CloudWeGo

Rust HTTP 框架

以 axum 为例

刘强

2023/09/12

CONTENT

目录

01 HTTP 协议

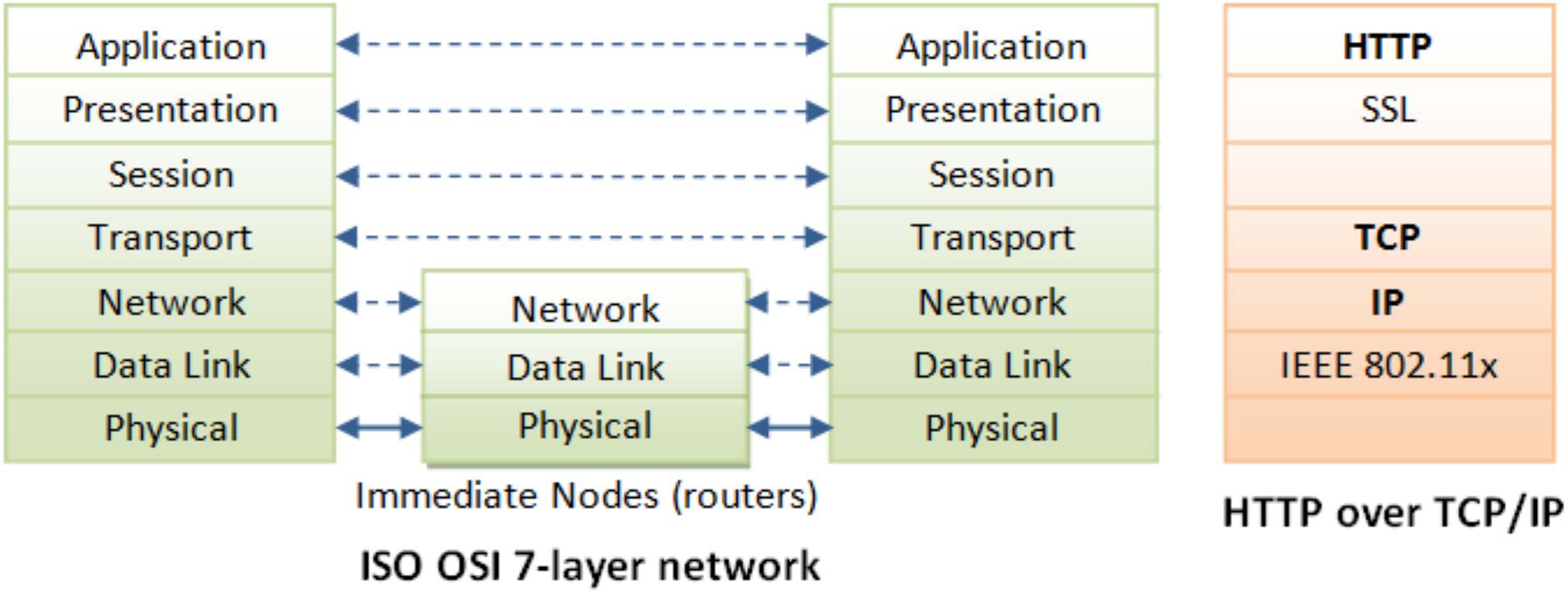
介绍HTTP协议

什么是 HTTP

在浏览器中输入URL会发生什么

什么是 HTTP

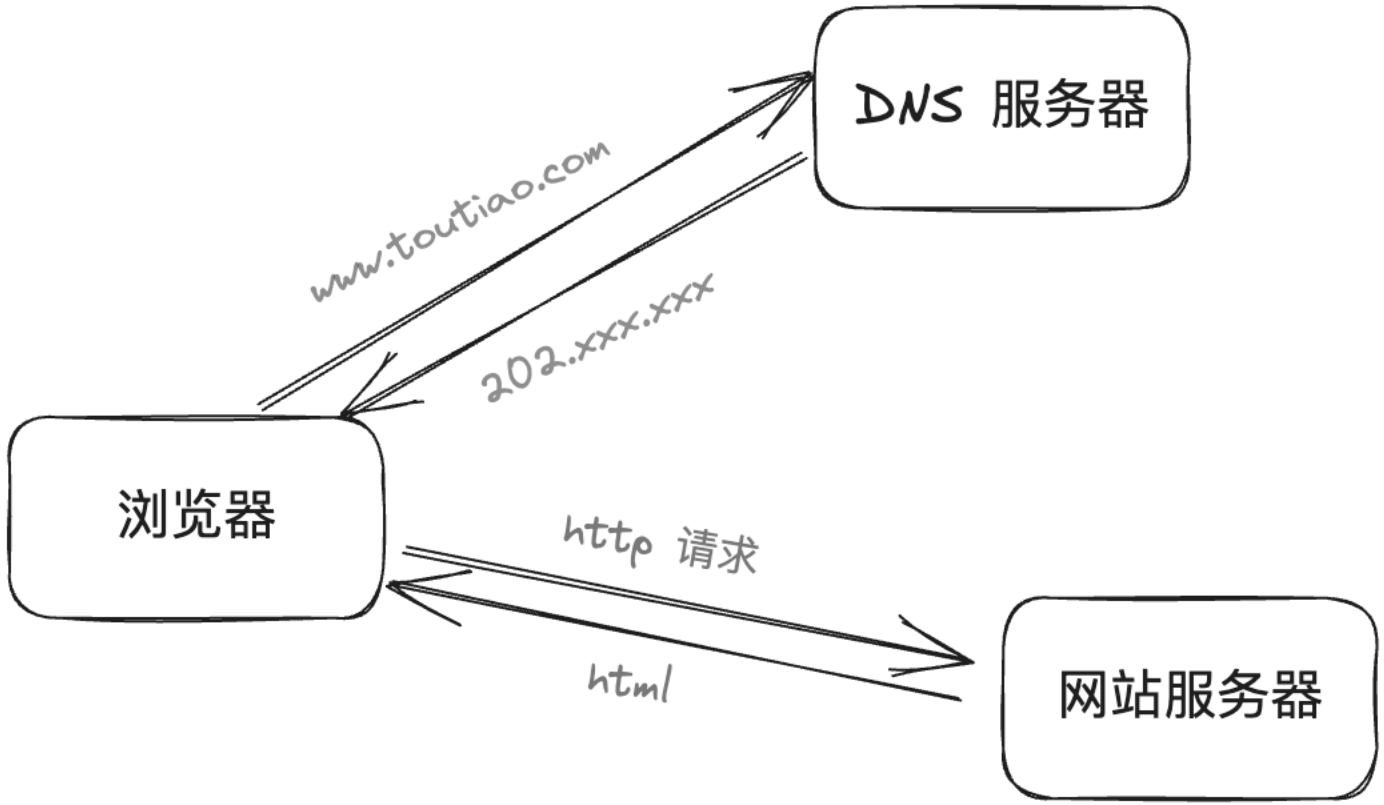
HTTP 位于应用层



Source: https://www.linkedin.com/pulse/when-you-type-any-url-your-browser-press-enter-andrew-godwin

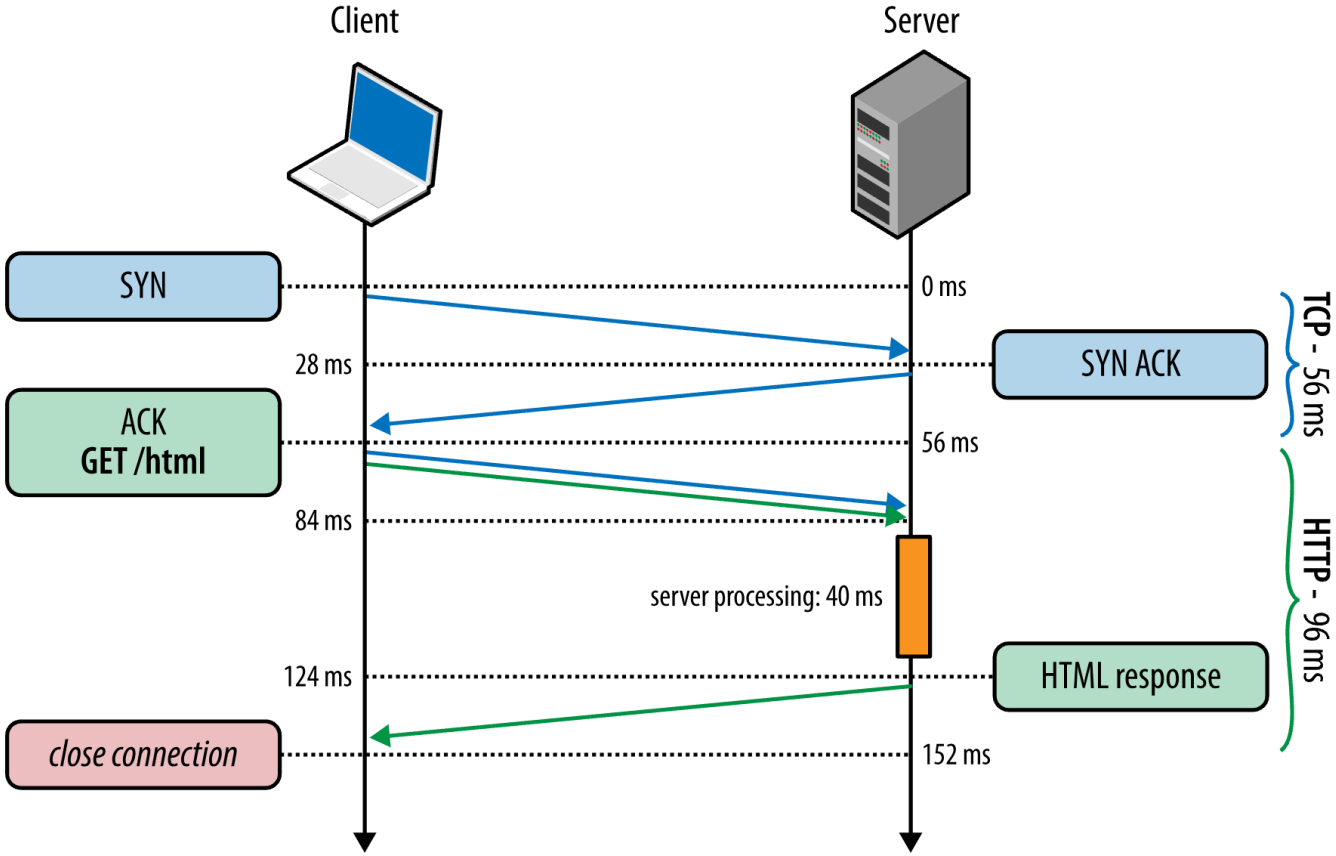
什么是 HTTP

第一步: 浏览器解析域名



什么是 HTTP

第二步: 和服务器建立连接

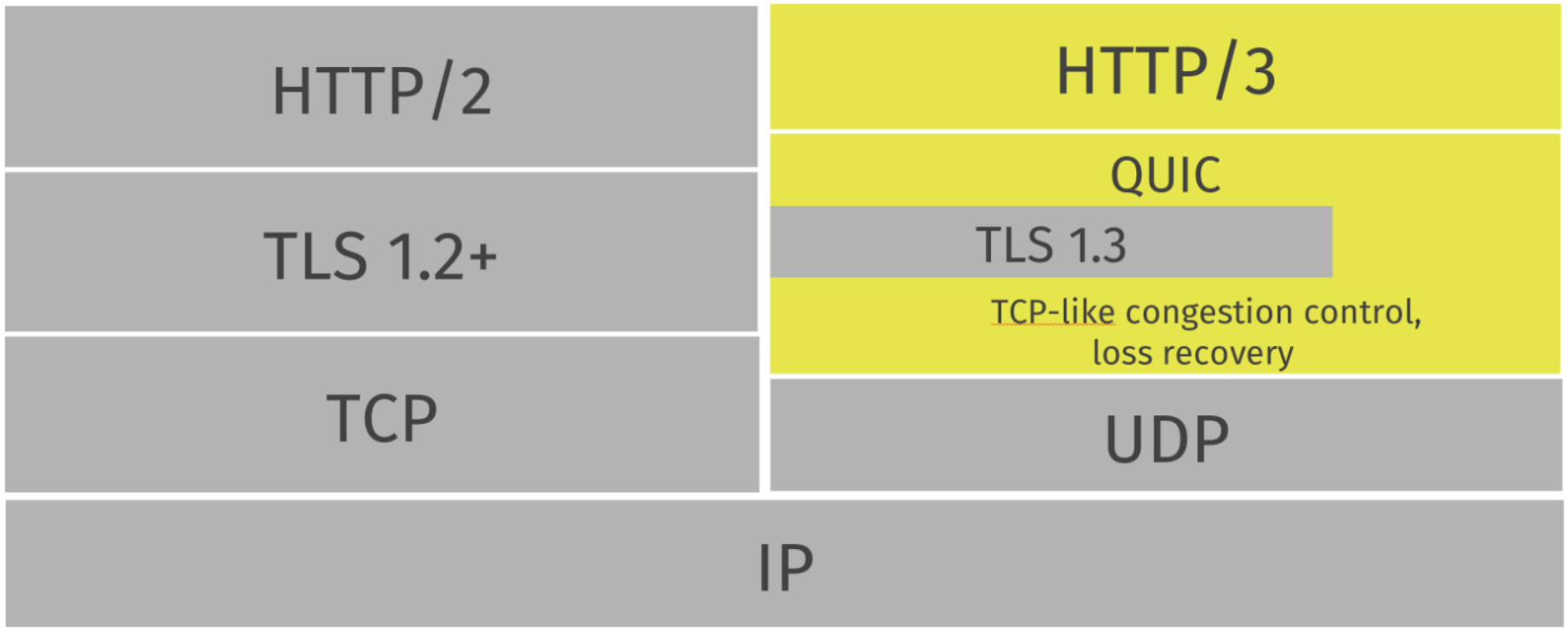


Source: https://hpbn.co/http1x/

什么是 HTTP

第二步: 和服务器建立连接

连接层可以使用不同的协议，例如 TCP、UDP

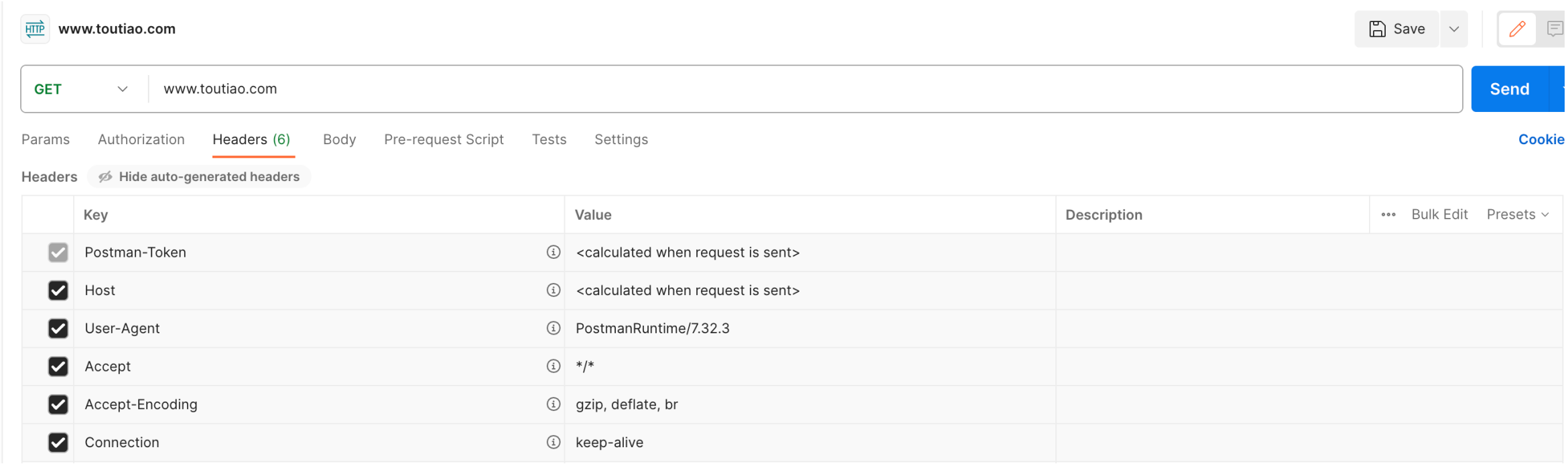


Source: https://http3-explained.haxx.se/zh-tw/the-protocol

什么是 HTTP

第三步: **浏览器发起 HTTP 请求**

使用 Postman 工具 模拟 一个 HTTP 请求



什么是 HTTP

第四步: 服务端处理**HTTP 请求并返回**

什么是 HTTP

第五步: 浏览器渲染

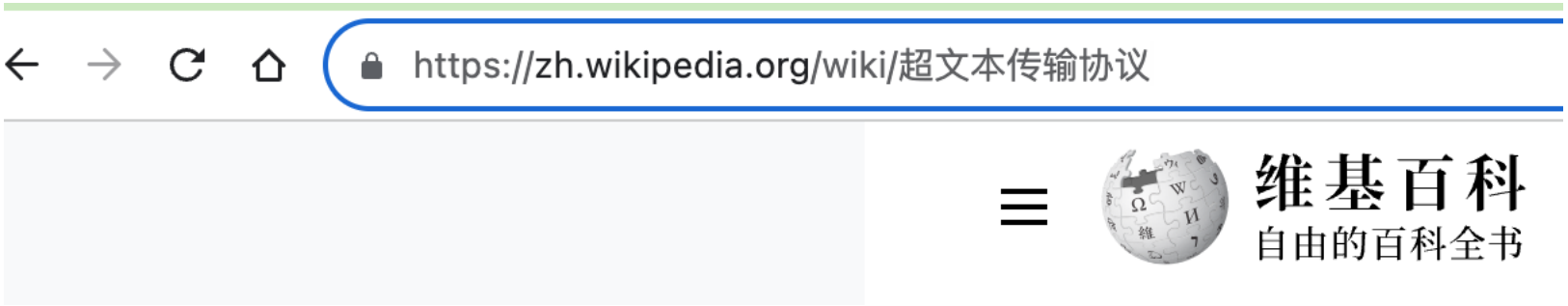
什么是 HTTP

浏览器输入 URL 之后的完整流程

- 浏览器查询 DNS
- 浏览器和服务端传输层建立连接
- 浏览器发送 HTTP 请求
- **服务端处理 HTTP 请求**
- 浏览器渲染

理性认识：什么是 HTTP

超文本传输协议（**H**yper**T**ext **T**ransfer **P**rotocol），是一种应用层协议

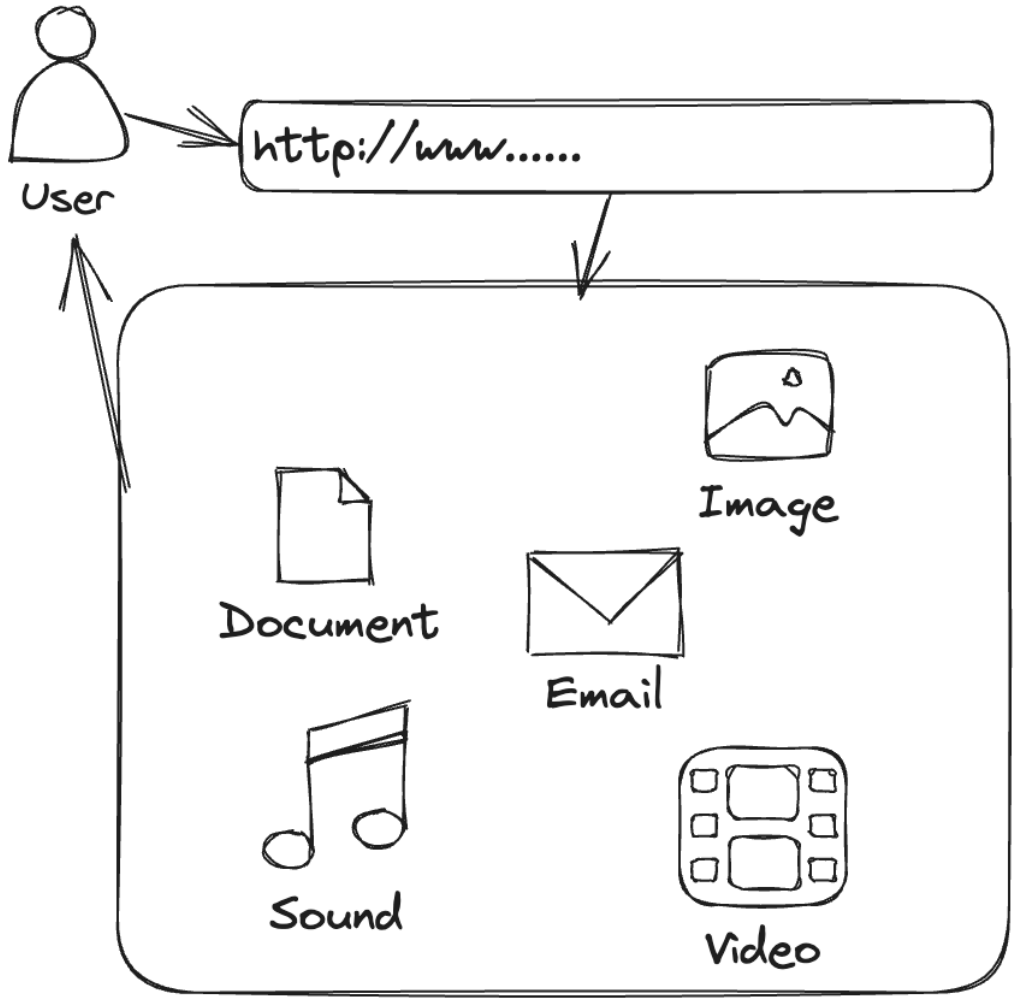


QA: 你会如何设计 HTTP 协议 ？

什么是 HTTP

HTTP 协议需要注意什么?

- 准确性
- 规范
- 可扩展
- 性能
- 安全
- ...



什么是 HTTP

HTTP 请求拆解示例

http: //localhost:3000/rust

Request Line: Method URI VERSION

```
GET /rust HTTP/1.1
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Postman-Token: 315e6a87-f1a7-4221-825a-446687791dfa
Host: localhost:3000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Header

什么是 HTTP

HTTP 请求拆解示例

http: //localhost:3000/rust

Request Line: Method URI VERSION

```
POST /rust HTTP/1.1
Content-Type: text/plain
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Postman-Token: 854f7398-003a-4803-b97f-aca858ac398a
Host: localhost:3000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 24

I'am a http post request
```

Header

Body

什么是 HTTP

HTTP 请求Body 的不同格式

```
POST /rust HTTP/1.1
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Postman-Token: 5d4a0265-9fdf-46f2-ab41-78bfb4d692a9
Host: localhost:3000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

name=xiaoming&age=19
```

Form 格式

```
POST /rust HTTP/1.1
Content-type: application/json; charset=utf-8
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Postman-Token: 854f7398-003a-4803-b97f-aca858ac398a
Host: localhost:3000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 55

{"name": "xiaoming", "info": {"age": 19, "male": true}}
```

JSON 格式

什么是 HTTP

HTTP 响应拆解示例

- 1xx：信息类
- 2xx：成功
- 3xx：重定向
- 4xx：客户端错误
- 5xx：服务端错误

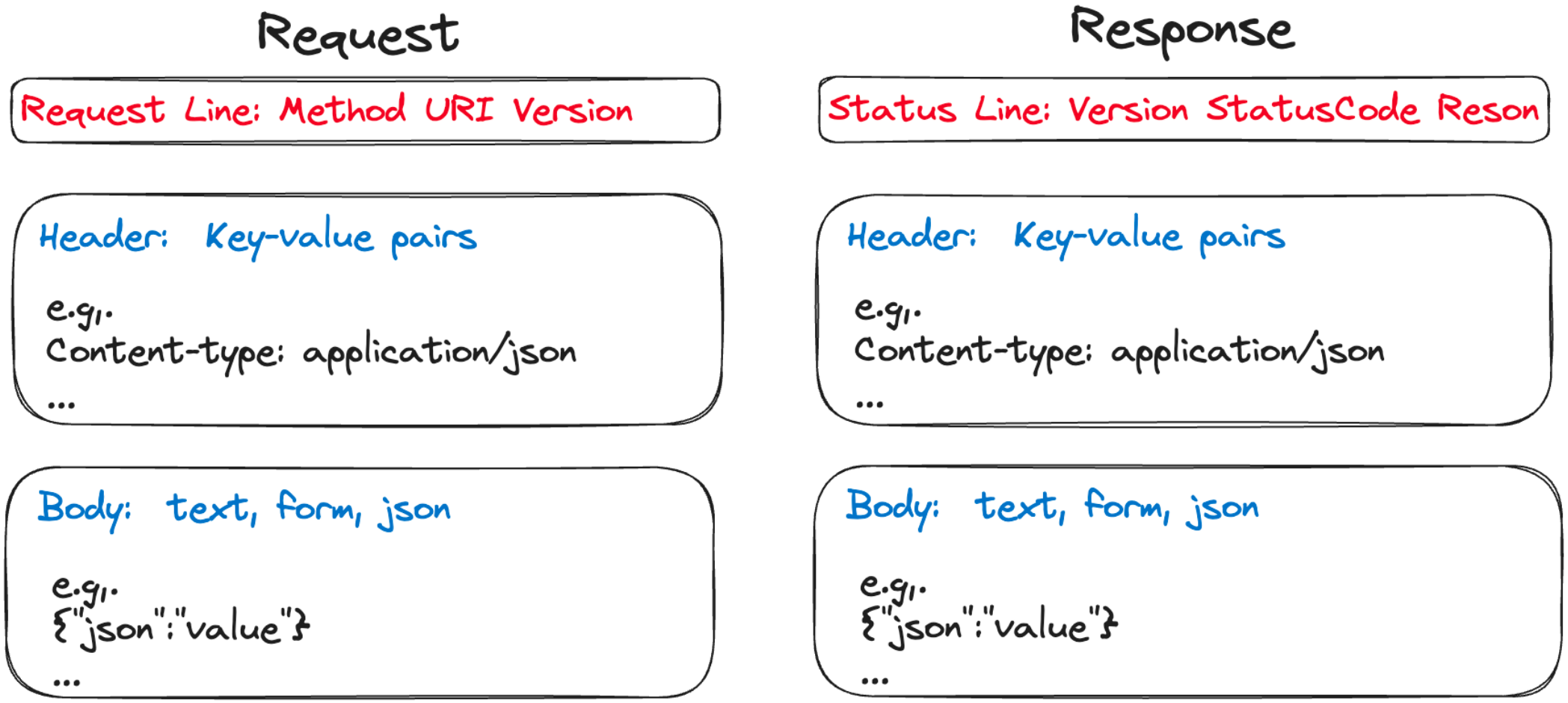
Status-Line : Version StatusCode Reason

```
HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8
content-length: 13
date: Thu, 07 Sep 2023 07:41:40 GMT

Hello, world!
```

Header

Body

什么是 HTTP


Request
Request Line: Method URI Version
Header: Key-value pairs
e.g.
Content-type: application/json
...
Body: text, form, json
e.g.
{"json":"value"}
...
Response
Status Line: Version StatusCode Reson
Header: Key-value pairs
e.g.
Content-type: application/json
...
Body: text, form, json
e.g.
{"json":"value"}
...

HTTP 各版本

版本	方法	特点
HTTP/0.9	GET	1. 仅用于静态 HTML 页面传输 2. 无状态码或元数据
HTTP/1.0	GET, POST, HEAD	1. 状态码、消息头及响应体 2. 支持 MIME 类型
HTTP/1.1	GET, POST, HEAD, PUT, DELETE, OPTIONS, PATCH, CONNECT, TRACE	1. 持久连接（Keep-Alive） 2. 管道传输（Pipelining ） 3. 分块传输编码（Chunked Transfer-Encoding） 4. 缓存策略改进
HTTP/2	同 HTTP/1.1	1. 二进制协议 2. 多路复用（Multiplexing） 3. 流优先级（Stream Prioritization ） 4. 头部压缩 5. 服务器推送（Server Push ）
HTTP/3	同 HTTP/1.1	1. 使用 QUIC 协议，有更低的延迟 2. 改进的拥塞控制 3. 自适应丢包恢复

HTTPs 就是 HTTP 的加密版本

Coding

reqwest : 一个易于使用的 HTTP client

例如: 使用 reqwest 发送 HTTP 请求，并且读取 HTTP 响应

```rust
    // reqwest::get() is a convenience function.
    //
    // In most cases, you should create/build a reqwest::Client and reuse
    // it for all requests.
    let res = reqwest::get(url).await?;

    eprintln!("Response: {:?} {}", res.version(), res.status());
    eprintln!("Headers: {:#?}\n", res.headers());

    let body = res.text().await?;
```

Source: https://github.com/seanmonstar/reqwest/blob/master/examples/simple.rs

Coding

Exercise: 使用 reqwest 实现小工具 HTTPie, 将 HTTP response 可视化展示

```
[➜  ~ http http://toutiao.com                                                  ]
HTTP/1.1 301 Moved Permanently
Connection: keep-alive
Content-Length: 160
Content-Type: text/html
Date: Sun, 10 Sep 2023 14:43:07 GMT
Location: http://www.toutiao.com/
Proxy-Status: 0000201301026000
Server: TLB
X-TT-LOGID: 20230910224307D86D730F754D51041B4E
x-tt-trace-host: 012571ef249a0f83e6134b7ad67f8905d12a71984bef20fc4037f20e481eec3
708e01355276aaf6f96906df4dbb9de53b4
x-tt-trace-tag: id=00;cdn-cache=miss

<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>TLB</center>
</body>
</html>
```

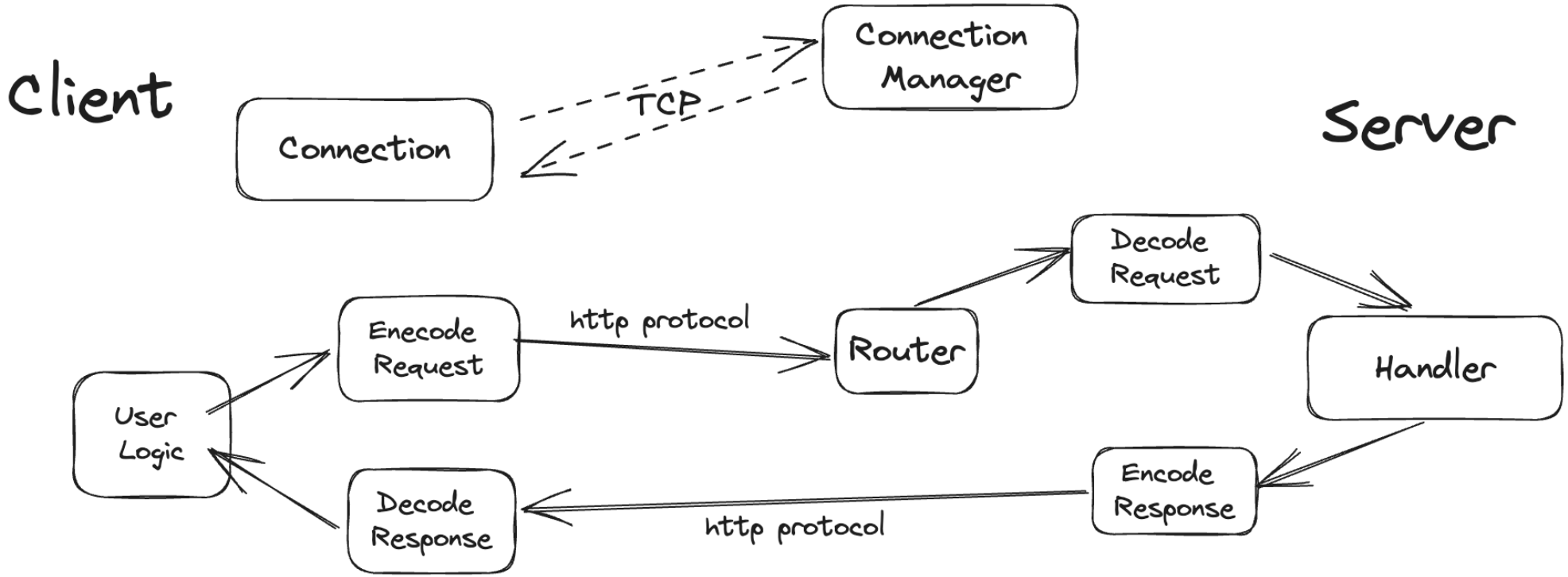
02

HTTP 框架

介绍HTTP 框架- 以 axum 为例子

什么是 HTTP 框架

HTTP 框架具体做了哪些工作？



什么是 HTTP 框架

HTTP 框架具体做了哪些工作？

Sever 端:

- 监听客户端请求
- Router: 路由
- Handler: 处理 request 并返回 response
- Handler 中间件

Client 端:

- 构造 request
- 请求 server

什么是 HTTP 框架

Rust HTTP 框架/组件有哪些？

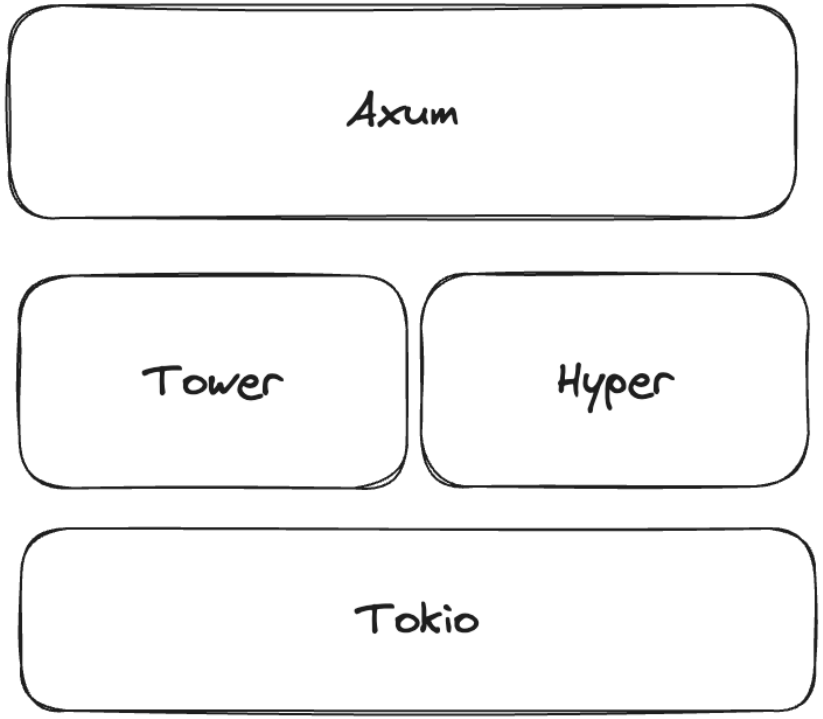
- Axum: 基于 tower 和 hyper 生态的HTTP 框架，易于使用
- Actix-web：内部使用 actor 模型，主打高性能
- Rocket: 使用宏提供简便容易上手的API
- Reqwest: 主要提供异步、易于使用的 HTTP Client
- Hyper: low-level 的 HTTP 框架，提供编解码和连接管理能力
- Matchit: 一个高性能URL路由组件
- Volo-HTTP: 基于 motore 中间件抽象，开发中

QA: 你会如何选择一个 HTTP 框架？

什么是 HTTP 框架

为什么以 Axum 为例子？

- 基于 Rust 类型系统处理 HTTP 请求/响应
- 简单准确的错误处理
- 充分利用已有生态: tower 和 hyper
- 开源社区活跃



使用 HTTP 框架 vs 不用

使用 Axum 写一个 HTTP Server，接受 HTTP 请求后 返回 “hello world”

```rust
use axum::{
    routing::get,
    Router,
};

#[tokio::main]
async fn main() {
    // build our application with a single route
    let app = Router::new().route("/", get(|| async { "Hello, World!" }));

    // run it with hyper on localhost:3000
    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

使用 HTTP 框架 vs 不用

```rust
use axum::{
    routing::get,
    Router,
};

#[tokio::main]
async fn main() {
    // build our application with a single route
    let app = Router::new().route("/", get(|| async { "Hello, World!" }));

    // run it with hyper on localhost:3000
    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

使用 HTTP 框架 vs 不用

如果不用 HTTP 框架实现，这里还省略了对 HTTP 请求的校验

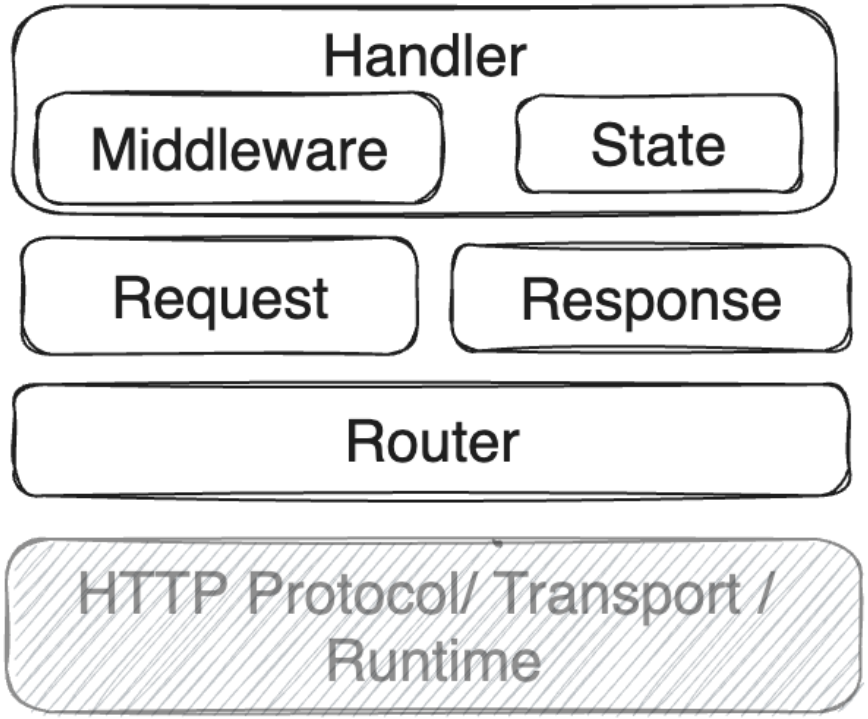

```rust
use std::error::Error;
use tokio::{
    io::{AsyncReadExt, AsyncWriteExt},
    net::TcpListener,
};

fn get_response(req: &[u8]) -> String {
    let request: Cow<'_, str> = String::from_utf8_lossy(req);

    let target_url: Option<&str> = request Cow<'_, str>
        .lines() Lines<'_>
        .next() Option<&str>
        .and_then(|line: &str| line.split_whitespace().nth(1));

    match target_url {
        Some("/") => "HTTP/1.1 200 OK\r\n\
                      Content-Type: text/plain\r\n\
                      Content-Length: 13\r\n\r\n\
                      Hello, World!" &str
                      .to_string(),
        Some("/not_found") => "HTTP/1.1 404 Not Found\r\n\
                               Content-Type: text/plain\r\n\
                               Content-Length: 9\r\n\r\n\
                               Not Found" &str
                               .to_string(),
        Some(_) | None => "HTTP/1.1 400 Bad Request\r\n\
                           Content-Type: text/plain\r\n\
                           Content-Length: 11\r\n\r\n\
                           Bad Request" &str
                           .to_string(),
    }
} fn get_response
```

```rust
#[tokio::main]
▶ Run | Debug
async fn main() -> Result<(), Box<dyn Error>> {
    let listener: TcpListener = TcpListener::bind("0.0.0.0:3000").await?;

    loop {
        let (mut socket: TcpStream, _addr: SocketAddr) = listener.accept().await?;
        tokio::spawn(future: async move {
            let mut buffer: [u8; 1024] = [0; 1024];

            match socket.read(buf: &mut buffer).await {
                Ok(_) => {
                    let response: String = get_response(req: &buffer);

                    if let Err(e: Error) = socket.write_all(src: response.as_bytes()).await {
                        eprintln!("Failed to send response: {}", e);
                    }
                }
                Err(e: Error) => eprintln!("Failed to read from socket: {}", e),
            }
        });
    }
}
```

Axum 整体框架

Axum 提供了 HTTP server 的抽象接口:

- Router: HTTP 路由
- Request/Response: HTTP 请求/响应类型
- Handler: 处理 HTTP 请求的函数
- Middleware: Handler 中间件
- State: Middleware 共享状态



- 基于 hyper 提供了 HTTP protocol 和 TCP transport 能力
- 使用 tokio runtime
- 复用 tower middleware 抽象和生态

路由

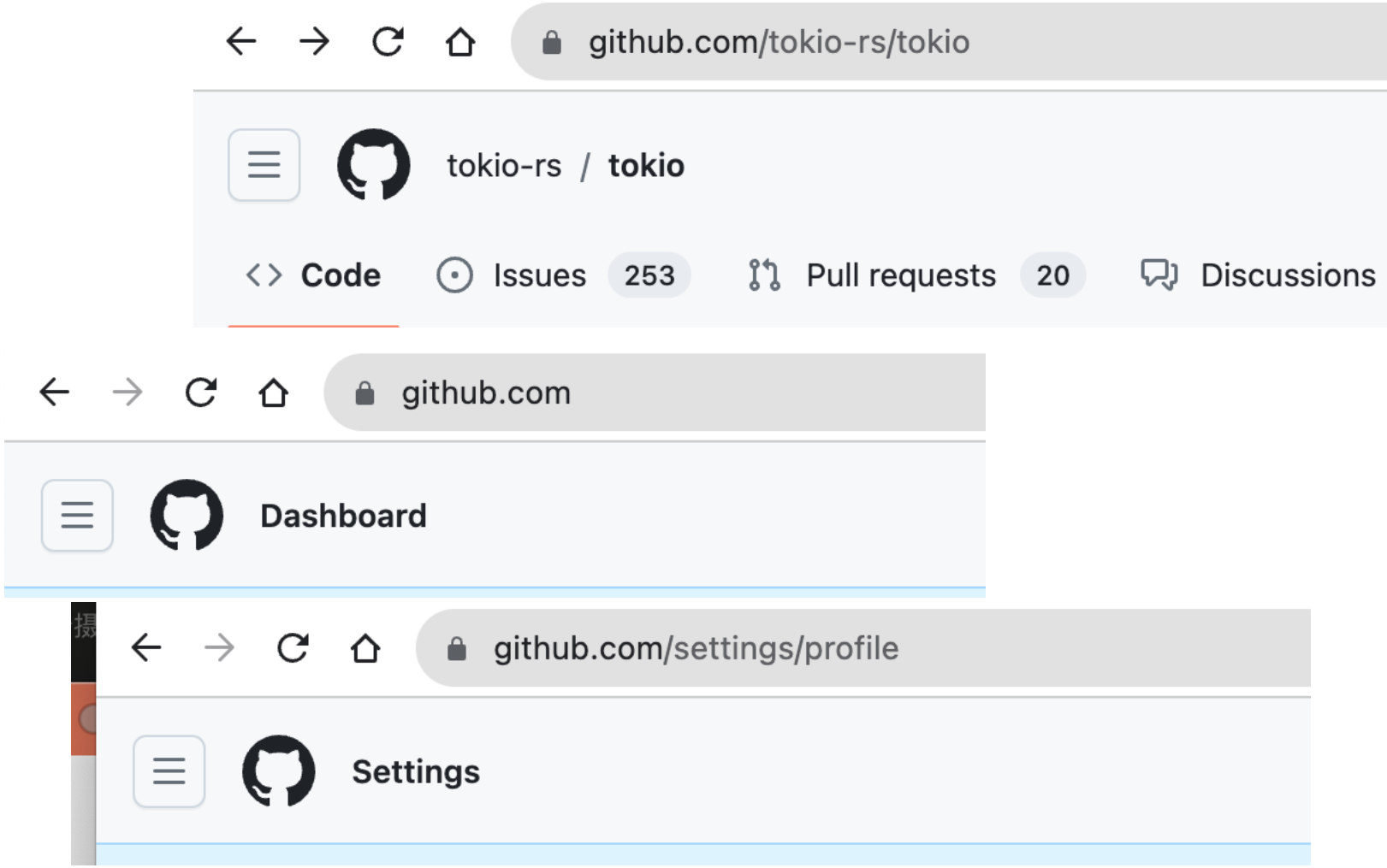
为什么需要路由？

https://github.com/tokio-rs/tokio

https://github.com/settings

https://github.com/login

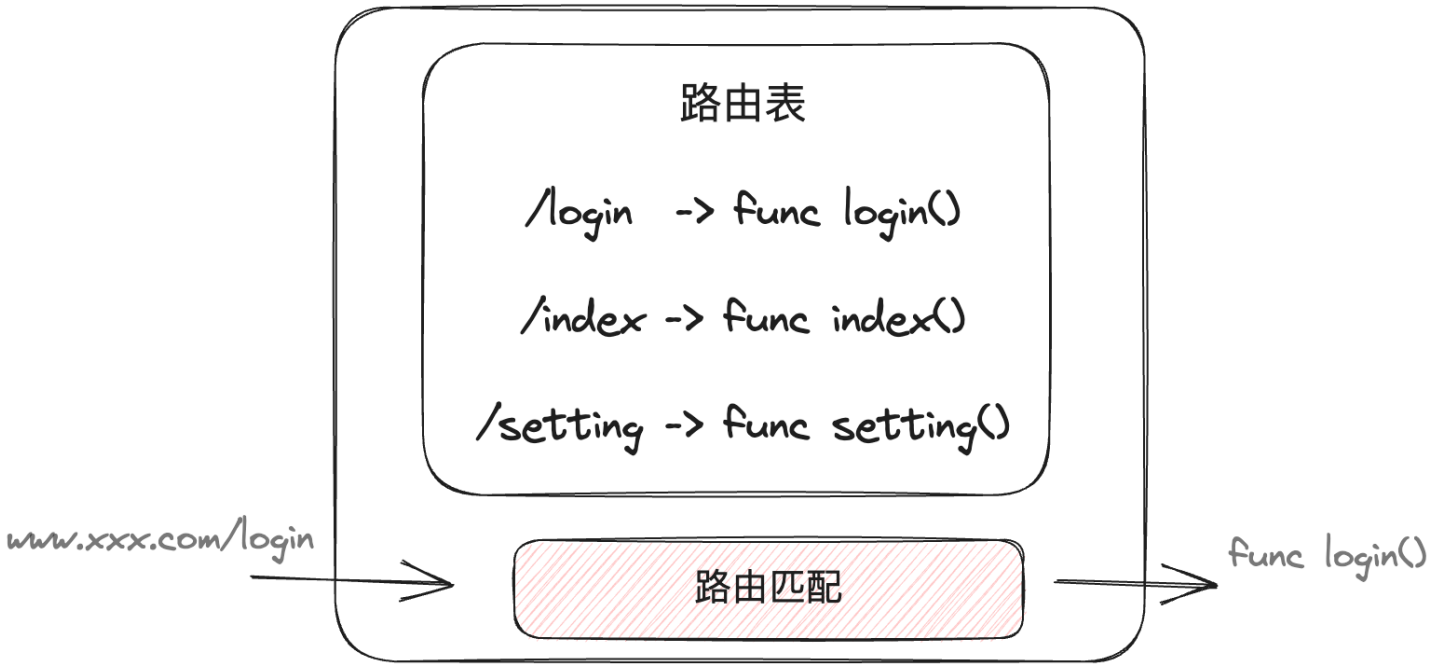
https://github.com/



路由

路由就是将 HTTP 请求与 Handler 映射起来，每一个 HTTP 请求都有会一个对应的 Handler

基于 HTTP 请求行（Request Line) 中的 **Method 和 URI**，可以完成路由



路由

基于 URI 的路由匹配支持多种模式

- 静态匹配
  - /
  - /login
  - /user/123
- 动态匹配
  - /user**/:id** 匹配 /user**/123** 等
  - /users/:id/tweets
- 通配符
  - /user **/*id** 匹配 /user**/123** /user**/123/twitters**等

```rust
use axum::{Router, routing::get};

// our router
let app = Router::new()
    .route("/", get(root))
    .route("/foo", get(get_foo).post(post_foo))
    .route("/foo/bar", get(foo_bar));

// which calls one of these handlers
async fn root() {}
async fn get_foo() {}
async fn post_foo() {}
async fn foo_bar() {}
```

路由

基于 Method 的路由匹配

例如: POST 和 GET 可以使用不同的 handler：

- GET 使用 list_users
- POST 使用 create_users

```
let app = Router::new()
    .route("/", get(root))
    .route("/users", get(list_users).post(create_user))
    .route("/users/:id", get(show_user))
    .route("/api/:version/users/:id/action", delete(do_users_action))
    .route("/assets/*path", get(serve_asset));
```

Handler

Handler 就是处理 HTTP request 的函数

Naïve Handler 函数抽象: 在handler 里面处理 HttpRequest， 然后返回 HttpResponse

Handler: Fn(req: HttpRequest) -> HttpResponse

Handler

Handler 中如何优雅地处理 request 和 response 参数？

- HTTP request/response 有不同组成: URI/StatusCode, Header, Body
- HTTP request/response Body 有不同数据格式: Json, Form, Plain text
- 直接处理 HTTP request/response 会在 handler 中引入 HTTP 协议的解析/编码
- **通常HTTP协议的解析/编码是与用户逻辑无关的**

Handler

Axum Handler:

- Handler 函数的参数可变，支持 0个或多个 ”extractor” 参数，能够从 HTTP request 中提取
- Extractor 基于Rust类型系统
  - 利用 FromRequest Trait 从 HTTP request 提取/解析 handler 所需参数，无需在handler 中解析 HTTP 请求
  - 对 Handler 参数进行了类型约束，减少运行期出错的可能
- Handler 返回值个数可变，能够自动转为 HTTP response
  - 利用 IntoResponse Trait 进行编码和组装，无需在 handler中进行
- Handler 函数是异步的

async fn handler(arg1: T1, arg2: T2 …) -> (res1, res2, …)

Handler 参数: Extractor

Axum Extractor 的例子

async fn create_user(**Json(playload): Json<CreateUser>**)
-> (StatusCode, Json<User>)

- Json(playload) 是 函数的参数，这里使用了 Rust Pattern
- Json<CreateUser> 是函数参数的类型，HTTP request 会通过 **FromRequest trait** 自动转换成该类型
- StatusCode 是 HTTP response 的 code
- Json<User> 是返回的body，会通过 **IntoResponse trait** 自动编码成 JSON 格式

```rust
// the input to our `create_user` handler
#[derive(Deserialize)]
struct CreateUser {
    username: String,
}

// the output to our `create_user` handler
#[derive(Serialize)]
struct User {
    id: u64,
    username: String,
}

async fn create_user(
    // this argument tells axum to parse the request body
    // as JSON into a `CreateUser` type
    Json(payload): Json<CreateUser>,
) -> (StatusCode, Json<User>) {
    // insert your application logic here
    let user = User {
        id: 1337,
        username: payload.username,
    };

    // this will be converted into a JSON response
    // with a status code of `201 Created`
    (StatusCode::CREATED, Json(user))
}
```

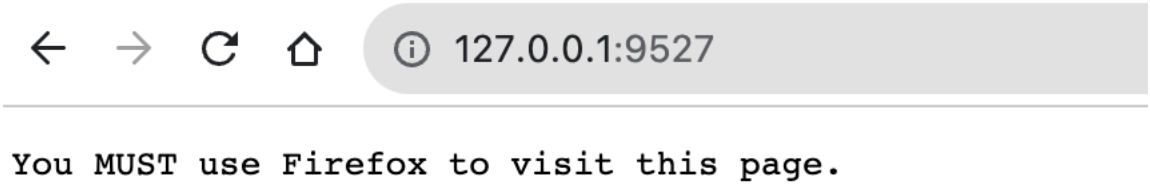
Handler 参数: Extractor

- Axum 定义的常见的 extractor

BodyStream	Extractor that extracts the request body as a `Stream`.
ConnectInfo `tokio`	Extractor for getting connection information produced by a `Connected`.
DefaultBodyLimit	Layer for configuring the default request body limit.
Host	Extractor that resolves the hostname of the request.
MatchedPath `matched-path`	Access the path in the router that matches the request.
Multipart `multipart`	Extractor that parses `multipart/form-data` requests (commonly used with file uploads).
OriginalUri `original-uri`	Extractor that gets the original request URI regardless of nesting.
Path	Extractor that will get captures from the URL and parse them using `serde`.
Query `query`	Extractor that deserializes query strings into some type.
RawBody	Extractor that extracts the raw request body.
RawForm	Extractor that extracts raw form requests.
RawPathParams	Extractor that will get captures from the URL without deserializing them.
RawQuery	Extractor that extracts the raw query string, without parsing it.
State	Extractor for state.
WebSocketUpgrade `ws`	Extractor for establishing WebSocket connections.

Handler 参数: Extractor

使用 FromRequest trait 自己实现一个 Extractor




```rust
#[async_trait]
impl<B> FromRequest<B> for UserAgentInfo
where
    B: Send,
{
    type Rejection = (StatusCode, String);
    async fn from_request(req: &mut RequestParts<B>) -> Result<Self, Self::Rejection> {
        let user_agent = req
            .headers()
            .and_then(|headers| headers.get(axum::http::header::USER_AGENT))
            .and_then(|value| value.to_str().ok())
            .unwrap_or("");
        tracing::debug!("该用户UserAgent是: {:?}", user_agent);
        if !user_agent.contains("Firefox") {
            tracing::error!("非Firefox浏览器, 禁止访问");
            return Err((
                StatusCode::BAD_REQUEST,
                "You MUST use Firefox to visit this page.".to_string(),
            ));
        }
        Ok(UserAgentInfo {})
    }
}
```

Handler 参数: Extractor

为什么 axum Handler 支持可变参数？

因为 axum 内部已经给 Tuple(Args) 类型 实现了 FromRequest trait

```
macro_rules! all_the_tuples {
    ($name:ident) => {
        $name!([], T1);
        $name!([T1], T2);
        $name!([T1, T2], T3);
        $name!([T1, T2, T3], T4);
        $name!([T1, T2, T3, T4], T5);
        $name!([T1, T2, T3, T4, T5], T6);
        $name!([T1, T2, T3, T4, T5, T6], T7);
        $name!([T1, T2, T3, T4, T5, T6, T7], T8);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8], T9);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9], T10);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10], T11);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11], T12);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12], T13);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13], T14);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14], T15);
        $name!([T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15], T16);
    };
}
```

Handler 返回值

与 Handler 参数 “extractor” 类似，

Axum 给常见的 Handler 的返回值都实现了 IntoResponse Trait：

- StatusCode
- ()/ Result
- String
- Json
- Html
- Tuple(T1, T2, ..., Tn), 需要 T1, T2 都实现了IntoResponseParts
- ...

```rust
pub trait IntoResponse {
    /// Create a response.
    fn into_response(self) -> Response;
}

impl IntoResponse for StatusCode {
    fn into_response(self) -> Response {
        let mut res: Response<Body> = ().into_response();
        *res.status_mut() = self;
        res
    }
}

impl IntoResponse for () {
    fn into_response(self) -> Response {
        Body::empty().into_response()
    }
}
```

中间件

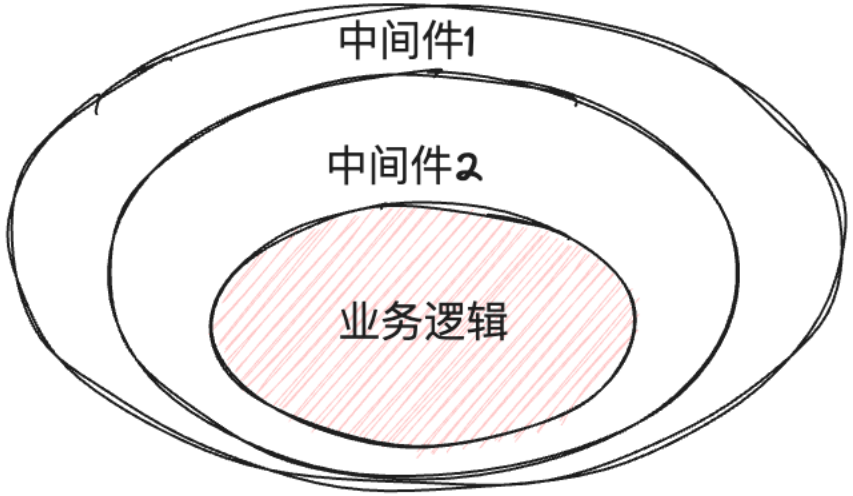
不同的 handler，如果有些处理逻辑是相同的，怎么办？

例如，打印日志、超时控制等这些通用功能

请使用 HTTP 框架中间件

中间件

洋葱模型： 一种比较好的中间件设计模式



中间件

Axum 中的 ”layer”

Layer 可以实现在不同层级：

- Router
- MethodRouter
- Handler

```rust
use axum::{
    routing::get,
    Extension,
    Router,
};
use tower_http::{trace::TraceLayer};
use tower::ServiceBuilder;

async fn handler() {}

#[derive(Clone)]
struct State {}

let app = Router::new()
    .route("/", get(handler))
    .layer(
        ServiceBuilder::new()
            .layer(TraceLayer::new_for_http())
            .layer(Extension(State {}))
    );
```

```rust
use axum::{
    routing::get,
    handler::Handler,
    Router,
};
use tower::limit::{ConcurrencyLimitLayer, ConcurrencyLimit};

async fn handler() { /* ... */ }

let layered_handler = handler.layer(ConcurrencyLimitLayer::new(64));
let app = Router::new().route("/", get(layered_handler));
```

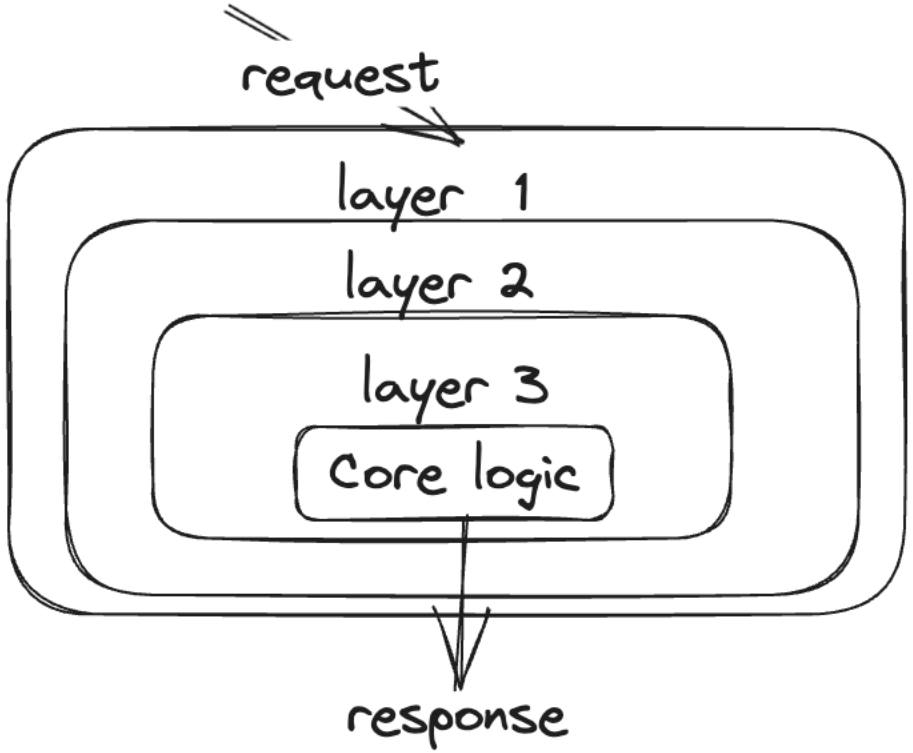
```rust
use axum::{routing::get, Router};
use tower::limit::ConcurrencyLimitLayer;

async fn hander() {}

let app = Router::new().route(
    "/",
    // All requests to `GET /` will be sent through `ConcurrencyLimitLayer`
    get(hander).layer(ConcurrencyLimitLayer::new(64)),
);
```

中间件

Axum 中的 ”layer”

Layer 的调用顺序 使用 ServiceBuilder 组装

```
let app = Router::new()
    .route("/", get(handler))
    .layer(
        ServiceBuilder::new()
            .layer(layer_one)
            .layer(layer_two)
            .layer(layer_three),
    );
```



使用 axum 写一个中间件

写一个中间件 打印 request 和 response

Source: https://github.com/tokio-rs/axum/blob/main/examples/print-request-response/src/main.rs

```rust
#[tokio::main]
async fn main() {
    tracing_subscriber::registry()
        .with(
            tracing_subscriber::EnvFilter::try_from_default_env()
                .unwrap_or_else(|_| "example_print_request_response=debug,tower_http=debug".into()),
        )
        .with(tracing_subscriber::fmt::layer())
        .init();

    let app = Router::new()
        .route("/", get(|| async move { "Hello from `GET /`" }))
        .layer(middleware::from_fn(print_request_response));

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .unwrap();
    tracing::debug!("listening on {}", listener.local_addr().unwrap());
    axum::serve(listener, app).await.unwrap();
}
```

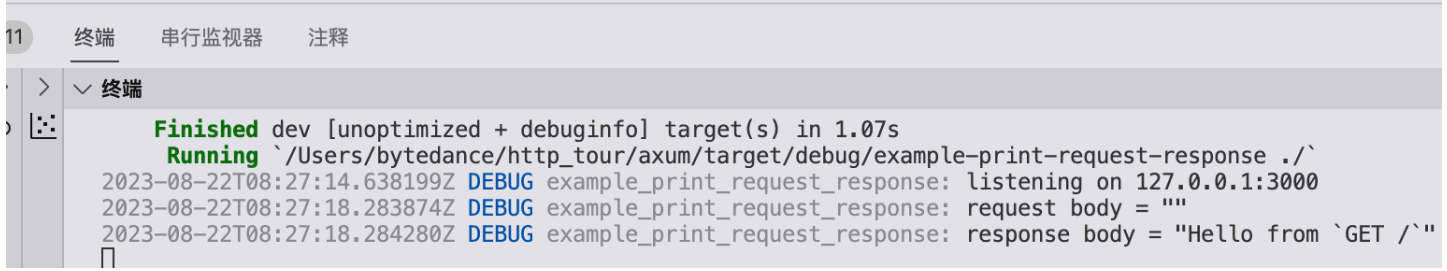
使用 axum 写一个中间件

写一个中间件 打印 request 和 response

预处理

后处理

```rust
async fn print_request_response(
    req: Request,
    next: Next,
) -> Result<impl IntoResponse, (StatusCode, String)> {
    let (parts, body) = req.into_parts();
    let bytes = buffer_and_print("request", body).await?;
    let req = Request::from_parts(parts, Body::from(bytes));

    let res = next.run(req).await;

    let (parts, body) = res.into_parts();
    let bytes = buffer_and_print("response", body).await?;
    let res = Response::from_parts(parts, Body::from(bytes));

    Ok(res)
}
```



错误处理

如果中间件需要返回错误怎么办？

下面哪种做法更好？

1. 中断连接

2. 保持连接，返回错误信息

错误处理

最佳实践: 将错误转换成 StatusCode 返回

- 使用 HandleErrorLayer 中间件处理错误
- 将错误转换成 StatusCode 和 Msg

```rust
use axum::{
    Router,
    BoxError,
    routing::get,
    http::StatusCode,
    error_handling::HandleErrorLayer,
};
use std::time::Duration;
use tower::ServiceBuilder;

let app = Router::new()
    .route("/", get(|| async {}))
    .layer(
        ServiceBuilder::new()
            // `timeout` will produce an error if the handler takes
            // too long so we must handle those
            .layer(HandleErrorLayer::new(handle_timeout_error))
            .timeout(Duration::from_secs(30))
    );

async fn handle_timeout_error(err: BoxError) -> (StatusCode, String) {
    if err.is::<tower::timeout::error::Elapsed>() {
        (
            StatusCode::REQUEST_TIMEOUT,
            "Request took too long".to_string(),
        )
    } else {
        (
            StatusCode::INTERNAL_SERVER_ERROR,
            format!("Unhandled internal error: {}", err),
        )
    }
}
```

共享状态

例如多个中间件需要访问同一个数据库、全局 hashmap等？

很简单，我们往 handler 中新增一个 State 参数

Handler: Fn(req: HttpRequest, **state: State**) -> HttpResponse

共享状态

如何传入 state 参数？

通过 with_state 传递 state 参数

- Router.with_state
- Handler.with_state
- ...

```rust
// setup connection pool
let pool: Pool<Postgres> = PgPoolOptions::new()
    .max_connections(5)
    .acquire_timeout(Duration::from_secs(3))
    .connect(&db_connection_str)
    .await Result<Pool<Postgres>, Error>
    .expect("can't connect to database");

// build our application with some routes
let app: Router = Router::new()
    .route(
        "/",
        get(handler: using_connection_pool_extractor).post
        (handler: using_connection_extractor),
    )
    .with_state(pool);
```

Source: https://github.com/tokio-rs/axum/blob/main/examples/sqlx-postgres/src/main.rs

共享状态

如何使用 state 参数？

- with_state 往 handler 中传入的 state 参数: pool
-
- Handler 使用时采用了和 Extractor 类似的模式，通过 State<PgPool> 提取 state 参数

```
// we can extract the connection pool with `State`
async fn using_connection_pool_extractor(
    State(pool: Pool<Postgres>): State<PgPool>,
) -> Result<String, (StatusCode, String)> {
    sqlx::query_scalar(sql: "select 'hello world from pg'") QueryScalar<'_, Postgres, …, …>
        .fetch_one(executor: &pool) impl Future<Output = Result<…, …>>
        .await Result<String, Error>
        .map_err(op: internal_error)
}
```

QA: state 里面如何共享可变数据

小结

Axum HTTP 框架：

- 路由: 处理 URL 和 Handler 之间的映射
- Handler: 实际的 request 处理函数
- Extractor: 借助 Rust 类型系统，优雅处理 handler 的参数
- 中间件: 通用的 Handler 处理逻辑
- 错误处理: 基于 Rust 类型系统，确保 HTTP 错误正确返回
- 共享状态: 使用 State，如果是可变数据，需要加锁

03 HTTP实战案例

介绍实战经验和案例

实战经验-错误处理

两种错误处理方式:

1. 使用 anyhow::Result
2. 使用 std::result::Result

```rust
use anyhow::Result;

fn get_cluster_info() -> Result<String> {
    let config: String = std::fs::read_to_string(path: "cluster.json")?;
    let map: String = serde_json::from_str(&config)?;
    Ok(map)
}

fn get_cluster_info2() -> std::result::Result<String, serde_json::Error> {
    let config: String = std::fs::read_to_string(path: "cluster.json")?;
    let map: String = serde_json::from_str(&config)?;
    Ok(map)
}
```

实战经验-错误处理

具体报错信息:

因为 io::Error 没法转换成 serde_json::Error

```
error[E0277]: `?` couldn't convert the error to `serde_json::Error`
  --> src/main.rs:24:57
   |
23 | fn get_cluster_info2() -> std::result::Result<String, serde_json::Error> {
   |                           ------------------------------------------- expected `serde_json::Error`
 because of this
24 |     let config = std::fs::read_to_string("cluster.json")?;
   |                                                         ^ the trait `From<std::io::Error>` is not imp
lemented for `serde_json::Error`
   |
```

实战经验–错误处理

推荐使用 anyhow Result,

anyhow::Result<T> = std::result::Result<T, Box<dyn, std::error::Error>>

只要**定义的Error实现了 std::error::Error trait**，就可以放心传递错误了

```
fn get_cluster_info3() -> std::result::Result<String, Box<dyn std::error::Error>> {
    let config: String = std::fs::read_to_string(path: "cluster.json")?;
    let map: String = serde_json::from_str(&config)?;
    Ok(map)
}
```

实战经验-日志处理

如何打 log？使用 tracing 库

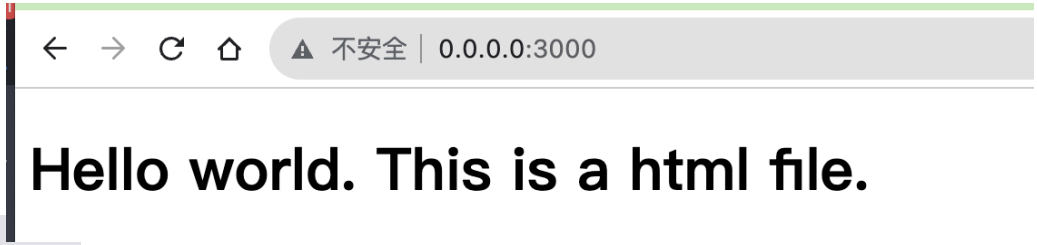
Log 有哪些级别:

- Fatal
- Error
- Warn
- Info
- Debug
- Trace

```rust
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

▶ Run | Debug
fn main() {
    // 需要注册 tracing 后，终端才会输出日志
    tracing_subscriber::registry().with(fmt::layer()).init();
    log::info!("Hello world from log");
    tracing::info!("Hello from tracing");
}
```

实战案例1 – 加载 static 网页

```rust
use axum::{response::Html, routing::get, Router};

async fn handler() -> Html<&'static str> {
    Html(include_str!("../files/index.html"))
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let app: Router = Router::new

    axum::Server::bind(&"0.0.0.0:
        .serve(app.into_make_serv
        .await
        .unwrap();
}
```



Struct axum::response::Html

```
pub struct Html<T>(pub T);
```

[–] An HTML response.

Will automatically get `Content-Type: text/html`.

Macro std::include_str

1.0.0 · source · [

```
macro_rules! include_str {
    ($file:expr $(,)?) => { ... };
}
```

[–] Includes a UTF-8 encoded file as a string.

The file is located relative to the current file (similarly to how modules are found). The provided path is interpreted in a platform-specific way at compile time. So, for instance, an invocation with a Windows path containing backslashes `\` would not compile correctly on Unix.

This macro will yield an expression of type `&'static str` which is the contents of the file.

实战案例2 – todos

使用 axum 实现 任务列表
source: https://github.com/tokio-rs/axum/tree/main/examples/todos

```
// Compose the routes
let app: Router = Router::new()
    .route("/todos", get(handler: todos_index).post(handler: todos_create))
    .route("/todos/:id", patch(handler: todos_update).delete(handler: todos_delete))
    // Add middleware to all routes
    .layer(
        ServiceBuilder::new() ServiceBuilder<Identity>
            .layer(HandleErrorLayer::new(|error: BoxError| async move {
                if error.is::<tower::timeout::error::Elapsed>() {
                    Ok(StatusCode::REQUEST_TIMEOUT)
                } else {
                    Err((
                        StatusCode::INTERNAL_SERVER_ERROR,
                        format!("Unhandled internal error: {}", error),
                    ))
                }
            })) ServiceBuilder<Stack<HandleErrorLayer<…, …>, …>>
            .timeout(Duration::from_secs(10)) ServiceBuilder<Stack<TimeoutLayer, …>>
            .layer(TraceLayer::new_for_http()) ServiceBuilder<Stack<TraceLayer<…>, …>>
            .into_inner(),
    )
    .with_state(db);
```

实战案例3 – 使用 reqwest

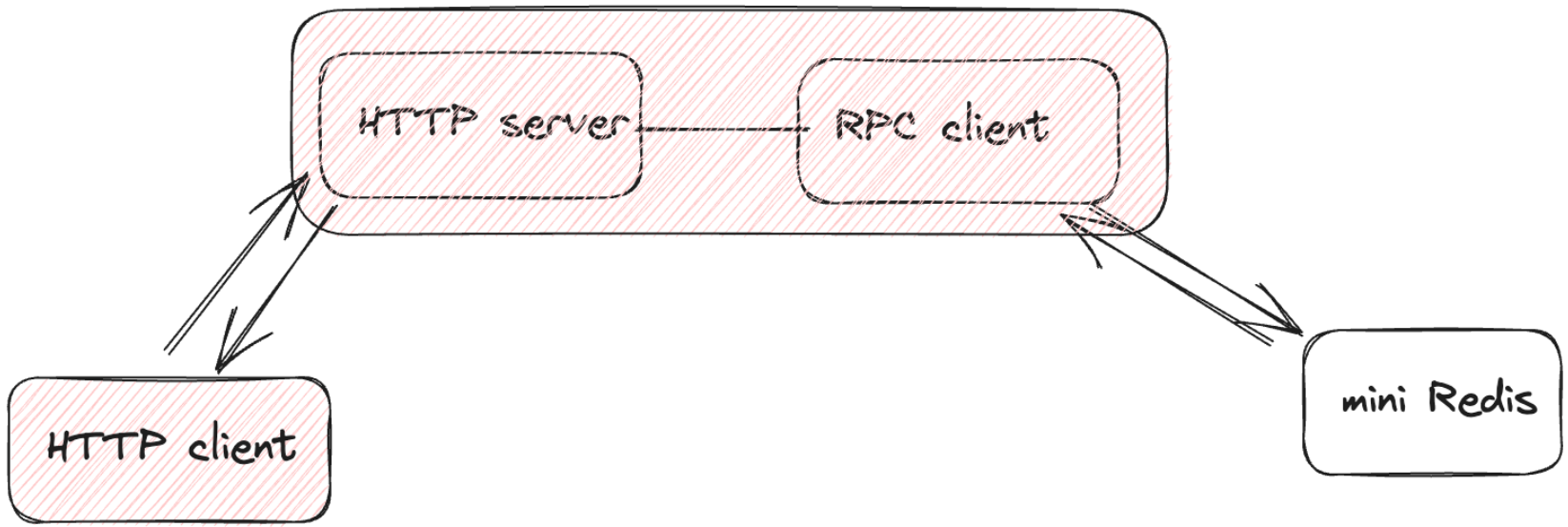
使用 axum + reqwest 实现 proxy
Source: https://github.com/tokio-rs/axum/tree/main/examples/reqwest-response

```rust
async fn proxy_via_reqwest(State(client: Client): State<Client>) -> Response {
    let reqwest_response: Response = match client.get(url: "http://127.0.0.1:3000/stream").send().await {
        Ok(res: Response) => res,
        Err(err: Error) => {
            tracing::error!(%err, "request failed");
            return StatusCode::BAD_GATEWAY.into_response();
        }
    };

    let mut response_builder: Builder = Response::builder().status(reqwest_response.status());

    // This unwrap is fine because we haven't insert any headers yet so there can't be any invalid
    // headers
    *response_builder.headers_mut().unwrap() = reqwest_response.headers().clone();

    response_builder Builder
        .body(Body::from_stream(reqwest_response.bytes_stream())) Result<Response<Body>, Error>
        // Same goes for this unwrap
        .unwrap()
}
```

上机实验

实验：

- 使用 axum 搭建一个 HTTP server，server 内部调用 mini-redis RPC，并且将RPC 结果以 HTTP 方式返回给 client
- 使用 reqwest 构造 HTTP client 请求



也可以自由发挥，参考 axum 例子: https://github.com/tokio-rs/axum/tree/v0.6.x/examples

上机实验

一个简单 demo 演示

https://github.com/liuq19/http_example

参考资料

1. Axum 文档 https://docs.rs/axum/latest/axum/
2. Axum 官方示例 https://github.com/tokio-rs/axum/tree/v0.6.x/examples
3. Axum Project https://github.com/tokio-rs/axum/blob/main/ECOSYSTEM.md#project-showcase
4. Reqwest https://docs.rs/reqwest/latest/reqwest/

THANKS

CloudWeGo