TypeScript的项目实战

本节课会分别从前端项目和服务端项目的搭建上，介绍TypeScript在实际应用开发场景中的应用方式。

1.TypeScript+Vue3+Vite+VueRouter+Pinia的项目搭建

1.1 使用Vite初始化Vue3+TypeScript项目

开发Vue3项目首先要确保电脑上安装了Vite脚手架，本节课程略过Vite脚手架的介绍。在命令行工具中初始化Vite项目，代码如下：

```
npm init vite
```

接下来选择项目搭配，项目命名为vite-test，技术框架选择vue框架，如下：

```
✔ Project name: … vite-test
? Select a framework: › - Use arrow-keys. Return to submit.
    vanilla
❯   vue
    react
    preact
    lit
    svelte
```

接下来是语种选择，这里选择使用TypeScript初始化项目，如下：

```
✔ Project name: … vite-test
✔ Select a framework: › vue
? Select a variant: › - Use arrow-keys. Return to submit.
    vue
❯   vue-ts
```

创建成功后命令行工具中会弹出如下提示：

```
✔ Project name: … vite-test
✔ Select a framework: › vue
✔ Select a variant: › vue-ts

Scaffolding project in /Users/zhangyunpeng/Desktop/ts/vite-test...

Done. Now run:

  cd vite-test
  npm install
  npm run dev
```

接下来按照日志提供的代码安装依赖并运行项目，测试项目是否正常访问。

1.2 路由的引入

在项目运行成功后，停止项目运行，在控制台中输入路由的安装命令，代码如下：

```
npm install vue-router@next -s
```

安装成功后，在项目的src文件夹下创建router/index.ts文件，在src下创建views/Index.vue和views/Login.vue文件，在两个视图组件中初始化默认结构。在router/index.ts文件中初始化路由的TypeScript配置。

```
import { createRouter,createWebHashHistory, RouteRecordRaw, RouterOptions,Router } from 'vue-router'
// 由于router的API默认使用了类型进行初始化，内部包含类型定义，所以本文件内部代码中的所有数据类型是可以省略的
// RouteRecordRaw是路由组件对象类型
const routes:RouteRecordRaw[] = [
  {
    path:'/',
    name:'Index',
    component:() => import('../views/Index.vue')
  },
  {
    path:'/login',
    name:'Login',
    component:() => import('../views/Login.vue')
  }
]
// RouterOptions是路由选项类型
const routerOptions:RouterOptions = {
  history:createWebHashHistory(),
  routes
}
// Router是路由对象类型
const router:Router = createRouter(routerOptions)
export default router
```

接下来，在main.ts文件中加载路由对象并安装到Vue框架中，代码如下：

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
createApp(App).use(router).mount('#app')
```

在App.vue中将代码改造为如下代码:

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

```
<template>
  <router-view></router-view>
</template>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

运行项目访问路由页面测试路由框架是否正常工作。

1.3 Pinia.js的植入

Pinia是Vue3.x中状态管理框架的新选择，他是又Vuex团队成员开发的，理论上于Vuex的API几乎一样，但是其更轻量，本期课程以Pinia作为状态管理保存登录信息。接下来停止项目，安装pinia，代码如下：

```
npm i pinia -s
```

在本地安装pinia的持久化插件pinia-plugin-persistedstate，代码如下：

```
npm i pinia-plugin-persistedstate -s
```

在项目的src文件夹中创建store/index.ts文件，在文件中初始化Pinia的配置，代码如下：

```
import { defineStore, StoreDefinition } from 'pinia'
export const useStore:StoreDefinition = defineStore('main',{
  state(){
    return {
      userInfo:{},
      token:''
    }
  },
  actions:{
    setUserInfo(userInfo){
      this.userInfo = userInfo
    },
    setToken(token){
      this.token = token
    }
  },
  // 配置持久化策略
```

```
  persist: {
    // 持久化key
    key: 'userData',
    // 持久化对象
    storage: window.sessionStorage,
    // 需要持久化存储的key
    paths: ['userInfo','token'],
    overwrite: true
  },
})
```

配置此步骤是，persist属性可能会报错，原因是Pinia默认的Store对象中并没有为persist声明类型和未知，所以接下来需要讲Pinia与持久化插件整合，并将Pinia与Vue整合，代码如下：

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import { createPinia } from 'pinia'
import piniaPluginPersistedstate from 'pinia-plugin-persistedstate'
const pinia = createPinia()
pinia.use(piniaPluginPersistedstate)
createApp(App).use(router).use(pinia).mount('#app')
```

最后在vite项目的vite.config.ts文件中加入反向代理配置以便于与后台通信，代码如下：

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

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  server:{
    proxy:{
      '/api':{
        target:'http://localhost:8088',
        changeOrigin:true
      }
    }
  }
})
```

到这为止Vite+Vue+TypeScript+VueRouter+Pinia的架构雏形便创建完毕。

2. NestJS的服务端项目搭建

为良好的实现登录业务，接下来在本地搭建一个服务端项目用来模拟接口通信。本节课程使用NestJS作为服务端项目，因为该项目为TypeScript的完美应用场景。确保电脑上安装了@nestjs/cli脚手架（本过程略过），在开发工具中创建一个NestJS的项目，代码如下：

```
nest new nest-test
```

安装依赖后，在main.ts文件中将服务端口改为8088，输入项目的本地开发命令，代码如下：

```
npm run start:dev
```

访问http://localhost:3000会出现欢迎页面。接下来了解一下项目的目录结构，代码如下：

```
.
├── README.md #项目的说明文件
├── nest-cli.json #nest相关配置
├── package-lock.json #依赖锁定文件
├── package.json #项目和依赖描述
├── src #源代码文件夹
│   ├── app.controller.spec.ts #测试用例
│   ├── app.controller.ts #项目的全局控制器
│   ├── app.module.ts #模块编排对象
│   ├── app.service.ts #业务对象
│   └── main.ts #项目的核心配置文件
├── test #测试工具
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json #ts配置文件
└── tsconfig.json #ts配置文件
```

接下来，改造现有文件结构，为用户模块创建响应的对象，代码如下：

```
.
├── README.md
├── nest-cli.json
├── package-lock.json
├── package.json
├── src
│   ├── app.module.ts
│   ├── controller #追加的文件
│   │   └── user.controller.ts#追加的文件
│   ├── main.ts
│   ├── module #追加的文件
│   │   └── user.module.ts #追加的文件
│   └── service #追加的文件
│       └── user.service.ts #追加的文件
├── test
│   ├── app.e2e-spec.ts
```

```
│   └── jest-e2e.json
├── tsconfig.build.json
└── tsconfig.json
```

在src/controller/user.controller.ts文件中初始化控制器代码，代码如下：

```
import { Controller, Get,Post } from '@nestjs/common';
import { UserService } from '../service/user.service';

@Controller('/user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Get("/login")
  login(): string {
    return "login";
  }
}
```

在src/service/user.service.ts文件中初始化业务对象，代码如下：

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class UserService {

}
```

在src/module/user.module.ts文件中初始化模块对象，代码如下：

```
import { Module } from '@nestjs/common';
import { UserController } from '../controller/user.controller';
import { UserService } from '../service/user.service';

@Module({
  imports: [],
  controllers: [UserController],
  providers: [UserService],
})
export class UserModule {}
```

在src/app.module.ts中即成user模块，代码如下：

```
import { Module } from '@nestjs/common';
import { UserModule } from './module/user.module';
@Module({
  imports: [UserModule]
})
export class AppModule {}
```

在src/main.ts中配置项目前缀，代码如下：

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  // 配置公共访问路径
  app.setGlobalPrefix('/api');
  await app.listen(3000);
}
bootstrap();
```

输入项目启动命令并访问http://localhost:3000/api/user/login测试项目运行结果。该项目在启动后会生成dist文件夹，项目src中编写的TypeScript代码仅仅是语法模版，实际运行的项目最终会将TypeScript代码转换成JavaScript执行。

3. 实现登录业务

3.1实现登录页面的搭建

打开vite脚手架项目，在src/views/Login.vue文件中初始化登录表单内容，代码如下：

```
<template>
  login
  <form >
    账号: <input v-model="userInfo.username" /> <br>
    密码: <input v-model="userInfo.password" /> <br>
    <button type="button">提交</button>
  </form>
</template>
<script setup lang="ts">
  import { reactive } from 'vue'
  const userInfo:object = reactive({
    username:'',
    password:''
  })
  const submit = ():void => {

  }
```

```
</script>
```

3.2 实现登录接口的定义

打开nest-test项目，在src/models/result.model.ts中定义接口返回对象，代码如下：

```
export class ResultModel<T>{
  data:T
  code:number
  msg:string
  constructor(code:number,data:T,msg:string){
    this.code = code
    this.data = data
    this.msg = msg
  }
}
```

在src/controller/user.service.ts中加入登录的业务实现，代码如下：

```
import { Injectable } from '@nestjs/common';
import { ResultModel } from '../model/result.model';

@Injectable()
export class UserService {
  login(username:string,password:string):ResultModel<object>{
    let obj = {
      username:'admin',
      password:'123456',
      nickname:'管理员'
    }
    if(obj.username == username&&obj.password ==password){
      return new ResultModel(200,{
        userInfo:obj,
        token:'abc'
      },'登录成功')
    }else{
      return new ResultModel(500,null,'账号或密码错误')
    }
  }
}
```

接下来在src/controller/user.controller.ts文件中加入控制器业务，代码如下：

```
import { Query, Controller, Get,Post } from '@nestjs/common';
import { ResultModel } from 'src/model/result.model';
import { UserService } from '../service/user.service';

@Controller('/user')
```

```
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Get("/login")
  login(@Query('username') username:string,@Query('password') password:string):
ResultModel<object> {
    console.log(username,password)
    return this.userService.login(username,password);
  }
}
```

3.3 登录功能的实现

在src/views/Login.vue文件中加入登录实现，代码如下：

```
<template>
  login
  <form >
    账号: <input v-model="userInfo.username" /> <br/>
    密码: <input v-model="userInfo.password" /> <br/>
    <button type="button" @click="submit">提交</button>
  </form>
</template>
<script setup lang="ts">
  import { reactive } from 'vue'
  import { useStore } from '../store/index'
  import { useRouter } from 'vue-router'
  const store = useStore()
  console.log(store)
  const router = useRouter()
  const userInfo = reactive({
    username:'',
    password:''
  })
  const submit = async ():Promise<any> => {
    let res =
      await fetch(`/api/user/login?
username=${userInfo.username}&password=${userInfo.password}`).then(res => res.json())
    console.log(res)
    if(res.code == 200){
      store.setUserInfo(res.data.userInfo)
      store.setToken(res.data.token)
      router.push('/')
    }else{
      alert(res.msg)
    }
  }

</script>
```

在src/views/Index.vue文件中加入登录信息展示，代码如下：

```
<template>
  登录信息：
  {{userInfo}}
</template>
<script setup lang="ts">
  import { computed } from 'vue'
  import { useStore } from '../store/index'
  const store = useStore()
  console.log(store)
  let userInfo = computed(() => store.userInfo)
</script>
```