TypeScript namespace

namespace 是一种将相关代码组织在一起的方式，中文译为“命名空间”。

它出现在 ES 模块诞生之前，作为 TypeScript 自己的模块格式而发明的。但是，自从有了 ES 模块，官方已经不推荐使用 namespace 了。

基本用法

namespace 用来建立一个容器，内部的所有变量和函数，都必须在这个容器里面使用。

typescript

```
namespace Utils {
  function isString(value: any) {
    return typeof value === "string";
  }

  // 正确
  isString("yes");
}

Utils.isString("no"); // 报错
```

上面示例中，命名空间 `Utils` 里面定义了一个函数 `isString()` ，它只能在 `Utils` 里面使用，如果用于外部就会报错。

如果要在命名空间以外使用内部成员，就必须为该成员加上 `export` 前缀，表示对外输出该成员。

typescript

```
namespace Utility {
  export function log(msg: string) {
    console.log(msg);
  }
  export function error(msg: string) {
```

```
    console.error(msg);
  }
}

Utility.log("Call me");
Utility.error("maybe!");
```

上面示例中，只要加上 `export` 前缀，就可以在命名空间外部使用内部成员。

编译出来的 JavaScript 代码如下。

typescript

```
var Utility;

(function (Utility) {
  function log(msg) {
    console.log(msg);
  }
  Utility.log = log;
  function error(msg) {
    console.error(msg);
  }
  Utility.error = error;
})(Utility || (Utility = {}));
```

上面代码中，命名空间 `Utility` 变成了 JavaScript 的一个对象，凡是 `export` 的内部成员，都成了该对象的属性。

这就是说，namespace 会变成一个值，保留在编译后的代码中。这一点要小心，它不是纯的类型代码。

namespace 内部还可以使用 `import` 命令输入外部成员，相当于为外部成员起别名。当外部成员的名字比较长时，别名能够简化代码。

typescript

```
namespace Utils {
  export function isString(value: any) {
    return typeof value === "string";
  }
}

namespace App {
  import isString = Utils.isString;
```

```
  isString("yes");
  // 等同于
  Utils.isString("yes");
}
```

上面示例中， `import` 命令指定在命名空间 `App` 里面，外部成员 `Utils.isString` 的别名为 `isString` 。

`import` 命令也可以在 namespace 外部，指定别名。

typescript

```
namespace Shapes {
  export namespace Polygons {
    export class Triangle {}
    export class Square {}
  }
}

import polygons = Shapes.Polygons;

// 等同于 new Shapes.Polygons.Square()
let sq = new polygons.Square();
```

上面示例中， `import` 命令在命名空间 `Shapes` 的外部，指定 `Shapes.Polygons` 的别名为 `polygons` 。

namespace 可以嵌套。

typescript

```
namespace Utils {
  export namespace Messaging {
    export function log(msg: string) {
      console.log(msg);
    }
  }
}

Utils.Messaging.log("hello"); // "hello"
```

上面示例中，命名空间 `Utils` 内部还有一个命名空间 `Messaging` 。注意，如果要在外部使用 `Messaging` ，必须在它前面加上 `export` 命令。

使用嵌套的命名空间，必须从最外层开始引用，比如 `Utils.Messaging.log()` 。

namespace 不仅可以包含实义代码，还可以包括类型代码。

typescript

```
namespace N {
  export interface MyInterface {}
  export class MyClass {}
}
```

上面代码中，命令空间 `N` 不仅对外输出类，还对外输出一个接口，它们都可以用作类型。

namespace 与模块的作用是一致的，都是把相关代码组织在一起，对外输出接口。区别是一个文件只能有一个模块，但可以有多个 namespace。由于模块可以取代 namespace，而且是 JavaScript 的标准语法，还不需要编译转换，所以建议总是使用模块，替代 namespace。

如果 namespace 代码放在一个单独的文件里，那么引入这个文件需要使用三斜杠的语法。

typescript

```
/// <reference path = "SomeFileName.ts" />
```

---

namespace 的输出

namespace 本身也可以使用 `export` 命令输出，供其他文件使用。

typescript

```
// shapes.ts
export namespace Shapes {
  export class Triangle {
    // ...
  }
  export class Square {
    // ...
  }
}
```

上面示例是一个文件 `shapes.ts` ，里面使用 `export` 命令，输出了一个命名空间 `Shapes` 。

其他脚本文件使用 `import` 命令，加载这个命名空间。

typescript

```typescript
// 写法一
import { Shapes } from "./shapes";
let t = new Shapes.Triangle();

// 写法二
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle();
```

不过，更好的方法还是建议使用模块，采用模块的输出和输入。

typescript

```typescript
// shapes.ts
export class Triangle {
  /* ... */
}
export class Square {
  /* ... */
}

// shapeConsumer.ts
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

上面示例中，使用模块的输出和输入，改写了前面的例子。

---

namespace 的合并

多个同名的 namespace 会自动合并，这一点跟 interface 一样。

typescript

```typescript
namespace Animals {
  export class Cat {}
}
namespace Animals {
  export interface Legged {
    numberOfLegs: number;
  }
  export class Dog {}
}
```

```
// 等同于
namespace Animals {
  export interface Legged {
    numberOfLegs: number;
  }
  export class Cat {}
  export class Dog {}
}
```

这样做的目的是，如果同名的命名空间分布在不同的文件中，TypeScript 最终会将它们合并在一起。这样就比较方便扩展别人的代码。

合并命名空间时，命名空间中的非 `export` 的成员不会被合并，但是它们只能在各自的命名空间中使用。

typescript

```
namespace N {
  const a = 0;

  export function foo() {
    console.log(a); // 正确
  }
}

namespace N {
  export function bar() {
    foo(); // 正确
    console.log(a); // 报错
  }
}
```

上面示例中，变量 `a` 是第一个名称空间 `N` 的非对外成员，它只在第一个名称空间可用。

命名空间还可以跟同名函数合并，但是要求同名函数必须在命名空间之前声明。这样做是为了确保先创建出一个函数对象，然后同名的命名空间就相当于给这个函数对象添加额外的属性。

typescript

```
function f() {
  return f.version;
}

namespace f {
  export const version = "1.0";
```

```
}

f(); // '1.0'
f.version; // '1.0'
```

上面示例中，函数 `f()` 与命名空间 `f` 合并，相当于命名空间为函数对象 `f` 添加属性。

命名空间也能与同名 class 合并，同样要求 class 必须在命名空间之前声明，原因同上。

typescript

```
class C {
  foo = 1;
}

namespace C {
  export const bar = 2;
}

C.bar; // 2
```

上面示例中，名称空间 `C` 为类 `C` 添加了一个静态属性 `bar` 。

命名空间还能于同名 Enum 合并。

typescript

```
enum E {
  A,
  B,
  C,
}

namespace E {
  export function foo() {
    console.log(E.C);
  }
}

E.foo(); // 2
```

上面示例中，命名空间 `E` 为枚举 `E` 添加了一个 `foo()` 方法。

注意，Enum 成员与命名空间导出成员不允许同名。

typescript

```typescript
enum E {
  A, // 报错
  B,
}

namespace E {
  export function A() {} // 报错
}
```

上面示例中，同名 Enum 与命名空间有同名成员，结果报错。

最后更新: 2023/8/13 15:25

Previous page
模块

Next page
装饰器