Mini-OpenClaw

一个轻量级、全透明的 AI Agent 系统。强调文件驱动（Markdown/JSON 取代向量数据库）、指令式技能（而非 function-calling）、以及 Agent 全部操作过程的可视化。

目录

技术选型

层级	技术	说明
后端框架	FastAPI + Uvicorn	异步 HTTP + SSE 流式推送
Agent 引擎	LangChain 1.x `create_agent`	非 `AgentExecutor`，非遗留 `create_react_agent`
LLM	DeepSeek（langchain-deepseek）	通过 `ChatDeepSeek` 原生接入，兼容 OpenAI API 格式
RAG	LlamaIndex Core	向量检索 + BM25 混合搜索
Embedding	OpenAI `text-embedding-3-small`	通过 `OPENAI_BASE_URL` 可切换代理

层级	技术	说明
Token 计数	tiktoken `cl100k_base`	精确 token 统计
前端框架	Next.js 14 App Router	TypeScript + React 18
UI	Tailwind CSS + Shadcn/UI 风格	Apple 风 毛玻璃效果
代码编辑器	Monaco Editor	在线编辑 Memory/Skill 文件
状态管理	React Context	无 Redux，单一 `AppProvider`
存储	本地文件系统	无 MySQL/Redis，JSON + Markdown 文件

项目结构

```
mini-OpenClaw/
├── backend/
│   ├── app.py                      # FastAPI 入口，路由注册，启动初始化
│   ├── config.py                   # 全局配置管理（config.json 持久化）
│   ├── requirements.txt            # Python 依赖
│   ├── .env.example                # 环境变量模板
│   │
│   ├── api/                        # API 路由层
│   │   ├── chat.py                 #   POST /api/chat — SSE 流式对话
│   │   ├── sessions.py             #   会话 CRUD + 标题生成
│   │   ├── files.py                #   文件读写 + 技能列表
│   │   ├── tokens.py               #   Token 统计
│   │   ├── compress.py             #   对话压缩
│   │   └── config_api.py           #   RAG 模式开关
│   │
│   ├── graph/                      # Agent 核心逻辑
│   │   ├── agent.py                #   AgentManager — 构建 & 流式调用
│   │   ├── session_manager.py      #   会话持久化（JSON 文件）
│   │   ├── prompt_builder.py       #   System Prompt 组装器
│   │   └── memory_indexer.py       #   MEMORY.md 向量索引（RAG）
│   │
│   ├── tools/                      # 5 个核心工具
│   │   ├── __init__.py             #   工具注册工厂
│   │   ├── terminal_tool.py        #   沙箱终端
│   │   ├── python_repl_tool.py     #   Python 解释器
│   │   ├── fetch_url_tool.py       #   网页抓取（HTML→Markdown）
│   │   ├── read_file_tool.py       #   沙箱文件读取
│   │   ├── search_knowledge_tool.py #   知识库搜索
│   │   └── skills_scanner.py       #   技能目录扫描器
│   │
│   ├── workspace/                  # System Prompt 组件
│   │   ├── SOUL.md                 #   人格、语气、边界
│   │   ├── IDENTITY.md             #   名称、风格、Emoji
│   │   ├── USER.md                 #   用户画像
│   │   └── AGENTS.md               #   操作指南 & 记忆/技能协议
│   │
│   ├── skills/                     # 技能目录（每个技能一个子目录）
```

```
|   |   └── get_weather/SKILL.md      #   示例：天气查询技能
|   ├── memory/MEMORY.md              # 跨会话长期记忆
|   ├── knowledge/                    # 知识库文档（供 RAG 检索）
|   ├── sessions/                     # 会话 JSON 文件
|   |   └── archive/                  #   压缩归档
|   ├── storage/                      # LlamaIndex 持久化索引
|   |   └── memory_index/             #   MEMORY.md 专用索引
|   └── SKILLS_SNAPSHOT.md            # 技能快照（启动时自动生成）
|
└── frontend/
    └── src/
        ├── app/
        |   ├── layout.tsx            # Next.js 根布局
        |   ├── page.tsx              # 主页面（三栏布局）
        |   └── globals.css           # 全局样式
        ├── lib/
        |   ├── store.tsx             # React Context 状态管理
        |   └── api.ts                # 后端 API 客户端
        └── components/
            ├── chat/
            |   ├── ChatPanel.tsx     #   聊天面板（消息列表 + 输入框）
            |   ├── ChatMessage.tsx   #   消息气泡（Markdown 渲染）
            |   ├── ChatInput.tsx     #   输入框
            |   ├── ThoughtChain.tsx  #   工具调用思维链（可折叠）
            |   └── RetrievalCard.tsx #   RAG 检索结果卡片
            ├── layout/
            |   ├── Navbar.tsx        #   顶部导航栏
            |   ├── Sidebar.tsx       #   左侧边栏（会话列表 + Raw Messages）
            |   └── ResizeHandle.tsx  #   面板拖拽分隔条
            └── editor/
                └── InspectorPanel.tsx #  右侧检查器（Monaco 编辑器）
```

环境配置

复制 `.env.example` 为 `.env` 并填入 API Key：

```
cd backend
cp .env.example .env
```

```
# DeepSeek（Agent 主模型）
DEEPSEEK_API_KEY=sk-xxx
DEEPSEEK_BASE_URL=https://api.deepseek.com
DEEPSEEK_MODEL=deepseek-chat

# OpenAI（Embedding 模型，用于知识库 & RAG 检索）
OPENAI_API_KEY=sk-xxx
OPENAI_BASE_URL=https://api.openai.com/v1
EMBEDDING_MODEL=text-embedding-3-small
```

> `OPENAI_BASE_URL` 支持换成任意兼容 OpenAI Embedding 接口的代理地址。

启动方式

```
# 后端（端口 8002）
cd backend
pip install -r requirements.txt
uvicorn app:app --port 8002 --host 0.0.0.0 --reload

# 前端（端口 3000）
cd frontend
npm install
npm run dev
```

本机访问 `http://localhost:3000`，局域网内其他设备访问 `http://<本机IP>:3000`。

后端架构详解

应用入口 app.py

启动时通过 `lifespan` 执行三步初始化：

```
1. scan_skills()          → 扫描 skills/**/SKILL.md，生成 SKILLS_SNAPSHOT.md
2. agent_manager.initialize() → 创建 ChatDeepSeek LLM 实例，注册 5 个工具
3. memory_indexer.rebuild_index() → 构建 MEMORY.md 向量索引（供 RAG 使用）
```

随后注册 6 个 API 路由模块，所有路由统一挂载在 `/api` 前缀下。

Agent 引擎 graph/

agent.py — AgentManager

核心单例类，管理 Agent 的生命周期。

方法	职责
`initialize(base_dir)`	创建 `ChatDeepSeek` LLM、加载工具列表、初始化 SessionManager
`_build_agent()`	**每次调用都重建**，确保读取最新的 System Prompt 和 RAG 配置
`_build_messages()`	将会话历史（dict 列表）转换为 LangChain 的 `HumanMessage` / `AIMessage`
`astream(message, history)`	核心流式方法，依次 yield 6 种事件

`astream()` 的流式事件序列：

```
[RAG模式] retrieval → token... → tool_start → tool_end → new_response → token... → done
[普通模式]              token... → tool_start → tool_end → new_response → token... → done
```

关键机制：

- **多段响应**：Agent 每次执行完工具后再次生成文本时，会 yield 一个 `new_response` 事件，前端据此创建新的助手消息气泡
- **RAG 注入**：如果开启 RAG 模式，在调用 Agent 之前先检索 MEMORY.md，将结果作为临时上下文追加到 history 尾部（不持久化到会话文件）

session_manager.py — 会话持久化

以 JSON 文件管理每个会话的完整历史。

核心方法：

方法	说明
`load_session(id)`	返回原始消息数组
`load_session_for_agent(id)`	**为 LLM 优化**：合并连续的 assistant 消息、注入 `compressed_context`
`save_message(id, role, content, tool_calls)`	追加消息到 JSON 文件
`compress_history(id, summary, n)`	归档前 N 条消息到 `sessions/archive/`，摘要写入 `compressed_context`
`get_compressed_context(id)`	获取压缩摘要（多次压缩用 `---` 分隔）

`load_session_for_agent()` 与 `load_session()` 的区别：LLM 要求严格的 user/assistant 交替，而实际存储中可能有连续多条 assistant 消息（工具调用产生的多段响应），此方法将它们合并为单条。如果存在 `compressed_context`，还会在消息列表头部插入一条虚拟的 assistant 消息承载历史摘要。

prompt_builder.py — System Prompt 组装

按固定顺序拼接 6 个 Markdown 文件为完整的 System Prompt：

```
① SKILLS_SNAPSHOT.md      — 可用技能清单
② workspace/SOUL.md       — 人格、语气、边界
③ workspace/IDENTITY.md   — 名称、风格
④ workspace/USER.md       — 用户画像
⑤ workspace/AGENTS.md     — 操作指南 & 协议
⑥ memory/MEMORY.md        — 跨会话长期记忆（RAG 模式下跳过）
```

每个文件内容上限 **20,000 字符**，超出则截断并标记 `...[truncated]`。

RAG 模式下的变化：跳过 MEMORY.md，改为追加一段 RAG 引导语，告知 Agent 记忆将通过检索动态注入。

memory_indexer.py — MEMORY.md 向量索引

专门为 `memory/MEMORY.md` 构建的 LlamaIndex 向量索引，独立于知识库索引（存储路径 `storage/memory_index/`）。

方法	说明
`rebuild_index()`	读取 MEMORY.md → `SentenceSplitter(chunk_size=256, overlap=32)` 切片 → 构建 VectorStoreIndex → 持久化
`retrieve(query, top_k=3)`	语义检索，返回 `[{text, score, source}]`
`_maybe_rebuild()`	每次检索前通过 MD5 检查文件是否变更，变更则自动重建

另外，当用户通过 Monaco 编辑器保存 MEMORY.md 时，`files.py` 的 `save_file` 端点也会主动触发 `rebuild_index()`。

五大核心工具 tools/

所有工具均继承 LangChain 的 `BaseTool`，通过 `tools/__init__.py` 的 `get_all_tools(base_dir)` 统一注册。

工具	文件	功能	安全措施
`terminal`	`terminal_tool.py`	执行 Shell 命令	黑名单（`rm -rf /`、`mkfs`、`shutdown` 等）；CWD 限制在项目根目录；30s 超时；输出截断 5000 字符
`python_repl`	`python_repl_tool.py`	执行 Python 代码	封装 LangChain 原生 `PythonREPLTool`
`fetch_url`	`fetch_url_tool.py`	抓取网页内容	自动识别 JSON/HTML；HTML 通过 html2text 转 Markdown；15s 超时；输出截断 5000 字符
`read_file`	`read_file_tool.py`	读取项目内文件	路径遍历检查（不可逃逸出 `root_dir`）；输出截断 10,000 字符
`search_knowledge_base`	`search_knowledge_tool.py`	搜索知识库	惰性加载索引；从 `knowledge/` 目录构建；top-3 语义检索；索引持久化到 `storage/`

skills_scanner.py

非工具，而是启动时执行的扫描器：遍历 `skills/*/SKILL.md`，解析 YAML frontmatter（`name`、`description`），生成 XML 格式的 `SKILLS_SNAPSHOT.md`。该快照被纳入 System Prompt，让 Agent 知道有哪些可用技能。

API 层 api/

chat.py — 流式对话

`POST /api/chat` 是系统的核心端点。

请求体：

```
{"message": "你好", "session_id": "abc123", "stream": true}
```

内部流程：

1. 调用 `session_manager.load_session_for_agent()` 获取经过合并优化的历史
2. 判断是否为会话的第一条消息（用于后续自动生成标题）
3. 创建 `event_generator()`，内部调用 `agent_manager.astream()`
4. 按段（segment）追踪响应——每次工具执行后 Agent 重新生成文本时开启新段
5. `done` 事件到达后：保存用户消息 + 每段助手消息到会话文件
6. 如果是首条消息，额外调用 DeepSeek 生成 ≤10 字的中文标题

SSE 事件类型：

事件	数据	触发时机
`retrieval`	`{query, results}`	RAG 模式检索完成后
`token`	`{content}`	LLM 输出每个 token
`tool_start`	`{tool, input}`	Agent 调用工具前
`tool_end`	`{tool, output}`	工具返回结果后
`new_response`	`{}`	工具执行完毕、Agent 开始新一轮文本生成
`done`	`{content, session_id}`	整轮响应结束
`title`	`{session_id, title}`	首次对话后自动生成标题
`error`	`{error}`	发生异常

sessions.py — 会话管理

端点	方法	说明
`/api/sessions`	GET	列出所有会话（按更新时间倒序）
`/api/sessions`	POST	创建新会话（UUID 命名）

端点	方法	说明
`/api/sessions/{id}`	PUT	重命名会话
`/api/sessions/{id}`	DELETE	删除会话
`/api/sessions/{id}/messages`	GET	获取完整消息（含 System Prompt）
`/api/sessions/{id}/history`	GET	获取对话历史（不含 System Prompt，含 tool_calls）
`/api/sessions/{id}/generate-title`	POST	AI 生成标题

files.py — 文件操作

端点	方法	说明
`/api/files?path=...`	GET	读取文件内容
`/api/files`	POST	保存文件（编辑器用）
`/api/skills`	GET	列出可用技能

路径白名单机制：

- **允许的目录前缀**：`workspace/`、`memory/`、`skills/`、`knowledge/`
- **允许的根目录文件**：`SKILLS_SNAPSHOT.md`
- 包含路径遍历检测（`..` 攻击防护）

保存 `memory/MEMORY.md` 时会自动触发 `memory_indexer.rebuild_index()`。

tokens.py — Token 统计

端点	方法	说明
`/api/tokens/session/{id}`	GET	返回 `{system_tokens, message_tokens, total_tokens}`
`/api/tokens/files`	POST	批量统计文件 token 数，body: `{paths: [...]}`

使用 `tiktoken` 的 `cl100k_base` 编码器，与 GPT-4 系列一致。

compress.py — 对话压缩

端点	方法	说明
`/api/sessions/{id}/compress`	POST	压缩前 50% 历史消息

流程：

1. 检查消息数量 ≥ 4
2. 取前 50% 消息（最少 4 条）

3. 调用 DeepSeek（temperature=0.3）生成中文摘要（≤500 字）
4. 调用 `session_manager.compress_history()` 归档 + 写入摘要
5. 返回 `{archived_count, remaining_count}`

归档文件存储在 `sessions/archive/{session_id}_{timestamp}.json`。

config_api.py — 配置管理

端点	方法	说明
`/api/config/rag-mode`	GET	获取 RAG 模式状态
`/api/config/rag-mode`	PUT	切换 RAG 模式，body: `{enabled: bool}`

配置持久化到 `backend/config.json`。

System Prompt 组装

Agent 每次被调用时都会**重新读取**所有 Markdown 文件并组装 System Prompt，确保 workspace 文件的实时编辑能立即生效：

```
┌──────────────────────────────────────┐
| <!-- Skills Snapshot -->             |  ← SKILLS_SNAPSHOT.md
| <!-- Soul -->                        |  ← workspace/SOUL.md
| <!-- Identity -->                    |  ← workspace/IDENTITY.md
| <!-- User Profile -->                |  ← workspace/USER.md
| <!-- Agents Guide -->                |  ← workspace/AGENTS.md
| <!-- Long-term Memory -->            |  ← memory/MEMORY.md（RAG 模式下替换为引导语）
└──────────────────────────────────────┘
```

每个组件间以 `\n\n` 分隔，每个组件带 HTML 注释标签便于调试定位。

会话存储格式

文件路径：`sessions/{session_id}.json`

```
{
  "title": "讨论天气查询",
  "created_at": 1706000000.0,
  "updated_at": 1706000100.0,
  "compressed_context": "用户之前询问了北京天气...",
  "messages": [
    { "role": "user", "content": "北京天气怎么样？" },
    {
      "role": "assistant",
      "content": "让我查一下...",
      "tool_calls": [
        { "tool": "terminal", "input": "curl wttr.in/Beijing", "output": "..." }
      ]
    },
    { "role": "assistant", "content": "北京今天晴，气温 25°C。" }
```

```
  ]
}
```

说明：

- **v1 兼容**：如果文件内容是纯数组 `[...]`，`_read_file()` 会自动迁移为 v2 格式
- **多段 assistant**：一次工具调用后会产生多条连续的 assistant 消息
- **compressed_context**：可选字段，多次压缩用 `---` 分隔

Skills 技能系统

技能不是 Python 函数，而是**纯 Markdown 指令文件**。Agent 通过 `read_file` 工具读取 SKILL.md，理解步骤后用核心工具执行。

目录结构：

```
skills/
└── get_weather/
    └── SKILL.md
```

SKILL.md 格式：

```
---
name: 天气查询
description: 查询指定城市的天气信息
---

## 步骤

1. 使用 `fetch_url` 工具访问 wttr.in/{城市名}
2. 解析返回的天气数据
3. 以友好的格式回复用户
```

启动时 `skills_scanner.py` 扫描所有技能，生成 `SKILLS_SNAPSHOT.md` 供 Agent 参考。

前端架构概览

三栏 IDE 风格布局，基于 Flexbox + 可拖拽分隔条：

```
┌──────────────────────────────────────────────────────────┐
│  Navbar（mini OpenClaw / 赋范空间）                      │
├──────────┬──────────────────────────┬────────────────────┤
│ Sidebar  │       ChatPanel          │  InspectorPanel    │
│          │                          │                    │
│ 会话列表  │   消息气泡               │  Memory / Skills   │
│          │   ├─ ThoughtChain        │  文件列表          │
│ Raw Msgs │   ├─ RetrievalCard       │  Monaco 编辑器     │
│ 扳手/RAG  │   └─ Markdown 内容       │  Token 统计        │
│ Token 统计│                          │                    │
│          │   ChatInput              │                    │
├──────────┴──────────────────────────┴────────────────────┤
```

```
|  ResizeHandle (可拖拽)                                                    |
└──────────────────────────────────────────────────────────────────────────┘
```

状态管理：全部通过 `store.tsx` 的 React Context 管理，包括消息列表、会话切换、面板宽度、流式状态、压缩状态、RAG 模式等。

API 客户端（`api.ts`）：

- `streamChat()` 实现了自定义的 SSE 解析器（因为浏览器原生 `EventSource` 只支持 GET，而聊天接口是 POST）
- `API_BASE` 动态取 `window.location.hostname`，自动适配本机 / 局域网访问

核心数据流

用户发送消息

```
前端                                          后端
 |
 ├─ store.sendMessage(text)
 |    ├─ 创建 user + assistant 占位消息
 |    └─ streamChat(text, sessionId) ──────→ POST /api/chat
 |                                            |
 |                                            ├─ load_session_for_agent()
 |                                            |    ├─ 合并连续 assistant 消息
 |                                            |    └─ 注入 compressed_context
 |                                            |
 |                                            ├─ [RAG] memory_indexer.retrieve()
 |                                            |    └─ yield retrieval 事件
 |                                            |
 |                                            ├─ _build_agent()
 |                                            |    ├─ build_system_prompt()
 |                                            |    └─ create_agent(llm, tools,
prompt)
 |                                            |
 |    ← SSE: token ──────────────────────────├─ agent.astream()
 |    ← SSE: tool_start ─────────────────────|    ├─ yield token/tool_start/tool_end
 |    ← SSE: tool_end ───────────────────────|    └─ yield done
 |    ← SSE: new_response ───────────────────|
 |    ← SSE: token ──────────────────────────|
 |    ← SSE: done ───────────────────────────├─ save_message(user + assistant
segments)
 |                                            |
 |    ← SSE: title ──────────────────────────└─ [首次] _generate_title()
 |
 ├─ 实时更新 messages state
 └─ 刷新 sessions 列表
```

RAG 检索模式

```
用户开启 RAG ──→ PUT /api/config/rag-mode {enabled: true}
                  └─ config.json 写入 {"rag_mode": true}

用户发送消息 ──→ agent.astream()
                  |
                  ├─ get_rag_mode() → true
                  ├─ memory_indexer.retrieve(query)
                  |    ├─ _maybe_rebuild()  // MD5 检测变更
                  |    └─ index.as_retriever(top_k=3)
                  |
                  ├─ yield {"type": "retrieval", results: [...]}
                  ├─ 将检索结果拼接为 "[记忆检索结果]" 上下文
                  └─ 追加到 history 末尾（仅当次请求，不持久化）

前端收到 retrieval 事件 ──→ 存入 message.retrievals
                             └─ RetrievalCard 渲染紫色折叠卡片
```

对话压缩

```
用户点击扳手 ──→ 确认弹窗 ──→ POST /api/sessions/{id}/compress
                                |
                                ├─ 取前 50% 消息（≥4 条）
                                ├─ DeepSeek 生成中文摘要（≤500字）
                                ├─ 归档到 sessions/archive/
                                ├─ 从 session 中删除这些消息
                                └─ 摘要写入 compressed_context

下次调用 Agent ──→ load_session_for_agent()
                    └─ 在消息列表头部插入:
                       {"role": "assistant", "content": "[以下是之前对话的摘要]\n{摘
要}"}
```

关键设计决策

决策	理由
使用 `create_agent()` 而非 `AgentExecutor`	LangChain 1.x 推荐的现代 API，支持原生流式
每次请求重建 Agent	确保 System Prompt 反映 workspace 文件的实时编辑
文件驱动而非数据库	降低部署门槛，所有状态对开发者透明可查
技能 = Markdown 指令	Agent 自主阅读并执行，不需要注册新的 Python 函数
多段响应分别存储	忠实保留工具调用前后的文本段，Raw Messages 可完整审查

决策	理由
System Prompt 组件截断 20K	防止 MEMORY.md 膨胀导致上下文溢出
RAG 检索结果不持久化	避免会话文件膨胀，检索上下文仅用于当次请求
路径白名单 + 遍历检测	双重防护，终端和文件读取工具均受沙箱约束
`window.location.hostname` 动态 API 地址	一份代码同时支持本机和局域网访问

API 接口速查

路径	方法	说明
`/api/chat`	POST	SSE 流式对话
`/api/sessions`	GET	列出所有会话
`/api/sessions`	POST	创建新会话
`/api/sessions/{id}`	PUT	重命名会话
`/api/sessions/{id}`	DELETE	删除会话
`/api/sessions/{id}/messages`	GET	获取完整消息（含 System Prompt）
`/api/sessions/{id}/history`	GET	获取对话历史
`/api/sessions/{id}/generate-title`	POST	AI 生成标题
`/api/sessions/{id}/compress`	POST	压缩对话历史
`/api/files?path=...`	GET	读取文件
`/api/files`	POST	保存文件
`/api/skills`	GET	列出技能
`/api/tokens/session/{id}`	GET	会话 Token 统计
`/api/tokens/files`	POST	文件 Token 统计
`/api/config/rag-mode`	GET	获取 RAG 模式状态
`/api/config/rag-mode`	PUT	切换 RAG 模式