MDSync - Markdown及其附件的同步

> 数据库期末项目

姓名	学号	联系方式
张烨禧	18340216	zetako@outlook.com
钟赫明	18340224	zhonghm5@mail2.sysu.edu.cn

1. 项目背景与简介

1.1 Markdown

与常用的商用文稿编辑软件，如Word、WPS、Pages等等相比，Markdown在标准公开、格式简洁上有着独特的优势，所以许多人喜欢使用Markdown来完成自己平时的文稿编辑、记录。

1.2 问题：Markdown的同步

商用软件中很多都有其自带的、基于相关账号或者云服务的多终端同步机制，如Word基于OneDrive和微软账号进行同步、WPS基于WPS账号、Pages基于iCloud和Apple账号。相比之下Markdown则缺乏类似的功能。而且，因为Markdown通常是以超链接的形式引用其他的多媒体文件的，所以这可能会给同步带来许多困难。

2. 项目研究内容（需求分析）

2.1 跨设备同步

Markdown作为一种追求专注的文件格式，其比较常见的使用方式是用来当作笔记。在不同的环境下，我们会需要使用到不同的设备进行记录；我们很自然的会希望这些记录的笔记可以随时在另一个设备中访问，修改等。当然，我们可以使用各种文件同步服务或云存储进行同步，然而当前的众多云存储服务并未对Markdown进行优化，他们只会将Markdown当作普通的文件进行同步，这样的操作欠缺灵活性。因此，我们需要一款专为同步Markdown设计的软件，并且它应该能支持各种各样的平台。

2.2 附件同步

一般来说，Markdown文件是一个纯文本文件；但是，我们在使用文件的过程中，不可避免的会引入一些多媒体资源，例如图片，视频等。对于这样的需求，Markdown给出的解决方案是超链接：利用URL将目标文件引入到Markdown中；通常来说，用户指定的链接会是一个相对URL，也就是指向本地某个文件的URL，这样的做法就导致了我们要同步Markdown时，必须将Markdown引用的其他资源也一并同步，并且还必须保持他们的相对位置不变，才能使其正常指向目标。因此，我们希望我们的服务能够在同步Markdown的同时，将其相关附件进行同步。

2.3 文件预览

很多时候，我们只是希望阅读Markdown文件，并不希望将文件下载后再打开，因此这个Markdown同步服务最好能够提供Markdown的预览功能。很遗憾的是，由于Markdown文件的纯文本属性，当前的大量同步服务都仅仅将其当作一个普通的文本文件打开，这不利于我们阅读。因此我们的服务最好能够提供一个云端预览的功能，以方便用户使用。

3. 技术路线与体系结构（包含数据库设计）

3.1 技术路线

3.1.1 通讯模型

3.1.1.1 基础概念

首先，作为一款同步软件，我们需要先构筑最基本的通讯模型。

我们的同步服务采用C-S结构，数据主要在服务端存储，在客户端缓存一部分。

图中展示了两个客户端访问服务器的情况，其中绿色表示有效副本，红色表示无效副本。客户端1首先需要将无效副本更新到有效；然后两个客户端都需要根据访问情况，决定是使用本地副本还是使用在线副本。

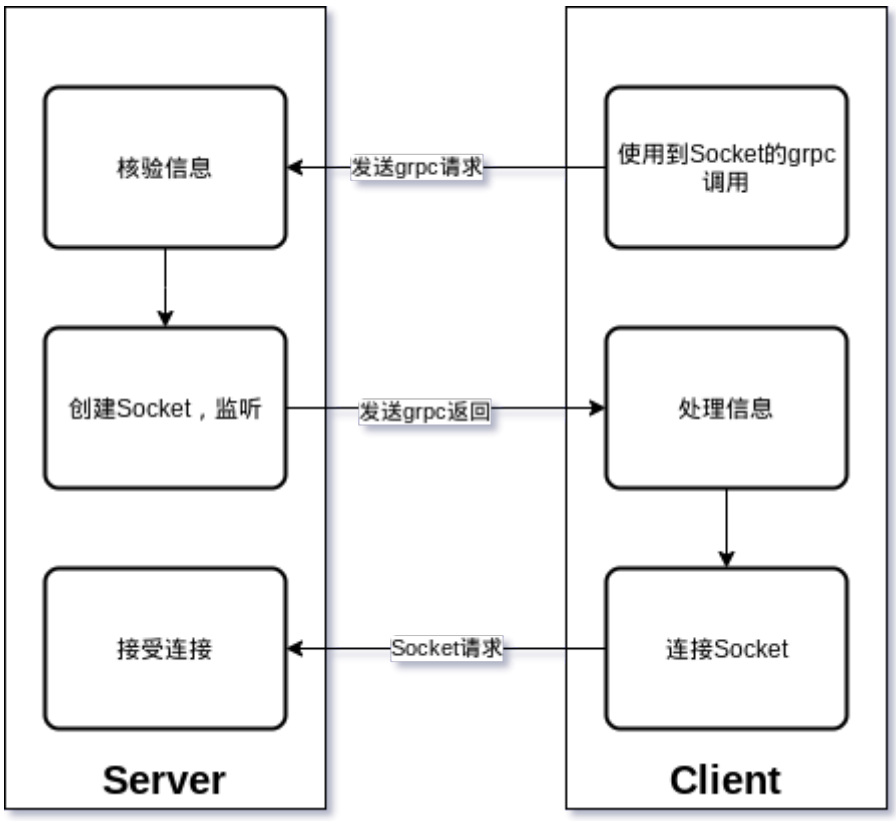
3.1.1.2 通讯协议

涉及到具体的通讯方式，我们选择使用grpc+TCP的通讯方式。

grpc是一个基于protobuf的RPC（远程调用方法）的实现，它有多种语言的API，可以保证在不同语言间的程序有相同的通信内容。由于我们可能需要支持多个平台，而在不同的平台上可能有不同的推荐语言（例如安卓上使用Java,IOS上使用OC等），因此一种有效的跨语言的通信方式是我们所需要的。我们将CS间的大部分的通讯都用grpc实现，这样，需要接入任何一个平台时，就可以不受到通讯协议的限制来选择实现语言。下面列出了我们的初期Demo使用到的grpc方法。

```
service serviceMDSync
{
    rpc login(loginInfo) returns (response);
    rpc getDirInfo(request) returns (stream fileInfo);
    rpc getFileInfo(request) returns (fileInfo);
    rpc fileOperation(request) returns (response);
    rpc uploadReq(request) returns (socketInfo);
    rpc downloadReq(request) returns (socketInfo);
    rpc newFileReq(request) returns (socketInfo);
    rpc getFileTree(request) returns (JSONString);
    rpc getTimeStamp(request) returns (JSONString);
};
```

而TCP则主要用来传输文件。由于上述的grpc作为一个RPC的实现，在传输大量数据上并没有很明显的优势，因此这里我们使用到最基础的一个TCP通讯协议来进行文件的传输，最大限度的保证兼容性。基本上，我们的TCP传输是由服务端进行监听，由客户端进行连接的；一个基本的流程如下图所示，首先由客户端发起某个需要用到Socket的请求，然后服务端准备好Socket连接的时候就发送连接所需的信息，最后由客户端去连接服务端提供的Socket，一个Socket就建立完成了。



3.1.1.3 语言选择

基于上述的技术，其实有多种语言可以选择；我们最初决定实现的是PC端（Windows/Linux/MacOS）的一套系统，因此我们的语言选择比较偏向于一些能够跨系统的语言。最后我们选择Node.js进行服务端的开发，并且选择基于Node.js的Electron进行服务端的开发。理由如下：

- Node.js的生态强大，有大量的完善的框架使用
- Node.js有一套完善的异步处理机制，这有利于我们这种长期驻守，并且比较注重IO的服务器的开发
- Electron由于其基于Web技术的特性，天生有强大的跨平台支持

3.1.2 同步检测

完成了基础的通讯模型之后，我们需要实现一个比较重要的功能：同步检测。同步检测指的是我们检测目标是否有本地缓存，本地缓存是否过期，是否需要提供对目标的更新等；要实现这个功能，我们初期使用的是时间戳的方式。

- 每个节点（文件/文件夹）都有一个时间戳的属性；
- 服务器自身有一个总体的时间戳，并且我们保证这个时间戳是单调递增的；
- 每次我们执行改变文件结构的操作，更新这个文件的时间戳为当前服务器的时间戳，并且递归更新它的父目录的时间戳；
- 客户端保存一个时间戳，当这个时间戳落后于服务端提供的时间，说明本地资源列表过期，从服务器拉取新的资源列表；
- 客户端每次访问一个文件，比较本地缓存的时间戳和资源列表的时间戳，若落后，说明本地缓存过期，需要重新拉取

3.1.3 文件拓展

有了上述的两个实现，我们就构造了一个稳定的同步系统；接下来，我们着手实现需求分析中提到的附件同步。附件同步的基础是文件解析；这个操作主要在客户端实现。注意到Markdown的任何外部链接都是以一个固定的格式指出的：

```
[title](URL)
![title](image-URL)
```

因此我们要做的事情非常明确：

1. 上传一个Markdown文件时，解析文件，提取出所有`[]()`的组合
2. 解析其中的URL，同样加入上传列表
3. 下载一个Markdown文件时，解析文件，提取出所有`[]()`的组合
4. 解析其中的URL，同样加入下载列表

3.1.4 拆分文件

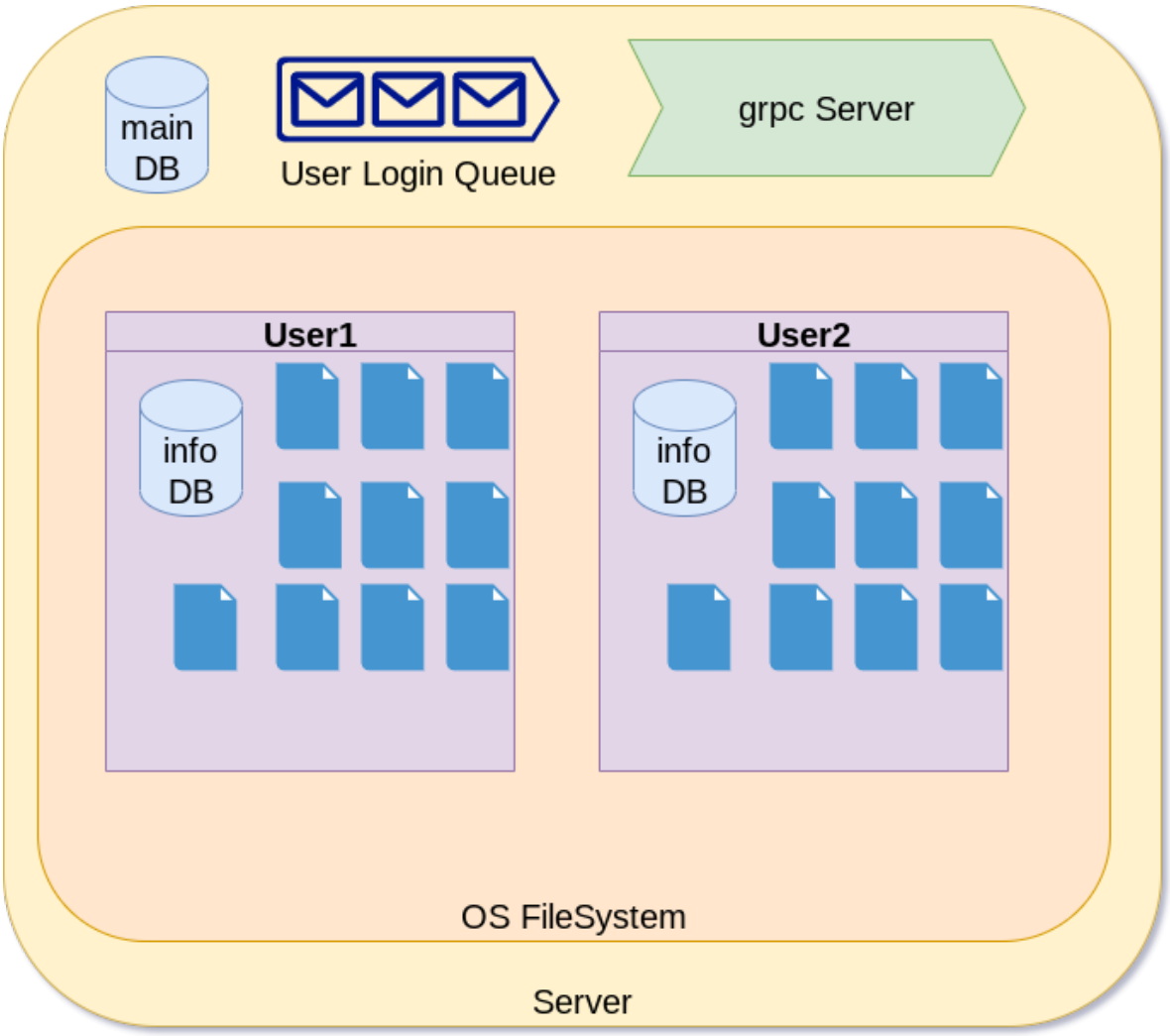
注意到Markdown文件是纯文本的文件，因此我们其实没有必要每次都同步整个文件；类似于Git，我们只需要传输“文件修改记录”就完成了文件的同步；这种同步方法可以大大的提升我们的同步效率。

但是使用这种方法，就意味着我们的服务端不能将Markdown文件作为一个完整的文件来储存，而是要记录文件及其修改记录，才能做到在客户端需要是提供其修改记录。

3.2 服务端设计

3.2.1 整体设计

服务端的主体是一个grpc服务器，一个用户管理队列，一个服务器数据的数据库以及存储用户信息的OS文件系统和记录用的数据库。



- 所有的请求都是通过grpc服务器接受的；
  - grpc方法绑定有不同的函数，grpc服务器根据请求，调用对应的函数执行功能；
- 服务器的相关信息（例如时间戳）写入到mainDB中，保证服务器失效后迅速恢复；
- 用户的登陆队列由队列管理器管理，及时踢出超时或失效的用户
  - 实际上，该队列是这样一个Map实例：`uuid->{username,TTL}`
  - 也就是说，每次用户登陆时，分配一个UUID
  - 用户通过这个UUID来访问资源时，系统查找队列来找到用户
  - 综上，可以通过管理这个Map来实现用户的访问权限管理
- 用户的文件以OS提供的FS API进行存储
  - 实际上使用的是模块 `fs-extra`
  - 每个用户有一个文件夹，作为自己的根目录
  - 每个用户有一个数据库记录自己的文件

3.2.2 数据库设计

3.2.2.1 数据库访问

基于方便部署的原则，该服务端的数据库不使用MySQL/MariaDB这类需要额外部署的数据库，而是使用SQLite3：

- 该数据库不需要部署，数据库存储本身作为FS的一个文件存在，而访问接口作为编程语言的一个函数存在
- 该数据库依旧能使用“较为完整规范”的SQL语句进行查询

同时，我们不使用官方提供的 `node-sqlite3` 模块，而是使用第三方的 `better-sqlite3`，有如下理由：

- 该模块提供的接口是同步函数，不会造成过长的回调函数链
- 该模块的操作效率比较高

3.2.2.2 主数据库

主数据库中有两张表：

- user（记录用户信息）
  - name：变长字符串，存储用户名
  - passwd：变长字符串，存储用户密码
- property（记录服务器属性）
  - name：变长字符串，记录属性名
  - value：变长字符串，存储属性值

前者用来记录用户信息，然后在登陆的时候可以进行查找比对；同时，也需要通过遍历这张表来维护runtime的完整性。

3.2.2.3 用户数据库

用户数据库中只有一张表：

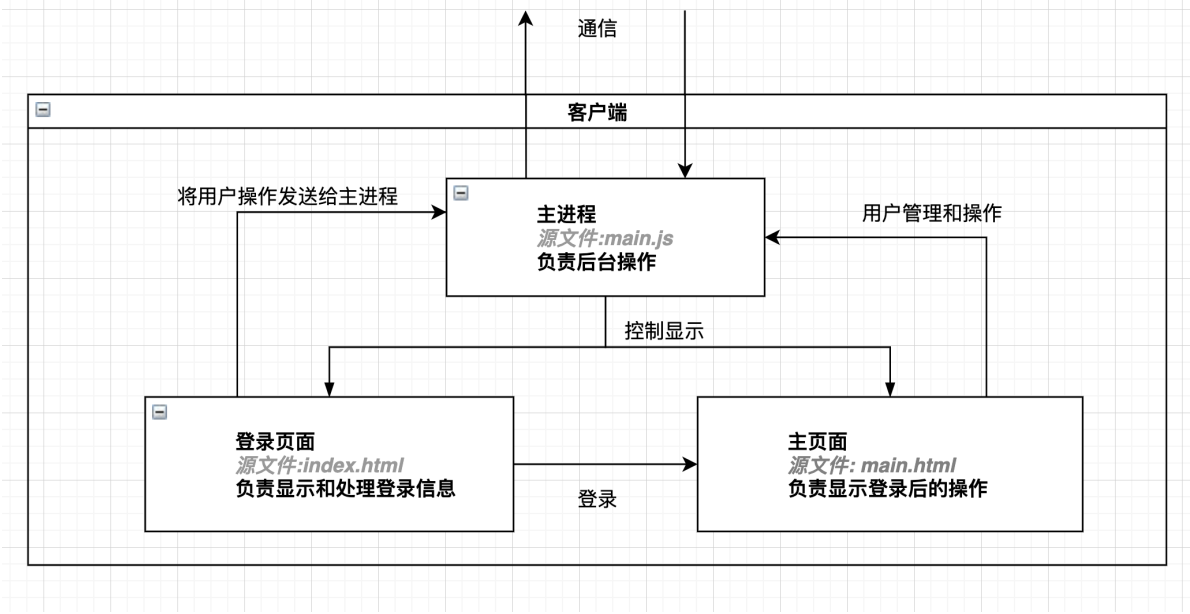
- file（记录文件的表）
  - id：整数；节点的ID,是主码，同时由数据库管理，自增
  - name：变长字符串；节点的名字
  - type：变长字符串；节点的类型（文件夹/Markdown文件/其他文件）

- path：变长字符串；节点的完整路径，方便访问
- parent：整数；记录父节点的ID
- timestamp：整数；记录文件最后一次修改对应的时间戳

3.3 客户端设计

3.3.1 整体设计

客户端的主体由一个登录页面、一个主页面和一个主进程组成，分别负责提供登录功能、登录后的操作功能和后台通信实现。具体结构如下图所示：



其中：

- 几乎所有与服务端的通信都是由后台的基于main.js的主进程完成的，该进程由electron提供，同时也负责管理窗口等功能。从显示登录页面跳转到主页面也是由其负责的。
- 登录页面和主页面都使用内嵌js代码来完成它们负责的处理，其中主要包括登录信息的获取、基于JsTree的文件树的操作、基于toastui-editor的markdown编辑器的操作。
- 主进程和页面(在electron中称为渲染进程，后面统一用该名称称呼登录和主页面)之间的数据交换使用electron提供的进程间通信API完成。
- 整个客户端基本上围绕着：登录、文件操作、文档编辑这几个机制来实现，其中包含了建立与服务端的rpc通信、通过socket实现文件传输、文件目录的定时同步，以及已编辑内容的更新上传等关键的具体实现内容。

3.3.2 登录机制设计

首先是登录机制的实现。这一部分主要围绕登录页面index.html和主进程main.js展开。由于我们的这个项目使用的是自建服务器，所以在登录的时候需要用户自己指定相应的服务器IP地址和端口号以供客户端与服务器建立连接。此外由于登录验证机制，我们还需要用户输入用户名和用户密码。有了这四项信息之后，我们就可以基于服务器IP和端口获得服务器提供的RPC存根，然后基于该存根以及用户信息来调用相应的登录rpc，然后根据rpc的响应内容来决定接下来的动作：

1. 登录成功：弹出一个提示框，然后发送信息给主进程，让主窗口加载main.html来跳转至主页，并进行接下来的操作；

2. 登录失败：这个是找到了服务器的情况下用户信息不正确，服务器拒绝登录导致的。这种情况下就弹出一个提示登录失败的弹窗，然后停留在登录页面，让用户自己去修正对应的输入内容或者退出程序；

3. 未找到服务器：当用户输入的IP地址和端口上没有一个对应的服务端在监听时就可能出现这种情况，可以通过rpc结果是否出错来判断，此时则弹出一个提示服务器未找到的提示框，然后也停留在登录页面，让用户自己修正相应内容或退出。

这就是登录机制的大体设计思路，具体实现可以查看client/index.html中的代码。

3.3.3 文件操作机制设计

然后是文件操作的相关设计。这部分发生在用户成功登录之后，所以是围绕主页面main.html和主进程main.js展开。按照我们的设想，在服务器提供的该用户根目录下，用户可以像对本地文件系统一样进行创建目录、添加文件、删除文件/目录、改名等文件操作，同时可以像查看文件系统一样查看自己在服务器上的文件。所以这里我们采用一个文件树的形式来显示所以保存在服务器上的相关内容，用户可以直接对该文件树中的每一个节点进行相应操作，这些操作(针对主页面)会对这个本地显示的树进行相应的修改，并且这些对应的操作会被发送给主进程，然后由主进程通过调用相应rpc来在服务器上也产生对应的操作，以此来实现本地操作对于服务器上内容的修改。这些操作主要包括以下六个：

1. 创建目录：在文件树选中的节点下创建一个子文件夹，如果选中的节点不是一个有效的文件夹，那么则弹出一个提示并取消操作，如果未取消则先在树中创建这个新节点，然后发送相应信息给主进程，主进程执行rpc来让服务器也执行一致的操作；

2. 修改名称：对于选中的树节点，通过调用JsTree提供的编辑API来修改名称，并且在改名完成之后，将相应的信息发送给主进程，并由主进程执行rpc来让服务器也执行对应操作；

3. 上传文件：利用electron提供的API打开一个文件选择对话框，用户选中一个文件并确定之后，将该文件的路径发送给主进程，然后主进程先将该文件拷贝到客户端本地固定目录上，然后执行rpc请求，并通过rpc响应中的信息建立socket通信来上传文件内容；

4. 下载文件：不同与前三个操作，这个操作在JsTree的一个事件触发中实现，在选中一个文件节点，并且该节点是一个文件的时候自动开始下载操作：先将下载请求发送给主进程，然后主进程检查是否存在未过时的本地拷贝，如果存在则取消下载，如果不存在则发送rpc请求，并通过rpc请求回应的socket信息来通过socket通信下载文件内容；

5. 删除项目：对于选中的树节点执行删除操作，删除本地节点之后，将操作发送给主进程，主进程执行rpc来让服务器也执行删除操作。

6. 移动项目：移动功能基于JsTree的节点拖拽插件dnd来实现。在拖拽操作完成后的事件中触发，并将相应信息发送给主进程，然后主进程调用rpc让服务器也执行对应操作。

上面这些就是客户端实现的所有文件操作了，可以看到这些操作有一个共同点，那就是它们都是先在渲染进程完成部分处理，然后再与主进程通信，然后由主进程完成接下来的处理，有些在主进程处理完之后还需要返还给渲染进程结果。

3.3.4 文档编辑设计

我们还提供了对文档进行编辑的功能，我们使用了开源编辑器项目：https://github.com/nhn/tui.editor 作为我们的markdown编辑器。对于已经下载的markdown文件(点击时)，我们可以将其文件内容显示在这个编辑器中，然后可以在这里对其内容进行编辑，如果这个内容已经被编辑过，那么就将其标记，然后如果这个时候用户点击其他文件，那么就弹出一个弹窗，然后询问用户是否放弃修改，如果不放弃，就继续修改，如果放弃，就转而显示其他文件内容。对于修改的文件内容，我们提供了一个保存按钮，能够将修改的内容通过对应的rpc发送给服务端，让服务端应用相应的修改。

4. 系统环境（软硬件环境）

4.1 服务端

当前的服务端可以运行在任何装有Node.js环境的计算机中。但是由于部分依赖的模块需要关联到其他语言编译的动态库（例如：`better-sqlite3` 需要C/C++编译），部署的计算机需要有相对应的编译工具链。基于未来可能的接入systemctl系统的需求，建议的部署环境是以systemctl方式进行初始化的Linux系统。

当前的服务端在Linux以及MacOS上测试通过。

4.2 客户端

当前的客户端基于electron+nodejs开发，理论上可以在Windows、Linux和macOS直接运行(但是开发过程中只在Linux和macOS下测试过)，但是可能需要安装配置electron和相关npm packages。

5. 系统功能设计

我们整个系统基本上可以说是基于grpc来构建的。作为服务端和客户端交互的核心，grpc负担了绝大多数客户端对服务端的操作请求，其中包括的rpc如下：

```
service serviceMDSync
{
    rpc login(loginInfo) returns (response);
    rpc getFileInfo(request) returns (fileInfo);
    rpc fileOperation(request) returns (response);
    rpc uploadReq(request) returns (socketInfo);
    rpc downloadReq(request) returns (socketInfo);
    rpc newFileReq(request) returns (socketInfo);
    rpc getFileTree(request) returns (JSONString);
    rpc getTimeStamp(request) returns (JSONString);
};
```

其中每个rpc的功能如下：

- login: 登录功能，将用户信息发送给服务端来执行登录请求；
- fileOperation: 文件操作功能，让客户端能够执行对应操作来对服务端上的文件进行相应管理；
- uploadReq: 上传请求，用于更新已存在于服务端上的文件的内容

- downloadReq：下载请求，用于让客户端获取服务端上的文件的内容
- newFileReq：新文件请求，用于让客户端上传某个在服务端上不存在的文件；
- getFileTree：获取文件树信息，用于让客户端获取服务端上最新的文件树信息
- getTimeStamp：时间戳，用于让客户端来获得服务器当前的时间戳，并以此决定是否要更新本地内容。

可以看到，这些通信及它们的功能共同组成了整个基于服务端-客户端交互的MDSync系统

6. 总结

在开发过程中，我们遇到了许多的困难：

6.1 技术栈的选择

虽然在技术路线中，已经提到我们选择的是Node.js和Electron这些编程框架以及grpc+TCP的通讯方式，但是实际上，我们在启动项目的时候，还是面临了很多的选择的。

比如，最早我们尝试过使用express来做一个网页客户端，同时也尝试过使用tomcat和apache作为后端，但是最后我们还是选择了Electron+Nodejs的技术栈。

6.1.1 通讯方式

实际上，作为跨平台的通讯，grpc绝对不是唯一的选择；实际上，grpc反而是一种比较小众的解决方案。比如，我们可以不使用两种通讯方式结合的方案，而是使用http协议来传输，借由http协议提供的不同方法以及http头中的参数，我们同样能实现之前提到的发送请求，传输文件的功能；并且，假使使用http协议，我们甚至能够用到https协议进行加密，这样就免去了我么在登陆/注册时的加密过程，传输文件的协议也更加安全。

更有甚者，考虑到JSON对象以及其转换的字符串的方法，我们实际上可以用任何支持文本传输的协议来进行通讯；只需要在通讯前将JSON对象字符串化，在接收方再解析这个字符串，也可以得到一个完整的对象，从而完成在不同编程语言之间的通讯。以这种观点来考虑的话，是加上，我们甚至可以直接使用TCP进行通讯，而不需要任何上层协议。

但是最后，我们依旧选择了grpc作为我们的主通讯协议，主要是看中它以下几点：

- grpc的这种远程调用的思路与我们的同步需求非常契合。在大部分时候，是由客户端执行一个动作，然后发送到服务端等待执行的；这样子的通讯正好对应到RPC中的远程调用的思路；
- 使用grpc，服务器的实现比较简明。如果使用传统的通讯协议，我们的处理函数将只有一个，并且根据条件分叉，这就导致服务端的结构耦合度会比较高；而使用grpc,由于每一个调用方法对应着一个函数，我们能够以多个函数的形式来编写客户端，并且不需要考虑这些函数间的互相影响。

6.1.2 语言选择

我们是先决定通讯协议再决定编程语言的；实际上，由于上述的通讯方式可以跨语言来通讯，所以每个端的选择都是自由的。对于桌面客户端，我们实际上没有太多选择；成熟的跨平台GUI框架实际上也就仅有Java，Qt，Electron等几种。考虑到开发难度，最后我们选择了Electron。

而服务端的实现语言就比较多种多样了，同样考虑到最大兼容的问题，我们选择了Nodejs。这个语言可以在有一定性能（重IO的服务端可以被Node的异步机制很好的处理）的同时，在不同架构，不同系统环境的情况下运行。

6.2 客户端框架的理解

Electron作为一个成熟的应用开发框架，有着丰富的相关资料和详细的官方文档，但是因为其多进程+IPC的开发特点，导致我们仍然花费了较长时间来理解如果较好地使用它。

6.2.1 前后端的管理

这里的前后端分别指负责显示的图形界面和在后台进行处理的两部分。开发一个GUI程序，最重要的两件事就是：图形界面的管理显示和后台任务的执行；以及这两种如何关联起来。在electron中这两者分别由html文件和js文件构成，html文件负责显示窗口内的内容，js文件在后台执行，前者称为渲染进程、后者称为主进程，可以看到，在electron中，这两部分内容分别是由两个不同的进程完成的，所以怎样控制图形界面的显示？怎样利用electron提供的API实现图形界面与后台处理的关联来实现特定任务？与服务端的通信需要用到rpc模块，将这种东西放在html文件中实现显然不合适，但相关的操作显然只能在渲染进程上触发，那么如何将渲染进程的触发操作与主进程的相关动作关联起来？这里Electron提供了一套IPC的API，能够让我们在渲染进程和主进程之间相互通信，以此来实现两者的关联。理解这个IPC的工作机制对于基于Electron的客户端开发至关重要。

6.3 服务端实现遇到的问题

服务端遇到的问题主要有三点：Nodejs带来的异步函数的执行顺序问题、访问OS文件系统的问题以及服务端意外退出的处理。

6.3.1 异步函数

异步的问题主要出现在两个方面，一个是访问数据库时，由于大部分数据库提供的接口是异步的，通过回调函数来返回查询结果，这就导致了一部分追求及时性的操作无法使用数据库查询，最后我是换用了一个采用同步接口的`better-sqlite3`来解决的这个问题。另一处困难则是在TCP传输文件时，改变时间戳的时刻很可能发生在文件读写结束前，这会导致同步逻辑暂时失效；这个问题则是将一部分文件操作改为同步，另一部分将改变时间戳调整到文件操作的回调链中执行。

6.3.2 OS文件系统

Nodejs提供的文件系统API虽然强力，但是却有一部分操作是没有实现的，需要自己处理，例如：递归删除文件夹（`rm -r`），移动文件夹等等；如果自己实现这些函数，不仅效率上有问题，还可能带来一些其他BUG，最后我使用了`fs-extra`模块作为`fs`模块的代替，其提供了更加完善的处理函数，解决了这个问题

6.3.3 服务端意外退出

实际上，服务端意外退出是常有的事，而如何处理退出后的数据恢复是比较重要的。对于用户的文件，由于遵循文件操作后再写数据库的操作，基本上不会出现数据库不够新的情况；主要的问题在于一些运行时的系统变量（例如时间戳），这些变量存在于内存中，在系统失效后无处恢复，对此我的解决方案是将这些变量记录为数据库中的一项，利用数据库的独立性来保证这些变量的安全。

6.4 不足与未来

我们当前的项目仍然存在许多不足，比如：

1.过于频繁的异步通信导致客户端和服务端的稳定性较差；

2.客户端与服务端同步的冗余过高，比如文档内容更新是通过发送整个文档的内容来实现的；

未来：

这个项目我们会持续开发下去，在未来，我们会努力提高该应用的可用性，添加更多的功能(如资源文件的自动同步、Markdown文件的管理等等)。