interface

interface(接口) 是对象的模板，使用了某个模板的对象，就拥有了指定的类型结构。

```
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。

只要指定该接口作为对象的类型即可实现该接口。

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

上例中，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。

```
interface Foo {
  a: string;
}
type A = Foo['a']; // type A = string
```

上例中，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

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

```
  y: number;
}
```

上例中，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。

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

如果属性是只读的，需要加上readonly修饰符。

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

const x: A = { a : '1' };
x.a = '2'; // Cannot assign to 'a' because it is a read-only property.
```

（2）对象的属性索引

```
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。

```
interface A {
  [prop: string]: number;
  [prop: number]: number;
  [prop: symbol]: number;
  [prop: bigint]: number; // An index signature parameter type must be 'string', 'number', 'symbol', or a template literal type. 索引签名参数类型必须是“字符串”、“数字”、“符号”或模板文字类型。
}
```

```
// 属性索引是 string 类型
interface A {
  [prop: string]: number;
```

```
}
let a:A = { 1: 1 } // 正确
let b:A = { '1': 1 } // 正确
let c:A = { [Symbol()]: 1 } // 正确
```

```
// 属性索引是 number 类型
interface A {
  [prop: number]: number;
}
let a:A = { 1: 1 } // 正确
let b:A = { '1': 1 } // 正确
let c:A = { [Symbol()]: 1 } // 正确
```

```
// 属性索引是 symbol 类型
interface A {
  [prop: symbol]: number;
}
let a:A = { 1: 1 };
// Type '{ 1: number; }' is not assignable to type 'A'. Object literal may only
specify known properties, and '1' does not exist in type 'A'. 类型“{1:数字;}'不能赋
值给类型'A'。对象字面量只能指定已知的属性，并且'1'在类型'A'中不存在。

let b:A = { '1': 1 }; // 和上面报错一致
let c:A = { [Symbol()]: 1 };
```

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。

```
interface MyObj {
  [prop: string]: number;
  a: boolean; // Property 'a' of type 'boolean' is not assignable to 'string'
index type 'number'. 类型为布尔的属性'a'不能赋值给索引类型为'number'的'string'。
}
```

上例中，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。将 `a` 的类型改为 `number` 就对了。

属性的数值索引，其实是指定数组的类型。

```
interface A {
  [prop: number]: string;
}
const obj:A = ['a', 'b', 'c'];
```

上例中，`[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。**

```
interface A {
  [prop: string]: number;
  [prop: number]: string; // 'number' index type 'string' is not assignable to
'string' index type 'number'.
}

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。

```
// 写法一
interface A {
  f(x: boolean): string;
}
let bool1 = true;
let a: A = { f: (bool1) => '12'};
// 等价于
let a: A = { f: function(bool1) { return '12'}};

// 写法二
interface B {
  f: (x: boolean) => string;
}
let bool2 = true;
let b: B = { f: (bool2) => '12'};

// 写法三
interface C {
  f: { (x: boolean): string };
}
let bool3 = true;
let c: C = { f: (bool3) => '12'};
```

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; }

（4）函数

interface 也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;
```

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。

```
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。

1. interface 的继承

interface 可以继承其他类型，主要有下面几种情况。

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

```
interface Shape {
  name: string;
}
interface Circle extends Shape {
  radius: number;
}
const c: Circle = {name: 'c', radius: 1};
```

上例中，`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。

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

上例中，`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。

```
interface Foo {
  id: string;
}
interface Bar extends Foo {
  id: number;
  // Interface 'Bar' incorrectly extends interface 'Foo'. Types of property 'id'
are incompatible. Type 'number' is not assignable to type 'string'. 接口'Bar'错误地
扩展了接口'Foo'。属性'id'的类型不兼容。类型'number'不能赋值给类型'string'。
}
```

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。

```
interface Foo {
  id: string;
}
interface Bar {
  id: number;
}
// 报错
interface Baz extends Foo, Bar { // Named property 'id' of types 'Foo' and 'Bar'
are not identical. 类型'Foo'和'Bar'的命名属性'id'不相同。
  type: string;
}
```

1.2. interface 继承 type

`interface` 可以继承 `type` 命令定义的对象类型。

```

```

```
type Country = {
  name: string;
  capital: string;
}
interface CountryWithPop extends Country {
  population: number;
}

type NewProps = {
    [Prop in keyof CountryWithPop]: boolean;
};
// keyof 运算符返回对象的键组成的联合类型，in 运算符取出联合类型中每一个成员。[Prop in keyof CountryWithPop] 取出了 'population', 'name', 'capital'
// type NewProps = { population: boolean; name: boolean; capital: boolean; }
```

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

1.3. interface 继承 class

interface 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
}
type props = {
  [prop in keyof B]: boolean;
}
// type props = { z: boolean; x: boolean; y: boolean; }
```

实现 `B` 接口的对象就需要实现这些属性。

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

2. 接口合并

多个同名接口会合并成一个接口。

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

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

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。

```
interface Document {
  foo: string;
}
document.foo = 'hello';
```

上例中，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。

```
interface A {
  a: number;
}
interface A {
  a: string; // Subsequent property declarations must have the same type.
Property 'a' must be of type 'number', but here has type 'string'. 随后的属性声明必须具有相同的类型。属性'a'必须是'number'类型，但这里有'string'类型。
}
```

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。

```
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。

```
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;
}
```

上例中，`f()` 方法有一个类型声明的参数 `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。

```
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。

```
interface Circle {
  area: bigint;
}
interface Rectangle {
  area: number;
}
declare const s: Circle | Rectangle;
s.area;  // bigint | number
```

上例中，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。

3. 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。

很多对象类型既可以用 `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。

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

`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。

```
type Animal = {
  name: string
}
type Bear = Animal & {
  honey: boolean
}
```

上例中，类型 `Bear` 在 `Animal` 的基础上添加了一个属性 `honey`。`&` 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。

```
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` 也可以继承 `interface`。

```
type Foo = { x: number; };
interface Bar extends Foo {
  y: number;
}
```

```
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。**

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

type B = { foo:number }; // Duplicate identifier 'B'.
type B = { bar:number }; // Duplicate identifier 'B'.
```

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。

```
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。

```
// 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。

```
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。

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

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

上例中，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。

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

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

上例中，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。**