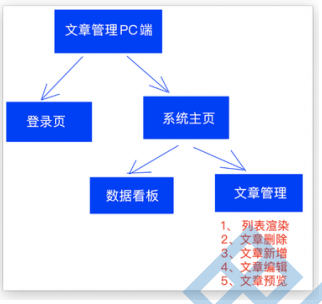
11-Vue3全家桶+ElementPlus实战

一、项目演示及收获

1. 效果演示



2. 项目收获

...

3. 项目技术栈

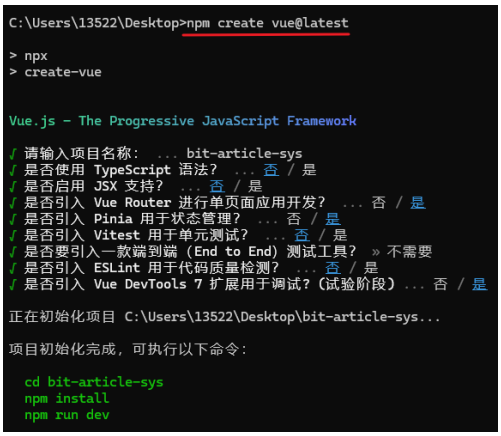
vue3全家桶(vue3 + vue-router + pinia) + vite + element-plus + axios + es6+(体系) + scss ...

二、创建项目

1. 选定项目创建位置

2. 打开终端(命令行)窗口

按下图操作:



3. 运行项目

3.1 进入项目

```
cd bit-article-sys
```

3.2 安装依赖

```
npm i
```

3.3 启动项目

```
npm run dev
```

4. 预览

浏览器打开: http://localhost:5173

三、认识并调整目录

脚手架默认创建的目录结构有些不满足我们的开发需求，需要做一些自定义改动。主要有：

1. 删除初始化的默认文件
2. 修改剩余代码内容
3. 新增调整我们需要的目录结构

1. 删除文件

- components 下的所有文件和目录

- views 下的所有文件
- assets 下的所有文件
- stores 下的 counter.js

2. 修改内容

`src/router/index.js`

```
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  // import.meta.env.BASE_URL: 指明路由启动后的基础路径，默认值是 /
  history: createWebHistory(import.meta.env.BASE_URL),
  // 配置路由规则
  routes: [

  ]
})

export default router
```

`src/main.js`

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router'

const app = createApp(App)

app.use(createPinia())
app.use(router)

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

`src/App.vue`

```
<script setup></script>

<template>
```

```vue
    <!-- 一级路由出口 -->
    <router-view />
</template>

<style scoped></style>
```

3. 新增目录

在 src 目录下中新建以下目录：

- /api： 存储请求函数模块
- /styles: 样式文件模块
- /utils: 工具函数模块

- 将需要的图片放置在 `assets` 目录下

四、引入Element组件库

1. 全部引入(推荐)

如果你对打包后的文件大小不是很在乎，那么使用完整导入会更方便。

1、安装

```bash
# npm
npm i element-plus

# yarn
yarn add element-plus

# pnpm
pnpm i element-plus
```

2、导入并注册

`main.js`

```
// 导入组件库
import ElementPlus from 'element-plus'
// 导入组件样式
import 'element-plus/dist/index.css'
// 全局注册所有组件
app.use(ElementPlus)
```

2. 按需引入

1、安装

```
# npm
npm i element-plus

# yarn
yarn add element-plus

# pnpm
pnpm i element-plus
```

2、安装两款插件

```
# npm
npm i unplugin-vue-components unplugin-auto-import -D

# yarn
yarn add unplugin-vue-components unplugin-auto-import -D

# pnpm
pnpm i unplugin-vue-components unplugin-auto-import -D
```

3、加入Vite配置

`vite.config.js`

```
import AutoImport from 'unplugin-auto-import/vite'
import Components from 'unplugin-vue-components/vite'
import { ElementPlusResolver } from 'unplugin-vue-components/resolvers'

```

```
export default defineConfig({// ...
  plugins: [

   // ...
   AutoImport({
    resolvers: [
      ElementPlusResolver()
    ]
  }),
   Components({
      resolvers: [
        ElementPlusResolver()
      ]
    })
  ]
})
```

3. 主题色定制

自定义主题: https://element-plus.org/zh-CN/guide/theming.html

1、新建 `styles/index.scss`

```
body {
  margin: 0;
  background: #fafbfe;
}

:root {
  // 修改 ElementPlus 样式颜色值
  --el-color-primary: rgba(114, 124, 245, 1);
}
```

2、main.js 引入

```
import '@/styles/index.scss'
```

五、公共模块封装

1. 封装request请求模块

1.1 为什么封装

我们会使用 axios 来请求后端接口, 一般会对 axios 进行一些统一配置 (比如: 基础地址等)

项目开发中, 都会对 axios 进行基本的二次封装, 单独封装到一个模块中, 便于使用

1.2 步骤

1、安装

```
npm i axios
```

2、新建 `utils/request.js` 请求模块

Axios中文网

```
/* 封装 axios 用于发送请求 */


// 导入 axios
import axios from 'axios'

// 创建一个新的axios实例
const instance = axios.create({
  // 请求基础地址
  baseURL: 'http://localhost:4000/api',
  // 请求超时时间
  timeout: 5000
})

// 请求拦截器
instance.interceptors.request.use((config) => {
  // 在发送请求之前做些什么
  return config
},  (error) => {
  // 对请求错误做些什么
  return Promise.reject(error)
})

// 响应拦截器
instance.interceptors.response.use((response) => {
  // 对响应数据做点什么
  return response
},  (error) => {
  // 对响应错误做点什么
  return Promise.reject(error)
})
```

```
32
33 // 默认导出请求实例
34 export default instance
```

2. 封装token本地管理模块

2.1 为什么封装

`token` 是用户登录的凭证, 非常重要。需要同时通过 Pinia和本地存储 共同保存

- 当用户登录成功后，我们要保存 token;
- 当用户退出登录时，要删除 token, 并且还需要通过
- 当刷新页面时, 我们要从本地获取token

为了便于后期使用方便, 我们封装一个对 token 进行本地存、取、删的公共模块

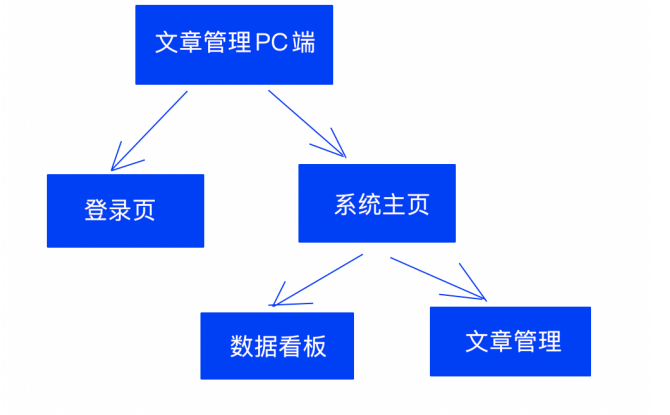
2.2 步骤

```
1  // 本地存储token的键名
2  const TOKEN_KEY = 'BIT_TOKEN_KEY'
3
4  // 存
5  export const setToken = (token) => {
6    localStorage.setItem(TOKEN_KEY, token)
7  }
8
9  // 取
10 export const getToken = () => {
11   return localStorage.getItem(TOKEN_KEY)
12 }
13
14 // 删
15 export const removeToken = () => {
16   localStorage.removeItem(TOKEN_KEY)
17 }
```


六、路由设计

1. 分析

凡是单个页面、独立展示的, 都是一级路由



整理思路: 新建页面 -> 配置规则 -> 路由出口

路由设计：

- 登录页 （一级） Login
- 首页架子（一级） Layout
  - 数据看板（二级）Dashboard
  - 文章管理（二级）Article

2. 新建页面

3. 配置规则

`router/index.js`

```
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  // import.meta.env.BASE_URL:  指明路由启动后的基础路径，默认值是 /
  history: createWebHistory(import.meta.env.BASE_URL),
  // 路由规则表
  routes: [
    {
      // 根路径
      path: '/',
      // 路由懒加载：当页面被访问的时候，才去加载对应的文件，从而提高首屏的加载速度
      component: () => import('@/views/Layout/index.vue'),
```

```
      // 重定向：防止访问根路径，页面空白
      redirect: '/dashboard',
      // 路由嵌套
      children: [
        {
          // 数据看板
          path: 'dashboard',
          component: () => import('@/views/Dashboard/index.vue')
        },
        {
          // 文章管理
          path: 'article',
          component: () => import('@/views/Article/index.vue')
        }
      ]
    },
    {
      // 登录
      path: '/login',
      component: () => import('@/views/Login/index.vue')
    }
  ]
})

export default router
```

4. 路由出口

`App.vue`

```
<script setup></script>
<template>
  <!-- 一级路由出口 -->
  <router-view />
</template>
<style lang="scss" scoped></style>
```

`views/Layout/index.vue`

```
<script setup></script>
<template>
  <div>
    <div>左侧</div>
```

```
    <div>
      <header>头部</header>
      <main>
        <!-- 二级路由出口 -->
        <router-view />
      </section>
    </div>
  </div>
</template>
<style lang="scss" scoped>

</style>
```

七、登录模块

1. 需求

2. 分析

3. 代码示例

3.1 静态标签结构

```
<script setup></script>
<template>
  <div class="login-box">
    <!-- 卡片组件 -->
```

```
    <el-card>
      <!-- 头部 -->
      <template #header>
        <div class="card-header">
          <h2>比特文章管理系统</h2>
        </div>
      </template>
      <!-- 主体：表单 -->
      <el-form>
        <el-form-item>
          <el-input
            type="text"
            clearable
            placeholder="请填写用户名"
            prefix-icon="User" />
        </el-form-item>
        <el-form-item>
          <el-input
            type="password"
            clearable
            show-password
            placeholder="请填写密码"
            prefix-icon="Lock" />
        </el-form-item>
        <el-form-item>
          <el-button type="primary">登 录</el-button>
          <el-button>取 消</el-button>
        </el-form-item>
      </el-form>
    </el-card>
  </div>
</template>
```


3.2 样式美化

```
<style scoped lang="scss">
  .login-box {
    display: flex;
    justify-content: center;
    align-items: center;
    width: 100vw;
    height: 100vh;
    background: url('@/assets/login-bg.svg') center/cover;
    .el-card {
      width: 40vw;
```

```
      height: 40vh;
      // :deep(选择器):  scss 样式穿透的语法，什么时候需要用到样式穿透:
      // 1. 当前组件的 style 添加了 scoped
      // 2. 修改的是第三方组件内部元素的样式
      :deep(.el-card__header) {
        display: flex;
        justify-content: center;
        align-items: center;
        height: 70px;
        // 注意: var(css变量)，好处就是利于今后的代码维护和修改
        background: var(--el-color-primary);
        color: #fff;
        h2 {
          font-size: 16px;
          font-weight: 400;
        }
      }
      .el-form {
        padding: 15px 25px;
        .el-form-item:nth-child(2) {
          margin: 40px 0;
        }
        .el-input {
          height: 45px;
        }
        .el-button {
          width: 65px;
          height: 35px;
        }
      }
    }
  }
</style>
```


4. 样式控制

默认情况下, 写在 scoped 中的样式，只会影响到当前组件中的元素；在添加了 scoped 的情况下，如果希望样式向下影响到(组件内部)子元素, 那么需要借助 `深层作用选择器` , 即可让样式穿透, 从而作用在子元素上

在 `scss` 中，深层作用选择器的语法:

```
:deep(当前选择器) {
  // some style code ...
}
```

```css
:deep(.el-card__header) {
    display: flex;
    justify-content: center;
    align-items: center;
    height: 70px;
    // 注意：var(css变量)，好处就是利于今后的代码维护和修改
    background: var(--el-color-primary);
    color: #fff;
}
```

5. 输入框添加小图标

5.1 安装字体模块

```
npm i @element-plus/icons-vue
```

5.2 导入并注册字体组件

```
// main.js

// 按需导入
import {
  User,
  Lock
} from '@element-plus/icons-vue'

// 全局注册
app.component(User.name, User)
app.component(Lock.name, Lock)
```

5.3 input 添加 `prefix-icon` 属性

```
<el-input prefix-icon="User" />

<el-input prefix-icon="Lock" />
```

八、表单校验

1. 说明

在向后端发请求、调用接口之前, 我们需要对表单进行验证，把用户的错误扼杀在摇篮之中。

2. 步骤

Form 表单组件提供了数据验证的功能

Form 表单组件的校验：

1. el-form: 绑定 `model` 表单对象 和 `rules` 验证规则
2. el-form-item: 绑定 `prop` 属性
3. el-input: 绑定 `v-model`

2.1 定义 model 和 rules 并绑定

```
import { reactive } from 'vue'

// 登录表单
const loginForm = reactive({
  username: '', // 用户名
  password: '' // 密码
})

// 校验规则
const loginFormRules = {
  username: [
    { required: true, message: '用户名不能为空', trigger: 'blur' },
    { min: 3, max: 10, message: '长度在 3 到 10 个字符', trigger: 'blur' }
  ],
  password: [
    { required: true, message: '密码不能为空', trigger: 'blur' },
    { min: 5, max: 20, message: '长度在 5 到 20 个字符', trigger: 'blur' }
  ]
}
```

```
<el-form
    :model="loginForm"
    :rules="loginFormRules"
  >

</el-form>
```

2.2 form-item 绑定 `prop` 属性

将 form-item 的 `prop` 属性设置为需校验的字

```
<el-form-item prop="username"></el-form-item>

<el-form-item prop="password"></el-form-item>
```

2.3 input 进行 v-model 双向绑定

```
<el-input v-model="loginForm.username" />

<el-input v-model="loginForm.password" />
```

九、提交表单校验 和 重置

1. 说明

兜底校验: 每次点击登录按钮, 进行 ajax 请求前, 应该先对整个表单内容校验, 不然会发送很多无效的请求!!!

要保证通过校验了, 才能发送请求!!!

2. 步骤

- 校验: 获取 Form 组件实例, 调用 `validate()` 方法
- 重置: 获取 Form 组件实例, 调用 `resetFields()` 方法

2.1 准备 ref 并绑定

```
<script setup>
  import { ref } from 'vue'

  const loginFormRef = ref(null)
</script>

<template>
    <el-form ref="loginFormRef">
      ...
    </el-form>
</template>
```

2.2 按钮绑定事件并处理

```
<script setup>
  // 登录
  const onLogin = () => {
    // 整体校验
    loginFormRef.value.validate((valid) => {
      if (!valid) return
      console.log('OK')
    })
  }

  // 重置
  const onCancel = () => {
    loginFormRef.value.resetFields()
  }
</script>
<template>
  <el-button type="primary" @click="onLogin">登 录</el-button>
  <el-button @click="onCancel">取 消</el-button>
</template>
```

十、封装登录请求

1. 封装接口并导出

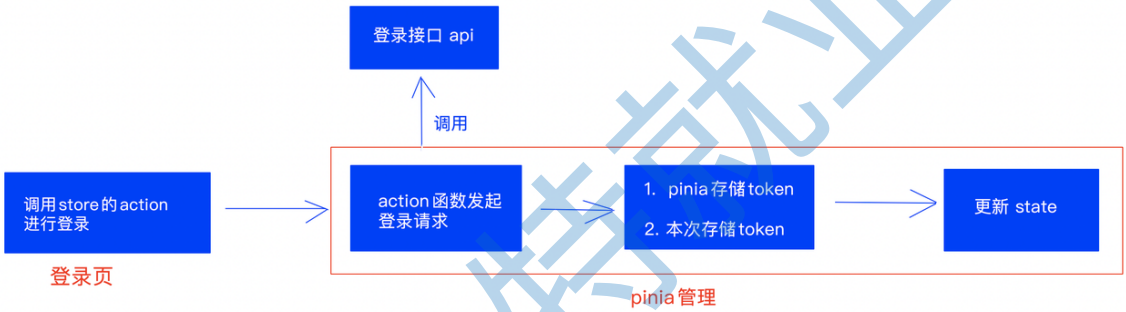
新建 `api/user.js` 模块，提供登录接口

```
// 导入公共请求函数
import request from '@/utils/request'

// 定义并导出登录接口函数
export function loginApi(loginForm) {
  // 注意：这里 return 不能丢
  return request({
    method: 'post',
    url: '/login',
    data: loginForm
  })
}
```

2. 登录页测试接口

```
import { loginApi } from '@/api/user'

// 登录
const onLogin = () => {
  loginFormRef.value.validate(async (valid) => {
    if (!valid) return

    // 调用接口
    const resp = await loginApi(loginForm)
    console.log(resp)
  })
}
```

十一、Pinia 管理 token



1. 新建 userStore 模块

`stores/user.js`

```
import { defineStore } from 'pinia'
import { loginApi } from '@/api/user'
import { getToken, setToken } from '@/utils/token'

// 定义并导出用户 store
export const useUserStore = defineStore('user', {
  state: () => ({
    // token
    token: getToken() || '',
    // 用户信息
    userInfo: {}
  }),
  actions: {
    setToken(token) {
```

```
      this.token = token
      // token 存本地
      setToken(token)
    },
    // 登录
    async loginAction(loginForm) {
      // 发起登录请求(调用登录接口)
      // await 只能获取成功态的 Promise 实例结果
      const resp = await loginApi(loginForm)
      // 解构出 token
      const { token } = resp
      // 对 token 赋值
      this.setToken(token)
    }
  }
})
```

2. 调用登录 action

`Login/index.vue`

```
<script setup>
// 导入用户仓库模块
import { useUserStore } from '@/store/user'
// 得到用户仓库
const userStore = useUserStore()

// 登录
const onLogin = () => {
  loginFormRef.value.validate((valid) => {
    if (!valid) return
    // 调用登录 action
    userStore.loginAction(loginForm)
  })
}
```

3. 登录后续操作

```
<script setup>
import { useRouter } from 'vue-router'
import { ElMessage } from 'element-plus'
const router = useRouter()

```

```
// 登录
const onLogin = () => {
  loginFormRef.value.validate(async (valid) => {
    if (!valid) return
    try {
      // 这里必须添加 await 的原因是 loginAction 是异步的,
      // 需要等该函数执行完，才能确定的登录成功还是失败
      await userStore.loginAction(loginForm)
      // 成功提示
      ElMessage.success('登录成功')
      // 跳转至首页
      router.push('/')
    } catch (e) {
      // 错误提示
      ElMessage.error(e.message)
    }
  })
}
</script>
```

十二、 axios 响应成功拦截

1. 说明

为了后续请求获取数据方便, 因为所有的响应成功了, 都会经过 axios 响应拦截器的成功回调,

因此可以在这里对成功的数据做统一处理

2. 代码示例

`utils/request.js`

```
// 自定义响应状态码映射对象
const RESPONSE_CODE = {
  success: 10000, // 成功
}
// 响应拦截器
instance.interceptors.response.use(
  (response) => {
    // 对响应成功做公共处理，响应状态码以2开头

    // 解构出 data
    const { data: resp } = response
    // 解构出 code, data, message
    const { code, data, message } = resp
    // 响应成功
```

```
    if (code === RESPONSE_CODE.success) {
      // 直接返回纯粹的数据
      return data
    }
    // 失败
    // 返回失败态的Promise实例，并携带错误信息
    return Promise.reject(new Error(message))
  }, (error) => {
    return Promise.reject(error)
  })
```

十三、登录访问拦截

1. 说明

在没有登录的情况下, 如果要访问的不是登录页, 则先去登录; 否则放行

2. 代码示例

`router/index.js`

```
// 导入用户仓库
import useUserStore from '@/store/user'

// 白名单：无需登录就可以访问的路由路径
const whiteList = ['/login']

// 全局前置守卫
router.beforeEach((to, from) => {
  const userStore = useUserStore()
  // 从 pinia 中获取 token
  const { token } = userStore
  if (token) {
    // 登录了
    // 还想去登录
    if (to.path === '/login') {
      // 直接重定向到首页
      return '/'
    } else {
      // 放行
      return true
    }
  } else {
    // 未登录
    if (whiteList.includes(to.path)) {
```

```
      // 放行
      return true
    } else {
      // 重定向到登录，并且携带回跳地址
      return `/login?redirectUrl=${to.path}`
    }
  }
})
```

3. 完善登录跳转

`views/Login/index.vue`

```
import { useRoute } from 'vue-router'
const route = useRoute()

// 登录
const onLogin = () => {
  // 整体校验
  loginFormRef.value.validate(async (valid) => {
    if (!valid) return ElMessage.error('用户名或密码不对')
    try {
      // 发起登录请求
      await userStore.loginAction(loginForm)
      // 成功的提示
      ElMessage.success('登录成功')
      // 登录成功后：优先跳转到回跳地址；否则跳转到首页
      router.push(route.query.redirectUrl || '/')
    } catch (e) {
      // 错误消息提示
      ElMessage.error(e.message)
    }
  })
}
```

十四、Layout布局及请求token拦截

1. 整体流程

2. 静态代码

```vue
<script setup>
  import { useRoute } from 'vue-router'

  const route = useRoute()
</script>

<template>
  <el-container class="layout-box">
    <el-aside width="200px">
      <div class="logo">
        <img
          src="@/assets/logo.jpg"
          alt="logo"
          width="100%"
          height="160" />
      </div>
      <el-menu
        router
        :default-active="route.path"
        background-color="#313a46"
        active-text-color="#fff"
        text-color="#8391a2">
        <el-menu-item index="/dashboard">
          <el-icon><PieChart /></el-icon>
          <span>数据看板</span>
        </el-menu-item>
        <el-menu-item index="/article">
          <el-icon><Notebook /></el-icon>
          <span>文章管理</span>
        </el-menu-item>
      </el-menu>
    </el-aside>
    <el-container>
      <el-header>
        <div class="user">
          <el-avatar :size="36" />
          <el-link :underline="false"> xxx </el-link>
        </div>
        <div class="logout">
          <el-icon>
            <SwitchButton />
          </el-icon>
        </div>
      </el-header>
      <el-main>
        <router-view />
```

```
      </el-main>
    </el-container>
  </el-container>
</template>

<style scoped lang="scss">
  .layout-box {
    .el-aside {
      height: 100vh;
      background: #313a46;
      .el-menu {
        border-right: none;
        .el-menu-item {
          display: flex;
          justify-content: center;
        }
      }
    }
    .el-header {
      display: flex;
      justify-content: flex-end;
      align-items: center;
      box-shadow: 0 0 35px 0 rgba(151, 161, 171, 0.2);
      background: #fff;
      .user {
        display: flex;
        align-items: center;
        height: 100%;
        background: #fafbfc;
        padding: 0 15px;
        border: 1px solid #f1f3f5;
        .el-avatar {
          margin-right: 15px;
        }
      }
      .logout {
        padding: 0 15px;
        font-size: 20px;
        color: #999;
      }
    }
  }
</style>
```


3. 封装接口

`api/user.js`

```
export function getUserInfoApi() {
  return request({
    method: 'get',
    url: '/profile'
  })
}
```

4. 请求拦截器携带token

`utils/request.js`

```
import useUserStore from '@/store/user'
// 请求拦截器
instance.interceptors.request.use(
  (config) => {
   // config: 包含了所有请求信息的对象，比如: method/url/headers...

    // 对请求做公共处理
    const userStore = useUserStore()
    // 获取 token
    const { token } = userStore
    // token 有值
    if (token) {
      // 在请求头中统一携带 token
      config.headers.Authorization = `Bearer ${token}`
    }
    // 给服务器返回请求对象
    return config
  },
  (error) => Promise.reject(error)
)
```

5. Pinia 管理用户信息

`stores/user.js`

```
import { getUserInfoApi } from '@/api/user'

export const useUserStore = defineStore('user', {

```

```
  actions: {
    setUserInfo(userInfo) {
      this.userInfo = userInfo
    },
    // 登录 action
    async loginAction(loginForm) {
      // 发请求
      const resp = await loginApi(loginForm)
      // 存储数据
      const { token } = resp
      this.setToken(token)
    },

    // 获取用户信息
    async getUserInfoAction() {
      try {
        // 发请求（调用接口函数）
        const userInfo = await getUserInfoApi()
        // 设置 userInfo
        this.setUserInfo(userInfo)
      } catch (e) {}
    }
  }
})
```

6. 进入 Layout 调用 action

```
<script setup>
import useUserStore from '@/store/user'
const userStore = useUserStore()

// 获取用户信息
userStore.getUserInfoAction()
</script>
```

7. Layout 头部渲染用户信息

```
<div class="user">
  <el-avatar :size="36" :src="userStore.userInfo.avatar" />
  <el-link :underline="false">{{ userStore.userInfo.name || userStore.userInfo.username }}</el-link>
</div>
```

十五、退出登录

1. Pinia 封装 action

```
actions: {
  // 退出
  logout() {
    this.token = ''
    removeToken()

    this.userInfo = {}
  }
}
```

2. 监听确认事件

```
<script setup>
  // 退出
  const onLogout = () => {

  }
</script>

<template>
  <el-popconfirm @confirm="onLogout">
    <template #reference>
      <el-icon><SwitchButton /></el-icon>
    </template>
  </el-popconfirm>
</template>
```

3. 调用 action 并回到登录

```
<script setup>
  import { useRouter } from 'vue-router'
  const router = useRouter()
 // 退出登录
  const onLogout = () => {
    // 调用 userStore 的 logout 函数
    userStore.logout()
```

```
    // 跳转至登录页
    router.push('/login')
  }
</script>
```

十六、处理 token 过期

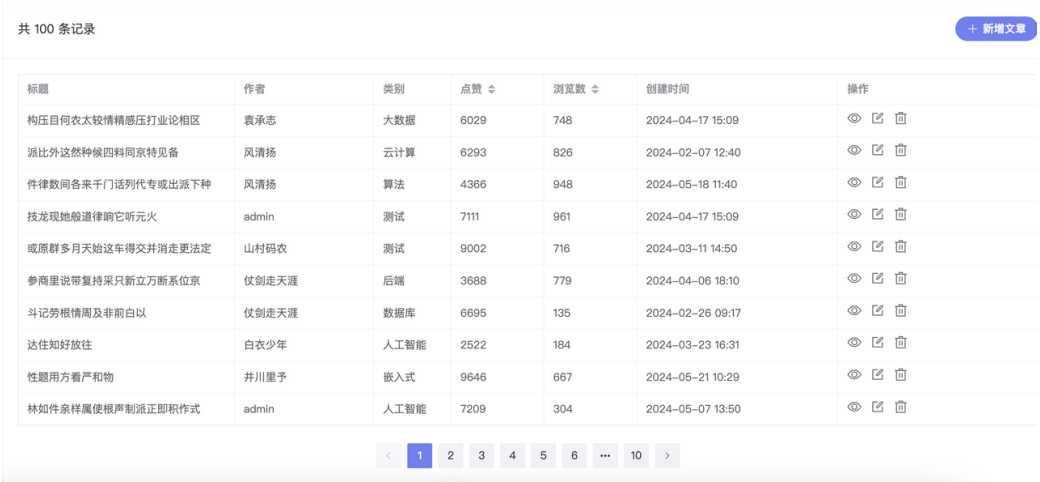
1. 整理流程

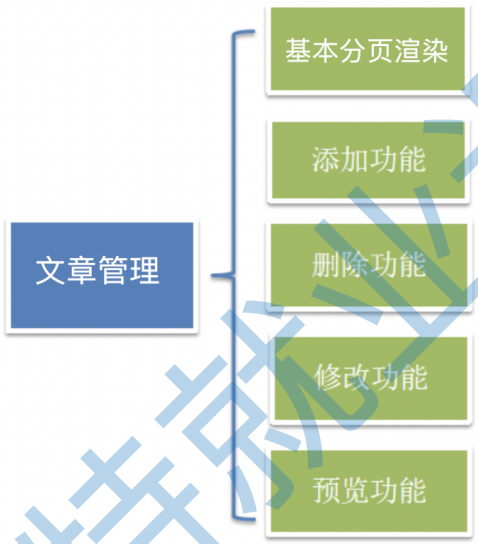
2. 代码示例

```
import router from '@/router'

const RESPONSE_CODE = {
  unauthorized: 401 // 未认证
}
// 响应拦截器
instance.interceptors.response.use(
  (response) => {

  },
  async (error) => {
    const { status, data: { message } } = error.response
    // 统一错误提示
    ElMessage.error(message)
    // token 过期
    if (status === RESPONSE_CODE.unauthorized) {
      const userStore = useUserStore()
      userStore.logout()
      router.push(`/login?redirectUrl=${router.currentRoute.value.path}`)
    }
    // 给组件返回错误信息
    return Promise.reject(new Error(message))
  }
)
```


十七、明确文章管理模块需求

1. 整体效果





十八、准备文章管理结构

1. 静态结构

```vue
<script setup>
</script>
<template>
  <div class="article-box">
    <el-breadcrumb separator-icon="arrow-right">
      <el-breadcrumb-item :to="{ path: '/' }">首页</el-breadcrumb-item>
      <el-breadcrumb-item>文章管理</el-breadcrumb-item>
    </el-breadcrumb>
    <el-card>
      <!-- 头部 -->
      <template #header>
        <div class="card-header">
          <span>共 xxx 条记录</span>
          <el-button type="primary" round>
```

```
            <el-icon><Plus /></el-icon>
            <span>新增文章</span>
          </el-button>
        </div>
      </template>
    </el-card>
  </div>
</template>

<style lang="scss" scoped>
  .article-box {
    .el-card {
      margin-top: 20px;
      .card-header {
        display: flex;
        justify-content: space-between;
        align-items: center;
        height: 40px;
      }
      :deep(.el-card__footer) {
        margin-top: -15px;
        border-top: none;
      }
      .el-table {
        .el-icon:nth-child(2) {
          margin: 0 15px;
        }
      }
      .el-pagination {
        display: flex;
        justify-content: center;
      }
    }
  }
</style>
```


十九、渲染功能-定义接口并调用

1. 定义接口

`api/article.js`

```
export const getArticleListApi = (query) => {
  return request({
    method: 'get',
```

```
    url: '/article',
    params: query
  })
}
```

2. 调用接口

`views/Article/index.vue`

```
import { reactive, ref } from 'vue'
// 文章列表
const articleList = ref([])
// 总数量
const totalCount = ref(0)

// 请求查询参数
const query = reactive({
  current: 1, // 当前页
  pageSize: 10 // 每页条数
})

getArticleList()

// 获取并保存文章列表
async function getArticeList() {
  // 发请求
  const { rows, total } = await getArticleListApi(query)
  // 保存列表数据
  articleList.value = rows
  // 保存数据总条数
  totalCount.value = total
}
```


二十、渲染功能-动态渲染表格

1. 数据渲染

`views/Article/index.vue`

```
<el-table :data="articleList" border striped>
  <el-table-column prop="stem" label="标题" width="280" />
  <el-table-column prop="creator" label="作者" width="180" />
  <el-table-column prop="category" label="类别" width="100" />
```

```
  <el-table-column prop="likeCount" label="点赞" width="120" sortable />
  <el-table-column prop="views" label="浏览数" width="120" sortable />
  <el-table-column prop="createdAt" label="创建时间" />
  <el-table-column label="操作"></el-table-column>
</el-table>
```

其中 `sortable` 表示该列可排序

2. 插槽自定义操作列

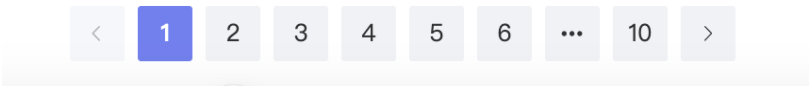
```
<el-table-column label="操作">
    <template>
      <el-tooltip placement="bottom" content="预览">
        <el-icon :size="18"><View /></el-icon>
      </el-tooltip>
      <el-tooltip placement="bottom" content="编辑">
        <el-icon :size="18"><Edit /></el-icon>
      </el-tooltip>
      <el-tooltip placement="bottom" content="删除">
        <el-icon :size="18"><Delete /></el-icon>
      </el-tooltip>
    </template>
 </el-table-column>
```

3. main.js导入并注册图标组件

```
import {
  View,
  Edit,
  Delete
} from '@element-plus/icons-vue'

app.component(View.name, View)
app.component(Edit.name, Edit)
app.component(Delete.name, Delete)
```

二十一、渲染功能-分页渲染

1. 效果



2. 认识分页组件

Pagination分页组件 | Element Plus

当数据量过多时，使用分页分解数据。

3. 代码示例

```
<script setup>
  // 当前页码改变时
  const onCurrentChange = (val) => {
    // 更新页码
    query.current = val
    // 请求列表
    getArticleList()
  }
</script>
<template #footer>
    <!-- 分页组件 -->
    <el-pagination
      @current-change="onCurrentChange"
      v-model:current-page="query.current"
      :total="totalCount"
      background
      layout="prev, pager, next"
     />
  </template>
```


二十二、删除文章

1. 定义接口

`api/article.js`

```
/**
 * 删除文章
 */
export const delArticleApi = (articleId) => {
  return request({
    method: 'delete',
    url: `/article/${articleId}`
```

```
  })
}
```

2. 点击删除

2.1 添加点击事件

```
<el-icon :size="18" @click="onDel(row.id)"><Delete /></el-icon>
```

2.2 删除确认

```
import { ElMessage, ElMessageBox } from 'element-plus'

// 删除
const onDel = async (articleId) => {
  try {
    // 确认了
    await ElMessageBox.confirm('此操作不可逆、确认删除么?', 'Warning')
    // ...
  }catch(e) {
    // 取消了
    console.log(e)
  }
}
```

2.3 完成删除

```
import { delArticleApi } from '@/api/article'

// 删除
const onDel = async (articleId) => {
  try {
    await ElMessageBox.confirm('此操作不可逆、确认删除么?', '友情提示')
    // 删除请求
    await delArticleApi(articleId)
    // 刷新列表
    getArticleList()
  } catch (e) {
    console.log(e)
  }
}
```

二十三、抽屉打开和关闭

1. 认识抽屉组件

Drawer抽屉组件 | Element Plus

```
<!--
    title="我是标题"
    visible="布尔值" 控制显示隐藏
    direction="direction"  控制方向
    size="60%"  窗体所占的区域多宽
-->

<script>
  // 抽屉是否可见
  const visible = ref(false)
</script>


<el-drawer
  v-model="visible"
  title="大标题"
  direction="rtl"
  size="45%"
 >
  <span>我来啦!</span>
</el-drawer>
```

2. 控制抽屉打开和关闭

添加、编辑、预览，都要打开抽屉,可以复用

```
<script setup>
  // 文章操作类型
  const ARTICLE_OPS_TYPE = {
    add: 'add', // 新增
    edit: 'edit', // 编辑
    preview: 'preview' // 预览
  }
  // 当前操作类型
  const currentOpsType = ref('')

```

```
  // 打开抽屉
  const onOpen = () => {
    visible.value = true
  }
</script>

<template>
  <el-button type="primary" round @click="onOpen">
   <el-icon><Plus /></el-icon>
   <span>新增文章</span>
 </el-button>


   <template #default="{ row }">
    <el-tooltip placement="bottom" content="预览">
     <el-icon :size="18" @click="onOpen"
      ><View
     /></el-icon>
   </el-tooltip>
   <el-tooltip placement="bottom" content="编辑">
    <el-icon :size="18" @click="onOpen"
      ><Edit
    /></el-icon>
   </el-tooltip>
   <el-tooltip placement="bottom" content="删除">
    <el-icon :size="18"><Delete /></el-icon>
   </el-tooltip>
  </template>
</template>
```


二十四、记录操作类型并控制标题

1. 记录操作类型

```
<script setup>
  // 记录当前的操作类型
  const currentOpsType = ref('')
  // 记录当前操作的文章 id
  const articleId = ref('')

  // 打开抽屉
  const onOpen = (opsType, id) => {
    visible.value = true
    // 保存当前操作类型
    currentOpsType.value = opsType
```

```
    // 保存当前的文章id
    articleId.value = id
  }
</script>
<template>
  <el-button
    type="primary"
    round
    @click="onOpen(ARTICLE_OPS_TYPE.add, '')">
    <el-icon><Plus /></el-icon>
    <span>新增文章</span>
  </el-button>

    <el-icon
      size="18"
      @click="onOpen(ARTICLE_OPS_TYPE.preview, row.id)"
    ><View/></el-icon>

    <el-icon
      size="18"
      @click="onOpen(ARTICLE_OPS_TYPE.edit, row.id)"
    ><Edit/></el-icon>
</template>
```

2. 声明映射对象

```
// 标题的映射对象
const TITLE_MAP = {
  add: '新增',
  edit: '编辑',
  preview: '预览'
}
```

3. 计算属性得到动态标题

```
// 当前展示的标题
const activeTitle = computed(() => {
  return TITLE_MAP[currentOpsType.value] + '面经'
})
```

4. 抽屉组件绑定计算属性

```vue
<el-drawer :title="activeTitle">
  ...
</el-drawer>
```

二十五、抽屉组件主体内容

1. 目标

新增、编辑、预览 共用一个组件

2. 新建组件

`Article/components/ArticleOps.vue`

```vue
<el-form label-width="55px">
  <el-form-item label="标题">
    <el-input  placeholder="请输入文章标题" />
  </el-form-item>
  <el-form-item label="内容">
    富文本编辑器
  </el-form-item>
  <el-form-item>
    <el-button type="primary">确 认</el-button>
    <el-button>取 消</el-button>
  </el-form-item>
</el-form>

<style lang="scss" scoped>
 .el-form {
    .el-input {
      height: 36px;
    }
  }
</style>
```

3. 使用组件

`Article/index.vue`

```vue
<script setup>
  import ArticleOps from './components/ArticleOps.vue'
</script>
```

```
4
5 <template>
6   <el-drawer>
7     <ArticleOps />
8   </el-drawer>
9 </template>
```

二十六、集成富文本编辑器

Vuejs优秀插件集合 | GitHub - vuejs/awesome-vue

1. 安装 VueQuill

VueQuill官方文档

```
1 npm install @vueup/vue-quill@latest
2
3 # or
4
5 yarn add @vueup/vue-quill@latest
```

2. 导入并注册

```
1 // main.js
2
3 // 导入组件
4 import { QuillEditor } from '@vueup/vue-quill'
5 // 导入样式
6 import '@vueup/vue-quill/dist/vue-quill.snow.css'
7
8 // 全局注册组件
9 app.component('QuillEditor', QuillEditor)
```

3. 使用组件

在 `ArticleOps.vue` 中

```
1 <QuillEditor
2   theme="snow"
3   toolbar="full"
4   placeholder="请输入文章内容"
```

```
/>
```

4. 样式美化

```
.el-form {
  .el-input {
    height: 36px;
  }
  :deep(.ql-container) {
    width: 100%;
    min-height: 180px;
  }
  .el-form-item:last-child {
    margin-top: 100px;
  }
}
```

二十七、文章操作组件区分操作类型

1. 分析

因为 新增、编辑、预览 共用一个组件, 后续会调用不同的接口, 为了清楚知道当前是何种操作, 进行父子通信

2. 父子通信

2.1 子接受

`ArticleOps.vue`

```
// 接受 props
const props = defineProps({
  // 当前操作类型
  opsType: {
    type: String,
    default:''
  },
  // 当前文章 id
  articleId: {
    type: String,
    default: ''
  }
})
```

2.2 父传递

`Article/index.vue`

```
<article-ops
   :opsType="currentOpsType"
   :articleId="articleId"
 />
```

二十八、预览文章

1. 分析

预览与新增、编辑不同, 展示另一套界面, 所以我们要新增一套预览的 HTML 结构

在 `ArticleOps.vue` 下

```
<template>
  <!-- 预览 -->
  <div class="preview">
    <h4> xxx </h4>
    <div> xxxxxxxxxx... </div>
  </div>

  <!-- 新增/编辑 -->
  <el-form label-width="55px">

  </el-form>
</template>
```

2. 条件控制渲染

```
 <template>
   <!-- 预览 -->
   <div class="preview" v-if="props.opsType === 'preview'">

   </div>

   <!-- 新增/编辑 -->
   <el-form label-width="55px" v-else>

```

```
    </el-form>
</template>
```

3. 封装常量模块

便于 `ARTICLE_OPS_TYPE` 复用和可维护性，建议将其抽离封装到常量模块

新建 `src/constant/article.js` 模块

```
// 文章操作类型
export const ARTICLE_OPS_TYPE = {
  add: 'add', // 新增
  edit: 'edit', // 编辑
  preview: 'preview' // 预览
}
```

4. 重构代码

在 `Article/index.vue` 和 `ArticleOps.vue` 下

```
// 1、导入文章操作常量
import { ARTICLE_OPS_TYPE } from '@/constant/article'

// 2、删除之前定义好的  ARTICLE_OPS_TYPE
```

`ArticleOps.vue` 修改判断条件

```
<!-- 预览结构 -->
  <div
    class="preview"
    v-if="props.opsType === ARTICLE_OPS_TYPE.preview">
    ...
  </div>
```

5. 定义获取文章详情接口

`api/article.js`

```
/**
 * 获取文章详情
```

```
 */
export const getArticleDetailApi = (articleId) => {
  return request({
    method: 'get',
    url: `/article/${articleId}`
  })
}
```

6. 请求并渲染文章详情

```
<script setup>
  // 导入详情接口
  import { getArticleDetailApi } from '@/api/article'

  // 当前文章表单对象
  const articleForm = reactive({
    id: '', // 文章 id
    stem: '', // 文章标题
    content: '' // 文章内容
  })

  getArticleDetail()

  // 获取文章详情
  async function getArticleDetail() {
    // 判断是否存在文章 id
    if (!props.articleId) return
    // 请求文章详情
    const detail = await getArticleDetailApi(props.articleId)
    // 给表单字段赋值
    Object.keys(articleForm).forEach((key) => {
      articleForm[key] = detail[key]
    })
  }
</script>
<template>
  <!-- 预览结构 -->
  <div
    class="preview"
    v-if="props.opsType === ARTICLE_OPS_TYPE.preview">
    <h4>{{ articleForm.stem }}</h4>
    <div v-html="articleForm.content"></div>
  </div>
</template>
```

7. 解决数据重复

在 `Article/index.vue` 下

`el-drawer` 默认不会每次创建一个新的组件, 而是会复用之前的组件, 这里就会导致数据重复

解决办法：给组件添加一个 `不重复的 key` , 这样因为每次key值不同, el-drawer 就会 `重新创建组件` , 不去复用之前的, 从而保证了每次数据的不重复

```
<ArticleOps
  :key="articleId"
/>
```

二十九、新增文章

1. 定义接口

```
/**
 * 新增文章
 */
export const postArticleApi = (articleForm) => {
  return request({
    method: 'post',
    url: '/article',
    data: articleForm
  })
}
```

2. 表单控件双向数据绑定

在 `ArticleOps.vue` 下

```
<template>
  <el-form label-width="55px">
    <el-form-item label="标题">
      <el-input v-model="articleForm.stem" />
    </el-form-item>
    <el-form-item label="内容">
      <QuillEditor
        contentType="html"
        v-model:content="articleForm.content"
      />
    </el-form-item>
```

```
  </el-form>
</template>
```

3. 发起新增请求

在 `ArticleOps.vue` 下

```
<script setup>
  import { postArticleApi } from '@/api/article'

  // 确认
  const onConfirm = async () => {
    if (props.articleId) {
      // 编辑

    } else {
      // 新增

      // 解构
      const { stem, content } = articleForm
      // 调接口
      await postArticleApi({
        stem,
        content
      })
    }
    // 成功提示
    ElMessage.success(props.articleId ? '编辑成功' : '新增成功')
    // 通过父组件（1.关闭抽屉 2.获取文章列表）
    emit('finish')
  }
</script>

<template>
  <el-button
    type="primary"
    @click="onConfirm"
    >确 认</el-button>
</template>
```

4. 父组件关闭抽屉+获取列表

```
<script setup>
```

```
    // 关闭抽屉
  const onClose = () => {
    visible.value = false
  }

  const onFinish = () => {
    // 关闭抽屉
    onClose()
    // 重新获取文章列表
    getArticeList()
  }
</script>
<template>
  <el-drawer>
    <!-- 文章操作组件 -->
    <ArticleOps @finish="onFinish" />
   </el-drawer>
</template>
```

三十、编辑文章

1. 定义接口

```
/**
 * 编辑文章
 */
export function putArticleApi(articleForm) {
  return request({
    method: 'put',
    url: `/article/${articleForm.id}`,
    data: articleForm
  })
}
```

2. 绑定点击事件并处理

在 `ArticleOps.vue` 下

```
<script setup>
  import { putArticleApi } from '@/api/article'

  import { ElMessage } from 'element-plus'

```

```
  // emit events
  const emit = defineEmits()

  // 确认
  const onConfirm = async () => {
    if (props.articleId) {
      // 编辑
      await putArticleApi(articleForm)

    } else {
      // 新增

      // ...
    }
}
</script>
```

三十一、取消和清空表单

1. 点击取消关闭抽屉

```
<script setup>
  // 取消
  const onCancel = () => {
    // 通知父组件：关闭抽屉组件
    emit('close')
  }
</script>

<el-button @click="onCancel">取 消</el-button>
```

2. 父组件关闭抽屉

```
<ArticleOps @close="onClose" />
```

3. 新增完毕清空表单

```
  // 确认
  const onConfirm = async () => {
    if (props.articleId) {
```

```
      // 编辑

    } else {
      // 新增

      // 解构
      const { stem, content } = articleForm
      // 调接口
      await postArticleApi({
        stem,
        content
      })

      // 清空表单
      Object.keys(articleForm).forEach((key) => {
        articleForm[key] = ''
      })

    }
    // ...
  }
```