1. 简介

使用大括号表示对象，在大括号内部声明每个属性和方法的类型。

```
const obj:{
  x:number;
  y:number;
} = { x: 1, y: 1 };
```

属性的类型可以用分号结尾，也可以用逗号结尾。最后一个属性后面，可以写分号或逗号，也可以不写。

```
// 属性类型以分号结尾
type MyObj = {
  x:number;
  y:number;
};

// 属性类型以逗号结尾
type MyObj = {
  x:number,
  y:number,
};

```

一旦声明了类型，对象赋值时，就不能缺少指定的属性，也不能有多余的属性。

读写不存在的属性也会报错，也不能删除类型声明中存在的属性，修改属性值是可以的。

ts不区分对象自身的属性和继承属性，一律视为对象的属性。

读取属性的类型

使用方括号读取属性的类型

```
type User = {
  name: string,
  age: number
};
type Name = User['name']; // string
```

interface

除了`type`命令可以为对象类型声明一个别名，ts 还提供了`interface`命令，可以把对象类型提炼为一个接口。

写法一：type命令

写法二：interface命令

```
// 写法一
```

```
type MyObj = {
  x:number;
  y:number;
};

const obj:MyObj = { x: 1, y: 1 };

// 写法二
interface MyObj {
  x: number;
  y: number;
}

const obj:MyObj = { x: 1, y: 1 };
```

2. 可选属性

如果某个属性后面加一个问号(?)，则这个属性是可选的，可选属性等同于允许赋值为undefined

```
type User = {
  firstName: string;
  lastName?: string;
};

// 等同于
type User = {
  firstName: string;
  lastName?: string|undefined;
};
```

读取一个没有赋值的可选属性时，返回`undefined`。

```
type MyObj = {
  x: string,
  y?: string
};

const obj:MyObj = { x: 'hello' };
obj.y.toLowerCase() // 报错
```

读取可选属性之前，必须检查一下是否为`undefined`。

如果将编译选项`ExactOptionalPropertyTypes`和`strictNullChecks`同时打开，则可选属性就不能设为undefined。

```
// 打开 ExactOptionsPropertyTypes 和 strictNullChecks
const obj: {
  x: number;
  y?: number;
} = { x: 1, y: undefined }; // 报错
```

可选属性与显示设置为undefined的必选属性是不等价的。

```
type A = { x:number, y?:number };
type B = { x:number, y:number|undefined };

const ObjA:A = { x: 1 }; // 正确
const ObjB:B = { x: 1 }; // 报错
```

3. 只读属性

如何使用

第一种方法：在属性名前加上readonly关键字，表示这个属性是只读属性，不能修改。只读属性只能在对象初始化期间赋值，此后就不能修改该属性。

```
interface MyInterface {
  readonly prop: number;
}
```

第二种方法：就是在赋值时，在对象的后面加上只读断言**`as const`**

就会变成只读对象了并且不能修改属性。

```
const myUser = {
  name: "Sabrina",
} as const;

myUser.name = "Cynthia"; // 报错
```

注意点：如果变量也声明了类型，则会以变量声明的类型为准。

这里虽然使用了只读断言`as const`，但是其变量声明的类型不是只读属性，所以可以修改。

```
const myUser:{ name: string } = {
  name: "Sabrina",
} as const;

myUser.name = "Cynthia"; // 正确
```

属性值为对象

如果属性值是一个对象，`readonly`修饰符并不禁止修改该对象的属性，只是禁止完全替换掉该对象。【相当于js的引用数据类型】

```
interface Home {
  readonly resident: {
    name: string;
    age: number
  };
}

const h:Home = {
```

```
  resident: {
    name: 'Vicky',
    age: 42
  }
};

h.resident.age = 32; // 正确
h.resident = {
  name: 'Kate',
  age: 23
} // 报错
```

如果一个对象有两个引用，即两个变量对应同一个对象，其中一个变量是可写的，另一个变量是只读的，那么从可写变量修改属性，会影响到只读变量。【相当于js中的浅拷贝】

4. 属性名的索引类型

对象的属性很多，如果一个个声明类型会很麻烦，并且有时候不知道对象有多少个属性，比如外部API，所以ts**可以采取属性名表达式的写法来描述类型。**

属性名的字符串索引

写法：采用表达式写法，写在方括号里面。其中`property`表示属性名，可以随意命名，它的类型是string，属性值的类型为string。

其中属性的类型有三种为string、number和symbol。

```
type MyObj = {
  [property: string]: string
};

const obj:MyObj = {
  foo: 'a',
  bar: 'b',
  baz: 'c',
};
```

因为对象可能存在多种类型的属性名索引，**但是不能同时存在数值索引和字符串索引，这是因为js内部所有的数值属性名都会自动转为字符串属性名，就会造成一样。**

既可以声明属性名索引，也能声明具体的单个属性名，如果单个属性名符合属性名索引的范围，二者就不能冲突，否则会报错。

这里属性名foo符合属性名的字符串索引，但是二者的属性值类型不一样，就会报错。

```
type MyType = {
  foo: boolean; // 报错
  [x: string]: string;
}
```

缺点

属性的索引类型写法，建议谨慎使用，因为属性名的声明太宽泛，约束太少。另外，属性名的数值索引不宜用来声明数组，因为采用这种方式声明数组，就不能使用各种数组方法以及`length`属性，因为类型里面没有定义这些东西。

```
type MyArr = {
  [n:number]: number;
};

const arr:MyArr = [1, 2, 3];
arr.length // 报错

```

上面示例中，读取`arr.length`属性会报错，因为类型`MyArr`没有这个属性。

5. 解构赋值

解构赋值的类型写法和对象声明类型是一样的，因为解构里的冒号，js指定了其他的用途。

```
const {id, name, price}:{
  id: string;
  name: string;
  price: number
} = product;
```

注意：

这里的冒号其实是改为新变量，而不是赋值类型，所以会报错。

```
function draw({
  shape: Shape,
  xPos: number = 100,
  yPos: number = 100
}) {
  let myShape = shape; // 报错
  let x = xPos; // 报错
}
```

6. 结构类型原则

只要对象 B 满足 对象 A 的结构特征，TypeScript 就认为对象 B 兼容对象 A 的类型，这称为“结构类型”原则（structural typing）。

下面代码中，对象A有一个属性x，类型为number的，对象B满足这个特征，所以说只要使用A的地方就可以使用B。所以B可以赋值给A。

```ts
type A = {
  x: number;
};

type B = {
  x: number;
  y: number;
};
```

根据“结构类型”原则，ts 检查某个值是否符合指定类型时，并不是检查这个值的类型名（即“名义类型”），而是检查这个值的结构是否符合要求（即“结构类型”）。

ts 之所以这样设计，是为了符合 js 的行为。js 并不关心对象是否严格相似，只要某个对象具有所要求的属性，就可以正确运行。

如果类型 B 可以赋值给类型 A，ts 就认为 B 是 A 的子类型（subtyping），A 是 B 的父类型。子类型满足父类型的所有结构特征，同时还具有自己的特征。凡是可以使用父类型的地方，都可以使用子类型，即子类型兼容父类型。

这样的设计可能会造成报错，比如对象函数的参数处理，如果直接遍历就有可能造成问题，因为只要符合该参数类型的值都能传入。

7. 严格字面量检查

如果对象使用字面量表示，会触发 ts 的严格字面量检查（strict object literal checking）。如果字面量的结构跟类型定义的不一样（比如多出了未定义的属性），就会报错。

定义类型少就会报错。

```ts
const point:{
  x:number;
  y:number;
} = {
  x: 1,
  y: 1,
  z: 1 // 报错
};
```

但是如果等号右边不是字面量，而是一个变量，根据结构类型原则，就不会报错。

```ts
const myPoint = {
  x: 1,
  y: 1,
  z: 1
};

const point:{
  x:number;
  y:number;
} = myPoint; // 正确
```

ts对字面量进行严格检查，是为了防止拼写错误，可以使用一个变量赋值，就不会进行严格检查。

由于严格字面量检查，所以对于字面量对象传入函数时必须很小心，不能有多余的属性。

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

function computeDistance(point: Point) { /*...*/ }

computeDistance({ x: 1, y: 2, z: 3 }); // 报错
computeDistance({x: 1, y: 2}); // 正确
```

编译器选项suppressExcessPropertyErrors，值为true时，可以关闭多余属性检测。

8. 最小可选属性规则

根据"结构类型"原则，如果一个对象的所有属性都是可选的，那么其他对象跟它都是结构类似的。

如果一个对象的属性都是可选的，那它就可以是一个空对象，也意味着任意对象都能满足这个结构，为了避免这种情况的发生，ts引入了**最小可选属性规则，也称为弱类型检测。**

解决方法：

- 在类型中新增一条索引属性[propName: string]: someType
- 使用类型断言opt as type

9. 空对象

空对象在ts中是一种特殊值，也是一种特殊类型。

ts的空对象没有自定义属性，只能使用继承的属性，即继承自原型对象Object.prototype的属性。

在ts中对于对象必须一次性声明所有属性。

如果确实需要分步声明，比较好的方法就是使用扩展运算符合成一个新对象

```ts
const pt0 = {};
const pt1 = { x: 3 };
const pt2 = { y: 4 };

const pt = {
  ...pt0, ...pt1, ...pt2
};
```

空对象作为类型，其实是`Object`类型的简写形式，跟Object类型的行为是一样的。

什么除了null和undefined以外其他类型的值都能赋值给Object，所以它不会有严格的字面量检查。

```
let d:{};
// 等同于
// let d:Object;

d = {};
d = { x: 1 };
d = 'hello';
d = 2;
```

如果想强制使用没有任何属性的对象，可以采用下面的写法。

```
interface WithoutProperties {
  [key: string]: never;
}

// 报错
const a:WithoutProperties = { prop: 1 };

```

上面的示例中，`[key: string]: never`表示属性值的number类型不能赋值给never类型，因此其他对象进行赋值时就会报错。