# 1、ts:js的超级

# 2、优势：

1）类型化的思维方式，使开发更加严谨。提前发现bug，减少bug时间。

2）提高了可读性，使维护和重构代码更加容易。

3）补充了枚举和接口等开发大型应用时js功能的缺失。

# 3、工具和环境

vsCode nodejs ts解析包（nodejs只认js不认ts-- npm install typescript -g）

1. typescript:用来解析TS的工具包，提供了tsc命令，实现了从ts-js的转换
2. 补充：quote设置引号样式，tab设置缩进

# 4、执行代码（两步）

1）tsc hello.ts会生成一个js文件

2）node hello.js

简化步骤：ts-node包 npm i -g ts-node

终端使用ts-node hello.ts

或者直接安装ts-node包，然后直接运行ts-node hello.ts就可以

# 5、基本的数据类：声明变量的时候必须写注解

1）a、原始数据类型：number :整数型和浮点型 +’1’可以将字符串类型的’1’转换为整数类型1

string ：

boolean ：

undifined ：

null：

literal:字面量类型：const num1 = 3；变量num1的类型就是3本身

union：混合类型：let person: string | number | string[]

person = 10;

null和undefined区别：卫生纸有纸筒没有纸（null）没有纸和纸筒（undefined）

b、对象类型：

type IUser = {

age: number;

name: string;

}

interface IUser{

age: number;

name: string;

}

let user1: Iuser = {

age: 10,

name: “1111”

}

c、any和unkown

any动态确定类型

unkown保证类型安全，因为它会校验类型。

d、void undefined never

void：未定义，变量本身不存在，函数最后不写return

undefined:定义了，但是没有赋值

如何定义函数类型为undefined 函数最后return

never:函数永远执行不完，这个时候函数的返回类型就是never

e、类型适配：

let a: any;

a= “String”;

(<string>a).endWith(“c”)

或者 (a as string).endWith(“g”)

f、类

class Person{}

const dell: Person = new Person();

g、函数类型

const getTotal : () => number = () => {return 10}

h、数组

const persons: string[] = [“张”, “王”]

2）约定了数据类型，将来只能赋值同样的数据类型

i、元组

const arr: [string,string, number][] = [[“1”, “2”, 3]]

const arr: [string, number] = [“1”, 3]

# vscode中调试ts代码的配置

开始调试，增加ts调试的配置

# 7、枚举类型

enum Color{

red,

green,

orange

}

1）数字枚举：enum Role {

Reporter,

Developer,

Owner,

Guest

}

console.log(Role.Reporter) 0

enum Role {

Reporter = 1,

Developer, //后面的值递增

Owner,

Guest

}

2）字符串枚举:字符串类型不具备反向映射

enum Str {

str1 = “ddd”,

str2 = “aaa”

}

3）常量枚举

# 8、数组

const names: string[] = [‘zhang’,’wang’]

或者

const names: string[] = new Array(“zhang”,”wang”)

const arr: (number|string)[] = [“1”, 2]

const objArr: ({name: string})[] = [{name: ‘10’}]

# 9、对象

对象的类型注解

let person: { //注意这里是冒号

age: number;

name: string;

sayHi: () => void;

sayName: ()=> **string**;

sayAge: (n1: number) => number;

}

type Person = { //注意这里是等号

name: string

}

interface Person { //注意这里没有冒号

name: string，

sayHi(): string

}

interface和type的区别：interface只能代表一个对象。但是type Person = string是可以的。

interface Max{

name: string,

age:number,

[propName:string]: any //表示属性名为字符串，值为任何值的属性

}

interface SayHi{ //interface定义函数类型 接收string类型的参数，并返回string类型的数据

(word: string): string

}

# 10、函数

函数的类型体现在两个地方，一个是参数，一个是返回值

如果某个参数是可选的，则参数后面跟一个?

function amon(num: number, name?: string) {

//如果不输入name这它的值为undefined;

}

函数可以设置默认值

function amon(num: number, name: string = ‘zhang’) {

//和es6的语法一样。

}

//默认参数和可选参数必须是在参数的末尾

参数解构的数据类型

function({first, second}: {first: string, second:string}): string {}

# 11、class

一组紧密相关的属性和函数的集合。

1）class OnePerson {

age: number;

name: string;

constructor(age: number, name: string) {

this.age = age;

this.name = name;

}

getAge: () => number;

getName: () => string;

}

const onePerson: OnePerson = new OnePerson(10, "zhang");

console.log(onePerson.age)

2）类实现接口

interface Iuser{ //接口定义了类的基本特征

name: string;

age: number;

getName: () => string;

getAge:() => number;

}

class OneUser implements Iuser {

age: number;

name: string;

constructor(name, age) {

this.age = age;

this.name = name;

}

getAge = () => {

return this.age;

}

getName = () => {

return this.name;

}

}

const oneUser: Iuser = new OneUser("zhang", 109)这里的类型可以是OneUser也可以是Iuser

也可以不写。不写的话是OneUser;

console.log(oneUser.age)

3）上面的语法过于繁琐,

我们引入public private protected关键词

class OnePerson {

constructor(public age: number, public name: string) { //等价于定义number和age参数，并且给类定义两个属性number和age，并将其赋值为传入的参数

}

getAge: () => number;

getName: () => string;

}

const onePerson: OnePerson = new OnePerson(10, "zhang");

console.log(onePerson.age)

public:类的内外都能使用

private:只能在类的内部使用

protected:在类及子类内部使用。

注意：接口中定义的属性和方法都是public属性：所以如果想定义类的私有属性，则接口中就不要有相关的属性。如果要实现对private属性的赋值，可以通过getter和setter方法。

1. Readonly

使用readonly关键字修饰该属性是只读的，只能修饰属性不能修饰方法。readonly修饰的属性的值在类的外部是不能修改的。在本类中的其他方法中也是不能修改的（在构造方法中可以修改）

如果在定义readonly修饰的属性的时候，属性如果不指定类型注解，则属性的类型为所赋的默认值，此时在构造函数中就不能重新赋值。

readonly不只可以用在类中，也可以用在对象或者接口中。

# 12、接口

interface Iuse{

age: number;

name: string

}

# 泛型generics

1. 函数的泛型

function getSome<T>(value: T):T{return value}

调用的时候可以省略<T>来简化泛型函数的调用。如：let num<number> = getSome(10);但是当编译器无法推断类型或者推断的类型不准确时，就需要手动输入类型。

let lastArray = <T>(arr: T[]) = []

function getSome<T, K extends keyof T>(value: T, key: K){}

keyof关键字表示值的类型必须是后面的泛型所代表的对象的key的联合类型

1. 接口泛型

interface IdFunc<T> {

id: (val: T) => T

}

1. 泛型工具类型：

a: Partial<Type>用来构造（创建）一个类型，将Type的所有属性设置为可选。

interface Props{name: String, children: String[]}

type PartialProps = Partial<Props>

let obj:PartialProps = {name: “zhang”} //可以不设置children

b: Readonly<Type> 用来够造一个类型，将Type的所有的属性设置为只读

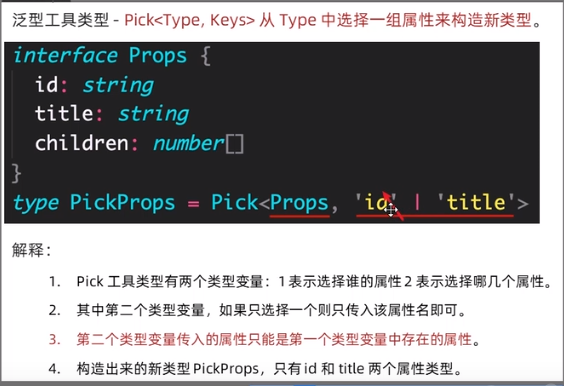
interface Props{name: string}

type ReadonlyProps = Readonly<Props>

let obj:Readonlyprops = {name: “hello”}

obj.name = “wang” //报错，因为name属性是只读的。

c:



d:



# 14、类型注解和类型推断

1）type annotation类型注解：我们来告诉Ts变量是什么类型

type interface类型推断：Ts会自动的尝试分析变量的类型

如果TS能自动分析变量类型，那么我们就什么也不需要做了

如果TS无法分析变量类型的化，我们就需要使用类型注解

# 15、抽象类

abstract class Geom{

width: number,

getType:() {

return “123”

}

abstract nda():number; //抽象方法只定义，不实现，继承类中去实现，而且必须实现。

}

class E extends Geom{

nda: () => 123

}

# Typeof

1. 可以像js中使用一样
2. 可以根据上下文中引用变量或者属性的类型（类型查询的功能）：根据已有变量的值，获取该值的类型，来简化类型书写。

let p = {x:1, y:2}

Function sum(point: typeof p){}

注意typeof不可以获取函数返回值的类型

# 类型兼容

1）Ts采用的结构化类型系统，对应的是标明类型系统（java），比如：两个类的结构相同，或者后者的成员至少与前者的相同，则x兼容y（成员多的可以赋值给少的）

比如：

class Point{

Name: string

}

class Point1{

Name: string，

Age: number

}

Let point:Point = new Point1();

类之间，对象之间，接口之间，或者接口与类之间都是可以进行类型兼容的。

1. 函数之间的类型兼容。

函数之间的类型兼容，需要参数个数，参数类型，和返回值类型

参数个数：参数多的兼容参数少的（参数少的可以赋值给参数多的-和对象正好相反）

返回值类型：如果返回值类型是对象，则遵循对象的类型兼容规则。

# 交叉类型

1）交叉类型&,类似于接口继承，用于组合多个类型为一个类型（常用于对象类型）。

比如：

Interface Person{

Name:string

}

Interface Person1{

Age: number

}

Type Person2 = Person & Person2

Let obj: Person2 = {

Name: “jack”，

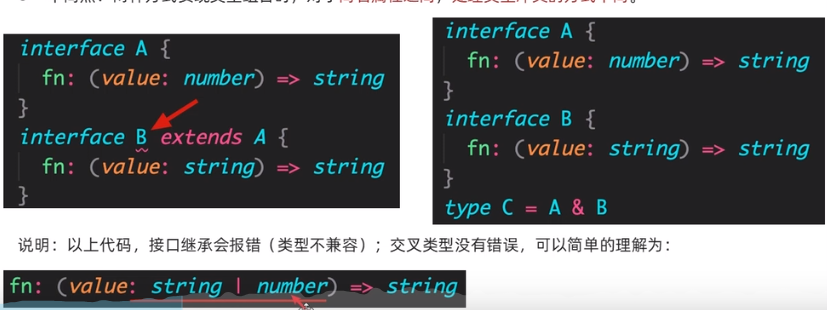
Phone: ‘133’

}

1. 接口类型和交叉类型的比较

相同点：都可以实现对象类型的组合。

不同点：两种方式实现类型组合时，对于同名属性之间，处理类型的冲突的方式不同。



# 索引签名类型

绝大情况下，我们可以在使用对象前就确定对象的结构，并为对象添加准确的类型。

使用场景:当无法确认对象中有哪些属性的时候，（或者说对象中可以出现任意多个属性），此时，就用到索引签明类型了。

intefface AnyObject{

[key: string]: number

}

[key: string]表示只要是string类型的属性名称，都可以出现在对象中。

这样对象就可以出现任意多个属性，key只是一个占位符，可以换成任意合法的变量名称。

# 映射类型

