Typescript 基础

介绍与安装

TypeScript 介绍

TypeScript 是由微软开发的 JavaScript 的一个超集，支持 ECMAScript 6 标准。

TypeScript 设计目标是开发大型应用，大幅度提高了 js 的可读性和可维护性。

相比于 javascript 的优势

1. 更早(写代码的同时)发现错误， 减少找 Bug、改 Bug 时间 ，提升开发效率。
2. 程序中任何位置的代码都有代码提示 ，随时随地的安全感，增强了开发体验。
3. 强大的类型系统提升了代码的可维护性，使得重构代码更加容易。
4. 支持最新的 ECMAScript 语法 ，优先体验最新的语法，让你走在前端技术的最前沿。
5. TS 类型推断机制， 不需要在代码中的每个地方都显示标注类型 ，让你在享受优势的同时，尽量降低了成本。

特性

- 面向项目： TS - 面向解决大型项目中，架构以及代码、共建等复杂维护场景 JS - 用于面向简单页面场景，脚本化语言
- 自主检测： TS - 编译期间，主动发现并且纠正错误 JS - 运行时报错
- 类型检测： TS - 弱类型，支持动态、静态的类型检测 JS - 弱类型，无类型选项检测
- 运行流程 TS - 依赖编译 => .ts -> .js -> 浏览器

NPM 安装 TypeScript

如果你的本地环境已经安装了 npm 工具，可以使用以下命令来安装。

```
1 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：

```
1 tsc -v
```

代码编写及编译

创建一个 ts 文件，编写代码

```
var msg: string = "Hello";
console.log(msg);
```

通常我们使用 .ts 作为 TypeScript 代码文件的扩展名。

然后执行以下命令将 TypeScript 转换为 JavaScript 代码：

编译 ts 为 js

```
tsc test.ts
```

编译后会在当前目录下（与 test.ts 同一目录）就会生成一个 test.js 文件，代码如下：

```
var msg = "Hello";
console.log(msg);
```

使用 node 命令来执行 app.js 文件：

```
node test.js
# 输出 hello
```

TS 语法

一、TS基础类型与写法

- boolean ｜ string | number | array | null | undefined

```
    // es
    let isEnabled = true
```

```
    let class = 'zhaowa'
    let classNum = 2
    let u = undefined
    let n = null
    let classArr = ['basic', 'execute']

    // TS
    let isEnabled: boolean = true
    let class: string = 'zhaowa'
    let classNum: number = 2
    let u: undefined = undefined
    let n: null = null

    let classArr: string[] = ['basic', 'execute']
    let classArr: Array<string> = ['basic', 'execute']
```

- tuple - 元组

```
    let tupleType: [string, boolean]
    tupleType = ['zhaowa', true]
```

- enum - 枚举

```
    // 数字类枚举 - 可无赋值，默认从零开始，依次递增
    enum Score {
        BAD, // 0
        NG,  // 1
        GOOD,
        PERFECT
    }

    let score: Score = Score.BAD

    // 字符串类型枚举
    enum Score {
        BAD = 'BAD',
        NG = 'NG',
        GOOD = 'GOOD',
        PERFECT = 'PERFECT'
    }
```

```
18
19      // 反向映射
20      enum Score {
21          BAD, // 0
22          NG,  // 1
23          GOOD,
24          PERFECT
25      }
26      let scoName = Score[0] // BAD
27      let scoVal = Score['BAD'] // 0
28
29      // 异构
30      enum Enum {
31          A,       // 0
32          B,       // 1
33          C = 'C', // C
34          D = 'D', // D
35          E = 6,   // 6
36          F,       // 7
37      }
```

- any | unknown | void

```
1       // any - 绕过所有类型检查 => 类型检测和编译筛查全部失效
2       let anyValue: any = 'anyValue'
3       anyValue = false
4       anyValue = 123
5
6       let value1:number = anyValue
7
8       // unknown - 绕过赋值检查 => 禁止更改传递
9       let unknownValue: unknown
10      unknownValue = true
11      unknownValue = 123
12      unknownValue = 'unknownValue'
13
14      let value1: unknown = unknownValue // OK
15      let value1: boolean = unknownValue // NOK
16      let value3: any = unknownValue // OK
17
18      // void - 声明函数返回值为空
19      function voidFunction(): void {
20          console.log('no return')
21      }
```

```
22
23      // never - 永不返回
24      function error(msg: string): never {
25          throw new Error(msg)
26      }
27
28      function longlongloop(): never {
29          while(true) {
30              // ...
31          }
32      }
```

- Object | object | {} - 对象

```
1       // ts
2       interface Object {
3           constructor: Function;
4           toString(): string;
5           hasOwnProperty(v: PropertyKey): boolean;
6           // ...
7       }
8       interface ObjectConstructor {
9           new(value?: any): Object;
10          // ……
11      }
12
13      // 自己的
14      function zhaowa(arg: Object): { toString(): string } {
15          return arg; // OK
16      }
17
18      // Object 包含原始类型
19      function course(courseName: Object) {
20          // ...
21      }
22      course('zhaowa') // OK
23
24      // object
25      function course(courseName: object) {
26          // ...
27      }
28      course('zhaowa') // ERROR
29
30      // {} - 描述了一个没有成员的对象
```

```
const obj = {}
obj.prop = 'zhaowa' // Error
```

二、接口 - interface

- 对行为的抽象，具体行为由类来实现

```
interface Class {
    name: string;
    time: number;
}

let zhaowa: Class = {
    name: 'ts',
    time: 2,
}

// 只读 & 任意
interface Class {
    readonly name: string;
    time: number;

    [propName: string]: any;
}
```

三、交叉类型 - &

```
interface A { name: D }
interface B { name: E }
interface C { name: F }

interface D { d: boolean }
interface E { e: string }
interface F { f: number }

type ABC = A & B & C

let abc: ABC = {
    name: {
```

```
            d: false,
            e: 'class',
            f: 5
        }
    }

    // 面试：合并冲突
    interface A {
        c: string;
        d: string;
    }
    interface B {
        c: number;
        e: string;
    }

    type AB = A & B;
    let ab: AB = {
        d: 'zhaowa',
        e: 'zhaowa'
    }
    // 合并 => c: never
```

四、断言 - 类型的中间执行时声明、转换（过程中和编译器的交流）

```
    // 尖括号形式
    let anyValue: any = 'zhaowa'
    let anyLength: number = (<string>anyValue).length; // 阶段性类型

    // as声明
    let anyValue: any = 'zhaowa'
    let anyLength: number = (anyValue as string).length;
```

五、类型守卫 - 保障在语法规定范围做进一步确认业务逻辑

```
    interface Teacher {
        name: string;
        courses: string[];
    }
```

```ts
interface Student {
    name: string;
    startTime: Date;
}

type Class = Teacher | Student;

// in - 定义属性场景下内容确认
function startCourse(cls: Class) {
    if ('courses' in cls) {
        // teacher 逻辑
    }
    if ('startTime' in cls) {
        // student 逻辑
    }
}

// typeof | instanceof - 类型分类场景下的身份确认
function class(name: string, score: string | number) {
    if (typeof score === 'number') {
        // teacher打的分数
    }
    if (typeof score === 'string') {
        // student打的分数
    }
}

interface Teacher {
    name: string;
    courses: string[];
}
interface Student {
    name: string;
    startTime: Date;
}

type Class = Teacher | Student;

const getName = (cls: Class) => {
    if (cls instanceof Teacher) {
        // teacher
    }
    if (cls instanceof Student) {
        // Student
    }
}
```

六、泛型

- 让模块可以支持多种类型的数据 - 让类型和值一样，可以被传递赋值

```
function startClass<T, U>(name: T, score: U): K {
    return name + score
}

function startClass<T, U>(name: T, score: U): String {
    return `${name}${score}`
}

function startClass<T, U>(name: T, score: U): T {
    return (name + String(score)) as any as T
}

console.log(startClass<string, number>('yy', 5))
```

七、装饰器

> 装饰器：就是一个方法，可以注入到类、方法、属性参数上来扩展类、属性、方法、参数的功能。

开启装饰器支持

1. 创建 tsconfig

```
tsc --init
```

2. 修改配置文件开启装饰器支持，添加 :

```
"experimentalDecorators": true
```

类装饰器（无法参数）

类装饰器在类声明之前被声明（紧靠着类声明）。类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。

```
function decoratorForPerson( target:any ){
    target.prototype.userName = '张三..';
    // 静态属性
    target.age = 18;
}

@decoratorForPerson
class Person{
    userName: any;
    static age: Number;
}

let pserson = new Person();
console.log( pserson.userName );

console.log( Person.age );
```

装饰器工厂（可传参）

```
function decoratorForPerson( options:any ){
    return function( target:any ){
        target.prototype.userName = options.userName;
        target.prototype.age = options.age;
    }
}

@decoratorForPerson({
    userName: '张三',
    age:19
})
class Person{
    userName: any;
    age: any;
}

let pserson = new Person();
console.log( pserson.userName );
console.log( pserson.age );
```

装饰器组合

```
function demo1( target:any ){
    console.log('demo1')
```

```
}
function demo2(  ){
    console.log('demo2')
    return ( target:any )=>{
        console.log('demo2里面')
    }
}
function demo3( ){
    console.log('demo3')
    return ( target:any )=>{
        console.log('demo3里面')
    }
}
function demo4( target:any ){
    console.log('demo4')
}

@demo1
@demo2()
@demo3()
@demo4
class Person{
}

/*结果是:
demo2
demo3
demo4
demo3里面
demo2里面
demo1
*/
```

属性装饰器

```
function fun3( arg:any ){
    return ( target:any , attr:any )=>{
        target[attr] = arg;
    }
}

class Obj3{
    @fun3('张三')
    //@ts-ignore
    userName:string
```

```
}
let obj3 = new Obj3();
console.log( obj3.userName );
```

方法装饰器

```
function test( target: any, propertyKey: string, descriptor:
PropertyDescriptor ) {
    console.log( target );
    console.log( propertyKey );
    console.log( descriptor );
}

class Person {
    @test
    sayName() {
        console.log( 'say name...' )
       return 'say name...';
    }
}

let p = new Person();
p.sayName()
```

TypeScript 中的内置类型

Partial

- 描述：将类型 T 的所有属性标记为可选属性
- 实现：

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

- 用法：`Partial<T>`

partial 这个类型，就是将某个类型里的属性添加上 `?` ，有了这个 modifier 之后，很多属性就可以是 undefined 了。

例如：

```ts
// 自定义的用户类型
interface Person {
 name: string;
 gender: 0 | 1; // 0: 女， 1: 男
 age: number;
}

const userList: Person[] = [];

// 当我们不需要用到所有属性进行过滤时，此时可以定义
const model: Partial<Person> = {
 name: '',
 gender: undefined,
};
```

Required

- 描述： Required 将类型 T 的所有属性标记为必选属性
- 实现：

```ts
type Required<T> = {
  [P in keyof T]-?: T[P];
};
```

- 用法： `Required<T>`

这个类型跟 Partial 相反，Partial 是将所有属性变成可选，Required 则是将所有属性改成必须。其中 -? 是代表移除 ?。

Readonly

- 描述：将所有属性标记为 readonly, 即不能修改
- 实现：

```ts
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};
```

- 用法: `Readonly<T>`

Pick

- 描述：从某个类型中过滤出某个或某些属性

- 实现：

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};
```

- 用法：`Pick<T, K>`

这个类型则可以将某个类型中的子属性挑出来，比如上面的那个 Person 属性，我们想把 name 和 age 挑出来

```
type otherPerson = Pick<Person, 'name' | 'age'>;
/*
{
  name: string
  age: number
}
*/
```

Record

- 描述：标记对象的属性（key）或者属性值（value）的类型
- 实现：

```
type Record<K extends keyof any, T> = {
  [P in K]: T;
};
```

- 用法：`Record<K, T>`

可以获得根据 K 中的所有可能值来设置 key 以及 value 的类型

```
// 设置key和value的类型
const userMap = Record<number, Person> = {
  1: {
    name: '张三',
    gender: 0,
    age: 12;
  }
};

```

```ts
// 设置value的类型
const info = Record<'name' | 'id', string> = {
  name: '张三',
  id: '1',
}
```

Exclude

- 描述：移除特定类型中的某个（些）属性
- 实现：

```ts
type Exclude<T, U> = T extends U ? never : T;
```

- 用法：`Exclude<T, U>`

比如

```ts
type Type1 = Exclude<'a' | 'b' | 'c' | 'd', 'a' | 'c' | 'f'>; // "b" | "d"
```

Exclude 和 Pick 一起用可以重写某些属性的类型，

```ts
type FilterPick<T, U extends keyof T> = Pick<T, Exclude<keyof T, U>>;

interface newPerson extends FilterPick<Person, 'name'> {
  name: number;
}
```

Extract<T, U>

- 描述：Exclude 的反操作，取 T，U 两者的交集属性
- 实现：

```ts
type Extract<T, U> = T extends U ? T : never;
```

- 用法：`Extract<T, U>`

```ts
type A = Extract<'a' | 'b' | 'c' | 'd', 'b' | 'c' | 'e'>; // 'b'|'c'
```

NonNullable

- 描述： 排除类型 T 的 null | undefined 属性
- 实现：

```ts
type NonNullable<T> = T extends null | undefined ? never : T;
```

- 用法： `NonNullable<T>`

ReturnType

- 描述： 获取函数类型 T 的返回类型
- 实现：

```ts
type ReturnType<T extends (...args: any) => any> = T extends (...args: any) =>
infer R ? R : any;
```

- 用法： `ReturnType<T>`

这个类型也非常好用，可以获取方法的返回类型，使用了 Conditional Types ，推论 (infer) 泛型 T 的返回类型 R 来拿到方法的返回类型。

元数据与reflect-metadata

元数据（Metadata）是一种描述性信息，用于描述和解释代码中的数据或结构。定义一个数组来存放数据，那么数组的length属性就是数组的元数据。定义一个类来表达特殊的数据结构，该类的类型就是类的元数据。通常我们可以通过设计时给对应的类、属性、方法、参数等设置元数据，用来标记注解或解释代码。元数据的定义、访问和修改通常使用 reflect-metadata 来实现。 TypeScript 中的装饰器经常用来定义元数据， TypeScript 在编译的时候会执行装饰器函数代码并将元数据附加到对应的目标上【类、属性、方法、函数参数等等】。

元数据通常用于以下场景上：

- 装饰器（decorator），在 TypeScript 中可以使用元数据来辅助修改和扩展类、方法、属性等行为。
- 依赖注入（DI）。元数据用于标记类的构造函数参数或属性，以便依赖注入容器在运行时自动解析和注入依赖项。

- ORM 对象关系映射（object relational mapping）。使用元数据来映射数据库表和类之间的关系，以及字段和属性之间的映射关系。
- 序列化和反序列化。在处理数据的存储和传输时，元数据可以用于制定数据对象的序列化和反序列化规则。

总的来说，元数据是一种描述性信息，可以提供关于代码结构、类型、注解、依赖关系等更多的信息，从而使代码可以更加灵活和可扩展。

在 TypeScript 中通常借助 Reflect-metadata 来解决提供元数据的处理能力。

```
// 通过给类、方法等数据定义元数据
// 元数据 —— 描绘数据的数据，会被附加到指定的类、方法之上，但是又不会影响类、方法本身

// 设置
// Reflect.defineMetadata(metadataKey, metadataValuearget, propertyKey)
// metadataKey: meta 数据的 key
// metadataValue: meta数据的值
// target: 对应的property key

console.log("######################")
console.log("元数据手动挂载开始")
// 自定义元数据
import 'reflect-metadata'

class Test {
    public static clsMethod() {}
    public instMethod() {}
}

let obj = new Test()

// 写
Reflect.defineMetadata('meta', 'class', Test)
Reflect.defineMetadata('meta', 'class method', Test, 'clsMethod')
Reflect.defineMetadata('meta', 'instance', obj)
Reflect.defineMetadata('meta', 'instance method', obj, 'instMethod')

// 读
console.log(Reflect.getMetadata('meta', Test))
console.log(Reflect.getMetadata('meta', Test, 'clsMethod'))
console.log(Reflect.getMetadata('meta', obj))
console.log(Reflect.getMetadata('meta', obj, 'instMethod'))

console.log("######################")
```

```
console.log('装饰器结合元数据')
function classDecorator(): ClassDecorator {
    return target => {
        console.log('classDecorator')
        Reflect.defineMetadata('meta', 'class', target)
    }
}

function staticMethodDecorator(): MethodDecorator {
    return (target, key, descriptor) => {
        console.log('staticMethodDecorator')
        Reflect.defineMetadata('meta', 'clsMethod', target, key)
    }
}

function methodDecorator(): MethodDecorator {
    return (target, key, descriptor) => {
        console.log('methodDecorator')
        Reflect.defineMetadata('meta', 'instMethod', target, key)
    }
}

function methodDecorator2(): MethodDecorator {
    return (target, key, descriptor) => {
        console.log('methodDecorator2')
    }
}

@classDecorator()
class Test2 {
    @staticMethodDecorator()
    public static clsMethod() {}
    @methodDecorator()
    @methodDecorator2()
    public instMethod() {}
}

let obj2 = new Test2()
Reflect.defineMetadata('meta', 'instance', obj2)
console.log(Reflect.getMetadata('meta', Test2))
console.log(Reflect.getMetadata('meta', Test2, 'clsMethod'))
console.log(Reflect.getMetadata('meta', obj2))
console.log(Reflect.getMetadata('meta', obj2, 'instMethod'))
console.log("######################")
console.log("直接绑定元数据")

@Reflect.metadata('meta', 'class')
```

```
class Test3 {
    @Reflect.metadata('meta', 'clsMethod')
    public static clsMethod() {}
    @Reflect.metadata('meta', 'method-meta')
    public instMethod() {}
}

console.log("######################")
```

实战

countdown

```
import EventEmitter from "events";

// 状态（枚举）
export enum CountdownEventName {
    START = 'start',
    STOP = 'stop',
    RUNNING = 'running'
}

enum CountdownStatus {
    running,
    paused,
    stoped
}

// 工具
export function fillZero(num: number) {
    return `0${num}`.slice(-2)
}

// 核心
export interface RemainTimeData {
    days: number;
    hours: number;
    mintes: number;
    seconds: number;
    count: number;
}

interface countdownEventMap {
```

```
    [CountdownEventName.RUNNING]: [RemainTimeData];
    [CountdownEventName.START]: [];
    [CountdownEventName.STOP]: [];
}

export class Countdown extends EventEmitter<countdownEventMap> {
    private static COUNT_IN_MILLSECOND: number = 1 * 100
    private static SECOND_IN_MILLSECOND: number = 10 * Countdown.COUNT_IN_MILLSECOND
    private static MINUTE_IN_MILLSECOND: number = 60 * Countdown.SECOND_IN_MILLSECOND
    private static HOUR_IN_MILLSECOND: number = 60 * Countdown.MINUTE_IN_MILLSECOND
    private static DAY_IN_MILLSECOND: number = 24 * Countdown.HOUR_IN_MILLSECOND

    private endTime: number
    private step: number
    private remainTime: number
    private status: CountdownStatus = CountdownStatus.stoped

    // 初始化
    constructor(endTime: number, step = 1e3) {
        super()

        this.endTime = endTime
        this.step = step
        this.remainTime = 0

        this.start()
    }

    public start() {
        this.emit(CountdownEventName.START)
        this.status = CountdownStatus.running

        this.countdown()
    }

    public stop() {
        this.emit(CountdownEventName.STOP)
        this.status = CountdownStatus.stoped
    }

    public countdown() {
        if (this.status !== CountdownStatus.running) {
            return
```

```
74          }
75
76          this.remainTime = Math.max(this.endTime - Date.now(), 0)
77          this.emit(CountdownEventName.RUNNING,
    this.parseRemainTime(this.remainTime))
78
79          if (this.remainTime > 0) {
80              setTimeout(() => this.countdown(), this.step)
81          } else {
82              this.stop()
83          }
84      }
85
86      private parseRemainTime(remainTime: number): RemainTimeData {
87          let time = remainTime
88
89          const days = Math.floor(time / Countdown.DAY_IN_MILLSECOND)
90          time = time % Countdown.DAY_IN_MILLSECOND
91
92          const hours = Math.floor(time / Countdown.HOUR_IN_MILLSECOND)
93          time = time % Countdown.HOUR_IN_MILLSECOND
94
95          const mintes = Math.floor(time / Countdown.MINUTE_IN_MILLSECOND)
96          time = time % Countdown.MINUTE_IN_MILLSECOND
97
98          const seconds = Math.floor(time / Countdown.SECOND_IN_MILLSECOND)
99          time = time % Countdown.SECOND_IN_MILLSECOND
100
101         const count = Math.floor(time / Countdown.COUNT_IN_MILLSECOND)
102
103         return {
104             days,
105             hours,
106             mintes,
107             seconds,
108             count
109         }
110     }
111 }
```

Nest.js