# 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：

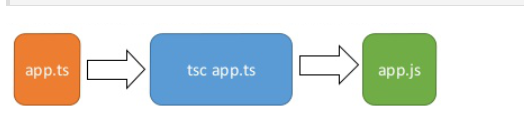
var message = "Hello World";

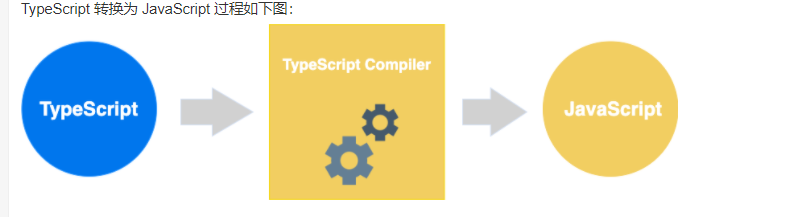
console.log(message);

使用 node 命令来执行 app.js 文件：

$ node app.js

Hello World

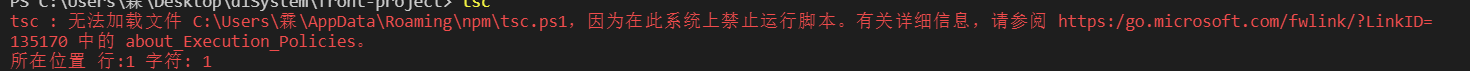




**同时编译多个 ts 文件**

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**



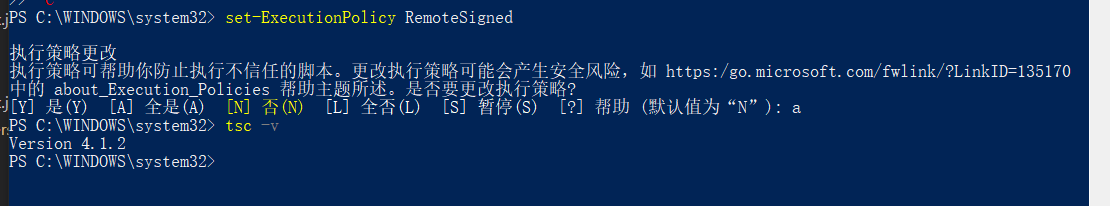
**解决:**

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