04｜get hands dirty：来写个实用的CLI小工具

time.geekbang.org/column/article/412883

陈天 2021-08-30

00:00

1.0x
讲述：陈天大小：10.24M时长：11:10

你好，我是陈天。

在上一讲里，我们已经接触了 Rust 的基本语法。你是不是已经按捺不住自己的洪荒之力，想马上用 Rust 写点什么练练手，但是又发现自己好像有点“拔剑四顾心茫然”呢？

那这周我们就来玩个新花样，做一周“learning by example”的挑战，来尝试用 Rust 写三个非常有实际价值的小应用，感受下 Rust 的魅力在哪里，解决真实问题的能力到底如何。

你是不是有点担心，我才刚学了最基本语法，还啥都不知道呢，这就能开始写小应用了？那我碰到不理解的知识怎么办？

不要担心，因为你肯定会碰到不太懂的语法，但是，先不要强求自己理解，当成文言文抄写就可以了，哪怕这会不明白，只要你跟着课程节奏，通过撰写、编译和运行，你也能直观感受到 Rust 的魅力，就像小时候背唐诗一样。

好，我们开始今天的挑战。

HTTPie

为了覆盖绝大多数同学的需求，这次挑选的例子是工作中普遍会遇到的：写一个 CLI 工具，辅助我们处理各种任务。

我们就以实现 HTTPie 为例，看看用 Rust 怎么做 CLI。HTTPie 是用 Python 开发的，一个类似 cURL 但对用户更加友善的命令行工具，它可以帮助我们更好地诊断 HTTP 服务。
下图是用 HTTPie 发送了一个 post 请求的界面，你可以看到，相比 cURL，它在可用性上做了很多工作，包括对不同信息的语法高亮显示：

```
❯ http post httpbin.org/post greeting=hola name=Tyr
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 541
Content-Type: application/json
Date: Tue, 24 Aug 2021 16:56:38 GMT
Server: gunicorn/19.9.0

{
    "args": {},
    "data": "{\"greeting\": \"hola\", \"name\": \"Tyr\"}",
    "files": {},
    "form": {},
    "headers": {
        "Accept": "application/json, */*;q=0.5",
        "Accept-Encoding": "gzip, deflate",
        "Content-Length": "35",
        "Content-Type": "application/json",
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/2.4.0",
```

你可以先想一想，如果用你最熟悉的语言实现 HTTPie ，要怎么设计、需要用到些什么库、大概用多少行代码？如果用 Rust 的话，又大概会要多少行代码？

带着你自己的这些想法，开始动手用 Rust 构建这个工具吧！我们的目标是，用大约 200 行代码实现这个需求。

功能分析

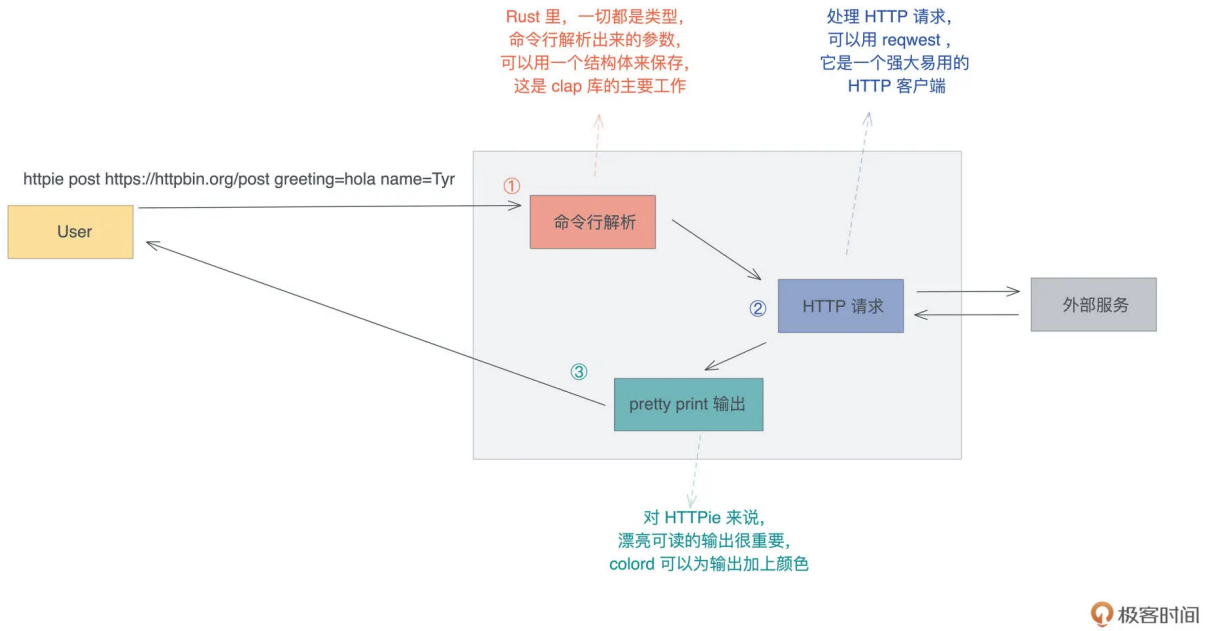
要做一个 HTTPie 这样的工具，我们先梳理一下要实现哪些主要功能：

首先是做命令行解析，处理子命令和各种参数，验证用户的输入，并且将这些输入转换成我们内部能理解的参数；

之后根据解析好的参数，发送一个 HTTP 请求，获得响应；

最后用对用户友好的方式输出响应。

这个流程你可以再看下图：



我们来看要实现这些功能对应需要用到的库：

对于命令行解析，Rust 有很多库可以满足这个需求，我们今天使用官方比较推荐的 clap。
对于 HTTP 客户端，在上一讲我们已经接触过 reqwest，我们就继续使用它，只不过我们这次尝个鲜，使用它的异步接口。
对于格式化输出，为了让输出像 Python 版本的 HTTPie 那样显得生动可读，我们可以引入一个命令终端多彩显示的库，这里我们选择比较简单的 colored。
除此之外，我们还需要一些额外的库：用 anyhow 做错误处理、用 jsonxf 格式化 JSON 响应、用 mime 处理 mime 类型，以及引入 tokio 做异步处理。

CLI 处理

好，有了基本的思路，我们来创建一个项目，名字就叫 httpie：

cargo new httpie

cd httpie

然后，用 VSCode 打开项目所在的目录，编辑 Cargo.toml 文件，添加所需要的依赖（注意：以下代码用到了 beta 版本的 crate，可能未来会有破坏性更新，如果在本地无法编译，请参考 GitHub repo 中的代码）：

[package]

name = "httpie"

version = "0.1.0"

edition = "2018"

[dependencies]

anyhow = "1" # 错误处理

```
clap = "3.0.0-beta.4" # 命令行解析
colored = "2" # 命令终端多彩显示
jsonxf = "1.1" # JSON pretty print 格式化
mime = "0.3" # 处理 mime 类型
reqwest = { version = "0.11", features = ["json"] } # HTTP 客户端
tokio = { version = "1", features = ["full"] } # 异步处理库
```

我们先在 main.rs 添加处理 CLI 相关的代码：

```
use clap::{AppSettings, Clap};
// 定义 HTTPie 的 CLI 的主入口，它包含若干个子命令
// 下面 /// 的注释是文档，clap 会将其作为 CLI 的帮助
/// A naive httpie implementation with Rust, can you imagine how easy it is?
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
struct Opts {
#[clap(subcommand)]
subcmd: SubCommand,
}
// 子命令分别对应不同的 HTTP 方法，目前只支持 get / post
#[derive(Clap, Debug)]
enum SubCommand {
Get(Get),
Post(Post),
// 我们暂且不支持其它 HTTP 方法
}
// get 子命令
/// feed get with an url and we will retrieve the response for you
#[derive(Clap, Debug)]
struct Get {
```

```
/// HTTP 请求的 URL

url: String,

}

// post 子命令。需要输入一个 URL，和若干个可选的 key=value，用于提供 json body

/// feed post with an url and optional key=value pairs. We will post the data

/// as JSON, and retrieve the response for you

#[derive(Clap, Debug)]

struct Post {

/// HTTP 请求的 URL

url: String,

/// HTTP 请求的 body

body: Vec<String>,

}

fn main() {

let opts: Opts = Opts::parse();

println!("{:?}", opts);

}
```

代码中用到了 clap 提供的宏来让 CLI 的定义变得简单，这个宏能够生成一些额外的代码帮我们处理 CLI 的解析。通过 clap ，我们只需要先用一个数据结构 T 描述 CLI 都会捕获什么数据，之后通过 T::parse() 就可以解析出各种命令行参数了。parse() 函数我们并没有定义，它是 #[derive(Clap)] 自动生成的。

目前我们定义了两个子命令，在 Rust 中子命令可以通过 enum 定义，每个子命令的参数又由它们各自的数据结构 Get 和 Post 来定义。

我们运行一下：

```
❯ cargo build --quiet && target/debug/httpie post httpbin.org/post a=1 b=2

Opts { subcmd: Post(Post { url: "httpbin.org/post", body: ["a=1", "b=2"] }) }
```

默认情况下，cargo build 编译出来的二进制，在项目根目录的 target/debug 下。可以看到，命令行解析成功，达到了我们想要的功能。

加入验证

然而，现在我们还没对用户输入做任何检验，如果有这样的输入，URL 就完全解析错误了：

```
❯ cargo build --quiet && target/debug/httpie post a=1 b=2

Opts { subcmd: Post(Post { url: "a=1", body: ["b=2"] }) }
```

所以，我们需要加入验证。输入有两项，就要做两个验证，一是验证 URL，另一个是验证 body。

首先来验证 URL 是合法的：

```
use anyhow::Result;
use reqwest::Url;
#[derive(Clap, Debug)]
struct Get {
/// HTTP 请求的 URL
#[clap(parse(try_from_str = parse_url))]
url: String,
}
fn parse_url(s: &str) -> Result<String> {
// 这里我们仅仅检查一下 URL 是否合法
let _url: Url = s.parse()?;
Ok(s.into())
}
```

clap 允许你为每个解析出来的值添加自定义的解析函数，我们这里定义了个 parse_url 检查一下。

然后，我们要确保 body 里每一项都是 key=value 的格式。可以定义一个数据结构 KvPair 来存储这个信息，并且也自定义一个解析函数把解析的结果放入 KvPair：

```
use std::str::FromStr;
use anyhow::{anyhow, Result};
#[derive(Clap, Debug)]
struct Post {
/// HTTP 请求的 URL
#[clap(parse(try_from_str = parse_url))]
url: String,
/// HTTP 请求的 body
#[clap(parse(try_from_str=parse_kv_pair))]
body: Vec<KvPair>,
}
/// 命令行中的 key=value 可以通过 parse_kv_pair 解析成 KvPair 结构
```

```
#[derive(Debug)]
struct KvPair {
k: String,
v: String,
}
/// 当我们实现 FromStr trait 后，可以用 str.parse() 方法将字符串解析成 KvPair
impl FromStr for KvPair {
type Err = anyhow::Error;
fn from_str(s: &str) -> Result<Self, Self::Err> {
// 使用 = 进行 split，这会得到一个迭代器
let mut split = s.split("=");
let err = || anyhow!(format!("Failed to parse {}", s));
Ok(Self {
// 从迭代器中取第一个结果作为 key，迭代器返回 Some(T)/None
// 我们将其转换成 Ok(T)/Err(E)，然后用 ? 处理错误
k: (split.next().ok_or_else(err)?).to_string(),
// 从迭代器中取第二个结果作为 value
v: (split.next().ok_or_else(err)?).to_string(),
})
}
}
/// 因为我们为 KvPair 实现了 FromStr，这里可以直接 s.parse() 得到 KvPair
fn parse_kv_pair(s: &str) -> Result<KvPair> {
Ok(s.parse()?)
}
```

这里我们实现了一个 FromStr trait，可以把满足条件的字符串转换成 KvPair。FromStr 是 Rust 标准库定义的 trait，实现它之后，就可以调用字符串的 parse() 泛型函数，很方便地处理字符串到某个类型的转换了。
这样修改完成后，我们的 CLI 就比较健壮了，可以再测试一下：

```
❯ cargo build --quiet
```

```
❯ target/debug/httpie post https://httpbin.org/post a=1 b

error: Invalid value for '<BODY>...': Failed to parse b

For more information try --help

❯ target/debug/httpie post abc a=1

error: Invalid value for '<URL>': relative URL without a base

For more information try --help

target/debug/httpie post https://httpbin.org/post a=1 b=2

Opts { subcmd: Post(Post { url: "https://httpbin.org/post", body: [KvPair { k: "a", v: "1" }, KvPair { k: "b", v: "2" }] }) }
```

Cool，我们完成了基本的验证，不过很明显可以看到，我们并没有把各种验证代码一股脑塞在主流程中，而是通过实现额外的验证函数和 trait 来完成的，这些新添加的代码，高度可复用且彼此独立，并不用修改主流程。

这非常符合软件开发的开闭原则（Open-Closed Principle）：Rust 可以通过宏、trait、泛型函数、trait object 等工具，帮助我们更容易写出结构良好、容易维护的代码。
目前你也许还不太明白这些代码的细节，但是不要担心，继续写，今天先把代码跑起来就行了，不需要你搞懂每个知识点，之后我们都会慢慢讲到的。

HTTP 请求

好，接下来我们就继续进行 HTTPie 的核心功能：HTTP 的请求处理了。我们在 main() 函数里添加处理子命令的流程：

```
use reqwest::{header, Client, Response, Url};

#[tokio::main]

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

let opts: Opts = Opts::parse();

// 生成一个 HTTP 客户端

let client = Client::new();

let result = match opts.subcmd {

SubCommand::Get(ref args) => get(client, args).await?,

SubCommand::Post(ref args) => post(client, args).await?,

};

Ok(result)

}
```

注意看我们把 main 函数变成了 async fn，它代表异步函数。对于 async main，我们需要使用 #[tokio::main] 宏来自动添加处理异步的运行时。

然后在 main 函数内部，我们根据子命令的类型，我们分别调用 get 和 post 函数做具体处理，这两个函数实现如下：

```
use std::{collections::HashMap, str::FromStr};

async fn get(client: Client, args: &Get) -> Result<()> {

let resp = client.get(&args.url).send().await?;

println!("{:?}", resp.text().await?);

Ok(())

}

async fn post(client: Client, args: &Post) -> Result<()> {

let mut body = HashMap::new();

for pair in args.body.iter() {

body.insert(&pair.k, &pair.v);

}

let resp = client.post(&args.url).json(&body).send().await?;

println!("{:?}", resp.text().await?);

Ok(())

}
```

其中，我们解析出来的 KvPair 列表，需要装入一个 HashMap，然后传给 HTTP client 的 JSON 方法。这样，我们的 HTTPie 的基本功能就完成了。

不过现在打印出来的数据对用户非常不友好，我们需要进一步用不同的颜色打印 HTTP header 和 HTTP body，就像 Python 版本的 HTTPie 那样，这部分代码比较简单，我们就不详细介绍了。

最后，来看完整的代码：

```
use anyhow::{anyhow, Result};

use clap::{AppSettings, Clap};

use colored::*;

use mime::Mime;

use reqwest::{header, Client, Response, Url};

use std::{collections::HashMap, str::FromStr};

// 以下部分用于处理 CLI
```

```
// 定义 HTTPie 的 CLI 的主入口，它包含若干个子命令
// 下面 /// 的注释是文档，clap 会将其作为 CLI 的帮助
/// A naive httpie implementation with Rust, can you imagine how easy it is?
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
struct Opts {
#[clap(subcommand)]
subcmd: SubCommand,
}
// 子命令分别对应不同的 HTTP 方法，目前只支持 get / post
#[derive(Clap, Debug)]
enum SubCommand {
Get(Get),
Post(Post),
// 我们暂且不支持其它 HTTP 方法
}
// get 子命令
/// feed get with an url and we will retrieve the response for you
#[derive(Clap, Debug)]
struct Get {
/// HTTP 请求的 URL
#[clap(parse(try_from_str = parse_url))]
url: String,
}
// post 子命令。需要输入一个 URL，和若干个可选的 key=value，用于提供 json body
/// feed post with an url and optional key=value pairs. We will post the data
/// as JSON, and retrieve the response for you
#[derive(Clap, Debug)]
```

```
struct Post {
/// HTTP 请求的 URL
#[clap(parse(try_from_str = parse_url))]
url: String,
/// HTTP 请求的 body
#[clap(parse(try_from_str=parse_kv_pair))]
body: Vec<KvPair>,
}
/// 命令行中的 key=value 可以通过 parse_kv_pair 解析成 KvPair 结构
#[derive(Debug, PartialEq)]
struct KvPair {
k: String,
v: String,
}
/// 当我们实现 FromStr trait 后，可以用 str.parse() 方法将字符串解析成 KvPair
impl FromStr for KvPair {
type Err = anyhow::Error;
fn from_str(s: &str) -> Result<Self, Self::Err> {
// 使用 = 进行 split，这会得到一个迭代器
let mut split = s.split("=");
let err = || anyhow!(format!("Failed to parse {}", s));
Ok(Self {
// 从迭代器中取第一个结果作为 key，迭代器返回 Some(T)/None
// 我们将其转换成 Ok(T)/Err(E)，然后用 ? 处理错误
k: (split.next().ok_or_else(err)?).to_string(),
// 从迭代器中取第二个结果作为 value
v: (split.next().ok_or_else(err)?).to_string(),
})
}
```

```
}
/// 因为我们为 KvPair 实现了 FromStr，这里可以直接 s.parse() 得到 KvPair
fn parse_kv_pair(s: &str) -> Result<KvPair> {
Ok(s.parse()?)
}
fn parse_url(s: &str) -> Result<String> {
// 这里我们仅仅检查一下 URL 是否合法
let _url: Url = s.parse()?;
Ok(s.into())
}
/// 处理 get 子命令
async fn get(client: Client, args: &Get) -> Result<()> {
let resp = client.get(&args.url).send().await?;
Ok(print_resp(resp).await?)
}
/// 处理 post 子命令
async fn post(client: Client, args: &Post) -> Result<()> {
let mut body = HashMap::new();
for pair in args.body.iter() {
body.insert(&pair.k, &pair.v);
}
let resp = client.post(&args.url).json(&body).send().await?;
Ok(print_resp(resp).await?)
}
// 打印服务器版本号 + 状态码
fn print_status(resp: &Response) {
let status = format!("{:?} {}", resp.version(), resp.status()).blue();
println!("{}\n", status);
}
```

```
// 打印服务器返回的 HTTP header
fn print_headers(resp: &Response) {
for (name, value) in resp.headers() {
println!("{}: {:?}", name.to_string().green(), value);
}
print!("\n");
}
/// 打印服务器返回的 HTTP body
fn print_body(m: Option<Mime>, body: &String) {
match m {
// 对于 "application/json" 我们 pretty print
Some(v) if v == mime::APPLICATION_JSON => {
println!("{}", jsonxf::pretty_print(body).unwrap().cyan())
}
// 其它 mime type，我们就直接输出
_ => println!("{}", body),
}
}
/// 打印整个响应
async fn print_resp(resp: Response) -> Result<()> {
print_status(&resp);
print_headers(&resp);
let mime = get_content_type(&resp);
let body = resp.text().await?;
print_body(mime, &body);
Ok(())
}
/// 将服务器返回的 content-type 解析成 Mime 类型
fn get_content_type(resp: &Response) -> Option<Mime> {
```

```
resp.headers()
.get(header::CONTENT_TYPE)
.map(|v| v.to_str().unwrap().parse().unwrap())
}
/// 程序的入口函数，因为在 HTTP 请求时我们使用了异步处理，所以这里引入 tokio
#[tokio::main]
async fn main() -> Result<()> {
let opts: Opts = Opts::parse();
let mut headers = header::HeaderMap::new();
// 为我们的 HTTP 客户端添加一些缺省的 HTTP 头
headers.insert("X-POWERED-BY", "Rust".parse()?);
headers.insert(header::USER_AGENT, "Rust Httpie".parse()?);
let client = reqwest::Client::builder()
.default_headers(headers)
.build()?;
let result = match opts.subcmd {
SubCommand::Get(ref args) => get(client, args).await?,
SubCommand::Post(ref args) => post(client, args).await?,
};
Ok(result)
}
// 仅在 cargo test 时才编译
#[cfg(test)]
mod tests {
use super::*;
#[test]
fn parse_url_works() {
assert!(parse_url("abc").is_err());
assert!(parse_url("http://abc.xyz").is_ok());
```

```
assert!(parse_url("https://httpbin.org/post").is_ok());
}
#[test]
fn parse_kv_pair_works() {
assert!(parse_kv_pair("a").is_err());
assert_eq!(
parse_kv_pair("a=1").unwrap(),
KvPair {
k: "a".into(),
v: "1".into()
}
);
assert_eq!(
parse_kv_pair("b=").unwrap(),
KvPair {
k: "b".into(),
v: "".into()
}
);
}
}
```

在这个完整代码的最后，我还撰写了几个单元测试，你可以用 cargo test 运行。Rust 支持条件编译，这里 #[cfg(test)] 表明整个 mod tests 都只在 cargo test 时才编译。

使用代码行数统计工具 tokei 可以看到，我们总共使用了 139 行代码，就实现了这个功能，其中还包含了约 30 行的单元测试代码：

```
❯ tokei src/main.rs
-------------------------------------------------------------------------------
Language Files Lines Code Comments Blanks
-------------------------------------------------------------------------------
Rust 1 200 139 33 28
```

Total 1 200 139 33 28

你可以使用 cargo build --release，编译出 release 版本，并将其拷贝到某个在 $PATH下的目录，然后体验一下：

到这里一个带有完整帮助的 HTTPie 就可以投入使用了。

我们测试一下效果：

这和官方的 HTTPie 效果几乎一样。今天的源代码可以在这里找到.

哈，这个例子我们大获成功。我们只用了 100 行代码出头，就实现了 HTTPie 的核心功能，远低于预期的 200 行。不知道你能否从中隐约感受到 Rust 解决实际问题的能力，以今天实现的 HTTPie 为例，

要把命令行解析成数据结构，我们只需要在数据结构上，添加一些简单的标注就能搞定。

数据的验证，又可以由单独的、和主流程没有任何耦合关系的函数完成。

作为 CLI 解析库，clap 的整体体验和 Python 的 click 非常类似，但比 Golang 的 cobra 要更简单。
这就是 Rust 语言的能力体现，明明是面向系统级开发，却能够做出类似 Python 的抽象和体验，所以一旦你适应了 Rust ，用起来就会感觉非常美妙。

小结

现在你应该有点明白，为什么我会在开篇词中会说，Rust 拥有强大的表现力。

或许你还是有点疑惑，这么学，我也太懵了，跟盲人摸象似的。其实初学者都会以为，必须要先搞明白所有的语法知识，才能动手写代码，不是的。

我们这周写三个实用例子的挑战，就是为了让你，在懵懂地撰写代码的过程中，直观感受 Rust 处理问题、解决问题的方式，同时可以跟你熟悉的语言去类比，无论是 Golang / Java，还是 Python / JavaScript，如果我用自己熟悉的语言怎么解决、Rust 给了我什么样的支持、我感觉它还缺什么。

在这个过程中，你脑子里会产生各种深度的思考，这些思考又必然会引发越来越多的问号，这是好事，带着这些问号，在未来的课程中才能更有目的地学习，也一定会学得深刻而有效。

今天的小挑战并不太难，你可能还意犹未尽。别急，下一讲我们会再写个难度大一点的、工作中都会用到的 Web 服务，继续体验 Rust 的魅力。

思考题

我们只是实现了 HTTP header 和 body 的高亮区分，但是 HTTP body 还是有些不太美观，可以进一步做语法高亮，如果你完成了今天的代码，觉得自己学有余力可以再挑战一下，你不妨试一试用 syntect 继续完善我们的 HTTPie。syntect 是 Rust 的一个语法高亮库，非常强大。
欢迎在留言区分享你的思考。你的 Rust 学习第四次打卡成功，我们下一讲见！

58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(49)

- 
Tyr

置顶
这堂课的源代码可以在这里找到：https://github.com/tyrchen/geektime-rust/tree/master/04_httpie

作者回复: 有同学反应不能编译通过，问题出在 clap 上。请升级到 Rust 1.54 重新编译。

另外，reqwest 默认的 TLS 使用的是系统的 openssl，对于某些 linux 用户如果没有安装好 openssl，可能会导致编译不过，对此，你可以修改 Cargo.toml:

```toml
reqwest = { version = "0.11", default-features = false, features = ["json", "rustls-tls"] } # HTTP 客户端
```

让 reqwest 使用 rustls。

2021-08-30

3

13

- 

Custer 

置顶

```
/// 打印服务器返回的 HTTP body
fn print_body(m: Option<Mime>, body: &String) {
    match m {
        // 对于 "application/json" 我们 pretty print
        Some(v) if v == mime::APPLICATION_JSON => {
            // println!("{}", jsonxf::pretty_print(body).unwrap().cyan())
            print_syntect(body);
```

```
        }
        // 其他 mime type，我们就直接输出
        _ => println!("{}", body),
    }
}

fn print_syntect(s: &str) {
    // Load these once at the start of your program
    let ps = SyntaxSet::load_defaults_newlines();
    let ts = ThemeSet::load_defaults();
    let syntax = ps.find_syntax_by_extension("json").unwrap();
    let mut h = HighlightLines::new(syntax, &ts.themes["base16-ocean.dark"]);
    for line in LinesWithEndings::from(s) {
        let ranges: Vec<(Style, &str)> = h.highlight(line, &ps);
        let escaped = as_24_bit_terminal_escaped(&ranges[..], true);
        println!("{}", escaped);
    }
}
```

作者回复: 非常棒！你似乎是第一个贴出来思考题答案的！我也更新了一下代码库，和你的代码基本一样你可以对比一下。两个点：1. print_syntect 可以再加一个参数 ext，这样灵活性更高；2. 打印时用 print! 效果更好一些。

2021-08-31

7

- 

王槐铤

置顶

环境
cargo --version
cargo 1.52.0 (69767412a 2021-04-21)

rustc --version
rustc 1.52.1 (9bc8c42bb 2021-05-09)

编译程序代码 clap 库部分报

```
8 | #![doc = include_str!("../README.md")]
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

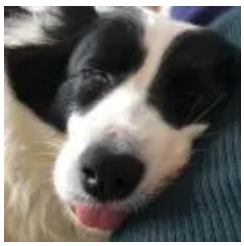
把 Cargo.toml 里 clap 依赖
clap = "=3.0.0-beta.4"
改为
clap = "=3.0.0-beta.2"
clap_derive = "=3.0.0-beta.2"
即可通过

具体原因 详见 https://github.com/dfinity/agent-rs/pull/260
作者回复: 👍

2021-08-30

1

4

- 
qinsi

习惯了 npm install 的可以试试 cargo-edit:

$ cargo install cargo-edit
$ cargo add anyhow colored jsonxf mime
$ cargo add clap --allow-prerelease
$ cargo add reqwest --features json
$ cargo add tokio --features full
作者回复: 好建议！

2021-08-30

28

- 
逸风

喜欢这样的教学方式！

作者回复: 谢谢！

2021-08-30

9

- [Rn]7s²

  clap更新到3.0.0-beta.5后，main.rs里需要改用use clap::{AppSettings, Parser};
  同时AppSettings里不再有ColoredHelp了，我暂时用[clap(setting = AppSettings::HelpRequired)]代替了。

  作者回复: 嗯，更新的代码参考：https://github.com/tyrchen/geektime-rust/tree/master/04_httpie

  2021-10-31

  6

- Marvichov
查了下colorize trait的doc (https://docs.rs/colored/2.0.0/colored/trait.Colorize.html), 没看到这个trait impl for String啊, 为啥可以call blue on String type呢?
```
format!("{:?} {}", resp.version(), resp.status()).blue();
```
老师知道这里面发生了什么转换么?

作者回复: 注意看 Colorize trait 的定义，它的方法 consume 的都是 self，而非 &self。所以当 impl Colorize for &str 时，self = &str。

在调用方法时，编译器会先看数据结构是否有对应的方法，如果有，按照方法的 signature，传 self / &self / &mut self。如果没有，再看引入的 trait 是否有对应的方法，必要时会根据 self 的类型做 auto Deref。所以这里编译器可以找到 blue()，因为它第一个参数 self = &str，String 可以 Deref 到 &str，所以可以调用。但如果 Colorize 的方法使用 &self，此时 &self = &&str，String 无法 Deref 到 &&str，所以编译器报错。

你可以看这个小例子：

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=dbd526df998fc8701ea9855c9e7de73c

```rust
pub trait Foo {
    fn foo(self);
}

pub trait Bar {
    fn bar(&self);
}

pub trait Baz {
    fn baz(&self);
}

impl Foo for &str {
    fn foo(self) {
        println!("Foo: {}", self);
    }
}

impl Bar for &str {
    fn bar(&self) {
        println!("Bar: {}", self);
    }
}

impl Baz for str {
    fn baz(&self) {
        println!("Baz: {}", self);
    }
}
```

```
fn main() {
  let s = String::from("Tyr");
  // foo 第一个参数是 self = &str，String 可以 auto Deref 到 &str，所以可以调用
  s.foo();
  // bar 第一个参数是 &self = &&str，String 无法 auto Deref 到 &&str
  // s.bar();
  // baz 第一个参数是 &self，但因为 impl Baz for str {}, 所以 &self = &str
  // 和 foo 类似，可以调用
  s.baz();
}
```

2021-09-07

1

6

- Arthur
  对Rust里的derive, impl, trait等概念，和Java/C++中面向对象编程概念里的封装、继承、方法等概念，有怎样的类比和不同，一直模糊不清，希望老师后面能讲到

  作者回复: 继承的概念，在 Rust 里是没有的。而且我们要避免使用这样的思维去建模。很多时候我们提到继承，其实并不是想用继承，而是想通过继承使用多态。在 Rust 下我们可以通过泛型，trait，trait object 实现面向对象语言中主要的多态手段。

  封装在任何编程语言中都或多或少存在，它本质上是一种对复杂性的控制。Rust 下 struct / enum / mod 都提供了封装的能力。把若干数据放在某个结构下，若干函数放在摸个模块下，只暴露该暴露的信息出去，这就是封装。

  方法可以理解为第一个参数为 self/this 这样的指向调用者自己的特殊函数，在 Python、Javascript 都有类似的概念。Rust 下可以为数据结构 impl 方法，或者 impl trait 来实现接口的方法。注意 Rust 下的方法可以消费 self（第一个参数是 self 而非 &self），其它大多数语言只能消费 &self。

  2021-09-05

  6

- 
Kerry

简洁的背后意味着大量的抽象。

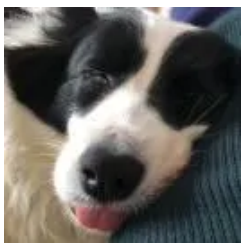
而初学者见到这么简洁的代码，会迷惑：”我复制了啥，怎么这么短就跑出这么多功能来？？？“

假以时日，就不禁感叹Rust语言表达力的强大。
作者回复: 👍是啊，这么底层的语言可以做出来这么强大的抽象。

2021-08-31

1

5

- 
qinsi

疑问：查了下reqwest似乎是依赖tokio运行时的，是否意味着用了reqwest就必须用tokio而不能用其他的运行时比如async-std?

作者回复: 如果想用 async-std，可以用 surf。我建议不要在 async-std 下花时间了。tokio 生态已经占据了绝对优势。

2021-08-30

4

- tider

  rust 1.56 按教程的代码编译不通过

  将Cargo.toml改了下就可以了

  [package]
  name = "httpie"
  version = "0.1.0"
  edition = "2021"

  # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

  [dependencies]
  anyhow = "1" # 错误处理
  clap = "=3.0.0-beta.4" #命令行解释
  clap_derive = "=3.0.0-beta.4"
  colored = "2" # 命令终端多彩显示
  jsonxf = "1.1" # JSON pretty print 格式化
  mime = "0.3" # 处理 mime 类型
  reqwest = { verson = "0.11", features = ["json"]} # HTTP 客户端
  tokio = { version = "1", features = ["full"]} # 异步处理库
  作者回复: 嗯，最新代码参考：https://github.com/tyrchen/geektime-rust/tree/master/04_httpie

  2021-10-31

  1

  3

- chinandy

  大家参考老师写的代码的时候，如果是网页不能正常的美化显示，有可能是网页是UTF8的原因，print_body函数match分支加一个TEXT_HTML_UTF_8判断， Some(v) if v == mime::TEXT_HTML_UTF_8 || v == mime::TEXT_HTML => print_syntect(body, "html"),即可。

  作者回复: 👍

  2021-09-18

  3

- gzgywh

  Rust里面的宏感觉跟Python里面的装饰器作用差不多嘛？

  作者回复: 宏可以操作并修改代码的 AST，装饰器只能对代码简单包装，无法改动 AST。

  2021-09-12

  3

- 罗杰

  ```
  fn print_body(m: Option<Mime>, body: &String) {
    match m {
      Some(v) if v.to_string().contains("application/json") => {
        println!("{}", jsonxf::pretty_print(body).unwrap().cyan())
      }
      _ => println!("{}", body),
    }
  }
  ```

  优化了一下程序。

  作者回复: 👍

  2021-09-03

  3

- CR

  很想知道老师的画图工具都有哪些🤔

  编辑回复: excalidraw

  2021-09-03

  3

- 阿海

  这一期小程序挺好的。上上周刚看完rust官网上的非官方文档，基本能看懂代码，但是要自己写出来就难了。期待下一期

  作者回复: 嗯，这需要一定时间的学习和历练。

  2021-08-30

  3

- Zilr

  陈老师， 为啥我用win10 powershell 和 cmd 运行的话， 都是乱码?

  作者回复: colored 应该可以在 windows 下的 powershell 工作。你可以试试在 main 函数最开始加一句：

  control::set_virtual_terminal(false).unwrap();

  来源：
  https://github.com/mackwic/colored/blob/015f2a58bfac8da6f0db8518c6813c79ec063f7f/src/control.rs#L7

  2021-09-19

  2

- 
  徐洲更

  这节课使用了clap进行命令行解析，我在看clap 的GitHub文档的时候，发现这种derive macros的参数写法应该是3.0.0版本后引进的，相关资料只看到一个exmaple。想知道老师是如何指导clap那么多用法的呢?

  作者回复: clap 合并了 structopt （https://docs.rs/structopt）的思路，我之前熟悉 structopt，所以了解。另外，看 clap 3.0 的文档需要访问：https://docs.rs/clap/3.0.0-beta.4/clap/index.html。注意由于它还是 beta，所以很多链接一不小心会跳到 2.33 的文档。

  2021-09-17

  1

  2

- soichir

  通过这节课介绍初识rust，对比c++感觉有如下区别:
  1. 看起来rust程序编译依赖网络，会编实时编译依赖的库，相应的引入三方包非常方便； 而不像c++要到三方库的网页上去找编译和引用的方式，要适配到自己的平台上。
  2. 通过rust、宏和trait等机制把非核心的诸如校验等逻辑隔离在各自的函数中, 大幅度降低了耦合。

  所以理论上rust可以写出更加简洁的代码吗?

  作者回复: 嗯，现代编程语言的依赖基本上都会从网络上直接拿。python，golang，nodejs 都是如此。不过 Rust 的 cargo 是用起来最方便的（毕竟有后发优势）。

  C++ 也有 template / interface 等类似的功能。从这个角度讲，大家旗鼓相当。不过 Rust 标准库和整个生态花了大力气做零成本抽象，把一些复杂的表述用编译器优雅地抽象掉了，比如 async/await 的实现，再比如 Rust 里大量的函数式编程方法。此外，Rust 还可以通过功能强大的过程宏，来自动完成很多复杂 trait 的定义，所以，从抽象的角度，Rust 更强大，在几乎不牺牲性能的情况下，可以写出更简洁的代码。

  2021-09-07

  3

- Honey拯救世界

  另外一种运行工程的命令是，cargo run -- post httpbin.org/post a=1 b=2
  -- 就相当于 target/debug/httpie

  作者回复: 对

  2021-09-05

  2

收起评论