1. interface简介

interface是对象的模板，可以看作是一种类型约定，中文译为`接口`，使用了某个模板的对象，就拥有了指定的类型结构。

指定了一个对象模板，有三个属性，任何要实现这个接口的对象，都必须部署这三个属性，并且符合规定的类型。

```
interface Person {
  firstName: string;
  lastName: string;
  age: number;
}
```

使用方括号运算符可以取出interface某个属性的类型

```
interface Foo {
  a: string;
}

type A = Foo['a']; // string
```

interface 可以表示对象的各种语法，它的成员有5种形式。

- 对象属性
- 对象的属性索引
- 对象方法
- 函数
- 构造函数

1. 对象属性

属性之间使用分号或者逗号分隔，最后一个属性结尾的分号或逗号可以省略

如果属性是可选的，就在属性名后面加一个问号

如果属性是只读的，需要加上readonly修饰符

```
interface Point {
  x: number;
  y?: number;
  readonly a: string;
}
```

2. 对象的属性索引

`[prop: string]`就是属性的字符串索引，表示属性名只要是字符串，都符合类型要求。属性索引 共有string、number和symbol三种类型。

```
interface A {
  [prop: string]: number;
}
```

一个接口中，最多只能定义一个字符串索引，因为字符串索引壶约束该类型中所有名字为字符串的属性。

这里属性名为字符串类型，属性值为number类型，但是此时又定义字符串属性a，属性值是布尔值的就会出错，如果还是number类型，就不会报错。

```
interface MyObj {
  [prop: string]: number;

  a: boolean;      // 编译错误
}
```

属性若为数值索引，其实指定的就是数组类型。

一个接口中最多只能定义一个数值索引。数值索引会约束所有名称为数值的属性。

```
interface A {
  [prop: number]: string;
}

const obj:A = ['a', 'b', 'c'];
```

如果一个 interface 同时定义了字符串索引和数值索引，那么数值索引必须服从于字符串索引。因为在 JavaScript 中，数值属性名最终是自动转换成字符串属性名

3. 对象的方法

对象的方法共有三种写法。

第一种：参数后接冒号+类型，返回值类型在参数列表圆括号外

第二种：箭头函数形式

第三种：圆括号定义参数类型，参数外定义返回值类型

```
// 写法一
interface A {
  f(x: boolean): string;
}

// 写法二
interface B {
  f: (x: boolean) => string;
}

// 写法三
interface C {
  f: { (x: boolean): string };
}
```

属性名可以采用表达式，所以下面的写法也是可以的。

```
const f = 'f';

interface A {
  [f](x: boolean): string;
}
```

重载描述

interface里面的函数重载，不需要给出实现，但是由于对象内部定义方法时，无法使用函数重载的语法，所以需要先在对象外部给出函数方法的实现，然后再赋值。

这是设置接口A的方法f()有函数重载，需要额外定义一个函数MyFunc()来实现这个重载，之后再部署接口A的对象a的属性f等于函数MyFunc()就可以了。

```
interface A {
  f(): number;
  f(x: boolean): boolean;
  f(x: string, y: string): string;
}

function MyFunc(): number;
function MyFunc(x: boolean): boolean;
function MyFunc(x: string, y: string): string;
function MyFunc(
  x?:boolean|string, y?:string
):number|boolean|string {
  if (x === undefined && y === undefined) return 1;
  if (typeof x === 'boolean' && y === undefined) return true;
  if (typeof x === 'string' && typeof y === 'string') return 'hello';
  throw new Error('wrong parameters');
}

const a:A = {
  f: MyFunc
}
```

4. 函数

可以用来声明独立的函数

```
interface Add {
  (x:number, y:number): number;
}

const myAdd:Add = (x,y) => x + y;
```

5. 构造函数

interface内部使用new关键字，表示构造函数

在ts中构造函数特指具有`constructor`属性的类，见**Class**

```
interface ErrorConstructor {
  new (message?: string): Error;
}
```

2. interface继承

interface可以继承其他类型

1. 继承interface

使用extends关键字，继承其他interface类型

Circle继承了Shape，Circle是子接口，Shape是父接口。

```
interface Shape {
  name: string;
}

interface Circle extends Shape {
  radius: number;
}
```

`extends`关键字会从继承的接口里面拷贝属性类型。

多重接口继承，实际上相当于多个父接口的合并。

```
interface Style {
  color: string;
}

interface Shape {
  name: string;
}

interface Circle extends Style, Shape {
  radius: number;
}
```

如果子接口与父接口存在同名属性，那么子接口的属性会覆盖父接口的属性。注意，子接口与父接口的同名属性必须是类型兼容的，不能有冲突，否则会报错。

```
interface Foo {
  id: string;
}

interface Bar extends Foo {
  id: number; // 报错
}
```

多重继承时，如果多个父接口存在同名属性，那么这些同名属性不能有类型冲突，否则会报错。

2. interface继承type

可以继承type命令定义的**对象类型，如果其他类型则无法继承。**

```
type Country = {
  name: string;
  capital: string;
}

interface CountryWithPop extends Country {
  population: number;
}
```

3. 继承class

还可以继承class，即继承该类的所有成员

```
class A {
  x:string = '';

  y():boolean {
    return true;
  }
}

interface B extends A {
  z: number
}
```

3. 接口合并

将多个同名接口合并成一个接口。

将两个同名为Box的接口合成一个接口，有三个属性。

```
interface Box {
  height: number;
  width: number;
}

interface Box {
  length: number;
}
```

为什么要这样设计?

主要是为了兼容 js 。因为在js中常常对全局对象或者外部库，添加自己的属性和方法。那么，只要使用 interface 给出这些自定义属性和方法的类型，就能自动跟原始的 interface 合并，使得扩展外部类型非常方便。

同名接口合并时，同一个属性如果有多个类型声明，彼此不能有类型冲突。

```ts
interface A {
  a: number;
}

interface A {
  a: string; // 报错
}
```

同名接口合并时，如果同名方法有不同的类型声明，那么会发生函数重载。而且，后面的定义比前面的定义具有更高的优先级。优先级最高就是最先执行该类型。

这里Cloner()方法有不同的类型声明，会发生函数重载，这时，越靠后的定义，优先级越高，排在函数重载的越前面。比如，`clone(animal: Animal)`是最先出现的类型声明，就排在函数重载的最后，属于`clone()`函数最后匹配的类型。

```ts
interface Cloner {
  clone(animal: Animal): Animal;
}

interface Cloner {
  clone(animal: Sheep): Sheep;
}

interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
}

// 等同于
interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
  clone(animal: Sheep): Sheep;
  clone(animal: Animal): Animal;
}
```

同名方法之中，如果有一个参数是字面量类型，字面量类型有更高的优先级。

```ts
interface A {
  f(x:'foo'): boolean;
}

interface A {
  f(x:any): void;
}

// 等同于
interface A {
  f(x:'foo'): boolean;
  f(x:any): void;
}
```

如果两个 interface 组成的联合类型存在同名属性，那么该属性的类型也是联合类型。

```ts
interface Circle {
  area: bigint;
}

interface Rectangle {
  area: number;
}

declare const s: Circle | Rectangle;

s.area;   // bigint | number
```

4. 与type的异同

`interface`命令与`type`命令作用类似，都可以表示对象类型。

很多对象类型既可以用 interface 表示，也可以用 type 表示。两者往往也可以换用，几乎所有的 interface 命令都可以改写为 type 命令。

相同点

- 都能为对象类型起名

```ts
type Country = {
  name: string;
  capital: string;
}

interface Country {
  name: string;
  capital: string;
}
```

不同点

- type能够表示非对象类型，而interface只能表示对象类型（数组、对象、函数）
- interface可以继承其他类型，type则不支持

继承主要是为了添加属性，type定义的对象类型如果想添加属性，需要使用&运算符，从新定义一个类型。

&运算符表示具有两个类型的特征，可以起到两个对象类型的合并作用。

interface添加属性，使用extends,采用继承写法。

```ts
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}
```

继承时，type 和 interface 是可以换用的。interface 可以继承 type。

```
type Foo = { x: number; };

interface Bar extends Foo {
  y: number;
}

```

type 也可以继承 interface。

```
interface Foo {
  x: number;
}

type Bar = Foo & { y: number; };
```

- 同名interface会自动合并，同名type会报错
- interface不能包含属性映射，type可以，见**映射**

```
interface Point {
  x: number;
  y: number;
}

// 正确
type PointCopy1 = {
  [Key in keyof Point]: Point[Key];
};

// 报错
interface PointCopy2 {
  [Key in keyof Point]: Point[Key];
};
```

- this关键字只能用interface

返回值为this，type会报错

```
// 正确
interface Foo {
  add(num:number): this;
};

// 报错
type Foo = {
  add(num:number): this;
};
```

- type可以扩展原始数据类型，interface不行

interface不能继承string，只能扩展命名的类型和类

```ts
// 正确
type MyStr = string & {
  type: 'new'
};

// 报错
interface MyStr extends string {
  type: 'new'
}
```

- interface无法表达某些复杂类型(比如交叉或联合类型)，但是type可以

类型`AorB`是一个联合类型，`AorBwithName`则是为`AorB`添加一个属性。这两种运算，`interface`都没法表达。

如果有复杂的类型运算，那么没有其他选择只能使用`type`；一般情况下，`interface`灵活性比较高，容易扩充类型或自动合并，可以优先使用。

```ts
type A = { /* ... */ };
type B = { /* ... */ };

type AorB = A | B;
type AorBwithName = AorB & {
  name: string
};
```