Rust 之 hyper (一): client API

西瓜在笑

hyper 是 Rust 实现的 HTTP 库。hyper 同时支持 HTTP/1 和 HTTP/2，并且同时提供 client 与 server API。

hyper 性能好，偏底层，而且面向 async 设计，应用广泛，已成为 Rust 网络程序生态的重要基石之一。 知名的 HTTP client reqwest, HTTP server warp 和 axum, Rust 的 gRPC 实现 tonic 等，都使用了 hyper。我们不一定会直接使用 hyper，但了解 hyper 对于我们了解 Rust 的网络程序生态，学习设计良好的网络程序，都有好处。

闲扯一句，说到 Rust 网络程序的基石，我觉得最重要的是 tower。 tower 的目标是提供模块化、可复用的网络程序组件，它着眼于「request/response 模型」及「协议无关性」，以 `Service` 与 `Layer` 两个 trait 为抽象原语，一步步构建出网络编程中几乎所有重要的组件，这是让人叹为观止的。 hyper/tonic/axum 等正是以 tower 为中心，才组成了完整的生态。

client API

首先， `Cargo.toml` 中添加依赖:

```
[dependencies]
anyhow = "1"
hyper = { version = "0.14", features = ["full"]}
tokio = { version = "1", features = ["full"]}
```

然后是代码:

```
use hyper::{Client, Uri};
use std::str;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let client = Client::new();
    let uri = Uri::from_static("http://httpbin.org/ip"); // panic if not valid
    let mut res = client.get(uri).await?;

    println!("status code: {}", res.status());
    for (key, value) in res.headers().iter() {
        println!("{}: {}", key, value.to_str().unwrap())
    }

    let body = res.body_mut();
    let buf = hyper::body::to_bytes(body).await?;
    let content = str::from_utf8(buf.as_ref())?;
    println!("{}", content);

    Ok(())
}
```

这个示例比较简单，但从中也能感受 hyper 的 low-level 特性。

使用 hyper 时，需要手动操作 Request, Response 及相关类型。 示例中使用 `get` 方法直接发送了 GET 请求，这只是特例。对于其他请求，都需要手动构造 Request 对象，并调用 `Client::request` 方法发送。

而如果要在 Request 的 body 中添加内容，或者从 Response 的 body 中读取内容，也很麻烦。示例使用了辅助函数 `to_bytes` 读取 body 的内容并返回 `Bytes` 对象。 在 hyper 中操作 body 需要通过 `HttpBody` trait(它其实是来自于 `http` crate 的 `Body` trait，hyper 通过 `pub use` 的方式换了个名字)。

而至于处理 cookie 的功能，就完全不在 hyper 的范围之内了。对于 hyper 来说，它只是 `cookie` 与 `set-cookie` 两个 header 而已。

另外，hyper 默认不支持 https。访问 https 资源需要使用 hyper-tls 的 `HttpsConnector` 。 这又涉及 hyper 的 `Connect` trait 了，实现了 `Connect` trait 的 struct 被称为 connector。 hyper 的 Client 默认使用 `HttpConnector` ，如果要使用其他 connector，则需要使用 Builder 创建 Client。 Builder 的 `build` 方法，接受 connector 作为参数，以设置 client 所使用的 connector。

接下来，我们展示一个更复杂的示例，它会使用更多的 hyper API。

client API II: advanced

先看看 Cargo.toml:

```
[package]
name = "hyperdemo"
version = "0.1.0"
edition = "2021"

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

[dependencies]
anyhow = "1"
base64 = "0.13"
bytes = "1"
http = "0.2"
hyper = { version = "0.14", features = ["full"] }
hyper-tls = "0.5"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["full"] }
```

接下来是 Rust 代码:

```
use bytes::Bytes;
use http::{HeaderMap, Method};
use hyper::body::{to_bytes, HttpBody};
use hyper::Client;
use hyper_tls::HttpsConnector;
```

```rust
use serde::{Deserialize, Serialize};
use serde_json::json;
use std::collections::HashMap;
use std::io::Write;
use std::pin::Pin;
use std::task::{Context, Poll};

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // token, 可在 https://github.com/settings/tokens 创建 Personal access token
    let (user, token) = ("<username>", "<token>");
    let gist = create_gist(user, token).await?;
    println!("created gist: {}", gist.url);
    Ok(())
}

// <https://docs.github.com/en/rest/gists/gists#create-a-gist>
async fn create_gist(user: &str, token: &str) -> anyhow::Result<Gist> {
    let client = Client::builder().build(HttpsConnector::new());

    let gist_param = CreateGistParam {
        description: "gist created via hyper".to_string(),
        public: true,
        files: vec![
            GistFile {
                name: "README.md".to_string(),
                content: README.to_string(),
            },
            GistFile {
                name: "main.rs".to_string(),
                content: CODE.to_string(),
            },
        ],
    };

    let mut header_value = b"Basic ".to_vec();
    {
        // create new scope to make borrow check happy
        // header 格式: authorization: Basic {user} {password}
        let mut encoder = base64::write::EncoderWriter::new(&mut header_value,
        write!(encoder, "{}:{}", user, token).unwrap();
    }

    let req = hyper::Request::builder()
        .method(Method::POST)
        .uri("https://api.github.com/gists")
        .header("User-Agent", "hyper")
        .header("Accept", "application/vnd.github.v3+json")
        .header("Authorization", header_value)
        .body(CreateGistBody::new(gist_param))?;

    let mut res = client.request(req).await?;
    if !res.status().is_success() {
        return Err(anyhow::format_err!("{}", res.status()));
    }
```

```
    let buf = to_bytes(res.body_mut()).await?;
    let gist: Gist = serde_json::from_slice(&buf)?;
    Ok(gist)
}

struct CreateGistParam {
    pub description: String,
    pub public: bool,
    pub files: Vec<GistFile>,
}

struct GistFile {
    pub name: String,
    pub content: String,
}

struct CreateGistBody(Option<CreateGistParam>);

impl CreateGistBody {
    fn new(param: CreateGistParam) -> CreateGistBody {
        CreateGistBody(Some(param))
    }
}

impl HttpBody for CreateGistBody {
    type Data = Bytes;
    type Error = anyhow::Error;

    fn poll_data(
        mut self: Pin<&mut Self>,
        _cx: &mut Context<'_>,
    ) -> Poll<Option<Result<Bytes, Self::Error>>> {
        match self.0.take() {
            Some(param) => {
                // github 创建 gitst 的 REST API，参数格式特别，不可直接将 CreateGis
                let mut file_map = HashMap::new();
                for file in &param.files {
                    let mut inner_map = HashMap::new();
                    inner_map.insert("content", &file.content);
                    file_map.insert(&file.name, inner_map);
                }

                let gist_body = json!({
                    "description": param.description,
                    "public": param.public,
                    "files": file_map,
                });

                match serde_json::to_string(&gist_body) {
                    Ok(s) => Poll::Ready(Some(Ok(Bytes::from(s)))),
                    Err(e) => return Poll::Ready(Some(Err(e.into()))),
                }
            }
            None => Poll::Ready(None),
        }
    }
```

```
    fn poll_trailers(
        self: Pin<&mut Self>,
        _cx: &mut Context<'_>,
    ) -> Poll<Result<Option<HeaderMap>, Self::Error>> {
        Poll::Ready(Ok(None))
    }
}

#[derive(Serialize, Deserialize)]
struct Gist {
    #[serde(rename = "html_url")]
    url: String,
}

const README: &str = r#"hyper
=====

hyper is awesome!
"#;

const CODE: &str = r#"pub fn main() {
    println!("hello hyper!");
}
"#;
```

这个示例通过 hyper 调用 github API 创建 gist。它更接近「访问 REST API 并将返回的 JSON 内容反序列化为对象」这种偏业务的场景。

这个示例比较全面的展示了 hyper 各个 API 的使用，包括 connector, request, response, header, status code 等等。而且展示了怎样手动实现 `HttpBody` trait(我们也可以直接使用 hyper 的 Body struct，从而避免手动实现 `HttpBody` trait)。

使用 hyper 进行业务处理，比较复杂，也很费代码。作为对比，我们也可以使用基于 hyper 的 high-level HTTP 客户端 reqwest 来实现同样的功能，示例代码见 Rust Cookbook，可见使用 reqwest 的话代码短小精悍了许多。

问题：hyper 既然如此底层，而且不好用，它存在的意义是什么呢？

答：hyper 实现了 HTTP/1.1 及 HTTP2 协议，这就是它最大的意义。另外，它可以作为更高层库的构建单元。

hyper 的 HTTP 相关依赖

hyper 依赖了如下与 http 有关的 crate:

- http，此 crate 定义了 HTTP 的常见类型，如 Request, Response 等。hyper 的 lib.rs 就包含对此 crate 的 pub use:

```
pub use crate::http::{header, Method, Request, Response, StatusCode, Uri, Versi
```

`http` crate 最大的意义也在于「提供统一的类型定义」，HTTP 协议相关的库都使用此 crate 提供的定义，那么它们之间的互操作性就大大提高了。

- http-body，一句话介绍: asynchronous HTTP request or response body。主要是定义了 `Body` trait。 `http` crate 的 Request 与 Response 类型，其 body 为泛型。正好与这里的 `Body` trait 可以配合起来。

值得注意的是，hyper `src/body/mod.rs` 模块中有一行 `pub use http_body::Body as HttpBody;` 。因此，hyper 中使用的 HttpBody trait，就是此 crate 的 `Body` 。而且 hyper 在 body 模块中定义了一个名为 `Body` 的 struct。不了解这一点，翻看代码时常常会觉得混乱。

- httpdate，与 HTTP 相关的日期与时间。只是提供了两个函数， `fmt_http_date` , `parse_http_date` 。不甚重要。
- httparse，一句话介绍: A push parser for the HTTP 1.x protocol. Avoids allocations. No copy. Fast. 此 crate 的作者 @seanmonstar 也是 hyper, reqwest, warp, num_cpus 的作者。
- h2 一句话介绍: A Tokio aware, HTTP/2 client & server implementation for Rust。

编辑于 2022-05-24 19:02

Rust（编程语言） HTTP

写下你的评论...

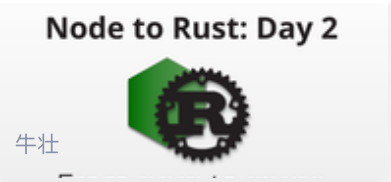
还没有评论，发表第一个评论吧

推荐阅读

Rust过程宏入门（三）——实现简易派生宏

在上一篇文章中介绍了如何手动实现Builder类，本文将介绍如何用

FGO 輝...　　发表于游戏的NE...



牛壮

生宏自动实现。
https://zhuanlan.zhihu.com/p/
再谈派生宏的原理简单起见，我

TsyunhKjit Yuang