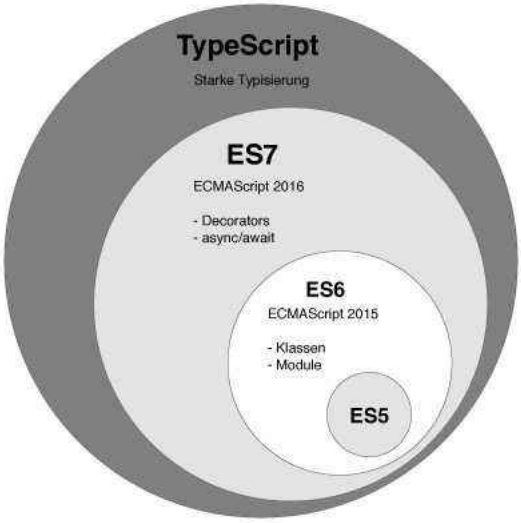
link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=773 sentences=833, words=7671

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 用于表示一个固定的结构 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';
```

```
let root:(HTMLElement|null)=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 参数用于新的严格空检查模式,在严格空检查模式下， null 和 undefined 值都不属于任何一个类型，它们只能赋值给自己这种类型或者 any

```
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。

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

}
```

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);
}
let resultl = error('hello');

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

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 Symbol

- 我们在使用 Symbol 的时候，必须添加 es6 的编译辅助库
- Symbol 是在ES2015之后成为新的原始类型,它通过 Symbol 构造函数创建
- Symbol 的值是唯一不变的

```
const sym1 = Symbol('key');
const sym2 = Symbol('key');
Symbol('key') === Symbol('key')
```

3.12 BigInt

- 使用 BigInt 可以安全地存储和操作大整数
- 我们在使用 `BigInt` 的时候，必须添加 `ESNext` 的编译辅助库
- 要使用 `1n`需要 `"target": "ESNext"`
- number和 BigInt类型不一样,不兼容

```
const max = Number.MAX_SAFE_INTEGER;
console.log(max + 1 === max + 2);
```

```
const max = BigInt(Number.MAX_SAFE_INTEGER);
console.log(max + 1n === max + 2n);
```

```
let foo: number;
let bar: bigint;
foo =bar;
bar = foo;
```

3.13 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;
```

3.14 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);
```

3.15 联合类型

- 联合类型（Union Types）表示取值可以为多种类型中的一种
- 未赋值时联合类型上只能访问两个类型共有的属性和方法

```
let name: string | number;
console.log(name.toString());
name = 3;
console.log(name.toFixed(2));
name = 'zhufeng';
console.log(name.length);

export {};
```

3.16 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 name: string | number;
console.log((name as string).length);
console.log((name as number).toFixed(2));
console.log((name as boolean));
```

双重断言

```
interface Person {
    name: string;
    age: number;
}
const person = 'zhufeng' as any as Person;
```

3.17 字面量类型和类型字面量

- 字面量类型的要和实际的值的字面量一一对应,如果不一致就会报错
- 类型字面量和对象字面量的语法很相似

```
const up:'Up'= 'Up';
const down: "Down" = "Down";
const left: "Left" = "Left";
const right: "Right" = "Right";
type Direction = 'Up' | 'Down' | 'Left' | 'Right';
function move(direction: Direction) {}
move("Up");
```

```
type Person = {
  name:string,
  age:number
};
```

3.18 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

- 可以指定参数的类型和返回值的类型

```
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);
    return undefined;
}
hello2('zhufeng');
```

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 === 'string') {
        obj.name=val;
    } else {
        obj.age=val;
    }
}
attr('zfpx');
attr(9);
attr(true);
console.log(obj);
```

5. 类

5.1 如何定义类

- "strictPropertyInitialization": true / *启用类属性初始化的严格检查*
- name!:string

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

```

class Component {
    static myName: string = '静态名称属性';
    myName: string = '实例名称属性';
}
let com = Component;

let c: Component = new Component();
let f: typeof Component = com;
```

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);
```

```
;
var User =  (function () {
    function User(myname) {
        this.myname = myname;
    }
    Object.defineProperty(User.prototype, "name", {
        get: function () {
            return this.myname;
        },
        set: function (value) {
            this.myname = value;
        },
        enumerable: true,
        configurable: true
    });
    return User;
}());
var 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:string) {
        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 装饰器

- 装饰器是一种特殊类型的声明，它能够被附加到类声明、方法、属性或参数上，可以修改类的行为
- 常见的装饰器有类装饰器、属性装饰器、方法装饰器和参数装饰器
- 装饰器的写法分为普通装饰器和装饰器工厂

```
class Person{
    say() {
        console.log('hello')
    }
}

function Person() {}
Object.defineProperty(Person.prototype, 'say', {
    value: function() { console.log('hello'); },
    enumerable: false,
    configurable: true,
    writable: true
});
```

5.8.1 类装饰器

- 类装饰器在类声明之前声明，用来监视、修改或替换类定义

```
namespace a {

    function addNameEat(constructor: Function) {
      constructor.prototype.name = "zhufeng";
      constructor.prototype.eat = function () {
        console.log("eat");
      };
    }
    @addNameEat
    class Person {
      name!: string;
      eat!: Function;
      constructor() {}
    }
    let p: Person = new Person();
    console.log(p.name);
    p.eat();
}

namespace b {

    function addNameEatFactory(name:string) {
    return function (constructor: Function) {
        constructor.prototype.name = name;
        constructor.prototype.eat = function () {
        console.log("eat");
        };
    };
    }
    @addNameEatFactory('zhufeng')
    class Person {
      name!: string;
      eat!: Function;
      constructor() {}}
    let p: Person = new Person();
    console.log(p.name);
    p.eat();
}

namespace c {

    function enhancer(constructor: Function) {
    return class {
        name: string = "jiagou";
        eat() {
        console.log("吃饭饭");
        }
    };
    }
    @enhancer
    class Person {
      name!: string;
      eat!: Function;
      constructor() {}}
    let p: Person = new Person();
    console.log(p.name);
    p.eat();

}
```

5.8.2 属性装饰器

- 属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数
- 属性装饰器用来装饰属性
    - 第一个参数对于静态成员来说是类的构造函数，对于实例成员是类的原型对象

    - 第二个参数是属性的名称
- 方法装饰器用来装饰方法
    - 第一个参数对于静态成员来说是类的构造函数，对于实例成员是类的原型对象
    - 第二个参数是方法的名称
    - 第三个参数是方法描述符

```
namespace d {

    function upperCase(target: any, propertyKey: string) {
        let value = target[propertyKey];
        const getter = function () {
            return value;
        }

        const setter = function (newVal: string) {
            value = newVal.toUpperCase()
        };

        if (delete target[propertyKey]) {
            Object.defineProperty(target, propertyKey, {
                get: getter,
                set: setter,
                enumerable: true,
                configurable: true
            });
        }
    }

    function noEnumerable(target: any, property: string, descriptor: PropertyDescriptor) {
        console.log('target.getName', target.getName);
        console.log('target.getAge', target.getAge);
        descriptor.enumerable = true;
    }

    function toNumber(target: any, methodName: string, descriptor: PropertyDescriptor) {
        let oldMethod = descriptor.value;
        descriptor.value = function (...args: any[]) {
            args = args.map(item => parseFloat(item));
            return oldMethod.apply(this, args);
        }
    }
    class Person {
        @upperCase
        name: string = 'zhufeng'
        public static age: number = 10
        constructor() { }
        @noEnumerable
        getName() {
            console.log(this.name);
        }
        @toNumber
        sum(...args: any[]) {
            return args.reduce((accu: number, item: number) => accu + item, 0);
        }
    }
    let p: Person = new Person();
    for (let attr in p) {
        console.log('attr=', attr);
    }
    p.name = 'jiagou';
    p.getName();
    console.log(p.sum("1", "2", "3"));
}
```

5.8.3 参数装饰器

- 会在运行时当作函数被调用，可以使用参数装饰器为类的原型增加一些元数据
    - 第1个参数对于静态成员是类的构造函数，对于实例成员是类的原型对象
    - 第2个参数的名称
    - 第3个参数在函数列表中的索引

```
namespace d {
    interface Person {
        age: number;
    }
    function addAge(target: any, methodName: string, paramsIndex: number) {
        console.log(target);
        console.log(methodName);
        console.log(paramsIndex);
        target.age = 10;
    }
    class Person {
        login(username: string, @addAge password: string) {
            console.log(this.age, username, password);
        }
    }
    let p = new Person();
    p.login('zhufeng', '123456')
}
```

5.8.4 装饰器执行顺序

- 有多个参数装饰器时：从最后一个参数依次向前执行
- 方法和方法参数中参数装饰器先执行。
- 类装饰器总是最后执行
- 方法和属性装饰器，谁在前面谁先执行。因为参数属于方法一部分，所以参数会一直紧紧挨着方法执行
- 类比React组件的componentDidMount 先上后下、先内后外

```
namespace e {
    function Class1Decorator() {
        return function (target: any) {
            console.log("类1装饰器");
        }
    }
    function Class2Decorator() {
        return function (target: any) {
            console.log("类2装饰器");
        }
    }
    function MethodDecorator() {
        return function (target: any, methodName: string, descriptor: PropertyDescriptor) {
            console.log("方法装饰器");
        }
    }
    function Param1Decorator() {
        return function (target: any, methodName: string, paramIndex: number) {
            console.log("参数1装饰器");
        }
    }
    function Param2Decorator() {
        return function (target: any, methodName: string, paramIndex: number) {
            console.log("参数2装饰器");
        }
    }
    function PropertyDecorator(name: string) {
        return function (target: any, propertyName: string) {
            console.log(name + "属性装饰器");
        }
    }

    @Class1Decorator()
    @Class2Decorator()
    class Person {
        @PropertyDecorator('name')
        name: string = 'zhufeng';
        @PropertyDecorator('age')
        age: number = 10;
        @MethodDecorator()
        greet(@Param1Decorator() p1: string, @Param2Decorator() p2: string) { }
    }
}
```

5.9 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 Animal {
    name!:string;
    abstract speak():void;
}
class Cat extends Animal{
    speak(){
        console.log('喵喵喵');
    }
}
let animal = new Animal();
animal.speak();
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

5.10 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
- 子类可以对抽象类进行不同的实现

```
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 Animal{
    speak(word:string):string{
        return '动作叫:'+word;
    }
}
class Cat extends Animal{
    speak(word:string):string{
        return '猫叫:'+word;
    }
}
let cat = new Cat();
console.log(cat.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行为

```
class Animal{
    speak(word:string):string{
        return 'Animal: '+word;
    }
}
class Cat extends Animal{
    speak(word:string):string{
        return 'Cat:'+word;
    }
}
class Dog extends Animal{
    speak(word:string):string{
        return 'Dog:'+word;
    }
}
let cat = new Cat();
console.log(cat.speak('hello'));
let dog = new Dog();
console.log(dog.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

6.1.1 对象的形状

```

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

let speakman:Speakable = {
    speak(){},
    name,
    age
}
```

6.1.2 行为的抽象

```

interface Speakable{
    speak():void;
}
interface Eatable{
    eat():void
}

class Person implements Speakable,Eatable{
    speak(){
        console.log('Person说话');
    }
    eat(){}
}
class TangDuck implements Speakable{
    speak(){
        console.log('TangDuck说话');
    }
    eat(){}
}
```

6.1.3 任意属性

```
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 Person implements SpeakChinese {
    speak() {
        console.log('Person')
    }
    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 index的类型是 number，那么值的类型必须是 string
- UserInterface2 表示： 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:string) {
        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);
```

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

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);
```

使用了泛型

```
function createArray<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
      result[i] = value;
    }
    return result;
  }
let result = 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() {
    let args: IArguments = arguments;
    for (let i = 0; i < args.length; i++) {
        console.log(args[i]);
    }
}
sum(1, 2, 3);

let root = document.getElementById('root');
let children: HTMLCollection = (root as HTMLElement).children;
children.length;
let nodeList: NodeList = (root as HTMLElement).childNodes;
nodeList.length;
```

7.3 泛型类

7.3.1 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);
```

7.3.2 泛型与 new

```
function factory<T>(type: {new():T}): T {
  return new type();
}
```

7.5 泛型接口

- 泛型接口可以用来约束函数

```
interface Calculate{
  (a:T,b:T):T
}
let add:Calculate = function<T>(a:T,b:T){
  return a;
}
add(1,2);
```

7.6 多个类型参数

- 泛型可以有多个

```
function swap(tuple:[A,B]):[B,A]{
  return [tuple[1],tuple[0]];
}
let swapped = swap(['a',1]);
console.log(swapped);
console.log(swapped[0].toFixed(2));
console.log(swapped[1].length);
```

7.7 默认泛型类型

```
function createArray3<T=number>(length: number, value: T): Array<T> {
  let result: T[] = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
}
let result2 = createArray3(3,'x');
console.log(result2);
```

7.8 泛型约束

- 在函数中使用泛型的时候，由于预先并不知道泛型的类型，所以不能随意访问相应类型的属性或方法。

```
function logger<T>(val: T) {
    console.log(val.length);
}

interface LengthWise {
    length: number
}

function logger2<T extends LengthWise>(val: T) {
    console.log(val.length)
}
logger2('zhufeng');
logger2(1);
```

7.9 泛型接口

- 定义接口的时候也可以指定泛型

```
interface Cart{
  list:T[]
}
let cart:Cartname:string,price:number}> = {
  list:[{name:'zhufeng',price:10}]
}
console.log(cart.list[0].name,cart.list[0].price);
```

7.10 compose

compose (https://gitee.com/zhufengpeixun/redux/blob/master/src/compose.ts)

```
import compose from ".";

console.log(compose()("zhufeng"));

interface F{
    (a:string):string
}
let f: F = (a:string):string=>a+'f';
console.log(compose(f)("zhufeng"));

type A = string;
type R = string;
type T = string[];

let f1 = (a: A): R => a + "f1";
let f2 = (...a: T): A => a + "f2";
console.log(compose(f1,f2)("zhufeng"));
```

7.11 泛型类型别名

- 泛型类型别名可以表达更复杂的类型

```
type Cart = {list:T[]} | T[];
let c1:Cart = {list:['1']};
let c2:Cart = [1];
```

7.12 泛型接口 vs 泛型类型别名

- 接口创建了一个新的名字，它可以在其他任意地方被调用。而类型别名并不创建新的名字，例如报错信息就不会使用别名
- 类型别名不能被 extends和 implements,这时我们应该尽量使用接口代替类型别名
- 当我们需要使用联合类型或者元组类型的时候，类型别名会更合适

8.结构类型系统

8.1 接口的兼容性

- 如果传入的变量和声明的类型不匹配，TS就会进行兼容性检查
- 原理是 Duck-Check,就是说只要目标类型中声明的属性变量在源类型中都存在就是兼容的

```
interface Animal {
    name: string;
    age: number;
}

interface Person {
    name: string;
    age: number;
    gender: number
}

function getName(animal: Animal): string {
    return animal.name;
}

let p = {
    name: 'zhufeng',
    age: 10,
    gender: 0
}

getName(p);

let a: Animal = {
    name: 'zhufeng',
    age: 10,
    gender: 0
}
```

8.2 基本类型的兼容性

```

let num : string|number;
let str:string='zhufeng';
num = str;

let num2 : {
  toString():string
}

let str2:string='jiagou';
num2 = str2;
```

8.3 类的兼容性

- 在TS中是结构类型系统，只会对比结构而不在意类型

```
class Animal{
    name:string
}
class Bird extends Animal{
   swing:number
}

let a:Animal;
a = new Bird();

let b:Bird;

b = new Animal();
```

```
class Animal{
  name:string
}

class Bird extends Animal{}

let a:Animal;
a = new Bird();

let b:Bird;
b = new Animal();
```

```
class Animal{
  name:string
}
class Bird{
  name:string
}
let a:Animal ;
a = new Bird();
let b:Bird;
b = new Animal();
```

8.4 函数的兼容性

- 比较函数的时候是要先比较函数的参数，再比较函数的返回值

8.4.1 比较参数

```
type sumFunc = (a:number,b:number)=>number;
let sum:sumFunc;
function f1(a:number,b:number):number{
  return a+b;
}
sum = f1;

function f2(a:number):number{
   return a;
}
sum = f2;

function f3():number{
    return 0;
}
sum = f3;

function f4(a:number,b:number,c:number){
    return a+b+c;
}
sum = f4;
```

8.4.2 比较返回值

```
type GetPerson = ()=>{name:string,age:number};
let getPerson:GetPerson;

function g1(){
    return {name:'zhufeng',age:10};
}
getPerson = g1;

function g2(){
    return {name:'zhufeng',age:10,gender:'male'};
}
getPerson = g2;

function g3(){
    return {name:'zhufeng'};
}
getPerson = g3;

getPerson().age.toFixed();
```

8.5 函数的协变与逆变

- 协变（Covariant）：只在同一个方向；
- 逆变（Contravariant）：只在相反的方向；
- 双向协变（Bivariant）：包括同一个方向和不同方向；
- 不变（Invariant）：如果类型不完全相同，则它们是不兼容的。
- A ⩽ B 意味着 A 是 B 的子类型。
- A → B 指的是以 A 为参数类型，以 B 为返回值类型的函数类型。
- x : A 意味着 x 的类型为 A
- 返回值类型是协变的，而参数类型是逆变的
- 返回值类型可以传子类,参数可以传父类
- 参数逆变父类 返回值协变子类 撩你父,返鞋子

```
class Animal{}
class Dog extends Animal{
    public name:string = 'Dog'
}
class BlackDog extends Dog {
    public age: number = 10
}
class WhiteDog extends Dog {
    public home: string = '北京'
}
let animal: Animal;
let blackDog: BlackDog;
let whiteDog: WhiteDog;
type Callback = (dog: Dog)=>Dog;
function exec(callback:Callback):void{
    callback(whiteDog);
}

type ChildToChild = (blackDog: BlackDog) => BlackDog;
const childToChild: ChildToChild = (blackDog: BlackDog): BlackDog => blackDog
exec(childToChild);

type ChildToParent = (blackDog: BlackDog) => Animal;
const childToParent: ChildToParent = (blackDog: BlackDog): Animal => animal
exec(childToParent);

type ParentToParent = (animal: Animal) => Animal;
const parentToParent: ParentToParent = (animal: Animal): Animal => animal
exec(parentToParent);

type ParentToChild = (animal: Animal) => BlackDog;
const parentToChild: ParentToChild = (animal: Animal): BlackDog => blackDog
exec(parentToChild);
```

```
type Callback2 = (a: string | number) => string | number;
function exec2(callback: Callback2):void{
    callback('');
}
type ParentToChild2 = (a: string | number | boolean) => string;
const parentToChild2: ParentToChild2 = (a: string | number | boolean): string => ''
exec2(parentToChild2);

type Callback3 = (a: string | number) => string | number;
function exec3(callback: Callback2): void {
    callback('');
}
type ParentToParent3 = (a: string) => string;
const parentToParent3: ParentToParent3 = (a: string): string => ''
exec3(parentToChild3);
```

- 在 TypeScript 中， 参数类型是双向协变的，也就是说既是协变又是逆变的，而这并不安全。但是现在你可以在 TypeScript 2.6 版本中通过 `--strictFunctionTypes` 或 `--strict` 标记来修复这个问题

8.6 泛型的兼容性

- 泛型在判断兼容性的时候会先判断具体的类型,然后再进行兼容性判断

```
interface Empty{}
let x!:Empty;
let y!:Empty;
x = y;

interface NotEmpty{
  data:T
}
let x1!:NotEmpty;
let y1!:NotEmpty;
x1 = y1;

interface NotEmptyString{
    data:string
}

interface NotEmptyNumber{
    data:number
}
let xx2!:NotEmptyString;
let yy2!:NotEmptyNumber;
xx2 = yy2;
```

8.7 枚举的兼容性

- 枚举类型与数字类型兼容，并且数字类型与枚举类型兼容
- 不同枚举类型之间是不兼容的

```
enum Colors {Red,Yellow}
let c:Colors;
c = Colors.Red;
c = 1;
c = '1';

let n:number;
n = 1;
n = Colors.Red;
```

9.类型保护

- 类型保护就是一些表达式，他们在编译的时候就能通过类型信息确保某个作用域内变量的类型
- 类型保护就是能够通过关键字判断出分支中的类型

9.1 typeof 类型保护

```
function double(input: string | number | boolean) {
    if (typeof input === 'string') {
        return input + input;
    } else {
        if (typeof input === 'number') {
            return input * 2;
        } else {
            return !input;
        }
    }
}
```

9.2 instanceof类型保护

```
class Animal {
    name!: string;
}
class Bird extends Animal {
    swing!: number
}
function getName(animal: Animal) {
    if (animal instanceof Bird) {
        console.log(animal.swing);
    } else {
        console.log(animal.name);
    }
}
```

9.3 null保护

- 如果开启了 strictNullChecks选项，那么对于可能为null的变量不能调用它上面的方法和属性

```
function getFirstLetter(s: string | null) {

    if (s == null) {
        return '';
    }

    s = s || '';
    return s.charAt(0);
}

function getFirstLetter2(s: string | null) {
    function log() {
        console.log(s!.trim());
    }
    s = s || '';
    log();
    return s.charAt(0);
}
```

9.4 链判断运算符

- 链判断运算符是一种先检查属性是否存在，再尝试访问该属性的运算符，其符号为 ?.
- 如果运算符左侧的操作数 ?. 计算为 undefined 或 null，则表达式求值为 undefined 。否则，正常触发目标属性访问，方法或函数调用。

```
a?.b;
a == null ? undefined : a.b;

a?.[x];
a == null ? undefined : a[x];

a?.b();
a == null ? undefined : a.b();

a?.();
a == null ? undefined : a();
```

> 链判断运算符 还处于 stage1 阶段,TS 也暂时不支持

9.5 可辨识的联合类型

- 就是利用联合类型中的共有字段进行类型保护的一种技巧
- 相同字段的不同取值就是可辨识

```
interface WarningButton{
  class:'warning',
  text1:'修改'
}
interface DangerButton{
  class:'danger',
  text2:'删除'
}
type Button = WarningButton|DangerButton;
function getButton(button:Button){
 if(button.class=='warning'){
  console.log(button.text1);
 }
 if(button.class=='danger'){
  console.log(button.text2);
 }
}
```

类型字面量+可辨识联合类型

```
interface User {
    username: string
}
type Action = {
    type:'add',
    payload:User
} | {
    type: 'delete'
    payload: number
}
const UserReducer = (action: Action) => {
  switch (action.type) {
    case "add":
      let user: User = action.payload;
      break;
    case "delete":
      let id: number = action.payload;
      break;
    default:
      break;
  }
};
```

9.6 in操作符

- in 运算符可以被用于参数类型的判断

```
interface Bird {
    swing: number;
}

interface Dog {
    leg: number;
}

function getNumber(x: Bird | Dog) {
    if ("swing" in x) {
      return x.swing;
    }
    return x.leg;
}
```

9.7 自定义的类型保护

- TypeScript 里的类型保护本质上就是一些表达式，它们会在运行时检查类型信息，以确保在某个作用域里的类型是符合预期的
- type is TypeClass就是类型谓词
- 谓词为 parameterName is Type这种形式, parameterName必须是来自于当前函数签名里的一个参数名
- 每当使用一些变量调用 isType1时，如果原始类型兼容， TypeScript会将该变量缩小到该特定类型

```
function isType1(type: Type1Class | Type2Class): type is Type1Class {
    return (<Type1Class>type).func1 !== undefined;
}
```

```
interface Bird {
  swing: number;
}

interface Dog {
  leg: number;
}

function isBird(x:Bird|Dog): x is Bird{
  return (<Bird>x).swing == 2;
  //return (x as Bird).swing == 2;
}

function getAnimal(x: Bird | Dog) {
  if (isBird(x)) {
    return x.swing;
  }
  return x.leg;
}
```

9.8 unknown

- TypeScript 3.0 引入了新的unknown 类型，它是 any 类型对应的安全类型
- unknown 和 any 的主要区别是 unknown 类型会更加严格：在对 unknown 类型的值执行大多数操作之前，我们必须进行某种形式的检查。而在对 any 类型的值执行操作之前，我们不必进行任何检查

9.8.1 any 类型

- 在 TypeScript 中，任何类型都可以被归为 any 类型。这让 any 类型成为了类型系统的 顶级类型 (也被称作 全局超级类型)。
- TypeScript允许我们对 any 类型的值执行任何操作，而无需事先执行任何形式的检查

```
let value: any;

value = true;
value = 42;
value = "Hello World";
value = [];
value = {};
value = Math.random;
value = null;
value = undefined;

let value: any;
value.foo.bar;
value.trim();
value();
new value();
```

9.8.2 unknown 类型

- 就像所有类型都可以被归为 any，所有类型也都可以被归为 unknown。这使得 unknown 成为 TypeScript 类型系统的另一种顶级类型（另一种是 any）

- 任何类型都可以赋值给 unknown类型

```
let value: unknown;

value = true;
value = 42;
value = "Hello World";
value = [];
value = {};
value = Math.random;
value = null;
value = undefined;
value = new TypeError();
```

- unknown类型只能被赋值给 any类型和 unknown类型本身

```
let value: unknown;
let value1: unknown = value;
let value2: any = value;
let value3: boolean = value;
let value4: number = value;
let value5: string = value;
let value6: object = value;
let value7: any[] = value;
let value8: Function = value;
```

9.8.3 缩小 unknown 类型范围

- 如果没有类型断言或类型细化时，不能在 unknown上面进行任何操作
- typeof
- instanceof
- 自定义类型保护函数
- 可以对 unknown 类型使用类型断言

```
const value: unknown = "Hello World";
const someString: string = value as string;
```

9.8.4 联合类型中的 unknown 类型

- 在联合类型中，unknown 类型会吸收任何类型。这就意味着如果任一组成类型是 unknown，联合类型也会相当于 unknown：

```
type UnionType1 = unknown | null;
type UnionType2 = unknown | undefined;
type UnionType3 = unknown | string;
type UnionType4 = unknown | number[];
```

9.8.5 交叉类型中的 unknown 类型

- 在交叉类型中，任何类型都可以吸收 unknown 类型。这意味着将任何类型与 unknown 相交不会改变结果类型

```
type IntersectionType1 = unknown & null;
type IntersectionType2 = unknown & undefined;
type IntersectionType3 = unknown & string;
type IntersectionType4 = unknown & number[];
type IntersectionType5 = unknown & any;
```

9.8.6 never是unknown的子类型

```
type isNever = never extends unknown ? true : false;
```

9.8.7 keyof unknown 等于never

```
type key = keyof unknown;
```

9.8.8 只能对unknown进行等或不等操作，不能进行其它操作

```
un1===un2;
un1!==un2;
un1 += un2;
```

9.8.9 不能做任何操作

- 不能访问属性
- 不能作为函数调用
- 不能当作类的构造函数不能创建实例

```
un.name
un();
new un();
```

9.8.10 映射属性

- 如果映射类型遍历的时候是unknown,不会映射属性

```
type getType = {
  [P in keyof T]:number
}
type t = getType;
```

10. 类型变换

10.1 类型推断

- TypeScript 能根据一些简单的规则推断变量的类型

10.1.1 从右向左

- 变量的类型可以由定义推断
- 这是一个从右向左流动类型的示例

```
let foo = 1;
let bar = 'zhufeng';
```

10.1.2 底部流出

- 返回类型能被 return 语句推断

```
function add(a: number, b: number) {
    return a + b;
}
let c = add(1,2);
```

10.1.3 从左向右

- 函数参数类型/返回值类型也能通过赋值来推断

```
type Sum = (a: number, b: number) => number;
let sum: Sum = (a, b) => {
    a='zhufeng';
    return a + b;
};
```

10.1.4 结构化

- 推断规则也适用于结构化的存在(对象字面量)

```
const person = {
    name: 'zhufeng',
    age: 11
};
let name =person.name;
let age =person.age;
age = 'hello';
```

10.1.5 解构

- 推断规则也适用于解构

```
const person = {
    name: 'zhufeng',
    age: 11
};
let { name,age } = person;

age = 'hello';

const numbers = [1, 2, 3];
numbers[0] = 'hello';
```

10.1.5 DefaultProps

```

interface DefaultProps{
    name?:string;
    age?:number;
}
let defaultProps: DefaultProps = {
   name:'zhufeng',
   age:10
}

let props = {
    ...defaultProps,
    home:'北京'
}
type Props = typeof props;
```

10.1.6 小心使用返回值

- 尽管 TypeScript 一般情况下能推断函数的返回值，但是它可能并不是你想要的

```
function addOne(a:any) {
    return a + 1;
}
function sum(a: number, b: number) {
    return a + addOne(b);
}

type Ret = ReturnType<typeof sum>;
```

10.1 交叉类型

- 交叉类型(Intersection Types)是将多个类型合并为一个类型
- 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性

```

export {}

interface Bird {
    name: string,
    fly(): void
}
interface Person {
    name: string,
    talk(): void
}
type BirdPerson = Bird & Person;
let p: BirdPerson = { name: 'zhufeng', fly() { }, talk() { } };
p.fly;
p.name
p.talk;
```

```
interface X {
    a: string;
    b: string;
}

interface Y {
    a: number;
    c: string
}

type XY = X & Y;
type YX = Y & X;
```

联合类型的交叉类型

```
type Ta = string | number;
type Tb = number | boolean;
type Tc = Ta & Tb;
```

mixin混入模式可以让你从两个对象中创建一个新对象，新对象会拥有着两个对象所有的功能

```
interface AnyObject {
    [prop: string]: any;
}

function mixin<T extends AnyObject, U extends AnyObject>(one: T,two: U): T & U {
    const result = {};
    for (let key in one) {
        (result)[key] = one[key];
    }
    for (let key in two) {
        (result)[key] = two[key];
    }
    return result;
}

const x = mixin({ name: "zhufeng" }, { age: 11 });
console.log(x.name, x.age);
```

10.2 typeof

- 可以获取一个变量的类型

```

type People = {
    name:string,
    age:number,
    gender:string
}
let p1:People = {
    name:'zhufeng',
    age:10,
    gender:'male'
}
```

```

let p1 = {
    name:'zhufeng',
    age:10,
    gender:'male'
}
type People = typeof p1;
function getName(p:People):string{
    return p.name;
}
getName(p1);
```

10.3 索引访问操作符

- 可以通过[]获取一个类型的子类型

```
interface Person{
    name:string;
    age:number;
    job:{
        name:string
    };
    interests:{name:string,level:number}[]
}
let FrontEndJob:Person['job'] = {
    name:'前端工程师'
}
let interestLevel:Person['interests'][0]['level'] = 2;
```

10.4 keyof

- 索引类型查询操作符

```
interface Person{
  name:string;
  age:number;
  gender:'male'|'female';
}

type PersonKey = keyof Person;

function getValueByKey(p:Person,key:PersonKey){
  return p[key];
}
let val = getValueByKey({name:'zhufeng',age:10,gender:'male'},'name');
console.log(val);
```

10.5 映射类型

- 在定义的时候用in操作符去批量定义类型中的属性

```
interface Person{
  name:string;
  age:number;
  gender:'male'|'female';
}

type PartPerson = {
  [Key in keyof Person]?:Person[Key]
}

let p1:PartPerson={};

type Part = {
  [key in keyof T]?:T[key]
}
let p2:Part={};
```

- 通过key的数组获取值的数组

```
function pick(o: T, names: K[]): T[K][] {
  return names.map((n) => o[n]);
}
let user = { id: 1, name: 'zhufeng' };
type User = typeof user;
const res = pick(user, ["id", "name"]);
console.log(res);
```

10.6 条件类型

- 在定义泛型的时候能够添加进逻辑分支，以后泛型更加灵活

10.6.1 定义条件类型

```
interface Fish {
    name: string
}
interface Water {
    name: string
}
interface Bird {
    name: string
}
interface Sky {
    name: string
}

type Condition = T extends Fish ? Water : Sky;
let condition: Condition = { name: '水' };
```

10.6.2 条件类型的分发

```
interface Fish {
    fish: string
}
interface Water {
    water: string
}
interface Bird {
    bird: string
}
interface Sky {
    sky: string
}

type Condition = T extends Fish ? Water : Sky;

let condition1: Condition = { water: '水' };
let condition2: Condition = { sky: '天空' };
```

- 条件类型有一个特性,就是「分布式有条件类型」,但是分布式有条件类型是有前提的,条件类型里待检查的类型必须是naked type parameter
- 找出T类型中U不包含的部分

```

type Diff = T extends U ? never : T;

type R = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">;

type Filter = T extends U ? T : never;
type R1 = Filter;
```

10.6.3 内置条件类型

- TS 在内置了一些常用的条件类型，可以在lib.es5.d.ts (https://github.com/Microsoft/TypeScript/blob/c48662c891ce810f5627a0f6a8594049cccceeb5/lib/lib.es5.d.ts#L1291) 中查看：
- utility-types (http://www.typescriptlang.org/docs/handbook/utility-types.html)

10.6.3.1 Exclude

- 从 T 可分配给的类型中排除 U

```
type Exclude = T extends U ? never : T;

type  E = Exclude;
let e:E = 10;
```

10.6.3.2 Extract

- 从 T 可分配的类型中提取 U

```
type Extract = T extends U ? T : never;

type  E = Extract;
let e:E = '1';
```

10.6.3.3 NonNullable

- 从 T 中排除 null 和 undefined

```
type NonNullable = T extends null | undefined ? never : T;

type  E = NonNullablenull|undefined>;
let e:E = null;
```

10.6.3.4 ReturnType

- infer (http://www.typescriptlang.org/docs/handbook/advanced-types.html#type-inference-in-conditional-types)最早出现在此PR (https://github.com/Microsoft/TypeScript/pull/21496) 中，表示在 `extends` 条件语句中待推断的类型变量
- 获取函数类型的返回类型

```
export {}
type ReturnType any> = T extends (...args: any[]) => infer R ? R : any;
function getUserInfo() {
    return { name: "zhufeng", age: 10 };
}

type UserInfo = ReturnType<typeof getUserInfo>;

const userA: UserInfo = {
    name: "zhufeng",
    age: 10
};
```

10.6.3.5 Parameters

- Constructs a tuple type of the types of the parameters of a function type T
- Parameters (http://www.typescriptlang.org/docs/handbook/utility-types.html#parameterst)

```
export {}
type Parameters = T extends (...args: infer R) => any ? R : any;

type T0 = Parameters<() => string>;
type T1 = Parameters<(s: string) => void>;
type T2 = Parameters<((arg: T) => T)>;
```

10.6.3.6 InstanceType

- 获取构造函数类型的实例类型
- InstanceType (http://www.typescriptlang.org/docs/handbook/utility-types.html#instancetypet)

```
type Constructor = new (...args: any[]) => any;
type ConstructorParameters = T extends new (...args: infer P) => any ? P : never;
type InstanceType = T extends new (...args: any[]) => infer R ? R : any;

class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    getName() { console.log(this.name) }
}

type constructorParameters = ConstructorParameters<typeof Person>;
let params: constructorParameters = ['zhufeng']

type Instance = InstanceType<typeof Person>;
let instance: Instance = { name: 'zhufeng', getName() { } };
```

10.6.3.7 infer+分布式

- distributive-conditional-types (https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-8.html#distributive-conditional-types)
- 「Distributive conditional types」主要用于拆分 extends 左边部分的联合类型
- 「Distributive conditional types」是由「naked type parameter」构成的条件类型。而「naked type parameter」表示没有被 Wrapped 的类型（如：Array、[T]、Promise 等都是不是「naked type parameter」）。「Distributive conditional types」主要用于拆分 extends 左边部分的联合类型，举个例子：在条件类型 T extends U ? X : Y 中，当 T 是 A | B 时，会拆分成 A extends U ? X : Y | B extends U ? X : Y；
- 利用在逆变位置上，同一类型变量的多个候选类型将会被推断为交叉类型的特性 (https://github.com/Microsoft/TypeScript/pull/21496)
- tuple转union

```
type ElementOf = T extends Array ? E : never;

type TTuple = [string, number];

type ToUnion = ElementOf;
```

```

type T1 = { name: string };
type T2 = { age: number };

type UnionToIntersection = T extends { a: (x: infer U) => void; b: (x: infer U) => void } ? U : never;
type T3 = UnionToIntersectiona: (x: T1) => void; b: (x: T2) => void }>;
```

10.7 内置工具类型

- TS 中内置了一些工具类型来帮助我们更好地使用类型系统
- utility-types ([utility-types](https://github.com/piotrwitek/utility-types))
- TypeScript中增加了对映射类型修饰符的控制
- 具体而言，一个 `readonly` 或 `?` 修饰符在一个映射类型里可以用前缀 `+` 或 `-`来表示这个修饰符应该被添加或移除

符号 含义 +? 变为可选 -? 变为必选

10.7.1 Partial

- Partial 可以将传入的属性由非可选变为可选，具体使用如下：

```
type Partial = { [P in keyof T]?: T[P] };

interface A {
  a1: string;
  a2: number;
  a3: boolean;
}

type aPartial = Partial;

const a: aPartial = {};
```

10.7.2 类型递归

```
interface Company {
    id: number
    name: string
}

interface Person {
    id: number
    name: string
    company: Company
}
type DeepPartial = {
    [U in keyof T]?: T[U] extends object
    ? DeepPartial
    : T[U]
};

type R2 = DeepPartial
```

10.7.3 Required

- Required 可以将传入的属性中的可选项变为必选项，这里用了 -? 修饰符来实现。

```
interface Person{
  name:string;
  age:number;
  gender?:'male'|'female';
}

let p:Required = {
  name:'zhufeng',
  age:10,

}
```

10.7.4 Readonly

- Readonly 通过为传入的属性每一项都加上 readonly 修饰符来实现。

```
interface Person{
  name:string;
  age:number;
  gender?:'male'|'female';
}

let p:Readonly = {
  name:'zhufeng',
  age:10,
  gender:'male'
}
p.age = 11;
```

10.7.5 Pick

- Pick 能够帮助我们从传入的属性中摘取某一项返回

```
interface Animal {
  name: string;
  age: number;
  gender:number
}

interface Person {
    name: string;
    age: number;
    married: boolean
}
function pick<T, K extends keyof T>(obj: T, keys: K[]): Pick<T, K> {
    const result: any = {};
    keys.map(key => {
        result[key] = obj[key];
    });
    return result
}
let person: Person = { name: 'zhufeng', age: 10, married: true };
let result: Pick'name' | 'age'> = pick'name' | 'age'>(person, ['name', 'age']);
console.log(result);
```

10.7.6 Record

- Record 是 TypeScript 的一个高级类型
- 他会将一个类型的所有属性值都映射到另一个类型上并创造一个新的类型

```
type Record = {
    [P in K]: T;
};
```

```
function mapObject<K extends string | number, T, U>(obj: Record, map: (x: T) => U): Record<K, U> {
    let result: any = {};
    for (const key in obj) {
        result[key] = map(obj[key]);
    }
    return result;
}
let names = { 0: 'hello', 1: 'world' };
let lengths = mapObject(names, (s: string) => s.length);
console.log(lengths);
```

```
type Point = 'x' | 'y';
type PointList = Recordvalue: number }>
const cars: PointList = {
    x: { value: 10 },
    y: { value: 20 },
}
```

10.8 自定义高级类型

- utility-types (https://github.com/piotrwitek/utility-types)

10.8.1 Proxy

```
type Proxy = {
    get(): T;
    set(value: T): void;
}
type Proxify = {
    [P in keyof T]: Proxy
}
function proxify<T>(obj: T): Proxify<T> {
    let result = {} as Proxify;
    for (const key in obj) {
        result[key] = {
            get: () => obj[key],
            set: (value) => obj[key] = value
        }
    }
    return result;
}
let props = {
    name: 'zhufeng',
    age: 10
}
let proxyProps = proxify(props);
console.log(proxyProps);

function unProxify<T>(t: Proxify): T {
    let result = {} as T;
    for (const k in t) {
        result[k] = t[k].get();
    }
    return result;
}

let originProps = unProxify(proxyProps);
console.log(originProps);
```

10.8.2 SetDifference

- SetDifference (same as Exclude)

```
export type SetDifference = A extends B ? never : A;
```

10.8.3 Omit

- Exclude 的作用是从 T 中排除出可分配给 U的元素.
- Omit
- Omit = Exclude + Pick

```
export type Omit = Pick>;
```

10.8.4 Diff

```
export type Diff = Pick<
  T,
  SetDifference
>;
```

10.8.5 Intersection

```
export type Intersection = Pick<
  T,
  Extract & Extract
>;
```

10.8.6 Overwrite

- Overwrite
- mapped-types (https://github.com/piotrwitek/utility-types/blob/master/src/mapped-types.ts)

```
export type Overwrite<
  T extends object,
  U extends object,
  I = Diff & Intersection
> = Pick;

type Props = { name: string; age: number; visible: boolean };
type NewProps = { age: string; other: string };

type ReplacedProps = Overwrite;
```

10.8.7 Merge

- Merge
- Merge

```
type O1 = {
  id: number;
  name: string;
};

type O2 = {
  id: number;
  age: number;
};

type Compute = A extends Function ? A : { [K in keyof A]: A[K] };

type R1 = Computex: "x" } & { y: "y" }>;
type Merge = Compute<
  O1 & Omit
>;

type R2 = Merge;
```

10.8.8 Mutable

- 将 **T** 的所有属性的 `readonly` 移除

```
type Mutable = {
  -readonly [P in keyof T]: T[P]
}
```

10.9 面试题综合实战

- `infer` 关键字就是声明一个类型变量,当类型系统给足条件的时候类型就会被推断出来
- typescript_zh (https://github.com/LeetCode-OpenSource/hire/blob/master/typescript_zh.md)
- codesandbox (https://codesandbox.io/s/4tmtp)

```
interface Action {
    payload?: T;
    type: string;
}

class EffectModule {
    count = 1;
    message = "hello!";

    delay(input: Promise): Promise> {
        let action: Promise> =  input.then(i => ({
            payload: `hello ${i}!`,
            type: 'delay'
        }));
        return action;
    }

    setMessage(action: Action<Date>): Action {
        let action2: Action = {
            payload: action.payload!.getMilliseconds(),
            type: "set-message"
        };
        return action2;
    }
}

type methodsPick = { [K in keyof T]: T[K] extends Function ? K : never }[keyof T];

type asyncMethod = (input: Promise) => Promise>
type asyncMethodConnect = (input: T) => Action
type syncMethod = (action: Action) => Action
type syncMethodConnect = (action: T) => Action

type EffectModuleMethodsConnect = T extends asyncMethod
    ? asyncMethodConnect
    : T extends syncMethod
    ? syncMethodConnect
    : never
type EffectModuleMethods = methodsPick

type Connect = (module: EffectModule) => {
    [M in EffectModuleMethods]: EffectModuleMethodsConnect
}
type Connected = {
    delay(input: number): Action;
    setMessage(action: Date): Action;
};
const connect: Connect = (m: EffectModule): Connected => ({
    delay: (input: number) => ({
        type: 'delay',
        payload: `hello 2`
    }),
    setMessage: (input: Date) => ({
        type: "set-message",
        payload: input.getMilliseconds()
    })
});

export const connected: Connected = connect(new EffectModule());
```

11. 模块VS命名空间

- namespace-and-module (https://blog.higan.me/namespace-and-module-in-typescript/)**11.1 模块#****11.1.1 全局模块#**
- 在默认情况下，当你开始在一个新的 TypeScript 文件中写下代码时，它处于全局命名空间中
- 使用全局变量空间是危险的，因为它会与文件内的代码命名冲突。我们推荐使用下文中将要提到的文件模块

foo.ts

```
const foo = 123;
```

bar.ts

```
const bar = foo;
```

** 11.1.2 文件模块 #**

- 文件模块也被称为外部模块。如果在你的 TypeScript 文件的根级别位置含有 import 或者 export，那么它会在这个文件中创建一个本地的作用域
- 模块是TS中外部模块的简称，侧重于代码和复用
- 模块在期自身的作用域里执行，而不是在全局作用域里
- 一个模块里的变量、函数、类等在外部是不可见的，除非你把它导出
- 如果想要使用一个模块里导出的变量，则需要导入

```
export const a = 1;
export const b = 2;
export default 'zhufeng';
```

```
import name, { a, b } from './1';
console.log(name, a, b);
```

** 11.1.3 模块规范 #**

- AMD：不要使用它，它仅能在浏览器工作；
- SystemJS：这是一个好的实验，已经被 ES 模块替代；
- ES 模块：它并没有准备好。
- 使用 module: commonjs 选项来替代这些模式，将会是一个好的主意

** 11.2 命名空间 #**

- 在代码量较大的情况下，为了避免命名空间冲突，可以将相似的函数、类、接口放置到命名空间内
- 命名空间可以将代码包裹起来，只对外暴露需要在外部访问的对象，命名空间内通过 `export`向外导出
- 命名空间是内部模块，主要用于组织代码，避免命名冲突

** 11.2.1 内部划分 #**

```
export namespace zoo {
    export class Dog { eat() { console.log('zoo dog'); } }
}
export namespace home {
    export class Dog { eat() { console.log('home dog'); } }
}
let dog_of_zoo = new zoo.Dog();
dog_of_zoo.eat();
let dog_of_home = new home.Dog();
dog_of_home.eat();
```

```
import { zoo } from './3';
let dog_of_zoo = new zoo.Dog();
dog_of_zoo.eat();
```

** 11.2.2 原理 #**

- 其实一个命名空间本质上一个对象，它的作用是将一系列相关的全局变量组织到一个对象的属性

```
namespace Numbers {
    export let a = 1;
    export let b = 2;
    export let c = 3;
}
```

```
var Numbers;
(function (Numbers) {
    Numbers.a = 1;
    Numbers.b = 2;
    Numbers.c = 3;
})(Numbers || (Numbers = {}));
```

** 11.3 文件，模块与命名空间 #**** 11.3.1 文件和模块 #**

- 每个 module都不一样 src\table1.ts

```
export module Box{
  export class Book1{}
}
```

src\table2.ts

```
export module Box{
    export class Book1{}
}
```

src\table3.ts

```
export module Box{
    export class Book1{}
}
```

** 11.3.2 空间 #**

- namespace 和 module 不一样，namespace 在全局空间中具有唯一性

src\table1.ts

```
namespace  Box{
    export class Book1{}
}
```

src\table2.ts

```
namespace  Box{
    export class Book1{}
}
```

src\table3.ts

```
namespace  Box{
    export class Book1{}
}
```

** 11.3.3 文件 #**

- 每个文件是独立的

src\table1.ts

```
export class Book1 { }
```

src\table2.ts

```
export class Book1 { }
```

src\table3.ts

```
export class Book1 { }
```

12.类型声明

- 声明文件可以让我们不需要将JS重构为TS，只需要加上声明文件就可以使用系统
- 类型声明在编译的时候都会被删除，不会影响真正的代码
- 关键字 declare 表示声明的意思,我们可以用它来做出各种声明:

```
declare var 声明全局变量
declare function 声明全局方法
declare class 声明全局类
declare enum 声明全局枚举类型
declare namespace 声明(含有子属性的)全局对象
interface 和 type 声明全局类型
```

12.1 普通类型声明

```
declare let name: string;
declare let age: number;
declare function getName(): string;  //方法
declare class Animal { name: string }
console.log(name, age);
getName();
new Animal();
export default {};
```

声明jQuery对象

```
declare const $: (selector: string) => {
    click(): void;
    width(length: number): void;
};
$('#root').click();
console.log($('#root').width);
```

12.2 外部枚举

- 外部枚举是使用 declare enum定义的枚举类型
- 外部枚举用来描述已经存在的枚举类型的形状

```
declare enum Seasons {
    Spring,
    Summer,
    Autumn,
    Winter
}

let seasons = [
    Seasons.Spring,
    Seasons.Summer,
    Seasons.Autumn,
    Seasons.Winter
];
```

declare 定义的类型只会用于编译时的检查，编译结果中会被删除。上例的编译结果如下

```
var seasons = [
    Seasons.Spring,
    Seasons.Summer,
    Seasons.Autumn,
    Seasons.Winter
];
```

也可以同时使用 declare 和 const

```
declare const enum Seasons {
    Spring,
    Summer,
    Autumn,
    Winter
}

let seasons = [
    Seasons.Spring,
    Seasons.Summer,
    Seasons.Autumn,
    Seasons.Winter
];
```

编译结果

```
var seasons = [
    0 ,
    1 ,
    2 ,
    3
];
```

12.3 namespace

- 如果一个全局变量包括了很多子属性，可能使用namespace
- 在声明文件中的 namespace表示一个全局变量包含很多子属性
- 在命名空间内部不需要使用 declare 声明属性或方法

```
declare namespace ${
    function ajax(url:string,settings:any):void;
    let name:string;
    namespace fn {
        function extend(object:any):void;
    }
}
$.ajax('/api/users',{});
$.fn.extend({
    log:function(message:any){
        console.log(message);
    }
});
export {};
```

12.4 类型声明文件

- 我们可以把类型声明放在一个单独的类型声明文件中
- 可以在类型声明文件中使用类型声明
- 文件命名规范为 *.d.ts
- 观看类型声明文件有助于了解库的使用方式

** 12.4.1 jquery.d.ts #**

typings\jquery.d.ts

```
declare const $:(selector:string)=>{
    click():void;
    width(length:number):void;
}
```

** 12.4.2 tsconfig.json #**

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "ES2015",
    "outDir":"lib"
  },
  "include": [
    "src/**/*",
    "typings/**/*"
  ]
}
```

** 12.4.3 test.js #**

src\test.ts

```
$('#button').click();
$('#button').width(100);
export {};
```

12.5 第三方声明文件

- 可以安装使用第三方的声明文件
- @types是一个约定的前缀，所有的第三方声明的类型库都会带有这样的前缀
- JavaScript 中有很多内置对象，它们可以在 TypeScript 中被当做声明好了的类型
- 内置对象是指根据标准在全局作用域（Global）上存在的对象。这里的标准是指 ECMAScript 和其他环境（比如 DOM）的标准
- 这些内置对象的类型声明文件，就包含在TypeScript 核心库的类型声明文件 (https://github.com/Microsoft/TypeScript/tree/master/src/lib)中

** 12.5.1 使用jquery #**

```
cnpm i jquery -S
```

```
import * as jQuery from 'jquery';
jQuery.ajax('/user/1');
```

** 12.5.2 安装声明文件 #**

```
cnpm i @types/jquery -S
```

** 12.5.3 自己编写声明文件 #**

- 模块查找规则 (https://www.tslang.cn/docs/handbook/module-resolution.html)
- node_modules@types\jquery/index.d.ts
- 我们可以自己编写声明文件并配置 `tsconfig.json`

** 12.5.3.1 index.d.ts #**
`types\jquery\index.d.ts`

```
declare function jQuery(selector:string):HTMLElement;
declare namespace jQuery{
  function ajax(url:string):void
}
export default jQuery;
```

** 12.5.3.2 tsconfig.json #**

- 如果配置了 `paths`,那么在引入包的的时候会自动去 `paths`目录里找类型声明文件
- 在 tsconfig.json 中，我们通过 `compilerOptions` 里的 `paths` 属性来配置路径映射
- `paths`是模块名到基于 `baseUrl`的路径映射的列表

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "*":["types/*"]
    }
}
```

```
import $ from "jquery";
$.ajax('get');
```

** 12.5.4 npm声明文件可能的位置 #**

- node_modules/jquery/package.json
    - "types":"types/xxx.d.ts"
- node_modules/jquery/index.d.ts
- node_modules/@types/jquery/index.d.ts
- typings\jquery\index.d.ts

** 12.5.5 查找声明文件 #**

- 如果是手动写的声明文件，那么需要满足以下条件之一，才能被正确的识别
- 给 `package.json` 中的 `types` 或 `typings` 字段指定一个类型声明文件地址
- 在项目根目录下，编写一个 `index.d.ts` 文件
- 针对入口文件（package.json 中的 main 字段指定的入口文件），编写一个同名不同后缀的 `.d.ts` 文件

```
{
    "name": "myLib",
    "version": "1.0.0",
    "main": "lib/index.js",
    "types": "myLib.d.ts",
}
`
```

- 先找myLib.d.ts
- 没有就再找index.d.ts
- 还没有再找lib/index.d.js
- 还找不到就认为没有类型声明了

12.6 扩展全局变量的类型

** 12.6.1 扩展局部变量类型 #**

```
declare var String: StringConstructor;
interface StringConstructor {
    new(value?: any): String;
    (value?: any): string;
    readonly prototype: String;
}
interface String {
    toString(): string;
}
```

```
interface String {
    double():string;
}

String.prototype.double = function(){
    return this+'+'+this;
}
console.log('hello'.double());

interface Window{
    myname:string
}
console.log(window.myname);
```

** 12.6.2 模块内全局扩展 #**

types\global\index.d.ts

```
declare global{
    interface String {
        double():string;
    }
    interface Window{
        myname:string
    }
}

export  {}
```

12.7 合并声明

- 同一名称的两个独立声明会被合并成一个单一声明
- 合并后的声明拥有原先两个声明的特性

关键字 作为类型使用 作为值使用 class yes yes enum yes yes interface yes no type yes no function no yes var,let,const no yes

- 类既可以作为类型使用，也可以作为值使用，接口只能作为类型使用

```
class Person{
    name:string=''
}
let p1:Person;
let p2 = new Person();

interface Animal{
    name:string
}
let a1:Animal;
let a2 = Animal;
```

** 12.7.1 合并类型声明 #**

- 可以通过接口合并的特性给一个第三方为扩展类型声明

use.js

```
interface Animal{
    name:string
}
let a1:Animal={name:'zhufeng',age:10};
console.log(a1.name);
console.log(a1.age);
```

types\animal\index.d.ts

```
interface Animal{
    age:number
}
```

** 12.7.2 使用命名空间扩展类 #**

- 我们可以使用 namespace 来扩展类，用于表示内部类

```
class Form {
  username: Form.Item='';
  password: Form.Item='';
}

namespace Form {
  export class Item {}
}
let item:Form.Item = new Form.Item();
console.log(item);
```

** 12.7.3 使用命名空间扩展函数 #**

- 我们也可以使用 namespace 来扩展函数

```
function greeting(name: string): string {
    return greeting.words+name;
}

namespace greeting {
    export let words = "Hello,";
}

console.log(greeting('zhufeng'))
```

** 12.7.4 使用命名空间扩展枚举类型 #**

```
enum Color {
    red = 1,
    yellow = 2,
    blue = 3
}

namespace Color {
    export const green=4;
    export const purple=5;
}
console.log(Color.green)
```

** 12.7.5 扩展Store #**

```
import { createStore, Store } from 'redux';
type StoreExt = Store & {
    ext: string
}
let store: StoreExt = createStore(state => state);
store.ext = 'hello';
```

12.8 生成声明文件

- 把TS编译成JS后丢失类型声明，我们可以在编译的时候自动生成一份JS文件

```
{
  "compilerOptions": {
     "declaration": true,
  }
}
```

12.9 类型声明实战

- events (https://nodejs.org/api/events.html)

```
npm link
npm link zf-events
```

** 12.9.1 index.js #**

```
import { EventEmitter } from "zf-events";
console.log(EventEmitter.defaultMaxListeners);
var e = new EventEmitter();
e.on('message', function (text:string) {
    console.log(text)
})
e.emit('message', 'hello');
```

** 12.9.2 index.d.ts #**

```
export type Listener = (...args: any[]) => void;
export type Type = string | symbol

export class EventEmitter {
   static defaultMaxListeners: number;
   emit(type: Type, ...args: any[]): boolean;
   addListener(type: Type, listener: Listener): this;
   on(type: Type, listener: Listener): this;
   once(type: Type, listener: Listener): this;
}
```