# TypeScript

面试知识：

===============================================================================

学习：

TypeScript概念

TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准（[ES6 教程](https://www.runoob.com/w3cnote/es6-tutorial.html" \t "https://www.runoob.com/typescript/_blank)）。

TypeScript 由微软开发的自由和开源的编程语言。

TypeScript 设计目标是开发大型应用，它可以编译成纯 JavaScript，编译出来的 JavaScript 可以运行在任何浏览器上。

### TypeScript语言特性

TypeScript 是一种给 JavaScript 添加特性的语言扩展。增加的功能包括：

* 类型批注和编译时类型检查
* 类型推断
* 类型擦除
* 接口
* 枚举
* Mixin
* 泛型编程
* 名字空间
* 元组
* Await

以下功能是从 ECMA 2015 反向移植而来：

* 类
* 模块
* lambda 函数的箭头语法
* 可选参数以及默认参数

### JavaScript 与 TypeScript 的区别

TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法，因此现有的 JavaScript 代码可与 TypeScript 一起工作无需任何修改**，TypeScript 通过类型注解提供编译时的静态类型检查。**

TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译。

### TypeScript 安装

**NPM 安装 TypeScript**

npm install -g typescript

安装完成后我们可以使用 **tsc** 命令来执行 TypeScript 的相关代码，以下是

查看版本号：

$ tsc -v Version 3.2.2

新建一个 app.ts 的文件，代码如下：

var message:string = "Hello World" console.log(message)

**将 TypeScript 转换为 JavaScript 代码**

tsc app.ts

这时候再当前目录下（与 app.ts 同一目录）就会生成一个 app.js 文件，代码如下：

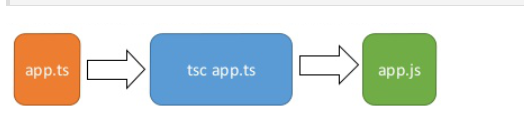
var message = "Hello World";

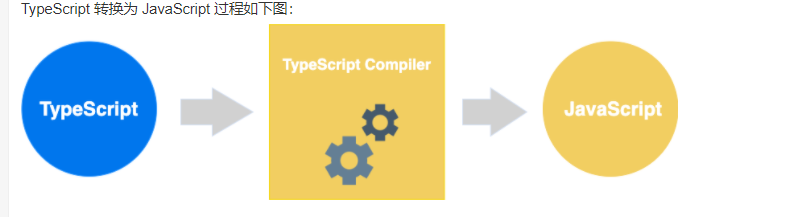
console.log(message);

使用 node 命令来执行 app.js 文件：

$ node app.js

Hello World

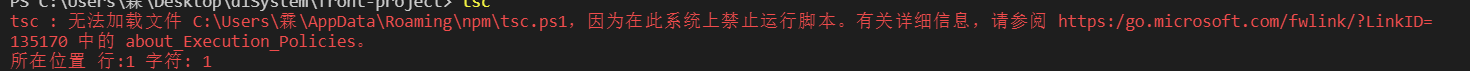




**同时编译多个 ts 文件**

tsc file1.ts file2.ts file3.ts

**BUG解决：全局安装typescript 报错 系统上禁止运行脚本**



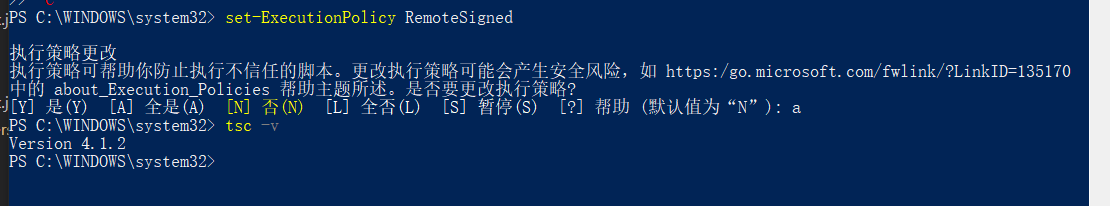
**解决:**

以管理员身份运行powerShell(****一定要以管路员身份打开****):

命令行:

set-ExecutionPolicy RemoteSigned

**敲 A 或者 Y 即可**



### TypeScript 基础语法

**tsc 常用编译参数**

|  |  |
| --- | --- |
| 1. | **--help**  显示帮助信息 |
| 2. | **--module**  载入扩展模块 |
| 3. | **--target**  设置 ECMA 版本 |
| 4. | **--declaration**  额外生成一个 .d.ts 扩展名的文件。  tsc ts-hw.ts --declaration  以上命令会生成 ts-hw.d.ts、ts-hw.js 两个文件。 |
| 5. | **--removeComments**  删除文件的注释 |
| 6. | **--out**  编译多个文件并合并到一个输出的文件 |
| 7. | **--sourcemap**  生成一个 sourcemap (.map) 文件。  sourcemap 是一个存储源代码与编译代码对应位置映射的信息文件。 |
| 8. | **--module noImplicitAny**  在表达式和声明上有隐含的 any 类型时报错 |
| 9. | **--watch**  在监视模式下运行编译器。会监视输出文件，在它们改变时重新编译。 |

#### **忽略空白和换行**

TypeScript 会忽略程序中出现的空格、制表符和换行符。

空格、制表符通常用来缩进代码，使代码易于阅读和理解。

#### **TypeScript 区分大小写**

TypeScript 区分大写和小写字符。

#### **TypeScript 注释**

注释是一个良好的习惯，虽然很多程序员讨厌注释，但还是建议你在每段代码写上文字说明。

注释可以提高程序的可读性。

注释可以包含有关程序一些信息，如代码的作者，有关函数的说明等。

编译器会忽略注释。

**TypeScript 支持两种类型的注释**

**单行注释 ( // )** − 在 // 后面的文字都是注释内容。

**多行注释 (/\* \*/)** − 这种注释可以跨越多行。

#### TypeScript 与面向对象

* **对象**：对象是类的一个实例（**对象不是找个女朋友**），有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
* **类**：类是一个模板，它描述一类对象的行为和状态。
* **方法**：方法是类的操作的实现步骤。

TypeScript 面向对象编程实例：

class Site { name():void { console.log("Runoob") } } var obj = new Site(); obj.name();

以上实例定义了一个类 Site，该类有一个方法 name()，该方法在终端上输出字符串 Runoob。

new 关键字创建类的对象，该对象调用方法 name()。

编译后生成的 JavaScript 代码如下：

var Site = /\*\* @class \*/ (function () { function Site() { } Site.prototype.name = function () { console.log("Runoob"); }; return Site; }());

var obj = new Site(); obj.name();

执行以上 JavaScript 代码，输出结果如下:

Runoob

### TypeScript 基础类型

**TypeScript 包含的数据类型如下表:**

|  |  |  |
| --- | --- | --- |
| **数据类型** | **关键字** | **描述** |
| 任意类型 | any | 声明为 any 的变量可以赋予任意类型的值。 |
| 数字类型 | number | 双精度 64 位浮点值。它可以用来表示整数和分数。  let binaryLiteral: number = 0b1010; // 二进制let octalLiteral: number = 0o744; // 八进制let decLiteral: number = 6; // 十进制let hexLiteral: number = 0xf00d; // 十六进制 |
| 字符串类型 | string | 一个字符系列，使用单引号（**'**）或双引号（**"**）来表示字符串类型。反引号（**`**）来定义多行文本和内嵌表达式。  let name: string = "Runoob";let years: number = 5;let words: string = `您好，今年是 ${ name } 发布 ${ years + 1} 周年`; |
| 布尔类型 | boolean | 表示逻辑值：true 和 false。  let flag: boolean = true; |
| 数组类型 | 无 | 声明变量为数组。  // 在元素类型后面加上[]let arr: number[] = [1, 2];  // 或者使用数组泛型let arr: Array<number> = [1, 2]; |
| 元组 | 无 | 元组类型用来表示已知元素数量和类型的数组，各元素的类型不必相同，对应位置的类型需要相同。  let x: [string, number];  x = ['Runoob', 1]; // 运行正常  x = [1, 'Runoob']; // 报错  console.log(x[0]); // 输出 Runoob |
| 枚举 | enum | 枚举类型用于定义数值集合。  enum Color {Red, Green, Blue};let c: Color = Color.Blue;  console.log(c); // 输出 2 |
| void | void | 用于标识方法返回值的类型，表示该方法没有返回值。  function hello(): void {  alert("Hello Runoob");} |
| null | null | 表示对象值缺失。 |
| undefined | undefined | 用于初始化变量为一个未定义的值 |
| never | never | never 是其它类型（包括 null 和 undefined）的子类型，代表从不会出现的值。 |

****注意：**TypeScript 和 JavaScript 没有整数类型。**

#### Any 类型

**任意值是 TypeScript 针对编程时类型不明确的变量使用的一种数据类型，**它常用于以下三种情况。

1、**变量的值会动态改变时**，比如来自用户的输入，任意值类型可以让这些变量跳过编译阶段的类型检查，示例代码如下：

let x: any = 1; // 数字类型

x = 'I am who I am'; // 字符串类型

x = false; // 布尔类型

2、改写现有代码时，任意值允许在编译时可选择地包含或移除类型检查，示例代码如下：

let x: any = 4;

x.ifItExists(); // 正确，ifItExists方法在运行时可能存在，但这里并不会检查

x.toFixed(); // 正确

3、定义存储各种类型数据的数组

let arrayList: any[] = [1, false, 'fine'];

arrayList[1] = 100;

**定义存储各种类型数据的数组**

//定义存储各种类型数据的数组

let arr: any[] = [1, 2,[1,2], false, 'fine'];

#### Null 和 Undefined

##### **null**

在 JavaScript 中 null 表示 "什么都没有"。

null是一个只有一个值的特殊类型。表示一个空对象引用。

用 typeof 检测 null 返回是 object。

##### **undefined**

在 JavaScript 中, undefined 是一个没有设置值的变量。

typeof 一个没有值的变量会返回 undefined。

Null 和 Undefined 是其他任何类型（包括 void）的子类型，可以赋值给其它类型，如数字类型，此时，赋值后的类型会变成 null 或 undefined。而**在TypeScript中启用严格的空校验（--strictNullChecks）特性**，就可以使得null 和 undefined 只能被赋值给 void 或本身对应的类型，示例代码如下：

// 启用 --strictNullChecks

let x: number;

x = 1; // 运行正确

x = undefined; // 运行错误

x = null; // 运行错误

上面的例子中变量 x 只能是数字类型。如果一个类型可能出现 null 或 undefined， 可以用 | 来支持多种类型，示例代码如下：

支持多种类型，可以用 | 来支持多种类型，

// 启用 --strictNullChecks

let x: number | null | undefined;

x = 1; // 运行正确

x = undefined; // 运行正确

x = null; // 运行正确

#### never 类型

never 是其它类型（包括 null 和 undefined）的子类型，代表从不会出现的值。这意味着声明为 never 类型的变量只能被 never 类型所赋值，在函数中它通常表现为抛出异常或无法执行到终止点（例如无限循环），示例代码如下：

let x: never;let y: number;

// 运行错误，数字类型不能转为 never 类型

x = 123;

// 运行正确，never 类型可以赋值给 never类型

x = (()=>{ throw new Error('exception')})();

// 运行正确，never 类型可以赋值给 数字类型

y = (()=>{ throw new Error('exception')})();

// 返回值为 never 的函数可以是抛出异常的情况

function error(message: string): never {

throw new Error(message);}

// 返回值为 never 的函数可以是无法被执行到的终止点的情况

function loop(): never {

while (true) {}}

### TypeScript 变量声明

变量是一种使用方便的占位符，用于引用计算机内存地址。

我们可以把变量看做存储数据的容器。

TypeScript 变量的命名规则：

变量名称可以包含数字和字母。

除了下划线 **\_** 和美元 **$** 符号外，不能包含其他特殊字符，包括空格。

变量名不能以数字开头。

变量使用前必须先声明，我们可以使用 var 来声明变量。

我们可以使用以下四种方式来声明变量：

声明变量的类型及初始值：

var [变量名] : [类型] = 值;

例如：

var uname:string = "Runoob";

声明变量的类型，但没有初始值，变量值会设置为 undefined：

var [变量名] : [类型];

例如：

var uname:string;

声明变量并初始值，但不设置类型，该变量可以是任意类型：

var [变量名] = 值;

例如：

var uname = "Runoob";

声明变量没有设置类型和初始值，类型可以是任意类型，默认初始值为 undefined：

var [变量名];

例如：

var uname;

## TypeScript 函数

### 函数返回值

有时，我们会希望函数将执行的结果返回到调用它的地方。

通过使用 return 语句就可以实现。

在使用 return 语句时，函数会停止执行，并返回指定的值。

语法格式如下所示：

function function\_name():return\_type {

// 语句

return value; }

return\_type 是返回值的类型。

return 关键词后跟着要返回的结果。

一个函数只能有一个 return 语句。

返回值的类型需要与函数定义的返回类型(return\_type)一致。

**实例**

TypeScript

// 函数定义 function greet():string { // 返回一个字符串 return "Hello World" }

function caller() { var msg = greet() // 调用 greet() 函数 console.log(msg) }

// 调用函数 caller()

实例中定义了函数 *greet()*，返回值的类型为 string。

*greet()* 函数通过 return 语句返回给调用它的地方，即变量 msg，之后输出该返回值。。

编译以上代码，得到以下 JavaScript 代码：

JavaScript

// 函数定义 function greet() { return "Hello World"; }

function caller() { var msg = greet(); // 调用 greet() 函数 console.log(msg); }

// 调用函数 caller();

### 带参数函数

在调用函数时，您可以向其传递值，这些值被称为参数。

这些参数可以在函数中使用。

您可以向函数发送多个参数，每个参数使用逗号 **,** 分隔：

语法格式如下所示：

function func\_name( param1 [:datatype], param2 [:datatype]) { }

param1、param2 为参数名。

datatype 为参数类型。

**实例**

TypeScript

function add(x: number, y: number): number { return x + y; } console.log(add(1,2))

实例中定义了函数 *add()*，返回值的类型为 number。

*add()* 函数中定义了两个 number 类型的参数，函数内将两个参数相加并返回。

编译以上代码，得到以下 JavaScript 代码：

JavaScript

function add(x, y) { return x + y; } console.log(add(1, 2));

输出结果为：

3

### 可选参数和默认参数

#### **可选参数**

在 TypeScript 函数里，如果我们定义了参数，则我们必须传入这些参数，除非将这些参数设置为可选，**可选参数使用问号标识 ？。**

**实例**

TypeScript

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

let result1 = buildName("Bob"); // 错误，缺少参数

let result2 = buildName("Bob", "Adams", "Sr."); // 错误，参数太多了

let result3 = buildName("Bob", "Adams"); // 正确

以下实例，我么将 lastName 设置为可选参数：

TypeScript

function buildName(firstName: string, lastName?: string) { if (lastName) return firstName + " " + lastName; else return firstName; } let result1 = buildName("Bob"); // 正确 let result2 = buildName("Bob", "Adams", "Sr."); // 错误，参数太多了 let result3 = buildName("Bob", "Adams"); // 正确

可选参数必须跟在必需参数后面。 如果上例我们想让 firstName 是可选的，lastName 必选，那么就要调整它们的位置，把 firstName 放在后面。

如果都是可选参数就没关系。

#### **默认参数**

我们也可以设置参数的默认值，这样在调用函数的时候，如果不传入该参数的值，则使用默认参数，语法格式为：

function function\_name(param1[:type],param2[:type] = default\_value) { }

<p**注意：参数不能同时设置为可选和默认。**

**实例**

以下实例函数的参数 rate 设置了默认值为 0.50，调用该函数时如果未传入参数则使用该默认值：

**TypeScript**

function calculate\_discount(price:number,rate:number = 0.50) { var discount = price \* rate; console.log("计算结果: ",discount); }

calculate\_discount(1000)

calculate\_discount(1000,0.30)

编译以上代码，得到以下 JavaScript 代码：

**JavaScript**

function calculate\_discount(price, rate) { if (rate === void 0) { rate = 0.50; } var discount = price \* rate; console.log("计算结果: ", discount); }

calculate\_discount(1000);

calculate\_discount(1000, 0.30);

输出结果为：

计算结果: 500计算结果: 300

### ****剩余参数****

有一种情况，我们不知道要向函数传入多少个参数，这时候我们就可以使用剩余参数来定义。

剩余参数语法允许我们将一个不确定数量的参数作为一个数组传入。

**TypeScript**

function buildName(firstName: string, ...restOfName: string[]) { return firstName + " " + restOfName.join(" "); } let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");

函数的最后一个命名参数 restOfName 以 ... 为前缀，它将成为一个由剩余参数组成的数组，索引值从0（包括）到 restOfName.length（不包括）。

**TypeScript**

function addNumbers(...nums:number[]) { var i; var sum:number = 0; for(i = 0;i<nums.length;i++) { sum = sum + nums[i]; } console.log("和为：",sum) } addNumbers(1,2,3) addNumbers(10,10,10,10,10)

编译以上代码，得到以下 JavaScript 代码：

**JavaScript**

function addNumbers() { var nums = []; for (var \_i = 0; \_i < arguments.length; \_i++) { nums[\_i] = arguments[\_i]; } var i; var sum = 0; for (i = 0; i < nums.length; i++) { sum = sum + nums[i]; } console.log("和为：", sum); } addNumbers(1, 2, 3); addNumbers(10, 10, 10, 10, 10);

输出结果为：

和为： 6和为： 50

### 匿名函数

匿名函数是一个没有函数名的函数。

匿名函数在程序运行时动态声明，除了没有函数名外，其他的与标准函数一样。

我们可以将匿名函数赋值给一个变量，这种表达式就成为函数表达式。

语法格式如下：

var res = function( [arguments] ) { ... }

**实例**

不带参数匿名函数：

TypeScript

var msg = function() { return "hello world"; } console.log(msg())

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var msg = function () { return "hello world"; }; console.log(msg());

输出结果为：

hello world

带参数匿名函数：

TypeScript

var res = function(a:number,b:number) { return a\*b; }; console.log(res(12,2))

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var res = function (a, b) { return a \* b; }; console.log(res(12, 2));

输出结果为：

24

### **匿名函数自调用**

**匿名函数自调用在函数后使用 () 即可：**

TypeScript

(function () { var x = "Hello!!"; console.log(x) })()

编译以上代码，得到以下 JavaScript 代码：

JavaScript

(function () { var x = "Hello!!"; console.log(x) })()

输出结果为：

Hello!!

### 构造函数

TypeScript 也支持使用 JavaScript 内置的构造函数 Function() 来定义函数：

语法格式如下：

var res = new Function ([arg1[, arg2[, ...argN]],] functionBody)

参数说明：

* **arg1, arg2, ... argN**：参数列表。
* **functionBody**：一个含有包括函数定义的 JavaScript 语句的字符串。

**实例**

TypeScript

var myFunction = new Function("a", "b", "return a \* b");

var x = myFunction(4, 3);

console.log(x);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var myFunction = new Function("a", "b", "return a \* b");

var x = myFunction(4, 3);

console.log(x);

输出结果为：

12

### 递归函数

递归函数即在函数内调用函数本身

**实例**

TypeScript

function factorial(number) {

if (number <= 0) { // 停止执行 return 1; } else { return (number \* factorial(number - 1)); // 调用自身 } };

console.log(factorial(6)); // 输出 720

编译以上代码，得到以下 JavaScript 代码：

JavaScript

function factorial(number) { if (number <= 0) { // 停止执行 return 1; } else { return (number \* factorial(number - 1)); // 调用自身 } } ; console.log(factorial(6)); // 输出 720

输出结果为：

720

### Lambda 函数

Lambda 函数也称之为箭头函数。

箭头函数表达式的语法比函数表达式更短。

函数只有一行语句：

( [param1, parma2,…param n] )=>statement;

**实例**

以下实例声明了 lambda 表达式函数，函数返回两个数的和：

TypeScript

var foo = (x:number)=>10 + x console.log(foo(100)) //输出结果为 110

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var foo = function (x) { return 10 + x; }; console.log(foo(100)); //输出结果为 110

输出结果为：

110

函数是一个语句块：

( [param1, parma2,…param n] )=> {

// 代码块}

**实例**

以下实例声明了 lambda 表达式函数，函数返回两个数的和：

TypeScript

var foo = (x:number)=> { x = 10 + x console.log(x) } foo(100)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var foo = function (x) { x = 10 + x; console.log(x); }; foo(100);

输出结果为：

110

**我们可以不指定函数的参数类型，通过函数内来推断参数类型:**

TypeScript

var func = (x)=> { if(typeof x=="number") { console.log(x+" 是一个数字") } else if(typeof x=="string") { console.log(x+" 是一个字符串") } } func(12) func("Tom")

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var func = function (x) { if (typeof x == "number") { console.log(x + " 是一个数字"); } else if (typeof x == "string") { console.log(x + " 是一个字符串"); } }; func(12); func("Tom");

输出结果为：

12 是一个数字Tom 是一个字符串

单个参数 **()** 是可选的：

TypeScript

var display = x => { console.log("输出为 "+x) } display(12)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var display = function (x) { console.log("输出为 " + x); }; display(12);

输出结果为：

输出为 12

无参数时可以设置空括号：

TypeScript

var disp =()=> { console.log("Function invoked"); } disp();

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var disp = function () { console.log("调用函数"); }; disp();

输出结果为：

调用函数

### 函数重载

重载是方法名字相同，而参数不同，返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

参数类型不同：

function disp(string):void;

function disp(number):void;

参数数量不同：

function disp(n1:number):void;

function disp(x:number,y:number):void;

参数类型顺序不同：

function disp(n1:number,s1:string):void;

function disp(s:string,n:number):void;

**如果参数类型不同，则参数类型应设置为 **any**。**

**参数数量不同你可以将不同的参数设置为可选。**

**实例**

以下实例定义了参数类型与参数数量不同：

TypeScript

function disp(s1:string):void;

function disp(n1:number,s1:string):void;

function disp(x:any,y?:any):void { console.log(x); console.log(y); }

disp("abc") disp(1,"xyz");

编译以上代码，得到以下 JavaScript 代码：

JavaScript

function disp(x, y) { console.log(x); console.log(y); }

disp("abc"); disp(1, "xyz");

输出结果为：

abc undefined

1 xyz

## TypeScript Number

TypeScript 与 JavaScript 类似，支持 Number 对象。

Number 对象是原始数值的包装对象。

### **语法**

var num = new Number(value);

**注意：** 如果一个参数值不能转换为一个数字将返回 NaN (非数字值)。

### **Number 对象属性**

下表列出了 Number 对象支持的属性：

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1. | **MAX\_VALUE**  可表示的最大的数，MAX\_VALUE 属性值接近于 1.79E+308。大于 MAX\_VALUE 的值代表 "Infinity"。 |
| 2. | **MIN\_VALUE**  可表示的最小的数，即最接近 0 的正数 (实际上不会变成 0)。最大的负数是 -MIN\_VALUE，MIN\_VALUE 的值约为 5e-324。小于 MIN\_VALUE ("underflow values") 的值将会转换为 0。 |
| 3. | **NaN**  非数字值（Not-A-Number）。 |
| 4. | **NEGATIVE\_INFINITY**  负无穷大，溢出时返回该值。该值小于 MIN\_VALUE。 |
| 5. | **POSITIVE\_INFINITY**  正无穷大，溢出时返回该值。该值大于 MAX\_VALUE。 |
| 6. | **prototype**  Number 对象的静态属性。使您有能力向对象添加属性和方法。 |
| 7. | **constructor**  返回对创建此对象的 Number 函数的引用。 |

TypeScript

console.log("TypeScript Number 属性: ");

console.log("最大值为: " + Number.MAX\_VALUE);

console.log("最小值为: " + Number.MIN\_VALUE);

console.log("负无穷大: " + Number.NEGATIVE\_INFINITY);

console.log("正无穷大:" + Number.POSITIVE\_INFINITY);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

console.log("TypeScript Number 属性: "); console.log("最大值为: " + Number.MAX\_VALUE); console.log("最小值为: " + Number.MIN\_VALUE); console.log("负无穷大: " + Number.NEGATIVE\_INFINITY); console.log("正无穷大:" + Number.POSITIVE\_INFINITY);

输出结果为：

TypeScript Number 属性:最大值为: 1.7976931348623157e+308最小值为: 5e-324负无穷大: -Infinity正无穷大:Infinity

### Number 对象方法

Number对象 支持以下方法：

|  |  |  |
| --- | --- | --- |
| **序号** | **方法 & 描述** | **实例** |
| 1. | toExponential()  把对象的值转换为指数计数法。 | //toExponential() var num1 = 1225.30 var val = num1.toExponential();  console.log(val) // 输出： 1.2253e+3 |
| 2. | toFixed()  把数字转换为字符串，并对小数点指定位数。 | var num3 = 177.234  console.log("num3.toFixed() 为 "+num3.toFixed()) // 输出：177  console.log("num3.toFixed(2) 为 "+num3.toFixed(2)) // 输出：177.23  console.log("num3.toFixed(6) 为 "+num3.toFixed(6)) // 输出：177.234000 |
| 3. | toLocaleString()  把数字转换为字符串，使用本地数字格式顺序。 | var num = new Number(177.1234);  console.log( num.toLocaleString()); // 输出：177.1234 |
| 4. | toPrecision()  把数字格式化为指定的长度。 | var num = new Number(7.123456);  console.log(num.toPrecision()); // 输出：7.123456  console.log(num.toPrecision(1)); // 输出：7  console.log(num.toPrecision(2)); // 输出：7.1 |
| 5. | toString()  把数字转换为字符串，使用指定的基数。数字的基数是 2 ~ 36 之间的整数。若省略该参数，则使用基数 10。 | var num = new Number(10);  console.log(num.toString()); // 输出10进制：10  console.log(num.toString(2)); // 输出2进制：1010  console.log(num.toString(8)); // 输出8进制：12 |
| 6. | valueOf()  返回一个 Number 对象的原始数字值。 | var num = new Number(10);  console.log(num.valueOf()); // 输出：10 |

## TypeScript String（字符串）

String 对象用于处理文本（字符串）。

### **语法**

var txt = new String("string");或者更简单方式：var txt = "string";

### **String 对象属性**

下表列出了 String 对象支持的属性：

|  |  |  |
| --- | --- | --- |
| **序号** | **属性 & 描述** | **实例** |
| 1. | constructor  对创建该对象的函数的引用。 | var str = new String( "This is string" );  console.log("str.constructor is:" + str.constructor)  输出结果：  str.constructor is:function String() { [native code] } |
| 2. | length  返回字符串的长度。 | var uname = new String("Hello World")  console.log("Length "+uname.length) // 输出 11 |
| 3. | prototype  允许您向对象添加属性和方法。 | function employee(id:number,name:string) {  this.id = id  this.name = name  }  var emp = new employee(123,"admin")  employee.prototype.email="admin@runoob.com" // 添加属性 email  console.log("员工号: "+emp.id)  console.log("员工姓名: "+emp.name)  console.log("员工邮箱: "+emp.email) |

### **String 方法**

下表列出了 String 对象支持的方法：

|  |  |  |
| --- | --- | --- |
| **序号** | **方法 & 描述** | **实例** |
| 1. | charAt()  返回在指定位置的字符。 | var str = new String("RUNOOB");  console.log("str.charAt(0) 为:" + str.charAt(0)); // R  console.log("str.charAt(1) 为:" + str.charAt(1)); // U  console.log("str.charAt(2) 为:" + str.charAt(2)); // N  console.log("str.charAt(3) 为:" + str.charAt(3)); // O  console.log("str.charAt(4) 为:" + str.charAt(4)); // O  console.log("str.charAt(5) 为:" + str.charAt(5)); // B |
| 2. | charCodeAt()  返回在指定的位置的字符的 Unicode 编码。 | var str = new String("RUNOOB");  console.log("str.charCodeAt(0) 为:" + str.charCodeAt(0)); // 82  console.log("str.charCodeAt(1) 为:" + str.charCodeAt(1)); // 85  console.log("str.charCodeAt(2) 为:" + str.charCodeAt(2)); // 78  console.log("str.charCodeAt(3) 为:" + str.charCodeAt(3)); // 79  console.log("str.charCodeAt(4) 为:" + str.charCodeAt(4)); // 79  console.log("str.charCodeAt(5) 为:" + str.charCodeAt(5)); // 66 |
| 3. | concat()  连接两个或更多字符串，并返回新的字符串。 | var str1 = new String( "RUNOOB" ); var str2 = new String( "GOOGLE" ); var str3 = str1.concat( str2 );  console.log("str1 + str2 : "+str3) // RUNOOBGOOGLE |
| 4. | indexOf()  返回某个指定的字符串值在字符串中首次出现的位置。 | var str1 = new String( "RUNOOB" );  var index = str1.indexOf( "OO" );  console.log("查找的字符串位置 :" + index ); // 3 |
| 5. | lastIndexOf()  从后向前搜索字符串，并从起始位置（0）开始计算返回字符串最后出现的位置。 | var str1 = new String( "This is string one and again string" ); var index = str1.lastIndexOf( "string" );  console.log("lastIndexOf 查找到的最后字符串位置 :" + index ); // 29    index = str1.lastIndexOf( "one" );  console.log("lastIndexOf 查找到的最后字符串位置 :" + index ); // 15 |
| 6. | localeCompare()  用本地特定的顺序来比较两个字符串。 | var str1 = new String( "This is beautiful string" );  var index = str1.localeCompare( "This is beautiful string");  console.log("localeCompare first :" + index ); // 0 |
| 7. | **match()**  查找找到一个或多个正则表达式的匹配。 | var str="The rain in SPAIN stays mainly in the plain"; var n=str.match(/ain/g); // ain,ain,ain |
| 8. | replace()  替换与正则表达式匹配的子串 | var re = /(\w+)\s(\w+)/; var str = "zara ali"; var newstr = str.replace(re, "$2, $1");  console.log(newstr); // ali, zara |
| 9. | search()  检索与正则表达式相匹配的值 | var re = /apples/gi; var str = "Apples are round, and apples are juicy.";if (str.search(re) == -1 ) {  console.log("Does not contain Apples" ); } else {  console.log("Contains Apples" ); } |
| 10. | slice()  提取字符串的片断，并在新的字符串中返回被提取的部分。 |  |
| 11. | split()  把字符串分割为子字符串数组。 | var str = "Apples are round, and apples are juicy."; var splitted = str.split(" ", 3);  console.log(splitted) // [ 'Apples', 'are', 'round,' ] |
| 12. | substr()  从起始索引号提取字符串中指定数目的字符。 |  |
| 13. | substring()  提取字符串中两个指定的索引号之间的字符。 | var str = "RUNOOB GOOGLE TAOBAO FACEBOOK";  console.log("(1,2): " + str.substring(1,2)); // U  console.log("(0,10): " + str.substring(0, 10)); // RUNOOB GOO  console.log("(5): " + str.substring(5)); // B GOOGLE TAOBAO FACEBOOK |
| 14. | toLocaleLowerCase()  根据主机的语言环境把字符串转换为小写，只有几种语言（如土耳其语）具有地方特有的大小写映射。 | var str = "Runoob Google";  console.log(str.toLocaleLowerCase( )); // runoob google |
| 15. | toLocaleUpperCase()  据主机的语言环境把字符串转换为大写，只有几种语言（如土耳其语）具有地方特有的大小写映射。 | var str = "Runoob Google";  console.log(str.toLocaleUpperCase( )); // RUNOOB GOOGLE |
| 16. | toLowerCase()  把字符串转换为小写。 | var str = "Runoob Google";  console.log(str.toLowerCase( )); // runoob google |
| 17. | toString()  返回字符串。 | var str = "Runoob";  console.log(str.toString( )); // Runoob |
| 18. | toUpperCase()  把字符串转换为大写。 | var str = "Runoob Google";  console.log(str.toUpperCase( )); // RUNOOB GOOGLE |
| 19. | valueOf()  返回指定字符串对象的原始值。 | var str = new String("Runoob");  console.log(str.valueOf( )); // Runoob |

# TypeScript Array(数组)

数组对象是使用单独的变量名来存储一系列的值。

数组非常常用。

假如你有一组数据（例如：网站名字），存在单独变量如下所示：

var site1="Google"; var site2="Runoob"; var site3="Taobao";

如果有 10 个、100 个这种方式就变的很不实用，这时我们可以使用数组来解决：

var sites:string[]; sites = ["Google","Runoob","Taobao"]

这样看起来就简洁多了。

TypeScript 声明数组的语法格式如下所示：

var array\_name[:datatype]; //声明

array\_name = [val1,val2,valn..] //初始化

或者直接在声明时初始化：

var array\_name[:data type] = [val1,val2…valn]

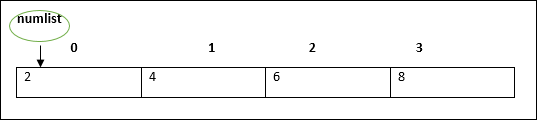
**如果数组声明时未设置类型，则会被认为是 any 类型，在初始化时根据第一个元素的类型来推断数组的类型。**

### **实例**

创建一个 number 类型的数组：

var numlist:number[] = [2,4,6,8]

整个数组结构如下所示：



索引值第一个为 0，我们可以根据索引值来访问数组元素：

TypeScript

var sites:string[]; sites = ["Google","Runoob","Taobao"] console.log(sites[0]); console.log(sites[1]);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites; sites = ["Google", "Runoob", "Taobao"]; console.log(sites[0]); console.log(sites[1]);

输出结果为：

GoogleRunoob

以下实例我们在声明时直接初始化：

TypeScript

var nums:number[] = [1,2,3,4] console.log(nums[0]); console.log(nums[1]); console.log(nums[2]); console.log(nums[3]);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var nums = [1, 2, 3, 4]; console.log(nums[0]); console.log(nums[1]); console.log(nums[2]); console.log(nums[3]);

输出结果为：

1 2 3 4

### Array 对象

我们也可以使用 Array 对象创建数组。

Array 对象的构造函数接受以下两种值：

* 表示数组大小的数值。
* 初始化的数组列表，元素使用逗号分隔值。

**实例**

指定数组初始化大小：

TypeScript

var arr\_names:number[] = new Array(4) for(var i = 0; i<arr\_names.length; i++) { arr\_names[i] = i \* 2 console.log(arr\_names[i]) }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var arr\_names = new Array(4); for (var i = 0; i < arr\_names.length; i++) { arr\_names[i] = i \* 2; console.log(arr\_names[i]); }

输出结果为：

0246

以下实例我们直接初始化数组元素：

TypeScript

var sites:string[] = new Array("Google","Runoob","Taobao","Facebook") for(var i = 0;i<sites.length;i++) { console.log(sites[i]) }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites = new Array("Google", "Runoob", "Taobao", "Facebook"); for (var i = 0; i < sites.length; i++) { console.log(sites[i]); }

输出结果为：

GoogleRunoobTaobaoFacebook

### 数组解构

我们也可以把数组元素赋值给变量，如下所示：

TypeScript

var arr:number[] = [12,13] var[x,y] = arr // 将数组的两个元素赋值给变量 x 和 y console.log(x) console.log(y)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var arr = [12, 13]; var x = arr[0], y = arr[1]; // 将数组的两个元素赋值给变量 x 和 y console.log(x); console.log(y);

输出结果为：

1213

### 数组迭代

我们可以使用 for 语句来循环输出数组的各个元素：

TypeScript

var j:any; var nums:number[] = [1001,1002,1003,1004] for(j in nums) { console.log(nums[j]) }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var j; var nums = [1001, 1002, 1003, 1004]; for (j in nums) { console.log(nums[j]); }

输出结果为：

1001100210031004

### 多维数组

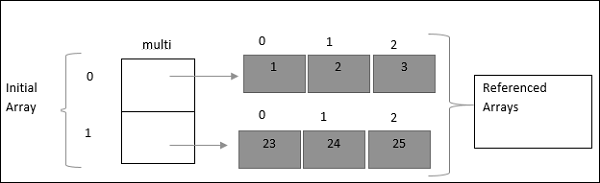
一个数组的元素可以是另外一个数组，这样就构成了多维数组（Multi-dimensional Array）。

最简单的多维数组是二维数组，定义方式如下：

var arr\_name:datatype[][]=[ [val1,val2,val3],[v1,v2,v3] ]

**实例**

定义一个二维数组，每一个维度的数组有三个元素。



TypeScript

var multi:number[][] = [[1,2,3],[23,24,25]] console.log(multi[0][0]) console.log(multi[0][1]) console.log(multi[0][2]) console.log(multi[1][0]) console.log(multi[1][1]) console.log(multi[1][2])

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var multi = [[1, 2, 3], [23, 24, 25]]; console.log(multi[0][0]); console.log(multi[0][1]); console.log(multi[0][2]); console.log(multi[1][0]); console.log(multi[1][1]); console.log(multi[1][2]);

输出结果为：

123232425

### 数组在函数中的使用

#### **作为参数传递给函数**

TypeScript

var sites:string[] = new Array("Google","Runoob","Taobao","Facebook") function disp(arr\_sites:string[]) { for(var i = 0;i<arr\_sites.length;i++) { console.log(arr\_sites[i]) } } disp(sites);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites = new Array("Google", "Runoob", "Taobao", "Facebook"); function disp(arr\_sites) { for (var i = 0; i < arr\_sites.length; i++) { console.log(arr\_sites[i]); } } disp(sites);

输出结果为：

GoogleRunoobTaobaoFacebook

#### **作为函数的返回值**

TypeScript

function disp():string[] { return new Array("Google", "Runoob", "Taobao", "Facebook"); } var sites:string[] = disp() for(var i in sites) { console.log(sites[i]) }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

function disp() { return new Array("Google", "Runoob", "Taobao", "Facebook"); } var sites = disp(); for (var i in sites) { console.log(sites[i]); }

输出结果为：

GoogleRunoobTaobaoFacebook

### 数组方法

下表列出了一些常用的数组方法：

|  |  |  |
| --- | --- | --- |
| **序号** | **方法 & 描述** | **实例** |
| 1. | concat()  连接两个或更多的数组，并返回结果。 | var alpha = ["a", "b", "c"]; var numeric = [1, 2, 3];  var alphaNumeric = alpha.concat(numeric);  console.log("alphaNumeric : " + alphaNumeric ); // a,b,c,1,2,3 |
| 2. | every()  检测数值元素的每个元素是否都符合条件。 | function isBigEnough(element, index, array) {  return (element >= 10); }  var passed = [12, 5, 8, 130, 44].every(isBigEnough);  console.log("Test Value : " + passed ); // false |
| 3. | filter()  检测数值元素，并返回符合条件所有元素的数组。 | function isBigEnough(element, index, array) {  return (element >= 10); }  var passed = [12, 5, 8, 130, 44].filter(isBigEnough);  console.log("Test Value : " + passed ); // 12,130,44 |
| 4. | forEach()  数组每个元素都执行一次回调函数。 | let num = [7, 8, 9];  num.forEach(function (value) {  console.log(value);});  编译成 JavaScript 代码：  var num = [7, 8, 9];  num.forEach(function (value) {  console.log(value); // 7 8 9}); |
| 5. | indexOf()  搜索数组中的元素，并返回它所在的位置。  如果搜索不到，返回值 -1，代表没有此项。 | var index = [12, 5, 8, 130, 44].indexOf(8);  console.log("index is : " + index ); // 2 |
| 6. | join()  把数组的所有元素放入一个字符串。 | var arr = new Array("First","Second","Third");  var str = arr.join();  console.log("str : " + str ); // First,Second,Third  var str = arr.join(", ");  console.log("str : " + str ); // First, Second, Third  var str = arr.join(" + ");  console.log("str : " + str ); // First + Second + Third |
| 7. | lastIndexOf()  返回一个指定的字符串值最后出现的位置，在一个字符串中的指定位置从后向前搜索。 | var index = [12, 5, 8, 130, 44].lastIndexOf(8);  console.log("index is : " + index ); // 2 |
| 8. | map()  通过指定函数处理数组的每个元素，并返回处理后的数组。 | var numbers = [1, 4, 9]; var roots = numbers.map(Math.sqrt);  console.log("roots is : " + roots ); // 1,2,3 |
| 9. | pop()  删除数组的最后一个元素并返回删除的元素。 | var numbers = [1, 4, 9];  var element = numbers.pop();  console.log("element is : " + element ); // 9  var element = numbers.pop();  console.log("element is : " + element ); // 4 |
| 10. | push()  向数组的末尾添加一个或更多元素，并返回新的长度。 | var numbers = new Array(1, 4, 9); var length = numbers.push(10);  console.log("new numbers is : " + numbers ); // 1,4,9,10  length = numbers.push(20);  console.log("new numbers is : " + numbers ); // 1,4,9,10,20 |
| 11. | reduce()  将数组元素计算为一个值（从左到右）。 | var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; });  console.log("total is : " + total ); // 6 |
| 12. | reduceRight()  将数组元素计算为一个值（从右到左）。 | var total = [0, 1, 2, 3].reduceRight(function(a, b){ return a + b; });  console.log("total is : " + total ); // 6 |
| 13. | reverse()  反转数组的元素顺序。 | var arr = [0, 1, 2, 3].reverse();  console.log("Reversed array is : " + arr ); // 3,2,1,0 |
| 14. | shift()  删除并返回数组的第一个元素。 | var arr = [10, 1, 2, 3].shift();  console.log("Shifted value is : " + arr ); // 10 |
| 15. | slice()  选取数组的的一部分，并返回一个新数组。 | var arr = ["orange", "mango", "banana", "sugar", "tea"];  console.log("arr.slice( 1, 2) : " + arr.slice( 1, 2) ); // mango  console.log("arr.slice( 1, 3) : " + arr.slice( 1, 3) ); // mango,banana |
| 16. | some()  检测数组元素中是否有元素符合指定条件。 | function isBigEnough(element, index, array) {  return (element >= 10);  }  var retval = [2, 5, 8, 1, 4].some(isBigEnough);  console.log("Returned value is : " + retval ); // false  var retval = [12, 5, 8, 1, 4].some(isBigEnough);  console.log("Returned value is : " + retval ); // true |
| 17. | sort()  对数组的元素进行排序。 | var arr = new Array("orange", "mango", "banana", "sugar"); var sorted = arr.sort();  console.log("Returned string is : " + sorted ); // banana,mango,orange,sugar |
| 18. | splice()  从数组中添加或删除元素。 | var arr = ["orange", "mango", "banana", "sugar", "tea"]; var removed = arr.splice(2, 0, "water");  console.log("After adding 1: " + arr ); // orange,mango,water,banana,sugar,tea  console.log("removed is: " + removed);    removed = arr.splice(3, 1);  console.log("After removing 1: " + arr ); // orange,mango,water,sugar,tea  console.log("removed is: " + removed); // banana |
| 19. | toString()  把数组转换为字符串，并返回结果。 | var arr = new Array("orange", "mango", "banana", "sugar"); var str = arr.toString();  console.log("Returned string is : " + str ); // orange,mango,banana,sugar |
| 20. | unshift()  向数组的开头添加一个或更多元素，并返回新的长度。 | var arr = new Array("orange", "mango", "banana", "sugar"); var length = arr.unshift("water");  console.log("Returned array is : " + arr ); // water,orange,mango,banana,sugar  console.log("Length of the array is : " + length ); // 5 |

## TypeScript Map 对象

**Map 对象保存键值对，并且能够记住键的原始插入顺序。**

**任何值(对象或者原始值) 都可以作为一个键或一个值。**

Map 是 ES6 中引入的一种新的数据结构，可以参考 [ES6 Map 与 Set](https://www.runoob.com/w3cnote/es6-map-set.html" \t "https://www.runoob.com/typescript/_blank)。

### 创建 Map

TypeScript 使用 Map 类型和 new 关键字来创建 Map：

let myMap = new Map();

初始化 Map，可以以数组的格式来传入键值对：

let myMap = new Map([

["key1", "value1"],

["key2", "value2"]

]);

**Map 相关的函数与属性：**

* **map.clear()** – 移除 Map 对象的所有键/值对 。
* **map.set()** – 设置键值对，返回该 Map 对象。
* **map.get()** – 返回键对应的值，如果不存在，则返回 undefined。
* **map.has()** – 返回一个布尔值，用于判断 Map 中是否包含键对应的值。
* **map.delete()** – 删除 Map 中的元素，删除成功返回 true，失败返回 false。
* **map.size** – 返回 Map 对象键/值对的数量。
* **map.keys()** - 返回一个 Iterator 对象， 包含了 Map 对象中每个元素的键 。
* **map.values()** – 返回一个新的Iterator对象，包含了Map对象中每个元素的值 。

实例 - test.ts 文件

let nameSiteMapping = new Map(); // 设置 Map 对象 nameSiteMapping.set("Google", 1); nameSiteMapping.set("Runoob", 2); nameSiteMapping.set("Taobao", 3); // 获取键对应的值 console.log(nameSiteMapping.get("Runoob")); // 2 // 判断 Map 中是否包含键对应的值 console.log(nameSiteMapping.has("Taobao")); // true console.log(nameSiteMapping.has("Zhihu")); // false // 返回 Map 对象键/值对的数量 console.log(nameSiteMapping.size); // 3 // 删除 Runoob console.log(nameSiteMapping.delete("Runoob")); // true console.log(nameSiteMapping); // 移除 Map 对象的所有键/值对 nameSiteMapping.clear(); // 清除 Map console.log(nameSiteMapping);

使用 **es6** 编译：

tsc --target es6 test.ts

编译以上代码得到如下 JavaScript 代码：

实例 - test.js 文件

let nameSiteMapping = new Map(); // 设置 Map 对象 nameSiteMapping.set("Google", 1); nameSiteMapping.set("Runoob", 2); nameSiteMapping.set("Taobao", 3); // 获取键对应的值 console.log(nameSiteMapping.get("Runoob")); //40 // 判断 Map 中是否包含键对应的值 console.log(nameSiteMapping.has("Taobao")); //true console.log(nameSiteMapping.has("Zhihu")); //false // 返回 Map 对象键/值对的数量 console.log(nameSiteMapping.size); //3 // 删除 Runoob console.log(nameSiteMapping.delete("Runoob")); // true console.log(nameSiteMapping); // 移除 Map 对象的所有键/值对 nameSiteMapping.clear(); //清除 Map console.log(nameSiteMapping);

执行以上 JavaScript 代码，输出结果为：

2truefalse3trueMap { 'Google' => 1, 'Taobao' => 3 }Map {}

### **迭代 Map**

Map 对象中的元素是按顺序插入的，我们可以迭代 Map 对象，每一次迭代返回 [key, value] 数组。

TypeScript使用 **for...of** 来实现迭代：

实例 -test.ts 文件

let nameSiteMapping = new Map(); nameSiteMapping.set("Google", 1); nameSiteMapping.set("Runoob", 2); nameSiteMapping.set("Taobao", 3); // 迭代 Map 中的 key for (let key of nameSiteMapping.keys()) { console.log(key); } // 迭代 Map 中的 value for (let value of nameSiteMapping.values()) { console.log(value); } // 迭代 Map 中的 key => value for (let entry of nameSiteMapping.entries()) { console.log(entry[0], entry[1]); } // 使用对象解析 for (let [key, value] of nameSiteMapping) { console.log(key, value); }

使用 **es6** 编译：

tsc --target es6 test.ts

编译以上代码得到如下 JavaScript 代码：

实例

let nameSiteMapping = new Map(); nameSiteMapping.set("Google", 1); nameSiteMapping.set("Runoob", 2); nameSiteMapping.set("Taobao", 3); // 迭代 Map 中的 key for (let key of nameSiteMapping.keys()) { console.log(key); } // 迭代 Map 中的 value for (let value of nameSiteMapping.values()) { console.log(value); } // 迭代 Map 中的 key => value for (let entry of nameSiteMapping.entries()) { console.log(entry[0], entry[1]); } // 使用对象解析 for (let [key, value] of nameSiteMapping) { console.log(key, value); }

执行以上 JavaScript 代码，输出结果为：

GoogleRunoobTaobao123Google 1Runoob 2Taobao 3Google 1Runoob 2

# TypeScript 元组

**我们知道数组中元素的数据类型都一般是相同的（any[] 类型的数组可以不同），如果存储的元素数据类型不同，则需要使用元组。**

**元组中允许存储不同类型的元素，元组可以作为参数传递给函数。**

创建元组的语法格式如下：

var tuple\_name = [value1,value2,value3,…value n]

**实例**

声明一个元组并初始化：

var mytuple = [10,"Runoob"];

或者我们可以先声明一个空元组，然后再初始化：

var mytuple = [];

mytuple[0] = 120

mytuple[1] = 234

### **访问元组**

元组中元素使用索引来访问，第一个元素的索引值为 0，第二个为 1，以此类推第 n 个为 n-1，语法格式如下:

tuple\_name[index]

**实例**

以下实例定义了元组，包含了数字和字符串两种类型的元素：

TypeScript

var mytuple = [10,"Runoob"]; // 创建元组 console.log(mytuple[0]) console.log(mytuple[1])

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var mytuple = [10, "Runoob"]; // 创建元组 console.log(mytuple[0]); console.log(mytuple[1]);

输出结果为：

10Runoob

### 元组运算

我们可以使用以下两个函数向元组添加新元素或者删除元素：

push() 向元组添加元素，添加在最后面。

pop() 从元组中移除元素（最后一个），并返回移除的元素。

TypeScript

var mytuple = [10,"Hello","World","typeScript"]; console.log("添加前元素个数："+mytuple.length) // 返回元组的大小 mytuple.push(12) // 添加到元组中 console.log("添加后元素个数："+mytuple.length) console.log("删除前元素个数："+mytuple.length) console.log(mytuple.pop()+" 元素从元组中删除") // 删除并返回删除的元素 console.log("删除后元素个数："+mytuple.length)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var mytuple = [10, "Hello", "World", "typeScript"]; console.log("添加前元素个数：" + mytuple.length); // 返回元组的大小 mytuple.push(12); // 添加到元组中 console.log("添加后元素个数：" + mytuple.length); console.log("删除前元素个数：" + mytuple.length); console.log(mytuple.pop() + " 元素从元组中删除"); // 删除并返回删除的元素 console.log("删除后元素个数：" + mytuple.length);

输出结果为：

添加前元素个数：4添加后元素个数：5删除前元素个数：512 元素从元组中删除删除后元素个数：4

### 更新元组

元组是可变的，这意味着我们可以对元组进行更新操作：

TypeScript

var mytuple = [10, "Runoob", "Taobao", "Google"]; // 创建一个元组

console.log("元组的第一个元素为：" + mytuple[0])

// 更新元组元素 mytuple[0] = 121

console.log("元组中的第一个元素更新为："+ mytuple[0])

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var mytuple = [10, "Runoob", "Taobao", "Google"]; // 创建一个元组 console.log("元组的第一个元素为：" + mytuple[0]); // 更新元组元素 mytuple[0] = 121; console.log("元组中的第一个元素更新为：" + mytuple[0]);

输出结果为：

元组的第一个元素为：10元组中的第一个元素更新为：121

### 解构元组

我们也可以把元组元素赋值给变量，如下所示：

TypeScript

var a =[10,"Runoob"] var [b,c] = a console.log( b ) console.log( c )

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var a = [10, "Runoob"]; var b = a[0], c = a[1]; console.log(b); console.log(c);

输出结果为：

10Runoob