

四. Vue 3

1. TypeScript

1) 动态类型的问题

前面我们讲过 js 属于动态类型语言，例如

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

obj 可能只是个字符串

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

obj 也有可能是个函数

```
test(()=>console.log('hello, world'))
```

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，代码就崩溃了

动态类型意味着

- 运行代码时才知道发生什么 (running the code to see what happens)

静态类型意味着

- 在代码运行前，就对它的行为做出预测 (make predications about what code is expected before it runs)

下面的 typescript 代码，就在代码运行前对参数加入了约束限制

```
function test(msg : string) {  
}
```

- 限制了参数只能做 string 那些事

```
function test(msg : Function) {  
  msg()  
}
```

- 限制了参数只能做函数那些事

2) 入门

安装 typescript 编译器

```
npm install -g typescript
```

编写 ts 代码

```
function hello(msg: string) {  
  console.log(msg)  
}  
  
hello('hello,world')
```

执行 tsc 编译命令

```
tsc xxx.ts
```

编译生成 js 代码，编译后进行了类型擦除

```
function hello(msg) {  
  console.log(msg);  
}  
  
hello('hello,world');
```

再来一个例子，用 interface 定义用户类型

```
interface User {  
  name: string,  
  age: number  
}  
  
function test(u: User): void {  
  console.log(u.name)  
  console.log(u.age)  
}  
  
test({ name: 'zhangs', age: 18 })
```

编译后

```
function test(u) {  
  console.log(u.name);  
  console.log(u.age);  
}  
  
test({ name: 'zhangs', age: 18 });
```

可见，typescript 属于编译时实施类型检查（静态类型）的技术

3) 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>	

标注位置

标注变量

```
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

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

标注参数

```
function greet(name: string) {  
  
}
```

很多时候，都能够推断出参数类型

```
const names = ['Alice', 'Bob', 'Eve']  
const lowercaseNames = names.map((e: string) => e.toLowerCase())
```

- 可以用类型推断，推断出 e 是 string 类型

标注返回值

```
function add(a: number, b: number) : number {  
    return a + b  
}
```

- 一般也可以省略，因为可以根据返回值做类型推断

复杂类型

type

```
type Cat = {  
    name: string,  
    age: number  
}  
  
const c1: Cat = { name: '小白', age: 1 }  
const c2: Cat = { name: '小花' } // 错误: 缺少 age 属性  
const c3: Cat = { name: '小黑', age: 1, sex: '公' } // 错误: 多出 sex 属性
```

interface

```
interface Cat {  
    name: string,  
    age: number  
}  
  
const c1: Cat = { name: '小白', age: 1 }  
const c2: Cat = { name: '小花' } // 错误: 缺少 age 属性  
const c3: Cat = { name: '小黑', age: 1, sex: '公' } // 错误: 多出 sex 属性
```

可选属性

如果需要某个属性可选，可以用下面的语法

```
interface Cat {  
    name: string,  
    age?: number  
}  
  
const c1: Cat = { name: '小白', age: 1 }  
const c2: Cat = { name: '小花' } // 正确: age 属性可选
```

- 可选属性要注意处理 undefined 值

鸭子类型

```
interface Cat {
  name: string
}

function test(cat: Cat) {
  console.log(cat.name)
}

const c1 = { name: '小白', age: 1 }
test(c1)
```

- const c1 并没有声明类型为 Cat，但它与 Cat 类型有一样的属性，也可以被当作是 Cat 类型

方法类型

```
interface Api {
  foo(): void,
  bar(str: string): string
}

function test(api: Api) {
  api.foo()
  console.log(api.bar('hello'))
}

test({
  foo() { console.log('ok') },
  bar(str: string) { return str.toUpperCase() }
})
```

字面量类型

```
function printText(s: string, alignment: "left" | "right" | "center") {
  console.log(s, alignment)
}

printText('hello', 'left')
printText('hello', 'aaa') // 错误：取值只能是 left | right | center
```

nullish 类型

```
function test(x?: string | null) {
  console.log(x?.toUpperCase())
}

test('aaa')
test(null)
test()
```

- `x?: string | null` 表示可能是 undefined 或者是 string 或者是 null

泛型

下面的几个类型声明显然有一定的相似性

```
interface RefString {
  value: string
}

interface RefNumber {
  value: number
}

interface RefBoolean {
  value: boolean
}

const r1: RefString = { value: 'hello' }
const r2: RefNumber = { value: 123 }
const r3: RefBoolean = { value: true }
```

可以改进为

```
interface Ref<T> {
  value: T
}

const r1: Ref<string> = { value: 'hello' }
const r2: Ref<number> = { value: 123 }
const r3: Ref<boolean> = { value: true }
```

- 泛型的要点就是 `<类型参数>`，把【类型】也当作一个变化的要素，像参数一样传递过来，这样就可以派生出结构相似的新类型

函数定义也支持泛型

```
function ref<T>(n: T): Ref<T> {
  return { value: n }
}

const v1 = ref("hello");    // Ref<string>
const v2 = ref(123.3333);   // Ref<number>

v1.value.toLocaleLowerCase()
v2.value.toFixed(2)
```

4) 意义

更好理解框架

现在越来越多的前端框架采用 typescript, 如果懂 typescript 语法, 可以更好地阅读框架代码

以 Map 为例

```
const map = new Map<string, string>()
map
  .set("a", "b")
  .set("c", "d")

map.forEach((value, key, m) => {
  console.log(value, key)
})
```

- 注意编译需要 `tsc --target es6 .\xxx.ts`

更好的提示

例如, 从服务器返回的一段 json, 如果不用 typescript, 则编辑器也不能给出准确的提示

```
interface User {
  name: string,
  age: number
}

const user: User = JSON.parse(`{ "name": "张三", "age": 18 }`)
```

5) 类

关于 TypeScript 与 JavaScript 中的类语法不是重点, class 相关语法只是起到辅助作用, 更重要的是前面讲的 interface

基本语法

```
class User {  
  name: string;  
  
  constructor(name: string) {  
    this.name = name  
  }  
}  
  
const u = new User('张三')
```

其实会被编译成这个样子 (默认 --target=es3)

```
var User = /** @class */ (function () {  
  function User(name) {  
    this.name = name;  
  }  
  return User;  
}());  
var u = new User('张三');
```

所以 js 中的 class, 并不等价于 java 中的 class, 它还是基于原型实现的, 原理参考第二章 (036、037)

只读属性

```
class User {  
  readonly name: string;  
  
  constructor(name: string) {  
    this.name = name  
  }  
}  
  
const u = new User('张三')  
u.name = '李四' // 编译错误
```

- readonly 是 typescript 特有的, 表示该属性只读

方法

```
class User {  
  readonly name: string;  
  
  constructor(name: string) {  
    this.name = name  
  }  
  
  study() {  
    console.log(`[${this.name}] 正在学习`)  
  }  
}
```



```

}

const u = new User('张三')
u.study()

```

get, set

```

class User {
  _name: string;

  constructor(name: string) {
    this._name = name
  }

  get name() {
    return this._name
  }

  set name(name: string) {
    this._name = name
  }
}

const u = new User('张三')
console.log(u.name)
u.name = '李四'
console.log(u.name)

```

- 注意，需要在编译时加上 `tsc --target es6 .\xxx.ts` 选项
- es6 等价于 es2015，再此之上还有 es2016 ... es2022

类与接口

```

interface User {
  name: string
  study(course: string): void
}

class UserImpl implements User {
  name: string;
  constructor(name: string) {
    this.name = name
  }
  study(course: string) {
    console.log(`${this.name} 正在学习[${course}]`)
  }
  foo() { }
}

const user: User = new UserImpl('张三')
user.study('Typescript')
user.foo() // 错误，必须是接口中定义的方法

```

继承与接口

```
interface Flyable {
  fly(): void
}

class Animal {
  name: string;
  constructor(name: string) {
    this.name = name
  }
}

class Bird extends Animal implements Flyable {
  fly() {
    console.log(`${this.name}在飞翔`)
  }
}

const b: Flyable & Animal = new Bird("小花")
b.fly()
```

- Flyable & Animal 表示变量是 flyable 类型，同时也是 Animal 类型

方法重写

```
class Father {
  study(): void {
    console.log(`father study`)
  }
}

class Son extends Father {
  study(): void {
    super.study()
    console.log(`son study`)
  }
}

const f: Father = new Son()
f.study()
```

2. Vue3 基础

技术选型

- Vue
 - 选项式 API 还是 组合式 API ✓
 - HTML 还是 单文件组件 ✓
- 语法

- javascript 还是 typescript ✓
- 构建工具
 - @vue/cli 还是 vite ✓
- 路由
 - vue-router ✓
- 共享存储
 - vuex 还是 pinia ✓
- 视图组件
 - ElementUI 还是 Antdv ✓

1) 环境准备

创建项目

采用 vite 作为前端项目的打包，构建工具

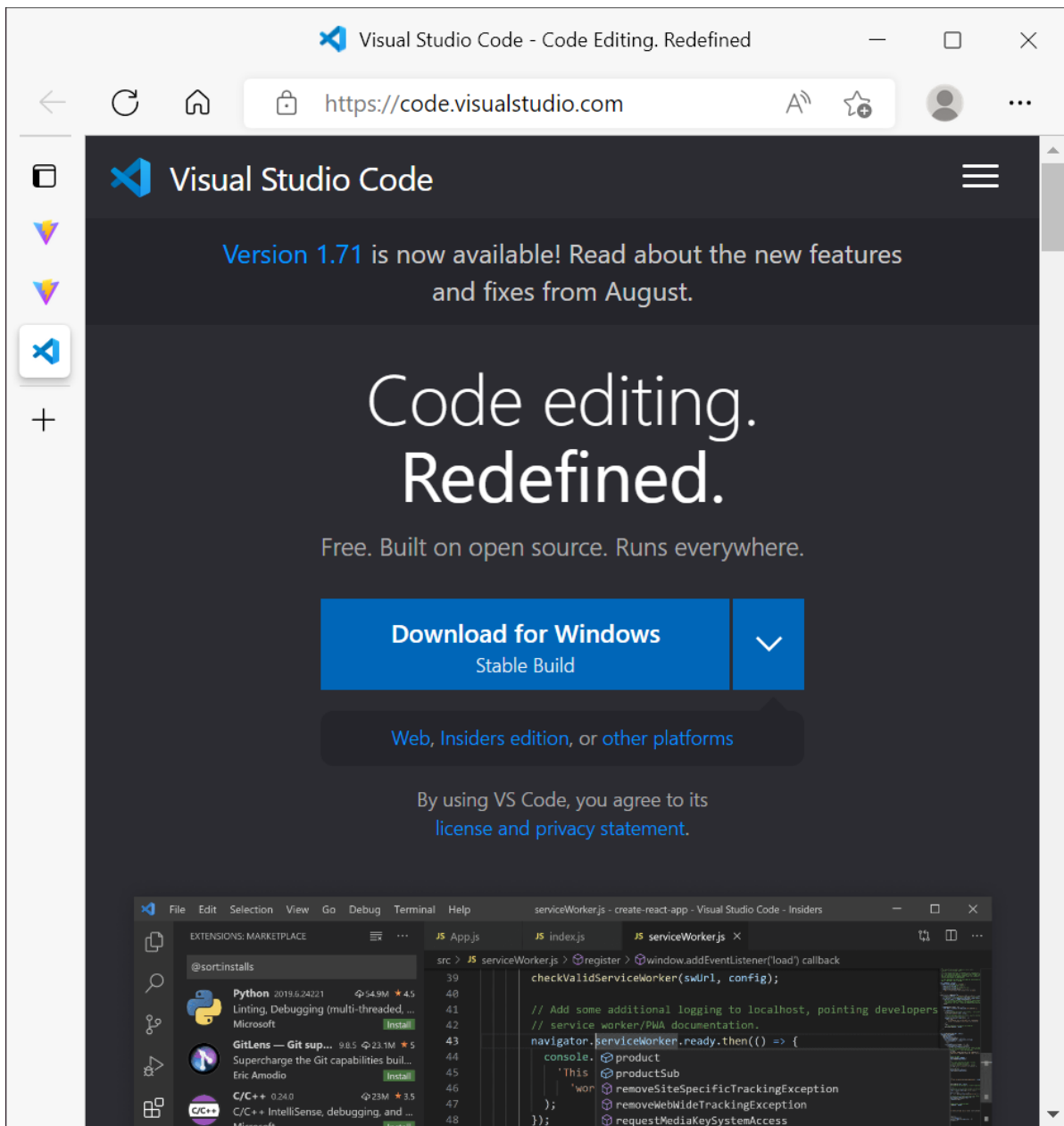
```
npm init vite@latest
```

按提示操作

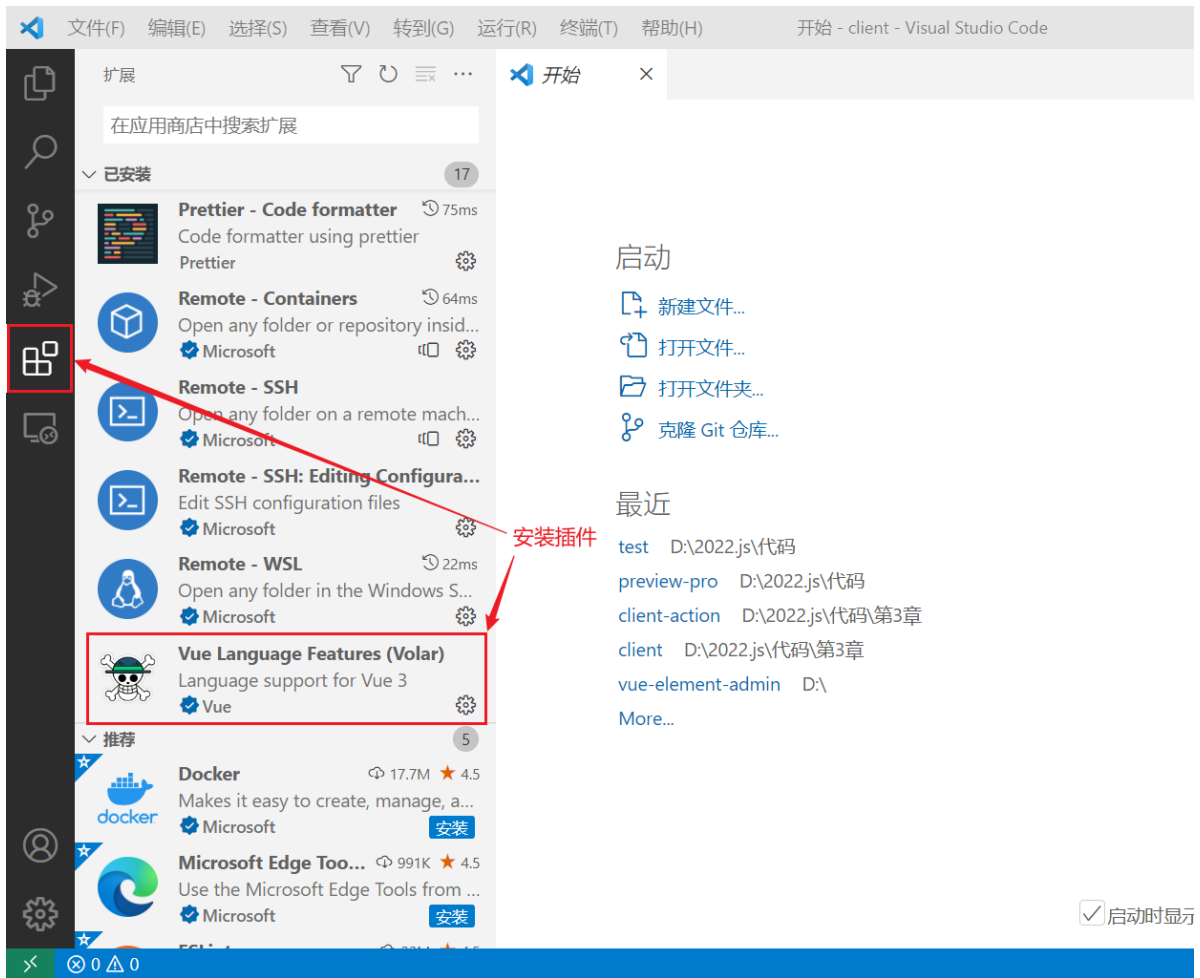
```
cd 项目目录  
npm install  
npm run dev
```

编码 IDE

推荐采用微软的 VSCode 作为开发工具，到它的官网 [Visual Studio Code - Code Editing. Redefined](https://code.visualstudio.com/) 下载安装即可

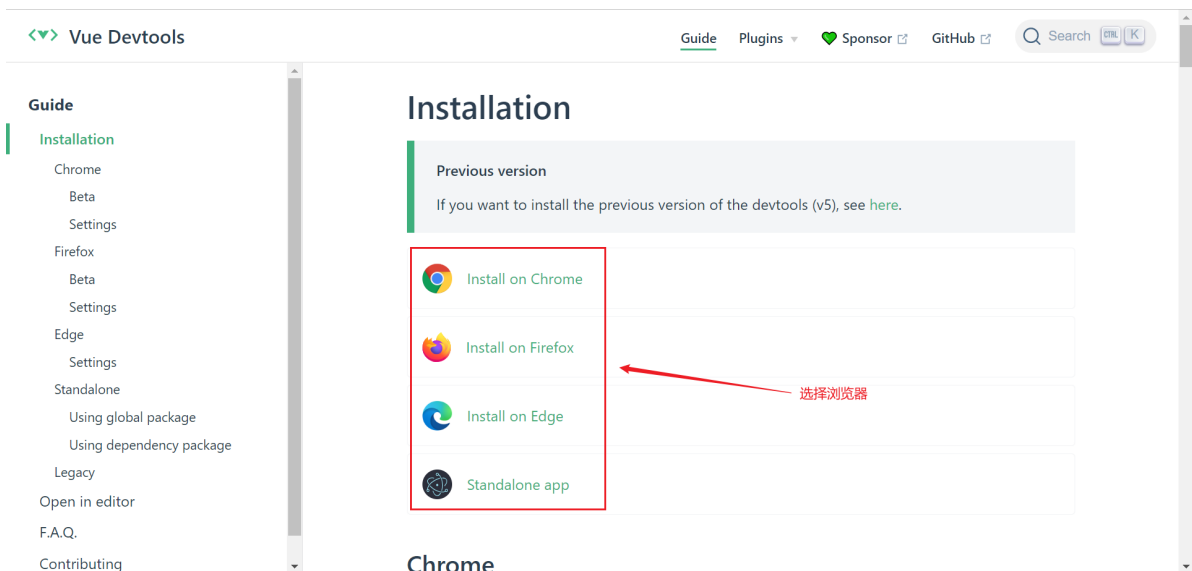


要对 *.vue 做语法支持，还要安装一个 Volar 插件



安装 devtools

- devtools 插件网址: <https://devtools.vuejs.org/guide/installation.html>



修改端口

打开项目根目录下 vite.config.ts

```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  server: {
    port: 7070
  }
})
```

- 文档地址: [配置 Vite {#configuring-vite}](#) | [Vite中文网 \(vitejs.cn\)](#)

配置代理

为了避免前后端服务器联调时, fetch、xhr 请求产生跨域问题, 需要配置代理, 同样是修改项目根目录下 vite.config.ts

```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  server: {
    port: 7070,
    proxy: {
      '/api': {
        target: 'http://localhost:8080',
        changeOrigin: true
      }
    }
  }
})
```

- 文档地址: [配置 Vite {#configuring-vite}](#) | [Vite中文网 \(vitejs.cn\)](#)

项目结构

```
index.html
package.json
tsconfig.json
vite.config.ts
├─public
└─src
  ├─assets
  ├─components
  ├─model
  ├─router
  ├─store
  └─views
```

- index.html 为主页面
- package.json npm 配置文件
- tsconfig.json typescript 配置文件
- vite.config.ts vite 配置文件
- public 静态资源
- src/components 可重用组件
- src/model 模型定义
- src/router 路由
- src/store 共享存储
- src/views 视图组件

2) Vue 组件

Vue 的组件文件以 .vue 结尾，每个组件由三部分组成

```
<script setup lang="ts"></script>

<template></template>

<style scoped></style>
```

- script 代码部分，控制模板的数据来源和行为
- template 模板部分，由它生成 html 代码
- style 样式部分，一般不咋关心

根组件是 src/App.vue，先来个 Hello,world 例子

```
<script setup lang="ts">
import { ref } from "vue";
let msg = ref("hello"); // 把数据变成响应式的

function change() {
  msg.value = "world";
  console.log(msg);
}
</script>
<template>
  <h1>{{ msg }}</h1>
  <input type="button" value="修改msg" @click="change" />
</template>
```

- {{msg}} 用来把一个变量绑定到页面上某个位置
- 绑定的变量必须用 ref 函数来封装
 - ref 返回的是【响应式】数据，即数据一旦变化，页面展示也跟着变化

main.ts

```
import { createApp } from 'vue'
import './style.css'
import App from './App.vue'

createApp(App)
  .mount('#app')
```

- createApp 是创建一个 Vue 应用程序，它接收的参数 App 即之前我们看到的根组件
- mount 就是把根组件生成的 html 代码片段【挂载】到 index.html 中 id 为 app 的 html 元素上

可以修改自己的组件文件，挂载到主页面

新建 src/views/E0.vue，内容如下

```
<script setup lang="ts">
import { ref } from 'vue'
const msg = ref('Hello, world!!')
</script>
<template>
  <h1>{{ msg }}</h1>
</template>
```

修改 main.ts 将自己的组件文件挂载

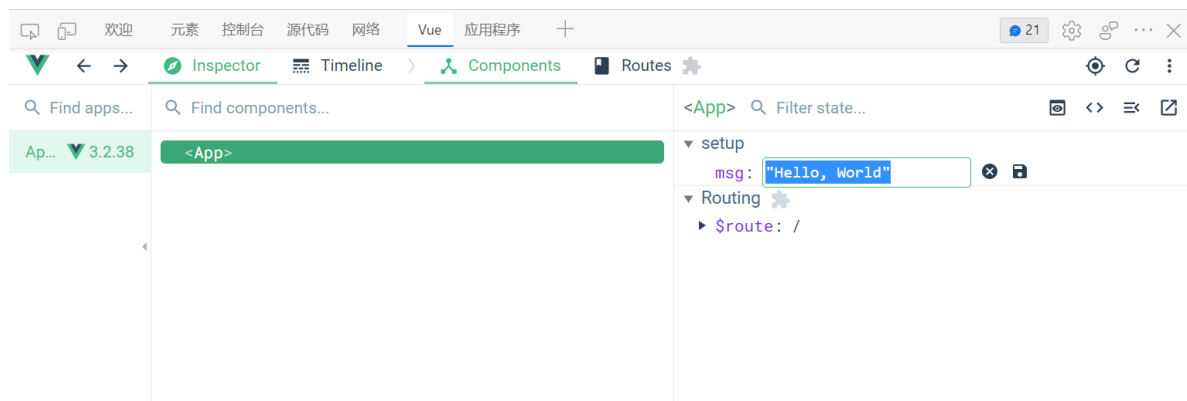
```
import { createApp } from 'vue'
import './style.css'
// import App from './App.vue'
import E0 from './views/E0.vue'

createApp(E0).mount('#app')
```

- 以后我们用这样的方式演示课堂案例

打开浏览器控制台，进入 Vue 的开发工具，尝试做如下修改

Hello, World



当把 msg 的值由 "Hello, World" 改为 "你好" 时，会发现页面展示同步发生了变化

ref 与 reactive

vue 提供了两个函数，都可以将数据变为【响应式】的

```
<script setup lang="ts">
import { ref, reactive } from 'vue'
const msg = ref('Hello, World')
const user = reactive({ name: '张三' })
</script>

<template>
  <h2>{{msg}}</h2>
  <h2>{{user.name}}</h2>
</template>
```

- ref 能将任意类型的数据变为【响应式】的
- reactive 只能将对象类型变为【响应式】，对基本类型无效（例如 string, number, boolean）

还有一点不同

```
<script setup lang="ts">
import { ref, reactive } from 'vue'
const u1 = ref({ name: '张三' })
const u2 = reactive({ name: '张三' })

function test() {
  console.log(u1.value)
  console.log(u2)
}

test()
```

```
</script>

<template>
  <h2>{{u1.name}}</h2>
  <h2>{{u2.name}}</h2>
</template>
```

- 在 template 模板中使用 ref 包装的数据，直接写【变量名】就可以了
- 但在代码中要使用 ref 包装的数据，必须用【变量名.value】才能访问到
- reactive 包装的数据，在模板中和代码中都是一致的

属性绑定

```
<script setup lang="ts">
import { ref } from 'vue'
const path = ref('/src/assets/vue.svg')

</script>

<template>
  
</template>
```

- 【:属性名】用来将标签属性与【响应式】变量绑定

事件绑定

```
<script setup lang="ts">
import { ref } from 'vue'
const count = ref(0)
function dec() {
  count.value--
}
function inc() {
  count.value++
}
</script>

<template>
  <input type="button" value="-" @click="dec">
  <h2>{{count}}</h2>
  <input type="button" value="+" @click="inc">
</template>
```

- 【@事件名】用来将标签属性与函数绑定，事件发生后执行函数内代码

表单绑定

```
<script setup lang="ts">
import { ref } from "vue";
const user = ref({
  name: '张三',
  age: 18,
  sex: '男',
  fav: ['游泳', '打球']
})

function saveUser() {
  console.log(user.value)
}
</script>

<template>
<div class="outer">
  <div>
    <label for="">请输入姓名</label>
    <input type="text" v-model="user.name"/>
  </div>
  <div>
    <label for="">请输入年龄</label>
    <input type="text" v-model="user.age"/>
  </div>
  <div>
    <label for="">请选择性别</label>
    男 <input type="radio" value="男" v-model="user.sex"/>
    女 <input type="radio" value="女" v-model="user.sex"/>
  </div>
  <div>
    <label for="">请选择爱好</label>
    游泳 <input type="checkbox" value="游泳" v-model="user.fav"/>
    打球 <input type="checkbox" value="打球" v-model="user.fav"/>
    健身 <input type="checkbox" value="健身" v-model="user.fav"/>
  </div>
  <div>
    <input type="button" value="保存" @click="saveUser">
  </div>
</div>
</template>

<style scoped>
div {
  margin-bottom: 8px;
}
.outer {
  width: 100%;
  position: relative;
  padding-left: 80px;
}
label {
  text-align: left;
  width: 100px;
  display: inline-block;
  position: absolute;
```

```
left :0;
}
</style>
```

- 用 v-model 实现双向绑定，即
 - javascript 数据可以同步到表单标签
 - 反过来用户在表单标签输入的新值也会同步到 javascript 这边
- 双向绑定只适用于表单这种带【输入】功能的标签，其它标签的数据绑定，单向就足够了
- 复选框这种标签，双向绑定的 javascript 数据类型一般用数组

计算属性

有时在数据展示时要做简单的计算

```
<script setup lang="ts">
import { ref } from 'vue'
const firstName = ref('三')
const lastName = ref('张')

</script>

<template>
  <h2>{{lastName + firstName}}</h2>
  <h3>{{lastName + firstName}}</h3>
  <h4>{{lastName + firstName}}</h4>
</template>
```

看起来较为繁琐，可以用计算属性改进

```
<script setup lang="ts">
import { ref, computed } from 'vue'
const firstName = ref('三')
const lastName = ref('张')
const fullName = computed(() => {
  console.log('enter')
  return lastName.value + firstName.value
})
</script>

<template>
  <h2>{{fullName}}</h2>
  <h3>{{fullName}}</h3>
  <h4>{{fullName}}</h4>
</template>
```

- fullName 即为计算属性，它具备缓存功能，即 firstName 和 lastName 的值发生了变化，才会重新计算
- 如果用函数实现相同功能，则没有缓存功能

```
<script setup lang="ts">
import { ref } from 'vue'
const firstName = ref('三')
const lastName = ref('张')
```

```
function fullName() {
  console.log('enter')
  return lastName.value + firstName.value
}
</script>

<template>
  <h2>{{fullName()}}</h2>
  <h3>{{fullName()}}</h3>
  <h4>{{fullName()}}</h4>
</template>
```

xhr

浏览器中有两套 API 可以和后端交互，发送请求、接收响应，fetch api 前面我们已经介绍过了，另一套 api 是 xhr，基本用法如下

```
const xhr = new XMLHttpRequest()
xhr.onload = function() {
  console.log(xhr.response)
}
xhr.open('GET', 'http://localhost:8080/api/students')
xhr.responseType = "json"
xhr.send()
```

但这套 api 虽然功能强大，但比较老，不直接支持 Promise，因此有必要对其进行改造

```
function get(url: string) {
  return new Promise((resolve, reject)=>{
    const xhr = new XMLHttpRequest()
    xhr.onload = function() {
      if(xhr.status === 200){
        resolve(xhr.response)
      } else if(xhr.status === 404) {
        reject(xhr.response)
      } // 其它情况也需考虑，这里简化处理
    }
    xhr.open('GET', url)
    xhr.responseType = 'json'
    xhr.send()
  })
}
```

- Promise 对象适合用来封装异步操作，并可以配合 await 一齐使用
- Promise 在构造时，需要一个箭头函数，箭头函数有两个参数 resolve 和 reject
 - resolve 是异步操作成功时被调用，把成功的结果传递给它，最后会作为 await 的结果返回
 - reject 在异步操作失败时被调用，把失败的结果传递给它，最后在 catch 块被捉住
- await 会一直等到 Promise 内调用了 resolve 或 reject 才会继续向下运行

调用示例1：同步接收结果，不走代理

```
try {
  const resp = await get("http://localhost:8080/api/students")
  console.log(resp)
} catch (e) {
  console.error(e)
}
```

调用示例2：走代理

```
try {
  const resp = await get('/api/students')
  console.log(resp)
} catch(e) {
  console.log(e)
}
```

- 走代理明显慢不少

axios

基本用法

axios 就是对 xhr api 的封装，手法与前面例子类似

安装

```
npm install axios
```

一个简单的例子

```
<script setup lang="ts">
import { ref, onMounted } from "vue";
import axios from "axios";

let count = ref(0);

async function getStudents() {
  try {
    const resp = await axios.get("/api/students");
    count.value = resp.data.data.length;
  } catch (e) {
    console.log(e);
  }
}

onMounted(() => {
  getStudents()
})
</script>

<template>
  <h2>学生人数为: {{ count }}</h2>
</template>
```

- onMounted 指 vue 组件生成的 html 代码片段，挂载完毕后被执行

再看一个 post 例子

```
<script setup lang="ts">
import { ref } from "vue";
import axios from "axios";

const student = ref({
  name: '',
  sex: '男',
  age: 18
})

async function addStudent() {
  console.log(student.value)
  const resp = await axios.post('/api/students', student.value)
  console.log(resp.data.data)
}
</script>

<template>
<div>
  <div>
    <input type="text" placeholder="请输入姓名" v-model="student.name"/>
  </div>
  <div>
    <label for="">请选择性别</label>
    男 <input type="radio" value="男" v-model="student.sex"/>
    女 <input type="radio" value="女" v-model="student.sex"/>
  </div>
  <div>
    <input type="number" placeholder="请输入年龄" v-model="student.age"/>
  </div>
  <div>
    <input type="button" value="添加" @click="addStudent"/>
  </div>
</div>
</template>
<style scoped>
div {
  font-size: 14px;
}
</style>
```

环境变量

- 开发环境下，联调的后端服务器地址是 `http://localhost:8080`，
- 上线改为生产环境后，后端服务器地址为 `http://itheima.com`

这就要求我们区分开发环境和生产环境，这件事交给构建工具 vite 来做

默认情况下，vite 支持上面两种环境，分别对应根目录下两个配置文件

- .env.development - 开发环境
- .env.production - 生产环境

针对以上需求，分别在两个文件中加入

```
VITE_BACKEND_API_BASE_URL = 'http://localhost:8080'
```

和

```
VITE_BACKEND_API_BASE_URL = 'http://itheima.com'
```

然后在代码中使用 vite 给我们提供的特殊对象 `import.meta.env`，就可以获取到 `VITE_BACKEND_API_BASE_URL` 在不同环境下的值

```
import.meta.env.VITE_BACKEND_API_BASE_URL
```

默认情况下，不能智能提示自定义的环境变量，做如下配置：新增文件 `src/env.d.ts` 并添加如下内容

```
/// <reference types="vite/client" />

interface ImportMetaEnv {
  readonly VITE_BACKEND_API_BASE_URL: string
  // 更多环境变量...
}

interface ImportMeta {
  readonly env: ImportMetaEnv
}
```

- 参考文档地址 [环境变量和模式 | Vite 官方中文文档 \(vitejs.dev\)](https://vitejs.dev/guide/env-and-mode)

baseUrl

可以自己创建一个 axios 对象，方便添加默认设置，新建文件 `/src/api/request.ts`

```
// 创建新的 axios 对象
import axios from 'axios'
const _axios = axios.create({
  baseUrl: import.meta.env.VITE_BACKEND_API_BASE_URL
})

export default _axios
```

然后在其它组件中引用这个 ts 文件，例如 `/src/views/E8.vue`，就不用自己拼接路径前缀了

```
<script setup lang="ts">
import axios from '../api/request'
// ...
await axios.post('/api/students', ...)
</script>
```


拦截器

```
// 创建新的 axios 对象
import axios from 'axios'
const _axios = axios.create({
  baseURL: import.meta.env.VITE_BACKEND_API_BASE_URL
})

// 请求拦截器
_axios.interceptors.request.use(
  (config)=>{ // 统一添加请求头
    config.headers = {
      Authorization: 'aaa.bbb.ccc'
    }
    return config
  },
  (error)=>{ // 请求出错时的处理
    return Promise.reject(error)
  }
)

// 响应拦截器
_axios.interceptors.response.use(
  (response)=>{ // 状态码 2xx
    // 这里的code是自定义的错误码
    if(response.data.code === 200) {
      return response
    }
    else if(response.data.code === 401) {
      // 情况1
      return Promise.resolve({})
    }
    // ...
  },
  (error)=>{ // 状态码 > 2xx, 400,401,403,404,500
    console.error(error) // 处理了异常
    if(error.response.status === 400) {
      // 情况1
    } else if(error.response.status === 401) {
      // 情况2
    }
    // ...
    return Promise.resolve({})
  }
)

export default _axios
```

处理响应时，又分成两种情况

1. 后端返回的是标准响应状态码，这时会走响应拦截器第二个箭头函数，用 `error.response.status` 做分支判断
2. 后端返回的响应状态码总是200，用自定义错误码表示出错，这时会走响应拦截器第一个箭头函数，用 `response.data.code` 做分支判断

另外

- Promise.reject(error) 类似于将异常继续向上抛出，异常由调用者（Vue组件）来配合 try ... catch 来处理
- Promise.resolve({}) 表示错误已解决，返回一个空对象，调用者中接到这个空对象时，需要配合 ?. 来避免访问不存在的属性

条件与列表

首先，新增模型数据 src/model/Model8080.ts

```
export interface Student {
  id: number;
  name: string;
  sex: string;
  age: number;
}

// 如果 spring 错误，返回的对象格式
export interface SpringError {
  timestamp: string,
  status: number,
  error: string,
  message: string,
  path: string
}

// 如果 spring 成功，返回 list 情况
export interface SpringList<T> {
  data: T[],
  message?: string,
  code: number
}

// 如果 spring 成功，返回 page 情况
export interface SpringPage<T> {
  data: { list: T[], total: number },
  message?: string,
  code: number
}

// 如果 spring 成功，返回 string 情况
export interface SpringString {
  data: string,
  message?: string,
  code: number
}

import { AxiosResponse } from 'axios'
export interface AxiosRespError extends AxiosResponse<SpringError> { }
export interface AxiosRespList<T> extends AxiosResponse<SpringList<T>> { }
export interface AxiosRespPage<T> extends AxiosResponse<SpringPage<T>> { }
export interface AxiosRespString extends AxiosResponse<SpringString> { }
```

其中

- AxiosRespPage 代表分页时的响应类型
- AxiosRespList 代表返回集合时的响应类型
- AxiosRespString 代表返回字符串时的响应类型
- AxiosRespError 代表 Spring 出错时的响应类型

```
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import axios from "../api/request";
import { Student, SpringList } from "../model/Model18080";

// 说明 students 数组类型为 Student[]
const students = ref<Student[]>([]);

async function getStudents() {
  // 说明 resp.data 类型是 SpringList<Student>
  const resp = await axios.get<SpringList<Student>>("/api/students");
  console.log(resp.data.data);
  students.value = resp.data.data;
}

onMounted(() => getStudents());
</script>
<template>
  <div class="outer">
    <div class="title">学生列表</div>
    <div class="thead">
      <div class="row bold">
        <div class="col">编号</div>
        <div class="col">姓名</div>
        <div class="col">性别</div>
        <div class="col">年龄</div>
      </div>
    </div>
    <div class="tbody">
      <div v-if="students.length === 0">暂无数据</div>
      <template v-else>
        <div class="row" v-for="s of students" :key="s.id">
          <div class="col">{{ s.id }}</div>
          <div class="col">{{ s.name }}</div>
          <div class="col">{{ s.sex }}</div>
          <div class="col">{{ s.age }}</div>
        </div>
      </template>
    </div>
  </div>
</template>
<style scoped>
.outer {
  font-family: 华文行楷;
  font-size: 20px;
  width: 500px;
}

.title {
  margin-bottom: 10px;
  font-size: 30px;
  color: #333;
}
```

```

    text-align: center;
}

.row {
  background-color: #fff;
  display: flex;
  justify-content: center;
}

.col {
  border: 1px solid #f0f0f0;
  width: 15%;
  height: 35px;
  text-align: center;
  line-height: 35px;
}

.bold .col {
  background-color: #f1f1f1;
}
</style>

```

- 加入泛型是为了更好的提示
- v-if 与 v-else 不能和 v-for 处于同一标签
- template 标签还有一个用途，就是用它少生成一层真正 html 代码
- 可以看到将结果封装为响应式数据还是比较繁琐的，后面会使用 useRequest 改进

监听器

利用监听器，可以在【响应式】的基础上添加一些副作用，把更多的东西变成【响应式的】

- 原本只是数据变化 => 页面更新
- watch 可以在数据变化时 => 其它更新

```

<template>
  <input type="text" v-model="name" />
</template>

<script setup lang="ts">
import { ref, watch } from "vue";
function useStorage(name: string) {
  const data = ref(sessionStorage.getItem(name) ?? "");
  watch(data, (newValue) => {
    sessionStorage.setItem(name, newValue);
  });
  return data;
}
const name = useStorage("name");
</script>

```

- 名称为 useXXXX 的函数，作用是返回带扩展功能的【响应式】数据
- localStorage 即使浏览器关闭，数据还在
- sessionStorage 数据工作在浏览器活动期间

vueuse

安装

```
npm install @vueuse/core
```

一些函数的用法

```
<template>
  <h3>X: {{x}}</h3>
  <h3>Y: {{y}}</h3>

  <h3>{{count}}</h3>
  <input type="button" @click="inc()" value="+">
  <input type="button" @click="dec()" value="-">

  <input type="text" v-model="name">
</template>
<script setup lang="ts">
import { useMouse, useCounter, useStorage } from '@vueuse/core'

const {x, y} = useMouse()

const {count, inc, dec} = useCounter()

const name = useStorage("name", "")
</script>
```

useRequest

响应式的 axios 封装，官网地址 [一个 Vue 请求库 | VueRequest \(atoojs.org\)](https://atoojs.org/en/packages/vue-request/)

首先安装 vue-request

```
npm install vue-request@next
```

组件

```
<template>
  <h3 v-if="students.length === 0">暂无数据</h3>
  <ul v-else>
    <li v-for="s of students" :key="s.id">
      <span>{{s.name}}</span>
      <span>{{s.sex}}</span>
      <span>{{s.age}}</span>
    </li>
  </ul>
</template>
<script setup lang="ts">
import axios from "../api/request"
import { useRequest } from 'vue-request'
import { computed } from 'vue'
import { AxiosResplist, Student } from '../model/Model18080'
```

```
// data 代表就是 axios 的响应对象
const { data } = useRequest<AxiosRespList<Student>>(() =>
  axios.get('/api/students'))

const students = computed(()=>{
  return data?.value?.data.data || []
})
</script>
<style scoped>
ul li {
  list-style: none;
  font-family: "华文行楷";
}

li span:nth-child(1) {
  font-size: 24px;
}
li span:nth-child(2) {
  font-size: 12px;
  color: crimson;
  vertical-align: bottom;
}
li span:nth-child(3) {
  font-size: 12px;
  color: darkblue;
  vertical-align: top;
}
</style>
```

- data.value 的取值一开始是 undefined，随着响应返回变成 axios 的响应对象
- 用 computed 进行适配

usePagination

在 src/model/Model8080.ts 中补充类型说明

```
export interface StudentQueryDto {
  name?: string,
  sex?: string,
  age?: string, // 18,20
  page: number,
  size: number
}
```

- js 中类似于 18,20 这样以逗号分隔字符串，会在 get 传参时转换为 java 中的整数数组

编写组件

```
<template>
  <input type="text" placeholder="请输入姓名" v-model="dto.name">
  <select v-model="dto.sex">
    <option value="" selected>请选择性别</option>
    <option value="男">男</option>
    <option value="女">女</option>
  </select>
```

```

<input type="text" placeholder="请输入年龄范围" v-model="dto.age">
<br>
<input type="text" placeholder="请输入页码" v-model="dto.page">
<input type="text" placeholder="请输入页大小" v-model="dto.size">
<input type="button" value="搜索" @click="search">
<hr>
<h3 v-if="students.length === 0">暂无数据</h3>
<ul v-else>
  <li v-for="s of students" :key="s.id">
    <span>{{s.name}}</span>
    <span>{{s.sex}}</span>
    <span>{{s.age}}</span>
  </li>
</ul>
<hr>
总记录数{{total}} 总页数{{totalPage}}
</template>
<script setup lang="ts">
import axios from "../api/request"
import { usePagination } from 'vue-request'
import { computed, ref } from 'vue'
import { AxiosRespPage, Student, StudentQueryDto } from '../model/Model18080'

const dto = ref<StudentQueryDto>({name:'', sex:'', age:'', page:1, size:5})

// data 代表就是 axios 的响应对象
// 泛型参数1: 响应类型
// 泛型参数2: 请求类型
const { data, total, totalPage, run } = usePagination<AxiosRespPage<Student>,
StudentQueryDto[]>(>
  (d) => axios.get('/api/students/q', {params: d}), // 箭头函数
  {
    defaultParams: [ dto.value ], // 默认参数, 会作为参数传递给上面的箭头函数
    pagination: {
      currentKey: 'page', // 指明当前页属性
      pageSizeKey: 'size', // 指明页大小属性
      totalKey: 'data.data.total' // 指明总记录数属性
    }
  }
) // 选项
)

const students = computed(()=>{
  return data?.value?.data.data.list || []
})

function search() {
  run(dto.value) // 会作为参数传递给usePagination的箭头函数
}
</script>
<style scoped>
ul li {
  list-style: none;
  font-family: "华文行楷";
}

li span:nth-child(1) {
  font-size: 24px;
}

```

```

li span:nth-child(2) {
  font-size: 12px;
  color: crimson;
  vertical-align: bottom;
}
li span:nth-child(3) {
  font-size: 12px;
  color: darkblue;
  vertical-align: top;
}
input,select {
  width: 100px;
}
</style>

```

- usePagination 只需要定义一次，后续还想用它内部的 axios 发请求，只需调用 run 函数

子组件

例1

定义子组件 Child1

```

<template>
  <div class="container">
    <div class="card">
      <div>
        <p class="name">{{name}}</p>
        <p class="location">{{country}}</p>
      </div>
      
    </div>
  </div>
</template>
<script setup lang="ts">
  // 定义属性， 编译宏
  defineProps<{name:string, country:string, avatar?:string}>()
</script>
<style scoped>
.container {
  width: 100%;
  display: flex;
  flex-wrap: wrap;
  justify-content: space-evenly;
  flex-direction: row-reverse;
}
.name {
  font-weight: bold;
}
.location {
  font-size: 0.8em;
  color: #6d597a;
}
.card {
  display: flex;
  justify-content: space-evenly;

```



```
padding: 1em;
margin: 1rem;
border-radius: 5px;
background: #fff;
width: 200px;
box-shadow: 0 14px 28px rgba(0, 0, 0, 0.25), 0 10px 10px rgba(0, 0, 0, 0.22);
}

.card:hover {
  transform: rotate(-5deg);
}

.card img {
  margin-left: 1em;
  border-radius: 50%;
  max-width: 55px;
  max-height: 55px;
}
</style>
```

父组件引用

```
<template>
  <Child1 name="张三" country="中国" avatar="/src/assets/vue.svg"></Child1>
  <Child1 name="李四" country="印度" avatar="/vite.svg"></Child1>
  <Child1 name="王五" country="韩国" ></Child1>
</template>
<script lang="ts" setup>
import Child1 from '../components/Child1.vue';
</script>
```

例2

首先添加类型说明 model/ModelRandomUser.ts

```
import { AxiosResponse } from "axios";
export interface AxiosRespResults extends AxiosResponse<Results>{}

export interface Results {
  info: {
    page: number,
    results: number
  },
  results: Result[]
}

export interface Result {
  gender: 'male' | 'female',
  name: {
    first: string,
    last: string
  },
  location: {
    country: string
  },
}
```

```

    picture: {
      medium: string
    },
    login: {
      username: string
    }
  }
}

```

子组件不变，父组件使用子组件

```

<!-- 父组件 -->
<template>
  <Child1 v-for="u of users"
    :name="u.name.first"
    :country="u.location.country"
    :avatar="u.picture.medium"
    :key="u.login.username"></Child1>
</template>
<script setup lang="ts">
import axios from "axios";
import { useRequest } from "vue-request";
import { computed } from "vue";
import { AxiosRespResults } from '../model/ModelRandomUser'
import Child1 from "../components/Child1.vue";

const { data } = useRequest<AxiosRespResults>(
  ()=>axios.get('https://randomuser.me/api/?results=3')
)

const users = computed(()=>{
  return data.value?.data.results || []
})
</script>

```

如果觉得 Result 数据结构嵌套太复杂，还可以做一个类型映射

```

<!-- 父组件 -->
<template>
  <Child1 v-for="u of users"
    :name="u.name"
    :country="u.country"
    :avatar="u.avatar"
    :key="u.username"></Child1>
</template>
<script setup lang="ts">
import axios from "axios";
import { useRequest } from "vue-request";
import { computed } from "vue";
import { AxiosRespResults, Result } from '../model/ModelRandomUser'
import Child1 from "../components/Child1.vue";

const { data } = useRequest<AxiosRespResults>(
  ()=>axios.get('https://randomuser.me/api/?results=3')
)

const users = computed(()=>{

```

```

    return data.value?.data.results.map(resultToUser) || []
  })

interface User {
  name: string,
  country: string,
  avatar: string,
  username: string
}

function resultToUser(r:Result):User {
  return {
    name: r.name.first,
    country: r.location.country,
    avatar: r.picture.medium,
    username: r.login.username
  }
}
</script>

```

- resultToUser 将 Result 类型映射为 User 类型

3. Vue 进阶

1) Antdv

添加必要插件

```
npm install ant-design-vue
```

- ant-design-vue 组件库插件

引入 antdv 功能, 修改 main.ts

```

import { createApp } from 'vue'
import './style.css'
import App from './App.vue'
import antd from 'ant-design-vue'
import 'ant-design-vue/dist/antd.css'

createApp(App).use(antd).mount('#app')

```

表格

```

<template>
  <!-- <a-table :columns="columns" :dataSource="students" rowKey="id"></a-table>
-->
  <a-table :columns="columns" :dataSource="students" :rowKey="rowKey"></a-table>
</template>
<script setup lang="ts">
import axios from "../api/request";
import { ref, computed } from "vue";
import { useRequest } from "vue-request";

```

```

import { AxiosResplist, Student } from "../model/Model18080";

const {data} = useRequest<AxiOSResplist<Student>>(
  ()=>axios.get('/api/students')
)

const students = computed(()=>{
  return data.value?.data.data || []
})

function rowKey(r:Student) {
  return r.id
}

const columns = ref([
  {
    title:'编号',
    dataIndex:'id'
  },
  {
    title:'姓名',
    dataIndex:'name'
  },
  {
    title:'性别',
    dataIndex:'sex'
  },
  {
    title:'年龄',
    dataIndex:'age'
  }
])
</script>

```

分页

```

<template>
  <a-table :columns="columns":data-source="students" row-key="id"
    :pagination="pagination" @change="tableChange"></a-table>
</template>
<script setup lang="ts">
import axios from "../api/request";
import { ref, computed } from "vue";
import { usePagination } from "vue-request";
import { AxiosRespPage, Student, StudentQueryDto } from "../model/Model18080";
import { PaginationProps } from "ant-design-vue";
import DateBody from "ant-design-vue/lib/vc-picker/panels/DatePanel/DateBody";

const dto = ref({page: 1, size: 5})

const {data, total, run} = usePagination<AxiOSRespPage<Student>,
StudentQueryDto[]>(
  (d)=> axios.get('/api/students/q', {params:d}),
  {
    defaultParams: [dto.value],

```

```

    pagination: {
      currentKey: "page",
      pageSizeKey: 'size',
      totalKey: 'data.data.total'
    }
  }
)

// 在页号或页大小改变时调用
function tableChange(pagination: PaginationProps) {
  console.log(pagination)
  dto.value.page = pagination.current ?? 1
  dto.value.size = pagination.pageSize ?? 5
  run(dto.value)
}

const pagination = computed<PaginationProps>(()=>{
  return {
    current: dto.value.page, // 当前页
    pageSize: dto.value.size, // 页大小
    total: total.value,      // 总记录数
    showSizeChanger: true,   // 显示页大小的下拉列表
    pageSizeOptions: ["1", "2", "3", "4", "5"] // 自定义下拉列表内容
  }
})

const students = computed(()=>{
  return data.value?.data.data.list || []
})

const columns = ref([
  {
    title: "编号",
    dataIndex: "id",
  },
  {
    title: "姓名",
    dataIndex: "name",
  },
  {
    title: "性别",
    dataIndex: "sex",
  },
  {
    title: "年龄",
    dataIndex: "age",
  },
]);
</script>

```

搜索、删除

```
<template>
  <a-row>
    <a-col :span="2">
      <a-button type="primary" size="small">新增</a-button>
    </a-col>
    <a-col :span="4">
      <a-popconfirm title="确认要删除选中学生吗?"
        ok-text="确定" cancel-text="取消" @confirm="onDeleteIds"
        @visibleChange="onVisibleChange" :visible="visible">
        <a-button type="primary" size="small">删除选中</a-button>
      </a-popconfirm>
    </a-col>
    <a-col :span="4">
    </a-col>
    <a-col :span="4">
      <a-input v-model:value="dto.name" placeholder="输姓名" size="small"></a-
input>
    </a-col>
    <a-col :span="4">
      <a-select v-model:value="dto.sex" placeholder="选性别" :allowClear="true"
size="small">
        <a-select-option value="男">男</a-select-option>
        <a-select-option value="女">女</a-select-option>
      </a-select>
    </a-col>
    <a-col :span="4">
      <a-select v-model:value="dto.age" placeholder="选年龄" :allowClear="true"
size="small">
        <a-select-option value="0,20">20以下</a-select-option>
        <a-select-option value="21,30">21~30</a-select-option>
        <a-select-option value="31,40">31~40</a-select-option>
        <a-select-option value="40,120">40以上</a-select-option>
      </a-select>
    </a-col>
    <a-col :span="2">
      <a-button @click="tableChange" type="primary" size="small">搜索</a-button>
    </a-col>
  </a-row>
  <hr>
  <a-table :columns="columns" :data-source="students" row-key="id"
:pagination="pagination" @change="tableChange"
:row-selection="{selectedRowKeys:ids,onChange:onSelectChange}">
    <template #bodyCell="{column, record}">
      <template v-if="column.dataIndex==='name'">
        {{record.name + (record.sex==='男'?'(大侠)':'(女侠')}}
      </template>

      <template v-else-if="column.dataIndex==='operation'">
        <a>修改</a>
        <a-divider type="vertical"></a-divider>
        <a-popconfirm title="确认要删除该学生吗?"
          ok-text="确定" cancel-text="取消" @confirm="onDelete(record.id)">
          <a>删除</a>
        </a-popconfirm>
      </template>
    </template>
```

[illegible]

[illegible]


```

        title: '操作',
        dataIndex: 'operation'
      }
    ]);
  </script>
  <style scoped>
    .ant-input, .ant-select {
      width: 80px;
    }
  </style>

```

新增、修改

子组件

```

<template>
  <a-modal :visible="visible" :title="title"
    @ok="onOk" @cancel="onCancel">
    <a-form>
      <a-form-item label="编号" v-if="id">
        <a-input readonly v-model:value="id"></a-input>
      </a-form-item>
      <a-form-item label="姓名">
        <a-input v-model:value="dto.name"></a-input>
      </a-form-item>
      <a-form-item label="性别">
        <a-radio-group v-model:value="dto.sex">
          <a-radio-button value="男">男</a-radio-button>
          <a-radio-button value="女">女</a-radio-button>
        </a-radio-group>
      </a-form-item>
      <a-form-item label="年龄">
        <a-input-number v-model:value="dto.age"></a-input-number>
      </a-form-item>
    </a-form>
  </a-modal>
</template>
<script setup lang="ts">
import axios from "../api/request";
import { ref, computed } from "vue";
import { useRequest } from "vue-request";
import { StudentSaveDto, AxiosRespString } from "../model/Model8080";
import { Form } from 'ant-design-vue'

// 定义属性
const props = defineProps<{id:number, dto:StudentSaveDto, visible:boolean}>()

const title = computed(()=> props.id===0?'新增学生':'修改学生')

// 定义事件
const emit = defineEmits(['update:visible', 'saved'])

async function onOk() {
  if(props.id === 0) {
    await insert(props.dto)
  }
}

```

```

    } else {
      await update(props.dto)
    }
    emit('saved')
    // 发送事件给父组件，希望把 visible 改为 false
    emit('update:visible', false)
  }

function onCancel() {
  // 发送事件给父组件，希望把 visible 改为 false
  emit('update:visible', false)
}

const {runAsync:insert} = useRequest<AxiosRespString,StudentSaveDto[]>(<
  (dto)=>axios.post('/api/students', dto),
  {
    manual: true
  }
)

const {runAsync:update} = useRequest<AxiosRespString,StudentSaveDto[]>(<
  (dto)=>axios.put(`/api/students/${props.id}`, dto),
  {
    manual: true
  }
)
</script>

```

父组件使用子组件

[illegible]

- saveDto 使用 reactive 包装，是为了解决后续表单校验失效问题
- Object.assign 是将源对象（参数2）的属性值赋值给目标对象（参数1）的同名属性，效果等价于

```
saveDto.name = record.name
saveDto.sex = record.sex
saveDto.age = record.age
```

全局消息

在 request.ts 中对响应消息统一处理

```
import { message } from 'ant-design-vue'

// ...
// 响应拦截器
axios.interceptors.response.use(
  (response) => { // 状态码 2xx
    if (response.data.message) {
      message.success(response.data.message, 3)
    }
    // ...
  },
  (error) => { // 状态码 > 2xx, 400, 401, 403, 404, 500
    // ...
  }
)
```

表单校验

```
<template>
  <a-modal :visible="visible" :title="title"
    @ok="onOk" @cancel="onCancel">
    <a-form>
      <a-form-item label="编号" v-if="id">
        <a-input readonly v-model:value="id"></a-input>
      </a-form-item>
      <a-form-item label="姓名" v-bind="validateInfos.name">
        <a-input v-model:value="dto.name"></a-input>
      </a-form-item>
      <a-form-item label="性别" v-bind="validateInfos.sex">
        <a-radio-group v-model:value="dto.sex">
          <a-radio-button value="男">男</a-radio-button>
          <a-radio-button value="女">女</a-radio-button>
        </a-radio-group>
      </a-form-item>
      <a-form-item label="年龄" v-bind="validateInfos.age">
        <a-input-number v-model:value="dto.age"></a-input-number>
      </a-form-item>
    </a-form>
  </a-modal>
</template>
<script setup lang="ts">
```

```

import axios from "../api/request";
import { ref, computed } from "vue";
import { useRequest } from "vue-request";
import { StudentSaveDto, AxiosRespString } from "../model/Model8080";
import { Form } from 'ant-design-vue'

// 定义属性
const props = defineProps<{id:number, dto:StudentSaveDto, visible:boolean}>()

const title = computed(()=> props.id===0?'新增学生':'修改学生')

// 定义事件
const emit = defineEmits(['update:visible', 'saved'])

async function onOk() {
  try {
    // 提交前校验
    await validate()
    if(props.id === 0) {
      await insert(props.dto)
    } else {
      await update(props.dto)
    }
    emit('saved')
    // 发送事件给父组件，希望把 visible 改为 false
    emit('update:visible', false)
  } catch (e) {
    console.error(e)
  }
}

function onCancel() {
  // 发送事件给父组件，希望把 visible 改为 false
  emit('update:visible', false)
}

const {runAsync:insert} = useRequest<AxiosRespString,StudentSaveDto[]>(<
  (dto)=>axios.post('/api/students', dto),
  {
    manual: true
  }
)

const {runAsync:update} = useRequest<AxiosRespString,StudentSaveDto[]>(<
  (dto)=>axios.put(`/api/students/${props.id}`, dto),
  {
    manual: true
  }
)

const rules = ref([
  name: [
    {required: true, message:'姓名必须'},
    {min:2, message:'字符数至少为2'}
  ],
  sex: [
    {required: true, message:'性别必须'}
  ],
],

```

```

age: [
  {required: true, message: '年龄必须'},
  {min: 10, message: '年龄最小为10岁', type: 'number'},
  {max: 120, message: '年龄最大为120岁', type: 'number'}
]
})

// 参数1: 待校验的数据
// 参数2: 校验规则
const { validateInfos, validate } = Form.useForm(props.dto, rules)
</script>

```

2) vue-router

安装

```
npm install vue-router@4
```

创建 router

首先创建一个 /src/router/a5router.ts 文件，在其中定义路由

```

import {createRouter, createWebHashHistory} from 'vue-router'
import A51 from '../views/A51.vue'
import A52 from '../views/A52.vue'
// 路由 => 路径和组件之间的对应关系
const routes = [
  {
    path: '/a1',
    component: A51
  },
  {
    path: '/a2',
    component: A52
  }
]

const router = createRouter({
  history: createWebHashHistory(), // 路径格式
  routes: routes // 路由数组
})

export default router

```

- createWebHashHistory 是用 # 符号作为【单页面】跳转技术，上面两个路由访问时路径格式为
 - <http://localhost:7070/#/a1>
 - <http://localhost:7070/#/a2>
- 每个路由都有两个必须属性
 - path: 路径
 - component: 组件
- createRouter 用来创建 router 对象，作为默认导出

需要在 main.ts 中导入 router 对象：

```
// ...
import A5 from './views/A5.vue' // vue-router
import router from './router/a5router'
createApp(A5).use(antdv).use(router).mount('#app')
```

A5 是根组件，不必在 router 中定义，但需要在其中定义 router-view，用来控制路由跳转后，A51、A52 这些组件的显示位置，内容如下

```
<template>
  <div class="a5">
    <router-view></router-view>
  </div>
</template>
```

效果如下



动态导入

```
import {createRouter, createWebHashHistory} from 'vue-router'
import A51 from './views/A51.vue'
import A52 from './views/A52.vue'
const routes = [
  // ...
  {
    path: '/a3',
    component: () => import('./views/A53.vue')
  }
]
```

- 用 import 关键字导入，效果是打包时会将组件的 js 代码都打包成一个大的 js 文件，如果组件非常多，会影响页面加载速度
- 而 import 函数导入（动态导入），则是按需加载，即

- 当路由跳转到 /a3 路径时，才会去加载 A53 组件对应的 js 代码
- vue-router 官方推荐采用动态导入

嵌套路由

如果希望再嵌套更深层次的路由跳转，例如：希望在 A53 组件内再进行路由跳转



首先，修改 A53.vue

```
<template>
  <div class="a53">
    <router-view></router-view>
  </div>
</template>
```

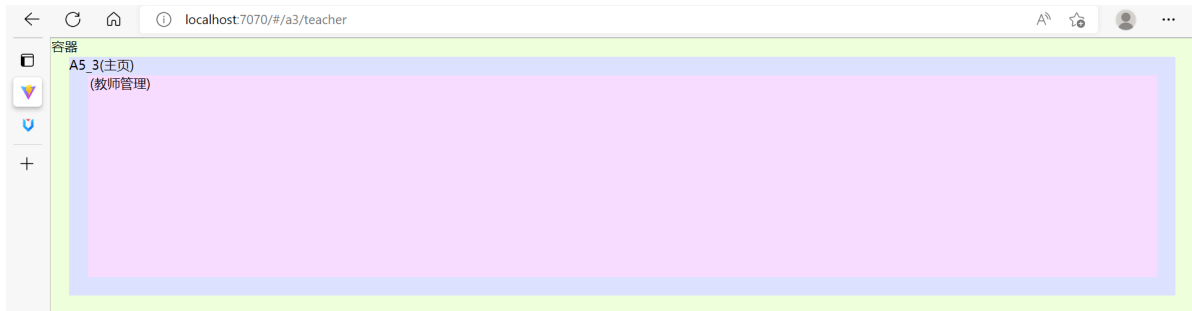
其次，再修改 /src/router/a5router.ts 文件内容

```
import {createRouter, createWebHashHistory} from 'vue-router'
import A51 from '../views/A51.vue'
import A52 from '../views/A52.vue'
const routes = [
  // ...
  {
    path: '/a3',
    component: () => import('../views/A53.vue'),
    children: [
      {
        path: 'student',
        component: () => import('../views/A531.vue')
      },
      {
        path: 'teacher',
        component: () => import('../views/A532.vue')
      }
    ]
  }
  // ...
]
```

将来访问 /a3/student 时，效果为



访问 /a3/teacher 时, 效果为



重定向

用法1

```
import {createRouter, createWebHashHistory} from 'vue-router'
import A51 from '../views/A51.vue'
import A52 from '../views/A52.vue'
const routes = [
  // ...
  {
    path: '/a3',
    component: () => import('../views/A53.vue'),
    redirect: '/a3/student', // 重定向到另外路径
    children: [
      {
        path: 'student',
        component: () => import('../views/A531.vue')
      },
      {
        path: 'teacher',
        component: () => import('../views/A532.vue')
      }
    ]
  }
]
// ...
```

效果是, 页面输入 /a3, 紧接着会重定向跳转到 /a3/student

用法2

```
import {createRouter, createWebHashHistory} from 'vue-router'
import A51 from '../views/A51.vue'
```



```
import A52 from '../views/A52.vue'
const routes = [
  {
    path: '/a1',
    component: A51
  },
  {
    path: '/a2',
    component: A52
  },
  // ...
  {
    path: '/*:pathMatcher(.*)*', // 可以匹配剩余的路径
    redirect: '/a2'
  }
]
// ...
```

效果是，当页面输入一个不存在路径 /aaa 时，会被 `path: '/*:pathMatcher(.*)*'` 匹配到，然后重定向跳转到 A52 组件去

主页布局

借助 antdv 的 layout 组件，可以实现主页【上】【左】【右】布局

```
<template>
  <div class="a53">
    <a-layout>
      <a-layout-header></a-layout-header>
      <a-layout>
        <a-layout-sider></a-layout-sider>
        <a-layout-content>
          <router-view></router-view>
        </a-layout-content>
      </a-layout>
    </a-layout>
  </div>
</template>
<style scoped>
.a53 {
  height: 100%;
  background-color: rgb(220, 225, 255);
  background-image: url("data:image/svg+xml,%3Csvg
xmlns='http://www.w3.org/2000/svg'%3E%3Ctext x='35' y='10' font-size='14' font-
family='system-ui, sans-serif' text-anchor='middle' dominant-
baseline='middle'%3EA53(主页)%3C/text%3E%3C/svg%3E");
  padding: 20px;
  box-sizing: border-box;
}
.ant-layout-header {
  height: 50px;
  background-color: darkseagreen;
}

.ant-layout-sider {
```

```

background-color: lightsalmon;
}

.ant-layout-content {
background-color: aliceblue;
}

.ant-layout-footer {
background-color: darkslateblue;
height: 30px;
}

.ant-layout {
height: 100%;
}

.ant-layout-has-sider {
height: calc(100% - 50px);
}
</style>

```

侧边栏菜单

```

<template>
  <div class="a53">
    <a-layout>
      <a-layout-header></a-layout-header>
      <a-layout>
        <a-layout-sider>
          <a-menu theme="dark" mode="inline">
            <a-menu-item :key="1">
              <router-link to="/a3/student">菜单1</router-link>
            </a-menu-item>
            <a-menu-item :key="2">
              <router-link to="/a3/teacher">菜单2</router-link>
            </a-menu-item>
            <a-menu-item :key="3">菜单3</a-menu-item>
            <a-sub-menu :key="4" title="菜单4">
              <a-menu-item :key="41">菜单41</a-menu-item>
              <a-menu-item :key="42">菜单42</a-menu-item>
            </a-sub-menu>
          </a-menu>
        </a-layout-sider>
        <a-layout-content>
          <router-view></router-view>
        </a-layout-content>
      </a-layout>
    </div>
  </template>

```

- a-menu-item 与 a-sub-menu 都必须为 key 属性唯一赋值，否则会产生混乱
- router-link 标签用来切换路由，to 是目标路由的路径
- theme 属性定义菜单的主题（默认亮色主题，dark 为暗色主题）

- mode 属性定义子菜单的展示模式（默认弹出，inline 显示在下方）

菜单图标

安装图标依赖

```
npm install @ant-design/icons-vue
```

菜单中使用图标

```
<template>
  <div class="a53">
    <a-layout>
      <a-layout-header></a-layout-header>
      <a-layout>
        <a-layout-sider>
          <a-menu theme="dark" mode="inline">
            <a-menu-item :key="1">
              <template #icon>
                <highlight-outlined />
              </template>
              <router-link to="/a3/student">菜单1</router-link>
            </a-menu-item>
            <a-menu-item :key="2">
              <template #icon>
                <align-center-outlined />
              </template>
              <router-link to="/a3/teacher">菜单2</router-link>
            </a-menu-item>
            <a-menu-item :key="3">
              <template #icon>
                <strikethrough-outlined />
              </template>
              菜单3</a-menu-item>
            <a-sub-menu :key="4" title="菜单4">
              <template #icon>
                <sort-descending-outlined />
              </template>
              <a-menu-item :key="41">菜单41</a-menu-item>
              <a-menu-item :key="42">菜单42</a-menu-item>
            </a-sub-menu>
          </a-menu>
        </a-layout-sider>
        <a-layout-content>
          <router-view></router-view>
        </a-layout-content>
      </a-layout>
    </div>
  </template>
<script setup lang="ts">
import {HighlightOutlined, AlignCenterOutlined, StrikethroughOutlined,
SortDescendingOutlined} from '@ant-design/icons-vue'
</script>
```

- 图标组件没有全局绑定，需要 import 之后才能使用
- 用 `<template #icon></template>` 插槽，才能确定图标展示的位置（菜单文字之前）

二次封装图标组件

最终希望用统一的图标组件去使用图标，图标名只是作为一个属性值传递进去，例如：

使用者

```
<template>
  <a-icon icon="highlight-outlined"></a-icon>
  <a-icon icon="align-center-outlined"></a-icon>
  <a-icon icon="striketrough-outlined"></a-icon>
  <a-icon icon="sort-descending-outlined"></a-icon>
</template>
<script setup lang="ts">
import AIcon from '../components/AIcon1.vue'
</script>
```

方法1，使用 vue 组件

```
<script lang="ts" setup>
import {HighlightOutlined, AlignCenterOutlined, StrikethroughOutlined,
SortDescendingOutlined} from '@ant-design/icons-vue'
const props = defineProps<{icon:string}>()
</script>
<template>
  <highlight-outlined v-if="icon==='highlight-outlined'"></highlight-outlined>
  <align-center-outlined v-else-if="icon==='align-center-outlined'"></align-
center-outlined>
  <striketrough-outlined v-else-if="icon==='striketrough-outlined'">
</striketrough-outlined>
  <sort-descending-outlined v-else-if="icon==='sort-descending-outlined'">
</sort-descending-outlined>
</template>
```

- 缺点：实现太笨

方法2，使用函数式组件

```
import { h } from "vue"
import * as Icons from '@ant-design/icons-vue'

interface Module {
  [p:string]: any
}

// 参数1: 组件属性
const AIcon = (props:{icon:string}) => {
  // console.log(props.icon)
  // console.log(Icons)
  // 参数1: 组件对象
  const im: Module = Icons
  return h(im[toCamelCase(props.icon)])
}
```

```

export default AIcon

// 将-分隔的单词转换为大驼峰命名的单词
function toCamelCase(str: string) { // highlight-outlined
  return str.split('-') // ['highlight', 'outlined']
    .map((e) => e.charAt(0).toUpperCase() + e.slice(1) ) // ['Highlight',
'outlined']
    .join('')
}
/*
Icons 的结构如下
{
  HighlightOutlined: HighlightOutlined组件对象,
  MonitorOutlined: MonitorOutlined组件对象,
  ...
}
*/

```

- 需要动态生成标签的时候，可以考虑使用函数式组件

方法3, 使用 jsx 组件

首先, 安装

```
npm install @vitejs/plugin-vue-jsx -D
```

配置 vite.config.ts

```

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueJsx from '@vitejs/plugin-vue-jsx'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue(), vueJsx()]
})

```

编写一个 Hi.tsx 组件

```

export default {
  props: {
    msg: String
  },
  setup(props: { msg: string }) {
    return () => <h5>{props.msg}</h5>
  }
}

```

然后被其它组件使用

```

<script setup lang="ts">
import Hi from '../components/Hi'
</script>

<template>
  <Hi msg="Hello,world"></Hi>
</template>

```

用 jsx 实现图标组件

```

import * as Icons from '@ant-design/icons-vue'

interface Module {
  [p:string]: any
}

function toCamelCase(str: string) { // highlight-outlined
  return str
    .split("-") // ['highlight', 'outlined']
    .map((e) => e.charAt(0).toUpperCase() + e.slice(1)) // ['Highlight',
'Outlined']
    .join(""); // HighlightOutlined
}

export default {
  props: {
    icon: String
  },
  setup(props: {icon: string}) {
    const im: Module = Icons
    const tag = im[toCamelCase(props.icon)] // 图标组件
    // HighlightOutlined
    return () => <tag></tag> // 返回组件标签
  }
}

```

动态路由与菜单

路由文件

a6router.js

```

import { createRouter, createWebHashHistory } from 'vue-router'
import { useStorage } from '@vueuse/core'
import { Route, Menu } from '../model/Model8080'
const clientRoutes = [
  {
    path: '/login',
    name: 'login',
    component: () => import('../views/A6Login.vue')
  },
  {
    path: '/404',
    name: '404',

```

```

    component: () => import('../views/A6NotFound.vue')
  },
  {
    path: '/',
    name: 'main',
    component: () => import('../views/A6Main.vue')
  },
  {
    path: '/*:pathMatcher(.*)*',
    name: 'remaining',
    redirect: '/404'
  }
]

const router = createRouter({
  history: createWebHashHistory(),
  routes: clientRoutes
})

export const serverMenus = useStorage<Menu[]>('serverMenus', [])
const serverRoutes = useStorage<Route[]>('serverRoutes', [])
addServerRoutes(serverRoutes.value)

export function addServerRoutes(routeList: Route[]) {
  for (const r of routeList) {
    if (r.parentName) {
      router.addRoute(r.parentName, {
        path: r.path,
        component: () => import(r.component),
        name: r.name
      })
    }
  }
  serverRoutes.value = routeList
}

export function resetRoutes() {
  for (const r of clientRoutes) {
    router.addRoute(r)
  }
  serverRoutes.value = null
  serverMenus.value = null
}

export default router

```

本文件重要的函数及变量

- addServerRoutes 函数向路由表中添加由服务器提供的路由，路由分成两部分
 - clientRoutes 这是客户端固定的路由
 - serverRoutes 这是服务器变化的路由，存储于 localStorage
- resetRoutes 函数用来将路由重置为 clientRoutes
 - vue-router@4 中的 addRoute 方法会【覆盖】同名路由，这是这种实现的关键
 - 因此，服务器返回的路由最好是 main 的子路由，这样重置时就会比较简单，用之前的 main 一覆盖就完事了
- serverMenus 变量记录服务器变化的菜单，存储于 localStorage

登录组件

动态路由应当在登录时生成, A6Login.vue

```
<template>
  <div class="login">
    <a-form :label-col="{ span: 6 }" autocomplete="off">
      <a-form-item label="用户名" v-bind="validateInfos.username">
        <a-input v-model:value="dto.username" />
      </a-form-item>
      <a-form-item label="密码" v-bind="validateInfos.password">
        <a-input-password v-model:value="dto.password" />
      </a-form-item>
      <a-form-item :wrapper-col="{ offset: 6, span: 16 }">
        <a-button type="primary" @click="onClick">Submit</a-button>
      </a-form-item>
    </a-form>
  </div>
</template>
<script setup lang="ts">
import { ref, onMounted } from 'vue'
import { Form } from 'ant-design-vue'
import { useRouter } from 'vue-router'
import axios from '../api/request'
import { useRequest } from 'vue-request'
import { AxiosRespToken, LoginDto, AxiosRespMenuAndRoute } from
'../model/Model18080'
import { resetRoutes, addServerRoutes, serverMenus } from '../router/a6router'
const dto = ref({username: '', password: ''})
const rules = ref({
  username: [
    {required: true, message: '用户名必填'}
  ],
  password: [
    {required: true, message: '密码必填'}
  ]
})
const { validateInfos, validate } = Form.useForm(dto, rules)
const router = useRouter()
const { runAsync: login } = useRequest<AxiosRespToken, LoginDto[]>((dto)=>
  axios.post('/api/loginJwt', dto), {manual: true})
const { runAsync: menu } = useRequest<AxiosRespMenuAndRoute, string[]>
((username)=> axios.get(`/api/menu/${username}`), {manual: true})
async function onClick() {
  try {
    await validate()
    const loginResp = await login(dto.value)
    if(loginResp.data.code === 200) { // 登录成功
      const token = loginResp.data.data.token
      const menuResp = await menu(dto.value.username)
      const routeList = menuResp.data.data.routeList
      addServerRoutes(routeList)
      serverMenus.value = menuResp.data.data.menuTree
      router.push('/')
    }
  } catch (e) {
```



```

        console.error(e)
      }
    }
  }
  onMounted(()=>{
    resetRoutes()
  })
</script>
<style scoped>
.login{
  margin: 200px auto;
  width: 25%;
  padding: 20px;
  height: 180px;
  background-color: antiquewhite;
}
</style>

```

- 登录成功后去请求 `/api/menu/{username}` 获取该用户的菜单和路由
- `router.push` 方法用来以编程方式跳转至主页路由

主页组件

A6Main.vue

```

<template>
  <div class="a6main">
    <a-layout>
      <a-layout-header>
      </a-layout-header>
      <a-layout>
        <a-layout-sider>
          <a-menu mode="inline" theme="dark">
            <template v-for="m1 of serverMenus">
              <a-sub-menu v-if="m1.children" :key="m1.id" :title="m1.title">
                <template #icon><a-icon :icon="m1.icon"></a-icon></template>
                <a-menu-item v-for="m2 of m1.children" :key="m2.id">
                  <template #icon><a-icon :icon="m2.icon"></a-icon></template>
                  <router-link v-if="m2.routePath" :to="m2.routePath">
                    {{m2.title}}</router-link>
                  <span v-else>{{m2.title}}</span>
                </a-menu-item>
              </a-sub-menu>
              <a-menu-item v-else :key="m1.id">
                <template #icon><a-icon :icon="m1.icon"></a-icon></template>
                <router-link v-if="m1.routePath" :to="m1.routePath">{{m1.title}}</router-link>
                <span v-else>{{m1.title}}</span>
              </a-menu-item>
            </template>
          </a-menu>
        </a-layout-sider>
        <a-layout-content>
          <router-view></router-view>
        </a-layout-content>
      </a-layout>
    </div>
  </a-layout>

```

```

    </div>
</template>
<script setup lang="ts">
import AIcon from '../components/AIcon3' // jsx icon 组件
import { serverMenus } from '../router/a6router'
</script>
<style scoped>
.a6main {
  height: 100%;
  background-color: rgb(220, 225, 255);
  box-sizing: border-box;
}
.ant-layout-header {
  height: 50px;
  background-color: darkseagreen;
}

.ant-layout-sider {
  background-color: lightsalmon;
}

.ant-layout-content {
  background-color: aliceblue;
}

.ant-layout-footer {
  background-color: darkslateblue;
  height: 30px;
}

.ant-layout {
  height: 100%;
}

.ant-layout-has-sider {
  height: calc(100% - 50px);
}

</style>

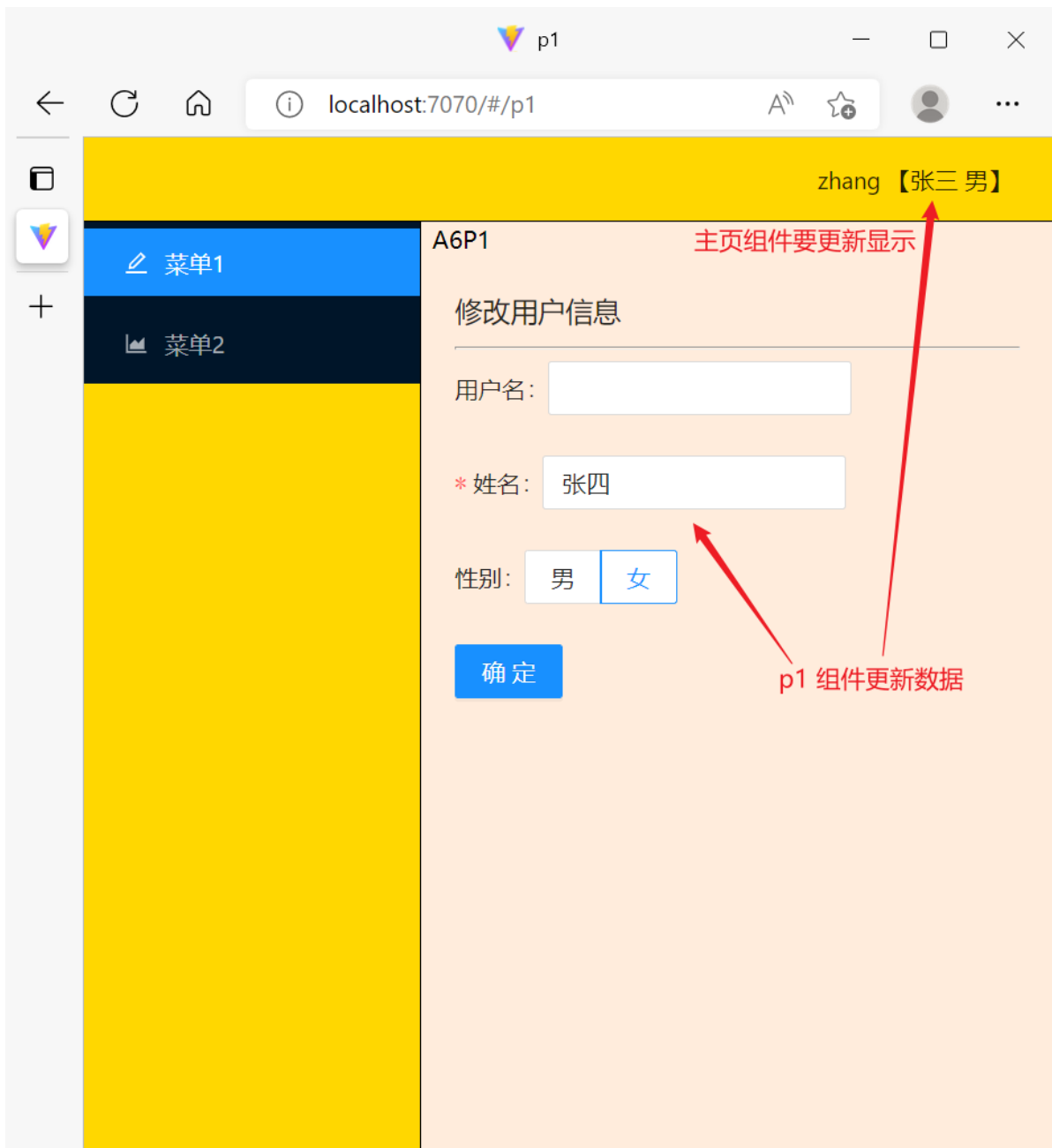
```

token 使用

1. 获取用户信息，例如服务器端可以把用户名、该用户的路由、菜单信息都统一从 token 返回
2. 前端路由跳转依据，例如跳转前检查 token，如果不存在，表示未登录，就避免跳转至某些路由
3. 后端 api 访问依据，例如每次发请求携带 token，后端需要身份校验的 api 需要用到

3) pinia

需求：在组件 p1 里更新了数据，主页组件也同步更新显示



- storage 虽然可以实现多个组件的数据共享，但是需要【主动访问】才能获取更新后的数据
- 本例中由于没有涉及主页组件的 mounted 操作，因此并不会【主动】获取 storage 的数据

安装

```
npm install pinia
```

在 main.ts 中引入

```
import { createPinia } from 'pinia'

// ...
createApp(A6).use(antdv).use(router).use(createPinia()).mount('#app')
```

定义Store

再新建 store 目录来管理共享数据，下面是 /src/store/UserInfo.ts

```
import axios from '../api/request'
import { defineStore } from "pinia"
import { UserInfoDto } from '../model/Model8080'

export const useUserInfo = defineStore('userInfo', {
  state: () => {
    return { username: '', name: '', sex: '' }
  },
  actions: {
    async get(username: string) {
      const resp = await axios.get(`/api/info/${username}`)
      Object.assign(this, resp.data.data)
    },
    async update(dto: UserInfoDto) {
      await axios.post('/api/info', dto)
      Object.assign(this, dto)
    }
  }
})
```

- 定义了 useUserInfo 函数，用来获取共享数据，它可能用于多个组件
 - 命名习惯上，函数变量以 use 打头
- state 定义数据格式
- actions 定义操作数据的方法
 - get 方法用来获取用户信息
 - update 方法用来修改用户信息
- 由于 useRequest 必须放在 setup 函数内，这里简化起见，直接使用了 axios

获取用户信息

```
<template>
  <div class="a6main">
    <a-layout>
      <a-layout-header>
        <span>{{serverUsername}} 【{{userInfo.name}} - {{userInfo.sex}}】 </span>
      </a-layout-header>
      <a-layout>
        <!-- ... -->
      </a-layout>
    </a-layout>
  </div>
</template>
<script setup lang="ts">
import { onMounted } from 'vue';
import AIcon from '../components/AIcon3' // jsx icon 组件
import { serverMenus, serverUsername } from '../router/a6router'
import { useUserInfo } from '../store/UserInfo'
const userInfo = useUserInfo()
```

```

onMounted(()=>{
  userInfo.get(serverUsername.value)
})
</script>

```

修改用户信息

```

<template>
  <div class="a6p1">
    <h3>修改用户信息</h3>
    <hr>
    <a-form>
      <a-form-item label="用户名">
        <a-input readonly v-model:value="dto.username"></a-input>
      </a-form-item>
      <a-form-item label="姓名" v-bind="validateInfos.name">
        <a-input v-model:value="dto.name"></a-input>
      </a-form-item>
      <a-form-item label="性别">
        <a-radio-group v-model:value="dto.sex">
          <a-radio-button value="男">男</a-radio-button>
          <a-radio-button value="女">女</a-radio-button>
        </a-radio-group>
      </a-form-item>
    </a-form>
    <a-button type="primary" @click="onClick">确定</a-button>
  </div>
</template>
<script setup lang="ts">
import { Form } from 'ant-design-vue'
import { onMounted, ref } from 'vue'
import { UserInfoDto } from '../model/Model18080'
import { useUserInfo } from '../store/UserInfo';
const dto = ref<UserInfoDto>({ username: '', name: '', sex: '' })
const userInfo = useUserInfo()
onMounted(()=>{
  Object.assign(dto.value, userInfo)
})
const rules = ref({
  name: [
    {required: true, message:'姓名必填'}
  ]
})
const { validateInfos, validate } = Form.useForm(dto, rules)
async function onClick() {
  try {
    await validate()
    await userInfo.update(dto.value)
  } catch (e) {
    console.error(e)
  }
}
</script>

```

- 不能直接把 userInfo 绑定到表单，需要 dto 中转一下

- userInfo.update 和 userInfo.get 返回的都是 Promise 对象，可以配合 await 一起用

后记

vite + vue3 + vue-router + ts 还没有太多成熟的项目范例，可以参考 [GitHub - sendya/preview-pro: Use pro-layout in vitejs. preview https://sendya.github.io/preview-pro/index.html](https://sendya.github.io/preview-pro/index.html)，它提供了基本的布局 and 样例代码