语义路由调度 Source BUPT

子系统2负责把**上层传入的自然语言任务**（`task`）和**类别关键词**（`keyword`）经由 **Registry** 检索候选代理，按相关性分数降序选取前 K 个候选并**依次尝试直连**目标 Agent 的 `/messages` 接口（A2A 兼容），拿到标准化的 `Task` 结果返回上层。

核心职责分层：

- `registry_models.py`：Registry 请求/响应的数据模型（Pydantic）。
- `registry_client.py`：调用 Registry 的 HTTP 客户端（"写请求体、解析响应体"）。
- `remote_agent_connection.py`：对**单个**远端 Agent 的**直连发送**能力（POST `{base_url}/messages`，解析 A2A 响应）。
- `routing_agent.py`：编排器。路由（向 Registry 取候选并排序）、构造连接、发送消息、聚合/返回结果。

一次完整调用

```
上层服务 → RoutingAgent.send_message_to_registry(keyword, task, tool_context, top_k)
    → RegistryClient.list_agents(keyword, RegistryListReq)
        → Registry（/api/v1/{keyword}/list）返回候选列表（含 url/score…）
    ← RoutingAgent 取前 K 个候选，按序构造 RemoteAgentConnections(agent_url)
    → 依次调用 {url}/messages（A2A payload）
      若任一返回 A2A 成功（SendMessageSuccessResponse + Task），立即返回
    ← 若都失败，返回 None（或上抛异常，取决于你的策略）
```

1. `registry_models.py` — Registry 数据模型

定位：定义与 Registry API 对应的**强类型数据结构**，用于**请求体构建**与**响应体校验解析**。

```python
from typing import List, Optional
from pydantic import BaseModel, Field

class RegistryListReq(BaseModel):
    # POST /api/v1/{keyword}/list 的请求体
    request_id: str              # 用于链路追踪
    task: str                    # 自然语言意图/上下文
    top_k: int = Field(default=3, ge=1, le=50)  # 期望返回 K 个候选

class RegistryAgentItem(BaseModel):
    # Registry 返回的候选项
    score: float
    agent_id: str
    name: str
    description: Optional[str] = None
    url: str                     # 目标 Agent 的可调用基地址
    version: float
```

```python
class RegistryListResp(BaseModel):
    # POST /api/v1/{keyword}/list 的响应体
    status: str                 # "success" / "error"
    request_id: str
    count: int                  # 服务端声明的候选数量
    agents: List[RegistryAgentItem] = []
```

- 使用 Pydantic，构造时即校验字段范围与类型（减少脏数据下沉到路由逻辑）。

---

2. `registry_client.py` — Registry HTTP 客户端

定位：Registry 是**下游服务**。这里**发起请求并解析返回消息**。

```python
import httpx
from registry_models import RegistryListReq, RegistryListResp

class RegistryClient:
    def __init__(self, base_url: str, timeout: float = 8.0) -> None:
        self.base_url = base_url.rstrip("/")
        self.timeout = timeout

    async def list_agents(self, keyword: str, req: RegistryListReq) -> RegistryListResp:
        url = f"{self.base_url}/api/v1/{keyword}/list"
        async with httpx.AsyncClient(timeout=self.timeout, trust_env=False) as cli:
            r = await cli.post(url, json=req.model_dump())
            r.raise_for_status()  # 非 2xx 抛 HTTPStatusError
            return RegistryListResp.model_validate(r.json())  # Pydantic 解析+校验
```

要点：

- **我们写请求体**（`RegistryListReq`），**Registry 返回响应体**（解析为 `RegistryListResp`）。
- 超时、非 2xx、结构不符分别抛出 `httpx.TimeoutException`、`httpx.HTTPStatusError`、`pydantic.ValidationError`，留给上层统一处理。
- `trust_env=False` 避免代理，可按需开启。

---

3. `remote_agent_connection.py` — 直连远端 Agent

定位：给一个具体的 Agent 基地址（`base_url`），把**A2A 的消息**发到 `{base_url}/messages` 并解析**A2A 响应**。

```python
import httpx
```

```python
from a2a.types import SendMessageRequest, SendMessageResponse

class RemoteAgentConnections:
    def __init__(self, agent_url: str, timeout: float = 30.0) -> None:
        if not agent_url:
            raise ValueError("agent_url is required for direct connection.")
        self.base_url = agent_url.rstrip("/")
        self._httpx = httpx.AsyncClient(timeout=timeout, trust_env=False)

    async def send_message(self, message_request: SendMessageRequest) ->
SendMessageResponse:
        url = f"{self.base_url}/messages"
        payload = message_request.params.model_dump()  # {'message': {...}}
        resp = await self._httpx.post(url, json=payload)
        resp.raise_for_status()
        return SendMessageResponse.model_validate(resp.json())

    async def aclose(self) -> None:
        await self._httpx.aclose()
```

协议约定：

- 请求：遵循 A2A `/messages` 输入结构（`message.role/parts/messageId/contextId/...`）。
- 响应：可被 `a2a.types.SendMessageResponse` 解析；成功时应为 `SendMessageSuccessResponse` 且含 `Task`。

---

4. `routing_agent.py` — 编排器（路由 + 发送 + 解析）

定位：子系统2的**核心编排**：

- 组装 `RegistryListReq` → 调用 Registry → 取候选（排序/裁剪）。
- 按分数降序**依次**尝试直连目标 Agent 的 `/messages`。
- **首个成功**即返回 `Task`；全失败则返回 `None`（或可选上抛）。

关键接口：

```python
class RoutingAgent:
    def __init__(self, registry_base_url: Optional[str] = None) -> None:
        self.registry = RegistryClient(registry_base_url or os.getenv("REGISTRY_BASE_URL",
"http://localhost:18080"))
        self._connections: dict[str, RemoteAgentConnections] = {}

    async def resolve_client(self, keyword: str, task: str, top_k: int = 3) ->
list[tuple[str, RemoteAgentConnections]]:
        # 1) 构造请求体
        req = RegistryListReq(request_id=f"req-{uuid.uuid4()}", task=task, top_k=top_k)
        # 2) 调 Registry（解析响应体）
```

```
        resp: RegistryListResp = await self.registry.list_agents(keyword, req)
        if resp.status != "success" or not resp.agents:
            raise LookupError(...)

        # 3) 排序（降序）
        agents_sorted = sorted(resp.agents, key=lambda a: a.score, reverse=True)
        count = len(agents_sorted)

        if top_k >= count:
            raise ValueError(f"top_k ({top_k}) must be < count ({count})")

        # 4) 构造前 K 个直连连接
        results: list[tuple[str, RemoteAgentConnections]] = []
        for item in agents_sorted[:top_k]:
            conn = RemoteAgentConnections(agent_url=item.url)
            results.append((item.name, conn))
            self._connections[item.name] = conn
        return results

    async def send_message_to_registry(self, keyword: str, task: str, tool_context:
ToolContext, top_k: int = 3) -> Optional[Task]:
        state = tool_context.state
        candidates = await self.resolve_client(keyword, task, top_k)

        # 固定一次 context_id 用于多次尝试
        context_id = state.get("context_id") or str(uuid.uuid4())
        state["context_id"] = context_id
        input_meta = state.get("input_message_metadata") or {}

        errors: list[str] = []
        for agent_name, connection in candidates:
            state["active_agent"] = agent_name
            message_id = input_meta.get("message_id") or uuid.uuid4().hex
            payload: dict[str, Any] = {
                "message": {
                    "role": "user",
                    "parts": [{"type": "text", "text": task}],
                    "messageId": message_id,
                    "contextId": context_id,
                }
            }
            req = SendMessageRequest(id=message_id,
params=MessageSendParams.model_validate(payload))

            try:
                resp: SendMessageResponse = await connection.send_message(req)
            except Exception as e:
                errors.append(f"{agent_name}: request failed ({e})")
                continue

            if not isinstance(resp.root, SendMessageSuccessResponse):
                errors.append(f"{agent_name}: non-success response")
```

```
                continue
            if not isinstance(resp.root.result, Task):
                errors.append(f"{agent_name}: success wrapper but no Task")
                continue

            return resp.root.result

        # 全部失败
        if errors:
            print("All candidates failed:\n" + "\n".join(errors))
        return None
```

最小示例（从上层调用）

```
import asyncio, uuid
from google.adk.tools.tool_context import ToolContext
from routing_agent import RoutingAgent

async def main():
    ra = RoutingAgent()  # 读取 REGISTRY_BASE_URL
    ctx = ToolContext(state={"request_id": f"req-{uuid.uuid4()}"})

    task = "weather in LA, CA for the next 3 days"
    result_task = await ra.send_message_to_registry(
        keyword="weather",
        task=task,
        tool_context=ctx,
        top_k=3,
    )

    if result_task:
        print("Task OK:", result_task.id)
    else:
        print("All candidates failed or non-success response")

if __name__ == "__main__":
    asyncio.run(main())
```