TS

TS基础类型定义

Ts声明类型以 ':类型' 声明 JavaScript | 复制代码

```
let isDone:boolean = false
let age:number = 10
let firstName:string = 'Tly'
let msg:string = `hell ${firstName}`
let u:undefind = undefind
let n:null = null
```

undefined 可以赋值给任意类型

:any 声明这是一个任意类型 避免使用any 全部使用 any 可以直接使用出门右转使用js

联合类型也就是'或者'

JavaScript | 复制代码

```javascript
let numOrString:number | string
//可以是字符串
numOrString = 'abc'
// 也可以是number
numOrString = 123
```

注意:

使用联合类型 比如他可以数字或字符串时

在使用这个变量时

只能使用数字和字符串共有方法

如两种类型共有的方法 .tostring

如使用 .length 字符串的方法就会报错

如要访问其中一个全部的方法

可先使用类型断言 使用方法如下

类型断言 as

JavaScript | 复制代码

```javascript
function getLength(input: string | number):number {
  const str = input as string
  if (str.length) {
    return str.length
  }else{
    const number = input as number
    return number.tostring().length
  }
}
```

这个例子中 input 可能是数字或字符串我们便不能使用 .length

我们可以先使用类型断言 as 把此入参断言为字符串再访问其 .length

注意:

类型断言不是把这个类型强制转换为另一个类型

只是使TS把这个类型标记为你断言的类型

从而使用该类型的方法不会报错

类型推断 JavaScript | 复制代码

```
//type guard
function getLength2(input: string | number) : number {
  if (typeof input === 'string') {
    return input.tength
  } else {
    return input.toString().length
  }
}
```

这个例子

TS会再从你的检查类型的语句中判断出这个类型是什么

从而你可以访问这个类型的所有方法

数组定义

数组定义:string[] JavaScript | 复制代码

```
let arrOfNumbers: number[] = [1, 2, 3]
arrOfNumbers.push(3)
```

例子：声明一个 string 数组

这个数组由 string 组成不能出现其他类型

注意:

定义完类型以后

如使用 push 等数组方法

不能添加定义类型以外的元素

如定义 :string[] 不能在数组添加数字

元组定义:[string,unmber] JavaScript | 复制代码

```
let user: [string,number] = ['Tly',20]
```

元组定义 :[string,unmber]

规定数组第一项第二项是什么类型

并且限定数组元素个数

数组 delet push 之类方法 可以正常操作数组

操作必须是定义内的类型

:[string,unmber]这个定义出的数组不能添加string 和unmber 以外的类型

Interface接口定义 也就是obj类型

定义: JavaScript | 复制代码

```
Interface Person{
  name: string;
  age: number;
}

let viking: Person{
  name:'tly'
  age: 20
}
```

例子

用 interface person 声明一个对象类型

下边变量用 :person 接收此类型

对象类型相同 且 组成元素不能多或少

可选属性？ JavaScript | 复制代码

```
Interface Person{
  name: string;
  age?: number;
}

let viking: Person{
  name:'tly'
}
```

加问号 创建对象时 属性为选填

只读属性 readonly JavaScript | 复制代码

```
Interface Person{
  readonly id: number;
  name: string;
  age?: number;
}

let viking: Person{
  id:1
  name:'tly'
}
viking.id = 222 //此时为报错因为有readonly属性
```

在 interface 属性前添加 readonly 代表此属性为只读

函数

函数:定义类型 JavaScript | 复制代码

```JavaScript
function add(x:number,y:number):number{
  //add(x:number 此参数定义为数字,y:number 此参数定义为数字):number 此函数返回定义为数字
  return x + Y
}
let res = add (1,2) //因为定义函数返回 所以res为数字
```

可选参数 ? JavaScript | 复制代码

```JavaScript
function add(x:number,y:number,z?:number):number{
  if(typeOf z === 'number'){
    return x + Y + z
  }else{
    return x + Y
  }
}

```

可选参数一定是最后一个 否则TS会报错 TS不能确定你最后一个是什么

提一嘴 void 表示 函数没有返回值 可以返回undifind

void JavaScript | 复制代码

```JavaScript
let num = 0
function add(x:number,y:number):void{
  num = x + Y
}
add(1,2)
console.log(num)
```

枚举

enum 枚举属性 JavaScript | 复制代码

```
enum Direction{
  Up,
  Down,
  Left,
  Right,
}
console.log(Direction.Up)//输出是0 且后边属性递增
console.log(Direction[0])//输出是up TS会做反方映射
```

枚举特性属性默认会被赋值从0开始 且是双向赋值 enum特别像数组

具体编译完成的文件 实现双向赋值的过程 JavaScript | 复制代码

```
var Direction;
(function(Direction){
  Direction[Direction["Up"] = 0] = "Up";
  Direction[Direction["Down"] = 1] = "Down";
  Direction[Direction["Left"] = 2] = "Left";
  Direction[Direction["Right"] = 3] = "Right";
})(Direction || (Direction = {}))
//js里 Direction["Up"] = 0 赋值操作其实最后返回的是赋的值 返回 0
//所以Direction[Direction["Up"] = 0] = "Up"其实等于
//Direction.up = 0 返回0 所以 Direction.0 = "Up" 巧妙
console.log(Direction.Up) //输出 0
console.log(Direction[0]) //输出 Up
```

enum 枚举赋值 JavaScript | 复制代码

```
enum Direction{
  Up = 'Up',
  Down = 'Down',
  Left,
  Right,
}
//此处会报错 enum 一旦赋值所有都需要赋相同类型的值
```

使用const定义枚举 可以优化性能 JavaScript 复制代码

```javascript
const enum Direction{
  Up = 'Up',
  Down = 'Down',
  Left = 'Left',
  Right = 'Right',
}
const value = 'up'
if( value === Direction.up){
  console.log('is her')
}
```

const定义枚举 编译过后 JavaScript 复制代码

```javascript
var value ='up'
if ( value === 'up' ) {
  console.log('is her')
}
// Ts会编译你使用的变量 其他不会编译从而优化性能
```

泛型

小引 JavaScript 复制代码

```javascript
funtion echo(arg){
  return arg
}
const res = echo(123) //此处res是any 变量丧失了类型
//所以我们改为
funtion echo(arg:number):number{
  return arg
}
const res = echo(123) //此处才为正常 number
//但是函数一般参数可以是很多类型 上边例子只能局限在声明函数处定义函数类型
```

泛型 JavaScript 复制代码

```JavaScript
funtion echo<T>(arg:T):T{
  //泛型的声明 <> 此处T可以随便命名
  //泛型声明过后 Ts会从实参拿实参的类型 赋值给形参 和 函数返回
  return arg
}
let str:string ='Tly'
const res = echo(str) //此处res为str定义时的string类型
//也可不指定
funtion echo<T>(arg:T):T{
  return arg
}
const res = echo("Tly")
//此处res为string 定义过泛型后 类型推论会从实参处获得类型

// 且如果定义过泛型后
const res:boolean = echo("Tly")  //此处报错 现在res已经是函数返回的string类型
```

泛型多值返回 JavaScript 复制代码

```JavaScript
function swap<T,U>(tup:[T,U]): [T,U]{
  return [tup[0],tup[1]]
}
const res = swap(['Tly',123]) //此处 返回为正常
res[0].length
res[1].toFixed(2)
```

泛型类似一个变量 我们传入什么类型 它会返回什么类型

解决了函数参数类型定义 过于死板的问题

泛型约束

小引 JavaScript 复制代码

```JavaScript
funtion echo<T>(arg:T):T{
  console.log(arg.legth) //此处报错因为我们虽声明了的泛型 但Ts不能从函数内部得知它的类型
  return arg
}
```

泛型约束 JavaScript | 复制代码

```javascript
funtion echo<T>(arg:T[]):T[]{
  //可以在泛型后加入类型
  console.log(arg.legth) //此处正常使用
  return arg
}
const res = echo([1,2,3])//此刻为正常
//但是我们发现现在实参只能数组
//其他类型会报错 我们又陷入到了函数类型定义死板的问题
```

约束泛型关键字 extends JavaScript | 复制代码

```javascript
interface lengthFn {
  length:number
}
funtion echo<T extends lengthFn>(arg:T):T{
  //此处拿到lengthFn里的length 告诉Ts实参必须要有length属性
  console.log(arg.legth) //此处不会报错
  return arg
}
const str = echo('123')
const str = echo({length:10})
// 以上都不报错
const str = echo(123) //此处报错number无.length属性
```

extends通过interface属性存在与否去判断这个类型是不是你需要的类型

interface的别名'鸭子类型'

只要会叫就是鸭子 不管叫的究竟是什么

只要有length 他就是你需要的 不管这个length是从哪里来

引子 JavaScript 复制代码

```
class cls {
  private data =[]
  // public 定义类的变量默认就是公共的，继承的子类可以通过this来访问
  // private 定义类的私有属性，只能在内部访问
  // protected 在类的内部和子类中可以访问,在外面就访问不到了
  push(item) {
    return this.data.push(item)
  }
  push(item) {
    return this.data.shift(item)
  }
}
const query = new cls()
query.push(1)
query.push('str')
console.log(query.pop().toFixed())
console.log(query.pop().toFixed())//此时会报错因为现在query是any TS不会报错编译报错
```

类种使用泛型 JavaScript 复制代码

```
class cls<T> {
  private data =[]
  push(item:T) {
    return this.data.push(item)
  }
  push(item:T) {
    return this.data.shift(item)
  }
}
const query = new cls<number>()
query.push(1)
query.push('str') //此刻正常添加string会报错
```

interface和泛型 JavaScript | 复制代码

```javascript
interface kepclass<T,U>{
  key: T ,
  value: U
}
let res1:kepclass<number,string> = {
  key:1,
  value:'str'
}
let res2:kepclass<string,number> = {
  key:'str',
  value:1
}
// 两个都不会报错 观察在使用kepclass传入顺序
```

数组和泛型 JavaScript | 复制代码

```javascript
let arr:number[] = [1,2,3]
let arr:Array<number>= [1,2,3]
```

泛型'我'理解为参数

特别类似与函数

传入是什么 返回是什么

主要是为了解决TS代码方法复用　类型不灵活的问题

解决了声明好的方法　在其他地方调用此方法时

此方法类型已经被声明

从而报错的尴尬情况

类型别名 TS声明

别名 type

JavaScript | 复制代码

```
let sum :(x:number,y:number):number{
  return x+y
}
const res = sum(1,2)
type sumType = (x:number,y:number)=>number
const res1 = sumType(1,2)
type test = number; //基本类型
let num: test = 10;
type userOjb = {name:string} // 对象
type getName = ()=>string  // 函数
type data = [number,string] // 元组
type numOrFun = Second | getName  // 联合类型
```

字面量 限制变量

字面量：

JavaScript | 复制代码

```
let string :'tly' = 'tly' //不等于:'tly'定义的'tly'就会报错
let number :1 = 1
//结合别名使用
type sumType = 'left' | 'right'
let toher:sumType ='left'
```

交叉类型 &

交叉类型 &

JavaScript | 复制代码

```
interface I {
  name:string
}
type anage = I & {age:10}
let Im :anage ={
  name:"tly",
  age:20
}
```

内置类型

内置对象 JavaScript 复制代码

```
// 内置对象
let date = new Date()
date.getTime() //TS会识别这些时间对象 正则对象等
//es对象等
Math.pow(2,2) //es等等对象也会识别
// dom or bom
let body = document.body //dom

```

Partial可选类型

转化可选类型 Partial<...> JavaScript 复制代码

```
interface I {
  name:string
  age:number
}
let Me :I ={
  name:"Tly",
  age:20
}
type Im = Partial<I>
let isMe :Im ={} //此时Im属性都为可选
```

Omit可选忽略

可选忽略 Omit<... , ...> JavaScript 复制代码

```
interface I {
  name:string
  age:number
}
let Me :I ={
  name:"Tly",
  age:20
}
type Im = Omit<I,'name'>
let isMe :Im ={age:20} //此时name为可选
```

TS配置文件等

TS 使用 tsconfig.json 作为其配文件， 它主要包含两块内容：

指定待编译的文件

定义编译选项

另外， 一般来说， tsconfig.json 文件所处的路径就是当前 TS 项目的根路径

tsconfig.json 的配置项众多并且复杂 。 所有的选项可以参考言方文档

https://www.typescriptlang.org/zh/tsconfig