整体架构（一句话）

前端用 `fetch` 调后端 `/api/run_frida`（模板）或 `/api/run_frida_custom`（自定义），后端渲染/接收 JS 脚本，选择 **attach/spawn** 方式把脚本注入到目标 App，并把脚本里的 `send(...)` 日志通过 **Flask-SocketIO** 转发到页面"控制台输出"。

小白总结版： 前端点按钮 → 后端生成/接收 JS → 后端用 Frida 注入 → 日志回推给网页。

模板注入：从按钮到日志的完整链路

1) 前端触发

- 在会话详情右上角，"运行"按钮会 POST 到 `/api/run_frida`，携带参数：`{ id/app/spawn/(可选)template/template_params }`。
- 也支持"Memory / Frida Templates"区域直接选模板（如 `ssl_pinning_bypass`、`sharedprefs_dump`），同样 POST `/api/run_frida`。

> 前端在 socket 上监听 `frida_started` 与 `frida_log`，把后端推送的运行状态和脚本输出追加到右侧"控制台输出"。

2) 模板渲染（后端）

- 模板集中定义在 `app.py` 的 `FRIDA_TEMPLATES` 字典，含：
  - `ssl_pinning_bypass`：改写 `CertificatePinner.check` / HostnameVerifier / 替换 `SSLContext.init` 的 TrustManager / 兜底 `HttpsURLConnection`。
  - `sharedprefs_dump`：Hook `SharedPreferencesImpl` 的 `getAll/getString`、`EditorImpl.put*` 并解析 `mFile` 拿到真实 XML 文件路径。
- 统一通过 `safe_format(...)` 渲染（缺参时占位符不报错）。也提供 `/api/generate_frida` 直接生成脚本文本用于下载/排查。

3) 选目标进程（后端）

- 你的后端提供了 **Probe**：`/api/probe` 会调用 `frida-ps -Uai`，并通过 `pick_exact_pid_from_ps(...)` 在多进程场景（`:push` / `:remote`）尽量选中**主进程** PID。实操时也可先跑 Probe 再注入。

4) 具体注入（后端）

- 你在 `app.py` 为"Frida 运行"准备了 helpers（`FRIDA_SESSIONS` 等），并标注了"wrapper to forward console.log/error"，用于把脚本内 `send({__frida_console: true, args:[...]})` 的消息**转换为 SocketIO 的** `frida_log` 事件——前端可直接显示。
- 若 Python 侧能 `import frida`，优先用 **Frida Python API**（`device.attach` / `spawn` + `session.create_script` + `script.on('message', ...)`）；否则回退到 **CLI 子进程**（`subprocess.Popen(['frida', ...])`）读取 stdout/stderr 推送。你在代码里已经为"Python 可选导入"与"子进程 fallback"做了准备。

> **你前端控制台里看到的这些行**：
> `[frida_started] pid=... script=...`、`[bypass] SSLContext.init -> replace TrustManager`、`[SharedPreferences][getString] file= ...` … 都是脚本里的 `log(...) -> send(...)`，被后端转为 `frida_log` 发到浏览器。模板里每个关键点都有 `log(...)`。

小白总结版： 点"运行模板"→ 后端按模板渲染 JS → 选对 PID → 通过 Frida 注入 → 模板里的 `send(...)` 被后端转发成网页控制台日志。

自定义脚本注入：你写啥就注入啥

1) 前端触发

- "自定义 Frida 脚本"Tab 接收文本框里的 JS 和包名，勾选可选的 Spawn 模式，POST 到 `/api/run_frida_custom`。

2) 后端执行

- 后端直接使用你提交的 `script` 字符串，不做模板渲染；注入流程与上面的模板注入一致（同样 attach/spawn、同样把 `send(...)` 转发到 `frida_log`）。你的"示例脚本"按钮会预填一个 OkHttp 打点示例，执行后会看到 `[example] Java runtime OK`、`[example][okhttp] url=...` 等日志。

小白总结版： 你在网页贴 JS → 后端把这段 JS 原样注入 → 控制台实时回显。

为什么你的"这次输出"说明注入链路是正确的?

- **SharedPreferences**：模板在 `getString/getAll/Editor.put*` 都有打点；你的输出出现了：
  - `[SharedPreferences] hooks installed`（模板初始化日志）
  - 随后读取键值：`[getString] file=(unknown-file) username = demo_user` / `token = ...`
  - 写入键值：`[Editor.putString] username = demo_user`、`[Editor.apply]`
    → 这对应模板里的实现路径，且顺序合理（读 → 写 → 再读），只是**文件路径显示** `(unknown-file)`，这是模板中反射 `mFile` 的兜底值；从代码看反射失败会返回 `(unknown-file)`，并不影响 KV 的捕获。
- **SSL Pinning Bypass**：模板会：
  - 改写 OkHttp `CertificatePinner.check`（含 `$okhttp` 变体）→ 日志：`[bypass] CertificatePinner.check$okhttp #0 -> bypass`
  - 放宽 `OkHostnameVerifier.verify` → 日志：`verify host=httpbin.org -> true`
  - 替换 `SSLContext.init` 注入 TrustAllManager → 日志：`[bypass] SSLContext.init -> replace TrustManager`
    → 你的一组输出刚好一一对应这些分支，说明脚本确实注入并命中相应路径。

小白总结版： 这些回显正是模板里设计的日志点，能看到它们 = 注入/Hook 成功。

关键实现点（Python 端）

- **模板仓库**：`FRIDA_TEMPLATES` 集中管理；其中 `ssl_pinning_bypass` / `sharedprefs_dump` 的 JS 逻辑就在这里。
- **渲染器**：`safe_format` 保守替换参数（缺键不报错，便于同一模板可多场景重用）。
- **探测与选 PID**：`/api/probe` + `pick_exact_pid_from_ps(...)`，避免误 attach 到 `:push/:remote`。
- **消息转发**：后端包装 `send({__frida_console:true,...})` 为 SocketIO `frida_log`，前端 `socket.on('frida_log', ...)` 统一打印。
- **回退策略**：`try: import frida`（可用则 Frida Python API；否则子进程 CLI），确保不同运行环境都能跑。

小白总结版： Python 这边负责"渲染脚本、找进程、注入、转发日志"，模板和自定义只在"脚本文本来源"不同。

你可以怎么复述"是如何通过 Python 实现的"

> "我们前端调用 Flask 的 /api/run_frida 或 /api/run_frida_custom。后端从 FRIDA_TEMPLATES 渲染或直接接收脚本，然后用 Frida Python API（可用时）或 subprocess 调 Frida CLI 的方式对目标包 **attach** 或 **spawn** 注入。脚本里所有 send({__frida_console:true, args:[...]}) 的消息会被后端包装后通过 **Flask-SocketIO** 推给前端，因此你在控制台能实时看到 [bypass] ...、[SharedPreferences] ... 等日志。Probe 接口和 PID 选择器保证我们附到主进程，不会误附到 :push/:remote 子进程。"
>
> （你可以指给导师看 app.py 的这些位置：模板集合、/api/generate_frida、Probe、FRIDA_SESSIONS、前端 socket.on('frida_log') 处理。）