05 TS中补充的六个类型

更新时间：2019-06-21 10:07:34

> 衡量一个人的真正品格，是看他在知道没人看见的时候干些什么。
>
> ——孟德斯鸠

上个小节我们学习了八个JavaScript中常见的数据类型，你也学会了如何给一个变量指定类型。本小节我们将接触几个TypeScript中引入的新类型，这里面可能有你在其他强类型语言中见过的概念，接下来让我们一起来学习。

2.2.1 元组

元组可以看做是数组的拓展，它表示已知元素数量和类型的数组。确切地说，是已知数组中每一个位置上的元素的类型，来看例子：

```
let tuple: [string, number, boolean];
tuple = ["a", 2, false];
tuple = [2, "a", false]; // error 不能将类型“number”分配给类型“string”。 不能将类型“string”分配给类型“number”。
tuple = ["a", 2]; // error Property '2' is missing in type '[string, number]' but required in type '[string, number, boolean]'
```

可以看到，上面我们定义了一个元组 tuple，它包含三个元素，且每个元素的类型是固定的。当我们为 tuple 赋值时：**各个位置上的元素类型都要对应，元素个数也要一致。**

我们还可以给单个元素赋值：

```
tuple[1] = 3;
```

这里我们给元组 tuple 的索引为 1 即第二个元素赋值为 3，第二个元素类型为 number，我们赋值给 3，所以没有问题。

当我们访问元组中元素时，TypeScript 会对我们在元素上做的操作进行检查：

```
tuple[0].split(":"); // right 类型"string"拥有属性"split"
tuple[1].split(":"); // error 类型“number”上不存在属性“split”
```

上面的例子中，我们访问的 tuple 的第二个元素的元素类型为 number，而数值没有 split 方法，所以会报错。

在 2.6 版本之前，TypeScript 对于元组长度的校验和 2.6 之后的版本有所不同，我们来看下面的例子，前后版本对于该情况的处理：

```
let tuple: [string, number];
tuple = ["a", 2]; // right 类型和个数都对应，没问题
// 2.6版本之前如下也不会报错
tuple = ["a", 2, "b"];
// 2.6版本之后如下会报错
tuple = ["a", 2, "b"]; // error 不能将类型“[string, number, string]”分配给类型“[string, number]”。 属性“length”的类型不兼容。
```

这个赋给元组的值有三个元素，是比我们定义的元组类型元素个数多的：

- 在 2.6 及之前版本中，超出规定个数的元素称作**越界元素**，但是只要越界元素的类型是定义的类型中的一种即可。比如我们定义的类型有两种：string 和 number，越界的元素是 string 类型，属于联合类型 `string | number`，所以没问题，联合类型的概念我们后面会讲到。
- 在 2.6 之后的版本，去掉了这个**越界元素是联合类型的子类型即可**的条件，要求元组赋值必须类型和个数都对应。

在 2.6 之后的版本，[string, number]元组类型的声明效果上可以看做等同于下面的声明：

```
interface Tuple extends Array<number | string> {
  0: string;
  1: number;
  length: 2;
}
```

上面这个声明中，我们定义接口 `Tuple`，它继承数组类型，并且数组元素的类型是 `number 和 string` 构成的联合类型，这样接口 `Tuple` 就拥有了数组类型所有的特性。并且我们明确指定索引为0的值为 `string` 类型，索引为1的值为 `number` 类型，同时我们指定 `length` 属性的类型字面量为 2，这样当我们再指定一个类型为这个接口 `Tuple` 的时候，这个值必须是数组，而且如果元素个数超过2个时，它的length就不是2是大于2的数了，就不满足这个接口定义了，所以就会报错；当然，如果元素个数不够2个也会报错，因为索引为0或1的值缺失。接口我们后面会在后面专门的一节来讲，所以暂时不懂也没关系。

如果你想要和 2.6 及之前版本一样的元组特性，那你可以这样定义接口：

```
interface Tuple extends Array<number | string> {
  0: string;
  1: number;
}
```

也就是去掉接口中定义的 `length: 2`，这样 `Tuple` 接口的 `length` 就是从 `Array` 继承过来的 `number` 类型，而不用必须是 `2` 了。

2.2.2 枚举

`enum` 类型在 C++这些语言中比较常见，TypeScript 在 ES 原有类型基础上加入枚举类型，使我们在 TypeScript 中也可以给一组数值赋予名字，这样对开发者来说较为友好。比如我们要定义一组角色，每一个角色用一个数字代表，就可以使用枚举类型来定义：

```
enum Roles {
  SUPER_ADMIN,
  ADMIN,
  USER
}
```

上面定义的枚举类型 Roles 里面有三个值，TypeScript 会为它们每个值分配编号，默认从 0 开始，依次排列，所以它们对应的值是：

```
enum Roles {
  SUPER_ADMIN = 0,
  ADMIN = 1,
  USER = 2
}
```

当我们使用的时候，就可以使用名字而不需要记数字和名称的对照关系了：

```
const superAdmin = Roles.SUPER_ADMIN;
console.log(superAdmin); // 0
```

你也可以修改这个数值，比如你想让这个编码从 1 开始而不是 0，可以如下定义：

```
enum Roles {
  SUPER_ADMIN = 1,
  ADMIN,
  USER
}
```

这样当你访问 `Roles.ADMIN` 时，它的值就是 2 了。

你也可以为每个值都赋予不同的、不按顺序排列的值：

```
enum Roles {
  SUPER_ADMIN = 1,
  ADMIN = 3,
  USER = 7
}
```

通过名字 Roles.SUPER_ADMIN 可以获取到它对应的值 1，同时你也可以通过值获取到它的名字，以上面任意数值这个例子为前提：

```
console.log(Roles[3]); // 'ADMIN'
```

更多枚举的知识我们会在后面专门的一节讲解，在这里我们只是先有个初步的认识即可。

2.2.3 Any

JavaScript 的类型是灵活的，程序有时也是多变的。有时，我们在编写代码的时候，并不能清楚地知道一个值到底是什么类型，这时就需要用到 any 类型，即任意类型。我们来看例子：

```
let value: any;
value = 123;
value = "abc";
value = false;
```

你可以看到，我们定义变量 value，指定它的类型为 any，接下来赋予任何类型的值都是可以的。

我们还可以在定义数组类型时使用 any 来指定数组中的元素类型为任意类型：

```
const array: any[] = [1, "a", true];
```

但是请注意，不要滥用 any，如果任何值都指定为 any 类型，那么 TypeScript 将失去它的意义。

所以如果类型是未知的，更安全的做法是使用unknown类型，我们本小节后面会讲到。

2.2.4 void

void 和 any 相反，any 是表示任意类型，而 void 是表示没有任意类型，就是什么类型都不是，这在我们定义函数，函数没有返回值时会用到：

```
const consoleText = (text: string): void => {
  console.log(text);
};
```

这个函数没有返回任何的值，所以它的返回类型为 void。现在你只需知道 void 表达的含义即可，后面我们会用专门的一节来学习函数。

void 类型的变量只能赋值为 `undefined` 和 `null`，其他类型不能赋值给 void 类型的变量。

2.2.5 never

never 类型指那些永不存在的值的类型，它是那些总会抛出异常或根本不会有返回值的函数表达式的返回值类型，当变量被永不为真的类型保护（后面章节会详细介绍）所约束时，该变量也是 never 类型。

这个类型比较难理解，我们先来看几个例子：

```
const errorFunc = (message: string): never => {
  throw new Error(message);
};
```

这个 errorFunc 函数总是会抛出异常，所以它的返回值类型时 never，用来表明它的返回值是永不存在的。

```
const infiniteFunc = (): never => {
  while (true) {}
};
```

`infiniteFunc` 也是根本不会有返回值的函数，它和之前讲 void 类型时的 `consoleText` 函数不同，`consoleText` 函数没有返回值，是我们在定义函数的时候没有给它返回值，而 `infiniteFunc` 是死循环是根本不会返回值的，所以它们二者还是有区别的。

never 类型是任何类型的子类型，所以它可以赋值给任何类型；而没有类型是 never 的子类型，所以除了它自身没有任何类型可以赋值给 never 类型，any 类型也不能赋值给 never 类型。我们来看例子：

```
let neverVariable = (() => {
  while (true) {}
})();
neverVariable = 123; // error 不能将类型"number"分配给类型"never"
```

上面例子我们定义了一个立即执行函数，也就是 `"let neverVariable = "` 右边的内容。右边的函数体内是一个死循环，所以这个函数调用后的返回值类型为 never，所以赋值之后 neverVariable 的类型是 never 类型，当我们给 neverVariable 赋值 123 时，就会报错，因为除它自身外任何类型都不能赋值给 never 类型。

2.2.6 unknown

unknown类型是TypeScript在3.0版本新增的类型，它表示未知的类型，这样看来它貌似和any很像，但是还是有区别的，也就是所谓的“**unknown相对于any是安全的**”。怎么理解呢？我们知道当一个值我们不能确定它的类型的时候，可以指定它是any类型；但是当指定了any类型之后，这个值基本上是“废”了，你可以随意对它进行属性方法的访问，不管有的还是没有的，可以把它当做任意类型的值来使用，这往往会产生问题，如下：

```
let value: any
console.log(value.name)
console.log(value.toFixed())
console.log(value.length)
```

上面这些语句都不会报错，因为value是any类型，所以后面三个操作都有合法的情况，当value是一个对象时，访问name属性是没问题的；当value是数值类型的时候，调用它的toFixed方法没问题；当value是字符串或数组时获取它的length属性是没问题的。

而当你指定值为unknown类型的时候，如果没有通过基于控制流的类型断言来缩小范围的话，是不能对它进行任何操作的，关于类型断言，我们后面小节会讲到。总之这里你知道了，unknown类型的值不是可以随便操作的。

我们这里只是先来了解unknown和any的区别，unknown还有很多复杂的规则，但是涉及到很多后面才学到的知识，所以需要我们学习了高级类型之后才能再讲解。

2.2.7 拓展阅读

这要讲的不是TypeScript中新增的基本类型，而是高级类型中的两个比较常用类型：联合类型和交叉类型。我们之所以要提前讲解，是因为它俩比较简单，而且很是常用，所以我们先来学习下。

(1) 交叉类型

交叉类型就是取多个类型的并集，使用 & 符号定义，被&符链接的多个类型构成一个交叉类型，表示这个类型同时具备这几个连接起来的类型的特点，来看例子：

```
const merge = <T, U>(arg1: T, arg2: U): T & U => {
  let res = <T & U>{}; // 这里指定返回值的类型兼备T和U两个类型变量代表的类型的特点
  res = Object.assign(arg1, arg2); // 这里使用Object.assign方法，返回一个合并后的对象；
                        // 关于该方法，请在例子下面补充中学习
  return res;
};
const info1 = {
  name: "lison"
};
const info2 = {
  age: 18
};
const lisonInfo = merge(info1, info2);

console.log(lisonInfo.address); // error 类型“{ name: string; } & { age: number; }”上不存在属性“address”
```

> 补充阅读：Object.assign方法可以合并多个对象，将多个对象的属性添加到一个对象中并返回，有一点要注意的是，如果属性值是对象或者数组这种保存的是内存引用的引用类型，会保持这个引用，也就是如果在Object.assign返回的的对象中修改某个对象属性值，原来用来合并的对象也会受到影响。

可以看到，传入的两个参数分别是带有属性 name 和 age 的两个对象，所以它俩的交叉类型要求返回的对象既有 name 属性又有 age 属性。

(2) 联合类型

联合类型在前面课时中几次提到，现在我们来看一下。联合类型实际是几个类型的结合，但是和交叉类型不同，联合类型是要求只要符合联合类型中任意一种类型即可，它使用 | 符号定义。当我们的程序具有多样性，元素类型不唯一时，即使用联合类型。

```
const getLength = (content: string | number): number => {
  if (typeof content === "string") return content.length;
  else return content.toString().length;
};
console.log(getLength("abc")); // 3
console.log(getLength(123)); // 3
```

这里我们指定参数既可以是字符串类型也可以是数值类型，这个getLength函数的定义中，其实还涉及到一个知识点，就是类型保护，就是 typeof content === “string” ，后面进阶部分我们会学到。

补充说明

有一个问题我需要在这里提前声明一下，以免你在自己联系专栏中例子的时候遇到困惑。在讲解语法知识的时候，会有很多例子，在定义一些类型值，比如枚举，或者后面讲的接口等的时候，对于他们的命名我并不会考虑重复性，比如我这里讲枚举的定义定义了一个名字叫Status的枚举值，在别处我又定义了一个同名的接口，那这个时候你可能会看到如下这种错误提示：

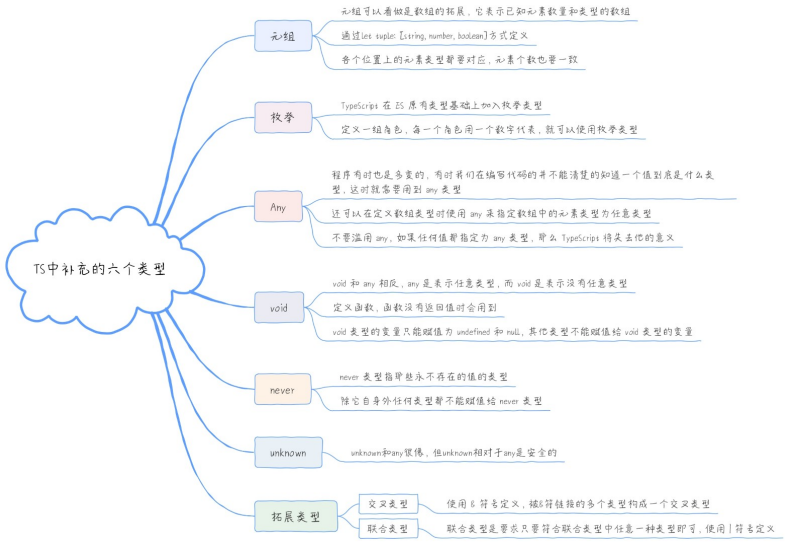
```
枚举声明只能与命名空间或其他枚举声明合并
```

正如你看到的，这里这个错误，是因为你在同一个文件不同地方、或者不同文件中，定义了相同名称的值，而由于TypeScript的声明合并策略，他会将同名的一些可合并的声明进行合并，当同名的两个值或类型不能合并的时候，就会报错；或者可以合并的连个同名的值不符合要求，也会有问题。关于声明合并和哪些声明可以合并，以及声明需要符合的条件等我们会在后面章节学到。这里你只要知道，类似于这种报错中提到“声明合并”的或者无法重新声明块范围变量，可能都是因为有相同名称的定义。

小结

本小节我们学习了六个TypeScript中新增的数据类型，它们是：元组、枚举、Any、void、never和unknown，其中枚举我们会在后面一个单独的小节进行详细学习，unknown会在我们学习了高级类型之后再补充。我们还学习了两个简单的高级类型：联合类型和交叉类型。我们还学习了any类型与never类型和unknown类型相比的区别，简单来说，any和never的概念是对立的，而any和unknown类型相似，但是unknown与any相比是较为安全的类型，它并不允许无条件地随意操作。我们学习的联合类型和交叉类型，是各种类型的结合，我们可以使用几乎任何类型，来组成联合类型和交叉类型。

下个小节我们将详细学习Symbol的所有知识，Symbol是ES6标准提出的新概念，TypeScript已经支持了该语法，下节课我们将进行全面学习。



04 八个JS中你见过的类型

06 Symbol-ES6新基础类型