项目梳理

2024.01-2024.04　　CMS-内容管理平台

- **项目内容：**CMS 平台旨在简化网站和数字内容的管理过程，可以对企业内部各类生产的内容进行管理，并且支持创建相关的流程进行内容控制。允许用户创建各种类型的内容，并提供编辑工具以便于修改和格式化内容。
- **技术栈：**Vue3 + Vite + Typescript + Pinia + Vue-Router
- **项目工作 ：**
  - 完成对平台功能的基本建设，能够支持对内容的过程管理，使用 vue-router 进行路由权限拦截，判断，路由懒加载。
  - 平台迭代过程中负责将 Pinia 对 Vuex 进行替代。
  - 使用 keep-alive 对整个页面进行缓存，支持多级嵌套页面。
  - 通过 echart 实现对内容流量数据和内容分析数据的可视化展示，负责主题化（黑暗模式）切换及持久化保持设置。

2023.01-2023.03　　基于 LSTM 和注意力机制的天气预测系统

- **项目介绍：**本系统利用 LSTM（长短期记忆网络）和注意力机制来预测天气。LSTM 用于处理时间序列数据，而注意力机制有助于模型聚焦于关键信息。通过结合这两种技术可以更准确地预测未来的天气情况，并根据用户的期望展示出相对应的天气数据。
- **技术栈：**Vue2+Vuex+Vue-router+Element-UI
- **项目工作 ：**
  - 使用深度学习框架 TensorFlow 来实现 LSTM 层。通过使用了多层 LSTM 来增强模型的表达能力。
  - 对 axios 进行了二次封装，并添加了请求拦截器，便于统一处理相关信息。
  - 采用组件化的开发方式，合理的将页面的模块或功能抽出对应的可复用组件。
  - 使用图片懒加载延迟加载页面上的图片，减少了页面加载时间和网络带宽消耗。

一般来说侧重问技术不怎么问业务

路由懒加载

```
const Home = () => import('@/components/Home');
const Login = () => import('@/components/Login');
const Dashboard = () => import('@/components/Dashboard');
const NotFound = () => import('@/components/NotFound');

const router = new Router({
  mode: 'history',
  routes: [
    {
      path: '/',
      name: 'Home',
      component: Home,
    },
    {
      path: '/login',
      name: 'Login',
      component: Login,
    },
```

```
    {
      path: '/dashboard',
      name: 'Dashboard',
      component: Dashboard,
      meta: { requiresAuth: true },
    },
    {
      path: '*',
      name: 'NotFound',
      component: NotFound,
    },
  ],
});
```

权限判断

可以从 Vuex 或本地存储（localstorage）中获取用户的认证状态，可以在路由守卫中添加更复杂的判断条件：

认证状态存储：通常将用户的认证状态存储在 Vuex 状态管理中或使用 `localStorage`、`sessionStorage`。

路由守卫：在 `router.beforeEach` 中进行认证检查

```
router.beforeEach((to, from, next) => {
  const isAuthenticated = /* 你的认证逻辑，例如从 Vuex 状态管理中获取 */;
  const userRole = /* 从 Vuex 或其他地方获取用户角色 */;

  if (to.matched.some(record => record.meta.requiresAuth)) {
    if (isAuthenticated) {
      if (to.meta.role && to.meta.role !== userRole) {
        next({ name: 'NotFound' }); // 或其他的拒绝访问处理
      } else {
        next();
      }
    } else {
      next({ name: 'Login' });
    }
  } else {
    next();
  }
});
```

基于用户角色的权限判断

基本概念： 不同角色的用户有不同的权限访问不同的路由。

实现步骤：

- **角色存储**：将用户的角色信息存储在 Vuex 状态管理中。
- **路由配置**：在路由元信息中添加角色信息。
- **路由守卫**：检查用户的角色是否符合路由要求。

```
const routes = [
  {
    path: '/admin',
    component: AdminComponent,
    meta: { requiresAuth: true, role: 'admin' }
  },
  // 其他路由
];

router.beforeEach((to, from, next) => {
  const isAuthenticated = store.getters.isAuthenticated;
  const userRole = store.getters.userRole;

  if (to.matched.some(record => record.meta.requiresAuth)) {
    if (isAuthenticated) {
      if (to.meta.role && to.meta.role !== userRole) {
        next({ name: 'NotFound' });
      } else {
        next();
      }
    } else {
      next({ name: 'Login' });
    }
  } else {
    next();
  }
});
```

pinia对比vuex有哪些优点 为什么要用pinia替换vuex

首先Pinia 是一个针对 Vue 3 的状态管理库，相比于vuex更适合vue3，然后pinia是开箱即用 学习成本比较低

更简洁的 API

- **Pinia**：提供了更简洁且易于理解的 API。状态管理的定义和使用更加直观，无需使用复杂的概念，如 mutations 和 actions。

- **Vuex**：需要使用更复杂的结构，包括 mutations、actions 和 getters，语法较为冗长。

对 Composition API 的支持

- **Pinia**：与 Vue 3 的 Composition API 设计原则紧密集成。它允许你使用 `defineStore` 创建和管理状态，风格更符合 Vue 3 的整体设计。
- **Vuex**：虽然也支持 Vue 3，但它的设计和 API 更加接近 Vue 2 的风格，不够完全符合 Composition API 的风格。

更好的 TypeScript 支持

- **Pinia**：内置了更好的 TypeScript 支持，类型推导更加完善。通过 `defineStore`，你可以轻松地利用 TypeScript 的类型检查和智能提示功能。
- **Vuex**：虽然也支持 TypeScript，但配置和使用上会相对复杂，需要更多的类型声明和手动配置。

状态持久化和管理

- **Pinia**：提供了内置的插件机制，可以很容易地与第三方库集成，如状态持久化。
- **Vuex**：需要额外的插件或手动实现状态持久化功能，配置上可能会更加复杂。

状态持久化的核心概念

1. **持久存储**：
   - **浏览器存储**：包括 `localStorage` 和 `sessionStorage`，用于在浏览器中保存数据。`localStorage` 数据在浏览器关闭后仍然存在，而 `sessionStorage` 数据在会话结束时（如关闭浏览器标签页）被清除。
   - **IndexedDB**：是一种在浏览器中存储大量结构化数据的机制，适用于更复杂的存储需求。
   - **Cookies**：用于存储少量数据，并且通常用于存储会话信息，如用户认证信息。
2. **持久化策略**：
   - **自动持久化**：在应用运行时自动将状态保存到持久存储中，并在加载时恢复状态。
   - **手动持久化**：开发者手动实现持久化逻辑，比如在状态变化时手动更新持久存储，并在应用初始化时从存储中加载状态。

Pinia 的持久化

Pinia 提供了简化状态持久化的机制，通常通过插件实现。这使得集成持久化变得更加直接和便捷。Pinia 的持久化插件会在状态发生变化时自动将状态保存到指定的存储中（如 `localStorage`），并在初始化时恢复状态。

vuex是只能使用第三方插件比如说 `vuex-persistedstate` 等，或者手动实现

keep-alive

使用 `<keep-alive>` 对整个页面进行缓存

1. 基本用法

要缓存整个页面，你可以将 `<keep-alive>` 包裹在 `<router-view>` 外部。这样，`<router-view>` 中显示的组件就会被缓存。

```
<template>
  <div id="app">
    <keep-alive>
      <router-view />
    </keep-alive>
  </div>
</template>

<script>
export default {
  name: 'App',
};
</script>
```

2. 配置 `<keep-alive>`

`<keep-alive>` 提供了两个主要的属性来配置缓存的组件：

- `include`：一个字符串或正则表达式，指定要缓存的组件名称（支持多个，用 `|` 分隔）。
- `exclude`：一个字符串或正则表达式，指定不缓存的组件名称（支持多个，用 `|` 分隔）。

```
<template>
  <div id="app">
    <keep-alive include="Home, About">
      <router-view />
    </keep-alive>
  </div>
</template>

<script>
export default {
  name: 'App',
};
</script>
```

`Home` 和 `About` 的组件会被缓存，其他组件将不会被缓存

支持多级嵌套页面

`<keep-alive>` 支持多级嵌套缓存。当你在 `<router-view>` 中使用多个子路由时，它会递归地缓存子组件。这意味着，嵌套的页面和子页面也会被缓存。

```
const routes = [
  {
    path: '/parent',
    component: ParentComponent,
    children: [
      {
        path: 'child1',
        component: Child1Component
      },
      {
        path: 'child2',
        component: Child2Component
      }
    ]
  }
];

<template>
  <div id="app">
    <keep-alive>
      <router-view />
    </keep-alive>
  </div>
</template>

<script>
export default {
  name: 'App',
};
</script>
```

`ParentComponent` 及其子组件 `Child1Component` 和 `Child2Component` 的状态都会被缓存

注意事项

1. **缓存限制**：`<keep-alive>` 的缓存策略基于组件名称，组件的状态会被缓存，但组件的数据和方法不会被缓存。如果你需要缓存更复杂的状态，可能需要手动处理组件状态的保存和恢复。
2. **内存使用**：缓存大量组件可能会导致内存使用增加。确保只缓存必要的组件，并定期检查应用的性能和内存使用情况。

3. **使用场景**：`<keep-alive>` 适合用于需要频繁切换的页面，尤其是在有复杂状态的组件中。对于简单组件或不需要保留状态的组件，可能不需要使用缓存。

主题切换，持久化

持久化这个看一下本地存储去存

然后切换就看elementplus的文档写的https://element-plus.org/zh-CN/guide/theming.html

组件化

将页面的不同模块和功能抽象成可复用的组件，从而提高代码的可维护性、可复用性和可测试性。

组件化的基础概念

组件化是指将一个复杂的界面拆分成多个小的、独立的组件，每个组件负责界面的一部分或某个功能。组件之间通过 props 和 events 进行交互。

组件的基本结构

每个 Vue 组件通常包含以下几个部分：

- **模板 (`template`)**：定义组件的 HTML 结构。
- **脚本 (`script`)**：定义组件的逻辑和数据。
- **样式 (`style`)**：定义组件的样式。

识别功能模块

首先，识别出页面中的不同功能模块。例如，在一个用户仪表盘页面中，可能包含以下模块：

- **用户信息**：显示用户的基本信息和头像。
- **消息列表**：显示用户的消息。
- **统计数据**：展示用户的统计信息。
- **设置面板**：允许用户修改设置。

创建可复用组件

将这些功能模块抽象成独立的 Vue 组件。每个组件应具备以下特性：

- **单一职责**：每个组件应有一个明确的功能。
- **可复用性**：组件应该可以在不同的地方重用。
- **自包含性**：组件应包含自己的模板、逻辑和样式。

UserInfo.vue 显示用户信息

```
1 <template>
2   <div class="user-info">
3     <img :src="user.avatar" alt="Avatar" />
```

```vue
    <h2>{{ user.name }}</h2>
  </div>
</template>

<script setup>
import { defineProps } from 'vue';

const props = defineProps({
  user: {
    type: Object,
    required: true
  }
});
</script>

<style scoped>
.user-info {
  display: flex;
  align-items: center;
}
.user-info img {
  border-radius: 50%;
  margin-right: 10px;
}
</style>
```

MessageList.vue - 显示消息列表

```vue
<template>
  <ul class="message-list">
    <li v-for="message in messages" :key="message.id">
      {{ message.text }}
    </li>
  </ul>
</template>

<script setup>
import { defineProps } from 'vue';

const props = defineProps({
  messages: {
    type: Array,
    required: true
  }
});
```

```
</script>

<style scoped>
.message-list {
  list-style: none;
  padding: 0;
}
.message-list li {
  padding: 10px;
  border-bottom: 1px solid #eee;
}
</style>
```

组件化的最佳实践

1. **单一职责**：每个组件应负责一个功能或一个界面部分。
2. **可复用性**：设计组件时要考虑其可复用性，避免硬编码数据。
3. **自包含性**：组件应尽可能自包含，包含自己的样式和逻辑。
4. **组件交互**：通过 props 传递数据，通过 events 发送消息，确保组件间的交互清晰明确。
5. **目录结构**：合理组织组件的目录结构，通常会按照功能或模块组织组件。

图片懒加载通常怎么实现

使用原生 HTML 属性

现代浏览器支持 `loading` 属性，可以很简单地实现图片懒加载。

示例：

```
<img src="path/to/image.jpg" alt="Description" loading="lazy" />
```

`loading="lazy"` 告诉浏览器仅在图片即将进入视口时才加载。

使用 Intersection Observer API

Intersection Observer API 是一个强大的 API，可以更灵活地实现懒加载。这种方法适用于需要支持旧版浏览器或有更复杂需求的场景。

步骤：

HTML 结构 lazyman

使用 `data-src` 或类似的自定义属性来存储图片的实际 URL，初始 `src` 可以设置为占位图像或为空。

```
1 html
2 复制代码
3 <img data-src="path/to/image.jpg" alt="Description" class="lazy-load" />
```

JavaScript 实现

使用 Intersection Observer API 来监听图片何时进入视口，然后更新 `src` 属性来加载实际图片。

```
1 javascript
2 复制代码
3 document.addEventListener('DOMContentLoaded', () => {const lazyImages =
  document.querySelectorAll('img.lazy-load');
4 const lazyLoad = (entries, observer) => {
5     entries.forEach(entry => {if (entry.isIntersecting) {const img =
  entry.target;
6         img.src = img.dataset.src;
7         img.classList.remove('lazy-load');
8         observer.unobserve(img);
9       }
10     });
11   };
12 const observer = new IntersectionObserver(lazyLoad, {root: null,rootMargin:
  '0px',threshold: 0.1
13   });
14   lazyImages.forEach(image => {
15     observer.observe(image);
16   });
17 });
```

CSS

使用 CSS 来确保懒加载时的占位图像样式正确。

```
1 css
2 复制代码
3 .lazy-load {opacity: 0;transition: opacity 0.3s;
4 }
5 .lazy-load[src] {opacity: 1;
6 }
```

使用 JavaScript 库

许多 JavaScript 库和框架提供了懒加载的功能，简化了实现过程。例如：

- **lazysizes**: 一个功能强大且轻量的懒加载库，支持图片和 iframe 的懒加载。
- **lozad.js**: 一个简单的懒加载库，易于使用和配置。

使用 `lazysizes` 的示例：

引入 lazysizes

```
1 html
2 <script
  src="https://cdnjs.cloudflare.com/ajax/libs/lazysizes/5.3.0/lazysizes.min.js"
  async></script>
```

HTML 结构

使用 `data-src` 替代 `src` 属性，并添加 `lazyload` 类。

```
1 html
2 <img data-src="path/to/image.jpg" class="lazyload" alt="Description" />
```

CSS

```
1 css
2 .lazyload {opacity: 0;transition: opacity 0.3s;
3 }
4 .lazyloaded {opacity: 1;
5 }
```

使用 Vue.js 实现图片懒加载

在 Vue.js 中，可以使用自定义指令或第三方插件来实现图片懒加载。例如，`vue-lazyload` 是一个常用的 Vue 插件。

使用 `vue-lazyload` 的示例：

安装插件

```
1 npm install vue-lazyload
```

在 Vue 应用中配置

```
import Vue from 'vue';
import VueLazyload from 'vue-lazyload';

Vue.use(VueLazyload, {preLoad: 1.3,error: 'path/to/error.jpg',loading:
'path/to/loading.jpg',attempt: 1
});
```

使用懒加载

```
<template>
  <img v-lazy="imageSrc" alt="Description" />
</template>
<script setup>
import { ref } from 'vue';
const imageSrc = ref('path/to/image.jpg');
</script>
```