

资源

1. [TypeScript参考](#)
2. [vue中的TypeScript](#)

知识点

准备工作

新建一个基于ts的vue项目

```
vue create vue-ts
```

选项选择：

- 自定义选项 - Manually select features
- 添加ts支持 - TypeScript
- 基于类的组件 - y
- tslint

类型注解和类型检查

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。

```
let name = "xx"; // 类型推论
let title: string = "开课吧"; // 类型注解
name = 2; // 错误
title = 4; // 错误

//数组使用类型

let names: string[];
names = ['Tom']; //或Array<string>

let foo: any = 'xx'
foo = 3

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

// 函数中使用类型注解

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

//void类型, 常用于没有返回值的函数

```
function warnUser(): void { alert("This is my warning message"); }
```

内置的类型

1. string
2. number
3. boolean
4. void 函数不返回值
5. any 任意类型

范例, Hello.vue

```
<template>
  <div>
    {{msg}}
    <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 {
  @Prop() msg!: string;
  features: string[];
  constructor() {
    super();
    this.features = ["类型注解", "编译型语言"];
  }
}
</script>
```

函数

```
// 此处name和age是必填参数
// 如果要变为可选参数, 加上?
// 可选参数在必选参数后面
function sayHello(name: string, age: number = 20, addr?: string): string {
  return '你好: ' + name + ' ' + age;
}
```

```
// 重载
// 参数数量或者类型或者返回类型不同 函数名却相同
// 先声明, 在实现
function info(a: { name: string }): string;
function info(a: string): object;
function info(a: { name: string } | string): any {
    if (typeof a === "object") {
        return a.name;
    } else {
        return { name: a };
    }
}
console.log(info({ name: "tom" }));
console.log(info("tom"));
```

范例: 新增特性, Hello.vue

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

```
// 生命周期钩子
created() {}

// 普通方法
private addFeature(event: any) {
    console.log(event);
    this.features.push(event.target.value);
    event.target.value = '';
}
```

类

ts中的类和es6中大体相同, 这里重点关注ts带来的特性

```
class MyComp {
    private _foo: string; // 私有属性, 不能在类的外部访问
    protected bar: string; // 保护属性, 还可以在派生类中访问
    readonly mua = 'mua'; // 只读属性必须在声明时或构造函数里初始化

    // 构造函数: 初始化成员变量
    // 参数加上修饰符, 能够定义并初始化一个成员属性
    constructor (private tua = 'tua') {
        this._foo = 'foo';
        this.bar = 'bar';
    }
}
```

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

// 存取器：存取数据时可添加额外逻辑；在vue里面可以用作计算属性
get foo() { return this._foo }
set foo(val) { this._foo = val }

}
```

范例：声明自定义类型约束数据结构，Hello.vue

```
// 定义一个特性类，拥有更多属性
class Feature {
  constructor(public id: number, public name: string) {}
}

// 可以对获取的数据类型做约束
@Component
export default class HelloWorld extends Vue {
  private features: Feature[];

  constructor() {
    super();
    this.features = [
      { id: 1, name: "类型注解" },
      { id: 2, name: "编译型语言" }
    ];
  }
}

// template中的变化
<li v-for="feature in features" :key="feature.id">
  {{feature.name}}
</li>
```

范例：利用getter设置计算属性

```
<li>特性数量: {{count}}</li>

get count() {
  return this.features.length;
}
```

class的原理

class是语法糖，它指向的就是构造函数。

```

class Person{ // 类指向构造函数
  constructor(name, age){ // constructor是默认方法, new实例时自动调用
    this.name = name; // 属性会声明在实例上, 因为this指向实例
    this.age = age;
  }
  say(){ // 方法会声明在原型上
    return "我的名字叫" + this.name + "今年" + this.age + "岁了";
  }
}

console.log(typeof Person); // function
console.log(Person === Person.prototype.constructor); // true

// 等效于
function Person(name,age) {
  this.name = name;
  this.age = age;
}
Person.prototype.say = function(){
  return "我的名字叫" + this.name+"今年"+this.age+"岁了";
}

```

接口

interface, 仅定义结构, 不需要实现

```

interface Person {
  firstName: string;
  lastName: string;
  sayHello(): string; // 要求实现方法
}
// 实现接口
class Greeter implements Person {
  constructor(public firstName='', public lastName=''){}
  sayHello(){
    return 'Hello, ' + this.firstName + ' ' + this.lastName;
  }
}
// 面向接口编程
function greeting(person: Person) {
  return person.sayHello();
}
// const user = {firstName: 'Jane', lastName: 'User'};
const user = new Greeter('Jane', 'User'); // 创建对象实例
console.log(user);
console.log(greeting(user));

```

范例: 修改Feature为接口形式

```
<script lang='ts'>
// 接口中只需定义结构, 不需要初始化
interface Feature {
  id: number;
  name: string;
}
</script>
```

泛型 Generics

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。

```
// 定义泛型接口
interface Result<T> {
  ok: 0 | 1;
  data: T[];
}

// 定义泛型函数
function getData<T>(): Result<T> {
  const data: any[] = [
    { id: 1, name: "类型注解", version: "2.0" },
    { id: 2, name: "编译型语言", version: "1.0" }
  ];
  return { ok: 1, data };
}

// 使用泛型
this.features = getData<Feature>().data;
```

返回Promise

```
function getData<T>(): Promise<Result<T>> {
  const data: any[] = [
    { id: 1, name: "类型注解", version: "2.0" },
    { id: 2, name: "编译型语言", version: "1.0" }
  ];
  return Promise.resolve<Result<T>>>({ ok: 1, data });
}
```

使用

```
async created() {
  const result = await getData<Feature>();
  this.features = result.data;
}
```

装饰器

装饰器实际上是工厂函数，传入一个对象，输出处理后的新对象。

```
// 类装饰器
@Component
export default class Hello extends Vue {
  // 属性装饰器
  @Prop({ required: true, type: String }) private msg!: string;

  // 函数装饰器
  @Watch("features", {deep: true})
  onFeaturesChange(val: string, oldval: any) {
    console.log(val, oldval);
  }

  // 函数装饰器
  @Emit('add')
  private addFeature(event: any) {
    const feature = {
      name: event.target.value,
      id: this.features.length + 1,
      version: "1.0"
    };
    this.features.push(feature);
    event.target.value = feature;

    return event.target.value;
  }
}
```

vuex使用: vuex-class

安装

```
npm i vuex-class -S
```

定义状态, store.js

```
export default new Vuex.Store({
  state: {
    features: ['类型检测', '预编译']
  },
  mutations: {
    addFeatureMutation(state: any, featureName: string){
      state.features.push({id: state.features.length+1, name: featureName})
    }
  },
  actions: {
    addFeatureAction({commit}, featureName: string) {
```

```

        commit('addFeatureMutation', featureName)
    }
}
})

```

使用, Hello.vue

```

import { State, Action, Mutation } from "vuex-class";

@Component
export default class Feature extends Vue {
    // 状态、动作、变更映射
    @State features!: string[];
    @Action addFeatureAction;
    @Mutation addFeatureMutation;

    private addFeature(event) {
        console.log(event);
        // this.features.push(event.target.value);
        this.addFeatureAction(event.target.value);
        // this.addFeaturMutation(event.target.value);
        event.target.value = "";
    }
}

```

装饰器原理

装饰器实际上是一个**函数**，通过定义劫持，能够对类及其方法、属性提供额外的扩展功能。

```

// 类装饰器
function isFoo(target) {
    // target是构造函数
    console.log(target === Foo); // true
    target.isFoo = true;
    return target;
}

// 属性装饰器
function mua(target, name) {
    // target是原型, name是属性名
    console.log(target === Foo.prototype);
    target[name] = 'mua~~~'
}

// 方法装饰器
function dong(target, name, descriptor) {
    // target是原型, name是方法名, descriptor是描述符,
    console.log(target === Foo.prototype);
    // 方法的定义方式: Object.defineProperty(target, name, descriptor)
    console.log(target[name] === descriptor.value);
    // 这里通过修改descriptor.value扩展了bar方法
}

```



```

    const bar = descriptor.value;
    descriptor.value = function() {
      console.log('dong~');
      bar();
    }
    return descriptor
  }
  @isFoo
  class Foo {
    @mua aaa: string;
    constructor(){
      console.log('Foo构造函数');
    }
    @dong
    bar() {
      console.log('bar~');
    }
  }

  console.log(Foo.isFoo)
  console.log(f.aaa);
  f.bar()

```

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(target) {
    const options: any = {
      data: {msg: 'mua~~mua~~'}
    };
    return Vue.extend(options);
  }

  @Component
  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