TypeScript 类型运算符

目录

- keyof 运算符
  - 简介
  - keyof 运算符的用途
- in 运算符
- 方括号运算符
- extends...?: 条件运算符
- infer 关键字
- is 运算符
- 模板字符串
- satisfies 运算符

TypeScript 提供强大的类型运算能力，可以使用各种类型运算符，对已有的类型进行计算，得到新类型。

keyof 运算符

简介

keyof 是一个单目运算符，接受一个对象类型作为参数，返回该对象的所有键名组成的联合类型。

```
type MyObj = {
  foo: number,
  bar: string,
};

type Keys = keyof MyObj; // 'foo'|'bar'
```

上面示例中，`keyof MyObj` 返回 `MyObj` 的所有键名组成的联合类型，即 `'foo'|'bar'`。

下面是另一个例子。

```
interface T {
  0: boolean;
  a: string;
  b(): void;
}

type KeyT = keyof T; // 0 | 'a' | 'b'
```

由于 JavaScript 对象的键名只有三种类型，所以对于任意对象的键名的联合类型就是 `string|number|symbol`。

```
// string | number | symbol
type KeyT = keyof any;
```

对于没有自定义键名的类型使用 keyof 运算符，返回 `never` 类型，表示不可能有这样类型的键名。

```
type KeyT = keyof object;  // never
```

上面示例中，由于 `object` 类型没有自身的属性，也就没有键名，所以 `keyof object` 返回 `never` 类型。

由于 keyof 返回的类型是 `string|number|symbol`，如果有些场合只需要其中的一种类型，那么可以采用交叉类型的写法。

```
type Capital<T extends string> = Capitalize<T>;

type MyKeys<Obj extends object> = Capital<keyof Obj>; // 报错
```

上面示例中，类型 `Capital` 只接受字符串作为类型参数，传入 `keyof Obj` 会报错，原因是这时的类型参数是 `string|number|symbol`，跟字符串不兼容。采用下面的交叉类型写法，就不会报错。

```
type MyKeys<Obj extends object> = Capital<string & keyof Obj>;
```

上面示例中，`string & keyof Obj` 等同于 `string & string|number|symbol` 进行交集运算，最后返回 `string`，因此 `Capital<T extends string>` 就不会报错了。

如果对象属性名采用索引形式，keyof 会返回属性名的索引类型。

```
// 示例一
interface T {
  [prop: number]: number;
}

// number
```

```
type KeyT = keyof T;

// 示例二
interface T {
  [prop: string]: number;
}

// string|number
type KeyT = keyof T;
```

上面的示例二，`keyof T` 返回的类型是 `string|number`，原因是 JavaScript 属性名为字符串时，包含了属性名为数值的情况，因为数值属性名会自动转为字符串。

如果 keyof 运算符用于数组或元组类型，得到的结果可能出人意料。

```
type Result = keyof ['a', 'b', 'c'];
// 返回 number | "0" | "1" | "2"
// | "length" | "pop" | "push" | ···
```

上面示例中，keyof 会返回数组的所有键名，包括数字键名和继承的键名。

对于联合类型，keyof 返回成员共有的键名。

```
type A = { a: string; z: boolean };
type B = { b: string; z: boolean };

// 返回 'z'
type KeyT = keyof (A | B);
```

对于交叉类型，keyof 返回所有键名。

```
type A = { a: string; x: boolean };
type B = { b: string; y: number };

// 返回 'a' | 'x' | 'b' | 'y'
type KeyT = keyof (A & B);

// 相当于
keyof (A & B) ≡ keyof A | keyof B
```

keyof 取出的是键名组成的联合类型，如果想取出键值组成的联合类型，可以像下面这样写。

```
type MyObj = {
  foo: number,
  bar: string,
};

type Keys = keyof MyObj;

type Values = MyObj[Keys]; // number|string
```

上面示例中，`Keys`是键名组成的联合类型，而`MyObj[Keys]`会取出每个键名对应的键值类型，组成一个新的联合类型，即`number|string`。

keyof 运算符的用途

keyof 运算符往往用于精确表达对象的属性类型。

举例来说，取出对象的某个指定属性的值，JavaScript 版本可以写成下面这样。

```
function prop(obj, key) {
  return obj[key];
}
```

上面这个函数添加类型，只能写成下面这样。

```
function prop(
  obj: { [p:string]: any },
  key: string
):any {
  return obj[key];
}
```

上面的类型声明有两个问题，一是无法表示参数`key`与参数`obj`之间的关系，二是返回值类型只能写成`any`。

有了 keyof 以后，就可以解决这两个问题，精确表达返回值类型。

```
function prop<Obj, K extends keyof Obj>(
  obj:Obj, key:K
):Obj[K] {
  return obj[key];
}
```

上面示例中，`K extends keyof Obj`表示`K`是`Obj`的一个属性名，传入其他字符串会报错。返回值类型`Obj[K]`就表示`K`这个属性值的类型。

keyof 的另一个用途是用于属性映射，即将一个类型的所有属性逐一映射成其他值。

```
type NewProps<Obj> = {
  [Prop in keyof Obj]: boolean;
};

// 用法
type MyObj = { foo: number; };

// 等于 { foo: boolean; }
type NewObj = NewProps<MyObj>;
```

上面示例中，类型`NewProps`是类型`Obj`的映射类型，前者继承了后者的所有属性，但是把所有属性值类型都改成了`boolean`。

下面的例子是去掉 readonly 修饰符。

```
type Mutable<Obj> = {
  -readonly [Prop in keyof Obj]: Obj[Prop];
};

// 用法
type MyObj = {
  readonly foo: number;
}

// 等于 { foo: number; }
type NewObj = Mutable<MyObj>;
```

上面示例中，`[Prop in keyof Obj]`是`Obj`类型的所有属性名，`-readonly`表示去除这些属性的只读特性。对应地，还有`+readonly`的写法，表示添加只读属性设置。

下面的例子是让可选属性变成必有的属性。

```
type Concrete<Obj> = {
  [Prop in keyof Obj]-?: Obj[Prop];
};

// 用法
```

```
type MyObj = {
  foo?: number;
}

// 等于 { foo: number; }
type NewObj = Concrete<MyObj>;
```

上面示例中，`[Prop in keyof Obj]`后面的`-?`表示去除可选属性设置。对应地，还有`+?`的写法，表示添加可选属性设置。

in 运算符

JavaScript 语言中，`in`运算符用来确定对象是否包含某个属性名。

```
const obj = { a: 123 };

if ('a' in obj)
  console.log('found a');
```

上面示例中，`in`运算符用来判断对象`obj`是否包含属性`a`。

`in`运算符的左侧是一个字符串，表示属性名，右侧是一个对象。它的返回值是一个布尔值。

TypeScript 语言的类型运算中，`in`运算符有不同的用法，用来取出（遍历）联合类型的每一个成员类型。

```
type U = 'a'|'b'|'c';

type Foo = {
  [Prop in U]: number;
};
// 等同于
type Foo = {
  a: number,
  b: number,
  c: number
};
```

上面示例中，`[Prop in U]`表示依次取出联合类型`U`的每一个成员。

上一小节的例子也提到，`[Prop in keyof Obj]`表示取出对象`Obj`的每一个键名。

方括号运算符

方括号运算符（`[]`）用于取出对象的键值类型，比如`T[K]`会返回对象`T`的属性`K`的类型。

```
type Person = {
  age: number;
  name: string;
  alive: boolean;
};

// Age 的类型是 number
type Age = Person['age'];
```

上面示例中，`Person['age']`返回属性`age`的类型，本例是`number`。

方括号的参数如果是联合类型，那么返回的也是联合类型。

```
type Person = {
  age: number;
  name: string;
  alive: boolean;
};

// number|string
type T = Person['age'|'name'];

// number|string|boolean
type A = Person[keyof Person];
```

上面示例中，方括号里面是属性名的联合类型，所以返回的也是对应的属性值的联合类型。

如果访问不存在的属性，会报错。

```
type T = Person['notExisted']; // 报错
```

方括号运算符的参数也可以是属性名的索引类型。

```
type Obj = {
  [key:string]: number,
};
```

```
// number
type T = Obj[string];
```

上面示例中，`Obj`的属性名是字符串的索引类型，所以可以写成`Obj[string]`，代表所有字符串属性名，返回的就是它们的类型`number`。

这个语法对于数组也适用，可以使用`number`作为方括号的参数。

```
// MyArray 的类型是 { [key:number]: string }
const MyArray = ['a','b','c'];

// 等同于 (typeof MyArray)[number]
// 返回 string
type Person = typeof MyArray[number];
```

上面示例中，`MyArray`是一个数组，它的类型实际上是属性名的数值索引，而`typeof MyArray[number]`的`typeof`运算优先级高于方括号，所以返回的是所有数值键名的键值类型`string`。

注意，方括号里面不能有值的运算。

```
// 示例一
const key = 'age';
type Age = Person[key]; // 报错

// 示例二
type Age = Person['a' + 'g' + 'e']; // 报错
```

上面两个示例，方括号里面都涉及值的运算，编译时不会进行这种运算，所以会报错。

extends...?: 条件运算符

TypeScript 提供类似 JavaScript 的`?:`运算符这样的三元运算符，但多出了一个`extends`关键字。

条件运算符`extends...?:`可以根据当前类型是否符合某种条件，返回不同的类型。

```
T extends U ? X : Y
```

上面式子中的`extends`用来判断，类型`T`是否可以赋值给类型`U`，即`T`是否为`U`的子类型，这里的`T`和`U`可以是任意类型。

如果 `T` 能够赋值给类型 `U`，表达式的结果为类型 `X`，否则结果为类型 `Y`。

```
// true
type T = 1 extends number ? true : false;
```

上面示例中，`1` 是 `number` 的子类型，所以返回 `true`。

下面是另外一个例子。

```
interface Animal {
  live(): void;
}
interface Dog extends Animal {
  woof(): void;
}

// number
type T1 = Dog extends Animal ? number : string;

// string
type T2 = RegExp extends Animal ? number : string;
```

上面示例中，`Dog` 是 `Animal` 的子类型，所以 `T1` 的类型是 `number`。`RegExp` 不是 `Animal` 的子类型，所以 `T2` 的类型是 `string`。

一般来说，调换 `extends` 两侧类型，会返回相反的结果。举例来说，有两个类 `Cat` 和 `Animal`，前者是后者的子类型，那么 `Cat extends Animal` 就为真，而 `Animal extends Cat` 就为伪。

如果对泛型使用 extends 条件运算，有一个地方需要注意。当泛型的类型参数是一个联合类型时，那么条件运算符会展开这个类型参数，即 `T<A|B> = T<A> | T<B>`，所以 extends 对类型参数的每个部分是分别计算的。

```
type Cond<T> = T extends U ? X : Y;

type MyType = Cond<A|B>;
// 等同于 Cond<A> | Cond<B>
// 等同于 (A extends U ? X : Y) | (B extends U ? X : Y)
```

上面示例中，泛型 `Cond` 的类型参数 `A|B` 是一个联合类型，进行条件运算时，相当于 `A` 和 `B` 分别进行条件运算，返回结果组成一个联合类型。也就是说，如果类型参数是联合类型，条件运算的返回结果依然是一个联合类型。

如果不希望联合类型被条件运算符展开，可以把 `extends` 两侧的操作数都放在方括号里面。

```
// 示例一
type ToArray<Type> =
  Type extends any ? Type[] : never;

// 返回结果 string[]|number[]
type T = ToArray<string|number>;

// 示例二
type ToArray<Type> =
  [Type] extends [any] ? Type[] : never;

// 返回结果 (string | number)[]
type T = ToArray<string|number>;
```

上面的示例一，泛型 `ToArray<Type>` 的类型参数 `string|number` 是一个联合类型，所以会被展开，返回的也是联合类型 `string[]|number[]`。示例二是 `extends` 两侧的运算数都放在方括号里面，左侧是 `[Type]`，右侧是 `[any]`，这时传入的联合类型不会展开，返回的是一个数组 `(string|number)[]`。

条件运算符还可以嵌套使用。

```
type LiteralTypeName<T> =
  T extends undefined ? "undefined" :
  T extends null ? "null" :
  T extends boolean ? "boolean" :
  T extends number ? "number" :
  T extends bigint ? "bigint" :
  T extends string ? "string" :
  never;
```

上面示例是一个多重判断，返回一个字符串的值类型，对应当前类型。下面是它的用法。

```
// "bigint"
type Result1 = LiteralTypeName<123n>;

// "string" | "number" | "boolean"
type Result2 = LiteralTypeName<true | 1 | 'a'>;
```

infer 关键字

`infer`关键字用来定义泛型里面推断出来的类型参数，而不是外部传入的类型参数。

它通常跟条件运算符一起使用，用在`extends`关键字后面的父类型之中。

```
type Flatten<Type> =
  Type extends Array<infer Item> ? Item : Type;
```

上面示例中，`infer Item`表示`Item`这个参数是 TypeScript 自己推断出来的，不用显式传入，而`Flatten<Type>`则表示`Type`这个类型参数是外部传入的。`Type extends Array<infer Item>`则表示，如果参数`Type`是一个数组，那么就将该数组的成员类型推断为`Item`，即`Item`是从`Type`推断出来的。

一旦使用`infer Item`定义了`Item`，后面的代码就可以直接调用`Item`了。下面是上例的泛型`Flatten<Type>`的用法。

```
// string
type Str = Flatten<string[]>;

// number
type Num = Flatten<number>;
```

上面示例中，第一个例子`Flatten<string[]>`传入的类型参数是`string[]`，可以推断出`Item`的类型是`string`，所以返回的是`string`。第二个例子`Flatten<number>`传入的类型参数是`number`，它不是数组，所以直接返回自身。

如果不用`infer`定义类型参数，那么就要传入两个类型参数。

```
type Flatten<Type, Item> =
  Type extends Array<Item> ? Item : Type;
```

上面是不使用`infer`的写法，每次调用`Flatten`的时候，都要传入两个参数，就比较麻烦。

下面的例子使用`infer`，推断函数的参数类型和返回值类型。

```
type ReturnPromise<T> =
  T extends (...args: infer A) => infer R
  ? (...args: A) => Promise<R>
  : T;
```

上面示例中，如果`T`是函数，就返回这个函数的 Promise 版本，否则原样返回。`infer A`表示该函数的参数类型为`A`，`infer R`表示该函数的返回值类型为`R`。

如果不使用`infer`，就不得不把`ReturnPromise<T>`写成`ReturnPromise<T, A, R>`，这样就很麻烦，相当于开发者必须人肉推断编译器可以完成的工作。

下面是 `infer` 提取对象指定属性的例子。

```
type MyType<T> =
  T extends {
    a: infer M,
    b: infer N
  } ? [M, N] : never;

// 用法示例
type T = MyType<{ a: string; b: number }>;
// [string, number]
```

上面示例中，`infer` 提取了参数对象的属性 `a` 和属性 `b` 的类型。

下面是 `infer` 通过正则匹配提取类型参数的例子。

```
type Str = 'foo-bar';

type Bar = Str extends `foo-${infer rest}` ? rest : never // 'bar'
```

上面示例中，`rest` 是从模板字符串提取的类型参数。

is 运算符

函数返回布尔值的时候，可以使用 `is` 运算符，限定返回值与参数之间的关系。

`is` 运算符用来描述返回值属于 `true` 还是 `false`。

```
function isFish(
  pet: Fish|Bird
):pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

上面示例中，函数 `isFish()` 的返回值类型为 `pet is Fish`，表示如果参数 `pet` 类型为 `Fish`，则返回 `true`，否则返回 `false`。

`is` 运算符总是用于描述函数的返回值类型，写法采用 `parameterName is Type` 的形式，即左侧为当前函数的参数名，右侧为某一种类型。它返回一个布尔值，表示左侧参数是否属于右侧的类型。

```
type A = { a: string };
type B = { b: string };

function isTypeA(x: A|B): x is A {
  if ('a' in x) return true;
  return false;
}
```

上面示例中，返回值类型 `x is A` 可以准确描述函数体内部的运算逻辑。

`is` 运算符可以用于类型保护。

```
function isCat(a:any): a is Cat {
  return a.name === 'kitty';
}

let x:Cat|Dog;

if (isCat(x)) {
  x.meow(); // 正确，因为 x 肯定是 Cat 类型
}
```

上面示例中，函数 `isCat()` 的返回类型是 `a is Cat`，它是一个布尔值。后面的 `if` 语句就用这个返回值进行判断，从而起到类型保护的作用，确保 `x` 是 Cat 类型，从而 `x.meow()` 不会报错（假定 `Cat` 类型拥有 `meow()` 方法）。

`is` 运算符还有一种特殊用法，就是用在类（ class）的内部，描述类的方法的返回值。

```
class Teacher {
  isStudent():this is Student {
    return false;
  }
}

class Student {
  isStudent():this is Student {
    return true;
  }
}
```

上面示例中，`isStudent()` 方法的返回值类型，取决于该方法内部的 `this` 是否为 `Student` 对象。如果是的，就返回布尔值 `true`，否则返回 `false`。

注意，`this is T`这种写法，只能用来描述方法的返回值类型，而不能用来描述属性的类型。

模板字符串

TypeScript 允许使用模板字符串，构建类型。

模板字符串的最大特点，就是内部可以引用其他类型。

```
type World = "world";

// "hello world"
type Greeting = `hello ${World}`;
```

上面示例中，类型`Greeting`是一个模板字符串，里面引用了另一个字符串类型`world`，因此`Greeting`实际上是字符串`hello world`。

注意，模板字符串可以引用的类型一共 7 种，分别是 string、number、bigint、boolean、null、undefined、Enum。引用这 7 种以外的类型会报错。

```
type Num = 123;
type Obj = { n : 123 };

type T1 = `${Num} received`; // 正确
type T2 = `${Obj} received`; // 报错
```

上面示例中，模板字符串引用数值类型的别名`Num`是可以的，但是引用对象类型的别名`Obj`就会报错。

模板字符串里面引用的类型，如果是一个联合类型，那么它返回的也是一个联合类型，即模板字符串可以展开联合类型。

```
type T = 'A'|'B';

// "A_id"|"B_id"
type U = `${T}_id`;
```

上面示例中，类型`U`是一个模板字符串，里面引用了一个联合类型`T`，导致最后得到的也是一个联合类型。

如果模板字符串引用两个联合类型，它会交叉展开这两个类型。

```
type T = 'A'|'B';
```

```
type U = '1'|'2';

// 'A1'|'A2'|'B1'|'B2'
type V = `${T}${U}`;
```

上面示例中，`T`和`U`都是联合类型，各自有两个成员，模板字符串里面引用了这两个类型，最后得到的就是一个4个成员的联合类型。

satisfies 运算符

`satisfies`运算符用来检测某个值是否符合指定类型。有时候，不方便将某个值指定为某种类型，但是希望这个值符合类型条件，这时候就可以用`satisfies`运算符对其进行检测。[TypeScript 4.9](#)添加了这个运算符。

举例来说，有一个对象的属性名拼写错误。

```
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255] // 属性名拼写错误
};
```

上面示例中，对象`palette`的属性名拼写错了，将`blue`拼成了`bleu`，我们希望通过指定类型，发现这个错误。

```
type Colors = "red" | "green" | "blue";
type RGB = [number, number, number];

const palette: Record<Colors, string|RGB> = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255] // 报错
};
```

上面示例中，变量`palette`的类型被指定为`Record<Colors, string|RGB>`，这是一个类型工具，用来返回一个对象，详细介绍见《类型工具》一章。简单说，它的第一个类型参数指定对象的属性名，第二个类型参数指定对象的属性值。

本例的`Record<Colors, string|RGB>`，就表示变量`palette`的属性名应该符合类型`Colors`，属性值应该符合类型`string|RGB`，要么是字符串，要么是元组`RGB`。属性名`bleu`不符合类型`Colors`

，所以就报错了。

这样的写法，虽然可以发现属性名的拼写错误，但是带来了新的问题。

```
const greenComponent = palette.green.substring(1, 6); // 报错
```

上面示例中，`palette.green`属性调用`substring()`方法会报错，原因是这个方法只有字符串才有，而`palette.green`的类型是`string|RGB`，除了字符串，还可能是元组`RGB`，而元组并不存在`substring()`方法，所以报错了。

如果要避免报错，要么精确给出变量`palette`每个属性的类型，要么对`palette.green`的值进行类型缩小。两种做法都比较麻烦，也不是很有必要。

这时就可以使用`satisfies`运算符，对`palette`进行类型检测，但是不改变 TypeScript 对`palette`的类型推断。

```
type Colors = "red" | "green" | "blue";
type RGB = [number, number, number];

const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255] // 报错
} satisfies Record<Colors, string|RGB>;

const greenComponent = palette.green.substring(1); // 不报错
```

上面示例中，变量`palette`的值后面增加了`satisfies Record<Colors, string|RGB>`，表示该值必须满足`Record<Colors, string|RGB>`这个条件，所以能够检测出属性名`bleu`的拼写错误。同时，它不会改变`palette`的类型推断，所以，TypeScript 知道`palette.green`是一个字符串，对其调用`substring()`方法就不会报错。

`satisfies`也可以检测属性值。

```
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  blue: [0, 0] // 报错
} satisfies Record<Colors, string|RGB>;
```

上面示例中，属性`blue`的值只有两个成员，不符合元组`RGB`必须有三个成员的条件，从而报错了。

版本号 #2
由 壳壳虫 创建于 2023-07-09 12:07:22 UTC
由 壳壳虫 更新于 2026-06-06 10:51:00 UTC