概述

一般来讲，你*组织*声明文件的方式取决于库是如何被使用的。
在JavaScript中一个库有很多使用方式，这就需要你书写声明文件去匹配它们。
这篇指南涵盖了如何识别常见库的模式，和怎样书写符合相应模式的声明文件。

针对每种主要的库的组织模式，在模版一节都有对应的文件。
你可以利用它们帮助你快速上手。

识别库的类型

首先，我们先看一下TypeScript声明文件能够表示的库的类型。
这里会简单展示每种类型的库的使用方式，如何去书写，还有一些真实案例。

识别库的类型是书写声明文件的第一步。
我们将会给出一些提示，关于怎样通过库的*使用方法*及其*源码*来识别库的类型。
根据库的文档及组织结构不同，这两种方式可能一个会比另外的那个简单一些。
我们推荐你使用任意你喜欢的方式。

全局库

*全局*库是指能在全局命名空间下访问的（例如：不需要使用任何形式的`import`）。
许多库都是简单的暴露出一个或多个全局变量。
比如，如果你使用过jQuery，`$`变量可以被够简单的引用：

```
$(() => { console.log('hello!'); } );
```

你经常会在全局库的指南文档上看到如何在HTML里用脚本标签引用库：

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

目前，大多数流行的全局访问型库实际上都以UMD库的形式进行书写（见后文）。
UMD库的文档很难与全局库文档两者之间难以区分。
在书写全局声明文件前，一定要确认一下库是否真的不是UMD。

从代码上识别全局库

全局库的代码通常都十分简单。
一个全局的“Hello, world”库可能是这样的：

```
function createGreeting(s) {
    return "Hello, " + s;
}
```

或这样：

```
window.createGreeting = function(s) {
    return "Hello, " + s;
```

```
}
```

当你查看全局库的源代码时，你通常会看到：

- 顶级的`var`语句或`function`声明
- 一个或多个赋值语句到`window.someName`
- 假设DOM原始值像`document`或`window`是存在的

你*不会*看到：

- 检查是否使用或如何使用模块加载器，比如`require`或`define`
- CommonJS/Node.js风格的导入如`var fs = require("fs");`
- `define(...)`调用
- 文档里说明了如果`require`或导入这个库

全局库的例子

由于把一个全局库转变成UMD库是非常容易的，所以很少流行的库还再使用全局的风格。
然而，小型的且需要DOM（或*没有*依赖）的库可能还是全局类型的。

全局库模版

模版文件`global.d.ts`定义了`myLib`库作为例子。
一定要阅读"防止命名冲突"补充说明。

模块化库

一些库只能工作在模块加载器的环境下。
比如，像`express`只能在Node.js里工作所以必须使用CommonJS的`require`函数加载。

ECMAScript 2015（也就是ES2015，ECMAScript 6或ES6），CommonJS和RequireJS具有相似的*导入*一个*模块*的表示方法。
例如，对于JavaScript CommonJS （Node.js），有下面的代码

```
var fs = require("fs");
```

对于TypeScript或ES6，`import`关键字也具有相同的作用：

```
import fs = require("fs");
```

你通常会在模块化库的文档里看到如下说明：

```
var someLib = require('someLib');
```

或

```
define(..., ['someLib'], function(someLib) {

});
```

与全局模块一样，你也可能会在UMD模块的文档里看到这些例子，因此要仔细查看源码

和文档。

从代码上识别模块化库

模块库至少会包含下列具有代表性的条目之一：

- 无条件的调用`require`或`define`
- 像`import * as a from 'b';` or `export c;`这样的声明
- 赋值给`exports`或`module.exports`

它们极少包含：

- 对`window`或`global`的赋值

模块化库的例子

许多流行的Node.js库都是这种模块化的，例如express，gulp和request。

UMD

*UMD*模块是指那些既可以作为模块使用（通过导入）又可以作为全局（在没有模块加载器的环境里）使用的模块。
许多流行的库，比如Moment.js，就是这样的形式。
比如，在Node.js或RequireJS里，你可以这样写：

```
import moment = require("moment");
console.log(moment.format());
```

然而在纯净的浏览器环境里你也可以这样写：

```
console.log(moment.format());
```

识别UMD库

UMD模块会检查是否存在模块加载器环境。
这是非常形容观察到的模块，它们会像下面这样：

```
(function (root, factory) {
    if (typeof define === "function" && define.amd) {
        define(["libName"], factory);
    } else if (typeof module === "object" && module.exports) {
        module.exports = factory(require("libName"));
    } else {
        root.returnExports = factory(root.libName);
    }
}(this, function (b) {
```

如果你在库的源码里看到了`typeof define`，`typeof window`，或`typeof module`这样的测试，尤其是在文件的顶端，那么它几乎就是一个UMD库。

UMD库的文档里经常会包含通过`require`“在Node.js里使用”例子，
和“在浏览器里使用”的例子，展示如何使用`<script>`标签去加载脚本。

UMD库的例子

大多数流行的库现在都能够被当成UMD包。
比如jQuery,Moment.js,lodash和许多其它的。

模版

针对模块有三种可用的模块，
`module.d.ts`, `module-class.d.ts` and `module-function.d.ts`.

使用`module-function.d.ts`，如果模块能够作为函数*调用*。

```
var x = require("foo");
// Note: calling 'x' as a function
var y = x(42);
```

一定要阅读补充说明：“ES6模块调用签名的影响”

使用`module-class.d.ts`如果模块能够使用`new`来*构造*：

```
var x = require("bar");
// Note: using 'new' operator on the imported variable
var y = new x("hello");
```

相同的补充说明作用于这些模块。

如果模块不能被调用或构造，使用`module.d.ts`文件。

模块插件或UMD插件

一个*模块插件*可以改变一个模块的结构（UMD或模块）。
例如，在Moment.js里，`moment-range`添加了新的`range`方法到`monent`对象。

对于声明文件的目标，我们会写相同的代码不论被改变的模块是一个纯粹的模块还是UMD模块。

模版

使用`module-plugin.d.ts`模版。

全局插件

一个*全局插件*是全局代码，它们会改变全局对象的结构。
对于*全局修改的模块*，在运行时存在冲突的可能。

比如，一些库往`Array.prototype`或`String.prototype`里添加新的方法。

识别全局插件

全局通常很容易地从它们的文档识别出来。

你会看到像下面这样的例子：

```
var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

模版

使用global-plugin.d.ts模版。

全局修改的模块

当一个*全局修改的模块*被导入的时候，它们会改变全局作用域里的值。
比如，存在一些库它们添加新的成员到`String.prototype`当导入它们的时候。
这种模式很危险，因为可能造成运行时的冲突，
但是我们仍然可以为它们书写声明文件。

识别全局修改的模块

全局修改的模块通常可以很容易地从它们的文档识别出来。
通常来讲，它们与全局插件相似，但是需要`require`调用来激活它们的效果。

你可能会看到像下面这样的文档:

```
// 'require' call that doesn't use its return value
var unused = require("magic-string-time");
/* or */
require("magic-string-time");

var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

模版

使用global-modifying-module.d.ts模版。

使用依赖

可能会有以下几种依赖。

依赖全局库

如果你的库依赖于某个全局库，使用`/// <reference types="..." />`指令：

```
/// <reference types="someLib" />

function getThing(): someLib.thing;
```

依赖模块

如果你的库依赖于模块，使用`import`语句：

```
import * as moment from "moment";

function getThing(): moment;
```

依赖UMD库

从全局库

如果你的全局库依赖于某个UMD模块，使用`/// <reference types`指令：

```
/// <reference types="moment" />

function getThing(): moment;
```

从一个模块或UMD库

如果你的模块或UMD库依赖于一个UMD库，使用`import`语句：

```
import * as someLib from 'someLib';
```

*不要*使用`/// <reference`指令去声明UMD库的依赖！

补充说明

防止命名冲突

注意，在书写全局声明文件时，允许在全局作用域里定义很多类型。
我们十分不建义这样做，当一个工程里有许多声明文件时，它会导致无法处理的命名冲突。

一个简单的规则是使用库定义的全局变量名来声明命名空间类型。
比如，库定义了一个全局的值`cats`，你可以这样写

```
declare namespace cats {
    interface KittySettings { }
}
```

不要

```
// at top-level
interface CatsKittySettings { }
```

这样也保证了库在转换成UMD的时候没有任何的破坏式改变，对于声明文件用户来说。

ES6模块插件的影响

一些插件添加或修改已存在的顶层模块的导出部分。
当然这在CommonJS和其它加载器里是允许的，ES模块被当作是不可改变的因此这种模式就不可行了。
因为TypeScript是能不预知加载器类型的，所以没没在编译时保证，但是开发者如果要转到ES6模块加载器上应该注意这一点。

ES6模块调用签名的影响

很多流行库，比如Express，暴露出自己作为可以调用的函数。
比如，典型的Express使用方法如下：

```
import exp = require("express");
var app = exp();
```

在ES6模块加载器里，顶层的对象（这里以`exp`导入）只能具有属性；
顶层的模块对象*永远不能*被调用。
十分常见的解决方法是定义一个`default`导出到一个可调用的/可构造的对象；
一会模块加载器助手工具能够自己探测到这种情况并且使用`default`导出来替换顶层对象。