基于 openEuler 的命令行网络请求工具（EuNet）

一、项目背景与总体目标

本项目源于课程竞赛要求，围绕“基于 openEuler 的命令行网络请求工具（EuNet）”这一主题展开。项目运行环境为 openEuler，所有设计与实现均基于该前提展开。

在立项阶段，团队即对项目目标进行了明确界定：本项目并非追求功能数量的堆叠式实现，而是希望在有限功能范围内，完成一次结构清晰、过程可追溯、可多次调整的软件工程实践。

因此，项目整体关注点始终集中在：

- 工程结构是否清晰、稳定
- 模块职责是否单一、边界是否明确
- 在多轮修改、重构和功能引入后，系统整体是否仍保持可控性

基于上述目标，项目确立了“先结构、后功能；先稳定、再扩展”的总体推进策略。本文档作为过程性文档，记录项目从立项、设计、实现、调整到趋于稳定阶段的关键工程决策与演进过程，而非单一版本说明。

二、总体设计思路与系统结构形成过程

2.1 初期设计取向

项目初期阶段，团队并未立即围绕具体协议或工具功能展开实现，而是优先对整

体系统结构进行拆分与规划。其核心出发点在于：

- 避免底层系统调用直接散落在业务逻辑中
- 避免网络通信细节与工具逻辑高度耦合
- 为后续功能试验、调整与重构预留空间

在此阶段，设计重点不在于“做什么功能”，而在于“未来功能应当如何被安放在结构中”。

2.2 结构演进与固化过程

随着实现推进与多轮调整，系统逐步形成了自底向上的分层结构，并在后续修改中不断被验证与固化。当前系统稳定采用如下层次结构（由下至上）：

1. openEuler 底层能力封装层
2. 网络通信与连接管理层
3. 核心调度与状态管理层
4. 请求与工具逻辑层
5. 终端交互展示层（TUI）

该结构并非一次性设计完成，而是在功能引入、问题修复与重构过程中不断调整、验证并最终稳定下来的结果。

三、openEuler 底层能力封装层设计

3.1 设计背景

在项目早期尝试中，若直接在上层逻辑中使用 socket、epoll 等系统接口，容易引发以下问题：

- 资源生命周期分散，释放责任不清晰
- 错误处理逻辑重复且难以统一
- 上层逻辑被迫感知系统调用细节，增加理解和维护成本

为控制复杂度并保持结构稳定性，项目将与 openEuler 底层系统直接相关的能力集中进行封装。

3.2 模块职责划分

该封装层主要负责：

- 文件描述符的封装与生命周期管理（如 Fd / FdView）
- Socket 基础操作的统一封装
- 事件轮询器（如 epoll）的集中管理
- 网络地址与端点的表示与解析

设计目标并非进行抽象层级的复杂堆叠，而是通过最小必要封装，减少上层模块对系统调用细节的直接依赖，并为后续日志、调试或功能扩展预留稳定接口。

3.3 实际工程效果

在后续多次重构与功能调整过程中，该层接口保持稳定，上层模块无需因底层实现变化而修改逻辑，显著降低了维护成本。

四、网络通信与客户端模型的形成与调整

4.1 初始实现中暴露的问题

项目早期的 TCP 客户端实现较为直接，同步通信逻辑与连接管理混合在一起。随着功能增加，该实现逐渐暴露出：

- 连接生命周期不清晰
- 状态判断逻辑分散
- 修改某一逻辑路径易对其他路径产生连锁影响

4.2 中期重构过程

针对上述问题，项目在中期对网络通信部分进行了集中整理与重构：

- 将连接能力拆分为独立的 Connection 模块
- 抽象并统一 TCP / UDP 的基础使用方式
- 上层客户端仅通过统一接口与连接对象交互

该阶段的调整重点在于消除隐式状态与不透明行为，使连接生命周期与状态转移更加清晰。

4.3 当前状态评估

该通信结构在后续 HTTP 请求能力引入过程中保持稳定，上层逻辑几乎未发生结构性调整，验证了该设计在复杂协议场景下的可扩展性与合理性。

五、HTTP 请求能力的引入过程

5.1 引入动机

在底层通信能力与客户端模型趋于稳定后，项目引入 HTTP 请求能力，用以验证整体结构在真实应用协议场景下的适用性。

5.2 实现方式

HTTP 相关逻辑通过独立模块完成，主要包括：

- HTTP 请求与响应数据的组织
- 请求生命周期的管理

该模块仅依赖既有客户端接口完成数据收发，不直接涉及底层连接与系统调用细节。HTTP 模块被视为一种“格式化与协议处理工具”，而非系统核心依赖。

5.3 实际问题与修复记录

在 HTTP 功能调试过程中，陆续发现并修复了多项问题，包括但不限于：

- 无参数启动时触发的异常路径问题
- 在连接被对端提前关闭（peer closed）情况下的错误处理缺陷

这些问题的定位与修复依赖于前期形成的分层结构与统一错误处理机制，未引发大范围结构调整。

六、通用格式化工具的引入

在项目早期开发阶段，为提升调试输出与错误信息的可读性，项目引入 fmt 库作为统一的字符串格式化工具。该库主要用于：

- 构造错误信息
- 输出调试信息
- 生成命令行反馈文本

fmt 的使用避免了大量手写字符串拼接带来的可读性下降问题，同时使不同模块输出风格保持一致，为后续接入日志系统或完善调试机制提供了良好基础。

七、统一错误与结果模型的引入

7.1 设计动机

随着模块数量和逻辑复杂度提升，仅通过返回值或隐式约定已难以支撑复杂路径下的错误传播与问题定位。

7.2 工程意义

项目引入统一的 Result / Error 模型后：

- 所有可能失败的操作均显式返回结果
- 降低 C++ 隐式异常或未定义行为带来的风险
- 统一成功与失败路径的表达方式
- 减少大量分支判断与特判逻辑

该模型在后续模块中被持续复用，未出现结构性冲突，成为系统稳定性的重要基础之一。

八、核心调度与状态管理机制

在系统核心层中，引入集中调度组件，对各模块协作过程进行统一管理，其主要目标在于保证执行流程的有序性与可控性。

具体体现在：

- 通过阻塞且有序的时间线，避免事件处理顺序混乱
- 使用请求周期状态机，自动完成状态转移，减少人工判断
- 由调度器集中分发事件，统一事件传递路径，降低模块间耦合

该机制在系统逻辑复杂度提升后，依然保持了清晰、可追踪的执行流程。

九、终端交互与展示层（TUI）

项目采用终端用户界面（TUI）形式作为交互入口，引入 FTXUI 库实现界面展示。

该层仅负责：

- 命令输入解析
- 请求结果展示

不直接参与网络请求与调度逻辑，确保展示层调整不会影响系统核心行为。

十、开发过程阶段性记录（更新）

阶段一：结构规划与环境准备

完成 openEuler 环境搭建、构建系统配置与基础目录规划，明确项目边界与目标。

阶段二：底层能力封装与通信基础

围绕文件描述符、Socket 与事件轮询机制反复调整实现，逐步形成稳定的底层封装层。

阶段三：客户端模型整理与 HTTP 引入

重构网络客户端结构，引入 HTTP 请求能力，并在调试过程中修复连接关闭、异常路径等实际问题。

阶段四：调度机制与错误模型完善

引入统一调度与 Result / Error 模型，提升系统在复杂执行路径下的稳定性与可维护性。

阶段五：文档对齐与工程规范整理

在功能趋于稳定后，对 commit 记录、CHANGELOG 与过程性文档进行系统梳理，明确不同文档的定位与职责，保证工程记录的一致性与可追溯性。

十一、总结

EuNet 项目在限定 openEuler 环境下，通过循序渐进的工程实践，逐步形成了结构清晰、职责明确的命令行网络请求工具框架。

项目在多次功能引入与重构过程中始终保持整体结构稳定，验证了前期设计思路的合理性，也使团队成员完整经历了从设计、实现、调整到规范化整理的软件工

程过程。