TypeScript 的类型映射

简介

映射（mapping）指的是，将一种类型按照映射规则，转换成另一种类型，通常用于对象类型。

举例来说，现有一个类型 `A` 和另一个类型 `B` 。

typescript

```
type A = {
  foo: number;
  bar: number;
};

type B = {
  foo: string;
  bar: string;
};
```

上面示例中，这两个类型的属性结构是一样的，但是属性的类型不一样。如果属性数量多的话，逐个写起来就很麻烦。

使用类型映射，就可以从类型 `A` 得到类型 `B` 。

typescript

```
type A = {
  foo: number;
  bar: number;
};

type B = {
  [prop in keyof A]: string;
};
```

上面示例中，类型 `B` 采用了属性名索引的写法， `[prop in keyof A]` 表示依次得到类型 `A` 的所有属性名，然后将每个属性的类型改成 `string` 。

在语法上， `[prop in keyof A]` 是一个属性名表达式，表示这里的属性名需要计算得到。具体的计算规则如下：

- `prop` ： 属性名变量，名字可以随便起。
- `in` ： 运算符，用来取出右侧的联合类型的每一个成员。
- `Keyof A` ： 返回类型 `A` 的每一个属性名，组成一个联合类型。

下面是复制原始类型的例子。

typescript

```typescript
type A = {
  foo: number;
  bar: string;
};

type B = {
  [prop in keyof A]: A[prop];
};
```

上面示例中，类型 `B` 原样复制了类型 `A` 。

为了增加代码复用性，可以把常用的映射写成泛型。

typescript

```typescript
type ToBoolean<Type> = {
  [Property in keyof Type]: boolean;
};
```

上面示例中，定义了一个泛型，可以将其他对象的所有属性值都改成 boolean 类型。

下面是另一个例子。

typescript

```typescript
type MyObj = {
  [P in 0 | 1 | 2]: string;
};

// 等同于
type MyObj = {
  0: string;
```

```
  1: string;
  2: string;
};
```

上面示例中，联合类型 `0|1|2` 映射成了三个属性名。

不使用联合类型，直接使用某种具体类型进行属性名映射，也是可以的。

typescript

```
type MyObj = {
  [p in "foo"]: number;
};

// 等同于
type MyObj = {
  foo: number;
};
```

上面示例中， `p in 'foo'` 可以看成只有一个成员的联合类型，因此得到了只有这一个属性的对象类型。

甚至还可以写成 `p in string` 。

typescript

```
type MyObj = {
  [p in string]: boolean;
};

// 等同于
type MyObj = {
  [p: string]: boolean;
};
```

上面示例中， `[p in string]` 就是属性名索引形式 `[p: string]` 的映射写法。

通过映射，可以某个对象的所有属性改成可选属性。

typescript

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

```
type B = {
  [Prop in keyof A]?: A[Prop];
};
```

上面示例中，类型 `B` 在类型 `A` 的所有属性名后面添加问号，使得这些属性都变成了可选属性。

事实上，TypeScript 的内置工具类型 `Partial<T>` ，就是这样实现的。

TypeScript 内置的工具类型 `Readonly<T>` 可以将所有属性改为只读属性，实现也是通过映射。

typescript

```
// 将 T 的所有属性改为只读属性
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};
```

它的用法如下。

typescript

```
type T = { a: string; b: number };

type ReadonlyT = Readonly<T>;
// {
//   readonly a: string;
//   readonly b: number;
// }
```

映射修饰符

映射会原样复制原始对象的可选属性和只读属性。

typescript

```
type A = {
  a?: string;
  readonly b: number;
};

type B = {
  [Prop in keyof A]: A[Prop];
};
```

```typescript
// 等同于
type B = {
  a?: string;
  readonly b: number;
};
```

上面示例中，类型 `B` 是类型 `A` 的映射，把 `A` 的可选属性和只读属性都保留下来。

如果要删改可选和只读这两个特性，并不是很方便。为了解决这个问题，TypeScript 引入了两个映射修饰符，用来在映射时添加或移除某个属性的 `?` 修饰符和 `readonly` 修饰符。

- `+` 修饰符：写成 `+?` 或 `+readonly` ，为映射属性添加 `?` 修饰符或 `readonly` 修饰符。
- `-` 修饰符：写成 `-?` 或 `-readonly` ，为映射属性移除 `?` 修饰符或 `readonly` 修饰符。

下面是添加或移除可选属性的例子。

typescript

```typescript
// 添加可选属性
type Optional<Type> = {
  [Prop in keyof Type]+?: Type[Prop];
};

// 移除可选属性
type Concrete<Type> = {
  [Prop in keyof Type]-?: Type[Prop];
};
```

注意， `+?` 或 `-?` 要写在属性名的后面。

下面是添加或移除只读属性的例子。

typescript

```typescript
// 添加 readonly
type CreateImmutable<Type> = {
  +readonly [Prop in keyof Type]: Type[Prop];
};

// 移除 readonly
type CreateMutable<Type> = {
  -readonly [Prop in keyof Type]: Type[Prop];
};
```

注意， `+readonly` 和 `-readonly` 要写在属性名的前面。

如果同时增删 `?` 和 `readonly` 这两个修饰符，写成下面这样。

typescript

```
// 增加
type MyObj<T> = {
  +readonly [P in keyof T]+?: T[P];
};

// 移除
type MyObj<T> = {
  -readonly [P in keyof T]-?: T[P];
};
```

TypeScript 原生的工具类型 `Required<T>` 专门移除可选属性，就是使用 `-?` 修饰符实现的。

注意， `-?` 修饰符移除了可选属性以后，该属性就不能等于 `undefined` 了，实际变成必选属性了。但是，这个修饰符不会移除 `null` 类型。

另外， `+?` 修饰符可以简写成 `?` ， `+readonly` 修饰符可以简写成 `readonly` 。

typescript

```
type A<T> = {
  +readonly [P in keyof T]+?: T[P];
};

// 等同于
type B<T> = {
  readonly [P in keyof T]?: T[P];
};
```

键名重映射

语法

TypeScript 4.1 引入了键名重映射（key remapping），允许改变键名。

typescript

```typescript
type A = {
  foo: number;
  bar: number;
};

type B = {
  [p in keyof A as `${p}ID`]: number;
};

// 等同于
type B = {
  fooID: number;
  barID: number;
};
```

上面示例中，类型 `B` 是类型 `A` 的映射，但在映射时把属性名改掉了，在原始属性名后面加上了字符串 `ID` 。

可以看到，键名重映射的语法是在键名映射的后面加上 `as + 新类型` 子句。这里的“新类型”通常是一个模板字符串，里面可以对原始键名进行各种操作。

下面是另一个例子。

typescript

```typescript
interface Person {
  name: string;
  age: number;
  location: string;
}

type Getters<T> = {
  [P in keyof T as `get${Capitalize<string & P>}`]: () => T[P];
};

type LazyPerson = Getters<Person>;
// 等同于
type LazyPerson = {
  getName: () => string;
  getAge: () => number;
  getLocation: () => string;
};
```

上面示例中，类型 `LazyPerson` 是类型 `Person` 的映射，并且把键名改掉了。

它的修改键名的代码是一个模板字符串 `get${Capitalize<string & P>}` ，下面是各个部分的解释。

- `get` ：为键名添加的前缀。
- `Capitalize<T>` ：一个原生的工具泛型，用来将 `T` 的首字母变成大写。
- `string & P` ：一个交叉类型，其中的 `P` 是 keyof 运算符返回的键名联合类型 `string|number|symbol` ，但是 `Capitalize<T>` 只能接受字符串作为类型参数，因此 `string & P` 只返回 `P` 的字符串属性名。

属性过滤

键名重映射还可以过滤掉某些属性。下面的例子是只保留字符串属性。

typescript

```typescript
type User = {
  name: string;
  age: number;
};

type Filter<T> = {
  [K in keyof T as T[K] extends string ? K : never]: string;
};

type FilteredUser = Filter<User>; // { name: string }
```

上面示例中，映射 `K in keyof T` 获取类型 `T` 的每一个属性以后，然后使用 `as Type` 修改键名。

它的键名重映射 `as T[K] extends string ? K : never]` ，使用了条件运算符。如果属性值 `T[K]` 的类型是字符串，那么属性名不变，否则属性名类型改为 `never` ，即这个属性名不存在。这样就等于过滤了不符合条件的属性，只保留属性值为字符串的属性。

联合类型的映射

由于键名重映射可以修改键名类型，所以原始键名的类型不必是 `string|number|symbol` ，任意的联合类型都可以用来进行键名重映射。

typescript

```typescript
type S = {
  kind: "square";
```

```
  x: number;
  y: number;
};

type C = {
  kind: "circle";
  radius: number;
};

type MyEvents<Events extends { kind: string }> = {
  [E in Events as E["kind"]]: (event: E) => void;
};

type Config = MyEvent<S | C>;
// 等同于
type Config = {
  square: (event: S) => void;
  circle: (event: C) => void;
};
```

上面示例中，原始键名的映射是 `E in Events` ，这里的 `Events` 是两个对象组成的联合类型 `S|C` 。所以， `E` 是一个对象，然后再通过键名重映射，得到字符串键名 `E['kind']` 。

参考链接

- Mapped Type Modifiers in TypeScript, Marius Schulz

限时抢

推荐机场 → 25元/月，500G 购买。

最后更新: 2023/8/13 15:25