**TS学习**



类型

类型注解：let  age: number = 18;  
**:number**就是**类型注解**  
作用：为变量添加类型约束，比如上述就是约束变量age的类型是number类型



概述：TS常见的类型可以概括为两类：  
一：JS已有类型  
二：TS新增类型



**（1）已有类型：**  
原始类型  
对象类型

原始类型：number/string/boolean/null/undefined/symbol  
**let  age : number = 18**

数组的写法：  
**let numbers : number [] = [1, 3, 5]  // 类型[]**  
**let numbers : Array<string> = ['a', 'b', 'c'] // Array[类型]**  
**let numbers : (numbers | string) [] = [1, 'a', 3, 'c']  （类型1 | 类型2）[]**对象数组：**先声明一个对象类型，再自定义一个对象数组类型**  
**type  Item = {**  
**id: number**  
**name: string**  
**}**  
**type  Arr= Item[]**  
**const  arr: Arr = [{ id:1,name: 'lisa' }]**



**（2）TS新增类型**  
联合类型，自定义类型，接口，元组，字面量类型，枚举，void，any



**1.联合类型**

由两个或者多个类型组成的类型，表示可以是这些类型中的任意一种  
**let  numbers :  (number | string)[] = [1, 2, 'a', 'b']**



**2.自定义类型**（**类型别名**）**type**  
使用场景：当同一类型(复杂)被多次使用时，可以通过类型别名简化该类型的使用。

比如：  
**type  CustomArray = (number | string) []  
let arr : CustomArray = [1, 2, 'a', 'b']  
let arr1 : CustomArray = [3, 4, 'c', 'd']**注释：  
（1）使用type关键字来创建类型别名，比如上述就是创建了一个**CustomArray**类型来表示**复杂类型(number | string) []**  
（2）类型别名可以是任意合法的变量名称  
（3）创建类型别名后，直接使用该类型别名作为变量的类型注解



**3.函数类型**

有返回值  
**function add (num1: number, num2: number ):number {  
    return num1 + num2  
}  
const add = (num1: number, num2: number): number => {  
    return num1 + num2  
}  
const add: (num1: number, num2: number) => number = (num1, num2) => {  
    return num1 + num2  
}**  
无返回值：此时函数的返回值类型为void  
**function test (name: string) : void {  
    console.log(name)  
}**  
函数可选参数 ？  
**function fn (num1: number, num2 ?: number) : number {  
    return num1 + num2  
}  
console.log( fn(2, 3) )**  
可选参数只能出现在参数列表的最后

**4.对象**  
定义方式：  
   **属性名： 类型**  
    **方法名(形参：形参类型) : 返回值类型///方法名：(形参：形参类型) =>返回值类型**   
**属性分割使用；**  
**let person : {  name: string;  say () : void } = { say () {}}**



**5.接口:  interface**当一个对象类型被多次使用时，一般会使用**接口interface**来描述对象的类型达到复用的目的

继承：如果两个接口之间有相同的属性或者方法，可以将公共的属性或者方法抽离出来，通过继承来实现复用。  
例如：  
**interface  point2D { x: number; y: number }  
interface  point3D { x: number; y: number; z: number }  
  
interface point2D { x: number; y: number }  
interface Point3D  extends Point2D {z: number  }**

接口定义完成后直接使用接口名称作为变量的类型。因为每一行只有一个属性类型，所以属性类型后没有；  
**interface IPerson {  
    name: string  
    age: number  
    say():void  
}  
let person : IPerson = {  
    name: 'lisa'  
    age: 18  
    say(){}  
}**

**6.元组**  
它能够准确的知道包含了多少个元素，以及索引对应的类型



**7.类型推论：**如果么有指定类型，那么它会根据初始化值推断出类型，此时我们可以**省略不写类型注解**

常见场景包括两种时候：  
(1)**声明变量并初始化时 let  num = 18**  
(2)**决定函数返回值时**：**function  add(num1: number,num2:number){return num1 + num2**}



**8.类型断言：as**解释：  
（1）**使用as 关键字实现类型断言**（2）as后面的类型是一个更加具体的类型

比如：**const  link =  document.getElementById('link')  as  HTMLAnchorElement**



**9.字面量类型**解释：常量 let的值可以改变，但是const的值不能改变，所以这种不能改变的类型就被称为字面量类型，  
**除了字符串意外任意的JS字面量比如数字都可以作为**

类型使用  
比如 **let  age: 18 = 18**  
应用场景：**字面量类型配合联合类型一起使用**，常用于可选值列表  
**function  changeDirection ( direction: 'up' | 'down' | 'left' | 'right'){  
    console.log(direction)  
}**



**10.枚举：enum**  
访问枚举对象**：通过 . 访问**  
定义一组命名常量，它描述一个值，该值可以是这些命名常量中的任意一个

**enum  Derection { Up, Down, Left, Right}  
function  changeDirection ( direction: Direction)  {  
    console.log(direction)  
}**

**11.any**



**12.typeof**  
TS中的typeof可以在类型 上下文中引用变量或对象属性的类型，**用作类型查询**

比如：  
**let p = {x:1, y: 2}  
function  formatPoint (point: typeof p){}  
formatPoint()**  
注意：typeof出现在类型注解的位置的所处环境就是**类型上下文**  
**typeof只能用来查询变量或者对象属性的类型**



**13.类  
readonly**:表示只读，用来**防止在构造函数之外的地方对属性进行赋值**，也就是说在class中使用readonly时，  
只能在constructor中修改被readonly修饰的属性。

例如：  
**class  Person {  
    readonly  age: number = 18  
    constructor (age: number ) {  
        this.age = age  
    }  
    setAge (age: number) {  
        this.age = age // 报错，不能修改readonly属性  
    }  
}**



类型兼容性

接口兼容性

函数兼容性  
解释：  
    1.参数邵的可以赋值给参数多的  
**type F1 = （a: number）=> void  
type F2 = (a: number, b: number) => void  
let f1: F1  
let f2: F2 = f1**



**14.交叉类型：**  
功能类似于接口继承，**用于组合多个类型为一个类型（常用于对象类型）**  
**interface Person ( name: string )  
interface  Contact { phone:  string }  
type PersonContact = Person & Contact  
let p : PersonContact = {  
    name: 'lisa',  
    phone: '18101016666'  
}**

交叉类型和接口继承的对比  
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。处理类型充入的方式不同  
**interface A { fn: (value : number ) => string}  
interface B extends  A { fn: (value : string ) => string}  
  
  
interface A { fn: (value : number ) => string}  
interface B  { fn: (value : string ) => string}  
type C = A&B  
let cc: CC = {  
    fn (value: number | string) {  
         return ''  
     }  
}  
let c: CC  
c.fn()  
fn : (value: string | number) => string**



**15.泛型**调用泛型函数  
**function  fn <Type> ( value: Type ): Type ( return value )  
const num = fn < number > (10)**调用时<>内容可以省略,因为TS的类型推断机制，比如  
**const num = fn (10)**



**泛型约束（收缩类型的可选范围）**解释：  
通过extends关键字使用，为泛型添加约束

实现方式：  
方式一：**指定为更加具体的类型**  
**function  fn <Type> (value:Type[]):Type{  
    console.log(value.length)// 添加约束类型为数组后就可以访问参数的length属性了  
    return value  
}**  
方式二：**添加约束**  
**interface  ILength { length: number  }  
function  fn<Type extends  ILength> (value:Type): Type{  
    console.log(value.length)  
    return value  
}**

**多个泛型变量**  
泛型的类型变量可以有多个，而且变量之间可以相互约束，相互约束的实现通过**keyof**实现  
**function  getProps<Type, Key  extends  keyof  Type>(obj:Type, key: Key) {  
   return obj[Key]  
}  
let  person = { name: 'lisa', age；18}  
getProps( person, name )**解释：  
keyof关键字接收一个对象类型，生成其键名称（可能为字符串或者数字）的联合类型

**泛型接口：**在**接口名称**的后面添加**<类型变量>**就变成了泛型接口  
接口配合泛型来增强接口的灵活性  
**interface  IdFunc<Type>{  
    id: (value:Type) => Type  
    ids: () => Type[]  
}  
let obj:IdFunc<number> = {  
    id (value) {  return value }  
ids () { return  [1, 2, 3] }  
}**

**泛型类**  
创建泛型类：**在类名后面添加<类型变量>**  
**class  GenericNUmber <numType> {  
    defaultValue: numType  
    add: (x: NumType, y: NUmType ) => NumType  
}**

**泛型工具类型**：TS提供了一些常用的工具类型，  
**（1）Partial<Type>用来构建一个类型，将Type的所有属性设置为可选**  
**interface  Props {  
   name： string  
    age :number  
}  
type PartialProps = Partial<Props>**  
**（2）Readonly<Type>用来构造一个类型**，将Type的**所有属**性都设置**为readonly  
interface  Props{  
  age: number  
  id: string  
}  
type  ReadonlyProps = Readonly<Props>  
let props: ReadronlyProps = {age: 18, id:1}  
props.id = 2 // 无法重新赋值，因为id是只读属性  
（3）Pick<Type, keys>从Type中选择一组属性来构造新类型**  
**interface  Props {  
     id:string  
     title: string  
     children: number []  
}  
type PickProps = Pick<Props, 'id' | 'title'>  
（4）Record<Keys, Type>构造一个对象类型，属性键为Keys，属性类型为Type**  
**type  RecordObj = Record<'a' | 'b' | 'c', string[]>  
    let obj: RecordObj = {  
        a: ['1'],  
        b: ['2'],  
        c: ['3']  
    }**

**16.索引签名类型：**  
使用场景：当你在使用对象前无法确定对象中有哪些属性时就可以使用索引签名类型  
**interface  AnyObject  {  
    [ key: string ]: number  
}  
let  obj: AnyObject = {  
    a:1  
}**



**17.映射类型：**  
基于旧类型创建新类型）（对象类型），减少重复，提升开发效率  
解释：  
（1）映射类型是**基于索引签名类型**的，所以该语法类似于索引签名类型,也使用[]  
（2）**Key  in PropKeys**表示Key可以是PropsKeys联合类型中的任意一个，类似于for in  
（3）使用映射类型创建的新对象类型Type2和Type1结构完全相同  
（4）注意：**映射类型只能在类型别名中使用，不能在接口中使用**

**type  PropsKeys = 'x' | 'y' | 'z'  
type Type1 = { x : number;  y : number; z : number }**  
可以替换成  
**type  PropsKeys = 'x' | 'y' | 'z'  
type  Type1 = { [ Key in  PropKeys ] : number }**

根据对象类型来创建  
**type  Props  = {  a: number; b: number ; c: number }  
 type type3 = { [key in  keyof  Props] : number}**  
解释：  
先执行**keyof**  Props获取到对象类型Props中所有键的联合类型即： 'a', 'b', 'c'  
然后Key **in**就表示key可以是Props中的任意一个

泛型工具类型的实现，(基于映射类型实现)  
**Partical<Type>  
type Partial<T> = {  
    [  P  in  keyof  T ]?: T[P]  
}  
type  Props = { a:number;  b:string; c: boolean }  
type  PartialProps = Partial<Props>**解释：  
**（1） keyof T** 表示获取Props的所有键  
（2）在**[]后面添加？**表示将这些属性变成可选的，以此来实现Partial的功能  
（3）冒号后面的**T[P]**表示获取T中每个键对应的类型。比如，如果是‘a’则表示类型是number，如果是'b'则表示类型为string  
（4）最终，新类型PartialProps和旧类型Props结构完全相同，只是让所有类型都变成了可选

**索引查询类型**：**T[P],**用来查询属性的类型  
**（1）查询单个索引**  
**type Props = { a: number; b: string; c: boolean }**  
**type  type4 = Props['a']**  
解释：**Props['a']表示查询类型Props中属性**‘a’对应的类型number  
  
**（2）同时查询多个索引**：  
**type  type5 = Props['a' | 'b']**  
或者  
**type  type6 = Props[keyof  Props]**



在react中使用TS



1.使用CRA创建支持TS的项目

npx  create-react-app 项目名称 --template  typescript



2.TS配置文件tsconfig.json

子主题



3.Reatc中的常用类型  
TS项目中推荐使用Typescript实现组件类型校验（代替PropTypes）



函数



1.组件和属性类型

函数组件类型  
**type Props = { name: string; age?: number }  
const Hello:FC<Props> = ({ name,age }) => (  
    <div>我的名字叫{name},年龄{age}</div>  
)**  
可以简化为  
**const  Hello = ({ name, age }:Props) => (  
    <div>我的名字叫{name},年龄{age}</div>  
)**



2.属性默认值

defaultProps  
Hello.defaultProps = {  
  age: 18  
}



3.事件绑定和事件对象

**const onClick = () => {}**  
**const onClick = (e: React.MouseEvent<HTMLButtonElement>**)  
对应的事件对象类型可以将鼠标放在在事件处理程序（e => {}）查看



类class



组件类型

**无props和state**  
**class  C1 extends React.Component {}**  
**// 有props无state**  
**class C2  extends React.Component<Props> {}**  
**// 无props有state**  
**class  C3 extends React.Component <{}, State> {}**  
**// 有props和state**  
**class  C4 extends  React.Component <Props, state> {}**



属性和属性默认值

**type  Props = { name: string; age?: number }**  
**class  Hello  extends  React.Component<Props> {**  
**static  defaultProps: Partial<Props> = {**  
**age: 18**  
**}**  
**render  () {**  
**const  { name,age  } = this.props**  
**return <div>my  name is {name}, age {age}<div>**  
**}**  
**}**



状态和事件

**type  State = { count: number }**  
**class  Counter  extends  React.Component<{}, State> {**  
 **state: State = {**  
**count: 0**  
**}**  
**onAdd = （） => {**  
**count: this.state.count + 1**  
**}**  
**return (**  
**<div>**  
**<button onClick={this.onAdd}>+1</button>**  
**</div>**  
**)**  
**}**

react-app-env.d.ts  
React的默认的类型声明文件  
**三斜线指令**：指定依赖的其他类型声明文件，types表示依赖的类型声明文件包的名称  
解释：  
告诉TS帮我们加载react-scripts这个包提供的类型声明  
**react-scripts的类型声明文件包含了两部分类型**  
**1.react,rect-dom,node的类型**  
**2.图片样式等模块的类型，以允许在代码中导入图片，svg等文件**



**类型声明文件**：  
TS中有两种文件类型：  
（1）**.ts文件**   
（2）**.d.ts文件**

.ts文件是implementation代码实现文件  
（1）既包含类型信息又可以执行代码  
（2）可以被编译为.js文件然后执行

.d.ts文件是declaration类型声明文件  
（1）**只包含类型信息**的类型声明文件  
（2）**不会生成.js文件**，仅用于提供类型信息  
（3）用途：为JS提供类型信息



使用

**1.内置类型声明文件**  
TS为JS运行时可用的所有便准化内置API都提供了声明文件

**2.第三方库的类型声明文件**第三方库的类型声明文件有两种存在形式：  
（1）库自带声明文件  
（2）由DefinitelyTyped提供  
DefinitelyTyped是一个github仓库，用来提供高质量TS类型声明  
包的名称格式为：@types/\* 比如@types/lodash  
typescriptlang.org/dt/search?search=react

**自定义类型声明文件**  
应用场景：  
**1.项目中共享类型**    
  如果多个。ts文件中都用到同一个类型，此时可以创建.d.ts文件提供该类型，实现类型共享  
.d.ts文件中使用export导出，在需要使用共享类型的ts文件中使用import导入  
**2.为已有JS文件提供类型声明**  
**在导入.js文件时，TS会自动加载与js同名的.d.ts文件,而不是创建一个新的变量**1.对于**type,interface**等这些**明确就是TS类型的**可以**省略declare**关键字  
2.对于let， function等**具有双重含义**的即既可以在js中使用又可以在TS中使用的。应该**使用declare**关键字



脑图模式

字数：16/9114

主题数：1/78

主题数限时不限制