[介绍 2](#_Toc366686092)

[模块 2](#_Toc366686093)

[引用库 2](#_Toc366686094)

[实际采用的方案 3](#_Toc366686095)

[网上资料 3](#_Toc366686096)

[Modularization in TypeScript 3](#_Toc366686097)

[Declare 9](#_Toc366686098)

[类 9](#_Toc366686099)

[接口 9](#_Toc366686100)

[泛型 9](#_Toc366686101)

[调试 9](#_Toc366686102)

[其它 10](#_Toc366686103)

[Enum枚举 10](#_Toc366686104)

[未来的支持 10](#_Toc366686105)

[优势 10](#_Toc366686106)

[注意事项 11](#_Toc366686107)

[存在的问题 11](#_Toc366686108)

[1. 继承树上，不同的类的成员名要相互干扰（如私有成员名不能一样！） 11](#_Toc366686109)

[问题描述 11](#_Toc366686110)

[原因分析 11](#_Toc366686111)

[解决方案 12](#_Toc366686112)

[IDE对typescript的支持 12](#_Toc366686113)

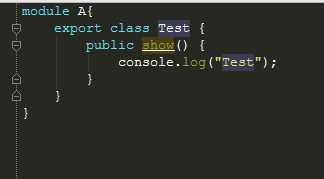
## 介绍

## 模块

### 引用库

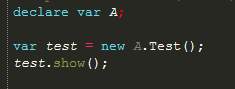
在a.ts中创建了模块（命名空间）A，在A中有一个供外部调用的类Test

a.ts：



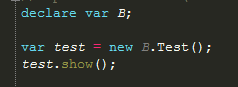
现在想在b.ts中引用a.ts的模块A，有以下几种方式：

* 使用declare，申明模块A



使用declare可以很方便地引用外部模块，不过tsc编译器不会检查申明的外部模块是否存在！也不会有智能提示

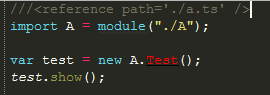
如这样写也不会报错（B模块并不存在，因为在a.ts中创建的是A模块）：



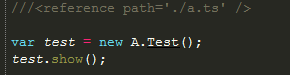
* 使用///<reference path='ts文件的路径' />来导入ts

在文件开头导入js，然后可以用import加载文件，也可以直接使用（import貌似没有加载起文件！效果与直接使用一样！）

使用import：



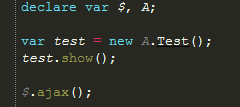
直接使用：



#### 实际采用的方案

使用declare申明。

如引用a.ts的模块A和jquery：



在ts中引用另一个ts申明的模块

### 网上资料

### [Modularization in TypeScript](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

In **TypeScript**, there are two ways to “include” another source file – **reference comments** and **import declarations**. Let’s have a look at each.

Reference comments

Reference comments add a dependency on the source file specified. They are only used for compilation purposes, and can be used to provide IntelliSense for JavaScript files. They do NOT affect the compiled JS output. Here is an example:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

*/// <reference path="pony.ts"/>*

var pony : Pony; *//compiler now recognizes Pony class.*

This tells the compiler that pony.ts will be **available at runtime**. It does **not** actually **import** the code. This can be used if you are not using AMD or CommonJS and just have files included on the page via script tags.

Import declarations

If we want to **load** a file via AMD or CommonJS, we need to use an **import** declaration.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

**import** myModule = module("pony");

This tells the compiler to load pony.ts via AMD or CommonJS. It **does** affect the output of the compiler.

Importing JavaScript

Let’s face it. Most of the libraries we will want to use are not written in **TypeScript** – they’re all in vanilla JavaScript. To use them in **TypeScript**, we’re going to have to do a little bit of porting. Ambient Declarations and Declaration Source Files will be our tools.

Ambient Declarations (declare var)

Ambient Declarations are used to define variables which will be available in JavaScript at runtime, but may not have originated as **TypeScript** files. This is done with the declare keyword.

As a simple example of how this could be used, let’s say your program is running in the browser, and you want to use the document variable. **TypeScript** doesn’t know that this variable exists, and if we start throwing document’saround, he’ll throw a nice compiler error. So we have to tell him, like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

declare var document;

Simple enough, right? Since no typing information is associated with document, **TypeScript** will **infer** the any type, and won’t make any assumptions about the contents of document. But what if we want to have some typing information associated with a library we are porting in? Read on.

Declaration source files (\*.d.ts)

Declaration source files are files with a special extension of \*.d.ts. Inside these files, the declare keyword is implicit on all declarations. The purpose of these files is to provide some typing information for JavaScript libraries. For a simple example, let’s say we have an amazing AMD JavaScript utility library, util.js which we just **have** to have in our**TypeScript** project.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

define([], function() {

return {

sayHello: function(name) {

alert( "Hello, " + name );

}

});

If we wanted to write a **TypeScript** declaration file for it, we would write something like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

export function sayHello(name:string): void;

Declaration source files stand in for the actual .js files in **TypeScript**-land. They do not compile to .js files, unlike their plain \*.ts peers. One way you can think of it is \*.d.ts files act as surrogates for their .js implementations, since plain .js files aren’t allowed in **TypeScript**-land. They simply **describe** their JavaScript implementations, and act as their representative. What this means is that now you can**import JavaScript**! Here’s how we would use our util library:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

**import** util = module("util");

util.sayHello("Dave");

This ***import*** statement here uses AMD or CommonJS to load the util.js file, the same as if util.js had been the compiled output of a util.ts file. Our util.d.ts provides the compiler/IDE with the IntelliSense to know that a sayHellomethod exists. The big takeaway here: **If you want to include a JavaScript file, you need to write a \*.d.ts file.**

Note here that we used an ***Import***Declaration to include our .js file. If we had merely wanted to tell the **TypeScript**compiler that util would be available at runtime (meaning we already loaded it somewhere else, via script tag or RequireJS), we could have used a Reference Comment in conjunction with an Ambient Declaration. To do that, we would first need to change our declaration source file:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

interface Util {

sayHello(name:string): void;

}

This defines a Util interface which we can then use for compilation purposes.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

*///<reference path="util.d.ts"/>*

declare var util: Util;

util.sayHello("Dave");

It turns out that lots of people in the open source community have been cranking out **TypeScript** interface definitions for some of the most popular JavaScript libraries, including jQuery, Backbone, Underscore, etc. You can find dozens of these on GitHub in the[DefinitelyTyped](https://github.com/borisyankov/DefinitelyTyped) project. The interface definition for jQuery can be found[here.](http://typescript.codeplex.com/SourceControl/changeset/view/92d9e637f6e1#typings/jquery.d.ts)

How to be lazy

What if I don’t want to take the time to re-write my .js library in **TypeScript**, or carefully craft an exhaustive \*.d.ts file? It’s not hard to get around. Back to our util.js example. Let’s say that we had another method, sayGoodbye, which we didn’t want to take the time to define in our util.d.ts file, because of our supreme laziness. Simple. Just define a singleexport function in util.d.ts (one line isn’t going to hurt you!). Then, when you want to use methods that **TypeScript**doesn’t know exist, just use an Ambient Declaration, like so:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

**import** util = module("util");

declare var unchained:any;

unchained = util;

unchained.sayGoodbye();

The magic here is the Ambient declaration of the unchained variable, which is of type any. The any type tells**TypeScript** not to worry about typing – this variable could be anything. Trust us.

Dynamically Importing Existing JavaScript Libraries – The Problem(s)

The trickiest part of all this is getting **TypeScript** to use AMD or CommonJS to **import** a JavaScript library or module, rather than just making it compile using a reference comment. There are two tricky components to this.

First, if you are using an interface definition like those found online, you can NOT use that file as your \*.d.ts in an***import*** declaration. The interface declaration is only used to provide compiler/Intellisense information to the**TypeScript** compiler – interfaces are different than classes. In order for you to **import** your external library like we did in our simple util.js example from earlier, you need a different sort of\*.d.ts file – one which uses the export keyword.

Second, recall from earlier how **TypeScript** has what I like to call Implicit Modules when **import**ing files? You can’t directly **import** a class – you **import** the file it is in and then get your class definition off the resulting module object. Well, this causes us some grief when it comes to **import**ing JavaScript modules which don’t follow the exports.\*pattern in their AMD implementation, since most of the time when you **import** a “class” (constructor function) in AMD, you expect the **import**ed object to be the constructor function itself.

Dynamic Solution

So how do we use **TypeScript** to dynamically **import** JavaScript libraries? To get around the first problem mentioned above, what I like to do is to define two separate \*.d.ts files: one containing the interface definition you probably pulled off the web, and another which exports a single variable of the type defined in the interface file. Let’s use jQuery as our example again. The [jquery.d.ts](http://typescript.codeplex.com/SourceControl/changeset/view/92d9e637f6e1#typings/jquery.d.ts) definition defines a JQueryStatic interface. Lets rename our interface definition file to jquery-int.d.ts, and create a new jquery.d.ts that looks like this:

jquery.d.ts

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

*///<reference path="jquery-int.d.ts"/>*

export var $:JQueryStatic;

This will allow **TypeScript** to compile if we **import** jQuery like below.

app.ts

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

**import** JQuery = module( "libs/jquery" );

var $:JQueryStatic = JQuery.$;

export class App {

start() {

$( "#content" ).html( "<h1>Hello World</h1>" );

}

}

Now we are able to compile in **TypeScript**. However, let’s say that we have a fairly standard AMD-compliant loader file for jQuery, which might look something like this:

jquery.js

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

define( ["libs/jquery-1.8.3.min"], function() {

return $;

});

Using an AMD “loader” file like this is a common way of modularizing non-AMD JavaScript libraries. The problem here though is although ourjquery.js loader returns $ when **import**ed, in **TypeScript** our **import** statement expects an object that has a **property** of $. This is the second problem I mentioned earlier. My workaround for this is to change my AMD loader file to make it use exports.\* just like **TypeScript** does.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/528295/ModularizationplusinplusTypeScript)

define(["exports", "libs/jquery-1.8.3.min"], function (exports) {

exports.$ = $;

});

Now when we **import** our jquery.d.ts in **TypeScript**, we will have the results we expected: a nice exported module with a $ property that happens to conform to our JQueryStatic definition. After weeks of scouring the web, this is the best method that I have come up with for dynamically **import**ing JavaScript libraries in **TypeScript**. Let’s review the steps:

Dynamic Strategy Summary

1. Snag an interface definition (\*.d.ts file) off the web, or create one yourself. Remember that you can always be lazy and fall back on the any type.
2. Rename this interface definition file example-int.d.ts.
3. Create a new example.d.ts file which exports a single variable of the type defined in your interface definition
4. Create a “loader”-style file example.js and use exports.\* to export the desired library.
5. Where desired, simply **import** the example module and find your desired library as a property off of the**import**ed module.

And that’s it! It is a bit involved and definitely more work that I would have liked. I am hoping that the maintainers of the **TypeScript** language someday soon add the ability to use module.exports like I mentioned earlier, but until then this sort of workaround seems to be the order of the day.

### Declare

The **declare** keyword is used for ambient declarations where you want to define a variable that may not have originated from a TypeScript file.

For example, lets imagine that we have a library called **myLibrary** that doesn’t have a TypeScript declaration file and have a namespace called **myLibrary** in the global namespace. If you want to use that library in your TypeScript code, you can use the following code:

**declare** **var** myLibrary;

The type that the TypeScript runtime will give to **myLibrary** variable is the **any** type. The problem here is that you won’t have Intellisense for that variable in design time but you will be able to use the library in your code. Another option to have the same behavior without using the **declare** keyword is just using a variable with the **any** type:

**var** myLibrary: any;

Both of the code examples will result in the same JavaScript output but the **declare**example is more readable and expresses an ambient declaration.

Summary

The TypeScript declare keyword is used to declare variables that may not have originated from a TypeScript file.

## 类

## 接口

## 泛型

## 函数

### 重载

可以在申明时重载，但必须给出参数判断的实现

According to the specification, TypeScript does support method overloading, but it's quite awkward and includes a lot of manual work checking types of parameters. I think it's mostly because the closest you can get to method overloading in plain JavaScript includes that checking too and TypeScript tries to not modify actual method bodies to avoid any unnecessary runtime performance cost.

If I understand it correctly, you have to first write a method declaration for each of the overloads and then one method implementation that checks its arguments to decide which overload was called. The signature of the implementation has to be compatible with all of the overloads.

示例：

class TestClass {

private someMethod\_Overload\_string(stringParameter: string): void {

// A lot of code could be here... I don't want to mix it with switch or if statement in general function

alert("Variant #1: stringParameter = " + stringParameter);

}

private someMethod\_Overload\_number\_string(numberParameter: number, stringParameter: string): void {

alert("Variant #2: numberParameter = " + numberParameter + ", stringParameter = " + stringParameter);

}

private someMethod\_Overload\_string\_number(stringParameter: string, numberParameter: number): void {

alert("Variant #3: stringParameter = " + stringParameter + ", numberParameter = " + numberParameter);

}

public someMethod(stringParameter: string): void;

public someMethod(numberParameter: number, stringParameter: string): void;

public someMethod(stringParameter: string, numberParameter: number): void;

public someMethod(): void {

switch (arguments.length) {

case 1:

if(typeof arguments[0] == "string") {

this.someMethod\_Overload\_string(arguments[0]);

return;

}

return; // Unreachable area for this case, unnecessary return statement

case 2:

if ((typeof arguments[0] == "number") &&

(typeof arguments[1] == "string")) {

this.someMethod\_Overload\_number\_string(arguments[0], arguments[1]);

}

else if ((typeof arguments[0] == "string") &&

(typeof arguments[1] == "number")) {

this.someMethod\_Overload\_string\_number(arguments[0], arguments[1]);

}

return; // Unreachable area for this case, unnecessary return statement

}

}

}

## 调试

目前我在webstorm中直接调试ts失败！

因此，只能在js中调试！

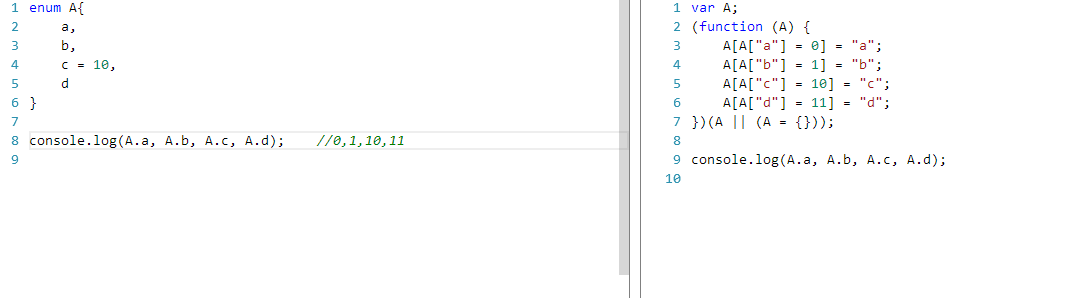
或许可试下在vs2012下的ts调试！

## 其它

### Enum枚举

枚举类型值默认从0开始，按1递增。

左边为ts文件，右边为编译后的js文件：



此处编译后的js文件貌似不是最优的，应该形如：A[“a”] = 0才是最优的吧！！！

## 未来的支持

* Protected关键字
* Abstract抽象类、抽象方法

## 优势

* 部分支持jsDoc，但是还是有问题（不能生产jsDoc完整结构，如@param等；只能生成/\*\* \*/）
* 编译器会去除Ts文件上的注释
* 我测试了下，TypeScript没有YOOP解决的4个问题
* 代码维护性好，语法更符合oo
* 有访问权限控制，public和private成员位置不固定（而不像YOOP中，需要将private成员写到Private中，public成员写到Public中）
* 自动翻译成js代码，部署时只部署翻译后的js代码。而不像YOOP在部署时还要部署YOOP.js，且需要在执行js时翻译一次（将YOOP的class在执行层面上翻译为js的prototype形式）

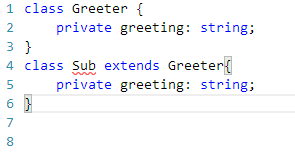
## 注意事项

* 在编译后，属性并不会加到原型上，而是直接使用this来引用；而方法会直接加到原型上。

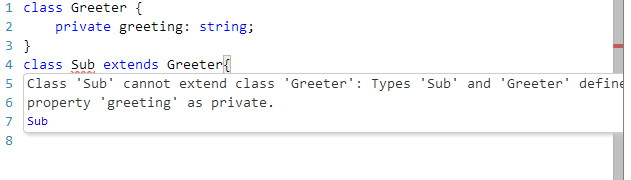
## 存在的问题

### 继承树上，不同的类的成员名要相互干扰（如私有成员名不能一样！）

#### 问题描述

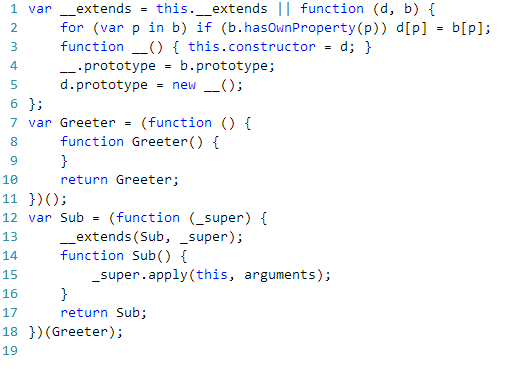


如Sub的私有成员与Greeter私有成员名相同，在编译时会报错：



#### 原因分析

编译后的js代码：



因为对于访问权限，只是在编译器层面检查，而在转换为js代码后，还是与js本身的实现方式一致（此处类的成员都是添加到prototype上（方法），或通过this添加的（属性））。

也就是说，这是受到js语言本身的限制！

#### 解决方案

采用与YOOP相同的策略，使用命名规范来解决这个问题。

如第一级类的私有成员加“\_”前缀；

第二季类的私有成员加“\_\_”前缀；

以此类推！

## IDE对typescript的支持