09 使用接口定义几乎任意结构

更新时间：2019-06-12 16:37:17

> 低头要有勇气，抬头要有底气。
>
> ——韩寒

本小节我们来学习接口，正如题目所说的，你可以使用接口定义几乎任意结构，本小节我们先来学习下接口的基本使用方法。

2.6.1. 基本用法

我们需要定义这样一个函数，参数是一个对象，里面包含两个字段：firstName 和 lastName，也就是英文的名和姓，然后返回一个拼接后的完整名字。来看下函数的定义：

```
// 注：这段代码为纯JavaScript代码，请在JavaScript开发环境编写下面代码，在TypeScript环境会报一些类型错误
const getFullName = ({ firstName, lastName }) => {
  return `${firstName} ${lastName}`;
};
```

使用时传入参数：

```
getFullName({
  firstName: "Lison",
  lastName: "Li"
}); // => 'Lison Li'
```

没有问题，我们得到了拼接后的完整名字，但是使用这个函数的人如果传入一些不是很理想的参数时，就会导致各种结果：

```
getFullName(); // Uncaught TypeError: Cannot destructure property `a` of 'undefined' or 'null'.
getFullName({ age: 18, phone: "13312345678" }); // 'undefined undefined'
getFullName({ firstName: "Lison" }); // 'Lison undefined'
```

这些都是我们不想要的，在开发时难免会传入错误的参数，所以 TypeScript 能够帮我们在编译阶段就检测到这些错误。我们来完善下这个函数的定义：

```
const getFullName = ({
  firstName,
  lastName,
}: { // 指定这个参数的类型，因为他是一个对象，所以这里来指定对象中每个字段的类型
  firstName: string; // 指定属性名为firstName和lastName的字段的属性值必须为string类型
  lastName: string;
}) => {
  return `${firstName} ${lastName}`;
};
```

我们通过对象字面量的形式去限定我们传入的这个对象的结构，现在再来看下之前的调用会出现什么提示：

```
getFullName(); // 应有1个参数，但获得0个
getFullName({ age: 18, phone: 123456789 }); // 类型“{ age: number; phone: number; }”的参数不能赋给类型“{ firstName: string; lastName: string; }”的参数。
getFullName({ firstName: "Lison" }); // 缺少必要属性lastName
```

这些都是在我们编写代码的时候 TypeScript 提示给我们的错误信息，这样就避免了在使用函数的时候传入不正确的参数。接下来我们用这节课要讲的接口来书写上面的规则，我们使用 `interface` 来定义接口：

```
interface Info {
  firstName: string;
  lastName: string;
}
const getFullName = ({ firstName, lastName }: Info) =>
  `${firstName} ${lastName}`;
```

注意在定义接口的时候，你不要把它理解为是在定义一个对象，而要理解为{}括号包裹的是一个代码块，里面是一条条声明语句，只不过声明的不是变量的值而是类型。声明也不用等号赋值，而是冒号指定类型。每条声明之前用换行分隔即可，或者也可以使用分号或者逗号，都是可以的。

2.6.2.可选属性

当我们定义一些结构的时候，一些结构对于某些字段的要求是可选的，有这个字段就做处理，没有就忽略，所以针对这种情况，*typescript*为我们提供了可选属性。

我们先定义一个描述传入蔬菜信息的句子的函数：

```
const getVegetables = ({ color, type }) => {
  return `A ${color ? color + " " : ""}${type}`;
};
```

我们可以看到这个函数中根据传入对象中的 color 和 type 来进行描述返回一句话，color 是可选的，所以我们可以给接口设置可选属性，在属性名后面加个 `?` 即可：

```
interface Vegetables {
  color?: string;
  type: string;
}
```

> 这里可能 tslint 会报一个警告，告诉我们接口应该以大写的 `I` 开头，如果你想关闭这条规则，可以在 tslint.json 的 rules 里添加 `"interface-name": [true, “never-prefix”]` 来关闭。

2.6.3.多余属性检查

```
getVegetables({
  type: "tomato",
  size: "big" // 'size'不在类型'Vegetables'中
});
```

我们看到，传入的参数没有 color 属性，但也没有错误，因为它是可选属性。但是我们多传入了一个 size 属性，这同样会报错，TypeScript 会告诉你，接口上不存在你多余的这个属性。只要接口中没有定义这个属性，就会报错，但如果你定义了可选属性 size，那么上面的例子就不会报错。

> 这里可能 tslint 会报一个警告，告诉我们属性名没有按开头字母顺序排列属性列表，如果你想关闭这条规则，可以在 tslint.json 的 rules 里添加 `"object-literal-sort-keys": [false]` 来关闭。

2.6.4.绕开多余属性检查

有时我们并不希望 TypeScript 这么严格地对我们的数据进行检查，比如我们只需要保证传入 `getVegetables` 的对象有 `type` 属性就可以了，至于实际使用的时候传入对象有没有多余的属性，多余属性的属性值是什么类型，这些都无所谓，那就需要绕开多余属性检查，有如下三个方法：

(1) 使用类型断言

我们在基础类型中讲过，类型断言就是用来明确告诉 TypeScript，我们已经自行进行了检查，确保这个类型没有问题，希望 TypeScript 对此不进行检查，所以最简单的方式就是使用类型断言：

```
interface Vegetables {
  color?: string;
  type: string;
}
const getVegetables = ({ color, type }: Vegetables) => {
  return `A ${color ? color + " " : ""}${type}`;
};
getVegetables({
  type: "tomato",
  size: 12,
  price: 1.2
} as Vegetables);
```

(2) 添加索引签名

更好的方式是添加字符串索引签名，索引签名我们会在后面讲解，先来看怎么实现：

```
interface Vegetables {
  color: string;
  type: string;
  [prop: string]: any;
}
const getVegetables = ({ color, type }: Vegetables) => {
  return `A ${color ? color + " " : ""}${type}`;
};
getVegetables({
  color: "red",
  type: "tomato",
  size: 12,
  price: 1.2
});
```

(3) 利用类型兼容性

这种方法现在还不是很好理解，也是不推荐使用的，先来看写法：

```
interface Vegetables {
  type: string;
}
const getVegetables = ({ type }: Vegetables) => {
  return `A ${type}`;
};

const option = { type: "tomato", size: 12 };
getVegetables(option);
```

上面这种方法完美通过检查，我们将对象字面量赋给一个变量 option，然后 getVegetables 传 入 option，这时没有报错。是因为直接将对象字面量传入函数，和先赋给变量再将变量传入函数，这两种检查机制是不一样的，后者是因为类型兼容性。我们后面会有专门一节来讲类型兼容性。简单地来说：如果 b 要赋值给 a，那要求 b 至少需要与 a 有相同的属性，多了无所谓。

在上面这个例子中， option 的类型应该是 Vegetables 类型，对象 { type: 'tomato', size: 12 } 要赋值给 option， optio n 中所有的属性在这个对象字面量中都有，所以这个对象的类型和 option (也就是 Vegetables 类型)是兼容的，所以上面例子不会报错。如果你现在还想不明白没关系，我们还会在后面详细去讲。

2.6.5.只读属性

接口也可以设置只读属性，如下：

```
interface Role {
  readonly 0: string;
  readonly 1: string;
}
```

这里我们定义了一个角色字典，有 0 和 1 两种角色 id。下面我们定义一个实际的角色 数据，然后来试图修改一下它的值：

```
const role: Role = {
  0: "super_admin",
  1: "admin"
};
role[1] = "super_admin"; // Cannot assign to '0' because it is a read-only property
```

我们看到 TypeScript 告诉我们不能分配给索引0，因为它是只读属性。设置一个值只读，我们是否想到ES6里定义常量的关键字 const ？使用 const 定义的常量定义之后不能再修改，这有点只读的意思。那 readonly 和 const 在使用时该如何选择呢？这主要看你这个值的用途，如果是定义一个常量，那用 const ，如果这个值是作为对象的属性，那请用 readonly 。我们来看下面的代码：

```
const NAME: string = "Lison";
NAME = "Haha"; // Uncaught TypeError: Assignment to constant variable

const obj = {
  name: "lison"
};
obj.name = "Haha";

interface Info {
  readonly name: string;
}
const info: Info = {
  name: "Lison"
};
info["name"] = "Haha"; // Cannot assign to 'name' because it is a read-only property
```

我们可以看到上面使用`const`定义的常量`NAME`定义之后再修改会报错，但是如果使用`const`定义一个对象，然后修改对象里属性的值是不会报错的。所以如果我们要保证对象的属性值不可修改，需要使用`readonly`。

2.6.6.函数类型

接口可以描述普通对象，还可以描述函数类型，我们先看写法：

```
interface AddFunc {
  (num1: number, num2: number): number;
}
```

这里我们定义了一个AddFunc结构，这个结构要求实现这个结构的值，必须包含一个和结构里定义的函数一样参数、一样返回值的方法，或者这个值就是符合这个函数要求的函数。我们管花括号里包着的内容为*调用签名*，它由带有参数类型的参数列表和返回值类型组成。后面学到**类型别名**一节时我们还会学习其他写法。来看下如何使用：

```
const add: AddFunc = (n1, n2) => n1 + n2;
const join: AddFunc = (n1, n2) => `${n1} ${n2}`; // 不能将类型'string'分配给类型'number'
add("a", 2); // 类型'string'的参数不能赋给类型'number'的参数
```

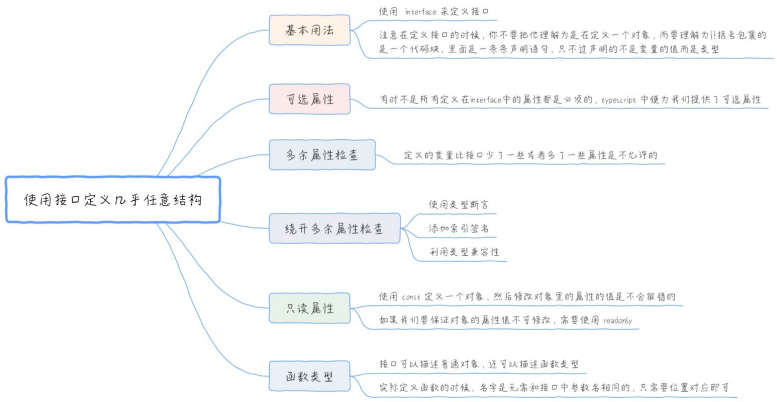
上面我们定义的add函数接收两个数值类型的参数，返回的结果也是数值类型，所以没有问题。而join函数参数类型没错，但是返回的是字符串，所以会报错。而当我们调用add函数时，传入的参数如果和接口定义的类型不一致，也会报错。

你应该注意到了，实际定义函数的时候，名字是无需和接口中参数名相同的，只需要位置对应即可。

小结

本小节我们学习了接口的一些基本定义和用法，通过使用接口，我们可以定义绝大部分的数据结构，从而限定值的结构。我们可以通过修饰符来指定结构中某个字段的可选性和只读性，以及默认情况下必选性。而接口的校验是严格的，在定义一个实现某个接口的值的时候，对于接口中没有定义的字段是不允许出现的，我们称这个为多余属性检查；同时我们讲了三种绕过多余属性检查的方法，来满足程序的灵活性。最后我们学习了如何通过接口，来定义函数类型，当然我们后面还会学习其他定义函数类型的方法。

下个小节，我们将学习接口的高级用法，学习完之后，除了涉及到*类*的知识的部分外，你就掌握了接口的所有知识。



08 使用类型断言达到预期
10 接口的高阶用法