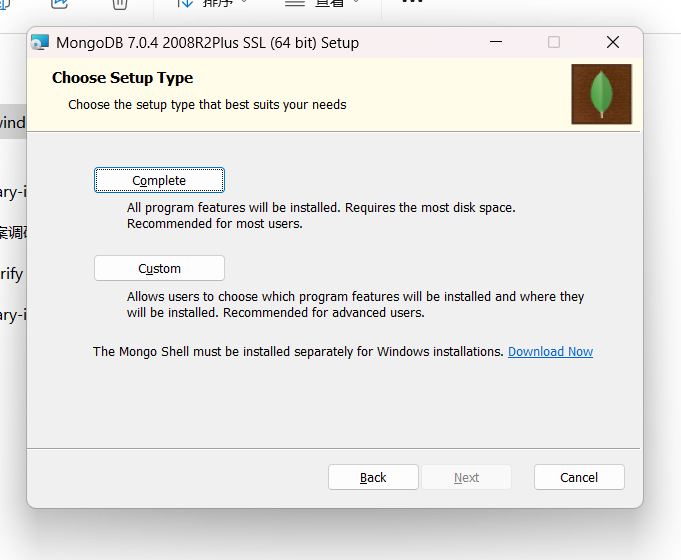
# 安装MongoDB

* 打开这个网站 <https://www.mongodb.com/try/download/community> 下载APP
* 点击下载好的APP 然后 next 在
* 在这个步骤时

在安装MongoDB的过程中，你通常会遇到“Complete”和“Custom”安装选项：

- \*\*Complete（完全安装）\*\*：这个选项将安装MongoDB的所有功能和组件，包括数据库服务器、Mongo Shell（命令行工具），以及其他相关工具。对于大多数用户来说，这是推荐的选项，因为它包含了你可能需要的所有功能。

- \*\*Custom（自定义安装）\*\*：如果你需要自定义安装，比如改变安装路径或者只选择安装某些特定的组件，那么可以选择这个选项。这适用于那些熟悉MongoDB并且知道他们具体需要哪些组件的高级用户。

根据截图中的提示，如果你不需要特定的自定义设置，可以选择“Complete”来进行全面安装。这样做可以确保你的开发环境中包含了所有可能需要的MongoDB工具和服务。

* 在上边这个步骤时图形用户界面, 应用程序

  描述已自动生成

在这个安装阶段，你需要配置MongoDB服务的一些选项。这包括如何运行服务、服务的名称以及数据和日志文件的存储位置。以下是你可以设置的一些选项：

1. \*\*安装MongoDB作为服务\*\*：确保选中了“Install MongoDB as a Service”，这样MongoDB将作为一个Windows服务安装，能够在Windows启动时自动运行。

2. \*\*服务运行方式\*\*：

- \*\*作为网络服务用户运行\*\*：选择“Run service as Network Service user”将使用Windows的网络服务账户来运行MongoDB服务。这通常是最简单的配置方式，不需要额外的用户管理。

- \*\*作为本地或域用户运行\*\*：如果选择“Run service as a local or domain user”，则需要提供特定的用户账户信息。这是一个更高级的选项，通常用于企业环境，需要你输入账户的域、名称和密码。

3. \*\*服务名称\*\*：

- 你可以保留默认的服务名称“MongoDB”，或者如果你希望自定义服务名称，也可以在此更改。

4. \*\*数据目录\*\*：

- 数据目录是MongoDB存储其数据文件的地方。点击“Browse”按钮选择一个你希望存放数据的路径。确认这个位置有足够的空间存储你的数据库。

5. \*\*日志目录\*\*：

- 日志目录是MongoDB将其日志文件写入的地方。同样，点击“Browse”按钮选择一个路径用于存储日志文件。

一旦你设置好这些选项，就可以点击“Next”继续安装过程。如果你更改了数据或日志的目录，请确保所选目录存在并且MongoDB服务具有读写这些目录的权限。如果目录不存在，安装程序通常会为你创建它们

* 这个步骤时图形用户界面, 文本, 应用程序, 电子邮件

  描述已自动生成

在这个安装步骤中，你被问到是否要安装MongoDB Compass。MongoDB Compass是MongoDB的官方图形用户界面（GUI），它可以帮助你更直观地管理和查询MongoDB数据库。

如果你是MongoDB的新用户，我建议安装MongoDB Compass，因为它会使得学习和操作MongoDB变得更简单。它提供了一个用户友好的界面，可以让你执行查询、查看数据结构、分析性能问题，以及更直观地查看数据库中的文档和集合。

如果你打算使用MongoDB Compass，只需选中“Install MongoDB Compass”复选框，然后点击“Next”继续安装过程。这将自动下载并安装最新版本的MongoDB Compass。

如果你决定不通过安装程序安装Compass，你可以随时从MongoDB官方网站手动下载并安装它。

* 安装成功后 这个界面

图形用户界面, 文本, 应用程序

描述已自动生成

MongoDB Compass 提供了多种语言选项，包括中文。你可以按照以下步骤来将界面设置为中文：

1. 打开 MongoDB Compass。

2. 在顶部菜单栏中，找到并点击 “Edit” 菜单。

3. 在下拉菜单中，选择 “Preferences” 或可能的类似选项。

4. 在 “Preferences” 界面中，查找语言或地区设置。

5. 从语言选项中选择中文（可能显示为 “Chinese” 或 “中文”）。

6. 确认你的选择，并根据需要重新启动 MongoDB Compass 以应用更改。

如果找不到直接的语言设置选项，可能是因为当前版本的 MongoDB Compass 不支持在应用内直接切换语言。在这种情况下，Compass 通常会使用操作系统的默认语言设置。

如果在 MongoDB Compass 中找不到中文设置选项，并且你的操作系统已经设置为中文，Compass 应该会自动显示为中文界面。如果操作系统是英文的，而你希望使用中文界面，可能需要检查 MongoDB Compass 的支持文档或在线资源，看看是否有其他用户找到了解决方案。有时候，这些设置可能会因为软件更新而发生变化。

# 运行mongodb

命令: net start MongoDB或者 mongod 若是 上述都不行 则 使用mongod --config "C:\Program Files\MongoDB\Server\7.0\bin\mongod.cfg" （对应启动 MongoDB的 cfg 配置文件）

* 这个是安装好后 默认的相关信息

# 文本 描述已自动生成

# 要创建一个表

* 图形用户界面, 文本, 应用程序

  描述已自动生成

在MongoDB Compass的“Create Database”窗口中，你需要填写两个字段来创建新的数据库和集合：

1. \*\*Database Name\*\*：在这里输入你想要创建的数据库的名称。例如，如果你正在创建一个用于用户信息的数据库，你可能会命名为 `UserDB` 或 `user\_accounts`。

2. \*\*Collection Name\*\*：在这里输入你想在新数据库中创建的第一个集合的名称。集合在MongoDB中类似于传统关系型数据库中的表。因为你想要创建一个用户登录表，你可以将这个集合命名为 `users` 或 `user\_logins`。

填写这些信息后，你可以点击 \*\*"Create Database"\*\* 按钮来创建数据库和集合。

这里是一个如何填写的例子：

- Database Name: `UserDB`

- Collection Name: `users`

这将创建一个新的数据库 `UserDB` 和一个新的集合 `users`。在这个集合中，你可以存储用户信息，如用户名、密码（最好是哈希过的）、电子邮件地址等。

完成这些步骤后，你可以在新创建的集合中添加和管理文档，每个文档都代表一个用户记录。

图形用户界面, 文本

描述已自动生成

从您提供的截图来看，您已经成功创建了一个名为 `users\_logins` 的集合，它位于 `user\_accounts` 数据库中。目前这个集合中没有数据。

要在此集合中添加数据，您可以按照以下步骤进行：

1. 点击 \*\*"ADD DATA"\*\* 按钮。这里您有两个选择：

- \*\*"Insert Document"\*\*：通过一个JSON格式的文档来手动插入单个文档。

- \*\*"Import File"\*\*：通过导入一个JSON或CSV文件来批量插入数据。

2. 如果您选择 \*\*"Insert Document"\*\*，会出现一个可以输入JSON数据的编辑器。例如，您可以输入以下内容创建一个用户登录记录：

```json

[

{

"\_id": "uniqueId1",

"username": "exampleUser1",

"password": "exampleHashedPassword1",

"email": "exampleUser1@email.com",

"last\_login": "2023-01-01T00:00:00Z"

},

{

"\_id": "uniqueId2",

"username": "exampleUser2",

"password": "exampleHashedPassword2",

"email": "exampleUser2@email.com",

"last\_login": "2023-01-02T00:00:00Z"

}

] ```

输入数据后，点击 \*\*"Insert"\*\* 按钮来添加文档到集合中。

3. 如果您选择从文件 \*\*"Import File"\*\*，则需要选择一个包含要导入的文档的JSON或CSV文件。这对于批量导入数据很有用。

请记住，密码字段应该存储密码的哈希值而不是明文密码，以确保安全。

一旦您添加了数据，这些文档就会出现在 `users\_logins` 集合中，并可以在MongoDB Compass中进行查看和进一步操作。如果您需要执行查询，可以使用顶部的查询栏来输入MongoDB查询语句。

# 关于MongoDB的一个错误

MongoDB 连接错误: MongooseServerSelectionError: connect ECONNREFUSED ::1:27017

at \_handleConnectionErrors (D:\nodejsCoreProject\nodeCore\node\_modules\mongoose\lib\connection.js:809:11)

at NativeConnection.openUri (D:\nodejsCoreProject\nodeCore\node\_modules\mongoose\lib\connection.js:784:11) {

reason: TopologyDescription {

type: 'Unknown',

servers: Map(1) { 'localhost:27017' => [ServerDescription] },

stale: false,

compatible: true,

heartbeatFrequencyMS: 10000,

localThresholdMS: 15,

setName: null,

maxElectionId: null,

maxSetVersion: null,

commonWireVersion: 0,

logicalSessionTimeoutMinutes: null

},

code: undefined

}

这表明您的MongoDB实例可能未配置为在IPv6地址`::1`上监听，或者您的系统上存在IPv6与IPv4之间的解析问题。将`localhost`更改为IPv4地址`127.0.0.1`，您明确地告诉您的应用程序连接到IPv4的本地地址，这样绕过了任何潜在的IPv6问题。

通常，`localhost`应该能够解析为IPv6和IPv4的本地地址，但在某些系统配置或网络配置中可能存在问题。直接使用`127.0.0.1`是一个常见的解决方法，因为它明确指定了IPv4地址。

如果您在将来的开发中遇到相似的问题，记住检查和测试两种IP版本的地址可能会帮助您快速定位问题。

const mongoURI: string = 'mongodb://127.0.0.1:27017/user\_accounts' –-**解决方案：**把localhost 改为127.0.0.1

# 关于nodejs项目的操作

* 先配置session中间件 再配置 passport中间件

这个错误表示Passport试图使用基于会话的认证，但是您的Express应用没有配置会话支持。要解决这个问题，您需要确保`express-session`中间件已经安装并且在您的Express应用中正确配置了。

这里是您可能需要执行的步骤：

1. \*\*安装express-session\*\*:

如果还没有安装`express-session`，您需要安装它。

```bash

npm install express-session

```

2. \*\*配置express-session中间件\*\*:

在您的主文件（`app.ts`或`server.ts`）中，您需要在使用任何Passport中间件之前配置express-session。

```typescript

import session from 'express-session';

import passport from 'passport';

const app = express();

// 设置express-session中间件

app.use(session({

secret: 'your\_secret\_key', // 替换为一个随机密钥

resave: false,

saveUninitialized: false,

cookie: {

maxAge: 1000 \* 60 \* 60 \* 24 // 设置cookie的过期时间，例如，1天

}

}));

// 初始化Passport

app.use(passport.initialize());

app.use(passport.session()); // 这里是passport使用会话

```

请注意，`secret`应该是一个只有您知道的字符串，用于签名会话ID cookie，这是保证会话安全的重要部分。`resave`和`saveUninitialized`的设置可能根据您的具体需求而有所不同。

3. \*\*确保顺序正确\*\*:

确保`app.use(session({...}))`在`passport.initialize()`和`passport.session()`调用之前。这是非常重要的，因为Passport依赖于express-session的功能。

4. \*\*重启服务器\*\*:

您修改配置后，需要重启服务器来使更新生效。

完成上述步骤后，您的登录应该不再报错。如果问题仍然存在，请再次检查您的代码，确认`express-session`的配置没有遗漏，并且所有相关中间件的顺序是正确的。

* Passport中间件
* 关于正确会话的设置

如果在`logoutUser`函数中`req.user`是`undefined`，但在`loginUser`函数中却存在，这通常表明在用户登录和注销之间的会话状态没有被正确维持。这可能是由几个不同的原因造成的：

1. \*\*会话配置问题\*\*：

确保您的应用正确配置了会话（`express-session`）。会话是用来在多个请求之间保持用户状态的，如果没有正确配置，那么用户在登录后的下一个请求（如注销）可能就不会包含他们的身份信息。

2. \*\*Cookie和前端行为\*\*：

如果您的前端和后端是分离的，确保登录后收到的cookie（通常包含会话ID）被正确保存并在随后的请求中发送。如果cookie没有被发送或被某种方式清除了（如用户手动清除或浏览器设置），会话信息就会丢失。

3. \*\*Passport中间件应用问题\*\*：

请确认您在所有需要用户身份信息的路由之前调用了`passport.initialize()`和`passport.session()`。这确保了Passport可以在每个请求中访问会话数据。

4. \*\*会话存储问题\*\*：

如果您的会话存储（如内存、数据库等）无法正确维持会话状态，也可能导致这个问题。例如，如果您使用的是默认的内存存储，那么在服务器重启时会话信息会丢失。

5. \*\*前端请求问题\*\*：

确保前端在发起请求时，例如注销请求，正确地携带了认证信息（如会话cookie）。

6. \*\*路由配置\*\*：

您的路由配置看起来没有问题。确保在`router.post('/logout', logoutUser);`之前已经设置了`passport.initialize()`和`passport.session()`。

为了进一步诊断问题，您可以检查以下几点：

- 当用户登录时，检查服务器是否发送了设置会话cookie的响应。

- 在客户端（浏览器）检查是否保存了这个cookie，并且确保后续请求（如注销）都包含了这个cookie。

- 检查会话存储是否正确配置，确保会话数据能够在请求之间持久化。

- 确保没有任何中间件或浏览器行为在登录和注销请求之间清除了会话数据。

# 关于这个 nodejs的入口index.ts

是的，在 Express.js 中，`app.use()` 的顺序非常重要，因为它决定了中间件的执行顺序。每个 `app.use()` 调用添加一个中间件到 Express 应用程序的请求处理栈中。当请求到达时，Express 按照这些中间件被添加的顺序依次执行它们。

以下是你的 `index.ts` 文件中 `app.use()` 调用的顺序及其重要性的概述：

1. \*\*CORS Middleware (`cors`)\*\*: 这应该在其他所有中间件之前设置，以确保跨域请求的正确处理。

2. \*\*Express Session (`session`)\*\*: 由于它涉及到设置和获取会话，所以通常较早地放置在中间件栈中。这对于后续的认证和会话管理很重要。

3. \*\*Passport Initialization\*\*: `passport.initialize()` 和 `passport.session()` 用于初始化 Passport 和会话管理，通常在 `session` 中间件之后立即调用。

4. \*\*Body Parser Middleware (`express.json()`)\*\*: 这用于解析 JSON 请求体，应在处理路由之前调用。

5. \*\*Cookie Parser (`cookie-parser`)\*\*: 用于解析 Cookie，这对于处理 CSRF 保护和会话管理很重要。

6. \*\*CSRF Protection (`csrf`)\*\*: 这个中间件添加 CSRF 保护。它应该在解析 Cookie 和会话之后，但在定义路由之前设置。

7. \*\*Setting CSRF Token\*\*: 这是一个自定义中间件，用于在每个请求中添加 CSRF 令牌。它应该在 CSRF 保护中间件之后。

8. \*\*Routes (`authRoutes`, `menuItem`)\*\*: 路由处理器通常放在最后，因为它们依赖于前面的中间件（如会话、认证、CSRF 保护等）。

将这些中间件以不同的顺序排列可能会导致应用程序行为异常或安全问题。例如，如果 CSRF 保护放在路由处理器之前，那么所有进入这些路由的请求都会受到 CSRF 保护。同样，如果会话中间件在认证中间件之后，认证流程可能无法正确访问会话数据。

因此，理解并正确地安排中间件顺序对于确保 Express 应用程序的正确和安全运行至关重要。

# 关于websocket的 nodejs配置

基于您提供的项目结构图，我们可以开始配置 WebSocket。在Node.js中，`ws`是一个常用的WebSocket库。以下是一些基本步骤来在您的Node.js TypeScript项目中设置WebSocket：

1. \*\*安装ws库\*\*:

首先，您需要安装`ws`库。在项目的根目录中运行以下命令来安装它：

```bash

npm install ws @types/ws --save

```

2. \*\*创建WebSocket服务器\*\*:

您可以在`index.ts`或创建一个新的文件专门用于WebSocket服务器。这里是一个简单的WebSocket服务器的例子：

```typescript

import \* as WebSocket from 'ws';

import { Server } from 'http';

// 假设您已经有一个创建好的HTTP服务器

const server: Server = ...;

// 创建WebSocket服务器

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

wss.on('connection', (ws: WebSocket) => {

// 当有客户端连接时触发

ws.on('message', (message: string) => {

// 当服务器接收到来自客户端的消息时触发

console.log('received: %s', message);

});

ws.send('something'); // 向客户端发送消息

});

// 启动服务器

server.listen(process.env.PORT || 8080, () => {

console.log(`Server started on port ${server.address().port} :)`);

});

```

3. \*\*整合WebSocket到现有的HTTP服务器\*\*:

如果您的项目中已经有一个HTTP服务器，您可以将WebSocket服务器附加到同一个端口。确保您的HTTP服务器是用Node.js的`http`或`https`模块创建的，以便能够共享相同的TCP/IP端口。

如果您的HTTP服务器是用Express或其他框架创建的，代码将稍有不同。通常，Express服务器是这样创建的：

```typescript

import express from 'express';

import { createServer } from 'http';

const app = express();

const server = createServer(app);

// 这里添加WebSocket服务器代码

server.listen(3000);

```

4. \*\*类型安全\*\*:

由于您正在使用TypeScript，确保在处理WebSocket事件时使用正确的类型注解，以便获得类型安全和代码自动补全的好处。

5. \*\*测试WebSocket服务器\*\*:

最后，您可以使用WebSocket客户端（如浏览器或专门的测试工具）来测试您的WebSocket服务器是否正常工作。

请记得，实际的实现可能需要根据您项目的具体需求进行调整。这只是一个基本的指南，以帮助您开始设置。

# 有关websocket的一些解读

当然可以。`websocketServer.ts`文件是您项目中专门用来设置WebSocket服务器的地方。WebSocket协议允许在用户的浏览器（客户端）和服务器之间建立一个持久的连接，通过这个连接，服务器可以实时推送数据到客户端，客户端也可以随时发送消息给服务器。

这里是一个典型的`websocketServer.ts`文件可能包含的内容：

```typescript

// 引入WebSocket库

import { Server as WebSocketServer } from 'ws';

import { Server as HttpServer } from 'http';

// 此函数将设置并返回一个WebSocket服务器

export function setupWebSocketServer(httpServer: HttpServer): WebSocketServer {

// 创建一个WebSocket服务器，它附加到现有的HTTP服务器上

const wss = new WebSocketServer({ noServer: true });

// 监听'upgrade'事件，它是HTTP协议切换到WebSocket协议的信号

httpServer.on('upgrade', (request, socket, head) => {

// 处理协议升级请求

wss.handleUpgrade(request, socket, head, (ws) => {

// 触发'connection'事件，完成握手

wss.emit('connection', ws, request);

});

});

// 当有新的客户端连接时触发

wss.on('connection', (ws) => {

console.log('WebSocket client connected');

// 监听来自客户端的消息

ws.on('message', (message) => {

console.log(`Received message: ${message}`);

});

// 当连接关闭时触发

ws.on('close', () => {

console.log('Client disconnected');

});

// 向新连接的客户端发送欢迎消息

ws.send('Welcome to the WebSocket server!');

});

// 返回WebSocket服务器实例

return wss;

}

```

现在，让我们一步步解释这段代码：

- `import`语句导入所需的模块。`ws`是一个流行的WebSocket库，它可以与Node.js的HTTP服务器配合使用。

- `setupWebSocketServer`函数接收一个HTTP服务器实例。这是因为WebSocket服务器通常与HTTP服务器共享相同的端口。

- `new WebSocketServer({ noServer: true })`创建了一个WebSocket服务器实例，但是它不会自动开始监听连接。设置`noServer: true`是因为我们要手动处理`upgrade`事件。

- `httpServer.on('upgrade', ...)`是监听HTTP服务器上的`upgrade`事件。当客户端想要建立WebSocket连接时，会发送一个特殊的`upgrade`请求。

- `wss.handleUpgrade`方法用来处理这个`upgrade`请求，并建立WebSocket连接。

- `wss.on('connection', ...)`监听新的WebSocket连接。当一个新的客户端通过WebSocket连接到服务器时，会触发这个事件。

- `ws.on('message', ...)`在一个特定的WebSocket连接上监听消息事件，当客户端发送消息到服务器时，这个事件被触发。

- `ws.on('close', ...)`监听连接关闭事件，当客户端断开连接时触发。

- `ws.send('Welcome to the WebSocket server!')`向连接的客户端发送一个欢迎消息。

- 最后，函数返回创建的`wss`（WebSocketServer）实例，以便在其他地方使用，如广播消息到所有连接的客户端。

整个文件的目的是为您的应用程序提供实时通信功能。通过WebSocket连接，您可以实现如实时聊天、实时通知更新等功能。

# 关于nodejsTS项目的websocket的模块化处理

根据您提供的代码片段，您已经将 WebSocket 服务器封装到了 `websocketServer.ts` 文件中，并将其连接到了 HTTP 服务器。您在入口文件 `index.ts` 中启动服务器，这是一个非常典型的结构。

如果要进一步提升代码的可维护性和可扩展性，您可以进行如下调整：

1. \*\*将 WebSocket 事件处理逻辑分离出来\*\*：

在 `src/events/` 中为每种类型的数据（例如，K线图数据、聊天消息等）创建单独的文件，并在这些文件中处理相关的事件。

2. \*\*提供一个中心化的事件注册函数\*\*：

在 `src/events/index.ts` 中导入所有事件处理器，并创建一个函数来注册所有这些处理器到给定的 `Socket` 实例。

3. \*\*优化 WebSocket 服务器的封装\*\*：

保持 `websocketServer.ts` 的清晰和专注。它应该主要负责配置和启动 WebSocket 服务器，并注册上述的事件处理器。

4. \*\*维护入口文件 `index.ts` 的整洁\*\*：

入口文件应该保持简洁，它负责连接所有的部分。确保所有环境变量、服务启动和配置逻辑在这里清晰地定义。

按照这个方向，您的项目结构可能会如下所调整：

```

src/

|-- config/

|-- controllers/

|-- middleware/

|-- models/

|-- routes/

|-- events/

| |-- index.ts

| |-- kLineDataHandlers.ts

| |-- chatMessageHandlers.ts

|-- utils/

|-- websocket/

| |-- websocketServer.ts

|-- index.ts # 入口文件

```

在 `src/events/index.ts` 中的代码可能如下：

```typescript

import handleKLineData from './kLineDataHandlers';

import handleChatMessage from './chatMessageHandlers';

// 这个函数将所有事件处理器注册到传入的 socket 实例

export const registerEventHandlers = (socket) => {

socket.on('kLineDataRequest', handleKLineData);

socket.on('chatMessage', handleChatMessage);

// ...更多事件...

};

```

`websocketServer.ts` 可能如下：

```typescript

import { Server as HttpServer } from 'http';

import { Server as SocketIOServer } from 'socket.io';

import { registerEventHandlers } from '../events';

export const setupWebSocketServer = (httpServer: HttpServer): SocketIOServer => {

const io = new SocketIOServer(httpServer, {

// ...CORS 配置等...

});

io.on('connection', (socket) => {

console.log('Client connected');

registerEventHandlers(socket);

});

return io;

};

```

最后，在您的入口文件 `index.ts` 中，您可以这样启动服务器：

```typescript

import express from 'express';

import http from 'http';

import { setupWebSocketServer } from './websocket/websocketServer';

const app = express();

const server = http.createServer(app);

const io = setupWebSocketServer(server);

const PORT = process.env.PORT || 5000;

server.listen(PORT, () => console.log(`Server started on port ${PORT}`));

```

这样，您的 WebSocket 逻辑就与 HTTP 逻辑分离，每种事件处理都清晰地组织在自己的模块中，而您的入口文件则专注于启动和连接这些模块。这会使得未来的维护和扩展变得更容易。

# Nodejs关于文件上传的问题，尤其是断点续传的问题