第 5 章

CHAPTER 5

Rocket 响应

本章将介绍 Rocket 框架中关于响应的处理方式，包括不负责任的响应方式、响应的标准规范、Rocket 的快速响应功能及如何使用 Responder 接口进行自定义响应。在 5.1 节中将讲解什么样的响应才是标准的，以帮助读者理解响应的标准，从而为后续定义 Rocket 的响应做铺垫，接着将理解如何使用 Rocket 提供给开发者的默认 Responder 类型来快速构建常见的响应类型，并且说明响应外壳的概念及如何实现自定义的 Responder 类型，以便使响应更加灵活和个性化。

5.1 Rocket

5.1.1 不负责任的响应方式

首先需要明确的是对于网络请求的响应其实有一套统一的标准，但实际上没有任何人强制任何应用程序都遵循这些标准，然而遵守 HTTP 标准是构建 Web 应用程序的基本原则之一。尽管没有绝对的强制要求，但遵循标准可以确保应用程序与其他系统和服务进行交互时能够正确地处理请求和响应。

实际上之前处理的请求的响应都是一些不负责任的响应方式，因为其他系统和服务难以通过这些响应获得具体的信息，以进行统一处理，并且遵循标准还有一些其他的好处。

(1) 兼容其他系统和服务：设想如下场景，若所有人都不遵守一套统一的标准，我行我素，想怎么返回响应就怎么返回响应，对于自己构建的一套系统而言貌似并不需要耗费太大的成本，但是若想要与他人的系统或服务进行交互就会发现几乎需要在所有的请求或响应中进行兼容与适配，这样对于自己的应用系统与第三方 API、库、服务或框架进行集成时将耗费大量的成本。

(2) 增加代码的可维护性：对于一个程序员而言长时间坚守一个项目几乎是不可能的，项目的代码经多个版本长时间的开发迭代，曾经的开发者或许已经没有精力再去对该项目的代码进行维护了，项目组根据这类情况会增添新鲜血液以继续维护项目，若不遵守一套统一的规范，则新接手项目的开发人员往往会因为难以适应具有强烈个人风格的代码而感

到心力交瘁，但如果按照标准编写代码，则可以增强代码的可读性和可维护性。此外，在应用程序需要进行扩展或升级时，符合标准的代码更容易进行改进和维护。

(3) 快速对错误进行识别与处理：统一的标准能够帮助运维人员、测试人员、开发人员快速地对应用程序产生的错误进行识别，然后定位错误位置，迅速制定出解决方案，以此对产生错误的系统进行恢复，使系统的健壮性进一步增强，同时还可以节省成本。

5.1.2 响应的标准

究竟何种响应构成了一个标准化的结构？对大多数 Web 响应来讲，其结构应当包括以下关键元素。

(1) 状态(status)：揭示服务器当前状态，帮助用户判断服务器是否运行正常。

(2) 请求(request)：这是一个可选字段，用以记录产生当前响应的原始请求信息，有时这项信息是必需的。

(3) 头信息(headers)：涵盖了服务器的响应头信息，如响应体的长度、语言等。

(4) 数据(data)：由服务器提供的实际响应数据，对开发者来讲，这通常是最关注的部分。

使用 Rust 构建的结构体，代码如下：

```
//响应结构体
struct Response {
        //data 是由服务器提供的响应数据
        data: ResponseData,
        //status 是服务器状态
        status: u32,
        //headers 是服务器响应头
        headers: HashMap<String, String>,
        //request 是原始请求信息
        request: HashMap<String,String>,
}

struct ResponseData {
        //code 是真实响应状态码
        code: u32,
        //data 是真实响应数据
        data: Box<dyn Serialize>,
        //msg 是真实响应消息
        msg: String,
}


//响应示例
{
    //data 是由服务器提供的响应数据
    "data": {
```

```
            //code 是真实响应状态码
            "code": 200,
            //data 是真实响应数据
            "data":{},
            //msg 是真实响应消息
            "msg":"",
        },
        //status 是服务器状态
        "status": 200,
        //headers 是服务器响应头
        "headers": {},
        //request 是原始请求信息
        "request":{}
    }
```

5.1.3 Rocket 快速响应

Rocket 框架提供了一套能够帮助开发者快速实现标准化响应的 API，使开发者只需关注响应的状态码和返回的响应数据，使用时需要导入 response 的 status 模块，在 status 模块中提供了一系列标准服务器响应码。

(1) status::Created：将响应状态码设置为 201，表示请求成功并且服务器创建了新的资源。

(2) status::Accepted：将响应状态码设置为 202，并从服务器中添加响应数据进行返回。

(3) status::NoContent：将响应状态码设置为 204，表示当前请求正在被处理，页面不会有任何特殊工作，并且响应也没有任何内容。

(4) status::BadRequest：将响应状态码设置为 400，表示请求的方法错误。

(5) status::Unauthorized：将响应状态码设置为 401，表示当前请求需要授权或鉴权后才能获取。

(6) status::Forbidden：将响应状态码设置为 403，通常，403 错误是由客户端的访问错误配置引起的，常见有权限不足、IP 拦截、防火墙拦截等问题。

(7) status::NotFound：将响应状态码设置为 404，表示无法找到请求资源，常用于请求静态文件数据。

(8) status::Conflict：将响应状态码设置为 409，表示服务器在完成请求时发生了冲突。

(9) status::Custom：自定义状态响应，需要设置响应码和响应数据。

以下是使用 status 模块的示例代码：

```
//第 5 章 hello_rocket/src/example/main.rs
//引入 Rocket 框架宏
#[macro_use]
```

```
extern crate rocket;

//引入标准库中的路径处理模块
use std::path::{Path, PathBuf};
//引入 Rocket 框架的文件服务和响应模块
use rocket::fs::NamedFile;
use rocket::response::status::NotFound;
//引入 Rocket 框架的序列化模块,用于 JSON 处理
use rocket::serde::json::{Json, serde_json};
use rocket::serde::{Serialize, Deserialize};
//引入 Rocket 框架的文件路径处理函数
use rocket::fs::relative;

//定义一个用户结构体,包含用户的名字和年龄
#[derive(Debug, Serialize, Deserialize)]
#[serde(crate = "rocket::serde")]
struct User {
    name: String,
    age: u8,
}

impl User {
    //实现 User 的构造函数,方便创建 User 实例
    pub fn new(name: &str, age: u8) -> Self {
        User {
            name: String::from(name),
            age,
        }
    }
}

//定义一个返回静态字符串的路由处理函数
#[get("/str")]
fn test_str() -> &'static str {
    //返回一个 JSON 格式的字符串
    r#"{\"name\":\"Matt1\",\"age\":10}"#
}

//定义一个返回 String 的路由处理函数
#[get("/string")]
fn test_string() -> String {
    //创建一个 User 实例并将其转换为 JSON 字符串
    let user = User::new("Matt1", 10);
    serde_json::to_string(&user).unwrap()
}

//定义一个异步路由处理函数,返回一个文件
#[get("/file")]
async fn test_option() -> Option<NamedFile> {
    //构建文件的路径并尝试打开该文件
```

```
    let path = Path::new(relative!("static")).join("index.html");
    NamedFile::open(path).await.ok()
}

//定义一个返回结果的路由处理函数,可能返回 User 的 JSON 或 NotFound 错误
#[get("/res/<name>")]
fn test_result(name:&str) -> Result<Json<User>, NotFound<String>> {
    //当请求的 name 参数为"Matt"时,返回一个 User 实例的 JSON
    if "Matt".eq(name){
        let user = User::new("Matt1", 10);
        return Ok(Json(user));
    }
    //否则返回一个 NotFound 错误
    Err(NotFound(String::from("only Matt can be responsed")))
}

//Rocket 框架的启动函数,用于构建和启动 Web 服务
#[launch]
fn rocket() -> _ {
    //构建 rocket 实例并挂载路由
    rocket::build().mount("/api", routes![test_str,test_option,test_result,test_string])
}
```

5.2 Responder

尽管程序中的响应似乎是随意产生的,但实际上,Rocket 框架在背后为开发者完成了大量的工作,充当了一个卓越的封装者,因此,即便是简单地返回一个字符串,Rocket 也会将其优雅地封装成标准的响应格式。Rocket 框架对 Rust 标准库中的多种类型进行了深入整合和处理,包括 String、&str、File、Option 等。

Responder 是 Rocket 框架中的一个关键特质,它负责定义类型的响应能力。简而言之,Responder 的职责在于将特定类型的数据转换为 HTTP 响应。这个机制旨在简化和规范化 HTTP 响应处理过程。通过实现 Responder trait,开发者可以轻松地将各种数据类型(如结构体、字符串、JSON 等)转换成规范的 HTTP 响应,这包括设置状态码、头信息和响应体等。

Responder 设计的核心目的在于提供一种统一且高效的方法来处理 HTTP 响应,从而让代码变得更加简洁和易读,同时充分利用 Rust 的强类型系统。借助 Responder,开发者可以将注意力集中在业务逻辑的实现上,而不需要深入钻研 HTTP 响应的各个细节,极大地提高了开发效率和应用的可维护性。

5.2.1 响应外壳

正如 5.2 节所讲,Rocket 已经帮助开发者将标准库中的一些类型实现了 Responder,这

些类型被称为响应外壳，因为这些类型的设计初衷并不是为了作为响应数据，而是作为包裹响应数据的手段。本节就这些已经实现了 Responder 的响应外壳及一些由 Rocket 提供的可作为响应数据的 Responder 进行示例说明，即使最终多数开发者可能并不会采用这些类型作为返回，但对于后续实现自定义的 Responder 也是一个很好的借鉴。

1. 字符串类型

字符串的响应方式是一种最简单的响应，它常作为一种简单有效的确认资源的返回方式。对于响应字符串而言，Rocket 会默认将 Content-Type 设置为 text/plain，即便所有类型都可以通过序列化的方式转换为字符串进行返回，但依然不提倡这种方法，示例代码如下：

```
//第 5 章 hello_rocket/src/example/default_responder.rs
use rocket::serde::json::serde_json;
use rocket::serde::{Serialize, Deserialize};

//定义一个用户结构体，包含用户的名字和年龄
#[derive(Debug, Serialize, Deserialize)]
#[serde(crate = "rocket::serde")]
struct User {
    name: String,
    age: u8,
}

impl User {
    //实现 User 的构造函数，方便创建 User 实例
    pub fn new(name: &str, age: u8) -> Self {
        User {
            name: String::from(name),
            age,
        }
    }
}

//定义一个返回静态字符串的路由处理函数
#[get("/str")]
fn test_str() -> &'static str {
    r#"{\"name\":\"Matt1\",\"age\":10}"#
}

#[get("/string")]
fn test_string() -> String {
    let user = User::new("Matt1", 10);
    serde_json::to_string(&user).unwrap()
}
```

2. Option < T >

Option 常在静态文件返回时作为响应数据的情况下的外壳，只有当请求的静态文件真

实存在时，Option 将正常返回响应，否则则会返回 404 Not Found，表示当前请求的静态文件并不存在，所以 Option 常与 rocket::fs::NameFile 联合进行使用，示例代码如下：

```
//第 5 章 hello_rocket/src/example/default_responder.rs
use rocket::fs::NamedFile;
use rocket::fs::relative;

#[get("/file")]
async fn test_option() -> Option<NamedFile> {
    let mut path = Path::new(relative!("static")).join("index.html");
    NamedFile::open(path).await.ok()
}
```

这段代码意味着当用户请求的地址为 file 时，Rocket 会构筑一个文件服务器，首先获取项目设置的相对路径 static，然后与 index.html 进行组合，合并为完整的文件地址，文件服务器的路径配置在 4.6.3 节中已经说明，这里就不赘述了。最后使用 NameFile 中的 open()方法尝试通过只读模式打开目标文件。若文件不存在，则返回 None，所以如果该文件地址不存在，则请求时将出现默认的 404 Not Found，除非开发者对 404 进行了重新构建。

3. Result<T,E>

Result 是一种比 Option 更加高级的响应外壳，使用 Result 意味着即使响应处理失败，也会将 Err 作为一种错误响应返回客户端，尽管在下方的示例代码中同样使用了 NotFound 作为错误的返回，但它只是设置了响应码，而不是像 Option 一样出现 404 Not Found。这意味着错误也是由开发者完全控制的，这是一种更加灵活的处理方式，示例代码如下：

```
//第 5 章 hello_rocket/src/example/default_responder.rs
use rocket::response::status::NotFound;
use rocket::serde::json::{Json, serde_json};
use rocket::serde::{Serialize, Deserialize};


//定义一个用户结构体，包含用户的名字和年龄
#[derive(Debug, Serialize, Deserialize)]
#[serde(crate = "rocket::serde")]
struct User {
    name: String,
    age: u8,
}

impl User {
    //实现 User 的构造函数，方便创建 User 实例
    pub fn new(name: &str, age: u8) -> Self {
        User {
            name: String::from(name),
            age,
```

```
        }
    }
}

//定义一个返回结果的路由处理函数,可能返回 User 的 JSON 或 NotFound 错误
#[get("/res/<name>")]
fn test_result(name: &str) -> Result<Json<User>, NotFound<String>> {
    //当请求的 name 参数为"Matt"时,返回一个 User 实例的 JSON
    if "Matt".eq(name) {
        let user = User::new("Matt1", 10);
        return Ok(Json(user));
    }
    //否则返回一个 NotFound 错误
    Err(NotFound(String::from("only Matt can be responsed")))
}
```

4. Rocket 内置 Responder

Rocket 框架的亮点之一是其丰富的内置 Responder 集合,这些 Responder 极大地降低了处理常见响应场景的复杂度,从而有效地减轻了开发者的负担。以下是这些内置 Responder 的说明。

(1) NamedFile:允许在响应中发送文件。它接受一个文件路径作为参数,并将该文件作为响应的内容发送给客户端。它常常在静态资源文件的请求上与 Option 或 Result 响应外壳联用。

(2) Redirect:用于资源重定向,对于文件服务器来讲必不可少,客户端可以永远使用相同的请求路径,通过客户端对其进行路径的重定向可实现同路径访问不同资源的效果,例如某个资源文档进行了版本更新,此时若提供一个新的资源请求路径,则无疑需要与客户端进行商议,资源请求的初衷是永远获取最新的资源,所以服务器端可利用重定向的方式将最新的资源地址返回客户端,结合 NameFile 会达到更优的效果。

(3) Content:用于发送任意类型的内容。它接受一个实现了 ToBytes trait 的值,并将其作为响应的内容发送给客户端。利用这个响应器可以廉价地设置 Content-Type 并与数据一起返回,对于某些快捷请求来讲优势巨大。

(4) Status:和 Content 一样属于一种快捷请求的响应器,Rocket 为该枚举内置了大量标准响应码。需要注意的是该响应器主要关注响应状态,若想要更加精细地进行处理,则需要自定义 Responder。

(5) Flash:用于在请求之间传递一次性的消息,通常用于显示用户操作的结果或提供简短的通知,它是一种特殊的消息,会在响应中设置一个 Cookie,在下一次请求中被读取和消耗,然后立即删除。这使 Flash 消息只在两次请求之间有效,并且用于一次性的瞬时消息传递。

(6) Json:用于将 JSON 数据发送到客户端中,这是最常用的一种响应方式,常见于各种 API 数据传递的场景中。有着可读性高、调试方便、跨语言传输的优点。

(7) MsgPack：和 JSON 一样，都是基于数据的序列化和反序列化的，并且支持跨语言，但 MsgPack 使用二进制格式来表达数据，解码与编码的数据更加紧凑，因此与 JSON 相比可以节省更多的空间和传输带宽，但也因此使它的可读性降低。在序列化和反序列化的速度上 MsgPack 也更快，对于需要大量数据传输的场景更优于 JSON。使用这种形式可以优化服务器端的性能(数据库、文件、消息队列)。

(8) Template：一种呈现动态模板的 Responder，使用 Handlebars 或 Tera 对动态模板进行渲染，这是一种响应式的概念，典型的例子是 React 和 Vue 这些前端框架。

5. 使用 JSON 作为响应

在 4.6.1 节中介绍了如何使用 JSON 数据作为请求的入参，本节将相应地介绍如何将 JSON 数据作为响应回传到客户端中，因此借用 4.6.1 节中的代码进行改写，代码如下：

```
//第 5 章 hello_rocket/src/example/test_json.rs
#[macro_use]
extern crate rocket;

use rocket::serde::{Serialize, Deserialize};
use rocket::serde::json::Json;

//采用 Rocket 框架提供给的 serde 进行序列化与反序列化
#[derive(Debug, Serialize, Deserialize)]
#[serde(crate = "rocket::serde")]
struct User {
    username: String,
    password: String,
}

impl User{
    pub fn new(username: &str, password: &str) -> Self{
        User{
            username: username.to_string(),
            password: password.to_string()
        }
    }
}

//使用 JSON 包装 User
#[post("/login", format = "application/json", data = "<user>")]
fn login(user: Json<User>) -> Json<User> {
    Json(
        User::new(user.username.as_str(),"")
    )
}


#[launch]
fn rocket() -> _ {
```

```
    rocket::build().mount("/api", routes![login])
}
```

接下来进行测试,测试结果如图 5-1 所示。

图 5-1 JSON 响应

5.2.2 自定义 Responder

尽管 Rocket 框架已经提供了众多便利的工具来简化开发过程,但为了进一步优化应用程序的响应并增强其兼容性,设计一个自定义的 Responder 变得格外关键。通常,应用程序的响应需要以 JSON 格式返回数据,因此,自定义 Responder 的关键在于能够高效地将数据封装成一个统一且广泛适用的 JSON 格式。以下是自定义 Responder 的代码示例:

```
//第 5 章 hello_rocket/src/example/define_responder.rs
#[macro_use]
extern crate rocket;

use std::io::Cursor;
use rocket::response::{status, Responder, Response};
use rocket::http::{Status, ContentType};
use rocket::Request;
use rocket::serde::json::{Json, serde_json};
use rocket::serde::{Serialize, Deserialize};

//自定义一个 JSON 形式的统一的 Responder
#[derive(Serialize, Deserialize, Debug)]
#[serde(crate = "rocket::serde")]
struct ResultJsonData<T: Serialize> {
    //返回码
    code: u16,
    //响应数据
```

```
    data: T,
    //响应消息
    msg: String,
}

impl<'r, T: Serialize> Responder<'r, 'static> for ResultJsonData<T> {
    fn respond_to(self, request: &'r Request<'_>) -> rocket::response::Result<'static> {
        let json = serde_json::to_string(&self).unwrap();
        //返回响应
        Response::build()
            //仅表示服务器返回响应状态
            .status(Status::Ok)
            //设置响应的 ContentType
            .header(ContentType::JSON)
            //通过序列化计算
            .sized_body(json.len(), Cursor::new(json))
            //完成构建
            .ok()
    }
}


impl<T: Serialize> ResultJsonData<T> {
    //常规构建
    pub fn new(code: u16, data: T, msg: &str) -> Self {
        ResultJsonData {
            code,
            data,
            msg: String::from(msg),
        }
    }
    //提供响应成功的快速构建方式
    pub fn success(data: T) -> Self {
        ResultJsonData::new(200, data, "success")
    }
    //提供响应失败的快速构建方式
    pub fn failure(data: T, msg: &str) -> Self {
        ResultJsonData::new(500, data, msg)
    }
}


//采用 Rocket 框架提供的 serde 进行序列化与反序列化
#[derive(Serialize, Deserialize, Debug)]
#[serde(crate = "rocket::serde")]
struct User {
    name: String,
    age: u8,
}
```

```
impl User {
    pub fn new(name: &str, age: u8) -> Self {
        User {
            name: String::from(name),
            age,
        }
    }
}

#[get("/test")]
fn define_response() -> ResultJsonData<User> {
    //....
    ResultJsonData::new(
        200, User::new("Matt", 16), "GET USER DATA SUCCESS",
    )
}


#[launch]
fn rocket() -> _ {
    rocket::build().mount("/api", routes![define_response])
}
```

在这里定义了一个统一的 JSON 返回 ResultJsonData，它由响应的返回码 code、响应数据 data 和响应消息 msg 组成，对于响应数据来讲应该是一个泛型，因为 ResultJsonData 应该是一个可被广泛使用的结构体，对于响应数据的泛型可以接收任意实现了 Serialize 的类型，需要注意这里的 Serialize trait 并不是标准库中的，而是使用 serde 库提供的 Serialize trait，这意味着所有可被作为响应返回数据的类型都需要是可序列化的，意思是所有需要被返回的数据都需要实现 serde 库中的 Serialize trait。接着为 ResultJsonData 实现了 Rocket 提供的 Responder trait，其目的是设计它如何进行响应，主要设置返回的状态码、响应头和返回的响应数据，在本示例中通过 Rocket::response::Response 构建返回的响应，设置其服务器返回响应的状态、响应的 Content-Type 及返回的数据实体，在 Response 的构建中通过序列化计算并返回数据实体的 sized_body()方法的两个入参似乎并不好理解，在这个示例中只是为让读者体验响应的构造。完成自定义 Responder trait 的实现后为 ResultJsonData 这个结构体实现了常规构建的 new()方法、提供了响应成功的快速构建 success()方法、响应失败的快速构建 failure()方法，这样可以对该结构体进行更加快捷和简便的复用访问，最后在本示例中使用 User 结构体作为最终真实的数据。在请求中只需将返回值设置为 ResultJsonData 便可廉价地将它构建为一个可用的响应了。