# 北京理工大學

## 本科生毕业设计（论文）外文翻译

**外文原文题目：** Writing Network Drivers in Rust

**中文翻译题目：** 使用 Rust 语言编写网卡驱动

# 跨操作系统的异步音频驱动模块设计与实现

**Design and Implementation of Cross-Platform Asynchronous Audio and Video Driver Module**


学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：07112005

学生姓名：穆新宇

学　　号：1120202695

指导教师：陆慧梅

# 使用 Rust 语言编写网卡驱动

## 摘　要

许多开发者认为编写网络驱动程序是一项不愉快的任务。主要有三个原因导致他们不愿意这么做：在内核空间工作的困难性，大多数驱动程序的复杂性以及对 C 语言开发的普遍不情愿。

然而，现在有其他选择。内核模块不必一定用 C 语言编写，现代网卡支持越来越多的硬件委托功能，可以大大简化驱动程序的复杂度。用户空间网络驱动程序的兴起也避免了编写内核代码的需要。

我们展示了一个用 Rust 编写的先进用户空间网络驱动程序，该驱动程序注重简单性、安全性和性能，从而证明了驱动程序开发既具有挑战性又具有回报性。这个驱动程序总行数 1306 行，不安全代码占比不到 10%，它注重核心的数据包处理（packet processing）功能，但在单个 3.3GHz 的 CPU 核心上，转发（forwarding）性能高达每秒超过 2600 万包，超过内核及其他一些用户空间驱动程序。

我们讨论了我们的实现细节，从不同角度对这个驱动进行评估。从结果来看，我们得出结论——Rust 是否适合编写网络驱动程序。

**关键词：北京理工大学；本科生；毕业设计（外文翻译）；Rust；网卡驱动；**

# 目 录

# 第 1 章 介绍

近年来，越来越多的开发者将驱动程序从内核空间移植到用户空间。与网络驱动相关的这个趋势背后的一个主要原因，就是套接字 API 的性能瓶颈问题。通用内核栈对现代需求来说简直太慢了。过去，开发者为规避这个问题而自己编写内核驱动程序。但是，内核驱动开发是一个繁琐的过程，因为在这么低级的代码层面上出现任何编程错误都可能和将导致内核崩溃。此外，内核还对开发环境和内核中可用工具施加各种限制。

幸运的是，内核驱动开发有其他选择。在第二章中，我们通过展示 Linux 中低级网络通信的不同方法来说明这些选择。我们还谈到用户空间驱动的优点之一就是可以选择任何编程语言进行实现。但是，大多数用户空间驱动仍然用 C 语言编写，这是一种早已过时的语言, 如果不小心处理的话很容易导致缓冲区溢出、堆栈溢出、段错误、内存泄漏等未定义行为的问题。由于可以选择任何编程语言，那么就产生了一个问题——哪种编程语言特别适合网络驱动开发?

为回答这个问题，我们在第三章中讨论了网络驱动编程语言应该具备的优良属性，并建议选择 Rust 这一先进编程语言来替代 C 进行网络编程。Rust 能够满足我们所有理想的特性。我们通过介绍 Rust 的核心特性和一些独特的 Rust 概念来详细讨论 Rust。

# 第 2 章 Linux 上的网络通信

虽然互联网如今无处不在，但网络通信、网卡和网络驱动程序仍然经常被开发人员和用户视为黑箱。对于用户来说，大多数时候，他们只需要将网线插入电脑就可以上网了。对于开发人员来说，操作系统提供的高级 API 让他们可以轻松地在其程序中包含网络通信功能，而无需处理低级网络编程的复杂性。

然而，如果还没有为某种网卡提供驱动程序，或者当报文处理性能成为一个重要考虑因素时，有必要对网络通信的基础知识，尤其是应用程序与操作系统和网卡交互的方式，有初步的了解。每次从应用程序发送数据时，数据都会经过几层处理，直到到达网卡。

| | |
|---|---|
| 7 | 应用层 |
| 6 | 表示层 |
| 5 | 会话层 |
| 4 | 传输层 |
| 3 | 网络层 |
| 2 | 数据链路层 |
| 1 | 物理层 |

图 2-1 OSI 参考模型中的通信层次

一个好的模型——也是事实上的标准——是 OSI 参考模型，OSI 模型描述了这些层次（layers），它的结构如图2-1所示。

模型中的每一层都为上层提供不同的服务，如数据包分割（第 4 层）、路由（第 3 层）或可靠传输（第 2 层）。

通常，应用程序负责应用层、表示层和会话层，而传输层和网络层则由操作系统通过通用内核网络堆栈进行处理。最低的两层，物理层和数据链路层，由网络接口卡（NIC）和网络驱动程序控制。

由于 Linux 操作系统的设计，传统上，网络应用程序和网络驱动程序的不同任

务在 Linux 中被用户空间和内核空间分隔开。然而，Linux 中有不同的低级数据包处理方法：大多数应用程序使用内核的套接字 API，而一些应用程序运行自己的内核模块，一些在内核空间中处理所有内容，一些在用户空间中处理所有内容。

## 2.1 内核空间

Linux 是基于宏内核的操作系统，这意味着整个操作系统都在内核空间中运行，而其他所有服务都在用户空间中运行。内核为用户空间应用程序提供套接字 API，即各种函数和数据类型，用于在所谓的 *Berkeley* 或 *POSIX* 套接字上提供易于使用的接口。这些套接字是特殊文件，可以通过简单的读写操作与本地进程或远程主机进行通信，遵循 Unix 的 “一切都是文件” 的概念。希望建立通信通道的用户空间应用程序会调用`socket()`函数，该函数最终会在内核中创建一个套接字结构实例，并向应用程序返回该套接字的文件描述符。随后，`bind()`和`connect()`或`listen()`和`accept()`将两个或多个进程之间的连接与套接字关联起来，`send()`和`recv()`则用于向套接字发送和接收数据。

为了减小通用内核堆栈的问题，一些应用实现了它们自己的内核模块。对于这种方式，两个著名的例子是 Open vSwitch[1]和 Click Modular Router[2]。

然而，由于内核空间与用户空间之间频繁地上下文切换，使用自定义内核模块可能仍然不够快。可以通过完全在内核空间中运行驱动程序来减少性能损耗。尽管内核空间中的应用程序允许非常快速地处理数据包，但由于内核中缺少调试工具，以及通常的限制（例如不允许浮点运算）和普遍存在的软件错误可能导致系统崩溃的线程，编写它们非常麻烦。

## 2.2 用户空间

另一种快速处理数据包的方法是使用完全在用户空间（包括它们自己的网络堆栈/驱动程序）中运行的应用程序，如 DPDK 和 Snabb[3]。具有 root 权限的特殊设备文件允许应用程序从用户空间访问网卡和其他物理设备。

尽管像中断处理这样的某些操作可能很棘手，但用户空间驱动程序可以缓解许多与内核空间驱动程序相关的问题。它们通常更简单、更灵活，可以用任何编程语言编写，并利用通常可用于软件开发的所有工具。出现 Bug 将会有更轻的后果，并且由于上下文切换较少，接收和发送的数据包不一定需要在用户空间和内核空间之

间进行复制，因此性能可以显著提高。例如，DPDK 能够以非常接近线路速率（line rate）的性能运行。

由于这些优势，我们选择在用户空间中开发我们的驱动程序，尽管也可以用 Rust 编写内核模块。

# 第 3 章 选择 Rust 编写驱动程序

编程语言是任何开发人员的基本工具。多年来，已经创建了数千种编程语言。最早的高级编程语言之一是 Plankalkül，由 Konrad Zuse 于 1942 年至 1945 年期间创建，随后是 FORTRAN、COBOL 等，直到 70/80 年代的 C、C++ 和现代语言如 Kotlin 和 Swift。仅编程语言的大量存在就引发了这样一个问题：哪些编程语言通常适合网络驱动程序，哪些特别适合?

每种语言都遵循各种概念，也称为编程范式，这些范式分类了代码的组织方式和执行方式。由于编程语言从根本上有所不同，因此有必要选择一种适合我们编写网络驱动程序任务的编程语言。一般来说，网络驱动程序应该具有高性能、可靠性，即没有任何未定义的行为和软件错误，简单、易用和易于理解。

这些要求导致了对编程语言的一些关键设计决策。首先，为了满足高性能，该语言应该是编译型语言而不是解释型语言。虽然解释型语言通常更容易实现并且可以“即时”执行，但它们在执行过程中需要解释（因此得名），并且在编译时缺乏进行强大优化的机会。因此，像 Python 这样的语言速度较慢，不应该是我们的首选。

关于可靠性和安全性，编译器应拒绝任何不安全的代码，并尽可能在编程错误时向我们发出警告。尽管 C 和 C++ 的编译器一直在持续改进，然而，在太多情况下，编译器并未通过警告开发者来避免未定义行为。

在开发阶段，另一个需要考虑的要点是编程语言的生态系统。它主要包括可用于编程的工具，如编译器、开发环境、错误检查器等，文档以及通常的开发者或用户社区。

总的来说，网络驱动程序的完美编程语言应该是一种高性能的编译型语言，具有内存安全、易于阅读和使用的特点，并提供丰富的文档和其他开发辅助工具。幸运的是，有一种相对较新的语言声称可以满足所有这些目标：Rust。

Rust 是一种由 Mozilla 和其他人开发的新型系统编程语言，旨在提供一种“安全、并发和实用”的编程语言。Mozilla 于 2010 年宣布 Rust，第一个稳定版本 Rust 1.0 于 2015 年 5 月 15 日发布。Rust 在语法上与 C++ 相似，但旨在通过零成本抽象提供更高的内存安全性和值得信赖的并发性。它是一种编译型编程语言，结合了并发、函数式和泛型编程等不同范式。Rust 由一个独特的所有权、移动和借用系统组

成，该系统在编译时强制执行，使垃圾收集变得不必要。这个系统是满足 Rust 内存安全目标的关键[4-5]。

## 3.1 语法

Rust 的语法与 C 和 C++ 的语法密切相关。语句由分号分隔，代码块由大括号包围。控制流由`if`、`else`和`while`等关键字控制。然而，Rust 中引入了 for-each-loops 的`for`关键字，`match`用于更强大的模式匹配而不是`switch`，并且没有三元条件运算符。函数使用`fn`关键字声明，变量使用`let`声明。函数和变量通常使用下划线命名法，而数据类型使用驼峰命名法。此外，Rust 还支持函数式编程，如具有自己语法的 lambda 函数。

## 3.2 类型系统

Rust 是一种静态类型编程语言。尽管 Rust 编译器必须在编译期知道所有变量的类型，然而这并不意味着这些变量的类型必须显示声明。Rust 具有类型推断功能，可以确定变量的类型。由于类型冲突或未知类型，将值分配给代码中的变量失败会导致编译时错误。要将值多次分配给变量，必须使用`mut`关键字声明变量，因为 Rust 中的变量默认是不可变的。

用户定义的数据类型可以声明为`struct`或`enum`（标记联合）。使用`impl`关键字，可以在这些用户定义的数据类型上声明方法（就像在其他语言中使用类一样）。Rust 中提供了一种类似于类型类的机制，称为“特性”（traits）。特性是一组可以扩展类功能的方法。受到 Haskell 的启发，通过向类型变量声明添加约束来实现多态性。例如，实现`Add` triat 允许使用`+`运算符作用于用户定义的类型。当一个函数是泛型函数时，它的泛型参数通常需要实现一个或多个`trait`。Rust 使用`trait`的组合来代替继承。

## 3.3 内存管理和安全性

Rust 最核心的特性是其独特的所有权系统。所有权使 Rust 能够在避免垃圾回收的同时保证内存安全。尽管许多语言都使用垃圾回收器自动跟踪对象并在不再使用时释放它们，但这有两个主要的缺点。

首先，使用垃圾回收器清理资源是非确定性的。垃圾回收器是复杂的软件组件，

理解为什么内存没有被释放可能是一个挑战。如果开发人员想要确保在某一时刻清理资源，这通常意味着他要么必须强制垃圾回收器清理所有内容，要么等待资源准备好被释放。这两种选择都剥夺了开发人员的控制权，令人失望。

其次，垃圾回收可能导致潜在的性能问题，因为垃圾回收器需要不同数量的 CPU 周期来执行清理。如果要清理的内容很多，垃圾回收所花费的时间尤其高。这对于实时应用程序（如在线游戏或我们的网络驱动程序）来说可能是一个问题。

没有垃圾回收器，开发人员负责在不再使用内存时将其返回给操作系统。公开数据库中列出的与内存误用相关的安全问题的数量证明，这是一项难以履行的责任。如果开发者忘记释放内存，那么就是在浪费内存，当内存耗尽时，应用程序将会崩溃。如果开发者过早地释放内存，那么将会产生无效引用，也被称为悬空指针，这是未定义的行为，就像两次释放内存一样。

C 和 C++ 因这些问题而广为人知。Rust 没有这些缺点，它采取了不同的方法：通过其所有权系统限制指针的使用方式。Rust 的规则使其编译器能够在编译时验证程序是否没有内存安全错误，即前面提到的悬空指针、双重释放等。然而，它的目标是让开发者负责内存管理，同时防止不安全操作。在运行时，Rust 编写的程序与 C/C++ 编写的程序没有区别，只是编译器已经证明内存得到了安全处理。事实上，相同的规则是安全并发编程的基础，因为它们可以防止数据竞争（data race）和数据损坏（data corruption）。拥有一个线程安全的编程语言是一个宝贵的优势，因为大多数网卡使用多个接收和传输队列，因此可能由不同的线程管理。

## 3.4 所有权

Rust 的所有权系统由三条简单的规则构成。在 Rust 中，每个值都有一个所有者。对于某个特定的值，一次只能有一个所有者。当这个所有者超出作用域，即其生命周期结束时，该值将被释放。在 Rust 术语中，这被称为“丢弃（dropping）”一个值，这与 C++ 中的资源获取即初始化（*Resource Acquisition Is Initialization, RAII*）模式类似。

变量的作用域是指在该程序中变量有效的范围，例如，在函数内部定义的变量在函数返回时变得无效，并且它们的值被释放。所有权可以从一个变量转移到另一个变量。这被称为“移动（moving）”一个值，当函数以参数调用时，将值赋给变量，或函数返回一个值时，会发生这种情况。然后，这个值不是被复制而是被移动。确

切地说，这并非适用于所有值。在编译时已知大小的类型，如整数、布尔值、浮点数等，都会被复制，因为它们完全存储在堆栈上，因此复制它们代价很低。

当一个值被移动后，之前的引用将变得无效，并且无法再访问，以确保内存安全，即没有悬空指针。为了将一个值传递给函数而不转移所有权，即不移动该值，Rust 使用**引用**（references）。

可以使用&运算符创建对值的引用。将引用作为函数参数被称为“借用（borrowing）”，因为在借用函数返回之前，如果已传递可变引用，则其他函数无法使用该值。

与变量类似，存在可变引用和不可变引用，并且默认情况下所有引用都是不可变的。关于引用，Rust 强制执行额外的限制：可以对变量或单个可变引用进行无限次不可变引用，但不能同时进行。值的生命周期必须至少与对该值的任何引用的生命周期一样长。这意味着我们不能返回在函数内部声明的值的引用。

```
1 fn main() {
2     let s = String::from("Hello");
3     let r = &s;
4     let t = s;
5 }
```

图 3-1 尝试移动一个存在引用的值

此外，只要存在对该值的引用，就不能移动该值。图3-1和图3-2说明了为什么在 Rust 中禁止这样做。s是一个指向堆上字串“hello”的变量。它包含字符串的长度以及内存缓冲区的容量和实际指针，并存储在堆栈上。r是对s的引用。到目前为止，即图3-1中的第 3 行，代码是完全合法的。然而，将s赋值给t会移动s，使之前使用的内

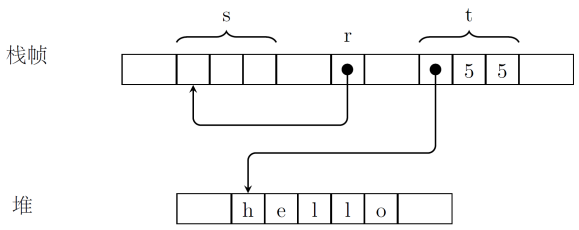


图 3-2 引用一个已经被移动的字符串，来源“Programming Rust”[4]

存变为空，并使r变成悬空指针。为了消除这个错误，Rust 不允许在引用时移动值，并且也不能修改它们以防止数据竞争。[1]

## 3.5 不安全代码

Rust 的所有权系统非常强大。然而，静态分析相当保守，仍然受到有限的决策能力的限制。当编译器无法确定代码是否满足所需的保证时，编译器会拒绝有效的程序。

为了让 Rust 编译器接受这种代码，Rust 使用了unsafe关键字。使用 unsafe 特性类似于签署合同：现在，开发人员有责任遵守规则，以避免未定义的行为，因为编译器无法自动强制执行它们。

使用 unsafe 代码，可以使用 Rust 的四个附加功能。这包括取消引用原始指针，调用 unsafe 函数，包括来自 Rust 的外部函数接口的函数，访问和修改可变静态变量以及实现 unsafe 特性。

使用这些功能时，开发人员必须注意遵守规则。例如，禁止在原始引用末尾之后解引用指针。这违反了使用unsafe的约定，并导致未定义的行为。Rust 中的许多功能都有要遵循的规则，但是只要滥用它们的可能后果不会导致未定义的行为，它们就不需要unsafe关键字。

因此，函数内部存在 unsafe code 并不一定意味着该函数必须标记为unsafe。实际上，将函数声明为unsafe与该函数是否使用需要不安全的代码无关。有一些罕见的情况——比如我们驱动程序中的Packet::new()——函数本身不包含任何不安全代码，但仍然应该声明为unsafe。

Rust 的习惯用法通常意味着不安全代码被包装在一个安全的 API 中。一个需要不安全代码的方法的好例子是split_at_mut()函数，它接受一个 slice 参数并在给定 index 处将其拆分为两个 slice。图3-3给出了split_at_mut()的一个简单实现。

该函数断言，切片应被拆分的索引位置位于切片内，即mid <= len。如果断言失败，程序将立即终止。在 Rust 中，由于不可恢复的编程错误而终止被称为 “panic”。不幸的是，Rust 的借用检查器无法理解该函数借用了切片的不同部分，这是有效的，但在编译时会返回错误，指出同一切片同时有两个可变引用。使用不安全的 Rust 代

[1]注意，新版本的 Rust 编译器对于图3-1中的代码不会报错，因为引用 r 的生命周期到第 2 行就结束了，而 s 的生命周期到第 3 行，满足“引用的生命周期小于值的生命周期”的定义，这也是 Rust 编译器不断改进的结果。

```rust
fn split_at_mut(slice: &mut[i32], mid: usize) -> (&mut[i32],
↪  &mut[i32]) {
    let len = slice.len();
    assert!(mid <= len);
    (&mut slice[..mid], &mut slice[mid..])
}
```

[5] 图 3-3 split_as_mut() 的简单实现，来源“The Rust Programming Language”

码，可以实现split_at_mut()。图3-4显示了 Rust 标准库中的通用实现，这里稍作修改以提高可读性。

```rust
fn split_at_mut(slice: &mut[i32], mid: usize) -> (&mut[i32],
↪  &mut[i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();
    unsafe {
        assert!(mid <= len);
        (from_raw_parts_mut(ptr, mid),
        ↪  from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

图 3-4 split_as_mut() 实际上的实现，包含了 unsafe code

就像简单的实现一样，该函数断言索引位于切片内部，并通过使用from_raw_parts_mut()和每个切片部分的原始指针以及新切片的长度，要么引发panic，要么返回对两个不同部分的可变引用。

from_raw_parts_mut()是不安全的，因为用户必须确保原始指针是有效的，并且确实指向一个切片（具有正确的长度）。offset方法是不安全的，因为它无法保证给定偏移量的指针是有效的。然而，将这两个不安全的函数与之前的断言结合起来，就形成了一个安全的抽象，因此是对不安全代码的适当使用。

通过强制开发人员使用不安全代码，可能导致未定义行为，Rust 确保开发人员意识到他们的操作，即不会无意中使用不安全功能，并希望尽可能避免不安全代码，这对于任何计算机程序，尤其是网络驱动程序来说都是可取的。

# 第 4 章 IXY

Ixy 是一个网络驱动程序，旨在展示网卡在驱动层面的工作原理。它完全在用户空间实现，其架构与 DPDK 和 Snabb 类似。

用 C 语言编写的 ixy 网络驱动程序的主要设计目标是简单、无依赖、易用和速度。虽然 Snabb 具有非常相似的设计目标，但 ixy 的 C 版本试图 “简化一个数量级”。因此，一个简单的转发器和驱动程序由不到 1000 行的 C 代码组成。由于没有外部库和内核代码，可以在几个步骤内探索不同的代码级别。每个函数都只有几个调用距离应用程序逻辑。为了保持驱动程序尽可能简单，已经省略了一些功能（如各种硬件卸载可能性）。

驱动程序的初始化和操作与 Snabb 非常相似，而内存管理、批处理和抽象来自 DPDK。Ixy 提供了两种不同的实现，ixgbe 和 VirtIO。在初始化时，将选择适当的驱动程序，并将包含指向特定驱动程序实现的函数指针的结构返回给用户。Ixy 提供了在给定 pci 地址上初始化网卡以及发送和接收数据包的功能。该框架还公开了设备统计信息，以便进行性能测量，以及一个安全的包 API（packet API），该 API 使用自定义数据结构在内存池中分配和释放数据包。应用程序直接包含 ixy，使用这些抽象可以方便地处理驱动程序。用户可以从上到下阅读它，没有任何复杂的层次结构。

Ixy 使用 ixgbe 系列的网卡，因为这些卡在通用服务器中非常常见，而且英特尔会发布关于它们的详细的 datasheet。

# 第 5 章 实现

ixgbe 驱动的实现主要依赖于原始的 ixy C 驱动和英特尔的 datasheet。除非另有说明，否则以下各节中的任何代码引用均指我们实现的`master`分支的提交`e193471`。关于英特尔 datasheet 的页码和节号指的是 82599ES 数据表的第 3.3 版（2016 年 3 月）。[6]

## 5.1 总体设计

Ixy 的目标是简单、快速和可用。我们的实现也遵循这些原则。但我们在项目中又增加了一个目标：安全。

只要开发人员避免使用`unsafe`关键字，Rust 本身就是一种安全的编程语言。因此，我们的驱动程序实现会尽量减少 unsafe 代码的使用，并在必要时进行解释。目前，驱动程序总共使用大约一百行`unsafe`代码。

与 C 语言版本的 ixy 不同，我们并不关注“无依赖”。借助`crates.io`和`cargo`，Rust 提供了一个出色的包管理系统，并鼓励开发人员积极使用他人提供的包，遵循 Unix 哲学，即让每个程序只做一件事。与 unsafe 代码一样，我们将说明我们在哪里以及为什么包含了包（在 Rust 中称为“crates”）。到目前为止，我们的依赖项中有四个包。其中之一是`log`包，用于提供日志 API，以便我们的驱动程序用户决定是否使用日志，以及如果使用，以何种形式使用。

我们提供了两个示例应用程序，以展示如何使用驱动程序。其中一个应用程序是数据包转发器（packet forwarder），另一个应用程序是数据包生成器（packet generator）。这两个应用程序通过将驱动程序的所有输出写入`stdout`来展示日志 API 的工作原理。

## 5.2 架构

我们的实现架构基于 C 驱动的架构。为了给我们的驱动用户提供简单的接口，我们使用了由 Intel ixgbe 驱动大幅简化版本实现的 trait，未来也可以由 VirtIO 版本实现（类似于 ixy）。该特性是所有使用我们驱动的应用程序的公共接口。它提供了在给定 PCI 地址上初始化 ixgbe 网卡并使用其发送和接收数据包的方法。使用 trait 比 C 实现提供的简单抽象更优雅。我们在设备上实现所有与驱动相关的函数，而在 C

中，必须将指向设备的指针传递给每个函数。与原始的 ixy 驱动一样，我们公开了设备统计信息以进行性能测量，并为所有内存操作提供了安全的 API。

## 5.3 安全考虑

值得注意的是，我们的实现需要 root 权限才能访问 PCIe 设备。然而，一个自定义的内核模块仍然会更糟糕，因为它由于其性质而具有 root 权限，并且必须用不安全的编程语言 C 编写。

理论上，驱动程序可以使用`seccomp(2)`初始化设备后降低其权限。不幸的是，这还不够，因为设备仍然完全受驱动程序控制，并且可以通过直接内存访问（DMA）写入任何内存区域。要编写真正安全的用户空间网络，必须利用 I/O 内存管理单元（IOMMU），这是将 PCIe 设备传递到虚拟机中的现代虚拟化功能。在 Linux 中，可以使用专门为“安全、非特权、用户空间驱动程序”设计的`vfio`框架来实现这一点。对于驱动程序未来的工作，应该考虑利用`vfio`，尽管它超出了本论文的范围。

## 5.4 初始化

准备 82599ES 系列网卡以发送和接收数据包需要调用`ixy_init`函数一次，该函数包含三个参数，分别是网卡的 PCIe 地址以及所需的发送和接收队列数量。

为了与 PCIe 设备进行通信和管理，Linux 提供了一个名为`sysfs`的伪文件系统。驱动程序通过读取和写入特殊文件来更改设备状态。例如，在设备初始化之前，为了解除任何使用网卡的内核驱动程序，驱动程序将 NIC 的 PCIe 地址写入`unbind`文件。为了启用 DMA，我们从配置文件中读取两个字节，修改后再写回。在这些操作之前，我们通过从`config`文件中读取设备类别来执行一个基本的检查，以确定 PCIe 设备是否确实是一个网卡。我们在驱动程序中包含了`byteorder` crate，以便能够从该文件中读取不同大小的整数（具有正确的字节序），而无需处理原始字节和不安全的代码。

解除绑定和启用 DMA 后，将导出设备的基地址寄存器（BARs）的文件映射为共享内存，这意味着对内存映射区域的更改（即不同的寄存器）将写回到文件中，反之亦然。基地址寄存器用于配置网卡。所有寄存器的偏移量在 Intel datasheet 中都有记录。我们使用了`ixgbe_type.h`的一个稍微简化的版本（包含作为`#define`的所有偏移量和一些结构体），该版本来自我们将其转换为 Rust 代码的 ixy 驱动程序，通过将

```rust
pub fn mmap_file(path: &str) -> Result<(*mut u8, usize), Box<Error>> {
    let file =
    ↪  fs::OpenOptions::new().read(true).write(true).open(&path)?;
    let len = fs::metadata(&path)?.len() as usize;
    let ptr = unsafe {
        libc::mmap(
            ptr::null_mut(),
            len,
            libc::PROT_READ | libc::PROT_WRITE,
            libc::MAP_SHARED,
            file.as_raw_fd(),
            0,
        ) as *mut u8
    };
    if ptr.is_null() || len == 0 {
        Err("mapping failed".into())
    } else {
        Ok((ptr, len))
    }
}
```

图 5-1 Rust 中的内存映射

结构体标注为 C 结构体，将`#define`替换为 Rust 的等效`const`，重写所有三元运算符等。从技术上讲，使用此常量文件使我们的代码库增加了 4000 多行代码。然而，我们在这 4000 行代码中只使用了其中的不到一百行，并且手头有一个包含所有寄存器的文件对于未来在驱动程序上的工作非常方便。

对于内存映射，我们并没有使用来自`crates.io`的特殊 crate，因为使用原始指针与我们的偏移量常量和偏移量函数一起工作更为方便，并且在原始指针之上实现安全的 API 也非常简单。

图5-1展示了如何在我们的驱动程序中映射文件内存。映射函数以读写权限打开给定路径的文件，并将文件描述符和文件长度传递给`libc::mmap()`，`libc::mmap()`将文件作为共享内存映射到内存中，并返回指向内存位置的指针。`libc::mmap()` 必须位于 unsafe 块中，因为它只是从 C 标准库（一个外部函数）调用`mmap(2)`。如果`mmap(2)`函数返回一个空指针，或者给定文件的长度等于 0，函数就返回一个错误。否则，将返回映射文件的指针和长度。

在 Rust 中调用外部函数时，我们必须指定函数的签名，并使用`extern`进行注释。

```rust
fn set_reg32(&self, reg: u32, value: u32) {
    assert!(reg as usize <= self.len - 4, "memory access out of
    ↪ bounds");

    unsafe {
        ptr::write_volatile((self.addr as usize + reg as usize) as
        ↪ *mut u32, value);
    }
}
```

图 5-2 在 Rust 中设置一个 32 位的寄存器

```
static inline void set_reg32(uint8_t *addr, int reg, uint32_t value)
↪ {
    __asm__ volatile("" : : : "memory");
    *((volatile uint32_t *)(addr + reg)) = value;
}
```

图 5-3 在 C 中设置一个 32 位的寄存器

这称为绑定（binding）。libc crate 为 libc 中的所有函数提供了绑定，因此我们在驱动程序中使用该 crate，无需担心绑定问题。

在将设备文件映射到内存后，按照数据手册中的建议，禁用中断以防止重新进入，重置设备，由于我们不支持中断处理，因此再次禁用中断，启用网络自动协商，并重置设备接收和发送数据包的统计计数器（statistical counters）。

所有这些操作都是通过获取和设置映射设备的一些寄存器来完成的。图5-3和图5-2展示了 Rust 和 C 中`set_reg32()`函数的差异。

C 实现中的第 2 行使用了编译器屏障，以防止编译器重新排序对`addr + reg`的后续内存访问，而第 3 行中的`volatile`关键字用于防止编译器完全优化掉内存访问。由于 x86 架构不需要，Rust 中缺少这种编译器屏障。通过使用`ptr`模块中的`write_volatile()`可以实现与`volatile`相同的效果。由于写入任意内存位置是不安全的，因此`write_volatile()`需要一个 unsafe 块。通过断言，寄存器确实位于设备的映射内存中，我们提供了一个安全的抽象，并且不必将`set_reg32()`标记为 unsafe。

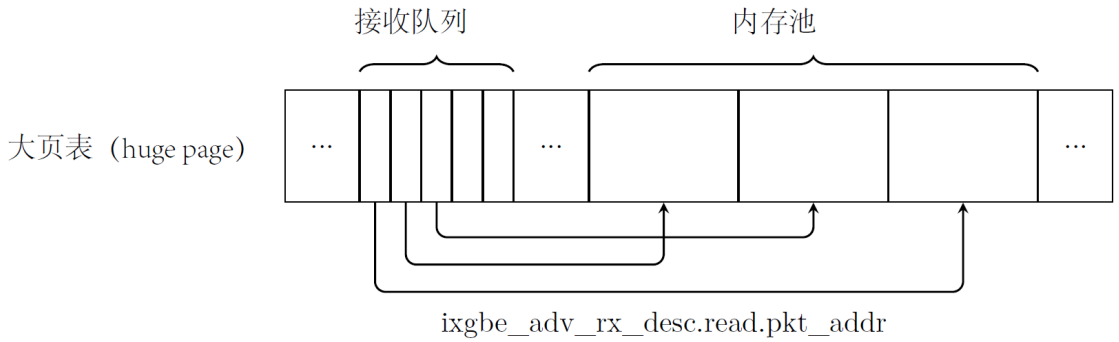


图 5-4 接收队列，其中包含指向内存池中数据包缓冲区的 DMA 描述符。

## 5.5 DMA

设备复位后，接收（RX）和传输（TX）数据包的队列将被初始化。图 5.1 显示了接收队列的一般结构。每个队列都是一个环形缓冲区，其中填充了描述符，描述符包含指向数据包物理地址的指针以及有关数据包的一些元数据，例如它们的大小。网卡可以被配置为使用过滤器（filter）或哈希（hashing）将流量分割到多个队列中。对于简单的配置，使用一个接收队列和一个发送队列就足够了。

为了初始化接收和发送队列，需要为队列分配内存。由于网卡独立于中央处理器（CPU）使用这些内存，即通过网卡的物理地址访问内存，因此这些内存必须驻留在物理内存中。在内核空间中，有一个用于分配 DMA 内存的 API，但在用户空间中，由于此 API 不可用，我们必须使用其他机制。为了禁用交换（swapping），我们可以使用`mlock(2)`。不幸的是，此函数不是 Rust 标准库的一部分，而是 libc 的一部分。因此，我们使用来自`libc` crate 的绑定并调用`mlock(2)`。然而，`mlock(2)`仅确保页面保留在内存中。内核仍然可以将页面移动到不同的物理地址。为了解决这个问题，我们使用大小为 2 MiB 的大页面。Linux 内核目前无法迁移这些页面，因此它们驻留在物理内存中。

在大页面上分配 DMA 内存相对简单。我们仓库中的`setup-hugetlbfs.sh`脚本创建一个`hugetlbfs`挂载点，并将所需的大页面数量写入`sysfs`文件。然后，通过在已挂载的目录中创建新文件并使用来自 libc 的`mmap(2)`将文件映射到内存中来完成内存分配。

可以通过`procfs`文件`/proc/self/pagemap`得出映射内存的物理地址。每个队列都与一个大页面匹配，并通过基址寄存器将物理地址通信到网卡。

```c
struct mempool {
    void* base_addr;
    uint32_t buf_size;
    uint32_t num_entries;
    uint32_t free_stack_top;
    uint32_t free_stack[]; // 包含 entry id
}
```

图 5-5 C 中的内存池

```rust
pub struct Mempool {
    base_addr: *mut u8,
    num_entries: usize,
    entry_size: usize,
    phys_addresses: Vec<usize>,
    pub(crate) free_stack: Vec<usize>
}
```

图 5-6 Rust 中的内存池

## 5.6 内存池

正如上一节所解释的，接收和发送队列的描述符包含指向数据包数据的指针，该数据包数据通过 DMA 由网卡读取和写入。因此，数据包数据也必须位于 DMA 内存中。

每次接收到数据包时都分配内存会产生巨大的开销。内核中的通用网络堆栈需要大约 100 个 CPU 周期来执行这些分配。使用管理已分配内存的内存池可以大大减少 CPU 周期的数量。

因此，每个接收队列都附有一个用于传入数据包的内存池。发送队列没有自己的内存池，因为将数据包从现有池中传递到它们就足够了。

图5-5和图5-6展示了 C 和 Rust 中内存池的结构。在 C 中，内存池被实现为包含一系列条目 id 的结构体，用于释放数据包缓冲区。缓冲区的虚拟地址通过`base_addr + (entry_id * entry_size)`计算得出。在 Rust 中，除了我们使用`Vec<T>`（称为 Vector）来替代`free_stack`，以及另一个用于存储内存池中缓冲区物理地址的 Vector 外，其余部分与 C 相同。Vector 是 Rust 中用于堆栈的原生数据类型，它们是连续可增长的数组。

在 C 中，来自内存池的数据包缓冲区本身就是结构体，包含一些像数据包大小

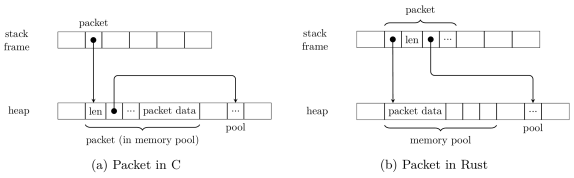


图 5-7 C 和 Rust 中 Packet 的布局

```rust
pub struct Packet {
    pub(crate) addr_virt: *mut u8,
    pub(crate) addr_phys: usize,
    pub(crate) len: usize,
    pub(crate) pool: Rc<RefCell<Mempool>>,
    pub(crate) pool_entry: usize,
}
```

图 5-8 Rust 中的 Packet 结构体

或对内存池的引用这样的头部字段，以及一个未指定大小的数据字段。在 Rust 中，可以创建具有未指定大小字段的结构体，但它们处理起来并不方便，而且通常相当不寻常（unusual）。此外，将数据包头部直接放在由网卡写入的数据包数据旁边似乎是一个奇怪的想法。

因此，我们采用了不同的方法，如图5-7所示。数据包缓冲区不包含任何头部，只包含数据包数据，它们的物理地址存储在内存池中的一个向量中。内存池中仍然有一个堆栈用于跟踪数据包缓冲区。每当数据包返回给用户时，都会实例化一个`Packet`结构体，其中包含所有必要的信息：指向数据包数据的指针、数据包数据的物理地址、数据包的大小、对内存池的引用以及数据包缓冲区在内存池中的位置。这个 Packet 结构体在图5-8中展示。

值得注意的是，对内存池的引用不是像`&Mempool`这样的正常引用，而是`Rc<RefCell<Mempool>>`。这样做有多种原因。首先，我们必须能够通过数据包（例如，当丢弃一个数据包时）修改引用的内存池。由于有多个数据包引用同一个内存池，而 Rust 在编译时不允许多个`&mut Mempool`，因此我们使用`RefCell<T>`，这是一个动态检查借用规则的可变内存字段。`RefCell<Mempool>`允许我们在运行时通过调用`pool.borrow_mut()`来可变地借用内存池。如果我们试图两次可变地借用同一个内存池，它会引发 panic。这种即使在存在不可变引用时也能修改数据的模式在 Rust 中被称为内部可变性（*Interior mutability*）。

```
fn forward(
    buffer: &mut VecDeque<Packet>,
    rx_dev: &mut impl IxyDevice,
    rx_queue: u32,
    tx_dev: &mut impl IxyDevice,
    tx_queue: u32,
) {
    let num_rx = rx_dev.rx_batch(rx_queue, buffer, BATCH_SIZE);

    if num_rx > 0 {
        // touch all packets for a realistic workload
        for p in buffer.iter_mut() {
            p[48] += 1;
        }

        tx_dev.tx_batch(tx_queue, buffer);

        // drop packets if they haven ' t been sent out
        buffer.drain(..);

    }

}
```

图 5-9 ixy 驱动转发器应用程序中的 forward()

其次，由于内存池需要在使用者发送或接收时以及还有任何数据包引用该内存池时（这可能会比IxgbeRxQueue存在的时间更长）一直存在，因此我们不能只有一个内存池的所有者。为了有多个内存池的所有者，我们使用Rc<T>，这是RefCell<Mempool>的引用计数器。当Rc<T>计数的引用数量减少到零时，内存池将被释放。

由于 Packet 内部有Rc<RefCell<Mempool>>，因此我们必须使用这种构造来引用驱动程序内部和外部的所有内存池变量。这就是为什么Mempool::allocate()返回的是Rc<RefCell<Mempool>>而不是Mempool的原因。

Packet结构体实现了 Rust 标准库中的三个特性：Deref、DerefMut和Drop。图5-9展示了转发器应用程序中forward()函数对这些特性的使用。我们实现了Deref和DerefMut，以便将Packet视为字节切片 (&[u8]) 来访问和修改数据包的数据。第 8 行显示了应用程序如何读取数据包p的第 49 个字节并将其增加 1。我们

实现了Drop，以便在数据包超出作用域时将数据包的缓冲区返回给内存池。在第 14 行，从缓冲区中移除了所有数据包。

与Deref和DerefMut特性的slice::from_raw_parts()返回的切片一样，对Packet结构体的原始指针进行解引用是不安全的。然而，由于我们确保返回的切片大小不超过相应的缓冲区，并且一次只有一个对象使用缓冲区，因此我们可以提供一个安全的抽象并实现这些特性。

## 5.7 接收数据包

在设备初始化期间，驱动程序将所有接收队列描述符填充为数据包缓冲区的物理指针。接收和传输描述符队列实际上是可由驱动程序和设备访问的环形队列。驱动程序控制环形队列的尾指针，而网卡控制头指针。当接收到数据包时，会实例化一个指向内存池中相应缓冲区的Packet结构体，将空闲缓冲区的物理地址存储在接收描述符中，并将该描述符的就绪标志重置。由于接收描述符不包含数据缓冲区的虚拟地址，我们必须跟踪哪个缓冲区属于哪个描述符。这是通过使用对缓冲区的Vec<usize>引用完成的，其中缓冲区的索引等于描述符环中描述符的索引。因此，Vec 充当引用相应缓冲区的环的副本。

为了提高性能，接收和传输数据包是以批处理的方式进行的。用户将VecDeque<T>（Rust 中队列的原生类型）传递给接收函数，以便接收函数可以将接收到的数据包推送到该队列中。这与 C 语言实现不同。在 C 语言中，用户将数组传递给函数，函数将接收到的数据包的缓冲区指针放入该数组中。在 Rust 中，我们将Packet结构体推送到队列中。我们也可以使用数据包的引用，但是通过返回数据包，我们可以明确地将数据包的内存所有权传递给用户。只要用户持有某个Packet，他就是该Packet及其内存的唯一所有者。当用户通过调用发送函数或丢弃它，将数据包返回给驱动程序时，数据包的内存将被释放，即通过将缓冲区引用推送到内存池的空闲堆栈中，将其返回给内存池。如果用户只持有数据包的引用，则必须显式地将该引用返回给驱动程序，因为没有办法验证引用是否已被丢弃。

## 5.8 传输数据包

发送数据包的工作方式与接收数据包类似，但更为复杂，因为发送数据包是异步的。当要发送数据包时，将传输描述符环中当前索引处的描述符更新为数据包数

据的物理地址，即内存池中的缓冲区，以及数据长度，并将缓冲区的条目 ID 存储在 `VecDeque<usize>`中。传输队列必须记住哪些缓冲区仍在使用中，否则缓冲区将过早返回给内存池，并且在实际数据发送出去之前可能会被重用，从而发送了意外的数据。

因此，发送函数由两部分组成：验证哪些数据包已经发送出去，并将相应的缓冲区返回给内存池（这称为 cleaning），以及发送新的数据包并存储它们的缓冲区以便清理。为了减少昂贵的 PCIe 传输的数量，发送队列的清理是批量进行的。

# 第 6 章 评估

为了评估我们的实现性能，我们在 29.76 Mpps 的完全双向负载下运行转发器应用程序，这等于两个 10 Gbit/s 连接的最小数据包行速率。为了模拟实际的工作负载，我们修改了每个数据包的一个字节。因此，至少有一个字节被加载到处理器的 L1 缓存中。我们将我们的实现与 C 语言参考实现（commit `d89d68b`）、Maximilian Stadlmeier 编写的 C# 实现（comomit `484485b`）和 Sebastian Voit 编写的 Go 实现（commit `4145aa8`）进行了比较。由于双端口 NIC 的性能不如单端口 NIC（可能是由于 PCIe 连接的硬件限制），因此我们在两个单端口 Intel X510（基于 82599）NIC 上进行了所有测试。

## 6.1 吞吐量

我们在不同的 CPU 频率下运行转发应用程序，以测量我们实现的总体性能并识别可能的瓶颈。图6-1比较了我们在 C、C# 和 Go 中的实现与 ixy 的吞吐量。基准测试表明，随着 CPU 频率的增加，不同实现之间的相对差异变得越小。C 和 Rust 之间最大的性能差异出现在 2.6 GHz，此时 C 实现的性能比 Rust 快约八分之一。在不超频的情况下，CPU 全速运行时，这两种实现之间的差异降至 3%。使用动态超频，即英特尔 Turbo Boost 功能，允许 CPU 加速到高达 3.6 GHz，C 和 Rust 版本之间没有明显的性能差异。为了转发数据包，C 版本实现最少需要 100 个 CPU 周期，而 Rust 实现至少需要 108 个 CPU 周期。

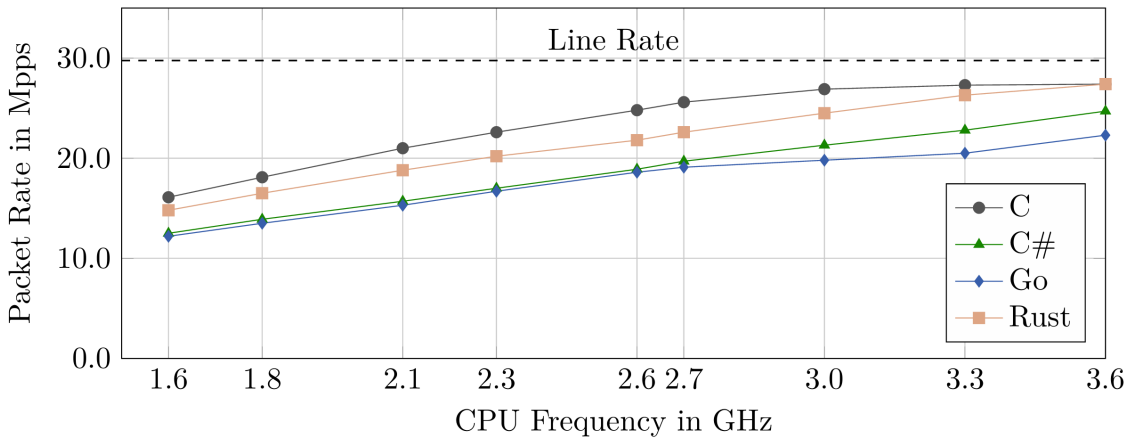


图 6-1 在改变 CPU 频率和批处理大小为 32 个数据包的情况下，双向单核转发性能。

有趣的是，在转发数据包时不修改它们对实现方式有不同的影响。当在 3.3GHz 的本机 CPU 时钟速率下不接触数据包时，C 版本的速度出人意料地稍微慢一些，而 Rust 版本的速度至少快 3%。

我们实现高性能的主要原因在于，一方面，在于通用的驱动程序设计；另一方面，则是使用的数据结构。通过使用内存池中已分配的缓冲区，避免在接收和发送函数中分配内存，并记住所有这些缓冲区的物理地址，而不是每次将虚拟地址转换为物理地址时都重新打开`/proc/self/pagemap`，这也会带来很大的性能差异。

与 C 版本不同，我们通过将数据包包装在`Packet`结构体中，为数据包提供了一个安全的接口。不幸的是，移动结构体而不是简单的原始指针会对性能产生负面影响。我们通过使用高效的数据结构，如`Vec<T>`和`VecDeque<T>`，来补偿这些性能损失，因为它们分别为`push()`和`pop()`、`push_back()`和`pop_front()`操作提供了 $O(1)$ 的摊还成本。

在 commit `6cbfac6`之前，接收和发送队列的内部堆栈和队列，即`IxgbeRxQueue`和`IxgbeTxQueue`，使用`Packet`结构体来存储当前队列正在使用的所有数据包。通过将结构体替换为指向内存池中相应缓冲区的单个`usize`，性能得到了极大的提升，每秒几乎可以处理四百万个数据包。

为了防止频繁的调整大小和复制操作，在已知所需最大容量的情况下，所有堆栈和队列都会以该容量进行分配。这就是我们的驱动程序中所有堆栈和队列的情况。

由于我们所有的内存池引用都使用了`Rc<RefCell<Mempool>>`，因此访问这些内存池的成本相当高。引用计数器必须增加和减少，并且`RefCell<Mempool>`必须验证一次只有一个可变引用指向一个内存池（类似于互斥锁）。为了在清理传输队列时减少访问内存池的次数，我们强制要求通过同一队列发送的数据包属于同一个内存池。因此，对于一批数据包，只需访问一次内存池就足够了。

为了提高内存池访问的性能，我们试图用普通引用 (`&RefCell<Mempool>`) 替换引用计数器，这要求我们必须为指向内存池的引用以及包含引用或引用内存池的所有结构体指定生命周期。不幸的是，由于 Rust 编译器中尚未修复的 bug，即关于`set_reg32()`方法的 issue 21906[1]和关于`reset_and_init()`方法的 issue 51132[2]，这意外地延长了一些值的作用域。因此，当这些问题得到解决后，将这些引用更改

[1]https://github.com/rust-lang/rust/issues/21906

[2]https://github.com/rust-lang/rust/issues/51132

为普通引用可能是驱动程序未来的工作。

## 6.2 批处理

批处理对性能有很大影响。当访问队列索引寄存器时，接收和发送数据包需要昂贵的 PCIe 往返时延。图6-2和图6-3显示了在不同批处理大小下，在最低和最高可能的 CPU 频率下性能如何受到影响。不幸的是，在 3.3 GHz 下，C# 实现会在批处理大小为 256 个数据包时崩溃，因此我们无法确定它是否可以达到与 C 和 Rust 相同的包速率。

可以指出，每批最多 64 个数据包的批处理大小对所有实现都导致更好的总体性能。每批大于 128 个数据包的批处理大小对性能的提升非常小，因为缓存未命中的数量也增加了。因此，批处理大小既不应选择太小也不应选择太大。对于我们的实现，每批 32 或 64 个数据包的批处理大小似乎是合理的。

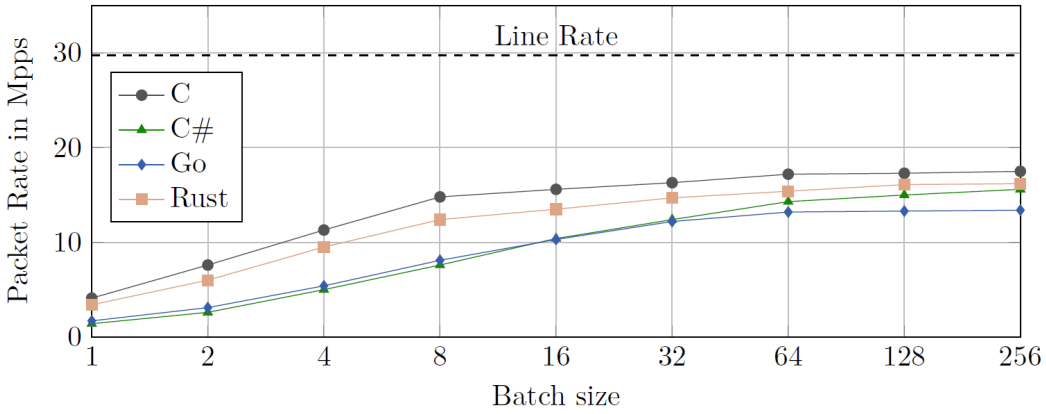


图 6-2 1.6 GHz 的频率下，不同 batch size 的双向单核心转发性能

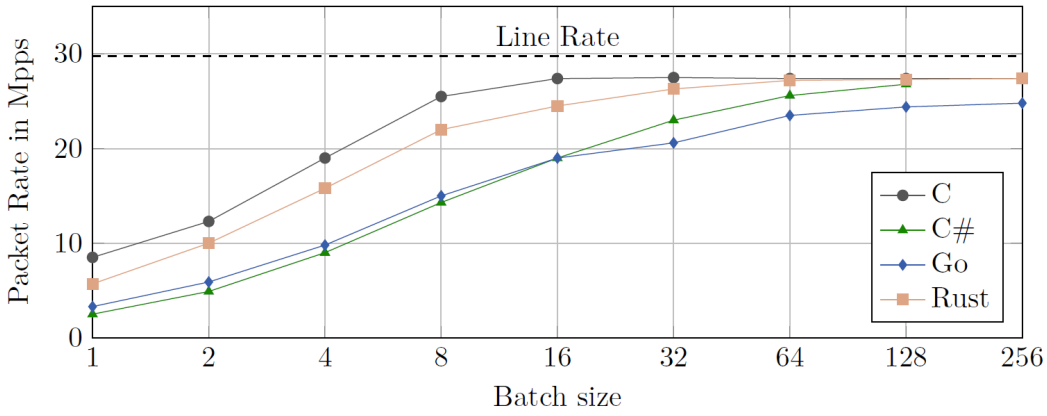


图 6-3 3.3 GHz 的频率下，不同 batch size 的双向单核心转发性能

## 6.3 性能分析

| **Application** | RX | TX | Forwarding | Memory Mmgt. |
|---|---|---|---|---|
| Rust Forwarder | 43.8 | 33.5 | 21.3 | 7.0 |
| C Forwarder | 39.8 | 16.9 | 22.0 | 21.0 |

图 6-4 用 CPU 周期表示的每个 packet 的处理时间

为了分析我们的转发器应用程序，我们在 CPU 最低速度 1.6GHz 下，以每批 32 个数据包的默认批次大小运行 perf，以确保 CPU 是瓶颈。图6-4显示了每个函数花费了多少 CPU 周期，并将其与 C 实现进行了比较。

在这两个驱动程序中，接收函数的速度要慢得多，因为它必须将数据提取到 L1 缓存中，而其他所有函数都在缓存上操作。在 C 中，内存管理的开销很大，尽管它仍然远低于 Linux 内核的 100 个 CPU 周期。Rust 内联了许多函数，因此内存管理所需的 7 个 CPU 周期必须谨慎对待。

DPDK 每个 packet 需要 61 个 CPU 周期，从这个角度看，我们的实现仍有改进空间。然而，事实是，DPDK 以增加复杂性的代价支持几乎所有硬件卸载功能，而 ixy 仅使用 CRC 校验和卸载（offloading）。

## 6.4 unsafe code

我们的 ixy 驱动程序实现包含了一定数量的不安全代码行。图6-5将我们实现中使用的不安全代码量与 Redox 中的另外两个 Rust 网络驱动程序 e1000 和 rtl8168 驱

动程序中的不安全代码量进行了比较，这两个驱动程序在第 7 章中进行了更详细的介绍。

| **Driver** | Speed | Lines of Code | Unsafe Code | %Unsafe |
|---|---|---|---|---|
| ixy | 10 Gbit/s | 1306 | 125 | 9.57% |
| e1000(Redox) | 1 Gbit/s | 393 | 140 | 35.62% |
| rtl8168(Redox) | 1 Gbit/s | 362 | 144 | 39.78% |

图 6-5 使用 Rust 编写的三种不同网络驱动程序的 unsafe 代码行数

结果清楚地表明，我们的实现使用的不安全代码比例低于其他两个驱动程序。事实上，虽然 ixy 由 10% 的不安全代码组成，但 e1000 和 rtl8168 几乎由 40% 的不安全代码组成。有趣的是，总代码量最多的驱动程序包含的最少的不安全代码，而总代码量最少的驱动程序包含的最多的不安全代码。

然而，许多不安全的代码行并不一定表明代码质量差或程序不安全。Rust 标准库的大部分由不安全代码组成，但 Rust 仍被认为是一种安全的编程语言。因此，有必要根据具体情况评估不安全代码的使用是否恰当。

在 Redox 驱动程序的情况下，它们大量使用不安全代码，在总共约 400 行的代码中，约有 150 行是不安全代码。查看其源代码，我们怀疑 Redox 的开发者优先考虑了这些标准驱动程序的实现，而不是尽量减少`unsafe`代码的使用和提供安全的抽象，因此将过多的代码声明为`unsafe`。

然而，我们必须指出，减少 unsafe 码的数量应该是每个开发者的目标，因为即使只有一行错误的 unsafe 代码，也可能导致未定义的行为。

# 第 7 章 背景以及相关工作

Rust 仍然是一种相对较新的编程语言。自从三年前发布第一个稳定版本以来，Rust 程序员和项目（尤其是与低级编程相关的）并不多。尽管如此，目前至少有三种操作系统是用 Rust 开发的：Blog OS、intermezzOS（受 Blog OS 影响很大）和 Redox。Blog OS 和 intermezzOS 都是教学操作系统，提供了大量文档，使读者能够自己编写操作系统。另一方面，Redox 是一个真正的操作系统，“旨在将 Rust 的创新引入现代微内核和全套应用程序中”。Redox 也提供了大量文档，但没有教学操作系统的文档详细。然而，作为一个可以工作的操作系统，Redox 包含实际驱动程序。因此，我们将在下一节中详细介绍 Redox。

## 7.1 Redox

Redox 是一个完全用 Rust 编写的操作系统。它由 Jeremy Soller 创建，于 2015 年 4 月 20 日首次发布，目前仍由大约 25 名开发人员积极开发。Redox 旨在通过提供一个功能齐全的 Unix 风格的微内核来确保安全、自由和可用。它目前支持所有 x86-64 CPU。

目前，Redox 包含 e1000 和 rtl8168 系列网络适配器（NIC）的驱动程序。阅读这两个驱动程序的源代码很有趣，因为它们采用了不同的方法，似乎是迄今为止唯一用 Rust 编写的网络驱动程序。不幸的是，关于这些驱动程序的文档还不多。虽然有一本由 Redox 开发者写的关于 Redox 的在线书籍，但是关于驱动的那一章仍然是空白的。

查看源代码，这两个驱动程序之间存在一些概念上的差异。虽然 e1000 驱动程序使用常量来设置 NIC 的寄存器，就像我们的实现一样，但 rtl8168 驱动程序使用了一个特殊的寄存器结构，该结构由用户定义的内存映射 I/O（MMIO）类型的数组定义的寄存器组成。

rtl8168 在 MMIO 类型上实现了读取和写入方法，用于从寄存器读取和写入数据，而 e1000 则使用与我们的实现几乎完全相同的`read_reg`和`write_reg`功能。

这两个驱动程序都实现了一个设备结构体，其中包含诸如`new`（返回尚未初始化的设备）、`init`（重置并初始化设备）或`next_read`（返回接收队列中下一个数据包的大小）等方法。它们还为这个设备结构体实现了`SchemeMut` trait，其中包

含`read`和`write`函数。与我们的驱动程序不同，这些函数会引用一个缓冲区，并将数据包数据从 DMA 内存复制到该缓冲区，然后再复制回来。因此，我们可以肯定地说，这些驱动程序的性能不如我们的实现。

然而，这两个现实世界中的驱动程序证明，在 Rust 中编写网络驱动程序不仅是一个深奥的科学理念，而且是一个实用且可行的任务，这个任务已经被实现了。

# 结　论

我们实现了用 Rust 编写的用户空间网络驱动程序，以展示 Rust 语言的具体优缺点，并回答 Rust 是否适合作为网络驱动程序的编程语言的问题。我们的实现代码以及参考实现的代码都可以在 GitHub 上找到。

正如第三章所明确指出的，对于网络驱动程序的编程语言来说，最重要的要求之一是性能。虽然这可能不适用于所有类型的驱动程序，但对于网络驱动程序来说，这确实是如此。因此，一个合理的驱动程序应该具有与 C 或 C++ 实现相近的性能，这两种语言以其高效和广泛的使用而闻名。第六章的性能测量表明，Rust 能够满足这一要求。尽管我们的实现比 C 语言的参考实现稍逊一筹，但它仍然足够快，可以在实际应用中使用，因为它比默认的内核网络堆栈快 6 倍以上。

除了性能之外，Rust 的另一个巨大优势在于其内存安全性。尽管像任何人为制造的对象一样，Rust 并非 100% 完美——标准库中存在大量不安全的代码，而且不时还会出现一些漏洞，比如最近关于`str::repeat()`的安全漏洞[1]——但 Rust 通过强制实施严格的内存处理规则，在提升计算机程序的安全性方面做得非常出色。

无论使用哪种编程语言，每 1000 行代码平均包含 15-50 个错误[7]。Rust 代码中的编程错误可能不会更少，但至少它们不太可能导致未定义的行为并危及我们的系统——就像 C/C++ 中的程序那样。

未来对驱动程序的改进可能包括实现 VirtIO 版本，以及对整体性能进行小幅改进。此外，还可以实现 ixgbe 驱动的多线程版本。我们的大部分代码，如大页面上的内存分配，应该已经是线程安全的，因此这可能是近期可以实现的目标。

根据我们的发现，我们可以得出结论，Rust 是一种非常适合编写网络驱动程序的编程语言。它不仅是一种快速且安全的系统编程语言，还在 2016 年、2017 年和 2018 年被评为最受喜爱的编程语言。使用 Rust 编写更多的驱动程序肯定会使计算机系统更加安全和可靠。

[1] https://blog.rust-lang.org/2018/09/21/Security-advisory-for-std.html

# 参考文献

[1] Pfaff B, Pettit J, Koponen T, et al. The design and implementation of open {vSwitch}[C]//12th USENIX symposium on networked systems design and implementation (NSDI 15). 2015: 117-130.
[2] Kohler E, Morris R, Chen B, et al. The Click modular router[J]. ACM Transactions on Computer Systems (TOCS), 2000, 18(3): 263-297.
[3] Gorrie L, et al. Snabb: Simple and fast packet networking[J]. URL https://github. com/snabbco/snabb, 2012.
[4] Blandy J, Orendorff J, Tindall L F. Programming Rust[M]. ” O’Reilly Media, Inc.”, 2021.
[5] Klabnik S, Nichols C. The Rust programming language[M]. No Starch Press, 2023.
[6] Intel I. 82599 10 GbE controller datasheet[Z]. 2019.
[7] McConnell S. Code complete[M]. Pearson Education, 2004.

# 附　录

## 附录 A 英文缩略词表

| | |
|---|---|
| BAR | Base address register |
| CPU | Central processing unit |
| DMA | Direct memory access |
| IOMMU | I/O memory management unit |
| MMIO | Memory-mapped I/O |
| NIC | Network interface card |
| OSI | Open Systems Interconnection |