22 使用映射类型得到新的类型

更新时间：2019-07-01 11:43:19

> 知识是一种快乐，而好奇则是知识的萌芽。
>
> ——培根

3.9.1 基础

TS 提供了借助旧类型创建一个新类型的方式，也就是映射类型，它可以用相同的形式去转换旧类型中每个属性。来看个例子：

```
interface Info {
  age: number;
}
```

我们可以使用这个接口实现一个有且仅有一个 age 属性的对象，但如果我们想再创建一个只读版本的同款对象，那我们可能需要再重新定义一个接口，然后让 age 属性 readonly。如果接口就这么简单，你确实可以这么做，但是如果属性多了，而且这个结构以后会变，那就比较麻烦了。这种情况我们可以使用映射类型，下面来看例子：

```
interface Info {
  age: number;
}
type ReadonlyType<T> = { readonly [P in keyof T]: T[P] }; // 这里定义了一个ReadonlyType<T>映射类型
type ReadonlyInfo = ReadonlyType<Info>;
let info: ReadonlyInfo = {
  age: 18
};
info.age = 28; // error Cannot assign to 'age' because it is a constant or a read-only property
```

这个例子展示了如何通过一个普通的接口创建一个每个属性都只读的接口，这个过程有点像定义了一个函数，这个函数会遍历传入对象的每个属性并做处理。同理你也可以创建一个每个属性都是可选属性的接口：

```
interface Info {
  age: number;
}
type ReadonlyType<T> = { readonly [P in keyof T]?: T[P] };
type ReadonlyInfo = ReadonlyType<Info>;
let info: ReadonlyInfo = {};
```

注意了，我们在这里用到了一个新的操作符 in，TS 内部使用了 for … in，定义映射类型，这里涉及到三个部分：

- 类型变量，也就是上例中的 P，它就像 for…in 循环中定义的变量，用来在每次遍历中绑定当前遍历到的属性名；
- 属性名联合，也就是上例中`keyof T`，它返回对象 T 的属性名联合；
- 属性的结果类型，也就是 T[P]。

因为这两个需求较为常用，所以 TS 内置了这两种映射类型，无需定义即可使用，它们分别是`Readonly`和`Partial`。还有两个内置的映射类型分别是`Pick`和`Record`，它们的实现如下：

```
type Pick<T, K extends keyof T> = { [P in K]: T[P] };
type Record<K extends keyof any, T> = { [P in K]: T };
```

先来使用一下 Pick，官方文档的例子并不完整，我们来看完整的例子：

```
interface Info {
  name: string;
  age: number;
  address: string;
}
const info: Info = {
  name: "lison",
  age: 18,
  address: "beijing"
};
function pick<T, K extends keyof T>(obj: T, keys: K[]): Pick<T, K> { // 这里我们定义一个pick函数，用来返回一个对象中指定字段的值组成的对象
  let res = {} as Pick<T, K>;
  keys.forEach(key => {
    res[key] = obj[key];
  });
  return res;
}
const nameAndAddress = pick(info, ["name", "address"]); // { name: 'lison', address: 'beijing' }
```

另外一个就是 Record，它适用于将一个对象中的每一个属性转换为其他值的场景，来看例子：

```
function mapObject<K extends string | number, T, U>(
  obj: Record<K, T>,
  f: (x: T) => U
): Record<K, U> {
  let res = {} as Record<K, U>;
  for (const key in obj) {
    res[key] = f(obj[key]);
  }
  return res;
}

const names = { 0: "hello", 1: "world", 2: "bye" };
const lengths = mapObject(names, s => s.length); // { 0: 5, 1: 5, 2: 3 }
```

我们输入的对象属性值为字符串类型，输出的对象属性值为数值类型。

讲完这四个内置的映射类型之后，我们需要讲一个概念——同态。同态在维基百科的解释是：两个相同类型的代数结构之间的结构保持映射。这四个内置映射类型中，Readonly、Partial 和 Pick 是同态的，而 Record 不是，因为 Record 映射出的对象属性值是新的，和输入的值的属性值不同。

3.9.2 由映射类型进行推断

我们学习了使用映射类型包装一个类型的属性后，也可以进行逆向操作，也就是拆包，先来看我们的包装操作：

```
type Proxy<T> = { // 这里定义一个映射类型，他将一个属性拆分成get/set方法
  get(): T;
  set(value: T): void;
};
type Proxify<T> = { [P in keyof T]: Proxy<T[P]> }; // 这里再定义一个映射类型，将一个对象的所有属性值类型都变为Proxy<T>处理之后的类型
function proxify<T>(obj: T): Proxify<T> { // 这里定义一个proxify函数，用来将对象中所有属性的属性值改为一个包含get和set方法的对象
  let result = {} as Proxify<T>;
  for (const key in obj) {
    result[key] = {
      get: () => obj[key],
      set: value => (obj[key] = value)
    };
  }
  return result;
}
let props = {
  name: "lison",
  age: 18
};
let proxyProps = proxify(props);
console.log(proxyProps.name.get()); // "lison"
proxyProps.name.set("li");
```

我们来看下这个例子，这个例子我们定义了一个函数，这个函数可以把传入的对象的每个属性的值替换为一个包含 get 和 set 两个方法的对象。最后我们获取某个值的时候，比如 name，就使用 proxyProps.name.get()方法获取它的值，使用 proxyProps.name.set()方法修改 name 的值。

接下来我们来看如何进行拆包：

```
function unproxify<T>(t: Proxify<T>): T { // 这里我们定义一个拆包函数，其实就是利用每个属性的get方法获取到当前属性值，然后将原本是包含get和set方法的对象改为这个属性值
  let result = {} as T;
  for (const k in t) {
    result[k] = t[k].get(); // 这里通过调用属性值这个对象的get方法获取到属性值，然后赋给这个属性，替换掉这个对象
  }
  return result;
}
let originalProps = unproxify(proxyProps);
```

3.9.3 增加或移除特定修饰符

TS 在 2.8 版本为映射类型增加了增加或移除特定修饰符的能力，使用`+`和`-`符号作为前缀来指定增加还是删除修饰符。首先来看我们如何通过映射类型为一个接口的每个属性增加修饰符，我们这里使用+前缀：

```
interface Info {
  name: string;
  age: number;
}
type ReadonlyInfo<T> = { +readonly [P in keyof T]+?: T[P] };
let info: ReadonlyInfo<Info> = {
  name: "lison"
};
info.name = ""; // error
```

这个例子中，经过 ReadonlyInfo 创建的接口类型，属性是可选的，所以我们在定义 info 的时候没有写 age 属性也没问题，同时每个属性是只读的，所以我们修改 name 的值的时候报错。我们通过+前缀增加了 readonly 和?修饰符。当然，增加的时候，这个+前缀可以省略，也就是说，上面的写法和`type ReadonlyInfo = { readonly [P in keyof T]?: T[P] }`是一样的。我们再来看下怎么删除修饰符：

```
interface Info {
  name: string;
  age: number;
}
type RemoveModifier<T> = { -readonly [P in keyof T]-?: T[p] };
type InfoType = RemoveModifier<Readonly<Partial<Info>>>;
let info1: InfoType = {
  // error missing "age"
  name: "lison"
};
let info2: InfoType = {
  name: "lison",
  age: 18
};
info2.name = ""; // right, can edit
```

这个例子我们定义了去掉修饰符的映射类型 RemoveModifier，`Readonly<Partial<Info>>`则是返回一个既属性可选又只读的接口类型，所以 InfoType 类型则表示属性必含而且非只读。

TS 内置了一个映射类型`Required<T>`，使用它可以去掉 T 所有属性的`?`修饰符。

3.9.4 keyof 和映射类型在 2.9 的升级

TS 在 2.9 版本中，keyof 和映射类型支持用 number 和 symbol 命名的属性，我们先来看 keyof 的例子：

```
const stringIndex = "a";
const numberIndex = 1;
const symbolIndex = Symbol();
type Obj = {
  [stringIndex]: string;
  [numberIndex]: number;
  [symbolIndex]: symbol;
};
type keys = keyof Obj;
let key: keys = 2; // error
let key: keys = 1; // right
let key: keys = "b"; // error
let key: keys = "a"; // right
let key: keys = Symbol(); // error
let key: keys = symbolIndex; // right
```

再来看个映射类型的例子：

```ts
const stringIndex = "a";
const numberIndex = 1;
const symbolIndex = Symbol();
type Obj = {
  [stringIndex]: string;
  [numberIndex]: number;
  [symbolIndex]: symbol;
};
type ReadonlyType<T> = { readonly [P in keyof T]?: T[P] };
let obj: ReadonlyType<Obj> = {
  a: "aa",
  1: 11,
  [symbolIndex]: Symbol()
};
obj.a = "bb"; // error Cannot assign to 'a' because it is a read-only property
obj[1] = 22; // error Cannot assign to '1' because it is a read-only property
obj[symbolIndex] = Symbol(); // error Cannot assign to '[symbolIndex]' because it is a read-only property
```

3.9.5 元组和数组上的映射类型

TS 在 3.1 版本中，在元组和数组上的映射类型会生成新的元组和数组，并不会创建一个新的类型，这个类型上会具有 push、pop 等数组方法和数组属性。来看例子：

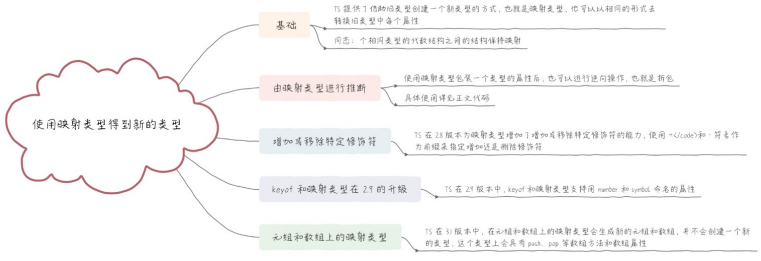
```ts
type MapToPromise<T> = { [K in keyof T]: Promise<T[K]> };
type Tuple = [number, string, boolean];
type promiseTuple = MapToPromise<Tuple>;
let tuple: promiseTuple = [
  new Promise((resolve, reject) => resolve(1)),
  new Promise((resolve, reject) => resolve("a")),
  new Promise((resolve, reject) => resolve(false))
];
```

这个例子中定义了一个MapToPromise映射类型。它返回一个将传入的类型的所有字段的值转为Promise，且Promise的resolve回调函数的参数类型为这个字段类型。我们定义了一个元组Tuple，元素类型分别为number、string和boolean，使用MapToPromise映射类型将这个元组类型传入，并且返回一个promiseTuple类型。当我们指定变量tuple的类型为promiseTuple后，它的三个元素类型都是一个Promise，且resolve的参数类型依次为number、string和boolean。

本节小结

本小节我们学习了映射类型的相关知识，我们学习了映射类型的基础应用，它的定义和使用像极了函数的定义和使用。函数是处理实际的值，而映射类型处理的是类型。我们还通过一个例子学习了由映射类型进行推断，根据映射类型推断出处理前的类型，也就是拆包操作。通过增加或移除特定修饰符"+“和”-“可以实现给字段添加或移除一些readonly等修饰符，但用的最多的是”-"。因为如果需要给某个字段加修饰符，"+"是可以省略不写的。最后我们补充了两个TypeScript在后面升级中对映射类型的更新。

下个小节我们将对前面讲TypeScript中补充的六个类型中跳过的unknown类型进行详细补充学习。



21 索引类型：获取索引类型和索引值类型

23 前面跳过的unkown类型详解