TypeScript

资源

1. TypeScript参考
2. vue中的TypeScript

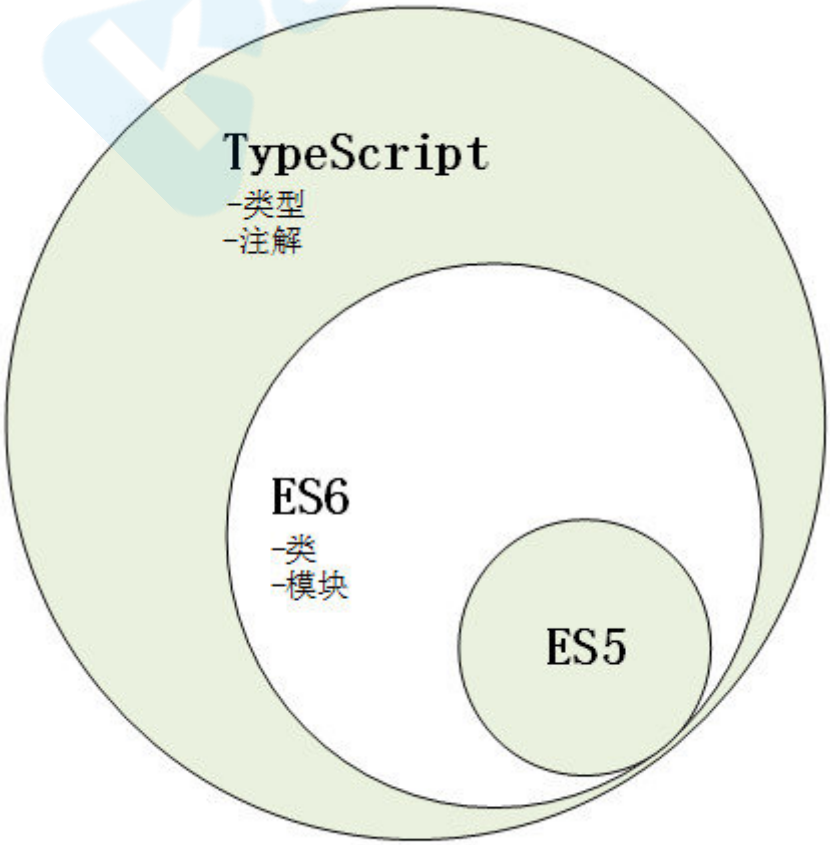
课堂目标

1. 掌握TypeScript核心语法
2. 能使用ts编写vue应用
3. 掌握装饰器原理
4. 能看ts编写的源码

知识点

TypeScript核心语言特性

TypeScript是JavaScript的超集，它可编译为纯JavaScript，是一种给 JavaScript 添加特性的语言扩展。



ES5、ES6和TypeScript
开课吧web全栈架构师

TS有如下特点：

- 类型注解和编译时类型检查
- 基于类的面向对象编程
- 泛型
- 接口
- 声明文件
- ...

TS开发环境搭建

安装typescript并初始化配置

```
npm i typescript -g
tsc --init
npm init -y
```

编写测试代码，./src/index.ts

```
const hello = "hello, typescript!";
console.log(hello);
```

编译

```
tsc ./src/index.ts
```

工程化，安装webpack, webpack-cli, webpack-dev-server

```
npm i webpack webpack-cli webpack-dev-server ts-loader typescript html-webpack-plugin -D
```

配置文件：build/webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
    entry: './src/index.ts',
    output: {
        filename: 'app.js'
    },
    resolve: {
        extensions: ['.js', '.ts', '.tsx']
    },
    devtool: 'cheap-module-eval-source-map',
    module: {
        rules: [
            {
                test: /\.tsx?$/i,
                use: [{
                    loader: 'ts-loader'
                }],
```

```
                exclude: /node_modules/
            }
        ]
    },
    plugins: [
        new HtmlWebpackPlugin({
            template: './public/index.html'
        })
    ]
}
```

> 创建宿主页面：public/index.html

添加开发脚本，package.json

```
"scripts": {
    "dev": "webpack-dev-server --config ./build/webpack.config.js"
}
```

类型注解和编译时类型检查

类型注解

变量后面通过**冒号+类型**来做类型注解

```
// 01-types.ts
let var1: string; // 类型注解

// 编译器类型推断可省略这个语法
let var2 = true;
```

类型基础

```
// 数组
let arr: string[];
arr = ['Tom']; // 或Array<string>

// 任意类型any
let varAny: any;
varAny = 'xx';
varAny = 3;

// any类型也可用于数组
let arrAny: any[];
arrAny = [1, true, "free"];
arrAny[1] = 100;

// 函数中的类型约束
function greet(person: string): string {
  return 'hello, ' + person;
}
// void类型，常用于没有返回值的函数
function warn(): void {}
```

```
// 对象object：不是原始类型的就是对象类型
function fn1(o: object) {}
fn1({ prop: 0 }); // OK
fn1(1); // Error
fn1("string"); // Error

// 更好的约束方式应该是下面这样
function fn2(o: { prop: number }) {}
fn2({ prop: 0 }) // OK

// 类型别名type：自定义类型
type Prop = { prop: number }
// fn3变得更清爽了
function fn3(o: Prop) {}
```

类型断言

某些情况下用户会比编译器更确定某个变量的具体类型，可用类型断言as

```
const someValue: any = "this is a string";
const strLength = (someValue as string).length;
```

联合类型

希望某个变量或参数的类型是多种类型其中之一

```
let union: string | number;
union = '1'; // ok
union = 1; // ok
```

交叉类型

想要定义某种由多种类型合并而成的类型使用交叉类型

```
type First = {first: number};
type Second = {second: number};
type FirstAndSecond = First & Second;
function fn3(param: FirstAndSecond): FirstAndSecond {
    return {first:1, second:2}
}
```

函数

必填参：参数一旦声明，就要求传递，且类型需符合

```
// 02-function.ts
function greeting(person: string): string {
  return "Hello, " + person;
}
greeting('tom')
```

可选参数：参数名后面加上问号，变成可选参数

```
function greeting(person: string, msg?: string): string {
  return "Hello, " + person;
}
```

默认值

```
function greeting(person: string, msg = ''): string {
  return "Hello, " + person;
}
```

*函数重载：以参数数量或类型区分多个同名函数

```
// 重载1
function watch(cb1: () => void): void;
// 重载2
function watch(cb1: () => void, cb2: (v1: any, v2: any) => void): void;
// 实现
function watch(cb1: () => void, cb2?: (v1: any, v2: any) => void) {
    if (cb1 && cb2) {
        console.log('执行watch重载2');
    } else {
        console.log('执行watch重载1');
    }
}
```

类

class的特性

ts中的类和es6中大体相同，这里重点关注ts带来的访问控制等特性

```
// 03-class.ts
class Parent {
  private _foo = "foo"; // 私有属性，不能在类的外部访问
  protected bar = "bar"; // 保护属性，可以在子类中访问

  // 构造函数参数加修饰符，能够定义为成员属性
  constructor(public tua = "tua") {}

  // 方法也有修饰符
  private someMethod() {}

  // 存取器：属性方式访问，可添加额外逻辑，控制读写性
  get foo() {
    return this._foo;
  }
  set foo(val) {
    this._foo = val;
  }
}

class Child extends Parent {
    baz() {
        this.foo;
```

```
        this.bar;
        this.tua;
    }
}
```

接口

接口仅约束结构，不要求实现，使用更简单

```
// 04-interface
// Person接口定义了解构
interface Person {
  firstName: string;
  lastName: string;
}
// greeting函数通过Person接口约束参数解构
function greeting(person: Person) {
  return 'Hello, ' + person.firstName + ' ' + person.lastName;
}
greeting({firstName: 'Jane', lastName: 'User'}); // 正确
greeting({firstName: 'Jane'}); // 错误
```

泛型

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。以此增加代码通用性。

```
// 05-generics
// 不用泛型
// interface Result {
//   ok: 0 | 1;
//   data: Feature[];
// }

// 使用泛型
interface Result<T> {
  ok: 0 | 1;
  data: T;
}

// 泛型方法
function getResult<T>(data: T): Result<T> {
  return {ok:1, data};
}
// 用尖括号方式指定T为string
getResult<string>('hello')
// 用类型推断指定T为number
getResult(1)

// 进一步约束类型变量
interface Foo {
    foo: string
}
```

```
// 约束T必须兼容Foo
function getResult<T extends Foo>(data: T): Result<T> {
  return {ok:1, data};
}
// 这样上面的两个调用就非法了
```

TS在Vue中的应用

准备工作

新建一个基于ts的vue项目

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, TS, Linter
? Use class-style component syntax? Yes
? Use Babel alongside TypeScript for auto-detected polyfills? Yes
? Pick a linter / formatter config: Basic
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert sele
n)Lint on save
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? (y/N) n
```

在已存在项目中安装typescript

```
vue add @vue/typescript
```

范例：ts特性列表

- 特性列表（类型系统、类等）
- 新增特性（函数）
- 特性总数统计（存取器）
- 异步数据获取（接口、泛型）
- 获取属性、派发新增事件（装饰器）

特性列表

创建models/feature.ts

```
interface Feature {
  id: number;
  name: string;
  version: string;
}
export default Feature;
```

修改components/HelloWorld.vue

```
<template>
  <div>
    <ul>
      <li v-for="feature in features" :key="feature.id">
```

```
        {{feature.name}} <span class="tag">{{feature.version}}</span>
      </li>
    </ul>
  </div>
</template>
<script lang='ts'>
import { Component, Vue } from "vue-property-decorator";
import Feature from "@/models/feature";

@Component
export default class Hello extends Vue {
  features: Feature[] = [{ id: 1, name: "类型注解", version: "1.0" }];
}
</script>
<style scoped>
li {
  padding: 8px;
}
.tag{
  background-color: rgb(30, 151, 199);
  color: white;
  border-radius: 4px;
  padding: 5px 10px;
}
</style>
```

新增特性

修改components/HelloWorld.vue

```
<template>
  <div>
    <!-- 1.添加输入框 -->
    <div>
      <input type="text" placeholder="输入新特性" @keyup.enter="addFeature" />
    </div>
  </div>
</template>
<script lang='ts'>
@Component
export default class Hello extends Vue {
  // 2.添加回调函数
  addFeature(event: KeyboardEvent) {
    const input = event.target as HTMLInputElement;
    this.features.push({
      id: this.features.length + 1,
      name: input.value,
      version: "1.0"
    });
    input.value = "";
  }
}
</script>
```

利用getter设置计算属性

```
<template>
    <li>特性数量：{{count}}</li>
</template>
<script lang="ts">
    export default class HelloWorld extends Vue {
        // 定义getter作为计算属性
        get count() {
            return this.features.length;
        }
    }
</script>
```

获取异步数据

创建api/feature.ts

```
import axios from 'axios';
import Feature from '@/models/feature';

export function getFeatures() {
    // 通过泛型约束返回值类型，这里是Promise<AxiosResponse<Feature[]>>
    return axios.get<Feature[]>('/api/list')
}
```

> 注意安装axios

装饰器

装饰器用于扩展类或者它的属性和方法。@xxx就是装饰器的写法

属性声明：@Prop

除了在@Component中声明，还可以采用@Prop的方式声明组件属性

```
export default class HelloWorld extends Vue {
  // Props()参数是为vue提供属性选项
  // !称为明确赋值断言，它是提供给ts的
  @Prop({type: String, required: true})
  private msg!: string;
}
```

事件处理：@Emit

新增特性时派发事件通知，Hello.vue

```
// 通知父类新增事件，若未指定事件名则函数名作为事件名（羊肉串形式）
@Emit()
private addFeature(event: any) {// 若没有返回值形参将作为事件参数
    return feature;// 若有返回值则返回值作为事件参数
}
```

变更监测：@Watch

```
@Watch('msg')
onMsgChange(val:string, oldVal:any){
    console.log(val, oldVal);
}
```

vuex推荐使用：[vuex-class]

`vuex-class`为vue-class-component提供Vuex状态绑定帮助方法。

装饰器原理

装饰器是工厂函数，将传入目标

类装饰器，07-decorator.ts

```
//类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。
function log(target: Function) {
    // target是构造函数
    console.log(target === Foo); // true
    target.prototype.log = function() {
      console.log(this.bar);
    }
    // 如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。
}

@log
class Foo {
  bar = 'bar'
}

const foo = new Foo();
// @ts-ignore
foo.log();
```

方法装饰器

```
function dong(target: any, name: string, descriptor: any) {
    // 这里通过修改descriptor.value扩展了bar方法
    const baz = descriptor.value;
    descriptor.value = function(val: string) {
        console.log('dong~~');
```

开课吧web全栈架构师

```
        baz.call(this, val);
    }
    return descriptor
}

class Foo {
  @dong
  setBar(val: string) {
    this.bar = val
  }
}

foo.setBar('lalala')
```

属性装饰器

```
// 属性装饰器
function mua(target, name) {
    target[name] = 'mua~~~'
}
class Foo {
  @mua ns!:string;
}

console.log(foo.ns);
```

稍微改造一下使其可以接收参数

```
function mua(param:string) {
    return function (target, name) {
        target[name] = param
    }
}
```

实战一下Component，新建Decor.vue

```
<template>
  <div>{{msg}}</div>
</template>

<script lang='ts'>
import { Vue } from "vue-property-decorator";

function Component(options: any) {
  return function(target: any) {
    return Vue.extend(options);
  };
}

@Component({
  props: {
```

```
    msg: {
      type: String,
      default: ""
    }
  }
})
export default class Decor extends Vue {}
</script>
```

显然options中的选项都可以从Decor定义中找到，去源码中找答案吧~

作业

1. 把手头的小项目改造为ts编写
2. 探究vue-property-decorator中各装饰器实现原理，能造个轮子更佳