L2 — 核心模块与接口契约层

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

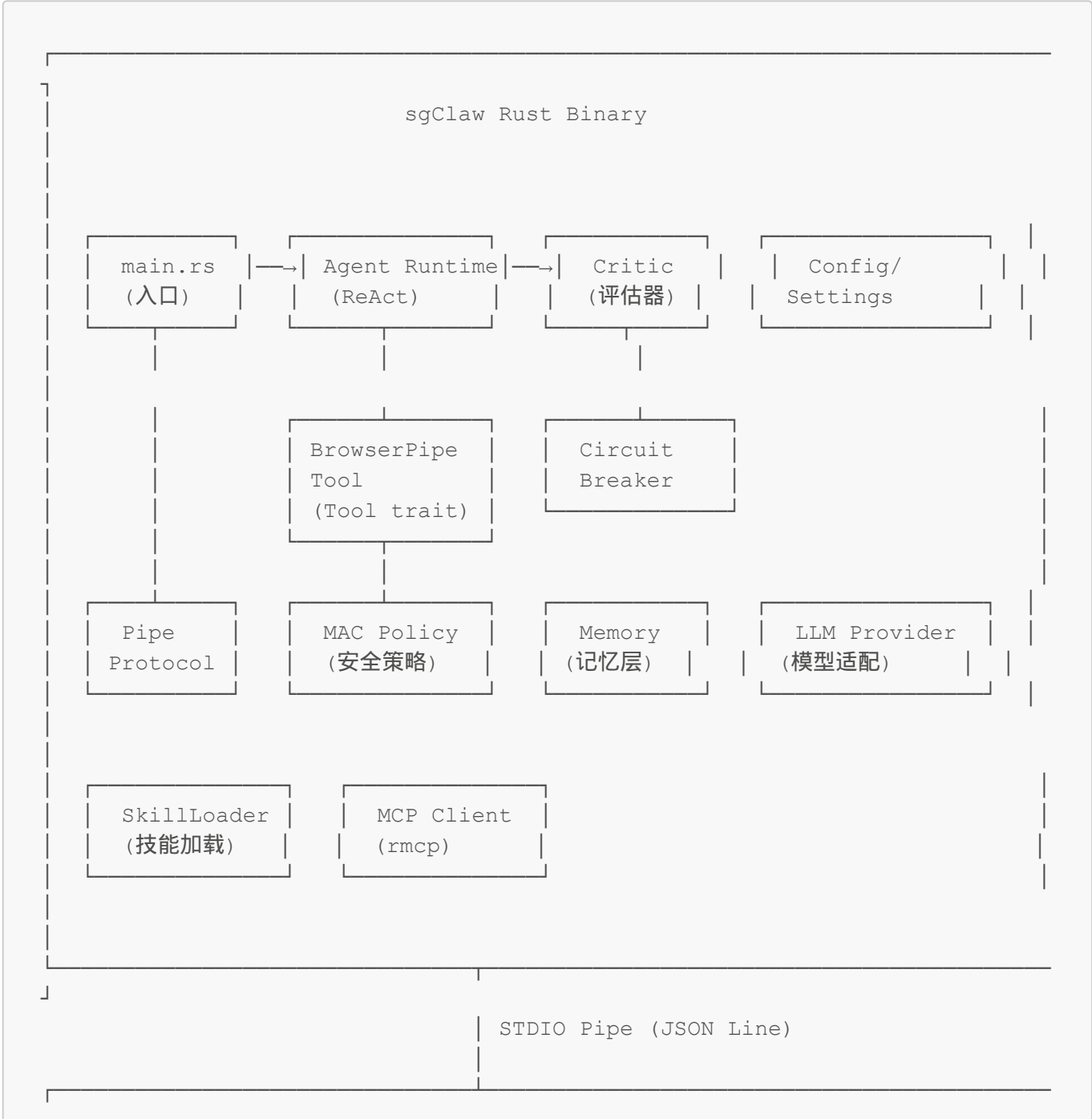
读者: 各组组长（Rust 组、浏览器 C++ 组、前端组）—— 需要理解模块边界、接口契约、依赖关系，指导各组并行开发。

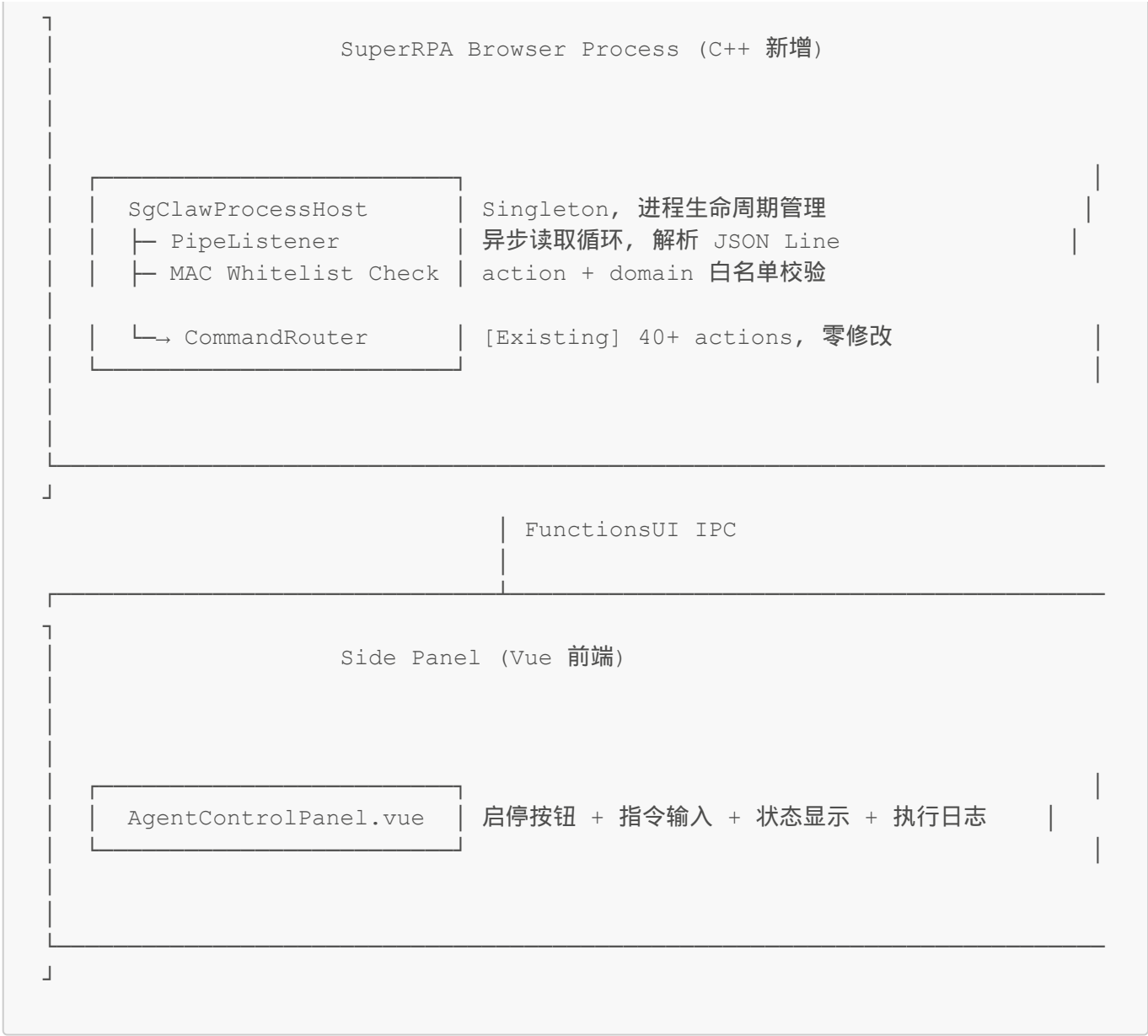
---

1. 模块总览与依赖拓扑

1.1 全局模块图

sgClaw 系统由三个独立的代码域组成：Rust 端（sgclaw binary）、C++ 端（Browser Process 新增模块）、前端（Side Panel Vue 组件）。以下拓扑展示了所有模块及其依赖关系。





1.2 模块依赖矩阵

以下矩阵展示 Rust 端各模块之间的依赖关系（行依赖列）：

	Runtime	BrowserPipe	Protocol	MAC	Memory	LLM	Skill	Critic	MCP	Config
Runtime	-	✓	-	-	✓	✓	✓	✓	✓	✓
BrowserPipe	-	-	✓	✓	-	-	-	-	-	-
Protocol	-	-	-	-	-	-	-	-	-	✓
MAC Policy	-	-	-	-	-	-	-	-	-	✓
Memory	-	-	-	-	-	-	-	-	-	✓
LLM Provider	-	-	-	-	-	-	-	-	-	✓
SkillLoader	-	-	-	-	-	-	-			

```
-       -       ✓
Critic            -         -           -       -       ✓       -       -
-       -       ✓
MCP Client        -         -           -       -       -       -       -
-       -       ✓
Config            -         -           -       -       -       -       -
-       -       -
```

设计原则：

- **Config 是叶节点**：所有模块依赖 Config，Config 不依赖任何业务模块
- **Runtime 是根节点**：Agent Runtime 协调所有模块，是唯一的"上帝模块"
- **BrowserPipeTool 仅依赖 Protocol 和 MAC**：确保 Tool 实现的纯粹性
- **无循环依赖**：依赖图是 DAG（有向无环图）

1.3 并行开发分工

基于模块依赖关系，三个组可以并行开发：

组别	负责模块	依赖条件	可并行阶段
Rust 组	Agent Runtime, BrowserPipeTool, Protocol, MAC Policy, Memory, LLM Provider, SkillLoader, Critic, MCP Client, Config	无外部依赖，可先行	第一阶段即可开始
C++ 组	SgClawProcessHost, PipeListener, MAC Whitelist Check	需与 Rust 组对齐 Pipe 协议格式	协议 JSON Schema 确定后开始
前端组	AgentControlPanel.vue	需 C++ 组提供 FunctionsUI handler	C++ 组 handler 就绪后开始

2. sgClaw Rust 端模块详细设计

2.1 main.rs — 进程入口

职责：初始化 Pipe I/O、配置加载、tokio 异步运行时启动、模块组装、handshake 握手。

启动序列：

```
main()
  │
  ├── 1. 初始化 tracing（日志）
  │      └── 日志输出到 stderr（stdout 保留给 pipe）
  │
  ├── 2. 加载 Config
  │      ├── 默认值（编译时内嵌）
  │      ├── 配置文件（sgclaw.toml，可选）
  │      └── 环境变量覆盖（SGCLAW_*）
  │
```

```
    │
    ├── 3. 初始化 tokio runtime
    │      └── multi_thread, worker_threads = 2
    │
    ├── 4. 创建 Pipe I/O
    │      ├── stdin  → BufReader (接收 Browser 消息)
    │      └── stdout → BufWriter (发送消息到 Browser)
    │
    ├── 5. Handshake
    │      ├── 等待 Browser 的 init 消息 (超时 5s)
    │      ├── 校验协议版本
    │      ├── 交换 HMAC 密钥种子
    │      └── 发送 init_ack 确认
    │
    ├── 6. 组装模块
    │      ├── MAC Policy  ← Config (rules 路径)
    │      ├── Protocol    ← Config (max_message_size)
    │      ├── BrowserPipeTool ← Protocol + MAC
    │      ├── LLM Provider ← Config (model, api_key)
    │      ├── Memory       ← Config (db_path)
    │      ├── SkillLoader  ← Config (skills_dir)
    │      ├── MCP Client   ← Config (mcp_servers)
    │      ├── Critic       ← Config (thresholds) + Memory
    │      └── Agent Runtime ← 以上全部
    │
    ├── 7. 启动 Agent 消息循环
    │      └── 监听 pipe 输入，分发到 Runtime
    │
    └── 8. 优雅退出
           ├── 收到 shutdown 命令 / EOF / SIGTERM
           ├── 停止 Agent loop
           ├── flush Memory 到磁盘
           └── 退出进程 (exit code 0)
```

关键设计决策：

- **stdout 专用于 pipe**：所有日志、调试信息通过 `tracing` 输出到 stderr。stdout 严格保留给 JSON Line 协议，防止日志混入 pipe 流导致解析失败。
- **tokio worker_threads = 2**：在 8 GB 内存约束下，2 个 worker 线程足以处理 pipe I/O + LLM HTTP 请求。不需要更多并发。
- **模块组装使用依赖注入**：所有模块通过构造函数接收依赖，便于测试时替换为 mock 实现。

2.2 Agent Runtime — ReAct 循环

职责：实现 think → act → observe 循环，协调 LLM、Tool、Memory、Critic 完成用户任务。

模块接口：

```rust
/// Agent Runtime 核心结构
pub struct AgentRuntime {
    provider: Box<dyn LlmProvider>,
```

```
    tools: Vec<Box<dyn Tool>>,          // 包含 BrowserPipeTool + MCP tools
    memory: Arc<dyn Memory>,
    critic: Critic,
    config: RuntimeConfig,
}

/// Runtime 配置
pub struct RuntimeConfig {
    pub max_steps: u32,                  // 单次任务最大步数（默认 50）
    pub max_thinking_tokens: u32,        // 单步 thinking 最大 token（默认
4096）
    pub system_prompt_template: String,  // 系统提示模板路径
    pub human_confirm_actions: Vec<String>, // 需人工确认的 action 列表
}

impl AgentRuntime {
    /// 创建 Runtime 实例
    pub fn new(
        provider: Box<dyn LlmProvider>,
        tools: Vec<Box<dyn Tool>>,
        memory: Arc<dyn Memory>,
        critic: Critic,
        config: RuntimeConfig,
    ) -> Self;

    /// 执行用户任务（核心入口）
    /// 返回任务执行结果或错误
    pub async fn execute_task(&mut self, task: &str) -> Result<TaskResult,
AgentError>;

    /// 停止当前任务（由外部 shutdown 信号触发）
    pub fn abort(&self);
}

/// 任务执行结果
pub struct TaskResult {
    pub success: bool,
    pub summary: String,            // LLM 生成的任务完成摘要
    pub steps: Vec<StepRecord>,     // 所有执行步骤记录
    pub token_usage: TokenUsage,    // token 消耗统计
}

/// 单步执行记录
pub struct StepRecord {
    pub step_num: u32,
    pub thinking: String,           // LLM 的推理内容
    pub action: Option<Action>,     // 选择的操作（None = 任务完成）
    pub observation: String,        // 操作结果观察
    pub duration_ms: u64,
}
```

ReAct 循环伪码：

```
execute_task(user_instruction):
    context = memory.load_context(user_instruction)
    system_prompt = render_template(config.system_prompt_template, {
        available_tools: tools.descriptions(),
        skills: skill_loader.list_skills(),
        context: context,
    })

    for step in 1..=config.max_steps:
        // === THINK ===
        llm_response = provider.chat(system_prompt, messages, tools)

        if llm_response.is_final_answer():
            return TaskResult { success: true, summary: llm_response.text }

        // === ACT ===
        action = llm_response.tool_call
        if action.name in config.human_confirm_actions:
            confirmation = request_human_confirm(action)  // 通过 pipe 发送
确认请求
            if not confirmation:
                messages.push(observation: "用户拒绝了此操作")
                continue

        result = tools.execute(action)

        // === OBSERVE ===
        observation = format_observation(result)
        messages.push(observation)

        // === CRITIC ===
        critic_verdict = critic.evaluate(step, action, result)
        if critic_verdict.should_abort:
            return TaskResult { success: false, summary:
critic_verdict.reason }

        memory.record_step(step, action, result)

    return TaskResult { success: false, summary: "达到最大步数限制" }
```

2.3 BrowserPipeTool — 自定义 Tool trait 实现

职责：将 Agent 的 action 请求序列化为 Pipe JSON 命令，发送到 Browser 并等待响应。这是 sgClaw 与 ZeroClaw 框架的核心对接点。

模块接口：

```
/// ZeroClaw 的 Tool trait（框架定义）
#[async_trait]
pub trait Tool: Send + Sync {
```

```
    fn name(&self) -> &str;
    fn description(&self) -> &str;
    fn parameters_schema(&self) -> serde_json::Value;
    async fn execute(&self, params: serde_json::Value) ->
Result<ToolOutput, ToolError>;
}

/// sgClaw 的浏览器操作工具实现
pub struct BrowserPipeTool {
    pipe_writer: Arc<Mutex<PipeWriter>>,
    pipe_reader: Arc<PipeResponseReceiver>,
    mac_policy: Arc<MacPolicy>,
    seq_counter: AtomicU64,
    hmac_key: [u8; 32],
}

impl BrowserPipeTool {
    pub fn new(
        pipe_writer: Arc<Mutex<PipeWriter>>,
        pipe_reader: Arc<PipeResponseReceiver>,
        mac_policy: Arc<MacPolicy>,
        hmac_key: [u8; 32],
    ) -> Self;
}

#[async_trait]
impl Tool for BrowserPipeTool {
    fn name(&self) -> &str { "browser_action" }

    fn description(&self) -> &str {
        "在浏览器中执行操作。支持的操作包括：click, type, navigate, \
         getText, getHtml, waitForSelector, pageScreenshot, select, \
         scrollTo, getAomSnapshot, storageSet, storageGet, \
         zombieSpawn, zombieKill"
    }

    fn parameters_schema(&self) -> serde_json::Value {
        // 返回所有 action 的 JSON Schema (oneOf 结构)
        // 详见 §5 接口契约
    }

    async fn execute(&self, params: serde_json::Value) ->
Result<ToolOutput, ToolError> {
        // 1. 解析 action + params
        let action: Action = serde_json::from_value(params)?;

        // 2. MAC 策略校验
        self.mac_policy.check(&action)?;

        // 3. 构造 pipe message
        let seq = self.seq_counter.fetch_add(1, Ordering::SeqCst);
        let msg = PipeMessage::command(seq, &action, &self.hmac_key);

        // 4. 发送到 Browser
```

```
        self.pipe_writer.lock().await.send(&msg)?;

        // 5. 等待对应 seq 的 response (超时 30s)
        let response = self.pipe_reader.recv(seq,
Duration::from_secs(30)).await?;

        // 6. 转换为 ToolOutput
        Ok(ToolOutput::from_pipe_response(response))
    }
}
```

Action 枚举（白名单）：

```
/// 允许通过 pipe 的操作类型 (封闭枚举)
#[derive(Debug, Serialize, Deserialize)]
#[serde(tag = "action", content = "params")]
pub enum Action {
    #[serde(rename = "click")]
    Click { selector: String, wait_after: Option<u64> },

    #[serde(rename = "type")]
    Type { selector: String, text: String, clear_first: Option<bool> },

    #[serde(rename = "navigate")]
    Navigate { url: String },

    #[serde(rename = "getText")]
    GetText { selector: String },

    #[serde(rename = "getHtml")]
    GetHtml { selector: String, outer: Option<bool> },

    #[serde(rename = "waitForSelector")]
    WaitForSelector { selector: String, timeout_ms: Option<u64> },

    #[serde(rename = "pageScreenshot")]
    PageScreenshot { full_page: Option<bool> },

    #[serde(rename = "select")]
    Select { selector: String, value: String },

    #[serde(rename = "scrollTo")]
    ScrollTo { selector: Option<String>, x: Option<i32>, y: Option<i32> },

    #[serde(rename = "getAomSnapshot")]
    GetAomSnapshot { root_selector: Option<String> },

    #[serde(rename = "storageSet")]
    StorageSet { key: String, value: String },

    #[serde(rename = "storageGet")]
    StorageGet { key: String },
```

```
    #[serde(rename = "zombieSpawn")]
    ZombieSpawn { url: String },

    #[serde(rename = "zombieKill")]
    ZombieKill { page_id: String },
}
```

2.4 SkillLoader — 技能加载器

职责：发现、校验、加载 JavaScript 业务技能脚本，将其注册到 Agent 的工具集中。

模块接口：

```
/// 技能加载器
pub struct SkillLoader {
    skills_dir: PathBuf,
    registry: HashMap<String, Skill>,
    signature_verifier: SignatureVerifier,
}

/// 技能定义
pub struct Skill {
    pub name: String,                // 技能名称 (如 "oa-approval")
    pub version: String,             // 语义版本 (如 "1.2.0")
    pub description: String,         // 自然语言描述 (供 LLM 选择)
    pub target_domains: Vec<String>,// 适用域名列表
    pub parameters: JsonSchema,      // 输入参数 schema
    pub script: String,              // JS 脚本内容
    pub signature: String,           // 数字签名 (Ed25519)
}

/// 技能清单文件 (registry.json)
pub struct SkillRegistry {
    pub version: String,
    pub skills: Vec<SkillManifestEntry>,
}

pub struct SkillManifestEntry {
    pub name: String,
    pub file: String,               // 相对路径 (如 "builtin/oa-approval.js")
    pub hash: String,               // SHA-256 文件哈希
    pub signature: String,          // Ed25519 签名
}

impl SkillLoader {
    /// 创建加载器并扫描技能目录
    pub fn new(skills_dir: PathBuf, public_key: &[u8]) -> Result<Self,
SkillError>;

    /// 获取所有已加载技能的描述 (供 LLM system prompt)
```

```rust
    pub fn list_skills(&self) -> Vec<SkillDescription>;

    /// 获取指定技能的 JS 脚本
    pub fn get_script(&self, name: &str) -> Option<&str>;

    /// 运行时重新加载技能（热更新）
    pub fn reload(&mut self) -> Result<(), SkillError>;
}
```

技能加载流程：

```
SkillLoader::new(skills_dir)
  │
  ├── 1. 读取 skills/registry.json
  │      └── 解析技能清单
  │
  ├── 2. 遍历每个 SkillManifestEntry
  │      │
  │      ├── 2a. 读取技能文件
  │      │
  │      ├── 2b. 计算文件 SHA-256
  │      │      └── 与 registry.json 中的 hash 比对
  │      │      └── 不匹配 → 跳过 + 警告日志
  │      │
  │      ├── 2c. 验证 Ed25519 签名
  │      │      └── 使用编译时内嵌的公钥
  │      │      └── 签名无效 → 跳过 + 警告日志
  │      │
  │      └── 2d. 解析技能元数据（JS 文件头部注释）
  │             └── 注册到 registry HashMap
  │
  └── 3. 日志："Loaded N skills, skipped M (signature failed)"
```

技能文件格式（JS 文件头部）：

```javascript
/**
 * @skill oa-approval
 * @version 1.2.0
 * @description OA系统审批单自动处理：查询待审批列表、批量审批、添加审批意见
 * @domains oa.example.com, oa-test.example.com
 * @params {
 *   "type": "object",
 *   "properties": {
 *     "action": { "enum": ["list", "approve", "reject"] },
 *     "opinion": { "type": "string" }
 *   }
 * }
 */
async function execute(params, browserAction) {
```

```
    // browserAction 是 BrowserAction 函数的引用
    // 技能代码使用 browserAction 调用浏览器操作

    if (params.action === 'list') {
        await browserAction('navigate',
'https://oa.example.com/approval/pending');
        await browserAction('waitForSelector', '.approval-list');
        const items = await browserAction('getText', '.approval-list');
        return { success: true, data: items };
    }
    // ...
}
```

2.5 MAC Policy — 强制访问控制策略

职责：在 Rust 端实施域名白名单和 Action 白名单校验，作为安全架构 Layer 2 的核心组件。

模块接口：

```
/// MAC 策略引擎
pub struct MacPolicy {
    allowed_domains: HashSet<String>,      // 允许操作的域名集合
    allowed_actions: HashSet<String>,      // 允许的 action 名称集合
    blocked_actions: HashSet<String>,      // 硬性禁止的 action 集合
    confirm_actions: HashSet<String>,      // 需人工确认的 action 集合
    storage_key_prefix: String,            // 存储 key 前缀限制（默认
"sgclaw."）
    rate_limits: HashMap<String, RateLimit>, // 每域速率限制
}

/// 速率限制配置
pub struct RateLimit {
    pub max_per_second: u32,            // 每秒最大操作数
    pub cooldown_seconds: u32,          // 超限后冷却时间
}

/// MAC 校验结果
pub enum MacVerdict {
    Allow,                              // 允许执行
    NeedConfirm(String),                // 需要用户确认（附确认提示）
    Deny(MacDenyReason),                // 拒绝执行
}

/// 拒绝原因
pub enum MacDenyReason {
    ActionNotAllowed(String),           // action 不在白名单
    ActionBlocked(String),              // action 在黑名单（eval 等）
    DomainNotAllowed(String),           // 域名不在白名单
    StorageKeyViolation(String),        // storage key 前缀不合规
    RateLimitExceeded(String, u32),     // 超过速率限制（域名，当前速率）
}
```

```
impl MacPolicy {
    /// 从 rules.json 加载策略
    pub fn from_rules_file(path: &Path) -> Result<Self, PolicyError>;

    /// 校验 action 是否允许
    pub fn check(&self, action: &Action) -> MacVerdict;

    /// 校验域名是否在白名单
    pub fn check_domain(&self, domain: &str) -> bool;

    /// 记录一次操作（用于速率统计）
    pub fn record_access(&self, domain: &str);
}
```

rules.json 策略文件格式：

```
{
  "version": "1.0",
  "domains": {
    "allowed": [
      "oa.example.com",
      "erp.example.com",
      "hr.example.com",
      "finance.example.com"
    ]
  },
  "pipe_actions": {
    "allowed": [
      "click", "type", "navigate", "getText", "getHtml",
      "waitForSelector", "pageScreenshot", "select", "scrollTo",
      "getAomSnapshot", "storageSet", "storageGet",
      "zombieSpawn", "zombieKill"
    ],
    "blocked": [
      "eval", "executeJsInPage", "registerJsFunction",
      "setRequestInterceptor", "exportCookies"
    ],
    "need_confirm": [
      "sessionLogin", "sessionLogout", "clearStorage"
    ]
  },
  "storage": {
    "key_prefix": "sgclaw."
  },
  "rate_limits": {
    "default": { "max_per_second": 10, "cooldown_seconds": 30 },
    "overrides": {
      "finance.example.com": { "max_per_second": 5, "cooldown_seconds": 60
}
    }
```

```
    }
}
```

2.6 Critic — 输出质量评估器

职责：在 Agent 每步 observe 后评估操作结果质量，检测异常模式，触发熔断保护。

模块接口：

```rust
/// 输出质量评估器
pub struct Critic {
    circuit_breaker: CircuitBreaker,
    memory: Arc<dyn Memory>,
    config: CriticConfig,
}

pub struct CriticConfig {
    pub consecutive_failure_threshold: u32,  // 连续失败熔断阈值（默认 10）
    pub same_action_repeat_limit: u32,       // 同一 action 重复上限（默认 5）
    pub max_task_duration_secs: u64,         // 单任务最大时长（默认 600）
}

/// 评估结果
pub struct CriticVerdict {
    pub should_abort: bool,              // 是否应终止任务
    pub reason: Option<String>,          // 终止原因
    pub warning: Option<String>,         // 警告（不终止但需记录）
}

/// 熔断器
pub struct CircuitBreaker {
    state: CircuitState,
    consecutive_failures: AtomicU32,
    last_failure_time: Mutex<Option<Instant>>,
    config: CircuitBreakerConfig,
}

pub enum CircuitState {
    Closed,          // 正常 — 允许操作
    Open,            // 断开 — 拒绝所有操作
    HalfOpen,        // 半开 — 允许一次试探
}

pub struct CircuitBreakerConfig {
    pub failure_threshold: u32,          // 失败次数阈值（默认 10）
    pub cooldown_base_secs: u64,         // 基础冷却时间（默认 1）
    pub cooldown_max_secs: u64,          // 最大冷却时间（默认 30）
    pub reset_on_success: bool,          // 成功后是否重置计数（默认 true）
}

impl Critic {
```

```
    pub fn new(memory: Arc<dyn Memory>, config: CriticConfig) -> Self;

    /// 评估单步结果
    pub fn evaluate(
        &self,
        step: u32,
        action: &Action,
        result: &ToolOutput,
        elapsed: Duration,
    ) -> CriticVerdict;

    /// 重置熔断器（用户手动恢复时调用）
    pub fn reset(&self);
}
```

评估逻辑：

```
evaluate(step, action, result, elapsed):
    // 1. 检查操作是否失败
    if result.is_error():
        circuit_breaker.record_failure()
        if circuit_breaker.state == Open:
            return Verdict { should_abort: true, reason: "连续失败次数达到熔断
阈值" }

    // 2. 检查是否在重复同一操作
    recent_actions =
memory.get_recent_actions(config.same_action_repeat_limit)
    if all_same(recent_actions, action):
        return Verdict { should_abort: true, reason: "检测到死循环：连续重复相
同操作" }

    // 3. 检查任务时长
    if elapsed > config.max_task_duration_secs:
        return Verdict { should_abort: true, reason: "任务执行超时" }

    // 4. 成功时重置计数
    if result.is_success() && config.reset_on_success:
        circuit_breaker.reset_failures()

    return Verdict { should_abort: false }
```

2.7 Memory — 记忆系统

职责：管理 Agent 的短期对话记忆和长期任务知识库，支持上下文检索和经验沉淀。

模块接口：

```rust
/// Memory trait (ZeroClaw 框架定义)
#[async_trait]
pub trait Memory: Send + Sync {
    /// 存储对话消息
    async fn store_message(&self, msg: &Message) -> Result<(),
MemoryError>;

    /// 检索相关上下文 (语义搜索)
    async fn retrieve(&self, query: &str, limit: usize) ->
Result<Vec<MemoryEntry>, MemoryError>;

    /// 清除当前会话记忆
    async fn clear_session(&self) -> Result<(), MemoryError>;
}

/// sgClaw 的复合记忆实现
pub struct CompositeMemory {
    short_term: ShortTermMemory,        // Ring Buffer 短期记忆
    long_term: LongTermMemory,          // SQLite 长期记忆
}

/// 短期记忆 (Ring Buffer)
pub struct ShortTermMemory {
    buffer: VecDeque<Message>,
    max_size: usize,                    // 默认 50 条消息
    max_tokens: usize,                  // 默认 8000 tokens
}

/// 长期记忆 (SQLite + 简单向量)
pub struct LongTermMemory {
    db: SqlitePool,                     // SQLite 连接池
    embedding_dim: usize,               // 向量维度 (默认 384)
}

/// 记忆条目
pub struct MemoryEntry {
    pub id: String,
    pub content: String,
    pub entry_type: MemoryType,
    pub relevance_score: f32,           // 与查询的相关度 (0.0 ~ 1.0)
    pub created_at: DateTime<Utc>,
}

pub enum MemoryType {
    Conversation,                       // 对话历史
    TaskResult,                         // 任务执行结果
    SkillExperience,                    // 技能执行经验
    UserPreference,                     // 用户偏好
}

impl CompositeMemory {
    pub fn new(config: MemoryConfig) -> Result<Self, MemoryError>;
```

```
    /// 获取最近 N 步的 action 记录（供 Critic 查重）
    pub fn get_recent_actions(&self, n: usize) -> Vec<Action>;

    /// 保存任务执行经验（供自进化学习）
    pub async fn save_task_experience(
        &self,
        task: &str,
        result: &TaskResult,
    ) -> Result<(), MemoryError>;
}
```

存储 Schema (SQLite):

```
-- 长期记忆表
CREATE TABLE memory_entries (
    id          TEXT PRIMARY KEY,
    content     TEXT NOT NULL,
    entry_type  TEXT NOT NULL,          -- 'conversation' | 'task_result'
| 'skill_exp' | 'user_pref'
    embedding   BLOB,                   -- f32 向量（384 维 = 1536 bytes）
    metadata    TEXT,                   -- JSON 附加信息
    created_at  TEXT NOT NULL,
    session_id  TEXT                    -- 关联的 trace_id
);

-- 向量索引（简单线性扫描，数据量小时足够）
CREATE INDEX idx_memory_type ON memory_entries(entry_type);
CREATE INDEX idx_memory_session ON memory_entries(session_id);

-- 技能执行记录表
CREATE TABLE skill_executions (
    id          TEXT PRIMARY KEY,
    skill_name  TEXT NOT NULL,
    input_hash  TEXT NOT NULL,          -- 输入参数 hash（用于精确匹配）
    success     INTEGER NOT NULL,
    duration_ms INTEGER,
    created_at  TEXT NOT NULL
);
```

2.8 LLM Provider — 模型适配层

职责：统一封装不同 LLM 服务的调用接口，支持 streaming、tool-use、token 计量。

模块接口:

```
/// LLM Provider trait（ZeroClaw 框架定义）
#[async_trait]
pub trait LlmProvider: Send + Sync {
    /// 发送聊天请求
```

```
    async fn chat(
        &self,
        messages: &[Message],
        tools: &[ToolDefinition],
        config: &ChatConfig,
    ) -> Result<LlmResponse, LlmError>;

    /// 发送 streaming 请求
    async fn chat_stream(
        &self,
        messages: &[Message],
        tools: &[ToolDefinition],
        config: &ChatConfig,
    ) -> Result<Pin<Box<dyn Stream<Item = Result<StreamChunk, LlmError>>>>,
LlmError>;

    /// 获取 provider 信息
    fn info(&self) -> ProviderInfo;
}

/// 聊天配置
pub struct ChatConfig {
    pub max_tokens: u32,              // 最大生成 token 数
    pub temperature: f32,             // 温度（默认 0.1，Agent 场景偏低）
    pub stop_sequences: Vec<String>,  // 停止序列
}

/// LLM 响应
pub struct LlmResponse {
    pub content: Option<String>,      // 文本回复
    pub tool_calls: Vec<ToolCall>,    // 工具调用请求
    pub usage: TokenUsage,            // token 使用量
    pub finish_reason: FinishReason,
}

/// Token 使用统计
pub struct TokenUsage {
    pub prompt_tokens: u32,
    pub completion_tokens: u32,
    pub total_tokens: u32,
}

/// Provider 信息
pub struct ProviderInfo {
    pub name: String,                 // "claude" | "openai" | "ollama"
    pub model: String,                // 如 "claude-sonnet-4-20250514"
    pub max_context_tokens: u32,      // 最大上下文窗口
    pub supports_tools: bool,
    pub supports_streaming: bool,
}
```

已实现的 Provider：

Provider	模块	支持功能	适用场景
Claude	`llm/claude.rs`	streaming + tool-use + vision	默认推荐，综合能力最强
OpenAI	`llm/openai.rs`	streaming + tool-use	兼容 GPT-4 及 API 兼容服务
Ollama	`llm/ollama.rs`	streaming + tool-use (部分模型)	本地部署，离线场景

Provider 选择逻辑：

```
/// 根据配置创建 Provider 实例
pub fn create_provider(config: &LlmConfig) -> Result<Box<dyn LlmProvider>,
LlmError> {
    match config.provider.as_str() {
        "claude" => Ok(Box::new(ClaudeProvider::new(
            &config.api_key,
            &config.model,   // 如 "claude-sonnet-4-20250514"
            config.base_url.as_deref(),
        )?)),
        "openai" => Ok(Box::new(OpenAiProvider::new(
            &config.api_key,
            &config.model,   // 如 "gpt-4o"
            config.base_url.as_deref(),
        )?)),
        "ollama" => Ok(Box::new(OllamaProvider::new(
            &config.model,   // 如 "qwen2.5:32b"
            config.base_url.as_deref().unwrap_or("http://localhost:11434"),
        )?)),
        other => Err(LlmError::UnknownProvider(other.to_string())),
    }
}
```

2.9 MCP Client — 外部工具扩展

职责：通过 rmcp（Rust MCP SDK）连接外部 MCP Server，动态获取额外工具供 Agent 使用。

模块接口：

```
/// MCP 客户端管理器
pub struct McpClientManager {
    clients: HashMap<String, McpClient>,    // server_name → client
    config: McpConfig,
}

pub struct McpConfig {
    pub servers: Vec<McpServerConfig>,
}

pub struct McpServerConfig {
    pub name: String,                       // 服务器标识名
    pub command: String,                    // 启动命令
```

```
    pub args: Vec<String>,                    // 命令参数
    pub env: HashMap<String, String>,         // 环境变量
}

impl McpClientManager {
    /// 创建并连接所有配置的 MCP Server
    pub async fn new(config: McpConfig) -> Result<Self, McpError>;

    /// 获取所有 MCP Server 提供的工具列表
    pub fn list_tools(&self) -> Vec<ToolDefinition>;

    /// 调用 MCP 工具
    pub async fn call_tool(
        &self,
        server: &str,
        tool: &str,
        params: serde_json::Value,
    ) -> Result<serde_json::Value, McpError>;

    /// 关闭所有连接
    pub async fn shutdown(&mut self);
}
```

---

3. 浏览器 C++ 端模块详细设计

3.1 SgClawProcessHost — 进程宿主

职责：Singleton，管理 sgclaw 子进程的完整生命周期（Start / Stop / OnCrash），创建和持有 STDIO Pipe。

接口声明：

```
// sg_claw_process_host.h

#ifndef CHROME_BROWSER_SUPERRPA_SGCLAW_SG_CLAW_PROCESS_HOST_H_
#define CHROME_BROWSER_SUPERRPA_SGCLAW_SG_CLAW_PROCESS_HOST_H_

#include <memory>
#include <string>
#include "base/process/launch.h"
#include "base/process/process.h"

namespace superrpa {
namespace sgclaw {

class PipeListener;

// sgClaw 进程状态
enum class ProcessState {
  kStopped,       // 初始状态 / 已停止
  kStarting,      // 正在启动（等待 handshake）
```

```
  kRunning,       // 正常运行
  kStopping,      // 正在停止
  kCrashed,       // 异常退出
};

// 进程事件观察者
class ProcessObserver {
 public:
  virtual ~ProcessObserver() = default;
  virtual void OnStateChanged(ProcessState new_state) = 0;
  virtual void OnPipeMessage(const std::string& json_line) = 0;
  virtual void OnProcessCrash(int exit_code, const std::string&
stderr_output) = 0;
};

// sgClaw 进程宿主 (Singleton)
class SgClawProcessHost {
 public:
  static SgClawProcessHost* GetInstance();

  // 启动 sgClaw 子进程
  // 成功返回 true，失败返回 false 并通过 observer 通知
  bool Start();

  // 优雅停止 sgClaw
  void Stop();

  // 发送消息到 sgClaw
  bool SendMessage(const std::string& json_line);

  // 获取当前状态
  ProcessState GetState() const;

  // 注册/移除观察者
  void AddObserver(ProcessObserver* observer);
  void RemoveObserver(ProcessObserver* observer);

 private:
  SgClawProcessHost();
  ~SgClawProcessHost();

  // 禁止拷贝
  SgClawProcessHost(const SgClawProcessHost&) = delete;
  SgClawProcessHost& operator=(const SgClawProcessHost&) = delete;

  // 内部方法
  base::FilePath GetSgClawBinaryPath() const;
  void OnChildProcessExited(int exit_code);
  void DoHandshake();

  // 成员
  ProcessState state_ = ProcessState::kStopped;
  base::Process child_process_;
  std::unique_ptr<PipeListener> pipe_listener_;
```

```
  // Pipe 文件描述符 (Linux: fd, Windows: HANDLE)
  base::ScopedFD pipe_write_fd_;     // Browser → sgClaw
  base::ScopedFD pipe_read_fd_;      // sgClaw → Browser

  std::vector<ProcessObserver*> observers_;
};

}  // namespace sgclaw
}  // namespace superrpa

#endif
```

关键实现说明:

- **Singleton**：使用 `base::NoDestructor<SgClawProcessHost>` 实现进程全局唯一实例
- **Start() 流程**：检查二进制存在 → 创建 pipe pair → `base::LaunchProcess` → 启动 PipeListener → 发送 handshake → 等待 ack (5s 超时) → 状态切换到 Running
- **Stop() 流程**：发送 `{"type":"shutdown"}` → 等待进程退出 (2s 超时) → SIGTERM → 再等 2s → SIGKILL
- **OnChildProcessExited()**：检查退出码，非零且非主动 Stop → 视为 Crash → 通知 observers → 不自动重启
- **线程安全**：所有方法在 Browser UI 线程调用，Pipe 读取在 IO 线程

3.2 PipeListener — 异步读取循环

职责：在 IO 线程上异步读取 sgClaw 的 stdout 输出，按行解析 JSON，分发到 MAC 校验和 CommandRouter。

接口声明:

```
// pipe_listener.h

#ifndef CHROME_BROWSER_SUPERRPA_SGCLAW_PIPE_LISTENER_H_
#define CHROME_BROWSER_SUPERRPA_SGCLAW_PIPE_LISTENER_H_

#include <string>
#include <functional>
#include "base/files/file_descriptor_watcher_posix.h"  // Linux
// Windows: base/win/object_watcher.h

namespace superrpa {
namespace sgclaw {

class PipeListener {
 public:
  using MessageCallback = std::function<void(const std::string&
json_line)>;
  using ErrorCallback = std::function<void(const std::string& error)>;
```

```
  PipeListener(base::ScopedFD read_fd,
               MessageCallback on_message,
               ErrorCallback on_error);
  ~PipeListener();

  // 开始异步监听
  void StartListening();

  // 停止监听
  void StopListening();

 private:
  void OnReadable();                    // fd 可读回调
  void ProcessBuffer();                 // 从 buffer 中提取完整行
  bool ValidateMessageSize(const std::string& line);  // 检查消息大小 (<=1MB)

  base::ScopedFD read_fd_;
  std::string read_buffer_;             // 累积读取缓冲
  MessageCallback on_message_;
  ErrorCallback on_error_;

  // Linux: FileDescriptorWatcher
  std::unique_ptr<base::FileDescriptorWatcher::Controller> fd_watcher_;
};

}  // namespace sgclaw
}  // namespace superrpa

#endif
```

读取流程：

```
StartListening()
  │
  └── 注册 fd_watcher_ 监听 read_fd_ 可读事件
        │
        ▼
OnReadable()  [IO 线程回调]
  │
  ├── read(read_fd_, temp_buffer, 4096)
  │     → 返回 0: EOF → 通知 process exited
  │     → 返回 -1: EAGAIN → 忽略 (非阻塞 I/O)
  │     → 返回 N: 追加到 read_buffer_
  │
  └── ProcessBuffer()
        │
        ├── 在 read_buffer_ 中查找 '\n'
        │     → 找到: 提取该行作为 json_line
        │     → 未找到: 等待下次 OnReadable()
        │
        ├── ValidateMessageSize(json_line)
        │     → > 1MB: 丢弃 + 日志警告
```

```
        │
        └── on_message_(json_line)
              │
              └── [UI 线程] MAC 校验 → CommandRouter
```

3.3 MAC Whitelist Check — Pipe 来源 MAC 校验

职责：对通过 Pipe 收到的命令进行强制访问控制校验，作为安全架构 Layer 3 的实现。

接口声明：

```cpp
// mac_whitelist_check.h

#ifndef CHROME_BROWSER_SUPERRPA_SGCLAW_MAC_WHITELIST_CHECK_H_
#define CHROME_BROWSER_SUPERRPA_SGCLAW_MAC_WHITELIST_CHECK_H_

#include <string>
#include <set>
#include "base/values.h"

namespace superrpa {
namespace sgclaw {

// MAC 校验结果
enum class MacResult {
  kAllow,                 // 允许
  kDeny,                  // 拒绝
  kNeedConfirm,           // 需用户确认
};

struct MacCheckResult {
  MacResult result;
  std::string reason;     // 拒绝/确认原因
};

class MacWhitelistCheck {
 public:
  // 从 rules.json 加载策略
  static std::unique_ptr<MacWhitelistCheck> CreateFromRules(
      const base::FilePath& rules_path);

  // 校验 pipe 命令
  MacCheckResult Check(const std::string& action,
                       const std::string& expected_domain,
                       const std::string& current_page_domain) const;

 private:
  MacWhitelistCheck();

  std::set<std::string> allowed_actions_;
  std::set<std::string> blocked_actions_;
```

```
  std::set<std::string> confirm_actions_;
  std::set<std::string> allowed_domains_;
};

}  // namespace sgclaw
}  // namespace superrpa

#endif
```

校验流程:

```
Check(action, expected_domain, current_page_domain):
  │
  ├── 1. action 在 blocked_actions_ 中?
  │     → 是: return Deny("Action is blocked: " + action)
  │
  ├── 2. action 不在 allowed_actions_ 中?
  │     → 是: return Deny("Action not in pipe whitelist: " + action)
  │
  ├── 3. expected_domain 不在 allowed_domains_ 中?
  │     → 是: return Deny("Domain not in whitelist: " + expected_domain)
  │
  ├── 4. expected_domain != current_page_domain?
  │     → 是: return Deny("Domain mismatch: expected " + expected_domain
  │                       + " but current is " + current_page_domain)
  │
  ├── 5. action 在 confirm_actions_ 中?
  │     → 是: return NeedConfirm("操作需要用户确认: " + action)
  │
  └── 6. return Allow
```

---

4. 前端 Side Panel 新增 UI

4.1 AgentControlPanel.vue

职责：在 Side Panel 中提供 Agent 控制面板，包括启停按钮、指令输入、状态显示、执行日志流。

组件接口:

```
AgentControlPanel.vue
│
├── Props: 无 (独立组件)
│
├── State:
│   ├── agentState: 'stopped' | 'starting' | 'running' | 'error'
│   ├── taskInput: string          // 用户输入的自然语言指令
│   ├── logs: LogEntry[]           // 执行日志条目
│   ├── currentTask: string        // 当前执行的任务描述
```

```
│   └── errorMessage: string       // 错误信息
│
├── Methods:
│   ├── startAgent()               // 调用 FunctionsUI →
SgClawProcessHost::Start()
│   ├── stopAgent()                // 调用 FunctionsUI →
SgClawProcessHost::Stop()
│   ├── submitTask(instruction)    // 发送任务到 Agent
│   ├── confirmAction(actionId)    // 用户确认 human-in-the-loop 操作
│   └── rejectAction(actionId)     // 用户拒绝操作
│
└── Events (from C++ via FunctionsUI):
    ├── onStateChanged(state)      // Agent 状态变更
    ├── onLogEntry(entry)          // 新日志条目
    ├── onConfirmRequired(action)  // 需要用户确认的操作
    └── onTaskCompleted(result)    // 任务完成
```

UI 布局：

```
┌──────────────────────────────────────────┐
│  sgClaw Agent                            │
├──────────────────────────────────────────┤
│                                          │
│  状态: ● 运行中                          │
│                                          │
│  ┌──────────────────────────────┐        │
│  │ 请输入业务指令...            │        │
│  │                              │        │
│  └──────────────────────────────┘        │
│  [ 发送 ]                                │
│                                          │
│  ── 执行日志 ──────────────────────      │
│                                          │
│  ▶ [10:23:01] 正在分析指令...            │
│  ▶ [10:23:03] 导航至 ERP 系统            │
│  ▶ [10:23:05] 点击"财务报表"菜单         │
│  ▶ [10:23:08] 设置时间范围: 本月         │
│  ▶ [10:23:10] 正在导出数据...            │
│                                          │
│  ── 确认请求 ──────────────────────      │
│                                          │
│  ⚠ Agent 请求执行登录操作                │
│  目标系统: erp.example.com               │
│  [ 允许 ]  [ 拒绝 ]                      │
│                                          │
├──────────────────────────────────────────┤
│  [ 启动 Agent ]  [ 停止 Agent ]          │
└──────────────────────────────────────────┘
```

与 C++ 层的通信：

前端通过 FunctionsUI 机制（SuperRPA 已有的 JS → C++ IPC）与 SgClawProcessHost 交互。

```
// 启动 Agent
window.sgFunctionsUI('sgclaw_start', {}, (response) => {
    // response: { success: true } 或 { success: false, error: "..." }
});

// 停止 Agent
window.sgFunctionsUI('sgclaw_stop', {}, (response) => { ... });

// 发送任务
window.sgFunctionsUI('sgclaw_submit_task', {
    instruction: "导出本月合规报表"
}, (response) => { ... });

// 确认/拒绝操作
window.sgFunctionsUI('sgclaw_confirm', {
    action_id: "abc123",
    approved: true
}, (response) => { ... });
```

事件接收（C++ → JS 推送）：

```
// C++ 通过 ExecuteJavaScript 推送事件到 Side Panel
// 前端注册全局事件处理器
window.sgClawEvents = {
    onStateChanged: function(state) {
        // state: 'stopped' | 'starting' | 'running' | 'error'
        vm.agentState = state;
    },
    onLogEntry: function(entry) {
        // entry: { time: "10:23:01", message: "正在分析指令...", level:
"info" }
        vm.logs.push(entry);
    },
    onConfirmRequired: function(action) {
        // action: { id: "abc123", type: "sessionLogin", domain:
"erp.example.com" }
        vm.pendingConfirm = action;
    },
    onTaskCompleted: function(result) {
        // result: { success: true, summary: "已成功导出3份报表" }
        vm.currentTask = null;
    }
};
```

---

5. 接口契约

5.1 Pipe JSON 协议完整 Schema

以下定义了 sgClaw 与 Browser 之间所有 Pipe 消息的完整 JSON Schema。

Handshake — Browser → sgClaw：

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "PipeInitMessage",
  "type": "object",
  "required": ["type", "version", "hmac_seed"],
  "properties": {
    "type": { "const": "init" },
    "version": {
      "type": "string",
      "pattern": "^\\d+\\.\\d+$",
      "description": "协议版本号 (如 1.0)"
    },
    "hmac_seed": {
      "type": "string",
      "minLength": 32,
      "maxLength": 64,
      "description": "HMAC 密钥种子 (hex 编码)"
    },
    "capabilities": {
      "type": "array",
      "items": { "type": "string" },
      "description": "Browser 支持的扩展能力列表"
    }
  }
}
```

Handshake — sgClaw → Browser：

```
{
  "title": "PipeInitAckMessage",
  "type": "object",
  "required": ["type", "version", "agent_id"],
  "properties": {
    "type": { "const": "init_ack" },
    "version": {
      "type": "string",
      "description": "sgClaw 协议版本 (必须与 init.version 一致)"
    },
    "agent_id": {
      "type": "string",
      "description": "Agent 实例唯一标识 (UUID v4)"
    },
    "supported_actions": {
      "type": "array",
```

```
        "items": { "type": "string" },
        "description": "sgClaw 可使用的 action 列表"
      }
    }
  }
```

Command — sgClaw → Browser：

```
{
  "title": "PipeCommandMessage",
  "type": "object",
  "required": ["seq", "type", "action", "params", "security"],
  "properties": {
    "seq": {
      "type": "integer",
      "minimum": 1,
      "description": "全局递增序列号"
    },
    "type": { "const": "command" },
    "action": {
      "type": "string",
      "enum": [
        "click", "type", "navigate", "getText", "getHtml",
        "waitForSelector", "pageScreenshot", "select", "scrollTo",
        "getAomSnapshot", "storageSet", "storageGet",
        "zombieSpawn", "zombieKill"
      ]
    },
    "params": {
      "type": "object",
      "description": "操作参数，格式取决于 action（见 §5.2）"
    },
    "security": {
      "type": "object",
      "required": ["expected_domain", "hmac"],
      "properties": {
        "expected_domain": {
          "type": "string",
          "description": "期望操作的目标域名"
        },
        "hmac": {
          "type": "string",
          "description": "HMAC-SHA256 签名（hex）"
        }
      }
    }
  }
}
```

Response — Browser → sgClaw：

```
{
  "title": "PipeResponseMessage",
  "type": "object",
  "required": ["seq", "type", "success"],
  "properties": {
    "seq": {
      "type": "integer",
      "description": "对应 command 的序列号"
    },
    "type": { "const": "response" },
    "success": { "type": "boolean" },
    "data": {
      "type": "object",
      "description": "操作结果数据 (成功时)"
    },
    "error": {
      "type": "object",
      "properties": {
        "code": { "type": "string" },
        "message": { "type": "string" }
      },
      "description": "错误信息 (失败时)"
    },
    "aom_snapshot": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "role": { "type": "string" },
          "name": { "type": "string" },
          "bounds": {
            "type": "array",
            "items": { "type": "integer" },
            "minItems": 4,
            "maxItems": 4,
            "description": "[x, y, width, height]"
          },
          "value": { "type": "string" },
          "children": { "type": "array" }
        }
      },
      "description": "操作后的 AOM 快照"
    },
    "timing": {
      "type": "object",
      "properties": {
        "queue_ms": { "type": "integer" },
        "exec_ms": { "type": "integer" }
      }
    }
  }
}
```

5.2 各 Action 参数 Schema

click：

```
{
  "type": "object",
  "required": ["selector"],
  "properties": {
    "selector": {
      "type": "string",
      "description": "CSS 选择器"
    },
    "wait_after": {
      "type": "integer",
      "minimum": 0,
      "maximum": 30000,
      "default": 1000,
      "description": "点击后等待时间 (ms)"
    }
  }
}
```

type：

```
{
  "type": "object",
  "required": ["selector", "text"],
  "properties": {
    "selector": {
      "type": "string",
      "description": "CSS 选择器"
    },
    "text": {
      "type": "string",
      "maxLength": 10000,
      "description": "要输入的文本"
    },
    "clear_first": {
      "type": "boolean",
      "default": true,
      "description": "输入前是否清空已有内容"
    }
  }
}
```

navigate：

```
{
  "type": "object",
  "required": ["url"],
  "properties": {
    "url": {
      "type": "string",
      "format": "uri",
      "description": "目标 URL（必须在域白名单内）"
    }
  }
}
```

getText：

```
{
  "type": "object",
  "required": ["selector"],
  "properties": {
    "selector": {
      "type": "string",
      "description": "CSS 选择器"
    }
  }
}
```

getHtml：

```
{
  "type": "object",
  "required": ["selector"],
  "properties": {
    "selector": {
      "type": "string"
    },
    "outer": {
      "type": "boolean",
      "default": false,
      "description": "true 返回 outerHTML，false 返回 innerHTML"
    }
  }
}
```

waitForSelector：

```
{
  "type": "object",
```

```
    "required": ["selector"],
    "properties": {
      "selector": {
        "type": "string"
      },
      "timeout_ms": {
        "type": "integer",
        "minimum": 100,
        "maximum": 30000,
        "default": 5000,
        "description": "等待超时 (ms)"
      }
    }
  }
```

pageScreenshot：

```
{
  "type": "object",
  "properties": {
    "full_page": {
      "type": "boolean",
      "default": false,
      "description": "是否截取整页 (true) 或仅可视区域 (false)"
    }
  }
}
```

select：

```
{
  "type": "object",
  "required": ["selector", "value"],
  "properties": {
    "selector": {
      "type": "string",
      "description": "select 元素的 CSS 选择器"
    },
    "value": {
      "type": "string",
      "description": "要选择的 option 的 value 值"
    }
  }
}
```

scrollTo：

```
{
  "type": "object",
  "properties": {
    "selector": {
      "type": "string",
      "description": "滚动到指定元素（优先于 x/y）"
    },
    "x": {
      "type": "integer",
      "description": "水平滚动位置（px）"
    },
    "y": {
      "type": "integer",
      "description": "垂直滚动位置（px）"
    }
  }
}
```

getAomSnapshot：

```
{
  "type": "object",
  "properties": {
    "root_selector": {
      "type": "string",
      "description": "从指定元素开始获取 AOM 子树（默认整页）"
    }
  }
}
```

storageSet：

```
{
  "type": "object",
  "required": ["key", "value"],
  "properties": {
    "key": {
      "type": "string",
      "pattern": "^sgclaw\\.",
      "description": "存储键名（必须以 sgclaw. 开头）"
    },
    "value": {
      "type": "string",
      "maxLength": 65536,
      "description": "存储值"
    }
  }
}
```

storageGet：

```
{
  "type": "object",
  "required": ["key"],
  "properties": {
    "key": {
      "type": "string",
      "pattern": "^sgclaw\\.",
      "description": "存储键名（必须以 sgclaw. 开头）"
    }
  }
}
```

zombieSpawn：

```
{
  "type": "object",
  "required": ["url"],
  "properties": {
    "url": {
      "type": "string",
      "format": "uri",
      "description": "Zombie page 加载的 URL（必须在域白名单内）"
    }
  }
}
```

zombieKill：

```
{
  "type": "object",
  "required": ["page_id"],
  "properties": {
    "page_id": {
      "type": "string",
      "description": "要销毁的 zombie page ID"
    }
  }
}
```

5.3 错误码体系

所有错误响应使用统一的错误码格式：

```
{
  "seq": 5,
  "type": "response",
  "success": false,
  "error": {
    "code": "MAC_DOMAIN_MISMATCH",
    "message": "Expected domain erp.example.com but current page is
oa.example.com"
  }
}
```

错误码分类：

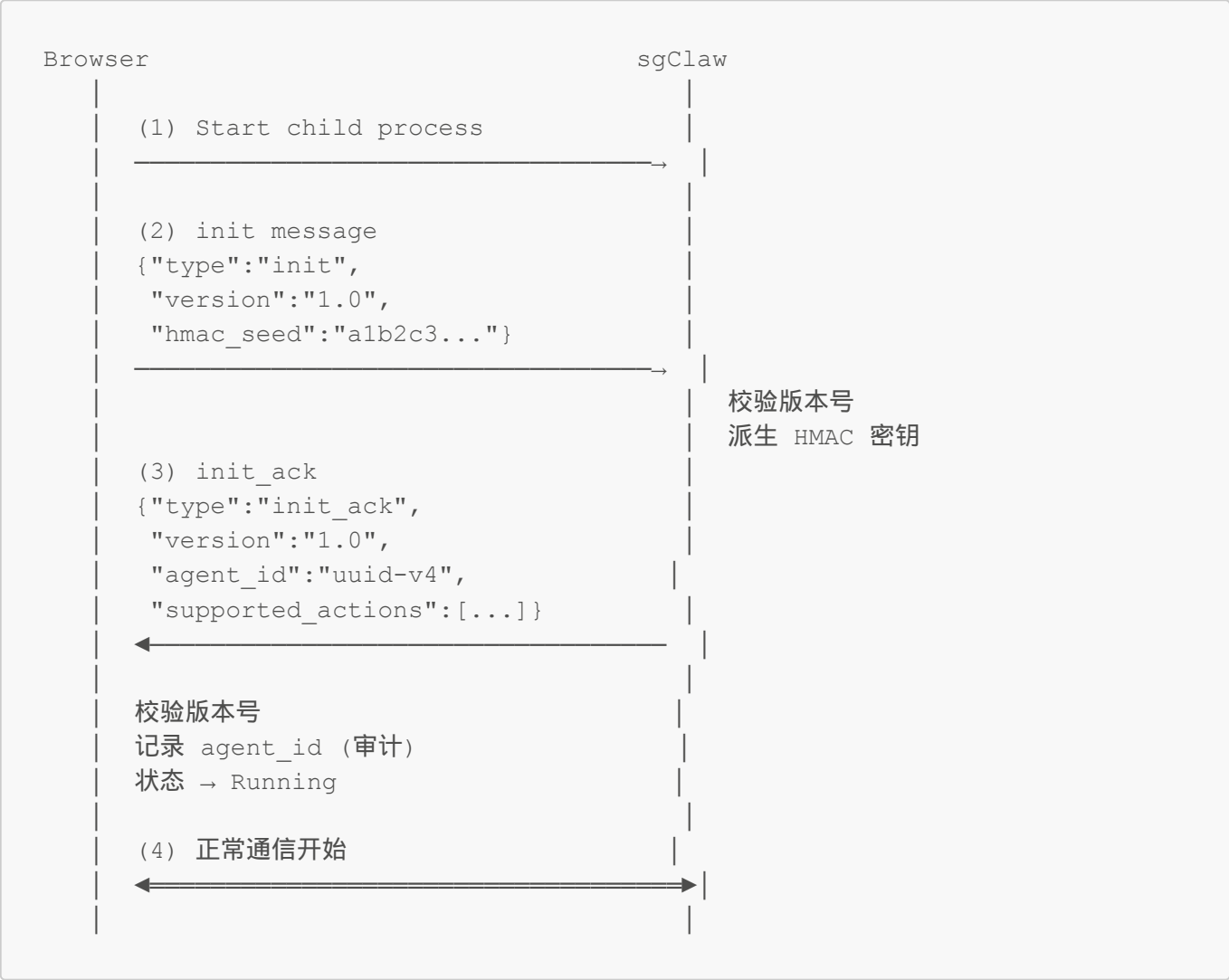
错误码前缀	来源	描述
`PIPE_*`	PipeListener	Pipe 传输层错误
`MAC_*`	MAC Whitelist Check	安全策略拒绝
`CMD_*`	CommandRouter	命令执行错误
`SESSION_*`	SessionManager	会话相关错误
`INTERNAL_*`	内部	系统内部错误

完整错误码列表：

错误码	描述	建议处理
`PIPE_INVALID_JSON`	JSON 解析失败	检查消息格式
`PIPE_MESSAGE_TOO_LARGE`	消息超过 1MB 限制	减小消息体积
`PIPE_SEQ_DUPLICATE`	序列号重复	检查 seq 生成逻辑
`PIPE_SEQ_OUT_OF_ORDER`	序列号乱序	检查消息发送顺序
`PIPE_HMAC_INVALID`	HMAC 签名校验失败	检查密钥和签名算法
`MAC_ACTION_BLOCKED`	Action 在黑名单中	使用允许的 action
`MAC_ACTION_NOT_ALLOWED`	Action 不在白名单中	检查 action 名称
`MAC_DOMAIN_NOT_ALLOWED`	域名不在白名单中	联系管理员添加域名
`MAC_DOMAIN_MISMATCH`	期望域名与实际不匹配	确认当前页面域名
`MAC_RATE_LIMIT`	超过速率限制	降低操作频率
`MAC_NEED_CONFIRM`	操作需要用户确认	等待用户确认
`CMD_SELECTOR_NOT_FOUND`	选择器未匹配到元素	检查选择器或等待元素
`CMD_SELECTOR_TIMEOUT`	等待元素超时	增加超时时间
`CMD_NAVIGATION_FAILED`	页面导航失败	检查 URL 可达性

错误码	描述	建议处理
CMD_ZOMBIE_POOL_FULL	Zombie 池已满 (max 5)	先销毁不需要的 zombie
CMD_ZOMBIE_NOT_FOUND	指定的 zombie page 不存在	检查 page_id
SESSION_EXPIRED	会话已过期	触发重新登录
SESSION_LOGIN_FAILED	登录失败	检查凭证或网络
INTERNAL_UNKNOWN	未知内部错误	查看日志

5.4 Handshake 协议流程



超时处理:

- Browser 发送 init 后等待 5 秒
- 超时未收到 init_ack → Kill 子进程 → 状态切换到 Crashed → 通知 UI

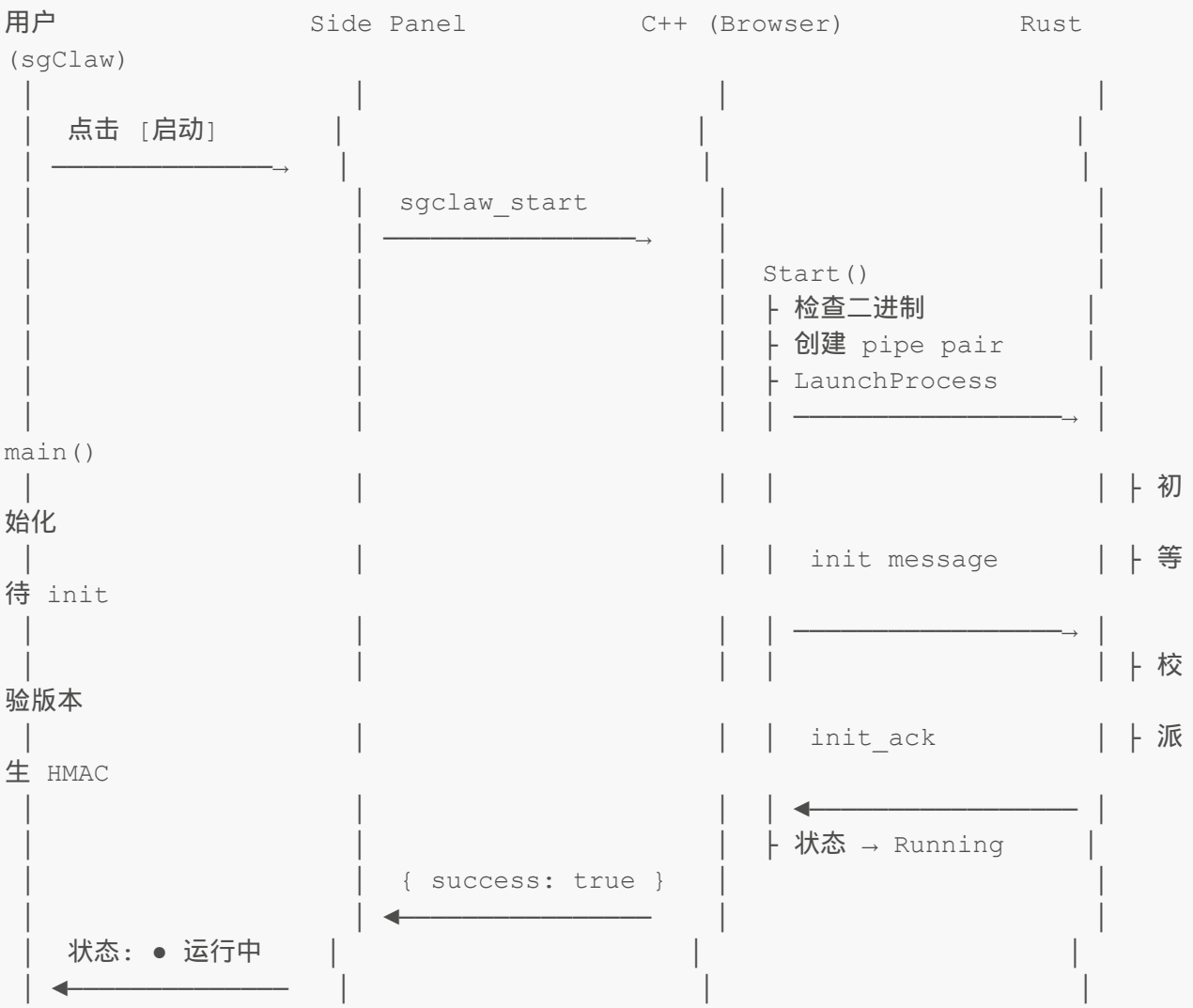
版本不匹配处理:

- sgClaw 收到的 version 与自身版本不一致 → 发送 error ack → 主动退出
- Browser 收到的 version 与自身不一致 → Kill 子进程 → 报错

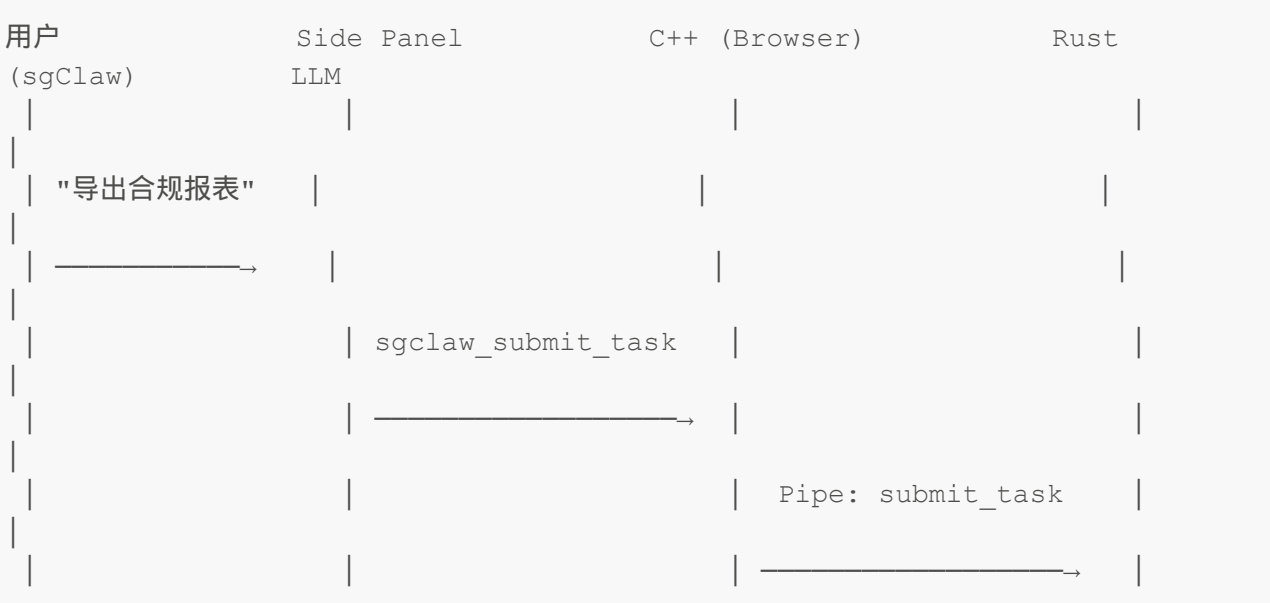
---

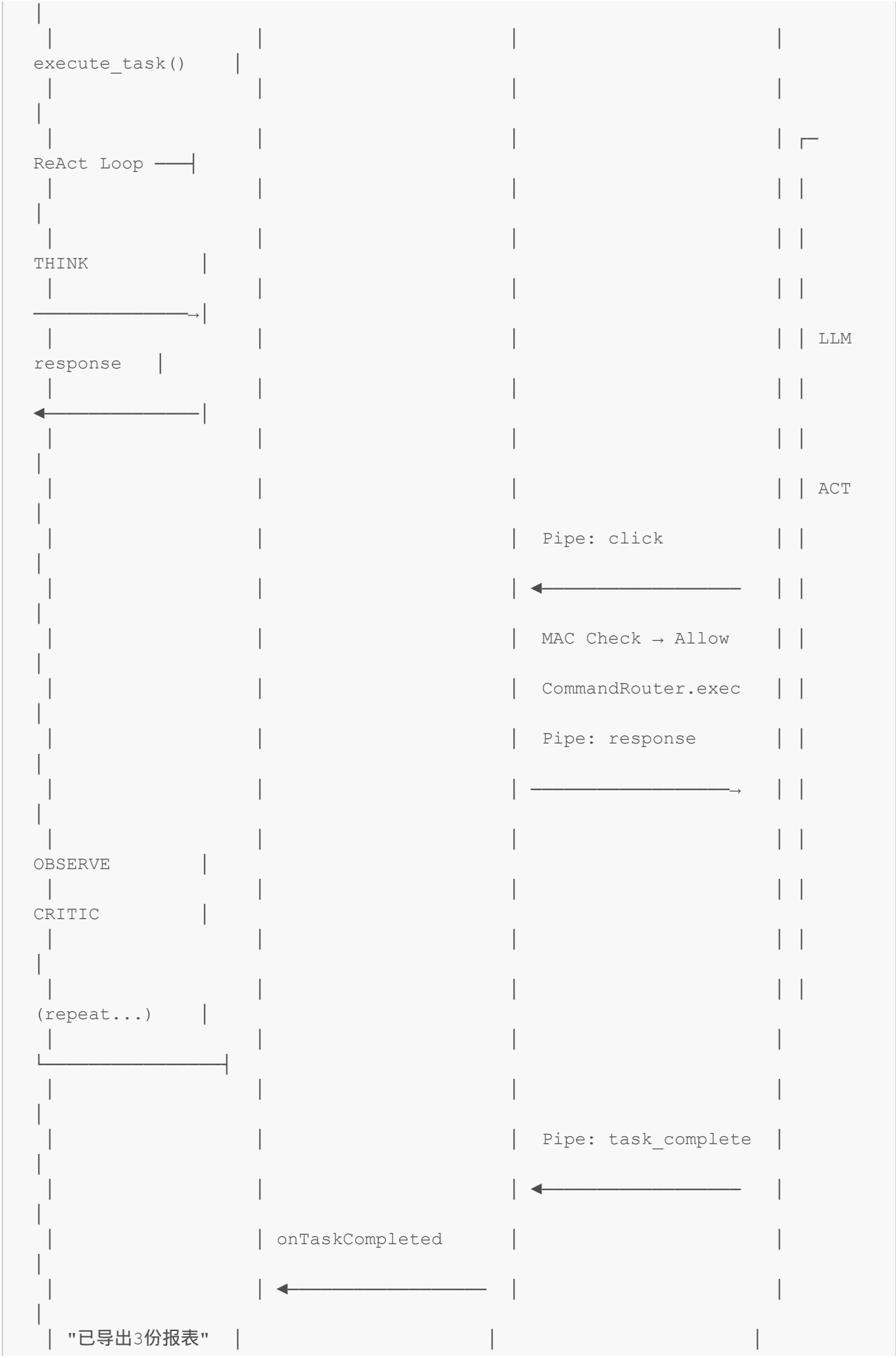
6. 跨模块交互时序

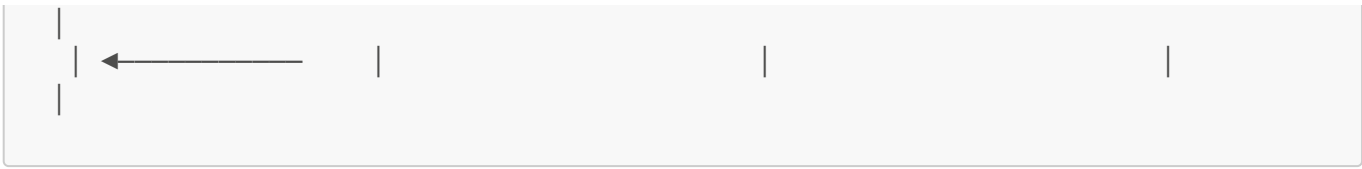
6.1 用户启动 Agent 完整时序



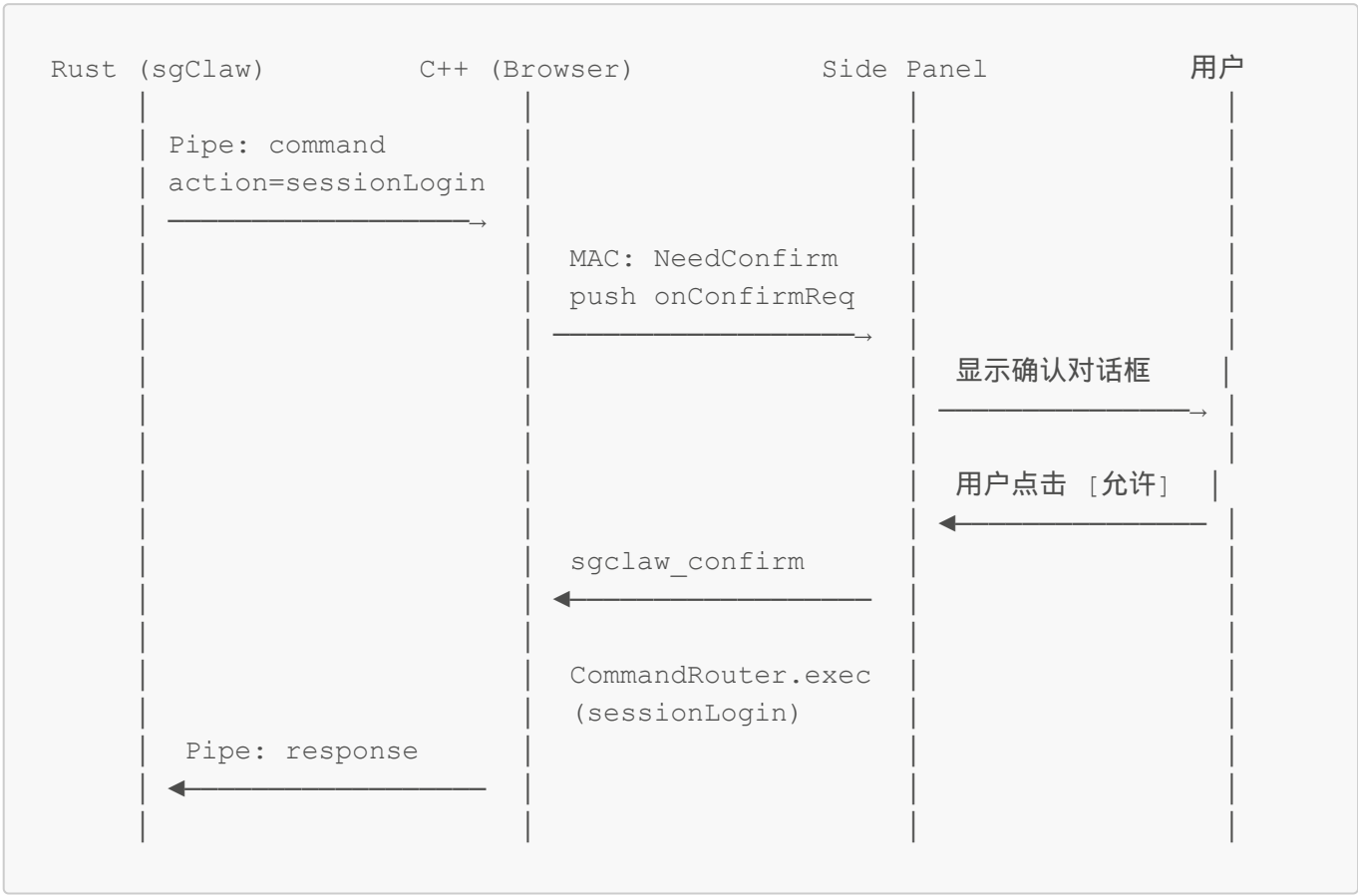
6.2 用户提交任务完整时序




execute_task()
ReAct Loop
THINK
response
OBSERVE
CRITIC
(repeat...)
LLM
ACT
Pipe: click
MAC Check → Allow
CommandRouter.exec
Pipe: response
Pipe: task_complete
onTaskCompleted
"已导出3份报表"



6.3 Human-in-the-loop 确认时序



---

7. 配置体系

7.1 sgclaw.toml 配置文件

```
# sgClaw 配置文件
# 路径: 与 sgclaw 二进制同目录, 或 $SGCLAW_CONFIG 指定

[general]
# Agent 日志级别: trace | debug | info | warn | error
log_level = "info"

[llm]
# LLM Provider: claude | openai | ollama
provider = "claude"
model = "claude-sonnet-4-20250514"
# API Key (建议通过环境变量 SGCLAW_LLM_API_KEY 设置)
# api_key = "sk-..."
# 自定义 API 端点 (可选)
# base_url = "https://api.example.com/v1"
```

```toml
[llm.config]
max_tokens = 4096
temperature = 0.1

[agent]
# 单次任务最大步数
max_steps = 50
# 系统提示模板路径（相对于 sgclaw 二进制）
system_prompt_template = "prompts/system.txt"

[memory]
# SQLite 数据库路径
db_path = "data/memory.db"
# 短期记忆最大条数
short_term_max_messages = 50
# 短期记忆最大 token 数
short_term_max_tokens = 8000

[security]
# rules.json 路径（域白名单 + action 白名单）
rules_path = "rules.json"
# Skill 公钥路径（Ed25519）
skill_public_key_path = "keys/skill_verify.pub"

[skills]
# 技能目录路径
skills_dir = "sgclaw-skills"

[circuit_breaker]
# 连续失败熔断阈值
failure_threshold = 10
# 基础冷却时间（秒）
cooldown_base_secs = 1
# 最大冷却时间（秒）
cooldown_max_secs = 30

[mcp]
# MCP Server 配置（可选）
# [[mcp.servers]]
# name = "filesystem"
# command = "npx"
# args = ["-y", "@anthropic/mcp-server-filesystem", "/tmp"]
```

7.2 环境变量覆盖

所有配置均可通过 `SGCLAW_` 前缀的环境变量覆盖：

环境变量	覆盖配置项	示例
`SGCLAW_LOG_LEVEL`	`general.log_level`	`debug`
`SGCLAW_LLM_PROVIDER`	`llm.provider`	`ollama`

环境变量	覆盖配置项	示例
SGCLAW_LLM_MODEL	llm.model	qwen2.5:32b
SGCLAW_LLM_API_KEY	llm.api_key	sk-...
SGCLAW_LLM_BASE_URL	llm.base_url	http://localhost:11434
SGCLAW_MAX_STEPS	agent.max_steps	100
SGCLAW_RULES_PATH	security.rules_path	/etc/sgclaw/rules.json

优先级：环境变量 > 配置文件 > 编译时默认值。

文档结束。本文档为 sgClaw L2 层模块设计与接口契约参考，各组应基于此文档中的接口定义 进行并行开发。接口变更需经三方组长会审后更新本文档。