# 接口

## 1.普通接口

接口是一系列抽象方法的声明，是一些方法特征的集合，这些方法都应该是抽象的，需要由具体的类去实现，然后第三方就可以通过这组抽象方法调用，让具体的类执行具体的方法。

TypeScript 接口定义如下：

interface interface\_name {

}

### 实例

以下实例中，我们定义了一个接口 IPerson，接着定义了一个变量 customer，它的类型是 IPerson。

customer 实现了接口 IPerson 的属性和方法。

|  |
| --- |
| interface IPerson {  firstName: string,  lastName: string,  sayHi: () => string  }  var customer: IPerson = {  firstName: "Tom", lastName: "Hanks", sayHi: (): string => {  return "Hi there"  }  }  console.log("Customer 对象 ")  console.log(customer.firstName)  console.log(customer.lastName)  console.log(customer.sayHi())  var employee: IPerson = {  firstName: "Jim", lastName: "Blakes", sayHi: (): string => {  return "Hello!!!"  }  }  console.log("Employee 对象 ")  console.log(employee.firstName)  console.log(employee.lastName) |

### 另外一个实例

|  |
| --- |
| interface Human{  name:string,  age:number,  gender:string,  eat:()=>void,  run:()=>void,  sayHi:()=>string }  let jade:Human = {  name:'Jade',  age:18,  gender:'female',  eat:()=>{console.log('eating rice')},  run:function(){console.log(`${this.name} is running`)},  sayHi:function (){  return `hi i am ${this.name}`  } }  console.log(jade.sayHi()); jade.run() |

### 效果

|  |
| --- |
|  |

## 2.有可选属性的接口

### 在接口里面也可以用可选属性，因为有些属性是可有可无的

|  |
| --- |
| interface Student{  name:string,  gender:string,  age:number,  email?:string }  let jane :Student ={  name:'Jane',  gender:'female',  age:30 } console.log(jane)  let janna :Student ={  name:'Janna',  gender:'female',  age:28,  email:'Janna123@gmail.com' }  console.log(janna) |

## 3.带有只读属性的接口

### 指的属性需要使用关键字：readonly来声明，如：我们给上面的接口添加一个privacy属性

|  |
| --- |
| interface Student{  name:string,  gender:string,  age:number,  readonly privacy:string,  email?:string }  let janna :Student ={  name:'Janna',  gender:'female',  age:28,  email:'Janna123@gmail.com',  privacy:'nothing' }  console.log(janna) //janna.privacy = '' //报错，因为你尝试修改一个只读属性 |

## 4.在一个接口中使用另外一个接口

### 可以将一个接口类型的变量作为另外一个接口的属性，

### 实例

|  |
| --- |
| //从一个接口中使用另外一个接口 interface Info{  name:string,  age:number,  gender:string,  address:string,  marriage:boolean }  interface Employee{  id:number,  info:Info,  email?:string }  let xm:Employee = {  id:1001,  info:{  name:'xiaoming',  age:20,  gender:'male',  address:'3 pawsey road,kgn 5',  marriage:false  } ,  email:'xiaoming123@hotmail.com' }  console.log(xm) |

#### 运行结果

|  |
| --- |
|  |

### 实例2.计算两个点之间的距离

#### 计算原理：取两个点的x坐标的差的平方加上两个y坐标的差是平方的和开方

|  |
| --- |
| interface Point{  x:number,  y:number }  interface GetDistance{  (pt1:Point,pt2:Point):number }  let getDist:GetDistance = (p1,p2)=>{  return Math.sqrt(Math.pow((p1.x - p2.x), 2) + Math.pow((p1.y - p2.y), 2)) }  let pt1 = {x:3,y:3} let pt2 = {x:4,y:4} console.log(getDist(pt1,pt2)) |

#### 效果

|  |
| --- |
|  |

# 接口和函数

可以使用一个接口来约束函数，格式如下

|  |
| --- |
| interface 接口名称{  (参数1:类型1,参数2:类型2) : 返回值类型} } |

### 实例

|  |
| --- |
| //使用接口来约束函数 interface MyFun{  (a:string,b:number):void }  let fun:MyFun=(a,b)=>{  console.log("function is running")  console.log(a)  console.log(b)  console.log("finished") } fun('hello',100) |

#### 效果：

|  |
| --- |
|  |

# 联合类型和接口

## 以下实例演示了如何在接口中使用联合类型：

|  |
| --- |
| **interface RunOptions {**  **program:string;**  **commandline:string[]|string|(()=>string);**  **}**    **// commandline 是字符串**  **var options:RunOptions = {program:"test1",commandline:"Hello"};**  **console.log(options.commandline)**    **// commandline 是字符串数组**  **options = {program:"test1",commandline:["Hello","World"]};**  **console.log(options.commandline[0]);**  **console.log(options.commandline[1]);**    **// commandline 是一个函数表达式**  **options = {program:"test1",commandline:()=>{return "\*\*Hello World\*\*";}};**    **var fn:any = options.commandline;**  **console.log(fn());** |

# 接口和数组

接口中我们可以将数组的索引值和元素设置为不同类型，索引值可以是数字或字符串。

设置元素为字符串类型：

|  |
| --- |
| 实例  interface namelist {  [index:number]:string  }    // 类型一致，正确  var list2:namelist = ["Google","Runoob","Taobao"]  // 错误元素 1 不是 string 类型  // var list2:namelist = ["Runoob",1,"Taobao"] |

# 接口继承

接口继承就是说接口可以通过其他接口来扩展自己。

Typescript 允许接口继承多个接口。

继承使用关键字 **extends**。

单接口继承语法格式：

Child\_interface\_name extends super\_interface\_name

多接口继承语法格式：

Child\_interface\_name extends super\_interface1\_name, super\_interface2\_name,…,super\_interfaceN\_name

继承的各个接口使用逗号 **,** 分隔。

## 单继承实例

|  |
| --- |
| interface Person {  age:number  }    interface Musician extends Person {  instrument:string  }    var drummer = <Musician>{};  drummer.age = 27  drummer.instrument = "Drums"  console.log("年龄: "+drummer.age)  console.log("喜欢的乐器: "+drummer.instrument) |

## 单继承实例2

|  |
| --- |
| interface Person{  name:string,  age:number }  interface Musician extends Person{  instrument:string,  toString:()=>string }  let mc:Musician = <Musician>{} mc.name='Jack' mc.age = 40 mc.instrument = 'Drum' mc.toString=function (){  return `Musician[name:${this.name},age:${this.age},playing:${this.instrument}]` } console.log(mc.toString()) |

### 运行结果

|  |
| --- |
|  |

## 多继承实例

|  |
| --- |
| interface IParent1 {  v1:number  }    interface IParent2 {  v2:number  }    interface Child extends IParent1, IParent2 { }  var Iobj:Child = { v1:12, v2:23}  console.log("value 1: "+Iobj.v1+" value 2: "+Iobj.v2) |

# 断言

[TypeScript](https://so.csdn.net/so/search?q=TypeScript&spm=1001.2101.3001.7020)类型断言是一个编译时语法，用于告诉编译器用户比编译器更加确定变量的类型，进而解除编译错误，类型断言有点类似于其他语言的类型转换，但它没有运行时的影响，只是在编译阶段起作用。所以，即使通过类型断言解除了编译错误，也不会影响运行错误。

## 类型断言

基本用法

1. 使用as

|  |
| --- |
|  |

值 as 类型

1. 使用尖括号

<类型>值

由于<类型>这种用法在TypeScript中除了表示类型[断言](https://so.csdn.net/so/search?q=%E6%96%AD%E8%A8%80&spm=1001.2101.3001.7020)之外，也可能是表示一个泛型。所以推荐使用第一种方式，避免混淆。

**实例**

|  |
| --- |
| let value: any = "this is a string";  let length: number = (value as string).length; |

### 将一个联合类型断言为其中一个类型

|  |
| --- |
| **interface Cat{  name:string,  catch\_mouse():void }  interface Fish{  name:string,  swim():void }  function isFish(animal:Cat|Fish) {  return typeof (animal as Fish).swim === 'function';  }  let nimo:Fish = {  name:'Nimo',  swim() {  console.log('I am swimming very fast')  } }  console.log("Is nimo fish? "+isFish(nimo))** |

### 运行结果

|  |
| --- |
|  |

## 非空断言

#### 当我们确认某个值不可能为 undefined 或 null 时，可以在变量后面加上一个 “!”。

|  |
| --- |
| function fun(value: string | undefined | null) {  const str: string = value; // 错误 value 可能为 undefined 和 null  const str1: string = value!; //ok  const length: number = value.length; // 错误 value 可能为 undefined 和 null  const length1: number = value!.length; //ok  } |

## 确定赋值断言

#### 确定赋值断言即允许在实例属性和变量声明后面放置一个 ! 号，从而告诉编译器该属性一定会被明确的赋值。

|  |
| --- |
| let x: number;  initialize();  console.log(2 \* x); // 错误，x在赋值之前被调用  function initialize() {  x = 10;  }  加上断言后，编译错误消失。  let x!: number;  initialize();  console.log(2 \* x); // ok  function initialize() {  x = 10;  } |

**总结**

* 联合类型可以被断言为其中一个类型
* 父类可以被断言为子类
* 任何类型都可以被断言为 any
* any 可以被断言为任何类型
* 当我们确认某个值不可能为 undefined 或 null 时，可以在变量后面加上一个 “!”。

# 泛型

## 什么是泛型？

泛型，从字面上理解，泛型就是一般的，广泛的的意思。

TypeScript中泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体类型，而是在使用的时候再指定类型的一种特性。

泛型中的 T 就像一个占位符、或者说一个变量，在使用的时候可以把定义的类型**像参数一样**传入，它可以原封不动地输出。  
**泛型在成员之间提供有意义的约束**，这些成员可以是：函数参数、函数返回值、类的实例成员、类的方法等。

## 使用泛型的好处

1. 函数和类可以轻松支持多种类型，增强程序的拓展性
2. 不必写冗长的业务代码，增强代码的可拓展性
3. 灵活控制类型之间的约束

## 泛型的基本使用

### 泛型函数

#### 1. 处理函数参数

泛型的语法是尖括号 <> 里写类型参数，一般可以用T来表示。

定义一个 print 函数，这个函数的功能是把传入的参数打印出来，再返回这个参数，传入参数的类型与函数返回类型一致。

|  |
| --- |
| function print<T>(arg:T):T {  console.log(arg);  return arg;  } |

这样，我们就**做到了输入和输出的类型统一，且可以输入输出任何类型**。

们在使用的时候可以有两种方式指定类型:

* 定义要使用的类型
* TS 类型推断，自动推导出类型

|  |
| --- |
| print<string>('hello') // 定义 T 为 string  print('hello') // TS 类型推断，自动推导类型为 string |

##### 实例

|  |
| --- |
| **function reverse<T>(items: T[]): T[] {**  **const toreturn = [];**  **for (let i = items.length - 1; i >= 0; i--) {**  **toreturn.push(items[i]);**  **}**  **return toreturn;**  **}**  **const sample = [1, 2, 3];**  **let reversed = reverse(sample);**  **reversed[0] = '1'; // Error**  **reversed = ['1', '2']; // Error**  **reversed[0] = 1; // ok**  **reversed = [1, 2]; // ok** |

##### 可以在一个普通类里面使用泛型成员函数

|  |
| --- |
| **class Utility {**  **reverse<T>(items: T[]): T[] {**  **const toreturn = [];**  **for (let i = items.length; i >= 0; i--) {**  **toreturn.push(items[i]);**  **}**  **return toreturn;**  **}**  **}** |

##### 实例：一个用于加载 json 返回值函数，它返回你任何传入类型的 Promise：

|  |
| --- |
| const getJSON = <T>(config: { url: string; headers?: { [key: string]: string } }): Promise<T> => {  const fetchConfig = {  method: 'GET',  Accept: 'application/json',  'Content-Type': 'application/json',  ...(config.headers || {})  };  return fetch(config.url, fetchConfig).then<T>(response => response.json());  }; |

请注意，你仍然需要明显的注解任何你需要的类型，但是 getJSON<T> 的签名 config => Promise<T> 能够减少你一些关键的步骤（你不需要注解 loadUsers 的返回类型，因为它能够被推出来）：

|  |
| --- |
| **type LoadUserResponse = {**  **user: {**  **name: string;**  **email: string;**  **}[];**  **};**  **function loaderUser() {**  **return getJSON<LoadUserResponse>({ url: 'https://example.com/users' });**  **}** |

与此类似：使用 Promise<T> 作为一个函数的返回值比一些如：Promise<any> 的备选方案要好很多

##### 配合 axios 使用

通常情况下，我们会把后端返回数据格式单独放入一个 interface 里：

|  |
| --- |
| **// 请求接口数据**  **export interface ResponseData<T = any> {**  **/\*\***  **\* 状态码**  **\* @type { number }**  **\*/**  **code: number;**  **/\*\***  **\* 数据**  **\* @type { T }**  **\*/**  **result: T;**  **/\*\***  **\* 消息**  **\* @type { string }**  **\*/**  **message: string;**  **}** |

当我们把 API 单独抽离成单个模块时：

|  |
| --- |
| **// 在 axios.ts 文件中对 axios 进行了处理，例如添加通用配置、拦截器等**  **import Ax from './axios';**  **import { ResponseData } from './interface.ts';**  **export function getUser<T>() {**  **return Ax.get<ResponseData<T>>('/somepath')**  **.then(res => res.data)**  **.catch(err => console.error(err));**  **}** |

接着我们写入返回的数据类型 User，这可以让 TypeScript 顺利推断出我们想要的类型：

|  |
| --- |
| **interface User {**  **name: string;**  **age: number;**  **}**  **async function test() {**  **// user 被推断出为**  **// {**  **// code: number,**  **// result: { name: string, age: number },**  **// message: string**  **// }**  **const user = await getUser<User>();**  **}** |

#### 2. 默认参数

如果要给泛型加默认参数，可以这么写：

|  |
| --- |
| **interface Iprint<T = number> {**  **(arg: T): T**  **}**  **function print<T>(arg:T) {**  **console.log(arg)**  **return arg**  **}**  **const myPrint: Iprint = print** |

#### 3. 处理多个函数参数

现在有这么一个函数，传入一个只有两项的元组，交换元组的第 0 项和第 1 项，返回这个元组。

[数组合并](https://so.csdn.net/so/search?q=%E6%95%B0%E7%BB%84%E5%90%88%E5%B9%B6&spm=1001.2101.3001.7020)了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

|  |
| --- |
| **let tom: [string, number] = [‘Tom’, 25]; // 定义一对值分别为 string 和 number 的元组**  **function swap<T, U>(tuple: [T, U]): [U, T]{**  **return [tuple[1], tuple[0]]**  **}** |

#### 4. 函数副作用操作

泛型不仅可以很方便地约束函数的参数类型，还可以用在函数执行副作用操作的时候。

比如我们有一个通用的[异步请求](https://so.csdn.net/so/search?q=%E5%BC%82%E6%AD%A5%E8%AF%B7%E6%B1%82&spm=1001.2101.3001.7020)方法，想根据不同的 url 请求返回不同类型的数据。

|  |
| --- |
| function request(url:string) {  return fetch(url).then(res => res.json())  }  // 调一个获取用户信息的接口：  request('user/info').then(res =>{  console.log(res)  }) |

这时候的返回结果 res 就是一个 any 类型，非常讨厌。

我们希望调用 API 都清晰的知道返回类型是什么数据结构，就可以这么做

|  |
| --- |
| **interface UserInfo {**  **name: string**  **age: number**  **}**  **function request<T>(url:string): Promise<T> {**  **return fetch(url).then(res => res.json())**  **}**  **request<UserInfo>('user/info').then(res =>{**  **console.log(res)**  **})** |

### 泛型类型别名

|  |
| --- |
|  |

### 泛型接口

|  |
| --- |
|  |

#### 泛型接口的基本使用

interface IInfo<T1, T2> {

name: T1

age: T2

}

const info: IInfo<string, number> = {

name: "zs",

age: 25

}

#### 泛型接口是没有类型推导的, 但是可以有泛型默认值。

// 泛型接口定义默认类型

interface IPerson<T1 = string, T2 = number> {

name: T1

age: T2

}

const p: IPerson = {

name: "chenyq",

age: 123

}

### 泛型类

#### 实例1

|  |
| --- |
| class Point<T> {  x: T  y: T  z: T  constructor(x: T, y: T, z: T) {  this.x = x  this.y = y  this.z = y  }  }  // 泛型类自动推导类型  const point1 = new Point("1.33.2", "2.22.3", "4.22.1")  // 泛型类明确泛型类型  const point2 = new Point<string>("1.33.2", "2.22.3", "4.22.1")  const point3: Point<string> = new Point("1.33.2", "2.22.3", "4.22.1") |

#### 实例2

|  |
| --- |
| **// 创建一个泛型类**  **class Queue<T> {**  **private data: T[] = [];**  **push = (item: T) => this.data.push(item);**  **pop = (): T | undefined => this.data.shift();**  **}**  **// 简单的使用**  **const queue = new Queue<number>();**  **queue.push(0);**  **queue.push('1'); // Error：不能推入一个 `string`，只有 number 类型被允许** |

### 数组泛型的写法：

// 定义字符串数组

const arr1: string[] = ["a", "b", "c"]

const arr2: Array<string> = ["a", "b", "c"]

### 泛型约束

有时候希望传入的类型具有某些共性，但是这些共性可能不是在同一种类型中。

比如string和array都是有length的，或者某些对象也是会有length属性的。

那么只要是拥有length的属性都可以作为我们的参数类型。

|  |
| --- |
| interface ILength {  length: number  }  // 泛型继承接口, 这样传入的泛型类型就必须和ILength接口一样具有length属性才可以  function getLength<T extends ILength>(arg: T) {  console.log(arg.length)  }  // getLength(123) // 123没有length无法传入  getLength("abcdefg") // 7  getLength([10, 20, 30, 40]) // 4  getLength({ length: 20 }) // 10 |