课程重点：

- 掌握hyper的使用
- client、server的流程
- http api json格式
- client下载网页 chunk方式读取数据
- http api简单网关 简单转发
- **回顾之前学习的语法，比如生命周期、模板等。**

官方：https://hyper.rs/
https://docs.rs/hyper/0.14.18/hyper/

参考书籍：Rust 语言圣经 - Rust语言圣经(Rust Course)

1 如何使用

Cargo.toml添加库依赖

```
[dependencies]
hyper = "0.14"
```

依赖Tokio

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

hyper是一个**偏底层的http库**:

- 支持HTTP/1和HTTP/2
- 支持异步Rust
- 同时提供了服务端和客户端的API支持
- 性能卓越

hyper是一个相对底层的库，旨在为其他库和应用构建底层的HTTP模块。
如果你需要使用更为方便的HTTP客户端，可以考虑reqwest；如果需要更为方便的HTTP服务器，可以考虑使用warp。它们两者都是基于hyper构建。
reqwest： https://github.com/seanmonstar/reqwest
warp：https://github.com/seanmonstar/warp
作者博客：https://seanmonstar.com/

2 hyper基本使用

2.1 Hello World简单范例

以最简单的GET操作为例。

2.1.1 服务器端

ex2_1_server
首先是依赖，除了hyper本身之外，我们还需要tokio的runtime去执行async函数

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

然后就是main.rs

```
use std::{convert::Infallible, net::SocketAddr};
use hyper::{Body, Request, Response, Server};
use hyper::service::{make_service_fn, service_fn};

// 返回200
async fn handle(r: Request<Body>) -> Result<Response<Body>, Infallible> {
    println!("handle req:{:?}", r);
    Ok(Response::new("Hello, World!\n".into()))
}

#[tokio::main]
async fn main() {
    println!("1. SocketAddr::from");
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));

    // 从handle创建一个服务
    println!("2. make_service_fn");
    let make_svc = make_service_fn(|_conn| async {
        Ok::<_, Infallible>(service_fn(handle))
    });
    println!("3. Server::bind");
    let server = Server::bind(&addr).serve(make_svc);

    // 运行server
    println!("4. Server await");
    if let Err(e) = server.await {
        eprintln!("server error: {}", e);
    }
}
```

2.1.2 客户端

依赖同服务器端

```
use hyper::Client;
use hyper::body::HttpBody;
use tokio::io::{stdout, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {

    // 构建一个client，调用GET
    let client = Client::new();
    let uri = "http://127.0.0.1:3000".parse()?;
    let mut resp = client.get(uri).await?;
    println!("Response: {}", resp.status());

    // 将response（是个stream）输出到stdout
    while let Some(chunk) = resp.body_mut().data().await {
        stdout().write_all(&chunk?).await?;
    }

    Ok(())
}
```

先启动服务端，然后启动客户端，就可以看到服务端成功相应客户端的GET请求啦～
Response: 200 OK
Hello, World!

2.2 echo服务范例

下面我们通过实现一个echo服务主要看一下服务器端如何进行路由，以及如何支持POST请求

2.2.1 服务器端

ex2_2_server
依赖

```
[dependencies]
hyper = { version = "0.14", features = ["full"] }
tokio = { version = "1", features = ["full"] }
futures-util = { version = "0.3", default-features = false }
```

代码

```
use futures_util::TryStreamExt;
use hyper::service::{make_service_fn, service_fn};
use hyper::{Body, Method, Request, Response, Server, StatusCode};

async fn echo(req: Request<Body>) -> Result<Response<Body>, hyper::Error> {
    let mut response = Response::new(Body::empty());

    // 通过req.method()和req.uri().path()来识别方法和请求路径
    match (req.method(), req.uri().path()) {
        (&Method::GET, "/") => {
            *response.body_mut() = Body::from("Try POSTing data to /echo");
        },
        (&Method::POST, "/echo") => {
            // 将POST的内容保持不变返回
            *response.body_mut() = req.into_body();
        },
        (&Method::POST, "/echo/uppercase") => {
            // 把请求stream中的字母都变成大写，并返回 abcd -> ABCD
            let mapping = req
                .into_body()
                .map_ok(|chunk| {
                    chunk.iter()
                        .map(|byte| byte.to_ascii_uppercase())
                        .collect::<Vec<u8>>()
                });
            // 把stream变成body
            *response.body_mut() = Body::wrap_stream(mapping);
        },
        (&Method::POST, "/echo/reverse") => {
            // 这里需要完整的body，所以需要等待全部的stream并把它们变为bytes 1234 ->
4321
            let full_body = hyper::body::to_bytes(req.into_body()).await?;

            // 把body逆向
            let reversed = full_body.iter()
                .rev()
                .cloned()
                .collect::<Vec<u8>>();

            *response.body_mut() = reversed.into();
        },
        _ => {
            *response.status_mut() = StatusCode::NOT_FOUND;
        },
    };

    Ok(response)
}

#[tokio::main]
async fn main() {
    let addr = ([127, 0, 0, 1], 3000).into();

    let make_svc = make_service_fn(|_conn| async {
        Ok::<_, hyper::Error>(service_fn(echo))
    });

    let server = Server::bind(&addr).serve(make_svc);

    if let Err(e) = server.await {
        eprintln!("server error: {}", e);
    }
}
```

2.2.2 客户端

ex2_2_client
依赖和之前客户端一样。这里：

1. /echo/reverse提交内容为echo的POST请求；
2. /json提交内容json的POST请求；

```
use hyper::Client;
use hyper::{Body, Method, Request};
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct UserRegister<'a> {
    user: &'a str,
    pwd: &'a str,
}
```

```
#[derive(Serialize, Deserialize, Debug)]
struct UserRegisterResp<'a> {
    user: &'a str,
    status: &'a str,
}
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // let req = Request::builder()
    //     .method(Method::POST)
    //     .uri("http://127.0.0.1:3000/echo/reverse")
    //     .body(Body::from("echo"))?;

    // let client = Client::new();
    // let resp = client.request(req).await?;
    // println!("Response: {}", resp.status());
    // println!("{:?}", hyper::body::to_bytes(resp.into_body()).await.unwrap());

    let user = UserRegister { user: "darren", pwd: "123456"};
    let req = Request::builder()
        .method(Method::POST)
        .uri("http://127.0.0.1:3000/json")
        .body(Body::from(serde_json::to_string(&user).unwrap()))?; // 注意?的意义，
正常时返回对应的value

    let client = Client::new();
    let resp = client.request(req).await?;
    println!("Response: {}", resp.status());

    let full_body = hyper::body::to_bytes(resp.into_body()).await?; // 读取返回的数
据
    // 反序列化
    let userResp: UserRegisterResp =
serde_json::from_slice(&full_body).unwrap();

    println!("body:{:?}", full_body);  // 字节
    println!("body:{:?}", String::from_utf8_lossy(&full_body)); // 字符串
    println!("{:?}", userResp); // 结构体

    Ok(())
}
```

依次启动服务端和客户端，就可以看到服务端响应了客户端的POST。

2.2.3 涉及的json序列化和反序列化

在echo范例中涉及的json库和序列化/反序列化操作。

1. 引入的库

```
serde = { version = "1.0.117", features = ["derive"] }
serde_json = "1.0.59"
```

2. 序列化

```
#[derive(Serialize, Deserialize, Debug)]
struct UserRegisterResp<'a> {
    user: &'a str,
    status: &'a str,
}

let userResp = UserRegisterResp { user: "darren", status: "ok"};
let json_str = serde_json::to_string(&userResp).unwrap();
```

3. 反序列化

```
#[derive(Serialize, Deserialize, Debug)]
struct UserRegister<'a> {
    user: &'a str,
    pwd: &'a str,
}
//full_body为 [u8]
let user: UserRegister = serde_json::from_slice(&full_body).unwrap();
```

3 更多范例练习

ex3_1_client下载网页

依赖

```
hyper = { version = "0.14", features = ["full"] }
tokio = { version = "1", features = ["full"] }
log = "0.4"
pretty_env_logger = "0.4"
```

比如cargo build，然后运行：
运行 ./target/debug/ex3_1_client http://www.baidu.com

```
use std::env;        // 解析命令行
use hyper::{body::HttpBody as _, Client};
use tokio::io::{self, AsyncWriteExt as _};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send +
Sync>>;

#[tokio::main]
async fn main() -> Result<()> {
    // 日志初始化
    pretty_env_logger::init();

    // 从命令行参数解析url
    let url = match env::args().nth(1) {
        Some(url) => url,
        None => {
            println!("Usage: client <url>"); // 比如./target/debug/ex4_1_client
http://www.baidu.com
            return Ok(());
        }
    };
    //封装http url
    let url = url.parse::<hyper::Uri>().unwrap();  // 传入模板hyper::Uri
    if url.scheme_str() != Some("http") {
        println!("This example only works with 'http' URLs.");
        return Ok(());
    }
    fetch_url(url).await
}

async fn fetch_url(url : hyper::Uri) -> Result<()> {
    // 创建一个客户端
    let client = Client::new();
    // get请求并返回结果
    let mut res = client.get(url).await?;
    println!("Response: {}", res.status());
    println!("Headers: {:#?}\n", res.headers());

    // 读取返回结果，读取完毕再退出
    while let Some(next) = res.data().await {
         let chunk = next?;
         io::stdout().write_all(&chunk).await?;
    }
    println!("\n\nDone!");

    Ok(())
}
```

ex3_2_clientjson

依赖

```
[dependencies]
hyper = { version = "0.14", features = ["full"] }
tokio = { version = "1", features = ["full"] }
serde = { version = "1.0.117", features = ["derive"] }
serde_json = "1.0.59"
```

```
use hyper::body::Buf;
use hyper::Client;
use serde::{Serialize, Deserialize};
// A simple type alias so as to DRY.
type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send +
Sync>>;

#[tokio::main]
async fn main() -> Result<()> {
    let url = "http://jsonplaceholder.typicode.com/users".parse().unwrap();
    let users = fetch_json(url).await?;
    // print users
    println!("users: {:#?}", users);

    // print the sum of ids
    let sum = users.iter().fold(0, |acc, user| acc + user.id);
    println!("sum of ids: {}", sum);
    Ok(())
}
```

```
async fn fetch_json(url: hyper::Uri) -> Result<Vec<User>> {
    let client = Client::new();

    // Fetch the url...
    let res = client.get(url).await?;

    // asynchronously aggregate the chunks of the body
    let body = hyper::body::aggregate(res).await?;

    // try to parse as json with serde_json
    let users = serde_json::from_reader(body.reader())?;
    Ok(users)
}

#[derive(Deserialize, Debug)]
struct User {
    id: i32,
    name: String,
}
```

ex3_3_webapi api访问

依赖

```
[dependencies]
hyper = { version = "0.14", features = ["full"] }
tokio = { version = "1", features = ["full"] }
serde = { version = "1.0.117", features = ["derive"] }
serde_json = "1.0.59"
futures-util = { version = "0.3", default-features = false }
```

```
use hyper::body::Buf;
use futures_util::{stream, StreamExt};
use hyper::client::HttpConnector;
use hyper::service::{make_service_fn, service_fn};
use hyper::{header, Body, Client, Method, Request, Response, Server,
StatusCode};

type GenericError = Box<dyn std::error::Error + Send + Sync>;
type Result<T> = std::result::Result<T, GenericError>;

static INDEX: &[u8] = b"<a href=\"test.html\">test.html</a>";
static INTERNAL_SERVER_ERROR: &[u8] = b"Internal Server Error";
static NOTFOUND: &[u8] = b"Not Found";
static POST_DATA: &str = r#"{"original": "data"}"#;
static URL: &str = "http://127.0.0.1:1337/json_api";

async fn client_request_response(client: &Client<HttpConnector>) ->
Result<Response<Body>> {
    let req = Request::builder()
        .method(Method::POST)
        .uri(URL)
        .header(header::CONTENT_TYPE, "application/json")
        .body(POST_DATA.into())
        .unwrap();

    let web_res = client.request(req).await?;
    // Compare the JSON we sent (before) with what we received (after):
    let before = stream::once(async {
        Ok(format!(
            "<b>POST request body</b>: {}<br><b>Response</b>: ",
            POST_DATA,
        )
        .into())
    });
    let after = web_res.into_body();
    let body = Body::wrap_stream(before.chain(after));

    Ok(Response::new(body))
}

async fn api_post_response(req: Request<Body>) -> Result<Response<Body>> {
    // 读取所有body数据
    let whole_body = hyper::body::aggregate(req).await?;
    // 范经理和
    let mut data: serde_json::Value =
serde_json::from_reader(whole_body.reader())?;
    // Change the JSON...
    data["test"] = serde_json::Value::from("test_value");
    // And respond with the new JSON.
    let json = serde_json::to_string(&data)?;
    let response = Response::builder()
        .status(StatusCode::OK)
        .header(header::CONTENT_TYPE, "application/json")
        .body(Body::from(json))?;
    Ok(response)
}

async fn api_get_response() -> Result<Response<Body>> {
    let data = vec!["foo", "bar"];
    let res = match serde_json::to_string(&data) {
        Ok(json) => Response::builder()
            .header(header::CONTENT_TYPE, "application/json")
            .body(Body::from(json))
            .unwrap(),
        Err(_) => Response::builder()
            .status(StatusCode::INTERNAL_SERVER_ERROR)
            .body(INTERNAL_SERVER_ERROR.into())
            .unwrap(),
    };
    Ok(res)
}

async fn response_examples(
    req: Request<Body>,
    client: Client<HttpConnector>,
) -> Result<Response<Body>> {
    match (req.method(), req.uri().path()) {
        (&Method::GET, "/") | (&Method::GET, "/index.html") =>
Ok(Response::new(INDEX.into())),
        (&Method::GET, "/test.html") => client_request_response(&client).await,
        (&Method::POST, "/json_api") => api_post_response(req).await,
        (&Method::GET, "/json_api") => api_get_response().await,
        _ => {
            // Return 404 not found response.
            Ok(Response::builder()
                .status(StatusCode::NOT_FOUND)
                .body(NOTFOUND.into())
                .unwrap())
        }
    }
}

#[tokio::main]
async fn main() -> Result<()> {
    let addr = "0.0.0.0:1337".parse().unwrap();

    // Share a `Client` with all `Service`s
    let client = Client::new();

    let new_service = make_service_fn(move |_| {
        // Move a clone of `client` into the `service_fn`.
        let client = client.clone();
        async {
            Ok::<_, GenericError>(service_fn(move |req| {
                // Clone again to ensure that client outlives this closure.
                response_examples(req, client.to_owned())
            }))
        }
    });

    let server = Server::bind(&addr).serve(new_service);

    println!("Listening on http://{}", addr);

    server.await?;

    Ok(())
}
```

ex3_4_gateway简单网关

```
use hyper::service::{make_service_fn, service_fn};
use hyper::{Client, Error, Server};
use std::net::SocketAddr;

#[tokio::main]
async fn main() {
    let in_addr = ([0, 0, 0, 0], 3001).into();
    let out_addr: SocketAddr = ([127, 0, 0, 1], 3000).into();

    let client_main = Client::new();

    let out_addr_clone = out_addr.clone();

    // The closure inside `make_service_fn` is run for each connection,
    // creating a 'service' to handle requests for that specific connection.
    let make_service = make_service_fn(move |_| {
        let client = client_main.clone();
        async move {
            // This is the `Service` that will handle the connection.
            // `service_fn` is a helper to convert a function that
            // returns a Response into a `Service`.
            Ok::<_, Error>(service_fn(move |mut req| {
                let uri_string = format!(
                    "http://{}{}",
                    out_addr_clone,
                    req.uri()
                        .path_and_query()
                        .map(|x| x.as_str())
                        .unwrap_or("/")
                );
                let uri = uri_string.parse().unwrap();
                *req.uri_mut() = uri;
                client.request(req)
            }))
        }
    });

    let server = Server::bind(&in_addr).serve(make_service);

    println!("Listening on http://{}", in_addr);
    println!("Proxying on http://{}", out_addr);

    if let Err(e) = server.await {
        eprintln!("server error: {}", e);
    }
}
```

#

4 性能测试方式

使用wrk测试http服务器性能，比如
wrk -t16 -c1000 -d120s http://127.0.0.1:8080/

4.1 安装教程

1. 先安装必要的依赖
   sudo apt install build-essential libssl-dev git
2. 从 github 拉取源代码
   git clone https://github.com/wg/wrk.git
3. 也可以使用 gitee 的镜像
   git clone https://gitee.com/mirrors/wrk.git
4. 使用 make 编译源代码
   cd wrk
   make
5. 把生成的wrk移到一个PATH目录下面, 比如
   sudo cp wrk /usr/local/bin

4.2 Options参数

```
-c, --connections <N> 跟服务器建立并保持的TCP连接数量
-d, --duration <T> 压测时间
-t, --threads <N> 使用多少个线程进行压测，压测时，是有一个主线程来控制我们设置的n个子线程间调
度
-s, --script      <S>   指定Lua脚本路径
-H, --header       <H>   为每一个HTTP请求添加HTTP头
    --latency            在压测结束后，打印延迟统计信息
    --timeout      <T>   超时时间
-v, --version            打印正在使用的wrk的详细版本信
```

代表数字参数，支持国际单位 (1k, 1M, 1G)
代表时间参数，支持时间单位 (2s, 2m, 2h)

4.3 测试范例

最简单的压力测试示例，get请求，固定url固定参数

./wrk -t10 -c30 -d 2s -T5s --latency http://www.baidu.com

./wrk -t4 -c4 -d 2s -T5s --latency http://www.baidu.com

参数释义：
-t：需要模拟的线程数
-c：需要模拟的连接数
-d：测试的持续时间
----timeout 或 -T：超时的时间
–latency：显示延迟统计
-s 或 --script: lua脚本,使用方法往下看
-H, --header: 添加http header, 比如. “User-Agent: wrk”

5 参考

用Rust做Web开发，时机成熟了 https://rustcc.cn/article?id=0e8e1b38-5180-4021-b6fe-e017eb8ff315