ClipBridge / CB — 项目背景总览（贴给新会话用）

0) 项目简介

名称：ClipBridge（简称 CB）

定位：跨平台剪贴板同步工具，先 局域网 (LAN)，后期支持 跨外网 (WAN)。

核心卖点：Lazy Fetch 按需取用——复制时只广播元数据，粘贴时才向源设备拉取正文（文本/图片/文件），节省网络与功耗。

目标平台：Windows（先做外壳 MVP）→ Android（外壳）→ v2 核心（剪贴板历史与同步）→ macOS / Linux（外壳）。

---

1) 功能与阶段目标

v1（Windows MVP + 基础核心）

局域网自动发现（mDNS/Bonjour）

端到端加密连接（自签证书 + 指纹固定 / 或 Noise）

Lazy Fetch：复制时广播“元数据”，粘贴时拉取正文

支持类型：文本、图片、文件（文件/图片按需拉取）

Windows 外壳：托盘图标、主窗口（登录/选择设备/基础设置）、暂停/退出菜单

全局热键：默认 Ctrl+Shift+V（Win+V 为系统保留）

v2（核心升级 + 历史）

剪贴板历史：本地持久化（SQLite/KV），在各设备同步历史元数据；粘贴时再取正文

历史小窗（快捷键呼出）与主窗历史页

过滤与安全：敏感内容排除、设备 ACL、到期时间

未来

WAN 直连/中继（QUIC + 自建信令/中继）

macOS / Linux 外壳

设备分组、权限与密钥轮换

---

2) 架构与协议（简版）

2.1 分层

Rust 核心 (Core)

发现与会话、加密、传输（HTTP/3/QUIC 或 gRPC/HTTP2）、缓存（LRU + sha256 去重）、元数据/正文编解码、Lazy Fetch 请求处理、事件回调。

平台外壳 (Shell)

集成系统剪贴板与 UI（Windows：C++/WinUI 3；Android：Java；macOS：Swift/ObjC；Linux：Qt/GTK 之后再定）。调用核心 API，接收核心事件。

2.2 数据流

复制：外壳监听系统剪贴板 → 解析类型/大小/哈希 → 调核心广播元数据给配对设备。

粘贴：外壳以“延迟渲染/承诺数据”的方式持有剪贴板 → 粘贴触发 → 调核心按需向源设备拉取正文 → 回填系统剪贴板 → 应用完成粘贴。

2.3 元数据结构（示例）

{

"protocol\_version": 1,

"item\_id": "uuid",

"owner\_device\_id": "device\_A",

"types": ["text/plain", "image/png", "application/octet-stream"],

"size": 1048576,

"sha256": "…",

"expires\_at": 1690000000,

"preview": "文本前若干字/缩略图hash",

"uri": "cb://device\_A/item/uuid",

"created\_at": 1690000000

}

2.4 发现 / 连接 / 安全

发现：mDNS/Bonjour，广播服务 \_lanclip.\_tcp

连接：QUIC(HTTP/3，Rust quinn) 或 gRPC(HTTP/2，tonic)

加密：TLS 1.3（自签 + 指纹固定）或 Noise（x25519）

配对：首次扫码/指纹确认 → 建立信任（持久化设备公钥）

2.5 平台要点

Windows：Win32 延迟渲染（SetClipboardData + WM\_RENDERFORMAT/WM\_RENDERALLFORMATS、CF\_UNICODETEXT、CF\_DIB、CF\_HDROP），C++/WinUI 3 外壳。

Android（Java 外壳，后继）：

前台服务 + 常驻通知；

通过 ContentProvider URI 提供大内容（粘贴时触发拉取）；

JNI 调 Rust FFI .so（非必须先做）。

---

3) 技术选型与接口

3.1 语言/框架

核心：Rust（tokio、quinn/tonic、rustls、mdns、lru 等）

Windows 外壳：C++ + WinUI 3（C++/WinRT），必要处使用 Win32 API

Android 外壳：Java（UI 设计器），JNI 连接 Rust（后做）

3.2 Core ↔ Shell 接口（方向）

Shell → Core：cb\_init(config) / cb\_send\_metadata(meta) / cb\_request\_content(item\_id, mime) / cb\_pause(bool) / cb\_shutdown()

Core → Shell（回调）：on\_device\_online(info) / on\_device\_offline(id) / on\_new\_metadata(meta) / on\_transfer\_progress(id, sent, total) / on\_error(code,msg)

---

4) 仓库与代码结构（当前状态）

ClipBridge/

├── core/ # Rust 核心库（crate 名称：core）

│ ├── Cargo.toml # crate-type 当前包含 ["rlib","cdylib"]

│ └── src/lib.rs # pub fn ping() -> &'static str { "clipbridge-core ok" }

│

├── platforms/

│ ├── windows/

│ │ └── core-ffi/ # Windows FFI 动态库（crate 名称：core-ffi-windows）

│ │ ├── Cargo.toml # 依赖 ../../../core，crate-type ["cdylib"]

│ │ └── src/lib.rs # 导出 extern "C" fn cb\_core\_ping() -> \*const c\_char

│ └── android/

│ └── core-ffi/ # Android FFI 动态库（crate 名称：core-ffi-android）

│ ├── Cargo.toml # 依赖 ../../../core，crate-type ["cdylib"]

│ └── src/lib.rs # 导出 extern "C" fn cb\_core\_ping() -> \*const c\_char

│

├── .github/workflows/ci.yml # CI：Rust / Windows / Android 三个 job（可并行）

├── .editorconfig # 通用编辑器规则（LF、末行换行、缩进宽度等）

├── rustfmt.toml # Rust 格式化规则（max\_width=100, 4空格, LF）

├── .clang-format # C++ 格式化（\*\*使用 Tab 缩进，宽度4\*\*）

├── .clang-tidy # C++ 静态分析（bugprone/modernize/performance 为主）

├── deny.toml # cargo-deny 依赖/许可证审计

├── .gitignore # Rust / VS / Android 常见产物忽略

├── Cargo.toml # \*\*workspace\*\* 根清单（members: core、两套 FFI）

└── LICENSE # 开源许可证（MIT 或 Apache-2.0 之一）

已验证（在仓库根）：

cargo metadata # OK

cargo build # OK（core、core-ffi-windows、core-ffi-android 都能编过）

---

5) Windows 外壳（WinUI 3）当前策略

模板：空白应用，已打包（桌面版中的 WinUI 3，C++/WinRT）

UI 骨架：左侧 NavigationView（设备 / 历史 / 设置）+ 右侧 Frame

托盘：Win32 Shell\_NotifyIcon，左键开主窗，右键菜单（暂停/退出）

热键：RegisterHotKey 默认 Ctrl+Shift+V（Win+V 为系统保留）

FFI 测试：运行时 LoadLibraryW("core\_ffi\_windows.dll") + GetProcAddress("cb\_core\_ping") 成功返回 "cb core ffi (windows) ok"

构建后复制 DLL到可执行目录（建议设为 VS 后期生成事件）

---

6) CI / 代码规范（已配置）

CI：.github/workflows/ci.yml

Rust（fmt/clippy/test/cargo-deny）

Windows（MSBuild 构建 WinUI 3 工程；可选先编 FFI 再复制 DLL）

Android（./gradlew -p platforms/android lint assembleDebug）

格式化与检查

.editorconfig（全项目）

rustfmt.toml（Rust）

.clang-format（C++，Tab=4）

.clang-tidy（C++ 静态分析）

deny.toml（漏洞与许可证审计）

---

7) 关键设计决策

按需取用（复制只发元数据；粘贴再取正文）

缓存：LRU + sha256 去重；图片可缓存缩略图；可设置上限

环路防止：元数据含 origin\_device\_id + 时间窗去抖

安全：端到端加密 + 设备配对指纹；可撤销信任

默认快捷键：Ctrl+Shift+V（可配置）

Windows 懒取实现：Win32 延迟渲染（WM\_RENDERFORMAT 等）；文件使用 CF\_HDROP，图片 CF\_DIB/PNG

Android 懒取实现（后续）：ContentProvider URI + Foreground Service

---

8) 你可以立即做的下一步

1. 在 VS 创建 WinUI 3 项目 platforms/windows/ClipBridgeShell/，先跑“空白窗”

2. cargo build -p core-ffi-windows --release，把 core\_ffi\_windows.dll 复制到 exe 同目录

3. 在 App.xaml.cpp 里按示例动态加载 DLL，调用 cb\_core\_ping 验证联通

4. 加入托盘与热键；主窗放设备列表页占位（后续填充核心事件）

5. 等 v1 骨干通了，再实现文本/图片/文件的 Lazy Fetch 全链路；随后做 v2 的历史存储与历史小窗

---

9) 许可证与合规（一句话）

建议 MIT 或 Apache-2.0（已在根 LICENSE）。

Rust 依赖用 cargo-deny 审计；不引入 GPL/AGPL。

未来如引入第三方二进制/图标等，保留出处与许可说明。

---

> 以上即本项目的完整上下文与当前状态。新会话请基于此继续：

生成/修改代码时，遵守现有文件结构与 CI 规则；

Windows 外壳沿用 C++/WinUI 3；

Rust 核心/FFI 的接口命名以 cb\_\* 为前缀；

保持 Lazy Fetch 架构与元数据协议不变；

默认快捷键 Ctrl+Shift+V，Win+V 不覆盖。