声明文件原理：深入探究

组织模块以提供你想要的API形式保持一致是比较难的。
比如，你可能想要这样一个模块，可以用或不用`new`来创建不同的类型，
在不同层级上暴露出不同的命名类型，
且模块对象上还带有一些属性。

阅读这篇指定后，你就会了解如果书写复杂的暴露出友好API的声明文件。
这篇指定针对于模块（UMD）库，因为它们的选择具有更高的可变性。

核心概念

如果你理解了一些关于TypeScript是如何工作的核心概念，
那么你就能够为任何结构书写声明文件。

类型

如果你正在阅读这篇指南，你可能已经大概了解TypeScript里的类型指是什么。
明确一下，*类型*通过以下方式引入：

- 类型别名声明（`type sn = number | string;`）
- 接口声明（`interface I { x: number[]; }`）
- 类声明（`class C { }`）
- 枚举声明（`enum E { A, B, C }`）
- 指向某个类型的`import`声明

以上每种声明形式都会创建一个新的类型名称。

值

与类型相比，你可能已经理解了什么是值。
值是运行时名字，可以在表达式里引用。
比如`let x = 5;`创建一个名为`x`的值。

同样，以下方式能够创建值：

- `let`，`const`，和`var`声明
- 包含值的`namespace`或`module`声明
- `enum`声明
- `class`声明
- 指向值的`import`声明
- `function`声明

命名空间

类型可以存在于*命名空间*里。
比如，有这样的声明`let x: A.B.C`，

我们就认为C类型来自A.B命名空间。

这个区别虽细微但很重要 -- 这里，A.B不是必需的类型或值。

简单的组合：一个名字，多种意义

一个给定的名字A，我们可以找出三种不同的意义：一个类型，一个值或一个命名空间。
要如何去解析这个名字要看它所在的上下文是怎样的。
比如，在声明let m: A.A = A;，
A首先被当做命名空间，然后做为类型名，最后是值。
这些意义最终可能会指向完全不同的声明！

这看上去另人迷惑，但是只要我们不过度的重载这还是很方便的。
下面让我们来看看一些有用的组合行为。

内置组合

眼尖的读者可能会注意到，比如，class同时出现在*类型*和*值*列表里。
class C { }声明创建了两个东西：
*类型*C指向类的实例结构，
*值*C指向类构造函数。
枚举声明拥有相似的行为。

用户组合

假设我们写了模块文件foo.d.ts:

```
export var SomeVar: { a: SomeType };
export interface SomeType {
  count: number;
}
```

这样使用它：

```
import * as foo from './foo';
let x: foo.SomeType = foo.SomeVar.a;
console.log(x.count);
```

这可以很好地工作，但是我们知道SomeType和SomeVar很相关
因此我们想让他们有相同的名字。
我们可以使用组合通过相同的名字Bar表示这两种不同的对象（值和对象）：

```
export var Bar: { a: Bar };
export interface Bar {
  count: number;
}
```

这提供了解构使用的机会：

```
import { Bar } from './foo';
let x: Bar = Bar.a;
console.log(x.count);
```

再次地，这里我们使用Bar做为类型和值。
注意我们没有声明Bar值为Bar类型 -- 它们是独立的。

高级组合

有一些声明能够通过多个声明组合。
比如，class C { }和interface C { }可以同时存在并且都可以做为C类型的属性。

只要不产生冲突就是合法的。
一个普通的规则是值总是会和同名的其它值产生冲突除非它们在不同命名空间里，
类型冲突则发生在使用类型别名声明的情况下（type s = string），
命名空间永远不会发生冲突。

让我们看看如何使用。

利用interface添加

我们可以使用一个interface往别一个interface声明里添加额外成员：

```
interface Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

这同样作用于类：

```
class Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

注意我们不能使用接口往类型别名里添加成员（type s = string;）

使用namespace添加

namespace声明可以用来添加新类型，值和命名空间，只要不出现冲突。

比如，我们可能添加静态成员到一个类：

```
class C {
}
// ... elsewhere ...
namespace C {
  export let x: number;
```

```
}
let y = C.x; // OK
```

注意在这个例子里，我们添加一个值到C的*静态*部分（它的构造函数）。
这里因为我们添加了一个*值*，且其它值的容器是另一个值
（类型包含于命名空间，命名空间包含于另外的命名空间）。

我们还可以给类添加一个命名空间类型：

```
class C {
}
// ... elsewhere ...
namespace C {
  export interface D { }
}
let y: C.D; // OK
```

在这个例子里，直到我们写了namespace声明才有了命名空间C。
做为命名空间的C不会与类创建的值C或类型C相互冲突。

最后，我们可以进行不同的合并通过namespace声明。
Finally, we could perform many different merges using namespace declarations.
This isn't a particularly realistic example, but shows all sorts of interesting behavior:

```
namespace X {
  export interface Y { }
  export class Z { }
}

// ... elsewhere ...
namespace X {
  export var Y: number;
  export namespace Z {
    export class C { }
  }
}
type X = string;
```

在这个例子里，第一个代码块创建了以下名字与含义：

- 一个值X（因为namespace声明包含一个值，Z）
- 一个命名空间X（因为namespace声明包含一个值，Z）
- 在命名空间X里的类型Y
- 在命名空间X里的类型Z（类的实例结构）
- 值X的一个属性值Z（类的构造函数）

第二个代码块创建了以下名字与含义：

- 值Y（number类型），它是值X的一个属性
- 一个命名空间Z
- 值Z，它是值X的一个属性
- 在X.Z命名空间下的类型C
- 值X.Z的一个属性值C
- 类型X

使用`export =`或`import`

一个重要的原则是`export`和`import`声明会导出或导入目标的*所有含义*。