11 为函数和函数参数定义类型

更新时间：2019-06-12 16:37:39

> 机遇只偏爱那些有准备的头脑。
>
> ——巴斯德

本小节我们来学习函数类型的定义，以及对函数参数的详细介绍。前面我们在讲object例子的时候见过简单的函数定义，在那个例子中我们学习了如何简单地为一个参数指定类型。在本小节你将学习三种定义函数类型的方式，以及关于参数的三个知识——即可选参数、默认参数和剩余参数。接下来我们开始学习。

2.8.1. 函数类型

(1) 为函数定义类型

我们可以给函数定义类型，这个定义包括对参数和返回值的类型定义，我们先来看简单的定义写法：

```
function add(arg1: number, arg2: number): number {
  return x + y;
}
// 或者
const add = (arg1: number, arg2: number): number => {
  return x + y;
};
```

在上面的例子中我们用function和箭头函数两种形式定义了add函数，以展示如何定义函数类型。这里参数 arg1 和 arg2 都是数值类型，最后通过相加得到的结果也是数值类型。

如果在这里省略参数的类型，TypeScript 会默认这个参数是 any 类型；如果省略返回值的类型，如果函数无返回值，那么 TypeScript 会默认函数返回值是 void 类型；如果函数有返回值，那么 TypeScript 会根据我们定义的逻辑推断出返回类型。

(2) 完整的函数类型

一个函数的定义包括函数名、参数、逻辑和返回值。我们为一个函数定义类型时，完整的定义应该包括参数类型和返回值类型。上面的例子中，我们都是在定义函数的指定参数类型和返回值类型。接下来我们看下，如何定义一个完整的函数类型，以及用这个函数类型来规定一个函数定义时参数和返回值需要符合的类型。先来看例子然后再进行解释：

```
let add: (x: number, y: number) => number;
add = (arg1: number, arg2: number): number => arg1 + arg2;
add = (arg1: string, arg2: string): string => arg1 + arg2; // error
```

上面这个例子中，我们首先定义了一个变量 add，给它指定了函数类型，也就是`(x: number, y: number) => number`，这个函数类型包含参数和返回值的类型。然后我们给 add 赋了一个实际的函数，这个函数参数类型和返回类型都和函数类型中定义的一致，所以可以赋值。后面我们又给它赋了一个新函数，而这个函数的参数类型和返回值类型都是 string 类型，这时就会报如下错误：

```
不能将类型"(arg1: string, arg2: string) => string"分配给类型"(x: number, y: number) => number"。
  参数"arg1"和"x" 的类型不兼容。
    不能将类型"number"分配给类型"string"。
```

> 函数中如果使用了函数体之外定义的变量，这个变量的类型是不体现在函数类型定义的。

(3) 使用接口定义函数类型

我们在前面的小节中已经学习了接口，使用接口可以清晰地定义函数类型。还拿上面的 add 函数为例，我们为它使用接口定义函数类型：

```
interface Add {
  (x: number, y: number): number;
}
let add: Add = (arg1: string, arg2: string): string => arg1 + arg2; // error 不能将类型“(arg1: string, arg2: string) => string”分配给类型“Add”
```

这里我们通过接口的形式定义函数类型，这个接口`Add`定义了这个结构是一个函数，两个参数类型都是`number`类型，返回值也是`number`类型。然后我们指定变量add类型为`Add`时，再要给add赋值，就必须是一个函数，且参数类型和返回值类型都要满足接口`Add`，显然例子中这个函数并不满足条件，所以报错了。

(4) 使用类型别名

我们可以使用类型别名来定义函数类型，类型别名我们在后面讲到**高级类型**的时候还会讲到。使用类型别名定义函数类型更直观易读，我们来看一下具体的写法：

```
type Add = (x: number, y: number) => number;
let add: Add = (arg1: string, arg2: string): string => arg1 + arg2; // error 不能将类型“(arg1: string, arg2: string) => string”分配给类型“Add”
```

使用`type`关键字可以为原始值、联合类型、元组以及任何我们定义的类型起一个别名。上面定义了 Add 这个别名后，`Add`就成为了一个和`(x: number, y: number) => number`一致的类型定义。例子中定义了`Add`类型，指定add类型为Add，但是给add赋的值并不满足`Add`类型要求，所以报错了。

2.8.2. 参数

(1) 可选参数

TypeScript 会帮我们在编写代码的时候就检查出调用函数时参数中存在的一些错误，先看下面例子：

```
type Add = (x: number, y: number) => number;
let add: Add = (arg1: string, arg2: string): string => arg1 + arg2;

add(1, 2); // right
add(1, 2, 3); // error 应有 2 个参数，但获得 3 个
add(1); // error 应有 2 个参数，但获得 1 个
```

在 JS 中，上面例子中最后两个函数调用都不会报错，只不过`add(1, 2, 3)`可以返回正确结果`3`，`add(1)`会返回`NaN`。

但有时候，我们的函数有些参数不是必须的，是可选的。在学习接口的时候我们学习过，可选参数只需在参数名后跟随一个`?`即可。但是接口形式的定义和今天学到的函数类型定义有一点区别，那就是参数位置的要求：

> 接口形式定义的函数类型必选参数和可选参数的位置前后是无所谓的，但是今天学到的定义形式，可选参数必须放在必选参数后面，这和在 JS 中定义函数是一致的。

来看下面的例子：

```
type Add = (x?: number, y: number) => number; // error 必选参数不能位于可选参数后。
```

在TypeScript中，可选参数放到最后才行，上面例子中把可选参数x放到了必选参数y前面，所以报错了；但是在 JavaScript 中，其实是没有可选参数这个概念的，只不过是我们在写逻辑的时候，我们可能会判断某个参数是否为 undefined，如果是则说明调用该函数的时候没有传这个参数，要做下兼容处理；而如果几个参数中，前面的参数是可不传的，后面的参数是需要传的，就需要在该可不传的参数位置传入一个 undefined 占位才行。

(2) 默认参数

在 ES6 标准出来之前，我们的默认参数实现起来比较繁琐：

```
// javascript
var count = 0;
function countUp(step) {
  step = step || 1;
  count += step;
}
```

上面我们定义了一个计数器增值函数，这个函数有一个参数 step，即每次增加的步长，如果不传入参数，那么 step 接受到的就是 undefined，undefined 转换为布尔值是 false，所以`step || 1`这里取了 1，从而达到了不传参数默认 step === 1 的效果。

在 ES6 中，我们定义函数时给参数设默认值就很方便了，直接在参数后面使用等号连接默认值即可：

```
// javascript
const count = 0;
const countUp = (step = 1) => {
  count += step;
};
```

你会发现，可选参数和带默认值的参数在函数调用时都是可以不传实参的，但是区别在于定义函数的时候，可选参数必须放在必选参数后面，而带默认值的参数则可放在必须参数前后都可。

当我们为参数指定了默认参数的时候，TypeScript 会识别默认参数的类型；当我们在调用函数时，如果给这个带默认值的参数传了别的类型的参数则会报错：

```
const add = (x: number, y = 2) => {
  return x + y;
};
add(1, "a"); // error 类型"string"的参数不能赋给类型"number"的参数
```

当然了，你也可以显式地给 y 设置类型：

```
const add = (x: number, y: number = 2) => {
  return x + y;
};
```

(3) 剩余参数

在 JS 中，如果我们定义一个函数，这个函数可以输入任意个数的参数，那么我们就无法在定义参数列表的时候挨个定义。在 ES6 发布之前，我们需要用到 arguments 来获取参数列表。arguments 是每一个函数都包含的一个类数组对象，它包含在函数调用时传入函数的所有实际参数（简称实参），它还包含一个 length 属性，记录参数个数。来看下面的例子，我们来模拟实现函数的重载：

```
// javascript
function handleData() {
  if (arguments.length === 1) return arguments[0] * 2;
  else if (arguments.length === 2) return arguments[0] * arguments[1];
  else return Array.prototype.slice.apply(arguments).join("_");
}
handleData(2); // 4
handleData(2, 3); // 6
handleData(1, 2, 3, 4, 5); // '1_2_3_4_5'
// 这段代码如果在TypeScript环境中，三个对handleData函数的调用都会报错，因为handleData函数定义的时候没有参数。
```

上面这个函数通过判断传入实参的个数，做出不同的处理并返回结果。else 后面的逻辑是如果实参个数不为 1 和 2，那么将这些参数拼接成以"_"连接的字符串。

> 你应该注意到了我们使用`Array.prototype.slice.apply(arguments)`对 arguments 做了处理，前面我们讲过 arguments 不是数组，而是类数组对象，如果直接在 arguments 调用 join 方法，它是没有这个方法的。所以我们通过这个处理得到一个包含 arguments 中所有元素的真实数组。

在 ES6 中，加入了`"..."`拓展运算符，它可以将一个函数或对象进行拆解。它还支持用在函数的参数列表中，用来处理任意数量的参数：

```
const handleData = (arg1, ...args) => {
  // 这里省略逻辑
  console.log(args);
};
handleData(1, 2, 3, 4, 5); // [ 2, 3, 4, 5 ]
```

可以看到，args 是除了 arg1 之外的所有实参的集合，它是一个数组。

> 补充："..."运算符可以拆解数组和对象，比如：arr1 = [1, 2]，arr2 = [3, 4]，那么[...arr1, ...arr2]的结果就是[1, 2, 3, 4]，他还可以用在方法的参数中：如果使用 arr1.push(arr2)，则 arr1 结果是[1, 2, [3, 4]]，如果你想让他们合并成一个函数而不使用 concat 方法，就可以使用 arr1.push(...arr2)。还有对象的使用方法：obj1 = { a: 'aa' }，obj2 = { b: 'bb' }，则{ ...obj1, ...obj2 }的结果是{ a: 'aa', b: 'bb' }。

在 TypeScript 中你可以为剩余参数指定类型，先来看例子：

```
const handleData = (arg1: number, ...args: number[]) => {
  //
};
handleData(1, "a"); // error 类型"string"的参数不能赋给类型"number"的参数
```

2.8.3 函数重载，此重载vs彼重载

在其他一些强类型语言中，函数重载是指定义几个函数名相同，但参数个数或类型不同的函数，在调用时传入不同的参数，编译器会自动调用适合的函数。但是 JavaScript 作为一个动态语言是没有函数重载的，只能我们自己在函数体内通过判断参数的个数、类型来指定不同的处理逻辑。来看个简单的例子：

```
const handleData = value => {
  if (typeof value === "string") {
    return value.split("");
  } else {
    return value
      .toString()
      .split("")
      .join("_");
  }
};
```

这个例子中，当传入的参数为字符串时，将它进行切割，比如传入的是`'abc'`，返回的将是数组`['a', 'b', 'c']`；如果传入的是一个数值类型，则将数字转为字符串然后切割成单个数字然后拼接成字符串，比如传入的是`123`，则返回的是`'1_2_3'`。你可以看到传入的参数类型不同，返回的值的类型是不同的，

在 TypeScript 中有函数重载的概念，但并不是定义几个同名实体函数，然后根据不同的参数个数或类型来自动调用相应的函数。TypeScript的函数重载是在类型系统层面的，是为了更好地进行类型推断。TypeScript的函数重载通过为一个函数指定多个函数类型定义，从而对函数调用的返回值进行检查。来看例子：

```
function handleData(x: string): string[]; // 这个是重载的一部分，指定当参数类型为string时，返回值为string类型的元素构成的数组
function handleData(x: number): string; // 这个也是重载的一部分，指定当参数类型为number时，返回值类型为string
function handleData(x: any): any { // 这个就是重载的内容了，他是实体函数，不算做重载的部分
  if (typeof x === "string") {
    return x.split("");
  } else {
    return x
      .toString()
      .split("")
      .join("_");
  }
}
handleData("abc").join("_");
handleData(123).join("_"); // error 类型"string"上不存在属性"join"
handleData(false); // error 类型"boolean"的参数不能赋给类型"number"的参数。
```

首先我们使用`function`关键字定义了两个同名的函数，但不同的是，函数没有实际的函数体逻辑，而是只定义函数名、参数及参数类型以及函数的返回值类型；而第三个使用`function`定义的同名函数，是一个完整的实体函数，包含函数名、参数及参数类型、返回值类型和函数体；这三个定义组成了一个函数——完整的带有类型定义的函数，前两个`function`定义的就称为*函数重载*，而第三个`function`并不算重载；

然后我们来看下匹配规则，当调用这个函数并且传入参数的时候，会从上而下在函数重载里匹配和这个参数个数和类型匹配的重载。如例子中第一个调用，传入了一个字符串"abc"，它符合第一个重载，所以它的返回值应该是一个字符串组成的数组，数组是可以调用`join`方法的，所以这里没问题；

第二个调用传入的是一个数值类型的123，从上到下匹配重载是符合第二个的，返回值应该是字符串类型。但这里拿到返回值后调用了数组方法`join`，这肯定会报错了，因为字符串无法调用这个方法；

最后调用时传入了一个布尔类型值false，匹配不到重载，所以会报错；

最后还有一点要注意的是，这里重载只能用 function 来定义，不能使用接口、类型别名等。

小结

本小节我们学习了函数类型的三种定义方式：

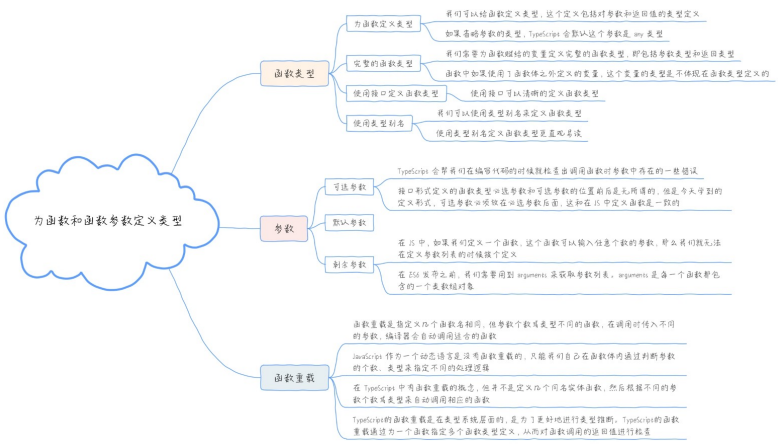
- 基本方式：直接在定义函数实体语句中，指定参数和返回值类型；
- 接口形式：这种方式我们在讲接口的时候已经学习过了；
- 类型别名：这种方式是比较推荐的写法，比较简洁清晰。

我们还详细学习了函数参数的三个知识点：

- 可选参数：可选参数在JavaScript中可以实现，TypeScript中需要在该参数后面加个`?`，且可选参数必须位于必选参数后面；；
- 默认参数：这是在ES6标准中添加的语法，为函数参数指定默认参数，写法就是在参数名后面使用`=`连接默认参数
- 剩余参数：这也是在ES6中添加的语法，可以使用`...参数名`来获取剩余任意多个参数，获取的是一个数组。

最后我们学习了函数重载。着重强调的是，这里的函数重载区别于其他语言中的重载，TypeScript中的重载是为了针对不同参数个数和类型，推断返回值类型。

下个小节我们将学习泛型，来弥补你使用any时丢失的类型校验。



10 接口的高阶用法
12 使用泛型拯救你的any