



清华大学

# 综合论文训练

## 操作系统宏内核网络管理模块接口的设计与实现

系 别 : 计算机科学与技术系

专 业 : 计算机科学与技术

姓 名 : 林 奕 辰

指导教师 : 戴 桂 兰 助理研究员

副指导教师 : 陈 渝 副教授

二〇二五年六月



# 关于论文使用授权的说明

本人完全了解清华大学有关保留、使用综合论文训练论文的规定，即：学校有权保留论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

作者签名：林奕辰

日期：2025年6月10日

导师签名：戴桂兰

日期：2025年6月10日



## 摘要

随着互联网技术的迅猛发展，网络协议栈在现代操作系统中扮演着越来越重要的角色。轻量级协议栈如 lwIP 和 smoltcp 已在嵌入式系统、物联网设备和资源受限环境中广泛应用。本论文研究并实现了在 ArceOS 操作系统基础上，兼容多架构的网络协议栈接口模块，以支持 lwIP 和 smoltcp 协议栈的集成。通过系统调用接口的设计和实现，提供了统一的网络操作接口，使得操作系统能够灵活切换不同的协议栈，从而满足不同应用场景的需求。

论文首先分析了 ArceOS 操作系统的设计架构，重点解决了网络协议栈与内存管理、系统调用接口的兼容性问题。基于 Rust 语言的安全性和高性能特性，本文设计了可扩展的网络系统调用接口，并实现了多种网络协议栈的并行支持。通过引入条件编译和特性标志，系统能够根据需求选择集成 lwIP 或 smoltcp 协议栈。为确保协议栈在多核环境中的线程安全，采用了互斥锁和其他同步机制。

在实现过程中，本论文设计并实现了与 Starry-Next 操作系统架构兼容的网络协议栈接口，通过四种架构实现了常见的网络系统调用，如 SOCKET、BIND、CONNECT、SENDTO、RECVFROM 等，支持多种网络协议栈的动态切换和无缝集成。实验结果表明，该设计能够显著提升网络通信的灵活性、可扩展性和性能，特别是在资源受限的嵌入式设备和高并发网络环境中。

通过在 QEMU 虚拟平台上进行功能和性能测试，本论文验证了所提出的网络协议栈接口设计的可行性。实验结果表明，该设计不仅能够提高操作系统的网络管理能力，还能提供更高效的网络协议栈支持，满足多样化应用的需求。

**关键词：**组件化操作系统；宏内核；ArceOS；starry-next；系统调用；轻量级协议栈；lwIP；smoltcp；内存安全；并发处理

## Abstract

With the rapid development of internet technologies, the network protocol stack plays an increasingly vital role in modern operating systems. Lightweight stacks such as lwIP and smoltcp have been widely adopted in embedded systems, IoT devices, and resource-constrained environments. This paper investigates and implements a cross-architecture network protocol stack interface module based on the ArceOS operating system, enabling the integration of both lwIP and smoltcp. By designing and implementing a unified system call interface, the operating system can flexibly switch between different protocol stacks to meet the demands of various application scenarios.

The paper first analyzes the architectural design of ArceOS, focusing on addressing compatibility issues between network protocol stacks, memory management, and system call interfaces. Leveraging the safety and performance features of the Rust programming language, we design an extensible system call interface that supports multiple protocol stacks in parallel. Through conditional compilation and feature flags, the system can selectively integrate either lwIP or smoltcp. To ensure thread safety in multicore environments, mutexes and other synchronization mechanisms are employed.

During implementation, a network stack interface compatible with the Starry-Next operating system architecture is developed, supporting common system calls such as SOCKET, BIND, CONNECT, SENDTO, and RECVFROM across four architectures. The design enables dynamic switching and seamless integration of multiple network stacks. Experimental results demonstrate that this approach significantly improves flexibility, scalability, and performance of network communication, particularly in resource-limited embedded devices and high-concurrency environments.

Functionality and performance tests conducted on the QEMU virtual platform validate the feasibility of the proposed network stack interface design. This design not only enhances the network management capabilities of the operating system but also provides efficient support for lightweight protocol stacks, meeting the diverse needs of modern applications.

**Keywords:** Componentized Operating System; Monolithic Kernel; ArceOS; starry-next; System Call; Lightweight Protocol Stack; lwIP; smoltcp; Memory Safety; Concurrent Processing

# 目 录

<b>第 1 章 引 言 .....</b>	<b>1</b>
1.1 研究背景与意义 .....	1
1.2 国内外研究现状 .....	2
1.2.1 相关理论研究现状 .....	3
1.2.2 相关实践研究现状 .....	5
1.2.3 ArceOS 与上述工作的比较 .....	6
1.3 研究目标与内容 .....	7
1.3.1 总体目标 .....	7
1.3.2 具体目标 .....	7
1.3.3 本文工作 .....	7
1.3.4 研究内容概述 .....	7
<b>第 2 章 ArceOS 操作系统架构分析与 starry-next 组件对接 .....</b>	<b>9</b>
2.1 ArceOS 架构总览与设计原则 .....	9
2.2 关键模块分析：任务管理与运行机制 .....	10
2.2.1 axruntime：运行时初始化核心 .....	10
2.2.2 axtask：线程级任务调度核心 .....	12
2.2.3 axprocess：多进程与会话支持 .....	13
2.2.4 axns：命名空间机制 .....	13
2.3 关键模块分析：文件系统管理与网络协议栈 .....	14
2.3.1 axdriver：设备驱动管理 .....	14
2.3.2 axfs：文件系统管理 .....	15
2.3.3 axnet：网络协议栈 .....	17
2.4 从 Unikernel 向宏内核的演进路径 .....	19
2.5 Starry-Next 宏内核框架设计与适配分析 .....	20
2.5.1 系统架构总览 .....	20
2.5.2 系统调用派发机制与运行流程 .....	22
2.5.3 核心模块协作与适配机制 .....	22
<b>第 3 章 lwIP 与 smoltcp 网络协议栈的集成与分析 .....</b>	<b>24</b>
3.1 lwIP 与 smoltcp 协议栈背景与设计理念 .....	24
3.2 功能与架构对比分析 .....	26

3.3 协议栈启动与数据处理流程 .....	26
3.4 lwIP 与 smoltcp 的实现机制与代码分析 .....	30
3.4.1 TCP Socket 实现机制对比 .....	30
3.4.2 TCP 数据收发与缓冲管理 .....	30
3.4.3 UDP 实现策略对比 .....	31
3.4.4 设备接口封装与拓展性分析 .....	31
3.4.5 lwIP 的内存结构与 pbuf 管理机制 .....	32
3.4.6 lwIP 回调机制与事件驱动流程 .....	32
3.4.7 smoltcp 的 poll 模型与异步适配机制 .....	32
3.5 ArceOS 中的协议栈集成与适配实现 .....	33
3.6 统一网络系统调用接口设计 .....	35
<b>第 4 章 网络管理组件设计与实现 .....</b>	<b>37</b>
4.1 开发环境与工具配置 .....	37
4.2 模块总体设计思路 .....	37
4.3 lwIP 协议栈适配与封装实现 .....	38
4.4 smoltcp 协议栈异步接口适配 .....	39
4.5 Socket 接口系统调用实现细节 .....	39
4.5.1 sys_socket .....	40
4.5.2 sys_bind, sys_connect, sys_listen, sys_accept .....	40
4.5.3 sys_sendto / sys_recvfrom .....	40
4.5.4 sys_setsockopt 与选项控制 .....	42
4.6 组件之间的协作与系统集成 .....	42
4.7 小结 .....	42
<b>第 5 章 网络管理组件的实验评估 .....</b>	<b>44</b>
5.1 测试用例构成 .....	44
5.2 通过本地编写的基础测例 .....	45
5.3 通过 libctest 测试套件验证网络功能 .....	47
5.3.1 测试结果分析 .....	48
<b>第 6 章 遇到的问题与解决方案 .....</b>	<b>51</b>
6.1 网络协议栈适配中的兼容性问题 .....	51
6.2 网络功能的测试问题 .....	51
6.3 网络模块的相关代码在开发中的问题 .....	52
6.3.1 recvfrom 函数问题 .....	53

6.3.2 sendto 问题 .....	53
6.3.3 setsockopt 问题.....	54
6.3.4 iperf 测试中的 IPv6 支持问题 .....	54
<b>第 7 章 总结与展望.....</b>	<b>56</b>
<b>7.1 本文工作总结 .....</b>	<b>56</b>
7.1.1 主要贡献.....	56
7.2 存在的不足 .....	56
7.3 后续研究方向 .....	57
7.4 总结 .....	58
<b>参考文献.....</b>	<b>59</b>
<b>附录 A 外文资料的书面翻译 .....</b>	<b>60</b>
<b>致 谢.....</b>	<b>91</b>
<b>声 明.....</b>	<b>92</b>
<b>在学期间参加课题的研究成果.....</b>	<b>93</b>

## 插图清单

图 2.1 ArceOS 系统架构图 .....	10
图 2.2 ArceOS 用户程序加载流程图 .....	20
图 2.3 ArceOS 系统调用流程图 .....	21
图 2.4 Starry-Next 系统调用流程图.....	23
图 3.1 lwIP 与 smoltcp 协议栈架构对比 .....	25
图 3.2 ArceOS 网络协议栈启动流程比较图 .....	28
图 4.1 sys_socket 系统调用的实现流程.....	40
图 4.2 sys_bind 系统调用的实现流程 .....	41
图 4.3 sys_connect 系统调用的实现流程 .....	41
图 4.4 sys_sendto 系统调用的实现流程.....	41
图 4.5 sys_recvfrom 系统调用的实现流程 .....	41
图 4.6 sys_setsockopt 系统调用的实现流程.....	42
图 4.7 网络组件与系统其他模块的协作流程.....	43
图 5.1 测试用例执行流程与日志输出示例.....	46
图 5.2 网络相关系统调用的依赖关系图.....	47
图 5.3 libctest 测试套件的执行流程 .....	48
图 5.4 各网络系统调用在测试中的调用次数统计.....	49
图 6.1 recvfrom 函数未能找到非阻塞套接字的问题示意图 .....	53
图 6.2 sendto 系统调用中 self_addr 为空的问题示意图 .....	54
图 6.3 setsockopt 系统调用未正确工作的示意图 .....	55

## 附表清单

表 3.1 lwIP 与 smoltcp 核心架构比较 .....	26
表 3.2 smoltcp 与 lwIP 在 HTTP 测试下的请求处理性能（单位：RPS） .....	29
表 5.1 网络系统调用功能测试覆盖情况.....	50



# 第1章 引言

## 1.1 研究背景与意义

操作系统作为现代计算系统中最基础的系统软件之一，其设计与实现一直是计算机系统研究中的核心课题。无论是个人计算机、服务器，还是嵌入式系统、云平台、边缘计算设备等，操作系统始终承担着资源调度、任务管理、设备驱动、文件系统、网络通信等核心职责。

从上世纪 60 年代最早期的批处理系统和时间共享系统，到今天支持虚拟化、容器化、异构计算的复杂系统架构，操作系统不断演进，以适应性能、安全性、可靠性、可扩展性等方面挑战。**内核架构**作为操作系统设计中的关键因素，不仅决定了系统的功能边界，也深刻影响了系统性能和模块间的协作方式。

随着应用场景的多样化与硬件架构的不断演进，传统通用操作系统难以在所有环境下均达到最佳性能和安全性。为此，**定制化操作系统**应运而生，即根据特定应用需求和场景对操作系统结构与功能进行专门设计与优化，以实现性能、安全性、资源利用率等方面的提升<sup>[1-2]</sup>。定制化操作系统不仅涉及对单个组件的优化，还可能重构整体内核架构，从而催生出多种内核设计范式，如宏内核（Monolithic Kernel）、微内核（Microkernel）、库操作系统（Library OS）、Unikernel、Hypervisor 等。

宏内核架构将大部分系统服务集中在内核空间，性能优越但缺乏灵活性；微内核则精简内核功能，将非核心服务移至用户空间，增强了安全性和模块化<sup>[3]</sup>；库操作系统将系统服务封装为可链接库，极大提升了定制能力和资源隔离<sup>[1,4]</sup>。Unikernel 将应用与最小化操作系统合二为一，适合云计算和边缘计算环境<sup>[4]</sup>。Hypervisor 通过虚拟化技术支持多租户资源隔离，广泛应用于数据中心<sup>[5]</sup>。此外，混合内核（Hybrid Kernel）和多核架构（Multikernel）也在兼顾性能与模块化方面取得了进展<sup>[6-7]</sup>。针对高性能计算，Exokernel 提供了对硬件资源的极致控制能力<sup>[8]</sup>。

典型的定制化操作系统如 Libra、Arrakis、IX、OSv 等在各自场景均展示了优于通用操作系统的表现<sup>[1-2]</sup>。例如，Libra 基于库操作系统思想，专为虚拟化环境设计；Arrakis 依托宏内核，着重资源管理与安全；OSv 则结合了 Hypervisor 和 Exokernel 的优势，优化云服务性能。

传统的**宏内核（Monolithic Kernel）**架构将大部分功能模块集中在一个统一的内核空间中运行，这种设计便于模块间高效通信和共享数据，因此在性能上表现优越。Linux、Windows 等主流操作系统都采用了宏内核设计。然而，宏内核的缺点也逐渐暴露：内核臃肿、维护困难、模块间耦合严重，导致系统在面对快速变化的软硬件需

求时缺乏灵活性。

相对地，**微内核（Microkernel）** 架构则试图将操作系统最小化，仅保留进程调度、内存管理、IPC 等核心功能，其它功能如文件系统、网络协议栈等运行于内核态。这种设计显著增强了系统的安全性和模块化程度，但也由于频繁的用户态/内核态切换带来了性能瓶颈。

在性能和灵活性之间，**组件化操作系统（Component-based OS）** 设计应运而生，尝试在宏内核与微内核之间寻求一种新的平衡。这类操作系统将系统核心功能划分为松耦合、高内聚的模块，各模块通过明确定义的接口协作运行，并能根据实际需求灵活组合和部署。

**ArceOS** 正是典型的组件化操作系统代表之一。它采用“库操作系统（Library OS）”与宏内核思想融合的方式，将传统内核功能打散为多个组件，使得系统具有类似微内核的模块化结构，又保留了宏内核的高性能通信特性。这一架构对于构建下一代适应“多平台、多场景、多功能”的操作系统具有重要意义。

尤其是在当下容器化与云原生技术快速发展的背景下，用户对操作系统提出了更高的要求——既要运行在性能受限的边缘设备上，也要能支持高度并发的分布式服务架构，还要具备运行标准 Linux 应用程序的能力。如何在一个灵活、可裁剪的操作系统架构下，提供与 Linux 应用兼容的运行环境，成为本研究的切入点和目标。

本论文围绕在 **ArceOS** 中设计与实现支持 Linux 应用的宏内核网络管理模块接口，不仅具备较强的现实需求，也具有学术研究的创新意义：

- 从理论上，探索组件化架构中宏内核模型如何更好地融合标准系统调用、进程模型与网络协议栈；
- 从实践上，推动面向 Linux 应用的轻量级兼容层构建，增强新兴操作系统的生态兼容能力；
- 从工程上，提供一个高性能、可验证、具备良好抽象分层的系统实现框架，具备推广与拓展潜力。

## 1.2 国内外研究现状

操作系统架构设计在过去几十年内经历了三次主要范式的变迁：

1. **单体式宏内核（Monolithic）**：如 UNIX、Linux 等，强调性能优先，系统功能高度内聚，所有模块运行在内核空间，适合传统服务器和桌面应用；
2. **微内核（Microkernel）**：如 MINIX、QNX、L4 系列，强调安全性与模块隔离，适合安全性要求高或嵌入式系统；
3. **模块化/组件化架构（Modular/Component-based）**：如 Fuchsia、Barrelfish、

**ArceOS**, 强调灵活配置、硬件多样性支持和跨平台兼容，适应“应用多样 + 硬件异构”趋势。

国外在操作系统模块化与兼容层设计方面已经取得了大量进展。比较有代表性的成果包括：

- **Microsoft Drawbridge (2011)**: 将 Windows 子系统封装为一个库操作系统，并运行在轻量级虚拟化容器中，实现高效兼容；
- **Google Fuchsia**: 其 Zircon 微内核提供统一的内核服务，用户空间运行多个组件服务，借助 FIDL 接口进行解耦通信；
- **Unikernel 项目（如 MirageOS、IncludeOS）**: 将应用与所需最小内核功能链接为一个镜像，用于轻量化部署和极致性能需求；
- **L4Linux**: 运行于 L4 微内核上的兼容 Linux 子系统，是微内核兼容层的重要实践；
- **WINE 与 WSL**: 分别以系统调用翻译和系统服务桥接的方式，在非 Windows/Linux 内核上运行其原生应用。

国内的研究也紧随其后。例如：

- 清华大学的 **ArceOS** 项目，结合库操作系统思想与模块化调度框架，旨在构建新一代灵活的嵌入式与云端 OS；更新的 **Starry/Starry-next** 项目，探索高度裁剪、高可配置性的组件操作系统架构，支持用户自定义内核功能裁剪；
- 北京大学、华中科技大学在 Linux 兼容层与模块虚拟化上也开展了相关研究。

### 1.2.1 相关理论研究现状

**Software Dock** 是一种支持操作系统组件化的系统，它采用基于代理的软件模型来管理生产者与消费者之间的交互。该系统的设计允许操作系统模块独立开发、独立部署，并能够灵活地组合使用。通过代理机制，Software Dock 实现了模块间的解耦，使得系统可以根据需求灵活调整功能和性能。此外，Software Dock 还能够通过灵活的配置管理机制，使得不同的模块能够在不同的应用场景下优化其性能，支持跨平台的模块适配。

在组件化操作系统的研究中，**Software Dock** 提供了一种高效的模块化管理方式，可以帮助开发者将操作系统的功能按照需求进行解耦，并灵活组装不同的模块，以便更好地满足特定的需求。

**THINK**<sup>[9]</sup> 是一种采用组件化编程模型的操作系统，其核心思想是通过定义模块间的交互接口，允许灵活组装组件，创建出适用于特定需求的操作系统。THINK 的设计重点在于通过抽象和接口的设计，使得操作系统能够灵活扩展，支持多种不同的应用场景。通过该模型，开发者能够快速地组装组件，实现操作系统功能的定制化，

而不需要从零开始编写整个操作系统内核。

与传统的操作系统架构相比，**THINK** 的模块化设计使得系统开发更加灵活，并且能够根据需求调整系统性能。其对模块化组件的支持为后续的操作系统功能扩展提供了便利，尤其在需要针对特定应用需求进行定制的场景中，展示了其强大的可扩展性。

**OpenCom** 是一种基于组件的系统，它定义了一个最小的运行时内核，并通过模块化组件来实现系统定制。该工作特别关注跨不同硬件平台的兼容性，并且由硬件提供商提供模块加载器和链接器，开发者基于这些工具进行应用开发。**OpenCom** 的设计思想是简化操作系统的内核功能，采用最小化内核的策略，并通过模块化设计确保操作系统能够适应各种硬件平台，具备高度的兼容性和灵活性。

通过 **OpenCom**，开发者可以通过加载和卸载不同的模块来定制操作系统，并根据硬件平台的需求调整系统功能。该设计理念为硬件平台的多样性和应用需求的变化提供了极大的适应性。

**Unikraft** 是一个开源的库操作系统（Library OS），它支持定制单地址空间的操作系统内核，主要面向在主机操作系统（如 Linux）上运行的云虚拟机应用。**Unikraft** 旨在提供一种轻量级、高效的操作系统架构，适用于云计算环境中的虚拟机和容器化应用。通过 **Unikraft**，开发者可以根据具体应用的需求，定制操作系统的各个功能，并将系统功能作为库进行编译，最终形成一个高度优化的运行时环境。

**Unikraft** 的设计理念与 **ArceOS** 中的组件化架构有很多相似之处，尤其是在灵活配置和按需加载方面。两者都强调通过组件化设计来提高系统的可定制性和可扩展性，同时都在云计算和虚拟化应用中提供了优化的操作系统支持。

**FlexOS** 是一种关注配置可隔离机制的操作系统，支持多种软硬协同的隔离策略，强调操作系统的安全性和可靠性。其设计重点在于支持多种隔离策略，如资源隔离、进程隔离和网络隔离等，确保操作系统在高安全性要求的环境中能够提供稳定和可靠的服务。**FlexOS** 采用模块化设计，允许开发者根据实际需求选择和配置隔离策略，以确保操作系统在不同环境下的安全性和可靠性。

与 **ArceOS** 的设计相似，**FlexOS** 也注重操作系统的灵活配置和模块化能力。通过灵活的模块配置，**FlexOS** 使得开发者可以根据特定的安全性和可靠性需求定制操作系统，进而在特定应用场景中提供更加稳定和安全的服务。

**Pebble** 是为特定应用程序设计的专用操作系统，采用服务器-客户端设计，包含一个精简的内核。该系统根据领域特定的需求进一步精简操作系统的功能，确保操作系统能够针对特定的应用场景进行优化。**Pebble** 的设计特别关注如何在有限的硬件资源下实现高效的操作系统功能，以满足嵌入式系统和物联网设备的需求。

虽然 **Pebble** 的目标与 ArceOS 的目标有所不同，但其精简内核和定制化设计的理念与 ArceOS 的组件化和灵活配置有相似之处。两者都强调根据具体的应用需求进行操作系统的定制，以便提高系统的运行效率和满足特定功能需求。

### 1.2.2 相关实践研究现状

**DragonOS** 是一个完全自主研发的 64 位操作系统，专为云计算环境中的轻量化需求而设计。该系统基于 Rust 编程语言开发，具备更高的内存安全性与系统可靠性，同时兼容 Linux 二进制接口，支持虚拟化技术，并在调度子系统和设备模型等方面具备良好的扩展性。

DragonOS 采用模块化的系统架构，允许开发者根据不同的场景灵活定制所需功能模块。例如，在资源受限或虚拟化场景中，可以剔除冗余模块，仅保留关键的系统组件，从而大幅提高运行效率与系统安全性。当前 DragonOS 已实现约四分之一的 Linux 接口，其长期目标是实现对 Linux 的完全兼容，构建一个开源、可控、适用于生产环境的大规模系统平台。

与 ArceOS 类似，**DragonOS** 强调组件化设计和可插拔模块机制，具备良好的可维护性和适应性，尤其适用于需要在高安全性与高可靠性环境中运行的云端服务系统。

**Asterinas**（星绽）是一个注重内存安全性和模块隔离性的现代操作系统内核。它完全基于 Rust 编写，并严格控制 `unsafe` 代码的使用范围，仅在可信计算基础（TCB）中允许出现，以最大限度提升系统的安全性。Asterinas 使用框内核架构，提供类似 Linux 的应用二进制接口（ABI），实现对 Linux 应用的无缝兼容。

该系统支持模块化开发，提供名为 OSDK 的开发工具包，专为内核模块与驱动开发者设计。模块可以被按需加载，开发者可以自由选择模块开源或闭源，极大提升了内核开发的灵活性。Asterinas 鼓励按需定制系统组件以满足不同用户场景，从而实现高效而可靠的内核运行时支持。

**Asterinas** 与 ArceOS 在组件化思路上高度契合。两者都主张通过模块抽象与接口设计实现系统功能的可重构性，并在虚拟化场景中提供安全、快速且可维护的内核基础。

**ByteOS** 是一个支持 POSIX 接口的模块化操作系统内核，目标是提供跨平台（如 riscv64、aarch64、x86\_64、loongarch64）的系统构建支持。它采用配置驱动的编译方式，用户可通过 `byteos.yaml` 配置文件定义根文件系统、文件系统实现（如 FAT32、ext4、ext4\_rs）及模块功能，最终通过 `Makefile` 在不同平台快速部署运行。

ByteOS 具备清晰的模块结构 (`crates → arch → modules → kernel`)，

其中各子模块独立构建，适合进行裁剪与定制化配置。其模块化设计适用于教学、实验以及构建定制化嵌入式系统，并具备良好的移植性和平台适配能力。

与 ArceOS 的组件架构类似，**ByteOS** 强调系统功能的最小核心与外围模块的解耦，并通过统一配置系统实现快速搭建与部署，是一个典型的现代轻量级可配置操作系统平台示例。

**BrickOS**<sup>[10]</sup> 是一个为适应多种异构平台而设计的积木式内核架构操作系统。该系统强调内核模块的高度解耦和结构清晰，通过明确定义各模块间的依赖关系，实现模块之间的边界隔离与接口协作，避免隐式状态共享问题。每个内核组件作为一个“积木单元”，可以根据需要灵活加载，构建出适配特定平台或应用场景的内核结构。

BrickOS 支持宏内核与微内核的灵活切换：在特权级下可将所有内核组件集中运行，构建类似宏内核的系统；也可将部分组件迁移至非特权级运行，实现类微内核的安全隔离。这种灵活的架构依赖于统一的硬件抽象层，使上层内核模块能够屏蔽底层资源差异，提供稳定一致的运行环境。

BrickOS 的设计包括四个关键方面：其一，内核机制的“积木化”抽象，确保各模块独立、可拼接；其二，根据平台特征与用户配置自定义系统架构与模块组合；其三，优化模块间通信延迟，提升整体性能；其四，统一异构平台的内存保护机制抽象，增强系统安全性。

BrickOS 与 ArceOS 在模块化思想上有高度契合点，特别是在模块解耦、按需加载与异构适配方面。其积木式内核理念为构建多场景、多平台统一支撑的系统平台提供了极具参考价值的设计路径。

### 1.2.3 ArceOS 与上述工作的比较

相比于这些现有的组件化操作系统，ArceOS 在设计上独具特色，尤其在网络管理、协议栈的接入方式、系统调用接口的兼容性等方面做出了显著的创新。例如，ArceOS 在网络子系统的实现上通过 axnet 组件，提供了对多种协议栈的支持，并通过模块化设计实现了协议栈与硬件的解耦。而且，ArceOS 还采用了轮询机制和零拷贝优化技术，在保证高性能的同时，确保了系统在资源受限环境下的高效运行。

ArceOS 的组件化设计不仅关注系统功能的可扩展性和灵活性，还特别注重系统调用层的兼容性，确保能够支持现有的 Linux 应用。这使得 ArceOS 成为一个兼具高效性、灵活性和兼容性的操作系统，能够适应各种应用场景的需求，特别是在嵌入式系统和高性能计算领域。

### 1.3 研究目标与内容

#### 1.3.1 总体目标

构建一个基于 ArceOS 宏内核架构的网络管理模块接口层，使其兼容主流 Linux 用户态应用（如 busybox、curl、netcat），并对接支持 lwIP/smoltcp 网络协议栈的用户程序运行环境，从而打通“系统调用 → 网络抽象 → 协议栈实现 → 硬件驱动”的数据通路。

#### 1.3.2 具体目标

- 实现一套兼容 Linux 系统调用语义的 syscall 接口层，具体包括网络管理等核心部分；
- 设计并封装一个网络接口抽象层（NetAPI），支持底层协议栈（lwIP、smoltcp）的可插拔机制；
- 支持 Linux ELF 格式程序加载与运行，提供必要的进程调度和内存空间支持；
- 在 starry-next 架构下完成接口对接，实现跨平台运行环境；
- 通过功能验证与性能评估，评估系统调用延迟、内存使用、网络吞吐等指标。

#### 1.3.3 本文工作

本研究的工作主要集中在 ArceOS 的网络管理模块接口层的设计与实现上，具体包括以下几个方面：

- 设计与实现 ArceOS 的网络管理模块接口层，支持 lwIP/smoltcp 协议栈的用户态应用；
- 研究与实现 Linux 系统调用的兼容性接口，确保 Linux 应用能够在 ArceOS 上运行；
- 设计与实现网络抽象层（NetAPI），支持不同协议栈的灵活切换与集成；
- 在 QEMU 虚拟化平台上进行实验验证，评估系统性能与功能完整性。
- 分析与解决在实现过程中遇到的兼容性问题，如页表不兼容、协议栈初始化失败等。
- 总结实现过程中的经验教训，提出改进建议与未来研究方向。

#### 1.3.4 研究内容概述

1. **操作系统架构分析：**分析 ArceOS 的系统架构、模块调度机制以及组件间接口设计，理解其支持模块解耦与系统性能的关键点；
2. **接口对接策略研究：**分析 starry-next 系统的初始化机制、页表结构、ELF 加载

方式等，提出兼容层的集成方案；

3. **模块设计与实现：**实现包括系统调用层、网络 IO 抽象层、信号与进程控制子系统，构建兼容 Linux 程序的运行支持；
4. **实验评估与性能分析：**设计测试场景，通过 QEMU 等平台对模块功能完整性与性能数据进行采集与分析。

#### 1.3.4.1 1.3.3 研究内容概述

本研究以 ArceOS 操作系统为基础，结合 starry-next 项目中的模块，探索组件化内核架构在 Linux 应用兼容与网络协议栈集成方面的设计与实现方法。研究工作涵盖架构分析、接口适配、功能实现与性能评估四个层面，具体内容包括：

1. **ArceOS 操作系统架构分析：**分析 ArceOS 的核心系统结构，包括模块调度机制、内核组件加载流程以及模块间接口设计。重点研究其如何通过组件化机制实现模块解耦、功能抽象与系统可扩展性，为后续模块对接提供理论基础与设计支撑（对应第 2 章）。
2. **starry-next 系统组件与兼容性机制分析：**研究 starry-next 在启动流程、页表管理、地址空间划分以及 ELF 文件加载等方面的机制，分析其组件设计特点，为构建 ArceOS 与 starry-next 的适配层和兼容接口提出对接方案（对应第 2 章）。
3. **网络协议栈的集成与对接策略：**深入分析 lwIP 和 smoltcp 两款主流轻量级网络协议栈的结构与接口风格，结合实际应用需求，设计统一的网络抽象接口，实现两者在 ArceOS 中的并行集成与可选切换机制，确保网络功能的灵活性与扩展性（对应第 3 章）。
4. **系统调用与兼容接口的实现：**构建与 Linux 应用程序兼容的系统调用接口层，支持基本的网络 IO 接口封装，解决接口语义差异带来的适配问题（对应第 4 章）。
5. **运行支持环境的构建与实验验证：**基于 QEMU 虚拟化平台搭建测试环境，通过加载用户态测试程序，验证系统各功能模块的正确性与兼容性，并对协议栈切换、系统调用延迟等关键性能指标进行测量与分析（对应第 5 章）。
6. **实现过程中遇到的问题与解决方案：**在系统集成与开发过程中，梳理实际遇到的问题，例如页表不兼容、协议栈初始化失败、调用上下文切换出错等，并分析其成因，提出有效的技术解决路径与系统优化方案（对应第 6 章）。

## 第 2 章 ArceOS 操作系统架构分析与 starry-next 组件对接

### 2.1 ArceOS 架构总览与设计原则

ArceOS 是一个采用 Rust 编写的实验性模块化操作系统，整体以 Unikernel 为基础形态进行设计，强调灵活性、可裁剪性与高效性。其目标在于通过组件化的方式，构建一个既可运行于嵌入式环境，又能适配通用计算场景的操作系统平台。整个系统围绕“高内聚、低耦合”的组件组织方式，将操作系统功能拆分为多个可重用的构建单元，并依据功能职责分层构建系统架构。

从系统构成角度看，ArceOS 的功能组件可划分为两类：一类为通用性强、与操作系统实现弱耦合的“元件”，如链表、页表、调度器、基本同步原语等，具有良好的可移植性；另一类为紧密结合 ArceOS 架构设计的“模块”，例如任务调度、虚拟内存抽象、系统调用处理、网络栈与文件系统接口等，通常依赖于 ArceOS 的特定设计理念，不易直接复用于其他系统。

为了支撑上述组件划分逻辑，ArceOS 借助 Rust 的 crate 管理机制构建了结构清晰、依赖有序的模块体系。每个组件以 crate 为最小单元存在，所有依赖关系均在 Cargo.toml 中显式声明，系统整体依赖图严格构建为有向无环图（DAG），有效避免传统系统中常见的循环引用和隐式耦合问题。对于极少数确实存在依赖回环需求的组件（例如任务调度与中断屏蔽之间的调用），ArceOS 提供了 extttcrate\_glue 特殊组件用于显式桥接，从而在结构上维持系统的清晰性。

目前，ArceOS 已实现超过 30 个可复用元件与 10 余个功能模块，涵盖运行时管理（axruntime）、内存管理（axalloc）、任务调度（axtask）、进程管理（axprocess）、命名空间机制（axns）、网络支持（axnet）、文件系统（axfs）、设备驱动（axdriver）等核心功能，具备良好的跨平台适配能力。系统已支持 x86\_64、RISC-V、AArch64 等主流处理器架构，并成功部署于 QEMU/KVM 虚拟化平台、树莓派、黑芝麻开发板等多种实际设备，显示出强大的可移植性。

从功能分层视角，ArceOS 的架构层次如下：

- **元件层**：位于架构底部，封装系统中与平台无关的基础能力，如内存管理算法、链表结构、自旋锁等，强调高复用性与最小依赖；
- **模块层**：构成系统的内核功能，包括任务调度、内存映射、网络协议栈、文件系统等，通常由 axruntime 在启动阶段按需初始化；
- **API 层**：将模块层能力以接口形式暴露给用户程序，支持 Rust 本地调用及 POSIX 兼容接口，方便移植已有 C 应用；

- **用户库层**: 提供对 Rust 标准库、libc 等用户态库的适配封装，提升生态兼容性；
- **应用层**: 运行于系统之上的最终用户程序，可直接使用上述接口访问系统服务。

系统在构建时允许通过 Rust 的 `feature` 条件编译机制裁剪模块功能。例如 `axalloc` 可支持 slab、TSLF、buddy 等多种内存分配算法，通过 Cargo 配置项可灵活启用所需算法，未启用部分不会参与编译，从而减小最终镜像体积，提高构建效率。

此外，在性能设计方面，ArceOS 秉持“本地直通调用”的思路。Rust 应用访问系统接口时可绕过传统系统调用的上下文切换，依赖零开销抽象（zero-cost abstraction）直接调用模块层接口，带来显著性能优势。实测数据显示，在文件 I/O 操作中，相比 POSIX 接口，ArceOS 的本地 API 访问路径可带来 50% 以上加速；在 Redis 应用场景下，系统延迟较 Linux 降低达 33%。

通过组件化结构、现代语言机制与跨平台能力的结合，ArceOS 构建出一个结构高度清晰、功能可裁剪、性能可优化、复用性强的操作系统体系架构，并为其向宏内核、虚拟化平台、微服务环境等多种部署形态演进提供了坚实的基础。对于元件层和模块层的架构分析见图 2.1。

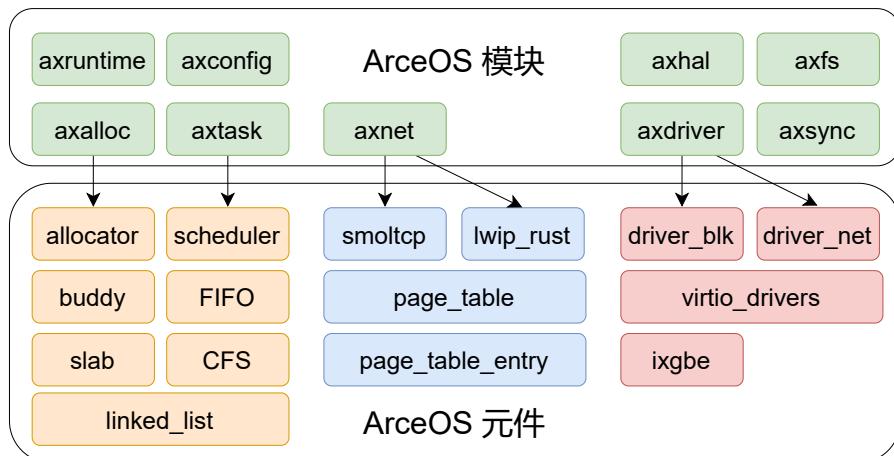


图 2.1 ArceOS 系统架构图

## 2.2 关键模块分析：任务管理与运行机制

ArceOS 的任务管理系统是通过 `axruntime`、`axtask`、`axprocess` 与 `axns` 四大模块共同支撑的。以下简要分析其职责与实现逻辑。

### 2.2.1 `axruntime`: 运行时初始化核心

`axruntime` 模块作为 ArceOS 的运行时核心，承担了在系统完成底层硬件初始化（由 `axhal` 模块完成）之后，进入用户主程序之前的所有内核初始化任务。它是

每个 ArceOS 应用构建镜像中必须链接的组件，负责构建运行时环境、调度关键功能模块的初始化、注册处理例程，并最终跳转执行用户应用主函数 `main()`。

其主要功能包括但不限于：

- **输出系统启动信息与平台参数：**在控制台输出 ArceOS 的标识、构建架构、目标平台、编译模式、SMP 核心数量等关键参数，方便调试与识别；
- **日志系统初始化：**调用 `axlog::init()` 初始化日志模块，并配置日志输出等级，后续的模块初始化过程均通过该日志系统进行记录；
- **内存管理器启动：**当启用 `alloc` 特性时，调用内存分配器初始化函数，优先选择最大的可用内存段作为主分配区，其余段注册为附加堆空间，构建全局堆；
- **页表与虚拟内存支持：**若启用 `paging` 特性，将加载内存映射相关元件（如页表管理），并建立初始的虚拟内存环境；
- **平台设备初始化：**调用 `axhal::platform_init()` 完成当前目标平台上串口、中断控制器、定时器等必要外设的统一初始化；
- **调度器与任务系统初始化：**启用 `multitask` 特性时，将加载并初始化内核调度器，构建任务队列、调度策略等运行时机制；
- **驱动与子系统加载：**若启用了 `fs`、`net` 或 `display` 特性，系统会通过 `axdriver::init_drivers()` 加载所有设备驱动并初始化对应子系统，如文件系统、网络协议栈、显示模块等；
- **多核处理器支持：**如启用对称多处理（SMP）功能，`axruntime` 将通过 `start_secondary_cpus()` 启动副处理器核心，并等待全部 CPU 完成同步初始化；
- **中断注册与启用：**启用中断功能（`irq` 特性）时，初始化定时器中断例程，完成中断分发器的注册，最后启用 CPU 中断机制；
- **线程本地存储初始化：**在未启用多任务功能但启用了线程局部存储（`tls`）的情形下，配置主线程的线程本地区域，初始化线程指针；
- **调用构造函数与异常处理钩子：**系统在最终转入用户主函数之前，统一调用所有构造函数（通过 `ctor_bare::call_ctors()`），并注册崩溃处理、任务标识记录等全局例程。

上述初始化过程通过条件编译机制进行裁剪，所有功能均通过 Cargo 的特性系统进行显式启用。源码中通过大量的 `#[cfg(feature = "...")]` 注解控制不同代码路径的编译与执行，使得运行时的行为严格由构建配置决定，避免无用代码被纳入最终镜像，提高系统定制性与安全性。

在多核启动场景下，主处理器首先进入 `rust_main` 函数并完成平台初始化，随

后通过统一的启动函数唤醒副核，副核进入 `rust_main_secondary`，执行包括内存页表、设备、调度器在内的独立初始化流程，最后完成同步后参与任务调度。

`axruntime` 作为 ArceOS 启动流程中的桥梁模块，不仅完成内核功能模块的统一调度与资源准备，还通过灵活的裁剪机制实现了运行时的高度可配置性，为系统构建多形态内核（如 Unikernel、宏内核或虚拟机管理程序）提供了良好的扩展起点。

## 2.2.2 axtask：线程级任务调度核心

`axtask` 模块是 ArceOS 中负责任务生命周期管理的核心组件，涵盖任务的创建、调度、阻塞、唤醒与销毁。其设计基于单地址空间的微虚拟机理念，以线程（任务）为最小调度单元，不区分用户态进程与线程。模块提供灵活的调度策略和完整的同步机制，是 ArceOS 支持多任务执行和并发管理的基础。

在 ArceOS 的设计中，任务被抽象为任务控制块（Task Control Block, TCB），每个任务包含唯一的 Task ID、调度状态、优先级、CPU 亲和性、栈信息和执行上下文（Context）。此外，还集成了线程本地存储（TLS）区域和用户自定义扩展字段，使得每个任务具备隔离的执行状态和灵活的扩展性。

该模块的核心功能包括：

- **任务创建与退出：**通过 `spawn` 或 `spawn_raw` 接口，可创建新任务并设置栈空间、任务名及入口函数，底层会构建 TCB 并初始化执行上下文。退出任务可通过 `exit` 接口，自动清理其占用的内存资源并触发等待队列的唤醒。
- **调度策略支持：**支持 FIFO（协作式）、轮转调度 RR（抢占式）和 CFS（完全公平调度），可通过 Cargo 特性灵活选择，适应不同实时性与公平性需求。调度器接收来自定时中断的时钟节拍，并根据就绪队列状态做出调度决策。
- **阻塞与唤醒机制：**提供 `WaitQueue` 实现线程阻塞与同步操作。任务可以在等待事件（如 I/O、条件变量）时主动进入等待队列，其他任务通过 `notify_one` 或 `notify_all` 唤醒被阻塞的任务，具备类似 Linux futex 的设计理念。
- **优先级与 CPU 亲和性：**提供 `set_priority` 和 `set_current_affinity` 等接口，可设置当前任务的调度优先级和 CPU 绑定策略（CPU mask）。若任务当前所在核心不在其亲和集内，调度器会将其迁移至指定 CPU 上执行。
- **空闲任务与线程回收：**每个处理器核心都会运行一个特殊的“idle task”，用于系统空转时节能等待中断。任务销毁由调度器统一完成，清理栈空间、上下文、TLS 和扩展字段，避免内存泄漏。

在实现层面，`axtask` 利用了 Rust 的 `Arc<T>` 智能指针构建任务引用系统，配合原子操作确保任务状态的并发安全性。调度队列由 `SpinLock` 保护，避免上下文切换期间产生竞态。多核支持（SMP）下，每个 CPU 拥有独立的运行队列，实现了任

务的负载均衡与跨核调度。

值得一提的是，`axtask` 还设计了扩展接口 `AxTaskExt`，支持用户为任务附加自定义结构体，可广泛应用于进程控制块（PCB）、资源统计、审计标识等扩展场景，提升了组件的可塑性与复用性。

`axtask` 模块不仅是 ArceOS 支持多任务执行的关键基础设施，同时也是其组件化理念的典范体现，通过灵活配置与清晰边界设计，达到了高可定制、可裁剪、可移植的系统任务管理能力。

### 2.2.3 axprocess：多进程与会话支持

为了在宏内核形态下支持多用户程序并发执行，ArceOS 引入了 `axprocess` 模块，作为进程与线程管理的核心支撑模块。该模块不仅实现了传统操作系统中的进程创建、销毁与调度关系维护，还构建了进程组（`ProcessGroup`）和会话（`Session`）等高级抽象，为进程间资源管理与协同提供了结构化机制。

每个进程（`Process`）作为系统调度的基本实体，拥有独立的地址空间、线程组、子进程链表、所属进程组与会话标识。用户可以通过 `Process::new_init` 创建初始化进程，也可以通过 `fork` 机制复制当前进程上下文生成子进程，从而模拟类 Unix 系统中的进程继承语义。进程终止由 `Process::exit` 触发，之后由 `Process::free` 完成资源回收及僵尸态清理。

为了实现精细化的进程管理，`axprocess` 提供了进程间关系接口，例如 `parent()` 和 `children()` 可获取当前进程的父子关系；`group()` 和 `create_group()` 支持进程组的动态构建与迁移；而 `create_session()` 则可新建独立的会话空间，使得终端控制和会话隔离成为可能。这些结构共同构成了进程间层级管理模型，支持系统调用的多进程交互场景。

在多线程支持方面，`axprocess` 允许用户在已有进程上下文中调用 `new_thread()` 接口生成附属线程，并将其加入线程组统一调度。线程终止由 `Thread::exit()` 实现，当线程组中所有线程退出后可自动触发资源回收。

`axprocess` 模块通过多级抽象管理机制，实现了从单线程任务模型向多进程、多线程并发模型的自然过渡。该模块的引入使得 ArceOS 不再仅限于运行单应用 `Unikernel`，而具备承载多应用、多用户程序的能力，为支持更复杂的系统调用、进程隔离、用户级服务等提供了基础设施。

### 2.2.4 axns：命名空间机制

`axns` 模块在 ArceOS 中引入了一种轻量级命名空间机制，为每个线程提供独立或共享的资源视图，从而在系统级别实现资源的粒度隔离与访问控制。与传统操作系统

全局共享资源的方式不同，axns 提供了灵活的资源隔离策略，尤其适用于需要高并发安全性和多任务运行的宏内核环境。

该模块的核心功能包括：

- **线程隔离与共享机制**：每个线程可独立拥有命名空间实例，从而隔离其工作目录、文件描述符等资源，也支持通过 ResArc 实现受控的资源共享；
- **懒初始化支持**：资源实例在首次访问时才被实际初始化，避免了无效的系统开销，提高了启动性能；
- **安全性与可维护性增强**：通过限制命名空间内资源的可见性，降低了线程间的资源竞争风险，提升系统鲁棒性；
- **自动化资源管理**：命名空间在销毁时自动回收内存，确保无资源泄漏。

在实际实现中，AxNamespace 类型定义了命名空间的结构体，支持创建全局命名空间（所有线程共享）和线程本地命名空间（通过 new\_thread\_local 方法为每个线程分配独立资源）。两者在底层使用统一的资源段布局（由 axns\_resource 链接段定义），保证资源访问逻辑一致性。线程本地命名空间在创建时会从全局命名空间拷贝初始值，进而实现资源隔离。

为了管理具体资源，axns 引入了 ResArc<T> 类型作为通用封装，它内部通过 LazyInit<Arc<T>> 实现资源的懒加载与引用计数共享。开发者可以使用 init\_new 或 init\_shared 接口分别创建独占或共享资源副本，确保资源仅在实际使用时初始化且被正确释放。

此外，axns 提供了 def\_resource! 宏，允许开发者以统一方式定义命名空间资源并自动注册至资源段中，实现跨线程的自动资源绑定和访问接口封装。该设计简化了资源接口实现，提高了模块间的解耦性。

axns 为 ArceOS 带来了灵活、低开销且高度安全的命名空间支持，在多线程或多进程系统中具备良好的扩展性与可维护性。

## 2.3 关键模块分析：文件系统管理与网络协议栈

在 ArceOS 中，axfs 和 axnet 模块分别负责文件系统与网络协议栈的管理与实现。它们通过统一的接口与底层驱动进行交互，为用户程序提供高效、灵活的文件与网络访问能力。

### 2.3.1 axdriver：设备驱动管理

axdriver 是 ArceOS 中用于设备抽象与管理的核心模块，主要功能是统一封装并调度不同类型的设备驱动，确保系统在多平台、多设备类型的运行环境下具备良好的

兼容性与可扩展性。该模块当前支持三大类主流设备：网络设备、块设备（如磁盘）以及图形显示设备。

其设计目标是通过模块化接口屏蔽设备底层差异，使设备驱动的集成、配置与运行更加高效灵活。axdriver 的主要功能包括：

- **设备抽象与接口统一**：为每类设备（如块设备、网卡、GPU）定义统一的操作接口，驱动开发者只需实现对应 trait 即可兼容框架调度，降低驱动编写与集成成本；
- **驱动自动探测与注册**：支持 PCI 总线与 MMIO 两种设备探测机制，可自动识别并注册系统中已启用的硬件设备；
- **静态与动态调度机制**：根据构建特性选择设备访问方式，支持静态分发（性能最优）与动态分发（灵活性强），例如在启用 dyn 特性时所有设备以 trait object 管理；
- **驱动模型与封装结构**：所有已探测驱动将被统一封装为 AllDevices 结构并分类保存（如 AxNetDevice、AxBlockDevice 等），上层模块如 axnet、axfs 通过解包访问具体驱动；
- **多设备支持与静态配置优化**：支持多实例设备的注册与调度；若明确只启用单一设备类型，可启用特性强制采用静态实例绑定，避免虚表分发带来的运行时开销。

目前，axdriver 模块已适配如下设备：

- **块设备**：ramdisk（内存盘）、virtio-blk（VirtIO 块设备）、bcm2835-sdhci（树莓派 SD 控制器）；
- **网络设备**：virtio-net（VirtIO 网卡）、ixgbe（Intel 82599 万兆网卡）、fxmac（飞腾平台适配网卡）；
- **图形设备**：virtio-gpu（VirtIO 显卡）。

在系统初始化过程中，axdriver 调用 `init_drivers()` 完成所有驱动的初始化与设备注册。所有设备的管理结构将在运行时传递给文件系统（axfs）、网络协议栈（axnet）等模块，用于构建具体的 I/O 通道和资源路径。通过特性组合与模块配置，axdriver 有效支撑了 ArceOS 在虚拟化平台、嵌入式板卡以及物理硬件等多场景下的运行需求。

### 2.3.2 axfs：文件系统管理

axfs 是 ArceOS 提供的文件系统模块，负责搭建虚拟文件系统（VFS）框架，并对多个具体文件系统进行统一管理与调度。该模块基于元件层的 `axfs vfs` 接口实现，定义了标准化的 inode、目录项、文件对象等抽象，使得不同类型的文件系统可以方

便地接入系统。

具体而言，axfs 实现了如下核心功能：

- **虚拟文件系统抽象层：**提供统一的接口封装，支持多种文件系统接入，如内存文件系统（ramfs）、设备文件系统（devfs）、进程文件系统（procfs）等；每种具体文件系统通过实现 `axfs_vfs` 接口即可被系统识别并挂载。
- **静态挂载与按需初始化：**系统在启动时根据配置文件或编译选项进行文件系统挂载，采用 Rust 的条件编译特性，自动选择启用的文件系统，避免无用代码的引入，降低系统体积。
- **类 Rust 标准库 API 接口：**axfs 模块对外暴露了接近 Rust `std::fs` 模块的文件操作接口，如 `File::open`、`File::read`、`File::write`、`File::metadata` 等，支持多种打开选项（`OpenOptions`），并集成 `Seek`、`Read`、`Write` 等 trait，使用户空间开发更加简洁直观。
- **目录管理与文件遍历：**提供 `Directory` 结构用于目录的打开与遍历，支持创建、删除、重命名、读取目录项等操作，并通过 `ReadDir` 迭代器返回 `DirEntry` 实体以支持按需遍历。
- **文件权限与元信息管理：**通过 `Metadata` 提供文件类型（`FileType`）、权限（`Permissions`）、大小、是否为目录等信息查询，支持对文件属性的访问与调试。
- **外部文件系统支持：**通过对第三方组件如 `rust-fatfs` 的集成，ArceOS 支持 FAT 格式等常见磁盘结构，提高系统与传统工具链的兼容性。

在内部实现中，axfs 通过封装 `VfsNode` 对象与统一的权限控制接口（如 `Cap`、`WithCap`）实现对文件节点的抽象与访问校验，保障文件操作的安全性。文件句柄（`File`）内部管理读写偏移，并支持追加模式与截断操作；目录对象（`Directory`）则维护遍历状态，支持迭代式读取与分层路径访问。

为了适配不同使用场景，axfs 中还实现了用于目录构建的 `DirBuilder`，支持递归创建与非递归控制，方便用户在运行时动态生成目录结构。此外，所有打开文件与目录均实现 `Drop` trait，确保系统资源在生命周期结束后被安全释放，避免文件句柄泄露。

axfs 以模块化设计提供了一个可扩展、可裁剪的文件系统支持框架，既适用于资源受限的嵌入式平台，也能满足复杂文件系统访问需求，为 ArceOS 构建通用性操作系统平台提供了稳定的数据存储基础。

### 2.3.3 axnet: 网络协议栈

axnet 模块是 ArceOS 中实现网络通信能力的关键组成，主要负责协议栈与网卡驱动的组织管理，并向应用程序提供统一的套接字（socket）接口。其设计充分考虑了可移植性、高性能和组件化需求，通过精细封装协议栈、驱动与接口逻辑，构建了独立于操作系统内核主线的网络子系统。

#### 协议栈抽象与选择机制

ArceOS 提供对两类主流网络协议栈的支持，分别为：

- **smoltcp**: 一个以安全性和可验证性为目标的 Rust 网络协议栈，适用于资源受限的嵌入式场景；
- **lwIP**: 一套成熟稳定、以 C 编写的轻量级 TCP/IP 协议栈，在嵌入式与实时系统中广泛应用。

这两种协议栈均以元件层组件的形式提供，分别通过 smoltcp\_impl 与 lwip\_impl 模块集成。lib.rs 中的 cfg\_if! 宏会根据编译特性选择使用哪一个协议栈，并为其注入统一接口实现，如 TcpSocket、UdpSocket 等结构体。

#### 驱动适配与初始化流程

axnet 通过 AxDeviceContainer 获取系统中的网卡设备（NIC），并调用 init\_network 函数对网络子系统进行初始化。该函数会选择一个主用网卡并绑定至当前协议栈上下文中。具体流程包括：

1. 解析系统配置或设备管理信息，选取主用网卡；
2. 调用协议栈模块的 init 方法初始化网络接口；
3. 创建 socket 集合结构，用于维护多套连接状态。

此外，loopback 接口以虚拟网卡形式集成在 smoltcp 中，并借助 LoopbackDev 实现内部数据回环机制，适配多协议栈统一使用。

#### 套接字接口与应用透明性

axnet 对协议栈提供的底层 socket 实现进行了统一封装，面向应用程序暴露兼容 POSIX 的标准 socket API。其支持的操作包括：

- TCP: connect、bind、listen、accept、send、recv；
- UDP: send\_to、recv\_from、connect、bind 等；
- DNS: resolve\_socket\_addr 支持通过域名解析获取 SocketAddr 地址结构。

该接口层具备可插拔性，能够动态绑定至不同的协议栈实现。开发者无需感知底层协议栈差异，即可编写一致性网络代码，大大简化了跨平台迁移与驱动维护成本。

## 传输模式与性能优化

为在资源受限系统中兼顾性能与实时性，axnet 当前采用基于轮询（polling）的网络调度策略。每次调用网络 API（如 `recv`、`accept` 等）时，会主动轮询一次网卡接口并刷新协议栈状态。

为减少内存复制开销，axnet 设计了一套零拷贝（zero-copy）缓冲区机制：

- 接收路径中，预先分配的缓冲区直接挂入网卡收包队列；
- 接收到的数据被协议栈直接消费并处理；
- 使用完毕后缓冲区被回收重用。

此机制在 lwIP 和 smoltcp 中均得到支持，并通过 `AxNetDevice` 统一管理，提高整体吞吐能力。

## 非阻塞控制与多态支持

axnet 所有 socket 实例均支持非阻塞（non-blocking）与重用地址（reuse address）配置，可通过如下接口调用进行管理：

- `set_nonblocking(bool)` 设置非阻塞模式；
- `is_nonblocking()` 查询当前状态；
- `set_reuse_addr(bool)` 设置是否允许地址复用。

此外，UDP 套接字支持连接模式（connect），允许指定固定收发端点以过滤非目标数据包，同时简化 `send/recv` 使用语义。

## 多协议栈兼容层设计

在内部实现层，axnet 利用了接口抽象与编译时特性选择机制，实现 smoltcp 与 lwIP 的统一封装。该设计具有如下特点：

- smoltcp 的实现以 Rust 安全特性为核心，Socket 状态通过原子操作与 `UnsafeCell` 管理，实现状态自动转移与共享访问控制；
- lwIP 的实现则基于 FFI 接口调用 C 语言栈结构，通过静态函数回调完成事件分发（如 `accept`、`recv`、`connect`）处理。

通过上述机制，axnet 成功将原生异构协议栈整合入统一运行框架，实现了在不同运行场景下灵活部署、透明切换的能力。

axnet 模块将底层协议栈的复杂性屏蔽在抽象层之下，构建了一套高性能、可插拔、接口统一的网络通信子系统。其支持的特性如 socket 多态、非阻塞控制、DNS 查询与轮询调度机制，为 ArceOS 在嵌入式与系统级应用中的网络支持提供了坚实基础。

## 2.4 从 Unikernel 向宏内核的演进路径

ArceOS 最初面向高隔离性与安全性的场景，采用 Unikernel 风格进行设计，将应用与内核逻辑高度集成于单一地址空间中运行。然而，随着多任务、多应用场景的需求增长，单地址空间限制逐渐暴露，组件的通用化与可扩展性变得愈发重要。ArceOS 借助模块化组件框架与统一系统调用接口，逐步演化为支持多用户程序和完整进程生命周期管理的基础宏内核系统。

本节将从系统初始化、运行时构建、用户程序加载、系统调用机制到退出流程，依次剖析 ArceOS 在演进过程中形成的关键支撑机制。

### 系统初始化与平台配置

系统启动阶段由 `axhal` 模块主导，通过读取 `axconfig` 中预定义的内存基址、内核加载地址、栈空间大小等信息，完成平台感知配置。随后生成目标平台对应的链接脚本，并跳转至启动入口（如 `_start`），依次执行内存控制寄存器配置、MMU 启用、页表创建及异常向量初始化，为操作系统提供统一抽象的硬件运行基础。

### 运行时构建与多特性初始化

当平台初始化完毕，控制权转交给 `axruntime` 模块，其通过特性宏条件加载对应子系统，如内存分配器、调度器、文件系统、网络栈等。特别是在多核平台上，`start_secondary_cpus` 用于启动副核；在启用 `irq` 特性下，系统完成中断派发逻辑注册，为后续内核调度与 IO 处理打下基础。

### 用户程序加载与独立地址空间建立

在支持用户程序运行方面，ArceOS 提供 `new_user_aspace` 接口创建隔离页表，并通过 `load_user_app` 完成 ELF 程序段解析、虚拟内存映射、权限标记与入口跳转等。随后，调度器通过 `spawn_task` 创建用户任务，并将其绑定至目标地址空间及用户栈，实现典型宏内核任务模型。相关逻辑图可以参考图 2.2 所示。

### 系统调用机制与资源访问抽象

不同的架构下有不同的系统编号表，系统调用为用户程序与内核模块的桥梁。系统通过注册统一陷阱入口（如 `register_trap_handler`），将用户态调用映射至 `handle_syscall` 中的派发逻辑处理。内核根据调用编号分发至对应函数，如 `sys_exit`、`sys_write`、`sys_clone` 等，实现任务退出、日志输出与资源申请等功能。相关逻辑图可以参考图 2.3 所示。

### 应用退出与资源回收

用户程序运行完毕后，通过 `sys_exit` 发起退出请求，调度器等待其终止，并释放其地址空间、栈内存与任务控制块等结构。若为单任务场景，系统终止运行；多任务情形下，调度器继续调度其他就绪任务。

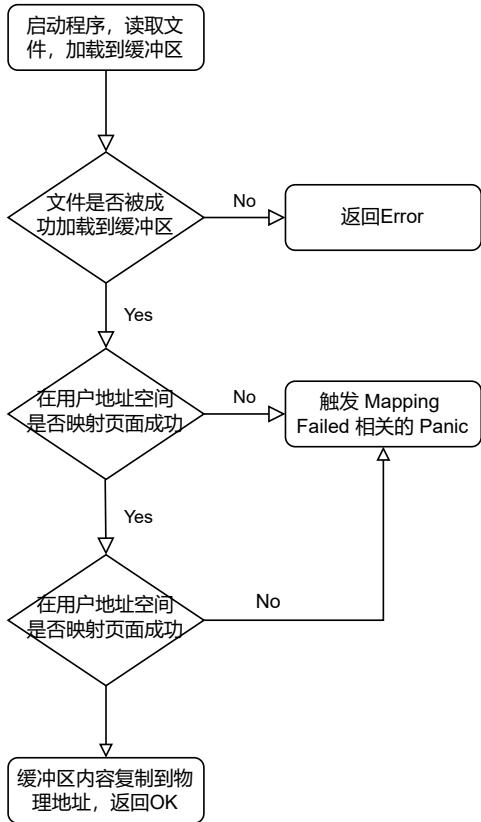


图 2.2 ArceOS 用户程序加载流程图

## 2.5 Starry-Next 宏内核框架设计与适配分析

Starry-Next 是基于 ArceOS 组件化基座发展而来的宏内核操作系统完整实现，其设计充分继承并深化了组件化与模块化理念。系统通过清晰划分核心功能模块，并定义统一且稳定的接口规范，实现了对多进程管理、多地址空间、协议栈多样化以及高效异步 IO 的全面支持，极大提升了内核的灵活性与可扩展性。

### 2.5.1 系统架构总览

Starry-Next 采用分层设计，将系统划分为三大逻辑层：

- **src 层**: 作为系统的用户接口入口，负责接收外部请求、解析用户输入，进行系统调用的分发与初步管理，是用户态程序与内核之间的桥梁。
- **api 层**: 封装系统调用的具体实现，为用户态应用提供丰富且稳定的系统服务接口。该层将系统调用映射至底层核心模块，按照功能划分为任务、内存、文件系统、网络等子模块，保证调用接口的一致性与扩展性。

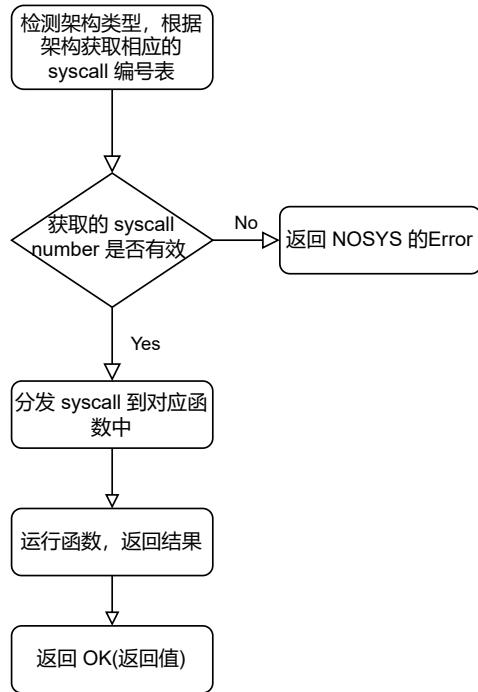


图 2.3 ArceOS 系统调用流程图

- **core 层：**承载系统的核心功能逻辑，包括任务调度、内存管理、文件系统操作、信号处理、网络协议栈等核心模块。该层依赖 ArceOS 提供的底层组件库，利用其完善的硬件抽象和基础服务支持实现复杂内核功能。

三层结构间通过明确定义的接口进行交互，模块间保持低耦合、高内聚，既支持组件的独立开发、测试与替换，又保证了整体系统的协同稳定运行。此外，Starry-Next 支持模块的动态加载与卸载，灵活应对运行时功能变更和裁剪需求。

其中，core 层的核心模块具体分析如下：

### 任务与进程管理模块

Starry-Next 支持完整的多进程模型。其任务模块通过调用 ArceOS 中 axtask 提供的 spawn\_task、yield\_now、exit 等接口创建调度实体。更高层的 axprocess 则封装了线程、线程组、进程组及会话管理逻辑，通过 api 层调用以支持 fork、execve、waitpid 等 POSIX 风格接口。

### 内存管理与地址空间控制

内存模块借助 ArceOS 的 axmm 组件完成地址空间的动态构建与映射维护。Starry-Next 通过接口函数如 map、unmap、mprotect 等实现用户空间堆、栈、共享内存段的分配与权限管理，支持按需映射、动态回收与安全隔离。

## 文件系统子模块

Starry-Next 使用 `axfs` 模块提供的通用文件操作接口，构建日志文件系统。通过系统调用封装层，用户可调用 `open`、`read`、`write`、`close` 等标准 API，实现对普通文件、目录及设备文件的统一访问。系统支持多文件系统挂载与动态加载，提升了文件存储灵活性。

## 网络协议栈与异步 IO 支持

Starry-Next 在网络模块设计上特别强化异步 IO 能力。其 `net` 子模块通过封装 TCP/IP 协议栈（如 lwIP 或 smoltcp），提供非阻塞 socket 接口。系统调用如 `sendto`、`recvfrom`、`setsockopt` 可实现事件驱动收发控制，结合 poll epoll 接口机制，实现用户层高并发网络服务支持。

此外，为适配多个协议栈共存，Starry-Next 在驱动层定义虚拟网络设备抽象，通过特征切换与注册机制自由切换实际使用协议栈。系统默认提供 IPv4 栈实现，并预留 IPv6、TLS 等后续扩展接口。

### 2.5.2 系统调用派发机制与运行流程

在 Starry-Next 中，系统调用的处理流程体现了清晰的层次划分和高效的调度机制，见流程图 2.4。系统启动时，框架完成核心模块初始化并注册系统调用函数表，建立从调用编号到处理函数的映射关系。运行时，用户态请求系统服务时，调用被封装为统一的 `syscall` 格式，由 `src` 层接收并进行初步参数校验。

随后，`src` 层将请求传递至 `api` 层，`api` 层依据系统调用号查找对应处理函数，执行前置准备与参数转换，并将调用下发至 `core` 层。`core` 层具体执行内核逻辑，如进程调度、内存操作、文件读写或网络数据传输，并返回执行结果。

当系统调用涉及阻塞或等待事件时，`core` 层会通知调度模块挂起当前任务，切换到其他就绪任务以保证系统整体的响应性和资源利用率。调用完成后，结果沿调用链反向返回至用户态程序，确保调用链的完整性和正确性。

整个流程强调了模块间责任清晰、接口统一和调用高效，兼顾了系统调用的兼容性和性能需求。

### 2.5.3 核心模块协作与适配机制

Starry-Next 的宏内核设计不仅依赖于模块划分，更强调模块之间的协同工作和灵活适配。核心模块如任务管理、内存管理和文件系统通过调用 ArceOS 提供的底层接口实现各自职责，保证了内核服务的稳定性和一致性。

任务管理模块通过调用 ArceOS 的 `axtask` 和 `axprocess` 组件，负责创建、调度和销毁内核任务与用户进程，支持线程组与会话管理，满足复杂多任务环境需求。

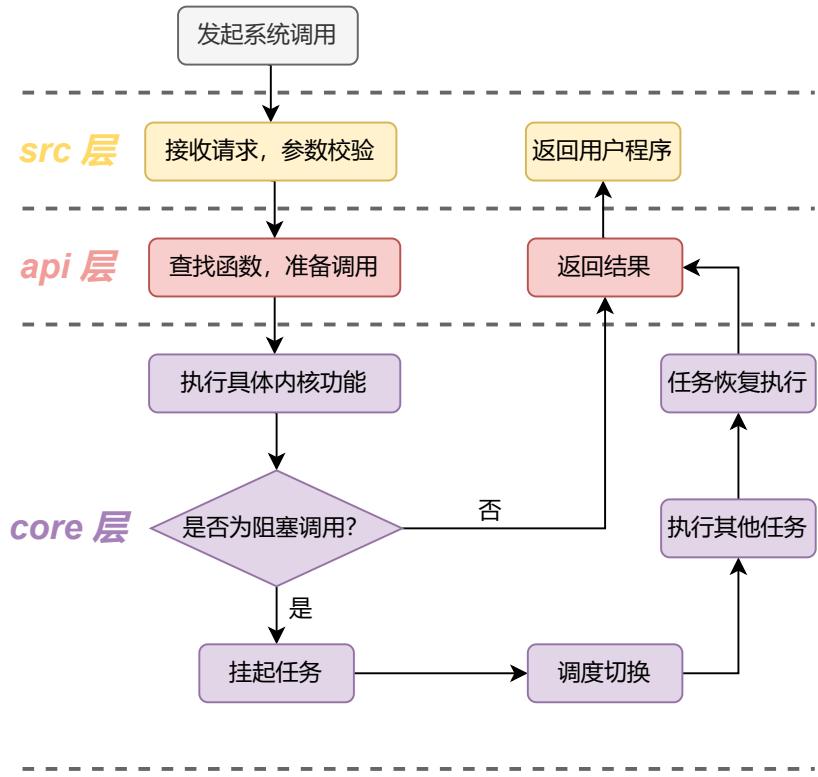


图 2.4 Starry-Next 系统调用流程图

内存管理模块基于 axmm 组件，动态维护用户地址空间的映射关系，实现内存权限保护和按需分配，确保多进程内存隔离和高效利用。

文件系统模块借助 axfs 接口提供统一的文件访问抽象，支持多文件系统挂载和设备文件管理，简化用户程序对存储资源的访问。网络模块则以抽象的网络设备和协议栈接口为基础，支持多协议栈并存和异步 IO，保障高并发网络服务的稳定运行。

这些核心模块之间通过标准接口和事件机制实现紧密协作：任务模块在进程创建时请求内存分配；文件系统操作中涉及内存缓存管理；网络通信时任务可能进入等待状态，触发调度器切换。模块间的接口设计和依赖关系既保证了功能的完备，也提高了系统的灵活性和可维护性。

通过以上协作机制，Starry-Next 能够在保持宏内核高性能的同时，实现模块化设计带来的灵活定制和跨平台适配，满足复杂多变的应用需求。

# 第3章 lwIP与smoltcp网络协议栈的集成与分析

## 3.1 lwIP与smoltcp协议栈背景与设计理念

在操作系统的网络子系统中，TCP/IP 协议栈扮演着核心角色。随着嵌入式设备、物联网和轻量级虚拟化的发展，传统全功能协议栈（如 Linux 内核网络栈）面临资源开销大、适配复杂等问题。因此，轻量化、模块化、高可移植性的 TCP/IP 协议栈成为操作系统设计中的关键组件。

在本论文所设计的 ArceOS 与 starry-next 系统中，出于灵活适配多种部署环境的需求，选取了两种轻量级协议栈：**lwIP** 与 **smoltcp**。二者在设计理念、目标平台、功能覆盖与性能特点上各具特色，满足从边缘节点到高性能虚拟机等多样化使用场景。

### lwIP 协议栈简介

lwIP（Lightweight IP）由 Adam Dunkels 于 2001 年提出，最初用于 uIP 项目的高性能补充版本，后由瑞典计算机科学研究院维护。它采用 C 语言实现，结构紧凑，功能全面，适用于具有操作系统支持或裸机环境的嵌入式平台。

**设计目标** 其核心目标是提供完整 TCP/IP 协议支持，最小化资源消耗，同时具备良好的可移植性。

### 主要特性

- 支持 TCP、UDP、ICMP、IGMP、IPv4/IPv6 协议；
- 支持多线程/无线程运行模式（通过 NO\_SYS 配置切换）；
- 具备 DNS、DHCP、SNMP、PPP 等高级协议模块；
- 内存使用灵活：支持内存池、堆、pbuf 链等多种分配方式。

### 适用场景

- 嵌入式系统与中型设备，运行 RTOS 或裸机主循环；
- 对网络兼容性要求高，需要支持 DNS、DHCP 等协议；
- 高吞吐、高并发的微服务器部署（如边缘网关）。

### smoltcp 协议栈简介

smoltcp 是一个由 Rust 社区开发、面向 IoT 和嵌入式平台的现代化网络协议栈。以安全、最小化和可组合为核心设计原则，其源码结构清晰，便于阅读和集成，尤其适合资源极端受限的系统。

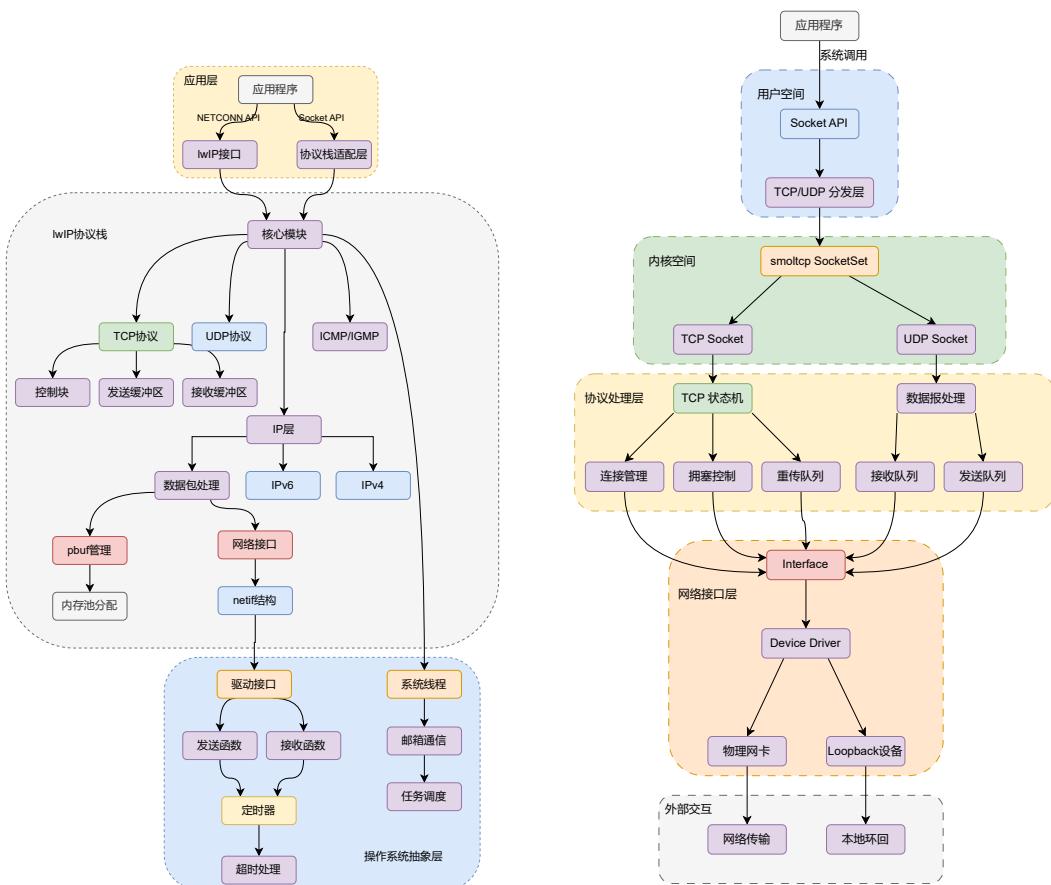
**设计目标** 实现高度模块化、零动态内存分配的 TCP/IP 协议栈，适应异步、事件驱动的系统架构。

## 主要特性

- 完全采用 Rust 编写，利用语言级内存安全机制；
- 支持 TCP、UDP、ICMPv4、ARP、IPv4 等协议；
- 零动态内存分配：所有资源在初始化阶段分配；
- 提供非阻塞 API，适用于 `async/await` 与轮询模型。

## 适用场景

- 极度资源受限设备，如 IoT 控制器、传感器节点；
- 使用 Rust 异步模型的裸机或嵌入式系统；
- 需要语言级内存安全保证的嵌入式研发环境。



(a) lwIP 协议栈架构示意图

(b) smoltcp 协议栈架构示意图

图 3.1 lwIP 与 smoltcp 协议栈架构对比

## 3.2 功能与架构对比分析

表 3.1 lwIP 与 smoltcp 核心架构比较

特性维度	lwIP	smoltcp
编程语言	C	Rust
内存管理	动态/静态混合, 支持堆与内存池	完全静态, 零动态分配
协议覆盖	TCP/UDP/ICMP/IGMP 等	TCP/UDP/ICMP/ARP 等
多线程支持	支持 (sys layer)	非阻塞设计, 支持 async
定时机制	内置定时器 + 任务调度	用户提供 tick 驱动
使用场景	嵌入式、软路由、RTOS 环境	IoT、嵌入式裸机
安全特性	无语言级保护, 依赖开发者控制	Rust 保证内存安全
可裁剪性	配置选项众多, 可模块裁剪	特性标志清晰, 易于组合
文档支持	完善的移植与集成文档	Rust 社区持续维护, 结构规范

lwIP 使用传统网络协议栈的同步模型, 数据包从驱动上传, 经过 pbuf 缓存池分发到各协议模块, 最后到达应用 socket 层。smoltcp 使用事件轮询机制, 应用主动 poll() 网络接口, 读取缓存区或传递发包数据, 由应用控制时序。

## 3.3 协议栈启动与数据处理流程

为了进一步理解 lwIP 与 smoltcp 在 ArceOS 中的运行机制与设计思路, 本节对两种协议栈在系统启动、网络设备初始化、数据包处理及与应用交互过程中的完整流程进行了深入分析, 并结合流程图进行直观展示。这不仅有助于理解其内在运行逻辑, 也为后续调试与性能优化提供技术基础。

### lwIP 启动与数据处理流程

lwIP 作为一款成熟的轻量级网络协议栈, 其启动过程主要包括协议栈结构初始化、网络接口注册、回调函数绑定以及定时机制的启用等步骤。由于 lwIP 支持两种运行模式 (带操作系统或裸机模式), 在 ArceOS 的集成中我们采用的是裸机模式, 即 NO\_SYS=1 配置下的手动轮询调度方式。

协议栈启动后的数据处理流程可概括如下:

- 在系统初始化阶段调用 `lwip_init()` 函数完成协议栈的全局初始化, 包括内存池、pbuf 缓冲区、TCP 控制块 (TCB) 等资源的配置;

- 使用 `netif_add()` 函数注册网络接口，并指定输入处理回调函数（如 `ethernet_input`），同时通过 `netif_set_up()` 启用该接口；
- 对于定时事件，如 TCP 重传、ARP 超时等，使用定时任务线程，来进行定期调用 `sys_check_timeouts()`，从而驱动协议栈内部的定时器；
- 网络驱动在接收到数据帧后，将其封装为 pbuf 结构，并通过输入回调函数传递至协议栈入口；
- 协议栈根据数据类型（IP、ARP、ICMP 等）进行解析，并分发到相应的协议模块；
- 应用程序通过 `socket` 或 `netconn` 接口与 TCP 或 UDP 层交互，完成数据的发送与接收。

### **smoltcp 启动与数据处理流程**

与 lwIP 不同，smoltcp 遵循高度模块化和最小化的设计原则，采用主动式轮询模型，并不依赖于中断或回调机制。这使得其在嵌入式或裸机系统中具备更好的时序控制能力，适合资源紧张或任务驱动型场景。

其启动与数据处理流程如下：

- 使用 `InterfaceBuilder` 创建网络接口结构体，配置本地 IP、MAC 地址、缓冲池、MTU 等信息；
- 使用 `SocketSet` 管理全部 `socket` 资源，每个 `socket` 均以静态方式注册；
- 通过周期性调用 `iface.poll()` 函数轮询网络设备，读取驱动中接收到的数据包，并触发协议层逻辑处理；
- `poll` 函数内部根据 `socket` 状态进行数据转发、连接维护、ACK 回应等操作；
- 应用程序使用非阻塞方式操作 `socket`，主动轮询读取或写入数据；
- 所有资源均在初始化时静态分配，运行过程中不进行动态内存分配，提升系统可预测性与实时性。

图 3.2 展示了 lwIP 与 smoltcp 在 ArceOS 中的启动与数据处理流程对比。两种协议栈在设计理念、资源配置、数据处理机制等方面存在显著差异，适用于不同的应用场景。

两种协议栈在启动流程、资源配置和数据处理机制上存在显著差异：

- **调度模式：** lwIP 依赖定时器驱动和回调函数，适用于具备定时机制或 RTOS 支持的系统；而 smoltcp 全部采用显式调用 `poll()`，适合裸机与 `async` 模型；
- **资源管理：** lwIP 使用动态和静态混合方式分配内存，而 smoltcp 完全避免动态内存分配，更适合对实时性要求高的场合；
- **使用方式：** lwIP 适合传统 `socket` 模型，smoltcp 则倾向于事件驱动架构，接口

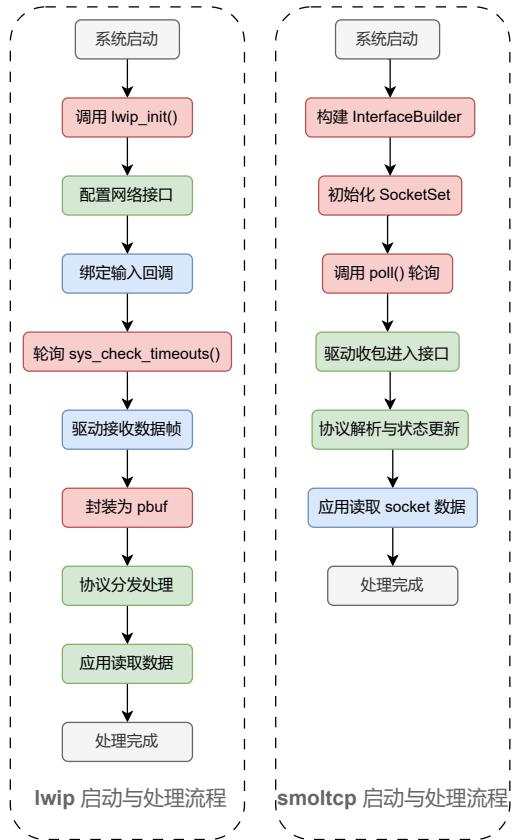


图 3.2 ArceOS 网络协议栈启动流程比较图

清晰但灵活度略低；

- **调试与扩展性：** lwIP 功能更丰富，但内部模块耦合较高；smoltcp 结构简洁，模块之间边界清晰，便于移植与扩展。

两种协议栈各有侧重，在 ArceOS 的网络系统中实现双协议栈支持，能够覆盖从轻量 IoT 到高负载边缘网关的多种使用场景，为系统的灵活部署和协议栈演进提供了坚实基础。

为了增强 ArceOS 在网络通信方面的性能与兼容性，系统同时集成了两种轻量级协议栈：Rust 实现的 smoltcp 与 C 实现的 lwIP。两者均通过统一的接口抽象在 axnet 中封装，对上层应用提供一致的 Socket API，包括对 TCP、UDP、DNS 等常用网络功能的支持。

相较而言，lwIP 在多连接处理、高并发负载与协议功能完备性方面展现出更佳表现，适合需要较强协议兼容性和吞吐性能的应用场景。

**协议栈适配机制** ArceOS 中通过实现 `NetDevices` 接口将底层驱动与协议栈解耦，并为 lwIP 提供裸机级初始化流程 (`lwip_init`)、设备注册 (`netif_add`) 与数据收发接口封装 (`myif_input`, `myif_link_output`)。该流程兼容 IPv4/IPv6，支持零拷贝内存复用机制，提高数据处理效率。

**TCP 支持优化** 在 TCP 支持方面，lwIP 通过回调方式处理连接与数据交互。适配时，为每个 TCP 连接创建固定地址的内存对象，并通过互斥锁实现阻塞等待功能，实现接口 `connect`、`bind`、`accept`、`recv`、`send` 等函数以支持标准 Socket 使用模式。

**UDP 与 DNS 支持** 对于 UDP 协议，lwIP 利用其 Raw API 实现非连接数据报通信，并支持自定义端口监听和目标发送功能。DNS 查询采用非阻塞模式，通过状态回调与缓存命中快速返回，最终提供一个统一的阻塞式地址解析接口。

**性能与稳定性对比** 在标准测试环境下使用 Apache Benchmark 工具对 HTTP 服务并发访问进行测试，结果如下：

表 3.2 smoltcp 与 lwIP 在 HTTP 测试下的请求处理性能（单位：RPS）

并发数	smoltcp	lwIP
1	7123	7726
2	7444	8698
5	7747	8752
10	7820	8737
50	7406	8704
100	7480	8764

由上可见，在多种并发情况下，lwIP 性能均优于 smoltcp，尤其在高并发下保持更强稳定性，且请求失败率更低，不易卡顿。

**优化措施** 为进一步提高 lwIP 性能，系统引入以下优化：

- 收包零拷贝：**利用 `pbuf_custom` 配合自定义析构函数，实现驱动层内存复用，避免重复拷贝。
- 内存池优化：**对 TCP 连接、数据结构、队列等进行预分配或内存池管理，显著降低堆分配频率。

- **参数调优:** 调整 TCP\_MSS、TCP\_WND、MEMP\_NUM\_TCP\_PCB 等参数，提升窗口大小和连接并发能力。
- **任务处理机制:** 基于裸机模式实现的轮询与超时检测逻辑结合 lwip\_loop\_once 调度函数，增强协议栈响应性。

## 3.4 lwIP 与 smoltcp 的实现机制与代码分析

### 3.4.1 TCP Socket 实现机制对比

#### 1. 连接与状态管理机制

lwIP 和 smoltcp 在 TCP 状态机处理方面有本质区别：lwIP 主要通过 C 语言的全局状态变量与回调注册处理连接建立和数据传输；smoltcp 则采用 Rust 的状态封装与安全抽象，通过状态枚举 State::SynSent, State::Established 等来判定连接状态。其连接建立过程使用接口：

- lwIP: tcp\_connect(pcb, ip, port, connect\_callback) 注册回调，在异步事件中设定状态；
- smoltcp: 直接使用 socket.connect() 发起连接，并封装连接检查逻辑于 poll() 中。

#### 2. 非阻塞与线程模型支持

smoltcp 提供 nonblock 原子布尔量，通过 block\_on() 包装轮询逻辑实现非阻塞 I/O。而 lwIP 则通过 NO\_SYS=1 模式实现轮询机制，并手动管理等待队列。

代码中，smoltcp 通过 TcpSocket::poll() 实现如下轮询：

---

#### 代码 1 smoltcp TCP 状态轮询示例

---

```

1  match socket.state() {
2      State::SynSent => false, // 等待服务器回应
3      State::Established => true, // 已连接
4      _ => false,
5 }
```

---

而 lwIP 中通过定期调用 lwip\_loop\_once() 推进协议状态。

### 3.4.2 TCP 数据收发与缓冲管理

#### 1. 数据接收缓冲

在数据接收方面：

- smoltcp 使用 recv\_slice() 直接读取内部环形缓冲区，结合 may\_recv() 控制连接关闭状态；

- lwIP 接收数据使用 `recv_callback` 存入 `recv_queue`, 以 pbuf 结构形式保存, 每次拷贝需计算偏移量。

如下是 lwIP 的数据读取逻辑:

---

#### 代码 2 lwIP TCP pbuf 数据读取片段

```

1 let (p, offset) = recv_queue.pop_front().unwrap();
2 let copy_len = min(len - offset, buf.len());
3 buf[0..copy_len].copy_from_slice(&payload[offset..offset + copy_len]);
4 if offset + copy_len < len {
5     recv_queue.push_front((PbuffPointer(p), offset + copy_len));
6 } else {
7     pbuf_free(p);
8 }
```

---

## 2. 数据发送路径

两者在发送逻辑上也存在差异:

- lwIP 调用 `tcp_write() + tcp_output()`, 要求手动控制缓冲区使用与传输输出;
- smoltcp 采用 `send_slice()` 接口, 由内部机制控制发送窗口与 ACK 协议。

### 3.4.3 UDP 实现策略对比

UDP 协议在 smoltcp 中作为轻量模块实现, 封装了基本的 `send_to` 与 `recv_from` 接口, 支持连接式与无连接式使用。smoltcp 使用 Rust 中的 `RwLock<Option<Endpoint>>` 封装本地和远程地址。

lwIP 则通过注册 `udp_recv` 回调, 将数据以 pbuf 加入队列, 类似 TCP 接收机制。

### 3.4.4 设备接口封装与拓展性分析

#### 1. smoltcp 的 Device Trait

smoltcp 通过 `Device trait` 设计接口, 要求实现 `receive()`, `transmit()`, `capabilities()` 三个方法。例如 loopback 接口如下:

---

#### 代码 3 smoltcp Loopback 接口结构体实现摘要

```

1 impl Device for LoopbackDev {
2     fn receive(...) -> Option<(RxToken, TxToken)> {
3         self.queue.pop_front().map(...)
4     }
5     fn transmit(...) -> Option<TxToken> { ... }
6     fn capabilities(...) -> DeviceCapabilities { ... }
7 }
```

---

## 2. lwIP 的回调与全局状态模型

lwIP 所有设备行为（收包、发包）都基于驱动调用 pbuf 提交与调用者调度进行，不支持多实例或 trait 抽象，扩展性受限。

### 3.4.5 lwIP 的内存结构与 pbuf 管理机制

在 lwIP 中，pbuf（protocol buffer）是网络数据在协议栈中传输的基本结构，起到缓存和抽象封装数据包的作用。pbuf 支持链式组织，以应对碎片化或 MTU 分段等情况。

pbuf 提供如下几种类型：

- **PBUF\_RAM**: 在 RAM 中分配的可修改数据块，常用于上层应用构造发出数据；
- **PBUF\_POOL**: 来自内存池的 pbuf，适合频繁分配释放的数据包；
- **PBUF\_REF**: 仅保存对现有数据的引用，避免内存拷贝；
- **PBUF\_ROM**: 用于只读数据（如静态响应内容），仅做引用，不拷贝数据。

在 ArceOS 中，pbuf 管理机制通过零拷贝优化得以强化。驱动层接收到网卡数据后，可直接封装为 pbuf\_custom，绑定自定义释放函数，从而将原始数据直接传入协议栈，无需额外拷贝。如下所示：

---

#### 代码 4 pbuf\_custom 零拷贝封装示例

---

```
1 struct pbuf_custom_custom *pc = alloc_custom_pbuf(buffer, len);
2 pc->pbuf.custom_free_function = my_pbuf_free;
3 pbuf_chain(p, &pc->pbuf);
```

---

该机制配合预分配的 pbuf 池和页对齐分配策略，显著降低内存开销并提升收包性能。

### 3.4.6 lwIP 回调机制与事件驱动流程

与传统的阻塞 IO 不同，lwIP 在裸机或 NO\_SYS 模式下采用事件驱动模型。其核心机制为对每个 TCP 或 UDP socket 注册回调函数，如 `tcp_recv`、`tcp_accept` 等。

以下展示了 TCP 收包回调函数在 ArceOS 中的注册与触发：

该机制通过预注册的函数实现从驱动中断到用户态事件的快速传递，结合 ArceOS 中断轮询框架，在极简系统中亦能维持 TCP 全功能通信。

### 3.4.7 smoltcp 的 poll 模型与异步适配机制

smoltcp 的核心为轮询驱动的非阻塞模型。应用需显式调用 `poll()` 接口进行状态更新与数据交互。在 ArceOS 中，系统通过异步任务调度框架（如

---

### 代码 5 lwIP TCP 收包回调流程

```
1  tcp_recv(pcb, Some(recv_callback));
2
3  extern "C" fn recv_callback(arg: *mut c_void, pcb: *mut tcp_pcb, p:
4      *mut pbuf, err: err_t) -> err_t {
5      if !p.is_null() {
6          let socket_inner = unsafe { &mut *(arg as *mut TcpSocketInner)
7              };
8          socket_inner.recv_queue.lock().push_back((PbuffPointer(p), 0));
9      }
10 }
```

axtask::yield\_now() 配合轮询操作实现伪阻塞行为：

---

### 代码 6 smoltcp 异步 poll 示例

```
1  loop {
2      SOCKET_SET.poll_interfaces();
3      match socket.recv(buf) {
4          Ok(len) => return Ok(len),
5          Err(AxError::WouldBlock) => yield_now(),
6          Err(e) => return Err(e),
7      }
8  }
```

此外，smoltcp 支持 poll\_connect、poll\_stream 等状态函数，结合内部状态机对连接状态进行精细化管理，如 TCP 状态迁移、握手确认、接收窗口管理等。

这种设计高度契合 Rust 的异步风格，可进一步封装为 Future 以支持 async/await 语法，提升代码清晰度与并发度。

#### 代码实现总结与优化建议

从代码实现层面可以得出如下结论：

- smoltcp 在架构上更适合安全抽象和多任务调度；lwIP 则更灵活但对状态一致性管理要求高；
- UDP 的收发处理在 smoltcp 中更简洁，但 lwIP 更具扩展性和调试性；
- 对于实时或高可靠通信，lwIP 通过强管控缓冲策略提供更稳定性能；
- 建议使用 SocketInner 枚举封装两种协议栈，并通过统一 trait 抽象提升系统适配性。

下面将进一步分析这两种协议栈在 ArceOS 中如何通过 axnet 模块实现统一集成与适配。

## 3.5 ArceOS 中的协议栈集成与适配实现

### 统一抽象层设计：axnet 接口

为实现 lwIP 与 smoltcp 的并行使用，ArceOS 定义了一层抽象接口 axnet，封装所有网络设备、地址类型、数据包缓冲区与协议栈交互逻辑。其目标是：

- 上层调用透明：应用层不感知底层协议栈实现；
- 动态切换协议栈：通过配置切换默认网络协议栈；
- 支持多实例：为多租户系统支持多个网络堆栈实例。

### lwIP 的封装适配要点

- 提供 socket 接口适配：将 lwIP 的 netconn/socket API 封装为 Rust trait 接口；
- 实现非阻塞接口：通过 select/poll 实现 pseudo async 支持；
- 启用 IPv6、DHCP、DNS、ICMPv6 模块；
- 使用独立线程轮询 lwIP 时间器（NO\_SYS=1 模式）。

### smoltcp 的集成要点

- 通过 trait Device 抽象 ArceOS 网络设备结构体；
- 将 socket 数据结构 statically 注册于 Interface；
- 使用 phy::RawSocket 模拟真实网卡接口；
- 调用 poll() 方法驱动状态机。

由于 lwIP 是一款应用广泛的轻量级 TCP/IP 协议栈，拥有完整的 TCP、UDP、ICMP、ARP 等协议支持，适用于嵌入式和资源受限环境。其设计关注内存效率和可扩展性，是多数微型操作系统默认选项。

在 ArceOS 中，我们对 lwIP 进行了如下集成和优化：

- **API 封装与接口统一：**将 lwIP 的 socket 接口封装在统一的 Rust 接口之下，对外暴露标准的 POSIX 风格操作，使上层无需感知底层实现差异；
- **协议栈模块优化：**对 lwIP 的内存分配策略进行分析，优先使用内存池进行缓冲区分配；收包采用 pbuf 引用零拷贝机制，减少内存拷贝；
- **裸机移植支持：**开启 NO\_SYS 模式，在无 RTOS 情况下通过主循环调度 lwIP 定时器，兼容 ArceOS 的无线程模型；
- **IPv6 与 DNS 支持：**通过配置 lwipopts.h 开启 IPv6 栈及 DNS 客户端功能，满足现代网络栈对多协议支持的需求；
- **调优配置参数：**对 TCP 窗口大小、最大报文段 MSS、连接池数量等参数进行调优，提升高并发环境下协议栈性能。

性能测试表明，在 Apache Benchmark 压力下，lwIP 相较 smoltcp 拥有更高的并发处理能力，在 100 并发连接下 RPS 提升达 ~12%。

由于 smoltcp 是专为嵌入式系统和 IoT 场景设计的轻量协议栈。其代码简洁、内存需求极低，且以无阻塞 API 实现，便于异步任务模型下的使用。

在 ArceOS 的适配过程中，我们主要完成以下工作：

- **设备抽象与适配**: 通过 `DeviceWrapper` 实现对底层驱动的封装，并在 `smoltcp` 中实现相应的 `phy::Device` 接口；
- **协议裁剪机制**: 通过 `Feature` 控制选择启用 TCP、UDP 等功能模块，按需裁剪协议栈体积；
- **内存集成管理**: 采用静态缓冲池与 ArceOS 的 `memory pool` 深度集成，提高内存复用效率；
- **异步轮询接口**: 提供 `poll()` 接口驱动数据包收发，适配 ArceOS 的异步调度机制。

在资源紧张或对延迟敏感的场景中，如 IoT 终端或边缘节点，`smoltcp` 的部署效果更加稳定高效。

### 多协议栈并列设计优势分析

将 lwIP 和 smoltcp 作为可选协议栈集成至 axnet 框架具有以下优势：

- **运行时灵活切换**: 系统在初始化阶段可根据用户配置或应用类型选择默认协议栈，灵活应对多样化部署场景；
- **面向场景适配**: 高负载服务器推荐使用 lwIP 以获得高吞吐量；而 IoT 等轻量场景可启用 smoltcp 降低资源占用；
- **简化上层开发**: 对上层统一暴露标准 API，协议栈变更对应用层透明，降低移植成本；
- **提升可测试性与健壮性**: 协议栈间可对比测试验证行为一致性，提升系统整体稳定性。

## 3.6 统一网络系统调用接口设计

为实现协议栈无关的套接字操作，系统定义了统一的 `syscall` 接口，并通过 `SocketInner` 枚举持有协议栈具体实现。

### 系统调用接口及功能描述

系统通过 `syscall` ID 分发实现以下主要网络操作：

- |                   |                             |
|-------------------|-----------------------------|
| • SOCKET          | • SENDTO, SENDMSG, SENDMMSG |
| • SOCKETPAIR      | • RECVFROM                  |
| • BIND            | • GETSOCKNAME, GETPEERNAME  |
| • LISTEN          | • SETSOCKOPT, GETSOCKOPT    |
| • ACCEPT, ACCEPT4 | • SHUTDOWN                  |
| • CONNECT         |                             |

---

## 代码 7 SocketInner 枚举封装协议栈实现

---

```
1 pub enum SocketInner {
2     Lwip(LwipSocket),
3     Smoltcp(SmoltcpSocket),
4 }
```

系统每次调用 `syscall()` 通过 `match` 分发到具体实现，隐藏协议栈底层细节。

### 线程安全与条件编译机制

为了支持 SMP 架构与多核并发访问，所有 socket 操作均封装在互斥锁 `Mutex` 内部，通过 Rust 安全抽象保证线程安全。协议栈选择通过 Cargo feature 标志实现裁剪与构建：

---

## 代码 8 Cargo.toml 中的 feature 定义

---

```
1 [features]
2 use-lwip = []
3 use-smoltcp = []
```

### 系统的可扩展性与未来支持

该框架的设计支持未来拓展其他网络协议栈，如 nanonet、uIP 或 Rust-native 实现，支持多租户网络栈隔离、用户态网络协议实现、或高速 NIC 的零拷贝接入等。

## 第 4 章 网络管理组件设计与实现

本章将围绕 ArceOS 系统下对网络管理组件的进一步设计与实现展开论述，旨在增强其对 Linux 应用中广泛使用的 socket 网络编程模型的支持，从而提升系统的网络兼容能力与实用性。

### 4.1 开发环境与工具配置

本章所描述的网络管理组件在 ArceOS 宏内核架构下完成，其开发环境与前述章节保持一致：以 Rust 语言为主，结合 Cargo 构建系统与模块依赖管理机制，提供组件化的开发流程。

项目支持包括 RISC-V、x86-64、AArch64 和 LoongArch64 在内的多种硬件架构，在 QEMU 模拟器平台上完成交叉编译与调试。通过构建 Makefile 脚本，支持不同架构下对网络协议栈（如 lwIP、smoltcp）的动态选择与功能测试。

为辅助网络功能的开发与验证，使用 Wireshark 进行数据包捕获分析，结合 Netcat 与 iperf3 工具模拟常见的客户端/服务端通信场景，对 socket 接口的兼容性与性能进行实际验证。

### 4.2 模块总体设计思路

网络管理组件的设计目标是提供统一的、具备 Linux 语义兼容性的 socket 抽象，支持 TCP 与 UDP 两种主要传输协议，同时保持 lwIP 与 smoltcp 协议栈的可插拔性。其核心架构设计围绕如下几个要点：

#### Socket 抽象与文件接口整合

ArceOS 将 socket 抽象统一封装为 Socket 枚举类型，内部以 Mutex 保护底层 TCP/UDP socket，确保在多线程环境下的并发访问安全。

该 Socket 类型同时实现了 FileLike 接口，与系统文件描述符机制对接，使得 socket 可被视为一种特殊的文件资源，从而统一了内核中不同 I/O 接口的调用方式。这意味着标准的文件操作（如 read/write）可以直接用于 socket，系统调用层可以使用统一入口处理来自用户空间的文件与网络请求。

此外，该设计还为未来扩展如 unix domain socket、raw socket 等提供了良好基础，仅需在枚举中增加新变体并对接口进行实现即可完成协议扩展。

#### 系统调用支持与语义还原

为满足 Linux 应用对 socket 的调用习惯，网络组件实现了如下关键系统调用接口：

- `sys_socket`, `sys_bind`, `sys_listen`, `sys_accept`,  
`sys_connect`
- `sys_send`, `sys_recv`, `sys_sendto`, `sys_recvfrom`
- `sys_getsockname`, `sys_getpeername`, `sys_setsockopt`,  
`sys_shutdown`

每个系统调用都遵循相似的处理流程：解析传参、校验合法性、调用 `Socket::from_fd()` 映射底层资源，最后执行 socket 上对应方法。部分调用（如 `sys_sendto`/`sys_recvfrom`）还需要处理地址结构与字节缓冲区的转换，并正确管理内存访问权限。

同时，通过复用 Rust 中 `Result` 类型与 `axerrno` 提供的错误码结构，网络模块能够优雅处理各类系统错误，保持与 Linux 错误语义对齐。

### 协议栈解耦与多栈选择机制

ArceOS 通过 `feature` 条件编译特性，实现对 lwIP 和 smoltcp 网络协议栈的动态替换。

在内核配置文件中用户可选择使用 lwIP 或 smoltcp，两者实现了统一的 trait 接口，使得在 socket 层操作中无需关注底层协议栈差异。

此外，在网络栈初始化过程中，系统会根据配置自动加载对应协议栈，初始化设备接口（如 TAP 网卡或 virtio 网卡），完成链路层与上层协议的绑定。

此机制的设计目标是为未来更多协议栈（如 `embedded-nal` 或 `async-tcp`）接入提供通用扩展点，实现灵活切换与无侵入式替换。

## 4.3 lwIP 协议栈适配与封装实现

### pbuf 管理机制解析

lwIP 使用 pbuf 结构作为数据包缓冲区，其支持 `PBUF_RAM`、`PBUF_ROM`、`PBUF_REF` 等多种类型，分别对应内存分配方式与引用策略。

为提升性能，ArceOS 将 pbuf 的内存管理逻辑与内核中的 slab/heap 机制结合，统一使用内存池分配，以减少碎片并提升缓存命中率。

在数据收发过程中，通过 `pbuf_chain` 构建链式结构，实现分片组包能力；释放时则统一使用 `pbuf_free` 并增加引用计数机制确保内存安全。

同时，针对 Rust 所需的内存安全与生命周期管理，引入回调钩子机制，允许在 pbuf 释放时调用预设释放器，如关闭等待队列、释放关联任务上下文等。

## 回调系统与事件驱动模型

为了实现事件驱动模型，ArceOS 为 lwIP 的回调接口增加了适配器层：

- TCP 接收事件通过 `tcp_recv` 注册回调，在数据到达时将任务唤醒
- 连接建立通过 `tcp_accept` 处理新连接，将新 `socket` 加入监听队列
- 发送完成事件通过 `tcp_sent` 通知发送缓冲区空闲，可继续写入数据
- 错误事件通过 `tcp_err` 回调报告异常，配合 `shutdown` 或自动回收资源

这些回调均结合 `WaitQueue` 与任务调度机制，在 `socket` 层实现阻塞与唤醒模型，兼顾 Linux 应用对同步 `socket` 行为的依赖。

通过事件注册与回调响应机制，lwIP 协议栈能够与 ArceOS 高效协作，完成连接建立、数据传输、异常关闭等完整连接生命周期管理。

## 4.4 smoltcp 协议栈异步接口适配

### poll 接口与非阻塞语义

smoltcp 作为轻量化协议栈，其核心设计即为事件驱动，其接口暴露的 `poll()` 方法用于主动轮询网络接口状态。

ArceOS 在实现中为每个 TCP/UDP `socket` 包装状态机，每次调用 `poll()` 返回当前 `socket` 是否可读/可写，并与 `poll/select/epoll` 系统调用的语义一致对齐。

每个 `Socket` 对象实现 `PollState` 查询接口，并在网络事件发生后通过中断或轮询唤醒对应任务线程，实现非阻塞式的数据收发机制。

### 结合任务调度的异步适配

为适配 Rust 中广泛使用的 `async/await` 模型，ArceOS 在 `socket` 层引入 `Waker` 机制。

每当 `socket` 状态变更（如缓冲区变为可读、可写），协议栈会触发注册的 `waker`，唤醒阻塞的异步任务。

`Waker` 的注册通过 `Future trait` 实现中的 `poll()` 方法完成，结合 `WakerQueue` 实现多个任务对同一 `socket` 状态的订阅与唤醒。

该机制为 `future-ready` 的协议栈如 smoltcp 提供良好支持，为未来基于 `async` 编写的高性能网络服务打下基础。

## 4.5 Socket 接口系统调用实现细节

本节给出若干典型 `socket` 系统调用在 ArceOS 中的实现过程。

#### 4.5.1 sys\_socket

用于创建新的套接字对象，根据传入的 domain、type、protocol 参数实例化对应类型的 socket，并将其加入文件描述符表。

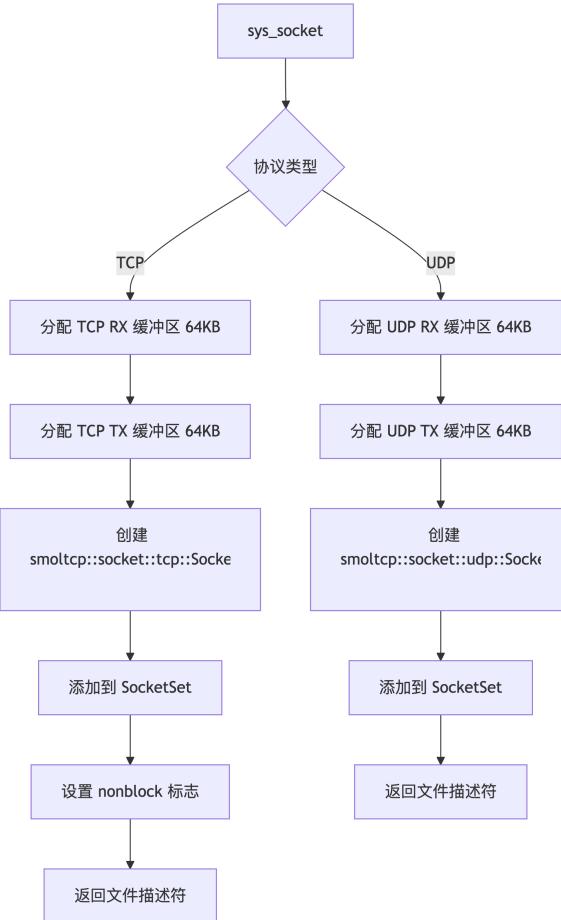


图 4.1 sys\_socket 系统调用的实现流程

#### 4.5.2 sys\_bind, sys\_connect, sys\_listen, sys\_accept

这些系统调用封装了 socket 生命周期中的典型状态转换：

- bind 将 socket 绑定至本地地址
- connect 主动连接远端主机
- listen 转为被动监听状态
- accept 从连接队列中提取 socket，返回新连接句柄

#### 4.5.3 sys\_sendto / sys\_recvfrom

这些调用支持 UDP 无连接数据报通信，允许在未连接状态下向任意地址发送数据。调用时会尝试临时 bind，确保套接字处于就绪状态。

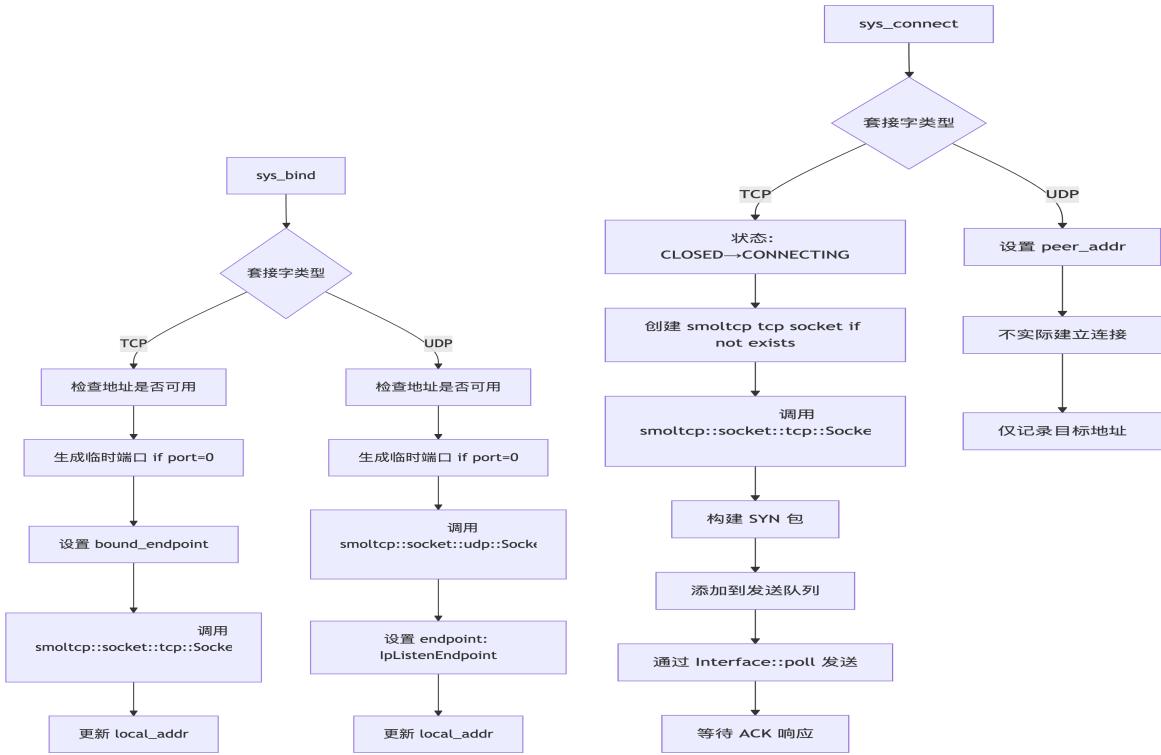


图 4.2 sys\_bind 系统调用的实现流程

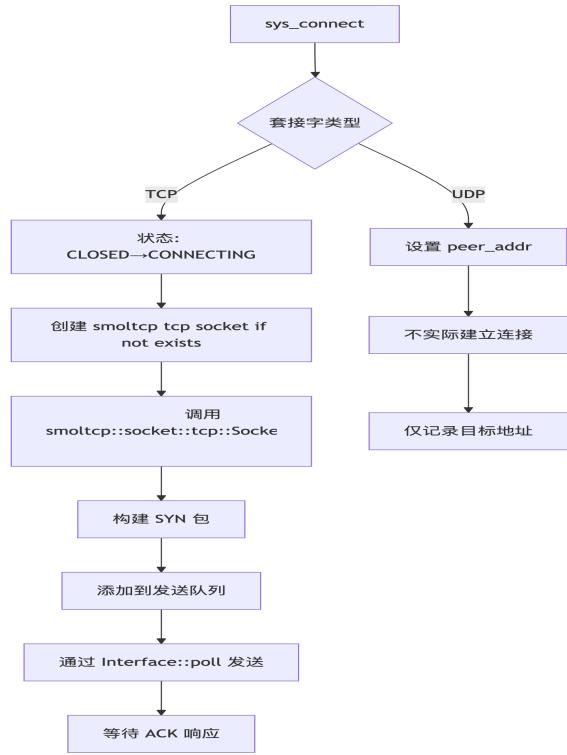


图 4.3 sys\_connect 系统调用的实现流程

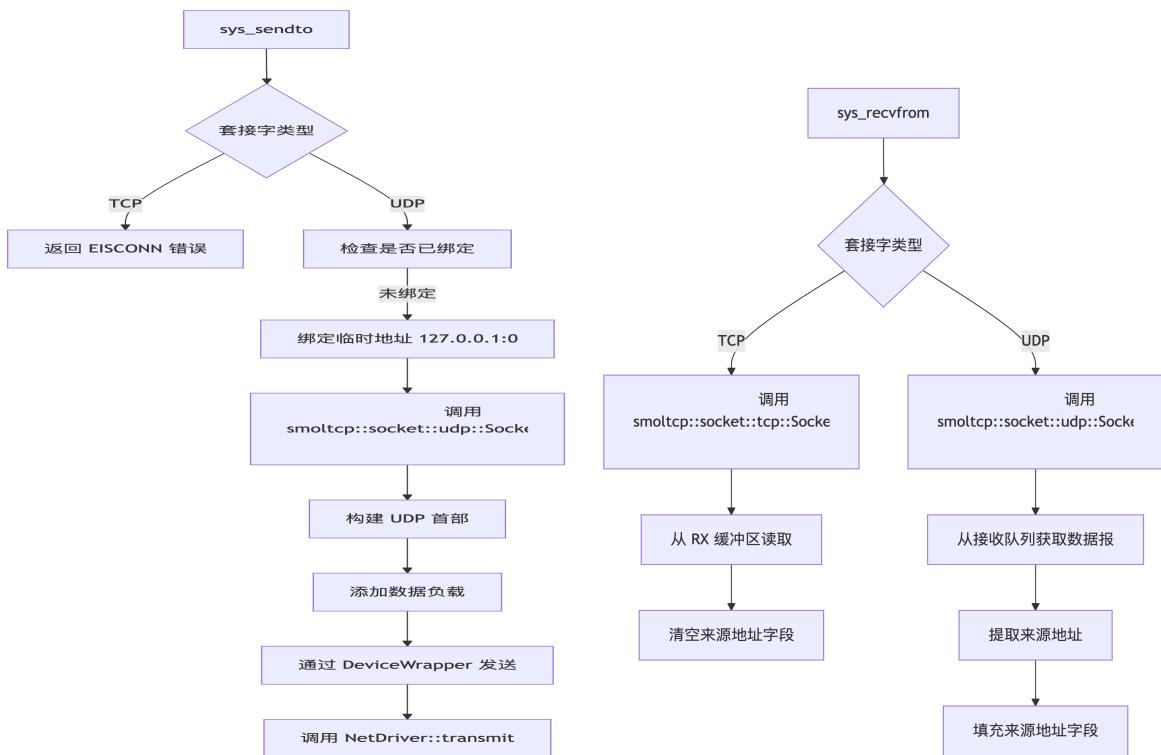


图 4.4 sys\_sendto 系统调用的实现流程

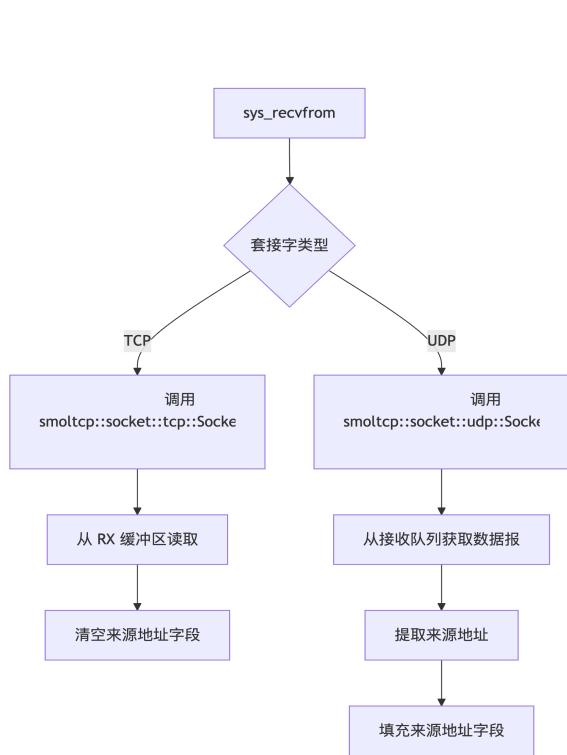


图 4.5 sys\_recvfrom 系统调用的实现流程

#### 4.5.4 sys\_setsockopt 与选项控制

用于设置 TCP/UDP 层面参数，如接收缓冲区大小、TCP\_NODELAY 等。当前版本支持基本的结构识别与参数存取，未来可扩展为完整 Linux socket option 支持集。

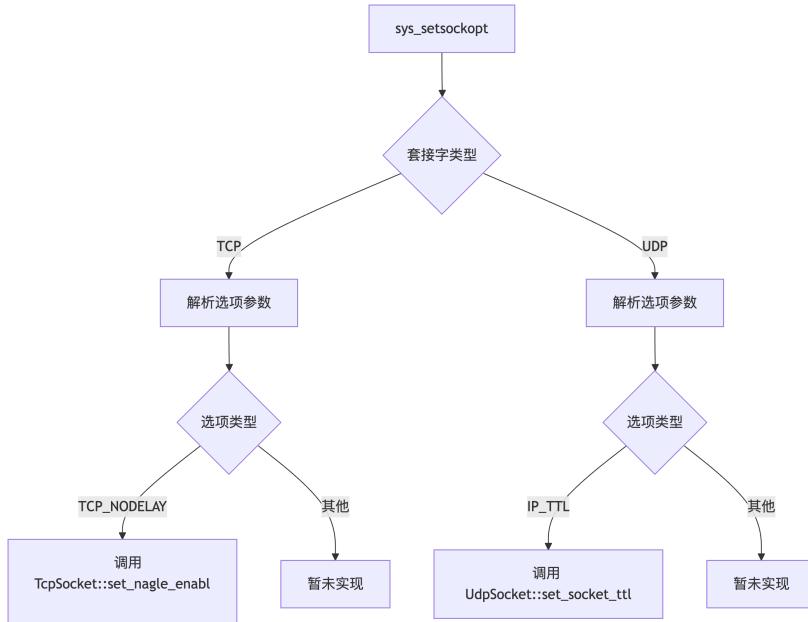


图 4.6 sys\_setsockopt 系统调用的实现流程

## 4.6 组件之间的协作与系统集成

网络组件在系统中与多个模块交互密切：

- 与任务调度组件配合实现阻塞与唤醒逻辑
- 与内存管理模块共享缓存区与内存池资源
- 与文件系统统一接口实现 read/write/stat 多态行为

具体流程见图 4.7，该图展示了 socket 创建、数据发送接收、连接管理等操作的协同工作。

## 4.7 小结

本章对 ArceOS 网络管理组件的设计与实现进行了系统性阐述，从系统调用支持、socket 接口封装、协议栈适配到异步语义整合，全面覆盖了宏内核下对网络通信子系统的构建过程。未来可进一步支持 IPv6、raw socket、TLS 加密通道等高级网络特性，完善网络子系统的功能与性能。

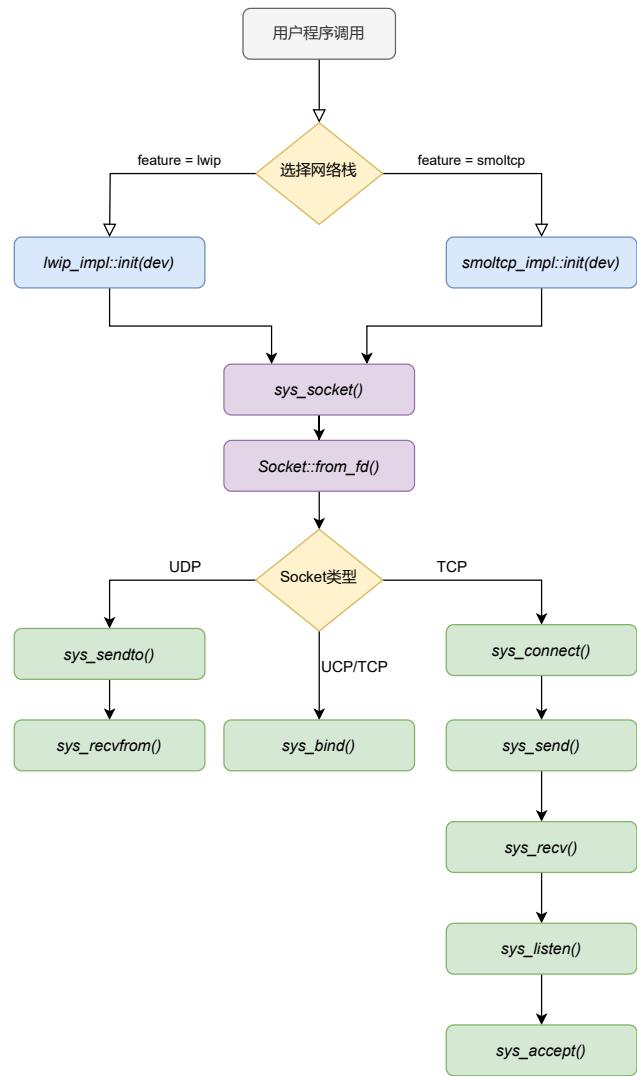


图 4.7 网络组件与系统其他模块的协作流程

## 第 5 章 网络管理组件的实验评估

为了验证本研究所设计的网络管理模块在宏内核架构下的功能完整性与性能表现，本文对其进行了系统性的测试与分析。本章将详述所采用的测试方法、用例构成与执行流程，并结合实验结果对网络组件的能力进行评估。

### 5.1 测试用例构成

本研究参考了操作系统竞赛中的标准测试体系，选取具有代表性和权威性的测试用例，以涵盖网络管理所涉及的多个关键功能点。测试用例从基础功能、系统兼容性到综合性能进行全方位覆盖，主要包括如下几类：

- **Basic 基础功能测试：**此类测试为手工编写的简单示例程序，通过系统调用直接与内核网络子系统交互。测试内容包括任务管理、文件管理、内存管理等基础操作，重点在于验证网络管理组件前置需要的其他基础模块对于系统调用的正确支持，确保基本功能的稳定性与可用性。
- **Libcetest 标准库兼容测试：**该测试集用于评估网络组件与 C 标准库中网络相关模块的交互兼容性。通过运行静态和动态的标准库测试程序，间接反映网络任务在系统中的执行正确性和资源调度效果，测试内容包括 socket 初始化、数据发送与接收、连接建立等。
- **BusyBox 命令集测试：**BusyBox 是一款集成了多种常用 UNIX 工具的轻量级软件包，广泛用于嵌入式系统和资源受限环境。本测试通过执行一系列网络相关的 BusyBox 命令（如 wget、ping 等），评估网络组件在实际指令运行时的调度能力、资源占用情况以及响应时延，从而间接测量其对网络任务的支持能力与系统兼容性。
- **iperf3：**一款被广泛使用的网络带宽测试工具，支持 TCP、UDP 以及 SCTP 协议。其主要用于测量在不同协议及参数配置下 IP 网络所能达到的最大吞吐量。iperf3 允许用户精细化地调整测试参数（如窗口大小、发送缓冲区、并发连接数等），以评估组件在不同压力下的表现。
- **netperf：**一款经典的网络性能评估工具，专注于在多种协议（如 TCP、UDP）与 socket 接口层面进行端到端的性能测量。其主要应用于分析系统的协议栈效率、数据传输延迟及处理能力，适用于评估服务器在不同任务类型下的网络处理能力。

其中，iperf3 和 netperf 是网络性能测试的核心工具，能够提供详尽的带宽、延迟

和吞吐量等指标，帮助我们全面评估 Starry-Next 网络子系统在不同负载下的表现。

iperf3 工具由美国能源部下属的 Lawrence Berkeley 国家实验室开发，具备如下特性：

- 提供详细的带宽、丢包率、时延等测量指标；
- 支持双向测试模式与 zero-copy 优化模式，可模拟极端条件下的传输性能；
- 支持 JSON 格式输出，方便集成到自动化测试系统中；
- 兼容主流类 Unix 平台（Linux、macOS、FreeBSD 等）。

在本实验中，iperf3 被用于验证网络管理组件在高负载 TCP 和 UDP 传输场景下的调度策略与资源利用率，并借助其输出的流量统计数据分析系统性能瓶颈。

netperf 支持包括：

- 测量 TCP 流吞吐量、UDP 报文处理速率；
- 模拟请求-响应型应用负载，考察网络服务的延迟敏感性；
- 可配置 payload 大小、测试持续时间、目标 CPU 核心等参数；
- 与 netserver 配合实现客户端—服务端对测。

虽然其用户界面相对简洁，但 netperf 在学术界和工业界的网络性能研究中仍被广泛采用，具有良好的可移植性和高度可定制性。在本项目中，netperf 被用于辅助分析网络栈在不同调度模式与内核状态下的性能变化趋势，作为系统调优的重要参考。

测试平台支持本地运行和远程评测两种方式。在本地测试中，研究者可通过修改 Makefile 配置，指定不同的目标架构和测试脚本，实现对组件的定向测试。而在线上环境中，借助操作系统竞赛平台提供的 CI/CD 自动化评估服务，开发者仅需提交代码仓库地址，即可触发自动构建与测试流程，系统会返回各类用例的测试报告，便于持续集成和快速验证。

为提升调试效率，在系统开发与测试过程中，广泛使用了日志机制与动态信息标注。开发者可在代码关键位置插入 `info!` 宏，并结合修改运行脚本中的日志等级配置，实现调试信息的可视化输出。通过分析日志内容，可迅速定位功能异常、调度不当或资源泄露等问题，进而提高测试反馈的效率与准确性。图 5.1 展示了测试用例的执行流程与日志输出示例。

## 5.2 通过本地编写的基础测例

在本研究过程中，为了验证网络管理组件在内核中的实际行为与系统调用实现的正确性，我在本地编写并执行了一系列针对最小功能单元的基础测例。这些测试程序聚焦于网络子系统中的核心系统调用，如 `socket`、`bind`、`connect`、`accept`、`sendto`、`recvfrom` 等，确保每个系统调用在脱离复杂上层应用的情况下也能单

```

[ 0.790696 1:11 arceos_posix_api::imp::net:298] sys_bind <= 3 0x3fffffd8 16
[ 0.792220 1:11 arceos_posix_api::imp::net:265] load sockaddr:0x3fffffd8 => 0.0.0.0
[ 0.792681 1:11 axnet::smoltcp_impl::udp:99] UDP socket #0: bound on *:49152
[ 0.793522 1:11 arceos_posix_api::imp::net:302] sys_bind => Ok(0)
[ 0.793992 1:11 starry::syscall_imp:176] Syscall bind return 0
[ 0.794499 1:11 starry::syscall_imp:59] Syscall getsockname
[ 0.794952 1:11 arceos_posix_api::imp::net:582] sys_getsockname <= 3 0x3fffffd8 0x3fffffd90
[ 0.795665 2:8 extask::api:221] idle task: waiting for IRQs...
[ 0.797988 1:11 arceos_posix_api::imp::net:238] Sockaddr: 0.0.0.0:49152
[ 0.798878 1:11 arceos_posix_api::imp::net:586] sys_getsockname => Ok(0)
[ 0.798895 3:6 extask::api:221] idle task: waiting for IRQs...
[ 0.799362 1:11 starry::syscall_imp:176] Syscall getsockname return 0
[ 0.800154 1:11 starry::syscall_imp:59] Syscall setssockopt
[ 0.795056 0:1 extask::api:221] idle task: waiting for IRQs...
[ 0.806681 1:11 arceos_posix_api::imp::net:632] sys_setssockopt <= 3 1 20 0x3fffffd8 16
[ 0.801575 1:11 arceos_posix_api::imp::net:636] sys_setssockopt => Err(EINVAL)
[ 0.802281 1:11 starry::syscall_imp:176] Syscall setssockopt return -2
[ 0.804752 1:11 starry::syscall_imp:59] Syscall write
[ 0.805194 1:11 arceos_posix_api::imp::io:54] sys_write <= 1 0x3fffffb40 140
src/functional/socket.c:24: setsockopt(s, 1, 20, &(struct timeval){.tv_sec=1, sizeof(struct timeval))==0 failed: errno = Invalid argument
[ 0.805710 0:1 extask::api:221] idle task: waiting for IRQs...
[ 0.806467 1:11 arceos_posix_api::imp::io:55] sys_write => Ok(140)
[ 0.807354 1:11 starry::syscall_imp:176] Syscall write return 140
[ 0.805810 2:8 extask::api:221] idle task: waiting for IRQs...
[ 0.807854 1:11 starry::syscall_imp:59] Syscall socket
[ 0.808560 1:11 arceos_posix_api::imp::net:273] sys_socket <= 2 2 17
[ 0.809942 3:6 extask::api:221] idle task: waiting for IRQs...
[ 0.809285 1:11 axnet::smoltcp_impl:100] socket #1: created
[ 0.809285 1:11 arceos_posix_api::imp::net:23] add_to_fd_table
[ 0.810187 1:11 arceos_posix_api::imp::net:275] sys_socket => Ok(4)
[ 0.810614 1:11 starry::syscall_imp:176] Syscall socket return 4
[ 0.811054 1:11 starry::syscall_imp:59] Syscall sendto
[ 0.811584 1:11 arceos_posix_api::imp::net:339] sys_sendto <= 4 0xb82d0 1 0 0x3fffffd8 16
[ 0.812160 1:11 arceos_posix_api::imp::net:265] load sockaddr:0x3fffffd8 => 127.0.0.1:49152
[ 0.812853 1:11 arceos_posix_api::imp::net:349] socket_fd: 4, addr: 127.0.0.1:49152, buf: [120]
[ 0.813834 1:11 axnet::smoltcp_impl::udp:211] [AxError::NotConnected] socket send() failed
[ 0.814556 1:11 arceos_posix_api::imp::net:343] sys_sendto => Err(ENOTCONN)

```

图 5.1 测试用例执行流程与日志输出示例

独运行，完成其应有的功能。

与上节中提及的集成测试不同，这些基础测例采用了“由下至上”的开发思路，针对单一系统调用接口进行验证，避免因调用链中其他未完成组件的影响而导致测试失败。例如，为测试 `accept` 系统调用的正确性，首先必须确保 `socket`、`bind` 和 `listen` 已成功实现并通过测试，因为它们构成了 TCP 服务端连接建立的前置条件。

为系统性厘清不同系统调用之间的依赖关系，我在设计测试流程前绘制了网络相关系统调用的调用依赖图。如图 5.2 所示，图中清晰地展示了各个系统调用之间的依赖关系。例如，`bind` 依赖于成功的 `socket` 创建操作；`connect` 则需要在 `socket` 成功后才能执行；而 `sendmsg` 和 `recvfrom` 则依赖于连接状态的建立。`connect` 依赖于成功的 `socket` 创建操作；`sendmsg` 和 `recvfrom` 又依赖于连接状态的建立；而更高阶的接口如 `sendmmsg` 和 `accept4` 则在基本调用之上提供批处理与额外选项支持。因此，只有在各个前置 `syscall` 成功运行的前提下，后续测试才具有实际意义。

在具体实现中，例如针对 `getsockname` 与 `getpeername` 的测试，我编写了一个简洁的客户端-服务端通信程序，客户端连接后立即获取连接对端和本地地址信息，并打印结果验证其准确性；而为了测试 `setsockopt` 和 `getsockopt`，我分别设置并读取了发送缓冲区大小与 `TCP_NODELAY` 选项，验证内核是否正确响应参数的更改。

通过这些小而精的基础测例，不仅实现了各系统调用接口的功能验证，也为后续集成测试与性能测试奠定了稳定的运行基础。这种分层、逐步推进的测试方式，有效

降低了调试难度，并提升了测试覆盖率和问题定位效率。

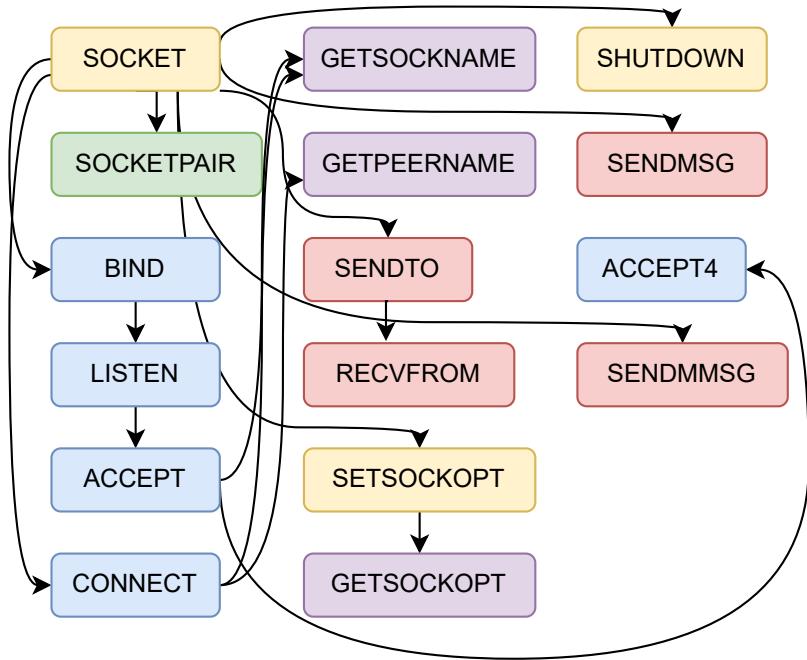


图 5.2 网络相关系统调用的依赖关系图

### 5.3 通过 libctest 测试套件验证网络功能

在本研究中，为了全面验证 Starry-Next 网络子系统的功能完整性与兼容性，我们采用了操作系统竞赛中提供的 libctest 测试套件。libctest 是一套专门为操作系统开发设计的测试框架，包含了大量针对系统调用、文件操作、网络通信等方面的标准测试用例。libctest 测试套件的核心优势在于其覆盖面广、测试用例丰富，能够有效验证网络子系统在不同协议（如 TCP、UDP）下的行为是否符合预期。通过运行 libctest 提供的网络相关测试用例，我们可以快速定位网络协议栈实现中的潜在问题，并确保其与 Linux 系统调用语义的一致性。在具体实施过程中，我们将 libctest 的 socket 相关测试套件集成到 Starry-Next 的构建流程中，并通过 QEMU 虚拟化环境运行测试。测试用例主要包括以下几个方面：

- **Socket 创建与管理：**测试套件验证了 socket 的创建、绑定、连接、监听、接受等基本操作是否符合预期，并检查套接字选项的设置与获取功能。
- **数据发送与接收：**包括对 TCP 和 UDP 套接字的数据发送（sendto、sendmsg）和接收（recvfrom、recvmsg）操作的测试，确保数据传输的正确性与可靠性。
- **错误处理与异常情况：**测试套件还覆盖了各种异常情况的处理，如无效参数、资

源不足等，确保系统在异常情况下能够正确返回错误码。

测试用例的测试流程如下图 5.3 所示。libctest 测试框架通过模拟各种网络操作场景，并对每个系统调用的返回值和行为进行验证：

- **UDP 服务端 (s):** 绑定本地地址、设置接收超时、等待接收
- **UDP 客户端 (c):** 向 s 指定地址发送数据”x”
- **TCP 服务端 (s):** 创建监听 socket，检查 CLOEXEC，接收连接
- **TCP 客户端 (c):** 创建非阻塞 socket，发起连接请求
- **TCP 连接 (t):** 服务端接收到的连接 socket

libctest 框架会自动比较实际输出与预期结果，生成详细的测试报告。通过运行 libctest 测试套件，我们发现 Starry-Next 网络子系统在大部分测试用例中表现良好，能够正确处理各种网络操作，并与 Linux 系统调用语义保持一致。然而，在某些特定情况下，如高并发连接或异常数据包处理时，仍存在一些边界情况需要进一步优化和调整。

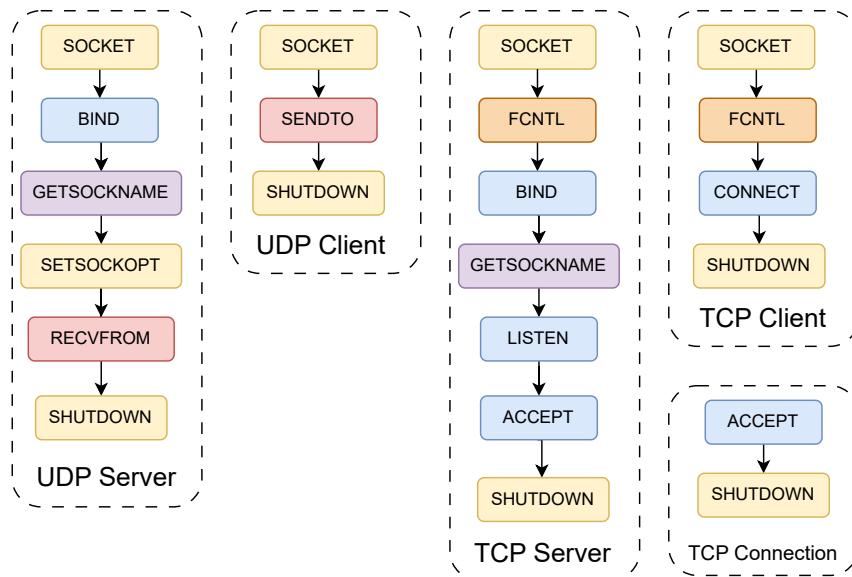


图 5.3 libctest 测试套件的执行流程

### 5.3.1 测试结果分析

本项目对网络子系统中的核心系统调用进行了全面的功能测试，涵盖了套接字创建、连接建立、数据收发、参数配置及连接关闭等典型操作路径。测试基于本地自编最小测例、libctest 测试框架、以及多种网络压力测试工具，采用逐层递进的测试策略，确保每一个系统调用都能在其最小功能上下文中独立运行，并返回预期结果。

为量化测试覆盖范围，我们对 `testsuits-for-oskernel1` 测试集中所有网络相关系统调用的调用频率进行了统计与分析。图 5.4 展示了各类系统调用在测试过

程中被触发的次数。可以看出，`socket`、`bind`、`connect`、`accept` 等基础功能在多种测试场景中均被频繁调用，调用量达到千次量级，说明这些基础网络功能已得到了充分验证与覆盖。

相较之下，一些扩展接口如 `accept4`、`sendmmsg` 和 `sendmsg` 的调用次数相对较少，主要原因在于这类系统调用通常用于高性能或特定功能优化场景，尚未在主路径测试用例中广泛引入。未来可以考虑针对这些接口设计更具针对性的测例，以进一步提升测试的完整性与覆盖率。

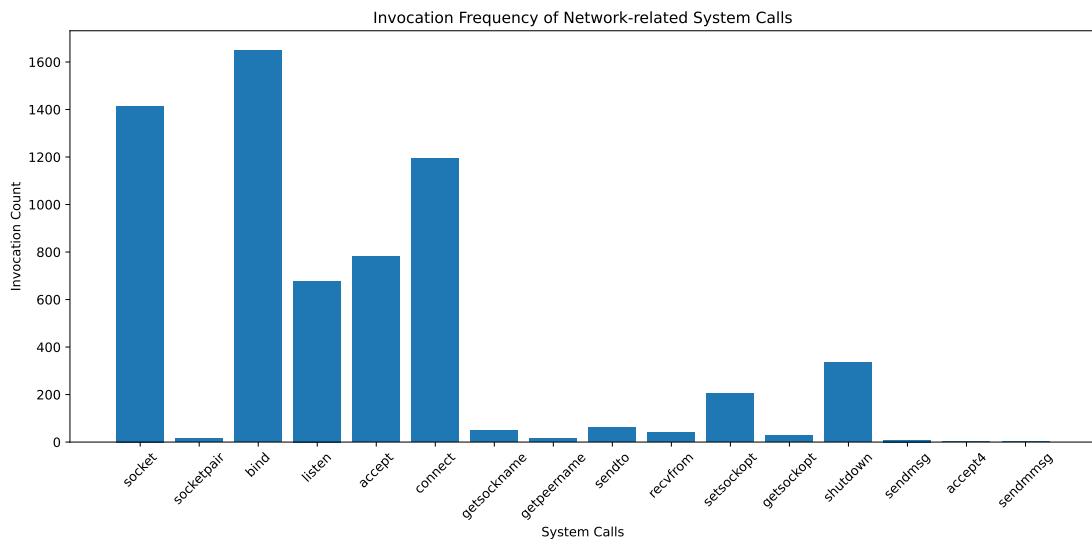


图 5.4 各网络系统调用在测试中的调用次数统计

此外，完整统计数据与生成脚本已随本项目一并开源，读者可参考附录或项目仓库获取更详细的信息。

表 5.1 列出了本项目中已覆盖的网络相关系统调用及其测试方式。目前，所有列出的系统调用均已通过本地测例的功能验证，测试结果稳定，未出现异常行为，表明网络管理组件的基本功能已具备良好的可用性与稳定性。

表 5.1 网络系统调用功能测试覆盖情况

系统调用	测试方式与验证要点
socket	创建 TCP/UDP 套接字，验证返回值与类型一致性
bind	套接字绑定地址，检查绑定状态与端口分配
listen	TCP 套接字监听状态设定，结合 accept 联合验证
accept	接收连接请求，测试连接 socket 的创建与有效性
connect	客户端主动连接服务端地址，验证连接状态码
getsockname	获取本地绑定地址，验证端口、地址格式正确
getpeername	获取对端地址信息，测试对已连接套接字的读取
sendto	无连接 UDP 发送测试，配合 recvfrom 验证数据一致性
recvfrom	接收指定地址数据，验证数据长度与内容正确性
sendmsg	发送带有控制信息的消息，验证传输结构支持
setsockopt	设置套接字参数（如超时），读取验证生效
getsockopt	获取套接字当前参数，配合 setsockopt 双向验证
shutdown	优雅关闭连接，测试关闭后 socket 行为符合预期

## 第6章 遇到的问题与解决方案

### 6.1 网络协议栈适配中的兼容性问题

在ArceOS中实现lwIP和smoltcp两款轻量级网络协议栈时，兼容性问题是面临的主要挑战之一。由于这两款协议栈在功能实现和内存管理上的差异，如何使其能够无缝集成到ArceOS中，并与其他模块（如进程管理和内存管理）顺利协作，是设计过程中必须解决的问题。网络协议栈的高效运行对操作系统的性能至关重要，因此在集成过程中我们需要特别关注内存管理、数据包缓冲区、连接管理等多个方面。

#### lwIP协议栈的内存管理适配

lwIP协议栈原本设计为嵌入式系统中的高效网络协议栈，但其内存管理机制与ArceOS的内存管理系统存在差异。lwIP使用了自有的内存池管理方式，而ArceOS则采用了基于页表的虚拟内存管理机制。为了确保lwIP在ArceOS中的高效运行，我们需要对其内存池管理和数据包缓冲区的使用进行定制化调整。

具体来说，lwIP需要更多地与ArceOS的内存分配系统进行整合，以减少内存碎片，保证数据传输的稳定性和高效性。在高并发场景下，lwIP的内存管理机制往往会导致内存分配过于分散，进而影响网络传输性能。为了弥补这一点，我们对lwIP进行了内存池的优化，使得内存池能够与ArceOS的内存管理机制进行更紧密的对接，避免了内存碎片化的发生，并提升了数据包传输的稳定性。

#### 协议栈选择与切换机制

为了提供灵活的网络配置，ArceOS允许用户选择在lwIP和smoltcp之间切换。尽管这为ArceOS提供了更大的灵活性，但也带来了一些适配问题。最显著的问题是协议栈切换时，如何保持正在进行的网络连接不中断，如何管理两个协议栈间的内存资源共享，成为了必须解决的挑战。

为了简化这一过程，我们设计了一个统一的网络接口层，能够根据系统配置动态加载和切换协议栈。通过这种设计，用户只需要配置协议栈的选择，系统会自动管理内存资源的分配，并确保协议栈切换时不会影响正在进行的网络通信。这个机制大大减少了手动切换和配置的复杂性，并简化了用户的操作流程。

### 6.2 网络功能的测试问题

在网络功能测试的过程中，我们不仅验证了各个系统调用的基础行为，还在实际测试中遇到了一些实现层面的复杂问题，下面列举了若干具有代表性的问题及其分

析过程。

#### **accept 函数在非阻塞连接下的异常行为**

在测试非阻塞套接字时，我们发现客户端使用 SOCK\_NONBLOCK 进行连接请求后，服务端调用 accept 可能无法及时返回连接结果，导致程序长时间阻塞或误判连接失败。问题的根源在于 connect 调用返回 EINPROGRESS 时，服务端的监听状态未能同步更新，accept 队列未能及时插入新连接。我们通过引入状态轮询机制以及改进连接完成的通知流程，成功解决了该问题，使非阻塞连接的建立更加稳定。

#### **getsockname 返回信息异常**

在对 getsockname 系统调用进行测试时，发现部分情况下返回的地址信息中端口号为 0。经过排查，发现该问题出现在未显式指定端口号进行 bind 的场景下，系统未能为套接字自动分配可用端口。我们通过调整 bind 的实现逻辑，确保在端口为 0 的情况下内核自动分配可用端口，并正确更新地址结构，最终使 getsockname 能够返回完整的本地地址信息。

#### **socket 标志位未生效问题**

在测试 SOCK\_CLOEXEC 与 SOCK\_NONBLOCK 标志时，通过 fcntl 检查文件描述符状态，发现部分系统调用返回的 socket 未设置对应标志位。这是由于套接字创建与文件描述符标志设置之间未正确衔接所致。我们通过在 socket 创建路径中统一添加标志传递与设定逻辑，确保在 socket 或 accept4 创建套接字时，相关标志能准确反映在描述符状态中。

#### **recvfrom 接收数据异常**

在 UDP 通信测试中，虽然 recvfrom 调用返回值为 1，表示成功接收到一个字节数据，但在验证数据内容时发现 buffer 中内容为空或被破坏。进一步排查发现问题出现在缓冲区未正确初始化或长度未设置的问题上。我们通过加强缓冲区管理逻辑、初始化结构体内容，并在 recvfrom 前明确指定 socklen\_t 长度字段，确保数据能准确写入并被用户程序读取。

这些问题的解决为网络子系统功能测试的稳定性提供了保障，也进一步暴露了不同系统调用之间在实现逻辑上的紧密依赖关系，为我们后续进行系统级测试提供了宝贵经验。

### **6.3 网络模块的相关代码在开发中的问题**

在开发过程中，遇到了以下几个实际问题，并通过以下方式解决：

### 6.3.1 recvfrom 函数问题

在开发过程中，发现 `recvfrom` 函数未能在 `socket_set` 中找到非阻塞的套接字（nonblocking socket）。此问题的根本原因在于 `socket` 的 `canrecv()` 方法总是返回 `false`，导致套接字的接收缓冲区（`rxbuffer`）始终为空。我们进一步调查发现，这与 UDP 套接字在创建时没有正确实现回环（loopback）机制有关，更加具体直观的错误情况如图 6.1 所示。解决方法是为 UDP 套接字添加回环支持，确保即使在没有数据的情况下，接收操作能够顺利完成。

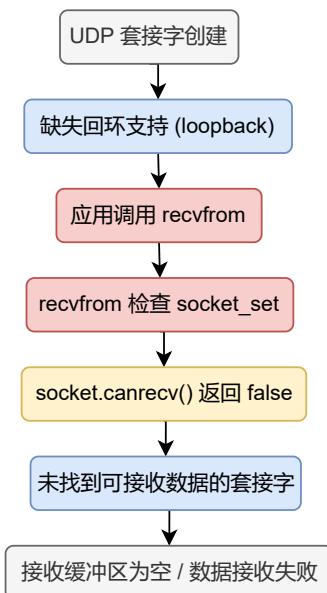


图 6.1 `recvfrom` 函数未能找到非阻塞套接字的问题示意图

### 6.3.2 sendto 问题

在进行 `sendto` 系统调用的开发和测试时，我们遇到了 `self_addr_load failed` 的错误。通过检查代码，我们发现这个问题是由于 `self_addr` 为空引起的。根据分析，ArceOS 中的 `sendto` 调用是依赖于 `socket_self_addr` 是否存在的，但对于未显式绑定的套接字，系统应自动分配本地地址，更加具体直观的错误情况如图 6.2 所示。我们通过修改代码，使得在没有绑定操作的情况下，系统自动为 UDP 套接字绑定 `0.0.0.0:0`，从而解决了这个问题。

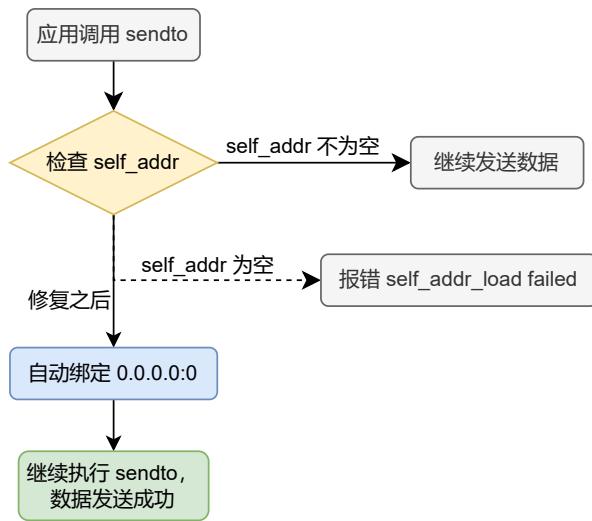


图 6.2 sendto 系统调用中 self\_addr 为空的问题示意图

### 6.3.3 setsockopt 问题

在调试过程中，我们发现 setsockopt 系统调用在 ArceOS 中未能正确工作，更加具体直观的错误情况如图 6.3 所示。为了解决这一问题，我们查看了其他操作系统的实现，并参考了星绽 OS 和 DragonOS 的处理方式，选择在 setsockopt 调用中进行伪实现，直接返回 Ok(0)。然而，这种做法仅适用于开发阶段，在生产环境中需要进一步的实现和优化。我们计划在后续版本中完善这一功能，并确保 setsockopt 可以支持更多选项。

### 6.3.4 iperf 测试中的 IPv6 支持问题

在使用 iperf 工具对 ArceOS 网络子系统进行性能测试时，遇到了 IPv6 协议栈支持不足的问题。由于 iperf 默认会尝试初始化 IPv6 套接字（即 AF\_INET6），并构造 28 字节的 sockaddr\_in6 地址结构用于 TCP 建连测试，这与当前系统只支持 IPv4（16 字节 sockaddr\_in）的实现产生了不兼容。

具体地，当 iperf 调用 socket(AF\_INET6, SOCK\_STREAM, 0) 初始化 TCP 套接字时，系统返回 EINVAL 错误，提示地址族不支持。进一步追踪发现，当前 ArceOS 的 socket 系统调用仅对 AF\_INET 进行了路径实现，未包含对 AF\_INET6 的任何处理逻辑。此外，与之关联的 bind、connect、recvfrom 等调用中，对 sockaddr\_in6 类型的字段解析也未做适配，从而导致一系列错误发生。

为了解决该问题，我们从以下几个方面入手：

- **添加 IPv6 地址族支持：**扩展 socket 系统调用，使其识别 AF\_INET6 类型，

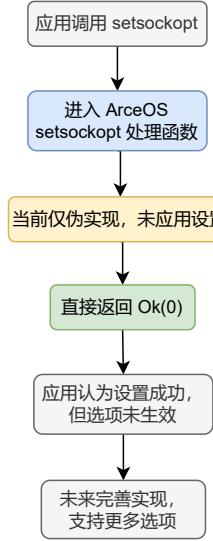


图 6.3 setsockopt 系统调用未正确工作的示意图

并在内部建立相应的套接字类型（如 `SocketDomain::Inet6`）。

- **实现 `sockaddr_in6` 的解析逻辑:** 在 `bind` 和 `connect` 中增加对 28 字节 IPv6 地址结构的解析支持，包括 `sin6_addr`（16 字节 IPv6 地址）、`sin6_port` 和 `sin6_scope_id` 等字段的正确处理。
- **未来改进方向:** 尽管上述修改使 `iperf` 在测试中能够继续工作，但 IPv6 支持仍需系统性地增强，包括支持多播地址、链路本地地址、ICMPv6 报文以及 Neighbor Discovery 等协议。我们计划在后续版本中完成完整的 IPv6 协议栈集成，以支持更广泛的网络测试场景。

## 第7章 总结与展望

### 7.1 本文工作总结

本论文深入研究了 ArceOS 操作系统中的网络管理模块，特别是 lwIP 和 smoltcp 协议栈的集成与适配。通过实现这两个协议栈的并行支持，我们提高了 ArceOS 在网络管理方面的灵活性和可定制性，能够根据不同的硬件平台和应用需求动态选择协议栈。此外，本论文还着重解决了网络协议栈的内存管理、协议栈切换机制等关键问题，并通过对实验验证了设计的有效性。本研究的相关源代码已开源，托管于 GitHub，详见：<https://github.com/LearningOS/oscomp-test-LIN-Matrix/>。

#### 7.1.1 主要贡献

本论文的主要贡献包括以下几个方面：

- **设计并实现了 ArceOS 网络管理模块中的 lwIP 和 smoltcp 协议栈，并通过统一的网络接口进行对接：**我们成功地在 ArceOS 中集成了 lwIP 和 smoltcp 两款轻量级协议栈，为操作系统提供了灵活的网络协议栈选择。通过这一设计，用户可以根据硬件资源和应用需求动态选择适合的协议栈，实现了网络管理的高度可定制性。
- **解决了网络协议栈与内存管理、系统调用接口等模块的耦合问题，优化了协议栈切换机制：**在集成 lwIP 和 smoltcp 的过程中，我们克服了协议栈和系统调用层、内存管理层之间的强耦合问题。通过模块化设计和抽象接口的实现，降低了各个模块之间的耦合度，提高了系统的灵活性和可扩展性。特别是在协议栈切换时，采用了统一的接口层，使得协议栈切换变得更加平滑和高效。
- **通过 QEMU 虚拟平台进行了系统功能和性能测试，验证了 ArceOS 的网络管理模块在高并发和低延迟场景下的表现：**为了验证 ArceOS 网络管理模块的功能和性能，我们采用了 QEMU 虚拟平台进行测试。实验结果表明，ArceOS 在高并发和低延迟应用场景下能够保持良好的性能，特别是在资源受限的环境中，lwIP 和 smoltcp 协议栈的表现都达到了预期。

### 7.2 存在的不足

尽管我们在 ArceOS 中实现了灵活的网络协议栈选择和高效的网络传输，但仍然存在一些不足之处。以下是当前实现的主要不足以及需要进一步改进的地方：

## 网络性能的进一步优化

尽管我们已经对 lwIP 和 smoltcp 协议栈进行了多次优化，但在一些高负载场景下，lwIP 协议栈的性能仍然存在瓶颈。特别是在数据传输延迟较高的情况下，lwIP 协议栈的性能表现不如预期。我们已经通过优化内存管理和数据传输路径做了一些改进，但在处理大量并发连接时，仍然需要进一步提升性能，尤其是在需要高吞吐量和低延迟的应用场景下，性能提升尤为重要。

## 协议栈兼容性问题

尽管 lwIP 和 smoltcp 已经能够兼容大部分的网络应用，但在某些特殊的网络应用中，仍然存在协议栈无法完全兼容的情况。特别是在一些高频率的数据传输操作中，协议栈的适应性问题仍然存在。例如，某些特殊的应用程序可能依赖于某些协议栈的特定实现方式，而我们当前的协议栈适配可能不能完全满足所有的使用需求。为了解决这个问题，未来我们需要对协议栈进行进一步的扩展和优化，使其能够适应更多类型的应用场景。

## 系统调用接口的扩展

目前 ArceOS 已经支持了大部分基础的网络系统调用，但对于一些复杂的网络操作，如高效的多线程网络通信、流量控制等，仍然缺乏足够的系统调用支持。未来，ArceOS 需要增加更多高级系统调用的支持，如动态流量管理、多线程支持等，尤其是在高负载的环境中，流量控制和高效的进程管理将是关键要素。

## 7.3 后续研究方向

未来的研究可以从以下几个方面进一步展开，针对目前存在的不足进行优化和拓展：

### 网络协议栈性能优化

网络协议栈的性能优化是一个长期的任务，特别是在高并发、高负载环境下的性能优化。我们可以通过深入分析协议栈在不同场景下的瓶颈，进一步优化内存分配、数据包处理等关键路径，提高协议栈的吞吐量和响应速度。例如，对于小数据包传输和低延迟应用场景，进一步优化协议栈中的延迟处理，降低数据包的传输延迟，将是未来的研究重点。

### 协议栈扩展

除了 lwIP 和 smoltcp，未来可以集成更多轻量级协议栈，如 uIP、Nettalk 等，以便更好地适应不同的硬件平台和应用需求。通过支持更多的协议栈，ArceOS 可以在不同的嵌入式系统和物联网设备中提供更高效、更灵活的网络支持。不同协议栈之间的兼容性和互操作性也是未来研究的一个重要方向。

### **更丰富的网络功能支持**

在协议栈的基础上，增加更多的网络功能支持，将进一步提高 ArceOS 的应用场景适应性。例如，我们可以在协议栈中实现 QoS (Quality of Service) 支持，保证不同网络流量的优先级，优化数据传输的效率。此外，流量控制、网络安全、加密等高级功能将成为 ArceOS 网络协议栈的必备功能，能够满足现代网络应用的需求。

### **网络管理的自动化和智能化**

随着物联网和云计算的发展，网络管理将越来越复杂。未来可以研究更智能化的网络管理策略，通过机器学习等技术实现自动化网络流量优化和故障检测，提高系统的自适应能力。例如，机器学习可以用于自动调节流量策略、预测网络拥塞并采取相应的防护措施，从而提高网络的可靠性和自愈能力。

## **7.4 总结**

通过本论文的研究，我们不仅成功地在 ArceOS 中实现了灵活可定制的网络协议栈支持，还解决了网络协议栈与操作系统其他模块之间的耦合问题，优化了协议栈的切换机制。通过在 QEMU 平台上的测试和评估，我们验证了 ArceOS 在高并发和低延迟应用场景下的表现，并证明了我们的设计思路和优化方法是有效的。

尽管目前还存在一些网络性能和协议栈兼容性方面的不足，但我们相信，随着进一步的优化和扩展，ArceOS 将能够提供更加高效、灵活和可扩展的网络管理能力。未来的研究将集中在协议栈的性能优化、系统调用接口扩展以及网络功能的智能化和自动化等方面，推动 ArceOS 在更多领域的应用。

## 参考文献

- [1] Ammons G, Appavoo J, Butrico M, et al. Libra: a library operating system for a jvm in a virtualized execution environment[C/OL]//VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2007: 44–54. <https://doi.org/10.1145/1254810.1254817>.
- [2] Peter S, Li J, Zhang I, et al. Arrakis: The operating system is the control plane[J/OL]. ACM Trans. Comput. Syst., 2015, 33(4). <https://doi.org/10.1145/2812806>.
- [3] Black D, Golub D, Julin D, et al. Microkernel operating system architecture and mach[J]. Journal of information processing, 1991, 14(4): 442-453.
- [4] Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: library operating systems for the cloud [C/OL]//ASPLOS '13: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013: 461–472. <https://doi.org/10.1145/2451116.2451167>.
- [5] Popek G J, Goldberg R P. Formal requirements for virtualizable third generation architectures [J/OL]. Commun. ACM, 1974, 17(7): 412–421. <https://doi.org/10.1145/361011.361073>.
- [6] Abdalkarim B A, Akeun D. Analysis of hybrid kernel-based operating systems[C]//Proceedings of a relevant conference. 2022.
- [7] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new os architecture for scalable multicore systems[C/OL]//SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2009: 29–44. <https://doi.org/10.1145/1629575.1629579>.
- [8] Engler D R, Kaashoek M F, O'Toole J. Exokernel: an operating system architecture for application-level resource management[C/OL]//SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1995: 251–266. <https://doi.org/10.1145/224056.224076>.
- [9] Fassino J, Stefani J, Lawall J L, et al. Think: A software framework for component-based operating system kernels[C/OL]//Proceedings of the General Track: 2002 USENIX Annual Technical Conference. USENIX, 2002: 73-86. <http://www.usenix.org/publications/library/proceedings/usenix02/fassino.html>.
- [10] 古金字, 李浩, 夏虞斌, 等. BrickOS: 面向异构硬件资源的积木式内核[J]. 中国科学 (信息科学), 2024, 54(3): 491-513.

## 附录 A 外文资料的书面翻译

An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise

### 目录

A.1 摘要 .....	60
A.2 引言 .....	61
A.3 Rust 在 Linux 中的基础原理 .....	63
A.3.1 Rust 的安全模型 .....	63
A.3.2 unsafe 关键字 .....	64
A.3.3 Rust 与 Linux 的集成方式 .....	64
A.3.4 RFL 的目标 .....	65
A.4 RQ1: RFL 的现状如何? .....	65
A.4.1 RFL 开发现状 .....	65
A.4.2 利用安全抽象实现 Linux 内核的 Rust 化 .....	67
A.4.3 使用 Rust 编写设备驱动 .....	71
A.5 RQ2: Rust-for-Linux 是否达到了预期? .....	72
A.5.1 RFL 提升了 Linux 的可保障性 (securability) .....	72
A.5.2 RFL 是否引入额外开销? .....	74
A.5.3 Rust 如何改善 Linux 的开发? .....	76
A.6 RQ3: 有哪些经验与教训? .....	79
A.7 相关工作 .....	81
A.8 结论 .....	82
参考文献 .....	82
A.A 案例研究: 内核空间中 Rust 编程的灵活性问题 .....	88
A.B 带调试信息的驱动体积开销 .....	89
A.C 内核社区对 Rust 的看法 .....	89

### A.1 摘要

Linux 发展已逾三十年，早已成为现代数字世界的计算基础：无论是庞大复杂的主机（如超级计算机），还是廉价轻量的嵌入式设备（如物联网设备），无数应用都构

建于其之上。然而，这一基础设施自诞生之日起便一直受到内存与并发错误的困扰，其根源在于 C 语言对内存操作的宽松约束，允许大量“野蛮”的访问行为。

近期项目 Rust-for-Linux (RFL) 有望一劳永逸地缓解 Linux 内核中的安全问题。RFL 通过将 Rust 的静态所有权机制与类型检查引入内核代码，使得内核在不牺牲性能的前提下，有可能摆脱内存与并发错误的困扰。尽管 RFL 正在逐步成熟并被并入 Linux 主线内核，其安全性与性能是否能够真正兼得仍未得到系统性研究。

为此，本文开展了对 RFL 的首次实证研究，旨在揭示其发展现状与带来的益处，特别关注 Rust 如何与 Linux 内核融合，以及该融合是否能在不增加系统开销的前提下保障驱动程序的安全性。我们收集并分析了六个关键的 RFL 驱动，涵盖数百个 issue 与 PR、数千次 GitHub 提交与邮件往来，以及超 1.2 万条 Zulip 论坛讨论。

研究发现：虽然 Rust 在一定程度上缓解了内核中的安全漏洞，但其能力仍不足以彻底根除此类问题。更重要的是，如果使用不当，Rust 所提供的安全保障反而可能带来显著的运行时开销与开发负担。

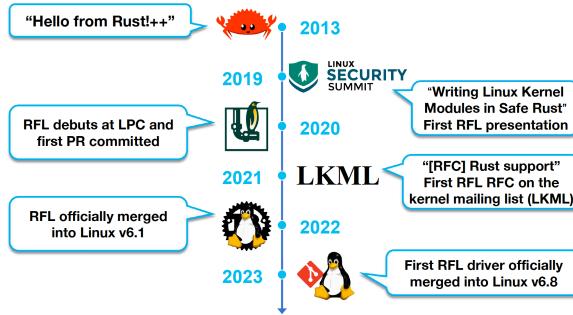
## A.2 引言

作为当今计算基础设施的事实标准，Linux 从未停止对长期困扰系统软件的内存和并发错误的消除<sup>[1-4]</sup>。尽管社区投入大量安全加固与工程努力，漏洞仍不断涌现<sup>[5-8]</sup>。究其根源，主要在于 C 语言允许不受限制地访问内存对象，Linux 内核也依赖野指针转换、裸指针操作等机制构建复杂抽象层与通用框架（如设备驱动<sup>[9]</sup>），以追求模块化和性能。

如何在不牺牲性能的前提下保障内存安全？Rust 语言被视为一个有前途的解决方案，有望根本缓解上述问题<sup>[10]</sup>。作为一种新兴的、静态且强类型的系统编程语言，Rust 承诺在无运行时开销的前提下实现安全与高性能<sup>[11]</sup>。其背后依托的是所有权机制<sup>[12]</sup>，用于根除内存与并发错误，其核心规则包括：1）每个内存位置在任意时刻只能被一个变量独占；2）所有权可通过移动或引用的方式在线程内或线程间（通过 Send 和 Sync traits）传递；3）所有者离开作用域时，对应内存自动释放。Rust 依赖静态检查在编译期强制上述规则，从而无需使用传统的重量级内存检查器<sup>[13-14]</sup>或垃圾回收器<sup>[15-16]</sup>，避免了运行时中断和不确定性延迟。

**Rust for Linux 项目：**正是由于 Rust 上述的安全属性，2013 年一项名为 Rust-for-Linux (RFL) 的尝试性项目应运而生<sup>[17]</sup>，如图 A.1 所示。它首次构建了一个基于内核头文件的 Rust 目标文件，并在可加载模块中调用了一个 Rust 函数输出“Hello from Rust!++”，标志着 Rust 第一次在 Linux 内核空间中发声。最初，Rust 只是 Linux 的一

个“边缘工具”，主要用于编写安全关键、独立且简短的内核模块函数。2019年，写整个内核模块的Rust提案开始出现<sup>[18-19]</sup>，该提案大胆地在上游内核中引入了对内核接口和数据结构的Rust封装层。2020年的Linux Plumbers Conference（LPC）上，该想法首次提出被广泛支持<sup>[20]</sup>，并促成了2021年首个RFL RFC发布<sup>[10]</sup>。此后，RFL项目迅速发展，通过Git仓库管理代码，引入持续集成（CI）机制确保补丁质量，交流平台也从传统邮件列表转向更活跃的Zulip社区。2022年，RFL作为实验性功能正式并入Linux v6.1主线版本<sup>[21]</sup>。



图A.1 RFL发展历程中的重要里程碑。

RFL推动了各种Rust驱动的开发，包括网络<sup>[22]</sup>、块设备<sup>[23-24]</sup>、文件系统<sup>[25-26]</sup>、Android<sup>[27]</sup>与GPU驱动<sup>[28]</sup>。其中，一个网络驱动<sup>[29]</sup>经过11轮联合审查后成功并入主线Linux v6.8，意味着RFL已从实验阶段逐步进入真实使用场景。尽管尚处于早期阶段，RFL已成为最活跃的内核子系统之一<sup>[30]</sup>，可与eBPF<sup>[31]</sup>和io\_uring<sup>[32]</sup>并驾齐驱。

**研究动机：**尽管Rust语言特性已经被广泛研究<sup>[33-35]</sup>，Rust与Linux的结合（即RFL）仍鲜有系统性分析。理解其融合过程对于指导安全高效驱动开发具有重要意义，也为将Rust应用于更多内核子系统提供参考。

此外，RFL提供了观察新语言如何融入大型传统代码库的独特视角。这一过程不仅包括将内核库封装为Rust crate，也涉及语法语义的适配（如将C风格的宏泛型范式翻译为Rust风格），还包括将Rust安全规则融入对执行内存更宽松的内核环境中，以及引入新协作模式和工具链，逐步影响Linux的开发流程。该过程融合了成功、争议与妥协。在2022年以来已有超2万行RFL代码合入主线背景下，我们认为此时对其发展历程进行回顾具有现实意义。

因此，本文旨在开展对RFL融合过程的首次系统研究。我们收集了RFL项目的6个驱动、269个Issue、763个PR、1540次提交、3611条邮件与12501条论坛讨论内容。

**研究问题：**本文将围绕以下三个关键研究问题（RQ）展开：

- **RQ1：RFL 的现状如何？**我们分析现有数据并深入驱动实现，发现尽管 RFL 的构建系统（如 Kbuild）已趋成熟，但缺乏主流驱动和文件系统，核心瓶颈在于代码审查进展缓慢。此外，尽管 Rust Traits 减轻了手动审计负担，但并非保障内存安全的“银弹”，Rust 与内核在内存操作的语义差异带来复杂的绕行方案，造成运行与开发开销。
- **RQ2：RFL 是否达成其初衷？**我们重评社区最初设定的三个目标：更安全、性能损失小、开发更容易。实测五个 RFL 驱动发现，Rust 提高了安全“可达性”但非绝对安全，unsafe 是驱动开发不可避免的选择。同时，Traits 与智能指针引发大量 icache miss，性能在部分场景不及 C；但 Rust 显著提升了代码质量，吸引了非 C 背景开发者。
- **RQ3：从 RFL 中可获得哪些经验？**我们提出了开发者应谨慎对待 Rust 安全保障的建议，并为社区提供一个评估“Rust 化”子系统价值的模型：通过对安全收益与开发代价的权衡评估，优先推荐将 ext4 和 linux-block 等 7 个子系统及其 25 个驱动作为推进重点。

**贡献：**相较于现有研究关注 Rust 安全与性能<sup>[36-37]</sup>，本文首次全面剖析 Rust 与 Linux 的融合过程，贡献如下：

- 构建包含 1540 次提交、269 个 Issue、763 个 PR 与 3611 封邮件的数据集，并将在公开发表后开源。
- 开发自动化工具集，实现 API 统计分析与子系统视角的结果展示。
- 提出对 RFL 当前发展状态的深入剖析，总结将 Rust 融入内核的主要挑战，并提供针对未来驱动开发的实用建议。

完整数据与工具已发布于：[https://github.com/Richardhongyu/rfl\\_empirical\\_tools](https://github.com/Richardhongyu/rfl_empirical_tools)。我们将持续追踪 RFL 项目演进并定期更新仓库内容。

## A.3 Rust 在 Linux 中的基础原理

### A.3.1 Rust 的安全模型

Rust 是一种静态、强类型的编程语言，其安全模型通过对内存访问进行规范，从而保障内存和线程安全。简而言之，在任一时刻，仅允许一个变量写入某个内存位置，但多个变量可以同时读取。为此，Rust 实施了以下严格规则：

- **所有权与生命周期。**Rust 要求每个内存位置（即值）在任意时间只能有一个所有者，其它变量仅可通过引用读取该值。这种机制类似于 affine 类型系统<sup>[38]</sup>。

所有者的生命周期定义其作用域：一旦所有者离开作用域，与其绑定的值以及其引用变量将被 Rust 自动释放。

- **移动与借用。** 变量的所有权可以通过赋值或作为函数参数传递而发生转移 (*Move*)。也可以通过生成引用在不转移所有权的前提下借用变量 (*Borrow*)。一旦发生 Move 或 Borrow，原所有者将失去修改权限。Rust 还允许通过 `Send` 和 `Sync` traits 在多线程之间 Move 或 Borrow 所有权。

### A.3.2 unsafe 关键字

尽管上述机制保证了内存与线程安全，但它也极大地限制了语言的表达能力。例如，双向链表等复杂数据结构由于每个节点需被前驱与后继同时引用（即共享所有权），便违反了所有权独占性规则。因此，Rust 引入了 `unsafe` 关键字，作为一种“逃生舱”，允许程序员绕过编译器的静态检查。

`unsafe` 代码块中，程序员可以进行多种操作，包括裸指针解引用、调用外部函数接口 (FFI) 函数，甚至嵌入内联汇编。Rust 编译器在这些区域内将安全性责任转交给程序员。因此，只要程序员能够通过安全 API 封装 `unsafe` 区域，就能确保整个程序依然维持 Rust 的安全保证<sup>[39]</sup>。

### A.3.3 Rust 与 Linux 的集成方式

为将 Rust 编写的设备驱动集成进 C 编写的 Linux 内核，Rust-for-Linux (RFL) 首先预处理 Rust 驱动所依赖的内核 API (如 `kmalloc`)。利用 `rust-bindgen` 工具，RFL 自动生成对应的 Rust FFI 接口，这些接口符合 Rust 的调用约定，并被封装为 *kernel crate* (即内核库)，可通过 FFI 直接调用。

由于这些 API 最终运行于内核地址空间，无法由 Rust 编译器检查，因此 RFL 会为其包装一层安全抽象层，Rust 驱动只能调用该层导出的安全 API。例如，图 A.2 展示了一个将 Rust 字符设备驱动集成至 Linux 的完整流程。

具体而言，`rust-bindgen` 工具根据驱动依赖的头文件 (如 `cdev.h`) 生成三个 `crate` 中的 FFI 接口 (图 A.2a)；开发者随后手动构造安全抽象层，将 `bindings::cdev_alloc` 等 `unsafe` 调用封装至安全函数 `alloc` 中 (图 A.2b)；最终，驱动仅通过调用这些抽象层 API 与内核基础设施交互 (图 A.2c)。

值得注意的是，RFL 使用注释作为开发者之间的契约，明确 `unsafe` 区块的前置条件与变量使用方式，以推理其安全性。

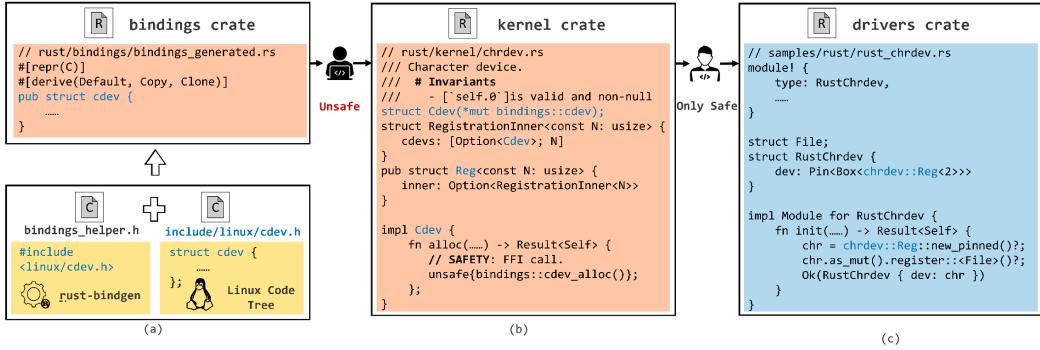


图 A.2 Rust-for-Linux 的架构示意图。(a) 使用 `rust-bindgen` 生成内核数据结构与接口的 FFI 绑定; (b) 开发者通过封装 unsafe FFI 构建安全抽象层 (kernel crate); (c) 驱动 (drivers crate) 调用安全抽象层接口以实现零开销的安全性。

### A.3.4 RFL 的目标

如表 A.1 所示, RFL 在最初的 RFC<sup>[10]</sup> 与官方展示<sup>[40]</sup> 中明确提出三大目标: 在不牺牲性能的前提下提升内核驱动的安全性, 引入现代开发体验与高效工具链, 并吸引更多开发者参与 Linux 内核开发。

表 A.1 RFL 项目的初始目标<sup>[10]</sup>

目标维度	具体内容
安全性 (Safety)	提供内存安全与线程安全的驱动开发机制。
性能 (Performance)	在抽象层不引入额外性能开销, 实现零开销。
工具链 (Tools)	提供更完善的文档与更高质量的 CI 测试支持。
开发效率 (Efficiency)	提高内核开发过程中的效率与可维护性。
社区 (Community)	吸引更多开发者参与内核社区开发。

## A.4 RQ1: RFL 的现状如何?

本节首先基于所收集的提交记录、问题追踪 (issues) 与邮件交流, 对 Rust-for-Linux (RFL) 当前的开发状态进行了系统分析。随后, 我们通过深入探讨 RFL 的安全抽象层与驱动实现, 揭示 Rust 安全模型与内核传统编程范式之间的张力。

### A.4.1 RFL 开发现状

我们的关键观察是: 尽管 RFL 项目仍处于早期, 但其底层基础设施 (如中断 irq、内存管理 mm、调度 sched) 已较为成熟; 而驱动与文件系统 (作为相对独立的子系统) 则明显不足, 预计将成为下一阶段发展重点。但由于缺乏同时熟悉 Rust 与内核

开发的高质量审稿人，该进展受到显著制约。

#### A.4.1.1 方法论

我们通过 GitHub 上的 PR 与提交记录，以及 Linux 邮件列表中的补丁记录，收集了 RFL 项目的代码，因为 RFL 社区通常在提交内核主线之前，会先在 GitHub 上进行协作开发<sup>[41]</sup>。

我们根据 RFL 社区的协作模型将代码分为三个阶段：

1. 待审（**pending**）：仍处于 GitHub PR 中，等待首次审查；
2. 候审（**staged**）：已通过 PR 阶段，进入邮件列表供内核子系统维护者正式审查；
3. 已合并（**merged**）：正式被合并至 Linux 主线或 Linux-next 分支。

总体而言，我们共收集到 160 余条已合并提交（约 1.9 万行代码），1300 多条候审提交（11.2 万行）与 500 多条待审提交（18.6 万行）。随后，我们使用自研工具基于正则表达式解析内核数据结构与函数接口，并按子系统与上述三阶段统计其演化与分布。

#### A.4.1.2 结果分析

**(1) 开发进度** 我们希望评估当前 RFL 代码库与理想中的全面 Rust 化内核之间的差距。从代码行数角度看，已合并代码仅占内核总量的约 0.125%，而其余 92.9% 的代码仍处于待审或候审阶段。如图 A.3 所示，我们进一步将已合并代码按其所属子系统分类展示。

**洞察 1：驱动、网络与文件系统构成 RFL 的“长尾”区域。**我们观察到一个明显的长尾现象：大部分 RFL 代码集中于调度器、内存管理与 IRQ 相关的基础设施。而与 Linux v6.2 中 78% 代码量相关的驱动、文件系统、网络与安全子系统，仅覆盖极少部分。这一现象合理：基础子系统为多数驱动共享，优先支持具有更高价值；而驱动与文件系统通常与特定设备强相关，需更高编程与审查成本。例如，netdev 社区审查一个网络 PHY 驱动补丁便历经了 6 个月与 11 个版本迭代<sup>[42]</sup>。

**(2) 补丁分布** 为分析 RFL 内部组成的演化过程，我们将补丁按使用场景分类为三类：Rust 编译器相关、Kbuild 系统支持与安全抽象层构建，并绘制其时间变化趋势，如图 A.4 所示。

**洞察 2：RFL 的基础设施趋于稳定，安全抽象与驱动将成为后续重心。**证据包括：

- Kbuild 相关补丁占比逐步下降，说明构建系统已基本完成；
- 安全抽象层补丁占比显著提升，在 18 个月内从 20% 提高至 60%；
- 在 2023 年 4 月出现一个提交激增点，对 RFL 库进行修改以支持固定初始化对

象的安全实现——修复前，`gpio_pl061` 与 `bcm2835_rng` 驱动仍使用 `unsafe` 初始化器。

**(3) 发展趋势** 图 A.5 展示了 RFL 项目的提交数与邮件活跃度随时间的演化，并标注 PR 平均审查时长。

**洞察 3：RFL 的瓶颈不在代码开发，而在于审查流程。**从左图斜率可见：RFL 在初期快速增长后逐渐趋于平台化；但右图显示，PR 审查速度显著下降。例如，2023 年上半年提交的 PR 平均需 280 小时才能被审阅，是三年前的 200 倍。这说明代码产出速度远快于其被“消费”的速度（即审查与合并）。

这一现象的原因包括：

- 缺乏熟悉 Rust 与内核的跨界审查者<sup>[43-44]</sup>；
- RFL 与内核子系统在协作习惯上的错位（如响应周期与风格）<sup>[45]</sup>；
- 存在发展“死锁”：子系统社区倾向于先看到 Rust 驱动才审查抽象层，而没有抽象层又难以构建驱动<sup>[46]</sup>。好在该问题已引起注意，并已有初步解决方案提出<sup>[47]</sup>。

令人鼓舞的是，RFL 正逐步被主线社区所接纳。例如，NVMe、NULL block、V4L2 与 e1000 等近期驱动，皆由内核社区主导开发并已采用 RFL 实现。

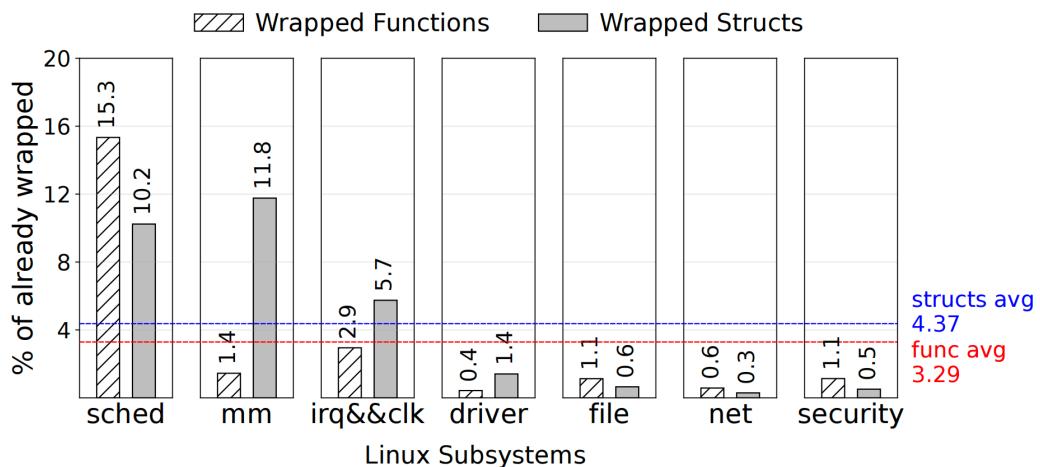


图 A.3 RFL 已合并的 API 封装进度概览

#### A.4.2 利用安全抽象实现 Linux 内核的 Rust 化

安全抽象是实现 Linux 内核 Rust化的关键路径之一，也是 RFL 代码量中最大的组成部分之一（见 §3.1）。顾名思义，该抽象层安全地将 C 编写的内核扩展到 Rust 驱动中：通过对内核数据结构和接口的抽象封装，使其在被调用时仍能保证内存与线程安全。

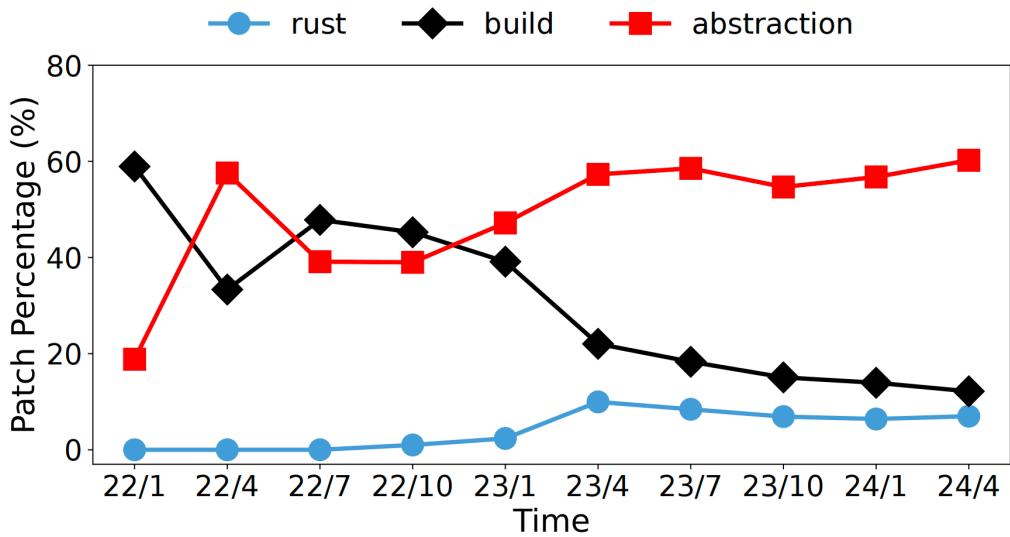


图 A.4 RFL 补丁类型时间分布：Rust 编译器、Kbuild、抽象层

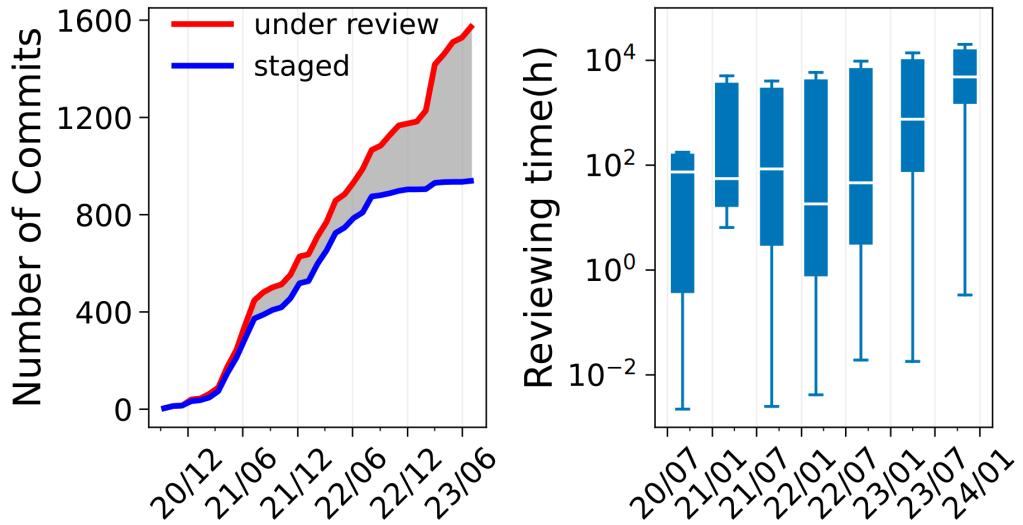


图 A.5 RFL 提交与审查趋势图（左：代码提交；右：平均审查时长）

**挑战：驯服内核编程范式** 为构建安全抽象层，RFL 首先将内核中的数据结构与接口翻译成 Rust 风格；随后，将其包装为 Rust 接口并导出给驱动使用（见 §2）。尽管该流程看似直接，但 RFL 需要应对一个关键挑战：如何将内核的编程范式与 Rust 的安全规则相协调？

例如，Linux 内核广泛使用类型转换、指针运算与位操作等技巧，这些都与 Rust 的理念存在冲突。为解决这一矛盾，RFL 采用了一系列规避策略，系统化地将内核状态纳入 Rust 风格管理体系，甚至推动语言自身演进以适应内核环境。

**内核数据结构转换** RFL 借助 bindgen 工具自动生成内核 C 结构体在 Rust 中的绑定。该过程基于规则驱动与语法转换机制，将 C 类型与符号翻译为其对应的 Rust 表达式。如 C 中的 `uint32_t` 会被映射为 `core::ffi::c_uint`，该类型是 Rust 原语类型 `u32` 的别名。该过程由 `rust-bindgen` 执行，遵循表 A.2 所列转换规则。

但并非所有 C 类型都能被一一映射为 Rust 原语。我们发现这种不兼容主要体现在内核中为强控内存布局而使用的语言特性上。

#### 洞察 4：内核对内存布局精细化控制的主动设计与 Rust 哲学冲突，带来额外开销。

- **模拟位字段与联合体。** 内核常用位字段和联合体提升内存效率，如 `e1000` 驱动用一个字存储多个标志位。但位操作违反 Rust 内存安全原则，因此无法被直接支持。RFL 使用字节数组模拟位字段，访问器方法负责读写位。虽然 Rust 支持 `union`，但无法提供与 C `union` 的 ABI 兼容性。因此，RFL 自定义一个 `_BindgenUnionField` 结构体，并用 `transmute` 操作在运行时重解释内存，全部位于 `unsafe` 块中。这些模拟带来的主要开销体现在二进制体积增加（见 §4.2）。
- **属性支持不完整。** 为提升局部性，内核结构常使用 `packed` 与 `aligned` 等属性。例如，`task_struct` 将调度信息按 cache line 对齐。尽管 Rust 提供 `#repr(C)` 支持，但仍可能错误处理某些属性，导致 `bindgen` 生成错误代码<sup>[48-49]</sup>。对于某些非常用属性，RFL 尚不支持，如 BTF 标记<sup>[50]</sup>。而如 `randomize_layout`，虽被内核用于缓解内存攻击，RFL 则忽略该属性，因为 Rust 的所有权机制与边界检查已能规避该类问题<sup>[51]</sup>。

尽管生成的绑定与 C 结构体在数据布局上等价，但因其大量使用裸指针(`*mut`)，无法直接向 Rust 驱动暴露。为此，RFL 使用辅助类型对绑定结构进行封装，并嵌入 Rust 风格的语义。

#### 洞察 5：RFL 使用辅助类型将内核数据的管理托管给 Rust，同时操作逻辑仍由内核负责。

具体而言，RFL 借助以下两种 Rust 特性实现内核数据结构的安全管理：

- **类型与 `Deref` 强制转换。** RFL 在嵌入内核结构体时重载其内存访问路径，将裸指针操作转化为类型安全的访问。举例而言，设备结构中常含 `void *` 指针指向设备私有数据，C 中运行时类型转换频繁。为此，RFL 为这些结构体实现 `Deref trait`，使解引用自动转化为正确类型。
- **生命周期自动管理。** RFL 引入三种低层类型(`ScopeGuard`、`ARef`、`opaque`)，绑定至辅助类型上，实现生命周期机制。例如，`ScopeGuard` 在作用域结束时

通过 Drop 自动释放资源，ARef 在引用计数变动时自动执行增减逻辑。相比传统内核开发中需显式调用 get\_task / put\_task，Rust 通过这些机制固化了非文档化的编程约定。

例如，在互斥锁或自旋锁中，RFL 重写其后端实现逻辑：在管理侧使用 Deref 转换访问锁保护数据，利用 ScopeGuard 进行内存回收；在操作侧仍调用原内核锁函数。

**内核函数包装** 内核函数的绑定与结构体转换使用类似规则。随后，RFL 在安全抽象层中广泛使用 Rust traits 将函数包装为结构体成员或 trait 方法，具体包括：

- **函数成员归类结构体中。** 将某一类结构体相关函数归类为其成员，形成 OOP 风格。例如，queue\_work\_on 与 \_\_INIT\_WORK\_WITH\_KEY 被集成成为 Queue 结构体的方法，增强代码可读性，并可避免空指针传参问题。
- **函数指针建模为 trait。** 内核中大量回调函数以函数指针形式存在，RFL 将其定义为辅助类型的 trait，并指定绑定类型与所有者结构，避免类型不匹配引发的漏洞<sup>[52]</sup>。
- **包装内联函数与宏。** 内核驱动广泛使用内联函数与宏（如 for\_each\_online\_cpu）。RFL 使用非内联 Rust 函数包装内联 C 函数，因当前 Rust 对 C FFI 的内联支持不友好，虽然可实现但不推荐<sup>[53]</sup>。而函数宏则更倾向于使用辅助函数而非 Rust 宏重写，主要因为 RFL 不希望维护两套易变的接口<sup>[54]</sup>。

表 A.2 rust-bindgen 中 C 到 Rust 的翻译规则

类别	C 中的定义	Rust 映射
类型	foo	core::ffi::c_foo
类型	foo *	*mut foo
	aligned	#repr(C) (有 caveats <sup>[48-49]</sup> )
属性	unused	忽略
	weak	忽略
属性	randomize_layout	忽略(因已由 Rust 所有权语义处理)
函数指针	fn	Option<fn>

### A.4.3 使用 Rust 编写设备驱动

随着编程范式从 C 语言转向 Rust，设备驱动开发中的“数据布局”也由传统的结构对齐逻辑转变为“数据所有权”视角的推理。在本节中，我们将从开发者视角出发，描述这种范式迁移对驱动开发的具体影响。

**设备探测（Device Probing）** 开发者需实现设备探测回调函数，在其中分配内核资源、初始化设备数据，并在必要时注册中断处理函数。相较于 C 写法，Rust 驱动的显著不同在于，开发者必须为设备数据显式标注所有权类型，即明确哪些实体可以访问这些数据。

例如：若数据可能被多个线程共享（如加锁结构），开发者需要用 `Arc` 包裹；若希望数据不可移动以支持与 C 接口共享，则需使用 `Pin`。更复杂的是，这些所有权标注往往是嵌套的，难以一目了然。

例如：`e1000` 网卡驱动中的接收环形缓冲区包含一个 `RxDesc` 描述符数组，并受自旋锁保护，其类型为：

```
Pin<Box<SpinLock<Box<Ring<RxDesc>>>>>
```

每一层嵌套类型都需调用对应的初始化器，如 `Pin::from`、`Box::try_new` 等<sup>[22]</sup>。

**实现驱动函数** 接下来，开发者需要实现驱动框架所要求的函数，例如网卡驱动需实现 `ndo_open` 与 `ndo_start_xmit` 回调函数（属于 `net_device_ops`）。这一步在流程上类似于传统 C 开发，即通过硬件手册中描述的行为逻辑，开发者完成设备驱动的核心操作逻辑。

然而，Rust 在内核空间的安全规则也带来了一些新的实现挑战：

- **(1) 动态数组实现复杂。** C 驱动常用裸指针实现可变长数组，用于存储动态内核对象（如 `pages`）。而在 Rust 驱动中，开发者需引入多个封装层并实现 `dyn_num trait` 以支持动态数量。如图 A.6 所示的 RROS 示例中，为实现等价的动态数组，Rust 实现增加了 83% 的代码行数，显著膨胀了目标文件体积<sup>[55]</sup>。
- **(2) 内核上下文问题仍需人工判断。** Rust 的安全规则并不自动检查执行上下文（如原子上下文与可睡眠上下文），无论是在编译时还是运行时。因此，调用方的上下文类型仍需由开发者自行判断与保证线程安全。

**设备清理** 当内核卸载设备或初始化过程出错时，驱动需释放已分配的资源。传统 C 驱动普遍采用 `goto` 跳转到统一的资源清理位置。而在 Rust 驱动中，这一过程可由 `Drop trait` 自动完成。通过 RFL 安全抽象层的封装，Rust 能在生命周期结束或错误发

生时自动回收相关资源，显著减少了人工管理负担。

**洞察 6：**使用 Rust 编写安全驱动的最大挑战，在于调和 Rust 的静态、不可变规则与内核编程的灵活性，这种矛盾往往被 RFL 与 Linux 社区所忽视。

```
// In C
struct elements { int len; void* inner; };
struct factory {
    struct elements inner;
};

// In Rust with Fixed N
struct elements<const N: usize> {
    inner: [foo; N],
}
struct factory { inner: elements<256/8> }

// In Rust with Dynamic change N
struct thread/proxy<const N: usize>{
    thread/proxy_elements: elements< N >,
}
impl dyn_num for thread<256>/proxy<8> {}
trait dyn_num { // fn use_elements(&self); }
struct factory { inner: &'static dyn dyn_n }
```

图 A.6 在 RFL 驱动中实现动态数组的复杂性示例<sup>[55]</sup>

## A.5 RQ2: Rust-for-Linux 是否达到了预期？

在本节中，我们回顾社区在启动 RFL 时设定的初始目标（见表 A.1），并围绕以下三个问题展开分析：(1) Rust 是否让 Linux 更加安全？(§ 4.1) (2) Rust 是否引入额外开销？(§ 4.2) (3) Rust 如何提升 Linux 开发体验？(§ 4.3)

### A.5.1 RFL 提升了 Linux 的可保障性 (securability)

**方法论：**我们聚焦于 Rust 语言在 RFL 及其驱动中的 bug 报告与 unsafe 代码块的使用情况。我们的基本逻辑是，RFL 的安全性依赖于 Rust 语言的安全保障能力，而这具体取决于驱动程序中是否消除了 unsafe 代码块，以及是否提供了安全抽象 API。因此，如果 RFL 存在安全漏洞，这些位置将是关键切入点。为此，我们首先收集了已

合入与待合入 RFL 代码中的所有 bug 报告与与安全相关的代码审查内容（如第??节所述），并依据 RFL GitHub 项目中使用的 issue 标签<sup>[56]</sup>将 bug 划分为“编译类 bug”与“健壮性（soundness）bug”；其中，死锁<sup>[57]</sup>与原子上下文中发生睡眠的并发类 bug<sup>[58]</sup>被归入健壮性 bug。随后，我们对上游仓库中的所有 RFL 驱动程序和 Rust 内核 crate<sup>[59]</sup>进行了静态分析，重点统计其 unsafe 代码块的使用情况。

**结果：**我们共发现 25 个来自 RFL 已合入与待合入代码的 bug，其中 15 个位于 Linux 主线，10 个位于 RFL 阶段性分支，详见表 A.4。在已合入主线的 bug 中，11 个为编译类 bug，其余 4 个与安全抽象有关。前者多由内核构建配置失误、Clang 工具链版本不兼容、Kbuild 与 rustc 之间的不匹配所致<sup>[60]</sup>，通常不引发实际安全风险。健壮性 bug 中，有 6 个位于安全抽象层，造成内存安全破坏；3 个破坏线程安全。

我们未在 Linux 主线中发现 RFL 驱动包含 unsafe 代码，因为截至目前，仅有一个约 130 行的小型驱动被合入<sup>[61]</sup>。但我们在提交至 RFL 邮件列表的驱动中，发现了若干 unsafe 使用情况，见表 A.3。

**后续分析：**通过对上述 bug 与 unsafe 代码的审计，我们总结了以下三个关于 RFL 安全性的核心发现：

**启示 7：**RFL 确实提升了 Linux 的可保障性（securability），但仍无法实现完全安全（security）。

- (1) Rust 的安全机制构成了内核安全的支柱。该语言层的支持可帮助开发者修复现有 bug 并规避潜在的内存与并发问题。我们将在第 5 节进一步展开说明。作为一门现代语言，Rust 拥有丰富的类型系统，支持如 klint<sup>[62]</sup>与 RustBelt<sup>[39]</sup>等形式化工具用于强化内核安全。相比于 C 语言，RFL 极大压缩了由内存错误导致的攻击面，开发者在进行安全推理时的负担显著减少。
- (2) unsafe 是无法避免的，尽管漏洞是可选的。我们的审计显示，所有主要驱动中普遍存在 unsafe 代码块，其根源在于两个方面：首先，内核需要直接控制内存与硬件，必须绕过 Rust 的所有权检查。例如，使用内联汇编管理 TLB 与发出内存屏障<sup>[28]</sup>，或解引用 MMIO 寄存器的裸指针等，这些操作超出了 Rust 所有权机制（即仿仿类型系统）的覆盖范围；其次，社区在部分 API 设计上不得不暂时妥协。例如，pin-init 接口的内存初始化机制被 Rust 安全检查工具判定为 unsafe，但在安全抽象层中长期存在，直到多轮资深开发者辩论后才被修复<sup>[63-65]</sup>。我们承认 unsafe 不等于漏洞，但它确实涉及可能的漏洞源（如 MMIO），无法完全消除。
- (3) Bug 不会消失，只是被“隐藏得更深”。一方面，被安全抽象与 Rust 驱动调用的内核函数仍可能存在被利用的 bug；另一方面，虽然 Rust 编译器可以即时

检测内存类 bug，却无法发现语义类 bug。此类 bug 往往源自 Rust 与内核在内存分配策略上的差异，难以检测，只能由精通 Rust 与内核的专家识别。例如，C 语言实现的 binder 驱动存在 use-after-free 漏洞<sup>[66]</sup>，若用 Rust 重写后，该漏洞不会以相同形式重现，而是转变为地址映射错误，逃过 Rust 编译器的所有检查。

表 A.3 RFL 驱动中的 unsafe 使用情况统计

驱动名称	因驱动逻辑过于复杂	缺乏安全抽象支持	总数
GPU <sup>[28]</sup>	107	7	114
NVMe <sup>[24]</sup>	44	16	60
Null block <sup>[23]</sup>	0	0	0
E1000 <sup>[22]</sup>	4	2	6
Binder <sup>[27]</sup>	45	9	54
GPIO <sup>[67]</sup>	0	3	3
Semaphore <sup>[68]</sup>	0	4	4

表 A.4 我们在 RFL 中发现的 bug 统计（括号内为合入/阶段性代码数量）

来源	编译类 bug	健壮性 bug
GitHub <sup>[59]</sup>	4(1/3)	7(3/4)
Intel LKP <sup>[69]</sup>	8(6/2)	0
邮件列表 <sup>[70]</sup>	4(4/0)	2(1/1)

### A.5.2 RFL 是否引入额外开销？

本节比较了 RFL 驱动与 C 语言实现的原生内核驱动，评估其性能和体积开销。我们的研究发现，Rust 驱动的可执行文件体积显著增加；其运行时性能与 C 相当，但在不同驱动与配置之间表现差异较大，波动明显。

**实验设置：**我们遍历了 RFL 的所有 Pull Request 与代码仓库，选取了 4 个具有实际应用场景的驱动，涵盖多种 IO 功能（如网络、存储），其中 NVME 与 Binder 被认为是首批合入 Linux 主线的 Rust 驱动；另外选取了 2 个 RFL 阶段性分支中的“玩具驱动”（即 gpio 与 semaphore），它们常用于探索 Rust 与 C 实现间的差异。

在这 6 个被测驱动中，仅 e1000 与两个玩具驱动实现了与 C 驱动功能一致的完整特性，其余驱动仅实现了部分功能。对于每个具有 C 对应版本的 Rust 驱动，我们分别编译二者、对比其二进制体积，并依据前人工作<sup>[37]</sup>进行基准测试，代表其典型

负载场景。表 A.5 展示了实验配置。

**二进制体积对比：**如图 A.7 所示，功能完备的 Rust 驱动二进制体积明显大于 C 实现：Binder 增长 1.2 倍，Gpio 增长 2.4 倍，Semaphore 增长 1.9 倍。我们进一步分析 ‘.text’ 段后发现，Rust 为支持泛型编程、边界检查、生命周期管理等特性而引入了 99% 的额外代码开销。即便是仅调用内核函数的简单封装，Rust 也会将其体积扩大约 33%。

值得注意的是，Rust 实现的 Binder 驱动体积开销相对较小，原因在于其广泛使用了带 `unsafe` 的函数指针而非泛型编程，从而在一定程度上牺牲了 RFL 设计初衷中的安全性。

虽然部署前一般会剥离调试信息，但在嵌入式系统中资源有限，调试模式下的开销亦不容忽视。我们将在附录 §B 中进一步讨论。

**性能表现：**总体来看，Rust 驱动与 C 驱动性能大致相当，差距在 20% 范围内。但部分场景中，Rust 明显逊色；而在另一些配置下，Rust 表现反而优于 C。

具体分析如下：

- 对于 `e1000`，Rust 驱动在 ping 延迟测试中性能是 C 驱动的 11 倍慢，如图 A.8 所示。主要原因在于 Rust 驱动未实现如预取（prefetch）等加速特性。
- 对于 `binder`，Rust 与 C 在 ping 延迟上表现相近，仅有 10% 性能差距。
- 对于存储驱动（如 NVME 与 NULL block），Rust 驱动在不同配置下出现了最高 61% 的性能下降，也有最多 67% 的性能提升，如图 A.9 与图 A.10 所示。我们观察到 Rust 驱动更适应较小任务数与 block size，这可能与 Rust 结构体尺寸更小、更易适配缓存行有关。

为了进一步分析相同行为路径下 Rust 与 C 驱动性能差异的根因，我们利用 `vtune` 与 `ftrace` 等内核工具开展微基准测试。总结如下：

**为何 Rust 驱动可能表现更差？**

- Rust 驱动的锁粒度较粗。虽然 Rust 通过语言规则保障线程安全，但高性能并发编程仍需开发者手动调优。
- Rust 的数组访问引入了边界检查，增加运行时开销，与已有研究结果一致，后者指出 Rust 程序在内存密集型场景下开销可能高达 C 的 2.49 倍<sup>[35]</sup>。
- Rust 采用模拟位域方式访问字段（见第 ?? 节），结合运行时检查进一步拖慢执行速度。
- Rust 广泛使用智能指针共享所有权，导致缓存/TLB/分支预测失效率上升。

**为何 Rust 驱动可能表现更好？**

- Rust 结构体通常比 C 更小，因其使用智能指针而非直接在结构体中分配字段内

存。使用 `pahole` 工具可发现，Rust 结构体更节省缓存行。

- 某些 Rust 驱动未实现全部功能，导致部分代码路径被省略。

**启示 8：**性能从来不是免费的——程序员的实现才是决定性因素！

表 A.5 测试 Rust/C 驱动的基准任务与指标（PC 配置：Intel i5-4590/4 核，Q87 主板，32GB DDR3，Samsung SSD 850 Evo，WD SN770，Intel 82545 网卡）

驱动	基准工具	测量指标	测试设备
NVME	fio	吞吐量、驱动体积	PC
Null Block	fio	吞吐量	PC
E1000	ping	延迟	PC
Binder	ping	延迟	Raspberry Pi 4B
Gpio_pl061	-	-	-
Semaphore	-	-	-

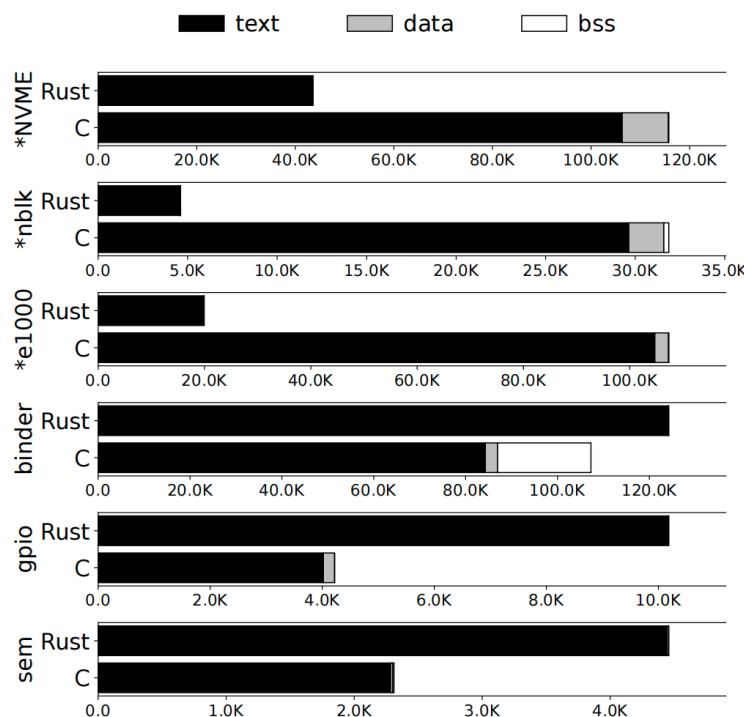


图 A.7 Rust 与 C 驱动的体积比较。带 \* 的表示 Rust 驱动未实现完整功能

### A.5.3 Rust 如何改善 Linux 的开发？

本节展示了 Rust 在提升内核代码质量、可读性以及吸引新开发者参与方面的显著作用。

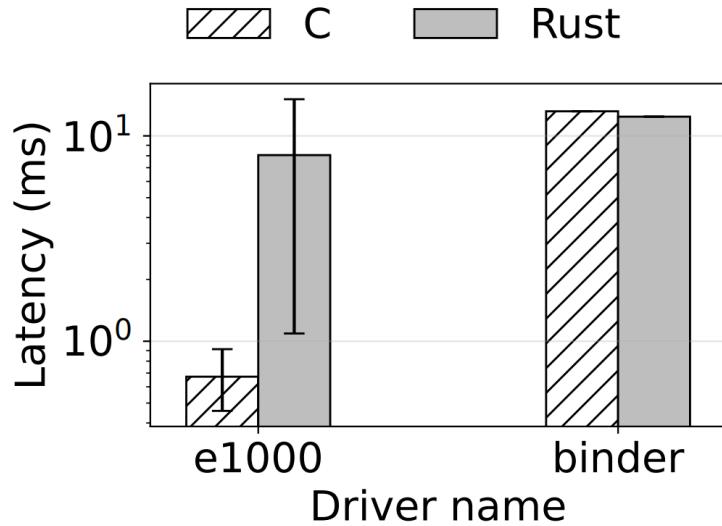


图 A.8 Rust 与 C 驱动延迟对比，e1000 驱动因缺失优化特性（如预取）性能落后

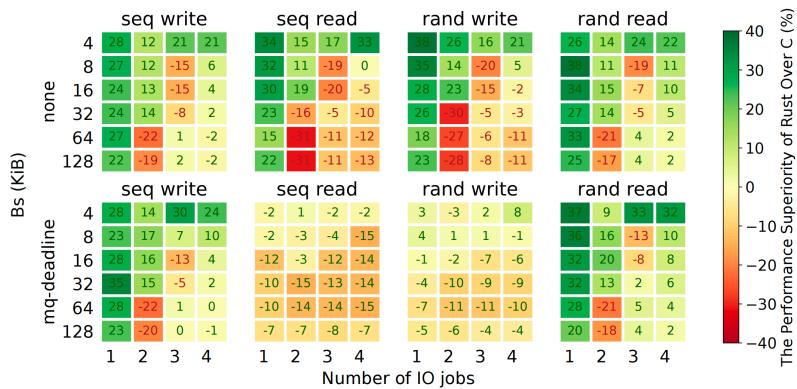


图 A.9 NULL block 驱动性能对比。绿色表示 Rust 表现更优，红色表示劣于 C

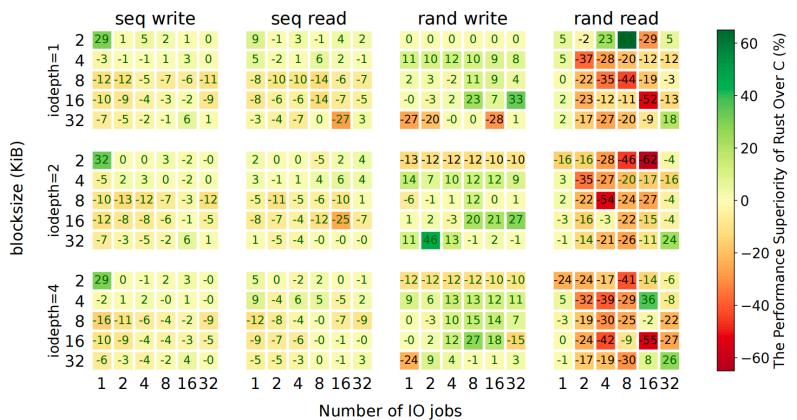


图 A.10 NVME 驱动性能对比

**代码质量与可读性的提升：**我们采用文档覆盖率与每千行代码的持续集成（CI）错误数作为衡量软件质量的两个关键指标<sup>[71-72]</sup>。

在文档覆盖率方面，我们统计了通过 EXPORT\_SYMBOL 与 EXPORT\_SYMBOL\_GPL 导出的 API；根据内核文档规范<sup>[73]</sup>，这些 API 应当全部配有文档。在 CI 错误方面，我们收集了来自 Intel LKP、Syzbot<sup>[74]</sup> 与 KernelCI<sup>[75]</sup> 的相关报告。

我们将 RFL 与两个近年来发展迅速的内核子系统 eBPF 与 io\_uring 进行了比较。结果如表 A.6 所示，RFL 在代码文档覆盖率方面表现优异（100%），而 eBPF 与 io\_uring 仅分别为 15% 与 31%。在每万行代码的 CI 错误数方面，RFL 也远低于其他两个子系统，分别减少了 49% 与 68%。

造成这一改善的原因主要有两点：

首先，RFL 利用了 Rust 的 rustdoc lints 功能，强制要求为所有（或部分）接口撰写文档。相比之下，传统内核开发主要依赖默认约定或人工审查，导致文档常被省略<sup>[76]</sup>。例如，io\_uring\_cmd\_complete\_in\_task 是 io\_uring 中通过 EXPORT\_SYMBOL\_GPL 导出的关键函数，用于在工作线程中异步完成 IO 操作，但由于缺乏文档，开发者常将其与 io\_uring\_cmd\_done 混淆，进而引发 AB-BA 死锁<sup>[77]</sup>。事实上，社区早已将“文档缺失”视为 Linux 中的第一号 bug<sup>[78]</sup>。

其次，Rust 内建的测试机制允许开发者在提交 PR 之前便运行测试。相比之下，现有的 Linux 内核测试框架（如 KUnit<sup>[79]</sup> 与 Intel LKP<sup>[69]</sup>）通常在代码合入或临近合入阶段才被触发。而在 RFL 中，测试代码可通过 #cfg(test) 注解集成，结合 GitHub CI，可在每次 PR 提交时自动运行。正是由于测试环节提前介入，使得 RFL 的 CI 错误显著减少，代码质量也随之提升。

**更多“新鲜血液”参与 Linux 社区开发：**我们进一步比较了 RFL 与 eBPF、io\_uring 以及传统子系统 netdev 的开发者构成。受到 netdev 相关研究启发<sup>[80]</sup>，我们使用开发者首次 commit 的时间跨度作为经验指标，将开发者划分为新手（0–24 个月）、熟练开发者（24–120 个月）与资深开发者（120 个月以上）。

采样区间为 Linux 版本 6.1 至 6.4。结果如图 A.11 所示，RFL 拥有最高比例的新手开发者（58%），显著高于 eBPF（39%）、io\_uring（38%）与 netdev（29%）。更有趣的是，我们发现其中有 29 位开发者此前从未向 Linux 提交过一行 C 代码。这表明，Rust 很可能是吸引他们进入内核社区的主要动因，打破了长期以来内核开发以 C 为中心的格局。

**然而，这些新手开发者尚未成为核心开发者或维护者：**尽管有更多新手被 Rust 吸引进入 Linux 社区，我们发现他们的贡献主要集中在 Rust 工具链或 crate 的构建上，

并未实质参与内核功能开发。相较之下，第 A.5 表中列出的 6 个驱动中，有 5 个主要由传统 Linux 社区的资深开发者完成。这说明年轻开发者与资深开发者之间仍存在断层，Rust 语言虽提升了吸引力，但并未真正降低内核开发的门槛。

表 A.6 代码质量测量结果。文档覆盖率以% 表示，CI 错误数按每 1 万行代码统计

子系统	文档覆盖率 (%)	CI 错误数/10K LoC
RFL	100%	3.8
eBPF	15%	7.5
io_uring	31%	11.9

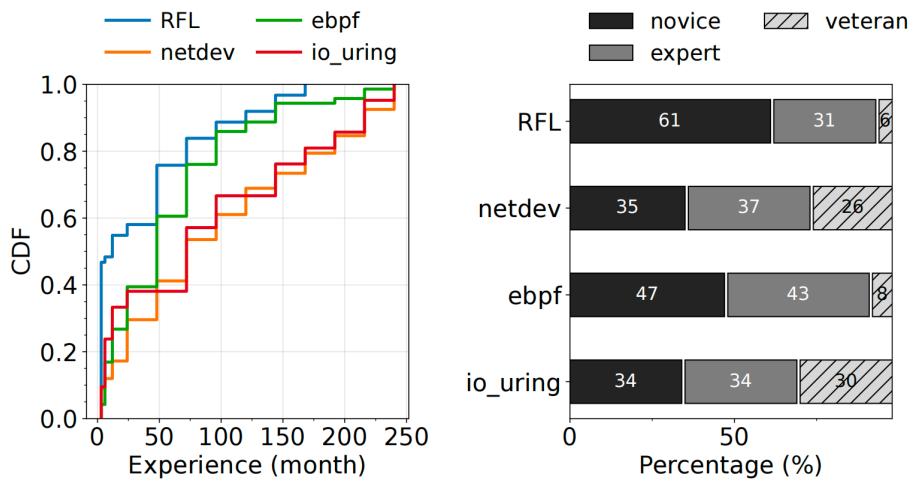


图 A.11 RFL 与其他主流内核系统的开发者经验分布。RFL 拥有最高比例的新手开发者

## A.6 RQ3：有哪些经验与教训？

本节总结了我们在本次实证研究中获得的关键经验与教训。

### 面向 RFL 开发者的经验教训

为提升 RFL 及其 Rust 驱动的安全性与实用性，开发者可考虑如下建议：

- 不要将 Rust 内建的安全检查器视为万能工具。正如第 ?? 节所示，Rust 的安全检查器在真实驱动中无法识别某些语义级 bug。因此，建议开发者结合更强大的分析工具（如 RustBelt<sup>[39]</sup> 与 miri<sup>[81]</sup>）以弥补 Rust 内建机制的不足。
- 从“所有权”视角构建安全的内核抽象，并管理驱动资源。这与传统内核编程中以“内存”视角出发的方式不同。编程前应先明确结构体间的所有权关系，否则在使用智能指针错误管理所有权后，修复代价将极为高昂。

- 接受 `unsafe` 作为最后的选项。Rust 的安全规则在处理内核内存操作时，常需大量使用智能指针与泛型编程，导致内存开销巨大（详见第 ?? 节）。此时，如若经过充分审查，可选择使用 `unsafe` 实现以换取实用性。

## 面向 RFL 社区的建议：如何扩展 RFL 的应用范围

决定将 RFL 优先应用于哪些未来内核驱动或子系统至关重要，因为这需要大量的开发资源与长期维护承诺。为此，我们构建了一个效益模型，用于评估“Rust 化 (Rustify)”的优先级。

该效益定义为：某驱动或子系统中可通过 Rust 机制修复的累计 bug/漏洞数量与其代码规模 (LoC) 的比值。直观理解是：一个规模较小但含有大量内存/线程类 bug 的子系统，其 Rust 化的价值更高。这一思路已得到社区共识的支持<sup>[82]</sup>。

为量化上述指标，我们遍历了各驱动的 git 历史，收集其至今出现的所有安全问题，并手动审查其中已修复的 bug，筛选出那些可通过 Rust 安全机制避免的内存/线程类 bug (依据第 ?? 节定义)。我们共分析了来自 79 个不同子系统的 2500 多个驱动，并将分析结果绘制于图 A.12。

平均而言，每个子系统包含 1.3 个 bug/每千行代码；但在不同子系统间，该比值差异显著。其中，`linux-block` 子系统由于每行代码含最多 bug (共 438 个修复中包括 113 个数据竞争 bug 与 98 个悬垂指针 bug)，因此被我们建议优先 Rust 化。令人欣喜的是，该子系统的 `null block` 驱动已经被社区用 RFL 重写，并在第 ?? 节进行了测试验证。

除 `linux-block` 外，我们的结果还表明 `linux-ext4` 子系统也具有较高的 Rust 化优先级。鉴于 VFS 安全抽象已被提出<sup>[83]</sup>，且已有多个基于 Rust 的文件系统实现<sup>[25-26]</sup>，我们预期 RFL 接下来将集中在 `ext4` 文件子系统上，并期待它能有效提升内存/线程安全保障能力。

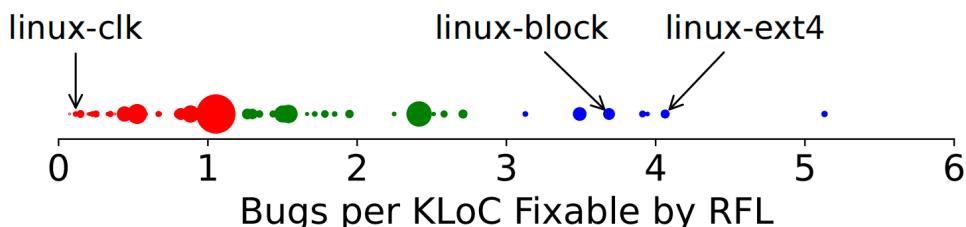


图 A.12 Linux 子系统分析结果。每个点代表一个子系统，点的大小表示其代码体量，相对颜色表示其 Rust 化的紧迫程度 (蓝色：最紧迫，红色：最不紧迫)

## A.7 相关工作

### 对野外环境中 Rust 使用的理解

已有研究从多个角度分析了 Rust 在实际项目中的 unsafe 使用情况。例如，部分工作以 Tock<sup>[84]</sup>、TiKV<sup>[85]</sup> 与 Redox<sup>[86]</sup> 等流行 Rust 项目为对象，研究其 unsafe 使用场景及背后的设计动因<sup>[33,87-89]</sup>，从而为 Rust 开发者提供如何更合理使用 unsafe 的实践指导。

另一些研究则聚焦于用户程序和 Rust 编译器中的 bug，分析其底层原因<sup>[34,90-91]</sup>。与这些工作不同，本文专注于对 Rust-for-Linux (RFL) 项目的研究，从 Rust 语言与 Linux 内核代码库的融合过程中提炼关键经验与教训，并评估 RFL 是否达到了其最初设定的目标，如提升安全性与实现零开销抽象。

### Rust 的运行时开销

尽管 Rust 旨在达到与 C 相当的运行效率，学界仍有不少研究探索其在实际中的开销表现<sup>[35-36,92]</sup>。这类研究通常将同一程序用 Rust 和 C 各自实现并比较其性能差异，但使用的基准程序大多较为简单，例如不足百行代码的经典算法示例，难以反映操作系统等复杂程序的真实情况。

Gonzalez 等<sup>[36]</sup> 实现了一个非平凡的 UDP 驱动，并从端到端视角比较 Rust 与 C 实现的差异。而我们则基于 Linux 驱动，系统性地度量了运行时指标与对象体积开销，从而更全面地评估 Rust 的开销表现。

### 基于 Rust 的操作系统内核

近年来，Rust 已成为系统软件开发的热门语言，学术界与工业界均探索基于 Rust 开发新型操作系统与内核，以适配嵌入式设备与个人计算机等不同场景<sup>[84,92-97]</sup>。

然而，这些工作大多关注如何构建纯 Rust 内核，并未深入探讨 Rust 与现有 C 代码融合的过程，而这正是当前面临的技术挑战。同时，已有研究亦未关注如何构建安全抽象以重用遗留代码。因此，本文所获得的经验与结论对于未来的 Rust 内核开发也具有重要参考价值。

### Rust 的增强机制

为使 Rust 更安全且更易于开发者使用，已有大量研究提出相关增强机制。一部分工作关注影响 Rust 普及率的因素，并提出降低学习门槛的建议<sup>[98-99]</sup>。另一些工作则致力于提升 unsafe 块的安全性，手段包括形式化验证<sup>[39]</sup>、静态分析<sup>[100]</sup>、沙箱机制<sup>[101-106]</sup> 等。

我们的研究同样为更好地将 Rust 融入 Linux 提供了启示，包括如何处理 unsafe 安全性、如何设计测试机制以及如何引导社区共建等方面。

## A.8 结论

本文对 Rust-for-Linux (RFL) 项目进行了系统性深入研究。RFL 是首个致力于将 Rust 引入 Linux 内核以增强其安全性与可维护性的开源项目，并日益受到社区关注与采纳。

我们首先从安全抽象层的构建与 Rust 驱动的实现两个维度出发，剖析了 RFL 当前的实现现状，揭示了 Rust 语言特性与传统内核编程范式之间所存在的张力。接着，我们分析了 RFL 是否以及在多大程度上兑现了其“构建更安全且零开销内核”的承诺。研究结果表明，RFL 在提升安全性方面确实有所贡献，但仍存在无法由编译器察觉的隐蔽性缺陷；而由 Rust 与 Linux 内核语义不兼容所引发的性能开销亦难以彻底避免。

最后，本文总结了研究过程中的关键经验与教训，希望为今后 RFL 的持续演进与开发提供有价值的指导与借鉴。

## 参考文献

- [1] Kernel self protection project. [https://kernsec.org/wiki/index.php/Kernel\\_Self\\_Protection\\_Project](https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project), 2024.
- [2] The documentation about how linux handle bugs. <https://docs.kernel.org/process/index.html#dealing-with-bugs>, 2024.
- [3] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021.
- [4] Duo Zhang, Om Rameshwar Gatla, Wei Xu, and Mai Zheng. A study of persistent memory bugs in the linux kernel. In Proceedings of the 14th ACM International Conference on Systems and Storage, 2021.
- [5] A linux cve caused by data race. <https://www.cvedetails.com/cve/CVE-2022-3566/>, 2022.
- [6] A linux cve caused by oob bugs. <https://www.cvedetails.com/cve/CVE-2023-38429/>, 2023.
- [7] A linux cve caused by use-after-free. <https://www.cvedetails.com/cve/CVE-2023-33288/>, 2023.
- [8] A linux cve caused by usedouble free. <https://www.cvedetails.com/cve/CVE-2023-28464/>, 2023.
- [9] The book linux device drivers, third edition. <https://lwn.net/Kernel/LDD3/>, 2024.
- [10] The rfc in the maillist for rust support. <https://lore.kernel.org/rust-for-linux/20210414184604.23473-1-ojeda@kernel.org/>, 2021.
- [11] The rust programming language. <https://www.rust-lang.org/>, 2023.

- [12] The ownership mechanism of the rust. <https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html>, 2023.
- [13] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In International Symposium on Formal Methods, 2006.
- [14] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In International Conference on Computer Aided Verification, 2021.
- [15] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Oil and water? high performance garbage collection in java with mmtk. In Proceedings. 26th International Conference on Software Engineering, 2004.
- [16] Mohamed Ismail and G Edward Suh. Quantitative overhead analysis for python. In 2018 IEEE International Symposium on Workload Characterization (IISWC), 2018.
- [17] A minimal linux kernel module written in rust. <https://github.com/tsgates/rust.ko>, 2013.
- [18] Why writing linux kernel modules in safe rust. <https://www.youtube.com/watch?v=RyY01fRyGhM>, 2019.
- [19] Linux kernel modules in safe rust. <https://github.com/fishinabarrel/linux-kernelmodule-rust>, 2021.
- [20] The barriers to in-tree rust talk in lpc 2020 llvm mc. <https://lpc.events/event/7/contributions/804/>, 2020.
- [21] Linux merges rust into the mainline. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b>, 2022.
- [22] The e1000 nic driver in rust. <https://github.com/fujita/rust-e1000>, 2024.
- [23] The null block driver written in rust. <https://lore.kernel.org/rust-for-linux/20230503090708.2524310-1-nmi@metaspacesdk/>, 2024.
- [24] The nvme device driver written in rust. <https://github.com/metaspacesdk/linux/tree/nvme>, 2024.
- [25] Puzzlefs build on the top of vfs abstractions in rfl. <https://lore.kernel.org/rust-for-linux/20230609063118.24852-1-amiculas@cisco.com/>, 2023.
- [26] Tarfs buiild on the top of vfs abstractions in rfl. <https://github.com/Rust-for-Linux/linux/pull/1037>, 2023.
- [27] The android binder driver in rust. <https://github.com/Darksonn/linux>, 2024.
- [28] The mac gpu driver in rust. <https://github.com/AsahiLinux/linux/tree/gpu/rust-wip>, 2024.
- [29] The first rust driver merged into in the linux mainline. <https://lore.kernel.org/rust-for-linux/20231213004211.1625780-1-fujita.tomonori@gmail.com/>, 2023.
- [30] The official statistics about the rfl community size. <https://kangrejos.com/2023/>, 2023.
- [31] The ebpf project. <https://ebpf.io/>, 2024.
- [32] The io\_uring project. <https://github.com/axboe/liburing>, 2024.
- [33] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? Proc. ACM Program. Lang., 4(OOPSLA), 2020.
- [34] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020.

- [35] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2023.
- [36] Amélie Gonzalez, Djob Mvondo, and Yérom-David Bromberg. Takeaways of implementing a native rust udp tunneling network driver in the linux kernel. In Proceedings of the 12th Workshop on Programming Languages and Operating Systems, 2023.
- [37] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Securing the device drivers of your embedded systems: framework and prototype. In Proceedings of the 14th International Conference on Availability, Reliability and Security, 2019.
- [38] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust. arXiv preprint arXiv:1903.00982, 2019.
- [39] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang., 2017.
- [40] The rust-for-linux official presentation in the rust meetup linz. <https://www.youtube.com/watch?v=fVEeqo40IyQ>, 2021.
- [41] The collaboration method of rfl community. <https://rust-for-linux.com/contributing>, 2023.
- [42] The network phy driver abstraction is merged. <https://lore.kernel.org/rust-for-linux/170263322444.1975.17234929609368010648.gitpatchwork-notify@kernel.org/>, 2023.
- [43] An evidence that rfl community lacks of enough available reviewers understanding netdev. <https://lore.kernel.org/rust-for-linux/CANiq72mDVQg9dbtbAYLSoxQo4ZTgyKk=e-DCe8itvwgc0=HOZw@mail.gmail.com/>, 2023.
- [44] An evidence that v4l2 community lacks of enough available reviewers understanding rfl. <https://lpc.events/event/17/contributions/1430/>, 2023.
- [45] An evidence that rfl community lacks of reviewers understanding netdev. <https://lore.kernel.org/rust-for-linux/20230614230128.199724bd@kernel.org/>, 2023.
- [46] The deadlock of rfl abstraction and real use drivers. <https://lore.kernel.org/rust-for-linux/8e9e2908-c0da-49ec-86ef-b20fb3bd71c3@lunn.ch/>, 2023.
- [47] Rfl breaks the rule of no duplicate drivers in linux. <https://lpc.events/event/17/contributions/1501/>, 2023.
- [48] Bindgen does not handle packed and aligned right. <https://github.com/rust-lang/rust-bindgen/issues/1538>, 2019.
- [49] Bindgen mishandles aligned typedefs. <https://github.com/rust-lang/rust-bindgen/issues/1753>, 2020.
- [50] Rfl discards btf\_type\_tag attribute. <https://github.com/rust-lang/rust-bindgen/issues/2244>, 2022.
- [51] The address space layout randomization wiki. [https://en.wikipedia.org/wiki/Address\\_space\\_layout\\_randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization), 2024.
- [52] A linux cve caused by incorrect type casting. <https://www.cvedetails.com/cve/CVE-2018-5861>, 2022.
- [53] The lto optimization for rust kernel modules. <https://kangrejos.com/2023/Inlining%20and%20LTO%20for%20Rust%20Kernel%20Modules.pdf>, 2023.

- [54] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. Transkernel: bridging monolithic kernels to peripheral cores. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.
- [55] Hongyu Li, Jiangtao Hu, Qichen Qiu, Yuxuan Shan, Bochen Wang, Jiajun Du, Yexuan Yang, Xinge Wang, Shangguang Wang, and Mengwei Xu. Rros: A dualkernel real-time operating system in rust, 2023.
- [56] The issue labels in the rust-for-linux github. <https://github.com/Rust-for-Linux/linux/issues>, 2024.
- [57] A deadlock bug in the rfl project. <https://github.com/Rust-for-Linux/linux/issues/998>, 2024.
- [58] A rfl driver bug report: potential sleep-in-atomic-context. <https://lore.kernel.org/rust-for-linux/87r0ykny6w.fsf@wdc.com/T/#t>, 2022.
- [59] The rust-for-linux project helping write drivers in linux. <https://github.com/Rust-for-Linux/linux/>, 2022.
- [60] A rfl bug due to the misconfig of kbuild. <https://github.com/Rust-for-Linux/linux/issues/735>, 2022.
- [61] The rust implementation of drivers/net/phy/ax88796b.c. <https://lore.kernel.org/rust-for-linux/20231213004211.1625780-5-fujita.tomonori@gmail.com/>, 2023.
- [62] Lints for kernel or embedded system development. <https://github.com/Rust-for-Linux/klint>, 2023.
- [63] The issue of "safe initialization of pinned structs" in github. <https://github.com/Rust-for-Linux/linux/issues/290>, 2021.
- [64] The 7 pin-init patches proposed in the github. <https://lore.kernel.org/all/?q=Rust+pin-init+API+for+pinned+initialization+of+structs>, 2024.
- [65] The patch of pin-init api. <https://github.com/Rust-for-Linux/linux/commit/90e53c5e70a69159ec255fec361f7dcf9cf36eae>, 2023.
- [66] Rewriting drivers in rfl can still have bugs. <https://lwn.net/Articles/953116/>, 2023.
- [67] The gpio driver written in rust. [https://github.com/Rust-for-Linux/linux/blob/rust/drivers/gpio/gpio\\_pl061\\_rust.rs](https://github.com/Rust-for-Linux/linux/blob/rust/drivers/gpio/gpio_pl061_rust.rs), 2024.
- [68] The semaphore driver written in rust. [https://github.com/Rust-for-Linux/linux/blob/rust/samples/rust/rust\\_semaphore.rs](https://github.com/Rust-for-Linux/linux/blob/rust/samples/rust/rust_semaphore.rs), 2024.
- [69] The intel linux kernel performance(lkp) project. <https://www.intel.com/content/www/us/en/developer/topic-technology/open/linuxkernel-performance/overview.html>, 2023.
- [70] The mainling list of rust-for-linux project. <https://lore.kernel.org/rust-for-linux/>, 2023.
- [71] Vikas S Chomal and Jatinderkumar R Saini. Significance of software documentation in software development process. International Journal of Engineering Innovations and Research, 2014.
- [72] Jorge Arturo Wong-Mozqueda, Robert Haines, and Caroline Jay. Is code quality related to test coverage? In Proceedings of the International Workshop on Sustainable Software Systems Engineering, 2015.
- [73] The linux kernel docuemnt requirement. <https://docs.kernel.org/doc-guide/kernel-doc.html>, 2023.
- [74] The google syzbot kernel fuzzer project. <https://syzkaller.appspot.com/upstream>, 2023.
- [75] The community-based distributed test automation system kernelci. <https://foundation.kernel.org/>, 2023.

- [76] Linus explained the documentation missing problem. [https://www.youtube.com/watch?v=bAop\\_8l6\\_ci&t=2275s](https://www.youtube.com/watch?v=bAop_8l6_ci&t=2275s), 2015.
- [77] Fix a ab-ba deadlock using io\_uring\_cmd\_complete\_in\_task. <https://lore.kernel.org/lkml/20230525183607.1793983-55-sashal@kernel.org/>, 2023.
- [78] Missing docuement is the #1 bug in the linux. <https://www.linuxtoday.com/blog/linux-bug-1-bad-documentation/>, 2009.
- [79] A unit testing framework for the linux kernel. <https://kunit.dev/>, 2023.
- [80] Time since first commit in the git history in netdev subsystem of linux. <https://people.kernel.org/kuba/more-development-statistics>, 2023.
- [81] An interpreter for rust’s mid-level intermediate representation. <https://github.com/rust-lang/miri>, 2024.
- [82] Rfl driver selection reasons. [https://lore.kernel.org/all/87y1ofj5tt.fsf@metaspacesdk/](https://lore.kernel.org/all/87y1ofj5tt.fsf@metaspacesdk), 2023.
- [83] Rfl vfs safety abstraction. <https://lore.kernel.org/rust-for-linux/CY6W7MLYLYEI.1DX1F6OL9I>IDV@suppilovahvero/T/#mbadc4049b874d7cc9621e0c4abcd36ec4bf9e4b>, 2024.
- [84] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles, 2017.
- [85] Tikv is an open-source, distributed, and transactional key-value database written in rust. <https://github.com/tikv/tikv>, 2016.
- [86] Redox is a unix-like operating system written in rust. [https://gitlab.redox-os.org/redox-os/redox/](https://gitlab.redox-os.org/redox-os/redox), 2015.
- [87] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020.
- [88] Sandra Hölterennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. ”i wouldn’t want my unsafe code to run my pacemaker”: An interview study on the use, comprehension, and perceived risks of unsafe rust. In 32nd USENIX Security Symposium (USENIX Security 23), 2023.
- [89] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. A closer look at the security risks in the rust ecosystem. ACM Transactions on Software Engineering and Methodology, 2023.
- [90] Xinmeng Xia, Yang Feng, and Qingkai Shi. Understanding bugs in rust compilers. pages 138–149, 2023.
- [91] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. ACM Trans. Softw. Eng. Methodol., 2021.
- [92] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. SIGOPS Oper. Syst. Rev., 2017.
- [93] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In Proceedings of the 10th Workshop on Programming Languages and Operating Systems, 2019.

- [94] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In Proceedings of the 8th Workshop on Programming Languages and Operating Systems, 2015.
- [95] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In Proceedings of the 8th Asia-Pacific Workshop on Systems, 2017.
- [96] A. Light, Thomas W. Doeppner, and Shriram Krishnamurthi. Reenix: Implementing a unix-like operating system in rust, 2015.
- [97] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, 2020.
- [98] Michael Coblenz, April Porter, Varun Das, Teja Nallagorla, and Michael Hicks. A multimodal study of challenges using rust. Plateau Workshop, 2020.
- [99] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and programming challenges of rust: A mixed-methods study. In Proceedings of the 44th International Conference on Software Engineering, 2022.
- [100] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021.
- [101] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021.
- [102] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In 32nd USENIX Security Symposium (USENIX Security 23), 2023.
- [103] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In Proceedings of the Seventeenth European Conference on Computer Systems, 2022.
- [104] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In Proceedings of the 9th Workshop on Programming Languages and Operating Systems, 2017.
- [105] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020.
- [106] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In Annual Computer Security Applications Conference, 2021.
- [107] Pierre Olivier, Jalil Boukhobza, and Eric Senn. Flashmon v2: Monitoring raw nand flash memory i/o requests on embedded linux. Acm Sigbed Review, 2014.
- [108] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012.

## A.A 案例研究：内核空间中 Rust 编程的灵活性问题

图 A.13 展示了一个典型案例，说明 Rust 的安全规则在某些场景下限制了内核编程的灵活性<sup>[55]</sup>。该例实现了一个字符设备的动态数组，左侧为内核 C 版本，右侧为 Rust 版本，其结构定义如图 A.13(a) 所示。

虽然两者结构上相似，Rust 实现存在关键缺陷：用于指定工厂数组大小的常量  $N$  一旦设定即不可更改（例如设置为 256），这一不可变常量将固定应用于所有工厂实例。无论线程工厂可能需要 256 个元素，代理工厂仅需 8 个元素，Rust 下二者都只能选择统一大小（要么都为 256，要么都为 8），从而造成内存浪费或碎片。

相比之下，C 内核实现则可通过指针修改 `len` 字段，从而轻松实现字符设备的动态注册与数组扩展。

尽管如此，开发者可借助一系列绕过方案克服上述限制，如图 A.13(b) 所示。首先需手动定义 `dyn_num` trait 并声明其使用（如 `use_elements` 函数），并通过 `dyn trait` 机制启用动态分发。接着，引入常量泛型参数  $T$  用以表达每个实例的不同元素数量。最后，为具体类型实现该 trait，以分别定义线程工厂（256）与代理工厂（8）的大小。

上述方案虽然可行，但会增加开发复杂度，并引入运行时检查，从而影响性能。

<pre>// In C struct elements {     int len; void* inner; }; struct factory {     struct elements factory_inner; };  struct factory thread_thread = {     .factory_inner = {         .len = 256, .inner = thread_ptr     } }; struct factory proxy_thread = {     .factory_inner = {         .len = 8, .inner = proxy_ptr     } };  // In Rust pub struct elements&lt;const N: usize&gt; {     inner: [i32; N], } pub struct factory {     factory_inner: elements&lt;8/256&gt;, }  let thread_factory = factory { factory_inner:     elements::&lt; 8/256 &gt; { inner: [0; 8/256] } }; let proxy_factory = factory { factory_inner:     elements::&lt; 8/256 &gt; { inner: [0; 8/256] } }; </pre>	<pre>// In Rust but inflexible pub struct elements&lt;const N: usize&gt; {     inner: [i32; N], } trait dyn_num { // fn use_elements(&amp;self); } pub struct factory {     factory_inner: &amp;'static dyn dyn_num }  struct thread&lt;const T: usize&gt;{     thread_elements: elements&lt;T&gt;, } impl dyn_num for thread&lt;256&gt; {}  struct proxy&lt;const T: usize&gt;{     proxy_elements: elements&lt;T&gt;, } impl dyn_num for proxy&lt;8&gt; {}  static thread_factory_inner: thread&lt;256&gt; = thread {     thread_elements: elements { inner: [0; 256] }, }; static proxy_factory_inner: proxy&lt;8&gt; = proxy {     proxy_elements: elements { inner: [1; 8] }, }; let thread_factory = factory { factory_inner:     &amp;thread_factory_inner }; let proxy_factory = factory {factory_inner:     &amp;proxy_factory_inner }; </pre>
--	---

(a)

(b)

图 A.13 示例展示了 RFL 驱动编写中 Rust 的灵活性问题

## A.B 带调试信息的驱动体积开销

在调试阶段，开发者可能需要保留 Rust 驱动的调试信息。图 A.14 展示了带调试信息（如 `debug_foo` 区段）下 Rust 与 C 驱动体积的对比。

即使是功能未完全实现的 Rust 驱动，其二进制体积也比对应的 C 实现大 3.9–6.6 倍。主要原因在于 Rust 广泛使用泛型编程，导致生成了更多符号及更长的符号名，从而显著扩大调试段的大小。

在资源受限的嵌入式设备中（如闪存和内存仅数 MB 级别<sup>[107]</sup>），这种开销是不可忽视的。因此，为实现调试功能，如何压缩驱动体积是嵌入式系统开发中的关键挑战<sup>[108]</sup>。

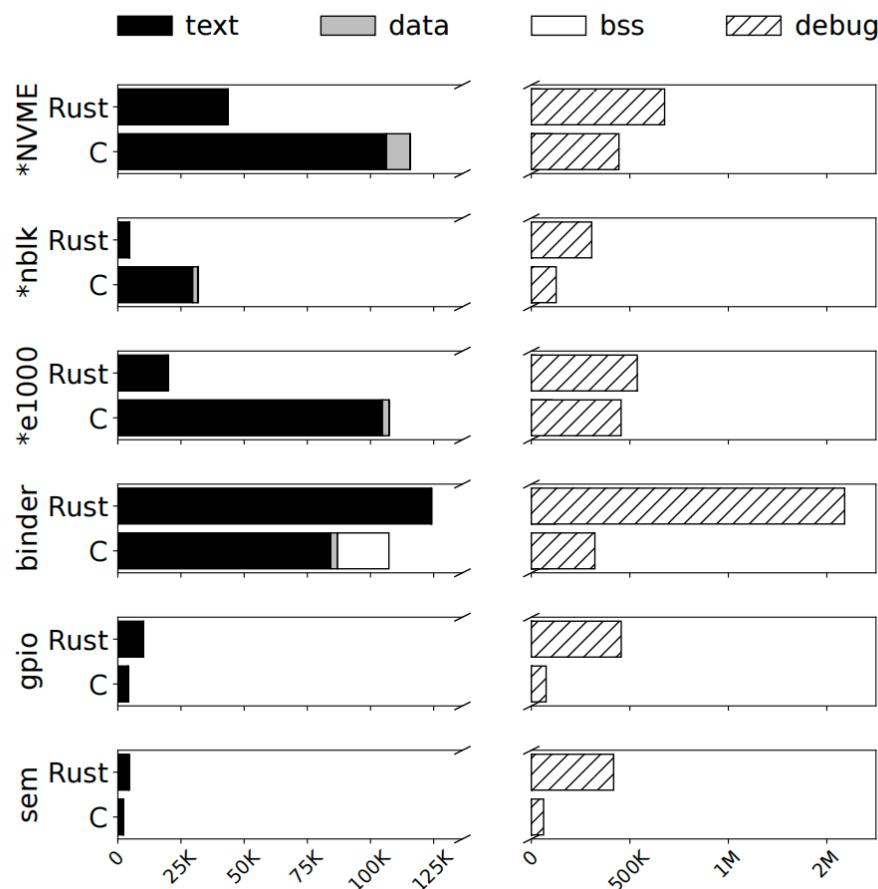


图 A.14 Rust 与 C 驱动在启用调试段下的体积对比。带 \* 的表示 Rust 驱动未实现完整功能

## A.C 内核社区对 Rust 的看法

为更深入了解 Linux 内核社区对 Rust 的态度，我们收集了截至 2023 年 8 月 5 日，在 LWN 与 YCombinator 平台上关于 Rust 驱动开发的相关帖子，并利用 ChatGPT 对

其进行分析。

分析分为两个部分：一是情感分析（sentiment analysis），将观点划分为正向与负向；二是意见挖掘（opinion mining），识别正负观点背后的具体原因。分析结果如图 A.15 所示。

总体来看，Rust 因其安全性与性能获得社区较高认可。然而，开发者最关心的问题仍是 Rust 较高的学习曲线。这表明，Rust 要在内核领域获得更广泛接受，还需时间逐步验证其稳定性与实用性。

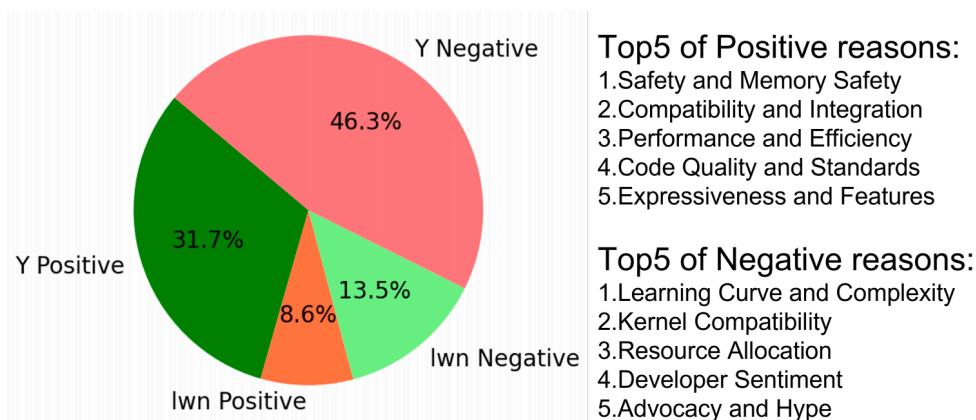


图 A.15 开发者对 RFL 在 Linux 中使用的观点统计与分析

## 致 谢

感谢陈渝老师的悉心指导和耐心帮助，让我在研究过程中受益匪浅。

## 声 明

本人郑重声明：所呈交的综合论文训练论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名： 林奕辰 日期： 2025年6月10日

## 在学期间参加课题的研究成果

暂无

## 综合论文训练记录表

学生姓名	林奕辰	学号	2021010550	班级	计 15	
论文题目	操作系统宏内核的网络管理组件设计与实现					
主要内容以及进度安排	<p>主要内容</p> <p>1. 分析其他现有工作如何实现网络管理，基于现有工作的基础上，参考相关已有实现，补全系统调用的网络管理功能</p> <p>2. 提供抽象化模块化的接口，保证模块的易用性</p> <p>3. 针对 lwIP、smoltcp 的对接进行优化，确保模块的高性能和鲁棒性</p> <p>进度安排</p> <p>1月：探究已有的操作系统中系统调用的网络管理的实现</p> <p>2月：完成对现有项目网络协议栈接口的具体分析和组件设计</p> <p>3月：参照其他项目，针对对接 lwIP、smoltcp 的细节进行处理</p> <p>4月：将模块抽象化、易用化，满足其他开发者的调用需求，提供清晰明了的文档说明</p> <p>5月：性能优化与测试，设计并执行测试方案，修复发现的问题</p> <p>6月：完成毕业论文的提交和修改，进行最终答辩</p>					
	<p>指导教师签字: <u>戴乾兰</u></p> <p>考核组组长签字: <u>陈涛</u></p>					
	2025年 1月 9 日					
	中期考核意见	本次中期考核通过。				
		<p>考核组组长签字: <u>陈涛</u></p>				
		2025年 3月 27 日				

指导教师评语	<p>在本科综合论文训练过程中，林奕辰同学基于 Rust 编程语言、ArceOS unikernel 和 Starry-next 宏内核，扩展网络管理功能，补全网络管理系统调用，通过扩展多种支持宏内核的网络管理组件和其它相关组件，来提升 Starry-next 宏内核运行 Linux 网络应用的能力。同时还对分析了 lwIP 和 smoltcp 两个网络协议栈组件。课题工作表明，林奕辰同学具有较好的计算机专业理论技术基础以及项目研发能力。</p> <p>指导教师签字: <u>戴艳兰</u></p> <p>2025年5月28日</p>
评阅教师评语	<p>林奕辰同学以 ArceOS 和 Starry-next 为研究对象，分析 unikernel 和宏内核的特征，并对网络管理相关的功能和系统调用进行扩展，扩展了宏内核中网络管理相关的功能和接口，并设计并运行测试用例，验证所实现的网络管理组件的功能正确性。</p> <p>评阅教师签字: <u>陈涛</u></p> <p>2025年5月29日</p>
答辩小组评语	<p>本次答辩过程讲述有条理，回答问题基本正确，论文撰写结构比较规范，逻辑比较清晰，掌握计算机专业技术，具备项目研发能力，达到本科毕业设计的水平，考核通过。同意林奕辰同学通过答辩，建议授予工学学士学位。</p> <p>答辩小组组长签字: <u>陈涛</u></p> <p>2025年6月5日</p>

总成绩: B+

教学负责人签字: 陈涛

2025年6月6日