link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=380 sentences=356, words=3289

1. typescript是什么

- Typescript是由微软开发的一款开源的编程语言
- Typescript是Javascript的超集，遵循最新的ES5/ES6规范。TypeScript扩展了Javascript语法
- TypeScript更像后端Java、C#这样的面向对象语言可以让JS开发大型企业应用
- 越来越多的项目是基于TS的，比如VSCode、Angular6、Vue3、React16
- TS提供的类型系统可以帮助我们在写代码的时候提供更丰富的语法提示
- 在创建前的编译阶段经过类型系统的检查，就可以避免很多线上的错误

2. TypeScript安装和编译

2.1 安装

```
cnpm i typescript -g
```

```
tsc helloworld.ts
```

2.2 Vscode+TypeScript

2.2.1 生成配置文件

```
tsc --init
```

```
{
  "compilerOptions": {

    "target": "es5",
    "module": "commonjs",

  }
}
```

2.2.2 执行编译

```
tsc
```

2.2.3 vscode运行

- Terminal->Run Task-> tsc:build 编译
- Terminal->Run Task-> tsc:watch 编译并监听

2.2.4 npm scripts

- npm run 实际上是调用本地的 Shell 来执行对应的 script value，所以理论上能兼容所有 bash 命令
- Shell 在类 Unix 系统上是 /bin/sh，在 Windows 上是 cmd.exe

2.2.5 npm scripts 的 PATH

- npm run 会预置 PATH，对应包下的 node_modules/.bin 目录

3. 数据类型

3.1 布尔类型(boolean)

```
let married: boolean=false;
```

3.2 数字类型(number)

```
let age: number=10;
```

3.3 字符串类型(string)

```
let firstname: string='zfpx';
```

3.4 数组类型(array)

```
let arr2: number[]=[4,5,6];
let arr3: Array=[7,8,9];
```

3.5 元组类型(tuple)

- 在 TypeScript 的基础类型中，元组（ Tuple ）表示一个已知 数量和 类型的数组

```
let zhufeng:[string,number] = ['zhufeng',5];
zhufeng[0].length;
zhufeng[1].toFixed(2);
```

元组 数组 每一项可以是不同的类型 每一项都是同一种类型 有预定义的长度 没有长度限制 用于表示一个结构 用于表示一个列表

```
const animal:[string,number,boolean] = ['zhufeng',10,true];
```

3.6 枚举类型(enum)

- 事先考虑某一个变量的所有的可能的值，尽量用自然语言中的单词表示它的每一个值
- 比如性别、月份、星期、颜色、单位、学历

3.6.1 普通枚举

```
enum Gender{
    GIRL,
    BOY
}
console.log(`李雷是${Gender.BOY}`);
console.log(`韩梅梅是${Gender.GIRL}`);

enum Week{
    MONDAY=1,
    TUESDAY=2
}
console.log(`今天是星期${Week.MONDAY}`);
```

3.6.2 常数枚举

- 常数枚举与普通枚举的区别是，它会在编译阶段被删除，并且不能包含计算成员。
- 假如包含了计算成员，则会在编译阶段报错

```
const enum Colors {
    Red,
    Yellow,
    Blue
}

let myColors = [Colors.Red, Colors.Yellow, Colors.Blue];
```

```
const enum Color {Red, Yellow, Blue = "blue".length};
```

3.7 任意类型(any)

- any就是可以赋值给任意类型
- 第三方库没有提供类型文件时可以使用 any
- 类型转换遇到困难时
- 数据结构太复杂难以定义

```
let root:any=document.getElementById('root');
root.style.color='red';
```

3.8 null 和 undefined

- null 和 undefined 是其它类型的子类型，可以赋值给其它类型，如数字类型，此时，赋值后的类型会变成 null 或 undefined
- strictNullChecks

```
let x: number;
x = 1;
x = undefined;
x = null;

let y: number | null | undefined;
y = 1;
y = undefined;
y = null;
```

3.9 void 类型

- void 表示没有任何类型
- 当一个函数没有返回值时，TS 会认为它的返回值是 void 类型。
- 当我们声明一个变量类型是 void 的时候，它的非严格模式下仅可以被赋值为 null 和 undefined;

```
function greeting(name:string):void {
    console.log('hello',name);
}
greeting('zfpx');
```

3.10 never类型

never是其它类型(null undefined)的子类型，代表不会出现的值

3.10.1

- 作为不会返回（ return ）的函数的返回值类型

```
function error(message: string): never {
    throw new Error(message);
}

function fail() {
    return error("Something failed");
}

function infiniteLoop(): never {
    while (true) {}
}
```

3.10.2 strictNullChecks

- 在 TS 中， null 和 undefined 是任何类型的有效值，所以无法正确地检测它们是否被错误地使用。于是 TS 引入了 --strictNullChecks 这一种检查模式
- 由于引入了 --strictNullChecks，在这一模式下，null 和 undefined 能被检测到。所以 TS 需要一种新的底部类型（ bottom type ）。所以就引入了 never。

```
function fn(x: number | string) {
  if (typeof x === 'number') {

  } else if (typeof x === 'string') {

  } else {

  }
}
```

3.10.3 never 和 void 的区别

- void 可以被赋值为 null 和 undefined的类型。 never 则是一个不包含值的类型。
- 拥有 void 返回值类型的函数能正常运行。拥有 never 返回值类型的函数无法正常返回，无法终止，或会抛出异常。

3.11 类型推论

- 是指编程语言中能够自动推导出值的类型的能力，它是一些强静态类型语言中出现的特性
- 定义时未赋值就会推论成any类型
- 如果定义的时候就赋值就能利用到类型推论

```
let username2;
username2 = 10;
username2 = 'zhufeng';
username2 = null;
```

3.12 包装对象（Wrapper Object）

- JavaScript 的类型分为两种：原始数据类型（Primitive data types）和对象类型（Object types）。
- 所有的原始数据类型都没有属性（property）
- 原始数据类型
    - 布尔值
    - 数值
    - 字符串
    - null
    - undefined
    - Symbol

```
let name = 'zhufeng';
console.log(name.toUpperCase());

console.log((new String('zhufeng')).toUpperCase());
```

- 当调用基本数据类型方法的时候，JavaScript 会在原始数据类型和对象类型之间做一个迅速的强制性切换

```
let isOK: boolean = true;
let isOK: boolean = Boolean(1)
let isOK: boolean = new Boolean(1);
```

3.13 联合类型

- 联合类型上只能访问两个类型共有的属性和方法

```
let name4: string | number;
name4 = 3;
name4 = 'zhufeng';
console.log(name4.toUpperCase());
```

3.14 类型断言

- 类型断言可以将一个联合类型的变量，指定为一个更加具体的类型
- 不能将联合类型断言为不存在的类型

```
let name5: string | number;
(name5 as number).toFixed(3);
(name5 as string).length;
(name5 as boolean);
```

3.15 字符串、数字、布尔值字面量

```
type Lucky = 1 | 'One'|true;
let foo:Lucky = 'One';
```

3.16 字符串字面量 vs 联合类型

- 字符串字面量类型用来约束取值只能是某 几个字符串中的一个, 联合类型（Union Types）表示取值可以为 多种类型中的一种
- 字符串字面量 限定了使用该字面量的地方仅接受特定的值,联合类型 对于值并没有限定，仅仅限定值的类型需要保持一致

4. 函数

4.1 函数的定义

```
function hello(name:string):void {
    console.log('hello',name);
}
hello('zfpx');
```

4.2 函数表达式

- 定义函数类型

```
type GetUsernameFunction = (x:string,y:string)=>string;
let getUsername:GetUsernameFunction = function(firstName,lastName){
  return firstName + lastName;
}
```

4.3 没有返回值

```
let hello2 = function (name:string):void {
    console.log('hello2',name);
}
hello('zfpx');
hello2('zfpx');
```

4.4 可选参数

在TS中函数的形参和实参必须一样，不一样就要配置可选参数,而且必须是最后一个参数

```
function print(name:string,age?:number):void {
    console.log(name,age);
}
print('zfpx');
```

4.5 默认参数

```
function ajax(url:string,method:string='GET') {
    console.log(url,method);
}
ajax('/users');
```

4.6 剩余参数

```
function sum(...numbers:number[]) {
    return numbers.reduce((val,item)=>val+=item,0);
}
console.log(sum(1,2,3));
```

4.7 函数重载

- 在Java中的重载，指的是两个或者两个以上的同名函数，参数不一样
- 在TypeScript中，表现为给同一个函数提供多个函数类型定义

```
let obj: any={};
function attr(val: string): void;
function attr(val: number): void;
function attr(val:any):void {
    if (typeof val === 'number') {
        obj.age=val;
    } else {
        obj.name=val;
    }
}
attr('zfpx');
attr(9);
attr(true);
console.log(obj);
```

5. 类

5.1 如何定义类

```
class Person{
    name:string;
    getName():void{
        console.log(this.name);
    }
}
let p1 = new Person();
p1.name = 'zhufeng';
p1.getName();
```

5.2 存取器

- 在 TypeScript 中，我们可以通过存取器来改变一个类中属性的读取和赋值行为
- 构造函数
    - 主要用于初始化类的成员变量属性
    - 类的对象创建时自动调用执行
    - 没有返回值

```
class User {
    myname:string;
    constructor(myname: string) {
        this.myname = myname;
    }
    get name() {
        return this.myname;
    }
    set name(value) {
        this.myname = value;
    }
}

let user = new User('zhufeng');
user.name = 'jiagou';
console.log(user.name);
```

5.3 参数属性

```
class User {
    constructor(public myname: string) {}
    get name() {
        return this.myname;
    }
    set name(value) {
        this.myname = value;
    }
}

let user = new User('zhufeng');
console.log(user.name);
user.name = 'jiagou';
console.log(user.name);
```

5.4 readonly

- readonly修饰的变量只能在 构造函数中初始化
- 在 TypeScript 中，const 是 常量标志符，其值不能被重新分配
- TypeScript 的类型系统同样也允许将 interface、type、 class 上的属性标识为 readonly
- readonly 实际上只是在 编译阶段进行代码检查。而 const 则会在 运行时检查（在支持 const 语法的 JavaScript 运行时环境中）

```
class Animal {
    public readonly name: string
    constructor(name) {
        this.name = name;
    }
    changeName(name:string){
        this.name = name;
    }
}

let a = new Animal('zhufeng');
a.changeName('jiagou');
```

5.5 继承

- 子类继承父类后子类的实例就拥有了父类中的属性和方法，可以增强代码的可复用性
- 将子类公用的方法抽象出来放在父类中，自己的特殊逻辑放在子类中重写父类的逻辑
- super可以调用父类上的方法和属性

```
class Person {
    name: string;
    age: number;
    constructor(name:string,age:number) {
        this.name=name;
        this.age=age;
    }
    getName():string {
        return this.name;
    }
    setName(name:string): void{
        this.name=name;
    }
}
class Student extends Person{
    no: number;
    constructor(name:string,age:number,no:number) {
        super(name,age);
        this.no=no;
    }
    getNo():number {
        return this.no;
    }
}
let s1=new Student('zfpx',10,1);
console.log(s1);
```

5.6 类里面的修饰符

```
class Father {
    public name: string;
    protected age: number;
    private money: number;
    constructor(name:string,age:number,money:number) {
        this.name=name;
        this.age=age;
        this.money=money;
    }
    getName():string {
        return this.name;
    }
    setName(name:string): void{
        this.name=name;
    }
}
class Child extends Father{
    constructor(name:string,age:number,money:number) {
        super(name,age,money);
    }
    desc() {
        console.log(`${this.name} ${this.age} ${this.money}`);
    }
}

let child = new Child('zfpx',10,1000);
console.log(child.name);
console.log(child.age);
console.log(child.money);
```

5.7 静态属性 静态方法

```
class Father {
    static className='Father';
    static getClassName() {
        return Father.className;
    }
    public name: string;
    constructor(name:string) {
        this.name=name;
    }

}
console.log(Father.className);
console.log(Father.getClassName());
```

5.8 抽象类

- 抽象描述一种抽象的概念，无法被实例化，只能被继承
- 无法创建抽象类的实例
- 抽象方法不能在抽象类中实现，只能在抽象类的具体子类中实现，而且必须实现

```

abstract class Animal3 {
    name:string;
    abstract speak();
}
class Cat extends Animal3{
    speak(){
        console.log('喵喵喵');
    }
}
let cat = new Cat();
cat.speak();
```

访问控制修饰符 private protected public 只读属性 readonly 静态属性 static 抽象类、抽象方法 abstract

5.9 抽象类 vs 接口

- 不同类之间公有的属性或方法，可以抽象成一个接口（Interfaces）
- 而抽象类是供其他类继承的基类，抽象类不允许被实例化。抽象类中的抽象方法必须在子类中被实现
- 抽象类本质是一个无法被实例化的类，其中能够实现方法和初始化属性，而接口仅能够用于描述,既不提供方法的实现，也不为属性进行初始化
- 一个类可以继承一个类或抽象类，但可以实现（implements）多个接口
- 抽象类也可以实现接口

```
abstract class Animal5{
  name:string;
  constructor(name:string){
    this.name = name;
  }
  abstract speak();
}
interface Flying{
    fly()
}
class Duck extends Animal5 implements Flying{
    speak(){
        console.log('汪汪汪');
    }
    fly(){
        console.log('我会飞');
    }
}
let duck = new Duck('zhufeng');
duck.speak();
duck.fly();
```

5.10 抽象方法

- 抽象类和方法不包含具体实现，必须在子类中实现
- 抽象方法只能出现在抽象类中

```
abstract class Animal{
    abstract speak():void;
}
class Dog extends  Animal{
    speak(){
        console.log('小狗汪汪汪');
    }
}
class Cat extends  Animal{
    speak(){
        console.log('小猫喵喵喵');
    }
}
let dog=new Dog();
let cat=new Cat();
dog.speak();
cat.speak();
```

5.11 重写（override） vs 重载（overload）

- 重写是指子类重写继承自父类中的方法
- 重载是指为同一个函数提供多个类型定义

```
class Cat6 extends Animal6{
    speak(word:string):string{
        return 'Cat:'+word;
    }
}
let cat6 = new Cat6();
console.log(cat6.speak('hello'));

function double(val:number):number
function double(val:string):string
function double(val:any):any{
  if(typeof val == 'number'){
    return val *2;
  }
  return val + val;
}

let r = double(1);
console.log(r);
```

5.12 继承 vs 多态

- 继承（Inheritance）子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性
- 多态（Polymorphism）由继承而产生了相关的不同的类，对同一个方法可以有不同的响应

```
class Animal7{
    speak(word:string):string{
        return 'Animal: '+word;
    }
}
class Cat7 extends Animal7{
    speak(word:string):string{
        return 'Cat:'+word;
    }
}
class Dog7 extends Animal7{
    speak(word:string):string{
        return 'Dog:'+word;
    }
}
let cat7 = new Cat7();
console.log(cat7.speak('hello'));
let dog7 = new Dog7();
console.log(dog7.speak('hello'));
```

6. 接口

- 接口一方面可以在面向对象编程中表示为 行为的抽象，另外可以用来描述 对象的形状

- 接口就是把一些类中共有的属性和方法抽象出来,可以用来约束实现此接口的类
- 一个类可以继承另一个类并实现多个接口
- 接口像插件一样是用来增强类的，而抽象类是具体类的抽象概念
- 一个类可以实现多个接口，一个接口也可以被多个类实现，但一个类的可以有多个子类，但只能有一个父类

6.1 接口

- interface中可以用分号或者逗号分割每一项，也可以什么都不加

```
interface Speakable{
  speak():void;
  name?:string;
}

let speakman:Speakable = {
  name:string;
  speak(){}
}
```

```
interface Speakable{
    speak():void;
}
interface Eatable{
    eat():void
}
class Person5 implements Speakable,Eatable{
    speak(){
        console.log('Person5说话');
    }
    eat(){}
}
class TangDuck implements Speakable{
    speak(){
        console.log('TangDuck说话');
    }
    eat(){}
}
```

```
interface Person {
  readonly id: number;
  name: string;
  [propName: string]: any;
}

let p1 = {
  id:1,
  name:'zhufeng',
  age:10
}
```

6.2 接口的继承

- 一个接口可以继承自另外一个接口

```
interface Speakable{
  speak():void
}
interface SpeakChinese extends Speakable{
  speakChinese():void
}
class Person5 implements SpeakChinese{
  speak(){
    console.log('Person5')
  }
  speakChinese(){
    console.log('speakChinese')
  }
}
```

6.3 readonly

- 用 readonly 定义只读属性可以避免由于多人协作或者项目较为复杂等因素造成对象的值被重写

```
interface Person{
  readonly id:number;
  name:string
}
let tom:Person = {
  id :1,
  name:'zhufeng'
}
tom.id = 1;
```

6.4 函数类型接口

- 对方法传入的参数和返回值进行约束

```
interface discount{
  (price:number):number
}
let cost:discount = function(price:number):number{
   return price * .8;
}
```

6.5 可索引接口

- 对数组和对象进行约束
- userInterface 表示：只要 index 的类型是 number，那么值的类型必须是 string
- UserInterface2 表示：只要 index 的类型是 string，那么值的类型必须是 string

```
interface UserInterface {
  [index:number]:string
}
let arr:UserInterface = ['zfpx1','zfpx2'];
console.log(arr);

interface UserInterface2 {
  [index:string]:string
}
let obj:UserInterface2 = {name:'zhufeng'};
```

6.6 类接口

- 对类的约束

```
interface Speakable{
  name:string;
  speak(words:string):void
}
class Dog implements Speakable{
   name:string;
   speak(words){
    console.log(words);
   }
}
let dog=new Dog();
dog.speak('汪汪汪');
```

6.7 构造函数的类型

- 在 TypeScript 中，我们可以用 interface 来描述类
- 同时也可以使用interface里特殊的new()关键字来描述类的构造函数类型

```
class Animal{
  constructor(public name:string){
  }
}
interface WithNameClass{
  new(name:string):Animal
}
function createAnimal(clazz:WithNameClass,name:string){
   return new clazz(name);
}
let a = createAnimal(Animal,'zhufeng');
console.log(a.name);
```

7. 泛型

- 泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性
- 泛型 T作用域只限于函数内部使用

7.1 泛型函数

- 首先，我们来实现一个函数 createArray，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值

```
function createArray(length: number, value: any): Array<any> {
  let result: any = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
}
let result = createArray(3,'x');
console.log(result);
```

```
function createArray(length: number, value: any): Array<any> {
  let result: any = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
}
let result = createArray(3,'x');
console.log(result);

function createArray2<T>(length: number, value: T): Array<T> {
  let result: T[] = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
}
let result2 = createArray2(3,'x');
console.log(result);
```

7.2 类数组

- 类数组（Array-like Object）不是数组类型，比如 arguments

```
function sum(...parameters:number[]){
  let args:IArguments = arguments;
  for(let i=0;iconsole.log(args[i]);
  }
}
sum(1,2,3);
let root = document.getElementById('root');
let children:HTMLCollection = root.children;
children.length;
let nodeList:NodeList = root.childNodes;
nodeList.length;
```

7.3 泛型类

```
class MyArray<T>{
    private list:T[]=[];
    add(value:T) {
        this.list.push(value);
    }
    getMax():T {
        let result=this.list[0];
        for (let i=0;i<this.list.length;i++){
            if (this.list[i]>result) {
                result=this.list[i];
            }
        }
        return result;
    }
}
let arr=new MyArray();
arr.add(1); arr.add(2); arr.add(3);
let ret = arr.getMax();
console.log(ret);
```

7.5 泛型接口

- 泛型接口可以用来约束函数