使用命名空间封装代码

更新时间：2019-07-10 14:36:27

> 理想必须要人们去实现它，它不但需要决心和勇敢而且需要知识。
>
> ——吴玉章

命名空间在 1.5 之前的版本中，是叫做“内部模块”。在 1.5 版本之前，ES6 模块还没正式成为标准，所以 TS 对于模块的实现，是将模块分为“内部模块”和“外部模块”两种。内部模块使用 `module` 来定义，而外部模块使用 `export` 来指定哪个内容对外部可见。

1.5 版本开始，使用“命名空间”代替“内部模块”说法，并且使用 `namespace` 代替原有的 `module` 关键字，而“外部模块”则改为“模块”。

命名空间的作用与使用场景和模块还是有区别的：

- 当我们是在程序内部用于防止全局污染，想把相关的内容都放在一起的时候，使用命名空间；
- 当我们封装了一个工具或者库，要适用于模块系统中引入使用时，适合使用模块。

4.2.1 定义和使用

命名空间的定义实际相当于定义了一个大的对象，里面可以定义变量、接口、类、方法等等，但是如果不使用 `export` 关键字指定此内容要对外可见的话，外部是没法访问到的。来看下怎么写，我们想要把所有涉及到内容验证的方法都放到一起，文件名叫 validation.ts：

```
namespace Validation {
  const isLetterReg = /^[A-Za-z]+$/; // 这里定义一个正则
  export const isNumberReg = /^[0-9]+$/; // 这里再定义一个正则，与isLetterReg的区别在于他使用export导出了
  export const checkLetter = (text: any) => {
    return isLetterReg.test(text);
  };
}
```

我们创建了一个命名空间叫做 Validation，它里面定义了三个内容，两个正则表达式，但是区别在于 isLetterReg 没有使用 export 修饰，而 isNumberReg 使用了 export 修饰。最后一个函数，也是用了 export 修饰。

这里要说明一点的是，命名空间在引入的时候，如果是使用 tsc 命令行编译文件，比如是在*index.ts*文件使用这个命名空间，先直接像下面这样写：

```
/// <reference path="validation.ts"/>
let isLetter = Validation.checkLetter("sdfsd");
const reg = Validation.isNumberReg;
console.log(isLetter);
console.log(reg);
```

来解释下，命名空间如果不是使用 webpack 等工具编译，而是使用 tsc 编译，那只需要在使用外部命名空间的地方使用 /// <reference path="namespace.ts"/> 来引入，注意三斜线 "///" 开头，然后在 path 属性指定相对于当前文件，这个命名空间文件的路径。然后编译时，需要指定一个参数 outFile，这个参数来制定输出的文件名：

```
tsc --outFile src/index.js src/index.ts
```

–outFile 用来指定输出的文件路径和文件名，最后指定要编译的文件。还有一点要注意，使用 outFile 只支持 amd 和 system 两种模块标准，所以需要在*tsconfig.json*里，设置 module 编译选项。

来看下编译后的文件 index.js：

```
var Validation;
(function(Validation) {
  var isLetterReg = /^[A-Za-z]+$/;
  Validation.isNumberReg = /^[0-9]+$/;
  Validation.checkLetter = function(text) {
    return isLetterReg.test(text);
  };
})(Validation || (Validation = {}));
/// <reference path="namespace.ts"/>
var isLetter = Validation.checkLetter("sdfsd");
var reg = Validation.isNumberReg;
console.log(isLetter);
console.log(reg);
```

可以看到，编译后的 JS 文件将命名空间定义的文件 Validation.ts 文件的内容和 index.ts 的内容合并到了最后输出的文件。

如果我们要在 webpack 等工具中开发项目，并时时运行，如果只通过 /// <reference path="Validation.ts"/> 来引入命名空间，你会发现运行起来之后，浏览器控制台会报 **Validation is not defined** 的错误。所以如果是要在项目中时时使用，需要使用 export 将命名空间导出，其实就是作为模块导出，然后在 index.ts 中引入，先来看 Validation.ts 文件：

```
export namespace Validation {
  const isLetterReg = /^[A-Za-z]+$/;
  export const isNumberReg = /^[0-9]+$/;
  export const checkLetter = (text: any) => {
    return isLetterReg.test(text);
  };
}
```

然后在 index.ts 文件中引入并使用：

```
import { Validation } from "./Validation.ts";
let isLetter = Validation.checkLetter("sdfsd");
const reg = Validation.isNumberReg;
console.log(isLetter); // true
console.log(reg); // /^[0-9]+$/
```

> 这里要提醒大家的是，命名空间本来就是防止变量污染，但是模块也能起到这个作用，而且使用模块还可以自己定义引入之后的名字。所以，并不建议导出一个命名空间，这种情况你应该是用模块。

4.2.2 拆分为多个文件

随着内容不断增多，我们可以将同一个命名空间拆成多个文件分开维护，尽管分成了多个文件，但它们仍然是同一个命名空间。下面我们将 Validation.ts 拆开成 LetterValidation.ts 和 NumberValidation.ts：

```
// LetterValidation.ts
namespace Validation {
  export const isLetterReg = /^[A-Za-z]+$/;
  export const checkLetter = (text: any) => {
    return isLetterReg.test(text);
  };
}
// NumberValidation.ts
namespace Validation {
  export const isNumberReg = /^[0-9]+$/;
  export const checkNumber = (text: any) => {
    return isNumberReg.test(text);
  };
}
// index.ts
/// <reference path="./LetterValidation.js"/>
/// <reference path="./NumberValidation.js"/>
let isLetter = Validation.checkLetter("sdfsd");
const reg = Validation.isNumberReg;
console.log(isLetter); // true
```

我们使用命令行来编译一下：

```
tsc --outFile src/index.js src/index.ts
```

最后输出的 index.js 文件是这样的：

```
var Validation;
(function(Validation) {
  Validation.isLetterReg = /^[A-Za-z]+$/;
  Validation.checkLetter = function(text) {
    return Validation.isLetterReg.test(text);
  };
})(Validation || (Validation = {}));
var Validation;
(function(Validation) {
  Validation.isNumberReg = /^[0-9]+$/;
  Validation.checkNumber = function(text) {
    return Validation.isNumberReg.test(text);
  };
})(Validation || (Validation = {}));
/// <reference path="./LetterValidation.ts"/>
/// <reference path="./NumberValidation.ts"/>
var isLetter = Validation.checkLetter("sdfsd");
var reg = Validation.isNumberReg;
console.log(isLetter); // true
```

可以看到，我们使用 reference 引入的两个命名空间都被编译在了一个文件，而且是按照引入的顺序编译的。我们先引入的是 LetterValidation，所以编译后的 js 文件中，LetterValidation 的内容在前面。而且看代码可以看出，两个验证器最后都合并到了一起，所以 Validation 对象有两个正则表达式，两个方法。

4.2.3. 别名

我们使用 import 给常用的对象起一个别名，但是要注意，这个别名和类型别名不是一回事，而且这儿的 import 也只是为了创建别名不是引入模块。来看怎么使用，这是官方文档的原始例子：

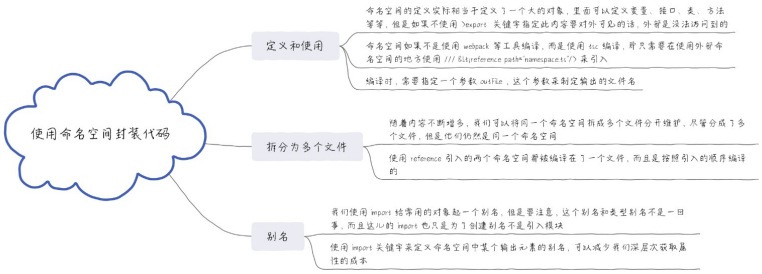
```
namespace Shapes {
  export namespace Polygons {
    export class Triangle {}
    export class Squaire {}
  }
}
import polygons = Shapes.Polygons; // 使用 import 关键字给 Shapes.Polygons 取一个别名polygons
let sq = new polygons.Square();
```

通过这个例子我们可以看到，使用 import 关键字来定义命名空间中某个输出元素的别名，可以减少我们深层次获取属性的成本。

小结

本小节我们学习了如何使用命名空间来封装逻辑相似的代码块，来提高复用性且避免命名污染。我们学习了命名空间的基本定义和使用，使用`namespace`来定义命名空间，形式和定义接口很像，只不过是使用namespace来定义，而且在命名空间内可以定义任何内容。需要提供给外部使用的内容，需要使用`export`关键字指出，外部才能访问到。我们还可以将一个命名空间拆分成多个文件，这种适合逻辑较多，放到同一个文件内内容较多的情况。最后我们还学习了使用别名，来给命名空间内的某个命名空间起一个别名，减少访问多级嵌套的属性的访问成本。最后我们要说的一点是，基本上我们是可以使用模块来代替命名空间的，所以我们应该尽量使用模块来封装代码。

下个小节我们将学习声明合并，编译器会对多个文件中、同一个文件中声明的同名的一些内容进行合并，我们下个小节来进行学习。



使用模块封装代码

对声明合并的爱与恨