可行性报告

项目简介

该项目通过高效、安全的 Rust 语言对17级项目“基于互联网网页的小型分布式文件系统”进行改写，并用极具计算效率和兼容性的 WebAssembly 来与 JavaScript 交互实现更高效的网页前端逻辑，并用 WebAssembly 把 Rust 写的分布式文件系统程序的包装，再通过 Node.js 部署 Web 服务器, 最终在 Node.js (V8 引擎)实现跨平台的，可提供移动式文件访问的分布式文件系统，并在性能、兼容性、创新性上取得突破。

Rust-Java

1 实现方案与可行性分析

1.1 客户端

1.1.1 客户端功能

原 17 级项目 “基于互联网网页的小型分布式文件系统“ 主要实现了如下功能：

- 客户端启动模块：启动后读取配置文件，根据配置文件的设置启动网络链接模块中的控制链接线程与文件夹监控线程，最后调用 wait 函数进入等待状态。
- 客户端文件分块模块：运用 erasure code 算法进行分块
- 客户端文件夹监控模块：该模块将根据配置文件的设置监控一系列文件夹并在发现新文件时发起数据连接将其上传到分布式文件系统中配置文件指定的逻辑位置并在本地删除之。
- 客户端网络链接模块：数据链接类 FragmentManager 、控制连接类 ServerConnecter 、文件传送类 FileTransporter

1.1.2 代码结构

- client
  - client
    - Client
    - SynItem
  - com
    - backblaze
      - erasure //原项目采用 backclaze 公司提供的开源实现，本项目采用 github 上的开源项目 reed-solomon-erasure 实现
  - connect
    - FileTransporter //其中有两个静态函数分别用于文件碎片的发送与接收
    - FragmentManager //数据链接类，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
    - ServerConnecter
  - fileDetector
    - FileAttrs //查询属性
    - FileUploader
    - FileUtil //寻找文件
    - FolderScanner //每隔一段时间确认一次是否有文件夹中被放入了新文件

1.1.3 Rust 实现

本项目使用 Rust 改写17级“基于互联网网页的小型分布式文件系统”项目，用 Rust 改写原项目的 Java 客户端。与原项目相同，本项目的客户端也采用与 tcp 协议的 socket 通信与服务器建立连接。

客户端利用 Config 读取配置文件，为 Rust 应用程序组织分层或分层配置。Config 可设置一组默认参数，然后可以通过合并各种来源的配置来扩展它们。

客户端提供以下几个功能：

- 获取本地目录

  原项目的 Java 客户端使用 `java.io.File` 类获取文件列表，本项目选用 Rust 改写，使用 `std::io::fs` 模块处理文件系统，列出本地机器参与共享的文件列表。

```
//std::io::fs 模块包含几个处理文件系统的函数
use std::fs;
use std::fs::{File, OpenOptions};
use std::io;
use std::io::prelude::*;
use std::os::unix;
use std::path::Path;
```

```
match fs::read_dir("a") {
    //在主函数中读取目录的内容，返回 `io::Result<Vec<Path>>`
    Err(why) => println!("! {:?}", why.kind()),
    Ok(paths) => for path in paths {
        println!("> {:?}", path.unwrap().path());
    },
}
```

- 对文件进行分块

  客户端使用 erasure code 算法，对服务器发来的文件进行分块。此算法可依据 github 上的开源项目 reed-solomon-erasure 实现。

- 向服务器发送文件碎片、接收服务器传来的碎片
  - 使用到的标准库：

    `std::net:TcpListener` 用于监听连接

    `std::net::TcpStream` 用于传输数据。

  - 过程：在 `TcpListener` `accept` 连接或 `connect` 到一个远程主机后，将在本地和远程套接字间创建一个 `TcpStream` ，数据（文件碎片）通过写入或读取它以进行传输。
  - 示例：

```
use std::io::prelude::*;
use std::net::TcpStream;

fn main() -> std::io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:34254")?;

    stream.write(&[1])?;
    stream.read(&mut [0; 128])?;
    Ok(())
} // the stream is closed here
```

- 响应服务器删除本地文件

  接收到服务器删除文件的请求后，依据文件路径，使用 `std::fs::remove_file` 删除文件，示例代码如下

```
use std::fs;
fn main() {
   fs::remove_file("data.txt").expect("could not remove file");
   println!("file is removed");
}
```

1.2 服务器

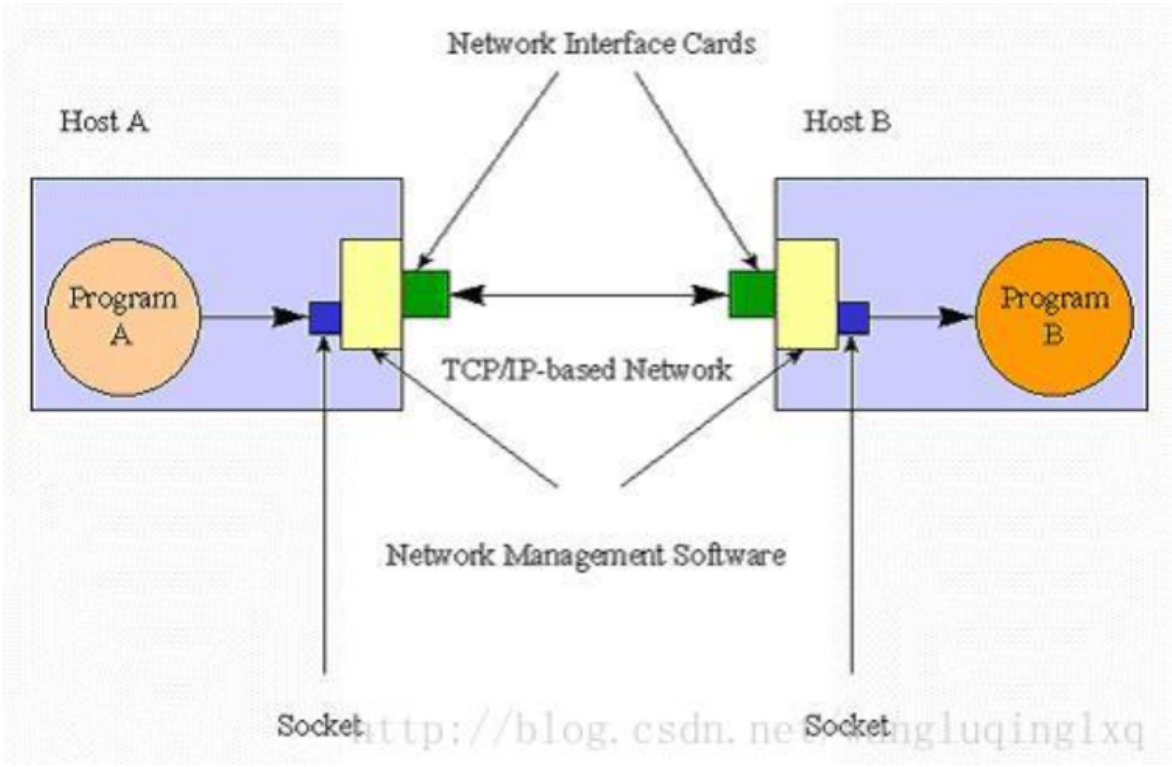
1.2.1 服务器功能

基于互联网网页的小型分布式文件系统主要实现了如下功能：

- 连接类：接收、回复、转发服务请求与控制信息；收发数据（文件碎片）。
- 数据管理类：维护云共享文件索引；维护各个客户端的状态信息；记录、处理当前等待响应的文件请求。

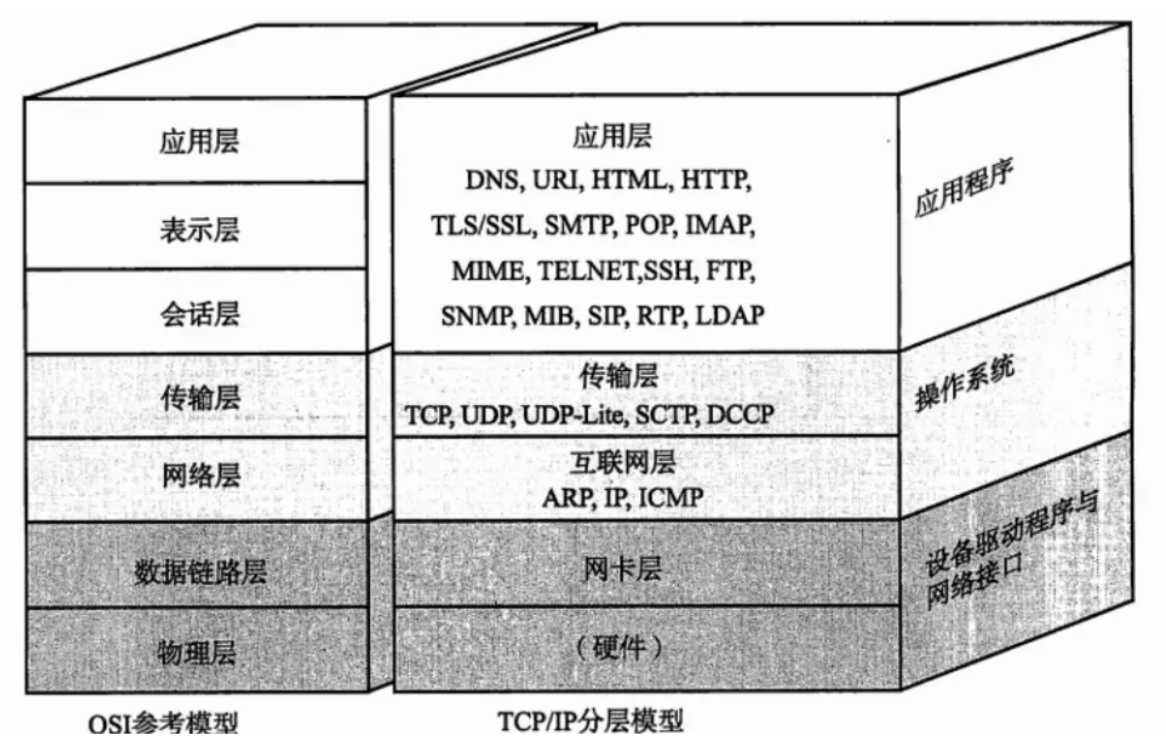
1.2.2 Java 源码实现方法

首先要介绍套接字（Socket）的含义：套接字是一种在应用程序与 TCP / IP 协议交互时，用来区分不同应用程序进程间的网络通信和连接的接口。主要有三个参数：通信的目的 IP 地址、使用的传输层协议(TCP 或 UDP)和使用的端口号。Socket 可以看成在两个程序进行通讯连接中的一个端点，一个程序将一段信息写入 Socket 中，该 Socket 将这段信息发送给另外一个 Socket 中，使这段信息能传送到其他程序中。如下图：



)

Host A 上的程序A将一段信息写入Socket 中，Socket 的内容被 Host A 的网络管理软件访问，并将这段信息通过 Host A 的网络接口卡发送到 Host B ， Host B 的网络接口卡接收到这段信息后，传送给 Host B 的网络管理软件，网络管理软件将这段信息保存在 Host B 的 Socket 中，然后程序B 才能在 Socket 中阅读这段信息。要通过互联网进行通信，至少需要一对套接字，一个运行于客户机端，称之为 ClientSocket ，另一个运行于服务器端，称之为 serverSocket 。



- ServerSocket 类在服务器端创建欢迎套接字。
- Socket 类在客户端或服务器端创建链接套接字。
- 用一系列在 MySQL 数据库中的 table 保存云共享文件索引，每台共享了文件的电脑均在数据库中对应两个 table，其一记录了这台电脑上共享的每个文件的唯一标识符和其逻辑位置；其二记录了这台电脑上各个文件的碎片的物理位置和其唯一标识符。
- 用一个在 MySQL 数据库中的 table 保存客户端，其中有：唯一标识符，在线情况，剩余空间及当前复杂维持与这个客户端的控制链接（TCP 链接）的线程的编号。
- 再用一个 table 记录当前网页提出的文件请求。当服务器收到来自客户端的心跳连接时，将查询文件请求表，如果发现有对客户端上文件的请求，则在回复心跳连接时将文件请求发给客户端并令其（通过服务器）将文件发往请求方，这样就能解决服务器不时时连接客户端的问题。

1.2.3 源码结构

- server
  - controlConnect
    - ClientThread
    - ServerThread
  - database
    - DeviceItem //各种信息参数查询设置
    - FileItem //各种信息参数查询设置
    - Query closeConnection queryFile //通过名字地址或ID或地址查找文件
      queryFragment//fragment, device, request, password, id
      或者对上述对象 add, delete, alter
    - RequestItem
  - dataConnect
    - ClientThread //send delete receive fragment confirm //确定在线主机，碎片数量，判断如何发送
    - FileTransporter //receive send files
    - ServerThread
  - DFS_server

1.2.4 Rust 实现

- `std :: net`

  TCP / UDP 通信的网络原语。该模块提供了传输控制和用户数据报协议的网络功能，以及 IP 和套接字地址的类型。

  > TcpListener 并 TcpStream 提供用于通过 TCP 进行通信的功能
  > UdpSocket 提供通过 UDP 进行通信的功能
  > IpAddr 表示 IPv4 或 IPv6 的 IP 地址；Ipv4Addr 和 Ipv6Addr 分别是 IPv4 和 IPv6 地址
  > SocketAddr 表示 IPv4 或 IPv6 的套接字地址；SocketAddrV4 和 SocketAddrV6 分别是 IPv4 和 IPv6 套接字地址
  > ToSocketAddrs 与网络对象，如交互时使用的通用地址解析服务的特质 TcpListener，TcpStream 或 UdpSocket
  > 其他类型是此模块中各种方法的返回值或参数类型

- `std::thread`

  正在执行的 Rust 程序由一组本机 OS 线程组成，每个本机线程都有自己的堆栈和本地状态。可以命名线程，并为低级同步提供一些内置支持。

  线程之间的通信可以通过通道，Rust 的消息传递类型以及其他形式的线程同步和共享内存数据结构来完成。

  当 Rust 程序的主线程终止时，即使其他线程仍在运行，整个程序也会关闭。但是，此模块提供了方便的功能，可以自动等待子线程的终止。

  可以使用thread::spawn函数产生一个新线程：

  ```
  use std::thread;

  thread::spawn(move || {
      // some work here
  });
  ```

  在此示例中，生成的线程与当前线程“分离”。这意味着它可以超过其父级（产生它的线程），除非该父级是主线程。

  父线程也可以等待子线程的完成。调用 `spawn` 产生 `JoinHandle` ，提供了 `join` 等待的方法：(该 `join` 方法返回一个 `thread::Result` 包含 `Ok` 子线程产生的最终值的内容，或者返回给子线程恐慌时 `Err` 调用的值 `panic!`)

  ```
  use std::thread;

  let child = thread::spawn(move || {
      // some work here
  });
  // some work here
  let res = child.join();
  ```

  该模块还为 `Rust` 程序提供了线程本地存储的实现。线程本地存储是一种将数据存储到全局变量中的方法，程序中的每个线程都有其自己的副本。线程不共享此数据，因此不需要同步访问。

  线程能够具有关联的名称以用于识别。默认情况下，生成的线程是未命名的。要为线程指定名称，请使用构建线程， `Builder` 并将所需的线程名称传递给 `Builder::name` 。要从线程内部检索线程名称，请使用 `Thread::name.`

- `rustc::traits::query`

  特征查询界面的实验类型。该模块中定义的方法全部基于规范化，该规范化通过替换未绑定的推理变量和区域进行规范查询，从而可以更广泛地重用结果。可以在中找到此处定义的查询的提供程序librustc_traits

- `crate mysql`

  提供了：

  - 完全用 Rust 写的 MySql 数据库驱动程序
  - 连接池

  特征：

  - macOS，Windows 和 Linux 支持
  - MySql 文本协议支持，即简单文本查询和文本结果集的支持；
  - MySql 二进制协议支持，即支持预备语句和二进制结果集；
  - 支持大于2 ^ 24的 MySql 数据包；
  - 支持 Unix 套接字和 Windows 命名管道；

  安装

```
[dependencies]
mysql = "*"
```

2 理论依据与技术依据

2.1 Rust 改写

- 纠删码（Erasure Code）：纠删码是一种前向错误纠正技术，用于在网络传输中避免包的丢失，以提高存储可靠性。它可将 n 份原始数据，增加 m 份数据，并能通过 n+m 份中的任意 n 份数据还原为原始数据。即如果有任意小于等于 m 份数据失效，仍然能通过剩下的数据还原出来。此为原17级项目“基于互联网的小型分布式文件系统”的一大特点。本项目将继承该项优势，用 Rust 实现纠删码。
- 网络数据传输：本项目所做的移动式文件访问的分布式文件系统，旨在使得分布在不同地点的各种设备可以共同维护，因此，采用使用TCP/IP协议的因特网进行数据交换。Rust 中有 `std::net:TcpListener` `std::net::TcpStream` 等标准库可以调用。

2.2 Rust 调用 Java

使用 j4rs 项目在 Rust 中调用 Java。其主要思想是实现一个 crate，让用户轻松调用 Java，这样他们就可以从庞大的 Java 生态系统中受益。

- 注意 JNI（Java Native Interface） 所以需要的配置（例如 jvm 包含/链接本地共享库）。
- 创建一个直观、简单的 API 进行 Java 调用（Rust -> Java 方向）。
- 允许 Java -> Rust 回调。
- 无缝在 Linux 或者 Windows 上使用 crate（当然，前提是安装了 Java）。
- 遵循 “Rust-first” 方式： Rust 代码可以创建和管理 JVM，而不是反过来

开始时候仅需要在 Cargo.toml 中定义 `j4rs` ：

```
[dependencies]
j4rs = "0.6"
```

使用 `j4rs`，下载 Maven 构件，以便在 Rust 应用程序中使用它调用其他 Java 库。

Rust-WebAssembly-JS

1 实现方案与可行性分析

1.1 项目Demo分析

- 通过研究和尝试将 Rust 实例转化为 WebAssembly ，调用相应的库并将其放在 Node.js（基于 Chrome V8 引擎的 JavaScript 运行环境）上运行，来说明此套流程的可行性
- JavaScript 的垃圾收集堆与 WebAssembly 的 Rust 值所在的线性内存空间不同。WebAssembly 当前无法直接访问垃圾收集的堆
- JavaScript 可以读取和写入到 WebAssembly 线性内存空间
- 一般而言，良好的 JavaScript ↔ WebAssembly 接口设计通常是将大型，长期存在的数据结构实现为 Rust 类型，并驻留在 WebAssembly 线性内存中，并作为不透明的句柄暴露给 JavaScript 。JavaScript 调用导出的 WebAssembly 函数，这些函数采用这些不透明的句柄，转换其数据，执行大量计算，查询数据并最终返回小的可复制结果。通过仅返回较小的计算结果，避免了在 JavaScript 垃圾回收堆和 WebAssembly 线性内存之间来回复制或串行化所有内容。

1.2 项目Demo构建

- 从 github 上克隆项目模板

  > cargo generate --git https://github.com/rustwasm/wasm-pack-template

  - 关键文件解析
    - **Cargo.toml**：指定了 cargo Rust 的包管理器和构建工具的依赖项和元数据
    - **/scr/lib.rs**：正在编译为 WebAssembly 的 Rust crate 的根文件，用于 wasm-bindgen 与 JavaScript 进行交互。
    - **/scr/utils.rs**：提供了通用程序，使 Rust 编译为 WebAssembly 的过程更加轻松。
- 在项目目录中运行

  > wasm-pack build

- 构建完成后，可在 pkg 目录中找到相应的文件

```
pkg/
├── package.json
├── README.md
├── wasm_game_of_life_bg.wasm
├── wasm_game_of_life.d.ts
├── wasm_game_of_life.js
└──...
```

  - **wasm_game_of_life_bg.wasm**：WebAssembly 的二进制文件，由 Rust 编译器从 Rust 的源代码生成，包含所有的 Rust 函数和数据的 wasm 版本。
  - **wasm_game_of_life.js**：由 wasm-gindgen JavaScript 胶水生成并包含 JavaScript 胶水，用于将 DOM 和 JavaScript 函数导入 Rust，并将 WebAssembly 函数 API 公开给

JavaScript。
    - **wasm_game_of_life.d.ts**：包含 JavaScript 胶水的 TypeScript 类型声明
    - **package.json**：包含和生成的 JavaScript 和 WebAssembly 包有关的元数据
- 放入网页
    - 在 wasm-game-of-life 中运行命令

      > npm init wasm-app example

    - 打开 example 子目录可以看到如下文件

      ```
      wasm-game-of-life/example/
      ├── bootstrap.js
      ├── index.html
      ├── index.js
      ├── LICENSE-APACHE
      ├── LICENSE-MIT
      ├── package.json
      ├── README.md
      ├── webpack.config.js
      └── ...
      ```

    - 分析
        - **package.json**：自带预先配置有 webpack 和 webpack-dev-server 依赖,以及初始的 hello-wasm-pack
        - **webpack.config.js**：配置 webpack 及其本地开发服务器
        - **index.html**：网页的根 HTML 文件，负载了 bootstrap.js

          ```
          <!DOCTYPE html>
          <html>
            <head>
              <meta charset="utf-8">
              <title>Hello wasm-pack!</title>
            </head>
            <body>
              <script src="./bootstrap.js"></script>
            </body>
          </html>
          ```

        - **index.js**：Web 页面 JavaScript 的主要入口点，用于导入 hello-wasm-pack 的 npm 包，其中包含默认 wasm-pack-template 的已编译 WebAssembly 和 JavaScript 胶水，然后调用 hello-wasm-pack 的 greet 函数

          ```
          import * as wasm from "hello-wasm-pack";

          wasm.greet();
          ```

- 安装依赖项
    - 在 example 子目录中运行如下指令，用于安装 webpack JavaScript 捆绑器及其开发服务器

      > npm install

- 使用本地的 example 包
    - 在 /example/package.json 中的 dependencies 中加入

> "wasm-game-of-life": "file:../pkg"

- 修改 /example/index.js

```
import * as wasm from "wasm-game-of-life";

wasm.greet();
```

- 重新执行

> npm install

- 开启一个新的终端，并在 example 目录中运行

> npm run start

并将浏览器导航到 http://localhost:8080/ ，即可看到写有 “Hello,wasm-game-of-life” 的提示框

- Canvas API 的使用，直接从内存渲染到 Canvas（不再使用 Unicode 文本）
  - 在 index.html 内部进行如下替换

```
<body>
  <canvas id="game-of-life-canvas"></canvas>
  <script src='./bootstrap.js'></script>
</body>
```

  - 对 index.js 进行如下改写

```
import {
    Universe,
  Cell
} from "wasm-game-of-life";

const CELL_SIZE = 5; // px
const GRID_COLOR = "#CCCCCC";
const DEAD_COLOR = "#FFFFFF";
const ALIVE_COLOR = "#000000";

// Construct the universe, and get its width and height.
const universe = Universe.new();
const width = universe.width();
const height = universe.height();

// Give the canvas room for all of our cells and a 1px border
// around each of them.
const canvas = document.getElementById("game-of-life-canvas");
canvas.height = (CELL_SIZE + 1) * height + 1;
canvas.width = (CELL_SIZE + 1) * width + 1;

const ctx = canvas.getContext('2d');

const renderLoop = () => {
    universe.tick();

    drawGrid();
    drawCells();
```

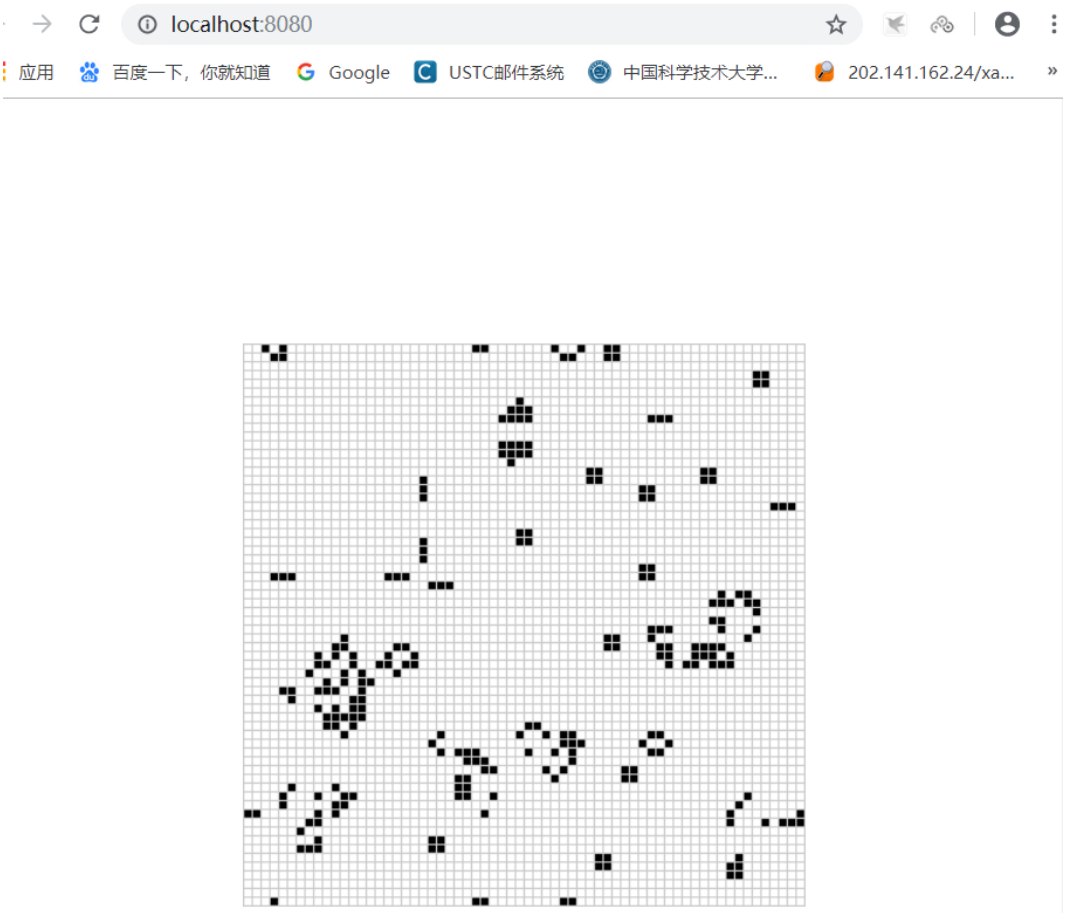
```
    requestAnimationFrame(renderLoop);
};
// Construct the universe, and get its width and height.

const drawGrid = () => {
    ctx.beginPath();
    ctx.strokeStyle = GRID_COLOR;

    // Vertical lines.
    for (let i = 0; i <= width; i++) {
        ctx.moveTo(i * (CELL_SIZE + 1) + 1, 0);
        ctx.lineTo(i * (CELL_SIZE + 1) + 1, (CELL_SIZE + 1) * height +
1);
    }

    // Horizontal lines.
    for (let j = 0; j <= height; j++) {
        ctx.moveTo(0, j * (CELL_SIZE + 1) + 1);
        ctx.lineTo((CELL_SIZE + 1) * width + 1, j * (CELL_SIZE + 1) +
1);
    }

    ctx.stroke();
};



// Import the WebAssembly memory at the top of the file.
import {
    memory
} from "wasm-game-of-life/wasm_game_of_life_bg";

// ...

const getIndex = (row, column) => {
    return row * width + column;
};

const drawCells = () => {
    const cellsPtr = universe.cells();
    const cells = new Uint8Array(memory.buffer, cellsPtr, width *
height);

    ctx.beginPath();

    for (let row = 0; row < height; row++) {
        for (let col = 0; col < width; col++) {
            const idx = getIndex(row, col);

            ctx.fillStyle = cells[idx] === Cell.Dead ?
                DEAD_COLOR :
                ALIVE_COLOR;

            ctx.fillRect(
                col * (CELL_SIZE + 1) + 1,
                row * (CELL_SIZE + 1) + 1,
                CELL_SIZE,
                CELL_SIZE
```

```
            );
        }
    }

    ctx.stroke();
};

drawGrid();
drawCells();
requestAnimationFrame(renderLoop);
```

- 重新在 wasm-game-of-life 中依次执行如下命令，并将浏览器导航到 http://localhost:8080/

  > wasm-pack build
  >
  > npm run start

  结果如下



- Rust 代码如下

```
mod utils;

use wasm_bindgen::prelude::*;
```

```
// When the `wee_alloc` feature is enabled, use `wee_alloc` as the
global
// allocator.
#[cfg(feature = "wee_alloc")]
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc = wee_alloc::WeeAlloc::INIT;

#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
#[repr(u8)]
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Cell {
    Dead = 0,
    Alive = 1,
}

#[wasm_bindgen]
pub struct Universe {
    width: u32,
    height: u32,
    cells: Vec<Cell>,
}

impl Universe {
    fn get_index(&self, row: u32, column: u32) -> usize {
        (row * self.width + column) as usize
    }

    // ...
}

impl Universe {
    // ...

    fn live_neighbor_count(&self, row: u32, column: u32) -> u8 {
        let mut count = 0;
        for delta_row in [self.height - 1, 0, 1].iter().cloned() {
            for delta_col in [self.width - 1, 0, 1].iter().cloned() {
                if delta_row == 0 && delta_col == 0 {
                    continue;
                }

                let neighbor_row = (row + delta_row) % self.height;
                let neighbor_col = (column + delta_col) % self.width;
                let idx = self.get_index(neighbor_row, neighbor_col);
                count += self.cells[idx] as u8;
            }
        }
        count
    }
}

/// Public methods, exported to JavaScript.
#[wasm_bindgen]
```

```
impl Universe {
    pub fn tick(&mut self) {
        let mut next = self.cells.clone();

        for row in 0..self.height {
            for col in 0..self.width {
                let idx = self.get_index(row, col);
                let cell = self.cells[idx];
                let live_neighbors = self.live_neighbor_count(row,
col);

                let next_cell = match (cell, live_neighbors) {
                    // Rule 1: Any live cell with fewer than two live
neighbours
                    // dies, as if caused by underpopulation.
                    (Cell::Alive, x) if x < 2 => Cell::Dead,
                    // Rule 2: Any live cell with two or three live
neighbours
                    // lives on to the next generation.
                    (Cell::Alive, 2) | (Cell::Alive, 3) => Cell::Alive,
                    // Rule 3: Any live cell with more than three live
                    // neighbours dies, as if by overpopulation.
                    (Cell::Alive, x) if x > 3 => Cell::Dead,
                    // Rule 4: Any dead cell with exactly three live
neighbours
                    // becomes a live cell, as if by reproduction.
                    (Cell::Dead, 3) => Cell::Alive,
                    // All other cells remain in the same state.
                    (otherwise, _) => otherwise,
                };

                next[idx] = next_cell;
            }
        }

        self.cells = next;
    }

    // ...
}

use std::fmt;

impl fmt::Display for Universe {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        for line in self.cells.as_slice().chunks(self.width as usize) {
            for &cell in line {
                let symbol = if cell == Cell::Dead { '□' } else { '■'
};
                write!(f, "{}", symbol)?;
            }
            write!(f, "\n")?;
        }

        Ok(())
    }
}
```

```rust
/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    // ...

    pub fn new() -> Universe {
        let width = 64;
        let height = 64;

        let cells = (0..width * height)
            .map(|i| {
                if i % 2 == 0 || i % 7 == 0 {
                    Cell::Alive
                } else {
                    Cell::Dead
                }
            })
            .collect();

        Universe {
            width,
            height,
            cells,
        }
    }

    pub fn render(&self) -> String {
        self.to_string()
    }
}

/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    // ...

    pub fn width(&self) -> u32 {
        self.width
    }

    pub fn height(&self) -> u32 {
        self.height
    }

    pub fn cells(&self) -> *const Cell {
        self.cells.as_ptr()
    }
}
```

- 可见，通过调用 wasm-bindgen 库来实现 JavaScript 和 Rust 之间的通信是可行的，并且 Rust 能很好地被打包为 wasm 文件，发布到 npm 上，并运行在 Node.js 上，这也为后续开发奠定了基础。

2 理论依据与技术依据

2.1 工具链

- **Rust工具链**
  - rust up：安装 Rust 、切换 Rust 版本、下载标准库文件等
  - rustc：Rust 编译器
  - cargo：项目管理工具
- **wasm-pack**
  - 用于构建、测试和发布由 Rust 生成的 WebAssembly，并与 Java、Web 和 Node.js 进行交互操作。
- **wasm-opt**
  - 读取 WebAssembly 作为输入，对其进行转换，优化和检测，并输出转换过后的 WebAssembly。
- **wasm2js**
  - 将 WebAssembly 编译为 "almost asm.js"，用于支持没有 WebAssembly 实现的浏览器。
- **wasm-gc**
  - 对 WebAssembly 模块进行垃圾回收，删除所有不需要的导出，导入，函数等。
- **wasm-snap**
  - 用 unreachable 指令替换 WebAssembly 函数的主体。
- **twiggy**、**wasm-objdump**、**wasm-nm**
  - 用于检查 .wasm 二进制文件。
- **cargo-generate**
  - 利用预先存在的 git 存储库作为模板，可以快速运行新的 Rust 项目。

    > cargo install cargo-generate

- **npm**
  - 全称 Node Package Manager，即 “node包管理器”，是 Node.js 默认的、以 JavaScript 编写的软件包管理系统。
  - npm 会随着 Node.js 自动安装。npm 模块仓库提供了一个名为 “registry” 的查询服务，用户可通过本地的 npm 命令下载并安装指定模块。此外用户也可以通过 npm 把自己设计的模块分发到 registry 上面。registry 上面的模块通常采用 CommonJS 格式，而且都包含一个 JSON 格式的元文件。
  - npm 可以管理本地项目的所需模块，并自动维护依赖情况，也可以管理全局安装的 JavaScript 工具。
  - 如果一个项目中存在 package.json 文件，那么用户可以直接使用 `npm install` 命令自动安装和维护当前项目所需的所有模块。在 package.json 文件中，开发者可以指定每个依赖项的版本范围，这样既可以保证模块自动更新，又不会因为所需模块功能大幅变化导致项目出现问题。开发者也可以选择将模块固定在某个版本之上。
- **Parcel**
  - 快速打包：Parcel 使用工作进程启用多核编译，并具有文件系统缓存，即使在重新启动后也可快速重新构建。
  - 打包所有资源：支持 JS，CSS，HTML，文件资源等等 ，不需要安装任何插件。
  - 自动转换：在需要时，代码使用 Babel，PostCSS 和 PostHTML 自动转换 ，即使是在 `node_modules` 文件里。
  - 零配置代码拆分：使用动态 `import()` 语法拆分输出包，只加载初始加载时所需的内容。
  - 模块热替换：在开发过程中进行更改时，Parcel 会自动更新浏览器中的模块，不需要进行任何配置。
  - 友好的错误记录：遇到错误时，Parcel 会以语法高亮的形式打印代码帧，用于查明问题。
- **Crates**
  - 与 JavaScript 和 DOM 进行交互

        - **wasm-bindgen**：定义了如何跨边界使用复合结构的共识。它涉及将 Rust 结构装箱，将指针包装在JavaScript 类中以提高可用性，或将其索引到 Rust 中的 JavaScript 对象表中。
            - 用于促进 Wasm 模块和 JavaScript 之间的高层交互，如导入 JavaScript 结构、函数和对象以在 WebAssembly 中调用。
            - 促进 Rust 和 JavaScript 之间的高级交互。允许 JS 使用字符串调用 Rust API，或 Rust 函数捕获 JS 异常。
            - 抹平了 WebAssembly 和 JavaScript 之间的阻抗失配，确保 JavaScript 可以高效地调用 WebAssembly 函数，并且无需 boilerplate，同时 WebAssembly 可以对 JavaScript 函数执行相同的操作。
        - **wasm-bindgen-futures**：连接 JavaSript Promise 和 Rust Future 的桥梁。它可以双向转换，在 Rust 中使用异步任务时非常有用，并且可以与 DOM 事件和 I/O 操作进行交互。
        - **js-sys**：用于所有的 JavaScript 全局类型和方法的 Raw wasm-bindgen，如 Object，Function，eval等。
        - **web-sys**：wasm-bindgen 中所有 Web API 的原始导入，如 DOM 操作 setTimeout，Web GL ， Web Audio。
    - 错误报告和记录
        - **console_error_panic_hook**
        - **console_log**
    - 动态分配
        - **wee_alloc**
    - 解析和生成 .wasm 二进制文件
        - **parity-wasm**：用于序列化，反序列化和构建 .wasm 二进制文件的低级 WebAssembly 格式库。
        - **wasmparser**：一个简单的事件驱动型库，用于解析 WebAssembly 二进制文件。
    - 解释和编译WebAssembly
        - **wasmi**：来自 Parity 的可嵌入 WebAssembly 解释器
        - **cranelift-wasm**：将 WebAssembly 编译为本机主机的机器代码
- 模板
    - **wasm-pack-template**：用于搭配 wasm-pack 启动 Rust 和 WebAssembly 项目。
    - **create-wasm-cpp**：用于 JavaScript 项目，从 npm 中获取代码依赖的包。
    - **rust-webpack-template**：预先配置了所有样板，用于将 Rust 编译为 WebAssembly 并将其直接挂钩到 Webpack 的 Webpack 构建管道中的 rust-loader。

2.2 JS与Node.js使用分析

2.2.1 JavaScript

- JSON：超轻量级数据交换格式

  JS内置Json的解析：
    - `JSON.stringify()`： 将JS值序列化为JSON格式字符串
    - `JSON.parse()`： 把JSON格式字符串变为JS对象
- jQuery
    - 优势：
        - 消除浏览器差异：你不需要自己写冗长的代码来针对不同的浏览器来绑定事件，编写AJAX等代码；

        - 简洁的操作 DOM 的方法：写 `$('#test')` 肯定比 `document.getElementById('test')` 来得简洁；
        - 轻松实现动画、修改 CSS 等各种操作。
    - 选择器：筛选出 HTML 中 DOM 节点元素
    - 查找和过滤
    - 操作 DOM：
        - jQuery 对象的 `text()` 和 `html()` 方法分别获取(亦可更改)节点的文本和原始 HTML 文本
        - 修改 CSS
        - 操作表单
        - 修改 DOM 结构: 添加删除等
    - 事件： JavaScript 在浏览器中以单线程模式运行，页面加载后，一旦页面上所有的 JavaScript 代码被执行完后，就只能依赖触发事件来执行 JavaScript 代码。
        - 浏览器在接收到用户的鼠标或键盘输入后，会自动在对应的 DOM 节点上触发相应的事件。如果该节点已经绑定了对应的 JavaScript 处理函数，该函数就会自动调用。
    - AJAX：用 jQuery 的相关对象来处理 AJAX，不但不需要考虑浏览器问题，代码也能大大简化。

      jQuery在全局对象 `jQuery`（也就是 `$`）绑定了 `ajax()` 函数，可以处理 AJAX 请求。

      `get()` 方法

      `post()`

      `getJSON()` 方法：快速通过 GET 获取一个 JSON 对象

2.2.2 Node.js

- 基于 JavaScript 语言和 V8 引擎的开源 Web 服务器项目
- Node.js 作为 JS 的运行环境运行在服务器端，作为 web server 运行在本地，作为打包工具或者构建工具。具体地可以应用于搭建分布式服务器框架，物联网开发框架，通讯，爬虫，开发动态网站，为各类网络应用提供API接口等。
- 模块: 一个 .js 文件就称之为一个模块。模块能提高代码的可维护性。
    - 要在模块中对外输出变量，用：

      ```
      module.exports = variable;
      ```

      输出的变量可以是任意对象、函数、数组等等。

      要引入其他模块输出的对象，用：

      ```
      var foo = require('other_module');
      ```

      引入的对象具体是什么，取决于引入模块输出的对象。
    - 基本模块
        - `fs`: 文件系统模块，负责读写文件(一般用异步读)
        - `stream`： Node.js 提供的又一个仅在服务区端可用的模块，目的是支持“流”这种数据结构，可借此实现更方便的文件读写，复制等。

```
fs.createWriteStream()
fs.createReadStream()
pipe()
```

- `http`： 提供`request`和`response`等对象，封装http请求和响应。

  一个简单的Web程序示例:

```
'use strict';

// 导入http模块:
var http = require('http');

// 创建http server，并传入回调函数:
var server = http.createServer(function (request, response) {
    // 回调函数接收request和response对象,
    // 获得HTTP请求的method和url:
    console.log(request.method + ': ' + request.url);
    // 将HTTP响应200写入response, 同时设置Content-Type: text/html:
    response.writeHead(200, {'Content-Type': 'text/html'});
    // 将HTTP响应的HTML内容写入response:
    response.end('<h1>Hello world!</h1>');
});

// 让服务器监听8080端口:
server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');
```

- `path`: 处理本地文件目录,可结合 http 模块，stream 模块等实现文件服务器。

- Web 开发：客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取 Web 页面，并把 Web 页面展示给用户即可。
  - koa：Express 的下一代基于 Node.js 的 web 框架
  - Nunjucks: JavaScript 编写的模板引擎
  - MVC：Model-View-Controller 模型
  - MySQL 数据库使用
  - WebSocket：HTML5 中协议，目的是在浏览器和服务器之间建立一个不受限的双向通信的通道（传统的HTTP协议是一个请求 - 响应协议，请求必须先由浏览器发给服务器，服务器才能响应这个请求，再把数据发送给浏览器，而服务器不能主动发数据给浏览器）。

  我们此次项目可能倾向于使用 Node.js 来搭建 web 服务器，一方面考虑到 Node.js 本就是打包后的 wasm 文件运行环境，实际上 Rust-wasm-js 也是基于 npm 包管理器，所以使用 Node.js 来搭建 web 服务器会有更好的兼容性；另一方面原本17级的 Apache + Tomcat + Servlet + Java web 本身对 java 依赖较高，而如果我们项目再过多使用 java，可能会因为跨度太大而导致调试复杂，所以暂时考虑放弃此框架。但我们还是会在17级已完成的前端基础上进行部署，毕竟我们项目的核心不是在前端页面上，而是在分布式文件系统的实现及部署。

创新点

1 Rust改写层面的创新

Rust 的优势：

- 低资源占用

控制资源使用，将内存与 CPU 占用降到最低。大多数运行条件下，一个Rust 程序比 Java 消耗的内存会少上一个数量级。

- 安全可靠

  Rust 的强大类型检查可防止多种级别的 Bug，确保开发者可随时明确状态是共享还是可变。在部署之前通过捕捉故障点来获得帮助。

- 生命周期及所有权规则

  虽然 Java 为使 GC 系统可管理，采用不分配内存的方式努力完善了内存回收机制，但有时却会导致代码过于复杂。Rust的生命周期及所有权规则，使得其可在没有GC（垃圾回收器）的情况下获取对象，使程序更少的出错。

- 优秀的鲁棒性

  Rust 在调试模式下的溢出检查，使得开发人员在测试期间能够发现更多问题，而发布模式下进行封装时不作检查，也提高了发布版本的执行效率。而Java的整型操作没有溢出检查。Rust 各种高要求的检查，与默认时的引用不变性，造就了 Rust 出色的鲁棒性。

- 错误处理

  任何一个线程发生 “panics” 时，都会被 Rust 认为是 RuntimeExceptions，Rust 会立即终止线程。并且 Rust 返回的错误信息比 Java 更具体，可以帮助程序员更好的理解错误点，完善代码。

2 rust-wasm-js使用的创新

- Rust 和 WebAssembly 尚处于蓬勃发展阶段，存在巨大潜力与上升空间。这样的尝试本身带有一定的不确定性及挑战性，但也在一定程度上经过了我们和社区其他开发者们的检验，这正是我们的创新点所在。
- Rust 和 WebAssembly 的结合可以让 Rust 在现存的 JavaScript 前端中使用。Rust 和 WebAssembly 能很好地将 JavaScript 工具如 npm 等结合在一起，让我们可以直接利用原本17级的部分前端实现。一系列工具链的出现，使得 Rust 与 JavaScript 互调，WebAssembly 与 JavaScript 的交互成为可能，并且可把 Rust 编译为WebAssembly，让我们得以充分利用 Rust，WebAssembly，JS 各自的优势。这正是值得尝试的地方，我们也期待这样的构思能够完善地实现，为 Rust 和 WebAssembly 社区做出一定贡献。

3 兼容性

- 充分利用 WebAssembly 的高效且可移植性，我们可在不同平台上以接近本机的性能执行 WebAssembly 代码。
- wasm-pack 等 Rust 里相关包的存在，使得我们可以生成 Rust 的 WebAssembly 软件包，与 JavaScript，Node.js，浏览器互操作，打通前端与后端界线，使 Rust 等语言能在 Web 上运行。

日程规划

第7周——第8周　客户端程序部分改写

第9周——第10周　服务器程序部分改写

第11周——第12周　把 Rust 改写成 WebAssembly

第13周——第16周　用 Node.js 搭建 web 服务器

第17周——第18周　善后工作

代码规范与文档规范

1. 代码规范

1.1 空格和缩进

- 行首缩进4空格(tab)
- 操作符左右加空格
- 分号和逗号之后加空格
- 函数间空一行
- 大括号不换行

  if(a>0){

  }

1.2 命名

- 通常，Rust 倾向于为“类型级”结构(类型和 traits)使用 `CamelCase` 而为“值级”结构使用 `snake_case` 。更确切的约定：

条目	约定
Crates	`snake_case` (但倾向于单个词)
Modules	`snake_case`
Types	`CamelCase`
Traits	`CamelCase`
Enum variants	`CamelCase`
Functions	`snake_case`
Methods	`snake_case`
General constructors	`new` 或 `with_more_details`
Conversion constructors	`from_some_other_type`
Local variables	`snake_case`

条目	约定
Static variables	`SCREAMING_SNAKE_CASE`
Constant variables	`SCREAMING_SNAKE_CASE`
Type parameters	简洁 `CamelCase`，通常单个大写字母：`T`
Lifetimes	短的小写: `'a`

1.3 注释

- int max() {

  ”函数注释：功能，参数“

  int a = 0　　#注释，尽量每一行注释左对齐

  }

2. 文档规范

2.1 标题

1，2，3级标题(5　5.1　5.1.2)

- 1
  - 2
    - 3

2.2 文本

全角中文字符与半角英文字符之间，应有一个半角空格。

2.3 参考链接

题目 + 超链接

参考文献

erasure code 的 Rust 实现

标准库：

- `std::net:TcpListener`
- `std::net::TcpStream`
- `std::fs`
- `std::net`
- `std::net`
- `std::traits::query`

套接字（Socket）简介

Rust 调用 Java 方法　j4rs

Rust and WebAssembly Documentation

编译 Rust 为 WebAssembly

npm

parcel

WebAssembly 时代，Rust 也想成为 Web 语言

nodejs和npm关系

廖雪峰官方网站

Rust 中文社区代码风格

中文技术文档的写作规范