基于 Rust 和 WebAssembly 的分布式文件系统

雷雨轩 裴启智 刘逸菲 孙一鸣 曲阳

一. 项目背景

分布式文件系统可以有效解决信息爆炸时代存在的成指数倍增长的数据的存储和管理难题：将固定于某个地点的某个文件系统，扩展到任意多个地点/多个文件系统，众多的节点组成一个文件系统网络。每个节点可以分布在不同的地点，通过网络进行节点间的通信和数据传输。人们在使用分布式文件系统时，无需关心数据是存储在哪个节点上、或者是从哪个节点从获取的，只需要像使用本地文件系统一样管理和存储文件系统中的数据。

通过与老师沟通以及对于各个文件系统的考察后，我们发现当前业界对于分布式文件系统的追求无非就是几个方面：

- 搭建部署系统操作是否繁复
- 传输和储存文件是否安全可靠
- 文件系统性能是否满足使用要求
- 对计算机的性能要求是否高
- 对文件的格式是否有限制

而具体到不同的应用场景下，对于以上的要求则各有取舍。

结合对往届项目的实现的研究，我们计划并完全实现了这样一个家庭式分布式文件系统：

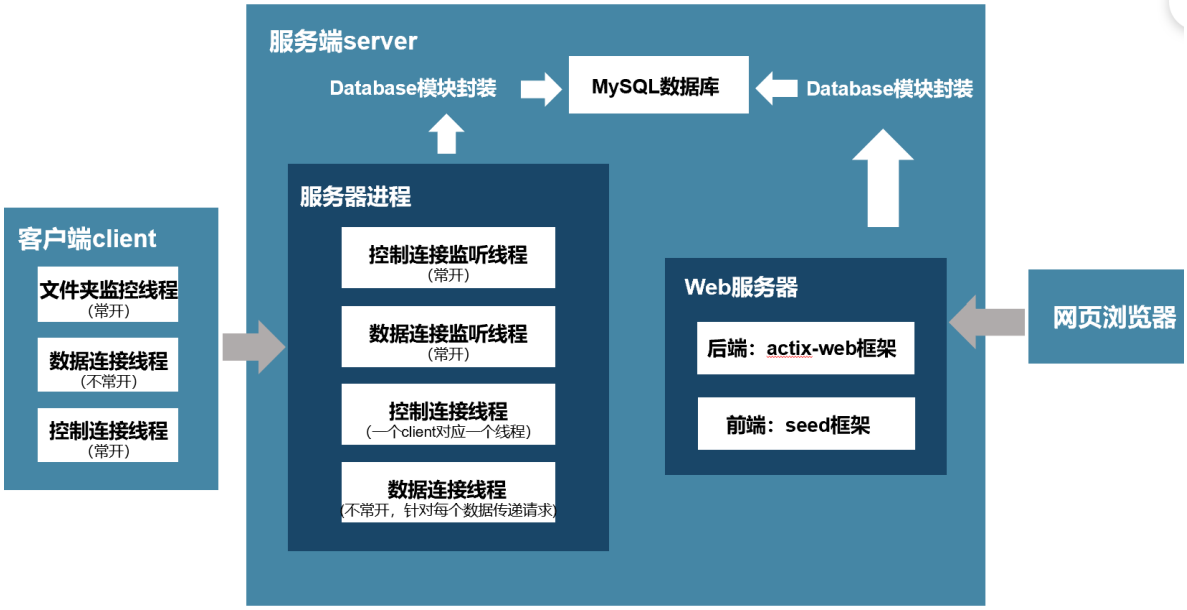
- 高效利用分散在不同设备上的存储空间，方便多人合作办公，避免资源浪费
- 所有安装了客户端的机器共享、贡献存储空间和存储文件碎片，并对数据做好高效的备份；一个专门用于协调处理请求，维持系统状态的服务端；
- 有浏览器的电脑都可访问分布式文件系统
- 在以上要求外，尽可能让性能相对好，安全性相对高，操作简单

在调研报告和可行性报告部分，我们已经对于我们实现思路做了分析。现在首先简要概括本项目的完成情况：

在原基于互联网的小型分布式文件系统项目的基础上，小组成员齐心协力，学习rust语言并完成了client、server端的完全改写，实现了整个分布式文件系统的逻辑架构。此外，采用前后端分离的模式，采用actix-web框架将web应用的后端用rust语言改写，并与原项目的web前端衔接，最终呈现出完整的分布式文件系统，并在浏览器端提供了用户对文件系统的管理操作。

二. 总体设计架构

2.1 系统结构



系统结构如上图。程序包括客户端程序、服务端程序、web 服务程序三部分。其中，客户端程序运行在客户主机（即存储端）上，服务端程序与web服务程序运行在服务端server上。通过网页浏览器访问该分布式文件系统，无需运行其他程序。

- 客户端
    - 文件夹监控线程（常开）

      对应 client 中的 FolderScanner 模块。该线程用于每隔一段时间扫描一次共享文件夹，查看共享文件夹中是否有新文件需要上传至服务器。
    - 控制连接线程（常开）

      对应 client 中的 SeverConnecter 模块。该线程用于每隔 5 秒向服务器发送一次报文，通过心跳连接的方式确保客户端与服务器的长期连接。同时，该线程还负责处理服务器发回至客户端的控制报文。
    - 数据连接线程

      该线程用于必要时与服务器交换数据。

- 服务器进程
  - 控制连接监听线程（常开）

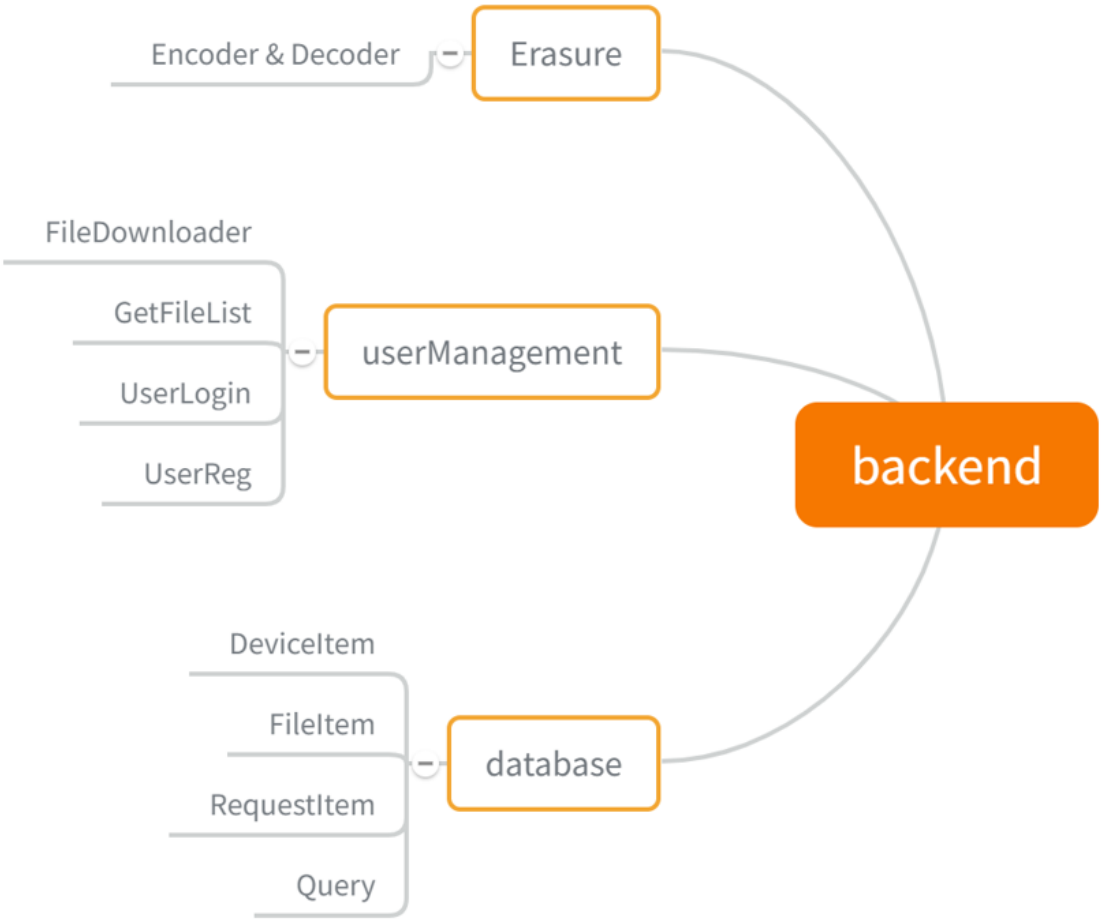
    对应 sever 中的 ControlConnect 中的 SeverThread 模块。该线程用于监听服务器的控制端口，若有客户端发来请求/连接，其将创建一个控制连接线程处理该连接。
  - 控制连接线程

    对应 sever 中的 ControlConnect 中的 ClientThread 模块。每个控制连接线程对应一个客户端，该线程用于读取并处理客户端通过控制端口发来的报文。
  - 数据连接监听线程（常开）

    对应 sever 中的 DataConnect 中的 SeverThread 模块。该线程用于监听服务器的数据端口，若有客户端发来请求/连接，其将创建一个数据连接线程处理该连接。
  - 数据连接线程（不常开）

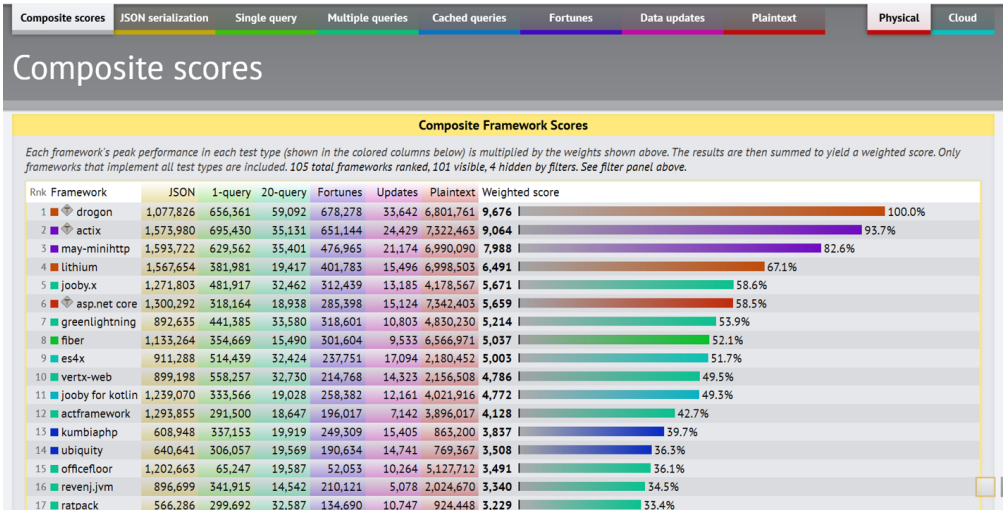
    对应 server 中的 DataConnect 中的 ClientThread 模块。该线程用于读取并处理客户端通过数据端口发来的报文。
- web 服务器

  考虑前后端分离的方式来完成整个web应用的设计，便于调试，同时便于各平台兼容。对后端而言，不提供任何和界面表现有关的内容，只需要接收前端的http请求并返回相应数据即可。为了统一数据格式，数据传输一律采用Json格式。



  - 后端
    - Actix web 是小型的、实用、快速 Rust web 框架。
    - 支持 Client/server WebSockets
    - 包含一个异步的 HTTP 客户端
    - 强大的请求定向

- 使用稳定的 Rust 版本（如 Rocket 需要 nightly 版本的 Rust ）
- Actix-web 框架是 Rust web 框架中性能最高的。它依靠 tokio 作为异步运行时，适于对性能有严格要求的，大流量 web 应用。
- 性能测试图:



参数	actix-web	rocket(async)	rocket(sync)	hyper(debug)	hyper(release)
qps	143281	113878	21117	69303	86202
Concurrency	100	100	50	100	100
Transfer rate（kps）	21408	18905	3010	7580	9428
Time per request（ms）	0.698	0.878	2.368	1.443	1.160

  - 前端

    前端主要借鉴了原项目与我们的Rust后端交互，对其中的文件进行了一定的删减和修改，并优化了界面，添加了下载的用时统计。

    采用的布局主要如下：

    - Tomcat Web应用服务器
    - Struts2 动态网站网站应用调度框架
    - BOOTSTRAP网页主题
    - JQery+AJAX异步C/S通讯+JSON+MySql

2.2 技术路线

- 数据库：按照数据结构来组织、存储和管理数据的仓库。是一个长期存储在计算机内的、有组织的、可共享的、统一管理的大量数据的集合。
  - MySQL：一种关系型数据库管理系统，关系数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。其 使用的SQL 语言是用于访问数据库的最常用标准化语言
  - 通过 MySQL 数据库来记录储存信息，比如文件和碎片的位置和属性等（具体信息参见部署文档），并通过数据库中的 Request 表单处理请求，达到在 server、client 和 web 端之间交互的功能。
  - Mysql中的表

数据库中有DEVICE,FRAGMENT,FILE,REQUEST,USER 五张表。

DEVICE 表登记存储端设备的ID,IP地址,端口号,是否在线，剩余空间等信息。

FRAGMENT 表登记碎片的ID和路径。

FILE 表登记分布式文件系统中的文件名称、文件逻辑路径、 属性、修改时间、分成的文件碎片数量、是否为文件夹等。

REQUEST 表登记请求的ID、请求类型（1服务器分块，2服务器将分块发送给客户端，3删除客户端上的分块）、需处理此请求的设备ID、需被处理的碎片ID等。

USER 表登记用户的姓名和密码。

- 客户端与服务器的通信

  客户端与服务器间通过报文通信，具体见部署说明文档。

  - 控制连接报文

报文	内容	含义
客户端状态报文	1 {设备 ID} {客户端剩余空间}	客户端的心跳报文，每隔 5 秒发送一次
处理请求报文	3 {设备 ID}	当客户端发现自己有未处理的请求时发送，服务器将回复等待其处理的请求的具体 内容。

  - 数据连接报文

报文	内容	含义
客户机碎片上传报文	1 {请求 ID} {碎片 ID}	
客户机碎片下载报文	2 {请求 ID} {碎片 ID}	从服务器下载其指定的碎片
客户机碎片删除报文	3 {请求 ID} {碎片 ID}	通知服务器客户端已经删除了其指定的碎片
上传文件报文	4 {设备 ID} {文件名} {路径} {属性} {碎片数量} {是不是文件夹}	申请向分布式文件系统中上传文件
文件碎片上传报文	5 {文件 ID} {碎片序号} {碎片总数}	在客户机发送了上传文件报文并收到了服务器回复的文件 ID 后，需要使用此报文上传该文 件的全部碎片

报文	内容	含义
检查并新建文件夹报文	6 {设备 ID} {文件夹数量} {文件夹路径} {文件夹名称} （每行一个文件夹信息）	当客户端启动时，其需要使用此报文保证被监控的文件夹对应的逻辑位置在服务器上有记录。当服务器收到此报文后，如发现该逻辑位置在服务器上没有记录，先新建一条对应的记录

- Rust工具链

工具	作用
cargo	包管理工具
cargo-fmt	源代码格式化工具
rutsc	编译器
rustfmt	源代码格式化工具
rust-gdb	调试器
rustup	管理工具链
rustdoc	文档生成器

- 纠删码：
  - 一种编码技术，相比于副本策略，纠删码具有更高的磁盘利用率。
  - 主要是通过纠删码算法将原始的数据进行编码得到冗余，并将数据和冗余一并存储起来，以达到容错的目的。其基本思想是将n块原始的数据元素通过一定的计算，得到m块冗余元素（校验块）。对于这n+m块的元素，当其中任意的m块元素出错（包括原始数据和冗余数据）时，均可以通过对应的重构算法恢复出原来的n块数据。生成校验的过程被成为编码（encoding），恢复丢失数据块的过程被称为解码（decoding）。磁盘利用率为n/(n+m)。基于纠删码的方法与多副本方法相比具有冗余度低、磁盘利用率高等优点。
- Actix-Web：用于Rust的功能强大、实用且快速的Web框架

三. 学习记录

3.1 rust 教程及 crate 文档

项目初期，我们认真阅读了 rust 的官方教程 ，初步了解了 rust 语言的构成、概念、特性与优势。在 client 和 server 的代码改写阶段，为寻找与原项目中使用的 java 库函数对应的 rust 函数，我们阅读了大量的 crate 文档，例如：mysql , reed-solomon-erasure 等。在改写过程中，我们深刻体会了 rust 在线程并发、生命周期及所有权方面的优越特性。代码调试的过程更是锻炼了我们的调试能力，帮助我们对 rust 的错误处理机制有了更深层次的理解。

3.2 数据库

首先，我们基于廖雪峰的 SQL 基础教程学习了数据库的基本知识，学习了有关关系模型、查询数据 SELECT 语句、修改数据 UPDATE 语句、删除数据 DELETE 语句、增加数据 INSERT 语句和具体使用 MySQL 的方法等等。而后我们搜索了 Rust 中的 MySQL 使用，找到了Rust 文档中的 MySQL 官方库，在学习了文档中的示例之后，我们首先使用示例进行了测试，而后仿照示例完成了我们的改写代码，在实际测试中也完善了错误处理，最后成功实现了数据库模块的功能。

3.3 WebAssembly

- 我们原计划将 Rust 应用实例转化为 WebAssembly，调用相应的库并利用 npm 对项目打包，将其部署到 Node.js 上运行。

  运行并研究了应用这套工具链的示例项目 wasm-game-of-life 。

  - 工具：Rust toolchain (rustup, rustc, cargo), wasm-pack, cargo-generate, npm
  - 简化的过程：
    - 克隆 wasm-pack-template 模板，在 /src/lib.rs 中编写 Rust 函数和方法，实现需要的功能。
    - 运行 `wasm-pack build` ，生成 /pkg 文件夹，其中包括 wasm 二进制文件、用于 Rust 和 Java 互相调用的 JavaScript 胶水等。
    - 运行 `npm init wasm-app example` ，生成 /example 文件夹，其中有 Web 页面的入口文件。
    - 运行 `npm install` 安装依赖项, `npm run start` 执行项目。在默认的 localhost:8080 中可以访问项目。
- 在进一步的调研中，我们发现 WebAssembly 的使用有诸多限制，将分布式文件系统整个项目编译为 WebAssembly 运行是不可行的。

  可能导致 Rust 不能与 WebAssembly 共同工作的特性：

  - C 库及系统库的依赖
  - 文件 IO：WebAssembly 中没有关于文件的接口，不能使用文件系统。如果 crates 需要对文件的操作，又没有 wasm 特定的工作区，那么它就不能工作。
  - 多线程：WebAssembly 本身不支持多线程。目前有相关工作，但如果需要在 WebAssembly 中使用多线程，仍需要自行实现很多底层的东西。

可能适宜与 WebAssembly 共同工作：

- 提供特定算法和数据结构的 Crate
- 处理文本的 Crate

  对于分布式文件系统来说，文件 IO 是不可能避免的；其次，我们在项目中也使用了很多 std::net, std::io 等 std 库内容；关于 web 端多线程的使用也是一种可能的考虑。

  综上所述，将整个项目编译为 WebAssembly 在网页端运行是不可行的。但纠删码编解码的过程可以视为一种对文本的处理，对纠删码部分应用 WebAssembly 是我们下一步的目标。

3.4 HTML

- 为了设计前端页面，我们学习了HTML语言的基本用法。HTML 是用来描述网页的一种语言，Web 浏览器的作用是读取 HTML 文档，并以网页的形式显示出它们。

3.5 JavaScript

- 常用来为网页添加各式各样的动态功能,为用户提供更流畅美观的浏览效果。通常JavaScript脚本是通过嵌入在HTML中来实现自身的功能的。
- 我们对前端使用的JS进行了一些改写，来和我们的Rust后端进行交互。

3.6 rust实现web应用的四个框架

考虑到利用WebAssembly编译高级程序语言运行在浏览器的优势，以及想要最终实现完全由Rust完成的分布式文件系统(即项目整体部署上的统一)，小组也对Rust实现web应用做了很多调研，具体从前后端框架文档及其开源项目做了大量学习与调研，对于前后端的实现有了一定规划，并在大作业中完成了Rust后端，与原项目前端成功衔接。

- 后端框架学习：
  - Actix-web：相对轻量，只提供基本的HTTP Web服务器，路由逻辑，中间件基础设施，基本的构建模块和抽象，从而解析、操作和响应HTTP请求。
  - Rocket：内置了ORM集成，可以管理和配置流行数据库，可满足构建坚固的Web应用程序时的日常需求

  两者的一些核心概念比较相近，都是从Routing，Mounting，以及Requests和Responses四个框架核心概念入手来讲解最简单的后端实现，然后才是诸如Cookies，state，Guard，Middlewares，database等一些高级概念实现。

  考虑到Actix-web异步性能更优，且database模块我们已经计划自己完成封装，所以最后选择了更轻量的Actix-web
- 前端框架学习：
  - Seed：Seed 是采用类似 elm 结构的 Rust 前端框架，同样在网页前端结合应用 Rust 和 WebAssembly 。它极少地需要额外费用、配置和样板文件。
  - Yew：受Elm和React启发的前端框架启发，并可以将Rust编译到WebAssembly，同时与JavaScript有良好的互操作性

  两者均是采用Elm结构的Rust前端框架，并采用宏的形式完成对于html的实现。相比于Yew组件的概念，Seed把各个板块拆分更全面，从Model，Msg，View，Update四个结构开始，通过不断细化各个结构来完成最终的前端页面。

  考虑到我们 Rust 全栈开发的目标和对项目稳定性的需求，我们决定选择 Seed 。

四. 模块改写分析

4.1 client

- client

  client和SynItem模块：

  - 主要功能是读取配置文件后对各个参数作一个初始化。然后会开启控制连接线程与server端相连，开启文件夹监控线程每隔一定间隔将会扫描文件夹看是否有新的文件要上传到文件系统。
  - 实现方面：与原来模块的一个区别是：原来java文件利用了synchronized关键字以及SynItem类来作为状态量以实现锁机制，最终实现线程间的同步（当子线程出现错误时，唤醒沉睡的主线程并结束所有子线程）

    但在rust中，本身没有类这个概念，锁机制的使用也有所不同，所以考虑直接创建带有条件变量的锁,并把锁分别clone到folderScanner和ServerConnecter两个线程；

```
//线程创建
    let status = Arc::new((Mutex::new(0), Condvar::new()));
    let connect_status = status.clone();
    let fileDetector_status = status.clone();//Arc<Mutex<i32>,Condvar>

    //let clientid = self.clientId.clone();

    let handle1 = thread::spawn(move || {
```

```
        let mut ServerConnecter =
crate::client::connect::ServerConnecter::ServerConnecter::new(clientId)
;
        ServerConnecter.run(connect_status);
     });//let mut num = counter.lock().unwrap(); *num += 1;


    let handle2 = thread::spawn(move || {
    let folderScanner =
crate::client::fileDetector::FolderScanner::FolderScanner::new(uploadFo
lders,uploadAddrs);
    folderScanner.run(fileDetector_status);
    });
```

然后主线程调用wait进入睡眠；由于该条件变量的存在，当子线程出错时，则子线程会互斥的修改该条件变量，再唤醒wait中的主线程，主线程检测到状态值改变，则会输出相应报错信息，最后终止所有client的线程。这样一来SynItem类就不再需要了

```
//主线程进入wait
let &(ref lock, ref cvar) = &*status;
let mut status_cur = lock.lock().unwrap();
while *status_cur==0 {//状态码未被改变时，则继续wait
    println!("before wait");
    status_cur = cvar.wait(status_cur).unwrap();
    println!("after wait");
}
```

```
//子线程出错时唤醒主线程
let &(ref lock, ref cvar) = &*status;
let mut status_cur = lock.lock().unwrap();
*status_cur = 2;
cvar.notify_all();
println!("notify main thread");
```

- com

  该模块使用 erasure code 算法对文件分块，具体采用 rust 的 crate reed-solomon-erasure（"4.0"）进行碎片构建。其中，Encoder 模块负责初始文件的编码与分片，Decoder 模块负责根据碎片还原出初始文件。

  - Encoder 模块
    - 功能

      该模块用于读取文件内容，然后将其整理为二维vector，以便使用 crate 中提供的 encode 方法进行编码，最后将结果写入文件碎片中。

      参数说明：inputFile_Path为要被分块的文件的路径；shardsFolder为存放文件碎片的文件夹位置，fid为文件的ID。

    - 实现时遇到的问题
      - 二维数组

        原项目的 java 代码中，通过 `byte [] [] shards = new byte [totalShards] [shardSize];` 一句声明了一个大小由变量 totalShards 和 shardSize 规定的二维数组，但 rust 的相关多个教程中均未提到二维数组的概念，查询许多文档、尝试多种实现后，最后选择使用 `Vec<Vec<u8>>` 这种灵活的结构实现二维 vector。

```
let mut shards:Vec<Vec<u8>> = vec![vec!
[0;shardSize.try_into().unwrap()];totalShards.try_into().unwrap
()];
```

另外，原 java 代码中，使用 buffersize 声明一维数组 allBytes 的大小，用于暂存文件大小与文件内容。在 rust 中也改为使用灵活的 vec 结构实现，这种改动也为使用 `struct std::fs::File` 结构的 `fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize>` 方法读取文件内容提供方便。

```
//java
final int bufferSize = shardSize * dataShards;
final byte [] allBytes = new byte[bufferSize];
```

```
//rust
let mut allBytes:Vec<u8> = Vec::new();
inputFile.read_to_end(&mut allBytes);   //append
```

原 java 代码中使用 `void java.lang.System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` 函数实现从一维数组 allBytes 到二维数组 shards 的拷贝，在 rust 中选择使用 vec 的 push 方法将每个字节依次放入 shards 中。

```
if dataShards > 1{
    for i in 0..(dataShards-1) as usize {
        &shards[i].clear();
        for j in 0..shardSize.try_into().unwrap(){
            &shards[i].push(allBytes[((i as u32) * (shardSize
as u32) + j) as usize]);
        }
    }
}
```

此外，rust 要求 vector的索引为 usize 类型，因此多处使用 as 进行类型转换。

注：shards 中的前四个字节为文件大小 fileSize

- Decoder 模块
  - 功能

    该模块用于读取多个文件碎片内容，然后根据碎片文件名的编号将其整理为二维 vector，以便使用 crate 中提供的 reconstruct 方法重建所有碎片，最后利用数据碎片还原初始文件。

    参数说明：shardsFolder为碎片所在的文件夹路径；decodedFile为要复原的文件路径；fid为文件ID；noa为文件碎片的数量。
  - 实现时遇到的问题
    - 构造各文件碎片对应的路径

    在已知 fid （文件 ID）与 noa （文件碎片数）时，原 java 代码通过 File 类的构造方法 `java.io.File.File(File parent, String child)` 构造各文件碎片对应的 File 实例。

```java
//java
File shardFile = new File(shardsFolder, Integer.toString(fid * 100
+ i));
```

在 rust 中，使用 `std::path::PathBuf` 结构的 `pub fn join<P:AsRef<Path>>(&self,path:P) -> PathBuf` 方法构造各文件碎片对应的路径。

```rust
//rust
let pathbuf = shardsFolder.join(Path::new(&(fid * 100 +i as
i32).to_string()));
```

- connect
  - 功能：与服务器通讯并进行文件传输。
    - FileTransporter
      - 功能：文件传送模块，负责与 Server 间使用 TCP 协议发送和接收文件。
      - 实现时遇到的问题

      Java 是面向对象语言，FileTransporter 模块原代码中定义了一个没有字段只有方法的抽象类；而 Rust 并非面向对象语言，因此不再定义 FileTransporter 类，直接定义 send_file, recv_file 两个函数。
      又考虑到 Rust 标准库与 Java 区别，结合本文件功能，确定 DataInputStream, DataOutputStream 类型对应参数改为 TcpStream 。再由于 Rust 所有权规则，直接使用 TcpStream 类型参数，会在函数返回时结束该变量的生命周期，因此改为 &TcpStream 。

      ```
      pub fn recv_file(mut f: File, mut soc_in: &TcpStream)->bool{
        ...
      }
      pub fn send_file(mut f: File, mut soc_out: &TcpStream)->bool{
      ...
      }
      ```

    - FragmentManager
      - 数据连接模块，维护客户端上的文件碎片，负责处理数据链接中的客户机碎片上传报文、客户机碎片下载报文与客户机碎片删除报文。
      - 调动FileTransporter中的函数进行文件碎片的传输
    - ServerConnect 控制连接模块，根据心跳连接机制维护客户端和服务端连接。

      客户端每 5s 发送报文 “1 x y”，1 是标志，x 是客户端ID，y 是客户端剩余存储空间。客户端每发送一次标志为 1 的报文，服务端会在数据库中根据客户端 ID 查询客户端未读请求，返回请求数给客户端，客户端会收到报文 “received with x unread request” 。

    Java java.net.Socket 库对心跳连接有很好的支持，只要设置 setKeepAlive(true) 以及心跳连接时间即可实现。 Rust std::net::TcpStream 没有这样的支持，因此心跳连接的实现更加底层。
  - 实现时遇到的问题

    考虑到 Java 是面向对象语言，FileTransporter 模块原代码中定义了一个没有字段只有方法的抽象类；而 Rust 并非面向对象语言，因此不再定义 FileTransporter 类，直接定义 send_file, recv_file 两个函数。另外，由于 Rust 没有 DataInputStream, DataOutputStream 类，我们在考察 FileTransporter 模块作用后将函数参数类型更改为 TcpStream ，并删去了多余的参数。
- fileDetector

- FileAttrs
  - 功能

    定义文件属性信息结构体模块，其中有 name，path，attr 和 noa 信息。
- FileUploader
  - 功能

    createConnection 负责使用 serverIP 和端口号创建与 server 端的 TcpStream 连接；checkFolders 负责向 server 端发送需要监察的文件夹，以报文 “6 0 ”开头，而后发送所有文件夹路径；registerFile 负责将文件信息发送给 server 用来后续在数据库中进行记录，报文为 “4 0 {name} {path} {attr} {noa} false”；pushFragment 负责调用 send_file 函数发送碎片，报文格式为“5 {fileId} {fragmentNum} {fragmentCount}”。
  - 实现时遇到的问题
    - 同样使用 Rust 中的 TcpStream 类型代替实现了 Java 中使用的 DataInputStream 和 DataOutputStream。
    - 使用 PathBuf 替代 java 中的 Path 和 File 类。
    - Java 中使用了 lastIndexOf 函数来找到字符串中的最后一个 /，Rust 实现时使用了循环来找到最后一个字符。而后针对不同情况分别发送路径到服务器。

```
while i < addr.len() {
            let c = addr[i].chars();
            let mut j = -1;
            let mut n = 0;
            for cur in c {
                  if cur == '/' {j = n;}
                  n = n+1;
            }
            if j==-1
                 {socket.write_fmt(format_args!("/ {} ",
&addr[i]));}
            else {
                 let mut number = 0;
                 let ch = addr[i].chars();
                 for cur in ch {
                       socket.write_fmt(format_args!("{}\n",
cur));
                       if number == j
{socket.write_fmt(format_args!("/ "));}
                       number = number + 1;
                 }
                 socket.write_fmt(format_args!("\n"));
            }
            socket.flush();
            socket.write_fmt(format_args!("\n"));
            socket.flush();
            i = i + 1;
}
```

- FileUtil
  - 功能

    工具方法模块。提供清空文件夹 clearFolder 方法和文件夹遍历getAllFiles 方法。
  - FolderScanner

- 功能

  在客户端启动模块（client）中，客户端创建文件夹监控线程执行 FolderScanner 的 run 方法。执行时，先调用 FileUploader 模块中的 checkfolder 方法与服务器连接，并向服务器发送新建文件夹报文（报文标志为6），以确保被监控的各文件夹在分布式文件系统中的对应逻辑位置存在。之后，每隔 interval_mills （设置为1分钟）时间，检查客户端的共享文件夹中是否有新文件。若有，则通过 handleFile 方法获取文件的属性，并调用 FileUploader 模块中的 registerfile 方法向服务器注册这个文件。若注册成功，则调用 文件分块模块的 Encoder 模块将文件分块，然后调用 FileUploader 模块中的 pushFragment 方法向服务器上传各文件碎片。处理文件夹中的所有文件后，调用 FileUtil 的 clearfolder 方法清空文件夹。

- 实现时遇到的问题

  - static 关键字

    由于 rust 不能像 java 一样使用 static 关键字声明类变量，因此选择先在 init 方法中，将“类变量”的值赋给可变全局变量（用static mut 声明）；再在每次使用 new 方法获得结构体时，利用全局变量给结构体中的字段赋值。

    ```
    pub fn init(tmp:&String){
       unsafe{
             //static_tmp为可变全局变量
           static_tmp = (*tmp).clone().to_string();
       }
    }

    pub fn new(f:Vec,addr:Vec,) -> FolderScanner {
    FolderScanner{
        //---snip---
    tmpFragmentFolder:unsafe{
                PathBuf::from(static_tmp.clone())
            }
        }
    }
    ```

  - File 类

    由于 rust 中的 File 结构体不能像 Java 中的 File 类一样既可指文件，又可指文件夹，因此使用 `std::path::PathBuf` 结构体代替原 Java 程序中的 File 类。并且经过比较，`std::path::PathBuf` 结构体是比 `std::path::Path` 结构体更加合适的。

  - 文件属性获取

    由于 rust 中的 `fs::metadata.permissions` 结构目前只实现了 readonly() （获取其是否为只读权限的）方法，因此无法获取文件的写属性。

  - 链表的实现

    在广度优先遍历文件夹及其子文件夹 getAllFiles 函数中，对应原 java 中的 `java.util.LinkedList` 采用 rust 中的 `std::collections::LinkedList` 结构体实现链表的相关功能。

  - 广度优先搜索获取文件夹及子文件夹下的文件

    对应于 java 中的 `java.io.File.listFiles()` 方法，在 rust 中采用 Function std::fs::read_dir 返回目录下的条目迭代器，并通过以下方式获取条目的路径，再对路径为目录还是文件进行判断。

```
if let Ok(entries) = fs::read_dir(dir){
    for entry in entries{
        if let Ok(entry) = entry{
            if let Ok(metadata) = entry.metadata(){

                let pathbuf = entry.path();
                if pathbuf.is_dir(){
                    queue.push_back(pathbuf);
                } else {
                    files.push_back(pathbuf);
                }
            }
        }
    }
}
```

4.2 server

- controlConnect
  - 功能：控制连接，与客户端进行通讯。
    - 常开一个 controlConnect ServerThread 线程，监听控制连接端口。对于接收到的每个客户机的连接和请求，创建一个新的 ClientThread 线程处理。
    - 创建的 ClientThread 新线程中，接收从 client 发回的报文并完成相应的任务。ClientThread 根据报文确定执行的操作：报文第一位为 '1' ，根据后几位的参数修改 device 的信息；报文第一位为 '2' ，根据参数查询需要的数据库 request 并发回客户端；其他情况直接返回 false 。
  - 实现时遇到的问题

    同样使用 Rust 中的 TcpStream 类型代替实现了 Java 中使用的 DataInputStream 和 DataOutputStream。
- database
  - 功能：

    DeviceItem、FileItem 和 RequestItem 定义了数据库所需结构体，并且也配备了相应的获取值和改变值的方法。

    ```
    pub fn get_port(&mut self) -> i32 {
            self.port
        }

    pub fn set_port(&mut self, port:i32) {
        self.port = port;
    }
    ```

    Query 中 new 方法连接到了 pool 连接池，从中进行查询。而后使用 MySQL 的 statement 语句对信息进行增删改查。查询信息处，查询失败返回 id 值为-1，查询成功但库中没有对应信息返回 id 值为0，查询成功并且库中有对应信息返回相应信息。若有可能查询到多个相应信息，则返回对应信息的向量。增加、删除、修改也使用对应 statement 进行操作。

    ```
    pub fn queryFile_Bypath(&self, path: Option<String>) -> Vec<FileItem>{
            let selected_files: Result<Vec<FileItem>, mysql::Error> =
            self.pool.prep_exec("SELECT * FROM DFS.file WHERE PATH =
    :path",
                    params!{"path" => path})
    ```

```
        .map(|result| {
            result.map(|x| x.unwrap()).map(|row| {
                let (id, name, path, attribute, time, noa, is_folder) =
my::from_row(row);
                FileItem {
                    id: id,
                    name: name,
                    path: path,
                    attribute: attribute,
                    time: time,
                    noa: noa,
                    is_folder: is_folder,
                }
            }).collect()
        });
        if let Err(e) = selected_files {
            let file = FileItem {
                id: -1,
                name: "".to_string(),
                path: "".to_string(),
                attribute: "".to_string(),
                time: "".to_string(),
                noa: 0,
                is_folder: false,
            };
            return vec![file];
        }
        let files = selected_files.unwrap();
        if files.len() == 0 {
            let file =  FileItem {
                id: 0,
                name: "".to_string(),
                path: "".to_string(),
                attribute: "".to_string(),
                time: "".to_string(),
                noa: 0,
                is_folder: false,
            };
            return vec![file];
        }
        files
    }
```

- 实现时遇到的问题
  - rust 不支持函数重载，改写时有些函数修改了名字，在后面加上 _Byid 等
  - 学习了很多有关数据库的知识。
  - 查询数量时，首先把符合要求的信息都储存到一个向量中，而后计算向量的长度，即为所需数量。

```
let mut i: i32 = 0;
for _f in selected_fragments.unwrap() {
    i = i+1;
}
return i;
```

- dataConnect
  - 功能：数据连接，与客户端进行文件传输。
  - 常开一个 dataConnect ServerThread 线程，监听数据连接端口。对于每个客户机的连接和请求，创建一个新的 ClientThread 线程处理。
    - ClientThread 接收从客户端发回的报文，根据报文调用相应的函数处理。可以进行发送、接收、查询、删除、记录碎片操作。报文第一位是操作命令，后几位是命令参数。

```
    status = match command[0] {
        "1" => self.recv_required_fragment(),
        "2" => self.send_fragment(),
        "3" => self.delete_fragment(),
        "4" => self.register_file(),
        "5" => self.recv_file_fragment(),
        "6" => self.check_folder(),
        _ => {
            self.client_socket.write(b"ERROR!\n");
            self.client_socket.flush();
            false
        },
    };
```

4.3 web

- com：采用的是client相应模块实现的代码
- database:采用的是server相应模块实现的代码
- userManagement
  - FileDownloader

    实现功能：相应下载的一系列操作，包括：向数据库登记文件碎片下载请求；获取当前文件下载进度；对下载得到的文件碎片解码得到原文件，并删除对应本地文件碎片。

    考虑语言特性的差异，我们将这个文件的组织形式从类与方法改为了直接定义的函数。
    - downloadRegister(path1:String, name1:String) -> String
      - 功能：下载登记。
      - 具体实现：

        查询在线的客户端数量，若无在线客户端，直接返回 "NotEnoughFragments" 。若文件碎片总数小于 1 ，直接返回 "Error" 。

        针对各个文件碎片编号（同时也是碎片文件名，计算方法：请求文件的id file_item.id * 100 + i, i in 0..noa, noa 为文件碎片数），用文件碎片编号去queryFragment，然后判断query的结果（结果即返回该碎片所在的客户端id）和 device id 是否相等，若有在线设备的 id 和此编号相等，则将对应请求加入 RequestItems 数组。

        上述循环完成后，若 RequestItem 数组长度大于 noa / 2 （我们令纠删码得到的冗余碎片和原文件碎片为 1:1 ，因此最少需要一半数量的碎片可以还原出文件），则针对 RequestItems 数组中前 noa / 2 个元素调用 query.addRequest 方法插入数据库查询请求，并返回 "OK" ；否则直接返回 "NotEnoughFragments" 。
      - 实现时遇到的问题：

原 Java 文件中变量 RequestItems 类型为一个功能封装好的链表。由于 Rust 对引用和指针的限制，用安全的 Rust 实现链表比较麻烦；而且 RequestItems 的结点类型是由四个 i32 (int) 字段组成的结构体，数据复制的代价很小，不必移动数据，因此也不必使用链表；最终改用数组实现。

```
let mut request_items: Vec<RequestItem> = Vec::new();
    for i in 0..noa {
        ...
        for j in 0..online_device.len() {
            if online_device[j].get_id() == device_id {
                request_items.push(RequestItem::init_2(1,
id*100 + i, device_id));
                break;
            }
        }
    }
    let temp = (noa / 2) as usize;
    if request_items.len() < temp {
        return String::from("NotEnoughFragments");
    }
    else {
        for i in 0..temp {
            query.addRequest(request_items[i].clone());
        }
    }
```

- progressCheck(path1:String, name1:String) -> String
  - 功能：获取当前文件下载进度。
  - 具体实现：

    如果待下载文件 id 为 -1 ，直接返回 "Error"。

    对 fragmentFolderPath ("D:webapps/DFS/CloudDriveServer/downloadFragment/") 中所有文件遍历查看，若文件名去掉后两位后与待下载文件 id 相同，则它是待下载文件的碎片，collected_file 变量加一。

    完成这个遍历后，计算文件下载进度百分比 percentage 为 2 * collected_file / file_item.noa （只需要一半的文件碎片），返回计算得到的百分比转化的字符串。
- decodeFile(path1:String, name1:String) -> String
  - 功能：从下载到的碎片文件（放在fragmentFolderPath路径下）中解码，得到待下载的原文件（放在fileFolderPath路径下）。
  - 具体实现：

    调用 Decoder::decode 函数解码，若其返回 false ，则直接返回 "Error" 。调用 decode函数成功后，遍历 fragmentFolderPath 中所有文件，查找对应文件的碎片并删除，遍历结束后返回 "OK" 。
  - 实现时遇到的问题：

    Rust 遍历当前文件系统路径的方法 read_dir() 返回值是该方法特有的 entry 遍历器类型，在类型转化和匹配上遇到困难。

- GetFileList

    - 功能：向数据库查询当前路径下的文件和目录列表，对虚拟文件系统中点击一个文件夹时的操作作响应，调用此模块来查询数据库里该文件夹下的子目录，并动态生成字符串形式的html语言返回到前端，为前端网页显示做准备。
- UserReg/UserLogin: 这两个模块为用户注册与登录模块，主要对从前端传过来的用户名和密码做操作，分别为
    - 注册：插入用户信息到数据库
    - 登录：根据用户名查询数据库，并判断密码是否与数据库中记录是否匹配
- main：利用actix-web框架的语法，开启端口 http://localhost:8080 进行监听，并匹配前端发出的每个http请求的url，以对不同的前端操作做出不同的响应。

```
#[post("/DownloadReg")]
async fn downloadreg(params: web::Json<FileDownloader_param>) ->
web::Json<Return_string> {
    println!("path: {0} ,name:{1}",params.path,params.name);
    let result:String =
FileDownloader::downloadRegister(params.path.clone(),params.name.clone(
));
    println!("{}", result);
    web::Json(Return_string {
        result,
    })
}
```

需要注意的是，前后端数据传输均采用的是Json格式，那么在Rust中，则需要web::Json来将某个rust类型数据Json格式化，且T这个类型需要有Serialize, Deserialize两个traits。针对需要在前后端传输的数据，需要自定义结构体，并由rust自动生成两个traits

```
#[derive(Serialize, Deserialize)]//添加下载请求时的数据格式
pub struct FileDownloader_param {
  path: String,
  name: String,
}
```

以下列出所有前端请求，对请求的处理所用到的API已经在之前做了解释

- 用户注册：#[post("/UserReg")]

  接受http请求的数据为用户名和用户密码；
- 用户登录：#[post("/UserLogin")]

  接受http请求的数据为用户名和用户密码；
- 文件目录刷新交互：#[post("/GetFileList")]

  接受http请求的数据为文件在虚拟文件系统中的逻辑路径
- 文件下载请求：#[post("/DownloadReg")]

  接受http请求的数据为文件在虚拟文件系统中的逻辑路径和名称

  接收到后调用函数 `FileDownloader::downloadRegister(path,name);` 来从数据库中插入下载请求。
- 文件下载进度请求：#[post("/progressCheck")]

  接受http请求的数据为文件在虚拟文件系统中的逻辑路径和名称

  接收到后调用函数 `FileDownloader::progressCheck(path,name);` 来检查web端存放碎片的文件夹中碎片数目。

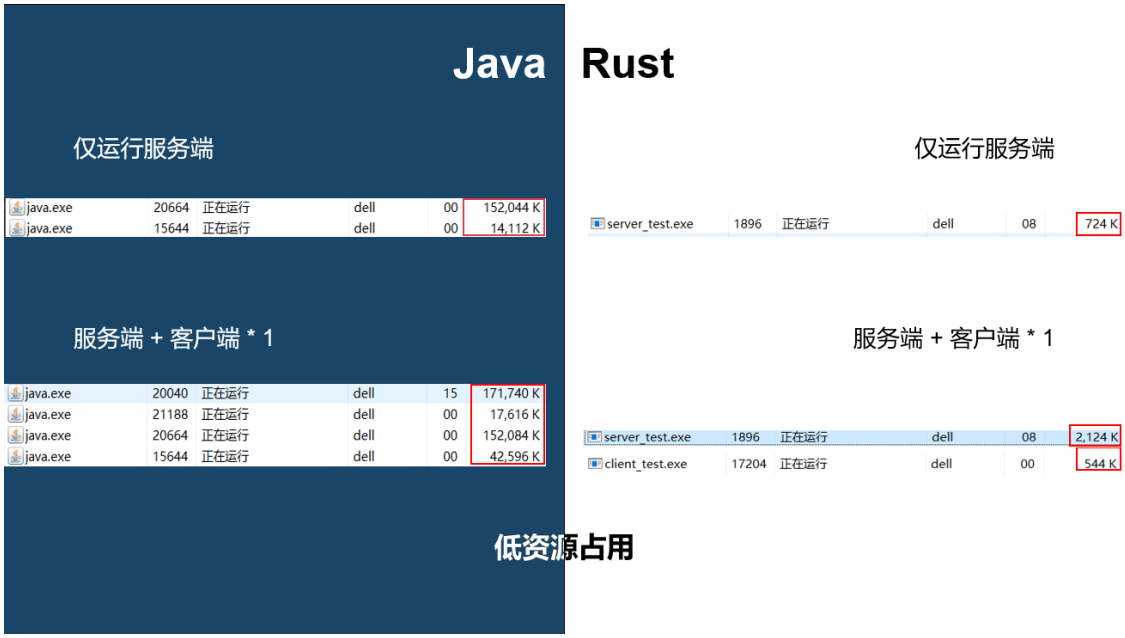
- 文件碎片解码请求:#[post("/decodeFile")]

  接受http请求的数据为文件在虚拟文件系统中的逻辑路径和名称

  接收到后调用函数`FileDownloader::decodeFile(path,name);`来利用纠删码模块对碎片进行解码，还原为文件，并删除所有碎片

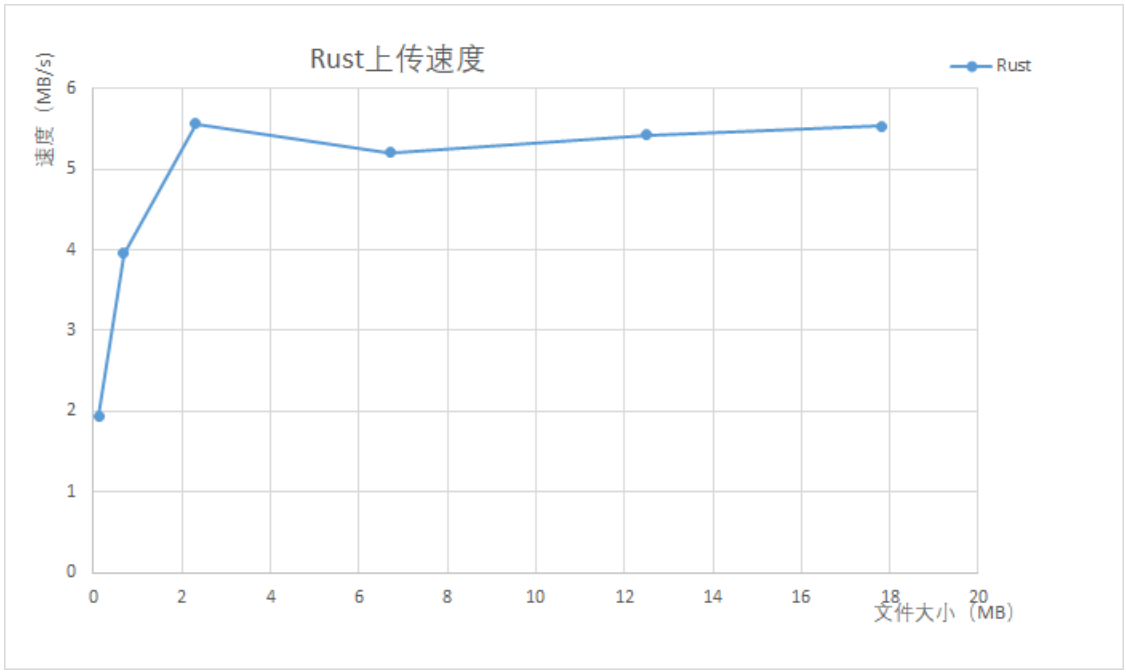
五. 系统性能分析

5.1 低资源占用

- 大多数运行条件下，一个Rust 程序比 Java 消耗的内存会少上一到两个数量级，如下图。而且原项目在结束客户端服务端之后还会有一些进程未被自动关闭，这也是一个小的缺陷。
- 

5.2 上传速度分析

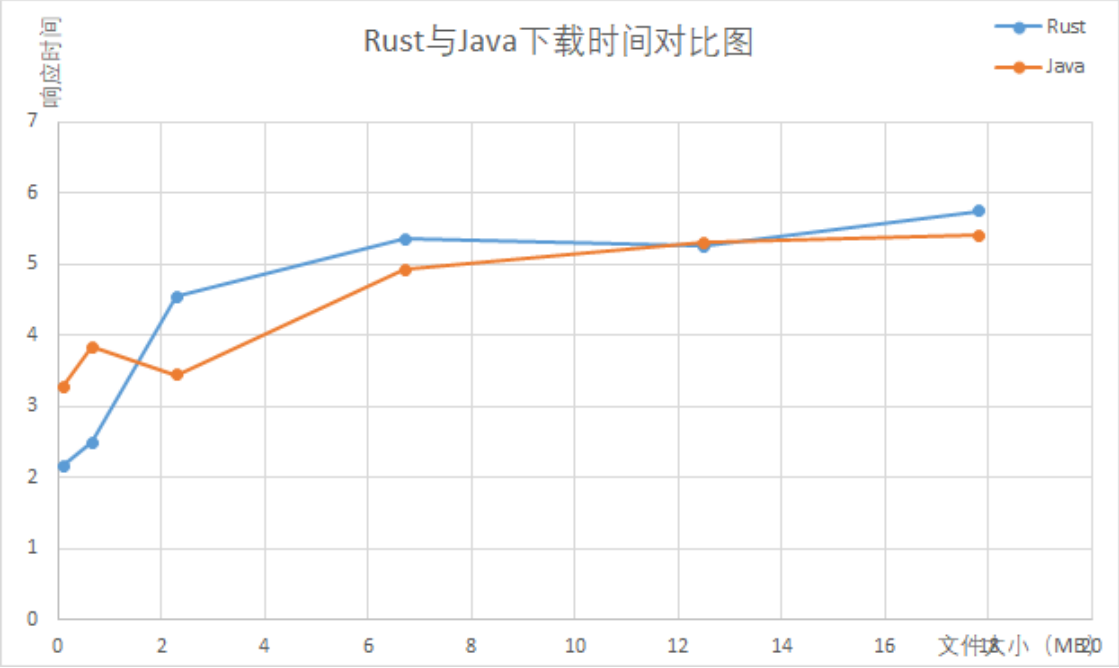
- 上传速度取决于在线客户端的数量和客户端与服务器数据传输速度。当多个客户机在线 时，客户机可以同时与服务器进行传输，从而摆脱用户网络上传速度的限制。
- 实际测试时只测试了一个客户端开启，并使用本地端口上传一系列不同大小的文件，结果仅供参考如下图



- 可以看到，在文件大小达到一定阈值后，速度整体大约在5-6MB/s

5.3 下载速度分析

- 下载速度取决于所用网络与服务器之间连接速度。并且由于通过网络下载时需要先收集碎 片，碎片上传到服务器的速度也会影响到实际下载速度。
- 在单客户端并使用本地端口进行连接的情况下，测得下载图（并与Java对比）如下



- 可以看到二者的速度大致相同
- 根据我们的预期，如果将前端采用Rust实现并打包为WebAssembly部署到浏览器上，那么本项目下载所需的时间会进一步下降

六. 整体运行方式与使用说明

部署说明文档

七. 未来展望

7.1前端设想

目前我们使用的是原项目的前端 Java 代码，它上传下载的一些功能还不完善。由于时间关系，我们计划在大作业结束后使用 Rust 重新编写前端，完善功能、并在网页端应用 WebAssembly 。

- Seed 简介

  Seed 采用 Elm 结构，它的核心包括以下三部分：

  - Model：应用的当前状态
  - View：把状态转化为网页端可视的 HTML
  - Update：根据网页端返回的 Msg 更新状态

  也因此 Seed 不需要额外的网页状态维护。另外，Seed 采用稳定版本的 Rust，不含有 unsafe 模块，保障应用的安全性。

- 前端模型设想

  参考 seed-realworld 示例和原项目网页，将项目分为注册、登录、用户使用页面三个主要页面，分别设计各自的 Model 和 Msg 模型。

  初步设想的网页模型：

    - 注册和登录页面

      Model 结构体：用户当前状态，注册和登录需要输入的信息栏（用户名、密码、邮箱等），网页错误

      Msg 枚举体：页面跳转、注册或登录提交、注册或登录完成、网页错误

    - 用户使用页面

      Model 结构体：用户当前状态，文件显示状态，网页错误，下载目标路径填写栏

      Msg 枚举体：页面跳转、下载请求、下载完毕、上传请求、上传完毕、文件目录刷新请求、文件目录刷新完毕、改写目标路径提交、改写目标路径完毕

- 可能的挑战

  Seed 总体使用方面：

    - Seed 仍处于快速发展中，一些内容可能不稳定；比起更成熟的网页框架，可复用的代码少。
    - 可参考的文档较少，遇到的一些错误可能没有参考的解决方案

  本项目方面：

    - 动态刷新文件目录的处理
    - 上传下载 button 与后端的交互

7.2 不足

- 前端的功能还不完整，目前所有用户都只能共享同一个虚拟文件系统，可以尝试在前端实现用户文件上传的功能，并为每个用户定制其文件系统（与我们家庭式分布式文件系统有一点偏了？）。
- 架构方面，目前我们的实现是服务器作为 web端与存储端(client)的转发节点，一方面转发文件碎片，另一方面将碎片解码为原文件。当用户数目过多，会使得服务器的机器性能成为分布式文件系统运行效率的瓶颈；那么如果考虑直接实现存储端和web端的碎片传输，并利用WebAssembly对密集型计算工作的性能优势，让文件编解码在web端完成，这样或许可以显著降低服务器端的性能需求，同时因为转发次数减少以及WebAssembly性能而使得文件系统的下载等功能速度更快，用户体验更好。
- 数据库存储结构不够合理，表项之间的关联没有很好地设置。(?这个关联？)
- 能够存储的单个文件大小有限制，目前不能存储 50MB 以上的文件。 (这个的解决方式？限制原因？)
- 运行时的配置比较复杂，但这具体体现在原项目前端的配置上，而这也是我们致力于未来把前端Rust化的动力之一：通过Rust的包管理器cargo能很轻松的部署整个项目。

7.3 未来目标规划

- 大创中的目标：

  完善项目的功能和架构。

    - 实现 Seed 框架的前端，完善前端功能，对浏览器页面加载交互，并利用 Rust 的 web-pack 等包将代码编译为 WebAssembly ，运行在浏览器中。
    - 改进架构，实现在 web 端应用 WebAssembly 计算纠删码编解码。
    - 改进文件碎片取名方式，改进代码以支持存储和上传下载更大的文件。
- 更进一步的目标：

    - 实现完整实用的基于 Rust 和 WebAssembly 的分布式文件系统，并将项目部署到公网上、投入使用。
    - 为 Rust 和 WebAssembly 开源社区做贡献。

八. 参考文献

[Rust docs]

Rust锁机制

Yew 中文文档

Ruukh 仓库

Rust工具简介及环境优化

纠删码（Erasure Code）浅析

HTML 简介

Rust-wasm book

Multithreading Rust and wasm

seed

Rust Web框架列表

Actix Document

actix-web

rocket

actix web框架小试

Instruction to Rust Web Applications

Using Web Workders