

对象类型

除了原始类型，对象是 JavaScript 最基本的数据结构。TypeScript 对于对象类型有很多规则。

对象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。

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。

```
type MyObj = {  
  x:number;  
  y:number;  
};  
const o1:MyObj = { x: 1 }; // 报错  
const o2:MyObj = { x: 1, y: 1, z: 1 }; // 报错
```

上例中，变量 `o1` 缺少了属性 `y`，变量 `o2` 多出了属性 `z`，都会报错。

读写不存在的属性也会报错。

```
const obj:{  
  x:number;  
  y:number;  
} = { x: 1, y: 1 };  
console.log(obj.z); // Property 'z' does not exist on type '{ x: number; y: number; }'.  
obj.z = 1; // Property 'z' does not exist on type '{ x: number; y: number; }'.
```

同样地，也不能删除类型声明中存在的属性，修改属性值是可以的。

```
const myUser = {  
  name: "Sabrina",  
};  
myUser.name = "Cynthia"; // 正确  
delete myUser.name // The operand of a 'delete' operator must be optional.  
'delete'操作符的操作数必须是可选的。
```

上面声明中，删除类型声明中存在的属性`name`会报错，但是可以修改它的值。

对象的方法使用函数类型描述。

```
const obj:{  
  x: number;  
  y: number;  
  add(x:number, y:number): number;
```

```
// 或者写成
// add: (x:number, y:number) => number;
} = {
  x: 1,
  y: 1,
  add(x, y) {
    return x + y;
  }
};
```

对象类型可以使用方括号读取属性的类型。

```
type User = {
  name: string,
  age: number
};
type Name = User['name']; // type Name = string
type Age = User.age; // User 是类型不是命名空间
```

除了 `type` 命令可以为对象类型声明一个别名，TypeScript 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 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 };
```

TypeScript 不区分对象自身的属性和继承的属性，一律视为对象的属性。

```
interface MyInterface {
  toString(): string; // 继承的属性
  prop: number; // 自身的属性
}
const obj:MyInterface = { // 正确
  prop: 123,
};
// 或者
const obj:MyInterface = { // 正确
```

```
    toString: () => '122',  
    prop: 123,  
};
```

1. 可选属性

如果某个属性是可选的（即可以忽略），需要在属性名后面加一个问号。

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

可选属性等同于允许赋值为 `undefined`，下面两种写法是等效的。

```
type User = {  
  firstName: string;  
  lastName?: string;  
}; // type User = { firstName: string; lastName?: string | undefined; }  
  
// 等价于  
type User = {  
  firstName: string;  
  lastName?: string|undefined;  
}; // type User = { firstName: string; lastName?: string | undefined; }
```

上例中，类型 `User` 的可选属性 `lastName` 可以是字符串，也可以是 `undefined`，即可选属性可以赋值为 `undefined`。

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

上例中，可选属性 `y` 赋值为 `undefined`，不会报错。

同样地，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() // 'obj.y' is possibly 'undefined'.
```

所以，读取可选属性之前，必须检查一下是否为undefined。

```
const user:{
  firstName: string;
  lastName?: string;
} = { firstName: 'Foo' };
if (user.lastName !== undefined) {
  console.log(`hello ${user.firstName} ${user.lastName}`)
}
```

上例中，`lastName` 是可选属性，需要判断是否为 `undefined` 以后，才能使用。

```
// 写法一
let firstName = (user.firstName === undefined) ? 'Foo' : user.firstName;
let lastName = (user.lastName === undefined) ? 'Bar' : user.lastName;

// 写法二
let firstName = user.firstName ?? 'Foo';
let lastName = user.lastName ?? 'Bar';
```

可选属性与允许设为 `undefined` 的必选属性是不等价的。 可选属性可以设置该属性，也可以不设置。但允许设置为 `undefined` 的属性必须设置，只是可以设置为 `undefined`。

```
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 }; // Property 'y' is missing in type '{ x: number; }' but
required in type 'B'.
```

2. 只读属性

属性名前面加上 `readonly` 关键字，表示这个属性是只读属性，不能修改。

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

上例中，`prop` 属性是只读属性，不能修改它的值。

```
const person:{
  readonly age: number
} = { age: 20 };
```

```
person.age = 21; // Cannot assign to 'age' because it is a read-only property. 不能赋值给'age'，因为它是一个只读属性。
```

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。

```
type Point = {  
  readonly x: number;  
  readonly y: number;  
};  
const p:Point = { x: 0, y: 0 };  
p.x = 100; // Cannot assign to 'x' because it is a read-only property.
```

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。

```
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  
} // Cannot assign to 'resident' because it is a read-only property.
```

上例中，`h.resident` 是只读属性，它的值是一个对象。修改这个对象的 `age` 属性是可以的，但是整个替换掉 `h.resident` 属性会报错。

另一个需要注意的地方是，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。

```
interface Person {  
  name: string;  
  age: number;  
}  
interface ReadonlyPerson {  
  readonly name: string;  
  readonly age: number;  
}  
let w:Person = {
```

```

    name: 'Vicky',
    age: 42,
  };
  let r: ReadonlyPerson = w;
  w.age += 1;
  r.age // 43
  r.age = 45; // Cannot assign to 'age' because it is a read-only property.

```

上例中，变量 `w` 和 `r` 指向同一个对象，其中 `w` 是可写的，`r` 是只读的。那么，对 `w` 的属性修改，会影响到 `r`。

如果希望属性值是只读的，除了声明时加上 `readonly` 关键字，还有一种方法，就是在赋值时，在对象后面加上只读断言 `as const`。

```

const myUser = {
  name: "Sabrina",
} as const;
myUser.name = "Cynthia"; // Cannot assign to 'name' because it is a read-only property.

```

上例中，对象后面加了只读断言 `as const`，就变成只读对象了，不能修改属性了。

上面的 `as const` 属于 TypeScript 的类型推断，如果变量明确地声明了类型，那么 TypeScript 会以声明的类型为准。

```

const myUser: { name: string } = {
  name: "Sabrina",
} as const;
myUser.name = "Cynthia"; // 正确

const myUser2: { readonly name: string } = {
  name: "Sabrina",
};
myUser2.name = "Cynthia"; // Cannot assign to 'name' because it is a read-only property.

```

上例中，根据变量 `myUser` 的类型声明，`name` 不是只读属性，但是赋值时又使用只读断言 `as const`。这时会以声明的类型为准，因为 `name` 属性可以修改。如果类型声明时明确了属性是 `readonly` 的便不可修改。

3. 属性名的索引类型

如果对象的属性非常多，一个个声明类型就很麻烦，而且有些时候，无法事前知道对象会有多少属性，比如外部 API 返回的对象。这时 TypeScript 允许采用属性名表达式的写法来描述类型，称为“属性名的索引类型”。

索引类型里面，最常见的就是属性名的字符串索引。

```

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

```

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

上例中，类型 `MyObj` 的属性名类型就采用了表达式形式，写在方括号里面。`[property: string]` 的 `property` 表示属性名，这个是可以随便起的，它的类型是 `string`，即属性名类型为 `string`。也就是说，**不管这个对象有多少属性，只要属性名为字符串，且属性值也是字符串，就符合这个类型声明。**

JavaScript 对象的属性名（即上例的 `property`）的类型有三种可能，除了上例的 `string`，还有 `number` 和 `symbol`。

```
type T1 = {  
  [property: number]: string  
};  
type T2 = {  
  [property: symbol]: string  
};
```

上例中，对象属性名的类型分别为 `number` 和 `symbol`。

```
type MyArr = {  
  [n:number]: number;  
};  
const arr:MyArr = [1, 2, 3];  
// 或者  
const arr:MyArr = {  
  0: 1,  
  1: 2,  
  2: 3,  
};
```

上例中，对象类型 `MyArr` 的属性名是 `[n:number]`，就表示它的属性名都是数值，比如 0、1、2。

对象可以同时有多种类型的属性名索引，比如同时有数值索引和字符串索引。但是，数值索引不能与字符串索引发生冲突，必须服从后者，这是因为在 JavaScript 语言内部，所有的数值属性名都会自动转为字符串属性名。

```
type MyType = {  
  [x: number]: boolean; // 'number' index type 'boolean' is not assignable to  
  'string' index type 'string'. 'number'索引类型'boolean'不能赋值给'string'索引类型'string'.  
  [x: string]: string;  
}
```

上例中，类型 `MyType` 同时有两种属性名索引，但是数值索引与字符串索引冲突了，所以报错了。由于字符串属性名的值类型是 `string`，数值属性名的值类型只有同样为 `string`，才不会报错。

同样地，可以既声明属性名索引，也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，两者不能有冲突，否则报错。

```
type MyType = {
  foo: boolean; // Property 'foo' of type 'boolean' is not assignable to 'string'
  index type 'string'.
  [x: string]: string;
}
```

上例中，属性名 `foo` 符合属性名的字符串索引，但是两者的属性值类型不一样，所以报错了。

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。**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 // Property 'length' does not exist on type 'MyArr'.
// 改成下面这样就正确
type MyArr = {
  [n: string]: number;
};
const arr: MyArr = { 0: 1, 1: 2, 2: 3, length: 3 };
arr.length;
```

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

4. 解构赋值

解构赋值用于直接从对象中提取属性。

`const {id, name, price} = product;` 语句从对象 `product` 提取了三个属性，并声明属性名的同名变量。

解构赋值的类型写法，跟为对象声明类型是一样的。

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


目前没法为解构变量指定类型，因为对象解构里面的冒号，JavaScript 指定了其他用途。

```
let { x: foo, y: bar } = obj;  
// 等同于  
let foo = obj.x;  
let bar = obj.y;
```

上例中，冒号不是表示属性 `x` 和 `y` 的类型，而是为这两个属性指定新的变量名。如果要为 `x` 和 `y` 指定类型，不得不写成下面这样。

```
let { x: foo, y: bar } : { x: string; y: number } = obj;
```

这一点要特别小心，TypeScript 里面很容易搞糊涂。

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

上例中，函数 `draw()` 的参数是一个对象解构，里面的冒号很像是为变量指定类型，其实是为对应的属性指定新的变量名。所以，TypeScript 就会解读成，函数体内不存在变量 `shape`，而是属性 `shape` 的值被赋值给了变量 `Shape`。

5. 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)。

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

上面示例中，对象 `A` 只有一个属性 `x`，类型为 `number`。对象 `B` 满足这个特征，因此兼容对象 `A`，只要可以使用 `A` 的地方，就可以使用 `B`。

```
type A = {  
  x: number;  
};  
const B = {  
  x: 1,  
  y: 1  
};  
const a:A = B; // 正确
```

根据“结构类型”原则，TypeScript 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”）。

TypeScript 之所以这样设计，是为了符合 JavaScript 的行为。JavaScript 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**，TypeScript 就认为 **B** 是 **A** 的子类型（**subtype**），**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。**

这种设计有时会导致令人惊讶的结果。

```
type myObj = { x: number, y: number, };  
  
function getSum(obj:myObj) { let sum = 0;  
  
for (const n of Object.keys(obj)) { const v = obj[n]; // 报错 sum += Math.abs(v); }  
  
return sum; }
```

上面示例中，函数getSum()要求传入参数的类型是myObj，但是实际上所有与myObj兼容的对象都可以传入。这会导致const v = obj[n]这一行报错，原因是obj[n]取出的属性值不一定是数值（number），使得变量v的类型被推断为any。如果项目设置为不允许变量类型推断为any，代码就会报错。写成下面这样，就不会报错。

```
type MyObj = { x: number, y: number, };  
  
function getSum(obj:MyObj) { return Math.abs(obj.x) + Math.abs(obj.y); }
```

上面示例就不会报错，因为函数体内部只使用了属性x和y，这两个属性有明确的类型声明，保证obj.x和obj.y肯定是数值。虽然与MyObj兼容的任何对象都可以传入函数getSum()，但是只要不使用其他属性，就不会有类型报错。严格字面量检查

如果对象使用字面量表示，会触发 TypeScript 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。

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

上面示例中，等号右边是一个对象的字面量，这时会触发严格字面量检查。只要有类型声明中不存在的属性（本例是z），就会导致报错。

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，是不会报错的。

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

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

上面示例中，等号右边是一个变量，就不会触发严格字面量检查，从而不报错。

TypeScript 对字面量进行严格检查的目的，主要是防止拼写错误。一般来说，字面量大多数来自手写，容易出现拼写错误，或者误用 API。

```
type Options = { title:string; darkMode?:boolean; };
```

```
const obj:Options = { title: '我的网页', darkmode: true, // 报错 };
```

上面示例中，属性darkMode拼写错了，成了darkmode。如果没有严格字面量规则，就不会报错，因为darkMode是可选属性，根据结构类型原则，任何对象只要有title属性，都认为符合Options类型。

规避严格字面量检查，可以使用中间变量。

```
let myOptions = { title: '我的网页', darkmode: true, };
```

```
const obj:Options = myOptions;
```

上面示例中，创建了一个中间变量myOptions，就不会触发严格字面量规则，因为这时变量obj的赋值，不属于直接字面量赋值。

如果你确认字面量没有错误，也可以使用类型断言规避严格字面量检查。

```
const obj:Options = { title: '我的网页', darkmode: true, } as Options;
```

上面示例使用类型断言as Options，告诉编译器，字面量符合 Options 类型，就能规避这条规则。

如果允许字面量有多余属性，可以像下面这样在类型里面定义一个通用属性。

```
let x: { foo: number, [x: string]: any };
```

```
x = { foo: 1, baz: 2 }; // Ok
```

上面示例中，变量x的类型声明里面，有一个属性的字符串索引（[x: string]），导致任何字符串属性名都是合法的。

由于严格字面量检查，字面量对象传入函数必须很小心，不能有多余的属性。

```
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
```

上面示例中，对象字面量传入函数computeDistance()时，不能有多余的属性，否则就通不过严格字面量检查。

编译器选项suppressExcessPropertyErrors，可以关闭多余属性检查。下面是它在 tsconfig.json 文件里面的写法。

```
{ "compilerOptions": { "suppressExcessPropertyErrors": true } }
```

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。

```
type Options = { a?:number; b?:number; c?:number; };
```

上面示例中，类型Options的所有属性都是可选的，所以它可以是一个空对象，也就意味着任意对象都满足Options的结构。

为了避免这种情况，TypeScript 2.4 引入了一个“最小可选属性规则”，也称为“弱类型检测”（weak type detection）。

```
type Options = { a?:number; b?:number; c?:number; };
```

```
const opts = { d: 123 };
```

```
const obj:Options = opts; // 报错
```

上面示例中，对象opts与类型Options没有共同属性，赋值给该类型的变量就会报错。

报错原因是，如果某个类型的所有属性都是可选的，那么该类型的对象必须至少存在一个可选属性，不能所有可选属性都不存在。这就叫做“最小可选属性规则”。

如果想规避这条规则，要么在类型里面增加一条索引属性（[propName: string]: someType），要么使用类型断言（opts as Options）。空对象

空对象是 TypeScript 的一种特殊值，也是一种特殊类型。

```
const obj = {}; obj.prop = 123; // 报错
```

上面示例中，变量obj的值是一个空对象，然后对obj.prop赋值就会报错。

原因是这时 TypeScript 会推断变量obj的类型为空对象，实际执行的是下面的代码。

```
const obj:{} = {};
```

空对象没有自定义属性，所以对自定义属性赋值就会报错。空对象只能使用继承的属性，即继承自原型对象Object.prototype的属性。

```
obj.toString() // 正确
```

上面示例中，toString()方法是一个继承自原型对象的方法，TypeScript 允许在空对象上使用。

回到本节开始的例子，这种写法其实在 JavaScript 很常见：先声明一个空对象，然后向空对象添加属性。但是，TypeScript 不允许动态添加属性，所以对象不能分步生成，必须生成时一次性声明所有属性。

```
// 错误 const pt = {}; pt.x = 3; pt.y = 4;
```

```
// 正确 const pt = { x: 3, y: 4 };
```

如果确实需要分步声明，一个比较好的方法是，使用扩展运算符（...）合成一个新对象。

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

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

上面示例中，对象pt是三个部分合成的，这样既可以分步声明，也符合 TypeScript 静态声明的要求。

空对象作为类型，其实是Object类型的简写形式。

```
let d:{}; // 等同于 // let d:Object;
```

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

上面示例中，各种类型的值（除了null和undefined）都可以赋值给空对象类型，跟Object类型的行为是一样的。

因为Object可以接受各种类型的值，而空对象是Object类型的简写，所以它不会有严格字面量检查，赋值时总是允许多余的属性，只是不能读取这些属性。

```
interface Empty { } const b:Empty = {myProp: 1, anotherProp: 2}; // 正确 b.myProp // 报错
```

上面示例中，变量b的类型是空对象，视同Object类型，不会有严格字面量检查，但是读取多余的属性会报错。

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表示字符串属性名是不存在的，因此其他对象进行赋值时就会报错。