

# TypeScript

## Handbook (中文版)

Patrick Zhong

Published  
with GitBook



# 目錄

介紹	0
快速上手	1
React与webpack	1.1
ASP.NET 4	1.2
ASP.NET Core	1.3
Knockout.js	1.4
新增功能	2
TypeScript 1.8	2.1
TypeScript 1.7	2.2
TypeScript 1.6	2.3
TypeScript 1.5	2.4
TypeScript 1.4	2.5
TypeScript 1.3	2.6
TypeScript 1.1	2.7
手冊	3
基础类型	3.1
变量声明	3.2
接口	3.3
类	3.4
函数	3.5
泛型	3.6
枚举	3.7
类型推论	3.8
类型兼容性	3.9
高级类型	3.10
Symbols	3.11
Iterators 和 Generators	3.12

模块	3.13
命名空间	3.14
命名空间和模块	3.15
模块解析	3.16
声明合并	3.17
书写.d.ts文件	3.18
JSX	3.19
Decorators	3.20
混入	3.21
三斜线指令	3.22
工程配置	4
tsconfig.json	4.1
NPM包的类型	4.2
编译选项	4.3
在MSBuild里使用编译选项	4.4
与其它构建工具整合	4.5
使用TypeScript的每日构建版本	4.6
Wiki	5
TypeScript里的this	5.1
编码规范	5.2
常见编译错误	5.3
支持TypeScript的编辑器	5.4
结合ASP.NET v5使用TypeScript	5.5
架构概述	5.6
发展路线图	5.7

# TypeScript Handbook (中文版)

从前打心眼儿里讨厌编译成JavaScript的这类语言，像Coffee，Dart等。但是在15年春节前后却爱上了TypeScript。同时非常喜欢的框架Dojo，Angularjs也宣布使用TypeScript做新版本的开发。那么TypeScript究竟为何物？又有什么魅力呢？

TypeScript是Microsoft公司注册商标。

TypeScript具有类型系统，且是JavaScript的超集。它可以编译成普通的JavaScript代码。TypeScript支持任意浏览器，任意环境，任意系统并且是开源的。

TypeScript目前还在积极的开发完善之中，不断地会有新的特性加入进来。因此本手册也会紧随官方的每个commit，不断地更新新的章节以及修改措词不妥之处。

如果你对TypeScript一见钟情，可以订阅[and star](#)本手册，及时了解ECMAScript 2015以及2016里新的原生特性，并借助TypeScript提前掌握使用它们的方式！如果你对TypeScript的爱愈发浓烈，可以与楼主一起边翻译边学习，[PRs Welcome!!!](#)在[相关链接](#)的末尾可以找到本手册的[Github地址](#)。

## 目录

- [快速上手](#)
  - [React与webpack](#)
  - [ASP.NET 4](#)
  - [ASP.NET Core](#)
  - [Knockout.js](#)
- [新增功能](#)
  - [TypeScript 1.8](#)
  - [TypeScript 1.7](#)
  - [TypeScript 1.6](#)
  - [TypeScript 1.5](#)
  - [TypeScript 1.4](#)
  - [TypeScript 1.3](#)
  - [TypeScript 1.1](#)
- [手册](#)

- [基础类型](#)
- [变量声明](#)
- [接口](#)
- [类](#)
- [函数](#)
- [泛型](#)
- [枚举](#)
- [类型推论](#)
- [类型兼容性](#)
- [高级类型](#)
- [Symbols](#)
- [Iterators 和 Generators](#)
- [模块](#)
- [命名空间](#)
- [命名空间和模块](#)
- [模块解析](#)
- [声明合并](#)
- [书写.d.ts文件](#)
- [JSX](#)
- [Decorators](#)
- [混入](#)
- [三斜线指令](#)
- [工程配置](#)
  - [tsconfig.json](#)
  - [NPM包的类型](#)
  - [编译选项](#)
  - [在MSBuild里使用编译选项](#)
  - [与其它构建工具整合](#)
  - [使用TypeScript的每日构建版本](#)
- [Wiki](#)
  - [TypeScript里的this](#)
  - [编码规范](#)
  - [常见编译错误](#)
  - [支持TypeScript的编辑器](#)
  - [结合ASP.NET v5使用TypeScript](#)
  - [架构概述](#)

- [发展路线图](#)

## 主要修改 (Latest 10 updates)

- 2016-04-23 新增章节：[使用TypeScript的每日构建版本](#)
- 2016-04-18 新增章节：[新增功能](#)
- 2016-04-10 新增章节：[快速上手：ASP.NET Core](#)
- 2016-04-10 新增章节：[三斜线指令](#)
- 2016-04-10 新增章节：[快速上手：Knockout.js](#)
- 2016-04-10 新增章节：[快速上手：新增功能](#)
- 2016-04-02 新增章节：[模块解析](#)
- 2016-04-01 新增特性：[多态的 `this` 类型](#)
- 2016-04-01 新增特性：[字符串字面量类型](#)
- 2016-02-27 新增章节：[快速上手：React和webpack](#)

## 相关链接

- [TypeScript官网](#)
- [TypeScript on Github](#)
- [TypeScript语言规范](#)
- [本手册中文版Github地址](#)

这个快速上手指南将会教你如何将TypeScript和React还有webpack连结在一起使用。

我们假设已经在使用Node.js和npm。

## 初始化项目结构

让我们新建一个目录。将会命名为 `proj`，但是你可以改成任何你喜欢的名字。

```
mkdir proj
cd proj
```

我们会像下面的结构组织我们的工程：

```
proj/
+- src/
|   +- components/
|
+- dist/
```

TypeScript文件会放在 `src` 文件夹里，通过TypeScript编译器编译，然后经webpack处理，最后生成一个 `bundle.js` 文件放在 `dist` 目录下。我们自定义的组件将会放在 `src/components` 文件夹下。

下面来创建基本结构：

```
mkdir src
cd src
mkdir components
cd ..
mkdir dist
```

## 初始化工程

现在把这个目录变成npm包。

```
npm init
```

你会看到一些提示。你可以使用默认项除了开始脚本。使用 `./lib/bundle.js` 做为开始脚本。当然，你也可以随时到生成的 `package.json` 文件里修改。

## 安装依赖

首先确保TypeScript，typings和webpack已经全局安装了。

```
npm install -g typescript typings webpack
```

Webpack这个工具可以将你的所有代码和可选择地将依赖捆绑成一个单独的 `.js` 文件。[Typings](#)是一个包管理器，它是用来获取定义文件的。

现在我们添加React和React-DOM依赖到 `package.json` 文件里：

```
npm install --save react react-dom
```

接下来，我们要添加开发时依赖[ts-loader](#)和[source-map-loader](#)。

```
npm install --save-dev ts-loader source-map-loader  
npm link typescript
```

这些依赖会让TypeScript和webpack在一起良好地工作。`ts-loader`可以让webpack使用TypeScript的标准配置文件 `tsconfig.json` 编译TypeScript代码。`source-map-loader`使用TypeScript输出的sourcemap文件来告诉webpack何时生成自己的sourcemaps。这就允许你在调试最终生成的文件时就好像在调试TypeScript源码一样。

链接TypeScript，允许ts-loader使用全局安装的TypeScript，而不需要单独的本地拷贝。如果你想要一个本地的拷贝，执行 `npm install typescript`。

最后，我们使用 `typings` 工具来获取React的声明文件：



```
typings install --ambient --save react
typings install --ambient --save react-dom
```

`--ambient` 标记告诉`typings`从[DefinitelyTyped](#)获取声明文件，这是由社区维护的 `.d.ts` 文件仓库。这个命令会创建一个名为 `typings.json` 的文件和一个 `typings` 目录在当前目录下。

## 写一些代码

下面使用`React`写一段`TypeScript`代码。首先，在 `src/components` 目录下创建一个名为 `Hello.tsx` 的文件，代码如下：

```
import * as React from "react";
import * as ReactDOM from "react-dom";

export class HelloComponent extends React.Component<any, any> {
  render() {
    return <h1>Hello from {this.props.compiler} and {this.props.framework}!
  }
}
```

注意一点这个例子已经很像类了，我们不再需要使用类。使用`React`的其它方式（比如[无状态的功能组件](#)）。

接下来，在 `src` 下创建 `index.tsx` 文件，源码如下：

```
import * as React from "react";
import * as ReactDOM from "react-dom";

import { HelloComponent } from "../components/Hello";

ReactDOM.render(
  <HelloComponent compiler="TypeScript" framework="React" />,
  document.getElementById("example")
);
```

我们仅仅将 `Hello` 组件导入 `index.tsx`。注意，不同于 `"react"` 或 `"react-dom"`，我们使用 `index.tsx` 的相对路径 - 这很重要。如果不这样做，TypeScript 只会尝试在 `node_modules` 文件夹里查找。其它使用 React 的方法也应该可以。

我们还需要一个页面来显示 `Hello` 组件。在根目录 `proj` 创建一个名为 `index.html` 的文件，如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
  </head>
  <body>
    <div id="example"></div>

    <!-- Dependencies -->
    <script src="./node_modules/react/dist/react.js"></script>
    <script src="./node_modules/react-dom/dist/react-dom.js"></script>

    <!-- Main -->
    <script src="./dist/bundle.js"></script>
  </body>
</html>
```

需要注意一点我们是从 `node_modules` 引入的文件。React 和 React-DOM 的 npm 包里包含了独立的 `.js` 文件，你可以在页面上引入它们，这里我们为了快捷就直接引用了。可以随意地将它们拷贝到其它目录下，或者从 CDN 上引用。Facebook 在 CDN 上提供了一系列可用的 React 版本，你可以在这里查看[更多内容](#)。

## 添加 TypeScript 配置文件

现在，可以把所有 TypeScript 文件放在一起 - 包括我们编写的代码和必要的 typings 文件。

现在需要创建 `tsconfig.json` 文件，它包含输入文件的列表和编译选项。在根目录下执行下在命令：

```
tsc --init ./typings/main.d.ts ./src/index.tsx --jsx react --outDir
```

你可以在[这里](#)学习到更多关于 `tsconfig.json` 。

## 创建webpack配置文件

新建一个 `webpack.config.js` 文件在工程根目录下。

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "./dist/bundle.js",
  },

  // Enable sourcemaps for debugging webpack's output.
  devtool: "source-map",

  resolve: {
    // Add '.ts' and '.tsx' as resolvable extensions.
    extensions: [ "", ".webpack.js", ".web.js", ".ts", ".tsx", ".json" ],
  },

  module: {
    loaders: [
      // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
      { test: /\.tsx?$/, loader: "ts-loader" },

      // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
      { test: /\.js$/, loader: "source-map-loader" }
    ],
  },

  // When importing a module whose path matches one of the following,
  // assume a corresponding global variable exists and use that instead of
  // defaulting to an empty module (this is important because it allows us to
  // avoid bundling all dependencies, which allows browsers to cache those libraries).
  externals: {
    "react": "React",
    "react-dom": "ReactDOM"
  },
};
```

大家可能对 `externals` 字段有所疑惑。我们想要避免把所有的`React`都放到一个文件里，因为会增加编译时间并且浏览器还能够缓存没有发生改变的库文件。理想情况下，我们只需要在浏览器里引入`React`模块，但是大部分浏览器还没有支持模块。因此大部分代码库会把自己包裹在一个单独的全局变量内，比

如：`jQuery` 或 `_`。这叫做“命名空间”模式，`webpack`也允许我们继续使用通过这种方式写的代码库。通过我们的设置 `"react": "React"`，`webpack`会神奇地将所有对 `"react"` 的导入转换成从 `React` 全局变量中加载。

你可以在[这里](#)了解更多如何配置`webpack`。

## 整合在一起

执行：

```
webpack
```

在浏览器里打开 `index.html`，工程应该已经可以用了！你可以看到页面上显示着“Hello from TypeScript and React!”

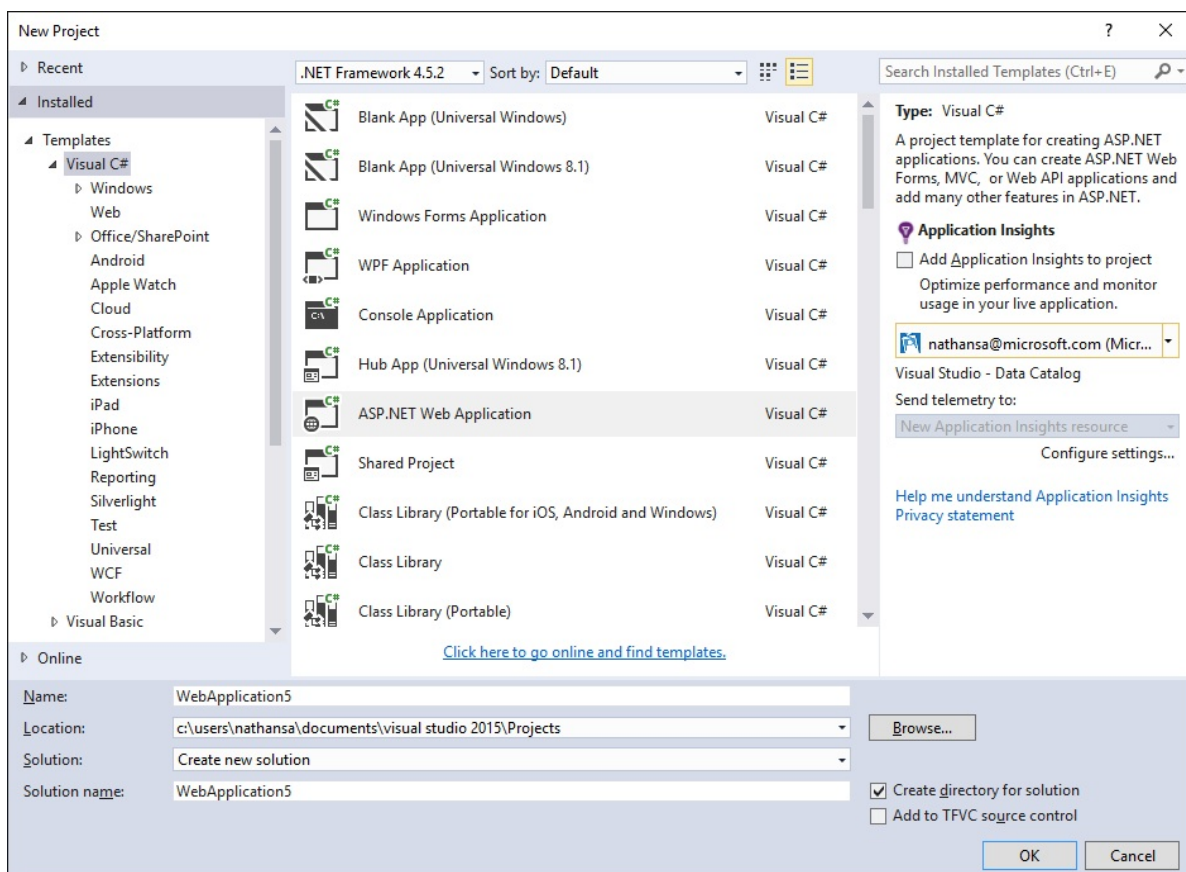
# ASP.NET 4

## 安装 TypeScript

如果你使用的 Visual Studio 版本还不支持 TypeScript，你可以安装 [Visual Studio 2015](#) 或者 [Visual Studio 2013](#)。这个快速上手指南使用的是 Visual Studio 2015。

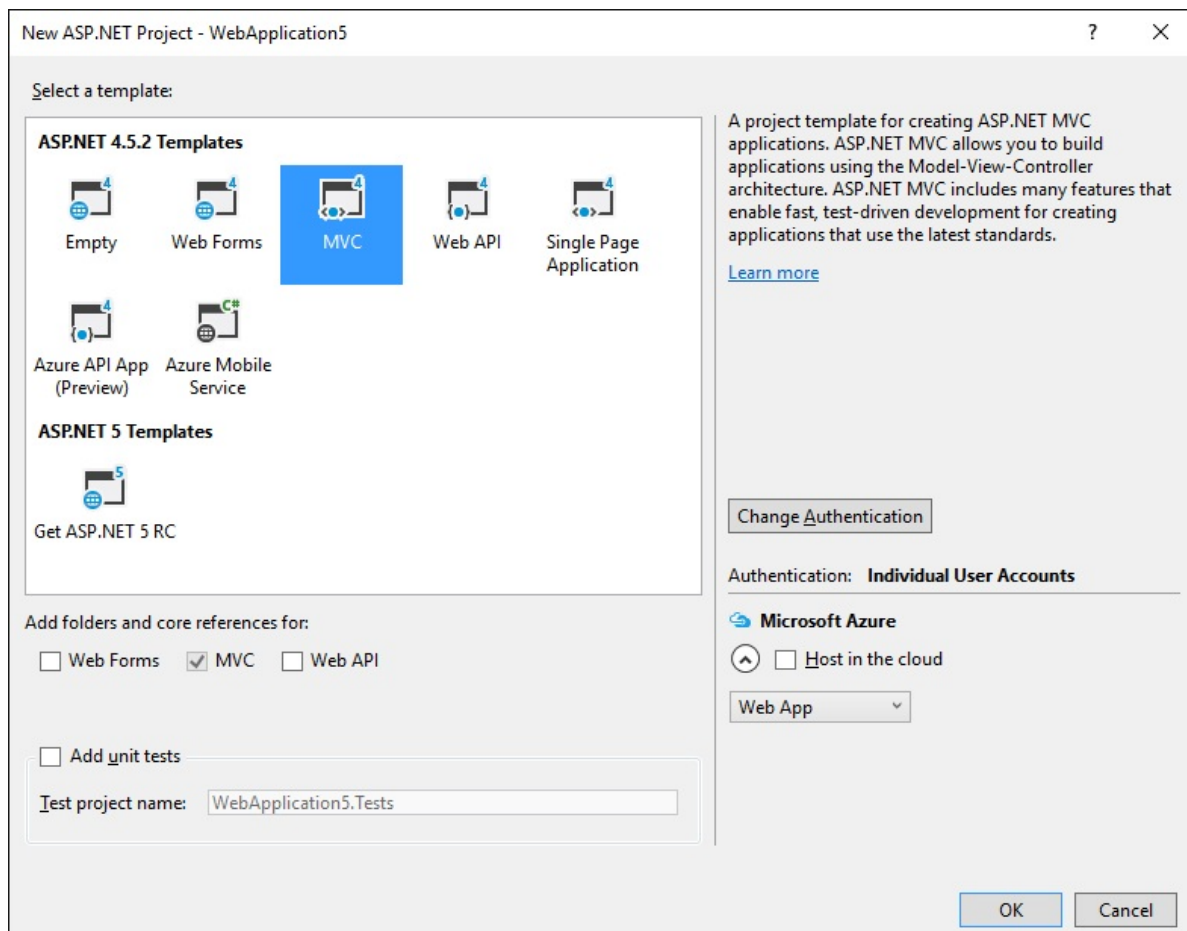
## 新建项目

1. 选择 **File**
2. 选择 **New Project**
3. 选择 **Visual C#**
4. 选择 **ASP.NET Web Application**



5. 选择 **MVC**

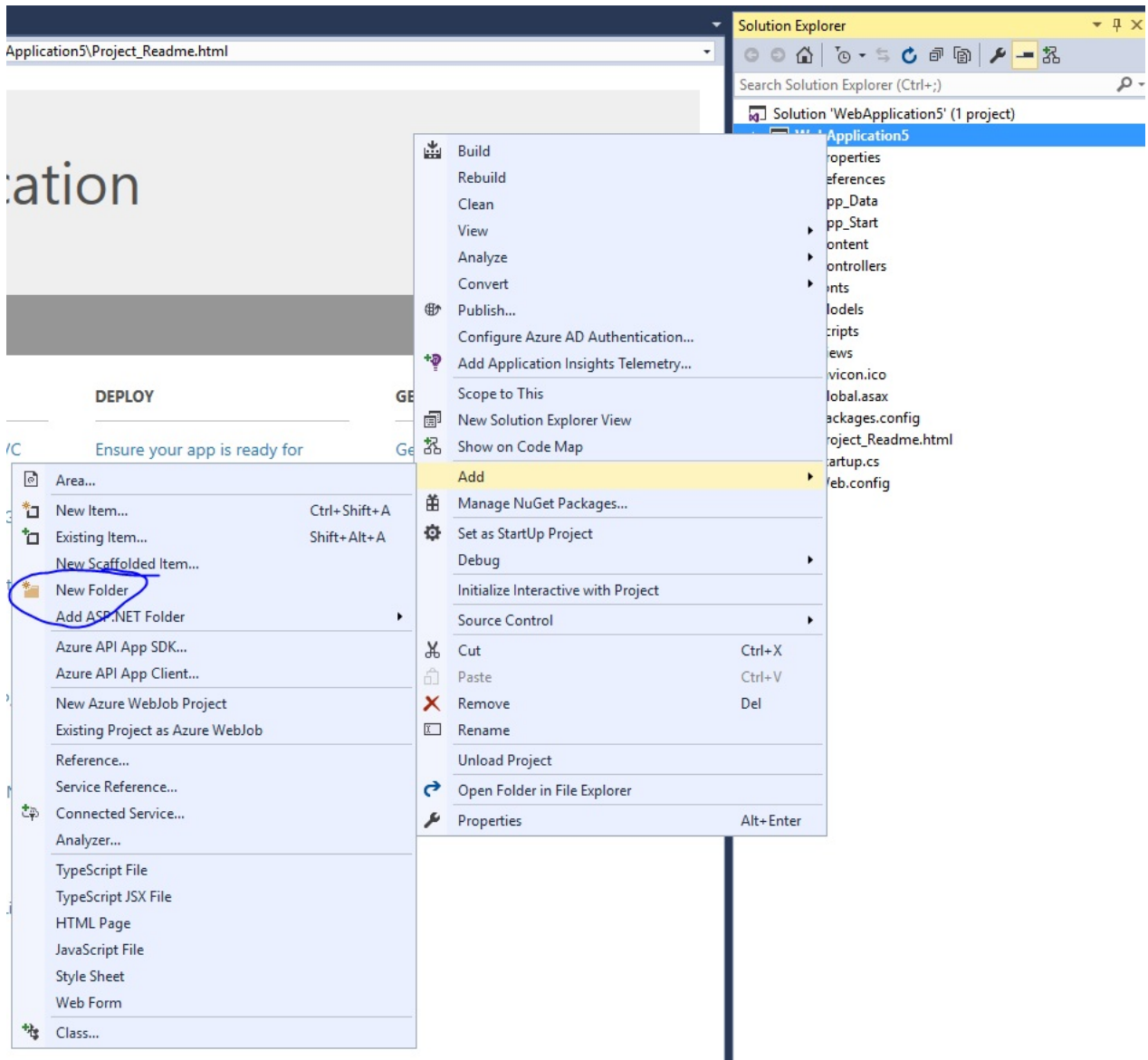
取消复选 "Host in the cloud" 本指南将使用一个本地示例。



运行此应用以确保它能正常工作。

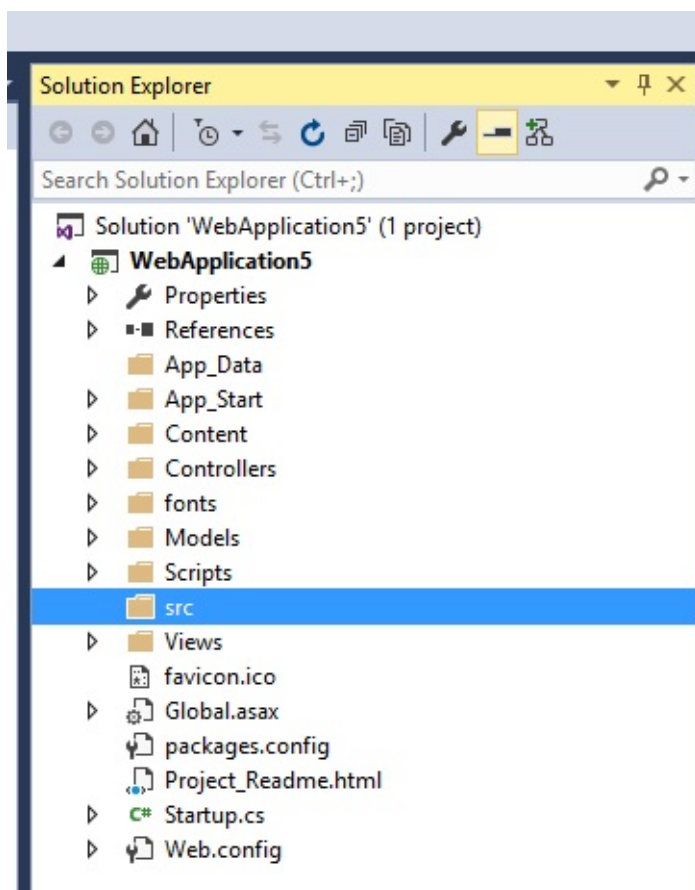
## 添加 TypeScript

下一步我们为 TypeScript 添加一个文件夹。



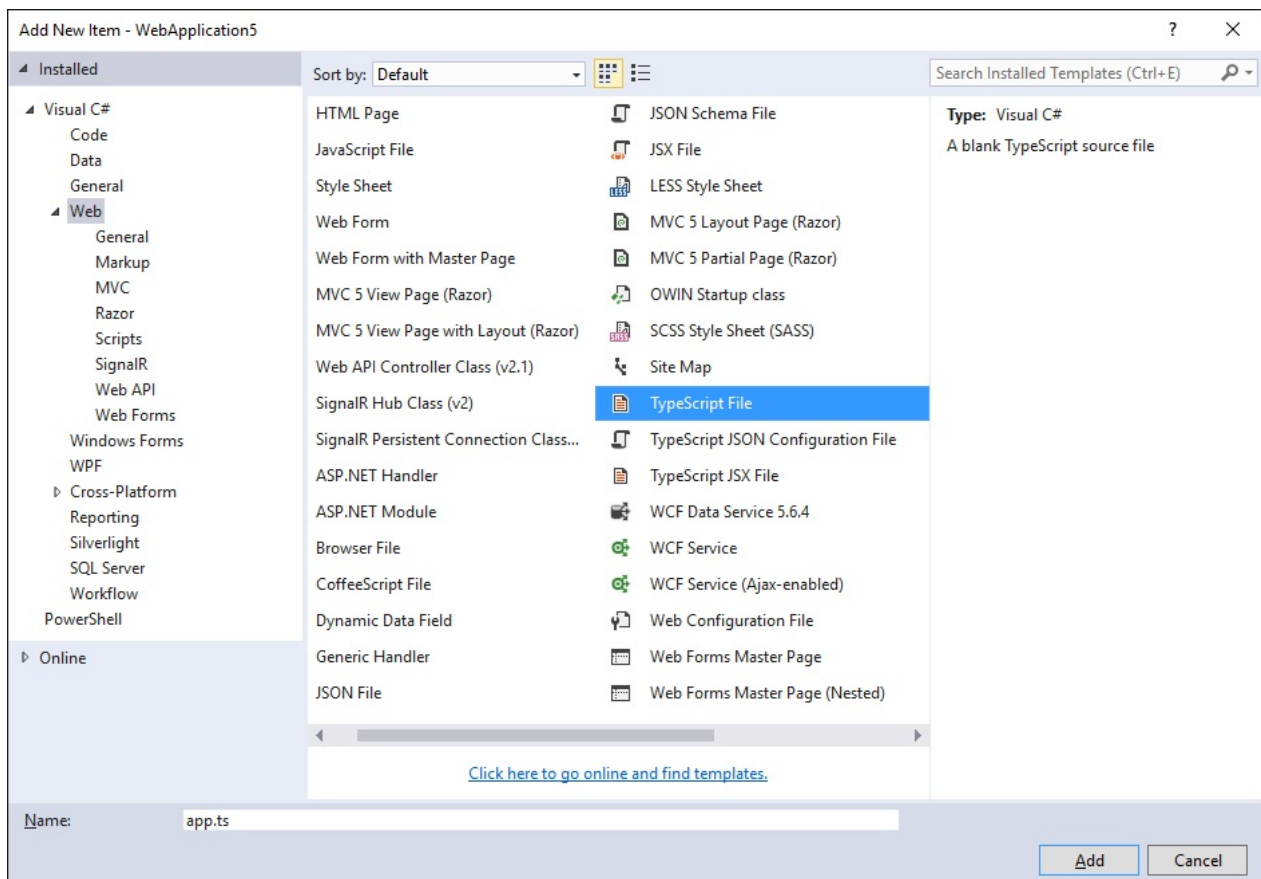
将文件夹命名为 `src`。





## 添加 TypeScript 代码

在 `src` 上右击并选择 **New Item**。接着选择 **TypeScript File** 并将此文件命名为 `app.ts`。



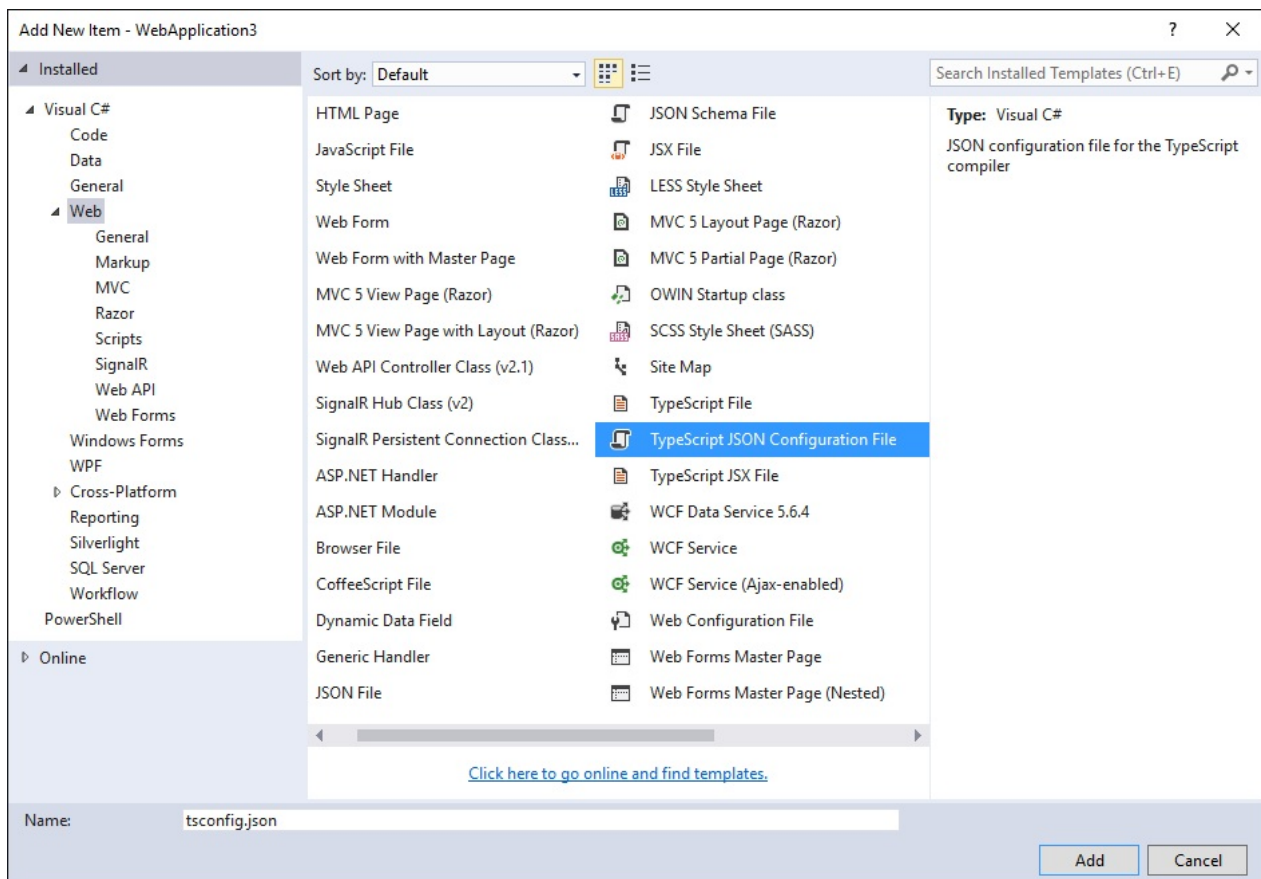
## 添加示例代码

将以下代码写入 `app.ts` 文件。

```
function sayHello() {  
    const compiler = (document.getElementById("compiler") as HTMLInputElement).value;  
    const framework = (document.getElementById("framework") as HTMLInputElement).value;  
    return `Hello from ${compiler} and ${framework}!`;  
}
```

## 构建设置

右击项目并选择 **New Item**。接着选择 **TypeScript Configuration File** 保持文件的默认名字为 `tsconfig.json`。



将默认的 `tsconfig.json` 内容改为如下所示：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5",
    "outDir": "./Scripts/App"
  },
  "files": [
    "./src/app.ts",
  ],
  "compileOnSave": true
}
```

看起来和默认的设置差不多，但注意以下不同之处：

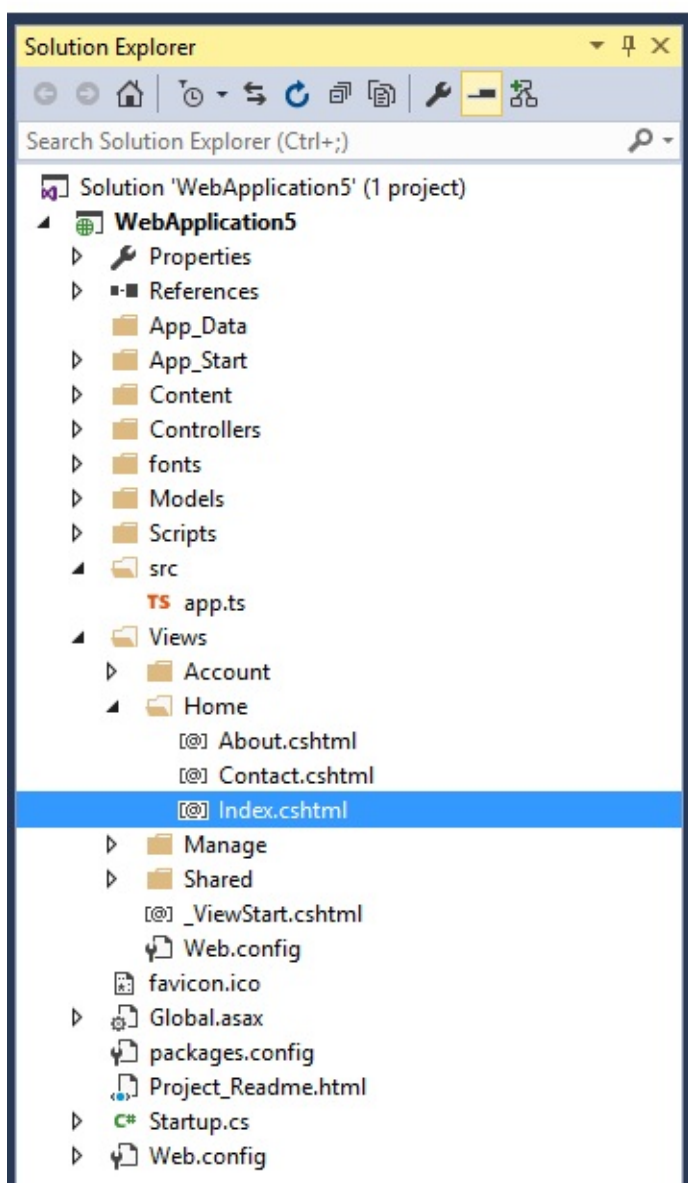
1. 设置 `"noImplicitAny": true`。
2. 特别是这里 `"outDir": "./Scripts/App"`。
3. 显式列出了 `"files"` 而不是依据 `"excludes"` 选项。

#### 4. 设置 `"compileOnSave": true`。

当你写新代码时，设置 `"noImplicitAny"` 选项是个好主意 — 这可以确保你不会错写任何新的类型。设置 `"compileOnSave"` 选项可以确保你在运行web程序前自动编译保存变更后的代码。更多信息请参见 [the tsconfig.json documentation](#)。

## 在视图中调用脚本

#### 1. 在 **Solution Explorer** 中, 打开 **Views | Home |** `Index.cshtml`。

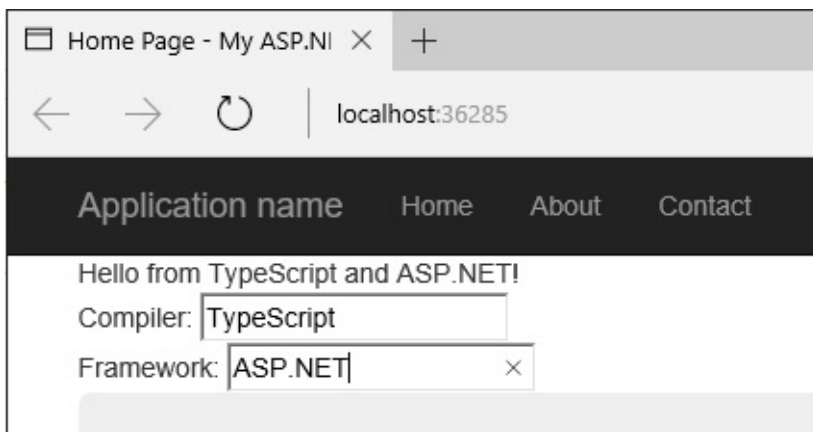


#### 2. 修改代码如下：

```
@{
    ViewBag.Title = "Home Page";
}
<script src="~/Scripts/App/app.js"></script>
<div id="message"></div>
<div>
    Compiler: <input id="compiler" value="TypeScript" onkeyup="
    Framework: <input id="framework" value="ASP.NET" onkeyup="c
</div>
```

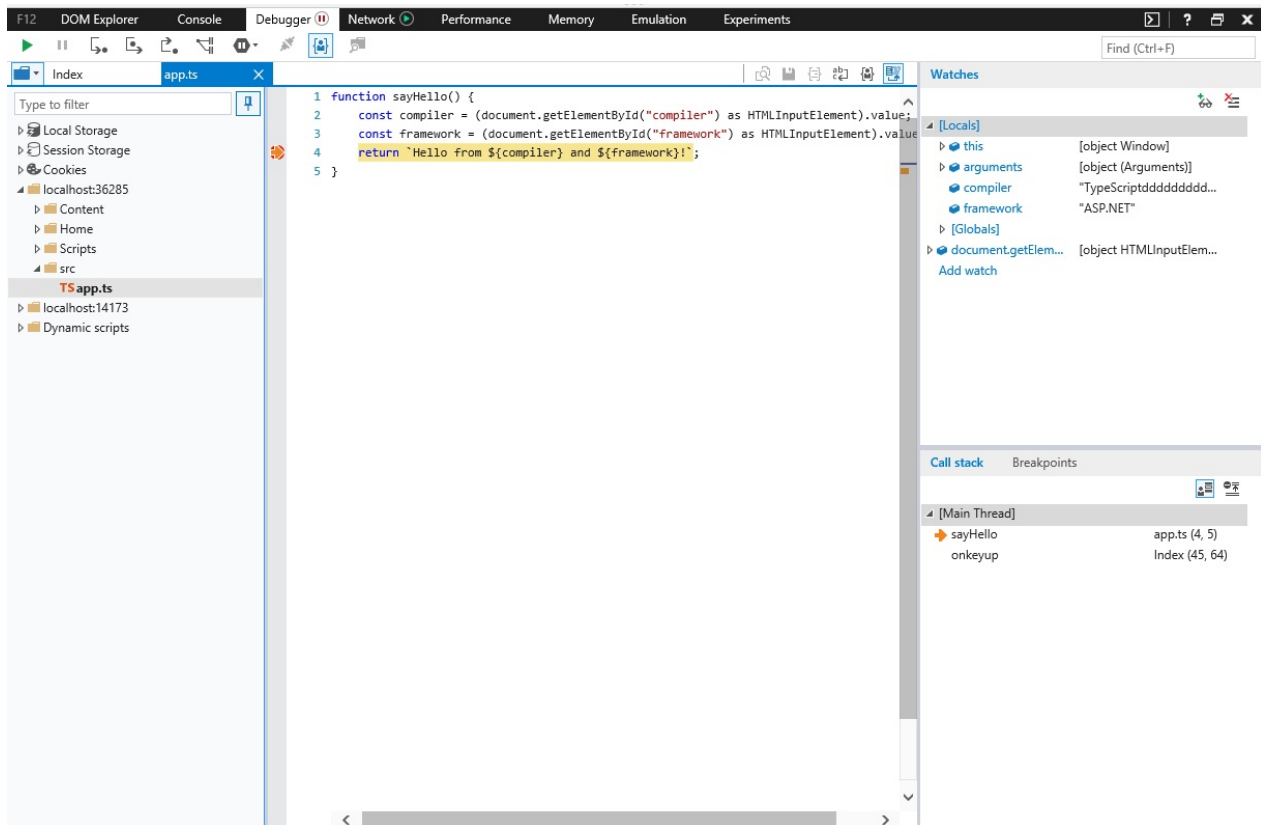
## 测试

1. 运行项目。
2. 在输入框中键入时，您应该看到一个消息：



## 调试

1. 在 Edge 浏览器中，按 F12 键并选择 **Debugger** 标签页。
2. 展开 localhost 列表，选择 src/app.ts
3. 在 `return` 那一行上打一个断点。
4. 在输入框中键入一些内容，确认 TypeScript 代码命中断点，观察它是否能正确地工作。



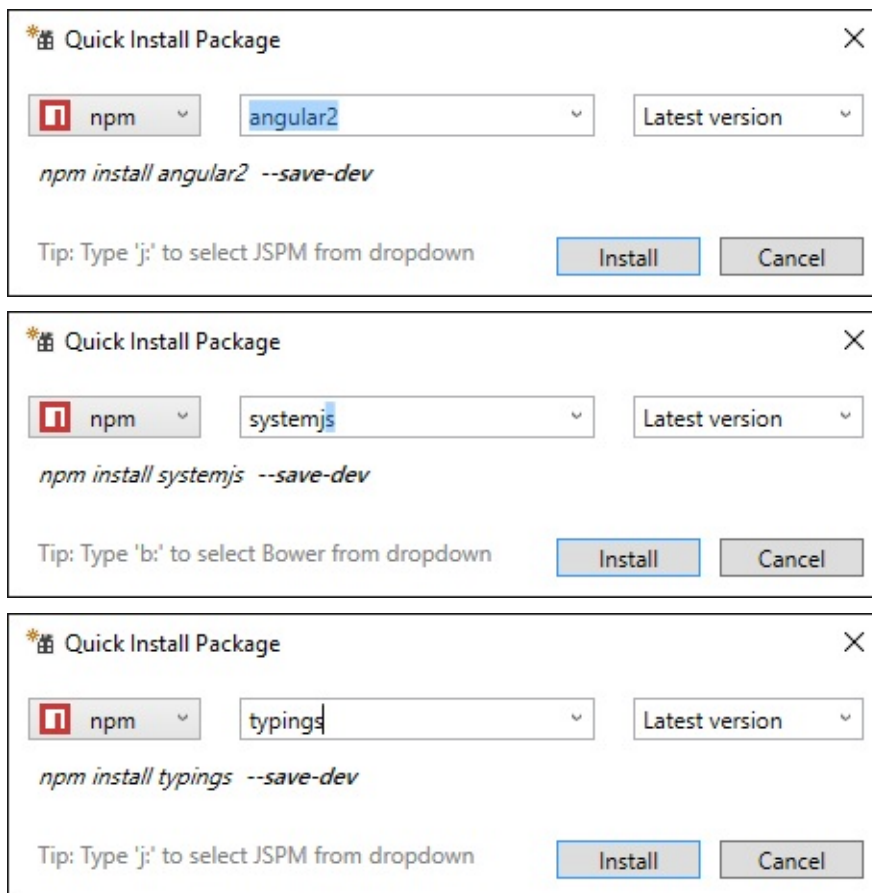
这就是你需要知道的在ASP.NET中使用TypeScript的基本知识了。接下来，我们引入Angular，写一个简单的Angular程序示例。

## 添加 Angular 2

### 使用 NPM 下载所需的包

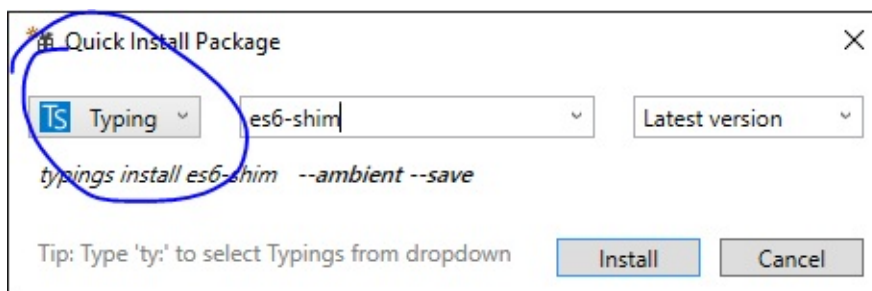
1. 安装 [PackageInstaller](#)。
2. 用 PackageInstaller 来安装 Angular 2，systemjs 和 Typings。

在project上右击, 选择 **Quick Install Package**。



3. 用 PackageInstaller 安装 es6-shim 的类型文件。

Angular 2 包含 es6-shim 以提供 Promise 支持, 但 TypeScript 还需要它的类型文件。在 PackageInstaller 中, 选择 Typing 替换 npm 选项。接着键入 "es6-shim" :



## 更新 tsconfig.json

现在安装好了 Angular 2 及其依赖项, 我们还需要启用 TypeScript 中实验性的装饰器支持并且引入 es6-shim 的类型文件。将来的版本中, 装饰器和 ES6 选项将成为默认选项, 我们就可以不做此设置了。添加 `"experimentalDecorators": true,` `"emitDecoratorMetadata": true` 选项到 `"compilerOptions"` 选项段, 添加

`"./typings/main.d.ts"` 到 `"files"` 选项段。最后，我们还将要创建新的代码文件 `"./src/model.ts"` 、 `"./src/main.ts"` ，也将它们添加到 `"files"` 中，现在 `tsconfig` 看起来像这样：

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "outDir": "./Scripts/App"
  },
  "files": [
    "./src/app.ts",
    "./src/model.ts",
    "./src/main.ts",
    "./typings/main.d.ts"
  ]
}
```

## 添加 **CopyFiles** 到 **build** 中

最后，我们需要确保 Angular 文件作为 `build` 的一部分复制进来。这样操作，右击项目选择 'Unload' ，再次右击项目选择 'Edit csproj' 。在 TypeScript 配置项 `PropertyGroup` 之后，添加一个 `ItemGroup` 和 `Target` 配置项来复制 Angular 文件。



```

<ItemGroup>
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\angular"
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\angular"
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\systemjs"
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\rxjs\b
</ItemGroup>
<Target Name="CopyFiles" BeforeTargets="Build">
  <Copy SourceFiles="@{(NodeLib)" DestinationFolder="$(MSBuildProject
</Target>

```

现在，右击 `project` 选择重新加载项目。此时应当能在 `Solution Explorer` 中看到 `node_modules`。

## 用 TypeScript 写一个简单的 Angular 应用

首先，将 `app.ts` 改成：

```

import {Component} from "angular2/core"
import {MyModel} from "../model"

@Component({
  selector: `my-app`,
  template: `<div>Hello from {{getCompiler()}}</div>`
})
class MyApp {
  model = new MyModel();
  getCompiler() {
    return this.model.compiler;
  }
}

```

接着在 `src` 中添加 TypeScript 文件 `model.ts`：

```

export class MyModel {
  compiler = "TypeScript";
}

```

再在 `src` 中添加 `main.ts` :

```
import {bootstrap} from "angular2/platform/browser";
import {MyApp} from "../app";
bootstrap(MyApp);
```

最后，将 `Views/Home/Index.cshtml` 改成：

```
@{
    ViewBag.Title = "Home Page";
}
<script src="~/Scripts/angular2-polyfills.js"></script>
<script src="~/Scripts/system.src.js"></script>
<script src="~/Scripts/rx.js"></script>
<script src="~/Scripts/angular2.js"></script>
<script>
    System.config({
        packages: {
            '/Scripts/App': {
                format: 'cjs',
                defaultExtension: 'js'
            }
        }
    });
    System.import('/Scripts/App/main').then(null, console.error.bind(console));
</script>
<my-app>Loading...</my-app>
```

这里加载了此应用。运行 ASP.NET 应用，你应该能看到一个 div 显示 "Loading..." 紧接着更新成显示 "Hello from TypeScript"。

# ASP.NET Core

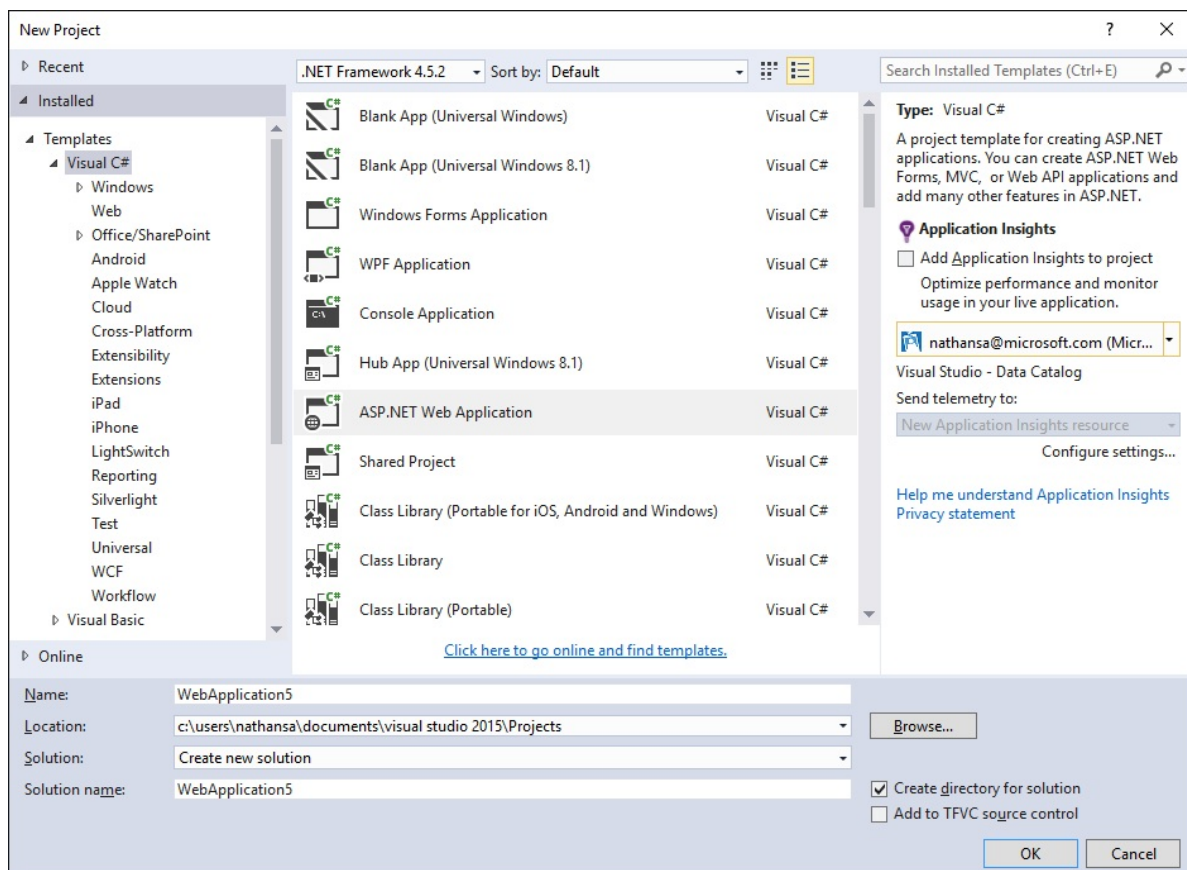
## 安装 ASP.NET Core 和 TypeScript

首先，若有必要请安装 [ASP.NET Core](#)。这个快速上手指南使用的是 Visual Studio，若要使用 ASP.NET Core 你需要有 Visual Studio 2015。

其次，如果你的 Visual Studio 中没有包含 TypeScript，你可以从这里安装 [TypeScript for Visual Studio 2015](#)。

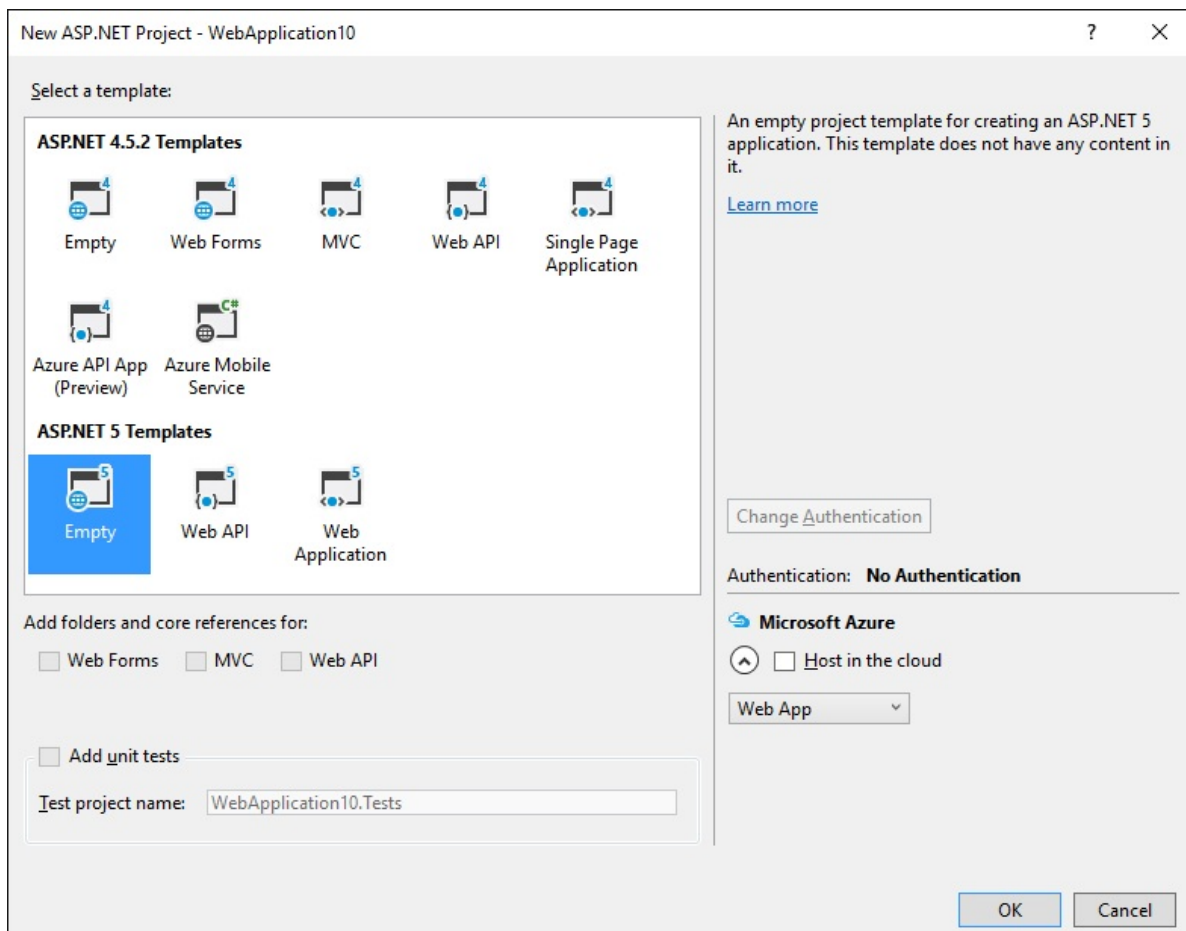
## 新建工程

1. 选择 **File**
2. 选择 **New Project** (Ctrl + Shift + N)
3. 选择 **Visual C#**
4. 选择 **ASP.NET Web Application**



5. 选择 **ASP.NET 5 Empty** 工程模板

取消复选 "Host in the cloud" 本指南将使用一个本地示例。



运行此应用以确保它能正常工作。

## 设置服务项

在 `project.json` 文件的 `"dependencies"` 字段里添加:

```
"Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
```

最终的 `dependencies` 部分应该类似于下面这样：

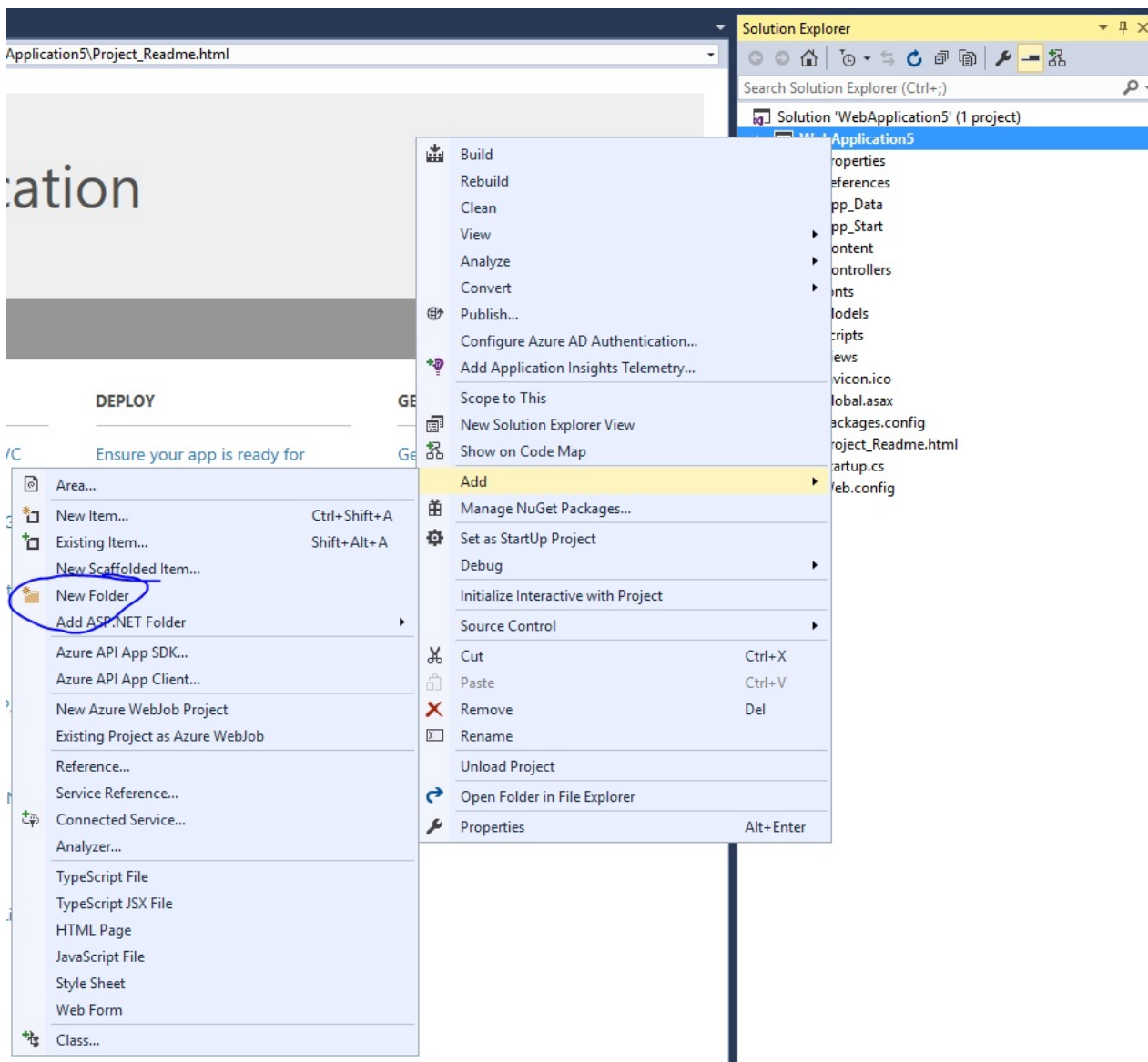
```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
  "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
},
```

用以下内容替换 `Startup.cs` 文件里的 `Configure` 函数：

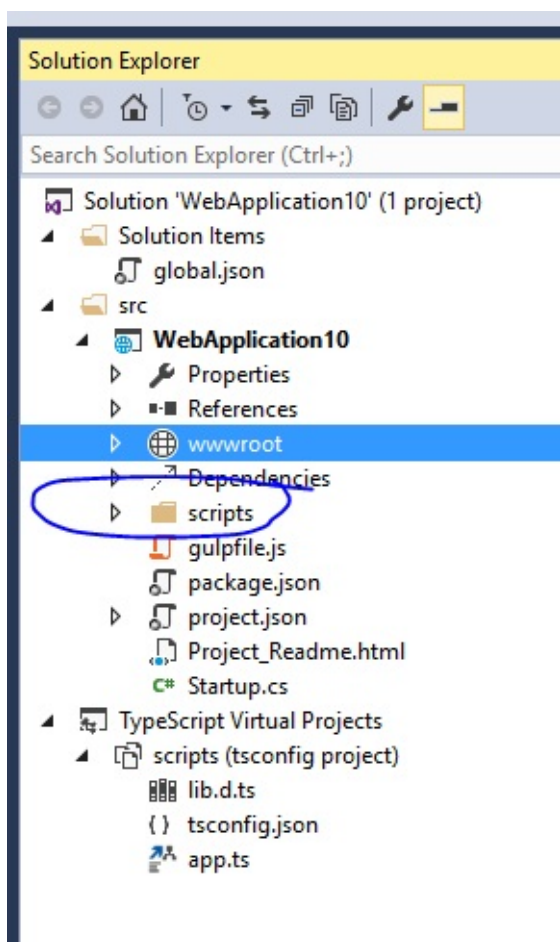
```
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

## 添加 TypeScript

下一步我们为 TypeScript 添加一个文件夹。

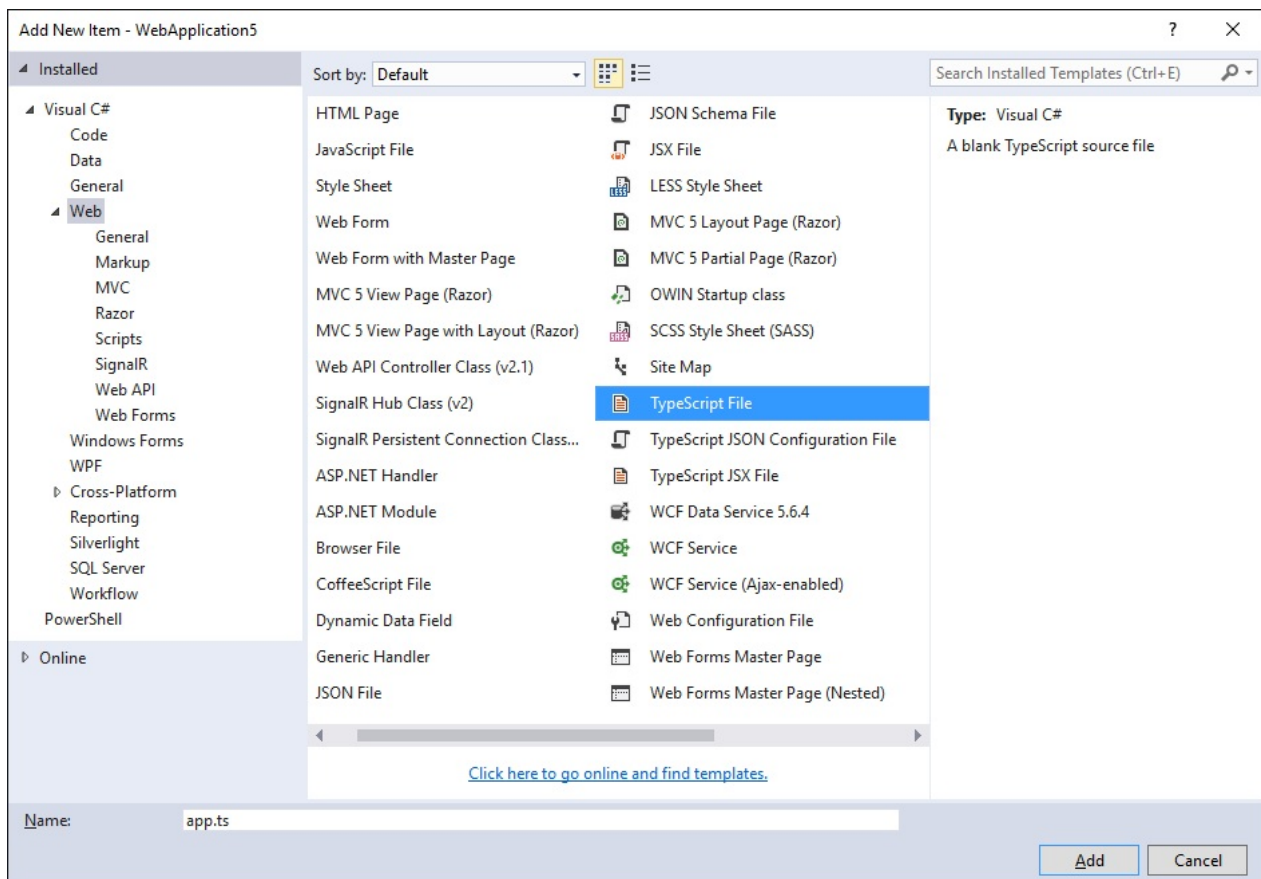


将文件夹命名为 `scripts`。



## 添加 TypeScript 代码

在 `scripts` 上右击并选择 **New Item**。接着选择 **TypeScript File**（也可能 .NET Core 部分），并将此文件命名为 `app.ts`。



## 添加示例代码

将以下代码写入app.ts文件。

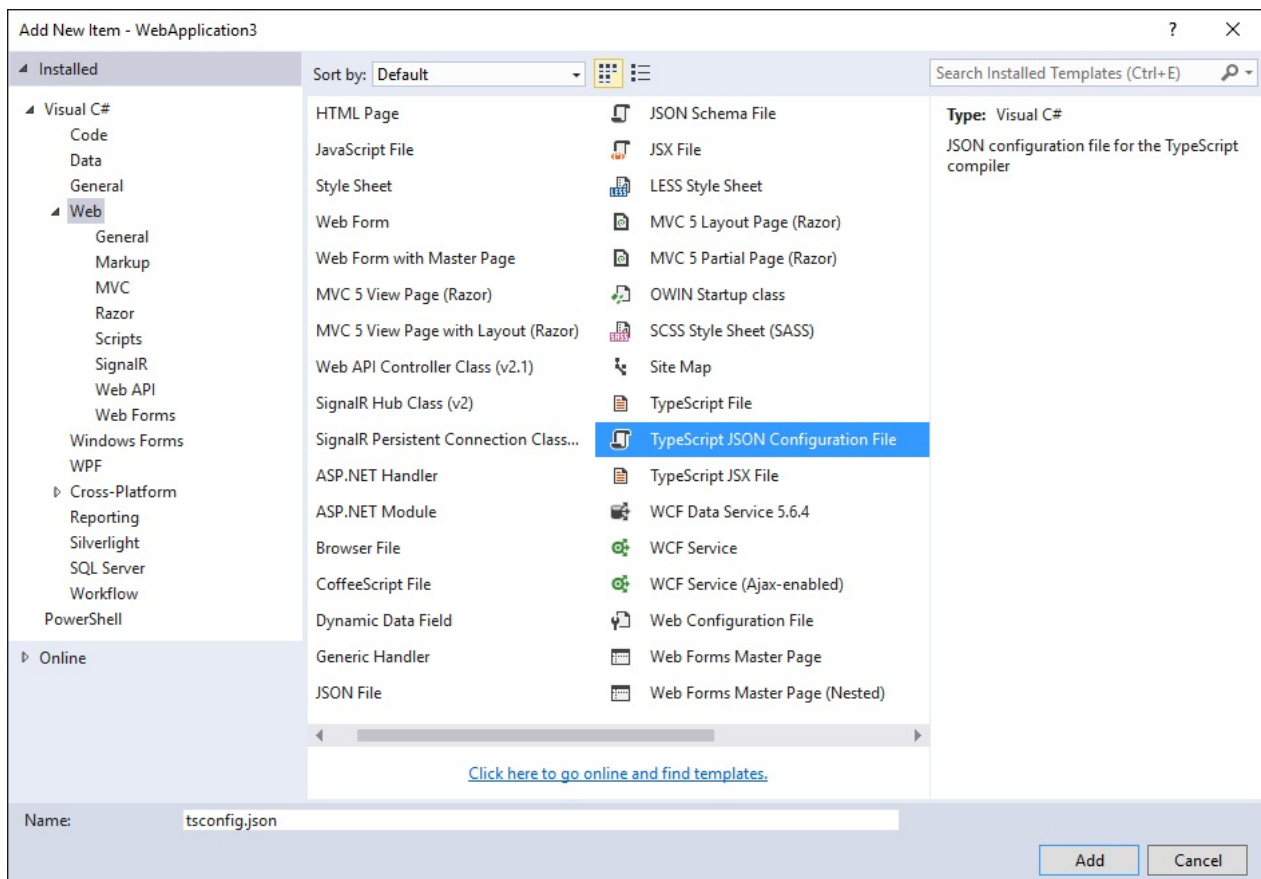
```
function sayHello() {  
    const compiler = (document.getElementById("compiler") as HTMLInput)  
    const framework = (document.getElementById("framework") as HTMLInput)  
    return `Hello from ${compiler} and ${framework}!`;  
}
```

## 构建设置

### 配置 TypeScript 编译器

我们先来告诉TypeScript怎样构建。右击scripts文件夹并选择**New Item**。接着选择**TypeScript Configuration File**，保持文件的默认名字为 `tsconfig.json`。





将默认的 `tsconfig.json` 内容改为如下所示：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5",
  },
  "files": [
    "./app.ts"
  ],
  "compileOnSave": true
}
```

看起来和默认的设置差不多，但注意以下不同之处：

1. 设置 `"noImplicitAny": true` 。
2. 显式列出了 `"files"` 而不是依据 `"excludes"` 。
3. 设置 `"compileOnSave": true` 。



当你写新代码时，设置 `"noImplicitAny"` 选项是个不错的选择 — 这可以确保你不会错写任何新的类型。设置 `"compileOnSave"` 选项可以确保你在运行web程序前自动编译保存变更后的代码。

## 配置 NPM

现在，我们来配置NPM以使用我们能够下载JavaScript包。在工程上右击并选择 **New Item**。接着选择 **NPM Configuration File**，保持文件的默认名字为 `package.json`。在 `"devDependencies"` 部分添加 `"gulp"` 和 `"del"`：

```
"devDependencies": {  
  "gulp": "3.9.0",  
  "del": "2.2.0"  
}
```

保存这个文件后，Visual Studio将开始安装gulp和del。若没有自动开始，请右击 `package.json` 文件选择 **Restore Packages**。

## 设置 gulp

最后，添加一个新JavaScript文件 `gulpfile.js`。键入以下内容：

```

/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and using
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=501756
*/

var gulp = require('gulp');
var del = require('del');

var paths = {
  scripts: ['scripts/**/*.js', 'scripts/**/*.ts', 'scripts/**/*.map'];
};

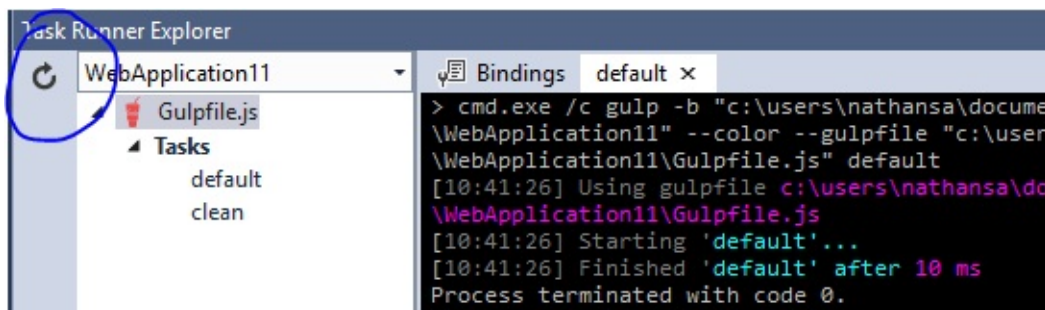
gulp.task('clean', function () {
  return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', function () {
  gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'));
});

```

第一行是告诉Visual Studio构建完成后，立即运行'default'任务。当你应答 Visual Studio 清除构建内容后，它也将运行'clean'任务。

现在，右击 `gulpfile.js` 并选择**Task Runner Explorer**。若'default'和'clean'任务没有显示输出内容的话，请刷新explorer：



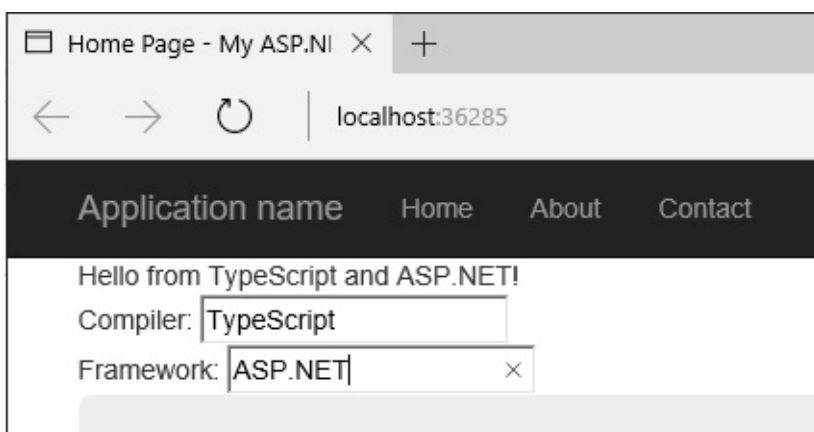
## 编写HTML页

在 `wwwroot` 中添加一个新建项 `index.html`。在 `index.html` 中写入以下代码：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <script src="scripts/app.js"></script>
  <title></title>
</head>
<body>
  <div id="message"></div>
  <div>
    Compiler: <input id="compiler" value="TypeScript" onkeyup="
    Framework: <input id="framework" value="ASP.NET" onkeyup="
  </div>
</body>
</html>
```

## 测试

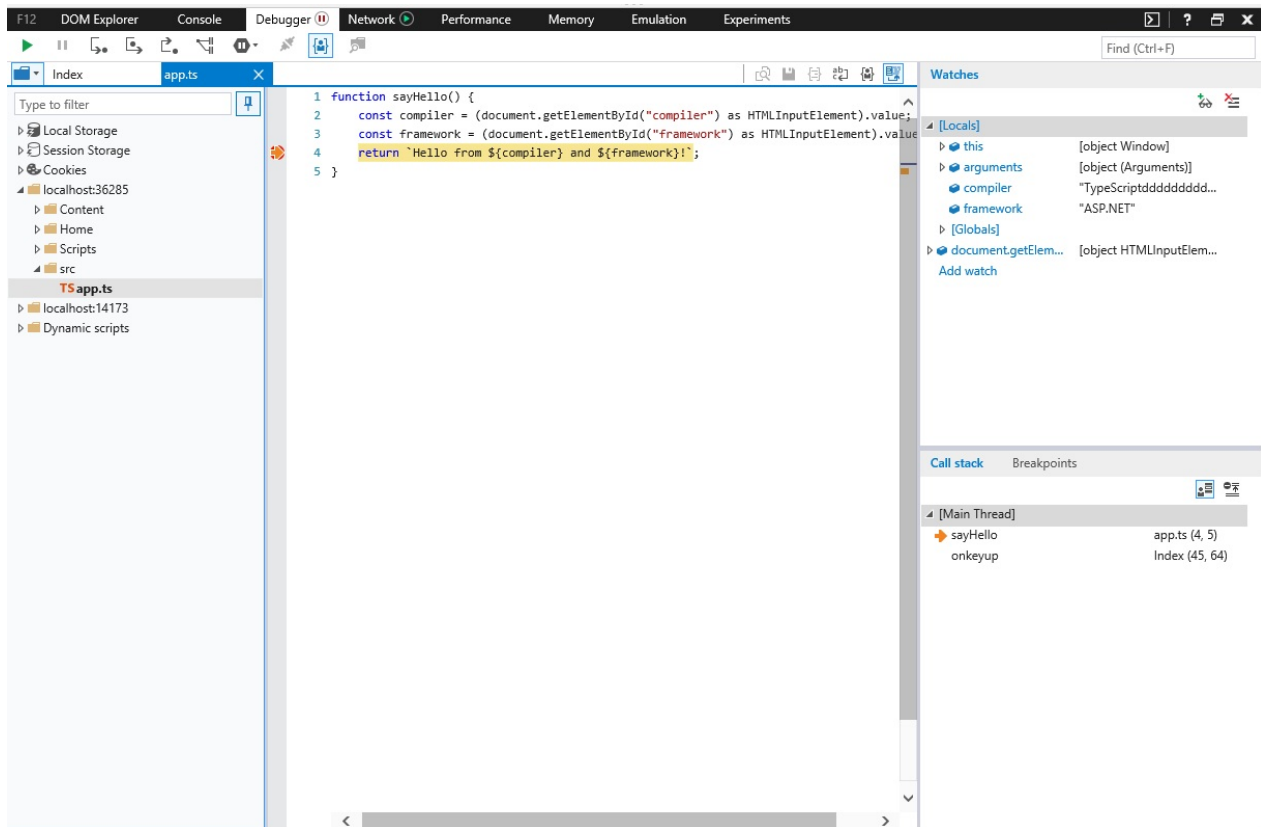
1. 运行项目。
2. 在输入框中键入时，您应该看到一个消息：



## 调试

1. 在 Edge 浏览器中，按 F12 键并选择 **Debugger** 标签页。

2. 展开 localhost 列表，选择 scripts/app.ts
3. 在 `return` 那一行上打一个断点。
4. 在输入框中键入一些内容，确认TypeScript代码命中断点，观察它是否能正确地工作。



这就是你需要知道的在ASP.NET中使用TypeScript的基本知识了。接下来，我们引入Angular，写一个简单的Angular程序示例。

## 添加 Angular 2

### 使用 NPM 下载所需的包

在 `package.json` 文件的 `"dependencies"` 添加 Angular 2 和 SystemJS：

```
"dependencies": {
  "angular2": "2.0.0-beta.11",
  "systemjs": "0.19.24",
},
```

## 安装 typings

Angular 2 包含 es6-shim 以提供 Promise 支持，但 TypeScript 还需要它的类型文件。打开一个 command prompt，切换到应用程序源文件目录中：

```
cd C:\Users\...\Documents\Visual Studio 2015\Projects\...\src\  
npm install -g typings  
typings install es6-shim --ambient
```

## 更新 tsconfig.json

现在安装好了 Angular 2 及其依赖项，我们还需要启用 TypeScript 中实验性的装饰器支持并且引入 es6-shim 的类型文件。将来的版本中，装饰器和 ES6 选项将成为默认选项，我们就可以不做此设置了。添加 "experimentalDecorators": true, "emitDecoratorMetadata": true 选项到 "compilerOptions" 选项段，添加 "./typings/main.d.ts" 到 "files" 选项段。最后，我们还将要创建新的代码文件 "./src/model.ts" 、 "./src/main.ts" ，也将它们添加到 "files" 中，现在 tsconfig 看起来像这样：

```
{  
  "compilerOptions": {  
    "noImplicitAny": true,  
    "noEmitOnError": true,  
    "sourceMap": true,  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "target": "es5"  
  },  
  "files": [  
    "./app.ts",  
    "./model.ts",  
    "./main.ts",  
    "../typings/main.d.ts"  
  ],  
  "compileOnSave": true  
}
```

## 将 Angular 添加到 gulp 构建中

最后，我们需要确保 Angular 文件作为 build 的一部分复制进来。我们需要添加：

1. 库文件目录。
2. 添加一个 `lib` 任务来输送文件到 `wwwroot`。
3. 在 `default` 任务上添加 `lib` 任务依赖。

更新后的 `gulpfile.js` 像如下所示：

```
/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and using
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=51
*/

var gulp = require('gulp');
var del = require('del');

var paths = {
  scripts: ['scripts/**/*.js', 'scripts/**/*.ts', 'scripts/**/*.r
  libs: ['node_modules/angular2/bundles/angular2.js',
        'node_modules/angular2/bundles/angular2-polyfills.js',
        'node_modules/systemjs/dist/system.src.js',
        'node_modules/rxjs/bundles/Rx.js']
};

gulp.task('lib', function () {
  gulp.src(paths.libs).pipe(gulp.dest('wwwroot/scripts/lib'))
});

gulp.task('clean', function () {
  return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', ['lib'], function () {
  gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'))
});
```

此外，保存了此gulpfile后，要确保 Task Runner Explorer 能看到 `lib` 任务。

## 用 TypeScript 写一个简单的 Angular 应用

首先，将 `app.ts` 改成：

```
import {Component} from "angular2/core"
import {MyModel} from "../model"

@Component({
  selector: `my-app`,
  template: `<div>Hello from {{getCompiler()}}</div>`
})
class MyApp {
  model = new MyModel();
  getCompiler() {
    return this.model.compiler;
  }
}
```

接着在 `scripts` 中添加 TypeScript 文件 `model.ts`：

```
export class MyModel {
  compiler = "TypeScript";
}
```

再在 `scripts` 中添加 `main.ts`：

```
import {bootstrap} from "angular2/platform/browser";
import {MyApp} from "../app";
bootstrap(MyApp);
```

最后，将 `index.html` 改成：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <script src="scripts/lib/angular2-polyfills.js"></script>
  <script src="scripts/lib/system.src.js"></script>
  <script src="scripts/lib/rx.js"></script>
  <script src="scripts/lib/angular2.js"></script>
  <script>
    System.config({
      packages: {
        'scripts': {
          format: 'cjs',
          defaultExtension: 'js'
        }
      }
    });
    System.import('scripts/main').then(null, console.error.bind(console));
  </script>
  <title></title>
</head>
<body>
  <my-app>Loading...</my-app>
</body>
</html>
```

这里加载了此应用。运行 ASP.NET 应用，你应该能看到一个 div 显示 "Loading..." 紧接着更新成显示 "Hello from TypeScript"。



这个快速上手指南会告诉你如何结合使用TypeScript和[Knockout.js](#)。

这里我们假设你已经会使用[Node.js](#)和[npm](#)

## 新建工程

首先，我们新建一个目录。暂时命名为 `proj`，当然了你可以使用任何喜欢的名字。

```
mkdir proj
cd proj
```

接下来，我们按如下方式来组织这个工程：

```
proj/
+- src/
+- built/
```

TypeScript源码放在 `src` 目录下，经过TypeScript编译器编译后，生成的文件放在 `built` 目录里。

下面创建目录：

```
mkdir src
mkdir built
```

## 安装构建依赖

首先确保TypeScript和Typings已经全局安装。

```
npm install -g typescript typings
```

你肯定已经很了解TypeScript了，但你有可能还不太了解Typings. [Typings](#)是一个包管理器用来获取声明文件。我们将会使用它来获取Knockout的声明文件。

```
typings install --ambient --save knockout
```

`--ambient` 标记会告诉Typings从[DefinitelyTyped](#)去获取声明文件，这是由社区维护的 `.d.ts` 文件仓库。这个命令会在当前目录下创建一个 `typings.json` 文件和一个 `typings` 文件夹。

## 获取运行时依赖

我们需要Knockout和RequireJS。[RequireJS](#)是一个库，它可以让我们在运行时异步地加载模块。

有以下几种获取方式：

1. 手动下载文件并维护它们。
2. 通过像[Bower](#)这样的包管理下载并维护它们。
3. 使用内容分发网络（CDN）来维护这两个文件。

我们使用第一种方法，它会简单一些，但是Knockout的官方文档上有讲解[如何使用CDN](#)，更多像RequireJS一样的代码库可以在[cdnjs](#)上查找。

下面让我们在工程根目录下创建 `externals` 目录。

```
mkdir externals
```

然后[下载Knockout](#)和[下载RequireJS](#)到这个目录里。最新的压缩后版本就可以。

## 添加TypeScript配置文件

下面我们想把所有的TypeScript文件整合到一起 - 包括自己写的和必须的声明文件。

我们需要创建一个 `tsconfig.json` 文件，包含了输入文件列表和编译选项。在工程根目录下创建一个新文件 `tsconfig.json`，内容如下：

```
{
  "compilerOptions": {
    "outDir": "./built/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "amd",
    "target": "es5"
  },
  "files": [
    "./typings/main.d.ts",
    "./src/require-config.ts",
    "./src/hello.ts"
  ]
}
```

这里引用了 `typings/main.d.ts`，它是Typings帮我们创建的。这个文件会自动地包含所有安装的依赖。

你可能会对 `typings` 目录下的 `browser.d.ts` 文件感到好奇，尤其因为我们将在浏览器里运行代码。其实原因是这样的，当目标为浏览器的时候，一些包会生成不同的版本。通常来讲，这些情况很少发生并且在这里我们不会遇到这种情况，所以我们可以忽略 `browser.d.ts`。

你可以在[这里](#)查看更多关于 `tsconfig.json` 文件的信息

## 写些代码

下面我们使用Knockout写一段TypeScript代码。首先，在 `src` 目录里新建一个 `hello.ts` 文件。

```
import * as ko from "knockout";

class HelloViewModel {
  language: KnockoutObservable<string>
  framework: KnockoutObservable<string>

  constructor(language: string, framework: string) {
    this.language = ko.observable(language);
    this.framework = ko.observable(framework);
  }
}

ko.applyBindings(new HelloViewModel("TypeScript", "Knockout"));
```

接下来，在 `src` 目录下再新建一个 `require-config.ts` 文件。

```
declare var require: any;
require.config({
  paths: {
    "knockout": "externals/knockout-3.4.0",
  }
});
```

这个文件会告诉RequireJS从哪里导入Knockout，好比我们在 `hello.ts` 里做的一样。你创建的所有页面都应该在RequireJS之后和导入任何东西之前引入它。为了更好地理解这个文件和如何配置RequireJS，可以查看[文档](#)。

我们还需要一个视图来显示 `HelloViewModel`。在 `proj` 目录的根上创建一个文件 `index.html`，内容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello Knockout!</title>
  </head>
  <body>
    <p>
      Hello from
      <strong data-bind="text: language">todo</strong>
      and
      <strong data-bind="text: framework">todo</strong>!
    </p>

    <p>Language: <input data-bind="value: language" /></p>
    <p>Framework: <input data-bind="value: framework" /></p>

    <script src="./externals/require.js"></script>
    <script src="./built/require-config.js"></script>
    <script>
      require(["built/hello"]);
    </script>
  </body>
</html>
```

注意，有两个`script`标签。首先，我们引入RequireJS。然后我们再在 `require-config.js` 里映射外部依赖，这样RequireJS就能知道到哪里去查找它们。最后，使用我们要去加载的模块去调用 `require`。

## 将所有部分整合在一起

运行

```
tsc
```

现在，在你喜欢的浏览器打开 `index.html`，所有都应该好用了。你应该可以看到页面上显示“Hello from TypeScript and Knockout!”。在它下面，你还会看到两个输入框。当你改变输入和切换焦点时，就会看到原先显示的信息改变了。

# TypeScript 新增内容

章节分离自 [TypeScript 新增特性一览](#)。

- [TypeScript 1.8](#)
- [TypeScript 1.7](#)
- [TypeScript 1.6](#)
- [TypeScript 1.5](#)
- [TypeScript 1.4](#)
- [TypeScript 1.3](#)
- [TypeScript 1.1](#)

# TypeScript 1.8

## 类型参数约束

在 TypeScript 1.8 中, 类型参数的限制可以引用自同一个类型参数列表中的类型参数. 在此之前这种做法会报错. 这种特性通常被叫做 [F-Bounded Polymorphism](#).

## 例子

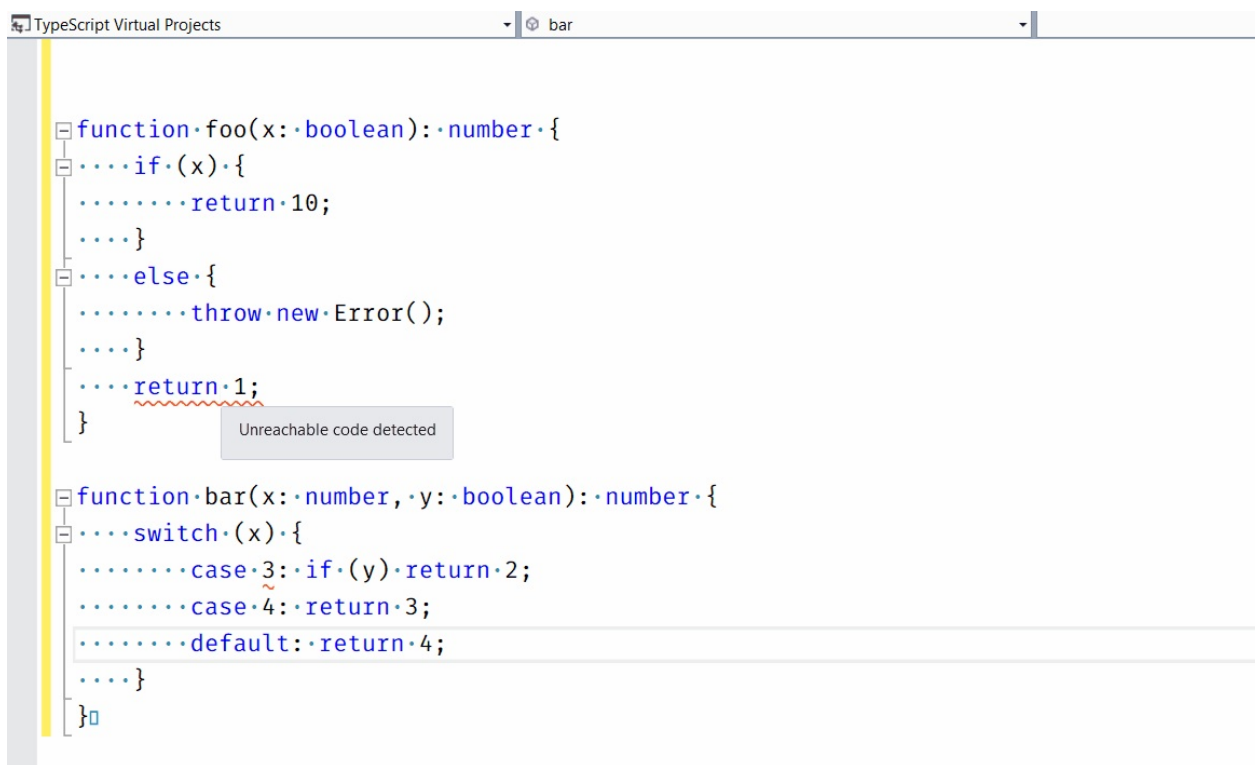
```
function assign<T extends U, U>(target: T, source: U): T {  
    for (let id in source) {  
        target[id] = source[id];  
    }  
    return target;  
}  
  
let x = { a: 1, b: 2, c: 3, d: 4 };  
assign(x, { b: 10, d: 20 });  
assign(x, { e: 0 }); // 错误
```

## 控制流错误分析

TypeScript 1.8 中引入了控制流分析来捕获开发者通常会遇到的一些错误.

详情见接下来的内容, 可以上手尝试:





## 不可及的代码

一定无法在运行时被执行的语句现在会被标记上代码不可及错误。举个例子，在无条件限制的 `return`，`throw`，`break` 或者 `continue` 后的语句被认为是不可及的。使用 `--allowUnreachableCode` 来禁用不可及代码的检测和报错。

## 例子

这里是一个简单的不可及错误的例子：

```
function f(x) {
  if (x) {
    return true;
  }
  else {
    return false;
  }

  x = 0; // 错误：检测到不可及的代码。
}
```

这个特性能捕获的一个更常见的错误是在 `return` 语句后添加换行：

```
function f() {  
    return          // 换行导致自动插入的分号  
    {  
        x: "string" // 错误：检测到不可及的代码。  
    }  
}
```

因为 JavaScript 会自动在行末结束 `return` 语句, 下面的对象字面量变成了一个代码块.

## 未使用的标签

未使用的标签也会被标记. 和不可及代码检查一样, 被使用的标签检查也是默认开启的. 使用 `--allowUnusedLabels` 来禁用未使用标签的报错.

### 例子

```
loop: while (x > 0) { // 错误：未使用的标签。  
    x++;  
}
```

## 隐式返回

JS 中没有返回值的代码分支会隐式地返回 `undefined`. 现在编译器可以将这种方式标记为隐式返回. 对于隐式返回的检查默认是被禁用的, 可以使用 `--noImplicitReturns` 来启用.

### 例子

```
function f(x) { // 错误: 不是所有分支都返回了值.
    if (x) {
        return false;
    }

    // 隐式返回了 `undefined`
}
```

## Case 语句贯穿

TypeScript 现在可以在 `switch` 语句中出现贯穿的几个非空 `case` 时报错. 这个检测默认是关闭的, 可以使用 `--noFallthroughCasesInSwitch` 启用.

### 例子

```
switch (x % 2) {
    case 0: // 错误: switch 中出现了贯穿的 case.
        console.log("even");

    case 1:
        console.log("odd");
        break;
}
```

然而, 在下面的例子中, 由于贯穿的 `case` 是空的, 并不会报错:

```
switch (x % 3) {
    case 0:
    case 1:
        console.log("Acceptable");
        break;

    case 2:
        console.log("This is *two much*!");
        break;
}
```

## React 无状态的函数组件

TypeScript 现在支持无状态的函数组件. 它是可以组合其他组件的轻量级组件.

```
// 使用参数解构和默认值轻松地定义 'props' 的类型
const Greeter = ({name = 'world'}) => <div>Hello, {name}!</div>;

// 参数可以被检验
let example = <Greeter name='TypeScript 1.8' />;
```

如果需要使用这一特性及简化的 props, 请确认使用的是[最新的 react.d.ts](#).

## 简化的 React props 类型管理

在 TypeScript 1.8 配合最新的 react.d.ts (见上方) 大幅简化了 props 的类型声明.

具体的:

- 你不再需要显式的声明 `ref` 和 `key` 或者 `extend React.Props`
- `ref` 和 `key` 属性会在所有组件上拥有正确的类型.
- `ref` 属性在无状态函数组件上会被正确地禁用.

## 在模块中扩充全局或者模块作用域

用户现在可以为任何模块进行他们想要, 或者其他人已经对其作出的扩充. 模块扩充的形式和过去的包模块一致 (例如 `declare module "foo" { }` 这样的语法), 并且可以直接嵌在你自己的模块内, 或者在另外的顶级外部包模块中.

除此之外, TypeScript 还以 `declare global { }` 的形式提供了对于全局声明的扩充. 这能使模块对像 `Array` 这样的全局类型在必要的时候进行扩充.

模块扩充的名称解析规则与 `import` 和 `export` 声明中的一致. 扩充的模块声明合并方式与在同一个文件中声明是相同的.

不论是模块扩充还是全局声明扩充都不能向顶级作用域添加新的项目 - 它们只能为已经存在的声明添加 "补丁".

## 例子

这里的 `map.ts` 可以声明它会在内部修改在 `observable.ts` 中声明的 `Observable` 类型, 添加 `map` 方法.

```
// observable.ts
export class Observable<T> {
  // ...
}
```

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

// 扩充 "../observable"
declare module "../observable" {

  // 使用接口合并扩充 'Observable' 类的定义
  interface Observable<T> {
    map<U>(proj: (el: T) => U): Observable<U>;
  }

}

Observable.prototype.map = /*...*/;
```

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

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

相似的, 在模块中全局作用域可以使用 `declare global` 声明被增强:

## 例子

```
// 确保当前文件被当做一个模块。
export {};

declare global {
    interface Array<T> {
        mapToNumbers(): number[];
    }
}

Array.prototype.mapToNumbers = function () { /* ... */ }
```

## 字符串字面量类型

接受一个特定字符串集合作为某个值的 API 并不少见. 举例来说, 考虑一个可以通过控制动画的渐变让元素在屏幕中滑动的 UI 库:

```
declare class UIElement {
    animate(options: AnimationOptions): void;
}

interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: string; // 可以是 "ease-in", "ease-out", "ease-in-out"
}
```

然而, 这容易产生错误 - 当用户错误不小心错误拼写了一个合法的值时, 并没有任何提示:

```
// 没有报错
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-in-out" });
```

在 TypeScript 1.8 中, 我们新增了字符串字面量类型. 这些类型和字符串字面量的写法一致, 只是写在类型的位置.

用户现在可以确保类型系统会捕获这样的错误. 这里是我们使用了字符串字面量类型的新的 `AnimationOptions` :

```
interface AnimationOptions {  
    deltaX: number;  
    deltaY: number;  
    easing: "ease-in" | "ease-out" | "ease-in-out";  
}  
  
// 错误: 类型 '"ease-inout"' 不能复制给类型 '"ease-in" | "ease-out" |  
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-inout";
```

## 更好的联合/交叉类型接口

TypeScript 1.8 优化了源类型和目标类型都是联合或者交叉类型的情况下的类型推导. 举例来说, 当从 `string | string[]` 推导到 `string | T` 时, 我们将类型拆解为 `string[]` 和 `T`, 这样就可以将 `string[]` 推导为 `T`.

### 例子

```
type Maybe<T> = T | void;

function isDefined<T>(x: Maybe<T>): x is T {
    return x !== undefined && x !== null;
}

function isUndefined<T>(x: Maybe<T>): x is void {
    return x === undefined || x === null;
}

function getOrElse<T>(x: Maybe<T>, defaultValue: T): T {
    return isDefined(x) ? x : defaultValue;
}

function test1(x: Maybe<string>) {
    let x1 = getOrElse(x, "Undefined"); // string
    let x2 = isDefined(x) ? x : "Undefined"; // string
    let x3 = isUndefined(x) ? "Undefined" : x; // string
}

function test2(x: Maybe<number>) {
    let x1 = getOrElse(x, -1); // number
    let x2 = isDefined(x) ? x : -1; // number
    let x3 = isUndefined(x) ? -1 : x; // number
}
```

## 使用 `--outFile` 合并 `AMD` 和 `System` 模块

在使用 `--module amd` 或者 `--module system` 的同时制定 `--outFile` 将会把所有参与编译的模块合并为单个包括了多个模块闭包的输出文件。

每一个模块都会根据其相对于 `rootDir` 的位置被计算出自己的模块名称。

## 例子



```
// 文件 src/a.ts
import * as B from "./lib/b";
export function createA() {
    return B.createB();
}
```

```
// 文件 src/lib/b.ts
export function createB() {
    return { };
}
```

结果为:

```
define("lib/b", ["require", "exports"], function (require, exports) {
    "use strict";
    function createB() {
        return {};
    }
    exports.createB = createB;
});
define("a", ["require", "exports", "lib/b"], function (require, exports) {
    "use strict";
    function createA() {
        return B.createB();
    }
    exports.createA = createA;
});
```

## 支持 **SystemJS** 使用 **default** 导入

像 SystemJS 这样的模块加载器将 CommonJS 模块做了包装并暴露为 **default** ES6 导入项. 这使得在 SystemJS 和 CommonJS 的实现由于不同加载器不同的模块导出方式不能共享定义.

设置新的编译选项 `--allowSyntheticDefaultImports` 指明模块加载器会进行导入的 `.ts` 或 `.d.ts` 中未指定的某种类型的默认导入项构建. 编译器会由此推断存在一个 `default` 导出项和整个模块自己一致.

此选项在 `System` 模块默认开启.

## 允许循环中被引用的 `let / const`

之前这样会报错, 现在由 TypeScript 1.8 支持. 循环中被函数引用的 `let / const` 声明现在会被输出为与 `let / const` 更新语义相符的代码.

### 例子

```
let list = [];  
for (let i = 0; i < 5; i++) {  
    list.push(() => i);  
}  
  
list.forEach(f => console.log(f()));
```

被编译为:

```
var list = [];  
var _loop_1 = function(i) {  
    list.push(function () { return i; });  
};  
for (var i = 0; i < 5; i++) {  
    _loop_1(i);  
}  
list.forEach(function (f) { return console.log(f()); });
```

然后结果是:

```
0  
1  
2  
3  
4
```

## 改进的 `for..in` 语句检查

过去 `for..in` 变量的类型被推断为 `any`，这使得编译器忽略了 `for..in` 语句内的一些不合法的使用。

从 TypeScript 1.8 开始：

- 在 `for..in` 语句中的变量隐含类型为 `string`。
- 当一个有数字索引签名对应类型 `T` (比如一个数组) 的对象被一个 `for..in` 索引有数字索引签名并且没有字符串索引签名 (比如还是数组) 的对象的变量索引，产生的值的类型为 `T`。

## 例子

```
var a: MyObject[];  
for (var x in a) { // x 的隐含类型为 string  
    var obj = a[x]; // obj 的类型为 MyObject  
}
```

## 模块现在输出时会加上 `"use strict;"`

对于 ES6 来说模块始终以严格模式被解析，但这一点过去对于非 ES6 目标在生成的代码中并没有遵循。从 TypeScript 1.8 开始，输出的模块总会为严格模式。由于多数严格模式下的错误也是 TS 编译时的错误，多数代码并不会有可见的改动，但是这也意味着有一些东西可能在运行时没有征兆地失败，比如赋值给 `NaN` 现在会有运行时错误。你可以参考这篇 [MDN 上的文章](#) 查看详细的严格模式与非严格模式的区别列表。

## 使用 `--allowJs` 加入 `.js` 文件

经常在项目中会有外部的非 TypeScript 编写的源文件. 一种方式是将 JS 代码转换为 TS 代码, 但这时又希望将所有 JS 代码和新的 TS 代码的输出一起打包为一个文件.

`.js` 文件现在允许作为 `tsc` 的输入文件. TypeScript 编译器会检查 `.js` 输入文件的语法错误, 并根据 `--target` 和 `--module` 选项输出对应的代码. 输出也会和其他 `.ts` 文件一起. `.js` 文件的 source maps 也会像 `.ts` 文件一样被生成.

## 使用 `--reactNamespace` 自定义 JSX 工厂

在使用 `--jsx react` 的同时使用 `--reactNamespace <JSX 工厂名称>` 可以允许使用一个不同的 JSX 工厂代替默认的 `React`.

新的工厂名称会被用来调用 `createElement` 和 `__spread` 方法.

### 例子

```
import {jsxFactory} from "jsxFactory";

var div = <div>Hello JSX!</div>
```

编译参数:

```
tsc --jsx react --reactNamespace jsxFactory --m commonJS
```

结果:

```
"use strict";
var jsxFactory_1 = require("jsxFactory");
var div = jsxFactory_1.jsxFactory.createElement("div", null, "Hello
```

## 基于 **this** 的类型收窄

TypeScript 1.8 为类和接口方法扩展了用户定义的类型收窄函数。

`this is T` 现在是类或接口方法的合法的返回值类型标注。当在类型收窄的位置使用时 (比如 `if` 语句), 函数调用表达式的目标对象的类型会被收窄为 `T`。

### 例子

```
class FileSystemObject {
    isFile(): this is File { return this instanceof File; }
    isDirectory(): this is Directory { return this instanceof Directory; }
    isNetworked(): this is (Networked & this) { return this.networked; }
    constructor(public path: string, private networked: boolean) {}
}

class File extends FileSystemObject {
    constructor(path: string, public content: string) { super(path, false); }
}

class Directory extends FileSystemObject {
    children: FileSystemObject[];
}

interface Networked {
    host: string;
}

let fso: FileSystemObject = new File("foo/bar.txt", "foo");
if (fso.isFile()) {
    fso.content; // fso 是 File
}
else if (fso.isDirectory()) {
    fso.children; // fso 是 Directory
}
else if (fso.isNetworked()) {
    fso.host; // fso 是 networked
}
```

## 官方的 TypeScript NuGet 包

从 TypeScript 1.8 开始, 将为 TypeScript 编译器 ( `tsc.exe` ) 和 MSBuild 整合 ( `Microsoft.TypeScript.targets` 和 `Microsoft.TypeScript.Tasks.dll` ) 提供官方的 NuGet 包.

稳定版本可以在这里下载:

- [Microsoft.TypeScript.Compiler](#)
- [Microsoft.TypeScript.MSBuild](#)

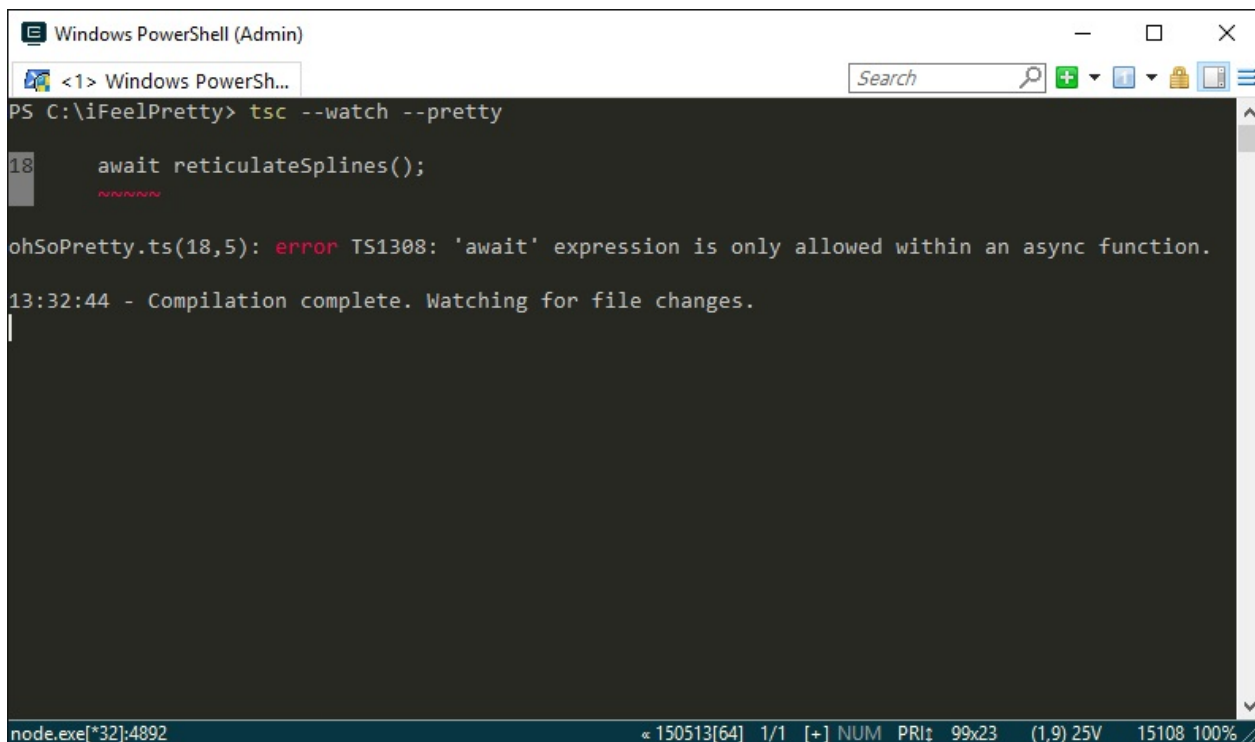
与此同时, 和每日 npm 包对应的每日 NuGet 包可以在 <https://myget.org> 下载:

- [TypeScript-Preview](#)

### tsc 错误信息更美观

我们理解大量单色的输出并不直观. 颜色可以帮助识别信息的始末, 这些视觉上的线索在处理复杂的错误信息时非常重要.

通过传递 `--pretty` 命令行选项, TypeScript 会给出更丰富的输出, 包含错误发生的上下文.



```
Windows PowerShell (Admin)
PS C:\iFeelPretty> tsc --watch --pretty

18      await reticulateSplines();
      ~~~~~
ohSoPretty.ts(18,5): error TS1308: 'await' expression is only allowed within an async function.

13:32:44 - Compilation complete. Watching for file changes.
```

## 高亮 VS 2015 中的 JSX 代码

在 TypeScript 1.8 中, JSX 标签现在可以在 Visual Studio 2015 中被分别和高亮.



通过 工具 -> 选项 -> 环境 -> 字体与颜色 页面在 VB XML 颜色和字体设置中还可以进一步改变字体和颜色来自定义.

## --project ( -p ) 选项现在接受任意文件路径

--project 命令行选项过去只接受包含了 tsconfig.json 文件的文件夹. 考虑到不同的构建场景, 应该允许 --project 指向任何兼容的 JSON 文件. 比如说, 一个用户可能会希望为 Node 5 编译 CommonJS 的 ES 2015, 为浏览器编译 AMD 的 ES5. 现在少了这项限制, 用户可以更容易地直接使用 tsc 管理不同的构建目标, 无需再通过一些奇怪的方式, 比如将多个 tsconfig.json 文件放在不同的目录中.

如果参数是一个路径, 行为保持不变 - 编译器会尝试在该目录下寻找名为 tsconfig.json 的文件.

## 允许 tsconfig.json 中的注释

为配置添加文档是很棒的! tsconfig.json 现在支持单行和多行注释.

```
{
  "compilerOptions": {
    "target": "ES2015", // 跑在 node v5 上, 呀!
    "sourceMap": true   // 让调试轻松一些
  },
  /*
   * 排除的文件
   */
  "exclude": [
    "file.d.ts"
  ]
}
```

## 支持输出到 **IPC** 驱动的文件

TypeScript 1.8 允许用户将 `--outFile` 参数和一些特殊的文件系统对象一起使用, 比如命名的管道 (pipe), 设备 (devices) 等.

举个例子, 在很多与 Unix 相似的系统上, 标准输出流可以通过文件 `/dev/stdout` 访问.

```
tsc foo.ts --outFile /dev/stdout
```

这一特性也允许输出给其他命令.

比如说, 我们可以输出生成的 JavaScript 给一个像 `pretty-js` 这样的格式美化工具:

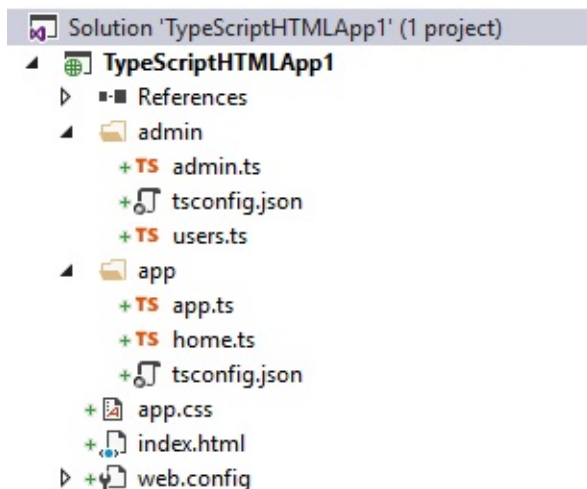
```
tsc foo.ts --outFile /dev/stdout | pretty-js
```

## 改进了 **Visual Studio 2015** 中对 `tsconfig.json` 的支持

TypeScript 1.8 允许在任何种类的项目中使用 `tsconfig.json` 文件. 包括 ASP.NET v4 项目, 控制台应用, 以及用 *TypeScript* 开发的 *HTML* 应用. 与此同时, 你可以添加不止一个 `tsconfig.json` 文件, 其中每一个都会作为项目的一部分被



构建. 这使得你可以在不使用多个不同项目的情况下为应用的不同部分使用不同的配置.



当项目中添加了 `tsconfig.json` 文件时, 我们还禁用了项目属性页面. 也就是说所有配置的改变必须在 `tsconfig.json` 文件中进行.

## 一些限制

- 如果你添加了一个 `tsconfig.json` 文件, 不在其上下文中的 TypeScript 文件不会被编译.
- Apache Cordova 应用依然有单个 `tsconfig.json` 文件的限制, 而这个文件必须在根目录或者 `scripts` 文件夹.
- 多数项目类型中都没有 `tsconfig.json` 的模板.

# TypeScript 1.7

## 支持 `async / await` 编译到 ES6 (Node v4+)

TypeScript 目前在已经原生支持 ES6 generator 的引擎 (比如 Node v4 及以上版本) 上支持异步函数. 异步函数前置 `async` 关键字; `await` 会暂停执行, 直到一个异步函数执行后返回的 `promise` 被 `fulfill` 后获得它的值.

### 例子

在下面的例子中, 输入的内容将会延时 200 毫秒逐个打印:

```
"use strict";

// printDelayed 返回值是一个 'Promise<void>'
async function printDelayed(elements: string[]) {
    for (const element of elements) {
        await delay(200);
        console.log(element);
    }
}

async function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
    console.log();
    console.log("打印每一个内容!");
});
```

查看 [Async Functions](#) 一文了解更多.

## 支持同时使用 `--target ES6` 和 `--module`

TypeScript 1.7 将 `ES6` 添加到了 `--module` 选项支持的选项的列表, 当编译到 `ES6` 时允许指定模块类型. 这让使用具体运行时中你需要的特性更加灵活.

### 例子

```
{
  "compilerOptions": {
    "module": "amd",
    "target": "es6"
  }
}
```

## `this` 类型

在方法中返回当前对象 (也就是 `this`) 是一种创建链式 API 的常见方式. 比如, 考虑下面的 `BasicCalculator` 模块:

```
export default class BasicCalculator {
  public constructor(protected value: number = 0) { }

  public currentValue(): number {
    return this.value;
  }

  public add(operand: number) {
    this.value += operand;
    return this;
  }

  public subtract(operand: number) {
    this.value -= operand;
    return this;
  }

  public multiply(operand: number) {
    this.value *= operand;
    return this;
  }

  public divide(operand: number) {
    this.value /= operand;
    return this;
  }
}
```

使用者可以这样表述 `2 * 5 + 1` :

```
import calc from "./BasicCalculator";

let v = new calc(2)
  .multiply(5)
  .add(1)
  .currentValue();
```

这使得这么一种优雅的编码方式成为可能; 然而, 对于想要去继承

`BasicCalculator` 的类来说有一个问题. 想象使用者可能需要编写一个 `ScientificCalculator` :

```
import BasicCalculator from "./BasicCalculator";

export default class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }

    public square() {
        this.value = this.value ** 2;
        return this;
    }

    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
}
```

因为 `BasicCalculator` 的方法返回了 `this`, TypeScript 过去推断的类型是 `BasicCalculator`, 如果在 `ScientificCalculator` 的实例上调用属于 `BasicCalculator` 的方法, 类型系统不能很好地处理.

举例来说:

```
import calc from "./ScientificCalculator";

let v = new calc(0.5)
    .square()
    .divide(2)
    .sin()    // Error: 'BasicCalculator' 没有 'sin' 方法.
    .currentValue();
```

这已经不再是问题 - TypeScript 现在在类的实例方法中, 会将 `this` 推断为一个特殊的叫做 `this` 的类型. `this` 类型也就写作 `this`, 可以大致理解为 "方法调用时点左边的类型".

`this` 类型在描述一些使用了 `mixin` 风格继承的库 (比如 `Ember.js`) 的交叉类型:

```
interface MyType {  
    extend<T>(other: T): this & T;  
}
```

## ES7 幂运算符

TypeScript 1.7 支持将在 ES7/ES2016 中增加的幂运算符: `**` 和 `**=`. 这些运算符会被转换为 ES3/ES5 中的 `Math.pow`.

### 举例

```
var x = 2 ** 3;  
var y = 10;  
y **= 2;  
var z = -(4 ** 3);
```

会生成下面的 JavaScript:

```
var x = Math.pow(2, 3);  
var y = 10;  
y = Math.pow(y, 2);  
var z = -(Math.pow(4, 3));
```

## 改进对象字面量解构的检查

TypeScript 1.7 使对象和数组字面量解构初始值的检查更加直观和自然.

当一个对象字面量通过与之对应的对象解构绑定推断类型时:

- 对象解构绑定中有默认值的属性对于对象字面量来说可选.

- 对象解构绑定中的属性如果在对象字面量中没有匹配的值, 则该属性必须有默认值, 并且会被添加到对象字面量的类型中.
- 对象字面量中的属性必须在对象解构绑定中存在.

当一个数组字面量通过与之对应的数组解构绑定推断类型时:

- 数组解构绑定中的元素如果在数组字面量中没有匹配的值, 则该元素必须有默认值, 并且会被添加到数组字面量的类型中.

## 举例

```
// f1 的类型为 (arg?: { x?: number, y?: number }) => void
function f1({ x = 0, y = 0 } = {}) { }
```

// And can be called as:

```
f1();
f1({});
f1({ x: 1 });
f1({ y: 1 });
f1({ x: 1, y: 1 });
```

```
// f2 的类型为 (arg?: (x: number, y?: number) => void
function f2({ x, y = 0 } = { x: 0 }) { }
```

```
f2();
f2({});           // 错误, x 非可选
f2({ x: 1 });
f2({ y: 1 });     // 错误, x 非可选
f2({ x: 1, y: 1 });
```

## 装饰器 (decorators) 支持的编译目标版本增加 ES3

装饰器现在可以编译到 ES3. TypeScript 1.7 在 `__decorate` 函数中移除了 ES5 中增加的 `reduceRight`. 相关改动也内联了对

`Object.getOwnPropertyDescriptor` 和 `Object.defineProperty` 的调用, 并  
向后兼容, 使 ES5 的输出可以消除前面提到的 `Object` 方法的重复<sup>[1]</sup>.

# TypeScript 1.6

## JSX 支持

JSX 是一种可嵌入的类似 XML 的语法. 它将最终被转换为合法的 JavaScript, 但转换的语义和具体实现有关. JSX 随着 React 流行起来, 也出现在其他应用中.

TypeScript 1.6 支持 JavaScript 文件中 JSX 的嵌入, 类型检查, 以及直接编译为 JavaScript 的选项.

### 新的 `.tsx` 文件扩展名和 `as` 运算符

TypeScript 1.6 引入了新的 `.tsx` 文件扩展名. 这一扩展名一方面允许 TypeScript 文件中的 JSX 语法, 一方面将 `as` 运算符作为默认的类型转换方式 (避免 JSX 表达式和 TypeScript 前置类型转换运算符之间的歧义). 比如:

```
var x = <any> foo;  
// 与如下等价:  
var x = foo as any;
```

## 使用 React

使用 React 及 JSX 支持, 你需要使用 [React 类型声明](#). 这些类型定义了 `JSX` 命名空间, 以便 TypeScript 能正确地检查 React 的 JSX 表达式. 比如:



```
/// <reference path="react.d.ts" />

interface Props {
  name: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent name="bar" />; // 没问题
<MyComponent name={0} />; // 错误, `name` 不是一个数字
```

## 使用其他 **JSX** 框架

JSX 元素的名称和属性是根据 `JSX` 命名空间来检验的. 请查看 [JSX](#) 页面了解如何为自己的框架定义 `JSX` 命名空间.

## 编译输出

TypeScript 支持两种 `JSX` 模式: `preserve` (保留) 和 `react`.

- `preserve` 模式将会在输出中保留 `JSX` 表达式, 使之后的转换步骤可以处理. 并且输出的文件扩展名为 `.jsx`.
- `react` 模式将会生成 `React.createElement`, 不再需要再通过 `JSX` 转换即可运行, 输出的文件扩展名为 `.js`.

查看 [JSX](#) 页面了解更多 `JSX` 在 TypeScript 中的使用.

## 交叉类型 (intersection types)

TypeScript 1.6 引入了交叉类型作为联合类型 (union types) 逻辑上的补充. 联合类型 `A | B` 表示一个类型为 `A` 或 `B` 的实体, 而交叉类型 `A & B` 表示一个类型同时为 `A` 或 `B` 的实体.

## 例子

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 });
var s = x.a;
var n = x.b;
```

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
interface A { a: string }
interface B { b: string }
interface C { c: string }

var abc: A & B & C;
abc.a = "hello";
abc.b = "hello";
abc.c = "hello";
```

查看 [issue #1256](#) 了解更多.

## 本地类型声明

本地的类, 接口, 枚举和类型别名现在可以在函数声明中出现. 本地类型为块级作用域, 与 `let` 和 `const` 声明的变量类似. 比如说:

```
function f() {  
  if (true) {  
    interface T { x: number }  
    let v: T;  
    v.x = 5;  
  }  
  else {  
    interface T { x: string }  
    let v: T;  
    v.x = "hello";  
  }  
}
```

推导出的函数返回值类型可能在函数内部声明的. 调用函数的地方无法引用到这样的本地类型, 但是它当然能从类型结构上匹配. 比如:

```
interface Point {
    x: number;
    y: number;
}

function getPointFactory(x: number, y: number) {
    class P {
        x = x;
        y = y;
    }
    return P;
}

var PointZero = getPointFactory(0, 0);
var PointOne = getPointFactory(1, 1);
var p1 = new PointZero();
var p2 = new PointZero();
var p3 = new PointOne();
```

本地的类型可以引用类型参数, 本地的类和接口本身即可能是泛型. 比如:

```
function f3() {
    function f<X, Y>(x: X, y: Y) {
        class C {
            public x = x;
            public y = y;
        }
        return C;
    }
    let C = f(10, "hello");
    let v = new C();
    let x = v.x; // number
    let y = v.y; // string
}
```

## 类表达式

TypeScript 1.6 增加了对 ES6 类表达式的支持. 在一个类表达式中, 类的名称是可选的, 如果指明, 作用域仅限于类表达式本身. 这和函数表达式可选的名称类似. 在类表达式外无法引用其实例类型, 但是自然也能够从类型结构上匹配. 比如:

```
let Point = class {  
    constructor(public x: number, public y: number) { }  
    public length() {  
        return Math.sqrt(this.x * this.x + this.y * this.y);  
    }  
};  
var p = new Point(3, 4); // p has anonymous class type  
console.log(p.length());
```

## 继承表达式

TypeScript 1.6 增加了对类继承任意值为一个构造函数的表达式的支持. 这样一来内建的类型也可以在类的声明中被继承.

`extends` 语句过去需要指定一个类型引用, 现在接受一个可选类型参数的表达式. 表达式的类型必须为有至少一个构造函数签名的构造函数, 并且需要和 `extends` 语句中类型参数数量一致. 匹配的构造函数签名的返回值类型是类实例类型继承的基类型. 如此一来, 这使得普通的类和与类相似的表达式可以在 `extends` 语句中使用.

一些例子:

```
// 继承内建类

class MyArray extends Array<number> { }
class MyError extends Error { }

// 继承表达式类

class ThingA {
  getGreeting() { return "Hello from A"; }
}

class ThingB {
  getGreeting() { return "Hello from B"; }
}

interface Greeter {
  getGreeting(): string;
}

interface GreeterConstructor {
  new (): Greeter;
}

function getGreeterBase(): GreeterConstructor {
  return Math.random() >= 0.5 ? ThingA : ThingB;
}

class Test extends getGreeterBase() {
  sayHello() {
    console.log(this.getGreeting());
  }
}
```

## abstract (抽象的) 类和方法

TypeScript 1.6 为类和方法增加了 `abstract` 关键字。一个抽象类允许没有实现的方法，并且不能被构造。

## 例子

```
abstract class Base {
    abstract getThing(): string;
    getOtherThing() { return 'hello'; }
}

let x = new Base(); // 错误, 'Base' 是抽象的

// 错误, 必须也为抽象类, 或者实现 'getThing' 方法
class Derived1 extends Base { }

class Derived2 extends Base {
    getThing() { return 'hello'; }
    foo() {
        super.getThing(); // 错误: 不能调用 'super' 的抽象方法
    }
}

var x = new Derived2(); // 正确
var y: Base = new Derived2(); // 同样正确
y.getThing(); // 正确
y.getOtherThing(); // 正确
```

## 泛型别名

TypeScript 1.6 中, 类型别名支持泛型. 比如:

```
type Lazy<T> = T | (() => T);

var s: Lazy<string>;
s = "eager";
s = () => "lazy";

interface Tuple<A, B> {
  a: A;
  b: B;
}

type Pair<T> = Tuple<T, T>;
```

## 更严格的对象字面量赋值检查

为了能发现多余或者错误拼写的属性, TypeScript 1.6 使用了更严格的对象字面量检查. 确切地说, 在将一个新的对象字面量赋值给一个变量, 或者传递给类型非空的参数时, 如果对象字面量的属性在目标类型中不存在, 则会视为错误.

### 例子

```
var x: { foo: number };
x = { foo: 1, baz: 2 }; // 错误, 多余的属性 `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 }; // 错误, 多余或者拼错的属性 `baz`
```

一个类型可以通过包含一个索引签名来现实指明未出现在类型中的属性是被允许的.

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 }; // 现在 `baz` 匹配了索引签名
```

## ES6 生成器 (generators)

TypeScript 1.6 添加了对于 ES6 输出的生成器支持.



一个生成器函数可以有返回值类型标注, 就像普通的函数. 标注表示生成器函数返回的生成器的类型. 这里有个例子:

```
function *g(): Iterable<string> {  
    for (var i = 0; i < 100; i++) {  
        yield ""; // string 可以赋值给 string  
    }  
    yield * otherStringGenerator(); // otherStringGenerator 必须可遍  
}
```

没有标注类型的生成器函数会有自动推演的类型. 在下面的例子中, 类型会由 `yield` 语句推演出来:

```
function *g() {  
    for (var i = 0; i < 100; i++) {  
        yield ""; // 推导出 string  
    }  
    yield * otherStringGenerator(); // 推导出 otherStringGenerator  
}
```

## 对 `async` (异步) 函数的试验性支持

TypeScript 1.6 增加了编译到 ES6 时对 `async` 函数试验性的支持. 异步函数会执行一个异步的操作, 在等待的同时不会阻塞程序的正常运行. 这是通过与 ES6 兼容的 `Promise` 实现完成的, 并且会将函数体转换为支持在等待的异步操作完成时继续的形式.

由 `async` 标记的函数或方法被称作异步函数. 这个标记告诉了编译器该函数体需要被转换, 关键字 `await` 则应该被当做一个一元运算符, 而不是标示符. 一个异步函数必须返回类型与 `Promise` 兼容的值. 返回值类型的推断只能在有一个全局的, 与 ES6 兼容的 `Promise` 类型时使用.

### 例子

```
var p: Promise<number> = /* ... */;
async function fn(): Promise<number> {
  var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "number"
  return 1 + i;
}

var a = async (): Promise<number> => 1 + await p; // 暂停执行.
var a = async () => 1 + await p; // 暂停执行. 使用 --target ES6 选项编译
var fe = async function(): Promise<number> {
  var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "number"
  return 1 + i;
}

class C {
  async m(): Promise<number> {
    var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "number"
    return 1 + i;
  }

  async get p(): Promise<number> {
    var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "number"
    return 1 + i;
  }
}
```

## 每天发布新版本

由于并不算严格意义上的语言变化<sup>[2]</sup>, 每天的新版本可以使用如下命令安装获得:

```
npm install -g typescript@next
```

## 对模块解析逻辑的调整

从 1.6 开始, TypeScript 编译器对于 "commonjs" 的模块解析会使用一套不同的规则. 这些规则 尝试模仿 Node 查找模块的过程. 这就意味着 node 模块可以包含它的类型信息, 并且 TypeScript 编译器可以找到这些信息. 不过用户可以通过使用 `--moduleResolution` 命令行选项覆盖模块解析规则. 支持的值有:

- 'classic' - TypeScript 1.6 以前的编译器使用的模块解析规则
- 'node' - 与 node 相似的模块解析

## 合并外围类和接口的声明

外围类的实例类型可以通过接口声明来扩展. 类构造函数对象不会被修改. 比如说:

```
declare class Foo {
    public x : number;
}

interface Foo {
    y : string;
}

function bar(foo : Foo) {
    foo.x = 1; // 没问题, 在类 Foo 中有声明
    foo.y = "1"; // 没问题, 在接口 Foo 中有声明
}
```

## 用户定义的类型收窄函数

TypeScript 1.6 增加了一个新的在 `if` 语句中收窄变量类型的方式, 作为对 `typeof` 和 `instanceof` 的补充. 用户定义的类型收窄函数的返回值类型标注形式为 `x is T`, 这里 `x` 是函数声明中的形参, `T` 是任何类型. 当一个用户定义的类型收窄函数在 `if` 语句中被传入某个变量执行时, 该变量的类型会被收窄到 `T`.

### 例子

```
function isCat(a: any): a is Cat {  
    return a.name === 'kitty';  
}  
  
var x: Cat | Dog;  
if(isCat(x)) {  
    x.meow(); // 那么, x 在这个代码块内是 Cat 类型  
}
```

## tsconfig.json 对 exclude 属性的支持

一个没有写明 `files` 属性的 `tsconfig.json` 文件 (默认会引用所有子目录下的 `*.ts` 文件) 现在可以包含一个 `exclude` 属性, 指定需要在编译中排除的文件或者目录列表. `exclude` 属性必须是一个字符串数组, 其中每一个元素指定对应的一个文件或者文件夹名称对于 `tsconfig.json` 文件所在位置的相对路径. 举例来说:

```
{  
    "compilerOptions": {  
        "out": "test.js"  
    },  
    "exclude": [  
        "node_modules",  
        "test.ts",  
        "utils/t2.ts"  
    ]  
}
```

`exclude` 列表不支持通配符. 仅仅可以是文件或者目录的列表.

## --init 命令行选项

在一个目录中执行 `tsc --init` 可以在该目录中创建一个包含了默认值的 `tsconfig.json`. 可以通过一并传递其他选项来生成初始的 `tsconfig.json`.

# TypeScript 1.5

## ES6 模块

TypeScript 1.5 支持 ECMAScript 6 (ES6) 模块. ES6 模块可以看做之前 TypeScript 的外部模块换上了新的语法: ES6 模块是分开加载的源文件, 这些文件还可能引入其他模块, 并且导出部分供外部可访问. ES6 模块新增了几种导入和导出声明. 我们建议使用 TypeScript 开发的库和应用能够更新到新的语法, 但不做强制要求. 新的 ES6 模块语法和 TypeScript 原来的内部和外部模块结构同时被支持, 如果需要也可以混合使用.

### 导出声明

作为 TypeScript 已有的 `export` 前缀支持, 模块成员也可以使用单独导出的声明导出, 如果需要, `as` 语句可以指定不同的导出名称.

```
interface Stream { ... }  
function writeToStream(stream: Stream, data: string) { ... }  
export { Stream, writeToStream as write }; // writeToStream 导出为
```

引入声明也可以使用 `as` 语句来指定一个不同的导入名称. 比如:

```
import { read, write, standardOutput as stdout } from "./inout";  
var s = read(stdout);  
write(stdout, s);
```

作为单独导入的候选项, 命名空间导入可以导入整个模块:

```
import * as io from "./inout";  
var s = io.read(io.standardOutput);  
io.write(io.standardOutput, s);
```

## 重新导出

使用 `from` 语句一个模块可以复制指定模块的导出项到当前模块, 而无需创建本地名称.

```
export { read, write, standardOutput as stdout } from "./inout";
```

`export *` 可以用来重新导出另一个模块的所有导出项. 在创建一个聚合了其他几个模块导出项的模块时很方便.

```
export function transform(s: string): string { ... }  
export * from "./mod1";  
export * from "./mod2";
```

## 默认导出项

一个 `export default` 声明表示一个表达式是这个模块的默认导出项.

```
export default class Greeter {  
    sayHello() {  
        console.log("Greetings!");  
    }  
}
```

对应的可以使用默认导入:

```
import Greeter from "./greeter";  
var g = new Greeter();  
g.sayHello();
```

## 无导入加载

"无导入加载" 可以被用来加载某些只需要其副作用的模块.

```
import "./polyfills";
```

了解更多关于模块的信息, 请参见 [ES6 模块支持规范](#).

## 声明与赋值的解构

TypeScript 1.5 添加了对 ES6 解构声明与赋值的支持.

### 解构

解构声明会引入一个或多个命名变量, 并且初始化它们的值为对象的属性或者数组的元素对应的值.

比如说, 下面的例子声明了变量 `x`, `y` 和 `z`, 并且分别将它们的值初始化为 `getSomeObject().x`, `getSomeObject().y` 和 `getSomeObject().z`:

```
var { x, y, z } = getSomeObject();
```

解构声明也可以用于从数组中得到值.

```
var [x, y, z = 10] = getSomeArray();
```

相似的, 解构可以用在函数的参数声明中:

```
function drawText({ text = "", location: [x, y] = [0, 0], bold = false }) {
    // 画出文本
}

// 以一个对象字面量为参数调用 drawText
var item = { text: "someText", location: [1, 2, 3], style: "italics" };
drawText(item);
```

### 赋值

解构也可以被用于普通的赋值表达式. 举例来讲, 交换两个变量的值可以被写作一个解构赋值:

```
var x = 1;
var y = 2;
[x, y] = [y, x];
```

## namespace (命名空间) 关键字

过去 TypeScript 中 `module` 关键字既可以定义 "内部模块", 也可以定义 "外部模块"; 这让刚刚接触 TypeScript 的开发者有些困惑. "内部模块" 的概念更接近于大部分人眼中的命名空间; 而 "外部模块" 对于 JS 来讲, 现在也就是模块了.

注意: 之前定义内部模块的语法依然被支持.

之前:

```
module Math {
    export function add(x, y) { ... }
}
```

之后:

```
namespace Math {
    export function add(x, y) { ... }
}
```

## let 和 const 的支持

ES6 的 `let` 和 `const` 声明现在支持编译到 ES3 和 ES5.

## Const



```
const MAX = 100;
```

```
++MAX; // 错误：自增/减运算符不能用于一个常量
```

## 块级作用域

```
if (true) {  
    let a = 4;  
    // 使用变量 a  
}  
else {  
    let a = "string";  
    // 使用变量 a  
}
```

```
alert(a); // 错误：变量 a 在当前作用域未定义
```

## for...of 的支持

TypeScript 1.5 增加了 ES6 `for...of` 循环编译到 ES3/ES5 时对数组的支持, 以及编译到 ES6 时对满足 `Iterator` 接口的全面支持.

### 例子:

TypeScript 编译器会转译 `for...of` 数组到具有语义的 ES3/ES5 JavaScript (如果被设置为编译到这些版本).

```
for (var v of expr) { }
```

会输出为:

```
for (var _i = 0, _a = expr; _i < _a.length; _i++) {  
    var v = _a[_i];  
}
```

## 装饰器

TypeScript 装饰器是局域 [ES7 装饰器](#) 提案的。

一个装饰器是：

- 一个表达式
- 并且值为一个函数
- 接受 `target` , `name` , 以及属性描述对象作为参数
- 可选返回一个会被应用到目标对象的属性描述对象

了解更多, 请参见 [装饰器](#) 提案。

### 例子:

装饰器 `readonly` 和 `enumerable(false)` 会在属性 `method` 添加到类 `C` 上之前被应用. 这使得装饰器可以修改其实现, 具体到这个例子, 设置了 `descriptor` 为 `writable: false` 以及 `enumerable: false` .

```
class C {
  @readonly
  @enumerable(false)
  method() { }
}

function readonly(target, key, descriptor) {
  descriptor.writable = false;
}

function enumerable(value) {
  return function (target, key, descriptor) {
    descriptor.enumerable = value;
  }
}
```

## 计算属性

使用动态的属性初始化一个对象可能会很麻烦. 参考下面的例子:

```
type NeighborMap = { [name: string]: Node };
type Node = { name: string; neighbors: NeighborMap; }

function makeNode(name: string, initialNeighbor: Node): Node {
    var neighbors: NeighborMap = {};
    neighbors[initialNeighbor.name] = initialNeighbor;
    return { name: name, neighbors: neighbors };
}
```

这里我们需要创建一个包含了 `neighbor-map` 的变量, 便于我们初始化它. 使用 TypeScript 1.5, 我们可以让编译器来干重活:

```
function makeNode(name: string, initialNeighbor: Node): Node {
    return {
        name: name,
        neighbors: {
            [initialNeighbor.name]: initialNeighbor
        }
    }
}
```

## 指出 **UMD** 和 **System** 模块输出

作为 **AMD** 和 **CommonJS** 模块加载器的补充, TypeScript 现在支持输出为 **UMD** ([Universal Module Definition](#)) 和 **System** 模块的格式.

用法:

```
tsc --module umd
```

以及

```
tsc --module system
```

## Unicode 字符串码位转义

ES6 中允许用户使用单个转义表示一个 Unicode 码位.

举个例子, 考虑我们需要转义一个包含了字符 " 的字符串. 在 UTF-16/USC2 中, " 被表示为一个代理对, 意思就是它被编码为一对 16 位值的代码单元, 具体来说就是 `0xD842` 和 `0xDFB7`. 之前这意味着你必须将该码位转义为 `"\uD842\uDFB7"`. 这样做有一个重要的问题, 就事很难讲两个独立的字符同一个代理对区分开来.

通过 ES6 的码位转义, 你可以在字符串或模板字符串中清晰地通过一个转义表示一个确切的字符: `"\u{20bb7}"`. TypeScript 在编译到 ES3/ES5 时会将该字符串输出为 `"\uD842\uDFB7"`.

## 标签模板字符串编译到 ES3/ES5

TypeScript 1.4 中, 我们添加了模板字符串编译到所有 ES 版本的支持, 并且支持标签模板字符串编译到 ES6. 得益于 [@ivogabe](#) 的大量付出, 我们填补了标签模板字符串对编译到 ES3/ES5 的支持.

当编译到 ES3/ES5 时, 下面的代码:

```
function oddRawStrings(strs: TemplateStringsArray, n1, n2) {
    return strs.raw.filter((raw, index) => index % 2 === 1);
}

oddRawStrings `Hello \n${123} \t ${456}\n world`
```

会被输出为:

```
function oddRawStrings(strs, n1, n2) {
    return strs.raw.filter(function (raw, index) {
        return index % 2 === 1;
    });
}
(_a = ["Hello \n", " \t ", "\n world"], _a.raw = ["Hello \\n", " \\n world"], _a);
```

## AMD 可选依赖名称

`/// <amd-dependency path="x" />` 会告诉编译器需要被注入到模块 `require` 方法中的非 TS 模块依赖; 然而在 TS 代码中无法使用这个模块.

新的 `amd-dependency name` 属性允许为 AMD 依赖传递一个可选的名称.

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

生成的 JS 代码:

```
define(["require", "exports", "legacy/moduleA"], function (require,
    moduleA.callStuff()
});
```

## 通过 `tsconfig.json` 指示一个项目

通过添加 `tsconfig.json` 到一个目录指明这是一个 TypeScript 项目的根目录.

`tsconfig.json` 文件指定了根文件以及编译项目需要的编译器选项. 一个项目可以由以下方式编译:

- 调用 `tsc` 并不指定输入文件, 此时编译器会从当前目录开始往上级目录寻找 `tsconfig.json` 文件.
- 调用 `tsc` 并不指定输入文件, 使用 `-project` (或者 `-p`) 命令行选项指定包含了 `tsconfig.json` 文件的目录.

例子:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "sourceMap": true,
  }
}
```

参见 [tsconfig.json wiki 页面](#) 查看更多信息.

## `--rootDir` 命令行选项

选项 `--outDir` 在输出中会保留输入的层级关系. 编译器将所有输入文件共有的最长路径作为根路径; 并且在输出中应用对应的子层级关系.

有的时候这并不是期望的结果, 比如输入 `FolderA\FolderB\1.ts` 和 `FolderA\FolderB\2.ts`, 输出结构会是 `FolderA\FolderB\` 对应的结构. 如果输入中新增 `FolderA\3.ts` 文件, 输出的结构将突然变为 `FolderA\` 对应的结构.

`--rootDir` 指定了会输出对应结构的输入目录, 不再通过计算获得.

## `--noEmitHelpers` 命令行选项

TypeScript 编译器在需要的时候会输出一些像 `__extends` 这样的工具函数. 这些函数会在使用它们的所有文件中输出. 如果你想要聚合所有的工具函数到同一个位置, 或者覆盖默认的行为, 使用 `--noEmitHelpers` 来告知编译器不要输出它们.

## `--newLine` 命令行选项

默认输出的换行符在 Windows 上是 `\r\n`, 在 \*nix 上是 `\n`. `--newLine` 命令行标记可以覆盖这个行为, 并指定输出文件中使用的换行符.

## `--inlineSourceMap` and `inlineSources` 命令行选项

`--inlineSourceMap` 将内嵌源文件映射到 `.js` 文件, 而不是在单独的 `.js.map` 文件中. `--inlineSources` 允许进一步将 `.ts` 文件内容包含到输出文件中.

# TypeScript 1.4

## 联合类型

### 概述

联合类型有助于表示一个值的类型可以是多种类型之一的情况。比如，有一个API 接命令行传入 `string` 类型，`string[]` 类型或者是一个返回 `string` 的函数。你就可以这样写：

```
interface RunOptions {  
  program: string;  
  cmdline: string[]|string|(() => string);  
}
```

给联合类型赋值也很直观 -- 只要这个值能满足联合类型中任意一个类型那么就可以赋值给这个联合类型：

```
var opts: RunOptions = /* ... */;  
opts.cmdline = '-hello world'; // OK  
opts.cmdline = ['-hello', 'world']; // OK  
opts.cmdline = [42]; // Error, 数字不是字符串或字符串数组
```

当读取联合类型时，你可以访问类型共有的属性：

```
if(opts.length === 0) { // OK, string和string[]都有'length'属性  
  console.log("it's empty");  
}
```

使用类型保护，你可以轻松地使用联合类型：

```
function formatCommandline(c: string|string[]) {
    if(typeof c === 'string') {
        return c.trim();
    } else {
        return c.join(' ');
    }
}
```

## 严格的泛型

随着联合类型可以表示有很多类型的场景，我们决定去改进泛型调用的规范性。之前，这段代码编译不会报错（出乎意料）：

```
function equal<T>(lhs: T, rhs: T): boolean {
    return lhs === rhs;
}

// 之前没有错误
// 现在会报错：在string和number之前没有最佳的基本类型
var e = equal(42, 'hello');
```

通过联合类型，你可以指定你想要的行为，在函数定义时或在调用的时候：

```
// 'choose' function where types must match
function choose1<T>(a: T, b: T): T { return Math.random() > 0.5 ? a : b; }
var a = choose1('hello', 42); // Error
var b = choose1<string|number>('hello', 42); // OK

// 'choose' function where types need not match
function choose2<T, U>(a: T, b: U): T|U { return Math.random() > 0.5 ? a : b; }
var c = choose2('bar', 'foo'); // OK, c: string
var d = choose2('hello', 42); // OK, d: string|number
```

## 更好的类型推断



当一个集合里有多种类型的值时，联合类型会为数组或其它地方提供更好的类型推断：

```
var x = [1, 'hello']; // x: Array<string|number>
x[0] = 'world'; // OK
x[0] = false; // Error, boolean is not string or number
```

## let 声明

在JavaScript里，`var` 声明会被“提升”到所在作用域的顶端。这可能会引发一些让人不解的bugs：

```
console.log(x); // meant to write 'y' here
/* later in the same block */
var x = 'hello';
```

TypeScript已经支持新的ES6的关键字 `let`，声明一个块级作用域的变量。一个 `let` 变量只能在声明之后的位置被引用，并且作用域为声明它的块里：

```
if(foo) {
    console.log(x); // Error, cannot refer to x before its declaration
    let x = 'hello';
} else {
    console.log(x); // Error, x is not declared in this block
}
```

`let` 只在设置目标为ECMAScript 6 ( `--target ES6` ) 时生效。

## const 声明

另一个TypeScript支持的ES6里新出现的声明类型是 `const`。不能给一个 `const` 类型变量赋值，只能在声明的时候初始化。这对于那些在初始化之后就不想去改变它的值的情况下是很有帮助的：

```
const halfPi = Math.PI / 2;
halfPi = 2; // Error, can't assign to a `const`
```

`const` 只在设置目标为ECMAScript 6 ( `--target ES6` ) 时生效。

## 模版字符串

TypeScript现已支持ES6模块字符串。通过它可以方便地在字符串中嵌入任何表达式：

```
var name = "TypeScript";
var greeting = `Hello, ${name}! Your name has ${name.length} characters`
```

当编译目标为ES6之前的版本时，这个字符串被分解为：

```
var name = "TypeScript!";
var greeting = "Hello, " + name + "! Your name has " + name.length + " characters"
```

## 类型守护

JavaScript常用模式之一是在运行时使用 `typeof` 或 `instanceof` 检查表达式的类型。在 `if` 语句里使用它们的时候，TypeScript可以识别出这些条件并且随之改变类型推断的结果。

使用 `typeof` 来检查一个变量：

```
var x: any = /* ... */;
if(typeof x === 'string') {
    console.log(x.substr(1)); // Error, 'substr' does not exist on 'string'
}
// x is still any here
x.unknown(); // OK
```

结合联合类型使用 `typeof` 和 `else` :

```
var x: string|HTMLElement = /* ... */;
if(typeof x === 'string') {
    // x is string here, as shown above
} else {
    // x is HTMLElement here
    console.log(x.innerHTML);
}
```

结合类和联合类型使用 `instanceof` :

```
class Dog { woof() { } }
class Cat { meow() { } }
var pet: Dog|Cat = /* ... */;
if(pet instanceof Dog) {
    pet.woof(); // OK
} else {
    pet.woof(); // Error
}
```

## 类型别名

你现在可以使用 `type` 关键字来为类型定义一个“别名”:

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

类型别名与其原始的类型完全一致；它们只是简单的替代名。

## `const enum` (完全嵌入的枚举)

枚举很有帮助，但是有些程序实际上并不需要它生成的代码并且想要将枚举变量所代码的数字值直接替换到对应位置上。新的 `const enum` 声明与正常的 `enum` 在类型安全方面具有同样的作用，只是在编译时会清除掉。

```
const enum Suit { Clubs, Diamonds, Hearts, Spades }  
var d = Suit.Diamonds;
```

Compiles to exactly:

```
var d = 1;
```

TypeScript也会在可能的情况下计算枚举值：

```
enum MyFlags {  
  None = 0,  
  Neat = 1,  
  Cool = 2,  
  Awesome = 4,  
  Best = Neat | Cool | Awesome  
}  
var b = MyFlags.Best; // emits var b = 7;
```

## **-noEmitOnError** 命令行选项

TypeScript编译器的默认行为是当存在类型错误（比如，将 `string` 类型赋值给 `number` 类型）时仍会生成.js文件。这在构建服务器上或是其它场景里可能会是不想看到的情况，因为希望得到的是一次“纯净”的构建。新的 `noEmitOnError` 标记可以阻止在编译时遇到错误的情况下继续生成.js代码。

它现在是MSBuild工程的默认行为；这允许MSBuild持续构建以我们想要的行为进行，输出永远是来自纯净的构建。

## AMD 模块名

默认情况下AMD模块以匿名形式生成。这在使用其它工具（比如，r.js）处理生成的模块的时可能会带来麻烦。

新的 `amd-module name` 标签允许给编译器传入一个可选的模块名：

```
//// [amdModule.ts]
<amd-module name='NamedModule'>
export class C {
}
```

结果会把 `NamedModule` 赋值成模块名，做为调用AMD `define` 的一部分：

```
//// [amdModule.js]
define("NamedModule", ["require", "exports"], function (require, ex
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

## TypeScript 1.3

### 受保护的

类里面新的 `protected` 修饰符作用与其它语言如C++，C#和Java中的一样。一个类的 `protected` 成员只在这个类的子类中可见：

```
class Thing {
    protected doSomething() { /* ... */ }
}

class MyThing extends Thing {
    public myMethod() {
        // OK，可以在子类里访问受保护的成员
        this.doSomething();
    }
}

var t = new MyThing();
t.doSomething(); // Error，不能在类外部访问受保护成员
```

### 元组类型

元组类型表示一个数组，其中元素的类型都是已知的，但是不一样是同样的类型。比如，你可能想要表示一个第一个元素是 `string` 类型第二个元素是 `number` 类型的数组：

```
// Declare a tuple type
var x: [string, number];
// 初始化
x = ['hello', 10]; // OK
// 错误的初始化
x = [10, 'hello']; // Error
```

但是访问一个已知的索引，会得到正确的类型：

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number'没有'substr'方法
```

注意在TypeScript1.4里，当访问超出已知索引的元素时，会返回联合类型：

```
x[3] = 'world'; // OK
console.log(x[5].toString()); // OK, 'string'和'number'都有toString
x[6] = true; // Error, boolean不是number或string
```

# TypeScript 1.1

## 改进性能

1.1版本的编译器速度比所有之前发布的版本快4倍。阅读[这篇博客里的有关图表](#)

## 更好的模块可见性规则

TypeScript现在只在使用 `--declaration` 标记时才严格强制模块里类型的可见性。这在Angular里很有用，例如：

```
module MyControllers {  
  interface ZooScope extends ng.IScope {  
    animals: Animal[];  
  }  
  export class ZooController {  
    // Used to be an error (cannot expose ZooScope), but now is onl  
    // an error when trying to generate .d.ts files  
    constructor(public $scope: ZooScope) { }  
    /* more code */  
  }  
}
```



## 介绍

为了让程序有价值，我们需要能够处理最简单的数据单元：数字，字符串，结构体，布尔值等。TypeScript支持与JavaScript几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用。

## 布尔值

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做 `boolean`（其它语言中也一样）。

```
let isDone: boolean = false;
```

## 数字

和JavaScript一样，TypeScript里的所有数字都是浮点数。这些浮点数的类型是 `number`。除了支持十进制和十六进制字面量，Typescript还支持ECMAScript 2015中引入的二进制和八进制字面量。

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
let binaryLiteral: number = 0b1010;
let octalLiteral: number = 0o744;
```

## 字符串

JavaScript程序的另一项基本操作是处理网页或服务器端的文本数据。像其它语言里一样，我们使用 `string` 表示文本数据类型。和JavaScript一样，可以使用双引号（`"`）或单引号（`'`）表示字符串。

```
let name: string = "bob";  
name = "smith";
```

你还可以使用模版字符串，它可以定义多行文本和内嵌表达式。这种字符串是被反引号包围（```），并且以 `${ expr }` 这种形式嵌入表达式

```
let name: string = `Gene`;  
let age: number = 37;  
let sentence: string = `Hello, my name is ${ name }.  
  
I'll be ${ age + 1 } years old next month.`;
```

这与下面定义 `sentence` 的方式效果相同：

```
let sentence: string = "Hello, my name is " + name + ".\n\n" +  
    "I'll be " + (age + 1) + " years old next month.";
```

## 数组

TypeScript像JavaScript一样可以操作数组元素。有两种方式可以定义数组。第一种，可以在元素类型后面接上 `[]`，表示由此类型元素组成的一个数组：

```
let list: number[] = [1, 2, 3];
```

第二种方式是使用数组泛型，`Array<元素类型>`：

```
let list: Array<number> = [1, 2, 3];
```

## 元组 Tuple

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。比如，你可以定义一对值分别为 `string` 和 `number` 类型的元组。

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

当访问一个已知索引的元素，会得到正确的类型：

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'subs
```

当访问一个越界的元素，会使用联合类型替代：

```
x[3] = 'world'; // OK, 字符串可以赋值给(string | number)类型

console.log(x[5].toString()); // OK, 'string' 和 'number' 都有 toSt

x[6] = true; // Error, 布尔不是(string | number)类型
```

联合类型是高级主题，我们会在以后的章节里讨论它。

## 枚举

`enum` 类型是对JavaScript标准数据类型的一个补充。像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

默认情况下，从 `0` 开始为元素编号。你也可以手动的指定成员的数值。例如，我们将上面的例子改成从 `1` 开始编号：

```
enum Color {Red = 1, Green, Blue};  
let c: Color = Color.Green;
```

或者，全部都采用手动赋值：

```
enum Color {Red = 1, Green = 2, Blue = 4};  
let c: Color = Color.Green;
```

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

```
enum Color {Red = 1, Green, Blue};  
let colorName: string = Color[2];  
  
alert(colorName);
```

## 任意值

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。那么我们可以使用 `any` 类型来标记这些变量：

```
let notSure: any = 4;  
notSure = "maybe a string instead";  
notSure = false; // okay, definitely a boolean
```

在对现有代码进行改写的时候，`any` 类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。你可能认为 `Object` 有相似的作用，就像它在其它语言中那样。但是 `Object` 类型的变量只是允许你给它赋任意值 -- 但是却不能够在它上面调用任意的方法，即便它真的有这些方法：

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't know)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'
```

当你只知道一部分数据的类型时，`any` 类型也是有用的。比如，你有一个数组，它包含了不同的类型的数据：

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

## 空值

某种程度上来说，`void` 类型像是与 `any` 类型相反，它表示没有任何类型。当一个函数没有返回值时，你通常会见到其返回值类型是 `void`：

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

声明一个 `void` 类型的变量没有什么大用，因为你只能为它赋予 `undefined` 和 `null`：

```
let unusable: void = undefined;
```

## 类型断言

有时候你会遇到这样的情况，你会比TypeScript更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。它没有运行时的影响，只是在编译阶段起作用。TypeScript会假设你，程序员，已经进行了必须的检查。

类型断言有两种形式。其一是“尖括号”语法：

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

另一个为 `as` 语法：

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

两种形式是等价的。至于使用哪个大多数情况下是凭个人喜好；然而，当你在TypeScript里使用JSX时，只有 `as` 语法断言是被允许地。

## 关于 `let`

你可能已经注意到了，我们使用 `let` 关键字来代替大家所熟悉的JavaScript关键字 `var`。 `let` 关键字是JavaScript的一个新概念，TypeScript实现了它。我们会在以后详细介绍它，很多常见的问题都可以通过使用 `let` 来解决，所以尽可能地使用 `let` 来代替 `var` 吧。

## 变量声明

`let` 和 `const` 是JavaScript里相对较新的变量声明方式。像我们之前提到过的，`let` 在很多方面与 `var` 是相似的，但是可以帮助大家避免在JavaScript里常见一些问题。`const` 是对 `let` 的一个增强，它能阻止对一个变量再次赋值。

因为TypeScript是JavaScript的超集，所以它本身就支持 `let` 和 `const`。下面我们会详细说明这些新的声明方式以及为什么推荐使用它们来代替 `var`。

如果你之前使用JavaScript时没有特别在意，那么这节内容会唤起你的回忆。如果你已经对 `var` 声明的怪异之处了如指掌，那么你可以轻松地略过这节。

### `var` 声明

一直以来我们都是通过 `var` 关键字定义JavaScript变量。

```
var a = 10;
```

大家都能理解，这里定义了一个名为 `a` 值为 `10` 的变量。

我们也可以在函数内部定义变量：

```
function f() {  
    var message = "Hello, world!";  
  
    return message;  
}
```

并且我们也可以在其它函数内部访问相同的变量。

```
function f() {  
  var a = 10;  
  return function g() {  
    var b = a + 1;  
    return b;  
  }  
}  
  
var g = f();  
g(); // returns 11;
```

上面的例子里，`g` 可以获取到 `f` 函数里定义的 `a` 变量。每当 `g` 被调用时，它都可以访问到 `f` 里的 `a` 变量。即使当 `g` 在 `f` 已经执行完后才被调用，它仍然可以访问及修改 `a`。

```
function f() {  
  var a = 1;  
  
  a = 2;  
  var b = g();  
  a = 3;  
  
  return b;  
  
  function g() {  
    return a;  
  }  
}  
  
f(); // returns 2
```

## 作用域规则

对于熟悉其它语言的人来说，`var` 声明有些奇怪的作用域规则。看下面的例子：



```
function f(shouldInitialize: boolean) {  
    if (shouldInitialize) {  
        var x = 10;  
    }  
  
    return x;  
}  
  
f(true); // returns '10'  
f(false); // returns 'undefined'
```

有些读者可能要多看几遍这个例子。变量 `x` 是定义在 `if` 语句里面，但是我们却可以在语句的外面访问它。这是因为 `var` 声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问（我们后面会详细介绍），包含它的代码块对此没有什么影响。有些人称此为 `var` 作用域或函数作用域。函数参数也使用函数作用域。

这些作用域规则可能会引发一些错误。其中之一就是，多次声明同一个变量并不会报错：

```
function sumMatrix(matrix: number[][]) {  
    var sum = 0;  
    for (var i = 0; i < matrix.length; i++) {  
        var currentRow = matrix[i];  
        for (var i = 0; i < currentRow.length; i++) {  
            sum += currentRow[i];  
        }  
    }  
  
    return sum;  
}
```

这里很容易看出一些问题，里层的 `for` 循环会覆盖变量 `i`，因为所有 `i` 都引用相同的函数作用域内的变量。有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

## 变量获取怪异之处

快速的猜一下下面的代码会返回什么：

```
for (var i = 0; i < 10; i++) {  
    setTimeout(function() {console.log(i); }, 100 * i);  
}
```

介绍一下，`setTimeout` 会在若干毫秒的延时后执行一个函数（等待其它代码执行完毕）。

好吧，看一下结果：

```
10  
10  
10  
10  
10  
10  
10  
10  
10  
10  
10
```

很多JavaScript程序员对这种行为已经很熟悉了，但如果你很不解，你并不是一个人。大多数人期望输出结果是这样：

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

还记得我们上面讲的变量获取吗？

每当 `g` 被调用时，它都可以访问到 `f` 里的 `a` 变量。

让我们花点时间考虑在这个上下文里的情况。`setTimeout` 在若干毫秒后执行一个函数，并且是在 `for` 循环结束后。`for` 循环结束后，`i` 的值为 `10`。所以当函数被调用的时候，它会打印出 `10`！

一个通常的解决方法是使用立即执行的函数表达式（IIFE）来捕获每次迭代时 `i` 的值：

```
for (var i = 0; i < 10; i++) {  
  // capture the current state of 'i'  
  // by invoking a function with its current value  
  (function(i) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
  })(i);  
}
```

这种奇怪的形式我们已经司空见惯了。参数 `i` 会覆盖 `for` 循环里的 `i`，但是因为我们起了同样的名字，所以我们不用怎么改 `for` 循环体里的代码。

## let 声明

现在你已经知道了 `var` 存在一些问题，这恰好说明了为什么用 `let` 语句来声明变量。除了名字不同外，`let` 与 `var` 的写法一致。

```
let hello = "Hello!";
```

主要的区别不在语法上，而是语义，我们接下来会深入研究。

## 块作用域

当用 `let` 声明一个变量，它使用的是词法作用域或块作用域。不同于使用 `var` 声明的变量那样可以在包含它们的函数外访问，块作用域变量在包含它们的块或 `for` 循环之外是不能访问的。

```
function f(input: boolean) {  
    let a = 100;  
  
    if (input) {  
        // Still okay to reference 'a'  
        let b = a + 1;  
        return b;  
    }  
  
    // Error: 'b' doesn't exist here  
    return b;  
}
```

这里我们定义了2个变量 `a` 和 `b`。 `a` 的作用域是 `f` 函数体内，而 `b` 的作用域是 `if` 语句块里。

在 `catch` 语句里声明的变量也具有同样的作用域规则。

```
try {  
    throw "oh no!";  
}  
catch (e) {  
    console.log("Oh well.");  
}  
  
// Error: 'e' doesn't exist here  
console.log(e);
```

拥有块级作用域的变量的另一个特点是，它们不能在声明之前读或写。虽然这些变量始终“存在”于它们的作用域里，但在直到声明它的代码之前的区域都属于时间死区。它只是用来说明我们不能在 `let` 语句之前访问它们，幸运的是TypeScript可以告诉我们这些信息。

```
a++; // illegal to use 'a' before it's declared;  
let a;
```

注意一点，我们仍然可以在一个拥有块作用域变量被声明前获取它。只是我们不能在变量声明前去调用那个函数。如果生成代码目标为ES2015，现代的运行时会抛出一个错误；然而，现今TypeScript是不会报错的。

```
function foo() {  
    // okay to capture 'a'  
    return a;  
}  
  
// 不能在'a'被声明前调用'foo'  
// 运行时应该抛出错误  
foo();  
  
let a;
```

关于时间死区的更多信息，查看这里[Mozilla Developer Network](#).

## 重定义及屏蔽

我们提过使用 `var` 声明时，它不在乎你声明多少次；你只会得到1个。

```
function f(x) {  
    var x;  
    var x;  
  
    if (true) {  
        var x;  
    }  
}
```

在上面的例子里，所有 `x` 的声明实际上都引用一个相同的 `x`，并且这是完全有效的代码。这经常会成为bug的来源。好的是，`let` 声明就不会这么宽松了。

```
let x = 10;  
let x = 20; // 错误，不能在1个作用域里多次声明`x`
```

并不是要求两个均是块级作用域的声明TypeScript才会给出一个错误的警告。

```
function f(x) {  
    let x = 100; // error: interferes with parameter declaration  
}  
  
function g() {  
    let x = 100;  
    var x = 100; // error: can't have both declarations of 'x'  
}
```

并不是说块级作用域变量不能在函数作用域内声明。而是块级作用域变量需要在不同的块里声明。

```
function f(condition, x) {  
    if (condition) {  
        let x = 100;  
        return x;  
    }  
  
    return x;  
}  
  
f(false, 0); // returns 0  
f(true, 0);  // returns 100
```

在一个嵌套作用域里引入一个新名字的行为称做屏蔽。它是一把双刃剑，它可能会不小心地引入新问题，同时也可能会解决一些错误。例如，假设我们现在用 `let` 重写之前的 `sumMatrix` 函数。

```
function sumMatrix(matrix: number[][]) {  
    let sum = 0;  
    for (let i = 0; i < matrix.length; i++) {  
        var currentRow = matrix[i];  
        for (let i = 0; i < currentRow.length; i++) {  
            sum += currentRow[i];  
        }  
    }  
  
    return sum;  
}
```

这个版本的循环能得到正确的结果，因为内层循环的 `i` 可以屏蔽掉外层循环的 `i`。

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。同时也有些场景适合利用它，你需要好好打算一下。

## 块级作用域变量的获取

在我们最初谈及获取用 `var` 声明的变量时，我们简略地探究了一下在获取到了变量之后它的行为是怎样的。直观地讲，每次进入一个作用域时，它创建了一个变量的环境。就算作用域内代码已经执行完毕，这个环境仍然是存在的。

```
function theCityThatAlwaysSleeps() {  
    let getCity;  
  
    if (true) {  
        let city = "Seattle";  
        getCity = function() {  
            return city;  
        }  
    }  
  
    return getCity();  
}
```

因为我们已经在 `city` 的环境里获取到了 `city`，所以就算 `if` 语句执行结束后我们仍然可以访问它。

回想一下前面 `setTimeout` 的例子，我们最后需要使用立即执行的函数表达式来获取每次 `for` 循环迭代里的状态。实际上，我们做的是为获取到的变量创建了一个新的变量环境。这样做挺痛苦的，但是幸运的是，你不必在 TypeScript 里这样做了。

当 `let` 声明出现在循环体里时拥有完全不同的行为。不仅是在循环里引入了一个新的变量环境，而是针对每次迭代都会创建这样一个新作用域。这就是我们在使用立即执行的函数表达式时做的事，所以在 `setTimeout` 例子里我们仅使用 `let` 声明就可以了。

```
for (let i = 0; i < 10 ; i++) {  
    setTimeout(function() {console.log(i); }, 100 * i);  
}
```

会输出与预料一致的结果：

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

## `const` 声明

`const` 声明是声明变量的另一种方式。

```
const numLivesForCat = 9;
```



它们与 `let` 声明相似，但是就像它的名字所表达的，它们被赋值后不能再改变。换句话说，它们拥有与 `let` 相同的作用域规则，但是不能对它们重新赋值。

这很好理解，它们引用的值是不可变的。

```
const numLivesForCat = 9;
const kitty = {
  name: "Aurora",
  numLives: numLivesForCat,
}

// Error
kitty = {
  name: "Danielle",
  numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

除非你使用特殊的方法去避免，实际上 `const` 变量的内部状态是可修改的。

## let vs. const

现在我们有两种作用域相似的声明方式，我们自然会问到底应该使用哪个。与大多数泛泛的问题一样，答案是：依情况而定。

使用[最小特权原则](#)，所有变量除了你计划去修改的都应该使用 `const`。基本原则就是如果一个变量不需要对它写入，那么其它使用这些代码的人也不能够写入它们，并且要思考为什么会需要对这些变量重新赋值。使用 `const` 也可以让我们更容易的推测数据的流动。

另一方面，用户很喜欢 `let` 的简洁性。这个手册大部分地方都使用了 `let`。

根据你的自己判断，如果合适的话，与团队成员商议一下。

# 解构

Another TypeScript已经可以解析其它 ECMAScript 2015 特性了。完整列表请参见[the article on the Mozilla Developer Network](#)。本章，我们将给出一个简短的概述。

## 解构数组

最简单的解构莫过于数组的解构赋值了：

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

这创建了2个命名变量 `first` 和 `second`。相当于使用了索引，但更为方便：

```
first = input[0];
second = input[1];
```

解构作用于已声明的变量会更好：

```
// swap variables
[first, second] = [second, first];
```

作用于函数参数：

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
f(input);
```

你可以使用 `...name` 语法创建一个剩余变量列表：

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

当然，由于是JavaScript, 你可以忽略你不关心的尾随元素：

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

或其它元素：

```
let [, second, , fourth] = [1, 2, 3, 4];
```

## 对象解构

你也可以解构对象：

```
let o = {
  a: "foo",
  b: 12,
  c: "bar"
}
let {a, b} = o;
```

这通过 `o.a` and `o.b` 创建了 `a` 和 `b`。注意，如果你不需要 `c` 你可以忽略它。

就像数组解构，你可以用没有声明的赋值：

```
({a, b} = {a: "baz", b: 101});
```

注意，我们需要用括号将它括起来，因为Javascript通常会将以 `{` 起始的语句解析为一个块。

## 属性重命名

你也可以给属性以不同的名字：

```
let {a: newName1, b: newName2} = o;
```

这里的语法开始变得混乱。你可以将 `a: newName1` 读做 "`a` 作为 `newName1`"。方向是从左到右，好像你写成了以下样子：

```
let newName1 = o.a;  
let newName2 = o.b;
```

令人困惑的是，这里的冒号不是指示类型的。如果你想指定它的类型，仍然需要在其后写上完整的模式。

```
let {a, b}: {a: string, b: number} = o;
```

## 默认值

默认值可以让你在属性为 `undefined` 时使用缺省值：

```
function keepWholeObject(wholeObject: {a: string, b?: number}) {  
    let {a, b = 1001} = wholeObject;  
}
```

现在，即使 `b` 为 `undefined`，`keepWholeObject` 函数的变量 `wholeObject` 的属性 `a` 和 `b` 都会有值。

## 函数声明

解构也能用于函数声明。看以下简单的情况：

```
type C = {a: string, b?: number}
function f({a, b}: C): void {
    // ...
}
```

但是，通常情况下更多的是指定默认值，解构默认值有些棘手。首先，你需要知道在设置默认值之前设置其类型。

```
function f({a, b} = {a: "", b: 0}): void {
    // ...
}
f(); // ok, default to {a: "", b: 0}
```

其次，你需要知道在解构属性上给予一个默认或可选的属性用来替换主初始化列表。要知道 `C` 的定义有一个 `b` 可选属性：

```
function f({a, b = 0} = {a: ""}): void {
    // ...
}
f({a: "yes"}) // ok, default b = 0
f() // ok, default to {a: ""}, which then defaults b = 0
f({}) // error, 'a' is required if you supply an argument
```

要小心使用解构。从前面的例子可以看出，就算是最简单的解构也会有很多问题。尤其当存在深层嵌套解构的时候，就算这时没有堆叠在一起的重命名，默认值和类型注解，也是令人难以理解的。解构表达式要尽量保持小而简单。你自己也可以直接使用解构将会生成的赋值表达式。

## 介绍

TypeScript的核心原则之一是对值所具有的`shape`进行类型检查。它有时被称做“鸭式辨型法”或“结构性子类型化”。在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

## 接口初探

下面通过一个简单示例来观察接口是如何工作的：

```
function printLabel(labelledObj: { label: string }) {  
    console.log(labelledObj.label);  
}  
  
let myObj = { size: 10, label: "Size 10 Object" };  
printLabel(myObj);
```

类型检查器会查看 `printLabel` 的调用。`printLabel` 有一个参数，并要求这个对象参数有一个名为 `label` 类型为 `string` 的属性。需要注意的是，我们传入的对象参数实际上会包含很多属性，但是编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。然而，有些时候TypeScript却并不会这么宽松，我们下面会稍做讲解。

下面我们重写上面的例子，这次使用接口来描述：必须包含一个 `label` 属性且类型为 `string`：

```
interface LabelledValue {  
    label: string;  
}  
  
function printLabel(labelledObj: LabelledValue) {  
    console.log(labelledObj.label);  
}  
  
let myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

`LabelledValue` 接口就好比一个名字，用来描述上面例子里的要求。它代表了有一个 `label` 属性且类型为 `string` 的对象。需要注意的是，我们在这里并不能像在其它语言里一样，说传给 `printLabel` 的对象实现了这个接口。我们只会去关注值的外形。只要传入的对象满足上面提到的必要条件，那么它就被允许的。

还有一点值得提的是，类型检查器不会去检查属性的顺序，只要相应的属性存在并且类型也是对的就可以。

## 可选属性

接口里的属性不全都是必需的。有些是只在某些条件下存在，或者根本不存在。可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

下面是应用了“option bags”的例子：

```
interface SquareConfig {  
  color?: string;  
  width?: number;  
}  
  
function createSquare(config: SquareConfig): {color: string; area:  
  let newSquare = {color: "white", area: 100};  
  if (config.color) {  
    newSquare.color = config.color;  
  }  
  if (config.width) {  
    newSquare.area = config.width * config.width;  
  }  
  return newSquare;  
}  
  
let mySquare = createSquare({color: "black"});
```

带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个 `?` 符号。

可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。比如，我们故意将 `createSquare` 里的 `color` 属性名拼错，就会得到一个错误提示：



```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    // Error: Property 'collor' does not exist on type 'SquareConfig'
    newSquare.color = config.collor; // Type-checker can catch the error
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

## 额外的属性检查

我们在第一个例子里使用了接口，TypeScript让我们传入 `{ size: number; label: string; }` 到仅期望得到 `{ label: string; }` 的函数里。我们已经学会了可选属性，并且知道他们在“option bags”模式里很有用。

然而，天真地将这两者结合的话就会像在JavaScript里那样搬起石头砸自己的脚。比如，拿 `createSquare` 例子来说：

```
interface SquareConfig {  
    color?: string;  
    width?: number;  
}  
  
function createSquare(config: SquareConfig): { color: string; area: number } {  
    // ...  
}  
  
let mySquare = createSquare({ colour: "red", width: 100 });
```

注意传入 `createSquare` 的参数拼写为 `colour` 而不是 `color`。在JavaScript里，这会默默地失败。

你可能会争辩这个程序已经正确地类型化了，因为 `width` 属性是兼容的，不存在 `color` 属性，而且额外的 `colour` 属性是无意义的。

然而，TypeScript会认为这段代码可能存在bug。对象字面量会被特殊对待而且会经过额外属性检查，当将它们赋值给变量或作为参数传递的时候。如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误。

```
// error: 'colour' not expected in type 'SquareConfig'  
let mySquare = createSquare({ colour: "red", width: 100 });
```

绕开这些检查非常简单。最好而简便的方法是使用类型断言：

```
let mySquare = createSquare({ colour: "red", width: 100 } as SquareConfig);
```

另一个方法，可能会让人有点惊讶，就是将一个对象赋值给另一个变量：

```
let squareOptions = { colour: "red", width: 100 };  
let mySquare = createSquare(squareOptions);
```

因为 `squareOptions` 不会经过额外属性检查，所以编译器不会报错。

要留意，在像上面一样的简单代码里，你可能不应该去绕开这些检查。对于包含方法和内部状态的复杂对象字面量来讲，你可能需要使用这些技巧，但是大部额外属性检查错误是真正的bug。就是说你遇到了额外类型检查出的错误，比如选择包，你应该去审查一下你的类型声明。在这里，如果支持传入 `color` 或 `colour` 属性到 `createSquare`，你应该修改 `SquareConfig` 定义来体现出这一点。

## 函数类型

接口能够描述JavaScript中对象拥有的各种各样的外形。除了描述带有属性的普通对象外，接口也可以描述函数类型。

为了使用接口表示函数类型，我们需要给接口定义一个调用签名。它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

```
interface SearchFunc {  
    (source: string, subString: string): boolean;  
}
```

这样定义后，我们可以像使用其它接口一样使用这个函数类型的接口。下例展示了如何创建一个函数类型的变量，并将一个同类型的函数赋值给这个变量。

```
let mySearch: SearchFunc;  
mySearch = function(source: string, subString: string) {  
    let result = source.search(subString);  
    if (result == -1) {  
        return false;  
    }  
    else {  
        return true;  
    }  
}
```

对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。比如，我们使用下面的代码重写上面的例子：

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    if (result == -1) {
        return false;
    }
    else {
        return true;
    }
}
```

函数的参数会逐个进行检查，要求对应位置上的参数类型是兼容的。如果你不想指定类型，Typescript的类型系统会推断出参数类型，因为函数直接赋值给了 `SearchFunc` 类型变量。函数的返回值类型是通过其返回值推断出来的（此例是 `false` 和 `true`）。如果让这个函数返回数字或字符串，类型检查器会警告我们函数的返回值类型与 `SearchFunc` 接口中的定义不匹配。

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    if (result == -1) {
        return false;
    }
    else {
        return true;
    }
}
```

## 数组类型

与使用接口描述函数类型差不多，我们也可以描述数组类型。数组类型具有一个 `index` 类型表示索引的类型，还有一个相应的返回值类型表示通过索引得到的元素的类型。

```
interface StringArray {  
    [index: number]: string;  
}  
  
let myArray: StringArray;  
myArray = ["Bob", "Fred"];
```

支持两种索引类型：**string**和**number**。数组可以同时使用这两种索引类型，但是有一个限制，数字索引返回值的类型必须是字符串索引返回值的类型的子类型。

索引签名能够很好的描述数组和 **dictionary** 模式，它们也要求所有属性要与返回值类型相匹配。因为字符串索引表明 `obj.property` 和 `obj["property"]` 两种形式都可以。下面的例子里，`name` 的类型与字符串索引类型不匹配，所以类型检查器给出一个错误提示：

```
interface NumberDictionary {  
    [index: string]: number;  
    length: number;    // 可以，length是number类型  
    name: string        // 错误，`name`的类型不是索引类型的子类型  
}
```

## 类类型

### 实现接口

与C#或Java里接口的基本作用一样，TypeScript也能够用它来明确的强制一个类去符合某种契约。

```
interface ClockInterface {  
    currentTime: Date;  
}  
  
class Clock implements ClockInterface {  
    currentTime: Date;  
    constructor(h: number, m: number) { }  
}
```

你也可以在接口中描述一个方法，在类里实现它，如同下面的 `setTime` 方法一样：

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date);  
}  
  
class Clock implements ClockInterface {  
    currentTime: Date;  
    setTime(d: Date) {  
        this.currentTime = d;  
    }  
    constructor(h: number, m: number) { }  
}
```

接口描述了类的公共部分，而不是公共和私有两部分。它不会帮你检查类是否具有某些私有成员。

## 类静态部分与实例部分的区别

当你操作类和接口的时候，你要知道类是具有两个类型的：静态部分的类型和实例的类型。你会注意到，当你用构造器签名去定义一个接口并试图定义一个类去实现这个接口时会得到一个错误：

```
interface ClockConstructor {  
    new (hour: number, minute: number);  
}  
  
class Clock implements ClockConstructor {  
    currentTime: Date;  
    constructor(h: number, m: number) { }  
}
```

这里因为当一个类实现了一个接口时，只对其实例部分进行类型检查。`constructor` 存在于类的静态部分，所以不在检查的范围内。

因此，我们应该直接操作类的静态部分。看下面的例子，我们定义了两个接口，`ClockConstructor` 为构造函数所用和 `ClockInterface` 为实例方法所用。为了方便我们定义一个构造函数 `createClock`，它用传入的类型创建实例。

```
interface ClockConstructor {  
    new (hour: number, minute: number): ClockInterface;  
}  
  
interface ClockInterface {  
    tick();  
}  
  
function createClock(ctor: ClockConstructor, hour: number, minute:  
    return new ctor(hour, minute);  
}  
  
class DigitalClock implements ClockInterface {  
    constructor(h: number, m: number) { }  
    tick() {  
        console.log("beep beep");  
    }  
}  
  
class AnalogClock implements ClockInterface {  
    constructor(h: number, m: number) { }  
    tick() {  
        console.log("tick tock");  
    }  
}  
  
let digital = createClock(DigitalClock, 12, 17);  
let analog = createClock(AnalogClock, 7, 32);
```

因为 `createClock` 的第一个参数是 `ClockConstructor` 类型，在 `createClock(AnalogClock, 7, 32)` 里，会检查 `AnalogClock` 是否符合构造函数签名。

## 扩展接口

和类一样，接口也可以相互扩展。这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。



```
interface Shape {  
    color: string;  
}  
  
interface Square extends Shape {  
    sideLength: number;  
}  
  
let square = <Square>{};  
square.color = "blue";  
square.sideLength = 10;
```

一个接口可以继承多个接口，创建出多个接口的合成接口。

```
interface Shape {  
    color: string;  
}  
  
interface PenStroke {  
    penWidth: number;  
}  
  
interface Square extends Shape, PenStroke {  
    sideLength: number;  
}  
  
let square = <Square>{};  
square.color = "blue";  
square.sideLength = 10;  
square.penWidth = 5.0;
```

## 混合类型

先前我们提过，接口能够描述JavaScript里丰富的类型。因为JavaScript其动态灵活的特点，有时你会希望一个对象可以同时具有上面提到的多种类型。

一个例子就是，一个对象可以同时做为函数和对象使用，并带有额外的属性。

```
interface Counter {
  (start: number): string;
  interval: number;
  reset(): void;
}

function getCounter(): Counter {
  let counter = <Counter>function (start: number) { };
  counter.interval = 123;
  counter.reset = function () { };
  return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

在使用JavaScript第三方库的时候，你可能需要像上面那样去完整地定义类型。

## 接口继承类

当接口继承了一个类类型时，它会继承类的成员但不包括其实现。就好像接口声明了所有类中存在的成员，但并没有提供具体实现一样。接口同样会继承到类的 **private** 和 **protected** 成员。这意味着当你创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）。

这是很有用的，当你有一个很深层次的继承，但是只想你的代码只是针对拥有特定属性的子类起作用的时候。子类除了继承自基类外与基类没有任何联系。例：

```
class Control {  
    private state: any;  
}  
  
interface SelectableControl extends Control {  
    select(): void;  
}  
  
class Button extends Control {  
    select() { }  
}  
class TextBox extends Control {  
    select() { }  
}  
class Image extends Control {  
}  
class Location {  
    select() { }  
}
```

在上面的例子里，`SelectableControl` 包含了 `Control` 的所有成员，包括私有成员 `state`。因为 `state` 是私有成员，所以只能是 `Control` 的子类们才能实现 `SelectableControl` 接口。因为只有 `Control` 的子类才能够拥有一个声明于 `Control` 的私有成员 `state`，这对私有成员的兼容性是必需的。

在 `Control` 类内部，是允许通过 `SelectableControl` 的实例来访问私有成员 `state` 的。实际上，`SelectableControl` 就像 `Control` 一样，并拥有一个 `select` 方法。`Button` 和 `TextBox` 类是 `SelectableControl` 的子类（因为它们都继承自 `Control` 并有 `select` 方法），但 `Image` 和 `Location` 类并不是这样的。

## 介绍

传统的JavaScript程序使用函数和基于原型的继承来创建可重用的组件，但这对于熟悉使用面向对象方式的程序员来说有些棘手，因为他们用的是基于类的继承并且对象是从类构建出来的。从ECMAScript 2015，也就是ECMAScript 6，JavaScript程序将可以使用这种基于类的面向对象方法。在TypeScript里，我们允许开发者现在就使用这些特性，并且编译后的JavaScript可以在所有主流浏览器和平台上运行，而不需要等到下个JavaScript版本。

## 类

下面看一个使用类的例子：

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

如果你使用过C#或Java，你会对这种语法非常熟悉。我们声明一个 `Greeter` 类。这个类有3个成员：一个叫做 `greeting` 的属性，一个构造函数和一个 `greet` 方法。

你会注意到，我们在引用任何一个类成员的时候都用了 `this`。它表示我们访问的是类的成员。

最后一行，我们使用 `new` 构造了 `Greeter` 类的一个实例。它会调用之前定义的构造函数，创建一个 `Greeter` 类型的新对象，并执行构造函数初始化它。

## 继承

在TypeScript里，我们可以使用常用的面向对象模式。当然，基于类的程序设计中最基本的模式是允许使用继承来扩展一个类。

看下面的例子：

```
class Animal {
  name:string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

这个例子展示了TypeScript中继承的一些特征，与其它语言类似。我们使用 `extends` 来创建子类。你可以看到 `Horse` 和 `Snake` 类是基类 `Animal` 的子类，并且可以访问其属性和方法。

包含`constructor`函数的派生类必须调用 `super()`，它会执行基类的构造方法。

这个例子演示了如何在子类里可以重写父类的方法。`Snake` 类和 `Horse` 类都创建了 `move` 方法，重写了从 `Animal` 继承来的 `move` 方法，使得 `move` 方法根据不同的类而具有不同的功能。注意，即使 `tom` 被声明为 `Animal` 类型，因为它的值是 `Horse`，`tom.move(34)` 调用 `Horse` 里的重写方法：

```
Slithering...
Sammy the Python moved 5m.
Gallopig...
Tommy the Palomino moved 34m.
```

## 公共，私有与受保护的修饰符

### 默认为公有

在上面的例子里，我们可以自由的访问程序里定义的成员。如果你对其它语言中的类比较了解，就会注意到我们在之前的代码里并没有使用 `public` 来做修饰；例如，**C#**要求必须明确地使用 `public` 指定成员是可见的。在TypeScript里，每个成员默认为 `public` 的。

你也可以明确的将一个成员标记成 `public`。我们可以用下面的方式来重写上面的 `Animal` 类：

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  public move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

## 理解 `private`

当成员被标记成 `private` 时，它就不能在声明它的类的外部访问。比如：

```
class Animal {  
    private name: string;  
    constructor(theName: string) { this.name = theName; }  
}  
  
new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript使用的是结构性类型系统。当我们比较两种不同的类型时，并不在乎它们从哪儿来的，如果所有成员的类型都是兼容的，我们就认为它们的类型是兼容的。

然而，当我们比较带有 `private` 或 `protected` 成员的类型的时候，情况就不同了。如果其中一个类型里包含一个 `private` 成员，那么只有当另外一个类型中也存在这样一个 `private` 成员，并且它们是来自同一处声明时，我们才认为这两个类型是兼容的。对于 `protected` 成员也使用这个规则。

下面来看一个例子，详细的解释了这点：

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: Animal and Employee are not compatible
```

这个例子中有 `Animal` 和 `Rhino` 两个类，`Rhino` 是 `Animal` 类的子类。还有一个 `Employee` 类，其类型看上去与 `Animal` 是相同的。我们创建了几个这些类的实例，并相互赋值来看看会发生什么。因为 `Animal` 和 `Rhino` 共享了来自 `Animal` 里的私有成员定义 `private name: string`，因此它们是兼容的。然而 `Employee` 却不是这样。当把 `Employee` 赋值给 `Animal` 的时候，得到一个错误，说它们的类型不兼容。尽管 `Employee` 里也有一个私有成员 `name`，但它明显不是 `Animal` 里面定义的那个。

## 理解 `protected`

`protected` 修饰符与 `private` 修饰符的行为很相似，但有一点不同，`protected` 成员在派生类中仍然可以访问。例如：



```
class Person {
  protected name: string;
  constructor(name: string) { this.name = name; }
}

class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}`;
  }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

注意，我们不能在 `Person` 类外使用 `name`，但是我们仍然可以通过 `Employee` 类的实例方法访问，因为 `Employee` 是由 `Person` 派生出来的。

## 参数属性

在上面的例子中，我们不得不定义一个受保护的成员 `name` 和一个构造函数参数 `theName` 在 `Person` 类里，并且立刻给 `name` 和 `theName` 赋值。这种情况经常会遇到。参数属性可以方便地让我们在一个地方定义并初始化一个成员。下面的例子是对之前 `Animal` 类的修改版，使用了参数属性：

```
class Animal {  
    constructor(private name: string) { }  
    move(distanceInMeters: number) {  
        console.log(`${this.name} moved ${distanceInMeters}m.`);  
    }  
}
```

注意看我们是如何舍弃了 `theName`，仅在构造函数里使用 `private name: string` 参数来创建和初始化 `name` 成员。我们把声明和赋值合并至一处。

参数属性通过给构造函数参数添加一个访问限定符来声明。使用 `private` 限定一个参数属性会声明并初始化一个私有成员；对于 `public` 和 `protected` 来说也是一样。

## 存取器

TypeScript 支持 `getters/setters` 来截取对对象成员的访问。它能帮助你有效的控制对对象成员的访问。

下面来看如何把一类改写成使用 `get` 和 `set`。首先是一个没用使用存取器的例子。

```
class Employee {  
    fullName: string;  
}  
  
let employee = new Employee();  
employee.fullName = "Bob Smith";  
if (employee.fullName) {  
    console.log(employee.fullName);  
}
```

我们可以随意的设置 `fullName`，这是非常方便的，但是这也可能会带来麻烦。

下面这个版本里，我们先检查用户密码是否正确，然后再允许其修改 `employee`。我们把对 `fullName` 的直接访问改成了可以检查密码的 `set` 方法。我们也加了一个 `get` 方法，让上面的例子仍然可以工作。

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

我们可以修改一下密码，来验证一下存取器是否是工作的。当密码不对时，会提示我们没有权限去修改`employee`。

注意：若要使用存取器，要求设置编译器输出目标为ECMAScript 5或更高。

## 静态属性

到目前为止，我们只讨论了类的实例成员，那些仅当类被实例化的时候才会被初始化的属性。我们也可以创建类的静态成员，这些属性存在于类本身上面而不是类的实例上。在这个例子里，我们使用 `static` 定义 `origin`，因为它是所有网格都

会用到的属性。每个实例想要访问这个属性的时候，都要在`origin`前面加上类名。如同在实例属性上使用 `this.` 前缀来访问属性一样，这里我们使用 `Grid.` 来访问静态属性。

```
class Grid {
  static origin = {x: 0, y: 0};
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {
    let xDist = (point.x - Grid.origin.x);
    let yDist = (point.y - Grid.origin.y);
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

## 抽象类

抽象类是供其它类继承的基类。他们一般不会直接被实例化。不同于接口，抽象类可以包含成员的实现细节。`abstract` 关键字是用于定义抽象类和在抽象类内部定义抽象方法。

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log('roaming the earch...');
  }
}
```

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。抽象方法的语法与接口方法相似。两者都是定义方法签名不包含方法体。然而，抽象方法必须使用 `abstract` 关键字并且可以包含访问符。

```
abstract class Department {  
  
    constructor(public name: string) {  
    }  
  
    printName(): void {  
        console.log('Department name: ' + this.name);  
    }  
  
    abstract printMeeting(): void; // 必须在派生类中实现  
}  
  
class AccountingDepartment extends Department {  
  
    constructor() {  
        super('Accounting and Auditing'); // constructors in derived  
    }  
  
    printMeeting(): void {  
        console.log('The Accounting Department meets each Monday at  
    }  
  
    generateReports(): void {  
        console.log('Generating accounting reports...');  
    }  
}  
  
let department: Department; // ok to create a reference to an abstract  
department = new Department(); // error: cannot create an instance of  
department = new AccountingDepartment(); // ok to create and assign  
department.printName();  
department.printMeeting();  
department.generateReports(); // error: method doesn't exist on department
```

## 高级技巧

## 构造函数

当你在TypeScript里定义类的时候，实际上同时定义了很多东西。首先是类的实例的类型。

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

在这里，我们写了 `let greeter: Greeter`，意思是 `Greeter` 类实例的类型是 `Greeter`。这对于用过其它面向对象语言的程序员来讲已经是老习惯了。

我们也创建了一个叫做构造函数的值。这个函数会在我们使用 `new` 创建类实例的时候被调用。下面我们来看看，上面的代码被编译成JavaScript后是什么样子的：

```
let Greeter = (function () {
  function Greeter(message) {
    this.greeting = message;
  }
  Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
  };
  return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

上面的代码里，`let Greeter` 将被赋值为构造函数。当我们使用 `new` 并执行这个函数后，便会得到一个类的实例。这个构造函数也包含了类的所有静态属性。换个角度说，我们可以认为类具有实例部分与静态部分这两个部分。

让我们来改写一下这个例子，看看它们之前的区别：

```
class Greeter {
  static standardGreeting = "Hello, there";
  greeting: string;
  greet() {
    if (this.greeting) {
      return "Hello, " + this.greeting;
    }
    else {
      return Greeter.standardGreeting;
    }
  }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";
let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());
```

这个例子里，`greeter1` 与之前看到的一样。我们实例化 `Greeter` 类，并使用这个对象。与我们之前看到的一样。

再之后，我们直接使用类。我们创建了一个叫做 `greeterMaker` 的变量。这个变量保存了这个类或者说保存了类构造函数。然后我们使用 `typeof Greeter`，意思是取 `Greeter` 类的类型，而不是实例的类型。或者更确切的说，"告诉我 `Greeter` 标识符的类型"，也就是构造函数的类型。这个类型包含了类的所有静态成员和构造函数。之后，就和前面一样，我们在 `greeterMaker` 上使用 `new`，创建 `Greeter` 的实例。

## 把类当做接口使用

如上一节里所讲的，类定义会创建两个东西：类实例的类型和一个构造函数。因为类可以创建出类型，所以你能够在可以使用接口的地方使用类。

```
class Point {  
    x: number;  
    y: number;  
}  
  
interface Point3d extends Point {  
    z: number;  
}  
  
let point3d: Point3d = {x: 1, y: 2, z: 3};
```



## 介绍

函数是JavaScript应用程序的基础。它帮助你实现抽象层，模拟类，信息隐藏和模块。在TypeScript里，虽然已经支持类，命名空间和模块，但函数仍然是主要的定义行为的地方。TypeScript为JavaScript函数添加了额外的功能，让我们可以更容易地使用。

## 函数

和JavaScript一样，TypeScript函数可以创建有名字的函数和匿名函数。你可以随意选择适合应用程序的方式，不论是定义一系列API函数还是只使用一次的函数。

通过下面的例子可以迅速回想起这两种JavaScript中的函数：

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

在JavaScript里，函数可以使用函数体外部的变量。当函数这么做时，我们说它‘捕获’了这些变量。至于为什么可以这样做以及其中的利弊超出了本文的范围，但是深刻理解这个机制对学习JavaScript和TypeScript会很有帮助。

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

## 函数类型

## 为函数定义类型

让我们为上面那个函数添加类型：

```
function add(x: number, y: number): number {  
    return x + y;  
}  
  
let myAdd = function(x: number, y: number): number { return x+y; };
```

我们可以给每个参数添加类型之后再为函数本身添加返回值类型。TypeScript能够根据返回语句自动推断出返回值类型，因此我们通常省略它。

## 书写完整函数类型

现在我们已经为函数指定了类型，下面让我们写出函数的完整类型。

```
let myAdd: (x:number, y:number)=>number =  
    function(x: number, y: number): number { return x+y; };
```

函数类型包含两部分：参数类型和返回值类型。当写出完整函数类型的时候，这两部分都是需要的。我们以参数列表的形式写出参数类型，为每个参数指定一个名字和类型。这个名字只是为了增加可读性。我们也可以这么写：

```
let myAdd: (baseValue:number, increment:number) => number =  
    function(x: number, y: number): number { return x + y; };
```

只要参数类型是匹配的，那么就认为它是有效的函数类型，而不在乎参数名是否正确。

第二部分是返回值类型。对于返回值，我们在函数和返回值类型之前使用( => )符号，使之清晰明了。如之前提到的，返回值类型是函数类型的必要部分，如果函数没有返回任何值，你也必须指定返回值类型为 `void` 而不能留空。

函数的类型只是由参数类型和返回值组成的。函数中使用的捕获变量不会体现在类型里。实际上，这些变量是函数的隐藏状态并不是组成API的一部分。

## 推断类型

尝试这个例子的时候，你会发现如果你在赋值语句的一边指定了类型但是另一边没有类型的话，TypeScript编译器会自动识别出类型：

```
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return x + y;

// The parameters `x` and `y` have the type number
let myAdd: (baseValue:number, increment:number) => number =
    function(x, y) { return x + y; };
```

这叫做“按上下文归类”，是类型推论的一种。它帮助我们更好地为程序指定类型。

## 可选参数和默认参数

TypeScript里的每个函数参数都是必须的。这不是指不能传递 `null` 或 `undefined` 作为参数，而是说编译器检查用户是否为每个参数都传入了值。编译器还会假设只有这些参数会被传递进函数。简短地说，传递给一个函数的参数个数必须与函数期望的参数个数一致。

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // error, too few
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many
let result3 = buildName("Bob", "Adams"); // ah, just right
```

JavaScript里，每个参数都是可选的，可传可不传。没传参的时候，它的值就是 `undefined`。在TypeScript里我们可以在参数名旁使用 `?` 实现可选参数的功能。比如，我们想让last name是可选的：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob"); // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many
let result3 = buildName("Bob", "Adams"); // ah, just right
```

可选参数必须跟在必须参数后面。如果上例我们想让first name是可选的，那么就必须调整它们的位置，把first name放在后面。

在TypeScript里，我们也可以为参数提供一个默认值当用户没有传递这个参数或传递的值是 `undefined` 时。它们叫做有默认初始化值的参数。让我们修改上例，把last name的默认值设置为 `"Smith"`。

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // works correctly
let result2 = buildName("Bob", undefined); // still works, as undefined
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many
let result4 = buildName("Bob", "Adams"); // ah, just right
```

在所有必须参数后面的带默认初始化的参数都是可选的，与可选参数一样，在调用函数的时候可以省略。也就是说可选参数与末尾的默认参数共享参数类型。

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

和

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

共享同样的类型 `(firstName: string, lastName?: string) => string`。默认参数的默认值消失了，只保留了它是一个可选参数的信息。

与普通可选参数不同的是，带默认值的参数不需要放在必须参数的后面。如果带默认值的参数出现在必须参数前面，用户必须明确的传入 `undefined` 值来获得默认值。例如，我们重写最后一个例子，让 `firstName` 是带默认值的参数：

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // error, too few
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many
let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```

## 剩余参数

必要参数，默认参数和可选参数有个共同点：它们表示某一个参数。有时，你想同时操作多个参数，或者你并不知道会有多少参数传递进来。在JavaScript里，你可以使用 `arguments` 来访问所有传入的参数。

在TypeScript里，你可以把所有参数收集到一个变量里：

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

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

剩余参数会被当做个数不限的可选参数。可以一个都没有，同样也可以有任意个。编译器创建参数数组，名字是你在省略号 ( `...` ) 后面给定的名字，你可以在函数体内使用这个数组。

这个省略号也会在带有剩余参数的函数类型定义上使用到：

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}  
  
let buildNameFun: (fname: string, ...rest: string[]) => string = bu
```

## Lambda表达式和使用 `this`

JavaScript里 `this` 的工作机制对JavaScript程序员来说已经是老生常谈了。的确，学会如何使用它绝对是JavaScript编程中的一件大事。由于TypeScript是JavaScript的超集，TypeScript程序员也需要弄清 `this` 工作机制并且当有bug的时候能够找出错误所在。`this` 的工作机制可以单独写一本书了，并确已有人这么做了。在这里，我们只介绍一些基础知识。

JavaScript里，`this` 的值在函数被调用的时候才会指定。这是个既强大又灵活的特点，但是你需要花点时间弄清楚函数调用的上下文是什么。众所周知这不是一件很简单的事，特别是函数当做回调函数使用的时候。

下面看一个例子：

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

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

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

如果我们运行这个程序，会发现它并没有弹出对话框而是报错了。因为 `createCardPicker` 返回的函数里的 `this` 被设置成了 `window` 而不是 `deck` 对象。当你调用 `cardPicker()` 时会发生这种情况。这里没有对 `this` 进行动态绑定因此为 `window`。（注意在严格模式下，会是 `undefined` 而不是 `window`）。

为了解决这个问题，我们可以在函数被返回时就绑好正确的 `this`。这样的话，无论之后怎么使用它，都会引用绑定的‘`deck`’对象。

我们把函数表达式变为使用 `lambda` 表达式（`() => {}`）。这样就会在函数创建的时候就指定了‘`this`’值，而不是在函数调用的时候。

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    // Notice: the line below is now a lambda, allowing us to c
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

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

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

为了解更多关于 `this` 的信息，请阅读Yahuda Katz的[Understanding JavaScript Function Invocation and "this"](#)。

## 重载

JavaScript本身是个动态语言。JavaScript里函数根据传入不同的参数而返回不同类型的数是很常见的。



```
let suits = ["hearts", "spades", "clubs", "diamonds"];

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") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 5 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

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

`pickCard` 方法根据传入参数的不同会返回两种不同的类型。如果传入的是代表纸牌的对象，函数作用是从中抓一张牌。如果用户想抓牌，我们告诉他抓到了什么牌。但是这怎么在类型系统里表示呢。

方法是为同一个函数提供多个函数类型定义来进行函数重载。编译器会根据这个列表去处理函数的调用。下面我们来重载 `pickCard` 函数。

```

let suits = ["hearts", "spades", "clubs", "diamonds"];

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") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 5 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

```

这样改变后，重载的 `pickCard` 函数在调用的时候会进行正确的类型检查。

为了让编译器能够选择正确的检查类型，它与JavaScript里的处理流程相似。它查找重载列表，尝试使用第一个重载定义。如果匹配的话就使用这个。因此，在定义重载的时候，一定要把最精确的定义放在最前面。

注意，`function pickCard(x): any` 并不是重载列表的一部分，因此这里只有两个重载：一个是接收对象另一个接收数字。以其它参数调用 `pickCard` 会产生错误。

## 介绍

软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像C#和Java这样的语言中，可以使用 泛型 来创建可重用的组件，一个组件可以支持多种类型的数据。这样用户就可以以自己的数据类型来使用组件。

## 泛型之Hello World

下面来创建第一个使用泛型的例子：identity函数。这个函数会返回任何传入它的值。你可以把这个函数当成是 echo 命令。

不用泛型的话，这个函数可能是下面这样：

```
function identity(arg: number): number {  
    return arg;  
}
```

或者，我们使用 any 类型来定义函数：

```
function identity(arg: any): any {  
    return arg;  
}
```

虽然使用 any 类型后这个函数已经能接收任何类型的arg参数，但是却丢失了一些信息：传入的类型与返回的类型应该是相同的。如果我们传入一个数字，我们只知道任何类型的值都有可能被返回。

因此，我们需要一种方法使用返回值的类型与传入参数的类型是相同的。这里，我们使用了类型变量，它是一种特殊的变量，只用于表示类型而不是值。

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

我们给`identity`添加了类型变量 `T`。`T` 帮助我们捕获用户传入的类型（比如：`number`），之后我们就可以使用这个类型。之后我们再次使用了 `T` 当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的 `identity` 函数叫做泛型，因为它可以适用于多个类型。不同于使用 `any`，它不会丢失信息，像第一个例子那样保持准确性，传入数值类型并返回数值类型。

我们定义了泛型函数后，可以用两种方法使用。第一种是，传入所有的参数，包含类型参数：

```
let output = identity<string>("myString"); // type of output will
```

这里我们明确的指定了 `T` 是字符串类型，并做为一个参数传给函数，使用了 `<>` 括起来而不是 `()`。

第二种方法更普遍。利用了类型推论，编译器会根据传入的参数自动地帮助我们确定 `T` 的类型：

```
let output = identity("myString"); // type of output will be 'str'
```

注意我们并没用 `<>` 明确的指定类型，编译器看到了 `myString`，把 `T` 设置为此类型。类型推论帮助我们保持代码精简和高可读性。如果编译器不能够自动地推断出类型的话，只能像上面那样明确的传入 `T` 的类型，在一些复杂的情况下，这是可能出现的。

## 使用泛型变量

使用泛型创建像 `identity` 这样的泛型函数时，编译器要求你在函数体必须正确的使用这个通用的类型。换句话说，你必须把这些参数当做是任意或所有类型。

看下之前 `identity` 例子：

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

如果我们想同时打印出 `arg` 的长度。我们很可能会这样做：

```
function loggingIdentity<T>(arg: T): T {  
    console.log(arg.length); // Error: T doesn't have .length  
    return arg;  
}
```

如果这么做，编译器会报错说我们使用了 `arg` 的 `.length` 属性，但是没有地方指明 `arg` 具有这个属性。记住，这些类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有 `.length` 属性的。

现在假设我们想操作 `T` 类型的数组而不直接是 `T`。由于我们操作的是数组，所以 `.length` 属性是应该存在的。我们可以像创建其它数组一样创建这个数组：

```
function loggingIdentity<T>(arg: T[]): T[] {  
    console.log(arg.length); // Array has a .length, so no more error  
    return arg;  
}
```

你可以这样理解 `loggingIdentity` 的类型：泛型函数 `loggingIdentity`，接收类型参数 `T`，和函数 `arg`，它是个元素类型是 `T` 的数组，并返回元素类型是 `T` 的数组。如果我们传入数字数组，将返回一个数字数组，因为此时 `T` 的类型为 `number`。这可以让我们把泛型变量 `T` 当做类型的一部分使用，而不是整个类型，增加了灵活性。

我们也可以这样实现上面的例子：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {  
    console.log(arg.length); // Array has a .length, so no more error  
    return arg;  
}
```

使用过其它语言的话，你可能对这种语法已经很熟悉了。在下一节，会介绍如何创建自定义泛型像 `Array<T>` 一样。

## 泛型类型

上一节，我们创建了`identity`通用函数，可以适用于不同的类型。在这节，我们研究一下函数本身的类型，以及如何创建泛型接口。

泛型函数的类型与非泛型函数的类型没什么不同，只是有一个类型参数在最前面，像函数声明一样：

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: <T>(arg: T) => T = identity;
```

我们也可以使用不同的泛型参数名，只要在数量上和使用方式上能对应上就可以。

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: <U>(arg: U) => U = identity;
```

我们还可以使用带有调用签名的对象字面量来定义泛型函数：

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: {<T>(arg: T): T} = identity;
```

这引导我们去写第一个泛型接口了。我们把上面例子中的对象字面量拿出来作为一个接口：

```
interface GenericIdentityFn {  
    <T>(arg: T): T;  
}  
  
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: GenericIdentityFn = identity;
```

一个相似的例子，我们可能想把泛型参数当作整个接口的一个参数。这样我们就能清楚的知道使用的具体是哪个泛型类型（比如：`Dictionary<string>`而不只是`Dictionary`）。这样接口里的其它成员也能知道这个参数的类型了。

```
interface GenericIdentityFn<T> {  
    (arg: T): T;  
}  
  
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: GenericIdentityFn<number> = identity;
```

注意，我们的示例做了少许改动。不再描述泛型函数，而是把非泛型函数签名作为泛型类型一部分。当我们使用`GenericIdentityFn`的时候，还得传入一个类型参数来指定泛型类型（这里是：`number`），锁定了之后代码里使用的类型。对

于描述哪部分类型属于泛型部分来说，理解何时把参数放在调用签名里和何时放在接口上是很有帮助的。

除了泛型接口，我们还可以创建泛型类。注意，无法创建泛型枚举和泛型命名空间。

## 泛型类

泛型类看上去与泛型接口差不多。泛型类使用 ( `<>` ) 括起泛型类型，跟在类名后面。

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

`GenericNumber` 类的使用是十分直观的，并且你可能已经注意到了，没有什么去限制它只能使用 `number` 类型。也可以使用字符串或其它更复杂的类型。

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

与接口一样，直接把泛型类型放在类后面，可以帮助我们确认类的所有属性都在使用相同的类型。

我们在[类](#)那节说过，类有两部分：静态部分和实例部分。泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

## 泛型约束



你应该会记得之前的一个例子，我们有时候想操作某类型的一组值，并且我们知道这组值具有什么样的属性。在 `loggingIdentity` 例子中，我们想访问 `arg` 的 `length` 属性，但是编译器并不能证明每种类型都有 `length` 属性，所以就报错了。

```
function loggingIdentity<T>(arg: T): T {  
    console.log(arg.length); // Error: T doesn't have .length  
    return arg;  
}
```

相比于操作 `any` 所有类型，我们想要限制函数去处理任意带有 `.length` 属性的所有类型。只要传入的类型有这个属性，我们就允许，就是说至少包含这一属性。为此，我们需要列出对于 `T` 的约束要求。

为此，我们定义一个接口来描述约束条件。创建一个包含 `.length` 属性的接口，使用这个接口和 `extends` 关键字还实现约束：

```
interface Lengthwise {  
    length: number;  
}  
  
function loggingIdentity<T extends Lengthwise>(arg: T): T {  
    console.log(arg.length); // Now we know it has a .length property  
    return arg;  
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3); // Error, number doesn't have a .length property
```

我们需要传入符合约束类型的值，必须包含必须的属性：

```
loggingIdentity({length: 10, value: 3});
```

## 在泛型约束中使用类型参数

你可以声明一个类型参数去约束另一个类型参数。比如，我们想要把一个对象的属性复制到另一个对象。我们要确保没有从 `source` 中写入额外的属性，为此，我们在这两个类型间设置了约束：

```
function copyFields<T extends U, U>(target: T, source: U): T {
  for (let id in source) {
    target[id] = source[id];
  }
  return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };

copyFields(x, { b: 10, d: 20 }); // okay
copyFields(x, { Q: 90 }); // error: property 'Q' isn't declared in
```

## 在泛型里使用类类型

在TypeScript使用泛型创建工厂函数时，需要引用构造函数的类类型。比如，

```
function create<T>(c: {new(): T; }): T {
  return new c();
}
```

一个更高级的例子，使用原型属性推断并约束构造函数与类实例的关系。

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function findKeeper<A extends Animal, K> (a: {new(): A;
    prototype: {keeper: K}}): K {

    return a.prototype.keeper;
}

findKeeper(Lion).nametag; // typechecks!
```

## 枚举

使用枚举我们可以定义一些有名字的数字常量。枚举通过 `enum` 关键字来定义。

```
enum Direction {  
    Up = 1,  
    Down,  
    Left,  
    Right  
}
```

一个枚举类型可以包含零个或多个枚举成员。枚举成员具有一个数字值，它可以是常数或是计算得出的值。当满足如下条件时，枚举成员被当作是常数：

- 不具有初始化函数并且之前的枚举成员是常数。在这种情况下，当前枚举成员的值为上一个枚举成员的值加1。但第一个枚举元素是个例外。如果它没有初始化方法，那么它的初始值为 `0`。
- 枚举成员使用常数枚举表达式初始化。常数枚举表达式是TypeScript表达式的子集，它可以在编译阶段求值。当一个表达式满足下面条件之一时，它就是一个常数枚举表达式：
  - 数字字面量
  - 引用之前定义的常数枚举成员（可以是在不同的枚举类型中定义的）如果这个成员是在同一个枚举类型中定义的，可以使用非限定名来引用。
  - 带括号的常数枚举表达式
  - `+`，`-`，`~` 一元运算符应用于常数枚举表达式
  - `+`，`-`，`*`，`/`，`%`，`<<`，`>>`，`>>>`，`&`，`|`，`^` 二元运算符，常数枚举表达式做为其一个操作对象。若常数枚举表达式求值后为 `NaN` 或 `Infinity`，则会在编译阶段报错。

所有其它情况的枚举成员被当作是需要计算得出的值。

```
enum FileAccess {  
    // constant members  
    None,  
    Read    = 1 << 1,  
    Write    = 1 << 2,  
    ReadWrite = Read | Write  
    // computed member  
    G = "123".length  
}
```

枚举是在运行时真正存在的一个对象。其中一个原因是因为这样就可以从枚举值到枚举名进行反向映射。

```
enum Enum {  
    A  
}  
let a = Enum.A;  
let nameOfA = Enum[Enum.A]; // "A"
```

编译成：

```
var Enum;  
(function (Enum) {  
    Enum[Enum["A"] = 0] = "A";  
})(Enum || (Enum = {}));  
var a = Enum.A;  
var nameOfA = Enum[Enum.A]; // "A"
```

生成的代码中，枚举类型被编译成一个对象，它包含双向映射（`name -> value`）和（`value -> name`）。引用枚举成员总会生成一次属性访问并且永远不会内联。在大多数情况下这是很好的并且正确的解决方案。然而有时候需求却比较严格。当访问枚举值时，为了避免生成多余的代码和间接引用，可以使用常数枚举。常数枚举是在 `enum` 关键字前使用 `const` 修饰符。

```
const enum Enum {  
    A = 1,  
    B = A * 2  
}
```

常数枚举只能使用常数枚举表达式并且不同于常规的枚举的是它们在编译阶段会被删除。常数枚举成员在使用的地方被内联进来。这是因为常数枚举不可能有计算成员。

```
const enum Directions {  
    Up,  
    Down,  
    Left,  
    Right  
}  
  
let directions = [Directions.Up, Directions.Down, Directions.Left,
```

生成后的代码为：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */]
```

## 外部枚举

外部枚举用来描述已经存在的枚举类型的形状。

```
declare enum Enum {  
    A = 1,  
    B,  
    C = 2  
}
```

外部枚举和非外部枚举之间有一个重要的区别，在正常的枚举里，没有初始化方法的成员被当成常数成员。对于非常数的外部枚举而言，没有初始化方法时被当做需要经过计算的。

## 介绍

这节介绍TypeScript里的类型推论。即，类型是在哪里如何被推断的。

## 基础

TypeScript里，在有些没有明确指出类型的地方，类型推论会帮助提供类型。如下面的例子

```
let x = 3;
```

变量 `x` 的类型被推断为数字。这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时。

大多数情况下，类型推论是直截了当地。后面的小节，我们会浏览类型推论时的细微差别。

## 最佳通用类型

当需要从几个表达式中推断类型时候，会使用这些表达式的类型来推断出一个最合适的通用类型。例如，

```
let x = [0, 1, null];
```

为了推断 `x` 的类型，我们必须考虑所有元素的类型。这里有两种选择：`number` 和 `null`。计算通用类型算法会考虑所有的候选类型，并给出一个兼容所有候选类型的类型。

由于最终的通用类型取自候选类型，有些时候候选类型共享相同的通用类型，但是却没有一个类型能做为所有候选类型的类型。例如：

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```



这里，我们想让`zoo`被推断为 `Animal[]` 类型，但是这个数组里没有对象是 `Animal` 类型的，因此不能推断出这个结果。为了更正，当候选类型不能使用的时候我们需要明确的指出类型：

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

如果没有找到最佳通用类型的话，类型推论的结果是空对象类型，`{}`。因为这个类型没有任何成员，所以访问其成员的时候会报错。

## 上下文类型

TypeScript类型推论也可能按照相反的方向进行。这被叫做“按上下文归类”。按上下文归类会发生在表达式的类型与所处的位置相关时。比如：

```
window.onmousedown = function(mouseEvent) {  
    console.log(mouseEvent.buton); //<- Error  
};
```

这个例子会得到一个类型错误，TypeScript类型检查器使用 `Window.onmousedown` 函数的类型来推断右边函数表达式的类型。因此，就能推断出 `mouseEvent` 参数的类型了。如果函数表达式不是在上下文类型的位置，`mouseEvent` 参数的类型需要指定为 `any`，这样也不会报错了。

如果上下文类型表达式包含了明确的类型信息，上下文的类型被忽略。重写上面的例子：

```
window.onmousedown = function(mouseEvent: any) {  
    console.log(mouseEvent.buton); //<- Now, no error is given  
};
```

这个函数表达式有明确的参数类型注解，上下文类型被忽略。这样的话就不报错了，因为这里不会使用到上下文类型。

上下文归类会在很多情况下使用到。通常包含函数的参数，赋值表达式的右边，类型断言，对象成员和数组字面量和返回值语句。上下文类型也会做为最佳通用类型的候选类型。比如：

```
function createZoo(): Animal[] {  
    return [new Rhino(), new Elephant(), new Snake()];  
}
```

这个例子里，最佳通用类型有4个候选

者：`Animal`，`Rhino`，`Elephant` 和 `Snake`。当然，`Animal` 会被做为最佳通用类型。

## 介绍

TypeScript里的类型兼容性是基于结构子类型的。结构类型是一种只使用其成员来描述类型的方式。它正好与名义(nominal)类型形成对比。(译者注：在基于名义类型的类型系统中，数据类型的兼容性或等价性是通过明确的声明和/或类型的名称来决定的。这与结构性类型系统不同，它是基于类型的组成结构，且不要求明确地声明。) 看下面的例子：

```
interface Named {  
    name: string;  
}  
  
class Person {  
    name: string;  
}  
  
let p: Named;  
// OK, because of structural typing  
p = new Person();
```

在使用基于名义类型的语言，比如C#或Java中，这段代码会报错，因为Person类没有明确说明其实现了Named接口。

TypeScript的结构性子类型是根据JavaScript代码的典型写法来设计的。因为JavaScript里广泛地使用匿名对象，例如函数表达式和对象字面量，所以使用结构类型系统来描述这些类型比使用名义类型系统更好。

## 关于可靠性的注意事项

TypeScript的类型系统允许某些在编译阶段无法确认其安全性的操作。当一个类型系统具此属性时，被当做是“不可靠”的。TypeScript允许这种不可靠行为的发生是经过仔细考虑的。通过这篇文章，我们会解释什么时候会发生这种情况和其有利的一面。

## 开始

TypeScript结构化类型系统的基本规则是，如果 `x` 要兼容 `y`，那么 `y` 至少具有与 `x` 相同的属性。比如：

```
interface Named {  
    name: string;  
}  
  
let x: Named;  
// y's inferred type is { name: string; location: string; }  
let y = { name: 'Alice', location: 'Seattle' };  
x = y;
```

这里要检查 `y` 是否能赋值给 `x`，编译器检查 `x` 中的每个属性，看是否能在 `y` 中也找到对应属性。在这个例子中，`y` 必须包含名字是 `name` 的 `string` 类型成员。`y` 满足条件，因此赋值正确。

检查函数参数时使用相同的规则：

```
function greet(n: Named) {  
    alert('Hello, ' + n.name);  
}  
greet(y); // OK
```

注意，`y` 有个额外的 `location` 属性，但这不会引发错误。只有目标类型（这里是 `Named`）的成员会被一一检查是否兼容。

这个比较过程是递归进行的，检查每个成员及子成员。

## 比较两个函数

相对来讲，在比较原始类型和对象类型的时候是比较容易理解的，问题是如何判断两个函数是兼容的。下面我们从两个简单的函数入手，它们仅是参数列表略有不同：

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

要查看 `x` 是否能赋值给 `y`，首先看它们的参数列表。`x` 的每个参数必须能在 `y` 里找到对应类型的参数。注意的是参数的名字相同与否无所谓，只看它们的类型。这里，`x` 的每个参数在 `y` 中都能找到对应的参数，所以允许赋值。

第二个赋值错误，因为 `y` 有个必需的第二个参数，但是 `x` 并没有，所以不允许赋值。

你可能会疑惑为什么允许忽略参数，像例子 `y = x` 中那样。原因是忽略额外的参数在JavaScript里是很常见的。例如，`Array#forEach` 给回调函数传3个参数：数组元素，索引和整个数组。尽管如此，传入一个只使用第一个参数的回调函数也是很有用的：

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

// Don't force these extra arguments
items.forEach((item, index, array) => console.log(item));

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

下面来看看如何处理返回值类型，创建两个仅是返回值类型不同的函数：

```
let x = () => ({name: 'Alice'});
let y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK
y = x; // Error because x() lacks a location property
```

类型系统强制源函数的返回值类型必须是目标函数返回值类型的子类型。

## 函数参数双向协变

当比较函数参数类型时，只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功。这是不稳定的，因为调用者可能传入了一个具有更精确类型信息的函数，但是调用这个传入的函数的时候却使用了不是那么精确的类型信息。实际上，这极少会发生错误，并且能够实现很多JavaScript里的常见模式。例如：

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly soundness
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

## 可选参数及剩余参数

比较函数兼容性的时候，可选参数与必须参数是可交换的。原类型上额外的可选参数并不会造成错误，目标类型的可选参数没有对应的参数也不是错误。

当一个函数有剩余参数时，它被当做无限个可选参数。

这对于类型系统来说是不稳定的，但从运行时的角度来看，可选参数一般来说是不强制的，因为对于大多数函数来说相当于传递了一些 `undefined`。

有一个好的例子，常见的函数接收一个回调函数并用对于程序员来说是可预知的参数但对类型系统来说是不确定的参数来调用：

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ', ' + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ', ' + y));
```

## 函数重载

对于有重载的函数，源函数的每个重载都要在目标函数上找到对应的函数签名。这确保了目标函数可以在所有源函数可调用的地方调用。

## 枚举

枚举类型与数字类型兼容，并且数字类型与枚举类型兼容。不同枚举类型之间是不兼容的。比如，

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

let status = Status.Ready;
status = Color.Green; //error
```

## 类

类与对象字面量和接口差不多，但有一点不同：类有静态部分和实例部分的类型。比较两个类类型的对象时，只有实例的成员会被比较。静态成员和构造函数不在比较的范围内。

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s; //OK
s = a; //OK
```

## 类的私有成员

私有成员会影响兼容性判断。当类的实例用来检查兼容时，如果它包含一个私有成员，那么目标类型必须包含来自同一个类的这个私有成员。这允许子类赋值给父类，但是不能赋值给其它有同样类型的类。

## 泛型

因为TypeScript是结构性的类型系统，类型参数只影响使用其做为类型一部分的结果类型。比如，

```
interface Empty<T> {
}

let x: Empty<number>;
let y: Empty<string>;

x = y; // okay, y matches structure of x
```

上面代码里，`x` 和 `y` 是兼容的，因为它们的结构使用类型参数时并没有什么不同。把这个例子改变一下，增加一个成员，就能看出是如何工作的了：



```
interface NotEmpty<T> {  
    data: T;  
}  
let x: NotEmpty<number>;  
let y: NotEmpty<string>;  
  
x = y; // error, x and y are not compatible
```

在这里，泛型类型在使用时就好比不是一个泛型类型。

对于没指定泛型类型的泛型参数时，会把所有泛型参数当成 `any` 比较。然后用结果类型进行比较，就像上面第一个例子。

比如，

```
let identity = function<T>(x: T): T {  
    // ...  
}  
  
let reverse = function<U>(y: U): U {  
    // ...  
}  
  
identity = reverse; // Okay because (x: any)=>any matches (y: any)
```

## 高级主题

### 子类型与赋值

目前为止，我们使用了 `兼容性`，它在语言规范里没有定义。在 `TypeScript` 里，有两种类型的兼容性：子类型与赋值。它们的不同点在于，赋值扩展了子类型兼容，允许给 `any` 赋值或从 `any` 取值和允许数字赋值给枚举类型或枚举类型赋值给数字。

语言里的不同地方分别使用了它们之中的机制。实际上，类型兼容性是由赋值兼容性来控制的甚至在 `implements` 和 `extends` 语句里。更多信息，请参阅 [TypeScript语言规范](#)。

## 联合类型

偶尔你会遇到这种情况，一个代码库希望传入 `number` 或 `string` 类型的参数。例如下面的函数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left.
 * If 'padding' is a number, then that number of spaces is added to the left.
 */
function padLeft(value: string, padding: any) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

`padLeft` 存在一个问题，`padding` 参数的类型指定成了 `any`。这就是说我们可以传入一个既不是 `number` 也不是 `string` 类型的参数，但是 TypeScript 却不报错。

```
let indentedString = padLeft("Hello world", true); // 编译阶段通过，运行时报错
```

在传统的面向对象语言里，我们可能会将这两种类型抽象成有层级的类型。这么做显然是非常清晰的，但同时也存在了过度设计。`padLeft` 原始版本的好处之一是允许我们传入原始类型。这么做的话使用起来既方便又不过于繁琐。如果我们就是想使用已经存在的函数的话，这种新的方式就不适用了。

代替 `any`，我们可以使用联合类型做为 `padding` 的参数：

```

/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left.
 * If 'padding' is a number, then that number of spaces is added to the left.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during

```

联合类型表示一个值可以是几种类型之一。我们用竖线（`|`）分隔每个类型，所以 `number | string | boolean` 表示一个值可以是 `number`，`string`，或 `boolean`。

如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的成员。

```

interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors

```

这里的联合类型可能有点复杂，但是你很容易就习惯了。如果一个值类型是 `A | B`，我们只能确定它具有成员同时存在于 `A` 和 `B` 里。这个例子里，`Bird` 具有一个 `fly` 成员。我们不能确定一个 `Bird | Fish` 类型的变量是否有 `fly` 方法。如果变量在运行时是 `Fish` 类型，那么调用 `pet.fly()` 就出错了。

## 类型保护与区分类型

联合类型非常适合这样的情形，可接收的值有不同的类型。当我们想明确地知道是否拿到 `Fish` 时会怎么做？JavaScript里常用来区分2个可能值的方法是检查它们是否存在。像之前提到的，我们只能访问联合类型的所有类型中共有的成员。

```
let pet = getSmallPet();

// 每一个成员访问都会报错
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

为了让这码代码工作，我们要使用类型断言：

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

## 用户自定义的类型保护

可以注意到我们使用了多次类型断言。如果我们只要检查过一次类型，就能够在后面的每个分支里清楚 `pet` 的类型的话就好了。

TypeScript里的类型保护机制让它成为了现实。类型保护就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。要定义一个类型保护，我们只要简单地定义一个函数，它的返回值是一个类型断言：

```
function isFish(pet: Fish | Bird): pet is Fish {  
    return (<Fish>pet).swim !== undefined;  
}
```

在这个例子里，`pet is Fish` 就是类型断言。一个断言是 `parameterName is Type` 这种形式，`parameterName` 必须是来自于当前函数签名里的一个参数名。

每当使用一些变量调用 `isFish` 时，TypeScript会将变量缩减为那个具体的类型，只要这个类型与变量的原始类型是兼容的。

```
// 'swim' 和 'fly' 调用都没有问题了  
  
if (isFish(pet)) {  
    pet.swim();  
}  
else {  
    pet.fly();  
}
```

注意TypeScript不仅知道在 `if` 分支里 `pet` 是 `Fish` 类型；它还清楚在 `else` 分支里，一定不是 `Fish` 类型，一定是 `Bird` 类型。

## typeof 类型保护

我们还没有真正的讨论过如何使用联合类型来实现 `padLeft`。我们可以像下面这样利用类型断言来写：

```
function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

然而，必须要定义一个函数来判断类型是否是原始类型，这太痛苦了。幸运的是，现在我们可以不必将 `typeof x === "number"` 抽象成一个函数，因为 TypeScript 可以将它识别为一个类型保护。也就是说我们可以在代码里检查类型了。

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

这些 `typeof` 类型保护只有2个形式能被识别：`typeof v === "typename"` 和 `typeof v !== "typename"`，`"typename"` 必须是 `"number"`，`"string"`，`"boolean"` 或 `"symbol"`。但是 TypeScript 并不

会阻止你与其它字符串比较，或者将它们位置对换，且语言不会把它们识别为类型保护。

## **instanceof** 类型保护

如果你已经阅读了 `typeof` 类型保护并且对JavaScript里的 `instanceof` 操作符熟悉的话，你可能已经猜到了这节要讲的内容。

`instanceof` 类型保护是通过其构造函数来细化其类型。比如，我们借鉴一下之前字符串填充的例子：



```

interface Padder {
    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder(" ");
}

// 类型为SpaceRepeatingPadder | StringPadder
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // 类型细化为'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // 类型细化为'StringPadder'
}

```

`instanceof` 的右侧要求为一个构造函数，TypeScript将细化为：

1. 这个函数的 `prototype` 属性，如果它的类型不为 `any` 的话
2. 类型中构造签名所返回的类型的联合，顺序保持一至。

## 交叉类型

交叉类型与联合类型密切相关，但是用法却完全不同。一个交叉类型，例如 `Person & Serializable & Loggable`，同时是 `Person` 和 `Serializable` 和 `Loggable`。就是说这个类型的对象同时拥有这三种类型的成员。实际应用中，你大多会在混入中见到交叉类型。下面是一个混入的例子：

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}

interface Loggable {
    log(): void;
}

class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

## 类型别名

类型别名会给一个类型起个新名字。类型别名有时和接口很像，但是可以作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    }
    else {
        return n();
    }
}
```

起别名不会新建一个类型 - 它创建了一个新名字来引用那个类型。给原始类型起别名通常没什么用，尽管可以做为文档的一种形式使用。

同接口一样，类型别名也可以是泛型 - 我们可以添加类型参数并且在别名声明的右侧传入：

```
type Container<T> = { value: T };
```

我们也可以使用类型别名来在属性里引用自己：

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

然而，类型别名不可能出现在声明右侧以外的地方：

```
type Yikes = Array<Yikes>; // 错误
```

## 接口 **vs.** 类型别名

像我们提到的，类型别名可以像接口一样；然而，仍有一些细微差别。

一个重要区别是类型别名不能被 `extends` 和 `implements` 也不能去 `extends` 和 `implements` 其它类型。因为软件中的对象应该对于扩展是开放的，但是对于修改是封闭的，你应该尽量去使用接口代替类型别名。

另一方面，如果你无法通过接口来描述一个类型并且需要使用联合类型或元组类型，这时通常会使用类型别名。

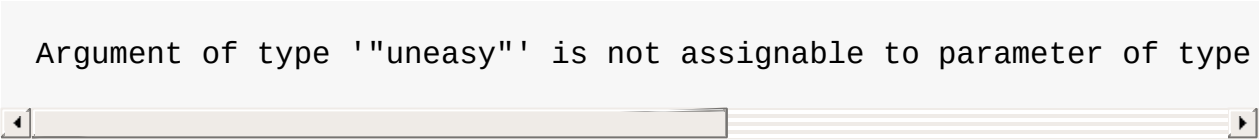
## 字符串字面量类型

字符串字面量类型允许你指定字符串必须的固定值。在实际应用中，字符串字面量类型可以与联合类型，类型保护和类型别名很好的配合。通过结合使用这些特性，你可以实现类似枚举类型的字符串。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
  animate(dx: number, dy: number, easing: Easing) {
    if (easing === "ease-in") {
      // ...
    }
    else if (easing === "ease-out") {
    }
    else if (easing === "ease-in-out") {
    }
    else {
      // error! should not pass null or undefined.
    }
  }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed t
```

你只能从三种允许的字符中选择其一来做为参数传递，传入其它值则会产生错误。



Argument of type '"uneasy"' is not assignable to parameter of type

字符串字面量类型还可以用于区分函数重载：

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

## 多态的 `this` 类型

多态的 `this` 类型表示的是某个包含类或接口的子类型。这被称做 **F-bounded** 多态性。它能很容易的表现连贯接口间的继承，比如。在计算器的例子里，在每个操作之后都返回 `this` 类型：

```
class BasicCalculator {  
    public constructor(protected value: number = 0) { }  
    public currentValue(): number {  
        return this.value;  
    }  
    public add(operand: number): this {  
        this.value += operand;  
        return this;  
    }  
    public multiply(operand: number): this {  
        this.value *= operand;  
        return this;  
    }  
    // ... other operations go here ...  
}  
  
let v = new BasicCalculator(2)  
    .multiply(5)  
    .add(1)  
    .currentValue();
```

由于这个类使用了 `this` 类型，你可以继承它，新的类可以直接使用之前的方法，不需要做任何的改变。

```
class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
    .multiply(5)
    .sin()
    .add(1)
    .currentValue();
```

如果没有 `this` 类型，`ScientificCalculator` 就不能够在继承 `BasicCalculator` 的同时还保持接口的连贯性。`multiply` 将会返回 `BasicCalculator`，它并没有 `sin` 方法。然而，使用 `this` 类型，`multiply` 会返回 `this`，在这里就是 `ScientificCalculator`。

## 介绍

自ECMAScript 2015起，`symbol` 成为了一种新的原生类型，就像 `number` 和 `string` 一样。

`symbol` 类型的值是通过 `Symbol` 构造函数创建的。

```
let sym1 = Symbol();

let sym2 = Symbol("key"); // 可选的字符串key
```

`Symbols`是不可改变且唯一的。

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols是唯一的
```

像字符串一样，`symbols`也可以被用做对象属性的键。

```
let sym = Symbol();

let obj = {
  [sym]: "value"
};

console.log(obj[sym]); // "value"
```

`Symbols`也可以与计算出的属性名声明相结合来声明对象的属性和类成员。



```
const getClassNameSymbol = Symbol();

class C {
  [getClassNameSymbol]() {
    return "C";
  }
}

let c = new C();
let className = c[getClassNameSymbol](); // "C"
```

## 众所周知的Symbols

除了用户定义的symbols，还有一些已经众所周知的内置symbols。内置symbols用来表示语言内部的行为。

以下为这些symbols的列表：

### Symbol.hasInstance

方法，会被 `instanceof` 运算符调用。构造器对象用来识别一个对象是否是其实例。

### Symbol.isConcatSpreadable

布尔值，表示当在一个对象上调用 `Array.prototype.concat` 时，这个对象的数组元素是否可展开。

### Symbol.iterator

方法，被 `for-of` 语句调用。返回对象的默认迭代器。

### Symbol.match

方法，被 `String.prototype.match` 调用。正则表达式用来匹配字符串。

## Symbol.replace

方法，被 `String.prototype.replace` 调用。正则表达式用来替换字符串中匹配的子串。

## Symbol.search

方法，被 `String.prototype.search` 调用。正则表达式返回被匹配部分在字符串中的索引。

## Symbol.species

函数值，为一个构造函数。用来创建派生对象。

## Symbol.split

方法，被 `String.prototype.split` 调用。正则表达式来用分割字符串。

## Symbol.toPrimitive

方法，被 `ToPrimitive` 抽象操作调用。把对象转换为相应的原始值。

## Symbol.toStringTag

方法，被内置方法 `Object.prototype.toString` 调用。返回创建对象时默认的字符串描述。

## Symbol.unscopables

对象，它自己拥有的属性会被 `with` 作用域排除在外。

## 可迭代性

当一个对象实现了 `Symbol.iterator` 属性时，我们认为它是可迭代的。一些内置的类型

如 `Array`，`Map`，`Set`，`String`，`Int32Array`，`Uint32Array` 等都已经实现了各自的 `Symbol.iterator`。对象上的 `Symbol.iterator` 函数负责返回供迭代的值。

### `for..of` 语句

`for..of` 会遍历可迭代的对象，调用对象上的 `Symbol.iterator` 方法。下面是在数组上使用 `for..of` 的简单例子：

```
let someArray = [1, "string", false];

for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}
```

### `for..of` vs. `for..in` 语句

`for..of` 和 `for..in` 均可迭代一个列表；但是用于迭代的值却不同，`for..in` 迭代的是对象的键的列表，而 `for..of` 则迭代对象的键对应的值。

下面的例子展示了两者之间的区别：

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

另一个区别是 `for..in` 可以操作任何对象；它提供了查看对象属性的一种方法。但是 `for..of` 关注于迭代对象的值。内置对象 `Map` 和 `Set` 已经实现了 `Symbol.iterator` 方法，让我们可以访问它们保存的值。

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
    console.log(pet); // "species"
}

for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

## 代码生成

### 目标为 **ES5** 和 **ES3**

当生成目标为ES5或ES3，迭代器只允许在 `Array` 类型上使用。在非数组值上使用 `for..of` 语句会得到一个错误，就算这些非数组值已经实现了 `Symbol.iterator` 属性。

编译器会生成一个简单的 `for` 循环做为 `for..of` 循环，比如：

```
let numbers = [1, 2, 3];
for (let num of numbers) {
  console.log(num);
}
```

生成的代码为：

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
  var num = numbers[_i];
  console.log(num);
}
```

## 目标为 **ECMAScript 2015** 或更高

当目标为兼容ECMAScript 2015的引擎时，编译器会生成相应引擎的 `for..of` 内置迭代器实现方式。

关于术语的一点说明: 请务必注意一点, TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”, 这是为了与ECMAScript 2015里的术语保持一致, (也就是说 `module X {` 相当于现在推荐的写法 `namespace X {` )。

## 介绍

从ECMAScript 2015开始, JavaScript引入了模块的概念。TypeScript也沿用这个概念。

模块在其自身的作用域里执行, 而不是在全局作用域里; 这意味着定义在一个模块里的变量, 函数, 类等等在模块外部是不可见的, 除非你明确地使用 `export` 形式之一导出它们。相反, 如果想使用其它模块导出的变量, 函数, 类, 接口等的时候, 你必须导入它们, 可以使用 `import` 形式之一。

模块是自声明的; 两个模块之间的关系是通过在文件级别上使用imports和exports建立的。

模块使用模块加载器去导入其它的模块。在运行时, 模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。大家最熟知的JavaScript模块加载器是服务于Node.js的CommonJS和服务于Web应用的Require.js。

TypeScript与ECMAScript 2015一样, 任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块。

## 导出

### 导出声明

任何声明 (比如变量, 函数, 类, 类型别名或接口) 都能够通过添加 `export` 关键字来导出。

#### Validation.ts

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

### ZipCodeValidator.ts

```
export const numberRegex = /^[0-9]+$/;  
  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}
```

## 导出语句

导出语句很便利，因为我们可能需要对导出的部分重命名，所以上面的例子可以这样改写：

```
class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}  
  
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```

## 重新导出

我们经常会去扩展其它模块，并且只导出那个模块的部分内容。重新导出功能并不会在当前模块导入那个模块或定义一个新的局部变量。

### ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// 导出原先的验证器但做了重命名
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";
```

或者一个模块可以包裹多个模块，并把他们导出的内容联合在一起通过语法：`export * from "module"`。

### AllValidators.ts

```
export * from "./StringValidator"; // exports interface StringValidator
export * from "./LettersOnlyValidator"; // exports class LettersOnlyValidator
export * from "./ZipCodeValidator"; // exports class ZipCodeValidator
```

## 导入

模块的导入操作与导出一样简单。可以使用以下 `import` 形式之一来导入其它模块中的导出内容。

### 导入一个模块中的某个导出内容

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

可以对导入内容重命名

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```



## 将整个模块导入到一个变量，并通过它来访问模块的导出部分

```
import * as validator from "./ZipCodeValidator";  
let myValidator = new validator.ZipCodeValidator();
```

## 具有副作用的导入模块

尽管不推荐这么做，一些模块会设置一些全局状态供其它模块使用。这些模块可能没有任何的导出或用户根本就不关注它的导出。使用下面的方法来导入这类模块：

```
import "./my-module.js";
```

## 默认导出

每个模块都可以有一个 `default` 导出。默认导出使用 `default` 关键字标记；并且一个模块只能够有一个 `default` 导出。需要使用一种特殊的导入形式来导入 `default` 导出。

`default` 导出十分便利。比如，像jQuery这样的类库可能有一个默认导出 `jQuery` 或 `$`，并且我们基本上也会使用同样的名字 `jQuery` 或 `$` 导出 `jQuery`。

### jQuery.d.ts

```
declare let $: JQuery;  
export default $;
```

### App.ts

```
import $ from "jQuery";  
  
$("button.continue").html( "Next Step..." );
```

类和函数声明可以直接被标记为默认导出。标记为默认导出的类和函数的名字是可以省略的。

### ZipCodeValidator.ts

```
export default class ZipCodeValidator {  
    static numberRegexp = /^[0-9]+$/;  
    isAcceptable(s: string) {  
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);  
    }  
}
```

### Test.ts

```
import validator from "./ZipCodeValidator";  
  
let myValidator = new validator();
```

或者

### StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;  
  
export default function (s: string) {  
    return s.length === 5 && numberRegexp.test(s);  
}
```

### Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

`default` 导出也可以是一个值

### OneTwoThree.ts

```
export default "123";
```

### Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

## `export =` 和 `import = require()`

CommonJS和AMD都有一个 `exports` 对象的概念，它包含了一个模块的所有导出内容。

它们也支持把 `exports` 替换为一个自定义对象。默认导出就好比这样一个功能；然而，它们却并不相互兼容。TypeScript模块支持 `export =` 语法，以配合传统的CommonJS和AMD的工作流。

`export =` 语法定义一个模块的导出对象。它可以是类，接口，命名空间，函数或枚举。

若要导入一个使用了 `export =` 的模块时，必须使用TypeScript提供的特定语法 `import let = require("module")`。

## ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export = ZipCodeValidator;
```

## Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${s}" - ${validator.isAcceptable(s) ? "matches"
});
```

## 生成模块代码

根据编译时指定的模块目标参数，编译器会生成相应的供Node.js ([CommonJS](#))，Require.js ([AMD](#))，isomorphic ([UMD](#))，[SystemJS](#)或[ECMAScript 2015 native modules](#) (ES6)模块加载系统使用的代码。想要了解生成代码

中 `define`，`require` 和 `register` 的意义，请参考相应模块加载器的文档。

下面的例子说明了导入导出语句里使用的名字是怎么转换为相应的模块加载器代码的。

## SimpleModule.ts

```
import m = require("mod");  
export let t = m.something + 1;
```

### AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports,  
    exports.t = mod_1.something + 1;  
});
```

### CommonJS / Node SimpleModule.js

```
let mod_1 = require("./mod");  
exports.t = mod_1.something + 1;
```

### UMD SimpleModule.js

```
(function (factory) {  
    if (typeof module === "object" && typeof module.exports === "object")  
        let v = factory(require, exports); if (v !== undefined) module.exports = v;  
    else if (typeof define === "function" && define.amd) {  
        define(["require", "exports", "./mod"], factory);  
    }  
})(function (require, exports) {  
    let mod_1 = require("./mod");  
    exports.t = mod_1.something + 1;  
});
```

### System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    let mod_1;
    let t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            },
        ],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

## Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export let t = something + 1;
```

## 简单示例

下面我们来整理一下前面的验证器实现，每个模块只有一个命名的导出。

为了编译，我们必需要在命令行上指定一个模块目标。对于Node.js来说，使用 `--module commonjs`；对于Require.js来说，使用 `--module amd`。比如：

```
tsc --module commonjs Test.ts
```

编译完成后，每个模块会生成一个单独的 `.js` 文件。好比使用了reference标签，编译器会根据 `import` 语句编译相应的文件。

## Validation.ts

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

### LettersOnlyValidator.ts

```
import { StringValidator } from "../Validation";  
  
const lettersRegexp = /^[A-Za-z]+$/;  
  
export class LettersOnlyValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return lettersRegexp.test(s);  
    }  
}
```

### ZipCodeValidator.ts

```
import { StringValidator } from "../Validation";  
  
const numberRegexp = /^[0-9]+$/;  
  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegexp.test(s);  
    }  
}
```

### Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s)}`);
    }
});
```

## 可选的模块加载和其它高级加载场景

有时候，你只想在某种条件下才加载某个模块。在 TypeScript 里，使用下面的方式来实现它和其它的高级加载场景，我们可以直接调用模块加载器并且可以保证类型完全。

编译器会检测是否每个模块都会在生成的 JavaScript 中用到。如果一个模块标识符只在类型注解部分使用，并且完全没有在表达式中使用，就不会生成 `require` 这个模块的代码。省略掉没有用到的引用对性能提升是很有益的，并同时提供了选择性加载模块的能力。

这种模式的核心是 `import id = require("...")` 语句可以让我们访问模块导出的类型。模块加载器会被动态调用（通过 `require`），就像下面 `if` 代码块里那样。它利用了省略引用的优化，所以模块只在被需要时加载。为了让这个模块工作，一定要注意 `import` 定义的标识符只能在表示类型处使用（不能在会转换成 JavaScript 的地方）。



为了确保类型安全性，我们可以使用 `typeof` 关键字。 `typeof` 关键字，当在表示类型的地方使用时，会得出一个类型值，这里就表示模块的类型。

示例：**Node.js**里的动态模块加载

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

示例：**require.js**里的动态模块加载

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

示例：**System.js**里的动态模块加载

```
declare let System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: type) => {
        let x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

## 使用其它的JavaScript库

为了描述不是用TypeScript编写的类库的类型，我们需要声明类库导出的API。

我们叫它声明因为它不是外部程序的具体实现。通常会在 `.d.ts` 里写这些定义。如果你熟悉C/C++，你可以把它们当做 `.h` 文件。让我们看一些例子。

### 外部模块

在Node.js里大部分工作是通过加载一个或多个模块实现的。我们可以使用顶级的 `export` 声明来为每个模块都定义一个 `.d.ts` 文件，但最好还是写在一个大的 `.d.ts` 文件里。我们使用与构造一个外部命名空间相似的方法，但是这里使用 `module` 关键字并且把名字用引号括起来，方便之后 `import`。例如：

#### node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashes?: boolean): Url;
}

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

现在我们可以 `/// <reference> node.d.ts` 并且使用 `import url = require("url");` 加载模块。

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");
```

## 创建模块结构指导

### 尽可能地在顶层导出

用户应该更容易地使用你模块导出的内容。嵌套层次过多会变得难以处理，因此仔细考虑一下如何组织你的代码。

从你的模块中导出一个命名空间就是一个增加嵌套的例子。虽然命名空间有时候有它们的用处，在使用模块的时候它们额外地增加了一层。这对用户来说是很不便的并且通常是多余的。

导出类的静态方法也有同样的问题 - 这个类本身就增加了一层嵌套。除非它能方便表述或便于清晰使用，否则请考虑直接导出一个辅助方法。

## 如果仅导出单个 `class` 或 `function`，使用 `export default`

就像“在顶层上导出”帮助减少用户使用的难度，一个默认的导出也能起到这个效果。如果一个模块就是为了导出特定的内容，那么你应该考虑使用一个默认导出。这会令模块的导入和使用变得些许简单。比如：

### MyClass.ts

```
export default class SomeType {  
  constructor() { ... }  
}
```

### MyFunc.ts

```
export default function getThing() { return 'thing'; }
```

### Consumer.ts

```
import t from "./MyClass";  
import f from "./MyFunc";  
let x = new t();  
console.log(f());
```

对用户来说这是最理想的。他们可以随意命名导入模块的类型（本例为 `t`）并且不需要多余的 `(.)` 来找到相关对象。

如果要导出多个对象，把它们放在顶层里导出

### MyThings.ts

```
export class SomeType { /* ... */ }  
export function someFunc() { /* ... */ }
```

相反地，当导入的时候：

## 明确地列出导入的名字

### Consumer.ts

```
import { SomeType, SomeFunc } from "./MyThings";  
let x = new SomeType();  
let y = someFunc();
```

使用命名空间导入模式当你要导出大量内容的时候

### MyLargeModule.ts

```
export class Dog { ... }  
export class Cat { ... }  
export class Tree { ... }  
export class Flower { ... }
```

### Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";  
let x = new myLargeModule.Dog();
```

## 使用重新导出进行扩展

你可能经常需要去扩展一个模块的功能。JS里常用的一个模式是jQuery那样去扩展原对象。如我们之前提到的，模块不会像全局命名空间对象那样去合并。推荐的方案是不要去改变原来的对象，而是导出一个新的实体来提供新的功能。

假设 `Calculator.ts` 模块里定义了一个简单的计算器实现。这个模块同样提供了一个辅助函数来测试计算器的功能，通过传入一系列输入的字符串并在最后给出结果。

## Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".c
        }
    }

    protected processOperator(operator: string) {
        if (["+","-","*","/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handelChar(char: string) {
        if (char === "=") {

```

```
        this.evaluate();
        return;
    }
    else {
        let value = this.processDigit(char, this.current);
        if (value !== undefined) {
            this.current = value;
            return;
        }
        else {
            let value = this.processOperator(char);
            if (value !== undefined) {
                this.evaluate();
                this.operator = value;
                return;
            }
        }
    }
    throw new Error(`Unsupported input: '${char}'`);
}

public getResult() {
    return this.memory;
}
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handelChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

这是使用导出的 `test` 函数来测试计算器。

## TestCalculator.ts

```
import { Calculator, test } from "./Calculator";

let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

现在扩展它，添加支持输入其它进制（十进制以外），让我们来创建 `ProgrammerCalculator.ts`。

## ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e", "f"];

    constructor(public base: number) {
        super();
        if (base <= 0 || base > ProgrammerCalculator.digits.length)
            throw new Error("base has to be within 0 to 16 inclusive");
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```



新的 `ProgrammerCalculator` 模块导出的API与原先的 `Calculator` 模块很相似，但却没有改变原模块里的对象。下面是测试`ProgrammerCalculator`类的代码：

## TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

## 模块里不要使用命名空间

当初次进入基于模块的开发模式时，可能总会控制不住要将导出包裹在一个命名空间里。模块具有其自己的作用域，并且只有导出的声明才会在模块外部可见。记住这点，命名空间在使用模块时几乎没什么价值。

在组织方面，命名空间对于在全局作用域内对逻辑上相关的对象和类型进行分组是很便利的。例如，在C#里，你会从 `System.Collections` 里找到所有集合的类型。通过将类型有层次地组织在命名空间里，可以方便用户找到与使用那些类型。然而，模块本身已经存在于文件系统之中，这是必须的。我们必须通过路径和文件名找到它们，这已经提供了一种逻辑上的组织形式。我们可以创建 `/collections/generic/` 文件夹，把相应模块放在这里面。

命名空间对解决全局作用域里命名冲突来说是很重要的。比如，你可以有一个 `My.Application.Customer.AddForm` 和 `My.Application.Order.AddForm` - 两个类型的名字相同，但命名空间不同。然而，这对于模块来说却不是一个問題。在一个模块里，没有理由两个对象拥有同一个名字。从模块的使用角度来说，使用者会挑出他们用来引用模块的名字，所以也没有理由发生重名的情况。

更多关于模块和命名空间的资料查看[命名空间和模块](#)

## 危险信号

以下均为模块结构上的危险信号。重新检查以确保你没有在对模块使用命名空间：

- 文件的顶层声明是 `export namespace Foo { ... }` （删除 `Foo` 并把所有内容向上层移动一层）

- 文件只有一个 `export class` 或 `export function` （考虑使用 `export default` ）
- 多个文件的顶层具有同样的 `export namespace Foo {` （不要以为这些会合并到一个 `Foo` 中！）

关于术语的一点说明: 请务必注意一点, TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”, 这是为了与ECMAScript 2015里的术语保持一致, (也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

## 介绍

这篇文章描述了如何在TypeScript里使用命名空间(之前叫做“内部模块”)来组织你的代码。

就像我们在术语说明里提到的那样, “内部模块”现在叫做“命名空间”。

另外, 任何使用 `module` 关键字来声明一个内部模块的地方都应该使用 `namespace` 关键字来替换。

这就避免了让新的使用者被相似的名称所迷惑。

## 第一步

我们先来写一段程序并将在整篇文章中都使用这个例子。我们定义几个简单的字符串验证器, 假设你会使用它们来验证表单里的用户输入或验证外部数据。

## 所有的验证器都放在一个文件里

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

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];
// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log("" + s + " " + (validators[name].isAcceptable(s) ? "pass" : "fail"));
    }
});
```

## 命名空间

随着更多验证器的加入，我们需要一种手段来组织代码，以便于在记录它们类型的同时还不用担心与其它对象产生命名冲突。因此，我们把验证器包裹到一个命名空间内，而不是把它们放在全局命名空间下。

下面的例子里，把所有与验证器相关的类型都放到一个叫做 `Validation` 的命名空间里。因为我们想让这些接口和类在命名空间之外也是可访问的，所以需要使  
用 `export`。相反的，变量 `lettersRegexp` 和 `numberRegexp` 是实现细节，  
不需要导出，因此它们在命名空间外是不能访问的。在文件末尾的测试代码里，由  
于是在命名空间之外访问，因此需要限定类型的名称，比  
如 `Validation.LettersOnlyValidator`。

## 使用命名空间的验证器

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

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];
// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s)}`);
    }
});
```

## 分离到多文件

当应用变得越来越大时，我们需要将代码分离到不同的文件中以便于维护。

## 多文件中的命名空间

现在，我们把 `Validation` 命名空间分割成多个文件。尽管是不同的文件，它们仍是同一个命名空间，并且在使用的时候就如同它们在一个文件中定义的一样。因为不同文件之间存在依赖关系，所以我们加入了引用标签来告诉编译器文件之间的关联。我们的测试代码保持不变。

### Validation.ts

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

### LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />  
namespace Validation {  
    const lettersRegexp = /^[A-Za-z]+$/;  
    export class LettersOnlyValidator implements StringValidator {  
        isAcceptable(s: string) {  
            return lettersRegexp.test(s);  
        }  
    }  
}
```

### ZipCodeValidator.ts

```

/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegex = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegex.test(s);
        }
    }
}

```

## Test.ts

```

/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];
// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log("" + s + " " + (validators[name].isAcceptable(s) ? "true" : "false"));
    }
});

```

当涉及到多文件时，我们必须确保所有编译后的代码都被加载了。我们有两种方式。

第一种方式，把所有的输入文件编译为一个输出文件，需要使用 `--outFile` 标记：

```
tsc --outFile sample.js Test.ts
```



编译器会根据源码里的引用标签自动地对输出进行排序。你也可以单独地指定每个文件。

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCo
```

第二种方式，我们可以编译每一个文件（默认方式），那么每个源文件都会对应生成一个JavaScript文件。然后，在页面上通过 `<script>` 标签把所有生成的JavaScript文件按正确的顺序引进来，比如：

### MyTestPage.html (excerpt)

```
<script src="Validation.js" type="text/javascript" />
<script src="LettersOnlyValidator.js" type="text/javascript" />
<script src="ZipCodeValidator.js" type="text/javascript" />
<script src="Test.js" type="text/javascript" />
```

## 别名

另一种简化命名空间操作的方法是使用 `import q = x.y.z` 给常用的对象起一个短的名字。不要与用来加载模块的 `import x = require('name')` 语法弄混了，这里的语法是为指定的符号创建一个别名。你可以用这种方法为任意标识符创建别名，也包括导入的模块中的对象。

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as "new Shapes.Polygons.Sq
```

注意，我们并没有使用 `require` 关键字，而是直接使用导入符号的限定名赋值。这与使用 `var` 相似，但它还适用于类型和导入的具有命名空间含义的符号。重要的是，对于值来讲，`import` 会生成与原始符号不同的引用，所以改变别名的 `var` 值并不会影响原始变量的值。

## 使用其它的JavaScript库

为了描述不是用TypeScript编写的类库的类型，我们需要声明类库导出的API。由于大部分程序库只提供少数的顶级对象，命名空间是用来表示它们的一个好办法。

我们称其为声明是因为它不是外部程序的具体实现。我们通常在 `.d.ts` 里写这些声明。如果你熟悉C/C++，你可以把它们当做 `.h` 文件。让我们看一些例子。

### 外部命名空间

流行的程序库D3在全局对象 `d3` 里定义它的功能。因为这个库通过一个 `<script>` 标签加载（不是通过模块加载器），它的声明文件使用内部模块来定义它的类型。为了让TypeScript编译器识别它的类型，我们使用外部命名空间声明。比如，我们可以像下面这样写：

#### D3.d.ts (部分摘录)

```
declare namespace D3 {  
  export interface Selectors {  
    select: {  
      (selector: string): Selection;  
      (element: EventTarget): Selection;  
    };  
  }  
  
  export interface Event {  
    x: number;  
    y: number;  
  }  
  
  export interface Base extends Selectors {  
    event: Event;  
  }  
}  
  
declare let d3: D3.Base;
```

关于术语的一点说明: 请务必注意一点, TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”, 这是为了与ECMAScript 2015里的术语保持一致, (也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

## 介绍

这篇文章将概括介绍在TypeScript里使用模块与命名空间来组织代码的方法。我们也会谈及命名空间和模块的高级使用场景, 和在使用它们的过程中常见的陷阱。

查看[模块](#)章节了解关于模块的更多信息。查看[命名空间](#)章节了解关于命名空间的更多信息。

## 使用命名空间

命名空间是位于全局命名空间下的一个普通的带有名字的JavaScript对象。这令命名空间十分容易使用。它们可以在多文件中同时使用, 并通过 `--outFile` 结合在一起。命名空间是帮你组织Web应用不错的方式, 你可以把所有依赖都放在HTML页面的 `<script>` 标签里。

但就像其它的全局命名空间污染一样, 它很难去识别组件之间的依赖关系, 尤其是在大型的应用中。

## 使用模块

像命名空间一样, 模块可以包含代码和声明。不同的是模块可以声明它的依赖。

模块会把依赖添加到模块加载器上 (例如CommonJs / Require.js)。对于小型的JS应用来说可能没必要, 但是对于大型应用, 这点点的花费会带来长久的模块化和可维护性上的便利。模块也提供了更好的代码重用, 更强的封闭性以及更好的使用工具进行优化。

对于Node.js应用来说, 模块是默认并推荐的组织代码的方式。

从ECMAScript 2015开始, 模块成为了语言内置的部分, 应该会被所有正常的解释引擎所支持。因此, 对于新项目来说推荐使用模块做为组织代码的方式。

## 命名空间和模块的陷阱

这部分我们会描述常见的命名空间和模块的使用陷阱和如何去避免它们。

### 对模块使用 `///`

一个常见的错误是使用 `/// 引用模块文件，应该使用 import。要理解这之间的区别，我们首先应该弄清编译器是如何根据 import 路径（例如，import x from "..."; 或 import x = require("...") 里面的 ...，等等）来定位模块的类型信息的。`

编译器首先尝试去查找相应路径下的 `.ts`，`.tsx` 再或者 `.d.ts`。如果这些文件都找不到，编译器会查找外部模块声明。回想一下，它们是在 `.d.ts` 文件里声明的。

- `myModules.d.ts`

```
// In a .d.ts file or .ts file that is not a module:
declare module "SomeModule" {
    export function fn(): string;
}
```

- `myOtherModule.ts`

```
///
```

这里的引用标签指定了外来模块的位置。这就是一些Typescript例子中引用 `node.d.ts` 的方法。

### 不必要的命名空间

如果你想把命名空间转换为模块，它可能会像下面这个文件一件：

- `shapes.ts`

```
export namespace Shapes {  
    export class Triangle { /* ... */ }  
    export class Square { /* ... */ }  
}
```

顶层的模块 `Shapes` 包裹了 `Triangle` 和 `Square`。对于使用它的人来说这是令人迷惑和讨厌的：

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";  
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript里模块的一个特点是不同的模块永远也不会在相同的作用域内使用相同的名字。因为使用模块的人会为它们命名，所以完全没有必要把导出的符号包裹在一个命名空间里。

再次重申，不应该对模块使用命名空间，使用命名空间是为了提供逻辑分组和避免命名冲突。模块文件本身已经是一个逻辑分组，并且它的名字是由导入这个模块的代码指定，所以没有必要为导出的对象增加额外的模块层。

下面是改进的例子：

- `shapes.ts`

```
export class Triangle { /* ... */ }  
export class Square { /* ... */ }
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";  
let t = new shapes.Triangle();
```

## 模块的取舍

就像每个JS文件对应一个模块一样，TypeScript里模块文件与生成的JS文件也是一一对应的。这会产生一个效果，就是无法使用 `--outFile` 来让编译器合并多个模块文件为一个JavaScript文件。

这节假设你已经了解了模块的一些基本知识 请阅读[模块](#)文档了解更多信息。

模块解析就是指编译器所要依据的一个流程，用它来找出某个导入操作所引用的具体值。假设有一个导入语句 `import { a } from "moduleA"`；为了去检查任何对 `a` 的使用，编译器需要准确的知道它表示什么，并且会需要检查它的定义 `moduleA`。

这时候，编译器会想知道“`moduleA` 的shape是怎样的？”这听上去很简单，`moduleA` 可能在你写的某个 `.ts / .tsx` 文件里或者在你的代码所依赖的 `.d.ts` 里。

首先，编译器会尝试定位表示导入模块的文件。编译会遵循下列二种策略之一：[Classic](#)或[Node](#)。这些策略会告诉编译器到哪里去查找 `moduleA`。

如果它们失败了并且如果模块名是非相对的（且是在 `"moduleA"` 的情况下），编译器会尝试定位一个[外部模块声明](#)。我们接下来会讲到非相对导入。

最后，如果编译器还是不能解析这个模块，它会记录一个错误。在这种情况下，错误可能为 `error TS2307: Cannot find module 'moduleA'.`

## 相对 **vs.** 非相对模块导入

根据模块引用是相对的还是非相对的，模块导入会以不同的方式解析。

相对导入是以 `/`，`./` 或 `../` 开头的。下面是一些例子：

- `import Entry from "../components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

所有其它形式的导入被当作非相对的。下面是一些例子：

- `import * as $ from "jQuery";`
- `import { Component } from "angular2/core";`

相对导入解析时是相对于导入它的文件来的，并且不能解析为一个外部模块声明。你应该为你自己写的模块使用相对导入，这样能确保它们在运行时的相对位置。

## 模块解析策略



共有两种可用的模块解析策略：[Node](#)和[Classic](#)。你可以使用 `--moduleResolution` 标记为指定使用哪个。默认值为[Node](#)。

## Classic

这种策略以前是TypeScript默认的解析策略。现在，它存在的理由主要是为了向后兼容。

相对导入的模块是相对于导入它的文件进行解析的。因此 `/root/src/folder/A.ts` 文件里的 `import { b } from "../moduleB"` 会使用下面的查找流程：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

对于非相对模块的导入，编译器则会从包含导入文件的目录开始依次向上级目录遍历，尝试定位匹配的声明文件。

比如：

有一个对 `moduleB` 的非相对导入 `import { b } from "moduleB"`，它是在 `/root/src/folder/A.ts` 文件里，会以如下的方式来定位 `"moduleB"`：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

## Node

这个解析策略试图在运行时模仿[Node.js](#)模块解析机制。完整的Node.js解析算法可以在[Node.js module documentation](#)找到。

### Node.js如何解析模块

为了理解TypeScript编译依照的解析步骤，先弄明白Node.js模块是非常重要的。通常，在Node.js里导入是通过 `require` 函数调用进行的。Node.js会根据 `require` 的是相对路径还是非相对路径做出不同的行为。

相对路径很简单。例如，假设有一个文件路径为 `/root/src/moduleA.js`，包含了一个导入 `var x = require("./moduleB");` Node.js以下面的顺序解析这个导入：

1. 将 `/root/src/moduleB.js` 视为文件，检查是否存在。
2. 将 `/root/src/moduleB` 视为目录，检查是否它包含 `package.json` 文件并且其指定了一个 `"main"` 模块。在我们的例子中，如果Node.js发现文件 `/root/src/moduleB/package.json` 包含了 `{ "main": "lib/mainModule.js" }`，那么Node.js会引用 `/root/src/moduleB/lib/mainModule.js`。
3. 将 `/root/src/moduleB` 视为目录，检查它是否包含 `index.js` 文件。这个文件会被隐式地当作那个文件夹下的`"main"`模块。

你可以阅读Node.js文档了解更多详细信息：[file modules](#) 和 [folder modules](#)。

但是，[非相对模块名](#)的解析是个完全不同的过程。Node会在一个特殊的文件夹 `node_modules` 里查找你的模块。`node_modules` 可能与当前文件在同一级目录下，或者在上层目录里。Node会向上级目录遍历，查找每个 `node_modules` 直到它找到要加载的模块。

还是用上面例子，但假设 `/root/src/moduleA.js` 里使用的是非相对路径导入 `var x = require("moduleB");`。Node则会以下面的顺序去解析 `moduleB`，直到有一个匹配上。

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
3. `/root/src/node_modules/moduleB/index.js`
4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
6. `/root/node_modules/moduleB/index.js`
7. `/node_modules/moduleB.js`

8. `/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
9. `/node_modules/moduleB/index.js`

注意Node.js在步骤（4）和（7）会向上跳一级目录。

你可以阅读Node.js文档了解更多详细信息：[loading modules from node\\_modules](#)。

## TypeScript如何解析模块

TypeScript是模仿Node.js运行时的解析策略来在编译阶段定位模块定义文件。因此，TypeScript在Node解析逻辑基础上增加了TypeScript源文件的扩展名（`.ts`，`.tsx` 和 `.d.ts`）。同时，TypeScript在 `package.json` 里使用字段 `"typings"` 来表示类似 `"main"` 的意义 - 编译器会使用它来找到要使用的`"main"`定义文件。

比如，有一个导入语句 `import { b } from`  
`"./moduleB"` 在 `/root/src/moduleA.ts` 里，会以下面的流程来定位 `"./moduleB"`：

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (如果指定了 `"typings"` 属性)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

回想一下Node.js先查找 `moduleB.js` 文件，然后是合适的 `package.json`，再之后是 `index.js`。

类似地，非相对的导入会遵循Node.js的解析逻辑，首先查找文件，然后是合适的文件夹。因此 `/src/moduleA.ts` 文件里的 `import { b } from "moduleB"` 会以下面的查找顺序解析：

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)

5. `/root/src/node_modules/moduleB/index.ts`
6. `/root/src/node_modules/moduleB/index.tsx`
7. `/root/src/node_modules/moduleB/index.d.ts`
8. `/root/node_modules/moduleB.ts`
9. `/root/node_modules/moduleB.tsx`
10. `/root/node_modules/moduleB.d.ts`
11. `/root/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)
12. `/root/node_modules/moduleB/index.ts`
13. `/root/node_modules/moduleB/index.tsx`
14. `/root/node_modules/moduleB/index.d.ts`
15. `/node_modules/moduleB.ts`
16. `/node_modules/moduleB.tsx`
17. `/node_modules/moduleB.d.ts`
18. `/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)
19. `/node_modules/moduleB/index.ts`
20. `/node_modules/moduleB/index.tsx`
21. `/node_modules/moduleB/index.d.ts`

不要被这里步骤的数量吓到 - TypeScript只是在步骤 (8) 和 (15) 向上跳了两次目录。这并不比Node.js里的流程复杂。

## 使用 `--noResolve`

正常来讲编译器会在开始编译之前解析模块导入。每当它成功地解析了对一个文件 `import`，这个文件被会加到一个文件列表里，以供编译器稍后处理。

`--noResolve` 编译选项告诉编译器不要添加任何不是在命令行上传入的文件到编译列表。编译器仍然会尝试解析模块，但是只要没有指定这个文件，那么它就不会被包含在内。

比如

**app.ts**

```
import * as A from "moduleA" // OK, moduleA passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'r
```

```
tsc app.ts moduleA.ts --noResolve
```

使用 `--noResolve` 编译 `app.ts` :

- 可能正确找到 `moduleA` , 因为它在命令行上指定了。
- 找不到 `moduleB` , 因为没有在命令行上传递。

## 常见问题

### 为什么在 `exclude` 列表里的模块还会被编译器使用

`tsconfig.json` 将文件夹转变一个“工程”如果不指定任何 `“exclude”` 或 `“files”` , 文件夹里的所有文件包括 `tsconfig.json` 和所有的子目录都会在编译列表里。如果你想利用 `“exclude”` 排除某些文件, 甚至你想指定所有要编译的文件列表, 请使用 `“files”` 。

有些是被 `tsconfig.json` 自动加入的。它不会涉及到上面讨论的模块解析。如果编译器识别出一个文件是模块导入目标, 它就会加到编译列表里, 不管它是否被排除了。

因此, 要从编译列表中排除一个文件, 你需要在排除它的同时, 还要排除所有对它进行 `import` 或使用了 `/// 指令的文件。`

## 介绍

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

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

## 基础概念

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

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

理解每个声明创建了什么，有助于理解当声明合并时有哪些东西被合并了。

## 合并接口

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

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

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

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

如下例所示：

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

这三个接口合并成一个声明：

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

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

这个规则有一个例外是当出现特殊的函数签名时。如果签名里有一个参数的类型是单一的字符串字面量（比如，不是字符串字面量的联合类型），那么它将会被提升到重载列表的最顶端。

比如，下面的接口会合并到一起：

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

合并后的 `Document` 将会像下面这样：

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



## 合并命名空间

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

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

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

**Animals** 声明合并示例：

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

等同于：

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

除了这些合并外，你还需要了解非导出成员是如何处理的。非导出成员仅在其原始存在于的命名空间（未合并的）之内可见。这就是说合并之后，从其它命名空间合并进来的成员无法访问非导出成员。

下例提供了更清晰的说明：

```

namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

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

```

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

## 命名空间与类和函数和枚举类型合并

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

### 合并命名空间和类

这让我们可以表示内部类。

```

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

```

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

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

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

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

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

相似的，命名空间可以用来扩展枚举型：

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

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

## 非法的合并

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

## 模块扩展

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

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

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

它也可以很好地工作在TypeScript中，但编译器对

`Observable.prototype.map` 一无所知。你可以使用扩展模块来将它告诉编译器：

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

// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

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

## 全局扩展

你也可以在模块内部添加声明到全局作用域中。

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

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

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

全局扩展与模块扩展的行为和限制是相同的。

## 介绍

当使用外部JavaScript库或新的宿主API时，你需要一个声明文件（.d.ts）定义程序库的shape。这个手册包含了写.d.ts文件的高级概念，并带有一些例子，告诉你怎么去写一个声明文件。

## 指导与说明

### 流程

最好从程序库的文档而不是代码开始写.d.ts文件。这样保证不会被具体实现所干扰，而且相比于JS代码更易读。下面的例子会假设你正在参照文档写声明文件。

## 命名空间

当定义接口（例如：“options”对象），你会选择是否将这些类型放进命名空间里。这主要是靠主观判断 -- 如果使用的人主要是用这些类型来声明变量和参数，并且类型命名不会引起命名冲突，则放在全局命名空间里更好。如果类型不是被直接使用，或者没法起一个唯一的名字的话，就使用命名空间来避免与其它类型发生冲突。

## 回调函数

许多JavaScript库接收一个函数做为参数，之后传入已知的参数来调用它。当用这些类型为函数签名的时候，不要把这些参数标记成可选参数。正确的思考方式是“(调用者)会提供什么样的参数？”，不是“(函数)会使用到什么样的参数？”。

TypeScript 0.9.7+不会强制这种可选参数的使用，参数可选的双向协变可以被外部的linter强制执行。

## 扩展与声明合并

写声明文件的时候，要记住TypeScript扩展现有对象的方式。你可以选择用匿名类型或接口类型的方式声明一个变量：

## 匿名类型var

```
declare let MyPoint: { x: number; y: number; };
```

## 接口类型var

```
interface SomePoint { x: number; y: number; }  
declare let MyPoint: SomePoint;
```

从使用者角度来讲，它们是相同的，但是SomePoint类型能够通过接口合并来扩展：

```
interface SomePoint { z: number; }  
MyPoint.z = 4; // OK
```

是否想让你的声明是可扩展的取决于主观判断。通常来讲，尽量符合library的意图。

## 类的分解

TypeScript的类会创建出两个类型：实例类型，定义了类型的实例具有哪些成员；构造函数类型，定义了类构造函数具有哪些类型。构造函数类型也被称做类的静态部分类型，因为它包含了类的静态成员。

你可以使用 `typeof` 关键字来拿到类静态部分类型，在写声明文件时，想要把类明确的分解成实例类型和静态类型时是有用且必要的。

下面是一个例子，从使用者的角度来看，这两个声明是等同的：

## 标准版



```
class A {  
    static st: string;  
    inst: number;  
    constructor(m: any) {}  
}
```

## 分解版

```
interface A_Static {  
    new(m: any): A_Instance;  
    st: string;  
}  
interface A_Instance {  
    inst: number;  
}  
declare let A: A_Static;
```

这里的利弊如下：

- 标准方式可以使用`extends`来继承；分解的类不能。也可能会在未来版本的TypeScript里做出改变：是否允许任意`extends`表达式
- 都允许之后为类添加静态成员(通过合并声明的方式)
- 分解的类允许增加实例成员，标准版不允许
- 使用分解类的时候，需要为多类型成员起合理的名字

## 命名规则

一般来讲，不要给接口加I前缀（比如：`IColor`）。因为TypeScript的接口类型概念比C#或Java里的意义更为广泛，`IFoo`命名不利于这个特点。

## 例子

下面进行例子部分。对于每个例子，首先使用应用示例，然后是类型声明。如果有多个好的声明表示方法，会列出多个。

## 参数对象

### 应用示例

```
animalFactory.create("dog");
animalFactory.create("giraffe", { name: "ronald" });
animalFactory.create("panda", { name: "bob", height: 400 });
// Invalid: name must be provided if options is given
animalFactory.create("cat", { height: 32 });
```

### 类型声明

```
namespace animalFactory {
    interface AnimalOptions {
        name: string;
        height?: number;
        weight?: number;
    }
    function create(name: string, animalOptions?: AnimalOptions): A
}
```

## 带属性的函数

### 应用示例

```
zooKeeper.workSchedule = "morning";
zooKeeper(giraffeCage);
```

### 类型声明

```
// Note: Function must precede namespace
function zooKeeper(cage: AnimalCage);
namespace zooKeeper {
    let workSchedule: string;
}
```

## 可以用**new**调用也可以直接调用的方法

### 应用示例

```
let w = widget(32, 16);
let y = new widget("sprocket");
// w and y are both widgets
w.sprock();
y.sprock();
```

### 类型声明

```
interface Widget {
    sprock(): void;
}

interface WidgetFactory {
    new(name: string): Widget;
    (width: number, height: number): Widget;
}

declare let widget: WidgetFactory;
```

## 全局或外部的未知代码库

### 应用示例

```
// Either
import x = require('zoo');
x.open();
// or
zoo.open();
```

## 类型声明

```
declare namespace zoo {
    function open(): void;
}

declare module "zoo" {
    export = zoo;
}
```

## 模块里的单一复杂对象

### 应用示例

```
// Super-chainable library for eagles
import Eagle = require('./eagle');

// Call directly
Eagle('bald').fly();

// Invoke with new
var eddie = new Eagle('Mille');

// Set properties
eddie.kind = 'golden';
```

## 类型声明

```
interface Eagle {  
  (kind: string): Eagle;  
  new (kind: string): Eagle;  
  
  kind: string;  
  fly(): void  
}  
  
declare var Eagle: Eagle;  
  
export = Eagle;
```

## 将模块做为函数

### 应用示例

```
// Common pattern for node modules (e.g. rimraf, debug, request, et  
import sayHello = require('say-hello');  
sayHello('Travis');
```

### 类型声明

```
declare module 'say-hello' {  
  function sayHello(name: string): void;  
  export = sayHello;  
}
```

## 回调函数

### 应用示例

```
addLater(3, 4, x => console.log('x = ' + x));
```

## 类型声明

```
// Note: 'void' return type is preferred here  
function addLater(x: number, y: number, (sum: number) => void): void
```

如果你想看其它模式的实现方式，请在[这里](#)留言！我们会尽可能地加到这里来。

## 介绍

**JSX**是一种嵌入式的类似XML的语法。它可以被转换成合法的JavaScript，尽管转换的语义是依据不同的实现而定的。JSX因React框架而流行，但是也被其它应用所使用。TypeScript支持内嵌，类型检查和将JSX直接编译为JavaScript。

## 基本用法

想要使用JSX必须做两件事：

1. 给文件一个 `.tsx` 扩展名
2. 启用 `jsx` 选项

TypeScript具有两种JSX模式：`preserve` 和 `react`。这些模式只在代码生成阶段起作用 - 类型检查并不受影响。在 `preserve` 模式下生成代码中会保留JSX以供后续的转换操作使用（比如：[Babel](#)）。另外，输出文件会带有 `.jsx` 扩展名。

`react` 模式会生成 `React.createElement`，在使用前不需要再进行转换操作了，输出文件的扩展名为 `.js`。

模式	输入	输出	输出文件扩展名
<code>preserve</code>	<code>&lt;div /&gt;</code>	<code>&lt;div /&gt;</code>	<code>.jsx</code>
<code>react</code>	<code>&lt;div /&gt;</code>	<code>React.createElement("div")</code>	<code>.js</code>

你可以通过在命令行里使用 `--jsx` 标记或`tsconfig.json`里的选项来指定模式。

注意：`React` 标识符是写死的硬代码，所以你必须保证`React`（大写的`R`）是可用的。*Note: The identifier `React` is hard-coded, so you must make `React` available with an uppercase `R`.*

## as 操作符

回想一下怎么写类型断言：

```
var foo = <foo>bar;
```

这里我们断言 `bar` 变量是 `foo` 类型的。因为 TypeScript 也使用尖括号来表示类型断言，JSX 的语法带来了解析的困难。因此，TypeScript 在 `.tsx` 文件里禁用了使用尖括号的类型断言。

为了弥补 `.tsx` 里的这个功能，新加入了一个类型断言符号：`as`。上面的例子可以很容易地使用 `as` 操作符改写：

```
var foo = bar as foo;
```

`as` 操作符在 `.ts` 和 `.tsx` 里都可用，并且与其它类型断言行为是等价的。

## 类型检查

为了理解 JSX 的类型检查，你必须首先理解固有元素与基于值的元素之间的区别。假设有这样一个 JSX 表达式 `<expr />`，`expr` 可能引用环境自带的某些东西（比如，在 DOM 环境里的 `div` 或 `span`）或者是你自定义的组件。这是非常重要的，原因有如下两点：

1. 对于 React，固有元素会生成字符串（`React.createElement("div")`），然而由你自定义的组件却不会生成（`React.createElement(MyComponent)`）。
2. 传入 JSX 元素里的属性类型的查找方式不同。固有元素属性本身就支持，然而自定义的组件会自己去指定它们具有哪个属性。

TypeScript 使用与 React 相同的规范来区别它们。固有元素总是以一个小写字母开头，基于值的元素总是以一个大写字母开头。

## 固有元素

固有元素使用特殊的接口 `JSX.IntrinsicElements` 来查找。默认地，如果这个接口没有指定，会全部通过，不对固有元素进行类型检查。然而，如果接口存在，那么固有元素的名字需要在 `JSX.IntrinsicElements` 接口的属性里查找。例如：



```
declare namespace JSX {  
  interface IntrinsicElements {  
    foo: any  
  }  
}  
  
<foo />; // 正确  
<bar />; // 错误
```

在上例中，`<foo />` 没有问题，但是 `<bar />` 会报错，因为它没在 `JSX.IntrinsicElements` 里指定。

注意：你也可以在 `JSX.IntrinsicElements` 上指定一个用来捕获所有字符串索引：

```
declare namespace JSX {  
  interface IntrinsicElements {  
    [elemName: string]: any;  
  }  
}
```

## 基于值的元素

基于值的元素会简单的在它所在的作用域里按标识符查找。

```
import MyComponent from "./myComponent";  
  
<MyComponent />; // 正确  
<SomeOtherComponent />; // 错误
```

可以限制基于值的元素的类型。然而，为了这么做我们需要引入两个新的术语：元素类的类型和元素实例的类型。

现在有 `<Expr />`，元素类的类型为 `Expr` 的类型。所以在上面的例子里，如果 `MyComponent` 是ES6的类，那么它的类类型就是这个类。如果 `MyComponent` 是个工厂函数，类类型为这个函数。

一旦建立起了类类型，实例类型就确定了，为类类型调用签名的返回值与构造签名的联合类型。再次说明，在ES6类的情况下，实例类型为这个类的实例的类型，并且如果是工厂函数，实例类型为这个函数返回值类型。

```
class MyComponent {
  render() {}
}

// 使用构造签名
var myComponent = new MyComponent();

// 元素类的类型 => MyComponent
// 元素实例的类型 => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// 使用调用签名
var myComponent = MyFactoryFunction();

// 元素类的类型 => FactoryFunction
// 元素实例的类型 => { render: () => void }
```

元素的实例类型很有趣，因为它必须赋值给 `JSX.ElementClass` 或抛出一个错误。默认的 `JSX.ElementClass` 为 `{}`，但是它可以被扩展用来限制JSX的类型以符合相应的接口。

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // 正确
<MyFactoryFunction />; // 正确

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // 错误
<NotAValidFactoryFunction />; // 错误
```

## 属性类型检查

属性类型检查的第一步是确定元素属性类型。这在固有元素和基于值的元素之间稍有不同。

对于固有元素，这是 `JSX.IntrinsicElements` 属性的类型。

```
declare namespace JSX {  
  interface IntrinsicElements {  
    foo: { bar?: boolean }  
  }  
}  
  
// `foo`的元素属性类型为`{bar?: boolean}`  
<foo bar />;
```

对于基于值的元素，就稍微复杂些。它取决于先前确定的在元素实例类型上的某个属性的类型。至于该使用哪个属性来确定类型取决于 `JSX.ElementAttributesProperty`。它应该使用单一的属性来定义。这个属性名之后会被使用。

```
declare namespace JSX {  
  interface ElementAttributesProperty {  
    props; // 指定用来使用的属性名  
  }  
}  
  
class MyComponent {  
  // 在元素实例类型上指定属性  
  props: {  
    foo?: string;  
  }  
}  
  
// `MyComponent`的元素属性类型为`{foo?: string}`  
<MyComponent foo="bar" />
```

元素属性类型用于的JSX里进行属性的类型检查。支持可选属性和必须属性。

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number }
  }
}

<foo requiredProp="bar" />; // 正确
<foo requiredProp="bar" optionalProp={0} />; // 正确
<foo />; // 错误, 缺少 requiredProp
<foo requiredProp={0} />; // 错误, requiredProp 应该是字符串
<foo requiredProp="bar" unknownProp />; // 错误, unknownProp 不存在
<foo requiredProp="bar" some-unknown-prop />; // 正确, `some-unknown-prop` 是合法的JS标识符
```

注意：如果一个属性名不是个合法的JS标识符（像 `data-*` 属性），并且它没出现在元素属性类型里时不会当做一个错误。

延展操作符也可以使用：

```
var props = { requiredProp: 'bar' };
; // 正确

var badProps = {};
; // 错误
```

## JSX结果类型

默认地JSX表达式结果的类型为 `any`。你可以自定义这个类型，通过指定 `JSX.Element` 接口。然而，不能够从接口里检索元素，属性或JSX的子元素的类型信息。它是一个黑盒。

## 嵌入的表达式

JSX允许你使用 `{ }` 标签来内嵌表达式。

```
var a =  
  
  [['foo', 'bar'].map(i => {i / 2})]
```

上面的代码产生一个错误，因为你不能用数字来除以一个字符串。输出如下，若你使用了 `preserve` 选项：

```
var a =  
  
  [['foo', 'bar'].map(function (i) { return {i / 2}; })]
```

## React整合

要想一起使用JSX和React，你应该使用[React类型定义](#)。这些类型声明定义了 `JSX` 合适命名空间来使用React。

```
/// <reference path="react.d.ts" />  
  
interface Props {  
  foo: string;  
}  
  
class MyComponent extends React.Component<Props, {}> {  
  render() {  
    return <span>{this.props.foo}</span>  
  }  
}  
  
<MyComponent foo="bar" />; // 正确  
<MyComponent foo={0} />; // 错误
```

## 介绍

随着TypeScript和ES6里引入了类，现在在一些场景下我们会需要额外的特性,用来支持标注或修改类及其成员。Decorators提供了一种在类的声明和成员上使用元编程语法添加标注的方式。Javascript里的Decorators目前处在[建议征集的第一阶段](#)，在TypeScript里做为实验性特性已经提供了支持。

**注意** Decorators是实验性的特性，在未来的版本中可能会发生改变。

若要启用实验性的decorator，你必须启用 `experimentalDecorators` 编译器选项，在命令行中或在 `tsconfig.json`：

命令行:

```
tsc --target ES5 --experimentalDecorators
```

`tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

## Decorators (后文译作装饰器)

装饰器是一种特殊类型的声明，它能够被附加到[类声明](#)，[方法](#)，[访问符](#)，[属性](#)，或[参数](#)上。装饰器利用 `@expression` 这种方式，`expression` 求值后必须为一个函数，它使用被装饰的声明信息在运行时被调用。

例如，有一个 `@sealed` 装饰器，我们会这样定义 `sealed` 函数：

```
function sealed(target) {  
    // do something with "target" ...  
}
```

注意 下面[类装饰器](#)小节里有一个更加详细的例子。

## 装饰器工厂

如果我们想自定义装饰器是如何作用于声明的，我们得写一个装饰器工厂函数。装饰器工厂就是一个简单的函数，它返回一个表达式，以供装饰器在运行时调用。

我们可以通过下面的方式来写一个装饰器工厂

```
function color(value: string) { // 这是一个装饰器工厂  
    return function (target) { // 这是装饰器  
        // do something with "target" and "value"...  
    }  
}
```

注意 下面[方法装饰器](#)小节里有一个更加详细的例子。

## 装饰器组合

多个装饰器可以同时应用到一个声明上，就像下面的示例：

- 写在同一行上：

```
@f @g x
```

- 写在多行上：

```
@f  
@g  
x
```



当多个装饰器应用于一个声明上，它们求值方式与[复合函数](#)相似。在这个模型下，当复合 $f$ 和 $g$ 时，复合的结果 $(f \circ g)(x)$ 等同于 $f(g(x))$ 。

同样的，在TypeScript里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：

1. 由上至下依次对装饰器表达式求值。
2. 求值的结果会被当作函数，由下至上依次调用。

如果我们使用[装饰器工厂](#)的话，可以通过下面的例子来观察它们求值的顺序：

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

在控制台里会打印出如下结果：

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

## 装饰器求值

类中不同声明上的装饰器将按以下规定的顺序应用：

1. 参数装饰器，其次是方法，访问符，或属性装饰器应用到每个实例成员。
2. 参数装饰器，其次是方法，访问符，或属性装饰器应用到每个静态成员。
3. 参数装饰器应用到构造函数。
4. 类装饰器应用到类。

## 类装饰器

类装饰器在类声明之前被声明（紧贴着类声明）。类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。类装饰器不能用在声明文件中( `.d.ts` )，也不能用在任何外部上下文中（比如 `declare` 的类）。

类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。

如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。

注意 如果你要返回一个新的构造函数，你必须注意处理好原来的原型链。在运行时的装饰器调用逻辑中不会为你做这些。

下面是使用类装饰器( `@sealed` )的例子，应用到 `Greeter` 类：

```
@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

我们可以这样定义 `@sealed` 装饰器

```
function sealed(constructor: Function) {  
    Object.seal(constructor);  
    Object.seal(constructor.prototype);  
}
```

当 `@sealed` 被执行的时候，它将密封此类的构造函数和原型。(注：参见 [Object.seal](#))

## 方法装饰器

方法装饰器声明在一个方法的声明之前（紧贴着方法声明）。它会被应用到方法的属性描述符上，可以用来监视，修改或者替换方法定义。方法装饰器不能用在声明文件（`.d.ts`），重载或者任何外部上下文（比如 `declare` 的类）中。

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的属性描述符。

注意 如果代码输出目标版本小于 `ES5`，*Property Descriptor* 将会是 `undefined`。

如果方法装饰器返回一个值，它会被用作方法的属性描述符。

注意 如果代码输出目标版本小于 `ES5` 返回值会被忽略。

下面是一个方法装饰器（`@enumerable`）的例子，应用于 `Greeter` 类的方法上：

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

我们可以用下面的函数声明来定义 `@enumerable` 装饰器：

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
    descriptor.enumerable = value;
  };
}
```

这里的 `@enumerable(false)` 是一个装饰器工厂。当装饰器 `@enumerable(false)` 被调用时，它会修改属性描述符的 `enumerable` 属性。

## 访问符装饰器

访问符装饰器声明在一个访问符的声明之前（紧贴着访问符声明）。访问符装饰器应用于访问符的属性描述符并且可以用来监视，修改或替换一个访问符的定义。访问符装饰器不能用在声明文件中（`.d.ts`），或者任何外部上下文（比如 `declare` 的类）里。

**注意** TypeScript不允许同时装饰一个成员的 `get` 和 `set` 访问符。相反，所有装饰的成员必须被应用到文档顺序指定的第一个访问符。这是因为，装饰器应用于一个属性描述符，它联合了 `get` 和 `set` 访问符，而不是分开声明的。

访问符装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。

2. 成员的名字。
3. 成员的属性描述符。

注意 如果代码输出目标版本小于 `ES5`，*Property Descriptor* 将会是 `undefined`。

如果访问符装饰器返回一个值，它会被用作方法的属性描述符。

注意 如果代码输出目标版本小于 `ES5` 返回值会被忽略。

下面是使用了访问符装饰器 (`@configurable`) 的例子，应用于 `Point` 类的成员上：

```
class Point {
  private _x: number;
  private _y: number;
  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }

  @configurable(false)
  get x() { return this._x; }

  @configurable(false)
  get y() { return this._y; }
}
```

我们可以通过如下函数声明来定义 `@configurable` 装饰器：

```
function configurable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
    descriptor.configurable = value;
  };
}
```

## 属性装饰器

属性装饰器声明在一个属性声明之前（紧贴着属性声明）。属性装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如 `declare` 的类）里。

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。

**注意** 属性描述符不会做为参数传入属性装饰器，这与TypeScript是如何初始化属性装饰器的有关。因为目前没有办法在定义一个原型对象的成员时描述一个实例属性，并且没办法监视或修改一个属性的初始化方法。因此，属性描述符只能用来监视类中是否声明了某个名字的属性。

如果属性装饰器返回一个值，它会被用作方法的属性描述符。

**注意** 如果代码输出目标版本小于 `ES5`，返回值会被忽略。

如果访问符装饰器返回一个值，它会被用作方法的属性描述符。

我们可以用它来记录这个属性的元数据，如下例所示：

```
class Greeter {
  @format("Hello, %s")
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    let formatString = getFormat(this, "greeting");
    return formatString.replace("%s", this.greeting);
  }
}
```

然后定义 `@format` 装饰器和 `getFormat` 函数：

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
    return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
    return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

这个 `@format("Hello, %s")` 装饰器是个 [装饰器工厂](#)。当 `@format("Hello, %s")` 被调用时，它添加一条这个属性的元数据，通过 `reflect-metadata` 库里的 `Reflect.metadata` 函数。当 `getFormat` 被调用时，它读取格式的元数据。

**注意** 这个例子需要使用 `reflect-metadata` 库。查看[元数据了解](#) `reflect-metadata` 库更详细的信息。

## 参数装饰器

参数装饰器声明在一个参数声明之前（紧贴着参数声明）。参数装饰器应用于类构造函数或方法声明。参数装饰器不能用在声明文件（`.d.ts`），重载或其它外部上下文（比如 `declare` 的类）里。

参数装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 参数在函数参数列表中的索引。

**注意** 参数装饰器只能用来监视一个方法的参数是否被传入。

参数装饰器的返回值会被忽略。

下例定义了参数装饰器（`@required`）并应用于 `Greeter` 类方法的一个参数：

```
class Greeter {  
  greeting: string;  
  
  constructor(message: string) {  
    this.greeting = message;  
  }  
  
  @validate  
  greet(@required name: string) {  
    return "Hello " + name + ", " + this.greeting;  
  }  
}
```

然后我们使用下面的函数定义 `@required` 和 `@validate` 装饰器：



```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex: number): void {
    let existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target);
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target);
}

function validate(target: any, propertyName: string, descriptor: TypedPropertyDescriptor): void {
    let method = descriptor.value;
    descriptor.value = function () {
        let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target);
        if (requiredParameters) {
            for (let parameterIndex of requiredParameters) {
                if (parameterIndex >= arguments.length || arguments[parameterIndex] === undefined) {
                    throw new Error("Missing required argument.");
                }
            }
        }
        return method.apply(this, arguments);
    };
}

// 使用示例
@required
function greet(name: string): void {
    console.log(`Hello, ${name}!`);
}

// 调用
greet("John"); // 正确
greet(); // 错误: Missing required argument.
```

`@required` 装饰器添加了元数据实体把参数标记为必须的。`@validate` 装饰器把 `greet` 方法包裹在一个函数里在调用原先的函数前验证函数参数。

注意 这个例子使用了 `reflect-metadata` 库。查看[元数据](#)了解 `reflect-metadata` 库的更多信息。

## 元数据

一些例子使用了 `reflect-metadata` 库来支持实验性的 [metadata API](#)。这个库还不是ECMAScript (JavaScript)标准的一部分。然而，当装饰器被ECMAScript官方标准采纳后，这些扩展也将被推荐给ECMAScript以采纳。

你可以通过npm安装这个库：

```
npm i reflect-metadata --save
```

TypeScript支持为带有装饰器的声明生成元数据。你需要在命令行或 `tsconfig.json` 里启用 `emitDecoratorMetadata` 编译器选项。

### Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

### tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

当启用后，只要 `reflect-metadata` 库被引入了，设计阶段额外的信息可以在运行时使用。

如下例所示：

```
import "reflect-metadata";

class Point {
  x: number;
  y: number;
}

class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
  set p0(value: Point) { this._p0 = value; }
  get p0() { return this._p0; }

  @validate
  set p1(value: Point) { this._p1 = value; }
  get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  let set = descriptor.set;
  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);
    if (!(value instanceof type)) {
      throw new TypeError("Invalid type.");
    }
  };
}

export { validate, Point, Line };
```

TypeScript编译器可以通过 `@Reflect.metadata` 装饰器注入设计阶段的类型信息。你可以认为它相当于下面的TypeScript：

```
class Line {  
    private _p0: Point;  
    private _p1: Point;  
  
    @validate  
    @Reflect.metadata("design:type", Point)  
    set p0(value: Point) { this._p0 = value; }  
    get p0() { return this._p0; }  
  
    @validate  
    @Reflect.metadata("design:type", Point)  
    set p1(value: Point) { this._p1 = value; }  
    get p1() { return this._p1; }  
}
```

注意 装饰器元数据是个实验性的特性并且可能在以后的版本中发生破坏性的改变 (breaking changes)。

## 介绍

除了传统的面向对象继承方式，还流行一种通过可重用组件创建类的方式，就是联合另一个简单类的代码。你可能在Scala等语言里对mixins及其特性已经很熟悉了，但它在JavaScript中也是很流行的。

## 混入示例

下面的代码演示了如何在TypeScript里使用混入。后面我们还会解释这段代码是怎么工作的。

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

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

class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + this), 1000)
    }
}
```

```

    interact() {
        this.activate();
    }

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

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

////////////////////////////////////
// In your runtime library somewhere
////////////////////////////////////

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

```

## 理解这个例子

代码里首先定义了两个类，它们将做为mixins。可以看到每个类都只定义了一个特定的行为或功能。稍后我们使用它们来创建一个新类，同时具有这两种功能。

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

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

下面创建一个类，结合了这两个mixins。下面来看一下具体是怎么操作的：

```
class SmartObject implements Disposable, Activatable {
```

首先应该注意到的是，没使用 `extends` 而是使用 `implements`。把类当成了接口，仅使用`Disposable`和`Activatable`的类型而非其实现。这意味着我们需要在类里面实现接口。但是这是我们在用mixin时想避免的。

我们可以这么做来达到目的，为将要mixin进来的属性方法创建出占位属性。这告诉编译器这些成员在运行时是可用的。这样就能使用mixin带来的便利，虽说需要提前定义一些占位属性。

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

最后，把mixins混入定义的类，完成全部实现部分。

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

最后，创建这个帮助函数，帮我们做混入操作。它会遍历mixins上的所有属性，并复制到目标上去，把之前的占位属性替换成真正的实现代码。

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



三斜线指令是包含单个XML标签的单行注释。注释的内容会做为编译器指令使用。

三斜线指令仅可放在包含它的文件的最顶端。一个三斜线指令的前面只能出现单行或多行注释，这包括其它的三斜线指令。如果它们出现在一个语句或声明之后，那么它们会被当做普通的单行注释，并且不具有特殊的涵义。

```
///
```

`/// 指令是三斜线指令中最常见的一种。它用于声明文件间的依赖。`

三斜线引用告诉编译器在编译过程中要引入的额外的文件。

当使用 `--out` 或 `--outFile` 时，它也可以做为调整输出内容顺序的一种方法。文件在输出文件内容中的位置与经过预处理后的输入顺序一致。

## 预处理输入文件

编译器会对输入文件进行预处理来解析所有三斜线引用指令。在这个过程中，额外的文件会加到编译过程中。

这个过程会以一些根文件开始；它们是在命令行中指定的文件或是在 `tsconfig.json` 中的 `"files"` 列表里的文件。这些根文件按指定的顺序进行预处理。在一个文件被加入列表前，它包含的所有三斜线引用都要被处理，还有它们包含的目标。三斜线引用以它们在文件里出现的顺序，使用深度优先的方式解析。

一个三斜线引用路径是相对于包含它的文件的，如果不是根文件。

## 错误

引用不存在的文件会报错。一个文件用三斜线指令引用自己会报错。

## 使用 `--noResolve`

如果指定了 `--noResolve` 编译选项，三斜线引用会被忽略；它们不会增加新文件，也不会改变给定文件的顺序。

## `///`

这个指令把一个文件标记成默认库。你会在 `lib.d.ts` 文件和它不同的变体的顶端看到这个注释。

这个指令告诉编译器在编译过程中不要包含这个默认库（比如，`lib.d.ts`）。这与在命令行上使用 `--noLib` 相似。

还要注意，当传递了 `--skipDefaultLibCheck` 时，编译器只会忽略检查带有 `/// 的文件。`

## `///`

默认情况下生成的AMD模块都是匿名的。但是，当一些工具需要处理生成的模块时会产生问题，比如 `r.js`。

`amd-module` 指令允许给编译器传入一个可选的模块名：

### `amdModule.ts`

```
///
```

这会将 `NamedModule` 传入到AMD `define` 函数里：

### `amdModule.js`

```
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

## ///**<amd-dependency />**

注意：这个指令被废弃了。使用 `import "moduleName";` 语句代替。

`///<amd-dependency path="x" />` 告诉编译器有一个非TypeScript模块依赖需要被注入，做为目标模块 `require` 调用的一部分。

`amd-dependency` 指令也可以带一个可选的 `name` 属性；它允许我们为 `amd-dependency` 传入一个可选名字：

```
///<amd-dependency path="legacy/moduleA" name="moduleA"/>  
declare var moduleA:MyType  
moduleA.callStuff()
```

生成的JavaScript代码：

```
define(["require", "exports", "legacy/moduleA"], function (require,  
    moduleA.callStuff()  
});
```

## 概述

如果一个目录下存在一个 `tsconfig.json` 文件，那么它意味着这个目录是 TypeScript 项目的根目录。`tsconfig.json` 文件中指定了用来编译这个项目的根文件和编译选项。一个项目可以通过以下方式之一来编译：

## 使用 `tsconfig.json`

- 不带任何输入文件的情况下调用 `tsc`，编译器会从当前目录开始去找 `tsconfig.json` 文件，逐级向上搜索父目录。
- 不带任何输入文件的情况下调用 `tsc`，且使用命令行参数 `--project`（或 `-p`）指定一个包含 `tsconfig.json` 文件的目录。

当命令行上指定了输入文件时，`tsconfig.json` 文件会被忽略。

## 示例

`tsconfig.json` 示例文件：

- 使用 `"files"` 属性

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "emitter.ts",
    "program.ts",
    "commandLineParser.ts",
    "tsc.ts",
    "diagnosticInformationMap.generated.ts"
  ]
}
```

- 使用 `"exclude"` 属性

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

## 细节

`"compilerOptions"` 可以被忽略，这时编译器会使用默认值。在这里查看完整的[编译器选项列表](#)。

如果 `tsconfig.json` 没有提供 `"files"` 属性，编译器会默认包含当前目录及子目录下的所有 TypeScript 文件（`*.ts` 或 `*.tsx`）。如果提供了 `"files"` 属性值，只有指定的文件会被编译。

如果指定了 `"exclude"` 选项，编译器会包含当前目录及子目录下的所有 TypeScript 文件（`*.ts` 或 `*.tsx`），不包括这些指定要排除的文件。

`"files"` 选项不能与 `"exclude"` 选项同时使用。如果同时指定了两个选项的话，只有 `"files"` 会生效。

所有被 `"files"` 属性里的文件所引用的文件同样会被包含进来。就好比，`A.ts` 引用了 `B.ts`，因此 `B.ts` 不能被排除，除非引用它的 `A.ts` 在 `"exclude"` 列表中。

`tsconfig.json` 可以是个空文件，那么编译器则使用默认编译选项，编译当前目录及其子目录下的所有文件。

命令行上提供的编译选项会覆盖 `tsconfig.json` 文件中的对应选项。

## compileOnSave

在最顶层设置 `compileOnSave` 标记，可以让IDE在保存文件的时候根据 `tsconfig.json` 重新生成文件。

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "noImplicitAny" : true
  }
}
```

要想支持这个特性需要Visual Studio 2015， TypeScript1.8.4以上并且安装`atom-typescript`插件。

## 模式

到这里查看模式: <http://json.schemastore.org/tsconfig>.

TypeScript编译器处理Node模块名时使用的是[Node.js模块解析算法](#)。TypeScript也可以同时加载与npm包绑在一起的类型声明文件。编译通过下面的规则来查找 "foo" 模块的类型信息：

1. 尝试加载相应代码包目录下 `package.json` 文件  
( `node_modules/foo/` ) 。

如果存在，从 "typings" 字段里读取类型文件的路径。比如，在下面的 `package.json` 里，编译器会认为类型文件位于 `node_modules/foo/lib/foo.d.ts` 。

```
{
  "name": "foo",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/foo.js",
  "typings": "./lib/foo.d.ts"
}
```

1. 尝试加载在相应代码包目录下的名字为 `index.d.ts` 的文件  
( `node_modules/foo/` ) - 这个文件应该包含了这个代码包的类型信息。

解析模块的详细算法可以在[这里](#)找到。

## 你的定义文件应该

- 是 `.d.ts` 文件
- 写做外部模块
- 不包含 `/// 引用`

基本的原理是类型文件不能引入新的可编译代码；否则真正的实现文件就可能会在编译时被重盖。另外，加载类型信息不应该污染全局空间，当从同一个库的不同版本中引入潜在冲突的实体的时候。



## 编译选项

选项	类型	默认值	
<code>--allowJs</code>	boolean	true	允许编译 JavaScript 文件。
<code>--allowSyntheticDefaultImports</code>	boolean	(module === "system")	允许从没有默认导出的模块中，使用默认导入。这通常用于在 TypeScript 中模拟 CommonJS 模块。
<code>--allowUnreachableCode</code>	boolean	false	不报告不可达代码的错误。
<code>--allowUnusedLabels</code>	boolean	false	不报告未使用的标签的错误。
<code>--charset</code>	string	"utf8"	输入文件的字符集。
<code>--declaration</code> <code>-d</code>	boolean	false	生成 .d.ts 文件。
<code>--diagnostics</code>	boolean	false	显示编译时的诊断信息。
<code>--emitBOM</code>	boolean	false	在输出文件的开头添加 UTF-8 BOM。
<code>--emitDecoratorMetadata [1]</code>	boolean	false	给源文件添加元数据信息。
<code>--experimentalDecorators [1]</code>	boolean	false	实验性支持类装饰器。
<code>--forceConsistentCasingInFileNames</code>	boolean	false	不允许在文件名中使用不一致的大小写。
<code>--help</code> <code>-h</code>			打印帮助信息。
<code>--init</code>			初始化一个 tsconfig.json 文件。
<code>--inlineSourceMap</code>	boolean	false	生成内联的 source map。
<code>--inlineSources</code>	boolean	false	在输出文件中包含源文件的文本内容。
<code>--isolatedModules</code>	boolean	false	无条件编译，不允许使用任何导入或导出。
<code>--jsx</code>	string	"Preserve"	在 .ts 文件中处理 JSX 元素。

<code>--listFiles</code>	boolean	false	编译时列出所有文件
<code>--locale</code>	string	<i>(platform specific)</i>	显示本地化信息 en-us
<code>--mapRoot</code>	string	null	为调测器提供源文件的路径 当 .ts 文件被编译成 .js 文件时，不同平台有不同的路径分隔符。指定到 source 文件的路径，以便找到它的位置。
<code>--module</code> <code>-m</code>	string	<code>(target === "ES6" ? "ES6" : "CommonJS")</code>	指定模块化的代码： 'umd' 有 'an' 起使用，能使
<code>--moduleResolution</code>	string	"Classic"	决定如何解析模块（Node 默认）
<code>--newLine</code>	string	<i>(platform specific)</i>	当生成 .js 文件时，使用什么换行符： '\n' 或 '\r\n'
<code>--noEmit</code>	boolean	false	不生成 .js 文件
<code>--noEmitHelpers</code>	boolean	false	不在 .js 文件中包含帮助函数
<code>--noEmitOnError</code>	boolean	false	报错时不生成 .js 文件
<code>--noFallthroughCasesInSwitch</code>	boolean	false	报告 switch 语句中的未匹配的 case（即 TypeScript 2.7 引入的）
<code>--noImplicitAny</code>	boolean	false	在表时报告未声明的变量
<code>--noImplicitReturns</code>	boolean	false	不是函数时报告未返回的值
<code>--noImplicitUseStrict</code>	boolean	false	模块时不添加 "use strict"
<code>--noLib</code>	boolean	false	不包含 lib.d.ts 文件
<code>--noResolve</code>	boolean	false	不把 .d.ts 文件放入编译
<del><code>--out</code></del>	string	null	弃用
<code>--outDir</code>	string	null	重定向输出目录

<code>--outFile</code>	<code>string</code>	<code>null</code>	将输出文件的顺序和输入文件的顺序一致。
<code>--preserveConstEnums</code>	<code>boolean</code>	<code>false</code>	保留看 <a href="#">const 枚举</a> 的情况。
<code>--pretty [1]</code>	<code>boolean</code>	<code>false</code>	给错误信息上下加空格。
<code>--project</code> <code>-p</code>	<code>string</code>	<code>null</code>	编译该包。管理更多包。
<code>--reactNamespace</code>	<code>string</code>	<code>"React"</code>	当前文件调用 <code>ReactDOM</code> 时，使用 <code>React</code> 命名空间。
<code>--removeComments</code>	<code>boolean</code>	<code>false</code>	删除注释。
<code>--rootDir</code>	<code>string</code>	<i>(common root directory is computed from the list of input files)</i>	仅用于 <code>outDir</code> 。
<code>--skipDefaultLibCheck</code>	<code>boolean</code>	<code>false</code>	
<code>--sourceMap</code>	<code>boolean</code>	<code>false</code>	生成源地图。
<code>--sourceRoot</code>	<code>string</code>	<code>null</code>	指定源地图的根目录。在运行时会使用。
<code>--stripInternal [1]</code>	<code>boolean</code>	<code>false</code>	不对 <code>__internal__</code> 注解的变量进行类型检查。
<code>--suppressExcessPropertyErrors [1]</code>	<code>boolean</code>	<code>false</code>	阻止对多余属性的错误检查。
<code>--suppressImplicitAnyIndexErrors</code>	<code>boolean</code>	<code>false</code>	阻止对 <code>any</code> 类型的索引的错误检查。 <a href="#">#123</a>
<code>--target</code> <code>-t</code>	<code>string</code>	<code>"ES5"</code>	指定目标环境 (默认)， 如 <code>ES5</code> 、 <code>ES6</code> 、 <code>ES2015</code> 、 <code>ES2016</code> 、 <code>ES2017</code> 、 <code>ES2018</code> 、 <code>ES2019</code> 、 <code>ES2020</code> 、 <code>ES2021</code> 、 <code>ES2022</code> 、 <code>ES2023</code> 、 <code>ES2024</code> 、 <code>ES2025</code> 、 <code>ES2026</code> 、 <code>ES2027</code> 、 <code>ES2028</code> 、 <code>ES2029</code> 、 <code>ES2030</code> 、 <code>ES2031</code> 、 <code>ES2032</code> 、 <code>ES2033</code> 、 <code>ES2034</code> 、 <code>ES2035</code> 、 <code>ES2036</code> 、 <code>ES2037</code> 、 <code>ES2038</code> 、 <code>ES2039</code> 、 <code>ES2040</code> 、 <code>ES2041</code> 、 <code>ES2042</code> 、 <code>ES2043</code> 、 <code>ES2044</code> 、 <code>ES2045</code> 、 <code>ES2046</code> 、 <code>ES2047</code> 、 <code>ES2048</code> 、 <code>ES2049</code> 、 <code>ES2050</code> 、 <code>ES2051</code> 、 <code>ES2052</code> 、 <code>ES2053</code> 、 <code>ES2054</code> 、 <code>ES2055</code> 、 <code>ES2056</code> 、 <code>ES2057</code> 、 <code>ES2058</code> 、 <code>ES2059</code> 、 <code>ES2060</code> 、 <code>ES2061</code> 、 <code>ES2062</code> 、 <code>ES2063</code> 、 <code>ES2064</code> 、 <code>ES2065</code> 、 <code>ES2066</code> 、 <code>ES2067</code> 、 <code>ES2068</code> 、 <code>ES2069</code> 、 <code>ES2070</code> 、 <code>ES2071</code> 、 <code>ES2072</code> 、 <code>ES2073</code> 、 <code>ES2074</code> 、 <code>ES2075</code> 、 <code>ES2076</code> 、 <code>ES2077</code> 、 <code>ES2078</code> 、 <code>ES2079</code> 、 <code>ES2080</code> 、 <code>ES2081</code> 、 <code>ES2082</code> 、 <code>ES2083</code> 、 <code>ES2084</code> 、 <code>ES2085</code> 、 <code>ES2086</code> 、 <code>ES2087</code> 、 <code>ES2088</code> 、 <code>ES2089</code> 、 <code>ES2090</code> 、 <code>ES2091</code> 、 <code>ES2092</code> 、 <code>ES2093</code> 、 <code>ES2094</code> 、 <code>ES2095</code> 、 <code>ES2096</code> 、 <code>ES2097</code> 、 <code>ES2098</code> 、 <code>ES2099</code> 、 <code>ES2100</code> 、 <code>ES2101</code> 、 <code>ES2102</code> 、 <code>ES2103</code> 、 <code>ES2104</code> 、 <code>ES2105</code> 、 <code>ES2106</code> 、 <code>ES2107</code> 、 <code>ES2108</code> 、 <code>ES2109</code> 、 <code>ES2110</code> 、 <code>ES2111</code> 、 <code>ES2112</code> 、 <code>ES2113</code> 、 <code>ES2114</code> 、 <code>ES2115</code> 、 <code>ES2116</code> 、 <code>ES2117</code> 、 <code>ES2118</code> 、 <code>ES2119</code> 、 <code>ES2120</code> 、 <code>ES2121</code> 、 <code>ES2122</code> 、 <code>ES2123</code> 、 <code>ES2124</code> 、 <code>ES2125</code> 、 <code>ES2126</code> 、 <code>ES2127</code> 、 <code>ES2128</code> 、 <code>ES2129</code> 、 <code>ES2130</code> 、 <code>ES2131</code> 、 <code>ES2132</code> 、 <code>ES2133</code> 、 <code>ES2134</code> 、 <code>ES2135</code> 、 <code>ES2136</code> 、 <code>ES2137</code> 、 <code>ES2138</code> 、 <code>ES2139</code> 、 <code>ES2140</code> 、 <code>ES2141</code> 、 <code>ES2142</code> 、 <code>ES2143</code> 、 <code>ES2144</code> 、 <code>ES2145</code> 、 <code>ES2146</code> 、 <code>ES2147</code> 、 <code>ES2148</code> 、 <code>ES2149</code> 、 <code>ES2150</code> 、 <code>ES2151</code> 、 <code>ES2152</code> 、 <code>ES2153</code> 、 <code>ES2154</code> 、 <code>ES2155</code> 、 <code>ES2156</code> 、 <code>ES2157</code> 、 <code>ES2158</code> 、 <code>ES2159</code> 、 <code>ES2160</code> 、 <code>ES2161</code> 、 <code>ES2162</code> 、 <code>ES2163</code> 、 <code>ES2164</code> 、 <code>ES2165</code> 、 <code>ES2166</code> 、 <code>ES2167</code> 、 <code>ES2168</code> 、 <code>ES2169</code> 、 <code>ES2170</code> 、 <code>ES2171</code> 、 <code>ES2172</code> 、 <code>ES2173</code> 、 <code>ES2174</code> 、 <code>ES2175</code> 、 <code>ES2176</code> 、 <code>ES2177</code> 、 <code>ES2178</code> 、 <code>ES2179</code> 、 <code>ES2180</code> 、 <code>ES2181</code> 、 <code>ES2182</code> 、 <code>ES2183</code> 、 <code>ES2184</code> 、 <code>ES2185</code> 、 <code>ES2186</code> 、 <code>ES2187</code> 、 <code>ES2188</code> 、 <code>ES2189</code> 、 <code>ES2190</code> 、 <code>ES2191</code> 、 <code>ES2192</code> 、 <code>ES2193</code> 、 <code>ES2194</code> 、 <code>ES2195</code> 、 <code>ES2196</code> 、 <code>ES2197</code> 、 <code>ES2198</code> 、 <code>ES2199</code> 、 <code>ES2200</code> 、 <code>ES2201</code> 、 <code>ES2202</code> 、 <code>ES2203</code> 、 <code>ES2204</code> 、 <code>ES2205</code> 、 <code>ES2206</code> 、 <code>ES2207</code> 、 <code>ES2208</code> 、 <code>ES2209</code> 、 <code>ES2210</code> 、 <code>ES2211</code> 、 <code>ES2212</code> 、 <code>ES2213</code> 、 <code>ES2214</code> 、 <code>ES2215</code> 、 <code>ES2216</code> 、 <code>ES2217</code> 、 <code>ES2218</code> 、 <code>ES2219</code> 、 <code>ES2220</code> 、 <code>ES2221</code> 、 <code>ES2222</code> 、 <code>ES2223</code> 、 <code>ES2224</code> 、 <code>ES2225</code> 、 <code>ES2226</code> 、 <code>ES2227</code> 、 <code>ES2228</code> 、 <code>ES2229</code> 、 <code>ES2230</code> 、 <code>ES2231</code> 、 <code>ES2232</code> 、 <code>ES2233</code> 、 <code>ES2234</code> 、 <code>ES2235</code> 、 <code>ES2236</code> 、 <code>ES2237</code> 、 <code>ES2238</code> 、 <code>ES2239</code> 、 <code>ES2240</code> 、 <code>ES2241</code> 、 <code>ES2242</code> 、 <code>ES2243</code> 、 <code>ES2244</code> 、 <code>ES2245</code> 、 <code>ES2246</code> 、 <code>ES2247</code> 、 <code>ES2248</code> 、 <code>ES2249</code> 、 <code>ES2250</code> 、 <code>ES2251</code> 、 <code>ES2252</code> 、 <code>ES2253</code> 、 <code>ES2254</code> 、 <code>ES2255</code> 、 <code>ES2256</code> 、 <code>ES2257</code> 、 <code>ES2258</code> 、 <code>ES2259</code> 、 <code>ES2260</code> 、 <code>ES2261</code> 、 <code>ES2262</code> 、 <code>ES2263</code> 、 <code>ES2264</code> 、 <code>ES2265</code> 、 <code>ES2266</code> 、 <code>ES2267</code> 、 <code>ES2268</code> 、 <code>ES2269</code> 、 <code>ES2270</code> 、 <code>ES2271</code> 、 <code>ES2272</code> 、 <code>ES2273</code> 、 <code>ES2274</code> 、 <code>ES2275</code> 、 <code>ES2276</code> 、 <code>ES2277</code> 、 <code>ES2278</code> 、 <code>ES2279</code> 、 <code>ES2280</code> 、 <code>ES2281</code> 、 <code>ES2282</code> 、 <code>ES2283</code> 、 <code>ES2284</code> 、 <code>ES2285</code> 、 <code>ES2286</code> 、 <code>ES2287</code> 、 <code>ES2288</code> 、 <code>ES2289</code> 、 <code>ES2290</code> 、 <code>ES2291</code> 、 <code>ES2292</code> 、 <code>ES2293</code> 、 <code>ES2294</code> 、 <code>ES2295</code> 、 <code>ES2296</code> 、 <code>ES2297</code> 、 <code>ES2298</code> 、 <code>ES2299</code> 、 <code>ES2300</code> 、 <code>ES2301</code> 、 <code>ES2302</code> 、 <code>ES2303</code> 、 <code>ES2304</code> 、 <code>ES2305</code> 、 <code>ES2306</code> 、 <code>ES2307</code> 、 <code>ES2308</code> 、 <code>ES2309</code> 、 <code>ES2310</code> 、 <code>ES2311</code> 、 <code>ES2312</code> 、 <code>ES2313</code> 、 <code>ES2314</code> 、 <code>ES2315</code> 、 <code>ES2316</code> 、 <code>ES2317</code> 、 <code>ES2318</code> 、 <code>ES2319</code> 、 <code>ES2320</code> 、 <code>ES2321</code> 、 <code>ES2322</code> 、 <code>ES2323</code> 、 <code>ES2324</code> 、 <code>ES2325</code> 、 <code>ES2326</code> 、 <code>ES2327</code> 、 <code>ES2328</code> 、 <code>ES2329</code> 、 <code>ES2330</code> 、 <code>ES2331</code> 、 <code>ES2332</code> 、 <code>ES2333</code> 、 <code>ES2334</code> 、 <code>ES2335</code> 、 <code>ES2336</code> 、 <code>ES2337</code> 、 <code>ES2338</code> 、 <code>ES2339</code> 、 <code>ES2340</code> 、 <code>ES2341</code> 、 <code>ES2342</code> 、 <code>ES2343</code> 、 <code>ES2344</code> 、 <code>ES2345</code> 、 <code>ES2346</code> 、 <code>ES2347</code> 、 <code>ES2348</code> 、 <code>ES2349</code> 、 <code>ES2350</code> 、 <code>ES2351</code> 、 <code>ES2352</code> 、 <code>ES2353</code> 、 <code>ES2354</code> 、 <code>ES2355</code> 、 <code>ES2356</code> 、 <code>ES2357</code> 、 <code>ES2358</code> 、 <code>ES2359</code> 、 <code>ES2360</code> 、 <code>ES2361</code> 、 <code>ES2362</code> 、 <code>ES2363</code> 、 <code>ES2364</code> 、 <code>ES2365</code> 、 <code>ES2366</code> 、 <code>ES2367</code> 、 <code>ES2368</code> 、 <code>ES2369</code> 、 <code>ES2370</code> 、 <code>ES2371</code> 、 <code>ES2372</code> 、 <code>ES2373</code> 、 <code>ES2374</code> 、 <code>ES2375</code> 、 <code>ES2376</code> 、 <code>ES2377</code> 、 <code>ES2378</code> 、 <code>ES2379</code> 、 <code>ES2380</code> 、 <code>ES2381</code> 、 <code>ES2382</code> 、 <code>ES2383</code> 、 <code>ES2384</code> 、 <code>ES2385</code> 、 <code>ES2386</code> 、 <code>ES2387</code> 、 <code>ES2388</code> 、 <code>ES2389</code> 、 <code>ES2390</code> 、 <code>ES2391</code> 、 <code>ES2392</code> 、 <code>ES2393</code> 、 <code>ES2394</code> 、 <code>ES2395</code> 、 <code>ES2396</code> 、 <code>ES2397</code> 、 <code>ES2398</code> 、 <code>ES2399</code> 、 <code>ES2400</code> 、 <code>ES2401</code> 、 <code>ES2402</code> 、 <code>ES2403</code> 、 <code>ES2404</code> 、 <code>ES2405</code> 、 <code>ES2406</code> 、 <code>ES2407</code> 、 <code>ES2408</code> 、 <code>ES2409</code> 、 <code>ES2410</code> 、 <code>ES2411</code> 、 <code>ES2412</code> 、 <code>ES2413</code> 、 <code>ES2414</code> 、 <code>ES2415</code> 、 <code>ES2416</code> 、 <code>ES2417</code> 、 <code>ES2418</code> 、 <code>ES2419</code> 、 <code>ES2420</code> 、 <code>ES2421</code> 、 <code>ES2422</code> 、 <code>ES2423</code> 、 <code>ES2424</code> 、 <code>ES2425</code> 、 <code>ES2426</code> 、 <code>ES2427</code> 、 <code>ES2428</code> 、 <code>ES2429</code> 、 <code>ES2430</code> 、 <code>ES2431</code> 、 <code>ES2432</code> 、 <code>ES2433</code> 、 <code>ES2434</code> 、 <code>ES2435</code> 、 <code>ES2436</code> 、 <code>ES2437</code> 、 <code>ES2438</code> 、 <code>ES2439</code> 、 <code>ES2440</code> 、 <code>ES2441</code> 、 <code>ES2442</code> 、 <code>ES2443</code> 、 <code>ES2444</code> 、 <code>ES2445</code> 、 <code>ES2446</code> 、 <code>ES2447</code> 、 <code>ES2448</code> 、 <code>ES2449</code> 、 <code>ES2450</code> 、 <code>ES2451</code> 、 <code>ES2452</code> 、 <code>ES2453</code> 、 <code>ES2454</code> 、 <code>ES2455</code> 、 <code>ES2456</code> 、 <code>ES2457</code> 、 <code>ES2458</code> 、 <code>ES2459</code> 、 <code>ES2460</code> 、 <code>ES2461</code> 、 <code>ES2462</code> 、 <code>ES2463</code> 、 <code>ES2464</code> 、 <code>ES2465</code> 、 <code>ES2466</code> 、 <code>ES2467</code> 、 <code>ES2468</code> 、 <code>ES2469</code> 、 <code>ES2470</code> 、 <code>ES2471</code> 、 <code>ES2472</code> 、 <code>ES2473</code> 、 <code>ES2474</code> 、 <code>ES2475</code> 、 <code>ES2476</code> 、 <code>ES2477</code> 、 <code>ES2478</code> 、 <code>ES2479</code> 、 <code>ES2480</code> 、 <code>ES2481</code> 、 <code>ES2482</code> 、 <code>ES2483</code> 、 <code>ES2484</code> 、 <code>ES2485</code> 、 <code>ES2486</code> 、 <code>ES2487</code> 、 <code>ES2488</code> 、 <code>ES2489</code> 、 <code>ES2490</code> 、 <code>ES2491</code> 、 <code>ES2492</code> 、 <code>ES2493</code> 、 <code>ES2494</code> 、 <code>ES2495</code> 、 <code>ES2496</code> 、 <code>ES2497</code> 、 <code>ES2498</code> 、 <code>ES2499</code> 、 <code>ES2500</code> 、 <code>ES2501</code> 、 <code>ES2502</code> 、 <code>ES2503</code> 、 <code>ES2504</code> 、 <code>ES2505</code> 、 <code>ES2506</code> 、 <code>ES2507</code> 、 <code>ES2508</code> 、 <code>ES2509</code> 、 <code>ES2510</code> 、 <code>ES2511</code> 、 <code>ES2512</code> 、 <code>ES2513</code> 、 <code>ES2514</code> 、 <code>ES2515</code> 、 <code>ES2516</code> 、 <code>ES2517</code> 、 <code>ES2518</code> 、 <code>ES2519</code> 、 <code>ES2520</code> 、 <code>ES2521</code> 、 <code>ES2522</code> 、 <code>ES2523</code> 、 <code>ES2524</code> 、 <code>ES2525</code> 、 <code>ES2526</code> 、 <code>ES2527</code> 、 <code>ES2528</code> 、 <code>ES2529</code> 、 <code>ES2530</code> 、 <code>ES2531</code> 、 <code>ES2532</code> 、 <code>ES2533</code> 、 <code>ES2534</code> 、 <code>ES2535</code> 、 <code>ES2536</code> 、 <code>ES2537</code> 、 <code>ES2538</code> 、 <code>ES2539</code> 、 <code>ES2540</code> 、 <code>ES2541</code> 、 <code>ES2542</code> 、 <code>ES2543</code> 、 <code>ES2544</code> 、 <code>ES2545</code> 、 <code>ES2546</code> 、 <code>ES2547</code> 、 <code>ES2548</code> 、 <code>ES2549</code> 、 <code>ES2550</code> 、 <code>ES2551</code> 、 <code>ES2552</code> 、 <code>ES2553</code> 、 <code>ES2554</code> 、 <code>ES2555</code> 、 <code>ES2556</code> 、 <code>ES2557</code> 、 <code>ES2558</code> 、 <code>ES2559</code> 、 <code>ES2560</code> 、 <code>ES2561</code> 、 <code>ES2562</code> 、 <code>ES2563</code> 、 <code>ES2564</code> 、 <code>ES2565</code> 、 <code>ES2566</code> 、 <code>ES2567</code> 、 <code>ES2568</code> 、 <code>ES2569</code> 、 <code>ES2570</code> 、 <code>ES2571</code> 、 <code>ES2572</code> 、 <code>ES2573</code> 、 <code>ES2574</code> 、 <code>ES2575</code> 、 <code>ES2576</code> 、 <code>ES2577</code> 、 <code>ES2578</code> 、 <code>ES2579</code> 、 <code>ES2580</code> 、 <code>ES2581</code> 、 <code>ES2582</code> 、 <code>ES2583</code> 、 <code>ES2584</code> 、 <code>ES2585</code> 、 <code>ES2586</code> 、 <code>ES2587</code> 、 <code>ES2588</code> 、 <code>ES2589</code> 、 <code>ES2590</code> 、 <code>ES2591</code> 、 <code>ES2592</code> 、 <code>ES2593</code> 、 <code>ES2594</code> 、 <code>ES2595</code> 、 <code>ES2596</code> 、 <code>ES2597</code> 、 <code>ES2598</code> 、 <code>ES2599</code> 、 <code>ES2600</code> 、 <code>ES2601</code> 、 <code>ES2602</code> 、 <code>ES2603</code> 、 <code>ES2604</code> 、 <code>ES2605</code> 、 <code>ES2606</code> 、 <code>ES2607</code> 、 <code>ES2608</code> 、 <code>ES2609</code> 、 <code>ES2610</code> 、 <code>ES2611</code> 、 <code>ES2612</code> 、 <code>ES2613</code> 、 <code>ES2614</code> 、 <code>ES2615</code> 、 <code>ES2616</code> 、 <code>ES2617</code> 、 <code>ES2618</code> 、 <code>ES2619</code> 、 <code>ES2620</code> 、 <code>ES2621</code> 、 <code>ES2622</code> 、 <code>ES2623</code> 、 <code>ES2624</code> 、 <code>ES2625</code> 、 <code>ES2626</code> 、 <code>ES2627</code> 、 <code>ES2628</code> 、 <code>ES2629</code> 、 <code>ES2630</code> 、 <code>ES2631</code> 、 <code>ES2632</code> 、 <code>ES2633</code> 、 <code>ES2634</code> 、 <code>ES2635</code> 、 <code>ES2636</code> 、 <code>ES2637</code> 、 <code>ES2638</code> 、 <code>ES2639</code> 、 <code>ES2640</code> 、 <code>ES2641</code> 、 <code>ES2642</code> 、 <code>ES2643</code> 、 <code>ES2644</code> 、 <code>ES2645</code> 、 <code>ES2646</code> 、 <code>ES2647</code> 、 <code>ES2648</code> 、 <code>ES2649</code> 、 <code>ES2650</code> 、 <code>ES2651</code> 、 <code>ES2652</code> 、 <code>ES2653</code> 、 <code>ES2654</code> 、 <code>ES2655</code> 、 <code>ES2656</code> 、 <code>ES2657</code> 、 <code>ES2658</code> 、 <code>ES2659</code> 、 <code>ES2660</code> 、 <code>ES2661</code> 、 <code>ES2662</code> 、 <code>ES2663</code> 、 <code>ES2664</code> 、 <code>ES2665</code> 、 <code>ES2666</code> 、 <code>ES2667</code> 、 <code>ES2668</code> 、 <code>ES2669</code> 、 <code>ES2670</code> 、 <code>ES2671</code> 、 <code>ES2672</code> 、 <code>ES2673</code> 、 <code>ES2674</code> 、 <code>ES2675</code> 、 <code>ES2676</code> 、 <code>ES2677</code> 、 <code>ES2678</code> 、 <code>ES2679</code> 、 <code>ES2680</code> 、 <code>ES2681</code> 、 <code>ES2682</code> 、 <code>ES2683</code> 、 <code>ES2684</code> 、 <code>ES2685</code> 、 <code>ES2686</code> 、 <code>ES2687</code> 、 <code>ES2688</code> 、 <code>ES2689</code> 、 <code>ES2690</code> 、 <code>ES2691</code> 、 <code>ES2692</code> 、 <code>ES2693</code> 、 <code>ES2694</code> 、 <code>ES2695</code> 、 <code>ES2696</code> 、 <code>ES2697</code> 、 <code>ES2698</code> 、 <code>ES2699</code> 、 <code>ES2700</code> 、 <code>ES2701</code> 、 <code>ES2702</code> 、 <code>ES2703</code> 、 <code>ES2704</code> 、 <code>ES2705</code> 、 <code>ES2706</code> 、 <code>ES2707</code> 、 <code>ES2708</code> 、 <code>ES2709</code> 、 <code>ES2710</code> 、 <code>ES2711</code> 、 <code>ES2712</code> 、 <code>ES2713</code> 、 <code>ES2714</code> 、 <code>ES2715</code> 、 <code>ES2716</code> 、 <code>ES2717</code> 、 <code>ES2718</code> 、 <code>ES2719</code> 、 <code>ES2720</code> 、 <code>ES2721</code> 、 <code>ES2722</code> 、 <code>ES2723</code> 、 <code>ES2724</code> 、 <code>ES2725</code> 、 <code>ES2726</code> 、 <code>ES2727</code> 、 <code>ES2728</code> 、 <code>ES2729</code> 、 <code>ES2730</code> 、 <code>ES2731</code> 、 <code>ES2732</code> 、 <code>ES2733</code> 、 <code>ES2734</code> 、 <code>ES2735</code> 、 <code>ES2736</code> 、 <code>ES2737</code> 、 <code>ES2738</code> 、 <code>ES2739</code> 、 <code>ES2740</code> 、 <code>ES2741</code> 、 <code>ES2742</code> 、 <code>ES2743</code> 、 <code>ES2744</code> 、 <code>ES2745</code> 、 <code>ES2746</code> 、 <code>ES2747</code> 、 <code>ES2748</code> 、 <code>ES2749</code> 、 <code>ES2750</code> 、 <code>ES2751</code> 、 <code>ES2752</code> 、 <code>ES2753</code> 、 <code>ES2754</code> 、 <code>ES2755</code> 、 <code>ES2756</code> 、 <code>ES2757</code> 、 <code>ES2758</code> 、 <code>ES2759</code> 、 <code>ES2760</code> 、 <code>ES2761</code> 、 <code>ES2762</code> 、 <code>ES2763</code> 、 <code>ES2764</code> 、 <code>ES2765</code> 、 <code>ES2766</code> 、 <code>ES2767</code> 、 <code>ES2768</code> 、 <code>ES2769</code> 、 <code>ES2770</code> 、 <code>ES2771</code> 、 <code>ES2772</code> 、 <code>ES2773</code> 、 <code>ES2774</code> 、 <code>ES2775</code> 、 <code>ES2776</code> 、 <code>ES2777</code> 、 <code>ES2778</code> 、 <code>ES2779</code> 、 <code>ES2780</code> 、 <code>ES2781</code> 、 <code>ES2782</code> 、 <code>ES2783</code> 、 <code>ES2784</code> 、 <code>ES2785</code> 、 <code>ES2786</code> 、 <code>ES2787</code> 、 <code>ES2788</code> 、 <code>ES2789</code> 、 <code>ES2790</code> 、 <code>ES2791</code> 、 <code>ES2792</code> 、 <code>ES2793</code> 、 <code>ES2794</code> 、 <code>ES2795</code> 、 <code>ES2796</code> 、 <code>ES2797</code> 、 <code>ES2798</code> 、 <code>ES2799</code> 、 <code>ES2800</code> 、 <code>ES2801</code> 、 <code>ES2802</code> 、 <code>ES2803</code> 、 <code>ES2804</code> 、 <code>ES2805</code> 、 <code>ES2806</code> 、 <code>ES2807</code> 、 <code>ES2808</code> 、 <code>ES2809</code> 、 <code>ES2810</code> 、 <code>ES2811</code> 、 <code>ES2812</code> 、 <code>ES2813</code> 、 <code>ES2814</code> 、 <code>ES2815</code> 、 <code>ES2816</code> 、 <code>ES2817</code> 、 <code>ES2818</code> 、 <code>ES2819</code> 、 <code>ES2820</code> 、 <code>ES2821</code> 、 <code>ES2822</code> 、 <code>ES2823</code> 、 <code>ES2824</code> 、 <code>ES2825</code> 、 <code>ES2826</code> 、 <code>ES2827</code> 、 <code>ES2828</code> 、 <code>ES2829</code> 、 <code>ES2830</code> 、 <code>ES2831</code> 、 <code>ES2832</code> 、 <code>ES2833</code> 、 <code>ES2834</code> 、 <code>ES2835</code> 、 <code>ES2836</code> 、 <code>ES2837</code> 、 <code>ES2838</code> 、 <code>ES2839</code> 、 <code>ES2840</code> 、 <code>ES2841</code> 、 <code>ES2842</code> 、 <code>ES2843</code> 、 <code>ES2844</code> 、 <code>ES2845</code> 、 <code>ES2846</code> 、 <code>ES2847</code> 、 <code>ES2848</code> 、 <code>ES2849</code> 、 <code>ES2850</code> 、 <code>ES2851</code> 、 <code>ES2852</code> 、 <code>ES2853</code> 、 <code>ES2854</code> 、 <code>ES2855</code> 、 <code>ES2856</code> 、 <code>ES2857</code> 、 <code>ES2858</code> 、 <code>ES2859</code> 、 <code>ES2860</code> 、 <code>ES2861</code> 、 <code>ES2862</code> 、 <code>ES2863</code> 、 <code>ES2864</code> 、 <code>ES2865</code> 、 <code>ES2866</code> 、 <code>ES2867</code> 、 <code>ES2868</code> 、 <code>ES2869</code> 、 <code>ES2870</code> 、 <code>ES2871</code> 、 <code>ES2872</code> 、 <code>ES2873</code> 、 <code>ES2874</code> 、 <code>ES2875</code> 、 <code>ES2876</code> 、 <code>ES2877</code> 、 <code>ES2878</code> 、 <code>ES2879</code> 、 <code>ES2880</code> 、 <code>ES2881</code> 、 <code>ES2882</code> 、 <code>ES2883</code> 、 <code>ES2884</code> 、 <code>ES2885</code> 、 <code>ES2886</code> 、 <code>ES2887</code> 、 <code>ES2888</code> 、 <code>ES2889</code> 、 <code>ES2890</code> 、 <code>ES2891</code> 、 <code>ES2892</code> 、 <code>ES2893</code> 、 <code>ES2894</code> 、 <code>ES2895</code> 、 <code>ES2896</code> 、 <code>ES2897</code> 、 <code>ES2898</code> 、 <code>ES2899</code> 、 <code>ES2900</code> 、 <code>ES2901</code> 、 <code>ES2902</code> 、 <code>ES2903</code> 、 <code>ES2904</code> 、 <code>ES2905</code> 、 <code>ES2906</code> 、 <code>ES2907</code> 、 <code>ES2908</code> 、 <code>ES2909</code> 、 <code>ES2910</code> 、 <code>ES2911</code> 、 <code>ES2912</code> 、 <code>ES2913</code> 、 <code>ES2914</code> 、 <code>ES2915</code> 、 <code>ES2916</code> 、 <code>ES2917</code> 、 <code>ES2918</code> 、 <code>ES2919</code> 、 <code>ES2920</code> 、 <code>ES2921</code> 、 <code>ES2922</code> 、 <code>ES2923</code> 、 <code>ES2924</code> 、 <code>ES2925</code> 、 <code>ES2926</code> 、 <code>ES2927</code> 、 <code>ES2928</code> 、 <code>ES2929</code> 、 <code>ES2930</code> 、 <code>ES2931</code> 、 <code>ES2932</code> 、 <code>ES2933</code> 、 <code>ES2934</code> 、 <code>ES2935</code> 、 <code>ES2936</code> 、 <code>ES2937</code> 、 <code>ES2938</code> 、 <code>ES2939</code> 、 <code>ES2940</code> 、 <code>ES2941</code> 、 <code>ES2942</code> 、 <code>ES2943</code> 、 <code>ES2944</code> 、 <code>ES2945</code> 、 <code>ES2946</code> 、 <code>ES2947</code> 、 <code>ES2948</code> 、 <code>ES2949</code> 、 <code>ES2950</code> 、 <code>ES2951</code> 、 <code>ES2952</code> 、 <code>ES2953</code> 、 <code>ES2954</code> 、 <code>ES2955</code> 、 <code>ES2956</code> 、 <code>ES2957</code> 、 <code>ES2958</code> 、 <code>ES2959</code> 、 <code>ES2960</code> 、 <code>ES2961</code> 、 <code>ES2962</code> 、 <code>ES2963</code> 、 <code>ES2964</code> 、 <code>ES2965</code> 、 <code>ES2966</code> 、 <code>ES2967</code> 、 <code>ES2968</code> 、 <code>ES2969</code> 、 <code>ES2970</code> 、 <code>ES2971</code> 、 <code>ES2972</code> 、 <code>ES2973</code> 、 <code>ES2974</code> 、 <code>ES2975</code> 、 <code>ES2976</code> 、 <code>ES</code>

<code>--version</code> <code>-v</code>			打印信息
<code>--watch</code> <code>-w</code>			在监视下 输出文件
<code>@&lt;file&gt;</code>			从一 例：t

[1] 这些选项是试验性的。

## 相关信息

- 在[tsconfig.json](#)文件里设置编译器选项。
- 在[MSBuild工程](#)里设置编译器选项。

## 概述

编译选项可以在使用MSBuild的项目里通过MSBuild属性指定。

## 例子

```
<PropertyGroup Condition="'$(Configuration)' == 'Debug'">
  <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
  <TypeScriptSourceMap>true</TypeScriptSourceMap>
</PropertyGroup>
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
  <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
  <TypeScriptSourceMap>false</TypeScriptSourceMap>
</PropertyGroup>
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\
  Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\
```

## 映射

编译选项	MSBuild属性名称
--declaration	TypeScriptGeneratesDeclarations
--module	TypeScriptModuleKind
--target	TypeScriptTarget
--charset	TypeScriptCharset
--emitBOM	TypeScriptEmitBOM
--emitDecoratorMetadata	TypeScriptEmitDecoratorMetadata
--experimentalDecorators	TypeScriptExperimentalDecorators
--inlineSourceMap	TypeScriptInlineSourceMap

<code>--inlineSources</code>	TypeScriptInlineSources
<code>--locale</code>	自动的
<code>--mapRoot</code>	TypeScriptMapRoot
<code>--newLine</code>	TypeScriptNewLine
<code>--noEmitOnError</code>	TypeScriptNoEmitOnError
<code>--noEmitHelpers</code>	TypeScriptNoEmitHelpers
<code>--noImplicitAny</code>	TypeScriptNoImplicitAny
<code>--noLib</code>	TypeScriptNoLib
<code>--noResolve</code>	TypeScriptNoResolve
<code>--out</code>	TypeScriptOutFile
<code>--outDir</code>	TypeScriptOutDir
<code>--preserveConstEnums</code>	TypeScriptPreserveConstEnums
<code>--removeComments</code>	TypeScriptRemoveComments
<code>--rootDir</code>	TypeScriptRootDir
<code>--isolatedModules</code>	TypeScriptIsolatedModules
<code>--sourceMap</code>	TypeScriptSourceMap
<code>--sourceRoot</code>	TypeScriptSourceRoot
<code>--suppressImplicitAnyIndexErrors</code>	TypeScriptSuppressImplicitAnyIndexErrors
<code>--suppressExcessPropertyErrors</code>	TypeScriptSuppressExcessPropertyErrors
<code>--moduleResolution</code>	TypeScriptModuleResolution
<code>--experimentalAsyncFunctions</code>	TypeScriptExperimentalAsyncFunctions
<code>--jsx</code>	TypeScriptJSXEmit
<code>--reactNamespace</code>	TypeScriptReactNamespace
<code>--skipDefaultLibCheck</code>	TypeScriptSkipDefaultLibCheck
<code>--allowUnusedLabels</code>	TypeScriptAllowUnusedLabels
<code>--noImplicitReturns</code>	TypeScriptNoImplicitReturns
<code>--noFallthroughCasesInSwitch</code>	TypeScriptNoFallthroughCasesInSwitch

<code>--allowUnreachableCode</code>	TypeScriptAllowUnreachableCode
<code>--forceConsistentCasingInFileNames</code>	TypeScriptForceConsistentCasingInF
<code>--allowSyntheticDefaultImports</code>	TypeScriptAllowSyntheticDefaultImpo
<code>--noImplicitUseStrict</code>	TypeScriptNoImplicitUseStrict
<code>--project</code>	VS不支持
<code>--watch</code>	VS不支持
<code>--diagnostics</code>	VS不支持
<code>--listFiles</code>	VS不支持
<code>--noEmit</code>	VS不支持
<code>--allowJs</code>	VS不支持
VS特有选项	TypeScriptAdditionalFlags

## 我使用的Visual Studio版本里支持哪些选项？

查找 `C:\Program Files`

`(x86)\MSBuild\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets` 文件。可用的MSBuild XML标签与相应的 `tsc` 编译选项的映射都在那里。

## ToolsVersion

工程文件里的 `<TypeScriptToolsVersion>1.7</TypeScriptToolsVersion>` 属性值表明了构建时使用的编译器的版本号（这个例子里是1.7）这样就允许一个工程在不同的机器上使用固定的版本去编译。

如果没有指定 `TypeScriptToolsVersion`，则会使用机器上安装的最新版本的编译器去构建。

如果用户使用的是更新版本的TypeScript，则会在首次加载工程的时候看到一个提示升级工程的对话框。

## TypeScriptCompileBlocked

如果你使用其它的构建工具（比如，gulp，grunt等等）并且使用VS做为开发和调试工具，那么在工程里设

置 `<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>`。这样VS只会提供给你编辑的功能，而不会在你按F5的时候去构建。



# Browserify

## 安装

```
npm install tsify
```

## 使用命令行交互

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

## 使用API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add('main.ts')
  .plugin('tsify', { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

更多详细信息：[smrq/tsify](https://github.com/smrq/tsify)

# Duo

## 安装

```
npm install duo-typescript
```

## 使用命令行交互

```
duo --use duo-typescript entry.ts
```

## 使用API

```
var Duo = require('duo');
var fs = require('fs')
var path = require('path')
var typescript = require('duo-typescript');

var out = path.join(__dirname, "output.js")

Duo(__dirname)
  .entry('entry.ts')
  .use(typescript())
  .run(function (err, results) {
    if (err) throw err;
    // Write compiled result to output file
    fs.writeFileSync(out, results.code);
  });
```

更多详细信息：[frankwallis/duo-typescript](https://github.com/frankwallis/duo-typescript)

## Grunt

### 安装

```
npm install grunt-ts
```

### 基本Gruntfile.js

```
module.exports = function(grunt) {
  grunt.initConfig({
    ts: {
      default : {
        src: ["**/*.ts", "!node_modules/**/*.ts"]
      }
    }
  });
  grunt.loadNpmTasks("grunt-ts");
  grunt.registerTask("default", ["ts"]);
};
```

更多详细信息：[TypeStrong/grunt-ts](#)

## gulp

### 安装

```
npm install gulp-typescript
```

### 基本gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
  var tsResult = gulp.src("src/*.ts")
    .pipe(ts({
      noImplicitAny: true,
      out: "output.js"
    }));
  return tsResult.js.pipe(gulp.dest('built/local'));
});
```

更多详细信息：[ivogabe/gulp-typescript](#)

## jspm

### 安装

```
npm install -g jspm@beta
```

注意：目前jspm的0.16beta版本支持TypeScript

更多详细信息：[TypeScriptSamples/jspm](https://github.com/systemjs/TypeScriptSamples/tree/master/jspm)

## webpack

### 安装

```
npm install ts-loader --save-dev
```

### 基本webpack.config.js

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "bundle.js"
  },
  resolve: {
    // Add '.ts' and '.tsx' as a resolvable extension.
    extensions: [ "", ".webpack.js", ".web.js", ".ts", ".tsx", ".mjs" ],
  },
  module: {
    loaders: [
      // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
      { test: /\.tsx?$/, loader: "ts-loader" }
    ]
  }
};
```

查看[更多关于ts-loader的详细信息](#)

或者

- [awesome-typescript-loader](#)

## MSBuild

更新工程文件，包含本地安装的 `Microsoft.TypeScript.Default.props`（在顶端）和 `Microsoft.TypeScript.targets`（在底部）文件：

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/build/2003"
  <!-- Include default props at the bottom -->
  <Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v9.0\BuildTools\Microsoft.Common.props"
    Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v9.0\BuildTools\Microsoft.Common.props')>
  </Import>

  <!-- TypeScript configurations go here -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Include default targets at the bottom -->
  <Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v9.0\BuildTools\Microsoft.Common.props"
    Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v9.0\BuildTools\Microsoft.Common.props')>
  </Import>
</Project>
```

关于配置MSBuild编译器选项的更多详细信息，请参考：[在MSBuild里使用编译选项](#)

## NuGet

- 右键点击 -> Manage NuGet Packages
- 查找 `Microsoft.TypeScript.MSBuild`
- 点击 `Install`
- 安装完成后，Rebuild。

更多详细信息请参考[Package Manager Dialog](#)和[using nightly builds with NuGet](#)

在太平洋标准时间每天午夜会自动构建TypeScript的 `master` 分支代码并发布到NPM和NuGet上。下面将介绍如何获得并在工具里使用它们。

## 使用 npm

```
npm install -g typescript@next
```

## 使用 NuGet 和 MSBuild

注意：你需要配置工程来使用NuGet包。详细信息参考[配置MSBuild工程来使用NuGet](#)。

[www.myget.org](http://www.myget.org)。

有两个包：

- `Microsoft.TypeScript.Compiler`：仅包含工具 ( `tsc.exe` , `lib.d.ts` , 等。 )。
- `Microsoft.TypeScript.MSBuild`：和上面一样的工具，还有MSBuild的任务和目标 ( `Microsoft.TypeScript.targets` , `Microsoft.TypeScript.Default.props` , 等。 )

## 更新IDE来使用每日构建

你还可以配置IDE来使用每日构建。首先你要通过npm安装包。你可以进行全局安装或者安装到本地的 `node_modules` 目录下。

下面的步骤里我们假设你已经安装好了 `typescript@next` 。

## Visual Studio Code

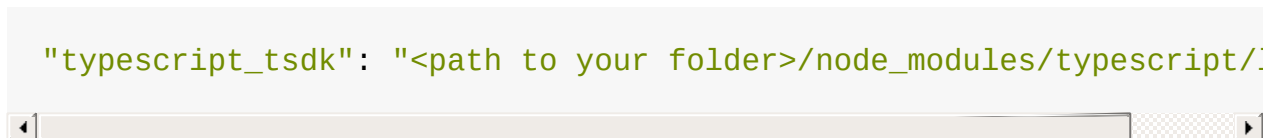
更新 `.vscode/settings.json` 如下：

```
"typescript.tsdk": "<path to your folder>/node_modules/typescript/"
```

详细信息参见[VSCode文档](#)。

## Sublime Text

更新 Settings - User 如下：



```
"typescript_tsdk": "<path to your folder>/node_modules/typescript/lib"
```

详细信息参见[如何在Sublime Text里安装TypeScript插件](#)。

## Visual Studio 2013 and 2015

注意：大多数的改变不需要你安装新版本的VS TypeScript插件。

当前的每日构建不包含完整的插件安装包，但是我们正在试着提供每日构建的安装包。

1. 下载[VSDevMode.ps1](#)脚本。

参考wiki文档：[使用自定义语言服务文件](#)。

2. 在PowerShell命令行窗口里执行：

VS 2015：

```
VSDevMode.ps1 14 -tsScript /node_modules/typescript/lib
```

VS 2013：

```
VSDevMode.ps1 12 -tsScript /node_modules/typescript/lib
```



## 介绍

在JavaScript里（还有TypeScript），`this` 关键字的行为与其它语言相比大为不同。这可能会很令人吃惊，特别是对于那些使用其它语言的用户，他们凭借其直觉来想象 `this` 关键字的行为。

这篇文章会教你怎么识别及调试TypeScript里的 `this` 问题，并且提供了一些解决方案和各自的利弊。

## 典型症状和危险系数

丢失 `this` 上下文的典型症状包括：

- 类的某字段（`this.foo`）为 `undefined`，但其它值没有问题
- `this` 的值指向全局的 `window` 对象而不是类实例对象（在非严格模式下）
- `this` 的值为 `undefined` 而不是类实例对象（严格模式下）
- 调用类方法（`this.doBa()`）失败，错误信息如“`TypeError: undefined is not a function`”，“`Object doesn't support property or method 'doBar'`”或“`this.doBar is not a function`”

程序中应该出现了以下代码：

- 事件监听，比如 `window.addEventListener('click', myClass.doThing);`
- Promise解决，比如 `myPromise.then(myClass.theNextThing);`
- 第三方库回调，比如 `$(document).ready(myClass.start);`
- 函数回调，比如 `someArray.map(myClass.convert)`
- ViewModel类型的库里的类，比如 `<div data-bind="click: myClass.doSomething">`
- 可选包里的函数，比如 `$.ajax(url, { success: myClass.handleData })`

## JavaScript里的 `this` 究竟是什么？

已经有大量的文章讲述了JavaScript里 `this` 关键字的危险性。查看[这里](#)，[这里](#)，或[这里](#)。

当JavaScript里的一个函数被调用时，你可以按照下面的顺序来推断出 `this` 指向的是什么（这些规则是按优先级顺序排列的）：

- 如果这个函数是 `function#bind` 调用的结果，那么 `this` 指向的是传入 `bind` 的参数
- 如果函数是以 `foo.func()` 形式调用的，那么 `this` 值为 `foo`
- 如果是在严格模式下，`this` 将为 `undefined`
- 否则，`this` 将是全局对象（浏览器环境里为 `window`）

这些规则会产生与直觉相反的效果。比如：

```
class Foo {
  x = 3;
  print() {
    console.log('x is ' + this.x);
  }
}

var f = new Foo();
f.print(); // Prints 'x is 3' as expected

// Use the class method in an object literal
var z = { x: 10, p: f.print };
z.p(); // Prints 'x is 10'

var p = z.p;
p(); // Prints 'x is undefined'
```

## `this` 的危险信号

你要注意的最大的危险信号是在要使用类的方法时没有立即调用它。任何时候你看到类方法被引用了却没有使用相同的表达式来调用时，`this` 可能已经不对了。

例子：

```
var x = new MyObject();
x.printThing(); // SAFE, method is invoked where it is referenced

var y = x.printThing; // DANGER, invoking 'y()' may not have correct context

window.addEventListener('click', x.printThing, 10); // DANGER, method is invoked in window context

window.addEventListener('click', () => x.printThing(), 10); // SAFE, method is invoked in window context
```

## 修复

可以通过一些方法来保持 `this` 的上下文。

## 使用实例函数

代替TypeScript里默认的原型方法，你可以使用一个实例箭头函数来定义类成员：

```
class MyClass {
  private status = "blah";

  public run = () => { // <-- note syntax here
    alert(this.status);
  }
}

var x = new MyClass();
$(document).ready(x.run); // SAFE, 'run' will always have correct context
```

- 好与坏：这会为每个类实例的每个方法创建额外的闭包。如果这个方法通常是正常调用的，那么这么做有点过了。然而，它经常会在回调函数里调用，让类实例捕获到 `this` 上下文会比在每次调用时都创建一个闭包来得更有效率一些。
- 好：其它外部使用者不可能忘记处理 `this` 上下文
- 好：在TypeScript里是类型安全的
- 好：如果函数带参数不需要额外的工作
- 坏：派生类不能通过使用 `super` 调用基类方法

- 坏：在类与用户之前产生了额外的非类型安全的约束：明确了哪些方法提前绑定了以及哪些没有

## 本地的胖箭头

在TypeScript里（这里为了讲解添加了一些参数）：

```
var x = new SomeClass();
someCallback((n, m) => x.doSomething(n, m));
```

- 好与坏：内存/效能上的利弊与实例函数相比正相反
- 好：在TypeScript，100%的类型安全
- 好：在ECMAScript 3里同样生效
- 好：你只需要输入一次实例名
- 坏：你要输出2次参数名
- 坏：对于可变参数不起作用（'rest'）

## Function.bind

```
var x = new SomeClass();
// SAFE: Functions created from function.bind are always preserve
window.setTimeout(x.someMethod.bind(x), 100);
```

- 好与坏：内存/效能上的利弊与实例函数相比正相反
- 好：如果函数带参数不需要额外的工作
- 坏：目前在TypeScript里，不是类型安全的
- 坏：只在ECMAScript 5里生效
- 坏：你要输入2次实例名

这个编码规范是给TypeScript开发团队在开发TypeScript时使用的。对于使用TypeScript的普通用户来说不一定适用，但是可以做为一个参考。

## 命名

1. 使用PascalCase为类型命名。
2. 不要使用 `I` 做为接口名前缀。
3. 使用PascalCase为枚举值命名。
4. 使用camelCase为函数命名。
5. 使用camelCase为属性或本地变量命名。
6. 不要为私有属性名添加 `_` 前缀。
7. 尽可能使用完整的单词拼写命名。

## 组件

1. 1个文件对应一个逻辑组件（比如：解析器，检查器）。
2. 不要添加新的文件。:)
3. `.generated.*` 后缀的文件是自动生成的，不要手动改它。

## 类型

1. 不要导出类型/函数，除非你要在不同的组件中共享它。
2. 不要在全局命名空间内定义类型/值。
3. 共享的类型应该在 `types.ts` 里定义。
4. 在一个文件里，类型定义应该出现在顶部。

## `null` 和 `undefined` :

1. 使用 `undefined`，不要使用 `null`。

## 一般假设

1. 假设像Nodes，Symbols等这样的对象在定义它的组件外部是不可改变的。不要去改变它们。

2. 假设数组是不能改变的。

## 类

1. 为了保持一致，在核心编译链中不要使用类，使用函数闭包代替。

## 标记

1. 一个类型中有超过2个布尔属性时，把它变成一个标记。

## 注释

为函数，接口，枚举类型和类使用JSDoc风格的注释。

## 字符串

1. 使用双引号 `""`
2. 所有要展示给用户看的信息字符串都要做好本地化工作（在 `diagnosticMessages.json` 中创建新的实体）。

## 错误提示信息

1. 在句子结尾使用 `.`。
2. 对不确定的实体使用不定冠词。
3. 确切的实体应该使用名字（变量名，类型名等）
4. 当创建一条新的规则时，主题应该使用单数形式（比如：`An external module cannot...`而不是`External modules cannot`）。
5. 使用现在时态。

## 错误提示信息代码

提示信息被划分类成了一般的区间。如果要新加一个提示信息，在上条代码上加1做为新的代码。

- 1000 语法信息
- 2000 语言信息
- 4000 声明生成信息
- 5000 编译器选项信息
- 6000 命令行编译器信息
- 7000 noImplicitAny信息

## 普通方法

由于种种原因，我们避免使用一些方法，而使用我们自己定义的。

1. 不使用ECMAScript 5函数；而是使用[core.ts](#)这里的。
2. 不要使用 `for..in` 语句；而是使用 `ts.forEach`，`ts.forEachKey` 和 `ts.forEachValue`。注意它们之间的区别。
3. 如果可能的话，尝试使用 `ts.forEach`，`ts.map` 和 `ts.filter` 代替循环。

## 风格

1. 使用arrow函数代替匿名函数表达式。
2. 只要需要的时候才把arrow函数的参数括起来。  
比如，`(x) => x + x` 是错误的，下面是正确的做法：

- i. `x => x + x`
- ii. `(x,y) => x + y`
- iii. `<T>(x: T, y: T) => x === y`

3. 总是使用 `{}` 把循环体和条件语句括起来。
4. 开始的 `{` 总是在同一行。
5. 小括号里开始不要有空白。

逗号，冒号，分号后要有一个空格。比如：

- i. `for (var i = 0, n = str.length; i < 10; i++) { }`
- ii. `if (x < 10) { }`
- iii. `function f(x: number, y: string): void { }`

6. 每个变量声明语句只声明一个变量

（比如 使用 `var x = 1; var y = 2;` 而不是 `var x = 1, y = 2;` ）。

7. `else` 要在结束的 `}` 后另起一行。



## 介绍

下面列出了一些在使用TypeScript语言和编译器过程中常见的容易让人感到困惑的错误信息。

## 令人困惑的常见错误

### "tsc.exe" exited with error code 1

修复：

- 检查文件编码，确保为UTF-8 - <https://typescript.codeplex.com/workitem/1587>

### external module XYZ cannot be resolved

修复：

- 检查模块路径是否大小写敏感 - <https://typescript.codeplex.com/workitem/2134>

## 快捷列表

- [Atom](#)
- [Eclipse](#)
- [Emacs](#)
- [NetBeans](#)
- [Sublime Text](#)
- [TypeScript Builder](#)
- [Vim](#)
- [Visual Studio](#)
- [Visual Studio Code](#)
- [WebStorm](#)

## Atom

[Atom-TypeScript](#)，由TypeStrong开发的针对Atom的TypeScript语言服务。

## Eclipse

[Eclipse TypeScript 插件](#)，由Palantir开发的Eclipse插件。

## Emacs

[tide](#) - TypeScript Interactive Development Environment for Emacs

## NetBeans

- [nbts](#) - NetBeans TypeScript editor plugin
- [Geertjan's TypeScript NetBeans Plugin](#)

## Sublime Text

[Sublime的TypeScript插件](#)，可以通过[Package Control](#)来安装，支持Sublime Text 2和Sublime Text 3。

## TypeScript Builder

[TypeScript Builder](#)，TypeScript专用IDE。

## Vim

### 语法高亮

- [leafgarland/typescript-vim](#)提供了语法文件用来高亮显示 `.ts` 和 `.d.ts`。
- [HerringtonDarkholme/yats.vim](#)提供了更多语法高亮和DOM关键字。

### 语言服务工具

有两个主要的TypeScript插件：

- [Quramy/tsuquyomi](#)
- [clausreinke/typescript-tools.vim](#)

如果你想要输出时自动补全功能，你可以安装[YouCompleteMe](#)并添加以下代码到 `.vimrc` 里，以指定哪些符号能用来触发补全功能。[YouCompleteMe](#)会调用它们各自TypeScript插件来进行语义查询。

```
if !exists("g:ycm_semantic_triggers")
  let g:ycm_semantic_triggers = {}
endif
let g:ycm_semantic_triggers['typescript'] = ['.']
```

## Visual Studio 2013/2015

[Visual Studio](#)里安装Microsoft Web Tools时就带了TypeScript。

TypeScript for Visual Studio 2015 [在这里](#)

TypeScript for Visual Studio 2013 在[这里](#)

## Visual Studio Code

[Visual Studio Code](#)，是一个轻量级的跨平台编辑器，内置了对TypeScript的支持。

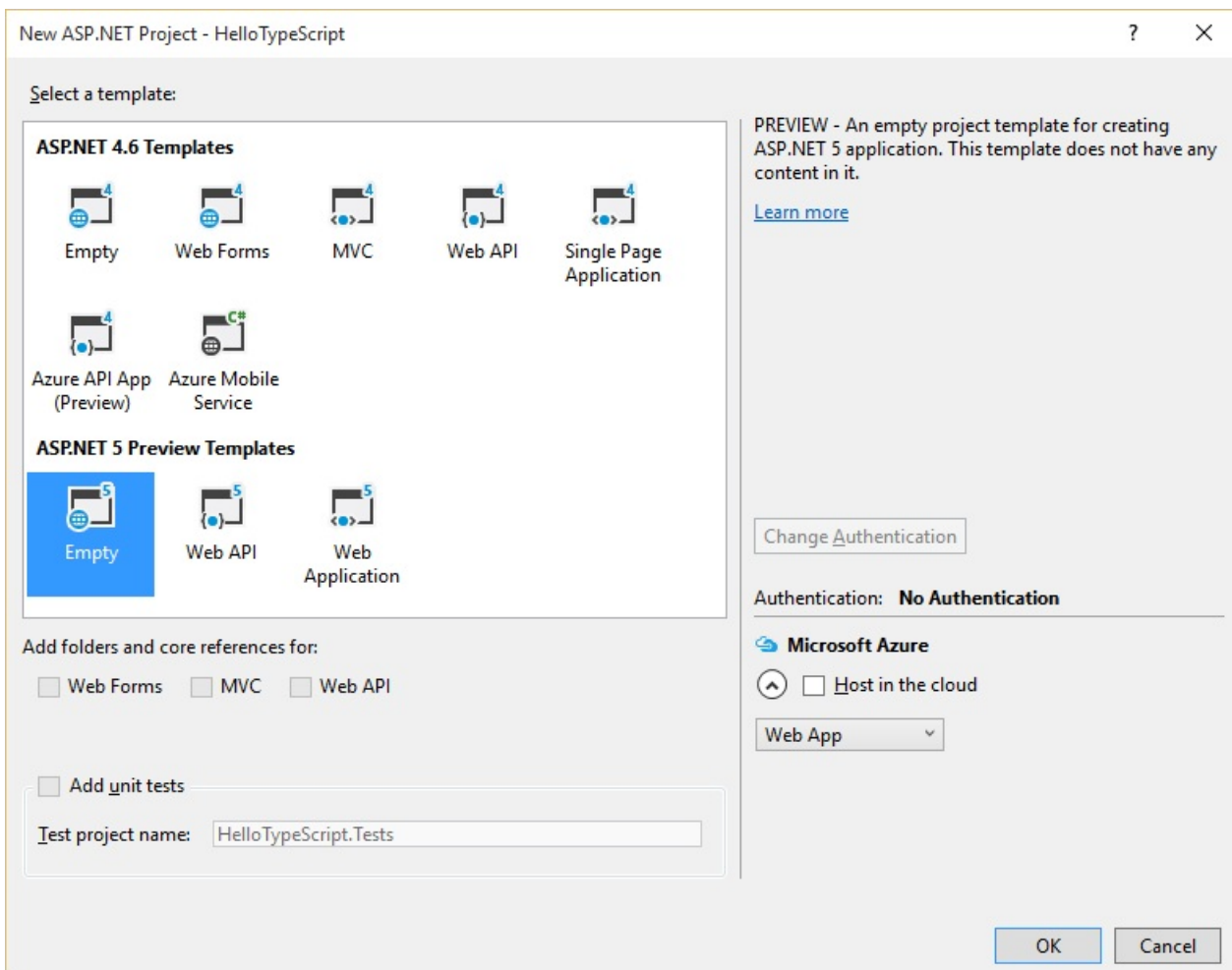
## Webstorm

[WebStorm](#)，同其它JetBrains IDEs一样，直接包含了对TypeScript的支持。

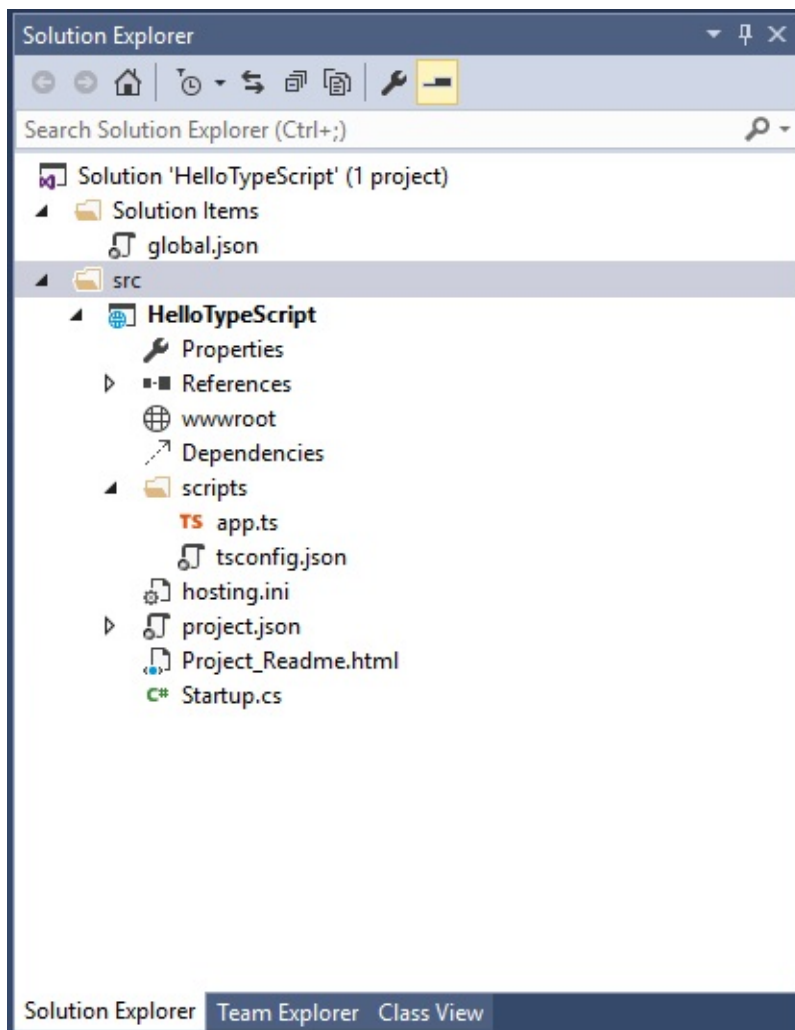
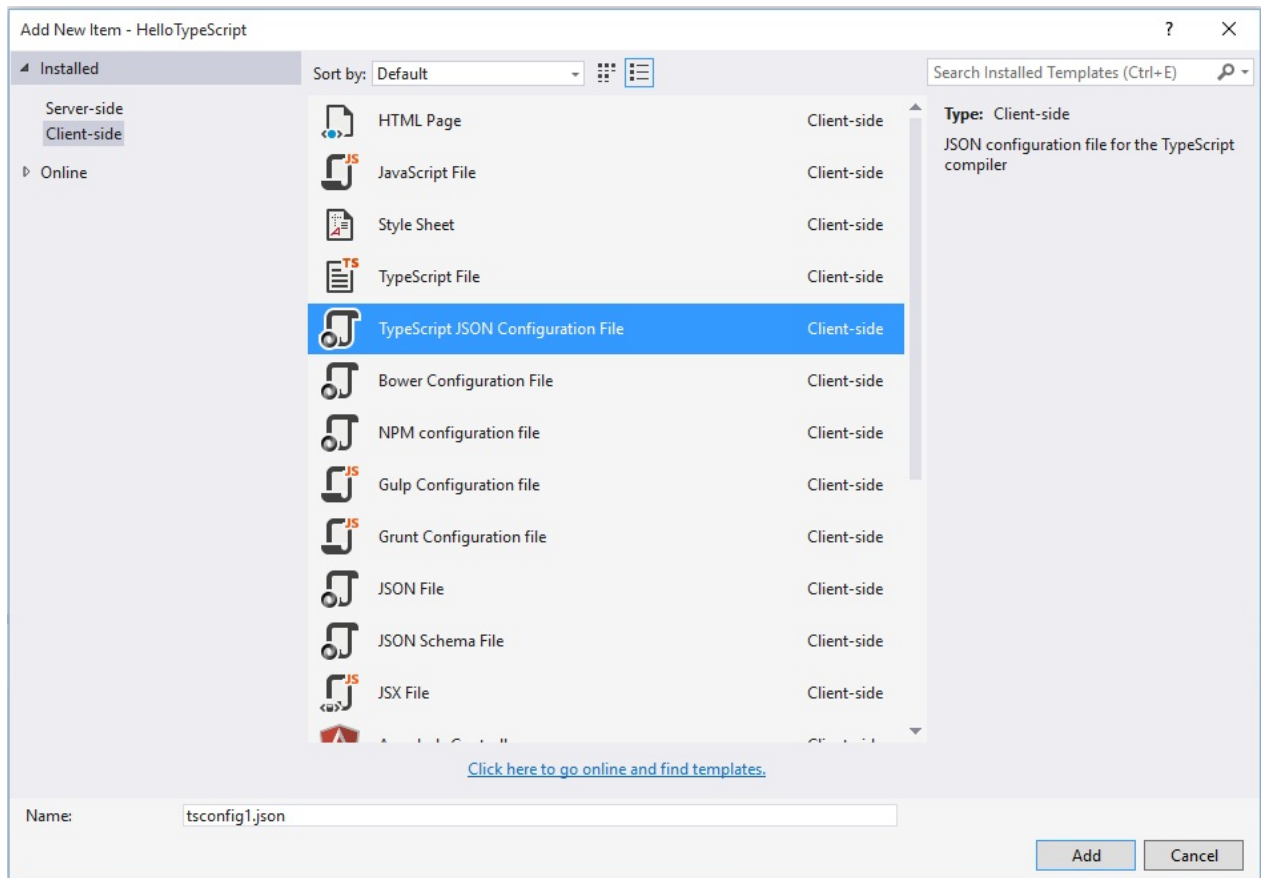
与ASP.NET v5一起使用TypeScript需要你用特定的方式来设置你的工程。更多关于ASP.NET v5的详细信息请查看[ASP.NET v5 文档](#) 在Visual Studio的工程里支持当前的tsconfig.json还在开发之中，可以在这里查看进度[#3983](#)。

## 工程设置

我们就以在Visual Studio 2015里创建一个空的ASP.NET v5工程开始，如果你对ASP.NET v5还不熟悉，可以查看[这个教程](#)。



然后在工程根目录下添加一个 `scripts` 目录。这就是我们将要添加TypeScript文件和 `tsconfig.json` 文件来设置编译选项的地方。请注意目录名和路径都必须这样才能正常工作。添加 `tsconfig.json` 文件，右键点击 `scripts` 目录，选择 `Add`，`New Item`。在 `Client-side` 下，你能够找到它，如下所示。



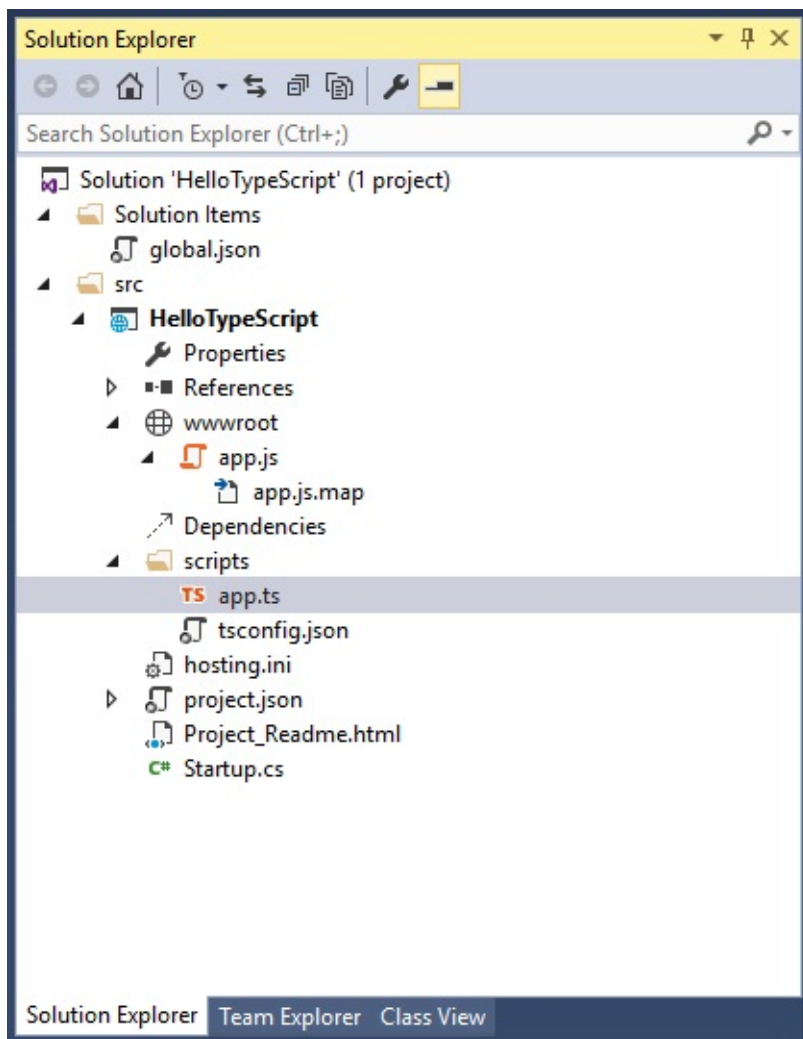
最后我们还要将下面的选项添加到 `tsconfig.json` 文件的 `"compilerOptions"` 节点里，让编译器输出重定向到 `wwwroot` 文件夹：

```
"outDir": "../wwwroot/"
```

下面是配置好 `tsconfig.json` 后可能的样子

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "outDir": "../wwwroot"
  }
}
```

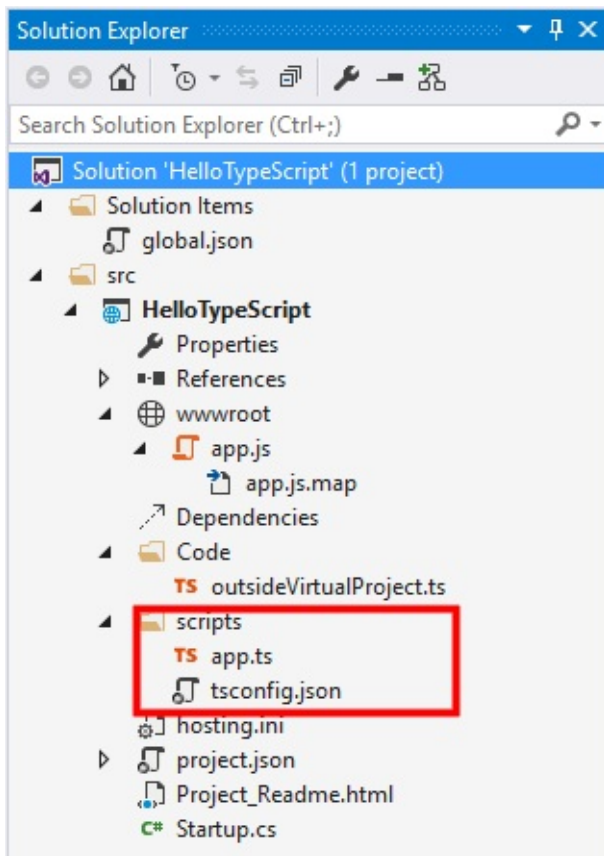
现在如果我们构建这个工程，你就会注意到 `app.js` 和 `app.js.map` 文件被创建在 `wwwroot` 目录里。



## 工程与虚拟工程

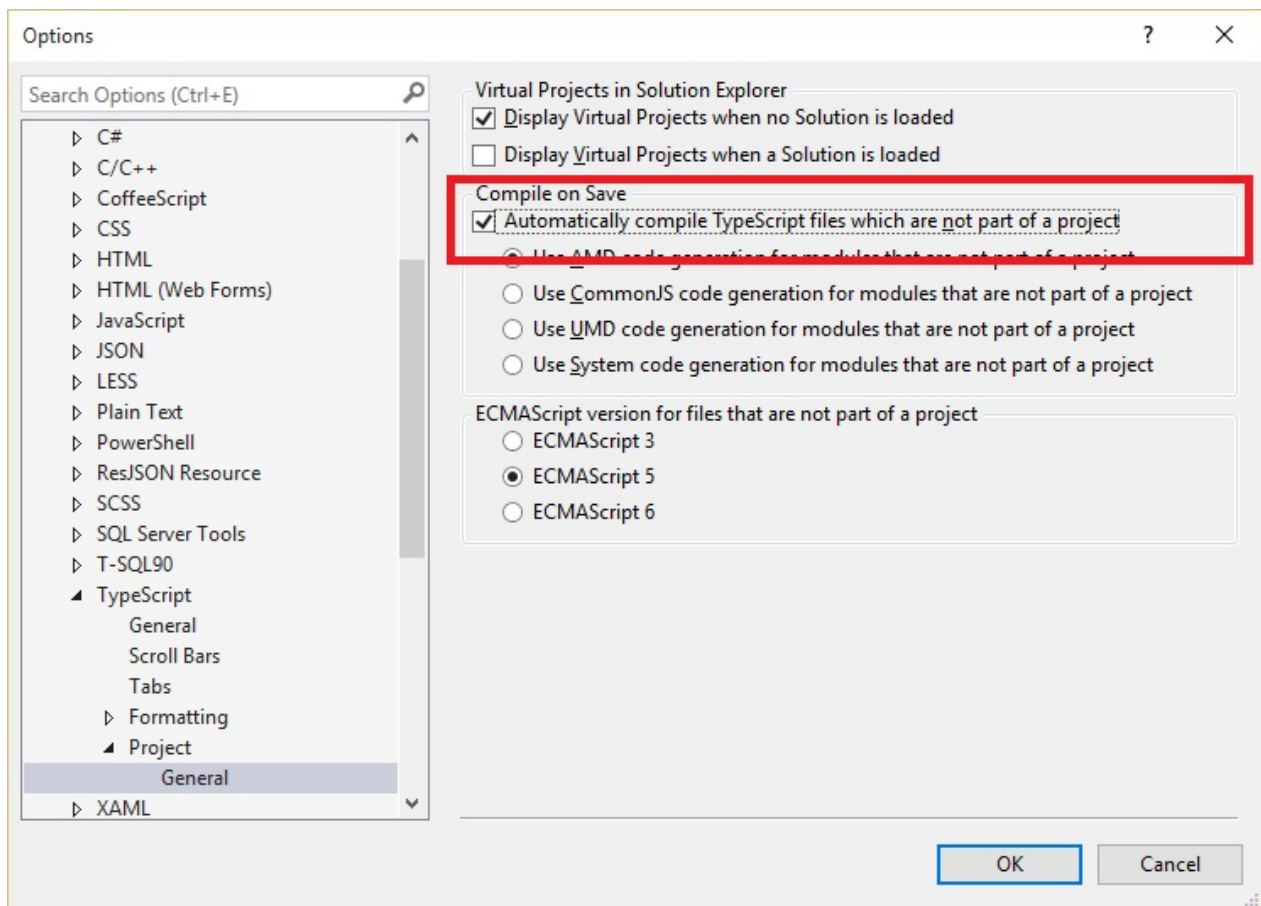
当添加了一个 `tsconfig.json` 文件，你要明白很重要的一点是我们创建了一个虚拟TypeScript工程，在包含 `tsconfig.json` 文件的目录下。被当作这个虚拟工程一部分的TypeScript文件是不会在保存的时候编译的。在包含 `tsconfig.json` 文件的目录外层里存在的TypeScript文件不会被当作虚拟工程的一部分。下图中，可以见到这个虚拟工程，在红色矩形里。



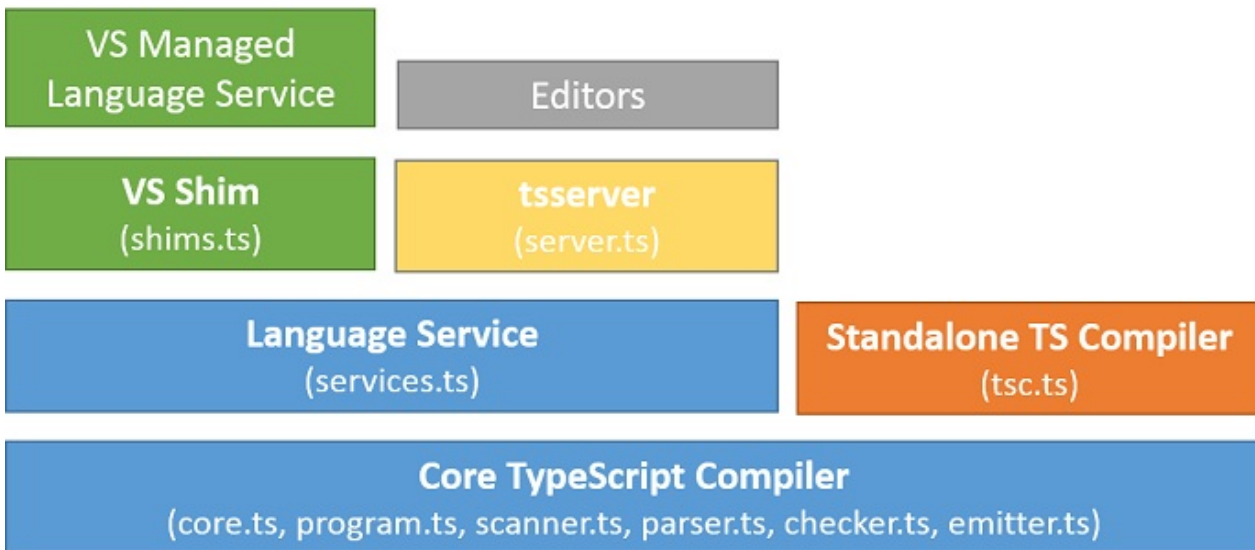


## 保存时编译

想要启用ASP.NET v5项目的保存时编译功能，你必须为不是虚拟TypeScript工程一部分的TypeScript文件启用保存时编译功能。如果工程里存在 `tsconfig.json` 文件，那么模块类型选项的设置会被忽略。



## 层次概述



- 核心TypeScript编译器

- 语法分析器 (**Parser**) : 以一系列原文件开始, 根据语言的语法, 生成抽象语法树 (AST)
- 联合器 (**Binder**) : 使用一个 `Symbol` 将针对相同结构的声明联合在一起 (例如: 同一个接口或模块的不同声明, 或拥有相同名字的函数和模块)。这能帮助类型系统推导出这些具名的声明。
- 类型解析器与检查器 (**Type resolver / Checker**) : 解析每种类型的构造, 检查读写语义并生成适当的诊断信息。
- 生成器 (**Emitter**) : 从一系列输入文件 (`.ts`和`.d.ts`) 生成输出, 它们可以是以下形式之一: JavaScript (`.js`), 声明 (`.d.ts`), 或者是source maps (`.js.map`)。
- 预处理器 (**Pre-processor**) : “编译上下文”指的是某个“程序”里涉及到的所有文件。上下文的创建是通过检查所有从命令行上传入编译器的文件, 按顺序, 然后再加入这些文件直接引用的其它文件或通过 `import` 语句和 `/// <reference path=... />` 标签间接引用的其它文件。

沿着引用图走下来你会发现它是一个有序的源文件列表, 它们组成了整个程序。当解析导出 (`import`) 的时候, 会优先选择“.ts”文件而不是“.d.ts”文件, 以确保处理的是最新的文件。编译器会进行与Nodejs相似的流程来解析导入, 沿着目录链查找与

将要导入相匹配的带`.ts`或`.d.ts`扩展名的文件。导入失败不会报`error`，因为可能已经声明了外部模块。

- **独立编译器 (`tsc`)**：批处理编译命令行界面。主要处理针对不同支持的引擎读写文件（比如：`Node.js`）。
- **语言服务**：“语言服务”在核心编译器管道上暴露了额外的一层，非常适合类编辑器的应用。

语言服务支持一系列典型的编辑器操作比如语句自动补全，函数签名提示，代码格式化和突出高亮，着色等。基本的重构功能比如重命名，调试接口辅助功能比如验证断点，还有TypeScript特有的功能比如支持增量编译（在命令行上使用 `--watch`）。语言服务是被设计用来有效的处理在一个长期存在的编译上下文中文件随着时间改变的情况；在这样的情况下，语言服务提供了与其它编译器接口不同的角度来处理程序和源文件。

请参考 [\[\[Using the Language Service API\]\]](#) 以了解更多详细内容。

## 数据结构

- **Node**: 抽象语法树（AST）的基本组成块。通常 `Node` 表示语言语法里的非终结符；一些终结符保存在语法树里比如标识符和字面量。
- **SourceFile**: 给定源文件的AST。`SourceFile` 本身是一个 `Node`；它提供了额外的接口用来访问文件的源码，文件里的引用，文件里的标识符列表和文件里的某个位置与它对应的行号与列号的映射。
- **Program**: `SourceFile` 的集合和一系列编译选项代表一个编译单元。`Program` 是类型系统和生成代码的主入口。
- **Symbol**: 具名的声明。`Symbols` 是做为联合的结果而创建。`Symbols` 连接了树里的声明节点和其它对同一个实体的声明。`Symbols` 是语义系统的基本构建块。
- **Type**: `Type` 是语义系统的其它部分。`Type` 可能被命名（比如，类和接口），或匿名（比如，对象类型）。
- **Signature**: 一共有三种 `Signature` 类型：调用签名（`call`），构造签名（`construct`）和索引签名（`index`）。

## 编译过程概述

整个过程从预处理开始。预处理器会算出哪些文件参与编译，它会去查找如下引用（`/// 标签和 import 语句）。`

语法分析器（**Parser**）生成抽象语法树（AST） `Node`。这些仅为用户输出的抽象表现，以树的形式。一个 `SourceFile` 对象表示一个给定文件的AST并且带有一些额外的信息如文件名及源文件内容。

然后，联合器（**Binder**）处理AST节点，结合并生成 `Symbols`。一个 `Symbol` 会对应到一个命名实体。这里有个一微妙的差别，几个声明节点可能会是名字相同的实体。也就是说，有时候不同的 `Node` 具有相同的 `Symbol`，并且每个 `Symbol` 保持跟踪它的声明节点。比如，一个名字相同的 `class` 和 `namespace` 可以合并，并且拥有相同的 `Symbol`。联合器也会处理作用域，以确保每个 `Symbol` 都在正确的封闭作用域里创建。

生成 `SourceFile`（还带有它的 `Symbols` 们）是通过调用 `createSourceFile` API。

到目前为止，`Symbol` 代表的命名实体可以在单个文件里看到，但是有些声明可以从多文件合并，因此下一步就是构建一个全局的包含所有文件的视图，也就是创建一个 `Program`。

一个 `Program` 是 `SourceFile` 的集合并带有一系列 `CompilerOptions`。通过调用 `createProgram` API来创建 `Program`。

通过一个 `Program` 实例创建 `TypeChecker`。`TypeChecker` 是TypeScript类型的核心。它负责计算出不同文件里的 `Symbols` 之间的关系，将 `Type` 赋值给 `Symbol`，并生成任何语义 `Diagnostic`（比如：`error`）。

`TypeChecker` 首先要做的是合并不同的 `SourceFile` 里的 `Symbol` 到一个单独的视图，创建单一的 `Symbol` 表，合并所有普通的 `Symbol`（比如：不同文件里的 `namespace`）。

在原始状态初始化完成后，`TypeChecker` 就可以解决关于这个程序的任何问题了。这些“问题”可以是：

- 这个 `Node` 的 `Symbol` 是什么？
- 这个 `Symbol` 的 `Type` 是什么？
- 在AST的某个部分里有哪些 `Symbol` 是可见的？

- 某个函数声明的 `Signature` 都有哪些？
- 针对某个文件应该报哪些错误？

`TypeChecker` 计算所有东西都是“懒惰的”；为了回答一个问题它仅“解决”必要的信息。`TypeChecker` 仅会检测和这个问题有关的 `Node`，`Symbol` 或 `Type`，不会检测额外的实体。

对于一个 `Program` 同样会生成一个 `Emitter`。`Emitter` 负责生成给定 `SourceFile` 的输出；它包括：`.js`，`.jsx`，`.d.ts` 和 `.js.map`。

## 术语

完整开始/令牌开始 (**Full Start/Token Start**)

令牌本身就具有我们称为一个“完整开始”和一个“令牌开始”。“令牌开始”是指更自然的版本，它表示在文件中令牌开始的位置。“完整开始”是指从上一个有意义的令牌之后扫描器开始扫描的起始位置。当关心琐事时，我们往往更关心完整开始。

函数	描述
<code>ts.Node.getStart</code>	取得某节点的第一个令牌起始位置。
<code>ts.Node.getFullStart</code>	取得某节点拥有的第一个令牌的完整开始。

## 琐碎内容 (**Trivia**)

语法的琐碎内容代表源码里那些对理解代码无关紧要的内容，比如空白，注释甚至一些冲突的标记。

因为琐碎内容不是语言正常语法的一部分（不包括ECMAScript API规范）并且可能在任意2个令牌中的任意位置出现，它们不会包含在语法树里。但是，因为它们对于像重构和维护高保真源码很重要，所以需要的时候还是能够通过我们的APIs访问。

因为 `EndOfFileToken` 后面可以没有任何内容（令牌和琐碎内容），所有琐碎内容自然地在非琐碎内容之前，而且存在于那个令牌的“完整开始”和“令牌开始”之间。

虽然这个一个方便的标记法来说明一个注释“属于”一个 `Node`。比如，在下面的例子里，可以明显看出 `genie` 函数拥有两个注释：

```

var x = 10; // This is x.

/**
 * Postcondition: Grants all three wishes.
 */
function genie([wish1, wish2, wish3]: [Wish, Wish, Wish]) {
    while (true) {
    }
} // End function

```

这是尽管事实上，函数声明的完整开始是在 `var x = 10;` 后。

我们依据 [Roslyn's notion of trivia ownership](#) 处理注释所有权。通常来讲，一个令牌拥有同一行上的所有的琐碎内容直到下一个令牌开始。任何出现在这行之后的注释都属于下一个令牌。源文件的第一个令牌拥有所有的初始琐碎内容，并且最后面的一系列琐碎内容会添加到 `end-of-file` 令牌上。

对于大多数普通用户，注释是“有趣的”琐碎内容。属于一个节点的注释内容可以通过下面的函数来获取：

函数	描述
<code>ts.getLeadingCommentRanges</code>	提供源文件和一个指定位置，返回指定位置后的第一个换行与令牌之间的注释的范围（与 <code>ts.Node.getFullStart</code> 配合会更有用）。
<code>ts.getTrailingCommentRanges</code>	提供源文件和一个指定位置，返回到指定位置后第一个换行为止的注释的范围（与 <code>ts.Node.getEnd</code> 配合会更有用）。

做为例子，假设有下面一部分源代码：

```

debugger; /*hello*/
    //bye
    /*hi*/    function

```

`function` 关键字的完整开始是从 `/*hello*/` 注释，但是 `getLeadingCommentRanges` 仅会返回后面2个注释：

```
debugger; /* hello */ _ _ _ _ [CR] [NL] _ _ _ _ /
```

↑                                  ↑                  ↑  
完整开始                                  查找                  第一  
  开始注释

适当地，在 `debugger` 语句后调用 `getTrailingCommentRanges` 可以提取出 `/*hello*/` 注释。

如果你关心令牌流的更多信息，`createScanner` 也有一个 `skipTrivia` 标记，你可以设置成 `false`，然后使用 `setText / setTextPos` 来扫描文件里的不同位置。



## 2.1

- 调查 `Function bind` 操作符
- 支持工程引用
- `readonly` 修饰符
- 调查 具名类型支持
- Language Service API里支持代码重构功能
- 扁平化声明

## 2.0

- 切换到基于转换的生成器
- 支持ES5/ES3 `async / await`
- 支持ES7对象属性展开及剩余属性
- 规定函数的 `this` 类型
- 属性访问上的类型保护
- 切换类型保护
- 支持常量和`Symbol`上计算属性的类型检查
- 可变类型
- 外部装饰器
- 弃用的装饰器
- 条件装饰器
- 函数表达式及箭头函数的装饰器
- 支持节点注册勾子
- 在`tsconfig.json`里支持`Glob`
- 在语言服务API里支持快速修复
- 在`tsserver`/语言服务API里集成`tsd`
- 从`js`文件的`JSDoc`里撮类型信息
- 增强`lib.d.ts`模块化
- 支持外部辅助代码库
- 调查语言服务的可扩展性

## 1.8

- 在TypeScript编译时使用 `--allowjs` 允许JavaScript
- 在循环里允许捕获的 `let / const`
- 标记死代码
- 使用 `--outFile` 连接模块输出
- `tsconfig.json`里支持注释
- 使用 `--pretty` 为终端里的错误信息添加样式
- 支持 `--outFile` 给命名的管道套接字和特殊设备
- 支持使用名字字面量的计算属性
- 字符串字面量类型
- JSX无状态的功能性组件
- 优化联合/交类型接口
- 支持F-Bounded多态性
- 支持全路径 `-project / -p` 参数
- 在SystemJS使用 `--allowSyntheticDefaultImports` 支持 `default` 导入操作
- 识别JavaScript里原型的赋值
- 在模块里使用路径映射
- 在其它模块里增加`global/module`作用域
- 在Visual Studio使用`tsconfig.json`做为高优先级的配置
- 基于 `this` 类型保护
- 支持自定义JSX工厂通过 `--reactNamespace`
- 增强`for-in`语句检查
- JSX代码在VS 2015里高亮
- 发布TypeScript NuGet 包

## 1.7

- ES7幂运算符
- 多态的 `this` 类型
- 支持 `--module` 的 `--target es6`
- 支持目标为ES3时使用装饰器
- 为ES6支持 `async / await` (Node v4)
- 增强的字面量初始化器解构检查

## 1.6

- [ES6 Generators](#)
- [Local types](#)
- [泛型别名](#)
- [类继承语句里使用表达式](#)
- [Class表达式](#)
- [tsconfig.json的 exclude 属性](#)
- [用户定义的类型保护函数](#)
- [增强外部模块解析](#)
- [JSX支持](#)
- [交叉类型](#)
- [abstract 类和方法](#)
- [严格的对象字面量赋值检查](#)
- [类和接口的声明合并](#)
- [新增--init](#)

## 1.5

- [支持解构](#)
- [支持展开操作符](#)
- [支持ES6模块](#)
- [支持for..of](#)
- [支持ES6 Unicode 规范](#)
- [支持Symbols](#)
- [支持计算属性](#)
- [支持tsconfig.json文件](#)
- [支持ES3/ES5的let和const](#)
- [支持ES3/ES5带标记的模版](#)
- [暴露一个新的编辑器接口通过TS Server](#)
- [支持ES7 装饰器提案](#)
- [支持装饰器类型元信息](#)
- [新增--rootDir](#)
- [新增ts.transpile API](#)
- [支持--module umd](#)

- 支持`--module system`
- 新增`--noEmitHelpers`
- 新增`--inlineSourceMap`
- 新增`--inlineSources`
- 新增`--newLine`
- 新增`--isolatedModules`
- 支持新的 `namespace` 关键字
- 支持Visual Studio 2015的`tsconfig.json`
- 增强Visual Studio 2013的模块字面量高亮

## 1.4

- 支持联合类型和类型保护
- 新增`--noEmitOnError`
- 新增`--target ES6`
- 支持`Let and Const`
- 支持模块字面量
- Library typings for ES6
- 支持`Const enums`
- 导出语言服务公共API

## 1.3

- 为新的编译器重写语言服务
- 支持受保护的成员 in classes
- 支持元组类型