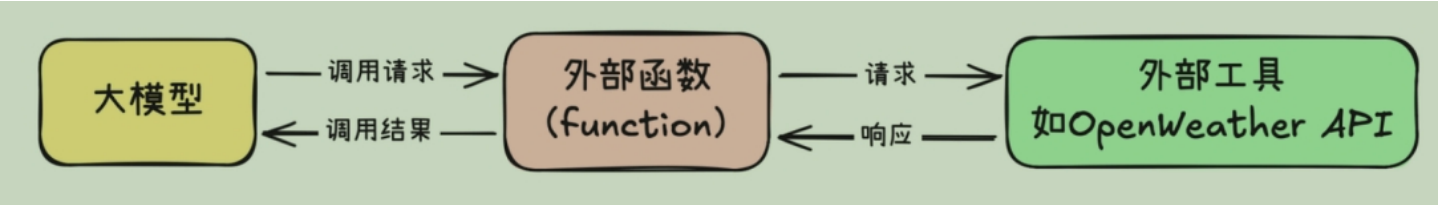
MCP

MCP

从 Function calling 到 MCP

MCP（Model Context Protocol），由Claude母公司Anthropic于24年11月正式提出。MCP是一种技术协议，是一种智能体Agent开发过程中共同约定的一种规范，如果大家都遵守一个规范，那么协作开发Agent的效率就能大幅提升。

MCP解决的最大痛点，**就是Agent开发中调用外部工具的技术门槛过高的问题。**由于大模型本身无法与外部工具直接通信，只能采用Function calling的方式（也叫tools），作为外部函数的中介：



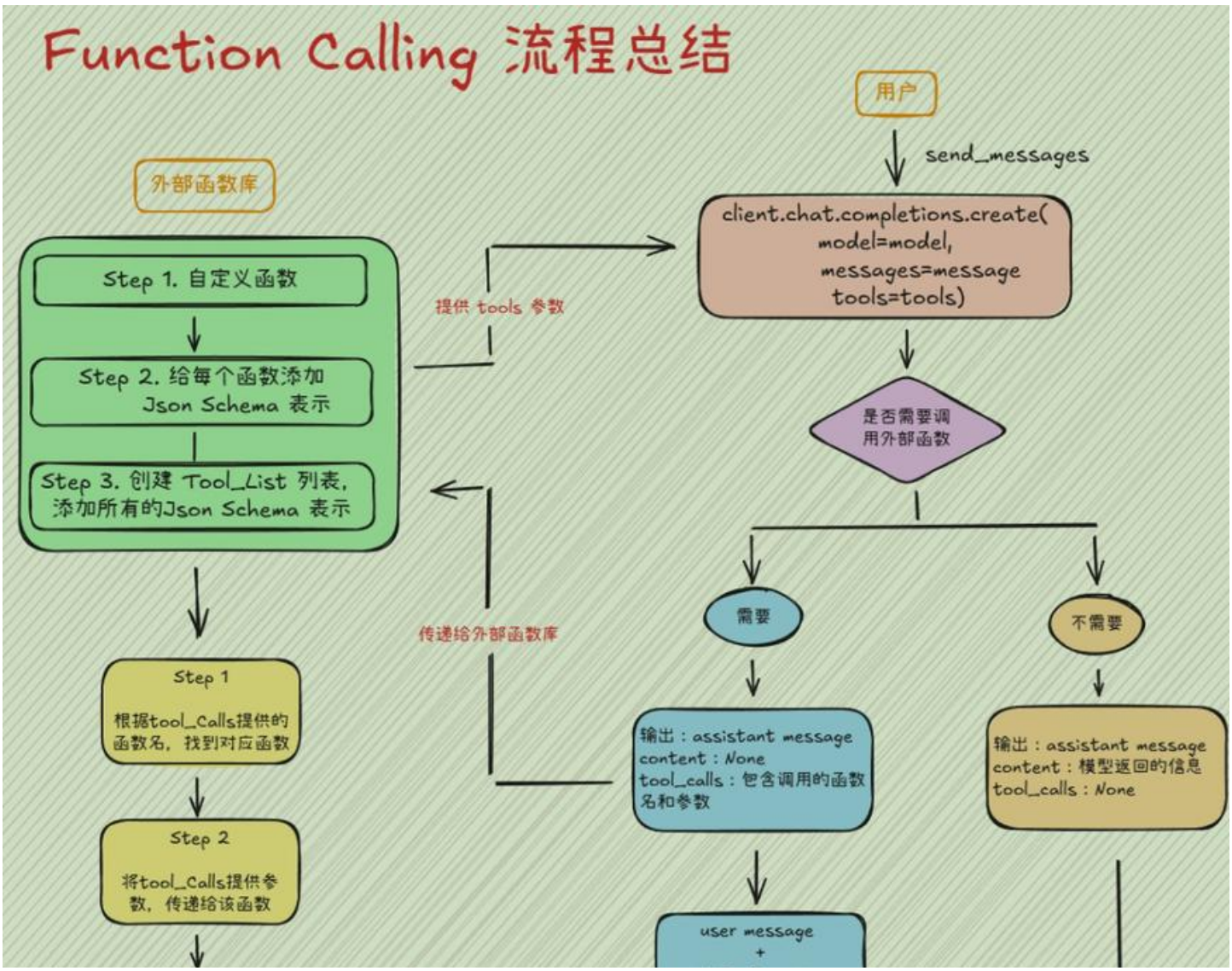
编写Function calling函数工作量很大（随便一个函数就要100+行代码），并且为了让大模型理解这个函数，需要用Json Schema格式编写功能说明，并设计提示词模板。

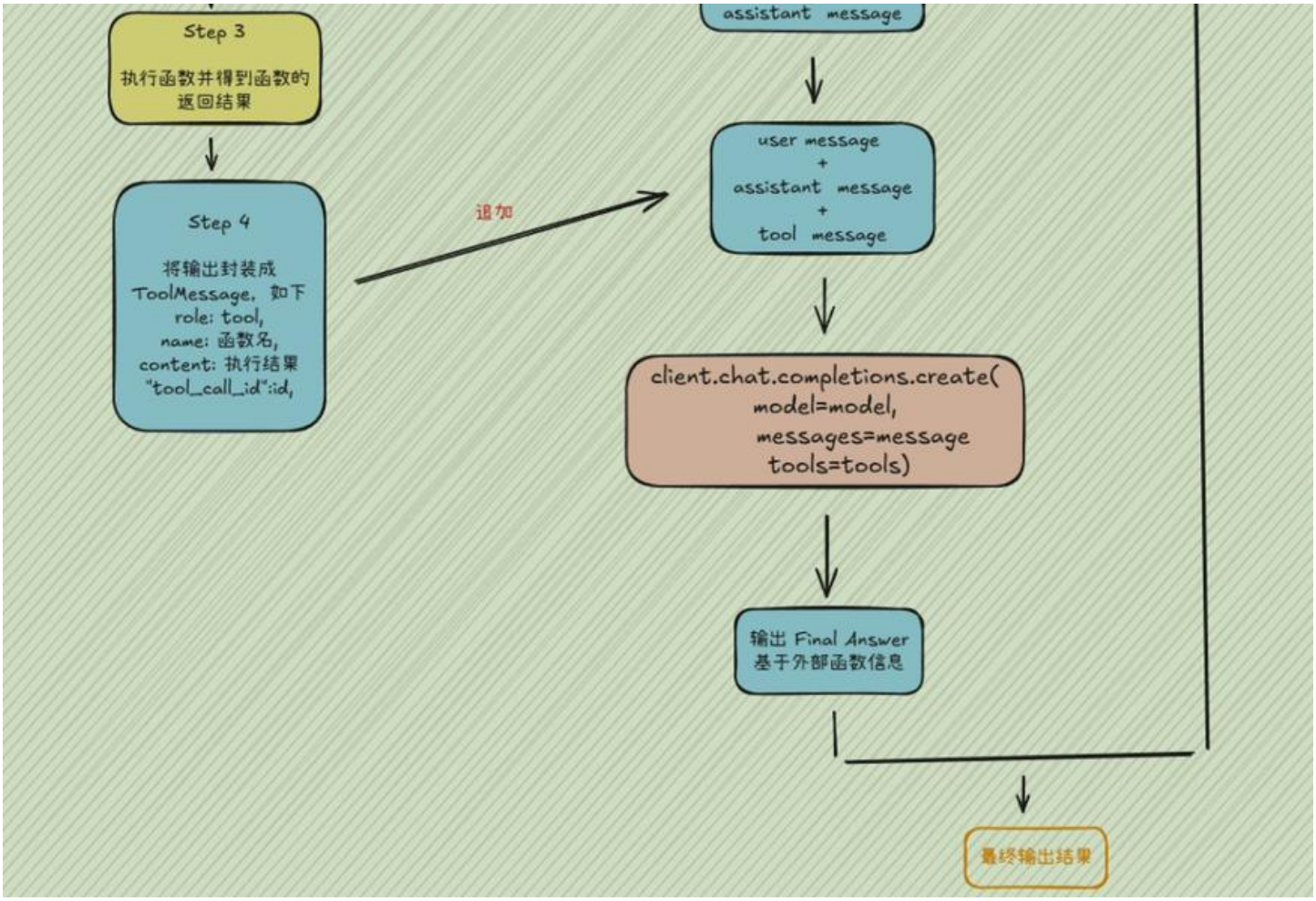
```
JSON Schema 是一种用于描述和验证JSON数据结构的标准化格式，在Function calling中扮演函数接口说明书
的角色。其核心作用是为大模型提供精准的函数调用规范，确保模型生成的参数格式正确。下面的例子中，指明了
Function名字和功能，以及入参类型、参数可选值、是否必须和参数描述等信息。
{
  "name": "get_weather",
  "description": "查询指定地点的天气信息",
  "parameters": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "城市名称，如'北京'"
      },
      "unit": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "温度单位"
      }
    },
```

```
      "required": ["location"]
    }
  }
```

提示词模板是预定义的**结构化指令框架**，用于引导大模型准确触发函数调用。其本质是通过工程化设计，
将自然语言指令转化为机器可解析的逻辑流。例如：

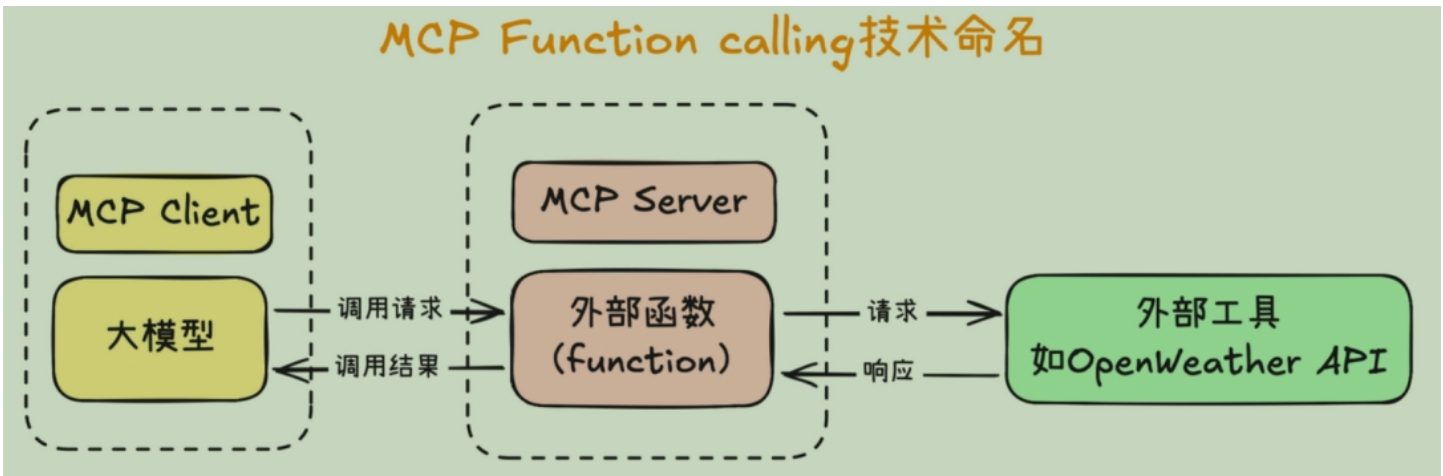
你是一个智能天气助手，请按以下步骤响应用户：
1. **意图识别**：判断用户是否在询问天气
2. **参数提取**：
   - 若需查询天气，提取地点(location)和单位(unit)
   - 若未明确单位，默认使用摄氏制
3. **函数调用**：严格按JSON格式返回调用指令：
   ```
   {
     "function": "get_weather",
     "arguments": {"location": "北京", "unit": "celsius"}
   }
   ```





MCP统一了Function calling的运行规范：

- 首先是先统一名称，MCP把大模型运行环境称作 **MCP Client**，也就是MCP客户端，同时，把外部函数运行环境称作**MCP Server**，也就是MCP服务器。
- 然后，统一MCP客户端和服务器的运行规范，并且要求MCP客户端和服务器之间，也统一按照某个既定的提示词模板进行通信。



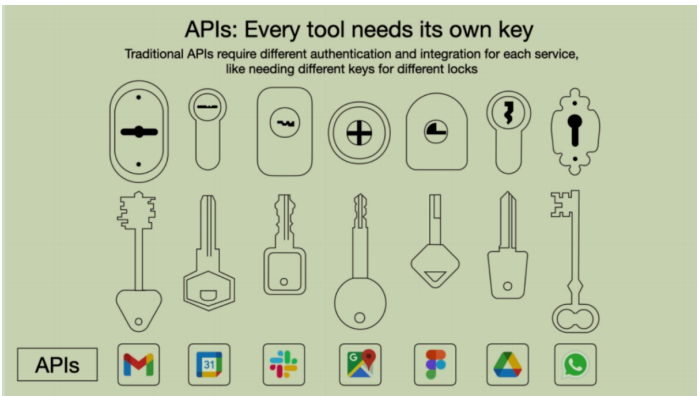
使用MCP的好处在于可以避免外部函数重复编写。 像查询天气、网页爬取、查询本地MySQL数据库这种通用的需求，只需要开发一个服务器就好，后续的开发者可以直接调用服务而不用重新实现。MCP开发工具支持Python、TS和Java等多种语言。想要使用MCP服务器就要构建MCP客户端（支持任意本

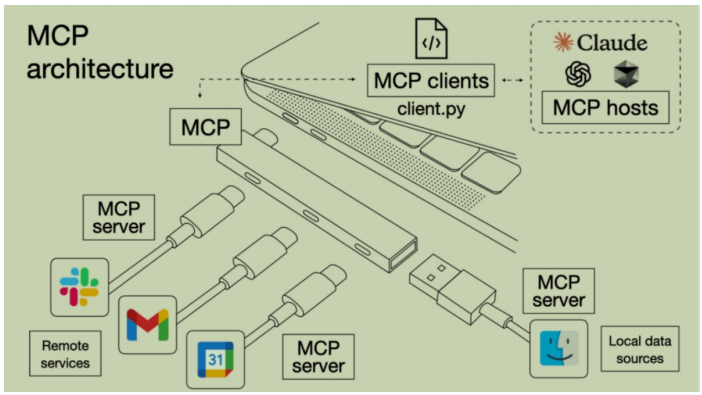
地和在线大模型，甚至是Cursor）。而如果没有所需要的MCP服务器，就要自己开发，下面的代码给出了一个简单的服务器示例

```python
# server.py
from mcp.server.fastmcp import FastMCP
# Create an MCP server
mcp = FastMCP("Demo")
# Add an addition tool
@mcp.tool()
def add(a:int,b:int)-> int:
    """Add two numbers"""
    return a+ b
# Add a dynamic greeting resource
@mcp.resource("greeting://{name}")
def get greeting(name:str)->str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"
```

> MCP针对agent的tools模块，目前不涉及memory和planning模块。

下图形象的对比了Function calling调用API和使用MCP的差异，MCP就像转接口，将多种多样的API封装成统一格式的mcp server，允许client端的大模型调用。





MCP客户端

uv环境管理

uv 是一个**Python 依赖管理工具**，类似于pip 和 conda ，但它更快、更高效，并且可以更好地管理 Python 虚拟环境和依赖项。它的核心目标是替代 pip 、 venv 和 pip-tools ，提供更好的性能和更低的管理开销。

uv **的特点**：

1. **速度更快**：相比 pip ， uv 采用 Rust 编写，性能更优。
2. **支持 PEP 582**：无需 virtualenv ，可以直接使用 pypackages 进行管理。
3. **兼容** pip ： 支持 requirements.txt 和 pyproject.toml 依赖管理。
4. **替代** venv ： 提供 uv venv 进行虚拟环境管理，比 venv 更轻量。
5. **跨平台**：支持 Windows、macOS 和 Linux。

```
# 安装uv
pip install uv
# 安装 Python 依赖，等效于pip install requests
uv pip install pandas
# 创建虚拟环境，等效于python -m venv myenv
uv venv myenv
# 激活虚拟环境
source myenv/bin/activate
# 安装所需的包，等效于pip install -r requirements.txt
uv pip install -r requirements.txt
# 运行 python 项目，等效于python script.py
uv run python script.py
```

MCP客户端搭建

```
# 创建目录
uv init mcp-client
cd mcp-client
# 创建虚拟环境并激活
uv venv
source .venv/bin/activate
# 安装 MCP SDK
uv add mcp
```

创建一个简单的MCP客户端，核心功能有：

- 初始化 MCP 客户端
- 提供一个命令行交互界面
- 模拟 MCP 服务器连接
- 支持用户输入查询并返回「模拟回复」
- 支持安全退出

```python
import asyncio # 让代码支持异步操作
from mcp import ClientSession # MCP 客户端会话管理
from contextlib import AsyncExitStack # 资源管理（确保客户端关闭时释放资源）

class MCPClient:
    def __init__(self):
        """初始化 MCP 客户端"""
        # 核心概念：会话，可以获取外部工具列表，保存当前会话状态等功能，暂时不链接MCP服务器
        self.session = None
        # 异步通信资源管理器
        self.exit_stack = AsyncExitStack()
    async def connect_to_mock_server(self):
        """模拟 MCP 服务器的连接（暂不连接真实服务器）"""
        print("✅ MCP 客户端已初始化，但未连接到服务器")
    async def chat_loop(self):
        """运行交互式聊天循环"""
        print("\nMCP 客户端已启动! 输入 'quit' 退出")
        while True:
            try:
                query = input("\nQuery: ").strip()
                if query.lower() == 'quit':
                    break
                print(f"\n🤖 [Mock Response] 你说的是: {query}")
            except Exception as e:
                print(f"\n⚠️ 发生错误: {str(e)}")
    async def cleanup(self):
        """清理资源"""
        await self.exit_stack.aclose() # 关闭资源管理器
async def main():
    client = MCPClient() # 创建 MCP 客户端
    try:
        await client.connect_to_mock_server() # 连接（模拟）服务器
        await client.chat_loop() # 进入聊天循环
    finally:
        await client.cleanup() # 确保退出时清理资源
if __name__ == "__main__":
    asyncio.run(main())
```

接入在线模型

```python
import asyncio
import os
from openai import OpenAI
```

```python
from dotenv import load_dotenv
from contextlib import AsyncExitStack
# 加载 .env 文件，确保 API Key 受到保护，需要在.env文件中写入：
    # BASE_URL="https://ai.devtool.tech/proxy/v1"
    # MODEL=gpt-4o
    # OPENAI_API_KEY="your_api_key"
load_dotenv()
class MCPClient:
    def __init__(self):
        """初始化 MCP 客户端"""
        self.exit_stack = AsyncExitStack()
        self.openai_api_key = os.getenv("OPENAI_API_KEY") # 读取 OpenAI API Key
        self.base_url = os.getenv("BASE_URL") # 读取 BASE YRL
        self.model = os.getenv("MODEL") # 读取 model
        if not self.openai_api_key:
            raise ValueError("❌ 未找到 OpenAI API Key，请在 .env 文件中设置 OPENAI_API_KEY")
        self.client = OpenAI(api_key=self.openai_api_key, base_url=self.base_url)
    async def process_query(self, query: str) -> str:
        """调用 OpenAI API 处理用户查询"""
        messages = [{"role": "system", "content": "你是一个智能助手，帮助用户回答问题。"},
        {"role": "user", "content": query}]
        try:
        # 调用 OpenAI API,将 OpenAI API 变成异步任务，防止程序卡顿。
            response = await asyncio.get_event_loop().run_in_executor(
                None,
                lambda: self.client.chat.completions.create(
                    model=self.model,
                    messages=messages
                )
            )
            return response.choices[0].message.content
        except Exception as e:
            return f"⚠️ 调用 OpenAI API 时出错: {str(e)}"
    async def chat_loop(self):
        """运行交互式聊天循环"""
        print("\n🤖 MCP 客户端已启动! 输入 'quit' 退出")
        while True:
            try:
                query = input("\n你: ").strip()
                if query.lower() == 'quit':
                    break
                response = await self.process_query(query) # 发送用户输入到 OpenAI API
```

```
                print(f"\n🤖 OpenAI: {response}")
            except Exception as e:
                print(f"\n⚠️ 发生错误: {str(e)}")
    async def cleanup(self):
        """清理资源"""
        await self.exit_stack.aclose()
async def main():
    client = MCPClient()
    try:
        await client.chat_loop()
    finally:
        await client.cleanup()
if __name__ == "__main__":
    asyncio.run(main())
```

部署本地模型

使用vllm库部署QwQ-32B

模型比较大，采用huggingface担心网络不稳定，可以用modelscope下载模型

代码块

```
pip install modelscope
modelscope download --model Qwen/QwQ-32B --local_dir ./QwQ-32B
```

安装vllm库

代码块

```
pip install vllm
```

开启vllm

代码块

```
vllm serve ./QwQ-32B --max-model-len 32768 # 32k上下文单卡
CUDA_VISIBLE_DEVICES=0,1 vllm serve ./QwQ-32B --tensor-parallel-size 2 # 128k上下文双卡
```

在jupyter中运行以下代码：

代码块

```
from openai import OpenAI
```

```
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)
prompt = "在单词\"strawberry\"中，总共有几个R？"
messages = [
    {"role": "user", "content": prompt}
]
response = client.chat.completions.create(
    model="QWQ-32B/",
    messages=messages,
)

print(response.choices[0].message.content)
```

后端接收到以下请求：

```
INFO:     127.0.0.1:39040 - "POST /v1/chat/completions HTTP/1.1" 200 OK
```

MCP服务器端

Server端可以提供以下三种标准能力：

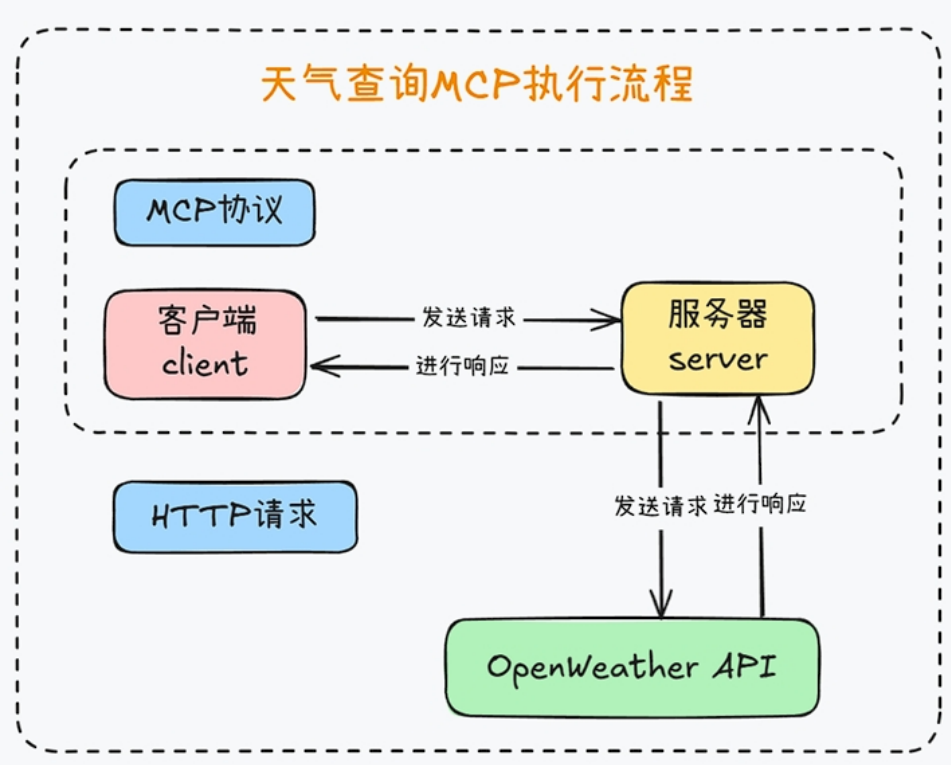
- **Resources**：资源，类似于文件数据读取，可以是文件资源或是API响应返回的内容。
- **Tools**：工具，第三方服务、功能函数，通过此可控制LLM可调用哪些函数。
- **Prompts**：提示词，为用户预先定义好的完成特定任务的模板。

通信机制

MCP目前支持两种传输方式：

- **标准输入输出（stdio）**：用于本地通信的传输方式。在这种模式下，MCP 客户端会将服务器程序作为子进程启动，双方通过标准输入（stdin）和标准输出（stdout）进行数据交换。这种方式适用于客户端和服务器在同一台机器上运行的场景，确保了高效、低延迟的通信。
- **HTTP+SSE**：适用于客户端和服务器位于不同物理位置的场景。在这种模式下，客户端和服务器通过 HTTP 协议进行通信，利用 SSE 实现服务器向客户端的实时数据推送。

天气查询服务器搭建



搭建了一个提供天气查询的工具的server，通过http请求查询天气。

代码块

```
uv add httpx
```

代码块

```python
import json
import httpx
from typing import Any
from mcp.server.fastmcp import FastMCP
import os
from dotenv import load_dotenv

# 加载环境变量
load_dotenv(" .env")

# 初始化 MCP 服务器
mcp = FastMCP("weatherServer")

# OpenWeather API 配置
OPENWEATHER_API_BASE = "https://api.openweathermap.org/data/2.5/weather"
API_KEY = os.getenv("OpenWeather_API_KEY")  # 请替换为你自己的 OpenWeather API Key
USER_AGENT = "weather-app/1.0"

# 异步获取天气数据
```

```python
async def fetch_weather(city: str) -> dict[str, Any] | None:
    """
    从 OpenWeather API 获取天气信息。
    :param city: 城市名称（需使用英文，如 Beijing）
    :return: 天气数据字典，若出错返回包含 error 信息的字典
    """
    params = {
        "q": city,
        "appid": API_KEY,
        "units": "metric",
        "lang": "zh_cn"
    }
    headers = {"User-Agent": USER_AGENT}
    # 使用 httpx.AsyncClient() 发送异步 GET 请求到 OpenWeather API。
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(OPENWEATHER_API_BASE, params=params, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()  # 返回字典类型
        except httpx.HTTPStatusError as e:
            return {"error": f"HTTP 错误: {e.response.status_code}"}
        except Exception as e:
            return {"error": f"请求失败: {str(e)}"}

# 天气数据格式化
def format_weather(data: dict[str, Any] | str) -> str:
    """
    将天气数据格式化为易读文本。
    :param data: 天气数据（可以是字典或 JSON 字符串）
    :return: 格式化后的天气信息字符串
    """
    # 如果传入的是字符串，则先转换为字典
    if isinstance(data, str):
        try:
            data = json.loads(data)
        except Exception as e:
            return f"无法解析天气数据: {e}"

    # 如果数据中含错误信息，直接返回错误提示
    if "error" in data:
        return f"{data['error']}"

    # 提取数据做容错处理
    city = data.get("name", "未知")
    country = data.get("sys", {}).get("country", "未知")
    temp = data.get("main", {}).get("temp", "N/A")
```

```
    humidity = data.get("main", {}).get("humidity", "N/A")
    wind_speed = data.get("wind", {}).get("speed", "N/A")

    # weather 是一个列表，因此此处用 [{}] 前先提供默认字典
    weather_list = data.get("weather", [{}])
    description = weather_list[0].get("description", "未知")

    return (
        f"🌍 {city}, {country}\n"
        f"🌡️ 温度: {temp}°C\n"
        f"💧 湿度: {humidity}%\n"
        f"🌬️ 风速: {wind_speed} m/s\n"
        f"☁️ 天气: {description}\n"
    )

@mcp.tool()
async def query_weather(city: str) -> str:
    """
    输入指定城市的英文名称，返回今日天气查询结果。
    :param city: 城市名称（需使用英文）
    :return: 格式化后的天气信息
```

接下来实现一个与调用这个server的client端，以与大模型对话的形式呈现，只有问天气查询的问题时调用工具，否则就是与大模型对话。

代码块

```
import asyncio
import os
import json
import sys
from typing import Optional
from contextlib import AsyncExitStack

from openai import OpenAI
from dotenv import load_dotenv

from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

# 加载环境变量
load_dotenv()

class MCPClient:
    def __init__(self):
        """初始化 MCP 客户端"""
        self.exit_stack = AsyncExitStack()# 统一管理异步上下文（如 MCP 连接）的生命周期。可以在退出（ cleanup ）时自动关闭
```

```
21          self.openai_api_key = os.getenv("OPENAI_API_KEY")
22          self.base_url = os.getenv("BASE_URL")
23          self.model = os.getenv("MODEL")
24
25          if not self.openai_api_key:
26              raise ValueError("  未找到 OpenAI API Key，请在 .env 文件中设置
    OPENAI_API_KEY")
27
28          self.client = OpenAI(api_key=self.openai_api_key,
    base_url=self.base_url)
29          self.session: Optional[ClientSession] = None # 用于保存 MCP 的客户端会话，
    默认是 None ，稍后通过 connect_to_server 进行连接
30
31      async def connect_to_server(self, server_script_path: str):
32          """连接到 MCP 服务器并列出可用工具"""
33          is_python = server_script_path.endswith('.py')
34          is_js = server_script_path.endswith('.js')
35
36          if not (is_python or is_js):
37              raise ValueError("服务器脚本必须是 .py 或 .js 文件")
38          # 判断服务器脚本是 Python 还是 Node.js，选择对应的运行命令。
39          command = "python" if is_python else "node"
40          server_params = StdioServerParameters(
41              command=command,
42              args=[server_script_path],
43              env=None
44          )
45
46          # 启动服务器连接
47          stdio_transport = await self.exit_stack.enter_async_context(
48              stdio_client(server_params)
49          )
50          self.stdio, self.write = stdio_transport
51          self.session = await self.exit_stack.enter_async_context(
52              ClientSession(self.stdio, self.write)
53          )# 发送初始化消息给服务器，等待服务器就绪。
54
55          await self.session.initialize()
56
57          # 列出可用工具
58          response = await self.session.list_tools()
59          print("\n已连接到服务器，支持以下工具:", [tool.name for tool in
    response.tools])
60
61      async def process_query(self, query: str) -> str:
62          """使用大模型处理查询并调用工具"""
63          messages = [{"role": "user", "content": query}]
```

```python
        response = await self.session.list_tools()

        # 构建可用工具列表
        available_tools = [
            {
                "type": "function",
                "function": {
                    "name": tool.name, # 工具的名字
                    "description": tool.description, # 外部函数的描述
                    "parameters": tool.inputSchema  # 如果要调用这个函数，需要的json_schema说明
                }
            } for tool in response.tools
        ]

        # 第一次模型调用
        response = self.client.chat.completions.create(
            model=self.model,
            messages=messages,
            tools=available_tools
```

代码块

```
uv run client.py server.py
```

📌 **MCPClient 的主要职责：**

- **启动 MCP 服务器**（通过 StdioServerParameters ）
- **建立 MCP 会话**，列出可用工具
- **处理用户输入**，将其发送给 OpenAI 模型
- **如果模型想调用 MCP 工具（Function Calling）**，就执行 call_tool
- **将结果重新发给模型**，并返回最终回答

测试服务器

Anthropic提供了一个非常便捷的debug工具：Inspector。借助Inspector，我们能够非常快捷的调用各类server，并测试其功能。

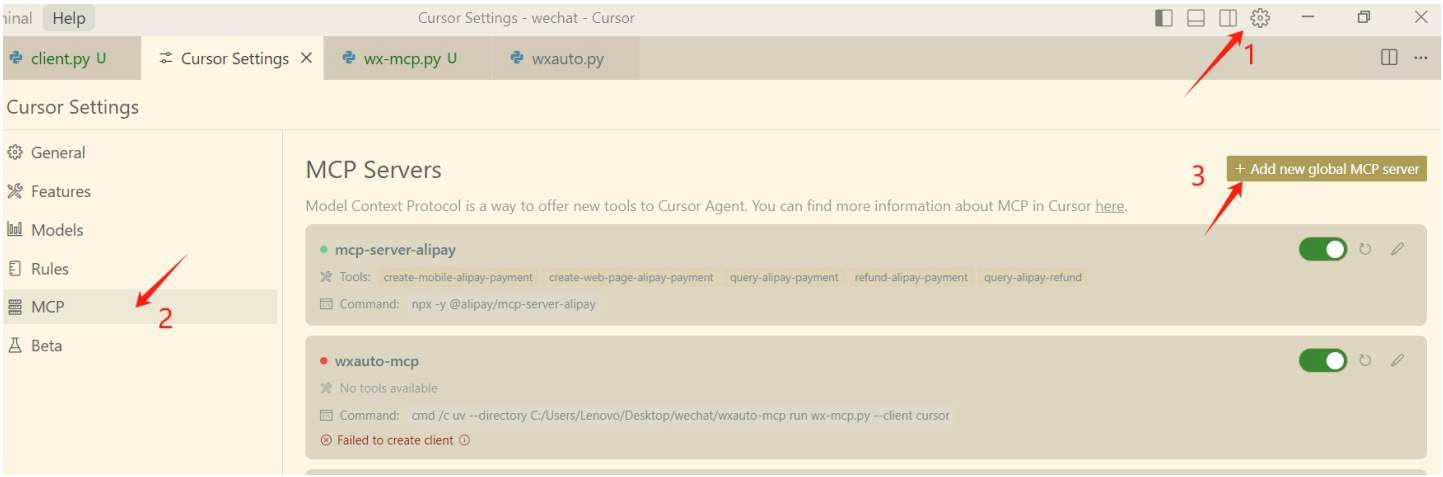
代码块

```
mcp dev xx.py
```

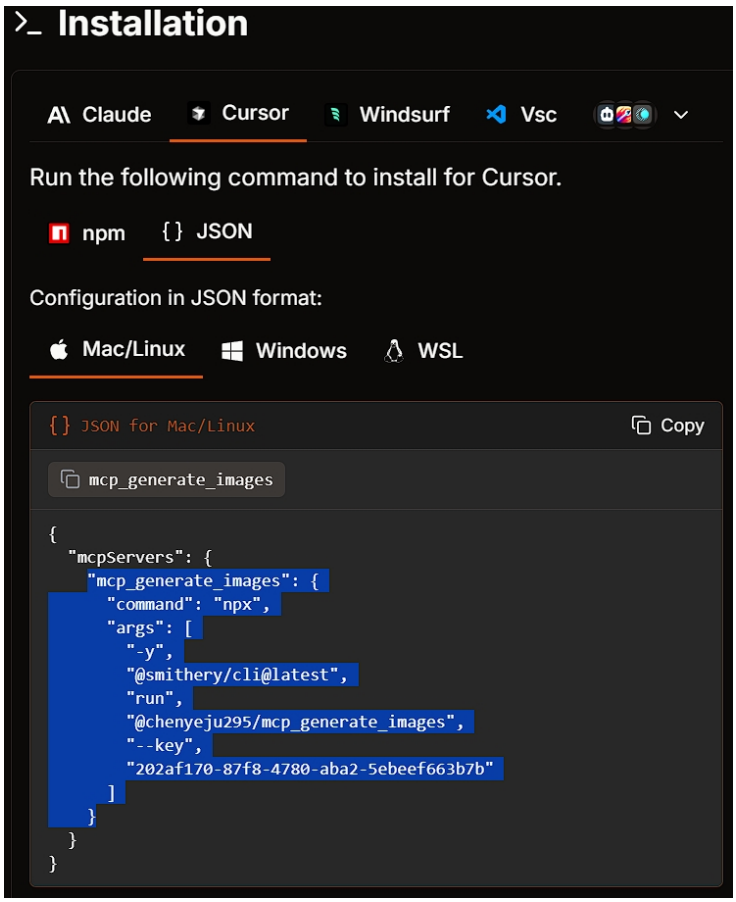
在线服务器导航：

- MCP官方服务器合集：https://github.com/modelcontextprotocol/servers
- MCP Github热门导航：https://github.com/punkpeye/awesome-mcp-servers
- https://mcp.so/
- https://mcpservers.cn/
- https://smithery.ai/

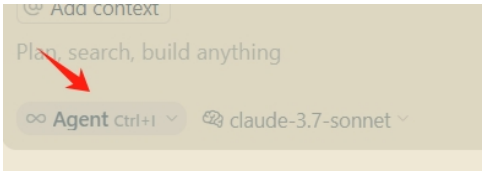
以cursor举例，将mcp server的功能添加到大模型之中。



将对应的json配置信息添加到mcp.json文件中，在上面页面刷新，变绿即添加成功。



调用大模型时要选择agent模式



- MCP是一种技术协议，是一种智能体Agent开发过程中共同约定的一种规范，如果大家都遵守一个规范，那么协作开发Agent的效率就能大幅提升。
- 首先是先统一名称，MCP把大模型运行环境称作 **MCP Client**，也就是MCP客户端，负责接收用户请求、调用MCP Server工具并整合结果。同时，把外部函数运行环境称作**MCP Server**，也就是MCP服务器，负责封装API、数据库等资源。
- MCP统一了客户端和服务器的运行规范，并且要求MCP客户端和服务器之间，也统一按照某个既定的提示词模板进行通信。
- 我实现的用MCP实现秒回功能
  - 视频：https://mp.weixin.qq.com/s/0EILiRUgHlNtPrCdDJm28g
  - 代码：https://github.com/saintGeorge13/wx-mcp/tree/main

代码链接：https://github.com/saintGeorge13/wx-mcp/tree/main

其他在线服务器导航：

- MCP官方服务器合集：https://github.com/modelcontextprotocol/servers
- MCP Github热门导航：https://github.com/punkpeye/awesome-mcp-servers
- https://mcp.so/
- https://mcpservers.cn/
- https://smithery.ai/

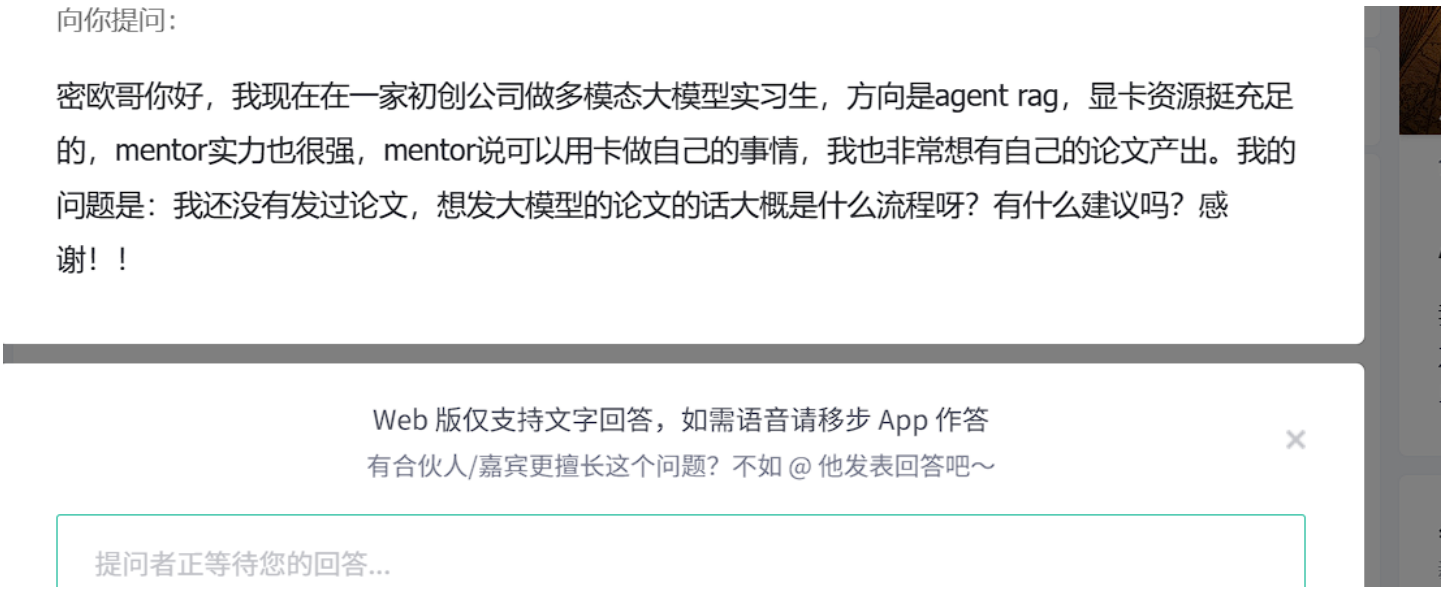
代码块

```
pip install uv
pip install mcp
pip install wxauto

# 运行inspector
```

```
mcp dev wx-mcp.py

# 本地mcp服务
uv run client.py wx-mcp.py
```



建议分为以下几步：

1. **确定研究方向**：首先明确自己的研究方向，agent 和 rag都是很大的概念，比如rag又有chunking、embedding、rerank等很多细分方向。如何确定研究方向？最好是来自你工作中的尝试，如果你在某个子方向做出了能提升效果的尝试，就可以在这个领域深入研究。
2. **阅读相关论文**：当你已经在某个方向做出一定尝试，接下来就要找这方面的论文看，一是为了避免你的idea已经被别人用了，二是对自己的方法做一定的完善，三是为了以后写related work。看论文一定要借助ai，让它快速总结一篇论文的核心思想，自己再判断要不要深入的看。一篇论文看完一定要记录笔记，不然一觉醒来就忘完了，重点总结这篇论文解决了什么问题？创新点是什么？论文中的方法能不能与自己的方法结合？
3. **运行实验**：如果你有比较熟悉的代码框架，比如llamaindex、langchain等，那可以直接在过去代码的基础上实现自己的方法。如果没有，可以在其他论文的源码的基础上改。实验数据集一定是公开数据集，方便跟其他方法对比（除非你的工作就是造数据集）。对照方法找近期的和经典的方法，最好有开源代码，跑出来的实验结果不如他们论文中的数据也没关系，一切以自己为主。
4. **写论文**：一定用latex格式写，论文模板找目标会议/期刊官方提供的模板。论文结构一般是introduction+related work+method+experiments+conclusion，论文长度参考往年的论文。写论文的核心是编好故事，这里也建议用ai辅助。

过程中有任何进展和卡点都及时与mentor沟通，论文大概率是要给他挂二作/通讯的，不问白不问。

投稿会议：中国计算机学会评级https://www.ccf.org.cn/ccftjgjxskwml/