# TypeScript

<https://segmentfault.com/a/1190000010969537>

<https://www.jianshu.com/p/076ee2751cfc>

<https://www.cnblogs.com/xgqfrms/p/13499335.html>

<http://www.srcmini.com/3507.html>

<https://www.jianshu.com/p/c8aaba6e8ce0>

<https://blog.csdn.net/frontend_frank/article/details/111148526>

面试知识：

**void 和 undefined 有什么区别？**

**void 是 JS 中的一个运算符，语法是：**

void expression

它返回 undefined 的原始值，同时语句中的 expression 会被运算，也即产生了副作用。可以这样理解：它运算了表达式，但是不返回值（或者说返回 undefined）。

所以，通常使用 void 0 来得到 undefined。

**undefined 这个词在不同语境下的含义。**

1. undefined 是**术语（glossary）**。被用于表示一个概念时，它是一个术语，这个术语表示这样一个概念：未定义的值（即 undefined）。在 JS 中，只声明而未被赋值的变量、函数里未传实参的形参，都对应此概念。
2. undefined 是**类型（type）**。为了在语言层面实现上述的 undefined 概念，JS 为这个概念提供了一种原始类型（primitive type），即 undefined 类型。
3. undefined 是**值（value）**。上述的 undefined 类型非常特殊，不像其他诸如 string、number 类型可以定义出无限多的不同变量，undefined 类型的值只可能有一个，可以称之为 undefined 原始值（primitive value）。这个值在 JS 中没有字面量，准确的说是 JS 没有为程序员提供一个表示 undefined 原始值的字面量（可能有人会说代码里 undefined 不就是字面量吗？请看下一条）。
4. undefined 是**属性（undefined）**。代码中出现的 undefined 是全局变量的属性，所以说 JS 代码里出现的 undefined 并不是字面量。可以这样理解：JS 一开始就在内部将 undefined 原始值赋给了 undefined 属性

**为什么需要使用 void 0 而不直接用 undefined。**

先说为什么使用 void 0。因为 void 0 返回的值是 undefined 原始值，这与我们写代码的意图完全一致。

再说为什么不使用 undefined。 因为在两种情况下它有可能与我们的意图不一致。

**第一种情况：**既然 undefined 是一个属性，那它就有可能被重新赋值。但是这个担心是是多余的，因为从 ES5 开始 undefined 属性就是是一个只读属性了，不可能被重新赋值。

**第二种情况：局部变量**。因为 undefined 在 JS 中并不是保留字，所以在局部作用域中完全可以定义一个变量名为 undefined 的局部变量。看如下代码：

(function(){

let undefined = 'a string';

console.log(undefined)

})()

代码运行的结果是:

a string

### **TypeScript 类型Never 和 类型断言**

#### never 类型

never类型表示的是那些永不存在的值的类型。 例如， never类型是那些总是会抛出异常或**根本就不会有返回值的函数表达式或箭头函数表达式**的返回值类型； 变量也可能是 never类型，当它们被永不为真的类型保护所约束时。

never**类型是任何类型的子类型，也可以赋值给任何类型**；然而，没有类型是never的子类型或可以赋值给never类型（除了never本身之外）。 即使 any也不可以赋值给never。

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) {}}

#### 类型断言

类型断言好比其它语言里的类型转换  
类型断言有两种形式。 其一是“尖括号”语法：

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

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

另一个为as语法：

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

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

## **const与readonly的区别**

****readonly 关键字与 const 关键字不同。****  
1. const 字段只能在该字段的声明中初始化。  
   readonly 字段可以在声明或构造函数中初始化。因此，根据所使用的构造函数，readonly 字段可能具有不同的值。  
2. const 字段是编译时常数，而 readonly 字段可用于运行时常数。  
3. const 默认就是静态的，而 readonly 如果设置成静态的就必须显示声明。  
4. const 对于引用类型的常数，可能的值只能是 string 和 null 。

   readonly 可以是任何类型

## **[ts中类的方法和抽象类](https://www.cnblogs.com/windcat/p/11705057.html)**

// 类里面的修饰符 typescript提供三种修饰符

/\*

public: 公有 在类里面、类外面、子类都可以访问 （默认不加就是public)

protected: 保护 在类里面和子类可以访问

private: 私有 在当前类可以访问别的都不可以访问

\*/

// 类的静态属性 静态方法

/\* es5中的写法 \*/

function Person () {

　　// this.run = function () {} // 实例方法

　　// this.run = ()=> {}

}

Person.run = function () {} // 静态方法

class Person1 {

　　name:string;

　　static sex = 'man'; // 静态属性

　　constructor(name: string) {

　　　　this.name = name;

　　}

　　eat() {

　　　　console.log(`${this.name}吃饭`)

　　}

　　static work() {

　　console.log(`这是一个静态方法` + Person1.sex)

　　// console.log(`${this.name}哈哈`) // 错误的写法，静态方法里面无法调用类的属性、方法。

　　}

}

var p = new Person1('aaa');

p.eat(); // 实例方法调用

Person1.work(); // 静态方法调用

// 多态：父类定义一个方法不去实现，让继承它的子类去实现

// typescript中的抽象类是提供其它类的基类，不能直接被实例化；

// 用abstract关键字定义的抽象方法和抽象类，不包括具体实现必须在派生类实现。

// 并且抽象方法只能放在 抽象类里面

// 抽象类用于定义标准

abstract class Animal {

　　abstract eat():void;

}

// var a = new Animal(); // 错误的写法抽象类不能被实例化。

class Dog extends Animal {

　　constructor() {

　　　　super()

　　}

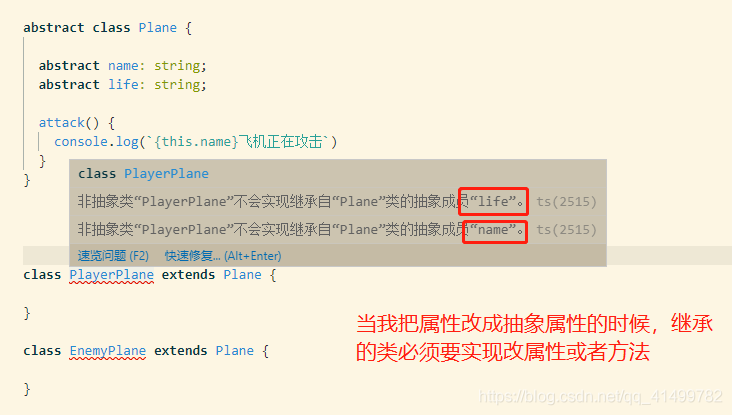
　　eat () {

　　　　console.log('抽象类的子类必须实现抽象类的抽象方法')

　　}

}

有时，某个类只表示一个抽象概念，主要用于提取子类共有的成员，而不能直接创建它的对象。该类可以作为抽象类。  
语法： abstract class xxx {}  
表示该类是一个抽象类，也可以抽象属性，抽象方法，但是不可以创建一个抽象类的对象

****改写上面的代码****：  


### **实现抽象属性**

#### **方式一：**直接重写改属性，并且在属性列表中赋上默认的值就好 IMG_257

#### **方式二：**通过构造函数来实现赋值： IMG_258

#### **方式三：**通过get函数来进行属性赋值: IMG_259

但是get函数的缺陷，如果只设置get,没有set函数,改属性是一个只读的哦

abstract class Plane {

abstract name: string;

abstract life: string;

attack() {

console.log(`{this.name}飞机正在攻击`)

*// 飞机相同的逻辑1*

*// 飞机相同的逻辑2*

this.otherFunction();

}

*// 飞机的不同逻辑*

abstract otherFunction()}

class PlayerPlane extends Plane {

name:string = '玩家1';

life:string = '100'

otherFunction(){

*// ...个性化*

}}

class EnemyPlane extends Plane {

name:string = '小罗罗1';

life:string = '10'

otherFunction(){

*// ...个性化*

模板模式：有些方法，所有的子类实现的流程完全一致，只是流程中的某个步骤的具体实现不一致，可以将该方法提取到父类，在父类中完成整个流程的实现(上面代码中的attack())，遇到实现不一致的方法时(把该方法做成抽象方法，字类必须去实现)，将该方法做成抽象方法。这种设计模式是大名鼎鼎的 模板模式

转载：https://blog.csdn.net/qq\_41499782/article/details/113043524

**问题代码考察**

### **下面代码会不会报错？怎么解决？**

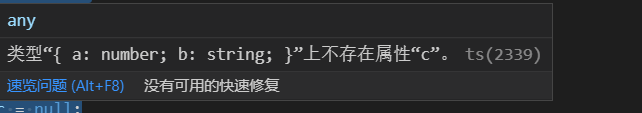
**const** obj = {

a: 1,

b: 'string',

};

obj.c = null;



解决方法：定义c就行

const obj = {

    a: 1,

    b: 'string',

    c: null

};

obj.c = null;

**ts中，定义了一个对象，去给这个对象赋值之前，别忘记先初始化！！！**

初始化代码如下：

private gun: Gun = {

oilgunNum: '0',

oilType: '',

};

### **下面代码中，foo 的类型应该如何声明**

**function** **foo** (a: number) {

**return** a + 1;

}

foo.bar = 123;

**解决**

function foo(a: number): number {

    return a + 1;

}

foo.bar = 123;

### **下面代码中，foo 的类型如何声明**

**let** foo = {};

**for** (**let** i = 0; i< 100; i++) {

foo[String(i)] = i;

}

解决

**接口中我们可以将数组的索引值和元素设置为不同类型，索引值可以是数字或字符串。**

TypeScript

interface namelist { [index:number]:string }

var list2:namelist = ["John",1,"Bran"] // 错误元素 1 不是 string 类型

interface ages { [index:string]:number }

var agelist:ages;

agelist["John"] = 15 // 正确 agelist[2] = "nine" // 错误

interface ages { [index: string]: number }

let foo: ages = {};

for (let i = 0; i < 100; i++) {

    foo[String(i)] = i;

}

console.log(foo)

下面代码中，foo 的类型如何声明

function fn(value: number): string {

    return String(value);

}

const foo = fn;

解决：

const foo：any = fn;

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

## **TS中的一些关键词总结**

### **TS中的一些符号**

#### **! 断言操作符**

**! 的作用是断言某个变量不会是 null/undefined，**告诉编辑器停止报错。

const obj = {

    name: '牧码的星星'

}

const a = obj!.name; *// 假设 obj是你从后端获取的获取*

确定 obj.name 一定是存在的且不是null/undefined，使用! 只是消除编辑器报错，不会对运行有任何影响。

属性或者参数中使用 !，表示强制解析（告诉 typescript 编译器，这里一定有值）; 变量后使用 !: 表示类型推荐排除 null/undefined。

#### **?. 链判断运算符**

const orderId = response.result.data.orderId;

上面这种写法，很容易出现这种问题 orderId is undefined，稍微有些经验的我们立马就能想到，肯定是代码中 response.result.data 为 null 或者 undefined ，这样，你肯定获取不到 orderId。所以经验丰富的我们在遇到获取层级比较多的对象的时候，一般都是像下面这样写。

*// 正确的写法*

const orderId = (response

  && response.result

  && response.result.data

  && response.result.data.orderId) || '';

我们可以使用 ?. 来简化上面的代码。

const orderId = response?.result?.data?.orderId || '';

上面代码使用了?.运算符，直接在链式调用的时候判断，左侧的对象是否为null或undefined。如果是的，就不再往下运算，而是返回undefined。

?. 支持如下语法

obj?.prop *// 对象属性*

obj?.[expr] *// 对象属性*

arr?.[index] *// 获取数据中 index 下标对应的值*

func?.(...args) // 函数或对象方法的调用

?.和 && 的区别

### **TS中的一些关键词**

#### **type 类型别名用来给一个类型起个新名字**

type SetUser = (name: string, age: number) => void;

类型别名常用于联合类型（联合类型表示取值可以为多种类型中的一种），比如常见的如下写法

type UploadType = 'drag' | 'select';

#### **interface**

interface 和 type 的用法其实差不多，interface 也是用来定义类型的

interface SetUser = {

    (name: string, age: number) => void;

}

#### **type和interface的区别**

都支持拓展，但是语法不同

interface Name {

  name: string;

}

interface User extends Name {

  age: number;

}

type Name = {

  name: string;

}

type User = Name & { age: number  };

#### **typeof 可以获取一个变量的声明类型**

在 JavaScript 中， typeof 可以判断一个变量的基础数据类型， 在 TS 中，它还可以获取一个变量的声明类型

const obj = { a: '1' };

type Foo = typeof obj;

*// type Foo = { a: string }*

#### **keyof 可以获取一个对象接口的所有 key 值**

type Obj = { a: string; b: string }

type Foo = keyof obj;

*// type Foo = 'a' | 'b';*

#### **in 可以遍历枚举类型**

type Keys = 'a' | 'b' | 'c';

type Obj = {

    [ T in Keys]: string;

}

*// in 遍历 Keys，并为每个值赋予 string 类型*

*// type Obj = {*

*//     a: string,*

*//     b: string,*

*//     c: string*

*// }*

### **TS中一些内置的类型**

官方文档：https://www.typescriptlang.org/docs/handbook/utility-types.html#partialt

用到了上面提到的一些关键词

#### **Partial<T> 将类型的属性变成可选**

功能是将类型的属性变成可选，注意这是浅Partial

type Partial<T> = {

    [P in keyof T]?: T[P]

};

举例说明

interface UserInfo {

    id: string;

    name: string;

}

*// error：Property 'id' is missing in type '{ name: string; }' but required in type 'UserInfo'*

const xiaoming: UserInfo = {

    name: 'xiaoming'

}

使用  Partial<T>

type NewUserInfo = Partial<UserInfo>;

const xiaoming: NewUserInfo = {

    name: 'xiaoming'

}

这个  NewUserInfo 就相当于

interface NewUserInfo {

    id?: string;

    name?: string;

}

但是 Partial<T> 有个局限性，就是只支持处理第一层的属性，如果我的接口定义是这样的

interface UserInfo {

    id: string;

    name: string;

    fruits: {

        appleNumber: number;

        orangeNumber: number;

    }

}

type NewUserInfo = Partial<UserInfo>;

*// Property 'appleNumber' is missing in type '{ orangeNumber: number; }' but required in type '{ appleNumber: number; orangeNumber: number; }'.*

const xiaoming: NewUserInfo = {

    name: 'xiaoming',

    fruits: {

        orangeNumber: 1,

    }

}

可以看到，第二层以后就不会处理了，如果要处理多层，就可以自己通过  Conditional Types[2]

DeepPartial

type DeepPartial<T> = {

*// 如果是 object，则递归类型*

    [U in keyof T]?: T[U] extends object

      ? DeepPartial<T[U]>

      : T[U]

};

type PartialedWindow = DeepPartial<Window>; *// 现在window 上所有属性都变成了可选啦*

#### **Required将类型的属性变成必选**

type Required<T> = {

    [P in keyof T]-?: T[P]

};

其中 -? 是代表移除 ? 这个 modifier 的标识。再拓展一下，除了可以应用于 ? 这个 modifiers ，还有应用在 readonly ，比如 Readonly<T> 这个类型

type Readonly<T> = {

    readonly [p in keyof T]: T[p];

}

#### **Pick 从某个类型中挑出一些属性出来**

type Pick<T, K extends keyof T> = {

    [P in K]: T[P];

};

interface UserInfo {

    id: string;

    name: string;

}

type NewUserInfo = Pick<UserInfo, 'name'>; *// {name: string;}*

可以看到 NewUserInfo 中就只有个 name 的属性了。

#### **Record**

可以获得根据 K 中所有可能值来设置 key 以及 value 的类型

type Record<K extends keyof any, T> = {

    [P in K]: T;

};

举个例子

type CurRecord = Record<'a' | 'b' | 'c', UserInfo>; *// { a: UserInfo; b: UserInfo; c: UserInfo; }*

#### **Mutable<T> 将类型的属性变成可修改**

功能是将类型的属性变成可修改，这里的 -指的是去除。-readonly 意思就是去除只读，也就是可修改啦。

type Mutable<T> = {

  -readonly [P in keyof T]: T[P];

};

#### **Readonly<T> 类型的属性变成只读**

type Readonly<T> = {

    readonly [P in keyof T]: T[P]

};

#### **ReturnType 用来得到一个函数的返回值类型**

type ReturnType<T extends (...args: any[]) => any> = T extends (

  ...args: any[]

) => infer R

  ? R

  : any;

infer在这里用于提取函数类型的返回值类型。ReturnType<T> 只是将 infer R 从参数位置移动到返回值位置，因此此时 R 即是表示待推断的返回值类型。

下面的示例用ReturnType获取到 Func 的返回值类型为 string，所以，foo 也就只能被赋值为字符串了。

type Func = (value: number) => string;

const foo: ReturnType<Func> = "1";

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：

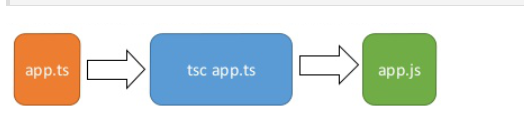
var message = "Hello World";

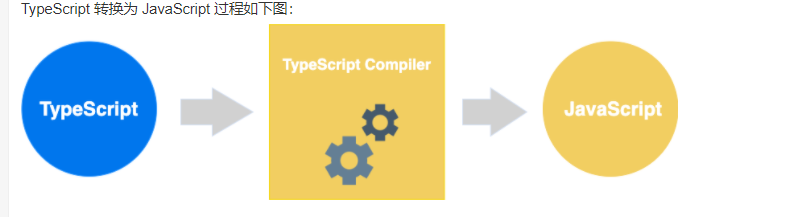
console.log(message);

使用 node 命令来执行 app.js 文件：

$ node app.js

Hello World

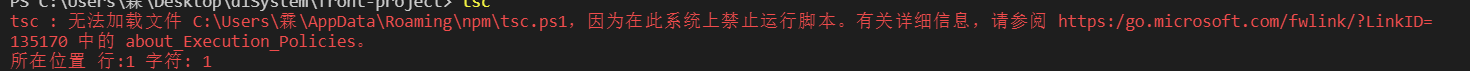




**同时编译多个 ts 文件**

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**



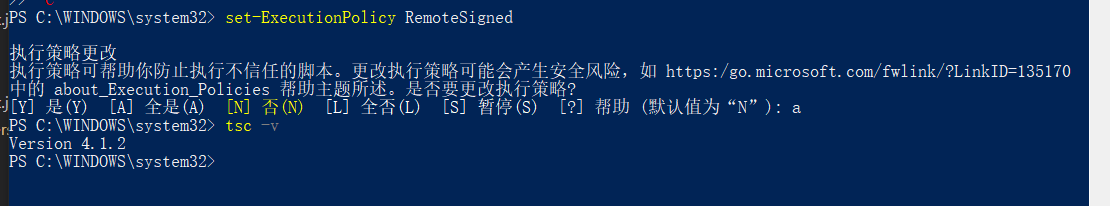
**解决:**

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]

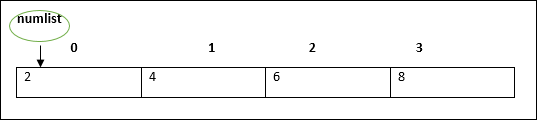
**如果数组声明时未设置类型，则会被认为是 any 类型，在初始化时根据第一个元素的类型来推断数组的类型。**

### **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

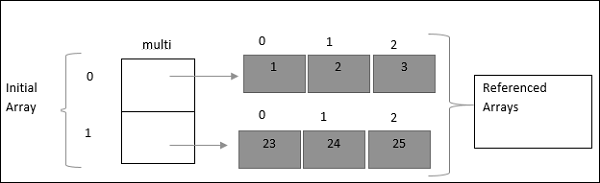
一个数组的元素可以是另外一个数组，这样就构成了多维数组（Multi-dimensional Array）。

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

### TypeScript 联合类型

**联合类型（Union Types）可以通过管道(|)将变量设置多种类型，赋值时可以根据设置的类型来赋值。**

****注意**：只能赋值指定的类型，如果赋值其它类型就会报错。**

创建联合类型的语法格式如下：

Type1|Type2|Type3

**实例**

声明一个联合类型：

TypeScript

var val:string|number val = 12 console.log("数字为 "+ val) val = "Runoob" console.log("字符串为 " + val)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var val; val = 12; console.log("数字为 " + val); val = "Runoob"; console.log("字符串为 " + val);

输出结果为：

数字为 12字符串为 Runoob

如果赋值其它类型就会报错：

var val:string|number

val = true

也可以将联合类型作为函数参数使用：

TypeScript

function disp(name:string|string[]) { if(typeof name == "string") { console.log(name) } else { var i; for(i = 0;i<name.length;i++) { console.log(name[i]) } } } disp("Runoob") console.log("输出数组....") disp(["Runoob","Google","Taobao","Facebook"])

编译以上代码，得到以下 JavaScript 代码：

JavaScript

function disp(name) { if (typeof name == "string") { console.log(name); } else { var i; for (i = 0; i < name.length; i++) { console.log(name[i]); } } } disp("Runoob"); console.log("输出数组...."); disp(["Runoob", "Google", "Taobao", "Facebook"]);

输出结果为：

Runoob输出数组....RunoobGoogleTaobaoFacebook

### 联合类型数组

我们也可以将数组声明为联合类型：

TypeScript

var arr:number[]|string[]; var i:number; arr = [1,2,4] console.log("\*\*数字数组\*\*") for(i = 0;i<arr.length;i++) { console.log(arr[i]) } arr = ["Runoob","Google","Taobao"] console.log("\*\*字符串数组\*\*") for(i = 0;i<arr.length;i++) { console.log(arr[i]) }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var arr; var i; arr = [1, 2, 4]; console.log("\*\*数字数组\*\*"); for (i = 0; i < arr.length; i++) { console.log(arr[i]); } arr = ["Runoob", "Google", "Taobao"]; console.log("\*\*字符串数组\*\*"); for (i = 0; i < arr.length; i++) { console.log(arr[i]); }

输出结果为：

\*\*数字数组\*\*124\*\*字符串数组\*\*RunoobGoogleTaobao

## TypeScript 接口

**接口是一系列抽象方法的声明，是一些方法特征的集合，**

**这些方法都应该是抽象的，需要由具体的类去实现，然后第三方就可以通过这组抽象方法调用，让具体的类执行具体的方法。**

TypeScript 接口定义如下：

interface interface\_name { }

**实例**

以下实例中，我们定义了一个接口 IPerson，接着定义了一个变量 customer，它的类型是 IPerson。

customer 实现了接口 IPerson 的属性和方法。

TypeScript

interface IPerson { firstName:string, lastName:string, sayHi: ()=>string }

var customer:IPerson = { firstName:"Tom", lastName:"Hanks", sayHi: ():string =>{return "Hi there"} } console.log("Customer 对象 ") console.log(customer.firstName) console.log(customer.lastName) console.log(customer.sayHi()) var employee:IPerson = { firstName:"Jim", lastName:"Blakes", sayHi: ():string =>{return "Hello!!!"} } console.log("Employee 对象 ") console.log(employee.firstName) console.log(employee.lastName)

需要注意接口不能转换为 JavaScript。 它只是 TypeScript 的一部分。

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var customer = { firstName: "Tom", lastName: "Hanks", sayHi: function () { return "Hi there"; } }; console.log("Customer 对象 "); console.log(customer.firstName); console.log(customer.lastName); console.log(customer.sayHi()); var employee = { firstName: "Jim", lastName: "Blakes", sayHi: function () { return "Hello!!!"; } }; console.log("Employee 对象 "); console.log(employee.firstName); console.log(employee.lastName);

输出结果为：

Customer 对象TomHanksHi thereEmployee 对象JimBlakes

### 联合类型和接口

以下实例演示了如何在接口中使用联合类型：

TypeScript

interface RunOptions { program:string; commandline:string[]|string|(()=>string); } // commandline 是字符串 var options:RunOptions = {program:"test1",commandline:"Hello"}; console.log(options.commandline) // commandline 是字符串数组 options = {program:"test1",commandline:["Hello","World"]}; console.log(options.commandline[0]); console.log(options.commandline[1]); // commandline 是一个函数表达式 options = {program:"test1",commandline:()=>{return "\*\*Hello World\*\*";}}; var fn:any = options.commandline; console.log(fn());

编译以上代码，得到以下 JavaScript 代码：

JavaScript

// commandline 是字符串 var options = { program: "test1", commandline: "Hello" }; console.log(options.commandline); // commandline 是字符串数组 options = { program: "test1", commandline: ["Hello", "World"] }; console.log(options.commandline[0]); console.log(options.commandline[1]); // commandline 是一个函数表达式 options = { program: "test1", commandline: function () { return "\*\*Hello World\*\*"; } }; var fn = options.commandline; console.log(fn());

输出结果为：

HelloHelloWorld\*\*Hello World\*\*

### 接口和数组

**接口中我们可以将数组的索引值和元素设置为不同类型，索引值可以是数字或字符串。**

TypeScript

interface namelist { [index:number]:string }

var list2:namelist = ["John",1,"Bran"] // 错误元素 1 不是 string 类型

interface ages { [index:string]:number }

var agelist:ages;

agelist["John"] = 15 // 正确 agelist[2] = "nine" // 错误

### 接口继承

**接口继承就是说接口可以通过其他接口来扩展自己。**

**Typescript 允许接口继承多个接口。**

**继承使用关键字 extends。**

单接口继承语法格式：

Child\_interface\_name extends super\_interface\_name

多接口继承语法格式：

Child\_interface\_name extends super\_interface1\_name, super\_interface2\_name,…,super\_interfaceN\_name

继承的各个接口使用逗号 **,** 分隔。

### **单继承实例**

TypeScript

interface Person { age:number }

interface Musician extends Person { instrument:string }

var drummer = <Musician>{};

drummer.age = 27 drummer.instrument = "Drums" console.log("年龄: "+drummer.age) console.log("喜欢的乐器: "+drummer.instrument)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var drummer = {}; drummer.age = 27; drummer.instrument = "Drums"; console.log("年龄: " + drummer.age); console.log("喜欢的乐器: " + drummer.instrument);

输出结果为：

年龄: 27喜欢的乐器: Drums

### **多继承实例**

TypeScript

interface IParent1 { v1:number } interface IParent2 { v2:number } interface Child extends IParent1, IParent2 { } var Iobj:Child = { v1:12, v2:23} console.log("value 1: "+Iobj.v1+" value 2: "+Iobj.v2)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var Iobj = { v1: 12, v2: 23 }; console.log("value 1: " + Iobj.v1 + " value 2: " + Iobj.v2);

输出结果为：

value 1: 12 value 2: 23

## TypeScript 类

**TypeScript 是面向对象的 JavaScript。**

**类描述了所创建的对象共同的属性和方法**。

TypeScript 支持面向对象的所有特性，比如 类、接口等。

TypeScript 类定义方式如下：

class class\_name {

// 类作用域}

**定义类的关键字为 class，后面紧跟类名，类可以包含以下几个模块（类的数据成员）：**

**字段** − 字段是类里面声明的变量。字段表示对象的有关数据。

**构造函数** − 类实例化时调用，可以为类的对象分配内存。

**方法** − 方法为对象要执行的操作。

**实例**

创建一个 Person 类：

TypeScript

class Person { }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var Person = /\*\* @class \*/ (function () { function Person() { } return Person; }());

### 创建类的数据成员

以下实例我们声明了类 Car，包含字段为 engine，构造函数在类实例化后初始化字段 engine。

this 关键字表示当前类实例化的对象。注意构造函数的参数名与字段名相同，this.engine 表示类的字段。

此外我们也在类中定义了一个方法 disp()。

TypeScript

class Car {

// 字段 engine:string;

// 构造函数 constructor(engine:string) { this.engine = engine }

// 方法 disp():void { console.log("发动机为 : "+this.engine) } }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var Car = /\*\* @class \*/ (function () { // 构造函数 function Car(engine) { this.engine = engine; } // 方法 Car.prototype.disp = function () { console.log("发动机为 : " + this.engine); }; return Car; }());

### 创建实例化对象

我们使用 new 关键字来实例化类的对象，语法格式如下：

var object\_name = new class\_name([ arguments ])

类实例化时会调用构造函数，例如：

var obj = new Car("Engine 1")

类中的字段属性和方法可以使用 **.** 号来访问：

// 访问属性

obj.field\_name

// 访问方法

obj.function\_name()

**完整实例**

以下实例创建来一个 Car 类，然后通过关键字 new 来创建一个对象并访问属性和方法：

TypeScript

class Car { // 字段 engine:string; // 构造函数 constructor(engine:string) { this.engine = engine } // 方法 disp():void { console.log("函数中显示发动机型号 : "+this.engine) } } // 创建一个对象 var obj = new Car("XXSY1") // 访问字段 console.log("读取发动机型号 : "+obj.engine) // 访问方法 obj.disp()

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var Car = /\*\* @class \*/ (function () { // 构造函数 function Car(engine) { this.engine = engine; } // 方法 Car.prototype.disp = function () { console.log("函数中显示发动机型号 : " + this.engine); }; return Car; }()); // 创建一个对象 var obj = new Car("XXSY1"); // 访问字段 console.log("读取发动机型号 : " + obj.engine); // 访问方法 obj.disp();

输出结果为：

读取发动机型号 : XXSY1函数中显示发动机型号 : XXSY1

### 类的继承

**TypeScript 支持继承类，即我们可以在创建类的时候继承一个已存在的类，这个已存在的类称为父类，继承它的类称为子类。**

**类继承使用关键字 extends，子类除了不能继承父类的私有成员(方法和属性)和构造函数，其他的都可以继承。**

**TypeScript 一次只能继承一个类，不支持继承多个类，但 TypeScript 支持多重继承（A 继承 B，B 继承 C）。**

语法格式如下：

class child\_class\_name extends parent\_class\_name

**实例**

类的继承：实例中创建了 Shape 类，Circle 类继承了 Shape 类，Circle 类可以直接使用 Area 属性：

TypeScript

class Shape { Area:number constructor(a:number) { this.Area = a } } class Circle extends Shape { disp():void { console.log("圆的面积: "+this.Area) } } var obj = new Circle(223); obj.disp()

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var \_\_extends = (this && this.\_\_extends) || (function () { var extendStatics = function (d, b) { extendStatics = Object.setPrototypeOf || ({ \_\_proto\_\_: [] } instanceof Array && function (d, b) { d.\_\_proto\_\_ = b; }) || function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; }; return extendStatics(d, b); }; return function (d, b) { extendStatics(d, b); function \_\_() { this.constructor = d; } d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_()); }; })(); var Shape = /\*\* @class \*/ (function () { function Shape(a) { this.Area = a; } return Shape; }()); var Circle = /\*\* @class \*/ (function (\_super) { \_\_extends(Circle, \_super); function Circle() { return \_super !== null && \_super.apply(this, arguments) || this; } Circle.prototype.disp = function () { console.log("圆的面积: " + this.Area); }; return Circle; }(Shape)); var obj = new Circle(223); obj.disp();

输出结果为：

圆的面积: 223

需要注意的是子类只能继承一个父类，TypeScript 不支持继承多个类，但支持多重继承，如下实例：

TypeScript

class Root { str:string; } class Child extends Root {} class Leaf extends Child {} // 多重继承，继承了 Child 和 Root 类 var obj = new Leaf(); obj.str ="hello" console.log(obj.str)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var \_\_extends = (this && this.\_\_extends) || (function () { var extendStatics = function (d, b) { extendStatics = Object.setPrototypeOf || ({ \_\_proto\_\_: [] } instanceof Array && function (d, b) { d.\_\_proto\_\_ = b; }) || function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; }; return extendStatics(d, b); }; return function (d, b) { extendStatics(d, b); function \_\_() { this.constructor = d; } d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_()); }; })(); var Root = /\*\* @class \*/ (function () { function Root() { } return Root; }()); var Child = /\*\* @class \*/ (function (\_super) { \_\_extends(Child, \_super); function Child() { return \_super !== null && \_super.apply(this, arguments) || this; } return Child; }(Root)); var Leaf = /\*\* @class \*/ (function (\_super) { \_\_extends(Leaf, \_super); function Leaf() { return \_super !== null && \_super.apply(this, arguments) || this; } return Leaf; }(Child)); // 多重继承，继承了 Child 和 Root 类 var obj = new Leaf(); obj.str = "hello"; console.log(obj.str);

输出结果为：

hello

### 继承类的方法重写

类继承后，子类可以对父类的方法重新定义，这个过程称之为方法的重写。

其中 super 关键字是对父类的直接引用，该关键字可以引用父类的属性和方法。

TypeScript

class PrinterClass { doPrint():void { console.log("父类的 doPrint() 方法。") } } class StringPrinter extends PrinterClass { doPrint():void { super.doPrint() // 调用父类的函数 console.log("子类的 doPrint()方法。") } }

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var obj = new StringPrinter() obj.doPrint() var \_\_extends = (this && this.\_\_extends) || (function () { var extendStatics = function (d, b) { extendStatics = Object.setPrototypeOf || ({ \_\_proto\_\_: [] } instanceof Array && function (d, b) { d.\_\_proto\_\_ = b; }) || function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; }; return extendStatics(d, b); }; return function (d, b) { extendStatics(d, b); function \_\_() { this.constructor = d; } d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_()); }; })(); var PrinterClass = /\*\* @class \*/ (function () { function PrinterClass() { } PrinterClass.prototype.doPrint = function () { console.log("父类的 doPrint() 方法。"); }; return PrinterClass; }()); var StringPrinter = /\*\* @class \*/ (function (\_super) { \_\_extends(StringPrinter, \_super); function StringPrinter() { return \_super !== null && \_super.apply(this, arguments) || this; } StringPrinter.prototype.doPrint = function () { \_super.prototype.doPrint.call(this); // 调用父类的函数 console.log("子类的 doPrint()方法。"); }; return StringPrinter; }(PrinterClass)); var obj = new StringPrinter(); obj.doPrint();

输出结果为：

父类的 doPrint() 方法。子类的 doPrint()方法。

### static 关键字

static 关键字用于定义类的数据成员（属性和方法）为静态的，静态成员可以直接通过类名调用。

TypeScript

class StaticMem { static num:number; static disp():void { console.log("num 值为 "+ StaticMem.num) } } StaticMem.num = 12 // 初始化静态变量 StaticMem.disp() // 调用静态方法

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var StaticMem = /\*\* @class \*/ (function () { function StaticMem() { } StaticMem.disp = function () { console.log("num 值为 " + StaticMem.num); }; return StaticMem; }()); StaticMem.num = 12; // 初始化静态变量 StaticMem.disp(); // 调用静态方法

输出结果为：

num 值为 12

### instanceof 运算符

instanceof 运算符用于判断对象是否是指定的类型，如果是返回 true，否则返回 false。

TypeScript

class Person{ } var obj = new Person() var isPerson = obj instanceof Person; console.log("obj 对象是 Person 类实例化来的吗？ " + isPerson);

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var Person = /\*\* @class \*/ (function () { function Person() { } return Person; }()); var obj = new Person(); var isPerson = obj instanceof Person; console.log(" obj 对象是 Person 类实例化来的吗？ " + isPerson);

输出结果为：

obj 对象是 Person 类实例化来的吗？ true

### 访问控制修饰符

TypeScript 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。TypeScript 支持 3 种不同的访问权限。

**public（默认）** : 公有，可以在任何地方被访问。

**protected** : 受保护，可以被其自身以及其子类和父类访问。

**private** : 私有，只能被其定义所在的类访问。

以下实例定义了两个变量 str1 和 str2，str1 为 public，str2 为 private，实例化后可以访问 str1，如果要访问 str2 则会编译错误。

TypeScript

class Encapsulate { str1:string = "hello" private str2:string = "world" } var obj = new Encapsulate() console.log(obj.str1) // 可访问 console.log(obj.str2) // 编译错误， str2 是私有的

### 类和接口

**类可以实现接口，使用关键字 implements，并将 interest 字段作为类的属性使用。**

以下实例红 AgriLoan 类实现了 ILoan 接口：

TypeScript

interface ILoan { interest:number }

class AgriLoan implements ILoan {

interest:number rebate:number

constructor(interest:number,rebate:number)

{ this.interest = interest this.rebate = rebate } }

var obj = new AgriLoan(10,1) console.log("利润为 : "+obj.interest+"，抽成为 : "+obj.rebate )

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var AgriLoan = /\*\* @class \*/ (function () { function AgriLoan(interest, rebate) { this.interest = interest; this.rebate = rebate; } return AgriLoan; }()); var obj = new AgriLoan(10, 1); console.log("利润为 : " + obj.interest + "，抽成为 : " + obj.rebate);

输出结果为：

利润为 : 10，抽成为 : 1

## TypeScript 对象

**对象是包含一组键值对的实例**。 值可以是标量、函数、数组、对象等，如下实例：

var object\_name = { key1: "value1", // 标量 key2: "value", key3: function() { // 函数 }, key4:["content1", "content2"] //集合 }

以上对象包含了标量，函数，集合(数组或元组)。

### **对象实例**

TypeScript

var sites = { site1:"Runoob", site2:"Google" }; // 访问对象的值 console.log(sites.site1) console.log(sites.site2)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites = { site1:"Runoob", site2:"Google" }; // 访问对象的值 console.log(sites.site1) console.log(sites.site2)

输出结果为：

RunoobGoogle

### TypeScript 类型模板

假如我们在 JavaScript 定义了一个对象：

var sites = { site1:"Runoob", site2:"Google" };

这时如果我们想在对象中添加方法，可以做以下修改：

sites.sayHello = function(){ return "hello";}

如果在 TypeScript 中使用以上方式则会出现编译错误，因为Typescript 中的对象必须是特定类型的实例。

TypeScript

var sites = { site1: "Runoob", site2: "Google", sayHello: function () { } // 类型模板 };

sites.sayHello = function () { console.log("hello " + sites.site1); }; sites.sayHello();

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites = { site1: "Runoob", site2: "Google", sayHello: function () { } // 类型模板 }; sites.sayHello = function () { console.log("hello " + sites.site1); }; sites.sayHello();

输出结果为：

hello Runoob

此外对象也可以作为一个参数传递给函数，如下实例：

TypeScript

var sites = { site1:"Runoob", site2:"Google", };

var invokesites = function(obj: { site1:string, site2 :string }) { console.log("site1 :"+obj.site1)

console.log("site2 :"+obj.site2) }

invokesites(sites)

编译以上代码，得到以下 JavaScript 代码：

JavaScript

var sites = { site1: "Runoob", site2: "Google" }; var invokesites = function (obj) { console.log("site1 :" + obj.site1); console.log("site2 :" + obj.site2); }; invokesites(sites);

输出结果为：

site1 :Runoob

site2 :Google

### 鸭子类型(Duck Typing)

鸭子类型（英语：duck typing）**是动态类型的一种风格，是多态(polymorphism)的一种形式。**

在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由"当前方法和属性的集合"决定。

*可以这样表述：*

*"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"*

在鸭子类型中，关注点在于对象的行为，能作什么；而不是关注对象所属的类型。例如，在不使用鸭子类型的语言中，我们可以编写一个函数，它接受一个类型为"鸭子"的对象，并调用它的"走"和"叫"方法。在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的"走"和"叫"方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。任何拥有这样的正确的"走"和"叫"方法的对象都可被函数接受的这种行为引出了以上表述，这种决定类型的方式因此得名。

interface IPoint { x:number y:number }

function addPoints(p1:IPoint,p2:IPoint):IPoint { var x = p1.x + p2.x var y = p1.y + p2.y return {x:x,y:y} }

// 正确

var newPoint = addPoints({x:3,y:4},{x:5,y:1})

// 错误

var newPoint2 = addPoints({x:1},{x:4,y:3})

## TypeScript 命名空间

**命名空间一个最明确的目的就是解决重名问题。**

假设这样一种情况，当一个班上有两个名叫小明的学生时，为了明确区分它们，我们在使用名字之外，不得不使用一些额外的信息，比如他们的姓（王小明，李小明），或者他们父母的名字等等。

命名空间定义了标识符的可见范围，一个标识符可在多个名字空间中定义，它在不同名字空间中的含义是互不相干的。这样，在一个新的名字空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他名字空间中。

TypeScript 中命名空间使用 **namespace** 来定义，语法格式如下：

namespace SomeNameSpaceName {

export interface ISomeInterfaceName { }

export class SomeClassName { }

}

**以上定义了一个命名空间 SomeNameSpaceName，如果我们需要在外部可以调用 SomeNameSpaceName 中的类和接口，**

**则需要在类和接口添加 export 关键字。**

要在另外一个命名空间调用语法格式为：

SomeNameSpaceName.SomeClassName;

如果一个命名空间在一个单独的 TypeScript 文件中，则应使用三斜杠 /// 引用它，语法格式如下：

/// <reference path = "SomeFileName.ts" />

以下实例演示了命名空间的使用，定义在不同文件中：

IShape.ts 文件代码：

namespace Drawing { export interface IShape { draw(); } }

Circle.ts 文件代码：

/// <reference path = "IShape.ts" /> namespace Drawing { export class Circle implements IShape { public draw() { console.log("Circle is drawn"); } } }

Triangle.ts 文件代码：

/// <reference path = "IShape.ts" /> namespace Drawing { export class Triangle implements IShape { public draw() { console.log("Triangle is drawn"); } } }

TestShape.ts 文件代码：

/// <reference path = "IShape.ts" /> /// <reference path = "Circle.ts" /> /// <reference path = "Triangle.ts" /> function drawAllShapes(shape:Drawing.IShape) { shape.draw(); } drawAllShapes(new Drawing.Circle()); drawAllShapes(new Drawing.Triangle());

使用 tsc 命令编译以上代码：

tsc --out app.js TestShape.ts

得到以下 JavaScript 代码：

JavaScript

/// <reference path = "IShape.ts" /> var Drawing; (function (Drawing) { var Circle = /\*\* @class \*/ (function () { function Circle() { } Circle.prototype.draw = function () { console.log("Circle is drawn"); }; return Circle; }()); Drawing.Circle = Circle; })(Drawing || (Drawing = {})); /// <reference path = "IShape.ts" /> var Drawing; (function (Drawing) { var Triangle = /\*\* @class \*/ (function () { function Triangle() { } Triangle.prototype.draw = function () { console.log("Triangle is drawn"); }; return Triangle; }()); Drawing.Triangle = Triangle; })(Drawing || (Drawing = {})); /// <reference path = "IShape.ts" /> /// <reference path = "Circle.ts" /> /// <reference path = "Triangle.ts" /> function drawAllShapes(shape) { shape.draw(); } drawAllShapes(new Drawing.Circle()); drawAllShapes(new Drawing.Triangle());

使用 node 命令查看输出结果为：

$ node app.jsCircle is drawnTriangle is drawn

嵌套命名空间

命名空间支持嵌套，即你可以将命名空间定义在另外一个命名空间里头。

namespace namespace\_name1 { export namespace namespace\_name2 { export class class\_name { } } }

成员的访问使用点号 **.** 来实现，如下实例：

Invoice.ts 文件代码：

namespace Runoob {

export namespace invoiceApp {

export class Invoice { public calculateDiscount(price: number) { return price \* .40; } }

} }

InvoiceTest.ts 文件代码：

/// <reference path = "Invoice.ts" />

var invoice = new Runoob.invoiceApp.Invoice();

console.log(invoice.calculateDiscount(500));

使用 tsc 命令编译以上代码：

tsc --out app.js InvoiceTest.ts

得到以下 JavaScript 代码：

JavaScript

var Runoob; (function (Runoob) { var invoiceApp; (function (invoiceApp) { var Invoice = /\*\* @class \*/ (function () { function Invoice() { } Invoice.prototype.calculateDiscount = function (price) { return price \* .40; }; return Invoice; }()); invoiceApp.Invoice = Invoice; })(invoiceApp = Runoob.invoiceApp || (Runoob.invoiceApp = {})); })(Runoob || (Runoob = {})); /// <reference path = "Invoice.ts" /> var invoice = new Runoob.invoiceApp.Invoice(); console.log(invoice.calculateDiscount(500));

使用 node 命令查看输出结果为：

$ node app.js

200

## TypeScript 模块

**TypeScript 模块的设计理念是可以更换的组织代码。**

**模块是在其自身的作用域里执行，并不是在全局作用域，这意味着定义在模块里面的变量、函数和类等在模块外部是不可见的，除非明确地使用 export 导出它们。类似地，我们必须通过 import 导入其他模块导出的变量、函数、类等。**

两个模块之间的关系是通过在文件级别上使用 import 和 export 建立的。

模块使用模块加载器去导入其它的模块。 在运行时，模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。 大家最熟知的JavaScript模块加载器是服务于 Node.js 的 CommonJS 和服务于 Web 应用的 Require.js。

此外还有有 SystemJs 和 Webpack。

模块导出使用关键字 **export** 关键字，语法格式如下：

// 文件名 : SomeInterface.ts

export interface SomeInterface { // 代码部分 }

要在另外一个文件使用该模块就需要使用 **import** 关键字来导入:

import someInterfaceRef = require("./SomeInterface");

**实例**

IShape.ts 文件代码：

/// <reference path = "IShape.ts" /> export interface IShape { draw(); }

Circle.ts 文件代码：

import shape = require("./IShape"); export class Circle implements shape.IShape { public draw() { console.log("Cirlce is drawn (external module)"); } }

Triangle.ts 文件代码：

import shape = require("./IShape");

export class Triangle implements shape.IShape { public draw() { console.log("Triangle is drawn (external module)"); } }

TestShape.ts 文件代码：

import shape = require("./IShape"); import circle = require("./Circle"); import triangle = require("./Triangle"); function drawAllShapes(shapeToDraw: shape.IShape) { shapeToDraw.draw(); } drawAllShapes(new circle.Circle()); drawAllShapes(new triangle.Triangle());

使用 tsc 命令编译以上代码（AMD）：

tsc --module amd TestShape.ts

得到以下 JavaScript 代码：

IShape.js 文件代码：

define(["require", "exports"], function (require, exports) { });

Circle.js 文件代码：

define(["require", "exports"], function (require, exports) { var Circle = (function () { function Circle() { } Circle.prototype.draw = function () { console.log("Cirlce is drawn (external module)"); }; return Circle; })(); exports.Circle = Circle; });

Triangle.js 文件代码：

define(["require", "exports"], function (require, exports) { var Triangle = (function () { function Triangle() { } Triangle.prototype.draw = function () { console.log("Triangle is drawn (external module)"); }; return Triangle; })(); exports.Triangle = Triangle; });

TestShape.js 文件代码：

define(["require", "exports", "./Circle", "./Triangle"], function (require, exports, circle, triangle) { function drawAllShapes(shapeToDraw) { shapeToDraw.draw(); } drawAllShapes(new circle.Circle()); drawAllShapes(new triangle.Triangle()); });

使用 tsc 命令编译以上代码（Commonjs）：

tsc --module commonjs TestShape.ts

得到以下 JavaScript 代码：

Circle.js 文件代码：

var Circle = (function () { function Circle() { } Circle.prototype.draw = function () { console.log("Cirlce is drawn"); }; return Circle; })(); exports.Circle = Circle;

Triangle.js 文件代码：

var Triangle = (function () { function Triangle() { } Triangle.prototype.draw = function () { console.log("Triangle is drawn (external module)"); }; return Triangle; })(); exports.Triangle = Triangle;

TestShape.js 文件代码：

var circle = require("./Circle"); var triangle = require("./Triangle"); function drawAllShapes(shapeToDraw) { shapeToDraw.draw(); } drawAllShapes(new circle.Circle()); drawAllShapes(new triangle.Triangle());

输出结果为：

Cirlce is drawn (external module)Triangle is drawn (external module)

## TypeScript 声明文件

TypeScript 作为 JavaScript 的超集，在开发过程中不可避免要引用其他第三方的 JavaScript 的库。虽然通过直接引用可以调用库的类和方法，但是却无法使用TypeScript 诸如类型检查等特性功能。为了解决这个问题，需要将这些库里的函数和方法体去掉后只保留导出类型声明，而产生了一个描述 JavaScript 库和模块信息的声明文件。通过引用这个声明文件，就可以借用 TypeScript 的各种特性来使用库文件了。

假如我们想使用第三方库，比如 jQuery，我们通常这样获取一个 id 是 foo 的元素：

$('#foo');// 或

jQuery('#foo');

但是在 TypeScript 中，我们并不知道 $ 或 jQuery 是什么东西：

jQuery('#foo');

// index.ts(1,1): error TS2304: Cannot find name 'jQuery'.

这时，我们需要使用 declare 关键字来定义它的类型，帮助 TypeScript 判断我们传入的参数类型对不对：

declare var jQuery: (selector: string) => any;

jQuery('#foo');

declare 定义的类型只会用于编译时的检查，编译结果中会被删除。

上例的编译结果是：

jQuery('#foo');

### **声明文件**

声明文件以 **.d.ts** 为后缀，例如：

runoob.d.ts

声明文件或模块的语法格式如下：

declare module Module\_Name {}

TypeScript 引入声明文件语法格式：

/// <reference path = " runoob.d.ts" />

当然，很多流行的第三方库的声明文件不需要我们定义了，比如 jQuery 已经有人帮我们定义好了：[jQuery in DefinitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/jquery/index.d.ts" \t "https://www.runoob.com/typescript/_blank)。

### **实例**

以下定义一个第三方库来演示：

CalcThirdPartyJsLib.js 文件代码：

var Runoob; (function(Runoob) { var Calc = (function () { function Calc() { } }) Calc.prototype.doSum = function (limit) { var sum = 0; for (var i = 0; i <= limit; i++) { sum = sum + i; } return sum; } Runoob.Calc = Calc; return Calc; })(Runoob || (Runoob = {})); var test = new Runoob.Calc();

如果我们想在 TypeScript 中引用上面的代码，则需要设置声明文件 Calc.d.ts，代码如下：

Calc.d.ts 文件代码：

declare module Runoob { export class Calc { doSum(limit:number) : number; } }

声明文件不包含实现，它只是类型声明，把声明文件加入到 TypeScript 中：

CalcTest.ts 文件代码：

/// <reference path = "Calc.d.ts" /> var obj = new Runoob.Calc(); // obj.doSum("Hello"); // 编译错误 console.log(obj.doSum(10));

下面这行导致编译错误，因为我们需要传入数字参数：

obj.doSum("Hello");

使用 tsc 命令来编译以上代码文件：

tsc CalcTest.ts

生成的 JavaScript 代码如下：

CalcTest.js 文件代码：

/// <reference path = "Calc.d.ts" /> var obj = new Runoob.Calc(); //obj.doSum("Hello"); // 编译错误 console.log(obj.doSum(10));

最后我们编写一个 runoob.html 文件，引入 CalcTest.js 文件及第三方库 CalcThirdPartyJsLib.js：

实例

<!DOCTYPE html> <html> <head> <meta charset="utf-8"> <title>菜鸟教程(runoob.com)</title> <script src = "CalcThirdPartyJsLib.js"></script> <script src = "CalcTest.js"></script> </head> <body> <h1>声明文件测试</h1> <p>菜鸟测试一下。</p> </body> </html>

浏览器打开该文件输出结果如下：

