中文标准库：https://rustwiki.org/zh-CN/std/

1 Rust IO

Rust 语言 IO 输入输出的三大块内容：**从标准输入读取数据、把数据写入标准输出、命令行参数。**

Rust 标准库通过两个特质 (Trait) 来组织 IO 输入输出

- Read 特质用于 读
- Write 特质用于 写

特质	说明	范例
Read	包含了许多方法用于从输入流读取字节数据	Stdin,File
Write	包含了许多方法用于向输出流中写入数据，包含字节数据和 UTF-8 数据两种格式	Stdout,File

Read Trait 特质 / 标准输入流
Read 特质是一个用于从输入流读取字节的组件。输入流包括 标准输入、键盘、鼠标、命令行、文件等等。
Read 特质中的 *read_line()* 方法用于从输入流中读取一行的字符串数据。它的相关信息如下

Trait 特质	方法	说明
Read read_line(&mut line)->Result	从输入流中读取一行的字符串数据并存储在 line 参数中。返回 Result 枚举，如果成功则返回读取的字节数。	

1.1 接收命令行参数

命令行程序是计算机程序最基础的存在形式，几乎所有的操作系统都支持命令行程序并将可视化程序的运行基于命令行机制。
命令行程序必须能够接收来自命令行环境的参数，这些参数往往在一条命令行的命令之后以空格符分隔。
在很多语言中（如 Java 和 C/C++）环境参数是以主函数的参数（常常是一个字符串数组）传递给程序的，但在 Rust 中主函数是个无参函数，环境参数需要开发者通过 std::env 模块取出，
ex1_1_1.rs

```
fn main() {
    let args = std::env::args();
    println!("{:?}", args);
}
```

ex1_1_2.rs打印参数个数和 遍历参数

```rust
fn main() {
    let args = std::env::args();
    println!("size = {}", args.len());
    for arg in args {
        println!("{}", arg);
    }
}
```

ex1_1_3.rs 将参数值保存进变量

```rust
// use std::env;
fn main() {
    // let args = std::env::args();
    let args: Vec<String> = std::env::args().collect();
    println!("size = {}",  args.len());
    let a = &args[1];
    println!("a = {}", a);
    let b = &args[2];
    println!("b = {}", b);
}
```

1.2 文件读取

中文库：https://rustwiki.org/zh-CN/std/fs/index.html

Rust 标准库提供了大量的模块和方法用于读写文件。
Rust 语言使用结构体 **File** 来描述/展现一个文件。
结构体 **File** 有相关的成员变量或函数用于表示程序可以对文件进行的某些操作和可用的操作方法。
所有对结构体 **File** 的操作方法都会返回一个 **Result** 枚举。
下表列出了一些常用的文件读写方法。

模块	方法/方法签名	说明
std::fs::File	open() / pub fn open(path: P) -> Result	静态方法，以 只读 模式打开文件
std::fs::File	create() / pub fn create(path: P) -> Result	静态方法，以 可写 模式打开文件。如果文件存在则清空旧内容。如果文件不存在则新建
std::fs::remove_file	remove_file() / pub fn remove_file(path: P) -> Result<()>	从文件系统中删除某个文件
std::fs::OpenOptions	append() / pub fn append(&mut self, append: bool) -> &mut OpenOptions	设置文件模式为 追加
std::io::Writes	write_all() / fn write_all(&mut self, buf: &[u8]) -> Result<()>	将 buf 中的所有内容写入输出流
std::io::Read	read_to_string() / fn read_to_string(&mut self, buf: &mut String) -> Result	读取所有内容转换为字符串后追加到 buf 中

1.2.1 Rust 打开文件

Rust 标准库中的 std::fs::File 模块提供了静态方法 open() 用于打开一个文件并返回文件句柄。
open() 函数的原型如下
pub fn open(path: **P**) -> Result
open() 函数用于以只读模式打开一个已经存在的文件，如果文件不存在，则会抛出一个错误。如果文件不可读，那么也会抛出一个错误。
ex1_2_1.rs

```rust
fn main() {
    let file = std::fs::File::open("data.txt").unwrap();
    println!("文件打开成功：{:?}",file);
}
```

1.2.2 Rust 创建文件

Rust 标准库中的 std::fs::File 模块提供了静态方法 create() 用于创建一个文件并返回创建的文件句柄。
create() 函数的原型如下
pub fn create(path: **P**) -> Result
create() 函数用于创建一个文件并返回创建的文件句柄。如果文件已经存在，则会内部调用 open() 打开文件。如果创建失败，比如目录不可写，则会抛出错误
ex1_2_2.rs

```rust
fn main() {
   let file = std::fs::File::create("data.txt").expect("create failed");
   println!("文件创建成功:{:?}",file);
}
```

###

1.2.3 Rust 写入文件

Rust 语言标准库 std::io::Writes 提供了函数 write_all() 用于向输出流写入内容。
因为文件流也是输出流的一种，所以该函数也可以用于向文件写入内容。
write_all() 函数在模块 std::io::Writes 中定义，它的函数原型如下
fn write_all(**&mut** self, buf: **&**[**u8**]) -> Result**<**()**>**
write_all() 用于向当前流写入 buf 中的内容。如果写入成功则返回写入的字节数，如果写入失败则抛出错误
ex1_2_3.rs

```rust
use std::io::Write;
fn main() {
   let mut file = std::fs::File::create("data.txt").expect("create failed");
   file.write_all("从零蛋开始教程".as_bytes()).expect("write failed");
   file.write_all("\n简单编程".as_bytes()).expect("write failed");
   println!("data written to file" );
}
```

1.2.4 Rust 读取文件

Rust 读取内容的一般步骤为:

1. 使用 open() 函数打开一个文件
2. 然后使用 read_to_string() 函数从文件中读取所有内容并转换为字符串。

open() 函数我们前面已经介绍过了，这次我们主要来讲讲 read_to_string() 函数。
read_to_string() 函数用于从一个文件中读取所有剩余的内容并转换为字符串。
read_to_string() 函数的原型如下
fn read_to_string(**&mut** self, buf: **&mut** String) -> Result
read_to_string() 函数用于读取文件中的所有内容并追加到 buf 中，如果读取成功则返回读取的字节数，如果读取失败则抛出错误。
ex1_2_4.rs

```rust
use std::io::Read;

fn main(){
   let mut file = std::fs::File::open("data.txt").unwrap();
   let mut contents = String::new();
   file.read_to_string(&mut contents).unwrap();
   print!("{}", contents);
}
```

###

1.2.5 追加内容到文件末尾

Rust 核心和标准库并没有提供直接的函数用于追加内容到文件的末尾。
但提供了函数 append() 用于将文件的打开模式设置为 追加。
当文件的模式设置为 追加 之后，写入文件的内容就不会代替原先的旧内容而是放在旧内容的后面。
函数 append() 在模块 std::fs::OpenOptions 中定义，它的函数原型为
pub fn append(**&mut** self, append: **bool**) -> **&mut** OpenOptions
ex1_2_5.rs

```rust
use std::fs::OpenOptions;
use std::io::Write;

fn main() {
   let mut file = OpenOptions::new().append(true).open("data.txt").expect(
      "cannot open file");
   file.write_all("www.baidu.com".as_bytes()).expect("write failed");
   file.write_all("\n从零开始教程".as_bytes()).expect("write failed");
   file.write_all("\n简单编程".as_bytes()).expect("write failed");
   println!("数据追加成功");
}
```

1.2.6 删除文件

Rust 标准库 std::fs 提供了函数 remove_file() 用于删除文件。
remove_file() 函数的原型如下
pub fn remove_file<P: AsRef>(path: **P**) -> Result<()>
注意，删除可能会失败，即使返回结果为 OK，也有可能不会立即就删除。

ex1_2_6.rs

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

1.2.7 复制文件

Rust 标准库没有提供任何函数用于复制一个文件为另一个新文件。
但我们可以使用上面提到的函数和方法来实现文件的复制功能。
下面的代码，我们模仿简单版本的 copy 命令
copy old_file_name new_file_name

具体的 Rust 代码如下
ex1_2_7.rs

```
use std::io::Read;
use std::io::Write;

fn main() {
   let mut command_line: std::env::Args = std::env::args();
   command_line.next().unwrap();

   // 跳过程序名
   // 原文件
   let source = command_line.next().unwrap();

   // 新文件
   let destination = command_line.next().unwrap();
   let mut file_in = std::fs::File::open(source).unwrap();
   let mut file_out = std::fs::File::create(destination).unwrap();
   let mut buffer = [0u8; 4096];
   loop {
      let nbytes = file_in.read(&mut buffer).unwrap();
      file_out.write(&buffer[..nbytes]).unwrap();
      if nbytes < buffer.len() { break; }
   }
}
```

2 网络编程

英文库： https://doc.rust-lang.org/std/net/struct.TcpStream.html
中文库：https://rustwiki.org/zh-CN/std/net/struct.TcpStream.html

ex2_server.rs

```
use std::thread;        // 线程
use std::net::{TcpListener, TcpStream, Shutdown}; // 网络
use std::io::{Read,Write};// io读写

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // 50字节的buffer，初始化为0
    while match stream.read(&mut data) {
        Ok(size) => {
            // 回写数据
            stream.write(&data[0..size]).unwrap();
            true  //返回的是 while循环需要处理的值
        },
        Err(e)=> {
            println!("An error:{} occurred, terminating connection with {}", e,
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false  // 数据读取完毕
        }
    }{}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
```

```
    println!("Server listening on port 3333");
    // accept connections and process them, spawning a new thread for each one
    for stream in listener.incoming() {
        match stream {
            Ok(stream)=> {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move||{
                    //连接成功
                    handle_client(stream);
                });
            },
            Err(e)=> {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // 关闭服务
    drop(listener);
}
```

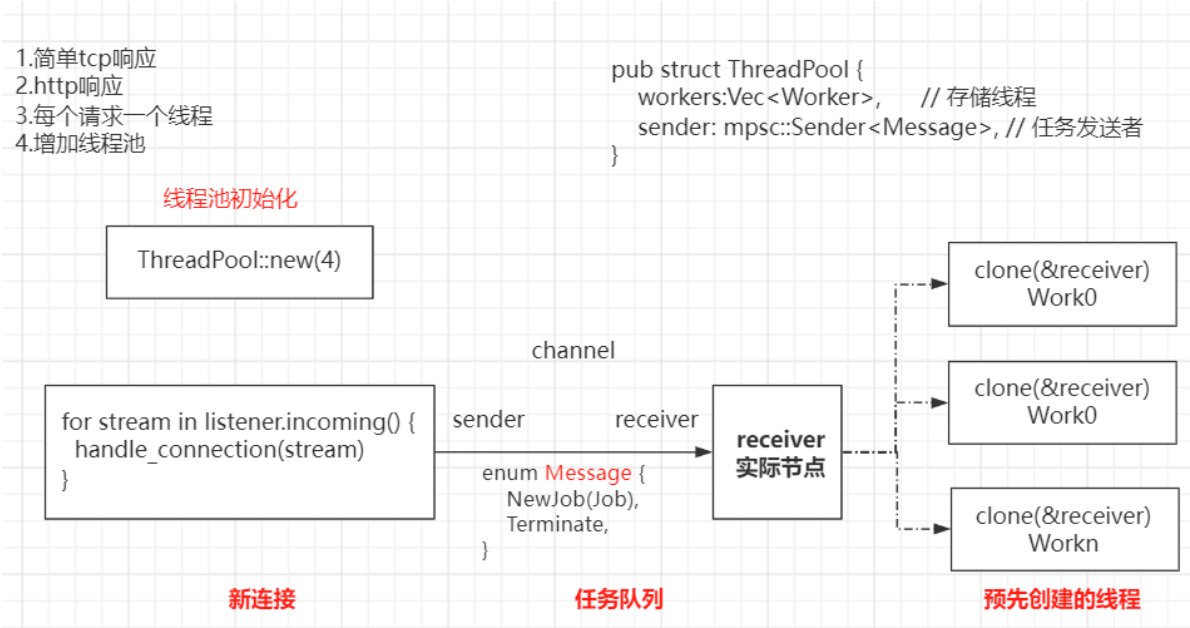
ex2_client.rs

```
use std::net::{TcpStream};
use std::io::{Read, Write};
use std::str::from_utf8;

fn main() {
    match TcpStream::connect("localhost:3333") {
        Ok(mut stream) => {
            println!("Successfully connected to server in port 3333");

            let msg =  b"Hello!";
            stream.write(msg).unwrap();
            println!("Sent Hello, awaiting reply...");
            let mut data = [0 as u8; 6]; // using 6 byte buffer
            match stream.read_exact(&mut data) {
               Ok(_) => {
                    if &data == msg {
                         println!("Reply is ok!");
                    } else {
                        let text = from_utf8(&data).unwrap();
                        println!("Unexpected reply: {}", text);
                    }
               } ,
               Err(e) => {
                    println!("Failed to receive data: {}", e);
               }
            }
        },
        Err(e) => {
            println!("Failed to connect: {}", e);
        }
    }
    println!("Terminated.");
}
```

3 线程池

将之前所学的知识实现线程池，**手把手写代码，加深对之前所学语法的理解。**
通过浏览器访问网页，服务器通过线程池处理发送数据。
涉及的知识点：

- usestd::fs;
- usestd::io::prelude::*;
- usestd::net::TcpListener;
- usestd::net::TcpStream;
- usestd::thread;
- usestd::time::Duration;
- usestd::sync::mpsc; // channel
- usestd::sync::Arc;
- usestd::sync::Mutex;
- usestd::thread;
- enum
- option
- Result<T, E>



代码：http_thread_pool

步骤：

1. ## 构建单线程web服务器

    1. ex3_1.rs监听tcp连接
    2. ex3_2.rs 读取完成，创建handle_connection 服务器收到请求
    3. ex3_3.rs编写响应 返回简单的html
    4. ex3_4.rs返回真正的响应，读取hello.html，并返回给浏览器
    5. ex3_5.rs验证请求有效性并选择性地返回，主要验证get
    6. ex3_6.rs 错误的访问连接，比如http://192.168.0.143:7878/else， 返回404访问错误
    7. ex3_7.rs 少许重构if和else模块，只在分支代码包含有区别的部分
    8. ex3_8.rs**模拟一个较慢的请求处理**，目前服务器只有一个线程处理，那则会影响其他的请求 b"GET /sleep HTTP/1.1\r\n"； sleep(Duration::from_secs(5))

2. ## 使用线程池改进吞吐量（线程池是一组预先分配出来的线程）

1. ex3_9.rs 每个连接 创建一个线程
2. ex3_10.rs通过线程池（有限数量的线程）代替每个连接创建一个线程
3. ex3_11 封装好 ThreadPool的 execute 、new接口

```
FnOnce  每个处理请求只执行一次；
Send约束的闭包才能从一个线程传递到另一个线程；
'static 我们不知道线程究竟会执行多久。
pub fn execute<F>(&self, f: F)
where
F: FnOnce() + Send + 'static,
{
}
```

4. ex3_12 用于存放线程的空间，thread::JoinHandle<()>

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
1. 为ThreadPool创建一个动态数组存放线程
pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}
2.定义封装存放单个线程的Worker
```

5. **ex3_13** 使用通道吧请求发送给线程
    1. 由ThreadPool创建通道并持有通道的发送端；
    2. 生成的每个Worker都会持有通道的接收端；
    3. 创建一个新的Job结构体来持有需要发送到通道中的闭包；
    4. 在execute方法中将它想要的任务传递给通道的发送端；
    5. Worker会在自己的线程中不断地查询接收端，并执行收取到的闭包任务。
    6. rust提供的通道是多生产者、单消费者模型，使用一个安全的方式共享和修改receiver，否则可能会触发竞争状态let receiver = Arc::new(Mutex::new(receiver)); 创建新的Worker时克隆Arc来增加引用计数，从而使所有的工作线程可以共享接收端的所有权。
    7. 实现execute方法 typeJob = Box<dyn FnOnce() + Send + 'static>；实现了对应的trait

```
pub fn execute<F>(&self, f: F)
   where
       F: FnOnce() + Send + 'static,
   {
       let job = Box::new(f);

       self.sender.send(job).unwrap();
   }
```

6. ex3_14 while let实现的循环，**这种方式影响性能**

优雅地停机实现

7. ex3_15 优雅地停机实现drop trait. **通过take方法提取Some变体的值**

4 其他参考

- [教程]Option的正确打开方式https://www.jianshu.com/p/ce5bddf4b335
- 理解 Rust 2018 edition 的两个新关键字 —— impl 和 dyn
  https://www.cnblogs.com/chen8840/p/12703463.html
- Rust语言圣经 https://github.com/sunface/rust-course