

# TypeScript 的对象类型

## 简介

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。

对象类型的最简单声明方法，就是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 };
```

typescript

上面示例中，对象 `obj` 的类型就写在变量名后面，使用大括号描述，内部声明每个属性的属性名和类型。

属性的类型可以用分号结尾，也可以用逗号结尾。

```
// 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;  
};
```

typescript

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。

typescript

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

上面示例中，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。

typescript

```
const obj: {  
  x: number;  
  y: number;  
} = { x: 1, y: 1 };  
  
console.log(obj.z); // 报错  
obj.z = 1; // 报错
```

上面示例中，读写不存在的属性 `z` 都会报错。

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。

typescript

```
const myUser = {  
  name: "Sabrina",  
};  
  
delete myUser.name; // 报错  
myUser.name = "Cynthia"; // 正确
```

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;
  },
};
```

上面示例中，对象 `obj` 有一个方法 `add()`，需要定义它的参数类型和返回值类型。

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"]; // string
```

上面示例中，对象类型 `User` 使用方括号，读取了属性 `name` 的类型（`string`）。

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 };
```

上面示例中，写法一是 `type` 命令的用法，写法二是 `interface` 命令的用法。`interface` 命令的详细解释，以及与 `type` 命令的区别，详见《Interface》一章。

注意，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,
};
```

typescript

上面示例中，`obj` 只写了 `prop` 属性，但是不报错。因为它可以继承原型上面的 `toString()` 方法。

---

## 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 };
```

typescript

上面示例中，属性 `y` 是可选的。

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

上面示例中，类型 `User` 的属性 `lastName` 可以是字符串，也可以是 `undefined`，就表示该属性可以省略不写。

同理，读取一个可选属性时，有可能返回 `undefined`。

```
type MyObj = {  
  x: string;  
  y?: string;  
};  
  
const obj: MyObj = { x: "hello" };  
obj.y.toLowerCase(); // 报错
```

上面示例中，最后一行会报错，因为 `obj.y` 有可能是 `undefined`，无法对其调用 `toLowerCase()`。

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}`);  
}
```

上面示例中，`lastName` 是可选属性，需要判断是否为 `undefined` 以后，才能使用。建议使用下面的写法。

typescript

```
// 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";
```

上面示例中，写法一使用三元运算符 `?:`，判断是否为 `undefined`，并设置默认值。写法二使用 `Null` 判断运算符 `??`，与写法一的作用完全相同。

---

## 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。

typescript

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

上面示例中，`prop` 属性是只读属性，不能修改它的值。

typescript

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

person.age = 21; // 报错
```

上面示例中，最后一行修改了只读属性 `age`，就报错了。

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。

typescript

```
type Point = {
  readonly x: number;
```

```
    readonly y: number;
};

const p: Point = { x: 0, y: 0 };

p.x = 100; // 报错
```

上面示例中，类型 `Point` 的属性 `x` 和 `y` 都带有修饰符 `readonly`，表示这两个属性只能在初始化期间赋值，后面再修改就会报错。

注意，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。

typescript

```
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
```

上面示例中，`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

```

上面示例中，变量 `w` 和 `r` 指向同一个对象，其中 `w` 是可写的，`r` 的只读的。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"; // 报错

```

上面示例中，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。

注意，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
```

上面示例中，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。

---

## 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",  
};
```

typescript

上面示例中，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 = {
```

typescript

```
[property: symbol]: string;
};
```

上面示例中，对象属性名的类型分别为 `number` 和 `symbol`。

typescript

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

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

上面示例中，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。

typescript

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

上面示例中，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。

typescript

```
type MyType = {
  foo: boolean; // 报错
```

```
[x: string]: string;
};
```

上面示例中，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。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
```

typescript

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。

---

## 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;
```

typescript

上面语句从对象 `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;
```

typescript

```
    price: number;
} = product;
```

注意，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;
```

typescript

上面示例中，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;
```

typescript

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
}
```

typescript

上面示例中，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`。

---

## 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）。

```
const A = {
    x: number;
```

typescript

```
};
```

```
const 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。

typescript

```
const B = {  
  x: 1,  
  y: 1,  
};
```

```
const A: { x: number } = B; // 正确
```

上面示例中，A 和 B 并不是同一个类型，但是 B 可以赋值给 A，因为 B 满足 A 的结构特征。

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（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。

这种设计有时会导致令人惊讶的结果。

typescript

```
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。

typescript

```
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。

typescript

```
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。

typescript

```
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。

typescript

```
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  
  }  
}
```

typescript

---

## 最小可选属性规则

如果一个对象的所有属性都是可选的，会触发最小可选属性规则。

```
type Options = {  
  a?: number;  
  b?: number;  
  c?: number;  
};  
  
const obj: Options = {  
  d: 123, // 报错  
};
```

typescript

上面示例中，类型 `Options` 是一个对象，它的所有属性都是可选的，这导致任何对象实际都符合 `Options` 类型。

为了避免这种情况，TypeScript 添加了最小可选属性规则，规定这时属于 `Options` 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。这就是为什么上例的 `myObj` 对象会报错的原因。

这条规则无法通过中间变量规避。

typescript

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

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

上面示例中，即使使用了中间变量 `myOptions`，由于存在最小可选属性规则，依然会报错。

---

## 空对象

空对象是 TypeScript 的一种特殊值，也是一种特殊类型。

typescript

```
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。

typescript

```
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。

typescript

```
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。

typescript

```
// 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。

typescript

```
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。

typescript

```
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。

typescript

```
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。

typescript

```
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。

### 限时抢

推荐机场 → [25元/月, 500G](#) 购买。

最后更新: 2023/8/13 15:25

Previous page  
[函数](#)

Next page  
[interface](#)

