张轩 • 更新于 2023-07-24

上一节 本章简介 本章导学 下一节

安装 Typescript

Typescript 官网地址: https://www.typescriptlang.org/zh/

使用 nvm 来管理 node 版本: https://github.com/nvm-sh/nvm

安装 Typescript:

代码块

```
npm install -g typescript
```

使用 tsc 全局命令:

代码块

```
// 查看 tsc 版本
tsc -v
// 编译 ts 文件
tsc fileName.ts
```

原始数据类型

Javascript 类型分类：

原始数据类型 - primitive values：

- Boolean
- Null
- Undefined
- Number
- BigInt
- String
- Symbol

代码块

```
let isDone: boolean = false

// 接下来来到 number，注意 es6 还支持2进制和8进制，让我们来感受下

let age: number = 10
let binaryNumber: number = 0b1111

// 之后是字符串，注意es6新增的模版字符串也是没有问题的
let firstName: string = 'viking'
let message: string = `Hello, ${firstName}, age is ${age}`

// 还有就是两个奇葩兄弟两，undefined 和 null
let u: undefined = undefined
let n: null = null

// 注意 undefined 和 null 是所有类型的子类型。也就是说 undefined 类型的变量，可以赋值给 numbe
let num: number = undefined
```

代码块

```
let notSure: any = 4
notSure = 'maybe it is a string'
notSure = 'boolean'
// 在任意值上访问任何属性都是允许的：
notSure.myName
// 也允许调用任何方法：
notSure.getName()
```

Array 和 Tuple

Typescript 文档地址：Array 和 Tuple

代码块

```
//最简单的方法是使用「类型 + 方括号」来表示数组：
let arrOfNumbers: number[] = [1, 2, 3, 4]
//数组的项中不允许出现其他的类型：
//数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：
arrOfNumbers.push(3)
arrOfNumbers.push('abc')

// 元祖的表示和数组非常类似，只不过它将类型写在了里面 这就对每一项起到了限定的作用
let user: [string, number] = ['viking', 20]
//但是当我们写少一项 就会报错 同样写多一项也会有问题
user = ['molly', 20, true]
```

interface 接口

Typescript 文档地址：Interface

Duck Typing 概念：

> 如果某个东西长得像鸭子，像鸭子一样游泳，像鸭子一样嘎嘎叫，那它就可以被看成是一只鸭子。

```
// 我们定义了一个接口 Person
interface Person {
  name: string;
  age: number;
}
// 接着定义了一个变量 viking，它的类型是 Person。这样，我们就约束了 viking 的形状必须和接口 Pe
let viking: Person ={
  name: 'viking',
  age: 20
}

//有时我们希望不要完全匹配一个形状，那么可以用可选属性：
interface Person {
    name: string;
    age?: number;
}
let viking: Person = {
    name: 'Viking'
}

//接下来还有只读属性，有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 readonly

interface Person {
  readonly id: number;
  name: string;
  age?: number;
}
```

```
viking.id = 9527
```

函数

Typescript 文档地址：Functions

代码块

```
// 来到我们的第一个例子，约定输入，约定输出
function add(x: number, y: number): number {
  return x + y
}
// 可选参数
function add(x: number, y: number, z?: number): number {
  if (typeof z === 'number') {
    return x + y + z
  } else {
    return x + y
  }
}

// 函数本身的类型
const add2: (x: number, y: number, z?:number) => number = add

// interface 描述函数类型
const sum = (x: number, y: number) => {
  return x + y
}
interface ISum {
  (x: number, y: number): number
}
const sum2: ISum = sum
```

Typescript 文档地址：类型推论 - type inference

联合类型 - union types

代码块

```
// 我们只需要用中竖线来分割两个
let numberOrString: number | string
// 当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型
numberOrString.length
numberOrString.toString()
```

类型断言 - type assertions

代码块

```
// 这里我们可以用 as 关键字，告诉typescript 编译器，你没法判断我的代码，但是我本人很清楚，这里我
function getLength(input: string | number): number {
  const str = input as string
  if (str.length) {
    return str.length
  } else {
    const number = input as number
    return number.toString().length
  }
}
```

类型守卫 - type guard

```
// typescript 在不同的条件分支里面，智能的缩小了范围，这样我们代码出错的几率就大大的降低了。
function getLength2(input: string | number): number {
  if (typeof input === 'string') {
```

```
    } else {
      return input.toString().length
    }
  }
```

Class 类

面向对象编程的三大特点

- **封装（Encapsulation）**：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，
- **继承（Inheritance）**：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性。
- **多态（Polymorphism）**：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。

类 - Class

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name
  }
  run() {
    return `${this.name} is running`
  }
}
const snake = new Animal('lily')

// 继承的特性
class Dog extends Animal {
  bark() {
    return `${this.name} is barking`
```

```
const xiaobao = new Dog('xiaobao')
console.log(xiaobao.run())
console.log(xiaobao.bark())

// 这里我们重写构造函数，注意在子类的构造函数中，必须使用 super 调用父类的方法，要不就会报错。
class Cat extends Animal {
  constructor(name) {
    super(name)
    console.log(this.name)
  }
  run() {
    return 'Meow, ' + super.run()
  }
}
const maomao = new Cat('maomao')
console.log(maomao.run())
```

类成员的访问修饰符

- **public** 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 public 的
- **private** 修饰的属性或方法是私有的，不能在声明它的类的外部访问
- **protected** 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的

类与接口

类实现一个接口

```
interface Radio {
```

```
}
class Car implements Radio {
  switchRadio(trigger) {
    return 123
  }
}
class Cellphone implements Radio {
  switchRadio() {
  }
}

interface Battery {
  checkBatteryStatus(): void;
}

// 要实现多个接口，我们只需要中间用 逗号 隔开即可。
class Cellphone implements Radio, Battery {
  switchRadio() {
  }
  checkBatteryStatus() {

  }
}
```

枚举 Enums

枚举 Enums

```
// 数字枚举，一个数字枚举可以用 enum 这个关键词来定义，我们定义一系列的方向，然后这里面的值，枚举
enum Direction {
  Up,
  Down,
  Left,
```

```
console.log(Direction.Up)

// 还有一个神奇的点是这个枚举还做了反向映射
console.log(Direction[0])

// 字符串枚举
enum Direction {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT',
}
const value = 'UP'
if (value === Direction.Up) {
  console.log('go up!')
}
```

泛型 Generics

泛型 Generics

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

```
function echo(arg) {
  return arg
}
const result = echo(123)
// 这时候我们发现了一个问题，我们传入了数字，但是返回了 any

function echo<T>(arg: T): T {
  return arg
```

```
12
13  // 泛型也可以传入多个值
14  function swap<T, U>(tuple: [T, U]): [U, T] {
15    return [tuple[1], tuple[0]]
16  }
17
18  const result = swap(['string', 123])
```

泛型第二部分 - 泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法

代码块

```
1   function echoWithArr<T>(arg: T): T {
2     console.log(arg.length)
3     return arg
4   }
5
6   // 上例中，泛型 T 不一定包含属性 length，我们可以给他传入任意类型，当然有些不包括 length 属性，
7
8   interface IWithLength {
9     length: number;
10  }
11  function echoWithLength<T extends IWithLength>(arg: T): T {
12    console.log(arg.length)
13    return arg
14  }
15
16  echoWithLength('str')
17  const result3 = echoWithLength({length: 10})
18  const result4 = echoWithLength([1, 2, 3])
```

```ts
class Queue<T> {
  private data: T[] = [];
  push(item: T) {
    return this.data.push(item)
  }
  pop() {
    return this.data.shift()
  }
}
const queue = new Queue<number>()

queue.push(1)

const poped = queue.pop()

if (poped) {
  poped.toFixed()
}

//在上述代码中存在一个问题，它允许你向队列中添加任何类型的数据，当然，当数据被弹出队列时，也可以是

class Queue<T> {
  private data = [];
  push(item: T) {
    return this.data.push(item)
  }
  pop(): T {
    return this.data.shift()
  }
}
const queue = new Queue<number>()

//泛型和 interface
```

```
  value: U;
}

let kp1: KeyPair<number, string> = { key: 1, value: "str"}
let kp2: KeyPair<string, number> = { key: "str", value: 123}
```

类型别名 和 交叉类型

类型别名 Type Aliases

类型别名，就是给类型起一个别名，让它可以更方便的被重用。

代码块

```
let sum: (x: number, y: number) => number
const result = sum(1,2)
type PlusType = (x: number, y: number) => number
let sum2: PlusType

// 支持联合类型
type StrOrNumber = string | number
let result2: StrOrNumber = '123'
result2 = 123

// 字符串字面量
type Directions = 'Up' | 'Down' | 'Left' | 'Right'
let toWhere: Directions = 'Up'

```

交叉类型 Intersection Types

```
interface IName  {
  name: string
```

```
type IPerson = IName & { age: number }
let person: IPerson = { name: 'hello', age: 12}
```

声明文件

声明文件

@types 官方声明文件库
@types 搜索声明库

内置类型

内置类型

代码块

```
const a: Array<number> = [1,2,3]
// 大家可以看到这个类型，不同的文件中有多处定义，但是它们都是 内部定义的一部分，然后根据不同的版
const date: Date = new Date()
const reg = /abc/
// 我们还可以使用一些 build in object，内置对象，比如 Math 与其他全局对象不同的是，Math 不是一

Math.pow(2,2)

// DOM 和 BOM 标准对象
// document 对象，返回的是一个 HTMLElement
let body: HTMLElement = document.body
// document 上面的query 方法，返回的是一个 nodeList 类型
let allLis = document.querySelectorAll('li')

//当然添加事件也是很重要的一部分，document 上面有 addEventListener 方法，注意这个回调函数，因为
document.addEventListener('click', (e) => {
  e.preventDefault()
})
```

Utility Types

Typescript 还提供了一些功能性，帮助性的类型，这些类型，大家在 js 的世界是看不到的，这些类型叫做 utility types，提供一些简洁明快而且非常方便的功能。

代码块

```
1
2   // partial，它可以把传入的类型都变成可选
3   interface IPerson {
4     name: string
5     age: number
6   }
7
8   let viking: IPerson = { name: 'viking', age: 20 }
9   type IPartial = Partial<IPerson>
10  let viking2: IPartial = { }
11
12  // Omit，它返回的类型可以忽略传入类型的某个属性
13
14  type IOmit = Omit<IPerson, 'name'>
15  let viking3: IOmit = { age: 20 }
16
```

配置文件

配置文件的官方文档

配置示例

```
{
  "files": ["test.ts", "test2.d.ts"],
```

```
5         "module": "ESNext",
6         "target":"ES5",
7         "declaration": true
8       }
```

本章简介 上一节 下一节 本章导学

我要提出意见反馈