1.ts

js超集，主要提供了类型推论和es6的支持

2.优

增强代码可读性和可维护性好

（编译阶段就可发现大部分错误、增强编辑器和ide的功能，包括代码补全，接口提示，跳转到定义，代码重构等）

非常包容

(a.未定义类型，也会自动作出类型推论 b.编译报错，也可生成js c.兼容第三方库,即使第三方库不是ts写的，也可编写单独类型文件供ts读取)

3. TypeScript 编译的时候即使报错了，还是会生成编译结果

tsconfig.json 中配置 noEmitOnError 即可

4.boolean定义布尔值类型，构造函数Boolean创造的是对象不是布尔值

5.任意值：在任意值上访问任何属性都是允许的

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型

6.如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 any 类型而完全不被类型检查

定义的时候有赋值，类型推论为初始赋值类型

6.联合类型

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法

联合类型的变量在被赋值的时候，会根据类型推论的规则推断出一个类型

7.对象的类型——接口

赋值的时候，变量的形状必须和接口的形状保持一致

可选属性 interface ExampleObj { prop? : type}

添加未定义的属性 [propName: string]: any;

一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集

readonly 定义只读属性

只读的约束存在于第一次给对象赋值的时候，而不是第一次给只读属性赋值的时候：

interface Person {

readonly id: number;

name: string;

age?: number;

[propName: string]: any;

}

let tom: Person = {

name: 'Tom',

gender: 'male'

};

tom.id = 89757;//报错

8.数组的类型

组类型定义

number[]

数组泛型 Array<elemType>

用接口表示数组

interface NumberArray {

[index: number]: number;

}

用元组定义类型

let tom: [string, number] = ['Tom', 25];

直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项

不能添加越界元素

9.类数组

不能用普通的数组的方式来描述，而应该用接口

interface Arguments {

[index: number]: any;

length: number;

callee: Function;

}

10.函数的类型

输入多余的（或者少于要求的）参数，是不被允许的

let mySum: (x: number, y: number) => number = function (x: number, y: number): number {

return x + y;

}; => 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。

用接口定义函数的形状---函数表达式场景

interface SearchFunc {

(source: string, subString: string): boolean;

}

let mySearch: SearchFunc;

mySearch = function(source, subString) {

return source.search(subString) !== -1;

}

? 表示可选的参数，可选参数后面不允许再出现必需参数了

参数默认值

TypeScript 会将添加了默认值的参数识别为可选参数，此时就不受「可选参数必须接在必需参数后面」的限制了

lastName: string = 'Cat'

剩余参数 ...items: any[]

重载

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理，并精确表达函数返回type

function reverse(x: number): number;

function reverse(x: string): string;

function reverse(x: number | string): number | string {

if (typeof x === 'number') {

return Number(x.toString().split('').reverse().join(''));

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

return x.split('').reverse().join('');

}

}

11.类型断言

在 tsx 语法（React 的 jsx 语法的 ts 版）中必须使用as 类型。

使用场景：将一个联合类型断言为其中一个类型（联合类型不确定类型时，会访问联合类型共有属性和方法）

将一个父类断言为更加具体的子类

12.声明文件

13.内置对象

14.类型别名

类型别名用来给一个类型起个新名字。

type Name = string;

15.字符串字面量类型

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。

type EventNames = 'click' | 'scroll' | 'mousemove';

function handleEvent(ele: Element, event: EventNames) {

// do something

}

16.枚举

枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等。

17.类

存取器

使用 getter 和 setter 可以改变属性的赋值和读取行为

静态方法

使用 static 修饰符修饰的方法称为静态方法，它们不需要实例化，而是直接通过类来调用

public private 和 protected 访问修饰符

private 修饰的属性或方法是私有的，不能在声明它的类的外部访问,但TypeScript 编译之后的代码中，并没有限制 private 属性在外部的可访问性。

使用 private 修饰的属性或方法，在子类中也是不允许访问的

当构造函数修饰为 private 时，该类不允许被继承或者实例化

protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的

当构造函数修饰为 protected 时，该类只允许被继承：

readonly 和其他访问修饰符同时存在的话，需要写在其后面。

// public readonly name;

抽象类

abstract 用于定义抽象类和其中的抽象方法。

首先，抽象类是不允许被实例化的：

抽象类中的抽象方法必须被子类实现

18. 不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），用 implements 关键字来实现。

interface Alarm {

alert(): void;

}

interface Light {

lightOn(): void;

lightOff(): void;

}

class Car implements Alarm, Light {

alert() {

console.log('Car alert');

}

lightOn() {

console.log('Car light on');

}

lightOff() {

console.log('Car light off');

}

}

接口继承类

为什么 TypeScript 会支持接口继承类呢？

实际上，当我们在声明 class Point 时，除了会创建一个名为 Point 的类之外，同时也创建了一个名为 Point 的类型（实例的类型）

既可以将 Point 当做一个类来用（使用 new Point 创建它的实例）

也可以将 Point 当做一个类型来用（使用 : Point 表示参数的类型）

class Point {

x: number;

y: number;

constructor(x: number, y: number) {

this.x = x;

this.y = y;

}

}

interface Point3d extends Point {

z: number;

}

let point3d: Point3d = {x: 1, y: 2, z: 3};

19. 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。

函数中使用

例：function createArray(value: any): Array<any> {

let result = [];

result[0] = value;

return result;

}

createArray('x')

// function createArray(value:any):Array<any>

此处泛型等同于函数重载，使用时根据参数确定输出类型

function createArr<T>(value:T):T[]{ // createArr<T> T函数参数类型

let arr:T[]=[]

arr.push(value)

return arr

}

createArr('123')// function createArr(value:string):string[]

20. 多个类型参数

21.泛型约束

interface Lengthwise {

length: number;

}

function loggingIdentity<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

22.多个类型参数之间也可以互相约束：

function sum<A extends B,B>(x:A,y:B):B[] {

const arr:B[]=[]

arr.push(x)

arr.push(y)

return arr

}要求 A 继承 B，这样就保证了 A 上不会出现 B 中不存在的字段。

23.泛型接口

interface Funiterface {

<T>(x:T,y:T):T[];

}

let sum:Funiterface=function<T>(x:T,y:T):T[] {

const arr:T[]=[]

arr.push(x)

arr.push(y)

return arr

}

泛型参数提前到接口名上

interface Funiterface<T> {

(x:T,y:T):T[];

}

let sum:Funiterface<number>=function<T>(x:T,y:T):T[] {

const arr:T[]=[]

arr.push(x)

arr.push(y)

return arr

}

泛型参数的默认类型

let sum:Funiterface<T >=function<T=number >(x:T,y:T):T[] {

const arr:T[]=[]

arr.push(x)

arr.push(y)

return arr

}

24.泛型类

class GenericNumber<T> {

zeroValue: T;

add: (x: T, y: T) => T;

}

let myGenericNumber = new GenericNumber<number>();

myGenericNumber.zeroValue = 0;

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

25.声明合并

定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型

26.代码检查方案：typescript-eslint