TypeScript 的 interface 接口

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。

typescript

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

上面示例中，定义了一个接口 `Person` ，它指定一个对象模板，拥有三个属性 `firstName` 、`lastName` 和 `age` 。任何实现这个接口的对象，都必须部署这三个属性，并且必须符合规定的类型。

实现该接口很简单，只要指定它作为对象的类型即可。

typescript

```
const p: Person = {
  firstName: "John",
  lastName: "Smith",
  age: 25,
};
```

上面示例中，变量 `p` 的类型就是接口 `Person` ，所以必须符合 `Person` 指定的结构。

方括号运算符可以取出 interface 某个属性的类型。

typescript

```
interface Foo {
  a: string;
```

```
}

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

上面示例中， `Foo['a']` 返回属性 `a` 的类型，所以类型 `A` 就是 `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

typescript

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

上面示例中， `x` 和 `y` 都是对象的属性，分别使用冒号指定每个属性的类型。

属性之间使用分号或逗号分隔，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。

typescript

```
interface Foo {
  x?: string;
}
```

如果属性是只读的，需要加上 `readonly` 修饰符。

typescript

```
interface A {
  readonly a: string;
}
```

（2）对象的属性索引

typescript

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

上面示例中，`[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。

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

typescript

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

  a: boolean; // 编译错误
}
```

上面示例中，属性索引指定所有名称为字符串的属性，它们的属性值必须是数值（ `number` ）。属性 `a` 的值为布尔值就报错了。

属性的数值索引，其实是指定数组的类型。

typescript

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

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

上面示例中，`[prop: number]` 表示属性名的类型是数值，所以可以用数组对变量 `obj` 赋值。

同样的，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。

如果一个 interface 同时定义了字符串索引和数值索引，那么数值索性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。

typescript

```typescript
interface A {
  [prop: string]: number;
  [prop: number]: string; // 报错
```

```
}

interface B {
  [prop: string]: number;
  [prop: number]: number; // 正确
}
```

上面示例中，数值索引的属性值类型与字符串索引不一致，就会报错。数值索引必须兼容字符串索引的类型声明。

（3）对象的方法

对象的方法共有三种写法。

typescript

```
// 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。

typescript

```
const f = "f";

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

类型方法可以重载。

typescript

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

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，所以需要额外在对象外部给出函数方法的实现。

typescript

```typescript
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,
};
```

上面示例中，接口 `A` 的方法 `f()` 有函数重载，需要额外定义一个函数 `MyFunc()` 实现这个重载，然后部署接口 `A` 的对象 `a` 的属性 `f` 等于函数 `MyFunc()` 就可以了。

（4）函数

interface 也可以用来声明独立的函数。

typescript

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

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

上面示例中，接口 `Add` 声明了一个函数类型。

（5）构造函数

interface 内部可以使用 `new` 关键字，表示构造函数。

typescript

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

上面示例中，接口 `ErrorConstructor` 内部有 `new` 命令，表示它是一个构造函数。

TypeScript 里面，构造函数特指具有 `constructor` 属性的类，详见《Class》一章。

---

interface 的继承

interface 可以继承其他类型，主要有下面几种情况。

interface 继承 interface

interface 可以使用 `extends` 关键字，继承其他 interface。

typescript

```
interface Shape {
  name: string;
}

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

上面示例中， `Circle` 继承了 `Shape` ，所以 `Circle` 其实有两个属性 `name` 和 `radius` 。这时，`Circle` 是子接口， `Shape` 是父接口。

`extends` 关键字会从继承的接口里面拷贝属性类型，这样就不必书写重复的属性。

interface 允许多重继承。

typescript

```typescript
interface Style {
  color: string;
}

interface Shape {
  name: string;
}

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

上面示例中， `Circle` 同时继承了 `Style` 和 `Shape` ，所以拥有三个属性 `color` 、 `name` 和 `radius` 。

多重接口继承，实际上相当于多个父接口的合并。

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。

typescript

```typescript
interface Foo {
  id: string;
}

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

上面示例中， `Bar` 继承了 `Foo` ，但是两者的同名属性 `id` 的类型不兼容，导致报错。

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。

typescript

```typescript
interface Foo {
  id: string;
}

interface Bar {
```

```
  id: number;
}

// 报错
interface Baz extends Foo, Bar {
  type: string;
}
```

上面示例中， `Baz` 同时继承了 `Foo` 和 `Bar` ，但是后两者的同名属性 `id` 有类型冲突，导致报错。

interface 继承 type

interface 可以继承 `type` 命令定义的对象类型。

typescript

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

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

上面示例中， `CountryWithPop` 继承了 `type` 命令定义的 `Country` 对象，并且新增了一个 `population` 属性。

注意，如果 `type` 命令定义的类型不是对象，interface 就无法继承。

interface 继承 class

interface 还可以继承 class，即继承该类的所有成员。关于 class 的详细解释，参见下一章。

typescript

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

  y(): boolean {
    return true;
  }
}
```

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

上面示例中， `B` 继承了 `A` ，因此 `B` 就具有属性 `x` 、 `y()` 和 `z` 。

实现 `B` 接口的对象就需要实现这些属性。

typescript

```typescript
const b: B = {
  x: "",
  y: function () {
    return true;
  },
  z: 123,
};
```

上面示例中，对象 `b` 就实现了接口 `B` ，而接口 `B` 又继承了类 `A` 。

某些类拥有私有成员和保护成员，interface 可以继承这样的类，但是意义不大。

typescript

```typescript
class A {
  private x: string = "";
  protected y: string = "";
}

interface B extends A {
  z: number;
}

// 报错
const b: B = {
  /* ... */
};

// 报错
class C implements B {
  // ...
}
```

上面示例中， `A` 有私有成员和保护成员， `B` 继承了 `A` ，但无法用于对象，因为对象不能实现这些成员。这导致 `B` 只能用于其他 class，而这时其他 class 与 `A` 之间不构成父类和子类的关系，使得 `x` 与 `y` 无法部署。

接口合并

多个同名接口会合并成一个接口。

typescript

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

interface Box {
  length: number;
}
```

上面示例中，两个 `Box` 接口会合并成一个接口，同时有 `height` 、 `width` 和 `length` 三个属性。

这样的设计主要是为了兼容 JavaScript 的行为。JavaScript 开发者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。

举例来说，Web 网页开发经常会对 `windows` 对象和 `document` 对象添加自定义属性，但是 TypeScript 会报错，因为原始定义没有这些属性。解决方法就是把自定义属性写成 interface，合并进原始定义。

typescript

```
interface Document {
  foo: string;
}

document.foo = "hello";
```

上面示例中，接口 `Document` 增加了一个自定义属性 `foo` ，从而就可以在 `document` 对象上使用自定义属性。

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。

typescript

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

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

上面示例中，接口 `A` 的属性 `a` 有两个类型声明，彼此是冲突的，导致报错。

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。

typescript

```typescript
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;
}
```

上面示例中， `clone()` 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。

这个规则有一个例外。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。

typescript

```typescript
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;
}
```

上面示例中， `f()` 方法有一个类型声明是，参数 `x` 是字面量类型，这个类型声明的优先级最高，会排在函数重载的最前面。

一个实际的例子是 Document 对象的 `createElement()` 方法，它会根据参数的不同，而生成不同的 HTML 节点对象。

typescript

```typescript
interface Document {
  createElement(tagName: any): Element;
}
interface Document {
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
  createElement(tagName: string): HTMLElement;
  createElement(tagName: "canvas"): HTMLCanvasElement;
}

// 等同于
interface Document {
  createElement(tagName: "canvas"): HTMLCanvasElement;
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
  createElement(tagName: string): HTMLElement;
  createElement(tagName: any): Element;
}
```

上面示例中， `createElement()` 方法的函数重载，参数为字面量的类型声明会排到最前面，返回具体的 HTML 节点对象。类型越不具体的参数，排在越后面，返回通用的 HTML 节点对象。

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。

typescript

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

interface Rectangle {
  area: number;
}

declare const s: Circle | Rectangle;

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

上面示例中，接口 `Circle` 和 `Rectangle` 组成一个联合类型 `Circle | Rectangle` 。因此，这个联合类型的同名属性 `area` ，也是一个联合类型。本例中的 `declare` 命令表示变量 `s` 的具体定义，由其他脚本文件给出，详见《declare 命令》一章。

---

interface 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。

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

它们的相似之处，首先表现在都能为对象类型起名。

typescript

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

interface Coutry {
  name: string;
```

```
  capital: string;
}
```

上面示例是 `type` 命令和 `interface` 命令，分别定义同一个类型。

`class` 命令也有类似作用，通过定义一个类，同时定义一个对象类型。但是，它会创造一个值，编译后依然存在。如果只是单纯想要一个类型，应该使用 `type` 或 `interface` 。

interface 与 type 的区别有下面几点。

（1） `type` 能够表示非对象类型，而 `interface` 只能表示对象类型（包括数组、函数等）。

（2） `interface` 可以继承其他类型， `type` 不支持继承。

继承的主要作用是添加属性， `type` 定义的对象类型如果想要添加属性，只能使用 `&` 运算符，重新定义一个类型。

typescript

```
type Animal = {
  name: string;
};

type Bear = Animal & {
  honey: boolean;
};
```

上面示例中，类型 `Bear` 在 `Animal` 的基础上添加了一个属性 `honey` 。

上例的 `&` 运算符，表示同时具备两个类型的特征，因此可以起到两个对象类型合并的作用。

作为比较， `interface` 添加属性，采用的是继承的写法。

typescript

```
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。

typescript

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

interface Bar extends Foo {
  y: number;
}
```

type 也可以继承 interface。

typescript

```
interface Foo {
  x: number;
}

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

（3）同名 `interface` 会自动合并，同名 `type` 则会报错。也就是说，TypeScript 不允许使用 `type` 多次定义同一个类型。

typescript

```
type A = { foo: number }; // 报错
type A = { bar: number }; // 报错
```

上面示例中， `type` 两次定义了类型 `A` ，导致两行都会报错。

作为比较， `interface` 则会自动合并。

typescript

```
interface A {
  foo: number;
}
interface A {
  bar: number;
}

const obj: A = {
  foo: 1,
  bar: 1,
};
```

上面示例中， `interface` 把类型 `A` 的两个定义合并在一起。

这表明，interface 是开放的，可以添加属性，type 是封闭的，不能添加属性，只能定义新的 type。

（4） `interface` 不能包含属性映射（mapping）， `type` 可以，详见《映射》一章。

typescript

```typescript
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];
};
```

（5） `this` 关键字只能用于 `interface` 。

typescript

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

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

上面示例中，type 命令声明的方法 `add()` ，返回 `this` 就报错了。interface 命令没有这个问题。

下面是返回 `this` 的实际对象的例子。

typescript

```typescript
class Calculator implements Foo {
  result = 0;
  add(num: number) {
```

```
    this.result += num;
    return this;
  }
}
```

（6）type 可以扩展原始数据类型，interface 不行。

typescript

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

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

上面示例中，type 可以扩展原始数据类型 string，interface 就不行。

（7） `interface` 无法表达某些复杂类型（比如交叉类型和联合类型），但是 `type` 可以。

typescript

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

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

上面示例中，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。

综上所述，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，便于扩充类型或自动合并，建议优先使用。

推荐机场 → 25元/月，500G 购买。

最后更新: 2023/8/13 15:25

Previous page
对象

Next page
类