

Actix-Web

Actix是一个强大的Rust的actor系统, 在它之上是actix-web框架。这是你在工作中大多使用的东西。Actix-web给你提供了一个有趣且快速的Web开发框架。

我们称actix-web为小而务实的框架。对于所有的意图和目的来说, 这是一个有少许曲折的微框架。如果你已经是一个Rust程序员, 你可能会很快熟悉它, 但即使你是来自另一种编程语言, 你应该会发现actix-web很容易上手。

使用actix-web开发的应用程序将在本机可执行文件中包含HTTP服务器。你可以把它放在另一个像nginx这样的HTTP服务器上。即使完全不存在另一个HTTP服务器的情况下, actix-web也足以提供HTTP 1和HTTP 2支持以及SSL/TLS。这对于构建微服务分发非常有用。

最重要的是: actix-web可以稳定发布。

[Awesome-Actix](#)

何为Actix

Actix 指多种事情

Actix是一个强大的Rust的actor系统, 在它之上是actix-web框架。这是你在工作中大多使用的东西。Actix-web给你提供了一个有趣且快速的Web开发框架。

我们称actix-web为小而务实的框架。对于所有的意图和目的来说, 这是一个有少许曲折的微框架。如果你已经是一个Rust程序员, 你可能会很快熟悉它, 但即使你是来自另一种编程语言, 你应该会发现actix-web很容易上手。

使用actix-web开发的应用程序将在本机可执行文件中包含HTTP服务器。你可以把它放在另一个像nginx这样的HTTP服务器上。即使完全不存在另一个HTTP服务器的情况下, actix-web也足以提供HTTP 1和HTTP 2支持以及SSL/TLS。这对于构建微服务分发非常有用。

最重要的是: actix-web可以稳定发布。

安装

既然actix-web是一个Rust框架，你需要Rust来使用它。如果您还没有它，我们建议您使用rustup来管理您的Rust安装。该[官方Rust指南](#)有精彩的入门部分。

我们目前至少需要Rust 1.24，因此请确保您运行的rustup update 是最新且最好的Rust版本。特别是本指南将假定您实际运行Rust 1.26或更高版本。

安装 actix-web

感谢Rust的cargo包管理，您不需要明确安装 actix-web。只需要声明依赖就行了。对于想要使用actix-web的开发版本的情况，您可以直接依赖git存储库。

Release版本：

```
[dependencies]
actix-web = "0.7"
```

Development 版本：

```
[dependencies]
actix-web = { git = "https://github.com/actix/actix-web" }
```

深入

在这里有两条你可以选择的路径。你可以沿着指南或者如果你非常不耐烦，你可能想看看我们广泛的[示例库](#)并运行包含的示例。这里举例说明如何运行包含的basics示例：

```
git clone https://github.com/actix/examples
cd examples/basics
cargo run
```

开始

我们来编写第一个actix web应用程序！

Hello, world

首先创建一个新的基于二进制的Cargo项目并进入新目录：

```
cargo new hello-world  
cd hello-world
```

现在，确保actix-web的Cargo.toml 包含以下项目依赖关系：

```
[dependencies]  
actix-web = "0.7"
```

为了实现一个Web服务器，我们首先需要创建一个请求处理程序。请求处理函数接受一个HttpRequest实例作为其唯一参数，并返回一个可转换为HttpResponse的类型：

文件名: `src/main.rs`

```
extern crate actix_web;  
use actix_web::{server, App, HttpRequest};  
  
fn index(_req: &HttpRequest) -> &'static str {  
    "Hello world!"  
}
```

接下来，创建一个Application实例，并将请求处理程序与应用程序的resource一起注册在特定HTTP方法和路径，然后，应用程序实例可以用于HttpServer来侦听将传入的连接。服务器接受一个应该返回一个HttpHandler实例的函数。简单来说server::new可以使用了，它是HttpServer::new的简写：

```
fn main() {  
    server::new(|| App::new().resource("/", |r| r.f(index))  
        .bind("127.0.0.1:8088")  
        .unwrap()  
        .run();  
}
```

仅此而已！现在编译并运行该程序cargo run。去<http://localhost:8088> 看结果。

如果你想要在开发过程中重新编译后自动重新加载服务器。请查看[自动重新加载模式](#)。

应用

actix-web提供了用Rust构建Web服务器和应用程序的各种基础类型。它提供了路由，中间件，请求的预处理，响应的后处理，websocket协议处理，multipart流，等等。

所有actix web服务器都是围绕该App实例构建的。它用于为资源和中间件注册路由。它还存储同一应用程序中所有处理程序之间共享的应用程序状态。

应用程序充当所有路由的命名空间，即特定应用程序的所有路由具有相同的url路径前缀。应用程序前缀总是包含一个前导的“/”斜杠。如果提供的前缀不包含前导斜杠，则会自动插入。前缀应该由路径值组成。

对于具有前缀的应用程序/app，与任何请求路径中有/app，/app/或/app/test匹配；然而，路径/application不匹配。

```
fn index(req: HttpRequest) -> impl Responder {
    "Hello world!"
}

let app = App::new()
    .prefix("/app")
    .resource("/index.html", |r| r.method(Method::GET).f(index))
    .finish()
```

在此示例中，将创建具有/app前缀和index.html资源的应用。该资源可通过/app/index.html路径获得。

有关更多信息，请查看[URL Dispatch](#)部分。

但单服务器服务多个应用：

```
let server = server::new(|| {
    vec![
        App::new()
            .prefix("/app1")
            .resource("/", |r| r.f(|r| HttpResponse::Ok())),
        App::new()
            .prefix("/app2")
            .resource("/", |r| r.f(|r| HttpResponse::Ok())),
        App::new().resource("/", |r| r.f(|r| HttpResponse::Ok())),
    ]
});
```

所有/app1请求路由到第一个应用程序，/app2到第二个，所有其他到第三个。应用程序根据注册顺序进行匹配。如果具有更通用的前缀的应用程序在不通用的应用程序之前注册，它将有效地阻止较不通用的应用程序匹配。例如，如果App将前缀"/"注册为第一个应用程序，它将匹配所有传入的请求。

状态

同一应用程序内应用程序状态被所有路由和资源共享。当使用http actor时，状态可以HttpRequest::state()作为只读访问，但内部可变性RefCell可用于实现状态可变性。状态也可用于路由匹配谓词和中间件。

我们来编写一个使用共享状态的简单应用程序。我们打算将请求计数存储在状态中：

```
use actix_web::{http, App, HttpRequest};
use std::cell::Cell;

// This struct represents state
struct AppState {
    counter: Cell<usize>,
}

fn index(req: HttpRequest<AppState>) -> String {
    let count = req.state().counter.get() + 1; // <- get count
    req.state().counter.set(count); // <- store new count in state

    format!("Request number: {}", count) // <- response with count
}
```

应用程序需要通过初始化状态来初始化。

```
App::with_state(AppState { counter: Cell::new(0) })
    .resource("/", |r| r.method(http::Method::GET).f(index))
    .finish()
```

注意：http服务器接受应用程序工厂而不是应用程序实例。Http服务器为每个线程构造一个应用程序实例，因此应用程序状态必须多次构建。如果你想在不同线程之间共享状态，应该使用共享对象，例如Arc。应用程序状态并不需要是Send和Sync，但应用程序的工厂必须是Send+ Sync。

要启动以前的应用程序，请为其创建闭包：

```
server::new(|| {
    App::with_state(AppState { counter: Cell::new(0) })
        .resource("/", |r| r.method(http::Method::GET).f(index))
}).bind("127.0.0.1:8080")
    .unwrap()
    .run()
```

结合不同状态的应用程序

将多个应用程序与不同状态组合也是可能的。

`server::new`需要handler具有单一类型。

使用`App::boxed`方法可以轻松解决此限制，该方法可将App转换为boxed trait object。

```
struct State1;
struct State2;

fn main() {
    server::new(|| {
        vec![
            App::with_state(State1)
                .prefix("/app1")
                .resource("/", |r| r.f(|r| HttpResponse::Ok()))
                .boxed(),
            App::with_state(State2)
                .prefix("/app2")
                .resource("/", |r| r.f(|r| HttpResponse::Ok()))
                .boxed(),
        ]
    }).bind("127.0.0.1:8080")
        .unwrap()
        .run()
}
```

使用应用程序前缀来组合应用程序

该`App::prefix()`方法允许设置特定的应用程序前缀。此前缀表示将添加到所有资源模式的资源前缀通过资源配置。这可以用来帮助挂载一组路由在不同的地点比所包含的可调用作者的意图仍保持原样资源名称。

例如：

```
fn show_users(req: HttpRequest) -> HttpResponse {
    unimplemented!()
}

fn main() {
    App::new()
        .prefix("/users")
        .resource("/show", |r| r.f(show_users))
        .finish();
}
```

在上面的示例中，`show_users`路由将具有`/users/show`的有效路由模式，而不是`/show`，因为应用程序的前缀参数将预先添加到该模式。只有当URL路径为`/users/show`，并且`HttpRequest.url_for()`路由名称`show_users`调用该函数时，路由才会匹配，它将生成具有相同路径的URL。

应用程序谓词和虚拟主机

您可以将谓词看作一个接受请求对象引用并返回true或false的简单函数。形式上，谓词是实现[Predicate trait](#)的任何对象。Actix提供了几个谓词，你可以检查 API文档的[functions section](#)部分。

任何这些谓词都可以用于App::filter()方法。提供的谓词之一是Host，它可以根据请求的主机信息用作应用程序的过滤器。

```
fn main() {
    let server = server::new() || {
        vec![
            App::new()
                .filter(pred::Host("www.rust-lang.org"))
                .resource("/", |r| r.f(|r| HttpResponse::Ok())),
            App::new()
                .filter(pred::Host("users.rust-lang.org"))
                .resource("/", |r| r.f(|r| HttpResponse::Ok())),
            App::new().resource("/", |r| r.f(|r| HttpResponse::Ok())),
        ]
    });
    server.run();
}
```

服务器

该[HttpServer](#)类型负责服务的HTTP请求。

[HttpServer](#)接受应用程序工厂作为参数，并且应用程序工厂必须具有Send+ Sync边界。p 要绑定到特定的套接字地址，[bind\(\)](#) 必须使用，并且可能会多次调用它。绑定ssl套接字使用[bind_ssl\(\)](#)或[bind_tls\(\)](#)。启动http服务器，启动方法之一是：

- use [start\(\)](#) for a server

[HttpServer](#)是一位actix actor。它必须在正确配置的actix系统中初始化：

```
use actix_web::{server::HttpServer, App, HttpResponse};

fn main() {
    let sys = actix::System::new("guide");

    HttpServer::new(|| App::new().resource("/", |r| r.f(|_| 
HttpResponse::Ok())))
        .bind("127.0.0.1:59080")
        .unwrap()
        .start();

    let _ = sys.run();
}
```

可以使用该run()方法在单独的线程中启动服务器。在这种情况下，服务器会产生一个新线程并在其中创建一个新的actix系统。要停止此服务器，请发送[StopServer](#)消息。

[HttpServer](#)被实施为actix actor。可以通过消息传递系统与服务器进行通信。启动方法，例如[start\(\)](#)，返回启动的http服务器的地址。它接受几种消息：

- PauseServer - 暂停接受传入连接
- ResumeServer - 继续接受传入连接
- StopServer - 停止传入连接处理，停止所有workers并退出

```
use actix_web::{server, App, HttpResponse};
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let sys = actix::System::new("http-server");
        let addr = server::new(|| {
            App::new()
                .resource("/", |r| r.f(|_| HttpResponse::Ok())))
    });
}
```

```

    })
        .bind("127.0.0.1:0").expect("Can not bind to 127.0.0.1:0")
        .shutdown_timeout(60) // <- Set shutdown timeout to 60
seconds
    .start();
let _ = tx.send(addr);
let _ = sys.run();
});

let addr = rx.recv().unwrap();
let _ = addr.send(server::StopServer { graceful: true }).wait(); // <-
Send `StopServer` message to server.
}

```

多线程

`HttpServer`自动启动一些http worker， 默认情况下这个数量等于系统中逻辑CPU的数量。该数量可以用该`HttpServer::workers()`方法覆盖。

```

use actix_web::{server::HttpServer, App, HttpResponse};

fn main() {
    HttpServer::new(|| App::new().resource("/", |r| r.f(|_| 
HttpResponse::Ok())))
        .workers(4); // <- Start 4 workers
}

```

服务器为每个创建的worker创建一个单独的应用实例。应用程序状态不在线程之间共享。分享状态，可以使用Arc。

应用程序状态并不需要是Send和Sync，但是工厂必须是Send+ Sync。

SSL

有两种功能的ssl服务器：`tls`和`alpn`。该`tls`功能由native-tls集成，`alpn`由openssl。

```

[dependencies]
actix-web = { version = "0.7", features = ["alpn"] }

```

```

use actix_web::{server, App, HttpRequest, Responder};
use openssl::ssl::{SslAcceptor, SslFiletype, SslMethod};

fn index(req: HttpRequest) -> impl Responder {
    "Welcome!"
}

```

```

fn main() {
    // load ssl keys
    let mut builder =
SslAcceptor::mozilla_intermediate(SslMethod::tls()).unwrap();
    builder
        .set_private_key_file("key.pem", SslFiletype::PEM)
        .unwrap();
    builder.set_certificate_chain_file("cert.pem").unwrap();

    server::new(|| App::new().resource("/index.html", |r| r.f(index)))
        .bind_ssl("127.0.0.1:8080", builder)
        .unwrap()
        .run();
}

```

注意: HTTP / 2.0协议需要[tls alpn](#)。目前，只有openssl有alpn支持。完整示例，请查看[examples/tls](#).

要创建key.pem和cert.pem，请使用以下命令。**Fill in your own subject**

```

$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem \
  -days 365 -sha256 -subj
"/C=CN/ST=Fujian/L=Xiamen/O=TVlinux/OU=Org/CN=muro.lxd"

```

要删除密码，请将nopass.pem复制到key.pem

```
$ openssl rsa -in key.pem -out nopass.pem
```

Keep-Alive

Actix可以等待keep-alive的请求。

keep-alive连接行为由服务器设置定义。

- `75, Some(75), KeepAlive::Timeout(75)` - 75秒keep alive定时器。
- `None` or `KeepAlive::Disabled` - 禁用 *keep alive*.
- `KeepAlive::Tcp(75)` - 使用 `SO_KEEPALIVE` socket 选项.

```

use actix_web::{server, App, HttpResponse};

fn main() {
    server::new(|| App::new().resource("/", |r| r.f(|_| {
        HttpResponse::Ok()
            .keep_alive(75); // <- Set keep-alive to 75 seconds
    }))
}

```

```

    server::new(|| App::new().resource("/", |r| r.f(|_|

HttpResponses::Ok())))
    .keep_alive(server::KeepAlive::Tcp(75)); // <- Use `SO_KEEPALIVE`  

socket option.

    server::new(|| App::new().resource("/", |r| r.f(|_|

HttpResponses::Ok())))
    .keep_alive(None); // <- Disable keep-alive
}

```

如果选择第一个选项，则`keep alive`状态根据响应的`connection-type`计算。默认情况下 `HttpResponses::connection_type` 未定义。在这种情况下，`keep alive` 状态由请求的http版本定义。

`keep alive` 是 **关闭** 对于 `HTTP/1.0` 然而是 **打开** 对于 `HTTP/1.1` 和 `HTTP/2.0`。

`Connection type` 可以用 `HttpResponseBuilder::connection_type()` 方法改变。

```

use actix_web::{http, HttpRequest, HttpResponse};

fn index(req: HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        .connection_type(http::ConnectionType::Close) // <- Close  

connection
        .force_close() // <- Alternative  

method
        .finish()
}

```

优雅的关机

`HttpServer` 支持优雅的关机。收到停止信号后，workers 会有特定的时间完成服务请求。任何在超时后仍然活着的 workers（工作线程）都会被迫停止。默认情况下，关机超时设置为 30 秒。您可以使用 `HttpServer::shutdown_timeout()` 方法更改此参数。

您可以使用服务器地址向服务器发送停止消息，并指定是否要进行正常关机。`start()` 方法返回服务器的地址。

`HttpServer` 处理几个 OS 信号。所有操作系统都提供 `CTRL-C`，其他信号在 unix 系统上可用。

- `SIGINT` - 强制关闭工作线程
- `SIGTERM` - 优雅的停止工作线程
- `SIGQUIT` - 强制关闭 workers/ 工作线程

可以用 `HttpServer::disable_signals()` 方法禁用信号处理。

Handler

请求处理器可以是实现[Handler](#) trait 的任何对象。

请求处理分两个阶段进行。首先调用 handler 对象，返回实现[Responder](#) 特征的任何对象。然后，[respond_to\(\)](#) 在返回的对象上调用，将自身转换为 [AsyncResult](#) 或 [Error](#)。

默认情况下 Actix 提供对于一些标准类型的 [Responder](#) 实现，诸如 `&'static str`, `String` 等有关实现的完整列表，请查看 [Responder](#) 文档。

有效 Handler 程序的示例：

```
fn index(req: HttpRequest) -> &'static str {
    "Hello world!"
}
```

```
fn index(req: HttpRequest) -> String {
    "Hello world!".to_owned()
}
```

如果涉及更复杂的类型，您还可以更改签名返回 [impl Responder](#)。

```
fn index(req: HttpRequest) -> impl Responder {
    Bytes::from_static("Hello world!")
}
```

```
fn index(req: HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>>
{
    ...
}
```

[Handler](#) 特征是通用的通过 `S`，它定义了应用程序状态的类型。可以使用该 `HttpRequest::state()` 方法从 Handler 访问应用程序状态；但是，可以将状态作为只读引用进行访问。如果您需要对状态进行可变访问，则必须实现它。

注意：或者，Handler 可以可变地访问其自己的状态，因为该 `handle` 方法对 `self` 进行了可变引用。请注意，actix 会创建应用程序状态和处理程序的多个副本，这些副本对于每个线程都是唯一的。如果在多个线程中运行应用程序，actix 将创建与应用程序状态对象和处理程序对象的线程数相同数量的副本。

以下是存储已处理请求数的处理程序示例：

```
use actix_web::{App, HttpRequest, HttpResponse, dev::Handler};

struct MyHandler(usize);

impl<S> Handler<S> for MyHandler {
    type Result = HttpResponse;

    /// Handle request
    fn handle(&mut self, req: HttpRequest<S>) -> Self::Result {
        self.0 += 1;
        HttpResponse::Ok().into()
    }
}
```

尽管此处理程序将起作用，但self.0根据线程数和每个线程处理的请求数将有所不同。一个适当的实现将使用[Arc](#)和[AtomicUsize](#)。

```
use actix_web::{server, App, HttpRequest, HttpResponse, dev::Handler};
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};

struct MyHandler(Arc<AtomicUsize>);

impl<S> Handler<S> for MyHandler {
    type Result = HttpResponse;

    /// Handle request
    fn handle(&mut self, req: HttpRequest<S>) -> Self::Result {
        self.0.fetch_add(1, Ordering::Relaxed);
        HttpResponse::Ok().into()
    }
}

fn main() {
    let sys = actix::System::new("example");

    let inc = Arc::new(AtomicUsize::new(0));

    server::new(
        move || {
            let cloned = inc.clone();
            App::new()
                .resource("/", move |r| r.h(MyHandler(cloned)))
        }
    )
    .bind("127.0.0.1:8088").unwrap()
    .start();

    println!("Started http server: 127.0.0.1:8088");
    let _ = sys.run();
}
```

小心使用Mutex或等同步原语RwLock。该actix-web框架异步处理请求。通过阻止线程执行，所有并发请求处理进程都将阻塞。如果需要从多个线程共享或更新某些状态，请考虑使用actix actor系统。

自定义响应类型

要直接从处理函数返回自定义类型，该类型需要实现Responder特征。

让我们为序列化为响应的自定义类型创建application/json响应：

```
# extern crate actix;
# extern crate actix_web;
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use actix_web::{server, App, HttpRequest, HttpResponse, Error, Responder,
http};

#[derive(Serialize)]
struct MyObj {
    name: &'static str,
}

/// Responder
impl Responder for MyObj {
    type Item = HttpResponse;
    type Error = Error;

    fn respond_to<S>(self, req: &HttpRequest<S>) -> Result<HttpResponse, Error> {
        let body = serde_json::to_string(&self)?;

        // Create response and set content type
        Ok(HttpResponse::Ok()
            .content_type("application/json")
            .body(body))
    }
}

fn index(req: HttpRequest) -> impl Responder {
    MyObj { name: "user" }
}

fn main() {
    let sys = actix::System::new("example");

    server::new(
        || App::new()
            .resource("/", |r| r.method(http::Method::GET).f(index))
            .bind("127.0.0.1:8088").unwrap()
            .start());
}
```

```

    println!("Started http server: 127.0.0.1:8088");
    let _ = sys.run();
}

```

Async handlers

异步handlers

有两种不同类型的异步处理程序。响应对象可以异步生成，也可以更精确地生成任何实现`Responder`trait的类型。

在这种情况下， handler程序必须返回一个Future解析为Responder类型的对象，即：

```

use actix_web::*;

use bytes::Bytes;
use futures::stream::once;
use futures::future::{Future, result};

fn index(req: &HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>> {
    result(Ok(HttpResponse::Ok()
        .content_type("text/html")
        .body(format!("Hello!"))))
        .responder()
}

fn index2(req: &HttpRequest) -> Box<Future<Item=&'static str, Error=Error>> {
    result(Ok("Welcome!"))
        .responder()
}

fn main() {
    App::new()
        .resource("/async", |r| r.route().a(index))
        .resource("/", |r| r.route().a(index2))
        .finish();
}

```

或者可以异步生成响应主体。在这种情况下， body必须实现流特征`Stream<Item=Bytes, Error=Error>`，即：

```

use actix_web::*;
use bytes::Bytes;
use futures::stream::once;

fn index(req: &HttpRequest) -> HttpResponse {
    let body = once(Ok(Bytes::from_static(b"test")));
}

```

```

    HttpResponse::Ok()
        .content_type("application/json")
        .body(Body::Streaming(Box::new(body)))
}

fn main() {
    App::new()
        .resource("/async", |r| r.f(index))
        .finish();
}

```

两种方法都可以组合使用。 (即与流体的异步响应)

这是可能的返回Result，其中Result::Item类型可以是Future。在此示例中，index处理程序可以立即返回错误或返回解析为a的future HttpResponse。

```

use actix_web::*;
use bytes::Bytes;
use futures::stream::once;
use futures::future::{Future, result};

fn index(req: &HttpRequest) -> Result<Box<Future<Item=HttpResponse,
Error=Error>>, Error> {
    if is_error() {
        Err(error::ErrorBadRequest("bad request"))
    } else {
        Ok(Box::new(
            result(Ok(HttpResponse::Ok()
                .content_type("text/html")
                .body(format!("Hello!"))))))
    }
}

```

不同的返回类型(Either)

有时，您需要返回不同类型的响应。例如，您可以进行错误检查并返回错误，返回异步响应或任何需要两种不同类型的结果。

对于这种情况，可以使用Either类型。 Either允许将两种不同的响应者类型组合成一种类型。

```

use futures::future::{Future, result};
use actix_web::{Either, Error, HttpResponse};

type RegisterResult = Either<HttpResponse, Box<Future<Item=HttpResponse,
Error=Error>>>;

fn index(req: &HttpRequest) -> impl Responder {
    if is_a_variant() { // <- choose variant A

```

```
    Either::A(  
        HttpResponse::BadRequest().body("Bad data"))  
    } else {  
        Either::B(      // <- variant B  
            result(Ok(HttpResponse::Ok()  
                .content_type("text/html")  
                .body(format!("Hello!")))).responder()  
    }  
}
```

提取器

类型安全的信息提取

Actix提供类型安全请求信息提取功能。默认情况下，actix提供了几个提取器实现。

访问提取器

如何访问Extractor取决于您使用的是处理函数还是自定义Handler类型。

在Handler函数内

提取器可以被传递到一个处理函数作为函数参数 或通过调用`ExtractorType::<...>::extract(req)`功能的功能范围内访问。

```
// Option 1: passed as a parameter to a handler function
fn index((params, info): (Path<(String, String,>), Json<MyInfo>)) ->
HttpServletResponse {
    ...
}

// Option 2: accessed by calling extract() on the Extractor
use actix_web::FromRequest;

fn index(req: &HttpRequest) -> HttpResponse {
    let params = Path::<(String, String)>::extract(req);
    let info = Json::<MyInfo>::extract(req);

    ...
}
```

在自定义Handler类型中

与Handler函数一样，自定义Handler类型可以通过调用`ExtractorType :: <...> :: extract (&req)`函数来访问`Extractor`。无法将Extractor 作为参数传递给自定义Handler类型，因为自定义Handler类型必须遵循handle实现的Handler trait指定的函数签名。

```
struct MyHandler(String);

impl<S> Handler<S> for MyHandler {
    type Result = HttpResponse;

    /// Handle request
    fn handle(&self, req: &HttpRequest<S>) -> Self::Result {
        ...
    }
}
```

```

let params = Path::<(String, String)>::extract(req);
let info = Json::<MyInfo>::extract(req);

...
HttpResponse::Ok().into()
}
}

```

Path

[Path](#)提供可从Request的路径中提取的信息。您可以从路径反序列化任何变量段。

例如，对于为`/users/{userid}/{friend}`路径注册的资源，可以对两个段进行反序列化，`userid`以及`friend`。这些片段可以被提取到一个tuple，即`Path<(u32, String)>`或任何`Deserialize`从`serde` crate实现trait的结构中。

```

use actix_web::{App, Path, Result, http};

/// extract path info from "/users/{userid}/{friend}" url
/// {userid} -- deserializes to a u32
/// {friend} -- deserializes to a String
fn index(info: Path<(u32, String)>) -> Result<String> {
    Ok(format!("Welcome {}! {}", info.1, info.0))
}

fn main() {
    let app = App::new().resource(
        "/users/{userid}/{friend}", // <- define path
        parameters
            |r| r.method(http::Method::GET).with(index)); // <- use `with` extractor
}

```

记得！必须使用[Route::with\(\)](#)方法注册使用提取器的处理函数。

还可以将路径信息提取到`Deserialize`从`serde`实现特征的特定类型。这是一个使用`serde`而不是元组类型的等效示例。

```

#[macro_use] extern crate serde_derive;
use actix_web::{App, Path, Result, http};

#[derive(Deserialize)]
struct Info {
    userid: u32,
    friend: String,
}

```

```

/// extract path info using serde
fn index(info: Path<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.friend))
}

fn main() {
    let app = App::new().resource(
        "/users/{userid}/{friend}", // <- define path
parameters
        |r| r.method(http::Method::GET).with(index)); // <- use `with`-
extractor
}

```

Query

可以使用请求的查询完成相同的操作。的查询类型提供提取功能。在它下面使用`serde_urlencoded`箱子。

```

#[macro_use] extern crate serde_derive;
use actix_web::{App, Query, http};

#[derive(Deserialize)]
struct Info {
    username: String,
}

// this handler get called only if the request's query contains `username`-
field
fn index(info: Query<Info>) -> String {
    format!("Welcome {}!", info.username)
}

fn main() {
    let app = App::new().resource(
        "/index.html",
        |r| r.method(http::Method::GET).with(index)); // <- use `with`-
extractor
}

```

Json

`Json` Json允许将请求主体反序列化为结构。要从请求的正文中提取类型信息，该类型T必须实现`serde`的`Deserialize` trait。

```

#[macro_use] extern crate serde_derive;
use actix_web::{App, Json, Result, http};

#[derive(Deserialize)]
struct Info {
    username: String,
}

```

```

}

/// deserialize `Info` from request's body
fn index(info: Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}

fn main() {
    let app = App::new().resource(
        "/index.html",
        |r| r.method(http::Method::POST).with(index)); // <- use `with` extractor
}

```

一些提取器提供了一种配置提取过程的方法。Json提取器 `JsonConfig` 类型用于配置。使用时注册处理程序时 `Route::with()`，它将返回配置实例。在Json提取器的情况下，它返回一个 `JsonConfig`。您可以配置json有效内容的最大大小以及自定义错误处理函数。

以下示例将有效负载的大小限制为4kb，并使用自定义错误处理程序。

```

#[macro_use] extern crate serde_derive;
use actix_web::{App, Json, HttpResponse, Result, http, error};

#[derive(Deserialize)]
struct Info {
    username: String,
}

/// deserialize `Info` from request's body, max payload size is 4kb
fn index(info: Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}

fn main() {
    let app = App::new().resource(
        "/index.html", |r| {
            r.method(http::Method::POST)
                .with_config(index, |cfg| {
                    cfg.limit(4096) // <- change json extractor configuration
                    cfg.error_handler(|err, req| { // <- create custom error response
                        error::InternalError::from_response(
                            err, HttpResponse::Conflict().finish()).into()
                    })
                });
})
}

```

目前只支持url编码的表单。可以将URL编码的主体提取为特定类型。此类型必须实现serde crate中的Deserialize特征。

[FormConfig](#)允许配置提取过程。

```
#[macro_use] extern crate serde_derive;
use actix_web::{App, Form, Result};

#[derive(Deserialize)]
struct FormData {
    username: String,
}

/// extract form data using serde
/// this handler gets called only if the content type is *x-www-form-urlencoded*
/// and the content of the request could be deserialized to a `FormData` struct
fn index(form: Form<FormData>) -> Result<String> {
    Ok(format!("Welcome {}!", form.username))
}
# fn main() {}


```

多提取器

Actix为元素实现的tuples（最多10个元素）提供了提取器实现FromRequest。

例如，我们可以同时使用路径提取器和查询提取器。

```
#[macro_use] extern crate serde_derive;
use actix_web::{App, Query, Path, http};

#[derive(Deserialize)]
struct Info {
    username: String,
}

fn index((path, query): (Path<(u32, String)>, Query<Info>)) -> String {
    format!("Welcome {}!", query.username)
}

fn main() {
    let app = App::new().resource(
        "/users/{userid}/{friend}",
        // <- define path
        parameters
            |r| r.method(http::Method::GET).with(index)); // <- use `with` extractor
}
```

其他

Actix还提供了其他几种提取器：

- *State* - 如果需要访问应用程序状态。类似 `HttpRequest::state()`.
- *HttpRequest* - `HttpRequest` 本身是一个提取器，它返回self，以防您需要访问请求。
- *String* - 您可以将请求的有效负载转换为*String*.*Example*在doc字符串中可用。
- *bytes::Bytes* - 您可以将请求的有效负载转换为字节: `Bytes`.*Example*

错误

Actix使用自己的`actix_web::error::Error`类型和`actix_web::error::ResponseError`特征来处理Web处理程序的错误。

如果处理程序返回一个`Error`（指一般的Rust特征`std::error::Error`）`Result`也实现了`ResponseError`特征，actix会将该错误呈现为HTTP响应。`ResponseError`有一个名为`return`的函数`error_response()`返回`HttpResponse`：

```
pub trait ResponseError: Fail {
    fn error_response(&self) -> HttpResponse {
        HttpResponse::new.StatusCode::INTERNAL_SERVER_ERROR
    }
}
```

一个`Responder`强制将兼容`Result`转换的HTTP响应：

```
impl<T: Responder, E: Into<Error>> Responder for Result<T, E>
```

`Error`在上面的代码中是actix的错误定义，并且实现的任何错误`ResponseError`都可以自动转换为一个。

Actix-web提供`ResponseError`一些常见的非actix错误的实现。例如，如果处理程序以a响应`io::Error`，则该错误将转换为`HttpInternalServerError`：

```
use std::io;

fn index(req: &HttpRequest) -> io::Result<fs::NamedFile> {
    Ok(fs::NamedFile::open("static/index.html")?)
}
```

有关外部实现的完整列表，请参阅actix-web API文档`ResponseError`。

自定义错误响应的示例

以下是一个示例实现`ResponseError`：

```
use actix_web::*;

#[derive(Fail, Debug)]
#[fail(display="my error")]
struct MyError {
    name: &'static str
}
```

```
// Use default implementation for `error_response()` method
impl error::ResponseError for MyError {}

fn index(req: &HttpRequest) -> Result<&'static str, MyError> {
    Err(MyError{name: "test"})
}
```

ResponseError 有一个默认的实现 `error_response()`，它将呈现 500（内部服务器错误），这就是 `index` 上面执行处理程序时会发生的事情。

覆盖 `error_response()` 以产生更有用的结果：

```
#[macro_use] extern crate failure;
use actix_web::{App, HttpRequest, HttpResponse, http, error};

#[derive(Fail, Debug)]
enum MyError {
    #[fail(display="internal error")]
    InternalError,
    #[fail(display="bad request")]
    BadClientData,
    #[fail(display="timeout")]
    Timeout,
}

impl error::ResponseError for MyError {
    fn error_response(&self) -> HttpResponse {
        match *self {
            MyError::InternalError => HttpResponse::new(
                http::StatusCode::INTERNAL_SERVER_ERROR),
            MyError::BadClientData => HttpResponse::new(
                http::StatusCode::BAD_REQUEST),
            MyError::Timeout => HttpResponse::new(
                http::StatusCode::GATEWAY_TIMEOUT),
        }
    }
}

fn index(req: &HttpRequest) -> Result<&'static str, MyError> {
    Err(MyError::BadClientData)
}
```

Error helpers

Actix 提供了一组错误辅助函数，可用于从其他错误生成特定的 HTTP 错误代码。在这里，我们使用以下方法将 `MyError` 未实现 `ResponseError` 特性的转换为 400（错误请求） `map_err`：

```
# extern crate actix_web;
use actix_web::*;

#[derive(Debug)]
struct MyError {
    name: &'static str
}

fn index(req: &HttpRequest) -> Result<&'static str> {
    let result: Result<&'static str, MyError> = Err(MyError{name: "test"});
    Ok(result.map_err(|e| error::ErrorBadRequest(e.name())))
}
```

有关可用错误帮助程序的完整列表，请参阅[actix-web error](#)模块的API文档。

与failure的兼容性

Actix-web提供与[failure](#)库的自动兼容性，以便将错误派生[fail](#)自动转换为actix错误。请记住，这些错误将使用默认的500状态代码呈现，除非您还[error_response\(\)](#)为它们提供自己的实现。

Error logging

Actix在[WARN](#)日志级别记录所有错误。如果应用程序的日志级别设置为[DEBUG](#)并[RUST_BACKTRACE](#)启用，则还会记录回溯。这些可以配置环境变量：

```
>> RUST_BACKTRACE=1 RUST_LOG=actix_web=debug cargo run
```

该[Error](#)类型使用cause的错误回溯（如果可用）。如果基础故障不提供回溯，则构造新的回溯指向转换发生的点（而不是错误的起源）。

错误处理的推荐做法

考虑将应用程序产生的错误划分为两大类可能是有用的：那些旨在面向用户的错误和那些不是面向用户的错误。

前者的一个例子是我可能会使用failure来指定一个[UserError](#)枚举[ValidationError](#)以便在用户发送错误输入时返回：

```
#[macro_use] extern crate failure;
use actix_web::{HttpResponse, http, error};

#[derive(Fail, Debug)]
```

```

enum UserError {
    #[fail(display="Validation error on field: {}", field)]
    ValidationError {
        field: String,
    }
}

impl error::ResponseError for UserError {
    fn error_response(&self) -> HttpResponse {
        match *self {
            UserError::ValidationError { .. } => HttpResponse::new(
                http::StatusCode::BAD_REQUEST),
            _ => HttpResponse::new(http::StatusCode::INTERNAL_SERVER_ERROR),
        }
    }
}

```

这将完全按预期运行，因为定义的错误消息 `display` 是用明确的意图写入的，用户可以读取。

但是，发送错误消息对于所有错误都是不可取的 - 在服务器环境中发生许多故障，我们可能希望从用户隐藏这些特定信息。例如，如果数据库出现故障并且客户端库开始产生连接超时错误，或者HTML模板格式不正确以及呈现时出错。在这些情况下，可能最好将错误映射到适合用户使用的一般错误。

这是一个 `InternalServerError` 使用自定义消息将内部错误映射到面向用户的示例：

```

#[macro_use] extern crate failure;
use actix_web::{App, HttpRequest, HttpResponse, http, error, fs};

#[derive(Fail, Debug)]
enum UserError {
    #[fail(display="An internal error occurred. Please try again later.")]
    InternalError,
}

impl error::ResponseError for UserError {
    fn error_response(&self) -> HttpResponse {
        match *self {
            UserError::InternalError => HttpResponse::new(
                http::StatusCode::INTERNAL_SERVER_ERROR),
            _ => HttpResponse::new(http::StatusCode::INTERNAL_SERVER_ERROR),
        }
    }
}

fn index(_: &HttpRequest) -> Result<&'static str, UserError> {
    fs::NamedFile::open("static/index.html").map_err(|e|
        UserError::InternalError)?;
    Ok("success!")
}

```

通过将错误划分为面向用户和非面向错误的错误，我们可以确保我们不会意外地将用户暴露给他们不应该看到的应用程序内部错误。

路由分发

URLDispatch提供了一种Handler使用简单模式匹配语言将URL映射到代码的简单方法。如果其中一个模式与请求相关联的路径信息匹配，则调用特定的处理程序对象。

处理程序是一个特定对象，它实现Handler在应用程序中定义的特征，该特征接收请求并返回响应对象。Handler程序部分提供了更多信息。

资源配置

资源配置是向应用程序添加新资源的行为。资源具有名称，该名称充当用于URL生成的标识符。该名称还允许开发人员向现有资源添加路由。资源也有一个模式，用于匹配URL的PATH部分。它不匹配针对QUERY部（方案和端口之后的部分，例如`/foo/bar`在URL `http://localhost:8080/foo/bar?q=value`）。

`App::route()`方法提供注册路由的简单方法。此方法将单个路由添加到应用程序路由表。此方法接受path pattern, http method和handler function。`route()`对于相同的路径，可以多次调用该方法，在这种情况下，多个路由注册相同的资源路径。

```
use actix_web::{http::Method, App, HttpRequest, HttpResponse};

fn index(req: HttpRequest) -> HttpResponse {
    unimplemented!()
}

fn main() {
    App::new()
        .route("/user/{name}", Method::GET, index)
        .route("/user/{name}", Method::POST, index)
        .finish();
}
```

虽然`App :: route ()` 提供了注册路由的简单方法，但是为了访问完整的资源配置，必须使用不同的方法。该应用`App::resource()`方法将单个资源应用路由表。此方法接受路径模式 和资源配置功能。

```
use actix_web::{http::Method, App, HttpRequest, HttpResponse};

fn index(req: &HttpRequest) -> HttpResponse {
    unimplemented!()
}

fn main() {
    App::new()
        .resource("/prefix", |r| r.f(index))
        .resource("/user/{name}", |r| {
            r.method(Method::GET).f(|req| HttpResponse::Ok())
        })
}
```

```

        .finish();
    }
}

```

该配置函数有以下类型：

```
FnOnce(&mut Resource<_>) -> ()
```

该配置函数，可以设置一个名称和登记的具体路线。如果资源不包含任何路由或没有任何匹配的路由，则返回NOT FOUND http响应。

配置路由

资源包含一组路由。每个路由依次有一组谓词和一个处理程序。可以使用**Resource::route()**方法创建新路由，该方法返回对新Route实例的引用。默认情况下，路由不包含任何谓词，因此匹配所有请求，默认处理程序为**HttpNotFound**。

应用程序根据在资源注册和路由注册期间定义的路由标准来路由传入请求。资源按照路由注册的顺序匹配它包含的所有路由**Resource::route()**。

一个路径可以包含任意数量的断言，但只有一个处理器。

```

use actix_web::{pred, App, HttpResponse};

fn main() {
    App::new()
        .resource("/path", |resource| {
            resource
                .route()
                .filter(pred::Get())
                .filter(pred::Header("content-type", "text/plain"))
                .f(|req| HttpResponse::Ok())
        })
        .finish();
}

```

在此示例中，**HttpResponse::Ok()**为GET请求返回。如果请求包含**Content-Type**标头，则此标头的值为**text / plain**，并且path等于/path，第一个匹配路由的资源调用句柄。

如果资源无法匹配任何路由，则返回“NOT FOUND”响应。

ResourceHandler :: route () 返回一个 **Route**对象。路径可以使用类似构建器的模式进行配置。以下配置方法可用：

- **Route::filter()**注册一个新的谓词。可以为每个路由注册任意数量的谓词。
- **Route::f()**注册此路由的处理函数。只能注册一个处理程序。通常，处理程序注册是最后的配置操作。处理程序函数可以是函数或闭包，具有类型 **Fn(HttpRequest<S>) -> R + 'static**

- `Route::h()` 注册一个实现 Handler 特征的处理程序对象。这与 `f()` 方法类似 - 只能注册一个处理程序。处理程序注册是最后一次配置操作。
- `Route::a()` 为此路由注册异步处理函数。只能注册一个处理程序。处理程序注册是最后一次配置操作。处理程序函数可以是函数或闭包，具有类型 `Fn(HttpRequest<S>) -> Future<Item = HttpResponseMessage, Error = Error> + 'static'`

路由匹配

The main purpose of route configuration is to match (or not match) the request's `path` against a URL path pattern. `path` represents the path portion of the URL that was requested.

The way that `actix` does this is very simple. When a request enters the system, for each resource configuration declaration present in the system, `actix` checks the request's path against the pattern declared. This checking happens in the order that the routes were declared via `App::resource()` method. If resource can not be found, the `default resource` is used as the matched resource.

When a route configuration is declared, it may contain route predicate arguments. All route predicates associated with a route declaration must be `true` for the route configuration to be used for a given request during a check. If any predicate in the set of route predicate arguments provided to a route configuration returns `false` during a check, that route is skipped and route matching continues through the ordered set of routes.

If any route matches, the route matching process stops and the handler associated with the route is invoked. If no route matches after all route patterns are exhausted, a `NOT FOUND` response get returned.

Resource pattern syntax

The syntax of the pattern matching language used by `actix` in the pattern argument is straightforward.

The pattern used in route configuration may start with a slash character. If the pattern does not start with a slash character, an implicit slash will be prepended to it at matching time. For example, the following patterns are equivalent:

```
{foo}/bar/baz
```

and:

```
/ {foo}/bar/baz
```

A `variable part` (replacement marker) is specified in the form `{identifier}`, where this means "accept any characters up to the next slash character and use this as the name in the `HttpRequest.match_info()` object".

A replacement marker in a pattern matches the regular expression `[^{}]/[+]`.

A `match_info` is the `Params` object representing the dynamic parts extracted from a `URL` based on the routing pattern. It is available as `request.match_info`. For example, the following pattern defines one literal segment (`foo`)

and two replacement markers (baz, and bar):

```
foo/{baz}/{bar}
```

The above pattern will match these URLs, generating the following match information:

```
foo/1/2      -> Params {'baz':'1', 'bar':'2'}
foo/abc/def   -> Params {'baz':'abc', 'bar':'def'}
```

It will not match the following patterns however:

```
foo/1/2/      -> No match (trailing slash)
bar/abc/def   -> First segment literal mismatch
```

The match for a segment replacement marker in a segment will be done only up to the first non-alphanumeric character in the segment in the pattern. So, for instance, if this route pattern was used:

```
foo/{name}.html
```

The literal path */foo/biz.html* will match the above route pattern, and the match result will be `Params{'name': 'biz'}`. However, the literal path */foo/biz* will not match, because it does not contain a literal *.html* at the end of the segment represented by *{name}.html* (it only contains biz, not biz.html).

To capture both segments, two replacement markers can be used:

```
foo/{name}.{ext}
```

The literal path */foo/biz.html* will match the above route pattern, and the match result will be `Params{'name': 'biz', 'ext': 'html'}`. This occurs because there is a literal part of *.* (period) between the two replacement markers *{name}* and *{ext}*.

Replacement markers can optionally specify a regular expression which will be used to decide whether a path segment should match the marker. To specify that a replacement marker should match only a specific set of characters as defined by a regular expression, you must use a slightly extended form of replacement marker syntax. Within braces, the replacement marker name must be followed by a colon, then directly thereafter, the regular expression. The default regular expression associated with a replacement marker `[^/]+` matches one or more characters which are not a slash. For example, under the hood, the replacement marker `{foo}` can more verbosely be spelled as `{foo:[^/]+}`. You can change this to be an arbitrary regular expression to match an arbitrary sequence of characters, such as `{foo:\d+}` to match only digits.

Segments must contain at least one character in order to match a segment replacement marker. For example, for the URL `/abc/`:

- `/abc/{foo}` will not match.
- `/{foo}/` will match.

Note: path will be URL-unquoted and decoded into valid unicode string before matching pattern and values representing matched path segments will be URL-unquoted too.

So for instance, the following pattern:

```
foo/{bar}
```

When matching the following URL:

```
http://example.com/foo/La%20Pe%C3%B1a
```

The matchdict will look like so (the value is URL-decoded):

```
Params{'bar': 'La Pe\xf1a'}
```

Literal strings in the path segment should represent the decoded value of the path provided to actix. You don't want to use a URL-encoded value in the pattern. For example, rather than this:

```
/Foo%20Bar/{baz}
```

You'll want to use something like this:

```
/Foo Bar/{baz}
```

It is possible to get "tail match". For this purpose custom regex has to be used.

```
foo/{bar}/{tail:.*}
```

The above pattern will match these URLs, generating the following match information:

```
foo/1/2/          -> Params{'bar':'1', 'tail': '2/'}
foo/abc/def/a/b/c -> Params{'bar':u'abc', 'tail': 'def/a/b/c'}
```

Scoping Routes

Scoping helps you organize routes sharing common root paths. You can nest scopes within scopes.

Suppose that you want to organize paths to endpoints used to manage a "Project", consisting of "Tasks". Such paths may include:

- /project
- /project/{project_id}
- /project/{project_id}/task
- /project/{project_id}/task/{task_id}

A scoped layout of these paths would appear as follows

```
< include-example example="url-dispatch" file="scope.rs" section="scope" >
```

A **scoped** path can contain variable path segments as resources. Consistent with unscoped paths.

You can get variable path segments from `HttpRequest::match_info()`. `Path` extractor also is able to extract scope level variable segments.

Match information

All values representing matched path segments are available in `HttpRequest::match_info`. Specific values can be retrieved with `Params::get()`.

Any matched parameter can be deserialized into a specific type if the type implements the `FromParam` trait. For example most standard integer types the trait, i.e.:

```
< include-example example="url-dispatch" file="minfo.rs" section="minfo" >
```

For this example for path '/a/1/2/', values v1 and v2 will resolve to "1" and "2".

It is possible to create a `PathBuf` from a tail path parameter. The returned `PathBuf` is percent-decoded. If a segment is equal to "...", the previous segment (if any) is skipped.

For security purposes, if a segment meets any of the following conditions, an `Err` is returned indicating the condition met:

- Decoded segment starts with any of: `.` (except `..`), `*`
- Decoded segment ends with any of: `:`, `>`, `<`
- Decoded segment contains any of: `/`
- On Windows, decoded segment contains any of: `\"`
- Percent-encoding results in invalid UTF8.

As a result of these conditions, a `PathBuf` parsed from request path parameter is safe to interpolate within, or use as a suffix of, a path without additional checks.

```
< include-example example="url-dispatch" file="pbuff.rs" section="pbuff" >
```

List of `FromParam` implementations can be found in [api docs](#)

Path information extractor

Actix provides functionality for type safe path information extraction. *Path* extracts information, destination type could be defined in several different forms. Simplest approach is to use *tuple* type. Each element in tuple must correspond to one element from path pattern. i.e. you can match path pattern `/{{id}}/{{username}}` against `Path<(u32, String)>` type, but `Path<(String, String, String)>` type will always fail.

```
< include-example example="url-dispatch" file="path.rs" section="path" >
```

It also possible to extract path pattern information to a struct. In this case, this struct must implement `*serde's *Deserialize` trait.

```
< include-example example="url-dispatch" file="path2.rs" section="path" >
```

Query provides similar functionality for request query parameters.

Generating resource URLs

Use the `HttpRequest.url_for()` method to generate URLs based on resource patterns. For example, if you've configured a resource with the name "foo" and the pattern "{a}/{b}/{c}", you might do this:

```
< include-example example="url-dispatch" file="urls.rs" section="url" >
```

This would return something like the string `http://example.com/test/1/2/3` (at least if the current protocol and hostname implied `http://example.com`) `url_for()` method returns *Url object* so you can modify this url (add query parameters, anchor, etc). `url_for()` could be called only for *named* resources otherwise error get returned.

External resources

Resources that are valid URLs, can be registered as external resources. They are useful for URL generation purposes only and are never considered for matching at request time.

```
< include-example example="url-dispatch" file="url_ext.rs" section="ext" >
```

Path normalization and redirecting to slash-appended routes

By normalizing it means:

- Add a trailing slash to the path.
- Double slashes are replaced by one.

The handler returns as soon as it finds a path that resolves correctly. The order if all enable is 1) merge, 2) both merge and append and 3) append. If the path resolves with at least one of those conditions, it will redirect to the new path.

If `append` is `true`, append slash when needed. If a resource is defined with trailing slash and the request doesn't have one, it will be appended automatically.

If `merge` is `true`, merge multiple consecutive slashes in the path into one.

This handler designed to be used as a handler for application's *default resource*.

```
< include-example example="url-dispatch" file="norm.rs" section="norm" >
```

In this example `/resource`, `//resource//` will be redirected to `/resource/`.

In this example, the path normalization handler is registered for all methods, but you should not rely on this mechanism to redirect `POST` requests. The redirect of the slash-appending `Not Found` will turn a `POST` request into a GET, losing any `POST` data in the original request.

It is possible to register path normalization only for `GET` requests only:

```
< include-example example="url-dispatch" file="norm2.rs" section="norm" >
```

Using an Application Prefix to Compose Applications

The `App::prefix()` method allows to set a specific application prefix. This prefix represents a resource prefix that will be prepended to all resource patterns added by the resource configuration. This can be used to help mount a set of routes at a different location than the included callable's author intended while still maintaining the same resource names.

For example:

```
< include-example example="url-dispatch" file="prefix.rs" section="prefix" >
```

In the above example, the `show_users` route will have an effective route pattern of `/users/show` instead of `/show` because the application's prefix argument will be prepended to the pattern. The route will then only match if the URL path is `/users/show`, and when the `HttpRequest.url_for()` function is called with the route name `show_users`, it will generate a URL with that same path.

Custom route predicates

You can think of a predicate as a simple function that accepts a `request` object reference and returns `true` or `false`. Formally, a predicate is any object that implements the `Predicate` trait. Actix provides several predicates, you can check [functions section](#) of api docs.

Here is a simple predicate that check that a request contains a specific `header`:

```
< include-example example="url-dispatch" file="pred.rs" section="pred" >
```

In this example, `index` handler will be called only if request contains `CONTENT-TYPE` header.

Predicates have access to the application's state via `HttpRequest::state()`. Also predicates can store extra information in [request extensions](#).

Modifying predicate values

You can invert the meaning of any predicate value by wrapping it in a `Not` predicate. For example, if you want to return "METHOD NOT ALLOWED" response for all methods except "GET":

```
< include-example example="url-dispatch" file="pred2.rs" section="pred" >
```

The `Any` predicate accepts a list of predicates and matches if any of the supplied predicates match. i.e:

```
pred::Any(pred::Get()).or(pred::Post())
```

The `All` predicate accepts a list of predicates and matches if all of the supplied predicates match. i.e:

```
pred::All(pred::Get()).and(pred::Header("content-type", "plain/text"))
```

Changing the default Not Found response

If the path pattern can not be found in the routing table or a resource can not find matching route, the default resource is used. The default response is *NOT FOUND*. It is possible to override the *NOT FOUND* response with `App::default_resource()`. This method accepts a *configuration function* same as normal resource configuration with `App::resource()` method.

```
< include-example example="url-dispatch" file="dhandler.rs" section="default" >
```

请求

Actix自动解压缩有效负载。支持以下编解码器：

- Brotli
- Gzip
- Deflate
- Identity

如果请求标头包含Content-Encoding标头，则根据标头值解压缩请求有效负载。不支持多个编解码器，即：Content-Encoding: br, gzip。

JSON请求

json正文反序列化有几种选择。

第一种选择是使用Json提取器。首先，定义一个Json<T>作为参数接受的处理函数，然后使用该.with()方法注册此处理程序。通过使用serde_json::Value作为类型，也可以接受任意有效的json对象T。

```
#[macro_use] extern crate serde_derive;
use actix_web::{App, Json, Result, http};

#[derive(Deserialize)]
struct Info {
    username: String,
}

/// extract `Info` using serde
fn index(info: Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}

fn main() {
    let app = App::new().resource(
        "/index.html",
        |r| r.method(http::Method::POST).with(index)); // <- use `with`
    extractor
}
```

另一种选择是使用HttpRequest::json()。此方法返回一个JsonBody对象，该对象解析为反序列化的值。

```
#[derive(Debug, Serialize, Deserialize)]
struct MyObj {
    name: String,
    number: i32,
}
```

```
fn index(req: &HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>>
{
    req.json().from_err()
        .and_then(|val: MyObj| {
            println!("model: {:?}", val);
            Ok(HttpResponse::Ok().json(val)) // <- send response
        })
        .responder()
}
```

您也可以手动将有效负载加载到内存中，然后对其进行反序列化。

在下面的示例中，我们将反序列化MyObj结构。我们需要先加载请求体，然后将json反序列化为一个对象。

```
extern crate serde_json;
use futures::{Future, Stream};

#[derive(Serialize, Deserialize)]
struct MyObj {name: String, number: i32}

fn index(req: &HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>>
{
    // `concat2` will asynchronously read each chunk of the request body
    // and
    // return a single, concatenated, chunk
    req.concat2()
        // `Future::from_err` acts like `?` in that it coerces the error
        // type from
        // the future into the final error type
        .from_err()
        // `Future::and_then` can be used to merge an asynchronous workflow
        // with a
        // synchronous workflow
        .and_then(|body| {
            let obj = serde_json::from_slice(&body)?;
            Ok(HttpResponse::Ok().json(obj))
        })
        .responder()
}
```

[示例目录](#)中提供了这两个选项的完整示例。

分块传输编码

Actix自动解码分块编码。[HttpRequest::payload\(\)](#)已经包含解码的字节流。如果使用所支持的压缩解码器之一(br, gzip, deflate)压缩请求有效负载，则解压缩字节流。

Multipart

Actix提供multipart stream支持。[Multipart](#)实现为[multipart items](#)。Each item可以是 Field或嵌套的 Multipart流。[HttpResponse::multipart\(\)](#)返回当前请求的Multipart流。

下面演示了一个简单表单的Multipart流处理：

```
use actix_web::*;

fn index(req: &HttpRequest) -> Box<Future<...>> {
    // get multipart and iterate over multipart items
    req.multipart()
        .and_then(|item| {
            match item {
                multipart::MultipartItem::Field(field) => {
                    println!("==== FIELD === {:#?} {:#?}", field.headers(),
                            field.content_type());
                    Either::A(
                        field.map(|chunk| {
                            println!("-- CHUNK: \n{}", std::str::from_utf8(&chunk).unwrap());
                            .fold((), |_, _| result(Ok(())))
                        }),
                    )
                },
                multipart::MultipartItem::Nested(mp) => {
                    Either::B(result(Ok(())))
                }
            }
        })
}
```

示例目录中提供了完整示例。

Urlencoded

Actix为[application / x-www-form-urlencoded](#)编码体提供支持。[HttpResponse::urlencoded\(\)](#)返回[UrlEncoded](#) future，它将解析为反序列化的实例。实例的类型必须实现[serde](#)的 [Deserialize](#)特征。

在几种情况下，[UrlEncoded](#)的future可以解决为错误：

- 内容类型不是 [application/x-www-form-urlencoded](#)
- 传输编码是 [chunked](#).
- content-length大于256k
- payload以错误终止。

```
#[macro_use] extern crate serde_derive;
use actix_web::*;


```

```
use futures::future::{Future, ok};

#[derive(Deserialize)]
struct FormData {
    username: String,
}

fn index(req: &HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>>
{
    req.urlencoded:<FormData>() // <- get UrlEncoded future
        .from_err()
        .and_then(|data| {           // <- serialized instance
            println!("USERNAME: {:?}", data.username);
            ok(HttpResponse::Ok().into())
        })
        .responder()
}
# fn main() {}
```

Streaming 请求

`HttpRequest` 是一个`Bytes`对象流。它可用于读取请求主体有效负载。

在下面的示例中，我们按块读取并打印请求有效负载块：

```
use actix_web::*;
use futures::{Future, Stream};

fn index(req: &HttpRequest) -> Box<Future<Item=HttpResponse, Error=Error>>
{
    req
        .payload()
        .from_err()
        .fold((), |_, chunk| {
            println!("Chunk: {:?}", chunk);
            result::<_, error::PayloadError>(Ok(()))
        })
        .map(|_| HttpResponse::Ok().finish())
        .responder()
}
```

响应

类似构建器的模式用于构造实例[HttpResponse](#)。[HttpResponse](#)提供了几个返回[HttpResponseBuilder](#)实例的方法，它实现了构建响应的各种便捷方法。

检查[文档](#)中的类型说明。

方法[.body](#), [.finish](#)和[.json](#)最终确定响应创建并返回构造的[HttpResponse](#)实例。如果多次在同一构建器实例上调用此方法，则构建器将发生混乱。

```
use actix_web::{HttpRequest, HttpResponse, http::ContentEncoding};

fn index(req: &HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        .content_encoding(ContentEncoding::Br)
        .content_type("plain/text")
        .header("X-Hdr", "sample")
        .body("data")
}
```

内容编码

Actix自动压缩有效负载。支持以下编解码器：

- Brotli
- Gzip
- Deflate
- Identity

响应有效负载基于content_encoding参数进行压缩。默认情况下，[ContentEncoding::Auto](#)使用。如果[ContentEncoding::Auto](#)选择，则压缩取决于请求的[Accept-Encoding](#)标头。

[ContentEncoding::Identity](#)可用于禁用压缩。如果选择了其他内容编码，则对该编解码器强制执行压缩。

例如，要启用[brotli](#)使用[ContentEncoding::Br](#):

```
use actix_web::{HttpRequest, HttpResponse, http::ContentEncoding};

fn index(req: HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        .content_encoding(ContentEncoding::Br)
        .body("data")
}
```

在这种情况下，我们通过将内容编码设置为一个Identity值来显式禁用内容压缩：

```
use actix_web::{HttpRequest, HttpResponse, http::ContentEncoding};

fn index(req: HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        // v- disable compression
        .content_encoding(ContentEncoding::Identity)
        .body("data")
}
```

此外，可以在应用程序级别设置默认内容编码，默认情况下ContentEncoding::Auto使用，这意味着自动内容压缩协商。

```
use actix_web::{App, HttpRequest, HttpResponse, http::ContentEncoding};

fn index(req: HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        .body("data")
}

fn main() {
    let app = App::new()
        // v- disable compression for all routes
        .default_encoding(ContentEncoding::Identity)
        .resource("/index.html", |r| r.with(index));
}
```

JSON响应

该Json类型允许使用格式良好的JSON数据进行响应：只返回Json类型的值哪个T是要序列化为JSON的结构的类型。该类型T必须实现serde的Serialize特征。

```
# extern crate actix_web;
#[macro_use] extern crate serde_derive;
use actix_web::{App, HttpRequest, Json, http::Method};

#[derive(Serialize)]
struct MyObj {
    name: String,
}

fn index(req: &HttpRequest) -> Result<Json<MyObj>> {
    Ok(Json(MyObj{name: req.match_info().query("name")?}))
}
```

```
fn main() {
    App::new()
        .resource(r"/a/{name}", |r| r.method(Method::GET).f(index))
        .finish();
}
```

分块传输编码

可以启用响应的分块编码`HttpResponseBuilder::chunked()`。这生效仅供`Body::Streaming(BodyStream)`或`Body::StreamingContext`机构。如果启用了响应有效负载压缩并使用了流体，则会自动启用分块编码。

禁止为HTTP / 2.0响应启用分块编码。

```
use actix_web::*;
use bytes::Bytes;
use futures::stream::once;

fn index(req: HttpRequest) -> HttpResponse {
    HttpResponse::Ok()
        .chunked()

        .body(Body::Streaming(Box::new(once(Ok(Bytes::from_static(b"data"))))))
}
```

测试

每个应用程序都应该经过充分测 Actix提供了执行单元和集成测试的工具.

单元测试

对于单元测试，actix提供了一个请求构建器类型和一个简单的处理程序运行器。[TestRequest](#) 实现类似构建器的模式。您可以[HttpRequest](#)使用[finish\(\)](#)或生成实例，也可以使用[run\(\)](#)或运行处理程序[run_async\(\)](#)。

```
use actix_web::{http, test, HttpRequest, HttpResponse, HttpMessage};

fn index(req: &HttpRequest) -> HttpResponse {
    if let Some(hdr) = req.headers().get(http::header::CONTENT_TYPE) {
        if let Ok(s) = hdr.to_str() {
            return HttpResponse::Ok().into()
        }
    }
    HttpResponse::BadRequest().into()
}

fn main() {
    let resp = test::TestRequest::with_header("content-type",
"text/plain")
        .run(index)
        .unwrap();
    assert_eq!(resp.status(), http::StatusCode::OK);

    let resp = test::TestRequest::default()
        .run(index)
        .unwrap();
    assert_eq!(resp.status(), http::StatusCode::BAD_REQUEST);
}
```

集成测试

有几种方法可用于测试您的应用程序。Actix提供 [TestServer](#)，可用于在真实的http服务器中使用特定处理程序运行应用程序。

[TestServer::get\(\)](#), [TestServer::post\(\)](#)和[TestServer::client\(\)](#) 方法可用于向测试服务器发送请求。

[TestServer](#)可以将简单表单配置为使用处理程序。 [TestServer::new](#)方法接受配置函数，此函数的唯一参数是测试应用程序实例。

有关更多信息，请查看[api文档](#)。

```
use actix_web::{HttpRequest, HttpMessage};
use actix_web::test::TestServer;
use std::str;

fn index(req: HttpRequest) -> &'static str {
    "Hello world!"
}

fn main() {
    // start new test server
    let mut srv = TestServer::new(|app| app.handler(index));

    let request = srv.get().finish().unwrap();
    let response = srv.execute(request.send()).unwrap();
    assert!(response.status().is_success());

    let bytes = srv.execute(response.body()).unwrap();
    let body = str::from_utf8(&bytes).unwrap();
    assert_eq!(body, "Hello world!");
}
```

另一种选择是使用应用程序工厂。在这种情况下，您需要以与实际http服务器配置相同的方式传递工厂函数。

```
use actix_web::{http, test, App, HttpRequest, HttpResponse};

fn index(req: &HttpRequest) -> HttpResponse {
    HttpResponse::Ok().into()
}

/// This function get called by http server.
fn create_app() -> App {
    App::new()
        .resource("/test", |r| r.h(index))
}

fn main() {
    let mut srv = test::TestServer::with_factory(create_app);

    let request = srv.client(
        http::Method::GET, "/test").finish().unwrap();
    let response = srv.execute(request.send()).unwrap();

    assert!(response.status().is_success());
}
```

如果需要更复杂的应用程序配置，请使用该[TestServer::build_with_state\(\)](#)方法。例如，您可能需要初始化应用程序状态或启动[SyncActor diesel](#)进程。此方法接受构造应用程序状态的闭包，并在配置actix系统时运行。因此，您可以初始化任何其他actor。

```

#[test]
fn test() {
    let srv = TestServer::build_with_state(|| {
        // we can start diesel actors
        let addr = SyncArbiter::start(3, || {
            DbExecutor(SqliteConnection::establish("test.db").unwrap())
        });
        // then we can construct custom state, or it could be `()``MyState{addr: addr}
    });

    // register server handlers and start test server
    .start(|app| {
        app.resource(
            "/{username}/index.html", |r| r.with(
                |p: Path<PPParam>| format!("Welcome {}!", p.username)));
    });

    // now we can run our test code
);

```

流响应测试

如果您需要测试流，将[ClientResponse](#)转换为future并执行它就足够了。例如，测试[Server Sent Events](#)。

```

extern crate bytes;
extern crate futures;
extern crate actix_web;

use bytes::Bytes;
use futures::stream::poll_fn;
use futures::{Async, Poll, Stream};

use actix_web::{HttpRequest, HttpResponse, Error};
use actix_web::http::{ContentEncoding, StatusCode};
use actix_web::test::TestServer;

fn sse(_req: HttpRequest) -> HttpResponse {
    let mut counter = 5usize;
    // yields `data: N` where N in [5; 1]
    let server_events = poll_fn(move || -> Poll<Option<Bytes>, Error> {
        if counter == 0 {
            return Ok(Async::NotReady);
        }
        let payload = format!("data: {}\n\n", counter);
        counter -= 1;
        Ok(Async::Ready(Some(Bytes::from(payload))))
    });
}

```

```
HttpResponse::build(StatusCode::OK)
    .content_encoding(ContentEncoding::Identity)
    .content_type("text/event-stream")
    .streaming(server_events)
}

fn main() {
    // start new test server
    let mut srv = TestServer::new(|app| app.handler(sse));

    // request stream
    let request = srv.get().finish().unwrap();
    let response = srv.execute(request.send()).unwrap();
    assert!(response.status().is_success());

    // convert ClientResponse to future, start read body and wait first
    chunk
    let (bytes, response) = srv.execute(response.into_future()).unwrap();
    assert_eq!(bytes.unwrap(), Bytes::from("data: 5\n\n"));

    // next chunk
    let (bytes, _) = srv.execute(response.into_future()).unwrap();
    assert_eq!(bytes.unwrap(), Bytes::from("data: 4\n\n"));
}
```

WebSocket服务器测试

它可以注册一个处理程序与TestApp::handler()，从而启动一个网络套接字连接。TestServer提供了ws()连接到websocket服务器并返回ws reader和writer对象的方法。TestServer还提供了一种execute()方法，该方法将未来的对象运行完成并返回未来计算的结果。

以下示例演示如何测试websocket处理程序：

```
use actix_web::*;
use futures::Stream;

struct Ws; // <- WebSocket actor

impl Actor for Ws {
    type Context = ws::WebsocketContext<Self>;
}

impl StreamHandler<ws::Message, ws::ProtocolError> for Ws {
    fn handle(&mut self, msg: ws::Message, ctx: &mut Self::Context) {
        match msg {
            ws::Message::Text(text) => ctx.text(text),
            _ => (),
        }
    }
}
```

```
    }

}

fn main() {
    let mut srv = test::TestServer::new(
        |app| app.handler(|req| ws::start(req, Ws)));
    let (reader, mut writer) = srv.ws().unwrap();
    writer.text("text");

    let (item, reader) = srv.execute(reader.into_future()).unwrap();
    assert_eq!(item, Some(ws::Message::Text("text".to_owned())));
}
```

中间件

Actix的中间件系统允许我们为请求/响应处理添加其他行为。中间件可以挂接到传入的请求进程，使我们能够修改请求以及暂停请求处理以及早返回响应。

中间件也可以挂钩响应处理。

通常，中间件涉及以下操作：

- 预处理请求
- 后处理响应
- 修改应用程序状态
- 访问外部服务 (redis, logging, sessions)

中间件在每个应用程序中注册，并以与注册相同的顺序执行。通常，中间件是实现*Middleware trait*的类型。此特征中的每个方法都有一个默认实现。每个方法都可以立即返回结果或未来的对象。

以下演示使用中间件添加请求和响应标头：

```
use http::{header, HttpTryFrom};
use actix_web::{App, HttpRequest, HttpResponse, Result};
use actix_web::middleware::{Middleware, Started, Response};

struct Headers; // <- Our middleware

/// Middleware implementation, middlewares are generic over application
state,
/// so you can access state with `HttpRequest::state()` method.
impl<S> Middleware<S> for Headers {
    /// Method is called when request is ready. It may return
    /// future, which should resolve before next middleware get called.
    fn start(&self, req: &HttpRequest<S>) -> Result<Started> {
        Ok(Started::Done)
    }

    /// Method is called when handler returns response,
    /// but before sending http message to peer.
    fn response(&self, req: &HttpRequest<S>, mut resp: HttpResponse)
        -> Result<Response>
    {
        resp.headers_mut().insert(
            header::HeaderName::try_from("X-VERSION").unwrap(),
            header::HeaderValue::from_static("0.2"));
        Ok(Response::Done(resp))
    }
}

fn main() {
    App::new()
```

```
// Register middleware, this method can be called multiple times
.middleware(Headers)
.resource("/", |r| r.f(|_| HttpResponse::Ok()));
}
```

Actix提供了一些有用的中间件，例如日志记录，用户会话等。

Logging

日志记录作为中间件实现。将日志记录中间件注册为应用程序的第一个中间件是很常见的。必须为每个应用程序注册日志记录中间件。

该Logger中间件使用标准箱日志记录信息。您应该为actix_web包启用logger以查看访问日志（env_logger或类似内容）。

用法

Logger使用指定的中间件创建format。Logger可以使用default方法创建默认值，它使用默认格式：

```
%a %t "%r" %s %b "%{Referer}i" "%{User-Agent}i" %T
```

```
extern crate env_logger;
use actix_web::App;
use actix_web::middleware::Logger;

fn main() {
    std::env::set_var("RUST_LOG", "actix_web=info");
    env_logger::init();

    App::new()
        .middleware(Logger::default())
        .middleware(Logger::new("%a %{User-Agent}i"))
        .finish();
}
```

以下是默认日志记录格式的示例：

```
INFO:actix_web::middleware::logger: 127.0.0.1:59934 [02/Dec/2017:00:21:43
-0800] "GET / HTTP/1.1" 302 0 "-" "curl/7.54.0" 0.000397
INFO:actix_web::middleware::logger: 127.0.0.1:59947 [02/Dec/2017:00:22:40
-0800] "GET /index.html HTTP/1.1" 200 0 "-" "Mozilla/5.0 (Macintosh; Intel
Mac OS X 10.13; rv:57.0) Gecko/20100101 Firefox/57.0" 0.000646
```

格式

%% 百分号

%a 远程IP地址（如果使用反向代理，则为代理的IP地址）

%t 请求开始处理的时间

%P 为请求提供服务的子进程ID

%r 第一行请求

%s 响应状态代码

%b 响应大小（以字节为单位），包括HTTP头

%T 服务请求的时间，以秒为单位，浮动分数为.06f格式

%D 服务请求所花费的时间，以毫秒为单位

%{FOO}i request.headers['FOO']

%{FOO}o response.headers['FOO']

%{FOO}e os.environ['FOO']

默认headers

要设置默认响应标头，`DefaultHeaders`可以使用中间件。所述 `DefaultHeaders` 中间件不设置标题如果响应头已经包含指定的报头。

```
use actix_web::{http, middleware, App, HttpResponse};

fn main() {
    let app = App::new()
        .middleware(
            middleware::DefaultHeaders::new()
                .header("X-Version", "0.2")
        )
        .resource("/test", |r| {
            r.method(http::Method::GET).f(|req| HttpResponse::Ok());
            r.method(http::Method::HEAD).f(|req|
                HttpResponse::MethodNotAllowed());
        })
        .finish();
}
```

User sessions

Actix为会话管理提供通用解决方案。所述的 `sessionStorage` 中间件可以与不同的后端类型被用于存储在不同的后端会话数据。

默认情况下，仅实现cookie会话后端。可以添加其他后端实现。

CookieSessionBackend 使用 cookie 作为会话存储。CookieSessionBackend 创建仅限于存储少于 4000 字节数据的会话，因为有效负载必须适合单个 cookie。如果会话包含超过 4000 个字节，则会生成内部服务器错误。

Cookie 可能具有签名或私有的安全策略。每个都有一个相应的 CookieSessionBackend 构造函数。

一个签署的 cookie 可以被查看但不能由客户端修改。甲私人 cookie 可既不由客户机观看也不修改。

构造函数将密钥作为参数。这是 cookie 会话的私钥 - 当此值更改时，所有会话数据都将丢失。

通常，您创建一个 SessionStorage 中间件并使用特定的后端实现对其进行初始化，例如 CookieSessionBackend。要访问会话数据，必须使用 `HttpRequest :: session ()`。此方法返回一个 Session 对象，该对象允许我们获取或设置会话数据。

```
use actix_web::{server, App, HttpRequest, Result};
use actix_web::middleware::session::{RequestSession, SessionStorage,
CookieSessionBackend};

fn index(req: &HttpRequest) -> Result<&'static str> {
    // access session data
    if let Some(count) = req.session().get::<i32>("counter")? {
        println!("SESSION value: {}", count);
        req.session().set("counter", count+1)?;
    } else {
        req.session().set("counter", 1)?;
    }

    Ok("Welcome!")
}

fn main() {
    let sys = actix::System::new("basic-example");
    server::new(
        || App::new().middleware(
            SessionStorage::new(
                CookieSessionBackend::signed(&[0; 32])
                    .secure(false)
            )))
        .bind("127.0.0.1:59880").unwrap()
        .start();
    let _ = sys.run();
}
```

错误处理

ErrorHandlers 中间件允许我们为响应提供自定义处理程序。

您可以使用该 `ErrorHandlers::handler()` 方法为特定状态代码注册自定义错误处理程序。您可以修改现有响应或创建完全新响应。错误处理程序可以立即返回响应或返回解析为响应的 future。

```
use actix_web::{
    App, HttpRequest, HttpResponse, Result,
    http, middleware::Response, middleware::ErrorHandlers};

fn render_500<S>(_: &HttpRequest<S>, resp: HttpResponse) ->
Result<Response> {
    let mut builder = resp.into_builder();
    builder.header(http::header::CONTENT_TYPE, "application/json");
    Ok(Response::Done(builder.into()))
}

fn main() {
    let app = App::new()
        .middleware(
            ErrorHandlers::new()
                .handler(http::StatusCode::INTERNAL_SERVER_ERROR,
render_500))
        .resource("/test", |r| {
            r.method(http::Method::GET).f(|_| HttpResponse::Ok());
            r.method(http::Method::HEAD).f(|_| HttpResponse::MethodNotAllowed());
        })
        .finish();
}
```

静态文件

单文件

可以使用自定义路径模式和NamedFile来提供静态文件服务。要匹配路径尾部，我们可以使用[.]正则表达式。

```
use std::path::PathBuf;
use actix_web::{App, HttpRequest, Result, http::Method, fs::NamedFile};

fn index(req: HttpRequest) -> Result<NamedFile> {
    let path: PathBuf = req.match_info().query("tail")?;
    Ok(NamedFile::open(path)?)
}

fn main() {
    App::new()
        .resource(r"/a/{tail:.*}", |r| r.method(Method::GET).f(index))
        .finish();
}
```

目录

StaticFiles可以用作特定目录和子目录文件服务。 StaticFiles必须注册一个App::handler()方法，否则它将无法服务子路径。

```
use actix_web::{App, fs};

fn main() {
    App::new()
        .handler(
            "/static",
            fs::StaticFiles::new(".")
                .show_files_listing())
        .finish();
}
```

该参数是根目录。默认情况下，子目录的文件列表被禁用。尝试加载目录列表将返回404 Not Found响应。要启用文件列表，请使用`* StaticFiles :: show_files_listing () *`方法。

与其显示目录的文件列表，另一种方法是重定向到特定的index文件。使用`StaticFiles::index_file()`方法来配置此重定向。

WebSocket

Actix支持WebSockets开箱即用。可以使用[ws :: WsStream](#)将请求的Payload转换为[ws :: Message](#)流，然后使用流组合器来处理实际的消息，但处理websocket通信使用http actor更简单。

以下是一个简单的websocket echo server的例子：

```
use actix::*;
use actix_web::*;

/// Define http actor
struct Ws;

impl Actor for Ws {
    type Context = ws::WebsocketContext<Self>;
}

/// Handler for ws::Message message
impl StreamHandler<ws::Message, ws::ProtocolError> for Ws {

    fn handle(&mut self, msg: ws::Message, ctx: &mut Self::Context) {
        match msg {
            ws::Message::Ping(msg) => ctx.pong(&msg),
            ws::Message::Text(text) => ctx.text(text),
            ws::Message::Binary(bin) => ctx.binary(bin),
            _ => (),
        }
    }
}

fn main() {
    App::new()
        .resource("/ws/", |r| r.f(|req| ws::start(req, Ws)))
        .finish();
}
```

[websocket directory](#)提供了一个简单的websocket echo server示例。

[websocket-chat directory](#)提供了一个聊天服务器，可以通过websocket或tcp连接进行聊天

HTTP2

如果可能[actix-web](#)自动升级到HTTP/2.0的连接。

协议

HTTP/2.0 protocol over tls without prior knowledge requires [tls alpn](#).

目前，只有[rust-openssl](#)支持

alpn协议需要启用该功能。启用后，HttpServer提供 `serve_tls`方法。[serve_tls](#) method.

```
[dependencies]
actix-web = { version = "0.7", features = ["alpn"] }
openssl = { version = "0.10", features = ["v110"] }
```

```
use std::fs::File;
use actix_web::*;

use openssl::ssl::{SslMethod, SslAcceptor, SslFiletype};

fn main() {
    // load ssl keys
    let mut builder =
        SslAcceptor::mozilla_intermediate(SslMethod::tls()).unwrap();
    builder.set_private_key_file("key.pem", SslFiletype::PEM).unwrap();
    builder.set_certificate_chain_file("cert.pem").unwrap();

    HttpServer::new(
        || App::new()
            .resource("/index.html", |r| r.f(index)))
        .bind("127.0.0.1:8080").unwrap();
        .serve_ssl(builder).unwrap();
}
```

不支持升级到[rfc section 3.2](#) 节中描述的HTTP/2.0模式。明文连接和tls连接都支持HTTP/2 with prior knowledge 启动,[rfc section 3.4](#)

查看具体示例[examples/tls](#).

自动重加载

在开发过程中，cargo自动重新编译变更代码会非常方便。这可以通过使用[cargo-watch](#)来完成。由于actix应用程序通常会绑定到端口以侦听传入的HTTP请求，因此将它与[listenfd](#)和[systemfd](#)实用程序结合起来以确保套接字在应用程序编译和重新加载时保持打开状态是有意义的。

[systemfd](#)将打开一个套接字并将其传递给[cargo-watch](#)以监视更改，然后调用编译器并运行您的actix应用程序。actix应用程序将使用[listenfd](#)获取 [systemfd](#)打开的套接字[systemfd](#)。

需要的二进制文件

对于自动重新加载体验，您需要安装[cargo-watch](#)和 [systemfd](#)。两者都用[cargo install](#)安装

```
cargo install systemfd cargo-watch
```

修改代码

此外，您需要稍微修改您的actix应用程序，以便它可以获取由[systemfd](#)打开的外部套接字。将[listenfd](#)添加到您的应用依赖项中：

```
[dependencices]
listenfd = "0.3"
```

然后修改您的服务器代码以仅以[bind](#)作为回调：

```
extern crate listenfd;

use listenfd::ListenFd;
use actix_web::{server, App, HttpRequest, Responder};

fn index(_req: HttpRequest) -> impl Responder {
    "Hello World!"
}

fn main() {
    let mut listenfd = ListenFd::from_env();
    let mut server = server::new(|| {
        App::new()
            .resource("/", |r| r.f(index))
    });

    server = if let Some(l) = listenfd.take_tcp_listener(0).unwrap() {
        server.listen(l)
    } else {
        server.bind("127.0.0.1:3000").unwrap()
    };
}
```

```
};  
  
server.run();  
}
```

运行服务器

现在运行开发服务器调用这个命令：

```
systemfd --no-pid -s http::3000 -- cargo watch -x run
```

数据库

Diesel

目前， Diesel 1.0不支持异步操作，但可以将actix同步actor系统用作数据库接口API。从技术上讲，同步actor是worker风格的actor。多个同步actors可以并行运行并处理来自同一队列的消息。同步actors以mpsc模式工作。

我们来创建一个简单的数据库api，它可以将一个新的 user row插入到SQLite表中。我们必须定义一个同步actor和该actor将使用的连接。其他数据库可以使用相同的方法。

```
use actix::prelude::*;

struct DbExecutor(SqliteConnection);

impl Actor for DbExecutor {
    type Context = SyncContext<Self>;
}
```

这是我们actor的定义。现在，我们必须定义创建用户消息和响应。

```
struct CreateUser {
    name: String,
}

impl Message for CreateUser {
    type Result = Result<User, Error>;
}
```

我们可以向演员发送CreateUser消息DbExecutor actor，因此我们将收到一个 User实例。接下来，我们必须为此消息定义处理程序实现。

```
impl Handler<CreateUser> for DbExecutor {
    type Result = Result<User, Error>;

    fn handle(&mut self, msg: CreateUser, _: &mut Self::Context) ->
    Self::Result
    {
        use self::schema::users::dsl::*;

        // Create insertion model
        let uuid = format!("{}", uuid::Uuid::new_v4());
        let new_user = models::NewUser {
            id: &uuid,
            name: &msg.name,
        };
    }
}
```

```

    // normal diesel operations
    diesel::insert_into(users)
        .values(&new_user)
        .execute(&self.0)
        .expect("Error inserting person");

    let mut items = users
        .filter(id.eq(&uuid))
        .load::<models::User>(&self.0)
        .expect("Error loading person");

    Ok(items.pop().unwrap())
}
}
}

```

仅此而已！现在，我们可以使用来在任何http处理程序或中间件的DbExecutor actor。我们需要的只是启动DbExecutor actors并将地址存储在http处理程序可以访问的状态中。

```

/// This is state where we will store *DbExecutor* address.
struct State {
    db: Addr<Syn, DbExecutor>,
}

fn main() {
    let sys = actix::System::new("diesel-example");

    // Start 3 parallel db executors
    let addr = SyncArbiter::start(3, || {
        DbExecutor(SqliteConnection::establish("test.db").unwrap())
    });

    // Start http server
    HttpServer::new(move || {
        App::with_state(State{db: addr.clone()})
            .resource("/{name}", |r| r.method(Method::GET).a(index)))
            .bind("127.0.0.1:8080").unwrap()
            .start().unwrap();

        println!("Started http server: 127.0.0.1:8080");
    let _ = sys.run();
}
}

```

我们将在请求处理程序中使用该地址。处理程序返回future对象；因此，我们异步接收响应消息。`Route::a()`必须用于异步处理注册。

```

/// Async handler
fn index(req: HttpRequest<State>) -> Box<Future<Item=HttpResponse,
Error=Error>> {
    let name = &req.match_info()["name"];
}
}
}
}

```

```
// Send message to `DbExecutor` actor
req.state().db.send(CreateUser{name: name.to_owned()})
    .from_err()
    .and_then(|res| {
        match res {
            Ok(user) => Ok(HttpResponse::Ok().json(user)),
            Err(_) => Ok(HttpResponse::InternalServerError().into())
        }
    })
    .responder()
}
```

[diesel directory](#)提供了一个完整的示例。

有关同步actors的更多信息可以在[actix documentation](#)文档中找到。

Sentry

Sentry崩溃报告

Sentry是一个崩溃报告系统，它支持基于actix错误报告的failure crate。使用Sentry中间件，可以自动将服务器错误报告给Sentry。

Sentry中间件

该中间件捕获服务器错误范围（500-599）中的任何错误，并通过其堆栈跟踪将附加的错误发送给哨兵。

要使用中间件，需要初始化和配置Sentry，并且需要添加sentry-actix中间件。此外，注册panic处理程序以通知困难panic也是有意义的。

```
extern crate sentry;
extern crate sentry_actix;

use sentry_actix::SentryMiddleware;

use std::env;

fn main() {
    sentry::init("SENTRY_DSN_GOES_HERE");
    env::set_var("RUST_BACKTRACE", "1");
    sentry::integrations::panic::register_panic_handler();

    let mut app = App::with_state(state)
        .middleware(SentryMiddleware::new())
        // ...
}
```

Reusing the Hub

如果使用这种集成，默认的sentry hub（Hub::current()）通常是错误的。要获得特定的请求，您需要使用ActixWebHubExt trait：

```
use sentry::{Hub, Level};
use sentry_actix::ActixWebHubExt;

let hub = Hub::from_request(req);
hub.capture_message("Something is not well", Level::Warning);
```

The hub can also be made current for the duration of a call. Then Hub::current() works correctly until the end of the run block.

```
use sentry::{Hub, Level};
use sentry_actix::ActixWebHubExt;

let hub = Hub::from_request(req);
Hub::run(hub, || {
    sentry::capture_message("Something is not well", Level::Warning);
});
```

资源

[官网](#)

[API文档](#)

[Github](#)