TypeScript

TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准（ES6 教程）。

TypeScript 由微软开发的自由和开源的编程语言，在 JavaScript 的基础上增加了静态类型检查的超集。

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。

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 特性

1. 静态类型

TypeScript 的最大特性就是增加了静态类型系统。在 TypeScript 中，开发者可以显式地声明变量、参数、返回值的类型，这样可以在编译时捕获很多潜在的类型错误。常见类型包括 number、string、boolean、array、tuple、enum 等，此外也支持自定义类型。

let name: string = "Alice";

let age: number = 25;

2. 类型推断

TypeScript 可以自动推断变量类型，即使不显式声明类型，TypeScript 也会根据变量的赋值内容来推断类型，从而在大多数情况下减少类型注解的书写量。

let name = "Alice"; // 推断为 string

3. 接口 (Interfaces)

TypeScript 提供了接口，允许定义复杂的对象结构。接口可以定义属性和方法，还可以通过 implements 关键字实现接口，或者通过 extends 进行扩展，便于定义复杂的数据类型。

interface Person {

name: string;

age: number;

greet(): void;

}

class Student implements Person {

constructor(public name: string, public age: number) {}

greet() {

console.log(`Hello, my name is ${this.name}`);

}

}

4. 类型别名 (Type Aliases)

类型别名 (type) 可以为复杂的类型定义简短的别名，便于代码复用。

type StringOrNumber = string | number;

let value: StringOrNumber = 42;

5. 枚举 (Enums)

TypeScript 引入了 enum 类型，用于定义一组命名的常量，提高代码的可读性。枚举在 JavaScript 中没有直接的对应。

enum Direction {

Up,

Down,

Left,

Right,

}

let dir: Direction = Direction.Up;

6. 元组 (Tuples)

元组允许定义具有固定数量和类型的数组。它适用于需要固定数据结构的场景，比如坐标或 RGB 颜色值。

let point: [number, number] = [10, 20];

7. 访问控制修饰符 (Access Modifiers)

TypeScript 在类中提供了 public、private 和 protected 修饰符，允许控制属性或方法的可见性，支持更好的封装。

class Person {

private name: string;

protected age: number;

public constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

}

8. 抽象类 (Abstract Classes)

TypeScript 支持抽象类，抽象类不能直接实例化，需要由子类实现。抽象类适用于定义通用行为和抽象方法的类层次结构。

abstract class Animal {

abstract makeSound(): void;

}

class Dog extends Animal {

makeSound() {

console.log("Woof!");

}

}

9. 泛型 (Generics)

TypeScript 支持泛型，允许在类、接口和函数中使用参数化类型，使得代码可以适应不同的类型需求，同时保持类型安全。

function identity<T>(value: T): T {

return value;

}

let num = identity<number>(42);

10. 模块和命名空间

TypeScript 提供了基于 ES6 的模块系统，使用 import 和 export 导入和导出模块。此外，TypeScript 还支持命名空间（Namespace），用于组织代码和避免命名冲突。

// math.ts

export function add(a: number, b: number): number {

return a + b;

}

// main.ts

import { add } from "./math";

console.log(add(2, 3));

11. 类型守卫 (Type Guards)

TypeScript 提供了类型守卫，可以在代码中检查变量类型，帮助编译器推断更加具体的类型。这对于联合类型尤为重要。

function printId(id: string | number) {

if (typeof id === "string") {

console.log(id.toUpperCase());

} else {

console.log(id.toFixed(2));

}

}

12. 可选链和空值合并运算符

TypeScript 增加了 JavaScript 的可选链 (?.) 和空值合并运算符 (??)，简化了代码中对可能为 null 或 undefined 值的处理。

let user = { name: "Alice", address: { city: "Wonderland" } };

console.log(user?.address?.city); // 如果 address 存在则输出 city，否则返回 undefined

let value = null;

console.log(value ?? "default"); // 如果 value 为 null 或 undefined，则返回 "default"

13. 类型兼容性和工具类型

TypeScript 提供了一些工具类型，如 Partial、Pick、Readonly、Record 等，这些类型可以帮助生成新的类型，简化类型定义。

interface Todo {

title: string;

description: string;

}

let partialTodo: Partial<Todo> = { title: "Learn TypeScript" }; // 可选属

14. 编译期错误检查

TypeScript 提供的编译期错误检查可以捕获 JavaScript 中不易发现的错误，如拼写错误、类型不匹配等，帮助提升代码质量。

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。

分号是可选的

每行指令都是一段语句，你可以使用分号或不使用， 分号在 TypeScript 中是可选的，建议使用。

1. 声明部分（Declarations）

类型声明：TypeScript 是一种静态类型的语言，可以通过类型声明来定义变量、函数、类等的类型。类型声明可以帮助代码更具可维护性和可读性。

实例

let name: string = "Alice";

let age: number = 30;

接口声明：用于定义对象的结构，包括对象的属性和方法。

实例

interface Person {

name: string;

age: number;

}

2. 变量声明（Variable Declarations）

在 TypeScript 中，可以使用 let, const, 和 var 来声明变量。推荐使用 let 和 const，var 用法不再推荐。

实例

let age: number = 25;

const pi: number = 3.14;

3. 函数声明（Function Declarations）

函数声明：TypeScript 允许声明带有类型注解的函数，包括参数类型和返回值类型。

实例

function greet(name: string): string {

return "Hello, " + name;

}

箭头函数：TypeScript 同样支持 ES6 的箭头函数，使用简洁的语法来声明函数。

实例

const greet = (name: string): string => "Hello, " + name;

4. 类声明（Class Declarations）

TypeScript 提供对面向对象编程的支持，允许定义类和类的方法、属性。

实例

class Person {

name: string;

age: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

greet() {

return `Hello, my name is ${this.name}`;

}

}

5. 接口与类型别名（Interfaces & Type Aliases）

接口（Interface）：用于描述对象的形状，接口可以继承和扩展。

实例

interface Animal {

name: string;

sound: string;

makeSound(): void;

}

类型别名（Type Alias）：允许为对象类型、联合类型、交叉类型等定义别名。

实例

type ID = string | number;

6. 模块和导入导出（Modules & Imports/Exports）

TypeScript 支持模块化编程，可以使用 import 和 export 来组织代码。

导出：

实例

export class Person {

constructor(public name: string) {}

}

导入：

实例

import { Person } from './person';

7. 类型断言（Type Assertions）

在某些情况下，TypeScript 无法推断出一个变量的准确类型，开发者可以使用类型断言来强制指定类型。

实例

let value: any = "hello";

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

8. 泛型（Generics）

泛型允许在定义函数、接口或类时不指定具体类型，而是使用占位符，让用户在使用时传入具体类型。泛型能够增加代码的复用性和类型安全性。

实例

function identity<T>(arg: T): T {

return arg;

}

9. 注释（Comments）

注释在 TypeScript 程序中用于解释代码的作用、思路等，增加代码的可读性。

单行注释：// 多行注释：/\* \*/

10. 类型推断（Type Inference）

TypeScript 在某些情况下会自动推断变量的类型。例如，在声明变量并赋值时，TypeScript 会推断出该变量的类型。

实例

let num = 10; // TypeScript 推断 num 为 number 类型

11. 类型守卫（Type Guards）

TypeScript 提供了类型守卫（如 typeof 和 instanceof），用于在运行时缩小变量的类型范围。

实例

function isString(value: any): value is string {

return typeof value === 'string';

}

12. 异步编程（Asynchronous Programming）

TypeScript 完全支持异步编程，可以使用 async/await 语法来处理异步操作。

实例

async function fetchData(): Promise<string> {

const response = await fetch("https://example.com");

const data = await response.text();

return data;

}

13. 错误处理（Error Handling）

TypeScript 允许使用 try/catch 块进行错误处理，还可以使用类型来描述错误的类型。

实例

try {

throw new Error("Something went wrong");

} catch (error) {

if (error instanceof Error) {

console.error(error.message);

}

}

TypeScript 基础类型

| **类型** | **描述** | **示例** |
| --- | --- | --- |
| string | 表示文本数据 | let name: string = "Alice"; |
| number | 表示数字，包括整数和浮点数 | let age: number = 30; |
| boolean | 表示布尔值 true 或 false | let isDone: boolean = true; |
| array | 表示相同类型的元素数组 | let list: number[] = [1, 2, 3]; |
| tuple | 表示已知类型和长度的数组 | let person: [string, number] = ["Alice", 30]; |
| enum | 定义一组命名常量 | enum Color { Red, Green, Blue }; |
| any | 任意类型，不进行类型检查 | let value: any = 42; |
| void | 无返回值（常用于函数） | function log(): void {} |
| null | 表示空值 | let empty: null = null; |
| undefined | 表示未定义 | let undef: undefined = undefined; |
| never | 表示不会有返回值 | function error(): never { throw new Error("error"); } |
| object | 表示非原始类型 | let obj: object = { name: "Alice" }; |
| union | 联合类型，表示可以是多种类型之一 | `let id: string |
| unknown | 不确定类型，需类型检查后再使用 | let value: unknown = "Hello"; |

注意：TypeScript 和 JavaScript 没有整数类型。

运算符

类型运算符

typeof 运算符

typeof 是一元运算符，返回操作数的数据类型。

查看以下实例:

var num = 12

console.log(typeof num); //输出结果: number

使用 tsc 命令编译以上代码得到如下 JavaScript 代码：

var num = 12;

console.log(typeof num); //输出结果: number

以上实例输出结果如下：

Number

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，我们将 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) {

}

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)

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)

数组方法

every()

检测数值元素的每个元素是否都符合条件。（有一个不满足就为false）

function isBigEnough(element, index, array) {

return (element >= 10);

}

var passed = [12, 5, 8, 130, 44].every(isBigEnough);

console.log("Test Value : " + passed ); // false

some()

检测数组元素中是否有元素符合指定条件。（有一个满足就为true）

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

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

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。

TypeScript 中的元组（Tuple）是一种特殊类型的数组，它允许在数组中存储不同类型的元素，与普通数组不同，元组中的每个元素都有明确的类型和位置。元组可以在很多场景下用于表示固定长度、且各元素类型已知的数据结构。

创建元组的语法格式如下：

let tuple: [类型1, 类型2, 类型3, ...];

实例

声明一个元组并初始化：

let mytuple: [number, string];

mytuple = [42,"Runoob"];

在上面的例子中，mytuple 是一个元组，它包含一个 number 类型和一个 string 类型的元素。

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做什么，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。