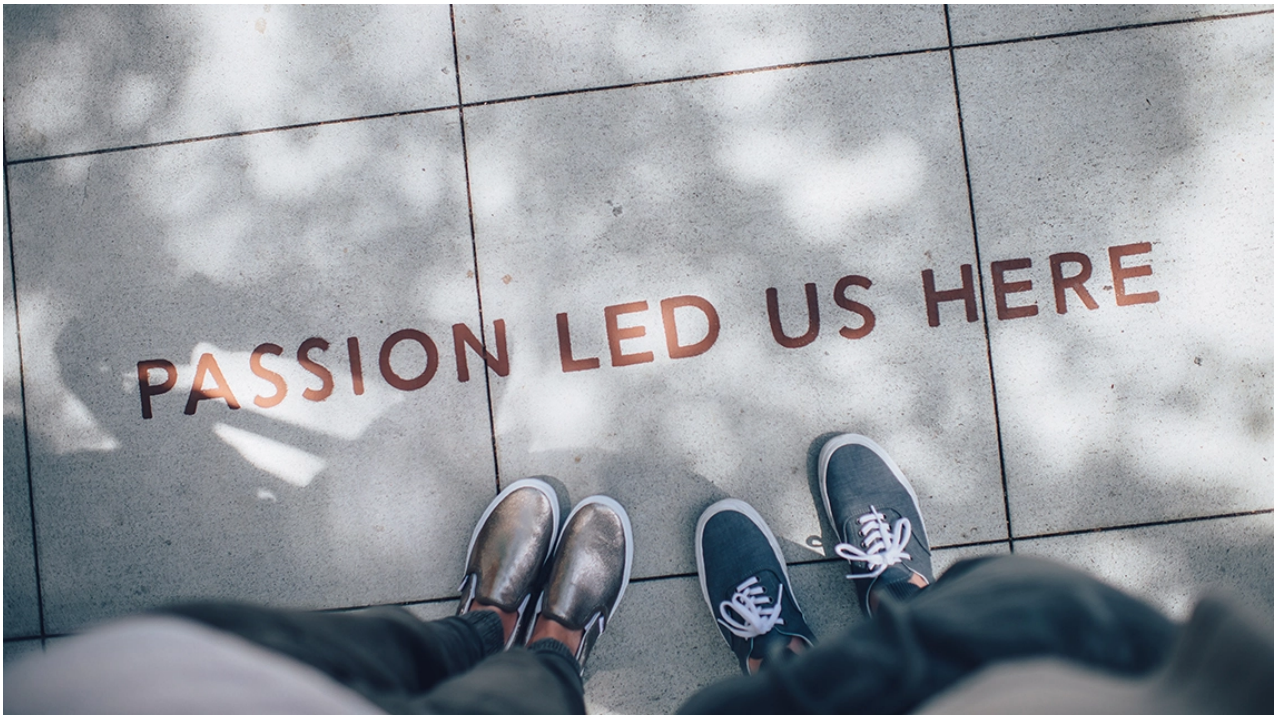
加餐｜期中测试：参考实现讲解

time.geekbang.org/column/article/425015



00:00

1.0x
讲述：陈天大小：4.74M时长：05:10

你好，我是陈天。

上一讲给你布置了一份简单的期中考试习题，不知道你完成的怎么样。今天我们来简单讲一讲实现，供你参考。

支持 grep 并不是一件复杂的事情，相信你在使用了 clap、glob、rayon 和 regex 后，都能写出类似的代码（伪代码）：

/// Yet another simplified grep built with Rust.

#[derive(Clap, Debug)]

#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]

#[clap(setting = AppSettings::ColoredHelp)]

pub struct GrepConfig {

/// regex pattern to match against file contents

```
pattern: String,
/// Glob of file pattern
glob: String,
}
impl GrepConfig {
pub fn matches(&self) -> Result<()> {
let regex = Regex::new(&self.pattern)?;
let files: Vec<_> = glob::glob(&self.glob)?.collect();
files.into_par_iter().for_each(|v| {
if let Ok(filename) = v {
if let Ok(file) = File::open(&filename) {
let reader = BufReader::new(file);
|- for (lineno, line) in reader.lines().enumerate() {
| if let Ok(line) = line {
| if let Some(_) = pattern.find(&line) {
| println!("{}: {}", lineno + 1, &line);
| }
| }
|- }
}
}
});
Ok(())
}
}
```

这个代码撰写的感觉和 Python 差不多，除了阅读几个依赖花些时间外，几乎没有难度。

不过，这个代码不具备可测试性，会给以后的维护和扩展带来麻烦。我们来看看如何优化，使这段代码更加容易测试。

如何写出好实现

首先，我们要剥离主要逻辑。

主要逻辑是什么？自然是对于单个文件的 grep，也就是代码中标记的部分。我们可以将它抽离成一个函数：

fn process(reader: BufReader<File>)

当然，从接口的角度来说，这个 process 函数定义得太死，如果不是从 File 中取数据，改天需求变了，也需要支持从 stdio 中取数据呢？就需要改动这个接口了。

所以可以使用泛型：

fn process<R: Read>(reader: BufReader<R>)

泛型参数 R 只需要满足 std::io::Read trait 就可以。

这个接口虽然抽取出来了，但它依旧不可测，因为它内部直接 println!，把找到的数据直接打印出来了。我们当然可以把要打印的行放入一个 Vec<String> 返回，这样就可以测试了。

不过，这是为了测试而测试，更好的方式是把输出的对象从 Stdout 抽象成 Write。现在 process 的接口变为：

fn process<R: Read, W: Write>(reader: BufReader<R>, writer: &mut Writer)

这样，我们就可以使用实现了 Read trait 的 &[u8] 作为输入，以及使用实现了 Write trait 的 Vec<u8> 作为输出，进行测试了。而在 rgrep 的实现时，我们用 File 作为输入，Stdout 作为输出。这样既满足了需求，让核心逻辑可测，还让接口足够灵活，可以适配任何实现了 Read 的输入以及实现了 Write 的输出。

好，有了这个思路，来看看我是怎么写这个 rgrep 的，供你参考。

首先 cargo new rgrep 创建一个新的项目。在 Cargo.toml 中，添加如下依赖：

[dependencies]

anyhow = "1"

clap = "3.0.0-beta.4" # 我们需要使用最新的 3.0.0-beta.4 或者更高版本

colored = "2"

glob = "0.3"

itertools = "0.10"

```
rayon = "1"

regex = "1"

thiserror = "1"
```

对于处理命令行的 clap，我们需要 3.0 的版本。不要在意 VS Code 插件提示你最新版本是 2.33，那是因为 beta 不算正式版本。

然后创建 src/lib.rs 和 src/error.rs，在 error.rs 中添加一些错误定义：

```
use thiserror::Error;

#[derive(Error, Debug)]

pub enum GrepError {

#[error("Glob pattern error")]

GlobPatternError(#[from] glob::PatternError),

#[error("Regex pattern error")]

RegexPatternError(#[from] regex::Error),

#[error("I/O error")]

IoError(#[from] std::io::Error),

}
```

它们都是需要进行转换的错误。thiserror 能够通过宏帮我们完成错误类型的转换。

在 src/lib.rs 中，添入如下代码：

```
use clap::{AppSettings, Clap};

use colored::*;

use itertools::Itertools;

use rayon::iter::{IntoParallelIterator, ParallelIterator};

use regex::Regex;

use std::{

fs::File,

io::{self, BufRead, BufReader, Read, Stdout, Write},
```

```
ops::Range,
path::Path,
};
mod error;
pub use error::GrepError;
/// 定义类型，这样，在使用时可以简化复杂类型的书写
pub type StrategyFn<W, R> = fn(&Path, BufReader<R>, &Regex, &mut W) -> Result<(), GrepError>;
/// 简化版本的 grep，支持正则表达式和文件通配符
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
pub struct GrepConfig {
/// 用于查找的正则表达式
pattern: String,
/// 文件通配符
glob: String,
}
impl GrepConfig {
/// 使用缺省策略来查找匹配
pub fn match_with_default_strategy(&self) -> Result<(), GrepError> {
self.match_with(default_strategy)
}
/// 使用某个策略函数来查找匹配
pub fn match_with(&self, strategy: StrategyFn<Stdout, File>) -> Result<(), GrepError> {
let regex = Regex::new(&self.pattern)?;
```

```
// 生成所有符合通配符的文件列表
let files: Vec<_> = glob::glob(&self.glob)?.collect();
// 并行处理所有文件
files.into_par_iter().for_each(|v| {
if let Ok(filename) = v {
if let Ok(file) = File::open(&filename) {
let reader = BufReader::new(file);
let mut stdout = io::stdout();
if let Err(e) = strategy(filename.as_path(), reader, &regex, &mut stdout) {
println!("Internal error: {:?}", e);
}
}
}
});
Ok(())
}
}
/// 缺省策略，从头到尾串行查找，最后输出到 writer
pub fn default_strategy<W: Write, R: Read>(
path: &Path,
reader: BufReader<R>,
pattern: &Regex,
writer: &mut W,
) -> Result<(), GrepError> {
let matches: String = reader
.lines()
```

```
.enumerate()
.map(|(lineno, line)| {
line.ok()
.map(|line| {
pattern
.find(&line)
.map(|m| format_line(&line, lineno + 1, m.range()))
})
.flatten()
})
.filter_map(|v| v.ok_or(()).ok())
.join("\n");
if !matches.is_empty() {
writer.write(path.display().to_string().green().as_bytes())?;
writer.write(b"\n")?;
writer.write(matches.as_bytes())?;
writer.write(b"\n")?;
}
Ok(())
}
/// 格式化输出匹配的行，包含行号、列号和带有高亮的第一个匹配项
pub fn format_line(line: &str, lineno: usize, range: Range<usize>) -> String {
let Range { start, end } = range;
let prefix = &line[..start];
format!(
"{0: >6}:{1: <3} {2}{3}{4}",
```

```
lineno.to_string().blue(),
// 找到匹配项的起始位置，注意对汉字等非 ascii 字符，我们不能使用 prefix.len()
// 这是一个 O(n) 的操作，会拖累效率，这里只是为了演示的效果
(prefix.chars().count() + 1).to_string().cyan(),
prefix,
&line[start..end].red(),
&line[end..]
)
}
```

和刚才的思路稍有不同的是，process 函数叫 default_strategy()。另外我们为 GrepConfig 提供了两个方法，一个是 match_with_default_strategy()，另一个是 match_with()，调用者可以自己传入一个函数或者闭包，对给定的 BufReader 进行处理。这是一种常用的解耦的处理方法。

在 src/lib.rs 里，继续撰写单元测试：

```
#[cfg(test)]
mod tests {
use super::*;
#[test]
fn format_line_should_work() {
let result = format_line("Hello, Tyr~", 1000, 7..10);
let expected = format!(
"{0: >6}:{1: <3} Hello, {2}~",
"1000".blue(),
"7".cyan(),
"Tyr".red()
);
assert_eq!(result, expected);
```

```
}

#[test]

fn default_strategy_should_work() {

let path = Path::new("src/main.rs");

let input = b"hello world!\nhey Tyr!";

let reader = BufReader::new(&input[..]);

let pattern = Regex::new(r"he\\w+").unwrap();

let mut writer = Vec::new();

default_strategy(path, reader, &pattern, &mut writer).unwrap();

let result = String::from_utf8(writer).unwrap();

let expected = [

String::from("src/main.rs"),

format_line("hello world!", 1, 0..5),

format_line("hey Tyr!\n", 2, 0..3),

];

assert_eq!(result, expected.join("\n"));

}

}
```

你可以重点关注测试是如何使用 default_strategy() 函数，而 match_with() 方法又是如何使用它的。运行 cargo test，两个测试都能通过。

最后，在 src/main.rs 中添加命令行处理逻辑：

```
use anyhow::Result;

use clap::Clap;

use rgrep::*;

fn main() -> Result<()> {

let config: GrepConfig = GrepConfig::parse();
```

```
config.match_with_default_strategy()?;

Ok(())

}
```

在命令行下运行：cargo run --quiet -- "Re[^\\s]+" "src/*.rs" ，会得到类似如下输出。注意，文件输出的顺序可能不完全一样，因为 rayon 是多个线程并行执行的。

```
❯ cargo run --quiet -- "Re[^\s]+" "src/*.rs"
src/main.rs
     1:13  use anyhow::Result;
     5:14  fn main() -> Result<()> {
src/error.rs
     7:14      #[error("Regex pattern error")]
     8:5       RegexPatternError(#[from] regex::Error),
src/lib.rs
     5:12  use regex::Regex;
     8:19      io::{self, BufRead, BufReader, Read, Stdout, Write},
    17:42  pub type StrategyFn<W, R> = fn(&Path, BufReader<R>, &Regex, &mut W) -> Result<(), GrepError>;
    32:50      pub fn match_with_default_strategy(&self) -> Result<(), GrepError> {
    37:69      pub fn match_with(&self, strategy: StrategyFn<Stdout, File>) -> Result<(), GrepError> {
    38:21          let regex = Regex::new(&self.pattern)?;
    45:37                      let reader = BufReader::new(file);
    59:38  pub fn default_strategy<W: Write, R: Read>(
    61:16      reader: BufReader<R>,
    62:15      pattern: &Regex,
    64:6   ) -> Result<(), GrepError> {
   127:25          let reader = BufReader::new(&input[..]);
   128:23          let pattern = Regex::new(r"he\w+").unwrap();
```

小结

rgrep 是一个简单的命令行工具，仅仅写了上百行代码，就完成了一个性能相当不错的简化版 grep。在不做复杂的接口设计时，我们可以不用生命周期，不用泛型，甚至不用太关心所有权，就可以写出非常类似脚本语言的代码。

从这个意义上讲，Rust 用来做一次性的、即用即抛型的代码，或者说，写个快速原型，也有用武之地；当我们需要更好的代码质量、更高的抽象度、更灵活的设计时，Rust 提供了足够多的工具，让我们将原型进化成更成熟的代码。

相信在做 rgrep 的过程中，你能感受到用 Rust 开发软件的愉悦。

今天我们就不布置思考题了，你可以多多体会 KV server 和 rgrep 工具的实现。恭喜你完成了 Rust 基础篇的学习，进度条过半，我们下节课进阶篇见。

欢迎你分享给身边的朋友，邀他一起讨论。

延伸阅读

在 YouTube 上，有一个新鲜出炉的视频：Visualizing memory layout of Rust's data types，用 40 分钟的时间，总结了我们前面基础篇二十讲里提到的主要数据结构的内存布局。我个人非常喜欢这个视频，因为它和我一直倡导的“厘清数据是如何在堆和栈上存储”的思路不谋而合，在这里也推荐给你。如果你想快速复习一下，查漏补缺，那么非常建议你花上一个小时时间仔细看一下这个视频。

16人觉得很赞给文章提建议


良师益友

Command + Enter 发表

0/2000字

提交留言

精选留言(5)

- 
  Honey拯救世界

  https://github.com/clap-rs/clap/issues/2917
  clap v3.0.0-beta.4 break change了，要调整一下

  作者回复: github 上面的代码已经改了

  2021-11-08

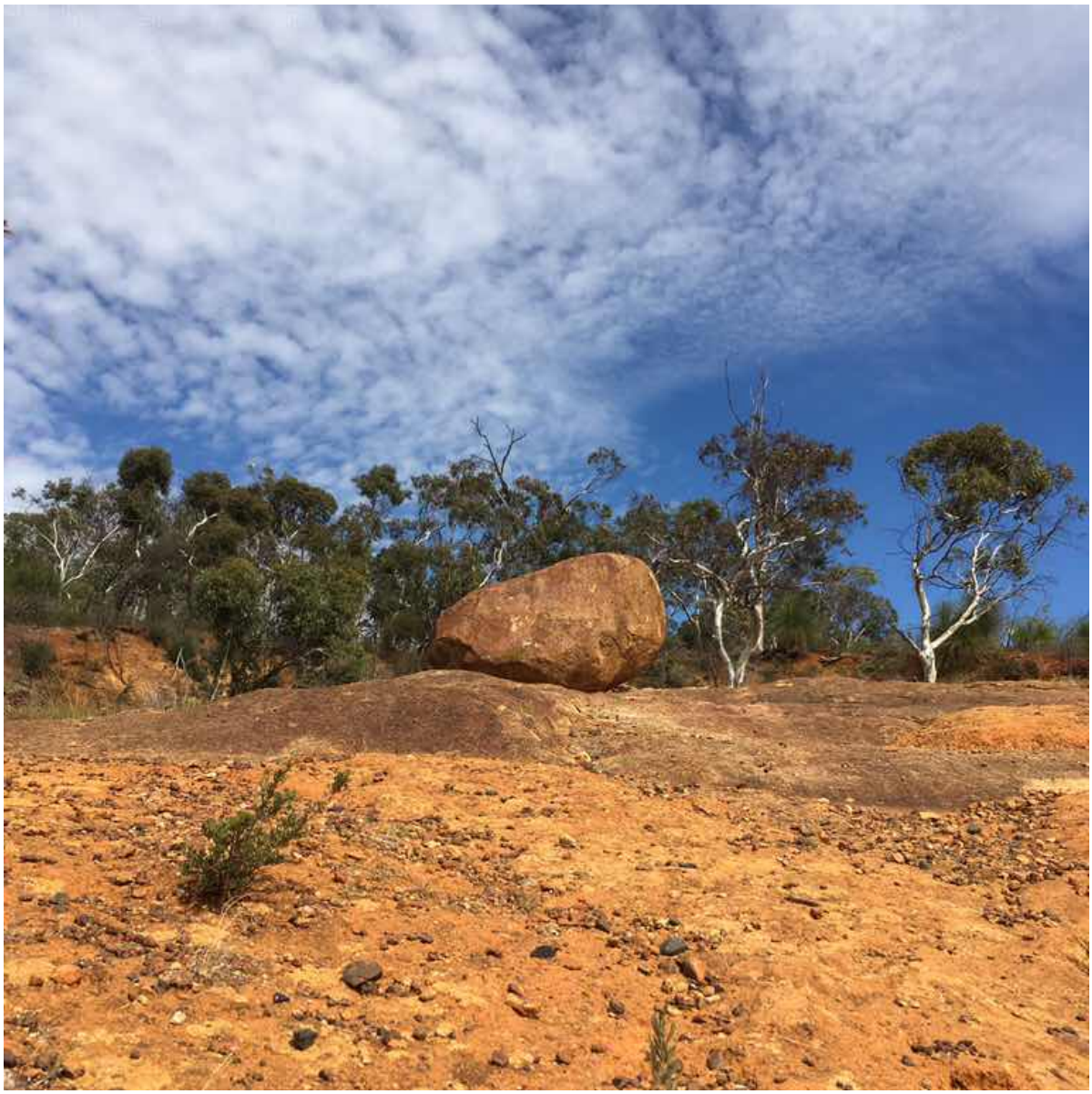
-

linuxfish

writer.write(path.display().to_string().green().as_bytes())?;

这一行的设置颜色貌似没有起作用，换了其它颜色也是没效果
作者回复: 你的 terminal 是否支持 ansi color？你需要使用支持颜色的 terminal

2021-11-01

- 

野山门

看完了附带的视频，对数据类型的内存结构有一个清晰的认识。感谢！

作者回复: 👍

2021-10-28

- 

记事本

老师,必须改成这样才可以换行生效啊 writer.write(b"\n")?;

作者回复: 呃，又是稿件粘贴的问题，自动多加了一个转义的 "\"。我让编辑帮忙更新。

你也可以看 github 上的完成代码：https://github.com/tyrchen/geektime-rust/blob/master/mid_term_rgrep/src/lib.rs。这个代码是经过测试和 cargo check/clippy 的。

2021-10-16

2

- 

记事本

use std::io::self self在这里指的是什么啊

作者回复: 就是 std::io。

一般 use std::io 就可以了，但如果想同时引入自己以及自己底下的结构，可以用 self，比如： use std::io::{self, Read};

2021-10-15

6

收起评论