ReL4 中异步网络协议栈设计与实现

摘　要

本文针对嵌入式微内核 Rel4 的网络协议栈在高并发场景下的性能瓶颈问题，提出一种基于 Rust 异步模型的协议栈优化方案。以轻量级 TCP/IP 协议栈 smoltcp 为研究对象，结合 Rel4 微内核系统特性，通过架构重构与异步化改造实现性能突破。

首先，通过层次化分析揭示 smoltcp 同步轮询机制存在的核心缺陷：全局 poll()函数导致收发逻辑耦合、高频空转引发 CPU 资源浪费的固有矛盾。基于此，提出了优化策略，分离收发路径为独立模块 poll_egress 与 poll_ingress，实现细粒度调度，并基于 async/await 的轻量级任务框架，利用 Waker 注册机制实现收发的异步执行。

实验采用 QEMU 虚拟化平台进行对比测试，结果表明：在本地使用 TUN/TAP 虚拟网络设备，手动构建数据包并进行连续传输对 poll 的异步调用测试下，改进后协议栈达到 2.7Gbps 吞吐量，较原同步版本有显著提升。

本研究创新性地将Rust异步技术适配到轻量化协议栈，在资源有限的嵌入式场景下通过主动的事件触发机制实现低运行时依赖的高效网络通信，为嵌入式微内核系统提供了可移植的高性能网络通信方案，具备在 ReL4 微内核环境中进一步集成和部署的潜力。

关键词：嵌入式系统；微内核；异步网络协议栈；Rust 语言

Design and Implementation of Network Protocol Stack based on Asynchronous Mechanism in Rel4

Abstract

This paper addresses the performance bottlenecks of the embedded microkernel Rel4's network protocol stack under high-concurrency scenarios and proposes an optimization scheme based on Rust's asynchronous model. Focusing on the lightweight TCP/IP protocol stack *smoltcp*, the study leverages the architectural features of the Rel4 microkernel to achieve performance improvements through structural refactoring and asynchronous transformation.

First, a hierarchical analysis reveals key drawbacks of *smoltcp*'s synchronous polling mechanism—specifically, the tight coupling of send and receive logic within a global poll() function and the inherent inefficiency caused by high-frequency idle polling that wastes CPU resources. In response, the paper introduces an optimization strategy that separates the send and receive paths into independent modules poll_egress and poll_ingress ,enabling fine-grained scheduling. It further implements asynchronous execution of these modules using a lightweight async/await task framework and a Waker registration mechanism.

Experiments conducted on the QEMU virtualization platform with TUN/TAP virtual network devices demonstrate that, under conditions of manually constructed and continuously transmitted packets, the asynchronous version of the stack achieves a throughput of 2.7 Gbps, significantly outperforming the original synchronous implementation.

This research innovatively adapts Rust's asynchronous programming model to a lightweight protocol stack. By introducing an event-driven execution mechanism with minimal runtime dependencies, it enables efficient network communication in resource-constrained embedded environments. The proposed approach offers a portable, high-

performance networking solution suitable for further integration and deployment within the Rel4 microkernel system.

Key Words: Embedded System; Microkernel; Network Protocol Stack based on Asynchronous Mechanism； Rust language

目　录

第 1 章 绪论

1.1 研究背景与意义

1.1.1 嵌入式系统发展背景

在半导体等技术的快速发展下，嵌入式设备在生活中逐渐普及，嵌入式系统作为其核心组成部分，被广泛应用于工业控制、物联网、汽车电子和医疗设备等领域，执行如控制流水线上各种电机和执行器，进行远程监控等重要任务[1-2]。从定义上，嵌入式系统是嵌入到对象体系中的专用计算机系统，其可以独立工作，也可以作为一个更大系统的一部分来完成特定的功能，其核心组件包括嵌入式硬件、实时操作系统、设备驱动、通信协议栈和嵌入式应用[3]。作为嵌入对象体系中的专用计算机系统，嵌入式系统面临着三个典型约束：

首先是资源约束性：嵌入式系统应用的设备通常只有很有限的内存、处理器频率以及能源供应，而这样的性能也就使得设计者需要在安全与执行效率间权衡，将资源利用率提升为设计的第一性原则。以网络协议栈为例，传统七层模型在嵌入式场景中就面临着性能问题，需要进行结构上的删减，本次使用的smoltcp协议栈就剔除了会话层、表示层等非必要层级，将核心功能聚焦于数据传输与路由，来提高代码的执行效率。

其次是专用性：其进一步强化了上述的删减优化方式，因为嵌入式系统是面向特定的设备设计的，其设计可以深度的修改通用模型。例如，在已知设备仅需UDP通信的场景中，协议栈可彻底移除TCP相关模块；若设备仅作为终端节点，甚至能简化路由算法。这种基于场景的特殊设计，使得嵌入式系统在有限资源下仍能提供高性能的服务。

最后是长生命周期：嵌入式系统以应用为中心，面向产品，需要较长的软件生命周期[4]。嵌入式设备通常需要在很少或者没有人员进行维护的情况下进行长时间的运行，如汽车电子有十余年的服役期要求，而这些要求需要系统具有充分的可靠性保证。

本次网络协议栈针对的系统Rel4就是一个面向嵌入式环境的微内核sel4的rust重写版本。在以上要求下，嵌入式系统的设计显得尤为重要，且嵌入式系统常是安全关键系统的一部分，系统的失效可能导致人员伤亡或重大的财产损失，常常还需要通过

动态冗余或使用不同设计和实现版本降低共同故障概率的方式来提高容错[5]。在资源约束，专用性，安全性等限制下，针对Rel4的网络协议栈也需要特殊设计，而不能套用传统的网络协议栈模式。

1.1.2 sel4 微内核现状

本次面向的Rel4是用Rust进行改写的sel4微内核，理解sel4就尤为重要，微内核的设计理念是只有操作系统的基本功能才在实际内核中实现。不那么重要的功能则建立在微内核之上，放入用户态，从而最小化内核功能，实现功能间的解耦。内核外部的所有系统组件均以服务器进程的形式实现，在用户模式下通过ipc调用使用微内核来相互发送消息，从而实现隔离。相比于宏内核直接相互进行调用的方式，这种形式更容易跟踪错误和意外。另外，模块化增加了灵活性，可以更加方便的对单模块进行开发和复用。在以此为设计理念的微内核中，sel4以其高安全性和高性能在被广泛使用。

sel4自2004年开始开发，于2014年实现开源，是L4微内核模型基础上设计出来的全球首个通过形式化验证的操作系统内核[6]。由于微内核是操作系统唯一在硬件特权模式下执行的部分，所以无法防止内核出现故障，每个错误都有可能对系统产生损伤，所以内核设计通过形式化验证十分重要。形式化验证研究的起点就是软件复杂度的提升会导致错误增多，这对任务关键型和安全关键型系统构成重大挑战，但为了构建可信系统，减少缺陷导致系统性失效的概率，即使证明复杂，sel4依然使用了高可信度的方法进行了形式化验证，其通过数学证明sel4内核不存在缓冲区溢出、空指针解引用、内存泄漏和未定义行为[7]。从结果上，这并不意味着它一定是安全的，还需要确保编译器，链接器等外部代码的安全性，但是从数学角度验证了其c代码的具体操作与内核应有的行为一致，使其有着极高的可靠性。

在另一方面，sel4在安全性的考虑上也进行了特殊的设计，其通过比传统linux更为严格的能力机制与保护域的结合使用实现了安全高效的访问控制，从而提供了细粒度的资源管理能力。能力机制要求线程保持最小权限原则，以降低外部攻击和意外错误的风险。实现了资源抽象与令牌化，也就是所有系统资源均被抽象为能力令牌。任何操作必须持有对应能力令牌才可以执行。而内核启动时通过显式授权链初始化能力空间。而保护域仅允许线程访问指定的内存区域，所有单元在内存中相互隔离，可以认为保护域是能力机制的边界，一个线程能获取的能力令牌不能超过保护域的

范围，从而实现了隔离化的管理[8]。系统将特定的内存隔离到单元中使用，实现一个与所有其他内存段完全虚拟隔离的内存段，处理器对这些内存进行虚拟寻址，然后分配给线程。除内核外，任何软件都不能访问直接内存映射，线程之间的唯一通信是通过内核的IPC进行的。这就为系统的所有组件提供了一个比单一用户空间更安全的访问环境。

1.1.3 rel4 微内核研究意义

使用Rust对sel4进行重写为rel4的原因是Rust作为一种专注于性能与可靠性的编程语言，其拥有优秀的内存管理机制和并发模型，在不牺牲低级语言性能的前提下，提供高级语言的安全性，是近年的热门系统编程语言[9]。传统的嵌入式内核依赖开发者手动管理内存，容易导致缓冲区溢出、悬垂指针和数据竞争等问题，而Rust与其他语言最显著的差异就是所有权机制：他强制编程者在给出的限制下编程，从而保证变量所有权的可靠安全性[10]。Rust 的所有权机制和借用检查器保证了在编译期发现通知这些问题。实验表明，在不使用unsafe语法的情况下，内存与并发漏洞发生率为0%，也就是可以充分避免内存风险[11]。相比C/C++，Rust实现的多线程服务器因为无锁实现，在吞吐量上提升35%[12]。除了安全性，在易用性角度，其无需依赖垃圾回收等机制和方便的编译器环境适合作为系统编程语言。如上面所述，嵌入式系统因为其运行特性需要较高的可靠性和性能要求，Rust语言正适合开发对可靠性和性能要求苛刻的系统组件。

1.2 国内外研究现状

1.2.1 基于 Rust 语言的 sel4 拓展

sel4作为内核不是一个完整的操作系统，有很大的可开发空间，而Rust作为近年的热门语言，有许多研究人员使用Rust针对sel4进行进一步研究。

sel4官方为开发人员出于对用户态组件的需求，提供了Rust语言在sel4微内核用户态开发的完整工具链包括sel4、sel4-sys等api的Rust绑定和运行时支持等，希望通过Rust的内存安全特性，来增强sel4用户态组件的可靠性，同时保留sel4内核的形式化验证优势。

在用户态组件的实现上，谷歌在2022年推出了使用Rust编写用户态组件的为嵌入式机器学习设备设计的开源操作系统KataOS，通过sel4解决内核安全，Rust解决用户

态安全的方式，去尝试解决智能设备，如摄像头、传感器中数据的安全性问题。尽管基于现有技术，KataOS仍有技术上的修改突破，如修改sel4内核以支持Rust编写的根服务器动态回收内存，同时保持安全性，使用上述sel4官方提供的sel4-sys来提供Rust绑定，优化机器学习的工作负载，对Rel4的开发有很大的借鉴意义。

1.2.2 适配宏内核的 smoltcp 协议栈开发研究

国外开发团队KAIST针对本次修改使用的协议栈smoltcp进行了二次开发为usnet_sockets，其核心为usnet_sockets网络库与底层的smoltcp协议栈，其设计目的是通过将TCP/IP协议栈移到内存安全的用户态进程中运行，来保证系统安全。其出发点是破坏者常使用远程恶意代码的注入来对网络服务进行破坏，而内存破坏类漏洞是导致远程恶意代码得以得到执行的根本原因。在内核中的TCP/IP协议栈高权限容易被攻击的情况下，协议栈移入应用程序，在用户态使用内存安全的编程语言情况下可以很好的防范内存破坏。这个目的类似微内核的概念，通过减少权限来提升安全性，这里设计者希望将这个协议栈应用在linux上，也就是宏内核中。

在研究成果上，KAIST使用一种利用应用内部网络协议栈去替换传统内核协议栈的技术，并在这个过程中保持了与标准库API的兼容性[13][12]。通过提供与标准库相同的socket类型，包括TcpStream和TcpListener，使得usnet_sockets的使用更加便捷。对于大多数基于标准socket开发的Rust应用而言，只需进行极小的改动便可接入该协议栈。

性能方面，usnet_sockets则仍有所欠缺。吞吐测试中，usnet_sockets未能实现线速传输。即使移除后台线程与usnetd，只用原生smoltcp实现，吞吐仍未达线速。对于这个情况，开发团队指出了smoltcp功能有限对usnet_socket性能的影响，如没有拥塞控制，ip分片等功能，导致特别是在高吞吐量和复杂网络交互场景中影响明显。

总结来说usnet_sockets提供了一个宏内核下安全、方便的协议栈方案，但是要达到与Linux协议栈等效的水平，还需更多的优化与功能补齐。

1.3 论文内容与结构安排

为了设计一个应用于rel4的异步网络协议栈，本文在选择smoltcp作为协议栈的基础上，首先对其设计架构以及收发模型进行了分析，之后针对其同步轮询模型进行了修改，对收发路径进行了分离，在面向嵌入式系统不能使用传统运行时的条件下，基于rust异步模型手动实现了对分离后发送和接收的唤醒机制，从而更好的发挥rel4在

实现异步后网络传输性能上的优势。设计完成的网络协议栈在qemu平台进行了性能测试并针对性能提升给出了相应分析。

论文总共分为五章。

（1）第一章为绪论，主要介绍本次协议栈开发的目标系统rel4的研究背景、意义与针对sel4微内核使用Rust开发的相关项目的研究进展。

（2）第二章为网络协议栈原理，介绍了TCP/IP网络协议栈的各层功能，以及TCP协议和IP协议的封装过程中协议头部意义，为后续针对smoltcp这个TCP/IP网络协议栈的修改做铺垫。

（3）第三章为Rust异步模型介绍，解释了Rust语言的async/await模型的相关执行流程以及优势，为后续对smoltcp协议栈的异步优化奠定理论基础

（4）第四章进行具体的实现细节介绍，首先进行smoltcp协议栈的五层架构功能分析，之后给出smoltcp收发模型的具体分析，引出针对性的异步优化方案，最后进行具体协议栈异步修改实现过程的解释。

（5）第五章阐述了性能测试结果，对优化前后协议栈的测试结果进行说明，并分析具体性能提升的原因。

第二章 网络协议栈原理介绍

2.1 什么是网络协议栈

网络协议栈是分层对网络io进行管理的通信框架，确保了数据可靠传输。与之对应的是网络设备，如路由器，是实际发送中涉及到的物理装置。可以认为网络协议栈就是对数据进行处理，使得其可以通过网络设备发送到目标节点，网络协议栈进行的处理保证了发送过程中出现的问题都有途径来解决，并在此过程中满足特定的要求，如传输性能或发送准确性，之后数据再通过网络协议栈反向处理解析出原本发送的数据。

2.2 TCP/IP 协议

TCP/IP协议是网络协议栈执行处理的理论依据之一，提供了一个各层协议如何进行处理的实际标准。

TCP/IP（Transmission Control Protocol/Internet Protocol）协议全称为传输控制协议/网际协议，其实际上由TCP，UDP，IP等多种协议组成，分别面向不同的应用场景。这些协议栈间不完全是并列关系，如上文对网络协议栈描述，网络协议栈是分层的架构，同个层级中的协议可以进行替换，高层与低层协助进行工作。

主流的理论模型是使用osi七层模型——物理层、数据链路层、网络层、传输层、会话层、表示层、应用层，以从上层到下层的方式依次处理发送的数据包。但是因为osi是理论模型，具体实现的话局限很多，工作复杂，且研发落后于实际情况。在这种情况下，TCP/IP模型成为了更常用实用模型，其针对osi协议进行了简化，共有四层，分别为应用层，传输层，网络层和数据链路层。TCP/IP协议的多种协议分布在这四个层级中，通过使用不同的模型可以满足不同的使用场景。

在上下层级的协议一起工作中，TCP/IP协议栈实现了功能上的叠加，构成一个完整的协议栈。以下会对其各层进行功能的介绍，各层次的关系与对应协议如图2-1所示。

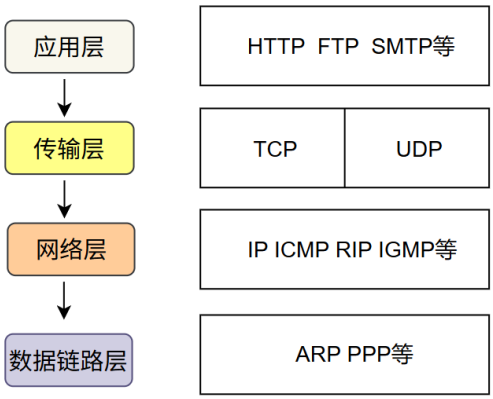

图2.1 TCP/IP协议栈层次模型

应用层：面向应用程序，为应用提供网络服务，实现应用间的连接管理。例如，HTTP协议用于网页访问，SMTP协议用于邮件传输。其主要负责通过协议来产生数据，从而指导传输层来完成端对端的连接，如使用dns协议获取ip和端口信息。

传输层：面向主机进程，确保进程间的可靠或高效通信，主要协议包括TCP和UDP。其中TCP协议主要用来实现可靠的通信，通过三次握手建立连接，并通过数据校验、重传等机制保证可靠性；而UDP负责低延迟的高效传输，如视频直播。其通过不建立连接的直接传输还有不进行重传直接丢弃传输失败的数据包的方式，在能容忍丢包的场景下获得了低延迟的优势。其中主机间的连接通过调用网络层服务实现。

网络层：面向两个主机，实现源主机和目标主机间的通信服务，包括逻辑寻址与路由等。IP协议使用IP地址去唯一标识一个网络接口，使数据包可以通过源IP和目标IP地址执行准确发送，其中路由器会根据这些地址来选择合适的发送路径。

数据链路层：面向两个直接相连的网络设备，实现相邻节点之间的传输，主要确保接收方所接收的帧是正确的，如以太网协议会通过差错控制确保物理链路上传输后的报文可靠。

综上所述，层次间是通过高层到低层的调用来实现一个完整的传输功能，通过同层内不同的协议来保证每一层的工作都能顺利执行并满足特定要求。

2.3 数据封装过程

上文对于TCP/IP各层级的功能进行了描述，通过分层的设计思路，各层功能明确、相互独立，网络层IP提供通用数据报服务，TCP在其上实现可靠数据流服务，UDP

提供简单的、不可靠的无连接服务，实现了功能全面而低耦合的网络服务[14]。而这些功能实现的重点就在于对数据的封装。封装指的是在发送的过程中，按照处理的顺序，每个层次都会在上层数据的基础上在头部和尾部根据其具体使用的协议添加信息，如图2.2所示。

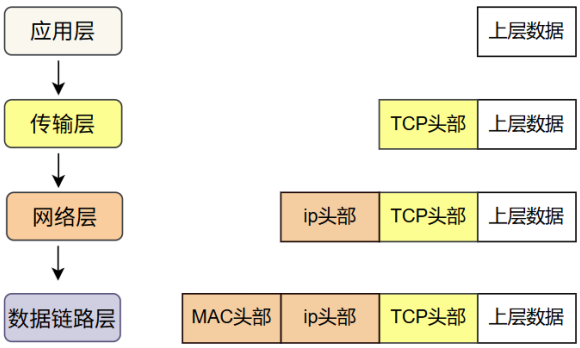

图2.2 TCP/IP协议栈的封装过程

在网络协议栈中，数据封装是个重要过程，从具体数据封装内容就可以区分不同协议，这个过程充分体现了每个层次实现的功能。本节会通过对TCP/IP协议中的核心协议TCP协议和IP协议的头部封装中的重要内容进行解释，介绍封装对协议实现的意义。

2.3.1 TCP 协议详解

图2.3展示了具体TCP头部封装的内容，包括源端口、目的端口、序列号、确认号、控制位、窗口大小等多个关键字段。这些字段共同构成了TCP协议实现可靠传输的基础。

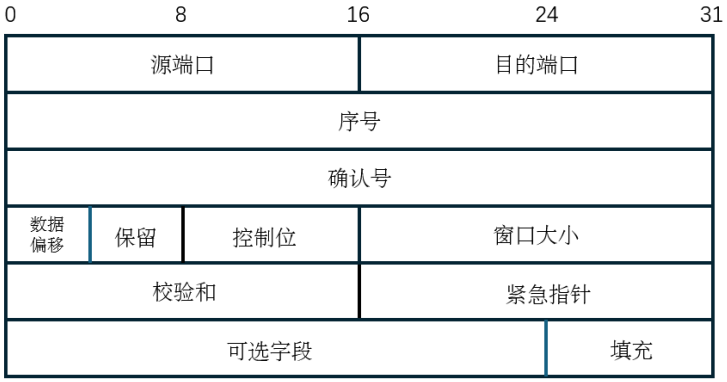

图2.3 TCP协议首部

TCP是面向进程的协议，而主机的端口唯一确定一个进程，端口的记录就实现了传输的收发方的确认，当数据到达主机时，操作系统会根据目的端口号将数据传输给相应应用，而源端口的保存则是为后文介绍的响应报文服务，让目的主机能向源主机传输回复报文。

序列号用来表示报文发送顺序的结构，连续对报文进行标号。在网络中，路由器使用的路由算法根据网络状态可能使数据包通过不同的路径执行发送，除此之外，为了确保送达而使用的重传机制也会影响数据包抵达接收方时的顺序。序列号使得接收方能够先缓存报文，之后进行排序组合得到正确报文。

确认号用来实现重传机制，TCP协议是追求可靠性的协议，为了确保接收方接收到发送的报文，要求接收方在接收后进行确认，如果长时间没有对某个报文确认，发送方会重新对没有进行确认的报文进行发送。此处确认号与序列号一起实现了累积确认机制。接收方只需要返回已接收的连续报文中最大的序列号作为确认号，就相当于已正确接收所有小于该序号的数据，从而减少了需要发送的报文，降低了协议栈性能开销。

控制位有有6位，包含URG,ACK,PSH,RST,SYN,FIN,主要用于实现三次握手的连接机制和滑动窗口协议。图2.3为三次握手连接的执行示意图。

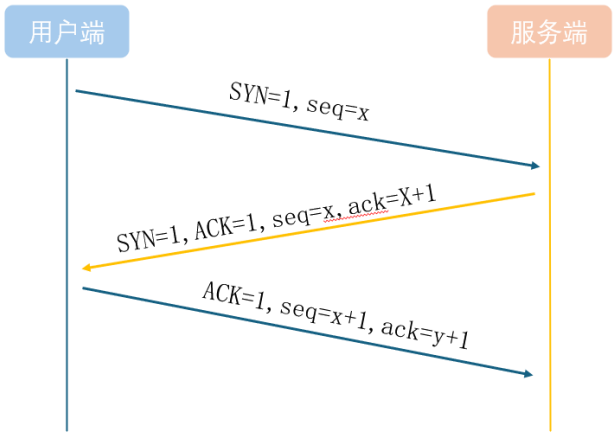

图2.4 三次握手流程示意图

TCP连接使用三次握手本质上是为了确保连接双方都成功进入连接状态状态，三次握手分为三个阶段：

（1）请求连接方选择一个随机初始序列号x，发送SYN=1,seq=x的报文给希望连接的主机，即服务端。等待对方回答。

（2）在服务端收到SYN报文后，会根据自己的初始序列号y，发送SYN=1,ACK=1,seq=y,ack=x+1的报文。表示自己接受到请求，可以进入连接状态。

（3）在二者都已经进入连接状态，需要有一个最终确认的阶段，网络条件不好的情况下，请求连接方不一定接收到了服务端发来的回复，所以收到服务端的回复后，还要发送ACK=1,seq=x+1,ack=y+1的确认报文，此时连接才正式建立，开始数据传输。

窗口大小用于告知发送方当前可用接收缓冲区大小，具体用于滑动窗口协议在不断变化的网络环境下提供合适发送的实现。滑动窗口协议通过动态窗口的调节来完成对于发送速度的调控，在接收方通过窗口大小告知其缓冲区大小后，发送方使用拥塞控制算法来计算拥塞窗口，通过对接收方窗口大小和拥塞窗口取小值来获得当前应该使用发送速率，实验证明TCP拥塞控制机制的重要理论基础AIMD是公平的、稳定的，并能实现对共享资源的均衡分配[15]。这证明了TCP拥塞算法的合理性，适合用于动态网络拥塞控制。

校验和是通过特定校验算法对封装的TCP头部的值进行计算，得到的一个值，用于确保发送中信息没有收到损坏，接收方通过使用相同的算法对接收到的报文TCP头部进行处理可以进行检验，如果检验失败，说明在传输过程中出现了干扰等问题导致数据与发送的报文产生较大差异，选择丢弃该报文。

2.3.2 IP 协议详解

IP协议作为网络层的协议，其头部封装主要是用于实现寻址和路由功能。图2.5展示了IP数据报的头部结构，其中包含多个关键字段，这些字段共同保障了数据包在网络中的正确传输。

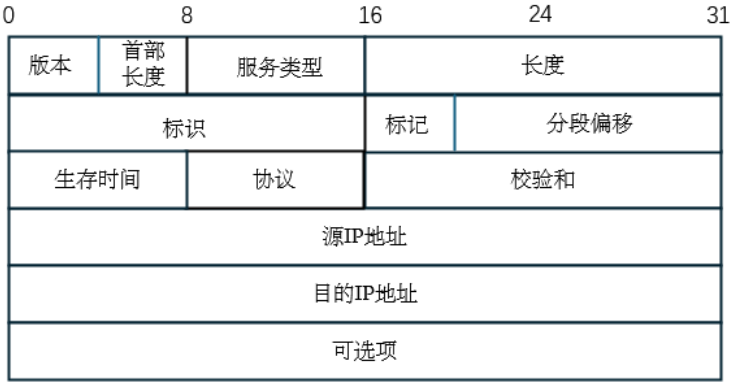

图2.5 IP协议首部

版本字段用于区分IP协议的版本号，目前主要使用IPv4和IPv6两个版本，其中IPv6用于应该IPv4地址数量即将耗尽的情况，增加了地址的位数。由于两个版本在地址长度和头部格式等方面存在差异，版本字段可以确保接收方使用正确的协议进行数据包的解析。

首部长度即IP头部的长度。由于IP头部包含可选字段，其长度可变，该字段确保接收方能准确定位数据部分的起始位置。

总长度表示整个IP数据报的长度，包括头部和数据部分。该字段的最大值为65535字节，但实际传输中受限于数据链路层的最大传输单元，通常需要进行分片处理。该字段确保接收方能正确识别数据报的边界。

标识位，标记位，片偏移三位共同用来进行IP分片功能的实现，具体来说当数据报长度超过网络最大传输单元时，需要对数据包进行拆分处理，分片所有分片保留相同的标识值，使接收方能够将属于同一数据报的分片重组。标识位通常由发送方维护一个计数器来生成。而标志字段一共由三位组成，控制数据报的分片行为。其中DF（Don't Fragment）位指示路由器不要对该数据报分片，若无法转发则丢弃并返回错误信息。而MF（More Fragments）位指示后续是否还有分片，最后一个分片将该位设置为0。第三位是保留位默认置为0。这些标志位对IP层的分片与重组过程进行管理。片偏移字段用来记录当前分片在原未分片的数据报中的位置，是以8字节为单位的相对位置。使接收方能够在分片到达顺序错乱时按正确顺序重组分片，还原完整的数据包。这一机制确保了大数据块在异构网络中的可靠传输。

生存时间占8位，用于防止数据报在网络中无限循环。每经过一个路由器，该值就会减1，当减至0时路由器会丢弃该数据报。该机制有效的限制了数据报在网络中的最长存活时间，确保即使出现环路，数据包也不会在网络中无限循环发送。同时依靠该字段可以实现对于路径的追踪：首先发送生存时间为1的数据包，之后逐次增加，每一跳会在将TTL减至0后获取到路由器返回的ICMP超时响应，通过不断记录返回消息的路由器的IP地址直至数据包到达目标主机，就可以完整的记录数据包从源到目标主机所经过的每一跳路由信息，从而可视化网络路径。

协议字段用于说明传输层使用的协议类型，如TCP为6，UDP为17。因为传输层协议的协议头结构差异较大，该字段使接收方的IP层处理后正确的使用对应的传输层协议进行处理。

源地址标识发送主机，目的地址标识接收主机，二者共同决定了数据报的路由路径。基于这两个字段，路由器可以根据路由表做出转发的路径规划，将数据报逐步发送到目标网络。

最后可选字段长度可变，提供了额外的控制功能，如记录路由、时间戳等。增强了IP协议的灵活性

综上所述，数据封装是TCP/IP协议栈实现分层处理的核心机制，其通过逐层添加协议头部信息，实现了从应用层到物理介质的完整通信过程。TCP协议通过端口号、序列号、控制位等字段确保了端到端的可靠传输，而IP协议则借助地址信息、分片控制、TTL等字段实现了网络层的寻址和路由功能。这种分层封装结构使各层的功能更加明确，还通过校验和、确认号等机制保障了数据传输的可靠性。可以说帧的封装对互联网通信框架进行了很好的实现。

2.4 本章小结

本章主要对TCP/IP协议进行了介绍。首先阐述了网络协议栈的定义，之后对TCP/IP协议进行了简单介绍并对其应用层，传输层，网络层，数据链路层的功能进行了分析，之后通过TCP协议，IP协议的帧格式分析,对TCP协议和IP协议主要实现的机制进行了说明。为后续针对TCP/IP协议的设计开发奠定了理论基础。

第三章 Rust 异步原理介绍

3.1 并发编程

本次选择的smoltcp在很多处理上为了适配嵌入式系统的特性从而进行了简化，设计上选择同步作为后续调用的方式，不断阻塞当前处理线程去判断是否需要执行发送接受的处理模式在性能上存在明显的性能瓶颈，尤其在网络的收发频率较低时会有较高的无用开销。同时因为处理导致的线程切换，也会提高系统的任务响应延迟，降低系统的实时性和并发能力，这对于要应用于强调并发，使用异步ipc的rel4的网络协议栈来说并不合适，可以通过并发的编程方式进行提升。其中Rust语言提供了多种实现方式。

3.2 并发模型对比

Rust提供了多线程和async/await来实现并发编程，而事件驱动是其他高级语言常用的模型，以下对三种模型进行比较。

1）多线程模型：最传统的并发模型，通过创建多个线程来同时执行任务，提高执行效率。每个线程由操作系统调度，共享进程的内存空间。

主要优点是简单直观，通过增加线程，使用多核CPU分别管理线程提升性能，对cpu有更好的利用，理解上很直观，而对于一个程序如果要使用多线程去提升性能，在使用上不需要改变原本的编程模型，编程思路相对简单。

在缺点上，首先对于设备性能有较高要求，每个线程需要独立的内存，在线程增多时内存消耗会显著增加，系统调控上下文切换的损耗大。其次线程间数据是共享使用的，读取修改数据都需要使用锁来保护，容易引发死锁等问题，增加了编程的复杂度。

2）事件驱动模型：通过非阻塞操作和回调函数实现并发，函数在编写为回调函数后，可以实现回调函数和后续函数并行。

事件驱动模型的优势是极佳的性能。事件驱动模型以单线程实现，在可以处理大量I/O事件的同时，避免了使用多线程时上下文切换开销和内存开销，适合高并发场景。此外，单线程避免了多线程的锁竞争问题，逻辑更加简单。

事件驱动的缺点是较高的风险。回调函数的大量使用会导致回调地狱，因为回调的执行顺序并不直观，在代码嵌套多的情况下很难理解当前执行关系，后期难以维护，如果出现bug难以处理。

3）async/await 模型

本次针对smoltcp修改使用的模型，其优点是高性能低开销。每个异步任务的内存开销极小，且Rust的异步的实现基本没有内部的性能损耗。

不过，Rust异步模型也有缺点。该模型相对复杂，且没有官方的运行时，需要使用的社区提供的运行时，增加学习成本。

在以上模型中，Rust实现了多线程和async/await模型，以async/await模型去替代事件驱动模型来保证代码的可靠性。在多线程模型和async/await模型之间进行比较，多线程模型更适用于需要并行计算且任务之间相互独立，阻塞较少的场景，可以充分的减少线程间同步和切换产生的性能开销。而async/await模型更适合io密集的模型，比如本次应用与网络协议栈上，如果使用多线程来实现，在进行io的时候会多次发生线程间切换，而网络协议栈会有大量时间处于不工作的状态，分配的线程又会处于闲置的状态，产生无用开销。综合各种情况，使用async/await模型有更好的可读性，并且避免了回调地狱，其无栈协程也做到了内存的节省，适合应用于本次对网络协议栈的异步优化中。

3.3 async/await 异步模型

3.3.1 Future trait

Rust的async/await模型是围绕Future trait构建的一套异步编程体系Future trait是Rust异步模型的基础也是核心，其本质是一个状态机，其具有两个状态Pending和Ready，并拥有一个成员函数poll。在实际运行过程中，Future的poll函数会被执行器多次调用。在调用时，Future会检查当前是否可以完成任务根据这个判断返回Ready或Pending。如果当前条件尚未满足，例如没有满足io的条件，poll会返回Pending并注册一个Waker到当前任务中，其允许执行器在未来的某个时间点唤醒这个任务，使其重新获得执行机会，继而再次调用poll函数进行状态的转换，如果此时任务已满足完成条件，就会执行并在结束后返回一个Ready元组结构体，在其中存储完成的结构数据。Future是一个trait，这意味着编程者可以针对不同的自定义的的数据结构实现这个Future trait，而不是Rust中固定的结构，在面向不同需求的时候只需要定义不同的

poll函数就可以实现对于各种任务的异步实现。除此之外Rust还有Pin语法的设计来控制Future的内存地址，保证数据的值在内存中不会移动导致引用错误，继而保证了引用结构的安全性问题，这也符合Rust对于安全性的高要求。

3.3.2 Waker 与执行器

正如上文所述，如果一个Future在第一次调用poll时返回了Pending，就意味着其当前不具备完成任务的条件，需要依靠Waker来实现触发后续的poll。Waker是一个被包装为arc的结构体，其作为智能指针能够在多线程之间安全传递，也就允许后台事件在任意线程中实现对于原Future的触发。当Waker注册所在的任务完成后会调用wake方法，执行器接收到Waker触发的通知后将该Waker对应的Future重新加入任务队列，并在合适的时间再次调用其poll函数。

显然上述的所有poll如果需要通过编程者主动控制过于复杂，也会影响系统的高效运行，不符合async/await低开销的特性，上文反复提及了执行器这一结构，事实上除了首次poll调用，其他的操作都是由执行器负责的，其内部会维护一个任务队列来管理Future的生命周期，并在Waker触发后对Future进行poll操作来实现异步操作。相比之下，普通的异步实现是对每个唤醒任务的控制都使用线程控制，不断的检查是否完成了相应的要求，之后返回执行，这种方法实现的异步线程上下文切换频繁，会消耗大量的资源，性能上没有明显提升。在Rust的异步运行时中，Rust所使用的Waker并不是独立使用线程进行的监控，且本质是使用了操作系统所提供的多路复用机制，来实现一个线程同时阻塞等待多个事件，并且可以实现精准的触发。在面向网络协议栈的场景下，往往有多个socket连接同时需要进行维护，此时为了每个socket的io操作的异步执行，需要其注册各自的Waker，这里运行时可以保证在同一个结构体的函数中注册的Waker进行区分处理，从而一对一的进行唤醒，保证了异步触发的准确性。通过使用Waker和执行器这种设计实现了低成本的唤醒。

3.3.3 异步运行时

在所有的poll调用和任务队列的管理都不是由开发者手动控制完成的情况下，对于以上复杂操作所使用的执行器的封装就需要依赖运行时的实现，其由Rust社区提供，负责实现整个异步系统的执行环境。

运行时的功能包括初始化执行器，维护任务队列与线程池，用于调度和驱动 Future 的执行。其利用底层操作系统的多路复用机制来监听事件，实现对Waker的触

发，确保事件发生后正确唤醒对应的任务。例如目前使用最广泛的异步运行时库Tokio，其内部封装了线程池、任务调度器等多个子系统并提供了多个调度策略。在前文提到的Waker注册和唤醒过程实际上是运行时的职责核心之一。当Future在poll中返回Pending并注册Waker后，运行时会把这个Waker绑定到某个io事件上。当该事件发生，运行时将负责调用wake，并将对应任务重新压入待执行队列，等待执行器再次调用其poll方法推进任务状态。这并不说明在没有运行时的环境中，Rust的async/await语法就不可用，事实上本次使用的协议栈smoltcp因为面向嵌入式系统，并没有运行时环境，就需要用户手动实现完整的poll控制流程，这依然是一条可选途径。

综合来说，运行时的优势就在于其将Future实现中的重要的Waker唤醒机制从需要使用线程来单独控制的高开销场景转为了单线程就可以处理，大大减少了线程创建、销毁、上下文切换等操作的开销，从而提升了资源利用效率与系统的响应能力。async/await正是因为有了运行时，才真正成为一套高性能、低开销、适合现代系统需求的并发方案。

3.4 本章小节

本章主要对rust异步原理进行了介绍。首先简要分析smoltcp的实现得出使用并发模型的重要性，之后对常用的并发模型进行了介绍和比较，说明本次使用async/await模型的原因，之后针对async/await异步模型从future trait，waker，执行器和运行时几个方面进行了详细的介绍，为后续针对smoltcp收发的异步修改奠定理论基础。

第四章 smoltcp 的异步设计与实现

4.1 smoltcp 的架构分析

smoltcp协议栈是一个轻量级、使用Rust实现的嵌入式TCP/IP协议栈，主要面向嵌入式系统。作为一个网络协议栈，因为其面向嵌入式设备，可使用的资源有限，所以并不是以常见的osi七层模型进行的模块区分，并且相比于osi模型中应用层，表示层逐层向下传递的层次关系，smoltcp的模块也不是严格的高层低层封装关系，而是分为了socket，iface，storage，wire和phy五层，分别负责不同的功能。

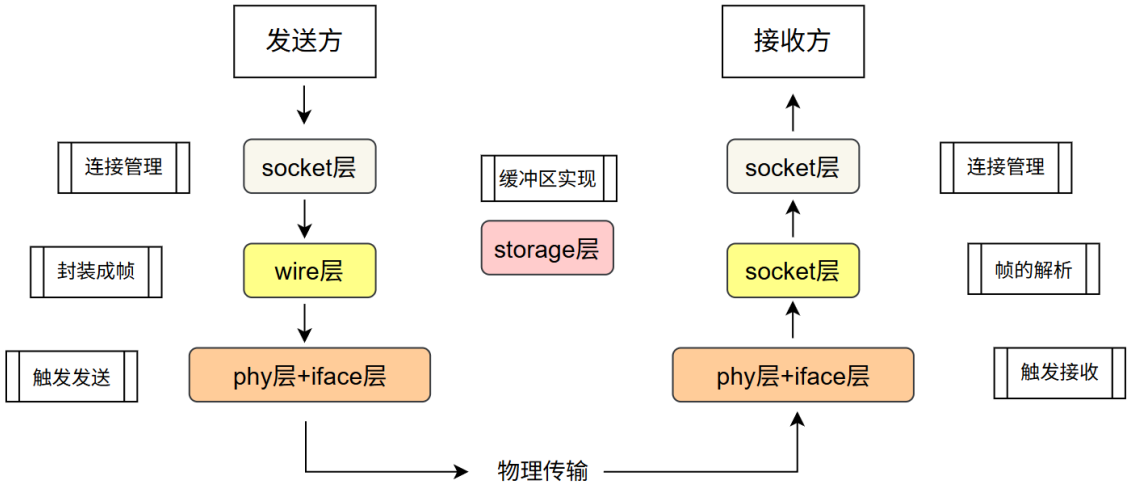

图4.1 smoltcp分层架构

首先Socket层是协议栈与应用的交互模块，负责多种协议的抽象以及连接管理。该层实现了TCP、UDP、ICMP、DNS等协议的Socket抽象，但其设计理念与传统协议栈有所区别，在socket层中并不直接参与数据包的封装与传输过程，而是负责进行高层次的传输管理。具体针对本次主要进行修改TCP协议为例，Socket层为其实现了三次握手、流量控制、超时重传等协议的核心机制，而具体的数据包构建工作是交给其他层去执行。这种设计使得协议的处理逻辑与数据传输封装路径实现解耦，为协议扩展提供了灵活性。值得一提的是，虽然smoltcp的运行逻辑是同步，但是其中提供了async编译选项，也就是与异步相关的函数，包括对于Waker的简单封装，还有包括Waker的注册函数为后续的开发者提供了便捷的异步修改途径

相对与上面不负责封装管理高层细节的socket层，wire层就是协议栈负责各层封装和后续接受处理的协议编解码引擎，其设计遵循"可构建即可解析"的核心原则。该层包含了以太网帧、IP数据包、TCP/UDP报文等各层协议的构建器与解析器。例如在构建IP数据包时，wire层会自动计算校验和并填充必要字段，确保生成的报文完全符合协议规范，其中也可以通过函数控制是否进行校验和的添加，来进行性能上的管控，面向低传输准确性要求的情况可以不进行校验和的添加和检验从而减少在这部分消耗的处理器资源；在解析过程中，wire层则会进行严格的格式验证，丢弃不符合规范的数据包。这种双向一致性保障机制，使得协议栈具备自我验证能力，从根本上避免了报文本身错误导致的解析错误。而wire层通过零拷贝技术直接操作原始字节流，也提升了处理效率。

上面两层完成了封装和发送管理后，需要的就是底层与设备的对接，iface层在负责协调Socket层与wire层的交互的基础上，会去调用设备层的函数来进行实际的收发管控。该层通过同步轮询机制驱动整个协议栈的运转：在发送的实现上，iface层会定期收集各Socket的发送缓冲区数据，调用wire层进行协议封装，最终将完整的数据帧提交至物理设备；在接收实现上，则从物理设备获取原始数据帧，经wire层解析后分发至对应Socket的接收缓冲区，交给socket层和应用进行后续的数据处理。这种显式的poll驱动模型，与常见操作系统内核的中断驱动架构形成鲜明对比，虽在实时性方面存在理论劣势，却显著降低了上下文切换开销。

内存管理系统是操作系统的重要组成部分，负责管理计算机内存资源的分配、回收和使用。在内存有限的情况下，其在优化系统性能方面发挥关键作用，更需要确保内存的高效使用，在动态分配的情况下，空闲内存会被划分成较小的块，根据进行分配。系统会跟踪哪些内存块正在使用，哪些是可用的，确保进程拥有执行任务所需的内存资源。当某个进程不再需要某块内存时，内存管理系统会回收该内存块，使其可以再次分配。显然这个过程需要内存的分配和回收过程被严格管理，对于处理器性能有较高要求，不适合嵌入式系统的环境。storage层通过预分配的静态缓冲区管理机制，实现了对内存使用的确定可控。该层提供了环形缓冲区、分片缓冲区等多种数据结构，支持零拷贝的数据传递。例如在TCP传输过程中，应用层数据直接写入Socket的发送环形缓冲区，iface层在poll周期内批量获取这些数据进行协议封装，整个过程避免了数据复制带来的性能损耗。这种设计可以有效的避免内存碎片化问题，适合长期运行的嵌入式系统。

最后phy层作为硬件抽象层，定义了统一的网络设备接口。该层支持环回接口、TUN/TAP虚拟设备、原始套接字等多种物理/虚拟设备的接入，通过trait抽象屏蔽具体设备的差异性。开发者可通过实现Device特质，其中对与设备进行了统一的定义，对不同的网络传输介质进行了传输函数的要求。对设备的传输能力也定义了四个关键属性：介质类型、最大传输单元、最大突发尺寸和校验和能力。对于之后实现的不同设备都需要填充上述属性为后续操作服务。如校验能力使用标志位定义，可以进行启用和关闭，体现了设计上的灵活性。在发送和接收上的主要设计点在于接收令牌（rxtoken）和发送令牌（txtoken）。其要求发送和接受时首先获取令牌，也就是获得对于缓冲区的处理权，在interface层完成处理后，通过consume方法进行实际的发送接收操作，分步的进行保证了发送数据的正确处理，在完成构建后进行发送，避免因为中断等情况导致发送被临时打断，发送无效数据包。这种设计保证了报文构造的可靠性，还实现了高效的批量发送机制。

4.2 smoltcp 收发处理分析

因为smoltcp中是没有给出高层的封装的，也就是并没有一个函数直接封装了完整的发送和接收操作，其需要使用者去进行编写，也就提供了进一步进行逻辑修改的可能性，适合对其进行修改。由上述smoltcp各层功能以及相应函数的分析后，可以结合得到他的运行逻辑。以下将从其发送的函数逻辑和具体的发送逻辑进行解释

首先是函数逻辑，如前文所述，socket层负责协议的高层处理，并不进行封装和解析操作，所以其需要与封装解析的wire层进行沟通，而其与wire层进行沟通的途径就是通过其缓冲区，还有send和recv函数，当应用产生需要发送的数据，socket层会在基本处理后存储在socket的发送缓冲区中，其中socket中有设计can_send等函数来管理每个连接的send状态，方便后续轮询操作，recv函数同理，有对应的缓冲区进行数据存储，调用recv函数可以进行后续处理。

对于所有类型的设备，其都会维护一个socketset，socketset负责保存当前网络设备所有的socket连接，在进行某个socket存储好发送数据后，会通过interface层调用poll来进行处理，其会遍历socketset中所有的连接，通过egress_permitted筛选出所有可以进行发送socket连接，然后获取到其缓冲区的处理权执行封装，根据当前传输介质和协议的类型进行区分处理，对数据进行标志位设置等封装操作。物理层面的发送就涉

及上述phy层的rxtoken获取和消费，首先申请发送权限，获得后通过consume处理，将封装后的数据包拷贝到驱动指定的发送缓冲区，并触发硬件进行实际的硬件发送。

除此之外，poll函数可以分为两个功能，除了发送之外，其还同时进行接收操作。在完成发送后，获取设备的rxtoken，即接收令牌，通过消费获取到具体数据包的处理权限，将其取出并进行数据报的解析，通过源IP、目的IP、源端口、目的端口和使用的协议从socketset中查找匹配的socket连接，将数据存储到相应的socket的接受缓冲区中，之后实际处理通过socket调用recv函数进行，整体的收发操作示意图如下图表示。

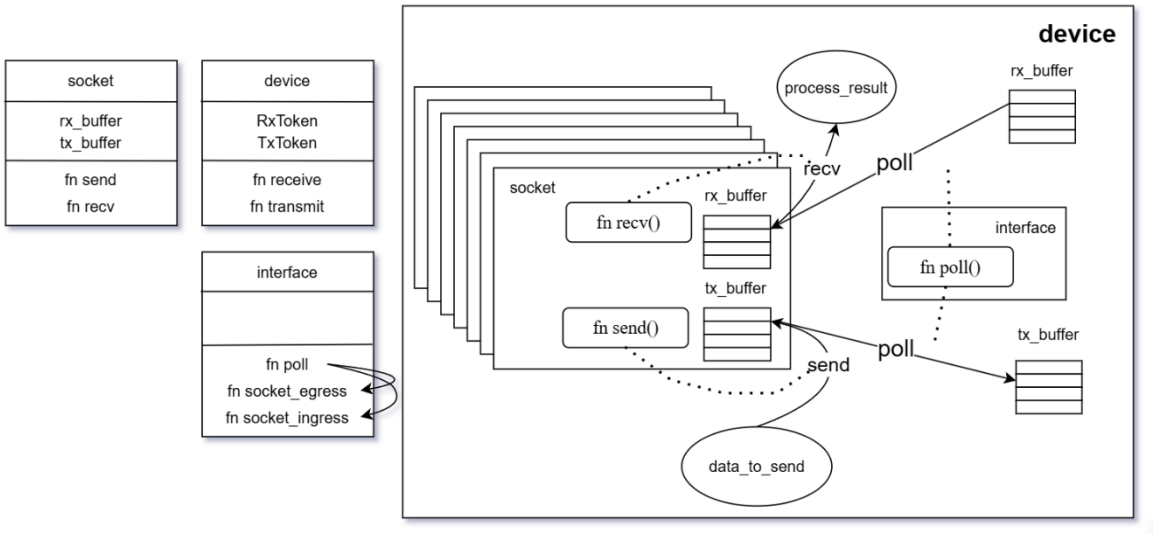

图4.2 smoltcp收发流程示意图

可以通过具体的收发测试程序对这个流程进行更细化的解释，在协议栈启动并完成网络接口的配置后。会根据物理设备类型，加载相应的驱动模块并进行初始化。之后为了在执行持续的发送中实现低延迟的相应，线程需要保持轮询状态，不断遍历socketset判断所有连接的状态，对于处于可发送状态的socket，通过send将数据存储到其缓冲区中。

因为发送和接受是通过poll函数进行的，应用会希望能在存储好数据后尽快的进行poll的调用，但是直接在send函数中调用poll函数并不合理，因为poll函数和send函数的同步执行可能导致阻塞等待网络层资源，smoltcp对此的解决方案是以轮询调用poll，通过poll_at函数来动态的计算休眠的时长管理轮询的时间，poll_at所给出的时

间是一个非强制执行的时间，以上的执行流程保证了smoltcp的低耦合，无阻塞调用，但也因此提升了发送和接收的延迟，在这方面存在优化空间。

4.3 Embassy-net 和 axnet 的改进

完成smoltcp的上述分析后，可以明显发现同步方式调用的劣势，一次循环所间隔的时间通过poll_at进行判断，其通过计算得到poll的适合调用时间，但socket将数据存储到缓冲区的send函数是否能执行的can_send等函数也在同一个循环中进行判断，导致一次判断失败后下一次判断需要等待自然的轮询而没有算法对这个过程进行时间上的缩减优化，导致应用发送信息的滞后。

embassy-net和starry-os中的网络模块axnet二者都是在smoltcp的基础上进行了封装，对socket中的send函数进行了优化，axnet在实现中使用了WouldBlock机制来协调非阻塞I/O，这里它的设计目标并非直接实现异步I/O，而是通过与操作系统的调度协作，在用户态模拟类似异步的效果，从而进行性能提升，减少无效判断，任务在完成的情况下则返回值，未完成则返回WouldBlock。然后分为两种模式，在non-blocking模式下，子任务未完成，父任务立即返回WouldBlock；而在blocking模式下，子任务未完成，父任务轮询到其完成为止。轮询中交替检查网络包状态和任务状态。这样的实现相比与异步会因为重试多次上下文切换，还是会有性能消耗。

而embassy-net中进行了进一步的提升，使用了异步的方式进行函数调用，基于异步运行时，将smoltcp的同步轮询接口改造为异步任务驱动模型。具体就是将send和recv定义为async函数，借用smoltcp中本身就有实现的Waker机制，在准备好数据的情况下调用Waker唤醒async函数，来实现异步执行，减少了空闲时的CPU占用，允许其他任务在等待网络事件时执行，提升系统整体效率。两种优化方式为我的后续性能优化改进提供了借鉴思路。

4.4 针对 smoltcp 的代码修改

基于上文内容，在socket层的send和recv函数已经通过外部封装实现了异步的情况下，对于使用轮询来触发实际的发送接收的poll函数进行异步的修改能更好的实现整个系统的异步调用模式，提升协议栈的性能。

进行工作的首要问题是，对于smoltcp来说，并不具备使用tokio运行时的条件。smoltcp是一个不依赖操作系统，面向裸机和嵌入式环境的网络协议栈，其针对的嵌

入式设备是面向特定的应用场景而存在，功能特化，通常在no_std的环境下完成系统的构建，此类系统不具备完整的操作系统调度能力，并缺少标准库的支持。而Rust常用的Tokio等异步运行时依赖与操作系统提供的异步I/O支持及库函数去工作，这些功能在嵌入式系统上基本不可用。由此需要精简化的运行时来替代常用运行时的功能。

从后果上，因为异步运行时主要负责的是执行器的实现，所以无运行时情况下Waker机制的自动实现存在问题，无法实现Future自动的任务调度，其次，在I/O就绪的时候无法进行通知，因为没有调用底层的函数来监控文件描述符，来获取到接收报文的信息。这将导致修改为异步的接收功能也只能以忙等待的形式进行。

以上问题并非没有解决方案，社区有提供为嵌入式系统使用的轻量化运行时embassy，embassy使用静态任务调度和无分配的方式实现了轻量级异步执行，其任务模型基于合作式调度，并通过执行器embassy_executor来管理任务调度，适合在裸机的环境中运行。

尽管embassy提供了基本的Future执行能力，但在具体的I/O驱动集成方面，仍需开发者手动补充部分运行时所需的功能，特别是在Waker的注册与唤醒机制方面。smoltcp 在其官方提供的Socket接口中，尽管整体模型为轮询式，但也预留了Waker注册点和基本封装，这一设计为在无运行时环境中引入异步通信机制提供了基础。

本论文在此基础上，结合embassy的异步执行能力，构建了完整的Waker注册与唤醒流程。在具体改造中，首先对poll函数进行修改，其同时进行发送和接收的执行模式很大程度上降低了接收的实时性，此处进行了分离为发送和接受两个函数，poll_egress和poll_ingress, 分别负责发送和接收的触发，

对于Waker的注册，发送和接收的封装略有不同。对于发送路径，协议栈工作的逻辑是在socket层提交待发送数据后，通过smoltcp的poll函数驱动物理设备发送。

其触发Waker的函数是协议栈中的函数，即socket中的send函数，通过在socket成功发送后返回前进行Waker的触发，从而实现poll_egress的异步调用，伪代码如下：

表4-1异步发送算法

算法1：发送future
输入：网络设备接口 device，socket集合 sockets
1: 获取当前时间now;
2: 调用poll_egress()函数发送数据:
3: **if** result有数据返回或状态改变 **Then**
4: 返回Poll::Ready(result);
5: **else**
6: **for** socket in sockets **do**
7: 注册写入waker(s_waker);
8: **end for**
9:**end if**
10: 注册walker期间，可能已经触发状态变化，再次调用poll_egress()函数：result = poll_egress(now, device, sockets)
11:**if** 有数据被发送 **Then**:
12: 返回Poll::Ready(result)；
13:**end if**
14:**if** 没有发送机会 **Then**
15: 返回 Poll::Pending
16:**end if**

在socket结构体中添加发送使用的s_waker，并完成注册函数，而async_poll_egress首先需要对Future trait进行实现，通过poll_fn封装成 Future，其核心机制可分为三个阶段：

1.Waker注册前的处理：调用poll_egress检查是否存在待处理的数据包发送任务，若此时已经具备发送条件，则执行发送，之后所有的socket连接都处于没有待发送数据的状态。

2.注册Waker：在当前所有socket均不可发送下，遍历socket使各自注册Waker将任务的Waker注册到每个socket内部的s_waker字段中，用于下一次发送时的唤醒。

3.再次检查状态：防止在注册 Waker的时候发生状态变化，需要重新调用同步的poll_egress以确保所有socket的当前状态仍处于无发送请求的状态。

以上结构严格模拟了运行时的自动化行为，在第一次 Poll 发现任务尚未就绪时，将Waker自动注册至Reactor，当底层IO状态变化后由内核发出通知唤醒任务。而本结构通过用户空间逻辑完成了等价流程，适应运行时缺失场景。

在接收中，机制与之类似，但必须额外关注底层驱动设备状态的变化。可执行发送条件的判断过程比较直白，在存储好数据后直接对Waker进行触发即可，相对于发送来说，接收的Waker触发更复杂。具体来说Embassy因为其是面向裸机和嵌入式环境的异步运行时，它只负责管理任务的调度和Waker的注册与唤醒逻辑，而并没有内置任何对系统的文件描述符或I/O事件的监测功能。这种设计使得Embassy能够在没有操作系统支持的环境下运行，但也因此无法自动进行异步接收触发。对于tokio这类在操作系统上运行的运行时，其内部会维护一个隐式的Reactor，可以利用系统的多路复用直接检测外设数据的接收，一旦检测到文件描述符可读或可写，就调用相应的Waker进行唤醒。Embassy没有类似的后台Reactor线程，在Embassy只管理异步任务调度的情况下，当外设接收到数据后，会要求手动在驱动程序中调用对应任务的Waker.wake()，通知异步任务继续执行。

本次对smoltcp的测试主要在Linux环境使用虚拟的Tun/Tap设备。Tun/Tap设备表现为一个普通文件描述符。Linux内核可以通过epoll等系统调用来检测文件描述符的状态，判断是否有数据可读或可写。然而为了更贴合smoltcp之后的应用场景，选择不使用tokio，在使用embassy作为运行时的情况下，不能使用不属于裸机环境的由系统内核提供的调用。

因此，在使用Embassy管理Tun/Tap设备异步接收时，异步函数依然无法实现自动的触发，因为使用虚拟设备，通过内核对协议栈进行发送接收模拟，无法修改驱动代码，所以本次对于接收的修改上针对是否有数据包可接收，采用轮询文件描述符去判断，替代实际中驱动进行的函数，并在发现有数据包可接收后，触发Waker。

表4-2 接收检测算法

算法2：接收waker触发算法
输入：套接字文件描述符fd，接收唤醒器rx_waker
1:获取虚拟设备文件描述符fd
2:创建轮询结构体数组fds，包含文件描述符fd和监听事件POLLIN
3:调用poll系统调用获取返回就绪描述符数量n
4. **if** n不为0 **Then**
5: **for** fd[i] in fds do
6. 获取fd[i]的返回事件
7. **if** 监听到数据到达 **Then**
8: 调用rx_waker.wake()，触发waker
9: **end if**
10: **end for**
11:**end if**

接收函数async_poll_ingress在设计上与上文的async_poll_engress类似，而异步函数负责通过对于文件描述符进行检测来判断设备是否有数据可读，若未能立即获得数据，则因为async_poll_ingress函数返回pending完成对Waker的注册。该函数在检测到接收之后会进行waker的触发，从而对async_poll_ingress函数再次进行poll的调用，实现接收的异步。

综上所述，尽管嵌入式系统缺乏通用运行时的调度支持，通过 Embassy 提供的poll_fn和Waker触发机制，结合WakerRegistration对于Waker封装，实现了在嵌入式的条件下，将smoltcp修改为了异步触发发送和接收的网络协议栈。并且该方法具备可移植性，在实际使用情况下，对smoltcp进行适配后，通过在网卡驱动代码中添加Waker的触发，即可实现收发的异步。

4.5 本章小节

本章详细介绍了smoltcp网络协议栈的框架以及设计修改方案。首先，给出了smoltcp五层的整体设计框架，并分析了各个模块的功能。然后具体对smoltcp进行了收发处理逻辑方面的分析。针对smoltcp以同步方式调用poll函数效率不高的情况，给

出了分离收发并使用async/await异步模型对其进行了修改。在没有传统运行时支持的情况下以手动注册waker并设计触发的形式实现了异步的发送和接受。

第五章 性能测试与结果分析

5.1 性能测试

针对以上修改后的异步网络协议栈，进行了协议栈传输性能的测试。在评估性能时，首先进行理论上限速率测试，即在测试代码中手动构建数据包，绕过了IP、TCP等协议的封装处理过程，直接构造链路层数据帧并通过虚拟网络适配器进行发包测试，目的是测得物理链路的最高吞吐能力，从而得到与实际封装发送的比较，更好的进行协议栈处理时间的分析。

实际测试中，首先面向对smoltcp使用WouldBlock机制来协调非阻塞I/O的axnet模块进行基础测试，使用axnet而不是smoltcp的原因是其完成了完整的系统搭建，提供了如iperf等工具来让测试结果更清晰，适合进行对smoltcp的处理性能进行比较分析。

```
[ 71.189351 0 axnet::smoltcp_impl::bench:36] Transmit: 0.131GBytes, Bandwidth: 1.054Gbits/sec.
[ 72.189361 0 axnet::smoltcp_impl::bench:36] Transmit: 0.131GBytes, Bandwidth: 1.052Gbits/sec.
[ 73.189362 0 axnet::smoltcp_impl::bench:36] Transmit: 0.132GBytes, Bandwidth: 1.059Gbits/sec.
[ 74.189368 0 axnet::smoltcp_impl::bench:36] Transmit: 0.134GBytes, Bandwidth: 1.078Gbits/sec.
[ 74.296655 0 axruntime:192] main task exited: exit_code=0
```

图5.1 性能测试接口层结果

使用bwbench作为运行程序启动测试，该测试程序会持续向设备发包并记录总传输字节数与耗时。其中设置的传输总量为10GB，实验结果上总用时为约为73秒，换算后速率约为140MB/s，该阶段测试的目标是记录底层链路在理想条件下的吞吐，作为后续协议处理延迟的参考基准。

为了对比协议栈处理带来的性能开销，进一步引入了完整的传输层测试。启动starry-os提供的标准iperf工具，使用smoltcp作为发送端，分别对TCP和UDP协议的发送进行评估，图5.2是结果示意图。

```
-----------------------------------------------------------
Server listening on 5555
-----------------------------------------------------------
Accepted connection from 10.0.2.2, port 42730
[  5] local 10.0.2.15 port 5555 connected to 10.0.2.2 port 42742
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00   sec  61.8 MBytes   518 Mbits/sec
[  5]   1.00-2.00   sec  61.2 MBytes   514 Mbits/sec
[  5]   2.00-3.00   sec  62.0 MBytes   520 Mbits/sec
[  5]   3.00-4.00   sec  62.0 MBytes   520 Mbits/sec
[  5]   4.00-5.00   sec  61.2 MBytes   513 Mbits/sec
[  5]   5.00-5.99   sec  60.7 MBytes   515 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-5.99   sec   369 MBytes   517 Mbits/sec                  sender
[  5]   0.00-5.99   sec  0.00 Bytes  0.00 bits/sec                  receiver
```

图5.2 性能测试协议栈处理发送结果

这里iperf是一个经典的网络带宽测试工具，它的运行模式为客户端和服务端两种皆可，在命令中运行的是apps/c/iperf，没有特别的进行指定，它的行为取决于后续进行的操作。本次测试中选择使用其作为发送端，并在另一个终端中使用iperf3 -c 127.0.0.1 -p 5555 -R设置为接受端作为服务器。该测试路径涉及完整的smoltcp协议栈处理，根据实际输出，发送速率降至约60MB/s左右。大致为接口层测试性能的40%左右，说明协议栈处理引入了显著开销。

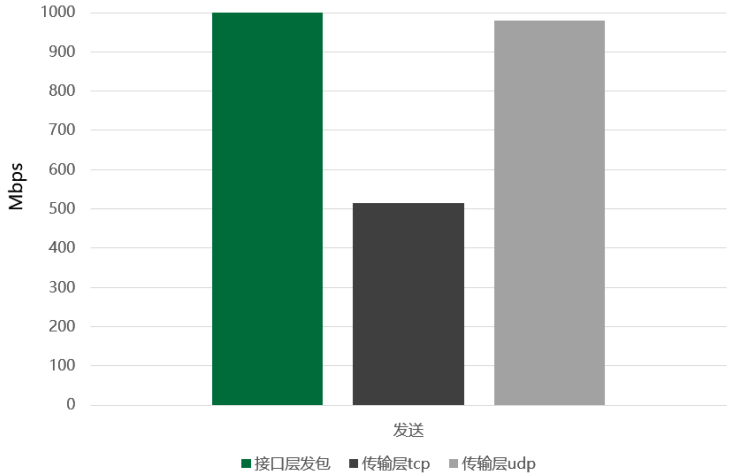

图5.3 性能测试数据比较

通过进一步对UDP发送路径的测试可以发现，在不建立连接的前提下，其性能更接近接口层测试的数据，表明UDP模型下协议栈干预较少。而相比之下，TCP的连接建立、状态维护、ACK确认等确保可靠的机制引入了较大处理开销，影响了整体吞吐性能。尤其在原始smoltcp中采用同步轮询的模型，使得CPU时间大量消耗于poll函数的空轮询上。

之后对修改后的版本进行测试，依然使用手动构建数据包的形式进行测试，这样可以掉过本身处理的干扰，仅仅针对smoltcp的发送poll机制的触发进行测试。

```
wyatt@wyatt:~/workspace/smoltcp-test/smoltcp$ cargo run -q --release --example benchmark -- --tap tap0 reader
throughput: 2.773 Gbps
wyatt@wyatt:~/workspace/smoltcp-test/smoltcp$ cargo run -q --release --example benchmark -- --tap tap0 writer
throughput: 6.838 Gbps
```

图5.4 修改后协议栈性能测试结果

图5.4为测试结果，对poll发送和接收分离并改造为异步后，可以有效的降低同步模式下的忙等开销，并提高接收的相应速度。实验结果显示，在接收和发送的吞吐量上均观察到性能的较高提升，发送侧能够更快地触发poll_egress逻辑，缩短数据写入后的等待时间，而接受侧不再与发送一起处理后，显著提高了处理效率。

5.2 结果分析

从对比上可以发现smoltcp更改前后性能有较大差异，初始smoltcp依赖在循环中不断调用poll函数来触发协议栈的发送与接收操作。这种同步模型的设计虽然实现简单、适用于嵌入式场景，但在面对高频次的密集IO的应用场景下存在着的性能瓶颈，其中关键在于poll_at函数对于poll的管理。

因为协议栈收发函数的调用途径只有poll函数，所有的除了基本消息收发之外的用于辅助协议栈工作的事件处理，例如超时重传，dns解析的发送全都集中在poll中实现，然而poll函数的逻辑是遍历所有的socket去执行收发操作，所以即使poll的调用时间正确，在处理上依然有大量的消耗。

其次，poll_at是控制poll循环调用频率的计算函数，但是其只有在存在可预测的发送情况时才能给出可靠的轮询时间，例如超时重传的计时器时间。而在发送零散的情况下，其并没有主动检测得知发送的机会，所以在默认模型中，poll_at没有检测到socket发送需求情况下，smoltcp的poll函数需要被周期性调用，不论是否有实际数据到达或需要发送，且为了维持实时性，系统必须以较高频率进行调用，否则可能会错过数据包，或者因为处理滞后导致发送方不能在有限的时间内的到确认，在重传计时器归0后触发重传。这个过程会对协议栈性能产生大量消耗。而如果长时间系统中没有数据接收或发送需求，那么cpu对于poll的循环调用就只是对于资源的消耗。从这个角度上，即使在异步在连续发送不能提升协议栈性能的情况下，实现异步依然是非常有意义的一件事。

最后，因为同步的设计模式下无法主动对接收进行检测，所以poll函数既负责接收，也处理发送逻辑，两者放在一起统一调度。这种设计本身就缺少细粒度控制，在最合理调用情况下依然有显著的性能无效开销，以上都是smoltcp同步设计的弊端。

综上，原始smoltcp同步模型的性能劣势并非出自协议封装等高层逻辑的实现上，而在于其调度机制，因为同步模型不能主动进行检测，而只能被动轮询，导致了其发送和接受的代码绑定，进而使得其无法充分利用 CPU 和网络资源，最终导致实际传

输速率远低于链路层的发送速率。

通过异步改造，引入了Waker注册，使用async/await模型按需触发poll_egress和poll_ingress，修改从根本上减少了无效轮询，提升协议栈的响应速度与吞吐量，是更合适的协议栈设计方案。

5.5 本章小节

本章主要介绍了在qemu环境下对smoltcp进行修改前后的对比测试，首先明确了物理链路在理想条件下的吞吐上限，为评估协议栈处理性能提供了基准。随后，在标准smoltcp同步模型下测试发现，TCP协议下由于其依赖轮询机制统一调度发送与接收操作，在高频 I/O 场景中引入了显著的性能开销，最终导致实际吞吐率仅为链路层测试结果的约40%。之后进一步对异步化改造后smoltcp进行测试，其通过引入async/await 与Waker机制，有效拆分poll_egress与poll_ingress操作，避免了同步模型下的无效轮询，显著提升了协议栈性能。

实验证实，该异步设计能够在不影响协议逻辑正确性的前提下显著降低CPU负载，提高协议栈整体的吞吐能力与实时性能。引入异步机制进行架构优化，为构建高性能嵌入式网络协议栈提供了更优解法。

结　论

本文聚焦于实现高性能的应用于嵌入式系统Rel4的异步网络协议栈方案，针对smoltcp协议栈收发使用同步模型，在密集io情况下性能差的情况，对收发模型进行了调用逻辑的修改，将收发的核心函数poll分离为poll_ingress和poll_egress函数，并封装为异步任务，使用异步模型进行了收发优化。本研究的主要创新点包括：

1.异步机制与轻量协议栈的融合：在资源受限的嵌入式环境中引入Rust异步模型，并结合轻量化的运行时，在不依赖传统运行时（tokio）对waker的自动注册前提下构建异步执行结构。

2.面向嵌入式系统和微内核的兼容性与移植性增强：对传统同步的socket API进行了异步调整，并优化了函数调用逻辑，为其在Rel4微内核中的嵌入部署与IPC协同通信奠定了可行路径。

从实际应用角度来看，该异步协议栈方案不仅显著降低了处理延迟与上下文切换开销，也具备良好的可移植性，可以很好的适配微内核和嵌入式系统的运行环境。

未来的研究工作可从以下几个方面进一步拓展：

1.与Rel4 IPC的深度集成：进一步与Rel4进行适配，优化协议栈与用户态网络服务之间的数据传输路径。

2.自定义异步执行器的构建：通过设计轻量级调度器模型，实现完全脱离外部运行时的更高效的任务驱动机制。

3.协议特性拓展：增加IP协议的分片等功能，从发送接收逻辑上提升协议栈面对高吞吐量和复杂网络。

综上所述，本文在微内核与异步协议栈结合方面做出了实质性的改进，具备较强的研究价值与工程意义，为后续Rel4的发展奠定了良好的基础。

参考文献

[1] 陈星光,雷先华.嵌入式控制系统在工业控制中的关键应用研究[J].电子元器件与信息技术, 2023(11):38-41.

[2] 魏煜康.嵌入式SMP环境下的TCP/IP协议栈并行优化研究与实现[D].北京邮电大学, 2024. DOI:10.26969/d.cnki.gbydu.2024.002188.

[3] 何立民.嵌入式系统的定义与发展历史[J].单片机与嵌入式系统应用, 2004,(01):6-8.

[4] 马义德,刘映杰,张新国.嵌入式系统的现状及发展前景[J].信息技术,2001,(12):57-59.

[5] Lano K, Yassipour Tehrani S. Safety-Critical and Embedded Systems Architectures[M]// Lano K, Yassipour Tehrani S. Introduction to Software Architecture: Innovative Design using Clean Architecture and Model-Driven Engineering. Cham: Springer Nature Switzerland, 2023: 191-209. DOI:10.1007/978-3-031-44143-1_10.

[6] Klein G, Andronick J, Elphinstone K, et al. SeL4: formal verification of an operating-system kernel[J]. Communications of the ACM, 2010, 53(6): 107-115.

[7] Klein G. The L4.verified Project — Next Steps[C]// *Verified Software: Theories, Tools, Experiments*. Berlin, Heidelberg: Springer, 2010: 86-96.

[8] Kurtz, Kevin C., "Microkernel security evaluation." (2012). Electronic Theses and Dissertations. Paper 784.https://doi.org/10.18297/etd/784

[9] 顾锡华. Rust语言在Web开发的应用研究[J].电脑知识与技术,2024,20(05):38-40.DOI:10.14004/j.cnki.ckt.2024.0186.

[10] Jung R, Jourdan J H, Krebbers R, et al. Safe systems programming in Rust[J]. Communications of the ACM, 2021, 64(4): 144-152. DOI:10.1145/3441307.

[11] 胡霜,华保健,欧阳婉容, 等. Rust语言安全研究综述[J].信息安全学报,2023,8(06):64-83.DOI:10.19363/J.cnki.cn10-1380/tn.2023.11.06.

[12] Bugden W, Alahmar A. Rust: The Programming Language for Safety and Performance[EB/OL]. (2022-06-11)[2023-12-01]. https://arxiv.org/abs/2206.05503.

[13] Jang H, Chung S H, Yoo D H. Design and implementation of a protocol offload engine for TCP/IP and remote direct memory access based on hardware/software coprocessing[J]. Microprocessors and Microsystems, 2009, 33(5-6): 333-342. DOI:10.1016/j.micpro.2009.02.008.

[14] Clark D D. The design philosophy of the DARPA internet protocols[C]// Proceedings of the ACM SIGCOMM Conference on Communications Architectures and Protocols. New York: ACM, 1988: 106-114. DOI:10.1145/52324.52336.

[15] Chiu D M, Jain R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks[J]. Computer Networks and ISDN Systems, 1989, 17(1): 1-14. DOI:10.1016/0169-7552(89)90019-6.

附　录

致　谢

值此