TypeScript 教程入门指南详解案例教程

一、简介

==========

==========Author:jilongliang==========
==========Date:2015/7/27 编著==========
==========Eamil:jilongliang@sina.com==========

1.1Typescript 诞生

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。安德斯·海尔斯伯格，C#的首席架构师，已工作于 TypeScript 的开发。2012 年十月份，微软发布了首个公开版本的 TypeScript，2013 年 6 月 19 日，在经历了一个预览版之后微软正式 发布了正式版 TypeScript 0.9，向未来的 TypeScript 1.0 版迈进了很大一步

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于 类的面向对象编程。安德斯·海尔斯伯格，C#的首席架构师，已工作于 TypeScript 的开发。

TypeScript 扩展了 JavaScript 的句法，所以任何现有的 JavaScript 程序可以不加改变的在 TypeScript 下工作。TypeScript 是为大型应用之开发而设计，而编译时它产生 JavaScript 以确保兼容性。

TypeScript 支持为已存在的 JavaScript 库添加类型信息的头文件，扩展了它对于流行的库如 jQuery，MongoDB，Node.js 和 D3.js 的好处

1.2Typescript 前景

TypeScript 起源于开发应用程序规模的 JavaScript 应用程序的需求。Microsoft 的语言开发者们说内部以及外部的客户都表示他们构建 JavaScript 代码的问题。

很多最终依赖于 JavaScript 的开发者通常用编译为 JavaScript 代码的另一种语言写脚本，例如 CoffeeScript 和 Script# (读作 ScriptSharp)。一个明显的劣势是也许无法从那另一种语言使用任何 JavaScript 的具体的语言特性，如果那种语言不支持它的话。

在 Microsoft 内部，它导致了自定义工具以简化 JavaScript 组件的编写的需求。

1.3Typescript 特性

TypeScript 是一种给 JavaScript 添加特性的语言扩展。

- 类型批注和编译时类型检查
- 类
- 接口
- 模块 [6]

●lambda 函数

语法上，TypeScript 很类似于 JScript.NET，另外一个添加了对静态类型，经典的面向对象语言特性如类，继承，接口和命名空间等的支持的 Microsoft 对 ECMA-262 语言标准的实现。

类型批注

TypeScript 通过类型批注提供静态类型以在编译时启动类型检查。这是可选的，而且可以被忽略而使用 JavaScript 常规的动态类型。

对于基本类型的批注是 number，bool 和 string。而弱或动态类型的结构则是 any 类型。

类型批注可以被导出到一个单独的声明文件以让使用类型的已被编译为 JavaScript 的 TypeScript 脚本的类型信息可用。批注可以为一个现有的 JavaScript 库声明，就像已经为 Node.js 和 jQuery 所做的那样。

当类型没有给出时，TypeScript 编译器利用类型推断以推断类型。如果由于缺乏声明，没有类型可以被推断出，那么它就会默认为是动态的 any 类型。

声明文件

当一个 TypeScript 脚本被编译时，有一个产生作为编译后的 JavaScript 的组件的一个接口而起作用的声明文件（具有扩展名 .d.ts）的选项。在这个过程中编译器基本上带走所有的函数和方法体而仅保留所导出类型的批注。当第三方开发者从 TypeScript 中使用它时，由此产生的声明文件就可以被用于描述一个 JavaScript 库或模块导出的虚拟的 TypeScript 类型。

声明文件的概念类似于 C/C++ 中头文件的概念。

类型声明文件可以为已存在的 JavaScript 库手写，就像为 jQuery 和 Node.js 所做的那样。

对 ECMAScript 6 的支持

TypeScript 增加了对为即将到来的 ECMAScript 6 标准所建议的特性的支持。

类（以及继承）模块 Arrow functions

尽管标准还未准备就绪，Microsoft 说它的目标是使 TypeScript 的特性与建议的标准看齐。

类

TypeScript 支持集成了可选的类型批注支持的 ECMAScript 6 的类。

泛型

这种语言的规范说明一个未来的版本将会支持基于类型擦除的泛型编程。

与 JavaScript 的兼容性

TypeScript 是 JavaScript 的一个超集。默认情况下编译器以 ECMAScript 3 (ES3) 为目标但 ES5 也是受支持的一个选项。一个 TypeScript 应用可以利用已存在的 JavaScript 脚本。编译后的 TypeScript 脚本也可以从 JavaScript 中使用。

现有框架如 jQuery 和 Node.js 等受到完全支持。这些库的类型声明在源代码中提供。

支持的浏览器和平台

运行于任何平台上的任何网页浏览器都可以运行 TypeScript 由于它仅仅是被编译为标准的 JavaScript。一个脚本既可以被预编译为 JavaScript 也可以通过为 TypeScript 包含 JavaScript 编译器实时编译。

1.4Typescript 工具

TypeScript 编译器，名称叫 tsc，是用可以被编译为可以被执行在任何 JavaScript 引擎中，在任何宿主 - 如浏览器 - 中的常规 JavaScript 的 TypeScript 写的。编译器包被绑定于一个可以执行编译器的脚本宿主。使用 Node.js 作为宿主的 Node.js 包同样可以获得。

也有用 JavaScript 写的客户端编译器的一个 alpha 版本，它在页面载入时，实时执行 JavaScript 代码。

这种编译器的当前版本默认支持 ECMAScript 3,一个选项是允许以 ECMAScript 5 为目标以利用该版本独有的语言特性。类，尽管是 ECMAScript 6 标准的一部分，在这两个模式下都可用。

IDE 和编辑器支持

Microsoft 为 Visual Studio 2012 和 WebMatrix 提供了一个插件，也为 Sublime Text, Emacs 和 Vim 提供了基本的文本编辑器支持。在线的 Cloud9 IDE 也支持 TypeScript。JetBrains 也计划在他们的 IDE 系列中支持 TypeScript，而且已经发行了具有部分支持的 PhpStorm 6 和 WebStorm 6 预览版本

1.5Typescript 开源

TypeScript 是开源的，其源代码可以在 Apache 2 License 下从 CodePlex 获得。这个项目由 Microsoft 维持，但是任何人可以通过经 CodePlex 项目页发送反馈，建议和 bugfixes 而做出贡献。[10]

已有一些批评提到这一想法，即使 TypeScript 鼓励强类型，当前也只有 Microsoft Visual Studio 允许为该语言容易的开发。最初的观点是在其它的编辑器上带来强类型，IntelliSense，代码完成和代码重构可能不是一个简单的任务。此外，允许为 TypeScript 开发的 Visual Studio 扩展不是开源的。最好的 TypeScript 开发体验是在 Microsoft Windows 上， 然而随着时间的流逝以及这种语言开放的本质，加之编译器自我托管，而且用 TypeScript 自身写的，这很有可能会改变。可以通过编译器的源代码访问到 AST (抽象句法树)，也可以获得详细的语言规范文档，社区已开始构建一个跨平台的编辑器， 利用和 Visual Studio 所用相同的语言服务以提供一个增强的编辑体验。编辑器仍然在概念检验的阶段，但已经运行于 Linux, OSX 和 Windows，提供针对之前对提供此类服务的困难度的估计的 IntelliSense，代码完成和句法高亮。

6 发布

2013 年 6 月 19 日，在经历了一个预览版之后微软正式发布了正式版 TypeScript 0.9，向未来的 TypeScript 1.0 版迈进了很大一步。

TypeScript 0.9 迎来了一些重大的新功能，除对语言本身特性进行了扩充之外，还更加完善地整合了 Visual Studio，微软开发部副总裁 Soma Somasegar 发布帖子称，新版本的 TypeScript 在交互式性能方面有了戏剧性的提高和改善。

与 JavaScript 相比，TypeScript 进步的地方包括：加入注释，让编译器理解所支持的对象和函数，编译器会移除注释，不会增加开销；增加一个完整的类结构，使之更新是传统的面向对象语言

二、TypeScript 安装

2.1Typescript 安装图解过程.

1、详细安装文章请看
http://blog.csdn.net/jilongliang/article/details/21942911

2、TypeScript 官方手册
http://www.typescriptlang.org/Handbook

3、TypeScript 官方例子
https://github.com/Microsoft/TypeScriptSamples
http://typescript.codeplex.com/

4、注意 TypeScript 的 ts 文件会多处有红色的 XX，并不代表它有错误，如果是 Myeclipse 的话可以在 Myeclipse-->> Exclude From Validation 忽略红色的问题

5、看 TypeScript 例子源碼情况.ts 文件，javascript 源碼是.js

6、参考博客..
http://ju.outofmemory.cn/entry/954
http://www.oschina.net/question/12_72250
http://www.cnblogs.com/whitewolf/p/4103328.html

2.2Typescript 安装前注意事项

A、必须选择 ECMAScript target version ES3 或 ES5，选择 ES6 版本太高，生成的 js 文件会有 class 关键字

B、eclipse 设置默认.ts 文件格式是 TypeScript
Window--->General-Editor--->File Association-->*.ts 选择默认 TypeScript

C、 不要勾选 Enable typeScript Builder，勾选了这个它不会帮你生成 js 文件
勾选 Disable typeScript Builder 即可

D、找到官方的 jquery.d.ts 文件拷贝到工程项目，在学习开发的时候会引用到.

E、在 window 看到有点象视频或音乐格式文件，用视频播放器是的不开的，必须用编辑器打开

三、TypeScript 基本类型

3.1Typescript 基本类型使用.

Ts 代码.

```
//对于程序来说我们需要基本的数据单元,如：numbers, strings, structures, boolean 等数据结构.在 TypeScript 中我们支持很多你所期望在 JavaScript 中所拥有的数据类型系统。
//var isFlag:boolean=false;这个 : 号代表应该是代表继承的意思，从 C#的.cs 文件可以看出来 public class IndexController : Controller,Typescript 也是微软开发出来的一个超级 js

//1、声明一个 boolean 类型默认值是 false
//在 JavaScript 和 TypeScript 中也具有最基本的逻辑断言值 true/false，采用'boolean'类型。
var isFlag:boolean=false;

//2、声明一个 number 类型值
//如 JavaScript，TypeScript 所有的数值类型采用浮点型计数，其类型为'number'。

var orderNumber:number=100;

//3、声明一个 String 类型
//在 webpages 的 JavaScript 或者服务端的应用程序最基本的功能就是处理文本数据。在其他语言中大多使用'string'去代表文本数据类型。
//TypeScript 和 JavaScript 一样也是用双引号(")或者单引号包裹文本数据
var userName:string="龙梅子";

//4、数组 Array
//在 TypeScript 中如 JavaScript 一样允许我们操结合操作。数组类型可以使用下边两种方式之
```

```
一
//第一种方式，你可以在数据类型之后带上’[]‘:
var list:number[] = [1, 2, 3];
//第二种方式，也可以采用泛型的数组类型：
var list1:Array<number> = [1, 2, 3];//泛型数组

//5、枚举 Enum
//TypeScript 为 JavaScript 新增了枚举这种标准的集合数据类型。和在 c#中一样，枚举是为一组数值类型一组更友好的名称：
//

//------------------------------
enum Color {Red, Green, Blue};//enum 关键字 枚举对象{声明变量}
var c1: Color = Color.Green;//从枚举里面拿出绿色出来赋给一个叫 c 的变量

//---------手动枚举所有值都设置----------------------
//默认枚举类型其实数值从 0 开始，你可以可用手动设置某一个成员的数值。例如我们可以将上文的起始值定为 1：
enum Color1 {Red = 1, Green = 2, Blue = 4};
var c2: Color1 = Color1.Green;

//---------手动设置全部的枚举成员：----------------------
enum Color2 {Red = 1, Green, Blue};
var colorName: string = Color2[2];

alert(colorName);

/**
 *我们可能需要描述变量的类型，当我们编写的应用程序，我们可能不知道。
 *这些值可能来自动态内容，例如从用户或第三方库。在这种情况下，我们要退出类型检查的，
 *让价值观通过编译时检查，要做到这一点，我们的标签，这些与'任何'类型：
 */

//6、any
//any’类型是一种强大的兼容存在的 JavaScript 库的类型系统。他允许跳过 TypeScript 的编译时类型的检查。
//‘any’类型对于我们只知道部分数据类型，但是不是所有的数据类型的类型系统。如一个混合了多种类型的集合数组
var notSure: any = 4;//notSure 这个是不确定的值，默认先给一个数字 4
notSure = "this string";//改变这个值为 this string
```

```
notSure = false; //最终确定的值是一个 boolean 值.
//------------------
var list2:any[] = [1, true, "free"];

list2[1] = 100;

//7、void 和‘any’相对的数据类型则是’Void‘，它代表没有任何数据类型。我们常用的一个方法没有任何返回值：
//,格式如：function doMain:void{}

function warnUser(): void {
    alert("This is my void");
}
```

Ts 编译生成的 js 代码

```
//对于程序来说我们需要基本的数据单元,如：numbers, strings, structures, boolean 等数据结构。
//在 TypeScript 中我们支持很多你所期望在 JavaScript 中所拥有的数据类型系统。
//var isFlag:boolean=false;这个 : 号代表应该是代表继承的意思，从 C#的.cs 文件可以看出来，
//public class IndexController : Controller,Typescript 也是微软开发出来的一个超级 js
//1、声明一个 boolean 类型默认值是 false
//在 JavaScript 和 TypeScript 中也具有最基本的逻辑断言值 true/false，采用’boolean’类型。
var isFlag = false;
//2、声明一个 number 类型值
//如 JavaScript，TypeScript 所有的数值类型采用浮点型计数，其类型为‘number’。
var orderNumber = 100;
//3、声明一个 String 类型
//在 webpages 的 JavaScript 或者服务端的应用程序最基本的功能就是处理文本数据。在其他语言中大多使用’string’去代表文本数据类型。
//TypeScript 和 JavaScript 一样也是用双引号(“)或者单引号包裹文本数据
var userName = "龙梅子";
//4、数组 Array
//在 TypeScript 中如 JavaScript 一样允许我们操结合操作。数组类型可以使用下边两种方式之一
//第一种方式，你可以在数据类型之后带上’[]‘:
var list = [1, 2, 3];
```

```
//第二种方式，也可以采用泛型的数组类型：
var list1 = [1, 2, 3]; //泛型数组
//5、枚举 Enum
//TypeScript 为 JavaScript 新增了枚举这种标准的集合数据类型。和在 c#中一样，枚举是为一组数值类型一组更友好的名称：
//
//------------------------------
var Color;
(function (Color) {
    Color[Color["Red"] = 0] = "Red";
    Color[Color["Green"] = 1] = "Green";
    Color[Color["Blue"] = 2] = "Blue";
})(Color || (Color = {}));
; //enum 关键字 枚举对象{声明变量}
var c1 = Color.Green; //从枚举里面拿出绿色出来赋给一个叫 c 的变量
//---------手动枚举所有值都设置---------------------
//默认枚举类型其实数值从 0 开始，你可以可用手动设置某一个成员的数值。例如我们可以将上文的起始值定为 1：
var Color1;
(function (Color1) {
    Color1[Color1["Red"] = 1] = "Red";
    Color1[Color1["Green"] = 2] = "Green";
    Color1[Color1["Blue"] = 4] = "Blue";
})(Color1 || (Color1 = {}));
;
var c2 = Color1.Green;
//---------手动设置全部的枚举成员：---------------------
var Color2;
(function (Color2) {
    Color2[Color2["Red"] = 1] = "Red";
    Color2[Color2["Green"] = 2] = "Green";
    Color2[Color2["Blue"] = 3] = "Blue";
})(Color2 || (Color2 = {}));
;
var colorName = Color2[2];
alert(colorName);
/**
 *我们可能需要描述变量的类型，当我们编写的应用程序，我们可能不知道。
 *这些值可能来自动态内容，例如从用户或第三方库。在这种情况下，我们要退出类型检查的，
 *让价值观通过编译时检查，要做到这一点，我们的标签，这些与'任何'类型：
```

```
 */
//6、any
//any’类型是一种强大的兼容存在的 JavaScript 库的类型系统。他允许跳过 TypeScript 的编译时类型的检查。
//‘any’类型对于我们只知道部分数据类型，但是不是所有的数据类型的类型系统。如一个混合了多种类型的集合数组
var notSure = 4; //notSure 这个是不确定的值，默认先给一个数字 4
notSure = "this string"; //改变这个值为 this string
notSure = false; //最终确定的值是一个 boolean 值.
//------------------
var list2 = [1, true, "free"];
list2[1] = 100;
//7、void 和‘any’相对的数据类型则是’Void‘，它代表没有任何数据类型。我们常用的一个方法没有任何返回值：
//,格式如：function doMain:void{}
function warnUser() {
    alert("This is my void");
}
```

3.2Typescript 基本 void,string 类型

Ts 代码

```
/****使用 reference 和 path 引入 jquery.d.ts 文件使用 jquery$就不会报错.**/

/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
/***返回 void 值***/
 function setTableRowHtml1():void{
    var userName:string="";
    $("tr").each(function(){
         userName=$(this).find("td:eq(1)").html();
    });
    alert(userName);
 }

/***返回 string 一个值***/
 function setTableRowHtml2():string{
    var userName:string="";
    $("tr").each(function(){
         userName=$(this).find("td:eq(1)").html();
```

```
    });
    return userName;
 }
//---jquery 执行.
$(function(){
    //setTableRowHtml1();
    var userName=setTableRowHtml2();
    alert(userName);
});
```

Ts 编译生成的 js 代码

```
/****使用 reference 和 path 引入 jquery.d.ts 文件使用 jquery$就不会报错.**/
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
/**
 * ts 的 void，string 结合 jquery 的使用.
 *
 */
/***返回 void 值***/
function setTableRowHtml1() {
    var userName = "";
    $("tr").each(function () {
        userName = $(this).find("td:eq(1)").html();
    });
    alert(userName);
}
/***返回 string 一个值***/
function setTableRowHtml2() {
    var userName = "";
    $("tr").each(function () {
        userName = $(this).find("td:eq(1)").html();
    });
    return userName;
}
//---jquery 执行.
$(function () {
    //setTableRowHtml1();
    var userName = setTableRowHtml2();
    alert(userName);
});
```

Html 代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script type="text/javascript" src="../test2.js"></script>
</head>
<body>
    <div align="center">
        <table border="1" id="tableId">
            <tr>
                <td>age</td>
                <td>username</td>
                <td>adddrss</td>
            </tr>
            <tr>
                <td>27</td>
                <td>龙梅子</td>
                <td>广东省广州</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

3.3Typescript 基本 void,any,Array 类型

Ts 代码.

```
/****使用 reference 和 path 引入 jquery.d.ts 文件使用 jquery.d.ts 声明的的$，不引就报错.**/
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

//var colorList:Array<string>=["red","blue","green","yellow"];//string 类型
var colorList:Array<any>=["red","blue","green","yellow"];//any 不知道类型
```

```
//---测试数组打印出来控制台/
function testArray():void{
    $.each(colorList,function(key,val){
        console.log(key+"="+val);
    });
    $(colorList).each(function(key,val){
        console.log(key+"="+val);
    });
}
/***返回一个值***/
 function setTableRowColor():void{
    $("tr").each(function(){
         //找到 tr 下面的所有 td 的第二个列都加上背景 blue,字体颜色加上 red
         $(this).find("td:eq(1)").css({ color: "red", background: "blue" });
    });
 }
 function setTableRowColor1():void{
     $("tr").each(function(){
        for(var i=0;i<colorList.length;i++){
            //找到 tr 下面的所有 td 的第二个列都加上背景 blue，颜色就加上 green 色
            if(i==1){
                 $(this).find("td:eq(1)").css({ color: colorList[i], background:
colorList[i+1] });
            }
        }
    });
 }
$(document).ready(function(){
    testArray();
    setTableRowColor();
    //setTableRowColor1();
});
```

Ts 编译生成的 js 代码

```
/****使用 reference 和 path 引入 jquery.d.ts 文件使用 jquery.d.ts 声明的的$，不引就报
错.**/
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

//var colorList:Array<string>=["red","blue","green","yellow"];//string 类型
var colorList:Array<any>=["red","blue","green","yellow"];//any 不知道类型
```

```
//---测试数组打印出来控制台/
function testArray():void{

    $.each(colorList,function(key,val){
        console.log(key+"="+val);
    });

    $(colorList).each(function(key,val){
        console.log(key+"="+val);
    });
}

/***setTableRowColor**function*/
 function setTableRowColor():void{
    $("tr").each(function(){
         //找到 tr 下面的所有 td 的第二个列都加上背景 blue,字体颜色加上 red
         $(this).find("td:eq(1)").css({ color: "red", background: "blue" });
    });
 }
/***setTableRowColor1*function**/
 function setTableRowColor1():void{
     $("tr").each(function(){
        for(var i=0;i<colorList.length;i++){
            //找到 tr 下面的所有 td 的第二个列都加上背景 blue，颜色就加上 green 色
            if(i==1){
                 $(this).find("td:eq(1)").css({ color: colorList[i], background:
colorList[i+1] });
            }
        }
    });
 }
$(document).ready(function(){
    testArray();
    setTableRowColor();
    //setTableRowColor1();
});
```

Html 文件

```
<!DOCTYPE html>
```

```html
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script type="text/javascript" src="../test3.js"></script>
</head>
<body>
    <div align="center">
        <table border="1" id="tableId">
            <tr>
                <td>age</td>
                <td>username</td>
                <td>adddrss</td>
            </tr>
            <tr>
                <td>27</td>
                <td>龙梅子</td>
                <td>广东省广州</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

四、TypeScript 接口

4.1 最简单接口使用

Ts 代码.

```typescript
/**--声明一个接口，这个接口不会在 js 上面出现，只会在显示一个 user 对象在 getUserInfo*/
interface IUserInfo{
    age : any;//定义一个任何变量的 age.
    userName :string;//定义一个 username.
}
```

```
/*********获取用户信息*******/
function getUserInfo(user : IUserInfo):string{
    return  user.age+"======"+user.userName;
}
//用一个数组对象作为一个 user 对象传值过 getUserInfo 函数方法..参数必须要以接口
IUserInfo 对应上.
//少传一个参数，typescript 会自动帮你检测报错,如果用纯 javascript 去写的话，不会报错，
ts 大大减少检查 js 问题
//如：var userObj={userName:'周伯通'};//error

var userObj={userName:'周伯通',age:100};

$(function(){
    //定义变量接收值..
   var userInfo:string= getUserInfo(userObj);
    console.log(userInfo);
});
```

Ts 编译生成的 js 文件代码

```
/*********获取用户信息*******/
function getUserInfo(user) {
    return user.age + "======" + user.userName;
}
//用一个数组对象作为一个 user 对象传值过 getUserInfo 函数方法..参数必须要以接口
IUserInfo 对应上.
//少传一个参数，typescript 会自动帮你检测报错,如果用纯 javascript 去写的话，不会报错，
ts 大大减少检查 js 问题
//如：var userObj={userName:'周伯通'};//error
var userObj = { userName: '周伯通', age: 100 };
$(function () {
    //定义变量接收值..
    var userInfo = getUserInfo(userObj);
    console.log(userInfo);
});
```

4.2 自选属性 Optional Properties

Ts 文件代码

```
/**
 * Not all properties of an interface may be required.
 * Some exist under certain conditions or may not be there at all
 *并非需要一个接口的所有属性。在某些条件下的一些存在或可以不存在的。
 *这句话说的是：就算你 SquareConfig 接口定义的变量是 color，到调用 createSquare 的时候你给 color1 变量照样可以取出 z 值来
 *这个值只不过是：默认的 newSquare 的 white 值，如果是一样的 color 变量他就会取出你给赋格对象的 color(red)
 */
interface SquareConfig {
  color?: string;//
  width?: number;
}
/***************创建一个对象 function.**************/
function createSquare(config: SquareConfig): {color: string; area: number} {
    //此时 newSquare 里面的参数必须与 :后面里面的参数名称一致.
     var newSquare = {color: "white", area: 100};

      if (config.color) {
          newSquare.color = config.color;
      }
      if (config.width) {
          newSquare.area =  newSquare.area * config.width;
      }
      return newSquare;
}

//--createSquare 返回的对象是 newSquare，所有只能获取 color 和 area 并获取不了 width 这个属性的值..

var mySquare1 = createSquare({color: "red"});//与接口的变量 color 一样，此时这个值是取出是默认值 color=red
var mySquare2 = createSquare({color1: "red"});//与接口的变量 color 不一样，此时这个值是取出是默认值 color=white
console.log(mySquare1.color+"=="+mySquare1.area);//
console.log(mySquare2.color+"=="+mySquare2.area);//

var mySquare3 = createSquare({color: "yellow",width:80});//这里给了两个变量值，一个是 color，一个是 width
```

```
console.log(mySquare3.color+"=="+mySquare3.area);//所以这个值必须等于 8000
```

Ts 编译生成的 js 文件代码

```
/**
 * Not all properties of an interface may be required.
 * Some exist under certain conditions or may not be there at all
 *并非需要一个接口的所有属性。在某些条件下的一些存在或可以不存在的。
 *这句话说的是：就算你 SquareConfig 接口定义的变量是 color，到调用 createSquare 的时候你给 color1 变量照样可以取出 z 值来
 *这个值只不过是：默认的 newSquare 的 white 值，如果是一样的 color 变量他就会取出你给赋格对象的 color(red)
 */
/***************创建一个对象 function.**************/
function createSquare(config) {
    //此时 newSquare 里面的参数必须与 :后面里面的参数名称一致.
    var newSquare = { color: "white", area: 100 };
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = newSquare.area * config.width;
    }
    return newSquare;
}
//--createSquare 返回的对象是 newSquare，所有只能获取 color 和 area 并获取不了 width 这个属性的值..
var mySquare1 = createSquare({ color: "red" }); //与接口的变量 color 一样，此时这个值是取出是默认值 color=red
var mySquare2 = createSquare({ color1: "red" }); //与接口的变量 color 不一样，此时这个值是取出是默认值 color=white
console.log(mySquare1.color + "==" + mySquare1.area); //
console.log(mySquare2.color + "==" + mySquare2.area); //
var mySquare3 = createSquare({ color: "yellow", width: 80 }); //这里给了两个变量值，一个是 color，一个是 width
console.log(mySquare3.color + "==" + mySquare3.area); //所以这个值必须等于 8000
```

4.3JavaScript 的 search 函数

Ts 文件代码

```
//--typescript 的 function 类型结合 javascript 的 search 函数使用
interface searchFunt{
    //声明一个两个变量..
    (source: string, subString: string): boolean;
}
var mySearch : searchFunt;//声明一个 interface 变量接收
    mySearch = function(source:string,subString:string){
   var result = source.search(subString);
      if (result == -1) {
        return false;
      }
      else {
        return true;
      }
}

$(function(){
      var source:string ="this is ok";
      var subString1:string ="ok";
      var subString2:string ="not";
      var result:boolean;
      var result1= mySearch(source,subString1);//从 source 字符串上面找 ok，返回值是 true
      var result2= mySearch(source,subString2);//从 source 字符串上面找 not，返回值是 false
      alert(result1);//
      alert(result2);
});
```

Ts 编译生成的 js 文件代码

```
//--typescript 的 function 类型结合 javascript 的 search 函数使用
var mySearch; //声明一个 interface 变量接收
mySearch = function (source, subString) {
    var result = source.search(subString);
    if (result == -1) {
        return false;
    }
```

```
    else {
        return true;
    }
};
$(function () {
    var source = "this is ok";
    var subString1 = "ok";
    var subString2 = "not";
    var result;
    var result1 = mySearch(source, subString1); //从 source 字符串上面找 ok，返回值是 true
    var result2 = mySearch(source, subString2); //从 source 字符串上面找 not，返回值是 false
    alert(result1); //
    alert(result2);
});
```

4.4 接口定义 Array 类型

Ts 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//Array Types

interface StringArray {
  [index: number]: string;
  //length: number;
}
var myArray:StringArray;
myArray = ["Bob", "Fred"];

$(function(){
     $.each(myArray,function(key,val){
        alert(val);
     });
});
```

Ts 编译生成的 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//Array Types
```

```
var myArray;
myArray = ["Bob", "Fred"];
$(function () {
    $.each(myArray, function (key, val) {
        alert(val);
    });
});
```

4.5class 实现 implements 接口

Ts 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//Class Types(implements)
 interface IClock {
    currentTime: Date;
    setTime(d: Date);
}
//--实现 IClock 接口
class Clock implements  IClock{
    currentTime:Date;
    constructor(h: number, m: number) { }//--构造函数方法
    setTime(d:Date){
        this.currentTime=d;
    }
}
 //-------------------------------------------------
interface IClock1 {
     new (hour: number, minute: number);
}
class Clock1  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
var cs: IClock1 = Clock1;
var newClock = new cs(7, 30);
console.log(newClock);
```

Ts 编译生成的 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//--实现 IClock 接口
var Clock = (function () {
    function Clock(h, m) {
    } //--构造函数方法
    Clock.prototype.setTime = function (d) {
        this.currentTime = d;
    };
    return Clock;
})();
var Clock1 = (function () {
    function Clock1(h, m) {
    }
    return Clock1;
})();
var cs = Clock1;
var newClock = new cs(7, 30);
console.log(newClock);
```

4.6 扩展接口 Extending Interfaces

Ts 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

//Extending Interfaces
interface IShape{
    color:string;
}
interface PenStroke {
    penWidth: number;
}
//--接口继承接口,用，分割开多继承.
interface ISquare extends IShape,PenStroke {
    sideLength: number;
}
//---赋值..
```

```
var square = <ISquare>{};

square.color="red";
square.sideLength=100;
square.penWidth=50;
```

Ts 编译成的 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//---赋值..
var square = {};
square.color = "red";
square.sideLength = 100;
square.penWidth = 50;
```

4.7 混合型 Hybrid Types

Ts 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//Hybrid Types(混合型)
//--计算器
interface Counter  {
   (start: number): string;//声明一个开始变量
   interval:number;//声明一个间隔变量
   reset(): void;//声明一个重置 function 方法
}
var c: Counter;
c(10);//开始.
c.interval=5.0;
c.reset();//重置.
```

Ts 编译成 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
var c;
c(10); //开始.
c.interval = 5.0;
c.reset(); //重置.
```

五、TypeScript 类

5.1 最简单 class 使用.

Ts 文件代码

```
 /// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//--这个是简单的 class
class Employee {
    fullName: string;
}
var employee = new Employee();
employee.fullName = "Long long";//赋值

//说明这个属性是存在的..
if (employee.fullName) {
    alert(employee.fullName);
}
```

Ts 文件编译成 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//--这个是简单的 class
var Employee = (function () {
    function Employee() {
    }
    return Employee;
})();
var employee = new Employee();
employee.fullName = "Long long"; //赋值
//说明这个属性是存在的..
if (employee.fullName) {
    alert(employee.fullName);
}
```

5.2 在 class 使用 constructor 关键字

Ts 文件代码

```
 /// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//--class 和 constructor 构造器使用。
 class UserInfo{
    username:string;
    //--默认的构造方法..
    constructor(msg : string){
        this.username=msg;//从构造方法传一个用户字符串过去.
    }
    getUserInfo(){
         return "欢迎您, " + this.username;
    }
 }
function printMsg():string{
    var resMsg:string="";
    var g=new UserInfo("龙梅子");//创建一个 UserInfo 对象,并且构造函数必须要传一个字符串.
    resMsg=g.getUserInfo();//调用对象方法.
    return resMsg;
}
/****************jQuery-执行..**********************/
$(function(){
    var result=printMsg();
    $("#msg").html("<span style='color:green;'>"+result+"<span>");

});
```

Ts 文件编译成 js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
//--class 和 constructor 构造器使用 var UserInfo= (function () {
    //--默认的构造方法..
    function UserInfo(msg) {
        this.username=msg;//从构造方法传一个用户字符串过去.
    }
    UserInfo.prototype.getUserInfo = function () {
        return "欢迎您, " + this.username;
    };
```

```
    return UserInfo;
})();
function printMsg() {
    var resMsg = "";
    var g = new UserInfo("龙梅子"); //创建一个 UserInfo 对象,并且构造函数必须要传一个字符串.
    resMsg = g.getUserInfo(); //调用对象方法.
    return resMsg;
}
/***************jQuery-执行..*******************/
$(function () {
    var result = printMsg();
    $("#msg").html("<span style='color:green;'>" + result + "<span>");
});
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script type="text/javascript" src="../test2.js"></script>
</head>
<body>
    <div id="msg"></div>
</body>
</html>
```

5.3 在 class 使用 super 关键字

Ts 文件代码

```
 /// <reference path="../plugins/typescript/typings/jquery.d.ts" />

//------class 与 supper 使用.----------------
class Person{
    userName:string;//声明一个名称
```

```
    //构造方法
    constructor(paramVal:string){
        this.userName=paramVal;
    }

    //--声明一个 getPersonInfo 方法，并在声明 age 变量
    getPersonInfo(age:number=120):string{
        return  this.userName+"\n"+age;
    }
}
class Student1 extends Person{
    constructor(username:string){
        super(username);
    }
    getPersonInfo(age=100){
        var superMsg=super.getPersonInfo(age);
        return this.userName+"\n"+age+"岁"+"\n\t\t"+"默认信息： " +superMsg;
    }
}
class Student2 extends Person{

    constructor(username:string){
        super(username);
    }
    getPersonInfo(age=120){
        var superMsg=super.getPersonInfo(age);
        return this.userName+"\n"+age+"岁"+"\n\t\t"+"默认信息： " +superMsg;
    }
}

var stu1=new Student1("周伯通");
var stu2=new Student2("老毒物");

var stuMsg1=stu1.getPersonInfo();
var stuMsg2=stu2.getPersonInfo(80);//传一个默认值给 getPersonInfo 方法

$(function(){

    $("#msg1").html("<span style='color:red;'>"+stuMsg1+"<span>");
    $("#msg2").html("<span style='color:blue;'>"+stuMsg2+"<span>");
```

```
});
```

Ts 编译成 Js 文件代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
//------class 与 supper 使用.----------------
var Person = (function () {
    //构造方法
    function Person(paramVal) {
        this.userName = paramVal;
    }
    //--声明一个 getPersonInfo 方法，并在声明 age 变量
    Person.prototype.getPersonInfo = function (age) {
        if (age === void 0) { age = 120; }
        return this.userName + "\n" + age;
    };
    return Person;
})();
var Student1 = (function (_super) {
    __extends(Student1, _super);
    function Student1(username) {
        _super.call(this, username);
    }
    Student1.prototype.getPersonInfo = function (age) {
        if (age === void 0) { age = 100; }
        var superMsg = _super.prototype.getPersonInfo.call(this, age);
        return this.userName + "\n" + age + "岁" + "\n\t\t" + "默认信息： " + superMsg;
    };
    return Student1;
})(Person);
var Student2 = (function (_super) {
    __extends(Student2, _super);
    function Student2(username) {
        _super.call(this, username);
```

```
    }
    Student2.prototype.getPersonInfo = function (age) {
        if (age === void 0) { age = 120; }
        var superMsg = _super.prototype.getPersonInfo.call(this, age);
        return this.userName + "\n" + age + "岁" + "\n\t\t" + "默认信息： " + superMsg;
    };
    return Student2;
})(Person);
var stu1 = new Student1("周伯通");
var stu2 = new Student2("老毒物");
var stuMsg1 = stu1.getPersonInfo();
var stuMsg2 = stu2.getPersonInfo(80); //传一个默认值给 getPersonInfo 方法
$(function () {
    $("#msg1").html("<span style='color:red;'>" + stuMsg1 + "<span>");
    $("#msg2").html("<span style='color:blue;'>" + stuMsg2 + "<span>");
});
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script type="text/javascript" src="../test4.js"></script>
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
</body>
</html>
```

5.4 在 class 使用 public/private 关键字

Ts 文件代码

```
/**
 * public/private
 * 默认是 public
 * 您可能已经注意到，在我们还没有使用这个词“公众”作出任何类可见的成员的上述例子。
 * 如 C＃语言要求每个成员被明确标记为'公共'可见。在打字稿，每个成员都是公共默认。
 * 您可能仍然标记成员的私人，所以你控制什么是公开可见的外部类的
 */
class MyAnimal {
    private name:string;
    //构造方法
    constructor(private  theName : string){
        this.name = theName;
    }
    getMsg(name : string):string{
        return this.name=name;
    }

}
//犀牛
class Rhino extends MyAnimal{
    constructor(){
        super("犀牛");
    }
    getMsg(name : string):string{
        return name;
    }
}
//员工
class Employees {
    private name:string;
     //构造方法
    constructor(theName : string) {
        this.name = theName;
    }
}

var animal = new MyAnimal("山羊");//Goat 山羊
```

```
var retMsg1=animal.getMsg("鹿");

var rhino = new  Rhino();

var employees = new Employees("洪七公");

animal = rhino;
//animal = employees;//此时这个值不能赋给 animal，并不能编译通过.

$(function(){
    $("#msg1").html("<span style='color:red'>"+retMsg1+"</span>");

});
```

Ts 编译成 js 代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
/**
 * public/private
 * 默认是 public
 * 您可能已经注意到，在我们还没有使用这个词“公众”作出任何类可见的成员的上述例子。
 * 如 C＃语言要求每个成员被明确标记为'公共'可见。在打字稿，每个成员都是公共默认。
 * 您可能仍然标记成员的私人，所以你控制什么是公开可见的外部类的
 */
var MyAnimal = (function () {
    //构造方法
    function MyAnimal(theName) {
        this.theName = theName;
        this.name = theName;
    }
    MyAnimal.prototype.getMsg = function (name) {
        return this.name = name;
    };
    return MyAnimal;
})();
```

```
//犀牛
var Rhino = (function (_super) {
    __extends(Rhino, _super);
    function Rhino() {
        _super.call(this, "犀牛");
    }
    Rhino.prototype.getMsg = function (name) {
        return name;
    };
    return Rhino;
})(MyAnimal);
//员工
var Employees = (function () {
    //构造方法
    function Employees(theName) {
        this.name = theName;
    }
    return Employees;
})();
var animal = new MyAnimal("山羊"); //Goat 山羊
var retMsg1 = animal.getMsg("鹿");
var rhino = new Rhino();
var employees = new Employees("洪七公");
animal = rhino;
//animal = employees;//此时这个值不能赋给 animal，并不能编译通过.
$(function () {
    $("#msg1").html("<span style='color:red'>" + retMsg1 + "</span>");
});
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script type="text/javascript" src="../test5.js"></script>
</head>
```

```
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
</body>
</html>
```

5.5 在 class 使用高级技巧

Ts 文件代码

```
/**
*高级技巧
*构造函数
*当您声明一个类，你实际上是在同一时间创建多个声明。第一个是类的实例的类型
*/
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    }
}
var greeter1: Greeter;
    greeter1 = new Greeter();
    alert(greeter1.greet());

var greeterMaker: typeof Greeter = Greeter;
    greeterMaker.standardGreeting = "Hey there!";
var greeter2:Greeter = new greeterMaker();
    alert(greeter2.greet());
```

Ts 编译成 js 文件代码

```
/**
*高级技巧
*构造函数
*当您声明一个类，你实际上是在同一时间创建多个声明。第一个是类的实例的类型
*/
var Greeter = (function () {
    function Greeter() {
    }
    Greeter.prototype.greet = function () {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    };
    Greeter.standardGreeting = "Hello, there";
    return Greeter;
})();
var greeter1;
greeter1 = new Greeter();
alert(greeter1.greet());
var greeterMaker = Greeter;
greeterMaker.standardGreeting = "Hey there!";
var greeter2 = new greeterMaker();
alert(greeter2.greet());
```

六、TypeScript 块

6.1 分多个 ts 文件实现 module 块

Validation.ts 代码

```
module Validation{
    export interface StringValidator {
         isAcceptable(s: string): boolean;//是否接受.
    }
}
```

ZipCodeValidator.ts 代码

```
/// <reference path="Validation.ts" />
module Validation {
   //匹配 0-9 的数字.
    var numberRegexp = /^[0-9]+$/;

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            //如果长度=5 并且是数字就返回一个 true
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

LettersOnlyValidator.ts 代码

```
/// <reference path="Validation.ts" />
module Validation {
    //匹配 A-Z,a-z 的英文
    var lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }}
```

test-1.ts 代码

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
/// <reference path="test1/Validation.ts" />
/// <reference path="test1/LettersOnlyValidator.ts" />
/// <reference path="test1/ZipCodeValidator.ts" />

/***
 * Splitting Across Files 分割跨文件
 */
```

```
// 声明一个数组.
var strings = ['Hello', '98052', '101'];

// 使用这个验证.
var validators: { [s: string]: Validation.StringValidator; } = {};

    validators['Zip Code'] = new Validation.ZipCodeValidator();//这个是验证邮政编码
    validators['Letters only'] = new Validation.LettersOnlyValidator();//这个是验证英文


function showMsg():void{

    //显示每个字符串是否通过每个验证
    strings.forEach(s => {
        for (var name in validators) {

            console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ') + name);
            $("#msg1").html('"' + s + '" ' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ') + name);
        }//--for--end

    });//--forEach--end
}

$(document).ready(function(){
    showMsg();

});
```

Html 文件

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
```

```
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script src="../test1/Validation.js" type="text/javascript"></script>
<script src="../test1/LettersOnlyValidator.js"
type="text/javascript"></script>
<script src="../test1/ZipCodeValidator.js" type="text/javascript"></script>
<script src="../test-1.js" type="text/javascript"></script>
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
</body>
</html>
```

6.2 不分文件实现 module 块

ValidationUtils.ts

```
/**
 *声明一个 ValidationUtils 工具块 module
 *推荐使用.
 */
module ValidationUtils{

    //-声明 StringValidator 字符串验证器.
     export interface StringValidator {
         isAcceptable(str: string): boolean;//是否接受.
    }

    //  匹配 email 正则表达式
    var emailReg = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$/;
    export class EmailValidator implements StringValidator {
        isAcceptable(s: string) {
            return emailReg.test(s);
        }
    }

    //匹配移动电话号码
```

```
    var telReg=/^(13[0-9]|15[0-9]|18[0-9])\d{8}$/;

    export class TelValidator implements StringValidator{
        isAcceptable(s:string){
           return  telReg.test(s);
        }
    }
}
```

测试文件 test-2.ts

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />
/// <reference path="test2/ValidationUtils.ts" />


var strs : Array<any> =["13697811800","jilongliang@sina.com"];

var validators1: { [s: string]: ValidationUtils.StringValidator; } = {};

validators1["Tel"]=new ValidationUtils.TelValidator;//验证码 QQ

validators1["Email"] = new ValidationUtils.EmailValidator;//验证 Email

//-------显示信息 1----------------------------
function showMsg1():void{
    strs.forEach
    (s=>
      {
        for(var name in validators1 ){
            console.log('"' + s + '" ' + (validators1[name].isAcceptable(s) ? '
匹配 ' : ' 不匹配 ') + name);
        }
      }
   );
}

//-------------------显示信息 2---------------------
function showMsg2():void{

    //--方法一---
    var telObj:ValidationUtils.TelValidator;
```

```
    telObj=new ValidationUtils.TelValidator;

     //--方法二---
     //var telObj=new ValidationUtils.TelValidator;

    var tel : string="13697811809";
    var flag : boolean=telObj.isAcceptable(tel);//调用 TelValidator 类的
isAcceptable 方法

    console.log(flag? tel+" 匹配 " : tel+"\t 不匹配 ");

    $("#msg2").html(flag? "<span style='color:red;'>"+tel+" 匹配</span> " :
"<span>"+tel+"\t 不匹配</span>");

}

$(function(){
    showMsg1();

    showMsg2();
});
```

Html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script src="../test2/ValidationUtils.js" type="text/javascript"></script>
<script src="../test-2.js" type="text/javascript"></script>
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
</body>
</html>
```

6.3import， require 关键字

ValidationUtils3.ts 文件

```
/* 这个 ts 没 module 关键字*/

export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

EmailValidator.ts

```
   /**import、require、export 关键的使用..***********/

    //--导入--ValidationUtils3.ts 文件---
    import validation = require('./ValidationUtils3');

    //  匹配 email 正则表达式
    var emailReg = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$/;
    export class EmailValidator implements validation.StringValidator {
        isAcceptable(s: string) {
            return emailReg.test(s);
        }
    }
```

TelValidator.ts

```
/**import、require、export 关键的使用..***********/

//--导入--ValidationUtils3.ts 文件---
import validation = require('./ValidationUtils3');

    //匹配移动电话号码
    var telReg=/^(13[0-9]|15[0-9]|18[0-9])\d{8}$/;

    export class TelValidator implements validation.StringValidator{
         isAcceptable(s:string){
            return  telReg.test(s);
         }
    }
```

Test-3.ts

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

/***
 * import 与 require 关键字使用..require(是命令，要求的意思.)
 */

//引入 ValidationUtils3.ts 文件,前面这个是用了 module 块关键字定义 ts 文件，需要用 reference 与 path 引入.
import validation = require('test3/ValidationUtils3');
import telValidator = require('test3/TelValidator');
import emailValidator = require('test3/EmailValidator');

//-------------------显示信息 1---------------------
function showMsgs1() : void {
    //--方法一---
    var  telObj=new emailValidator.EmailValidator();//
    var tel : string="13697811809";
    var flag : boolean=telObj.isAcceptable(tel);//调用 TelValidator 类的 isAcceptable 方法
    console.log(flag? tel+" 匹配 " : tel+"\t 不匹配 ");
    $("#msg1").html(flag? "<span style='color:red;'>"+tel+" 匹配</span> " : "<span>"+tel+"\t 不匹配</span>");
}


$(function() {
    //showMsgs1();
var strings = ['13697811809', 'jilongliang@sina.com'];
var validators: { [s: string]: validation.StringValidator; } = {};
    validators['email'] =new emailValidator.EmailValidator();
    validators['tel'] = new telValidator.TelValidator();

strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ') + name);
    }
```

```
});
});
```

6.4import， export， require 关键字

ValidationUtils4.ts

```
/**
 * 不使用 Module,如果我们在 typescript 使用了 module 函数，则生成的代码在浏览器端执行时，需要有一些 script loader 的支持。
 * 对于浏览器端代码，我们一般生成 amd 风格的代码，所以需要找一个支持 amd 的库放在前端。这样的库有很多
 */

export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

export = 对象 的使用

EmailValidator4.ts

```
    /**export = 对象 的使用*/

    import validation = require('./ValidationUtils4');

    //  匹配 email 正则表达式
    var emailReg = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$/;
     class EmailValidator4 implements validation.StringValidator {
        isAcceptable(s: string) {

            return emailReg.test(s);
        }
    }
    export = EmailValidator4; //export = 对象 的使用
```

TelValidator4.ts

```
   /**export = 对象 的使用*/
    import validation = require('./ValidationUtils4');
```

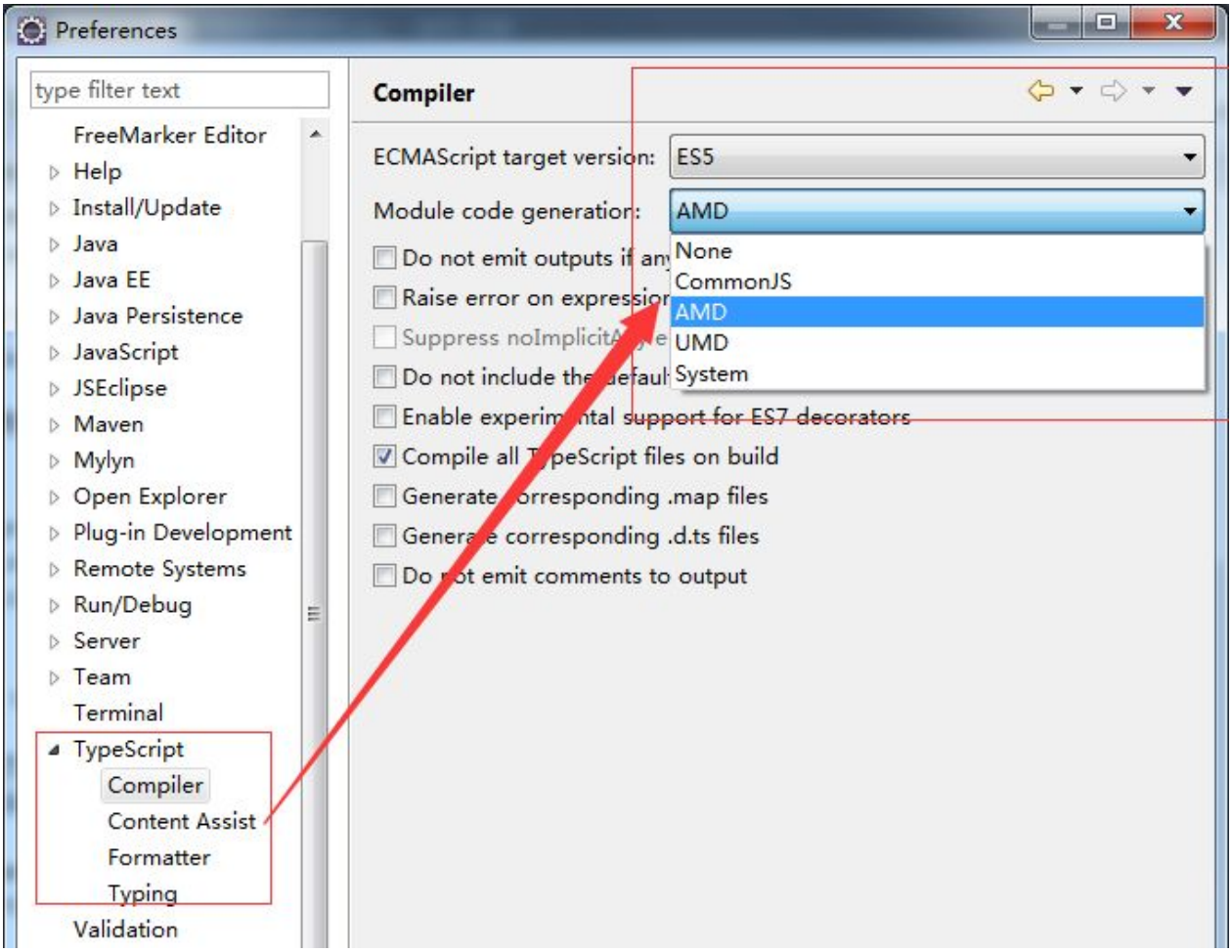
```
    //匹配移动电话号码
    var telReg=/^(13[0-9]|15[0-9]|18[0-9])\d{8}$/;

     class TelValidator4 implements validation.StringValidator{
         isAcceptable(s:string){
            return  telReg.test(s);
         }
    }
    export = TelValidator4;//
```

Test-4.ts 文件

```
/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

//引入 ValidationUtils3.ts 文件,前面这个是用了 module 块关键字定义 ts 文件，需要用 reference 与 path 引入.
import validation = require('test4/ValidationUtils4');
import telValidator = require('test4/TelValidator4');
import emailValidator = require('test4/EmailValidator4');
//-------------------显示信息 1---------------------
function showMsgs1() : void {

    //--方法一---
    var  telObj=new telValidator();//

    var tel : string="13697811809";
    var flag : boolean=telObj.isAcceptable(tel);//调用 TelValidator 类的 isAcceptable 方法

    console.log(flag? tel+" 匹配 " : tel+"\t 不匹配 ");

    $("#msg1").html(flag? "<span style='color:red;'>"+tel+" 匹配</span> " : "<span>"+tel+"\t 不匹配</span>");

}

$(function (){

    showMsgs1();
});
```

注意：6.3 和 6.4 不使用 Module，如果我们在 typescript 使用了 module 函数，则生成的代码在浏览器端执行时，需要有一些 script loader 的支持。对于浏览器端代码，我们一般生成 amd 风格的代码，所以需要找一个支持 amd 的库放在前端，我这里首选的是 AMD 这样的库有很多，比如

1. RequireJS
2. Nodules
3. JSLocalnet
4. curl.js



不然的话运行这个 html 会报 ReferenceError: define is not defined

6.5module 别名的使用

```
//------------别名的使用..
//--声明一个--Shapes 块别名--
module Shapes {
    //============================多边形============================
    export module Polygons {
       //============================三角形============================
       export class Triangle {
           side : number = 3;//声明边一个变量，并且给一个默认值..
           theName : string;//声明一个名字
           //声明构造方法--传一个名字的参数..
           constructor(strName : string) {
               this.theName = strName;
           }
           //计算三角形,获取面积，这里为了返回一个构造方法的传进来的字符串，故返回类型给了一个 any 类型..
           getTriangleArea(side : number) : any{
              return this.theName+ this.side*side;
           }
       }
       //============================正方形============================
       export class Square {
           side : number = 8;//声明边一个变量，并且给一个默认值..
           theName : string;//声明一个名字
           //声明构造方法--传一个名字的参数..
           constructor(strName : string) {
               this.theName = strName;
           }
           //---计算正方形,获取面积
           getSquareArea(side : number) : any{
              return this.theName+ this.side*side;
           }
       }
   }
}
```

ts 编译成 js 文件

```
//------------别名的使用..
//--声明一个--Shapes 块别名--
```

```
var Shapes;
(function (Shapes) {
    //============================多边形============================
    var Polygons;
    (function (Polygons) {
        //============================三角形============================
        var Triangle = (function () {
            //声明构造方法--传一个名字的参数..
            function Triangle(strName) {
                this.side = 3; //声明边一个变量，并且给一个默认值..
                this.theName = strName;
            }
            //计算三角形,获取面积，这里为了返回一个构造方法的传进来的字符串，故返回类型给了一个 any 类型..
            Triangle.prototype.getTriangleArea = function (side) {
                return this.theName + this.side * side;
            };
            return Triangle;
        })();
        Polygons.Triangle = Triangle;
        //============================正方形============================
        var Square = (function () {
            //声明构造方法--传一个名字的参数..
            function Square(strName) {
                this.side = 8; //声明边一个变量，并且给一个默认值..
                this.theName = strName;
            }
            //---计算正方形,获取面积
            Square.prototype.getSquareArea = function (side) {
                return this.theName + this.side * side;
            };
            return Square;
        })();
        Polygons.Square = Square;
    })(Polygons = Shapes.Polygons || (Shapes.Polygons = {}));
})(Shapes || (Shapes = {}));
```

Test.ts 文件

```
//-------import 入 module 块
```

```
import polygons = Shapes.Polygons;

var tg = new polygons.Triangle("三角形面积是： ");
var triangleArea=tg.getTriangleArea(3);//传一个 3 进去..


var sq = new polygons.Square("正方形面积是： ");
var squareArea = sq.getSquareArea(8);


$(function () {

    $("#msg1").html("<span style='color:red;'>"+triangleArea+"</span>");
    $("#msg2").html("<span style='color:red;'>"+squareArea+"</span>");

})
```

6.6module 内部模块

D3.d.ts 文件，**.d.ts** 文件不会编译成 js 文件

```
//环境内部模块

declare module D3{
   //声明一个 Selectors 选择器接口
   export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }
     //声明一个 Event 事件
    export interface Event {
        x: number;
        y: number;
    }
     //声明一个 Base 接口继承 Selectors 接口
    export interface Base extends Selectors {
```

```
        event: Event;
    }
}

declare var d3: D3.Base;
```

6.7module 外部模块

node.d.ts

```
//环境外部模块

//在 node.js 中，大多数的任务是由加载一个或多个模块来实现的。我们可以定义自己的.d.ts 文件顶层出口报关单每个模块，但它更方便他们写为一个较大的.d.ts 文件。要做到这一点，我们使用了模块的引用名，这将提供给一个后来进口
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }
    export function parse(urlStr: string, parseQueryString?,
slashesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
}
```

```
//引入 node.d.ts 文件
///<reference path="test6/node.d.ts"/>

import url = require("url");

var myUrl = url.parse("http://www.typescriptlang.org");
```

```
$(function() {

    alert(myUrl);
    //$("#msg1").html(myUrl);
});
```

七、TypeScript 函数

7.1 最简单 function 函数

Ts 代码

```
/*******声明一个 add 方法********/
function add(x: number, y: number): number {
    return x+y;
}

/*******声明一个 myAdd1 方法********/
var myAdd1 = function(x: number, y: number): number {
    return x+y;
};
/*******声明一个 myAdd2 方法********/
//现在我们已经输入的功能，让我们写了完整类型的功能出来通过查看每件功能类型。
var myAdd2: (x:number, y:number)=>number = function(x: number, y: number): number
{
    return x+y;
};

var number3=myAdd2(1,3);

$(function () {

    var number1=add(1,2);
    var number2=myAdd1(1,2);
```

```
    $("#msg1").html("<span style='color:red;'>"+number1+"</span>");
    $("#msg2").html("<span style='color:red;'>"+number2+"</span>");
    $("#msg3").html("<span style='color:red;'>"+number3+"</span>");

});
```

Js 代码

```
/*******声明一个 add 方法********/
function add(x, y) {
    return x + y;
}
/*******声明一个 myAdd1 方法********/
var myAdd1 = function (x, y) {
    return x + y;
};
/*******声明一个 myAdd2 方法********/
//现在我们已经输入的功能，让我们写了完整类型的功能出来通过查看每件功能类型。
var myAdd2 = function (x, y) {
    return x + y;
};
var number3 = myAdd2(1, 3);
$(function () {
    var number1 = add(1, 2);
    var number2 = myAdd1(1, 2);
    $("#msg1").html("<span style='color:red;'>" + number1 + "</span>");
    $("#msg2").html("<span style='color:red;'>" + number2 + "</span>");
    $("#msg3").html("<span style='color:red;'>" + number3 + "</span>");
});
```

Html 代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script src="../test-1.js" type="text/javascript"></script>
```

```
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
    <br/>
    <div id="msg3"></div>
</body>
</html>
```

7.2 函数其余的参数

Ts 文件代码

```
/**
 * 一、function 没有返回值，却返回了,虽然在写 function 的的时候不报错，调用的时候就报 Error
 * 不像 JavaScript 中，在打字稿每参数的函数被假定为所要求的功能。这并不意味着它不是一个“空”值，
 * 而是，当函数调用编译器将检查该用户已经提供了对每个参数的值。编译器还假定这些参数是将被传递给函数的唯一参数。
 * 总之，参数的函数的数目必须匹配的参数的函数需要的数量。
 */
function buildName1(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

//var result1 = buildName1("Bob");  //error, too few parameters
//var result2 = buildName1("Bob", "Adams", "Sr.");  //error, too many parameters
//var result3 = buildName1("Bob", "Adams");  //ah, just right

/**
 * Rest parameters
 * 其余的参数
 *在 JavaScript 中，每一个参数被认为是可选的，用户可以不用管它，因为他们认为合适的。当他们这样做，他们认为是不确定的。
 *我们可以通过使用得到打字稿这个功能'？“旁边的参数，我们想要可选。例如，我们说，我们要
```

```
的姓氏是可选：
 */
function buildName2(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

var result1 = buildName2("Bob");  //works correctly now
//var result2 = buildName2("Bob", "Adams", "Sr.");  //error, too many parameters
var result3 = buildName2("Bob", "Adams");  //ah, just right



/**
 * 必需，可选和默认参数都有一个共同点：他们大约在同一时间谈论一个参数。有时候，你想与多个参数的工作作为一个群体
 * ，或者你可能不知道有多少参数的函数将最终取,在 JavaScript 中，你可以使用的参数变量，它是每一个函数体中可见
 * 在 TypeScript：您可以收集这些参数汇集成一个变量
 */

function buildName3(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

var employeeName = buildName3("Joseph", "Samuel", "Lucas", "MacKinzie");
/**
 * Rest parameters
 * 其余的参数都被视为可选参数无边号码。你可能会离开他们，或者你想要的。编译器将建立你的下省略号（...）
 * 后给予的名称传递给函数的参数数组，允许你使用它在你的函数
 */
function buildName4(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

var buildNameFun: (fname: string, ...rest: string[])=>string = buildName4;
```

```
/**
 * 一、function 没有返回值，却返回了,虽然在写 function 的的时候不报错，调用的时候就报 Error
 * 不像 JavaScript 中，在打字稿每参数的函数被假定为所要求的功能。这并不意味着它不是一个“空”值，
 * 而是，当函数调用编译器将检查该用户已经提供了对每个参数的值。编译器还假定这些参数是将被传递给函数的唯一参数。
 * 总之，参数的函数的数目必须匹配的参数的函数需要的数量。
 */
function buildName1(firstName, lastName) {
    return firstName + " " + lastName;
}
//var result1 = buildName1("Bob");  //error, too few parameters
//var result2 = buildName1("Bob", "Adams", "Sr.");  //error, too many parameters
//var result3 = buildName1("Bob", "Adams");  //ah, just right
/**
 * Rest parameters
 * 其余的参数
 *在 JavaScript 中，每一个参数被认为是可选的，用户可以不用管它，因为他们认为合适的。当他们这样做，他们认为是不确定的。
 *我们可以通过使用得到打字稿这个功能'？“旁边的参数，我们想要可选。例如，我们说，我们要的姓氏是可选：
 */
function buildName2(firstName, lastName) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
var result1 = buildName2("Bob"); //works correctly now
//var result2 = buildName2("Bob", "Adams", "Sr.");  //error, too many parameters
var result3 = buildName2("Bob", "Adams"); //ah, just right
/**
 * 必需，可选和默认参数都有一个共同点：他们大约在同一时间谈论一个参数。有时候，你想与多个参数的工作作为一个群体
 * ，或者你可能不知道有多少参数的函数将最终取,在 JavaScript 中，你可以使用的参数变量，它是每一个函数体中可见
 * 在 TypeScript：您可以收集这些参数汇集成一个变量
 */
```

```
function buildName3(firstName) {
    var restOfName = [];
    for (var _i = 1; _i < arguments.length; _i++) {
        restOfName[_i - 1] = arguments[_i];
    }
    return firstName + " " + restOfName.join(" ");
}
var employeeName = buildName3("Joseph", "Samuel", "Lucas", "MacKinzie");
/**
 * Rest parameters
 * 其余的参数都被视为可选参数无边号码。你可能会离开他们，或者你想要的。编译器将建立你的下省略号（...）
 * 后给予的名称传递给函数的参数数组，允许你使用它在你的函数
 */
function buildName4(firstName) {
    var restOfName = [];
    for (var _i = 1; _i < arguments.length; _i++) {
        restOfName[_i - 1] = arguments[_i];
    }
    return firstName + " " + restOfName.join(" ");
}
var buildNameFun = buildName4;
```

7.3 函数 this 关键字

Ts 代码

```
/**
 * this 的使用..
 *在 JavaScript，this 的变量一个函数被调用的变量。这使得它成为非常强大和灵活的功能，
 *但它是在总是具有了解，一个功能被执行的情况下的成本。这可以是出了名的混乱，例如，当一个函数被用作一个回调。
 */
function getMeMsg1():void{
    var suits1=["hearts", "spades", "clubs", "diamonds"];

    var deck1 = {
        suits:suits1,
        cards: Array(52),
```

```
        createCardPicker: function() {
            return function() {
                var pickedCard = Math.floor(Math.random() * 52);
                var pickedSuit = Math.floor(pickedCard / 13);
                return {suit: this.suits[pickedSuit], card: pickedCard % 13};
            }
        }
    }
    var cardPicker1 = deck1.createCardPicker();
    var pickedCard1 = cardPicker1();
    alert("card1: " + pickedCard1.card + " of " + pickedCard1.suit);
}

/**
 * 我们可以通过确保修复此功能被绑定到正确的'this'之前，我们返回函数供以后使用。这样一来，
 * 不管如何其以后使用时，它仍可以看到原来的“甲板”对象
 * 为了解决这个问题，我们切换函数表达式使用 lambda 语法（（）=>{}），而不是 JavaScript 函数表达式。
 * 这将自动捕捉“这个”可被创建，而不是被调用时，它的功能时：
 */

function getMeMsg2():void{
    var deck2 = {
        suits: ["hearts", "spades", "clubs", "diamonds"],
        cards: Array(52),
        createCardPicker: function() {
            //使用 lambda 表达式去捕捉 this 用法.
            return () => {
                var pickedCard = Math.floor(Math.random() * 52);
                var pickedSuit = Math.floor(pickedCard / 13);
                console.log(this.suits[pickedSuit]);
                console.log(pickedCard % 13);

                return {suit: this.suits[pickedSuit], card: pickedCard % 13};
            }
        }
    }

    var cardPicker2 = deck2.createCardPicker();
```

```
    var pickedCard2 = cardPicker2();

    alert("card2: " + pickedCard2.card + " of " + pickedCard2.suit);

}
$(function(){
    getMeMsg2();
});
```

Ts 编译成的 js 代码

```
/**
 * this 的使用..
 *在 JavaScript，this 的变量一个函数被调用的变量。这使得它成为非常强大和灵活的功能，
 *但它是在总是具有了解，一个功能被执行的情况下的成本。这可以是出了名的混乱，例如，当一
个函数被用作一个回调。
 */
function getMeMsg1() {
    var suits1 = ["hearts", "spades", "clubs", "diamonds"];
    var deck1 = {
        suits: suits1,
        cards: Array(52),
        createCardPicker: function () {
            return function () {
                var pickedCard = Math.floor(Math.random() * 52);
                var pickedSuit = Math.floor(pickedCard / 13);
                return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
            };
        }
    };
    var cardPicker1 = deck1.createCardPicker();
    var pickedCard1 = cardPicker1();
    alert("card1: " + pickedCard1.card + " of " + pickedCard1.suit);
}
/**
 * 我们可以通过确保修复此功能被绑定到正确的'this'之前，我们返回函数供以后使用。这样一
来，
 * 不管如何其以后使用时，它仍可以看到原来的“甲板”对象
 * 为了解决这个问题，我们切换函数表达式使用 lambda 语法（（）=>{}），而不是 JavaScript
函数表达式。
 * 这将自动捕捉“这个”可被创建，而不是被调用时，它的功能时：
```

```
 */
function getMeMsg2() {
    var deck2 = {
        suits: ["hearts", "spades", "clubs", "diamonds"],
        cards: Array(52),
        createCardPicker: function () {
            var _this = this;
            //使用 lambda 表达式去捕捉 this 用法.
            return function () {
                var pickedCard = Math.floor(Math.random() * 52);
                var pickedSuit = Math.floor(pickedCard / 13);
                console.log(_this.suits[pickedSuit]);
                console.log(pickedCard % 13);
                return { suit: _this.suits[pickedSuit], card: pickedCard % 13 };
            };
        }
    };
    var cardPicker2 = deck2.createCardPicker();
    var pickedCard2 = cardPicker2();
    alert("card2: " + pickedCard2.card + " of " + pickedCard2.suit);
}
$(function () {
    getMeMsg2();
});
```

Ts 代码

```
/**
 *JavaScript 是本质上是一种非常好的动态的语言。这并意味都通用的一个 JavaScript 函数返回基于对传入的参数的形状不同类型的对象。
 */
var suits = ["hearts", "spades", "clubs", "diamonds"];

//--实例一
function getThisMsgs1():void{

        function pickCard1(x): any {
        // 判断这个 x 是不是 object/array
        if (typeof x == "object") {
            var pickedCard = Math.floor(Math.random() * x.length);
            return pickedCard;
```

```
        }
        //  判断
        else if (typeof x == "number") {
            var pickedSuit = Math.floor(x / 13);
            return { suit: suits[pickedSuit], card: x % 13 };
        }
    }

    var myDeck1 = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 },
{ suit: "hearts", card: 4 }];
    var pickedCard1 = myDeck1[pickCard1(myDeck1)];
    alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

    var pickedCard2 = pickCard1(15);
    alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
    }

//--实例二
function getThisMsgs2():void{
    function pickCard(x: {suit: string; card: number; }[]): number;
    function pickCard(x: number): {suit: string; card: number; };
    function pickCard(x): any {
        // Check to see if we're working with an object/array
        // if so, they gave us the deck and we'll pick the card
        if (typeof x == "object") {
            var pickedCard = Math.floor(Math.random() * x.length);
            return pickedCard;
        }
        // Otherwise just let them pick the card
        else if (typeof x == "number") {
            var pickedSuit = Math.floor(x / 13);
            return { suit: suits[pickedSuit], card: x % 13 };
        }
    }

    var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit:
"hearts", card: 4 }];
    var pickedCard1 = myDeck[pickCard(myDeck)];
    alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
```

```
    var pickedCard2 = pickCard(15);
    alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
}

$(function(){
    getThisMsgs2();

});
```

Ts 编译成 js 代码

```
/**
 *
 *JavaScript 是本质上是一种非常好的动态的语言。这并意味都通用的一个 JavaScript 函数返回基于对传入的参数的形状不同类型的对象。
 *
 */
var suits = ["hearts", "spades", "clubs", "diamonds"];
//--实例一
function getThisMsgs1() {
    function pickCard1(x) {
        // 判断这个 x 是不是 object/array
        if (typeof x == "object") {
            var pickedCard = Math.floor(Math.random() * x.length);
            return pickedCard;
        }
        else if (typeof x == "number") {
            var pickedSuit = Math.floor(x / 13);
            return { suit: suits[pickedSuit], card: x % 13 };
        }
    }
    var myDeck1 = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 },
{ suit: "hearts", card: 4 }];
    var pickedCard1 = myDeck1[pickCard1(myDeck1)];
    alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
    var pickedCard2 = pickCard1(15);
    alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
}
//--实例二
function getThisMsgs2() {
    function pickCard(x) {
```

```
        // Check to see if we're working with an object/array
        // if so, they gave us the deck and we'll pick the card
        if (typeof x == "object") {
            var pickedCard = Math.floor(Math.random() * x.length);
            return pickedCard;
        }
        else if (typeof x == "number") {
            var pickedSuit = Math.floor(x / 13);
            return { suit: suits[pickedSuit], card: x % 13 };
        }
    }
    var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit:
"hearts", card: 4 }];
    var pickedCard1 = myDeck[pickCard(myDeck)];
    alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
    var pickedCard2 = pickCard(15);
    alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
}
$(function () {
    getThisMsgs2();
});
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script src="../test-3.js" type="text/javascript"></script>
<script src="../test-4.js" type="text/javascript"></script>
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
    <br/>
    <div id="msg3"></div>
</body>
</html>
```

八、TypeScript 泛型

8.1 最简单泛型例子

Ts 代码

```
/**
 * 没有泛型，我们要么必须给身份功能的特定类型
 */

function identity1(arg: number): number {
    return arg;
}
/**
 * 或者:我们可以描述使用“任意”类型的标识功能:
 */
function identity2(arg: any): any {
    return arg;
}
```

Js 文件

```
/**
 * 没有泛型，我们要么必须给身份功能的特定类型
 */
function identity1(arg) {
    return arg;
}
/**
 * 或者:我们可以描述使用“任意”类型的标识功能:
 */
function identity2(arg) {
    return arg;
}
```

8.2 泛型类型与接口

Ts 代码一

```
/**
```

```
 * Working with Generic Type Variables
 * 与泛型类型变量
 */
 function _identity1<T>(arg: T): T {
    return arg;
}

/**
 * 如果还想记录的说法“arg”与每个调用控制台的长度。我们也许会这样写
 */
function loggingIdentity1<T>(arg: T): T {
   // console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
/**
 *当我们这样做时，编译器会给出我们，我们使用的是“ARG”的“.length”成员的错误，但远不具备，我们说，“ARG”有这样的成员。请记住，我们在前面说，这些类型变量站在任何和所有类型的，所以有人使用此功能可以通过在一个'数'，而不是，它没有一个“.length”成员。
 *比方说，我们实际上已经预期该功能直接对 T 的阵列，而那件 T 的工作。由于我们正在使用的阵列，对象.length 成员应该可用。我们可以这样描述这就像我们将创建其他类型的数组：
 */

function loggingIdentity2<T>(arg: T[]): T[] {
    console.log(arg.length);
    return arg;
}

/**
 *我们还可以写成这样的模式 ， 一个数组.length,这样避免更多的错误
 */
function loggingIdentity3<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);
    return arg;
}
```

Ts 编译 js 代码一

```
/**
 * Working with Generic Type Variables
 * 与泛型类型变量
```

```
 */
function _identity1(arg) {
    return arg;
}
/**
 * 如果还想记录的说法“arg”与每个调用控制台的长度。我们也许会这样写
 */
function loggingIdentity1(arg) {
    // console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
/**
 *当我们这样做时，编译器会给出我们，我们使用的是“ARG”的“.length”成员的错误，但远不具备，我们说，“ARG”有这样的成员。请记住，我们在前面说，这些类型变量站在任何和所有类型的，所以有人使用此功能可以通过在一个'数'，而不是，它没有一个“.length”成员。
 *比方说，我们实际上已经预期该功能直接对 T 的阵列，而那件 T 的工作。由于我们正在使用的阵列，对象.length 成员应该可用。我们可以这样描述这就像我们将创建其他类型的数组：
 */
function loggingIdentity2(arg) {
    console.log(arg.length);
    return arg;
}
/**
 *我们还可以写成这样的模式 ，一个数组.length,这样避免更多的错误
 */
function loggingIdentity3(arg) {
    console.log(arg.length);
    return arg;
}
```

Ts 代码二

```
/**
 *在前面的章节中，我们创建了工作的范围内的类型的通用身份的功能。在本节中，我们将探讨的功能类型本身，
 *以及如何创建通用接口.通用函数的类型就像那些非通用功能，具有类型参数首家上市，类似于函数声明
 */
function identity3<T>(arg: T): T {
    return arg;
}
var myIdentity3: <T>(arg: T)=>T = identity3;
```

```
/**
 * 我们也可以用不同的名称在类型一般类型参数，所以只要类型变量的数量和如何类型变量用于排队
 */
function identity4<T>(arg: T): T {
    return arg;
}
var myIdentity4: <U>(arg: U)=>U = identity4;

/**
 * 我们也可以写泛型类型为对象文本类型的调用签名
 */
function identity5<T>(arg: T): T {
    return arg;
}
var myIdentity5: {<T>(arg: T): T} = identity5;

/**
 * 这使我们写我们的第一个通用 interface 接口。让我们以字面对象从以前的例子，它移动到一个界面：
 */

interface GenericIdentityFn1 {
    <T>(arg: T): T;
}
function identity6<T>(arg: T): T {
    return arg;
}
var myIdentity6: GenericIdentityFn1 = identity6;

/**
 * 在一个类似的例子，我们可能要移动的通用参数是整个接口的参数。这让我们看到什么类型，我们是在通用
 * （如：Dictionary<String>而不仅仅是字典）。这使得该类型参数可见的接口的所有其他成员。
 */
interface GenericIdentityFn2<T> {
    (arg: T): T;
}
```

```
function identity7<T>(arg: T): T {
    return arg;
}
var myIdentity7: GenericIdentityFn2<number> = identity7;
```

Ts 编译 js 代码二

```
/**
 *
 *在前面的章节中，我们创建了工作的范围内的类型的通用身份的功能。在本节中，我们将探讨的功能类型本身，
 *以及如何创建通用接口.通用函数的类型就像那些非通用功能，具有类型参数首家上市，类似于函数声明
 */
function identity3(arg) {
    return arg;
}
var myIdentity3 = identity3;
/**
 * 我们也可以用不同的名称在类型一般类型参数，所以只要类型变量的数量和如何类型变量用于排队
 */
function identity4(arg) {
    return arg;
}
var myIdentity4 = identity4;
/**
 * 我们也可以写泛型类型为对象文本类型的调用签名
 */
function identity5(arg) {
    return arg;
}
var myIdentity5 = identity5;
function identity6(arg) {
    return arg;
}
var myIdentity6 = identity6;
function identity7(arg) {
    return arg;
}
var myIdentity7 = identity7;
```

8.3 泛型类型与类

Ts 代码

```
//泛型类也有类似形状的通用接口。泛型类在尖括号泛型类型参数列表
//--T
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}
/*------number 数字类型-----*/
var myGenericNumber = new GenericNumber<number>();
    myGenericNumber.zeroValue = 0;
    myGenericNumber.add = function(x, y) { return x + y; };
/*------string 字符串类型-----*/
var stringNumeric = new GenericNumber<string>();
    stringNumeric.zeroValue = "";
    stringNumeric.add = function(x, y) { return x + y; };
alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
//-----------------Generic Constraints----

/*****声明一个接口,来约束**********/
interface ILength {
    length: number;//声明一个 number 类型
}
//-function 用 extends 关键继承这个 ILength 接口约束。。
function loggingIdentity<T extends ILength>(arg: T): T {
    console.log(arg.length);  //获取这个 length 值
    return arg;
}
//调用这个 loggingIdentity 方法
var object=loggingIdentity({length: 10, value: 3});
/**
 *在使用泛型类的类型
 *当我们用 typescript 去创建工厂的时候，因此有必要通过其构造函数来引用类类型
 */
function create<T>(c: {new(): T; }): T {
    return new c();
}

/**
```

```
 *使用:一个更高级的示例使用原型属性来推断和约束的构造函数和类类型的实例侧之间的关系
 */
//养蜂人
class BeeKeeper {
    hasMask: boolean;
}
//动物管理人.
class ZooKeeper {
    nametag: string;
}
//动物
class Animals {
    numLegs: number;
}
//蜜蜂
class Bee extends Animals {
    keeper: BeeKeeper;
}
//狮子
class Lion extends Animals {
    keeper: ZooKeeper;
}
//管理人.
function findKeeper<A extends Animals, K> (a: {new(): A;
    prototype: {keeper: K}}): K {

    return a.prototype.keeper;
}
 //findKeeper(Lion).nametag;  // 检查类型!
/**
 *jQuery----
 *
 */
$(function(){
   var  len=$(object).attr("length");//获取这个 length 值
   var  value=$(object).attr("value");//获取这个 value 值
    //alert(len);
    //alert(value);
    //var obj1:Animals=Lion;
     //console.log( findKeeper(Lion).nametag);//检查类型!
```

```
});
```

Ts 文件编译 js 代码

```
/**
 *Generic Classes
 *
 */
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
//泛型类也有类似形状的通用接口。泛型类在尖括号泛型类型参数列表
//--T
var GenericNumber = (function () {
    function GenericNumber() {
    }
    return GenericNumber;
})();
/*------number 数字类型-----*/
var myGenericNumber = new GenericNumber();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) { return x + y; };
/*------string 字符串类型-----*/
var stringNumeric = new GenericNumber();
stringNumeric.zeroValue = "";
stringNumeric.add = function (x, y) { return x + y; };
alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
//-function 用 extends 关键继承这个 ILength 接口约束。。
function loggingIdentity(arg) {
    console.log(arg.length); //获取这个 length 值
    return arg;
}
//调用这个 loggingIdentity 方法
var object = loggingIdentity({ length: 10, value: 3 });
/**
 *在使用泛型类的类型
 *当我们用 typescript 去创建工厂的时候，因此有必要通过其构造函数来引用类类型
 */
```

```
function create(c) {
    return new c();
}
/**
 *使用:一个更高级的示例使用原型属性来推断和约束的构造函数和类类型的实例侧之间的关系
 */
//养蜂人
var BeeKeeper = (function () {
    function BeeKeeper() {
    }
    return BeeKeeper;
})();
//动物管理人.
var ZooKeeper = (function () {
    function ZooKeeper() {
    }
    return ZooKeeper;
})();
//动物
var Animals = (function () {
    function Animals() {
    }
    return Animals;
})();
//蜜蜂
var Bee = (function (_super) {
    __extends(Bee, _super);
    function Bee() {
        _super.apply(this, arguments);
    }
    return Bee;
})(Animals);
//狮子
var Lion = (function (_super) {
    __extends(Lion, _super);
    function Lion() {
        _super.apply(this, arguments);
    }
    return Lion;
})(Animals);
```

```
//管理人.
function findKeeper(a) {
    return a.prototype.keeper;
}
//findKeeper(Lion).nametag;  // 检查类型!
/**
 *jQuery----
 *
 */
$(function () {
    var len = $(object).attr("length"); //获取这个 length 值
    var value = $(object).attr("value"); //获取这个 value 值
    //alert(len);
    //alert(value);
    //var obj1:Animals=Lion;
    //console.log( findKeeper(Lion).nametag);//检查类型!
});
```

Html 文件测试

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript"
src="../../plugins/jquery-2.1.4.min.js"></script>
<script src="../test-4.js" type="text/javascript"></script>
</head>
<body>
    <div id="msg1"></div>
    <br/>
    <div id="msg2"></div>
</body>
</html>
```

九、TypeScript 混入

9.1Mixin 使用

Ts 代码

```
/**
 * Mixin 使用.
 *
 *随着传统的面向对象的层次结构，从可重用的组件建立类的另一种流行的方式是通过简单的组合部分类来构建他们。
 *你可能熟悉混入或性状比如 Scala 语言的理念，模式也达到了 JavaScript 的一些社区人气
 */

// Disposable Mixin(一次性)
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin(激活混入)
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}

//SmartObject 类实现 Disposable 与 Activatable 类
class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + this.isDisposed),
500);
```

```
    }

    //相互作用
    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable])

var smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////
// In your runtime library somewhere
//在您的运行时库的地方
////////////////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        })
    });
}
```

Js 代码

```
/**
 * Mixin 使用.
 *
 *随着传统的面向对象的层次结构，从可重用的组件建立类的另一种流行的方式是通过简单的组合部分类来构建他们。
 *你可能熟悉混入或性状比如 Scala 语言的理念，模式也达到了 JavaScript 的一些社区人气
```

```
 */
// Disposable Mixin(一次性)

var Disposable = (function () {
    function Disposable() {
    }
    Disposable.prototype.dispose = function () {
        this.isDisposed = true;
    };
    return Disposable;
})();
// Activatable Mixin(激活混入)
var Activatable = (function () {
    function Activatable() {
    }
    Activatable.prototype.activate = function () {
        this.isActive = true;
    };
    Activatable.prototype.deactivate = function () {
        this.isActive = false;
    };
    return Activatable;
})();
//SmartObject 类实现 Disposable 与 Activatable 类
var SmartObject = (function () {
    function SmartObject() {
        var _this = this;
        // Disposable
        this.isDisposed = false;
        // Activatable
        this.isActive = false;
        setInterval(function () { return console.log(_this.isActive + " : " + _this.isDisposed); }, 500);
    }
    //相互作用
    SmartObject.prototype.interact = function () {
        this.activate();
    };
    return SmartObject;
})();
applyMixins(SmartObject, [Disposable, Activatable]);
```

```
var smartObj = new SmartObject();
setTimeout(function () { return smartObj.interact(); }, 1000);
////////////////////////////////////////
// In your runtime library somewhere
//在您的运行时库的地方
////////////////////////////////////////
function applyMixins(derivedCtor, baseCtors) {
    baseCtors.forEach(function (baseCtor) {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(function (name) {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}
```

十、TypeScript 声明合并

10.1 合并 interface 使用

```
/**
 * Merging Interfaces
 * 合并接口..
 * 最简单的，也许是最常见的，类型声明合并是合并接口。将两个相同的接口合并在一块.
 */
interface Box {
    height: number;
    width: number;
}
interface Box {
    scale: number;
}
var box: Box = {height: 5, width: 6, scale: 10};

interface Document {
    createElement(tagName: any): Element;
```

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


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

```
/**
 * Merging Interfaces
 * 合并接口..
 */
var box = { height: 5, width: 6, scale: 10 };
```

10.2 合并 module 使用

```
/**
 * Merging Modules
 * 合并块.
  *要合并的价值，在每一个网站的声明，如果一个模块已经给定名称存在，它进一步采取现有的模块
  *并添加第二个模块第一的出口成员扩展。
  */
module MAnimals {
    export class Zebra { }
}
```

```
module MAnimals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}

//---这个 MYAnimals 块等同上面两个 module
module MYAnimals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}

/**
 * 模块合并的这种模式是一个有用的起点，但要得到我们还需要了解同非成员国导出会发生什么更完整的场景画面。
 * 非导出成员只有原来的（未合并）模块中可见,这意味着合并后，合并的成员，来自其他声明无法看到非成员导出
 */

module myAnimal {
    var haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

module myAnimal {
    export function doAnimalsHaveMuscles() {
        //return haveMuscles;  // 这里这个值是不可以返回.
    }
}
/**
 * Merging Modules with Classes, Functions, and Enums
 * 合并模块与类，函数和枚举
 * 模块具有足够的灵活性，以也与其它类型的声明合并。要做到这一点，该模块声明必须遵循的声明，将与合并。
 * 由此产生的声明有两种申报类型的属性。在 typescript 使用这个性能来模拟一些在
```

```
JavaScript 模式以及其它的编程语言
 */
class Album {
    label: Album.AlbumLabel;
}
module Album {
    export class AlbumLabel { }
}
```

```
/**
 * Merging Modules
 * 合并块.
  *要合并的价值，在每一个网站的声明，如果一个模块已经给定名称存在，它进一步采取现有的模块
  *并添加第二个模块第一的出口成员扩展。
  */
var MAnimals;
(function (MAnimals) {
    var Zebra = (function () {
        function Zebra() {
        }
        return Zebra;
    })();
    MAnimals.Zebra = Zebra;
})(MAnimals || (MAnimals = {}));
var MAnimals;
(function (MAnimals) {
    var Dog = (function () {
        function Dog() {
        }
        return Dog;
    })();
    MAnimals.Dog = Dog;
})(MAnimals || (MAnimals = {}));
//---这个 MYAnimals 块等同上面两个 module
var MYAnimals;
(function (MYAnimals) {
    var Zebra = (function () {
```

```
        function Zebra() {
        }
        return Zebra;
    })();
    MYAnimals.Zebra = Zebra;
    var Dog = (function () {
        function Dog() {
        }
        return Dog;
    })();
    MYAnimals.Dog = Dog;
})(MYAnimals || (MYAnimals = {}));
/**
 * 模块合并的这种模式是一个有用的起点，但要得到我们还需要了解同非成员国导出会发生什么更完整的场景画面。
 * 非导出成员只有原来的（未合并）模块中可见,这意味着合并后，合并的成员，来自其他声明无法看到非成员导出
 */
var myAnimal;
(function (myAnimal) {
    var haveMuscles = true;
    function animalsHaveMuscles() {
        return haveMuscles;
    }
    myAnimal.animalsHaveMuscles = animalsHaveMuscles;
})(myAnimal || (myAnimal = {}));
var myAnimal;
(function (myAnimal) {
    function doAnimalsHaveMuscles() {
        //return haveMuscles;  // 这里这个值是不可以返回.
    }
    myAnimal.doAnimalsHaveMuscles = doAnimalsHaveMuscles;
})(myAnimal || (myAnimal = {}));
/**
 * Merging Modules with Classes, Functions, and Enums
 * 合并模块与类，函数和枚举
 *
 * 模块具有足够的灵活性，以也与其它类型的声明合并。要做到这一点，该模块声明必须遵循的声明，将与合并。
 * 由此产生的声明有两种申报类型的属性。在 typescript 使用这个性能来模拟一些在
```

```
JavaScript 模式以及其它的编程语言
 */
var Album = (function () {
    function Album() {
    }
    return Album;
})();
var Album;
(function (Album) {
    var AlbumLabel = (function () {
        function AlbumLabel() {
        }
        return AlbumLabel;
    })();
    Album.AlbumLabel = AlbumLabel;
})(Album || (Album = {}));
```

十一、TypeScript 类型比较

11.1 类型比较

```
/**
 * Type Compatibility
 * 类型比较.
 */
interface Named {
    name: string;
}
var x: Named;
//判断这个 y 的类型是{ name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y;

var items = [1, 2, 3];
```

```
//不压迫强迫这些额外参数...
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach((item) => console.log(item))

var k = () => ({name: 'Alice'});
var z = () => ({name: 'Alice', location: 'Seattle'});

k = z; // OK
//z = k; //这 k()方法缺少了 location 属性.所以赋给 z()方法是会报错的.
```

```
var x;
//判断这个 y 的类型是{ name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y;
var items = [1, 2, 3];
//不压迫强迫这些额外参数...
items.forEach(function (item, index, array) { return console.log(item); });
// Should be OK!
items.forEach(function (item) { return console.log(item); });
var k = function () { return ({ name: 'Alice' }); };
var z = function () { return ({ name: 'Alice', location: 'Seattle' }); };
k = z; // OK
//z = k; //这 k()方法缺少了 location 属性.所以赋给 z()方法是会报错的.
```

十二、总结

本文档是参考 TypeScript 官方手册去学习，在学习过程中多少都会有出入，如需详细文档请参看，Typescript 官方，在学习过程中要懂得怎么去解决，这样才能进步，希望指出错误与建议，同时也希望我整理出来的文档能帮助大家伙学习与进步，让我们在编程学习道路上过程中一起共同进步,最后俺送大家伙一句话:内事问度娘,外事问谷歌，谢谢！手机微信扫一扫有惊喜.