# ·一、介绍

## 1.1、什么是typescript

在引入编程社区 20 多年后，JavaScript 现在已成为有史以来应用最广泛的跨平台语言之一。JavaScript 最初是一种用于向网页添加微不足道的交互性的小型脚本语言，现已发展成为各种规模的前端和后端应用程序的首选语言。

虽然用 JavaScript 编写的程序的大小、范围和复杂性呈指数级增长，但 JavaScript 语言表达不同代码单元之间关系的能力却没有。结合 JavaScript 相当奇特的运行时语义，语言和程序复 杂性之间的这种不匹配使得 JavaScript 开发成为一项难以大规模管理的任务。 程序员编写的最常见的错误类型可以描述为类型错误：在预期不同类型的值的地方使用了某种类型的 值。这可能是由于简单的拼写错误、无法理解库的 API 表面、对运行时行为的错误假设或其他错误。

TypeScript 的目标是成为 JavaScript 程序的静态类型检查器——换句话说，是一个在代码运行之前运行的工具（静态）并确保程序的类型正确（类型检查）。 TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这 个语言添加了可选的静态类型和基于类的面向对象编程。

TypeScript 是一种非常受欢迎的 JavaScript 语言扩展。它在现有的 JavaScript 语法之上加入了一层类型层，而这一层即使被删除，也丝毫不会影响运行时的原有表现。许多人认为 TypeScript "只是一个编译器"，但更好的理解其实是把 TypeScript 看作两个独立的系统：编译器（即处理语法的部分）和语言工具（即处理与编辑器集成的部分）。通过独立看待这两个系统，就可以得到能够解释我们之前所做决策的两个重要视角。

## 1.2、JS ,ES ,TS的关系

广义的 JS : ECMAScript(js的语法规范), DOM(文档对象模型), BOM(浏览器对象模型)

TypeScript 是 JavaScript 的超集，即包含JavaScript 的所有元素，能运行JavaScript 的代码，并扩展了 JavaScript 的语法。相比于JavaScript ，它还增加了静态类型、类、模块、接口和类型注解方面的功能，使得代码更加规范，协作更加友好，维护更好维护，更易于大项目的开发。

特点:

增加了静态类型的检查

编写时报错，js 是运行时报错

## 1.3、安装

安装 ts: npm i typescript -g

编译 ts: tsc ./src/index.ts --outFile ./dist/index.js

实时监听: tsc --init && tsc --watch

安装 ts-node 可直接执行ts文件

npm install ts-node -g

执行ts文件: ts-node index.ts

# 二、typescript入门

## 2.1、发现问题

JavaScript 中的每个值都有一组行为，您可以通过运行不同的操作来观察。这听起来很抽象，我们来举一个简单的例子，考虑我们可能对名为 message 的变量运行的一些操作：

// 在 'message' 上访问属性 'toLowerCase'，并调用它

message.toLowerCase();

// 调用 'message'

message();

如果分解它，第一行可运行的代码访问一个属性 toLowerCase ，然后调用它。第二个尝试

message 直接调用。

但是假设我们不知道 message 。这很常见——我们无法可靠地说出尝试运行任何这些代码会得到什么结果。每个操作的行为完全取决于我们最初给 message 的赋值。

·可以调用 message 吗？

·它有 toLowerCase 这个属性吗？

·如果能， toLowerCase 可以调用吗？

·如果这两个值都是可调用的，它们返回什么？

这些问题的答案通常是我们在编写 JavaScript 时牢记在心的东西，我们必须希望所有细节都正确。假设 message 按以下方式定义:

const message = "Hello World!";

正如您可能猜到的，如果我们尝试运行 message.toLowerCase() ，我们只会得到相同的小写字符串。那第二行代码呢？如果您熟悉 JavaScript，您就会知道这会失败并出现异常：

TypeError: message is not a function

如果我们在开发时能避免这样的错误，那就太好了。

当我们运行我们的代码时，我们的 JavaScript 运行时选择做什么的方式是通过确定值的类型——它具有什么样的行为和功能。这 TypeError 就是暗指的一部分- 它说字符串 "Hello World!" 不能作为函数调用。

对于某些值，例如基本类型 string 和 number ，我们可以在运行时使用 typeof 运算符识别它们的类型。但是对于函数之类的其他东西，没有相应的运行时机制来识别它们的类型。例如，考虑这个函数：

function fn(x) {

return x.flip();

}

我们可以通过阅读代码观察到这个函数只有在给定一个具有可调用 flip 属性的对象时才能工作，但是JavaScript 并没有以我们可以在代码运行时检查的方式来显示这些信息。

在纯 JavaScript 中，告诉 fn 特定值做什么的唯一方法是调用它并查看会发生什么。这种行为使得在运行之前很难预测代码会做什么，这意味着在编写代码时更难知道代码会做什么。

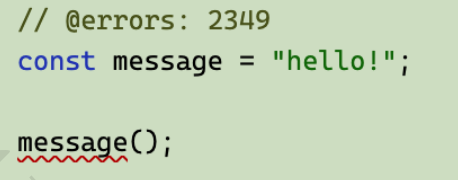
这样看来，类型是描述可以传递给 fn 哪些值会崩溃的概念。JavaScript 只真正提供动态类型——运行代码看看会发生什么。

另一种方法是使用静态类型系统在运行之前预测预期的代码。

## 2.2、静态类型检查

回想一下 TypeError 我们之前尝试将 string 作为函数调用的情况。 大多数人不喜欢在运行他们的代码时出现任何类型的错误 - 这些被认为是错误！当我们编写新代码时，我们会尽量避免引入新的错误。

理想情况下，我们可以有一个工具来帮助我们在代码运行之前发现这些错误。这就是像 TypeScript 这样的静态类型检查器所做的。 静态类型系统描述了当我们运行程序时我们的值的形状和行为。像TypeScript 这样的类型检查器，告诉我们什么时候事情可能会出轨。

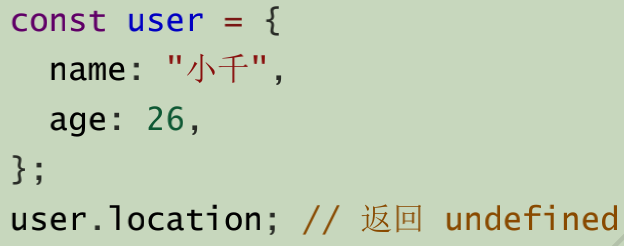


在我们运行代码之前，使用 TypeScript 运行最后一个示例会给我们一条错误消息。

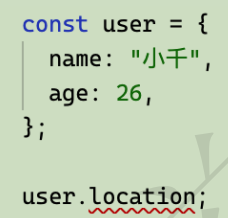
## 2.3、非异常故障

到目前为止，我们一直在讨论运行时错误——JavaScript 运行时告诉我们它认为某些东西是无意义的情况。出现这些情况是因为ECMAScript 规范明确说明了语言在遇到意外情况时应该如何表现。

例如，规范说尝试调用不可调用的东西应该抛出错误。也许这听起来像是“明显的行为”，但您可以想象访问对象上不存在的属性也应该抛出错误。相反，JavaScript 给了我们不同的行为并返回值undefined ：



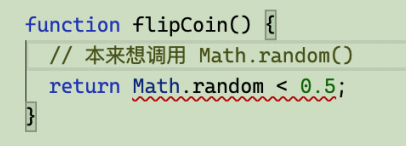
最终，静态类型系统要求必须调用哪些代码，应该在其系统中标记，即使它是不会立即抛出错误的“有效”JavaScript。比如：在 TypeScript 中，以下代码会产生关于 location 未定义的错误：TypeScript 可以在我们的程序中捕获很多合法的错误。例如：



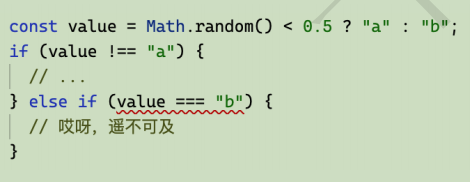
·错别字



·未调用的函数



·或基本逻辑错误



# 三、常用类型

// 定义字符串

let str1: string = "hello";

let str2: string;

str2 = "hi";

// 自动类型推断

let str3 = "abc";

// str3 = 5;

// 定义数字

let num1: number = 5;

num1 = 7;

// 定义布尔值

let done: boolean = true;

done = false;

// 定义undefined

let udf: undefined = undefined;

// 定义null

let nl: null = null;

// 联合类型

let a: number | boolean | string = 3;

a = true;

a = "hi";

// const list: string | null = localStorage.getItem("list");

// list && JSON.parse(list);

// any表示任何类型

let arr1: any = [1, 2, 3, 4];

arr1 = 3;

arr1 = true;

// 数组类型

let arr2: number[] = [1, 2, 3, 4, 5];

let arr3: Array<string> = ["1", "2", "3"];

let arr4: (number | boolean)[] = [1, 2, true];

let arr5: Array<number | boolean> = [1, 2, true];

let arr6: Array<number | boolean | any> = [1, 2, true, { a: 3 }];

let arr7: Array<any> = [1, 2, true, { a: 3 }];

let arr8: any = [1, 2, true, { a: 3 }];

// 元组类型

// 元组还是一种特殊的数组，它规定了数组的长度，和每一项的类型

let arr9: [number, string] = [1, "2"];

// console.log(arr9[1]); // 元组不能进行越界访问

// 对象类型

interface ObjType {

name: string;

age: number | string;

// 可选属性

sex?: number;

// 索引签名

[propname: string]: any;

}

let obj: ObjType = {

name: "zhangsan",

age: "20",

a: 3,

b() {},

c: true,

};

obj.sex = 1;

interface ItemType {

name: string;

id: number;

children?: ItemType[];

}

const arr10: ItemType[] = [

{

name: "zhangsan",

id: 1,

children: [

{

name: "lisi",

id: 11,

},

{

name: "wangwu",

id: 12,

},

],

},

{

name: "lisi",

id: 2,

},

];

// 枚举类型 enum

// 类似于对象，可以通过key找value，也可以通过value找key

enum Color {

Red = 3,

Green = 5,

Blue = 7,

}

// let g: Color = Color.Green;

// console.log(g);

let colorname: string = Color[5];

// console.log(colorname);

// 函数的参数要加类型

// 函数的类型是返回值的类型

// 可选参数要放在最后

function add(x: number, y?: string): string {

return y ? x + y : x + "";

}

add(3, "4");

add(3);

const add2: (x: number, y: number) => number = function (

x: number,

y: number

): number {

return x + y;

};

// void类型，用于函数，没有返回值的时候用

const add3 = (x: number, y: number): void => {

console.log(x, y);

};

// never类型，表示从不，用于函数,报错的时候，死循环的时候

// const add4 = (): never => {

// throw new Error()

// }

// const add5 = (): never => {

// while (true) {}

// }

// 泛型类型

function add6<T>(x: T, y: T) {

return `${x}${y}`;

}

add6<string>("3", "4");

// const count = ref<number>(3);

// class类型

interface ObjType2 {

name: string;

}

class Animal {

constructor(obj: ObjType2) {

this.name = obj.name;

}

name: string = "cat";

sayName(): string {

return this.name;

}

}

// 类的继承

class Cat extends Animal {

// constructor是类里面的一个特殊的函数

// 它在这个构造函数被实例化的时候调用

constructor(obj: ObjType2) {

// super函数调用，触发了父类的constructor

super(obj);

// console.log(obj);

}

}

// 类的实例化

const cat = new Cat({

name: "dog",

});

// console.log(cat.name);

// console.log(cat.sayName());

console.log(cat.name); // dog