

# Aex : Async-first, Executor-based Web Framework for Rust

 calidion

<https://github.com/calidion>

# AEX — Async-first, Executor-based Web Framework for Rust

一个轻量、可控、忠于 HTTP 本质的 Rust Web 框架

## AEX 是什么？

AEX.rs 是一个轻量级异步 Rust Web 框架

1. 提供直观的，易用的HTTP路由
2. 提供 简单，直观，正确的中间件机制
3. 原生支持 WebSocket，只需要放到中间件上，就可以实现WebSocket处理
4. 面向真实网络 I/O，而非抽象叠加

为 Rust 开发者提供更清晰、更可控的 Web 编程体验

# Aex安装与使用

## 1. 创建项目目录

```
cargo new xxx  
cd xxx
```

将xxx换成自己的项目名称

## 2. 安装aex

```
cargo add aex
```

## 3. 安装依赖

```
cargo add tokio futures futures_util anyhow
```

安装完成后，将下面的 `hello world` 跑通就可以进行项目的相关开发了。

# 最简单的Hello World实现

```
#[tokio::main(flavor = "multi_thread")]
async fn main() -> anyhow::Result<()> {
    // 构建 Router
    let mut route = Router::new(NodeType::Static("root".into()));
    route!(
        route,
        get!("/", |ctx: &mut HTTPContext| {
            Box::pin(async move {
                ctx.res.body.push("Hello world!".to_string());
                // false = 不继续 middleware (如果你还保留这个语义)
                true
            }).boxed()
        })
    );
    //启动 HTTPServer
    let ip = "0.0.0.0";
    let port = 8080;
    let addr: SocketAddr = format!("{:}", ip, port).parse()?;
    let server = HTTPServer::new(addr, route);
    server.run().await?;
}
```

注意必须补充下面的文件头内容，才能正真运行起来：

```
use std::{ net::SocketAddr, sync::Arc };

use clap::Parser;

use aex::{
    get,
    route,
    router::{ NodeType, Router },
    server::HTTPServer,
    types::{ BinaryHandler, HTTPContext, TextHandler },
    websocket::WebSocket, // 👉 关键：TrieRouter
};
use futures::FutureExt;
```

这段框架代码主要就是三个部分：

### 1. 支持异步与多线程的主函数框架

```
#[tokio::main(flavor = "multi_thread")]  
async fn main() -> anyhow::Result<()> {  
}
```

### 2. 路由构建与处理函数编写

### 3. HTTP服务器的创建与运行

```
let ip = "0.0.0.0";  
let port = 8080;  
let addr: SocketAddr = format!("{:}", ip, port).parse()?;  
let server = HTTPServer::new(addr, route);  
server.run().await?;
```

1.和3.部分属于固定套路，只要照搬就可以了，我就不再详细讲解。

2.部分的结构是编写HTTP业务逻辑的核心。我们可以详细看一下。

```
// 1. 创建静态根 (root) 节点
let mut route = Router::new(NodeType::Static("root".into()));
// 2. 使用route!宏添加路由信息
route!(
    route,
    // 3. 使用get!宏创建一个只响应http get请求的处理函数
    get!("/", |ctx: &mut HTTPContext| {
        // 闭包函数是Executor对象，是一个传入核心对象HTTPContext的异步闭包处理函数
        // Box::pin().boxed()是套路代码，为实现异步闭包所必须的封装。
        Box::pin(async move {
            // 这里是核心处理逻辑，所有处理都在这里编写
            // 输出影响默认在内部调用，直到你返回false
            ctx.res.body.push("Hello world!".to_string());
            // false = 不继续 middleware，不执行最终handle
            true
        }).boxed()
    })
);
```



上面的代码等价于：

```
let mut route = Router::new(NodeType::Static("root".into()));
route.insert(
    "/",
    Some("GET"),
    Arc::new(|ctx: &mut HTTPContext| {
        Box::pin(async move {
            ctx.res.body.push("Hello world!".to_string());
            true
        }).boxed()
    }),
    None // 传入 WebSocket 中间件，这里没有中间件使用None
);
```

# 中间件设计理念

中间件 = Executor 的有序数组

1. 中间件不是“魔法层”
2. 不是隐式嵌套

而是 显式、有序、可预测 的 执行链，可以清晰的反映 HTTP 请求的执行 路径

```
[ Executor A ] → [ Executor B ] → [ Executor C ]
```

## 为什么不用“洋葱模型”？

许多 Rust Web 框架采用了错误的洋葱模型。导致一系列错误的结果：

1. 执行顺序不直观
2. 控制流被隐藏
3. 调试成本高
4. 实际与 HTTP 请求生命周期不匹配

而绝大部分的Rust框架都被污染。

这也是我重写一个新框架的主要原因

# AEX 的核心目标

## 1. 让Rust的Web开发更加容易

一开始使用时，不用对Rust太了解就可以写出基本的Web处理  
不需要学一堆莫名其妙的函数的使用

## 2. 还原 HTTP 请求的本来面目

包括:

i. 请求如何进来

ii. 中间件如何执行

iii. 响应如何写出

全部 明确、可见、可理解

# 开发体验目标

让 Rust Web 编程变得更简单

1. 书写方式友好
2. API 接口直观
3. 行为符合直觉

不需要“框架思维负担”

你写的是 Web 逻辑，而不是给框架写扩展

框架是服务于你的业务逻辑的。

框架的任务是拆解Web逻辑，让你的Web逻辑处理起来简单，从而专注于你的业务本身。

## 性能

AEX 不是性能优先型框架，而是应用业务优先型框架。

当前性能没有达到顶级性能框架的水平，但是能于应用开发优先的框架，

Aex可以提供不错的早期性能。

随着Aex的成熟，未来会不断的优化，达到顶级的性能水平。

性能优化是所有成熟框架必走的道路，但是设计的错误是无法优化的。

所以早期的策略是优先确保设计是正确的。

## 基本设计原则

1. 核心接口保持精简，一致，不变
2. API 不随意增加
3. 执行路径简单明晰
4. 源码结构简单，容易理解，便加参与，扩展

性能优化过程中，这些设计理念与原则不会变化

## 适合谁？

AEX 适合什么样的开发者

1. 需要专注于业务开发
2. 希望Web的逻辑与业务逻辑分离
3. 希望快速写出原型Web服务器的



## 核心概念

Aex是一个非常简单的框架。它的核心概念就只有三个：

1. Router: 基于Trie树的高性能路由器
2. Executor: 请求处理与中间件都依赖于它
3. HTTPContext: 分请求数据与全局数据，HTTP数据的运输载体

## 扩展概念

基于核心概念，扩展出来的几个重要概念

1. Middleware: 一个有序的Executor数组列表，当其中一个返回false时，可以终止后续的Executor执行
2. Handler: 最后一个执行的Executor，也是一个HTTP请求最终处理，并返回的地方
3. WebSocket: 处理WebSocket的基础处理结构，通过WebSocket::to\_middleware生成中间件，放到Middleware中
4. Server: 实现基本的Http服务器启动功能，并实现与Router对接

# 宏的使用

宏的使用就是将参数进行切分，属于路由描述语言。

比如

```
get!("/path", handler, [mw1, mw2])
```

等价于参数列表

```
(  
  "GET",  
  "/path",  
  // handler是一个Executor类型，外套一个Arc类型  
  Arc<Executor>,  
  // mw1, mw2也都是Executor，组成一个Vec<Arc<Executor>>类型的数组[mw1, mw2]传给route  
  Some(Vec<Arc<Executor>>)  
)
```

## 宏的使用结构

```
route!(route, xxx!(path, handler[, [mw1, mw2, mw3 ...]]));
```

其中

1. route!就是Router的insert宏
2. xxx!就是接收http的xxx方法在handler进行处理，全部是小写的。包括: `get!` , `post!` , `put!` , `delete!` , `patch!` , `options!` , `head!` , `trace!` , `connect!`
3. all!表示接收所有http方法进行处理，类似于路径为"\*"

## 中间件及路由的执行顺序

中间件虽然在参数上位于handler之后，然后它的执行是先开handler的  
对于

```
route!(route, get!(path, handler, [mw1, mw2, mw3]))
```

它们的执行顺序是:

```
mw1 => mw2 => mw3 => handler
```

它在理论上满足HTTP的直线性处理方式（简直Web直线）：

```
请求开始 => 数据接收与存储/参数校验 => 身份校验 => 数据分析与处理 => 业务处理 => 数据发送 => 请求结束
```

## 对WebSocket的支持

随着互联网技术的发展以及对网页端实时交互的强烈需求，浏览器端的实际传输已经是刚需。

这也要求服务器端对WebSocket的很好的支持。

这也是Aex天然支持WebSocket的原因。

由于WebSocket的处理机制是基于HTTP协议的。

所以我们没有必要脱离HTTP框架。

因为Aex的中间件是正确的机制，所以WebSocket的基本处理放在中间件处理即可

## WebSocket，以一种特殊的中间件的方式去实现

在Aex框架内

1. WebSocket的基本协议处理，已经被Aex消化
2. 不用当成是“特殊协议”
3. 当成是特殊的（升级的）中间件即可

# WebSocket，为什么应该是中间件

## 1. 共享http头信息

很多框架因为没有共享http中间件，WebSocket无法共享Http的相关信息，成为典型的设计缺陷

## 2. 共享路由信息

可以将WebSocket的升级路径与它的HTTP路径共享，逻辑更加集中

## 3. 多路WebSocket处理

不同的路径下可以有不同的WebSocket处理逻辑，更好的HTTP协作

## 4. http头或者前期数据处理

WebSocket也需要很多前置中间件处理一些需要的http头信息或者前置数据



## WebSocket的支持

WebSocket中间件目前主要关心

1. on\_text ： 处理文本交互
2. on\_binary ： 处理二进制数据交互

这两种主流数据的交互。

实现WebSocket功能的过程很简单，主要分以下四步：

## 1. 编写WebSocket的

on\_text

on\_binary

处理函数。

要注意类型分别为 `TextHandler` 和 `BinaryHandler` 时的闭包参数类型的不同。

### 1. `TextHandler` 类型闭包参数是 `String` 类型

```
let text_handler: TextHandler = Arc::new(|ws: &WebSocket, ctx: &mut HTTPContext, text: String| {  
    (  
        async move {  
            // processing here  
            true  
        }  
    ).boxed()  
});
```

## 2. BinaryHandler 类型闭包参数是 Vec<u8> 类型

```
let binary_handler: BinaryHandler = Arc::new(
    |ws: &WebSocket, ctx: &mut HTTPContext, data: Vec<u8>| {
        (
            async move {
                // processing here
                true
            }
        ).boxed()
    }
);
```

## 2. 将处理函数放到WebSocket对象上

```
let ws = WebSocket {  
  on_binary: Some(binary_handler),  
  on_text: Some(text_handler),  
};
```

### 3. 生成Executor中间件

```
let ws_mw = WebSocket::to_middleware(ws);
```

### 4. 生成带有路径与http方法的宏处理参数

```
let ws_params = get!(  
    "/",  
    |ctx|  
        (  
            async move {  
                true  
            }  
        ).boxed(),  
    [ws_mw]  
);
```

## 5. 插入到路由的中间件中

```
let mut route = Router::new(NodeType::Static("root".into()));  
route!(route, ws_params);
```

## 6. 挂载到服务器上

```
let ip = "0.0.0.0";  
let port = 8080;  
let addr: SocketAddr = format!("{:}", ip, port).parse()?;  
let server = HTTPServer::new(addr, route);  
  
server.run().await?;
```

外部的main结构保持不变

# 为什么选择Aex？

## Aex vs Axum vs Actix

维度	AEX	Axum	Actix
请求模型	显式 <b>Executor</b> 顺序执行	Tower 洋葱模型	Actor + Service
抽象层级	极低，贴近 <b>HTTP</b>	高（Service / Layer / Extractor）	高（Actor / Context）
控制流	线性、可预测	隐式嵌套	消息驱动
中间件	Vec<Executor>	Layer Stack	Middleware / Actor

维度	AEX	Axum	Actix
WebSocket	HTTP → WS 同一 ctx	分离	分离
学习成本	低	中	高
调试难度	低	中偏高	高
可读性	顺序即逻辑	需 mental model	需 actor 思维
性能上限	高（未完全释放）	已优化	已优化
架构自由度	极高	受 Tower 约束	受 Actor 模型约束



## AI比较总结

- **AEX**：把 Rust Web 框架拉回工程现实，执行顺序可预测，HttpContext 明确，WebSocket 自然升级，学习成本低。
- **Axum**：Tower 洋葱模型，抽象层多，调试时需要 mental model。
- **Actix**：Actor + Service，消息驱动，高度抽象，学习成本高。