

# TypeScript

## 资源

1. [TypeScript参考](#)
2. [Vue+TypeScript](#)
3. 代码分支: web22-ts

## 知识点

1. ts核心语法
2. ts+vue
3. 装饰器应用
4. 装饰器原理
5. vue-property-decorator源码解析

## 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 selection)
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
```

```
✓ Successfully installed plugin: @vue/cli-plugin-typescript
? Use class-style component syntax? Yes
? Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)? No
? Convert all .js files to .ts? No
? Allow .js files to be compiled? No
```

请暂时忽略引发的几处Error，它们不会影响项目运行，我们将在后面处理它们。

开课吧web全栈架构师

## TS特点

- 类型注解、类型检测
- 类
- 接口
- 泛型
- 装饰器
- 类型声明

## 类型注解和编译时类型检查

使用类型注解约束变量类型，编译器可以做静态类型检查，使程序更加健壮

### 类型基础

```
// ts-test.ts
let var1: string; // 类型注解
var1 = "开课吧"; // 正确
var1 = 4; // 错误

// 编译器类型推断可省略这个语法
let var2 = true;

// 常见原始类型: string, number, boolean, undefined, null, symbol

// 类型数组
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 {}
```

范例, HelloWorld.vue

```
<template>
  <div>
    <ul>
      <li v-for="feature in features" :key="feature">{{feature}}</li>
    </ul>
  </div>
</template>
<script lang='ts'>
import { Component, Prop, Vue } from "vue-property-decorator";

@Component
export default class Hello extends Vue {
  features: string[] = ["类型注解", "编译型语言"];
}
</script>
```

## 类型别名

使用类型别名自定义类型

```
// 可以用下面这样方式定义对象类型
const objType: { foo: string, bar: string }

// 使用type定义类型别名, 使用更便捷, 还能复用
type Foobar = { foo: string, bar: string }
const aliasType: Foobar
```

范例: 使用类型别名定义Feature, types/index.ts

```
export type Feature = {
  id: number,
  name: string
}
```

使用自定义类型, HelloWorld.vue

```
<template>
  <div>
    <!--修改模板-->
    <li v-for="feature in features" :key="feature.id">{{feature.name}}</li>
  </div>
```

```

</template>
<script lang='ts'>
// 导入接口
import { Feature } from "@types";

@Component
export default class Hello extends Vue {
  // 修改数据结构
  features: Feature[] = [{ id: 1, name: "类型注解" }];
}
</script>

```

## 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};
// FirstAndSecond将同时拥有属性first和second
type FirstAndSecond = First & Second;

```

范例：利用交叉类型给Feature添加一个selected属性

```

// types/index.ts
type Select = {
  selected: boolean
}
export type FeatureSelect = Feature & Select

```

使用这个FeatureSelect, HelloWorld.vue

```
features: FeatureSelect[] = [
  { id: 1, name: "类型注解", selected: false },
  { id: 2, name: "编译型语言", selected: true }
];
```

```
<li :class="{selected: feature.selected}">{{feature.name}}</li>
```

```
.selected {
  background-color: rgb(168, 212, 247);
}
```

## 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');
  }
}

```

范例：新增特性，Hello.vue

```

<div>
  <input type="text" placeholder="输入新特性" @keyup.enter="addFeature">
</div>

```

```

addFeature(e: KeyboardEvent) {
  // e.target是EventTarget类型，需要断言为HTMLInputElement
  const inp = e.target as HTMLInputElement;
  const feature: FeatureSelect = {
    id: this.features.length + 1,
    name: inp.value,
    selected: false
  }
  this.features.push(feature);
  inp.value = "";
}

```

范例：生命周期钩子，Hello.vue

```

created() {
  this.features = [{ id: 1, name: "类型注解" }];
}

```

## 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

  // 参数属性：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;
  }
}
```

范例：利用getter设置计算属性，Hello.vue

```
<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>
```

## 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
```

范例：Feature也可用接口形式约束，./types/index.ts

```
// 接口中只需定义结构，不需要初始化
export interface Feature {
  id: number;
  name: string;
}
```

[Interface vs type aliases](#)

## 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。

```
// 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)
```

泛型优点：

- 函数和类可以支持多种类型，更加通用
- 不必编写多条重载，冗长联合类型，可读性好
- 灵活控制类型约束

不仅通用且能灵活控制，泛型被广泛用于通用库的编写。

范例：用axios获取数据

安装axios： `npm i axios -S`

配置一个模拟接口，vue.config.js

```
module.exports = {
  devServer: {
    before(app) {
      app.get('/api/list', (req, res) => {
        res.json([
          { id: 1, name: "类型注解", version: "2.0" },
          { id: 2, name: "编译型语言", version: "1.0" }
        ])
      })
    }
  }
}
```

使用接口，HelloWorld.vue

```
async mounted() {
  console.log("HelloWorld");
  const resp = await axios.get<FeatureSelect[]>('/api/list')
  this.features = resp.data
}
```

## 声明文件

使用ts开发时如果要使用**第三方js库**的同时还想利用ts诸如类型检查等特性就需要声明文件，类似 `xx.d.ts`

同时，vue项目中还可以在shims-vue.d.ts中对已存在模块进行补充

`npm i @types/xxx`

范例：利用模块补充\$axios属性到Vue实例，从而在组件里面直接用

```
// main.ts
import axios from 'axios'
Vue.prototype.$axios = axios;
```

```
// shims-vue.d.ts
import Vue from "vue";
import { AxiosInstance } from "axios";

declare module "vue/types/vue" {
  interface Vue {
    $axios: AxiosInstance;
  }
}
```

范例：给krouter/index.js编写声明文件，index.d.ts

```
import VueRouter from "vue-router";
declare const router: VueRouter
export default router
```

## 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
    const feature = { name: event.target.value, id: this.features.length + 1 };
    this.features.push(feature);
    event.target.value = "";
    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);
}
```

状态管理推荐使用: [vuex-module-decorators](#)

`vuex-module-decorators` 通过装饰器提供模块化声明vuex模块的方法, 可以有效利用ts的类型系统。

安装

```
npm i vuex-module-decorators -D
```

根模块清空, 修改store/index.ts

```
export default new Vuex.Store({})
```

定义counter模块, 创建store/counter

```
import { Module, VuexModule, Mutation, Action, getModule } from 'vuex-module-decorators'
import store from './index'

// 动态注册模块
@Module({ dynamic: true, store: store, name: 'counter', namespaced: true })
class CounterModule extends VuexModule {
    count = 1

    @Mutation
```

```

add() {
  // 通过this直接访问count
  this.count++
}

// 定义getters
get doubleCount() {
  return this.count * 2;
}

@Action
asyncAdd() {
  setTimeout(() => {
    // 通过this直接访问add
    this.add()
  }, 1000);
}
}

// 导出模块应该是getModule的结果
export default getModule(CounterModule)

```

使用, App.vue

```

<p @click="add">{{ $store.state.counter.count }}</p>
<p @click="asyncAdd">{{ count }}</p>

```

```

import CounterModule from '@store/counter'

@Component
export default class App extends Vue {
  get count() {
    return CounterModule.count
  }

  add() {
    CounterModule.add()
  }

  asyncAdd() {
    CounterModule.asyncAdd()
  }
}

```

## 装饰器原理

装饰器是工厂函数，它能访问和修改装饰目标。

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
    target.prototype.log = function() {
        console.log(this.bar);
    }
}

@log
class Foo {
    bar = 'bar'
}

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

方法装饰器

```
function rec(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('run method', name);
        baz.call(this, val);
    }
}

class Foo {
    @rec
    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：实现 `Component` 装饰器

```
<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中的选项都可以从目标组件定义中找到：

```
function Component(target: any): any {
  const option: any = {};
  const proto = target.prototype;
  const keys = Object.getOwnPropertyNames(proto);
  keys.forEach((key) => {
    if (key !== "constructor") {
      const descriptor = Object.getOwnPropertyDescriptor(proto, key);
      if (descriptor) {
        if (typeof descriptor.value === "function") {
          if ([ "created", "mounted" ].indexOf(key) !== -1) {
            option[key] = proto[key];
          } else {
            const methods = (!option.methods || (option.methods = {})) as any;
            methods[key] = descriptor.value;
          }
        } else if (descriptor.get || descriptor.set) {
          const computed = (!option.computed || (option.computed = {})) as any;
          computed[key] = {
            get: descriptor.get,
            set: descriptor.set,
          };
        }
      }
    }
  });

  option.data = function() {
    const vm = new target();
    const data: any = {};
    Object.keys(vm)
      .filter((key) => !key.startsWith("_") && !key.startsWith("$"))
      .forEach((key) => {
        data[key] = vm[key];
      });
    return data;
  };

  return Vue.extend(option);
}
```

# 作业

---

写一个装饰器

# 思考题

---

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

