**表1**

****

**本科生毕业论文（设计）**

****

**题 目**

**学 院**

**专 业**

**学生姓名**

**学 号 　 年级**

**指导教师**

**教务处制表**

**年 　 月 　 日**

**面向私人开发团队的云盘管理系统**

专业 计算机科学

学生 李正尧 指导教师 周尧

**摘要**：随着软件基础建设的完善，编程从专业的生产方式这一应用逐渐发展出更为广泛的应用——生产工具。随着各行各业的人加入到编程学习，越来越多的生产与研究团队都不可避免地进入了团队合作的进路。至今，以功能全面的 git 类工具和平台为代表的中大型项目管理工具占据了开发者市场；而在文件存储上，各大厂商的云盘产品一直占据消费者市场。但当消费者成为开发者的时候，这些业余开发者便不再适应用于专业开发或生活场景的云存储产品。本文主要探讨对于特定用户群体和特定业务需求，将研究课题限定在“软件开发者“和”小型团队协作“上，以放弃传统的因素，如：强大安全性能、海量数据存储性能、丰富交互功能为代价，以换取其它核心要素，如：少量交互、朴素操作逻辑、管理高效率等特征。基于上述理由，本文采用 JavaScript 及其周边技术，主要以 Web 网页前端和 API 接口服务后端 构成产品主体。首先对用户市场和相关竞品作出分析，设计出产品需求，根据产品需求细化为设计文档，确定了主体的下载与上传、用户系统、文件系统三个模块，并基于三个模块设计出 API 接口服务后端与 Web 网页前端间的接口协议细则，最后参考当前对网盘产品的研究做法，测试了各项功能的可行性与服务器面对不同用户数量同时请求的并发响应能力，确认产品主体足以应对小型开发办公的应用场景需求。

**关键词**：网盘 node.js Express Vue.js JavaScript 软件工程设计 软件测试

1. **绪论**

本章主要介绍网盘产品的研究主要重点，从此前的研究重点分析出针对如今互联网时代下涌入的大量开发者所需要的网盘产品应具备怎样的特征，及其重难点。

**1.1 综述**

目前对网盘产品的研究覆盖了很多应用场景，其中不乏有基于属性加密（ABE）的云存储方案的针对商用加密网盘系统[1]；以满足一般公司或机构对网盘使用的需求为目的，基于VPN的SAMBA网盘在私有云平台[2]；基于内容分块的TTTD（Two Threshold Two Divisors）算法，并在TTTD算法的基础上针对不同的文件类型创新出不同文件类型的分块算法，定制化的解决文件分块过程中产生的各类问题的网盘文件管理系统[3]；通过分析现有个人网盘系统的实现方式，结合自动同步和版本控制等系统关键技术，设计和实现的基于版本控制的网盘系统[4]等。

就以上五类具有代表性的例子而言：第一类产品的研究重点在于商业用途和安全，其产品可以考虑用别的性能指标换取安全性的提升；第二类产品的研究重点在于私有产权和商用，其产品重点在于利用内网穿透和代理实现广域网访问局域网服务器；第三类产品的研究重点在于使用一种分块算法，将上传的文件划分为若干块，并加以哈希进行索引，从而使用映射碰撞删除重复的文件块，使得允许使用各项性能换取存储效率最大；第四类产品的研究重点在于多终端的个人产品情况下，如何存储和同步数据、如何进行分量版本控制、如何实现自动检测与同步、如何实现差量版本控制。

不难看出，针对不同环节做出优秀的技术，都有相应的应用场景。其中技术难度最大的就是存储的版本控制和分量差量存储，其次是多端文件触发自动存储，再者是对数据存储的算法处理（如 TTTD 算法的数据组织结构），第四是网络建设中的通信与代理技术（如 VPN 连接和内网穿透、网盘挂载等），最后是加密算法保障安全操作。但随着技术体量的膨胀，随之而来的是部署成本使用成本增加。每一种独特应用场景的网盘产品都对应特定的用户，对用户的要求也有着较为多数的约束。自从互联网时代的爆发，业余开发者大量涌入，上述典型的网盘产品从定位开始必然不能满足下沉市场的多数用户。一种简化需求和功能而削减使用成本的网盘产品思路由于技术门槛低，便无人问津。但满足多数开发者用户的产品难点反而在基于对市场的理解与考察，对消费者习惯和思维的把握，从而进行的产品设计。

**1.2 网盘市场现状分析**

自以人工智能研究方法的成本降低为代表的时期始，各行业对计算机，尤其是编码的需求猛增。编码者群体日益庞大，团队合作的需求也日益增加。过去，编程语言的主要使用群体是职业开发者和研究人员，生产对象主要为科研或商业的中大型项目；现在，编程语言的使用群体被业余用户稀释，生产对象也多为小型日常工作的环节。

随着编程语言的生产对象规模缩小，项目代码本质上对管理的需求也从版本控制、安全性、分支更新和社交性转移到快速迁移、跨设备、简易性了。但市面上的产品两极分化，一方面是以 git 为代表的版本管理工具和平台，它们有着优秀且强大的分支版本管理系统，丰富的第三方服务平台与可操作性（GitHub Action），和互动性极强的社交系统；另一方面是以微云和百度网盘为代表的云盘产品，它们有着操作简单易上手，逻辑朴素的优点。

但双方的缺点也很明显：git 类的版本管理工具和平台对非专业人事不友好、冗余功能多、不符合消费者逻辑，故不宜作为日常小型开发工具使用；而商业云盘产品则以下载速度、文件体积、存储空间、缺乏团队交互操作、会员制收费等作为诸多限制，方便日常小型开发的同时也带来了人为成本。

除此之外，也有以专业开发者为首的小型团队，搭建了自己的项目服务器，有自己专属的工具（REPL 类占多数）。由于此类方案属于定制方案，且不能解决普遍的小型团队日常开发中团队协作问题，故不进行讨论。

对百度网盘和 GitHub 的用户数量进行讨论：百度网盘的用户基数超过6亿，大部分用户用百度网盘用习惯了，虽然有上传、下载限速的限制，但百度网盘推出单次付费下载后，有些用户已经接受了这种付费模式，而如果是会员用户，百度网盘的限速弊端根本就不存在。在诸多商业网盘产品中，百度网盘用户量最大，达6亿，超过已经上市的云存储服务商Dropbox（Dropbox的注册用户量才超过5亿），115、腾讯微云也在网盘倒闭潮后获得了不少新增用户。[5]而根据 GitHub 2021 Octoverse 报告数据，目前在 GitHub 上的全球开发者用户共有 7300 万+，今年新增用户 1600万+，《财富》100 强里就有 84% 的企业使用 GitHub。该报告数据显示，目前在 GitHub 上的全球开发者用户共有 7300 万+，其中来自中国的开发者用户已达 755 万，位居全球第二，而 JavaScript 则成为最受开发者欢迎的编程语言。（本次研究是通过对 400 万以上数据库、 12000 多名开发人员共同调查得出的结果，大体上揭示了开发者社区当前的发展趋势。）[6]

**1.3 论文组织结构**

1. **竞品分析到需求提取和技术选型**

本节主要针对主流上游和下沉市场的两大类产品作为竞品分析，来探讨一种优秀的、面向多数下沉市场的编码人员（下文简称“多数用户”）的云盘系统该具备的特征。先进行用户特性的分析，再将用户置于两类产品之不同场景下进行分析。

针对新编码者的涌入，多数用户的特点如下：他们大多来自于大专、本科、企业部门、和诸多小型实验室团队。文化水平以大专及以上居多，年龄多在 20 岁，使用语言主要为解释型语言（以Python为例）、嵌入脚本语言（以 PHP、JavaScript 为例）。由于技术栈和编程语言、和团队规模与项目需求的限制，他们的项目规模一般在万行级别内，且框架使用单一，项目运行平台和使用场景固定。这些因素决定了他们的团队规模一般在几人，项目不依赖过于强大的云服务和大型版本管理系统，需要高频从团队存储中下载和上传文件。

从对用户的特点分析上看，适合多数用户的小型团队开发的网盘系统应具备如下特性：

1. 易上手：迎合消费者使用习惯，所有基础操作功能接近“一看就会”，而不用阅读额外的用户说明。
2. 操作逻辑朴素：文件系统的空间与时间操作朴素，没有过于繁琐的功能。
3. 无限制：对非必要功能不做任何限制。
4. 权限自主：整个系统的权限归于团队领导，而非服务商。其中包括知识产权，数据安全责任，实体与虚拟财产权等。
5. 团队性：方便且友好的支持团队文件操作，应当不亚于办公室 NAS 的便捷程度。

接下来对竞品作纵向比较分析，以百度网盘为例：百度网盘作为面向 C 端的产品，满足了 1、2 两点（即易上手和操作逻辑朴素）。但由于其商业性质，导致百度网盘中后期运营时采用了和迅雷类似的限制带宽收费的策略，且发展出了会员制和云端操作付费制（云端解压收费）。其产品的权限在百度公司，

以github为例：github 作为面向专业开发者和开发团队的平台产品，有着面向 B 端和 C 端的性质，满足了 3、5 两点。由于其被微软收购，实体财产权归属于微软公司。但其过高的门槛和完全的版本控制系统，是多数用户所不能短期内接受，且长期培训成本过高的。在早期，第 4 点中以 private repo 为代表的私密性服务并不是所有用户都有能力购买，部分强大功能都是为 B 端用户所设计，这不满足多数用户位于 C 端范畴的现状。

因此，一款适合多数用户的产品已经从竞品分析中能得出初步的具体构想：

1. 像 C 端产品一样的符合消费者使用习惯的基础操作。
2. 像 C 端产品一样的朴素操作逻辑。
3. 对专业工具类产品的强大功能，除安全功能全面开放给用户外，一律移除。
4. 所有权限与责任留给团队自身。
5. 保留团队协作，用户管理，移除版本控制。

从竞品分析初步刻画了产品具体构想，在此进一步提取产品需求：

1. 带有 UI 的 APP 形式逻辑的操作。
2. 采用朴素文件管理系统，且仅支持增删功能。
3. 保证文件存储时安全，和通信时安全（即本地安全和传输安全）。
4. 实体和软件运行于团队基础设施内，即用户自己搭建系统。
5. 团队内成员共用网盘资源，团队领导享有对团队系统所有用户的管理权限。

根据以上需求，下面进行技术选型：

对于第 1 点，目前主流的可被接受的前端实现技术有：使用 Android SDK 开发 Android App、使用 Web 技术开发 Web App、使用 .NET 及其 SDK 开发 Windows 应用程序、使用第三方UI 库开发的应用程序（如ImGUI、JEUI、DWZ ）。

对于第 2 点，满足编写文件系统的需求，只需要支持 OS 中的 FS（file stream） 即可。JAVA 针对服务端的优秀 JVM 和 JIT 技术可以应对；而 C++ 作为老牌语言，其性能与编写者水平相关，且优化和维护复杂，上限高；脚本语言的内存资源成本较大（如 Python、Node.js）易编写适合快速开发。

对于第 3 点，本地安全的保证由 OS 和编程语言的内存管理机制保证，而通信时安全则由传输协议保证。目前已知可选的协议为 HTTPS。

对于第 4 点，可选技术有 S-C（Server-Client）模型，保证前后端分离即可实现软件和实体都运行于团队基础设施内。

对于第 5 点，需要设计一套用户管理系统。使用面向对象的编程方法即可实现。

综上，对于技术选型，能满足五点的方案之一的网盘系统如下：采用 S-C 模型，前后端分离。前端使用 Web App 的形式展现给用户，后端使用 Node.js 编写服务端，给用户提供上传下载的功能。

Web App 选型确定后，当前可选的主流动态 Web App 框架有 Angular、Vue、React。

AngularJS 诞生于2009年，由 Misko Hevery 等人创建，是一款构建用户界面的前端框架，后为Google所收购。AngularJS 是一个应用设计框架与开发平台，用于创建高效、复杂、精致的单页面应用，通过新的属性和表达式扩展了 HTML，实现一套框架，多种平台，移动端和桌面端。AngularJS 有着诸多特性，最为核心的是：MVVM、模块化、自动化双向数据绑定、语义化标签、依赖注入等等。[7]

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your applications.

The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks of the Angular framework are Angular components that are organized into NgModules. NgModules collect related code into functional sets; an Angular application is defined by a set of NgModules. An application always has at least a root module that enables bootstrapping, and typically has many more feature modules.

Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.

Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Modules, components and services are classes that use decorators. These decorators mark their type and provide metadata that tells Angular how to use them.

The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.

The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An application's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.[8]

Vue.js是一套构建用户界面的渐进式框架。与其他重量级框架不同的是，Vue采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用单文件组件和Vue生态系统支持的库开发的复杂单页应用。Vue.js 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。Vue.js 自身不是一个全能框架——它只聚焦于视图层。因此它非常容易学习，非常容易与其它库或已有项目整合。另一方面，在与相关工具和支持库一起使用时，Vue.js 也能驱动复杂的单页应用。[9]

Vue (读音 /vjuː/，类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。[10]

由于 React的设计思想极其独特，属于革命性创新，性能出众，代码逻辑却非常简单。所以，越来越多的人开始关注和使用，认为它可能是将来 Web 开发的主流工具。这个项目本身也越滚越大，从最早的UI引擎变成了一整套前后端通吃的 Web App 解决方案。衍生的 React Native 项目，目标更是宏伟，希望用写 Web App 的方式去写 Native App。如果能够实现，整个互联网行业都会被颠覆，因为同一组人只需要写一次UI ，就能同时运行在服务器、浏览器和手机。[11]

Node.js 对于 HTTP 有着很好的支持，通常，行为是通过脚本开头的回调来定义的，而在结尾处，服务器是通过阻塞调用启动的。在 Node.js 中，没有这样的启动事件循环调用。Node.js只是在执行输入脚本后进入事件循环。Node.js 在没有更多要执行的回调时退出事件循环。此行为类似于浏览器 JavaScript — 事件循环对用户隐藏。

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection, the callback is fired, but if there is no work to be done, Node.js will sleep.

const http = require('http');

const hostname = '127.0.0.1';

const port = 3000;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello World');

});

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

This is in contrast to today's more common concurrency model, in which OS threads are employed. Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node.js are free from worries of dead-locking the process, since there are no locks. Almost no function in Node.js directly performs I/O, so the process never blocks except when the I/O is performed using synchronous methods of Node.js standard library. Because nothing blocks, scalable systems are very reasonable to develop in Node.js.

If some of this language is unfamiliar, there is a full article on Blocking vs. Non-Blocking.

Node.js is similar in design to, and influenced by, systems like Ruby's Event Machine and Python's Twisted. Node.js takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems, there is always a blocking call to start the event-loop. Typically, behavior is defined through callbacks at the beginning of a script, and at the end a server is started through a blocking call like . In Node.js, there is no such start-the-event-loop call. Node.js simply enters the event loop after executing the input script. Node.js exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript — the event loop is hidden from the user.EventMachine::run()

HTTP is a first-class citizen in Node.js, designed with streaming and low latency in mind. This makes Node.js well suited for the foundation of a web library or framework.

Node.js being designed without threads doesn't mean you can't take advantage of multiple cores in your environment. Child processes can be spawned by using our child\_process.fork() API, and are designed to be easy to communicate with. Built upon that same interface is the cluster module, which allows you to share sockets between processes to enable load balancing over your cores.[12]

综合以上结论，前端采用 Vue.js 制作 Web App，后端采用 Node.js 制作 Server端。对于 Server 后端，再对几种技术框架进行讨论。目前主流的 Node.js 的开发框架分别有以下几类：Express、Meteor、Nest、Koa。

目前star 和下载量最高的老牌框架。Express 是一款基于Node.js以及Chrome V8引擎，快速、极简的JS服务端开发框架，它提供了用来开发强壮的 Web/移动应用，以及 API 的所有功能。并且开发人员还能够方便地为它开发插件和扩展，从而增加 Express 的能力。

Meteor是一个基于nodejs和mongodb数据库的实时web框架。你可以用js搞定客户端、服务端的开发。另外，客户端、服务端的界限被极大的模糊。客户端的界面跟服务端的数据是双向绑定的，修改服务端的数据，用户界面会随着更新；你也可以在客户端直接修改服务端的数据库。

作为目前上榜框架中发布最晚，也是star 最高且增长最快的 typescript 后端框架。Nest 是一个用于构建高效，可扩展的 Node.js 服务器端应用程序的框架。它使用渐进式 JavaScript，内置并完全支持 TypeScript（但仍然允许开发人员使用纯 JavaScript 编写代码）并结合了 OOP（面向对象编程），FP（函数式编程）和 FRP（函数式响应编程）的元素。Nest 框架底层 HTTP 平台默认是基于 Express 实现的，所以无需担心第三方库的缺失。 Nest 旨在成为一个与平台无关的框架。 通过平台，可以创建可重用的逻辑部件，开发人员可以利用这些部件来跨越多种不同类型的应用程序。 从技术上讲，Nest 可以在创建适配器后使用任何 Node HTTP 框架。 Nest 提供了一个开箱即用的应用程序架构，允许开发人员和团队创建高度可测试，可扩展，松散耦合且易于维护的应用程序。

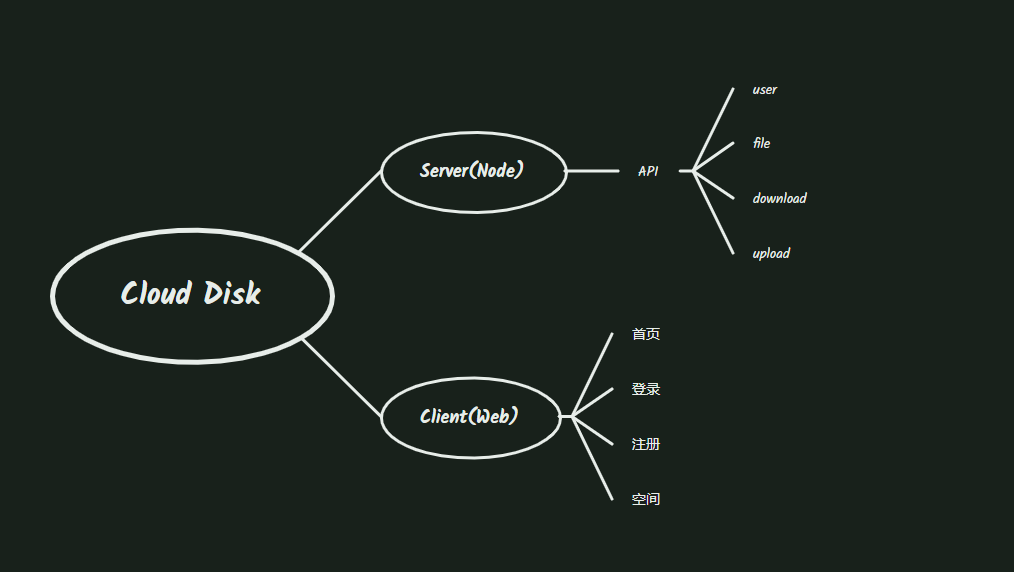
Koa框架由Express原班人马打造，它的核心是 ES6 的 Generator。Koa 使用 Generator 来实现中间件的流程控制，使用try/catch 来增强异常处理，同时在 Koa 框架中你再也看不到复杂的 callback 回调了。Koa框架本身非常小，只打包了一些必要的功能，但是它本身通过良好的模块化组织，让开发人员可以按照自己的想法来实现一个扩展性非常好的应用。[13]

结合以上对 Node.js 后端框架讨论，最终方案确定为：前端采用 Vue.js 制作 Web App，后端采用 Express 制作 Server端。并且制定好两方沟通的协议，设计 Server 端的 api 接口。

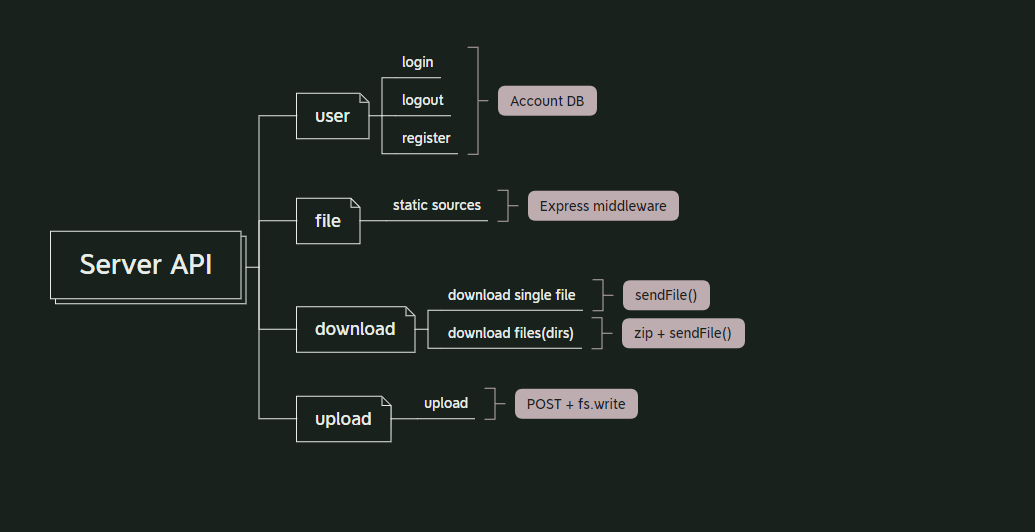
1. **设计和实现思路**

**3.1 产品结构设计图与功能设计图**

对产品进行整体设计，如下图：



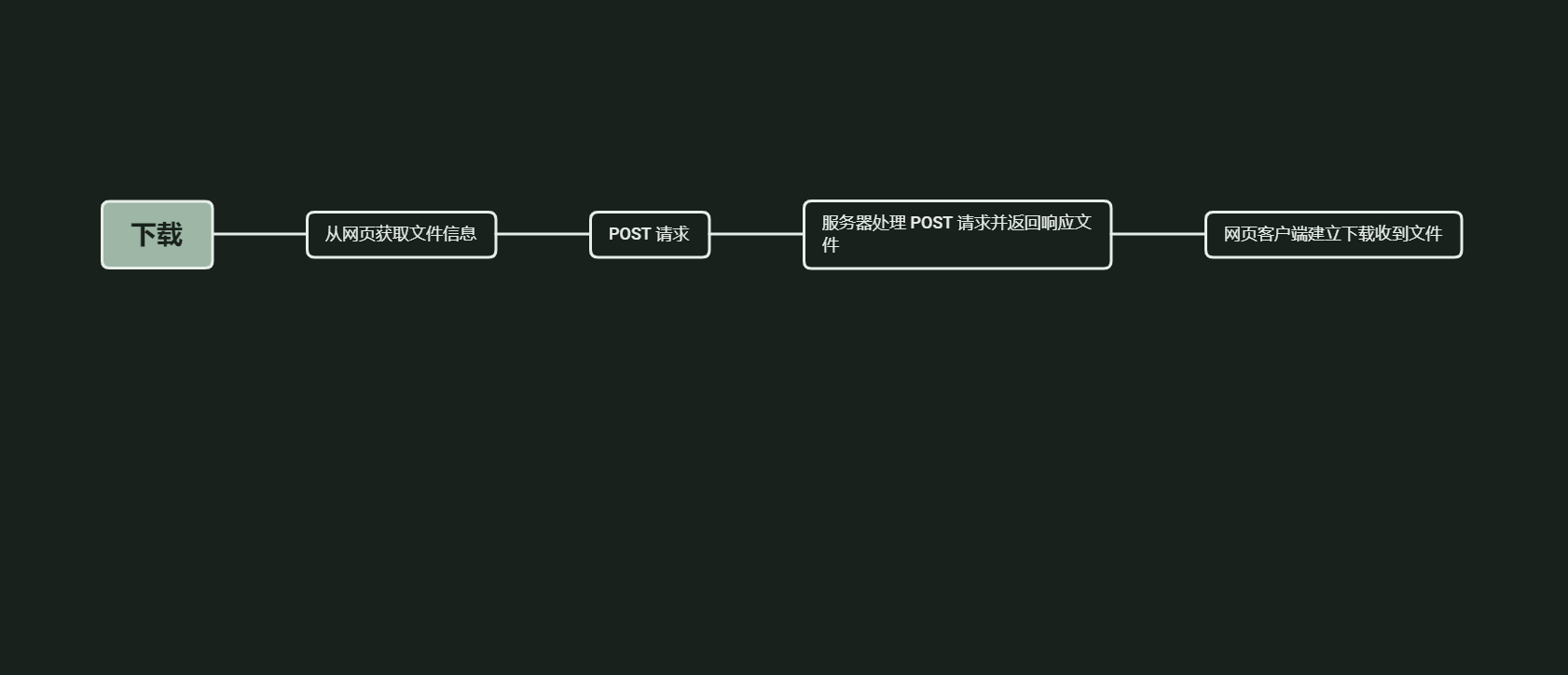
整个产品分为前后端：前端设计四个子页面，分别用于首页展示、登录账号界面、注册账号界面、文件空间界面；后端主要处理前端网页发送来的请求，以处理 HTTP 请求分类包装成 API ，通过对不同路由的不同种类请求，大致分为 user（处理账户登录、登出与注册）、file（文件展示与访问）、download（下载文件）和upload（上传文件）四类请求。服务端对前端网页发送来的请求，进入 API 后在服务器上做出相应处理动作，然后将处理结果返回给网页前端。



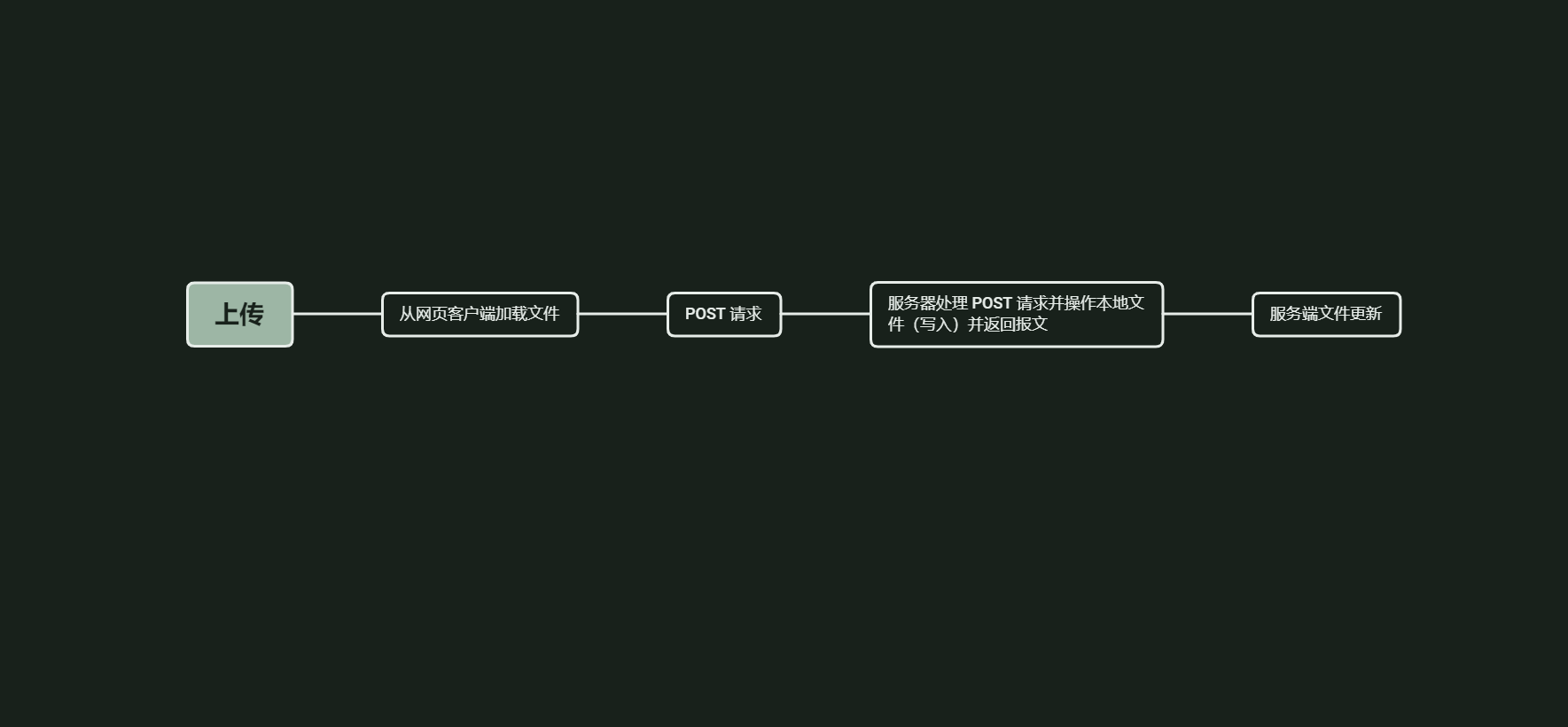
在 Server 端 API 中，user 部分处理三类请求，分别是登录，登出与注册账号。它们的实现均由服务端开启的数据库处理，它们的操作分别对应数据库对账户表单进行增删查改；file 部分只负责将用户存储在本地（相对于服务器来说）的文件进行静态资源托管，它由 Express 的middleware（中间件）实现；download 部分分为两类请求，一类是单文件下载，一类是多文件下载。其中单文件下载使用 Express 针对 HTTP 请求设计的返回报文中的 sendFile() 方法实现相应文件的发送（至客户端）。而多文件和文件夹整体的下载思路一致，都先经过服务器对多文件 zip 压缩，然后将压缩文件采用单文件一样的思路使用 sendFile() 方法实现相应文件的发送（至客户端）；upload 部分采用将文件通过 POST 请求写入报文正文中通过 HTTP 协议传输至服务端，由服务器解析报文再将文件从缓冲写入本地。



网页前端由 Vue.js 制作，分别分为四个页面：首页页面、空间页面、登录页面和注册页面。首页页面负责展示首页信息；空间页面负责展示网盘空间下团队文件目录，支持通过点击访问文件树和通过点击下载单个文件，并提供下载文件和上传文件的功能；登录页面负责登录和登出账号，如果未拥有账号还提供注册按钮供跳转到注册页面；注册页面负责注册账号，需要两次确认密码，以防止错误。



下载功能的重点流程如上图：下载的使用方法分为两种，一种是单文件下载，另一种则是多文件下载。单文件下载直接点击文件名即可触发浏览器接受服务器发送的文件下载，而多文件下载则需要依次点击文件名（或者文件夹名）前的可选框。再选择完成后，点击下载按钮，触发浏览器接受服务器发送的文件下载。下载完成后，可以得到一个已选文件的压缩包。



上传功能的重点流程如上图：在当前网盘空间的文件路径下，点击上传按钮，触发浏览器的上传文件选择框。选择文件进行上传即可。

**3.2 功能实现**

服务端下载 API（download API）实现如下：

const express = require("express");

const router = express.Router();

const path = require("path");

const zipFolder = require("zip-folder");

router.post("/", (req, res) => {

const type = req.body.type;

if (type === "dir") {

const dir = req.body.path.split("/").pop();

const cource = path.join(APPROOT, req.body.path);

const target = path.join(APPROOT, "public", `${dir}.zip`);

zipFolder(cource, target, (err) => {

if (err) return console.error(err);

res.sendFile(target);

});

} else {

res.sendFile(path.join(APPROOT, req.body.path));

}

});

module.exports = router;

额外使用了 zip-folder 模块，便于在多文件下载时将选择列表中的文件压缩打包，然后发送给客户端浏览器。在 download API 的实现中对请求的正文中的下载路径进行字符串处理，然后调用 res.sendFile() 将目标文件发送到服务端。

相应的，Web 前端中的下载页面中，逻辑部分如下：

<script>

import Button from "@/components/Button";

const fileDownload = require("js-file-download");

export default {

name: "SelectList",

components: { Button },

data: () => {

return {

listDir: [],

listFile: [],

};

},

mounted() {

this.GetList();

},

methods: {

SelectAll: function () {

const input = document.querySelector("input");

this.SetAll(input.checked);

},

SetAll(state) {

const inputs = document.querySelectorAll("input");

inputs.forEach((e) => (e.checked = state));

},

ClickUpload() {

document.getElementById("UploadInput").click();

},

async Upload() {

const input = document.getElementById("UploadInput");

const form = new FormData();

const file = input.files[0];

form.append("file", file, file.name);

form.append("path", window.location.pathname);

await fetch("/api/upload", {

method: "POST",

body: form,

});

this.GetList();

},

async DownLoad() {

let selectedDir = [];

let selectedFile = [];

document

.querySelectorAll(".dir input:checked")

.forEach((x) => selectedDir.push(x.value));

document

.querySelectorAll(".file input:checked")

.forEach((x) => selectedFile.push(x.value));

var msg = "准备下载";

if (selectedDir.length > 0) msg += "\n以下文件夹：" + selectedDir;

if (selectedFile.length > 0) msg += "\n以下文件：" + selectedFile;

if (msg === "准备下载") {

alert("您还未选择内容");

return;

} else {

alert(msg);

const url = this.$route.path;

selectedDir.map(async (dir) => {

try {

const res = await fetch("/api/download", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

type: "dir",

path: `${url}/${dir}`,

}),

});

const text = await res.blob();

fileDownload(text, `${dir}.zip`);

} catch (err) {

console.error(err);

}

});

selectedFile.map(async (file) => {

try {

const res = await fetch("/api/download", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

type: "file",

path: `${url}/${file}`,

}),

});

const text = await res.blob();

fileDownload(text, file);

} catch (err) {

console.error(err);

}

});

this.SetAll(false);

}

},

async GetList() {

const res = await fetch(`/api/file/list?dir=${this.$route.path}`);

if (res.redirected) {

alert("请登录");

return this.$router.push(res.url);

}

const list = await res.json();

console.log(list);

this.listDir = list["dir"];

this.listFile = list["file"];

},

},

};

</script>

引入了 js-file-download 模块和自定义组件 Button。

在data() 中注册了 listDir 和 listFile 数组用于存放已选择文件夹和已选择已选择文件。其中存储的是相对于云存储空间的文件夹和文件的路径字符串。

在 mounted() 中挂载了 GetList() 方法，使得每次初始请求云存储空间页面时都会率先请求一次服务器，以获得相应的文件目录。

在 methods() 中注册了如下六个方法，分别是：SelectAll()、SetAll()、ClikcUpload()、Upload()、Download()、GetList()。SelectAll() 用于在当前云存储空间中文件路径下，勾选所有文件和文件夹；SetAll() 用于将 input 框标记为已选；ClikcUpload() 用于点击上传（之后会提到为何要多此一举）；Upload() 用于触发上传，从浏览器自带的文件选择框选择文件，然后打包，处理字符串路径后，加入到 form 中，最后利用 fetch() 将 form 文件体发送到服务器；Download() 用于下载文件，先初始化 selectedDir 和 selectedFile 为空数组，然后查询当前网页中已被勾选的文件和文件夹项目并计数，之后将被勾选项目添加进 selectedDir 和 selectedFile中，此后使用 alert 提醒用户即将开始下载（并对空下载等意外情况进行处理），之后在 fetch() 中打包好文件处于服务器的相对路径等报文主体与参数信息并分别发送对文件夹和对文件的 POST下载请求；GetList() 用于每次访问云盘空间时用 fetch() 向服务器发送 GET 请求以获取当前页面的文件树，以展示给用户。

与之并列的，Web 前端中相应的下载页面中，Html 部分如下：

<template>

<div class="login">

<div class="selection">

<div class="option">

<input type="checkbox" @input="SelectAll" />

全选

</div>

<div class="dir">

<a :href="$route.path.substring(0, $route.path.lastIndexOf('/'))">..</a>

<div v-for="dir in listDir" v-bind:key="dir">

<input type="checkbox" :value="dir.name" />

<a :href="$route.path + '/' + dir.name">{{ dir.name }}</a>

</div>

</div>

<div class="file">

<div v-for="file in listFile" v-bind:key="file">

<input type="checkbox" :value="file.name" />

<a :href="$route.path + '/' + file.name">{{ file.name }}</a>

</div>

</div>

</div>

<Button @click="DownLoad">下载</Button>

<Button @click="ClickUpload">上传</Button>

<input id="UploadInput" @change="Upload" type="file" />

</div>

</template>

值得一提的是这里有一个细节：浏览器只为文件选择器提供了 <input> 标签的实现，但要想将文件选择的上传按钮做得与下载按钮风格一致，则最好使用和下载按钮一样的写法（即采用同样的模块），之后用 @click 属性绑定上 ClickUpload 方法，通过 JavaScript 触发 <input> 标签的选择器。

与之并列的，Web 前端中相应的下载页面中，CSS 部分如下：

<style scoped>

.selection {

font-size: 1.5em;

}

.login a {

text-decoration: none;

}

.login {

outline: 3px solid #32b97b;

border-radius: 3px;

background-color: #87deb6;

color: #fcfcfd;

font-weight: bold;

padding: 30px;

}

.option {

outline: 2px solid #42ba84;

border-radius: 3px;

margin-top: 25px;

margin-bottom: 15px;

text-align: left;

padding: 10px;

width: 500px;

background: white;

color: #2c3e50;

}

.dir {

outline: 2px solid #eea069;

border-radius: 3px;

margin-top: 25px;

margin-bottom: 15px;

text-align: left;

padding: 10px;

width: 500px;

background: white;

color: #2c3e50;

}

.file {

outline: 2px solid #5879b7;

border-radius: 3px;

margin-top: 25px;

margin-bottom: 15px;

text-align: left;

padding: 10px;

width: 500px;

background: white;

color: #2c3e50;

}

</style>

服务端文件 API（file API）实现如下：

const express = require("express");

const router = express.Router();

const fs = require("fs");

const path = require("path");

let spacePath = path.join(\_\_dirname, "../../disk/");

function getAllFile(dir) {

var list = { dir: {} };

function traverse(res, dir) {

fs.readdirSync(dir).forEach((file) => {

const pathname = path.join(dir, file);

if (fs.statSync(pathname).isDirectory()) {

res.dir["file"] = {};

traverse(res.dir, pathname);

} else {

res.dir.join(file);

}

});

}

traverse(list, dir);

return list;

}

router.get("/list", (req, res) => {

const folderPath = path.join(\_\_dirname, "../../", req.query.dir);

const list = {};

list["dir"] = fs

.readdirSync(folderPath, { withFileTypes: true })

.filter((l) => l.isDirectory());

list["file"] = fs

.readdirSync(folderPath, { withFileTypes: true })

.filter((l) => l.isFile());

res.json(list);

});

getAllFile() 方法用于递归遍历所有文件和文件夹，用于获取云盘空间的文件树结构。/list API 用于向客户端发送文件树结构。

服务端上传 API（upload API）实现如下：

const express = require("express");

const router = express.Router();

const path = require("path");

router.post("/", (req, res) => {

const file = req.files.file;

file.mv(path.join(APPROOT, req.body.path, file.name));

res.end();

});

对上传的文件进，按照报文主体中路径参数进行文件写操作。

服务端用户 API（user API）实现如下：

const express = require("express");

const req = require("express/lib/request");

const path = require("path")

const fs = require("fs");

const router = express.Router();

let pathUserList = path.resolve(\_\_dirname,"../../account/UserList.json")

const userList = JSON.parse(fs.readFileSync(pathUserList,(err,data) =>{

if(err)

throw "Loading UserList.json wrong."

}))

router.get("/login", (req, res) => {

res.json({ username: req.session.username });

});

router.post("/login", (req, res) => {

if (

userList["data"].filter(

(user) =>

user.username === req.body.username && user.password === req.body.password

).length === 1

) {

req.session.username = req.body.username;

res.redirect("/space");

} else {

if(userList["data"].filter(

(user) => user.username === req.body.username).length === 0

)

res.status(401).json({ msg: "User is not exist." });

else

res.status(401).json({msg: "Wrong password."})

}

});

router.get("/logout", (req, res) => {

req.session.username = "";

res.end();

});

router.post("/register", (req, res) => {

if (userList["data"].filter((user) => user.username === req.body.username).length) {

res.json({ msg: "User already exist." });

} else {

userList["data"].push({ "username": req.body.username, "password": req.body.password , "authority": "member"});

try {fs.writeFileSync(pathUserList,JSON.stringify(userList))}

catch(err){

res.status(500).json({msg: "Register failed on server."})

console.log(err)

return

}

res.json({msg: "Register success."})

}

});

module.exports = router;

/login API 中，GET 请求的处理是将返回 session 中的 username，POST 请求的处理是将报文主体中的 username 和 password 和数据库中的账户密码进行对比，若正确则将 session 中的用户名设置为发送来的 username，返回登陆成功信息并发送给前端 Web；若错误则返回相应的登录失败信息并发送给前端 Web。

/logout API 中，GET 请求的处理是直接将 session 清空，达到登出的效果。

/register API 中，POST 请求的处理是直接将发送来的报文中 username 和 password 与数据库中用户表单进行对比，若可注册，则直接写入数据库，返回注册成功信息并发送给前端 Web；若与现有表单冲突，则返回注册失败信息并发送给前端 Web。

相应的，Web 前端中的登录页面中，逻辑部分如下：

<script>

import Button from "@/components/Button";

export default {

name: "Login",

components: { Button },

data: () => {

return {

username: "",

password: "",

};

},

methods: {

async login() {

this.$store.commit("set", true);

const res = await fetch("/api/user/login", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({

username: this.username,

password: this.password,

}),

});

if (res.redirected) {

this.$router.push(res.url);

}

if (res.status == 200) this.$store.commit("set", this.username);

else {

const json = await res.json();

alert(json["msg"]);

}

},

regis() {

this.$router.push("/register");

},

},

};

</script>

主要的方法为 login()，将页面中填入两个 <input> 元素内的数据通过 fetch() 发送给服务端，并接受响应报文。通过返回的响应报文状态码，做出响应提示。

与之并列的，Web 前端中相应的登录页面中，Html 部分如下：

<template>

<div class="login">

<div class="sub-item">

<div class="label">用户名</div>

<input v-model="username" />

</div>

<div class="sub-item">

<div class="label">密码</div>

<input v-model="password" type="password" />

</div>

<div>

<Button @click="login">登录</Button>

<Button @click="regis">注册</Button>

</div>

</div>

</template>

可以看到，只需在登录和注册按钮上，分别使用 @click 挂在上 login 和 regis 方法即可。

与之并列的，Web 前端中相应的登录页面中，CSS 部分如下：

<style scoped>

input {

outline: 3px solid #32b97b;

border-radius: 3px;

text-align: center;

font-size: 1.6em;

}

.login {

outline: 3px solid #32b97b;

border-radius: 3px;

background-color: #87deb6;

color: #fcfcfd;

font-weight: bold;

padding: 30px;

}

.sub-item {

font-size: 1.6em;

display: flex;

flex-direction: row;

align-items: center;

justify-content: center;

margin-bottom: 30px;

}

.label {

width: 120px;

display: flex;

align-items: center;

justify-content: center;

}

</style>

相应的，Web 前端中的注册页面中，逻辑部分如下：

<script>

import Button from "@/components/Button";

export default {

name: "Login",

components: { Button },

data: () => {

return {

username: "",

password: "",

repeat: "",

};

},

methods: {

async Register() {

if (this.password !== this.repeat)

alert("you've repeat the wrong password");

else {

const res = await fetch("/api/user/register", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({

username: this.username,

password: this.password,

}),

});

const json = await res.json();

alert(JSON.stringify(json["msg"]));

}

},

},

};

</script>

主要方法为 Register()，其实现中先检验重复密码是否一致，若不一致则用 alert() 警告用户，一致则使用 POST 请求将欲注册的用户名和密码发送至服务端的 register API，并使用 alert() 提醒用户返回报文信息。

与之并列的，Web 前端中相应的注册页面中，Html部分如下：

<template>

<div class="login">

<div class="sub-item">

<div class="label">用户名</div>

<input v-model="username" />

</div>

<div class="sub-item">

<div class="label">密码</div>

<input v-model="password" type="password" />

</div>

<div class="sub-item">

<div class="label">重复密码</div>

<input v-model="repeat" type="password" />

</div>

<Button @click="Register">注册</Button>

</div>

</template>

可以看到，重点是将 register() 方法通过 @click 绑定在 <Button> 控件上。

与之并列的，Web 前端中相应的注册页面中，CSS 部分如下：

<style scoped>

input {

outline: 3px solid #32b97b;

border-radius: 3px;

margin-top: 30px;

margin-bottom: 15px;

text-align: center;

font-size: 1.6em;

}

.login {

outline: 3px solid #32b97b;

border-radius: 3px;

background-color: #87deb6;

color: #fcfcfd;

font-weight: bold;

padding: 30px;

}

.sub-item {

font-size: 1.6em;

}

.label {

display: inline-block;

width: 140px;

text-align: center;

}

</style>

1. **难点汇总**

在前端 Web App 中，很难想象每一层及的模块节点都通过 props 来传输一些全局数据（或者对象）。这是由 props 的功能决定了的。而在前端 Web App 中，用户都希望自己的账户登录过后可以长期保持（这几乎已经是业界的通用做法了）。用来保存用户登录数据的技术，其最优做法是使用 session 技术。

Session 是一种持久网络协议，它可以在用户或者用户代理端与服务端创建一条关联，用以交换数据的机制。在不包含会话层（例如UDP 协议）或者是无法长时间保留会话层（例如HTTP）的传输协议中，session 的维持需要依靠在传输数据中的高级别程序。例如，在浏览器和远程主机之间的HTTP传输中，HTTP cookie就会被用来包含一些相关的信息，例如session ID，参数和权限信息等。在动态页面完成解析的时候，储存在会话中的变量会被压缩后传输给客户端的Cookie。此时完全依靠客户端的文件系统来保存这些数据（或者内存）。在每一个成功的请求中，Cookie中都保存有服务器端用户所具有的身份证明（例如 PHP 中的 session id）或者更为完整的数据。虽然这样的机制可以保存数据的前后关联，但是必须要保障数据的完整性和安全性。[]

于是，看似数据安全成了问题。但安全交给 https 传输协议保证，而不做本地加密解密。HTTPS 是一种通过计算机网络进行安全通信的传输协议。HTTPS经由HTTP进行通信，但利用SSL/TLS来加密数据包。HTTPS开发的主要目的，是提供对网站服务器的身份认证，保护交换资料的隐私与完整性。HTTP协议和安全协议同属于应用层（OSI模型的最高层），具体来讲，安全协议工作在HTTP之下，传输层之上：安全协议向运行HTTP的进程提供一个类似于TCP的套接字，供进程向其中注入报文，安全协议将报文加密并注入运输层套接字；或是从运输层获取加密报文，解密后交给对应的进程。严格地讲，HTTPS并不是一个单独的协议，而是对工作在一加密连接（TLS或SSL）上的常规HTTP协议的称呼。HTTPS报文中的任何东西都被加密，包括所有报头和荷载。除了可能的选择密文攻击（参见局限小节）之外，一个攻击者所能知道的只有在两者之间有一连接这一事实。

这样，使用 HTTPS 协议作为保障，然后利用 session 来存储用户信息。使用 express-session 模块来将 session 中间件使用在服务端上：

const session = require("express-session");

app.use(

session({

secret: "keyboard cat",

resave: false,

saveUninitialized: true,

cookie: {},

})

);

然后在前端 Web App 中使用 Vuex 模块将服务端的 session 信息更新存储到前端：

import { createStore } from "vuex";

export default createStore({

state: {

username: "",

},

mutations: {

set(state, value) {

state.username = value;

},

},

});

文件下载在 JavaScript中不算难点，但始终是麻烦的操作。中国国内网络上许多的做法是使用 Ajax、或者是使用 <a> 标签自带的 download 属性、又或者是使用 window.URL 这个 API 来完成操作。本文选择一种更优雅的做法，具体原因则不再赘述。

使用 js-file-download 模块，调用其中 fileDownlaod API 来完成下载操作：

const fileDownload = require("js-file-download");

async DownLoad() {

let selectedDir = [];

let selectedFile = [];

document

.querySelectorAll(".dir input:checked")

.forEach((x) => selectedDir.push(x.value));

document

.querySelectorAll(".file input:checked")

.forEach((x) => selectedFile.push(x.value));

var msg = "准备下载";

if (selectedDir.length > 0) msg += "\n以下文件夹：" + selectedDir;

if (selectedFile.length > 0) msg += "\n以下文件：" + selectedFile;

if (msg === "准备下载") {

alert("您还未选择内容");

return;

} else {

alert(msg);

const url = this.$route.path;

selectedDir.map(async (dir) => {

try {

const res = await fetch("/api/download", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

type: "dir",

path: `${url}/${dir}`,

}),

});

const text = await res.blob();

fileDownload(text, `${dir}.zip`);

} catch (err) {

console.error(err);

}

});

selectedFile.map(async (file) => {

try {

const res = await fetch("/api/download", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

type: "file",

path: `${url}/${file}`,

}),

});

const text = await res.blob();

fileDownload(text, file);

} catch (err) {

console.error(err);

}

});

this.SetAll(false);

}

}

1. **测试**
   1. **测试目的**

本章对本文所设计的基于 Express.js 和 Vue.js 的云盘系统进行测试与分析，从功能上测试其功能是否完整以及性能表现如何。

* 1. **测试环境**

测试时，需要将前后端运行的主机分离，且将前端 Web 网页代码编译后打包进 API 接口服务后端中，可以理解为：使 Web 网页的提供成为一个特殊的 API 响应。前端使用了普通家用 PC ，配置如表 5-1 所示；后端使用的服务器为天翼云弹性云主机，配置如表 5-2 所示，测试网络拓扑图如图 5-1 所示：

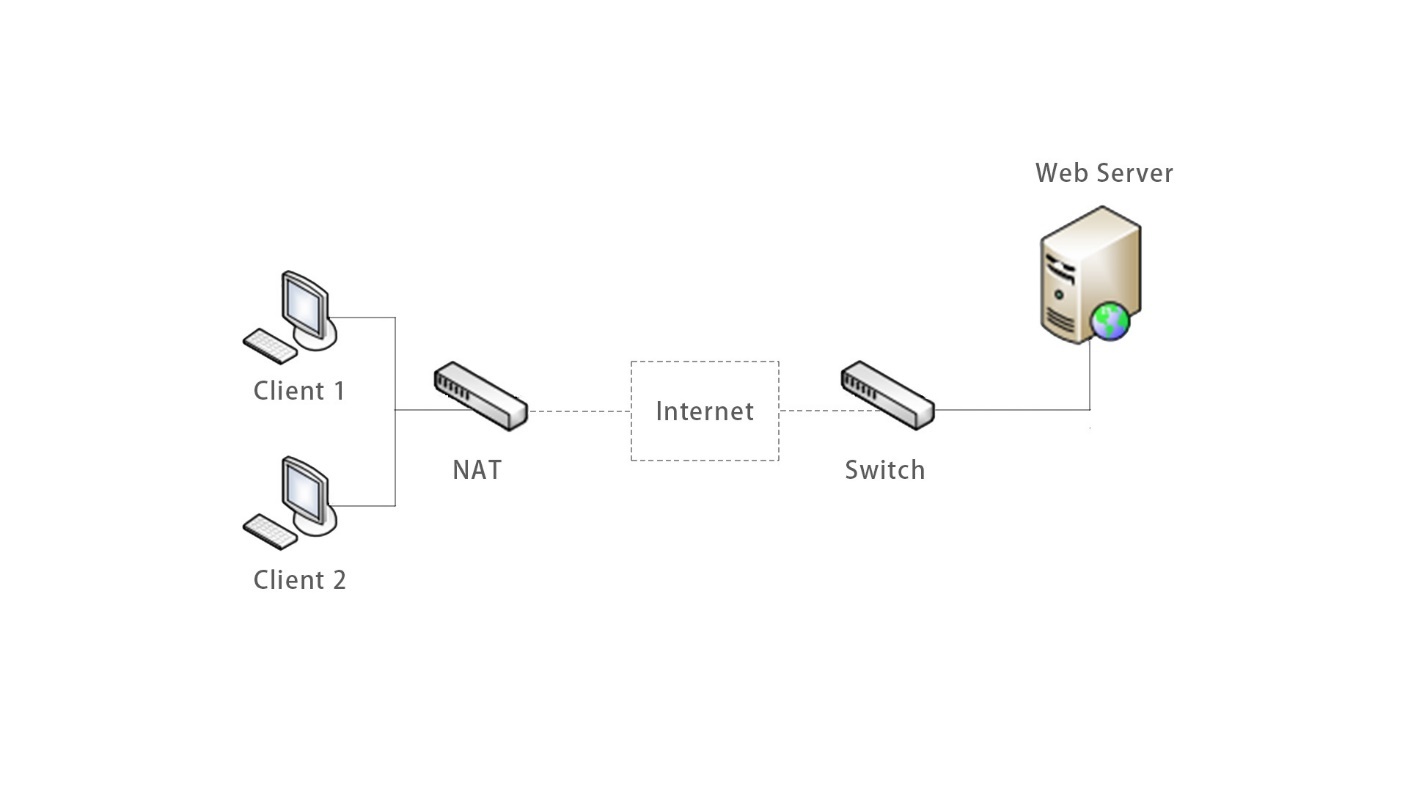
表5-1 客户端 测试用 PC 配置表

|  |  |
| --- | --- |
| CPU | 12th Gen Intel® Core™ i5-12600KF |
| 内存 | TEAMGROUP-UD4-4000 16G |
| 操作系统 | Win10 |
| 网卡 | Intel® Wi-Fi 6 AX201 160MHz |

表5-2 服务端 测试用服务器配置表

|  |  |
| --- | --- |
| CPU | 1核 |
| 内存 | 1G |
| 操作系统 | CentOS |
| 规格 | 内存优化型 | m6.3xlarge.8 | 12vCPUs |
| 磁盘 | 系统盘: 普通IO 40GB |
| 虚拟私有云 | vpc-ufis |
| 网卡 | 主网卡 subnet-3pbw, ipv4: 自动分配IPv4地址 |
| 安全组 | Sys-WebServer |

图 5-1 测试网络拓扑图

****

* 1. **测试过程**

首先进行对登录功能的测试，使用测试用账号密码登录，并测试其成功和失败结果，登录界面和失败提醒如图 5-2 和图 5-3、图 5-4 所示：

图 5-2 登录界面

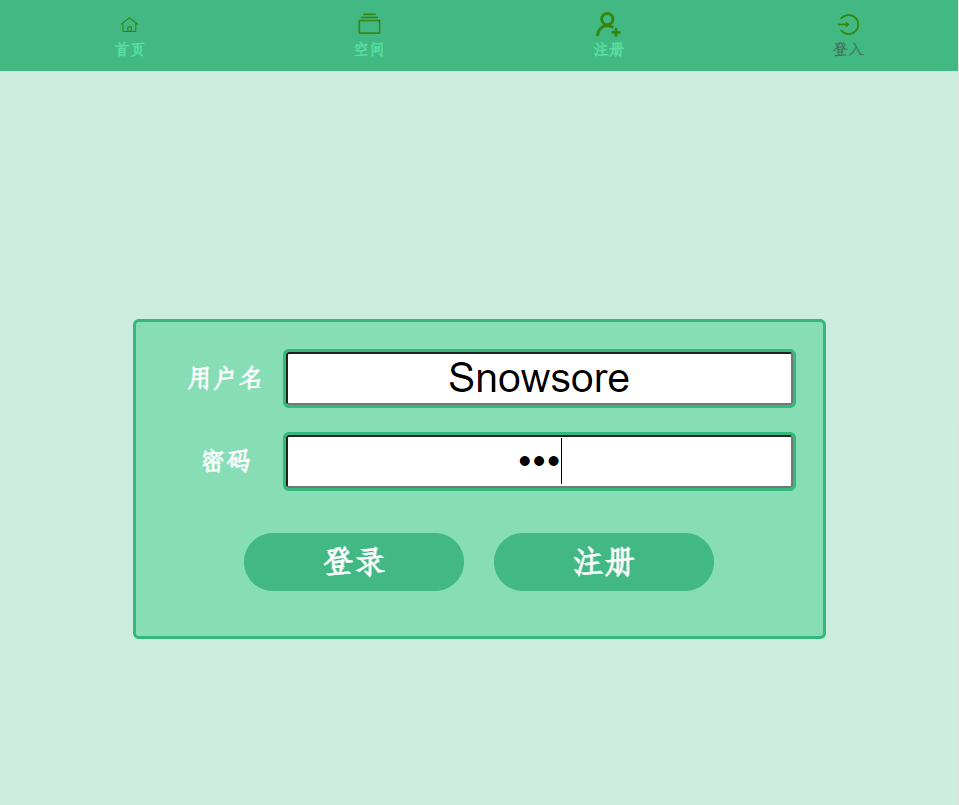


图 5-3 登录失败警告 1

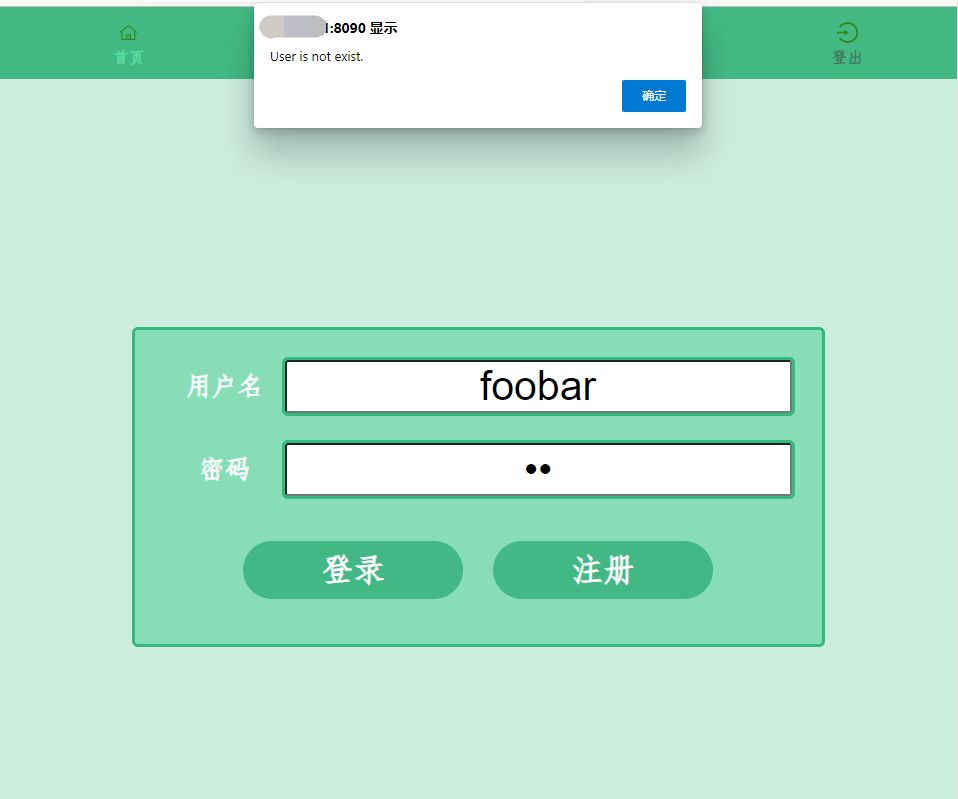
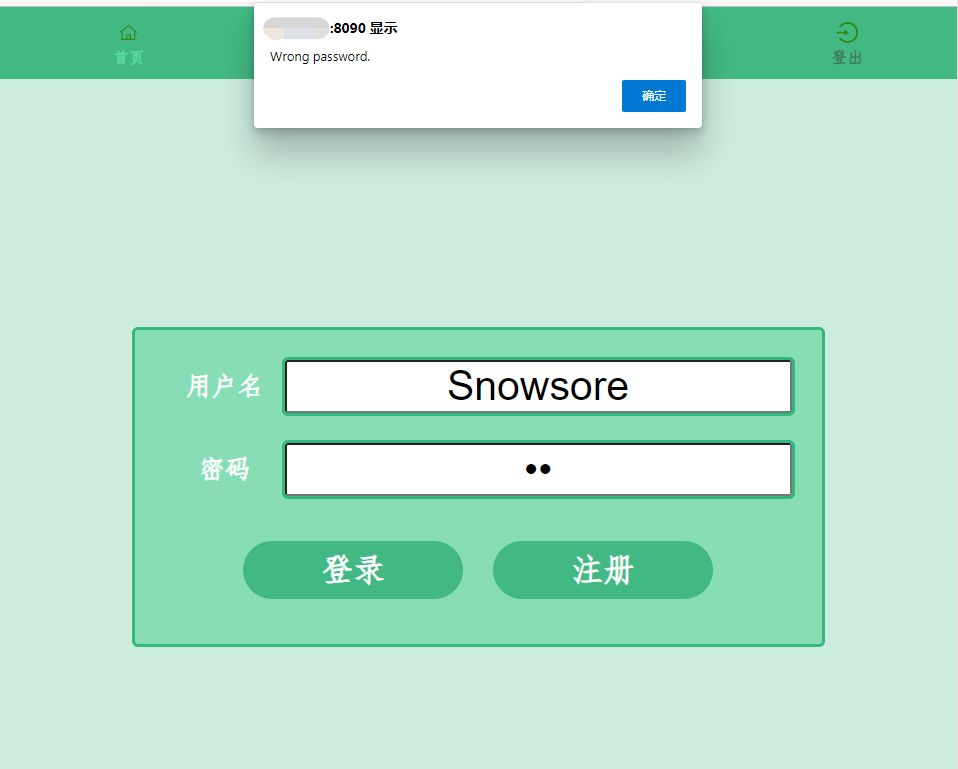
****

图 5-4 登录失败警告 2

****

其次进行对下载功能的测试。登陆后紧接着进入空间界面，选择文件进行下载，如图 5-5、5-6 所示：

图 5-5 空间界面下载

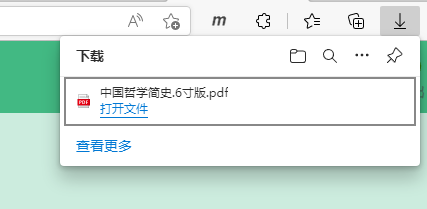


图 5-6 空间界面下载警告



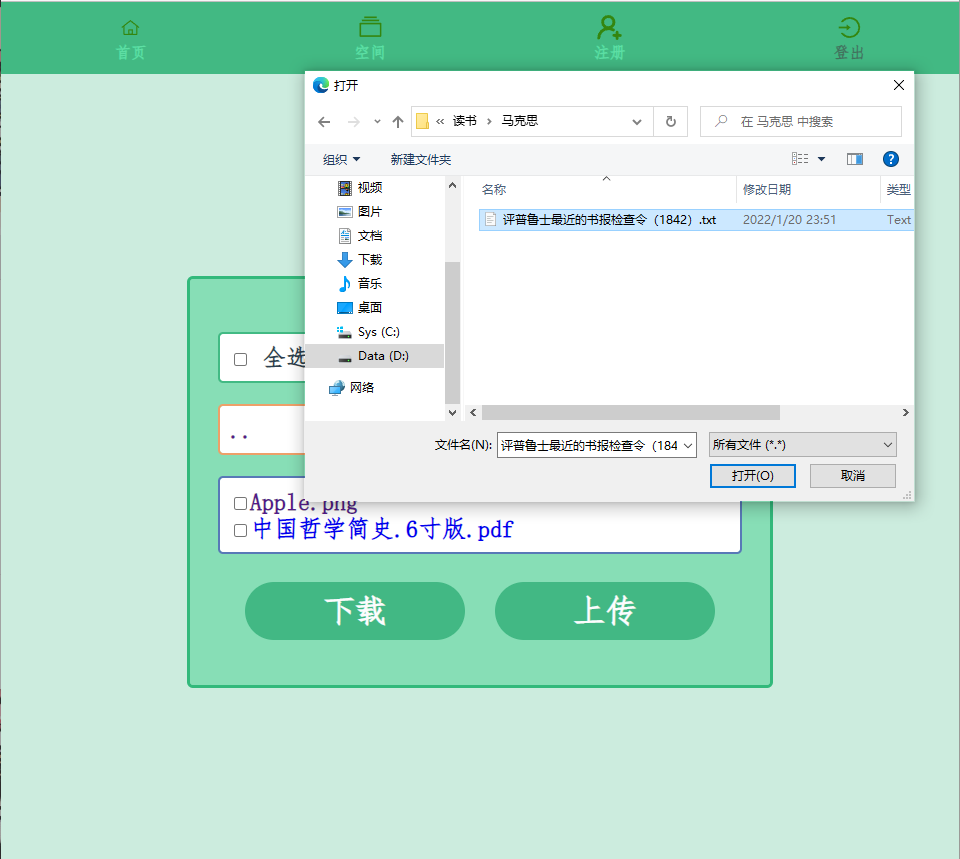
下载文件成功，结果如图 5-7 所示：

图 5-7 下载文件结果



再次进行对上传功能的测试。在空间界面下，点击上传，如图 5-8 所示：

图 5-8 空间界面上传



上传文件成功，结果如图 5-9 所示：

图 5-9 上传文件结果



接下来进行对注册功能的测试。在注册界面下，创建账号，如图 5-10 所示：

图 5-10 注册界面创建账号



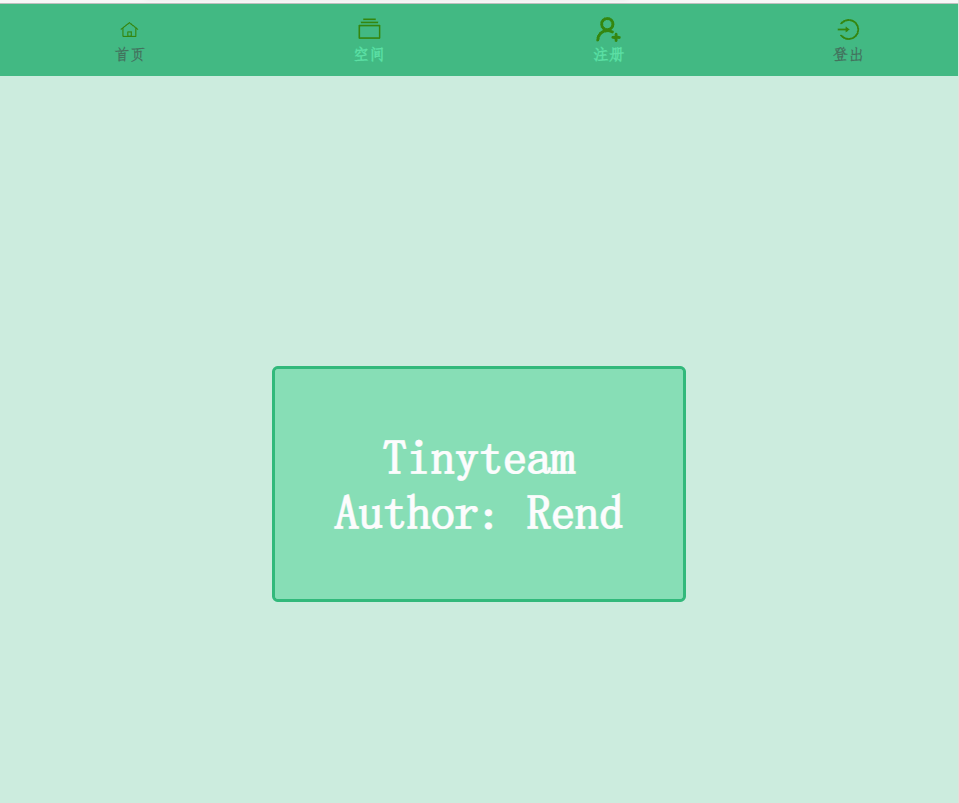
很显然，两次密码重复不相同。当注册成功时，不会触发任何警告，且会直接登录，以注册的用户身份和服务器利用 session 机制通信；反之当注册失败时，Web 网页将触发注册失败警告，如图 5-11 所示：

图 5-11 注册失败警告

****

最后进行对登出功能的测试。在任意界面下，点击右上角登出，如图 5-12 所示：

图 5-12 从任意界面登出



在任意界面点击登出时，右上角的登出按钮将会被重置为登入。并且，与之同时Web 页面将会被重新定向到登录界面，以提醒用户登录，如图 5-13 所示：

图 5-13 登出结果



在退出登录时，我们尝试进入空间界面。可以看到，未登录下访问空间界面是被服务器静止的，空间界面的内容也为空，此时 Web 网页会触发请先登录的警告，如图 5-14 所示：

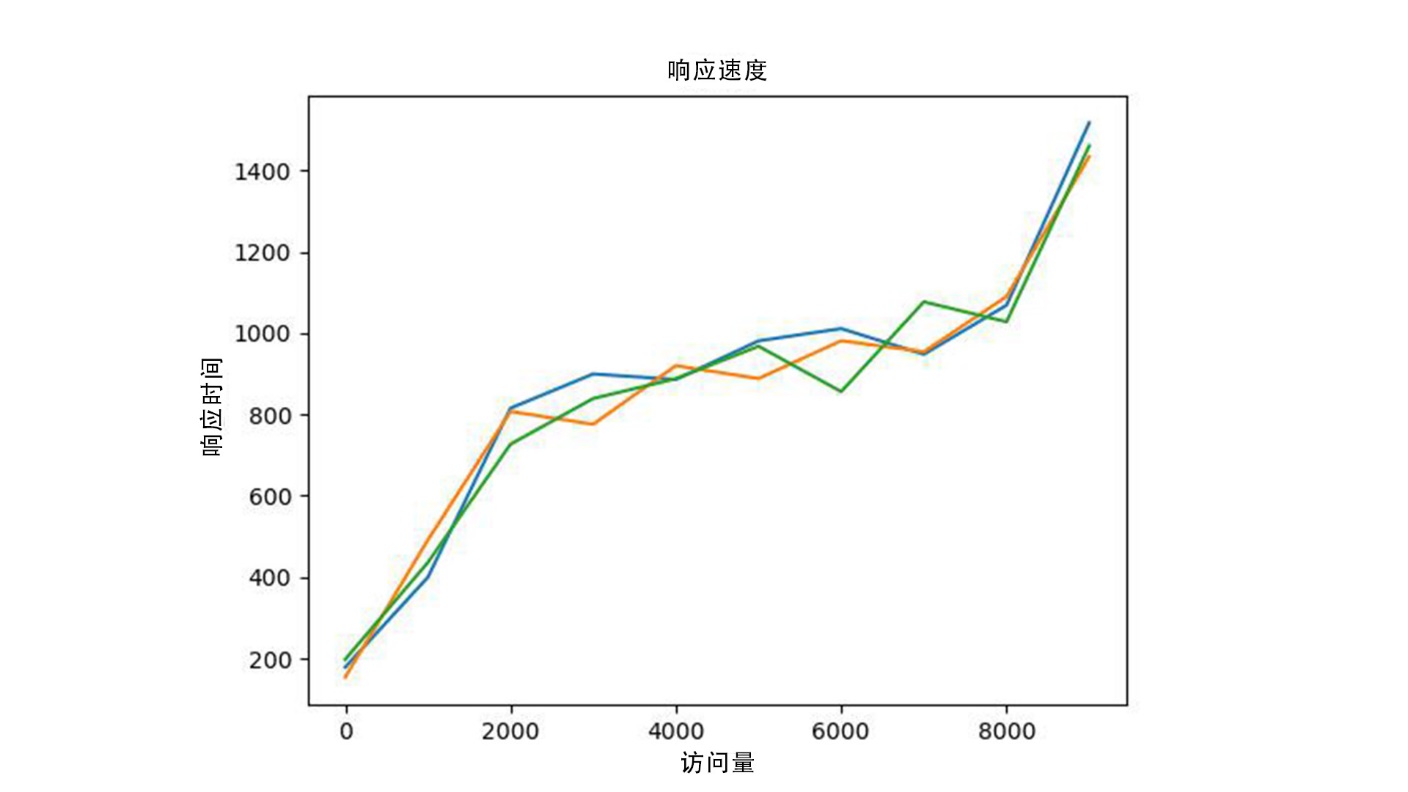
图 5-14 未登录下访问空间界面



* 1. **测试结果**

由于在 API 接口服务后端的设计中，没有对同一账户同时在线数做出限制，所以采用使用同一账户访问云盘空间的文件列表信息的测试手段。利用脚本同时发送请求，来测试服务器响应速度，结果如图 5-15 所示：

图 5-15 服务器响应速度测试（单位：毫秒）

****

可以看到，访问量在 8000 之前，基于 Express.js 和普通服务器的服务端延迟将近 1000ms，属于可接受范围内。考虑到小型团队的用户数量远不及 1000，情况将更为可观。

1. **总结**

本文通过对开发者市场现状进行分析，归纳出多数底层市场开发者在开发中的文件存储与共享需求。进一步地通过竞品分析，与高端与民用网盘产品对比分析，得出适合多数开发者需求的产品需求。以下载和共享文件功能为主，运用 Web 网页前端和 API 接口服务后端模块，设计了该网盘系统。

在 Web 网页前端的视图设计过程中，首先确定了主要功能区域：首页、空间、注册、登录（登出）。随后对各主要功能区域做出了不同的具体功能细化，数据交换的协议与格式，继而完成了整个 Web 网页前端的设计。

随后在针对 Web 网页前端的数据请求需求上，设计出了 API 接口服务器的主要接口类型：file、download、upload、user。所有接口响应请求的方法均采用了异步处理的形式，使得产品可以正常服务用户，最终实现了 Web 网页前端与 API 接口服务后端分离的适用于小型开发团队的网页云盘产品。

由于前端 Web 网页是基于 Vue.js 技术，运行在 V8引擎 上，使得客户端可以不依赖于平台，仅需要浏览器的支持，因此实现了跨平台用户体验。

之后对云盘产品整体进行下载负载的测试，在本文最后得出了产品不同访问量下延迟的测试。对一般小型办公场景的运用进行了评估。

参考文献：

[1]刘栋. 基于属性加密的网盘用户群组管理系统设计与实现[D]. 陕西:西安电子科技大学,2018.

[2]肖开强. 基于VPN的SAMBA网盘在私有云平台的设计与实现[D]. 四川:电子科技大学,2015. DOI:10.7666/d.D662407.

[3]侯银花. 基于TTTD算法的企业网盘文件管理系统的设计与实现[D]. 陕西:西安电子科技大学,2018.

[4]王韬. 基于版本控制的网盘研究与实现[D]. 四川:电子科技大学,2018.

[5]https://cn.bing.com/search?q=%E7%99%BE%E5%BA%A6%E4%BA%91%E7%9B%98%E7%94%A8%E6%88%B7%E6%95%B0%E9%87%8F&qs=n&form=QBRE&sp=-1&pq=%E7%99%BE%E5%BA%A6%E4%BA%91%E7%9B%98%E7%94%A8%E6%88%B7%E6%95%B0%E9%87%8F&sc=0-8&sk=&cvid=AC3A16554431498FA1015950C562BF5A Bing数据

[6]<https://octoverse.github.com/#what-makes-developers-and-teams-perform-better-be-more-productive-and-have-a-great-developer-experience> GitHub Octoverse 2021

[7]https://baike.baidu.com/item/AngularJS/7140293?fromtitle=Angular&fromid=9056478&fr=aladdin

[8]<https://angular.cn/guide/architecture>

[9]https://baike.baidu.com/item/Vue.js/19884851?fr=aladdin

[10]https://v3.cn.vuejs.org/guide/introduction.html#vue-js-%E6%98%AF%E4%BB%80%E4%B9%88

[11]https://baike.baidu.com/item/react/18077599?fr=aladdin

[12] <https://nodejs.org/en/about/>

[13]https://www.zhihu.com/question/346799170/answer/1618622477