L4 -- 工程实现与部署拓扑层

文档版本: 1.0 **适用项目**: sgClaw (业数融合一平台 AI Agent 底座) **编制日期**: 2026-03-03

读者: 开发者、DevOps、测试工程师 -- 需要搭建开发环境、理解构建流程、执行测试、部署发布。

---

1. 仓库结构

sgClaw 系统的代码分布在两个仓库中：独立的 sgClaw Rust 仓库和 SuperRPA Chromium 仓库。 前者承载 Agent 核心逻辑，后者承载浏览器侧集成代码和前端 UI。

1.1 sgClaw 仓库 (Rust 端)

```
/home/zyl/projects/sgClaw/
├── Cargo.toml                   # 项目配置、依赖声明、编译优化
├── Cargo.lock                   # 依赖版本锁定
├── src/
│   ├── main.rs                  # 入口: pipe I/O 初始化, tokio runtime 启动
│   ├── agent/
│   │   ├── mod.rs               # Agent 模块导出
│   │   ├── runtime.rs           # ZeroClaw ReAct 循环 (think -> act -> observe)
│   │   └── critic.rs            # 输出质量评估 + Circuit Breaker 熔断器
│   ├── pipe/
│   │   ├── mod.rs               # Pipe 模块导出
│   │   ├── protocol.rs          # JSON Line 编解码, 消息类型定义
│   │   ├── handshake.rs         # 版本协商 + HMAC 密钥交换
│   │   └── browser_tool.rs      # BrowserPipeTool: 自定义 Tool trait 实现
│   ├── skill/
│   │   ├── mod.rs               # Skill 模块导出
│   │   ├── loader.rs            # 技能发现 + 签名校验 + 沙箱加载
│   │   └── registry.rs          # 技能注册表 (名称 -> 技能映射)
│   ├── llm/
│   │   ├── mod.rs               # LLM 模块导出
│   │   ├── provider.rs          # Provider trait 定义 (统一接口)
│   │   ├── claude.rs            # Claude API 实现 (streaming)
│   │   ├── openai.rs            # OpenAI/兼容 API 实现
│   │   └── ollama.rs            # 本地 Ollama 实现
│   ├── memory/
│   │   ├── mod.rs               # Memory 模块导出
│   │   ├── short_term.rs        # Ring Buffer 短期对话记忆
│   │   └── long_term.rs         # SQLite + 向量存储 长期知识库
│   ├── security/
│   │   ├── mod.rs               # Security 模块导出
│   │   ├── mac_policy.rs        # 域名 / Action 白名单校验
│   │   └── hmac.rs              # HMAC-SHA256 消息签名与验证
│   └── config/
│       ├── mod.rs               # Config 模块导出
│       └── settings.rs          # 配置文件加载 (TOML / 环境变量)
```

```
├── skills/                          # 内置技能目录
│   ├── builtin/                     # 预置业务技能脚本
│   └── registry.json                # 技能清单 + 签名哈希
├── tests/
│   ├── integration/                 # 集成测试 (pipe 协议, 端到端)
│   └── fixtures/                    # 测试数据 (mock LLM 响应, JSON 样例)
├── docs/                            # 设计文档 (L0-L4 分层)
├── scripts/
│   ├── build-linux.sh               # Linux musl 静态链接构建脚本
│   ├── build-windows.sh             # Windows MSVC 交叉编译脚本
│   └── install-to-superrpa.sh       # 二进制部署到 SuperRPA 目录
└── .github/
    └── workflows/                   # CI/CD 流水线定义
```

1.2 SuperRPA 新增文件 (C++ 端)

在已有 SuperRPA 仓库中新增一个 `sgclaw/` 子目录，约 500-600 行 C++ 代码。 所有新增文件位于 `src/chrome/browser/superrpa/sgclaw/` 下，与已有子系统平级。

```
src/chrome/browser/superrpa/
├── BUILD.gn                          # [修改] 新增 :sgclaw source_set 依赖
├── sgclaw/                           # [新增目录]
│   ├── sg_claw_process_host.h        # 进程宿主: 生命周期管理
(Start/Stop/OnCrash)
│   ├── sg_claw_process_host.cc       # ~200-300 行, Singleton, 管道创建与进程启
动
│   ├── pipe_listener.h               # 异步读取循环: 解析 JSON Line 消息
│   ├── pipe_listener.cc              # ~150 行, base::FileDescriptorWatcher
│   ├── mac_whitelist_check.h         # Pipe 来源 MAC 校验: action + domain 白名
单
│   ├── mac_whitelist_check.cc        # ~100 行, 与已有 rules.json 集成
│   └── BUILD.gn                      # 子模块构建声明
```

已有文件的修改点 (最小侵入):

文件	修改内容	行数变更
`superrpa/BUILD.gn`	deps 新增 `":sgclaw"`	+1 行
`ui/ui_page_controller.cc`	新增 Agent 启停的 FunctionsUI handler	+~50 行

1.3 Side Panel 新增文件 (前端)

前端变更位于 agent-vue 应用中，新增一个 Vue 组件用于 Agent 控制面板。 agent-vue 基于 Vue 2.6 + Element UI，构建产物部署到 Side Panel 的 agent/ 目录。

```
agent-vue/src/
├── components/
```

```
│   └── SgClawControl.vue          # [新增] Agent 启停控制 + 状态展示 + 任务输
入
```

构建产物部署位置:

```
src/chrome/browser/resources/superrpa/panels/apps/agent/
├── css/
├── js/
├── index.html
└── ...                            # npm run build:production 输出
```

---

2. 构建系统

2.1 Rust 构建 (sgClaw)

2.1.1 Cargo.toml 核心配置

```
[package]
name = "sgclaw"
version = "0.1.0"
edition = "2021"
description = "AI Agent engine for SuperRPA browser"

[dependencies]
zeroclaw = { version = "0.x", features = ["react-loop", "mcp"] }
rmcp = { version = "0.x", features = ["client", "transport-stdio"] }
tokio = { version = "1", features = ["full"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
rusqlite = { version = "0.31", features = ["bundled"] }
hmac = "0.12"
sha2 = "0.10"
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["json"] }
clap = { version = "4", features = ["derive"] }
thiserror = "1"
anyhow = "1"

[profile.release]
opt-level = "z"          # 最小体积优化
lto = true               # Link-Time Optimization
codegen-units = 1         # 单 codegen unit, 更好的优化
strip = true              # 去除符号表
panic = "abort"           # abort 而非 unwind, 减小体积
```

2.1.2 构建命令

```
# --- 开发构建（快速编译，含调试信息）---
cargo build

# --- Release Linux（银河麒麟 V10 目标）---
# musl 静态链接，无 glibc 依赖
cargo build --release --target x86_64-unknown-linux-musl
strip target/x86_64-unknown-linux-musl/release/sgclaw

# 验证产物
file target/x86_64-unknown-linux-musl/release/sgclaw
# 预期输出: ELF 64-bit LSB executable, x86-64, statically linked
ls -lh target/x86_64-unknown-linux-musl/release/sgclaw
# 预期大小: ~8.8 MB

# --- Release Windows ---
cargo build --release --target x86_64-pc-windows-msvc

# 验证产物
ls -lh target/x86_64-pc-windows-msvc/release/sgclaw.exe
# 预期大小: ~9.0 MB
```

2.1.3 构建产物位置

```
target/
├── debug/
│   └── sgclaw                          # 开发构建 (~30-50 MB, 含调试符号)
├── x86_64-unknown-linux-musl/
│   └── release/
│       └── sgclaw                      # Linux release (~8.8 MB)
└── x86_64-pc-windows-msvc/
    └── release/
        └── sgclaw.exe                  # Windows release (~9.0 MB)
```

2.1.4 交叉编译配置

在开发机上为 Linux musl 目标编译需要安装 musl 工具链:

```
# Ubuntu / 银河麒麟
sudo apt install musl-tools

# 添加 Rust target
rustup target add x86_64-unknown-linux-musl

# 如果需要 Windows 交叉编译（从 Linux 编译 Windows 产物）
# 需要 cargo-xwin 或在 Windows CI 上构建
cargo install cargo-xwin
cargo xwin build --release --target x86_64-pc-windows-msvc
```

2.2 C++ 构建 (SuperRPA 新增部分)

2.2.1 BUILD.gn 配置

新增 `sgclaw/BUILD.gn`:

```
# src/chrome/browser/superrpa/sgclaw/BUILD.gn

source_set("sgclaw") {
  sources = [
    "sg_claw_process_host.cc",
    "sg_claw_process_host.h",
    "pipe_listener.cc",
    "pipe_listener.h",
    "mac_whitelist_check.cc",
    "mac_whitelist_check.h",
  ]
  deps = [
    "//base",
    "//content/public/browser",
  ]
}

source_set("sgclaw_unit_tests") {
  testonly = true
  sources = [
    "sg_claw_process_host_unittest.cc",
    "pipe_listener_unittest.cc",
    "mac_whitelist_check_unittest.cc",
  ]
  deps = [
    ":sgclaw",
    "//base",
    "//base/test:test_support",
    "//testing/gtest",
  ]
}
```

在父级 `superrpa/BUILD.gn` 中添加依赖:

```
source_set("superrpa") {
  # ... 已有 sources ...
  deps = [
    ":ai",
    ":sgclaw",                    # <-- 新增
    "//base",
    # ... 已有 deps ...
  ]
}
```

2.2.2 构建命令

```
# 进入 Chromium 源码目录
cd /home/zyl/projects/superRpa/src

# 生成构建配置（仅首次或 BUILD.gn 变更时）
gn gen out/Default

# 增量编译 Chrome（24 并发）
autoninja -j 24 -C out/Default chrome

# 单独编译 sgclaw 单元测试
autoninja -j 24 -C out/Default sgclaw_unittests

# 运行 sgclaw 单元测试
./out/Default/sgclaw_unittests
```

2.3 前端构建 (agent-vue)

agent-vue 使用 Vue CLI 构建，产物部署到浏览器资源目录。

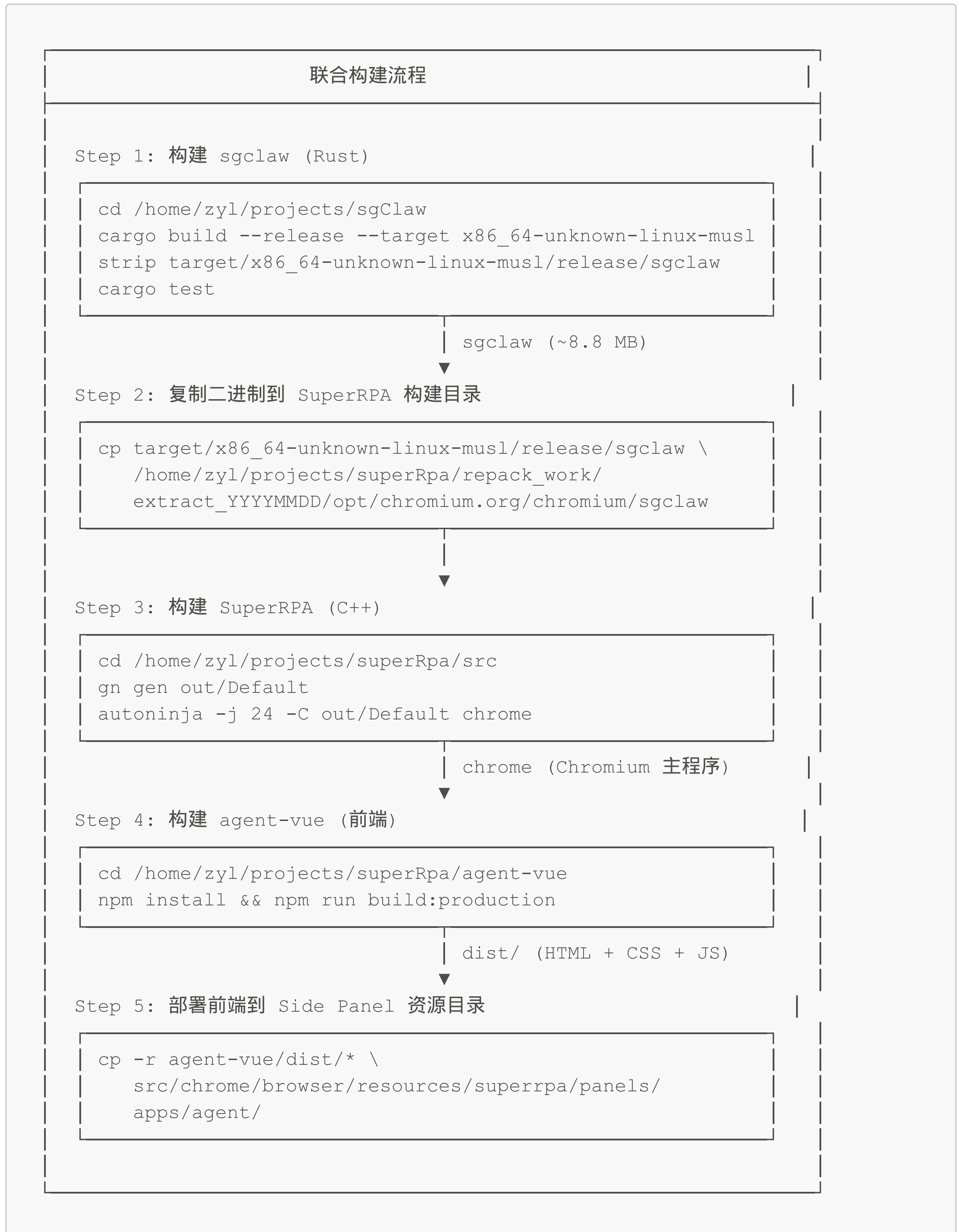
```
# 进入 agent-vue 目录
cd /home/zyl/projects/superRpa/agent-vue

# 安装依赖（首次）
npm install

# 开发模式
npm run serve

# 生产构建
npm run build:production

# 构建产物位于 dist/ 目录
ls dist/
# css/  js/  index.html  favicon.ico  ...
```

前端技术栈:

- Vue 2.6.14 + Vue Router 3.5 + Vuex 3.6
- Element UI 2.15
- Axios 1.9 (HTTP 请求)
- 无新增依赖，SgClawControl.vue 使用已有框架组件

2.4 联合构建流程

完整的从零构建步骤，适用于全新开发环境或 CI 流水线:



自动化构建脚本 (`scripts/build-linux.sh`):

```bash
#!/bin/bash
set -euo pipefail

SGCLAW_DIR="/home/zyl/projects/sgClaw"
SUPERRPA_DIR="/home/zyl/projects/superRpa"
TARGET="x86_64-unknown-linux-musl"

echo "=== [1/4] Building sgclaw (Rust) ==="
cd "$SGCLAW_DIR"
cargo build --release --target "$TARGET"
strip "target/$TARGET/release/sgclaw"
echo "  Binary size: $(du -h target/$TARGET/release/sgclaw | cut -f1)"

echo "=== [2/4] Running sgclaw tests ==="
cargo test --release --target "$TARGET"

echo "=== [3/4] Building SuperRPA (C++) ==="
cd "$SUPERRPA_DIR/src"
autoninja -j 24 -C out/Default chrome

echo "=== [4/4] Building agent-vue ==="
cd "$SUPERRPA_DIR/agent-vue"
npm run build:production

echo "=== Build complete ==="
```

---

3. 依赖管理

3.1 Rust 依赖 (Cargo.toml)

依赖	版本	用途	可选
`zeroclaw`	latest	Agent Runtime: ReAct 循环, Tool/Provider trait	否
`rmcp`	latest	MCP Client: 连接外部 MCP Server 获取工具	features: client, transport-stdio
`tokio`	1.x	异步运行时: 事件循环, I/O, 定时器	features: full
`serde`	1.x	序列化框架: derive 宏	features: derive
`serde_json`	1.x	JSON 编解码: pipe 协议消息	否
`rusqlite`	0.31+	SQLite 存储: 长期记忆, Skill 缓存	features: bundled (内嵌 SQLite)
`hmac`	0.12	HMAC 消息认证码: pipe 消息签名	否
`sha2`	0.10	SHA-256 哈希: 配合 hmac 使用	否
`tracing`	0.1	结构化日志: span, event, 链路追踪	否

依赖	版本	用途	可选
`tracing-subscriber`	0.3	日志输出: JSON 格式, 文件/stderr 输出	features: json
`clap`	4.x	命令行参数解析: 配置文件路径, 日志级别	features: derive
`thiserror`	1.x	错误类型定义: 自定义 Error enum	否
`anyhow`	1.x	错误传播: 顶层错误处理	否

依赖锁定策略:

- `Cargo.lock` 提交到版本控制，确保可重复构建
- 依赖更新通过 `cargo update` 手动触发，经 CI 验证后合并
- `rusqlite` 使用 `bundled` feature，内嵌 SQLite 源码编译，避免系统库版本差异

3.2 C++ 依赖

sgClaw 的 C++ 部分仅使用 Chromium 已有的基础库，无新增第三方依赖:

依赖	GN 路径	用途
base	`//base`	线程 (SequencedTaskRunner), 回调 (OnceCallback), 进程管理 (LaunchProcess), 文件监听 (FileDescriptorWatcher)
content	`//content/public/browser`	BrowserContext, WebContents 交互接口

无需引入 Mojo IPC -- sgClaw 使用更轻量的 STDIO Pipe 通信。 Side Panel UI 通过已有的 `rpa_page_handler` Mojo 接口发送启停命令， 该接口已在 SuperRPA 中实现，仅需新增 2 个 handler method。

3.3 前端依赖

无新增 npm 依赖。SgClawControl.vue 使用 agent-vue 已有的技术栈:

依赖	版本	用途
`vue`	2.6.14	组件框架
`element-ui`	2.15.14	UI 组件库 (Button, Input, Message)
`vuex`	3.6.2	状态管理 (Agent 运行状态)
`axios`	1.9.0	HTTP 请求 (如需与 local_service 通信)

---

4. 打包与发布

4.1 Linux .deb 打包

SuperRPA 以 `.deb` 包形式发布到银河麒麟 V10 SP1。sgClaw 的产物集成到现有打包流程中。

当前 .deb 包结构 (`repack_work/extract_YYYYMMDD/`):

```
DEBIAN/
├── control                   # 包元数据 (版本、大小、描述)
└── postinst                  # 安装后脚本 (chmod +x)
opt/chromium.org/chromium/
├── chrome                    # Chromium 主程序 (~1.6 GB)
├── superrpa-chromium         # 启动脚本 (wrapper)
├── node/                     # Node.js 运行时
├── local_service/            # 本地 HTTP 服务
├── locales/                  # 语言包
├── *.pak                     # 资源文件
├── sgclaw                    # [新增] AI Agent 引擎
└── sgclaw-skills/            # [新增] 技能目录
    ├── builtin/              # 内置技能脚本
    └── registry.json         # 技能清单
usr/bin/
└── superrpa-chromium         # 系统级启动链接
```

4.1.1 修改 repack-with-service.sh

在现有 `scripts/repack-with-service.sh` 中新增 sgclaw 安装步骤:

```
# --- 新增: Install sgclaw binary ---
echo "[N/N] Installing sgclaw AI Agent..."
SGCLAW_BIN="/home/zyl/projects/sgClaw/target/x86_64-unknown-linux-musl/release/sgclaw"
SGCLAW_SKILLS="/home/zyl/projects/sgClaw/skills"

if [ -f "$SGCLAW_BIN" ]; then
    cp "$SGCLAW_BIN" "$CHROMIUM_DIR/sgclaw"
    chmod +x "$CHROMIUM_DIR/sgclaw"
    echo "  sgclaw binary: $CHROMIUM_DIR/sgclaw ($(du -h "$SGCLAW_BIN" | cut -f1))"
fi

if [ -d "$SGCLAW_SKILLS" ]; then
    mkdir -p "$CHROMIUM_DIR/sgclaw-skills"
    cp -r "$SGCLAW_SKILLS/"* "$CHROMIUM_DIR/sgclaw-skills/"
    echo "  sgclaw skills: $CHROMIUM_DIR/sgclaw-skills/"
fi
```

4.1.2 postinst 修改

```
#!/bin/bash
set -e

CHROMIUM_DIR=/opt/chromium.org/chromium

# 已有权限设置
```

```
chmod +x "$CHROMIUM_DIR/node/bin/node" 2>/dev/null || true
chmod +x "$CHROMIUM_DIR/superrpa-chromium" 2>/dev/null || true

# 新增: sgclaw 权限
chmod +x "$CHROMIUM_DIR/sgclaw" 2>/dev/null || true

echo "SuperRPA Chromium with LocalService installed successfully"
```

4.1.3 包大小影响

```
当前 .deb 包大小:              ~447 MB (含 Node.js local_service)
sgclaw 二进制:                  + ~8.8 MB
sgclaw-skills/:                 + ~0.2 MB (JSON + JS 技能脚本)
────────────────────────────────────────
预计新 .deb 包大小:             ~456 MB (增加约 2%)
```

4.2 Windows 安装包

Windows 端的 sgclaw.exe 放置在 SuperRPA 浏览器安装目录下:

```
C:\Program Files\SuperRPA Chromium\
├── chrome.exe
├── sgclaw.exe                  # [新增] ~9 MB
├── sgclaw-skills\              # [新增] 技能目录
│   ├── builtin\
│   └── registry.json
└── ...
```

安装程序修改要点:

- 在 NSIS/Inno Setup 脚本中新增 sgclaw.exe 和 sgclaw-skills/ 的文件拷贝项
- 无需注册服务或添加注册表项 -- sgclaw 由 chrome.exe 按需启动
- 卸载时一并删除 sgclaw 相关文件和 SQLite 数据库

4.3 升级策略

4.3.1 sgclaw 二进制升级

sgclaw 二进制本身无状态（运行时状态在内存中，持久状态在 SQLite 中）， 升级策略为直接覆盖:

```
# 停止 Agent (如果正在运行，由 SgClawProcessHost::Stop() 完成)
# 替换二进制
cp sgclaw_new /opt/chromium.org/chromium/sgclaw
chmod +x /opt/chromium.org/chromium/sgclaw
# 下次用户点击 [启动] 时加载新版本
```

4.3.2 Skills 仓库更新

技能文件支持增量更新:

```
更新流程:
  1. 下载新版 registry.json (含每个技能的 SHA-256 哈希)
  2. 对比本地 registry.json, 识别变更的技能文件
  3. 仅下载变更的 .js 文件
  4. 校验每个文件的 SHA-256 签名
  5. 原子替换 (写入临时文件 -> rename)
```

4.3.3 数据库迁移

SQLite 数据库使用 schema 版本管理:

```sql
-- 内置 user_version pragma 记录 schema 版本
PRAGMA user_version;  -- 返回当前版本号

-- sgclaw 启动时检查并执行迁移
-- migration_001.sql: 初始 schema
-- migration_002.sql: 新增索引
-- migration_003.sql: 新增表
```

迁移规则:

- 向前兼容: 新版 sgclaw 可读取旧版数据库
- 自动迁移: 启动时检测 user_version，依次执行未应用的迁移脚本
- 备份优先: 迁移前自动备份 `.db` 文件为 `.db.bak.{timestamp}`

---

5. 测试策略

5.1 测试金字塔

```
                │  Rust 单元测试      │  大量, 覆盖核心逻辑
                │  cargo test         │  mock LLM + mock pipe
                │                     │
                └─────────────────────┘
```

测试比例指导:

- Rust 单元测试: ~70% (快速、隔离、可重复)
- C++ 单元测试: ~15% (进程管理、协议解析)
- Pipe 协议集成测试: ~10% (真实进程间通信)
- E2E 验收测试: ~5% (端到端场景，mock LLM 保证可重复)

5.2 Rust 单元测试

5.2.1 Agent Runtime 测试

```
# 运行全部 Rust 测试
cargo test

# 运行特定模块测试
cargo test agent::runtime
cargo test agent::critic
```

测试内容:

- ReAct 循环: mock LLM 返回固定输出, 验证 think -> act -> observe 流转
- Circuit Breaker: 模拟连续失败, 验证熔断触发与恢复
- 最大步数限制: 验证超出 50 步自动终止

```
// 测试示例: Circuit Breaker 在连续 10 次失败后触发
#[tokio::test]
async fn test_circuit_breaker_triggers_after_threshold() {
    let mut breaker = CircuitBreaker::new(10);
    for _ in 0..10 {
        breaker.record_failure();
    }
    assert!(breaker.is_open());
}
```

5.2.2 BrowserPipeTool 测试

测试内容:

- 命令序列化: Rust struct -> JSON Line 格式正确
- 响应反序列化: JSON Line -> Rust struct 解析正确
- 未知 action 拒绝: 非枚举值被正确拒绝

- HMAC 签名: 签名生成与校验一致

```
cargo test pipe::protocol
cargo test pipe::browser_tool
```

5.2.3 MAC Policy 测试

测试内容:

- 域名白名单: 允许 / 拒绝 / 通配符匹配
- Action 白名单: 允许的 action 通过, eval/executeJsInPage 被拒绝
- 空白名单: 默认拒绝所有

```
cargo test security::mac_policy
```

5.2.4 Memory 测试

测试内容:

- 短期记忆 Ring Buffer: 容量溢出时正确淘汰最旧条目
- 长期记忆 SQLite: CRUD 操作, schema 迁移, 并发读写安全
- 向量搜索: 相似度计算正确性

```
cargo test memory::short_term
cargo test memory::long_term
```

5.3 C++ 单元测试

新增 `sgclaw_unittests` target, 集成到 SuperRPA 现有测试体系。

```
# 构建
autoninja -j 24 -C out/Default sgclaw_unittests

# 运行
./out/Default/sgclaw_unittests
```

5.3.1 SgClawProcessHost 测试

```
测试用例:
├─ Start() 在二进制存在时成功启动子进程
├─ Start() 在二进制不存在时返回错误
├─ Start() 重复调用不创建多个进程 (Singleton 保证)
```

```
├─ Stop() 发送 shutdown 命令后等待退出
├─ Stop() 超时后强制 kill
├─ OnProcessCrash() 记录崩溃日志, 不自动重启
└─ 析构函数确保清理子进程
```

5.3.2 PipeListener 测试

```
测试用例:
├─ 正确解析单行 JSON 消息
├─ 正确解析多行连续消息 (JSON Line 流)
├─ 拒绝超过 1 MB 的消息 (防内存 DoS)
├─ 拒绝非 JSON 格式的行
├─ 管道断开时触发 OnPipeDisconnected 回调
└─ sequence_id 乱序时触发告警
```

5.3.3 MAC Whitelist Check 测试

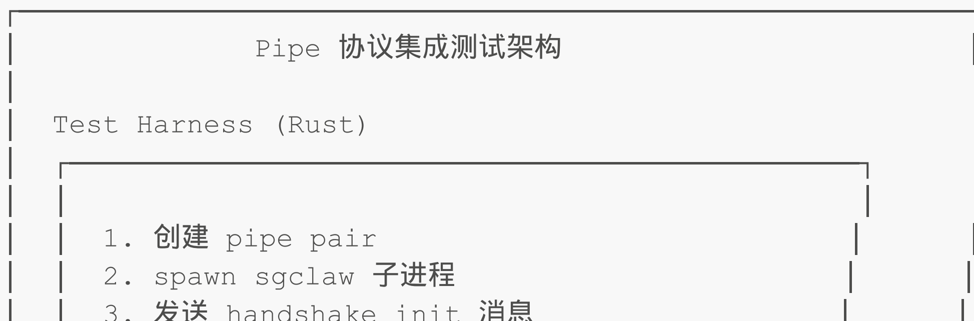
```
测试用例:
├─ 允许列表内的 action (click, type, navigate) 通过
├─ 禁止列表的 action (eval, executeJsInPage) 被拒绝
├─ expected_domain 与当前页面域名匹配时通过
├─ expected_domain 与当前页面域名不匹配时拒绝
├─ 速率限制: 同一域超过 10 次/秒后拒绝
└─ rules.json 加载失败时默认拒绝所有
```

5.4 Pipe 协议集成测试

集成测试启动真实的 sgclaw 进程，通过 pipe 发送命令并验证响应。 位于 `tests/integration/` 目录。

```
# 运行 Rust 集成测试 (需要先构建 sgclaw)
cargo test --test integration
```

5.4.1 测试场景



```
┌──────────────────────────────────────────────────────────┐
│                Pipe 协议集成测试架构                       │
│                                                          │
│  Test Harness (Rust)                                     │
│  ┌──────────────────────────────────────────────────┐    │
│  │                                                  │    │
│  │  1. 创建 pipe pair                               │    │
│  │  2. spawn sgclaw 子进程                          │    │
│  │  3. 发送 handshake init 消息                     │    │
```

```
│  │  4. 验证 init_ack 响应                        │       │
│  │  5. 发送 command 消息                         │       │
│  │  6. 验证 response 格式和内容                  │       │
│  │  7. 发送 shutdown                             │       │
│  │  8. 验证进程正常退出                          │       │
│  │                                               │       │
│  │  sgclaw 使用 mock LLM (--llm-mock 标志)       │       │
│  │  mock LLM 返回固定 action 序列                │       │
│  │                                               │       │
│  └───────────────────────────────────────────────┘       │
└──────────────────────────────────────────────────────────┘
```

测试用例:

- Handshake 成功: init -> init_ack, 版本匹配
- Handshake 版本不匹配: sgclaw 返回错误并退出
- 命令执行: command -> response, 验证 seq 匹配
- 错误处理: 发送格式错误的 JSON, 验证 error response
- 管道断开: 关闭写端, 验证 sgclaw 退出码 0
- 大消息拒绝: 发送 > 1 MB 的消息, 验证被丢弃

5.5 E2E 验收测试

E2E 测试覆盖完整的用户场景: 从浏览器启动 Agent 到任务执行完成。 使用 mock LLM 确保测试可重复、无外部依赖。

5.5.1 测试环境

```
┌──────────────────────────────────────────────────────────┐
│                  E2E 测试环境                            │
│                                                          │
│  SuperRPA Chrome (headless 模式)                         │
│  ├─ SgClawProcessHost 启动 sgclaw                        │
│  ├─ sgclaw 使用 mock LLM (固定输出)                      │
│  └─ 目标页面: 本地 test HTTP server (固定 HTML)          │
│                                                          │
│  验证链:                                                 │
│  用户输入 -> Agent 理解 -> 生成 action -> Pipe 传输      │
│  -> MAC 校验 -> CommandRouter 执行 -> 页面 DOM 变更      │
│  -> 响应返回 -> Agent observe -> 任务完成                │
└──────────────────────────────────────────────────────────┘
```

5.5.2 验收场景

场景	用户输入	预期行为	验收标准
基本点击	"点击提交按钮"	Agent 生成 click action	按钮状态变为 disabled
表单填写	"在用户名框输入 admin"	Agent 生成 type action	input.value === "admin"

场景	用户输入	预期行为	验收标准
页面导航	"打开审批列表页面"	Agent 生成 navigate action	URL 变更为目标地址
域名拦截	"访问 evil.com"	MAC 拒绝, Agent 报告失败	未发生导航
熔断触发	连续 10 次无效 action	Circuit Breaker 打开	Agent 自动停止

5.6 测试运行汇总

```
# === Rust 全量测试 ===
cd /home/zyl/projects/sgClaw
cargo test                                # 单元测试 + 集成测试
cargo test --release                      # Release 模式测试（更贴近生产）

# === C++ 全量测试 ===
cd /home/zyl/projects/superRpa/src
autoninja -j 24 -C out/Default sgclaw_unittests
./out/Default/sgclaw_unittests

# === 已有 SuperRPA 测试（回归验证）===
autoninja -j 24 -C out/Default superrpa_unittests
./out/Default/superrpa_unittests          # 14 test files

autoninja -j 24 -C out/Default superrpa_ai_unittests
./out/Default/superrpa_ai_unittests       # 3 test files

# === 代码质量检查 ===
cargo clippy -- -D warnings              # Rust lint
cargo fmt -- --check                      # Rust 格式化检查
cd /home/zyl/projects/superRpa/src && git cl format  # C++ 格式化
```

---

6. 开发环境搭建

6.1 Rust 开发环境

6.1.1 基础安装

```
# 安装 Rust 工具链 (stable channel)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env

# 验证安装
rustc --version        # 预期: rustc 1.7x.x (stable)
cargo --version        # 预期: cargo 1.7x.x

# 安装编译目标
rustup target add x86_64-unknown-linux-musl     # Linux 静态链接
rustup target add x86_64-pc-windows-msvc        # Windows 交叉编译（可选）
```

```
# 安装 musl 工具链 (Linux 静态链接依赖)
sudo apt update && sudo apt install -y musl-tools

# 安装开发工具
rustup component add clippy        # Lint 工具
rustup component add rustfmt       # 格式化工具
```

6.1.2 项目克隆与首次构建

```
# 克隆项目
git clone <repo-url> /home/zyl/projects/sgClaw
cd /home/zyl/projects/sgClaw

# 开发构建 (首次编译需下载依赖，约 2-5 分钟)
cargo build

# 运行测试
cargo test

# 运行代码检查
cargo clippy -- -D warnings

# 格式化代码
cargo fmt
```

6.1.3 开发模式运行

sgclaw 通过 STDIO 与浏览器通信，开发时可直接在终端模拟 pipe 输入:

```
# 方式一: 直接运行，手动输入 JSON Line
cargo run

# 在终端中输入 (模拟 Browser 发来的 init 消息):
{"type":"init","version":"1.0","hmac_key":"dev-test-key"}

# 方式二: 使用文件作为输入
cargo run < tests/fixtures/sample_session.jsonl

# 方式三: 使用 echo + pipe
echo '{"type":"init","version":"1.0"}' | cargo run
```

6.2 C++ 开发环境

C++ 开发依赖已有的 SuperRPA 开发环境。以下假设 depot_tools 和 Chromium 源码已就绪。

```
# 确认 depot_tools 在 PATH 中
which gn          # /path/to/depot_tools/gn
which autoninja   # /path/to/depot_tools/autoninja

# Chromium 源码目录
ls /home/zyl/projects/superRpa/src/chrome/browser/superrpa/
# 预期看到: BUILD.gn, ai/, cdp/, router/, session/, zombie/, ...

# 新增 sgclaw 文件后，重新生成构建配置
cd /home/zyl/projects/superRpa/src
gn gen out/Default

# 增量编译 (仅编译变更的文件)
autoninja -j 24 -C out/Default chrome

# 编译并运行 sgclaw 单元测试
autoninja -j 24 -C out/Default sgclaw_unittests
./out/Default/sgclaw_unittests
```

6.3 联合调试

6.3.1 sgclaw 独立调试 (Rust 侧)

```
# 使用 RUST_LOG 控制日志级别
RUST_LOG=debug cargo run 2>sgclaw-debug.log

# 使用 lldb 或 gdb 调试
cargo build
lldb target/debug/sgclaw

# VS Code 调试 (需要 CodeLLDB 扩展)
# launch.json 配置见 6.4 节
```

6.3.2 附加到浏览器进程调试 (C++ 侧)

```
# 启动 SuperRPA 浏览器
cd /home/zyl/projects/superRpa/src
./out/Default/chrome --no-sandbox

# 在另一终端查找 sgclaw 子进程 PID
pgrep -a sgclaw

# 使用 gdb 附加
gdb -p <sgclaw_pid>

# 或使用 gdb 附加到 chrome 主进程，在 sgclaw 相关代码设断点
gdb -p <chrome_pid>
```

```
(gdb) break SgClawProcessHost::Start
(gdb) continue
```

6.3.3 日志查看

```
# sgclaw Rust 日志 (输出到 stderr, 不干扰 pipe 通信)
# 日志格式: JSON (tracing-subscriber json feature)
RUST_LOG=sgclaw=debug ./sgclaw 2>agent.log

# 实时查看日志
tail -f agent.log | python3 -m json.tool

# Chrome 日志 (含 SgClawProcessHost 日志)
./out/Default/chrome --no-sandbox --enable-logging --v=1 2>&1 | grep -i
sgclaw
```

6.4 IDE 配置

6.4.1 VS Code + rust-analyzer (推荐)

`.vscode/settings.json`:

```
{
  "rust-analyzer.cargo.target": "x86_64-unknown-linux-musl",
  "rust-analyzer.check.command": "clippy",
  "rust-analyzer.check.extraArgs": ["--", "-D", "warnings"],
  "rust-analyzer.inlayHints.typeHints.enable": true,
  "rust-analyzer.inlayHints.parameterHints.enable": true,
  "[rust]": {
    "editor.formatOnSave": true,
    "editor.defaultFormatter": "rust-lang.rust-analyzer"
  }
}
```

`.vscode/launch.json` (调试配置):

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "lldb",
      "request": "launch",
      "name": "Debug sgclaw",
      "cargo": {
        "args": ["build", "--bin=sgclaw"]
      },
```

```
        "args": [],
        "env": {
          "RUST_LOG": "debug"
        },
        "stdio": [null, null, null]
      },
      {
        "type": "lldb",
        "request": "launch",
        "name": "Debug unit tests",
        "cargo": {
          "args": ["test", "--no-run"]
        }
      }
    ]
  }
```

推荐扩展:

- `rust-lang.rust-analyzer` -- Rust 语言支持
- `vadimcn.vscode-lldb` -- LLDB 调试器
- `tamasfe.even-better-toml` -- TOML 语法高亮
- `serayuzgur.crates` -- Cargo.toml 依赖版本提示

6.4.2 CLion + Rust 插件

CLion 原生支持 Rust (通过内置插件)。打开 `Cargo.toml` 即可识别项目。

关键配置:

- File -> Settings -> Languages -> Rust -> Cargo target: `x86_64-unknown-linux-musl`
- Run Configuration -> Cargo Command: 选择 `run` 或 `test`
- External Tool: 可配置 `cargo clippy` 为保存时自动运行

---

7. 代码规范

7.1 Rust 规范

7.1.1 格式化

使用 `rustfmt` 统一代码格式。配置文件 `rustfmt.toml`:

```
edition = "2021"
max_width = 100
tab_spaces = 4
use_field_init_shorthand = true
use_try_shorthand = true
```

执行命令:

```
# 格式化所有代码
cargo fmt

# 检查格式 (CI 中使用)
cargo fmt -- --check
```

7.1.2 Lint

使用 `clippy` 进行静态分析，CI 中以 `-D warnings` 模式运行 (warnings 即 error):

```
cargo clippy -- -D warnings
```

7.1.3 命名约定

元素	风格	示例
变量 / 函数	snake_case	`process_message`, `pipe_reader`
类型 / Trait	CamelCase	`BrowserPipeTool`, `MacPolicy`
常量	SCREAMING_SNAKE_CASE	`MAX_MESSAGE_SIZE`, `DEFAULT_TIMEOUT`
模块	snake_case	`browser_tool`, `mac_policy`
文件名	snake_case	`browser_tool.rs`, `mac_policy.rs`

7.1.4 错误处理

- 库级别错误: 使用 `thiserror` 定义具体的 Error enum
- 应用级别传播: 使用 `anyhow::Result` 简化错误链
- 禁止 `unwrap()` / `expect()` 出现在非测试代码中 (clippy 规则)

```
// 正确: 定义具体错误类型
#[derive(thiserror::Error, Debug)]
pub enum PipeError {
    #[error("message too large: {size} bytes (max {max})")]
    MessageTooLarge { size: usize, max: usize },

    #[error("invalid JSON: {0}")]
    InvalidJson(#[from] serde_json::Error),

    #[error("pipe disconnected")]
    Disconnected,
}

// 正确: 使用 ? 传播
```

```
fn read_message(reader: &mut BufReader<Stdin>) ->
anyhow::Result<PipeMessage> {
    let line = reader.read_line()?;
    let msg: PipeMessage = serde_json::from_str(&line)?;
    Ok(msg)
}
```

7.1.5 日志

使用 `tracing` 框架，提供结构化日志和链路追踪:

```
use tracing::{info, warn, error, debug, instrument};

#[instrument(skip(self), fields(seq = msg.seq))]
async fn handle_message(&self, msg: PipeMessage) -> Result<Response> {
    info!(action = %msg.action, "processing command");
    // ...
    if let Err(e) = result {
        error!(error = %e, "command failed");
    }
}
```

7.2 C++ 规范 (延续 SuperRPA)

延续 Chromium / SuperRPA 现有编码风格:

规则	说明
缩进	2 空格
命名	CamelCase (类型), lower_case (变量), kCamelCase (常量)
头文件	`#ifndef` 守卫，使用 `#pragma once` 亦可
智能指针	`std::unique_ptr` 优先，`scoped_refptr` 用于引用计数对象
线程安全	`base::SequencedTaskRunner` 绑定，避免直接使用锁
格式化	`git cl format` (调用 clang-format)

```
# 格式化 C++ 代码
cd /home/zyl/projects/superRpa/src
git cl format
```

7.3 Git 规范

7.3.1 分支策略

```
main (保护分支, 始终可部署)
  │
  ├── feature/sgclaw-pipe-protocol      # 功能分支
  ├── feature/sgclaw-mac-check          # 功能分支
  ├── fix/sgclaw-crash-on-large-msg     # 修复分支
  └── release/v1.0.0                    # 发布分支
```

规则:

- `main` 分支受保护，仅通过 PR 合并
- 功能分支从 `main` 创建，完成后合并回 `main`
- 命名格式: `{type}/{scope}-{brief-description}`
- type: feature, fix, refactor, test, docs

7.3.2 Commit Message 格式

采用 Conventional Commits 规范:

```
<type>(<scope>): <subject>

<body>

<footer>
```

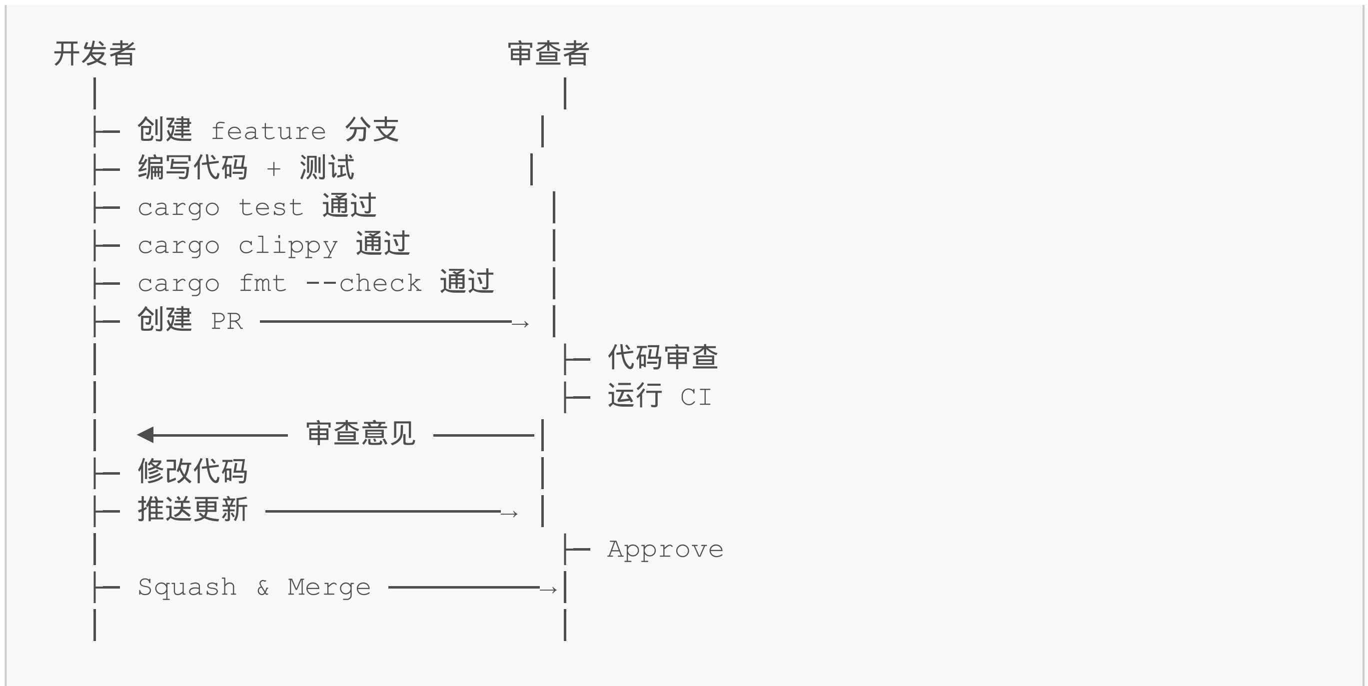
类型:

- `feat`: 新功能
- `fix`: Bug 修复
- `refactor`: 重构 (不改变行为)
- `test`: 测试
- `docs`: 文档
- `chore`: 构建/工具

示例:

```
feat(pipe): implement JSON Line protocol encoder/decoder

Add PipeMessage struct with serde serialization.
Support command, response, event message types.
Maximum message size: 1 MB.

Refs: #42
```

7.3.3 Code Review 流程



7.4 文档规范

7.4.1 代码文档

```rust
//! 模块级文档：描述模块的职责、设计决策和使用方式。
//!
//! # 示例
//!
//! ```rust
//! let tool = BrowserPipeTool::new(writer);
//! tool.execute(Action::Click { selector: "#btn".into() }).await?;
//! ```

/// 公开 API 文档：描述函数的作用、参数、返回值和错误情况。
///
/// # Arguments
///
/// * `msg` - 待发送的 pipe 消息
///
/// # Errors
///
/// 返回 `PipeError::Disconnected` 当管道已断开。
pub async fn send(&mut self, msg: &PipeMessage) -> Result<(), PipeError> {
    // ...
}
```

规则:

- 所有 `pub` item 必须有 `///` 文档注释
- 每个模块的 `mod.rs` 必须有 `//!` 模块级文档
- 复杂逻辑在代码中使用 `//` 行内注释说明 "为什么"，而非 "做什么"

7.4.2 CHANGELOG 维护

```
# Changelog

## [Unreleased]

### Added
- Pipe protocol JSON Line encoder/decoder
- BrowserPipeTool with click, type, navigate actions

### Fixed
- Circuit Breaker not resetting after cooldown period

## [0.1.0] - 2026-03-03

### Added
- Initial project structure
- Agent Runtime based on ZeroClaw
```

---

8. 部署拓扑

8.1 目标环境

环境	用途	操作系统	硬件配置
开发环境	日常开发调试	Ubuntu 22.04 / 银河麒麟 V10	16 GB RAM, SSD
测试环境	CI + 集成测试	银河麒麟 V10 SP1	8 GB RAM
生产环境	最终部署	银河麒麟 V10 SP1 (x86_64)	8 GB RAM
Windows 环境	开发 + 部分客户	Windows 10 / 11	8 GB RAM

8.2 生产部署目录结构

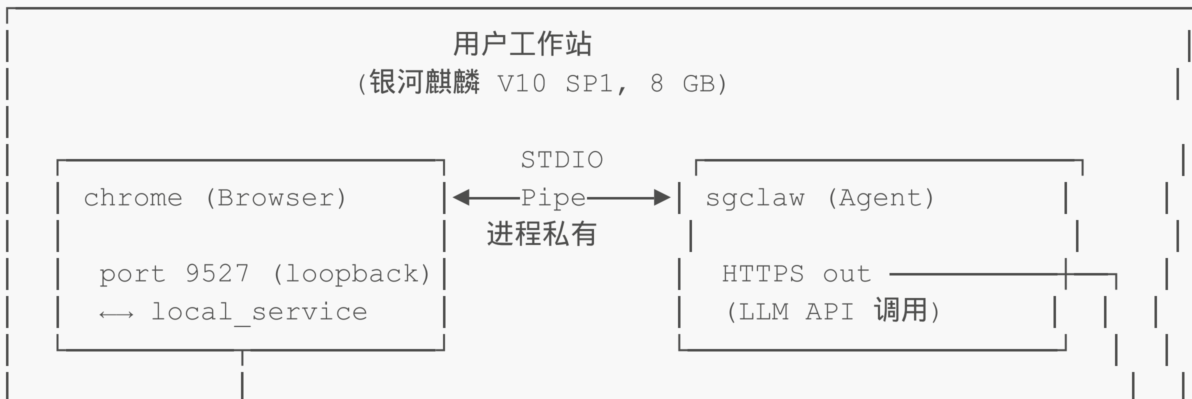
```
/opt/chromium.org/chromium/                 # SuperRPA 安装根目录
├── chrome                                  # Chromium 主程序 (~1.6 GB)
├── superrpa-chromium                       # 启动脚本 (包含 watchdog)
├── sgclaw                                  # [新增] AI Agent 引擎 (~8.8 MB)
├── sgclaw-skills/                          # [新增] 技能目录
│   ├── builtin/                            # 内置业务技能
│   │   ├── oa-approval.js                 # OA 审批流程
│   │   └── finance-report.js              # 财务报表导出
│   └── registry.json                       # 技能清单 + SHA-256 签名
├── node/                                   # Node.js 运行时
│   └── bin/node
├── local_service/                          # 本地 HTTP 服务
│   ├── bin/www
│   └── routes/
```

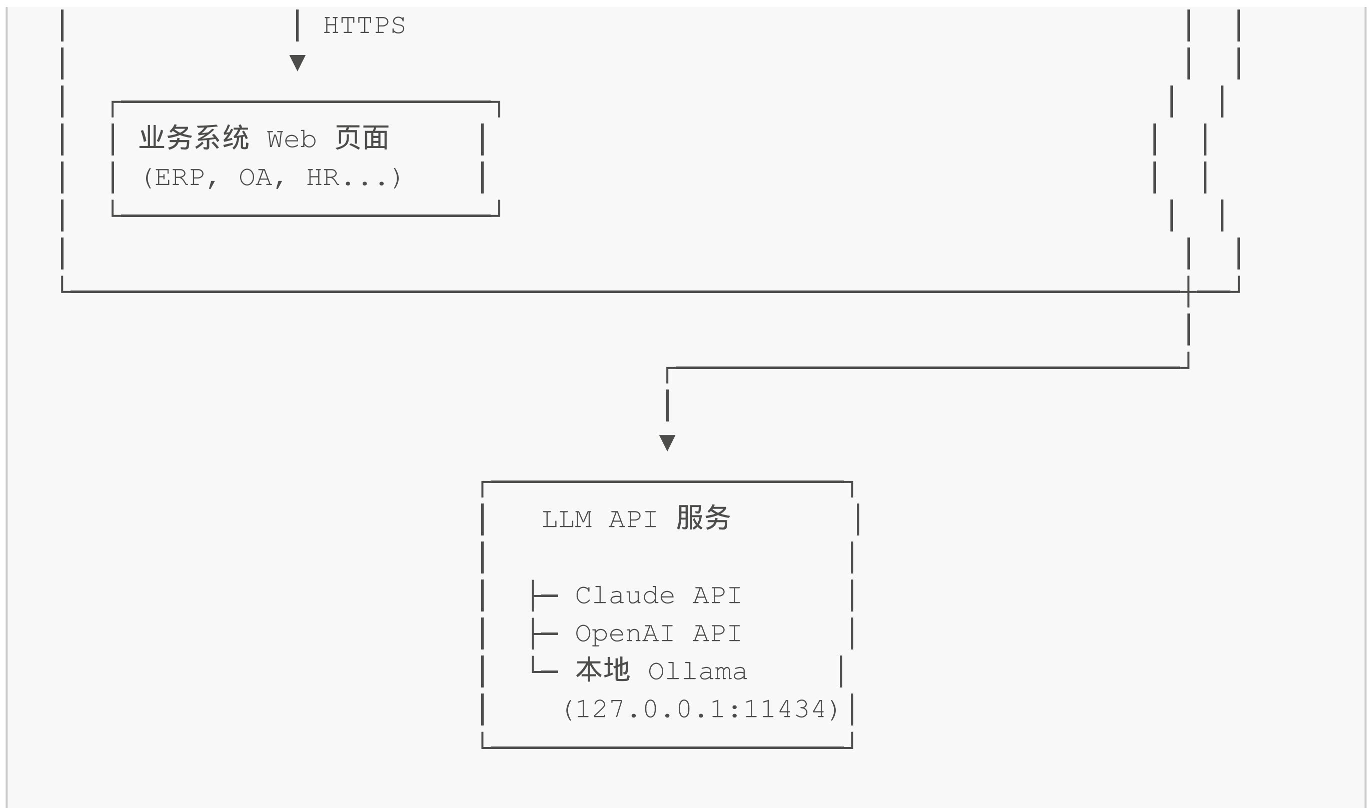
```
├── locales/                                    # 国际化语言包
├── resources.pak                               # Chromium 资源
├── *.pak                                       # 其他资源文件
└── *.so                                        # 共享库
```

8.3 运行时进程拓扑

```
superrpa-chromium (wrapper script, PID 1)
  │
  ├── watchdog (background, PID 2)
  │   └── node local_service (PID 3)
  │       └── 监听 127.0.0.1:9527
  │
  └── chrome --no-sandbox (foreground, PID 4)
       │
       ├── GPU Process (PID 5)
       ├── Network Service (PID 6)
       ├── Renderer: 前台标签页 (PID 7, 8, ...)
       ├── Renderer: Side Panel (PID 9)
       ├── Renderer: Zombie Pages (PID 10, 11, ...)
       │
       └── sgclaw (PID 12, 按需启动)              # [新增]
            │
            ├── stdin  <── chrome pipe 写端
            ├── stdout ──> chrome pipe 读端
            ├── stderr ──> 日志文件 / 丢弃
            │
            ├── tokio runtime (async)
            │   ├── pipe reader task
            │   ├── pipe writer task
            │   ├── agent runtime task
            │   └── LLM HTTP client task
            │
            └── SQLite (文件 I/O)
                └── ~/.sgclaw/memory.db
```

8.4 网络拓扑





网络访问说明:

- sgclaw 仅需出站 HTTPS 连接 (LLM API)
- sgclaw 不监听任何端口 (零网络攻击面)
- local_service 仅监听 127.0.0.1:9527 (loopback, 不对外暴露)
- chrome 与 sgclaw 之间通过 STDIO Pipe 通信 (不经过网络)
- 本地 Ollama 部署时，LLM 调用也走 loopback，无外网依赖

8.5 数据目录

sgclaw 的持久化数据存放在用户主目录下:

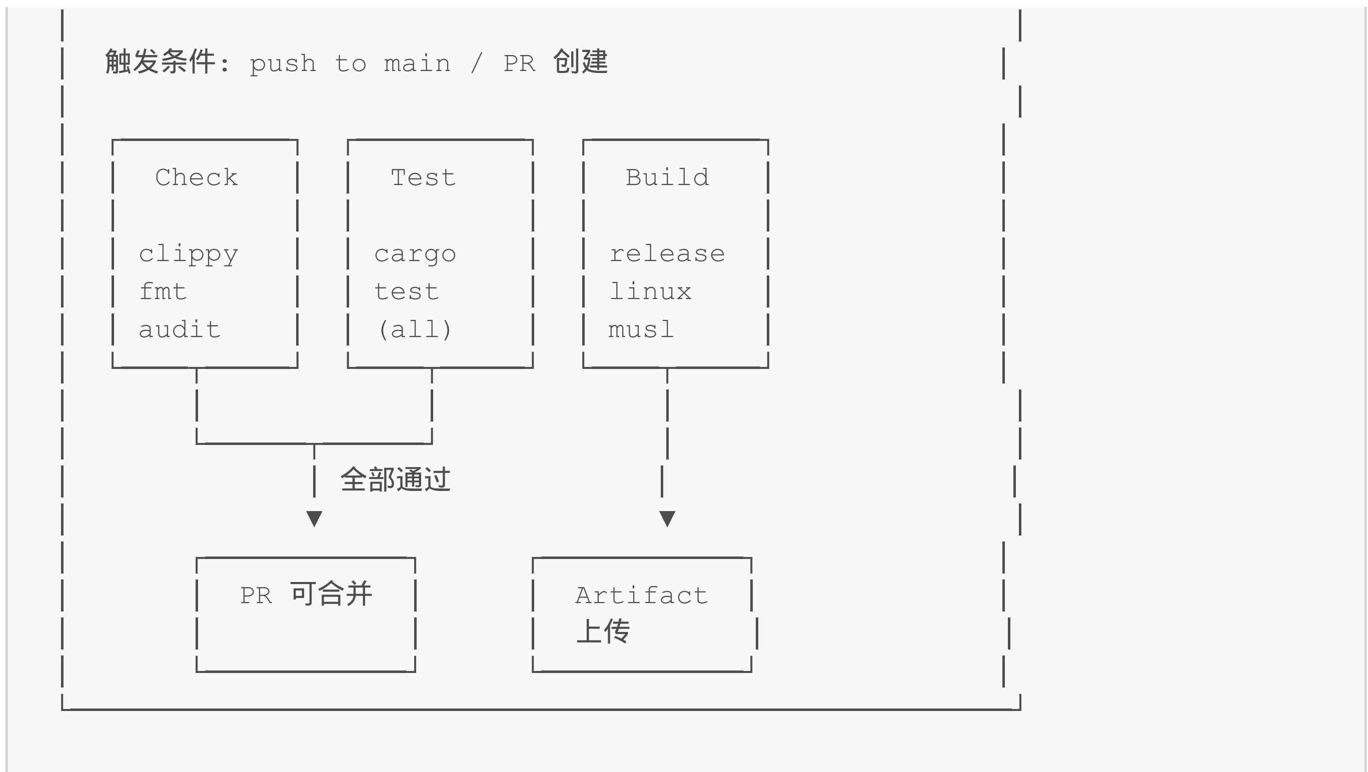
```
~/.sgclaw/                                  # sgclaw 数据根目录
├── memory.db                               # SQLite 长期记忆数据库
├── memory.db.bak.{timestamp}               # 迁移前自动备份
├── config.toml                             # 用户级配置 （可选）
└── logs/                                   # 日志目录 （可选）
    └── sgclaw-YYYY-MM-DD.log
```

数据目录在首次运行时自动创建，权限设置为 `0700` (仅用户可访问)。

---

9. CI/CD 流水线

9.1 Rust CI (sgClaw 仓库)



GitHub Actions 配置要点:

```yaml
# .github/workflows/ci.yml
name: CI

on:
  push:
    branches: [main]
  pull_request:

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
        with:
          targets: x86_64-unknown-linux-musl
          components: clippy, rustfmt
      - run: cargo fmt -- --check
      - run: cargo clippy -- -D warnings

  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
      - run: cargo test

  build:
    runs-on: ubuntu-latest
    needs: [check, test]
```

```
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
        with:
          targets: x86_64-unknown-linux-musl
      - run: sudo apt install -y musl-tools
      - run: cargo build --release --target x86_64-unknown-linux-musl
      - run: strip target/x86_64-unknown-linux-musl/release/sgclaw
      - uses: actions/upload-artifact@v4
        with:
          name: sgclaw-linux
          path: target/x86_64-unknown-linux-musl/release/sgclaw
```

9.2 C++ CI (SuperRPA 仓库)

C++ CI 依赖 Chromium 构建环境，通常在专用构建服务器上执行:

```
# CI 脚本 (在构建服务器上运行)
cd /home/zyl/projects/superRpa/src

# 增量编译
autoninja -j 24 -C out/Default chrome

# 运行所有 SuperRPA 测试 (含新增的 sgclaw 测试)
./out/Default/superrpa_unittests
./out/Default/superrpa_ai_unittests
./out/Default/sgclaw_unittests

# 格式化检查
git cl format --diff
```

9.3 发布流程

```
版本发布流程:

1. 版本号更新
   ├─ sgClaw: Cargo.toml version
   ├─ CHANGELOG.md: 新版本记录
   └─ 创建 Git tag: v0.1.0

2. 构建产物
   ├─ Linux: cargo build --release --target x86_64-unknown-linux-musl
   ├─ Windows: cargo build --release --target x86_64-pc-windows-msvc
   └─ 验证: file, size, strip

3. 集成到 SuperRPA
   ├─ 复制 sgclaw 到 repack_work/extract_YYYYMMDD/opt/chromium.org/chromium/
   ├─ 复制 sgclaw-skills/ 到同目录
   ├─ 运行 repack-with-service.sh
```

```
    └─ 生成 .deb: superrpa-chromium-1.0.0-kylin-v10-YYYYMMDD-with-service.deb

4. 验证
    ├─ 在银河麒麟 V10 虚拟机安装 .deb
    ├─ 启动浏览器, 打开 Side Panel
    ├─ 点击 [启动 Agent], 验证进程启动
    ├─ 执行简单指令, 验证 pipe 通信
    └─ 验证 .deb 包大小在预期范围内

5. 发布
    ├─ 上传 .deb 到内部分发平台
    └─ 通知测试团队
```

---

10. 常见问题与故障排查

10.1 构建问题

问题	原因	解决方案
`linker 'musl-gcc' not found`	未安装 musl 工具链	`sudo apt install musl-tools`
`cargo build` 依赖下载超时	网络问题或 crates.io 镜像	配置 `~/.cargo/config.toml` 使用镜像源
rusqlite 编译失败	SQLite C 代码编译问题	确保使用 `features = ["bundled"]`
`gn gen` 报错 `sgclaw not found`	BUILD.gn 路径错误	检查 `sgclaw/BUILD.gn` 文件是否存在
strip 后二进制过大 (>15 MB)	LTO 或 opt-level 配置问题	检查 `[profile.release]` 配置

10.2 运行时问题

问题	排查方式	解决方案
sgclaw 启动后立即退出	查看 stderr 输出或日志	检查 handshake 超时、配置文件路径
pipe 消息无响应	`RUST_LOG=debug` 查看日志	检查 JSON 格式、sequence_id 是否连续
MAC 拒绝合法操作	查看 C++ 侧日志	检查 rules.json 域名白名单配置
LLM 调用失败	检查网络连接和 API key	确认 LLM API endpoint 可达
Circuit Breaker 触发	查看 Agent 日志	分析连续失败原因，可能是目标页面变更
SQLite 数据库锁定	检查是否有多个 sgclaw 实例	确保 Singleton 机制生效

10.3 Cargo 镜像配置 (国内网络)

```toml
# ~/.cargo/config.toml
[source.crates-io]
replace-with = "ustc"

[source.ustc]
registry = "sparse+https://mirrors.ustc.edu.cn/crates.io-index/"
```

文档结束。本文档为 sgClaw L4 层工程实现参考，覆盖仓库结构、构建系统、依赖管理、 打包发布、测试策略、开发环境、代码规范、部署拓扑与 CI/CD 流程。 如有疑问请联系 sgClaw 开发团队。