介绍

TypeScript中有些独特的概念可以在类型层面上描述JavaScript对象的模型。
这其中尤其独特的一个例子是“声明合并”的概念。
理解了这个概念，将有助于操作现有的JavaScript代码。
同时，也会有助于理解更多高级抽象的概念。

对本文件来讲，“声明合并”是指编译器将针对同一个名字的两个独立声明合并为单一声明。
合并后的声明同时拥有原先两个声明的特性。
任何数量的声明都可被合并；不局限于两个声明。

基础概念

TypeScript中的声明会创建以下三种实体之一：命名空间，类型或值。
创建命名空间的声明会新建一个命名空间，它包含了用（.）符号来访问时使用的名字。
创建类型的声明是：用声明的模型创建一个类型并绑定到给定的名字上。
最后，创建值的声明会创建在JavaScript输出中看到的值。

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

理解每个声明创建了什么，有助于理解当声明合并时有哪些东西被合并了。

合并接口

最简单也最常见的声明合并类型是接口合并。
从根本上说，合并的机制是把双方的成员放到一个同名的接口里。

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

let box: Box = {height: 5, width: 6, scale: 10};
```

接口的非函数的成员必须是唯一的。
如果两个接口中同时声明了同名的非函数成员编译器则会报错。

对于函数成员，每个同名函数声明都会被当成这个函数的一个重载。
同时需要注意，当接口`A`与后来的接口`A`合并时，后面的接口具有更高的优先级。

如下例所示：

```
interface Cloner {
    clone(animal: Animal): Animal;
}

interface Cloner {
    clone(animal: Sheep): Sheep;
}

interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
```

这三个接口合并成一个声明：

```
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
```

注意每组接口里的声明顺序保持不变，但各组接口之间的顺序是后来的接口重载出现在靠前位置。

这个规则有一个例外是当出现特殊的函数签名时。
如果签名里有一个参数的类型是*单*一的字符串字面量（比如，不是字符串字面量的联合类型），那么它将会被提升到重载列表的最顶端。

比如，下面的接口会合并到一起：

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

合并后的`Document`将会像下面这样：

```
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
```

```
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

合并命名空间

与接口相似，同名的命名空间也会合并其成员。
命名空间会创建出命名空间和值，我们需要知道这两者都是怎么合并的。

对于命名空间的合并，模块导出的同名接口进行合并，构成单一命名空间内含合并后的接口。

对于命名空间里值的合并，如果当前已经存在给定名字的命名空间，那么后来的命名空间的导出成员会被加到已经存在的那个模块里。

`Animals`声明合并示例：

```
namespace Animals {
    export class Zebra { }
}

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

等同于：

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

除了这些合并外，你还需要了解非导出成员是如何处理的。
非导出成员仅在其原有的（合并前的）命名空间内可见。这就是说合并之后，从其它命名空间合并进来的成员无法访问非导出成员。

下例提供了更清晰的说明：

```
namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // <-- error, haveMuscles is not visible here
    }
}
```

因为haveMuscles并没有导出，只有animalsHaveMuscles函数共享了原始未合并的命名空间可以访问这个变量。
doAnimalsHaveMuscles函数虽是合并命名空间的一部分，但是访问不了未导出的成员。

命名空间与类和函数和枚举类型合并

命名空间可以与其它类型的声明进行合并。
只要命名空间的定义符合将要合并类型的定义。合并结果包含两者的声明类型。
TypeScript使用这个功能去实现一些JavaScript里的设计模式。

合并命名空间和类

这让我们可以表示内部类。

```
class Album {
    label: Album.AlbumLabel;
}
namespace Album {
    export class AlbumLabel { }
}
```

合并规则与上面合并命名空间小节里讲的规则一致，我们必须导出AlbumLabel类，好让合并的类能访问。
合并结果是一个类并带有一个内部类。
你也可以使用命名空间为类增加一些静态属性。

除了内部类的模式，你在JavaScript里，创建一个函数稍后扩展它增加一些属性也是很常见的。
TypeScript使用声明合并来达到这个目的并保证类型安全。

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
    export let suffix = "";
    export let prefix = "Hello, ";
}

alert(buildLabel("Sam Smith"));
```

相似的，命名空间可以用来扩展枚举型：

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
```

```
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

非法的合并

TypeScript并非允许所有的合并。
目前，类不能与其它类或变量合并。
想要了解如何模仿类的合并，请参考TypeScript的混入。

模块扩展

虽然JavaScript不支持合并，但你可以为导入的对象打补丁以更新它们。让我们考察一下这个玩具性的示例：

```
// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

它也可以很好地工作在TypeScript中， 但编译器对 `Observable.prototype.map`一无所知。
你可以使用扩展模块来将它告诉编译器：

```
// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}


// consumer.ts
import { Observable } from "./observable";
import "./map";
```

```
let o: Observable<number>;
o.map(x => x.toFixed());
```

模块名的解析和用`import/export`解析模块标识符的方式是一致的。
更多信息请参考 Modules。
当这些声明在扩展中合并时，就好像在原始位置被声明了一样。
但是，你不能在扩展中声明新的顶级声明－仅可以扩展模块中已经存在的声明。

全局扩展

你也以在模块内部添加声明到全局作用域中。

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

全局扩展与模块扩展的行为和限制是相同的。