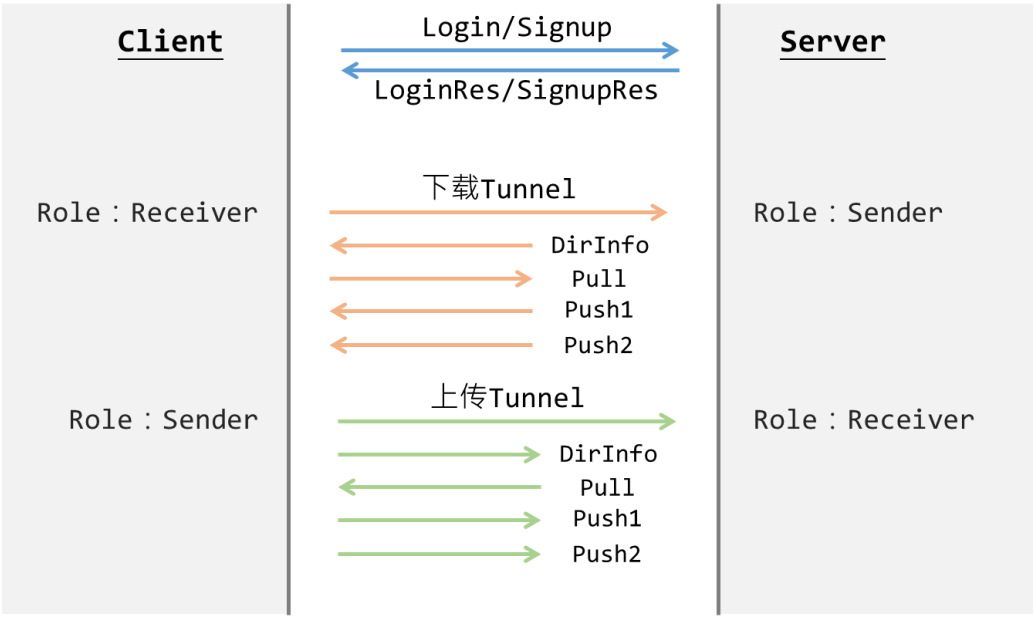
1 工作流程

本网络同步盘在工作时分为三个信道，分别为控制信道、上传信道和下载信道，每一个信道对应一个 TCP 长连接。

一台客户机登陆时使用的是控制信道，登陆成功后客户机又建立两个 socket，发送 Tunnel 包提示服务器信道角色。此后的所有同步过程都发生在这两个信道中。

工作流程如图：



2 通信协议

本同步盘协议有控制、上传和下载三个部分共 8 种消息，全部采用 JSON 编码。\n 代表一换行符，\0 代表一尾零。具体罗列如下

名称	例子	作用
Signup	Signup\n { username: "naeioi", password: "ToOMoOYoO" }\0	客户机注册
SignupRes	SignupRes\n { /* code * 0 成功 * 1 密码太短 * 2 密码太长 * 3 用户名已存在 * 4 用户名太长 */ code: 0, session: "9e107d9d372bb6826bd81d3542a419d6", message: "额外信息" }\0	服务器返回注册结果
Login	Login\n { username: "naeioi", password: "ToOMoOYoO", session: "sdfaf" }\0	客户机登陆
LoginRes	LoginRes\n { /* code * 0 成功 * 1 用户名不存在 * 2 密码错误 */ code: 0, session: "9e107d9d372bb6826bd81d3542a419d6", message: "额外信息" }\0	服务器返回登陆结果
Tunnel	Tunnel\n { /* role * - Tunnel 指令发送端角色 * 0 Sender * 1 Receiver */ role: 0, /* mode * - 同步模式 * 0 初始同步	创建隧道

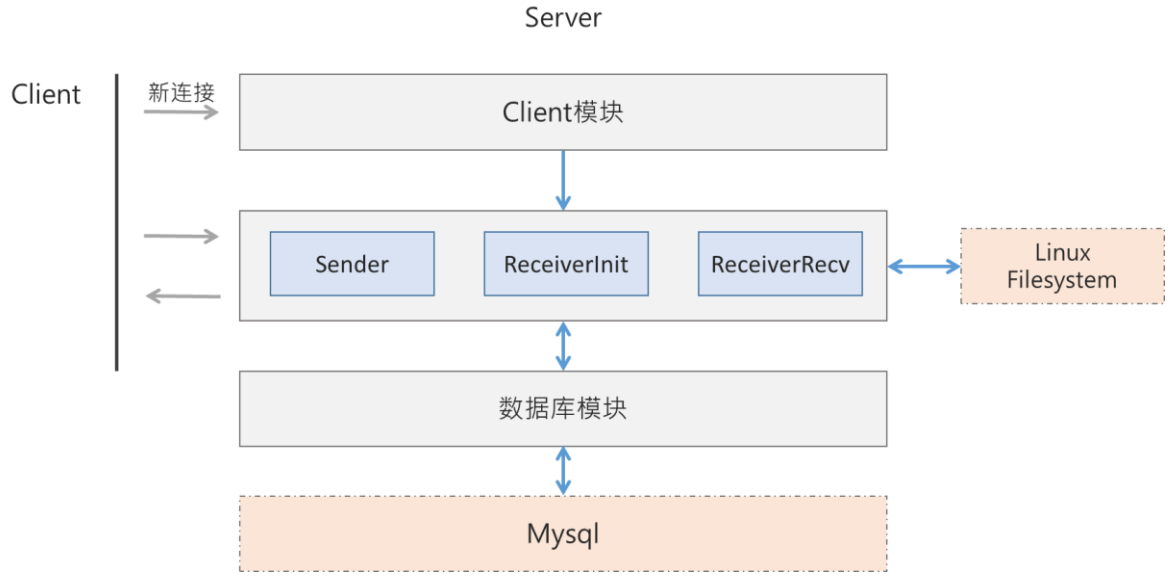
	* 1 增量同步 * 2 恢复同步 */ mode: 0, session: "asdfasdf" }\0	
DirInfo	DirInfo\n [{ "filename": "v.mp4", "path": "/video", "md5": "9e107d9d372bb6826bd81d3542a419d6", "len": 1024, "modtime": "2011-10-08T07:07:09Z", "deleted": 0/1 }, { ... }]\0	发送文件信息
Push	Push\n [{ "filename": "v.mp4", "path": "/video", "len": 1024, "offset": 0 }, { ... }]\0 [Binary Data1] [Binary Data2] ……	发送文件内容
Pull	Pull\n [{ "filename": "v.mp4", "path": "/video", "len": 1024, "offset": 0 }, { ... }]\0	发送文件抓取请求

注：

1. DirInfo 协议中，文件是**压平存储**的，即不包含文件夹嵌套，每个文件在 JSON 数组中有一项。
2. SignupRes 中服务器将返回一随机 session，唯一标识一台客户机，保证不与已有客户机相同。客户机登陆时，若是初次登陆，session 留空。服务器发现 session 为空，则判定这是一台初次登陆的机器，随机生成 session 并在 LoginRes 中返回。

3 服务端设计与实现

本网络同步盘的服务端大致可分为 5 个模块，工作流程如下



Client 负责处理新的连接，并根据客户机发来的第一个包决定信道类型。若发来的是 Tunnel 包，则将这个信道的控制权转交给 Sender 或 ReceiverInit/ReceiverRecv 模块。其中 Sender 模块用于服务器给客户机发送文件，两个 Receiver 模块用于客户机上传。此处，ReceiverInit 模块与 ReceiverRecv 模块唯一的不同在于，ReceiverInit 用于初次上传，遇到同一文件夹下的冲突文件会更名，而 ReceiverRecv 遇到冲突文件会选择保留修改时间较新的那一个。

3.1 数据库设计

数据库共有 3 张表。分别是用户表 User，文件表 Files 和文件索引表 Inode。具体如下表所示。

表名	字段名	类型	说明
User	Uid	Int(11)	用户编号
	cid	Char(32)	客户机标识
Files	Uid	Int(11)	文件的用户编号
	Filename	Varchar(56)	文件名
	Len	Int(11)	文件大小(单位:Byte)
	Path	Varchar(128)	路径名
	Md5	Char(32)	MD5 码
	Complete	Tinyint(1)	完成标识
	Modtime	Datetime	修改时间
	Chunks	Mediumtext	已完成 chunks 记录
Inode	Md5	Char(32)	文件 md5
	Refcount	Int(11)	文件引用数

其中 files 表的 chunks 字段使用 JSON 记录了一个上传文件的传输状态。一个已上传完的文件，chunks 为 NULL。未上传完的文件，Chunks 记录了已上传的文件块。结构如下所示：

```
[
    { "offset": 0, "len": 1024 },
    { ... },
    ...
]
```

Chunks 是一个 JSON 数组，使用 offset 和 len 表示已上传的块的偏移和长度，单位均为字节。

3.2 Sender 模块

服务器端的 Sender 逻辑较为简单。它首先将服务器上的文件信息打包成 DirInfo 包发送给客户机，并等待客户机发来的文件抓取(pull)请求，然后根据 pull 信息推送文件(push)。

在本项目中，Push 包和 Pull 包是可以包含多个文件的。Sender 模块处理 Pull 请求时，对一个大文件的 Pull 会分割发送，对多个小的 pull 发送，使得每个 Push 包控制在 1M 左右。分割发送保证了大文件断点续传功能的实现，小文件合并发送减少了带宽消耗，降低了延迟。分割、合并功能由 Chunkify 函数实现。

3.3 Receiver 模块

相对 Sender 模块 Receiver 模块要复杂得多。Receiver 从 Client 模块接管信道后，按照以下步骤拉取客户机发来的文件。

(1). 等待客户机发送 DirInfo
(2). 决定要抓取哪些文件，生成 Pull 包
(3) 发送 Pull 包
(4) 接受 push 包
(5) 根据 push 包写文件、数据库
(6) 跳转到(4)，直到对方关闭信道

其中(2)生成 pull 包的逻辑又分成两趟。

a) 将 DirInfo 与服务器 files 表进行比对，决定哪些需要抓取，生成待抓取文件清单 pullreq
b) 将 pullreq 与服务器 Inode 表进行比对，查看哪些文件服务器已有，已有的文件从 Pullreq 中剔除，直接更改 files 表，并把 inode 表中该文件的 refcount 加一。该步骤实现了文件秒传

在本项目中，Push 包是不含 MD5 码、修改时间等信息的。这些信息在 DirInfo 中已经发送过了。因此这些信息要提前保存，以便 Receiver 收到 push 包时能够调取。对于 pullreq 的每一项需要做一个从<path,filename>到 Pullreq 的映射。我们采用了 C++的 map。

接收 Push 包时，首先将文件二进制数据写入保存文件的 store 目录中。文件名是文件的 md5 码，写入成功后改写 files 表，将收到的块与 chunks 字段中的块合并。当判断文件完全写入时，向 inode 表写入该文件的信息，并置 files 表该文件的 complete 的字段为 1。

完整的 Receiver 逻辑如下所示

```
接收 dirinfo

vector<file> toPull;

for f in dirinfo
    if 相同文件 (f.local, f.peer)
        if( 增量同步 || 恢复同步 ) {
            if( f.peer 修改时间迟于 f.local )
                将 f.peer 加到待 pull 序列 toPull
        } else { // 初次同步
            if( f.peer 修改时间迟于 f.local ) {
                将 f.peer 加到待 pull 序列 toPull
                修改 files 表，将 f.local.filename 改为 f.local.filename.old
            }
            else { // 服务器文件更新
                // 客户机保证这种情况不发生
                // 客户机发现服务器上的文件更新时首先将自己的文件重命名
            }
        }
    else
        将 f 加到待 Pull 序列 toPull

json pullreq;
json files;
for file in toPull
    if( inode 中存在 file.md5 )
        修改 inode 表，令 file.refcount 加一
        修改 files 表，加入名为 file.filename 的文件，complete 为 0  else
        将 files 表中 file 的信息存入 files[filename][path] 中（主要是要使用 chunks）
        取出 file.chunks, 存入 // 有可能是 NULL
        根据 file.chunks 将 file 未完成部分加入 pullreq

发送 pullreq.dump()

while(1)
    接收 PushReq 头部
    for file in PushReq
        接收 file.len 字节的 file.buffer
        写入文件 ./data/chunks/(file.md5).tmydownload
        更新 files[filename][path].chunks，合并相邻项
        if(收取了整个文件)
            将文件从 ./data/chunks 移动到 ./data/store
            修改 inode 表，新增文件 file.md5
            修改 files 表，置 complete 为 1，chunks 清空
        else
            更新 files 表中的 chunks 字段
```

4 客户端设计与实现

本网络同步盘的客户端 UI，采用 QT 编写，以方便跨平台移植。各平台处理逻辑均一致，仅在程序实现上由于平台差异，有所区别。

下面分模块阐述客户端设计。

4.1 各模块简介

Logger：

静态使用的类，程序运行之初对其进行初始化，会根据当前时间生成一个 log 文件。客户端其他部分所有部分调用 log 都会发送给此类，此类再分发给三部分——窗口的 log 显示、log 文件以及 qDebug 的输出。考虑到多线程，写文件部分使用了互斥锁。

LoginWindow：

程序最初的界面，运行之初即读取本地配置文件，建立与服务器的登陆、注册用通道。
登陆过程中出现的错误在服务器发送回来的登陆结果包中。
可以转移至 MainWindow 或者 SignUpDialog。

SignUpDialog：

负责注册的对话框，在本地进行了密码强度的检测，其他一切错误由服务器发回的注册结果包提供。

关闭可转移回 LoginWIndow。

MainWindow：

转移至此窗口后，首先读取本地配置文件，如果已有绑定目录，立刻生成 ReceiverThread 和 SenderThread 两个工作线程，再让两个工作线程进入恢复同步状态。若没有，在绑定目录后再生成两个工作线程，并让它们进入初次同步状态。

提供解除绑定功能，会立刻终止两个工作线程，等待再次绑定。

此外，本窗口还提供了 log 的显示，以及当前下载与上传文件的基础信息与进度条。这些信息由两个工作线程负责传递。

ReceiverThread：

“接收者”线程，开启后首先生成 Receiver 对象，进入死循环，等待主线程给的进入初次同步或者恢复同步流程的通知。初次同步和恢复同步流程仅在生成文件的细节上有所不同。
开始同步后，用 Receiver 接收服务器发来的 DirInfo，然后对于 DirInfo 中的每一项，获取本地该文件信息，若存在，进行 md5、修改时间等进一步比较，若没有，则创建新的，并且创建一个空的 metafile。

随后开始接收文件流程，对于每个待接收文件，生成 LocalFileWriter，由其生成 PullReq，送给服务器之后，等待接收服务器的 PushReq，然后再由 LocalFileWriter 写入文件。全部写入后接受流程结束。

若是初次同步，结束后改变自身状态至恢复同步，并且每 5s 执行一次。

SenderThread：

“发送者”线程，开启后首先生成 Sender 对象，进入死循环，等待主线程给的进入初次同步或者恢复同步流程的通知。而其实 Sender 在初次同步和恢复同步上并无不同。

开始同步后，首先生成 DirScanner，扫描本地文件夹，生成 DirInfo，由 Sender 发送给服务器，随后等待服务器发送来的 PullReq。如果 PullReq 为 null，则同步流程结束。若不为空，则对于每个 Pull，生成 LocalFileReader，读取特定数据段，生成 PushReq，整理后发送回服务器（为了合并小包以加速，每次生成 PushReqEntry 后加入待发送的 PushReq 队列中，一旦队列超过一定阈值（程序中设定为 1MB）即立刻发送）。全部发送完毕后结束同步流程。同步流程结束后，还会结合发送的 DirInfo 和 PullReq 来判断是否秒传。

若是初次同步，结束后改变自身状态至恢复同步，并且每 5s 执行一次。

LocalFileWriter：

负责生成文件的 Pull 信息，以及将 Push 包的数据写入文件。生成 Pull 信息需要根据该文件对应的 metafile，此处会假定需要接收的文件的 metafile 已经由 ReceiverThread 全部生成，如果没有 metafile，即认定其已经接收完毕。打开 metafile 后，读取其中的信息至 Chunks 中，根据 Chunks 来生成 PullReq。

如果需要写入文件，首先要调用 SetFile 转移状态至那个文件，包括将上个处理的文件的 Chunks 写入 metafile，以及读取新文件的 Chunks。然后对于每个 Push 包，在写入文件之后，用插入排序的方式将新的 Chunk 插入进 Chunks 中，随后立即进行 Chunk 合并，即两个 Chunk 若首尾相连即合并。更改 Chunks 至一定次数后，写入 metafile 中。

LocalFileReader：

负责读取文件的某段数据，根据 Pull 包来读取特定字段的数据并打包成 Push 包，仅此而已。

DirScanner：

本地文件夹扫描器，用递归的方式扫描本地文件夹，随后将所有文件的路径信息压扁存入 DirInfo 中输出。

4.2 各模块依赖