2024年度红山开源创新大赛
操作系统竞赛

项目功能说明书

学　　校：	国防科技大学
参 赛 队 伍：	勇争精美小礼品队
队 伍 成 员：	蔡伦、汤光明、张坦然
指 导 教 师：	任怡、贾周阳
完 成 日 期：	2024年10月23日

目 录

第 1 章 作品概述

1.1 背景分析

在当今软件开发领域，远程过程调用（RPC）技术已成为构建分布式系统的关键工具。随着技术的不断进步，对高效、安全和可扩展的 RPC 框架的需求日益增长。Rust 语言作为一种新兴的系统编程语言，以其独特的内存安全特性和高性能优势，为 RPC 框架的设计和实现提供了新的可能性。

通过将 Rust 应用于 RPC 框架的设计和实现，可以进一步拓宽 Rust 的应用场景，推动其在更广泛的领域中得到应用。同时，利用 Rust 的语言特性，如所有权和生命周期管理，可以提高 RPC 的性能和安全性，从而推动 RPC 技术的发展。

Rust 的所有权模型确保了内存的安全使用，有效避免了悬垂指针、双重释放等内存管理错误，这对于 RPC 框架的稳定性和可靠性至关重要。Rust 的所有权和借用检查器机制使得编写高效的并发代码变得更加容易，这对于需要处理大量并发请求的 RPC 服务来说是一个显著的优势。

基于 Rust 的 RPC 设计和实现不仅具有重要的理论和实践意义，而且对于推动技术创新、提升软件质量、增强系统安全性和稳定性、促进跨平台和多语言支持、优化网络通信性能以及推动开源社区的发展都具有重要的作用。因此，这一选题具有极高的研究价值和应用前景。

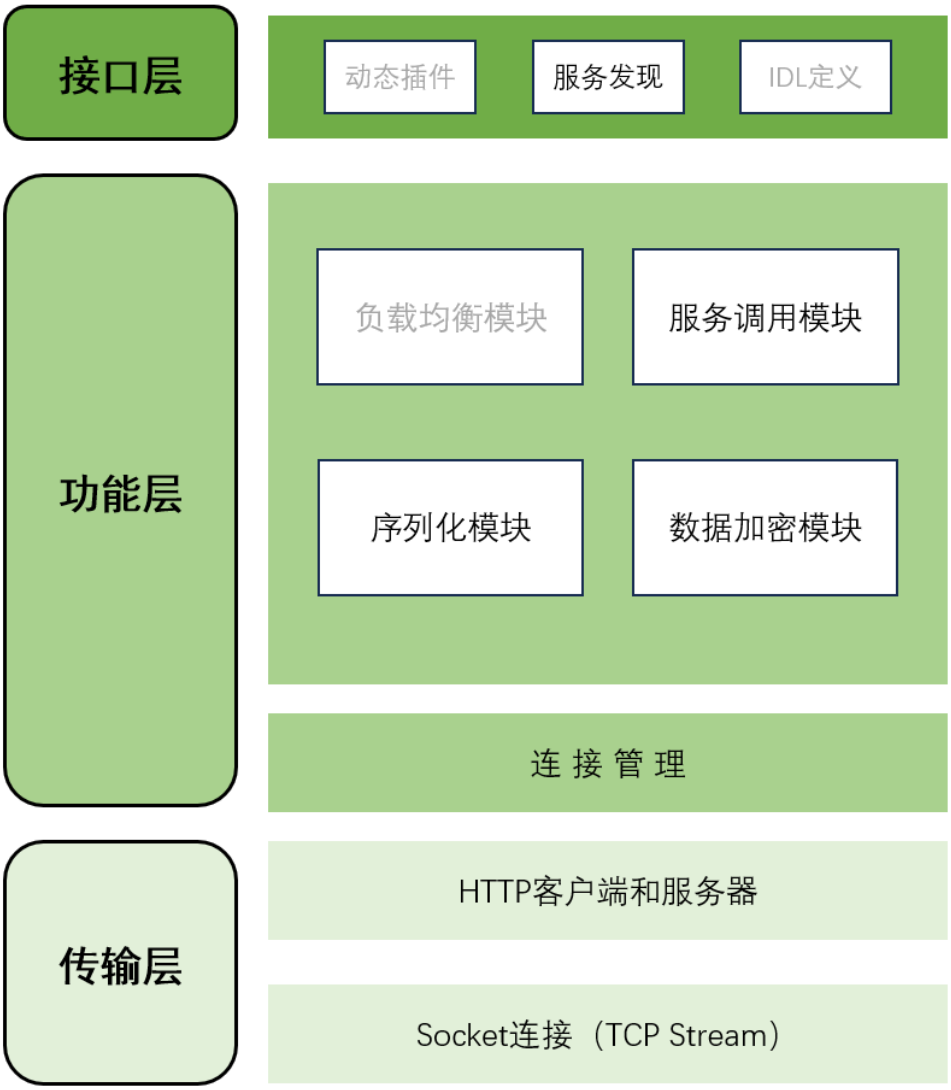
1.2 相关工作

在 Rust 语言的推动下，RPC 框架领域迎来了新的变革。gRPC 作为 Google 推出的高效开源 RPC 框架，凭借其集成的负载均衡、身份验证及加密通信等先进功能，成为应对大规模分布式系统挑战的佼佼者。而基于 Rust 的 Tonic 框架，则以其简化服务创建流程的特点，迅速获得了 Rust 社区的广泛认可。与此同时，Rust-RPC 以其轻量级和易用性，为快速构建小型应用提供了理想选择。此外，RSocket 作为一种创新的反应式通信协议，也借助 Rust 的强大性能与低延迟优势，在网络通信领域展现出了卓越的性能。这些进展不仅彰显了 Rust 在构建高效 RPC 解决方案中的巨大潜力，也为未来的分布式计算技术发展开辟了新的道路。

第 2 章 作品设计与实现

2.1 系统整体框架

在本作品中，项目团队以精湛的技艺和创新的思维，精心打造了一款基于 Rust 语言的高性能 RPC 框架。该框架构建了一个多层次、多模块的实用系统，旨在提供稳定、高效且易于扩展的远程过程调用解决方案。



在传输层，我们采用了 HTTP C/S 架构的通信框架，通过 Socket 连接实现客户端与服务器之间的数据交互。底层则使用了 TCP 流来确保数据的可靠传输，这种设计不仅保证了数据传输的效率，同时也提供了良好的兼容性和可扩展性。

功能层作为整个框架的核心，承接了传输层的数据流，并进一步处理这些数据。它被细分为四个关键模块：负载均衡、服务调用、序列化和数据加密。负载均衡模块负责在多个服务器实例之间分配请求，以确保系统的高可用性和伸缩性。服务调用模块则处理实际的远程过程调用逻辑，确保调用的正确执行。序列化模块负责将数据结构转换为可以在网络上传输的格式，同时也支持将接收到的数据转换回原始结构。数据加密模块则为整个通信过程提供了安全保障，确保数据的机密性和完整性。

接口层是面向用户的层面，它提供了动态服务自动发现技术，使得用户能够轻松地定义和管理服务。此外，我们还开发了用户友好的 IDL 接口定义和服务插件化功能，这极大地简化了服务的动态加载以及增删维护工作。通过这种方式，用户可以根据自己的需求灵活地定制和扩展 RPC 框架的功能。

2.2 系统工作流程

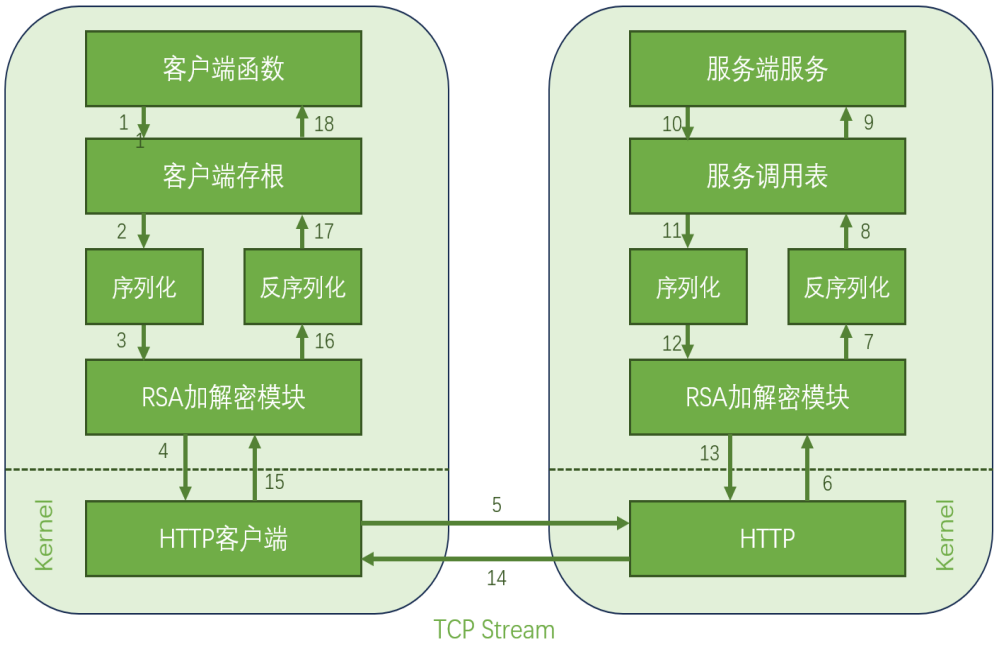
在本项目中，团队精心打造的 RPC 框架展现了其高效与便捷的远程服务调用能力。当客户端需要发起远程服务调用时，它首先会在本地查找存根（stub），这个存根是预先为远程服务生成的代理，用于处理本地与远程服务之间的通信。找到存根后，客户端会将服务需求序列化，即将数据结构转换为可以在网络上传输的格式，然后使用 RSA 加密算法对序列化后的数据进行加密，以确保数据传输的安全性。

完成序列化和加密后，客户端通过 HTTP 客户端以 TCP 流的形式向服务器发送请求消息。这里采用 TCP 流是因为 TCP 提供了可靠的、面向连接的数据传输服务，确保数据能够准确无误地到达服务器。

服务器端接收到来自客户端的请求消息后，首先会对其进行解密操作，恢复出原始的请求数据。接着，服务器会对解密后的数据进行反序列化，将其转换回原始的数据结构。根据请求的内容，服务器会查询服务调用表，找到对应的服务处理函数，并执行相应的处理。处理完成后，服务器会将结果封装成序列化加密的响应消息，再次通过 HTTP 客户端以 TCP 流的形式发送回客户端。

客户端在接收到服务器的响应消息后，会对其进行解密和反序列化操作，最终解析出返回值。这一过程对于用户来说是完全透明的，他们无需关心底层的通信细节和数据处理

流程，只需像调用本地函数一样调用远程服务即可。这种设计极大地简化了用户的操作，提高了开发效率。



2.3 关键技术路线

2.3.1 RPC 调用

RPC（Remote Procedure Call）—远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。比如两个不同的服务 A、B 部署在两台不同的机器上，那么服务 A 如果想要调用服务 B 中的某个方法该怎么办呢？使用 HTTP 请求 当然可以，但是可能会比较慢而且一些优化做的并不好。 RPC 的出现就是为了解决这个问题。 从上面对 RPC 介绍的内容中，概括来讲 RPC 主要解决了：让分布式或者微服务系统中不同服务之间的调用像本地调用一样简单。

服务消费方（client）调用以本地调用方式调用服务； client stub 接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体； client stub 找到服务地址，并将消息发送到服务端； server stub 收到消息后进行解码； server stub 根据解码结果调用本地的服务； 本地服务执行并将结果返回给 server stub； server stub 将返回结果打包成消息并发送至消费方； client stub 接收到消息，并进行解码； 服务消费方得到最终结果。

2.3.2 基于 HTTP 传输协议的通信框架

HTTP（超文本传输协议）是一种用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示。

在 RPC 中，HTTP 可以作为一种传输协议使用。RPC 允许程序调用另一台计算机上的子程序，而不需要开发人员处理这个调用过程中涉及的底层通信细节。当 RPC 基于 HTTP 协议实现时，可以利用 HTTP 的请求-响应模型来执行远程方法调用。这种方式下，客户端通过发送一个 HTTP 请求到服务器来启动一个远程过程，服务器处理该请求并返回一个 HTTP 响应，其中包含了调用结果或错误信息。

使用 HTTP 搭建 RPC 而不直接使用 socket 通信搭建的好处主要体现在以下几个方面：

- **简单易用**：HTTP 协议隐藏了底层网络细节，使得开发者可以专注于业务逻辑的实现，而无需过多关注网络通信的具体实现。这种简单易用的特性降低了开发门槛，提高了开发效率；
- **跨平台和跨语言**：HTTP 协议几乎适用于所有的平台和语言，这使得基于 HTTP 的 RPC 服务可以轻松地与不同平台和语言的应用进行交互。这种跨平台和跨语言的特性为系统的集成和扩展提供了极大的便利；
- **灵活性高**：HTTP 还支持多种数据格式（如 JSON、XML 等），方便数据的传输和解析；
- **易于调试和测试**：由于 HTTP 协议的广泛应用和成熟性，存在大量的工具和库用于 HTTP 服务的调试和测试；
- **利用现有基础设施**：许多企业和组织已经部署了大量的 HTTP 代理、负载均衡器和防火墙等基础设施来处理 HTTP 流量。通过使用 HTTP 搭建 RPC，可以利用这些现有基础设施来实现服务的负载均衡、故障转移和安全防护等功能；
- **易于集成第三方服务**：许多第三方服务（如 API 网关、监控工具等）都支持 HTTP 协议。通过使用 HTTP 搭建 RPC，可以方便地将这些第三方服务集成到系统中，以实现更丰富的功能和更好的可观测性。

2.3.3 基于 Json 序列化和反序列化方法

在 RPC 框架中，序列化和反序列化的作用是确保数据能够有效地在网络上传输并被正确解析。序列化和反序列化是一种数据转化的技术，从数据的用途来看，序列化就是为了将数据按照规定的格式就行重组，在保持原有的数据语义不变的情况下，达到存储或者传输的目的；反序列化则是为了将序列化后的数据重新还原成具有语义的数据，以达到重新使用或者复用原有数据的目的。

JSON，全称是 JavaScript Object Notation，即 JavaScript 对象标记法，是一种轻量级（Light-Meight)、基于文本的(Text-Based)、可读的(Human-Readable)格式。 JSON 无论对于人，还是对于机器来说，都是十分便于阅读和书写的，而且相比 XML(另一种常见的数据交换格式)，文件更小，因此迅速成为网络上十分流行的交换格式。 因为 JSON 本身就是参考 JavaScript 对象的规则定义的，其语法与 JavaScript 定义对象的语法几乎完全相同。JSON 格式的创始人声称此格式永远不升级，这就表示这种格式具有长时间的稳定性，10 年前写的文件，10 年后也能用,没有任何兼容性问题。

JSON 的语法规则十分简单，可称得上“优雅完美”，总结起来有：

- 数组（Array）用方括号(“[ ]”)表示；
- 对象（0bject）用大括号(“{ }”)表示；
- 名称/值对(name/value）组合成数组和对象；
- 名称(name）置于双引号中，值（value）有字符串、数值、布尔值、null、对象和数组。 并列的数据之间用逗号(“,”)分隔。

2.3.4 基于 RSA 算法的数据流加密技术

RSA 加解密算法是一种非对称加密算法，广泛用于数据加密和数字签名。通过使用 RSA 对客户端和服务器的数据流进行加密，确保了服务调用者的隐私安全。总结起来，RSA 加密的过程有如下三个阶段。

2.3.4.1 密钥生成

在密钥生成阶段，首先需要选择两个大素数：p 和 q。并计算它们的模数乘积$n = p \times q$，接着使用如下的公式计算欧拉函数。

$$\phi(n) = (p-1)(q-1)$$

下一步需要选择一个公钥指数e，满足$1 < e < \phi(n)$且e与$\phi(n)$互质，并使用如下的公式计算私钥指数d。

$$ed \equiv 1 \ (mod \ \phi(n))$$

此时，密钥生成完毕，公开公钥：(n, e)，保留私钥：(n, d)。

2.3.4.2 消息加密

对于明文消息m，使用如下加密方法得到密文 c。

$$c \equiv m_e \ (mod \ n)$$

2.3.4.3 密文解密

对于密文c使用如下方法解密方法得到明文 m。

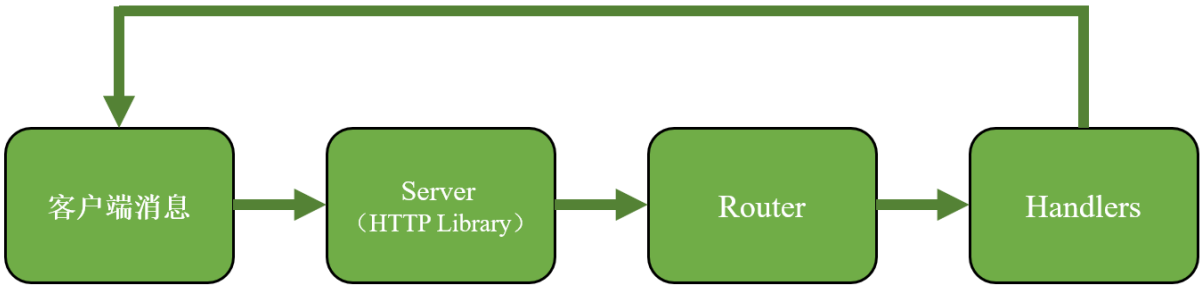
$$m \equiv c_d \ (mod \ n)$$

2.4 系统功能实现

2.4.1 通信模块设计

2.4.1.1 消息流动

经过前期的分析调研，当客户端函数发起远程 RPC 调用时，客户端的请求先发送到 Server（HTTP Library）， 再依次经过 Router 和 Handlers 模块，处理请求回复客户端。



2.4.1.2 通信模块各部分功能

根据上述消息流动分析，构建 Server、Router、Handler、HTTP Library 四个子模块，各部分功能如下：

- Server：监听进来的 TCP 字节流；
- Router：接受 HTTP 请求，并决定调用哪个 Handler；
- Handler：处理 HTTP 请求，构建 HTTP 响应；
- HTTP Library：解释字节流，把它转化为 HTTP 请求；把 HTTP 响应转化回字节流。

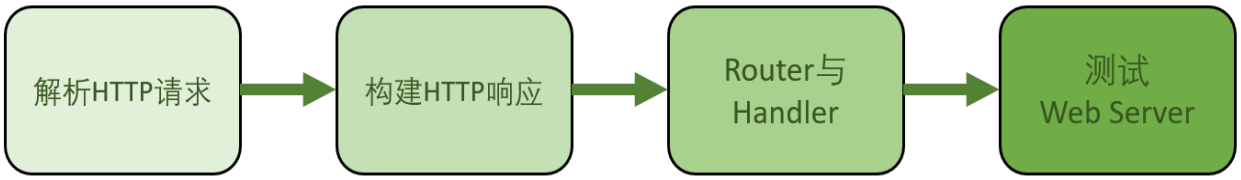
2.4.1.3 构建步骤

在构建 HTTP 通信模块的过程中，首先需要解析 HTTP 请求消息。这包括从客户端接收到的请求中提取出 URL、HTTP 方法（如 GET、POST 等）、头部信息以及可能的请求体数据。解析请求消息是处理 HTTP 请求的第一步，它为后续的处理提供了必要的信息。

接下来，根据解析出的 URL 和 HTTP 方法，进行路由选择，以确定应该将请求交给哪个处理函数（Handler）来处理。路由选择是基于预定义的规则进行的，这些规则通常基于 URL 的模式和 HTTP 方法。一旦选择了合适的处理函数，就会调用该函数来处理请求。处理函数会根据请求的内容执行相应的操作，如访问数据库、执行计算等，并将处理结果封装成响应消息。

在构建 HTTP 响应消息时，需要设置适当的 HTTP 状态码、添加必要的响应头信息，并将处理结果作为响应体返回给客户端。响应消息的构建是根据 HTTP 协议的标准格式进行的，以确保客户端能够正确解析和理解响应内容。

最后，为了验证 HTTP 模块的正确性和稳定性，需要进行测试 Web Server 的操作。通过启动一个测试服务器，可以模拟真实的网络环境，对 HTTP 模块进行各种测试，包括功能测试、性能测试、安全测试等。测试 Web Server 可以帮助发现潜在的问题，并提供反馈以便进行改进和优化。



2.4.1.4 解析 HTTP 请求

首先分析解析 HTTP 请求所需要创建的数据结构如下。

数据结构名称	数据类型	描述
HttpRequest	struct	表示 HTTP 请求
Method	Enum	指定所允许的 HTTP 方法
Version	Enum	指定所允许的 HTTP 版本

接下来是各模块的具体实现，http/lib.rs 是 http 模块的入口点，构建如下。

```
pub mod httprequest;
```

修改 Cargo.toml 配置如下

```
[workspace]

members = ["tcpserver", "tcpclient", "http", "httpserver"]
http/src/httprequest.rs
```

http/src/httprequest.rs 文件主要实现将字符串类型的 http 请求转化为结构体类型的 http 请求，实现了从字符串类型的 http 请求中读取数据构建 http 请求结构体的函数功能。

```
use std::collections::HashMap;

#[derive(Debug, PartialEq)]
//debug {:?} 格式化输出
//自动派生 PartialEq 特性，使得枚举类型可以进行相等性比较
//enum 关键字用于定义枚举类型
pub enum Method
{
    Get,
    Post,
    Uninitialized,
}

//impl 关键字用于实现某个特性（trait）。
//From<&str> 是一个标准库中的特性，表示可以从 &str 类型转换为 Method 类型
impl From<&str> for Method
{
    fn from(s: &str) -> Method
    {
```

```
20            match s
21            {
22                "GET" => Method::Get,
23                "POST" => Method::Post,
24                _ => Method::Uninitialized,
25                //其他任何值都会返回 Method::Uninitialized
26            }
27        }
28    }
29
30    #[derive(Debug, PartialEq)]
31    pub enum Version
32    {
33        V1_1,
34        V2_0,
35        Uninitialized,
36    }
37
38    impl From<&str> for Version
39    {
40        fn from(s: &str) -> Version
41        {
42            match s
43            {
44                "HTTP/1.1" => Version::V1_1,
45                "HTTP/2.0" => Version::V2_0,
46                _ => Version::Uninitialized,
47            }
48        }
49    }
50
51    #[derive(Debug, PartialEq)]
52    pub enum Resource {
53        Path(String),
54    }
55
56    #[derive(Debug)]
57    pub struct HttpRequest {
58        pub method: Method,// 请求方法
59        pub version: Version,// HTTP 版本
60        pub resource: Resource,// 请求资源
61        pub headers: HashMap<String, String>,// 请求头
62        pub msg_body: String, // 请求体
63    }
64
```

```
65  impl From<String> for HttpRequest
66  {
67      fn from(req: String) -> Self
68      {
69          let mut parsed_method = Method::Uninitialized;
70          let mut parsed_version = Version::V1_1;
71          let mut parsed_resource = Resource::Path("".to_string());
72          let mut parsed_headers = HashMap::new();
73          let mut parsed_msg_body = "";
74
75          for line in req.lines()
76          {
77              if line.contains("HTTP")
78              {
79                  let (method, resource, version) = process_req_line(line);
80                  parsed_method = method;
81                  parsed_resource = resource;
82                  parsed_version = version;
83              }
84              else if line.contains(":")
85              {
86                  let (key, value) = process_header_line(line);
87                  parsed_headers.insert(key, value);
88              }
89              else if line.len() == 0 {}
90              else
91              {
92                  parsed_msg_body = line;
93                  // parsed_msg_body = parsed_msg_body + line;
94              }
95          }
96
97          HttpRequest
98          {
99              method: parsed_method,
100             version: parsed_version,
101             resource: parsed_resource,
102             headers: parsed_headers,
103             msg_body: parsed_msg_body.to_string(),
104         }
105     }
106 }
107
108 fn process_req_line(s: &str) -> (Method, Resource, Version)
109 {
```

```
110         //将字符串按空白字符（空格、制表符、换行符等）分割成一个迭代器
111         let mut words = s.split_whitespace();
112         //words.next() 方法从迭代器中获取下一个元素。
113         let method = words.next().unwrap();
114         let resource = words.next().unwrap();
115         let version = words.next().unwrap();
116 
117         //构造返回值
118         (
119             method.into(),
120             Resource::Path(resource.to_string()),
121             version.into(),
122         )
123     }
124 
125     fn process_header_line(s: &str) -> (String, String)
126     {
127         let mut header_items = s.split(":");
128         let mut key = String::from("");
129         let mut value = String::from("");
130         if let Some(k) = header_items.next()
131         {
132             key = k.to_string();
133         }
134         if let Some(v) = header_items.next()
135         {
136             value = v.to_string();
137         }
138 
139         (key, value)
140     }
```

2.4.1.5 构建 HTTP 响应

2.4.1.5.1 需求分析

经过调研准备，搭建 HTTP 服务器客户端时需要实现如下所示的方法或特性来构建 HTTP 响应。

需要实现的方法或特性	用途
Default trait	指定成员的默认值
new()	使用默认值创建一个新的结构体
send_response()	构建响应，将原始字节通过 TCP 传送
getter	获得成员的值
From trait	能够将 HttpResponse 转化为 String

2.4.1.5.2 代码实现

http/lib.rs

```
pub mod httprequest;
pub mod httpresponse;
```

http/src/httpresponse.rs，该文件负责实现够 http 响应的结构体，实现将 http 响应返回客户端的功能。

```
use std::collections::HashMap;
use std::io::{Result, Write};

#[derive(Debug, PartialEq, Clone)]
//'a 是生命周期参数，表示结构体中引用的生命周期。
pub struct HttpResponse<'a>
{
    version: &'a str,
    status_code: &'a str,
    status_text: &'a str,
    headers: Option<HashMap<&'a str, &'a str>>,
    body: Option<String>,
}
// 这意味着 version、status_code、status_text 和 headers 中的字符串切片必须在 HttpResponse 结构体存在期间保持有效。
impl<'a> Default for HttpResponse<'a> // 为类型提供一个默认值
{
```

```
        fn default() -> Self
        {
            Self {
                version: "HTTP/1.1".into(),//版本号
                status_code: "200".into() //状态码
                status_text: "OK".into(),//状态文本
                headers: None,// HTTP 响应头
                body: None,// HTTP 响应体
            }
        }
    }

    impl<'a> From<HttpResponse<'a>> for String //HttpResponse 类型转换为 String 类型
    {
        fn from(res: HttpResponse<'a>) -> String
        {
            let res1 = res.clone();
            format!(
                "{} {} {}\r\n{}Content-Length: {}\r\n\r\n{}",
                &res1.version(),
                &res1.status_code(),
                &res1.status_text(),
                &res1.headers(),
                &res.body.unwrap().len(),
                &res1.body()
            )
        }
    }

    impl<'a> HttpResponse<'a>
    {
        pub fn new(
            status_code: &'a str,
            headers: Option<HashMap<&'a str, &'a str>>,
            body: Option<String>,
        ) -> HttpResponse<'a>
        {
            let mut response: HttpResponse<'a> = HttpResponse::default();
            if status_code != "200"
            {
                response.status_code = status_code.into();
            };

            response.headers = match &headers
            {
```

```
62                 Some(_h) => headers,
63                 None => {
64                     let mut h = HashMap::new();
65                     h.insert("Content-Type", "text/html");
66                     Some(h)
67                 }
68             };
69             response.status_text = match response.status_code
70             {
71                 "200" => "OK".into(),
72                 "400" => "Bad Request".into(),
73                 "404" => "Not Found".into(),
74                 "500" => "Internal Server Error".into(),
75                 _ => "Not Found".into(),
76             };
77 
78             response.body = body;
79             //返回构造好的响应
80             response
81         }
82 
83         pub fn send_response(&self, write_stream: &mut impl Write) -> Result<()>
84         {
85             let res = self.clone();
86             let response_string: String = String::from(res);
87             let _ = write!(write_stream, "{}", response_string);
88             // println!("{}", response_string);
89 
90             Ok(())
91         }
92 
93         fn version(&self) -> &str
94         {
95             self.version
96         }
97 
98         fn status_code(&self) -> &str
99         {
100             self.status_code
101         }
102 
103         fn status_text(&self) -> &str
104         {
105             self.status_text
106         }
```

```
107
108         fn headers(&self) -> String
109         {
110             let map: HashMap<&str, &str> = self.headers.clone().unwrap();
111             let mut header_string: String = "".into();
112             for (k, v) in map.iter()
113             {
114                 header_string = format!("{}{}:{}\r\n", header_string, k, v);
115             }
116             header_string
117         }
118
119         pub fn body(&self) -> &str
120         {
121             match &self.body
122             {
123                 Some(b) => b.as_str(),//将 String 转换为 &str
124                 None => "",
125             }
126         }
127     }
```

2.4.1.5.3 构建 server 模块

在 httpserver/Cargo.toml 中配置。

```
1         [package]
2         name = "httpserver"
3         version = "0.1.0"
4         edition = "2021"
5
6         [dependencies]
7         http = {path = "../http"}
```

httpserver/main.rs 负责启动服务器，确定运行的端口。

```
1     mod handler;
2     mod router;
3     mod server;
4
5     use server::Server;
6
7     fn main()
8     {
9         let server = Server::new("localhost:3000");
```

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

httpserver/server.rs 接受 http 请求，将其转化为 httprequest 结构体，由路由根据其请求的服务类型进行处理。

```
use super::router::Router;
use http::httprequest::HttpRequest;
use std::io::prelude::*;
use std::net::TcpListener;
use std::str;

pub struct Server<'a>
{
    socket_addr: &'a str //服务器套接字地址
}

impl<'a> Server<'a>
{
    //构造函数
    pub fn new(socket_addr: &'a str) -> Self
    {
        Server { socket_addr }
    }

    pub fn run(&self)
    {

        let connection_listener = TcpListener::bind(self.socket_addr).unwrap();
        println!("Running on {}", self.socket_addr);

        for stream in connection_listener.incoming() {
            let mut stream = stream.unwrap();
            println!("Connection established");

            let mut read_buffer = [0; 200];
            stream.read(&mut read_buffer).unwrap();

            // 打印原始的 HTTP 请求
            let request_str = match str::from_utf8(&read_buffer)
            {
                Ok(v) => v,
                Err(e) => panic!("Invalid UTF-8 sequence: {}", e),
            };
```

```
                println!("Received HTTP request:\n{}\n",
            request_str.trim_end_matches(char::from(0)));

                //将数据转化成 http 包格式
                let req= HttpRequest::from(request_str.to_string());
                println!("{:?}", req);
                let req: HttpRequest = String::from_utf8(read_buffer.to_vec()).unwrap().into();
                // println!("{:?}", req);
                Router::route(req, &mut stream);
            }
        }
    }
```

2.4.1.5.4 构建 router 和 handler 模块

router 模块负责分析请求内容，handler 模块负责处理消息，在 httpserver/Cargo.toml 中配置。

```
    [package]
    name = "httpserver"
    version = "0.1.0"
    edition = "2021"

    # See more keys and their definitions at https://doc.rust-
    lang.org/cargo/reference/manifest.html

    [dependencies]
    http = {path = "../http"}
    serde = {version="1.0.163", features = ["derive"]}
    serde_json = "1.0.96"
```

httpserver/src/handler.rs 主要有三种类型的处理，一个是网络服务的处理，一个是静态页面请求的处理（这个目前主要用来做测试），另一个是无法找到资源的处理。网络服务请求的处理是该项目的核心模块之一，与系统的服务调用模块进行对接。

```
    use http::{httprequest::HttpRequest, httpresponse::HttpResponse};
    use serde::{Deserialize, Serialize};//用于序列化和反序列化 JSON 数据
    use std::collections::HashMap;
    use std::env;
    use std::fs;

    pub trait Handler
    {
```

```
9         //定义了一个抽象方法，用于处理 HTTP 请求并返回 HttpResponse。
10        fn handle(req: &HttpRequest) -> HttpResponse;
11
12        fn load_file(file_name: &str) -> Option<String>
13        {
14            let default_path = format!("{}/public", env!("CARGO_MANIFEST_DIR"));
15            let public_path = env::var("PUBLIC_PATH").unwrap_or(default_path);
16            let full_path = format!("{}/{}", public_path, file_name);
17
18            let contents = fs::read_to_string(full_path);
19            //读取文件内容，并返回 Option<String>。
20            contents.ok()
21        }
22    }
23
24    pub struct StaticPageHandler;
25    pub struct PageNotFoundHandler;
26    pub struct WebServiceHandler;
27
28    #[derive(Serialize, Deserialize)]
29    pub struct OrderStatus
30    {
31        order_id: i32,
32        order_date: String,
33        order_status: String,
34    }
35
36    impl Handler for PageNotFoundHandler
37    {
38        fn handle(_req: &HttpRequest) -> HttpResponse
39        {
40            HttpResponse::new("404", None, Self::load_file("404.html"))
41        }
42    }
43
44    impl Handler for StaticPageHandler
45    {
46        fn handle(req: &HttpRequest) -> HttpResponse
47        {
48            let http::httprequest::Resource::Path(s) = &req.resource;
49            let route: Vec<&str> = s.split("/").collect();
50            match route[1]
51            {
52                "index" => HttpResponse::new("200", None, Self::load_file("index.html")),
53                "health" => HttpResponse::new("200", None, Self::load_file("health.html")),
```

```
54
55              path => match Self::load_file(path)
56              {
57                  Some(contents) =>
58                  {
59                      let mut map: HashMap<&str, &str> = HashMap::new();
60                      if path.ends_with(".css")
61                      {
62                          map.insert("Content-Type", "text/css");
63                      }
64                      else if path.ends_with(".js")
65                      {
66                          map.insert("Content-Type", "text/javascript");
67                      }
68                      else
69                      {
70                          map.insert("Content-Type", "text/html");
71                      }
72                      HttpResponse::new("200", Some(map), Some(contents))
73                  }
74                  None => HttpResponse::new("404", None, Self::load_file("404.html")),
75              },
76          }
77      }
78  }
79
80  impl WebServiceHandler
81  {
82      fn load_json() -> Vec<OrderStatus>
83      {
84          let default_path = format!("{}/data", env!("CARGO_MANIFEST_DIR"));
85          let data_path = env::var("DATA_PATH").unwrap_or(default_path);
86          let full_path = format!("{}/{}", data_path, "orders.json");
87          let json_contents = fs::read_to_string(full_path);
88          let orders: Vec<OrderStatus> =
89              serde_json::from_str(json_contents.unwrap().as_str()).unwrap();
90          orders
91      }
92  }
93
94  impl Handler for WebServiceHandler
95  {
96      fn handle(req: &HttpRequest) -> HttpResponse
97      {
98          //获取资源路径 服务类型
```

```
99          let http::httprequest::Resource::Path(s) = &req.resource;
100         // 获取请求体 数据
101         // let req_data =&req.msg_body;
102         // 将路径分割成数组
103         let route: Vec<&str> = s.split("/").collect();
104         // localhost:3000/api/shipping/orders
105         match route[2]
106         {
107             //果 route[2] 是 "shipping" 并且路径有至少四个部分（即 route.len() > 2），并且第四部分 route[3] 是 "orders"。
108             "shipping" if route.len() > 2 && route[3] == "orders" =>
109             {
110                 let body = Some(serde_json::to_string(&Self::load_json()).unwrap());
111                 let mut headers: HashMap<&str, &str> = HashMap::new();
112                 headers.insert("Content-Type", "application/json");
113                 HttpResponse::new("200", Some(headers), body)
114             }
115             _ => HttpResponse::new("404", None, Self::load_file("404.html")),
116         }
117     }
118 }
```

httpserver/src/router.rs

```
1   use super::handler::{Handler, PageNotFoundHandler, StaticPageHandler, WebServiceHandler};
2   use http::{httprequest, httprequest::HttpRequest, httpresponse::HttpResponse};
3   use std::io::prelude::*;
4
5   pub struct Router;
6
7   impl Router
8   {
9       pub fn route(req: HttpRequest, stream: &mut impl Write) -> ()
10      {
11          match req.method
12          {
13              httprequest::Method::Get => match &req.resource
14              {
15                  httprequest::Resource::Path(s) =>
16                  {
17                      let route: Vec<&str> = s.split("/").collect();
18                      match route[1]
19                      {
20                          "api" =>
```

```
                    {
                        let resp: HttpResponse = WebServiceHandler::handle(&req);
                        let _ = resp.send_response(stream);
                    }
                    _ =>
                    {
                        let resp: HttpResponse = StaticPageHandler::handle(&req);
                        let _ = resp.send_response(stream);
                    }
                }
            }
            _ =>
            {
                let resp: HttpResponse = PageNotFoundHandler::handle(&req);
                let _ = resp.send_response(stream);
            }
        }
    }
}
```

httpserver/data/orders.json

```
[
    {
        "order_id": 1,
        "order_date": "21 Jan 2024",
        "order_status": "Delivered"
    },
    {
        "order_id": 2,
        "order_date": "2 Feb 2024",
        "order_status": "Pending"
    }
]
```

2.4.2 序列和反序列化模块设计

经过前期调研发现使用 json 为基础来设计序列化，相比于 xml 而言，能在保证准确性的同时，较大程度地提升效率。 在一次 RPC 调用的过程中客户端（服务调用方）需要提交服务名称（service_name）、数据长度（data_len）、服务参数（data）三个字段。由序

列化模块转化为 json 格式添加并到 HTTP 报文的 body 中，序列化的数据会在服务端（服务提供方）进行反序列化。

2.4.2.1 序列化模块设计

```
fn serde_obj(service_name: String,len_service_data: String,service_data: String) -> String {
    let request: String = Message{
        service_name: service_name,
        data_len: len_service_data,
        data: service_data,

    };
    let request = serde_json::to_string(&request ).expect("Failed to serialize to JSON");
    return request;
}

fn serde_vec(numbers: Vec<u8>) -> String {
    // 将数组序列化为 JSON 字符串
    let json_string = serde_json::to_string(&numbers).expect("Failed to serialize to JSON");
    // println!("Serialized JSON: {}", json_string);
    return json_string;
}
```

2.4.2.2 反序列化模块设计

```
fn deserder_obj(request: String) -> Message {
    let request: Message = serde_json::from_str(&request).expect("Failed to deserialize from JSON");
    return request;
}

fn deserder_vec(json_string: String) -> Vec<u8> {
    // 将 JSON 字符串反序列化为数组
    let numbers: Vec<u8> = serde_json::from_str(&json_string).expect("Failed to deserialize from JSON");
    // println!("Deserialized numbers: {:?}", numbers);
    return numbers;
}
```

2.4.3 服务调用模块设计

在服务端接收到客户端所传递的参数之后，将其反序列化，之后进行服务匹配，从所提供服务的列表中找到对应服务的函数地址，并进行调用。

```
use service::stub;
use service::message::Message;


pub fn handle_client(req_data: &String,private_key: &[u8]) -> String
{
    let mes=serde_json::from_slice::<Message>(req_data.as_bytes());
    //获取服务类型
    match mes {
        Ok(mes) => {
            //服务匹配
            let res= match_service(mes,private_key);
            // stream.write(res.as_bytes()).unwrap();
            return res
        },
        Err(e) => {
            let error_message = format!("Error parsing message: {}", e);
            // stream.write(error_message.as_bytes()).unwrap();
            return error_message
        }
    };


}

pub fn match_service(mes: Message,private_key: &[u8]) -> String {
    let mes_sname = mes.service_name.trim();
    let mut mes_data=crypto::decrypto(private_key,mes.data);
    let mes_data: Vec<&str> = mes.data.trim().split(' ').collect();
    // let mes_data=mes.data.trim().split(' ').collect();
    let res=match mes_sname {
        "add" => format!("{}", stub::add(mes_data[0].parse::<i32>().unwrap_or(0), mes_data[1].parse::<i32>().unwrap_or(0))),
        "subtract" => format!("{}", stub::subtract(mes_data[0].parse::<i32>().unwrap_or(0), mes_data[1].parse::<i32>().unwrap_or(0))),
        _ => "Unknown request".to_string(),
    };
    return res;
}
```

2.4.4 加解密模块设计

在本项目所构建的 RPC 框架下，采用 RSA 算法对所需要传输的数据进行加解密。加解密模块根据加密对象的类型来实现函数，在整个数据传递和处理的过程中，使用较多的两种数据类型是 String 和 Vec[u8]，类型不同函数的处理也就不同，因此针对这两种数据类型，分别实现了对其进行加密和解密的函数。针对 String 的加密结果类型是 Vec[u8]，在在实际使用的过程中，String 类型的加密和 Vec[u8]的解密使用的较多。

2.4.4.1 密钥生成

```
use rsa::{Pkcs1v15Encrypt, RsaPrivateKey, RsaPublicKey};
pub fn generate_keys() -> (RsaPrivateKey,RsaPublicKey) {
    let mut rng = rand::thread_rng();
    let bits = 2048;
    let priv_key = RsaPrivateKey::new(&mut rng, bits).expect("failed to generate a key");
    let pub_key = RsaPublicKey::from(&priv_key);
    (priv_key, pub_key)
}
```

2.4.4.2 加密模块

```
pub fn encrypt_u8(pub_key: &RsaPublicKey, message: &[u8]) -> Vec<u8> {


    let mut rng2 = rand::thread_rng();
    let enc_data = pub_key.encrypt(&mut rng2, Pkcs1v15Encrypt, message).expect("failed to encrypt");
    return enc_data;
}

pub fn encrypt_str(pub_key: &RsaPublicKey, message: &str) -> Vec<u8> {
    // 将字符串转换为字节数组
    let message_bytes = message.as_bytes();
    let mut rng2 = rand::thread_rng();
    let enc_data = pub_key.encrypt(&mut rng2, Pkcs1v15Encrypt, message_bytes).expect("failed to encrypt");
    return enc_data;
```

2.4.4.3 解密模块

```
pub fn decrypt_str(priv_key: &RsaPrivateKey, ciphertext: &str) -> Vec<u8> {
    let ciphertext = ciphertext.as_bytes();
    let dec_data = priv_key.decrypt(Pkcs1v15Encrypt, &ciphertext).expect("failed to decrypt");
    return dec_data;
}

pub fn decrypt_u8(priv_key: &RsaPrivateKey, ciphertext: &[u8]) -> Vec<u8> {
    let dec_data = priv_key.decrypt(Pkcs1v15Encrypt, &ciphertext).expect("failed to decrypt");
    return dec_data;
}
```

第 3 章 作品测试与分析

3.1 测试目的

本章节的目的在于验证本作品中提出的方法真实有效、应用于实际应用中的稳定性，以及 RPC 整体框架是否符合用户需求，是否已达到预期的功能目标，同时对测试质量进行分析。

3.2 项目全模块冒烟测试

3.2.1 通信模块可靠性测试

3.2.1.1 测试用例编写

简单写了网页进行测试

httpserver/public/404.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Not Found!</title>
</head>

<body>
    <h1>404 Error</h1>
    <p>Sorry the requested page does not exist</p>
</body>

</html>
```

httpserver/public/health.html

```
    <!DOCTYPE html>
    <html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Health!</title>
```

```
    </head>

    <body>
        <h1>Hello welcome to health page!</h1>
        <p>This site is perfectly fine</p>
    </body>

    </html>
```

httpserver/public/index.html

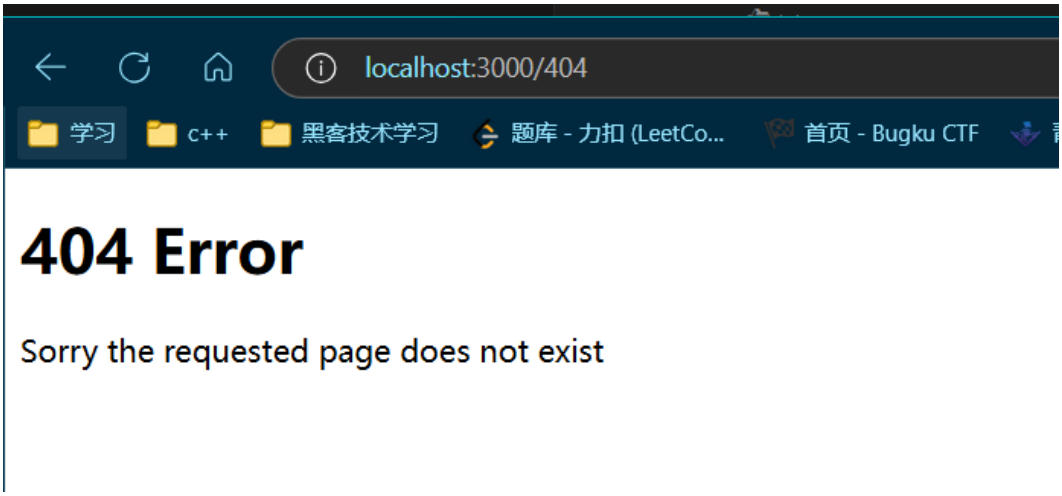
```
    <!DOCTYPE html>
    <html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="styles.css">
        <title>Index!</title>
    </head>

    <body>
        <h1>Hello, welcome to home page</h1>
        <p>This is the index page for the web site</p>
    </body>

    </html>
```

httpserver/public/styles.css

```
    h1
    {
        color: red;
        margin-left: 25px;
    }
```

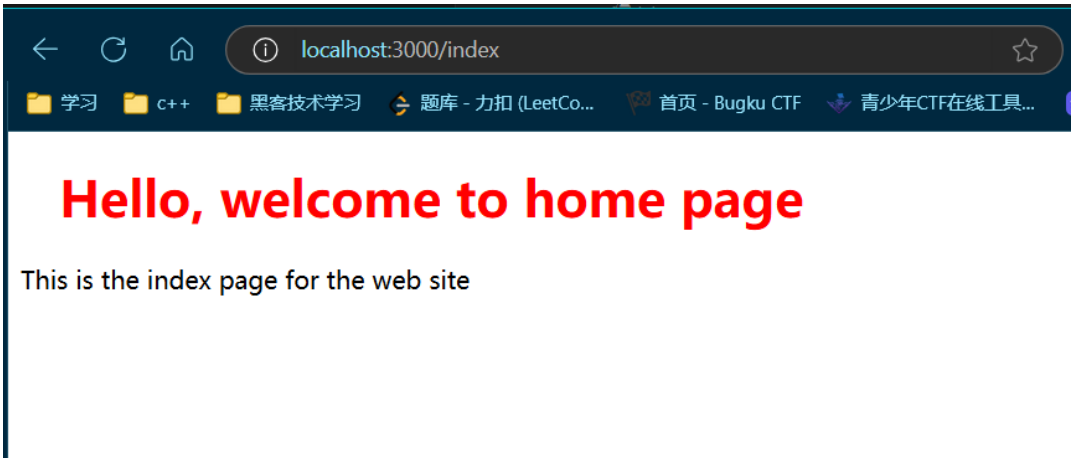
3.2.1.2 测试结果分析

对网页资源进行访问，得到如下测试结果，成功访问。说明项目能够实现网页路由转换和请求处理，服务器能正常运行。



Hello welcome to health page!

This site is perfectly fine



3.2.2 序列化和反序列化模块准确性测试

3.2.2.1 测试用例编写

序列化和反序列化针对数组和结构体，分别测试数组序列化、数组反序列化、结构体序列化和结构体反序列化四个功能

```
#[cfg(test)]
mod tests{
    use super::*;

    #[test]
    fn test_objserde() {
        let mes =Message {
            service_name: "add".to_string(),
            data_len: 3,
            data: "1 2".to_string(),
        };

        // 将结构体序列化为 JSON 字符串
        let json_string =serde_obj(&mes);

        println!("Serialized JSON: {}", json_string);
    }

    #[test]
    fn test_objdeserde() {
        let json_string = r#"{
            "service_name": "add",
            "data_len": 3,
            "data": "1 2"
        }"#;

        // 将 JSON 字符串反序列化为结构体
        let person: Message = deserder_obj(json_string.to_string());

        println!("Deserialized struct: {:?}", person);
    }

    #[test]
    fn test_vecserde() {
        let numbers = [108, 123, 122];

    // 将数组序列化为 JSON 字符串
        let json_string =serde_vec(&numbers);
```

```
39
40              println!("Serialized JSON: {}", json_string);
41          }
42
43          #[test]
44          fn test_vecdeserde() {
45              let json_string = r#"[108, 123, 122]"#;
46
47          // 将 JSON 字符串反序列化为数组
48              let numbers: Vec<u8> = deserder_vec(json_string.to_string());
49
50              println!("Deserialized array: {:?}", numbers);
51          }
52      }
```

3.2.2.2 测试结果分析

测试成功，4 个测试用例都能通过

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.05s
     Running unittests src\lib.rs (target\debug\deps\service-d9045de09d56fa1c.exe)

running 4 tests
test serialization_json::tests::test_objdeserde ... ok
test serialization_json::tests::test_objserde ... ok
test serialization_json::tests::test_vecdeserde ... ok
test serialization_json::tests::test_vecserde ... ok

successes:

---- serialization_json::tests::test_objdeserde stdout ----
Deserialized struct: Message { service_name: "add", data_len: 3, data: "1 2" }

---- serialization_json::tests::test_objserde stdout ----
Serialized JSON: {"service_name":"add","data_len":3,"data":"1 2"}

---- serialization_json::tests::test_vecdeserde stdout ----
Deserialized array: [108, 123, 122]

---- serialization_json::tests::test_vecserde stdout ----
Serialized JSON: [108,123,122]


successes:
    serialization_json::tests::test_objdeserde
    serialization_json::tests::test_objserde
    serialization_json::tests::test_vecdeserde
    serialization_json::tests::test_vecserde

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

3.2.3 服务调用模块功能测试

3.2.3.1 测试用例编写

服务调用部分主要分为 stub.rs 和 call.rs 两个文件，stub.rs 存储服务函数，call.rs 负责处理请求和匹配服务，所以对这两部分分别进行测试。

```
#[cfg(test)]
mod tests{
    use super::*;
    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }
    #[test]
    fn test_subtract() {
        assert_eq!(subtract(5, 2), 3);
    }
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_to_vec_u8() {
        let input = "[15, 69, 40]";
        let result = parse_to_vec_u8(input);
        assert_eq!(result, Ok(vec![15, 69, 40]));
    }



    #[test]
    fn test_match_service() {
        let mes = Message {
            service_name: "add".to_string(),
            data_len: 2,
            data: "1 2".to_string(),
        };

        let result = match_service(mes);
        assert_eq!(result, "3");
    }

```

```
    #[test]
    fn test_match_service_unknown_request() {
        let mes = Message {
            service_name: "unknown".to_string(),
            data_len: 2,
            data: "1 2".to_string(),
        };

        let result = match_service(mes);
        assert_eq!(result, "Unknown request");
    }

    #[test]
    fn test_handle_client() {
        let req_data = "{\"service_name\":\"add\",\"data_len\":2,\"data\":\"1 2\"}";
        let (priv_key, pub_key) = rsa_crypto::generate_keys();
        let req_data=rsa_crypto::encrypt_str(&pub_key,req_data);
        //转化为字符串
        let mut string = String::from("[");
        for (i, &value) in req_data.iter().enumerate() {
            if i != 0 {
                string.push_str(", ");
            }
            string.push_str(&value.to_string());
        }
        string.push_str("]");

        let result = handle_client(&string, &priv_key, &pub_key);
        assert_eq!(result, "3");
    }
}
```

3.2.3.2 测试结果分析

两部分测试结果如下，stub 中各个函数功能能正常运行。

```
*  正在执行任务: C:\Users\admin\.cargo\bin\cargo.exe test --package service --lib -- stub::tests --show-output

   Compiling service v0.1.0 (D:\Desk\rust\s2\service)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.68s
     Running unittests src\lib.rs (target\debug\deps\service-d9045de09d56fa1c.exe)

running 2 tests
test stub::tests::test_add ... ok
test stub::tests::test_subtract ... ok

successes:

successes:
    stub::tests::test_add
    stub::tests::test_subtract

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 4 filtered out; finished in 0.00s
```

call.rs 能正常处理服务请求和匹配服务类型。

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.77s
     Running unittests src/main.rs (target\debug\deps\httpserver-eaed719967b5f7a0.exe)

running 4 tests
test handler::call::tests::test_match_service ... ok
test handler::call::tests::test_match_service_unknown_request ... ok
test handler::call::tests::test_parse_to_vec_u8 ... ok
test handler::call::tests::test_handle_client ... ok

successes:

---- handler::call::tests::test_handle_client stdout ----
Message { service_name: "add", data_len: 2, data: "1 2" }
结果 "3"


successes:
    handler::call::tests::test_handle_client
    handler::call::tests::test_match_service
    handler::call::tests::test_match_service_unknown_request
    handler::call::tests::test_parse_to_vec_u8

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 4.70s
```

3.2.4 加解密模块测试

3.2.4.1 测试用例编写

```
1       #[cfg(test)]
2       mod tests {
3           use super::*;
4
5           #[test]
6           fn test_encrypt_decrypt() {
7               let (priv_key, pub_key) = generate_keys();
```

```
        let message = "Hello, world!";
        let encrypted = encrypt_str(&pub_key, message);
        let decrypted = decrypt_u8(&priv_key,&encrypted);
    }

    #[test]
    fn test_encrypt_decrypt_u8() {
        let (priv_key, pub_key) = generate_keys();
        let message = "Hello, world!";
        let encrypted = encrypt_u8(&pub_key, message.as_bytes());
        let decrypted = decrypt_u8(&priv_key,&encrypted);
    }
}
```

3.2.4.2 测试结果分析

测试通过，能正常进行加解。

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.70s
     Running unittests src\lib.rs (target\debug\deps\crypto-eaaaf3666210d6d3.exe)

running 2 tests
test rsa_crypto::tests::test_encrypt_decrypt_u8 ... ok
test rsa_crypto::tests::test_encrypt_decrypt ... ok

successes:

successes:
    rsa_crypto::tests::test_encrypt_decrypt
    rsa_crypto::tests::test_encrypt_decrypt_u8

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 11.98s
```

第 4 章 总结与展望

4.1 作品总结

本项目所构建的 RPC 框架采用 HTTP 作为通信协议，确保了与 Web 技术的兼容性和网络穿透性；利用 JSON 进行数据的序列化与反序列化，实现了轻量级且广泛兼容的数据交换方式；服务模块的插件化设计提升了系统的可扩展性和灵活性，允许按需加载功能；集成 RSA 加解密技术加强了数据传输的安全性；IDL 接口描述语言的使用统一了不同服务间的接口定义，促进了多语言环境下的互操作性；远程服务管理中心集中处理服务的注册、发现和监控，提高了系统的管理效率；跨语言支持和多体系结构兼容性则保证了框架在不同开发环境和硬件平台上的高效运行。这些特色的综合应用，使得该 RPC 框架成为一个强大、灵活且安全的分布式系统构建工具。

4.2 未来展望

本次竞赛由于时间仓促参赛成员琐事缠身导致前期推进缓慢，在大家的齐心协力之下，使用 Rust 搭建起了一个比较简单的 RPC 框架，而且额外实现了数据加密的功能，在保证功能正常、效率较高的同时保证了用户的数据的隐私安全，并顺利通过了模块全功能冒烟测试，顺利完成初赛的目标。

如果能顺利晋级决赛，本项目团队将继续完成 IDL 接口描述语言设计、远程服务管理中心构建、跨语言支持、多体系结构支持等相关模块的设计和构建，并加入负载均衡使得所搭建的 RPC 项目在承担繁重的任务时具有较好的鲁棒性。

4.3 参赛选手感言

通过本次竞赛项目团队从零开始系统地学习了 Rust 语言，并深入学习了 RPC 原理，调研了多个成熟的 RPC 框架。在开发过程中，我们遇到了很多的困难，但是在老师的指导下，小组成员齐心协力共同攻坚克难，顺利搭建起了简单的 RPC 架构，完成了初赛的计划。

4.4 鸣谢

在本次红山开源赛题中，我们团队基于 Rust 语言初步完成了 RPC 设计和实现的项目。在此过程中，我们得到了多方面的支持与帮助，特此表示衷心的感谢。

感谢红山组委会为我们提供了这个展示创新和技术实力的平台。没有这样的舞台，我们的工作将无法得到如此广泛的关注和认可。对于 Rust 社区，我们表示深深的敬意和感激。是你们的不懈推广和技术分享，让 Rust 这门优秀的语言得以在全球范围内蓬勃发展。

感谢任老师和贾老师的辛勤指导，老师们极强的指导如同指明灯为我们在黑暗中照亮了前进的道路。同时感谢所有参与项目讨论、提供反馈和建议的团队成员，是你们的智慧和努力让这个项目得以不断完善。

全员敬上！