目 录

[第1章 网络应用编程基础知识 8](#_Toc4279)

[1.1 引言 8](#_Toc18483)

[1.2 Rust集成开发环境 9](#_Toc11428)

[1.2.1 简介 9](#_Toc29654)

[1.2.2 安装Rust集成开发环境 11](#_Toc16673)

[1.2.3 使用cargo命令创建第一个Rust项目 12](#_Toc12111)

[1.3 网络应用编程模型 14](#_Toc26244)

[1.3.1 C/S架构 14](#_Toc28630)

[1.3.2 B/S架构 15](#_Toc27259)

[1.3.3 客户机和服务器的交互 16](#_Toc3184)

[1.4 TCP/IP网络协议 17](#_Toc12249)

[1.4.1 套接字 17](#_Toc11512)

[1.4.2 TCP/IP体系结构 20](#_Toc13816)

[1.4.3 IP地址、MAC地址与端口 22](#_Toc13962)

[1.4.4 DNS域名解析 25](#_Toc19471)

[1.5 网络通信服务 27](#_Toc18433)

[1.5.1 面向连接的服务 27](#_Toc22556)

[1.5.2 无连接服务 28](#_Toc12200)

[1.6 Rust网络编程特征 29](#_Toc5727)

[（1） 友好的开发环境 29](#_Toc19288)

[（2） 支持高并发 29](#_Toc7548)

[（3） 零成本抽象 29](#_Toc3733)

[（4） 高可靠性 30](#_Toc5295)

[（5） 轻量级 30](#_Toc1632)

[1.7 小结 30](#_Toc13509)

[第2章 Rust初步 31](#_Toc12508)

[2.1 引言 31](#_Toc12723)

[2.2 概述 31](#_Toc28420)

[2.2.1 特色 32](#_Toc23039)

[2.2.2 基本成分 33](#_Toc18675)

[2.3 所有权、借用与生存期 38](#_Toc25177)

[2.3.1 所有权 38](#_Toc16788)

[2.3.2 借用 42](#_Toc18195)

[2.3.3 生存期 45](#_Toc7348)

[2.3.4 内存不安全示例 47](#_Toc14948)

[2.4 泛型与特型 52](#_Toc21199)

[2.4.1 泛型与代码复用 52](#_Toc3467)

[2.4.2 特型与抽象 55](#_Toc2060)

[2.5 错误处理 65](#_Toc13398)

[2.5.1 可恢复错误 65](#_Toc18497)

[2.5.2 不可恢复错误处理 68](#_Toc15770)

[2.6 并发处理 70](#_Toc6135)

[2.6.1 线程管理 70](#_Toc12436)

[2.6.2 线程间消息传递机制 74](#_Toc10959)

[2.6.3 共享状态并发 75](#_Toc32348)

[2.7 测试 78](#_Toc20908)

[2.7.1 调试 79](#_Toc28613)

[2.7.2 测试 82](#_Toc1936)

[2.8 小结 86](#_Toc31582)

[第3章 数据序列化、反序列化和解析 87](#_Toc31252)

[3.1 引言 87](#_Toc3195)

[3.2 使用Serde库实现序列化和反序列化 87](#_Toc4830)

[3.2.1 序列化和反序列化 87](#_Toc19297)

[3.2.2 自定义序列化和反序列化 94](#_Toc29057)

[3.3 解析文本数据 102](#_Toc14357)

[3.4 解析二进制数据 108](#_Toc429)

[3.5 小结 112](#_Toc13403)

[第4章 异步编程 113](#_Toc30621)

[4.1 引言 113](#_Toc16100)

[4.2 异步编程软件包 113](#_Toc3356)

[4.2.1 futures包 113](#_Toc28863)

[4.2.2 tokio程序包 121](#_Toc27553)

[4.3 异步网络编程 134](#_Toc6028)

[4.3.1 stream和sink程序包 134](#_Toc21578)

[4.3.2 tokio套接字复用 142](#_Toc32513)

[4.4 示例：流协议 151](#_Toc9969)

[4.5 小结 163](#_Toc6744)

[第5章 TCP和UDP应用编程 164](#_Toc7988)

[5.1 引言 164](#_Toc21511)

[5.2 std∷net命名空间和crates单元包程序 164](#_Toc8260)

[5.2.1 std∷net命名空间 164](#_Toc7322)

[5.2.2 Crates网络编程 170](#_Toc1383)

[5.3 TCP编程的基本概念 180](#_Toc21646)

[5.3.1 编程特点 180](#_Toc28289)

[5.3.2 TcpListener 和 TcpStream 180](#_Toc14676)

[5.4 示例：TCP协议 181](#_Toc4041)

[5.5 UDP编程的基本概念 188](#_Toc2316)

[5.5.1 编程特点 188](#_Toc19124)

[5.5.2 UdpSocket 189](#_Toc13875)

[5.6 示例：UDP协议 189](#_Toc26950)

[5.7 使用tokio::net进行网络编程 193](#_Toc21503)

[5.7.1 tokio::net::TcpListener 194](#_Toc25253)

[5.7.2 tokio::net::TcpSocket 195](#_Toc105)

[5.7.3 TcpStream 196](#_Toc10499)

[第6章 HTTP应用编程 201](#_Toc693)

[6.1 引言 201](#_Toc12169)

[6.2 Hyper框架 202](#_Toc7683)

[6.2.1 服务器编程 202](#_Toc19366)

[6.2.2 HTTP 客户端编程 210](#_Toc29680)

[6.3 Rocket框架 219](#_Toc5740)

[6.3.1 监听web请求 222](#_Toc4860)

[6.3.2 发送HTTP响应 226](#_Toc4399)

[6.4 Reqwest库 231](#_Toc7358)

[6.4.1 Reqwest库简介 231](#_Toc16567)

[6.4.2 安装与引用 232](#_Toc31181)

[6.4.3 一些常见的HTTP请求方法 233](#_Toc17085)

[6.4.4 JSON 237](#_Toc8645)

[6.4.5 cookie 239](#_Toc21463)

[6.4.6 Tokio 240](#_Toc6676)

[6.5 小结 243](#_Toc22714)

[第7章 其他应用层协议编程 244](#_Toc26144)

[7.1 引言 244](#_Toc4609)

[7.2 远程过程调用RPC 244](#_Toc9047)

[7.2.1 RPC和协议缓冲区 244](#_Toc15579)

[7.2.2 gRPC 249](#_Toc8808)

[7.3 SMTP应用编程 259](#_Toc16383)

[7.3.1 web邮件服务 259](#_Toc15552)

[7.3.2 简单电子邮件系统 262](#_Toc19795)

[7.3.3 示例 268](#_Toc24020)

[7.4 FTP和TFTP应用编程 273](#_Toc2475)

[7.4.1 FTP服务应用与实现 273](#_Toc95)

[7.4.2 TFTP服务应用与实现 277](#_Toc16252)

[7.5 小结 282](#_Toc1694)

[第8章 安全保障 283](#_Toc1528)

[8.1 引言 283](#_Toc19257)

[8.2 利用证书保障Web应用的安全性 284](#_Toc11643)

[8.2.1 Rust 中常用的数字证书库和工具 284](#_Toc15574)

[8.2.2 在 Rust 中使用 TLS/SSL 进行加密通信 286](#_Toc10591)

[8.2.3 服务器证书的验证与管理 287](#_Toc13352)

[8.2.4 防御网络攻击的常见方法 288](#_Toc8373)

[8.2.5 自动化证书部署和更新 289](#_Toc8898)

[8.3 利用Ring Crate进行密钥交换 291](#_Toc14893)

[8.3.1 对称加密算法 291](#_Toc32045)

[8.3.2 非对称加密算法 295](#_Toc2170)

[8.3.3 密钥派生函数 295](#_Toc4117)

[8.3.4 消息认证码 297](#_Toc8358)

[8.3.5 随机数生成器 299](#_Toc4976)

[8.3.6 小结 300](#_Toc1395)

[8.4 示例：安全应用开发 300](#_Toc10262)

[8.5 小结 303](#_Toc3697)

# 网络应用编程基础知识

本章要点：熟悉Rust集成开发环境，认识网络应用程序交互方式，理解TCP/IP网络协议栈的基本思想，掌握IP地址转换、DNS域名解析和套接字等相关类的使用方法，熟悉常见的网络服务模型，理解网络应用程序的执行过程。

## 引言

Rust 是一种快速、高并发、安全且具有授权性的编程语言，特别适合用来处理复杂的系统任务。Rust语言最初由 Graydon Hoare 于2006 年创建和发布，现在它是一种开源语言，主要由 Mozilla 团队和众多开源社区成员共同维护和开发。

与C#一样，Rust也是依托.NET框架提供网络应用编程服务。Rust的标准库std::net提供了对 TCP/IP协议栈使用的封装，对同步网络应用编程进行支持，而如果要进行异步高性能网络应用编程，就需要使用Rust的异步编程库tokio::net，其依赖于mio、Future、async/await等机制的应用，开发者可以让代码在产生极高生产力的同时，保持网络应用程序的性能与C语言一致。std::net和tokio::net的对外api几乎一模一样，这样做的好处是，在保证程序生产效率的同时，进一步降低了程序开发的难度。官方的测试结果表明，得益于编译期内存安全、零成本抽象和支持高并发等内生技术和措施，Rust网络编程框架的性能相比Java和GO语言均有明显提升，开发的高性能网络应用程序的运行速度甚至比C#的更快。

依据我们上述的介绍，Rust仿佛是各种语言集大成者，那么利用它进行网络编程会不会很困难呢？答案是否定的。

首先，Rust的编译器非常好用。与C和C++相比，在编译代码时，Rust编译器给出的错误原因相当简洁明了，可读性和可理解性更强，而且还可以给出改正错误的行之有效的建议和方法。因此在写代码时不用担心出错，Rust编译器会给你提供解决问题的最佳方法，也可以这样说，试错正是学习这门语言的有效方法之一。此外，Rust编译代码的速度特别快。Rust 源码经过分词和解析，生成AST（Abstract Syntax Tree），然后把 AST 进一步简化为 HIR(High-level Intermediate Representation)，之后HIR被进一步编译为 MIR（Middle IR），以便缩短编译时间、执行时间，以及进行更精确的类型检查。最终，MIR 会被翻译成LLVM IR（Low Level Virtual Machine IR），然后被 LLVM 编译为能在各个平台上运行的安全且高效的机器代码。

其次，Rust提供了对WebAssembly 的支持。除了上述介绍的Rust本身提供的丰富的用于网络编程的各种库文件（std::net和tokio::net等）之外，WebAssembly （Wasm）的加持更加丰富了Rust的Web应用程序开发工具。Wasm是一种高效的、低级别的编程语言，更类似一种低级的汇编语言， 具有运行高效、内存安全、无未定义行为和平台独立开放等特点，支持在 Web 上快速部署客户端和服务器应用程序，作为JS的扩展可以方便地调用各种浏览器API用于项目开发，更有JS所不具备的多种高性能组件，如编码工具，图像音频处理，数据分析等。 Wasm被设计为编程语言的可移植编译目标，可以使用Rust编写程序，然后将其编译成WebAssembly，进而生成一个加载和执行速度非常快的Web应用程序。

最后，Rust是C语言的升华版。Rust提供了类似C和C++风格的语法机制，又弥补了C语言表达能力的不足之处。C++ 的调用形式无法表达传参类型，导致其表达混乱。但Rust的函数声明和调用形式不仅可以完全一一对应，语法、语义也更加简单明了。Rust面向对象编程本质上就是C语言的Struct。Rust中的许多新技术都是对C和C++的补充和升华，可安全且无缝地与 C 和 C++ 语言沟通。因此在掌握了C和C++的基本编程方法的基础之上，学习Rust是比较容易的，并且有助于更深刻地理解程序设计的方法、技术和理念。

综上所述，Rust语言拥有广泛的特性，利用Rust语言进行网络应用编程，可以构建高性能、高并发的网络应用程序，代码易于维护、调试，并且更安全更高效，可支持大型网络系统的开发和快速部署。

## 网络应用编程模型

软件架构为软件系统提供了一个结构、行为和属性的高级抽象，由组件的描述、组件的相互作用、指导组件集成的模式以及这些模式的约束组成。软件架构不仅显示了软件需求和软件结构之间的对应关系，而且指定了整个软件系统的组织和拓扑结构，提供了一些设计决策的基本原理。C/S（Client/Server，客户端/服务器）和B/S（Browser/Server，浏览器/服务器）是分布式的网络应用软件通常采用的两种架构方式。

### C/S架构

C/S架构是一种典型的两层架构，分为客户端软件和服务器端软件，如图1.1所示。客户端需要实现大部分的业务逻辑和界面展示，包含一个或多个在用户电脑上运行的进程，为用户提供操作界面和请求服务的接口。客户端可以随时提出请求，获得服务器提供的服务，可以随时关机离开，一次请求与服务的过程是客户端首先发起的。服务器端运行服务进程，服务进程以轮询的方式监听服务器各端口上进来的服务请求，被动地接受并处理来自客户端的请求，然后将处理结果返回给客户端。服务器一经启动，除特殊情况外将不再关机。相对客户端来说，服务器端的业务相对单一，这样可以充分利用本地资源完成软件系统的很多工作，以减轻服务器的压力，因此C/S架构也被称为胖客户端编程架构。从形式上来说，采用C/S架构的软件的客户端拥有不同于其他软件的专门的用户界面，因此非常好辨识，例如QQ、微信、火绒、爱奇艺等都有专属于自己的用户界面，它们都属于C/S架构软件。

C/S架构由于只有一层交互，所以响应速度较快，但是维护成本较高，一次软件升级，需要所有的客户端重新安装软件，所以C/S架构更适合于需要快速响应，安全需求高，用户群体相对固定的中小型网络应用软件。本书主要采用C/S架构进行网络应用编程，交互过程如图1.1所示。

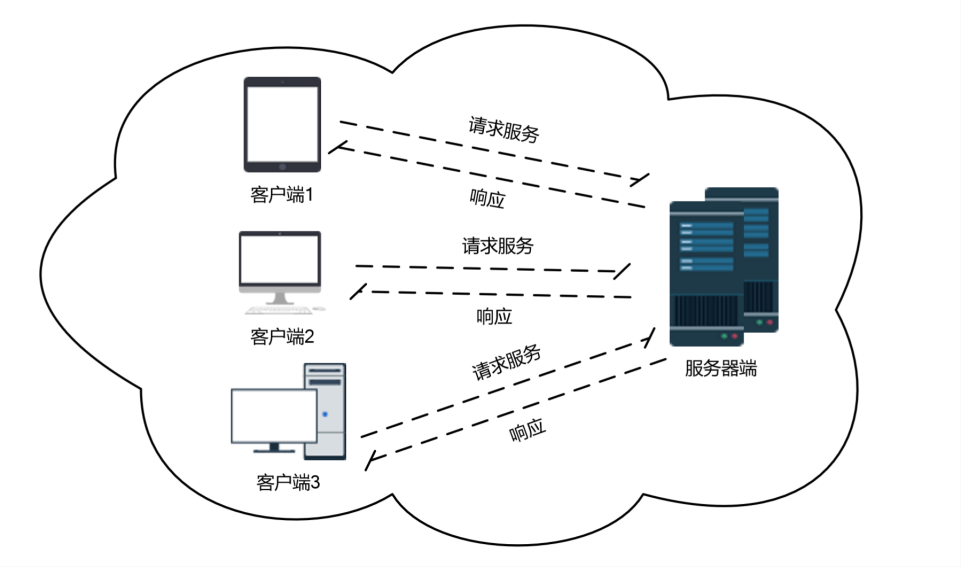


图1.1 C/S架构

### B/S架构

B/S是三层架构设计，分为浏览器客户端，Web服务器端和数据库服务器端，如图1.2所示，其中浏览器客户端主要负责显示用户界面，绝大部分业务逻辑的实现都放在Web服务器上，数据库服务器提供数据支持，包括对数据的存储、增删改查等操作的支持。B/S架构软件系统的客户端就是通用的web浏览器，输入网址后，由HTTP协议提供支持，即可访问web服务器的内容，因此，客户端只需要处理很少的事务逻辑，从而减少了客户端的压力，因此B/S架构又被称为瘦客户端编程架构。凡需要在浏览器中输入网址访问服务器进行操作的软件都是B/S架构的软件。

由于B/S架构的业务处理主要在web服务器上完成，客户端只是通过web浏览器发出业务请求，并接收业务处理结果，因此与可以在本地进行大部分事务处理的C/S架构相比，处理速度相对较慢。BS架构部署和维护相对简单，无需逐个升级所有的客户端，只需要一次性地升级服务器即可，并且可通过一定的权限控制实现多客户访问的目的，交互性较强，交互过程如图1.2所示。



图1.2 B/S架构

### 客户机和服务器的交互

不管是C/S架构还是B/S架构，网络应用软件都有两个主体，一个是客户机，另一个是服务器，这两个主体分别由运行在不同计算机上的两个应用进程（客户机进程和服务器进程）承担。网络应用进程通信时，通常采用客户机/服务器交互模式。客户机主动向服务器发出服务请求，服务器被动地接受请求，处理请求，反馈结果以提供服务，具体工作过程如图1.3所示。

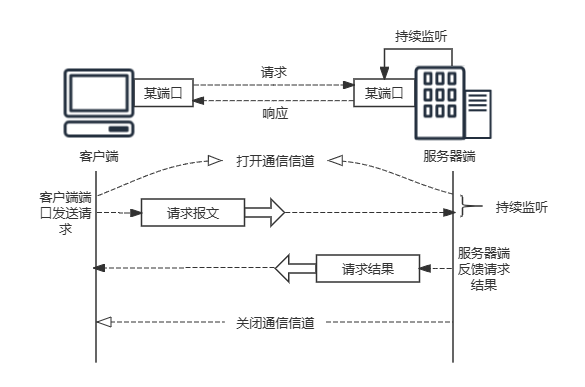


图1.3 客户机/服务器交互过程

客户机/服务器交互过程中，客户机进程始终处于主动地位，主动向服务器发出请求，主动终止服务，其具体工作过程如下。

1. 打开一通信信道，并与服务器的某特定监听端口建立连接。
2. 向服务器的特定监听端口发出请求报文，等待并接受来自服务器的应答，之后重复此步骤。
3. 请求结束，关闭通信信道，终结与服务器的交互。

客户机/服务器交互过程中，服务器进程始终处于被动服务的地位，服务器要先启动（一经启动，除特殊原因不再关闭），并开始持续监听特定端口上是否有服务请求，如果有，就接受请求，并将请求的处理结果反馈给客户端，其具体工作过程如下。

1. 打开一通信信道，并告知所在主机的操作系统，它需要在某一端口上接收来自客户机的特定的服务请求。
2. 持续地监听端口，直到有请求到达。
3. 接受服务请求，处理请求并向客户机进程反馈该请求的处理结果，服务完成，关闭客户机和服务器进程间的通信信道。
4. 返回第②步，等待并处理下一个客户请求。

从上述描述可知，客户机进程和服务器进程都属于高层的应用层进程，它们之间的交互过程依赖于网络协议栈提供的底层支持。客户机和服务器的角色并不是固定不变的，一台计算机可能既是客户机又是服务器，比如，计算机1上的进程A向计算机2上的进程B发出服务请求，则计算机1为客户机，计算机2为服务器；之后，计算机2上的进程C向计算机1上的进程D发出服务请求，则此时，计算机1变成了服务器，计算机2变成了客户机。

## TCP/IP网络协议

网络协议是控制双方或多方之间进行通信的规则的集合，规定了发送者和接收者之间所交换的信息的格式，以及要完成的操作。TCP/IP网络协议(Transmission Control Protocol/Internet Protocol)指能够在多个不同网络间实现信息传输的协议簇，包含了一系列构成互联网基础的网络协议，是互联网使用的核心协议。

### 套接字

在网络应用编程中，套接字（socket）是网络通信的基本操作单元，用来描述通信端点，提供了应用层进程利用网络协议交换数据的机制。针对某台主机上的某个进程与另外一台主机上的另外一个进程的通信，这两个进程就是通信的端点，那么如何将这两个端点描述出来呢？因为进程位于主机上，所以只要明确标识出这个通信端点所在的主机和进程，就可以准确地标识这个通信端点了。正如上一节所述，主机用IP地址标识，进程用端口号标识，因此，套接字用IP地址连接端口号构成，如公式1-1所示。

socket =(IP地址︰端口号) （1-1）

TCP是面向连接的协议，每一条TCP连接被相互通信的两个端点唯一确定，因此一条TCP连接可由公式1-2定义。

TCP连接∷={socket1, socket2}={(IP1∶port1), (IP2∶port2)} （1-2）

就TCP/IP协议栈来说，套接字隶属于TCP/IP协议栈，但却是独立于各层协议的网络编程接口，处于应用层和运输层之间，它向上承接应用进程，向下连接运输层协议，如图1.4所示。由于目前TCP/IP协议栈被实现于操作系统内核，因此套接字也成为操作系统内核的一部分。在进行网络应用编程时，使用socket系统调用，控制权从应用进程转交给操作系统，操作系统为应用进程提供网络通信服务；网络通信结束后，关闭socket，控制权再从操作系统转给应用进程。因此，在网络应用编程中，套接字已成为应用进程为获取网络通信服务而与操作系统交互的一种机制。



图1.4 分层协议中的socket套接字

套接字有三种不同的类型：流式套接字（SOCK\_STREAM），数据报套接字（SOCK\_DGRAM），原始套接字（SOCK\_RAW）。其中，流式套接字使用TCP协议，提供面向连接的、可靠的数据传输服务；数据报套接字使用UDP协议，提供无连接且不保证可靠性的数据传输服务；原始套接字实现IP数据包通信，实现非TCP和UDP协议的数据通信。在网络编程中，要进行网络通信就必须创建网络连接，而套接字作为网络连接中端点的标识必须被首先定义，在套接字定义中必须指明套接字类型，以便让操作系统能够识别应用此套接字的网络连接要使用的通信协议。

以下面向客户/服务器通信模型，以TCP连接服务为例，说明基于套接字通信中的系统调用。

服务器端：

第1步创建套接字： int socket(int domain, int type, int protocol)

socket函数在服务器端创建一个新套接字和一个套接字数据结构，并返回一个整型套接字描述符，其中有指针指向套接字数据结构。套接字数据结构中存放分配给这个套接字的各种资源（如存储空间、CPU时间等）、客户端和服务器端各自的IP地址和端口号、协议族和服务类型等。domain参数指明套接字协议族，可取如下值：PF\_INET（TCP/IP协议族），AF\_UNIX（用于同一台计算机的进程间通信），AF\_INET6（IPv6网络协议）；type参数指定套接字类型，可取如下值：SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW；protocol参数指定数据包的特定协议类型，可取如下值：IPPROTO\_TCP、IPPROTO\_UDP、IPPROTO\_ICMP等，如果type指定的协议中只有一种特定的协议，则protocol取默认值0。

第2步绑定：int bind(int socket, const struct sockaddr \*address, size\_t address\_len)

bind函数将套接字与本地IP地址和端口号绑定在一起，即把本地IP地址和端口号写入套接字数据结构。socket参数为套接字描述符；address参数指明了IP地址和端口号；address\_len参数指明套接字地址结构的长度。

第3步侦听连接请求：int listen(int socket, int backlog)

Listen函数在指定的套接字上侦听客户端连接请求，并创建一个队列来缓存未处理的连接。socket参数指明在其上侦听连接请求的套接字的描述符；backlog指明连接队列的最大长度。

第4步接受连接：int accept(int socket, struct sockaddr \*address, size\_t \*address\_len);

accept函数创建一个新的套接字和套接字数据结构，把bind函数调用之后的套接字数据结构复制到这个新的套接字的数据结构中，再在其中填入客户端的IP地址和端口号等信息，形成信息完全的套接字数据结构，最后返回套接字描述符。至此，这个新套接字就可以为客户端和服务器端的通信服务了。

第5步释放连接：int close(int socket)

close函数在通信结束后释放连接并撤销套接字及分配给它的所有资源。

客户端：

第1步创建套接字：int socket(int domain, int type, int protocol)

同于服务器端的socket函数，在客户端创建一个新套接字和一个套接字数据结构，并返回一个整型套接字描述符，其中有指针指向套接字数据结构。

第2步请求连接服务器：int connect(int socket, const struct sockaddr \*address, size\_t address\_len);

connect函数将参数socket指定的客户端套接字连接到参数address指定的服务器套接字上，如果connect函数成功执行，将建立起客户端和服务器之间的连接，之后就可以用该连接所关联的一对套接字进行双向数据通信了。

第3步释放连接：int close(int socket)

释放连接，并撤销套接字及其分配给它的所有资源。

图1.5以TCP连接服务为例描述了客户端和服务器端通信时，系统调用的顺序。由于UDP是无连接服务，所以在UDP通信时没有listen和connect系统调用。



图1.5 TCP通信时序图

### TCP/IP体系结构

TCP/IP体系结构分为四层，从上到下依次为：应用层（Application Layer）、运输层（Transport Layer）、网际层（Inter-network Layer）和网络接口层（Host-to-Net Layer），图1.6简要示意了这四层协议构建网络的方法，注意，协议的最终表现形式是安装在终端设备和网络设备中的软件。需要注意的是，路由器作为在互联网中进行数据转发的主要设备，只包含了TCP/IP体系结构中的网际层和网络接口层两层协议。

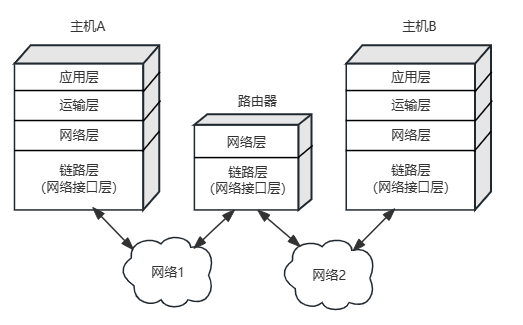


图1.6 TCP/IP体系结构

由上图知，TCP/IP协议栈的每一层都完成了某些特定的功能，对于上一层来说，下一层完成的功能就成为可为上一层提供的服务。各层间的关系为：通过各层间的接口，下一层为上一层提供服务，上一层调用下一层的服务，每一层协议的实现依赖于其下层提供的服务。针对软硬件设施和应用场景等的不同，TCP/IP协议栈的每一层都设置了多种协议来实现其功能，各层的功能、常用协议和软硬件设施具体如表1-1所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 表1-1 TCP/IP体系结构各层功能、协议、软硬件设施 | | | |
| 层名 | 功能 | 主要协议 | 软硬件 |
| 应用层 | 标识进程、数据格式化以及与所有应用程序的交互 | FTP （文件传输协议）  HTTP（超文本传输协议）  DNS（域名服务器协议）  SMTP（简单邮件管理协议）  SNMP（简单网络管理协议）  TELNET（远程登录服务协议） | 应用进程  应用层网关 |
| 运输层 | 负责进程到进程之间的通信，包括面向连接的可靠的有序通信（TCP）和无连接的随机序通信（UDP） | TCP（传输控制协议 ）  UDP（用户数据报协议） | 传输层网关 |
| 网际层 | 负责计算机到计算机之间的通信，实现寻址和路由选择 | IP（网际协议）  ICMP（网际控制报文协议）  IGMP（网际组管理协议）  ARP（地址解析协议）  RIP（路由信息协议）  OSPF（开放最短路径优先）  BGP（边界网关协议） | 路由器、防火墙、多层交换机 |
| 网络接口层 | 负责介质访问控制，处理比特的传输和接收、以及编码 | 802.3（以太网协议）  802.5（令牌环网协议）  HDLC（高级数据链路控制协议）FDDI（光纤分布式数据接口） | 网卡、中继器、集线器、网桥、以太网交换机 |

在Rust网络应用编程中，主要是利用.NET框架提供的各种技术、库和服务包，实现应用层和运输层的某些特定协议，建立客户端和服务器之间的通信，在此基础之上，再利用Rust语言构建实现符合实际需求的各种应用系统。

### IP地址、MAC地址与端口

网络应用编程的第一步就是要明白通过网络进行数据交互的各方都是谁，并把他们标识出来。比如两个人通过微信聊天，在用代码实现这一过程时，就需要知道是哪两台设备上的微信程序进行通信，所以在代码中必须标识出设备和微信进程。那么在网络应用编程时，该如何标识它们呢？所有的编程语言都是用IP地址标识设备，用端口号标识进程。

#### IP地址

IP地址是用于在全球范围内唯一标识互联网上每一台主机（或路由器）的每一个接口的标识符，是TCP/IP协议栈中网络层使用的地址。IP地址分为IPV4地址和IPV6地址。IPv4地址被定义为32位二进制数，为便于记忆，将其每8位分成一段，转换为十进制数，段和段之间用小数点分隔，这就是点分十进制记法，如图1.7（a）所示。但在上世纪90年代，爆发了IP地址短缺危机，因此在1995年又提出了128位的IPv6地址，使用冒号十六进制记法，把每16个二进制位的值用一个十六进制数表示，各数之间用冒号分隔，如图1.7（b）所示。目前的操作系统，如Windows、Linux、Android、HarmonyOS等，都支持这两种地址协议。



（a）PV4地址



（b）PV6地址

图1.7 IP地址

IP地址的设立就是为了在浩瀚的网络中快速寻找到某一台主机，为便于快速寻址，IP地址内部又分为两部分：前一部分用于识别主机所在的网络，称为网络号；后一部分用于标识网络中的主机，称为主机号。图1.5（c）以IPv4地址为例表示出IP地址的结构。网络号在整个互联网范围内必须是唯一的，可在一定程度上标识该网络所在的地理位置，主机号在所连接的网络(即前面的网络号所指明的网络)中必须是唯一的，这种二级地址结构非常有利于简化网络结构并进行互联网范围内的快速寻址，因为互联网是网络与网络相连构成的更大网络，因此在寻找某一台主机时，首先找到其所在的网络，再在网络中找到这台主机，要比直接找到主机快得多的多（为什么？）。现在的问题是，如何明确地表示出网络号的位数并计算出网络号也就是网络地址呢？无分类编址（CIDR）是目前互联网使用的编址方法，它用斜线记法，即在IP地址后加上斜线和十进制数字的方法，明确表示出网络号的位数。在CIDR中，把网络号改称为网络前缀，因此斜线后的数字即是网络前缀的位数。例如：IPv4地址192.168.68.24/26的网络前缀为26位，则主机号就有6位。下面的问题来了，计算机并不知道斜线计法，那么计算机是如何从CIDR地址中计算出网络地址的？子网掩码解决了这个问题。

子网掩码由 一 连串1接着 一 连串0组成，其中1 的个数就是网络前缀的长度，即斜线后的数字，同上例：192.168.68.24/26的子网掩码为11111111111111111111111111000000。将32位二进制地址与子网掩码按位进行与运算后就得到了网络地址，点分十进制表示为：192.168.68.0，如图1.8所示。

图1.8 CIDR网络地址计算过程

CIDR中把具有相同网络地址的所有IP地址组成一个地址块，其中最小的IP地址（网络地址后连接连续的多个0的地址）就是这个地址块的网络地址，也就是子网掩码与IP地址的与运算的结果。此网络中可容纳的主机数为（d是主机位数，最小主机号0...0分配去构成网络地址，最大主机号1...1对本网络的广播地址）。例如192.168.68.24/26的网络地址为（其中最后一个字节二进制展开）：192.168.68.00000000，可容纳的主机号的变化范围：192.168.68.00000001-192.168.68.00111110，本网络的广播地址为：192.168.68.00111111。CIDR可灵活构建适合用户需要的地址块，尽可能减少地址空间的浪费。

IP地址实质上标识的是一台设备连接到网络的接口，因为普通用户设备只用一个接口（网卡）连接到网络，所以也可以简单地说IP地址标识了网络上的设备。但是某些设备，比如路由器，是将不同的网络连接起来的，那么它就会有两个、甚至更多的接口（网卡）连接到网络，每一个接口都要用IP地址标识，所以该设备就会有多个IP地址。一般情况下，一个设备通过几个网卡连接到网络就会有几个IP地址。另外，目前大部分设备的IP地址是由动态主机配置协议（DHCP）自动分配的，DHCP会依据固定不变的。

现在我们再来讨论这样一个问题：在互联网中，数据是如何依据IP地址在通信的两个端点之间传输的呢？这就是大名鼎鼎的IP路由问题。互联网是网络的网络，网络和网络之间用路由器连接（[Router](https://baike.baidu.com/item/Router/2105729?fromModule=lemma_inlink)），如图1.9所示。

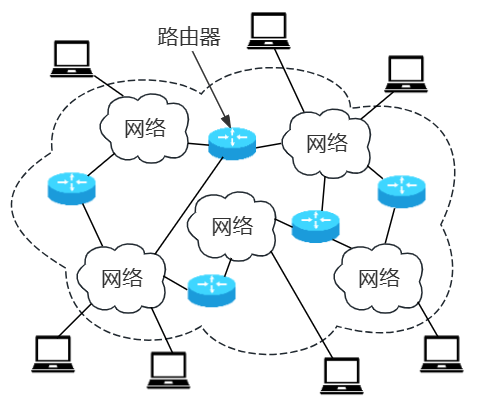


图1.9 路由器连接的网络

每一个路由器都维护着一张路由表，如表1-2所示，路由表是依据路由选择协议计算而来。常用的路由选择协议有RIP、OSPF、BGP等，由于路由选择协议已超出了本书所讨论的范围，这里不再赘述。依据路由表生成转发表，转发数据包的过程如下：路由器读取[数据包](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%8C%85/489739?fromModule=lemma_inlink)中的IP地址，按照从上到下的顺序，将此IP地址与转发表每一行中的目的网络的子网掩码按位相与，如果计算结果与目的网络一致，则将数据包发送给本行所标识的下一跳接口；如果计算结果与目的网络不一致，则按照上述方法依次与路由表中的第二、三……行比对，直到目的网络匹配成功，如果直至转发表最后一行都没有匹配上目的网络，说明此数据包不可转发，将被丢弃。把数据包发送到下一跳路由器，实质上就是把数据包发送到下一个网络，在下一个路由器中重复上述转发过程，直至到达目的网络，把数据包交付给目的主机。至此，一次端到端的数据通信过程结束。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表1-2 路由表 | | |  | 表1-3 转发表 | |
| 目的网络 | 距离 | 下一跳路由器 | 前缀匹配 | 下一跳 |
| Net2 | 5 | R2 | 128.1.2.192/26 | 直接，接口1 |
| Net5 | 2 | R4 | 128.1.2.128/26 | 直接，接口2 |
| … | … | … | 128.1.2.64/26 | R2，接口3 |

#### MAC地址

MAC地址是网络接口层的硬件设备网卡的地址，在制作网卡时就被厂家固化在了网卡的 ROM中了，因此不会随着网卡地理位置的改变而改变，所以它又被称为硬件地址或物理地址。MAC地址在世界范围内是唯一的，标识发送和接收数据的网卡。

MAC地址被定义为48位二进制数，为便于书写，48位二进制数从左至右每8位（一个字节）分成一段，用2个十六进制数表示，所以统共用12个十六进制数来表示1个MAC地址，如图1.10所示。

图1.10 MAC地址表示

MAC地址的前3个字节代表网络硬件制造商的编号，由[IEEE](https://baike.baidu.com/item/IEEE/150905?fromModule=lemma_inlink)(电气与电子工程师协会)分配，后3个字节由制造商自行设定的、对其所生产的某个网络产品(如网卡)的编号。

有了IP地址和MAC地址就可以通信了吗？还不行，还要标识出数据产生自哪里。数据由进程产生，因此在网络通信中还要标识进程，端口号就是进程的标识。

#### 端口号

端口号标识的是软件端口（不同于路由器等设备上的硬件端口），是应用层的进程与运输层实体进行交互的端口的编址。通俗地讲就是，在发送方：应用进程把数据发送到适当的端口，然后运输层从该端口读取数据；在接收方：运输层是依据目的端口号把数据交付给应用层的目的进程的。所以端口更像是运输层和应用层之间的接口，端口号就是端口的标识。

TCP/IP协议规定端口号为16位，可标识65535个端口，其中0~1024以内的为全球通用端口号，表1-2列出了一些网络应用中最常用的端口号。1024~49151为登记端口号，必须在IANA（互联网号码指派管理局）登记后才可使用。49152~65535为客户可使用的端口号，程序员可以将这部分端口号临时分配给应用进程。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表1-4 常用的熟知端口号 | | | | | | | | | |
| 应用程序 | FTP | TELNET | SMTP | DNS | TFTP | HTTP | SNMP | SNMP (trap) | HTTPS |
| 熟知端口号 | 21 | 23 | 25 | 53 | 69 | 80 | 161 | 162 | 443 |

### DNS域名解析

正如前所述，无论是32位的IPv4地址还是128位的IPv6地址，对于人类来说，记忆这些“01”串都是异常困难的，更何况，随着互联网的发展，网络上的设备和服务越来越多，要记住所有的IP地址更是不可能。相较而言，字符形式的地址更便于人类记忆（例如：百度的字符地址是“www.baidu.com”，IP地址是“182.61.200.6”，用户在要使用百度时，一般都是在浏览器中输入前者，而后者却少有人知道）。然而，计算机却只能识别IP地址。为便于人类记忆，同时又能让计算机理解，TCP/IP协议栈的应用层增加了DNS（Domain Name System）域名系统，提供字符地址到IP地址的映射，即域名解析服务。

互联网的DNS被设计成联机分布式数据库系统。DNS使大多数名字都在本地进行解析（resolve），仅少量解析需要在互联网上通信，因此DNS的效率很高，具体地址解析过程如图1.11所示。



图1.11 域名解析过程

1. 当用户在浏览器中输入字符地址并回车后，浏览器首先检查自己的DNS缓存，如果在其中找到字符地址的相应条目，并且未过期，则取出条目中的IP地址，供浏览器使用，地址解析到此结束。
2. 否则，浏览器将检查操作系统的DNS缓存，如果在其中找到对应条目，并且未过期，则取出条目中的IP地址，供浏览器使用，地址解析到此结束。
3. 如果仍然没有找到相应的条目，浏览器将发起一个DNS系统调用，以客户端的身份向本地域名服务器（由Internet服务提供商自动配置或者用户手动配置）发送DNS请求，DNS服务器接收到请求后，检索域名文件，如果在其中找到字符地址对应的条目，并且未过期，则将条目中的IP地址返回给发起域名解析请求的浏览器，地址解析到此结束。
4. 否则，本地域名服务器将在自己的域名文件中查找根域名服务器的IP地址，并代理浏览器向其发起域名解析请求。然后根域名服务器向本地域名服务器返回一个包含查询域的顶级域名服务器的IP地址。
5. 本地域名服务器向上一步返回的顶级域名服务器发送相同的域名解析请求，顶级域名服务器向本地域名服务器返回域名解析请求对应的域名服务器（域名服务提供商的服务器）的IP地址。
6. 本地域名服务器向上一步返回的域名服务器发送同一个的域名解析请求，域名服务器在自己的域名文件中检索字符地址对应的条目，如果找到且未过期，将其中的IP地址返回给本地域名服务器。
7. 本地域名服务器再将上一步得到的IP地址返回给发起域名解析请求的浏览器，地址解析结束。

由于域名解析是个很小的问题，所以DNS请求报文和回复报文都很短，并不需要高可靠的网络传输，因此一般情况下使用UDP用户数据报承载域名解析请求和回复报文。但是大多数的DNS协议实现中仍然包含了一个选项，使得在网络状况非常糟糕的情况下，可以使用TCP报文承载域名解析服务。

## 网络通信服务

Internet为主机通信提供了两种可选择的服务：面向连接的服务和无连接服务，每一种服务都有其适合的场景。从TCP/IP协议栈的角度来说，TCP协议支持面向连接的服务，提供可靠的数据传输保障，即确保数据不出错、不丢失、不重复、不失序；UDP协议支持无连接服务，提供简单快速的数据通信服务，但不保证数据传输的可靠性。网络应用的设计者可根据主机间数据传输的可靠性和效率等特性的不同要求选择不同的服务。

### 面向连接的服务

面向连接的服务要求通信双方在交换数据之前，必须先建立连接。连接就好像是一个通道，在通信过程中，发送方将数据放入通道，接收方从通道中取出数据。在数据交换结束后，必须释放连接。面向连接的服务特点是，接收方接收的数据和发送方发送的数据无论在内容上还是顺序上都是一致的。

TCP协议提供面向连接的服务。假设A为客户端，B为服务器端，建立连接的过程如下：一开始，B被打开并进入监听状态。A的TCP客户进程向B发出一个连接请求报文。B收到连接请求后，如果同意建立连接，就向A发送确认报文。A收到确认报文后，还要再向B发送确认报文，目的是通知B，A已知道B同意了建立连接。此时A，B都已进入连接建立状态，之后就可以利用此连接交换数据了。由以上描述可知，在连接建立的过程中，客户端和服务器一共交换了三个TCP报文，因此TCP连接建立也被称为三报文握手过程。

由于TCP提供全双工通信，所以在释放连接时，必须把两个方向上的连接全都释放掉。TCP协议释放连接的过程如下。当A数据传输结束后，A向B发送连接释放报文，停止发送任何数据，并主动关闭TCP连接。B收到连接释放报文后，向A发送确认报文，并进入关闭等待状态。此时，A到B的连接就释放了。当B没有数据要传输给A时，B向A发送连接释放报文，并进入最后确认状态，A接收到此报文后，向B发出确认报文，并进入关闭状态，B到A的连接就释放了。直到此时，AB之间的TCP连接才完全释放掉。由于在TCP连接释放过程中，AB之间一共交换了4个TCP报文，所以，TCP连接释放也被称为四报文挥手过程。

在连接存活期间，TCP采用以字节为单位的滑动窗口实现流量控制和拥塞控制，再辅以确认机制、超时重传等来保证数据传输的可靠性。

每一条TCP连接只能有两个端点，所以TCP协议只支持面向连接的、点对点的通信服务，不能支持多播和组播服务。

### 无连接服务

无连接的服务在通信双方之间并不建立连接，发送方简单地把接收到的数据依据IP地址和端口号发到网络上，至于接收方是否能够接收到数据，收到的数据是否与发送的数据一致，发送方均不关心，也不提供任何机制和技术来保证数据传输的可靠性。无连接服务的优点在于灵活方便、快速高效、开销低，因此特别适合于音频、视频等对延迟要求特别苛刻，对准确性要求相对宽松的场景的实时应用。

UDP协议提供面向报文的无连接服务，也就是说，发送方UDP对应用程序交下来的报文既不合并也不拆分，而是保留原来的边界，直接发送到网上，接收方UDP在收到UDP报文后，就直接交付给应用进程。其实，UDP这样简单的处理迫使应用进程必须注意调整报文的长度，因为过长的报文在网络层必须分片传输，加重了网络层的负担；过短的报文又会降低网络层的效率。由于在通信时，UDP没有建立连接，所以各种基于连接的保证可靠性的措施（滑动窗口等）很难在UDP协议中应用，因此UDP只提供尽最大努力的交付，并不保证数据的可靠传输（不出错、不丢失、不重复、不失序）。那么UDP传输时是如何提高可靠性的呢？UDP把此问题交给了应用进程来解决。应用进程可增加前向纠错和重传等措施提高数据传输的可靠性。

UDP不受连接的限制，可提供一对一、一对多、多对一和多对多的数据通信，特别适合实时数据通信、组播和多播服务。

## Rust网络编程特征

正如Rust的设计者Graydon Hoare所期待的那样，Rust拥有广泛的特性，能够让程序员写出易于维护、调试，且更安全更高效的代码。针对网络编程，Rust不仅提供了系统级的性能，还提供了许多高质量的库和开发工具。此外，语言本身的安全性、支持高并发和零成本抽象等特性也进一步为开发构建安全、可靠、快速的网络应用系统提供了强有力的保障。下面对Rust网络编程的主要特征加以介绍。

1. 友好的开发环境

Rust的std::net模块包含了一套完整的TCP/IP协议栈编程工具，可利用TCPListener创建TCP服务器、TCPStream创建TCP客户端、UDPSocket发送和接收数据。Rust还提供了一些非常出色的HTTP库用于HTTTP编程。例如reqwest是一个高层次的HTTP客户端库，而hyper是一个低层次的HTTP库，可提供HTTP服务器和客户端的API。Wasm为Rust提供了丰富的浏览器开发组件支持Web应用程序的开发。Rust支持最大规模的代码复用，可在资源管理器中加载任何已经实现的代码为我所用。目前，Rust中已经实现了许多出色的网络应用编程工具，在这里就不一一列举了。

1. 支持高并发

Rust 语言是首个支持异步编程的系统级语言，支持高并发零成本的异步编程。Mio是Rust的轻量级快速低级IO库，可为epoll，kqueue和IOCP支持的I/O事件通知队列提供跨平台、高效的接口。futures是Rust中的零成本异步编程库，可在没有标准库的情况下工作（例如在裸机环境中），提供了许多用于编写异步代码的核心抽象，如Future、 Streams、 Sinks、Executors等，使用他们可构建大型异步任务。Tokio整合了上述两种抽象，是一个高并发网络编程框架，集成了mio和futures库，以支持异步非阻塞事件驱动编程，为构建协议无关、没有争用的并发、高性能的异步网络应用程序提供了一种解决方案。

1. 零成本抽象

零成本抽象的基本原则是：你不使用的，你不负担成本；你使用的，你也没法更优化。零成本抽象是Rust的核心原则，不会增加运行时成本。Rust在网络编程中的零成本抽象表现在：所有权和借用机制取代垃圾收集器，保证内存和线程安全；迭代和闭包允许在切片上编写map、filter、迭代循环等，并可优化到与手写C相当的程度；traits一般在编译阶段就会为抽象的每种实例类型生成单独的副本，即使有些类型方法在运行时动态分配，也不会有更好的优化；Async/await and Futures实现异步/等待，跨等待引用等高层抽象，支持Rust实现高性能网络应用程序。

1. 高可靠性

Rust利用以下机制来保证高并发异步网络编程的可靠性。Tokio利用Rust的类型系统的限制来提供难以滥用的API；Backpressure工具简单易用、开销很小，可安全高效地解决连接请求过载的问题；Rust的所有权模型允许Tokio自动检测何时不再需要计算，并自动取消而无需用户调用cancel函数。当然Tokio并不能阻止所有错误的发生，但可最小化它们。

1. 轻量级

Rust在利用Tokio框架进行网络应用程序开发时，可以尽量降低系统开销，以最轻便的方式构建系统。Tokio可以很好地扩展，而不会增加应用程序的开销，使其能够在资源受限的环境中茁壮成长。首先，Tokio不包括垃圾收集器或其他语言运行时；其次，Tokio是一个小组件的集合，用户可以选择最适合手头应用的部件，而无需支付未使用功能的成本。

总之，在Rust中进行网络编程是一种愉悦的体验，因为Rust不仅提供了系统级的性能，还提供了许多高质量的库可以使用，而且它的安全性和并发性也让网络应用程序更加可靠。

## Rust集成开发环境

Rust 语言集成开发环境是可视化的，是Visual Studio 的扩展。在Rust语言集成开发环境中，程序员操作的平台就是一个可创建、编译和运行Rust项目的代码编辑器，它也集中体现了Rust 语言集成开发环境的各项功能。Rust工具链主要包含两个组件：编译器Rustc和软件包管理器Cargo。VScode、Rustc和Cargo共同构成了Rust语言集成开发环境最小闭包。

### 简介

#### VScode代码编辑器

目前还没有一款专门为Rust所用的语言编辑器，但是多种编辑器都有支持Rust的插件，其中使用最多的是VSCode。VSCode是Microsoft开发的简洁且高效的代码编辑器，它支持Windows，Linux和macOS等操作系统以及开源代码。它同时支持诸如调试，任务执行和版本管理之类的开发操作。它的目标是提供一种快速的编码编译调试工具。在VSCode中安装Rust analyzer插件，可以让VSCode支持Rust编译器，实现智能的自动补全、类型检验、自动格式化代码以及提供清晰的错误提示和修改建议；安装Rust Syntex插件，可支持语法高亮；安装TOML插件，可使用TOML文件管理配置。当然VSCode中还可以安装其他的有关Rust的插件，它们都分别提供了不同的功能支持，开发者可视各自的需要安装这些插件。另外，VSCode编辑器一般都会显示其最新支持的各种插件，并对其有详细的介绍，开发者也可以视情况安装。图1.12为安装了rust-analyzer插件，支持Rust编译器的VScode语言编辑器界面。



图1.12可用于Rust的VSCode界面

#### Rustc编译器

Rust语言集成了Java的内存安全和C的底层控制权，因此允许在安全地进行系统编程的同时，像C一样进行优化，以实现更高性能。Rust的编译器是 Rustc，是一种多道编译器。实质上，Rustc是 LLVM（Low Level Virtual Machine）的编译前端，它负责将文本代码一步步编译到LLVM中间码LLVM IR（LLVM Intermediate Representation），然后将LLVM IR交给LLVM来最终编译生成相应平台的机器码。Rust的具体编译过程如下：

首先Rustc将源码经过分词和解析，生成 AST（抽象语法树），然后把 AST 进一步优化为 HIR（High-level IR），目的是让编译器更方便地做类型检查，之后HIR会被进一步编译为 MIR（Middle IR），这是一种中间表示。引入MIR的目的在于：①缩短编译时间：MIR 可以帮助实现增量编译，当你修改完代码重新编译的时候，编译器只计算更改过后的部分，从而缩短了编译时间；②缩短执行时间：MIR可以在LLVM编译之前实现更细粒度的优化，因为单纯依赖LLVM的优化粒度太粗，而且Rust无法控制，引入MIR就可以增加更多的优化空间；③更精确的类型检查：MIR将有助于实现更灵活的借用检查，从而可以提升Rust的使用体验。最终，MIR会被翻译成LLVM IR，然后被LLVM编译为能在各个平台上运行的机器代码。

#### Cargo包管理器

Cargo是Rust的构建系统和包管理器，实质上就是一个功能强大的项目管理工具。程序员可以利用简单的Cargo命令来创建、编译、运行、发布项目，并能够以下载和编译依赖库的方式，借用已发布的rust程序包来方便、高效地构建代码。因此，cargo也是Rust语言拥有广泛拥趸的重要原因之一。在安装Rust的时候会一并安装Cargo，无需我们再专门安装。常用的cargo命令如表1-4所列。在下一节中，我们将展示如何利用这些命令创建并运行一个rust项目。

|  |  |
| --- | --- |
| 表1-5 常用的cargo命令 | |
| cargo --version | 查看cargo版本号 |
| cargo new | 创建项目 |
| cargo build | 编译构建项目、生成可执行文件 |
| cargo run | 同时编译并运行生成的可执行文件 |
| cargo check | 快速检查代码确保其可以编译，但并不产生可执行文件 |
| cargo build --release | 优化编译项目，并生成可执行文件 |

### 安装Rust集成开发环境

在Windows平台上，Rust编译器需要依赖MSVC提供的链接器，因此还需要下载安装Visual Studio（VS）予以支持。安装 Visual Studio 时，建议选择以下几个 Windows 开发程序： .NET 桌面开发、使用 C++ 的桌面开发和通用 Windows 平台开发。 你可能认为不需要全部三种，但很有可能会出现某依赖项需要全部三种，因此我们认为选择全部三种会更方便。新的 Rust 项目默认使用 Git， 因此还要将独立组件 Git for Windows 添加到组合中（使用搜索框按名称搜索它）。

Rust并没有专用的IDE，推荐使用Visual Studio Code（vscode），在vscode中安装Rust插件，即可优雅高效地编写Rust代码、调试执行Rust应用程序。

安装Rust工具链。Rust官网地址为：[https://www.rust-lang.org/，进入Rust官网，点击“Install”链接进入Rust安装界面，选择适配操作系统的安装选项，根据提示运行Rustup，安装Rustc和Cargo](https://www.rust-lang.org/，进入Rust官网，点击)。在此页面中还可查看Rust编译器的下载和安装方法的文档说明。运行rustup update命令可更新Rust工具链，运行rustc --version命令查看已安装的Rust版本号，卸载Rust用rustup self uninstall命令。

安装VS支持工具。Visual Studio官网地址为: https://visualstudio.microsoft.com/zh-hans/ ，进入Visual Studio官网，点击“下载”链接进入下载安装界面，选择合适的版本，根据提示下载安装Visual Studio和vscode。

在vscode中配置Rust开发环境。打开vscode，添加rust插件或者rust-analyzer插件，重新启动vscode后，即可在vscode中轻松编写、编译、调试和运行Rust程序了。

### 使用cargo命令创建第一个Rust项目

打开VScode，在命令行窗格的Terminal页面下进入存放代码的目标目录，输入命令cargo new hello\_world命令，回车后，Rust会自动创建一个hello\_world文件夹，进入这个文件夹后执行cargo run命令，可以看到输出结果为“Hello, world！”，如图1.13所示。

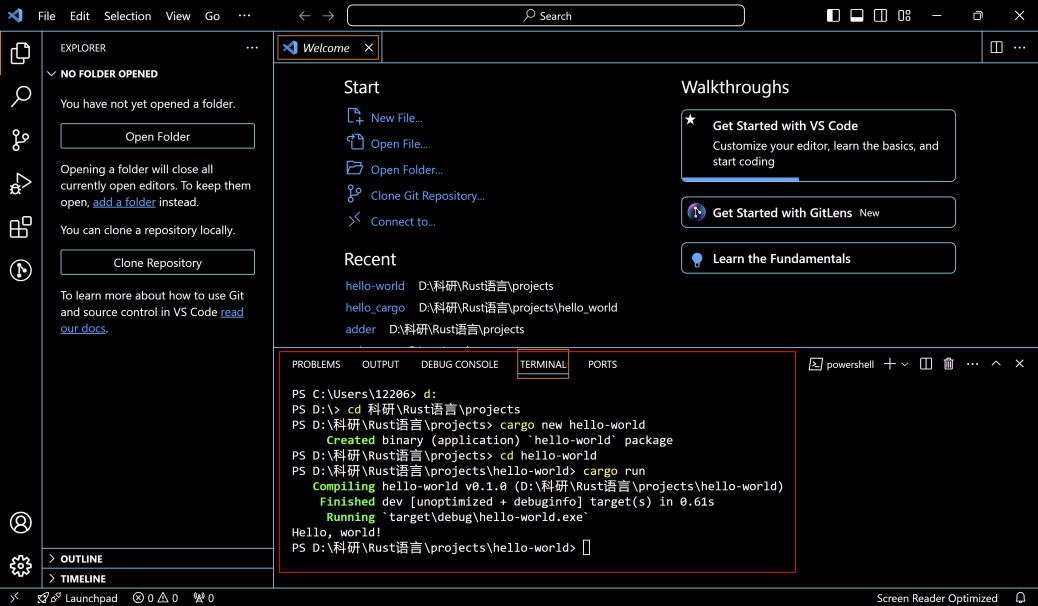


图1.13 创建并运行hello-world项目的命令

至此，我们的第一个程序已经运行完毕，可是我们还没有看见代码呢。执行如下命令序列“打开文件菜单—选择打开文件夹—选择hello-world文件夹—点击确定”，之后VScode窗口的左窗格就出现了hello-world项目的目录，由两个文件夹和三个文件组成，单击src文件夹下的main函数，右上窗格就显示出hello-world项目中的主函数的代码。如图1.14所示。src文件夹存放源代码，其中main函数是程序的入口函数。target文件夹存放debug/release时的文件，不需要手动编辑。Rust在创建项目的同时初始化了一个Git仓库，gitignore文件用来控制版本系统。

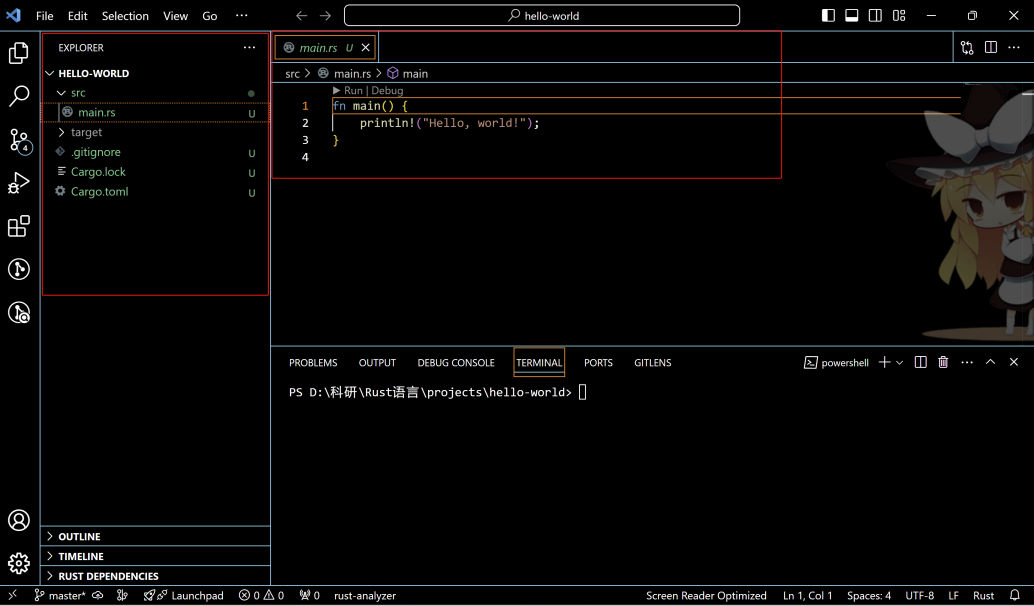


图1.14 hello-world文件夹及main函数

Cargo.toml 文件使用 TOML (Tom's Obvious, Minimal Language) 格式，显示了项目的配置信息。如图1.15所示，[package]小节中设置了Cargo编译程序所需的配置：项目名称、版本，以及使用的Rust大版本号（edition不同于version，是Rust 的核心版本，即 2015、2018、2021 版等）。[dependencies]小节列出项目依赖的所有包，在Rust中，代码包是crate。Cargo.lock文件由cargo维护的项目相关依赖的具体信息组成，不需要手动编辑。



图1.15 cargo.toml文件内容

下面我们在main函数中添加一行代码“println! ("Let's embark on network application programming!");”，保存后执行cargo build命令编译文件，执行cargo run命令运行项目，在右下窗格的terminal页面显示出了运行结果，在图1.16中用红色方框标出。

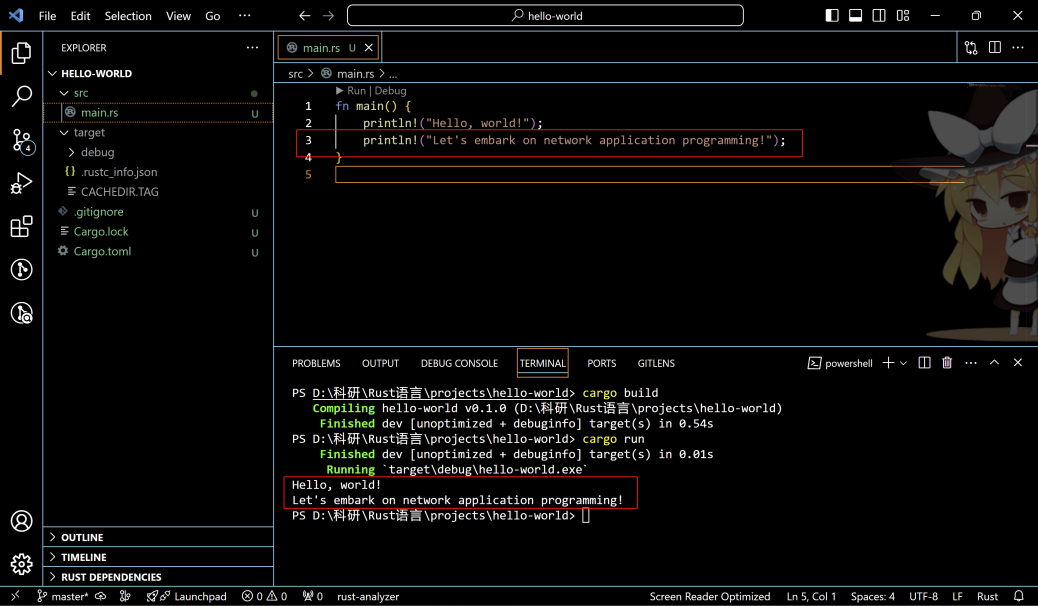


图1.16 项目运行结果展示

至此，我们用cargo命令创建并运行了第一个Rust项目“hello-world”，那么让我们现在就开始我们的Rust网络应用编程之旅吧！

## 小结

本章主要介绍了网络应用编程的基础知识。首先，网络应用编程架构方式的选择（B/S还是C/S）必须依据实际要开发的系统的特点和性质来确定，并且一般是在需求分析的最后阶段就要确定下来的；其次，TCP/IP协议是Internet的基本协议，它规范了网络上的所有通信设备，尤其是一个主机与另一个主机之间的数据往来格式以及传送方式，其中IP地址、端口号、MAC地址等概念皆是网络应用编程的理论基础，本节也是本章的重点；最后，Rust集成开发环境是本书构建网络应用系统所采用的平台。本章是Rust网络应用编程的开篇之章，本章的内容是后续章节知识学习的基础，也是攀爬利用Rust语言进行网络应用系统开发实践高地的理论基地。

# Rust初步

本章要点：认识Rust的基本语言成分；深刻理解Rust中所有权、借用和生存期的概念，并能够在编写代码时正确应用；熟练掌握函数和泛型的命名规则、应用方法和技巧；熟悉Rust的并行处理机制，掌握线程管理方法和消息传递机制；熟悉Rust的测试和调试技术，能够综合运用Rust的语言成分、编程方法和技术，开发具有并发和异步特征的小型应用系统。

## 引言

过去几十年中，C++语言一直在系统编程领域占主导地位。然而由于没有内生的安全措施，C++的安全性、可靠性和性能更依赖于代码编写者的经验和技能。其次，C++在内存管理方面存在的问题（空指针引用等）容易导致程序崩溃、数据损坏等严重后果。以上问题的纵横交错导致了利用C++语言开发的复杂软件系统的可靠性和稳定性不能得到很好的保证，在特定情况下甚至会崩溃。因此，业界急需一种更强大、更安全的系统级编程语言来满足不断增长的复杂系统开发需求，Rust便在这种背景下横空出世。

Rust是一种具有混合范式的、面向过程的编程语言。它借鉴了C和C++的底层控制能力、Haskell和ML的函数式编程特性，以及其他一些现代编程语言的创新思想。Rust的设计者们希望通过结合这些不同语言的特性，创造出一门既能够提供高性能和底层控制，又能够保证内存安全和并发的新型编程语言。Rust的零成本抽象原则提供了与C++相媲美的强大性能，灵活的消息传递机制和严格的编译属性提供了“无畏并发”的特点。现代的内存管理技术、独特的所有权管理机制、完善的借用规则、以及严格的生存期措施保证了内存和线程的强安全性，利用Cross工具实现跨端和跨平台编译，可构建超大规模的复杂系统，并加速系统开发的效率、以及保证系统的安全性和稳定性。Rust的高性能、高并发和强内存安全属性使其成为一门优秀的系统级编程语言。

此外，Rust拥有活跃的开源社区和丰富的生态系统。社区中有大量的贡献者和开发者分享经验和贡献开源项目、提供了众多的常用库和工具，使得开发者可以更快地构建应用程序，进一步丰富Rust生态。

## 概述

相比其它编程语言，Rust具有内生安全、零成本抽象、无畏并发等鲜明特色。为迎合和落实这些特色，Rust设计者们对语言成分进行了重新规划和开发，为进行安全可靠的系统级复杂应用系统开发提供了保障。

### 特色

Rust是一门通用的多范式语言，它借鉴了Haskell的类型系统、C++的RAII原则（Resource Acquisition Is Initialization）、Cylone的基于区域的内存管理技术和C的细粒度语言控制资源管理思想等形成了自己鲜明的特色：在安全高效系统编程的同时又对底层具有控制权。接下来，我们将对Rust语言的主要特色进行更为详细的阐述。

1. **内存安全保证**

Rust是静态强类型语言，这就意味着Rust可在编译时跟踪变量所分配资源的变化情况，可及时报错并给出原因和修改意见。Rust采用虚拟内存空间在栈和堆上分配内存，为防止认为错误，它并不需要开发者显式地通过malloc/new或free/delete之类的函数去分配和回收内存，而是通过使用所有权的概念自动分配和回收内存，以保证内存安全。具体来说就是，Rust引入智能指针来管理内存，智能指针存储在栈中，且在堆上开辟内存空间，并拥有其所有权。智能指针与生存期标记隐式关联。在生存期中，Rust通过智能指针的生存期标签来运行借用/引用检查算法。当智能指针在栈元素被自动清空时，自动调用析构函数，来释放其管理的堆内存空间。这样可以保证永远不能访问已释放的内存空间，也可以防止释放已被其他变量调用的指针，从而很好地保证了内存安全。

1. **零成本抽象**

Rust中的零成本抽象是指使用trait和泛型等实现的抽象机制，该机制在编译时会进行类型擦除，使得在运行时没有额外的开销，因此代码在性能方面更加高效。由于Rust的静态类型系统和编译器能够在编译时进行类型检查和优化，因此可以实现零成本抽象。零成本抽象是Rust的核心原则，在提高代码质量和代码可读性的同时，不会增加任何运行时成本。

例如，traits一般在编译阶段就会为抽象的每种实例类型生成单独的副本，即使有些类型方法在运行时动态分配，也不会有更好的优化。

1. **无数据争用的线程安全保证**

Rust提供了基于所有权系统、消除了数据争用的线程安全保证。所有权系统使得两个线程永远不能拥有具有写访问权限的同一变量。Rust通过一整套设计精良的基础设施来保证这一点，例如：进行语义限制的Send / Sync Trait、共享不可变所有权的Arc 、提供内部可变性的Mutex / Rwlock、管道模型channel等，这些基础设施配合类型检查和借用模型能够把崩溃或泄漏暴露在编译期，从而强迫程序员写出正确的代码。线程安全也是Rust明显优于其他语言的重要特色之一。

1. **高效的C绑定**

Rust可以在没有开销的情况下调用C函数。Rust提供了一个“外部功能接口”来与C语言的API进行通信，与C语言进行互操作。并且Rust还可以将其所有权系统泛化到C函数，同时保证其内存安全。

1. **灵活的模式匹配**

Rust的模式匹配允许根据数据的形状和内容进行条件分支，类似于其他语言中的switch语句，但更加灵活和强大。与其他语言的switch语句相比，Rust可以匹配多种不同类型的数据，而不仅仅是整数，比如Rust模式匹配可以用来解构元组和结构体、匹配枚举类型、匹配引用和指针、匹配范围等等。Rust中还可以用条件进行复杂匹配，这是其他语言所没有的。

1. **友好的错误处理**

Rust将错误分为可恢复错误和不可恢复错误（panic）。使用Option和Result两种方式处理可恢复错误。Result类型声明时有个 #[must\_use] 标注，如果该类型返回的值没有被显式使用，编译时会报警，确保错误能被妥善处理。Option和Result类型都有unwrap和expect方法，expect可以自定义错误信息，能更友好地给开发者提示。Rust中提供了特殊的异常处理能力panic!，panic发生后程序会立刻停止运行。Rust还可以自定义错误类型，并可以利用“?”操作符传播错误。

1. **强大生产力**

Rust提供了丰富的工具和库，这是因为Rust具有友好的语法和内置的文档工具。Cargo包管理器可以轻松地构建、测试和发布Rust代码。此外，Rust的标准库中提供了许多常用的数据结构和算法，使得开发人员可以更快地编写高效的代码。

### 基本成分

基本成分是Rust编程的基石，它们构筑起了我们编写代码的基本框架。引用和借用、字符串、表达式、生存期、顺序结构、分支结构、循环结构、函数结构以及常量变量等构成了Rust的核心语言成分。基本成分的灵活运用，可以使我们编写的代码更加健壮、高效和可维护。Rust语言的基本成分与其它编程语言类似，却又别具特色。在本节中，我们将主要介绍Rust语言成分中不同于其他语言的特色部分。

1. **引用**

在Rust中，引用是一种轻量级的指向数据的方式，它允许我们在不获取所有权的情况下访问和操作数据。Rust借用检查器可以在编译时发现许多常见的错误，如空指针和非法内存访问等，这在极大程度上保障了内存的安全性。

Rust中的引用分为两种类型：不可变引用和可变引用。不可变引用使用&符号来声明，允许以只读方式访问数据；可变引用则使用&mut来声明，允许我们以读写方式访问和修改数据。如下例所示：

let mut x = 5; //x为可变变量

let y1 = &x; //y1为不可变引用，不能通过y来修改x的值

let y2 = &x; //多个不可变引用指向同一变量

let mut x = 5; //x为可变变量

let y = &mut x; //y为可变引用，只能有一个可变引用指向一个变量

\*y += 1; // 通过y使x的值加1

在使用引用时，需要遵守以下两条规则以避免数据竞争和内存访问错误：

①在某一时刻要么有且只有一个可变引用、要么只能有多个不可变引用、不能既有可变引用又有不可变引用指向某一变量。

②引用必须始终有效，被引用的数据不能在引用的生存期内被销毁。

1. **字符串**

通常情况下，Rust中的字符串类型指字符串切片&str和字符串String。字符串切片&str是对UTF-8编码的字符串的引用（比如字符串字面值就是一种字符串切片），没有所有权，只能查看、不能改变字符串中的数据。字符串String来自标准库，同样采用UTF-8编码，拥有数据所有权，是可增长、内容可修改的字符串类型。例如：

Let s="hello"; //以字符串字面值“hello”创建字符串切片s

s.to\_string( ); //将字符串切片s转换成string类型

let mut s = String::from("hello"); //创建可变字符串类型变量s

s.push\_str(" world!"); //push\_str函数将" world!"追加到s尾部

通过使用字符串切片&str，Rust还支持对不可变字符串的引用操作，例如：

let s = "Hello World"; //创建变量s

let sl = &s[6..11]; //提取字符串s

println!("{}", sl); //输出 "World"

总之，通过String类型和字符串切片，Rust提供了丰富的功能来处理和操作文本数据，保证了字符串处理的高效性和安全性。

1. **表达式**

Rust中的表达式与其他语言类似，都用于计算和产生值。Rust支持多种类型的表达式，包括算术表达式、逻辑表达式和比较表达式等。算术表达式主要用于执行数值运算，算数运算符：加（+）、减（-）、乘（\*）、除（/）、求余(%)，位运算符：按位取反（!）、按位与（&）、按位或（|）、按位异或（^)、左移（<<）、右移（>>）。逻辑表达式用于处理布尔值和逻辑运算，常见的逻辑运算符包括与（&&）、或（||）、非（!）。比较表达式对各种量进行比较，比较运算符有大于（>）、大于等于（>=）、小于（<）、小于等于（<=）、等于（==）、不等于（!=）。

语句块表达式：在Rust中，语句块也可以是表达式的一部分。语句和表达式的区分方式是后面带不带分号（;）。如果带了分号，意味着这是一条语句，它的类型是（）；如果不带分号，它的类型就是表达式的类型。Rust将按顺序执行语句块内的语句，并返回最后一个表达式的值，可以利用这一特点来写返回值。如下所示：

let x : () = { println!("Hello."); }; //一条语句作为表达式，输出“Hello”

let y : i32 = { println!("Hello."); 5 }; //把语句块中的最后一个表达式的值赋给变量y，

//所以y的值为5，输出“Hello”

1. **赋值语句**

通过赋值语句，可以将表达式的运算结果（也可以是函数调用的运行结果）分配给变量，并在程序中进行数据操作。Rust的赋值语句采用变量绑定的方式，严格遵循所有权和借用规则，确保代码的安全性和可靠性。Rust默认变量值是不能随意更改的，如果要更改，需要加上mut关键字，使变量成为可变变量。Rust还可以通过结构赋值、利用一条语句就可以将多个值赋给多个变量。另外，在Rust中如果变量声明后没有赋值就不能使用；变量声明后却不使用，编译时将会报警。举例如下：

let y = 10; //创建一个不可变变量y，并赋值为10

let mut x = 5; //创建一个可变变量x，并赋值为5

let (a, b) = (3, 4); //将一个整数元组(3, 4)解构为两个值，并分别赋给a和b两个变量

fn main() {

let x: i32;

println!("Hello world!");} //编译时会出现一条警告信息，但还能正常运行

fn main( ) {

let x: i32;

println!("The value of x is: {}", x);} //编译出错，x变量没有赋值不能使用

1. **流程控制**

Rust和其他语言一样，基本控制结构包含顺序、分支和循环。顺序和循环与其他语言大体一样，不再赘述，下面主要来叙述一下条件分支中与其他语言不同的地方。Rust的条件分支的判断条件必须是布尔类型，这一点与 C、Ruby 以及 JavaScript 等语言不同，需要特别注意。另外一个与众不同的地方是，Rust增加了let...if...语句，如下所示：

let number = if condition {1 } else { 0}; //如果条件为真，number赋值1，否则赋值0

当然if条件后面和else后面都可以是语句块，这时要根据if条件是否为真，执行其中一个语句块，并将这一语句块的最后一个表达式的值赋给let后的变量，需要注意的是，if条件之后的两个分支中表达式的类型必须一致。例如：

fn main() {

    let x=5;  let y=6;

    let s= if x>y {println!("x>y"); 1} else {println!("x<y"); 2};

//x>y条件为假，执行else后面的语句块，输出“x<y”，并将2赋给s

    println!{"{}", s}; //输出2

}

1. **模式匹配**

Match语句是进行多模式匹配的语句，完成了多条件分支的功能，类似于C语言中的Switch语句，但更加灵活，如下所示。

fn main() {

    let s=98;

    match s {

        100 => println!("excellent"),

        \_other => println!("keep on doing"); // s可能取到的所有值都要有相应的分支臂

    }

   }

在这里要强调的是模式变量的所有取值（模式）都要有相关联的分支代码（表达式），即匹配必须是详尽的，匹配分支（匹配臂）中的最后一个表达式的值是整个match表达式返回的值，所有匹配臂的返回值的类型必须相同。如果模式过多，可以用\_other代表其他所有未列出的模式。如果分支代码只有一条语句或者一个表达式，通常不使用花括号，如果一个分支中有多行代码，则必须使用花括号。

在许多情况下，将match和枚举变量结合起来是很有用的。在Rust代码中会经常看到这种模式：match一个枚举，将一个变量绑定到里面的数据，然后基于它执行分支代码，例如：

enum UsState { Alabama, Alaska,} // 枚举变量

enum Coin { Penny, Nickel, Dime, Quarter(UsState),}     // Quarter中嵌套了一个枚举

fn value\_in\_cents(coin: Coin) -> u8 {

    match coin {

        Coin::Penny => 1,

        Coin::Nickel => 5,

        Coin::Dime => 10,

        Coin::Quarter(state) => { println!("State quarter from {:?}!", state); 25 }

    }

}

fn main() {

    value\_in\_cents(Coin::Quarter(UsState::Alaska));

} //屏幕输出为：State quarter from Alaska

1. **常量和变量**

相比于C++语言，Rust中的常量和变量增加了遮蔽（shadowing）的概念，即在同一作用域内使用同一名称声明一个新变量，这个新变量会“遮蔽”之前同名的变量。这种机制提供了一种灵活的方式来管理和修改作用域内的标识符，同时保持代码的清晰、安全、可读性和可维护性。可能有人会问：在Rust中为什么不用一个不加mut声明的变量来表示常量呢？甚至这样做还可以减少语言的概念数量。与常量的值永恒不变所不同的是，我们可以用let再次声明一个不可变变量，重新赋值，旧的值就会被遮蔽，显露出来的就是一个新的值。例如：

let a=100; a=a+10; //出错，a是不可变变量

Let mut a=100; a=a+10; //正确，a为可变变量

let a=100; let a=30;//正确，变量遮蔽，a的值为最后一次let赋值

1. **函数**

Rust中的函数有着独特的性质。首先，函数在Rust中是显式声明的，必须以`fn`关键字开始定义函数。这种明确的声明方式让代码更加清晰易读。其次，Rust中的函数是类型安全的。这意味着函数的参数和返回值都必须指定类型，并且在编译时进行类型检查，确保类型的一致性。这种类型安全性有助于在编写代码时避免一些常见的错误。然后，Rust中的函数还支持模式匹配和模式重载，程序员可以根据不同的输入情况编写不同的函数实现，以适应不同的需求。这种灵活性使得函数可以根据具体的情况执行不同的操作，提供更加定制化的功能。最后，Rust中的函数支持闭包（一种特殊的函数类型，可以捕获周围环境的变量，并在需要的时候执行）。闭包使得函数可以具有更高的灵活性和重用性，可以作为参数传递给其他函数，或者存储在数据结构中。函数的概念在Rust中得到了精心的设计和实现，使得我们能够以一种安全、高效和可靠的方式编写代码。例如：

fn math(x:u32){ println!("x is {}",x); } //定义函数math

fn main(){let x=1; math(x);} //Rust特性使得函数的定义没有位置限制

## 所有权、借用与生存期

Rust中变量对内存空间的所有权规则确保了运行时内存的安全高效利用，借用则允许在不获取内存空间所有权的情况下访问内存资源，生存期主要是为了避免悬垂引用。三个概念紧密联系为Rust内存安全提供保障。

### 所有权

所有权（ownership)是Rust最核心的概念之一，对保证内存安全起着巨大作用。任何语言都必须具有管理内存的机制，Rust既没有采取Java语言的垃圾回收机制，也没有采取C语言的程序员人工分配和释放内存，而是选择通过所有权系统来管理内存。所有权系统包含了一套编译时检查规则，编译器在编译时会根据这些规则检查代码，如发现任何违反所有权规则的代码，程序都不能编译通过。由于所有权系统是在编译而非运行时实施检查，所以它并不会影响程序运行的速度。

与所有权密切相关的两个概念是栈（Stack) 和 堆（Heap）。堆和栈都是程序运行时在内存中开辟的存储数据的空间，但两者的结构不同。栈是一种后进先出的数据结构，是一种特殊的线性表，占据连续的内存空间。栈只能在栈的一端进行出入栈操作，并且栈中的所有数据都占用已知且固定大小的空间。然而，在编译时我们往往会遇到大小，长度可变的数据。例如在控制台输入一个字符串，这个字符串的大小就由用户决定，像这样的数据就不能放在栈上，而是要存放在堆上。当我们向堆中存放数据时，操作系统内存分配器会在堆中找出一块足够大的内存空间并将它标记为已用，返回它的地址指针，称为“在堆上分配内存”或叫做“分配”，可见堆中数据是离散村发的。同时由于返回的指针的大小是固定的，所以将指针存放在栈中。在存储数据时由于入栈操作不需要操作系统寻找新的内存空间，只需要在栈顶进行操作即可，而在堆中存放数据需要寻找可用的内存空间，还要进行登记，因此在栈上存储数据更快。

所有权系统可以很好地管理堆数据，它可以跟踪代码并发现代码哪一部分在使用堆数据，还可以最小化堆中重复数据，及时清理堆中不再使用的数据。所有权规则如下：

1.Rust中的每一个值都有一个所有者（owner）。

2.值在任一时刻有且只有一个所有者。

3.当所有者（变量）离开作用域，这个值将被丢弃。

在Rust中，每个值都有且仅有一个所有者，并且所有者负责在其作用域结束时释放该值的内存。当值被绑定到一个变量上时，值的所有权就被转移到该变量上，当该变量超出其作用域时，Drop函数会被自动调用，以释放相关内存。下面是一个简单的例子：

1. fn main() {

2. let a = String::from("String"); // 创建一个String值并将其所有权转移到变量a上

3. println!("{}", a); // 打印"String"

4. } // 变量a超出作用域，a被释放，与其相关的内存被回收

String类型变量的内容是可变的，其所占的内存空间的大小也不固定，因此其值放在堆上。另外，为调用这个值，还必须设置一个指针指向这个值，这个指针其实就是string类型的变量，他所占的内存空间固定，所以string类型的变量放在栈中。如图1所示：左侧是存放在栈上的数据包括指针、长度和容量，右侧是存放在堆上的内存数据。

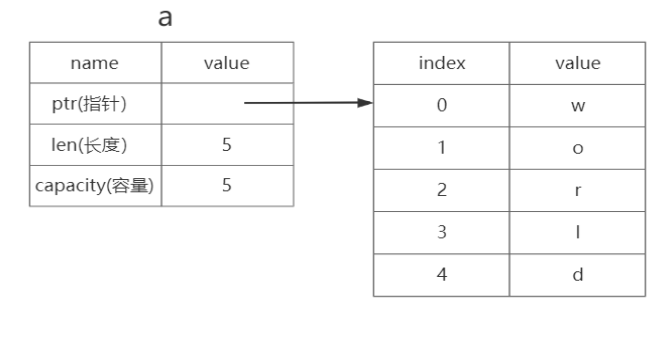


图2.1 string类型内存空间分配情况

我们重点来看下面的例子

1. fn main() {

2. let a = String::from("Hello world");

3. let b = a; // 所有权发生转移，a失去"Hello world"的所有权，b接管了所有权

4. // println!("a: {}", a);这行代码会引发编译错误，因为a失去了对值的所有权。

5. println!("b的值是: {}", b); // 输出: Hello world

6. }

按照正常的思维逻辑，我们会自然的想到a的值只是简单的复制了一份给b，但是在Rust中并非如此，假使a占用的内存特别大，那么b在复制的时候就会非常慢，这就会导致性能下降，在Rust中是不允许发生这种情况的。这意味着在Rust中不能直接将一个值赋给多个变量。事实上，在计算机底层，Rust中内存分配如图2所示。在a赋值给b的过程中，在栈上a的数据被复制了一份，包括指针、长度、容量，但是并不复制指针指向的堆上的值，这叫“浅拷贝”(shallow copy)，复制之后，a失效，所以这样的操作叫作“移动”。这样做的好处就是可以防止a和b都离开作用域的时候对内存二次释放(double free)带来的不安全性。

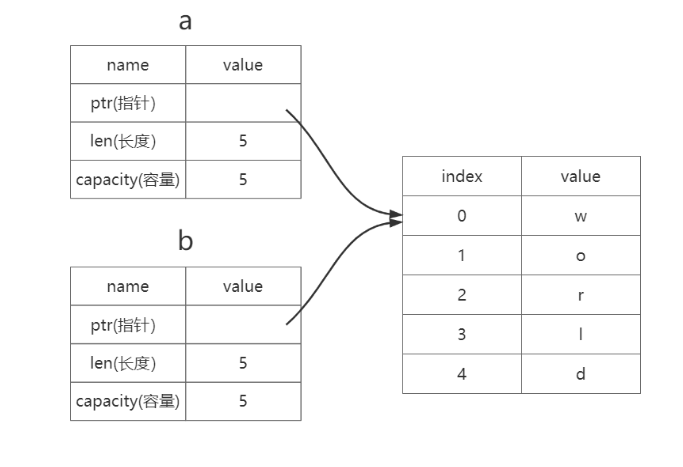


图2.2 浅拷贝

如果我们希望仍然能够使用原变量并可保留新变量的所有权，这时候我们可以使用另一种复制方式：克隆(clone)，来创建原变量的副本，例如：

1. fn main() {

2. let a = String::from("a");

3. let b = a.clone(); // 用clone来创建a的副本（包括栈和堆上），并将副本的所有权给b

4. println!("a ： {}, b ： {}", a, b); // 打印"a : a, b : a"

5. }

clone操作是显式的“深拷贝”（deep copy），可能会导致性能损耗，所以我们应当谨慎使用，其在内存中分配如下图所示。

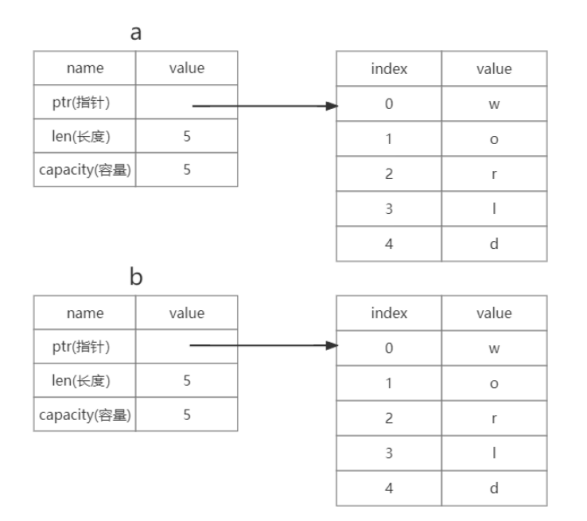


图2.3 深拷贝

值得注意的是，u32、u64、f64、char、bool等数据类型，它们所占空间固定且都存放在栈当中，当离开作用域时便出栈。因为复制它们的速度很快，没有必要令原变量失效，所以这些类型的变量在被复制给另一个变量后，所有权并未改变，相当于String类型的深拷贝。例如：

1. fn main() {

2. let a = 100;

3. let b = a; // a的值被复制到了b

4. println!("{},{}", a, b); // 打印可知"a = 100, b = 100"

5. }

在Rust中，函数调用也涉及所有权的转移。当将值传递给函数时，可以选择将所有权转移给函数或者通过借用的方式让函数暂时使用该值的引用。默认情况下，值的所有权会被传递给函数，这意味着函数成为该值的所有者，并在函数结束时即作用域结束时负责释放该值的内存。

1. fn main() {

2. let s = String::from("我的值是a");

3. a\_given(s); // 将所有权转移到函数中，s失效

4. // println!("{}", s); // 这行代码会产生编译错误，因为所有权已经转移

5. }

6. fn a\_given(mut a: String) {

7. a.push\_str(“ I love you! “);

8. println!("{}", a);

} // 函数结束，参数a的所有权被释放，关联的内存被回收

当然，我们可以通过设计函数的返回值来实现函数内外所有权的传递。如下所示：

1. fn a\_given\_back(a: String) -> String {

2. println!("Given: {}", a);

3. a // 函数返回的是最后一个表达式的值，所以a的所有权会传递回调用函数

4. }

5. fn main() {

6. let a = String::from("我的值是a");

7. let b = a\_given\_back(a); // 重新获取返回值的所有权

8. println!("Now: {}", b);

9. }

### 借用

在Rust中，创建一个引用(references)的行为称为借用（borrowing）。具体来说，引用允许使用变量的值但并不获取其所有权，这种行为即被称为变量所有权的借用。通过借用，我们可以在不转移所有权的情况下对值进行读取、修改或传递。Rust编译器依据借用规则（borrowing rules），通过借用检查器（borrow checker）静态分析代码来确定引用的有效范围，保证程序的安全性。Rust通过＆T和＆mut T定义将引用分为两种：不可变引用和可变引用。

1. **不可变引用**

不可变引用（immutable reference）又称为共享引用，是一种不允许修改值的借用方式，使用“&类型”来定义不可变引用，如下例中的第一行代码。通过不可变引用，我们可以访问一个值，但不能修改它。不可变引用遵循的规则是：某一个值可以有多个不可变引用，也称为别名。

1. fn borrow\_immutable(a: &String) { //用&String定义不可变引用a

2. println!("Immutable Borrow: {}", a);

3. }

4. fn main() {

5. let a = String::from("我是a");

6. borrow\_immutable(&a); // 借用a的不可变引用，函数内部只能读取a的值而没有其所有权

7. println!("Still owns: {}", a); // 现在依然拥有a的所有权

8. }

在borrow\_immutable函数定义中，使用&符号来接受字符串的不可变引用。在主函数中，将a的不可变引用传递给borrow\_immutable函数。在函数内部，通过不可变引用来读取a的值，但是不能修改它。类似这样的借用操作不会改变a对值“我是a”的所有权，因此在函数结束后，a仍然拥有对值“我是a”的所有权。

1. **可变引用**

可变引用（mutable reference）是一种允许修改所引用值的借用方式，使用“&mut 类型”来定义可变引用，如下例中的第一行代码。通过可变引用，我们可以在不获取值所有权的情况下修改值。可变引用有一个重要的限制：在特定作用域内，某一个值只能有一个可变引用。这样可以有效防止数据竞争(data race)

1. fn borrow\_mutable(a: &mut String) { //用&mut String定义可变引用a

2. a.push\_str(" World!");

3. }

4. fn main() {

5. let mut a = String::from("Hello");

6. borrow\_mutable(&mut a); // 借用a的可变引用，那么在函数内部就可以修改a的值

7. println!("Modified: {}", a); // 输出 "Hello World!"

8. }

在borrow\_mutable函数定义中，使用&mut来接受字符串的可变引用。在主函数中，将a的可变引用传递给borrow\_mutable函数。在函数内部，我们可以通过可变引用来修改a的值。这个例子中，我们使用push\_str方法向字符串添加了新的内容。类似这样的借用操作不会改变a对“Hello”值的所有权，因此在函数结束后，仍然可以打印a修改后的值。但是可变引用有一个很大的限制：如果对一个值创建了一个可变引用，就不能再对它创建其他任何的引用，包括可变和不可变引用，否则编译器就会报错说这段代码是无效的。例如：

1. fn main()

2. {

3. let mut a = String::from("Hello");

4. let a1 = &mut a;

5. let a2 = &mut a;

6. println!("{},{}",a1,a2);

7. } //报错：不能将a作为可变借用超过一次

值得注意的是，一个引用的作用域是从声明的地方开始一直持续到最后一次使用为止。不能在同一时间内多次将 a 作为可变引用，变量a的第一个可变引用在 a1 ，其生存期持续到在 println！中使用它。但是如果在可变引用的创建和它的使用之后，即其生存期完结后，再尝试创建另一个可变引用，就不会发生错误。如下例所示：

1. fn main()

2. {

3. let mut a = String::from("I'm a");

4. let a1 = &mut a;

5. println!("{}",a1);

6. let a2 = &mut a;

7. println!("{}",a2);

8. }

这种限制正体现了Rust的安全性所在，它防止同一时间对同一数据存在多个可变引用，这样做的好处是 Rust 可以在编译时就避免数据竞争。数据竞争会导致一系列错误，并且难以在运行时发现，难以诊断和修复。Rust 为了避免了这种情况的发生，直接在编译时就不允许存在有数据竞争的代码。类似的，Rust 在同时使用可变与不可变引用时也有类似的限制，例如：

1. fn main() {

2. let mut a = String::from("Hello");

3. let a1 = &a; // 不可变引用

4. let a2 = &mut a; //可变引用

5. println!("{},{}",a1,a2);}// 编译错误：不可变引用和可变引用同时存在

一个变量不能同时拥有可变和不可变引用，因为有了可变引用，值改变后不可变引用就会失效，所以两者不能同时存在。但是可以拥有多个不可变引用，因为不可变引用借用的值不会被修改，因此允许多个不可变引用同时存在。例如：

1. fn main()

2. {

3. let a = String::from("Hello");

4. let a1 = &a;//不可变引用

5. let a2 = &a;//不可变引用

6. println!("{},{}",a1,a2);

7. }

综上，某一个变量同一时刻只能有一个可变引用或者多个不可变引用。

另外我们再来看一个常见的内存错误：悬垂引用(dangling reference)，当一个指针指向了内存中的某个地址，而这块地址却被释放或者分配给了其他人，这个指针就变成了悬垂指针(dangling pointer)。在Rust中，编译器保证引用不会变成悬垂指针引用。如下例所示：

1. fn dangling\_reference() -> &String {

2. let b=String::from("hello world");//创建变量b

3. &b;//将b的引用返回

4. }//函数结束，b的作用域完结，b的值被释放，而b的引用被返回

5. fn main() { let a=dangling\_reference(); }//a成为悬垂指针，指向的内存中没有值

6. // error[E0106]: missing lifetime specifier

以上代码中，我们在dangling\_reference中创建变量b，然后返回b的引用，而变量b在出作用域后就会被释放，返回的引用就指向了一个被释放的内存地址，就会出错。那么如何解决这一问题呢？生存期的使用是解决这一问题的关键。

### 生存期

生存期（lifetimes）是Rust语言中用于描述变量在内存中存在的时长及其有效的作用范围，即描述了变量从创建到销毁的时间段，以及在此期间该变量可以被访问和修改的规则。变量的生存期是可以推断的，但在复杂的情况下需要显式声明。Rust编译器通过借用检查器（borrow checker）能够对变量的生存期进行静态分析和检查，保证了在编译时期就可以发现潜在的内存安全问题，从而避免了悬垂引用（dangling references）或引用非法内存等严重问题的产生。

在Rust中，生存期注解常用“单引号+小写字母”的形式表示，例如：'a表示，用于将引用的生存期与引用的使用范围关联起来。生存期注解的主要作用是告诉编译器引用的有效范围，使编译器能够在编译时进行静态检查，以确保生成的代码能够正确处理引用的生存期。为了保证引用在特定生存期内有效，可以使用生存期边界。生存期边界通过以下语法指定：'a: 'b，其中'a和'b是不同的生存期标识符，表示'a的生存期要长于或等于'b的生存期。此外，Rust还有一个特殊的生存期，称为静态生存期，代表整个程序的持续时间。静态生存期通常用于静态变量或全局变量等长期存在的数据。下面的代码展示了如何在Rust中使用生存期注解。

1. // 定义一个函数，接受一个字符串切片并返回同样的字符串切片

2. fn get\_slice<'a>(s: &'a str) -> &'a str {s}

3./\*通过生存期注解 'a告诉编译器，返回的引用的生存期与s 的生存期相同，即它们在同一作用

4. 域内有效\*/

5.fn main() {

6. let s = String::from("Hello, Rust!"); // 创建一个新的字符串

7. let result; // 声明一个变量来存储结果

8. {

9. result = get\_slice(&s); // 调用函数获取字符串切片的引用

10. } // 限定作用域结束，result 变量仍然有效，因为result与s具有相同的生存期

11. println!("Result: {}", result); // 输出结果

12. }

在这个简短的例子中，首先定义了一个函数 get\_slice，它接受一个字符串切片的引用并返回同样的字符串切片的引用。生存期注解 'a 用于确保输入和输出的引用具有相同的生存期，因此，尽管result的内部作用域结束，result 变量仍然有效。

下面的例子展示如何通过结构体表示员工信息，并使用生存期参数处理这些信息中的字符串引用，确保引用在结构体实例的生存期内有效。

1. struct Employee<'a> {

2. name: &'a str, //name与Employee的生存期注解一致，因此两者有效范围一致

3. department: &'a str, // 同上

4. age: u32, // 员工年龄是一个32位无符号整数

5. }

6. fn main() {

7. let name = "ZhangSan";

8. let department = "Engineering";

9. let age = 25;

10. let employee = create\_employee(name, department, age); //创建一个新的员工实例

11.   fn create\_employee<'a>(name: &'a str, department: &'a str, age: u32) -> Employee<'a> {

12. Employee { // 构造一个新的 'Employee' 实例

13. name,

14. department,

15. age

16. }

17. }

18. println!("Employee Name: {}", employee.name);

19. println!("Department: {}", employee.department);

20. println!("Age: {}", employee.age);

24. }//因为employee.name、employee.department与employee实例生存期一致，所以至此仍然有效

在本节中，我们对生存期进行了探讨。接下来，我们将继续探索更多关于Rust内存安全的一些知识和实例。

### 内存不安全示例

所有权、借用和生存期是Rust为了提供内存安全和并发安全而设计的概念。涉及所有权的内存不安全问题有：悬垂指针、重复释放、空指针引用、多线程竞争、引用无效内存或者未初始化的内存等，处理这些问题的关键在于正确地管理所有权。通过遵循Rust的所有权规则和约定，如正确使用 `Box`、分析代码以检查所有权转移和合理使用引用等来避免内存不安全问题。涉及借用的内存不安全问题有：可变借用冲突、借用无效数据、非法借用、可变和不可变借用冲突等，借用检查器可以确保在编译时捕获这些错误。涉及生存期的内存不安全问题有：引用超出有效生存期、生存期不合理的交叉、空指针引用、循环引用等，正确管理和注释引用的生存期是解决这类问题的关键。下面将利用几个实例来加强对内存不安全的诠释。

例1.在Rust中，不可变引用保证了数据不会在引用的同时被修改。然而在本例违反了这个规则，尝试通过不可变引用修改已被借用的向量。

//注意！！！这段代码是为了演示内存不安全的情况，不能成功的编译运行

1. fn main() {

2. let mut x = vec![1, 2, 3, 4, 5];

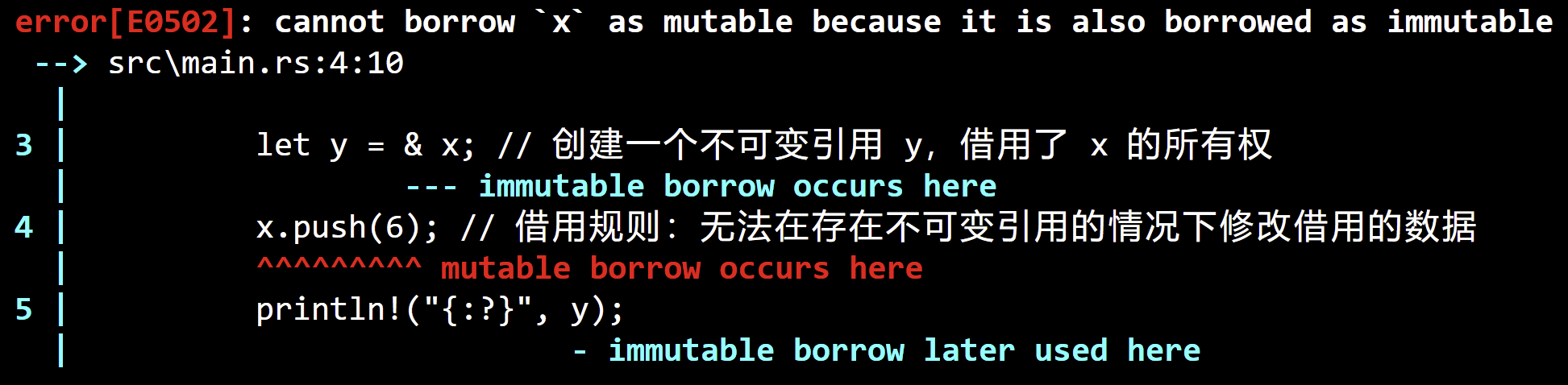
3. let y = &x; // 创建一个不可变引用 y，借用了 x 的所有权

4.   x.push(6); // 借用规则：无法在存在引用的情况下修改借用的数据

5.   println!("{:?}", y);

6. }

程序运行后，编译报错如图2.4所示。

2.4 不可变引用在改变变量值之后使用的错误

此例的错误在于：尝试在存在不可变引用的情况下修改借用的数据。具体来说，错误发生在 x.push(6); 这一行代码。如果想要通过编译并正常运行，可将这行代码移到“println!("{:?}", y);”之后。改动如下。

1. fn main() {

2. let mut x = vec![1, 2, 3, 4, 5];

3. let y = &x; // 创建一个不可变引用 y，借用了 x 的所有权

4.   println!("{:?}", y);//依次输出x的5个元素

5. x.push(6); // 借用规则：无法在存在引用的情况下修改借用的数据

6.   println!("{:?}", x);//依次输出x的6个元素

7. }//此段代码可以通过编译并正常运行

通过这个例子，我们可以看到在Rust中正确处理所有权是非常重要的。不正确的所有权处理可能导致悬垂引用、数据竞争和未定义行为等内存不安全问题。Rust的静态检查机制有助于在编译时捕获这些错误，并给出错误和修改提示，这有助于及时修正代码。在编写程序时，我们应该遵循Rust的所有权原则，确保正确使用和传递所有权，以避免潜在的内存不安全问题。

例2.悬垂引用示例1。

//注意！！！这段代码是为了演示内存不安全的情况，不能成功的编译运行

1. fn main() {

2. let dangling\_ref: &String; // 创建一个未初始化的引用

3.   {

5. let value = String::from("Hello World");

6. dangling\_ref = &value; // dangling\_ref指向了局部变量 value 的内存

7. }//value生存期结束，其值被释放，dangling\_ref变为悬垂指针

8.   println!("Dangling Reference: {}", dangling\_ref);// 尝试使用悬垂引用

11. }

程序运行后，编译报错如图2.5所示。

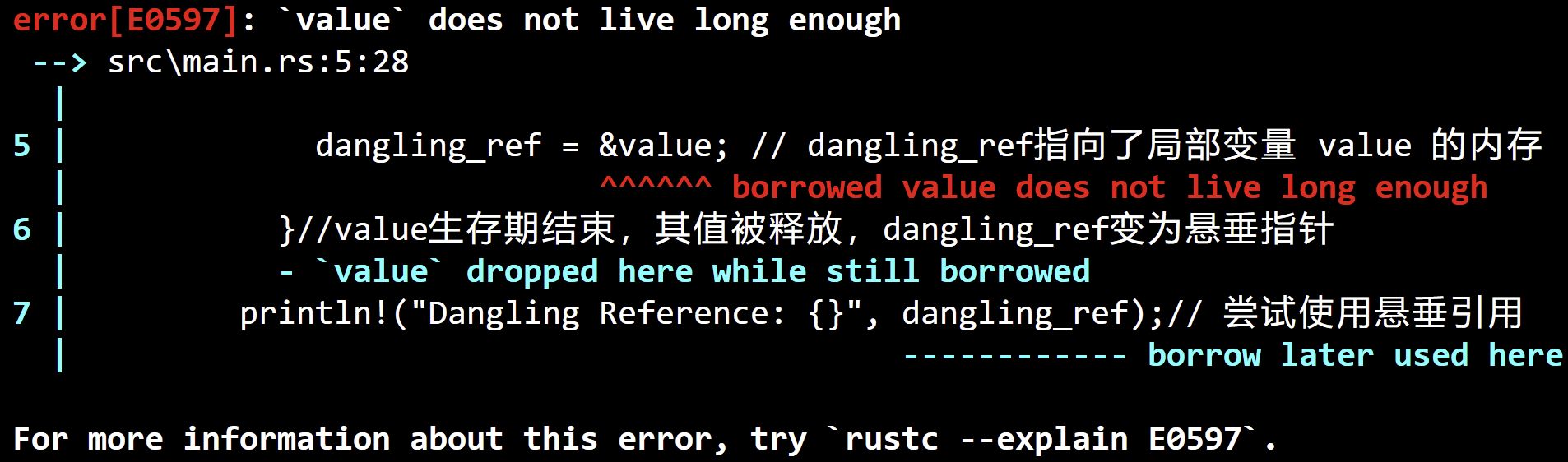


图2.5悬垂引用错误

在这段代码中，问题出现在尝试使用悬垂指针的地方。具体来说，当dangling\_ref被创建时，它指向了局部变量value，而value在它的作用域结束后被销毁了，dangling\_ref也随之变成了悬垂指针。因此，当尝试在println! 宏中使用dangling\_ref时，就出现悬垂引用的问题，因为此时它指向的内存已经无效。要解决此问题，可以把dangling\_ref变量的类型改为string，并将value变量的所有权转移给 dangling\_ref。修改后的代码如下：

1. fn main() {

2. let dangling\_ref: String; // 创建一个未初始化的string类型的变量

3.   {

5. let value = String::from("Hello World");

6. dangling\_ref = value; // 将 value 的所有权转移给 dangling\_ref

7. }

8.   println!("Dangling Reference: {}", dangling\_ref);// 使用 dangling\_ref

11. }

修改后，dangling\_ref 将拥有 value 的所有权，避免了悬垂引用的问题。Rust的所有权系统和借用规则旨在防止悬垂引用（Dangling References）的出现，确保引用总是指向有效的内存。

例3.悬垂引用示例2。

//注意！！！这段代码是为了演示内存不安全的情况，不能成功的编译运行

1. struct Data<'a> {

2. value: &'a mut i32,

3. } //定义结构体 Data，其中包含一个可变的 i32 类型数据的可变引用

4.  fn main() {

5. let mut value = 10;

6. let mut data = Data { value: &mut value }; //创建了一个 Data 结构体实例data

7. let reference = data.value;//创建可变引用reference

8.   value = 20;// 在这里修改了value的值，导致reference成为了悬垂引用

9.   println!("Reference: {}", reference);

10. }

//该代码示例的报错信息如下：

1. | let mut data = Data { value: &mut value };

2. | --------- `value` is borrowed here

3. ...

4. | value = 20;

5. | ^^^^^^^^^^ `value` is assigned to here but it was already borrowed

6. |

7. | println!("Reference: {}", reference);

8. | --------- borrow later used here

在这段代码中，问题出现在尝试在修改value后使用 reference 的地方。具体来说，在修改了value的值之后，尝试打印输出reference，但是由于reference是一个指向value的可变引用，当修改了value的值后，就会导致 reference 成为悬垂引用。为避免此类问题，应正确管理和注释引用的生存期。代码修改如下：

1. struct Data<'a> {

2. value: &'a mut i32,

3. } //定义了一个结构体 Data，其中value的生存期与Data实例一致

4.  fn main() {

6. let mut value = 10;

7. let mut data = Data { value: &mut value };

8. let reference = &mut data.value;

12. \*\*reference = 20;

/\* reference的生存期与data一致，因此即使在这里修改了value的值也不会释放value的内存空间，不会导致reference成为悬垂引用\*/

13.   println!("Reference: {}", reference);

在这里引入双重解引用“\*\*reference”对value进行修改。由于“reference”是一个&mut i32类型的可变引用，需要执行两次解引用操作才能修改所指向的值。代码的核心改进是通过使用生存期注解符号’a保证了reference的生存期与data一致，这样即使修改了值，也不会产生悬垂引用的错误。

例3.客户端与服务器连接示例，在建立连接中保证内存安全。

1. use std::io::{Read, Write};

2. use std::net::{TcpListener, TcpStream};

3. use std::thread;

4.

5. fn handle\_request(mut stream: TcpStream) {

6. let mut buffer = [0; 512]; // 用于存储请求的缓冲区

7.  // 从流中读取请求数据

8. match stream.read(&mut buffer) {

## 泛型与特型

泛型和特型是减少冗余代码的有效工具，可以有效提高代码的复用能力、增强程序的可读性和可理解性。泛型是具体类型或其他属性的抽象替代，即类型的参数化。特型是一种抽象地定义共享行为的方法。特型与泛型结合可以将泛型限制为只接受拥有特定行为的类型。

### 泛型与代码复用

对于众多重复的功能性代码可以通过函数进行提取和抽象，但是函数参数的类型是固定的，如果功能相近但是参数类型不一样，就需要将相同功能的函数定义两个，为解决这一问题，泛型应运而生。泛型（Generics）是一种编程范式，通过在定义时使用泛型参数（代表任意类型），可以将类型作为参数传递给函数或结构体等，并在需要时再对泛型参数具化，从而提高代码的重用性和灵活性。需要注意的是，泛型参数通常用单个大写字母表示，如T，U，W等。泛型本质是解决代码复用问题，一般在函数、结构体、枚举中应用较多。

1. **在函数定义中使用泛型**

在函数定义中使用泛型的方法是用一对尖括号将泛型参数括起来放于函数名之后，其后指定函数参数的数据类型和返回值类型，这些数据类型也可以是泛型参数，如下例所示。

1. fn print\_array<T>(arr: &[T]) //T是泛型参数，表示任意类型

2. where T: std::fmt::Debug, //T泛型参数所代表的类型必须实现Debug特型

3. {

4. for element in arr {

5. println!("{:?}", element);

6. }

7. }

8. fn main() {

9. let numbers = [1, 2, 3, 4, 5];

10. print\_array(&numbers);// 调用print\_array函数并传递numbers的引用作为参数

11.   let names = ["Alice", "Bob", "Charlie"];

12. print\_array(&names);// 调用print\_array函数并传递names的引用作为参数

13. }

在上面的代码中，我们定义了一个名为 print\_array 的函数，它接受一个类型为 &[T] 的参数 arr，其中 T 是一个泛型参数。这意味着 print\_array 函数可以接受任意类型的切片作为参数。在函数体内部使用了一个 for循环来遍历切片 arr中的元素，并使用 println! 宏打印每个元素。需要注意的是，where从句来指定 T 必须实现 std::fmt::Debug trait，以确保可以使用 {:?} 格式化符号打印元素。在 main 函数中，分别以不同类型的切片作为参数调用了 print\_array 函数。由于使用了泛型，print\_array 函数可以适用于不同类型的数据，无需为每个类型编写单独的函数。运行这段代码将换行输出：1, 2, 3, 4, 5，"Alice", "Bob", "Charlie"。

1. **在结构体定义中使用泛型**

定义结构时也可以使用 < > 运算符在一个或多个字段中使用泛型类型参数，具体方法为将<T >放于结构名之后，并在结构体中利用T来指明元素的类型。在实现结构体的方法时，要在impl和结构名之后都加上<T >，如下例所示。

1. struct Point<T> {

2. x: T,

3. y: T,

4. }

5.  impl<T> Point<T> {//实现带有泛型的结构体的方法，包括new，get\_x，get\_y三个方法

6. fn new(x: T, y: T) -> Point<T> {

7. Point { x, y }

8. } // 创建一个新的 Point 实例,

9.

10. fn get\_x(&self) -> &T {

11. &self.x

12. } // 获取 x 值的引用

13.

14. fn get\_y(&self) -> &T {

15. &self.y

16. }// 获取 y 值的引用

17. }

18.  fn main() {

19. let int\_point = Point::new(3, 5);// 创建一个整数类型的 Point 实例

20. println!("x: {}, y: {}", int\_point.get\_x(), int\_point.get\_y());

21. let float\_point = Point::new(1.5, 2.5);// 创建一个浮点数类型的 Point 实例

22. println!("x: {}, y: {}", float\_point.get\_x(), float\_point.get\_y());

23. }

在上面的代码定义了一个名为 Point 的结构体，它有两个泛型字段 x 和 y。这意味着 Point 结构体可以存储任意类型的数据作为其坐标。在 impl 块中，为 Point<T> 实现了3个方法。new方法接受两个参数x和y，并返回一个 Point<T> 实例。get\_x和get\_y方法分别返回x和y字段的引用。在 main 函数中，分别创建了一个整数类型的 Point 和一个浮点数类型的 Point。由于使用了泛型，Point 结构体和其方法可以适用于不同类型的数据，无需为每个类型编写单独的结构体和方法。

1. **在枚举定义中使用泛型**

枚举也可以使用通用数据类型。Rust标准库提供了Result<T, E>枚举，其中T和E都是通用数据类型。 Result<T, E>的使用方法如下例所示。

1. enum Result<T, E> {

2. Ok(T),

3. Err(E),

4. }

5.

6. fn divide(x: i32, y: i32) -> Result<i32, String> {

7. if y != 0 {

8. Result::Ok(x / y)

9. } else {

10. Result::Err(String::from("Division by zero"))

11. }

12. }

13.

14. fn main() {

15. let result1 = divide(10, 2); // 调用 divide 函数进行除法运算

17. match result1 {// 使用模式匹配处理结果

19. Result::Ok(value) => println!("Result: {}", value),

20. // 如果结果为 Ok，则打印结果值

21. Result::Err(err) => println!("Error: {}", err),

22. // 如果结果为 Err，则打印错误信息

23. }

24. let result2 = divide(10, 0);// 再次调用 divide 函数进行除法运算

26. match result2 {// 使用模式匹配处理结果

28. Result::Ok(value) => println!("Result: {}", value),

29. // 如果结果为 Ok，则打印结果值

30. Result::Err(err) => println!("Error: {}", err),

31. // 如果结果为 Err，则打印错误信息

32. }

33. }

上面的代码定义了一个名为 Result 的枚举，它有两个泛型参数 T 和 E。Result 枚举表示操作的结果，可以是成功的值 Ok(T) 或错误的值 Err(E)。在 divide 函数中，使用了 Result 枚举来表示除法操作的结果。如果除数不为零，返回Result::Ok枚举变体，并将结果作为Ok的值。如果除数为零，返回Result::Err枚举变体，并将错误消息作为Err的值。在 main 函数中，调用了divide函数两次，并使用match表达式对结果进行模式匹配。运行这段代码将换行输出以下结果：Result: 5，Error: Division by zero。

从以上示例可知，泛型不仅可以提高代码的复用性，还可以提高代码的可读性和可维护性，因为它可以使代码更加抽象和通用。泛型还可以提高代码的安全性，因为它可以确保在编译时对类型进行检查，从而避免运行时的类型错误。

### 特型与抽象

在创建和应用泛型时会发现，有时候多个泛型会用到同样的方法，这时能不能再进一步进行代码复用，减少冗余代码呢？Trait（特型）便是解决这一问题的有效工具。Rust中的trait与Java和C++中的接口的理念相同，可以在其中定义多种类型的共享行为。

1. **Trait定义**

trait的定义比较简单，以trait关键字开头，然后是特型名称，之后是一对花括号。在花括号内是一系列的方法签名和默认方法的实现代码，非默认方法的实现则交给了后续调用trait时进行。后面跟着trait的名称。还可以在trait中定义常量，所有实现者都可以共享它。在这里，我们定义了一个名为network的特征，其中包含三个方法，如下所示。

1. trait Network {

2. fn connect(&self);

3. fn send(&self, data: &[u8]);

4. fn receive(&self) -> Vec<u8>;

5. }

Trait中定义了一组方法，用于描述某些类型所具有的共享行为，而不关心具体的实现细节。要实现一个trait，需要在类型上使用impl关键字，并为trait中定义的每个方法提供具体的实现代码。这样，类型就可以被视为实现了该trait，并可以使用trait中定义的方法。以下展示如何实现trait的方法：

1. struct TcpSocket {

2. address: String,

3. port: u16,

4. }// 首先定义实现`Network` trait的结构体

5.

6. impl Network for TcpSocket {

7. fn connect(&self) {

8. println!("连接到 {}:{}", self.address, self.port);

9. }

10.   fn send(&self, data: &[u8]) {

11. println!("发送数据: {:?}", data);

12. }

13.  fn receive(&self) -> Vec<u8> {

14. println!("接收数据");

15. vec![1, 2, 3, 4, 5]

16. }

17. }

18.  fn communicate<T: Network>(device: T) {//限制T为实现了Network 特型的类型，即trait约束

19. device.connect();

20. device.send(b"Hello, World!");

21. let data = device.receive();

22. println!("接收到的数据: {:?}", data);

23. }

24.

25. fn main() {

26. let socket = TcpSocket {

27. address: String::from("127.0.0.1"),

28. port: 8080,

29. };

30.   communicate(socket);

31. }

在上面的例子中，定义了一个名为Network的trait，它包含了connect、send和receive方法。然后，创建了一个名为TcpSocket的结构体，并为其实现了Network trait。在main函数中，调用了communicate函数，并将一个TcpSocket实例socket作为参数传递给它。communicate函数的定义中使用了泛型参数T: Network，这意味着必须是实现了Network trait的类型才可以作为函数参数device的类型，而socket是TcpSocket的一个实例，因此可以调用connect、send和receive方法，而无需关心具体的实现细节。

通过使用trait，我们可以在不同的类型之间共享行为，这在网络编程中特别有用。例如，我们可以定义一个UdpSocket结构体，并为其实现Network trait，然后将其传递给communicate函数，以便与UDP套接字进行通信。这样，我们便可以轻松地切换不同类型的网络设备，而无需更改communicate函数的实现。

Trait还可以提供方法的默认实现，用于在类型不重写这些方法时提供默认的行为代码，默认实现只需直接将方法代码写于trait定义中的方法签名后即可。当然，在某一类型实现Trait时还可以选择重写这些方法。Trait的默认实现如下例所示。

1. trait MyTrait {

2. fn my\_method(&self) {

3. println!("This is the default implementation of my\_method");

4. } // 默认实现

5. } // MyTrait的定义中直接实现了my\_method方法的代码，因此它为默认实现

6.

7. struct MyStruct;

8. impl MyTrait for MyStruct {

9. // 不提供具体的实现，使用默认实现

10. }

11. fn main() {

12. let my\_struct = MyStruct;

13. my\_struct.my\_method(); // 输出：This is the default implementation of my\_method

14. }

在上面的例子中，MyTrait定义了一个方法my\_method，并为其提供了一个默认实现。为MyStruct实现MyTrait时没有为my\_method提供具体的实现。因此，在main函数中调用my\_struct.my\_method时，会使用默认实现输出相应的信息。

在使用默认实现时要注意，默认实现只能在trait的定义中提供，而不能在为某一类型具体实现trait时提供。

1. **Trait约束**

在Rust中，trait约束用于指定泛型类型参数必须满足的特定行为或功能。通过使用trait约束，可以在编译时对泛型类型参数进行静态检查，以确保其具有所需的方法或功能。通过在泛型类型参数后面使用where关键字（有些像sql语句），可以指定一个或多个trait作为约束条件。要定义一个trait约束，可以在泛型类型参数后面使用冒号，然后紧跟着trait名称。下面即是一个使用trait约束的示例：

1. fn print<T: ToString>(value: T) {

2. let s = value.to\_string();

3. println!("{}", s);

4. }

在上面的示例中，print函数接受一个泛型类型参数T，并对其进行了trait约束。具体来说，T必须实现ToString trait，该trait定义了将值转换为字符串的方法。通过这个trait约束，我们可以在函数内部调用to\_string方法，而不需要知道具体的类型。这样，我们可以在编译时确保传递给print函数的值具有ToString trait所需的功能。

需要注意的是，可以同时指定多个trait约束，只需使用加号将它们连接起来即可。例如：

1. fn process<T: Display + Clone>(value: T) {

2.

3. }//T为既实现了Display特型又实现了Clone特型的类型

以上代码也可以如下这样写:

1. fn process<T> (value: T) where T: Display + Clone{

2.

3. }//用where子句约束T可取的类型

例4. 在网络通信中使用trait约束定义网络相关行为和功能的综合示例。该例首先定义了一个名为Network的trait，它表示网络共有行为功能的组合。然后定义了结构体TcpConnection，并为其提供了Network trait的具体实现。接下来，定义了一个名为perform\_network\_operations的函数，它使用泛型参数T并对其施加了Network trait约束。这意味着我们只能使用实现了Network trait的类型作为函数泛型参数。在函数内部可以调用T类型的trait方法。最后，在main函数中，我们创建了一个TcpConnection实例，并将其传递给perform\_network\_operations函数。由于TcpConnection实现了Network trait，所以它满足了函数对泛型参数的约束。

1. trait Network {

2. fn connect(&self);

3. fn send(&self, data: &[u8]);

4. fn receive(&self) -> Vec<u8>;

5. fn disconnect(&self);

6. }// 定义一个名为`Network`的trait，其中定义了网络实体的行为功能

7.

8. struct TcpConnection {

9.

10. }// 定义网络实体的类型为结构体TcpConnection

11.

12. impl Network for TcpConnection {

13. fn connect(&self) {

14. println!("连接到TCP服务器");

15. }

16.   fn send(&self, data: &[u8]) {

17. println!("发送数据: {:?}", data);

18. }

19.   fn receive(&self) -> Vec<u8> {

20. println!("接收数据");

21. vec![1, 2, 3, 4, 5]

22. }

23.   fn disconnect(&self) {

24. println!("断开与TCP服务器的连接");

25. }

26. }

27. fn perform\_network\_operations<T: Network>(network: T) {//使用泛型参数和trait约束的//函数

28. network.connect();

29. network.send(&[1, 2, 3]);

30. let received\_data = network.receive();

31. println!("接收到的数据: {:?}", received\_data);

32. network.disconnect();

33. }

34. fn main() {

35. let tcp\_connection = TcpConnection {};

36. perform\_network\_operations(tcp\_connection);

37. }

约束的优点是：通过使用trait约束，我们可以在泛型代码中限制类型的行为，以确保其满足特定的要求。这有助于提高代码的可靠性和可复用性。

1. **Trait继承**

通过trait的继承可以构建更通用和更灵活的trait层次结构，以便在不同的类型和场景中重用代码。子trait可以继承父trait的方法和关联类型。这意味着实现了子trait的类型必然实现了父trait中定义的所有方法，并可以在子trait中扩展父trait的行为。需要注意的是，Rust中的trait继承是单向的，即子trait继承父trait，但父trait不会自动继承子trait，也就是说实现了子trait的类型也必须显式地实现父trait。在子trait名后面跟上“：父trait名”即定义了他们之间的继承关系。

例5. 使用trait继承定义不同层次的网络操作，实现网络实体之间的通信。其中定义一个父trait：NetworkConnection ，它包含了与网络连接相关的方法，如connect和disconnect。然后，定义子 trait：TcpConnection，它继承了NetworkConnection trait，并添加了与TCP连接相关的方法，如send和receive。示例如下：

1. trait NetworkConnection {

2. fn connect(&self);

3. fn disconnect(&self);

4. }

5.

6. trait TcpConnection: NetworkConnection {//定义TcpConnection和NetworkConnection的//继承关系

7. fn send(&self, data: &[u8]);

8. fn receive(&self) -> Vec<u8>;

9. }

10.

11. struct TcpSocket;

12.

13. impl NetworkConnection for TcpSocket {

14. fn connect(&self) {

15. println!("TCP socket connected");

16. }

17.

18. fn disconnect(&self) {

19. println!("TCP socket disconnected");

20. }

21. }

22.

23. impl TcpConnection for TcpSocket {

24. fn send(&self, data: &[u8]) {

25. println!("Sending data over TCP: {:?}", data);

26. }

27.

28. fn receive(&self) -> Vec<u8> {

29. println!("Receiving data over TCP");

30. vec![1, 2, 3, 4, 5]

31. }

32. }

33.

34. fn main() {

35. let tcp\_socket = TcpSocket;

36. tcp\_socket.connect();

37. tcp\_socket.send(&[1, 2, 3]);

38. let data = tcp\_socket.receive();

39. tcp\_socket.disconnect();

40. }

在这个例子中，首先定义了两个trait：NetworkConnection和TcpConnection。TcpConnection继承了NetworkConnection，因此它包含了NetworkConnection中定义的connect和disconnect方法。然后创建了一个名为TcpSocket的结构体，并为其实现了NetworkConnection和TcpConnection这两个trait。在TcpSocket中实现了所有trait中定义的方法。在main函数中，创建了一个TcpSocket实例，并调用了它的连接、发送、接收和断开连接的方法。由于TcpSocket实现了TcpConnection和NetworkConnection，因此我们可以使用这两个trait中定义的方法。

例6.客户端和服务器通信的综合示例。客户端首先发送连接请求，服务器收到连接请求后，建立与客户端的连接，之后双方开始发送数据，数据发送完毕，双方断开连接。

1. use std::io::{Read, Write};

2. use std::net::{TcpListener, TcpStream};

3. use std::thread;

4.

5. // 定义一个处理客户端连接的函数

6. fn handle\_client<T: Read + Write>(mut stream: T) {

7. let mut buffer = [0; 512];

8. loop {

9. // 从客户端读取数据

10. let bytes\_read = stream.read(&mut buffer).expect("Failed to read from client");

11. if bytes\_read == 0 {

12. // 客户端关闭连接

13. break;

14. }

15.

16. // 处理接收到的数据

17. let received\_data = String::from\_utf8\_lossy(&buffer[..bytes\_read]);

18. println!("Received data: {}", received\_data);

19.

20. // 将数据发送回客户端

21. stream.write\_all(&buffer[..bytes\_read]).expect("Failed to write to client");

22. }

23. }

24.

25. fn main() {

26. // 启动服务器并监听指定的地址和端口

27. let listener = TcpListener::bind("127.0.0.1:8080").expect("Failed to bind address");

28.

29. // 接受客户端连接并为每个连接启动一个线程处理

30. for stream in listener.incoming() {

31. match stream {

32. Ok(stream) => {

33. // 为每个连接启动一个新线程处理

34. thread::spawn(move || {

35. handle\_client(stream);

36. });

37. }

38. Err(e) => {

39. println!("Failed to establish a connection: {}", e);

40. }

41. }

42. }

43. }

这个例子创建了一个简单的TCP服务器，监听本地地址（127.0.0.1）的8080端口。服务器接受客户端连接，并为每个连接启动一个新线程来处理客户端的请求。handle\_client函数用于处理客户端连接，它使用泛型类型T来支持不同类型的流（例如TcpStream、BufStream等）。在handle\_client函数中，循环读取客户端发送的数据，并将其发送回客户端。当客户端关闭连接时退出循环。

## 错误处理

程序中的错误通常分为两类：不可恢复的错误和可以恢复的错误。对于可恢复错误，比如文件未找到，一般是报告给用户，让其重试；而[不可恢复错误](https://zhida.zhihu.com/search?content_id=241124925&content_type=Article&match_order=1&q=%E4%B8%8D%E5%8F%AF%E6%81%A2%E5%A4%8D%E9%94%99%E8%AF%AF&zhida_source=entity)，比如数组访问越界了，则会引起程序进入异常状态。在有[异常处理](https://zhida.zhihu.com/search?content_id=241124925&content_type=Article&match_order=1&q=%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86&zhida_source=entity)的编程语言中，通常并不详细区分这两种错误，而是统一交由异常处理机制处理。Rust没有异常处理机制，通常对于不可恢复错误，会采用panic结束程序；而对于可恢复错误，则更倾向通过显示的机制进行[错误捕获](https://zhida.zhihu.com/search?content_id=241124925&content_type=Article&match_order=1&q=%E9%94%99%E8%AF%AF%E6%8D%95%E8%8E%B7&zhida_source=entity)和传递。对于可恢复错误，Rust采用Result类型来描述。本节主要对可恢复错误和不可恢复错误这两种错误的处理机制进行探讨。

### 空值处理

在 [Rust](https://so.csdn.net/so/search?q=Rust&spm=1001.2101.3001.7020) 中，Option提供了一种安全且优雅的方式来处理可能存在或不存在的值，避免了空指针异常的问题。Option<T>是Rust标准库中的一个[枚举类型](https://so.csdn.net/so/search?q=%E6%9E%9A%E4%B8%BE%E7%B1%BB%E5%9E%8B&spm=1001.2101.3001.7020)，可以用它来表示任意一个可能存在空值的值，定义如下。

1. pub enum Option<T> {//T为泛型，表示要判断是否为None的表达式的类型

2. Some(T), //包含T类型的某些值

3. None, //没有值

4. }

Rust中没有Null值的概念。在Rust中，可能失败并希望指示缺失相应值的API会返回Option类型的值。Option中的None与Null类似，但None把Null检查显式化了，由此避免了其它语言中常见的Null引起的各种错误和异常。Option常常与Match搭配来判断变量值是否为空，并给出进一步的动作，如下例所示。

1. use std::collections::HashMap; //使用std标准库的HashMap类型
2. fn main() {

3.   let mut a=HashMap::new();

4.    a.insert("math", 98);

5.    a.insert ("chinese", 58);//在数组a中插入两个键值对：“math” 98, “chinese” 58

6.   let incremented\_value = match a.get("chinese"){

7.        Some(val)=> val+2, //get方法调用成功，匹配此分支，incremented\_value=60

8.       none => 0 //get方法调用不成功，匹配此分支，incremented\_value=0

9.    };

10.    println!("{}", incremented\_value);

11. }

程序中第6行，当get方法调用成功时，some变体中包含"chinese"键值对应的分数58；当get方法调用不成功时，返回none值。此例利用option类型保证了get方法的返回值的各种情况都得到了妥善处理，从而提升了程序的可靠性。这里要强调的是，使用模式匹配来处理返回值，调用者必须处理结果为None的情况，这往往是一个好的编程习惯，可以减少潜在的错误。

### 可恢复错误处理

Rust中的可回复错误由Resultt来处理。与Option<T>一样，Result<T，E>也是Rust标准库中的一个[枚举类型](https://so.csdn.net/so/search?q=%E6%9E%9A%E4%B8%BE%E7%B1%BB%E5%9E%8B&spm=1001.2101.3001.7020)，其中包括Ok和Err两个变体。Result枚举通过Err变体来捕获并传递错误，具体定义如下。

1. enum Result<T,E> {//T和E为泛型，T为成功时返回的值的类型，E为失败时返回的值的类型

2. Ok(T), //表示操作成功，操作结果赋给OK()变体中的变量

3. Err(E), //表示操作失败，操作结果赋给Err()变体中的变量

4. }

下面的示例展示了如何使用 Result 类型处理错误。示例实现的功能是，读取文件，若读取成功，返回一个包含文件句柄的Ok变体，若读取失败(多种原因导致失败，如文件不存在，或没有权限访问这个文件)返回一个包含错误信息的Err变体。

1. use std::io::Error; //标准库中的错误类型，提供了一种统一的方式来处理各种 I/O 错误情况

2. fn main(){

3. let path="hello.txt";

4. let try\_read=read\_file(path); // 从Rust类型推导机制得出try\_read为Result类型

/\*上句中read\_file函数返回值类型为Result。如果读取文件成功，将文件内容读入Ok（T）变体中的变量；否则将错误信息写入Err(E)中的变量。\*/

6. match try\_read{ //匹配返回值并处理

7. Ok(file)=> println!("{}",file), //读取成功,打印文件内容

8. Err(e)=>println!("Error reading {} owing to {}",path,e),

//上句代码的意图是读取失败,打印错误信息

9. };

10. }

11. fn read\_file(path:&str)->Result<String,Error>{ //返回Result类型的值

12. std::fs::read\_to\_string(path) //将文件内容读入Result的OK变量中

13. }

图2.4是程序运行后的输出结果。从此程序的运行结果中我们可以发现，读取文件hello.txt出错了，其原因是系统找不到该文件。os error 2是Rsut编译器给出的错误号，意指操作系统错误，没有这样的文件或目录。

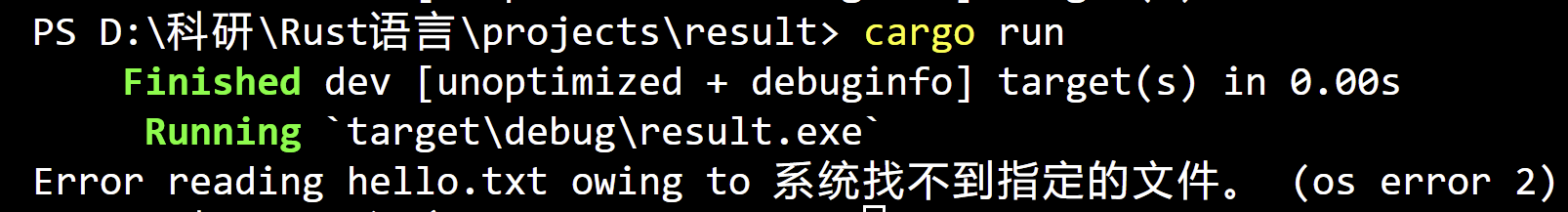


图2.4 Result使用示例程序运行结果

由于Option和Result均是包装器类型，如若需要与其内部值进行交互就需要用模式匹配等方式，这样使得程序代码冗长且可读性差，为解决这一问题，Rust给出了与包装器类型配合使用的组合器，它允许程序员直接提取成员的内部值，而不再使用match表达式。常与OPtion/Result 配合使用的组合器方法是unwrap和expect，下面对其进行详细介绍。

如果Option变量的值为None，则导致程序panic而终止。在确认Option变量的值不为None的情况下，可以用unwrap方法拆解出其中的值，并获取值的所有权。这里要强调的是，unwrap会消费Option本身的值，后面就不能再用了，如下例所示。

1. let a = Some(5);
2. let d = a.unwrap();

// println!("{:?}", a); // 变量a已经被消耗掉，不能再使用

1. println!("{:?}", d); //变量a的所有权转移给变量d，所以d可以被使用

unwrap( )与Result配合使用时，若Result变量的值是成员Err，则unwrap( )会自动调用panic终止程序，在panic中还会显示错误信息；若Result变量的值是成员Ok，unwrap( )方法直接返回Ok中的值，如下例所示。

1. use std::fs::File;

2. fn main(){

3. let\_file=File::open("hello.txt").unwrap(); //File::open()返回值类型为Result

4. }

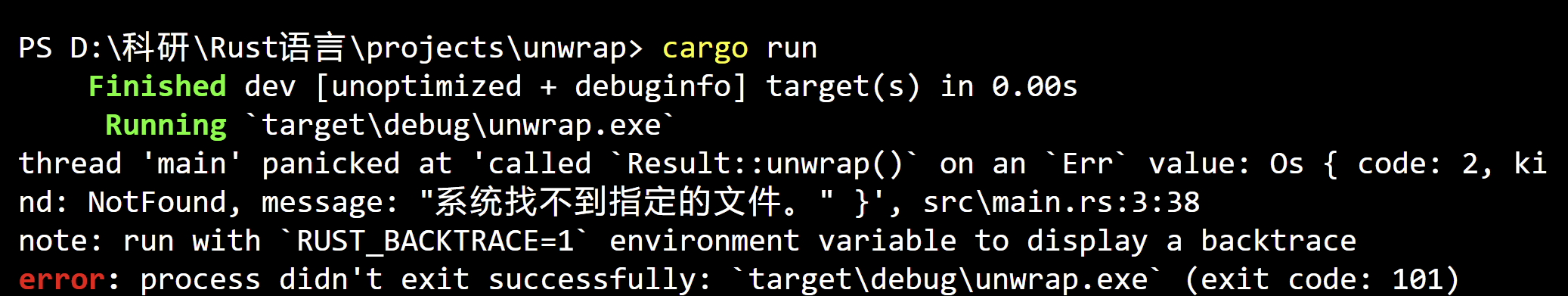


图2.5 unwrap使用示例程序运行结果

图2.5是程序运行后的输出。从结果得知，程序运行时引起了panic（不可恢复错误，将在下一节中重点阐述），究其原因是File::open("hello.txt")方法不能打开“hello.txt”文件，因为此文件并不存在，因此File::open("hello.txt")方法的运行结果是Result的Err成员，它再调用unwrap方法时引起了panic。

下例显示了Result变量值是Ok变体时，用unwrap方法提取Ok(T)内部值的方法。该例首先定义了一个Result类型的变量result，并给它赋值Ok(58)，之后便用unwrap方法把58提取出来加2后赋给i32变量value。显然，与利用match表达式提取Result类型变量的内部值相比，unwrap方法更加简单方便。

1. fn main(){

 2.   let result: Result<i32, &str> = Ok(58);

 3.   let value = result.unwrap()+2;

 4.   println!("the score is: {}", value);

   }

expect( )提供了与unwrap( )相同的功能，但允许附加一个自定义的错误消息。具体来说，当Result值是 Ok(T) 类型时，它会返回其中包含的有效值T；当结果是Err(E)类型时，它会引发panic，并打印出作为参数传入的错误信息。这在调试和错误追踪中极为有用，因为它可以提供更多的上下文信息，如下例所示。

1. use std::fs::File;

2. fn main() {

3. let file = File::open("hello.txt").expect("hello.txt should be exist");

4. }

图2.6是该程序运行后的输出。其中“panicked at ”后面便是我们自己设置的提示信息“hello.txt should be exist”，之后是系统给出的错误信息，包含错误码为2，错误类型为“NotFound”，错误提示信息为“系统找不到指定的文件”。

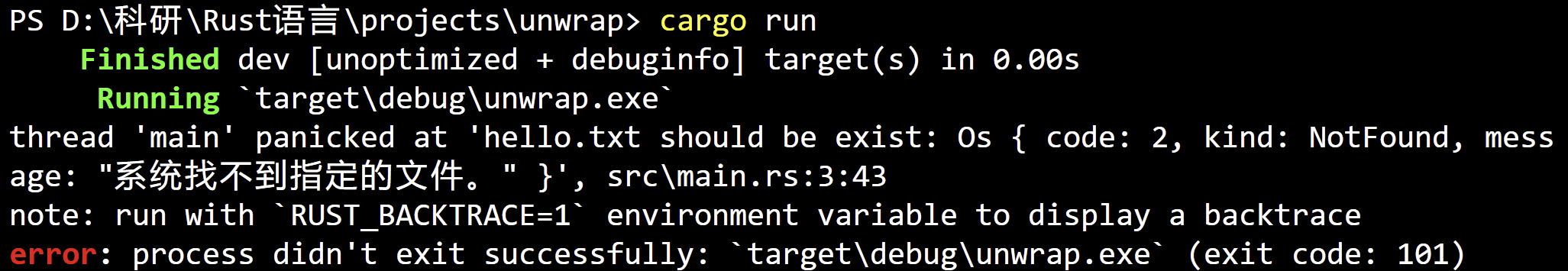


图2.6 expect使用示例程序运行结果

### 不可恢复错误处理

不可恢复错误通常是由于编程中的逻辑错误导致的，比如尝试访问数组时索引超出了数组的范围、Result变量的值为Err（这种错误是无法解决的，如文件不存在时却要读取文件内容）、整数除以0、断言失败等，这样的操作是不可能成功的，是典型的不可恢复错误。当发生这些错误时，Rust就会调用panic!宏来处理这些错误，打印错误信息、展开调用栈、清理当前函数及其调用者的数据、终止程序运行。下面的例子展示了程序调用panic！宏的效果。

1. fn main() {

2.     panic!("it is a panic!");

3. }

该程序的运行效果如图2.7所示。由于在main函数中调用了panic！宏，所以在输出的提示信息中，第三行显示了panic所提供的信息thread 'main' panicked at 'it is a panic!'，src\main.rs:2:5，说明panic发生在程序的第二行第五个字符处，并给出了程序中panic！宏携带的提示信息。第四行提示通过设置环境变量`RUST\_BACKTRACE=1`可以展示回溯信息。第五行报错表明程序没有成功退出。

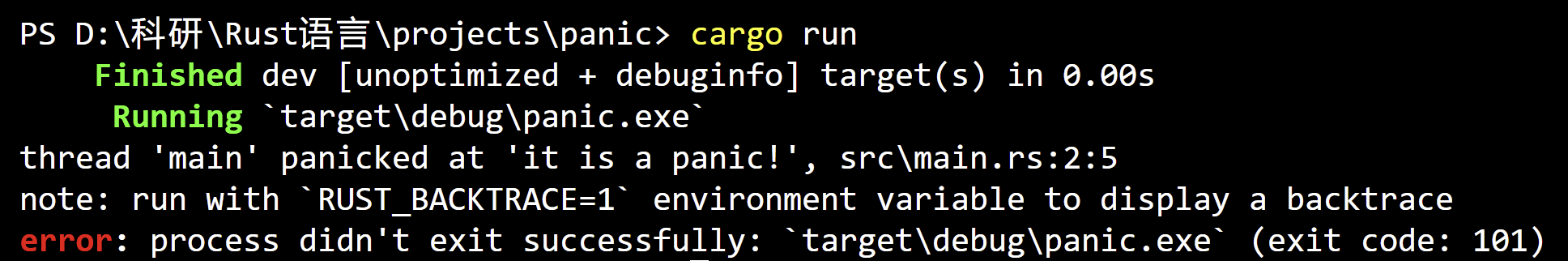


图2.7 panic示例程序运行结果

什么是回溯信息呢？回溯信息能够帮助我们定位错误发生的位置。回溯信息中包含了到达错误点的所有调用函数列表。可通过设置RUST\_BACKTRACE=1得到回溯信息，进而确定触发错误的原因，如图2.8所示。图中第一行为设置环境变量，输出回溯信息，之后再运行程序即可得到带有回溯信息的运行结果。图中展示了所有调用函数列表。

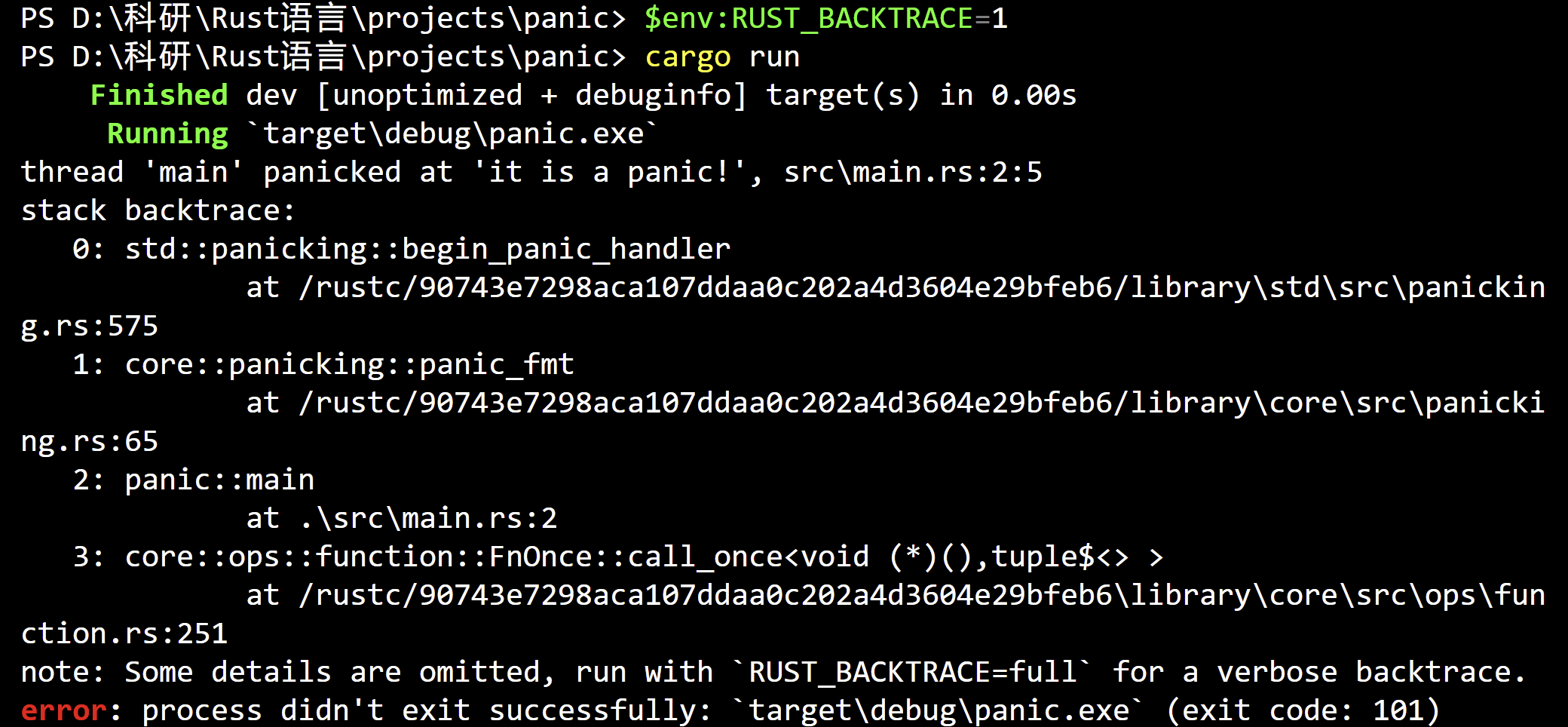


图2.8 带有回溯信息的运行结果

在上例中我们在程序中直接调用了panic！宏，但更常见的情况是因为发生了不可恢复错误而由系统自动调用panic！，并终止程序运行。下例展示了因数组越界错误而调用panic！的情况。

1. fn main() {

2.     let v = vec1,2,3,4];

3. v[99];

4. }

运行结果如图2.9所示。编译器给出了导致panic的原因"panicked at 'index out of bounds: the len is 4 but the index is 99', src\main.rs:4:5"。Rust为保证内存安全，当尝试读取非法索引时，抛出了panic并终止程序运行。这就是著名的数组越界访问问题。

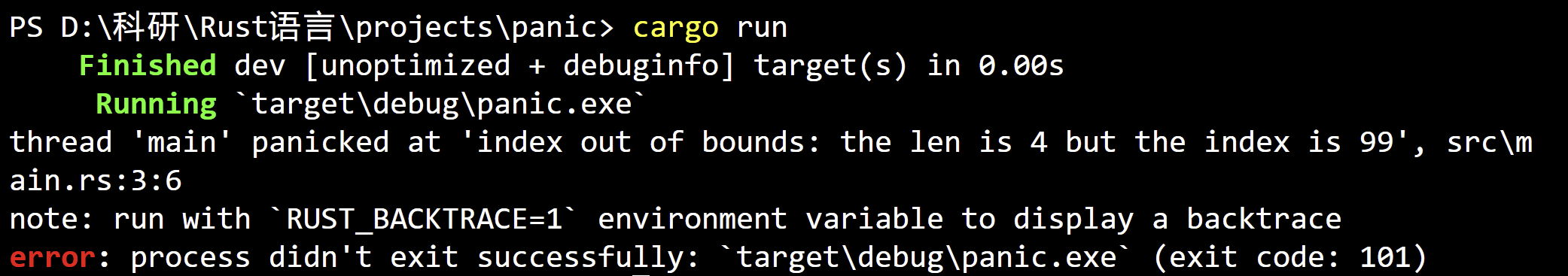


图2.9 程序终止运行输出

## 并发处理

随着多核处理器成为主流，并发程序设计已成为大多数编程语言的基本概念，各编程语言均提供了多种技术、方法和机制来保证多线程并发执行的安全性和效率。Rust借助其特有的所有权和类型系统将许多并发问题在编译期检查并解决，即编译器会检查出具有并发缺陷的代码，并提供解释问题的错误信息，而不是在运行后再暴露并发问题。因此，Rust成功编译后的代码是不会出现常见的并发问题的，这也被称为Rust的“无畏并发”(fearless concurrency)。“无畏并发”成就了Rust易于操作且安全高效的并发编程。在本节中，我们将介绍与并发编程相关的一些基本概念和Rust的并发机制。

### 线程管理

进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位，然而使用进程进行并发处理的时空开销较大。为使多个程序并发执行的同时尽量减少系统开销、提高并发效率，人们又引入了“线程”的概念。线程是进程的基本执行单元，是操作系统（OS）中独立调度和分派的基本单位。同一进程的不同线程、不同进程的多个线程均能并发执行，从而有效提高了系统资源的利用率和吞吐量。

1. 使用thread::spawn函数创建线程

Rust语言采用调用操作系统API创建线程，即一个OS线程对应一个Rust线程。Rust提供了thread标准库进行多线程编程。调用thread库中的spawn 函数可以创建一个新的线程。spawn 函数接受一个闭包或者函数类型的参数，其中包含希望新线程运行的代码，Rust会启动一个新线程来运行该闭包或函数的代码，如下例所示，运行结果如图2.10所示。

1. use std::thread;

2. fn main() {

3. for i in 1..5{

4. thread::spawn(move || { println!("线程{}已创建！",i); })

5. };//上述for语句利用 thread::spawn 函数创建了新线程，函数参数为一个无参闭包；

//move关键字将i的所有权移动到闭包内部

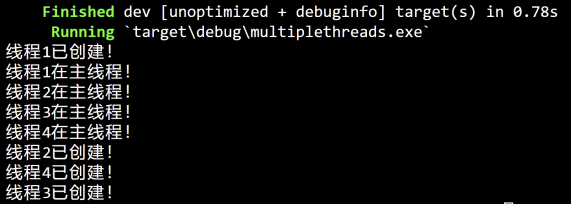
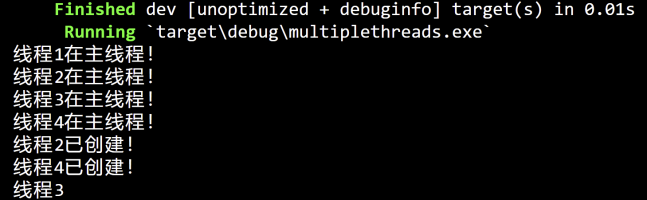
6.   for i in 1..5{

7. println!("线程{}在主线程！",i);

8. }

9. }

本例通过将无参闭包作为thread::spawn函数的参数创建新的线程。第3-5行的for语句循环4遍，创建了4个线程，打印输出“线程i已创建！”。Main函数中第6-8行还有一个for语句，循环4遍打印输出“线程i在主线程！”，程序运行结果图2.10所示。左图为上例第一次运行结果，右图为上例第二次运行结果。细心的读者会发现，相同的代码，两次运行的结果并不相同，这是为什么呢？

(a)第一次运行 (b)第二次运行

图2.10 多线程并发运行结果

原来在main函数中每一次对spawn的调用都会创建一个新的子线程并立即返回main函数执行spawn函数之后的代码。这个子线程将和main函数主线程并发执行，其执行顺序由OS依据程序运行时的上下文环境而定、不受用户控制，因此我们看到两次执行结果并不相同。值得注意的是，当主线程结束时，不论子线程的任务是否完成都要立即停止执行，这也解释了上右图的现象。从中我们感受到主线程的for语句执行结束时主线程也就结束了，但此时子线程的for语句还没有执行结束就被强制停止了。

1. **使用join方法等待所有线程结束**

thread::spawn函数的返回值类型为JoinHandle，调用它的join方法即可阻塞主线程直到子线程运行结束。通常情况下，首先将thread::spawn函数的返回值赋给一个JoinHandle类型的变量，再在main 函数退出之前，让该变量调用 join 方法，其返回值为Result类型。下例展示join句柄的使用方法。

1. use std::thread;

2. fn main() {

3. let handle = thread::spawn(|| {

4. for i in 1..5{

5. println!("线程{}已创建！",i);

6. }

7. });//利用 thread::spawn 函数创建了一个新线程，函数参数为一个无参闭包

8.   for i in 1..5{

9. println!("线程{}在主线程！",i);

10. }

11. handle.join().unwrap();

12. }

程序运行结果如图2.11所示。通过运行结果可以观察到，主线程和子线程均执行完毕，也就是说子线程执行完毕后主线程才会退出，所以该例的输出更加完整，join方法保证了线程运行的可靠性。

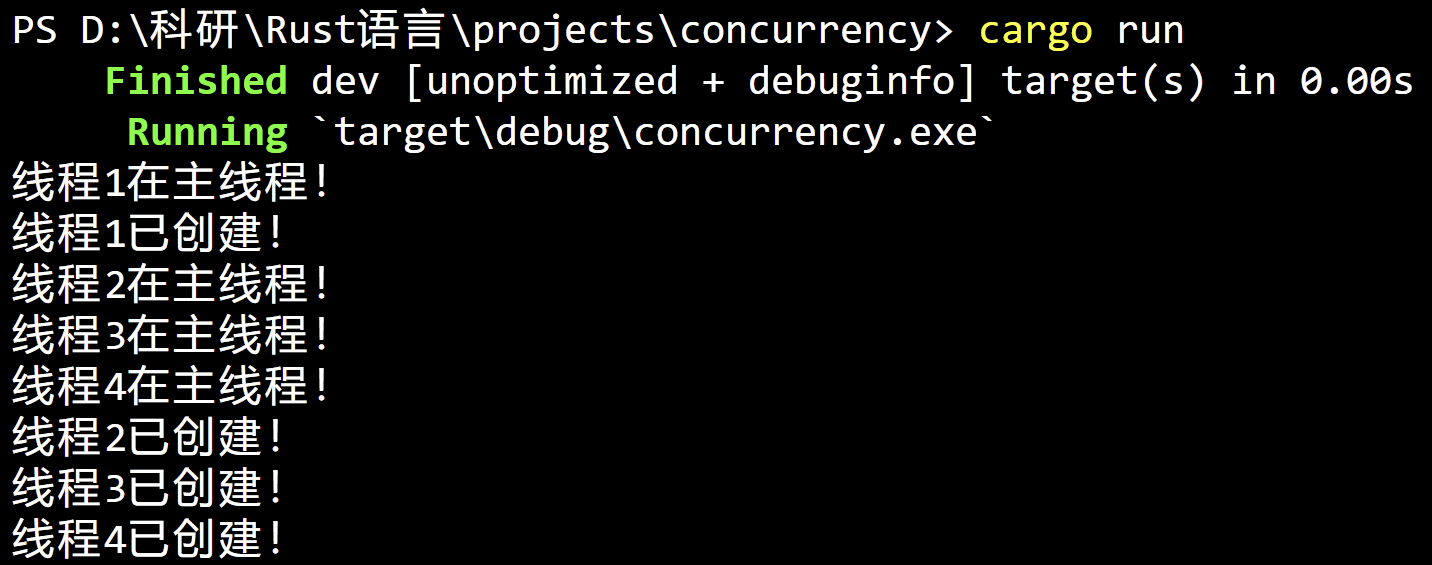


图2.11 线程运行状态

1. **move关键字**

当创建的新线程需要调用其它线程的数据时，就会碰到一个棘手的问题：无法判断其他线程变量的生存期，即在我们使用该变量时，它可能已经被销毁了！我们来看下面的例子。

1. use std::thread;

2. fn main() {

3. let v = vec![1, 2, 3];

4.   let handle = thread::spawn(|| {

5. println!("这是一个动态数组: {:?}", v);

6. });

7. drop(v);

8.   handle.join().unwrap();

9. }

在本例中，子线程在闭包中调用了主线程的动态数组，之后在主线程中利用drop函数释放掉了动态数组的所有权，这就导致了如上文所讲的所有权问题，因此，在运行结果中报告了错误。图2.12给出了编译报错的清单，其中第4行提示“闭包中借用了数组v的值，有可能超出其生存期”，即在子线程中调用了主线程中的变量，此变量有可能在主线程中被释放的情况下在子线程中仍然使用，这是非常严重的错误。如果 Rust 允许前面的代码通过编译，那么子线程可能已经访问了主线程返回后包含一些垃圾值的v，并且可能导致分段错误。Rust提供的帮助信息告诉我们，需要在闭包前添加move关键字，使得新建线程获得变量的所有权，从而使程序能够正常运行。

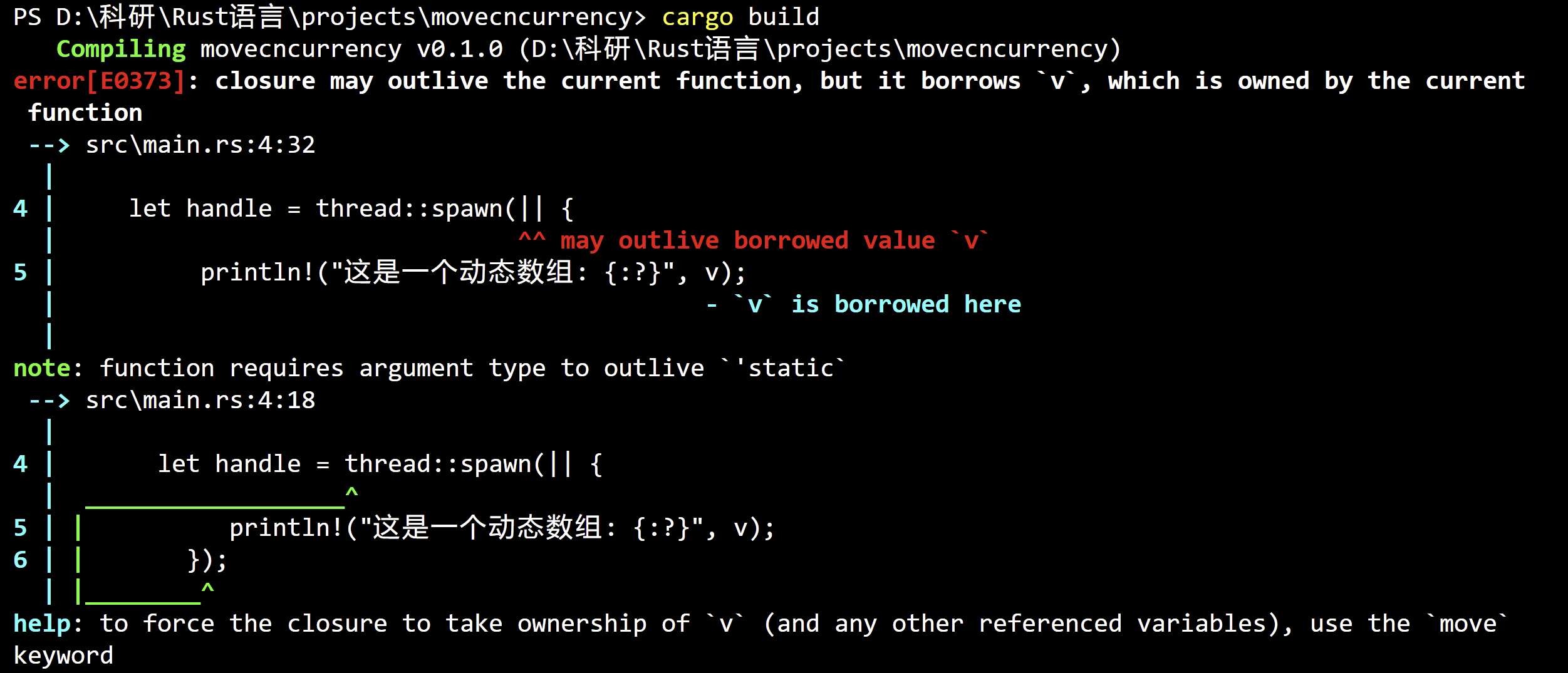


图2.12 “所有权”异常抛出

如果不使用move关键字，Rust只是提供数据的一个借用，那么Rust不知道这个线程会运行多久，也就不知道数据的引用是否一直有效。这是强调内存安全的Rust语言所不能容忍的事情。所以，我们可以在闭包前通过引入move关键字强制闭包从外界获取数据的所有权，这样就保证了Rust生存期和所有权的安全特性。需要注意的是，一旦子线程获取了数据的所有权，父线程就不能再访问此数据了，如下例所示。

1. use std::thread;

2. fn main() {

3. let v = vec![1, 2, 3];

4.   let handle = thread::spawn(move || {

5. println!("这是一个动态数组: {:?}", v);

6. });

7. drop(v);

8.   handle.join().unwrap();

9. }

这里虽然已经在闭包前添加了move关键字，却仍然出错了，如图2.13所示的运行结果。编译器给出了这样的出错信息：在程序的第7行，主线程中使用了已经移动到子线程里的值。所以，要想移除这个错误，就需要删掉“drop(v)”这一条语句。



图2.13 程序错误结果输出

### 线程间消息传递机制

消息传递模型（message passing model）广泛应用于并发程序设计中。在这种模型中，每个进程都有自己的私有地址空间，进程之间不能直接访问对方的内存，只能通过发送消息来进行通信。这种机制避免了共享内存中的竞态条件和死锁问题，保证了数据的一致性，提高了系统的稳定性和可靠性。进程之间是利用通道来传递消息的，Rust标准库std::sync::mpsc提供了通道（channels）的实现。通道由发送者（sender）和接收者(receiver)构成，前者负责向通道中发送信息，后者负责从通道中接收信息。mpsc（multiple producer, single consumer ）指一个通道可以有一个或多个发送者，但只能有一个接收者。当通道的一端被释放时通道就被关闭了。我们将通过下面的例子去观察通道的创建，以及利用通道传递消息的过程。为简单起见，这里的通道只有一个发送者和一个接收者。

1. use std::sync::mpsc;

2. use std::thread;

3. fn main() {

4. let (sender, receiver) = mpsc::channel();

5. thread::spawn(move || {

6. let s = String::from(" Rust Network Programming");

7. sender.send(s).unwrap();

8. });

9. let received = receiver.recv().unwrap();

10. println!("Got: {}", received);

11. }

上述代码的第4行调用mpsc::channel函数创建了一个通道，函数的返回值是元组类型，包含一个发送者和一个接收者（sender, receiver)。在第5行中，利用闭包定义了一个新线程，并利用move关键字将sender的所有权移动到闭包内部。在第7行中，调用发送者的send方法，将数据及其所有权传入通道。在第9行中，接收者receiver调用recv方法从通道中接收数据，程序运行结果如图2.14所示。

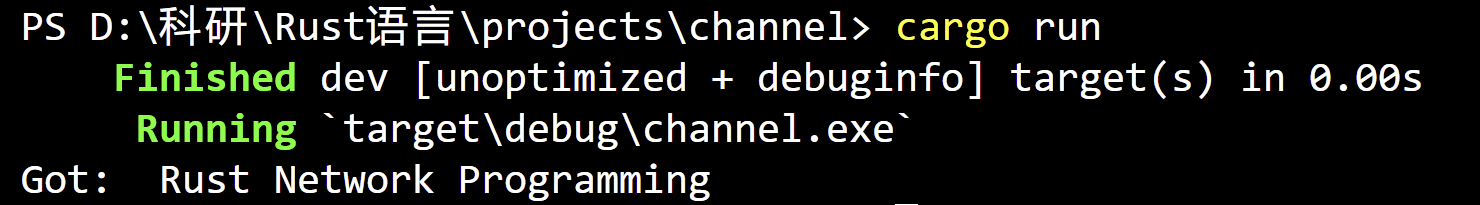


图2.14 程序运行结果图

在通道的接收端有两种方法可以接收通道中的数据，一个是recv，另一个是try\_recv。在本例中，我们利用recv成功接收到了通道数据，但这个方法会阻塞线程执行直到从信道中接收一个值，即此时在接收到消息前主线程无法处理其他的工作。try\_recv方法和rec的用法相同，但不会阻塞线程执行。当线程还有其它工作要做时，try\_recv方法就非常有用了，可以编写一个循环来定期调用 try\_recv ，如果有消息可用，则处理一条消息，否则可以先做一段时间其他工作，之后再来检查通道中是否有消息到来。

通道是目前常用的线程间共享数据的方式，但通道属于单所有权系统，一旦将值的所有权转移至通道，那么除在通道内接收该值外，其他任何地方都不能再使用它。共享状态并发机制可以使不同线程在不转移所有权的情况下共享数据。

### 共享内存空间

共享内存是指多个线程可以同时访问同一位置的内存。当然多个线程同时访问同一块内存（特别是在同一时期既有写操作又有读操作）会带来内存数据不确定性等风险，给内存安全提出了极大挑战。Rust提供了互斥器（Mutex）并发原语进行共享内存，保证了任意时刻只有一个线程访问互斥器锁定的数据，从而保证了内存数据的确定性。

互斥器规定要访问互斥器中的数据，线程必须首先要获得互斥器的锁(lock)，lock是一种数据结构，是Mutex的一部分，可以跟踪谁对数据拥有独占访问权。如果线程希望访问互斥器中的数据，就必须等待已经获得互斥锁的线程释放锁，这就保证了同一时间只有一个线程访问内存数据。因此，Mutex通常被描述为通过锁定系统来保护它所持有的数据。Lock的返回值为MetuxGuard的智能指针，指向数据所在的内存地址，然后用“\*”取出数据并进行修改。在lock的作用域结束后，Rust会自动释放锁，因此我们不用手动释放锁，也不用承担忘记释放锁而对其他线程造成的风险。下例展示如何在程序中使用互斥器。

1. use std::sync::Mutex;

2. fn main(){

3. let x=String::from("hello");

4. let m=Mutex::new(x);//创建一个互斥器m，并取得x的所有权，m是只可以互斥访问的变量

5. {

6. let mut num=m.lock().unwrap(); //num指向x所在内存，并锁定其不能再被其他线程使用

7. \*num=String::from("hi");

8. } //大括号内为lock的作用域

9. println!("{:?}",m);

10. }

在上例中的第6行调用lock()方法，锁定范围是lock所在的大括号内的代码，这个调用会阻塞当前线程直至获得锁为止，若其他线程已经拥有锁，则这个线程会panic!， 因此调用了unwrap()以应对这种失败情况。

下面我们再来看一个利用Mutex在线程间共享内存的例子。该例子启动了两个线程，依次对同一个计数器值进行加1操作，例子会出现编译错误，我们先来看代码：

1. use std::thread;

2. use std::sync::Mutex;

3. fn main(){

4. let count=Mutex::new(0);

5. let mut v=vec![];

6. let handle1=thread::spawn(move||{

7. let mut num=count.lock().unwrap();

8. \*num+=1;

9. });

10. v.push(handle1);

11. let handle2=thread::spawn(move||{

12. let mut num=count.lock().unwrap();

13. \*num+=1;

14. });

15. v.push(handle2);

16. for i in v{

17. i.join().unwrap();

18. }

19. println!("Result:{}",\*count.lock().unwrap());

20. }

在本例中，首先利用Mutex创建只可以互斥访问的count变量作为计数器，然后利用Thread::spawn创建两个线程，其中每一个线程都会调用lock来锁定互斥器count并修改其值，当一个线程结束后会自动释放锁，下一个线程就可以继续锁定互斥器count并执行相同操作，之后对两个子线程进行join以保证它们线程都执行完毕，最后打印互斥器count中的值。然而本程序显示编译错误。Rust编译器提示，不能将count锁的所有权移动到多个线程当中去，该如何修复这个错误呢？

Rust的Arc（Atomic Reference Counter）类型能够安全的用于并发环境。Arc<T>是一种线程安全的引用计数指针类型，提供在堆中分配的T类型值的共享权。Arc变量通过调用clone方法生成源Arc指针的拷贝，指向堆上同一地址空间，同时增加引用计数。Arc会追踪这个指针的所有拷贝，直到最后一份拷贝离开作用域时，将会安全释放堆上的空间。利用Arc将上例修改如下。

1. use std::thread;

2. use std::sync::{Mutex, Arc};

3. fn main(){

4. let count=Arc::new(Mutex::new(0)); //创建了一个互斥器并用一个Arc进行包装

5. let mut v=vec![];

6. let count1=Arc::clone(&count); //对count进行克隆，克隆了第一份引用

7. let handle1=thread::spawn(move||{

8. let mut num=count1.lock().unwrap(); //将count的克隆值引入第一个线程锁定数据

9. \*num+=1;

10. });

11. v.push(handle1);

12. let count2=Arc::clone(&count); //对count进行克隆，克隆了第二份引用

13. let handle2=thread::spawn(move||{

14. let mut num=count2.lock().unwrap(); //将count的克隆值引入第二个线程锁定数据

15. \*num+=1;

16. });

17. v.push(handle2);

18. for i in v{

19. i.join().unwrap();

20. } //保证两个子线程运行结束后，才能结束主线程

21. println!("Result:{}",\*count.lock().unwrap());

22. }

这次程序运行成功了，输出结果为2。究其原因在于，在这个例子中总共有3个指针指向互斥器，其中一个通过创建Arc得到，另外两个通过克隆得到，这些操作实现了同一变量的多所有权，再将拥有数据所有权的不同借用移动到不同的线程中去，自然不会出现一个所有权转移到多个线程的错误。只要三个指针中有任何一个未销毁，Rust就不会销毁互斥器释放内存。因此使用Arc包装Mutex是在多线程之间共享所有权的正确做法。

从上述例子和概念的讲解中我们了解到，在以共享内存的方式进行数据共享时，Arc和Mutex是必不可少的工具，Arc是一种智能指针，它能够让你在多线程之间安全的共享同一个数据；而Mutex是一种包装器，它使得某个数据在多线程之间被安全的修改。

## 测试

在软件开发过程中，调试和测试是至关重要的环节。它们能够帮助开发者发现和修复代码中的问题，确保软件的可靠性和稳定性。Rust作为一门现代化的系统级编程语言，为调试和测试提供了丰富而强大的工具和方法。

调试是一种通过定位和纠正错误来改进代码质量的过程。Rust为开发者提供了多种调试工具，如断言和打印调试信息。断言是一种快速而有效的方法，通过在代码中插入断言条件，当条件失败时，程序会立即停止并显示有关错误的详细信息，使得开发者能够快速定位问题所在。同时，Rust还支持打印调试信息，通过使用宏来输出变量的值和其他相关信息，从而加深对程序执行过程的理解，有助于发现隐藏的错误。

测试是验证代码是否按照预期工作的过程。Rust内置了一个强大的测试框架，使得编写和运行测试变得简单而直观。开发者可以编写测试函数和宏来定义各种测试场景，并使用断言来验证代码的正确性。Rust的测试框架还支持并发测试和基准测试，让开发者能够更全面地评估代码性能和正确性。

下面我们将深入探索Rust调试和测试的方法，探讨如何使用断言和打印调试信息来迅速定位和解决问题，以及如何编写有效的测试来保证代码的质量。无论您是初学者还是有经验的开发者，通过学习和应用这些Rust调试和测试技术，您将能够更加高效地开发高质量、健壮的软件。

### 调试

调试是软件开发中至关重要的一环，它可以帮助我们及时找到错误并改正，无论是在Rust还是其他编程语言中都是如此。拥有好的调试技术和手段对写出优良的代码起着至关重要的作用，可以帮助我们提升程序的稳定性和可靠性。下面我们将学习一些在Rust中进行调试的常用技术和工具，帮助我们更好地定位和解决代码问题。

1. **使用断言（assertions）**

断言是一种有效的调试技巧，用于验证代码的假设和预期行为。在Rust中，你可以使用 assert! 宏进行断言。当断言条件不满足时，程序会立即中止并输出错误信息。以下是一个示例：

1. fn divide(a: i32, b: i32) -> i32 {

2. assert!(b != 0, "Divisor must not be zero!");

3. a / b

4. }

在这个例子中，如果 b 的值为零，程序将立即中止并输出错误信息。

1. **使用打印语句（print statements）**

打印语句是另一种常用的调试技巧，它允许你在代码中输出变量的值或其他调试信息。在Rust中，你可以使用 println! 宏进行打印输出。以下是一个示例：

1. fn main() {

2. let name = "Alice";

3. let age = 25;

4.

5. println!("Name: {}", name);

6. println!("Age: {}", age);

7. }

通过打印 name 和 age 的值，我们可以观察它们的实际取值。

1. **使用断点调试器（debugger）**

Rust提供了与多个调试器集成的能力，包括GDB和LLDB。调试器允许你逐行调试代码，观察变量状态和调用栈，以及设置断点和单步执行等。以下是一个简单的调试器示例过程：

在Cargo.toml中启用调试信息：

1. [profile.dev]

2. debug = true

3. [profile.release]

4. debug = true

构建项目：使用 cargo build 命令构建项目。

1. $ cargo build

使用调试器：例如，使用GDB调试器可以执行以下命令：

1. $ gdb target/debug/hello\_world

运行程序：在调试器中使用 run 命令来运行程序。

设置断点：使用 break 命令设置断点。

单步执行：使用 next 命令进行单步执行。

1. **使用Rust插件**

**Rust**：这是官方提供的Rust插件，提供了Rust语言的语法高亮、代码片段、智能补全等功能。你可以通过在VsCode插件商店中搜索并安装"Rust"插件来使用它。

**CodeLLDB**：这个插件允许与LLDB交互，用于在Visual Studio Code中进行Rust代码的调试。它提供了断点设置、单步调试、查看变量值等功能。你可以通过在VsCode插件商店中搜索并安装"CodeLLDB"插件来使用它。使用示例：1、安装"CodeLLDB"插件，并确保你已经安装了LLDB调试器。在你的Rust项目中打开要调试的源文件。2、在代码行号旁边单击设置断点。3、打开调试视图（Ctrl+Shift+D）。4、点击调试视图顶部的"Run and Debug"按钮。5、在弹出的配置选项中选择"LLDB"配置。6、点击"Start Debugging"按钮开始调试。7、在调试过程中，你可以使用调试工具栏的按钮执行调试操作，如单步执行、继续执行等

**Native Debug**：这个插件支持GDB调试器，用于在Visual Studio Code中进行Rust代码的调试。它提供了类似于CodeLLDB的功能，但使用的是GDB作为调试器。你可以通过在VsCode插件商店中搜索并安装"Native Debug"插件来使用它。使用示例：1、安装"Native Debug"插件，并确保你已经安装了GDB调试器。2、在你的Rust项目中打开要调试的源文件。3、在代码行号旁边单击设置断点。4、打开调试视图（Ctrl+Shift+D）。4、点击调试视图顶部的"Run and Debug"按钮。5、在弹出的配置选项中选择"GDB"配置。6、点击"Start Debugging"按钮开始调试。7、在调试过程中，你可以使用调试工具栏的按钮执行调试操作，如单步执行、继续执行等。

1. **使用日志记录**

日志记录是一种强大的调试技术，通过在代码中添加日志输出，帮助你跟踪程序的执行并找到问题所在。在Rust中，你可以使用 log 和 env\_logger 库进行日志记录。以下是一个示例：

1. use log::{info, error};

2. use env\_logger;

3. fn main() {

4. env\_logger::init();

5. info!("This is an info log message.");

6. error!("This is an error log message.");

7. }

在这个例子中，日志记录消息将输出到控制台。我们可以根据需要来设置日志级别和输出格式。

在调试Rust代码时，断言和打印语句是最简单直接且常用的调试技巧；调试器和Rust插件提供更高级的调试功能，可以逐行跟踪程序执行过程和观察变量状态；日志记录是一种高级调试技术，可以通过在代码中添加日志输出来深入分析程序的执行过程。根据具体情况，我们可以按照需求选择合适的调试方法和工具，提高Rust代码调试的效率和性能。

### 测试

在Rust中，有多种测试方法可用于验证代码的正确性、性能和安全性。其中单元测试通过针对最小代码单元进行验证，确保其在各种输入情况下产生正确的输出；集成测试用于测试系统各个组件之间的交互和整体功能；属性测试通过使用随机输入和断言来验证代码的属性和不变式；性能测试可以衡量代码的效率和性能，帮助找到潜在的性能问题；属性验证可以进行形式化验证，验证代码的行为和规范。通过综合运用这些测试方法，开发者可以确保代码质量、发现潜在问题，提高软件的健壮性。下面我们将更详细地介绍每种测试方法及其使用细节。

1. **单元测试（Unit Testing）：**

单元测试的目标是测试最小代码单元，如函数、方法或模块。它旨在确保这些单元在给定输入下产生正确的输出，并处理边缘情况和异常情况。

在Rust中，如果要对代码块进行单元测试，需要在该代码块的同级目录下创建一个名为`tests`的文件夹。Rust的约定是将测试代码与实际代码分开，以确保测试代码只在执行`cargo test`命令时编译和执行。在`tests`文件夹中编写测试代码，使用`#[cfg(test)]`属性指定测试代码模块。使用`#[test]`宏将测试函数标记为测试用例，并编写断言来验证函数的行为和结果。例如，使用`assert\_eq!`宏来比较实际结果和期望结果。运行单元测试使用命令`cargo test`。它将自动发现并执行`tests`文件夹中的测试代码，并提供更详细的输出和统计信息。

接下来让我们看一个例子来进一步了解单元测试，下面是一个字符串反转函数reverse\_string的单元测试示例代码：

1. // src/lib.rs

2. pub fn reverse\_string(s: &str) -> String {

3. let reversed: String = s.chars().rev().collect();

4. reversed

5. }//此函数的代码要放在src文件夹下的lib.rs库文件中

6. // tests/reverse\_string\_tests.rs

7. #[cfg(test)]

8. mod tests {

9. use super::\*;

10. #[test]

11. fn test\_reverse\_string() {

12. assert\_eq!(reverse\_string("hello"), "olleh"); // 测试常规情况

13. assert\_eq!(reverse\_string(""), "");// 测试空字符串

14. assert\_eq!(reverse\_string("hello, 世界!"), "!界世 ,olleh");//测试包含特殊字符的字符串

15. }

16. }//tests模块的代码放在tests/reverse\_string\_tests.rs文件中

在上面的代码中，我们首先在src/lib.rs文件中定义了一个reverse\_string函数，它接受一个字符串参数并返回反转后的字符串。接下来，在同级目录下创建了一个名为tests的文件夹，并在其中创建了名为reverse\_string\_tests.rs的文件，用于编写测试代码。

在tests文件夹中的测试代码模块上使用了#[cfg(test)]属性，以确保该模块只在运行测试时才会被编译和执行。接着，使用#[test]宏标记了一个名为test\_reverse\_string的测试函数，用来测试reverse\_string函数的行为和结果。在该测试函数中，我们使用了assert\_eq!宏来比较实际结果和期望结果。最后，使用命令cargo test来运行单元测试。它将自动发现并执行tests文件夹中的测试代码，并输出测试结果和统计信息。

1. **集成测试（Integration Testing）：**

集成测试旨在对整个程序或一组模块进行测试，模拟实际运行环境并测试它们之间的交互。要在Rust中进行集成测试，需要在项目的根目录下创建一个名为`tests`的文件夹，其中包含各种集成测试文件，每个文件包含一个或多个测试场景的函数，用于对整个系统的各个部分进行测试。集成测试通常还需要设置一些测试环境和共享状态，可以使用`std::process::Command`来启动和控制外部进程。使用`cargo test`命令来运行集成测试，它会自动发现并执行`tests`文件夹中的测试代码，并提供详细的输出和统计信息。

在 Rust 中，可以使用内置的测试框架和相关的宏来编写集成测试。下面创建和运行Rust集成测试的步骤举例。

第一步创建测试文件：在项目的根目录下创建一个名为 tests 的目录，并在其中创建测试文件 integration\_tests.rs。

第二步导入必要的模块：在 integration\_tests.rs 文件中，导入需要用到的模块和函数。例如：

1. use crate::module1::function1;

2. use crate::module2::function2;

第三步编写测试函数：在 integration\_tests.rs 文件中，编写测试函数，并使用测试框架提供的断言宏来验证预期结果，如下所示。

1. #[test]

2. fn test\_integration() {

3. let result1 = function1();// 执行模块1中的函数

4. let result2 = function2(result1);// 执行模块2中的函数

5.   assert\_eq!(result2, expected\_result); // 验证结果是否符合预期

6. }

这个示例展示了一个集成测试函数 test\_integration，它调用了两个不同模块中的函数，并使用断言宏 assert\_eq! 验证最终结果是否符合预期。

第四步运行集成测试：在项目的根目录下，使用cargo test 测试命令运行集成测试。此命令将会执行项目中的所有测试，包括单元测试和集成测试。也可以通过--test参数来指定只运行集成测试：cargo test --test integration\_tests。这样，Rust 将会编译并运行 integration\_tests.rs 文件中的集成测试函数。

通过以上的步骤就可以编写和运行 Rust 中的集成测试了。记住，在编写测试代码时，需要将测试方案考虑得尽可能全面，以便验证系统的整体功能和各模块之间的交互。

下面是一个简单的 Rust 集成测试的示例。这个示例将测试一个加法函数的功能是否符合预期。注意：在编写实际的代码时，请确保导入了要测试的模块，并且根据需要进行适当的修改。

1. mod tests {

2. use my\_crate::add;

3. #[test]

4. fn test\_addition\_num() {

5. let num1 = 16;

6. let num2 = 34;// 准备测试数据

7. let result = add(num1, num2); // 调用被测试的函数进行计算

8. assert\_eq!(result, 50);// 验证结果是否符合预期

9. }

10. }

测试代码执行完毕时，如果没有遇到断言失败的情况，那么说明被测试的函数通过了所有的验证，功能符合预期。集成测试对于验证系统的整体行为非常有帮助，并可以发现潜在的问题和错误。

1. **属性测试（Property Testing）：**

属性测试是一种基于属性或规范描述的自动化测试方法，通过在大量输入上运行代码来验证其属性是否始终成立。属性测试可以帮助发现边缘情况、潜在的错误行为和不变式，并验证代码的正确性。

Rust有一些属性测试库可供选择，如quickcheck和proptest。这些库通过生成随机输入来自动测试，并验证代码的特定属性。编写属性测试时，需要定义输入的生成器和需要验证的属性。通常，属性是基于函数的前提条件、后置条件或期望行为。属性测试库会自动运行多个随机生成的输入，并检查是否满足属性。如果找到任何反例，库将生成输入、回溯和帮助找出失败的案例。

下面我们以一个简单的Rust属性测试代码为例，介绍如何使用属性来标记测试函数，并通过断言来验证代码逻辑。

1. #[cfg(test)] // 只在测试环境下编译此代码

2. mod tests {

3. #[test] // 测试函数

4. #[should\_panic(expected = "assertion failed")] // 用于测试断言失败是否引发 panic

5. fn test\_assertion\_addnum() {

6. assert\_eq!(15 + 15, 40); // 断言 15 + 15 等于 40，将会失败

7. }

8. }

上述代码包含了一个测试模块 tests，其中定义了一个测试函数 test\_assertion\_addnum。该函数使用了 Rust 的断言宏 assert\_eq! 来验证15 + 15 是否等于 40 。由于断言失败，因此在测试运行时会引发 panic。该代码只会在测试环境下进行编译。

综上，无论是单元测试、集成测试，还是属性测试，选择合适的工具和库非常重要，程序员需要根据具体需求使用不同的测试方法和工具。同时，编写测试代码、准备适当的测试数据和环境，以及编写清晰的测试断言和属性描述都是编写高质量测试的重要因素。当然，Rust还提供有其他多种灵活的测试方法，来帮助开发者构建高质量的软件，这里我们不再逐一介绍。

## 小结

第二章的学习为后续Rust网络应用编程的学习打下了基础，因为本章讲述的所有权、生存周期等概念，或是Rust独有，或是在Rust中起到了至关重要的作用，促成了Rust内存安全，运行高效的特点。不止于此，泛型和特型简化并严谨了代码，错误处理一定程度上增加了程序的健壮性，可以对可预见的错误进行处理。

通过本章的学习，我们将了解Rust的核心概念和功能。所有权、借用、生存期、函数、泛型、特性、宏、错误处理和并发处理是Rust编程的基石。掌握这些概念和技术，将使我们能够编写高效、安全和可维护的Rust代码。

# 数据序列化、反序列化和解析

本章要点：理解Serde的序列化和反序列化库。掌握Serde库实现序列化和反序列化的方法，掌握文本数据和二进制数据的解析方法。

## 引言

在使用Rust的套接字编写网络应用程序时，可以使用TCP和UDP等传输层协议传输消息。由于TCP和UDP协议主要处理字节数据，因此需要将要发送的字符串先转换为字节的格式（在网络传输时，该格式为字节流）。这种将一段数据转换为可以用于存储或者传输的格式的过程，被称为序列化，反之则是反序列化。也就是说，序列化或反序列化就是将一种数据结构转换为另一种数据结构。任何网络应用程序都必须对已收到或即将发送的数据进行序列化或反序列化处理。当然，也不能进行任意的转换，比如用户定义的类型，或简单的集合类型。而Rust生态系统提供一个特殊的crate架构，可以实现多种格式下数据转换，即数据的序列化和反序列化。

## 使用Serde库实现序列化和反序列化

Serde库是Rust语言用来实现序列化和反序列化的解决方案，它体现了定义与实现分离的设计思想。Serde库是建立在Rust的trait系统之上的，提供了序列化和反序列化数据的标准方法，具有很好的性能。它支持多种数据结构（包括标准数据类型和用户自定义数据类型）与多种数据格式（包括JSON、YAML、TOML和CSV等）的相互转换，即序列化和反序列化。

### 序列化和反序列化

在网络应用编程中，序列化就是将内存中的数据结构或对象转换成适于网络传输或是本地保存的二进制序列的过程；而反序列化则是序列化的逆过程，即将在序列化过程中所生成的二进制序列转换成数据结构或者对象的过程。

Serde库采用过程宏机制所提供的两个自定义派生类，Serialize和Deserialize，来实现序列化和反序列化操作。Serialize和Deserialize可以由Serde所支持的标准数据类型或用户自定义数据类型实现。下面的示例可以体现其工作原理。

使用Cargo包管理器创建空项目：$ cargo new --bin ch3-1-serde-basic



修改Cargo.toml文件，添加依赖项：

1. // chapter3/ch3-1-serde-basic/src/main.rs
2. [package]
3. Name = “ch3-1-serde-basic”
4. version = “0.1.0”
5. [dependencies]
6. serde={version = "1",features = ["derive"]}
7. serde\_derive = “1.0”
8. serde\_json = “1.0”
9. serde\_yaml = “0.9.3”

serde crate包是Serde库的核心。serde\_derive crate包提供必要的工具，使用过程宏来派生Serialize和Deserialize。但是serde只提供序列化和反序列化的框架，具体的操作还需要依赖具体的包，如serde\_json和serde\_yaml等。下面的示例介绍了这两个包的具体功能：

1. // chapter3/ch3-1-serde-basic/src/main.rs
2. #[macro\_use]
3. extern crate serde\_derive;
4. extern crate serde;
5. extern crate serde\_json;
6. extern crate serde\_yaml;
7. // We will serialize and deserialize instances of this struct
8. #[derive(Serialize, Deserialize, Debug)]
9. struct ServerConfig {
10. workers: u64,
11. ignore: bool,
12. auth\_server: Option<String>
13. }
14. fn main() {
15. let config = ServerConfig {
16. workers: 100,
17. ignore: false,
18. auth\_server: Some("auth.server.io".to\_string())
19. };
20. {
21. println!("To and from YAML");
22. let serialized = serde\_yaml::to\_string(&config).unwrap();
23. println!("{}", serialized);
24. let deserialized: ServerConfig =
25. serde\_yaml::from\_str(&serialized).unwrap();
26. println!("{:?}", deserialized);
27. }
28. println!("\n\n");
29. {
30. println!("To and from JSON");
31. let serialized = serde\_json::to\_string(&config).unwrap();
32. println!("{}", serialized);
33. let deserialized: ServerConfig =
34. serde\_json::from\_str(&serialized).unwrap();
35. println!("{:?}", deserialized);
36. }
37. }

首先，使用serde\_derive crate包导出宏，用macro\_use声明标记这个宏；然后，将所有的依赖项声明为extern crates。之后再定义自定义数据类型。在这种情况下，需要对服务器一系列不同类型参数进行配置。结构体ServerConfig中，参数auth\_server是可选的，其他参数可以Serde的trait特质获得，此外编译器提供的Debug trait特质，这些都便于之后反序列化后显示。在主函数中，实例化后调用serde\_yaml::to\_string可以将yaml格式序列化为字符串；反之，调用serde\_yaml::from\_str则可以将字符串反序列化为yaml格式。

上述示例的运行结果如下：

![电脑的屏幕截图

描述已自动生成

下面我们用一个更复杂的示例来演示在网络传输过程中如何使用Serde库实现序列化和反序列化。该示例包括一个TCP服务器和一个客户端。客户端向服务器发送一个三维空间坐标，服务器则收到坐标信息后计算并返回该坐标与原点的距离。

建立Cargo项目：**$ cargo new --bin ch3-2-serde-server**



配置代码如下：

1. $ cat Cargo.toml
2. [package]
3. name = "ch3-2-serde-server"
4. version = "0.1.0"
5. [dependencies]
6. serde = "1.0"
7. serde\_derive = "1.0"
8. serde\_json = "1.0"

为了方便演示和简化代码，本例中在同一个文件内实现服务器和客户端。运行时，程序首先接收一个标志，该标志用作区分程序作为服务器还是作为客户端运行。作为服务器运行，程序将所有本地接口绑定到一个指定端口的，并监听传入的连接。作为客户端运行时，程序将连接到指定端口上的服务器，并等待控制台上的用户输入。客户端的用户输入是一个用逗号分隔的三个整数列表，用来表示空间坐标的X、Y、Z轴坐标。这个用户输入被传递到一个表示三维坐标的结构体内，先使用Serde将其序列化为字节流，再将该字节流发送到服务器。服务器则将收到的字节流反序列化为相同类型的结构体，然后进行距离计算，并将结果发送回客户端，客户端接收并显示该结果。其代码如下：

1. #[macro\_use]
2. extern crate serde\_derive;
3. extern crate serde;
4. extern crate serde\_json;
5. use std::io::{stdin, BufRead, BufReader, Error, Write};
6. use std::net::{TcpListener, TcpStream};
7. use std::{env, str, thread};
8. use std::num::ParseIntError;
9. #[derive(Serialize, Deserialize, Debug)]
10. struct Point3D {
11. x: u32,
12. y: u32,
13. z: u32,
14. }
15. // Like previous examples of vanilla TCP servers, this function handles
16. // a single client.
17. fn handle\_client(stream: TcpStream) -> Result<(), Error> {
18. println!("Incoming connection from: {}", stream.peer\_addr()?);
19. let mut data = Vec::new();
20. let mut stream = BufReader::new(stream);
21. loop {
22. data.clear();
23. let bytes\_read = stream.read\_until(b'\n', &mut data)?;
24. if bytes\_read == 0 {
25. return Ok(());
26. }
27. let input: Point3D = serde\_json::from\_slice(&data)?;
28. let value = input.x.pow(2) + input.y.pow(2) + input.z.pow(2);
29. write!(stream.get\_mut(), "{}", f64::from(value).sqrt())?;
30. write!(stream.get\_mut(), "{}", "\n")?;
31. }
32. }
33. fn main() {
34. let args: Vec<\_> = env::args().collect();
35. if args.len() != 2 {
36. eprintln!("Please provide --client or --server as argument");
37. std::process::exit(1);
38. }
39. // The server case
40. if args[1] == "--server" {
41. let listener = TcpListener::bind("0.0.0.0:8888").expect("Could not bind");
42. for stream in listener.incoming() {
43. match stream {
44. Err(e) => eprintln!("failed: {}", e),
45. Ok(stream) => {
46. thread::spawn(move || {
47. handle\_client(stream).unwrap\_or\_else(|error|eprintln!("{:?}", error));
48. });
49. }
50. }
51. }
52. }
53. // Client case begins here
54. else if args[1] == "--client" {
55. let mut stream = TcpStream::connect("127.0.0.1:8888").expect("Could not connect to server");
56. println!("Please provide a 3D point as three comma separated integers");
57. loop {
58. let mut input = String::new();
59. stdin().read\_line(&mut input).expect("Failed to read from stdin");
60. let parts: Vec<&str> = input.trim().split(',').collect();
61. if parts.len() != 3 {
62. eprintln!("Invalid input format. Please provide three comma-separated integers.");
63. continue;
64. }
65. let point = match parse\_point(parts) {
66. Ok(point) => point,
67. Err(e) => {
68. eprintln!("Error parsing input: {}", e);
69. continue;
70. }
71. };
72. stream.write\_all(serde\_json::to\_string(&point).unwrap().as\_bytes())
73. .expect("Failed to write to server");
74. stream.write\_all(b"\n").expect("Failed to write to server");
75. let mut reader = BufReader::new(&stream);
76. let mut buffer = String::new();
77. reader.read\_line(&mut buffer).expect("Could not read into buffer");
78. print!("Response from server: {}", buffer);
79. }
80. }
81. }
82. fn parse\_point(parts: Vec<&str>) -> Result<Point3D, ParseIntError> {
83. let x = parts[0].trim().parse()?;
84. let y = parts[1].trim().parse()?;
85. let z = parts[2].trim().parse()?;
86. Ok(Point3D { x, y, z })
87. }

示例代码中首先设置Serde，再将三维坐标定义为三个元素的结构体。在主函数中，通过命令行输入标志，根据输入的标志内容选择执行客户端或服务器。当然，不管选择客户端还是服务器，都是通过发送一个换行符表示传输结束的。客户端从stdin中读取一行，并在循环中创建三维坐标结构体的实例。在客户端或服务器情况下，都将字节流封装在一个BufReader中以便后续处理。

服务器的会话示例如下：

文本

描述已自动生成

对于客户端而言，先读取输入，再对所输入的内容序列化，并将其发送到服务器，然后等待响应，获得响应后，将响应结果打印输出。

客户端的会话示例如下：

文本

描述已自动生成

### 自定义序列化和反序列化

借助宏，Serde库提供了针对标准数据类型以及很多复合数据类型的内置序列化和反序列化功能。然而，在某些情况下，比如一些更复杂的数据类型，就需要手动实现这些操作。这就涉及到Serde的高级功能——自定义序列化和反序列化。这些高级功能还允许重命名输出中的字段，主要用在处理新协议等网络方面。

下面的示例演示了自定义序列化和反序列化的实现过程。

新建项目：**$ cargo new --bin ch3-3-serde-custom**

配置文件如下：

1. [package]
2. name = "ch3-3-serde-custom"
3. version = "0.1.0"
4. [dependencies]
5. serde = "1.0"
6. serde\_derive = "1.0"
7. serde\_json = "1.0"
8. serde\_test = "1.0"

自定义数据结构如下：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. // We will implement custom serialization and deserialization
3. // for this struct
4. #[derive(Debug, PartialEq)]
5. struct KubeConfig {
6. port: u8,
7. healthz\_port: u8,
8. max\_pods: u8,
9. }

派生出Debug和PartialEq供Serde库的内部使用。在实际应用中，也可能需要手动实现这些操作。自定义结构体Kubeconfig实现序列化的trait特质，定义如下：

1. pub trait Serialize {
2. fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
3. where S: Serializer;
4. }

简单来说，序列化自定义结构体的基本工作流就是先序列化结构体名，然后是按顺序序列化每个元素，最后则是序列化结束。Serde有内置方法来处理的基本数据类型的序列化，因此，我们主要看看如何序列化自定义的结构体：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. // Implementing Serialize for our custom struct defines
3. // how instances of that struct should be serialized.
4. // In essence, serialization of an object is equal to
5. // sum of the serializations of it's components
6. impl Serialize for KubeConfig {
7. fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
8. where S: Serializer
9. {
10. let mut state = serializer.serialize\_struct("KubeConfig", 3)?;
11. state.serialize\_field("port", &self.port)?;
12. state.serialize\_field("healthz\_port", &self.healthz\_port)?;
13. state.serialize\_field("max\_pods", &self.max\_pods)?;
14. state.end()
15. }
16. }

结构体的序列化始终以调用serialize\_struct开始，将结构体名称和字段数作为参数（其他类型也有类似的命名方法）。然后，按照每个字段出现的顺序依次进行序列化，同时传递一个将在json中使用的关键名。一旦完成上面的步骤，就调用特殊的end方法作为完成信号。

上面介绍的序列化的过程，而实现反序列化要更复杂一些，需要使用一些代码模板。其trait特质定义如下：

1. pub trait Deserialize<'de>: Sized {
2. fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
3. where D: Deserializer<'de>;
4. }

要对某个类型实现这个操作，需要实现访问者模式。Serde定义了一个特殊的访问者Visitor trait特质，如下例所示。注意，定义中实际上是有针对所有内置类型的visis\_\*方法的，只是这里没有显示。为了突出重点，示例中用符号...表示省略了其他更多的方法。

1. pub trait Visitor<'de>: Sized {
2. type Value;
3. fn expecting(&self, formatter: &mut Formatter) -> Result;
4. fn visit\_bool<E>(self, v: bool) -> Result<Self::Value, E>
5. where
6. E: Error,
7. { }
8. ...
9. }

反序列化器在内部使用此特性的实现来构造结果类型。示例如下：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. // Implementing Deserialize for our struct defines how
3. // an instance of the struct should be created from an
4. // input stream of bytes
5. impl<'de> Deserialize<'de> for KubeConfig {
6. fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
7. where D: Deserializer<'de>
8. {
9. enum Field { Port, HealthzPort, MaxPods };
10. impl<'de> Deserialize<'de> for Field {
11. fn deserialize<D>(deserializer: D) ->
12. Result<Field,
13. D::Error>
14. where D: Deserializer<'de>
15. {
16. struct FieldVisitor;
17. impl<'de> Visitor<'de> for FieldVisitor {
18. type Value = Field;
19. fn expecting(&self, formatter: &mut fmt::Formatter)
20. -> fmt::Result {
21. formatter.write\_str("`port` or `healthz\_port`
22. or `max\_pods`")
23. }
24. fn visit\_str<E>(self, value: &str) ->
25. Result<Field,
26. E>
27. where E: de::Error
28. {
29. match value {
30. "port" => Ok(Field::Port),
31. "healthz\_port" =>
32. Ok(Field::HealthzPort),
33. "max\_pods" => Ok(Field::MaxPods),
34. \_ => Err(de::Error::unknown\_field(value,
35. FIELDS)),
36. }
37. }
38. }
39. deserializer.deserialize\_identifier(FieldVisitor)
40. }
41. }
42. }

现在，如果将反序列化过程视作一个映射的话，其输入就是JSON格式数据。因此，我们只需要实现来自Visitor特质的visit\_map。如果反序列化程序收到任何非JSON数据，它将在调用来自该trait特质的其他函数时出错。之前的大多数实现都是模板文件。它可以归结为以下步骤：第一步是为各字段实现Visitor，以及实现visit\_str（因为我们的所有字段都是字符串）。这就是对各字段反序列化。第二步是实现整个结构体的Visitor，并实现visit\_map。最后一步是，调用deserializer.deserialize\_struct，并传递结构体名、字段列表和Visitor完成整个反序列化过程。实现代码如下：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. impl<'de> Deserialize<'de> for KubeConfig {
3. fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
4. where D: Deserializer<'de>
5. {
6. struct KubeConfigVisitor;
7. impl<'de> Visitor<'de> for KubeConfigVisitor {
8. type Value = KubeConfig;
9. fn expecting(&self, formatter: &mut fmt::Formatter) ->
10. fmt::Result {
11. formatter.write\_str("struct KubeConfig")
12. }
13. fn visit\_map<V>(self, mut map: V) ->
14. Result<KubeConfig,
15. V::Error>
16. where V: MapAccess<'de>
17. {
18. let mut port = None;
19. let mut hport = None;
20. let mut max = None;
21. while let Some(key) = map.next\_key()? {
22. match key {
23. Field::Port => {
24. if port.is\_some() {
25. return
26. Err(de::Error::duplicate\_field("port"));
27. }
28. port = Some(map.next\_value()?);
29. }
30. Field::HealthzPort => {
31. if hport.is\_some() {
32. return
33. Err(de::Error::duplicate\_field
34. ("healthz\_port"));
35. }
36. hport = Some(map.next\_value()?);
37. }
38. Field::MaxPods => {
39. if max.is\_some() {
40. return
41. Err(de::Error::duplicate\_field
42. ("max\_pods"));
43. }
44. max = Some(map.next\_value()?);
45. }
46. }
47. }
48. let port = port.ok\_or\_else(||
49. de::Error::missing\_field("port"))?;
50. let hport = hport.ok\_or\_else(||
51. de::Error::missing\_field("healthz\_port"))?;
52. let max = max.ok\_or\_else(||
53. de::Error::missing\_field("max\_pods"))?;
54. Ok(KubeConfig {port: port, healthz\_port: hport,
55. max\_pods: max})
56. }
57. }
58. const FIELDS: &'static [&'static str] = &["port",
59. "healthz\_port", "max\_pods"];
60. deserializer.deserialize\_struct("KubeConfig", FIELDS,
61. KubeConfigVisitor)
62. }
63. }

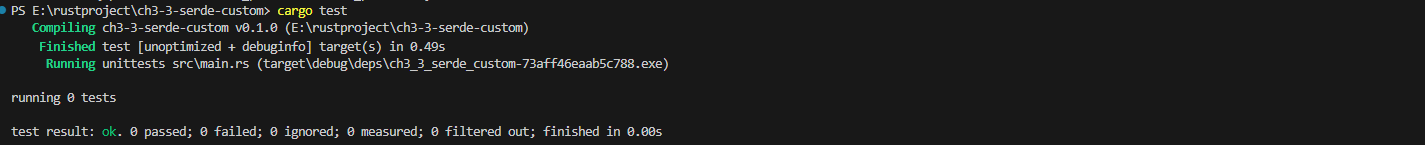
Serde库还提供了一个crate包，可以用于使用类似于令牌流的接口来统一测试自定义序列化器和反序列化器。使用时，需要先将serde\_test添加到配置文件Cargo.toml中，再在主文件中声明其为一个外部crate包。下面是反序列化程序的一个测试示例：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. #[test]
3. fn test\_ser\_de() {
4. let c = KubeConfig { port: 10, healthz\_port: 11, max\_pods: 12};
5. assert\_de\_tokens(&c, &[
6. Token::Struct { name: "KubeConfig", len: 3 },
7. Token::Str("port"),
8. Token::U8(10),
9. Token::Str("healthz\_port"),
10. Token::U8(11),
11. Token::Str("max\_pods"),
12. Token::U8(12),
13. Token::StructEnd,
14. ]);
15. }

其中，调用assert\_de\_tokens是用来检查指定令牌流是否反序列化到自定义结构体，从而测试我们的反序列化器。演示代码如下所示：

1. // chapter3/ch3-3-serde-custom/src/main.rs
2. fn main() {
3. let c = KubeConfig { port: 10, healthz\_port: 11, max\_pods: 12};
4. let serialized = serde\_json::to\_string(&c).unwrap();
5. println!("{:?}", serialized);
6. }

上述代码的运行和测试结果如下：





## 解析文本数据

数据解析是一个与反序列化密切相关的工作。对解析最常见的思考方法是从形式化语法规则来构造解析器。先用较小的规则解析输入较小的组件，再用组合规则将所有小规则组合起来，最终形成一个自下而上的解析器，这种形式化定义有限规则集的方法称为解析表达式语法(Parsing Expression Grammar，PEG)。这种方法确保了解析是确定的：如果解析成功，则只有一个有效的解析树。

在Rust生态系统中，有多种方法来实现PEG，各有优缺点。第一种方法是使用宏来定义一种特定领域的语言来进行解析。该方法通过新的宏系统可以很好地与编译器集成一起，进而生成快速的代码。然而，这种方法通常调试和维护更复杂。并且这种方法也不允许重载操作符，因此必须定义一个DSL，这又会增加学习和使用的难度。采用该方法的解析器是nom。第二种方法是使用trait特质系统。该方法有助于定义自定义操作符，且更容易调试和维护。采用该方法的解析器是pom和pest。

数据解析的使用场景主要是在网络应用编程环境下。在此环境下，有时处理原始字符串（或字节流）并解析信息比反序列化到复杂的数据结构更有用。解析数据通常应用于基于文本的协议场景下，比如HTTP。服务器可能通过套接字作为字节流接收原始请求，并解析它以提取信息。下面介绍Rust生态系统中的一些常见的解析技术。

nom是一个解析器组合子框架，这意味着它可以结合更小的解析器来构建更强大的解析器。这是一种自底而上的方法，通常从编写非常具体的解析器开始，它们从输入中解析具有完整和明确定义的信息。然后，该框架提供了将这些小解析器链接成一个完整的解析器的方法。这种方法与lex和yacc案例中的自上而下的方法相反，后者可以从定义语法开始，而前者可以处理两个字节流（二进制数据）或字符串。我们先演示解析简单的字符串的功能，这是一个HTTP GET或POST请求。

新建项目：**$ cargo new --bin nom-http**



配置文件如下：

1. $ cat Cargo.toml
2. [package]
3. name = "nom-http"
4. version = "0.1.0"
5. [dependencies.nom]
6. version = "3.2.1"
7. features = ["verbose-errors", "nightly"]

该Crate库还提供了一些用于调试的额外特性，这些特性在默认情况下是禁用的，使用时可以通过特性标志来打开。主文件代码如下：

1. // chapter4/nom-http/src/main.rs
2. #[macro\_use]
3. extern crate nom;
4. use std::str;
5. use nom::{ErrorKind, IResult};
6. #[derive(Debug)]
7. enum Method {
8. GET,
9. POST,
10. }
11. #[derive(Debug)]
12. struct Request {
13. method: Method,
14. url: String,
15. version: String,
16. }
17. // A parser that parses method out of a HTT request
18. named!(parse\_method<&[u8], Method>,
19. return\_error!(ErrorKind::Custom(12), alt!(map!(tag!("GET"), |\_|
20. Method::GET) | map!(tag!("POST"), |\_| Method::POST))));
21. // A parser that parses the request part
22. named!(parse\_request<&[u8], Request>, ws!(do\_parse!(
23. method: parse\_method >>
24. url: map\_res!(take\_until!(" "), str::from\_utf8) >>
25. tag!("HTTP/") >>
26. version: map\_res!(take\_until!("\r"), str::from\_utf8) >>
27. (Request { method: method, url: url.into(), version: version.into() })
28. )));
29. // Driver function for running the overall parser
30. fn run\_parser(input: &str) {
31. match parse\_request(input.as\_bytes()) {
32. IResult::Done(rest, value) => println!("Rest: {:?} Value: {:?}",
33. rest, value),
34. IResult::Error(err) => println!("{:?}", err),
35. IResult::Incomplete(needed) => println!("{:?}", needed)
36. }
37. }
38. fn main() {
39. let get = "GET /home/ HTTP/1.1\r\n";
40. run\_parser(get);
41. let post = "POST /update/ HTTP/1.1\r\n";
42. run\_parser(post);
43. let wrong = "WRONG /wrong/ HTTP/1.1\r\n";
44. run\_parser(wrong);
45. }

代码中，nom大量使用宏进行代码生成。这些宏中最重要的是named!，它接受一个函数签名，并基于该签名定义一个解析器。nom解析器返回一个IResult类型的实例；这是一个枚举类型，有三个变量，具体介绍如下：

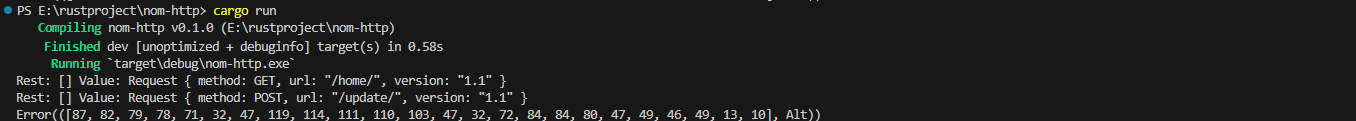
1. 变量Done(rest, value），表示当前解析器成功的情况。在这种情况下，value值是当前解析的值，rest值是要解析的剩余的输入值。
2. 变量Error(Err<E>)，表示解析过程中的错误。潜在的错误信息包含错误代码、错误位置等信息。在一个大型的解析树中，也可以包含指向更多错误的指针。
3. 变量Incomplete(needed)，表示由于某种原因解析不完整的情况。需要的是一个同样有两个枚举变量，分别表示不知道所需数据大小的情况以及所需数据的具体大小。

示例从HTTP的请求消息的格式定义和方法开始。为了突出重点，在示例中，只处理请求消息GET和POST，而忽略消息。然后，HTTP的请求方法定义了一个解析器，解析器将接受一段字节流并返回枚举类型Method，这可以通过读取输入并查找字符串GET或POST来实现。在每种情况下，基本解析器都是使用tag!宏来构建的，它对输入进行解析用来提取指定的字符串。如果解析成功，使用map！宏将解析器的结果映射到一个函数后转换为Method。上述解析过程，处理的请求命令为POST或GET，二者占其一，不能同时出现。还可以使用alt!宏来表示以上面两个解析器的逻辑运算OR构造一个新的解析器，也就是可以用复合宏去解析指定的输入。最后，可以使用repurn\_error!宏处理解析失败的情况。如果在当前解析器中解析失败，立即返回，而不会继续进行到解析树的下一个解析器。

接下来可以通过定义parse\_request函数来解析整个请求消息。先使用ws!宏将输入中多余空格去掉，然后调用do\_parse!宏连接多个子解析器。该操作不同于其他组合器，它可以存储中间解析器的结果。这一点在返回结果时构造结构体实例时很有用。在do\_parse!宏中，先调用parse\_method，并将其结果存入一个变量。然后调用take\_until!(“ “)，该方法会不断处理输入直到遇到一个空格。其结果被map\_res!宏转换为一个字符串str。下一个解析器则会使用tag!宏删除消息序列的HTTP/。这时，将通过读取输入来解析HTTP版本，直到遇到符号\r，再将其映射回一个str。一旦所有解析完成，则返回Request对象。请注意，将使用>>符号作为解析器序列内解析器直接的分隔符。

为了方便运行解析器，定义一个辅助函数run\_parser。该函数调用解析器并对结果进行匹配，用来显示结果或错误。示例运行时，输入3个HTTP请求演示效果，其中前两个有效，第三个错误且无效。演示结果如下：

**$ cargo run**



上面的演示结果可见，在前两种情况下，所有内容都按照预期进行了解析并得到解析结果，而第三种情况下解析失败并返回自定义错误。

正如我们前面讨论的，nom的一个常见问题是调试，因为调试宏要困难得多。尤其是宏还鼓励使用特定的DSLs（比如使用>>分隔符），这可能使一些人难以使用它。目前，来自nom的一些错误消息对于找出指定解析器的错误问题帮助还不够，不过相信未来会得到改进。当然，nom也提供了一些调试辅助宏。例如，如果下层解析器没有返回“Done”，dbg！则会打印结果和输入。dbg\_dump！宏也具有类似的功能，只是还会打印出输入缓冲区的十六进制信息。根据前人的经验，还可以通过以下方法进行调试：

1. 通过将编译器选项传递给rustc来扩展宏。Cargo允许使用以下调用： cargo rustc -- -Z unstable-options-- pretty=expanded扩展和打印指定项目中的所有宏。这里，把宏扩展到用来跟踪执行和调试是很有用的。Cargo的扩展宏相关命令是：rustc -- -Z trace-macros。
2. 单独运行较小的解析器。给定一组解析器及其组合解析器，在其运行时如果出现一个错误，可以定位到出错的某个子解析器。然后，可以继续调试失败的小解析器。这对于隔离故障非常有用。
3. 使用所提供的调试宏dbg!和dbg\_dump!。这些方法可以用于调试打印语句来跟踪执行过程。

提示：pretty=expanded的选项表示目前这不是一个稳定版本的编译器。也许在将来会成为稳定版本也可能被移除。此时，不需要用-Z unstable-option标志来使用它。

下面是一个解析器组合器pom的示例。正如我们之前讨论的，这解析在很大程度上依赖于trait和操作符重载来实现解析器组合。其当前版本是1.1.0，首先新建项目：**$ cargo new --bin pom-string**



Cargo.toml配置如下：

1. [package]
2. name = "pom-string"
3. version = "0.1.0"
4. [dependencies]
5. pom = "1.1.0"：

在这个示例中，我们将解析一个示例HTTP请求，就像上次一样。这将是它的代码：

1. // chapter4/pom-string/src/main.rs
2. extern crate pom;
3. use pom::DataInput;
4. use pom::parser::{sym, one\_of, seq};
5. use pom::parser::\*;
6. use std::str;
7. // Represents one or more occurrence of an empty whitespace
8. fn space() -> Parser<'static, u8, ()> {
9. one\_of(b" \t\r\n").repeat(0..).discard()
10. }
11. // Represents a string in all lower case
12. fn string() -> Parser<'static, u8, String> {
13. one\_of(b"abcdefghijklmnopqrstuvwxyz").repeat(0..).convert(String::from\_utf8)
14. }
15. fn main() {
16. let get = b"GET /home/ HTTP/1.1\r\n";
17. let mut input = DataInput::new(get);
18. let parser = (seq(b"GET") | seq(b"POST")) \* space() \* sym(b'/') \*
19. string() \* sym(b'/') \* space() \* seq(b"HTTP/1.1");
20. let output = parser.parse(&mut input);
21. println!("{:?}", str::from\_utf8(&output.unwrap()));
22. }

首先声明pom的依赖项。然后在主函数中，将最终的组合解析器定义为多个子解析器的序列。操作符\*用于按序列顺序重载多个解析器。运算符seq表示一个从输入中匹配给定的字符串的内置解析器。操作符|表示对两个操作数执行逻辑运算OR。定义函数space()，用来清空输入中的空白位置。该函数会将接收到的空白位置丢弃。因此，该函数返回一个没有返回类型的Parser，由()表示。字符串函数同样被定义为一个英文字符，重复0次或更多次，然后转换为std::String。

此函数的返回类型是一个具有一个String的Parser。主解析器将有一个空格，一个符号/，一个字符串，一个符号/，一个空格的形式，最后以序列HTTP/1.1结束。因此，示例的解析器在解析一个示例字符串时，会产生一个OK，运行结果如下：

**$ cargo run**



基于PEG的解析器组合器可以更容易地进行调试和使用。它们也往往会产生更好的错误消息，但是这些错误消息现在还不够问题。其社区影响力还nom的大。因此，难以获取社区的帮助和支持。

## 解析二进制数据

另一个解析问题是如何解析二进制数据。这种方法适用的常见情况包括解析二进制文件和二进制协议。下面将通过示例演示如何使用nom来解析二进制数据。在示例中，为解析IPv6报头创建一个解析器。新建项目：**$ cargo new --bin nom-ipv6**

主文件代码如下：

1. // chapter4/nom-ipv6/src/main.rs
2. #[macro\_use]
3. extern crate nom;
4. use std::net::Ipv6Addr;
5. use nom::IResult;
6. // Struct representing an IPv6 header
7. #[derive(Debug, PartialEq, Eq)]
8. pub struct IPv6Header {
9. version: u8,
10. traffic\_class: u8,
11. flow\_label: u32,
12. payload\_length: u16,
13. next\_header: u8,
14. hop\_limit: u8,
15. source\_addr: Ipv6Addr,
16. dest\_addr: Ipv6Addr,
17. }
18. // Converts a given slice of [u8] to an array of 16 u8 given by
19. // [u8; 16]
20. fn slice\_to\_array(input: &[u8]) -> [u8; 16] {
21. let mut array = [0u8; 16];
22. for (&x, p) in input.iter().zip(array.iter\_mut()) {
23. \*p = x;
24. }
25. array
26. }
27. // Converts a reference to a slice [u8] to an instance of
28. // std::net::Ipv6Addr
29. fn to\_ipv6\_address(i: &[u8]) -> Ipv6Addr {
30. let arr = slice\_to\_array(i);
31. Ipv6Addr::from(arr)
32. }
33. // Parsers for each individual section of the header
34. named!(parse\_version<&[u8], u8>, bits!(take\_bits!(u8, 4)));
35. named!(parse\_traffic\_class<&[u8], u8>, bits!(take\_bits!(u8, 8)));
36. named!(parse\_flow\_label<&[u8], u32>, bits!(take\_bits!(u32, 20)));
37. named!(parse\_payload\_length<&[u8], u16>, bits!(take\_bits!(u16, 16)));
38. named!(parse\_next\_header<&[u8], u8>, bits!(take\_bits!(u8, 8)));
39. named!(parse\_hop\_limit<&[u8], u8>, bits!(take\_bits!(u8, 8)));
40. named!(parse\_address<&[u8], Ipv6Addr>, map!(take!(16), to\_ipv6\_address));
41. // The primary parser
42. named!(ipparse<&[u8], IPv6Header>,
43. do\_parse!(
44. ver: parse\_version >>
45. cls: parse\_traffic\_class >>
46. lbl: parse\_flow\_label >>
47. len: parse\_payload\_length >>
48. hdr: parse\_next\_header >>
49. lim: parse\_hop\_limit >>
50. src: parse\_address >>
51. dst: parse\_address >>
52. (IPv6Header {
53. version: ver,
54. traffic\_class: cls,
55. flow\_label: lbl,
56. payload\_length: len,
57. next\_header: hdr,
58. hop\_limit: lim,
59. source\_addr: src,
60. dest\_addr : dst
61. })
62. ));
63. // Wrapper for the parser
64. pub fn parse\_ipv6\_header(i: &[u8]) -> IResult<&[u8], IPv6Header> {
65. ipparse(i)
66. }
67. fn main() {
68. const EMPTY\_SLICE: &'static [u8] = &[];
69. let bytes = [0x60,
70. 0x00,
71. 0x08, 0x19,
72. 0x80, 0x00, 0x14, 0x06,
73. 0x40,
74. 0x2a, 0x02, 0x0c, 0x7d, 0x2e, 0x5d, 0x5d, 0x00,
75. 0x24, 0xec, 0x4d, 0xd1, 0xc8, 0xdf, 0xbe, 0x75,
76. 0x2a, 0x00, 0x14, 0x50, 0x40, 0x0c, 0x0c, 0x0b,
77. 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xbd
78. ];
79. let expected = IPv6Header {
80. version: 6,
81. traffic\_class: 0,
82. flow\_label: 33176,
83. payload\_length: 20,
84. next\_header: 6,
85. hop\_limit: 64,
86. source\_addr:
87. "2a02:c7d:2e5d:5d00:24ec:4dd1:c8df:be75".parse().unwrap(),
88. dest\_addr: "2a00:1450:400c:c0b::bd".parse().unwrap(),
89. };
90. assert\_eq!(ipparse(&bytes), IResult::Done(EMPTY\_SLICE, expected));
91. }

在这里，首先按照RFC 2460（https://tools.ietf.org/html/rfc2460）中的定义将IPv6固定报头声明为一个结构体。再定义一个名为to\_ipv6\_address的辅助函数，它将u8的流段转换为一个IPv6地址。还需要另外的辅助函数来将切片转换为固定大小的数组（在本例中为16）。之后，定义了多个解析器来借助named!宏解析结构体的每个成员。

parse\_version函数包含一个字节片段，并返回u8类型的版本号。这是通过take\_bits！宏将输入中读取4位二进制当做一个u8类型。然后使用bits！宏将输入转换为底层解析器的位流。以同样的方式，继续为头结构中的所有其他字段定义解析器。对于每一个位数，我们根据RFC获取它们所占用的位数，并将其转换为一个足够大的类型来容纳它。最后再解析IP地址。此时，先用take!宏读取16字节的IP地址，再使用map!宏将地址映射到to\_ipv6\_address函数再转换成字节流。

此时，所有要解析整个结构体的每个组件都准备好了，可以使用do\_parse！宏定义一个函数。这里，将解析得到的结果存储在临时变量中，构建并返回IPv6头结构IPv6Header的实例。在主函数中，得到一个从IPv6数据包转储中获取的字节数组，它用来表示一个有效的IPv6报头。使用自定义的解析器进行解析，并将输出与预期的匹配。因此，先前成功运行的解析器不会引发异常。

让我们回顾一下迄今为止nom使用的所有宏，列表如下：

|  |  |
| --- | --- |
| **Macro** | **Purpose** |
| named! | 通过组合较小的函数来创建一个解析函数，是链中的顶级调用。 |
| ws! | 允许解析器消耗标记之间的所有空白（\t、\r和\n）。 |
| do\_parse! | 在给定的序列中应用子解析器，可以存储中间的结果。 |
| tag! | 声明封闭解析器应该识别的静态字节序列。 |
| take\_until! | 消耗输入，直到给定的标签。 |
| take\_bits! | 使用该从输入中给定的比特数并将它们转换为给定的类型 |
| take! | 从输入中消耗指定的字节数。 |
| map\_res! | 在解析器的输出上映射一个函数（返回一个结果）。 |
| map! | 将一个函数映射到一个解析器的输出。 |
| bits! | 将给定的切片转换为一个位流。 |

## 小结

在本节中，我们详细地研究（去）序列化和解析。主要通过Serde库和相关的crate框架实现数据的序列化和反序列化，而我们还介绍了与序列化相关的数据解析方法，介绍了nom这种最常用的解析组合器，并且通过示例分别介绍了Rust中如何实现文本数据和二进制数据的解析。

# 异步编程

本章要点：认识tokio生态系统，理解Rust的异步编程软件包，熟悉futures软件包和tokio程序包的使用方法，掌握stream和sink程序包和tokio套接字多路复用的使用方法

## 引言

本章我们介绍Rust中的异步编程。要了解什么是异步编程，首先要从了解什么是同步编程开始。在同步编程模型中，代码总是按照编程语言的语义规定顺序执行。这种模式在正常操作下是没有问题的，但是当代码需要在当前上下文等待其他内容时会出现暂停执行的情况，这种情况通常称为阻塞。换句话说，当一段代码被阻塞时，它实际上处于暂停的状态，等待特定操作完成后才能继续。在这种阻塞状态下，程序保持空闲状态，无法同时执行其他任务，由此可能会引发一系列的问题。比如一个操作出于某种原因（等待资源等等）处于等待状态，则整个执行代码只有在该操作完成后才能向前推进执行。这种情况下，主线程将忙于等待某一个操作，导致资源利用率低。对于GUI应用程序，如果负责管理GUI的主线程正忙于等待其他任务，可能导致用户交互性较差。在网络应用程序执行时，往往需要等待网络中传输过来的数据，因此采用同步编程带来的问题更甚。以往多采用多线程来解决上述的问题。多线程编程将一个操作委托给后台线程处理，从而使主线程得以从该操作中解脱出来，用于处理用户交互或其他任务。与同步编程模型不同，异步编程模型是非阻塞的。它可以在操作进行时释放主线程并继续执行其他任务，待操作完成后再回到主线程继续处理结果。异步编程可以提高应用程序的响应性，避免阻塞主线程。当然，异步编程也会变得比较复杂，比如需要考虑检查自身进程完成情况并通知主线程，处理线程之间的协调问题以及其他大量的潜在问题等。Rust提供了多个crate包，支持使用基于Futures的事件循环驱动模型专门用于处理异步编程，简化了异步编程的复杂性。

## 异步编程软件包

### futures包

Rust异步编程的支柱是futures crate包。在Rust中，异步编程主要通过async关键字和Future特质来实现。在函数前加上async关键字可以创建一个异步函数，该函数返回一个实现了Future特质的类型。而Future是一个表示未来某个时刻完成的值的特质，它提供了.await语法来暂停当前任务的完成，直到Future完成。本质上Future是表示异步操作结果的占位符，这个操作的结果不是一个立即执行的值，而是一个在将来的某个时刻才会揭晓的值。

Future特质由futures crate包提供，如Trait std::future::Future，任何类型都可以实现它，通常实现future有两种模式：一种基于推模式，也被称为基于完成的模式；另一种基于拉模式，也被称为基于就绪的模式。Rust中future包实现了基于拉模式的future，其特质定义如下：

1. trait Future {
2. type Output;
3. type Error;
4. fn poll(&mut self) -> Poll<Self::Output, Self::Error>;
5. ...
6. }

这里，Output表示操作成功完成时返回的结果类型，Error表示操作失败时返回的结果类型。这些指定的方法必须实现，并且还要实现获取计算的当前状态的轮询方法。如果当前操作已经完成，则将返回结果。如果没有完成，future结构体将记录当前任务关注的给定操作的结果类型。此函数将返回一个轮询结果Poll，其定义如下：

type Poll<T, E> = Result<Async<T>, E>;

轮询结果Poll的类型可以用Async类型或错误类型E的结果表示，Async类型定义如下：

pub enum Async<T> {

Ready(T),

NotReady,

}

由上面的定义可见，Async类型是一个枚举类型，其值为Ready(T) 或 Not Ready，分别表示操作完成状态和未完成状态。因此对于轮询函数Poll，其可能返回的状态有以下三种：

（1）Ok(Async::Ready(result))：当操作成功完成，结果存储在名为result的内部变量中。

（2）Ok(Async::NotReady)：当操作尚未完成且结果不可用时。这里结果不可用并不表示有错误。

（3）Err(e)：当操作遇到错误时，这种情况下是没有任何结果的。

很容易注意到，Future本质上是一个将来可能出现的运行结果Result。如果不考虑Result可能永远无法准备好的情况，那就只有两个选项，即Ok和Err情况，这与Result完全对应。因此，一个Future可以表示任何需要大量时间来完成的事件。比如网络事件、磁盘读取事件等等。那么如何从一个给定的函数中返回一个future呢？其实有多种方法可以实现，先看下面的例子：

创建项目：**$ cargo new --bin futures-example**



配置Cargo config文件：

1. [package]
2. name = "futures-example"
3. version = "0.1.0"
4. [dependencies]
5. futures = "0.1.17"
6. futures-cpupool = "0.1.7"

在主文件中实现确定一个给定的整数是否是素数的功能。下面的例子中用两个方法分别展示两种不同的futures返回方式。

主文件代码如下：

1. // ch4/futures-example/src/main.rs
2. #feature(conservative\_impl\_trait)]
3. extern crate futures;
4. extern crate futures\_cpupool;
5. use std::io;
6. use futures::Future;
7. use futures\_cpupool::CpuPool;
8. // This implementation returns a boxed future
9. fn check\_prime\_boxed(n: u64) -> Box<Future<Output = bool, Error = io::Error>>
10. {
11. for i in 2..n {
12. if n % i == 0 { return Box::new(futures::future::ok(false)); }
13. }
14. Box::new(futures::future::ok(true))
15. }
16. // This returns a future using impl trait
17. fn check\_prime\_impl\_trait(n: u64) -> impl Future<Output = bool, Error =
18. io::Error> {
19. for i in 2..n {
20. if n % i == 0 { return futures::future::ok(false); }
21. }
22. futures::future::ok(true)
23. }
24. // This does not return a future
25. fn check\_prime(n: u64) -> bool {
26. for i in 2..n {
27. if n % i == 0 { return false }
28. }
29. true
30. }
31. fn main() {
32. let input: u64 = 58466453;
33. println!("Right before first call");
34. let res\_one = check\_prime\_boxed(input);
35. println!("Called check\_prime\_boxed");
36. let res\_two = check\_prime\_impl\_trait(input);
37. println!("Called check\_prime\_impl\_trait");
38. println!("Results are {} and {}", res\_one.wait().unwrap(),
39. res\_two.wait().unwrap());
40. let thread\_pool = CpuPool::new(4);
41. let res\_three = thread\_pool.spawn\_fn(move || {
42. let temp = check\_prime(input);
43. let result: Result<bool, ()> = Ok(temp);
44. result
45. });
46. println!("Called check\_prime in another thread");
47. println!("Result from the last call: {}", res\_three.wait().unwrap());
48. }

上面的例子中展示了两种futures的返回方式。第一种返回方式是使用特质对象返回，例如check\_prime\_boxed函数中所示。例子中对应的Box是指向堆上对象的指针类型，这是一个托管指针，当对象超出范围时，它将被自动清理干净。check\_prime\_boxed函数的返回类型是一个future的特质对象，该特质的Output设置为bool和Error设置为io:Error。因此，这代表着动态调度。第二种返回Future的方式是使用impl特质特征返回，例如上例中check\_prime\_impl\_trait函数中所示。check\_prime\_impl\_trait函数返回一个实现Future<Output=bool, Error=io::Error>的类型。由于任何实现future特质的类型都是一个future，因此该函数也能返回一个future。值得注意的是，在这种情况下，我们不需要在返回结果之前装包。因此，该方式的优点是不需要为返回future做任何的分配。上面的两个函数都使用future::ok函数来表示计算已成功完成并给出结果。

此外，还有一种间接的方式，该方式不会直接返回一个future，而是使用基于futures的线程池crate包创建和管理future。当然这种方式是比较复杂，在仅返回一个 bool类型的check\_prime 的情况下才比较适用的。在主函数中，使用cpupool的futures crate包去设置一个线程池，并运行该线程池中的最后一个函数。然后，返回一个可以调用wait得到结果的future。还有一种完全不同的方式可以返回一个实现Future 特质的自定义类型。这种方式需要额外编写代码，不太符合我们习惯，但它却是最灵活的。需要说明的是，impl特质还不是一个稳定的特征。因此，check\_prime\_impl\_trait只适用于Rust的开发版。

在构建了一个future后，下一步就是执行它。这里有三种方法：

（1）在当前线程中：这将会阻塞当前线程，直到该future执行完成。在前面的示例中，res\_one和res\_two在主线程上执行，阻止用户交互。

（2）在线程池中：这是res\_three的情况，它在名称为thread\_pool的线程池中执行。因此，在这种情况下，线程池可以自己自由地处理调用的线程。

（3）在事件循环中：在某些情况下，上述两种情况都不可能实现。因此，唯一的选择是在一个事件循环中执行futures。此时，可以使用tokio的核心crate包所提供的futures的APIs处理事件循环。在下一节中，我们将更深入地研究这个模型。

在主函数中，我们调用主线程中的前两个函数。这两个函数将阻塞主线程的执行，而最后一个函数则会在另外的线程上运行的。这种情况下，主线程可以立即自由地输出已被调用的check\_prime。它再次阻塞对future上wait的调用。请注意，futures在所有情况下都是有延迟的。输入以下命令运行该程序：

**$ cargo run**

执行结果如下：

![形状

描述已自动生成

与常规线程处理方式不同，futures可以按照正常的编程习惯链接并执行。就像是依次执行下载网页、解析 HTML、提取给定单词一样。这一系列操作中的每一个步都是一个future，下一步只能在前一步完成之后开始执行。当然，整个操作也是一个Future，只不过是由一系列futures组成的futures。当执行更大的Future时，被称为一个任务（task）。Future crate库提供了许多APIs，可用于与命名空间futures::task的任务进行交互。当用给定的类型实现Future特质（实现poll方法）时，编译器提供所有这些实现的组合算子。下面是一个使用链接实现超时功能的示例。示例使用tokio的timer crate包实现超时future。代码中存在两个相互竞争的函数，它们可以随机睡眠一段时间，然后向调用者返回一个固定的字符串。我们将同时分配所有的数据，如果返回结果与第一个函数对应的字符串匹配，则玩家一获胜。同样地，如果返回结果与第二个函数对应的字符串匹配，则玩家二获胜。如果都未匹配，则超时future库的timeout future已经被触发。示例代码如下：

创建项目：**$ cargo new --bin futures-chaining**



Cargo.toml配置文件：

1. [package]
2. name = "futures-chaining"
3. version = "0.1.0"
4. [dependencies]
5. tokio-timer = "0.1.2"
6. futures = "0.1.17"
7. futures-cpupool = "0.1.7"
8. rand = "0.3.18"

这里也使用一个线程池来执行futures\_cpupool crate包的futures。主文件代码如下：

1. // ch4/futures-chaining/src/main.rs
2. extern crate futures;
3. extern crate futures\_cpupool;
4. extern crate tokio\_timer;
5. extern crate rand;
6. use futures::future::select\_ok;
7. use std::time::Duration;
8. use futures::Future;
9. use futures\_cpupool::CpuPool;
10. use tokio\_timer::Timer;
11. use std::thread;
12. use rand::{thread\_rng, Rng};
13. // First player, identified by the string "player\_one"
14. fn player\_one() -> &'static str {
15. let d = thread\_rng().gen\_range::<u64>(1, 5);
16. thread::sleep(Duration::from\_secs(d));
17. "player\_one"
18. }
19. // Second player, identified by the string "player\_two"
20. fn player\_two() -> &'static str {
21. let d = thread\_rng().gen\_range::<u64>(1, 5);
22. thread::sleep(Duration::from\_secs(d));
23. "player\_two"
24. }
25. fn main() {
26. let pool = CpuPool::new\_num\_cpus();
27. let timer = Timer::default();
28. // Defining the timeout future
29. let timeout = timer.sleep(Duration::from\_secs(3))
30. .then(|\_| Err(()));
31. // Running the first player in the pool
32. let one = pool.spawn\_fn(|| {
33. Ok(player\_one())
34. });
35. // Running second player in the pool
36. let two = pool.spawn\_fn(|| {
37. Ok(player\_two())
38. });
39. let tasks = vecone, two];
40. // Combining the players with the timeout future
41. // and filtering out result
42. let winner = select\_ok(tasks).select(timeout).map(|(result, \_)| result);
43. let result = winner.wait().ok();
44. match result {
45. Some(("player\_one", \_)) => println!("Player one won"),
46. Some(("player\_two", \_)) => println!("Player two won"),
47. Some((\_, \_)) | None => println!("Timed out"),
48. }
49. }

两个玩家的行为非常相似；二者都生成一个1到5之间的随机数d，然后睡眠d秒。之后返回一个其对应的名称的固定字符串。最后，输出的字符串就可以唯一地标识是哪个玩家。在主函数中，首先初始化线程池和定时器。使用定时器上的组合算子在3秒之后返回一个出错的future。然后，在线程池中自动生成两个玩家，并将它们作为futures返回结果Result。请注意，由于futures是惰性的，这些函数此时还没有真正运行。之后，我们将这些futures存入一个列表中，并使用select\_ok组合算子并行运行这些futures。这个函数接受一组可迭代的futures，并选中第一个成功的future；这里只有一个限制，那就是传递给这个函数的所有futures都应该是相同类型的。因此，超时的futures不能通过。select组合算子处理两个futures，并等待其中一个执行完毕。我们可以使用select组合算子将select\_ok的结果传递给超时futures。由此，就出现了已完成的futures和未完成的futures。那么，我们可以使用map组合算子丢弃未完成的futures。最后，阻塞futures，并使用ok()通知执行链的末端。这样我们就可以将结果与已知的字符串进行比较，确定哪个futures获胜，并打印出相应的消息。

上述示例可以看出。当超时时间低于两个函数中的最大睡眠时间时，就会出现超时。而当函数所需的时间低于超时时间时，它就会首先完成并获胜。

执行下列命令：**$ cargo run**

执行结果如下：

![文本

描述已自动生成

### tokio程序包

tokio是Rust最优秀的异步运行时框架之一，具有高可靠、简单易用和使用灵活等优点。它提供了编写异步网络服务所需的几乎所有功能，主要用于开发高性能网络服务，适用从有数十个核心的大型服务器到小型嵌入式设备的各种环境。

tokio是基于Rust的Futures和async/await的，具有事件循环、异步I/O和任务调度等核心特性。其主要以下组件构成：

（1）多线程版本的异步运行时，可以运行使用 async/await 编写的代码。。

（2）标准库中阻塞API的异步版本，例如thread::sleep会阻塞当前线程，tokio中就提供了相应的异步实现版本。

（3）构建异步编程所需的生态，甚至还提供了tracing用于日志和分布式追踪，提供console用于Debug异步编程。

tokio生态系统是Rust中网络堆栈的实现。它具有标准库的所有主要功能，主要的区别是它是非阻塞的。tokio生态系统中在GitHub组织常用的crate包如下表所示：

|  |  |
| --- | --- |
| Crate | 功能说明 |
| tokio-minihttp | 在tokio中实现的简单的HTTP服务器；不应该用在产品中。 |
| tokio-core | 面向Future的网络实现；核心事件循环。 |
| tokio-io | tokio的IO原语。 |
| tokio-curl | 一个使用tokio实现的基于libcurl的HTTP客户端。 |
| tokio-uds | 使用tokio的非阻塞的unix域套接字。 |
| tokio-tls | 基于tokio的TLS和SSL实现。 |
| tokio-service | 提供了已经被广泛使用的Service特质。 |
| tokio-proto | 提供了一个使用tokio来构建网络协议的框架；这些已经广泛被使用了。 |
| tokio-socks5 | 一个使用tokio的SOCKS5服务器，非生产就绪。 |
| tokio-middleware | tokio服务的中间制品；目前缺乏必要的服务。 |
| tokio-times | 基于tokio的计时器相关功能 |
| tokio-line | 演示tokio的样本线协议。 |
| tokio-redis | 基于tokio的概念验证redis客户机；不应用于生产。 |
| service-fn | 提供一个为给定的闭包实现Service特质的函数。 |

请注意，上表所列的crate包中存在长期没有更新的情况。

尽管tokio生态系统十分丰富，但有两个基本的crates库，tokio-proto和tokio-core，是十分重要的，其他实现都是围绕这两个库工作的。

（1）tokio-proto，它为构建异步服务器和客户端提供了基础。与适于底层网络的mio架构和futures抽象库配合使用。

（2）tokio-core，它提供了事件循环来运行futures和相关的APIs，能够实现对IO的细粒度控制。

正如我们在上一节中提到的，运行futures的一种方法是运行事件循环。事件循环（在tokio中称为reactor）不断地监听已定义的事件，并在接收到事件后执行相应的操作。这里我们仍将实现功能为确定给定的输入是否为素数的例子。

创建项目如下：

**$ cargo new --bin futures-loop**



Cargo.toml配置文件如下：

1. [package]
2. name = "futures-loop"
3. version = "0.1.0"
4. [dependencies]
5. futures = "0.1"
6. tokio-core = "0.1"

本例中我们对获取输入值采用无限循环。每次输入，都会删除换行符和空格，并将其解析为u64。主文件代码如下：

1. // ch4/futures-loop/src/main.rs
2. extern crate futures;
3. extern crate tokio\_core;
4. use std::io;
5. use std::io::BufRead;
6. use futures::Future;
7. use tokio\_core::reactor::Core;
8. fn check\_prime\_boxed(n: u64) -> Box<dyn Future<Output = bool, Error = io::Error>> {
9. {
10. for i in 2..n {
11. if n % i == 0 {
12. return Box::new(futures::future::ok(false));
13. }
14. }
15. Box::new(futures::future::ok(true))
16. }
17. fn main() {
18. let mut core = Core::new().expect("Could not create event loop");
19. let stdin = io::stdin();
20. loop {
21. let mut line = String::new();
22. stdin
23. .lock()
24. .read\_line(&mut line)
25. .expect("Could not read from stdin");
26. let input = line.trim()
27. .parse::<u64>()
28. .expect("Could not parse input as u64");
29. let result = core.run(check\_prime\_boxed(input))
30. .expect("Could not run future");
31. println!("{}", result);
32. }
33. }

以上代码的主函数中，我们创建变量core并启动无限循环监听。我们使用core的run方法来启动一个要异步执行future的任务，获取其结果并使用标准输出打印该结果。以下是执行结果：

图形用户界面, 应用程序

描述已自动生成

Tokio-proto crate包是构建异步服务器（和客户端）的工具包。使用crate包构建的服务器涉及从低到高三个层次的编码：

（1）底层主要涉及编码解码器。该层最接近物理介质，也是最底层。编码解码器用于将原始字节解码为rust的数据类型，这里主要是规定了如何从套接字读写数据。编写编码解码器代码相当于实现处理字节流的库中的给定特质。

（2）中间层主要涉及各种协议。该层工作在编码解码器和运行协议的事件循环之间，工作在该层的各种协议起到承上启下的作用。tokio库支持请求-响应类型协议、多路复用协议和流协议等多种类型的协议，不同的应用程序涉及不同的协议。

（3）最高层涉及各种服务。这里的服务实际上以future的形式运行的，这可以看作一个异步函数，即将输入转换为最终响应结果。大部分计算工作都是在该层完成的。

上面每个层次都是可以自定义的，协议类型，服务和编码解码器都是可替换的。下面是使用tokio-proto实现简单服务的示例。该示例中采用了传统的请求-响应服务，提供了一个基于文本的接口。当其接收一个整数数字，将其排序序列作为一个数组返回。如果接收非整数，则会发送相同的消息。

创建项目：**$ cargo new --bin collatz-proto**



Cargo config配置文件如下：

1. [package]
2. name = "collatz-proto"
3. version = "0.1.0"
4. [dependencies]
5. bytes = "0.4"
6. futures = "0.1"
7. tokio-io = "0.1"
8. tokio-core = "0.1"
9. tokio-proto = "0.1"
10. tokio-service = "0.1"

如前所述，需要实现不同的层的编码。此时，每层都不需要保持太多的状态。因此，它们可以使用单元结构来表示。如果不是这样，我们需要在这些地方放一些数据。

1. // ch4/collatz-proto/src/main.rs
2. extern crate bytes;
3. extern crate futures;
4. extern crate tokio\_io;
5. extern crate tokio\_proto;
6. extern crate tokio\_service;
7. use std::io;
8. use std::str;
9. use bytes::BytesMut;
10. use tokio\_io::codec::{Encoder, Decoder};
11. use tokio\_io::{AsyncRead, AsyncWrite};
12. use tokio\_io::codec::Framed;
13. use tokio\_proto::pipeline::ServerProto;
14. use tokio\_service::Service;
15. use futures::{future, Future};
16. use tokio\_proto::TcpServer;
17. // Codec implementation, our codec is a simple unit struct
18. pub struct CollatzCodec;
19. // Decoding a byte stream from the underlying socket
20. impl Decoder for CollatzCodec {
21. type Output = String;
22. type Error = io::Error;
23. fn decode(&mut self, buf: &mut BytesMut) -> io::Result<Option<String>>
24. {
25. // Since a newline denotes end of input, read till a newline
26. if let Some(i) = buf.iter().position(|&b| b == b'\n') {
27. let line = buf.split\_to(i);
28. // and remove the newline
29. buf.split\_to(1);
30. // try to decode into an UTF8 string before passing
31. // to the protocol
32. match str::from\_utf8(&line) {
33. Ok(s) => Ok(Some(s.to\_string())),
34. Err(\_) => Err(io::Error::new(io::ErrorKind::Other,
35. "invalid UTF-8")),
36. }
37. } else {
38. Ok(None)
39. }
40. }
41. }
42. // Encoding a string to a newline terminated byte stream
43. impl Encoder for CollatzCodec {
44. type Output = String;
45. type Error = io::Error;
46. fn encode(&mut self, msg: String, buf: &mut BytesMut) ->
47. io::Result<()> {
48. buf.extend(msg.as\_bytes());
49. buf.extend(b"\n");
50. Ok(())
51. }
52. }
53. // Protocol implementation as an unit struct
54. pub struct CollatzProto;
55. impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
56. type Request = String;
57. type Response = String;
58. type Transport = Framed<T, CollatzCodec>;
59. type BindTransport = Result<Self::Transport, io::Error>;
60. fn bind\_transport(&self, io: T) -> Self::BindTransport {
61. Ok(io.framed(CollatzCodec))
62. }
63. }
64. // Service implementation
65. pub struct CollatzService;
66. fn get\_sequence(n: u64) -> Vec<u64> {
67. let mut n = n.clone();
68. let mut result = vec];
69. result.push(n);
70. while n > 1 {
71. if n % 2 == 0 {
72. n /= 2;
73. } else {
74. n = 3 \* n + 1;
75. }
76. result.push(n);
77. }
78. result
79. }
80. impl Service for CollatzService {
81. type Request = String;
82. type Response = String;
83. type Error = io::Error;
84. type Future = Box<Future<Output = Self::Response, Error = Self::Error>>;
85. fn call(&self, req: Self::Request) -> Self::Future {
86. match req.trim().parse::<u64>() {
87. Ok(num) => {
88. let res = get\_sequence(num);
89. Box::new(future::ok(format!("{:?}", res)))
90. }
91. Err(\_) => Box::new(future::ok("Could not parse input as an
92. u64".to\_owned())),
93. }
94. }
95. }
96. fn main() {
97. let addr = "0.0.0.0:9999".parse().unwrap();
98. let server = TcpServer::new(CollatzProto, addr);
99. server.serve(|| Ok(CollatzService));
100. }

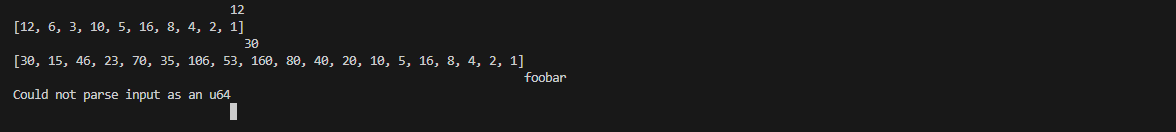
第一步是编写编码解码器，实现tokio\_io::codec的Encoder函数和Decoder函数完成从套接字读写数据的功能。这里，我们不需要改变原始套接字，只要把字节流作为输入，就可以任意处理它。根据自定义协议CollatzProto，用一个换行符代表输入结束。因此，解码器Decoder在读取数据过程中，如果遇到一个换行符，表示读取结束，此时返回读取到的除换行符之外的UTF-8编码字符串。如果出现错误，则返回None。而编码器Encoder正好相反：它将一个字符串转换为一个字节流。

第二步是定义协议，本例中的定义协议部分比较简单，没有实现多路复用或流传输。该步主要实现bind\_transport，用于将编码解码器绑定到原始套接字，这部分稍后会详细讨论。下面只需要保证Request类型和响应Response类型与编码解码器的类型相匹配即可。

第三步是实现该服务，这一步是通过声明一个单元结构并在其上实现Service特质实现的。示例中辅助函数get\_sequence返回以输入类型为u64的colatz序列。Service中的call方法实现了响应的执行操作。编码解码器以字符串String的形式返回输入，我们需要将输入转换为u64类型。如果执行正常，就调用辅助函数并返回静态字符串类型的结果，否则会返回错误。整个过程主函数看起来类似于使用标准网络类型的函数，而实际上我们是使用来自tokio库的TcpServer类型，实现接收套接字，与编码解码器绑定，以及自定义的协议的执行过程。最后调用serve方法闭包传递服务。serve方法负责管理事件循环和在退出释放资源。

调用telnet命令测试结果如下：**$ telnet localhost 9999**

****



在客户端实现在事件循环中运行future的示例如下：

创建项目：**$ cargo new --bin collatz-client**

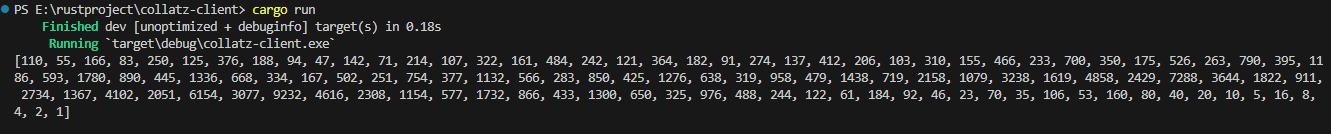
Cargo配置文件如下：

1. [package]
2. name = "collatz-client"
3. version = "0.1.0"
4. [dependencies]
5. futures = "0.1"
6. tokio-core = "0.1"
7. tokio-io = "0.1"

主文件如下：

1. // ch4/collatz-client/src/main.rs
2. extern crate futures;
3. extern crate tokio\_core;
4. extern crate tokio\_io;
5. use std::net::SocketAddr;
6. use std::io::BufReader;
7. use futures::Future;
8. use tokio\_core::reactor::Core;
9. use tokio\_core::net::TcpStream;
10. fn main() {
11. let mut core = Core::new().expect("Could not create event loop");
12. let handle = core.handle();
13. let addr: SocketAddr = "127.0.0.1:9999".parse().expect("Could not parse as SocketAddr");
14. let socket = TcpStream::connect(&addr, &handle);
15. let request = socket.and\_then(|socket| {
16. tokio\_io::io::write\_all(socket, b"110\n")
17. });
18. let response = request.and\_then(|(socket, \_request)| {
19. let sock = BufReader::new(socket);
20. tokio\_io::io::read\_until(sock, b'\n', Vec::new())
21. });
22. let (\_socket, data) = core.run(response).unwrap();
23. println!("{}", String::from\_utf8\_lossy(&data));
24. }

客户端向服务器发送一个整数（十进制110），将其放在循环中读取输入并发送这些整数。这里创建事件循环并获取其句柄。然后，使用异步TcpStream连接到给定地址上的服务器，返回一个future。使用and\_then该future与闭包组合向给定套接字写入。整个构造返回称为request的future，与读取的future链接。response的future运行在事件循环，读取响应并打印。以上步骤都遵守之前自定义的协议。运行结果如下：**$ cargo run**



出现之前的列表中的tokio-curl库也可用于异步编程。下面的示例演示了tokio-curl的用法，实现从某个地址下载单个文件，将其写入本地磁盘，并打印从服务器返回的报头的功能。新建项目：**$ cargo new --bin tokio-curl**



Cargo配置如下：

1. [package]
2. name = "tokio-curl"
3. version = "0.1.0"
4. edition = "2021"
5. # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
6. [dependencies]
7. curl = "0.4.10"
8. futures = "0.1.15"
9. log = "0.4"
10. tokio-io = "0.1"
11. tokio-core = "0.1"
12. reqwest = "0.11"

tokio-curl库是围绕Rust curl库的动态库，我们也需要包括它。

主文件代码如下：

1. // ch4/toki-curl/src/main.rs
2. extern crate curl;
3. extern crate tokio\_core;
4. use curl::easy::Easy;
5. use tokio\_core::reactor::Core;
6. use std::fs::File;
7. use std::io::Write;
8. fn main() {
9. // 创建一个 tokio 的事件循环
10. let mut core = Core::new().unwrap();
11. let handle = core.handle();
12. // 创建一个 Curl easy 句柄
13. let mut easy = Easy::new();
14. // 设置 URL
15. easy.url("https://nodejs.org/dist/v16.14.0/node-v16.14.0-win-x64.zip").unwrap();
16. // 设置下载文件路径
17. let mut file = File::create("foo.zip").unwrap();
18. // 设置写入回调函数
19. easy.write\_function(move |data| {
20. file.write\_all(data).unwrap();
21. Ok(data.len())
22. }).unwrap();
23. // 执行下载操作
24. let result = easy.perform();
25. // 检查结果
26. match result {
27. Ok(\_) => {
28. // 打印响应头
29. println!("Response headers:");
30. let mut headers = Vec::new();
31. easy.get(true).unwrap();
32. {
33. let mut transfer = easy.transfer();
34. transfer.header\_function(|header| {
35. headers.push(String::from\_utf8\_lossy(header).into\_owned());
36. true
37. }).unwrap();
38. transfer.perform().unwrap();
39. }
40. for header in &headers {
41. println!("{}", header);
42. }
43. // 打印响应码
44. println!("Response code: {:?}", easy.response\_code().unwrap());
45. }
46. Err(err) => {
47. // 处理错误
48. eprintln!("Error: {:?}", err);
49. }
50. }
51. }

上面的示例中，首先创建事件循环和HTTP会话，接着使用来自curl crate库的API Easy创建了一个句柄，libcurl使用该句柄来处理请求。调用get方法，其参数是个bool值，用来表示是否要对HTTP GET处理。继而将URL传递给该句柄。然后将两个传递来的回调函数header\_function和write\_function作为闭包。两个回调函数分别用来显示了每个客户端报头以及将所得数据写入文件。接下来，通过调用perform函数来创建请求。最后，在事件循环中执行该请求，并打印返回的状态代码。

执行命令**$ cargo run** ，显示运行结果如下：

![文本

低可信度描述已自动生成

执行过程中，先在当前目录中生成foo.zip文件，再从服务器下载该文件，最后在比较服务器文件与当前目录文件的SHA和验证是否相同。

## 异步网络编程

### stream和sink程序包

futures crate库提供了Stream特质，其与Future类似。但是Future对应的是一个Output的状态变化，而Stream则是类似iterator，在结束之前能够得到多个值。或者简单理解为，Stream是由以协议的Future组成，我们可以从Stream读取各个Future的结果，直到Stream结束。Stream trait特质的定义如下：

1. trait Stream {
2. type Output;
3. fn poll\_next(self: Pin<&mut Self>, Iw:&LocalWaker) -> Poll<Option<Self::Output>;
4. }

其中，poll\_next函数有三种可能的返回值，分别如下：

Poll::Pending，说明下一个值还没有就绪，仍然需要等待。

Poll::Ready(Some(val))，已经就绪，成功返回一个值，程序可以通过调用poll\_next再获取下一个值。

Poll::Ready(None)，表示Stream已经结束，不应该调用poll\_next。

1. trait Stream {
2. type Output;
3. type Error;
4. fn poll(& mut self) -> Poll<Option<Self::Output>, Self::Error>;
5. ...
6. }

这里唯一的区别是，返回类型被包装在一个选项Option中，就像迭代器Iterator特质一样。因此，这里的None表示流已经终止。此外，所有的流都是futures，可以使用into\_future来转换。让我们看一个使用这个构造的示例。我们将部分重用上一章中的科拉茨示例（collatz example）。

创建项目：**$ cargo new --bin streams**



Cargo config配置如下：

1. [package]
2. name = "streams"
3. version = "0.1.0"
4. [dependencies]
5. futures = "0.1.17"
6. rand = "0.3.18"

在这种情况下，结构体CollatzStream中两个域，分别表示当前状态和结束状态（始终为1）。我们将在此基础上实现Stream特质，使其表现为流：

主文件代码如下：

1. // ch4/streams/src/main.rs
2. extern crate futures;
3. extern crate rand;
4. use std::{io, thread};
5. use std::time::Duration;
6. use futures::stream::Stream;
7. use futures::{Poll, Async};
8. use rand::{thread\_rng, Rng};
9. use futures::Future;
10. // This struct holds the current state and the end condition
11. // for the stream
12. #[derive(Debug)]
13. struct CollatzStream {
14. current: u64,
15. end: u64,
16. }
17. // A constructor to initialize the struct with defaults
18. impl CollatzStream {
19. fn new(start: u64) -> CollatzStream {
20. CollatzStream {
21. current: start,
22. end: 1
23. }
24. }
25. }
26. // Implementation of the Stream trait for our struct
27. impl Stream for CollatzStream {
28. type Output = u64;
29. type Error = io::Error;
30. fn poll(&mut self) -> Poll<Option<Self::Output>, io::Error> {
31. let d = thread\_rng().gen\_range::<u64>(1, 5);
32. thread::sleep(Duration::from\_secs(d));
33. if self.current % 2 == 0 {
34. self.current = self.current / 2;
35. } else {
36. self.current = 3 \* self.current + 1;
37. }
38. if self.current == self.end {
39. Ok(Async::Ready(None))
40. } else {
41. Ok(Async::Ready(Some(self.current)))
42. }
43. }
44. }
45. fn main() {
46. let stream = CollatzStream::new(10);
47. let f = stream.for\_each(|num| {
48. println!("{}", num);
49. Ok(())
50. });
51. f.wait().ok();
52. }

用随机数字表示睡眠时间来模拟返回延迟。轮询返回Ok(ASync::Reade(None))，表示流在达到1时已经完成。否则，返回当前状态Ok(Async::Ready(Some(self.current)))。实现部分除了定义了流之外，与迭代器相同。在主函数中初始化结构体后，使用for\_each组合算子来打印流中的每个项，并返回一个futures。然后调用wait和ok阻塞并获取所有结果。示例运行结果如下：

文本

描述已自动生成

就像Futures特质一样，Stream特质还支持其他组合算子。与Stream对应的是Sink，Sink是一个异步事件的接收器。这用来表示Rust通道、网络套接字、文件描述符等的发送终端非常有用。

任何异步系统中的一个常见模式都是同步。这一点变得很重要，因为组件通常需要相互通信来传递数据或协调任务。我们在过去用渠道解决了这个问题。但是这些构造在这里并不适用，因为标准库中的通道实现并不是异步的。因此，futures有自己的通道实现，它提供了对异步系统的所有保证。示例如下：

新建项目：**$ cargo new --bin futures-ping-pong**



Cargo config配置如下：

1. [package]
2. name = "futures-ping-pong"
3. version = "0.1.0"
4. [dependencies]
5. futures = "0.1"
6. tokio-core = "0.1"
7. rand = "0.3.18"

示例中有两个函数sender和receiver分别表示发送方和接收方。发送方在等待随机时间后，随机地返回“ping”或“pong”。其代码如下：

1. // ch4/futures-ping-pong/src/main
2. extern crate futures;
3. extern crate rand;
4. extern crate tokio\_core;
5. use std::thread;
6. use std::fmt::Debug;
7. use std::time::Duration;
8. use futures::Future;
9. use rand::{thread\_rng, Rng};
10. use futures::sync::mpsc;
11. use futures::{Sink, Stream};
12. use futures::sync::mpsc::Receiver;
13. // Randomly selects a sleep duration between 1 and 5 seconds. Then
14. // randomly returns either "ping" or "pong"
15. fn sender() -> &'static str {
16. let mut d = thread\_rng();
17. thread::sleep(Duration::from\_secs(d.gen\_range::<u64>(1, 5)));
18. d.choose(&["ping", "pong"]).unwrap()
19. }
20. // Receives input on the given channel and prints each Output
21. fn receiver<T: Debug>(recv: Receiver<T>) {
22. let f = recv.for\_each(|Output| {
23. println!("{:?}", Output);
24. Ok(())
25. });
26. f.wait().ok();
27. }
28. fn main() {
29. let (tx, rx) = mpsc::channel(100);
30. let h1 = thread::spawn(|| {
31. tx.send(sender()).wait().ok();
32. });
33. let h2 = thread::spawn(|| {
34. receiver::<&str>(rx);
35. });
36. h1.join().unwrap();
37. h2.join().unwrap();
38. }

futures crate包提供了两种类型的channel组件：一种是oneshot channel，即一次只能发送一次消息；它的特点是发送端只能发送一次消息，接收端只能接收一次消息。一旦消息被发送或接收，channel 就会被关闭。另一种是可以多次使用的常规mpsc通道，即多生产者但消费者通道（Multi-Producers Single Consumer Channel）；MPSC的特点就是可以有多个生产者，但只有一个消费者。因此，发送端tx可以被随意复制多份，但是接收端rx只能有一个。在主函数中，我们得到了通道的两端，并在另外的线程中生成发送者为future，而接收者在另一个线程中生成。在这两种情况下，都记录下对应的句柄，等待它们之后完成（使用join）。接收者将通道的接收端作为参数传入，由于接收者实现了Stream，所以可以使用and\_then组合算子来打印出该值。最后，在接收者函数结束前，调用future的wait()和ok()。在主函数中，通过操作两个线程句柄实现相关功能。

运行示例如下图，可见根据通道发送的内容，在接收端随机打印出结果“ping”或“pong”。

文本

描述已自动生成

futures crate包还提供了一个futures::sync::BiLock的锁定机制，这与std::sync::Mutex的加锁机制的。这是一个面向future的互斥锁，它在两个所有者之间仲裁一个共享资源。请注意，一个BiLock受限于只针对两个futures。它是这样工作的：我们有兴趣的是修改最后一个例子，以在调用发送者函数时显示计数器。现在，我们的计数器需要有线程安全的，以便它可以在消费者之间共享。

新建项目：**$ cargo new --bin future-bilock**

Cargo.toml配置文件不变，主文件代码如下：

1. // ch4/future-bilock/src/main.rs
2. extern crate futures;
3. extern crate rand;
4. use std::thread;
5. use std::fmt::Debug;
6. use std::time::Duration;
7. use futures::{Future, Async};
8. use rand::{thread\_rng, Rng};
9. use futures::sync::{mpsc, BiLock};
10. use futures::{Sink, Stream};
11. use futures::sync::mpsc::Receiver;
12. // Increments the shared counter if it can acquire a lock, then
13. // sleeps for a random duration between 1 and 5 seconds, then
14. // randomly returns either "ping" or "pong"
15. fn sender(send: &BiLock<u64>) -> &'static str {
16. match send.poll\_lock() {
17. Async::Ready(mut lock) => \*lock += 1,
18. Async::NotReady => ()
19. }
20. let mut d = thread\_rng();
21. thread::sleep(Duration::from\_secs(d.gen\_range::<u64>(1, 5)));
22. d.choose(&["ping", "pong"]).unwrap()
23. }
24. // Tries to acquire a lock on the shared variable and prints it's
25. // value if it got the lock. Then prints each Output in the given
26. // stream
27. fn receiver<T: Debug>(recv: Receiver<T>, recv\_lock: BiLock<u64>) {
28. match recv\_lock.poll\_lock() {
29. Async::Ready(lock) => println!("Value of lock {}", \*lock),
30. Async::NotReady => ()
31. }
32. let f = recv.for\_each(|Output| {
33. println!("{:?}", Output);
34. Ok(())
35. });
36. f.wait().ok();
37. }
38. fn main() {
39. let counter = 0;
40. let (send, recv) = BiLock::new(counter);
41. let (tx, rx) = mpsc::channel(100);
42. let h1 = thread::spawn(move || {
43. tx.send(sender(&send)).wait().ok();
44. });
45. let h2 = thread::spawn(|| {
46. receiver::<&str>(rx, recv);
47. });
48. h1.join().unwrap();
49. h2.join().unwrap();
50. }

虽然这与上一个例子基本相同，但也有一些差异。在主函数中，先将计数器置为零，再对计数器创建一个BiLock，其构造函数返回通道两端的句柄。然后，创建了mpsc通道并创建发送方的spawn子线程。现在，发送方已经被修改为引用一个BiLock。函数中使用poll\_lock获取一个锁，如果成功，计数器加1。否则，什么也不做。之后，继续返回“ping”或“pong”。接收方也被修改为使用BiLock。同样尝试获取一个锁，如果成功，将打印出被锁定的数据值。在主函数中，对这些futures生成spawn子线程，并等待这些futures完成。

以下是在一个不成功的运行中，当双方都未能获得锁。在一个真实的例子中，我们希望优雅地处理这个错误，然后重试。为了简洁起见，我们忽略了这部分：

**$ cargo run**

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/futures-bilock`

thread '<unnamed>' panicked at 'no Task is currently running',

libcore/option.rs:917:5

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:

Any', libcore/result.rs:945:5

下面则是一个成功的运行：

文本

描述已自动生成

### tokio套接字复用

服务器中的异步请求处理的一个模型是通过多路复用传入的连接。在这种情况下，每个连接都被分配了某种类型的唯一ID，并且只要准备好就会发出回复，而不管接收它的顺序如何。因此，这允许更高的吞吐量，因为最短的工作job隐式地获得最高的优先级。该模型还使服务器高度响应大量不同复杂度的传入请求。传统的类似于unix的系统使用选择和轮询系统调用套接字复用来支持这一点。

在tokio生态系统中，这反映在许多能够实现多路协议的特质上。该服务器的基本结构与一个简单的服务器相同：具有编解码器，使用编解码器的协议，以及一个实际运行该协议的服务。这里唯一的区别是，我们将为每个传入的请求分配一个请求ID。这将在稍后用于在发回响应时消除歧义。我们还需要实现来自tokio\_proto::multiplex命名空间中的一些特质。例如，修改collatz服务器，并向其添加多路复用。在本例中，项目的创建有所不同，因为计划使用Cargo运行二进制文件，而我们的项目将是一个库。因此项目创建命令如下：

**$ cargo new collatz-multiplexed**

而Cargo config配置文件如下：

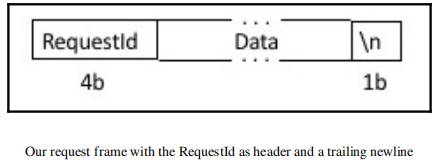
1. [package]
2. name = "collatz-multiplexed"
3. version = "0.1.0"
4. [dependencies]
5. bytes = "0.4"
6. futures = "0.1"
7. tokio-io = "0.1"
8. tokio-core = "0.1"
9. tokio-proto = "0.1"
10. tokio-service = "0.1"

这里的lib.rs代码文件如下：

1. // ch4/collatz-multiplexed/src/lib.rs
2. extern crate bytes;
3. extern crate futures;
4. extern crate tokio\_core;
5. extern crate tokio\_io;
6. extern crate tokio\_proto;
7. extern crate tokio\_service;
8. use futures::{future, Future};
9. use tokio\_io::{AsyncRead, AsyncWrite};
10. use tokio\_io::codec::{Decoder, Encoder, Framed};
11. use tokio\_core::net::TcpStream;
12. use tokio\_core::reactor::Handle;
13. use tokio\_proto::TcpClient;
14. use tokio\_proto::multiplex::{ClientProto, ClientService, RequestId,
15. ServerProto};
16. use tokio\_service::Service;
17. use bytes::{BigEndian, Buf, BufMut, BytesMut};
18. use std::{io, str};
19. use std::net::SocketAddr;
20. // Everything client side
21. // Represents a client connecting to our server
22. pub struct Client {
23. inner: ClientService<TcpStream, CollatzProto>,
24. }
25. impl Client {
26. pub fn connect(
27. addr: &SocketAddr,
28. handle: &Handle,
29. ) -> Box<Future<Output = Client, Error = io::Error>> {
30. let ret = TcpClient::new(CollatzProto)
31. .connect(addr, handle)
32. .map(|service| Client {
33. inner: service,
34. });
35. Box::new(ret)
36. }
37. }
38. impl Service for Client {
39. type Request = String;
40. type Response = String;
41. type Error = io::Error;
42. type Future = Box<Future<Output = String, Error = io::Error>>;
43. fn call(&self, req: String) -> Self::Future {
44. Box::new(self.inner.call(req).and\_then(move |resp| Ok(resp)))
45. }
46. }
47. // Everything server side
48. pub struct CollatzCodec;
49. pub struct CollatzProto;
50. // Represents a frame that has a RequestId and the actual data (String)
51. type CollatzFrame = (RequestId, String);
52. impl Decoder for CollatzCodec {
53. type Output = CollatzFrame;
54. type Error = io::Error;
55. fn decode(&mut self, buf: &mut BytesMut) ->
56. Result<Option<CollatzFrame>, io::Error> {
57. // Do not proceed if we haven't received at least 6 bytes yet
58. // 4 bytes for the RequestId + data + 1 byte for newline
59. if buf.len() < 5 {
60. return Ok(None);
61. }
62. let newline = buf[4..].iter().position(|b| \*b == b'\n');
63. if let Some(n) = newline {
64. let line = buf.split\_to(n + 4);
65. buf.split\_to(1);
66. let request\_id = io::Cursor::new(&line[0..4]).get\_u32::<BigEndian>();
67. return match str::from\_utf8(&line.as\_ref()[4..]) {
68. Ok(s) => Ok(Some((u64::from(request\_id),
69. s.to\_string()))),
70. Err(\_) => Err(io::Error::new(io::ErrorKind::Other,
71. "invalid string")),
72. };
73. }
74. // Frame is not complete if it does not have a newline at the end
75. Ok(None)
76. }
77. }
78. impl Encoder for CollatzCodec {
79. type Output = CollatzFrame;
80. type Error = io::Error;
81. fn encode(&mut self, msg: CollatzFrame, buf: &mut BytesMut) ->
82. io::Result<()> {
83. // Calculate final message length first
84. let len = 4 + msg.1.len() + 1;
85. buf.reserve(len);
86. let (request\_id, msg) = msg;
87. buf.put\_u32::<BigEndian>(request\_id as u32);
88. buf.put\_slice(msg.as\_bytes());
89. buf.put\_u8(b'\n');
90. Ok(())
91. }
92. }
93. impl<T: AsyncRead + AsyncWrite + 'static> ClientProto<T> for CollatzProto {
94. type Request = String;
95. type Response = String;
96. type Transport = Framed<T, CollatzCodec>;
97. type BindTransport = Result<Self::Transport, io::Error>;
98. fn bind\_transport(&self, io: T) -> Self::BindTransport {
99. Ok(io.framed(CollatzCodec))
100. }
101. }
102. impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
103. type Request = String;
104. type Response = String;
105. type Transport = Framed<T, CollatzCodec>;
106. type BindTransport = Result<Self::Transport, io::Error>;
107. fn bind\_transport(&self, io: T) -> Self::BindTransport {
108. Ok(io.framed(CollatzCodec))
109. }
110. }
111. pub struct CollatzService;
112. fn get\_sequence(mut n: u64) -> Vec<u64> {
113. let mut result = vec];
114. result.push(n);
115. while n > 1 {
116. if n % 2 == 0 {
117. n /= 2;
118. } else {
119. n = 3 \* n + 1;
120. }
121. result.push(n);
122. }
123. result
124. }
125. impl Service for CollatzService {
126. type Request = String;
127. type Response = String;
128. type Error = io::Error;
129. type Future = Box<Future<Output = Self::Response, Error = Self::Error>>;
130. fn call(&self, req: Self::Request) -> Self::Future {
131. match req.trim().parse::<u64>() {
132. Ok(num) => {
133. let res = get\_sequence(num);
134. Box::new(future::ok(format!("{:?}", res)))
135. }
136. Err(\_) => Box::new(future::ok("Could not parse input as an u64".to\_owned())),
137. }
138. }
139. }

Tokio库提供了一个内置类型RequestId来表示请求帧的唯一ID，所有与之相关的状态都由tokio在内部管理。请求帧被定义一个自定义数据类型CollatzFrame；它由RequestId类型和String类型组成。最后还需要实现CollatzCodec的解码器Decoder和编码器Encoder。实现编码解码器时，必须考虑帧头中的RequestID类型和结尾的换行符。因为RequestId类型是一个u64，占用4个字节，加上换行符占用1个字节，所以请求帧最少占用5个字节。因此，如果接收的字节数少于5个，说明数据帧没有被完全接收。当然，这并不是说此时就是传输错误了，因为帧仍然在传输中，所以返回一个Ok(None)。然后，检查缓冲区是否有一个换行符（按照自定义的协议规定）。正常执行过程中，解码器将从前4个字节中解析RequestID（请注意，这将是按照网络字节的顺序进行的），再创建CollatzFrame的实例并返回它。编码器则正相反，即将RequestID填入数据帧中，再填入实际数据，最后以换行符结束。

下一步是需要实现CollatzProto的ServerProto和ClientProto；这部分处于编码解码器与传输层之间的，需要进行绑定。最后再实现该服务。请注意，在实现编码解码器之后，我们不需要关心处理RequestID，因为后期阶段根本看不到它。编解码器处理和管理它，同时将实际数据传递到以后的层。所以请求数据帧结构是以RequestId作为报头开始，以换行符\n作为结束的，数据帧表示如下：



客户端也是基于tokio的。Client结构包装了一个ClientService的实例，它包含了底层的TCP流和要使用的协议实现。对Client类型有一个便捷函数connect，它连接到指定的服务器并返回一个future。最后，为Client实现了Service，其中call方法返回一个future。我们将服务器和客户端作为示例运行，并将它们放在一个称为示例examples的目录中。这样，cargo库就知道这些应该作为这个crate单元包的相关例子来运行。该服务器的代码如下：

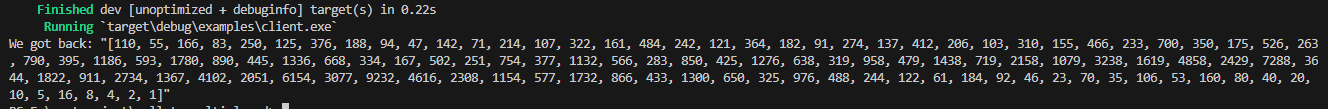
1. // ch4/collatz-multiplexed/examples/server.rs
2. extern crate collatz\_multiplexed as collatz;
3. extern crate tokio\_proto;
4. use tokio\_proto::TcpServer;
5. use collatz::{CollatzService, CollatzProto};
6. fn main() {
7. let addr = "0.0.0.0:9999".parse().unwrap();
8. TcpServer::new(CollatzProto, addr).serve(|| Ok(CollatzService));
9. }

这和上次差不多，只是在一个不同的文件中。我们必须声明我们的父crate单元包作为一个外部依赖关系，以便Cargo可以正确地连接一切。下面是客户端的代码：

1. // ch4/collatz-multiplexed/examples/client.rs
2. extern crate collatz\_multiplexed as collatz;
3. extern crate futures;
4. extern crate tokio\_core;
5. extern crate tokio\_service;
6. use futures::Future;
7. use tokio\_core::reactor::Core;
8. use tokio\_service::Service;
9. pub fn main() {
10. let addr = "127.0.0.1:9999".parse().unwrap();
11. let mut core = Core::new().unwrap();
12. let handle = core.handle();
13. core.run(
14. collatz::Client::connect(&addr, &handle)
15. .and\_then(|client| {
16. client.call("110".to\_string())
17. .and\_then(move |response| {
18. println!("We got back: {:?}", response);
19. Ok(())
20. })
21. })
22. ).unwrap();
23. }

使用tokio-core在一个事件循环中运行客户端。我们使用在客户端上定义的连接方法来获得一个future的包装连接。我们使用and\_then组合算子，并使用调用方法向服务器发送一个字符串。由于这个方法也返回一个future，我们可以在内部future上使用and\_then组合算子来提取响应，然后通过返回一个Ok(())来解析它。这也解决了外部的future。

现在，如果我们打开两个终端并在一个中运行服务器，在另一个中运行客户端，这里是我们应该在客户端中看到的内容。请注意，由于我们没有复杂的重试和错误处理，服务器应该在客户端之前运行：**$ cargo run --example client**。不出所料，这个输出与我们之前得到的结果相匹配。



## 示例：流协议

在许多情况下，协议规定了报文格式。对协议的实现，就体现为封装好的带有报头的数据单元。服务器通常先读取报头，再根据报头内容决定下一步如何处理数据。以IP协议为例，IP协议定义了一个带有目的地址的报头，服务器先处理报头，运行最长的前缀匹配算法去匹配对应的目的地址，再去读取IP报文的消息体。

下面使用tokio来编写服务器程序。对之前的collatz协议进行扩展，该协议包括报头和数据体。创建项目：**$ cargo new collatz-streaming**



Cargo config配置如下：

1. [package]
2. name = "collatz-streaming"
3. version = "0.1.0"
4. [dependencies]
5. bytes = "0.4"
6. futures = "0.1"
7. tokio-io = "0.1"
8. tokio-core = "0.1"
9. tokio-proto = "0.1"
10. tokio-service = "0.1"

该例子包括四个部分：

第一部分主要用于设置客户端：

1. // ch4/collatz-streaming/src/lib.rs
2. extern crate bytes;
3. extern crate futures;
4. extern crate tokio\_core;
5. extern crate tokio\_io;
6. extern crate tokio\_proto;
7. extern crate tokio\_service;
8. use futures::{future, Future, Poll, Stream};
9. use futures::sync::mpsc;
10. use tokio\_io::{AsyncRead, AsyncWrite};
11. use tokio\_io::codec::{Decoder, Encoder, Framed};
12. use tokio\_core::reactor::Handle;
13. use tokio\_proto::TcpClient;
14. use tokio\_proto::streaming::{Body, Message};
15. use tokio\_proto::streaming::pipeline::{ClientProto, Frame, ServerProto};
16. use tokio\_proto::util::client\_proxy::ClientProxy;
17. use tokio\_service::Service;
18. use std::str::FromStr;
19. use bytes::{BufMut, BytesMut};
20. use std::{io, str};
21. use std::net::SocketAddr;
22. // Everything about clients
23. type CollatzMessage = Message<String, Body<String, io::Error>>;
24. #[derive(Debug)]
25. pub enum CollatzInput {
26. Once(String),
27. Stream(CollatzStream),
28. }
29. pub struct CollatzProto;
30. pub struct Client {
31. inner: ClientProxy<CollatzMessage, CollatzMessage, io::Error>,
32. }
33. impl Client {
34. pub fn connect(
35. addr: &SocketAddr,
36. handle: &Handle,
37. ) -> Box<Future<Output = Client, Error = io::Error>> {
38. let ret = TcpClient::new(CollatzProto)
39. .connect(addr, handle)
40. .map(|cp| Client { inner: cp });
41. Box::new(ret)
42. }
43. }
44. impl Service for Client {
45. type Request = CollatzInput;
46. type Response = CollatzInput;
47. type Error = io::Error;
48. type Future = Box<Future<Output = Self::Response, Error =
49. io::Error>>;
50. fn call(&self, req: CollatzInput) -> Self::Future {
51. Box::new(self.inner.call(req.into()).map(CollatzInput::from))
52. }
53. }

首先加载依赖包，包括外部crates包。然后定义 CollatzProto协议相关的类型。CollatzMessage表示按协议接收到的消息；它包括一个报头和一个消息体，都均为String类型。CollatzInput表示协议的输入流，为枚举类型，有Once和Stream两个值，分别为表示非流方式接收数据和流方式接收数据。协议被定义为一个单元结构体CollatzProto。将客户端定义为一个结构体，它包括一个内部实例ClientProxy，即实际的客户端实现。服务器定义了三个类型，前两个类型Request和Response表示对整个服务器的请求和响应，第三个类型Error是对错误的响应。然后，我们为使用CollatzProto连接的Client结构实现了一个连接方法，返回一个带有连接的future。最后实现客户端Client的服务Service，其输入输出都是CollatzInput类型，因此我们必须在future使用映射将输出转换为该类型。

让我们进入服务器，它是这样的：

1. //ch4/collatz-streaming/src/lib.rs
2. // Everything about server
3. #[derive(Debug)]
4. pub struct CollatzStream {
5. inner: Body<String, io::Error>,
6. }
7. impl CollatzStream {
8. pub fn pair() -> (mpsc::Sender<Result<String, io::Error>>,
9. CollatzStream) {
10. let (tx, rx) = Body::pair();
11. (tx, CollatzStream { inner: rx })
12. }
13. }
14. impl Stream for CollatzStream {
15. type Output = String;
16. type Error = io::Error;
17. fn poll(&mut self) -> Poll<Option<String>, io::Error> {
18. self.inner.poll()
19. }
20. }
21. pub struct CollatzCodec {
22. decoding\_head: bool,
23. }
24. // Decodes a frame to a byte slice
25. impl Decoder for CollatzCodec {
26. type Output = Frame<String, String, io::Error>;
27. type Error = io::Error;
28. fn decode(&mut self, buf: &mut BytesMut) -> Result<Option<Self::Output>, io::Error> {
29. if let Some(n) = buf.as\_ref().iter().position(|b| \*b == b'\n') {
30. let line = buf.split\_to(n);
31. buf.split\_to(1);
32. return match str::from\_utf8(line.as\_ref()) {
33. Ok(s) => {
34. if s == "" {
35. let decoding\_head = self.decoding\_head;
36. self.decoding\_head = !decoding\_head;
37. if decoding\_head {
38. Ok(Some(Frame::Message {
39. message: s.to\_string(),
40. body: true,
41. }))
42. } else {
43. Ok(Some(Frame::Body { chunk: None }))
44. }
45. } else {
46. if self.decoding\_head {
47. Ok(Some(Frame::Message {
48. message: s.to\_string(),
49. body: false,
50. }))
51. } else {
52. Ok(Some(Frame::Body {
53. chunk: Some(s.to\_string()),
54. }))
55. }
56. }
57. }
58. Err(\_) => Err(io::Error::new(io::ErrorKind::Other,
59. "invalid string")),
60. };
61. }
62. Ok(None)
63. }
64. }
65. // Encodes a given byte slice to a frame
66. impl Encoder for CollatzCodec {
67. type Output = Frame<String, String, io::Error>;
68. type Error = io::Error;
69. fn encode(&mut self, msg: Self::Output, buf: &mut BytesMut) ->
70. io::Result<()> {
71. match msg {
72. Frame::Message { message, body } => {
73. buf.reserve(message.len());
74. buf.extend(message.as\_bytes());
75. }
76. Frame::Body { chunk } => {
77. if let Some(chunk) = chunk {
78. buf.reserve(chunk.len());
79. buf.extend(chunk.as\_bytes());
80. }
81. }
82. Frame::Error { error } => {
83. return Err(error);
84. }
85. }
86. buf.put\_u8(b'\n');
87. Ok(())
88. }
89. }
90. impl<T: AsyncRead + AsyncWrite + 'static> ClientProto<T> for CollatzProto {
91. type Request = String;
92. type RequestBody = String;
93. type Response = String;
94. type ResponseBody = String;
95. type Error = io::Error;
96. type Transport = Framed<T, CollatzCodec>;
97. type BindTransport = Result<Self::Transport, io::Error>;
98. fn bind\_transport(&self, io: T) -> Self::BindTransport {
99. let codec = CollatzCodec {
100. decoding\_head: true,
101. };
102. Ok(io.framed(codec))
103. }
104. }
105. impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
106. type Request = String;
107. type RequestBody = String;
108. type Response = String;
109. type ResponseBody = String;
110. type Error = io::Error;
111. type Transport = Framed<T, CollatzCodec>;
112. type BindTransport = Result<Self::Transport, io::Error>;
113. fn bind\_transport(&self, io: T) -> Self::BindTransport {
114. let codec = CollatzCodec {
115. decoding\_head: true,
116. };
117. Ok(io.framed(codec))
118. }
119. }

结构体CollatzStream定义为由字符串或错误组成的Body结构。CollatzStream的实现部分，定义了一个pair函数，能够返回流的发送者Sender；接下来，为自定义流实现Stream特质，其中的poll方法只是轮询内部Body获取数据。之后开始定义其编码解码器CollatzCodez，再实现其解码器 Decoder和编码器Encoder。在这里，我们将需要保持一种了解目前正在处理的数据帧frame的哪些部分的方法。这可以使用一个称为decoding\_head的布尔值来完成的，我们可以根据需要进行翻转。我们需要为编解码器实现解码器Decoder，这和最近几次几乎一样；请注意，我们需要追踪流和非流的情况以及之前定义的布尔值。而编码器Encoder的实现正好相反。我们还需要将协议实现绑定到编解码器；这些工作是通过实现对CollatzProto的客户端原型ClientProto和服务器原型ServerProto来完成的。在这两种情况下，我们都将布尔值设置为true，因为在收到消息后要读取的第一件事是报头。

堆栈中的最后一步是实现服务，这是通过实现CollatzService的Service特质来实现的。在其中，我们读取报头，并试图将其解析为一个u64。如果一切正常的话，继续计算该u64的collatz序列，并返回结果到一个叶子future的CollatzInput::Once。而在另一种情况下，我们遍历主体并将其打印到控制台上。最后，我们向客户端返回一个固定的字符串。具体代码如下：

1. //ch4/collatz-streaming/src/lib.rs
2. pub struct CollatzService;
3. // Given an u64, returns it's collatz sequence
4. fn get\_sequence(mut n: u64) -> Vec<u64> {
5. let mut result = vec![];
6. result.push(n);
7. while n > 1 {
8. if n % 2 == 0 {
9. n /= 2;
10. } else {
11. n = 3 \* n + 1;
12. }
13. result.push(n);
14. }
15. result
16. }
17. // Removes leading and trailing whitespaces from a given line
18. // and tries to parse it as a u64
19. fn clean\_line(line: &str) -> Result<u64, <u64 as FromStr>::Err> {
20. line.trim().parse::<u64>()
21. }
22. impl Service for CollatzService {
23. type Request = CollatzInput;
24. type Response = CollatzInput;
25. type Error = io::Error;
26. type Future = Box<Future<Output = Self::Response, Error = Self::Error>>;
27. fn call(&self, req: Self::Request) -> Self::Future {
28. match req {
29. CollatzInput::Once(line) => {
30. println!("Server got: {}", line);
31. let res = get\_sequence(clean\_line(&line).unwrap());
32. Box::new(future::done(Ok(CollatzInput::Once
33. (format!("{:?}", res)))))
34. }
35. CollatzInput::Stream(body) => {
36. let resp = body.for\_each(|line| {
37. println!("{}", line);
38. Ok(())
39. }).map(|\_| CollatzInput::Once("Foo".to\_string()));
40. Box::new(resp) as Box<Future<Output = Self::
41. Response, Error = io::Error>>
42. }
43. }
44. }
45. }

我们还编写了两个转换助手，从CollatzMessage到CollatzInput，反之亦然，通过实现From特质。与其他情况一样，我们将不得不处理这两种情况：当消息有一个消息体的情况，当它没有消息体的情况（换句话说，报头已经到达，但没有消息的其余部分）。以下是这些内容：

1. // ch4/collatz-streaming/src/lib.rs
2. // Converts a CollatzMessage to a CollatzInput
3. impl From<CollatzMessage> for CollatzInput {
4. fn from(src: CollatzMessage) -> CollatzInput {
5. match src {
6. Message::WithoutBody(line) => CollatzInput::Once(line),
7. Message::WithBody(\_, body) =>
8. CollatzInput::Stream(CollatzStream { inner: body }),
9. }
10. }
11. }
12. // Converts a CollatzInput to a Message<String, Body>
13. impl From<CollatzInput> for Message<String, Body<String, io::Error>> {
14. fn from(src: CollatzInput) -> Self {
15. match src {
16. CollatzInput::Once(line) => Message::WithoutBody(line),
17. CollatzInput::Stream(body) => {
18. let CollatzStream { inner } = body;
19. Message::WithBody("".to\_string(), inner)
20. }
21. }
22. }
23. }

在设置了服务器和客户端之后，我们将实现我们的测试作为示例，就像上次一样。以下是它们看起来的样子：测试示例

1. // ch4/collatz-streaming/examples/server.rs
2. extern crate collatz\_streaming as collatz;
3. extern crate futures;
4. extern crate tokio\_proto;
5. use tokio\_proto::TcpServer;
6. use collatz::{CollatzProto, CollatzService};
7. fn main() {
8. let addr = "0.0.0.0:9999".parse().unwrap();
9. TcpServer::new(CollatzProto, addr).serve(|| Ok(CollatzService));
10. }

客户端代码如下：

1. // ch4/collatz-streaming/examples/client.rs
2. extern crate collatz\_streaming as collatz;
3. extern crate futures;
4. extern crate tokio\_core;
5. extern crate tokio\_service;
6. use collatz::{CollatzInput, CollatzStream};
7. use std::thread;
8. use futures::Sink;
9. use futures::Future;
10. use tokio\_core::reactor::Core;
11. use tokio\_service::Service;
12. pub fn main() {
13. let addr = "127.0.0.1:9999".parse().unwrap();
14. let mut core = Core::new().unwrap();
15. let handle = core.handle();
16. // Run the client in the event loop
17. core.run(
18. collatz::Client::connect(&addr, &handle)
19. .and\_then(|client| {
20. client.call(CollatzInput::Once("10".to\_string()))
21. .and\_then(move |response| {
22. println!("Response: {:?}", response);
23. let (mut tx, rx) = CollatzStream::pair();
24. thread::spawn(move || {
25. for msg in &["Hello", "world", "!"] {
26. tx =
27. tx.send(Ok(msg.to\_string()))
28. .wait().unwrap();
29. }
30. });
31. client.call(CollatzInput::Stream(rx))
32. })
33. .and\_then(|response| {
34. println!("Response: {:?}", response);
35. Ok(())
36. })
37. })
38. ).unwrap();
39. }

我们使用前面定义的connect方法来在已知地址和端口上设置到服务器的连接。我们使用and\_then组合算子向服务器发送一个固定的字符串，然后打印响应。在这一点上，我们已经传送了我们的报头，接着我们继续传送消息体。这是通过将流分成两半并使用发送方发送多个字符串来完成的。最后一个组合算子打印响应并处理future。以前的所有内容都是在一个事件循环中运行的。

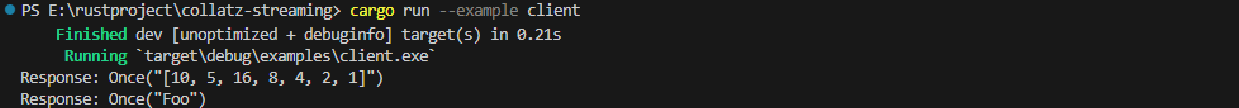
这是服务器的会话情况：

![文本

描述已自动生成

**$ cargo run --example server**

这就是客户机的情况：



**$ cargo run --example client**

正如预期的那样，服务器在获得请求的实际消息体之前处理了报头（我们知道这是因为我们在发送报头之后发送了消息体）。除了我们在这里讨论的内容之外，tokio还支持许多其他特性。例如，为了在构建BindTransport future之前交换设置消息，您可以通过更改服务器原型ServerProto和客户端原型ClientProto的实现来实现协议握手。这一点非常重要，因为许多网络协议都需要某种形式的握手来设置一个共享状态。注意，协议完全有可能是流和流水线，或者流和多路复用。对于这些，实现需要分别替换来自streaming::pipeline或streaming::multiplex命名空间的特质。

## 小结

这一章介绍了future和tokio，它们的生态系统提供了强大的原语，可以广泛应用于包括网络软件在内的各种应用程序，是Rust生态系统中的两个重要部分。就future本身而言，其Futures可以用来建模任何速度和/或依赖于外部资源的计算。与tokio相结合，它可以用来建模流水线化或多路复用的复杂协议行为等等。

使用这些方法的一些主要缺点是围绕着缺乏适当的文档和示例。此外，来自这些应用程序的错误消息通常是高度模板化的，因此是冗长。因为Rust编译器本身并不知道抽象本身，所以它经常抱怨类型不匹配，并且由用户通过深度嵌套的类型进行推理。在futures的某个地方，实现一个编译器插件可能是有意义的，它可以将这些错误转换为更直观的形式。

# TCP和UDP应用编程

本章要点：理解标准库的Std∷net命名空间的结构，了解其中涉及网络编程的函数，掌握通过crates单元包实现网络编程的方法，掌握TCP客户端和服务器端程序的编写方法，掌握UDP客户端和服务器端程序的编写方法。

## 引言

作为一种系统编程语言，Rust标准库支持与网络堆栈进行交互。所有与网络相关的函数都位于std::net命名空间中；对套接字的读写操作也使用来自std::io的Read和Write特质。这里的一些最重要的结构是IpAddr，它代表一个通用的IP地址，可以v4或v6，SocketAddr，代表一个通用的套接字地址（一个IP和主机上的端口），TcpListener和TcpStream用于TCP通信，UdpSocket用于UDP通信，等等。目前，标准库不提供任何APIs来处理底层的网络堆栈。虽然这种情况在以后可能会改变，但一些crates单元包填补了这一空白。其中最重要的是libpnet，它为底层网络提供了一组APIs。其他一些重要的网络crate单元包是net2和socket2。这些都是为了作为可能被转移到标准库中的APIs的孵化器。它被认为是足够有用和稳定的，这里的一些功能被移植到Rust core repo中。当然，这并不是在所有情况下都按计划奏效。总的来说，社区现在建议使用tokio生态系统的crates单元包来编写不需要对套接字进行细粒度语义控制的高性能网络应用程序。

本章首先介绍std::net命名空间的结构及其重要的网络函数，然后讨论crates单元包中网络编程相关的重要API，比如net2、ipnetwork和libpnet，上述介绍的内容会在以简单的多线程TCP客户端和服务器程序设计以及多线程UDP客户端和服务器程序设计的这两个案例中体现。

## std∷net命名空间和crates单元包程序

### std∷net命名空间

Rust标准库提供了std∷net命名空间。该命名空间封装了整个TCP/IP协议栈，包含了几乎所有与网络相关的函数。主要函数如表5.1所示。

|  |  |
| --- | --- |
| 5.1 std::net命名空间常用的函数或特质 | |
| 函数或特质 | 功能 |
| TcpListener和TcpStream | 提供TCP通信功能 |
| UdpSocket | 提供UDP通信功能 |
| IpAddress | 代表IPv4或IPv6的IP地址；IP4Addr和IPv6Addr分别是IPv4或IPv6地址 |
| SocketAddr | 代表IPv4或IPv6的套接字地址，即IP地址和端口号的组合；SocketAddrV4和SocketAddrV6分别是IPv4或IPv6套接字地址。 |
| ToSocketAddr | 可以转换或解析为一个或多个SocketAddr值的对象的trait特质，在与TcpListener、TcpStream或UdpSocket等网络对象交互时用于通用地址解析的trait特质 |

Rust标准库中，IP地址用IpAddr类型表示，该类型是一个枚举类型，包括IP4Addr和IPv6Addr分别表示v4地址和v6地址。在使用这些类型或函数时，用特性标志ip加以声明，如示例5-1第2行中feature(ip)的声明所示，且必须包含在crate包根目录中。另一个网络编程相关的概念是SocketAddr，表示一个IP地址和一个端口号的组合。详细介绍如下：

**1、IpAddr类型**

IpAddr封装了IP地址，包括IP v4地址和IP v6地址：

pub enum IpAddr {

V4(Ipv4Addr),

V6(Ipv6Addr),

}

IpAddr实现了FromStr，可直接将代表IP地址的字符串解析为IpAddr：

let localhsot: IpAddr = "127.0.0.1".parse().unwrap();

例如：

use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};

let localhost = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));

assert\_eq!("127.0.0.1".parse(), Ok(localhost));

IpAddr还有一些方法，主要是一些布尔判断方法：

* is\_ipv4()：是否是一个ipv4地址
* is\_ipv6()：是否是一个ipv6地址
* is\_loopack()：是否是一个loopback地址
* is\_multicast()：是否是一个多播地址
* is\_unspecified()：是否是一个0.0.0.0地址

IpAddr封装了ip v4地址或ip v6地址，以代表ip v4地址的Ipv4Addr为例。可使用new()并提供4个u8参数来创建ip v4地址：

use std::net::Ipv4Addr;

let localhost = Ipv4Addr::new(127, 0, 0, 1);

Ipv4Addr实现了FromStr，也可以很方便地直接将字符串解析为ip地址：

let localhost = "127.0.0.1".parse().unwrap();

可使用octets()将一个IP地址转换为u8数组，即new()的反向操作：

use std::net::Ipv4Addr;

let addr = Ipv4Addr::new(127, 0, 0, 1);

assert\_eq!(addr.octets(), [127, 0, 0, 1]);

Ipv4Addr还有其它一些方法，多数都是布尔判断方法:

* is\_broadcast(): 是否是广播地址(255.255.255.255)
* is\_multicast(): 是否是多播地址(224.0.0.0/4)
* is\_private(): 是否是私有地址(10.0.0.0/8、172.16.0.0/12、192.168.0.0/16)
* is\_link\_local(): 是否是链路本地地址(169.254.0.0/16)
* is\_loopback(): 是否是环回地址(127.0.0.0/8)
* is\_unspecified(): 是否是0.0.0.0

此外，可直接对地址进行大小比较和等值比较。

**2、SocketAddr**

SocketAddr代表包含了IP地址和端口号的套接字地址，它封装了ipv4套接字地址和ipv6套接字地址：

pub enum SocketAddr {

V4(SocketAddrV4),

V6(SocketAddrV6),

}

SocketAddr实现了FromStr，因此可直接将代表套接字地址的字符串解析为SocketAddr:

use std::net::{IpAddr, Ipv4Addr, SocketAddr};

let socket: SocketAddr = "127.0.0.1:8080".parse().unwrap();

SocketAddr自身也提供了new()方法，需提供IpAddr和端口号(u16)作为参数：

use std::net::{IpAddr, Ipv4Addr, SocketAddr};

let ip = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));

let socket = SocketAddr::new(ip, 8080);

此外，还有以下几个方法：

* is\_ipv4(): 是否是ip v4套接字地址
* is\_ipv6(): 是否是ip v6套接字地址
* ip(): 返回IP地址
* port(): 返回端口号
* set\_ip(): 修改IP地址
* set\_port(): 修改端口号

SocketAddr封装的代表ipv4套接字的SocketAddrV4也很简单直接，可由代表ipv4套接字的字符串解析得到，也可由new()方法创建，其也包括ip()、port()、set\_ip()以及set\_port()这几个方法。

use std::net::{Ipv4Addr, SocketAddrV4};

let socket = SocketAddrV4::new(Ipv4Addr::new(127, 0, 0, 1), 8080);

assert\_eq!("127.0.0.1:8080".parse(), Ok(socket));

assert\_eq!(socket.ip(), &Ipv4Addr::new(127, 0, 0, 1));

assert\_eq!(socket.port(), 8080);

示例5-1：IpAddr和SocketAddr的用法。

1. // chapter5/ch5-1-ip-socket-addr.rs
2. #feature(ip)]
3. use std::net::{IpAddr, SocketAddr};
4. fn main() {
5. // construct an IpAddr from a string and check it
6. // represents the loopback address
7. let local: IpAddr = "127.0.0.1".parse().unwrap();
8. assert!(local.is\_loopback());
9. // construct a globally routable IPv6 address from individual
10. octets
11. // and assert it is classified correctly
12. let global: IpAddr = IpAddr::from([0, 0, 0x1c9, 0, 0, 0xafc8, 0,
13. 0x1]);
14. assert!(global.is\_global());
15. // construct a SocketAddr from a string an assert that the
16. underlying
17. // IP is a IPv4 address
18. let local\_sa: SocketAddr = "127.0.0.1:80".parse().unwrap();
19. assert!(local\_sa.is\_ipv4());
20. // construct a SocketAddr from a IPv6 address and a port, assert
21. that
22. // the underlying address is indeed IPv6
23. let global\_sa = SocketAddr::new(global, 80u16);
24. assert!(global\_sa.is\_ipv6());
25. }

网络编程中另一个常见功能是DNS查找，即指定一个主机名查找对应IP地址。之前Rust可以使用lookup\_host函数来实现这一功能，目前已经弃用并移除了。这里使用ToSocketAddrs trait 来执行域名解析。示例5-2代码如下：

1. // chapter5/ch5-2-lookup-host.rs
2. use std::env;
3. use std::net::{SocketAddr, ToSocketAddrs};
4. fn main() {
5. // 收集命令行参数
6. let args: Vec<String> = env::args().collect();
7. // 检查是否只提供了一个参数（主机名）
8. if args.len() != 2 {
9. eprintln!("请只提供一个主机名");
10. std::process::exit(1);
11. }
12. // 从参数中提取主机名
13. let host = &args[1];
14. // 执行域名解析
15. match (host.as\_str(), 0).to\_socket\_addrs() {
16. Ok(addresses) => {
17. // 遍历解析后的地址
18. for addr in addresses {
19. match addr {
20. // 处理IPv6地址
21. SocketAddr::V6(ipv6\_addr) => {
22. println!("{}", ipv6\_addr.ip());
23. }
24. // 处理IPv4地址
25. SocketAddr::V4(ipv4\_addr) => {
26. println!("{}", ipv4\_addr.ip());
27. }
28. }
29. }
30. }
31. Err(err) => {
32. // 如果解析失败则打印错误信息
33. println!("域名解析失败: {}", err);
34. std::process::exit(1);
35. }
36. }
37. }

运行结果如下：

![文本

描述已自动生成

### Crates网络编程

上面Rust标准库中给出的都是常见的基本网络相关功能，难以处理IP网络（CIDR）。这里可以使用ipnetwork crate包处理IP网络的相关功能。见示例5-3如下。

创建项目：**$ cargo new --bin ch5-3-ipnetwork-example**



Cargo.toml配置：

1. [package]
2. name = "ch5-3-ipnetwork-example"
3. version = "0.1.0"
4. [dependencies]
5. ipnetwork = "0.12.7"

主文件代码如下：

1. // chapter5/ch5-3-ipnetwork-example/src/main.rs
2. extern crate ipnetwork;
3. use std::net::Ipv4Addr;
4. use ipnetwork::{IpNetwork, Ipv4Network, Ipv6Network};
5. fn main() {
6. let net = IpNetwork::new("192.168.122.0".parse().unwrap(), 22)
7. .expect("Could not construct a network");
8. let str\_net: IpNetwork = "192.168.122.0/22".parse().unwrap();
9. assert!(net == str\_net);
10. assert!(net.is\_ipv4());
11. let net4: Ipv4Network = "192.168.121.0/22".parse().unwrap();
12. assert!(net4.size() == 2u64.pow(32 - 22));
13. assert!(net4.contains(Ipv4Addr::new(192, 168, 121, 3)));
14. let \_net6: Ipv6Network = "2001:db8::0/96".parse().unwrap();
15. for addr in net4.iter().take(10) {
16. println!("{}", addr);
17. }
18. }

主函数的前两行分别使用两种方法构建IpNetwork实例，使用构造函数或借助解析字符串。接下来的assert将确保二者效果相同的，以及所建网络为IPv4网络。然后创建Ipv4网络对象Ipv4Network，其网络规模符合2^（32-前缀）。下一行的assert中contains方法用来检测指定的IP地址是否为IPv4地址。然后再创建一个Ipv6Network，由于这些类型都实现了迭代器协议，我们可以在网络上迭代并在for循环中打印单个地址。执行及输出结果如下：

文本

描述已自动生成

此外，Rust标准库还缺乏对套接字和连接的细粒度控制，比如设置SO\_REUSEADDR的能力。主要原因是Rust社区未能达成共识。这里可以引入mio库，它提供了一种可以替代基于线程并发的方法。mio本质上运行一个多方注册的事件循环。当出现一个事件时，每个监听器都会被触发，并选择如何处理该事件。示例5-4如下：

创建项目：**$ cargo new --bin ch5-4-mio-example**



Cargo.toml配置如下：

1. [package]
2. name = "ch5-4-mio-example"
3. version = "0.1.0"
4. [dependencies]
5. mio = "0.6.11"

主文件代码如下：

1. // chapter5/ch5-4-mio-example/src/main.rs
2. extern crate mio;
3. use mio::\*;
4. use mio::tcp::TcpListener;
5. use std::net::SocketAddr;
6. use std::env;
7. // This will be later used to identify the server on the event loop
8. const SERVER: Token = Token(0);
9. // Represents a simple TCP server using mio
10. struct TCPServer {
11. address: SocketAddr,
12. }
13. // Implementation for the TCP server
14. impl TCPServer {
15. fn new(port: u32) -> Self {
16. let address = format!("0.0.0.0:{}", port)
17. .parse::<SocketAddr>().unwrap();
18. TCPServer {
19. address,
20. }
21. }
22. // Actually binds the server to a given address and runs it
23. // This function also sets up the event loop that dispatches
24. // events. Later, we use a match on the token on the event
25. // to determine if the event is for the server.
26. fn run(&mut self) {
27. let server = TcpListener::bind(&self.address)
28. .expect("Could not bind to port");
29. let poll = Poll::new().unwrap();
30. poll.register(&server,
31. SERVER,
32. Ready::readable(),
33. PollOpt::edge()).unwrap();
34. let mut events = Events::with\_capacity(1024);
35. loop {
36. poll.poll(&mut events, None).unwrap();
37. for event in events.iter() {
38. match event.token() {
39. SERVER => {
40. let (\_stream, remote) =
41. server.accept().unwrap();
42. println!("Connection from {}", remote);
43. }
44. \_ => {
45. unreachable!();
46. }
47. }
48. }
49. }
50. }
51. }
52. fn main() {
53. let args: Vec<String> = env::args().collect();
54. if args.len() != 2 {
55. eprintln!("Please provide only one port number as argument");
56. std::process::exit(1);
57. }
58. let mut server = TCPServer::new(args[1].parse::<u32>()
59. .expect("Could not parse as u32"));
60. server.run();
61. }

与我们之前的示例不同，这是一个TCP服务器，它只打印客户端的源IP地址和端口。在mio中，事件循环上的每个监听器都被分配了一个令牌，然后可以用来在传递事件时区分这些监听器。我们在其构造函数中为服务器（TCPServer）定义了一个结构体，并bind绑定到所有本地地址并返回该结构体的一个实例。该结构体的run方法将套接字绑定到给定的套接字地址；然后，它使用Poll结构体来实例化事件循环。

然后，它在实例上注册了一个令牌注册服务器套接字。我们还指出，当事件准备好读取或写入时，我们应该收到通知。最后，我们指出我们只需要边缘触发的事件，这意味着事件在接收时应该完全消耗，否则对同一令牌的后续调用可能会阻止它。然后，我们为我们的事件设置了一个空容器。在完成所有样板之后，我们进入一个无限循环，并使用刚刚创建的事件容器开始轮询。我们遍历事件列表，如果事件的任何令牌与服务器的令牌匹配，我们就知道它是针对服务器的。然后，我们可以接受连接，并打印远端的信息。然后我们回到下一个活动中，以此类推。在我们的main函数中，我们首先处理CLI参数，以确保我们传递了一个端口号作为一个整数。然后，我们实例化服务器并对其调用run方法。

下面是一个在两个客户端连接到服务器时运行服务器的示例会话。请注意，nc或早期的TCP客户端可以用来连接到这个服务器：

执行及输出结果



这里讨论的标准库和板条箱中缺少的其他一些主要的东西是与物理网络设备一起工作的能力，一个用来编写和解析数据包的更好的API，等等。一个有助于处理低级网络相关事物的crate单元包。

示例5-5. 用crate实现一个小型数据包存储器。

创建项目：**$ cargo new --bin pnet-example**



配置文件：

**$ cat Cargo.toml**

1. [package]
2. name = "ch5-5-pnet-example"
3. version = "0.1.0"
4. [dependencies]
5. pnet = "0.20.0"

主文件代码：

1. // chapter5/ch5-5-pnet-example/src/main.rs
2. extern crate pnet;
3. use pnet::datalink::{self, NetworkInterface};
4. use pnet::datalink::Channel::Ethernet;
5. use pnet::packet::ethernet::{EtherTypes, EthernetPacket};
6. use pnet::packet::ipv4::Ipv4Packet;
7. use pnet::packet::tcp::TcpPacket;
8. use pnet::packet::ip::IpNextHeaderProtocols;
9. use pnet::packet::Packet;
10. use std::env;
11. // Handles a single ethernet packet
12. fn handle\_packet(ethernet: &EthernetPacket) {
13. match ethernet.get\_ethertype() {
14. EtherTypes::Ipv4 => {
15. let header = Ipv4Packet::new(ethernet.payload());
16. if let Some(header) = header {
17. match header.get\_next\_level\_protocol() {
18. IpNextHeaderProtocols::Tcp => {
19. let tcp = TcpPacket::new(header.payload());
20. if let Some(tcp) = tcp {
21. println!(
22. "Got a TCP packet {}:{} to {}:{}",
23. header.get\_source(),
24. tcp.get\_source(),
25. header.get\_destination(),
26. tcp.get\_destination()
27. );
28. }
29. }
30. \_ => println!("Ignoring non TCP packet"),
31. }
32. }
33. }
34. \_ => println!("Ignoring non IPv4 packet"),
35. }
36. }
37. fn main() {
38. let interface\_name = env::args().nth(1).unwrap();
39. // Get all interfaces
40. let interfaces = datalink::interfaces();
41. // Filter the list to find the given interface name
42. let interface = interfaces
43. .into\_iter()
44. .filter(|iface: &NetworkInterface| iface.name == interface\_name)
45. .next()
46. .expect("Error getting interface");
47. let (\_tx, mut rx) = match datalink::channel(&interface,
48. Default::default()) {
49. Ok(Ethernet(tx, rx)) => (tx, rx),
50. Ok(\_) => panic!("Unhandled channel type"),
51. Err(e) => {
52. panic!(
53. "An error occurred when creating the datalink channel:
54. {}",e
55. )
56. }
57. };
58. // Loop over packets arriving on the given interface
59. loop {
60. match rx.next() {
61. Ok(packet) => {
62. let packet = EthernetPacket::new(packet).unwrap();
63. handle\_packet(&packet);
64. }
65. Err(e) => {
66. panic!("An error occurred while reading: {}", e);
67. }
68. }
69. }
70. }

然后我们将pnet作为依赖项添加，将其固定到一个特定版本（当前可用的最新版本）。然后像往常一样，我们从声明pnet为一个外部crate开始。然后我们导入一堆稍后将使用的东西。我们将我们应该嗅探的接口的名称作为一个CLI参数。datalink::interfaces()为我们提供当前主机中所有可用接口的列表，我们根据给出的接口的名称过滤该列表。如果我们没有找到匹配项，那么我们将抛出一个错误并退出。datalink::channel()调用给了我们一个发送和接收数据包的通道。在这种情况下，我们并不关心发送端，因为我们只是对嗅探数据包感兴趣。我们匹配返回的通道类型，以确保我们只使用以太网工作。通道的接收端rx给了我们一个迭代器，它在每个next()调用中产生数据包。

然后将数据包传递给handle\_packet函数，该函数提取相关信息并打印这些信息。对于这个玩具示例，我们将只处理基于ipv4的TCP数据包。一个真正的网络显然会得到具有UDP和TCP的IPv6和ICMP数据包。所有这些组合在这里都将被忽略。

在handle\_packet函数中，我们匹配数据包的其他类型，以确保我们只处理IPv4数据包。由于以太网数据包的整个有效负载是IP数据包，所以我们从有效负载构建一个IP数据包。get\_next\_level\_protocol()调用返回传输协议，如果它与TCP匹配，我们将从前一层的有效负载构建一个TCP数据包。此时，我们就可以从TCP数据包中打印出源端口和目标端口。源IP和目标IP将在封装的IP数据包中。我们看到了标准库中实现的与dns相关的功能是有限的，可以使用crate库来实现dns的相关功能。

例5-6. 使用crate查询给定名称

创建项目：**$ cargo new --bin ch5-6-trust-dns-example**



配置文件：

（需要在src下面创建一个空的lib.rs文件）

1. [package]
2. name = "ch5-6-trust-dns-example"
3. version = "0.1.0"
4. [dependencies]
5. trust-dns-resolver = "0.18.0"
6. trust-dns = "0.19.0"
7. [lib]
8. name = "trust\_dns"
9. path = "src/lib.rs"
10. crate-type = ["lib"]

主文件代码：

1. // chapter5/ch5-6-trust-dns-example/src/main.rs
2. extern crate trust\_dns\_resolver;
3. extern crate trust\_dns;
4. use std::env;
5. use trust\_dns\_resolver::Resolver;
6. use trust\_dns\_resolver::config::\*;
7. use trust\_dns\_resolver::proto::rr::RecordType;
8. use std::str::FromStr;
9. fn main() {
10. let args: Vec<String> = env::args().collect();
11. if args.len() != 2 {
12. eprintln!("Please provide a name to query");
13. std::process::exit(1);
14. }
15. let resolver = Resolver::new(ResolverConfig::default(), ResolverOpts::default()).unwrap();
16. let query = format!("{}.", args[1]);
17. let response = resolver.lookup\_ip(query.as\_str());
18. println!("Using the synchronous resolver");
19. for ans in response.iter() {
20. println!("{:?}", ans);
21. }
22. println!("Using the system resolver");
23. let system\_resolver = Resolver::from\_system\_conf().unwrap();
24. let system\_response = system\_resolver.lookup\_ip(query.as\_str());
25. for ans in system\_response.iter() {
26. println!("{:?}", ans);
27. }
28. // Specify the record type as a string
29. let record\_type\_str = "NS";
30. let record\_type = RecordType::from\_str(record\_type\_str).expect("Failed to parse RecordType");
31. let ns = resolver.lookup(query.as\_str(), record\_type);
32. println!("NS records using the synchronous resolver");
33. for ans in ns.iter() {
34. println!("{:?}", ans);
35. }
36. }

我们建立了所有必需的进口和extern crate单元包声明。在这里，我们希望将名称解析为CLI参数，如果一切顺利，它应该在args[1]中。这个crate支持两种类型的同步DNS解析器。Resolver::new创建了一个同步解析器，并且使用默认选项，它将使用谷歌的公共DNS作为上游服务器。Resolver::from\_system\_conf使用系统的resolv.conf配置创建一个同步解析器。因此，这第二个选项仅在Unix系统上可用。在将查询传递给解析器之前，我们通过附加a.到名称来将其格式化为FQDN。使用format！“宏”。我们使用lookup\_ip函数传递查询，然后它在DNS问题的答案上返回一个迭代器。一旦我们得到了它，我们就可以遍历它并打印出每个答案。顾名思义，lookup\_ip函数只查找A和AAAA的记录。有一个更通用的查找函数，它可以接受一个记录类型来进行查询。在最后一步中，我们希望获得给定名称的所有NS记录。一旦我们得到了一个答案，我们就会循环检查它并打印出结果。

运行结果如下：**$ cargo run google.com**

电脑的屏幕截图

描述已自动生成

## TCP编程的基本概念

### 编程特点

TCP协议是为了在不可靠的互联网上提供可靠的端到端字节流而专门设计的，是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC793定义。TCP协议提供可靠的通信服务，其通信特点包括：面向连接、全双工通信、字节流传输、使用缓冲区、可靠传输、面向字节流。

### TcpListener 和 TcpStream

在主流操作系统中，网络堆栈的传输层为开发者提供了一系列名为套接字（socket）的API。套接字包括一组接口，用于在两个进程之间建立通信连接。套接字允许在本地或远程两个进程之间传递数据，而无需开发者了解底层的网络协议，这极大地降低了网络编程的复杂性。Rust的标准库和tokio库都提供了net模块，用于封装整个TCP/IP协议栈的API接口，包括套接字接口。对于TCP通信，存在两种主要的类型：TcpStream和TcpListener。

Rust标准库中提供了std::net，为整个TCP/IP协议栈的使用提供了封装。然而std::net是同步执行的。如果需要构建一个高性能的异步网络，则需要使用tokio库的tokio::net。

**（1）TcpListener**

pub struct TcpListener( );

TCP 套接字服务器，侦听连接。

通过将 TcpListener 绑定到套接字地址来创建 TcpListener 之后，它会侦听传入的 TCP 连接。 可以通过调用 accept 或在 incoming 返回的 Incoming 迭代器上进行迭代来接受它们。

丢弃该值时，套接字将关闭。

传输控制协议在 IETF RFC 793 中指定。

**（2）TcpStream**

pub struct TcpStream(\_);

本地套接字和远程套接字之间的 TCP 流。

在通过 connect 到远程主机或 accept 在 TcpListener 上创建连接来创建 TcpStream 后，数据可以由 读取 和 写入 传输到该 TcpStream。

丢弃该值时，连接将关闭。也可以使用 shutdown 方法单独关闭连接的读取和写入部分。

传输控制协议在 IETF RFC 793 中指定。

## 示例：TCP协议

大多数网络示例都从回显服务器开始。所以，让我们继续在Rust中编写一个基本的回显服务器（Echo server），看看所有的部分是如何组合在一起的。我们将使用来自标准库的线程模型来并行处理多个客户端。

例5-7. 使用TCP协议的网络应用通信程序。

服务器代码如下：

1. // chapter5/ch5-7-tcp-echo-server.rs
2. use std::net::{TcpListener, TcpStream};
3. use std::thread;
4. use std::io::{Read, Write, Error};
5. // Handles a single client
6. fn handle\_client(mut stream: TcpStream) -> Result<(), Error> {
7. println!("Incoming connection from: {}", stream.peer\_addr()?);
8. let mut buf = [0; 512];
9. loop {
10. let bytes\_read = stream.read(&mut buf)?;
11. if bytes\_read == 0 { return Ok(()); }
12. stream.write(&buf[..bytes\_read])?;
13. }
14. }
15. fn main() {
16. let listener = TcpListener::bind("0.0.0.0:8888")
17. .expect("Could not bind");
18. for stream in listener.incoming() {
19. match stream {
20. Err(e) => { eprintln!("failed: {}", e) }
21. Ok(stream) => {
22. thread::spawn(move || {
23. handle\_client(stream)
24. .unwrap\_or\_else(|error| eprintln!("{:?}", error));
25. });
26. }
27. }
28. }
29. }

在主函数中，创建一个新的监听器TcpListener，这是一个用于正在监听来自客户端的传入连接的TCP套接字。在示例5-7中，本地地址和端口是固定的。请注意，如果将本地地址设置为0.0.0.0，则内核会将该套接字绑定到本机上的所有可用端口。

这里需要设置一个公共端口，客户端是从该端口连接到服务器。在实际的应用程序中，可以从控制台命令或配置文件进行设置。调用bind绑定本地地址和端口来创建一个本地监听套接字。在指定IP地址后，如果主机端口为8888时，套接字将会绑定到主机上的所有可用端口。此时，任何连接到此主机的客户端都能与该主机进行通信。正常情况下，expect函数会返回监听器，否则，它就返回出错消息并导致程序崩溃（panic）。如果出现绑定端口失败导致程序崩溃的情况也是可以的，因为此时服务器也不可能继续工作。监听器listener的incoming方法会返回一个已连接到服务器的流之上的迭代器。循环检查其中是否遇到了错误。在这种情况下，我们可以打印错误信息并切换到下一个已连接的客户端。请注意，在这种情况下，引发panicking是不合适的，因为即使某些客户端由于某种原因遇到错误，服务器仍然可以继续正常工作。

读取数据的操作相对比较复杂的。因为如果每个客户端都要在无限循环地执行读取数据的任务，那么主线程在运行这个无限循环时将出现阻塞，进而导致其他客户端无法连接服务器。所以这种方式是不合适的。因此，我们可以创建一个工作线程来专门处理每个客户端连接。每个流读取和写回的操作被封装在函数handle\_client中。每个线程都会接收到一个调用此函数的闭包。此闭包必须是一个move闭包，因为它必须从封闭范围中读取一个变量（stream）。在该函数中，我们打印远程主机的地址和端口，再定义一个缓冲区来临时保存数据并确保缓冲区归零，然后在无限循环中读取流中的所有数据。流的read方法的返回值是已读取数据的长度。如果读到流的末尾或者给定的缓冲区长度为零是返回值为零。因此，当read方法返回零时，就跳出循环（和函数）。此时，返回Ok()。然后，我们使用切片语法将相同的数据写入流。注意，可以已经使用宏eprintln！输出错误。宏eprintln！是用来将给定的字符串写入一个标准错误的。

在读写流时，使用操作符来处理这些调用中出现的错误。如果一切正常，该操作符将结果展开为Ok；否则，它会提前将错误返回到调用函数。鉴于此设置，函数的返回类型可以是空类型，用来处理成功的情况，或是io::Error类型，用来处理错误的情况。

下面实现一个简单的TCP客户端。客户端将从stdin中读取输入的字符串，并将其发送到服务器。在收到服务器的回复消息后，会通过stdout打印回复消息。为了方便演示，假设客户机和服务器运行在同一台物理主机上，这里，设置服务器地址为127.0.0.1：

客户端代码如下：

1. // chapter5/ch5-7-tcp-client.rs
2. use std::net::TcpStream;
3. use std::str;
4. use std::io::{self, BufRead, BufReader, Write};
5. fn main() {
6. let mut stream = TcpStream::connect("127.0.0.1:8888")
7. .expect("Could not connect to server");
8. loop {
9. let mut input = String::new();
10. let mut buffer: Vec<u8> = Vec::new();
11. io::stdin().read\_line(&mut input)
12. .expect("Failed to read from stdin");
13. stream.write(input.as\_bytes())
14. .expect("Failed to write to server");
15. let mut reader = BufReader::new(&stream);
16. reader.read\_until(b'\n', &mut buffer)
17. .expect("Could not read into buffer");
18. print!("{}", str::from\_utf8(&buffer)
19. .expect("Could not write buffer as string"));
20. }
21. }

本例中使用TcpStream::connect建立一个到服务器的连接，这里将远程端点地址作为字符串参数。客户端需要知道所要连接的远程主机IP地址和端口。如果连接失败，我们将中止程序。然后在无限循环中初始化一个空字符串用来读取本地用户的输入并且用一个Vec<u8>向量来存储从服务器接收的响应信息。由于Rust中的向量会根据需要增加，因此不需要在每次迭代中手动对数据分块。read\_line函数从标准输入中读取一行，并将其存储在变量input中。然后，将其作为一个字节流写入到连接中。此时，如果执行正常，服务器就应该已经发回了一个响应。我们使用BufReader来处理内部分组数据，这样不需要更多的系统调用，使得读取效率更高。使用read\_until方法来读取缓冲区中的数据。最后，使用from\_utf8方法将缓冲区内容转换为一个字符串并打印出来。

运行客户端，结果如下：**$ rustc ch5-7-tcp-client.rs && ./ch5-7-tcp-client**

形状

中度可信度描述已自动生成

现实应用的情况要比上面的例子更复杂。比如服务器在返回响应之前，可能需要一些时间来处理输入。为了更贴近现实应用，可以通过随机睡眠时间在handle\_client函数中模拟这一过程，见例5-8，主函数不变。

例5-8. 随机睡眠时间的使用。

创建项目：**$ cargo new --bin tcp-echo-random**



配置 Cargo.toml文件：

1. [package]
2. name = "ch5-8-tcp-echo-random"
3. version = "0.1.0"
4. [dependencies]
5. rand = "0.3.17"

修改handle\_client函数，在发送响应之前加上随机延迟睡眠：修改后主文件代码如下：

1. // chapter5/ch5-8-tcp-echo-random/src/main.rs
2. extern crate rand;
3. use std::net::{TcpListener, TcpStream};
4. use std::thread;
5. use rand::{thread\_rng, Rng};
6. use std::time::Duration;
7. use std::io::{Read, Write, Error};
8. fn handle\_client(mut stream: TcpStream) -> Result<(), Error> {
9. let mut buf = [0; 512];
10. loop {
11. let bytes\_read = stream.read(&mut buf)?;
12. if bytes\_read == 0 { return Ok(()) }
13. let sleep = Duration::from\_secs(\*thread\_rng()
14. .choose(&[0, 1, 2, 3, 4, 5])
15. .unwrap());
16. println!("Sleeping for {:?} before replying", sleep);
17. std::thread::sleep(sleep);
18. stream.write(&buf[..bytes\_read])?;
19. }
20. }
21. fn main() {
22. let listener = TcpListener::bind("127.0.0.1:8888").expect("Could not bind");
23. for stream in listener.incoming() {
24. match stream {
25. Err(e) => eprintln!("failed: {}", e),
26. Ok(stream) => {
27. thread::spawn(move || {
28. handle\_client(stream).unwrap\_or\_else(|error|
29. eprintln!("{:?}", error));
30. });
31. }
32. }
33. }
34. }

在主文件中，首先将rand crate包作为外部依赖extern crate声明，再使用thread\_rng函数随机选择一个介于0到5之间的整数，然后使用std::thread::sleep在该时间内睡眠。因为服务器的回复操作不会立即完成，我们在客户端设置读取和连接超时。

客户端代码如下：

1. // chapter5/ch5-8-tcp-client-timeout.rs
2. use std::net::TcpStream;
3. use std::str;
4. use std::io::{self, BufRead, BufReader, Write};
5. use std::time::Duration;
6. use std::net::SocketAddr;
7. fn main() {
8. let remote: SocketAddr = "127.0.0.1:8888".parse().unwrap();
9. let mut stream = TcpStream::connect\_timeout(&remote,
10. Duration::from\_secs(1))
11. .expect("Could not connect to server");
12. stream.set\_read\_timeout(Some(Duration::from\_secs(3)))
13. .expect("Could not set a read timeout");
14. loop {
15. let mut input = String::new();
16. let mut buffer: Vec<u8> = Vec::new();
17. io::stdin().read\_line(&mut input).expect("Failed to read from
18. stdin");
19. stream.write(input.as\_bytes()).expect("Failed to write to
20. server");
21. let mut reader = BufReader::new(&stream);
22. reader.read\_until(b'\n', &mut buffer)
23. .expect("Could not read into buffer");
24. print!("{}", str::from\_utf8(&buffer)
25. .expect("Could not write buffer as string"));
26. }
27. }

这里，使用set\_read\_timeout将超时时间设置为3秒。如果服务器休眠时间超过3秒，客户端将中止连接。该函数需要设置Option<Duration>来指定一个为none的持续时间Duration，这里Duration是作为Some的参数的执行的，通过Some的返回值才传递到set\_read\_timeout函数。运行代码，打开服务器端和客户端两个会话，结果如下，服务器打印出其所接收的客户端对应的睡眠的时间：

服务器端执行及输出结果如下：**$ cargo run**

文本

描述已自动生成

客户端执行结果如下：**$ rustc ch5-8-tcp-client-timeout.rs && ./ch5-8-tcp-client-timeout**

文本

描述已自动生成

从上面的结果可见，对于前三个消息，服务器选择了小于3秒的延迟。客户端在3秒钟内得到了响应，并且没有中止连接。对于最后一条消息，延迟为5秒，这导致客户端中止读取。

## UDP编程的基本概念

### 编程特点

用户数据报协议（UDP，User Datagram Protocol）是一个无连接的传输协议，提供面向事务的简单不可靠信息传送服务。UDP 为应用程序提供了一种无需建立连接就可以发送封装的IP 数据包的方法。IETF RFC 768是UDP的正式规范。UDP在IP报文的协议号是17。

UDP协议与TCP协议一样都位于传输层，处于IP协议的上一层。UDP的特点是：不提供数据包分组、组装和不能对数据包进行排序，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。对于包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都需要使用UDP协议。而且许多应用只支持UDP，比如：多媒体数据流，不产生任何额外的数据，即使知道有坏包也不进行重发。当强调传输性能而不是传输的完整性时，如：音频和多媒体应用，大多选择UDP传输。当数据传输时间很短，以至于建立连接的时间相对整个数据流量而言显得较长时，选择UDP更适合。

### UdpSocket

Rust标准库中用于创建和操作UDP套接字的类型是std::net::UdpSocket，常用于对延迟要求较高、数据完整性要求较低的应用场景，如多媒体流传输、实时游戏等。通过std::net::UdpSocket，可以实现以下功能：

（1）创建和绑定UDP套接字：可以使用UdpSocket::bind函数创建和绑定一个UDP套接字，以便于接收和发送UDP数据包。

（2）发送和接收UDP数据包：使用UdpSocket::send和UdpSocket::recv函数可以分别发送和接收UDP数据包。

（3）异步UDP操作：std::net::UdpSocket还支持异步操作，你可以使用async-std、tokio等异步运行时进行异步UDP通信。

除了上述基本功能外，std::net::UdpSocket还提供了其他一些方法，如获取远程端口信息、设置套接字选项等。

总之，std::net::UdpSocket在Rust中提供了操作UDP套接字的接口，方便进行UDP网络通信。

## 示例：UDP协议

与TCP不同，UDP不需要建立，也没有流结构。因此UDP服务器端与TCP服务器端的流程不同，下面通过一个示例体现其不同之处。

例5-9. 使用UDP的网络应用通信协议。

UDP服务器代码如下：

1. // chapter5/ch5-9-udp-echo-server.rs
2. use std::thread;
3. use std::net::UdpSocket;
4. fn main() {
5. let socket = UdpSocket::bind("0.0.0.0:8888")
6. .expect("Could not bind socket");
7. loop {
8. let mut buf = [0u8; 1500];
9. let sock = socket.try\_clone().expect("Failed to clone socket");
10. match socket.recv\_from(&mut buf) {
11. Ok((\_, src)) => {
12. thread::spawn(move || {
13. println!("Handling connection from {}", src);
14. sock.send\_to(&buf, &src)
15. .expect("Failed to send a response");
16. });
17. },
18. Err(e) => {
19. eprintln!("couldn't recieve a datagram: {}", e);
20. }
21. }
22. }
23. }

与TCP一样，首先绑定到给定端口上的本地地址，并处理绑定失败的错误。由于UDP是一个无连接协议，不需要使用滑动窗口来读取所有数据，因此可以只分配一个给定大小的静态缓冲区。该缓冲区大小可以根据动态检测底层网卡的MTU设置，最大不会超过UDP数据包的最大限值。由于常见的局域网的MTU大约是1500，我们可以为此分配一个相应大小的缓冲区。try\_clone方法复制给定的套接字，并返回一个新套接字，将其移到闭包中。

接下来，从套接字中读取数据，在Ok()情况下返回读取数据的长度和源地址信息。然后生成一个新的线程，将缓冲区中读取的数据写回到指定的套接字。

现在，编写一个简单的UDP客户端程序，这与在TCP服务器之间有一些细微的差别。

UDP客户端代码如下：

1. // chapter5/ch5-9-udp-client.rs
2. use std::net::UdpSocket;
3. use std::{str,io};
4. fn main() {
5. let socket = UdpSocket::bind("127.0.0.1:8000")
6. .expect("Could not bind client socket");
7. socket.connect("127.0.0.1:8888")
8. .expect("Could not connect to server");
9. loop {
10. let mut input = String::new();
11. let mut buffer = [0u8; 1500];
12. io::stdin().read\_line(&mut input)
13. .expect("Failed to read from stdin");
14. socket.send(input.as\_bytes())
15. .expect("Failed to write to server");
16. socket.recv\_from(&mut buffer)
17. .expect("Could not read into buffer");
18. print!("{}", str::from\_utf8(&buffer)
19. .expect("Could not write buffer as string"));
20. }
21. }

由上面的示例可见，UDP客户端和TCP客户端之间的主要区别在于，在连接到服务器之前，必须首先绑定bing到客户端套接字。一旦完成绑定，其余部分基本相同的。

运行服务器端代码，结果如下：

**$ rustc ch5-9-udp-echo-server.rs && ./ch5-9-udp-echo-server**

图片包含 图形用户界面

描述已自动生成

此时，编译运行客户端代码，结果如下：**$ rustc ch5-9-udp-client.rs && ./ch5-9-udp-client**

形状

中度可信度描述已自动生成

### UDP组播

UdpSocket类型还支持组播和广播通信。下面通过示例，演示如何实现UDP组播通信。示例中将把客户端和服务器合并在一个文件中。在主函数中，首先检查是否传递了一个控制台参数。如果有，将运行客户端；否则，将运行该服务器端。

例5-10. 使用UDP协议实现组播功能。

UDP组播程序代码：

1. // chapter5/ch5-10-udp-multicast.rs
2. use std::{env, str};
3. use std::net::{UdpSocket, Ipv4Addr};
4. fn main() {
5. let mcast\_group: Ipv4Addr = "239.0.0.1".parse().unwrap();
6. let port: u16 = 6000;
7. let any = "0.0.0.0".parse().unwrap();
8. let mut buffer = [0u8; 1600];
9. if env::args().count() > 1 {
10. // client case
11. let socket = UdpSocket::bind((any, port))
12. .expect("Could not bind client socket");
13. socket.join\_multicast\_v4(&mcast\_group, &any)
14. .expect("Could not join multicast group");
15. socket.recv\_from(&mut buffer)
16. .expect("Failed to write to server");
17. print!("{}", str::from\_utf8(&buffer)
18. .expect("Could not write buffer as string"));
19. } else {
20. // server case
21. let socket = UdpSocket::bind((any, 0))
22. .expect("Could not bind socket");
23. socket.send\_to("Hello world!".as\_bytes(), &(mcast\_group, port))
24. .expect("Failed to write data");
25. }
26. }

示例中客户端部分和服务器部分都与之前的单播非常相似。实现组播主要是通过调用join\_multicast\_v4，使当前套接字加入一个组播组。对于服务器和客户端，在绑定时都不指定单个地址，而是使用特殊的地址0.0.0.0，它表示任何可用的地址。这相当于将INADDR\_ANY传递给底层的设置选择调用setsockopt。在服务器的情况下，将它发送到组播组。由于无法在标准库中设置SO\_REUSEADDR和SO\_REUSEPORT，因此我们需要在不同主机上运行客户端，而在另外的主机上运行服务器。为此，所有主机都需要在同一个网络中，并且组播组的地址必须一个有效的组播地址（前四位应该是1110）。当然，UdpSocket类型还支持离开多播组、广播等等。

分别运行服务器和客户端程序。此时，服务器端的输出结果如下：

**$ rustc ch5-10-udp-multicast.rs && ./ch5-10-udp-multicast server**

Hello world!

## 使用tokio::net进行网络编程

tokio提供了类似std::net所提供的基本设施以便进行异步网络编程，主要包括tcp、udp和unix domain三方面，其中unix-domain仅在UNIX系统上可用。

网络编程需要大量的网络编程知识，且和IO编程息息相关，因暂时还未介绍tokio::io，所以本节暂且仅介绍tokio::net的tcp编程相关的基础设施，不涉及具体的网络编程逻辑。

要使用tokio::net，需在Cargo.toml文件中启用net特性：

tokio = {version = "1.13", features = ["rt", "net", "rt-multi-thread"]}

启用该特性之后，将可使用以下三个组件：

* TcpSocket：创建和操作套接字的基础组件
* TcpListener：对TcpSocket的一些封装，主要提供服务端套接字的相关操作
* TcpStream：代表已建立的可直接传递数据的连接，对客户端来说，表示已经被服务端接收；对服务端来说，表示accept后的套接字。

通常客户端可直接使用TcpStream，服务端可直接使用TcpListener和TcpStream。如果需要自定义修改套接字的选项或属性，则考虑使用TcpSocket。

### tokio::net::TcpListener

TcpListener代表服务端套接字，可使用bind()方法指定要绑定的地址，bind()之后再await，即可开始监听。

1. use tokio::net::TcpListener;
2. #[tokio::main]
3. async fn main(){
4. let listener = TcpListener::bind("127.0.0.1:8888").await.unwrap();
5. }

这里的listener代表的是服务端负责监听的套接字。

注意，TcpListener::bind()默认会开启TCP的地址重用选项(SO\_REUSEADDR)。如果想要修改该选项或设置其它TCP选项，应使用TcpSocket来创建套接字并设置选项，然后再调用bind()方法得到监听套接字。

得到监听套接字之后，可使用accept()去接收来自客户端的连接请求。accept()会阻塞(等待)，直到有新的客户端发起连接请求。

accept()成功，表示和客户端之间成功建立TCP连接(连接进入Established状态)，同时它会返回一个新的套接字(TcpStream)和代表客户端的套接字地址(SocketAddr)。可通过该TcpStream和客户端传输数据，可通过该SocketAddr获取客户端的地址和端口信息。如果要获取本地套接字地址相关的信息，可使用listener的local\_addr()方法。

通常来说，会在一个无限循环中去accept()，这样可以保证多次接收客户端的连接请求。此外，通常会为每个accept()成功后返回的TcpStream分配一个独立的线程或异步任务，这样可以异步地与每个客户端进行通信，且不影响监听套接字继续监听更多的客户端连接请求。

因此，tcp编程的服务端最基本的处理模式大致如下：

1. async fn main(){
2. let listener = TcpListener::bind("127.0.0.1:8888").await.unwrap();
3. loop {
4. let (client, client\_sock\_addr) = listener.accept().await.unwrap();
5. tokio::spawn(async move {
6. // 该任务负责处理client
7. });
8. }
9. }

此外，tokio的监听套接字可和标准库的监听套接字(std::TcpListener)来回转换。由于tokio只提供了成品套接字，无法设置很多的套接字选项，因此如果需要修改或设置某些套接字选项，需要先构建标准库的套接字并设置选项，然后使用from\_std()将标准库套接字转换为tokio的套接字。与from\_std()对应的是into\_std()。

### tokio::net::TcpSocket

TcpSocket用于创建和设置套接字选项，它是未进行连接的套接字，可通过bind()和listen()操作得到服务端的监听套接字，可通过connect()得到客户端的套接字。

例如，创建监听套接字，下面的操作等价于TcpListener.bind()操作，它将监听127.0.0.1:8080端口：

1. use tokio::net::TcpSocket;
2. #[tokio::main]
3. async fn main() {
4. let addr = "127.0.0.1:8080".parse().unwrap();
5. let socket = TcpSocket::new\_v4().unwrap();
6. socket.set\_reuseaddr(true).unwrap();
7. socket.bind(addr).unwrap();
8. let listener = socket.listen(1024).unwrap();
9. }

下面的操作等价于TcpStream::connect()操作，它将连接127.0.0.1:8080并返回该连接的TcpStream：

1. use tokio::net::TcpSocket;
2. #[tokio::main]
3. async fn main() {
4. let addr = "127.0.0.1:8080".parse().unwrap();
5. let socket = TcpSocket::new\_v4().unwrap();
6. let stream = socket.connect(addr).await.unwrap();
7. }

### TcpStream

TcpStream代表客户端和服务端之间已经建立的可以进行数据通信的TCP连接。当然，TcpStream也提供了connect()方法来方便地建立和TCP服务端的连接。

let mut stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();

TcpStream用于客户端和服务端的通信，因此可对其进行读和写。读操作表示接收来自对端发送过来的数据，写操作表示将数据通过TCP连接发送给对端。但是，通常会使用tokio::io::AsyncReadExt和tokio::io::AsyncWriteExt提供的读写API来读写TcpStream，因尚未介绍tokio::io，因此先跳过相关的读写操作。

TcpStream本身也提供了和读写相关的一些api：

* readable(): 等待TcpStream有数据可读
* writable(): 等待TcpStream可写入数据
* ready(): 类似Linux的select系统调用，注册可读、可写、读写关闭等事件后等待这些事件的出现
* try\_read(): 尝试以不等待的方式读取TcpStream
* try\_read\_buf(): 尝试以不等待的方式读取TcpStream，并将读取成功的数据追加到给定的buf中和try\_read()不同的是，try\_read()每次读取数据后都会从前向后覆盖buf的字节，而try\_read\_buf()则是将读取的数据追加到buf的尾部
* try\_read\_vectored(): 尝试以不等待的方式读取TcpStream，并将读取成功的数据分别填充到给定的一个或多个buf中

例如，给定了两个64K大小的buf，读取了100K数据，则前64K填充到第一个buf中，剩余的36K填充到第二个buf中

* try\_write(): 尝试以不等待的方式写入TcpStream
* try\_write\_vectored(): 尝试以不等待的方式写入TcpStream，写入的数据源来自于给定的一个或多个buf
* peek(): 从TcpStream中读取数据，但不消费TcpStream中本次读取的数据。即，peek后还可以再次读取这部分数据
* split(): 将TcpStream的读和写进行分离，得到的读、写两端不可跨线程(或任务)
* into\_split(): 将TcpStream的读和写进行分离，得到的读、写两端可跨线程(或任务)

稍后将简单介绍这些和读写相关的API的基本用法。

除了以上和IO相关的API，TcpSteam还提供了几个TCP连接选项设置的API：

* set\_linger(): 修改TCP连接的SO\_LINGER选项。在关闭连接时如果仍有未发送数据(比如仍然在缓冲等待着更多数据进入)，设置该选项决定是否要等待一段时间(期待后续会将缓冲的数据发送出去)才允许关闭TCP连接。若不设置该选项，则默认不等待
* linger(): 获取linger设置的值
* set\_nodelay(): 修改TCP连接的TCP\_NODELAY选项。设置该选项后，写入TcpStream的数据都将立即发送，而不会缓冲并等待凑够数据后才发送
* nodelay(): 是否设置了nodelay选项

再来介绍TcpStream提供的和读写相关的API。

通常，读取相关的操作(try\_read、peek等)会结合readable()来使用，写入相关的操作(try\_write)会结合writable()来使用。但是注意，即便readable()、writable()的返回分别代表了可读和可写，但这个可读、可写的就绪事件并不能确保真的可读可写，因此读、写时要做好判断，readable()结合try\_read()读数据如下。

1. use tokio::net::TcpStream;
2. use std::io;
3. #[tokio::main]
4. async fn main() {
5. let stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
6. let mut msg = vec![0; 1024];
7. loop {
8. // 等待可读事件的发生
9. stream.readable().await.unwrap();
10. // 即便readable()返回代表可读，但读取时仍然可能返回WouldBlock
11. match stream.try\_read(&mut msg) {
12. Ok(n) => { // 成功读取了n个字节的数据
13. msg.truncate(n);
14. break;
15. }
16. Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
17. continue;
18. }
19. Err(e) => {
20. return;
21. }
22. }
23. }
24. println!("GOT = {:?}", msg);
25. }

当然，读写操作也可以结合ready()来使用，调用ready()时可注册感兴趣的事件，当注册的事件之一发生之后，ready()将返回Ready结构体，Ready结构体有一些布尔判断方法，用来判断某个事件是否发生，如下所示。

1. use tokio::io::Interest;
2. use tokio::net::TcpStream;
3. use std::io;
4. #[tokio::main]
5. async fn main() {
6. let stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
7. loop {
8. // 注册可读和可写事件，并等待事件的发生
9. let ready = stream.ready(Interest::READABLE | Interest::WRITABLE).await.unwrap();
10. // 如果注册的事件中，发生了可读事件，则执行如下代码
11. if ready.is\_readable() {
12. let mut data = vec![0; 1024];
13. match stream.try\_read(&mut data) {
14. Ok(n) => {
15. println!("read {} bytes", n);
16. }
17. Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
18. continue;
19. }
20. Err(e) => {
21. return;
22. }
23. }
24. }
25. // 如果注册的事件中，发生了可写事件，则执行如下代码
26. if ready.is\_writable() {
27. match stream.try\_write(b"hello world") {
28. Ok(n) => {
29. println!("write {} bytes", n);
30. }
31. Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
32. continue
33. }
34. Err(e) => {
35. return;
36. }
37. }
38. }
39. }
40. }

peek()可读取TcpStream中的数据，但是和其它读取操作不同，peek()读取之后不会消费TcpStream中的数据。

1. use tokio::net::TcpStream;
2. use tokio::io::AsyncReadExt;
3. #[tokio::main]
4. async fn main() {
5. let mut stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
6. let mut b1 = [0; 10];
7. let mut b2 = [0; 10];
8. let n = stream.peek(&mut b1).await.unwrap();
9. let n1 = stream.read(&mut b2[..n]).await.unwrap();
10. }

比较关键的是split()方法。TCP连接是全双工通信的，无论是TCP连接的客户端还是服务端，每一端都可以进行读操作和写操作。为了方便描述，此处将其称为读端和写端。即，客户端和服务端都有读端和写端。

通过TcpStream，可进行读操作，也可以进行写操作，正如前面几个示例代码所示。但是，通过TcpStream同时进行读写有时候会很麻烦，甚至无解。很多时候，需要将TcpStream的读端和写端进行分离，然后将分离的读、写两端放进独立的异步任务中去执行读或写操作(此时需跨线程)，即一个线程(或异步任务)负责读，另一个线程(或异步任务)负责写。

split()和into\_split()正是用来分离TcpStream的读写两端的。

split()可将TcpStream分离为ReadHalf和WriteHalf，ReadHalf用于读，WriteHalf用于写。

let mut conn = TcpStream::connect("127.0.0.1:8888").await.unwrap();

let (mut read\_half, mut write\_half) = conn.split();

split()并没有真正将TcpStream的读写两端进行分离，仅仅只是引用TcpStream中的读端和写端。因此，split()得到的读写两端只能在当前任务中进行读写操作，不允许跨线程跨任务。

into\_split()是split()的owned版，分离后可得到OwnedReadHalf和OwnedWriteHalf。它是真正地分离TcpStream的读写两端，它会消耗掉TcpStream。OwnedReadHalf和OwnedWriteHalf可跨任务进行读写操作。

let conn = TcpStream::connect("127.0.0.1:8888").await.unwrap();

let (mut read\_half, mut write\_half) = conn.into\_split();

请记住TcpStream的split()和into\_split()方法，这两个方法在tokio网络编程时都非常常用。

# HTTP应用编程

本章要点：熟悉Rust中的Web应用，掌握Hyper软件包构建短网址应用、了解Web框架的需求，熟悉actix-web框架，使用actix-web框架构建HTTP Rest API。

## 引言

Rust是一种编译型语言，因此开发Web程序的方式与其他编译型语言类似。构建的应用程序会监听HTTP请求，然后根据路由逻辑将这些请求导向处理函数，这些函数会返回响应。由于Rust程序默认是静态链接的，因此在生产环境中部署这样的程序通常只需简单地复制可执行文件并重启应用程序。由于Rust能提供更好的性能和可预测性，因此其在Web开发中极具吸引力。

在Rust生态系统中，有多个库和框架用于处理HTTP请求和响应，比较常用的包括Hyper、Rocket和Reqwest等。

* Hyper是Rust中最常用的HTTP库，许多高级框架都基于它来构建。与许多HTTP库中将所有内容描述为字符串的常见做法相反，Hyper被设计为原始HTTP的类型安全抽象。例如，在Hyper中，HTTP状态码被定义在Hyper::status::StatusCode中，这是一个包含所有标准状态码的枚举，几乎所有可以强类型化的内容，如HTTP方法、MIME类型、HTTP头等，都以这种方式处理。这种设计方式可以提高代码的可读性和可维护性，同时也减少了因字符串操作引起的潜在错误。然而，这也意味着开发者需要熟悉hyper的API和Rust的类型系统，以便有效地使用它。
* Rocket是一个快速、无开销的Web框架，用于构建高性能的Web应用程序。它提供了路由、中间件、请求解析等功能，并且非常注重性能和易用性。Rocket的核心理念是简单和快速，它提供了直观的方式来定义路由和处理器，同时内置了对异步和同步请求的支持。
* Reqwest是一个功能强大的HTTP客户端库，用于发送HTTP请求并接收响应。它支持同步和异步请求，提供了简单易用的API来处理各种HTTP操作。它允许开发者通过简单的函数调用来发送HTTP请求，提供了丰富的配置选项来满足各种需求，同时支持自动重试、连接池、代理等功能，使HTTP请求更加可靠和高效。

这些库和框架在Rust生态系统中各有其独特之处，接下来我们将详细介绍它们，开发者可以根据具体需求选择适合的库来处理HTTP请求和响应。

## Hyper框架

Hyper是一个基于Rust的低级HTTP库，用于构建高性能的网络应用程序。它提供了处理HTTP请求和响应的功能，以及构建HTTP服务器和客户端的工具。

Hyper库的主要特点和功能如下：

* 异步支持：Hyper使用Rust的异步运行时tokio来实现异步处理，充分利用了Rust的async/await语法，便于编写高效、非阻塞的网络代码。
* HTTP/1和 HTTP/2 支持：Hyper支持HTTP/1和HTTP/2协议，可以处理这两种协议的请求和响应。
* 路由和中间件：Hyper提供了路由和中间件的支持，可以轻松地定义和管理不同的路由和中间件，以实现更复杂的应用逻辑。
* 强大的请求和响应处理：Hyper提供了丰富的API来处理HTTP请求和响应，包括设置请求头、解析请求体、设置响应状态码、设置响应头等。
* 可扩展性：可以根据需要自定义和扩展各个组件，以满足特定的应用需求。

### 服务器编程

**（1）Hyper 框架简介**

Hyper是一个基于Rust编程语言的高性能、灵活的网络框架，专门用于构建可扩展的HTTP和异步Web应用程序，完全用Rust编写，充分发挥Rust的内存安全和并发性能优势。

作为一个现代化的网络框架，Hyper提供了许多强大的功能和特点，使得它成为构建高效可靠的服务器应用程序的理想选择。Hyper支持异步处理和非阻塞I/O，高效处理大量并发请求；提供灵活的API，支持各种HTTP方法，简化构建RESTfulAPI的过程；提供了路由器和中间件机制，方便开发人员实现自定义的请求处理、响应转换和认证授权等功能；提供多样的错误类型和自定义错误响应，提升应用程序的可靠性和用户体验；支持可插拔的中间件，如日志记录、压缩和缓存等。

总之，Hyper框架在网络编程领域具有独特的优势，它适用于从小型项目到大规模分布式系统的多种场景，满足各类网络应用开发需求。

**（2） HTTP服务端基础**

HTTP（超文本传输协议）是Web上数据交换的基础协议，它定义了客户端和服务器之间的通信规则，通过请求-响应的方式进行通信，使用URL（统一资源定位符）来标识和定位资源。

HTTP的请求由请求行、请求头部和请求体组成，请求行包含请求方法（GET、POST等）和请求的目标URL；请求头部包含关于请求的元数据信息，如内容类型、cookie等；请求体包含客户端发送给服务器的数据。HTTP的响应由状态行、响应头部和响应体组成，状态行包含响应的状态码和状态消息；响应头部包含响应的元数据信息，如内容类型、响应时间等；响应体包含服务器返回给客户端的数据。

HTTP的头部包含一些关键的信息，用于请求和响应的处理，在服务端编程中，需要解析和处理头部信息，以便正确地理解和处理请求。通过解析头部信息，可以获取到客户端传递的参数、身份验证信息等，并根据需要进行相应的处理。

在编写HTTP服务端应用程序时，还需要根据HTTP协议规范和要求进行请求的解析和响应的生成。这包括正确解析和处理请求行、请求头部、请求体，并根据业务逻辑生成合适的响应状态行、响应头部和响应体。

总的来说，HTTP服务端基础知识包括了HTTP协议的概述和基本概念、请求和响应的结构和组成部分，以及头部信息的解析和处理，理解和掌握这些关键知识点对于编写可靠、高效的HTTP服务端应用程序至关重要。

**（3） Hyper框架的的使用方法**

使用Hyper框架构建Rust服务端时的步骤如下：

* 在Cargo.toml文件中添加Hyper的依赖项；
* 在Rust源文件中导入所需的Hyper库和其他相关库，处理请求和构建响应；
* 定义一个处理函数用于接收请求并返回响应，这个函数应该是异步的，以支持高性能的并发处理；
* 创建一个服务工厂，负责创建服务，并与处理函数进行绑定，在每个连接上调用处理函数来处理请求；
* 定义服务器的绑定地址和端口，然后通过Hyper的Server::bind方法将服务器绑定到指定的地址和端口上；
* 在主函数中，使用tokio::main宏创建一个异步运行时，以支持异步代码执行；
* 启动服务器并等待它在后台运行，服务器将监听绑定的地址和端口，并根据每个请求调用处理函数来生成相应的响应。

通过以上步骤，可以使用Hyper框架构建一个简单的Rust服务端，按照我们的需求修改处理函数、处理各种不同的请求、返回相应的响应。同时，Hyper的异步特性和高性能保证服务器能够处理大量并发请求，并提供出色的性能和响应速度。

**（4） 使用 Hyper 框架构建 Rust 服务端操作步骤的实施**

1. **添加依赖**

在 `Cargo.toml` 文件中，添加 Hyper 的依赖项。可以使用以下语法添加依赖：

1. [dependencies]

2. hyper = "0.13.10"

3. tokio = { version = "0.2.25", features = ["full"] }

第2行hyper = "0.14.4"指定了hyper包作为项目的依赖项，0.14.4 是指要使用该库的特定版本号。；第3行tokio = { version = "1", features = ["full"] }：这一行指定了 tokio 包作为项目的依赖项。tokio 是一个异步运行时库，用于开发基于异步操作的应用程序。{ version = "1", features = ["full"] } 表示要使用 tokio 的 1.x 版本，并且希望启用其提供的全部特性。

1. 导入必要的库

在 Rust 源文件中，导入需要使用的 Hyper 库和其他相关库。可以使用以下语法导入库：

1. use hyper::{Body, Request, Response, Server};

2. use hyper::service::{make\_service\_fn, service\_fn};

3. use std::convert::Infallible;

4. use std::net::SocketAddr;

第1行使用了 hyper 模块并导入了 Body、Request、Response和Server类型。Body表示 HTTP 请求或响应的消息体，Request表示 HTTP 请求，Response表示 HTTP 响应，Server表示运行 HTTP 服务器的类型。第2行hyper模块中的service子模块，并导入了make\_service\_fn和service\_fn函数。make\_service\_fn函数用于创建一个服务工厂函数，service\_fn函数用于将一个异步函数包装为一个 HTTP 服务。第3行使用了标准库中的convert模块，并导入了Infallible类型。Infallible是一个表示不可能发生错误的类型，用于在某些情况下简化函数签名。第4行使用了标准库中的net模块，并导入了SocketAddr类型。SocketAddr表示一个套接字的网络地址，用于指定服务器应该监听的 IP 地址和端口。

1. 定义请求处理函数

创建一个处理函数，该函数接收一个请求并返回一个响应。可以根据具体的业务逻辑定制处理函数。例如：

1. async fn handle\_request(\_: Request<Body>) -> Result<Response<Body>, Infallible> {

2. let response = Response::builder()

3. .status(200)

4. .header("Content-Type", "text/plain")

5. .body(Body::from("Hello, Rust Hyper Server!"))

6. .unwrap();

7. Ok(response)

8. }

异步函数handle\_request，它接受一个Request<Body>参数，并返回一个Result<Response<Body>, Infallible>类型的结果。

1. 启动服务器

使用 Hyper 创建并启动一个服务器，监听指定的 Socket 地址，并在每个请求上调用处理函数。

1. #[tokio::main]

2. async fn main() {

3. // 定义服务器绑定的地址和端口

4. let addr = SocketAddr::from(([127, 0, 0, 1], 8080));

5. // 创建一个服务工厂

6. let make\_svc = make\_service\_fn(|\_conn| async {

7. Ok::<\_, Infallible>(service\_fn(handle\_request))

8. });

9. // 创建服务器并绑定地址

10. let server = Server::bind(&addr).serve(make\_svc);

11. println!("Server running at HTTP://{}", addr);

12. // 启动服务器并等待它在后台运行

13. if let Err(e) = server.await {

14. eprintln!("Server error: {}", e);

15. }

16. }

1. 异步运行

为了支持异步代码，需要在 `main` 函数中使用 `tokio::main` 宏来创建异步运行时。例如：

1. #[tokio::main]

2. async fn main() {

3. // ...

4. }

1. 处理请求

在请求处理函数中，可以根据具体的需求处理请求，并构建相应的响应。例如，可以直接返回一个固定的文本响应：

1. async fn handle\_request(\_: Request<Body>) -> Result<Response<Body>, Infallible> {

2. let response = Response::builder()

3. .status(200)

4. .header("Content-Type", "text/plain")

5. .body(Body::from("Hello, Rust Hyper Server!"))

6. .unwrap();

7. Ok(response)

8. }

1. 运行服务端

最后，在 `main` 函数中创建服务器对象并绑定地址，在后台运行服务器：

1. #[tokio::main]

2. async fn main() {

3. // ...

4. let server = Server::bind(&addr).serve(make\_svc);

5. if let Err(e) = server.await {

6. eprintln!("Server error: {}", e);

7. }

8. }

以上是使用 Hyper 框架构建 Rust 服务端的示例代码，我们可以根据实际需求扩展和定制代码，以满足特定的服务端功能要求。

依据上面给出的代码，下面给出完整的依赖以及代码：

1. [dependencies]

2. hyper = "0.13.10"

3. tokio = { version = "0.2.25", features = ["full"] }

1. use hyper::{Body, Request, Response, Server};

2. use hyper::service::{make\_service\_fn, service\_fn};

3. use std::convert::Infallible;

4. use std::net::SocketAddr;

5.

6.

7. async fn handle\_request(\_: Request<Body>) -> Result<Response<Body>, Infallible> {

8. let response = Response::builder()

9. .status(200)

10. .header("Content-Type", "text/plain")

11. .body(Body::from("Hello, Rust Hyper Server!"))

12. .unwrap();

13. Ok(response)

14. }

15.

16. #[tokio::main]

17. async fn main() {

18. // 定义服务器绑定的地址和端口

19. let addr = SocketAddr::from(([127, 0, 0, 1], 8080));

20.

21. // 创建一个服务工厂

22. let make\_svc = make\_service\_fn(|\_conn| async {

23. Ok::<\_, Infallible>(service\_fn(handle\_request))

24. });

25.

26. // 创建服务器并绑定地址

27. let server = Server::bind(&addr).serve(make\_svc);

28.

29. // 启动服务器并等待它在后台运行

30. println!("Server running at HTTP://{}", addr);

31. if let Err(e) = server.await {

32. eprintln!("Server error: {}", e);

33. }

34. }

运行结果如下：



上述代码是一个简单的 Rust Hyper 服务器示例，它使用了 Hyper 库来构建异步的 HTTP 服务器。首先引入必要的类型和函数，以便在后续代码中直接使用它们而不必在每次使用时都写出完整的路径。接下来的代码是一个异步函数 handle\_request，它接收一个 Request<Body> 参数，并返回一个 Result<Response<Body>, Infallible> 类型的结果。在这个函数中，它创建了一个包含 “Hello, Rust Hyper Server!” 字符串的响应，并将状态码设置为 200。在 main 函数中，首先定义了服务器绑定的地址和端口（IPv4 地址 127.0.0.1，端口 8080）。然后使用 make\_service\_fn 创建了一个服务工厂，它会在每个连接上调用 service\_fn，而 service\_fn 又会将请求交给 handle\_request 处理。接着使用 Server::bind 绑定地址并启动服务器，最后使用 server.await 启动服务器并等待它在后台运行。如果出现错误，会打印错误信息。

例6.1创建一个简单的 HTTP 服务器。它使用 Hyper 库提供的功能，监听本地地址 127.0.0.1 和端口 8080。当有连接请求时，会调用handle\_request 函数来处理请求，并返回一个带有 “Hello, Hyper!” 消息的响应。此服务器可以用于接收和处理 HTTP 请求，并向客户端发送相应的数据。

Cargo.toml 文件中添加 Hyper 的依赖项：

1. hyper = "0.13.10"

2. tokio = { version = "0.2.25", features = ["full"] }

Rust 源文件中的代码：

1. use hyper::{Body, Request, Response, Server};

2. use hyper::service::{make\_service\_fn, service\_fn};

3.

4. async fn handle\_request(req: Request<Body>) -> Result<Response<Body>, hyper::Error> {

5. // 在这里处理请求逻辑，生成相应的响应

6. let body = Body::from("Hello, Hyper!");

7. Ok(Response::new(body))

8. }

9. #[tokio::main]

10. async fn main() {

11. // 创建一个服务工厂，每个连接都会调用处理函数 handle\_request 来处理请求

12. let make\_svc = make\_service\_fn(|\_conn| {

13. async {

14. Ok::<\_, hyper::Error>(service\_fn(handle\_request))

15. }

16. });

17.

18. // 定义服务器的绑定地址和端口

19. let addr = ([127, 0, 0, 1], 8080).into();

20. // 绑定服务器到指定的地址和端口上

21. let server = Server::bind(&addr).serve(make\_svc);

22. println!("Server running at HTTP://{}", addr);

23. // 启动服务器并等待它在后台运行

24. if let Err(e) = server.await {

25. eprintln!("Server error: {}", e);

26. }

27. }

以上代码实现一个简单的 Hyper 框架下的 Rust 服务端，处理客户端请求并返回 "Hello, Hyper!" 的响应。

代码开头引入了 `hyper` 和 `tokio` 库，前者用于构建和处理 HTTP 请求，后者用于创建异步运行时环境。第4行的`handle\_request` 函数用于处理请求并生成响应，其返回类型为 `Result<Response<Body>, hyper::Error>`。第12行的 `make\_service\_fn` 创建了一个服务工厂，会在每个连接上调用 `handle\_request` 函数来处理请求。21行的 `Server::bind` 方法将服务器绑定到指定的地址和端口上。对于第9行中使用 `tokio::main` 宏创建异步运行时环境，在 `main` 函数中异步执行代码。最后启动服务器并等待它在后台运行，处理来自客户端的请求。

运行结果如下：



运行该代码后，控制台将输出以下内容：

“Server running at HTTP://127.0.0.1:8080”（服务器运行在 HTTP://127.0.0.1:8080）。这是在代码的第22行打印的一条消息，指示服务器已经成功地绑定到指定的地址和端口上，并准备好接受来自客户端的连接与请求。若运行过程中出现错误，将在第25行打印相应的错误信息。

### HTTP 客户端编程

**（1）HTTP客户端基础**

HTTP（Hypertext Transfer Protocol）协议客户端是一种用于与服务器进行通信的软件或程序，它遵循HTTP协议规范来发送请求并接收响应。HTTP客户端的主要功能是与服务器建立连接，发送请求以获取资源，以及处理服务器返回的响应。

* 建立连接：客户端通过网络与服务器建立连接，通常使用TCP/IP协议进行通信。
* 发送请求：客户端构造HTTP请求并发送给服务器。
* 接收响应：客户端接收服务器返回的HTTP响应。
* 处理响应：客户端解析响应数据，提取所需信息，并根据需要做出相应的处理。

1. **HTTP的请求（Request）**

HTTP请求由请求行、请求头和请求体组成。请求行包含请求方法（如GET、POST、PUT、DELETE等）、请求的URL和使用的协议版本；请求头包含附加的元数据信息，如User-Agent（用户代理，标识请求的客户端）、Content-Type（请求体内容的类型）等；请求体可选，用于传递请求的实际数据，比如在POST请求中传递表单数据或JSON数据。

1. GET请求：用于从服务器获取资源。它是一种幂等的请求，即多次发送同样的GET请求会得到相同的结果。GET请求的主要特点如下：

* 请求数据附加在URL中的查询字符串中，以键值对的形式出现，通过"?"后面的字符串（建议加上“进行”）传递。
* GET请求是基于URL的请求，因此请求参数会暴露在URL中，可以通过在浏览器中直接访问URL来发送GET请求。
* GET请求不适合发送大量数据，因为URL长度有限制。
* GET请求的响应结果通常是缓存的，可以被浏览器缓存和其他代理服务器缓存。

1. POST请求：用于向服务器发送数据并在服务器上创建新资源或更新现有资源。POST请求的主要特点如下：

* 请求数据包含在请求体中，而不是URL的查询字符串中，可以传递更多数据和更大的数据量。
* POST请求不会在URL中暴露请求参数，因此更适合发送敏感数据。
* POST请求不是幂等的，即多次发送相同的POST请求可能会创建或修改多个资源。
* POST请求的响应结果一般不会被缓存，每次发送POST请求都会从服务器获取最新的数据。

1. **HTTP的响应（Response）**

HTTP响应由响应行、响应头和响应体组成。响应行包含响应的协议版本、状态码和状态信息；响应头包含服务器返回的元数据信息，如Content-Type（响应体内容的类型）、Content-Length（响应体长度）等；响应体包含服务器返回的实际数据，例如HTML文档、JSON数据等。

1. **HTTP的状态码（Status Code）**

状态码是服务器对请求的响应状态的标识。常见的状态码如表6.1所示:

表6.1 HTTP的状态码

|  |  |  |
| --- | --- | --- |
| 状态码 | 表示状态 | 常见状态码 |
| 2xx | 成功 | 204表示请求成功但无响应体等 |
| 3xx | 重定向，告知客户端需要进一步操作访问资源 | 301永久重定向  302临时重定向 |
| 4xx | 客户端错误 | 400表示请求语法错误  404表示请求的资源不存在 |
| 5xx | 服务器错误 | 500表示服务器内部错误  503表示服务器暂时不可用 |

1. **客户端使用标头（Headers）传递附加信息**
2. **请求标头（Request Headers）**

请求标头用于传递客户端的附加信息给服务器。常见的请求标头如表6.2所示：

表6.2 请求标头

|  |  |
| --- | --- |
| 请求标头 | 作用 |
| User-Agent | 标识请求的客户端类型，如浏览器、移动应用程序 |
| Accept | 指定客户端能够接受的响应内容类型，以便服务器返回适合的响应 |
| Authorization | 用于身份验证的凭据，在需要访问受限资源时发送给服务器 |
| Cookie | 包含已存储在客户端的Cookie信息，以便与服务器建立会话 |
| Content-Type | 指定请求体的媒体类型，如"text/plain"、"application/json" |

1. **响应标头（Response Headers）**

响应标头用于传递服务器的附加信息给客户端。常见的响应标头如表6.3所示：

表6.3 响应标头

|  |  |
| --- | --- |
| 响应标头 | 作用 |
| Content-Type | 指明响应体的媒体类型，表示返回数据的格式 |
| Content-Length | 指定响应体的长度以字节为单位 |
| Location | 用于重定向响应，指示客户端继续请求的新位置 |
| Set-Cookie | 服务器通过这个标头将Cookie信息附加到响应中，客户端收到后进行存储 |

1. **通用标头（General Headers）**

通用标头用于传递与请求和响应消息相关的一般信息，它们不特定于特定的消息部分，可以在请求和响应中使用，用于提供关于消息的通用元数据。常见的通用标头如表6.4所示：

表6.4 通用标头

|  |  |
| --- | --- |
| 通用标头 | 作用 |
| Date | 表示消息创建的日期和时间 |
| Connection | 指定是否需要保持持久连接或断开连接的方式 |
| Transfer-Encoding | 指定响应消息所使用的传输编码方式。常见的传输编码方式包括"chunked"（分块传输编码）和"compress"（压缩传输编码） |

1. 实体标头（Entity Headers）

实体标头描述了请求或响应的主体部分，例如请求体或响应体的内容。常见的实体标头如表6.5所示：

表6.5 实体标头

|  |  |
| --- | --- |
| 实体标头 | 作用 |
| Content-Type | 指定请求体或响应体的媒体类型 |
| Content-Length | 指定请求体或响应体的长度以字节为单位 |

理解HTTP协议客户端标头的基本概念有助于我们设置适当的标头，传递附加信息给服务器或解析服务器的响应标头。特定的标头可以实现更灵活和定制化的通信。

**（2）用Hyper框架构建HTTP客户端**

Hyper框架构建HTTP客户端的整体流程如下：

* 引入Hyper依赖：通过引入Hyper库，可以使用其提供的功能构建HTTP客户端。
* 创建HTTP客户端实例：使用Hyper提供的Client类型，创建一个HTTP客户端实例，根据需要配置客户端的选项，如连接超时、连接池大小等。
* 构建HTTP请求：使用Hyper的Request类型，构建各种类型的HTTP请求，并设置请求方法、URL、请求头、请求体等。
* 发送HTTP请求：通过调用客户端实例的request方法，将构建的请求发送到目标服务器，Hyper将处理与服务器的连接管理、请求发送和响应接收等细节。
* 处理HTTP响应：Hyper以异步的方式接收响应，并将其封装在Response类型中，可以访问响应的状态码、响应头、响应体等，以便对响应进行处理。
* 处理错误和异常：使用Result和ResultExt trait来处理可能出现的错误，并采取适当的措施进行恢复或处理。

一个简单的HTTP客户端创建过程如下：

1. 添加依赖：在Cargo.toml文件中，添加Hyper的依赖项。

1. [dependencies]

2. hyper = "0.14.4"

3. tokio = { version = "1", features = ["full"] }

1. 导入必要的库：在Rust源文件中，导入需要使用的Hyper库和其他相关库。

1. use hyper::{Body, Client, Request, Uri};

2. use hyper::body::HTTPBody as \_;

3. use tokio::runtime::Runtime;

1. 创建客户端和请求：使用Hyper提供的Client和Request来创建HTTP客户端和请求，在需要发送HTTP请求的代码中创建客户端对象和请求对象。

1. async fn send\_request(url: &str) -> Result<(), Box<dyn std::error::Error>> {

2. let client = Client::new();

3. let request = Request::builder()

4. .method(Method::GET)

5. .uri(url)

6. .header(header::USER\_AGENT, "My Rust Client")

7. .body(Body::empty())

8. .unwrap();

9. }

1. 发送请求和处理响应：使用创建的客户端对象发送HTTP请求，并异步等待服务器的响应，使用client.request()方法发送请求，并使用await关键字等待响应的到来。

1. async fn send\_request(url: &str) -> Result<(), Box<dyn std::error::Error>>

2. {

3. // ...

4. let response = client.request(request).await?;

5. // ...

6. }

1. 处理响应数据：从响应中获取状态码、响应头和响应体等信息，并根据需求对数据进行处理。

1. async fn send\_request(url: &str) -> Result<(), Box<dyn std::error::Error>> {

2. // ...

3. let response = client.request(request).await?;

4. println!("Response status: {}", response.status());

5. let response\_bytes = hyper::body::to\_bytes(response.into\_body()).await?;

6. let response\_string = String::from\_utf8\_lossy(&response\_bytes);

7. println!("Response body: {}", response\_string);

8. // ...

9. }

1. 异步运行时（Optional）：如果使用异步代码，需要创建一个异步运行时并执行异步方法，可以使用Runtime来创建运行时，并在运行时中执行异步方法。

1. fn main()

2. {

3. let rt = Runtime::new().unwrap();

4. let result = rt.block\_on(send\_request("https://example.com"));

5. // 处理返回结果

6. }

通过以上步骤，可以使用Hyper框架编写简单的HTTP客户端。Hyper提供了高性能、异步的API以及方便的请求和响应处理功能，使得构建HTTP客户端变得更简单。我们可以根据具体需求扩展和定制客户端功能，并使用其他Rust工具和功能来增强HTTP客户端的能力。

**（3）实现HTTP客户端的实例**

通过以上内容的学习，我们已经初步掌握了Hyper框架和HTTP客户端的创建流程，接下来将引入一组实例来进一步体会HTTP客户端创建的过程。

1. 添加依赖：在Cargo.toml文件中，添加Hyper的依赖项。

1. [dependencies]

2. hyper = "0.13.10"

3. tokio = { version = "0.2.25", features = ["full"] }

1. 创建客户端和请求。

1. use hyper::{Body, Client, Uri}; // 导入需要的 Hyper 模块

2. use hyper::body::HTTPBody as \_; // 重命名 HTTPBody trait 为 \_

3. use hyper::Request; // 导入 Request 类型

4. use tokio::runtime::Runtime; // 导入 Runtime 类型

5. fn main() -> Result<(), Box<dyn std::error::Error>> {

6. // 创建同步运行时

7. let mut rt = Runtime::new()?; // 创建一个可以异步执行的运行时实例

8. // 在同步运行时中执行异步代码

9. rt.block\_on(async {

10. // 创建客户端

11. let client = Client::new(); // 创建 HTTP 客户端

12.

13. // 构建请求

14. let url = "HTTP://localhost:8088".parse::<Uri>().unwrap(); // 解析请求 URL

15. let request = Request::builder() // 创建请求构造器

16. .method("GET") // 请求方法

17. .uri(url) // 请求 URL

18. .body(Body::empty()) // 请求体为空

19. .unwrap();

20. // 发送请求

21. let response = client.request(request).await.unwrap(); // 发送请求并等待响应

22. // 处理响应数据

23. let body\_bytes = hyper::body::to\_bytes(response.into\_body()).await.unwrap(); // 读取响应体数据

24. let body\_string = String::from\_utf8(body\_bytes.to\_vec()).unwrap(); // 将响应体数据转换为字符串

25. println!("Response body: {}", body\_string); // 打印响应体内容

26. });

27. Ok(()) // 返回成功标志

28. }

29.

我们在11行创建HTTP客户端对象，15行创建请求构造器，Request::builder()方法创建一个请求构造器实例，可以用于构建 HTTP 请求。在16行设置请求方法，使用 method() 方法设置请求的方法，这里使用 "GET" 方法发送请求。最后在23行，读取响应体数据，使用 hyper::body::to\_bytes() 方法将响应体数据读取为字节流。

以上代码通过使用 Hyper 库发送了一个简单的 HTTP GET 请求，并打印出了响应体的内容。它完成了以下几个步骤：

1. 使用rt.block\_on(async { ... })创建一个同步运行时实例，在同步运行时中执行了异步代码。
2. 创建一个 Hyper 客户端，使用 Request::builder() 创建了一个请求构造器，设置请求方法为 GET，请求 URL 为 HTTP://localhost:8088，设置请求体为空。
3. 发送请求，等待响应。
4. 通过调用 hyper::body::to\_bytes 方法读取响应体的数据，将响应体数据转换为字符串形式，使用 println! 宏打印了响应体的内容。
5. 返回一个成功标志给 main 函数。

代码展示了如何使用 Hyper 库发送一个简单的 HTTP GET 请求，并处理响应体的内容，想要实现其他内容可以通过添加更加复杂的逻辑来实现，在此不再赘述。

根据以上的客户端代码，我们给出以下给出相应服务端代码供参考：

1. use hyper::{Body, Request, Response, Server}; // 导入需要的 Hyper 模块

2. use std::convert::Infallible; // 导入 Infallible 类型，表示无法发生错误

3. use hyper::service::{make\_service\_fn, service\_fn}; // 导入构建服务的相关类型和函数

4. use tokio::task; // 导入 tokio 任务调度器

5. // 处理请求的异步函数

6. async fn handle\_request(\_req: Request<Body>) -> Result<Response<Body>, Infallible> {

7. // 构建响应体

8. let body = "Hello World!".to\_string();

9. // 构建响应

10. let response = Response::builder()

11. .header("Content-Type", "text/plain")

12. .body(Body::from(body))

13. .unwrap();

14. Ok(response) // 返回响应

15. }

16. #[tokio::main] // 入口点宏，创建 tokio 运行时

17. async fn main() {

18. // 创建 Hyper 服务器

19. let address = ([127, 0, 0, 1], 8088).into(); // 定义服务器监听的地址为 localhost:8088

20. let server = Server::bind(&address).serve(make\_service\_fn(|\_socket| async {

21. Ok::<\_, Infallible>(service\_fn(handle\_request)) // 创建服务并传入处理请求的函数

22. }));

23.  println!("Server listening on HTTP://localhost:8088");

24. // 启动服务器并等待它在新的任务中运行

25. if let Err(e) = server.await {

26. eprintln!("Error: {}", e);

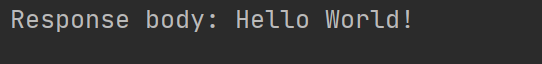
27. }

服务端代码创建了一个简单的服务器，并绑定到本地地址的 8088 端口上将所有进来的请求都返回一个包含"Hello World!"消息的响应体，监听本地的 8088 端口。客户端向端口8088发出请求，服务端接收到来自客户端的请求，将响应体Hello World!响应给客户端，最后客户端得到响应结果，控制台显示Response body: Hello World!。显示结果如下：

服务端输出：



客户端输出：



## Rocket框架

Rocket框架是Rust 生态中比较流行的Web框架之一，其最大的特点是拥有类似Flask那样比较简洁的写法，可以非常轻松地编写RESTful API，同时还支持中间件等机制，易于扩展。Rocket的目标是快速、简单和灵活，同时尽可能提供有保证的安全性。但是由于Rocket框架是基于Rust的Nightly版本的特性，所以使用前需要将Rust的版本切换成对应版本。

Rocket的设计以三个核心理念为中心：

1. **安全性、正确性和开发人员体验感至关重要**

最安全、最正确的Web应用程序开发路径应该是最不费力的，不过安全性和正确性不应以牺牲开发人员体验为代价，Rocket易于使用，同时采取了多项措施来确保应用程序安全且正确，不会增加额外的负担。

1. **所有请求处理信息都应该是类型化的并且是独立的**

由于Web和HTTP本身是无类型的（或者可以说是字符串类型），这意味着需要有某种方式将字符串转换为本地类型，Rocket可以完成这项任务，而且它的请求处理是独立的，没有全局状态的负担，处理程序是常规函数，具有常规参数。

1. **不应强制做出决策**

模板、序列化、会话以及几乎所有其他内容都是可插拔的可选组件。虽然Rocket为每个组件提供了官方支持和库，但它们完全是可选的，可以进行替换。

在我们深入学习Rocket框架之前，为了读者能够了解Rokcet应用程序创建和运行的基本步骤，我们以一个打印“Hello World”的应用程序为例进行说明。

首先创建一个基于二进制的 Cargo 项目，并切换到新的目录中。

1. cargo new hello-Rocket --bin
2. cd hello-Rocket

然后，将Rocket作为依赖项添加到Cargo.toml。

1. [dependencies]
2. Rocket = "=0.5.0-rc.3"

更改main.rs文件以使其包含Rocket程序的代码，如下所示。

1. #[macro\_use] extern crate Rocket;

2. #[get("/")]

3. fn index() -> &'static str {

4. "Hello, world!"

5. }

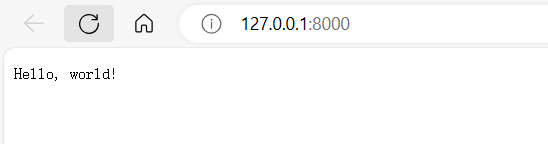
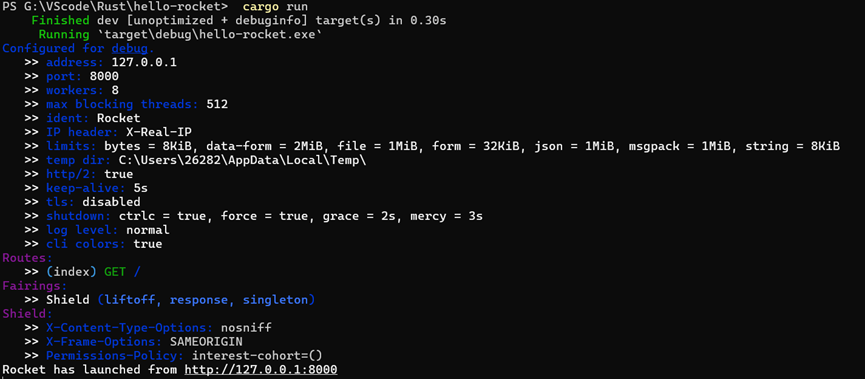
6. #[launch]

7. fn Rocket() -> \_ {

8. Rocket::build().mount("/", routes![index])

9. }

运行结果如下：



在了解Rocket应用程序之后，接下来我们将深入学习Rokcet框架中生存期（Lifecycle）、路由（Routing) ）、挂载（Mounting）、启动（Launching）、期货和同步（Futures and Async)的知识。

1. **生存期**

Rocket的主要任务是侦听传入的Web请求，将请求分派给应用程序代码，并向客户端返回响应。从请求到响应的过程称为生存期，包括路由、验证、加工和响应等步骤：

* 路由。Rocket将传入的HTTP请求解析为代码间接操作的本机结构。Rocket通过匹配应用程序中声明的路由属性来确定要调用的请求处理程序。
* 验证：Rocket根据匹配路由中存在的类型和防护来验证传入的请求。如果验证失败，Rocket会将请求转发到下一个匹配的路由或调用错误处理程序。
* 加工：与路由关联的请求处理程序使用经过验证的参数调用。这是应用程序的主要业务逻辑。
* 响应：返回的将得到处理。Rocket生成适当的HTTP响应并将其发送到客户端，这样就完成了生存期。Rocket继续侦听请求，重新启动每个传入请求的生存期。

1. **路由**

Rocket 应用程序以路线和处理程序为中心，路线是用于匹配传入请求的一组参数和处理请求并返回响应的处理程序的组合，处理程序只是一个函数，它接受任意数量的参数并返回任意类型。要匹配的参数包括静态路径、动态路径、路径段、表单、查询字符串、请求格式说明符和正文数据。路由是通过注释一个函数（处理程序）以及要匹配的参数集来声明的。

完整的路由声明如下所示：

1. #[get("/world")]              // <- 路由属性

2. fn world() -> &'static str {  // <- 请求处理程序

3. "hello, world!"

4. }

1. **挂载**

在Rocket可以向路由调度请求之前，需要挂载该路由：

Rocket::build().mount("/hello", routes![world]);

该mount方法以下输入：

path：用于在其下命名空间路由列表的基本路径，例如 /hello

routes：通过routes!宏定义的路由列表，可以包含多个路由，例如 routes![world]，也可以是多个路由的组合，例如routes![a, b, c]

1. **启动**

Rocket在启动后开始服务请求，它启动一个多线程异步服务器，并在请求到达时将请求分派到匹配的路由。Rocket有两种启动方法：

方法1，使用#[launch]路由属性。生成一个main设置异步运行时并启动服务器的函数，示例如下。

1. #[macro\_use] extern crate Rocket;

2. #[get("/world")]

3. fn world() -> &'static str {

4. "Hello, world!"

5. }

6. #[launch]

7. fn Rocket() -> \_ {

8.     Rocket::build().mount("/hello", routes![world])

9. }

方法2，使用#[Rocket::main]路由属性。#[Rocket::main]也生成一个main设置异步运行时的函数，但与#[launch]不同的是，它允许我们启动服务器，示例如下。

1. #[Rocket::main]

2. async fn main() -> Result<(), Rocket::Error> {

3. let \_Rocket = Rocket::build()

4.         .mount("/hello", routes![world])

5.         .launch()

6.         .await?;

7.     Ok(())

8. }

### 监听web请求

监听Web请求是指在服务器端等待并接收来自客户端的HTTP请求。服务器通过监听指定的网络地址和端口，以便能够接收到客户端发送的请求。监听web包括以下步骤，需要注意，这个步骤并不仅限于Rust语言的，不论是哪种语言，都可以通过以下步骤进行网络编程中的监听web请求。

1. 选择地址和端口。首先选择一个地址和端口来监听，地址可以是服务器的IP地址或域名，而端口是一个数字，用于标识服务器上的不同服务。常见的HTTP服务端口是80（非加密）和443（加密）。
2. 创建服务器。接下来创建一个服务器来监听Web请求，在不同的编程语言和框架中，创建服务器的方式可能会有所不同。通常情况下，需要使用相应的库或框架提供的API来创建服务器对象，并指定要监听的地址和端口。
3. 监听请求。一旦服务器启动并开始监听请求，它将等待客户端发送HTTP请求。服务器会持续监听指定的地址和端口，以便能够接收到来自客户端的请求。当服务器接收到请求时，它会触发相应的事件或调用你提供的处理函数来处理请求。
4. 处理请求。当服务器接收到请求后，会调用我们提供的处理函数或方法来处理请求，处理函数通常会根据请求的路径、方法、参数等信息来执行相应的逻辑，并生成一个HTTP响应。处理函数可以访问请求的内容、头部、参数等信息，并根据需要进行相应的处理。生成响应。在处理请求的过程中，需要生成一个HTTP响应来回复客户端，HTTP响应通常包括状态码、响应头和响应体。可以根据请求的内容和服务器的逻辑，生成不同的响应来满足客户端的需求。响应可以是HTML、JSON、文件等不同类型的数据。
5. 返回响应。处理函数应该返回一个包含响应的对象或数据结构，这个响应将被服务器发送回客户端作为对请求的回应，响应可以包含状态码、响应头和响应体等信息。客户端将根据响应的内容和状态码来处理服务器的回应。
6. 运行服务器。使用相应的命令或工具来启动服务器，一旦服务器运行起来，它将开始监听指定的地址和端口，并等待客户端发送HTTP请求。

下面将详细讲述Rust如何实现监听Web请求。编写代码之前，首先要明确目标，我们要实现的功能如下：监听指定端口（如8080）上的HTTP请求；当有请求过来时,根据请求方法（GET/POST等）和路径，调用不同的处理函数进行响应；在处理函数中读取请求体，设置响应头和响应体返回给客户端。

下面我们开始着手代码的编写，首先导入所需要的库文件：

1.use hyper::{Body, Response, Server};

2.use hyper::service::{make\_service\_fn, service\_fn};

3.use std::convert::Infallible;

接下来要写一个处理函数来处理web响应请求。在处理函数中，可以根据请求的路径、方法等信息来生成相应的响应。函数的签名如下：

async fn handle\_request(req: Request<Body>) ->Result<Response<Body>, Infallible>

函数代码如下：

1.async fn handle\_request(\_: hyper::Request<Body>) -> Result<Response<Body>, Infallible> {

 2.   let response\_body = Body::from("Hello, World!");

 3.   let mut response = Response::new(response\_body);

  4.  \*response.status\_mut() = hyper::StatusCode::OK;

  5. response.headers\_mut().insert(

  6.      hyper::header::CONTENT\_TYPE,

  7.      hyper::header::HeaderValue::from\_static("text/plain"),

  8. );

  9. Ok(response)

}

10.#[tokio::main]

11.async fn main() {

   12.let make\_svc = make\_service\_fn(|\_conn| {

   13.     async {

   14.         Ok::<\_, Infallible>(service\_fn(handle\_request))

   15.     }

   15.   });

   16. let addr = ([127, 0, 0, 1], 3000).into();

   17. let server = Server::bind(&addr).serve(make\_svc);

   18. println!("Server running at HTTP://{}", addr);

   19. if let Err(e) = server.await {

   20.     eprintln!("server error: {}", e);

    }

}

然后，创建一个服务器来监听Web请求并调用处理函数。

1.    #[tokio::main]

2.    async fn main() {

3.        // 创建服务

4.        let make\_svc = make\_service\_fn(|\_conn| {

5.            async {

6.                Ok::<\_, Infallible>(service\_fn(handle\_request))

7.            }

8.        });

9.        // 绑定地址和端口，并启动服务器

10.        let addr = ([127, 0, 0, 1], 3000).into();

11.        let server = Server::bind(&addr).serve(make\_svc);

12.      // 打印服务器地址

13.        println!("Server listening on HTTP://{}", addr);

14.

15.        // 等待服务器运行

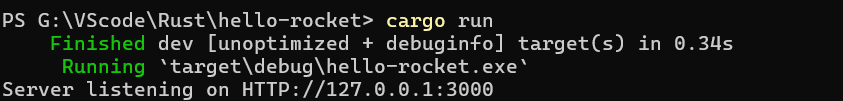
16.        if let Err(e) = server.await {

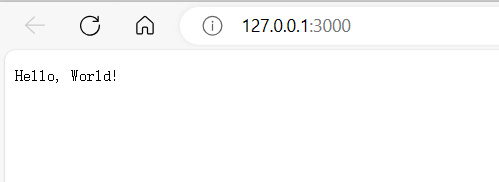
17. eprintln!("Server error: {}", e);

28. }

19.    }

至此一个监听程序就创建好了，运行的结果如下：





以上代码的主要作是监听并响应来自浏览器或其他客户端的HTTP请求。代码使用actix\_web框架，它提供了非阻塞的Web服务器功能。调用HTTPServer::new()创建服务器实例。调用.bind()绑定端口监听。定义路由，使用get()匹配GET请求，post()匹配POST请求等。给每个路由绑定对应的处理函数,函数签名要匹配Handler<\_> trait。在处理函数中可以通过HTTPRequest::body()读取请求体，HTTPResponse::header()设置响应头，HTTPResponse::body()设置响应体。最后调用.run()运行服务器。

客户端发送请求后服务器会调用对应的路由函数进行处理响应，使用到的方法和库如表6.6所示。

表6.6 使用的方法和库

|  |  |
| --- | --- |
| 方法和库 | 介绍 |
| Hyper::Body | Body是Hyper库中的一个类型，表示HTTP请求或响应的消息体。它可以是一个字节流、一个字符串或一个文件等。在示例中，使用Body::from方法创建了一个包含响应体内容的Body对象。 |
| Hyper::Request | Request是Hyper库中的一个类型，表示HTTP请求。它包含了请求的方法、路径、头部和消息体等信息。在示例中，处理函数的参数req的类型被指定为Request<Body>，表示接收到的HTTP请求。 |
| Hyper::Response | Response是Hyper库中的一个类型，表示HTTP响应。它包含了状态行、响应头和响应体等信息。在示例中，处理函数的返回类型被指定为Result<Response<Body>, Infallible>，表示处理函数返回一个可能包含响应的Result类型。 |
| Hyper::Server | Server是Hyper库中的一个类型，用于创建和启动HTTP服务器。在示例中，使用Server::bind方法绑定了服务器的地址和端口，并使用Server::serve方法启动了服务器。 |
| Hyper::service::make\_service\_fn | make\_service\_fn是Hyper库中的一个函数，用于创建一个服务工厂。在示例中，使用make\_service\_fn函数创建了一个服务工厂，该工厂会在每个连接上创建一个新的服务。 |
| Hyper::service::service\_fn | service\_fn是Hyper库中的一个函数，用于创建一个服务函数。在示例中，使用service\_fn函数创建了一个服务函数，该函数会处理每个HTTP请求并生成相应的HTTP响应。 |
| std::convert::Infallible | Infallible是Rust标准库中的一个类型，表示不可能发生错误的类型。在示例中，处理函数的返回类型中使用了Infallible，表示处理函数不会返回错误。 |
| tokio::main | tokio::main是tokio库中的一个宏，用于定义一个异步的main函数。在示例中，使用tokio::main宏来标记main函数为异步函数。 |

这些方法和库的组合使用，实现了一个简单的HTTP服务器，能够接收HTTP请求并返回一个包含"Hello, World!"的HTTP响应。

### 发送HTTP响应

在互联网时代，绝大多数人都参与其中，这就造成了网络环境中用户极多，需要同时处理的信息极为繁杂。但是我们发现平时访问网站好像也没有遇到很大的困难，这是因为网络中信息的传输按照严格的规则：当自己（客户端）想要访问网站时，会通过网络对目标（服务器）发送一个请求，然后服务器会将这个请求处理后发送回客户端。这就是一个简单的网络交流过程，也就是我们要学习的HTTP响应。

1. **HTTP响应**

当客户端发送HTTP请求到服务器时，服务器会根据请求进行处理，并生成一个HTTP响应返回给客户端。在这个过程中，服务器和客户端之间的响应遵循一定的格式。

HTTP响应由以下几个部分组成：

* 状态行（Status Line）：访问网页时，时常会碰到网页显示404 Not Found的情况，这就是状态行中的一种状态码。状态行包含了HTTP协议版本号、状态码和状态消息。如：HTTP/1.1 200 OK。状态码表示服务器对请求的处理结果，常见的状态码有：200表示成功，404表示资源未找到，500表示服务器内部错误等。
* 响应头（Response Headers）：响应头是HTTP响应的一部分，它包含了关于响应的元数据信息。响应头以键值对的形式表示，每个键值对占据一行，键和值之间使用冒号进行分隔。常见的有以下类型：
* Content-Type：指定响应体的内容类型。它告诉客户端如何解析和处理响应体的数据。常见的Content-Type值如表6.7所示：

表6.7 Content-type值

|  |  |
| --- | --- |
| 常见的值 | 介绍 |
| text/html | 表示响应体是HTML文档。 |
| application/json | 表示响应体是JSON数据。 |
| mage/jpeg | 表示响应体是JPEG格式的图片。 |
| Content-Length | 指定响应体的长度，以字节为单位。客户端可以使用Content-Length字段来确定响应体的大小。 |
| Date | 指定响应生成的日期和时间。它用于告知客户端响应的生成时间。 |
| Server | 指定服务器的软件和版本信息。它告诉客户端响应是由哪个服务器生成的。 |

* Cache-Control：指定响应的缓存策略。它可以控制客户端和中间代理服务器对响应的缓存行为。
* Set-Cookie：用于在响应中设置Cookie。服务器可以使用Set-Cookie字段将Cookie信息发送给客户端。
* Location：用于重定向响应。当服务器需要将客户端重定向到另一个URL时，可以使用Location字段指定重定向的目标URL。
* 空行（Blank Line）：空行用于分隔响应头和响应体。
* 响应体（Response Body）：响应体包含了实际的响应数据，可以是HTML、JSON、图片等。响应体的内容长度可以通过响应头中的Content-Length字段来确定。

1. **HTTP响应在Rust中实现**

在Rust中，可以使用Rocket框架和Hyper库来实现HTTP响应，具体步骤如下。

1. 导入所需的库：导入Hyper库和其他必要的依赖项。
2. 定义处理请求的异步函数：在handle\_request函数中，您可以编写处理HTTP请求的逻辑。在这个示例中，它简单地返回一个包含"Hello, World!"的响应体。
3. 创建响应对象：使用Response::new函数创建一个响应对象，并将响应体传递给它。
4. 设置状态码和响应头：使用status\_mut方法设置响应的状态码，使用headers\_mut方法插入响应头。
5. 创建服务：使用make\_service\_fn函数创建一个服务，该服务将在每个连接上调用handle\_request函数。
6. 绑定地址并启动服务器：使用Server::bind函数绑定服务器的地址和端口，并使用serve方法启动服务器。
7. 打印服务器地址：使用println!宏打印服务器的地址。
8. 处理服务器错误：使用server.await等待服务器运行，并处理可能的错误。

首先将当前项目环境切到 nightly 版本：

1. // 将当前项目环境切到 nightly 版本

2. Rustup override set nightly && Rustup update

接下来导入Rocket的crate依赖

1. cargo add Rocket //命令行执行

可以在toml文件中的dependencies属性中找到以下信息：

1. Rocket= "0.4.0"

接着导入Hyper库，可以直接在toml文件中的dependencies属性中添加：

1. Hyper= "0.14"

最后是代码部分：

1.use hyper::{Body, Request, Response, Server};

2.use hyper::service::{make\_service\_fn, service\_fn};

3.use std::convert::Infallible;

4.use hyper::Method;

5.use hyper::StatusCode;

6.

7.async fn handle\_request(req: Request<Body>) -> Result<Response<Body>, Infallible> {

8.   match (req.method(), req.uri().path()) {

9.        (&Method::GET, "/") => {

10.            let response\_body = Body::from("Hello, World!");

11.            let response = Response::new(response\_body);

12.            Ok(response)

13.        }

14.        \_ => {

15.            let response\_body = Body::from("Not Found");

16.            let response = Response::builder()

17.                .status(StatusCode::NOT\_FOUND)

18.                .body(response\_body)

19.                .unwrap();

20.            Ok(response)

21.        }

22.    }

23.}

24.#[tokio::main]

25.async fn main() {

26.    let make\_svc = make\_service\_fn(|\_conn| {

27.        async {

28.            Ok::<\_, Infallible>(service\_fn(handle\_request))

29.        }

30.    });

31.

32.    let addr = ([127, 0, 0, 1], 3000).into();

33.    let server = Server::bind(&addr).serve(make\_svc);

34.

35.    println!("Server listening on HTTP://{}", addr);

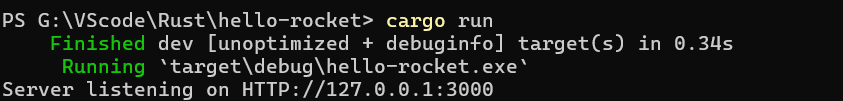
36.

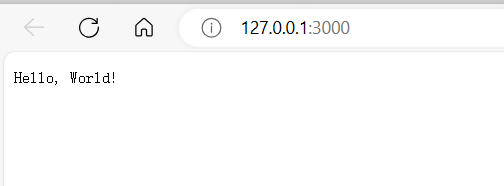
37.    if let Err(e) = server.await {

38.        eprintln!("Server error: {}", e);

40.    }

在终端中使用cargo build和cargo run命令编译运行，运行的结果会打印在终端：





现在，我们的Rust项目将启动一个HTTP服务器，监听在本地地址的端口3000上。当有HTTP请求到达时，服务器将返回一个包含"Hello, World!"的HTTP响应。

示例中使用的方法和组件如表6.8所示。

表6.8 Content-type值

|  |  |
| --- | --- |
| 方法和库 | 介绍 |
| Hyper::Body | Body是Hyper库中的一个类型，表示HTTP请求或响应的消息体。它可以是一个字节流、一个字符串或一个文件等。在示例中，使用Body::from方法创建了一个包含响应体内容的Body对象。 |
| Hyper::Response | Response是Hyper库中的一个类型，表示HTTP响应。它包含了状态行、响应头和响应体等信息。在示例中，使用Response::builder方法创建了一个ResponseBuilder对象，然后使用ResponseBuilder对象的方法设置了状态码、响应头和响应体等信息，最后调用ResponseBuilder对象的build方法构建了一个完整的Response对象。 |
| Hyper::Server | Server是Hyper库中的一个类型，用于创建和启动HTTP服务器。在示例中，使用Server::bind方法绑定了服务器的地址和端口，并使用Server::serve方法启动了服务器。 |
| Hyper::service::make\_service\_fn | make\_service\_fn是Hyper库中的一个函数，用于创建一个服务工厂。在示例中，使用make\_service\_fn函数创建了一个服务工厂，该工厂会在每个连接上创建一个新的服务。 |
| Hyper::service::service\_fn | service\_fn是Hyper库中的一个函数，用于创建一个服务函数。在示例中，使用service\_fn函数创建了一个服务函数，该函数会处理每个HTTP请求并生成相应的HTTP响应。 |

通过将这些方法和组件组合使用，实现了一个简单的HTTP服务器，能够接收HTTP请求并返回一个包含"Hello, World!"的HTTP响应。

## Reqwest库

### Reqwest库简介

Reqwest库是一个基于Rust编写的HTTP客户端库，用于发送 HTTP 请求和处理响应，它提供了简单易用的API，支持异步和同步请求，并且可以与tokio和async-std集成，下面我们首先了解一下Reqwest的一些主要特性：

1. **简单易用**

Reqwest提供了一个简单的API，使得发送HTTP请求变得非常容易。它支持GET、POST、PUT、DELETE等常见的HTTP方法，并且可以利用header方法来设置请求头，headers方法用于获取响应的头部信息、请求体和查询参数。

1. **支持同步和异步**

Reqwest既支持同步的HTTP请求，也支持异步的HTTP请求。对于同步请求，可以使用*Reqwest::blocking*模块；对于异步请求，可以使用*Reqwest::Client*结构。

1. **功能强大**

Reqwest支持HTTPS、重定向、代理、Cookie等常见的HTTP功能，还提供了一些高级功能，如连接池、连接超时、请求超时等。

1. **高性能。**

Reqwest基于Hyper库构建，使用了高效的异步I/O模型，可以处理大量的并发请求。

1. **良好的文档和社区支持**

Reqwest库有详细的文档，包括使用指南、API文档和示例代码。此外，它还有一个活跃的社区，可以提供支持和解答问题。

Reqwest库提供了发送HTTP请求和处理HTTP响应的功能，可用于构建Web应用程序，与服务器进行通信并获取所需要的数据。Reqwest还可以用于网络爬虫，向目标网站发送HTTP请求，等待服务器响应并获取到服务器返回的网页内容，然后对所获取到的网页内容进行解析，并将解析后的数据保存到数据库或者其他数据存储介质中，网络爬虫可广泛应用于电子商务、旅游预订、招聘信息收集、航班和酒店价格比较、影视资源搜索等领域。此外，Reqwest库库提供了身份验证和授权的功能，可以被用于身份验证和授权，发送带有身份验证信息的HTTP请求，与需要授权的服务进行交互。

接下来我们就正式进入到Reqwest库的学习，首先从查询公网ip开始学习如何在程序中添加依赖并且引入Reqwest库；然后依次学习HTTP中一些常见的请求方法，如get、post、put、delete等；紧接着我们会学习json是如何与Reqwest库相结合一起使用；之后我们再来简单学习cookie，最后利用toiko并调用Reqwest库构建处理HTTP请求的服务器。

### 安装与引用

下例通过一个返回请求者ip的例子，演示Rust引用Reqwest库的使用方法。

首先在项目的Cargo.toml中添加依赖：

1. [dependencies]

2. Reqwest = { version = "0.11", features = ["json"] }

3. tokio = { version = "1", features = ["full"] }

引用Reqwest库，查询自己所在网络的公网ip地址：

1. use std::collections::HashMap;

2. #[tokio::main]

3. async fn main() -> Result<(), Box<dyn std::error::Error>> {

4. let resp = Reqwest::get("HTTPs://HTTPbin.org/ip")

5. //HTTPbin.org 这个网站能测试 HTTP 请求和响应的各种信息，比如cookie、ip、//headers 和登录验证等，且支持 GET、POST 等多种方//法，对 web 开发和测试很有帮助。

6. .await?

7. .json::<HashMap<String, String>>()

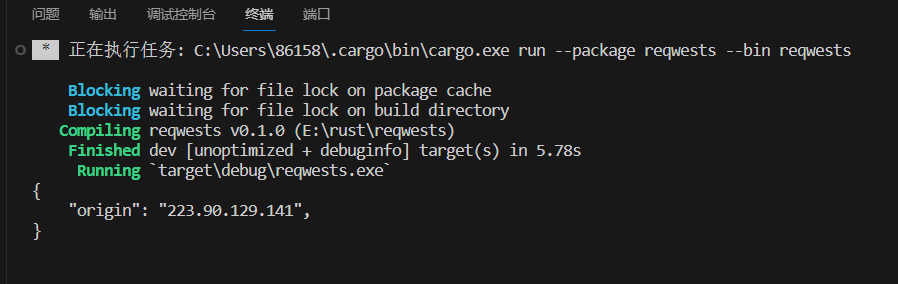
8. .await?;

9. println!("{:#?}", resp);

10. Ok(())

11. }

运行结果如下图所示。



### 一些常见的HTTP请求方法

下例演示如何使用Reqwest库发送一个GET请求并处理响应，程序的主要功能是发送一个GET请求到https://www.baidu.com并打印相应的内容：

1. use Reqwest::Error;

2. #[tokio::main]

3. async fn main() -> Result<(), Error> {

4. let response = Reqwest::get("HTTPs://www.baidu.com")

5. .await?

6. .text()

7. .await?;

8. println!("{}", response);

9. Ok(())

10. }

运行结果如下图所示：



本例使用了Rust的*Reqwest*库来发送HTTP请求并获取响应，其中async/.await是Rust的特殊语法，在程序发生阻塞时，其让放弃当前的线程的控制权成为可能，在等待操作完成时允许其它代码执行。首先，程序使用*use*语句导入了*Reqwest::Error*，这是*Reqwest*库中用于表示错误的类型。接下来，程序使用*#[tokio::main]*宏来指定*main*函数在异步上下文中运行。这意味着*main*函数可以使用异步操作，如*await*关键字。在*main*函数中，程序使用*Reqwest::get*函数发送一个GET请求到*HTTPs://www.baidu.com*。这个函数返回一个*Result*类型，其中*Ok*分支包含了一个*Response*对象，表示请求的响应，而*Err*分支包含了一个*Error*对象，表示请求过程中发生的错误。接着，程序使用*await*关键字等待请求的响应，并使用*.text()*方法将响应的内容读取为文本格式。最后，程序打印响应的内容，并返回一个*Result<(), Error>*类型的值，表示程序执行成功或失败。

爬取的内容和实际网页所显示源代码的内容并不完全相同，爬取到的只是一小部分，因为百度网页会检测到这是非法爬虫操作，并采取一系列措施会阻碍爬取到全部内容。那怎么样才能爬取与百度相同的内容呢? HTTP请求头中有这样一个字段User-Agent，用于标识发送请求的客户端应用程序、操作系统和设备信息。它允许服务器根据不同的客户端类型或版本来返回不同的内容或采取不同的行为，其可以用来进行客户端识别,服务器可以根据User-Agent字段来判断请求是来自浏览器、移动应用程序还是其他类型的客户端，因此我们可以用这个字段来模拟浏览器发出的请求并作出响应，代码示例如下面所示：

1. use Reqwest::header::{HeaderMap, USER\_AGENT};

2. use Reqwest::Client;

3.  #[tokio::main]

4. async fn main() -> Result<(), Box<dyn std::error::Error>> {

5.     // 创建一个HTTP客户端

6.     let client = Client::new();

7.    // 创建一个自定义的HeaderMap

8.     let mut headers = HeaderMap::new();

9.     headers.insert(USER\_AGENT, "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36 Edg/118.0.2088.61".parse().unwrap());

10.     // 发送GET请求，并设置自定义的User-Agent头部字段

11.     let response = client

12.         .get("HTTPs://www.baidu.com")

13.         .headers(headers)

14.         .send()

15.         .await?;

16.     // 处理响应

17.     let body = response.text().await?;

18.     println!("{}", body);

19.     Ok(())

20. }

部分运行结果如下图所示：



首先使用Client::new()方法创建一个HTTP客户端对象；然后创建一个空的HeaderMap对象，并使用insert方法插入一个自定义的User-Agent头部字段，这里使用了USER\_AGENT常量来表示User-Agent字段；接着使用client对象的get方法创建一个GET请求，并使用headers方法设置自定义的头部字段；然后调用send方法发送请求，并使用await关键字等待响应最后使用response对象的text方法获取响应的文本内容，并将其打印出来。

在*Reqwest*库中，*post*方法用于发送HTTP POST请求。它的函数签名如下：

1. pub async fn post(

2. url: impl IntoUrl,

3. ) -> Result<Response, Error>

4.

*post*方法接受一个参数*url*，该参数可以是一个字符串类型的URL，也可以是一个实现了*IntoUrl* trait的类型。*IntoUrl* trait可以将不同类型的值转换为URL。post方法返回一个*Result*类型，其中*Ok*分支包含了一个*Response*对象，表示请求的响应，而*Err*分支包含了一个*Error*对象，表示请求过程中发生的错误。

以下是一个使用*post*方法发送POST请求的示例：

1. use Reqwest::Error;

2. #[tokio::main]

3. async fn main() -> Result<(), Error> {

4. let client = Reqwest::Client::new();

5. let response = client

6. .post("HTTPs://HTTPbin.org/post")

7. .body("Hello, world!")

8. .send()

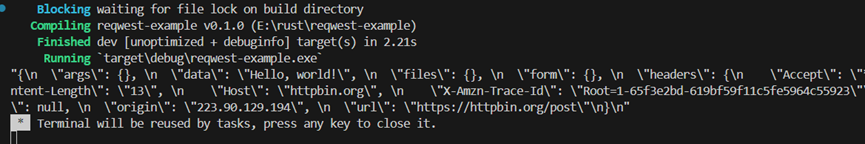
9. .await?;

10.   println!("{:?}", response.text().await?);

11. Ok(())

12. }

运行结果如下。



在这个示例中，我们首先创建了一个*Client*对象，这是*Reqwest*库中用于发送HTTP请求的主要类型。然后，我们使用*post*方法指定了要发送的POST请求的URL。接着，我们使用*body*方法设置请求的主体内容，这里是一个字符串*"Hello, world!"*。最后，我们使用*send*方法发送请求，并使用*await*等待响应。

Get请求和post请求有什么区别呢？首先是在参数传递方式上，GET请求将参数以查询字符串的形式附加在URL后面，请求的参数暴露在URL中，可以被浏览器缓存、历史记录等保存，不适合传递敏感信息，而POST请求的参数在请求体中，相对安全一些。其次，在参数长度限制方面中，GET请求对URL长度有限制，不同浏览器和服务器的限制不同，一般在2KB到8KB之间。POST请求没有长度限制，但是服务器可能对请求体的大小进行限制。最后，在幂等性方面：GET请求是幂等的，多次请求同一个URL的结果是相同的。POST请求不是幂等的，多次请求同一个URL可能会产生不同的结果。因此要注意的是GET请求适合获取数据，不会对服务器产生影响而POST请求适合提交数据，可能对服务器产生影响，例如创建、更新或删除资源。

接下来我们来看put和delete方法：

1. use reqwest::Client;

2. use serde\_json::json;

3.

4. #[tokio::main]

5. async fn main() -> Result<(), reqwest::Error> {

6.     let client = Client::new();

7.     let url = "https://HTTPbin.org/put";

8.     let response = client.put(url)

9.         .json(&json!({

10.             "title": "Updated Title",

11.             "body": "Updated Body"

12.         }))

13.         .send()

14.         .await?;

15.

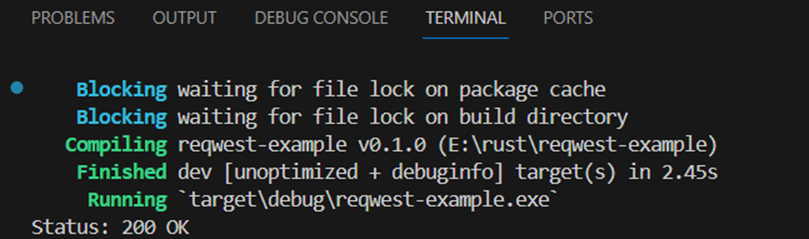
16.     println!("Status: {}", response.status());

17.     Ok(())

18. }

19.

运行结果如下。



我们使用reqwest::Client创建了一个HTTP客户端，然后使用put方法发送了一个PUT请求到指定的URL。我们还使用json!宏来创建一个JSON对象作为请求的body数据。运行程序，它会发送一个PUT请求到指定的URL，并打印出响应的状态码。put和delete方法类似，在此不再赘述。

### JSON

Reqwest是一个用于发送HTTP请求的Rust库。它支持发送JSON请求和解析JSON响应。在Reqwest中，可以使用*json!*宏来生成JSON对象。*json!*宏接受一个类似于JavaScript对象的语法，用于创建一个serde\_json::Value对象，表示一个JSON值。在Reqwest库中，json方法的作用是将请求体序列化为JSON格式。它可以接受任何可以被序列化为JSON的值，并将其转换为合适的请求体格式。使用json方法，可以将请求体直接传递给Reqwest的请求构建器（RequestBuilder）的方法，例如post、put、patch等。它会自动将请求体序列化为JSON，并设置请求的Content-Type头为application/json。

序列化和反序列化首先要在cargo.toml里添加依赖:

1. serde = { version = "1.0.144", features = ["derive"] }

2. serde\_json = "1.0.85"

以下是一个使用Reqwest库的json方法发送POST请求的示例代码。

1. use reqwest::Client;

2.  use serde\_json::json;

3.   #[tokio::main]

4.  async fn main() -> Result<(), Box<dyn std::error::Error>> {

5.     let client = Client::new();

6.     let url = "HTTPs://HTTPbin.org/post";

7.      let body = json!({

8.         "男生": "张三",

9.          "女生": "小红"

10.     });

11.      let response = client

12.         .post(url)

13.         .json(&body)

14.         .send()

15.         .await?;

16.    // 处理响应

17.      if response.status().is\_success() {

18.         let json\_response: serde\_json::Value = response.json().await?;

19.         // 在这里处理JSON响应

20.         println!("Response: {:?}", json\_response);

21.     } else {

22.         println!("Request failed with status code: {}", response.status());

23.      }

24.      Ok(())

25.  }

26.

运行结果如下。



在上面的示例中，我们首先创建了一个Reqwest的Client对象。然后，我们定义了要发送请求的URL和请求体。我们使用json!宏创建了一个serde\_json::Value对象，表示一个包含两个键值对的JSON对象。接下来，我们使用post方法创建了一个POST请求，并使用json方法将请求体设置为我们刚刚创建的JSON对象。最后，我们使用*send*方法发送请求，并使用*json*方法将响应解析为serde\_json::Value对象并处理JSON响应。

### cookie

HTTP Cookie（也叫 Web Cookie 或浏览器 Cookie）是服务器发送到用户浏览器并保存在本地的一小块数据。浏览器会存储 cookie 并在下次向同一服务器再发起请求时携带并发送到服务器上。通常，它用于告知服务端两个请求是否来自同一浏览器——如保持用户的登录状态。Cookie 使基于[无状态](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Overview#http_是无状态，有会话的)的 HTTP 协议记录稳定的状态信息成为可能。值得注意的是，使用cookie存储的信息往往是安全性较低或者容易被窃取的。因此不建议利用cookie存储密码、隐私性较强的重要信息。下面演示Reqwest库引入cookie特性示例。

首先引入cookie特性

1. Reqwest = { version = "0.11.9",

2. features = ["json", "cookies"] }

然后自定义client指定相关配置，开启cookie储存权限，并可选择修改重定向次数

1. let client = Reqwest::Client::builder()

2. // redirect（重定向）默认是10，可以修改，本例修改为2次

3. .redirect(Policy::limited(2))

4. // 开启cookie支持

5. .cookie\_store(true)

6. .build()

7. .unwrap();

8. let res = client.get(url.trim())

9. .send()

10. .await

11. .unwrap();

Cookie 主要用于以下三个方面：

1. 会话状态管理，如用户登录状态、购物车、游戏分数或其他需要记录的信息管理；
2. 个性化设置，如用户自定义设置、主题和其他设置等；
3. 浏览器行为跟踪，如跟踪分析用户行为等。

Reqwest库中，可以使用cookie\_store方法来启用自动存储和发送会话Cookie。该方法是在ClientBuilder上调用的，用于配置Client的Cookie存储和发送行为。

### Tokio

前面我们学习了Reqwest库的基本用法，近几年Reqwest增加了使用tokio的异步编程支持，Tokio是[Rust](https://so.csdn.net/so/search?q=Rust&spm=1001.2101.3001.7020)编程语言的一个异步运行时间(async runtime)，它提供了编写网络应用所需的构建模块。异步编程对于网络编程的性能有重要的提高作用。除此之外，tokio在线程池操作中有独特的worksteal特性，大大提高了线程排队任务处理的稳定性与效率，从有几十个内核的大型服务器到小型嵌入式设备的系统利用tokio都能得到效率上的提高。

Tikio有以下组件：一个用于执行异步代码的多线程运行时间；一个标准库的异步版本；一个庞大的库的生态系统。

Tikio的优势为：

1. **快速。**Tokio建立在async/await语言特性之上，可以增加并发操作的数量、扩展大量的并发任务，快速处理任务。
2. **可靠。**Tokio注重提供一致的行为，其目标是让用户部署可预测的软件，在没有意外的情况下，Tikio能保证其每天都有相同的表现，有可靠的响应时间，没有不可预测的延迟峰值，保证安全性。
3. **易用。**有了Rust的async/await功能，编写异步应用程序的复杂性大大降低，因此Tokio的程序具有很强的实用性且配有充满活力的生态系统，配合Rust强大的所有权系统，使用tokio非常简单。
4. **灵活。**Tokio提供了运行时的多种变化，从多线程的、减重的运行时到轻量级的、单线程的运行时，每个运行时都允许用户根据自己的需要进行调整，大大增加了系统的灵活性。

为了方便大家掌握Tokio的用法，下面给出利用toiko构建处理HTTP请求的服务器示例，本例利用tokio异步执行框架，实现客户端的构建。

1. //在Rust清单文件中对于tokio与其他项目进行配置

2. [package]

3. name = "Reqwest-async"

4. version = "0.1.0"

5. authors = ["Foo <foo@bar.com>"]

6. [dependencies]

7. serde\_json = "1.0.6"

8. serde = "1.0.21"

9. serde\_derive = "1.0.21"

10. futures = "0.1.17"

11. //引入tioko版本及特性

12. tokio-core = "0.1.10"

13. [dependencies.Reqwest]

14. version = "0.8.1"

15. features = ["unstable"]

调用Reqwest库与tokio内核

1. extern crate

2. #[macro\_use]

3. extern crate

4. serde\_json;

5. serde\_derive;

6. extern crate Reqwest;

7. extern crate futures;

8. extern crate tokio\_core;

9. use futures::Future;

10. use tokio\_core::reactor::Core;

11. use Reqwest::unstable::async::{Client, Decoder};

12. use std::mem;

13. use std::io::{self, Cursor};

14. use futures::Stream;

15. #[derive(Debug, Serialize, Deserialize)]

16. //利用结构体定义储存post所需的信息

post请求的构建

1. struct Post {

2. title: String,

3. body: String,

4. pinned: bool,

5. }

服务器的构建与实现

1. fn main() {

2. let mut core = Core::new().expect("Could not create core");

3. println!("core createing sucessfully");

4. let url = "HTTP://localhost:8000/posts";

5. let post: Post = Post {

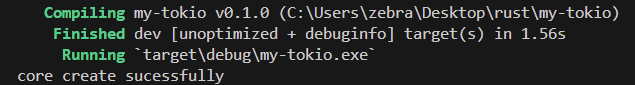
6. title: "Testing this".to\_string(),

7. body: "Try to write something".to\_string(),

8. pinned: true,

9. };

运行结果：



本例中，我们通过对于post请求的定义，利用tokio执行框架，完成对于HTTP请求的实现，相较于传统实现方法，本例应具有更高的性能。值得注意的是，tokio相关的API仍在Reqwest::unstable里，从名字中可以看出，这些API还不稳定。在使用时应综合考虑，根据项目特性灵活使用。

## 小结

在本章中，我们探讨了很多关于如何使用Rust 构建Web 应用程序的内容，它有很多高质量的软件包供我们选择，很容易上手。大多数Web 框架都是由解释型的动态语言主导的，它们可能会占用大量CPU 资源但执行性能不高，但是很方便构建Web 应用程序。而Rust作为一种编译型语言，不需要解释器，编写的Web 应用程序在运行时占用的空间很少。Rust即可以让我们获得动态语言开发的体验，又具备类似C 语言的高效性，这对Web 开发者来说是一个不错的选择。

# 其他应用层协议编程

本章要点：了解RPC工作原理，熟悉gRPC，掌握通过编程方式发送电子邮件的库调用lettre，学习编写一个简单的FTP客户端和TFTP服务器。

## 引言

在之前的几章中，我们看到网络中的两个主机通过流或离散的数据包交换字节。通常，更高层次的应用程序需要处理这些字节，以使其对应用程序有意义。这些应用程序在传输层之上定义了一个新的协议层，通常被称为应用层协议。

Rust本身并不直接提供RPC（远程过程调用）、SMTP（简单邮件传输协议）或FTP（文件传输协议）等应用层协议的实现。这些协议是独立于编程语言的，它们定义了如何在网络上传输数据和进行通信的规则。Rust可以通过标准库和第三方库来实现这些协议。例如，对于RPC，Rust有一些流行的库，如grpc-rs（用于实现gRPC协议）和tonic（一个轻量级的RPC框架）；对于SMTP和FTP，Rust也有相应的库可以使用，如rust-smtp和rust-ftp等。

需要注意的是，虽然Rust有这些库的支持，但在实际使用中还需要根据具体需求进行选择和配置。此外，由于Rust的并发和异步编程能力强大，因此在实现这些协议时可以考虑使用异步编程模型来提高性能和响应速度。在本章中，我们将研究其中的一些协议。

## 远程过程调用RPC

### RPC和协议缓冲区

当涉及到不同系统、语言和平台之间的通信时，RPC（远程过程调用）和协议缓冲区（Protocol Buffers）是两种常用的技术。

RPC是一种通信协议，允许程序在不同的计算机或进程之间进行远程调用。它使得分布式架构下的系统可以像调用本地函数一样调用远程函数，无需关注底层网络通信细节。RPC的优点之一是跨语言支持，不同的编程语言都有相应的RPC框架可以使用。请求方将参数打包成一段二进制数据发送到远程服务端，服务端返回经过编码的二进制数据。RPC框架负责封装和解析这些数据，实现跨网络的通信。常见的RPC框架有gRPC、Apache Thrift和Dubbo等，它们使用不同的序列化方式和传输协议。

协议缓冲区是一种序列化结构化数据的机制，旨在实现高效的数据交换和存储。使用协议缓冲区，可以定义数据的结构和字段类型，并生成相应的代码用于数据的序列化和反序列化。与其他序列化格式相比，协议缓冲区具有更小的消息体积和更快的序列化和反序列化速度。协议缓冲区的数据具有自描述性，可读性较高。此外，它还支持数据版本兼容性，即使在数据结构发生变化时，仍能够正常解析旧版本的数据。协议缓冲区可以与RPC相结合使用，作为RPC的序列化和传输格式。在RPC中，请求和响应的数据可以使用协议缓冲区定义的消息类型进行编码和解码，实现高效、可扩展和跨语言的通信。

总而言之，RPC和协议缓冲区是用于实现跨系统、语言和平台之间的远程通信的常用技术。RPC简化了远程调用的过程，使得分布式系统开发更加便捷；协议缓冲区提供了一种高效的数据序列化和传输方式，确保数据的可靠性和可扩展性。它们的结合可以实现高效、可扩展和跨语言的远程通信。

1. **使用协议缓冲区定义RPC消息格式**

当使用Rust语言开发分布式系统时，常使用RPC来进行跨网络的函数调用。可以使用协议缓冲区（protobuf）定义RPC消息的格式。使用步骤如下：

1. 首先，在Rust项目中添加 protobuf crate 的依赖；
2. 然后创建一个.proto文件来定义RPC消息的格式；
3. 接下来，使用 protoc 命令将 .proto 文件编译为Rust代码。
4. 最后在命令行中运行命令：protoc --rust\_out=. message.proto，这将生成一个名为 message.rs 的Rust模块，其中包含了根据 .proto 文件生成的代码。
5. **在Rust中使用协议缓冲区进行序列化与反序列化**

当在Rust中进行网络通信或跨服务的数据交换时，序列化和反序列化是非常重要的步骤。在Rust中使用协议缓冲区进行序列化和反序列化是一种广泛应用于数据交换和存储的重要技术。序列化指将结构化的数据对象转换为字节流的过程，而反序列化则是将字节流重新还原为数据对象的过程。通过序列化和反序列化，我们可以在不同的系统、语言和平台间高效地传输和存储数据。通过在Rust中使用协议缓冲区进行序列化和反序列化，我们可以轻松地实现结构化数据的高效传输和存储，这为不同系统、语言和平台之间的数据交换提供了便利，同时也提升了应用程序的性能和可维护性。

为了简化这个过程，我们可以使用协议缓冲区。在Rust中，我们可以使用 protobuf crate 来方便地在代码中使用协议缓冲区。

首先，在Rust项目的 Cargo.toml 文件中添加 protobuf crate 的依赖：

1. [dependencies]

2. protobuf = "2.11"

然后，创建一个 .proto 文件来定义要序列化的消息格式，创建一个名为 message.proto 的文件，定义一个简单的消息格式：

1. syntax = "proto3";

2. message Person {

3. string name = 1;

4. int32 age = 2;

5. }

本例定义了一个 Person 消息，它包含一个字符串字段 name 和一个整数字段 age。然后，我们使用 protoc 命令将 .proto 文件编译为Rust代码。

接下来在命令行中运行以下命令：

1. protoc --rust\_out=. message.proto

这将生成一个名为 message.rs 的Rust模块，其中包含了根据 .proto 文件生成的代码。



这是根据给定的 .proto 文件生成的 Rust 代码。在这个文件中，我们定义了一个名为 Person 的消息结构体，其中包含了一个 name 字段和一个 age 字段，它们分别是字符串类型和整数类型。这些字段使用 Protocol Buffers 的 prost 宏进行了标注，以便在序列化和反序列化过程中使用。

1. **不同的编码与解码方案及其在RPC中的应用**

在RPC中，数据的编码和解码是至关重要的步骤。编码是将数据转换为二进制格式以便在网络上传输的过程，而解码是将接收到的二进制数据还原为可操作的对象。

在RPC中，常见的编码和解码方案包括以下几种：

1. JSON（JavaScript Object Notation）

JSON是一种广泛使用的文本格式，它以键值对的形式表示数据。JSON编码和解码在RPC中使用广泛，因为它易于阅读和调试，并且支持多种编程语言。然而，由于它是文本格式，可能对于包含大量数据的消息来说会比较冗长，并且在解码时需要进行解析。

1. Protobuf（Protocol Buffers）

Protobuf是一种二进制格式的序列化框架，具有很高的效率和较小的数据体积。它使用 `.proto` 文件定义消息格式，并根据这些文件生成相应的编码和解码代码。Protobuf在RPC中被广泛使用，因为它能够提供高性能和可扩展性，并且支持多种编程语言。

1. MessagePack

MessagePack是一种轻量级的二进制序列化格式，它的目标是在数据大小和性能之间提供一个平衡。它支持多种编程语言，并具有较高的编码/解码速度。

1. Avro

Avro是一种数据序列化系统，它结合了高性能和动态数据结构的能力。它使用一个模式定义文件来描述数据的结构，并支持多种编程语言。

在RPC中，不同的编码和解码方案具有不同的优势和适用场景：JSON是一种通用且易于使用的格式，适合在简单的RPC场景中使用；Protobuf和MessagePack适用于对性能和数据大小有较高要求的场景；Avro则适用于需要动态数据结构的场景。

根据具体需求和系统的特点，选择合适的编码和解码方案对提高RPC的性能和效率非常重要。此外，还需要注意在跨不同编程语言和平台的情况下，确保选用的编码和解码方案具有良好的跨语言兼容性。

1. **错误处理和协议缓冲区**

在使用协议缓冲区进行数据传输时，错误处理是一个重要的考虑因素。在RPC等分布式系统中，错误可能在网络通信、数据序列化和业务逻辑中发生，因此，合适的错误处理机制对于保证系统的稳定性和可靠性至关重要。

协议缓冲区提供了一些内置的错误处理机制来帮助我们处理异常情况。在Rust中使用协议缓冲区，以便利用它的错误类型和返回值来捕获和处理错误。

在Rust中，协议缓冲区的序列化和反序列化操作返回一个Result类型，其中包含一个Ok变体表示成功的操作结果和一个Err变体表示错误的结果，我们可以使用match或Result的各种方法来处理这些错误。例如，当将消息对象序列化为字节流时，可以使用write\_to\_bytes方法返回一个Result类型，如果操作成功则返回序列化后的字节流，否则返回错误，代码示范如下所示：

1. use message::Person;

2. use protobuf::{Message, ProtobufError};

3. fn main() -> Result<(), ProtobufError> {

4. let person = Person::new();

5. // 设置 person 对象的字段值...

6. // 尝试将 person 对象序列化为字节流

7. let bytes = person.write\_to\_bytes()?;

8. // ... 之后的操作

9. Ok(())

10. }

在上述代码中，我们使用 ? 运算符来传播错误。如果序列化操作成功，它将返回字节流，否则会将错误传播到函数的调用方。

1. **协议缓冲区的版本控制与演化**

在分布式系统开发中，协议缓冲区的版本控制和演化是一个重要的考虑因素。当系统的消息格式需要进行更改或添加新功能时，我们需要确保旧版本的消息与新版本的消息可以兼容，并且能够正确地进行序列化和反序列化。协议缓冲区提供了一些机制来帮助我们处理版本控制和演化的问题。几种常见的策略和技巧如下：

1. **向后兼容性**

当向消息格式中添加新字段时，保持旧版本的消息可以被新版本的代码正确解析是很重要的。协议缓冲区通过给字段分配唯一的标识符来实现这一点。解析器可以忽略它们不能理解的字段，从而避免在反序列化时出现问题。

1. **向前兼容性**

当旧版本的代码遇到新版本的消息时，它们可能会缺少新版本中引入的字段。为了保持向前兼容性，协议缓冲区可以为字段设置默认值，以确保旧版本的代码在反序列化新版本的消息时不会引发错误。

1. **可选字段**

协议缓冲区允许字段被标记为可选的。可选字段在序列化时，只有在有值时才会被包含在消息中。这使得我们可以在不破坏兼容性的情况下向消息中添加可选字段。

1. **One of**

One of特性允许在同一层次结构中只选择一个字段，一旦选择了一个字段，其他的字段将被忽略。当我们需要引入互斥的字段时，使用one of特性可以很好地处理版本演化。

在进行版本控制和演化时，我们还应该遵循一些最佳实践：

* 良好的文档记录：准确记录每个版本的消息格式和字段变化，并对其进行文档化，以便团队中的开发人员和其他利益相关方能够理解和适应消息格式的变化。
* 版本协商：定义一个版本协商的机制，使得通信双方能够协商并确定使用哪个版本的消息格式，确保通信各方都能够正确解析和处理消息。
* 渐进升级：为了确保平滑的版本演化，应该采用渐进升级的策略，逐渐引入新的消息格式和字段，而不是一次性地进行巨大的改变。

协议缓冲区的版本控制和演化是一个复杂的问题，需要在设计和开发过程中精心考虑。通过合理地使用向后兼容性、向前兼容性、可选字段、One of等特性，并遵循最佳实践，我们可以更好地处理协议缓冲区版本的变化，确保在分布式系统中消息的正确传输和解析。

### gRPC

1. **gRPC概述**

gRPC是由Google开源的高性能远程过程调用（RPC）框架，它基于HTTP/2协议和Protocol Buffers，旨在简化不同服务之间的通信。gRPC提供了跨语言的支持，并具备出色的性能和可扩展性，使开发者能够构建可靠、高效的分布式系统，实现高效可靠的通信服务。同时它还提供了多种负载均衡策略和错误处理机制，以及支持认证、授权和流控等安全性和可靠性的特性。

gRPC是基于HTTP/2协议进行通信，这使得它具有了较低的延迟和更高的并发性能，gRPC可以复用连接、多路复用请求，以及使用二进制编码和压缩技术，从而减少了网络传输的开销和带宽消耗，相对于其他RPC框架具有更低的延迟和更高的吞吐量。除此之外，gRPC通过使用Protocol Buffers作为IDL，可以自动生成各种编程语言的客户端和服务端代码，这意味着可以使用不同的编程语言来编写不同部分的系统，并且它们可以无缝地进行通信和交互。

与其他RPC框架相比，gRPC还有一些明显的优势。gRPC使用了二进制编码和HTTP/2协议，以及提供了流式传输的能力，相对于传统的RESTful API更加高效；相对于其他RPC框架，gRPC的生态系统更加成熟，有着广泛的支持和社区贡献，可以提供更好的开发和调试工具、文档和示例代码等。

1. **gRPC基本概念**

gRPC使用协议缓冲区来定义底层数据和消息，消息通过TCP/IP之上的HTTP/2交换，这种方式可以更好地利用现有的连接。gRPC框架由服务（Service）、消息（Message）、服务接口定义语言（IDL）和传输协议等组成。

服务（Service）是指一组相互关联的操作或行为的集合，通过远程过程调用（RPC）的方式来提供。在gRPC中，服务由一组方法组成，每个方法都具有输入参数和输出参数。服务定义了系统中的功能和行为，客户端可以通过调用这些方法来远程调用服务端的功能。

消息（Message）是在客户端和服务端之间传输的数据单元。在gRPC中，消息使用Protocol Buffers进行定义和表示。消息可以包含不同类型的字段，比如整数、字符串、枚举和嵌套消息等。通过定义消息结构，我们可以描述数据的组织方式和字段的含义。消息可以作为方法的输入参数和输出参数使用，用于传输数据和信息。

传输协议是用于客户端和服务端之间进行通信的协议。在gRPC中，默认使用HTTP/2作为传输协议。HTTP/2协议提供了多路复用、流控、服务器推送和头部压缩等特性，可以提高通信的效率和性能。gRPC利用HTTP/2协议的这些特性，实现了高效的、跨语言的远程过程调用。

服务接口定义语言（IDL）是用于定义服务和消息结构的语言。在gRPC中，默认使用Protocol Buffers作为IDL。通过定义IDL，我们可以声明服务的接口、方法、输入参数和输出参数，以及消息的结构和字段类型。IDL提供了一种统一的方式来描述系统中的服务和消息，使得不同语言的开发者可以根据IDL来生成相应的代码。使用IDL来定义服务接口和消息结构十分简便。首先，在IDL中定义服务接口，包括服务名称和方法列表。每个方法都指定了输入参数和输出参数的消息类型。接着，定义消息的结构和字段。通过指定字段的类型和名称，可以描述消息的组成和含义。最后，通过IDL工具，根据IDL文件自动生成客户端和服务端的代码。生成的客户端代码可以用来发起远程过程调用，将输入参数打包成消息并发送给服务端。服务端代码可以用来接收请求、解析消息，并根据方法的定义执行相应的操作，然后将结果打包成消息返回给客户端，这样我们可以轻松地构建跨语言的通信系统，实现高效可靠的远程过程调用。

1. **gRPC的优势**

gRPC作为一种现代化的RPC框架，与其他常见的RPC框架（如RESTful API、Thrift、Apache Dubbo等）相比，它在高效性、自动化代码生成、跨语言支持以及强大的序列化机制方面显得格外突出。gRPC还支持多种编程语言，使得开发者能够使用自己熟悉的语言来开发不同组件的系统，同时还保持了强大的跨语言通信能力。在对性能有较高要求的系统、需要跨语言通信的场景、有版本管理和自动化代码生成需求的项目，都可以使用gPRC框架。需要处理大量请求并追求性能的系统，gRPC可提供高吞吐量和低延迟；需要不同语言编写的组件之间进行通信，gRPC的跨语言支持也是理想选择。同时，gRPC还提供了Protocol Buffers和代码生成工具对接口进行版本管理并实现代码的自动生成。

因此，无论是构建大规模分布式系统还是开发跨语言应用，gRPC都能够为开发者提供可靠的解决方案。

1. **gRPC核心功能**

gRPC的核心功能包括方便的远程过程调用和灵活的流式传输，使开发人员能够构建高效、可靠的分布式应用程序和服务。

1. 远程过程调用（RPC）：gRPC使用RPC作为通信模式，允许客户端应用程序调用位于远程服务器上的方法，就像调用本地方法一样。通过定义服务接口和消息结构，客户端可以直接调用远程服务器上的方法，并传递参数和接收返回结果。
2. 流式传输：gRPC支持两种类型的流式传输：客户端流式和服务器端流式。客户端流式允许客户端发送多个消息给服务器，而服务器返回单个响应。服务器端流式允许服务器发送多个消息给客户端，而客户端返回单个响应。双向流式允许客户端和服务器之间同时发送和接收多个消息。

在服务接口中定义方法可以通过.proto文件来完成，使用IDL语言（如Protocol Buffers）指定参数和返回类型。方法可以指定为单向调用（unary），即客户端发送一个请求，服务器返回一个响应。也可以指定为服务器流式（server streaming），即客户端发送一个请求，服务器返回一个流式的响应。还可以指定为客户端流式（client streaming），即客户端发送一个流式的请求，服务器返回一个响应。还可以指定为双向流式（bidirectional streaming），即客户端和服务器都可以同时发送和接收多个消息。

1. **gRPC的一般使用方式**

gRPC的一般使用步骤如下：

1. 使用 Protocol Buffers (protobuf) 定义服务接口和消息格式；
2. 使用protobuf编译器生成服务器和客户端的代码；
3. 利用生成的代码实现服务器端的服务逻辑并启动gRPC服务器，监听指定的网络地址和端口；
4. 根据生成的代码创建gRPC客户端实例，以调用远程方法，将请求参数传递给服务器，并接收返回结果；
5. 根据服务器端返回的结果，进行相应的处理和解析。

下面我们具体来介绍gRPC各步骤如何实现。

1. 定义服务接口和消息结构：编写.proto文件来定义服务接口和消息结构。创建一个名为helloworld.proto的文件：

syntax = "proto3";

option java\_multiple\_files = true;

option java\_package = "io.grpc.examples.helloworld";

option java\_outer\_classname = "HelloWorldProto";

option objc\_class\_prefix = "HLW";

package helloworld;

service Greeter {

rpc SayHello (HelloRequest) returns (HelloReply) {}

}

message HelloRequest {

string name = 1;

}

message HelloReply {

string message = 1;

}

1. 添加相关依赖，在toml中添加以下内容：

[lib]

doctest = false

test = false

[dependencies.grpc]

path = "../../grpc"

[dependencies.grpc-protobuf]

path = "../../grpc-protobuf"

[dependencies]

protobuf = "2.23"

futures = "0.3.\*"

tls-api = "0.6.0"

tls-api-stub = "0.6.0"

tls-api-native-tls = "0.6.0"

#httpbis = "~0.9"

httpbis = { git = "https://github.com/stepancheg/rust-http2", rev = "04410b1a07f937d94e075c813d05a68022f0c7c8" }

#httpbis = { path = "../../../rust-http2/httpbis" }

env\_logger = "~0.9"

[build-dependencies]

protoc-rust-grpc = { path = "../../protoc-rust-grpc" }

[[bin]]

name = "greeter\_client"

test = false

[[bin]]

name = "greeter\_server"

test = false

1. 在src目录下新增lib.rs文件，添加以下代码：

pub mod helloworld;

pub mod helloworld\_grpc;

1. 生成Rust代码：根目录下新建文件build.rs，添加以下代码，通过helloworld.proto文件生成相应的Rust代码。（注：生成相应代码要使用protoc工具，下载完成后将bin文件目录地址配置在环境变量中即可）

fn main() {

protoc\_rust\_grpc::Codegen::new()

.out\_dir("src")

.input("helloworld.proto")

.rust\_protobuf(true)

.run()

.expect("protoc-rust-grpc");

}

1. 服务端实现：在src文件下添加bin文件，文件中新建server.rs,实现代码如下：

use std::env;

use std::thread;

use grpc\_examples\_greeter::helloworld::\*;

use grpc\_examples\_greeter::helloworld\_grpc::\*;

use grpc::ServerRequestSingle;

use grpc::ServerResponseUnarySink;

use tls\_api::TlsAcceptor;

use tls\_api::TlsAcceptorBuilder;

struct GreeterImpl;

impl Greeter for GreeterImpl {

fn say\_hello(

&self,

req: ServerRequestSingle<HelloRequest>,

resp: ServerResponseUnarySink<HelloReply>,

) -> grpc::Result<()> {

let mut r = HelloReply::new();

let name = if req.message.get\_name().is\_empty() {

"world"

} else {

req.message.get\_name()

};

println!("greeting request from {}", name);

r.set\_message(format!("Hello {}", name));

resp.finish(r)

}

}

fn test\_tls\_acceptor() -> tls\_api\_native\_tls::TlsAcceptor {

let pkcs12 = include\_bytes!("../foobar.com.p12");

let builder = tls\_api\_native\_tls::TlsAcceptor::builder\_from\_pkcs12(pkcs12, "mypass").unwrap();

builder.build().unwrap()

}

fn is\_tls() -> bool {

env::args().any(|a| a == "--tls")

}

fn main() {

let tls = is\_tls();

let port = if !tls { 50051 } else { 50052 };

let mut server = grpc::ServerBuilder::new();

server.http.set\_port(port);

server.add\_service(GreeterServer::new\_service\_def(GreeterImpl));

if tls {

server.http.set\_tls(test\_tls\_acceptor());

}

let \_server = server.build().expect("server");

println!(

"greeter server started on port {} {}",

port,

if tls { "with tls" } else { "without tls" }

);

loop {

thread::park();

}

}

1. 客户端实现：bin文件中新建client.rs,实现代码如下：

use std::env;

use std::sync::Arc;

use grpc\_examples\_greeter::helloworld::\*;

use grpc\_examples\_greeter::helloworld\_grpc::\*;

use grpc::ClientStub;

use grpc::ClientStubExt;

use futures::executor;

use tls\_api::TlsConnector;

use tls\_api::TlsConnectorBuilder;

fn test\_tls\_connector() -> tls\_api\_native\_tls::TlsConnector {

let root\_ca = include\_bytes!("../root-ca.der");

let mut builder = tls\_api\_native\_tls::TlsConnector::builder().unwrap();

builder

.add\_root\_certificate(root\_ca)

.expect("add\_root\_certificate");

builder.build().unwrap()

}

fn is\_tls() -> bool {

env::args().any(|a| a == "--tls")

}

fn main() {

env\_logger::init();

let tls = is\_tls();

let name = env::args()

.filter(|a| a != "--tls")

.nth(1)

.map(|s| s.to\_owned())

.unwrap\_or\_else(|| "world".to\_owned());

let port = if !tls { 50051 } else { 50052 };

let client\_conf = Default::default();

let client = if tls {

// This is a bit complicated, because we need to explicitly pass root CA here

// because server uses self-signed certificate.

// TODO: simplify it

let tls\_option = httpbis::ClientTlsOption::Tls(

"foobar.com".to\_owned(),

Arc::new(test\_tls\_connector().into\_dyn()),

);

let grpc\_client = Arc::new(

grpc::ClientBuilder::new("::1", port)

.explicit\_tls(tls\_option)

.build()

.unwrap(),

);

GreeterClient::with\_client(grpc\_client)

} else {

GreeterClient::new\_plain("::1", port, client\_conf).unwrap()

};

let mut req = HelloRequest::new();

req.set\_name(name);

let resp = client

.say\_hello(grpc::RequestOptions::new(), req)

.join\_metadata\_result();

println!("{:?}", executor::block\_on(resp));

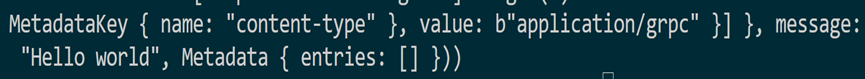
}

1. 构建和运行应用程序：编译代码并运行gRPC服务器和客户端，实现服务器与客户端之间的远程过程调用,分别输入命令：

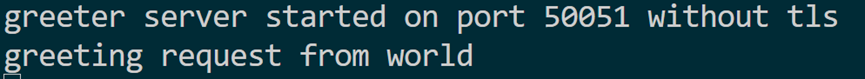
cargo run --bin greeter\_server

cargo run --bin greeter\_client

客户端运行结果如下图所示。



服务器端运行结果如下图所示。



## SMTP应用编程

### web邮件服务

在1996年以前，电子邮件的收发主要依赖于客户端软件。这种软件必须安装在用户的设备上，并进行相应的配置，通常只能在特定的设备上使用，比如个人电脑或工作站。这种情况给那些经常更换设备或需要在不同地点办公的人带来了极大的不便。

由于客户端软件的限制，用户必须在每台设备上安装和配置邮件客户端，这不仅耗费时间，还增加了管理和维护的复杂性。而且，由于每个设备都有自己独立的设置和存储，如果用户想要访问已收到或已发送的邮件，就必须在不同的设备之间进行数据同步，这可能导致数据丢失或不一致。此外，由于客户端软件通常与特定的操作系统或设备类型相关联，用户在不同设备上的体验可能存在差异，有时甚至可能无法在某些设备上访问电子邮件。

为了解决这些问题，网页邮件服务迅速崭露头角。相比传统的客户端软件，网页邮件服务无需用户在设备上安装任何软件，只需通过普通的网页浏览器登录邮箱即可访问电子邮件，完全摆脱了设备和地点的限制。

网页邮件的出现极大地简化了电子邮件的使用流程，用户无需再担心在不同设备上安装和配置邮件客户端，也不必担心数据同步和一致性的问题。无论是在个人电脑、笔记本电脑、平板电脑还是智能手机上，只要有网络连接和浏览器，用户就能轻松地访问自己的电子邮箱。此外，网页邮件服务通常还提供了一系列便捷的功能，如自动同步、搜索、过滤和标记等，进一步提升了用户的使用体验。最重要的是，随着对网络安全意识的增强和技术的改进，网页邮件服务的安全性得到了大幅提升，加密技术、身份验证机制、垃圾邮件过滤等安全措施被广泛采用，以保护用户的个人信息和通信内容不受未经授权的访问和窃取。总的来说，网页邮件服务的出现极大地改变了人们对电子邮件的使用方式，使其变得更加灵活、便捷、安全。

Web邮件系统一般提供邮件收发、用户在线服务和系统服务管理等功能。

邮件收发功能是Web邮件系统的核心。通过这一功能，用户可以方便地发送、接收和管理电子邮件。用户可以使用系统提供的界面编写邮件、添加附件、设置邮件的优先级和重要性，并选择收件人。一旦邮件发送完成，系统将负责将邮件传递给目标收件人。在收件方面，用户可以查看收件箱，阅读邮件内容，并进行相应的回复或转发操作。系统通常会提供对邮件进行标记、归档和删除等管理功能，以便用户有效地组织和管理收到的邮件。

用户在线服务方面，Web邮件系统提供了一系列功能，以便用户更好地管理其邮件账户。这包括账户注册功能，允许新用户创建自己的账户并开始使用系统。登录功能允许已注册用户通过用户名和密码安全地访问其邮件账户。密码重置功能允许用户在忘记密码时通过安全验证方式重置密码。个人资料管理功能允许用户更新其个人信息，如姓名、联系方式等。邮件过滤设置功能允许用户配置邮件过滤规则，以便自动将特定类型的邮件分类或屏蔽。自动回复设置功能允许用户设置自动回复消息，以便在暂时无法回复邮件时自动向发件人发送通知。

系统服务管理功能可以管理整个系统的运行和配置。管理员登录功能允许系统管理员通过安全身份验证方式访问系统管理界面。用户管理功能允许管理员创建、删除和禁用用户账户，以便对系统的用户进行管理。权限管理功能允许管理员定义不同用户角色和权限，以便对用户访问系统的范围和权限进行控制。存储管理功能涵盖了管理邮件和附件的存储空间，以确保系统的存储资源被充分利用和合理分配。日志记录和监控功能允许管理员监视系统的运行状态和活动日志，以及及时发现和解决潜在的问题或异常情况。

如果要使用web邮件服务，在安装和使用Web邮件服务器时，需要注意一些配置和设置。例如，需要安装Internet服务和DNS服务，配置域名和IP绑定，添加邮件用户，设置收发规则，启用SMTP域名验证功能等。这些配置和设置对于确保邮件服务器的正常运行和邮件的安全传输非常重要。Web邮件服务的内容如表7.1所示。

表7.1 Web邮件服务内容

|  |  |
| --- | --- |
| 内容 | 说明 |
| 用户访问 | 用户通过打开Web浏览器，并输入特定的Web邮件服务提供商的网址，例如Gmail、Outlook等。 |
| 用户身份验证 | 用户需要提供其账户的凭据（用户名和密码）进行身份验证，以便访问其电子邮件账户。 |
| 邮件访问和管理 | 一旦用户成功登录，他们可以使用Web界面来查看收件箱、发送新邮件、回复邮件、删除邮件等。用户可以通过简单的点击和拖放操作来管理邮件，例如创建文件夹、标记邮件、设置过滤器等。 |
| 邮件发送 | 用户可以使用Web界面编写和发送新的电子邮件。他们可以输入收件人地址、主题、正文和附件等信息，并通过点击发送按钮将邮件发送出去。 |
| 附加功能 | Web邮件服务通常还提供其他功能，如联系人管理、日历、任务列表等。用户可以使用这些功能来组织和管理他们的个人信息。 |

Web邮件服务的优点包括易于访问和使用，无需安装额外的软件，可以在任何设备上使用，并且可以通过云存储保留邮件和个人设置。然而，它也有一些限制，如对网络连接的依赖性、可能的安全风险等。

### 简单电子邮件系统

电子邮件系统是因特网上使用的最多且最受用户欢迎的一种应用。电子邮件系统包括用户代理和邮件服务器，提供收发邮件、邮件传送服务，邮件被传送到收信人邮件服务器的收信人信箱中，收信人可随时上网到其邮件服务器读取。

1982年制定了简单邮件传送协议SMTP（Simple Mail Transfer Protocol）和因特网文本报文格式，他们都已成为因特网的正式标准。

1. **SMTP（Simple Mail Transfer Protocol）**

SMTP是一种专用于传输邮件的协议，主要用于系统之间的邮件信息传递，并提供有关来信的通知。使用 SMTP可实现相同网络处理进程之间的邮件传输，也可通过中继器或网关实现进程与其它网络之间的邮件传输。

与其他网络协议一样，SMTP 允许计算机和服务器交换数据，无论其底层硬件或软件是什么。正如使用信封地址书写的标准化格式允许邮政服务得以运作一样，SMTP需要对收件人和寄件人的信息进行标准化，从而实现广泛的邮件网络。

SMTP是一种邮件传递协议，不是邮件检索协议。邮政服务将邮件传递到邮箱，但收件人仍然必须从邮箱中提取邮件。同样，SMTP将电子邮件传递到某个电子邮件提供商的邮件服务器，但需要使用其他协议来从邮件服务器检索该电子邮件，以便收件人读取邮件。

举个简单的例子，我们平常也经常使用各个厂家的邮箱来接收邮件，当我们使用邮箱给别人发送邮件时，通过SMTP协议，邮件就会发送过去。注意，例子中只提到了发送邮件，没错，SMTP作用是发送邮件，而在客户端接收和管理邮件时用到了另外的IMAP（互联网消息访问协议）和POP（邮局协议）协议，可以说是各司其职，这里我们主要介绍SMTP协议。

所有的网络协议都是为了数据在互联网中更高效地传递而定制的一系列过程，数据传输中遵循了这一数据交换过程，SMTP也不例外，在电子邮件客户端和服务器之间定义了特定的传输过程。电子邮件客户端是用户与之交互的对象，如计算机或 Web 应用程序，用户可以在其中访问和发送电子邮件。邮件服务器是用于发送、接收和转发邮件的专用计算机，用户不会与邮件服务器直接进行交互。

让我们先来了解两个概念：SMTP信封和SMTP命令。

SMTP信封：是客户端发送给邮件服务器的一组信息，说明电子邮件来自何处和将前往何处。SMTP信封对电子邮件收件人不可见，这一点与电子邮件头和正文不同。

SMTP命令：是预定义的基于文本的指令，告诉客户端或服务器要采取什么操作及如何处理任何伴随的数据。常见的SMTP命令如表7.2所示。

表7.2 常见的SMTP命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| HELO/EHLO | 这些命令发出 “Hello” 并在客户端和服务器之间启动 SMTP 连接。“HELO” 是该命令的基本版本；“EHLO” 用于一种特别类型的 SMTP。 |
| MAIL FROM | 这个命令告诉服务器谁在发送该电子邮件。如果 Alice 试图给她的朋友 Bob 发电子邮件，客户端可能会发送 “MAIL FROM：<alice@example.com>”。 |
| RCPT TO | 这个命令用于列出电子邮件的收件人。如果有多个收件人，客户端可多次发送该命令。如上例子中，Alice 的电子邮件客户端将发送： “RCPT TO: <bob@example.com>"。 |
| 备注：除此之外还有例如DATA等命令，在后面进行Rust的邮件系统编写时会提到。 | |

接下来，我们了解一下SMTP中客户端和服务器发送邮件时的内容传递：

1. SMTP 连接打开

由于SMTP使用传输控制协议（TCP）作为传输协议，因此第一步从客户端和服务器之间建立 TCP 连接开始。接下来，电子邮件客户端以一个专门的“Hello”命令（HELO or EHLO，如下所述）开始电子邮件发送过程。

1. 电子邮件数据传递

客户端向服务器发送一系列带有邮件实际内容的命令：邮件头（包括目的地和主题行）、邮件正文以及任何其他组件。

1. 邮件传送代理（MTA）

服务器运行一个称为邮件传送代理（MTA）的程序。MTA 检查收件人的电子邮件地址，如果该地址不同于发件人地址，则查询域名系统（DNS）以找到收件人的 IP 地址。这个过程好比邮局查找邮件收件人的邮政编码。

1. 连接关闭

客户端在数据传输完成时通知服务器，服务器关闭连接。此时，服务器将不会从客户端接受任何额外电子邮件数据，除非客户端打开新的 SMTP 连接。

需要注意，通常情况下，上述第一个电子邮件服务器不是实际电子邮件的最终目的地。该服务器收到客户端发来的邮件后，会与另一个邮件服务器重复此 SMTP 连接过程。第二个服务器也进行同样的操作，直到电子邮件最终到达收件人的收件箱（位于收件人电子邮件提供商控制的邮件服务器上）。

这个过程好比邮件从发件人传送到收件人的方式。邮差不会直接将信件从发件人送到收件人手上，而是先将信件带回邮局，邮局将信件运送到另一个城镇的邮局，然后是在另一个城镇的邮局，如此重复，直至信件到达收件人。同样，电子邮件通过 SMTP 协议从一个服务器传送到另一个服务器，直至到达收件人的收件箱。

1. **扩展SMTP（ESMTP）**

扩展简单邮件传输协议（ESMTP）是SMTP协议的一个版本，扩展了原有功能，支持发送电子邮件附件、使用 TLS 及其他能力。几乎所有电子邮件客户端和电子邮件服务都使用 ESMTP，而非基本的 SMTP。

ESMTP 有一些额外的命令，包括 “EHLO”（即“扩展的 hello”）消息，实现从连接开始时使用 ESMTP。

下面我们用Rust实现一个简易电子邮件系统的编写，方便我们深入理解SMTP协议。步骤如下：

1. 导入SMTP客户端库

我们需要使用Rust的smtp客户端库来与SMTP服务器通信。例如rust-smtp客户端库。

1. smtp::Client::new()

1. 连接SMTP服务器

使用smtp客户端连接到SMTP服务器,例如Gmail的SMTP服务器。

1. let mut client = smtp::Client::new("smtp.gmail.com", 587);

2. client.connect(&hostname, username, password);

1. 发送邮件

使用smtp客户端发送邮件,包括发件人、收件人、主题和邮件内容。

1. let message = Message::builder()

2. .from(Address::new("me@gmail.com", "My Name"))

3. .to(Address::new("you@gmail.com", "Your Name"))

4. .subject("Hello")

5. .body("Hello World!");

6.

7. client.send(&message);

1. 关闭连接

发送完邮件后关闭与SMTP服务器的连接。

1. client.quit();

1. 接收邮件

使用POP3或IMAP协议从邮件服务器获取邮件。

1. 解析和存储邮件

解析邮件内容，如发件人、收件人、主题、正文等，并存储到数据库中。

1. 提供Web界面

开发一个Web界面,用户可以通过浏览器发送和查看邮件。

有了这些步骤，我们再看下面的具体编程实现。

1. use smtp::{Client, Message};

2. use mailparse::parse;

3. use r2d2\_sqlite::SqliteConnectionManager;

4. use r2d2::Pool;

5.

6. // ① SMTP客户端

7. fn send\_mail(client: &mut Client, message: &Message) -> Result<(), smtp::Error> {

8. client.send(message)?;

9. client.quit()?;

10. Ok(())

11. }

12.

13. // ②连接SMTP服务器

14. fn connect() -> Result<Client, smtp::Error> {

15. let hostname = "smtp.example.com";

16.

17. let mut client = Client::new(hostname, 587)?;

18. client.connect(hostname, "username", "password")?;

19.

20. Ok(client)

21. }

22.

23. // ③ 邮件解析

24. fn parse\_mail(mail: &[u8]) -> Result<Mail, mailparse::Error> {

25. let mut parser = mailparse::parse(mail)?;

26.

27. let mail = Mail {

28. from: parser.from.unwrap(),

29. to: parser.to.unwrap(),

30. subject: parser.subject.unwrap(),

31. body: parser.body.unwrap()

32. };

33.

34. Ok(mail)

35. }

36.

37. // ④ 数据库

38. type DbPool = Pool<SqliteConnectionManager>;

39.

40. fn main() {

41. // 连接数据库

42. let db\_pool = DbPool::new("mails.db").unwrap();

43.

44. // 连接SMTP服务器

45. let mut client = connect().unwrap();

46.

47. // 发送邮件

48. let message = Message::builder()

49. .from("User <user@example.com>")

50. .to("Recipient <recipient@example.com>")

51. .subject("Test Mail")

52. .body("Hello")

53. .unwrap();

54.

55. send\_mail(&mut client, &message).unwrap();

56.

57. // 接收邮件

58. let mail = fetch\_mail().unwrap();

59.

60. // 解析并存储到数据库

61. let parsed = parse\_mail(mail).unwrap();

62. save\_to\_db(&db\_pool, &parsed).unwrap();

63. }

本例用到了以下几个包和库：

1. smtp crate
2. smtp crate提供了SMTP协议的客户端实现，可以用来连接SMTP服务器发送邮件。主要用到以下功能：

* Client struct: SMTP客户端会话
* send(): 发送邮件
* connect(): 连接SMTP服务器
* quit(): 断开连接

1. mailparse crate

mailparse crate可以解析邮件内容,提取邮件头部信息如发件人、收件人、主题等。主要用到以下功能：

* parse(): 解析邮件内容
* from/to/subject/body属性:提取邮件头部字段

1. r2d2和r2d2\_sqlite

r2d2是一个连接池库,用于管理数据库连接。r2d2\_sqlite提供了sqlite数据库的连接管理实现。主要用到以下功能：

* Pool: 数据库连接池
* SqliteConnectionManager: sqlite数据库连接管理器
* new(): 创建连接池

1. sqlite

sqlite是一款轻量级的嵌入式SQL数据库，这里用来存储解析后的邮件内容。主要用到以下功能：

* 数据库表结构设计
* 保存/查询邮件记录

这些库和包分别实现了SMTP客户端、邮件解析、数据库连接池管理和数据存储四个主要功能模块。通过整合这些组件就可以构建一个简单的电子邮件系统原型。

### 示例

本节我们主要是通过一个具有总结性的例子对SMTP应用编程进行一个总结性的讲解，编程思路如下：

1. 定义邮件数据结构，包含发件人、收件人、主题、内容等字段
2. 实现SMTP客户端连接函数，连接SMTP服务器
3. 实现邮件发送函数，根据邮件数据结构构建邮件，通过SMTP客户端发送
4. 实现邮件接收函数，从SMTP服务器接收邮件
5. 实现邮件解析函数，解析收到的邮件提取头部信息
6. 测试函数，连接SMTP服务器，发送和接收邮件，验证功能

这段 Rust 代码展示了一个简单但完整的邮件处理流程，涵盖了连接到 SMTP 服务器、发送邮件、接收邮件以及解析邮件内容等功能。首先，代码通过引入所需的库和模块，如 rust\_smtp 和 rust\_mailparse，建立了必要的依赖关系。接着，通过定义 Mail 结构体，明确了邮件的基本信息结构，包括发件人、收件人、主题和正文。

随后，代码实现了几个核心功能函数：

①connect 函数负责建立到 SMTP 服务器的连接，并返回一个可用的客户端对象。这一步骤对于后续的邮件发送和接收至关重要，因为它确保了程序能够与外部邮件服务器进行通信。

②send 函数利用建立好的 SMTP 连接，将构建好的邮件消息发送到指定的收件人。这一功能使得程序能够自动化地发送邮件，无需人工干预，提高了效率和便捷性。

③receive 函数允许程序从已建立的 SMTP 连接中接收邮件。通过这个功能，程序可以及时获取到新的邮件，从而及时做出相应的处理，或者将邮件内容展示给用户。

④parse 函数则负责解析从 SMTP 服务器接收到的原始邮件内容，提取出其中的关键信息，如发件人、收件人、主题等。这个步骤是邮件处理流程中至关重要的一环，因为它将原始的邮件数据转化为程序可以理解和处理的格式，为后续的操作提供了基础。

⑤最后通过 main 函数将上述功能串联起来，构建了一个完整的邮件处理流程。程序先连接到 SMTP 服务器，然后发送邮件，接着接收邮件并解析其内容，最终将解析结果输出到控制台上。

实现代码如下：

1. use rust\_smtp::Client;

2. use rust\_mailparse::{Parser, Message};

3. use std::{error::Error, result::Result};

4. use bytes::Bytes;

5.

6. struct Mail {

7.       from: String,

8.        to: String,

9.        subject: String,

10.       body: String

11.      }

12.

13.      // 2. 实现SMTP连接

14.      fn connect() -> Result<Client, Error> {

15.       // 连接SMTP服务器

16.       let client = Client::new("smtp.example.com:587");

17.       client.connect("username", "password");

18.       Ok(client)

19.     }

20.

21.     // 3. 实现邮件发送

22.     fn send(client: &Client, mail: &Mail) -> Result<(), Error> {

23.       let message = Message::builder()?

24.         .from(mail.from.as\_str())

25.         .to(mail.to.as\_str())

26.        .subject(mail.subject.as\_str())

27.         .body(mail.body.as\_str());

28.

29.       client.send(&message);

30.       Ok(())

31.     }

32.

33.     // 4. 实现邮件接收

34.     fn receive(client: &Client) -> Result<Vec<u8>, Error> {

35.       // 接收邮件

36.       let mut buffer = Vec::new();

37.       client.recv(&mut buffer);

38.       Ok(buffer)

39.     }

40.

41.     // 5. 实现邮件解析

42.     fn parse(mail: &[u8]) -> Result<Mail, Error> {

43.       // 解析邮件提取头部信息

44.       let mut parser = Parser::from\_reader(mail);

45.       let mail = Mail {

46.         // ...

47.       };

48.       Ok(mail)

49.     }

50.

51.     // 6. 测试函数

52.     fn main() {

53.       let client = connect()?;

54.

55.       let mail = Mail { /\* ... \*/ };

56.       send(&client, &mail)?;

57.

58.       let received = receive(&client)?;

59.       let parsed = parse(&received);

60.

61.       println!("Parsed mail: {parsed}");

62.     }

63.

运行成功时的结果如下：



在实现网页版邮件服务的编码中有以下几个难点。

① **rust\_smtp::Client：**这是一个用于 SMTP 客户端的类型，来自 rust\_smtp 库。Client 类型提供了与 SMTP 服务器进行通信的功能，包括建立连接、发送邮件等。在代码中，使用该类型连接到SMTP服务器，并发送和接收邮件。

② **rust\_mailparse::{Parser, Message}：**这两个类型来自 rust\_mailparse 库，用于解析邮件内容。Parser 类型用于解析原始邮件数据，提取出邮件的各个部分。Message 类型用于表示解析后的邮件消息，包括发件人、收件人、主题、正文等信息。

③ **connect() 函数：**一个自定义函数，用于建立到 SMTP 服务器的连接，并返回一个 Client 对象。在函数内部，通过 Client::new 创建一个新的 SMTP 客户端对象，然后调用 connect 方法进行连接。函数使用 Result 类型来处理可能的连接错误，并返回一个 Result<Client, Error>。

④ **send() 函数：**这个函数用于发送邮件到 SMTP 服务器。首先，通过 Message::builder() 创建一个邮件构建器对象，然后设置邮件的各个部分，包括发件人、收件人、主题和正文。最后，调用 SMTP 客户端的 send 方法发送构建好的邮件消息。函数返回一个 Result<(), Error>，用于处理可能的发送错误。

5、receive() 函数：

这个函数用于从 SMTP 服务器接收邮件。

它接受一个指向 SMTP 客户端的引用，并使用客户端的 recv 方法接收邮件内容。

接收到的邮件内容以字节向量的形式返回。

函数返回一个 Result<Vec<u8>, Error>，用于处理可能的接收错误。

6、parse() 函数：

这个函数用于解析从 SMTP 服务器接收到的原始邮件内容。

它接受一个字节向量作为参数，通过 Parser::from\_reader 方法创建一个解析器对象，并提取出邮件的各个部分。

然后，根据解析后的数据构建一个 Mail 结构体，并返回。

函数返回一个 Result<Mail, Error>，用于处理可能的解析错误。

以上的代码可以帮助大家加深对SMTP协议的理解。SMTP是电子邮件的基础协议，负责在邮件服务器之间进行通信，实现邮件的发送和接收功能。实现一个SMTP客户端需要解决的主要问题包括建立与邮件服务器的连接、构建并发送邮件、接收邮件并进行后续处理等。在这段Rust代码中，我们选择了 rust-smtp 库，它提供了一个简单而强大的SMTP客户端实现，可以帮助我们完成这些任务。

建立连接是SMTP客户端工作的第一步。在这段代码中，我们通过 connect 函数指定了服务器地址和端口，并提供了用户名和密码进行身份验证。连接建立后，客户端与服务器即可进行后续通信。

发送邮件是SMTP客户端的核心功能之一。在这段代码中，我们通过构建 Message 对象来指定邮件的发件人、收件人、主题和内容，然后调用 send 函数将该邮件通过SMTP连接发送给服务器。在发送的过程中，客户端会发送 MAIL FROM、RCPT TO 和 DATA 等命令来指定邮件相关信息。

接收邮件同样重要。在这段代码中，我们并未直接实现接收邮件的功能，但可以通过使用其他库或者API来完成。一般而言，接收邮件的流程包括从服务器下载邮件，保存为字节流，然后使用邮件解析库进行解析。比如，可以使用 rust-mailparse 库来解析邮件头部字段。

我们通常会对解析后的邮件信息进行后续处理，比如保存到数据库进行管理，或者展示在用户界面上。这可以支持基于Web的邮件客户端应用，使用户能够方便地查看和管理邮件。

总而言之，SMTP应用编程对我们来说意义重大。通过选择合适的SMTP库，掌握基本流程和命令使用，可以实现一个功能完整的SMTP邮件客户端程序，在应用开发中发挥重要作用。这段Rust代码不仅展示了如何使用 rust-smtp 库构建一个简单的SMTP客户端，还帮助我们理解了SMTP协议的基本工作原理和核心概念。

## FTP和TFTP应用编程

### FTP服务应用与实现

1. **FTP简介**

FTP（File Transfer Protocol）是一种用于在计算机网络上进行文件传输的标准协议，它允许用户通过网络将文件从一台计算机传输到另一台计算机，即通过网络实现异构计算机间文件的“拷贝”。同时FTP使用C/S（客户端-服务器）模型，客户端通过FTP客户端软件与服务器建立连接，并发送命令来控制文件传输，服务器则负责接收和处理这些命令，并根据客户端的请求进行文件传输。FTP支持两种传输模式：ASCII模式和二进制模式。ASCII模式适用于传输文本文件，而二进制模式适用于传输非文本文件，如图片、音频和视频文件。FTP还提供了一些基本的文件操作功能，如上传（将文件从客户端发送到服务器）、下载（将文件从服务器下载到客户端）、删除和重命名等。尽管FTP是一个常用的文件传输协议，但它的安全性相对较低，因为它在传输过程中不对数据进行加密。为了增加安全性，可以使用安全的FTP协议（FTPS）或SSH文件传输协议（SFTP），它们提供了加密和身份验证功能。

1. **FTP的连接和工作方式**

FTP 采用TCP可靠传输协议实现网络之间文件传送的基本服务。FTP使用C/S模型，下面，我们根据这一特点详细介绍FTP的连接和工作方式。

FTP的服务器进程由两部分组成，一个主进程用于负责接收新的请求，而其他从属进程分别用来处理单个请求，好比酒店有前台和服务人员，前台用于办理新人入住手续，服务人员则在前台办理完成之后给住户提供洗漱和入住必备用品，主进程与从属进程的处理并发进行。客户端使用任意分配的端口号（>1023）与服务器建立连接，服务器端则采用端口号21来传输控制信息，端口号20来传输数据，但根据ftp工作方式的不同，服务端传输数据所采用的端口不一定是20。

对服务器而言，FTP协议分为主动（服务器主动连接）和被动（服务器被动连接）。主动连接详细过程如下：客户端端口N与服务器21号端口建立连接后（N>1023），客户端开始监听端口N+1，并发送FTP命令“PORT N+1”到ftp服务器，服务器则会从自己的数据端口20主动向客户端N+1号端口建立连接；服务器被动连接过程如下：客户端端口N与服务器21号端口建立连接后（N>1023），客户端打开N+1号端口，并发送PASV命令给服务器，服务器则会开启端口P(P>1023)，并发送“PORT P”命令给客户端，然后客户端从本地端口N+1到服务器端口P建立连接用来传送数据。因为高位随机端口可能会被服务器端的防火墙给阻塞掉，因此合理运用主动和被动方式则会大大增加ftp传输的安全性。

1. **搭建简单FTP应用程序**

接下来我们用Rust构建一个简单的FTP应用程序。首先需要一个FTP服务器，可以连接免费的FTP服务器，或者搭建私有的FTP服务器，这里我们来借助FileZilla Server工具来搭建FTP服务器，FileZilla Server工具是一个免费开源的FTP软件，分为客户端版本和服务器版本，具备所有的FTP软件功能。

在FileZilla Server中添加用户名为admin的用户，同时修改密码为admin，选择要共享目录的路径，设置操作权限，然后添加并保存操作权限。本例中服务器的根目录为​​E:\ftpShares，操作界面如图7.1和图7.2所示。

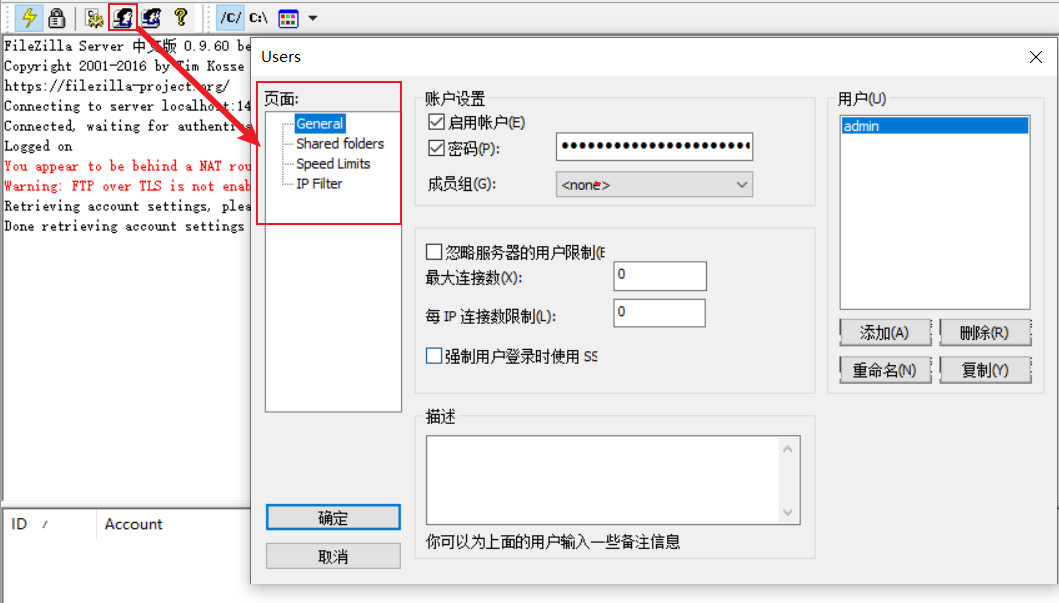


图7.1 admin用户添加

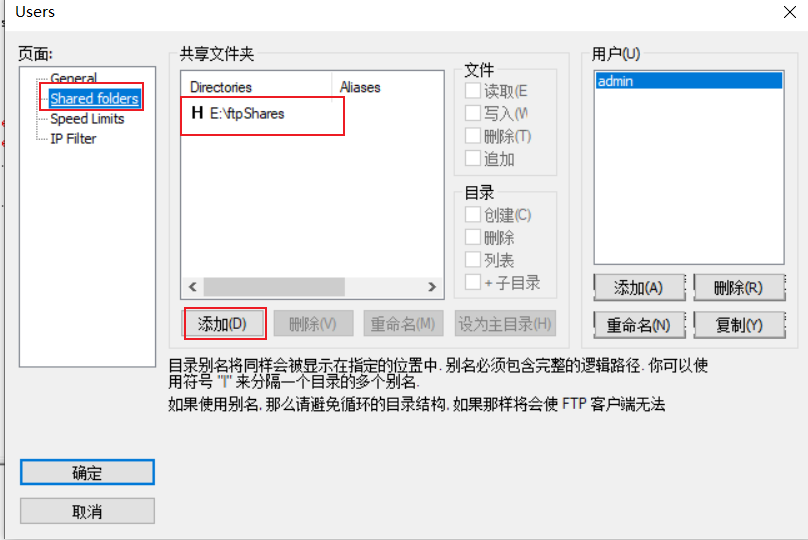


图7.2 共享目录添加

FTP服务器搭建完毕，下面我们通过Rust来实现如何在共享目录下创建新目录并且在新创建的目录下添加文件以及在文件中添加内容。Rust程序如下所示：

首先利用指令创建名为rustftp的项目：

1. cargo new rustftp

cargo.toml的内容如下：

1. [package]

2. name = "rustftp"

3. version = "0.1.0"

4. edition = "2021"

5. ​[dependencies]

6. //添加ftp依赖

7. ftp = { version="3.0.1"}

修改main.rs内容：

1. extern crate ftp;

2. use std::io::Cursor;

3. use ftp::FtpStream;

4. fn main() {

5. println!("Hello rustftp!");

6. //创建ftp连接，连接本机端口21

7. let mut ftp\_stream = FtpStream::connect("127.0.0.1:21").unwrap();

8. //登录服务器

9. let \_ = ftp\_stream.login("admin", "admin").unwrap();

10. //创建新目录在服务器上

11. let \_ = ftp\_stream.mkdir("ftpCreat").unwrap();

12. //进入当前已更新后共享目录

13. let \_ = ftp\_stream.cwd("ftpCreat").unwrap();

14. //获取当前共享目录

15. println!("Current directory:{}",ftp\_stream.pwd().unwrap());

16. //添加文件在共享目录下并且添加内容

17. let mut reader = Cursor::new("I 'm already create a file".as\_bytes());

18. let \_ = ftp\_stream.put("readFile.txt", &mut reader);

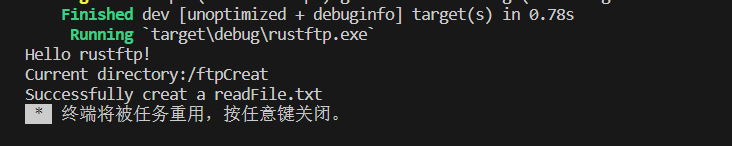
19. println!("Successfully creat a readFile.txt");

20. //终止连接

21. let \_ = ftp\_stream.quit();

22. }

运行结果如下图所示。

****

上述代码通过用FtpStream的connect方法创建FTP连接，然后采用实例中的login方法用账号和密码连接到本地ftp服务器切换到共享目录ftpShares下，接着利用FtpStream实例的mkdir方法创建目录ftpCreat并且通过cwd方法进入ftpCreat下，打印当前共享目录；之后通过FtpStream实例的put方法向该目录内添加文件readFile.txt，并向里添加内容I 'm already create a file，最后终止连接。效果如图7.3所示。

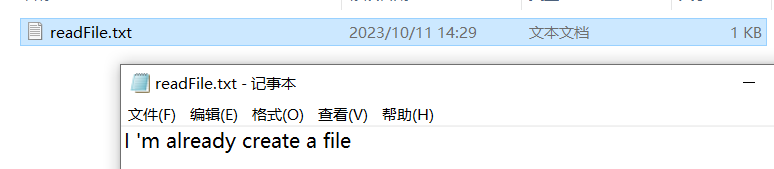


图7.3 代码演示效果

这里需要注意cursor的用法，它代表一个内存中的缓冲区，并在该缓冲区上实现读取、写入和查找；put 函数需要一个实现读取的输入，将数据包装在Cursor中会自动执行此操作。

此为简单FTP应用程序的构建和使用，详情可参考Struct [ftp](https://docs.rs/ftp/latest/ftp/index.html)::[FtpStream](https://docs.rs/ftp/latest/ftp/struct.FtpStream.html)掌握更多相关方法的使用，同时也会让你更深一步的了解FTP应用协议。

### TFTP服务应用与实现

在上一节的讲解中，我们了解了FTP文件传输协议的定义与基本特性，并演示了利用rust-ftp库与FTP服务器进行交互的过程。在本节中，我们将介绍TFTP的基本概念、特性、注意事项等，并在本小节的最后演示利用tftp sever库实现一个简单的TFTP服务实例。

1. **TFTP简介**

TFTP（Trivial File Transfer Protocol），又名简单文件传输协议，基于UDP（User Datagram Protocol）协议实现。TFTP协议最初是由RFC（Request for Comments，请求评论）定义的，主要目的是为了在网络上快速、简单地传输文件。TFTP协议最开始设计用于系统引导进程，由于报文较小，所以能快速复制这些文件。由于其简单、轻量级、易于实现，客户端服务器之间不需要复杂交互等特点，当前主要应用于网络引导、配置文件传输等场景。

1. **TFTP的特点**

TFTP是一种无连接协议，使用UDP作为传输层协议，因此没有像TCP那样的可靠性保证机制，传输过程中可能会出现数据包丢失或重复等问题，但正因其简单性，它可以被方便地嵌入到各种不同的应用中。TFTP的主要特点包括：

1. 每次传送的数据PDU（Protocol Data Unit，协议数据单元）中有512字节的数据，但最后一次传输的数据可能不足512字节。数据PDU也被称为文件块（block），每个块按序编号，从1开始。
2. TFTP支持ASCII码或二进制传送，可以对文件进行读或写操作。
3. TFTP使用非常简单的首部，没有庞大的命令集，也不支持列目录或用户身份鉴别功能。
4. TFTP主要用于文件传输，不支持交互操作。
5. TFTP的工作方式类似于停止等待协议。在发送完一个文件块后，它会等待对方的确认，确认时应指明所确认的块编号。如果在规定时间内未收到确认，它将重发数据PDU。同样，发送确认PDU的一方在规定时间内若收不到下一个文件块，也会重发确认PDU。这种方式有助于确保文件的传输不会因为某个数据包的丢失而失败。

总的来说，TFTP主要用于嵌入式系统、网络交换机等设备的固件升级，路由器、防火墙等网络设备的配置文件传输，以及上传设备日志和调试信息等场景。如需了解更多关于TFTP的信息，建议咨询计算机领域的专业人士。

1. **FTP和TFTP对比**

FTP和TFTP都是用于在网络上传输文件的协议，但它们在设计、功能和用途上有一些重要的区别。

1. 设计目标：FTP是一个完整、面向会话、常规用途的文件传输协议，而TFTP则是一个简单、特殊目的的文件传输协议。FTP旨在提供可靠、高效的文件传输服务，支持多种文件类型和传输模式，而TFTP则更注重简单性和灵活性，通常用于特定的应用场景。
2. 安全性：FTP提供了较高级别的安全性，包括数据加密和用户身份验证等机制。TFTP则没有提供这些安全特性，因此在安全性方面相对较低。
3. 传输模式：FTP支持两种传输模式：主动模式和被动模式。在主动模式下，FTP客户端主动连接到FTP服务器的指定端口，而在被动模式下，FTP服务器会创建一个新的端口来接受客户端的连接。TFTP则只支持一种传输模式，即客户端向服务器发送请求，服务器响应并发送文件。
4. 错误处理：FTP在传输过程中遇到错误时，会采取相应的措施来确保文件的完整性和正确性，例如重传损坏的数据包或重新建立连接。TFTP则没有这些错误处理机制，因此在传输过程中可能会遇到数据丢失或损坏的问题。
5. 应用场景：FTP广泛应用于各种文件传输场景，包括互联网上的文件共享、网站文件上传下载等。而TFTP则通常用于一些特定的应用场景，例如嵌入式系统中的文件传输或简单的文件共享任务。

综上所述，FTP和TFTP在设计、功能和用途上都有所不同。FTP更适合用于需要高可靠性、安全性和灵活性的文件传输场景，而TFTP则更适合用于简单、特定的文件传输任务。

1. **利用Rust实现TFTP实例**

为了深入我们对TFTP使用方法的掌握，我们使用tftp\_server库实现一个简单的TFTP服务器实例，搭建一个TFTP服务器使得服务端与客户端能够通过该服务器实现基于TFTP的通信。

首先配置manifests文件

1. [package]

2. //设置项目基本信息

3. name = "tftp-example"

4. version = "0.1.0"

5. authors = ["Foo <foo@bar.com>"]

6. [dependencies]

7. tftp\_server = "0.0.2"

配置TFTP服务器

1. //调用tftp\_sever等实现服务所需的包

2. extern crate tftp\_server;

3. use tftp\_server::server::TftpServer;

4. use std::net::SocketAddr;

5. use std::str::FromStr;

6. fn main() {

7. let addr = format!("0.0.0.0:{}", 69);

8. let socket\_addr = SocketAddr::from\_str(addr.as\_str()).expect("Error

9. parsing address");

10. //创建TFTP服务器实例

11. let mut server =

12. TftpServer::new\_from\_addr(&socket\_addr).expect("Error creating

13. server");

14. //运行实例并返回运行结果状态

15. match server.run() {

16. Ok(\_) => println!("Server completed successfully!"),

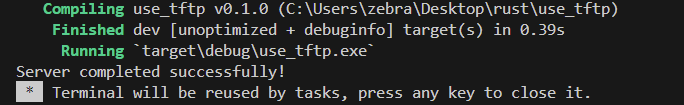
17. Err(e) => println!("Error: {:?}", e),

18. }

19. }

20.

运行结果：



在得到"Server completed successfully!"的提示后，就表明TFTP服务端配置成功了。我们可以在一个终端上运行服务器程序，在另一个终端上运行客户端程序，将客户端连接到端口69的本地主机，然后就可以从服务器端下载文件。我们同样也可以通过使用UdpSocket类完成TFTP服务器端的配置，得到的效果是一样的，代码如下所示。

1. use std::net::{UdpSocket, SocketAddr};

2. use std::io::{self, Read, Write};

3.

4. fn main() -> io::Result<()> {

5.     let bind\_addr: SocketAddr = "127.0.0.1:69".parse().unwrap();

6.     let socket = UdpSocket::bind(&bind\_addr)?;

7.

8.     loop {

9.         // 接收请求

10.         let mut buf = [0; 1024];

11.         let (size, src\_addr) = socket.recv\_from(&mut buf)?;

12.         let request = &buf[..size];

13.

14.         // 处理请求

15.         let opcode = request[0];

16.         match opcode {

17.             // RRQ (Read Request)

18.             1 => {

19.                 let filename = &request[2..request.len() - 1];

20.                 let mut file = std::fs::File::open(filename)?;

21.                 let mut buf = Vec::new();

22.                 file.read\_to\_end(&mut buf)?;

23.                 socket.send\_to(&buf, src\_addr)?;

24.             }

25.             // WRQ (Write Request)

26.             2 => {

27.                 let filename = &request[2..request.len() - 1];

28.                 let data\_len = size - 2;

29.                 let data = &buf[2..data\_len];

30.                 let mut file = std::fs::File::create(filename)?;

31.                 file.write\_all(data)?;

32.                 let ack\_packet = [0x00, 0x04, 0x00, 0x00];

33.                 socket.send\_to(&ack\_packet, src\_addr)?;

34.             }

35.             \_ => {

36.                 // 忽略其他操作码和错误处理

37.             }

38.         }

39.     }

40. }

41.

在以上代码中，利用loop循环接收请求，在接收到请求后，利用match循环依次实现对于报文的初始化与具体处理。该方法使用传统的套接字请求，代码实现过程较为复杂，这里推荐在具体使用过程中使用tftp\_sever库较为简洁的方法，得到更好的效率。

## 小结

在这一章中，我们以之前学习的内容为基础。从本质上讲，我们将网络堆栈向上移动到了应用层。首先研究了构建应用层协议的一些主要考虑因素；接着研究了RPC，特别是gRPC，研究它是如何使开发人员能够构建大规模的网络服务；然后研究了一个可以通过SMTP服务器发送电子邮件的Rust库；最后编写了FTP客户机和TFTP服务器的示例。与本书介绍的其他应用层协议一起，我们为深入理解这些协议打下了一个很好的基础。

# 安全保障

本章要点：熟悉Web保护的原理，熟悉和掌握Letsencrypt，掌握使用证书保护基于web的应用程序，掌握使用Diffie-Hellman算法交换密钥。

## 引言

在Rust语言中，网络安全是一个重要的主题。要确保Web应用程序的安全性，需要了解一些基本概念，包括加密、认证和证书的作用。

* 加密是指将数据转换为密文，以确保只有授权的人可以读取它。在Web应用程序中，加密可用于保护敏感数据，如密码和信用卡号码。通常使用对称加密和公钥加密两种加密技术。
* 认证是指确认身份或授权特定用户访问资源。在Web应用程序中，认证通常采用用户名和密码的方式，但也可以使用其他方式，如数字证书、指纹识别和生物识别技术。
* 证书是一种数字文档，可以用来验证身份和进行加密通信。在网络安全领域，证书通常用于HTTPS连接和其他加密通信协议中，以确保数据在传输时不被窃取或篡改。
* 证书通过数字签名方式保证通信双方的身份合法性，保证通信内容的完整性和机密性，确保通信的安全性，包含了发证机构、证书持有者的信息、公钥以及数字签名等必要信息。证书一般由一个权威的第三方机构颁发，并在其到期之前持续有效。证书机构发放证书时会对申请者进行身份验证，以确保其真实身份和合法权利，然后将颁发的数字证书和验证信息录入到颁发机构的证书库中。

在进行HTTPS连接时，服务器会将自己的证书发送给客户端，客户端通过验证证书的数字签名和有效期等信息来确认服务器的身份，然后使用服务器的公钥来加密通信内容。由于证书的信任链，客户端可以信任证书中的公钥并向其发送加密数据，保证了通信的机密性和完整性。在Rust语言中，通常使用Rustls库来进行加密通信和操作证书。Rustls提供了一系列API，可以用来验证证书、验证服务器身份、生成自己的证书等。使用Rustls库，可以确保Web应用程序的通信安全，并防止因为证书问题而导致的数据泄露和安全漏洞。

以上这些概念在Web应用程序安全中扮演着非常重要的角色，在进行数据传输和用户认证时，应该采取适当的加密措施、使用可靠的认证方式，并确保证书的有效性和准确性，这样才能更好地保障Web应用程序的安全性。

## 利用证书保障Web应用的安全性

Rust中主要通过数字证书保障Web应用的安全性。数字证书是由权威公正的第三方机构即CA中心签发，它是以数字证书为核心的加密技术，可以对网络上传输的信息进行加密和解密、数字签名和签名验证，确保网上传递信息的机密性、完整性，以及交易实体身份真实性，签名信息不可否认性，从而保障网络应用的安全性。

### Rust 中常用的数字证书库和工具

在Rust中，有几个常用的数字证书库和工具可用于保障Web应用的安全性，如rustls、native-tls、openssl等，下面将逐个介绍它们的特点、用法和性能比较。

1. **rustls**

rustls是一个纯Rust实现的TLS库，它提供了简单且易于使用的API接口，适合用于构建安全的Web应用程序。它支持最新的TLS1.3协议，具有丰富的功能，如客户端和服务端的验证、证书链验证和密钥交换算法等。

使用rustls进行加密通信非常简单，首先需要加载服务器的证书和私钥，并创建一个rustls::ServerConfig对象；然后调用rustls::ServerSession::new(&config)创建一个TLS会话；最后使用会话对象处理来自客户端的请求和响应数据。

rustls在性能方面表现良好，具有较低的内存占用和快速的握手速度。它还针对性能进行了优化，可以在大流量和高并发的情况下提供高效的安全通信。

1. **native-tls**

native-tls是一个与操作系统本地TLS库集成的Rust库，它使用操作系统提供的加密算法和证书存储机制，具有良好的交叉平台支持，适用于需要与操作系统和本地证书存储进行交互的场景。

使用native-tls创建TLS连接时，首先使用native\_tls::TlsConnector加载服务器证书和私钥，并创建一个TLS连接。然后使用连接对象进行加密通信。

由于native-tls集成了操作系统的本地TLS库，因此性能与操作系统的原生实现相当，因此在某些场景下，可能会比纯Rust实现的库更高效。

1. **openssl**

openssl是一个强大且广泛使用的开源加密库，它提供了一系列的加密算法和功能。虽然openssl并不是Rust原生库，但Rust提供了与openssl进行绑定的接口，适用于需要使用特定OpenSSL特性的高级应用场景。

使用openssl可以进行高级的加密和证书操作，如可以使用rust openssl来加载和解析证书、创建TLS连接等。由于openssl是一个庞大的加密库，它可能对性能有一些影响，此外由于其非Rust原生实现，可能会导致一些安全隐患，因此在Rust项目中使用openssl库时，需要谨慎处理并确保安全性。

以下是一个使用rustls进行TLS服务器连接的最简化的Rust代码。这段代码实现了一个基于rustls的简单TCP服务器，能够接收客户端的连接请求，使用rustls保证通信的安全性，并对客户端发送的数据作出相应的响应。

1. use std::io::{Read, Write};

2. use std::net::TcpListener;

3. use rustls::ServerConfig;

4. use rustls::Session;

5. use rustls::StreamOwned;

6.

7. fn main() {

8. let config = ServerConfig::new(rustls::NoClientAuth::new());

9. let listener = TcpListener::bind("127.0.0.1:8443").expect("Failed to bind TCP listener");

10. for stream in listener.incoming() {

11. match stream {

12. Ok(stream) => {

13. let mut session= rustls::ServerSession::new(&std::sync::Arc::new(config.clone()));

14. let mut stream = StreamOwned::new(session, stream);

15. let mut buf = [0; 1024];

16. let \_ = stream.read(&mut buf).unwrap();

17. let \_ = stream.write\_all(b"Hello, this is a secure connection using rustls.").unwrap();

18. }

19. Err(e) => println!("Error accepting connection: {}", e),

20. }

21. }

22. }

本代码创建了一个 TCP 监听器，并通过 rustls::ServerSession 创建了一个服务器会话。在每个接受的连接上，通过 rustls::StreamOwned 包装会话和 TCP 流，实现了安全的通信。代码运行后，会创建一个基于 rustls 的 TCP 服务器，并且开始侦听 127.0.0.1:8443。当有客户端连接到这个服务器时，服务器会发送一条简单的消息："Hello, this is a secure connection using rustls."。

代码中第8行创建一个服务器配置 config，该配置指定了服务器的安全策略。这里使用了 NoClientAuth，表示服务器不要求客户端进行身份验证；第9行创建一个 TCP 监听器，并绑定到本地地址 127.0.0.1 的 8443 端口，如果绑定失败，会打印错误信息；第10行通过 incoming 方法获取到的迭代器，遍历所有接收到的客户端的连接；第12行的Ok(stream) => { ... }对于每个成功接收的客户端连接，执行匹配分支；第13行let mut session = rustls::ServerSession::new(&std::sync::Arc::new(config.clone()))为当前连接创建一个服务器会话 session，使用之前创建的服务器配置 config 进行初始化；第14行创建一个针对当前连接的 StreamOwned 对象，该对象封装了服务器会话和套接字流；第15行let mut buf = [0; 1024]创建一个大小为 1024 字节的临时缓冲区 buf，用于接收从客户端发送的数据；第16行let \_ = stream.read(&mut buf).unwrap()从套接字流中读取数据，并将其存储到缓冲区 buf 中。在这个示例中，我们忽略了读取字节数的返回结果；第17行let \_ = stream.write\_all(b"Hello, this is a secure connection using rustls.\n").unwrap()向客户端发送一条消息。

### 在 Rust 中使用 TLS/SSL 进行加密通信

在 Rust 中可以使用 rustls 库来实现 TLS/SSL 加密通信，以保护 Web 应用的通信安全，通信步骤如下。

1. 生成和使用证书

使用 OpenSSL 或类似工具生成一对公钥和私钥。私钥需要保持安全，以便服务器端进行证书签名和解密通信数据。使用私钥对公钥进行签名，生成自签名证书，该证书将用于服务器端身份验证。在客户端，可能需要加载服务器的证书作为受信任的根证书，用于进行服务器身份验证。

1. Rust 代码设置服务器端

首先引入 rustls 和其他相关库的依赖，然后创建一个 rustls::ServerConfig 实例，用于配置服务器端的 TLS/SSL 设置，接下来为 ServerConfig 设置相应的证书链和私钥，以进行服务器的身份验证和解密通信数据，最后监听服务器的地址和端口，接受来自客户端的连接。对于每个连接，使用 rustls::ServerSession 创建一个服务器会话，并使用 rustls::StreamOwned 包装会话和 TCP 流，以实现安全的通信。

1. Rust 代码设置客户端

首先依赖 rustls 和其他相关库，然后创建一个rustls::ClientConfig 实例，用于配置客户端的 TLS/SSL 设置。接下来，配置客户端加载受信任的根证书，以进行服务器身份验证，最后创建一个 TCP 连接到服务器的地址和端口。对于每个连接，使用 rustls::ClientSession 创建一个客户端会话，并使用 rustls::StreamOwned 包装会话和 TCP 流，以实现安全的通信。

通过以上步骤，在 Rust 中可以使用 rustls 来配置服务器和客户端 TLS/SSL 设置，并通过加密通信来保护 Web 应用的通信安全。这样可以确保安全传输敏感信息，并防止数据被中间人窃听或篡改。

### 服务器证书的验证与管理

在 Rust 中，确保服务器证书的合法性是保护 Web 应用安全的一个重要方面。在 Rust 中验证服务器证书的合法性并管理证书的步骤和相关概念如下。

1. **证书链验证**

在 Rust 中，验证服务器证书的合法性通常涉及验证其所属的证书链。可以使用 rustls 库的 ServerConfig 或 ClientConfig 对象来配置证书链验证。在配置验证时，需要提供可信的根证书或受信任的证书颁发机构（CA）根证书。rustls 库会自动检查证书链的有效性和完整性，以确保服务器证书可以追溯到受信任的根证书。

1. **证书吊销列表 (CRL)**

证书吊销列表 (CRL) 是一种证书颁发机构（CA）广播的用于标识已经吊销的证书的机制。在 Rust 中，可以通过 rustls 的 ClientConfig 或 ServerConfig 对象启用和配置 CRL 的使用。在配置时，需要提供 CRL 文件或 URL 来检查服务器证书是否在吊销列表中。rustls 库会定期验证服务器证书是否存在于 CRL 中，以确保证书的合法性。

1. **证书管理和更新**

对于服务器证书，更新通常涉及生成新的证书和私钥，并将新证书与私钥配对。在 Rust 中，可以使用 OpenSSL 或类似的工具生成新的证书和私钥。更新后，需要使用新证书和私钥配置 rustls 的 ServerConfig 对象，以便服务器可以使用新证书进行身份验证和加密通信。可以使用 rustls 库提供的方法来加载和配置新证书和私钥，例如使用 `load\_cert\_chain()` 方法加载新的证书和私钥文件，并将其设置到 ServerConfig 对象中。这样服务器就可以使用更新后的证书和私钥进行安全通信了。

通过以上步骤，在 Rust 中可以验证服务器证书的合法性，包括证书链验证和 CRL 的使用，同时，通过证书管理和更新机制，可以确保服务器证书的及时更新和替换，从而保护 Web 应用的安全性，这样可以防止恶意证书或已吊销的证书被接收，并确保与服务器的通信是安全可信的。

### 防御网络攻击的常见方法

为了保护 Web 应用免受常见的网络攻击，如中间人攻击、重放攻击、暴力破解等，在 Rust 中保护 Web 应用免受网络攻击的常见方法如下。

1. **中间人攻击防御**

中间人攻击 (MITM) 是一种常见的网络攻击，攻击者可以在网络通信过程中窃取或篡改数据，为了防止中间人攻击，可以使用 TLS/SSL 加密通信。在 Rust 中，可以使用 rustls 库来实现 TLS/SSL 加密通信，并配置证书链验证和证书吊销列表 (CRL)。通过使用 rustls 库，可以确保与服务器建立安全的加密连接，并验证服务器证书的合法性；配置证书链验证和启用 CRL 可以进一步增加安全性，确保服务器证书的有效性，这样可以有效地防止中间人攻击，并保护数据的安全性和完整性。

1. **安全协议的版本选择**

确保选择安全的协议版本对于保护 Web 应用安全至关重要。在 Rust 中，可以使用 rustls 库来自动选择最新和最安全的协议版本，也可以通过手动配置 ClientConfig 或 ServerConfig 对象来选择支持的协议版本。rustls 库会自动维护一个协议版本列表，并根据安全性和兼容性选择合适的协议版本，同时，也可以使用版本方法来手动指定支持的协议版本，或者使用特定的方法来禁用不安全的协议版本。为了保持 Web 应用的安全性，建议及时更新 rustls 库和相关依赖项，以确保使用的协议版本和密码套件是最新且安全的。这样能最大程度地提高应用的安全性，减少潜在的安全风险。

1. **安全加密套件的配置**

选择适当的加密套件是 Web 应用安全的重要方面。确保选择强大的加密套件可以确保通信数据的加密强度和稳定性，加强应用的安全性。在 Rust 中，使用 rustls 库，可以使用 ClientConfig 或 ServerConfig 对象来手动配置合适的加密套件，提高通信安全。可以使用 `ciphersuites` 方法来指定支持的加密套件列表，或使用具有预定义套件的方法，如 `modern()`、`compat()`、`tls13() `等方法来指定支持的加密套件。同时，也可以使用 `version` 方法来指定应该使用的协议版本。在选择适当的加密套件时，需要考虑多个因素，如安全性、性能、兼容性等。应该使用具有强大安全性的加密套件，比如 AES、SHA 等标准的安全算法，同时还要考虑套件对性能和兼容性的影响。综上所述，通过选择合适的加密套件，可以保证通信数据的加密强度和稳定性，从而提高应用的安全水平。

在 Rust 中可以采取各种有效措施，以确保 Web 应用程序免受网络攻击的威胁。这些最佳实践包括中间人攻击防御、安全协议的版本选择和安全加密套件的配置，不仅可以提高 Web 应用程序的安全性，而且可以改进用户体验和用户的机密信息的保护。

### 自动化证书部署和更新

在现代网络应用中，使用SSL/TLS协议来保护用户数据的安全性已经成为一项重要的任务。为了实现安全的通信，网站需要安装和配置数字证书。然而，证书的管理和更新通常是一项繁琐且容易出错的任务。为了简化这个过程，自动化证书管理工具和技术应运而生。

自动化证书管理是一种通过使用工具和协议来实现自动化部署和更新SSL/TLS证书的方法。这种方法可以帮助简化证书管理的流程，并确保网站在安全连接中使用最新有效的证书。在Rust Web应用中，可以使用Let’s Encrypt、ACME协议、Certbot等工具和技术来实现自动化证书的部署和更新。

1. **ACME 协议**

ACME（Automatic Certificate Management Environment）是一种用于自动化证书颁发和管理的协议，通过验证域名所有权来生成证书，而不需要手动的证书签名请求（Certificate Signing Request），它还定义了如何与证书颁发机构（如 Let’s Encrypt）进行交互，以进行域名验证、证书签发和证书更新等操作。

1. **Let’s Encrypt**

Let’s Encrypt 是一个免费的、开放的、非盈利的数字证书颁发机构（CA），旨在推动整个Web从HTTP到HTTPS的转变。它提供了一个简单、自动化的方式来获取和管理 SSL/TLS证书，通过使用了ACME协议，我们可以通过程序化的方式自动请求和更新证书，而无需手动介入，为自动化证书部署和更新提供了便利。

1. **Certbot**

Certbot是一个用于自动化证书管理的命令行工具。它支持 ACME 协议，并提供了与 Let’s Encrypt一起使用的简化流程。Certbot可以帮助用户自动验证域名所有权、请求并获取证书，并自动将证书配置到 Web 服务器中。

Rust Web 应用中实现自动化证书部署和更新如下：

1. **集成 ACME 协议库**

在 Rust 中，有一些ACME协议的库可以用来与ACME服务器进行交互。我们可以选择一个与 Let’s Encrypt 兼容的库，如acme-lib（或类似的库），并将其添加到我们的Rust Web应用的依赖中。

1. **域名验证**

在申请证书之前，我们需要通过域名验证来证明我们拥有要申请证书的域名。这通常涉及将特定的验证文件（例如HTTP文件或DNS TXT记录）添加到你的域名配置中，我们可以通过编写代码来自动完成这些验证步骤，并确保在Certbot或ACME协议库的帮助下正确配置域名验证。

1. **请求和获取证书**

在代码中，我们可以使用ACME协议库提供的功能来请求和获取证书。这通常需要提供与ACME服务器通信所需的配置信息和证书请求数据。一旦请求成功并得到响应，我们将获得一个证书文件和相关的密钥文件。

1. **配置证书到 Web 服务器**

在获取到证书文件后，需要将其配置到我们RustWeb应用的Web服务器中（如 Nginx、Apache 等），我们可以编写脚本或使用相应的库来自动化这个过程，以便将证书文件正确地配置到服务器。

1. **自动更新证书**

Let**’**sEncrypt的证书有一个较短的有效期，通常为90天，因此，我们需要定期自动更新证书，以确保网站始终使用有效的证书。我们可以编写一个后台任务来定期检查证书的到期日期，并在适当的时候自动触发证书更新。

自动化证书部署和更新是通过使用 ACME 协议和相应的工具、技术来简化证书管理过程的一种方法。在RustWeb应用中，我们可以使用acme-client等 Rust库来实现自动化证书管理，确保我们的Web应用始终使用最新的、有效的证书来保护用户数据的安全性。

## 利用Ring Crate进行密钥交换

本节我们学习一个专门用于密码学操作的Ring库，这是一个非常有用的Rust库，它提供了一套完整的密码学功能，可以帮助我们实现安全的加密和认证功能。

密码学是关于保护数据和通信安全的科学，它涉及加密、解密、数字签名和验证等技术。在现代社会中，数据安全变得越来越重要，我们需要使用密码学来保护我们的个人信息、敏感数据和通信内容，Ring库就是为了满足这个需求而设计的。

Ring库经过广泛的测试和审查，遵循现代密码学的最佳实践，可以保护我们的数据和通信安全，此外，Ring库提供了一套完整的密码学功能，包括对称加密算法、非对称加密算法、密钥派生函数、消息认证码等，这意味着我们可以使用ring库来实现数据加密和解密、数字签名和验证、密钥交换、消息完整性和真实性验证等操作。

Ring库提供了清晰的API和详细的文档，帮助我们快速上手并使用库中的功能，同时Ring库还具有优化的性能，它使用了高效的算法和数据结构，可以在处理密码学操作时提供高性能的执行速度。

由于Ring库是用Rust语言编写的，可以在各种平台上运行，包括Windows、Linux、macOS等，这使得我们可以在不同的操作系统上使用相同的代码库，提高了代码的可移植性和跨平台性。

Ring crate可以实现的功能对称/非对称加密算法、密钥派生函数、消息认证码、随机数生成等多种功能。

### 对称加密算法

Ring库提供了多种对称加密算法，用于数据的加密和解密，支持的常见对称加密算法如下.

1. **AES（Advanced Encryption Standard）**

AES是一种广泛使用的对称加密算法，被认为是目前最安全和最常用的对称加密算法之一。Ring库提供了对AES的支持，可以使用不同的密钥长度（128位、192位、256位）进行数据的加密和解密。

1. **ChaCha20**

ChaCha20是一种快速且安全的对称加密算法，适用于各种平台和设备，在性能和安全性之间取得了很好的平衡。Ring库提供了对ChaCha20的支持，可以使用不同的密钥和随机数进行数据的加密和解密。

1. **Poly1305**

Poly1305是一种消息认证码算法，用于验证消息的完整性和真实性，它通常与ChaCha20一起使用，提供了高效的加密和认证功能。Ring库提供了对Poly1305的支持，可以用于生成和验证消息的认证码。

除了以上提到的算法，Ring库还支持AES-GCM（AES加密和GCM模式的组合）、AES-CCM（AES加密和CCM模式的组合）等其他算法，这些算法都经过了广泛的测试和审查，以确保其在实际应用中的安全性和可靠性。

这里以AES算法为例，展示一下Ring crate的使用，具体的思路如下：

1. 导入所需的Ring库和rand模块。
2. 在主函数中，首先生成一个随机的32字节密钥，用于加密和解密操作。
3. 定义明文数据，这里使用的是"Hello, World!"。
4. 创建AES-GCM加密器所需的算法和随机数。这里使用的是AES-256-GCM算法，并生成一个12字节的唯一nonce（用于确保每次加密的结果都不同）。
5. 创建加密器的sealing\_key，使用之前生成的密钥。
6. 创建一个足够大的缓冲区ciphertext来存储加密后的数据。
7. 使用sealing\_key对明文数据进行加密，将结果存储在ciphertext中，并返回一个tag（用于验证数据的完整性）。
8. 创建解密器的opening\_key，使用之前生成的密钥。
9. 创建一个足够大的缓冲区decrypted来存储解密后的数据。
10. 使用opening\_key对ciphertext进行解密，将结果存储在decrypted中。
11. 输出明文数据、加密后的数据和解密后的数据。
12. 代码的运行结果将显示明文数据、加密后的数据（以字节数组形式显示），以及解密后的数据（以字符串形式显示）。

示例代码如下：

1.use ring::aead;

2.use ring::rand::SystemRandom;

3.use ring::rand::SecureRandom;

4.

5.fn main() {

6.   // 生成随机密钥

7.    let mut key = [0u8; 32];

8.    let rng = SystemRandom::new();

9.    rng.fill(&mut key).unwrap();

10.

11.    // 明文数据

12.    let plaintext = b"Hello, World!";

13.

14.    // 创建AES-GCM加密器

15.    let algorithm = &aead::AES\_256\_GCM;

16.    let nonce = aead::Nonce::assume\_unique\_for\_key([0u8; 12]);

17.    let unbound\_key = aead::UnboundKey::new(algorithm, &key).unwrap();

18.    let sealing\_key = aead::LessSafeKey::new(unbound\_key);

19.    let mut in\_out = plaintext.to\_vec();

20.    in\_out.resize(plaintext.len() + algorithm.tag\_len(), 0);

21.    sealing\_key.seal\_in\_place\_append\_tag(nonce, aead::Aad::empty(), &mut in\_out).unwrap();

22.

23.    // 解密数据

24.    let nonce = aead::Nonce::assume\_unique\_for\_key([0u8; 12]);

25.    let unbound\_key = aead::UnboundKey::new(algorithm, &key).unwrap();

26.    let opening\_key = aead::LessSafeKey::new(unbound\_key);

27.    let decrypted = opening\_key.open\_in\_place(nonce, aead::Aad::empty(), &mut in\_out).unwrap();

28.

29.    // 输出结果

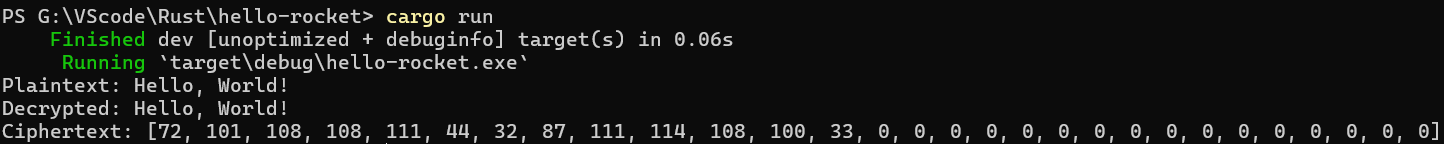
30.    println!("Plaintext: {}", String::from\_utf8\_lossy(plaintext));

31.    println!("Decrypted: {}", String::from\_utf8\_lossy(decrypted));

32.    println!("Ciphertext: {:?}", &in\_out[..plaintext.len() + algorithm.tag\_len()]);

33.}

运行之后会有以下结果：



Plaintext: 显示明文数据 "Hello, World!"。Decrypted: 显示解密后的数据，应该与原始明文数据相同。Ciphertext: 显示加密后的数据，包括验证标签。这段代码演示了如何使用 ring 库来进行 AES-GCM 加密和解密操作。下面将具体解释以上程序中的关键代码。

use ring::aead: 导入 ring 库中的 AEAD（Authenticated Encryption with Associated Data）模块，用于进行加密和解密操作。

use ring::rand::SystemRandom: 导入 ring 库中的随机数生成模块，用于生成随机密钥。

use ring::rand::SecureRandom: 导入 ring 库中的安全随机数生成模块，用于生成随机数。

fn main() 定义了程序的入口点。

let mut key = [0u8; 32]: 这一行声明了一个可变的名为 key 的数组，数组中包含了 32 个 8 位无符号整数（即字节）。这里用来存储生成的随机密钥，密钥长度为 32 字节。

let rng = SystemRandom::new(): 创建了一个系统随机数生成器实例 rng。这个生成器用于生成随机数，是基于操作系统的安全随机数生成器。

rng.fill(&mut key).unwrap(): 使用系统随机数生成器 rng 来填充 key 数组。.fill() 方法用于填充给定的可变引用，这里即填充了数组 key。如果填充过程中出现错误，则通过 .unwrap() 方法触发 panic。

let plaintext = b"Hello, World!": 这一行定义了一个名为 plaintext 的字节字符串。在 Rust 中，b"" 表示字节字符串。这里的 plaintext 存储了要加密的明文数据。

let algorithm = &aead::AES\_256\_GCM: 定义了一个不可变引用 algorithm，指向 AES-GCM 算法的实例。这个算法用于加密和解密数据，采用了 256 位的密钥长度。

let nonce = aead::Nonce::assume\_unique\_for\_key([0u8; 12]): 这行代码创建了一个用于加密的 nonce（一次性数字）。在加密过程中，nonce 用于确保加密数据的唯一性和完整性。这里使用了长度为 12 字节的全零数组作为 nonce。

17和18行代码，创建了用于 AES-GCM 加密的密钥。首先，使用aead::UnboundKey::new 函数将密钥绑定到 AES-GCM 算法上，然后使用 aead::LessSafeKey::new 函数创建一个较低级别的密钥对象，用于实际的加密操作。

let mut in\_out = plaintext.to\_vec(): 这行代码将明文数据复制到一个名为 in\_out 的可变字节向量中。这个向量将用于存储加密后的数据，同时也将在加密过程中被修改。

in\_out.resize(plaintext.len() + algorithm.tag\_len(), 0): 这行代码调整了 in\_out 的大小，以容纳加密后的数据和验证标签。这里的 algorithm.tag\_len() 返回了加密算法使用的验证标签的长度，这个长度会影响加密后数据的总长度。

sealing\_key.seal\_in\_place\_append\_tag(nonce, aead::Aad::empty(), &mut in\_out).unwrap()：这行代码调用了 seal\_in\_place\_append\_tag 方法来对 in\_out 中的数据进行 AES-GCM 加密，并在数据末尾添加验证标签。如果加密过程中出现错误，则通过 .unwrap() 方法触发 panic。

### 非对称加密算法

Ring库提供了多种非对称加密算法，用于生成和验证数字签名、进行密钥交换等操作，支持的非对称加密算法如下。

1. **RSA（Rivest-Shamir-Adleman）**

RSA是一种常用的非对称加密算法，用于生成和验证数字签名、进行密钥交换等操作。你的代码中使用的 RsaKeyPair::generate\_pkcs8和 encrypt\_blinded、decrypt方法在 ring库中并不存在。ring库的 RSA 功能目前只支持签名和验证，不支持加密和解密。

1. **ECC（Elliptic Curve Cryptography）**

ECC是一种基于椭圆曲线的非对称加密算法，具有较短的密钥长度和高强度的安全性。Ring库提供了对ECC的支持，包括生成ECC密钥对、进行加密和解密操作，以及生成和验证数字签名。

1. **Diffie-Hellman（DH）**

Diffie-Hellman是一种密钥交换协议，用于在不安全的通信渠道上协商共享密钥。Ring库提供了对Diffie-Hellman的支持，可以生成Diffie-Hellman密钥对，进行密钥交换操作。

1. **ECDH（Elliptic Curve Diffie-Hellman）**

ECDH是基于椭圆曲线的Diffie-Hellman密钥交换协议，具有较短的密钥长度和高强度的安全性。Ring库提供了对ECDH的支持，可以生成ECDH密钥对，进行密钥交换操作。

### 密钥派生函数

Ring库提供了一些常用的密钥派生函数，用于从一个密码或密钥生成更强大的密钥。这些函数可以帮助我们在密码学应用中生成安全的密钥，并提供更高的安全性。以下是Ring库中一些常用的密钥派生函数。

1. **PBKDF2（Password-Based Key Derivation Function 2）**

PBKDF2是一种基于密码的密钥派生函数，它使用一个密码和一个盐值作为输入，通过多次迭代的哈希计算来生成密钥，可以增加破解密码的难度，提高密码的安全性。

1. **HKDF（HMAC-based Extract-and-Expand Key Derivation Function）**

HKDF是一种基于HMAC的密钥派生函数，它使用一个输入密钥、一个盐值和一个上下文信息作为输入，通过两个阶段的处理（提取和扩展）生成密钥，可以从一个较短的输入密钥派生出更长、更强的密钥。

这些密钥派生函数在Ring库中都有相应的实现，并提供了简单易用的API，我们可以根据具体的需求选择合适的密钥派生函数，并使用它们来生成安全的密钥。需要注意的是，在使用密钥派生函数时，我们应该选择合适的参数，如迭代次数、盐值的长度和生成密钥的长度等，以确保生成的密钥具有足够的安全性。此外，密钥派生函数的性能也是需要考虑的因素，我们可以根据具体的应用场景选择适当的密钥派生函数。

这里以HKDF算法为例，展示一下Ring crate的使用，具体的思路如下：

1. 导入所需的Ring库和hkdf模块。
2. 在主函数中，定义输入密钥、盐值和上下文信息。
3. 创建一个足够大的缓冲区output\_key来存储输出密钥。
4. 使用hkdf::derive函数，传入HKDF\_SHA256算法、盐值、输入密钥、上下文信息和output\_key，进行密钥派生。
5. 输出输入密钥、盐值、上下文信息和输出密钥。
6. 代码的运行结果将显示输入密钥、盐值、上下文信息和输出密钥（以字节数组形式显示）。

代码如下。

1. use ring::hkdf;

2. use ring::rand::SystemRandom;

3.

4. fn main() {

5. // 定义输入密钥、盐值和上下文信息

6. let input\_key = b"input key";

7. let salt = b"salt";

8. let info = b"context info";

9.

10. // 生成输出密钥

11. let mut output\_key = [0u8; 32];

12. hkdf::derive(&hkdf::HKDF\_SHA256, Some(salt), input\_key, info, &mut output\_key);

13.

14. // 输出结果

15. println!("Input Key: {:?}", input\_key);

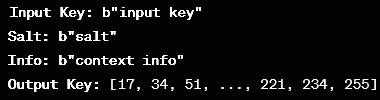
16. println!("Salt: {:?}", salt);

17. println!("Info: {:?}", info);

18. println!("Output Key: {:?}", output\_key);

19. }

运行之后会有以下结果。



### 消息认证码

Ring库提供了多种消息认证码（MAC）算法，用于验证消息的完整性和真实性。下面是Ring库支持的一些常见消息认证码算法。

1. **HMAC（Hash-based Message Authentication Code）**

HMAC是一种基于哈希函数的消息认证码算法，它使用一个密钥和一个哈希函数，将消息与密钥进行混合运算，生成一个固定长度的认证码。Ring库支持多种哈希函数，如SHA-256、SHA-384、SHA-512等，可以与HMAC一起使用。

1. **Poly1305**

Poly1305是一种高性能的消息认证码算法，它用一个密钥和一个多项式函数，将消息与密钥进行混合运算，生成一个固定长度的认证码，用于验证消息的完整性和真实性。Ring库提供了对Poly1305的支持。

使用Ring库的消息认证码功能，可以轻松地生成和验证消息的认证码，确保消息的完整性和真实性，这对于保护数据的完整性和防止篡改非常重要。我们可以选择适合自己需求的消息认证码算法，并使用Ring库提供的API进行操作。

这里以HMAC算法为例，展示一下Ring crate的使用，具体的思路如下：

1. 导入所需的Ring库和rand模块。
2. 在主函数中，首先生成一个随机的32字节密钥。
3. 定义消息，这里使用的是"Hello, World!"。
4. 创建一个hmac::SigningKey，使用SHA256哈希算法和密钥，用于生成HMAC认证码。
5. 使用hmac::sign函数，传入签名密钥和消息，生成HMAC认证码。
6. 创建一个hmac::VerificationKey，使用SHA256哈希算法和密钥，用于验证HMAC认证码。
7. 使用hmac::verify函数，传入验证密钥、消息和HMAC认证码，返回验证结果。
8. 输出消息、HMAC认证码（以字节数组形式显示）和验证结果（true表示验证通过，false表示验证失败）。
9. 代码的运行结果将显示消息、HMAC认证码和验证结果。

代码如下所示。

1.use ring::{hmac, rand};

2.use ring::rand::SecureRandom;

3.

4.fn main() {

5.   // 创建一个新的随机数生成器

6.    let rng = rand::SystemRandom::new();

7.

8.    // 生成一个随机密钥

9.    let mut key\_value = [0u8; 48];

10.    rng.fill(&mut key\_value).unwrap();

11.    let key = hmac::Key::new(hmac::HMAC\_SHA256, &key\_value);

12.

13.    // 定义一个消息

14.    let message = "Hello, world!";

15.

16.    // 使用HMAC算法和密钥对消息进行签名

17.    let signature = hmac::sign(&key, message.as\_bytes());

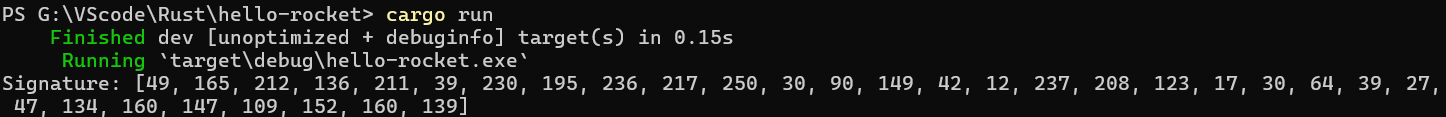
18.

19.    // 打印签名

20.    println!("Signature: {:?}", signature.as\_ref());

21.}

运行结果如下所示。



### 随机数生成器

ring库提供了一个安全的随机数生成器，用于生成密码学安全的随机数。这个随机数生成器是基于操作系统提供的安全随机数源的，以确保生成的随机数具有高度的随机性和不可预测性。

使用ring库的随机数生成器，可以执行以下操作。

1. **生成随机字节序列**

可以使用ring::rand::SystemRandom来生成随机的字节序列。例如，可以使用fill方法来填充一个字节数组，以生成随机的密钥或初始化向量。

1.use ring::rand::{SecureRandom, SystemRandom};

2.

3.fn main() {

4.    let mut key = [0u8; 32];

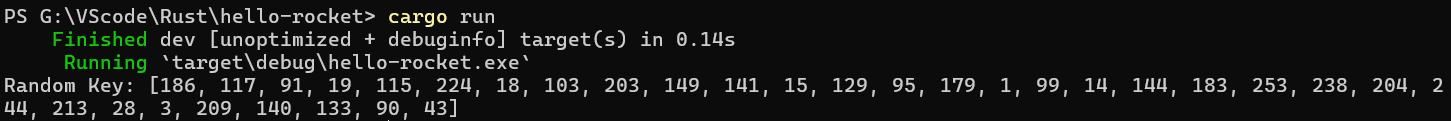
5.    let rng = SystemRandom::new();

6.    rng.fill(&mut key).unwrap();

7.    println!("Random Key: {:?}", key);

8.}

运行结果如下。

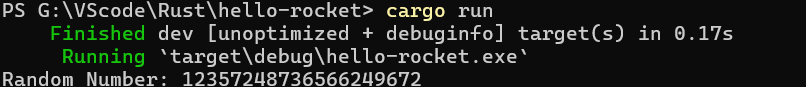


1. **生成随机整数**

可以使用ring::rand::SystemRandom生成随机的整数。例如，可以使用gen方法生成一个随机的64位整数。

|  |
| --- |
| 1. use ring::rand::SecureRandom;  2. fn main() {  3. let rng = ring::rand::SystemRandom::new();  4. let mut buffer = [0u8; 8];  5. rng.fill(&mut buffer).unwrap();  6. let random\_number = u64::from\_le\_bytes(buffer);  7. println!("Random Number: {}", random\_number);  8. } |

运行结果如下。



注意，使用随机数生成器时，应始终使用来自可信的安全随机数源的生成器，如SystemRandom。这样可以确保生成的随机数具有足够的随机性和安全性，以满足密码学应用的要求。

### 小结

Ring crate是一个用于密码学操作的功能强大且可靠的Rust库，它提供了一套完整的密码学功能，可以帮助开发者实现安全的加密和认证功能，适用于构建安全的有加密和认证功能的Rust应用程序。它遵循现代密码学的最佳实践，经过广泛的测试和审查，以确保在实际应用中的可靠性和安全性。

## 示例

通信中的一个常见问题是对信息进行加密，使第三方无法破译它。在私钥系统中，客户机和服务器都需要就使用的密钥达成一致。现在，这个密钥不能在不安全的连接上以明文形式传输。Diffie-Hellman密钥交换方法定义了一种机制，在该机制中，通过安全链路进行对话的双方可以协商一个密钥，该密钥在他们之间共享，但没有通过连接传输，该方法在许多平台上都有许多实现。

下面我们利用上面所学知识来搭建客户机和服务器，并以Diffle-Hellman密钥交换方法进行传输信息的加密和解密。

首先我们可通过下图来了解协议的工作原理。



具体步骤如下所示。

1. 首先建立TCP会话;
2. 服务器和客户端都生成私钥和公钥;
3. 客户端将其公钥发送给服务器;
4. 服务器通过发送它生成的公钥来响应;
5. 双方都可以使用各自的私钥和收到的公钥生成共享密钥；
6. 进一步的通信可以使用共享密钥进行加密。

下面，通过一个简单的例子来详细阐述Diffle-Hellman密钥交换方法的工作原理。

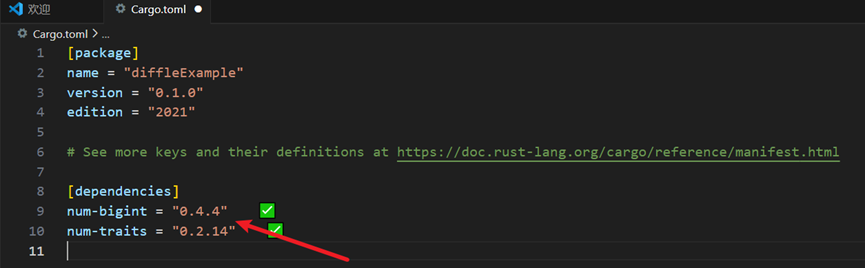
首先创建项目：Cargo new diffleExample



然后在vscode编译器中打开并在cargo.toml中添加以下依赖：

num-bigint = "0.4.4"

num-traits = "0.2.14"



之后在main.rs中添加如下代码：

1. use num\_bigint::BigUint;

2.

3. fn main() {

4.     // 公共素数和生成元

5.     let prime = BigUint::from(2300 as u32);

6.     let generator = BigUint::from(500 as u32);

7.

8.     // Alice 和 Bob 分别生成私钥

9.     let alice\_private\_key = BigUint::from(6 as u32);

10.     let bob\_private\_key = BigUint::from(15 as u32);

11.

12.     // Alice 计算公钥

13.     let alice\_public\_key = generator.modpow(&alice\_private\_key, &prime);

14.

15.     // Bob 计算公钥

16.     let bob\_public\_key = generator.modpow(&bob\_private\_key, &prime);

17.

18.     // Alice 和 Bob 交换公钥后计算共享密钥

19.     let alice\_shared\_secret = bob\_public\_key.modpow(&alice\_private\_key, &prime);

20.     let bob\_shared\_secret = alice\_public\_key.modpow(&bob\_private\_key, &prime);

21.

22.     // 打印共享密钥

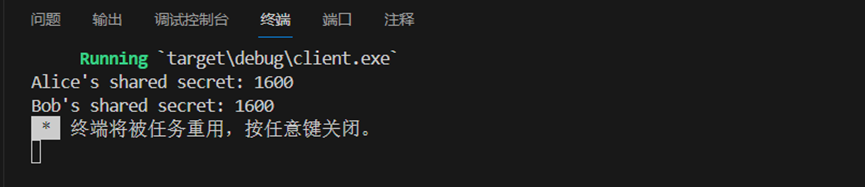
23.     println!("Alice's shared secret: {}", alice\_shared\_secret);

24.     println!("Bob's shared secret: {}", bob\_shared\_secret);

25. }

26.

运行结果如下图所示：



通过本章节，我们快速了解了Diffle-Hellman密钥交换方法的使用，这个机制的巧妙之处在于需要安全通信的双方用这个方法确定[对称密钥](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E5%AF%86%E9%92%A5/7453635?fromModule=lemma_inlink)。然后可以用这个密钥进行加密和解密。但是应注意的是，这个密钥交换协议/算法只能用于密钥的交换，而不能进行消息的加密和解密。双方确定要用的密钥后，要使用上一节我们所学习到的各种加密解密算法来实现加解密消息。

## 小结

在本章中，我们简要介绍了通过公共网络进行通信的安全性。我们首先概述了证书以及如何使用证书来识别网络上的服务器；然后学习了如何在Rust中使用letsencrypt和openssl；接着，使用相同的技术保障Tokio应用程序的 安全；最后，简要介绍了使用DH方法进行密钥交换的情况。