自动编译

- 文件夹右键打开终端，输入tsc --init（报错的话，先执行cmd）
- 修改tsconfig.json配置
  - "outDir": "./js",
  - "strict": false,
- 启动监视任务
  - 选中tsconfig.json ->终端 -> 运行任务 -> 监视tsconfig.json

webpack打包

- 新建public-index.html
- 新建src-main.ts
- 新建build-webpack.config.js复制配置
- 总文件夹中右键打开终端
  - npm init -y：生成package.json
  - tsc --init：生成tsconfig.json
  - 下载依赖 注意版本
    - npm install -D typescript
    - npm install -D webpack webpack-cli
    - npm install -D webpack-dev-server
    - npm install -D html-webpack-plugin clean-webpack-plugin
    - npm install -D ts-loader
    - npm install -D cross-env
- 配置打包命令
  - "dev": "cross-env NODE_ENV=development webpack-dev-server --config build/webpack.config.js",
- "build": "cross-env NODE_ENV=production webpack --config build/webpack.config.js"

基础类型

- 基本语法
  - let 变量名: 数据类型 = 值
- 布尔类型

```
let isDone: boolean = false;
isDone = true;
```

- 数字类型

```
let a1: number = 10 // 十进制
let a2: number = 0b1010  // 二进制
let a3: number = 0o12 // 八进制
let a4: number = 0xa // 十六进制
```

- 字符串类型

```
let str1: string = '床前明月光'
// 字符串和数字之间能够一起拼接
```

- undefined和null

```
let und: undefined = undefined
let nll: null = null
```

- 数组类型

```
// 数组定义方式1
// 语法: let 变量名: 数据类型[] = [值1,值2,值3]
let arr1: number[] = [10, 20, 30, 40, 50]

// 数组定义方式2:泛型的写法
// 语法: let 变量名: Array<数据类型> = [值1,值2,值3]
let arr2: Array<number> = [100, 200, 300]

// 数组定义后,里面的数据的类型必须和定义数组的时候的类型一致
```

- 元组类型

```
// 在定义数组的时候,类型和数据的个数一开始就已经限定了
let arr3: [string, number, boolean] = ['小甜甜', 100.12345, true]
// 元组类型在使用的时候，数据的类型的位置和数据的个数，应该和在定义元组的时候的数据类型及位置应该一致
// 当访问一个已知索引的元素，会得到正确的类型
```

- 枚举类型

```
// 枚举里面的每个数据值都可以叫元素,每个元素都有自己的编号,编号是从0开始的,依次的递增加1
// 也可以手动的指定成员的数值，也可以全部都采用手动赋值
enum Color {
  red = 1,  // 从1开始编号，依次递增
  green,
  blue
}
// 定义一个Color的枚举类型的变量来接收枚举的值
let color: Color = Color.red
console.log(color)
console.log(Color.red, Color.green, Color.blue)
// 由枚举的值得到它的名字
console.log(Color[3])
```

- any类型

```
// 当一个数组中要存储多个数据,个数不确定,类型不确定,此时也可以使用any类型来定义数组
let str: any =100
str='年少不知富婆好,错把少女当成宝'
// 有一个数组，包含不同的类型的数据
let arr: any[] = [100, '年少不知软饭香,错把青春倒插秧', true]
```

- void类型

```
// void 类型,在函数声明的时候,小括号的后面使用:void,代表的是该函数没有任何的返回值
function showMsg(): void {
  console.log('只要富婆把握住,连夜搬进大别墅')
  // return
  // return undefined
  return null
}
console.log(showMsg())
// 定义void类型的变量,可以接收一个undefined的值,但是意义不是很大
let vd: void = undefined
```

- object类型

```
// 非原始类型，除 number, string, boolean之外的类型。
// 定义一个函数,参数是object类型,返回值也是object类型
function getObj(obj: object): object {
  console.log(obj)
  return {
    name: '卡卡西',
    age: 27
  }
}
```

- 联合类型

```
// 表示取值可以为多种类型中的一种
// 需求1: 定义一个函数得到一个数字或字符串值的字符串形式值
function getString(str:number|string):string{
    return str.toString()
}

// 需求2: 定义一个函数得到一个数字或字符串值的长度
function getLength(x: number | string) {
  if (x.length) {
    return x.length
  } else {
    return x.toString().length
  }
}
```

- 类型断言

```
// 类型断言:告诉编译器,我知道我自己是什么类型,也知道自己在干什么
// 好比其它语言里的类型转换，但是不进行特殊的数据检查和解构
// 类型断言的语法方式1: <类型>变量名
// 类型断言的语法方式2: 值 as 类型
function getLength(x: number | string) {
  if ((<string>x).length) {
    return (x as string).length
  } else {
    return x.toString().length
  }
}
```

- 类型推断

```
/* 定义变量时赋值了，推断为对应的类型 */
let b9 = 123 // number

/* 定义变量时没有赋值，推断为any类型 */
let b10  // any类型
b10 = 123
b10 = 'abc'
```

接口

- 接口是对象的状态(属性)和行为(方法)的抽象(描述)
- 是一种类型，是一种规范，是一种规则，是一个能力，是一种约束

```
// 需求: 创建人的对象，需要对人的属性进行一定的约束

// id是number类型，必须有，只读的
// name是string类型，必须有
// age是number类型，必须有
// sex是string类型，可以没有

// 定义一个接口,该接口作为person对象的类型使用,限定或者是约束该对象中的属性数据
interface IPerson {
  // 只读属性--readonly id是只读的,是number类型,const修饰属性,想要设置该属性是只读的,是不能使用的
  readonly id:  number
  name: string
  age: number
  // 可选属性--? 可有可无的
  sex?: string
}

// 定义一个对象,该对象的类型就是我定义的接口IPerson
const person: IPerson = {
  id: 1,
  name: '小甜甜',
  age: 18,
  // sex这个属性没有也是可以的
```

```
  // sex: '女'
}
```

- 函数类型：通过接口的方式作为函数的类型来使用

```
// 为了使用接口表示函数类型，我们需要给接口定义一个调用签名。
// 它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

// 定义一个接口,用来作为某个函数的类型使用，可以描述函数类型(参数的类型与返回的类型)
interface ISearchFunc {
  // 定义一个调用签名
  (source: string, subString: string): boolean
}
// 定义一个函数,该类型就是上面定义的接口
const searchString: ISearchFunc = function (source: string, subString: string):
boolean {
  // 在source字符串中查找subString这个字符串
  return source.search(subString) > -1
}
// 调用函数
console.log(searchString('哈哈,我又变帅了', '帅'))
```

- 类类型

```
// 类 类型：类的类型,类的类型可以通过接口来实现
// 定义一个接口
interface IFly {
  // 该方法没有任何的实现(方法中什么都没有)
  fly()
}
// 定义一个类,这个类的类型就是上面定义的接口(实际上也可以理解为,IFly接口约束了当前的这个Person类)
class Person implements IFly {
  // 实现接口中的方法
  fly() {
    console.log('我会飞了,我是超人')
  }
}
// 实例化对象
const person = new Person()
person.fly()

// 再定义一个接口
interface ISwim {
  swim()
}
// 【一个类可以实现多个接口】
// 定义一个类,这个类的类型是IFly和ISwim(这个类可以实现多个接口,一个类同时也可以被多个接口进行约束)
class Person2 implements IFly,ISwim {
  fly() {
    console.log('我飞了2')
  }
  swim() {
```

```
    console.log('我会游泳啦2')
  }
}
// 实例化对象
const person2 = new Person2()
person2.fly()
person2.swim()

// 总结: 类可以通过接口的方式,来定义当前这个类的类型
// 类可以实现一个接口,类也可以实现多个接口,要注意,接口中的内容都要真正的实现

// 【接口继承接口】定义了一个接口,继承其他的多个接口
interface IMyFlyAndSwim extends IFly, ISwim { }

// 定义一个类,直接实现IMyFlyAndSwim这个接口
class Person3 implements IMyFlyAndSwim {
  fly() {
    console.log('我飞了3')
  }
  swim() {
    console.log('我会游泳啦3')
  }
}
const person3 = new Person3()
person3.fly()
person3.swim()

// 总结:接口和接口之间叫继承(使用的是extends关键字),类和接口之间叫实现(使用的是implements)
```

类

- 类：可以理解为模版，通过模版可以实例化对象，面向对象的编程思想

```
class Person {
  // 定义属性
  name: string
  age: number
  gender: string
  // 定义构造函数:为了将来实例化对象的时候,可以直接对属性的值进行初始化
  constructor(name: string='小甜甜', age: number=16, gender: string='女') {
    // 更新对象中的属性数据
    this.name = name
    this.age = age
    this.gender = gender
  }
  // 定义实例方法
  sayHi(str: string) {
    console.log(`大家好,我是${this.name},今年已经${this.age}岁了,是个${this.gender}孩子,`,
str)
  }
}
// ts中使用类,实例化对象,可以直接进行初始化操作
```

```
const person = new Person('佐助',18,'男')
person.sayHi('你叫什么名字啊')
```

- 继承：类与类之间的关系

```
// A类继承了B这个类,那么此时A类叫子类,B类叫基类
// 子类---->派生类
// 基类---->超类(父类)
// 一旦发生了继承的关系,就出现了父子类的关系(叫法)
// 定义一个类,作为基类(超类/父类)
class Person {
  // 定义属性
  name: string // 名字
  age: number // 年龄
  gender: string // 性别
  // 定义构造函数
  constructor(name: string='小明', age: number=18, gender: string='男') {
    // 更新属性数据
    this.name = name
    this.age = age
    this.gender = gender
  }
  // 定义实例方法
  sayHi(str: string) {
    console.log(`我是:${this.name},${str}`)
  }
}
// 定义一个类,继承自Person
class Student extends Person {
  constructor(name: string, age: number, gender: string) {
    // 调用的是父类中的构造函数,使用的是super
    super(name, age, gender)
  }
  // 可以调用父类中的方法
  sayHi() {
    console.log('我是学生类中的sayHi方法')
    // 调用父类中的sayHi方法
    super.sayHi('哈哈')
  }
}

// 实例化Person
const person = new Person('大明明',89,'男')
person.sayHi('嘎嘎')
// 实例化Student
const stu = new Student('小甜甜',16,'女')
stu.sayHi()

// 总结:类和类之间如果要有继承关系,需要使用extends关键字
// 子类中可以调用父类中的构造函数,使用的是super关键字(包括调用父类中的实例方法,也可以使用super)
// 子类中可以重写父类的方法
```

- 多态：父类型的引用指向了子类型的对象,不同类型的对象针对相同的方法,产生了不同的行为

```
// 定义一个父类
class Animal {
  // 定义一个属性
  name: string
  // 定义一个构造函数
  constructor(name: string) {
    // 更新属性值
    this.name = name
  }
  // 实例方法
  run(distance: number = 0) {
    console.log(`跑了${distance} 米这么远的距离`, this.name)
  }
}
// 定义一个子类
class Dog extends Animal {
  // 构造函数
  constructor(name: string) {
    // 调用父类的构造函数,实现子类中属性的初始化操作
    super(name)
  }
  // 实例方法,重写父类中的实例方法
  run(distance: number = 5) {
    console.log(`跑了${distance} 米这么远的距离`, this.name)
  }
}
// 定义一个子类
class Pig extends Animal {
  // 构造函数
  constructor(name: string) {
    // 调用父类的构造函数,实现子类中属性的初始化操作
    super(name)
  }
  // 实例方法,重写父类中的实例方法
  run(distance: number = 10) {
    console.log(`跑了${distance} 米这么远的距离`, this.name)
  }
}
// 实例化父类对象
const ani: Animal = new Animal('动物')
ani.run()
// 实例化子类对象
const dog: Dog = new Dog('大黄')
dog.run()
// 实例化子类对象
const pig: Pig = new Pig('八戒')
pig.run()

// 父类和子类的关系:父子关系,此时,父类类型创建子类的对象
const dog1: Animal = new Dog('小黄')
dog1.run()
```

```
const pig1: Animal = new Pig('小猪')
pig1.run()

// 该函数需要的参数是Animal类型的
function showRun(ani: Animal) {
  ani.run()
}
showRun(dog1)
showRun(pig1)
```

- 修饰符：类中的成员的修饰符，主要是描述类中的成员(属性,构造函数,方法)的可访问性
  - 类中的成员都有自己的默认的访问修饰符,public
  - public修饰符---公共的,类中成员默认的修饰符,代表的是公共的,任何位置都可以访问类中的成员
  - private修饰符---私有的,类中的成员如果使用private来修饰,那么外部是无法访问这个成员数据的,当然,子类中也是无法访问该成员数据的
  - protected修饰符----受保护的,类中的成员如果使用protected来修饰,那么外部是无法访问这个成员数据的,当然,子类中是可以访问该成员数据的

```
// 定义一个类
class Person {
  // 属性 public 修饰了属性成员
  // public name: string
  // 属性 private 修饰了属性成员
  // private name: string
  // 属性protected 修饰了属性成员
  protected name:string
  // 构造函数
  public constructor(name: string) {
    // 更新属性
    this.name = name
  }
  // 方法
  public eat() {
    console.log('嗯,这个骨头真好吃', this.name)
  }
}
// 定义一个子类
class Student extends Person {
  // 构造函数
  constructor(name: string) {
    super(name)
  }
  play() {
    console.log('我就喜欢玩布娃娃',this.name)
  }
}
// 实例化对象
const per = new Person('大蛇丸')
// 类的外部可以访问类中的属性成员
// console.log(per.name)
per.eat()
const stu = new Student('红豆')
```

```
stu.play()
// console.log(stu.name)
```

- readonly修饰符：一个关键字,对类中的属性成员进行修饰,修饰后,该属性成员,不能在外部被随意的修改了

```
// 构造函数中,可以对只读的属性成员的数据进行修改
// 如果构造函数中没有任何的参数,类中的属性成员此时已经使用readonly进行修饰了,那么外部也是不能对这个属性值进行更改的
// 构造函数中的参数可以使用readonly进行修饰,一旦修饰了,那么该类中就有了这个只读的成员属性了,外部可以访问,但是不能修改
// 构造函数中的参数可以使用public及privte和protected进行修饰,无论是哪个进行修饰,该类中都会自动的添加这么一个属性成员

// 【readonly修饰类中的成员属性操作】
// 定义一个类型
class Person {
  // 属性
  // readonly name: string='大甜甜' // 初始值
  readonly name: string
  // 构造函数
  constructor(name: string = '大甜甜') {
    this.name = name
  }
  sayHi() {
    console.log('考尼奇瓦', this.name)
    // 类中的普通方法中,也是不能修改readonly修饰的成员属性值
    // this.name = '大甜甜'
  }
}
// 实例化对象
const person: Person = new Person('小甜甜')

console.log(person)
console.log(person.name)
// 此时无法修改,因为name属性是只读的
person.name = '大甜甜'
console.log(person.name)

// 【readonly修饰类中的构造函数中的参数(参数属性)】
// 定义一个类型
class Person {
  // 构造函数
  // 构造函数中的name参数,一旦使用readonly进行修饰后,那么该name参数可以叫参数属性
  // 构造函数中的name参数,一旦使用readonly进行修饰后,那么Person中就有了一个name的属性成员
  // 构造函数中的name参数,一旦使用readonly进行修饰后,外部也是无法修改类中的name属性成员值的
  constructor(readonly name: string = '大甜甜') {
    this.name = name
  }
  // 构造函数中的name参数,一旦使用public进行修饰后,那么Person类中就有了一个公共的name属性成员了
  constructor(public name: string = '大甜甜') {
    this.name = name
  }
  // 构造函数中的name参数,一旦使用private进行修饰后,那么Person类中就有了一个私有的name属性成员了
```

```
  constructor(private name: string = '大甜甜') {
    this.name = name
  }
  // 构造函数中的name参数,一旦使用protected进行修饰后,那么Person类中就有了一个受保护的name属性成员了(只能在本类和派生类中访问及使用)
  constructor(protected name: string = '大甜甜') {
    this.name = name
  }

}
// 实例化对象
const person: Person = new Person('小甜甜')
console.log(person)
person.name = '佐助'
console.log(person.name)
```

- 存取器：让我们可以有效的控制对对象中的成员的访问,通过getters和setters来进行操作

```
// 外部可以传入姓氏和名字数据,同时使用set和get控制姓名的数据,外部也可以进行修改操作
class Person {
  firstName: string // 姓氏
  lastName: string // 名字
  constructor(firstName: string, lastName: string) {
    this.firstName = firstName
    this.lastName = lastName
  }
  // 姓名的成员属性(外部可以访问,也可以修改)
  // 读取器----负责读取数据的
  get fullName() {
    console.log('get中...')
    // 姓名====>姓氏和名字的拼接
    return this.firstName + '_' + this.lastName
  }
  // 设置器----负责设置数据的(修改)
  set fullName(val) {
    console.log('set中...')
    // 姓名---->把姓氏和名字获取到重新的赋值给firstName和lastName
    let names = val.split('_')
    this.firstName = names[0]
    this.lastName = names[1]
  }
}
// 实例化对象
const person: Person = new Person('东方', '不败')
console.log(person)
// 获取该属性成员属性
console.log(person.fullName)
// 设置该属性的数据
person.fullName = '诸葛_孔明'
console.log(person.fullName)
```

- 静态成员：在类中通过static修饰的属性或者方法,那么就是静态的属性及静态的方法,也称之为:静态成员

```
// 静态成员在使用的时候是通过类名.的这种语法来调用的
// 定义一个类
class Person {
  // 类中默认有一个内置的name属性,所以呢,此时会出现错误的提示信息
  // 静态属性
  static name1: string = '小甜甜'
  // 构造函数是不能通过static来进行修饰的
  constructor() {
    // 此时this是实例对象,name1是静态属性,不能通过实例对象直接调用静态属性来使用
    // this.name1 = name
  }
  // 静态方法
  static sayHi() {
    console.log('萨瓦迪卡')
  }
}
// 实例化对象
// const person: Person = new Person()
// 通过实例对象调用的属性(实例属性)
// console.log(person.name1)
// 通过实例对象调用的方法(实例方法)
// person.sayHi()
// 通过类名.静态属性的方式来访问该成员数据
console.log(Person.name1)
// 通过类名.静态属性的方式来设置该成员数据
Person.name1 = '佐助'
console.log(Person.name1)
// 通过类名.静态方法的方式来调用内部的静态的方法
Person.sayHi()
```

- 抽象类：包含抽象方法(抽象方法一般没有任何的具体内容的实现),也可以包含实例方法,抽象类是不能被实例化,为了让子类进行实例化及实现内部的抽象方法

```
// 抽象类的目的或者是作用最终都是为子类服务的
// 定义一个抽象类
abstract class Animal{
  // 抽象属性
  // abstract name:string
  // 抽象方法
  abstract eat()
  // 报错的,抽象方法不能有具体的实现
  // abstract eat(){
  //   console.log('趴着吃,跳着吃')
  // }
  // 实例方法
  sayHi(){
    console.log('您好啊')
  }
}
// 定义一个子类(派生类)Dog
class Dog extends Animal{
  // name:string='小黄'
```

```
  // 重新的实现抽象类中的方法,此时这个方法就是当前Dog类的实例方法了
  eat(){
    console.log('舔着吃,真好吃')
  }
}
// 实例化Dog的对象
const dog:Dog = new Dog()
dog.eat()
// 调用的是抽象类中的实例方法
dog.sayHi()
// console.log(dog.name)
// 不能实例化抽象类的对象
// const ani:Animal = new Animal()
```

函数

- 封装了一些重复使用的代码，在需要的时候直接调用即可

```
// js中的书写方式----->在ts中同样的可以这么写
// 函数声明,命名函数
function add(x, y) { // 求和的函数
    return x + y
}
// 函数表达式,匿名函数
const add2 = function (x, y) {
    return x + y
}

// ts中的书写方式
// 函数声明,命名函数
// 函数中的x和y参数的类型都是string类型的,小括号后面的:string,代表的是该函数的返回值也是string类型的
function add(x: string, y: string): string { // 求和的函数
  return x + y
}
const result1: string = add('111', '222')
console.log(result1)
console.log()
// 函数表达式,匿名函数
// 函数中的x和y参数的类型都是number类型的,小括号后面的:number,代表的是该函数的返回值也是number类型的
const add2 = function (x: number, y: number): number {
  return x + y
}
console.log(add2(10, 20))

// 函数的完整的写法
// add3---->变量名--->函数add3
// (x: number, y: number) => number 当前的这个函数的类型
// function (x: number, y: number): number { return x+y }  就相当于符合上面的这个函数类型的值
const add3: (x: number, y: number) => number = function (x: number, y: number): number {
  return x+y
}
console.log(add3(10,100))
```

- 可选参数和默认参数

```
// 可选参数:函数在声明的时候,内部的参数使用了?进行修饰,那么就表示该参数可以传入也可以不用传入,叫可选参数
// 默认参数:函数在声明的时候,内部的参数有自己的默认值,此时的这个参数就可以叫默认参数

// 定义一个函数:传入姓氏和名字,可以得到姓名(姓氏+名字=姓名)
// 需求:如果不传入任何内容,那么就返回默认的姓氏
// 需求:如果只传入姓氏,那么就返回姓氏
// 需求:如果传入姓氏和名字,那么返回来的就是姓名
const getFullName = function (firstName: string='东方', lastName?: string): string {
  // 判断名字是否传入了
  if (lastName) {
    return firstName + '_' + lastName
  } else {
    return firstName
  }
}
// 函数调用
// 什么也不传入
console.log(getFullName())
// 只传入姓氏
console.log(getFullName('诸葛'))
// 传入姓氏和名字
console.log(getFullName('诸葛','孔明'))
```

- 剩余参数

```
// 剩余参数(rest参数)
// 剩余参数是放在函数声明的时候所有的参数的最后
// ...args:string[] ---->剩余的参数,放在了一个字符串的数组中,args里面
function showMsg(str: string,str2:string, ...args: string[]) {
  console.log(str) // a
  // console.log(str2) // b
  console.log(args) // b ,c ,d ,e
}
showMsg('a','b','c','d','e')
```

- 函数重载：函数名字相同，函数的参数及个数不同

```
// 定义一个函数
// 需求: 有一个add函数，可以接收2个string类型的参数进行拼接，也可以接收2个number类型的参数进行相加

// 函数重载声明
function add(x: string, y: string): string
function add(x: number, y: number): number

// 函数声明
function add(x: string | number, y: string | number): string | number {
  if (typeof x === 'string' && typeof y === 'string') {
    return x + y // 字符串拼接
```

```
  } else if (typeof x === 'number' && typeof y === 'number') {
    return x + y // 数字相加
  }
}
// 函数调用
// 两个参数都是字符串
console.log(add('诸葛', '孔明'))
// 两个参数都是数字
console.log(add(10, 20))
// 此时如果传入的是非法的数据，ts应该给我提示出错误的信息内容,报红色错误的信息
// console.log(add('真香', 10))
// console.log(add(100, '真好'))
```

泛型

- 在定义函数、接口、类的时候不能预先确定要使用的数据的类型，而是在使用函数、接口、类的时候才能确定数据的类型

```
// 需求:定义一个函数,传入两个参数,第一参数是数据,第二个参数是数量,函数的作用:根据数量产生对应个数的数据,存放在一个数组中
// 定义一个函数
function getArr1(value: number, count: number): number[] {
  // 根据数据和数量产生一个数组
  const arr: number[] = []
  for (let i = 0; i < count; i++) {
    arr.push(value)
  }
  return arr
}
const arr1 = getArr1(100.123, 3)
console.log(arr1)
// 定义一个函数,同上,只不过传入的是字符串类型
function getArr2(value: string, count: number): string[] {
  // 根据数据和数量产生一个数组
  const arr: string[] = []
  for (let i = 0; i < count; i++) {
    arr.push(value)
  }
  return arr
}
const arr2 = getArr2('abc', 3)
console.log(arr2)

// 需求:可以传入任意类型的数据,返回来的是存储这个任意类型数据的数组
function getArr3(value: any, count: number): any[] {
  // 根据数据和数量产生一个数组
  const arr: any[] = []
  for (let i = 0; i < count; i++) {
    arr.push(value)
  }
  return arr
```

```
}
const arr1 = getArr3(100.123, 3)
const arr2 = getArr3('abc', 3)
console.log(arr1)
console.log(arr2)
// arr1中存储的是数字类型的数据
// arr2中存储的是字符串类型的数据

function getArr4<T>(value: T, count: number): T[] {
  // 根据数据和数量产生一个数组
  // const arr: T[] = []
  const arr: Array<T> = []
  for (let i = 0; i < count; i++) {
    arr.push(value)
  }
  return arr
}
const arr1 = getArr4<number>(200.12345, 5)
const arr2 = getArr4<string>('abcdefg', 5)
console.log(arr1)
console.log(arr2)
console.log(arr1[0].toFixed(3))
console.log(arr2[0].split(''))
// arr1中存储的是数字类型的数据
// arr2中存储的是字符串类型的数据
```

- 多个泛型参数的函数

```
function getMsg<K, V>(value1: K, value2: V): [K, V] {
    return [value1, value2]
}
const arr1 = getMsg<string,number>('jack',100.2345)
console.log(arr1[0].split(''))
console.log(arr1[1].toFixed(1))
```

- 泛型接口：在定义接口时, 为接口中的属性或方法定义泛型类型,在使用接口时, 再指定具体的泛型类型

```
// 需求:定义一个类,用来存储用户的相关信息(id,名字,年龄)
// 通过一个类的实例对象调用add方法可以添加多个用户信息对象,调用getUserId方法可以根据id获取某个指定的用户信息对象
// 定义一个泛型接口
interface IBaseCRUD<T> {
  data: Array<T>
  add: (t: T) => T
  getUserId: (id: number) => T
}
// 定义一个用户信息的类
class User {
  id?: number // 用户的id  ? 代表该属性可有可无
  name: string // 用户的姓名
  age: number // 用户的年龄
  // 构造函数
```

```
  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}
// 定义一个类,可以针对用户的信息对象进行增加及查询的操作
// CRUD---->create,Read,Update,Delete
class UserCRUD implements IBaseCRUD<User> {
  // 用来保存多个User类型的用户信息对象
  data: Array<User> = []
  //方法用来存储用户信息对象的
  add(user: User): User {
    // 产生id
    user.id = Date.now() + Math.random()
    // 把用户信息对象添加到data数组中
    this.data.push(user)
    return user
  }
  // 方法根据id查询指定的用户信息对象
  getUserId(id: number): User {
    return this.data.find(user => user.id === id)
  }
}
// 实例化添加用户信息对象的类UserCRUD
const userCRUD: UserCRUD = new UserCRUD()
// 调用添加数据的方法
userCRUD.add(new User('jack', 20))
userCRUD.add(new User('tom', 25))
const { id } = userCRUD.add(new User('lucy', 23))
userCRUD.add(new User('rousi', 21))
console.log(userCRUD.data)
// 根据id查询用户信息对象数据
const user = userCRUD.getUserId(id)
console.log(user)
```

- 泛型类：定义一个类,类中的属性值的类型是不确定,方法中的参数及返回值的类型也是不确定

```
// 定义一个泛型类
class GenericNumber<T>{
  // 默认的属性的值的类型是泛型类型
  defaultValue: T
  add: (x: T, y: T) => T
}
// 在实例化类的对象的时候,再确定泛型的类型
const g1: GenericNumber<number> = new GenericNumber<number>()
// 设置属性值
g1.defaultValue = 100
// 相加的方法
g1.add = function (x, y) {
  return x + y
}
console.log(g1.add(g1.defaultValue,20))
// 在实例化类的对象的时候,再确定泛型的类型
```

```
const g2: GenericNumber<string> = new GenericNumber<string>()
// 设置属性值
g2.defaultValue = '哈哈'
// 相加的方法
g2.add = function (x, y) {
  return x + y
}
console.log(g2.add('帅杨',g2.defaultValue))
```

- 泛型约束

```
// 如果我们直接对一个泛型参数取 length 属性，会报错，因为这个泛型根本就不知道它有这个属性
// 定义一个接口,用来约束将来的某个类型中必须要有length这个属性
interface ILength{
    // 接口中有一个属性length
    length:number
}
function getLength<T extends ILength>(x: T): number {
    return x.length
}
console.log(getLength<string>('what are you no sha lei'))
// console.log(getLength<number>(123))
```

声明文件

```
// 引入第三方的库jQuery
// import jQuery from 'jquery'
// 使用jQuery
// jQuery('选择器')

/*
当使用第三方库时，我们需要引用它的声明文件，才能获得对应的代码补全、接口提示等功能。
声明语句：如果需要ts对新的语法进行检查，需要要加载了对应的类型说明代码
  declare var jQuery: (selector: string) => any;
声明文件：把声明语句放到一个单独的文件（jQuery.d.ts）中，ts会自动解析到项目中所有声明文件
下载声明文件：npm install @types/jquery --save-dev
*/

// import jQuery from 'jquery'
// jQuery('选择器')
```

内置对象

- JavaScript 中有很多内置对象，可以直接在 TypeScript 中当做定义好了的类型。

```
/* 1.ECMAScript 的内置对象
   Boolean
   Number
   String
```

```
   Date
   RegExp
   Error*/
let b: Boolean = new Boolean(1)
let n: Number = new Number(true)
let s: String = new String('abc')
let d: Date = new Date()
let r: RegExp = /^1/
let e: Error = new Error('error message')
b = true
console.log(b)
let bb: boolean = new Boolean(2)  // error

/* 2.BOM 和 DOM 的内置对象
   Window
   Document
   HTMLElement
   DocumentFragment
   Event
   NodeList*/
const div: HTMLElement = document.getElementById('test')
const divs: NodeList = document.querySelectorAll('div')
document.addEventListener('click', (event: MouseEvent) => {
  console.dir(event.target)
})
const fragment: DocumentFragment = document.createDocumentFragment()
```