简历项目

实习经历

为什么要引入Prettier和ESLint?

引入Prettier和ESLint是为了提升**代码的质量和可读性。Prettier可以统一代码的格式**，避免团队成员之间的代码风格不一致，而**ESLint则可以帮助我们发现和修复潜在的代码问题**，遵循最佳实践，确保代码的健壮性和稳定性。

你为什么决定将项目从JavaScript重构为TypeScript

重构为**TypeScript的主要原因是为了增强代码的类型标注**，这样可以在**编译时发现潜在的类型错误，减少 运行时错误的可能性。**同时，TypeScript的类型系统可以提高代码的可读性和可维护性，使得团队成员在合作时更容易理解和使用代码。

TypeScript是由微软开发的一种编程语言，是JavaScript的超集。它在JavaScript的基础上增加了静态类型和其他一些功能。TypeScript的主要特点包括：

1. **静态类型**： 允许开发者在编写代码时定义变量的类型，这有助于在编译阶段捕获错误。
2. **编译型语言**： TypeScript代码需要编译成JavaScript代码才能在浏览器中运行。
3. **丰富的工具支持**： 由于类型信息的存在，TypeScript可以提供更好的代码补全、重构和导航等开发工具支持。
4. **向下兼容**： 任何合法的JavaScript代码也是合法的TypeScript代码。

主要区别

1. **类型系统**： TypeScript有静态类型检查，JavaScript没有。这意味着在使用TypeScript时，开发者可以在编写代码时捕获更多的错误。
2. **编译过程**： TypeScript需要编译成JavaScript才能运行，而JavaScript是解释执行的。
3. **开发工具支持**： TypeScript由于类型信息的存在，能提供更强大的代码编辑和调试支持。

TypeScript通过增加静态类型和其他一些特性，增强了JavaScript的可靠性和可维护性。对于大型项目或团队协作，TypeScript可以帮助减少错误，并提高开发效率。JavaScript则更灵活，适合快速开发和小型项目。选择哪种语言主要取决于项目的需求和团队的偏好。

ferry工单系统

在Ferry工单中，我主要负责撰写和管理我的工单，并且封装了可复用的表格组件。这些组件可以简化表单的编写和管理，提高了工作效率。此外，我还确保这些组件具有良好的用户体验和可维护性，方便团队其他成员使用和扩展。

例如封装了table组件，弹窗组件。

table组件的封装

1. 先想好封装这个table的一些什么功能，需要一个分页更能，展示数据，多选功能
2. 所以定义props由父组件传递进来，首先就是colums，数据，是否显示分页，一页显示多少。
3. 接着就是定义emits来子组件传递数据给父组件
4. 接着table内容部分我才用插槽来传递进来，先将这个table里面的colum、record、index通过插槽的方式传递给父组件来获取这个table的详细信息

UI界面的优化

按照设计图重新将页面进行了布局。

为什么不留在这个公司了

1. 我在目前的公司学到了很多，也很感谢这个平台给我的成长机会。然而，我感觉已经达到了一个瓶颈，希望能够寻找到新的挑战和成长机会，进一步提升自己的技术和职业素养。
2. 我对自己的职业发展方向有了新的规划，希望能够在新的环境中接触到更多的技术和项目，丰富我的经验，并为公司带来更多的价值。
3. 由于家庭原因或个人生活安排，我需要在一个新的地点工作。这个决定更多是基于个人生活和家庭的考虑。

Merikle网盘项目

介绍一下你这个项目

这个网盘项目是一个文件存储和共享平台，用户可以上传，分享文件，预览图片，协作文档功能。我在里面进行前后端开发，前端使用的是vue框架，后端使用的nestjs来实现的。

你负责的部分

1. 页面的设计
2. 环境的搭建
3. 大文件上传和断点续传
4. 文件分享
5. 文件预览:支持word、ppt、excel、图片、视频预览
6. 文档协同
7. Excel协同

封装了哪些组件

table组件

1. 先想好封装这个table的一些什么功能，需要一个分页功能，展示数据，多选功能
2. 所以定义props由父组件传递进来，首先就是colums，数据，是否显示分页，一页显示多少。
3. 接着就是定义emits来子组件传递数据给父组件，rowSelection是在父组件里面设置的事件

4. 接着table内容部分我才用插槽来传递进来，先将这个table里面的colum、record、index通过插槽的方式传递给父组件来获取这个table的详细信息

封装了导航组件

首先这个导航组件不仅仅是根据页面的路由来进行切换的，有文件保存到哪个文件夹里会弹出窗口来让你选择，所以这部分就不需要监听路由了。

1. 传入的props是：是否监听路由
2. 接着就是打开文件夹的时候，设置路径的时候就需要根据传入的是否监听路由来进行区分，如果不监听路由的话就直接进行导航页的改变这个方法是定义在父组件里的，如果监听路由的话就要拼接路由字符串来改变路由了。
3. 还在这个组件定义了返回上一级的方法。

弹窗组件

倒计时组件

预览组件

分析一下需求：

需要展示一个弹窗，里面有着预览的pdf、video、excel等类型的文件，这些预览的pdf、video、excel等类型的文件又得拆成组件，因为每一个文件所展示需要的库是不相同的。

大文件上传

思路

1. 每个文件要有自己唯一的标识，因此在进行分片上传前，需要对整个文件进行MD5加密，生成MD5码，在后面上传文件每次调用接口时以formData格式上传给后端。可以使用spark-md5 计算文件的内容hash，以此来确定文件的唯一性将文件hash发送到服务端进行查询。以此来确定该文件在服务端的存储情况，这里可以分为三种：未上传、已上传、上传部分。
2. 根据服务端返回的状态执行不同的上传策略。已上传：执行秒传策略，即快速上传，实际上没有对该文件进行上传，因为服务端已经有这份文件了。未上传、上传部分：执行计算待上传分块的策略并发上传还未上传的文件分块。当传完最后一个文件分块时，向服务端发送合并的指令，即完成整个大文件的分块合并，实现在服务端的存储。

上传过程：

1. 分割文件：将要上传的文件切割成多个小文件片段。主要使用JavaScript的File API中的slice方法来实现。
2. 上传文件分片：使用XMLHttpRequest或者Fetch API将分片信息以formData格式，并携带相关信息，如文件名、文件ID、当前片段序号等参数传给分片接口。
3. 后端接收并保存文件片段：后端接收到每个文件片段后，将其保存在临时位置，并记录文件片段的序号、文件ID和文件MD5 hash值等信息。
4. 续传处理：如果上传过程中断，下次继续上传时，通过查询后端已保存的文件片段信息，得知需要上传的文件片段，从断点处继续上传剩余的文件片段。
5. 合并文件：当所有文件片段都上传完成后，后端根据文件ID将所有片段合并成完整的文件。

并发上传

并发上传相对要优雅一下，将文件分割成小片段后，使用Promise.all()把所有请求都放到一个Promise.all里，它会自动判断所有请求都完成然后触发 resolve 方法。并发上传可以同时上传多个片段而不是依次上传，进一步提高效率。

实现思路：

1. 使用slice方法对二进制文件进行拆分，并把拆分的片段放到chunkList里面。
2. 使用map将chunkList里面的每个chunk映射到一个Promise上传方法。
3. 把所有请求都放到一个Promise.all里，它会自动判断所有请求都完成然后触发 resolve 方法，上传成功后通知后端合并分片文件。

```
const eleFile = document.getElementById('file');
eleFile.addEventListener('change', (event) => {
    const file = event.target.files[0];
    // 上传分块大小，单位Mb
    const chunkSize = 1024 * 1024 * 1;
    // 当前已执行分片数位置
    let currentPosition = 0;
    // 存储文件的分片
    let chunkList = [];
    //初始化分片方法，兼容问题
    let blobSlice = File.prototype.slice || File.prototype.mozSlice ||
File.prototype.webkitSlice;
    while(currentPosition < file.size) {
        const chunk = blobSlice.call(file, currentPosition, currentPosition +
chunkSize);
        chunkList.push(chunk);
        currentPosition += chunkSize;
    }
    uploadChunk(chunkList, file.name)
})


function uploadChunk(chunkList, fileName) {
    const uploadPromiseList = chunkList.map((chunk, index) => {
        // 将分片信息以formData格式作为参数传给分片接口
        let formData = new FormData();
        formData.append('fileChunk', chunk);
        // 可以根据实际的需要添加其它参数，比如切片的索引
        formData.append('index', index);
        // 根据项目实际情况
        return axios.post(
            '/api/oss/upload/file',
            formData,
            {
                headers: { 'Content-Type': 'multipart/form-data' },
                timeout: 600000,
            }
        )
```

```
    })


    Promise.all(uploadPromiseList).then(res => {
        // 上传成功并通知后端合并分片文件
        axios.post(
            '/api/oss/file/merge',
            {
                message: fileName
            },
            {
                headers: { 'Content-Type': 'application/json' },
                timeout: 600000,
            }
        ).then(data => {
            console.log('文件合并成功', data)
        })
    }).catch(error => {
        // 上传错误
        console.log('上传失败', error)
    })
}
```

如何实现大文件上传的

【WEB前端开发中如何实现大文件上传？ - CSDN App】http://t.csdnimg.cn/hyXUf

可以看这里面的介绍的很清楚。

我使用了分片上传的技术来实现大文件上传。文件被分成多个小片段（chunks）,例如我会定义一个固定的值1024*1024来根据这个量来进行切割，每个片段单独上传到服务器。这样即使上传过程中出现网络问题，也可以从失败的片段继续上传，确保上传的稳定性和可靠性。在客户端，我使用 Axios 进行 HTTP 请求，在服务器端使用 Node.js 处理分片并合并文件，使用 MongoDB 存储文件的元数据。

//秒传

我是做了一个秒传的功能的，所以使用到了md5，将文件生成md5之后传递给后端，后端再在数据库或者哪里进行查找是否有于其对应的md5有的话直接进行秒传了。在这里我封装一个判断文件状态的函数，来判断这个文件在服务器的数据库中是怎样的形式存储，返回需要上传的文件片段的序列。例如如果返回的长度为0说明就可以秒传上去了，设置上传的进度以及文件上传的状态，否则的话就获取到这个返回来文件切片序列的第一个作为开始切片的数，和返回切片的长度作为终止条件，来继续上传。在切片传输的时候如果用户点击了停止按钮，就会去停止，否则继续循环切片上传。等到最后一次上传的时候就调用合并的接口。

断点续传

1. **记录上传进度**：

   客户端记录和服务端都要进行记录一下进度

   - 在上传过程中，客户端需要记录每个块的上传状态。可以使用本地存储（如浏览器的 LocalStorage）来保存已上传的块信息。
   - 服务器也需要记录每个文件的上传进度，可以使用数据库来存储每个文件的块上传状态。

2. **上传恢复**：

   当中断了上传之后，客户端重新上传，需要检查哪些块已经上传成功了，所以只要上传未完成的块。

   - 当上传中断后，客户端重新开始上传时，需要先检查哪些块已经上传成功，然后只上传未完成的块。
   - 服务器端提供一个接口来查询文件的上传状态，客户端可以调用这个接口来获取已上传的块列表。

3. **支持断点续传的接口设计**：

   - `/upload_chunk`：上传文件块，参数包括文件ID、块序号、块的内容等。
   - `/merge_chunks`：通知服务器合并文件块，参数包括文件ID。
   - `/upload_status`：查询文件的上传状态，参数包括文件ID，返回已上传的块列表。

你是如何处理多用户协同编辑的冲突问题的

文档上的协同编辑：我使用了quill编辑器和y-quill和y-websocket来实现协同编辑。这些工具基于 CRDT（Conflict-free Replicated Data Types）模型，能够自动处理并合并冲突的编辑操作，确保所有用户的编辑内容最终一致。具体实现过程中，我会监听编辑器的变化事件，并通过 WebSocket 将变化同步到服务器，再由服务器广播给其他客户端。

过程概述
1.初始化 Yjs 文档 ：在客户端创建一个 Yjs 文档 (Y.Doc)，这是一个用来存储协同编辑数据的地方。
2.设置 WebSocket 提供者 ：使用 y-websocket 创建一个 WebSocket 提供者 (WebsocketProvider)，它负责在客户端和服务器之间同步 Yjs 文档的状态。
3.初始化 Quill 编辑器:创建一个 Quill 编辑器实例，用于在前端展示和编辑文本内容。
4.绑定 Quill 和 Yjs:
使用 y-quill 的 QuillBinding 将 Quill 编辑器和 Yjs 文档绑定在一起，这样 Quill 的编辑操作会自动更新 Yjs 文档，反之亦然。

```
import * as Y from 'yjs';
import { WebsocketProvider } from 'y-websocket';
import Quill from 'quill';
import { QuillBinding } from 'y-quill';

const ydoc = new Y.Doc();
const provider = new WebsocketProvider('ws://localhost:1234', 'quill-demo', ydoc);
const quill = new Quill('#editor', {
    theme: 'snow'
});

const ytext = ydoc.getText('quill');
```

```
const binding = new QuillBinding(ytext, quill, provider.awareness);
```

实现永久存储

1. **在服务器端持久化数据**：
    - 监听 **Yjs 文档的更新事件，将更新的数据保存到数据库或文件系统中。**
    - 在**客户端连接时，从持久化存储中加载 Yjs 文档的初始状态，并将其发送给客户端。**
2. **在客户端初始化时加载数据**：
    - 当客户端连接到服务器时，接收初始的文档状态，并使用这些数据初始化 Yjs 文档。

```
const http = require('http');
const WebSocket = require('ws');
const Y = require('yjs');
const { WebsocketProvider } = require('y-websocket');
const { MongoClient } = require('mongodb');

const server = http.createServer();
const wss = new WebSocket.Server({ server });

const mongoUrl = 'mongodb://localhost:27017';
const client = new MongoClient(mongoUrl, { useNewUrlParser: true,
useUnifiedTopology: true });

let db, ydoc;

client.connect().then(() => {
    db = client.db('collaborative-editor');
    return db.collection('documents').findOne({ name: 'quill-demo' });
}).then(doc => {
    ydoc = new Y.Doc();
    if (doc) {
        Y.applyUpdate(ydoc, doc.data);
    }

    wss.on('connection', (ws) => {
        const provider = new WebsocketProvider('ws://localhost:1234', 'quill-demo',
ydoc);

        ydoc.on('update', (update) => {
            db.collection('documents').updateOne(
                { name: 'quill-demo' },
                { $set: { data: Y.encodeStateAsUpdate(ydoc) } },
                { upsert: true }
            );
        });

        ws.on('message', (message) => {
            provider.awareness.setLocalStateField('user', JSON.parse(message));
        });
    });
```

```
});

server.listen(1234, () => {
    console.log('Server is listening on port 1234');
});
```

如何实现文件预览

文件预览功能通过集成第三方库实现

```
exceljs、jspdf、dplayer、docx-preview
```

项目中遇到的最大挑战是什么，如何解决的?

回答： 项目中遇到的最大挑战是实现多用户实时协同编辑时的同步和冲突处理问题。为了解决这个问题，我深入研究了 CRDT 模型，选择了 y-quill 和 y-websocket 进行实现。这些工具能够自动处理冲突并保持文档的一致性。同时，我在实际应用中不断测试和优化，确保在不同网络环境下的稳定性和性能。

这个项目有什么难点

文件上传和下载以及文件共享这一方面比较难实现。

文件上传采用的是大文件切片上传，秒传，来提高上传速度，并且在传输的过程还是用了加密算法对文件进行加密，确保数据在传输和存储过程中的安全性。

大文件上传前端实现:

```
    1. 设置一个切片的初始量 ，const chunkSize = 1024 * 1024; // 1MB
    2. 进行切片，向上取整
    3. 接着就是循环进行切片上传了，在上传的时候先去检查文件上传的情况，如果已经有这个文件就上传下一个，否则继续上传
    4. 最后的时候进行合并切片就可以了。
```

后端实现:

1.检验切片状态的接口:返回没有上传的文件

```
2. 接收到了这个切片:接受切片并存储下来
2. 当切片上传完毕之后进行合并位完整的文件
```

其中这个共享文档的是实现:

使用了quill编辑器和y-websocket来进行过

3. 其中这个文件上传怎么进行处理和文件下载如何处理的

其中大文件上传:前端将大文件拆分成小块进行上传。客户端使用Axios发送每一块数据，后端接收到所有块后再进行合并，并保存到服务器指定的文件夹。

下载:

直接使用的来将这个文件的url放入里面来进行下载。

4. 文件预览

使用了一些现有的库，例如pdf预览使用的是pdf.js

5. 协作文档的功能如何实现

为了实现文档的实时协同编辑，我们使用了Quill编辑器作为基础的文本编辑工具，结合了y-quill和y-websocket来实现实时同步。具体步骤如下：

1. **Quill编辑器：** Quill是一个强大的、易于扩展的富文本编辑器，支持多种格式的文本编辑。我们选择它作为基础编辑器，提供给用户一个直观的编辑界面。
2. **y-quill和yjs：** y-quill是基于yjs的协同编辑解决方案。yjs是一个高效的CRDT（Conflict-free Replicated Data Type）库，专为协同编辑设计。通过yjs，我们可以在多个用户之间同步编辑内容，而无需担心数据冲突。
3. **WebSocket通信：** 为了实现实时通信，我们使用了WebSocket。每个打开文档的用户会通过WebSocket连接到服务器，并加入一个对应文档的房间。当用户进行编辑时，编辑内容会通过y-quill转换为操作，然后通过WebSocket发送到服务器。
4. **服务器端处理：** 服务器使用Node.js和WebSocket库处理来自客户端的操作。服务器将这些操作广播给所有连接到相同文档房间的其他客户端，确保每个用户都能实时看到其他用户的编辑内容。
5. **前端同步：** 收到来自服务器的编辑操作后，前端通过y-quill将这些操作应用到本地Quill编辑器，实现实时同步。

通过这种方式，我们能够确保文档在多个用户之间实时同步，无论谁进行编辑，所有人都能立即看到更新。

协作Excel怎么实现的

应聘者：我们使用了LuckySheet作为Excel的编辑器，并通过WebSocket实现前后端通信，确保每个用户的操作能够实时同步到其他用户的界面上。我们还解决了不同用户操作同一个Excel文档时显示相应光标的问题。具体实现步骤如下：

1. **LuckySheet：** LuckySheet是一个功能强大的在线表格编辑器，类似于Excel，支持多种复杂的表格操作和公式计算。我们选择它作为我们的Excel编辑器，为用户提供一个熟悉且功能强大的表格编辑环境。
2. **WebSocket通信：** 每个用户在打开Excel文档时，会通过WebSocket连接到服务器，并加入一个对应表格文档的房间。当用户对表格进行编辑（例如修改单元格内容、调整格式、添加公式）时，这些操作会被实时捕捉并通过WebSocket发送到服务器。
3. **操作序列化：** 用户的每次操作（例如修改单元格、插入行列、应用公式）被序列化为操作对象，通过WebSocket发送到服务器。
4. **服务器端处理：** 服务器端使用Node.js和WebSocket库接收这些操作对象，并将其广播给所有连接到相同文档房间的其他客户端。每个操作都附带有用户的唯一标识符，以便区分不同用户的操作。
5. **前端同步：** 前端接收到服务器广播的操作对象后，通过LuckySheet的API将这些操作应用到本地表格视图中，实现实时同步。
6. **光标显示：** 为了显示不同用户的光标，我们在操作对象中添加光标位置的信息。当用户移动光标或选择单元格时，客户端会发送包含光标位置信息的操作对象到服务器。

- 使用的是websocke技术，接收到用户传递的信息，客户端传递来了信息，后端给这个客户端发送一些信息，给这个客户端进行一下表示，加上wid，fileId，用户名字来在前端显示当前是谁在这个单元格里面书写内容。
- 接着就是更新数据和用户图标根据官方文档的解释，2是更新数据，3是处理用户的光标的的，当用户停止了书写内容的时候根据传递来的'mv'值的表示来切换用户图标还是更新数据

食之韵项目

一个react native项目

在实现的过程中有着

其中React 和Native是怎样通信的

1. **桥接机制（Bridge）**：
   React Native 的核心是一个桥接机制，它允许 JavaScript 代码和原生代码（如 Java 或 Objective-C/Swift）之间进行通信。这个桥接机制是异步的，意味着 JavaScript 和原生代码运行在各自独立的线程上，通过桥接来交换消息和数据。
2. **JavaScriptCore**：
   在 iOS 上，React Native 使用 JavaScriptCore 来运行 JavaScript 代码。在 Android 上，React Native 使用 V8 或 Hermes 引擎。JavaScript 代码通过这些引擎执行，并通过桥接机制与原生代码进行通信。
3. **Native Modules**：
   React Native 允许开发者创建自定义的原生模块，这些模块可以在 JavaScript 中调用。通过这种方式，开发者可以利用平台特定的功能，如访问设备硬件、操作系统服务等。

```
import { NativeModules } from 'react-native';
const { MyNativeModule } = NativeModules;

MyNativeModule.doSomething();
```

4. **RCTEventEmitter**：
   这是一个用于从原生代码向 JavaScript 发送事件的类。通过这个机制，原生代码可以通知 JavaScript 发生了某些事件。

```
// Objective-C 示例
#import <React/RCTEventEmitter.h>

@interface MyNativeModule : RCTEventEmitter <RCTBridgeModule>
@end

@implementation MyNativeModule

RCT_EXPORT_MODULE();

- (void)someNativeMethod {
  [self sendEventWithName:@"EventName" body:@{@"key": @"value"}];
}

@end
```

组件库项目

Message组件的实现

渲染虚拟节点的两种方式

1.调用函数传入对象参数来渲染虚拟节点

2. 根据type来进行渲染不同主题

3. 这个elMessage导出是一个对象，里面有着四个type类型来进行判断主题

多个message进行展示

全局挂载

message的测试

Tree组件的实现思想

预约挂号平台

支付功能

1. 选择支付服务提供商

首先，你需要选择一个支付服务提供商（PSP，如Stripe、PayPal、Square等），因为他们提供了安全的API和SDK，简化了支付处理。

2. 获取API密钥

你需要在支付服务提供商的开发者平台上注册并获取API密钥。这些密钥将用于认证和授权支付请求。

3. 安全性考虑

确保在前端收集和传输支付信息时遵循安全标准，如PCI-DSS。尽量避免直接处理和存储敏感信息，如信用卡号。

4. 集成支付SDK

大多数支付服务提供商提供了前端SDK，可以帮助你快速集成支付功能。

5. 创建支付表单

使用支付服务提供商的SDK创建一个支付表单，用于收集用户的支付信息。一般来说，支付表单包括信用卡号、有效期、安全码等信息。

8. 处理支付结果

根据支付结果（成功或失败），在前端给予用户相应的反馈，并执行后续操作（如订单确认、显示支付成功页面等）。

9. 测试

在上线之前，确保对支付流程进行全面的测试，使用支付服务提供商提供的测试卡和测试环境。

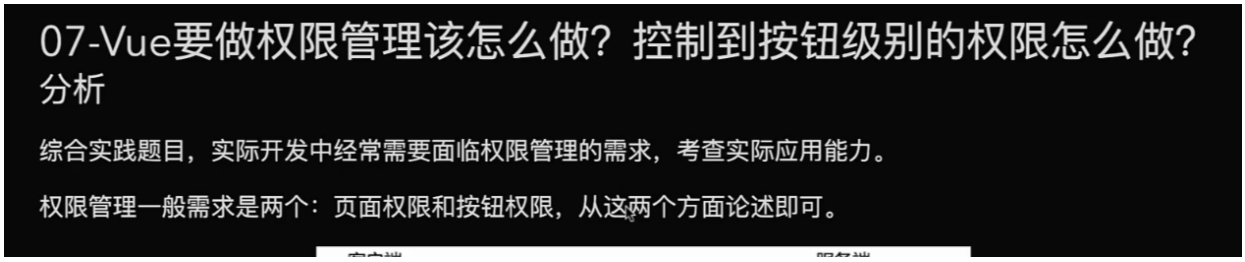
10. 上线

在确保支付流程安全和稳定后，将支付功能上线，并监控支付系统的运行情况。

通过这些步骤，你可以在前端实现一个功能完整的支付系统，同时确保支付信息的安全和用户体验的流畅。

新闻管理系统

权限管理



回答范例

1. 权限管理一般需求是**页面权限**和**按钮权限**的管理

2. 具体实现的时候分后端和前端两种方案：

   前端方案会**把所有路由信息在前端配置**，通过路由守卫要求用户登录，用户**登录后根据角色过滤出路由表**。比如我会配置一个`asyncRoutes`数组，需要认证的页面在其路由的`meta`中添加一个`roles`字段，等获取用户角色之后取两者的交集，若结果不为空则说明可以访问。此过滤过程结束，剩下的路由就是该用户能访问的页面，**最后通过**`router.addRoutes(accessRoutes)`**方式动态添加路由**即可。

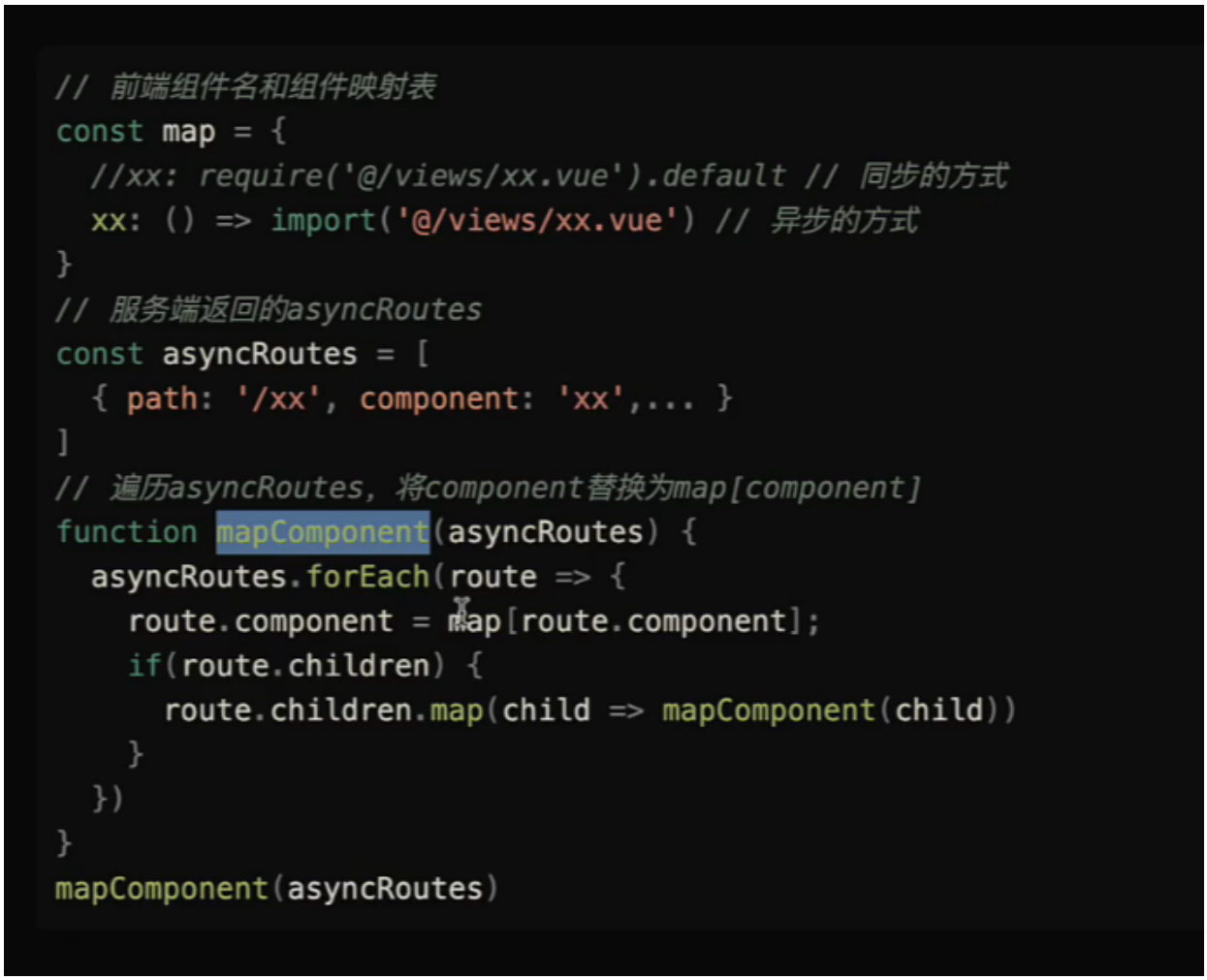
   后端方案会**把所有页面路由信息存在数据库**中，用户登录的时候根据其角色**查询得到其能访问的所有页面路由信息**返回给前端，前端**再通过**`addRoutes`**动态添加路由**信息

   按钮权限的控制通常会**实现一个指令**，例如`v-permission`，**将按钮要求角色通过值传给v-permission指令**，在指令的`moutned`钩子中可以**判断当前用户角色和按钮是否存在交集**，有则保留按钮，无则移除按钮。

3. 纯前端方案的优点是实现简单，不需要额外权限管理页面，但是维护起来问题比较大，有新的页面和角色需求就要修改前端代码重新打包部署；服务端方案就不存在这个问题，通过专门的角色和权限管理页面，配置页面和按钮权限信息到数据库，应用每次登陆时获取的都是最新的路由信息，可谓一劳永逸！

1. 类似`Tabs`这类组件能不能使用`v-permission`指令实现按钮权限控制?

```
<el-tabs>
  <el-tab-pane label="用户管理" name="first">用户管理</el-tab-pane>
  <el-tab-pane label="角色管理" name="third">角色管理</el-tab-pane>
</el-tabs>
```



```
// 前端组件名和组件映射表
const map = {
  //xx: require('@/views/xx.vue').default // 同步的方式
  xx: () => import('@/views/xx.vue') // 异步的方式
}
// 服务端返回的asyncRoutes
const asyncRoutes = [
  { path: '/xx', component: 'xx',... }
]
// 遍历asyncRoutes，将component替换为map[component]
function mapComponent(asyncRoutes) {
  asyncRoutes.forEach(route => {
    route.component = map[route.component];
    if(route.children) {
      route.children.map(child => mapComponent(child))
    }
  })
}
mapComponent(asyncRoutes)
```

大学四年你想要前端达到什么水平

大学四年是一个非常宝贵的学习和成长阶段，尤其是在前端开发领域。以下是一个比较全面的目标和计划，希望能帮助你在大学四年内达到一个高水平的前端开发能力：

第一学年

目标：打好基础

1. **HTML & CSS：** 掌握HTML5和CSS3的基本标签、属性和布局技巧。
2. **JavaScript：** 学习JavaScript的基础语法、数据类型、控制结构和函数。
3. **版本控制：** 学习使用Git和GitHub进行代码管理和协作。

资源：

- 在线课程（如Codecademy、freeCodeCamp）
- 书籍（如《JavaScript权威指南》、《CSS权威指南》）

第二学年

目标：深入理解前端技术

1. **高级JavaScript：** 理解闭包、原型链、异步编程、事件循环等高级概念。
2. **前端框架：** 学习至少一个主流框架，如React、Vue或Angular。
3. **响应式设计：** 掌握媒体查询和Flexbox、Grid布局，以及如何设计响应式网页。

资源：

- 在线课程（如Udemy、Coursera）
- 官方文档和社区论坛

第三学年

目标：项目实践和优化

1. **项目开发：** 参与或主导几个实际项目，从中学习项目管理、代码优化和团队协作。
2. **性能优化：** 学习如何优化前端性能，包括代码拆分、缓存策略、图片优化等。
3. **测试和调试：** 掌握前端测试工具（如Jest、Mocha）和调试技巧。

资源：

- GitHub上的开源项目
- 前端性能优化书籍和博客

第四学年

目标：专业化和就业准备

1. **全栈开发：** 学习基础的后端技术（如Node.js、Express）和数据库知识（如MongoDB、MySQL）。
2. **构建工具：** 掌握Webpack、Babel等构建工具的使用和配置。
3. **准备简历和面试：** 完善个人简历，准备技术面试题，模拟面试场景。

资源：

- 在线教程和文档
- 面试题库和模拟面试平台

持续学习和改进

1. **社区参与：** 积极参与前端开发社区，关注前沿技术动态。
2. **博客写作：** 记录学习过程和心得，分享技术文章。
3. **开源贡献：** 参与开源项目，提升代码质量和团队协作能力。

总结

通过系统的学习和不断的实践，四年后你应该能够：

1. 熟练掌握前端开发的基础和高级技术。
2. 能够独立设计和开发复杂的前端应用。
3. 具备良好的项目管理和团队协作能力。
4. 为进入职场做好充分准备，并在面试中表现出色。

希望这些建议对你有所帮助，祝你在前端开发的道路上取得成功！