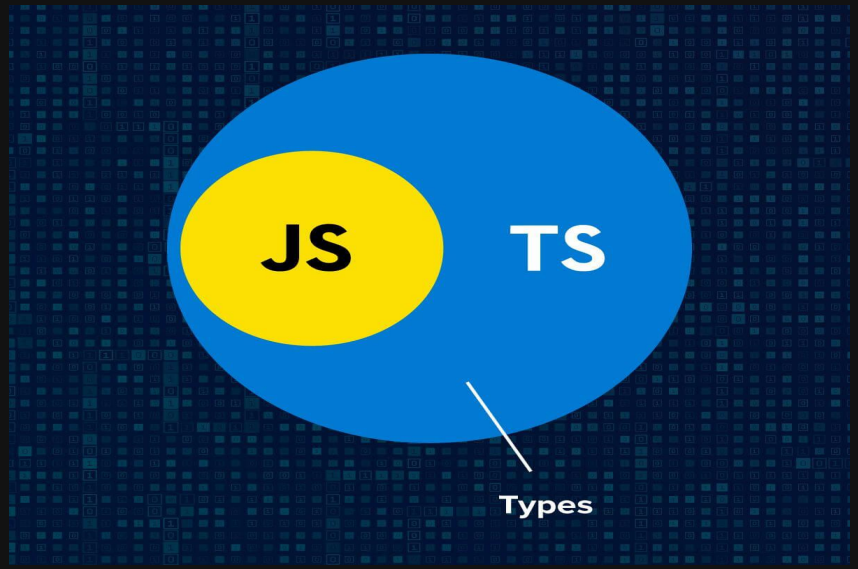
About Typescript

覃光明

- 前言
- 数据类型
- 函数(Function)
- 类型别名(type)与接口(interface)
- 联合类型(|)与交叉类型(&)
- 泛型(T)
- 常见类型操作符
- 断言
- 装饰器(@)

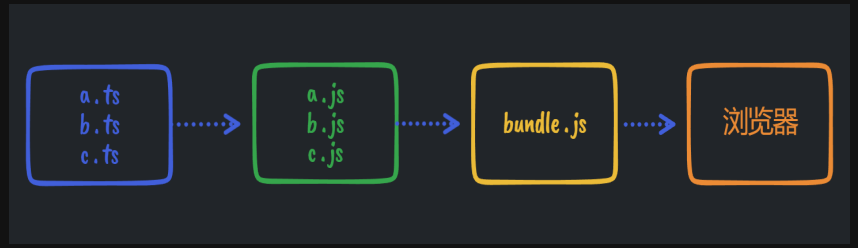
前言

Why You Should Choose TypeScript Over JavaScript



" TypeScript is JavaScript with syntax for types. "

- 是javascript的超集
- 类型语法
- 强类型语言



- ts文件最终还是会被编译成js文件去执行

数据类型

string类型

```
const name: string = '天气真好'
```

boolean类型

```
const isSuccess: boolean = true
```

number类型

```
const age: number = 18
```

symbol类型

```
const sy: symbol = Symbol('hhh')
```

Array类型

```
const list1: number[] = [1, 2, 3]

// 数组泛型写法
const list2: Array<number> = [1, 2, 3]
```

null类型

```
const isNull: null = null
```

undefined类型

```
const isUndefined: undefined = undefined
```

Object类型

```
type Person = {
  name: string,
  age: number
}

const o: Person= { name: '小明', age: 18 }
```

TS独有的数据类型 - any 与 unknown

any类型（top-type | bottom-type）

```
let xx: any = '小明'
xx = true
xx = 20
xx = '你好'
xx = []
xx = {}
```

unknown类型（top-type）

```
//
let un: unknown
un = true
un = 20
un = '你好'
un = []
un = {}
...
```

相同点

- 都属于是顶级类型
- 都能被任意类型赋值

不同点

- any可以是任意类型的父类型，同时也是任意类型的子类型；unknown只是任意类型的父类型，只是unknown和any的子类型

```
// ok
let anyV: any
let value1: unknown = anyV
let value2: any = anyV
let value3: boolean = anyV
let value4: undefined = anyV
...
```

```
// ok
let unV: unknown
let value1: unknown = unV
let value2: any = unV

// error
let value3: boolean = unV
let value4: undefined = unV
...
```

TS独有的数据类型 - any 与 unknown

不同点

- any: 我不在乎它是什么类型；unknown: 我不知道它是什么类型（意味着unknown类型需要重点关注）

```ts
// any 语法检查都会通过
let value: any

// ok
value.name
value.trim()
value.push()
...
```

```ts
// unknown 语法检查都会失败
let value: unknown

// error
value.name
value.trim()
value.push()
...
```

```ts
// any
function addItem (list: any, item: number) {
  list.push(item)
}

const arr: number[] = []

// 语法检查会通过，但是执行结果会报错
addItem('你好', 2)
```

```ts
// unknown 迫使开发者手动去做一些类型检查
function addItem (list: unknown, item: number) {
  // list.push(item) // error Object is of type 'unknown'
  if (list instanceof  Array) {
    list.push(item)
  }
}

const arr: number[] = []

// 语法检查时就会报错
addItem('你好', 2)
```

TS独有的数据类型 - Enum类型（枚举）

数字枚举

```
enum HttpCode {
  SUCCESS = 200
  FAILED = 500
}

const code: HttpCode = HttpCode.SUCCESS // 200
```

常量枚举

```
const enum NumberCollection {
  NORTH, // 0
  SOUTH, // 1
  EAST, // 2
  WEST, // 3
}
```

字符串枚举

```
enum Status {
  S = 'success',
  F = 'failed'
}

const s: Status = Status.s // success
```

异构枚举

```
enum EnumMix {
  A,       // 0
  B,       // 1
  C = "C", // C
  D = "D", // D
  E = 8,   // 8
  F,       // 9
}
```

TS独有的数据类型 - Tuple（元组）、Void、Never

元组类型：数组一般支持同种类型，元组可以支持不同类型且长度固定

```
let tupleType: [string, number]
tupleType = ['小明', 18]
```

元组：设置别名

```
let tupleType: [name: string, age: number];
tupleType = ['小明', 18]

// tupleType[0]的别名是name  tupleType[1]的别名是age
// 但是无法通过别名访问到具体的值 tupleType.name 为 undefined
```

Void：函数特有的类型

```
function doSomething(): void {
  //...
}
```

never类型：表示永远不存在的值的类型

```
function rejectError(message: string): never {
  throw new Error(message);
}

// 可以利用never类型尽可能的保证类型的绝对安全
type Info = string | number
// type Info = string | number | boolean
function getInfo(msg: Info): void {
  if (typeof msg === 'string') {
    // ...
  } else if (typeof msg === 'number') {
    // ...
  } else {
    let value: never = msg;
    // ...
  }
}
```

函数(Function)

类型：函数类型、入参类型、返回值类型

```
// 定义一个函数类型
type DoSumFn = (a: number, b: number) => number;

const doSum: DoSumFn = function(a: number, b: number): number {
  return a + b
}

doSum(1, 2) // 3
```

可选参数、默认参数

```
// 注意把可选参数放在后面

function printInfo(name: string, age: number = 18, hobby?:string ): string {
  return `姓名: ${name}, 年龄: ${age}, 爱好: ${hobby}`
}

printInfo('小明') // 姓名: 小明, 年龄: 18, 爱好: undefined
printInfo('小明', 20, '睡觉') // 姓名: 小明, 年龄: 20, 爱好: 睡觉
```

函数(Function)

函数重载

在javascript中没有重载的概念，重名函数会被覆盖

```
function a () {
  console.log('出太阳了')
}

function a () {
  console.log('下雨了')
}

a() // 下雨了
```

函数(Function)

函数重载

typescript中的重载指的是，声明多个同名函数，它们的参数类型不同

```
// 声明
type NumberOrString = string | number
function add (arg1: string, arg2: string): string
function add (arg1: number, arg2: number): number

// 实现
function add(a: NumberOrString, b: NumberOrString) {
  // 实现上要严格判断两个参数的类型是否相等
  if (typeof a === 'string' && typeof b === 'string') {
    return a + b
  } else if (typeof a === 'number' && typeof b === 'number') {
    return a + b
  }

  return ''
}
```

类型别名(type)与接口(interface)

type：用来给类型起一个新的名字，定义类型

```
type isString = string
const msg: isString = 'hello ts'
```

只读属性、可选属性、任意属性

```
type Person = {
  name: string,
  age: number,
  // 只读 readonly
  readonly sex: string,
  // 可选 ?
  email?: string,
  // 任意属性：索引签名
  [prop: string]: any
}
```

interface: 定义一个对象类型

```
interface Person {
  name: string;
  age: number;
}
const p:Person = { name: '小明', age: 18 }
```

只读属性、可选属性、任意属性

```
interface Person {
  name: string,
  age: number,
  // 只读 readonly
  readonly sex: string,
  // 可选 ?
  email?: string,
  // 任意属性：索引签名
  [prop: string]: any
}
```

type与interface的区别

type可以为基本类型、联合类型、元组类型定义别名，而interface不行

```
type N = number // 基本类型
type A = string | number // 联合类型
type B = [number, string] // 元组类型
```

```
interface A {
  name: string,
}

interface A {
  age: number
}

const value: A = {
  name: '小明',
  age: 18
}
```

```
type A = {
  name: string,
}

type A = {
  age: number
}

const value: A = {
  name: '小明',
  age: 18
}

// error: 重复定义类型A
```

type与interface的扩展(& 与 extends)

interface从interface扩展

```
interface A { x: number }

interface B extends A { y: number }

let value:B = { x: 1, y: 2 }
```

interface从type扩展

```
type A = {
  x: number
}

interface B extends A { y: number }

let value:B = { x: 1, y: 2 }
```

type从type扩展

```
type A = { x: number }

type B = A & { y: number }

let value:B = { x: 1, y: 2 }
```

type从interface扩展

```
interface A { x: number }

type B = A & { y: number }

let value:B = { x: 1, y: 2 }
```

WARNING：type扩展仅能使用&，interface扩展可以使用extends 或者 &

type & (type 或者 interface)

interface & (type 或者 interface)

interface extends (type 或者interface)

type与interface的使用场景

type

- 定义基本类型的别名时
- 定义元组类型时
- 定义函数类型时

```
type StringReturnFn = (name: string) => string

let test: StringReturnFn = function (name: string):string {
  // ...
  return name
}
```

- 定义联合类型时

interface

- 需要利用接口自动合并特性的时
- 定义对象类型时

联合类型与交叉类型

- 联合类型：产生一个包含所有类型的选择集
- 交叉类型：产生一个所有属性的新类型

联合类型（|）

```
type EmptyValue = null | undefined | ''
// null、undefined、''
const v: EmptyValue = null

// 定义对象
type A = { name: string}
type B = { age: number }
type C = A | B

let o: C = { name: '小明', age: 20 }
```

交叉类型（&）：用于对象

```
type A = { name: string}
type B = { age: number }
type C = A & B

let o: C = { name: '小明', age: 20 }
```

联合类型与交叉类型

WARNING

当有多个类型中有重复属性且类型不同时，联合类型会表现正常，交叉类型会表现异常

```
type A = {
  value: string
}
type B = {
  value: boolean
}

// 联合类型
type C = A | B
let v:C = {
  value: 'hello'
}
v = {
  value: true
}

// 交叉类型
type D = A & B // 此时D为never类型
```

泛型T

```
function sayHi(value: string): string {
  console.log(value)
  return value
}

sayHi('hi')
```

```
function sayHi<T>(value: T): T {
  console.log(value)
  return value
}

sayHi<string>('hi')
```

多个泛型参数

```
function saySomething<T, U>(value: T, msg: U) : T {
  console.log(msg)
  return value
}

// 显示指定泛型变量的类型 T => string 、U => number
saySomething<string, number>('你好', 20)
// 让typescript自己去推导泛型变量的类型
saySomething('你好', 20)
```

泛型变量不一定只能用T表示，理论上可以用任意字母

常见的有

- T (Type)： 表示类型
- K (Key)： 表示对象的键的类型
- V (Value)：表示对象中的值的类型

类型常见操作符

typeof

判断基本类型

```
let s = 'hello'
let n = 1
let b = true

typeof s === 'string'  // true
typeof n === 'number'  // true
typeof b === 'boolean' // true
```

获取变量的声明或者对象的类型

```
interface P {
  name: string
  age: number
}
const a: P = {
  name: '小明',
  age: 20
}
type S = typeof somebody // Person

const obj = { name: '小明', age: 20 }
type O = typeof obj // { name: sting, age: number }
```

in：常用于`映射类型`中遍历枚举值

```
type Keys = 'name' | 'age' | 'hobby'
type Info = {
  [prop in Keys]: string
}

// { 'name': string, 'age': 'string', 'hobby': sting }
```

keyof：常用于`映射类型`中遍历key

```
interface Person { name: string; age: number }
type P = {
  [ name in keyof Person ]: Person[name]
}
// { name: string, age: number }
```

类型常见操作符

extends

用于`条件类型`进行类型推断

```
type IsNumber<T> = T extends number ? true : false

type I = IsNumber<number> // true
```

用于`类型扩展`

```
interface A {
  name: string
}

interface B extends A {
  age: number
}

// B => { name: string; age: number }
```

用于`泛型约束`

```
interface Person {
  name: string
  age: number
}

function getInfo<T extends Person>(info: T): T {
  console.log(info)
  return info
}

// error: age类型错误
getInfo({ name: '1', age: '你好', h: 1 })
```

类型常见操作符

infer

用于`条件类型`中声明一个`类型变量`并对其使用

```
// 获取数组元素类型
type GetValueType<T> = T extends (infer U)[] ? U : T
type StingArray = string[]

type ValueType = GetValueType<StingArray>

// string[] => T
// sting[] extends (infer U)[]  执行类型匹配
// 匹配成功并返回类型U => U => sting
```

```
// 获取函数的返回值类型
type getFnReturnType<T> = T extends (value: any) => (infer U) ?

type StringFn = () => string

// string
type ReturnValueType = getFnReturnType<StringFn>
```

WARNING

- 仅能在条件类型的extends的子句使用

```
// error
(infer U)[] extends T ? U : T
```

- 仅能在条件类型的true分支中使用

```
// error
T extends (infer U)[] ? T : U
```

断言

有时候你会比编译器更清楚的知道某个变量的类型，此时就可以通过`断言`告诉编译器

```ts
function getLength(value: unknown): number {
  return value.length // error
}
```

```ts
// as语法
function getLength(value: unknown): number {
  return ( value as [] ).length
}

// <>尖括号语法
function getLength(value: unknown): number {
  return ( value as Array<any> ).length
}
```

非空断言!

```ts
function test(value: string | undefined | null) {
  const res: string = value!
  return res
}
```

确定赋值断言!

```ts

let s: string
function init () {
  s = 'hello'
}

init()

const num = s.length // error
const num = s!.length // ok
```

装饰器(@)

装饰器是一个函数，可以注入到类、方法、属性、参数上来扩展其功能

类装饰器：在类声明之前声明，应用于类构造函数

普通装饰器

```
// target: 装饰的类 => Example
function sayHi(target: Function): void {
  target.prototype.sayHi = () => {
    console.log('say hi')
  }
}

@sayHi
class Example {
  constructor() {}
}

const ex = new Example();
(ex as any).sayHi() // say hi
```

装饰器工厂：可以传参

```
// target: 装饰的类 => Example
function sayHi(msg: string): Function {
  return function(target: Function) {
    target.prototype.sayHi = () => {
      console.log(msg)
    }
  }
}

@sayHi('你好吃饭了吗')
class Example {
  constructor() {}
}

const ex = new Example();
(ex as any).sayHi() // 你好吃饭了吗
```

装饰器(@)

属性装饰器

```
/**
  target(装饰的类):
    如果是static静态属性是构造函数，如果是public属性是原型对象
  key(装饰的属性名称): name
*/
function changeName(target: any, key: string) {
  target[key] = '小明'
}

class Person {
  @changeName
  public name: string | undefined;

  constructor() {}
}

const p = new Person()
console.log(p.name) // 小明
```

参数装饰器

```
/**
  target(装饰类): Person;
  key(装饰的方法名): test;
  parameterIndex(装饰的参数位置索引值): 1
*/
function Log(target: Function, key: string, parameterIndex: numb
  console.log(target, key, parameterIndex)
}

class Person {
  constructor() {}
  public name: string | undefined
  public age: number | undefined

  test(name: string, @Log age: number) {
    this.name = name;
    this.age = age;
  }
}

const n = new Person();
(n as any).test('小明', 18)
```

装饰器(@)

方法装饰器

```
/**
 * target(被装饰的类): 如果是static静态方法是构造函数，如果是public公共方法是原型对象
 * key(装饰的方法名称): doSomething
 * descriptor: { "writable": true,"enumerable": false,"configurable": true, "value": function }
 */
function logInfo(target: any, key: string, descriptor: PropertyDescriptor) {
  let originMethod = descriptor.value; // 获取到key方法
  descriptor.value = function(...args: any[]) { // 更改key方法
    console.log(`${key}方法执行之前`)
    const result = originMethod.apply(this, args)
    console.log(`${key}方法执行之后`)
    return result
  }
}
class Person {
  constructor() {}
  @logInfo
  doSomething(args: any) {
    console.log('doSomething方法执行')
  }
}
const p = new Person();
(p as any).doSomething();  // doSomething方法执行之前 doSomething方法执行 doSomething方法执行之后
```

Thanks !