型推断
```

3.泛型类型

泛型函数的类型与非泛型函数的类型没什么不同，只是有一个类型参数在最前面，像函数声明一样

```
// 泛型类型
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity: <T>(arg: T) => T = identity;
let myIdentity2: <U>(arg: U) => U = identity;
```

4.泛型接口

泛型也可以应用在函数型接口定义之中，例如：

```
// 泛型接口
interface GenericIdentityFn {
    <T>(arg: T): T;
}
function identity3<T>(arg: T): T {
    return arg;
}
let myIdentity3: GenericIdentityFn = identity3;
```

5.泛型类

泛型类看上去与泛型接口差不多。 泛型类使用（ <> ）括起泛型类型，跟在类名后面。

```
// 泛型类
class CreateNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}
let myCreateNumber = new CreateNumber<number>();
myCreateNumber.zeroValue = 0;
myCreateNumber.add = function(x, y) { return x + y; };
myCreateNumber.add(1, 2);
myCreateNumber.add('1', '2'); // error, 只接受数值型参数
```

6.泛型约束

可以定义一个接口来描述约束条件。

创建一个包含 length 属性的接口，使用 extends 关键字和这个接口来实现对类型的约束。

```
// 泛型约束
function identity4<T>(arg: T): T {
    console.log(arg.length);  // 此处不能保证 length 属性存在
    return arg;
}
interface ILength {
    length: number;
}
function identity5<T extends ILength>(arg: T): T {
    console.log(arg.length);  // 此处可保证 length 属性存在
    return arg;
}
let arr1 = identity5<number>(122112); // 错误：number 没有length 属性
let arr2 = identity5<string>('122112'); // 正确
```

九、高级类型

1.Number类型

通过 new Number() 创建引用（对象）类型数值变量，而 Number() 创建字面量（值）类型数值变量。

```
// Number 对象类型
let n1 = new Number(99);
let n2 = Number(99);
let n3 = new Number('sss');
console.log(typeof n1);      // object
console.log(typeof n2);      // number
console.log(n3, typeof n3);  // [Number: NaN] object
```

Number 对象属性：

constructor	返回对创建此对象的 Number 函数的引用。
MAX_VALUE	可表示的最大的数。
MIN_VALUE	可表示的最小的数。
NaN	非数字值
NEGATIVE_INFINITY	负无穷大，溢出时返回该值。
POSITIVE_INFINITY	正无穷大，溢出时返回该值。
prototype	可以向对象添加属性和方法。

Number 对象方法：

toString	把数字转换为字符串，使用指定的基数。
toLocaleString	把数字转换为字符串，使用本地数字格式顺序。
toFixed	把数字转换为字符串，结果的小数点后有指定位数的数字。
toExponential	把对象的值转换为指数计数法。
toPrecision	把数字格式化为指定的长度。
valueOf	返回一个 Number 对象的基本数字值。

Number 对象 toString() 方法实例

```
// Number 对象 toString 方法
let n = new Number('99');
console.log(n.toString(2));   // 1100011
console.log(n.toString(8));   //143
console.log(n.toString(16));  // 63
```

Number 对象 toLocaleString() 方法实例, 输出不同格式数字字符串。

```
// Number 对象 toLocaleString 方法
let n1 = new Number('123456.789');
console.log(n1.toLocaleString());  // 默认英文格式
// => 123,456.789
console.log(‘de-
DE’, { style: ‘currency’, currency: ‘EUR’ }));  // 欧元格式
// => € 123,456.79
console.log(n1.toLocaleString('zh-
CN', { style: 'currency', currency: 'CNY' })); // 人民币格式
// => CN¥ 123,456.79
// 自定义格式
console.log(n1.toLocaleString(undefined, {useGrouping:false, maximumFractionDigi
ts: 2}));
// => 123456.79
```

Number 对象 toPrecision() 方法输出指定长度精度，valueOf() 方法返回原始数字值。

```
// Number 对象 toPrecision 方法
let n4 = new Number('123456.78912');
console.log(n4.toPrecision(7));  // 格式化指定长度精度,四舍五入
// => 123456.8
console.log(n4.toPrecision(4));
// => 1.235e+5

// Number 对象 valueOf 方法
console.log(n4.valueOf()); // 返回原始数字值
// => 123456.78912
```

```
ES6 新增方法
Number.isFinite(2)          // 判断是否有限
Number.isNaN(0)             // 判断是否为NaN
Number.parseInt('1')        // 字符串类型转整型
Number.parseFloat('1')      // 字符串类型转浮点型
Number.isInteger(1.1)       // 判断是否为整型
Number.isSafeInteger(1.1)   // 判断是否为安全范围内的整型
Number.MAX_SAFE_INTEGER     // 最大安全整型值
Number.MIN_SAFE_INTEGER     // 最小安全整型值
```

通过 new String() 创建引用（对象）类型字符串变量，而 String() 创建字面量（值）类型字符串变量。

```
// String 类型
let s1 = new String("我爱中国");
let s2 = String("我爱中国");
let s3 = "我爱中国";
console.log(typeof s1);  // object
console.log(typeof s2);  // string
console.log(typeof s3);  // string
```

2.String类型

String 对象属性：

constructor	对创建该对象的函数的引用
length	字符串的长度
prototype	允许向对象添加属性和方法

```
String 对象方法：
charAt()            // 返回指定位置字符
charCodeAt()        // 返回指定的位置字符的 Unicode 编码
concat()            // 连接两个或更多字符串，并返回新的字符串
indexOf()           // 返回某个指定的字符串值在字符串中首次出现的位置
lastIndexOf()       // 从后向前搜索字符串，返回字符串最后出现的位置
match()             // 查找找到一个或多个正则表达式的匹配
replace()           // 替换与正则表达式匹配的子串
search()            // 检索与正则表达式相匹配的值
slice()             // 提取字符串的片断，并在新的字符串中返回被提取的部分
split()             // 把字符串分割为子字符串数组
substr()            // 从起始索引号提取字符串中指定数目的字符
substring()         // 提取字符串中两个指定的索引号之间的字符
toLowerCase()       // 转换为小写
toUpperCase()       // 转换为大写
```

String 类型 indexOf() 方法实例：

```
// String 类型 indexOf 方法
let str = '我爱你，中国';
let index = str.indexOf('，');
console.log(`逗号出现的位置是第 ${index + 1} 个`);
// => "逗号出现的位置是第 4 个"
let index2 = str.indexOf('.'); // 注意：未找到时返回值为 -1
console.log(`点号出现的位置是第 ${index2 != -1 ? index2 + 1: index2 } 个`);
// => "点号出现的位置是第 -1 个"
```

String 类型 concat() 方法实例，用于字符串的连接操作：

```
// String 类型 concat 方法
let str = "我爱你";
str = str.concat('，','中','国'); // 可传多个参数，返回一个新字符串
console.log(str);
// => "我爱你，中国"
```

String 类型 match() 方法用于查询匹配指定规则的字符串，replace() 方法用于替换指定规则的子字符串。

```
// String 类型 match 方法
let str = "我爱你,中国,中国,我爱你";
let result  = str.match(/中国/g);
console.log(`matched: ${result}`);
// => "matched: 中国,中国"

// String 类型 replace 方法
let str2 = "我爱你,中国,中国,我爱你";
let result2  = str2.replace(/中国/g,"China");
console.log(result2);
// => "我爱你,China,China,我爱你"
```

String 类型 search() 方法用于查找指定字符串的所在位置，slice() 方法用于根据位置截取子字符串。

```
// String 类型 search 方法
let str3 = "我爱你,China";
let index  = str3.search(/china/i);
console.log(`China 的位置在第 ${index} 个`);
// => "China 的位置在第 4 个"

// String 类型 slice 方法
let str4 = "我爱你,China";
let r1  = str4.slice(0,3);
let r2  = str4.slice(-5);
console.log(`${r1},${r2}`);
// => "我爱你,China"
```

String 类型 split() 方法用于分割字符串的为子字符串数组。

```
// String 类型 split 方法
let str5 = “我爱你中国”;
let arr  = str5.split('');
console.log(arr);
// => [ '我', '爱', '你', '中', '国' ]
let str6  = arr.join('-');
console.log(str6);
// => 我-爱-你-中-国
let str7  = str6.split('-').join(' . ');
console.log(str7);
// => 我 . 爱 . 你 . 中 . 国
```

String 类型 substr() 和 substring() 方法都是用于截取字符串的子串，区别在于第2个参数的意义。

```
// String 类型 substr 和 substring 方法
let str8 = "我爱你中国”;
// 第二个参数是指取多少个字符
let str9 = str8.substr(3,2);
// 第二个参数是指取子串结束位置
let str10 = str8.substring(3,5);
console.log(str9); // => 中国
console.log(str10); // => 中国
```

Date 类型实例表达的是时间中的某个时刻。

Date 对象基于 Unix Time Stamp，即自1970年1月1日（UTC）起经过的毫秒数。

实例化方法如下:

```
// Date 类型, 实例化Date对象
new Date();
new Date(value);
new Date(dateString);
new Date(year, monthIndex [, day [, hours [, minutes [, seconds [, milliseconds]
]]]]);
```

注意：

参数monthIndex 是从“0”开始计算的，这就意味着一月份为“0”，十二月份为“11”。

dateString 2019-10-09T09:57:21.106Z, Wed Oct 09 2019 17:57:21 GMT+0800 (GMT+08:00)
dateString 遵

循 IETF-compliant RFC 2822 timestamps 或 version of ISO8601

3.Date类型

Date 类型包含的类方法

```
// Date 类方法
Date.now()   // 返回自 1970-1-1 00:00:00  UTC（世界标准时间）至今所经过的毫秒数。
Date.parse() // 解析一个表示日期的字符串，并返回从 1970-1-1 00:00:00 所经过的毫秒数。
Date.UTC()   // 接受和构造函数最长形式的参数相同的参数并返回从 1970-01-
01 00:00:00 UTC 开始所经过的毫秒数。
```

Date 类型实例方法

```
Date.prototype.getDate() // 根据本地时间返回指定日期对象的月份中的第几天（1-31）
Date.prototype.getDay()  // 根据本地时间返回指定日期对象的星期中的第几天（0-6）
Date.prototype.getFullYear() // 根据本地时间返回指定日期对象的年份（四位数年份时返回四位数
字）
Date.prototype.getHours()    // 根据本地时间返回指定日期对象的小时（0-23）
Date.prototype.getMilliseconds() // 根据本地时间返回指定日期对象的毫秒（0-999）
Date.prototype.getMinutes()  // 根据本地时间返回指定日期对象的分钟（0-59）
Date.prototype.getMonth()    // 根据本地时间返回指定日期对象的月份（0-11）
Date.prototype.getSeconds()  // 根据本地时间返回指定日期对象的秒数（0-59）
Date.prototype.getTime()     // 返回从1970-1-1 00:00:00 UTC 到该日期经过的毫秒数
Date.prototype.getTimezoneOffset() // 返回当前时区的时区偏移
Date.prototype.getUTCDate()  // 根据世界时返回特定日期对象一个月的第几天（1-31）
Date.prototype.getUTCDay()   // 根据世界时返回特定日期对象一个星期的第几天（0-6）
Date.prototype.getUTCFullYear() // 根据世界时返回特定日期对象所在的年份（4位数）
Date.prototype.getUTCHours()   // 根据世界时返回特定日期对象当前的小时（0-23）
Date.prototype.getUTCMilliseconds() // 根据世界时返回特定日期对象的毫秒数（0-999）
Date.prototype.getUTCMinutes() // 根据世界时返回特定日期对象的分钟数（0-59）
Date.prototype.getUTCMonth()   // 根据世界时返回特定日期对象的月份（0-11）
Date.prototype.getUTCSeconds() // 根据世界时返回特定日期对象的秒数（0-59）
Date.prototype.getYear()       // 根据特定日期返回年份 (通常 2-3 位数) 使
用 getFullYear()
```

Date 类型实例格式转换方法

```
Date.prototype.toDateString() // 以人类易读（human-readable）的形式返回该日期对象日期部分的字符串
Date.prototype.toISOString()  // 把一个日期转换为符合 ISO 8601 扩展格式的字符串
Date.prototype.toJSON()       // 使用 toISOString() 返回一个表示该日期的字符串
Date.prototype.toLocaleDateString() // 返回一个表示该日期对象日期部分的字符串，该字符串格式与系统设置的地区关联
Date.prototype.toLocaleFormat()     // 使用格式字符串将日期转换为字符串
Date.prototype.toLocaleString()     // 返回一个表示该日期对象的字符串，该字符串与系统设置的地区关联
Date.prototype.toLocaleTimeString() // 返回一个表示该日期对象时间部分的字符串，该字符串格式与系统设置的地区关联
Date.prototype.toSource()     // 返回一个与Date等价的原始字符串对象，你可以使用这个值去生成一个新的对象
Date.prototype.toString()     // 返回一个表示该日期对象的字符串
Date.prototype.toTimeString() // 以人类易读格式返回日期对象时间部分的字符串
Date.prototype.toUTCString()  // 把一个日期对象转换为一个以UTC时区计时的字符串
Date.prototype.valueOf()      //返回一个日期对象的原始值
```

Date 类型实例代码

```
var today = new Date();
var birthday1 = new Date('December 17, 1995 03:24:00');
var birthday2 = new Date('1995-12-17T03:24:00');
var birthday3 = new Date(1995, 11, 17);
var birthday4 = new Date(1995, 11, 17, 3, 24, 0);
var birthday5 = new Date(today.getTime());
var birthday = new Date(Date.now());
console.log(birthday);
// => 2019-10-09T09:57:21.106Z
console.log(birthday.toString());
// => Wed Oct 09 2019 17:57:21 GMT+0800 (GMT+08:00)
console.log(new Intl.DateTimeFormat('en-US',
{ timeZone: "America/New_York"}).format(birthday));
// => 10/10/2019
console.log(new Intl.DateTimeFormat('zh-CN',
{ timeZone: "Asia/Shanghai"}).format(birthday));
// => 2019-10-10
```

4.RegExp类型

RegExp 构造函数创建了一个正则表达式对象，用于将文本与一个模式匹配。

语法如下：

```
/pattern/flags
new RegExp(pattern [, flags])
RegExp(pattern [, flags])
flags: i = 忽略大小写, m = 多行匹配, g = 全局匹配
```

有两种方法来创建一个 RegExp 对象：一是字面量、二是构造函数。

字面量方式的参数不使用引号，而构造函数的参数需要使用引号。

```
// RegExp 类型实例
/ab+c/i;
new RegExp('ab+c', 'i');
new RegExp(/ab+c/, 'i');
```

正则表达式中特殊字符的含义（一）：

```
.      匹配任意单个字符
\d     匹配任意阿拉伯数字。等价于[0-9]
\D     匹配任意一个不是阿拉伯数字的字符。等价于[^0-9]
\w     匹配任意字母数字字符，还包括下划线。等价于 [A-Za-z0-9_]
\W     匹配任意不是字母数字下划线的字符。等价于 [^A-Za-z0-9_]
\s     匹配一个空白符，包括空格、制表符、换页符、换行符和其他 Unicode 空
\t     匹配一个水平制表符（tab）
\r     匹配一个回车符
\n     匹配一个换行符
```

正则表达式中特殊字符的含义（二）：

```
(x)     匹配 x 并且捕获匹配项
[xyz]   匹配一个字符集合，匹配集合中的任意一个字符
[^xyz]  匹配任意不在括号内的字符
[0-9]   匹配0到9区间内的任意一个字符
[a-z]   匹配a到z区间内的任意一个字符
^       匹配输入开始
$       匹配输入结尾
```

正则表达式中特殊字符的含义（三）：

```
x*      匹配前面的模式 x 0 或多次
x+      匹配前面的模式 x 1 或多次
x?      匹配前面的模式 x 0 或 1 次
x|y     匹配 x 或 y
x{n}    n 是一个正整数。前面的模式 x 连续出现 n 次时匹配
x{n,}   n 是一个正整数。前面的模式 x 连续出现至少 n 次时匹配
x{n,m}  n 和 m 为正整数。前面的模式 x 连续出现至少 n 次，至多 m 次时匹配
```

正则表达式实例代码：

```
// RegExp 类型实例
let re = /(\w+)\s(\w+)/;
let str = "John Smith";
let newstr = str.replace(re, "$2, $1");
console.log(newstr); // => "Smith, John"•let result = /^\d{4}-\d{2}-
\d{2}$/.test('2019-10-01'); // => true
let result2 = /[a-z]{4,5}/i.test('32_Abc333sss'); // => false•const text = "你
好，China 中国";
let regex = /[\u4E00-\u9FA5\uF900-\uFA2D]{2}/g;
let match = regex.exec(text);
console.log(match[0]);  // => [ '你好', index: 0, input: '你好，China 中国' ]
let match2 = regex.exec(text);
console.log(match2[0]);  // => [ '中国', index: 9, input: '你好，China 中国' ]
```

5.Array类型

Array 类型即数组类型，表示一组相同类型元素的有序集合。

数组类型的语法如下：

```
// Array 类型语法
let array_name[:datatype];        //声明
array_name = [val1,val2,valn...]   //初始化
// 在声明时初始化
let array_name[:datatype] = [val1,val2,...]
var array_name[:datatype] = new Array([size])
var array_name[:datatype] = new Array(val1,val2,...)
```

Array 类型是通过字面量方式和 new Array() 方式创建实例，通过下标（索引）方式访问与赋值。

```
// Array 类型实例
var sites:string[];
sites = ["Apple","Banana","Pear"]
console.log(sites[0]);
console.log(sites[1]);
var nums:number[] = [101,102,103]
console.log(nums[0]);
console.log(nums[1]);
console.log(nums[2]);
var nums:number[] = new Array(3)
nums[0] = 1;
nums[1] = 2;
nums[2] = 3;
```

Array 类型实例解构与迭代代码：

```
// 数组解构
let arr:number[] = [12,13]
let [x,y] = arr // 将数组的两个元素赋值给变量 x 和 y
console.log(x)
console.log(y)
// 数组迭代
var j:any;
var nums:number[] = [1001,1002,1003,1004]
for(j in nums) {
    console.log(nums[j])
}
```

Array 类型二维数组实例：

```
// 二维数组
var multi:number[][] = [[1,2,3],[23,24,25]]
console.log(multi[0][0])
console.log(multi[0][1])
console.log(multi[0][2])
console.log(multi[1][0])
console.log(multi[1][1])
console.log(multi[1][2])
```

Array 类型实例方法列表：

```
concat()   // 连接两个或更多的数组，并返回结果
every()    // 检测数值元素的每个元素是否都符合条件
some()     // 检测数组元素中是否有元素符合指定条件
forEach()  // 数组每个元素都执行一次回调函数
indexOf()  // 搜索数组中的元素，并返回它所在的位置
lastIndexOf() // 返回一个指定的字符串值最后出现的位置
join()     // 把数组的所有元素放入一个字符串
map()      // 通过指定函数处理数组的每个元素，并返回处理后的数组
filter()   // 检测数值元素，并返回符合条件所有元素的数组
pop()      // 删除数组的最后一个元素并返回删除的元素
push()     // 向数组的末尾添加一个或更多元素，并返回新的长度
reduce()   // 将数组元素计算为一个值（从左到右）
reduceRight() // 将数组元素计算为一个值（从右到左）
reverse()  // 反转数组的元素顺序
shift()    // 删除并返回数组的第一个元素
unshift()  // 向数组的开头添加一个或更多元素，并返回新的长度
```

Array 类型实例代码：

```
// Array 类型实例
let arr: number[] = [1,2,3];
arr.concat(4,5,6);      // => [1,2,3,4,5,6]
arr.every(i => i > 0)   // => true
arr.some(i => i < 2)    // => true
arr.filter(i => i <= 2) // => [1,2]
arr.map(i => i + 1)     // => [2,3,4]
arr.pop()               // => 3
arr.pop()               // => 2
arr.push(2)             // [1,2]
arr.push(3)             // [1,2,3]
arr.unshift(0)          // [0,1,2,3]
arr.shift()             // [1,2,3]
arr.slice(0,2)          // => [1,2]
arr.slice(-2)           // => [2,3]
arr.splice(3,0,4,5,6)   // [1,2,3,4,5,6]
arr.sort((a,b)=> b - a) // [6,5,4,3,2,1]
arr.reverse()           // [1,2,3,4,5,6]
```

6.元组类型

元组与数组的区别：

元组中的元素可以是不同类型，而数组只能是相同类型的，除此之外，其它属性与方法与数组一致。

```
// 元组类型定义
let myTuple1: [number, string, boolean] = [1, 'Poplar', true]; // 正确
let myTuple2: [number, string, boolean, any] = [1, 'Poplar', true]; // 长度不符
let myTuple3: [number, string] = [1, 'Poplar', true]; // 长度不符
let myTuple4: [string, number, string] = [1, 'Poplar', true]; // 类型不符
let myTuple5 = [1, 'Poplar', true];
// 类型推断,等同 let myTuple5: (string | number | boolean)[]
let myTuple6 = new Array(1,"ts",false); // 错误：数组中元素类型必须相同
let myTuple7:(string | number | boolean)[] = [1,“ts”,false,2,true,“js”]; // 正
确，长度没有限制
```

7.Set类型

Set 类型对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

```
// Set 类型实例化语法
new Set([iterable]);

// Set 类型实例化
let set1 = new Set();
let set2 = new Set([1,2,3]);
let set3 = new Set([1,2,3,2,1]);
console.log(set1); // => Set {}
console.log(set2); // => Set { 1, 2, 3 }
console.log(set3); // => Set { 1, 2, 3 }
```

Set 类型实例的属性与方法：

```
// Set 类型属性与方法
Set.prototype.size        // 返回Set对象的值的个数
Set.prototype.add(value)  // 在Set对象尾部添加一个元素。返回该Set对象。
Set.prototype.clear()     // 移除Set对象内的所有元素。
Set.prototype.delete(value)  // 移除Set的中与这个值相等的元素
Set.prototype.entries()      // 返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值的数组。
Set.prototype.forEach(callbackFn[, thisArg]) // 按照插入顺序，为Set对象中的每一个值调用一次callBackFn。
Set.prototype.has(value)     // 返回一个布尔值，表示该值在Set中存在与否。
Set.prototype.keys()    // 返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值。
Set.prototype.values()  // 返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值。
```

Set 类型与数组类型互相转换，举例：

```
// Array 与 Set 互转
let myArr = [1, 2, 3];
// 创建Set实例
let mySet = new Set(myArr); // Set { 1, 2, 3 }
// 解构为数组
[...mySet];        // [1, 2, 3]
// 返回数组
Array.from(mySet); // [1, 2, 3]
```

字符串转Set 类型，利用 Set 为数组去重，举例：

```
// 字符串转 Set
var text = 'CHINA';
var mySet = new Set(text);  // Set {'C', 'H', 'I', 'N', 'A'}
mySet.size;  // 5
// 数组去重
const numbers = [1,2,2,3,3,3,4,5,4,5,1];
console.log([...new Set(numbers)]);
// [1, 2, 3, 4, 5]
```

8.Map类型

Objects 和 maps 的比较：

- Object的键只能是字符串或者 Symbols，但 Map 的键可以是任意值，包括函数、对象、基本类型。
- Map 中的键值是有序的，而添加到 Object 对象中的键则不是。因此，当对它进行遍历时，Map 对象是按插入的顺序返回键值。
- 可以通过 size 属性直接获取一个 Map 的键值对个数，而 Object 的键值对个数只能手动计算。
- Map 可直接进行迭代，而 Object 的迭代需要先获取它的键数组，然后再进行迭代。
- Map 在涉及频繁增删键值对的场景下会有些性能优势。

Map 类型实例化语法：

```
new Map([iterable])

let map = new Map([['key1','value1'],['key2','value2']])
```

Map 类型实例属性与方法：

```
// Map 类型实例的属性和方法
Map.prototype.size        // 元素数量
Map.prototype.clear()     // 移除Map对象的所有键/值对 。
Map.prototype.delete(key) // 如果 Map 对象中存在该元素，则移除它并返回 true；否则如果该元素不存在则返回 false
Map.prototype.entries()   // 返回一个新的 Iterator 对象，它按插入顺序包含了Map对象中每个元素的 [key, value] 数组。
Map.prototype.forEach(callbackFn[, thisArg]) // 按插入顺序，为 Map对象里的每一键值对调用一次callbackFn函数。
Map.prototype.get(key)    // 返回键对应的值，如果不存在，则返回undefined。
Map.prototype.has(key)    // 返回一个布尔值，表示Map实例是否包含键对应的值。
Map.prototype.keys()      // 返回一个新的 Iterator对象， 它按插入顺序包含了Map对象中每个元素的键 。
Map.prototype.set(key, value)  // 设置Map对象中键的值。返回该Map对象。
Map.prototype.values()    // 返回一个新的Iterator对象，它按插入顺序包含了Map对象中每个元素的值 。
```

Map 类型实例属性与方法：

```
// Map 实例
let myMap = new Map();
let keyObj = {},
    keyFunc = function () {},
    keyString = "a string";
// 添加键
myMap.set(keyString, "和键'a string'关联的值");
myMap.set(keyObj, "和键keyObj关联的值");
myMap.set(keyFunc, "和键keyFunc关联的值");
myMap.size; // 3
// 读取值
myMap.get(keyString);    // "和键'a string'关联的值"
myMap.get(keyObj);       // "和键keyObj关联的值"
myMap.get(keyFunc);      // "和键keyFunc关联的值"
myMap.get("a string");   // "和键'a string'关联的值",因为keyString === 'a string'
myMap.get({});           // undefined, 因为keyObj !== {}
myMap.get(function() {}) // undefined, 因为keyFunc !== function () {}
```

Map 类型迭代，举例：

```
// 迭代 Map
let myMap = new Map();
myMap.set(0, "zero");
myMap.set(1, "one");
for (let [key, value] of myMap) { console.log(key + " = " + value); }
// 将会输出两个log，一个是"0 = zero"，另一个是"1 = one"
for (let key of myMap.keys()) { console.log(key); }
// 将会输出两个log， 一个是 "0" 另一个是 "1"
for (let value of myMap.values()) { console.log(value); }
// 将会输出两个log， 一个是 "zero" 另一个是 "one"
for (let [key, value] of myMap.entries()) { console.log(key + " = " + value); }
// 将会输出两个log，一个是 "0 = zero" 另一个是 "1 = one"
myMap.forEach(function(value, key) { console.log(key + " = " + value); })
// 将会输出两个logs，一个是 "0 = zero" 另一个是 "1 = one"
```

Map 类型与数组类型互转，举例：

```
// Map 与 数组互转
var kvArray = [["key1", "value1"], ["key2", "value2"]];
// 使用常规的Map构造函数可以将一个二维键值对数组转换成一个Map对象
var myMap = new Map(kvArray);
myMap.get("key1"); // 返回值为 "value1"
// 使用Array.from函数可以将一个Map对象转换成一个二维键值对数组
console.log(Array.from(myMap)); // 输出和kvArray相同的数组
// 在键或者值的迭代器上使用Array.from，进而得到只含有键或者值的数组
console.log(Array.from(myMap.keys())); // 输出 ["key1", "key2"]
console.log(Array.from(myMap.values())); // 输出 ["value1", "value2"]
```

Map 对象间可以合并，有重复键值时，后面的会覆盖前面的；

```
// Map对象间可以进行合并，但是会保持键的唯一性
var first = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three'],
]);
var second = new Map([
  [1, 'uno'],
  [2, 'dos']
]);
// 合并两个Map对象时，如果有重复的键值，则后面的会覆盖前面的。
// 展开运算符本质上是将Map对象转换成数组。
var merged = new Map([...first, ...second]);
console.log(merged.get(1)); // uno
console.log(merged.get(2)); // dos
console.log(merged.get(3)); // three
```

9.Enum类型

使用枚举可以清晰地表达意图或创建一组有区别的用例；

TypeScript 支持数字的和基于字符串的枚举。

```
// Enum 类型,字面量枚举成员
enum Direction {
    Up,       // 0
    Down,     // 1
    Left,     // 2
    Right,    // 3
}
enum Direction2 {
    Up = 1,   // 当初始化第一个常量后，其余后续的成员会从 1开始自动增长
    Down,     // 2
    Left,     // 3
    Right,    // 4
}
enum Status {
    Yes = 'yes',   // 为string 类型时，不会自动初始化
    No = 'no'      // 必须为其初始化
}
```

Enum 类型可由计算的枚举成员定义；

```
// 计算的枚举成员
enum FileAccess {
    // constant members
    None,
    Read    = 1 << 1,
    Write   = 1 << 2,
    ReadWrite  = Read | Write,
    // computed member
    G = "123".length
}
```

Enum 类型应用实例：

```
// 枚举成员类型
enum ShapeKind {
    Circle,  // = 0
    Square,  // = 1
}
interface Circle {
    kind: ShapeKind.Circle;
    radius: number;
}
interface Square {
    kind: ShapeKind.Square;
    sideLength: number;
}
let c: Circle = {
    kind: ShapeKind.Square,
    // Error, kind 应该是 ShapeKind.Circle 类型
    radius: 100,
}
```

Enum 类型反射映射，举例：

```
// 反向映射
enum Enum {
    A
}
let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

10.Symbol类型

自ES6起，Symbol 成为了一种新的原生类型，symbol 类型的值是通过Symbol构造函数创建的。

```
// Symbol 类型
// 创建 Symbol 类型变量
let s1 = Symbol(); // 参数可为空，接受任意类型参数
let s2 = Symbol('abc');
let s3 = Symbol('abc');
typeof s1  // 'symbol'
s1 === s2 // false
s2 === s3 // false
```

应用场景一：使用Symbol来作为对象属性名(key)

```
// 使用Symbol来作为对象属性名(key)
let hobbyKey = Symbol();
let student = {
    age: 18,
    name: 'Jack'
}
student[hobbyKey] = 'Singing';
console.log(student);
// => { age: 18, name: 'Jack', [Symbol()]: 'Singing' }
console.log(JSON.stringify(student));
// => {"age":18,"name":"Jack"}
let props = Object.getOwnPropertyNames(student);
console.log(props);
// => [ 'age', 'name' ]
```

那应该如何获取这些symbol key 对应的值？

```
// 使用Object的API
let allProps = Object.getOwnPropertySymbols(student)
console.log(allProps); // [Symbol()]
// 使用新增的反射API
let result = Reflect.ownKeys(student);
console.log(result); // [ 'age', 'name', Symbol() ]
console.log(student[result[2]]); // Singing
```

应用场景二：使用Symbol来替代常量

```
// 使用 Symbol类型，即能保证唯一，又省去命名的麻烦
const STATUS_UNPAID = Symbol()
const STATUS_PAID = Symbol()
const STATUS_FINISHED = Symbol()
function handleOrder(id: string, status) {
```

```
  switch(status) {
    case STATUS_UNPAID:
      // todo
    case STATUS_PAID:
      // todo
    case STATUS_FINISHED:
      // todo
    default:
      throw new Error('Unknown status')
  }
}
```

应用场景三：使用Symbol定义类的私有属性/方法量

```
// a.ts
const PASSWORD = Symbol()
class Login {
  username: string
  constructor(username, password) {
    this.username = username
    this[PASSWORD] = password
  }
  checkPassword(pwd) {
      return this[PASSWORD] === pwd
  }
}
// b.ts
const login = new Login('admin', '123456')
login.checkPassword('123456')  // true
login.PASSWORD    // undefined
login[PASSWORD]   // undefined
login["PASSWORD"] // undefined
```

十、其他类型

1.联合类型

联合类型表示一个值可以是几种类型之一；

使用竖线（ | ）分隔每个类型。

```
// number | string | boolean
// 表示一个值可以是 number， string，或 boolean。
let v : number | string | boolean;
v = 9           // OK
v = 'yes'       // OK
v = false       // OK
v = new Date()  // Error
```

2.字符串字面量类型

联合类型应用实例代码：

```
// 联合类型（Union Types）实例 1
```

```
function display(name: string|string[]) { }
// 联合类型（Union Types）实例 2
interface Bird {
    fly();
    layEggs();
}
interface Fish {
    swim();
    layEggs();
}
function getSmallPet(): Fish | Bird {
    // ...
    return ;
}
let pet = getSmallPet();
pet.layEggs(); // OK
pet.swim();    // Error, 因为不能确定返回类型一定包含swim 方法
```

3.数字字面量类型

数字字面量类型允许指定必须的固定值；

```
// 数字字面量类型
type WeekDay = 1 | 2 | 3 | 4 | 5 | 6 | 7;
function setDay(day : WeekDay)  {
    // code
}
setDay(8); // Error, 参数不被允许
setDay("3"); // Error, 参数不被允许
```

4.映射类型

TypeScript 提供了从旧类型中创建新类型的一种方式 — 映射类型；

```
// 映射类型
interface Person {
    name: string;
    age: number;
}
// 可选版本
interface PersonPartial {
    name?: string;
    age?: number;
}
// 只读版本：
interface PersonReadonly {
    readonly name: string;
    readonly age: number;
}
```

泛型映射及其应用举例：

```
// 泛型映射
type Readonly1<T> = {
    readonly [P in keyof T]: T[P];
}
type Partial1<T> = {
    [P in keyof T]?: T[P];
}
// 使用泛型映射实例
type PersonPartial1 = Partial1<Person>;
type ReadonlyPerson1 = Readonly1<Person>;
```

简单映射类型举例：

```
// 简单映射类型
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
type Flags1 = {        // 与 Flags 等效
    option1: boolean;
    option2: boolean;
}
```

5.类型别名

类型别名举例：

```
// 类型别名
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    }
    else {
        return n();
    }
}
// 类型别名也可以是泛型
type Container<T> = { value: T };
```

章总结

重难点

- 面象对应【重点】
- 函数【重点】
- 泛型【难点】【易错点】
- 高级类型【重点】
- 其它类型【难点】
- 高级类型：RegExp 类型【易错点】
- 高级类型：Set 类型【易错点】
- 高级类型：Map 类型【易错点】
- 高级类型：Symbol 类型【易错点】

拓展延伸

思考：如何应用 TypeScript 的类型解决具体业务需求中的问题？

延伸资料：

- TypeScript 官网文档 http://www.typescriptlang.org/docs/home.html
- TypeScript 中文文档 https://www.tslang.cn/docs/home.html
- JavaScript 权威指南 https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide