# 开始

教程：<https://www.tslang.cn/>（官网教程）、<https://ts.xcatliu.com/>（阮一峰教程）

安装：npm install -g typescript

编译：项目目录下打开CMD，输入：tsc xxx.ts或者tsc xxx.ts -t es5回车

# 基本数据类型

1. 数据类型分为“**基本类型**”和“**对象**”，开头大写是对象，小写是基本类型
2. “对象”可转“类型”，“类型”不能转“对象”
3. 变量可为多种类型，“|”分割
4. 变量类型被确认后不可以被赋予其它类型值

## 布尔

let \_bool:boolean = false;

let bools:boolean[]=[true,false,true,false,false];

let Bool:Boolean = true;

let Bools:Boolean[] = [true,false,true,false,true,false];

let Bools\_copy:Boolean = bools[1];

~~let bools\_copy:boolean = Bools[1];~~ 错误，不能类型转对象

## 数字

let decLiteral:number = 0o744;//8进制

let DecLiteral:Number = 0o744;//8进制

let binaryLiteral:number = 0xf00d;//16进制

let BinaryLiteral:Number = 0xf00d;//16进制

let octalLiteral:number = 0b100;//2进制

let OctalLiteral:Number = 0b100;//2进制

let numberArr:number[] = [1,2,3,4];

let NumberArr:Number[] = [1,2,3,4];

~~let oct\_copy:number = OctalLiteral;~~ 错误，不能类型转对象

## 字符串

let str:string = "string";

let STR:String = "string";

let strArr:string[] = ["a","b","c"];

let STRArr:String[] = ["a","b","c"];

~~let str\_copy:string = STR;~~ 错误，不能类型转对象

## Array

建议使用数组的情景下使用它。

let list:Array<number> = [1,20,3,40,5,60,7,80];

## 元组Tuple

可以存放不同类型数据的数组

let tuple:[string,number,boolean] = ['good',18,true];

## 枚举enum

可以理解为一个kv对集合，v值默认从0开始向后依次编号，可自定义头部或全部编号。

enum fruit {Apple,Orange,Banala,WaterMelon};

let apple:fruit = fruit.Apple;

let orange:fruit = fruit.Orange;

enum Fruit {Apple = 1,Orange = 3,Banala = 5,WaterMelon = 7};

let apple:**string** = fruit[3];//用value取key，string类型

## 任意类型any

可赋任何类型值

let \_bool:any = true;

let \_boolArr:arr[] = [true,false,true,false];

let \_num:any = 123;

Let \_numArr:any[] = [1,2,3,4,5];

let \_str:any = "abc";

let \_str:any[] = ["abc"];

let \_enum:any = fruit.Apple;

## null、undefined、void

void定义的变量可以被null，undefined赋值；用处不大！

## Symbol数据类型

和ES6的symbol一样，Symbol被用来定义唯一标识；**注意：括号里的不是值，而是对这个唯一值的描述，而且symbol的描述可以重复，虽然重复了但所代表的类型和是不同的**。TS和ES6支持使用class定义类，用类实例化对象，不过Symbol不允许在类的定义中使用

    let student = {

        name:"张三",

        [Symbol("age")]:18,

        [Symbol("age")]:21

    }

    let res:any[] = Object.getOwnPropertySymbols(student);

    console.log(student[res[1]],student[res[0]]);

# 接口

接口的作用就是为了约束类型。比如对象，参数。“属性名?”代表可选。“readonly 属性”代表定义后只读。readonly仅在接口有效。接口添加 [propName: string]: any;属性代表额外允许任意类型的属性。

interface params {readonly name:String,age:Number,sex?:Boolean}

//参数

function prin(params:params):void {

console.log(params.name,params.age,params.sex);

}

//对象

let obj:params = {name:'alice',sex:true,age:19}

//类，需要全部实现接口中内容

class person implements params {

name:String;

age:Number;

sex:Boolean;

constructor(params:params){

this.name = params.name;

this.age = params.age;

this.sex = params.sex;

}

}

## 接口继承接口

可继承多个父接口

interface NAME {

name:String

}

interface ID {

id:String

}

interface AGE extends NAME {

age: Number

}

interface SEX extends AGE ,ID{

sex: Boolean

}

let Obj:SEX = {name:'',sex:null,age:0}

## 交叉类型和联合类型

将多个字典类型合并为一个新的字典类型，交叉类型中相同的属性并不会发生冲突

//被ageType修饰的变量可以是string或者number类型中的任意一种(联合类型)

    type ageType = string | number

    interface obj\_1 { name:string, age:ageType, sex?:boolean }

    interface obj\_2 { school:string, tel:string, email:string }

//合并obj\_1和obj\_2成为obj\_3，obj\_1将拥有obj\_2和obj\_3的所有属性（交叉类型）

    type obj\_3 = obj\_1 & obj\_2

let obj:obj\_3={

name:'jack',age:10,sex:true,school:'18学院',tel:'010', email:'xx@mail.com'

}

    console.log('第一次输出:',obj)

    obj.age = '10'//对age赋string类型值

console.log('第二次输出:',obj)

/\*\*

\* 类型保护与区分类型，基本类型支持使用typeof判断，延伸类型支持instanceof

\*/

    interface student  {

        learn():void

    }

    interface teacher {

        teach():void

    }

    function persons(person:student|teacher):person is teacher {

        return (person as teacher).teach != undefined

    }

    let tea:teacher = {teach:function(){alert("教育")}}

    let res = persons(tea)

console.log(res)

## 索引类型

 /\*\*

  \* **索引类型**：通过string或是number类型索引数组或对象，索引到具体值。

  \* 数组规定索引类型可以尽可能地约束接口使用者操作，加readonly将限制替换值

 \*/

   //对象就用string类型作为索引

    interface subscript {

        readonly [index:string]:any

    }

    let persons:subscript = {

        name:"jack",

        age:10,

        sex:true

    }

    console.log(persons['age'])

    //persons["age"] = false//报错

    //数组就用number类型作为索引

    interface subscript\_1 {

        readonly [index:number]:any

    }

    let persons\_1:subscript\_1 = [1,'jack',true]

    console.log(persons\_1[1]);

# 类

/\*\*

 \* person类被student和teacher两个类继承使用。子类在构造器中必须使用super(…)实

 \* 现父类的构造，然后才能是自己的特有属性的赋值（为避免遗忘，一般将super写在

 \* constructor方法中第一行）；类同样可以用readonly限制属性，被readonly限制的属

 \* 性只能在constructor中赋值；使用(<any>对象)[属性名]可以很方便地访问对象

\*/

class person {

    name: string

    readonly age: number //只读属性

    sex: boolean

    constructor(name: string, age: number, sex: boolean) {

        this.name = name

        this.age = age

        this.sex = sex

    }

}

//继承

class student extends person {

    schoolID: string

    constructor(name: string,age: number, sex: boolean, schoolID: string) {

        super(name, age, sex) //必须先实现父类的构造方法

        this.schoolID = schoolID

    }

    learn(): void {

        console.log(`${this.name}在学习`)

    }

}

class teacher extends person {

    jobID: string

    constructor(name: string, age: number, sex: boolean, jobID: string) {

        super(name, age, sex)

        this.jobID = jobID

    }

    teach(): void {

        console.log(`${this.name}在上课`)

    }

}

let stu: student = new student('张三', 19, true, '1231231')

let tea: teacher = new teacher('李四', 39, false, '2341123')

console.log(stu, tea);

stu.learn()

tea.teach()

/\*\*

 \* 类实现接口实例

 \* \*/

interface action {

    name: string,

    age: number,

    sex: boolean

    eat()

    say()

    walk()

}

class person implements action {

    //继承接口属性

    name: string

    age: number

    sex: boolean

    constructor(name: string, age: number, sex: boolean) {

        this.name = name

        this.age = age

        this.sex = sex

    }

    //实现接口方法

    eat() {

        console.log(`${this.name}吃饭`)

    }

    say() {

        console.log(`${this.name}说话`)

    }

    walk() {

        console.log(`${this.name}走路`)

    }

}

let xiaoming: person = new person('小明', 19, true)

xiaoming.eat()

xiaoming.say()

xiaoming.walk()

## 私有属性

private修饰类中私有属性；仅类中可访问

## 只读属性

readonly修饰类中只读属性；赋值后仅可读

## 静态属性

静态属性：static修饰的属性不属于类实体，而属于类本身，所有类实体共用读取修改

## get/set方法

    class account {

        private \_username:string;

        private \_password:string;

        get username():string{

            return this.\_username;

        }

        set username(newName:string){

            this.\_username = newName;

        }

        get password():string{

            return this.\_password;

        }

        set password(psword:string){

            this.\_password = psword;

        }

    }

    let account1 = new account();

    account1.password  = '12331';//set

    console.log(account1.password);//get

## 抽象类

/\*abstract修饰抽象类。abstract修饰抽象方法；抽象方法且能实现，需在继承类中实现\*/

abstract class person {

    name: string

    age: number

    sex: boolean

    constructor(name: string, age: number, sex: boolean) {

        this.name = name

        this.age = age

        this.sex = sex

     }

     abstract doSomeThing():void

}

class student extends person {

        schoolID: string

        doSomeThing(): void {

            console.log("我是继承的抽象类的抽象方法")

        }

        constructor(name:string,age:number,sex:boolean, schoolID: string) {

            super(name, age, sex)

            this.schoolID = schoolID

        }

}

# 泛型

/\*不一定是T，也可以是其他的字母\*/

function getLength<T>(value:Array<T>):number {

    return value.length

}

function getType<T>(value:T):string{

    return typeof value

}

function returnSelf<T>(value:T):T{

    return value

}

# 枚举

/\*\*

 \* 数字枚举被指定首项后依次分配（默认从0开始）也可以手动写全，而其他类型枚

 \* 举只能依次赋值。枚举值必须是确定的，不能是函数。仅数字枚举具备**反向映射**

\*/

enum action{ fisrt = 1,second,third,forth }

enum Shop{

    start='a',

    continue='b',

    end='c'

}

enum letters{A,B,C,D,E,F,G}//未指定首项，默认从0开始

console.log(letters[0],letters.A);//**反向映射**打印结果 A 0

# 别名

  type age = number //将number命名age别名，以后就可以使用age定义变量了，下同！

  type sex = boolean//将boolean命名sex别名

  type name = string//将string命名name别名

  type all = age//将age命名all别名

  let a:all = 10

  console.log(a)

## 字面量类型

//不仅是基础类型，type还可以给基础对象名别名

    type choose = 1 | 2 | 3 | 4 | 5 | 6

    function getChooseValue(param: choose) {

        switch (param) {

            case 1: window.alert(1); break;

            case 2: window.alert(2); break;

            case 3: window.alert(3); break;

            default: window.alert('other'); break;

        }

    }

    getChooseValue(3)

# 断言

//开发者绕过编译器，手动判断数据类型，这种方式很冒险；断言有两种语法格式，1：<type>value,2:value as type推荐使用第二种

function say(value:any):string {

       return value as string;//断言value为string类型

   }

   let s= say("123123");

   console.log(s);

# 数组

/\*\*

 \* 定义数组时一定要定义这个数组存储的数据类型，typescript不允许string数组中存

\* 在数字，也不允许number数组中存在字符串；若要克服这种局限请使用any定义数组

\*/

 let num\_arr:number[] = [1,2,3,4,5,6,7]

 let str\_arr:String[] = [new String('a'),new String('b'),new String('c')]

 let any\_arr:any[] = [new String('a'), 6,true]

/\* 使用泛型定义数组 \*/

 let bool\_arr:**Array<**boolean**>** = [true,false,true,false,false]

## iterator

Array,map,set,string,int32Array,unit32Array支持iterator遍历。for…of遍历值，for…in遍历键且可以操作对象。

  let obj = ['jack',true,19]

    //遍历值

    for (const iterator of obj) {

        console.log(iterator)

    }

    //遍历键（下标）

    for (const key in obj) {

        console.log(key)

    }

## generator

//迭代器，需要自己定义，定义格式为function\* 迭代器名(){}

    let obj = ['jack',true,19]

    //定义迭代器

    function\* ints(array:Array<any>){

        let length = (array as Array<any>).length

        do {//**关键字yield很重要**，标志着向结果返回的value值

            yield array[length-=1]

        } while (length > 0);

    }

    let res = ints(obj)//初始化迭代器

    //使用迭代器

    console.log(res.next(),res.next().value)

    console.log(res.next(),res.next().value)

    console.log(res.next(),res.next().value)

    console.log(res.next(),res.next().value)

# 函数

/\*\*

 \* 函数定义的3种方法；函数必须指定参数类型，需要返回值必须指定返回值类型，不需要就

 \* 写:void

\*/

let fun\_1 = function (val:string):string {

     return val.toUpperCase()

}

function fun\_2(val:string):string{

    return val.toLowerCase()

}

let val:string = `i am good,THANK YOU`

//打印结果：I AM GOOD,THANK YOU i am good,thank you

console.log(fun\_1(val),fun\_2(val))

//最后一种比较少用，是通过接口定义的无名函数，然后通过定义不同名字实现，其实interface只定义了一个框架，并未实现内容；

interface P {

       (name:string,age:number):void;

   }

   let ps:P = (name:string,age:number):void => {

       console.log(`${name}已经${age}岁了`);

   }

# Shape类

/\*\*

  \* 对一个对象的各个属性的描述，这个对象不由任何类实现

 \*/

let tom = {

name : "123",

age : 18,

sex : true

}

# 声明文件

声明语法格式速查表

|  |  |  |
| --- | --- | --- |
| 格式 | 解释 | 用途说明 |
| declare let | 声明全局变量 |  |
| declare function | 声明全局方法 |  |
| declare class | 声明全局类 |  |
| declare enum | 声明全局枚举 |  |
| declare namespace | 声明全局对象 |  |
| interface和type | 声明全局接口和类 |  |
| export | 导出变量 |  |
| export namespace | 导出对象 |  |
| export = | CommonJS导出模块 |  |
| export as namespace | UMD库声明全局变量 |  |
| declare global | 扩展全局变量 |  |
| declare module | 扩展模块 |  |
| ///<reference /> | 三斜线指令 |  |

TypeScript编程约定，将声明放入xxx.d.ts命名的文件中才可被全局访问。

# 命名空间与模块

模块：与ES6一样，模块导入 import …from ‘…’和模块导出export {…}，模板导出可以使用别名export {… as xxx}，模板导入也可以使用别名import { … as xxx } from ‘…’。

命名空间：*namespace 命名空间名{……}*将不同ts文件里的数据放入同一命名空间内，虽然在文件系统的层面上看这些数据属于不同的文件，不过在typescript编译器层面上看，同一命名空间下定义的变量均在文件之间使用时可以不需export和import，即可使用对方文件中定义的方法，类，接口等属性。