智能体开发平台 AgentArts

高代码开发

文档版本 01

发布日期 2026-04-15

华为云计算技术有限公司

版权所有 © 华为云计算技术有限公司 2026。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 高代码开发概述 1

2 高代码开发流程 3

3 示例：构建你的首个高码智能体 6

4 组件库 31

4.1 沙箱工具 31
4.1.1 沙箱工具介绍 31
4.1.2 示例：基于控制台的工具开发与应用实践 33
4.1.3 示例：基于 SDK 的工具开发与应用实践 35
4.1.4 创建工具 37
4.1.5 在智能体中集成工具 40
4.1.6 查看代码解释器日志 42
4.2 记忆库 42
4.2.1 记忆库概述 42
4.2.2 创建记忆库 44
4.2.3 记忆检索 47
4.2.4 在智能体中集成记忆库 48
4.3 网关 49
4.3.1 网关介绍 50
4.3.2 创建网关 51
4.3.3 在网关中创建 Target 55
4.3.4 在智能体中使用网关 59
4.3.5 查看网关日志 60

5 部署智能体运行时 61

5.1 智能体运行时介绍 61
5.2 部署智能体运行时 62
5.3 管理智能体运行时 66
5.4 更新智能体运行时 67

6 观测智能体 68

7 AgentRuntime SDK 参考 69

7.1 文档导读 69
7.2 SDK 简介 70

7.3 SDK 应用框架 ... 70
7.4 CLI ... 73
7.5 快速开始 ... 80
7.6 Runtime SDK ... 83
7.7 Tools SDK ... 87
7.8 Memory SDK ... 89
7.9 Identity SDK ... 94
7.10 MCP Gateway SDK ... 96
8 常见问题 ... 113
8.1 认证鉴权 ... 113

1 高代码开发概述

AgentArts高代码开发为专业开发者提供了一条从代码到生产的完整链路——在熟悉的本地环境中使用主流框架开发，通过标准化SDK接入平台能力，以镜像形式一键部署至云端托管环境，并享受生产级的可观测与安全隔离保障。

核心价值：

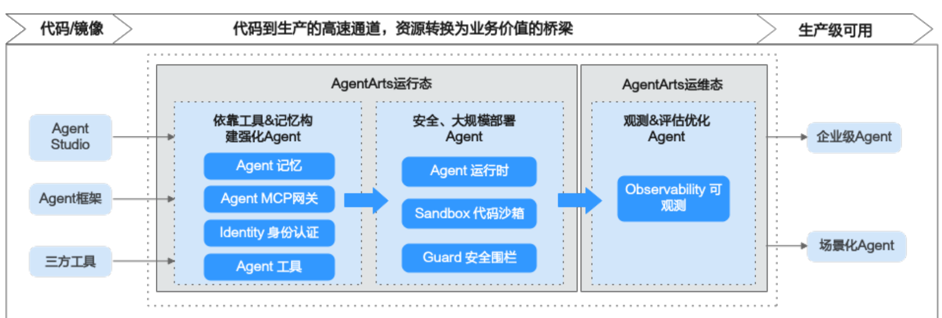
开发自由度高：使用LangChain/LangGraph等主流框架，代码完全自主可控

平台能力即插即用：记忆、MCP 网关、工具集成等组件开箱可用，无需从零搭建

生产级运行保障：安全隔离、弹性部署、灰度发布、全链路可观测一应俱全

两种开发模式：依托平台能力配置开发或直接迁移已有代码包，灵活适配不同场景

产品架构



AgentArts高代码开发的整体架构由开发态、运行态与运维态三部分构成：

- 本地代码开发：支持基于主流LangChain/LangGraph框架编写的代码、AgentArts平台在线编排（Agent Studio）产出的资产，以及第三方定制的工具，均可通过标准化的镜像形式无缝接入平台。
- 运行态：为智能体提供云端托管环境与业务能力扩展组件。
  - 托管环境：包含Agent运行时（Runtime）、代码沙箱（Sandbox）与安全围栏（Guard），支持智能体的弹性部署与安全隔离执行。
  - 扩展环境：包含Agent记忆（Memory）、MCP网关（Gateway）、工具集成（Tools）与身份认证（Identity）。开发者可开箱即用地实现上下文持久化、企业内部工具的安全调用与标准化鉴权。
- 运维态：提供智能体上线后的可观测能力，保障生产级可用性。

版权所有 © 华为云计算技术有限公司

- 可观测：。智能体运行时、网关、沙箱工具的日志，开发者可以实时监控智能体运行状态、排查故障。

产品功能

表 1-1 产品功能

功能	说明
记忆库	● 提供长短期记忆、分级存储及多种长期记忆抽取策略，支持自定义built-in策略，完善企业级记忆能力。 ● 通过高码SDK和API接入，支持九问、LangChain等平台的平滑接入。
工具	为Agent提供安全隔离的代码执行环境，支持运行代码、操作文件及执行系统命令，确保操作合规。
网关	提供API转换MCP能力，支持多种网关Target集成，灵活适配不同场景需求。
智能体运行时	● 构建企业级安全、高可靠、高性能的Agent原生运行环境，保障Agent稳定运行。 ● 支持LangChain/LangGraph Agents SDK框架开发的Agent一键部署，提供灰度发布能力，支持版本管理和多版本访问。 ● 企业级运行时可靠性及安全保障，包含99.95% SLA、身份认证、灵活网络配置及安全委托能力，确保权限合规。

2 高代码开发流程

在使用高代码开发智能体的过程中，可以根据具体需求选择合适的构建方式：“从零开始构建高码Agent”和“代码云上托管”。

适用场景

- 从零开始构建高码Agent

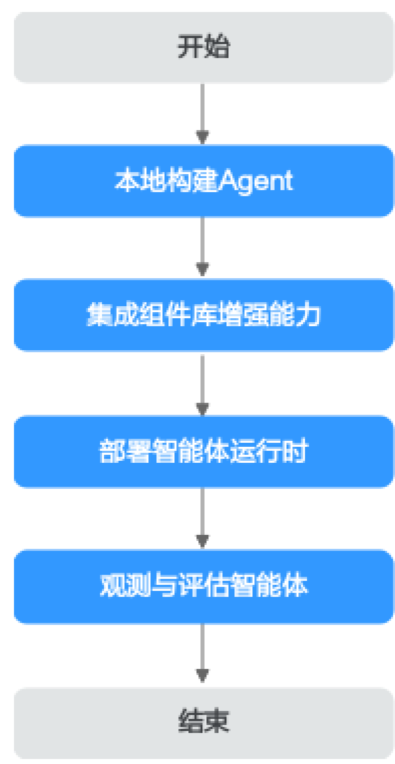
  强调灵活性和可扩展性，用户可以根据需求逐步增强Agent的能力，最终打包成镜像并部署到生产环境中。此流程适合对Agent有较高定制化需求的企业，能够充分释放开发者的创新空间。

- 代码云上托管

  简化了开发步骤，用户只需关注Agent的核心逻辑，打包成镜像后直接部署，利用云服务提供的观测和评估能力，确保Agent稳定运行。此流程适合对Agent功能有基础需求的企业，能够快速实现云化转型。

从零开始构建高码 Agent 开发流程

图 2-1 使用流程



1. 本地构建Agent：
   - 在本地开发环境中创建和开发Agent。
   - 使用AgentArts平台或LangChain/LangGraph Agents SDK框架进行开发。
2. 集成组件库增强能力：
   - **知识库**：集成知识库，提升Agent的知识处理能力。
   - **记忆**：实现Agent的记忆功能，使其能够存储和回忆历史数据。
   - **网关**：通过MCP网关，实现Agent与外部系统的通信和数据交换。
   - **沙箱工具**：使用沙箱工具进行安全隔离和测试，确保Agent的稳定性和安全性。
3. 部署智能体运行时：
   - 将开发好的Agent打包成Docker镜像。
   - 将镜像部署到目标运行环境中。
4. 观测与评估智能体：
   使用监控工具对Agent的运行状态进行实时监控。

代码云上托管开发流程

图 2-2 使用流程

1. **本地构建Agent**：
   - 在本地开发环境中创建和开发Agent。
   - 使用AgentArts平台或LangChain/LangGraph Agents SDK框架框架进行开发。
2. 部署智能体运行时：
   - 将开发好的Agent打包成Docker镜像。
   - 将镜像部署到云上运行环境。
3. 观测与评估智能体：
   使用云平台提供的监控工具对Agent的运行状态进行实时监控。

3 示例：构建你的首个高码智能体

本示例展示了一个基于AgentArts平台的完整Agent实现，充分利用平台提供的企业级能力来构建生产级别的AI Agent。

步骤1 本地搭建智能体。

1. 创建agent.py文件，编辑代码如下：

   示例代码具备基础的模型访问能力、本地记忆以及简单文件处理工具。

```
import os
import uuid
import json
import requests
import urllib3
from typing import List, Optional, Dict, Any
from pathlib import Path
from dotenv import load_dotenv
# 禁用 SSL 证书警告
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
# 
==============================================================================
# 基础依赖：LangGraph + LangChain
# 
==============================================================================
# LangGraph: AI Agent 工作流编排框架
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode
# 注：InMemorySaver 作为内存型记忆
from langgraph.checkpoint.memory import InMemorySaver

# 
==============================================================================
# 第一部分：配置模块
# 
==============================================================================
# 从 .env 文件加载环境变量
load_dotenv()
class ModelConfig:
    """
    模型配置类

    从环境变量读取模型配置：
    - MODEL_NAME: 模型名称（默认 gpt-4）
    - MODEL_URL: 模型API地址（默认 OpenAI 地址）
    - MODEL_API_KEY: 模型API密钥
    - MODEL_TYPE: 模型类型（默认 openai）
    """

    def __init__(self):
        self.name = os.getenv("MODEL_NAME", "gpt-4")
```


版权所有 © 华为云计算技术有限公司

```
        self.url = os.getenv("MODEL_URL", "https://api.openai.com/v1")
        self.api_key = os.getenv("MODEL_API_KEY")
        self.model_type = os.getenv("MODEL_TYPE", "openai")
class AgentConfig:
    """
    Agent全局配置类

    整合所有配置项:
    - model: 模型配置
    - max_iterations: Agent最大迭代次数（防止无限循环）
    - memory_space: AgentArts 记忆空间ID
    """

    def __init__(self):
        self.model = ModelConfig()
        self.max_iterations = int(os.getenv("MAX_ITERATIONS", "50"))
        self.memory_space = os.getenv("MEMORY_SPACE_ID", "default")
        self.region = os.getenv("HUAWEICLOUD_SDK_REGION", "cn-north-4")
# 全局配置实例（单例模式）
config = AgentConfig()

_global_checkpointer: Optional[InMemorySaver] = None


def get_checkpointer() -> InMemorySaver:

    global _global_checkpointer
    if _global_checkpointer is None:
        _global_checkpointer = InMemorySaver()
    return _global_checkpointer
def get_thread_config(session_id: str) -> Dict[str, Any]:
    """
    获取 LangGraph 运行时配置

    LangGraph 通过 config 中的 thread_id 来区分不同的会话

    Args:
        session_id: 会话ID（对应 LangGraph 的 thread_id）

    Returns:
        LangGraph 运行时配置字典
    """
    return {"configurable": {"thread_id": session_id}}
#
==============================================================================
# 第三部分：LLM 客户端
#
==============================================================================

class LLMClient:
    """
    LLM 客户端封装类

    封装 LangChain 的 ChatOpenAI，提供：
    - 模型调用能力
    - 工具绑定能力（让模型能够调用工具）
    """

    def __init__(self):
        self._client = self._create_client()
    def _create_client(self):
        """创建 OpenAI 兼容的 Chat 客户端"""
        model_cfg = config.model
        return ChatOpenAI(
            model=model_cfg.name,
            base_url=model_cfg.url,
            api_key=model_cfg.api_key,
            temperature=0,  # 设为0以获得更确定性的输出
            streaming=False,
```

```
        )
    def invoke(self, messages: List[BaseMessage]) -> BaseMessage:
        """调用 LLM 生成回复"""
        return self._client.invoke(messages)
    def bind_tools(self, tools: List[Any]):
        """绑定工具到 LLM，使模型能够调用这些工具"""
        return self._client.bind_tools(tools)
# 全局 LLM 客户端实例
llm_client = LLMClient()

#
==========================================================================
# 第四部分：工具定义
#
==========================================================================
# 工具说明：
# - 使用 @tool 装饰器将函数定义为 LangChain 工具
# - 工具函数必须有清晰的文档字符串，模型会根据描述决定是否调用
# - 工具返回值必须是字符串（会被返回给模型作为上下文）
@tool
def read_file(file_path: str) -> str:
    """
    读取文件内容

    Args:
        file_path: 文件路径（绝对路径或相对路径）

    Returns:
        文件内容（最大1MB）
    """
    try:
        path = Path(file_path)
        if not path.exists():
            return f"错误: 文件不存在 - {file_path}"
        if path.stat().st_size > 1024 * 1024:
            return f"错误: 文件过大 (最大支持1MB) - {file_path}"
        return path.read_text(encoding='utf-8')
    except Exception as e:
        return f"错误: 读取文件失败 - {str(e)}"


@tool
def write_file(file_path: str, content: str) -> str:
    """
    写入内容到文件（覆盖模式）

    Args:
        file_path: 文件路径
        content: 要写入的内容

    Returns:
        操作结果
    """
    try:
        path = Path(file_path)
        path.parent.mkdir(parents=True, exist_ok=True)
        path.write_text(content, encoding='utf-8')
        return f"成功写入文件: {file_path}"
    except Exception as e:
        return f"错误: 写入文件失败 - {str(e)}"


@tool
def append_file(file_path: str, content: str) -> str:
    """
    追加内容到文件末尾

    Args:
        file_path: 文件路径
```

```
        content: 要追加的内容

    Returns:
        操作结果
    """
    try:
        path = Path(file_path)
        path.parent.mkdir(parents=True, exist_ok=True)
        with open(path, 'a', encoding='utf-8') as f:
            f.write(content)
        return f"成功追加内容到文件: {file_path}"
    except Exception as e:
        return f"错误: 追加文件失败 - {str(e)}"


@tool
def list_directory(dir_path: str = ".") -> str:
    """
    列出目录内容

    Args:
        dir_path: 目录路径（默认当前目录）

    Returns:
        目录内容列表
    """
    try:
        path = Path(dir_path)
        if not path.exists():
            return f"错误: 目录不存在 - {dir_path}"
        if not path.is_dir():
            return f"错误: 不是目录 - {dir_path}"

        items = []
        for item in sorted(path.iterdir()):
            item_type = "DIR" if item.is_dir() else "FILE"
            size = item.stat().st_size if item.is_file() else 0
            items.append(f"{item_type:6} | {size:>10} | {item.name}")
        return "类型      | 大小       | 名称\n" + "\n".join(items)
    except Exception as e:
        return f"错误: 列出目录失败 - {str(e)}"

#
==========================================================================
# 第六部分：Agent 主类
#
==========================================================================
class AgentState(TypedDict):
    """Agent 状态类型定义"""
    messages: List
    iteration: int

class LangGraphAgent:
    """
    LangGraph Agent 主类

    对外提供的核心类，封装了：
    - 工作流图的创建和执行
    - 会话状态管理
    - 对话历史的存取

    使用示例:
        agent = LangGraphAgent()
        response = agent.run("你好", session_id="user-001")
        history = agent.get_history(session_id="user-001")
    """

    def __init__(self, system_prompt: Optional[str] = None, checkpointer=None):
        """
```


版权所有 © 华为云计算技术有限公司

```
        初始化 Agent

        Args:
            system_prompt: 系统提示词（可选，默认值见 _default_system_prompt）
            checkpointer: 自定义 Checkpointer（可选，默认使用全局单例）
        """
        self._checkpointer = checkpointer or get_checkpointer()
        self.graph = create_agent_graph(checkpointer=self._checkpointer)
        self.system_prompt = system_prompt or self._default_system_prompt()
        self._last_session_id: Optional[str] = None

    def _default_system_prompt(self) -> str:
        """默认系统提示词"""
        return """你是一个智能助手，可以通过调用工具来帮助用户完成任务。

可用的工具：
- read_file: 读取本地文件内容
- write_file: 写入本地文件
- append_file: 追加内容到本地文件
- list_directory: 列出本地目录内容
"""

    def _get_config(self, session_id: str) -> Dict[str, Any]:
        """获取运行时配置"""
        return get_thread_config(session_id)
    def run(self, user_input: str, session_id: Optional[str] = None) -> str:
        """
        运行 Agent 处理用户输入

        这是主要入口方法，每次调用都会：
        1. 从 Checkpoint 恢复会话状态（如有）
        2. 添加用户消息
        3. 执行工作流
        4. 自动保存状态到 Checkpoint

        Args:
            user_input: 用户输入的文本
            session_id: 会话ID（可选）
                - 不传：使用上次会话ID或自动生成新ID
                - 传值：使用指定会话ID，实现多会话隔离

        Returns:
            Agent 的回复文本
        """
        # 确定会话ID
        if session_id is None:
            session_id = self._last_session_id or str(uuid.uuid4())
        # 创建会话（如果已存在则忽略，继续执行）
        memory_client = MemoryClient()
        try:
            memory_client.create_memory_session(space_id=config.memory_space, id=session_id)
        except Exception as e:
            # Session already exists, continue with existing session
            if "Session id already exists" in str(e):
                pass
            else:
                raise

        self._last_session_id = session_id
        thread_config = self._get_config(session_id)
        # 从 checkpointer 获取历史消息
        checkpoint = self._checkpointer.get(thread_config)
        if checkpoint and "channel_values" in checkpoint:
            history_messages = list(checkpoint["channel_values"].get("messages", []))
        else:
            history_messages = []
        # 如果有历史消息，添加到现有消息后面（避免重复添加用户消息）
        if history_messages:
            # 检查最后一条是否是相同的用户消息
```

```
            if not (history_messages and isinstance(history_messages[-1], HumanMessage) and
                    history_messages[-1].content == user_input):
                messages = history_messages + [HumanMessage(content=user_input)]
            else:
                messages = history_messages
        else:
            # 首次对话，添加系统消息
            messages = [
                SystemMessage(content=self.system_prompt),
                HumanMessage(content=user_input)
            ]
        # 执行工作流
        result = self.graph.invoke({"messages": messages}, config=thread_config)
        # 返回最后一条消息的内容
        last_message = result["messages"][-1]
        return last_message.content if hasattr(last_message, "content") else str(last_message)
    @property
    def session_id(self) -> Optional[str]:
        """获取当前会话ID"""
        return self._last_session_id
```

2. 创建main.py文件简单测试agent响应。

```
""工具使用示例 - 展示如何让Agent使用工具"""
from agent import LangGraphAgent
def main():
    # 创建Agent
    agent = LangGraphAgent()
    print("=" * 50)
    print("示例1: 读取文件")
    print("=" * 50)
    # 让Agent读取当前目录
    response = agent.run("请列出当前目录的文件名，有哪几个文件")
    print(f"Agent: {response}")
    print("=" * 50)
    print("示例2: 查看历史记录")
    print("=" * 50)
    # 让Agent读取当前目录
    response = agent.run("请列上一个问题")
    print(f"Agent: {response}")

if __name__ == "__main__":
    main()
```

3. 创建.env文件配置如下模型相关环境变量

```
# 模型配置
MODEL_NAME=your_model_name
MODEL_URL=your_model_url
MODEL_API_KEY=your_model_api_key

# Agent配置
MAX_ITERATIONS=50
SESSION_TTL=3600
```

4. 运行命令python -m main.py执行。

步骤2 集成组件库增强能力。

1. 对接AgentArts runtime，使用SDK封装成http server。创建app.py文件，代码参考如下：

```
"""
================================================================================
AgentArts 平台部署入口 - 快速将 Agent 部署为 HTTP 服务
================================================================================

本文件展示了如何使用 AgentArts 平台的 @entrypoint 注解，
只需编写业务逻辑，即可快速将 Agent 部署为可调用的 HTTP 服务。

平台部署能力:
1. 零配置部署 - 只需编写业务逻辑，自动生成 HTTP 接口
2. 自动请求解析 - 平台自动将 JSON 请求转换为 payload
```

```
3. 内置会话管理 - 自动处理 session_id，支持多用户隔离
4. 全平台能力集成 - 自动继承记忆、身份认证、工具等平台能力
5. 水平扩展 - 支持多副本部署，自动负载均衡
6. 多协议支持 - HTTP REST + WebSocket 双通道
==============================================================================
"""
from agentarts.sdk import AgentArtsRuntimeApp, RequestContext
# 导入我们编写的 LangGraph Agent
from agent import LangGraphAgent
#
==============================================================================
# 第一步：创建平台应用实例
#
==============================================================================
# AgentArtsRuntimeApp: AgentArts 平台核心应用类
# - 自动启动 HTTP 服务
# - 自动处理请求路由
# - 自动集成平台能力（记忆、认证、工具等）
app = AgentArtsRuntimeApp()
# 创建 Agent 实例（所有请求共享此实例）
# 平台会自动处理并发和会话隔离
myagent = LangGraphAgent()
#
==============================================================================
# 第二步：定义入口函数（核心业务逻辑）
#
==============================================================================
# @app.entrypoint: 声明式入口点装饰器
#
# 工作原理:
# 1. 平台接收 HTTP 请求
# 2. 自动解析请求体为 payload 字典
# 3. 自动提取 session_id 用于会话隔离
# 4. 调用被装饰的函数，传入 payload 和 context
# 5. 函数的返回值自动序列化为 JSON 响应
#
# 参数说明:
# - payload: 请求体解析后的字典，包含用户传入的参数
# - context: 请求上下文，包含 session_id、用户信息等
#
# 平台自动处理:
#  HTTP 请求解析
#  JSON 序列化/反序列化
#  会话 ID 提取与管理
#  异常捕获与错误返回
#  请求日志与监控

@app.entrypoint
def my_agent(payload, context: RequestContext):
    """
    Agent 入口处理函数

    只需编写业务逻辑，平台负责其余一切:
    - 请求解析
    - 会话管理
    - 响应封装
    - 错误处理
    - 监控告警

    Args:
        payload: 请求参数字典
            - prompt: 用户输入（支持多种参数名）
            - message: 用户输入（备选参数名）
            - session_id: 会话ID（可选，平台自动管理）
            - 其他自定义参数...

        context: 请求上下文（平台注入）
            - context.session_id: 当前会话ID
            - context.request_id: 请求追踪ID
```

```
            - ...

    Returns:
        dict: 响应内容（自动序列化为 JSON）
            - response: Agent 回复内容
            - status: 执行状态 (success/error)
            - 其他自定义字段...
    """
    #
===========================================================================
    # 第三步：编写业务逻辑
    #
===========================================================================
    # 参数获取（平台已自动解析）
    prompt = payload.get("prompt", "")
    message = payload.get("message", "")
    # session_id 由平台自动管理，无需手动处理
    session_id = context.session_id
    # 统一用户输入（支持多种参数名）
    user_input = prompt or message
    # 调用 Agent 处理（平台能力自动生效）
    # - 会话记忆：自动恢复历史上下文
    # - 工具调用：自动注入访问令牌
    # - 代码执行：自动使用安全沙箱
    result = myagent.run(user_input, session_id=session_id)
    #
===========================================================================
    # 第四步：返回响应（平台自动封装）
    #
===========================================================================
    # 只需返回字典，平台自动处理：
    # - JSON 序列化
    # - HTTP 响应头
    # - 跨域处理
    # - 错误码映射
    return {
        "response": result,
        "status": "success"
    }
#
===========================================================================
# 第五步：启动服务
#
===========================================================================
# 本地开发模式
# 运行: python app.py
# 服务启动后，访问 http://localhost:8080 查看 API 文档
if __name__ == "__main__":
    # 平台自动启动 HTTP 服务器
    # 默认端口: 8080
    # 可通过环境变量配置: AGENT_RUN_PORT
    app.run(port=8080)
```

执行python app.py启动http serve，执行以下命令调用：

```
curl --location --request POST 'http://localhost:8080/invocations' \
--header 'Content-Type: application/json'\
--data-raw '{"message": "请列出当前目录的文件名，有哪几个文件"}'
```

2. 对接AgentArts memory组件，参考代码如下：

```
# - AgentArtsMemorySessionSaver: 企业级会话状态持久化
# - 替代标准 InMemorySaver，提供生产级能力
from agentarts.sdk.integration.langgraph import AgentArtsMemorySessionSaver

#
===========================================================================
# 第二部分：平台级记忆系统 (AgentArts Checkpoint)
#
===========================================================================
# AgentArts 平台提供的企业级会话记忆能力
#
```

```
# 与标准 LangGraph InMemorySaver 的区别:
# ┌────────────┬──────────────────────┬──────────────────────────────┐
# │ 特性        │ InMemorySaver (标准)    │ AgentArtsMemorySessionSaver │
# ├────────────┼──────────────────────┼──────────────────────────────┤
# │ 存储位置     │ 进程内存               │ 平台持久化存储                │
# │ 多实例共享   │ 不支持                 │ 支持（跨实例共享）             │
# │ 会话隔离     │ 基础隔离               │ 企业级租户隔离                │
# │ 状态恢复     │ 仅当前进程             │ 跨应用、跨会话恢复             │
# │ 数据安全     │ 重启丢失               │ 企业级数据保护                │
# │ 扩展性       │ 单机                   │ 分布式集群支持                │
# └────────────┴──────────────────────┴──────────────────────────────┘
#
# 核心能力:
# - space_id: 记忆空间标识，支持多租户隔离
# - region: 区域配置，支持跨区域部署
# - 自动状态持久化：每次 Agent 执行后自动保存状态
# - 快速状态恢复：从持久化存储中恢复会话上下文

_global_checkpointer: Optional[AgentArtsMemorySessionSaver] = None


def get_checkpointer() -> AgentArtsMemorySessionSaver:
    """
    获取平台级 Checkpointer 实例（单例模式）

    使用 AgentArtsMemorySessionSaver 替代标准的 InMemorySaver，
    实现企业级的会话状态持久化能力。

    平台优势:
        - 多实例共享：支持 Agent 部署多副本时共享会话状态
        - 租户隔离：通过 space_id 实现企业级数据隔离
        - 状态恢复：服务重启后自动恢复会话上下文
        - 零运维：无需自行搭建 Redis 等存储服务

    Returns:
        AgentArtsMemorySessionSaver: AgentArts 平台级记忆存储
    """
    global _global_checkpointer
    if _global_checkpointer is None:
        _global_checkpointer = AgentArtsMemorySessionSaver(space_id="your_space_id",
region="your_space_region")
    return _global_checkpointer
```

3. 使用云上内置代码解释器工具，参考代码如下：

```
@tool
def execute_python(code: str, description: str = "") -> str | None:
    """
    在平台沙箱环境中安全执行 Python 代码

    本工具使用 AgentArts 平台提供的 code_session 企业级代码执行能力，
    实现了完全隔离的安全沙箱环境，可以安全地执行 Agent 生成的代码。

    平台沙箱能力:
     ┌──────────────────────────────────────────────┐
     │ 安全隔离                                       │
     ├──────────────────────────────────────────────┤
     │ 完全隔离的执行环境，代码无法访问宿主机器资源          │
     │ 网络隔离：仅允许特定域名访问（可选配置）              │
     │ 文件系统隔离：仅能访问临时工作目录                   │
     │ 禁止危险操作：禁止 subprocess/threading/文件直接访问等 │
     ├──────────────────────────────────────────────┤
     │ 资源控制                                       │
     ├──────────────────────────────────────────────┤
     │ CPU 限制：防止无限循环占用资源                     │
     │ 内存限制：防止内存泄漏导致系统崩溃                  │
     │ 执行超时：自动终止长时间运行的代码                  │
     │ 磁盘配额：防止恶意写入大量数据                     │
```

```
│                                                      │
│ 企业级特性                                            │
├──────────────────────────────────────────────────────┤
│  执行日志：完整记录代码执行过程                          │
│  审计追溯：记录谁在什么时候执行了什么代码                  │
│  异常捕获：自动捕获并安全处理执行中的异常                  │
│  多区域支持：可选择不同区域的执行环境                      │
└──────────────────────────────────────────────────────┘

对比自建沙箱:
          自建方案                vs    AgentArts 平台方案
┌────────────────────────────┬──────────────────────────┐
│ 需要自行处理:                │ 平台全托管:                │
│  Docker 容器管理             │  开箱即用的容器化执行环境     │
│  资源限制配置                │  自动 CPU/内存/超时控制      │
│  安全漏洞修补                │  平台持续安全更新            │
│  执行监控告警                │  统一监控与告警              │
│  日志收集分析                │  结构化日志与审计            │
└────────────────────────────┴──────────────────────────┘

使用示例:
    # Agent 可以直接生成并执行代码
    result = execute_python.invoke("print([x**2 for x in range(10)])")
    # 返回: "[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]"

Args:
    code: 要执行的 Python 代码
    description: 代码的简要描述（可选，用于日志和调试）

Returns:
    执行结果的 JSON 字符串
"""
from agentarts.sdk.tools import code_session
if description:
    code = f"# {description}\n{code}"

print(f"\n Generated Code: {code}")
# 使用平台提供的 code_session 沙箱执行代码
with code_session("your_interpreter_region", "your_interpre_name") as code_client:
    response = code_client.invoke(
        operate_type="execute_code",
        arguments={
            "code": code,
            "language": "python",
            "clearContext": False  # 保持执行上下文，允许跨调用共享变量
        }
    )
    print(response)
return json.dumps(response["result"])
```

4. 使用云上identity获取出站认证凭据，参考如下代码示例：

```
@tool
@require_access_token(
    provider_name="your_provider_name",  # Agent Identity 平台注册的 GitHub 身份提供商
    scopes=["user:email"],               # 请求的 OAuth2 权限范围
    auth_flow="USER_FEDERATION",         # 用户联邦认证流程
ignore_ssl_verification=True,
)
def github_get(url: str, headers: Optional[Dict] = None, access_token: Optional[str] = None) -> str:
    """
    访问 GitHub API（平台级身份认证）

    本函数使用 AgentArts 平台的 require_access_token 装饰器实现自动身份认证，
    无需手动管理 GitHub 访问令牌。

    平台能力 vs 传统方式对比:
    ┌────────────────┬──────────────────┬──────────────────────────┐
    │ 特性            │ 传统方式          │ AgentArts + require_access_ │
    │                │                  │ token                    │
    ├────────────────┼──────────────────┼──────────────────────────┤
    │                │                  │                          │
```

```
│ 令牌管理        │ 手动创建、存储、轮换       │ 平台自动管理              │
│ 令牌获取        │ 用户自行配置 env 或配置    │ 装饰器自动注入            │
│ 令牌刷新        │ 需要编写刷新逻辑           │ 平台自动处理              │
│ 安全性          │ 令牌暴露在代码或环境变量   │ 令牌存储在平台安全存储    │
│ 用户体验        │ 配置复杂，容易出错         │ 声明式配置，开箱即用      │
└─────────────────┴────────────────────────────┴───────────────────────────┘

认证流程说明:
    1. 用户首次调用时，平台自动发起 OAuth2 授权流程
    2. 用户在 GitHub 授权页面完成授权
    3. 平台获取并安全存储访问令牌
    4. 后续调用时，access_token 自动注入到函数参数
    5. 令牌过期前，平台自动刷新

Args:
    url: GitHub API 路径（如 /repos/owner/repo/issues）
    headers: 自定义请求头
    access_token: [平台自动注入] OAuth2 访问令牌

Returns:
    API 响应内容
"""
# 记录 token 状态
has_token = bool(access_token)
print(f"[GITHUB] access_token: {'已注入' if has_token else '未注入'}")
base_url = "https://api.github.com"

default_headers = {
    "User-Agent": "LangGraph-Agent",
    "Accept": "application/vnd.github.v3+json",
}
if access_token:
    default_headers["Authorization"] = f"token {access_token}"
if headers:
    default_headers.update(headers)
# 获取代理配置
proxies = {}
http_proxy = os.environ.get("HTTP_PROXY") or os.environ.get("http_proxy")
https_proxy = os.environ.get("HTTPS_PROXY") or os.environ.get("https_proxy")
if http_proxy:
    proxies["http"] = http_proxy
if https_proxy:
    proxies["https"] = https_proxy
try:
    full_url = base_url + url if url.startswith("/") else url
    # 禁用 SSL 证书验证（仅用于测试）
    response = requests.get(full_url, headers=default_headers, timeout=30, proxies=proxies if proxies
else None, verify=False)
    # 记录响应状态
    print(f"[GITHUB] 响应状态: {response.status_code}")
    if len(response.content) > 1024 * 1024:
        return f"响应内容过大，已截断\n\n{response.text[:10000]}"

    return f"状态码: {response.status_code}\n\n{response.text}"
except Exception as e:
    print(f"[GITHUB] 请求失败: {str(e)}")
    return f"错误: GitHub API请求失败 - {str(e)}"
```

步骤3 部署智能体运行时。

按上述步骤完成后，基本代码开发已经完成，接下来准备部署到平台。

首先准备依赖文件requirements.txt内容可参考如下：

```
# ==============================================================================
# AgentArts LangGraph Agent Demo - 依赖清单
# ==============================================================================
#
# 本项目基于 AgentArts 平台，使用 LangGraph 框架构建 AI Agent
# 依赖分为两部分：基础框架依赖 + 平台 SDK 依赖
```

```
#
# 安装方式:
#   pip install -r requirements.txt
#
# ==============================================================================


# ==============================================================================
# 第一部分：LangGraph & LangChain 核心框架
# ==============================================================================
# LangGraph: AI Agent 工作流编排框架
# - 状态图定义与执行
# - Checkpoint 持久化机制
# - 条件边与节点路由
langgraph>=0.2.0


# LangChain: LLM 应用开发工具链
# - 消息类型定义 (HumanMessage, AIMessage, SystemMessage)
# - 工具系统 (@tool 装饰器)
# - LLM 客户端封装
langchain>=0.3.0
langchain-core>=0.3.0
langchain-community>=0.3.0


# ==============================================================================
# 第二部分：LLM Provider 支持
# ==============================================================================
# OpenAI 兼容接口
openai>=1.0.0
langchain-openai>=0.1.0


# Anthropic (Claude) 支持
anthropic>=0.18.0
langchain-anthropic>=0.1.0


# ==============================================================================
# 第三部分：HTTP & 网络
# ==============================================================================
requests>=2.31.0
httpx>=0.27.0


# ==============================================================================
# 第四部分：工具与配置
# ==============================================================================
# 环境变量管理
python-dotenv>=1.0.0


# 异步支持
aiofiles>=23.0.0


# JSON/YAML 支持
pyyaml>=6.0


# ==============================================================================
# 第五部分：AgentArts 平台 SDK（核心依赖） todo 待定
# ==============================================================================
#
#
# ==============================================================================
```

执行命令配置智能体

 版权所有 © 华为云计算技术有限公司

agentarts configure --entrypoint app:app

执行命令部署智能体

agentarts launch

----结束

完整agent.py代码样例可参考如下：

```
核心平台能力:
1. 记忆系统 (AgentArtsMemorySessionSaver)
   - 基于 LangGraph Checkpoint 的会话状态持久化
   - 支持跨会话、跨应用的状态恢复
   - 企业级数据安全与隔离

2. 身份认证 (AgentArtsIdentity + require_access_token)
   - 自动 OAuth2 凭据获取与管理
   - 支持多种第三方平台集成（GitHub/GitLab等）
   - 无需手动管理令牌，平台自动处理刷新

3. 安全代码执行 (AgentArts Code Sandbox)
   - 企业级沙箱环境，完全隔离
   - 自动资源控制（CPU/内存/超时）
   - 危险操作自动拦截
   - 完整执行日志与审计追溯

4. 工具生态
   - 文件操作、URL访问
   - GitHub API 深度集成（自动获取访问令牌）
   - 平台级代码执行沙箱
   - 可扩展的工具注册机制

平台优势:
    - 开箱即用的企业级记忆能力
    - 简化的第三方服务认证流程
    - 安全的代码执行沙箱
    - 完善的会话隔离与安全机制
================================================================================
"""

import os
import uuid
import json
import requests
import urllib3
from typing import List, Optional, Dict, Any
from pathlib import Path
from dotenv import load_dotenv
from langchain.agents import AgentState
# 禁用 SSL 证书警告
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# ============================================================================
# 基础依赖：LangGraph + LangChain
# ============================================================================
# LangGraph: AI Agent 工作流编排框架
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode
# 注：InMemorySaver 仅保留作为参考，生产环境使用 AgentArtsMemorySessionSaver
# from langgraph.checkpoint.memory import InMemorySaver

# LangChain: LLM 应用开发工具链
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage, SystemMessage,
ToolMessage
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
# ============================================================================
# AgentArts 平台能力依赖
```


版权所有 © 华为云计算技术有限公司

```
# ==============================================================================
# AgentArtsIdentity: 平台级身份认证服务
# - require_access_token: 声明式 OAuth2 凭据注入装饰器
# - 自动处理令牌获取、刷新、存储
from agentarts.sdk.identity.auth import require_access_token
# AgentArtsMemory:
# - AgentArtsMemorySessionSaver: 企业级会话状态持久化，集成 LangGraph 框架
# - 替代标准 InMemorySaver，提供生产级能力
from agentarts.sdk.integration.langgraph import AgentArtsMemorySessionSaver
from agentarts.sdk.memory import MemoryClient
# AgentArts Runtiem: 核心启动类
from agentarts.sdk import AgentArtsRuntimeApp, RequestContext
# ==============================================================================
# 第一部分：配置模块
# ==============================================================================
# 从 .env 文件加载环境变量
load_dotenv()
class ModelConfig:
    """
    模型配置类

    从环境变量读取模型配置：
    - MODEL_NAME: 模型名称（默认 gpt-4）
    - MODEL_URL: 模型API地址（默认 OpenAI 地址）
    - MODEL_API_KEY: 模型API密钥
    - MODEL_TYPE: 模型类型（默认 openai）
    """

    def __init__(self):
        self.name = os.getenv("MODEL_NAME", "gpt-4")
        self.url = os.getenv("MODEL_URL", "https://api.openai.com/v1")
        self.api_key = os.getenv("MODEL_API_KEY")
        self.model_type = os.getenv("MODEL_TYPE", "openai")
class AgentConfig:
    """
    Agent全局配置类

    整合所有配置项：
    - model: 模型配置
    - max_iterations: Agent最大迭代次数（防止无限循环）
    - memory_space: AgentArts 记忆空间ID
    """

    def __init__(self):
        self.model = ModelConfig()
        self.max_iterations = int(os.getenv("MAX_ITERATIONS", "50"))
        self.memory_space = os.getenv("MEMORY_SPACE_ID", "default")
        self.region = os.getenv("HUAWEICLOUD_SDK_REGION", "cn-north-4")
# 全局配置实例（单例模式）
config = AgentConfig()
# ==============================================================================
# 第二部分：平台级记忆系统 (AgentArts Checkpoint)
# ==============================================================================
# AgentArts 平台提供的企业级会话记忆能力
#
# 与标准 LangGraph InMemorySaver 的区别：
# ┌─────────────────────────┬────────────────────────────────┬──────────────────────────────────┐
# │ 特性             │ InMemorySaver (标准)        │ AgentArtsMemorySessionSaver │
# ├─────────────────────────┼────────────────────────────────┼──────────────────────────────────┤
# │ 存储位置          │ 进程内存                   │ 平台持久化存储           │
# │ 多实例共享          │ 不支持                    │ 支持（跨实例共享）         │
# │ 会话隔离          │ 基础隔离                   │ 企业级租户隔离           │
# │ 状态恢复          │ 仅当前进程                  │ 跨应用、跨会话恢复          │
# │ 数据安全          │ 重启丢失                   │ 企业级数据保护           │
# │ 扩展性           │ 单机                     │ 分布式集群支持           │
# └─────────────────────────┴────────────────────────────────┴──────────────────────────────────┘
```

```
#
# 核心能力:
# - space_id: 记忆空间标识，支持多租户隔离
# - region: 区域配置，支持跨区域部署
# - 自动状态持久化：每次 Agent 执行后自动保存状态
# - 快速状态恢复：从持久化存储中恢复会话上下文

_global_checkpointer: Optional[AgentArtsMemorySessionSaver] = None


def get_checkpointer() -> AgentArtsMemorySessionSaver:
    """
    获取平台级 Checkpointer 实例（单例模式）

    使用 AgentArtsMemorySessionSaver 替代标准的 InMemorySaver，
    实现企业级的会话状态持久化能力。

    平台优势:
        - 多实例共享：支持 Agent 部署多副本时共享会话状态
        - 租户隔离：通过 space_id 实现企业级数据隔离
        - 状态恢复：服务重启后自动恢复会话上下文
        - 零运维：无需自行搭建 Redis 等存储服务

    Returns:
        AgentArtsMemorySessionSaver: AgentArts 平台级记忆存储
    """
    global _global_checkpointer
    if _global_checkpointer is None:
        _global_checkpointer = AgentArtsMemorySessionSaver(space_id=config.memory_space,
region=config.region)
    return _global_checkpointer
def get_thread_config(session_id: str) -> Dict[str, Any]:
    """
    获取 LangGraph 运行时配置

    LangGraph 通过 config 中的 thread_id 来区分不同的会话

    Args:
        session_id: 会话ID（对应 LangGraph 的 thread_id）

    Returns:
        LangGraph 运行时配置字典
    """
    return {"configurable": {"thread_id": session_id}}
# ============================================================================
# 第三部分：LLM 客户端
# ============================================================================

class LLMClient:
    """
    LLM 客户端封装类

    封装 LangChain 的 ChatOpenAI，提供:
    - 模型调用能力
    - 工具绑定能力（让模型能够调用工具）
    """

    def __init__(self):
        self._client = self._create_client()
    def _create_client(self):
        """创建 OpenAI 兼容的 Chat 客户端"""
        model_cfg = config.model
        return ChatOpenAI(
            model=model_cfg.name,
            base_url=model_cfg.url,
            api_key=model_cfg.api_key,
            temperature=0, # 设为0以获得更确定性的输出
            streaming=False,
        )
```

```
    def invoke(self, messages: List[BaseMessage]) -> BaseMessage:
        """调用 LLM 生成回复"""
        return self._client.invoke(messages)
    def bind_tools(self, tools: List[Any]):
        """绑定工具到 LLM，使模型能够调用这些工具"""
        return self._client.bind_tools(tools)
# 全局 LLM 客户端实例
llm_client = LLMClient()
# ============================================================================
# 第四部分：工具定义
# ============================================================================
# 工具说明：
# - 使用 @tool 装饰器将函数定义为 LangChain 工具
# - 工具函数必须有清晰的文档字符串，模型会根据描述决定是否调用
# - 工具返回值必须是字符串（会被返回给模型作为上下文）
#
# 平台工具 vs 自建工具:
# ┌──────────────────┬──────────────────────────┬──────────────────────────┐
# │ 特性             │ 自建工具                 │ AgentArts 平台工具       │
# ├──────────────────┼──────────────────────────┼──────────────────────────┤
# │ 代码执行         │ 自行搭建沙箱             │ 企业级安全沙箱           │
# │ 安全性           │ 需要自行处理恶意代码     │ 自动隔离危险操作         │
# │ 资源限制         │ 自行实现                 │ 自动 CPU/内存/超时控制   │
# │ 状态管理         │ 自行处理会话状态         │ 跨调用状态持久化         │
# │ 监控告警         │ 自行搭建                 │ 平台统一监控             │
# └──────────────────┴──────────────────────────┴──────────────────────────┘


@tool
def execute_python(code: str, description: str = "") -> str | None:
    """
    在平台沙箱环境中安全执行 Python 代码

    本工具使用 AgentArts 平台提供的 code_session 企业级代码执行能力，
    实现了完全隔离的安全沙箱环境，可以安全地执行 Agent 生成的代码。

    平台沙箱能力:
    ┌─────────────────────────────────────────────────────────────┐
    │ 安全隔离                                                    │
    ├─────────────────────────────────────────────────────────────┤
    │  完全隔离的执行环境，代码无法访问宿主机器资源               │
    │  网络隔离：仅允许特定域名访问（可选配置）                   │
    │  文件系统隔离：仅能访问临时工作目录                         │
    │  禁止危险操作：禁止 subprocess/threading/文件直接访问等     │
    ├─────────────────────────────────────────────────────────────┤
    │ 资源控制                                                    │
    ├─────────────────────────────────────────────────────────────┤
    │  CPU 限制：防止无限循环占用资源                             │
    │  内存限制：防止内存泄漏导致系统崩溃                         │
    │  执行超时：自动终止长时间运行的代码                         │
    │  磁盘配额：防止恶意写入大量数据                             │
    ├─────────────────────────────────────────────────────────────┤
    │ 企业级特性                                                  │
    ├─────────────────────────────────────────────────────────────┤
    │  执行日志：完整记录代码执行过程                             │
    │  审计追溯：记录谁在什么时候执行了什么代码                   │
    │  异常捕获：自动捕获并安全处理执行中的异常                   │
    │  多区域支持：可选择不同区域的执行环境                       │
    └─────────────────────────────────────────────────────────────┘
```

```
    对比自建沙箱:
                自建方案                    vs    AgentArts 平台方案
    ┌─────────────────────────────────────┬──────────────────────────────────
────────────┐
    │ 需要自行处理:                        │ 平台全托管:                        │
    │  Docker 容器管理                     │  开箱即用的容器化执行环境              │
    │  资源限制配置                        │  自动 CPU/内存/超时控制              │
    │  安全漏洞修补                        │  平台持续安全更新                   │
    │  执行监控告警                        │  统一监控与告警                     │
    │  日志收集分析                        │  结构化日志与审计                   │
    └─────────────────────────────────────┴──────────────────────────────────
────────────┘

    使用示例:
        # Agent 可以直接生成并执行代码
        result = execute_python.invoke("print([x**2 for x in range(10)])")
        # 返回: "[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]"

    Args:
        code: 要执行的 Python 代码
        description: 代码的简要描述（可选，用于日志和调试）

    Returns:
        执行结果的 JSON 字符串
    """
    from agentarts.sdk.tools import code_session
    if description:
        code = f"# {description}\n{code}"

    print(f"\n Generated Code: {code}")
    # 使用平台提供的 code_session 沙箱执行代码
    with code_session(config.region, "your-interpreter-name") as code_client:
        response = code_client.invoke(
            operate_type="execute_code",
            arguments={
                "code": code,
                "language": "python",
                "clearContext": False  # 保持执行上下文，允许跨调用共享变量
            }
        )
        print(response)
    return json.dumps(response["result"])
@tool
def read_file(file_path: str) -> str:
    """
    读取文件内容

    Args:
        file_path: 文件路径（绝对路径或相对路径）

    Returns:
        文件内容（最大1MB）
    """
    try:
        path = Path(file_path)
        if not path.exists():
            return f"错误: 文件不存在 - {file_path}"
        if path.stat().st_size > 1024 * 1024:
            return f"错误: 文件过大 (最大支持1MB) - {file_path}"
        return path.read_text(encoding='utf-8')
    except Exception as e:
        return f"错误: 读取文件失败 - {str(e)}"


@tool
def write_file(file_path: str, content: str) -> str:
    """
    写入内容到文件（覆盖模式）
```


版权所有 © 华为云计算技术有限公司

```
    Args:
        file_path: 文件路径
        content: 要写入的内容

    Returns:
        操作结果
    """
    try:
        path = Path(file_path)
        path.parent.mkdir(parents=True, exist_ok=True)
        path.write_text(content, encoding='utf-8')
        return f"成功写入文件: {file_path}"
    except Exception as e:
        return f"错误: 写入文件失败 - {str(e)}"


@tool
def append_file(file_path: str, content: str) -> str:
    """
    追加内容到文件末尾

    Args:
        file_path: 文件路径
        content: 要追加的内容

    Returns:
        操作结果
    """
    try:
        path = Path(file_path)
        path.parent.mkdir(parents=True, exist_ok=True)
        with open(path, 'a', encoding='utf-8') as f:
            f.write(content)
        return f"成功追加内容到文件: {file_path}"
    except Exception as e:
        return f"错误: 追加文件失败 - {str(e)}"


@tool
def list_directory(dir_path: str = ".") -> str:
    """
    列出目录内容

    Args:
        dir_path: 目录路径（默认当前目录）

    Returns:
        目录内容列表
    """
    try:
        path = Path(dir_path)
        if not path.exists():
            return f"错误: 目录不存在 - {dir_path}"
        if not path.is_dir():
            return f"错误: 不是目录 - {dir_path}"

        items = []
        for item in sorted(path.iterdir()):
            item_type = "DIR" if item.is_dir() else "FILE"
            size = item.stat().st_size if item.is_file() else 0
            items.append(f"{item_type:6} | {size:>10} | {item.name}")
        return "类型     | 大小       | 名称\n" + "\n".join(items)
    except Exception as e:
        return f"错误: 列出目录失败 - {str(e)}"


@tool
def fetch_url(url: str, method: str = "GET", headers: Optional[Dict] = None, body: Optional[str] = None) ->
```

```
str:
    """
    访问 URL 获取内容

    Args:
        url: 目标URL
        method: HTTP方法（GET/POST）
        headers: 自定义请求头
        body: 请求体（用于POST）

    Returns:
        响应内容
    """
    try:
        default_headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36"}
        if headers:
            default_headers.update(headers)
        request_kwargs = {"url": url, "headers": default_headers, "timeout": 30}
        if method.upper() == "POST":
            request_kwargs["data"] = body.encode() if body else None
        elif method.upper() != "GET":
            return f"错误: 不支持的HTTP方法 - {method}"

        response = requests.request(method, **request_kwargs)
        if len(response.content) > 1024 * 1024:
            return f"响应内容过大 ({len(response.content)} bytes)，已截断\n\n" + response.text[:10000]
        return f"状态码: {response.status_code}\n\n{response.text}"
    except Exception as e:
        return f"错误: 访问URL失败 - {str(e)}"


@tool
@require_access_token(
    provider_name="your_provider", # AgentArts 平台注册的 GitHub 身份提供商
    scopes=["user:email"],          # 请求的 OAuth2 权限范围
    auth_flow="USER_FEDERATION",     # 用户联邦认证流程
ignore_ssl_verification=True,
)
def github_get(url: str, headers: Optional[Dict] = None, access_token: Optional[str] = None) -> str:
    """
    访问 GitHub API（平台级身份认证）

    本函数使用 AgentArts 平台的 require_access_token 装饰器实现自动身份认证，
    无需手动管理 GitHub 访问令牌。

    平台能力 vs 传统方式对比:
    ┌──────────────┬──────────────────────┬──────────────────────────
    ──────────────┐
    │ 特性          │ 传统方式              │ AgentArts + require_access_ │
    │               │                      │ token                  │
    ├──────────────┼──────────────────────┼──────────────────────────
    ──────────────┤
    │ 令牌管理      │ 手动创建、存储、轮换    │ 平台自动管理          │
    │ 令牌获取      │ 用户自行配置 env 或配置 │ 装饰器自动注入        │
    │ 令牌刷新      │ 需要编写刷新逻辑       │ 平台自动处理          │
    │ 安全性       │ 令牌暴露在代码或环境变量 │ 令牌存储在平台安全存储 │
    │ 用户体验      │ 配置复杂，容易出错     │ 声明式配置，开箱即用   │
    └──────────────┴──────────────────────┴──────────────────────────
    ──────────────┘

    认证流程说明:
        1. 用户首次调用时，平台自动发起 OAuth2 授权流程
        2. 用户在 GitHub 授权页面完成授权
        3. 平台获取并安全存储访问令牌
        4. 后续调用时，access_token 自动注入到函数参数
        5. 令牌过期前，平台自动刷新

    Args:
        url: GitHub API 路径（如 /repos/owner/repo/issues）
```


版权所有 © 华为云计算技术有限公司

```
        headers: 自定义请求头
        access_token: [平台自动注入] OAuth2 访问令牌

    Returns:
        API 响应内容
    """
    # 记录 token 状态
    has_token = bool(access_token)
    print(f"[GITHUB] access_token: {'已注入' if has_token else '未注入'}")
    base_url = "https://api.github.com"

    default_headers = {
        "User-Agent": "LangGraph-Agent",
        "Accept": "application/vnd.github.v3+json",
    }
    if access_token:
        default_headers["Authorization"] = f"token {access_token}"
    if headers:
        default_headers.update(headers)
    # 获取代理配置
    proxies = {}
    http_proxy = os.environ.get("HTTP_PROXY") or os.environ.get("http_proxy")
    https_proxy = os.environ.get("HTTPS_PROXY") or os.environ.get("https_proxy")
    if http_proxy:
        proxies["http"] = http_proxy
    if https_proxy:
        proxies["https"] = https_proxy
    try:
        full_url = base_url + url if url.startswith("/") else url
        # 禁用 SSL 证书验证（仅用于测试）
        response = requests.get(full_url, headers=default_headers, timeout=30, proxies=proxies if proxies else
None, verify=False)
        # 记录响应状态
        print(f"[GITHUB] 响应状态: {response.status_code}")
        if len(response.content) > 1024 * 1024:
            return f"响应内容过大，已截断\n\n{response.text[:10000]}"

        return f"状态码: {response.status_code}\n\n{response.text}"
    except Exception as e:
        print(f"[GITHUB] 请求失败: {str(e)}")
        return f"错误: GitHub API请求失败 - {str(e)}"


@tool
def github_get_file(repo: str, file_path: str, ref: str = "main") -> str:
    """
    获取 GitHub 仓库中的文件内容

    Args:
        repo: 仓库名称（如 "owner/repo"）
        file_path: 文件路径（如 "src/main.py"）
        ref: 分支或标签名（默认 "main"）

    Returns:
        文件内容
    """
    import base64
    url = f"/repos/{repo}/contents/{file_path}?ref={ref}"
    result = github_get.invoke(url)
    if "状态码: 200" in result:
        try:
            json_start = result.find("{")
            json_end = result.rfind("}") + 1
            if json_start >= 0 and json_end > json_start:
                data = json.loads(result[json_start:json_end])
                content = data.get("content", "")
                if content:
                    decoded = base64.b64decode(content).decode("utf-8")
                    return f"文件: {file_path}\n分支: {ref}\n\n{decoded}"
```

 版权所有 © 华为云计算技术有限公司

```
        except Exception as e:
            return f"错误: 解析文件内容失败 - {str(e)}\n\n原始响应:\n{result}"

    return result
@tool
def github_search(query: str, search_type: str = "repositories") -> str:
    """
    搜索 GitHub

    Args:
        query: 搜索关键词
        search_type: 搜索类型（repositories/code/issues/commits）

    Returns:
        搜索结果
    """
    url = f"/search/{search_type}?q={requests.utils.quote(query)}"
    return github_get.invoke(url)
@tool
def github_get_issues(repo: str, state: str = "open", per_page: int = 30) -> str:
    """
    获取 GitHub 仓库的 Issue 列表

    Args:
        repo: 仓库名称（如 "langchain-ai/langgraph"）
        state: Issue 状态（open/closed/all）
        per_page: 返回数量（默认30）

    Returns:
        Issue 列表（JSON 格式）
    """
    url = f"/repos/{repo}/issues?state={state}&per_page={per_page}"
    return github_get.invoke(url)
@tool
def github_get_issue_comments(repo: str, issue_number: int) -> str:
    """
    获取 GitHub 仓库某个 Issue 的评论

    Args:
        repo: 仓库名称（如 "langchain-ai/langgraph"）
        issue_number: Issue 编号

    Returns:
        评论列表（JSON 格式）
    """
    url = f"/repos/{repo}/issues/{issue_number}/comments"
    return github_get.invoke(url)
@tool
def github_get_pull_requests(repo: str, state: str = "open", per_page: int = 30) -> str:
    """
    获取 GitHub 仓库的 Pull Request 列表

    Args:
        repo: 仓库名称（如 "langchain-ai/langgraph"）
        state: PR 状态（open/closed/merged/all）
        per_page: 返回数量（默认30）

    Returns:
        PR 列表（JSON 格式）
    """
    url = f"/repos/{repo}/pulls?state={state}&per_page={per_page}"
    return github_get.invoke(url)
def get_all_tools() -> List:
    """
    获取所有可用工具

    Returns:
        工具列表
    """
```

```
    tools = [
        execute_python,
        read_file,
        write_file,
        append_file,
        list_directory,
        fetch_url,
        github_get,
        github_get_file,
        github_search,
        github_get_issues,
        github_get_issue_comments,
        github_get_pull_requests,
    ]
    return tools

class AgentState(TypedDict):
    """Agent 状态类型定义"""
    messages: List
    iteration: int
def should_continue(state: AgentState) -> str:
    """
    判断是否继续执行（条件边函数）

    如果 LLM 返回了工具调用，则继续执行工具节点

    Args:
        state: 当前 Agent 状态

    Returns:
        "continue": 继续执行工具
        "end": 结束执行
    """
    last_message = state["messages"][-1]
    has_tool_calls = hasattr(last_message, "tool_calls") and last_message.tool_calls
    return "continue" if has_tool_calls else "end"


def model_node(state: AgentState) -> AgentState:
    """
    模型节点：调用 LLM 生成回复

    将工具绑定到 LLM，让模型能够决定是否需要调用工具

    Args:
        state: 当前 Agent 状态

    Returns:
        更新后的状态（包含 LLM 回复）
    """
    tools = get_all_tools()
    llm_with_tools = llm_client.bind_tools(tools)
    response = llm_with_tools.invoke(state["messages"])
    # ====== 工具调用日志 ======
    has_tool_calls = hasattr(response, 'tool_calls') and response.tool_calls
    if has_tool_calls:
        for tc in response.tool_calls:
            print(f"[MODEL] 调用工具: {tc.get('name', 'unknown')}")
    # ====== 日志结束 ======

    new_messages = state["messages"] + [response]
    iteration = state.get("iteration", 0) + 1

    return {
        "messages": new_messages,
        "iteration": iteration,
    }
def create_agent_graph(checkpointer=None):
    """
```

```
    创建 Agent 工作流图

    使用 LangGraph 的 StateGraph 构建工作流：
    1. 添加 agent 节点（调用 LLM）
    2. 添加 tools 节点（执行工具）
    3. 设置条件边（根据是否有工具调用决定走向）
    4. 编译图并绑定 checkpointer

    Args:
        checkpointer: 状态持久化器

    Returns:
        编译后的 LangGraph 图
    """
    tools = get_all_tools()
    # 自定义 ToolNode - 合并工具返回的消息和原有消息
    def tool_node_with_log(state: AgentState) -> AgentState:
        """工具节点：执行工具并合并消息"""
        original_messages = list(state["messages"])
        result = ToolNode(tools).invoke(state)
        # 合并原有消息和工具返回的 ToolMessage
        tool_messages = result["messages"]
        merged_messages = original_messages + tool_messages
        return {
            "messages": merged_messages,
            "iteration": state.get("iteration", 0),
        }
    workflow = StateGraph(AgentState)
    # 添加节点
    workflow.add_node("agent", model_node)
    workflow.add_node("tools", tool_node_with_log)
    # 设置入口点
    workflow.set_entry_point("agent")
    # 添加条件边：检查是否需要调用工具
    workflow.add_conditional_edges(
        "agent",
        should_continue,
        {"continue": "tools", "end": END}
    )
    # 工具执行后返回 agent 节点
    workflow.add_edge("tools", "agent")
    # 编译图（传入checkpointer）
    return workflow.compile(checkpointer=checkpointer)
# ============================================================================
# 第六部分：Agent 主类
# ============================================================================

class LangGraphAgent:
    """
    LangGraph Agent 主类

    对外提供的核心类，封装了：
    - 工作流图的创建和执行
    - 会话状态管理
    - 对话历史的存取

    使用示例:
        agent = LangGraphAgent()
        response = agent.run("你好", session_id="user-001")
        history = agent.get_history(session_id="user-001")
    """

    def __init__(self, system_prompt: Optional[str] = None, checkpointer=None):
        """
        初始化 Agent

        Args:
            system_prompt: 系统提示词（可选，默认值见 _default_system_prompt）
            checkpointer: 自定义 Checkpointer（可选，默认使用全局单例）
```


版权所有 © 华为云计算技术有限公司

```
        """
        self._checkpointer = checkpointer or get_checkpointer()
        self.graph = create_agent_graph(checkpointer=self._checkpointer)
        self.system_prompt = system_prompt or self._default_system_prompt()
        self._last_session_id: Optional[str] = None

    def _default_system_prompt(self) -> str:
        """默认系统提示词"""
        return """你是一个智能助手，可以通过调用工具来帮助用户完成任务。

可用的工具：
- execute_python: 执行Python代码
- read_file: 读取本地文件内容
- write_file: 写入本地文件
- append_file: 追加内容到本地文件
- list_directory: 列出本地目录内容
- fetch_url: 访问URL获取网页内容
- github_get: 访问GitHub API
- github_search: 搜索GitHub仓库/代码/issues
- github_get_issues: 获取仓库的Issue列表
- github_get_issue_comments: 获取Issue的评论
- github_get_pull_requests: 获取仓库的PR列表
- github_get_file: 获取仓库中的文件内容

当用户询问GitHub相关问题时（如查看仓库、Issue、PR等），请使用GitHub工具来获取信息。
GitHub仓库是线上的（如 langchain-ai/langgraph），不是本地目录。
"""

    def _get_config(self, session_id: str) -> Dict[str, Any]:
        """获取运行时配置"""
        return get_thread_config(session_id)
    def run(self, user_input: str, session_id: Optional[str] = None) -> str:
        """
        运行 Agent 处理用户输入

        这是主要入口方法，每次调用都会：
        1. 从 Checkpoint 恢复会话状态（如有）
        2. 添加用户消息
        3. 执行工作流
        4. 自动保存状态到 Checkpoint

        Args:
            user_input: 用户输入的文本
            session_id: 会话ID（可选）
                - 不传：使用上次会话ID或自动生成新ID
                - 传值：使用指定会话ID，实现多会话隔离

        Returns:
            Agent 的回复文本
        """
        # 确定会话ID
        if session_id is None:
            session_id = self._last_session_id or str(uuid.uuid4())
        # 创建会话（如果已存在则忽略，继续执行）
        memory_client = MemoryClient()
        try:
            memory_client.create_memory_session(space_id=config.memory_space, id=session_id)
        except Exception as e:
            # Session already exists, continue with existing session
            if "Session id already exists" in str(e):
                pass
            else:
                raise

        self._last_session_id = session_id
        thread_config = self._get_config(session_id)
        # 从 checkpointer 获取历史消息
        checkpoint = self._checkpointer.get(thread_config)
        if checkpoint and "channel_values" in checkpoint:
```

```
                history_messages = list(checkpoint["channel_values"].get("messages", []))
            else:
                history_messages = []
            # 如果有历史消息，添加到现有消息后面（避免重复添加用户消息）
            if history_messages:
                # 检查最后一条是否是相同的用户消息
                if not (history_messages and isinstance(history_messages[-1], HumanMessage) and
                        history_messages[-1].content == user_input):
                    messages = history_messages + [HumanMessage(content=user_input)]
                else:
                    messages = history_messages
            else:
                # 首次对话，添加系统消息
                messages = [
                    SystemMessage(content=self.system_prompt),
                    HumanMessage(content=user_input)
                ]
            # 执行工作流
            result = self.graph.invoke({"messages": messages}, config=thread_config)
            # 返回最后一条消息的内容
            last_message = result["messages"][-1]
            return last_message.content if hasattr(last_message, "content") else str(last_message)
        @property
        def session_id(self) -> Optional[str]:
            """获取当前会话ID"""
            return self._last_session_id
```


版权所有 © 华为云计算技术有限公司

4 组件库

4.1 沙箱工具

4.1.1 沙箱工具介绍

什么是沙箱工具

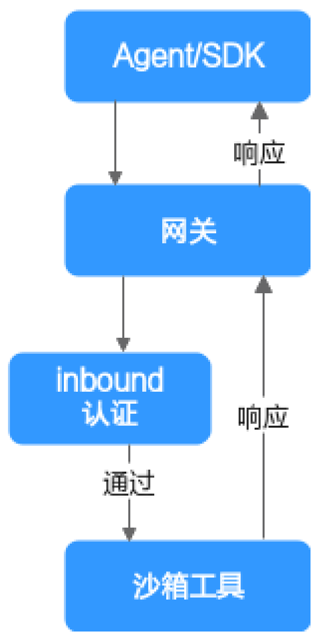
沙箱工具是我们为Agent提供的在安全的沙箱环境中运行的技能增强包。通过代码解释器等原生工具，Agent能够超越纯文本对话，具备处理复杂逻辑计算 、实时信息获取乃至跨系统操作的能力。所有工具均运行在受控的沙箱环境中，确保执行过程的高效与安全。

应用场景

为Agent提供在安全、隔离的代码执行环境中运行的内置工具，使Agent具备运行代码、操作文件、执行系统命令的能力，支撑数据分析、自动化脚本、可视化报告生成等业务场景。

工作原理

图 4-1 工作原理

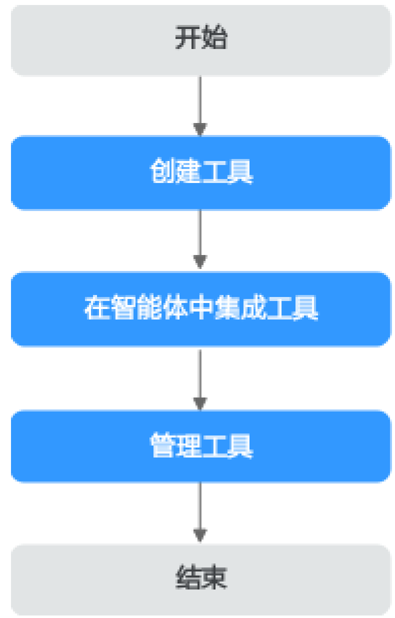


1. **Agent/SDK**：应用程序或客户端通过Agent或SDK发起请求。
2. **网关**：Agent/SDK发起的请求首先会到达网关。网关作为系统的入口，负责接收和处理来自客户端的所有请求。
3. **inbound认证**：请求通过网关后，会进入inbound认证阶段。在此阶段，系统会对请求进行身份验证和授权检查，确保请求来自合法的用户或服务。只有通过认证的请求才能继续向下传递。
4. **沙箱工具**：通过inbound认证的请求最终会到达沙箱工具。沙箱工具是一个隔离的环境，用于在不影响生产环境的情况下测试和验证代码、功能或服务。它可以模拟真实的运行环境，提供安全的测试平台，帮助开发人员调试和优化他们的应用程序。
5. **响应**：沙箱工具处理完请求后，会生成相应的响应，并通过网关返回给Agent/SDK。这个响应可能包含测试结果、错误信息或其他相关数据，帮助开发人员了解请求的处理情况。

版权所有 © 华为云计算技术有限公司

使用流程

图 4-2 使用流程



1. 创建工具：允许用户在安全的环境中测试和验证智能体的功能，确保在实际部署前能够发现和解决问题，当前仅支持创建代码解释器工具。具体请参见创建工具。
2. 在智能体中集成工具：将工具集成至智能体代码，可让智能体调用隔离环境完成代码运行等复杂任务。具体请参见在智能体中集成工具。
3. 管理工具：在工具详情页面，您可以查看网关在使用过程中产生的日志。具体请参见查看代码解释器日志。

4.1.2 示例：基于控制台的工具开发与应用实践

本示例介绍基于控制台的工具开发与应用实践。

基于控制台的工具开发与应用实践

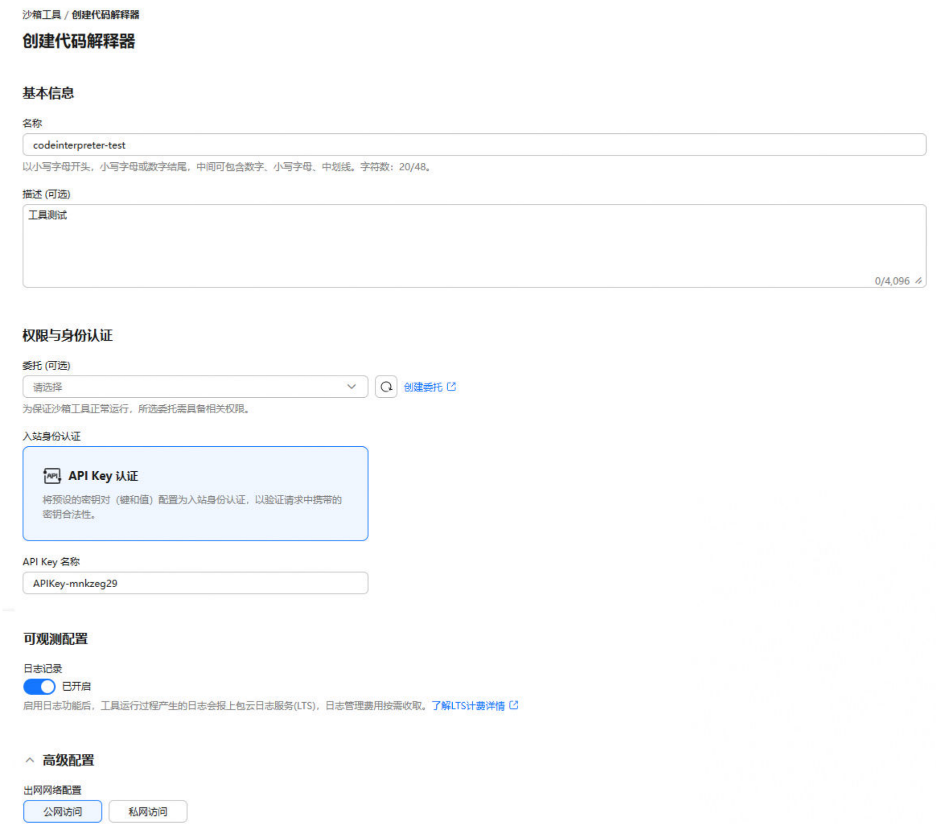
步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“沙箱工具”页签，进入沙箱工具界面。

步骤3 单击右上角“创建代码解释器”，参考**图4-3**配置后单击“立即创建”。

版权所有 © 华为云计算技术有限公司

图 4-3 创建代码解释器



步骤4 创建代码解释器创建完成后，在列表可查看已创建的代码解释器的名称。

步骤5 单击列表操作列的“调用代码”，在开发智能体应用的代码中定义能够执行指定python代码的工具。

```
from hw-agentarts-sdk.tools import code_session

@tools
def execute_python_tool(code: str, description: str = ""):
    with code_session("your_region", "your_code_interpreter_name") as code_client:
        response = code_client.invoke(
            operate_type="execute_code",
            arguments={
                "code": code,
                "language": "python",
                "clearContext": False
            }
        )
```

步骤6 将步骤5中的执行代码能力封装成工具，即可集成到智能体中，此处以LangGraph框架开发智能体为例。

```
agent = Agent(
    model=model,
    tools=[execute_python_tool],
    system_prompt=""
    callback_handler=None
)
```

----结束

4.1.3 示例：基于 SDK 的工具开发与应用实践

本示例介绍基于SDK的工具开发与应用实践，通过LangGraph框架开发Agent并集成hw-agentarts-sdk工具，展示从开发到部署的完整流程。

前提条件

已安装必须的Python包，包括hw-agentarts-sdk、langgraph， langchain-openai等。

基于 SDK 的工具开发与应用实践

步骤1 下载AgentRuntime SDK包及其它依赖包。

```
pip install hw-agentarts-sdk
pip install langgraph
pip install langchain_openai
```

步骤2 导入代码解释器及其它包。

```
import json
import os
from typing import TypedDict, Union
from langgraph.graph import StateGraph, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage, AIMessage
from langchain_core.tools import tool
from hw-agentarts-sdk import HuaweiAgentRunApp
from hw-agentarts-sdk.tools import code_session
from langgraph.prebuilt import ToolNode
app = HuaweiAgentRunApp()
```

步骤3 创建工具。

代码解释器工具创建需要时间拉起，此步骤请提前执行，拉起时间大约为5s。

```
def create():
    client = CodeInterpreter(region="your_region")
    code_interpreter = client.create_code_interpreter(
        name="your_code_interpreter_name",
        api_key_name="your_api_key_name"
    )
```

步骤4 开发工具。

此步骤以开发一个执行python代码的工具为例，工具接受模型生成的代码和描述，执行代码并返回代码的执行结果。

```
@tool
def execute_python_tool(code: str, description: str = "") -> str | None:
    with code_session("your_region", "your_code_interpreter_name") as code_client:
        response = code_client.invoke(
            operate_type="execute_code",
            arguments={
                "code": code,
                "language": "python",
                "clearContext": False
            }
        )
    return json.dumps(response["result"])
```

步骤5 编写系统提示词。

```
SYSTEM_PROMPT = """你是一个通过代码执行验证所有答案的优秀AI助手

验证原则:
1. 当需要代码、算法或计算来验算时，你需要编写代码来验证它们。
```

```
2. 使用 execute_python 来测试数学计算、算法和逻辑。
3. 返回答案之前，使用测试脚本来验证你的理解。
4. 只能通过实际的代码执行展示工作过程。
5. 如果存在不确定的情况，详细说明限制条件并做尽可能的验证

方法:
- 如果问题涉及编程的概念，就通过代码实现
- 如果要求你计算，编写程序计算并显示具体代码
- 如果需要实现算法，你还要编写测试用例来进行确认
- 记录验证的过程展示给用户

工具:
- execute_python: 执行Python代码并获得输出

响应格式: execute_python工具将返回一个json响应，包括:
- content: 内容对象的数组，每个对象包含type和text/data"""
```

步骤6 开发Agent并集成工具。

此步骤请填写拥有使用权限的模型（此处以DeepSeek-V3为例），并补充下面的api_key，base_url信息，并可按你所需自定义模型的参数。同时，将自定义工具集成到Agent中

```
# 创建Agent
llm = ChatOpenAI(
    model="DeepSeek-V3",
    api_key=os.environ.get("MODEL_API_KEY", ""),
    base_url=os.environ.get("BASE_URL", ""),
    max_tokens=1000,
    temperature=0.7,
)
# 创建工具列表
tools = [execute_python_tool]
# 工具绑定Agent
llm.bind_tools(tools)
# 定义graph状态
class AgentState(TypedDict):
    messages: list[Union[HumanMessage, SystemMessage, AIMessage]]
def call_model(state: AgentState):
    """调用模型并返回响应"""
    # 只在初始消息为空时添加系统提示
    if not state["messages"] or all(not isinstance(msg, SystemMessage) for msg in state["messages"]):
        messages = [SystemMessage(content=SYSTEM_PROMPT)] + state["messages"]
    else:
        messages = state["messages"]
    response = llm.invoke(messages)
    return {"messages": [response]}
def should_continue(state):
    """判断是否需要继续使用工具"""
    last_message = state["messages"][-1]
    # 如果包含工具调用，则继续执行
    if last_message.tool_calls:
        return "tools"

    # 否则结束
    return END
# 创建LangGraph工作流
workflow = StateGraph(AgentState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", ToolNode(tools))
# 设置入口
workflow.set_entry_point("agent")
# 添加边
workflow.add_conditional_edges(
    "agent",
    should_continue,
    {
        "tools": "tools"
    }
```

```
)
workflow.add_edge("tools", "agent")
agent = workflow.compile()
```

至此，已经完成了一个简易的Agent开发。

步骤7 使用Agent。

```
@app.entrypoint
def agent_chat():
    query = "告诉我1到100之间最大的随机质数"

    # 运行Agent
    result = agent.invoke({
        "messages": [HumanMessage(content=query)]
    })
    print(result["messages"][-1].content)
if __name__ == '__main__':
    app.run()
```

----结束

4.1.4 创建工具

创建沙箱工具允许用户在安全的环境中测试和验证智能体的功能，确保在实际部署前能够发现和解决问题。当前仅支持创建代码解释器。代码解释器能够理解并解释代码的逻辑，帮助用户（特别是初学者）更好地理解代码的运行机制和功能。

创建沙箱工具有两种方式：

- 通过控制台创建代码解释器
- 通过SDK创建代码解释器

通过控制台创建代码解释器

支持用户在控制台上通过配置相关参数创建代码解释器。

前提条件

- （可选）已创建委托，具体请参考创建委托。
- 如需通过私有网络VPC进行访问工具，请确保已开通虚拟私有云，并已创建私有网络、子网、安全组。详细操作请参见创建虚拟私有云和子网、创建安全组。

操作步骤

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“沙箱工具”页签，进入沙箱工具界面。

步骤3 单击右上角“创建代码解释器”。

步骤4 在“创建代码解释器”的弹框中，输入相关配置信息，参数说明请参考**表4-1**。

表 4-1 参数说明

<table>
<tr><th colspan="2">参数</th><th>说明</th></tr>
<tr><td rowspan="2">基本信息</td><td>名称</td><td>工具的名称，同一账号下沙箱工具名称不可重复，创建后不支持修改。
命名规则：
● 以字母开头，之母或数字结尾。
● 名称中需包含数字、字母、中划线。
● 最大长度为48个字符。</td></tr>
<tr><td>描述</td><td>用于对工具内容和用途的简要说明。
命名规则：长度不大于4096个字符。</td></tr>
<tr><td rowspan="2">权限与访问控制</td><td>委托</td><td>授予的代理权限或代理功能，允许代表智能体与外部系统进行通信和交互。
选择IAM中已创建的委托，如未创建，请单击“创建委托”，具体请参考创建委托。</td></tr>
<tr><td>API Key名称</td><td>系统默认选择为“API Key认证”，API Key是用于身份验证和授权访问的唯一标识符。
系统会自动生成API Key的名称，可根据需求自定义修改。</td></tr>
<tr><td>可观测配置</td><td>日志记录</td><td>● 未开启，工具运行过程产生的日志无法上报至云日志服务。
● 开启后，工具运行过程产生的日志会报上包云日志服务(LTS)。</td></tr>
<tr><td>高级配置</td><td>出网网络配置</td><td>● 公网访问：能够连接到外部的互联网，访问外部的资源和服务。
● 私网访问：能够连接到内部的私有网络，访问内部的资源和服务。
当选择“私网访问”时，需配置如下参数：
– 选择已配置的VPC，如未配置，请单击“新建VPC”，具体请参考创建虚拟私有云和子网。
– 选择已配置的子网，如未配置，请单击“新建子网”，具体请参考创建虚拟私有云和子网。
– 选择已配置的安全组，如未配置，请单击“新建安全组”，具体请参考创建安全组。
配置后可单击刷新。</td></tr>
</table>

----结束

通过 SDK 创建代码解释器

代码解释器工具支持在代码沙箱中进行上传、下载和执行代码等操作。SDK提供create_code_interpreter接口用于创建代码解释器工具，通过SDK创建的代码解释器将在代码解释器列表中展示。

前提条件

- 拥有华为云AgentArts Tools的访问权限。
- 拥有创建和管理Tools资源所需的权限。

操作步骤

在Python文件中通过SDK创建代码解释器，相关参数说明以及配置示例如下：

- **方法名**：create_code_interpreter
- **配置以下参数**：

参数名	类型	描述
name	str	**Required** code_interpreter名称。
api_key_name	str	**Required** apiKey名称。
description	str	代码解释器的描述。 **Default**: None
auth_type	str	认证方式。 **Default**: None
execution_agency_name	str	为代码解释器提供访问云服务的权限的IAM委托名。 **Default**: None
observability	Dict	可观测性配置（日志+指标）。 **Default**: None
network_config	Dict	出站网络配置。 **Default**: None
agent_gateway_id	str	Agent Gateway ID. **Default**: None
tags	List	标签列表。 **Default**: None

- **返回值**如下：
  包含代码解释器信息的字典，包括：
  – id (str): 代码解释器的id。
  – name (str): 代码解释器的名称。该名称在您的帐户中必须是唯一的。
  – description (str): 代码解释器的描述，用于LLM上下文分析。
  – created_at (str): 创建时间。
  – updated_at (str): 更新时间。
  – execution_agency_name(str): IAM委托名。
  – agent_gateway_id (str): Agent Gateway ID。
  – workload_identity (Dict): 认证信息。
  – observability (Dict): 可观测性配置（日志+指标）。
  – access_endpoint(str): 访问域名。

- observability(Dict): 可观测性配置（日志+指标）。
- tags(List): 标签列表。
- network_config(Dict): 出站网络配置。

代码示例：

```
from hw-agentarts-sdk.tools import CodeInterpreter
demo_client = CodeInterpreter(region="your_region")
code_interpreter = demo_client.create_code_interpreter(
    name="your_code_interperter_name",
    api_key_name="your_api_key_name"
)
```

后续操作

创建成功后，您可以将工具集成到您的Agent代码中，使Agent能够调用隔离环境来完成代码运行、浏览器操作等复杂任务。详细操作请参见在智能体中集成工具。

更多操作

您还可以对已创建的代码解释器执行如下操作。

表 4-2 更多操作

操作	说明
查看代码解释器	1. 进入工具界面。 2. 在代码解释器列表中单击代码解释器名称，查看代码解释器详情。 3. 在“配置信息”页签，修改代码解释器信息，置灰项不可修改。
删除代码解释器	代码解释器删除后不可恢复，请谨慎操作。 1. 进入工具界面。 2. 单击代码解释器列表操作列的“删除”。 3. 在弹框中确认要删除后一键输入“DELETE”，然后单击“确定”。
复制代码解释器	支持用户基于已配置的工具快速创建新的工具。 1. 进入工具界面。 2. 单击代码解释器列表操作列的“复制”。 3. 修改其基础信息，如名称、描述、参数配置等，配置完成后单击“立即创建”。

4.1.5 在智能体中集成工具

创建工具后，您需要将工具集成到智能体代码中，以便智能体可以调用隔离环境来完成代码运行等复杂任务。

前提条件

- 已创建完成代码解释器工具，具体请参见**创建工具**。
- 已安装Python，且版本不低于3.10。

在智能体中集成工具

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“工具”页签，进入工具界面。

步骤3 在网关列表单击操作列的“调用代码”，通过SDK调用工具实现执行代码的能力，并通过tools装饰器封装成工具，SDK提供code_session函数，可快速启动代码解释器的会话，并支持自动停止会话。

支持的操作类型:

- execute_code：执行代码。
- execute_command：执行命令。
- read_files：读取文件。
- write_files：写入文件。

参数说明:

- code：在代码解释器会话中执行的代码。这是指定编程语言的源代码，将由代码解释器执行。

  长度限制：最小长度为 0，最大长度为 1048576
- command： 要使用该工具执行的命令。长度限制：最小长度为 0，最大长度为 65536。
- language： 要执行的代码所使用的编程语言，告诉代码解释器使用哪种语言运行时来执行代码。支持语言：python
- write_contents：当请求操作为write_files时，待写入的内容，包含待写入数据内容和文件路径。列表限制：最小元素数量为0，最大元素数量为1000。
- paths：当请求操作为read_files时， 所需读取的文件路径，需保证每个路径唯一。列表限制：最小元素数量为0，最大元素数量为1000。

```
from hw-agentarts-sdk.tools import code_session

@tools
def custom_tool_name(code: str, description: str = ""):
    with code_session("your_region", "your_code_interpreter_name") as code_client:
        response = code_client.invoke(
            operate_type="your_operate_type",
            arguments={your_arguments}
        )

    return json.dumps(response["result"])
```

步骤4 将**步骤3**中的执行代码能力封装成工具，即可集成到智能体中，此处以LangGraph框架开发智能体为例。

```
llm = ChatOpenAI(
    model="DeepSeek-V3",
    api_key=os.environ.get("MODEL_API_KEY", ""),
    base_url=os.environ.get("BASE_URL", ""),
    max_tokens=1000,
    temperature=0.7,
)
```

```
# 创建工具列表
tools = [execute_python_tool]
# 将工具列表转换为字典格式（工具名到函数的映射）
tools_dict = {tool.name: tool for tool in tools}
# 工具绑定Agent
llm.bind_tools(list(tools_dict.values()))
```

----结束

后续操作

将开发好的Agent打包成镜像部署到智能体运行时，详细操作请参见部署智能体运行时。

相关文档

创建工具详情请参考创建工具。

4.1.6 查看代码解释器日志

工具创建完成后，在“工具”界面，您可以通过工具状态（所有状态、可用和创建中）或搜索（按工具名称或ID）功能来查找工具。在工具详情页面，您可以查看工具在使用过程中产生的日志。

查看代码解释器日志

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“工具”页签，进入工具界面。

步骤3 单击工具名称，查看工具详情。

步骤4 在“日志”页签，查看工具使用过程中产生的日志。

如未启用日志功能，可在“日志”页签单击“去启用”开启日志记录。

----结束

4.2 记忆库

4.2.1 记忆库概述

什么是记忆库

记忆库是智能体的记忆中枢，可以通过控制台创建记忆库并配置短期和长期记忆策略，有效抽取并整合上下文信息，完成记忆数据的全生命周期管理。

使用优势：

- 开箱即用：
  - 短期记忆 + 长期记忆：支持短期记忆（7~365天）和长期记忆（持久化存储），满足不同时间跨度的记忆需求。
  - 多种记忆策略：支持语义记忆、用户偏好、会话摘要、情景记忆等策略，满足不同场景的记忆需求。

- 多维度隔离：

  按策略类型隔离：支持按空间、会话、用户维度进行记忆隔离，确保数据的安全性和独立性。

- 全托管免运维：

  云上全托管：无需管理数据库等基础设施和记忆处理引擎，实现业务快速上线，降低运维成本和复杂度。

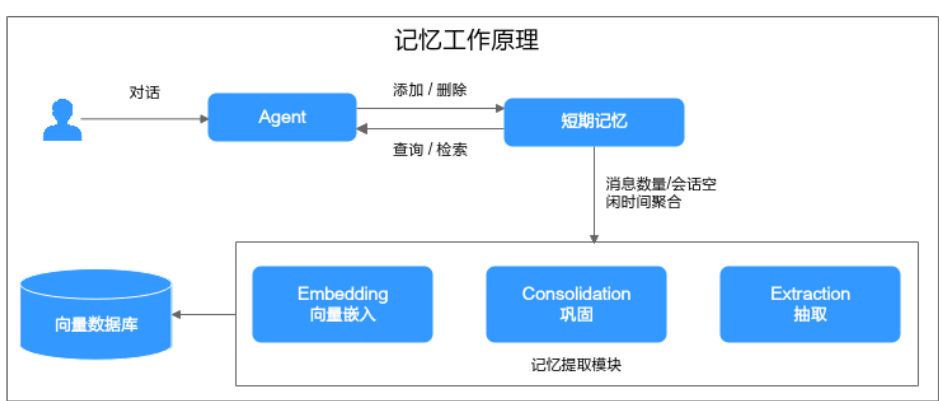
应用场景

应用场景：适用于智能客服、个人助手、问诊等需跨会话连贯交互的场景，不同场景需要用户偏好等不同记忆策略，以支持跨会话个性化与连续性。

使用价值：提升回答精准度与上下文连贯性，避免用户重复输入，提高智能体运行效率、交互精准度与个性化用户体验，灵活适配各类业务场景需求。

工作原理

图 4-4 记忆库工作原理



记忆库工作原理如下：

1. **用户与Agent交互**
   - 用户通过对话与Agent进行交互。用户可以提出问题、发出指令或进行其他形式的交流。
   - Agent接收用户的输入，并对其进行处理。
2. **Agent与短期记忆的交互**
   - **添加/删除**：Agent将用户的对话内容添加到短期记忆中。同时，Agent也可以根据需要从短期记忆中删除某些信息。
   - **查询/检索**：Agent可以从短期记忆中查询和检索相关信息，以便更好地理解和回应用户的请求。
3. **短期记忆与记忆提取模块的交互**

   消息数量/会话空闲时间聚合：短期记忆中的信息会根据消息数量或会话空闲时间进行聚合。当满足一定的条件（如达到一定数量的消息或会话空闲时间超过某个阈值）时，短期记忆中的信息会被传递给记忆提取模块。
4. **记忆提取模块**
   - **Embedding（向量嵌入）**：记忆提取模块首先对聚合后的信息进行嵌入处理，将其转换为向量表示。这些向量能够捕捉信息的语义和结构特征。

 版权所有 © 华为云计算技术有限公司

- **Consolidation（巩固）**：嵌入后的向量会被进一步巩固，可能包括降维、特征提取等操作，以生成更加紧凑和有效的表示。
- **Extraction（抽取）**：从巩固后的向量中抽取关键信息，生成最终的记忆表示。这些记忆表示能够反映用户对话中的重要信息和上下文。

5. **向量数据库**

   记忆提取模块生成的记忆表示会被存储到向量数据库中。向量数据库用于长期存储和管理这些记忆表示，以便后续的查询和使用。

记忆策略说明

- 语义记忆策略：从对话中提取通用事实和知识，支持语义相似度检索，提高信息的准确性和相关性。
- 用户偏好策略：提取并持久化用户偏好信息（常用地点、音乐风格等），实现个性化用户体验，提升用户满意度。
- 会话摘要策略：自动对原始会话进行压缩，保留关键上下文，优化上下文Token用量，提高处理效率和响应速度。
- 情景记忆策略：抽取情景片段并生成反思洞察，支持经验学习和复用，提升系统的智能性和适应性。

4.2.2 创建记忆库

创建记忆库用于存储和管理用户信息、历史交互和偏好，以便智能体能够提供更个性化、高效和情境适应的响应。

支持在控制台上通过配置长期记忆提取策略、短期记忆等参数来创建记忆库。

创建记忆库

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击左上角“记忆库”页签，进入记忆库界面。

步骤3 单击右上角“创建记忆库”。

步骤4 在“创建记忆库”的页面中，输入记忆库的配置信息，参数说明请参考**表4-3**。

表 4-3 参数说明

参数		说明
基本信息	记忆库名称	记忆库的名称，同一账号下记忆库名称不可重复。 命名规则：包含a-z、A-Z、0-9 和 (中划线)等有效字符，最多可包含 48 个字符。
	描述	用于对记忆库内容和用途的简要说明。 命名规则：长度不大于100个字符。

参数		说明
高级配置	私网访问	记忆库能够连接到内部的私有网络，访问内部的资源和服务。开启后，私有网络、子网暂时不支持修改。 选择已配置的VPC和子网，如未配置，请单击“新建私有网络”或“新建子网”并参考创建虚拟私有云和子网进行配置，配置后可单击刷新。
	公网访问	开启公网访问后，存在一定安全风险，建议仅用于开发调测场景。开启后，暂不支持关闭。
身份认证与鉴权	API Key	系统默认为“API Key”，通过接口的访问密钥进行身份验证和授权。 系统会自动为记忆创建和绑定API Key，可通过记忆详情页查看。
短期记忆	短期记忆（原始事件）过期	存储智能体在当前任务或会话中的临时信息和状态，通常在任务完成后清空。 用于设置短期（原始事件）记忆持续时间，超过设置的持续时间的事件过期后不再存储。时间设置可设置为7～365天之间。
长期记忆提取策略	总结 summary	勾选后，将互动的大量详细信息或数据压缩成关键上下文和关键洞察信息，帮助智能体更高效地检索和利用长期记忆中的信息。如会议总结、年度工作总结等。
	语义记忆 Semantic Memory	勾选后，使用与上下文无关的格式从原始对话中提取一般的事实性知识、概念和含义，帮助智能体理解和处理抽象信息。如个人信息、专业信息、教育经历等。
	用户偏好 User Preference	勾选后，存储和管理用户在与智能体交互过程中表现出的偏好、习惯和个性化需求，用户偏好帮助智能体更好地理解和适应用户的需求。如个人爱好、运动习惯等。
	情景记忆 Episodic Memory	勾选后，保留上下文的格式，从原始对话或经历中提取具有时空序列的具体事件、个人体验和场景细节，帮助智能体记住具体的事件和经历，支持其在类似情境下做出更准确的决策和响应。如旅游中的个人体验、看电影中的场景等。

参数		说明
自定义策略（可选）	自定义策略	用户可通过选择基础模型和定义提示模板，灵活地处理和生成特定类型的记忆内容。 单击“创建自定义策略”。 ● 策略名称：自定义策略名称。命名规则：包含a-z、A-Z、0-9 和 (下划线)等有效字符，最多可包含 48 个字符。 ● 策略类型：选择策略类型，然后配置抽取（Extraction）、整合（Consolidation）和reflection的提示词模板。请根据页面展示的实际内容进行配置。 – 抽取 Extraction：从大量数据中提取关键信息和重要特征，可以使用默认用户提示或创建自定义提示。 – 整合 Consolidation：将提取出的信息进行汇总、合并和优化，确保信息的完整性和一致性，可以使用默认用户提示或创建自定义提示。 – reflection：系统对过去的经验、行为或数据进行分析和总结，以提取有价值的信息和洞察的过程。
长期记忆提取触发条件	空闲时间	在没有接收到新任务或用户交互的一段时间后，触发长期记忆的提取。 请输入具体的时间，单位为秒。
	最大累计token数量	在处理用户输入时，累计的token数量达到预设的最大值时，触发长期记忆的提取。 请输入具体的token数量。
	最大累计消息数	在处理用户输入时，累计的消息数量达到预设的最大值时，触发长期记忆的提取。 请输入具体的消息数。
高级配置	标签	可选参数。 由“标签键”和“标签值”组成，用于标识和分类云服务资源。 单击“添加标签”，可添加一个或多个标签，最多添加20个标签。

步骤5 单击“立即创建”，在“API Key”弹框中，请复制或下载API Key，以便在SDK客户端进行鉴权使用。

API Key只显示一次请妥善保存。

步骤6 单击“继续创建”，则记忆库创建成功，即可在记忆库列表展示已创建的记忆库。

----结束

后续操作

创建记忆库成功后，您可以在Agent中集成记忆库，以保留上下文、知识和用户偏好，实现跨会话的持久化。详细操作请参见在智能体中集成记忆库。

更多操作

您还可以对已创建的记忆库执行如下操作。

表 4-4 更多操作

操作	说明
查看并修改记忆库	1. 进入记忆库界面。 2. 在记忆库列表中单击记忆库名称，查看记忆库详情。 3. 在“记忆库详情”页签，支持修改短期记忆持续时间、为记忆库新增内置策略或新增自定义策略、修改长期记忆触发条件等。
删除记忆库	记忆库删除后不可恢复，请谨慎操作。 1. 进入记忆库界面。 2. 单击记忆库列表操作列的“删除”。 3. 在弹框中确认要删除后一键输入“DELETE”，然后单击“确定”。

4.2.3 记忆检索

从记忆库中快速检索已有的记忆，帮助智能体迅速获取用户的历史信息、偏好和过往交互。

前提条件

已创建记忆库并已集成记忆库。

记忆检索

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“记忆库”页签，进入记忆库界面。

步骤3 在记忆库列表中单击已创建的记忆库名称。

步骤4 在“记忆检索”页签，参考**表4-5**配置检索条件。

表 4-5 参数说明

参数	说明
用户ID（可选）	输入要检索记忆的用户ID。
会话ID（可选）	输入要检索记忆的会话ID。
记忆策略类型（可选）	选择要检索的记忆策略类型。
记忆策略名称（可选）	选择要检索的记忆策略名称。
日期范围（可选）	选择要检索记忆的日期范围。

参数	说明
返回结果数量	检索后返回的结果数量，输入值必须在1到100之间。
查询内容（可选）	输入要检索的内容。

步骤5 单击“确认”后，在搜索框中输入需要检索的问题，平台将返回与该问题相关的文本内容。

----结束

4.2.4 在智能体中集成记忆库

在智能体中集成记忆库，实现智能体与记忆库的快速对接。

前提条件

- 已创建记忆库，并获取记忆库的api_key。
- 已安装Python，且版本不低于3.10。
- 已安装AgentRunSDK：

```
pip install AgentRunSDK
```

在智能体中集成记忆库（操作步骤）

步骤1 在左侧导航栏选择“开发中心 > 组件库 ”，单击“记忆库”页签，进入记忆库界面。

步骤2 在记忆库列表单击操作列的“调用代码”，然后根据开发框架获取环境变量信息。

步骤3 配置环境变量。

```
# 创建space后，会返回API_KEY，请将其设置为环境变量，用于调用数据面接口时鉴权
export AGENTARTS_MEMORY_API_KEY="你的数据面API密钥"
```

步骤4 集成记忆库。

```
import os
import time

# 确保环境变量已设置
assert os.getenv('AGENTARTS_MEMORY_API_KEY'), "请设置 AGENTARTS_MEMORY_API_KEY"

def client_mode_example():
    """Client模式完整示例"""
    print("=== Client模式完整示例 ===")
        # 2. 创建会话
        print("\n2. 创建会话")
        session_data = client.create_memory_session(
            space_id=space_id,
            actor_id="client-test-user",
            assistant_id="client-test-assistant"
        )
        session_id = session_data.id
        print(f"✓ Session创建成功: {session_id}")

        # 3. 发送消息（使用TextMessage对象）
        print("\n3. 发送对话消息")
        messages = [
            TextMessage(
                role="user",
                content="你好，我想了解电商推荐算法，能够根据用户行为进行个性化推荐的算法有哪些？",
```

```
            actor_id="client-test-user"
        ),
        TextMessage(
            role="assistant",
            content="电商推荐算法主要包括：1) 协同过滤算法，基于用户行为相似性推荐；2) 基于内容的推荐，分析商品特征；3) 深度学习算法，能更准确捕捉用户偏好。推荐组合使用多种算法提升准确率。",
            actor_id="client-test-assistant"
        ),
        TextMessage(
            role="user",
            content="我对机器学习的监督学习特别感兴趣，深度学习和传统机器学习有什么区别？",
            actor_id="client-test-user"
        )
    ]
    add_result = client.add_messages(
        space_id=space_id,
        session_id=session_id,
        messages=messages
    )
    print(f"✓ 已添加 {len(add_result.items)} 条消息")

    # 4. 等待记忆系统处理消息
    print("\n4. 等待记忆系统处理消息...")
    time.sleep(30)
    # 5. 列出记忆
    print("\n5. 查询记忆列表")
    memories = client.list_memories(
        space_id=space_id,
        limit=10
    )
    print(f"✓ 发现 {len(memories.items)} 条记忆")
    for i, memory in enumerate(memories.items[:3]):
        print(f"  {i+1}. {memory.content[:50]}...")
        print(f"     策略: {memory.strategy_type}")
    # 6. 搜索记忆（使用MemorySearchFilter对象）
    print("\n6. 搜索相关记忆")
    from hw_agentrun_wrapper.memory.inner.config import MemorySearchFilter
    search_results = client.search_memories(
        space_id=space_id,
        filters=MemorySearchFilter(query="机器学习", top_k=3)
    )
    print(f"✓ 找到 {len(search_results.results)} 条相关记忆")
    for i, result in enumerate(search_results.results):
        score = result.get('score', 0)
        content = result.get('record', {}).get('content', '')[:60]
        print(f"  {i+1}. [{score:.2f}] {content}...")

    # 7. 获取特定记忆详情
    if memories.items:
        print("\n7. 获取记忆详情")
        memory_id = memories.items[0].id
        memory_detail = client.get_memory(space_id, memory_id)
        print(f"✓ 记忆ID: {memory_detail.id}")
        print(f"   内容: {memory_detail.content[:80]}...")
        print(f"   策略: {memory_detail.strategy_type}")
    return space_id, session_id
```

----结束

4.3 网关

4.3.1 网关介绍

什么是网关

网关是连接AgentArts与外部系统/服务的统一接口层，负责数据交互与安全访问控制。通过提供标准化入口，实现AI能力与存量系统的快速对接，并支持多协议、多认证方式的集成。

使用优势：

- 通过API自动转换能力，可快速将企业内部系统对接至AgentArts，减少开发工作。
- 多层身份认证（如APIKey、OAuth认证）与网络隔离（公网/私网访问配置）保障数据访问安全。

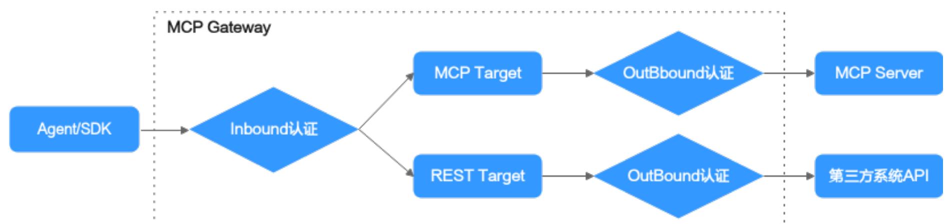
应用场景

应用场景：一站式集成企业异构API资产，将存量REST转换为AI Agent可感知的标准MCP工具，也为SSE及HTTP Streamable传输方式的MCP服务提供安全可靠的访问入口，实现跨部门数据的语义化调用。

使用价值：降低企业存量系统AI转型成本，通过统一鉴权与资产自动转换，保障数据主权安全，实现企业私有能力‘零改造’接入大模型生态。

工作原理

图 4-5 网关原理图



网关的使用原理如下：

1. 请求发起

   Agent/SDK：应用程序或客户端通过Agent或SDK发起请求，并将请求发送给MCP Gateway（网关）。
2. Inbound认证：MCP Gateway接收到请求后，首先进行Inbound认证，确保请求的合法性。只有通过认证的请求才能继续进行处理。
3. 请求分发与处理
   - MCP Target：
     - MCP Target负责处理与MCP Server相关的请求。
     - 处理完请求后，MCP Target会进行Outbound认证，确保请求的合法性。
     - 通过Outbound认证后，请求会被发送到MCP Server。

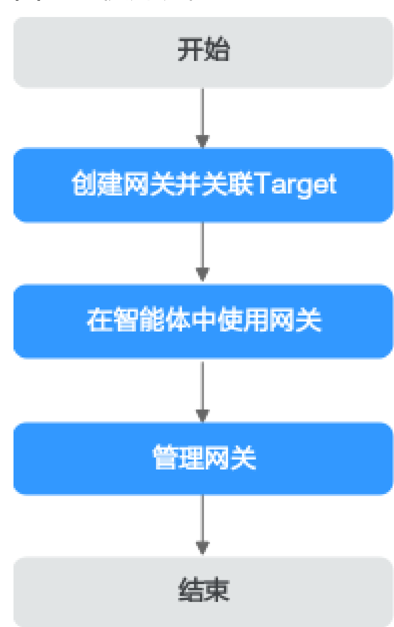
– REST Target：
  - REST Target负责处理与第三方系统API相关的请求。
  - 处理完请求后，REST Target会进行Outbound认证，确保请求的合法性。
  - 通过Outbound认证后，请求会被发送到第三方系统API。

Target是网关对接的下游服务，当前支持的Target类型有：

- MCP Server：直接转发请求到自研或公开MCP服务。
- REST API：通过上传OpenAPI定义文件转换为MCP服务。

使用流程

图 4-6 使用流程



1. 创建网关并关联Target：网关作为智能体与外部系统之间的通信中介，负责双向通信和协议转换，绑定的Target用于定义和配置一个目标端点或服务，网关将接收到的请求转发到该服务进行处理。详细请参见**创建网关**、**在网关中创建Target**。
2. 在智能体中使用网关：创建网关后，需要将已经创建的网关集成到智能体的代码中或添加到已有的MCP智能体客户端，以便后续调用网关能力。详细请参见**在智能体中使用网关**
3. 管理网关：在网关详情页面，您可以查看网关在使用过程中产生的日志。详细请参见**查看网关日志**。

4.3.2 创建网关

网关作为智能体与外部系统之间的通信中介，不仅负责双向通信和协议转换，还提供了日志记录的功能，确保系统的安全、高效和可扩展性。

创建网关有两种方式：通过控制台创建和通过SDK创建。如果您选择使用SDK创建网关，可以参考**MCP Gateway SDK**，本章节主要介绍通过控制台创建网关。

版权所有 © 华为云计算技术有限公司

前提条件

- 在创建或删除网关时，请确保用户除了具备网关的FullAccessPolicy权限外，还需拥有以下特定的身份策略权限：

表 4-6 授权项

操作	授权项	功能介绍
创建网关	iam:agencies:pass	授予向云服务传递委托的权限。
	vpc:nativePorts:create	授予原生API创建端口权限。
	vpc:routeTables:update	授予更新路由表权限。
	eip:publicIps:associateInstance（涉及公网访问时需要）	授予将弹性公网IP绑定网卡的权限。
删除网关	vpc:routeTables:update	授予更新路由表权限。
	vpc:nativePorts:delete	授予原生API删除端口权限。
	eip:publicIps:disassociateInstance（涉及公网访问时需要）	授予将弹性公网IP解绑网卡的权限。

- 创建网关时需要创建委托并授权，请确保传入的委托包含以下身份策略权限：

表 4-7 授权项

授权项	功能介绍
csms:secret:getVersion	授予查询指定凭据版本的信息和其明文凭据值的权限。
agentIdentity::getResourceApiKey	授予获取与API密钥凭证提供者关联的API密钥的权限。
agentIdentity::getResourceOauth2Token	授予通过OAuth2两方/三方授权流程获取访问令牌，以访问外部资源的权限。
agentIdentity::getResourceStsToken	授予从STS凭证提供者获取IAM临时凭证的权限。

- 创建网关需要进行入站认证，入站认证功能需要用户确保有网关的FullAccessPolicy外，还包含如下身份策略权限：

授权项	功能介绍
csms:secret:getVersion	授予查询指定凭据版本的信息和其明文凭据值的权限

- 如出站网络使用VPC网络访问，请确保已开通虚拟私有云，并已创建私有网络、子网、安全组。详细操作请参见创建虚拟私有云和子网，创建安全组。并确保用户拥有网关的FullAccessPolicy外，还需拥有如下身份策略权限：

表 4-8 授权项

授权项	功能介绍
vpcep:endpoints:create	授予指定服务创建VPC终端节点的权限
vpcep:endpoints:delete	授予权限删除终端节点

- 管理员可在IAM控制台为创建的IAM用户授予权限，授权后，用户即可根据权限使用账号中的云服务资源，为IAM用户授权的操作指导可参考：**给IAM用户授权**。

约束与限制

一个租户最多可创建10个网关。

创建网关

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“网关”页签，进入网关界面。

步骤3 单击右上角“创建网关”。

步骤4 在“创建网关”的页面中，输入网关的配置信息，参数说明请参考**表4-9**。

表 4-9 参数说明

<table>
<tr><th colspan="2">参数</th><th>说明</th></tr>
<tr><td rowspan="2">基本信息</td><td>名称</td><td>网关的名称，同一账号下网关名称不可重复。
命名规则：
● 以字母或数字开头和结尾。
● 支持大小写字母、数字和连字符。
● 最大长度为40个字符。</td></tr>
<tr><td>描述</td><td>用于对网关内容和用途的简要说明。
命名规则：长度不大于100个字符。</td></tr>
<tr><td>权限与身份认证</td><td>委托</td><td>授予的代理权限或代理功能，允许代表智能体与外部系统进行通信和交互。
选择IAM中已创建的委托，如未创建，请单击“创建委托”，具体请参考创建委托。</td></tr>
</table>

参数		说明
	入站身份认证	配置网关的身份认证方式。入站身份认证是指网关在接收和处理来自外部系统的请求时，对请求进行身份验证，确保只有经过验证的请求才能进入网关，提高系统的安全性和可靠性。 支持以下认证方式： ● IAM认证：使用登录管理控制台时使用的IAM用户名进行认证。 ● OAuth 2.0认证：将OAuth 2.0配置为入站身份认证。选择“OAuth 2.0认证”后，相关参数配置请参考**表4-10**。 ● API Key认证：通过接口的访问密钥进行身份验证和授权。系统会自动生成API Key的名称，可根据需求自定义修改。
可观测配置	日志记录	● 未开启，新建网关及网关使用过程产生的日志无法上报至云日志服务。 ● 开启后，新建网关及网关使用过程产生的日志会报上包云日志服务(LTS)。
关联Target	创建Target	单击“创建Target”按钮完成Target创建后，网关将与新创建的Target关联。 创建Target详情请参考**在网关中创建Target**。
高级配置	私网访问	连接内部的私有网络，访问内部的资源和服务。 ● 选择已配置的VPC，如未配置，请单击“新建VPC”，具体请参考**创建虚拟私有云和子网**。 ● 选择已配置的子网，如未配置，请单击“新建子网”，具体请参考**创建虚拟私有云和子网**。 ● 选择已配置的安全组，如未配置，请单击“新建安全组”，具体请参考**创建安全组**。 配置后可单击刷新。
	公网访问	开启公网访问

表 4-10 OAuth 2.0 认证参数配置说明

参数	说明
Discovery URL	输入身份提供商（例如Okta、Cognito等）提供的 Discovery URL，该URL通常可在该提供商的文档中找到。
JWT授权配置	
允许的受众	用于认证为OAuth 2.0指定的受众是否在Agent Identity运行中指定的受众匹配或为其子集。
允许的客户端	用于认证为OAuth 2.0指定的客户端标识符是否被允许访问Agent Identity。

参数	说明
允许的范围	仅当令牌包含此处配置的至少一个必须范围时才允许访问自定义声明。
自定义声明	仅当令牌中的特定声明与预定义字符串值相匹配时才允许访问。

步骤5 单击“确定”。即可在网关列表中展示已创建的网关，单击网关名称，查看网关详情。

----结束

后续操作

创建网关后，您需要将已经创建的网关集成到智能体的代码中或添加到已有的MCP智能体客户端，以便后续调用网关能力。详细操作请参见在智能体中使用网关。

更多操作

您还可以对已创建的网关执行如下操作。

表 4-11 更多操作

操作	说明
查看生成的域名	1. 在网关列表中单击网关名称。 2. 在“基本信息”页签中的“基本信息”区域，查看MCP服务域名。
修改网关基本信息	1. 在网关列表中单击网关名称。 2. 在“基本信息”页签修改网关的基本信息，其中置灰项不可修改。
删除网关	网关删除后不可恢复，请谨慎操作。且删除网关时，必须先删除所有关联的Target，才能成功删除网关。 1. 单击网关列表操作列的“删除”。 2. 在弹框中单击“确定”。

4.3.3 在网关中创建 Target

创建网关时，需要创建并绑定Target。创建Target用于定义和配置一个目标端点或服务，网关将接收到的请求转发到该服务进行处理。

前提条件

- 已创建网关。
- 创建Target时，若选择OPEN API类型的目标服务类型，且需要从OBS获取规范文档时，用户需要确保有网关的FullAccessPolicy外，还包含如下身份策略权限：

表 4-12 授权项

授权项	功能介绍
obs:bucket:listBucket	授予获取桶内对象列表的权限
obs:bucket:listBucketVersions	授予获取桶内多版本对象列表的权限
obs:bucket:listAllMyBuckets	授予查询创建的桶列表的权限
obs:object:getObject	授予下载非指定版本对象的权限
obs:object:getObjectVersion	授予下载指定版本对象的权限

在IAM控制台为用户授权的操作指导可参考：**给IAM用户授权**。

约束与限制

一个网关最多可关联10个Target。

创建 Target

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击左上角“网关”页签，进入网关界面。

步骤3 选择已创建的网关，单击网关名称，在“关联Target”区域，单击“创建Target”。

步骤4 输入Target的配置信息，参数说明请参考**表4-13**，然后单击“确定”。

表 4-13 参数说明

参数		说明
基本信息	名称	必选参数。 Target的名称，同一网关下Target名称不可重复。 命名规则： ● 以字母或数字开头和结尾。 ● 支持大小写字母、数字和连字符。 ● 最大长度为50个字符。
	描述	可选参数。 用于对Target内容和用途的简要说明。 命名规则：长度不大于200个字符。

参数		说明
配置信息	类型	选择Target的类型。 ● REST API：使用OPEN API schema文件调用。 – 使用“OBS”进行OAS文件导入。 使用OpenAPI Specification (OAS) 文件来定义和配置目标端点或服务，并且这些文件存储在对象存储服务（Object Storage Service, OBS）中，并在创建Target时从OBS中导入这些文件。 单击“从OBS中选择”，选择在OBS中创建的桶列表，直至选择到桶中的.json文件后单击“确定”。 – 使用“自定义”进行OAS文件导入。 从本地文件系统中选择并上传OAS文件，用于定义和配置目标端点或服务。 ■ 单击“从本地导入”，浏览本地文件系统，选择OAS文件并上传。上传后json文件或yaml即可展示在“内联编辑器”中。 ■ 单击“从OBS中导入”，选择在OBS中创建的桶列表，直至选择到桶中的.json文件后单击“确定”，将OBS中的文件下载下来展示到内联编辑器中。 ● MCP：使用已有的MCP服务器调用。 – 传输方式 ■ SSE：基于HTTP的协议，用于从服务器向客户端推送实时更新，适用于需要实时更新的应用。 ■ HTTP Streamable：基于HTTP的流式传输方式，允许服务器向客户端发送连续的数据流，适用于需要实时数据流的应用。 – MCP地址：输入MCP地址，如：https://mcp.example.com/sse。

参数		说明
出站身份认证	出站身份认证	配置出站身份认证方式，用于在网关将请求转发到后端服务（Target）时，对请求进行身份验证和授权。 支持以下认证方式： ● API Key – 选择出站身份，如下拉框无选项请单击“创建出站身份”。 – 选择位置。 ■ 标头：在HTTP请求的头部字段中传递API Key。 ■ 查询参数：在HTTP请求的URL中传递API Key。 – 参数名称（可选）：输入参数名称，在请求中传递API的密钥时使用的具体名称。 – 前缀（可选）：在请求头中传递API Key时，API密钥值之前的前缀部分，如在授权标头中使用的“Bearer”。 ● OAuth – 选择出站身份，如下拉框无选项请单击“创建出站身份”。 – 选择授权类型。 客户端授权：适用于用户授权应用程序（客户端）访问其资源。 – 作用域：用于定义可以访问的资源和操作的范围，确保只能访问其被授权的资源。输入范围名称，最大可添加100个范围。 – 自定义参数：在请求中传递的额外参数，可以用于传递特定的信息或满足特定的认证需求。请输入自定义参数信息的键和值，最大可添加255个参数。 ● 无认证：在网关将请求转发到Target时，不对请求进行身份验证和授权。

----结束

更多操作

您还可以对已创建的Target执行如下操作。

表 4-14 更多操作

操作	说明
查看Target	1. 在网关详情页面的“关联Target”区域。 2. 单击Target名称，查看Target详情。

操作	说明
修改Target	1. 在网关详情页面的“关联Target”区域。 2. 单击Target列表操作列的“编辑”。 3. 修改Target信息，其中置灰项不可修改。
删除Target	Target删除后不可恢复，请谨慎操作。 1. 在网关详情页面的“关联Target”区域。 2. 单击Target列表操作列的“删除”。 3. 在弹框中单击“确定”。

4.3.4 在智能体中使用网关

创建网关后，您需要将已经创建的网关集成到智能体的代码中或添加到已有的MCP智能体客户端，以便后续调用网关能力，可以参考以下示例代码，并根据需求自定义修改其中的信息，生成实际使用的代码。

前提条件

已创建网关。

在智能体中使用网关

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“网关”页签，进入网关界面。

步骤3 在网关列表单击操作列的“调用代码”，查看并复制代码示例，并根据需求自定义修改其中的信息，生成实际使用的代码。

您可以在请求体中指定tools/list作为请求方法获取网关提供的所有可用工具。

您可以在请求体中指定tools/call作为请求方法用于调用特定的工具。

tools/list调用示例如下：

```
import requests
import json


def list_tools(gateway_url, access_token):
    headers = {
        "Content-Type": "application/json",
        "mcp-session-id": "c3fb3b87-dcc6-asda-asda-458962457896", # 相同的sessionid会使用同一个sandbox
        "Authorization": f"Bearer {access_token}"
    }
    payload = {
        "jsonrpc": "2.0",
        "id": "list-tools-request",
        "method": "tools/list"
    }
    return requests.post(gateway_url, headers=headers, json=payload, verify=False) #verify=False忽略ssl校验

# Example usage
gateway_url = "https://xxx/mcp" # Replace with your actual gateway endpoint
access_token = "xxx" # 如果是APIKey的认证，这里换成APIKey的值，无认证可以忽略
```

```
response = list_tools(gateway_url, access_token)
print(json.dumps(response.json(), indent=2))
```

tools/call调用示例如下：

```
import requests
import json


def call_tool(gateway_url, access_token, tool_name, arguments):
    headers = {
        "Content-Type": "application/json",
        "mcp-session-id": "c3fb3b87-dcc6-asda-asda-458962457896", # 相同的sessionid会使用同一个sandbox
        "Authorization": f"Bearer {access_token}"
    }
    payload = {
        "jsonrpc": "2.0",
        "id": "call-tool-request",
        "method": "tools/call",
        "params": {
            "name": tool_name,
            "arguments": arguments
        }
    }
    return requests.post(gateway_url, headers=headers, json=payload, verify=False) #verify=False忽略ssl校验

# Example usage
gateway_url = "https://xxx/mcp" # 从创建的网关详情中获取
access_token = "xxx" # 如果是APIKey的认证，这里换成APIKey的值，无认证可以忽略
tool_name = "gaode-api___getWeatherInfo"  # 替换成tool的名称<{TargetId}__{ToolName}>，可以根据tools/
list进行查看
arguments = {"city": "320100", "extensions": "all"}
response = call_tool(gateway_url, access_token, tool_name, arguments)
print(json.dumps(response.json(), indent=2))
```

----结束

4.3.5 查看网关日志

网关创建完成后，在“网关”界面，您可以通过网关状态或搜索（按网关名称或ID）功能来查找网关，在网关详情页面，您可以查看网关在使用过程中产生的日志。

查看网关日志

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“网关”页签，进入网关界面。

步骤3 单击网关名称，查看网关详情。

步骤4 在“日志”页签，查看网关使用过程中产生的日志。

如未启用日志功能，可在“日志”页签单击“去启用”开启日志记录。

----结束

5 部署智能体运行时

5.1 智能体运行时介绍

概述

智能体运行时是一个安全隔离、框架无关的托管执行环境，专为生产级智能体设计，提供企业级的弹性伸缩与权限治理能力，帮助开发者专注于业务逻辑，无需关心底层基础设施的运维。

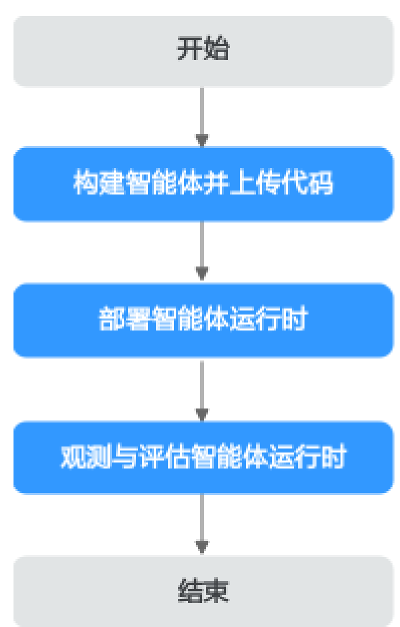
使用优势

- 安全沙箱：基于容器化技术实现进程级隔离，提供企业级的运行时可靠性及安全保障。
- 弹性伸缩：支持基于业务流量（QPS/并发数）的自动扩缩容能力，从容应对流量洪峰。
- 细粒度权限：深度集成身份与访问管理服务，支持身份认证、灵活的网络配置及安全委托能力，保证智能体的权限合规。

版权所有 © 华为云计算技术有限公司

使用流程

图 5-1 使用流程



1. 构建智能体并上传代码：支持基于主流的Agent框架开发的智能体部署，在部署前您需要将智能体打包为镜像后上传至容器镜像服务（SWR）。
2. 创建智能体运行时：指定镜像，并配置安全访问等参数信息，即可将智能体托管至运行时。
3. 观测与评估智能体运行时：支持日志监控，以实时洞悉您的智能体性能、准确性和可靠。

5.2 部署智能体运行时

将本地构建的高代码智能体部署至运行时上，为智能体提供一个安全隔离、与框架和模型无关的托管运行环境，确保智能体在独立、受控的条件下运行。

本文为您介绍如何在控制台上创建智能体运行时，定义智能体的基本信息、来源方式、权限与访问控制等。

前提条件

- 创建运行时需要创建委托并授权给安全沙箱服务，请确保创建的委托名称为AgentArtsRuntimeDeploymentAgency，且委托包含以下身份策略权限：

表 5-1 授权项

授权项	功能介绍
swr:repo:download	授予共享版仓库下载镜像的权限。
swr::createAuthorizationToken	授予共享版仓库生成新的临时登录指令的权限。

版权所有 © 华为云计算技术有限公司

授权项	功能介绍
sts::createServiceBearerToken	授予权限获取一个绑定至某服务的Bearer Token。

- 创建运行时需要进行入站认证，入站认证功能需要为智能体网络创建服务关联委托，该服务关联委托包含以下策略权限：

授权项	功能介绍
agentIdentity::getAuthorizerConfiguration	授予查询工作负载身份授权配置的权限。
agentIdentity::createWorkloadAccessToken	授予获取代理工作负载访问令牌的权限。
agentIdentity::createWorkloadAccessTokenForJWT	授予使用JWT令牌获取代表用户操作的代理工作负载访问令牌的权限。
agentIdentity::createWorkloadAccessTokenForUserId	授予使用用户ID获取代表用户操作的代理工作负载访问令牌的权限。
csms:secret:getVersion	授予查询指定凭据版本的信息和其明文凭据值的权限。
kms:cmk:decryptDataKey	授予解密数据密钥的权限。

- 如果需要在运行时通过身份提供商获取出站凭据访问外部服务，请确保传入的委托包含以下身份策略权限：

授权项	功能介绍
agentIdentity::getResourceApiKey	授予获取与API密钥凭证提供者关联的API密钥的权限。
agentIdentity::getResourceOauth2Token	授予通过OAuth2两方/三方授权流程获取访问令牌，以访问外部资源的权限。
agentIdentity::getResourceStsToken	授予从STS凭证提供者获取IAM临时凭证的权限。

- 如果需要在运行时上报AOM，APM可观测性指标，请确保传入的委托包含以下身份策略权限：

授权项	功能介绍
apm:application:get	授予获取应用信息数据的权限。
aom:metric:list	授予权限以查询普罗实例。
aom:icmgr:get	授予权限以获取采集组件版本信息。

- 如出站网络使用VPC网络访问，请确保已开通虚拟私有云，并已创建私有网络、子网、安全组。详细操作请参见创建虚拟私有云和子网，创建安全组。
- 管理员可在IAM控制台为创建的IAM用户授予权限，授权后，用户即可根据权限使用账号中的云服务资源，为IAM用户授权的操作指导可参考：**给IAM用户授权**。

约束与限制

- 最多可创建运行时的数量为1000个。
- 单个运行时可以创建的版本数量为1000个。
- 单个运行时可以创建的访问方式数量为10个。

托管智能体

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“部署运行 > 智能体运行时 ”。

步骤3 单击“托管智能体”，输入相关配置信息，具体参数请参考**表5-2**，配置完成后单击“确定”。

表 5-2 参数说明

参数		说明
基本信息	名称	智能体运行时的名称，同一账号下名称不可重复。 命名规则： ● 以小写字母开头，小写字母或数字结尾。 ● 支持数字、小写字母、中划线。 ● 最大长度为63个字符。
	描述	用于对智能体运行时内容和用途的简要说明。 命名规则：长度不大于4096个字符。
来源方式	镜像	指定一个包含智能体应用程序及其所有依赖项的容器镜像。 单击“选择镜像”，在“我的镜像”、“共享镜像”或“镜像中心”下选择需要的镜像。 在“镜像选择”页面中，列表数据来源于在容器镜像服务已上传的镜像。
权限与访问控制	委托	授予的代理权限或代理功能，允许代表智能体与外部系统进行通信和交互。 选择IAM中已创建的委托，如未创建，请单击“创建委托”，具体请参考创建委托。
	入栈协议	选择入栈协议，定义智能体接收和处理外部请求的方式。

版权所有 © 华为云计算技术有限公司

参数		说明
	入站身份认证	配置入站身份认证方式。在智能体接收到请求时，对请求进行身份验证和授权，确保请求来自可信的来源，并且具有访问特定资源的权限。 支持以下认证方式： ● IAM认证：使用登录管理控制台时使用的IAM用户名进行认证。 ● OAuth 2.0认证：将OAuth 2.0配置为入站身份认证。选择“OAuth 2.0认证”后，相关参数配置请参考**表4-10**。 ● API Key 认证：通过接口的访问密钥进行身份验证和授权。输入API Key的名称后，为智能体运行时创建和绑定API Key。
可观测配置	日志记录	● 未开启，托管运行时过程产生的日志无法上报至云日志服务。 ● 开启后，托管运行时过程产生的日志会报上包云日志服务(LTS)。

高级配置	出网网络配置	配置智能体运行时的网络访问方式。
		● 公网访问：智能体可以访问互联网上的公共资源。
		● 私网访问：连接到内部的私有网络，访问内部的资源和服务。
		– 选择已配置的VPC，如未配置，请单击“新建VPC”，具体请参考创建虚拟私有云和子网。
		– 选择已配置的子网，如未配置，请单击“新建子网”，具体请参考创建虚拟私有云和子网。
		– 选择已配置的安全组，如未配置，请单击“新建安全组”，具体请参考创建安全组。
		配置后单击刷新。
		● 启动命令（可选）：智能体启动时需要执行的命令。
		● 监听端口：智能体在运行时监听的网络端口，用于接收外部的网络请求。
		● 环境变量：智能体运行时传递给应用程序的变量。
		– 表单编辑：通过键值对方式配置环境变量。单击“添加新变量”，设置键和值。
		– JSON格式编辑：通过JSON文件方式配置环境变量。JSON文件格式要求如下：
		{
		"键": "值",
		"Path": "/dir/xxx"
		}

----结束

相关操作

支持从运行时、工具、记忆库和网关等多维度监控全链路的关键指标，实时洞察您的Agent性能、准确性和可靠性。详细操作请参加观测智能体。

更多操作

智能体创建完成后，您还可以执行如下操作。

表 5-3 更多操作

操作	说明
查看智能体运行时基本信息	在智能体运行时页面，单击运行时名称，在“基本信息”页签，查看运行时的基本信息、访问方式及高级配置。
修改智能体运行时	1. 单击智能体运行时列表操作列的“编辑”。 2. 修改运行时的基本信息，权限与访问控制等，其中置灰项不可修改。
删除智能体运行时	运行时删除后不可恢复，请谨慎操作。 1. 单击智能体运行时列表操作列的“删除”。 2. 在弹框中单击“确定”。

相关文档

智能体运行时创建成功后，您还可以在Agent中集成以下功能：

在智能体中集成工具：将工具集成到您的Agent代码中，使Agent能够调用隔离环境来完成代码运行、浏览器操作等复杂任务。

在智能体中集成记忆库：在智能体中集成记忆库，实现智能体与记忆库的快速对接。

在智能体中使用网关：将网关集成到智能体的代码中或添加到已有的MCP智能体客户端，以便后续调用网关能力，

集成后，将开发好的Agent打包上传至镜像仓库，然后更新智能体运行时。

5.3 管理智能体运行时

智能体运行时创建完成后，您可以管理并查看运行时过程中的日志以及查看智能体运行时版本。

查看运行时日志

步骤1 登录**AgentArts智能体开发平台**。

步骤2 在左侧导航栏选择“部署运行 > 智能体运行时 ”。

步骤3 单击智能体运行时名称，查看智能体运行时详情。

步骤4 在“日志”页签，查看智能体运行时运行过程产生的日志。

如未启用日志功能，可在“日志”页签单击“去启用”开启日志记录。

----结束

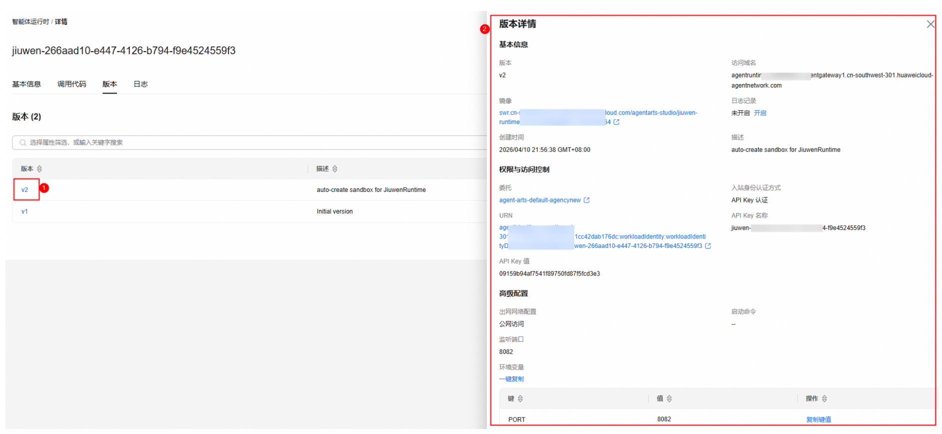
查看智能体运行时版本

当修改智能体运行时的基本信息、权限与访问控制，或者智能体代码更新并打包成新镜像重新上传后，均可进行运行时版本的更新。

步骤1 进入智能体运行时界面。

步骤2 单击智能体运行时名称，在“版本”页签，查看运行时的版本列表，单击版本号，查看版本详情。

图 5-2 版本详情



----结束

5.4 更新智能体运行时

智能体的代码修改后，可以重新打包成新的镜像，将新的镜像上传，确保智能体的更新过程有序、可控。

前提条件

已创建智能体运行时，具体操作请参见部署智能体运行时。

更新智能体运行时

步骤1 登录AgentArts智能体开发平台。

步骤2 在左侧导航栏选择“部署运行 > 智能体运行时 ”。

步骤3 在智能体运行时列表中，单击智能体运行时操作列的“编辑”。

步骤4 在“来源方式”区域，选择目标镜像。

步骤5 单击“确定”。

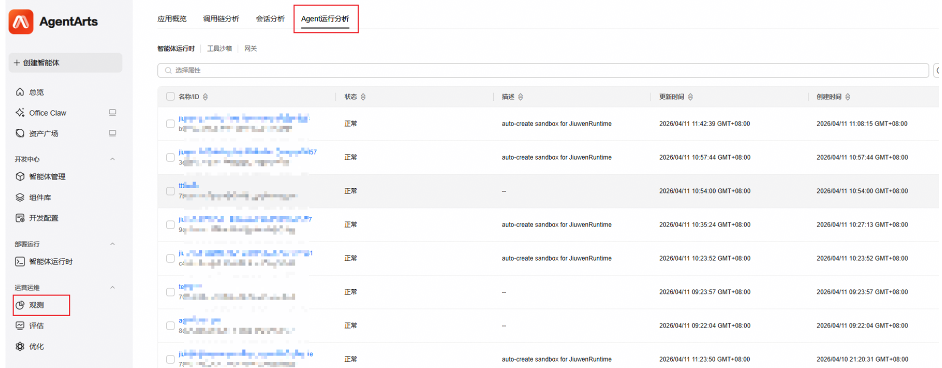
----结束

6 观测智能体

AgentArts智能体开发平台提供了完善的可观测性能力，支持将智能体运行过程中的对话数据自动上报至运营运维平台。通过分析这些数据，开发者可以实时监控智能体运行状态、排查故障。

在创建智能体运行时、网关、沙箱工具时开启日志，即可在“观测 > Agent运行分析”展示智能体运行时、网关、沙箱工具的日志。

图 6-1 观测智能体



版权所有 © 华为云计算技术有限公司

7 AgentRuntime SDK 参考

7.1 文档导读

本文档指导您如何安装和配置开发环境、构建和部署智能体。它提供从入门到进阶的全流程支持。

您可以通过参考**快速开始**章节快速掌握AgentArts SDK的基础操作，也可通过**示例：构建你的首个高码智能体**章节全面掌握如何使用沙箱工具、记忆库、网关全面增强Agent能力。本文也提供了**CLI命令行工具**，用于在终端中管理Agent的部署、调用、状态查询和资源销毁等操作。

章节	内容
SDK简介	简要介绍AgentRuntime SDK的概念、使用场景、SDK版本等信息。
SDK应用框架	介绍SDK框架的设计和架构，确保开发者能够高效、正确地使用工具。
CLI	介绍命令行工具的使用，用于在终端中管理Agent的部署、调用、状态查询和资源销毁等操作。
快速开始	介绍如何快速创建并部署一个支持问答能力的agent。介绍从环境配置、本地开发调试，到最终将Agent部署到生产环境的全过程。
Runtime SDK	介绍Agent运行时所需的所有功能，包括请求处理、响应生成、上下文管理等。
Tools SDK	以代码执行工具为例，介绍如何在智能体中集成工具。
Memory SDK	介绍如何集成云上记忆组件库。
Identity SDK	介绍如何集成华为云身份认证服务。
MCP Gateway SDK	介绍如何集成MCP网关的生命周期管理。

版权所有 © 华为云计算技术有限公司

7.2 SDK 简介

在开发智能代理应用时，开发者经常面临接口不统一、开发工具链缺失、最佳实践缺乏封装以及扩展性不足等问题，导致开发效率低下。AgentArts SDK通过以下方式解决上述痛点：

- **标准化接口**：提供统一的API规范，简化平台集成。
- **开发工具链**：内置CLI工具，支持快速创建、调试和部署。
- **最佳实践封装**：内置错误处理、日志记录、性能优化等最佳实践。
- **灵活扩展**：支持自定义中间件、插件和工具集成。

华为云AgentArts SDK是一个专为华为云AgentArts平台设计的面向开发者的Python开发工具包。该SDK支持使用Langchain/LangGraph、GoogleADK等开源高码Agent框架开发的智能体，实现一键部署到华为云，从而实现零基础设施管理及完全可控的云端部署。它通过本地代码开发Agent和能力增强（沙箱工具、记忆库、网关）的核心优势，帮助用户将大模型能力快速转化为可落地的业务智能体。

使用场景

1. 纯托管模式：您已经在本地使用LangChain、LangGraph等开源框架开发了一个成熟的Agent，目前只需要一个安全、高可用的生产环境将其运行起来。
2. 能力增强模式：您的Agent正在开发中，且需要使用复杂的记忆管理、安全的执行沙箱或对接企业内网API。您希望直接调用云上组件，拒绝重复造轮子。

SDK 版本说明

表1提供了AgentArts SDK支持的SDK列表，您可以在GitHub仓库查看SDK更新历史、获取安装包以及查看指导文档。

表 7-1 SDK 列表

编程语言	Github地址	参考文档
Python	*huaweicloud-agentarts-sdk-python*	*Python SDK开发指南*

支持的区域

西南-贵阳一（cn-southwest-2）

7.3 SDK 应用框架

框架概述

SDK应用框架基于Starlette Web框架实现，采用装饰器模式将用户函数封装为HTTP端点。这种设计使得开发者无需关心底层网络通信，只需专注于业务逻辑的实现。

核心入口类: HuaweiAgentRunApp

HuaweiAgentRunApp是SDK的核心类，继承自Starlette框架，提供AI代理部署所需的所有Web服务功能。

文件位置：src/hw_agentrun_wrapper/runtime/app.py

基本用法：

```
from hw_agentrun_wrapper import HuaweiAgentRunApp
app = HuaweiAgentRunApp()
@app.entrypoint
def my_agent(payload, context):
    return {"response": "Hello, World!"}
app.run(port=8080)
```

装饰器详解

@app.entrypoint - 主入口装饰器

用于注册主处理函数，处理所有到达/invocations端点的请求。

```
@app.entrypoint
def handle_request(payload, context):
    """
    处理Agent调用请求

    Args:
        payload: 请求载荷，包含用户输入
        context: 请求上下文，包含用户ID、会话ID等信息

    Returns:
        响应数据，可以是普通对象、生成器或异步生成器
    """
    prompt = payload.get("prompt")
    # 处理逻辑
    return {"response": f"处理: {prompt}"}
```

支持的返回类型：

- 普通对象：自动序列化为JSON响应
- 同步生成器：支持流式响应（SSE）
- 异步生成器：支持异步流式响应

@app.ping - 健康检查装饰器

用于注册自定义的健康检查处理器。

```
@app.ping
def custom_ping():
    """返回自定义的ping状态"""
    return "Healthy"  # 或返回 PingStatus.HEALTHY
```

PingStatus枚举值：

- HEALTHY：健康状态
- HEALTHY_BUSY：忙碌状态
- UNHEALTHY：不健康状态

@app.websocket - WebSocket装饰器

用于注册WebSocket处理函数，支持双向实时通信。

```
@app.websocket
async def handle_websocket(websocket, context):
    """处理WebSocket连接"""
    await websocket.accept()
    while True:
        try:
            data = await websocket.receive_text()
            # 处理消息
            await websocket.send_text(f"Echo: {data}")
        except Exception:
            break
```

@app.async_task - 异步任务装饰器

用于注册异步任务，SDK会自动追踪任务健康状态。

```
@app.async_task
async def background_task():
    """后台异步任务"""
    # 任务执行期间，ping状态会自动设置为HEALTHY_BUSY
    # 任务完成后，自动恢复为HEALTHY
    await asyncio.sleep(10)
```

请求上下文

SDK提供两种上下文对象用于获取请求信息：

AgentArtsRuntimeContext

基于contextvars的线程安全运行时上下文，用于在请求处理过程中存储和获取上下文信息。

```
from hw_agentrun_wrapper.runtime.context import AgentArtsRuntimeContext
# 设置上下文
AgentArtsRuntimeContext.set_user_id("user123")
AgentArtsRuntimeContext.set_session_id("session456")
AgentArtsRuntimeContext.set_workload_access_token("token789")
# 获取上下文
user_id = AgentArtsRuntimeContext.get_user_id()
session_id = AgentArtsRuntimeContext.get_session_id()
token = AgentArtsRuntimeContext.get_workload_access_token()
```

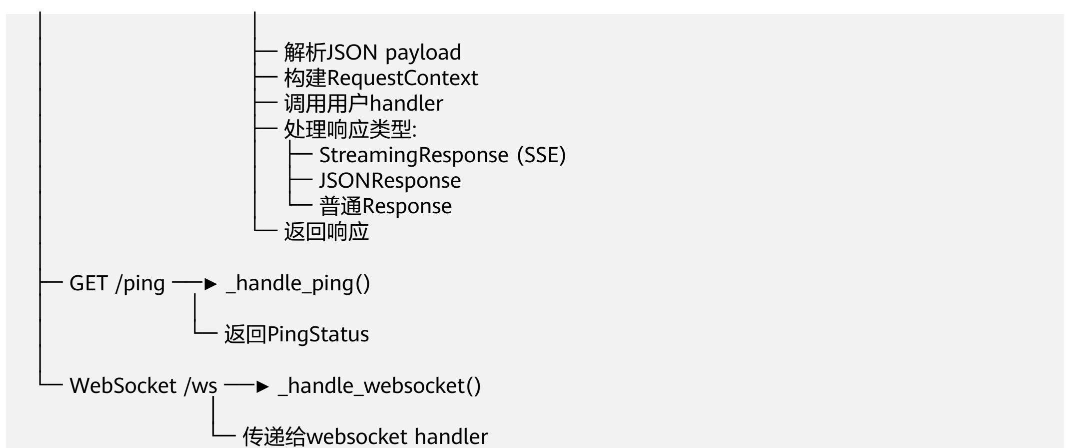
RequestContext

请求级别的上下文对象，包含请求的详细信息。

```
@app.entrypoint
def handle_request(payload, context: RequestContext):
    """
    context包含以下属性：
    - user_id: 用户ID
    - session_id: 会话ID
    - request_id: 请求ID
    - workload_access_token: 工作负载访问令牌
    """
    print(f"User: {context.user_id}")
    print(f"Session: {context.session_id}")
    return {"status": "ok"}
```

框架请求处理流程

```
HTTP请求
    │
    ▼
Starlette Router
    │
    ├─ POST /invocations ──▶ _handle_invocation()
```



7.4 CLI

CLI 概述

AgentArts Starter toolkit是SDK提供的命令行工具，用于在终端中管理Agent的部署、调用、状态查询和资源销毁等操作。

安装与配置

CLI工具通过hw_agentrun_toolkit包提供，安装SDK后即可使用：

```
pip install hw-agentrun
```

验证安装：

```
agentarts --version
```

命令列表

表 7-2 命令列表

命令	说明
init - 初始化agent项目	命令用于初始化一个新的AgentArts项目。该命令会创建完整的项目结构，包括 Agent 实现代码、依赖文件和配置文件，帮助开发者快速开始 Agent 应用开发。
configure - 配置Agent	命令用于管理AgentArts项目配置。支持创建、更新、查看和管理 Agent 配置信息，包括基本配置、SWR配置和运行时配置等。
deploy/launch- 部署Agent	deploy和launch命令用于将Agent部署到华为云AgentArts平台。该命令会构建Docker镜像、上传到SWR（容器镜像服务），并创建或更新Agent实例。
invoke - 调用Agent	用于调用已部署的Agent，向其发送请求并获取响应。支持同步调用和流式调用两种模式，适用于测试Agent功能、调试业务逻辑以及生产环境调用。

版权所有 © 华为云计算技术有限公司

命令	说明
status - 查询状态	用于检查已部署Agent的健康状态。该命令会向Agent发送健康检查请求，返回当前运行状态、响应时间和健康指标，帮助开发者监控Agent运行情况。
destroy - 销毁资源	用于从华为云AgentArts平台删除Agent及其相关资源。该命令会永久删除Agent实例、配置信息和相关资源，执行前会提示确认，防止误操作。

init - 初始化 agent 项目

命令用于初始化一个新的AgentArts项目。该命令会创建完整的项目结构，包括Agent实现代码、依赖文件和配置文件，帮助开发者快速开始Agent应用开发。

表 7-3 可选参数

参数	简写	说明	默认值
--name	-n	项目名称	交互式提示输入
--template	-t	项目模板类型	交互式选择
--path	-p	项目创建路径	. (当前目录)
--region	-r	华为云区域	cn-southwest-2
--swr-org	无	SWR组织名称	agentarts-org
--swr-repo	无	SWR仓库名称	agent_{Agent名称}

表 7-4 模板类型

模板名称	说明
basic	基础模板，适合快速开始。
langgraph	LangGraph框架模板，支持状态工作流。
langchain	LangChain框架模板，支持工具集成。
google-adk	Google ADK框架模板，支持Gemini模型。

执行效果

执行 init 命令后，会在指定路径下创建以下项目结构：

```
{project_name}/
├── agent.py            # Agent 实现代码
```


版权所有 © 华为云计算技术有限公司

```
├── requirements.txt      # 项目依赖文件
├── .agentarts_config.yaml # AgentArts 配置文件
└── Dockerfile           # Docker 构建文件
```

- agent.py: Agent的主要实现文件，包含入口函数和业务逻辑。
- requirements.txt: 项目依赖列表，包括SDK和框架包。
- .agentarts_config.yaml: AgentArts配置文件，包含部署和运行配置。
- Dockerfile: Docker镜像构建文件，用于容器化部署。

使用示例

- 示例 1: 交互式创建项目
  ```
  agentarts init
  ```
  执行后会提示：
  - 输入项目名称
  - 选择模板类型（输入数字选择）
  - 输入华为云区域（默认cn-southwest-2）
- 示例 2: 使用 LangGraph模板创建项目
  ```
  agentarts init --name my-agent --template langgraph
  ```
- 示例 3: 指定路径和区域创建项目
  ```
  agentarts init -n my-agent -t langchain --path ./projects --region cn-southwest-2
  ```
- 示例 4: 完整参数示例
  ```
  agentarts init \
    --name my-agent \
    --template langgraph \
    --path ./my-projects \
    --region cn-southwest-2 \
    --swr-org my-organization \
    --swr-repo my-repository
  ```

configure - 配置 Agent

命令用于管理AgentArts项目配置。支持创建、更新、查看和管理Agent配置信息，包括基本配置、SWR配置和运行时配置等。

说明

configure是config 命令的别名，两者功能完全相同。

表 7-5 config (交互式配置)

参数	简写	说明	是否必填
--name	-n	Agent名称	否，交互式提示
--entrypoint	-e	Agent入口函数	否，交互式提示
--region	-r	华为云区域	否，默认cn-southwest-2
--dependency-file	-d	依赖文件路径	否，自动检测
--swr-org	无	SWR组织名称	否，默认agentarts-org

版权所有 © 华为云计算技术有限公司

参数	简写	说明	是否必填
--swr-repo	无	SWR仓库名称	否，默认agent_{Agent名称}
--set-default	无	设置为默认Agent	否，默认是

执行agentarts config后，会依次提示输入：

1. Agent 名称（显示已有Agent列表）
2. 入口函数（格式：module:function）
3. 华为云区域（默认cn-southwest-2）
4. 依赖文件（自动检测requirements.txt）
5. SWR组织名称
6. SWR仓库名称

配置完成后：

- 保存配置到 .agentarts_config.yaml
- 自动生成 Dockerfile（如果不存在）

使用示例

- 示例 1: 交互式配置
  ```
  agentarts config
  ```
  按提示输入配置信息，适合首次配置或不确定参数时使用。
- 示例 2: 快速配置
  ```
  agentarts config --name my-agent --entrypoint agent:create_app
  ```
  仅提供必要参数，其他使用默认值。
- 示例 3: 完整参数配置
  ```
  agentarts config \
    --name my-agent \
    --entrypoint agent:create_app \
    --region cn-southwest-2 \
    --dependency-file requirements.txt \
    --swr-org my-organization \
    --swr-repo my-repository
  ```
- 示例 4: 列出所有Agent
  ```
  agentarts config list
  ```
- 示例 5: 设置默认Agent
  ```
  agentarts config set-default my-agent
  ```

deploy/launch- 部署 Agent

deploy和launch命令用于将Agent部署到华为云 AgentArts 平台。该命令会构建 Docker 镜像、上传到 SWR（容器镜像服务），并创建或更新 Agent 实例。

说明

launch 是 deploy 命令的别名，两者功能完全相同。

表 7-6 参数解释

参数	简写	说明	默认值
--agent	-a	Agent 名称	使用默认 Agent
--mode	-m	部署模式（local/cloud）	cloud
--tag	-t	Docker 镜像标签	latest
--port	-p	服务端口（覆盖配置）	从配置文件读取
--local-port	-l	本地端口映射（本地模式）	从配置文件读取
--swr-org	无	SWR 组织（覆盖配置）	从配置文件读取
--swr-repo	无	SWR 仓库（覆盖配置）	从配置文件读取
--description	-d	Agent 描述（覆盖配置）	从配置文件读取

部署流程

执行 deploy 命令会依次执行以下步骤：

1. 配置验证
   - 检查 .agentarts_config.yaml 配置文件
   - 验证必要配置项是否完整
   - 确认 Agent 入口函数存在
2. 镜像构建
   - 读取 Dockerfile 构建配置
   - 构建包含 Agent 代码的 Docker 镜像
   - 标记镜像版本（latest 和时间戳版本）
3. 镜像推送
   - 登录华为云 SWR 服务
   - 推送镜像到指定组织和仓库
   - 验证镜像推送成功
4. Agent 创建/更新
   - 调用 AgentArts 控制平面 API
   - 创建新的 Agent 或更新现有 Agent
   - 配置运行时参数和环境变量
5. 部署验证
   - 检查 Agent 状态

版权所有 © 华为云计算技术有限公司

- 验证服务可用性
- 输出访问端点信息

使用示例

- 示例 1: 基本部署

```
agentarts deploy
```

使用默认 Agent 和配置进行部署。

- 示例 2: 指定 Agent 部署

```
agentarts deploy --agent my-agent
```

或使用简写：

```
agentarts deploy -a my-agent
```

invoke - 调用 Agent

用于调用已部署的 Agent，向其发送请求并获取响应。支持同步调用和流式调用两种模式，适用于测试 Agent 功能、调试业务逻辑以及生产环境调用。

表 7-7 参数解释

参数	简写	说明	默认值
payload	-	JSON 格式的请求数据（位置参数）	必填
--agent	-a	Agent 名称	使用默认 Agent
--region	-r	华为云区域	从配置文件读取
--mode	-m	调用模式（local/cloud）	cloud
--port	-p	本地模式端口	8080
--endpoint	-e	指定端点名称	无
--session	-s	会话 ID（用于有状态 Agent）	自动生成 UUID
--bearer-token	-bt	Bearer 认证令牌	无
--timeout	-	请求超时时间（秒）	900

使用示例

- 示例 1: 基本调用

```
agentarts invoke '{"message": "你好"}'
```

- 示例 2: 指定 Agent 调用

```
agentarts invoke '{"message": "你好"}' --agent my-agent
```

- 示例 3: 使用简写参数

```
agentarts invoke '{"message": "你好"}' -a my-agent
```

- 示例 4: 指定区域调用

```
agentarts invoke '{"message": "你好"}' --agent my-agent --region cn-southwest-2
```

- 示例 5: 本地模式调用

```
agentarts invoke '{"message": "你好"}' --mode local
```

status - 查询状态

用于检查已部署 Agent 的健康状态。该命令会向 Agent 发送健康检查请求，返回当前运行状态、响应时间和健康指标，帮助开发者监控 Agent 运行情况。

表 7-8 参数解释

参数	简写	说明	默认值
--agent	-a	Agent 名称	使用默认 Agent
--region	-r	华为云区域	从配置文件读取
--mode	-m	检查模式（local/cloud）	cloud
--port	-p	本地模式端口	8080
--bearer-token	-bt	Bearer 认证令牌	无
--endpoint	-e	指定端点名称	无
--session	-s	会话 ID（用于有状态 Agent）	无

使用示例

- 示例 1: 检查默认 Agent 状态

```
agentarts status
```

- 示例 2: 检查指定 Agent 状态

```
agentarts status --agent my-agent
```

  或使用简写:

```
agentarts status -a my-agent
```

- 示例 3: 指定区域检查

```
agentarts status --agent my-agent --region cn-southwest-2
```

  或使用简写:

```
agentarts status -a my-agent -r cn-southwest-2
```

- 示例 4: 本地模式检查

```
agentarts status --mode local
```

  或指定端口:

```
agentarts status --mode local --port 8080
```

destroy - 销毁资源

用于从华为云 AgentArts 平台删除 Agent 及其相关资源。该命令会永久删除 Agent 实例、配置信息和相关资源，执行前会提示确认，防止误操作。

说明

此操作不可逆，删除后无法恢复 Agent 数据和配置。

表 7-9 参数解释

参数	简写	说明	默认值
--agent	-a	Agent 名称	使用默认 Agent
--region	-r	华为云区域	从配置文件读取
--yes	-y	跳过确认提示	false

使用示例

- 示例 1: 交互式删除默认 Agent
```
agentarts destroy
```
  执行后会提示确认，输入 y 确认删除。
- 示例 2: 删除指定 Agent
```
agentarts destroy --agent my-agent
```

7.5 快速开始

环境准备

- 安装Python：请确保Python 3.10及以上版本已安装。
- 操作系统: Linux，macOS，并安装docker。
- 执行以下命令安装SDK。（强烈建议在Python虚拟环境中安装，以避免与系统包产生冲突）
```
# 创建并激活虚拟环境 (linux)
python -m venv venv
source venv/bin/activate

# 安装sdk
pip install hw-agentrun
```
- 执行以下命令配置华为云凭证，获取华为云凭证请参考**认证鉴权**。
```
export HUAWEICLOUD_SDK_AK="your-access-key"
export HUAWEICLOUD_SDK_SK="your-secret-key"
```

本地开发

步骤1 执行以下命令初始化一个langgraph项目，支持基本的问答能力。

```
agentarts init -n my_agent -t langgraph
```

执行完成后会在目录下创建如下目录和文件：

```
my_agent/
├── agent.py              # Agent implementation
├── requirements.txt      # Python dependencies
├── Dockerfile
└── .agentarts_config.yaml # Project configuration
```

其中，agent.py代码如下：

```
# agent.py
"""
my_agent - LangGraph Agent Implementation

An agent built using LangGraph for stateful workflows, wrapped with AgentArts SDK runtime.
```

```
Environment Variables:
    OPENAI_API_KEY: Your OpenAI API key (required)
        - Get it from: https://platform.openai.com/api-keys
        - Set in .agentarts_config.yaml under runtime.environment_variables
    OPENAI_BASE_URL: Custom API endpoint URL (optional)
        - Use this to connect to OpenAI-compatible APIs (e.g., Azure OpenAI, local LLMs)
        - Example: https://your-api-endpoint.com/v1
        - Leave empty to use default OpenAI API
    OPENAI_MODEL_NAME: Model name to use (optional, default: gpt-4o-mini)
        - Examples: gpt-4o, gpt-4o-mini, gpt-4-turbo, gpt-3.5-turbo
        - Or custom model names for OpenAI-compatible APIs

Configuration:
    Edit .agentarts_config.yaml to customize:
    - runtime.environment_variables: Set OPENAI_API_KEY, OPENAI_MODEL_NAME and optionally
OPENAI_BASE_URL
    - runtime.invoke_config: Change protocol or port
    - runtime.network_config: Configure VPC if needed

Usage:
    1. Set your OPENAI_API_KEY in .agentarts_config.yaml
    2. (Optional) Set OPENAI_MODEL_NAME to use a different model
    3. (Optional) Set OPENAI_BASE_URL for custom endpoints
    4. Run: agentarts deploy
    5. Invoke: agentarts invoke '{"message": "Hello"}'
"""
import os
from typing import Dict, Any, TypedDict, Annotated
from operator import add
# 步骤1 导入sdk相关包
from agentarts.sdk import AgentArtsRuntimeApp, RequestContext
# AgentArtsSDK runtime执行入口app
app = AgentArtsRuntimeApp()

# 步骤2 定义主体Agent业务逻辑
class State(TypedDict):
    messages: Annotated[list, add]
    query: str
    response: str

class LangGraphAgent:
    def __init__(self):
        self.model_name = os.environ.get("OPENAI_MODEL_NAME", "gpt-4o-mini")
        self._graph = None

    def _build_graph(self):
        from langgraph.graph import StateGraph, END
        from langchain_openai import ChatOpenAI
        from langchain_core.messages import HumanMessage, AIMessage
        llm = ChatOpenAI(
            model=self.model_name,
            api_key=os.environ.get("OPENAI_API_KEY"),
            base_url=os.environ.get("OPENAI_BASE_URL")
        )
        async def process_node(state: State) -> Dict[str, Any]:
            query = state.get("query", "")
            messages = state.get("messages", []) or [HumanMessage(content=query)]
            response = await llm.ainvoke(messages)
            return {
                "messages": [AIMessage(content=response.content)],
                "response": response.content,
            }
        workflow = StateGraph(State)
        workflow.add_node("process", process_node)
        workflow.set_entry_point("process")
        workflow.add_edge("process", END)
        return workflow.compile()
```


版权所有 © 华为云计算技术有限公司

```
    async def run(self, query: str) -> Dict[str, Any]:
        graph = self._graph or self._build_graph()
        self._graph = graph
        result = await graph.ainvoke({"messages": [], "query": query, "response": ""})
        return {"response": result.get("response", "")}

_agent = LangGraphAgent()

# 步骤3 定义http执行入口
@app.entrypoint  # 装饰器/invocations端点
async def handler(payload: Dict[str, Any], context: RequestContext = None) -> Dict[str, Any]:
    query = payload.get("message", "")
    return await _agent.run(query)

if __name__ == "__main__":
    app.run()
```

步骤2 配置环境变量等信息，跳转到my_agent目录下，编辑.agentarts_config.yaml如下：

```
runtime:
  environment_variables:
    - key: OPENAI_API_KEY
      value: "your-openai-api-key"
    - key: OPENAI_MODEL_NAME
      value: "gpt-4o-mini"  # Optional: gpt-4o, gpt-4-turbo, etc.
    - key: OPENAI_BASE_URL
      value: ""  # Optional: custom API endpoint
```

步骤3 执行以下命令进行本地调试。

```
pip install -r requirements.txt

agentarts dev
```

在服务启动后，可以访问如下地址：

- 健康检查

  Curl调用示例

  ```
  curl http://localhost:8080/ping
  ```

  检查结果示例

  ```
  {"status":"Healthy","time_of_last_update":1775718928}
  ```

- Agent调用

  Curl请求示例

  ```
  curl --location --request POST 'http://localhost:8080/invocations' --data-raw '{"message": "你好，你是谁"}'
  ```

  调用成功结果示例

  ```
  {"response": "你好！我是 **xxx**"}
  ```

----结束

云端部署

步骤1 配置Agent部署region

```
agentarts config set region cn-southwest-2
```

也可执行agentarts config命令进行交互式配置，配置向导会引导您完成以下设置：（按“回车”使用默认值）

- 部署region: 默认西南贵阳一cn-southwest-2
- SWR组织：默认自动创建
- SWR仓库：默认自动创建

- 依赖文件：默认requirements.txt

如果需要自定义配置（例如，指定部署区域、镜像仓库或者agent入站认证、环境变量等），可以手动编辑 .agentarts_config.yaml文件中的配置。

步骤2 配置完成后，通过命令一键部署到AgentArts平台的云端(需要本地安装docker)

```
agentarts launch
```

该命令会自动完成以下步骤：

1. 本地构建Docker镜像。
2. 将Docker镜像推送到SWR仓库。
3. 部署到AgentArts运行时。

步骤3 调用云端Agent

```
agentarts invoke '{"message": "Hello World"}'
```

----结束

7.6 Runtime SDK

场景介绍

Runtime SDK是SDK的核心组成部分，提供Agent运行时所需的所有功能，包括请求处理、响应生成、上下文管理等。

原理优势

1. 开箱即用，一键部署
   - **零配置部署**：一行命令即可将本地Agent部署到云端Runtime，无需手动构建镜像、配置网络。
   - **CLI工具完备**：agentarts dev本地运行、agentarts launch一键部署、agentarts invoke本地运行。
   - **模板丰富**：内置多种Agent框架模板（Langchain/LangGraph、GoogleADK等），开箱即用。
2. 框架无关，极致兼容
   - **多框架支持**：原生支持Langchain/LangGraph、GoogleADK等主流框架。
   - **统一入口**：无论使用何种框架，都可以通过统一的SDK接口进行管理和部署。
   - **平滑迁移**：现有Agent代码只需少量适配即可接入。
3. 云端托管，Serverless体验
   - **无需运维**：云端自动托管运行环境，无需关心服务器、容器、网络配置。
   - **弹性伸缩**：根据流量自动扩缩容，从零到大规模并发自适应。
   - **按量付费**：仅在实际调用时计费，空闲时自动释放资源。
4. 企业级安全隔离
   - **会话隔离**：每个用户/会话独立运行环境，防止数据串扰。
   - **权限控制**：与IAM深度集成，支持精细化的访问控制。
   - **安全防护**：基于云原生的安全机制，保障Agent和数据安全。

5. 可观测性内置
   - **实时监控**：开箱即用的健康状态监控和指标展示。
   - **日志追踪**：自动收集和展示运行日志，快速定位问题。
   - **调用洞察**：完整的请求/响应日志和性能分析。
6. 丰富的生态集成
   - **多协议支持**：原生支持HTTP、WebSocket、MCP等协议。
   - **云服务集成**：深度集成华为云SWR（镜像仓库）、IAM（身份认证）、APM（应用监控）等服务。

核心特性

1. AgentArtsRuntimeApp类
   - 初始化参数
```
app = AgentArtsRuntimeApp(
    debug=False,           # 调试模式
    lifespan=None,         # 生命周期管理
    middleware=None,       # 中间件列表
    protocol="http",       # 协议类型: "http" 或 "https"
)
```
   - run() 方法

     启动ASGI服务器：
```
app.run(host="0.0.0.0", port=8080)
```
     参数说明：

     host: 绑定地址，默认在Docker环境中为 0.0.0.0，本地为 127.0.0.1

     port: 绑定端口，默认 8080

     **kwargs: 传递给uvicorn的其他参数
2. 装饰器
   - @app.entrypoint

     注册Agent主入口函数，处理 /invocations端点的请求。

     基本用法
```
@app.entrypoint
def handler(payload: dict) -> dict:
    """同步处理函数"""
    return {"result": payload["message"].upper()}

@app.entrypoint
async def async_handler(payload: dict) -> dict:
    """异步处理函数"""
    result = await some_async_operation(payload)
    return result
```
     使用请求上下文
```
@app.entrypoint
async def handler(payload: dict, context: RequestContext = None) -> dict:
    """带上下文的处理函数"""
    session_id = context.session_id if context else None
    request_id = context.request_id if context else None

    return {
        "response": "OK",
        "session_id": session_id,
        "request_id": request_id,
    }
```
     流式响应-返回生成器以实现流式输出（SSE格式）
```
@app.entrypoint
async def streaming_handler(payload: dict) -> AsyncGenerator:
```

```
    """流式响应处理函数"""
    message = payload.get("message", "")

    for char in message:
        await asyncio.sleep(0.1)
        yield {"chunk": char}

@app.entrypoint
def sync_streaming_handler(payload: dict) -> Generator:
    """同步流式响应"""
    for i in range(10):
        yield {"count": i}
```

– @app.ping

注册健康检查处理函数，处理 /ping端点的请求。

```
from agentarts.sdk.runtime.model import PingStatus

@app.ping
def health_check() -> PingStatus:
    """自定义健康检查"""
    if is_healthy():
        return PingStatus.HEALTHY
    else:
        return PingStatus.UNHEALTHY
```

表 7-10 PingStatus 状态值

状态	说明
HEALTHY	服务健康，无正在执行的任务。
HEALTHY_BUSY	服务健康，有任务正在执行。
UNHEALTHY	服务不健康。

强制设置状态

```
# 设置维护模式app.force_ping_status(PingStatus.UNHEALTHY)  # 恢复正常
app.force_ping_status(None)
```

– @app.websocket

注册WebSocket处理函数，处理 /ws端点的连接。

```
from starlette.websockets import WebSocket

@app.websocket
async def ws_handler(websocket: WebSocket, context: RequestContext = None):
    """WebSocket 处理函数"""
    await websocket.accept()

    try:
        while True:
            data = await websocket.receive_json()
            response = await process_message(data)
            await websocket.send_json(response)
    except WebSocketDisconnect:
        print("Client disconnected")
```

WebSocket示例：聊天应用

```
@app.websocket
async def chat_handler(websocket: WebSocket, context: RequestContext = None):
    await websocket.accept()
    session_id = context.session_id if context else "default"

    try:
        while True:
```

```
            message = await websocket.receive_text()
            response = await agent.chat(session_id, message)
            await websocket.send_text(response)
    except WebSocketDisconnect:
        await agent.end_session(session_id)
```

– @app.async_task

注册异步后台任务，任务执行状态会被自动追踪。

```
@app.async_tas
async def background_job(payload: dict):
    """后台异步任务"""
    await asyncio.sleep(10)
    result = await process_data(payload)
    return result

@app.entrypoint
async def handler(payload: dict, context: RequestContext = None):
    # 启动后台任务
    asyncio.create_task(background_job(payload))

    # 检查是否有运行中的任务
    if app.has_running_tasks():
        print("有后台任务正在执行")

    return {"status": "accepted"}
```

3. RequestContext

– RequestContext是请求数据的上下文，包含请求的元信息。

表 7-11 属性

属性	类型	说明
request_id	Optional[str]	请求唯一标识符
session_id	Optional[str]	会话标识符
request	Optional[Any]	原始请求对象

– 使用示例

```
@app.entrypoint
async def handler(payload: dict, context: RequestContext = None):
    if context:
        print(f"Request ID: {context.request_id}")
        print(f"Session ID: {context.session_id}")

        # 访问原始请求对象
        if context.request:
            headers = context.request.headers
            client_host = context.request.client.host

    return {"status": "ok"}
```

4. AgentArtsRuntimeContext

– AgentArtsRuntimeContext是全局上下文管理器，基于Python的contextvars实现，支持异步安全的上下文访问。

– 核心特性

  ■ 协程安全：每个异步任务都有独立的上下文视图。

  ■ 全局访问：在调用栈的任何位置都可以访问上下文数据。

- 无需实例化：直接使用类方法访问。

– 可用方法

方法	说明
get_session_id()	获取会话ID
set_session_id(value)	设置会话ID
get_request_id()	获取请求ID
set_request_id(value)	设置请求ID
get_user_id()	获取用户ID
set_user_id(value)	设置用户ID
get_workload_access_token()	获取工作负载访问令牌
set_workload_access_token(value)	设置工作负载访问令牌
get_user_token()	获取用户令牌
set_user_token(value)	设置用户令牌
get_oauth2_callback_url()	获取OAuth2回调URL
set_oauth2_callback_url(value)	设置OAuth2回调URL
clear()	清除所有上下文变量

– 使用示例

```
from agentarts.sdk.runtime.context import AgentArtsRuntimeContext

@app.entrypoint
async def handler(payload: dict, context: RequestContext = None):
    # 方式一：通过 context 参数获取
    session_id = context.session_id if context else None

    # 方式二：通过全局上下文获取
    session_id = AgentArtsRuntimeContext.get_session_id()
    request_id = AgentArtsRuntimeContext.get_request_id()
    user_id = AgentArtsRuntimeContext.get_user_id()

    # 在深层调用中使用
    result = await process_with_context()
    return {"session_id": session_id}

async def process_with_context():
    """在任意函数中访问上下文"""
    session_id = AgentArtsRuntimeContext.get_session_id()
    # 使用 session_id 进行业务处理
    return {"processed": True, "session": session_id}
```

7.7 Tools SDK

场景介绍

支持CodeInterpreter生命周期管理和Session生命周期管理。

原理优势

用户空间已存在代码解释器时，SDK支持快速拉起代码解释器会话

```
with code_session("your_region"，"your_code_interpreter_name") as code_client:
    response = code_client.invoke(
        operate_type="your_operate_type",
        arguments={your_arguments}
    )
```

前提条件

- 已开通AgentArts。
- 登录控制台获取AK/SK，和API_KEY信息，详情请参考**如何获取华为云AK/SK**和**如何获取沙箱工具的API_KEY**。
- 已安装Python，且版本不低于3.10。查看Python版本的命令示例：python --version。
- 拥有华为云AgentArts Tools的访问权限。
- 拥有创建和管理Tools资源所需的权限。

操作步骤

步骤1 执行如下命令或者使用pip install方式进行环境准备。

```
# Clone the SDK (use HTTPS or SSH)
git clone -b release_opensource https://codehub-dg-g.huawei.com/applicationplatform/ServiceStage/
AgentRun/AgentRunSDK.git
# Use Virtual Environment (Recommended)
python -m venv venv
source venv/bin/activate
# Install SDK from the SDK parent directory where pyproject.toml is
cd AgentRunSDK
make install
```

pip install方式：

```
pip install hw-agentarts-sdk
```

步骤2 配置华为云访问凭证：

- **管理面（控制面）**：使用AK/SK认证，通过HUAWEICLOUD_SDK_AK和HUAWEICLOUD_SDK_SK设置环境变量。获取AK/SK请参考**如何获取华为云AK/SK**。
- **数据面**：使用API_KEY认证，通过HUAWEICLOUD_SDK_CODE_INTERPRETER_API_KEY设置环境变量，SDK内部自动处理。获取API_KEY请参考**如何获取沙箱工具的API_KEY**。

步骤3 创建代码解释器

```
from agentarts.sdk.tools import CodeInterpreter

client = CodeInterpreter(region="your_region")
code_interpreter = client.create_code_interpreter(
        "name"="your_code_interpreter_name"
        "api_key_name"="your_api_key_name"
    )
```

步骤4 配置工具，这里以代码执行工具为例。用户自定义region信息和需要使用的代码解释器，代码解释器名称和API_KEY一一对应。

```
from agentarts.sdk.tools import code_session

@tool
```

```
def execute_python_tool(code: str, description: str = "") -> str | None:
    with code_session("your_region", "your_code_interpreter_name") as code_client:
        response = code_client.invoke(
            operate_type="execute_code",
            arguments={
                "code": code,
                "language": "python",
                "clearContext": False
            }
        )

    return json.dumps(response["result"])
```

步骤5 配置集成**步骤4**中配置的工具，以如下执行代码为例。

```
# 创建Agent
llm = ChatOpenAI(
    model="DeepSeek-V3",
    api_key=os.environ.get("MODEL_API_KEY", ""),
    base_url=os.environ.get("BASE_URL", ""),
    max_tokens=1000,
    temperature=0.7,
)
# 创建工具列表
tools = [execute_python_tool]
# 工具绑定Agent
llm.bind_tools(tools)
```

----结束

7.8 Memory SDK

场景介绍

AgentRunSDK Memory SDK提供两种使用模式：

- **Client模式**：功能完整的客户端，适合需要控制所有操作的应用场景。
- **Session模式**：基于绑定的会话进行记忆管理，适合特定用户/对话场景。

原理优势

- Client模式完整案例

```
#!/usr/bin/env python3
"""
Client模式完整使用案例
创建Space -> 创建Session -> 发送Message -> 查询Memory -> 搜索Memory
"""
import os
import time
from agentarts.sdk.memory import MemoryClient
from agentarts.sdk.memory.inner.config import TextMessage
# 确保环境变量已设置
assert os.getenv('HUAWEICLOUD_SDK_AK'), "请设置 HUAWEICLOUD_SDK_AK"
assert os.getenv('HUAWEICLOUD_SDK_SK'), "请设置 HUAWEICLOUD_SDK_SK"
def client_mode_example():
    """Client模式完整示例"""
    print("=== Client模式完整示例 ===")
    # 1. 创建Space
    with MemoryClient() as client:
        print("1. 创建测试Space")
        space = client.create_space(
            name=f"client_test_space_{int(time.time())}",
            message_ttl_hours=168,
            description="Client模式测试专用空间",
            memory_strategies_builtin=["semantic", "user_preference", "episodic"]
```

```
        )
        space_id = space.id
        print(f"✓ Space创建成功: {space_id}")
        print(f"  API Key ID: {space.api_key_id}")
        # 2. 创建会话
        print("\n2. 创建会话")
        session_data = client.create_memory_session(
            space_id=space_id,
            actor_id="client-test-user",
            assistant_id="client-test-assistant"
        )
        session_id = session_data.id
        print(f"✓ Session创建成功: {session_id}")
        # 3. 发送消息（使用TextMessage对象）
        print("\n3. 发送对话消息")
        messages = [
            TextMessage(
                role="user",
                content="你好，我想了解电商推荐算法，能够根据用户行为进行个性化推荐的算法有哪些？",
                actor_id="client-test-user"
            ),
            TextMessage(
                role="assistant",
                content="电商推荐算法主要包括：1) 协同过滤算法，基于用户行为相似性推荐；2) 基于内容
的推荐，分析商品特征；3) 深度学习算法，能更准确捕捉用户偏好。推荐组合使用多种算法提升准确率。
",
                actor_id="client-test-assistant"
            ),
            TextMessage(
                role="user",
                content="我对机器学习的监督学习特别感兴趣，深度学习和传统机器学习有什么区别？",
                actor_id="client-test-user"
            )
        ]
        add_result = client.add_messages(
            space_id=space_id,
            session_id=session_id,
            messages=messages
        )
        print(f"✓ 已添加 {len(add_result.items)} 条消息")
        # 4. 等待记忆系统处理消息
        print("\n4. 等待记忆系统处理消息...")
        time.sleep(30)
        # 5. 列出记忆
        print("\n5. 查询记忆列表")
        memories = client.list_memories(
            space_id=space_id,
            limit=10
        )
        print(f"✓ 发现 {len(memories.items)} 条记忆")
        for i, memory in enumerate(memories.items[:3]):
            print(f"  {i+1}. {memory.content[:50]}...")
            print(f"     策略: {memory.strategy_type}")
        # 6. 搜索记忆（使用MemorySearchFilter对象）
        print("\n6. 搜索相关记忆")
        from hw_agentrun_wrapper.memory.inner.config import MemorySearchFilter
        search_results = client.search_memories(
            space_id=space_id,
            filters=MemorySearchFilter(query="机器学习", top_k=3)
        )
        print(f"✓ 找到 {len(search_results.results)} 条相关记忆")
        for i, result in enumerate(search_results.results):
            score = result.get('score', 0)
            content = result.get('record', {}).get('content', '')[:60]
            print(f"  {i+1}. [{score:.2f}] {content}...")
        # 7. 获取特定记忆详情
        if memories.items:
            print("\n7. 获取记忆详情")
            memory_id = memories.items[0].id
```

```
            memory_detail = client.get_memory(space_id, memory_id)
            print(f"✓ 记忆ID: {memory_detail.id}")
            print(f"   内容: {memory_detail.content[:80]}...")
            print(f"   策略: {memory_detail.strategy_type}")
        return space_id, session_id
if __name__ == "__main__":
    client_mode_example()
```

- **Session模式完整案例**

```
#!/usr/bin/env python3
"""
Session模式完整使用案例
创建Space -> 绑定会话 -> 发送Message -> 查询Memory -> 搜索Memory
"""
import os
import time
from agentarts.sdk.memory import MemoryClient
from agentarts.sdk.memory.session import MemorySession
from agentarts.sdk.memory.inner.config import TextMessage
# 确保环境变量已设置
assert os.getenv('HUAWEICLOUD_SDK_AK'), "请设置 HUAWEICLOUD_SDK_AK"
assert os.getenv('HUAWEICLOUD_SDK_SK'), "请设置 HUAWEICLOUD_SDK_SK"
def session_mode_example():
    """Session模式完整示例"""
    print("=== Session模式完整示例 ===")
    # 1. 创建Space
    with MemoryClient() as client:
        print("1. 创建测试Space")
        space = client.create_space(
            name=f"session-test-space-{int(time.time())}",
            message_ttl_hours=168,
            description="Session模式测试专用空间",
            memory_strategies_builtin=["semantic", "user_preference", "episodic"]
        )
        space_id = space.id
        print(f"✓ Space创建成功: {space_id}")
        # 2. 创建并绑定会话
        print("\n2. 创建并绑定会话")
        session_obj = MemorySession(
            space_id=space_id,
            actor_id="session-test-user",
            assistant_id="session-test-assistant"
        )
        session_id = session_obj.session_id
        print(f"✓ Session创建并绑定成功: {session_id}")
        # 3. 发送对话消息（使用TextMessage对象）
        print("\n3. 模拟用户与AI助手对话")
        messages = [
            TextMessage(
                role="user",
                content="我是一个数据分析师，主要使用Python和SQL进行数据处理"
            ),
            TextMessage(
                role="assistant",
                content="数据分析师是很有前景的职业！Python方面，pandas、numpy、matplotlib是核心
库。"
            ),
            TextMessage(
                role="user",
                content="我最感兴趣的是可视化和机器学习方向，有什么推荐的学习路径吗？"
            ),
            TextMessage(
                role="assistant",
                content="可视化推荐学习matplotlib、seaborn、plotly；机器学习可以从scikit-learn入门，然
后学习深度学习框架如TensorFlow或PyTorch。"
            ),
            TextMessage(
                role="user",
                content="我对Python数据分析很熟练，但机器学习经验还比较少，应该从哪里开始？"
            )
```

```
        ]
        add_result = session_obj.add_messages(messages)
        print(f"✓ 已添加 {len(messages)} 条对话消息")
        # 4. 等待记忆系统处理
        print("\n4. 等待记忆系统生成...")
        time.sleep(30)
        # 5. 查询消息（在绑定会话上下文中）
        print("\n5. 查询当前会话的消息")
        messages_response = session_obj.list_messages(limit=10)
        print(f"✓ 会话中有 {messages_response.total} 条消息")
        # 6. 获取会话中的记忆列表
        print("\n6. 查看生成的记忆")
        memories = session_obj.list_memories(limit=10)
        memory_list = memories.items
        print(f"✓ 总共发现 {len(memory_list)} 条记忆")
        for i, memory in enumerate(memory_list):
            content = memory.content[:50]
            strategy = memory.strategy_type or 'unknown'
            print(f"  {i+1}. [{strategy}] {content}...")
        # 7. 在会话上下文中搜索记忆（使用MemorySearchFilter对象）
        print("\n7. 搜索Python相关记忆")
        from hw_agentrun_wrapper.memory.inner.config import MemorySearchFilter
        search_results = session_obj.search_memories(
            filters=MemorySearchFilter(query="Python", top_k=3)
        )
        print(f"✓ 找到 {len(search_results.results)} 条相关记忆")
        for i, result in enumerate(search_results.results):
            score = result.get('score', 0)
            content = result.get('record', {}).get('content', '')[:60]
            print(f"  {i+1}. [{score:.2f}] {content}...")
        # 8. 获取特定记忆详情
        if memory_list:
            print("\n8. 获取第一条记忆详情")
            memory_id = memory_list[0].id
            memory_detail = session_obj.get_memory(memory_id)
            print(f"✓ 记忆ID: {memory_detail.id}")
            print(f"   内容: {memory_detail.content[:80]}...")
            print(f"   策略: {memory_detail.strategy_type}")
            print(f"   来源策略: {memory_detail.strategy_id or 'N/A'}")
        return space_id, session_id
if __name__ == "__main__":
    session_mode_example()
```

前提条件

认证鉴权：

- 华为云访问密钥，用于调用管理面创建space接口认证。获取华为云凭证请参考**如何获取华为云AK/SK**。

```
export HUAWEICLOUD_SDK_AK="你的AK"
export HUAWEICLOUD_SDK_SK="你的SK"
```

- 创建space后，会返回HW_API_KEY，可将其设置为环境变量，用于调用数据面接口认证。

```
export HUAWEICLOUD_SDK_MEMORY_API_KEY="你的数据面API密钥"
```

  也可以在代码中传入。

```
from agentarts.sdk.memory import MemoryClient

client = MemoryClient(api_key="your-memory-api-key")
```

参数详解

认证信息

- **管理面（控制面）**：使用AK/SK认证，通过HUAWEICLOUD_SDK_AK和HUAWEICLOUD_SDK_SK环境变量。

版权所有 © 华为云计算技术有限公司

- **数据面**：使用API_KEY认证，SDK内部自动处理。

请求流程

1. 使用AK/SK创建Space（控制面）。
2. SDK自动创建API Key并返回api_key（仅创建时可见）和api_key_id。
3. SDK内部自动使用API Key进行数据面操作。
4. 配置数据面端点（可选，通过环境变量）。

类型化返回值

SDK使用类型化对象作为返回值，便于代码提示和类型检查：

返回类型	说明	重要属性
SpaceInfo	Space信息	id, name, api_key, api_key_id, status, created_at
SpaceListResponse	Space列表	items (SpaceInfo列表), total, limit, offset
SessionInfo	Session信息	id, space_id, actor_id, assistant_id, created_at
MessageInfo	消息信息	id, session_id, role, parts, actor_id, created_at
MessageListResponse	消息列表	items (MessageInfo列表), total, limit, offset
MessageBatchResponse	批量消息响应	items (MessageInfo列表)
MemoryInfo	记忆信息	id, space_id, content, strategy_type, strategy_id, created_at
MemoryListResponse	记忆列表	items (MemoryInfo列表), total, limit, offset
MemorySearchResponse	搜索结果	results (列表，含record和score), total, query

类型化请求对象

SDK提供类型化的请求对象用于构建参数：

请求类型	用途
TextMessage	文本消息，参数：role, content, actor_id, assistant_id
ToolCallMessage	工具调用消息，参数：id, name, arguments

请求类型	用途
ToolResultMessage	工具结果消息，参数：tool_call_id, content, asset_ref
MemorySearchFilter	记忆搜索过滤，参数：query, top_k, min_score等
MemoryListFilter	记忆列表过滤，参数：strategy_type, actor_id, sort_by等

记忆使用建议

- 始终检查环境变量是否正确设置。
- 根据场景选择Client或Session模式。
- 理解记忆生成的延迟时间。
- 合理设置Space的TTL时间。
- 及时清理测试数据。

7.9 Identity SDK

场景介绍

Identity SDK提供华为云身份认证服务的管理能力，支持工作负载身份、凭证提供者、访问令牌等管理功能。

原理优势

IdentityClient是身份认证服务的高级客户端。

文件位置：src/hw_agentrun_wrapper/services/identity/identity_client.py

```
from hw_agentrun_wrapper.services.identity import IdentityClient
client = IdentityClient(region="cn-southwest-2")
```

操作步骤

步骤1 创建工作负载身份

```
workload = client.create_workload_identity(
    name="my-workload",
    description="我的工作负载"
)
print(f"Workload ID: {workload.id}")
print(f"URN: {workload.urn}")
```

步骤2 创建访问令牌

```
# 使用用户Token创建
token = client.create_workload_access_token(
    workload_name="my-workload",
    user_token="user-oauth-token"
)
# 使用用户ID创建
token = client.create_workload_access_token_for_user_id(
```


版权所有 © 华为云计算技术有限公司

```
    workload_name="my-workload",
    user_id="user-123"
)
print(f"Access Token: {token.access_token}")
print(f"Expires At: {token.expires_at}")
```

步骤3 配置凭证提供者管理。

- API Key凭证提供者

```
provider = client.create_api_key_credential_provider(
    name="my-api-key-provider",
    api_key="your-api-key"
)
print(f"Provider ID: {provider.id}")
```

- OAuth2凭证提供者

```
# GitHub OAuth2
provider = client.create_oauth2_credential_provider(
    name="github-provider",
    vendor="github",
    client_id="your-client-id",
    client_secret="your-client-secret"
)
# Google OAuth2
provider = client.create_oauth2_credential_provider(
    name="google-provider",
    vendor="google",
    client_id="your-client-id",
    client_secret="your-client-secret"
)
# Microsoft OAuth2
provider = client.create_oauth2_credential_provider(
    name="microsoft-provider",
    vendor="microsoft",
    client_id="your-client-id",
    client_secret="your-client-secret"
)
```

- STS凭证提供者

```
provider = client.create_sts_credential_provider(
    name="my-sts-provider",
    agency_urn="agency-urn:your-agency",
    tags=[{"key": "env", "value": "prod"}]
)
```

步骤4 配置资源凭证获取。

- 获取OAuth2 Token

```
token = client.get_resource_oauth2_token(
    provider_name="github-provider",
    scopes=["user:email", "read:user"],
    agent_identity_token="agent-identity-token"
)
print(f"Token: {token.access_token}")
```

- 获取API Key

```
api_key = client.get_resource_api_key(
    provider_name="my-api-key-provider",
    workload_access_token="workload-token"
)
print(f"API Key: {api_key.key}")
```

- 获取STS Token

```
sts_token = client.get_resource_sts_token(
    provider_name="my-sts-provider",
    workload_access_token="workload-token",
    agency_session_name="session-name"
)
print(f"Access Key: {sts_token.access_key}")
print(f"Secret Key: {sts_token.secret_key}")
print(f"Security Token: {sts_token.security_token}")
```


版权所有 © 华为云计算技术有限公司

步骤5 使用SDK提供装饰器以保护Agent端点。

```
from hw_agentrun_wrapper.identity.auth import (
    require_access_token,
    require_api_key,
    require_sts_token
)
app = HuaweiAgentRunApp()
# 需要访问令牌
@app.entrypoint
@require_access_token
def protected_agent(payload, context):
    return {"message": "需要有效的访问令牌"}
# 需要API Key
@app.entrypoint
@require_api_key
def api_key_protected_agent(payload, context):
    return {"message": "需要有效的API Key"}
# 需要STS Token
@app.entrypoint
@require_sts_token
def sts_protected_agent(payload, context):
    return {"message": "需要有效的STS Token"}
```

----结束

7.10 MCP Gateway SDK

场景介绍

MCP Gateway CLI提供管理MCP网关及相关操作的命令行工具，支持网关和网关目标的创建、查询、更新、删除等功能。

AgentRunSDK Gateway SDK提供两种使用模式：

- **client模式**：与MCP Gateway API交互的客户端。
- **cli模式**：通过命令行进行生命周期管理。

Client 模式

- **创建MCP Gateway**

```
def create_mcp_gateway(
    self,
    name: Optional[str] = None,
    description: Optional[str] = None,
    protocol_type: Optional[str] = "MCP",
    authorizer_type: Optional[str] = "IAM",
    agency_name: Optional[str] = None,
    authorizer_configuration: Optional[CoreGatewayAuthorizerConfiguration] = None,
    log_delivery_configuration: Optional[CoreGatewayLogDeliveryConfigurationRequestBody] = None,
    outbound_network_configuration: Optional[CoreGatewayOutboundNetworkConfiguration] = None,
```

表 7-12 创建 MCP Gateway 参数说明

参数	类型	是否必选	默认值	说明
name	Optional[str]	否	gateway-{8个随机字符后缀}	网关名称

参数	类型	是否必选	默认值	说明
description	Optional[str]	否	-	网关的详细描述
protocol_type	Optional[str]	否	MCP	网关协议类型
authorizer_type	Optional[str]	否	IAM	授权器类型，可选值：custom_jwt、iam、api_key
agency_name	Optional[str]	否	AgentArksCoreGateway	网关委托身份的委托名称
authorizer_configuration	Optional[CoreGatewayAuthorizerConfiguration]	否	-	授权器配置
log_delivery_configuration	Optional[CoreGatewayLogDeliveryConfigurationRequestBody]	否	-	日志投递配置
outbound_network_configuration	Optional[CoreGatewayOutboundNetworkConfiguration]	否	-	出站网络配置

授权器配置要求：

– 当 authorizer_type = 'custom_jwt' 时，必须提供 custom_jwt_authorizer 配置。

– 当 authorizer_type = 'api_key' 时，必须提供 key_auth 配置。

- 查询MCP Gateway信息

```
def get_mcp_gateway(self, gateway_id: str) -> ShowCoreGatewayResponse
```

表 7-13 查询 MCP Gateway 参数说明

参数	类型	是否必选	说明
gateway_id	str	是	要查询的Gateway的唯一标识符

- 更新指定的MCP Gateway信息

```
def update_mcp_gateway(
    self,
```


版权所有 © 华为云计算技术有限公司

```
    gateway_id: str,
    description: Optional[str] = None,
    authorizer_configuration: Optional[CoreGatewayAuthorizerConfiguration] = None,
    log_delivery_configuration: Optional[CoreGatewayLogDeliveryConfigurationRequestBody] = None,
```

表 7-14 更新 MCP Gateway 参数说明

参数	类型	是否必选	说明
gateway_id	str	是	要更新的Gateway的唯一标识符
description	Optional[str]	否	网关的详细描述
authorizer_configuration	Optional[CoreGatewayAuthorizerConfiguration]	否	授权器配置
log_delivery_configuration	Optional[CoreGatewayLogDeliveryConfigurationRequestBody]	否	日志投递配置

说明

更新操作为增量更新，只传入需要更新的字段，未传入的字段将保持原值不变。

- 删除指定的MCP Gateway信息

```
def delete_mcp_gateway(self, gateway_id: str) -> DeleteCoreGatewayResponse
```

表 7-15 更新 MCP Gateway 参数说明

参数	类型	是否必选	说明
gateway_id	str	是	要删除的Gateway的唯一标识符

- 获取MCP Gateway列表

```
def list_mcp_gateways(
    self,
    name: Optional[str] = None,
    status: Optional[str] = None,
    gateway_id: Optional[str] = None,
    tags: Optional[str] = None,
    limit: Optional[int] = None,
    offset: Optional[int] = None) -> ListCoreGatewaysResponse
```

表 7-16 获取 MCP Gateway 实例列表参数说明

参数	类型	是否必选	默认值	说明
name	Optional[str]	否	-	按名称过滤网关，支持模糊匹配
status	Optional[str]	否	-	按状态过滤网关
gateway_id	Optional[str]	否	-	按网关ID过滤，支持多个ID（最多20个），用逗号分隔
tags	Optional[str]	否	-	按标签过滤网关，支持多个标签，用逗号分隔
limit	Optional[int]	否	50	返回结果的最大数量
offset	Optional[int]	否	0	返回结果的偏移量

- 在指定的MCP Gateway下创建新的Target实例

```
def create_mcp_gateway_target(
    self,
    gateway_id: str,
    name: Optional[str] = None,
    description: Optional[str] = None,
    target_configuration: Optional[CoreGatewayTargetConfiguration] = None,
    credential_provider_configuration: Optional[CoreGatewayCredentialProviderConfiguration] = None)
-> CreateCoreGatewayTargetResponse
```

表 7-17 创建 MCP Target 参数说明

参数	类型	是否必选	默认值	说明
gateway_id	str	是	-	目标Gateway的唯一标识符
name	Optional[str]	否	target-{8个随机字符后缀}	Target服务名称
description	Optional[str]	否	-	Target服务描述
target_configuration	Optional[CoreGatewayTargetConfiguration]	否	-	Target服务配置，mcp_server内容必须提供

参数	类型	是否必选	默认值	说明
credential_provider_configuration	Optional[CoreGatewayCredentialProviderConfiguration]	否	-	凭证提供者配置

凭证提供者配置要求：

– 如果不提供，credential_provider_type 默认为 none，表示无认证。
– 当 credential_provider_type='oauth' 时，必须提供 oauth_credential_provider。
– 当 credential_provider_type='api_key' 时，必须提供 api_key_credential_provider。

- 查询指定Gateway下的特定Target实例的详细信息

```
def get_mcp_gateway_target(self, gateway_id: str, target_id: str) -> ShowCoreGatewayTargetResponse
```

表 7-18 查询 MCP Target 信息参数说明

参数	类型	是否必选	说明
gateway_id	str	是	Target所属Gateway的唯一标识符
target_id	str	是	要查询的Target的唯一标识符

- 更新指定Gateway下的特定Target实例

```
def update_mcp_gateway_target(
    self,
    gateway_id: str,
    target_id: str,
    name: Optional[str] = None,
    description: Optional[str] = None,
    target_configuration: Optional[CoreGatewayTargetConfiguration] = None,
    credential_provider_configuration: Optional[CoreGatewayCredentialProviderConfiguration] = None)
-> UpdateCoreGatewayTargetResponse
```

表 7-19 更新 MCP Target 参数信息

参数	类型	是否必选	说明
gateway_id	str	是	Target所属Gateway的唯一标识符
target_id	str	是	要更新的Target的唯一标识符
name	Optional[str]	否	Target服务名称

参数	类型	是否必选	说明
description	Optional[str]	否	Target服务描述
target_configuration	Optional[CoreGatewayTargetConfiguration]	否	Target服务配置
credential_provider_configuration	Optional[CoreGatewayCredentialProviderConfiguration]	否	凭证提供者配置

- 删除指定Gateway下的特定Target实例

```
def delete_mcp_gateway_target(self, gateway_id: str, target_id: str) ->
DeleteCoreGatewayTargetResponse
```

表 7-20 删除 MCP Target 参数说明

参数	类型	是否必选	说明
gateway_id	str	是	Target所属Gateway的唯一标识符
target_id	str	是	要删除的Target的唯一标识符

- 查询指定Gateway下的所有Target列表

```
def list_mcp_gateway_targets(
    self,
    gateway_id: str,
    limit: Optional[int] = None,
    offset: Optional[int] = None) -> ListCoreGatewayTargetsResponse
```

表 7-21 查询指定 Gateway 下的 Target 列表参数说明

参数	类型	是否必选	默认值	说明
gateway_id	str	是	-	要查询Targets的Gateway的唯一标识符
limit	Optional[int]	否	50	每页返回的最大结果数量
offset	Optional[int]	否	0	返回结果的偏移量

CLI 模式操作

- 创建MCP Gateway

```
agentarts mcp-gateway create-mcp-gateway
```


版权所有 © 华为云计算技术有限公司

表 7-22 创建 MCP Gateway 参数说明

参数	类型	是否必选	默认值	说明
--name	string	否	TestGateway_{随机后缀}	网关名称
--description	string	否	-	网关描述
--protocol-type	string	否	MCP	网关协议类型
--agency-name	string	否	-	用于指定网关委托身份的委托名称
--authorizer-type	string	否	IAM	授权器类型（CUSTOM_JWT、IAM、API_KEY）
--authorizer-configuration	JSON string	否	-	授权器配置的JSON字符串
--log-delivery-configuration	JSON string	否	-	日志投递配置的JSON字符串
--outbound-network-configuration	JSON string	否	-	网络配置的JSON字符串
--tags	JSON array	否	-	资源标签列表的JSON字符串

授权器配置要求：

– 当 authorizer-type = 'CUSTOM_JWT' 时，必须提供 custom_jwt_authorizer 配置。
– 当 authorizer-type = 'API_KEY' 时，必须提供 key_auth 配置。

示例：

```
agentarts mcp-gateway create-mcp-gateway \     --name "my-gateway" \     --description "测试网关" \     --authorizer-type "API_KEY" \     --authorizer-configuration '{"key_auth": {"api_keys": ["key1", "key2"]}}' \     --tags '["tag1", "tag2"]'
```

- 获取指定的MCP Gateway详情

```
agentarts mcp-gateway create-mcp-gateway
```

表 7-23 获取 MCP Gateway 详情参数说明

参数	类型	是否必选	说明
<gateway-id>	string	否	网关ID

响应信息包含：

- gateway_id: 网关唯一标识符
- name: 网关名称
- description: 网关详细描述
- status: 网关当前状态
- protocol_type: 网关协议类型（MCP）
- authorizer_type: 授权器类型
- agency_name: 委托名称
- authorizer_configuration: 授权器配置
- endpoint_url: 网关访问的URL端点
- log_delivery_configuration: 日志投递配置
- workload_identity: 工作负载身份标识符
- outbound_network_configuration: 出站网络配置
- tags: 资源标签列表
- created_at: 网关创建时间戳
- updated_at: 网关最后更新时间戳

- 更新指定的MCP Gateway实例

```
agentarts mcp-gateway update-mcp-gateway <gateway-id> [参数]
```

表 7-24 更新 MCP 网关参数说明

参数	类型	是否必选	说明
<gateway-id>	string	是	网关ID
--description	string	否	网关描述
--authorizer-configuration	JSON string	否	授权器配置的JSON字符串
--log-delivery-configuration	JSON string	否	日志投递配置的JSON字符串
--outbound-network-configuration	JSON string	否	网络配置的JSON字符串
--tags	JSON array	否	资源标签列表的JSON字符串

授权器配置要求：

- 当 authorizer-type = 'CUSTOM_JWT' 时，必须提供 custom_jwt_authorizer 配置
- 当 authorizer-type = 'API_KEY' 时，必须提供 key_auth 配置

说明

更新操作为增量更新，只传入需要更新的字段，未传入的字段将保持原值不变。

示例：

```
agentarts mcp-gateway update-mcp-gateway "gateway-123" \     --description "更新后的描述" \     --tags '["new-tag1", "new-tag2"]'
```

- 删除指定的MCP Gateway实例

```
agentarts mcp-gateway delete-mcp-gateway <gateway-id>
```

表 7-25 更新 MCP 网关参数说明

参数	类型	是否必选	说明
<gateway-id>	string	是	网关ID

响应信息包含删除操作的状态和相关操作日志信息。

- 获取MCP Gateway实例列表

```
# 列出所有网关
agentarts mcp-gateway list-mcp-gateways

# 按条件过滤
agentarts mcp-gateway list-mcp-gateways [参数]
```

表 7-26 列出 MCP 网关参数说明

参数	类型	是否必选	默认值	说明
--name	string	否	-	按名称过滤网关，支持模糊匹配
--status	string	否	-	按状态过滤网关（如"active"、"inactive"）
--gateway-id	string	否	-	按网关ID过滤，支持多个ID（最多20个），用逗号分隔
--tags	string	否	-	按标签过滤网关，支持多个标签，用逗号分隔
--offset	integer	否	0	分页偏移量，最小 0，最大 100000
--limit	integer	否	50	每页大小，最小 1，最大 100

版权所有 © 华为云计算技术有限公司

- 在指定的MCP Gateway下创建新的Target实例

```
agentarts mcp-gateway create-mcp-gateway-target <gateway-id> [参数]
```

表 7-27 创建 MCP 网关目标参数说明

参数	类型	是否必选	默认值	说明
<gateway-id>	string	是	-	网关ID
--name	string	否	TestGatewayTarget_{随机后缀}	目标名称
--description	string	否	-	目标描述
--target-configuration	JSON string	否	-	目标服务的配置JSON字符串
--credential-provider-configuration	JSON string	否	-	凭证提供者配置的JSON字符串

凭证提供者配置要求：

- 如果未提供，credential_provider_type 默认为 GATEWAY_IAM_ROLE，表示使用网关IAM角色。
- 当 credential_provider_type='none' 时，表示无认证。
- 当 credential_provider_type='oauth' 时，表示使用 OAuth 2.0，必须提供 oauth_credential_provider。
- 当 credential_provider_type='api_key' 时，表示使用 API 密钥，必须提供 api_key_credential_provider。

示例：

```
agentarts mcp-gateway create-mcp-gateway-target "gateway-123" \     --target-configuration
'{"mcp_server": {"command": "npx -y @modelcontextprotocol/server-examples echo", "env":
{"VARIABLE": "value"}}}'
```

- 获取MCP网关目标详情

```
agentarts mcp-gateway get-mcp-gateway-target <gateway-id> <target-id>
```

功能说明：此命令用于获取指定网关下特定目标的详细信息，包括目标名称、描述、状态、端点、超时时间等所有相关配置信息。如果目标不存在，系统会返回错误提示。

表 7-28 获取 MCP 网关目标详情参数说明

参数	类型	是否必选	默认值	说明
<gateway-id>	string	是	网关ID	按名称过滤网关，支持模糊匹配

版权所有 © 华为云计算技术有限公司

参数	类型	是否必选	默认值	说明
<target-id>	string	是	目标ID	按状态过滤网关（如"active"、"inactive"）

- 更新指定Gateway下的特定Target实例

```
agentarts mcp-gateway update-mcp-gateway-target <gateway-id> <target-id> [参数]
```

表 7-29 更新 MCP 网关目标参数说明

参数	类型	是否必选	说明
<gateway-id>	string	是	网关ID
<target-id>	string	是	目标ID
--name	string	否	目标名称
--description	string	否	目标描述
--target-configuration	JSON string	否	目标服务的配置JSON字符串
--credential-provider-configuration	JSON string	否	凭证提供者配置的JSON字符串

示例：

```
agentarts mcp-gateway update-mcp-gateway-target "gateway-123" "target-456" \     --description "更新后的目标描述"
```

- 删除指定Gateway下的特定Target实例

```
agentarts mcp-gateway delete-mcp-gateway-target <gateway-id> <target-id>
```

表 7-30 删除 MCP 网关目标参数说明

参数	类型	是否必选	说明
<gateway-id>	string	是	网关ID
<target-id>	string	是	目标ID

- 查询指定Gateway下的所有Target实例列表

```
agentarts mcp-gateway list-mcp-gateway-targets <gateway-id> [参数]
```

表 7-31 列出 MCP 网关目标参数说明

参数	类型	是否必选	默认值	说明
<gateway-id>	string	是	-	网关ID

版权所有 © 华为云计算技术有限公司

参数	类型	是否必选	默认值	说明
--offset	integer	否	0	分页偏移量，最小 0，最大 100000
--limit	integer	否	50	每页大小，最小 1，最大 100

前提条件

- 已开通AgentArts。
- 登录控制台获取AK/SK，详情请参考认证鉴权。
- 已安装Python，且版本不低于3.10。查看Python版本的命令示例：

```
python --version
```

快速入门

步骤1 创建网关。

```
# Client模式
gateway = client.create_mcp_gateway(
    name="my-gateway",
    description="我的第一个MCP网关"
)
gateway_id = gateway.gateway_id


# CLI模式
agentarts mcp-gateway create-mcp-gateway --name "my-gateway" --description "我的第一个MCP网关"
```

步骤2 创建目标。

```
# Client模式
target = client.create_mcp_gateway_target(
    gateway_id=gateway_id,
    target_configuration={
        "mcp_server": {
            "command": "npx -y @modelcontextprotocol/server-examples echo"
        }
    }
)


# CLI模式
agentarts mcp-gateway create-mcp-gateway-target "$gateway_id" \
    --target-configuration '{"mcp_server": {"command": "npx -y @modelcontextprotocol/server-examples
echo"}}'
```

步骤3 查询信息。

```
# Client模式
gateway_info = client.get_mcp_gateway(gateway_id)
target_info = client.get_mcp_gateway_target(gateway_id, target.target_id)


# CLI模式
agentarts mcp-gateway get-mcp-gateway "$gateway_id"
agentarts mcp-gateway get-mcp-gateway-target "$gateway_id" "$target_id"
```

----结束

版权所有 © 华为云计算技术有限公司

参数说明

- 请求参数

表 7-32 创建 MCP Gateway 参数

参数	类型	说明
name	String	网关名称，如果不提供则默认为TestGateway-{4个随机字符后缀}
description	String	网关的详细描述
protocol_type	String	协议类型，默认为"MCP"
authorizer_type	String	授权器类型，可选值：CUSTOM_JWT、IAM、API_KEY
agency_name	String	委托名称，默认为"AgentArksCoreGateway"
authorizer_configuration	Object	授权器配置对象
log_delivery_configuration	Object	日志投递配置对象
outbound_network_configuration	Object	出站网络配置对象
tags	List[String]	资源标签列表

表 7-33 Target 配置对象

参数	类型	说明
mcp_server	Object	MCP服务器配置
mcp_server.command	String	启动命令
mcp_server.env	Object	环境变量

表 7-34 凭证提供者配置对象

参数	类型	说明
credential_provider_type	String	凭证提供者类型

版权所有 © 华为云计算技术有限公司

参数	类型	说明
credential_provider	Object	凭证提供者配置

- 响应参数

表 7-35 CreateCoreGatewayResponse

字段	类型	说明
gateway_id	String	网关的唯一标识符
name	String	网关名称
description	String	网关的详细描述
status	String	网关当前状态
protocol_type	String	网关协议类型
authorizer_type	String	授权器类型
authorizer_configuration	Object	授权器配置
endpoint_url	String	网关访问的URL端点
log_delivery_configuration	Object	日志投递配置
workload_identity	Object	工作负载身份标识符
outbound_network_configuration	Object	网络配置
tags	List	资源标签列表
agency_name	String	网关委托名称
created_at	String	网关创建时间戳
updated_at	String	网关最后更新时间戳

表 7-36 ListCoreGatewaysResponse

字段	类型	说明
gateways	List[ShowCoreGatewayResponse]	网关详情列表
size	Integer	当前页返回的网关数量
total	Integer	网关总数量

表 7-37 ShowCoreGatewayTargetResponse

字段	类型	说明
target_id	String	目标唯一标识符
name	String	目标名称
description	String	目标描述
status	String	目标状态
connection_status	String	连接状态
target_configuration	Object	目标配置
credential_provider_configuration	Object	凭证提供者配置
created_at	String	创建时间戳
updated_at	String	更新时间戳

示例代码

```
from typing import Optional, List
from agentarts.wrapper.mcpgateway import MCPGatewayClient


# 初始化客户端
client = MCPGatewayClient()


# 创建MCP Gateway
gateway = client.create_mcp_gateway(
    name="production-echo-gateway",
    description="生产环境 Echo 服务网关",
    authorizer_type="IAM",
    tags=["production", "echo-service"]
)
print(f"创建的网关ID: {gateway.gateway_id}")
print(f"端点URL: {gateway.endpoint_url}")


# 更新网关描述
updated_target = client.update_mcp_gateway(
    gateway_id=gateway.gateway_id,
    description="更新后的网关描述"
)
print(f"更新后的描述: {updated_target.description}")




# 在网关下创建Target（MCP服务器类型）
target = client.create_mcp_gateway_target(
    gateway_id=gateway.gateway_id,
    name="echo-service-target",
    description="Echo服务目标",
    target_configuration={
        "mcp_server": {
            "endpoint": "https://echo.example.com",
            "server_type": "sse"
        }
```


版权所有 © 华为云计算技术有限公司

```
    }
)


# 在网关下创建Target（OpenAPI内联文档类型）
target_openapi = client.create_mcp_gateway_target(
    gateway_id=gateway.gateway_id,
    name="petstore-api",
    description="PetStore API目标",
    target_configuration={
        "openapi": {
            "payload": """
openapi: 3.0.0
info:
  title: PetStore API
  version: 1.0.0
servers:
  - url: https://petstore.example.com
paths:
  /pets:
    get:
      summary: List all pets
      responses:
        '200':
          description: A paged array of pets
"""
        }
    }
)


# 在网关下创建Target（OpenAPI OBS文档类型）
target_openapi_obs = client.create_mcp_gateway_target(
    gateway_id=gateway.gateway_id,
    name="petstore-api-obs",
    description="PetStore API目标（OBS）",
    target_configuration={
        "openapi": {
            "obs": {
                "bucket_name": "api-specs-bucket",
                "object_key": "specs/petstore.yaml"
            }
        }
    }
)
print(f"创建的目标ID: {target.target_id}")


# 查询网关信息
gateway_info = client.get_mcp_gateway(gateway.gateway_id)
print(f"网关状态: {gateway_info.status}")


# 查询目标信息
target_info = client.get_mcp_gateway_target(gateway.gateway_id, target.target_id)
print(f"目标状态: {target_info.status}")


# 列出网关下的所有目标
targets = client.list_mcp_gateway_targets(gateway.gateway_id)
print(f"网关下有 {targets.total} 个目标")


# 更新Target描述
updated_target = client.update_mcp_gateway_target(
    gateway_id=gateway.gateway_id,
    target_id=target.target_id,
    description="更新后的Echo服务目标描述"
)
```


版权所有 © 华为云计算技术有限公司

```
print(f"更新后的描述: {updated_target.description}")


# 删除Target
client.delete_mcp_gateway_target(gateway.gateway_id, target.target_id)
print("目标已删除")


# 删除Gateway
client.delete_mcp_gateway(gateway.gateway_id)
print("网关已删除")
```


版权所有 © 华为云计算技术有限公司

8 常见问题

8.1 认证鉴权

如何获取华为云 AK/SK

AK/SK认证就是使用AK/SK对请求进行签名，在请求时将签名信息添加到消息头，从而通过身份认证。

- AK(Access Key ID)：访问密钥ID。与私有访问密钥关联的唯一标识符；访问密钥ID和私有访问密钥一起使用，对请求进行加密签名。
- SK(Secret Access Key)：与访问密钥ID结合使用的密钥，对请求进行加密签名，可标识发送方，并防止请求被修改。

如果之前没有生成过AK/SK，可参考以下步骤来获取。更多详情请参考**获取AK/SK**。

1. 进入控制台首页。
2. 将鼠标移至页面右上角的用户名处，在下拉列表中单击“我的凭证”。

版权所有 © 华为云计算技术有限公司

3. 单击“访问密钥”。
4. 单击“新增访问密钥”，进入“新增访问密钥”页面。
   - 每个用户最多可创建2个访问密钥，不支持增加配额。如果您已拥有2个访问密钥，将无法创建访问密钥。
   - 如需修改访问密钥，请删除访问密钥后重新创建。
   - 为了保证历史兼容性，系统会使用访问密钥创建时间作为最近使用时间的初始值。在您使用该访问密钥时，系统将自动刷新最近使用时间。
5. 输入描述信息，单击“确定”。
6. 创建成功后，在“创建成功”弹窗中，单击“立即下载”下载密钥，并妥善保管。

版权所有 © 华为云计算技术有限公司

您可以在访问密钥列表中查看访问密钥ID（AK），在下载的.csv文件中查看访问密钥（SK）。

- 获取临时AK/SK，请参考**IAM接口文档**。
- 请及时下载保存，弹窗关闭后将无法再次获取该密钥信息，但您可重新创建新的密钥。
- 当您下载访问密钥后，可以在浏览器页面左下角打开格式为.csv的访问密钥文件，或在浏览器“下载内容”中打开。
- 为了账号安全性，建议您妥善保管并定期修改访问密钥，修改访问密钥的方法为删除旧访问密钥，然后重新生成。

如何获取沙箱工具的 API_KEY

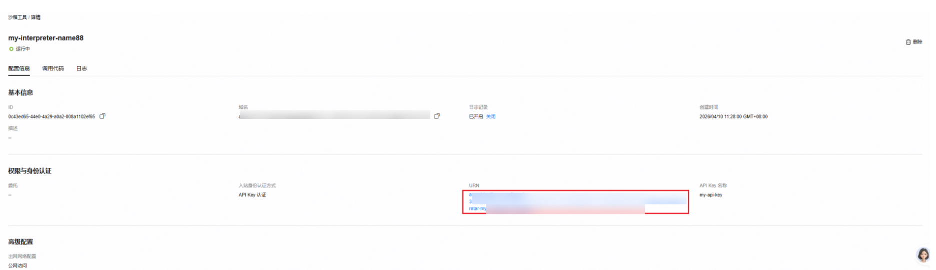
API_KEY认证就是在请求时将API_KEY信息添加到消息头，从而通过身份认证。

步骤1 登录**AgentArts智能体开发平台**。

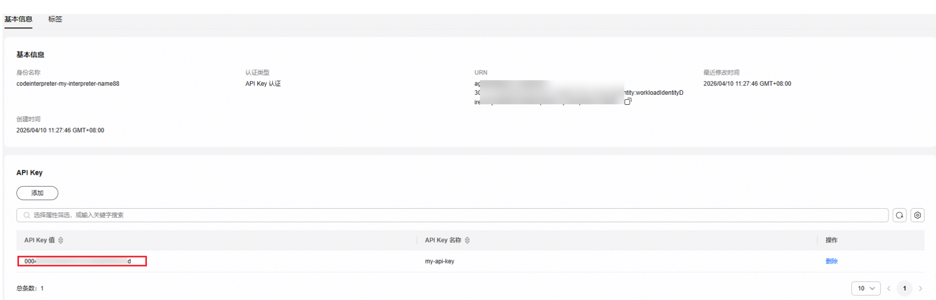
步骤2 在左侧导航栏选择“开发中心 > 组件库 ”，单击“沙箱工具”页签，进入“沙箱工具”界面。

步骤3 单击代码解释器名称，进入“配置信息”界面。

步骤4 单击URN链接，进入基本信息页签。



步骤5 在API KEY模块即可查找API KEY值。



----结束