TypeScript手册中文版

Ouyang Yadong

Published
with GitBook

目錄

TypeScript手册中文版

原文：TypeScript Handbook

目录：

援助

因水平有限，文档中可能会出现各种疏漏，错误。请您发PR或issues来帮助我们改善文档！

项目地址： https://github.com/oyyd/typescript-handbook-zh

基本类型

程序的运行离不开基本的数据类型，如：numbers, strings, structures, boolean等。TypeScript支持所有你在JavaScript中会用到的数据类型的同时，还添加了便利的枚举类型（enumeration type）以供使用。

布尔类型

真/假值是最基本的数据类型，这种数据类型在JavaScript和TypeScript中（以及其他语言）称为布尔类型（boolean）。

```
var isDone: boolean = false;
```

数字

同JavaScript一样，TypeScript中数字都是浮点数。这些浮点数都被称作数字类型（number）。

```
var height: number = 6;
```

字符串

文本类型的数据是用JavaScript编写网页和服务器等程序的基础。同其他语言一样，我们使用字符串（string）来指代这些文本类型的数据。在TypeScript中，你可以像在JavaScript中那样，使用双引号（""）或单引号（''）来表示字符串。

```
var name: string = "bob";
name = 'smith';
```

数组

同JavaScript一样，TypeScript中我们也可以使用数组。我们可以使用两种不同的方式来写数组。第一种是在元素类型后面附上中括号（[]），来表示这种类型元素的数组：

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

第二种方式是使用泛型数组类型，形式如Array：

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

枚举

TypeScript拓展了JavaScript原生的标准数据类型集，增加了枚举类型（enum）。枚举是一种很有用的数据类型，就像C#等语言中一样，它提供了一种给数字类型的值，设置易于辨别的名字的方法。

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

在默认情况下，枚举类型会从数字0开始标记它的元素。我们可以通过人为地设置元素的数值来改变默认值。例如，上面的例子我们可以设置成从1开始计数：

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

我们甚至可以给所有的枚举元素设置数值：

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

枚举类型有一个便捷特性，我们也可以直接用数值来查找其对应的枚举元素的名称。举例来说，如果我们有一个值为2,但我们不确定这个数值对应枚举类型中的哪个元素，那我们可以直接查找这个数值对应的名称：

```
enum Color {Red = 1, Green, Blue};
var colorName: string = Color[2];

alert(colorName);
```

Any

当我们编写应用时，我们可能会需要描述一些类型不明确的变量。因为这些变量的值可能来源于一些动态的内容，如用户或第三方提供的库。在这种情况下，我们需要略过对这些变量进行的类型检查，让它们直接通过编译时的检查。为了实现这一目的，我们可以把它们标识为'any'类型：

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

使用'any'类型是处理我们已有的JavaScript代码的一种强大的方式。我们可以用它来逐渐增加或减少在编译过程中的类型检查。

当我们知道一个类型的部分数据类型，却又不确定所有的数据类型时，使用'any'可以为我们提供不少方便。比如你有一个数组，但是这个数组中的元素属于不同的数据类型，那你可以这么做：

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

list[1] = 100;
```

Void

与'any'对应的数据类型是'void'，它代表缺省类型。没有返回值的函数就可以认为是'void'类型：

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

接口

TypeScript的核心原则之一，是类型检查会集中关注数据的“结构”（shape）。这一行为有时被称作“鸭子类型”（duck typing）或“结构子类型化”（structural subtyping）。在TypeScript中，接口起到了为这些数据类型命名的作用，同时接口也是定义你代码之间的关系，或你的代码和其他项目代码之间关系的有效方法。

我们的第一个接口

让我们来看看下面这个简单的例子，来了解接口是如何工作的：

```
function printLabel(labelledObj: {label: string}) {
  console.log(labelledObj.label);
}

var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

类型检查器会检查'printLabel'的调用。其中'printLabel'方法有一个参数，我们需要给这个参数传入一个带有名为'label'的字符串类型属性的对象。注意我们传入的这个对象实际上不只有'label'属性，但编译器只检查那些指定的属性，查看它们的类型是否相符。

让我们重写上面的例子，这次我们将使用接口来描述参数的需求，即传入的对象要有字符串类型的label属性。

```
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}

var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

我们可以用这个名为'LabelledValue'的接口来描述我们前面例子中的需求。它仍旧表示需要有一个名为'label'的字符串属性。值得注意的是，与其他编程语言不同，我们不需要明确地说 传给'printLabel'的对象实现了这个接口。这里只关注数据的“结构”。只要我们传给函数的对象满足指定的需求，那这个对象就是合法的。

必须指出的是，类型检查器并不要求这些属性遵循一定的顺序。只要接口要求的属性存在，并符合类型即可。

可选属性（Optional Properties）

接口中的属性并不都是必要的。在遵循一定的条件时，有些属性甚至可以不存在。在创建“option bags”这样的模式时，用户传给函数作为参数的对象，往往只包含部分属性在里面。在这种情况下，可选属性就显得很有用了。

下面是这种模式的一个例子：

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

function createSquare(config: SquareConfig): {color: string; area:
  var newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

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

除了在声明可选属性时需要加上'?'作为标识以外，带有可选属性的接口的写法与其他接口相似。

使用可选属性的优势在于，我们可以在描述可能存在的属性的同时，捕捉那些我们不希望存在的属性。举例来说，如果我们错误地拼写了传给'createSquare'方法的属性名的话，就会有一条错误信息提示我们：

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

function createSquare(config: SquareConfig): {color: string; area:
  var newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.collor;  // Type-checker can catch the
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

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

函数类型

接口可以描述各式各样的JavaScript对象。然而我们除了用接口来描述一个对象的属性以外，也可以用它来描述函数类型。

我们需要给接口一个调用标记来描述函数类型。它看起来就像是只有参数列表和返回类型的函数的定义。

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

定义好了这个接口以后，我们就可以像使用其他接口一样使用这个函数类型接口。下面展示了我们要如何创建一个函数类型变量并给它赋值一个同样类型的函数值。

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

为了能正确地对函数类型进行类型检查，我们允许参数名称与接口不一致。就是说上面的例子也可以这么写：

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

在对函数的参数进行类型检查时，同一时间我们只会对一个参数进行类型检查，检查在接口对应位置上的参数的类型与其是否一致。而我们也会对函数表达式的返回类型进行检查（这里是true和false）。如果这里函数返回的是数字或字符串，那类型检查器就会警告我们返回的类型与SearchFunc接口不相符。

数组类型

我们也可以用接口来描述数组类型，它的声明方式与函数类型相似。数组类型会有一个'index'类型，我们用它来表示数组索引（数组下标）的类型。这样我们也需要索引所对应的返回值的类型。

```
interface StringArray {
  [index: number]: string;
}

var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

TypeScript支持两种索引类型：string和number。同时使用这两种类型的索引也是可能的，只要我们保证数字类型的索引所对应的值的类型，必须是字符串索引对应的值的类型的子类型。

虽然索引标识是描述数组和字典类型的数据的好方法，它同时也会强迫其他所有属性都与索引的返回类型相同。在下面的例子中，'length'属性的类型不符合索引的返回类型，这会导致类型检查抛出错误：

```
interface Dictionary {
  [index: string]: string;
  length: number;    // error, the type of 'length' is not a subtyp
}
```

类的类型

实现一个接口

在C#和Java中，让一个类符合某种特定的约定，是一种很常见的接口的使用方式。在TypeScript中我们也可以这样使用接口。

```
interface ClockInterface {
    currentTime: Date;
}

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

我们可以在一个接口中描述一个类需要实现的方法。就像下面的例子中的'setTime'方法：

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

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

接口只能描述类的公共部分，而不关注私有部分。这种机制不允许我们通过接口来检查一个类的实例的私有部分。

类中的静态部分和实例部分的区别

当使用类和接口时，我们应该要记得一个类有静态部分和实例特有的部分。你可能注意到了，如果创建一个带有构造函数标记的接口，并尝试创建一个类来实现这个接口的话，我们会收到个错误：

```
interface ClockInterface {
    new (hour: number, minute: number);
}

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

这是因为当一个类实现一个接口时，只有实例的部分会被进行检查。构造函数属于静态的部分，它并不在检查的范围之内。

对应地，我们应该直接检查类的静态部分。就像在下面的例子中，我直接检查类本身：

```
interface ClockStatic {
    new (hour: number, minute: number);
}

class Clock  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}

var cs: ClockStatic = Clock;
var newClock = new cs(7, 30);
```

扩展接口

同类一样，接口也可以相互扩展。扩展机制负责将一个接口中的成员拷贝到另一个接口中，这意味着我们可以根据自己的意愿把接口分离成可重用的组件。

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

一个接口可以扩展多个接口，从而成为多接口的组合。

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

混合类型（Hybrid Types）

就像我们之前提到过的，接口可以描述现实中JavaScript所能表现的丰富的数据类型。由于JavaScript动态、灵活的特性，我们有时可能会碰到需要综合使用前面描述的接口的使用方法，来处理一个对象的情景。

举个例子，一个带有额外属性的函数：

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

var c: Counter;
c(10);
c.reset();
c.interval = 5.0;
```

当同第三方JavaScript代码进行交互时，我们可能就需要使用上面的模式，来完整地描述一个数据的类型和结构。

类

传统的JavaScript主要用函数和原型继承作为构建可重用部件的基本方法。然而这对于习惯面向对象方法的程序员来说有些别扭。从下一个版本的JavaScript，即ECMAScript 6开始，JavaScript程序员将可以用基于类的面向对象来构建应用，而TypeScript则允许开发者现在就使用这些新技术。TypeScript将它们编译成可在主流浏览器和平台上运行的JavaScript，从而使得开发者不必等待下一个版本的JavaScript。

类

让我们来看类的一个简单例子：

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

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

如果你曾经使用过C#或Java，那你应该对上面的语法很熟悉。我们声明了一个有三个成员的类'Greeter'，这三个成员分别为'greeting'属性、构造函数和'greet'方法。

你会注意到当我们在类中使用某个成员时，我们使用了'this.'前缀。这表明它是对类成员的一次访问。

代码的最后一行，我们用'new'操作符构建了一个Greeter类的实例。构造过程是：调用我们先前定义的构造函数，创建了一个Greeter类型的新对象，执行构造函数初始化这个对象。

继承

我们可以在TypeScript中使用常见的面向对象模式。而在使用类的编程中，最基本的一个模式便是通过继承来扩展已有的类，创建新的类。

让我们来看一个例子：

```
class Animal {
    name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number = 0) {
        alert(this.name + " moved " + meters + "m.");
    }
}

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

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

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

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

这个例子涵盖了不少其他语言中常见的，同时也属于TypeScript的继承特性。我们看到这里使用了'extends'关键字来创建一个子类。这里的'Horse'和'Snake'继承超类'Animal'并能访问超类的成员。

这个例子同时也展示了我们可以通过在子类上进行特定的定义以重写超类中的方法。这里的'Snake'和'Horse'都创建了一个'move'方法,来重写'Animal'中的'move'方法，从而给予每个类特定的功能。

Private/Public修饰语

默认为Public

你可能已经注意到了，在前面的例子中，我们没有用关键词'public'来标识类成员的可见性。像在C#等语言中，每个对外部可见的成员都需要用'public'进行明确地标识。而在TypeScript中，每个成员都被默认为公有。

你仍旧可以将一个成员标识为private，这样你就能够控制对类外部来说可见的部分。我们可以像下面这样来写前面的例子：

```
class Animal {
    private name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

理解private

TypeScript使用结构化类型系统（structural type system）。当我们比较两种不同的数据类型时，我们会忽略它们是怎么来的。只要它们的每个成员都是一致的，我们就说这两种类型是一致的。

而当比较拥有私有成员的类型时，情况会稍有不同。当比较两种类型是否兼容时，如果其中一种类型拥有私有成员，那么只有当另一种类型也对应拥有具有相同定义的私有成员时，我们才说这两种类型是兼容的。

为了更好地理解这是怎么回事，我们可以看看下面的例子：

```
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; }
}

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

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

在这个例子中，我们有一个'Animal'和'Animal'的子类——'Rhino'。同时我们也有一个看起来和'Animal'结构一样的'Employee'。我们生成了这些类的实例并尝试把它们互相赋值给对方，来看看会产生什么结果。因为'Animal'和'Rhino'的'private name: string'声明的来源相同，私有部分相同，所以我们说它们是兼容的。
而'Employee'的则不同。当尝试将一个'Employee'的实例赋值给'Animal'类型的变量时，我们会得到这些类型不兼容的错误。尽管'Employee'同样有一个名为'name'的私有成员，但它与'Animal'中的'name'来源不同。

参数属性

'public'和'private'关键字同样也允许我们通过创建参数属性，来便捷地创建并初始化类的成员的。这些属性允许我们只通过一个步骤就创建并初始化一个成员。下面是前面例子的另一个版本。注意这里我们没有用'theName'，而是直接在构造函数里声明了个'private name: string'的参数，就并创建和初始化了'name'成员。

```
class Animal {
    constructor(private name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

这里使用'private'创建并初始化了一个私有成员。这对'public'来说也是相似的。

访问器（Accessors）

TypeScript支持用getters/setters与对象中的成员进行交互。这为我们提供了一个控制对对象成员的访问的好方法。

让我们用'get'和'set'来重写一个类。首先是一个没有getters和setters的例子。

```
class Employee {
    fullName: string;
}

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

虽然允许直接设置fullName确实很方便，但如果我们能胡乱地设置它的值话，这种方式就可能会给我们来带麻烦。

在下面这个版本的例子中，在允许用户修改雇员的信息之前，我们会检查并确保用户提供了密码。我们用一个'set'方法替代了直接访问'fullName'的方式，并对密码进行了检查。同时我们也对应添加了一个'get'方法来保证前面例子的代码在这里也能够继续运行。

```
var 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 {
            alert("Error: Unauthorized update of employee!");
        }
    }
}

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

为了证明我们的存取器确实在检查密码，我们可以尝试修改一下密码。结果我们得到了一个警告，提示我们没有获取和修改employee的权利。

注意：要使用访问器的话，我们需要设置编译器生成ECMAScript 5代码。

静态属性

到目前为止，我们都在谈论类的实例上的成员，它们只有在对象被实例化后才能从该对象获取。但我们也可以给一个类创建静态成员，这些静态成员在类上就是可见的，而不是在实例生成后才能获得。在这个例子中，由于'origin'是所有'grids'都具备的一个通用的值，所以我们用'static'来声明'origin'。每一个实例都可以通过在成员名之前加上类的名字来获得静态成员，这和'this.'前缀很相似。这里我们在获取静态成员时，在前面加上了'Grid.'。

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

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

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

高级技巧

构造函数

当你在TypeScript中声明一个类时，实际上你同时创建了多个定义。其中第一个创建的便是类对应的实例的类型。

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

var greeter: Greeter;
greeter = new Greeter("world");
alert(greeter.greet());
```

这里的'var greeter: Greeter'表示我们把Greeter当作Greeter实例的类型。这种做法都快成为习惯面向对象的程序员的天性了。

我们也创建了一个我们称作构造函数的函数。当我们'new'一个实例时这个方法就会被调用。让我们看看前面例子的代码所编译成的JavaScript代码，来看看这到底是怎么回事。

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

var greeter;
greeter = new Greeter("world");
alert(greeter.greet());
```

这里的'var Greeter'被赋值给了构造函数。当我们使用'new'并运行这个函数时，我们获得了这个类的一个实例。这个构造函数同样也包含了这个类所有的静态成员。我们可以认为每个类中都有属于实例的部分和静态部分。

让我们稍微修改一下这个例子，看看会有什么不同：

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

var greeter1: Greeter;
greeter1 = new Greeter();
alert(greeter1.greet());

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

在这个例子中，'greeter1'运行得和之前差不多。我们生成了'Greeter'类的实例并使用了这个实例对象。这种用法我们之前已经见过了。

接着，我们直接使用这个类。我们创建了一个名为'greeterMaker'的新变量。这个变量获得的是这个类本身，或者应该说是这个类的构造函数。这里我们用'typeof Greeter'的意义是"给我Greeter类本身的类型"而不是实例的类型。或者更准确地来讲应该是"给我这个名为Greeter的标识的类型"，即构造函数的类型。Greeter类型（的变量）会包含所有Greeter的静态成员，这些静态成员会存在于Greeter构造函数的里面。为了展示这一点，我们在'greeterMaker'上使用'new'来创建'Greeter'的新实例，并像之前那样使用他们。

把一个类当作接口来用

就像我们前面说过的，一个类的声明会创造两个东西：一个是这个类的实例的类型，另一个是构造函数。因为类会创造类型，所以我们可以在使用接口的地方使用类。

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

interface Point3d extends Point {
    z: number;
}

var point3d: Point3d = {x: 1, y: 2, z: 3};
```

模块

本部分将概述TypeScript中的各种用模块组织代码的方式。内容不仅会覆盖内部模块和外部模块，我们还将讨论每种模块应当在何时使用以及如何使用。同时我们也会对如何使用外部模块，以及在TypeScript中使用模块可能会产生的隐患等进阶话题进行讨论。

第一步

我们首先从下面这段程序讲起，我们通篇都会用到它。我们写了一些非常简单的字符串验证方法，你平时可能也会用这些方法来检查网页用户在表单上的输入，或是用它们检查来自外部的数据的格式。

写在一个文件中的验证器

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

var lettersRegexp = /^[A-Za-z]+$/;
var numberRegexp = /^[0-9]+$/;

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
var strings = ['Hello', '98052', '101'];
// Validators to use
var 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 (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable
    }
});
```

加入模块

当我们添加了更多的验证方法以后，我们会想要以某种方式来组织我们的代码，好让我们能够追踪我们创建的这类类型，同时也不用担心它们与其他对象发生命名冲突。我们应该把所创建的对象包裹进一个模块中，而不在全局环境给它们取一堆不

同的名字。

在这个例子中，我们已经把所有和验证相关的类型都放进了一个名为'Validation'的模块中。为了使这里的接口和类对模块外部可见，我们在开头用export关键词修饰它们。相反的，变量'lettersRegexp'和'numberRegexp'都是验证中的细节部分，我们将保持它们的非输出的状态，使其在外部不可见。由下面例子后面的测试代码我们可以知道，当在模块外面使用时，我们需要指定验证类型，如：Validation.LettersOnlyValidator。

模块化后的验证器

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

    var lettersRegexp = /^[A-Za-z]+$/;
    var numberRegexp = /^[0-9]+$/;

    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
var strings = ['Hello', '98052', '101'];
// Validators to use
var 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 (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable
    }
});
```

分割文件

随着应用规模的逐渐扩大，我们需要将代码分割成多个文件，以使其更易维护。

在这里，我们将验证模块分成了多个文件。尽管这些文件是分散开的，但它们都像是定义在同一个地方一样，作用于同一个模块中。因为这些文件之间互有依赖关系，所以我们添加了引用标签来告诉编译器这些文件之间的关系。测试代码并没有变化。

多文件的内部模块

Validation.ts

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

LettersOnlyValidator.ts

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

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
module Validation {
    var numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var 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 (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable
    }
});
```

当项目涉及到多个文件时，我们就必须保证所有编译后的代码都能被加载进来。我们有两种方法来实现这一点。

第一种方法是通过--out flag来将输入的所有文件内容连接起来，并将结果输出到单个JavaScript文件中：

```
tsc --out sample.js Test.ts
```

编译器将会根据在文件中出现的引用标签自动地排序输出文件的内容。但你也可以手动指定每个文件的输出顺序：

```
tsc --out sample.js Validation.ts LettersOnlyValidator.ts ZipCodeVa
```

另外，我们也可以使用分别输出每个文件的编译方式（这是默认选项）。生成多个JS文件以后，我们需要在网页上用<script>标签按恰当的顺序加载每个文件，像是下面的例子：

MyTestPage.html (部分代码)

```
    <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" />
```

使用外部代码

TypeScript同样也有外部模块的概念。在用到node.js或require.js时我们需要使用外部模块。对于没有用到node.js或require.js的应用则不需要使用外部模块，因为前面讲到的内部模块就能够很好地组织起这类应用的代码。

在外部模块中，文件之间的关系是通过文件级别的输入和输出来指定的。在TypeSciprt中，任何包含顶级import和export关键字的文件都被认为是外部模块。

在下面的例子中，我们将前面的例子转换成了使用外部模块的形式。注意这里我们不再使用module关键字，文件本身就构成了一个模块。我们通过它们的文件名来识别它们。

这里我们用import声明替代了引用标签，import指定了模块间的依赖关系。import声明由两部分构成：require关键字用来指定当前文件所依赖的模块的路径，import后面指定的输入的模块在这个文件中的名称。

```
import someMod = require('someModule');
```

与我们用export定义一个内部模块的公共部分相似，这里我们在顶级声明上用export关键字来指定哪些对象对外部是可见的。

编译时，我们需要在命令行上指定该模块的编译类型。对于node.js来说，要用 --module commonjs；对于require.js来说，要用--module amd。来看下面的例子：

```
tsc --module commonjs Test.ts
```

编译以后，每个外部模块都会变成一个分离的.js文件。类似引用标签，编译器会根据import声明来编译相互依赖的文件。

Validation.ts

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

LettersOnlyValidator.ts

```
import validation = require('./Validation');
var lettersRegexp = /^[A-Za-z]+$/;
export class LettersOnlyValidator implements validation.StringValic
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import validation = require('./Validation');
var numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements validation.StringValidator
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import validation = require('./Validation');
import zip = require('./ZipCodeValidator');
import letters = require('./LettersOnlyValidator');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validation.StringValidator; } = {};
validators['ZIP code'] = new zip.ZipCodeValidator();
validators['Letters only'] = new letters.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable
    }
});
```

外部模块代码生成

编译器会根据编译时指定的类型，为node.js (commonjs)和require.js (AMD)的模块加载系统生成合适的代码。你可以查看每个模块加载器的文档来了解生成代码中的define和require到底做了什么。

下面这个例子展示了import和export语句是如何转换成模块加载代码的。

SimpleModule.ts

```
import m = require('mod');
export var t = m.something + 1;
```

AMD / RequireJS SimpleModule.js:

```
define(["require", "exports", 'mod'], function(require, exports, m
    exports.t = m.something + 1;
});
```

CommonJS / Node SimpleModule.js:

```
var m = require('mod');
exports.t = m.something + 1;
```

Export =

在前面的例子中，我们使用validator时，每个模块都只输出一个值。在这种情况下，虽然我们用一个标识符就可以了，但我们仍需要一个限定名。这样的做法显得很繁重。

而"export ="句法可以指定模块要输出的单一对象。这个对象可以是类，接口，模块，函数或枚举类型。每当这个模块被输入时，其输出的东西就可以被直接使用，而不需要再在模块上加上任何名称。

在下面的例子中，我们用"export ="句法简化了前面Validator的实现，每个模块只会输出一个对象。这种做法简化了使用模块的代码——我们可以直接引用'zipValidator'而不需要用'zip.ZipCodeValidator'。

Validation.ts

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

LettersOnlyValidator.ts

```
import validation = require('./Validation');
var lettersRegexp = /^[A-Za-z]+$/;
class LettersOnlyValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
export = LettersOnlyValidator;
```

ZipCodeValidator.ts

```
import validation = require('./Validation');
var numberRegexp = /^[0-9]+$/;
class ZipCodeValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import validation = require('./Validation');
import zipValidator = require('./ZipCodeValidator');
import lettersValidator = require('./LettersOnlyValidator');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validation.StringValidator; } = {};
validators['ZIP code'] = new zipValidator();
validators['Letters only'] = new lettersValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable
    }
});
```

别名（Alias）

另一种简化我们工作的做法是用"import q = x.y.z"来为常用对象创建短一些的名称。"import q = x.y.z"和用于加载外部模块的"import x = require('name')"是不一样的，它只会为指定的符号创建一个别名。我们可以将这类import（通常只是用作别名）用于任何类型标识符上，包括输入的外部模块所创建的对象。

Basic Aliasing

```
module Shapes {
    export module Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
var sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Squ
```

注意这里我们没有使用require关键字，而是import的右侧赋值为我们导入对象的限定名。它的用法和var相似，但它同时也对导入对象的类型以及命名空间起作用。更重要的是，import的值是独立于原对象的一个引用，所以对别名变量的修改不会作用在原变量上。

可选模块加载和其它一些特殊的加载情景

在某些场景下，你可能需要在某些条件成立的情况下才加载一个模块。在TypeScript中，我们可以用后面展示的一些使用模式，来在保障数据类型安全的前提下触发模块加载器，以满足这种需求及其他特殊的模块加载场景。

编译器会检测在生成的JavaScript中，一个模块到底有没有被使用。对于那些只会被用在类型系统上的模块来说，我们不需要调用require。这种挑选出被引用但却未被使用的模块的做法对性能大有脾益，同时也使加载可选模块成为了可能。

这一模式的核心思想在于我们可以通过import id = require('...')获取外部模块暴露出来的类型。就像下面的if语句块那样，模块加载器会被（require）动态触发， 使得我们可以筛选（reference-culling）优化，从而使模块可以按需加载。让这种模式起作用的关键在于保证我们通过import定义标识符只会发生在类型上（即编译时不会生成JavaScript代码）。

我们可以用typeof关键字来保证类型的安全。用在类型位置上的typeof关键字会生成值所对应的类型，在这个例子中所对应的是外部模块的类型。

node.js中的动态模块加载

```
declare var require;
import Zip = require('./ZipCodeValidator');
if (needZipValidation) {
    var x: typeof Zip = require('./ZipCodeValidator');
    if (x.isAcceptable('.....')) { /* ... */ }
}
```

require.js中的动态模块加载

```
declare var require;
import Zip = require('./ZipCodeValidator');
if (needZipValidation) {
    require(['./ZipCodeValidator'], (x: typeof Zip) => {
        if (x.isAcceptable('...')) { /* ... */ }
    });
}
```

使用其他的JavaScript代码库

我们需要声明非TypeScript代码库所暴露出来的API，才能描述代码库中的数据类型和结构。用模块的形式来表示这些库是很合适的，因为大多数JavaScript的代码库都只会暴露出一些顶层的对象。我们的声明并没有实现环境，通常这些定义都写在在.d.ts的文件中。如果你对C/C++很熟悉的话，你可以把它们当作是.h文件或是"extern" 。接下来让我们来看一些内部模块和外部模块的例子。

包裹成内部模块

代码库D3把它的功能都定义在了一个名为'D3'的全局对象中。由于这个库是通过script标签加载的（而不是通过模块加载器），我们可以使用内部模块来定义它的结构。我们用一个ambient内部模块声明来让TypeScript的编译器可以了解它的结构。举个例子：

D3.d.ts (简要代码**)**

```
declare module 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 var d3: D3.Base;
```

包裹成外部模块

node.js中的大多数的任务是通过加载多个模块来完成的。虽然我们可以在每个模块对应的.d.ts文件中用顶层的输出声明来定义这个模块，但把它们写成一个大的.d.ts文件会更加方便。我们会使用模块的引用名称来实现这一点，这个名称可以在后面的import中使用。举个例子：

node.d.ts (简要代码)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashe
}

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

现在我们可以用"/// "来引用node.d.ts，并且我们可以用类似import url = require('url')的语句来加载相应模块。

```
///<reference path="node.d.ts"/>
import url = require("url");
var myUrl = url.parse("http://www.typescriptlang.org");
```

模块使用中的一些问题

我们将在这一部分讲述使用内部模块和外部模块时会碰到的一些常见的陷阱，以及如何避免它们。

/// to an external module

一种常见的错误是用"/// "句法来引用一个外部模块文件（应该用import）。为了理解它们之间的差异，我们首先需要了解编译器定位外部模块信息的三种方式。

第一种方式是通过"import x = require(...);"声明查找对应的.ts文件。这个文件应该是带有顶层import或export声明的实现文件。

第二种方式是查找一个.d.ts文件。这种做法与第一种做法相似，但这个文件是个声明文件（同样有顶层的import或export声明），而并没有具体的实现的代码。

最后一种方式是利用"ambient外部模块声明"。对于这种方式，我们需要根据代码库来'声明'一个模块。 **myModules.d.ts**

```
// In a .d.ts file or .ts file that is not an external module:
declare module "SomeModule" {
    export function fn(): string;
}
```

myOtherModule.ts

```
/// <reference path="myModules.d.ts" />
import m = require("SomeModule");
```

这里的引用标签允许我们定位包含ambient外部模块的声明的声明文件。这同时也是不少TypeScript样例中的node.d.ts文件的用法。

冗余命名空间

如果你想把一个程序从内部模块转换成外部模块，你很有可能会把文件写成这个样子：

shapes.ts

```
export module Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

这个例子用顶层模块Shapes无缘无故地包裹起了Triangle和Square。这么做会让该模块的使用者感到很费解和麻烦：

shapeConsumer.ts

```
import shapes = require('./shapes');
var t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript中的外部模块有一个重要的特性，即两个不同的外部模块永远都不会把名称附着到同一个作用域上。因为外部模块的用户可以决定使用这个模块时的名称。所以你没有必要事先把要暴露出来的标识用一个命名空间包裹起来。

这里重申一下为什么我们不应该把外部模块放在命名空间里。命名空间是为了提供一个有逻辑意义的分组结构，并防止命名冲突。因为外部模块文件本身就已经是逻辑分组，并且它的顶层名称是由引入（import）它的代码所定义的，所以我们没有必要用额外的模块层来包裹这些输出对象。

修改后的例子：

shapes.ts

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

shapeConsumer.ts

```
import shapes = require('./shapes');
var t = new shapes.Triangle();
```

外部模块的一些副作用

就像每个JS文件和每个模块之间有一一对应的关系一样，TypeScript的外部模块的源码文件和它们生成的JS文件之间也有一一对应的关系。这种做法会产生一个副作用，即我们不可能用--out编译器开关把多个外部文件源码编译连接进同一个JavaScript文件中去。

函数

函数是构建JavaScript应用的基础。通过函数，我们可以把我们的业务逻辑抽象成多层，可以模仿类的实现，可以隐藏信息，可以构建模块。虽然TypeScript中已经有了类和模块，但在描述事物的执行过程时，函数仍旧起着关键的作用。TypeScript给标准的JavaScript函数添加了新的特性以方便我们更好地使用。

函数

首先，TypeScript和JavaScript一样，既可以创建有名称的函数也可以创建匿名函数。这允许我们在实现应用时选择最合适的方式。你既可以在API中生成一堆函数，也可以先构建个一次性的函数，之后再把它替换成另一个函数。

让我们看看JavaScript中这两种应用方式的例子：

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

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

和在JavaScript中一样，函数可以获得函数体外的变量。当它们获得了函数体外的变量时，我们说函数'捕获'了这些变量。虽然这一机制的工作方式以及它的副作用等内容可能已经超出了本文的叙述范围，但清楚地认识这一机制对我们使用JavaScript和TypeScript来说是不可或缺的一步。

```
var z = 100;

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

函数类型

给函数添加类型

让我们给前面的例子加上类型：

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

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

我们可以给函数的每个参数和返回值指定类型。TypeScript可以通过返回的语句知道返回值的类型。所以在很多情况下，我们也可以不指定类型。

书写函数类型

既然我们已经给函数定义了类型，现在让我们完整地写出这个函数各个部分的类型：

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

一个函数上的类型包含两个部分：参数的类型和返回值的类型。如果我们想要完整地写出函数的类型的话，那这两部分都将是必不可少的。我们给每个参数一个名称和类型，罗列出它们。因为参数的名称只是为了程序的可读性，所以我们也可以这么写上面的例子：

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

不管你给函数的参数取了什么名字，只要你列出了这个参数的类型，那它对函数来说就是有效的。

让我们再来看看返回值的类型。我们通过在参数和返回值的类型之间使用'=>'符号来指定一个函数返回值的类型。如果一个函数不返回值的话，我们需要使用'void'类型。

记住，参数类型和返回值的类型共同构成了函数的类型。函数中被捕获的变量并不会影响函数的类型。这些变量实际上是被当作函数的'隐藏状态'，它们并不会成为函数API的一部分。

推断类型

通过前面的例子你可能注意到了，虽然等号的一边有指定类型而另一边没有，但TypeScript的编译器仍能够理解这二者的类型。

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

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

这种类型被称作'语境类型'（'contextual typing'），是一种类型推断。它有助于减少我们维护类型的工作。

可选参数和默认参数

与JavaScript不同，TypeScript函数中的每个参数都被认为是必须的。但这并不是说参数的值不能是'null'，只是当一个函数被调用时，编译器会检查用户提供的参数。编译器同样也会假设这些参数是传入函数的唯一参数。简而言之，我们必须保证传给函数的参数的数量和函数指定的参数数量是一致的。

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

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

在JavaScript中，每个参数被认为是可选的。用户可以按照自己的意愿去掉一部分参数，而没有被传入的参数会被当成是undefined。在TypeScript中，我们可以在一个参数的旁边使用'?'符号来指定这个参数是可选的。举例来说，如果我们想要last name是可选的话：

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

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

可选参数必须放在必选参数的后面。前面的例子中，如果我们想要first name是可选的，而last name是必选的话，我们就需要把first name放在后面。

在TypeScript中，我们也可以设置一个值，用作用户没有传入参数时的默认值。这种参数被称作默认参数（default parameters）。下面的例子把前面例子中的last name的默认值改为了"Smith"。

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

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

和可选参数一样，默认参数必须出现在必选参数的后面。

可选参数和默认参数会共享数据类型。下面的：

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

和

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

享有同样的类型"(firstName: string, lastName?: string)=>string"。默认参数的的默认值不会生效，它表明了这个参数是可选的。

剩余参数

必选参数，可选参数和默认参数有一个共同点：这些参数一次只描述一个参数。有时候你可能希望把多个参数设成一组，或者你没法确定一个函数最终会有多少个参数。在JavaScript中处理这些情况时，你可以直接使用函数体中可以获取的arguments来获得每一个参数。

而在TypeScript中，你可以把这些参数聚集到一个变量中：

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

var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinz
```

剩余参数（rest parameters）可以包含多个可选参数。你可以按你的意愿随意使用这些参数。编译器会把传入函数的参数放入一个以省略号（...）开头为名字的变量之中，以供你在函数中使用。

同时省略号也可以用在带有剩余参数的函数上：

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

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

Lambdas和'this'的使用

学习'this'在函数中的工作方式几乎是每一个学习JavaScript的编码人员的必修课。实际对'this'的学习也是开发者习惯使用JavaScript的一个重要过程。而TypeScript是JavaScript超集，它要求开发人员在懂得如何使用'this'的同时，也能够发现代码中没有被正确使用的'this'。关于JavaScript中的'this'完全足够写一篇文章了，并且实际上也有很多人这么干了。这里我们只关注一些基本的东西。

JavaScript中的函数在被调用时会设置一个'this'变量。虽然这个特性强大而又灵活，但使用这个特性却需要我们时刻关注函数执行的环境。举例来说，当我们在回调函数上执行一个函数时，这个函数的上下文环境就会变得难以预料。

让我们看一个例子：

```
var deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            var pickedCard = Math.floor(Math.random() * 52);
            var pickedSuit = Math.floor(pickedCard / 13);

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

var cardPicker = deck.createCardPicker();
var pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

虽然看起来这段代码会返回一个警告框，但实际上我们只会得到一个错误提示。因为'createCardPicker'创建的函数上的'this'指向的是'window'而不是'deck'对象。执行'cardPicker()'就会产生这样的结果。这里的'this'只能被动态绑定到'window'上（记住：在严格模式下，这里的'this'的值会是'undefined'而不是'window'）。

要解决这一问题，我们就要保证这个函数在被调用之前是被绑定到了正确的'this'对象上。不管这个函数之后会被如何调用，只要我们正确地进行了绑定，这个函数就总能获得原始的'deck'对象。

这里我们用lambda句法（lambda syntax，()=>{}）来代替JavaScript的函数表达式，以解决这个问题。它不是在函数被触发时寻找'this'的对象，而是在函数被创建时就自动捕获'this'指代的对象。

```
var deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // Notice: the line below is now a lambda, allowing us to c
        return () => {
            var pickedCard = Math.floor(Math.random() * 52);
            var pickedSuit = Math.floor(pickedCard / 13);

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

var cardPicker = deck.createCardPicker();
var pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

你可以阅读Yahuda Katz的Understanding JavaScript Function Invocation and “this”来获取更多的信息。

重载

JavaScript本质上是一门动态性极强的语言。一个JavaScript函数可以根据传入参数的类型和数量来返回不同类型的对象，并且这样的使用方式并不少见。

```
var 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") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

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

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

这里的'pickCard'函数会根据用户传入参数的不同来返回两种不同的结果。如果用户传入的是一个表示'deck'的对象，那这个函数就会返回'pickedCard'；如果用户要选择一张card，这个函数就会告诉用户card的结果。那么我们要如何用类型系统来描述这种情景呢？

答案是用一个重载列表来描述函数的多个类型。编译器会用这个列表来处理函数调用。现在让我们用一个重载列表来描述'pickCard'所接收的参数和返回的类型。

```
var 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") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

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

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

通过过载，我们现在就可以对'pickCard'函数进行类型检查了。

编译器在进行类型检查时，其处理方式与普通的JavaScript相似。编译器会查看重载列表，并尝试与函数调用时的参数进行匹配。编译器会把第一个成功匹配的过载认做是正确的一个。也因为如此，我们需要把重载按照从最详细的到最粗略的顺序进行排序。

注意'function pickCard(x): any'并不会成为过载列表的一部分，即这个函数只有两个过载：其中一个重载以一个对象作为参数，另一个以一个数字作为参数。以任何其他的参数调用'pickCard'都会产生错误。

泛型

构建部件是软件工程重要的一部分。这些部件不仅要有精心设计过的，高一致性的API，同时也要具有复用性。如果我们构建的部件不仅能够满足现在的需求，并且也能够应对未来的变化，那我们在构建大型系统时就能够更加地游刃有余。

在C#和Java这样的语言中创建可重用的部件时，我们最常用的工具之一便是泛型('generics')。它允许我们创建一个可同时作用在多种数据类型上的部件。这就使得用户在使用这些部件时可以用它们自己定义的类型。

泛型中的Hello World

让我们用泛型中的"hello world" —— identity函数来作为开始的例子。这个identity函数可以返回传入给它的任何类型的参数。你可以把它想成是'echo'命令。

如果不使用泛型的话，我们要么得给identity函数一个特定的类型：

```
function identity(arg: number): number {
    return arg;
}
```

要么用'any'类型：

```
function identity(arg: any): any {
    return arg;
}
```

虽然用'any'类型时，我们的函数也能够接收各种类型的'arg'，这样看起来也是泛用的。但实际上我们失去了函数返回时的类型信息。比如我们传入一个数字，但我们只能知道函数返回的是'any'类型。

取而代之的，我们需要有一种方式能够获取参数类型，并用这个类型作为函数返回值的类型。这里我们将使用一种特别的变量——类型变量。它只对类型起作用而不对值起作用。

```
function identity<T>(arg: T): T {
    return arg;
}
```

我们现在要给identity函数添加一个类型变量'T'。通过这个'T'我们能够获取用户提供的数据类型（比如：number），然后我们就能在函数中使用这个类型。这里我们把'T'用在函数返回值的类型上。这样在看代码时，我们就知道参数的类型和函数返回的类型是一样的，TypeScript就可以在函数的内部或外部使用这个函数的类型信息。

因为这个版本的'identity'函数可以用在多种类型上，所以我们说它是泛用的（generic）。但这又不同于使用'any'的函数，因为它仍旧保留了准确的类型信息。它能够像我们的第一个'identity'函数那样把number作为参数类型和函数返回值类型。

定义完identity的泛型后，我们就可以通过两种方式来调用它。一种是传入所有的参数给函数，包括类型参数：

```
var output = identity<string>("myString");  // type of output will
```

这里我们把'T'设置为string（string同时也是传入函数的参数的类型）。类型变量要用<>而非()来包裹。

第二种方法可能更常见，即类型参数推断（type argument inference）。编译器会根据传入参数的类型来自动地设置T的值：

```
var output = identity("myString");  // type of output will be 'stri
```

注意这里我们没有显式地用尖括号(<>)来传入类型，编译会根据"myString"来设置类型。类型参数推断在保持代码的简短和可读性上很有用，但在很多复杂的情况下编译器无法推测出类型。这时候我们就需要明确地定义泛型的类型了。

使用泛型类型变量

开始用泛型时你会注意到，创建像'identity'这样的泛型函数时，编译器会强制要求你在函数体中正确地使用泛型类型的参数。就是说，你需要明确这些参数的类型是任意的，并对其进行相应的处理。

让我们看看前面的'identity'函数：

```
function identity<T>(arg: T): T {
    return arg;
}
```

假如你想要在每次调用时获得'arg'的长度的话，你可能会自觉地写成这样：

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

如果你这么写了的话，编译器就会报错说你尝试在'arg'上使用'.length'，但却没有在任何地方声明过'arg'的这个成员。刚才我们说过这些类型变量可能会是任何类型的变量，使用这个函数的人可能会传入一个'number'类型，这样的这个参数就没有'.length'成员了。

假设我们实际上希望这个泛型函数是以T的数组而不是直接以T来运行的，那么我们可以将参数描述成类型的数组，这样我们就可以获得'.length成员了：

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more er
    return arg;
}
```

你可以把logging Identity的类型认为是logginIdentity泛型。它需要一个类型参数T，一个T类型的数组参数'arg'，它最终返回的是T类型的一个数组。如果我们给泛型传入一个数字组成的数组，它就会把T类型绑定成数字，并返回一个数字类型的数组。也就是说现在我们可以使用泛型变量T作为我们可使用类型的一部分，而不是使用参数的整个类型。这让我们能够根据实际情况进行更加灵活的处理。

我们也可以把它写成这样：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more er
    return arg;
}
```

如果你接触过其他语言的话，你可能已经很熟悉这种类型的写法了。我们将在下一部分中讲到如何创建Array这样的泛型类型。

泛型类型

在上一部分中，我们创建了可以与多种类型进行工作的identity泛型函数。在这一部分中，我们将探索函数本身的类型以及泛型接口的创建。

在泛型函数中我们需要先把类型参数列举出来，就像其他的非泛型函数的定义方式一样。

```
function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: <T>(arg: T)=>T = identity;
```

我们也可以给泛型的类型参数一个不同的名称，我们只需要明确地表示出类型变量的数量和它们的使用方式。

```
function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: <U>(arg: U)=>U = identity;
```

我们也可以像描述对象一样来描述泛型的类型：

```
function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: {<T>(arg: T): T} = identity;
```

让我们遵循这种写法来写我们的第一个泛型接口。让我们把前面例子中像是在声明对象一样的部分移到一个接口中：

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: GenericIdentityFn = identity;
```

类似地，我们可能会想把这个泛型参数独立出来，作为整个接口的一个参数。这样我们就可以知道我们泛型接口的类型参数了（即Dictionary而不是Dictionary）。这也可以使得接口的类型参数对其他成员可见。

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: GenericIdentityFn<number> = identity;
```

注意到现在我们的例子已经变得跟前面有些不同了。我们现在用一个非泛型函数的标识作为一个泛型类型的一部分，而不是直接描述一个泛型函数。我们在使用GenericIdentityFn时也需要指定一个对应的类型参数（这里是number）。这种做法

可以有效地限制各种潜在的调用情况。如果我们想要更好地描述一个类型的泛用范围的话，我们就需要明确在调用时，何时应该直接使用类型参数，何时应该要把类型参数放在接口上。

除了泛型接口，我们也可以创建泛型类。但我们无法为枚举和模块创建泛型。

泛型类

泛型类与泛型接口的结构相似。泛型类会有泛型类型参数列表，这个列表是在类名后面以尖括号（<>）定义。

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

上面例子中的'GenericNumber'类是一个非常典型的使用情景。但你可能也注意到了这个泛型类本身并没有限制自己只供'number'类型使用。我们也可在以在其上使用'string'甚至更复杂对象。

```
var stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

同接口一样，在类上设置类型参数就可以确保这个类所有的属性都是同一类型的。

就像前面讲述类时一样，一个类中的类型也可以分成两部分：静态部分和实例部分。而泛型类的"泛型"只是对类的实例部分而言的，就是说当我们在使用泛型类时，静态成员不可以使用类的类型参数。

泛型限定

如果你记得我们更早的一个例子的话，你可能会想要写一个专门用于一部分类型的泛型函数。因为你可能知道这些类型的特点，想更好地使用它们。在我们'logginIdentity'的例子中，我们想要获得'arg'参数的'.length'属性。可是编译时编译器会警告我们，因为它不能确定每个类型是否都会有一个'.length'属性，而我们也不能这么假设。

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

虽然我们不能在任何可能出现的类型上做这种假设，但我们可以限定函数作用在任何有'.length'属性的类型上。这使得只要一个类型有这个成员，我们就认为这个类型是有效的。想要这样做的话，我么就必须列举出我们对T的需求，并做出限制。

我们将用一个接口来描述我们的限制。在这里，我们创建了一个有'.length'属性的接口，并用'extends'关键字来使用这个接口。

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length prope
    return arg;
}
```

因为泛型函数被限制了，它现在就不能接收任意类型了：

```
loggingIdentity(3);  // Error, number doesn't have a .length proper
```

我们应该传入一个值，它的类型的属性要符合要求：

```
loggingIdentity({length: 10, value: 3});
```

在泛型限定上使用类型参数

在某些情况下，声明一个类型参数，并让它被另一个类型参数限制是很有用的，比如：

```
function find<T, U extends Findable<T>>(n: T, s: U) {   // errors b
  // ...
}
find (giraffe, myAnimals);
```

直接限制参数的类型也可以达到同样的效果。我们可以这样重写上面的例子：

```
function find<T>(n: T, s: Findable<T>) {
  // ...
}
find(giraffe, myAnimals);
```

注意：上面的两个例子并不严格相等。第一个函数返回的类型可以是'U'，而第二个函数并没办法拿到'U'类型。

在泛型中使用class类型

当我们在工厂模式上使用TypeScript的泛型时，构造函数就会涉及到类的类型。举例来说：

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

下面是一个进阶例子。它用原型属性（prototype property）推测并限制构造函数和实例之间类型的关系。

```
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!
```

常见错误

下面列举了一些常见的，令人很费解的错误信息。当你在使用TypeScript或进行编译时，也有可能会遇到这些问题。

常见错误信息

"tsc.exe" exited with error code 1.

解决办法: 检查文件的编码格式是否为UTF-8 - https://typescript.codeplex.com/workitem/1587

external module XYZ cannot be resolved

解决办法: 检查模块的路径字母的大小写 - https://typescript.codeplex.com/workitem/2134

混合（Mixins）

在传统的面向对象继承（OO hierarchies）中，通过组合可复用的组件（即一些更简单的类）来构造新的类是非常流行的做法。如果你熟悉像Scala这类语言的话，你对这种做法应该不会陌生。并且这种模式在JavaScript社区中也比较流行。

Mixin的例子

在下面的代码中，我们展示了如何在TypeScript中模仿混合的构建方式。我们会在后面讲述它是怎么运作的。

```
// 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.
    }

    interact() {
```

```
        this.activate();
    }

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

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

////////////////////////////////////////
// In your runtime library somewhere
////////////////////////////////////////

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

这个例子先定义了两个将要被我们混合的类。你会发现这两个类每个都只关注一种特定的功能或活动。然后我们会把这两个类混合起来来构建一个同时带有这两种功能的新的类。

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

}

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

接着，我们将创建一个混合了这二者的类。让我们更详细地看看这是如何实现的。

```
class SmartObject implements Disposable, Activatable {
```

你可能会马上注意到在上面的例子中，我们用'implements'来代替'extends'关键字。这表示我们把这两个类当作是接口。并且与其说我们使用了'Disposable'和'Activatable'的实现，倒不如说是使用了它们的类型。也就是说我们还需要在类中提供这二者的实现。Except, that's exactly what we want to avoid by using mixins.

为了满足我们的需求，我们要为被混合的类的成员创建替身（stand-in）属性及类型。这使得编译器在运行时可以获得这个类的成员。虽然这种记账式的做法会花费我们些时间，但之后我们就能从mixins中受益了。

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

最后，我们把这两个类混合到了一个类中，完整地实现了这个类。

```
applyMixins(SmartObject, [Disposable, Activatable])
```

我们最后还创建了一个辅助函数来帮我们完成混合的工作。它会查询并复制每个被混合的类的属性，填充实现的类中的每一个属性。

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

声明合并

TypeScript上的一些独特的理念，源自我们从类型上描述JavaScript对象结构（shape）变化的需求。TypeScript中独特的'声明合并'（'declaration merging'）就是其中的一个例子。理解这个概念能够帮助你在TypeScript中更好地处理现有的JavaScript代码。同时它也让实现更高级别的抽象成为了可能。

在理解声明是如何合并的之前，先让我们了解到底什么是'声明合并'。

在这篇文章中，声明合并特指由编译器完成的，将拥有同样名称的，两个独立的的声明合并成一个定义（a single definition）的工作。这个合并而来的定义将同时拥有原来的两个声明的特性。声明合并不限于合并两个声明，符合条件的任意个声明都可以被合并。

基本概念

在TypeScript中，一个声明会来源于下面的三种情况中的一种：命名空间/模块，类型，值。用于创建命名空间/模块的声明可以通过点分隔的表示法获得。用于创建类型的声明会确定一个类型的名称及其结构。第三种创建值的声明在编译输出的JavaScript中可见（如函数和变量）。

声明类型	命名空间	类型	值
模块	X		X
类		X	X
接口		X	
函数			X
变量			X

在执行一个声明合并时，理解每个声明到底创建了什么能够帮助你更好地确定到底是什么被合并了。

合并接口

最简单也最常见的声明合并莫过于接口的合并。在最简单的情况下，这种合并只是机械地把两个同名接口的成员合并到同一个接口中。

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

interface Box {
    scale: number;
}

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

接口的非函数的成员必须是独特的。如果两个接口同时声明了一个有着同样名子的非函数的成员的话，编译器会抛出一个错误。

而对于函数成员来说，拥有同样名称的每个函数成员都会被当作是同一个函数的重载情况。同样值得注意的是，当一个接口A在融合另一个接口A时（这里称为A'），A'上的重载的集合将会比接口A上的拥有更高优先级。

比如在下面这个例子中：

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

这两个接口将会合并创造出一个声明。注意每个接口中的元素将会维持同样的顺序，只是这些接口身合并时，后出现的重载集合会出现在前面：

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

合并模块

和接口相似，拥有相同名称的模块也会合并它们的成员。由于模块会同时创建一个命名空间和一个值，这里我们需要同时理解这二者是如何合并的。

当合并两名命名空间时，每个模块输出的接口的声明中所创建的类型的定义会相互合并。它们会创建一个带有合并后的接口的命名空间。

当合并值时，两个模块中的值会合并在一起。如果两个模块中有同名称的值，则第二个模块中的值优先。

下面这个例子中的'Animals'的声明合并：

```
module Animals {
    export class Zebra { }
}

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

等同于：

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

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

虽然这个模块合并的例子对我们的理解很有帮助，但我们同样也需要知道在未被输出的成员上到底发生了什么，来帮助我们更完整地理解模块合并。未被输出的成员只在原模块（未被合并的模块）中可见。这意味着来自不同声明中的成员即便在合并以后也不能看到对方未被输出的成员。

我们可以通过下面的例子更好地理解这一点：

```
module Animal {
    var haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

module Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // <-- error, haveMuscles is not visib
    }
}
```

因为haveMuscles并未被输出，所以只有在未合并前处在同一个模块中的animalsHaveMuscles函数才能知道haveMuscles的存在。虽然在被合并以后，doAnimalsHaveMuscles函数也是Animal模块的一部分，但它无法得知另一个模块中的那些未被输出的成员。

用模块合并类，函数和枚举

事实上模块非常灵活，它也可以合并其他类型的声明。想要合并其他类型的声明的话，这个模块的声明就必须要紧跟在它要合并的其他声明的后面。这样声明得到的结果，会同时拥有这两种声明类型的属性。TypeScript正是通过这一能力来模拟JavaScript和其他一些编程语言上的设计模式的。

我们要看的第一个模块合并的例子是用来合并一个模块和一个类的。这让声明内部类成为了可能。

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

这里合并后的成员的可见性同我们在'Merging Modules'部分中描述的是一样的。所以我们必须输出AlbumLabel类，来使被合并的类能够看见它。合并的结果便是是一个类出现在了另一个类中。你用样也可以用模块来给一个已经存在的类添加更多的静态成员。

除了内部类的模式以外，你可能也很熟悉下面这种JavaScript应用：先创建一个函数，然后再给这个函数添加其他的属性。通过TypeScript的声明合并，你可以在保障类型安全（type-safe）的情况下来实现这种定义。

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

module buildLabel {
    export var suffix = "";
    export var prefix = "Hello, ";
}

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

相似地，模块也可以用静态成员来扩展枚举。

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

module 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;
        }
    }
}
```

不被允许的合并

在TypeScript中，并不是所有类型都能够合并。到目前为止，一个类不能合并另一个类，变量与类之间不能够合并，接口和类之间也不能够合并。如果你想要模拟类的合并的话，你可以通过Mixins部分来了解更多。

类型推断

在本节中，我们将讲到TypeScript中的类型推断（type inference）。我们将详细讲述类型推断发生的时机及发生的方式。

基础

当没有关于类型的明确解释时，有几种情况会使得TypeScript使用类型推断来提供类型信息。举例来说，在下面的代码中：

```
var x = 3;
```

变量x的类型会被推断为数字。这种类型推断会发生在下列情况中：初始化变量和成员，设置参数的默认值，在决定函数的返回类型时。

在大多数情况下，类型推断的结果很明了。我们将在后面探索类型推断中一些比较微妙的情况。

最佳通用类型

当我们需要从多个表达式中推断类型时，我们会通过这些表达式计算出一个"最佳通用类型"（"best common type"）。举例来说，

```
var x = [0, 1, null];
```

在推断上面例子中x的类型时，我们必须要考虑到数组中每个元素的类型。这里我们有数字和null两种可能，而最佳通用类型算法会根据每种类型的情况，选择一种可以兼容所有其他选择的类型。

因为我们需要从候选类型中选择最佳通用类型，所以也可能出现没有一种类型可以兼容所有类型的情况。举例来说：

```
var zoo = [new Rhino(), new Elephant(), new Snake()];
```

在理想的状况下，我们希望zoo 被推断为Animal[]。但是由于数组中并没有Animal类型，我们无法推断出这个数组元素的类型。为了应对这类情况，在数组中如果没有一个元素的类型是其他类型的超类型（super type）的话，我们需要明确提供这个数组的类型：

```
var zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

当找不到最佳通用类型时，类型推断的结果就会变成空对象类型，即{}。因为这种类型并没有成员，所以尝试使用该类型上的任何属性都会产生错误。但在这种我们不能隐式推断出对象的类型的情况下，你仍旧能够通过不涉及类型的操作来安全地使用这个对象。

上下文类型

类型推断也能作用于TypeScript中的一些其他情况，其中一种情况便是"上下文类型"（"contextual typing"）。上下文类型作用在“一个表达式的类型可以通过它出现的位置被推断出”的情况下。举例来说：

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.buton);  //<- Error
};
```

上面的代码会抛出类型错误。因为TypeScript的类型检查器会用Window.onmousedown函数的类型来推断赋值符号另一侧的函数表达式的类型。在这种情况下，它就能推断出mouseEvent参数的类型。如果函数表达式不在这个关乎上下文类型的位置上的话，mouseEvent参数就可以是any类型，而且不会有错误抛出。

如果我们明确地指定了这个表达式的类型的话，上下文类型就会被忽略。我们可以这样来写上面的例子：

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.buton);  //<- Now, no error is given
};
```

带有明确类型注释的函数表达式的参数可以覆盖上下文类型。这样就不会有上下文类型其作用，也就不会有错误产生了。

上下文类型可以作用在很多场景中。常见的例子包括：函数调用中的参数，赋值时等号的右边部分，类型判断，对象的成员，迭代数组，以及返回声明（return statements）。上下文类型也会成为最佳通用类型的一个候选类型。举例来说：

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

在这个例子中，我们有四个最佳候选类型：Animal, Rhino, Elephant, and Snake。在这里最佳通用类型算法当然会选择Animal。

类型兼容性

TypeScript中的类型之间是否兼容，取决于其结构性子类型（structural subtyping）。结构性子类型是仅根据类型的成员来判断类型之间是否兼容，它和名义类型（nominal typing）形成了鲜明的对比。让我们来看看下面的代码：

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

var p: Named;
// OK, because of structural typing
p = new Person();
```

因为Person类并没有把它自己明确地描述为Named接口的实现，所以在像C#和Java这样的名义型类型的语言中，等价的代码会抛出错误。

TypeScript的结构性类型系统（structural type system）是依据JavaScript代码的典型写法设计而来的。因为JavaScript中会广泛使用函数表达式和对象字面量，所以用结构性类型系统代替名义性的结构系统来表达JavaScript代码之间关系会显得更加自然。

关于可靠性

TypeScript中的类型系统允许一些特定的操作，这些操作虽然在编译阶段是不可预测的，但TypeScript却可以保证它们的安全性。当类型系统有这样的属性时，我们就说它是"不可靠"的。TypeScript所允许的这些不可靠的行为都是经过仔细考虑的。通过这个文档，我们将阐述这些行为会在何时产生，以及这些行为产生背后的情景与动机。

起步

TypeScript的结构性类型系统的一个基本原则是：如果y拥有x上的所有成员，我们就说x与y是兼容的。举例来说：

```
interface Named {
    name: string;
}

var x: Named;
// y's inferred type is { name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y;
```

当编译器在检查y能否赋值给x时，它会检查x上的每个属性是否都能在y上被找到对应的，兼容的属性。在这个例子中，y必须有一个名为'name'的string属性，我们才能说它和x兼容。因为这里y确实有，所以这个赋值是被允许的。

在检查用来调用函数的参数时，这个规则也同样适用：

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

记住虽然'y'有一个额外的'location'属性，但这里并不会产生错误。只有目标类型（这个例子中为'Named'）的成员会被用来检查它们之间的兼容性。

这个比较的过程会递归地进行，以遍历这个类型的每一个成员及其子成员。

比较两个函数

当我们在比较原生的类型（primitive types）和对象类型（object types）时，整个过程会直接了当。让我们从两个只有参数列表不同的函数这个基本的例子说起：

```
var x = (a: number) => 0;
var y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

当检查x能否赋值给y时，我们首先会检查它们的参数列表。对于y中的每个参数，我们必须能在x中也找到对应的，兼容类型的参数，我们才能说它没有问题。注意参数的名称并不在考虑范围之内，我们只关注它们的类型。在这个例子中，x中的每个参数都能对应y上的参数，所以这个赋值是被允许的。

而第二个赋值就会产生一个错误。因为y需要的第二个参数'x'并没有，所以这个赋值是不被允许的。

你可能会好奇为什么我们会允许像上面y = x这样'丢弃'参数。实际上在在JavaScript中，忽略额外的函数参数的情况非常普遍。比如Array#forEach会提供三个参数给回调函数：数组元素，它的索引（index）以及包含这个元素的数组。所以允许提供一个只使用第一个参数的回调函数是很有用的：

```
var 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));
```

现在再让我们看看函数的返回类型是如何被处理的。让我们使用两个只有返回类型不一样的函数：

```
var x = () => ({name: 'Alice'});
var y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK
y = x; // Error because x() lacks a location property
```

类型系统会强制要求原函数的返回类型是目标函数的返回类型的子类型。

函数参数双向协变（Bivariance）

当我们在比较函数参数的类型时，如果原来的参数可以赋值给目标参数的话，那么赋值就会成功。反之则会失败。由于我们在最终调用函数时，可能调用的是某个只需要特定类型参数的函数，而却给它传入了类型更宽泛的参数，所以这种做法理论上并不安全。但实际上这种类型的错误是很少见的，并且允许这种做法能允许我们使用更多在JavaScript上通用的模式。下面是一个简短的例子：

```
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) => v
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x +

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) =

// Still disallowed (clear error). Type safety enforced for wholly
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

可选参数和剩余参数

当我们在比较函数之间的兼容性时，可选参数与必选参数之间是可以互换的。源类型（source type）上额外的可选参数不会产生错误，目标类型（target type）上的可选参数即便没有对应的参数传入也不会产生错误。

如果一个函数有剩余参数的话，剩余参数就会被当成是无数个可选参数。

虽然这种做法从类型系统的角度上来看是不可靠的，但是从运行的角度来说，可选参数做法并不是那么容易实现的。因为对于大多数函数而言，在可选参数的位置上传入个‘undefined’也是等价的。

使用函数的常见的模式之一是：传入一个回调函数并执行它。在这一过程中，调用函数参数的数量对于开发人员来说是可以预见的，但类型系统却无从得知参数数量。下面这个例子正体现了这一点：

```
function invokeLater(args: any[], callback: (...args: any[]) => voi
    /* ... 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));
```

函数与重载

对于有重载的函数来说，源类型上的每个重载都需要在目标类型上有一个可兼容的签名（signature）。这可以确保我们在任意状况下都能够像调用源函数一样调用目标函数。在检查兼容性时，函数上特定的重载签名（使用字面量的重载）并不参与检查。

枚举

枚举和数字之间是相互兼容的，不同枚举类型的枚举值之间是不兼容的。举例来说：

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

var status = Status.Ready;
status = Color.Green;  //error
```

类

类和对象字面量类型及接口相似，但它们同时拥有静态部分和实例部分。当我们在比较同一个类下的两个对象之间兼容性时，我们只比较实例上的成员。静态成员和构造函数不会影响它们的兼容性。

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

var a: Animal;
var s: Size;

a = s;  //OK
s = a;  //OK
```

类中的私有成员

类中的私有成员会影响它们的兼容性。当我们在检查类实例的兼容性时，如果这个实例包含一个私有成员，那么只有在目标类型上也包含一个来源于同一个类的同一个私有成员，我们才认为它们是兼容的。这意味这一个类和它的超类是兼容的，但和另一个与它结构相同而继承结构不同的类是不兼容的。

泛型

由于TypeScript是结构性类型系统，所以当类型参数被用作成员类型的一部分时，它只对使用了类型参数的成员产生影响。举例来说：

```
interface Empty<T> {
}
var x: Empty<number>;
var y: Empty<string>;

x = y;  // okay, y matches structure of x
```

在上面的例子中，由于x和y的结构中并没有使用类型参数，所以它们是兼容的。现在让我们给Empty添加一个成员看看结果如何：

```
interface NotEmpty<T> {
    data: T;
}
var x: NotEmpty<number>;
var y: NotEmpty<string>;

x = y;  // error, x and y are not compatible
```

在这种情况下，使用了类型参数，并使其生效的泛型类型和非泛型类型没有什么不同。

对于没有使用指定的类型参数的泛型类型来说，所有未指定类型的参数都会被当作'any'类型来进行兼容性检查。检查方式和非泛型类型是一样的。

举例来说：

```
var identity = function<T>(x: T): T {
    // ...
}

var reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // Okay because (x: any)=>any matches (y: any)
```

进阶话题

子类型 vs 赋值

我们一直在使用'兼容'（compatible）这个词，但它本身并不是这门语言规定中的细则。实际上TypeScript中有两种类型的兼容：子类型上的和赋值上的。它们之间的不同只在于，赋值时会有额外的子类型兼容性。它会允许把'any'或枚举类型赋值为其他类型，或把'any'或枚举类型赋值给其他类型，其中枚举类型的数值必须对应。

TypeScript会根据场景的不同使用两种兼容机制。从实际角度出发，类型兼容性会由赋值兼容性来决定，甚至是在implements和extends子句上。你可以查阅TypeScript上的细则以获取更多信息。

编写.d.ts文件

当我们要使用一个外部JavaScript库或是新的API时，我们需要用一个声明文件（.d.ts）来描述这个库的结构。本节会讲述写这种定义文件（definition files）时会涉及到的一些高级概念。之后会用一些例子，来展示这些概念所对应的定义文件描述。

指导与细则

工作流

写.d.ts文件最好的方式不是根据代码来写，而是根据文档来写。根据文档来写代码能够保证你要表达的东西不会被实现细节所影响。而且文档通常也比JS代码要容易理解。所以我们后面的例子都会假设你正在读一个解释表达代码例子的文档的情景来写。

命名空间

当你在定义接口时（比如说"options"对象），你可以决定是否要把这些类型放到一个模块中。你需要根据具体情况来做这个决定 -- 如果用户很可能时常需要声明这个类型的变量或参数，并且需要它的命名不与其他类型冲突的话，那你大可把它放到一个全局命名空间中。如果这个类型很可能不需要被直接引用，或是不太适合以一个独特的名字来命名的话，那你应该把它放在模块内以避免与其他类型发生冲突。

回调

很多JavaScript的库会事先把一个函数作为参数，并在之后用获取到的参数来调用它。当我们在写这种类型的函数签名（function signatures）时，我们不应该把它们当作是可选参数。我们需要认真考虑"我们需要传入什么参数"而不是"我们要用什么参数"。虽然TypeScript 从0.9.7版本开始不再限制我们传入函数作为可选参数，我们仍旧可以通过外部工具在参数的可选性上强制进行双向协变（bivariance）。

扩展性和声明合并

当我们在写定义文件时，我们需要格外注意TypeScript在扩展已有对象时的规则。你可以用匿名类型或接口来声明一个变量：

匿名类型变量

```
declare var MyPoint: { x: number; y: number; };
```

接口类型变量

```
interface SomePoint { x: number; y: number; }
declare var MyPoint: SomePoint;
```

从使用者的角度来说，这两中声明是等价的。但我们可以通过接口合并来扩展SomePoint类型：

```
interface SomePoint { z: number; }
MyPoint.z = 4; // OK
```

你需要根据实际情况来决定是否允许你的声明通过这种方式被扩展。从这里也可以展示出你对这个库的想法。

解构类

TypeScript中的类会创建出两种独立的类型：一种是实例类型，它定义了类实例上的成员；另一种是构造函数类型，它定义了构造函数的成员。因为构造函数类型包括了类中的静态成员，我们可以把当作是"静态的"类型。

虽然我们可以使用typeof关键字来引用一个类中的静态部分，但在写定义文件时，我们有时候需要用被解构过的类。它能明确地将类中的实例部分和静态部分分开。

举个例子，从一个使用者的角度来看，下面的两种声明方式几乎是等价的：

标准声明方式

```
class A {
    static st: string;
    inst: number;
    constructor(m: any) {}
}
```

解构方式

```
interface A_Static {
    new(m: any): A_Instance;
    st: string;
}
interface A_Instance {
    inst: number;
}
declare var A: A_Static;
```

它们的不同之处在于：

我们可以用extends继承标准的类，但不能继承解构过的类。除非下一个版本的TypeScript允许这种继承表达式。

我们可以（通过声明合并）给标准的类和解构类上的静态部分添加成员。

You'll need to come up with sensible names for more types when writing a decomposed class 我们可以给解构过的类添加新的实例成员，但不能给标准类添加。在写解构类的时候，如果你想要添加更多类型的话，你可能需要想一些更有意义的名字。

命名习惯

总的来说，不要在接口前加I（比如：IColor）。因为TypeScript中接口的概念要比C#或Java更宽泛。IFoo这类命名习惯没有太多作用。

例子

来看看下面的这些例子。每个例子都先给出了一个库的使用方式，然后再给出对于这样的库来说比较合适的类型定义。如果一个库有多种比较合适的定义方式的话，这里会把它们都列举出来。

Options对象

用法

```
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

```
module animalFactory {
    interface AnimalOptions {
        name: string;
        height?: number;
        weight?: number;
    }
    function create(name: string, animalOptions?: AnimalOptions): A
}
```

带有属性的函数

用法

```
zooKeeper.workSchedule = "morning";
zooKeeper(giraffeCage);
```

类型声明

```
// Note: Function must precede module
function zooKeeper(cage: AnimalCage);
module zooKeeper {
    var workSchedule: string;
}
```

用new调用的函数

用法

```
var w = widget(32, 16);
var 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 var widget: WidgetFactory;
```

全局/对外部不可知的（external-agnostic）函数库

用法

```
// Either
import x = require('zoo');
x.open();
// or
zoo.open();
```

类型声明

```
module zoo {
  function open(): void;
}

declare module "zoo" {
    export = zoo;
}
```

外部模块中的单个复杂的对象

用法

```
// Super-chainable library for eagles
import eagle = require('./eagle');
// Call directly
eagle('bald').fly();
// Invoke with new
var eddie = new eagle(1000);
// Set properties
eagle.favorite = 'golden';
```

类型声明

```
// Note: can use any name here, but has to be the same throughout t
declare function eagle(name: string): eagle;
declare module eagle {
    var favorite: string;
    function fly(): void;
}
interface eagle {
    new(awesomeness: number): eagle;
}

export = eagle;
```

回调

用法

```
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): voi
```

TypeScript 1.5

ES6模块

TypeScript 1.5支持ECMAScript 6 （ES6）模块（modules）。你可以把ES6模块当作是带着新语法的TypeScript外部模块（external modules）：ES6的模块是被零散载入的源码文件，它们可以输入（import）其他的模块，也可以输出内容供外部使用（exports）。ES6模块给import和export声明提供了一些新的特性。虽然TypeScript并不强制要求在项目使用ES6模块，但我们还是推荐你在库和应用上使用这种新的模块语法，并更新原来的模块。ES6模块的和TypeScript原来的内部模块和外部模块是可以共存的。你甚至可以按你的意愿来构造并混合它们。

Export声明

除了TypeScript现有的export装饰声明（decorating declarations）外，我们也可以零散地用export声明来输出模块中的成员。我们甚至可以用子句（clauses）来给export中的内容替换名称。

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

import声明也可以使用子句来给import后面的内容指定用于本地的名称。举例来说：

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

除了独立的import外，我们也可以通过导入命名空间来导入一整个模块：

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

通过使用from子句，一个模块可以复制给定的模块所输出的内容到当前模块中，而不需要生成用于本地的名称。

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

export * 常被用在将一个模块的输出再次进行输出。在创建专门用来聚集其他模块输出的模块时，这种方式非常有用。

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

Default Export

export default声明是用来指定一个表达式的。这个表达式的内容会成为模块默认的输出内容：

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

对应的，我们可以用default imports导入这些内容：

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

Bare Import

我们可以用"bare import"导入一个模块，以获得导入这个模块时所带来的附加作用（side-effects）。

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

你可以通过查阅ES6 module的支持说明来了解更多关于模块的信息。

声明和赋值时的解构

TypeScript 1.5添加了对ES6中的解构声明和解构赋值（destructuring declarations and assignments）的支持。

声明

解构声明会把从对象的属性（或数组的元素）中抽取出来的值，赋值给一个或多个变量。

举例来说，下面的例子声明了x, y, z变量，并将他们的值分别初始化为getSomeObject().x，getSomeObject().y和getSomeObject().z。

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

我们同样可以在数组上使用解构声明来抽取出数组中的值：

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

同样的，解构也可以被用在函数参数的声明上：

```
function drawText({ text = "", location: [x, y] = [0, 0], bold = fa
    // Draw text
}

// Call drawText with an object literal
var item = { text: "someText", location: [1,2,3], style: "italics"
drawText(item);
```

赋值

解构的模式也可以用在通常的赋值表达式上。举个例子，我们可以用一个解构赋值来实现交换两个变量的值：

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

支持 `let` 和 `const`

现在我们也可以在ES3和ES5上使用 `let` 和 `const` 声明了。

常量

```
const MAX = 100;

++MAX; // Error: The operand of an increment or decrement
       //        operator cannot be a constant.
```

块级作用域

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

alert(a); // Error: a is not defined in this scope
```

支持for..of

你现在可以在TypeScript 1.5的ES3/ES5数组上使用ES6 for..of循环了。同时它也完整支持ES6上的迭代器接口（Iterator interfaces）。

例子：

当我们让TypeScript支持ES3/ES5时，编译器会将使用for..of的数组转换成我们惯用的ES3/ES5代码：

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

将会被转换成：

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

装饰器(Decorators)

TypeScript装饰器（decorator）是基于ES7的装饰器提案实现的。

装饰器是：

- 一个表达式
- 它会作为一个函数执行
- 它把目标，名称，属性描述（property descriptor）作为参数
- 它可以返回一个属性描述，以作用在目标对象上

你可以查阅Decorators提案来获取更多信息。

例子：

`readonly` 和 `enumerable(false)` 这两个装饰器会在method属性被放置在类 `C` 上之前作用于method。装饰器会改变属性和方法的实现。比如在这个例子中，method的属性会被添加writable: false和enumerable: false两项属性描述。

```
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;
  }
}
```

动态计算的属性

如果你想在初始化对象时给对象添加一些动态的属性的话，可能会碰上些小麻烦。举下面的例子来说：

```
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 };
}
```

我们需要先为neighbor-map创建一个对象变量，然后才能动态初始化它。在TypeScript 1.5中，我们可以让编译器替我们完成这类工作：

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

字符串中的Unicode编码点转义

ES6引入了escapes，使得用户仅需要使用一次escape就能表示一个Unicode编码点（codepoint）。

举例来说，假设我们需要转义一个包含UTF-16/UCS2字符" 的字符串。其中" 由一个代理对（surrogate pair）来表示，这就是说它是由一对16比特的编码单元——`0xD842` 和 `0xDFB7` 来表示的。这意味着你必须转义 `"\uD842\uDFB7"` ，但实际上我们很难辨别这是两个独立的字符还是一个代理对。

通过ES6的编码点转义，你可以通过像 `"\u{20bb7}"` 这样的转义来清楚地表达字符串或字符串模板中的字符表达的到底是什么。TypeScript会把这个字符串转换成ES3/ES5中的 `"\uD842\uDFB7"` 。

在ES3/ES5中实现加标记的模板字符串

在TypeScript 1.4中，我们添加了对字符串模板的支持（ES3/ES5/ES6），以及对ES6中的标记模板（tagged templates）的支持。这里要感谢@ivogabe为我们提供的一些深思熟虑的想法和工作，使得我们在ES3和ES5中也可以使用标记模板。

当我们想要将代码转换成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`
```

会变成：

```
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", " \\
var _a;
```

可选给AMD依赖添加名称

`/// <amd-dependency path="x" />` 会告诉编译器，由于当前模块的调用，有一个非TS模块的依赖需要被注入。然而TS代码并不能使用这些代码。

而新添加的 `amd-dependency name` 属性则允许我们给一个amd依赖命名。

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

上面的代码生成的JS代码是：

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

支持用tsconfig.json配置项目

我们可以给一个文件夹添加一个 `tsconfig.json` 文件来表示当前文件夹是一个TypeScript项目的根目录。通过tsconfig.json，我们可以指定根文件（root files）以及编译选项。每个项目对会以下面的方式之一进行编译：

- 如果我们使用tsc指令但是不指定输入文件，编译器会就会从当前文件夹开始，向上寻找tsconfig.json文件。

- 如果我们使用tsc指令但是不指定输入文件，那我们可以使用-project（或 -p）命令行选项来指定包含tsconfig.json文件的文件夹。

例子:

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

你可以查看tsconfig.json的维基页面获取详细信息。

`--rootDir` 命令行

在进行输出文件时使用 `--outDir` 选项会根据输入文件的路径结构来进行。编译器会以根目录下的文件的路径作为所有输入文件的最长路径截点（即路径不会超过根目录），使用每个输入文件的路径来放置输出文件。

但有时候我们并不希望这样。比如我们输入 `FolderA\FolderB\1.ts` 和 `FolderA\FolderB\2.ts`，则输出文件都会出现在一个 `FolderA\FolderB\` 这样的镜像结构的文件中。如果我们再增加一个 `FolderA\3.ts` 作为输入文件，则这个文件对应的输出文件就会出现在一个 `FolderA\` 文件中。

而 `--rootDir` 允许我们指定那些输出时，其路径要被镜像的文件夹，来代替计算输出路径。