

TypeScript学习笔记

TypeScript

概述

官网

安装

动态类型的问题

在Vue中使用ts

入门

类型

类型标注位置

标注变量

标注参数

标注返回值

复杂类型

type

interface

可选属性

鸭子类型

方法类型

字面量类型

nullish 类型

泛型

类

基本语法

访问修饰符

只读属性

方法

get和set

类与接口

继承与接口

方法重写

TypeScript

概述

TypeScript是微软开发的一个开源的编程语言，通过在JavaScript的基础上添加静态类型定义构建而成。TypeScript通过TypeScript编译器或Babel转译为JavaScript代码，可运行在任何浏览器，任何操作系统。

TypeScript 起源于使用JavaScript开发的大型项目。由于JavaScript语言本身的局限性，难以胜任大型项目的开发和维护。因此微软开发了TypeScript，使得其能够胜任大型项目的开发。

TypeScript可以在任何浏览器运行、任何计算机和任何操作系统上运行，并且是开源的

TypeScript与js相比的优势：

- TypeScript工具使重构变更的容易、快捷。
- TypeScript 引入了 JavaScript 中没有的“类”概念。
- TypeScript 中引入了模块的概念，可以把声明、数据、函数和类封装在模块中
- 类型安全功能能在编码期间检测错误，这为开发人员创建了一个更高效的编码和调试过程。

官网

网站首页：<https://www.tslang.cn/index.html>

文档地址：<https://www.tslang.cn/docs/home.html>

安装

npm全局安装：

```
1 | npm install -g typescript
```

也可以指定版本：

```
1 | npm install -g typescript@4.1.5
```

```

1 PS C:\Users\mao\Desktop> npm install -g typescript@4.1.5
2 C:\Users\mao\AppData\Roaming\npm\tsc ->
  C:\Users\mao\AppData\Roaming\npm\node_modules\typescript\bin\tsc
3 C:\Users\mao\AppData\Roaming\npm\tsserver ->
  C:\Users\mao\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
4 + typescript@4.1.5
5 updated 1 package in 10.618s
6 PS C:\Users\mao\Desktop> tsc -v
7 version 4.1.5
8 PS C:\Users\mao\Desktop>

```

```

1 PS C:\Users\mao\Desktop> tsc --help
2 version 4.1.5
3 Syntax:  tsc [options] [file...]
4
5 Examples: tsc hello.ts
6           tsc --outFile file.js file.ts
7           tsc @args.txt
8           tsc --build tsconfig.json
9
10 Options:
11 -h, --help                Print this message.
12 -w, --watch               Watch input files.
13 --pretty                 Stylize errors and
                           messages using color and context (experimental).
14 --all                    Show all compiler
                           options.
15 -v, --version            Print the compiler's
                           version.
16 --init                   Initializes a TypeScript
                           project and creates a tsconfig.json file.
17 -p FILE OR DIRECTORY, --project FILE OR DIRECTORY  Compile the project
                           given the path to its configuration file, or to a folder with a
                           'tsconfig.json'.
18 -b, --build              Build one or more
                           projects and their dependencies, if out of date
19 -t VERSION, --target VERSION  Specify ECMAScript
                           target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017',
                           'ES2018', 'ES2019', 'ES2020', or 'ESNEXT'.
20 -m KIND, --module KIND  Specify module code
                           generation: 'none', 'commonjs', 'amd', 'system', 'umd', 'es2015', 'es2020',
                           or 'ESNext'.
21 --lib                    Specify library files to
                           be included in the compilation.

```

22	'es5' 'es6' 'es2015'	
	'es7' 'es2016' 'es2017' 'es2018' 'es2019' 'es2020' 'esnext' 'dom'	
	'dom.iterable' 'webworker' 'webworker.importscripts' 'webworker.iterable'	
	'scripthost' 'es2015.core' 'es2015.collection' 'es2015.generator'	
	'es2015.iterable' 'es2015.promise' 'es2015.proxy' 'es2015.reflect'	
	'es2015.symbol' 'es2015.symbol.wellknown' 'es2016.array.include'	
	'es2017.object' 'es2017.sharedmemory' 'es2017.string' 'es2017.intl'	
	'es2017.typedarrays' 'es2018.asyncgenerator' 'es2018.asynciterable'	
	'es2018.intl' 'es2018.promise' 'es2018.regexp' 'es2019.array'	
	'es2019.object' 'es2019.string' 'es2019.symbol' 'es2020.bigint'	
	'es2020.promise' 'es2020.sharedmemory' 'es2020.string'	
	'es2020.symbol.wellknown' 'es2020.intl' 'esnext.array' 'esnext.symbol'	
	'esnext.asynciterable' 'esnext.intl' 'esnext.bigint' 'esnext.string'	
	'esnext.promise' 'esnext.weakref'	
23	<code>--allowJs</code>	Allow javascript files
	to be compiled.	
24	<code>--jsx</code> KIND	Specify JSX code
	generation: 'preserve', 'react-native', or 'react'.	
25	<code>-d, --declaration</code>	Generates corresponding
	' <code>.d.ts</code> ' file.	
26	<code>--declarationMap</code>	Generates a sourcemap
	for each corresponding ' <code>.d.ts</code> ' file.	
27	<code>--sourceMap</code>	Generates corresponding
	' <code>.map</code> ' file.	
28	<code>--outFile</code> FILE	Concatenate and emit
	output to single file.	
29	<code>--outDir</code> DIRECTORY	Redirect output
	structure to the directory.	
30	<code>--removeComments</code>	Do not emit comments to
	output.	
31	<code>--noEmit</code>	Do not emit outputs.
32	<code>--strict</code>	Enable all strict type-
	checking options.	
33	<code>--noImplicitAny</code>	Raise error on
	expressions and declarations with an implied ' <code>any</code> ' type.	
34	<code>--strictNullChecks</code>	Enable strict null
	checks.	
35	<code>--strictFunctionTypes</code>	Enable strict checking
	of <code>function</code> types.	
36	<code>--strictBindCallApply</code>	Enable strict ' <code>bind</code> ',
	' <code>call</code> ', and ' <code>apply</code> ' methods on functions.	
37	<code>--strictPropertyInitialization</code>	Enable strict checking
	of property initialization <code>in</code> classes.	
38	<code>--noImplicitThis</code>	Raise error on ' <code>this</code> '
	expressions with an implied ' <code>any</code> ' type.	
39	<code>--alwaysStrict</code>	Parse <code>in</code> strict mode and
	emit " <code>use strict</code> " for each <code>source</code> file.	
40	<code>--noUnusedLocals</code>	Report errors on unused
	locals.	
41	<code>--noUnusedParameters</code>	Report errors on unused
	parameters.	
42	<code>--noImplicitReturns</code>	Report error when not
	all code paths <code>in function</code> return a value.	
43	<code>--noFallthroughCasesInSwitch</code>	Report errors <code>for</code>
	fallthrough cases <code>in</code> switch statement.	

```
44  --types                                Type declaration files
    to be included in compilation.
45  --esModuleInterop                      Enables emit
    interoperability between CommonJS and ES Modules via creation of namespace
    objects for all imports. Implies 'allowSyntheticDefaultImports'.
46  @<file>                                Insert command line
    options and files from a file.
47  PS C:\Users\mao\Desktop>
```

动态类型的问题

js 属于动态类型语言

如下一个函数：

```
1  function test(obj) {
2  }
```

传递过去的参数可能是一个字符串：

```
1  test('hello, world')
```

也可能是一个对象：

```
1  test({a:1,b:2})
```

也有可能是个函数：

```
1  test(()=>{})
```

obj 类型不确定，就给后期使用者带来了麻烦，一旦参数传不对，代码可能出现问题

动态类型是在代码运行时才知道是什么

静态类型是在代码运行前，就对它的行为做出预测

在Vue中使用ts

全新项目：使用vue cli脚手架工具创建vue项目时，勾选 ts

已有项目：添加vue官方配置的ts适配插件，使用@vue/cli 安装 ts插件

```
1 | vue add @vue/typescript
```

或者手动安装：

```
1 | npm install ts-loader typescript tslint tslint-loader tslint-config-standard  
   | --save-dev
```

```
1 | npm install vue-class-component vue-property-decorator --save
```

- vue-class-component：扩展vue支持typescript，将原有的vue语法通过声明的方式来支持ts
- vue-property-decorator：基于vue-class-component扩展更多装饰器
- ts-loader：让webpack能够识别ts文件
- tslint-loader：tslint用来约束文件编码
- tslint-config-standard：tslint 配置 standard风格的约束

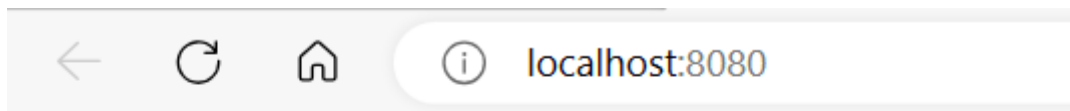
在vue中使用要加上 lang="ts"

```
1 | <template>  
2 |  
3 | </template>  
4 |  
5 | <script lang="ts">  
6 | export default {  
7 |   name: "View1",  
8 | }  
9 | </script>  
10 |  
11 | <style scoped>  
12 |  
13 | </style>
```

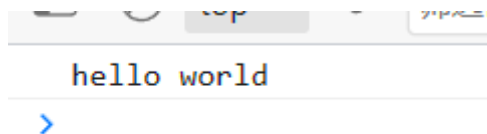
入门

输出helloworld，并打印显示helloworld

```
1 <template>
2   <div>
3     <h1>{{ str }}</h1>
4   </div>
5 </template>
6
7 <script setup lang="ts">
8
9   const str: string = 'hello world';
10
11   function hello(str: string)
12   {
13     console.log(str);
14   }
15
16   hello(str);
17
18
19 </script>
20
21 <style scoped>
22
23 </style>
```



hello world



用 interface 定义用户类型:

```
1 <template>
2   <div>
```

```

3     <h1>{{ user }}</h1>
4   </div>
5 </template>
6
7 <script setup lang="ts">
8   interface User
9   {
10     id: number
11     name: string,
12     age: number,
13   }
14
15   function printUser(user: User): void
16   {
17     console.log("user.id=" + user.id)
18     console.log("user.name=" + user.name)
19     console.log("user.age=" + user.age)
20   }
21
22   const user: User = {
23     id: 10001,
24     name: '张三',
25     age: 19
26   }
27
28   printUser(user);
29
30 </script>
31
32 <style scoped>
33
34 </style>
35

```



{ "id": 10001, "name": "张三", "age": 19 }

```

user.id=10001
user.name=张三
user.age=19
>

```


类型

类型	例	备注
字符串类型	string	
数字类型	number	
布尔类型	boolean	
数组类型	number[],string[], boolean[] 依此类推	
任意类型	any	相当于又回到了没有类型的时代
复杂类型	type 与 interface	
函数类型	() => void	对函数的参数和返回值进行说明
字面量类型	"a" "b" "c"	限制变量或参数的取值
nullish类型	null 与 undefined	
泛型	<T>, <T extends 父类型>	

类型标注位置

标注变量

```
1 | let message: string = 'hello,world'
```

一般可以省略，因为可以根据后面的字面量推断出前面变量类型

```
1 | let message = 'hello,world'
```

标注参数

```
1 function test(name: string) {  
2  
3 }
```

```
<script lang="ts" setup>  
function add(a: number, b: number): number  
{  
    return a + b;  
}  
  
console.log(add(a: 1, b: 2))  
console.log(add(a: 1, b: '11414'))
```

'11414''))

TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

doc: '11414'

标注返回值

```
1 <template>  
2  
3 </template>  
4  
5 <script lang="ts" setup>  
6 function add(a: number, b: number): number  
7 {  
8     return a + b;  
9 }  
10  
11 console.log(add(1, 2))  
12  
13 </script>  
14
```

```
15 <style scoped>
16
17 </style>
```

复杂类型

type

```
1 <template>
2
3 </template>
4
5 <script lang="ts" setup>
6
7   type Student = {
8     id: number,
9     name: string,
10    sex: string,
11    age: number
12  }
13  const student1: Student = {id: 10001, name: "张三", sex: "男", age: 19};
14  //报错, 缺少age
15  const student2: Student = {id: 10001, name: "张三", sex: "男"};
16  //报错, 多了address
17  const student3: Student = {id: 10001, name: "张三", sex: "男", age: 19,
18    address: "中国"};
19
20  console.log(student1);
21  console.log(student2);
22  console.log(student3);
23
24 </script>
25
26 <style scoped>
27
28 </style>
```

```
//报错, 缺少age
const student2: Student = {id: 10001, name: "张三", sex: "男"};
//报错, 多了
const stud

console.log
console.log
console.log
```

TS2741: Property 'age' is missing in type '{ id: number; name: string; sex: string; }' but required in type 'Student'.

View4.vue(11, 3): 'age' is declared here.

Add missing properties Alt+Shift+Enter 更多操作... Alt+Enter

const student2: Student

报错, 缺少age

src/view/View4.vue

```
</script>
```

```
age: 19, address: "中国"};
```

TS2322: Type '{ id: number; name: string; sex: string; age: number; address: string; }' is not assignable to type 'Student'.

Object literal may only specify known properties, and 'address' does not exist in type 'Student'.

address: string

src/view/View4.vue

interface

和type区别在于, 少了一个等号

```
1 <template>
2
3 </template>
4
5 <script lang="ts" setup>
6
7 interface Student
8 {
9   id: number,
10  name: string,
11  sex: string,
12  age: number
13 }
14
15 const student1: Student = {id: 10001, name: "张三", sex: "男", age: 19};
16 //报错, 缺少age
17 const student2: Student = {id: 10001, name: "张三", sex: "男"};
18 //报错, 多了address
```

```

19  const student3: Student = {id: 10001, name: "张三", sex: "男", age: 19,
    address: "中国"};
20
21
22  console.log(student1);
23  console.log(student2);
24  console.log(student3);
25
26  </script>
27
28  <style scoped>
29
30  </style>

```

可选属性

如果需要某个属性可选，可以用下面的语法

```

1  <template>
2
3  </template>
4
5  <script lang="ts" setup>
6
7  interface Student
8  {
9      id: number,
10     name: string,
11     sex: string,
12     age?: number
13 }
14
15 const student1: Student = {id: 10001, name: "张三", sex: "男", age: 19};
16 //并不会报错，缺少age，但是会出现undefined
17 const student2: Student = {id: 10001, name: "张三", sex: "男"};
18 console.log(student1);
19 console.log(student2);
20 console.log(student2.age)
21
22 </script>
23
24 <style scoped>
25
26 </style>
27

```

```
console.log(student1);
console.log(student2);
console.log(student2.)

</script>
<style scoped>
</style>
```

id	number
sex	string
age	number undefined
name	string
const	const name = expr
var	var name = expr
arg	functionCall(expr)
cast	(<any>value)
let	let name = expr
not	!expr

▼ {id: 10001, name: '张三', sex: '男', age: 19} ⓘ

age: 19

id: 10001

name: "张三"

sex: "男"

▶ [[Prototype]]: Object

▼ {id: 10001, name: '张三', sex: '男'} ⓘ

id: 10001

name: "张三"

sex: "男"

▶ [[Prototype]]: Object

undefined

>

鸭子类型

```
1 <template>
2
3 </template>
4
5 <script lang="ts" setup>
6
7 interface Student
8 {
9   id: number,
10  name: string,
11  sex: string,
```

```

12     age: number
13   }
14
15   const student1: Student = {id: 10001, name: "张三", sex: "男", age: 19};
16
17   //报错, 多了address
18   //const student3: Student = {id: 10001, name: "张三", sex: "男", age: 19,
19   //  address: "中国"};
20
21   const student3 = {id: 10001, name: "张三", sex: "男", age: 19, address: "中
22   国"};
23   //鸭子类型, student3并没有声明类型为Student, 但它与 Student 类型有一样的属性, 也可以被
24   //当作是 Student 类型
25
26   console.log(student1);
27   console.log(student3);
28
29   </script>
30
31   <style scoped>
32
33   </style>

```

student3并没有声明类型为Student, 但它与 Student 类型有一样的属性, 也可以被当作是 Student 类型

方法类型

interface中包含方法 (函数)

```

1   <template>
2     <div>
3       <h2>{{ user }}</h2>
4     </div>
5   </template>
6
7   <script lang="ts" setup>
8
9   import {onBeforeMount, onMounted} from "vue";
10
11   interface User
12   {
13     id: number,
14     name: string,
15     age: number,
16
17     getName(): string
18

```

```

19   getAgeString(): string
20
21   setName(name: string): void
22 }
23
24 const user: User = {
25   id: 10002,
26   name: "张三",
27   age: 12,
28   getName(): string
29   {
30     return this.name
31   },
32   getAgeString(): string
33   {
34     if (this.age < 0 || this.age > 120)
35     {
36       return "年龄输入错误"
37     }
38     if (this.age < 18)
39     {
40       return "未成年";
41     }
42     if (this.age < 30)
43     {
44       return "青年"
45     }
46     if (this.age < 60)
47     {
48       return "中年"
49     }
50     return "老年"
51   },
52   setName(name: string)
53   {
54     this.name = name;
55   }
56 }
57
58 onBeforeMount(()=>
59 {
60   console.log(user.getName())
61   console.log(user.getAgeString())
62   user.setName("李四")
63   console.log(user.getName())
64 })
65
66 </script>
67
68 <style scoped>
69
70 </style>

```


张三
未成年
李四
>

localhost:8080

{ "id": 10002, "name": "李四", "age": 12 }

字面量类型

```
1 <template>
2   <div>
3
4   </div>
5 </template>
6
7 <script lang="ts" setup>
8
9   /**
10    * 打印字符串到控制台
11    * @param str 字符串
12    * @param alignment 对齐方式，只能取值left、right和center
13    */
14   function print(str: string, alignment: "left" | "right" | "center")
15   {
16     console.log(str, alignment)
17   }
18
19   print("hello", "left")
20   print("hello", "right")
21   print("hello", "center")
22   //以下报错: Argument of type '"131412351"' is not assignable to parameter of
23   type '"left" | "right" | "center"'
24   print("hello", "131412351")
25
26 </script>
27
28 <style scoped>
```

```
29  
30 </style>
```

```
of type '"131412351"' is not assignable to parameter of type '"left" | "right"  
alignment: "131412351")
```

```
TS2345: Argument of type '"131412351"' is not assignable to parameter of type '"left" | "right" | "center"'.  
doc: "131412351"
```

nullish 类型

在冒号前面加一个问号，表示字段可以为空

```
1 <template>  
2  
3 </template>  
4  
5 <script lang="ts" setup>  
6  
7 interface StudentV1  
8 {  
9   id: number,  
10  name: string,  
11  sex: string,  
12  age: number  
13 }  
14  
15 interface StudentV2  
16 {  
17   id: number,  
18   name: string,  
19   sex?: string,  
20   age?: number  
21 }  
22  
23 //Property 'age' is missing in type '{ id: number; name: string; sex:  
string; }' but required in type 'StudentV1'.  
24 const studentV1: StudentV1 = {id: 10001, name: "张三", sex: "男"};  
25 //Type 'undefined' is not assignable to type 'number'.  
26 const student2V1: StudentV1 = {id: 10001, name: "张三", sex: "男", age:  
undefined};  
27 //并不会报错  
28 const studentV2: StudentV2 = {id: 10001, name: "张三"};  
29 const student2V2: StudentV2 = {id: 10001, name: "张三",sex:undefined};  
30
```

```
31 console.log(studentV1)
32 console.log(studentV2)
33 console.log(student2V1)
34 console.log(student2V2)
35
36 </script>
37
38 <style scoped>
39
40 </style>
```

```
1 <template>
2
3 </template>
4
5 <script lang="ts" setup>
6
7 function toUpperCase(str?: string | null): string
8 {
9   return str?.toUpperCase() || "无效字符串"
10 }
11
12 function toUpperCase2(str: string): string
13 {
14   return str.toUpperCase()
15 }
16
17 console.log(toUpperCase("hello"))
18 console.log(toUpperCase(null))
19 console.log(toUpperCase())
20
21 console.log(toUpperCase2("hello2"))
22 //Argument of type 'null' is not assignable to parameter of type 'string'.
23 console.log(toUpperCase2(null))
24 //Expected 1 arguments, but got 0.
25 console.log(toUpperCase2())
26
27 </script>
28
29 <style scoped>
30
31 </style>
```



泛型

下面的几个类型声明显然有一定的相似性

```
1 interface RefString {
2   value: string
3 }
4
5 interface RefNumber {
6   value: number
7 }
8
9 interface RefBoolean {
10   value: boolean
11 }
12
13 const r1: RefString = { value: 'hello' }
14 const r2: RefNumber = { value: 123 }
15 const r3: RefBoolean = { value: true }
```

可以改进为：

```
1  <template>
2
3  </template>
4
5  <script lang="ts" setup>
6
7  interface Ref<T>
8  {
9    value: T,
10
11    /**
12     * 得到value值
13     */
14    getValue(): T
15  }
16
17  const r1: Ref<string> = {
18    value: 'hello', getValue()
19    {
20      return this.value
21    }
22  }
23  const r2: Ref<number> = {
24    value: 123, getValue()
25    {
26      return this.value
27    }
28  }
29  const r3: Ref<boolean> = {
30    value: true, getValue()
31    {
32      return this.value
33    }
34  }
35
36  console.log(r1.getValue())
37  console.log(r2.getValue())
38  console.log(r3.getValue())
39
40  </script>
41
42  <style scoped>
43
44  </style>
```

```

console.log(r1.)
console.log(r1.m)
console.log(r1.p)
const
var

```

getValue()	string
value	string
const name = expr	
var name = expr	

```

console.log(r1.getValue())
console.log(r2.)
console.log(r2.m)
console.log(r2.p)
const
var
</script>
<style scoped>
arg
cast

```

getValue()	number
value	number
const name = expr	
var name = expr	
functionCall(expr)	
(<any>value)	

```

console.log(r2.getValue())
console.log(r3.)
const
var
arg
cast
let
</script>
<style scoped>
</style>
script

```

value	boolean
getValue()	boolean
const name = expr	
var name = expr	
functionCall(expr)	
(<any>value)	
let name = expr	

```

hello
123
true
>

```

函数定义也支持泛型:

```

1 <template>

```

```
2
3 </template>
4
5 <script lang="ts" setup>
6
7 function ref<T>(arg: T): T
8 {
9   return arg;
10 }
11
12 console.log(ref("hello"))
13 console.log(ref(12345))
14 console.log(ref(true))
15 console.log(ref(null))
16 console.log(ref(undefined))
17
18 console.log(typeof ref("hello"))
19 console.log(typeof ref(12345))
20 console.log(typeof ref(true))
21 console.log(typeof ref(null))
22 console.log(typeof ref(undefined))
23
24 </script>
25
26 <style scoped>
27
28 </style>
```

hello

12345

true

null

undefined

string

number

boolean

object

undefined

>

类

基本语法

```
1  <template>
2    <div>
3      <h2>{{stu1}}</h2>
4      <h2>{{stu2}}</h2>
5      <h2>{{stu3}}</h2>
6      <h2>{{stu4}}</h2>
7
8
9    </div>
10 </template>
11
12 <script setup lang="ts">
13
14 /**
15  * 学生类
16  */
17 class Student
18 {
19   /**
20    * id
21    */
22   id: number;
23   /**
24    * 姓名
25    */
26   name: string;
27   /**
28    * 性别
29    */
30   sex: string;
31
32
33   /**
34    * 无参构造方法
35    */
36   constructor();
37   /**
38    *
39    * @param id 学生学号
40    */
41   constructor(id: number);
42   /**
43    *
44    * @param id 学生学号
45    * @param name 姓名
```



```

46     */
47     constructor(id: number, name: string);
48     /**
49     *
50     * @param id 学生学号
51     * @param name 姓名
52     * @param sex 性别
53     */
54     constructor(id: number, name: string, sex: string);
55     /**
56     *
57     * @param id 学生学号
58     * @param name 姓名
59     * @param sex 性别
60     */
61     constructor(id?: number, name?: string, sex?: string)
62     {
63         console.log("构造方法被调用了")
64         this.id = id ? id : 10001
65         this.name = name ? name : "张三"
66         this.sex = sex ? sex : "男"
67     }
68 }
69
70 const stu1: Student = new Student()
71 console.log(stu1)
72
73 const stu2: Student = new Student(102222)
74 console.log(stu2)
75
76 const stu3: Student = new Student(102223, "李四")
77 console.log(stu3)
78
79 const stu4: Student = new Student(102224, "王五", '女')
80 console.log(stu4)
81
82 console.log(stu4.name)
83
84 </script>
85
86 <style scoped>
87
88 </style>
89

```

```
{ "id": 10001, "name": "张三", "sex": "男" }  
{ "id": 102222, "name": "张三", "sex": "男" }  
{ "id": 102223, "name": "李四", "sex": "男" }  
{ "id": 102224, "name": "王五", "sex": "女" }
```

构造方法被调用了

```
▶ Student {id: 10001, name: '张三', sex: '男'}
```

构造方法被调用了

```
▶ Student {id: 102222, name: '张三', sex: '男'}
```

构造方法被调用了

```
▶ Student {id: 102223, name: '李四', sex: '男'}
```

构造方法被调用了

```
▶ Student {id: 102224, name: '王五', sex: '女'}
```

王五

>

js 中的 class，并不等价于 java 中的 class，它还是基于原型实现的

访问修饰符

有三类：

- public
- protected
- private

默认为 public，可以自由的访问程序里定义的成员

当成员被标记成 private 时，它就不能在声明它的类的外部访问

protected修饰符与 private修饰符的行为很相似，但有一点不同，protected成员在派生类中仍然可以访问

可以参考java

```
1  <template>
2    <div>
3      <h2>{{ stu4 }}</h2>
4    </div>
5  </template>
6
7  <script setup lang="ts">
8
9    /**
10     * 学生类
11     */
12    class Student
13    {
14      /**
15       * id
16       */
17      private id: number;
18      /**
19       * 姓名
20       */
21      protected name: string;
22      /**
23       * 性别
24       */
25      public sex: string;
26
27
28      /**
29       * 无参构造方法
30       */
31      constructor();
32      /**
33       *
34       * @param id 学生学号
35       */
36      constructor(id: number);
37      /**
38       *
39       * @param id 学生学号
40       * @param name 姓名
41       */
42      constructor(id: number, name: string);
43      /**
44       *
45       * @param id 学生学号
46       * @param name 姓名
47       * @param sex 性别
```

```

48     */
49     constructor(id: number, name: string, sex: string);
50     /**
51      *
52      * @param id 学生学号
53      * @param name 姓名
54      * @param sex 性别
55      */
56     constructor(id?: number, name?: string, sex?: string)
57     {
58         console.log("构造方法被调用了")
59         this.id = id ? id : 10001
60         this.name = name ? name : "张三"
61         this.sex = sex ? sex : "男"
62     }
63 }
64
65 const stu4: Student = new Student(102224, "王五", '女')
66
67 console.log(stu4)
68
69 //Property 'id' is private and only accessible within class 'Student'.
70 console.log(stu4.id)
71 //Property 'name' is protected and only accessible within class 'Student'
    and its subclasses.
72 console.log(stu4.name)
73 console.log(stu4.sex)
74
75 </script>
76
77 <style scoped>
78
79 </style>

```

```

//Property 'id' is private and only accessible within class 'Student'.
console.log(stu4.id)
//Property 'name'
console.log(stu4.n
console.log(stu4.s

```

TS2341: Property 'id' is private and only accessible within class 'Student'. s 'Student'

doc: id

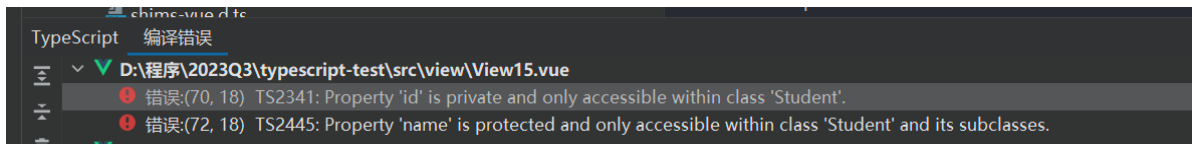
```

stu4.name)
stu4.se

```

TS2445: Property 'name' is protected and only accessible within class 'Student' and its subclasses.

doc: name



只读属性

readonly 是 typescript 特有的，表示该属性只读

```
1  <template>
2
3  </template>
4
5  <script lang="ts" setup>
6
7  class User
8  {
9      /**
10       * 用户编号，只读
11       */
12     readonly id: number;
13     /**
14      * 用户名称
15      */
16     name: string | undefined;
17
18     constructor()
19     {
20         this.id = 10001;
21     }
22 }
23
24 const user: User = new User();
25 //Cannot assign to 'id' because it is a read-only property.
26 user.id = 10002;
27 user.name = "张三";
28
29 console.log(user.id)
30 console.log(user.name)
31
32 </script>
33
34 <style scoped>
35
36 </style>
```



方法



```

31  /**
32   * 转json
33   */
34  toJson()
35  {
36      return JSON.stringify(this);
37  }
38
39  }
40
41  const user: User = new User();
42  user.name = "张三";
43
44  console.log(user.id)
45  console.log(user.name)
46
47  console.log(user.toString())
48  console.log(user.toJson())
49
50  </script>
51
52  <style scoped>
53
54  </style>

```

10001

张三

用户编号: 10001, 用户名称: 张三

{"id":10001,"name":"张三"}

>

get和set

```

1  <template>
2
3  </template>
4
5  <script setup lang="ts">
6
7  class Student
8  {
9      private _id: number;
10     private _name: string;
11
12     get name(): string
13     {

```

```

14     console.log("调用name 的get方法")
15     return this._name;
16 }
17
18 set name(value: string)
19 {
20     console.log("调用name 的set方法")
21     this._name = value;
22 }
23
24 get id(): number
25 {
26     console.log("调用id 的get方法")
27
28     return this._id;
29 }
30
31 set id(value: number)
32 {
33     console.log("调用id 的set方法")
34     this._id = value;
35 }
36
37 constructor()
38 {
39     this._id = 10001
40     this._name = "";
41 }
42 }
43
44 const student: Student = new Student();
45
46 //Property '_id' is private and only accessible within class 'Student'.
47 //console.log(student._id)
48 //Property '_name' is private and only accessible within class 'Student'.
49 //console.log(student._name)
50
51 student.id = 99999;
52 student.name = "李四"
53
54 console.log(student.id)
55 console.log(student.name)
56
57 </script>
58
59 <style scoped>
60
61 </style>

```


调用id 的set方法
调用name 的set方法
调用id 的get方法
99999
调用name 的get方法
李四
>

类与接口

```
1  <template>
2    <div>
3      <h2>
4        {{userService.getLoginUser()}}
5      </h2>
6    </div>
7  </template>
8
9  <script setup lang="ts">
10
11    /**
12     * 实体类
13     */
14    interface User
15    {
16      id: number;
17      name: string;
18    }
19
20    /**
21     * 接口
22     */
23    interface UserService
24    {
25      /**
26       * 登录
27       */
28      login(user: User): void
29
30      /**
31       * 得到当前登录人的信息
32       */
33      getLoginUser(): User
34    }
35
36    /**
```

```

37  * 实现类
38  */
39  class UserServiceImpl implements UserService
40  {
41      getLoginUser(): User
42      {
43          return {id: 100001, name: '张三'};
44      }
45
46      login(user: User): void
47      {
48          console.log(user)
49      }
50  }
51
52  const userService: UserService = new UserServiceImpl();
53
54  const loginUser = userService.getLoginUser();
55  console.log(loginUser)
56  userService.login(loginUser);
57
58  console.log()
59
60  </script>
61
62  <style scoped>
63
64  </style>
65

```

```

▼ {id: 100001, name: '张三'} ⓘ
  id: 100001
  name: "张三"
  ► [[Prototype]]: Object
▼ {id: 100001, name: '张三'} ⓘ
  id: 100001
  name: "张三"
  ► [[Prototype]]: Object
>

```

← ↻ 🏠 ⓘ localhost:8080

{ "id": 100001, "name": "张三" }

继承与接口

```
1 <template>
2   <div>
3   </div>
4 </template>
5
6 <script setup lang="ts">
7
8   interface Flyable
9   {
10    fly(): void
11  }
12
13  class Animal
14  {
15    name: string;
16
17    constructor(name: string)
18    {
19      this.name = name
20    }
21  }
22
23  class Bird extends Animal implements Flyable
24  {
25    fly()
26    {
27      console.log(`${this.name}在飞翔`)
28    }
29  }
30
31  const b: Flyable & Animal = new Bird("小黄鸟")
32  b.fly()
33
34 </script>
35
36 <style scoped>
37
38 </style>
```

小黄鸟在飞翔



方法重写

```
1  <template>
2    <div>
3    </div>
4  </template>
5
6  <script setup lang="ts">
7
8    class C1
9    {
10      study()
11      {
12        console.log("C1 study")
13      }
14    }
15
16    class C2 extends C1
17    {
18      study()
19      {
20        super.study();
21        console.log("C2 study")
22      }
23    }
24
25    let c: C1 = new C2()
26    c.study();
27
28    c = new C1()
29    c.study();
30
31
32  </script>
33
34  <style scoped>
35
36  </style>
```

C1 study

C2 study

C1 study

>

end

2023 06 29
