云原生 AI · AI Lab

AI Lab 用户手册

开发控制台 · 任务管理 · 数据管理 · 模型服务 · 运维管理

全流程使用指南

文档名称	AI Lab 用户手册
产品名称	AI Lab（模块代号 baize，曾用名智能算力）
所属平台	DaoCloud Enterprise（DCE）云原生 AI
产品定位	软硬一体的 AI 智算平台，整合异构算力、统一调度 GPU 资源，并提供优化的 AI 开发框架
内容范围	介绍与安装、快速入门、Notebook 开发环境、任务管理、数据管理、模型服务、运维管理、最佳实践、故障排查、升级注意事项与 Release Notes
整理来源	https://docs.daocloud.io/drun/baize/intro/
更新时间	2026 年 9 月

本手册基于 DaoCloud 官方文档中心「云原生 AI → AI Lab」全站页面（52 个页面）整理，供开发者与平台管理员参考。

目录

一、AI Lab 介绍

AI Lab 是 DaoCloud 云原生 AI 体系中的 AI 算力平台，面向 MLOps / LLMOps 工程师与平台管理员，提供从开发环境、任务训练到模型推理的一站式能力。本章介绍 AI Lab 的产品定位、功能特性、组件安装方式，以及首次使用时的基本操作。

（一）什么是 AI Lab

AI Lab 是 DaoCloud 推出的基于云原生操作系统的 AI 算力平台（曾用名智能算力）。AI Lab 提供软硬一体的 AI 智算体验，整合异构算力，优化 GPU 性能，实现算力资源统一调度和运营，最大化算力效用并降低算力开销，并且还提供了优化的 AI 开发框架，简化 AI 开发和部署，加速推动各行业的 AI 应用场景落地。

功能特性

特性	说明
算力资源全托管	依托于 DCE（DaoCloud Enterprise），提供强大的基础设施能力，支持超大规模算力集群、异构 GPU 等一站式托管，并提供一系列如 vGPU 等软硬一体加速方案。
数据编排	支持模型开发过程中数据管理与编排能力，提供如数据集管理、多数据源接入、数据集预热等功能，从底层容器存储引擎进行优化，保证数据的高效与稳定。
开发环境管理	满足 MLOps 和 LLMOps 工程师对开发环境的需求，提供多种开发环境，包括 JupyterLab、CodeServer，支持自定义开发环境，一键挂载各种 GPU、数据集等资源。
任务管理	支持训练任务的全生命周期管理，提供多种快速创建任务的方式；支持 Pytorch、TensorFlow、PaddlePaddle 等主流任务框架，天然支持单机、分布式、多节点、多卡等多种类任务调度。
GPU 管理	可以查看全部的 GPU 资源和 GPU 使用情况，支持 GPU 中当前和历史运行的任务情况查看，方便进行 GPU 压力评估。
队列管理	支持创建队列，并将队列与工作空间进行关联，保障在各个集群中的队列资源的统筹与隔离。

产品逻辑架构

图 1　AI Lab 产品逻辑架构

（二）安装 AI Lab 组件

在 DCE 的安装器 v0.17.0 之后商业版安装时可以同步安装 AI Lab 组件，**无需自行安装**；请联系交付支持团队获取商业版安装包。

在全局服务集群

> AI Lab 模块仅需安装在**全局服务集群**。

打开全局服务集群，然后在 **Helm 应用** -> **Helm 模板** 找到 `baize` 执行安装步骤。

> **注意事项**
> 在配置 UI 或 YAML 时要注意：
> • 命名空间为 baize-system
> • 替换环境地址后打开 <YOUR_DCE_HOST>/kpanda/clusters/kpanda-global-cluster/helm/charts/addon/baize
> • kpanda-global-cluster 是全局服务集群名称

在 baize-agent 工作集群

> 如果在 AI Lab 中对应的集群无法选择或提示缺少 baize-agent，也就是该工作集群的组件并未成功部署。

在每个有算力资源的工作集群内，需要部署对应的基础组件，主要组件包含：

- `gpu-operator`：初始化集群中的 GPU 资源，**这部分会因 GPU 资源类型安装方式不同**，详情参考「GPU 管理」文档。
- `insight-agent`：可观测组件，用于采集集群的基础设施信息，包含日志、指标、事件。

- baize-agent：包含了 AI Lab 的核心组件，调度、监控、Pytorch、Tensorflow 等算力组件。
- 【可选】nfs 存储服务，用于数据集的预热。

以上组件必须安装，否则会导致功能使用不正常。

界面化安装 baize-agent

baize-agent 需要在工作集群部署。按照下方提示，进入工作集群，然后在 **Helm 应用** -> **Helm 模板** 找到 baize-agent 执行安装步骤。

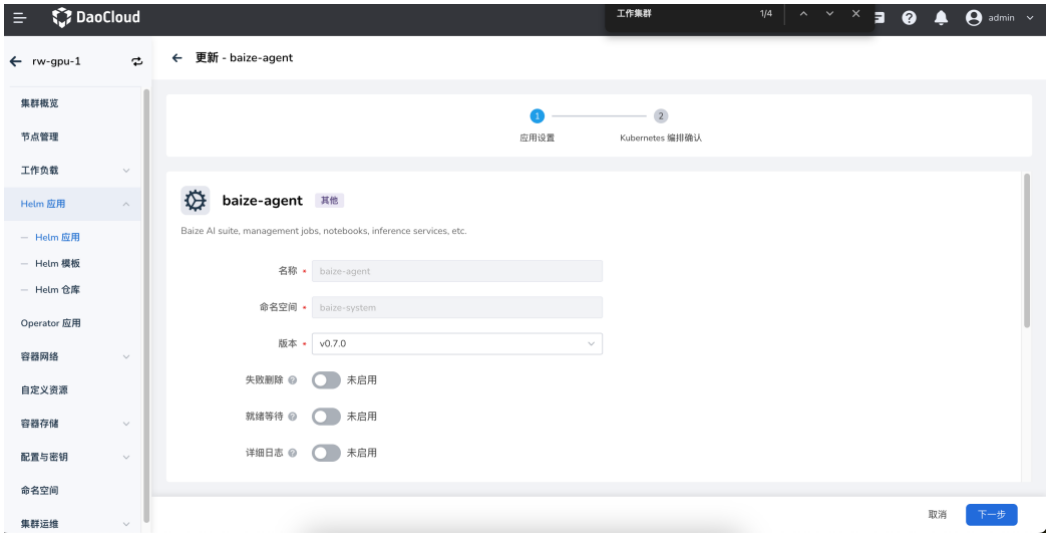

图 2　在 Helm 模板中安装 baize-agent

注意事项

在配置 UI 或 YAML 时要注意：

- 命名空间为 baize-system
- 替换环境地址后打开 <YOUR_DCE_HOST>/kpanda/clusters/<cluster_name>/helm/charts/addon/baize
- cluster_name 是对应工作集群的名称

YAML 示例：

```
cluster-controller:
  image:
    registry: ''
    repository: baize/baize-cluster-controller
    tag: v0.4.1
global:
  cluster:
    schedulers: []
  config:
    cluster_name: ''
    dataset_job_spec: {}
    inference_config:
      triton_image: m.daocloud.io/nvcr.io/nvidia/tritonserver:24.01-py3
      triton_images_map:
        VLLM: m.daocloud.io/nvcr.io/nvidia/tritonserver:24.01-vllm-python-py3
  debug: false
  high_available: false
```

```
  imagePullPolicy: IfNotPresent
  imagePullSecrets: []
  imageRegistry: release.daocloud.io
  prod: baize-agent
  resources: {}
kubeRbacProxy:
  image:
    registry: ''
    repository: baize/kube-rbac-proxy
    tag: v0.8.0
kueue:
  enablePlainPod: false
  fullnameOverride: kueue
  image:
    registry: ''
    repository: baize/kueue
    tag: v0.6.2
loader:
  image:
    registry: ''
    repository: baize/baize-data-loader
    tag: v0.4.1
notebook:
  image:
    registry: ''
    repository: baize/baize-notebook
    tag: v0.4.1
notebook-controller:
  image:
    registry: ''
    repository: baize/notebook-controller
    tag: v1.8.0
priority:
  high:
    value: 100000
  low:
    value: 1000
  medium:
    value: 10000
training-operator:
  image:
    registry: ''
    repository: baize/training-operator
    tag: v1-5525468
```

Helm 安装 baize-agent

确保全局服务集群内已经安装了 AI Lab 组件，可以通过在管理界面查看是否有 AI Lab 模块。

> 需要在一级导航栏有 AI Lab 入口，保障管理组件部署成功。

```
# baize 是 AI Lab 组件的开发代号
helm repo add baize https://release.daocloud.io/chartrepo/baize
helm repo update baize
helm search repo baize # 获取最新的版本编号
export VERSION=<version> # 注意使用当前最新版本
helm upgrade --install baize-agent baize/baize-agent \
    --create-namespace \
    -n baize-system \
    --set global.imageRegistry=release.daocloud.io \
    --version=${VERSION}
```

以上工作完成后，工作集群初始就成功了，可以在 AI Lab 模块，进行任务训练和模型开发。

预热组件介绍

AI Lab 模块提供的数据管理中，数据集的预热能力依赖存储服务，推荐使用 NFS 服务：

- 部署 NFS Server
  - 如果已存在 NFS 可以跳过此步骤；
  - 如果不存在，可以参考「部署 NFS 做数据集预热」。
- 部署 `nfs-driver-csi`
- 部署 `StorageClass`

以上完成后，就可以在工作集群内正常体验 AI Lab 的全部功能了。

（三）首次使用

第一次进入 DCE AI Lab 时，你需要了解如何选择工作空间、选择集群以及平台提供的角色。

另请参阅：

- 开发控制台快速入门（见第二章）
- 平台管理员运维管理（见第七章）
- AI Lab 最佳实践（见第八章）
- AI Lab 故障排查（见第九章）

选择工作空间

首次进入 DCE AI Lab 时，首先必须选择一个工作空间。

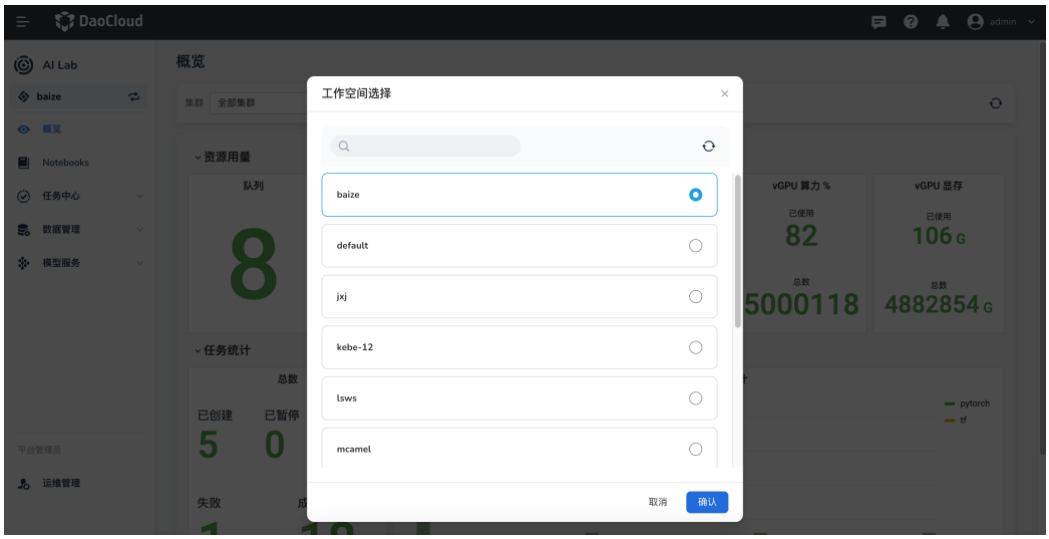

图 3　选择工作空间

如需更改当前所在的工作空间，可以在左侧边栏点击更换图标重新选择工作空间。

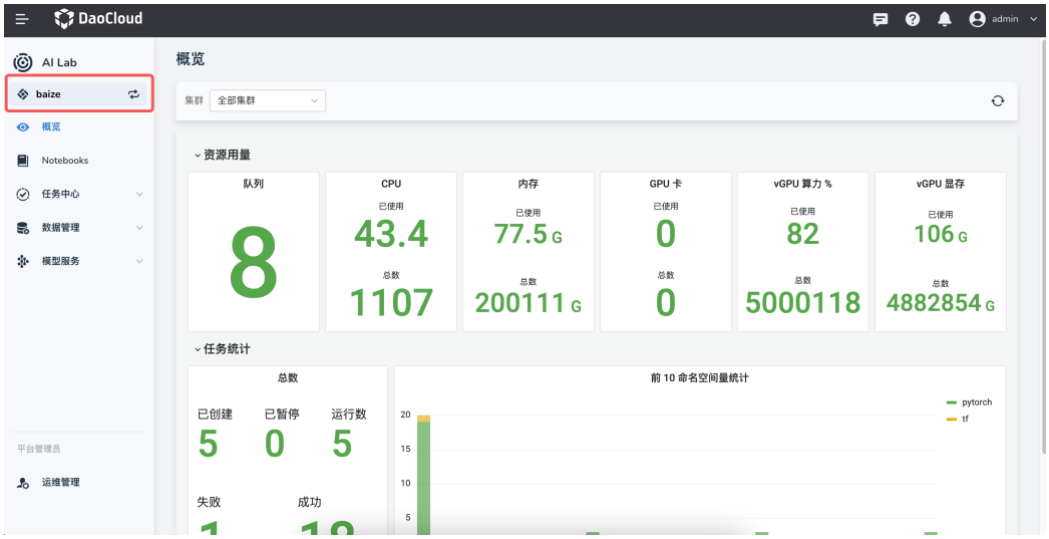

图 4　更改工作空间

> 如果当前没有可选的工作空间，需要先联系管理员创建一个工作空间。

选择集群

您可以选择在哪个集群部署和执行 AI Lab 相关的操作。

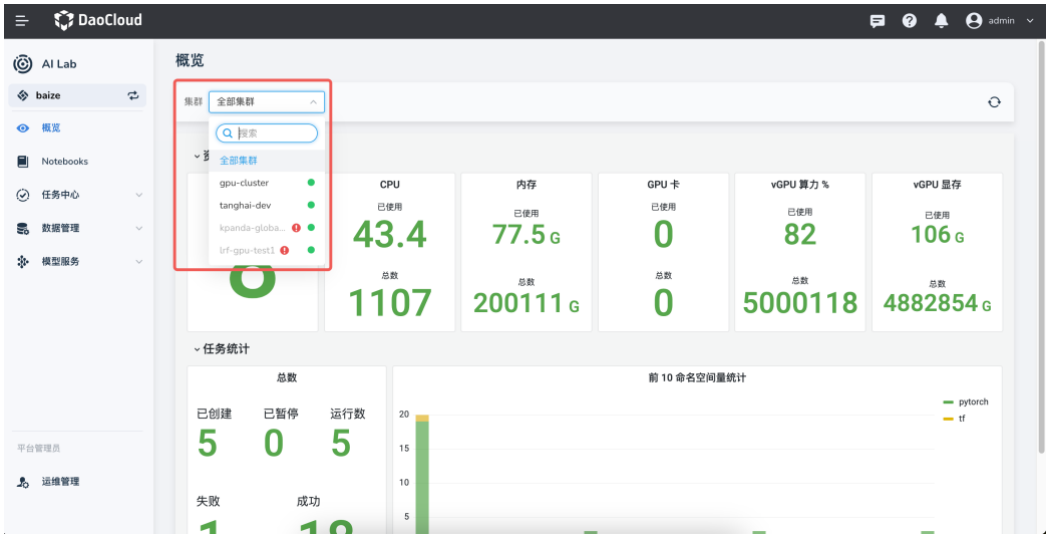

图 5　选择集群

如果您想要的集群不在集群列表中，需要先去绑定集群和命名空间。

1. 点击左上角的图标，打开导航栏，点击 **全局管理** -> **工作空间与层级**。

图 6　打开导航栏

2. 从列表中找到你的工作空间（例如 `baize`），点击 **资源组** 页签，点击右侧的 **绑定资源** 按钮。

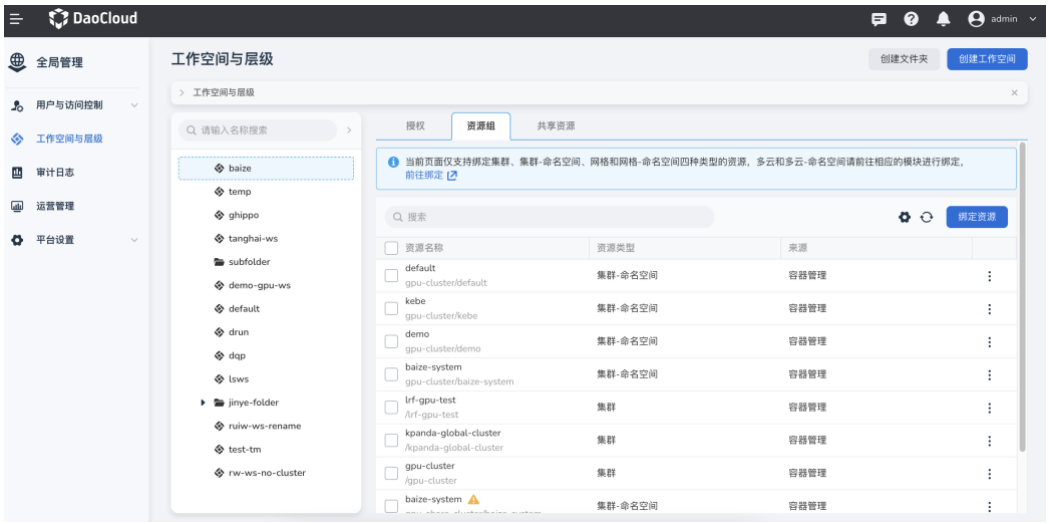

图 7　点击绑定资源

3. 勾选要绑定的命名空间、集群后，点击 **确定**。

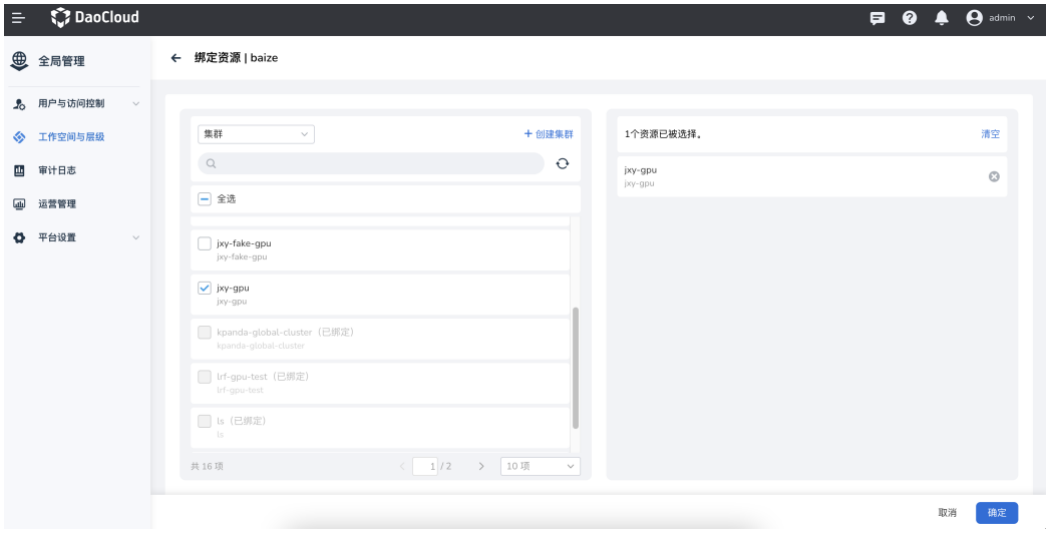

图 8　勾选命名空间与集群

4. 屏幕提示绑定成功，被绑定的集群和命名空间将显示在列表中。您还可以在这个页面授权给某个用户，添加更多资源组，创建共享资源等。

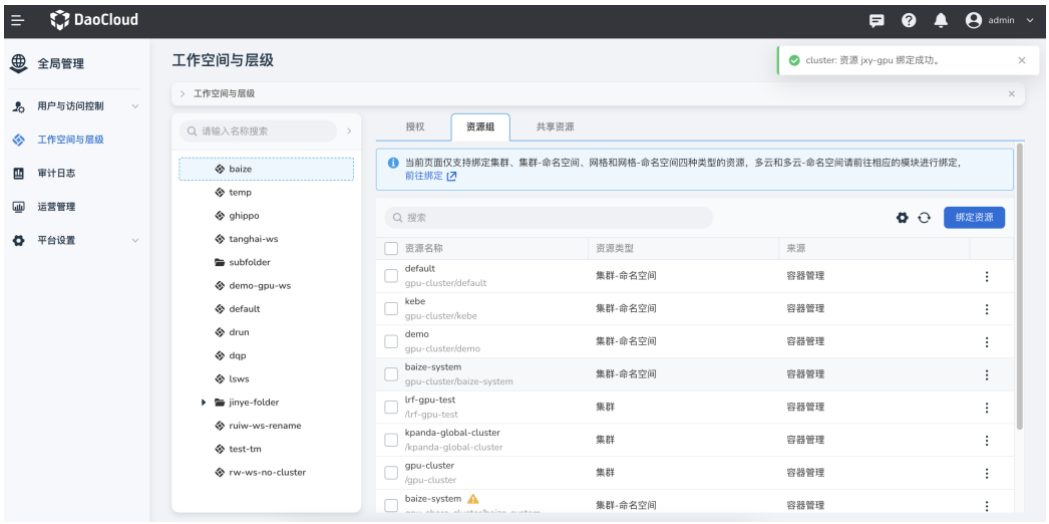

图 9　资源组列表

角色

DCE AI Lab 提供了两种角色，可以点击左下角的菜单项切换两种管理员角色：

- **管理员 - 开发控制台**：可以处理 Notebook、训练任务和数据集等。参阅第二章 开发控制台快速入门。
- **平台管理员 - 运维管理**：可以管理 GPU 资源、队列等。参阅第七章 运维管理。

每个角色都有一个概览页面，通过图形仪表展示当前可以处理的数据。

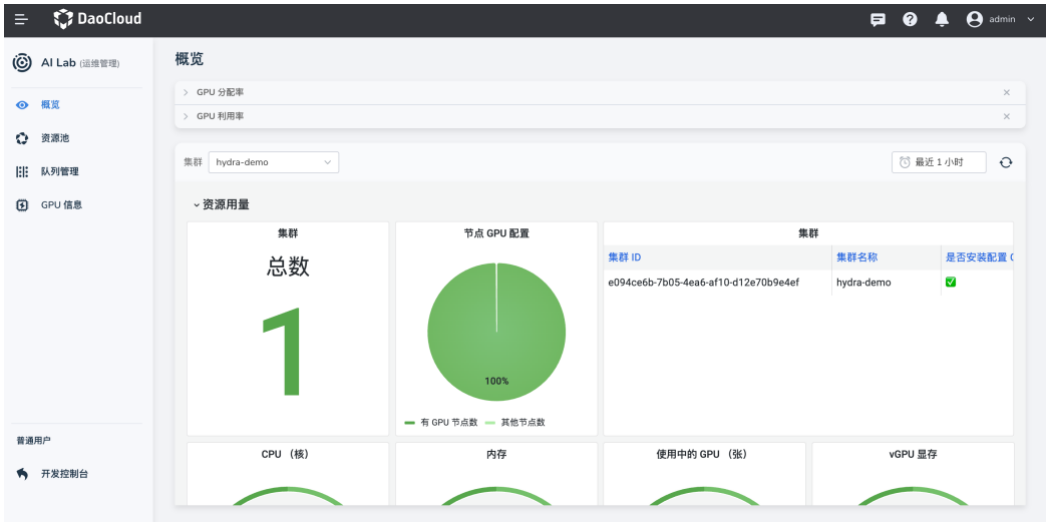

图 10　运维概览

二、开发控制台快速入门

本章面向日常执行 AI 推理、大模型训练等任务的开发者，介绍开发控制台的概览信息，并通过一个完整示例串起「数据集 → 环境依赖 → Notebook 调试 → 训练任务 → 任务分析」的开发流程。

（一）开发控制台概览

开发控制台是开发者日常执行 AI 推理、大模型训练等任务的控制台。

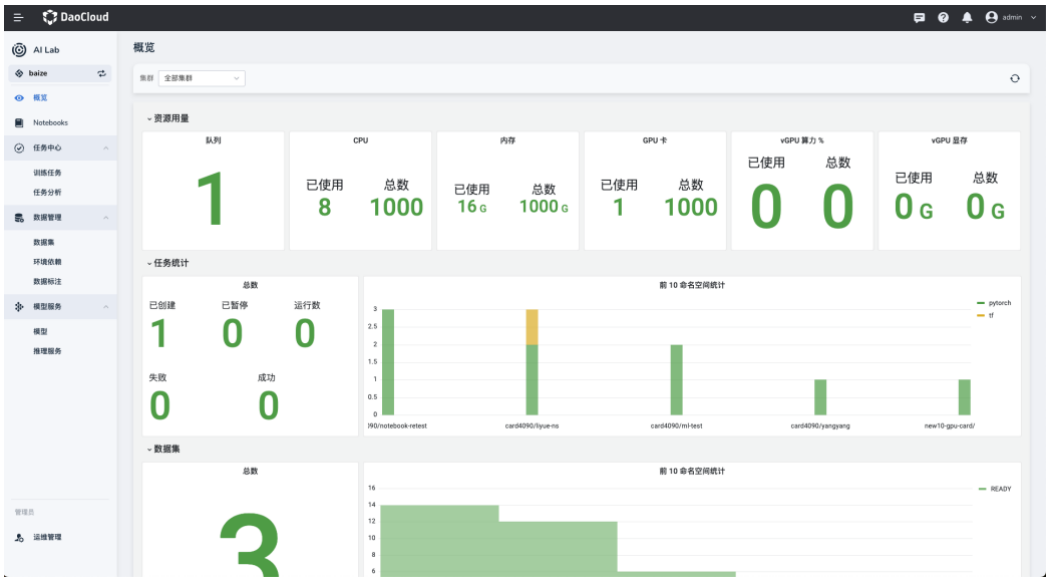

图 11　开发控制台概览

方便用户通过概览快速了解，当前工作空间的资源及用量情况，包含了 GPU 资源、Notebook、任务以及数据集的数量信息。

（二）快速入门：从数据集到训练任务

本节提供了简单的操作手册以便用户使用 DCE AI Lab 进行数据集、Notebook、任务训练的整个开发、训练流程。

准备数据集

点击 **数据管理** -> **数据集**，选择 **创建** 按钮，分别创建以下三个数据集。

1. 数据集：训练代码

- 代码数据源：`https://github.com/samzong/training-sample-code.git`，主要是一个简单的 Tensorflow 代码。
- 如果是中国境内的用户，可以使用 Gitee 加速：`https://gitee.com/samzong_lu/training-sample-code.git`
- 代码路径为 `tensorflow/tf-fashion-mnist-sample`

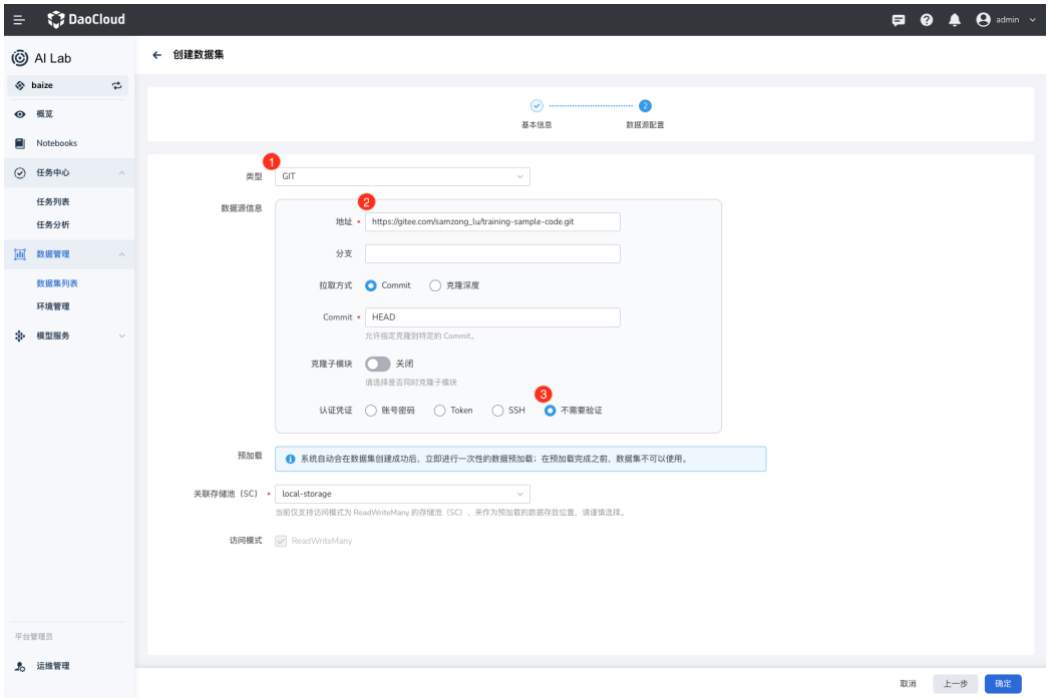

图 12　创建「训练代码」数据集

> 目前仅支持读写模式为 ReadWriteMany 的 StorageClass，请使用 NFS 或者推荐的 JuiceFS。

2. 数据集：训练数据

本次训练使用的数据为 `https://github.com/zalandoresearch/fashion-mnist.git`，这是 Fashion-MNIST 数据集。

如果是中国境内的用户，可以使用 Gitee 加速：`https://gitee.com/samzong_lu/fashion-mnist.git`

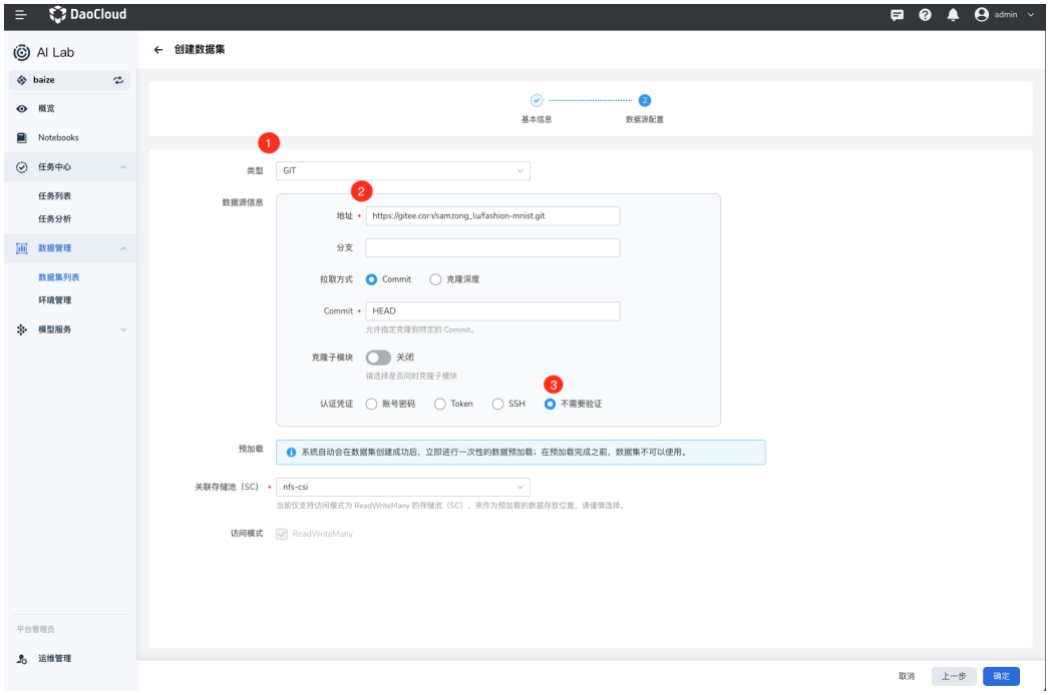

图 13　创建「训练数据」数据集

> 如果未创建训练数据的数据集，通过训练脚本也会自动下载；提前准备训练数据可以提高训练速度。

3. 数据集：空数据集

AI Lab 支持将 PVC 作为数据集的数据源类型，所以创建一个空 PVC 绑定到数据集后，可将空数据集作为存放后续训练任务的输出数据集，存放模型和日志。

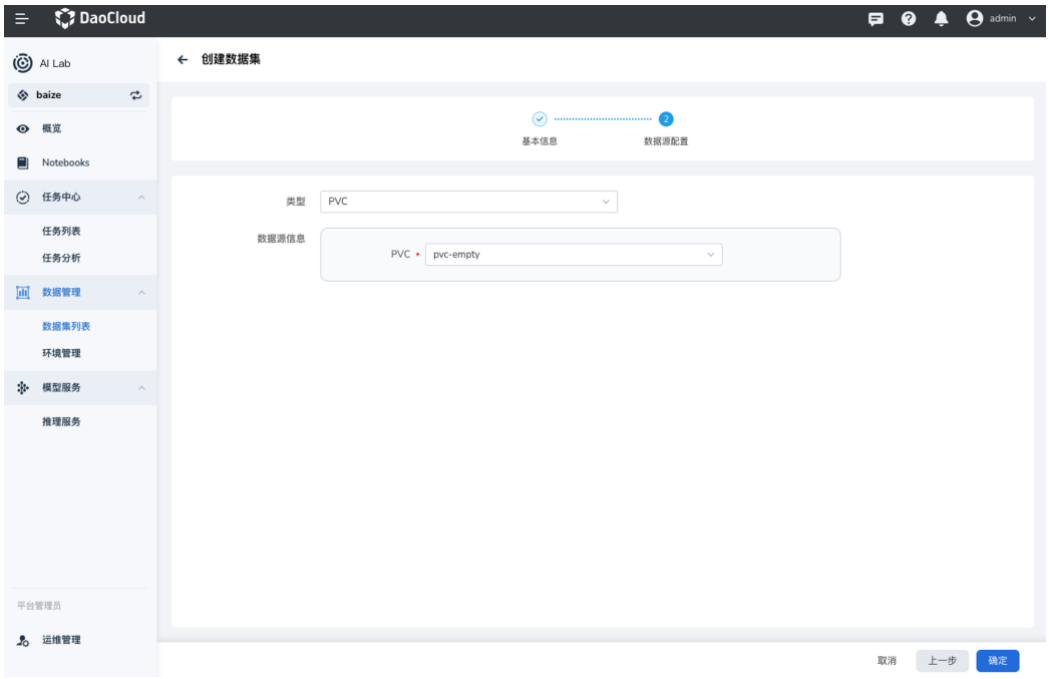

图 14　创建空数据集（PVC 类型）

环境依赖：Tensorflow

脚本在运行时，需要依赖 Tensorflow 的 Python 库，可以使用 AI Lab 的环境依赖管理功能，提前将需要的 Python 库下载和准备完成，无需依赖镜像构建。

参考「环境依赖」的操作方式，添加一个 CONDA 环境。

```
name: tensorflow
channels:
  - defaults
  - conda-forge
dependencies:
  - python=3.12
  - tensorflow
prefix: /opt/conda/envs/tensorflow
```

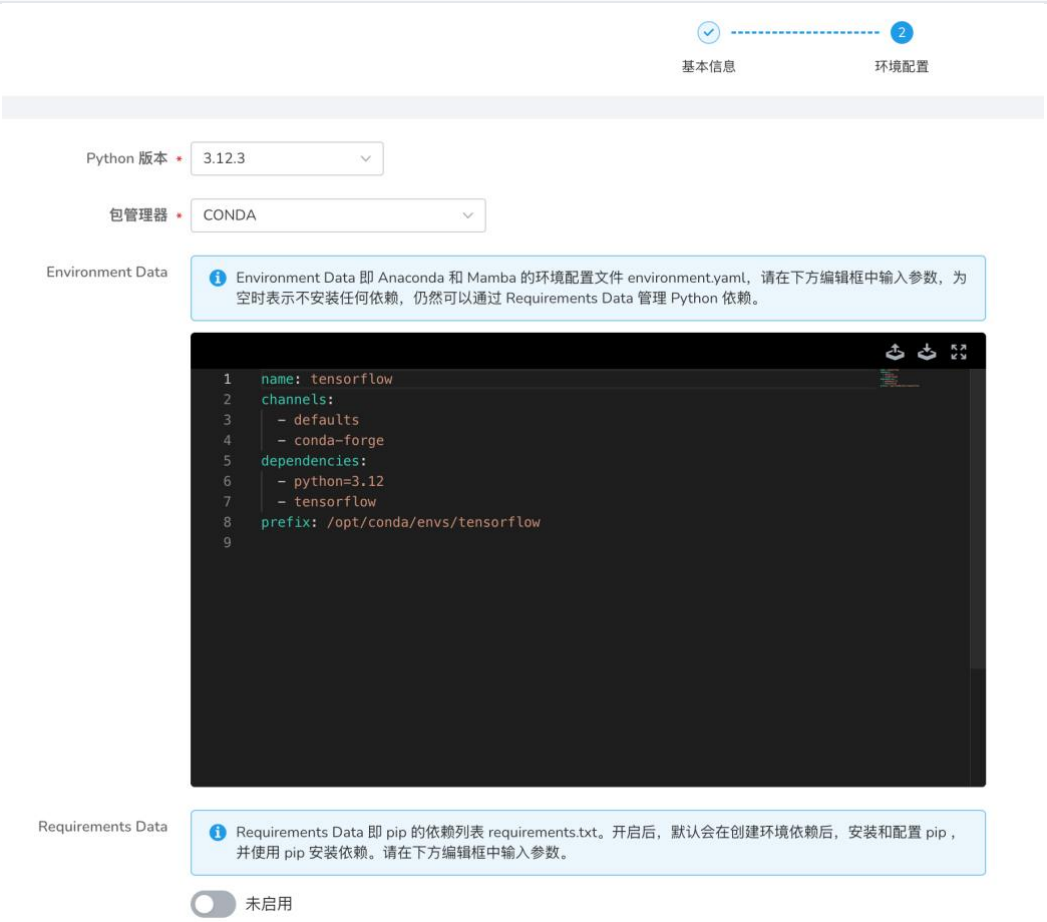

图 15　创建环境依赖

> 等待环境预热成功后，只需要将此环境挂载到 Notebook、训练任务中，使用 AI Lab 提供的基础镜像即可。

使用 Notebook 调试脚本

准备开发环境，点击导航栏的 Notebooks，点击 **创建**。

1. 将准备好的三个数据集进行关联，挂载路径请参照下图填写，注意需要配置好要使用的空数据集。

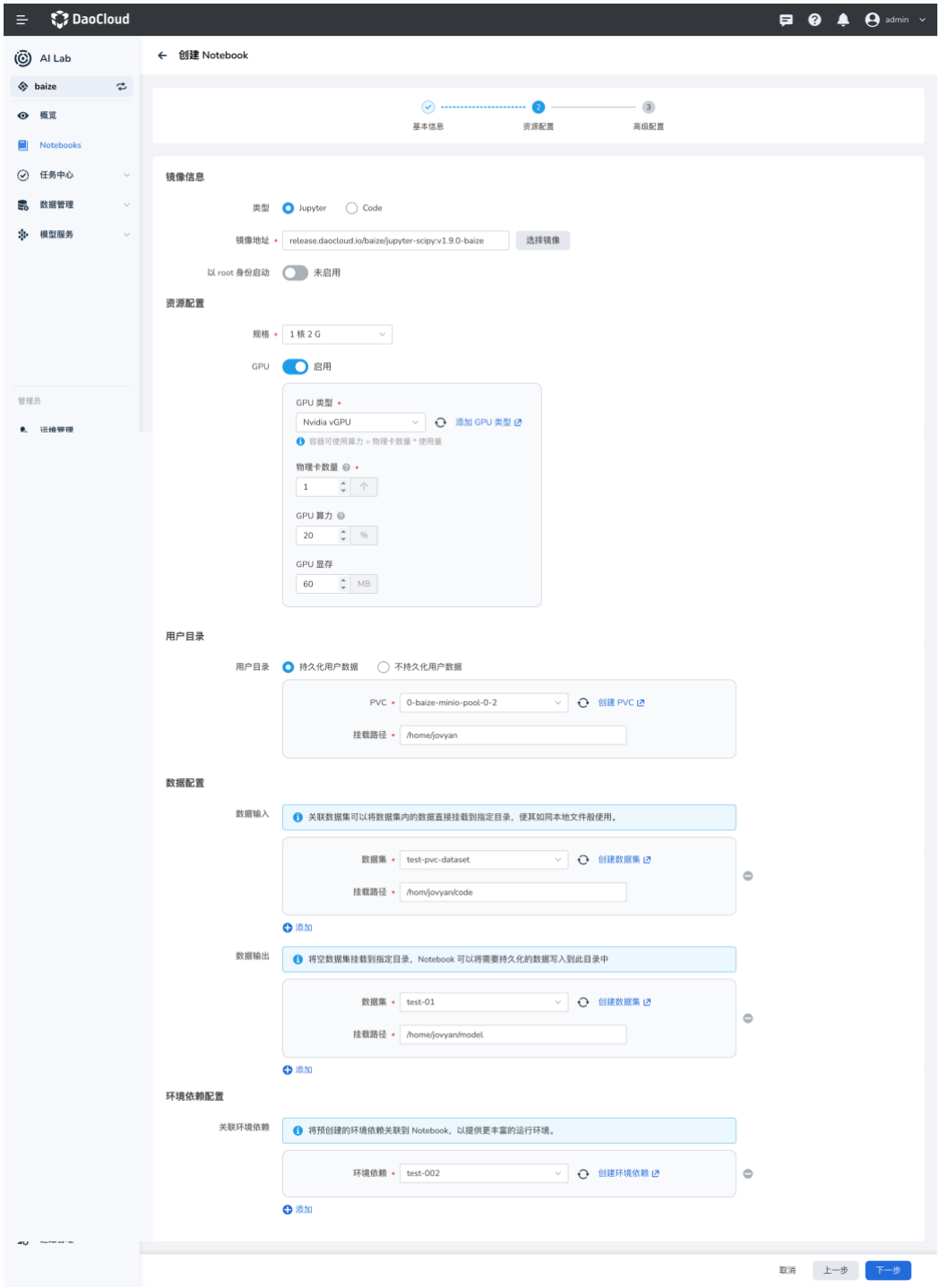

图 16　关联并挂载三个数据集

2. 选择并绑定环境依赖包，等待 Notebook 创建成功，点击列表中的访问地址，进入 Notebook。并在 Notebook 的终端中执行以下命令进行任务训练。

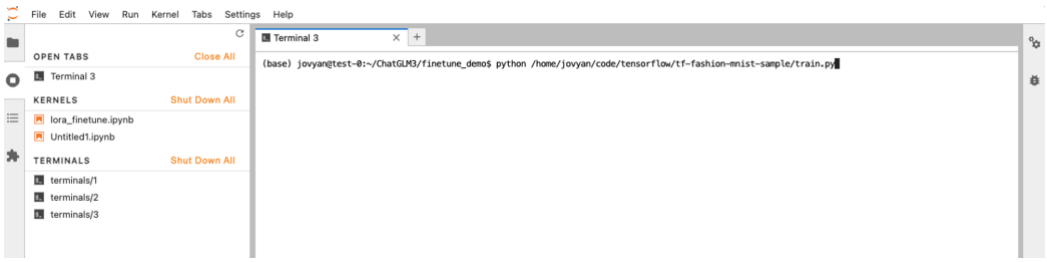

图 17　进入 Notebook

脚本使用 Tensorflow，如果忘记关联依赖库，也可以临时用 pip install tensorflow 安装。

```
python /home/jovyan/code/tensorflow/tf-fashion-mnist-sample/train.py
```

创建训练任务

1. 点击导航栏的 **任务中心** -> **训练任务**，创建一个 Tensorflow 单机任务。
2. 先填写基本参数后，点击 **下一步**。
3. 在任务资源配置中，正确配置任务资源后，点击 **下一步**：

- 镜像：如果前序环境依赖包准备好了，使用默认镜像即可；如果未准备，要确认镜像内有 `tensorflow` 的 Python 库。
- shell：使用 `bash` 即可。
- 启用命令：

```
python /home/jovyan/code/tensorflow/tf-fashion-mnist-sample/train.py
```

4. 在高级配置中，启用 **任务分析（Tensorboard）**，点击 **确定**。

日志所在位置为输出数据集的 /home/jovyan/model/train/logs/。

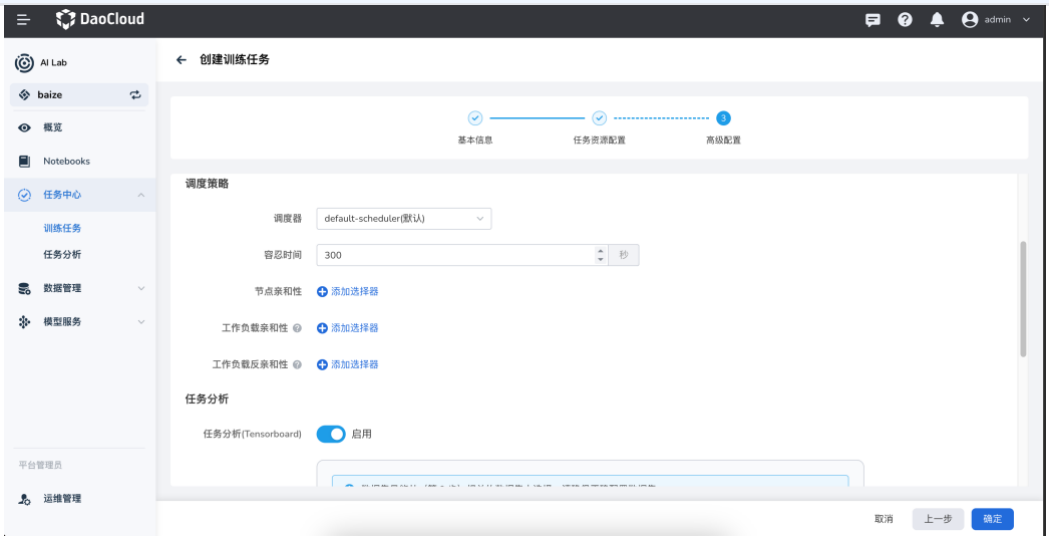

图 18　在高级配置中启用任务分析

5. 返回训练任务列表，等到状态变为 **成功**。点击列表右侧的 ⋮，可以查看详情、克隆任务、更新优先级、查看日志和删除等操作。

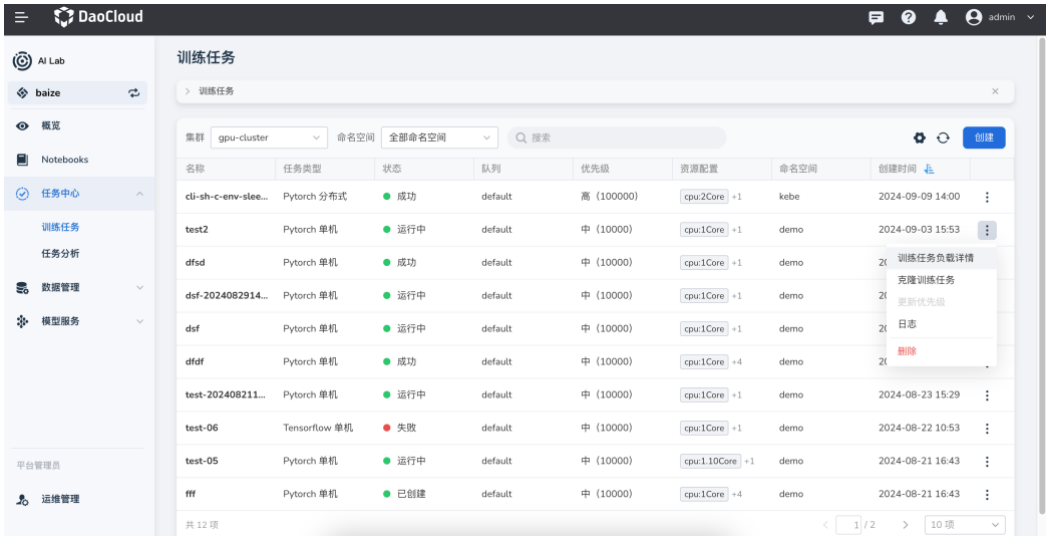

图 19　任务列表中的更多操作

6. 成功创建任务后，在左侧导航栏点击 **任务分析**，可以查看任务状态并对任务训练进行调优。

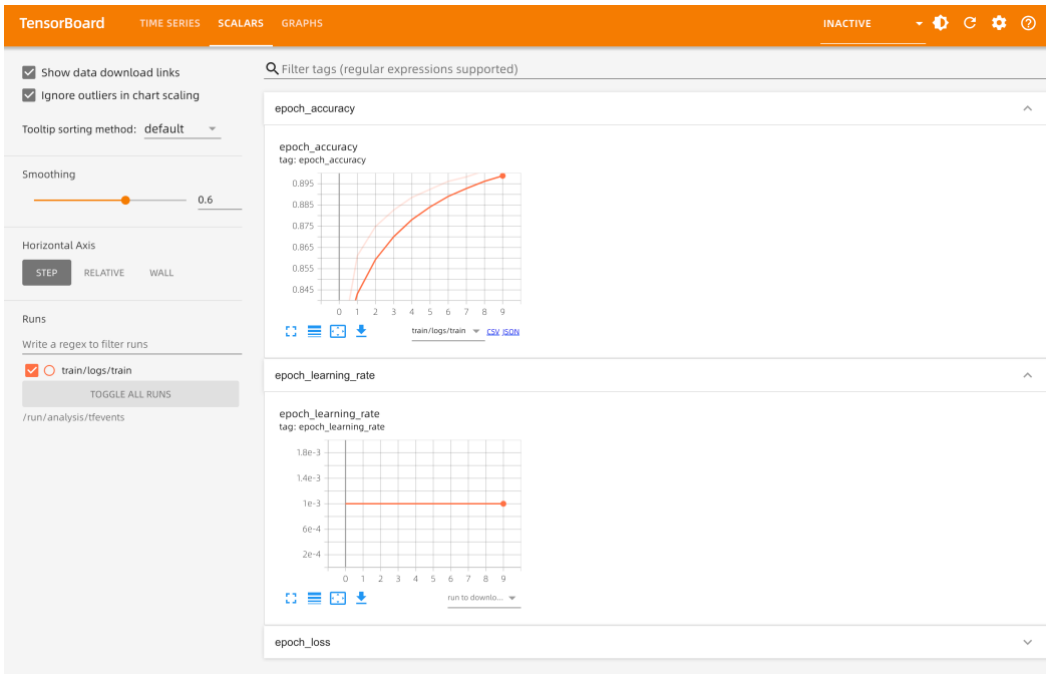

图 20　通过任务分析查看训练过程

三、Notebook 开发环境

Notebook 是 AI Lab 提供的在线交互式编程环境，集成了 JupyterLab 与 CodeServer，支持挂载 GPU、数据集、环境依赖，并可通过 SSH、Docker 等方式扩展开发体验。本章介绍 Notebook 的完整使用方法。

（一）创建 Notebook

Notebook 提供了一个在线交互式编程环境，方便开发者快速进行数据科学和机器学习实验。为迎接 AI 浪潮，AI Lab 已集成了当前最流行的 Jupyter Notebook 数据科学交互式开发环境（IDE）。

您进入 AI Lab 开发者控制台后，可以在不同集群、命名空间中创建和管理 Notebook。

1. 在左侧导航栏中点击 **Notebooks**，进入 Notebook 列表。点击右侧的 **创建** 按钮。

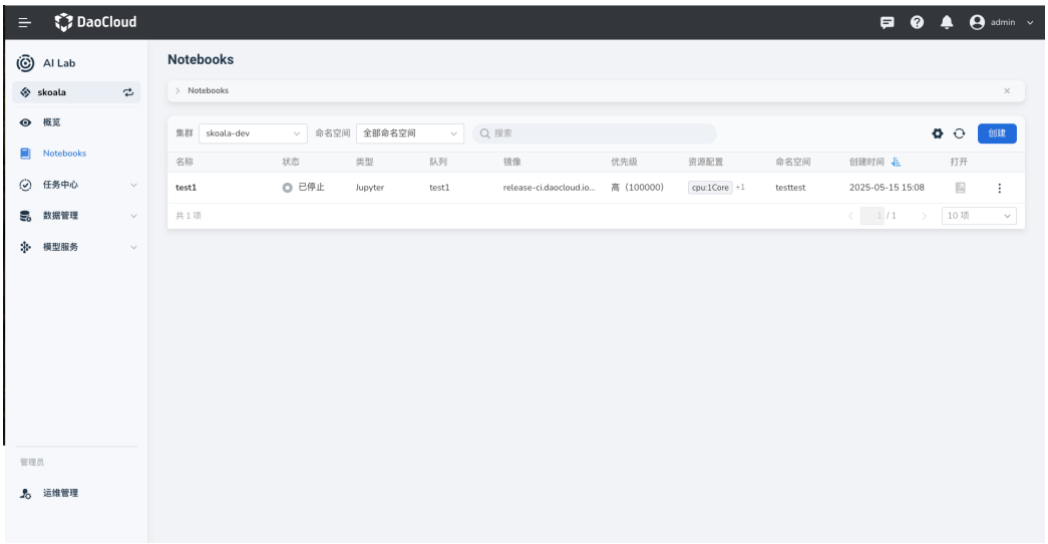

图 21　Notebook 列表点击创建

2. 系统会预先填充基础配置数据，包括要部署的集群、命名空间、队列和优先级。

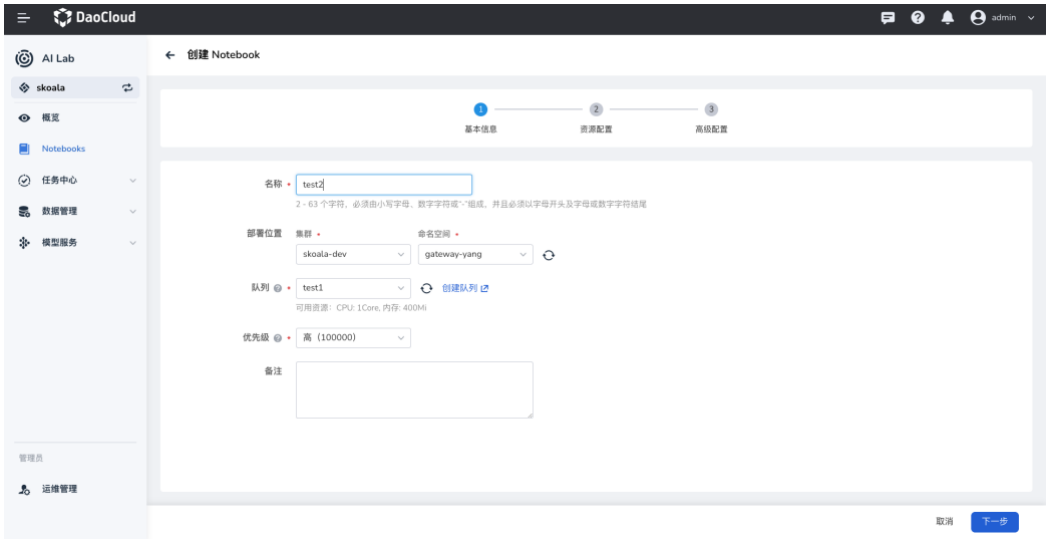

图 22　填写基础配置数据

3. 在 **资源配置** 中，系统预先填充部分配置信息，包括 Notebook 镜像类型、镜像地址、资源规格、用户目录。也可手动添加数据配置和环境依赖配置。

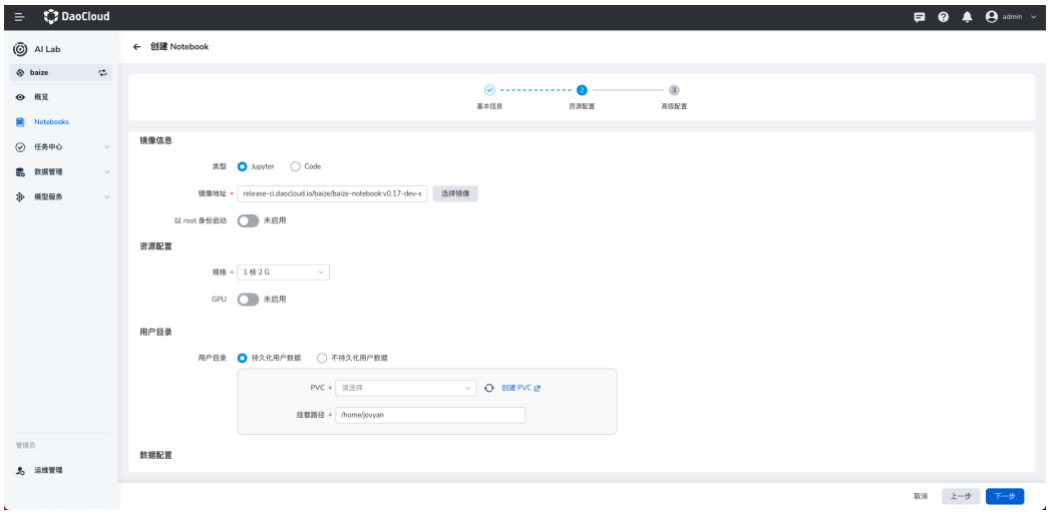

图 23　配置 Notebook 资源

4. 在 **高级配置** 中，可自定义任务参数、调度策略、任务分析、SSH 访问、自动保存等参数。调整完所有参数后点击 **确定**。

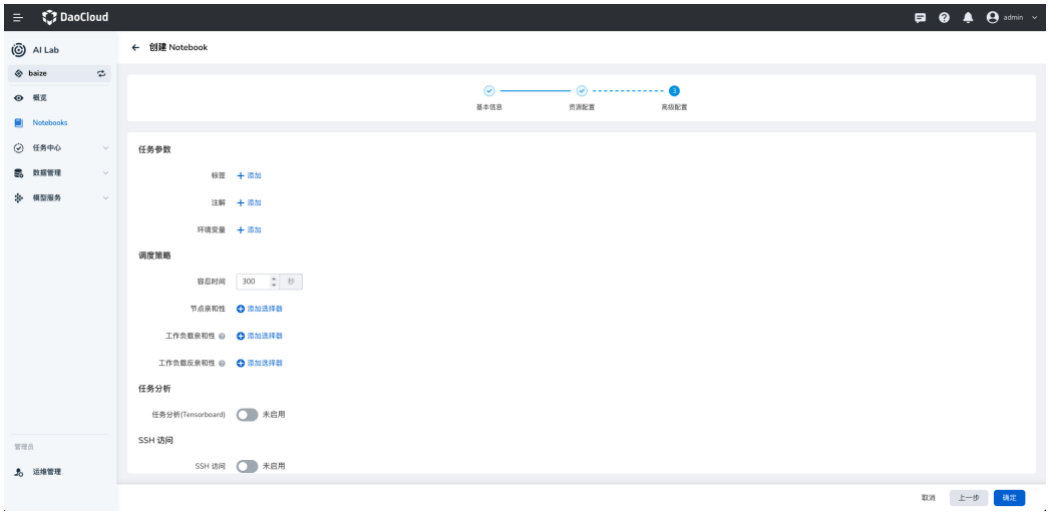

图 24　Notebook 高级配置

5. 刚创建的 Notebook 状态为 **等待中**，片刻后将变为 **运行中**，默认最新的位于列表顶部。

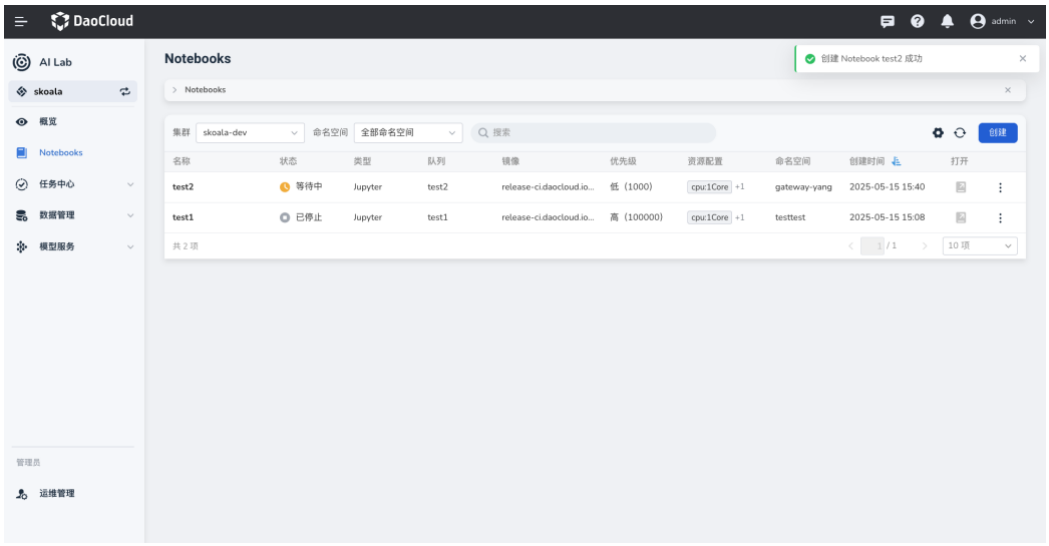

图 25　Notebook 创建成功

6. 点击右侧的 ⋮，可以执行更多操作：更新参数、启动/暂停、克隆 Notebook、查看工作负载详情和删除。

> 如果选择纯 CPU 资源后，发现挂载了节点上的所有 GPU 卡，可以尝试添加一条 container env 来解决此问题：

```
NVIDIA_VISIBLE_DEVICES=""
```

（二）启动和暂停 Notebook

Notebook 创建成功后，通常会有几个状态：

- 等待中
- 运行中

- 已停止

如果某个 Notebook 的状态为 **已停止**，在列表右侧点击 ⋮，在弹出菜单中选择 **启动**。

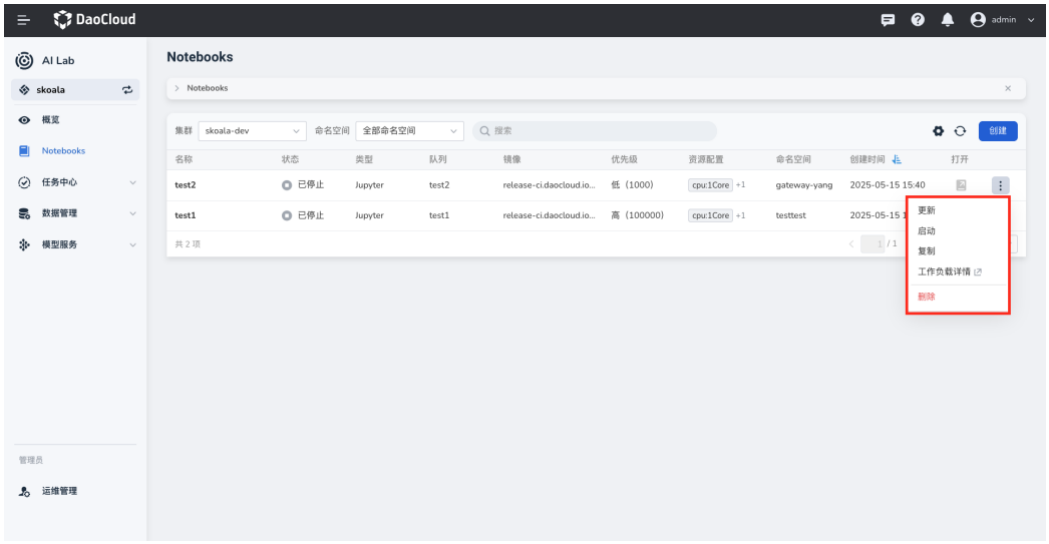

图 26　启动 Notebook

此 Notebook 将进入运行队列中，状态变为 **等待中**，如果一切正常，片刻后其状态将变为 **运行中**。

如果使用结束，可以从菜单中选择 **暂停**，将其状态变为 **已停止**。

（三）Notebook 工作负载

如果想要查看某个 Notebook 的工作负载，可以执行以下操作：

1. 在 Notebook 列表右侧点击 ⋮，在弹出菜单中选择 **工作负载详情**。

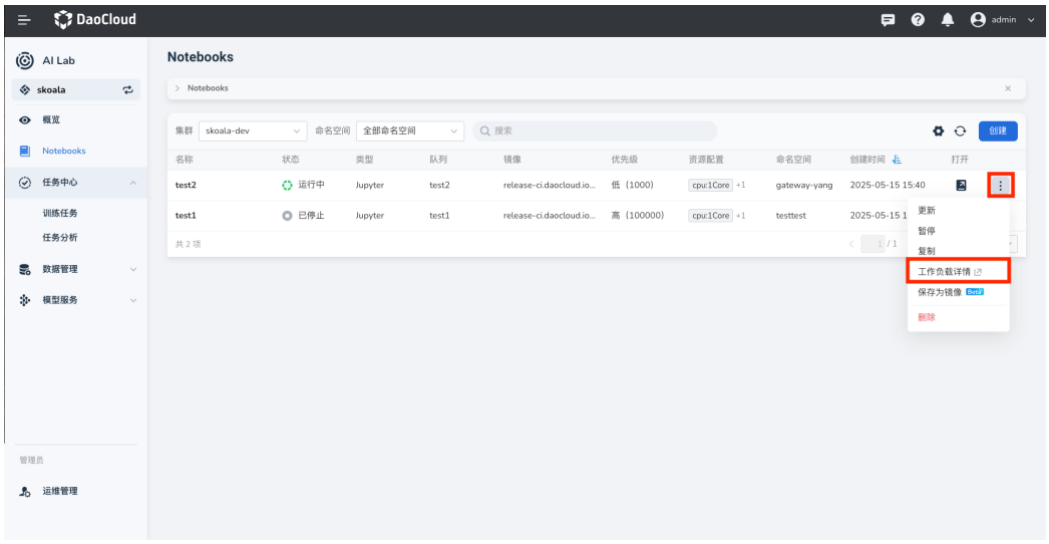

图 27　选择工作负载详情

2. 跳转到有状态工作负载（StatefulSet）列表，可以查看：

- 容器组 Pod 的运行状态、IP、资源请求和使用情况
- 容器配置信息
- 访问方式：ClusterIP、NodePort
- 调度策略：节点和工作负载的亲和性、反亲和性
- 标签与注解：工作负载、Pod 的标签与注解键值对
- 弹性伸缩：支持 HPA、CronHPA、VPA 等方式
- 事件列表：警告、通知等消息

图 28　查看工作负载详情

3. 在容器组列表，点击右侧的 ⋮，可以针对 Pod 执行更多操作。

图 29　针对 Pod 的更多操作

（四）删除 Notebook

如果发现某个 Notebook 冗余、过期或因其他缘故不再需要，可以从 Notebook 列表中删除。

1. 在 Notebook 列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

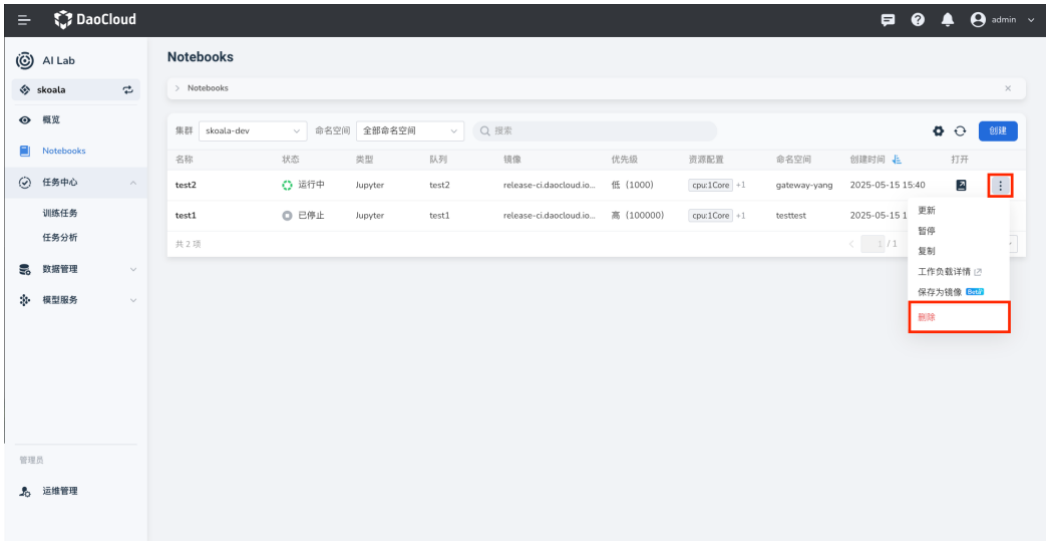

图 30　选择删除 Notebook

2. 在弹窗中输入 Notebook 名称，确认无误后点击 **删除**。

图 31　确认删除 Notebook

3. 屏幕提示删除成功，该 Notebook 从列表中消失。

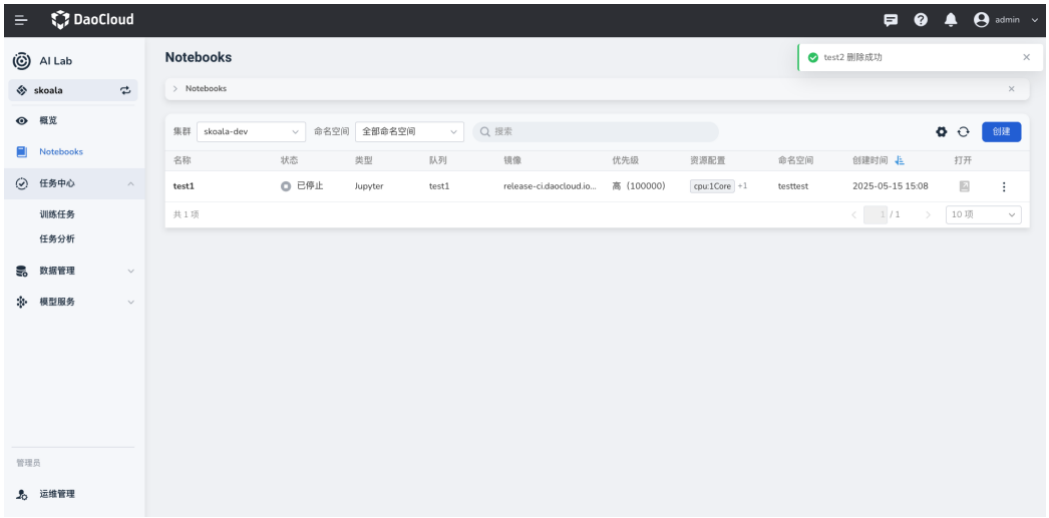

图 32　Notebook 已删除

> Notebook 一旦被删除将不可恢复，请谨慎操作。

（五）SSH 访问 Notebook

AI Lab 提供的 Notebook 支持在本地通过 SSH 的方式访问；通过简单的配置，即可使用 SSH 访问 Jupyter Notebook 的功能。无论您是使用 Windows、Mac 还是 Linux 操作系统，都可以按照以下步骤进行操作。

配置 SSH 访问凭证

1. 生成 SSH 密钥对

首先，您需要在您的计算机上生成 SSH 公钥和私钥对。这个密钥对将用于认证过程，确保安全访问。

Mac/Linux

1. 打开终端
2. 输入命令：

```
ssh-keygen -t rsa -b 4096
```

3. 当系统提示您「Enter a file in which to save the key」，您可以直接敲击 Enter 键使用默认路径，或者指定一个新的路径。
4. 接下来，系统会提示您输入密码（可选），这将增加一个额外的安全层。如果选择输入密码，请记住这个密码，因为每次使用密钥时都会需要它。

Windows

1. 安装 Git Bash（如果您尚未安装）
2. 打开 Git Bash
3. 输入命令：

```
ssh-keygen -t rsa -b 4096
```

4. 同 Mac/Linux 步骤

2. 添加 SSH 公钥到个人中心

> 具体操作可以参考 DCE「配置 SSH 公钥」文档。

1. 打开生成的公钥文件，通常位于 `~/.ssh/id_rsa.pub`（如果您没有更改默认路径）。
2. 复制公钥内容。
3. 登录到 DCE，从右上角进入 **个人中心**。
4. 在 SSH 公钥配置页，添加你本地生成的公钥文件，保存更改。

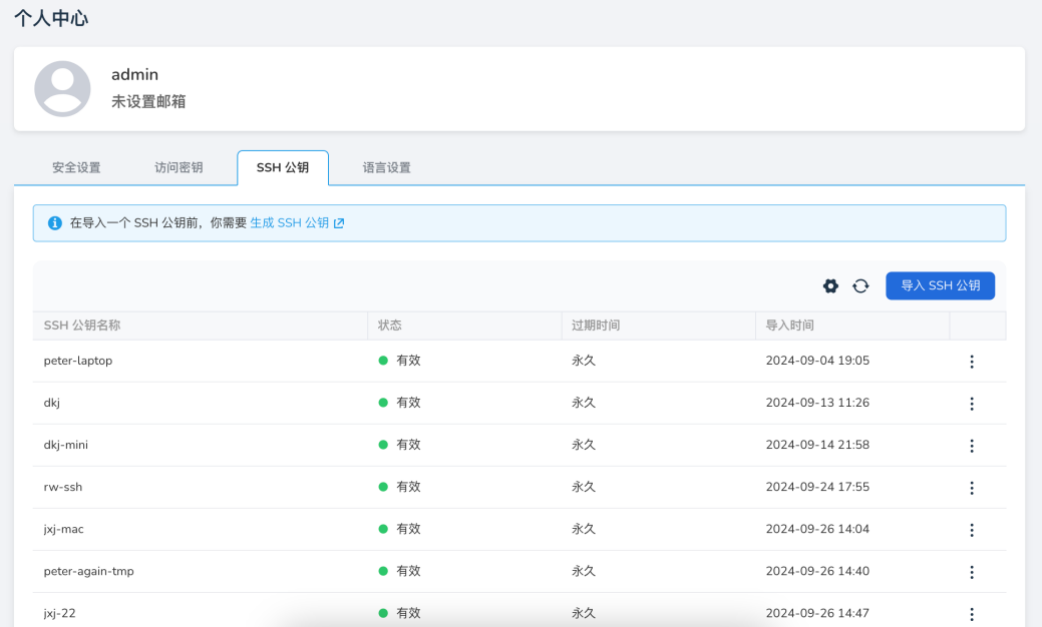

图 33　在个人中心添加 SSH 公钥

在 Notebook 中开启 SSH 访问

1. 登录到 Jupyter Notebook 的 Web 界面。

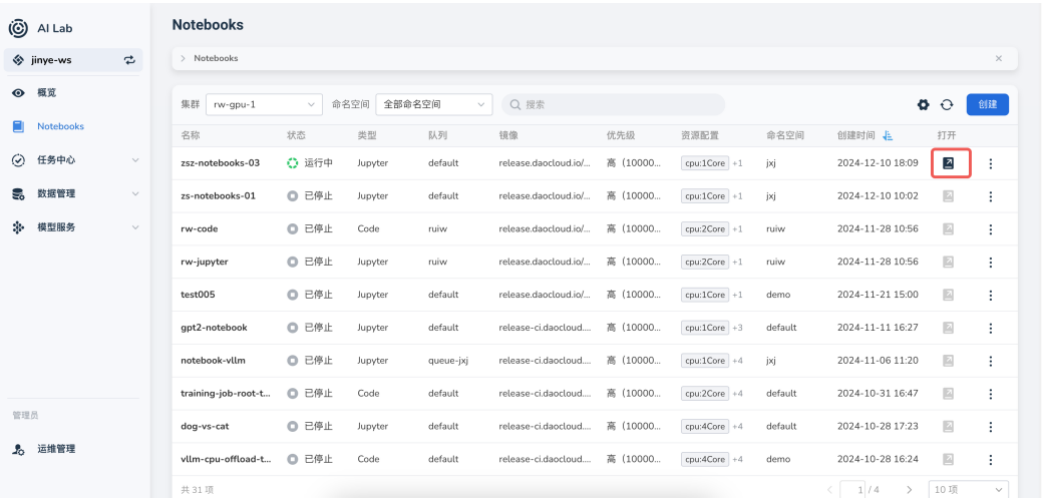

图 34　打开 Notebook

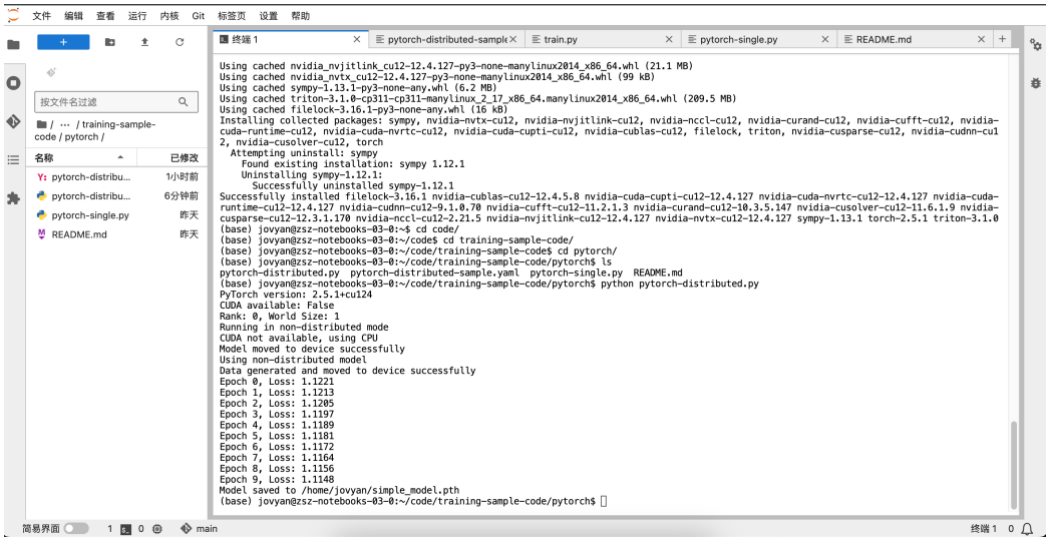

图 35 JupyterLab 界面

2. 寻找您想要启用 SSH 访问的 Notebook。

3. 在 Notebook 的设置或详情页面，找到 **开启 SSH 访问** 的选项并启用它。

图 36 开启 SSH 访问

4. 记录或复制显示的 SSH 访问命令。这个命令将用于后续步骤中的 SSH 连接。

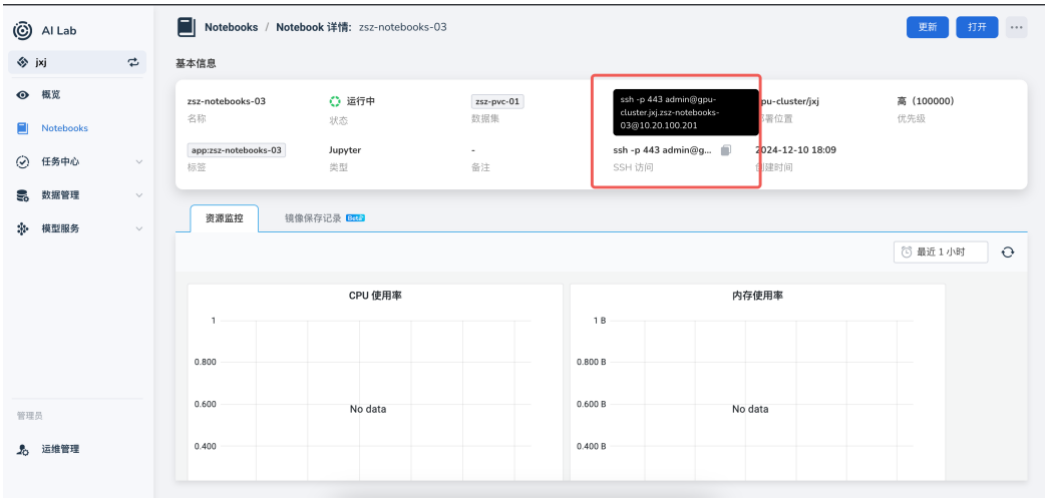

图 37　复制 SSH 访问命令

不同环境下的 SSH 访问方式

访问示例

假设您获得的 SSH 访问命令如下：

```
# ssh {DCE5_USERNAME}@{CLUSTER}.{NAMESPACE}.{NOTEBOOK_NAME}@{DCE5_UI_LOGIN_IP} -p
{DCE5_UI_LOGIN_IP}
ssh baizeuser01@gpu-cluster.demo.demo-notebook@10.20.100.201 -p 80 -i private_key
```

- `DCE5_USERNAME` 替换为您的用户名
- `DCE5_UI_LOGIN_IP` 替换为实际的主机名
- `DCE5_UI_LOGIN_IP` 替换为实际的端口号

Windows

推荐使用 PuTTY 或 Git Bash 进行 SSH 连接。

PuTTY

1. 打开 PuTTY
2. 在 **Host Name (or IP address)** 栏输入 `mockhost`（实际的主机名）
3. 输入端口号 `2222`（实际的端口号）
4. 点击 **Open** 开始连接
5. 第一次连接时，可能会提示验证服务器的身份，点击 **Yes**

Git Bash

1. 打开 Git Bash
2. 输入访问命令：

```
# ssh {DCE5_USERNAME}@{CLUSTER}.{NAMESPACE}.{NOTEBOOK_NAME}@{DCE5_UI_LOGIN_IP} -p
{DCE5_UI_LOGIN_IP}
```

```
ssh baizeuser01@gpu-cluster.demo.demo-notebook@10.20.100.201 -p 80 -i private_key
```

3. 按 Enter 键

Mac/Linux

1. 打开终端。
2. 输入访问命令：

```
# ssh {DCE5_USERNAME}@{CLUSTER}.{NAMESPACE}.{NOTEBOOK_NAME}@{DCE5_UI_LOGIN_IP} -p
{DCE5_UI_LOGIN_IP}
ssh baizeuser01@gpu-cluster.demo.demo-notebook@10.20.100.201 -p 80 -i private_key
```

3. 如果系统提示您接受主机的身份，请输入 `yes`。

配合 IDE 实现远程开发

除了使用命令行工具进行 SSH 连接，您还可以利用现代 IDE 如 Visual Studio Code (VSCode) 和 PyCharm 的 SSH 远程连接功能，直接在本地 IDE 中开发并利用远程服务器的资源。

在 VSCode 中使用 SSH 远程连接

VSCode 通过 **Remote - SSH** 扩展支持 SSH 远程连接，允许您直接在本地 VSCode 环境中编辑远程服务器上的文件，并运行命令。操作步骤为：

1. 确保您已安装 VSCode 和 **Remote - SSH** 扩展。
2. 打开 VSCode，点击左侧活动栏底部的远程资源管理器图标。
3. 选择 **Remote-SSH: Connect to Host...** 选项，然后点击 **+ Add New SSH Host...**。
4. 输入 SSH 连接命令，例如：

```
# ssh {DCE5_USERNAME}@{CLUSTER}.{NAMESPACE}.{NOTEBOOK_NAME}@{DCE5_UI_LOGIN_IP} -p
{DCE5_UI_LOGIN_IP}
ssh baizeuser01@gpu-cluster.demo.demo-notebook@10.20.100.201 -p 80 -i private_key
```

5. 敲击 Enter 键。请将 username、mockhost 和 2222 替换为实际的用户名、主机名和端口号。
6. 选择一个配置文件来保存此 SSH 主机，通常选择默认即可。

完成后，您的 SSH 主机将添加到 SSH 目标列表中。点击您的主机进行连接。如果是第一次连接，可能会提示您验证主机的指纹。接受后，您将被要求输入密码（如果 SSH 密钥设置了密码）。连接成功后，您可以像在本地开发一样在 VSCode 中编辑远程文件，并利用远程资源。

在 PyCharm 中使用 SSH 远程连接

PyCharm Professional 版支持通过 SSH 连接到远程服务器，并在本地 PyCharm 中直接开发。操作步骤为：

1. 打开 PyCharm，并打开或创建一个项目。
2. 选择 **File -> Settings**（在 Mac 上是 **PyCharm -> Preferences**）。
3. 在设置窗口中，导航到 **Project: YourProjectName -> Python Interpreter**。
4. 点击右上角的齿轮图标，选择 **Add...**：
    - 在弹出的窗口中，选择 **SSH Interpreter**；
    - 输入远程主机的信息：主机名（mockhost）、端口号（2222）、用户名（username）。请使用您的实际信息替换这些占位符；
    - 点击 **Next**，PyCharm 将尝试连接到远程服务器。如果连接成功，您将被要求输入密码或选择私钥文件。
1. 配置完成后，点击 **Finish**。现在，您的 PyCharm 将使用远程服务器上的 Python 解释器。

安全限制

在同一个 Workspace 内，任意用户都可以通过自己的 SSH 访问凭证来登录到启用了 SSH 的 Notebook。这意味着，只要用户配置了自己的 SSH 公钥到个人中心，并且 Notebook 启用了 SSH 访问，就可以使用 SSH 进行安全连接。

请注意，不同用户的访问权限可能会根据 Workspace 的配置而有所不同。确保您了解并遵守您所在组织的安全和访问策略。

（六）在 Notebook 中使用环境

环境管理是 AI Lab 的重要功能之一，通过在 **Notebook** 中关联对应的环境，可以快速切换不同的环境，方便用户进行开发和调试。

创建 Notebook 时选择环境

在创建 **Notebook** 时，可以选择一个或多个的环境 **Envs**。

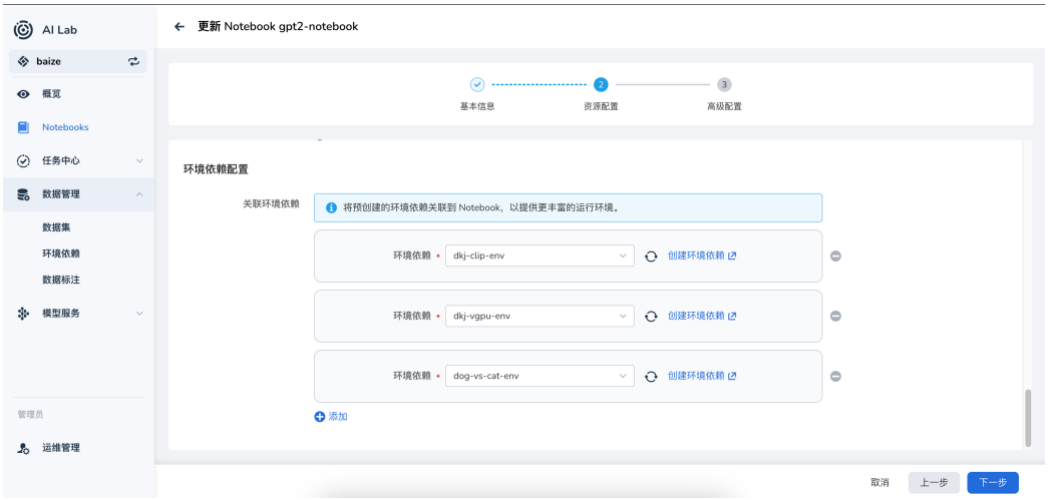

图 38　创建 Notebook 时关联环境

如果没有合适的环境，可以去创建一个新的环境。

在 Notebook 使用环境

> 在 Notebook 中，我们同时提供了 conda 和 mamba 两种，用户可以根据自己的需求选择合适的环境管理工具。

AI Lab 中，我们采用了 `conda` 环境管理工具，用户可以在 Notebook 中通过 `!conda env list` 命令查看当前环境列表。

```
(base) jovyan@chuanjia-jupyter-0:~/yolov8$ conda env list
# conda environments:
#
dkj-python312-pure       /opt/baize-runtime-env/dkj-python312-pure/conda/envs/dkj-
python312-pure
python-3.10              /opt/baize-runtime-env/python-3.10/conda/envs/python-3.10
torch-smaple             /opt/baize-runtime-env/torch-smaple/conda/envs/torch-smaple
base                  *  /opt/conda     # 当前激活的环境
baize-base               /opt/conda/envs/baize-base
```

这个命令会列出所有的 `conda` 环境，并在当前激活的环境前面加上一个星号（*）。

JupyterLab 的 Kernel 环境管理

在 JupyterLab 中，我们自动将 Notebook 关联的环境绑定到 Kernel 列表中，用户可以通过 Kernel 快速切换环境。

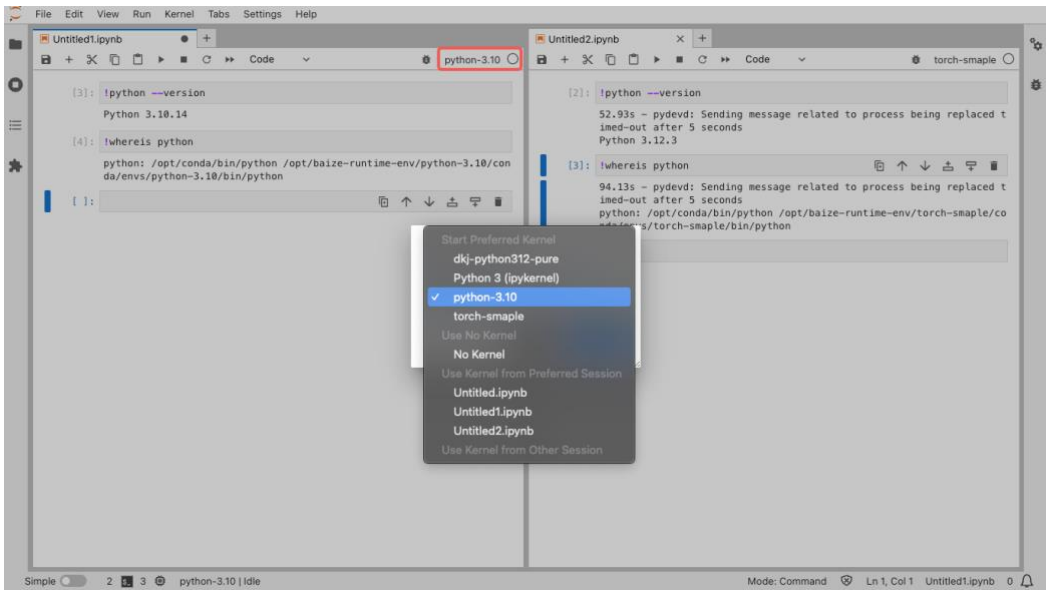

图 39　通过 Kernel 切换环境

通过以上办法，可以同时在一个 Notebook 中使用不同编写和调试算法。

Terminal 切换环境

AI Lab 的 Notebook 目前也已经支持了 VSCode。如果您更喜欢在 Terminal 中管理和切换环境，可以按照如下步骤操作。

在首次启动并使用 Notebook 时，需要先执行 `conda init`，然后再执行 `conda activate <env_name>` 切换到对应的环境。

```
(base) jovyan@chuanjia-jupyter-0:~/yolov8$ conda init bash# 初始化 bash 环境，仅首次使用需要执行
no change     /opt/conda/condabin/conda
 change     /opt/conda/bin/conda
 change     /opt/conda/bin/conda-env
 change     /opt/conda/bin/activate
 change     /opt/conda/bin/deactivate
 change     /opt/conda/etc/profile.d/conda.sh
 change     /opt/conda/etc/fish/conf.d/conda.fish
 change     /opt/conda/shell/condabin/Conda.psm1
 change     /opt/conda/shell/condabin/conda-hook.ps1
 change     /opt/conda/lib/python3.11/site-packages/xontrib/conda.xsh
 change     /opt/conda/etc/profile.d/conda.csh
 change     /home/jovyan/.bashrc
 action taken.
Added mamba to /home/jovyan/.bashrc

==> For changes to take effect, close and re-open your current shell. <==

(base) jovyan@chuanjia-jupyter-0:~/yolov8$ source ~/.bashrc  # 重新加载 bash 环境
(base) jovyan@chuanjia-jupyter-0:~/yolov8$ conda activate python-3.10   # 切换到 python-3.10 环境
(python-3.10) jovyan@chuanjia-jupyter-0:~/yolov8$ conda env list

              mamba version : 1.5.1
# conda environments:
#
dkj-python312-pure       /opt/baize-runtime-env/dkj-python312-pure/conda/envs/dkj-python312-pure
python-3.10           *  /opt/baize-runtime-env/python-3.10/conda/envs/python-3.10    # 当前激活的环境
torch-smaple             /opt/baize-runtime-env/torch-smaple/conda/envs/torch-smaple
base                     /opt/conda
baize-base               /opt/conda/envs/baize-base
```

如果您更喜欢使用 `mamba`，这里需要使用 `mamba init` 和 `mamba activate <env_name>`。

查看环境中的包

通过不同环境管理的一个很重要的功能是，可以在一个 Notebook 中通过快速切换环境，使用不同的包。我们可以通过下方的命令来使用 `conda` 查看当前环境中的所有包。

```
(python-3.10) jovyan@chuanjia-jupyter-0:~/yolov8$ conda list
# packages in environment at /opt/baize-runtime-env/python-3.10/conda/envs/python-3.10:
#
# Name                    Version                   Build  Channel
_libgcc_mutex             0.1                        main    defaults
_openmp_mutex             5.1                       1_gnu    defaults
... # 省略部分输出
idna                      3.7             py310h06a4308_0    defaults
ipykernel                 6.28.0          py310h06a4308_0    defaults
ipython                   8.20.0          py310h06a4308_0    defaults
```

```
jinja2                    3.1.4           py310h06a4308_0    defaults
jsonschema                4.19.2          py310h06a4308_0    defaults
jupyter_core              5.5.0           py310h06a4308_0    defaults
xz                        5.4.6                h5eee18b_1    defaults
yaml                      0.2.5                h7b6447c_0    defaults
zeromq                    4.3.5                h6a678d5_0    defaults
zlib                      1.2.13               h5eee18b_0    defaults
```

更新环境的包

目前，可以通过在 AI Lab 的界面中 **环境依赖** 来更新环境中的包。

（七）定制开发环境镜像

Notebook 提供了一套功能强大的在线开发环境。然而，在面对特定算法模型时，您可能需要不同版本的依赖库（例如特定版本的 CUDA、PyTorch 或 TensorFlow）。此时，平台内置镜像无法完全覆盖需求。

本节将指导您如何基于 AI Lab 提供的基础镜像，构建并注册包含自定义依赖的 Notebook 镜像，使平台用户在创建 Notebook 实例时能够直接选择该自定义镜像。

前提条件

权限要求

- 拥有目标 Kubernetes 集群的 kubectl 管理权限，或具备通过平台界面编辑 **baize-system** 命名空间下 ConfigMap 与 Deployment 资源的权限；
- 拥有一个可用的容器镜像仓库，并具备向该仓库推送镜像的权限。

工具要求

- 本地或服务器上已安装容器构建工具；
- 本地已配置好 kubectl 命令行工具并连接到目标集群。

操作步骤

整个过程分为四个核心步骤：

1. 获取基础 Notebook 镜像：获取当前 AI Lab 版本对应的 Notebook 基础镜像地址。
2. 定制并构建镜像：编写 Dockerfile 添加自定义依赖，并构建、推送镜像。
3. 注册新镜像：将新镜像的地址注册到 AI Lab 的配置中，并重启服务使其生效。
4. 使用新环境：在创建 Notebook 实例时，从下拉列表中选择您新添加的镜像。

1. 获取基础 Notebook 镜像

首先，我们需要找到当前 AI Lab 版本所依赖的基础 Notebook 镜像地址，该地址将作为 Dockerfile 的 `FROM` 源。

命令行操作

（1）通过 SSH 登录到 Kubernetes 集群的管理节点。

（2）执行以下命令，查看已安装的 AI Lab 版本。这将帮助您定位对应版本的镜像。

```
helm list -n baize-system | grep baize
```

您将看到类似如下的输出，请记下 `APP VERSION` 列的版本号（例如 `v0.19.1`）。

```
NAME    NAMESPACE     REVISION    UPDATED                                 STATUS
CHART            APP VERSION
baize   baize-system  5           2025-08-18 11:39:12.328965189 +0800 CST  deployed
baize-v0.19.1    v0.19.1
```

（3）执行以下命令，在输出的 YAML 内容中，找到 `notebook_images` 或类似字段，定位到与版本号匹配的镜像地址。复制其中一个与版本匹配的镜像地址备用，例如 `192.168.157.30/release.daocloud.io/baize/baize-notebook:v0.19.1`。

```
kubectl get cm baize -n baize-system -o yaml
```

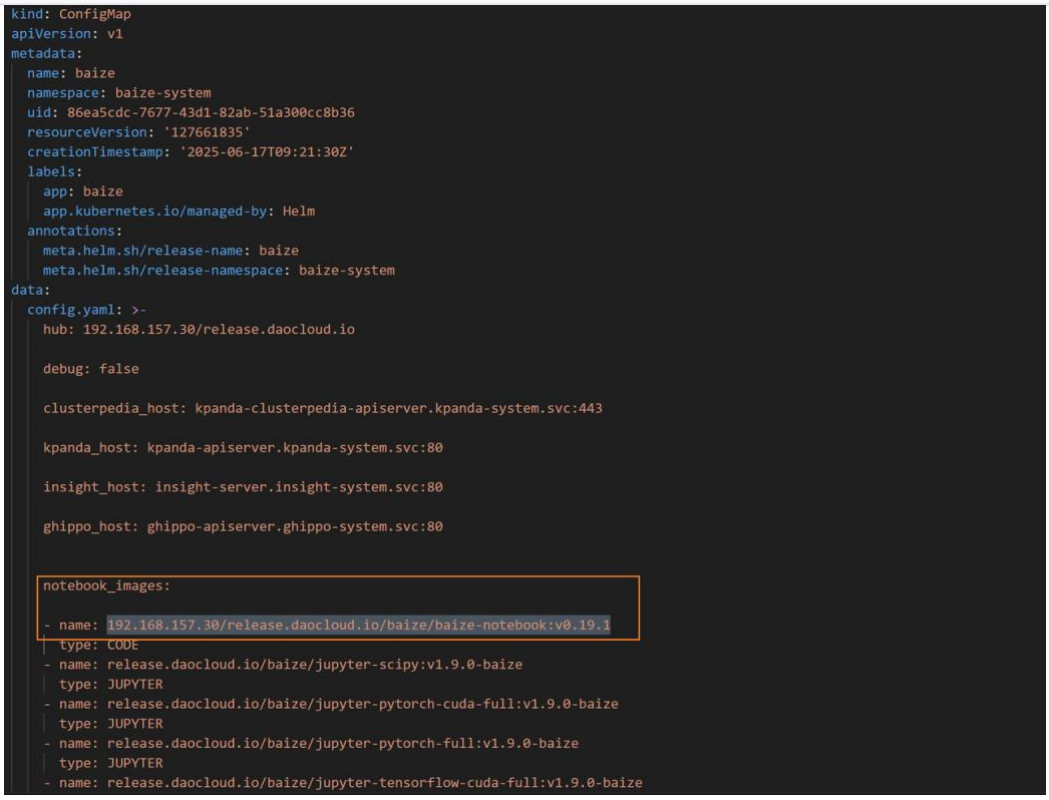

图 40　查看基础 Notebook 镜像

界面化操作

（1）登录 DCE，从左侧导航栏进入 **容器管理** -> **集群列表**，找到并点击 kpanda-global-cluster 全局服务集群。

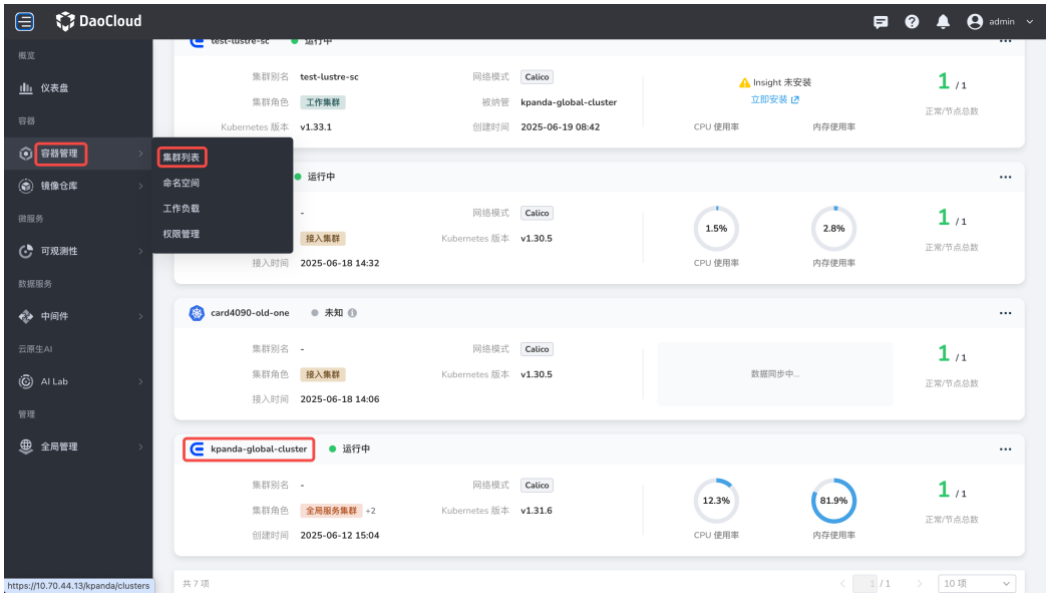

图 41　进入集群列表界面

（2）在 **kpanda-global-cluster** 集群的左侧导航栏，选择 **配置与密钥** -> **配置项**（ConfigMap），并在页面顶部的 **命名空间** 下拉框中，选择 **baize-system**。

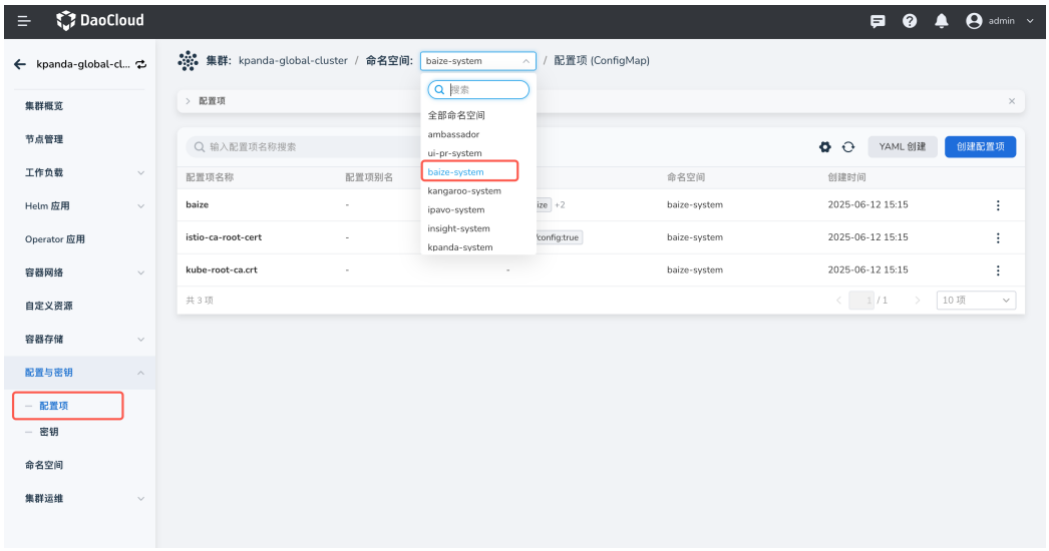

图 42　进入 baize-system 命名空间

（3）在列表中找到并点击进入名为 **baize** 的配置项。

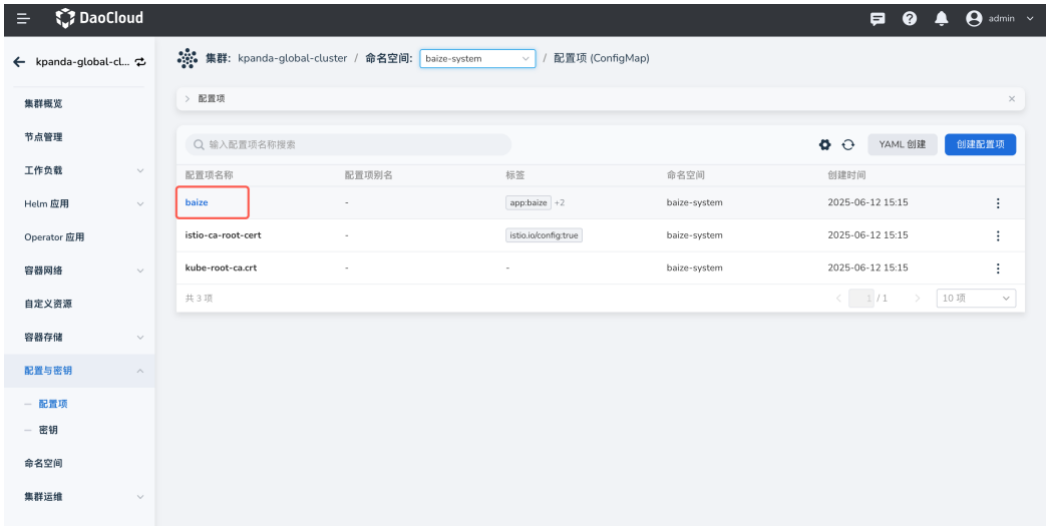

图 43　进入 baize 配置项

（4）点击页面右上角的 **编辑 YAML**。

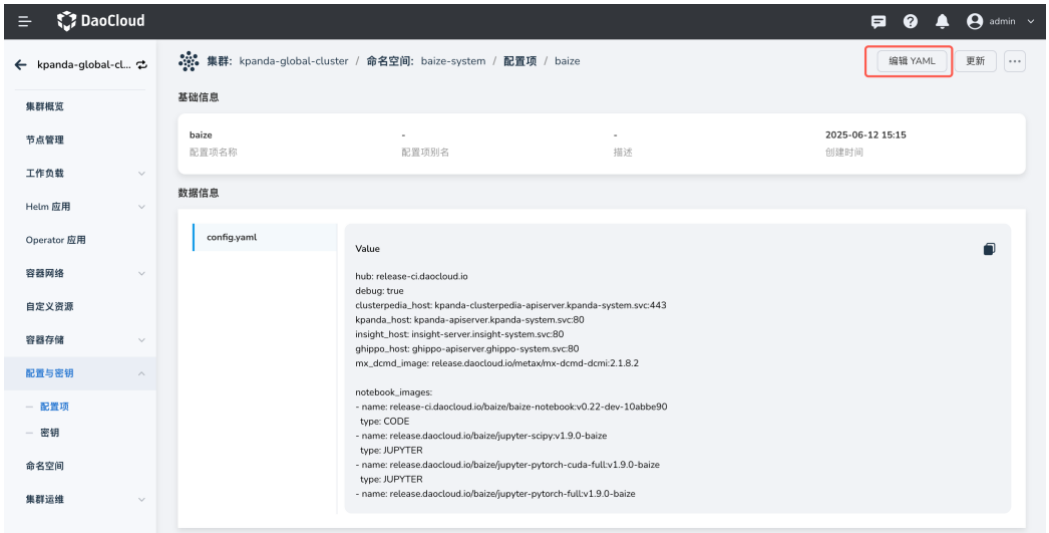

图 44　编辑 YAML

（5）在弹出的 YAML 编辑器中，找到 `data.config.yaml -> notebook_images:` 字段。列表中展示了当前所有内置的 Notebook 镜像。复制其中一个作为基础镜像地址，例如 `192.168.157.30/release.daocloud.io/baize/baize-notebook:v0.19.1`。

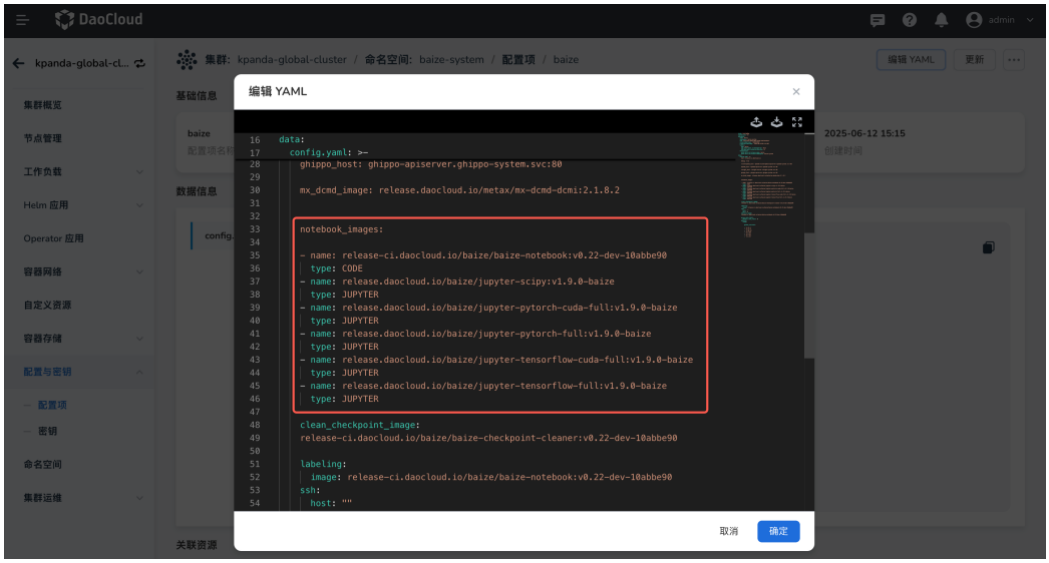

图 45　查看 notebook 基础镜像

2. 定制并构建镜像

本示例演示如何安装特定 CUDA 版本的 PyTorch。

（1） 在本地开发环境或构建服务器上创建 `Dockerfile`，以上一步获取的基础镜像为起点，安装所需依赖。

```
# 使用上一步获取的基础镜像地址
FROM 192.168.157.30/release.daocloud.io/baize/baize-notebook:v0.19.1

# 安装所需要的依赖，例如支持 CUDA 11.8 的 PyTorch
RUN pip install torch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2 --index-url
https://download.pytorch.org/whl/cu118
```

（2） 构建并推送定制镜像：

```
# 定义新镜像的完整名称和标签
export CUSTOM_IMAGE_NAME="<your-registry-address>/baize/baize-notebook:v0.19.1-cuda11.8-
torch2.0.1"

# 构建镜像
podman build -t ${CUSTOM_IMAGE_NAME} -f Dockerfile .

# 推送镜像到您的仓库
podman push ${CUSTOM_IMAGE_NAME}
```

> 请将 <your-registry-address>/... 替换为您自己的镜像仓库地址，并为其设定一个清晰可辨的标签（Tag），例如包含版本号、CUDA 版本和关键库。

3. 注册新镜像

镜像推送到仓库后，我们需要修改 AI Lab 的配置，让平台「知道」这个新环境的存在，并重启服务以应用配置。

命令行操作

（1）在管理节点上，执行以下命令进入 `baize` ConfigMap 的编辑模式。

```
kubectl edit cm baize -n baize-system
```

（2）在 YAML 编辑器中，找到 `notebook_images` 列表。在列表中添加一个新的条目，指向推送的定制镜像。最后保存并退出编辑器。

```
# ... (已有配置)
data:
  config.yaml: |-
    ...
    notebook_images:
      # - name: ... (已有的内置镜像)
      # - name: ... (已有的内置镜像)
      - name: <your-registry-address>/baize/baize-notebook:v0.19.1-cuda11.8-torch2.0.1
        type: JUPYTER
      ...
# ... (其他配置)
```

- `name`：这是上一节定制并构建出的镜像 `CUSTOM_IMAGE_NAME`。
- `type`：指定镜像的分类。如果主要用于 Jupyter，请设置 `JUPYTER`。

（3）在管理节点上，请依次执行以下命令，重启服务。

```
# 重启 baize-apiserver
kubectl rollout restart deployment baize-apiserver -n baize-system
```

界面化操作

（1）返回「获取基础 Notebook 镜像」中打开的 Baize ConfigMap **编辑 YAML** 界面。找到 `data.config.yaml -> notebook_images:` 列表。在列表末尾添加一个新条目，`name` 值为「定制并构建镜像」时推送的 `CUSTOM_IMAGE_NAME`。点击右下角的 **确定** 保存更改。

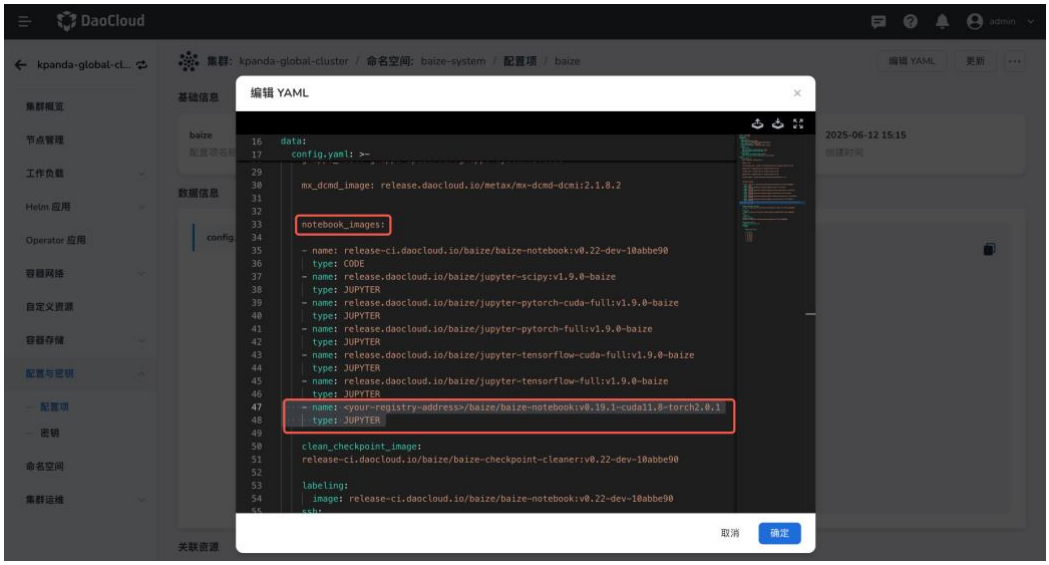

图 46　在 ConfigMap 中添加新镜像

（2）重启服务。仍在 kpanda-global-cluster 集群页面。从左侧导航栏进入 **工作负载** -> **无状态负载**，并在页面上方将 **命名空间** 选择为 baize-system。

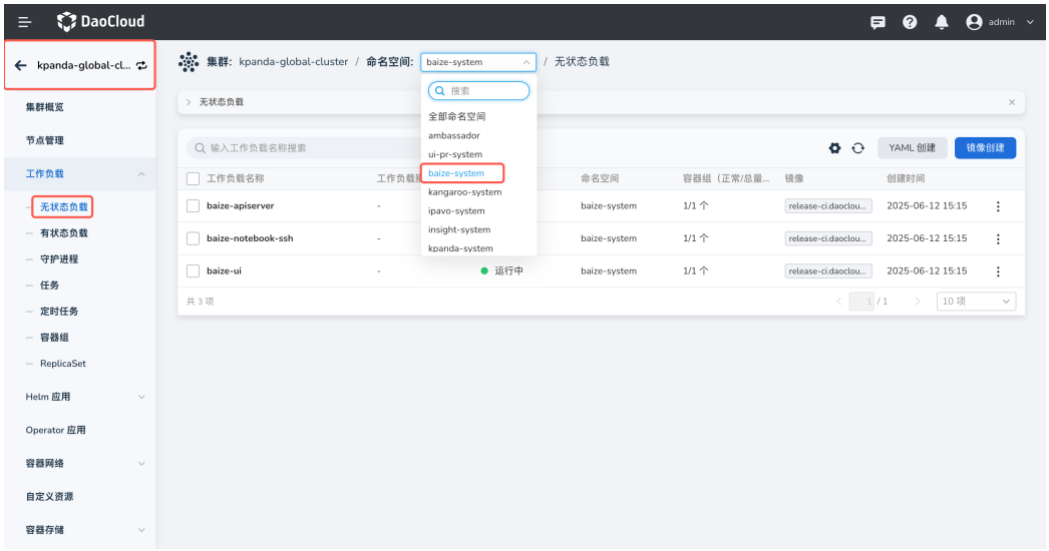

图 47　查看 baize-system 的工作负载

（3）在列表中找到 baize-apiserver，点击右侧的 ⋮ 操作按钮，选择 **修改状态** -> **重启**。

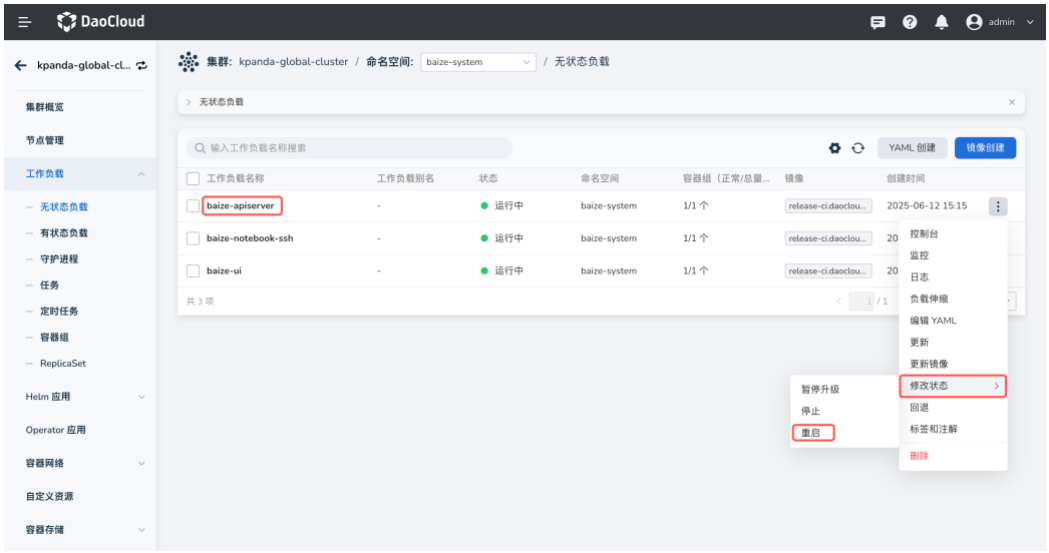

图 48　重启 baize-apiserver

4. 使用新环境

完成以上所有步骤后，您的新环境就已经准备就绪了。

（1） 导航至 AI Lab -> **开发控制台** -> Notebooks。点击右上角的 **创建**。

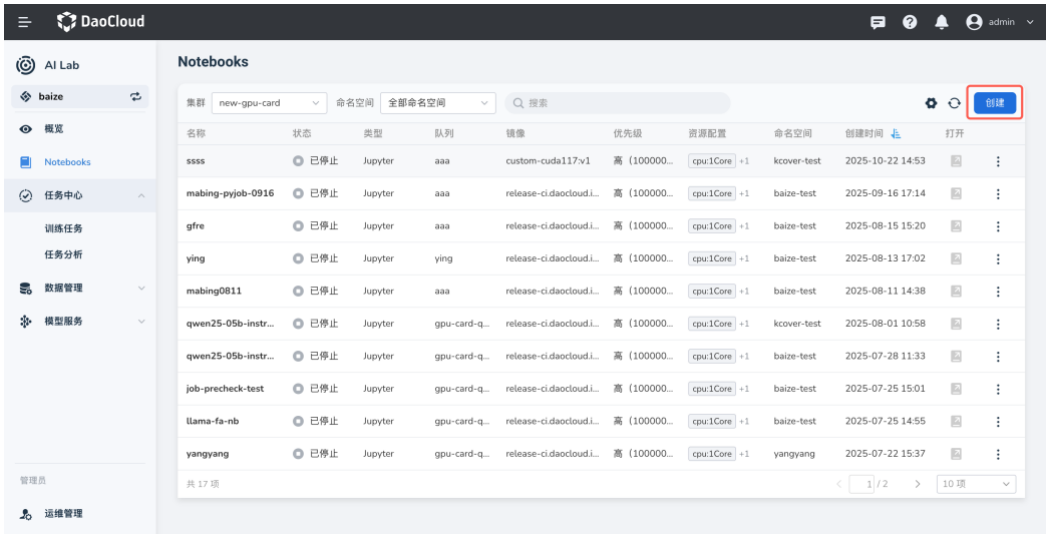

图 49　进入 Notebook 创建页面

（2） 在创建 Notebook 的 **资源配置** 环节中选择配置的 Notebook 类型，**镜像类型** 选择 **预制镜像**。在 **镜像地址** 的下拉菜单中，您现在应该能看到刚刚添加的自定义镜像选项（例如 `...:v0.19.1-cuda11.8-torch2.0`）。选择该镜像，配置其他资源后即可创建实例。

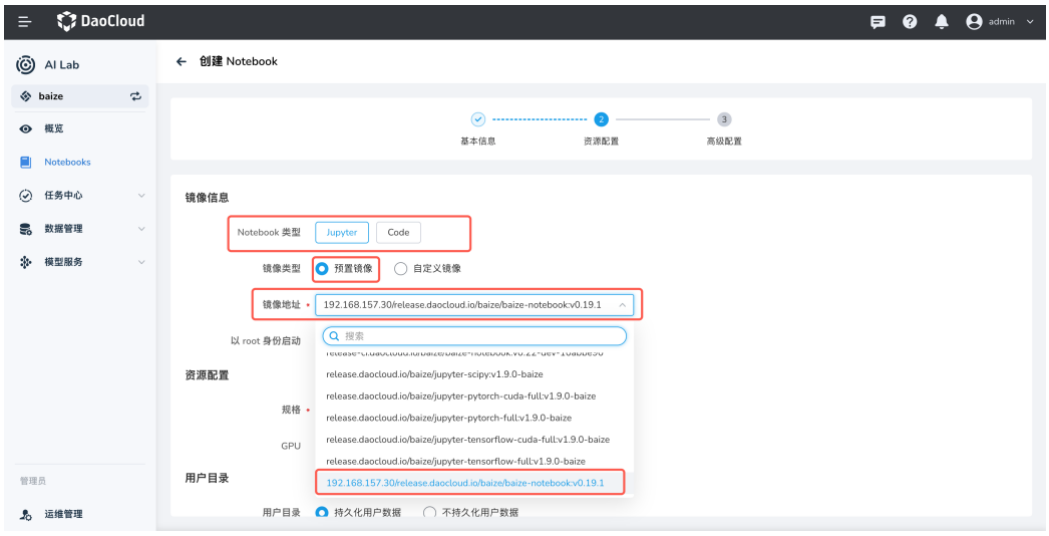

图 50　选择自定义镜像

（八）在 Notebook 中使用 Docker

在 Notebook 中使用 Docker 功能，可以将实验环境、依赖配置与计算过程进行统一封装，使 Notebook 不再依赖本地环境差异。

通过 Docker，Notebook 能够在不同节点、不同平台之间保持一致的运行行为，便于复现实验结果、共享分析过程，并与生产环境更好地对齐，尤其适用于数据分析、机器学习和模型验证等场景。

本节简要说明如何在 Notebook 实例中启用、验证和使用 Docker 功能，还列出了一些 Docker 高级功能和常见的故障排查案例。

启用 Docker

前提条件

参阅 DCE「管理 Helm 应用」文档安装所需插件。

启用 Docker 功能

1. 登录 **AI Lab** 平台，进入 Notebook 界面，点击 **创建** 按钮。

> **更新操作**：若要在已创建的实例中打开 Docker，则先点击对应实例右侧的 ⋮ 按钮，然后选择 **更新**。

2. 在创建 Notebook 界面中填写基本信息后，点击 **下一步**。
3. 完成资源配置后，点击 **下一步**。
4. 在高级配置中，勾选 **启用 Docker** 选项，点击 **确定**。
5. 创建完成后，返回 Notebook 列表页面。

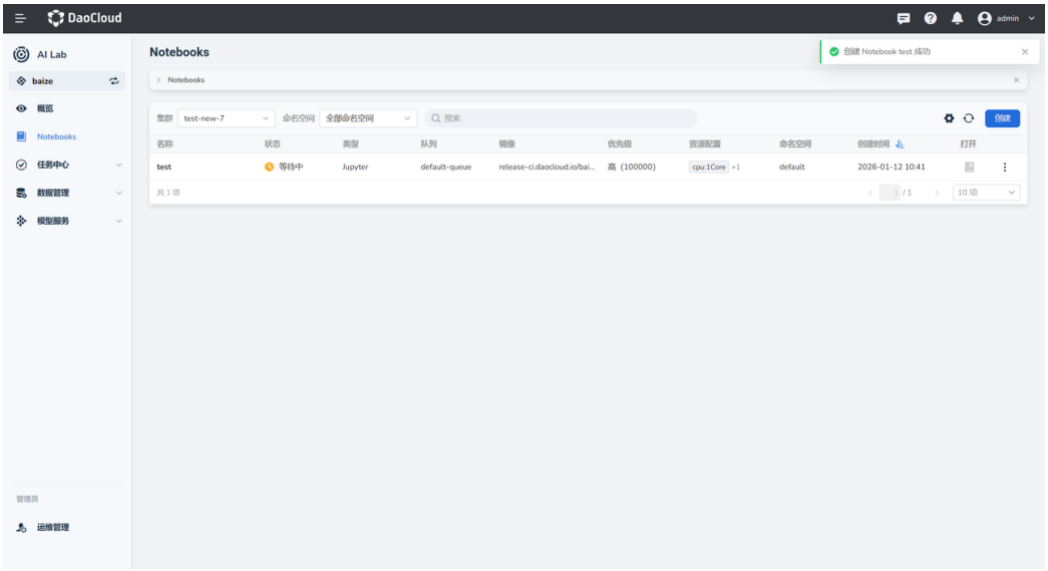

图 51　在高级配置中启用 Docker

6. 当 Notebook 实例状态由 **等待中** 变为 **运行中** 时，表示实例已成功启动。如一直在 **等待中**，请刷新页面。
7. 此时，实例右侧的 **打开** 列的图标处于黑色可点击状态。点击该图标，进入对应的 Notebook 实例。

验证 Docker 功能是否可用

在 Notebook 的终端环境中，可通过执行以下命令验证 Docker 环境是否已成功启用。

```
# 查看正在运行的容器
docker ps

# 查看所有容器（包括已停止的）
docker ps -a

# 查看指定容器的详细信息（将 <container_name> 替换为实际容器名或容器 ID）
docker inspect <container_name>
```

若命令可正常执行，并返回 Docker 容器列表信息（即使列表为空），则表示 Docker 服务已正常启动。

若提示 Docker 命令不存在或无法连接 Docker daemon，请确认 Docker 已被正确启用。

基本容器管理

Docker 功能提供完整的容器生命周期管理能力，支持容器的创建、启动、停止、删除等基本操作。

创建和运行容器

使用 docker run 命令创建并启动容器：

```
# 基本语法
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

# 运行一个简单的 Ubuntu 容器
docker run -it ubuntu:20.04 /bin/bash

# 后台运行容器
docker run -d --name my-app nginx:latest

# 指定端口映射
docker run -d -p 8080:80 --name web-server nginx:latest
```

常用参数说明如下：

参数	说明
-d	后台运行容器
-it	交互式运行，分配伪终端
--name	指定容器名称
-p	端口映射，格式为 宿主机端口:容器端口
-v	挂载数据卷

管理容器

查看容器状态：

```
# 查看正在运行的容器
docker ps

# 查看所有容器（包括已停止的）
docker ps -a

# 查看容器详细信息
docker inspect <container_name>
```

启动和停止容器：

```
# 启动已停止的容器
docker start <container_name>

# 停止运行中的容器
docker stop <container_name>

# 重启容器
docker restart <container_name>
```

进入容器

进入正在运行的容器：

```
# 进入容器的交互式终端
docker exec -it <container_name> /bin/bash

# 查看某个容器内的所有文件
docker exec <container_name> ls -la /app
```

> 建议使用 docker exec 而不是 docker attach 来进入容器，因为 exec 会创建新的进程，退出时不会影响容器的运行状态。

挂载存储

Notebook 可以挂载 PVC 或通过数据空间实现数据的持久化存储，其挂载路径可以关联到 Docker，从而实现容器间的数据共享和数据的持久化存储。

```
# 挂载到指定目录
docker run -d -v /root/data:/workspace/data --name dev-env python:3.9
```

镜像制作

Docker 功能支持多种方式制作和保存自定义镜像，满足不同场景的需求。

docker build

使用 Dockerfile 构建镜像是最常用的方式：

```
# 基本构建命令
docker build -t my-app:latest .

# 指定 Dockerfile 路径
docker build -f /path/to/Dockerfile -t my-app:v1.0 .
```

docker save

将镜像导出为 tar 文件：

```
# 导出单个镜像
docker save -o my-image.tar my-app:latest
```

导出的镜像可以通过 `docker load` 命令导入：

```
# 导入镜像
docker load -i my-image.tar
```

使用 GPU

在 Notebook 中启用 Docker 功能后，可以将 GPU 资源挂载到容器中，为 AI 训练和推理提供硬件加速能力。

先挂载 GPU

使用 `--gpus` 参数将 GPU 挂载到容器：

```
# 挂载所有 GPU
docker run --gpus all -it pytorch/pytorch:latest python

# 挂载指定数量的 GPU
docker run --gpus 2 -it tensorflow/tensorflow:latest-gpu python

# 挂载指定的 GPU
docker run --gpus device=0 -it nvidia/cuda:11.8-devel-ubuntu20.04
```

支持哪些 GPU

Notebook 支持多种 GPU，详情见 DCE「GPU 支持矩阵」文档。

网络配置

Docker 容器可以通过多种网络模式与宿主机和外部网络进行通信。

端口映射

将容器端口映射到宿主机端口，实现外部访问：

```
# 映射单个端口
docker run -d -p 8080:80 --name web-app nginx:latest

# 映射多个端口
docker run -d \  -p 8080:80 \  -p 8443:443 \  --name web-server nginx:latest

# 映射到指定 IP
docker run -d -p 127.0.0.1:8080:80 --name local-app nginx:latest

# 映射随机端口
docker run -d -P --name random-port nginx:latest
```

高级功能

Notebook 的 Docker 功能支持 buildx 和 Compose 等高级工具，满足复杂场景下的容器化开发需求。

Docker buildx

Docker buildx 是 Docker 的扩展构建功能，支持多平台构建和高级构建特性。

基本用法：

```
# 查看 buildx 版本
docker buildx version

# 查看可用的构建器
docker buildx ls

# 创建新的构建器
docker buildx create --name mybuilder --use

# 启动构建器
docker buildx inspect --bootstrap
```

多平台构建：

```
# 构建多平台镜像
docker buildx build --platform linux/amd64,linux/arm64 -t my-app:latest .

# 构建并推送到仓库
docker buildx build --platform linux/amd64,linux/arm64 -t my-app:latest --push .

# 构建特定平台
docker buildx build --platform linux/amd64 -t my-app:amd64 .
```

Docker Compose

Docker Compose 用于定义和运行多容器应用程序。

安装和基本用法：

```
# 检查 Compose 版本
docker compose version

# 启动服务
docker compose up -d

# 查看服务状态
docker compose ps

# 停止服务
docker compose down

# 查看日志
docker compose logs
```

访问镜像仓库

Notebook 的 Docker 功能支持访问镜像仓库以及其他公有和私有镜像仓库。

镜像仓库

本平台提供内置的镜像仓库服务，用户可以存储和管理自定义镜像。

访问仓库：

```
# 查看仓库地址（示例）
# 实际地址请参考平台提供的「我的镜像」中仓库信息
REGISTRY_URL="harbor.io"

# 拉取镜像
docker pull ${REGISTRY_URL}/my-namespace/my-app:latest

# 推送镜像
docker push ${REGISTRY_URL}/my-namespace/my-app:latest
```

认证配置：

> 当前版本暂不支持自动注入用户名和密码，需要手动配置认证信息。

手动配置认证：

```
# 登录到 AI Lab 镜像仓库
docker login registry.io -u <your-username>

# 输入密码
Password: <your-password>

# 验证登录状态
docker info | grep -A 5 "Registry Mirrors"
```

故障排查

常见问题及解决方法：

- **容器启动失败**

```
# 查看容器日志
docker logs <container_name>

# 查看容器详细信息
docker inspect <container_name>
```

- **端口访问问题**

```
# 检查端口映射
docker port <container_name>

# 检查防火墙设置
netstat -tlnp | grep :8080
```

- GPU 不可用

```
# 检查 GPU 状态
nvidia-smi

# 验证容器内 GPU 访问
```

```
docker exec <container_name> nvidia-smi
```

重要提醒

- Notebook 关机时，运行中的 Docker 容器会被停止。
- 重启 Notebook 后，需要手动重启 Docker 容器。
- 删除 Notebook 会同时删除所有 Docker 容器和未持久化的数据。

最佳实践

- 使用 Docker Compose 管理复杂的多容器应用。
- 定期清理未使用的镜像和容器以节省存储空间。
- 为生产环境的容器配置健康检查和重启策略。
- 使用标准化的镜像命名和版本管理规范。

通过合理使用 Notebook 的 Docker 功能，开发者可以构建灵活、高效的容器化开发和部署环境，充分利用 AI Lab 平台的计算资源和存储能力。

（九）Notebook 闲置超时自动关机

在默认情况下，为优化资源利用率，AI Lab 启用了 Notebook 闲置超时自动关机功能；当 Notebook 长时间无操作时，系统会自动关机 Notebook，释放资源。

- 优点：通过这个方式，可以极大减少因为长时间无操作导致的资源浪费，提高资源利用效率。
- 缺点：如果 Notebook 未配置相关备份策略，可能导致数据丢失。

当前，此功能为集群级别配置，默认开启，默认超时时长为 30 分钟。

配置变更

目前配置修改方式为手动修改，后续会提供更加便捷的配置方式。修改工作集群中 `baize-agent` 的部署参数，正确的修改方式为更新 Helm 应用。

界面化修改

1. 在集群管理界面找到对应的工作集群，进入集群详情，选择 **Helm 应用**，在 `baize-system` 命名空间下找到 `baize-agent`，在右上角点击 **更新** 按钮。

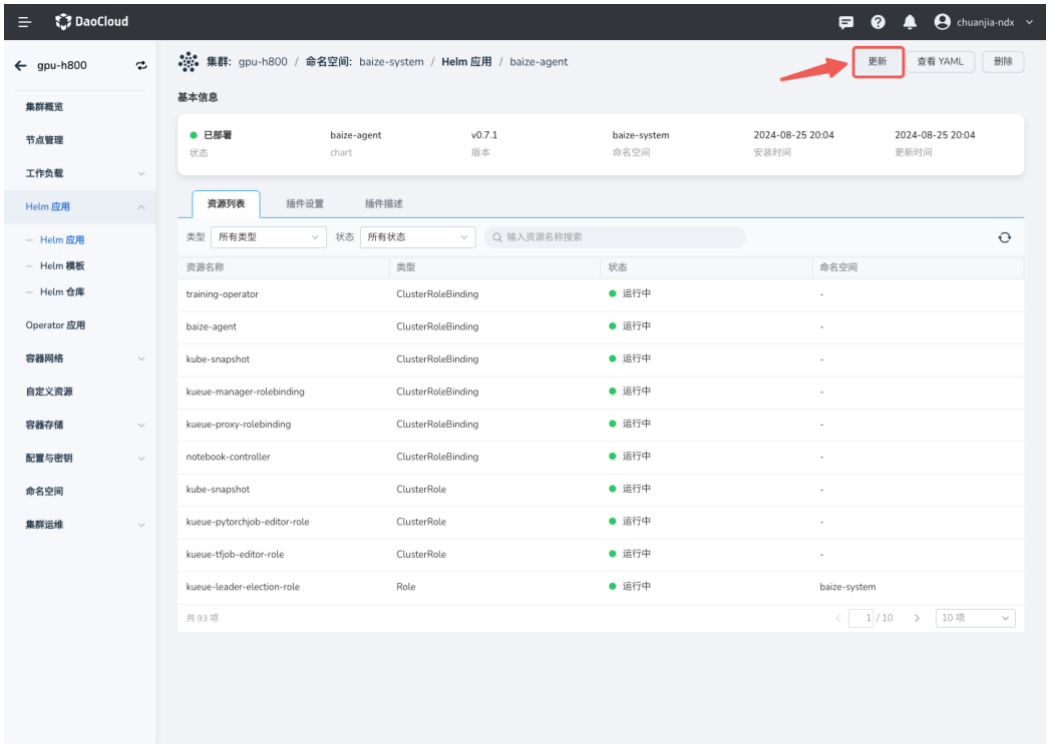

图 52　进入 baize-agent Helm 应用

2. 如图修改 YAML 代码：

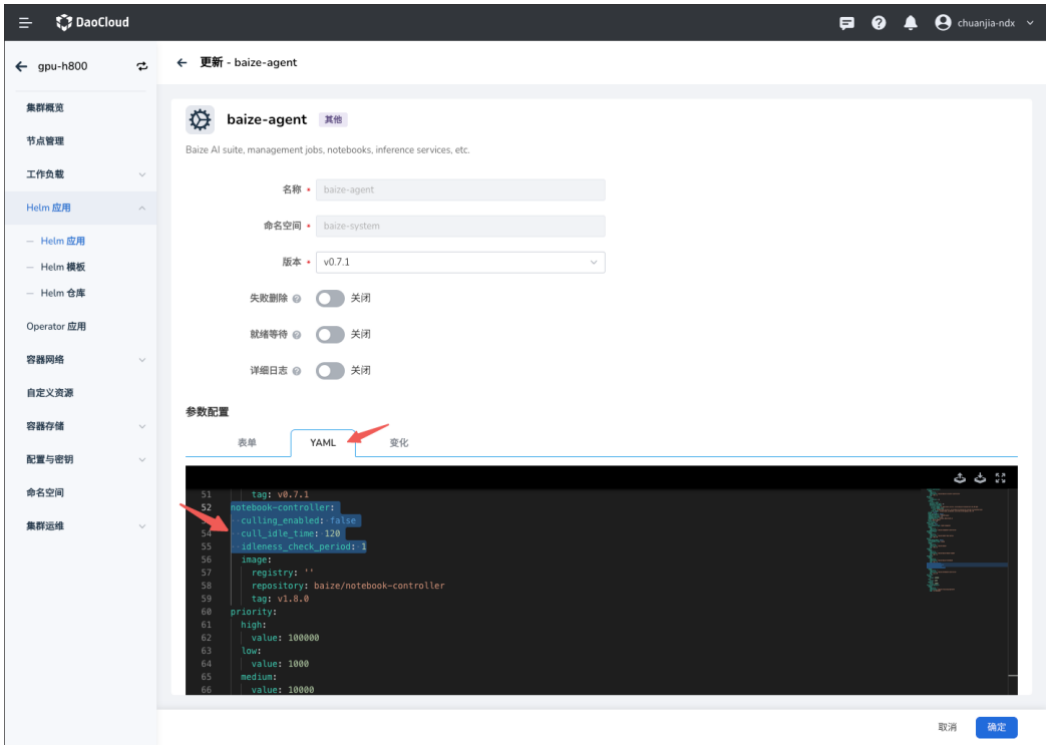

图 53　修改 notebook-controller 参数

```
...
notebook-controller:
  culling_enabled: false
```

```
  cull_idle_time: 120
  idleness_check_period: 1
...
```

3. 确认参数修改成功后，点击 **下一步** 和 **确定**。

命令行修改

进入控制台以后，使用 `helm upgrade` 命令更改配置：

```
# 设定版本号
export VERSION=0.8.0

# 更新 Helm Chart
helm upgrade --install baize-agent baize/baize-agent \
    --namespace baize-system \
    --create-namespace \
    --set global.imageRegistry=release.daocloud.io \
    --set notebook-controller.culling_enabled=true \    # 开启自动关机，默认为 true
    --set notebook-controller.cull_idle_time=120 \      # 设置闲置超时时间为 120 分钟，默认为 30 分钟
    --set notebook-controller.idleness_check_period=1 \ # 设置检查间隔为 1 分钟，默认为 1 分钟
    --version=$VERSION
```

> 为了避免自动关机后丢失数据，您可以将 AI Lab 升级到 v0.8.0 及更高版本，在 Notebook 配置中启用关机自动保存功能。

（十）baizectl 命令行工具

`baizectl` 是在 DCE AI Lab 模块中专门服务于模型开发者与数据科学家们使用的命令行工具。它提供了一系列命令来帮助用户管理分布式训练作业、查看任务状态、管理数据集等操作，同时支持连接 Kubernetes 工作集群和 DCE 工作空间，帮助用户更高效地使用和管理 Kubernetes 平台资源。

安装

目前，`baizectl` 已经集成在 DCE AI Lab 中。你在创建 Notebook 后，即可在 Notebook 中直接使用 `baizectl`。

快速上手

基本信息

`baizectl` 命令的基本格式如下：

```
jovyan@19d0197587cc:/$ baizectl
AI platform management tool

Usage:
  baizectl [command]

Available Commands:
  completion  Generate the autocompletion script for the specified shell
```

```
  data        Management datasets
  help        Help about any command
  job         Manage jobs
  login       Login to the platform
  version     Show cli version

Flags:
      --cluster string     Cluster name to operate
  -h, --help               help for baizectl
      --mode string        Connection mode: auto, api, notebook (default "auto")
  -n, --namespace string   Namespace to use for the operation. If not set, the default
Namespace will be used.
  -s, --server string      DCE5 access base url
      --skip-tls-verify    Skip TLS certificate verification
      --token string       DCE5 access token
  -w, --workspace int32    Workspace ID to use for the operation

Use "baizectl [command] --help" for more information about a command.
```

以上是 `baizectl` 的基本信息，用户可以通过 `baizectl --help` 查看帮助信息，或者通过 `baizectl [command] --help` 查看具体命令的帮助信息。

查看版本信息

`baizectl` 支持通过 `version` 命令查看版本信息。

```
(base) jovyan@den-0:~$ baizectl version
baizectl version: v0.5.0, commit sha: ac0837c4
```

命令格式

`baizectl` 命令的基本格式如下：

```
baizectl [command] [flags]
```

其中，`[command]` 是具体的操作命令，如 `data`、`job` 等，`[flags]` 是可选的参数，用于指定操作的详细信息。

常用选项

- `--cluster string`：指定要操作的集群名称。
- `-h, --help`：显示帮助信息。
- `--mode string`：连接模式，可选值为 `auto`、`api`、`notebook`（默认值为 `auto`）。
- `-n, --namespace string`：指定操作的命名空间。如果未设置，将使用默认命名空间。
- `-s, --server string`：DCE5 访问基础 URL。
- `--skip-tls-verify`：跳过 TLS 证书验证。
- `--token string`：DCE5 访问令牌。
- `-w, --workspace int32`：指定操作的工作区 ID。

功能介绍：任务管理

baizectl 提供了一系列命令来管理分布式训练任务，包含了查看任务列表、提交任务、查看日志、重启任务、删除任务等。

```
jovyan@19d0197587cc:/$ baizectl job
Manage jobs

Usage:
  baizectl job [command]

Available Commands:
  delete      Delete a job
  logs        Show logs of a job
  ls          List jobs
  restart     restart a job
  submit      Submit a job

Flags:
  -h, --help            help for job
  -o, --output string   Output format. One of: table, json, yaml (default "table")
      --page int        Page number (default 1)
      --page-size int   Page size (default -1)
      --search string   Search query
      --sort string     Sort order
      --truncate int    Truncate output to the given length, 0 means no truncation
(default 50)

Use "baizectl job [command] --help" for more information about a command.
```

提交训练任务

`baizectl` 支持使用 `submit` 命令提交一个任务，用户可以通过 `baizectl job submit --help` 查看详细信息。

```
(base) jovyan@den-0:~$ baizectl job submit --help
Submit a job

Usage:
  baizectl job submit [flags] -- command ...

Aliases:
  submit, create

Examples:
# Submit a job to run the command "torchrun python train.py"
baizectl job submit -- torchrun python train.py
# Submit a job with 2 workers(each pod use 4 gpus) to run the command "torchrun python
train.py" and use the image "pytorch/pytorch:1.8.1-cuda11.1-cudnn8-runtime"
baizectl job submit --image pytorch/pytorch:1.8.1-cuda11.1-cudnn8-runtime --workers 2 --
requests-resources nvidia.com/gpu=4 -- torchrun python train.py
# Submit a tensorflow job to run the command "python train.py"
baizectl job submit --tensorflow -- python train.py

Flags:
      --annotations stringArray                      The annotations of the job, the
format is key=value
      --auto-load-env                                It only takes effect when executed
in Notebook, the environment variables of the current environment will be automatically
read and set to the environment variables of the Job, the specific environment variables
to be read can be specified using the BAIZE_MAPPING_ENVS environment variable, the
default is PATH,CONDA_*,*PYTHON*,NCCL_*, if set to false, the environment variables of
the current environment will not be read. (default true)
      --commands stringArray                         The default command of the job
  -d, --datasets stringArray                         The dataset bind to the job, the
format is datasetName:mountPath, e.g. mnist:/data/mnist
  -e, --envs stringArray                             The environment variables of the
job, the format is key=value
  -x, --from-notebook string                         Define whether to read the
configuration of the current Notebook and directly create tasks, including images,
resources, Dataset, etc.
```

```
                                                    auto: Automatically determine the
mode according to the current environment. If the current environment is a Notebook, it
will be set to notebook mode.
                                                    false: Do not read the
configuration of the current Notebook.
                                                    true: Read the configuration of
the current Notebook. (default "auto")
  -h, --help                                        help for submit
      --image string                                The image of the job, it must be
specified if fromNotebook is false.
  -t, --job-type string                             Job type: PYTORCH, TENSORFLOW,
PADDLE (default "PYTORCH")
      --labels stringArray                          The labels of the job, the format
is key=value
      --max-retries int32                           number of retries before marking
this job failed
      --max-run-duration int                        Specifies the duration in seconds
relative to the startTime that the job may be active before the system tries to
terminate it
      --name string                                 The name of the job, if empty, the
name will be generated automatically.
      --paddle                                      PaddlePaddle Job, has higher
priority than --job-type
      --priority string                             The priority of the job, current
support baize-medium-priority, baize-low-priority, baize-high-priority
      --pvcs stringArray                            The pvcs bind to the job, the
format is pvcName:mountPath, e.g. mnist:/data/mnist
      --pytorch                                     Pytorch Job, has higher priority
than --job-type
      --queue string                                The queue to used
      --requests-resources stringArray              Similar to resources, but sets the
resources of requests
      --resources stringArray                       The resources of the job, it is a
string in the format of cpu=1,memory=1Gi,nvidia.com/gpu=1, it will be set to the limits
and requests of the container.
      --restart-policy string                       The job restart policy (default
"on-failure")
      --runtime-envs                                The runtime environment to use for
the job, you can use baizectl data ls --runtime-env to get the runtime environment
      --shm-size int32                              The shared memory size of the job,
default is 0, which means no shared memory, if set to more than 0, the job will use the
shared memory, the unit is MiB
      --tensorboard-log-dir string                  The tensorboard log directory, if
set, the job will automatically start tensorboard, else not. The format is /path/to/log,
you can use relative path in notebook.
      --tensorflow                                  Tensorflow Job, has higher
priority than --job-type
      --workers int                                 The workers of the job, default is
1, which means single worker, if set to more than 1, the job will be distributed.
(default 1)
      --working-dir string                          The working directory of job
container, if in notebook mode, the default is the directory of the current file
```

提交任务的命令参数说明：

- --name：任务名称，如果为空，则会自动生成。
- --image：镜像名称，必须指定。
- --priority：任务优先级，支持 高=baize-high-priority、中=baize-medium-priority、低=baize-low-priority。
- --requests-resources：requests 的任务资源，格式为 cpu=1 memory=1Gi,nvidia.com/gpu=1。
- --resources：limits 的任务资源，格式为 cpu=1 memory=1Gi,nvidia.com/gpu=1。
- --workers：任务工作节点数，默认为 1，当设置大于 1 时，任务将会分布式运行。
- --queue：任务队列，需要提前创建队列资源。
- --working-dir：工作目录，如果在 Notebook 模式下，会默认使用当前文件目录。
- --datasets：数据集，格式为 datasetName:mountPath，例如 mnist:/data/mnist。

- --shm-size：共享内存大小，在分布式训练任务时，可以启用，表示使用共享内存，单位为 MiB。
- --labels：任务标签，格式为 key=value。
- --max-retries：最大重试次数，任务失败后重试次数，失败后会重启任务，默认不限制。
- --max-run-duration：最大运行时间，任务运行时间超过指定时间后，会被系统终止，默认不限制。
- --restart-policy：重启策略，支持 on-failure、never、always，默认为 on-failure。
- --from-notebook：是否从 Notebook 中读取配置，支持 auto、true、false，默认为 auto。

PyTorch 单机任务示例

提交训练任务示例，用户可以根据实际需求修改参数，以下为创建一个 PyTorch 任务的示例：

```
baizectl job submit --name demojob-v2 -t PYTORCH \
    --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --priority baize-high-priority \
    --requests-resources cpu=1,memory=1Gi \
    --workers 1 \
    --queue default \
    --working-dir /data \
    --datasets fashion-mnist:/data/mnist \
    --labels job_type=pytorch \
    --max-retries 3 \
    --max-run-duration 60 \
    --restart-policy on-failure \
    -- sleep 1000
```

PyTorch 分布式任务示例

```
baizectl job submit --name demojob-v2 -t PYTORCH \
    --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --priority baize-high-priority \
    --requests-resources cpu=1,memory=1Gi \
    --workers 2 \   # 多任务副本会自动创建分布式任务
    --shm-size 1024 \
    --queue default \
    --working-dir /data \
    --datasets fashion-mnist:/data/mnist \
    --labels job_type=pytorch \
    --max-retries 3 \
    --max-run-duration 60 \
    --restart-policy on-failure \
    -- sleep 1000
```

Tensorflow 任务示例

使用 `-t` 参数指定任务类型，以下为创建一个 Tensorflow 任务的示例：

```
baizectl job submit --name demojob-v2 -t TENSORFLOW \
    --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --priority baize-high-priority \
    --from-notebook auto \
    --workers 1 \
    --queue default \
    --working-dir /data \
    --datasets fashion-mnist:/data/mnist \
    --labels job_type=pytorch \
    --max-retries 3 \
    --max-run-duration 60 \
    --restart-policy on-failure \
    -- sleep 1000
```

也可以使用 `--job-type` 或者 `--tensorflow` 参数指定任务类型。

Paddle 任务示例

```
baizectl job submit --name demojob-v2 -t PADDLE \
    --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --priority baize-high-priority \
    --queue default \
    --working-dir /data \
    --datasets fashion-mnist:/data/mnist \
    --labels job_type=pytorch \
    --max-retries 3 \
    --max-run-duration 60 \
    --restart-policy on-failure \
    -- sleep 1000
```

查看任务列表

`baizectl job` 支持通过 `ls` 命令查看任务列表，默认显示 `pytorch` 任务，用户可以通过 `-t` 指定任务类型。

```
(base) jovyan@den-0:~$ baizectl job ls  # 默认查看 pytorch 任务
 NAME         TYPE     PHASE      DURATION  COMMAND
 demong       PYTORCH  SUCCEEDED  1m2s      sleep 60
 demo-sleep   PYTORCH  RUNNING    1h25m28s  sleep 7200
(base) jovyan@den-0:~$ baizectl job ls demo-sleep  # 查看指定任务
 NAME         TYPE     PHASE      DURATION  COMMAND
 demo-sleep   PYTORCH  RUNNING    1h25m28s  sleep 7200
(base) jovyan@den-0:~$ baizectl job ls -t TENSORFLOW   # 查看 tensorflow 任务
 NAME        TYPE        PHASE    DURATION  COMMAND
 demotfjob   TENSORFLOW  CREATED  0s        sleep 1000
```

任务列表默认情况下使用 `table` 作为展示形式，如果希望查看更多信息，可使用 `json` 或 `yaml` 格式展示，可以通过 `-o` 参数指定。

```
(base) jovyan@den-0:~$ baizectl job ls -t TENSORFLOW -o yaml
- baseConfig:
    args:
    - sleep
    - "1000"
    image: release.daocloud.io/baize/baize-notebook:v0.5.0
    labels:
      app: den
    podConfig:
      affinity: {}
      kubeEnvs:
      - name: CONDA_EXE
        value: /opt/conda/bin/conda
      - name: CONDA_PREFIX
        value: /opt/conda
      - name: CONDA_PROMPT_MODIFIER
        value: '(base) '
      - name: CONDA_SHLVL
        value: "1"
      - name: CONDA_DIR
        value: /opt/conda
      - name: CONDA_PYTHON_EXE
        value: /opt/conda/bin/python
      - name: CONDA_DEFAULT_ENV
        value: base
      - name: PATH
        value:
```

```
/opt/conda/bin:/opt/conda/condabin:/command:/opt/conda/bin:/usr/local/sbin:/usr/local/bi
n:/usr/sbin:/usr/bin:/sbin:/bin
      priorityClass: baize-high-priority
      queue: default
  creationTimestamp: "2024-06-16T07:47:27Z"
  jobSpec:
    runPolicy:
      suspend: true
    tfReplicaSpecs:
      Worker:
        replicas: 1
        restartPolicy: OnFailure
        template:
          metadata:
            creationTimestamp: null
          spec:
            affinity: {}
            containers:
            - args:
              - sleep
              - "1000"
              env:
              - name: CONDA_EXE
                value: /opt/conda/bin/conda
              - name: CONDA_PREFIX
                value: /opt/conda
              image: release.daocloud.io/baize/baize-notebook:v0.5.0
              name: tensorflow
              resources:
                limits:
                  memory: 1Gi
                requests:
                  cpu: "1"
                  memory: 2Gi
              workingDir: /home/jovyan
            priorityClassName: baize-high-priority
  name: demotfjob
  namespace: ns-chuanjia-ndx
  phase: CREATED
  roleConfig:
    TF_WORKER:
      replicas: 1
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: "1"
          memory: 2Gi
  totalResources:
    limits:
      memory: "1073741824"
    requests:
      cpu: "1"
      memory: "2147483648"
  trainingConfig:
    restartPolicy: RESTART_POLICY_ON_FAILURE
  trainingMode: SINGLE
  type: TENSORFLOW
```

查看任务日志

`baizectl job` 支持使用 `logs` 命令查看任务日志，用户可以通过 `baizectl job logs --help` 查看详细信息。

```
(base) jovyan@den-0:~$ baizectl job logs --help
Show logs of a job
```

```
Usage:
  baizectl job logs <job-name> [pod-name] [flags]

Aliases:
  logs, log

Flags:
  -f, --follow            Specify if the logs should be streamed.
  -h, --help              help for logs
  -t, --job-type string   Job type: PYTORCH, TENSORFLOW, PADDLE (default "PYTORCH")
      --paddle            PaddlePaddle Job, has higher priority than --job-type
      --pytorch           Pytorch Job, has higher priority than --job-type
      --tail int          Lines of recent log file to display.
      --tensorflow        Tensorflow Job, has higher priority than --job-type
      --timestamps        Show timestamps
```

- `--follow` 参数实时查看日志。
- `--tail` 参数指定查看日志的行数，默认为 50 行。
- `--timestamps` 参数显示时间戳。

示例查看任务日志：

```
(base) jovyan@den-0:~$ baizectl job log -t TENSORFLOW tf-sample-job-v2-202406161632-
evgrbrhn -f
2024-06-16 08:33:06.083766: I tensorflow/core/util/port.cc:110] oneDNN custom operations
are on.
2024-06-16 08:33:06.086189: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda
drivers on your machine, GPU will not be used.
2024-06-16 08:33:07.223046: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
TRT Warning: Could not find TensorRT
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Conv1 (Conv2D)              (None, 13, 13, 8)         80
 flatten (Flatten)           (None, 1352)              0
 Softmax (Dense)             (None, 10)                13530
=================================================================
Total params: 13610 (53.16 KB)
Trainable params: 13610 (53.16 KB)
Non-trainable params: 0 (0.00 Byte)
...
```

删除任务

`baizectl job` 支持使用 `delete` 命令删除任务，并且同时支持删除多个任务。

```
(base) jovyan@den-0:~$ baizectl job delete --help
Delete a job

Usage:
  baizectl job delete [flags]

Aliases:
  delete, del, remove, rm

Flags:
  -h, --help              help for delete
  -t, --job-type string   Job type: PYTORCH, TENSORFLOW, PADDLE (default "PYTORCH")
      --paddle            PaddlePaddle Job, has higher priority than --job-type
      --pytorch           Pytorch Job, has higher priority than --job-type
      --tensorflow        Tensorflow Job, has higher priority than --job-type
```

示例删除任务：

```
(base) jovyan@den-0:~$ baizectl job ls
 NAME         TYPE     PHASE      DURATION  COMMAND
 demong       PYTORCH  SUCCEEDED  1m2s      sleep 60
 demo-sleep   PYTORCH  RUNNING    1h20m51s  sleep 7200
 demojob      PYTORCH  FAILED     16m46s    sleep 1000
 demojob-v2   PYTORCH  RUNNING    3m13s     sleep 1000
 demojob-v3   PYTORCH  CREATED    0s        sleep 1000
(base) jovyan@den-0:~$ baizectl job delete demojob      # 删除单个任务
Delete job demojob in ns-chuanjia-ndx successfully
(base) jovyan@den-0:~$ baizectl job delete demojob-v2 demojob-v3     # 删除多个任务
Delete job demojob-v2 in ns-chuanjia-ndx successfully
Delete job demojob-v3 in ns-chuanjia-ndx successfully
```

重启任务

`baizectl job` 支持使用 `restart` 命令重启任务，用户可以通过 `baizectl job restart --help` 查看详细信息。

```
(base) jovyan@den-0:~$ baizectl job restart --help
restart a job

Usage:
  baizectl job restart [flags] job

Aliases:
  restart, rerun

Flags:
  -h, --help              help for restart
  -t, --job-type string   Job type: PYTORCH, TENSORFLOW, PADDLE (default "PYTORCH")
      --paddle            PaddlePaddle Job, has higher priority than --job-type
      --pytorch           Pytorch Job, has higher priority than --job-type
      --tensorflow        Tensorflow Job, has higher priority than --job-type
```

功能介绍：数据集管理

`baizectl` 支持管理数据集，目前支持查看数据集列表，方便在任务训练时，快速绑定数据集。

```
(base) jovyan@den-0:~$ baizectl data
Management datasets

Usage:
  baizectl data [flags]
  baizectl data [command]

Aliases:
  data, dataset, datasets, envs, runtime-envs

Available Commands:
  ls          List datasets

Flags:
  -h, --help            help for data
  -o, --output string   Output format. One of: table, json, yaml (default "table")
      --page int        Page number (default 1)
      --page-size int   Page size (default -1)
      --search string   Search query
      --sort string     Sort order
      --truncate int    Truncate output to the given length, 0 means no truncation
(default 50)
```

```
Use "baizectl data [command] --help" for more information about a command.
```

查看数据集列表

`baizectl data` 支持通过 `ls` 命令查看数据集列表，默认显示 `table` 格式，用户可以通过 `-o` 参数指定输出格式。

```
(base) jovyan@den-0:~$ baizectl data ls
 NAME             TYPE  URI                                                  PHASE
 fashion-mnist    GIT   https://gitee.com/samzong_lu/fashion-mnist.git       READY
 sample-code      GIT   https://gitee.com/samzong_lu/training-sample-code....  READY
 training-output  PVC   pvc://training-output                                READY
```

在提交训练任务时，可以使用 `-d` 或者 `--datasets` 参数指定数据集，例如：

```
baizectl job submit --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --datasets sample-code:/home/jovyan/code \
    -- sleep 1000
```

同时挂载多个数据集，可以按照如下格式：

```
baizectl job submit --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --datasets sample-code:/home/jovyan/code fashion-mnist:/home/jovyan/data \
    -- sleep 1000
```

查看依赖库（环境）

环境 `runtime-env` 是 DCE 的特色环境管理能力，通过将模型开发、训练任务以及推理中所需的依赖库解耦，提供了一种更加灵活的依赖库管理方式，无需重复构建复杂的 Docker 镜像，只需选择合适的环境即可。

同时 `runtime-env` 支持热更新，动态升级，无需重新构建镜像，即可更新环境依赖库。

`baizectl data` 支持通过 `runtime-env` 命令查看环境列表，默认显示 `table` 格式，用户可以通过 `-o` 参数指定输出格式。

```
(base) jovyan@den-0:~$ baizectl data ls --runtime-env
 NAME               TYPE   URI                                                  PHASE
 fashion-mnist      GIT    https://gitee.com/samzong_lu/fashion-mnist.git       READY
 sample-code        GIT    https://gitee.com/samzong_lu/training-sample-code....  READY
 training-output    PVC    pvc://training-output                                READY
 tensorflow-sample  CONDA  conda://python?version=3.12.3
PROCESSING
```

在提交训练任务时，可以使用 `--runtime-env` 参数指定环境，例如：

```
baizectl job submit --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --runtime-env tensorflow-sample \
    -- sleep 1000
```

高级用法

baizectl 支持更多高级用法，例如自动补全脚本生成、使用特定集群和命名空间、使用特定工作空间等。

自动补全脚本生成

```
baizectl completion bash > /etc/bash_completion.d/baizectl
```

上述命令生成 `bash` 的自动补全脚本，并将其保存到 `/etc/bash_completion.d/baizectl` 目录中，用户可以通过 `source /etc/bash_completion.d/baizectl` 加载自动补全脚本。

使用特定集群和命名空间

```
baizectl job ls --cluster my-cluster --namespace my-namespace
```

该命令将列出 `my-cluster` 集群中 `my-namespace` 命名空间下的所有作业。

使用特定工作空间

```
baizectl job ls --workspace 123
```

常见问题

- **问题**：为什么无法连接到服务器？**解决方法**：检查 `--server` 参数是否正确设置，并确保网络连接正常。如果服务器使用自签名证书，可以使用 `--skip-tls-verify` 跳过 TLS 证书验证。
- **问题**：如何解决权限不足的问题？**解决方法**：确保使用正确的 `--token` 参数登录，并检查当前用户是否具有相应的操作权限。
- **问题**：为什么无法列出数据集？**解决方法**：检查命名空间和工作区是否正确设置，确保当前用户有权限访问这些资源。

通过以上指南，用户可以快速上手 `baizectl` 命令，并在实际应用中高效地管理 AI 平台资源。如果有任何疑问或问题，建议参考 `baizectl [command] --help` 获取更多详细信息。

（十一）baizess 换源工具

`baizess` 是 DCE AI Lab 模块中 Notebook 内置的开箱即用的换源小工具。它提供了简洁的命令行界面，方便用户管理各种编程环境的包管理器源。通过 baizess，用户可以轻松切换常用包管理器的源，确保顺利访问最新的库和依赖项。该工具通过简化包源管理流程，提升了开发者和数据科学家的工作效率。

安装

目前，`baizess` 已经集成在 DCE AI Lab 中。你在创建 Notebook 后，即可在 Notebook 中直接使用 `baizess`。

快速上手

基本信息

`baizess` 命令的基本信息如下：

```
jovyan@19d0197587cc:/$ baizess
source switch tool

Usage:
  baizess [command] [package-manager]

Available Commands:
  set     Switch the source of specified package manager to current fastest source
  reset   Reset the source of specified package manager to default source

Available Package-managers:
  apt     (require root privilege)
  conda
  pip
```

命令格式

`baizess` 命令的基本格式如下：

```
baizess [command] [package-manager]
```

其中，`[command]` 是具体的操作命令，`[package-manager]` 用于指定操作对应的包管理器。

command：

- `set`：备份源，测速，将所指定的包管理器的源切换为测速结果最快的国内源。
- `reset`：将所指定的包管理器重置为默认源。

目前支持的 package-manager：

- `apt`（源的切换与重置需要 `root` 权限）
- `conda`（原先的源将被备份在 `/etc/apt/backup/`）
- `pip`（更新后源信息将被写入 `~/.condarc`）

四、任务管理

任务管理是指通过作业调度和管控组件来创建和管理任务生命周期的功能。AI Lab 采用 Kubernetes 的 Job 机制来调度各项 AI 推理、训练任务，支持 Pytorch、Tensorflow、PaddlePaddle、MPI、MXNet 等主流框架的单机与分布式训练。

（一）创建任务

DCE AI Lab 采用 Kubernetes 的 Job 机制来调度各项 AI 推理、训练任务。

1. 在左侧导航栏中点击 **任务中心** -> **训练任务**，点击右侧的 **创建** 按钮。

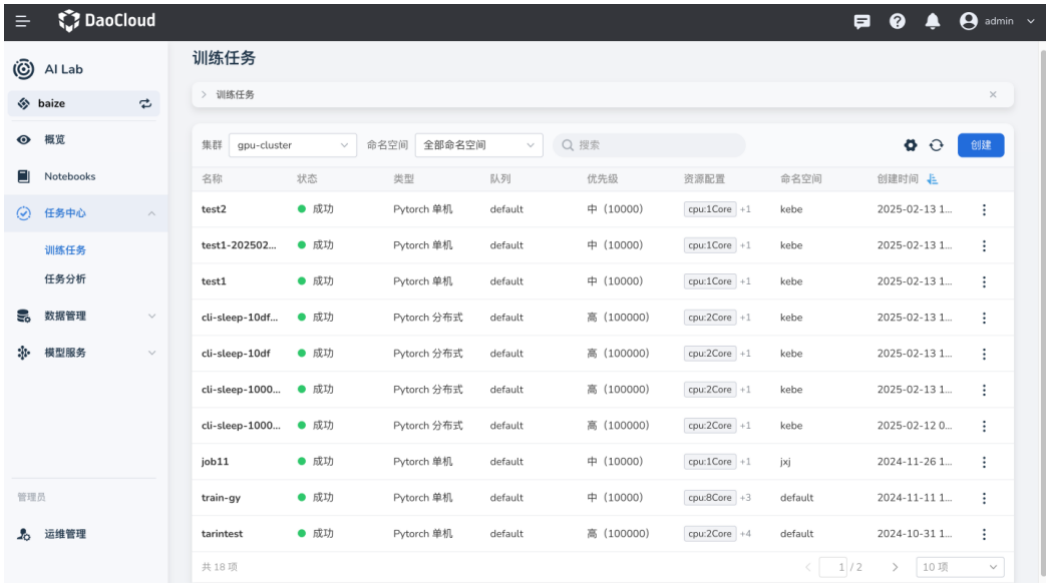

图 54　训练任务列表点击创建

2. 系统会预先填充基础配置数据，包括要部署的集群、命名空间、任务类型、队列、优先级等。调整这些参数后点击 **下一步**。

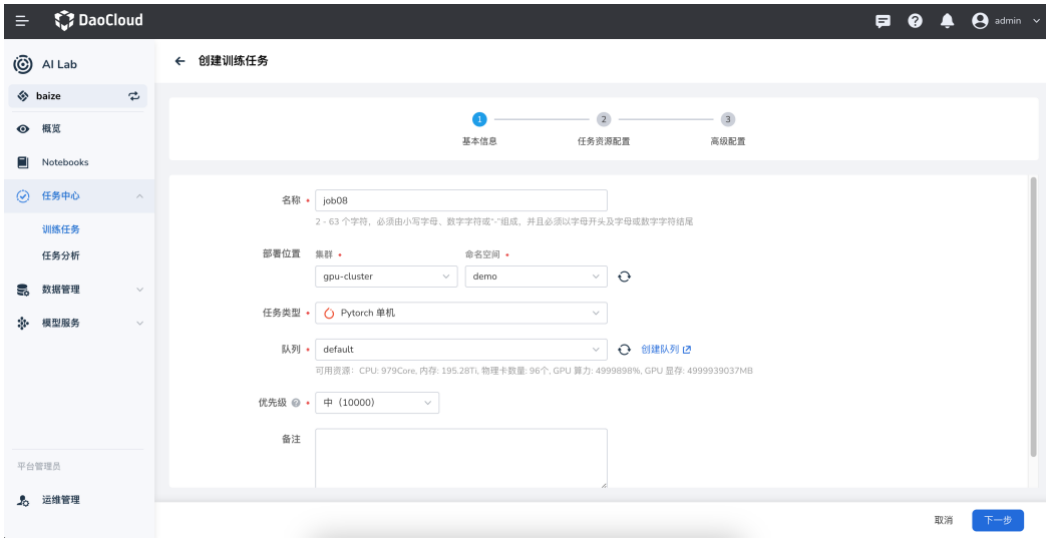

图 55　填写基础配置数据

3. 配置镜像地址、运行参数以及关联的数据集、环境和资源后，点击 **下一步**。

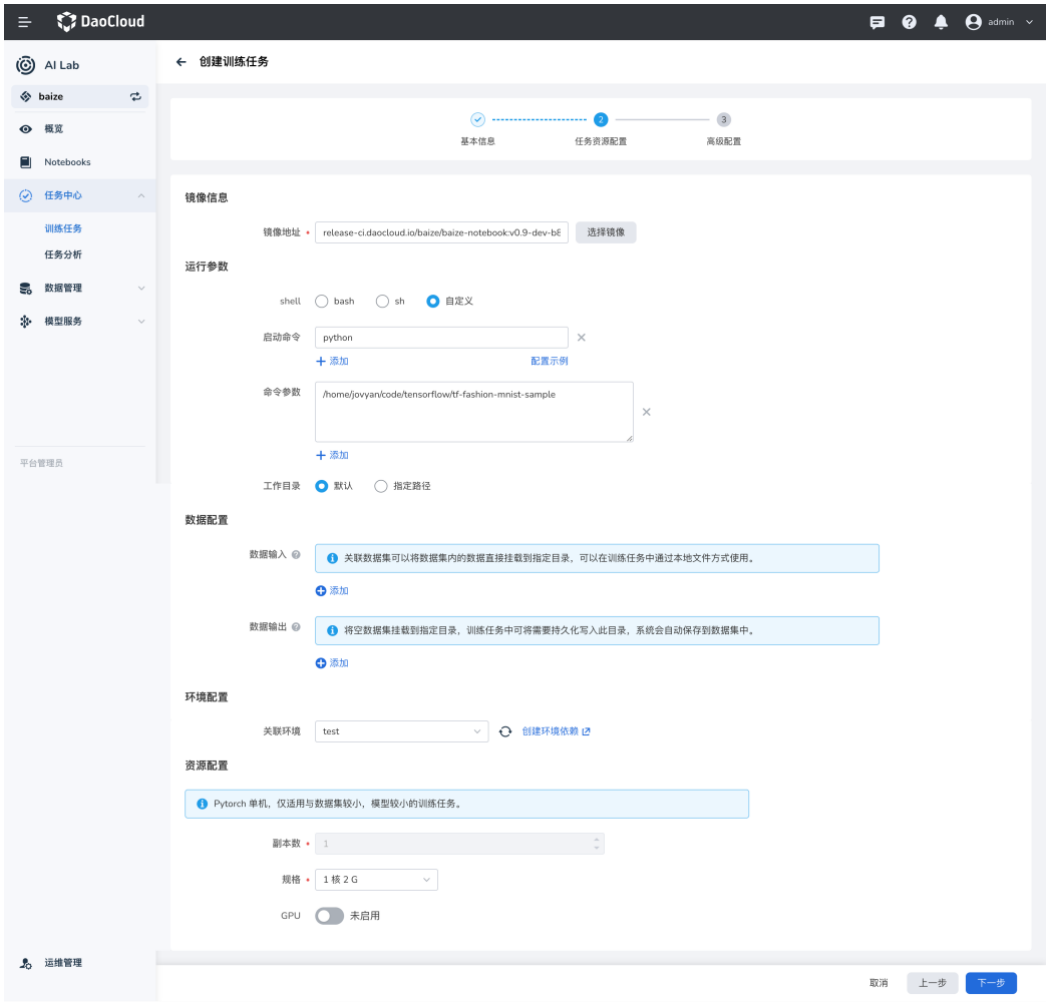

图 56　配置任务资源

4. 按需添加标签、注解、环境变量等任务参数，选择调度策略后点击 **确定**。

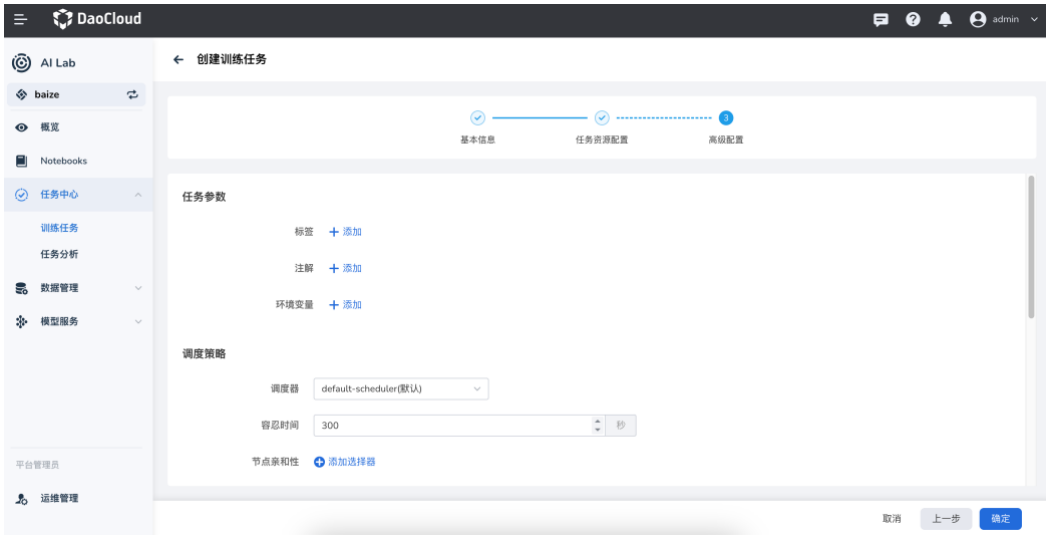

图 57　任务高级配置

5. 任务创建成功后，会有几种运行状态：

- 运行中
- 排队中
- 提交成功、提交失败
- 任务成功、任务失败

（二）Pytorch 任务

Pytorch 是一个开源的深度学习框架，它提供了一个灵活的训练和部署环境。Pytorch 任务是一个使用 Pytorch 框架的任务。

在 AI Lab 中，我们提供了 Pytorch 任务支持和适配，您可以通过界面化操作快速创建 Pytorch 任务，进行模型训练。

任务配置介绍

- 任务类型同时支持 `Pytorch 单机` 和 `Pytorch 分布式` 两种模式。
- 默认的 `baize-notebook` 镜像不预装深度学习框架，请先在环境管理中安装 PyTorch 或 TensorFlow，并将其挂载到任务运行环境。

任务运行环境

在这里我们使用 `baize-notebook` 基础镜像和 `关联环境` 的方式来作为任务基础运行环境。挂载环境后，需通过 `conda activate <env_name> && python - <<'EOF'` 的形式在 Conda 环境中执行任务命令。

创建 Pytorch 单机任务

图 58　创建 Pytorch 单机任务

1. 登录 AI Lab 平台，点击左侧导航栏中的 **任务中心**，进入 **训练任务** 页面。
2. 点击右上角的 **创建** 按钮，进入任务创建页面。
3. 选择任务类型为 `Pytorch 单机`，点击 **下一步**。
4. 填写任务名称、描述后点击 **确定**。

运行参数：

- 启动命令使用 `bash`
- 命令参数参考下方示例

- 将 `<env_name>` 替换为已挂载的 Conda 环境名称（例如 `torch-env`）

```
conda activate <env_name> && python - <<'EOF'
import torch
import torch.nn as nn
import torch.optim as optim

# 定义一个简单的神经网络
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc = nn.Linear(10, 1)

    def forward(self, x):
        return self.fc(x)

# 创建模型、损失函数和优化器
model = SimpleNet()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# 生成一些随机数据
x = torch.randn(100, 10)
y = torch.randn(100, 1)

# 训练模型
for epoch in range(100):
    # 前向传播
    outputs = model(x)
    loss = criterion(outputs, y)

    # 反向传播和优化
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')

print('Training finished.')
EOF
```

以上 heredoc 写法适合快速临时验证。正式训练时，建议通过挂载数据空间引入真实的数据集和训练代码，并直接在命令中调用这些文件。

运行结果：任务提交成功，我们可以进入任务详情查看到资源的使用情况，从右上角去往**工作负载详情**，可以查看训练过程中的日志输出。

```
[HAMI-core Warn(1:140244541377408:utils.c:183)]: get default cuda from (null)
[HAMI-core Msg(1:140244541377408:libvgpu.c:855)]: Initialized
Epoch [10/100], Loss: 1.1248
Epoch [20/100], Loss: 1.0486
Epoch [30/100], Loss: 0.9969
Epoch [40/100], Loss: 0.9611
Epoch [50/100], Loss: 0.9360
Epoch [60/100], Loss: 0.9182
Epoch [70/100], Loss: 0.9053
Epoch [80/100], Loss: 0.8960
Epoch [90/100], Loss: 0.8891
Epoch [100/100], Loss: 0.8841
Training finished.
[HAMI-core Msg(1:140244541377408:multiprocess_memory_limit.c:468)]: Calling exit handler
1
```

创建 Pytorch 分布式任务

1. 登录 AI Lab 平台，点击左侧导航栏中的 **任务中心**，进入 **任务列表** 页面。
2. 点击右上角的 **创建** 按钮，进入任务创建页面。
3. 选择任务类型为 `Pytorch 分布式`，点击 **下一步**。
4. 填写任务名称、描述后点击 **确定**。

运行参数：启动命令使用 `bash`，命令参数参考下方示例，将 `<env_name>` 替换为已挂载的 Conda 环境名称（例如 `torch-env`）。

```
conda activate <env_name> && python - <<'EOF'
import os
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(10, 1)

    def forward(self, x):
        return self.fc(x)

def train():
    # 打印环境信息
    print(f'PyTorch version: {torch.__version__}')
    print(f'CUDA available: {torch.cuda.is_available()}')
    if torch.cuda.is_available():
        print(f'CUDA version: {torch.version.cuda}')
        print(f'CUDA device count: {torch.cuda.device_count()}')

    rank = int(os.environ.get('RANK', '0'))
    world_size = int(os.environ.get('WORLD_SIZE', '1'))

    print(f'Rank: {rank}, World Size: {world_size}')

    # 初始化分布式环境
    try:
        if world_size > 1:
            dist.init_process_group('nccl')
            print('Distributed process group initialized successfully')
        else:
            print('Running in non-distributed mode')
    except Exception as e:
        print(f'Error initializing process group: {e}')
        return

    # 设置设备
    try:
        if torch.cuda.is_available():
            device = torch.device(f'cuda:{rank % torch.cuda.device_count()}')
            print(f'Using CUDA device: {device}')
        else:
            device = torch.device('cpu')
            print('CUDA not available, using CPU')
    except Exception as e:
        print(f'Error setting device: {e}')
```

```
        device = torch.device('cpu')
        print('Falling back to CPU')

    try:
        model = SimpleModel().to(device)
        print('Model moved to device successfully')
    except Exception as e:
        print(f'Error moving model to device: {e}')
        return

    try:
        if world_size > 1:
            ddp_model = DDP(model, device_ids=[rank % torch.cuda.device_count()] if
torch.cuda.is_available() else None)
            print('DDP model created successfully')
        else:
            ddp_model = model
            print('Using non-distributed model')
    except Exception as e:
        print(f'Error creating DDP model: {e}')
        return

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # 生成一些随机数据
    try:
        data = torch.randn(100, 10, device=device)
        labels = torch.randn(100, 1, device=device)
        print('Data generated and moved to device successfully')
    except Exception as e:
        print(f'Error generating or moving data to device: {e}')
        return

    for epoch in range(10):
        try:
            ddp_model.train()
            outputs = ddp_model(data)
            loss = loss_fn(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if rank == 0:
                print(f'Epoch {epoch}, Loss: {loss.item():.4f}')
        except Exception as e:
            print(f'Error during training epoch {epoch}: {e}')
            break

    if world_size > 1:
        dist.destroy_process_group()

if __name__ == '__main__':
    train()
EOF
```

同样地，该命令主要用于临时测试。生产训练任务请挂载包含数据集和训练脚本的数据空间，在容器内直接引用这些文件。

任务副本数

注意 `Pytorch 分布式` 训练任务会创建一组 `Master` 和 `Worker` 的训练 Pod，`Master` 负责协调训练任务，`Worker` 负责实际的训练工作。

> 本次演示中：Master 副本数为 1，Worker 副本数为 2；所以我们需要在 **任务配置** 中设置副本数为 3，即 Master 副本数 + Worker 副本数。Pytorch 会自动调谐 Master 和 Worker 的角色。

运行结果：同样，我们可以进入任务详情，查看资源的使用情况，以及每个 Pod 的日志输出。

（三）Tensorflow 任务

Tensorflow 是除了 Pytorch 另外一个非常活跃的开源的深度学习框架，它提供了一个灵活的训练和部署环境。

在 AI Lab 中，我们同样提供了 Tensorflow 框架的支持和适配，您可以通过界面化操作，快速创建 Tensorflow 任务，进行模型训练。

任务配置介绍

- 任务类型同时支持 `Tensorflow 单机` 和 `Tensorflow 分布式` 两种模式。
- 运行镜像内已经默认支持 Tensorflow 框架，无需额外安装。

任务运行环境：在这里我们使用 `baize-notebook` 基础镜像和 `关联环境` 的方式来作为任务基础运行环境。

示例 TFJob 单机任务

图 59 创建 Tensorflow 单机任务

1. 登录 AI Lab 平台，点击左侧导航栏中的 **任务中心**，进入 **训练任务** 页面。
2. 点击右上角的 **创建** 按钮，进入任务创建页面。
3. 选择任务类型为 `Tensorflow 单机`，点击 **下一步**。
4. 填写任务名称、描述后点击 **确定**。

提前预热代码仓库

使用 AI Lab -> **数据集列表**，创建一个数据集，并将远端 Github 的代码拉取到数据集中，这样在创建任务时，可以直接选择数据集，将代码挂载到任务中。

演示代码仓库地址：`https://github.com/d-run/training-sample-code/`

运行参数：启动命令使用 `bash`，命令参数使用 `python /code/tensorflow/tf-single.py`。

```
"""
  pip install tensorflow numpy
"""

import tensorflow as tf
import numpy as np

# 创建一些随机数据
x = np.random.rand(100, 1)
y = 2 * x + 1 + np.random.rand(100, 1) * 0.1

# 创建一个简单的模型
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=(1,))
])

# 编译模型
model.compile(optimizer='adam', loss='mse')

# 训练模型，将 epochs 改为 10
history = model.fit(x, y, epochs=10, verbose=1)

# 打印最终损失
print('Final loss: {' + str(history.history['loss'][-1]) +'}')

# 使用模型进行预测
test_x = np.array([[0.5]])
prediction = model.predict(test_x)
print(f'Prediction for x=0.5: {prediction[0][0]}')
```

运行结果：任务提交成功后，可以进入任务详情查看到资源的使用情况，从右上角去往 **工作负载详情**，可以查看训练过程中的日志输出。

TFJob 分布式任务

1. 登录 AI Lab，点击左侧导航栏中的 **任务中心**，进入 **任务列表** 页面。
2. 点击右上角的 **创建** 按钮，进入任务创建页面。
3. 选择任务类型为 `Tensorflow 分布式`，点击 **下一步**。
4. 填写任务名称、描述后点击 **确定**。

镜像信息

镜像地址 * release-ci.daocloud.io/baize/baize-notebook:v0.8-dev-ef752fd3

运行参数

shell ● bash ○ sh ○ 自定义

启动命令 * /bin/bash

-c

配置示例

命令参数 python /code/tensorflow/tensorflow-distributed.py

工作目录 ● 默认 ○ 指定路径

数据配置

通过数据集可以将远端数据（NFS、S3、http、git）挂载到本地使用，可以加速数据访问。

关联数据集

数据集 * training-sample-code 创建数据集

挂载路径 * /code

添加

环境配置

关联环境 tensorflow 创建环境

资源配置

节点配置

Tensorflow 分布式任务，可以自行添加 PS、Chief、Evaluator 角色节点。

WORKER

副本数 * 2

规格 * 1 核 2 G

GPU 启用

GPU 类型 *

Nvidia vGPU

容器可使用算力 = 物理卡数量 * 使用量

物理卡数量 * 1 个

GPU 算力 50 %

GPU 显存 200 MB

CHIEF

副本数 * 1

规格 * 1 核 2 G

GPU 启用

GPU 类型 *

Nvidia vGPU

容器可使用算力 = 物理卡数量 * 使用量

物理卡数量 * 1 个

GPU 算力 50 %

GPU 显存

图 60　创建 Tensorflow 分布式任务

示例任务介绍：本次包含了三种角色：Chief、Worker 和 Parameter Server (PS)。

- Chief：主要负责协调训练过程和模型检查点的保存。
- Worker：执行实际的模型训练。
- PS：在异步训练中用于存储和更新模型参数。

为不同的角色分配了不同的资源。Chief 和 Worker 使用 GPU，而 PS 使用 CPU 和较大的内存。

运行参数：启动命令使用 bash，命令参数使用 python /code/tensorflow/tensorflow-distributed.py。

```
import os
import json
import tensorflow as tf

class SimpleModel(tf.keras.Model):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = tf.keras.layers.Dense(1, input_shape=(10,))

    def call(self, x):
        return self.fc(x)

def train():
    # 打印环境信息
    print(f"TensorFlow version: {tf.__version__}")
    print(f"GPU available: {tf.test.is_gpu_available()}")
    if tf.test.is_gpu_available():
        print(f"GPU device count: {len(tf.config.list_physical_devices('GPU'))}")

    # 获取分布式训练信息
    tf_config = json.loads(os.environ.get('TF_CONFIG') or '{}')
    task_type = tf_config.get('task', {}).get('type')
    task_id = tf_config.get('task', {}).get('index')

    print(f"Task type: {task_type}, Task ID: {task_id}")

    # 设置分布式策略
    strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

    with strategy.scope():
        model = SimpleModel()
        loss_fn = tf.keras.losses.MeanSquaredError()
        optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)

    # 生成一些随机数据
    data = tf.random.normal((100, 10))
    labels = tf.random.normal((100, 1))

    @tf.function
    def train_step(inputs, labels):
        with tf.GradientTape() as tape:
            predictions = model(inputs)
            loss = loss_fn(labels, predictions)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        return loss
```

```
    for epoch in range(10):
        loss = train_step(data, labels)
        if task_type == 'chief':
            print(f'Epoch {epoch}, Loss: {loss.numpy():.4f}')

if __name__ == '__main__':
    train()
```

运行结果：同样，我们可以进入任务详情，查看资源的使用情况，以及每个 Pod 的日志输出。

（四）PaddlePaddle 任务

PaddlePaddle（飞桨）是百度开源的深度学习平台，支持丰富的神经网络模型和分布式训练方式。PaddlePaddle 任务可以通过单机或分布式模式进行训练。在 AI Lab 平台中，我们提供了对 PaddlePaddle 任务的支持，您可以通过界面化操作，快速创建 PaddlePaddle 任务，进行模型训练。

任务配置介绍

- **任务类型：**PaddlePaddle，支持单机和分布式两种模式。
- **运行环境：**选择包含 PaddlePaddle 框架的镜像，或在任务中安装必要的依赖。

我们使用 registry.baidubce.com/paddlepaddle/paddle:2.4.0rc0-cpu 镜像作为任务的基础运行环境。该镜像预装了 PaddlePaddle 框架，适用于 CPU 计算。如果需要使用 GPU，请选择对应的 GPU 版本镜像。

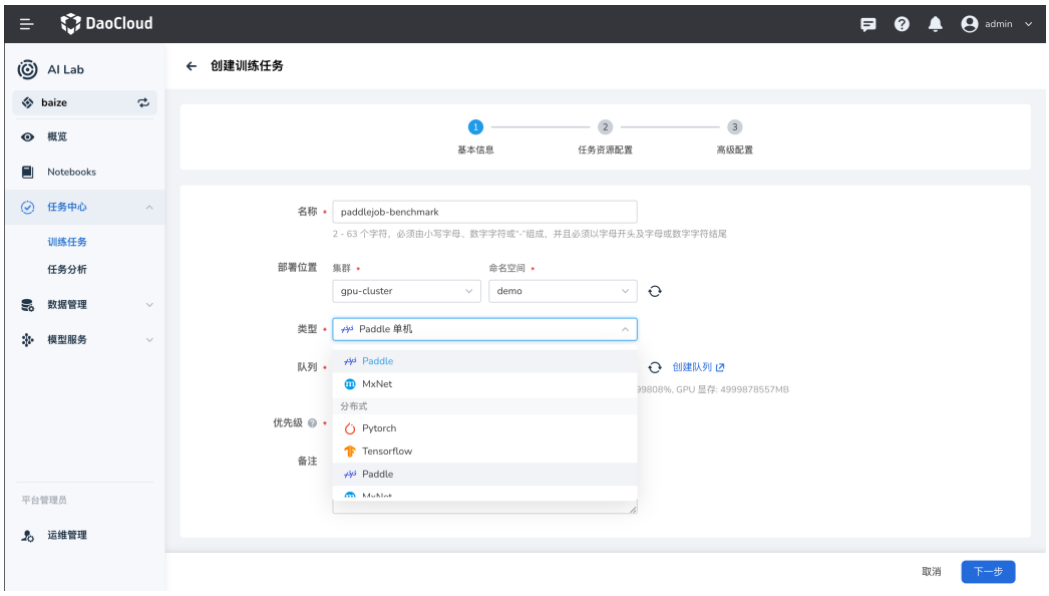

图 61　创建 PaddlePaddle 任务

PaddlePaddle 单机训练任务

创建步骤

1. 登录平台：登录 AI Lab 平台，点击左侧导航栏中的 **任务中心**，进入 **训练任务** 页面。
2. 创建任务：点击右上角的 **创建** 按钮，进入任务创建页面。
3. 选择任务类型：在弹出的窗口中，选择任务类型为 `PaddlePaddle`，然后点击 **下一步**。
4. 填写任务信息：填写任务名称和描述，例如「PaddlePaddle 单机训练任务」，然后点击 **确定**。
5. 配置任务参数：根据您的需求，配置任务的运行参数、镜像、资源等信息。

运行参数：启动命令为 `python`，命令参数如下：

```
-m paddle.distributed.launch run_check
```

- `-m paddle.distributed.launch`：使用 PaddlePaddle 提供的分布式启动模块，即使在单机模式下也可以使用，方便将来迁移到分布式。
- `run_check`：PaddlePaddle 提供的测试脚本，用于检查分布式环境是否正常。

资源配置：

- **副本数**：1（单机任务）
- **资源请求**：
  - CPU：根据需求设置，建议至少 1 核
  - 内存：根据需求设置，建议至少 2 GiB
  - GPU：如果需要使用 GPU，选择 GPU 版本的镜像，并分配相应的 GPU 资源

完整的 PaddleJob 配置示例如下：

```
apiVersion: kubeflow.org/v1
kind: PaddleJob
metadata:
    name: paddle-simple-cpu
    namespace: kubeflow
spec:
    paddleReplicaSpecs:
        Worker:
            replicas: 1
            restartPolicy: OnFailure
            template:
                spec:
                    containers:
                        - name: paddle
                          image: registry.baidubce.com/paddlepaddle/paddle:2.4.0rc0-cpu
                          command:
                              [
                                  'python',
                                  '-m',
                                  'paddle.distributed.launch',
                                  'run_check',
                              ]
```

配置解析：

- `apiVersion` 和 `kind`：指定资源的 API 版本和类型，这里是 `PaddleJob`。

- `metadata`：元数据，包括任务名称和命名空间。
- `spec`：任务的详细配置。
  - `paddleReplicaSpecs`：PaddlePaddle 任务的副本配置。
    - `Worker`：指定工作节点的配置。
    - `replicas`：副本数，这里为 1，表示单机训练。
    - `restartPolicy`：重启策略，设为 `OnFailure`，表示任务失败时自动重启。
    - `template`：Pod 模板，定义容器的运行环境和资源。

提交任务：配置完成后，点击 **提交** 按钮，开始运行 PaddlePaddle 单机任务。

查看运行结果：任务提交成功后，您可以进入 **任务详情** 页面，查看资源的使用情况和任务的运行状态。从右上角进入 **工作负载详情**，可以查看运行过程中的日志输出。

示例输出：

```
run check success, PaddlePaddle is installed correctly on this node :)
```

这表示 PaddlePaddle 单机任务成功运行，环境配置正常。

PaddlePaddle 分布式训练任务

在分布式模式下，PaddlePaddle 任务可以使用多台计算节点共同完成训练，提高训练效率。

创建步骤：同单机模式，选择任务类型为 `PaddlePaddle` 后填写任务名称与描述，再配置运行参数、镜像、资源等。

运行参数：启动命令为 `python`，命令参数如下：

```
-m paddle.distributed.launch train.py --epochs=10
```

- `-m paddle.distributed.launch`：使用 PaddlePaddle 提供的分布式启动模块。
- `train.py`：您的训练脚本，需要放在镜像中或挂载到容器内。
- `--epochs=10`：训练的轮数，这里设置为 10。

资源配置：任务副本数根据 `Worker` 副本数设置，这里为 2；CPU 建议至少 1 核，内存建议至少 2 GiB；如需使用 GPU，选择 GPU 版本的镜像并分配相应资源。

完整的分布式 PaddleJob 配置示例如下：

```
apiVersion: kubeflow.org/v1
kind: PaddleJob
metadata:
    name: paddle-distributed-job
    namespace: kubeflow
spec:
    paddleReplicaSpecs:
        Worker:
            replicas: 2
```

```
            restartPolicy: OnFailure
            template:
                spec:
                    containers:
                        - name: paddle
                          image: registry.baidubce.com/paddlepaddle/paddle:2.4.0rc0-cpu
                          command:
                              [
                                  'python',
                                  '-m',
                                  'paddle.distributed.launch',
                                  'train.py',
                              ]
                          args:
                              - '--epochs=10'
```

设置任务副本数：总副本数 = `Worker` 副本数。本示例中 `Worker` 副本数为 2，因此需要在任务配置中将 **任务副本数** 设置为 2。

提交任务：配置完成后，点击 **提交** 按钮，开始运行 PaddlePaddle 分布式任务。

查看运行结果：进入 **任务详情** 页面，查看任务的运行状态和资源使用情况。您可以查看每个工作节点的日志输出，确认分布式训练是否正常运行。

示例输出：

```
Worker 0: Epoch 1, Batch 100, Loss 0.5
Worker 1: Epoch 1, Batch 100, Loss 0.6
...
Training completed.
```

附录 • 注意事项：

- **训练脚本**：确保 `train.py`（或其他训练脚本）在容器内存在。您可以通过自定义镜像、挂载持久化存储等方式将脚本放入容器。
- **镜像选择**：根据您的需求选择合适的镜像，例如使用 GPU 时选择 `paddle:2.4.0rc0-gpu` 等。
- **参数调整**：可以通过修改 `command` 和 `args` 来传递不同的训练参数。

（五）MPI 任务

MPI（Message Passing Interface）是一种用于并行计算的通信协议，它允许多个计算节点之间进行消息传递和协作。MPI 任务是使用 MPI 协议进行并行计算的任务，适用于需要大规模并行处理的应用场景，例如分布式训练、科学计算等。

在 AI Lab 中，我们提供了 MPI 任务的支持，您可以通过界面化操作，快速创建 MPI 任务，进行高性能的并行计算。

任务配置介绍

- **任务类型**：`MPI`，用于运行并行计算任务。
- **运行环境**：选用预装了 MPI 环境的镜像，或者在任务中指定安装必要的依赖。

- **MPIJob 配置**：理解并配置 MPIJob 的各项参数，如副本数、资源请求等。

在这里我们使用 `baize-notebook` 基础镜像和 **关联环境** 的方式来作为任务的基础运行环境。确保运行环境中包含 MPI 及相关库，如 OpenMPI、`mpi4py` 等。

创建 MPI 任务

图 62　创建 MPI 任务

1. **登录平台**：登录 AI Lab 平台，点击左侧导航栏中的 **任务中心**，进入 **训练任务** 页面。
2. **创建任务**：点击右上角的 **创建** 按钮，进入任务创建页面。
3. **选择任务类型**：在弹出的窗口中，选择任务类型为 `MPI`，然后点击 **下一步**。
4. **填写任务信息**：填写任务名称和描述，例如「benchmarks-mpi」，然后点击 **下一步**。
5. **配置任务参数**：根据您的需求，配置任务的运行参数、镜像、资源等信息。

运行参数

- **启动命令**：使用 `mpirun`，这是运行 MPI 程序的命令。
- **命令参数**：输入您要运行的 MPI 程序的参数。

示例：运行 TensorFlow Benchmarks

在本示例中，我们将运行一个 TensorFlow 的基准测试程序，使用 Horovod 进行分布式训练。首先，确保您使用的镜像中包含所需的依赖项，例如 TensorFlow、Horovod、Open MPI 等。

镜像选择：使用包含 TensorFlow 和 MPI 的镜像，例如 `mai.daocloud.io/docker.io/mpioperator/tensorflow-benchmarks:latest`。

命令参数：

```
mpirun --allow-run-as-root -np 2 -bind-to none -map-by slot \
  -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
  -mca pml ob1 -mca btl ^openib \
  python scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
  --model=resnet101 --batch_size=64 --variable_update=horovod
```

说明：

- `mpirun`：MPI 的启动命令。
- `--allow-run-as-root`：允许以 root 用户运行（在容器中通常是 root 用户）。
- `-np 2`：指定运行的进程数为 2。
- `-bind-to none`、`-map-by slot`：MPI 进程绑定和映射的配置。
- `-x NCCL_DEBUG=INFO`：设置 NCCL（NVIDIA Collective Communication Library）的调试信息级别。
- `-x LD_LIBRARY_PATH`、`-x PATH`：在 MPI 环境中传递必要的环境变量。
- `-mca pml ob1 -mca btl ^openib`：MPI 的配置参数，指定传输层和消息层协议。
- `python scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py`：运行 TensorFlow 基准测试脚本。
- `--model=resnet101`、`--batch_size=64`、`--variable_update=horovod`：TensorFlow 脚本的参数，指定模型、批量大小和使用 Horovod 进行参数更新。

资源配置

在任务配置中，需要为每个节点（Launcher 和 Worker）分配适当的资源，例如 CPU、内存和 GPU。

资源示例：

- **Launcher（启动器）**：副本数 1；CPU 2 核、内存 4 GiB。
- **Worker（工作节点）**：副本数 2；CPU 2 核、内存 4 GiB；GPU 根据需求分配。

完整的 MPIJob 配置示例

```
apiVersion: kubeflow.org/v1
kind: MPIJob
metadata:
  name: tensorflow-benchmarks
spec:
  slotsPerWorker: 1
  runPolicy:
    cleanPodPolicy: Running
  mpiReplicaSpecs:
    Launcher:
      replicas: 1
      template:
        spec:
          containers:
            - name: tensorflow-benchmarks
              image: mai.daocloud.io/docker.io/mpioperator/tensorflow-benchmarks:latest
```

```
            command:
              - mpirun
              - --allow-run-as-root
              - -np
              - "2"
              - -bind-to
              - none
              - -map-by
              - slot
              - -x
              - NCCL_DEBUG=INFO
              - -x
              - LD_LIBRARY_PATH
              - -x
              - PATH
              - -mca
              - pml
              - ob1
              - -mca
              - btl
              - ^openib
              - python
              - scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py
              - --model=resnet101
              - --batch_size=64
              - --variable_update=horovod
            resources:
              limits:
                cpu: "2"
                memory: 4Gi
              requests:
                cpu: "2"
                memory: 4Gi
    Worker:
      replicas: 2
      template:
        spec:
          containers:
            - name: tensorflow-benchmarks
              image: mai.daocloud.io/docker.io/mpioperator/tensorflow-benchmarks:latest
              resources:
                limits:
                  cpu: "2"
                  memory: 4Gi
                  nvidia.com/gpumem: 1k
                  nvidia.com/vgpu: "1"
                requests:
                  cpu: "2"
                  memory: 4Gi
```

配置解析：

- `apiVersion` 和 `kind`：表示资源的 API 版本和类型，`MPIJob` 是 Kubeflow 定义的自定义资源，用于创建 MPI 类型的任务。
- `metadata`：元数据，包含任务的名称等信息。
- `spec`：任务的详细配置。
  - `slotsPerWorker`：每个 Worker 节点的槽位数量，通常设置为 1。
  - `runPolicy`：运行策略，例如任务完成后是否清理 Pod。
  - `mpiReplicaSpecs`：MPI 任务的副本配置。

- `Launcher`：启动器，负责启动 MPI 任务，副本数通常为 1；`template` 定义容器运行的镜像、命令、资源等。
- `Worker`：工作节点，实际执行任务的计算节点；`replicas` 根据并行需求设置，这里设置为 2。

设置任务副本数：创建 MPI 任务时，需要根据 `mpiReplicaSpecs` 中配置的副本数，正确设置 **任务副本数**。总副本数 = `Launcher` 副本数 + `Worker` 副本数。本示例中总副本数为 1 + 2 = 3，因此需要将 **任务副本数** 设置为 **3**。

提交任务：配置完成后，点击 **提交** 按钮，开始运行 MPI 任务。

查看运行结果

任务提交成功后，您可以进入 **任务详情** 页面，查看资源的使用情况和任务的运行状态。从右上角进入 **工作负载详情**，可以查看运行过程中每个节点的日志输出。

示例输出：

```
TensorFlow:  1.13
Model:       resnet101
Mode:        training
Batch size:  64
...

Total images/sec: 125.67
```

这表示 MPI 任务成功运行，TensorFlow 基准测试程序完成了分布式训练。

附录：如果您的运行环境未预装所需的库（如 `mpi4py`、Horovod 等），请在任务中添加安装命令，或者使用预装了相关依赖的镜像。在实际应用中，您可以根据需求修改 MPIJob 的配置，例如更改镜像、命令参数、资源请求等。

（六）MXNet 任务

> 由于 Apache MXNet 项目已存档，因此 Kubeflow MXJob 将在未来的 Training Operator 1.9 版本中弃用和删除。

Apache MXNet 是一个高性能的深度学习框架，支持多种编程语言。MXNet 任务可以使用多种方式进行训练，包括单机模式和分布式模式。在 AI Lab 中，我们提供了对 MXNet 任务的支持，您可以通过界面化操作，快速创建 MXNet 任务，进行模型训练。

任务配置介绍

- **任务类型**：`MXNet`，支持单机和分布式两种模式。
- **运行环境**：选择包含 MXNet 框架的镜像，或在任务中安装必要的依赖。

我们使用 `release-ci.daocloud.io/baize/kubeflow/mxnet-gpu:latest` 镜像作为任务的基础运行环境。该镜像预装了 MXNet 及其相关依赖，支持 GPU 加速。

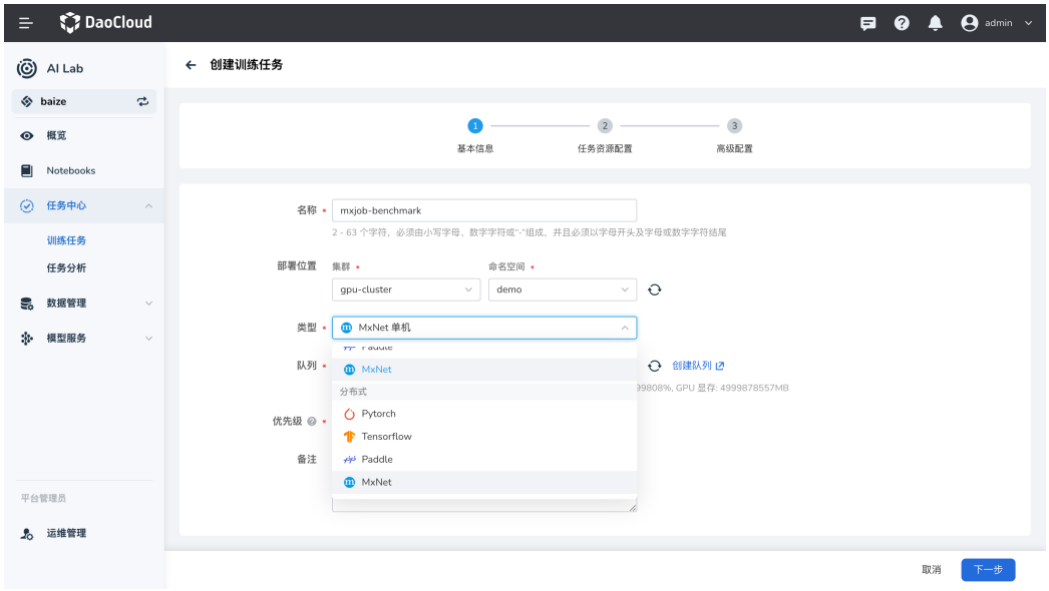

图 63　创建 MXNet 任务

MXNet 单机任务

创建步骤：登录平台后进入 **任务中心** -> **训练任务**，点击右上角的 **创建**，选择任务类型为 `MXNet`，点击 **下一步**，填写任务名称和描述（例如「MXNet 单机训练任务」），点击 **确定**，然后配置任务的运行参数、镜像、资源等信息。

运行参数：启动命令为 `python3`，命令参数如下：

```
/mxnet/mxnet/example/gluon/mnist/mnist.py --epochs 10 --cuda
```

- `/mxnet/mxnet/example/gluon/mnist/mnist.py`：MXNet 提供的 MNIST 手写数字识别示例脚本。
- `--epochs 10`：设置训练轮数为 10。
- `--cuda`：使用 CUDA 进行 GPU 加速。

资源配置：副本数 1（单机任务）；资源请求为 CPU 2 核、内存 4 GiB、GPU 1 块。

完整的单机 MXJob 配置示例如下：

```
apiVersion: "kubeflow.org/v1"
kind: "MXJob"
metadata:
  name: "mxnet-single-job"
spec:
  jobMode: MXTrain
  mxReplicaSpecs:
    Worker:
      replicas: 1
      restartPolicy: Never
      template:
        spec:
          containers:
            - name: mxnet
              image: release-ci.daocloud.io/baize/kubeflow/mxnet-gpu:latest
```

```
            command: ["python3"]
            args:
              [
                "/mxnet/mxnet/example/gluon/mnist/mnist.py",
                "--epochs",
                "10",
                "--cuda",
              ]
            ports:
              - containerPort: 9991
                name: mxjob-port
            resources:
              limits:
                cpu: "2"
                memory: 4Gi
                nvidia.com/gpu: 1
              requests:
                cpu: "2"
                memory: 4Gi
                nvidia.com/gpu: 1
```

配置解析：

- `apiVersion` 和 `kind`：指定资源的 API 版本和类型，这里是 `MXJob`。
- `metadata`：元数据，包括任务名称等信息。
- `spec`：任务的详细配置。
  - `jobMode`：设置为 `MXTrain`，表示训练任务。
  - `mxReplicaSpecs`：MXNet 任务的副本配置。
    - `Worker`：指定工作节点的配置。
    - `replicas`：副本数，这里为 1。
    - `restartPolicy`：重启策略，设为 `Never`，表示任务失败时不重启。
    - `template`：Pod 模板，定义容器的运行环境和资源，包括 `image`、`command`、`args`、`ports`、`resources`。

提交任务：配置完成后，点击 **提交** 按钮，开始运行 MXNet 单机任务。

示例输出：

```
Epoch 1: accuracy=0.95
Epoch 2: accuracy=0.97
...
Epoch 10: accuracy=0.98
Training completed.
```

这表示 MXNet 单机任务成功运行，模型训练完成。

MXNet 分布式任务

运行参数：启动命令为 `python3`，命令参数如下：

```
/mxnet/mxnet/example/image-classification/train_mnist.py --num-epochs 10 --num-layers 2
--kv-store dist_device_sync --gpus 0
```

- `train_mnist.py`：MXNet 提供的图像分类示例脚本。

- `--num-epochs 10`：训练轮数为 10。
- `--num-layers 2`：模型的层数为 2。
- `--kv-store dist_device_sync`：使用分布式设备同步模式。
- `--gpus 0`：使用 GPU 进行加速。

资源配置：任务副本数为 3（包括 Scheduler、Server 和 Worker），各角色副本数均为 1，资源请求均为 CPU 2 核、内存 4 GiB、GPU 1 块。

完整的分布式 MXJob 配置示例如下：

```
apiVersion: "kubeflow.org/v1"
kind: "MXJob"
metadata:
  name: "mxnet-job"
spec:
  jobMode: MXTrain
  mxReplicaSpecs:
    Scheduler:
      replicas: 1
      restartPolicy: Never
      template:
        spec:
          containers:
            - name: mxnet
              image: release-ci.daocloud.io/baize/kubeflow/mxnet-gpu:latest
              ports:
                - containerPort: 9991
                  name: mxjob-port
              resources:
                limits:
                  cpu: "2"
                  memory: 4Gi
                  nvidia.com/gpu: 1
                requests:
                  cpu: "2"
                  memory: 4Gi
    Server:
      replicas: 1
      restartPolicy: Never
      template:
        spec:
          containers:
            - name: mxnet
              image: release-ci.daocloud.io/baize/kubeflow/mxnet-gpu:latest
              ports:
                - containerPort: 9991
                  name: mxjob-port
              resources:
                limits:
                  cpu: "2"
                  memory: 4Gi
                  nvidia.com/gpu: 1
                requests:
                  cpu: "2"
                  memory: 4Gi
    Worker:
      replicas: 1
      restartPolicy: Never
      template:
        spec:
          containers:
            - name: mxnet
              image: release-ci.daocloud.io/baize/kubeflow/mxnet-gpu:latest
```

```
            command: ["python3"]
            args:
              [
                "/mxnet/mxnet/example/image-classification/train_mnist.py",
                "--num-epochs",
                "10",
                "--num-layers",
                "2",
                "--kv-store",
                "dist_device_sync",
                "--gpus",
                "0",
              ]
            ports:
              - containerPort: 9991
                name: mxjob-port
            resources:
              limits:
                cpu: "2"
                memory: 4Gi
                nvidia.com/gpu: 1
              requests:
                cpu: "2"
                memory: 4Gi
```

配置解析：

- **Scheduler（调度器）**：负责协调集群中各节点的任务调度。
- **Server（参数服务器）**：用于存储和更新模型参数，实现分布式参数同步。
- **Worker（工作节点）**：实际执行训练任务。
- **资源配置**：为各角色分配适当的资源，确保任务顺利运行。

设置任务副本数：总副本数 = Scheduler 副本数 + Server 副本数 + Worker 副本数。本示例中为 1 + 1 + 1 = 3，因此需要将 **任务副本数** 设置为 **3**。

示例输出：

```
INFO:root:Epoch[0] Batch [50]     Speed: 1000 samples/sec   accuracy=0.85
INFO:root:Epoch[0] Batch [100]    Speed: 1200 samples/sec   accuracy=0.87
...
INFO:root:Epoch[9] Batch [100]    Speed: 1300 samples/sec   accuracy=0.98
Training completed.
```

这表示 MXNet 分布式任务成功运行，模型训练完成。

附录 • 注意事项：确保您使用的镜像包含所需的 MXNet 版本和依赖；根据实际需求调整资源配置，避免资源不足或浪费；如需使用自定义的训练脚本，请修改启动命令和参数。

（七）查看任务工作负载

任务创建好后，都会显示在训练任务列表中。

1. 在训练任务列表中，点击某个任务右侧的 ⋮ -> **任务负载详情**。

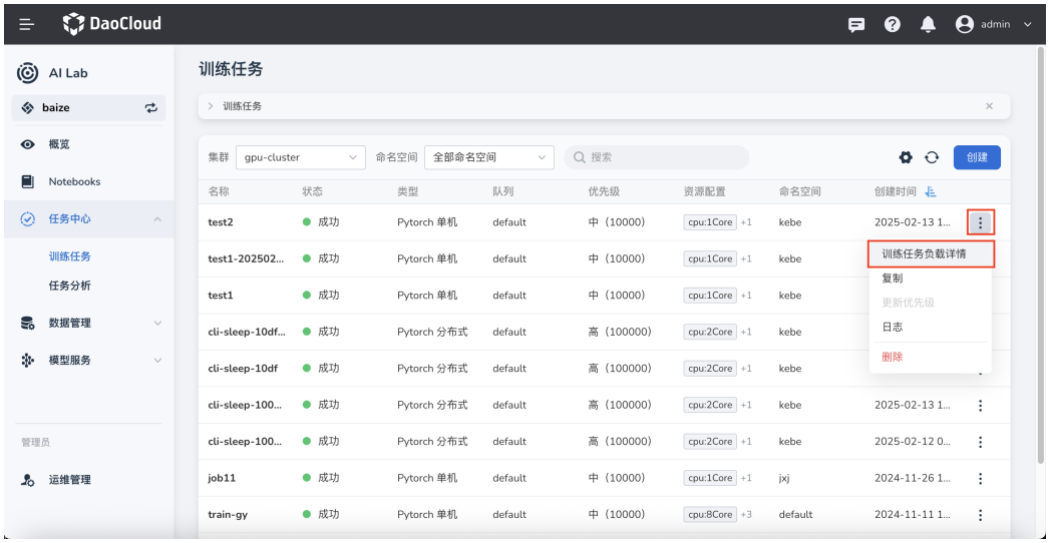

图 64　选择任务负载详情

2. 出现一个弹窗选择要查看哪个 Pod 后，点击 **进入**。

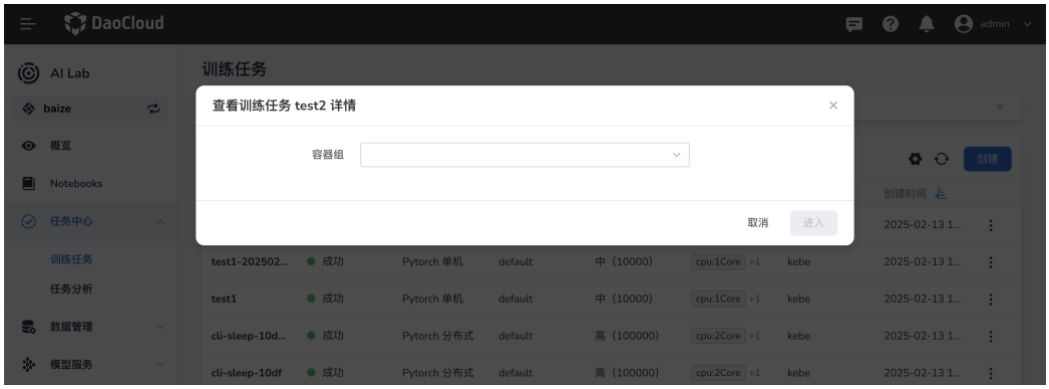

图 65　选择要查看的 Pod

3. 跳转到容器管理界面，可以查看容器的工作状态、标签与注解以及发生的事件。

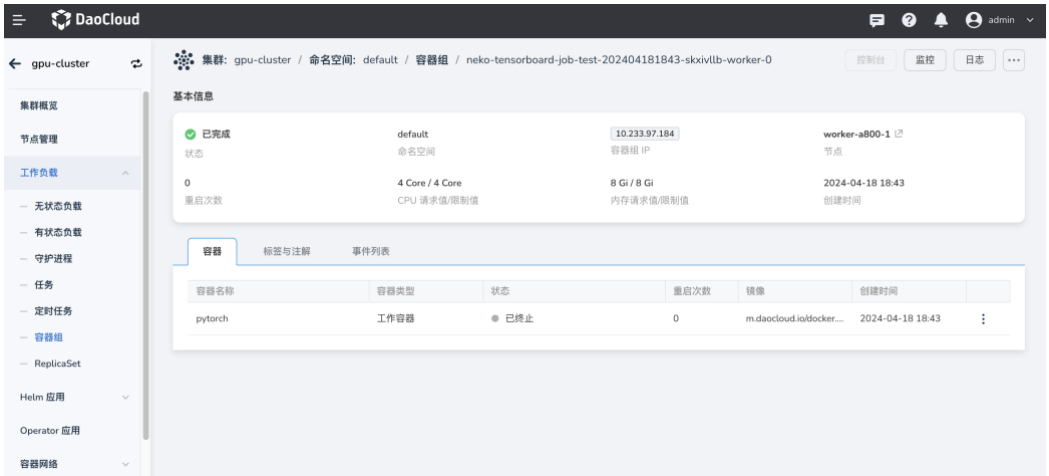

图 66　查看容器详情

4. 你还可以查看当前 Pod 最近一段时间的详细日志。此处默认展示 100 行日志，如果要查看更详细的日志或下载日志，请点击顶部的蓝色 **可观测性** 文字。

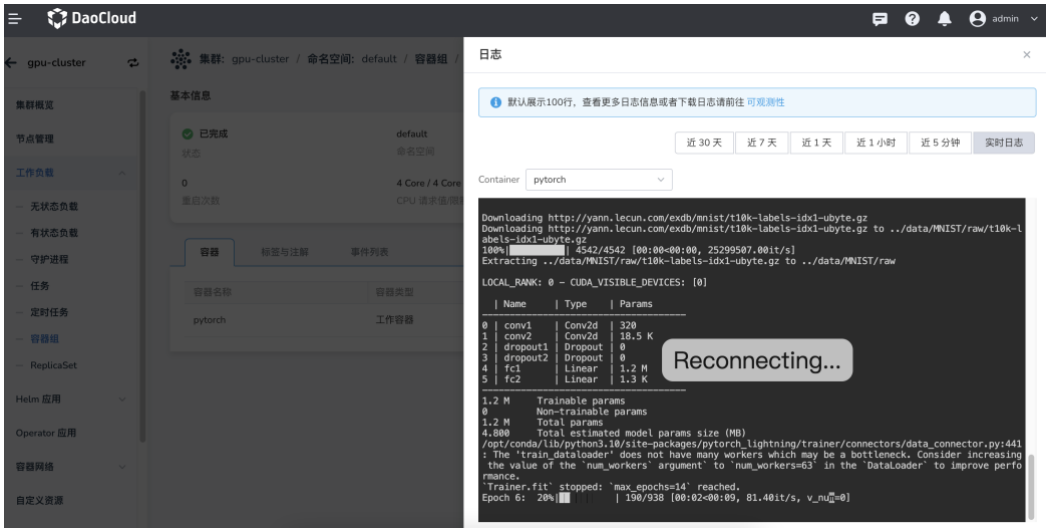

图 67　查看 Pod 日志

5. 当然你还可以通过右上角的 ...，查看当前 Pod 的 YAML、上传和下载文件。以下是一个 Pod 的 YAML 示例。

```
kind: Pod
apiVersion: v1
metadata:
  name: neko-tensorboard-job-test-202404181843-skxivllb-worker-0
  namespace: default
  uid: ddedb6ff-c278-47eb-ae1e-0de9b7c62f8c
  resourceVersion: '41092552'
  creationTimestamp: '2024-04-18T10:43:36Z'
  labels:
    training.kubeflow.org/job-name: neko-tensorboard-job-test-202404181843-skxivllb
    training.kubeflow.org/operator-name: pytorchjob-controller
    training.kubeflow.org/replica-index: '0'
    training.kubeflow.org/replica-type: worker
  annotations:
    cni.projectcalico.org/containerID:
0cfbb9af257d5e69027c603c6cb2d3890a17c4ae1a145748d5aef73a10d7fbe1
    hami.io/bind-phase: success
    hami.io/bind-time: '1713437016'
    hami.io/vgpu-devices-allocated: GPU-29d5fa0d-935b-2966-aff8-
483a174d61d1,NVIDIA,1024,20:;
    hami.io/vgpu-node: worker-a800-1
  ownerReferences:
    - apiVersion: kubeflow.org/v1
      kind: PyTorchJob
      name: neko-tensorboard-job-test-202404181843-skxivllb
      uid: e5a8b05d-1f03-4717-8e1c-4ec928014b7b
      controller: true
      blockOwnerDeletion: true
spec:
  volumes:
    - name: 0-dataset-pytorch-examples
      persistentVolumeClaim:
        claimName: pytorch-examples
  containers:
    - name: pytorch
      image: m.daocloud.io/docker.io/pytorch/pytorch
```

```
      command:
        - bash
      args:
        - '-c'
        - >-
          ls -la /root && which pip && pip install pytorch_lightning tensorboard
          && python /root/Git/pytorch/examples/mnist/main.py
      ports:
        - name: pytorchjob-port
          containerPort: 23456
          protocol: TCP
      env:
        - name: PYTHONUNBUFFERED
          value: '1'
        - name: PET_NNODES
          value: '1'
      resources:
        limits:
          cpu: '4'
          memory: 8Gi
          nvidia.com/gpucores: '20'
          nvidia.com/gpumem: '1024'
          nvidia.com/vgpu: '1'
        requests:
          cpu: '4'
          memory: 8Gi
          nvidia.com/gpucores: '20'
          nvidia.com/gpumem: '1024'
          nvidia.com/vgpu: '1'
      volumeMounts:
        - name: 0-dataset-pytorch-examples
          mountPath: /root/Git/pytorch/examples
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
      imagePullPolicy: Always
  restartPolicy: Never
  terminationGracePeriodSeconds: 30
  dnsPolicy: ClusterFirst
  serviceAccountName: default
  nodeName: worker-a800-1
  securityContext: {}
  affinity: {}
  schedulerName: hami-scheduler
  priorityClassName: baize-high-priority
  priority: 100000
  enableServiceLinks: true
  preemptionPolicy: PreemptLowerPriority
status:
  phase: Succeeded
  hostIP: 10.20.100.211
  podIP: 10.233.97.184
  startTime: '2024-04-18T10:43:36Z'
  qosClass: Guaranteed
```

（八）任务分析（TensorBoard）

在 DCE AI Lab 模块中，提供了模型开发过程重要的可视化分析工具，用于展示机器学习模型的训练过程和结果。本节将介绍任务分析（Tensorboard）的基本概念、在 AI Lab 系统中的使用方法，以及如何配置数据集的日志内容。

> Tensorboard 是 TensorFlow 提供的一个可视化工具，用于展示机器学习模型的训练过程和结果。它可以帮助开发者更直观地理解模型的训练动态，分析模型性能，调试模型问题等。

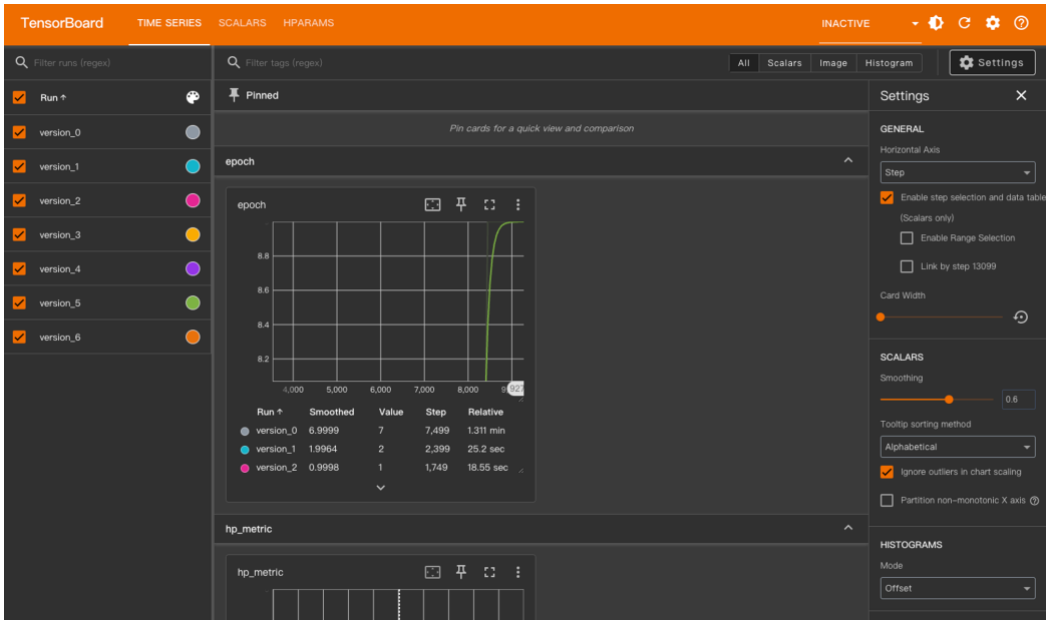

图 68 TensorBoard 任务分析

Tensorboard 在模型开发过程中的作用及优势：

- **可视化训练过程**：通过图表展示训练和验证的损失、精度等指标，帮助开发者直观地观察模型的训练效果。
- **调试和优化模型**：通过查看不同层的权重、梯度分布等，帮助开发者发现和修正模型中的问题。
- **对比不同实验**：可以同时展示多个实验的结果，方便开发者对比不同模型和超参数配置的效果。
- **追踪训练数据**：记录训练过程中使用的数据集和参数，确保实验的可复现性。

如何创建 Tensorboard

在创建时 Notebook 启用 Tensorboard

1. 创建 Notebook：在 AI Lab 平台上创建一个新的 Notebook。

2. 启用 Tensorboard：在创建 Notebook 的页面中，启用 **Tensorboard** 选项，并指定数据集和日志路径。

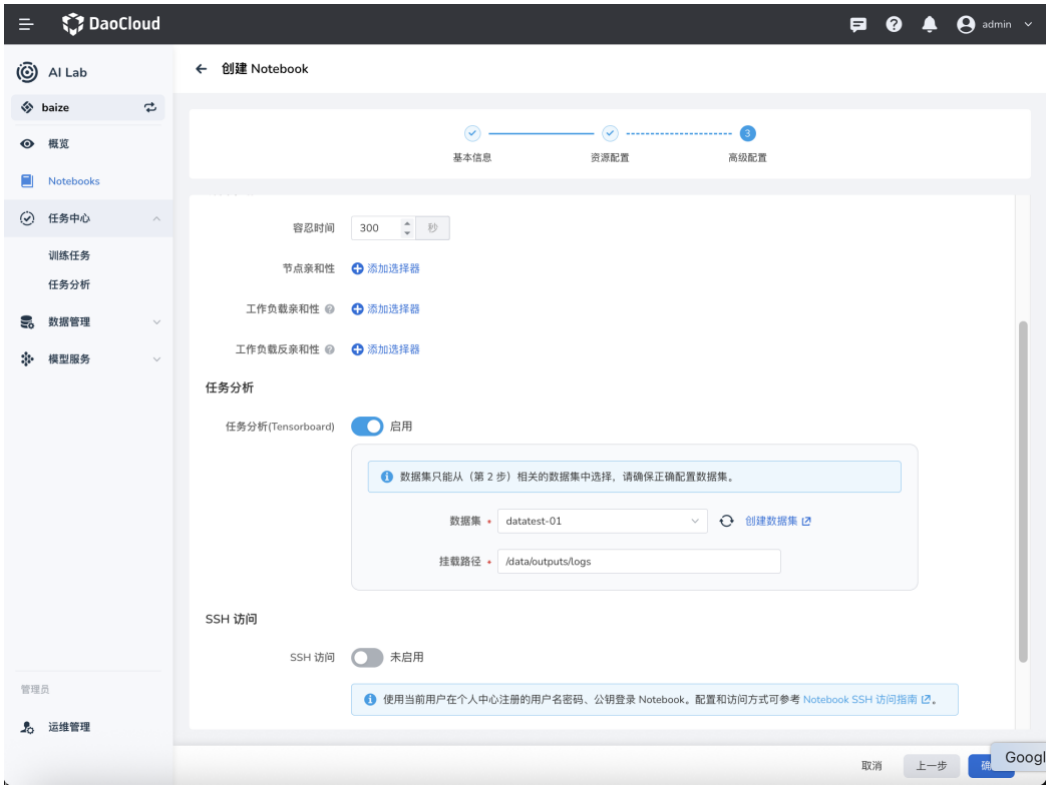

图 69　在 Notebook 中启用 Tensorboard

在分布式任务创建及完成后启用 Tensorboard

1. 创建分布式任务：在 AI Lab 平台上创建一个新的分布式训练任务。

2. 配置 Tensorboard：在任务配置页面中，启用 **Tensorboard** 选项，并指定数据集和日志路径。

3. 任务完成后查看 Tensorboard：任务完成后，可以在任务详情页面中查看 Tensorboard 的链接，点击链接即可查看训练过程的可视化结果。

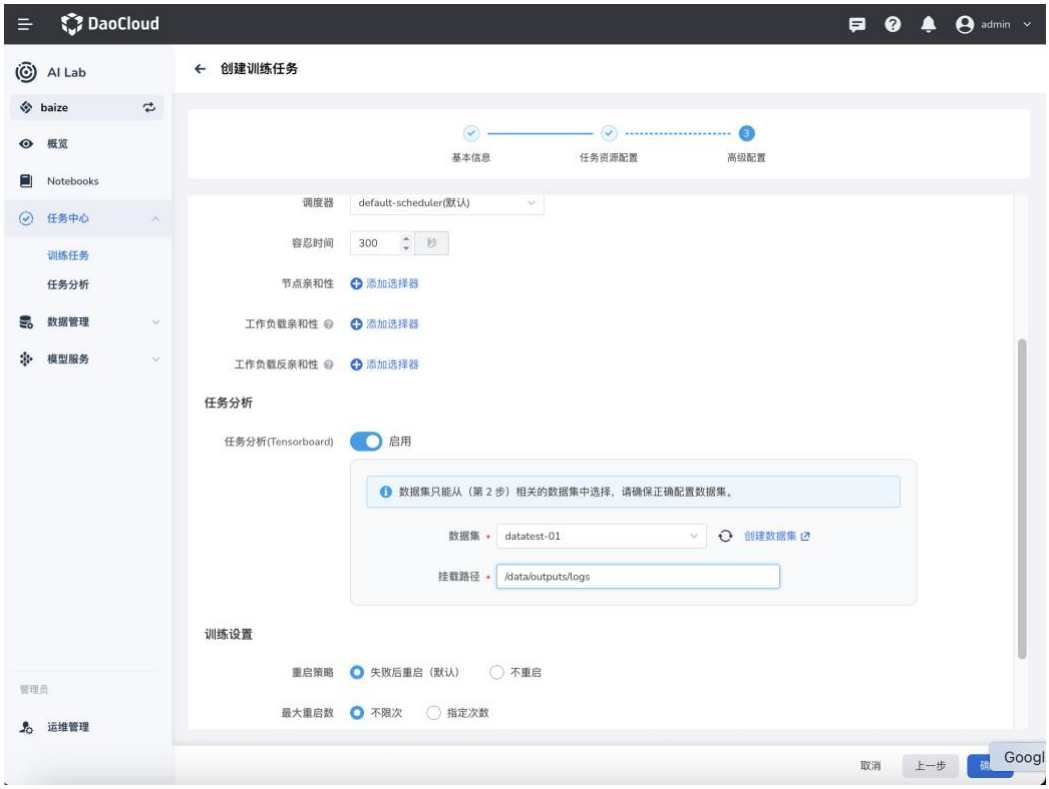

图 70　在训练任务中启用 Tensorboard

在 Notebook 中直接引用 Tensorboard

在 Notebook 中，可以通过代码直接启动 Tensorboard。以下是一个示例代码：

```
# 导入必要的库
import tensorflow as tf
import datetime

# 定义日志目录
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# 创建 Tensorboard 回调
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

# 构建并编译模型
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 训练模型并启用 Tensorboard 回调
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test),
callbacks=[tensorboard_callback])
```

如何配置数据集的日志内容

在使用 Tensorboard 时，可以记录和配置不同的数据集和日志内容。以下是一些常见的配置方式。

配置训练和验证数据集的日志

在训练模型时，可以通过 TensorFlow 的 `tf.summary` API 来记录训练和验证数据集的日志。以下是一个示例代码：

```
# 导入必要的库
import tensorflow as tf

# 创建日志目录
train_log_dir = 'logs/gradient_tape/train'
val_log_dir = 'logs/gradient_tape/val'
train_summary_writer = tf.summary.create_file_writer(train_log_dir)
val_summary_writer = tf.summary.create_file_writer(val_log_dir)

# 训练模型并记录日志
for epoch in range(EPOCHS):
    for (x_train, y_train) in train_dataset:
        # 训练步骤
        train_step(x_train, y_train)
        with train_summary_writer.as_default():
            tf.summary.scalar('loss', train_loss.result(), step=epoch)
            tf.summary.scalar('accuracy', train_accuracy.result(), step=epoch)

    for (x_val, y_val) in val_dataset:
        # 验证步骤
        val_step(x_val, y_val)
        with val_summary_writer.as_default():
            tf.summary.scalar('loss', val_loss.result(), step=epoch)
            tf.summary.scalar('accuracy', val_accuracy.result(), step=epoch)
```

配置自定义日志

除了训练和验证数据集的日志外，还可以记录其他自定义的日志内容，例如学习率、梯度分布等。以下是一个示例代码：

```
# 记录自定义日志
with train_summary_writer.as_default():
    tf.summary.scalar('learning_rate', learning_rate, step=epoch)
    tf.summary.histogram('gradients', gradients, step=epoch)
```

Tensorboard 管理

在 AI Lab 中，通过各种方式创建出来的 Tensorboard 会统一展示在任务分析的页面中，方便用户查看和管理。

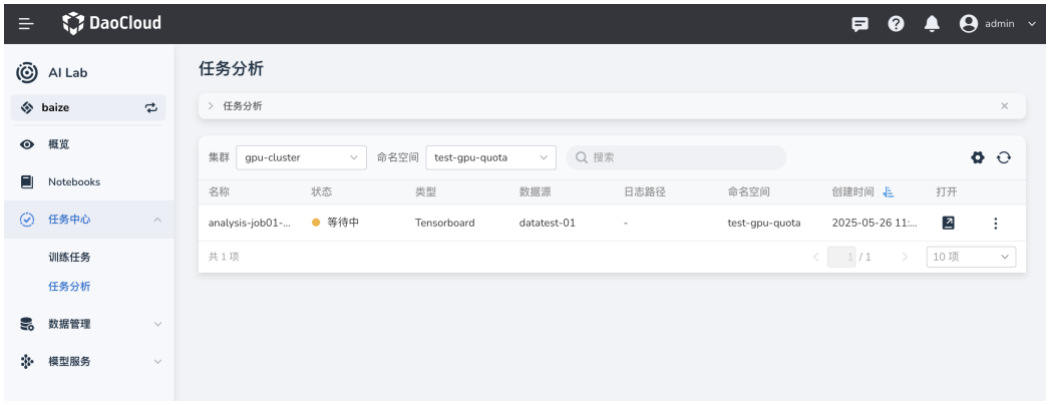

图 71　任务分析列表

用户可以在任务分析页面中查看 Tensorboard 的链接、状态、创建时间等信息，并通过链接直接访问 Tensorboard 的可视化结果。

（九）删除任务

如果发现任务冗余、过期或因其他缘故不再需要，可以从训练任务列表中删除。

1. 在训练任务列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

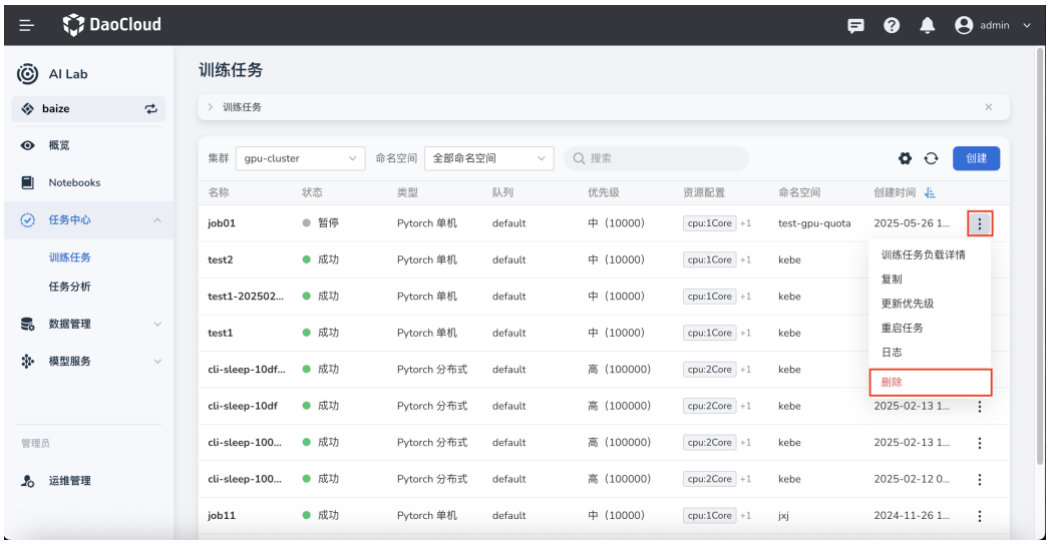

图 72　选择删除任务

2. 在弹窗中确认要删除的任务，输入任务名称后点击 **删除**。

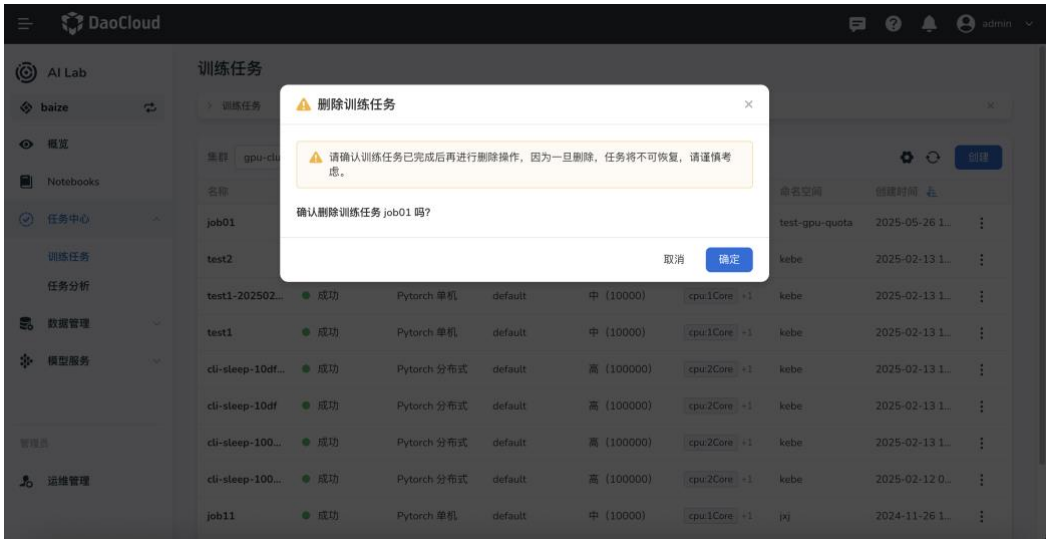

图 73　确认删除任务

3. 屏幕提示删除成功，该任务从列表中消失。

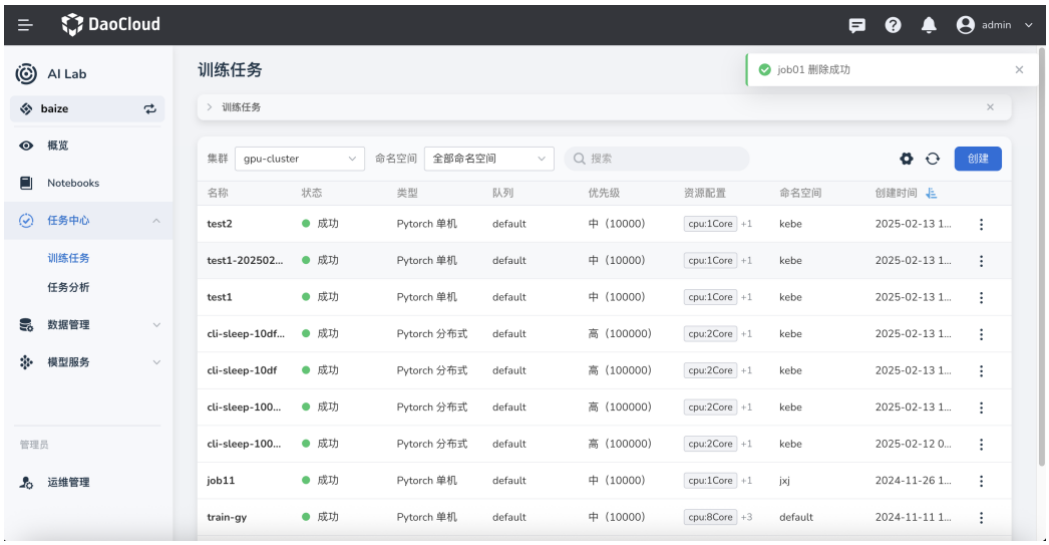

图 74　任务已删除

> 任务一旦删除将不可恢复，请谨慎操作。

五、数据管理

数据管理提供数据集纳管、环境依赖与数据标注三类能力：数据集负责把分散在 Git、S3、HTTP、PVC、NFS 上的数据统一接入并预热到集群；环境依赖把 Python 依赖从镜像中解耦；数据标注则为训练数据提供标注能力。

（一）数据集列表

AI Lab 提供模型开发、训练以及推理过程所有需要的数据集管理功能。目前支持将多种数据源统一接入能力。

通过简单配置即可将数据源接入到 AI Lab 中，实现数据的统一纳管、预热、数据集管理等功能。

创建数据集

1. 在左侧导航栏中点击 **数据管理** -> **数据集**，点击右侧的 **创建** 按钮。

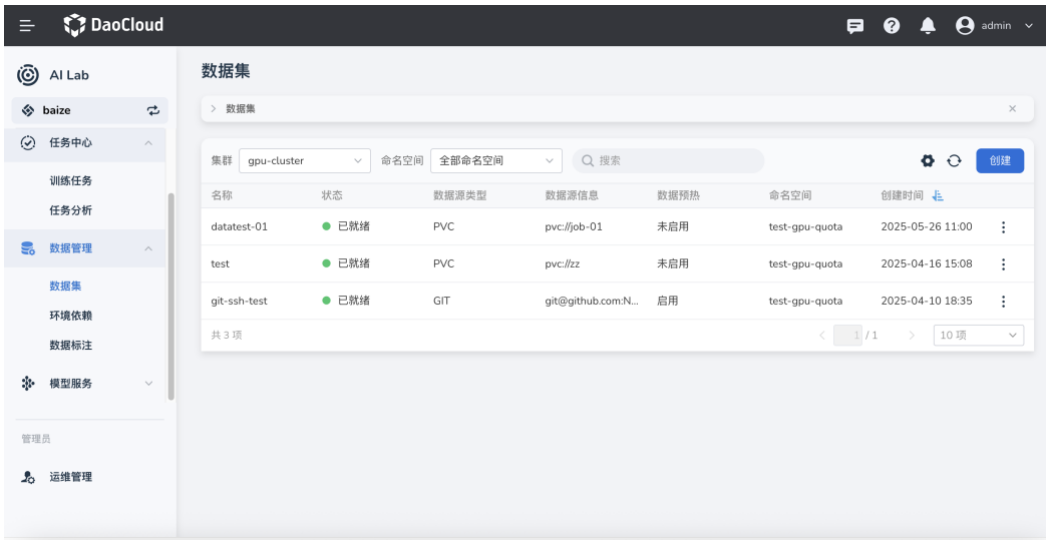

图 75　数据集列表点击创建

2. 选择数据集归属的工作集群、命名空间，点击 **下一步**。

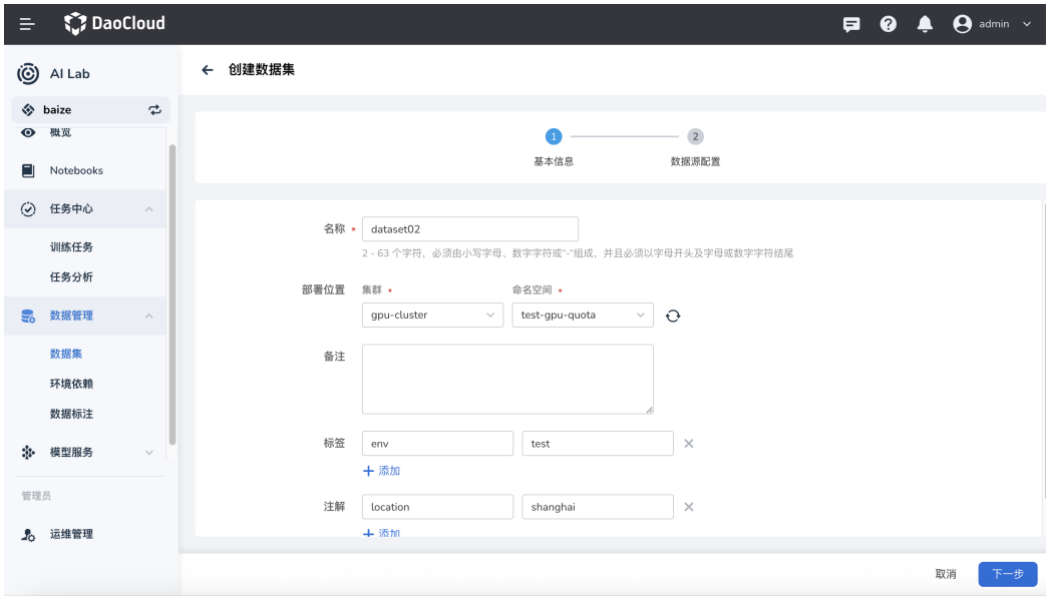

图 76　选择工作集群与命名空间

3. 配置目标数据的数据源类型，然后点击 **确定**。

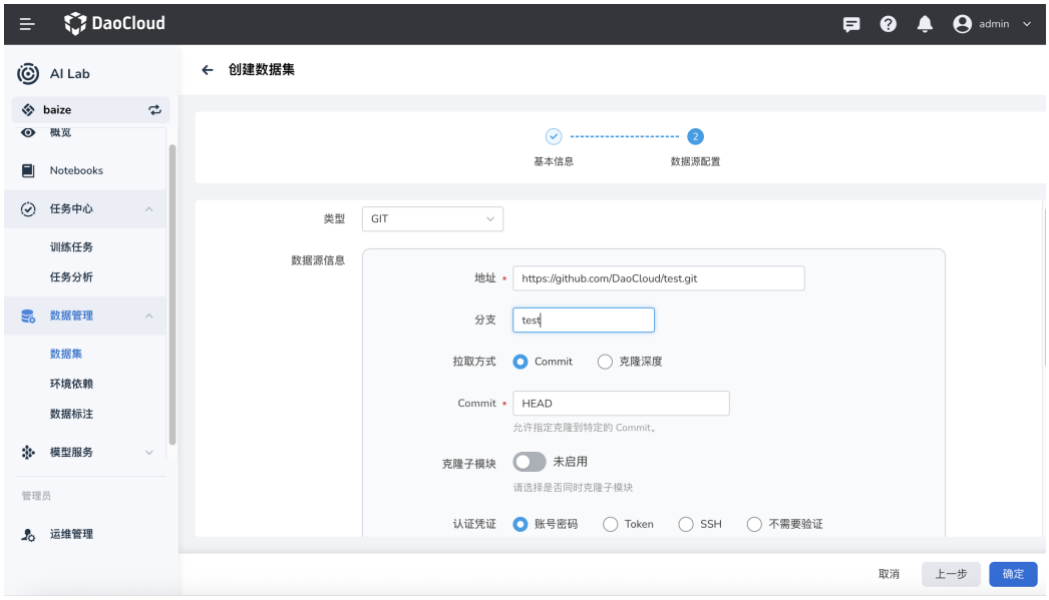

图 77　配置数据源类型

目前支持这几种数据源：

- **GIT**：支持 GitHub、GitLab、Gitee 等仓库
- **S3**：支持 Amazon 云等对象存储
- **HTTP**：直接输入一个有效的 HTTP 网址
- **PVC**：支持预先创建的 Kubernetes PersistentVolumeClaim
- **NFS**：支持 NFS 共享存储

4. 数据集创建成功将返回数据集列表。你可以通过右侧的 ⋮ 执行更多操作。

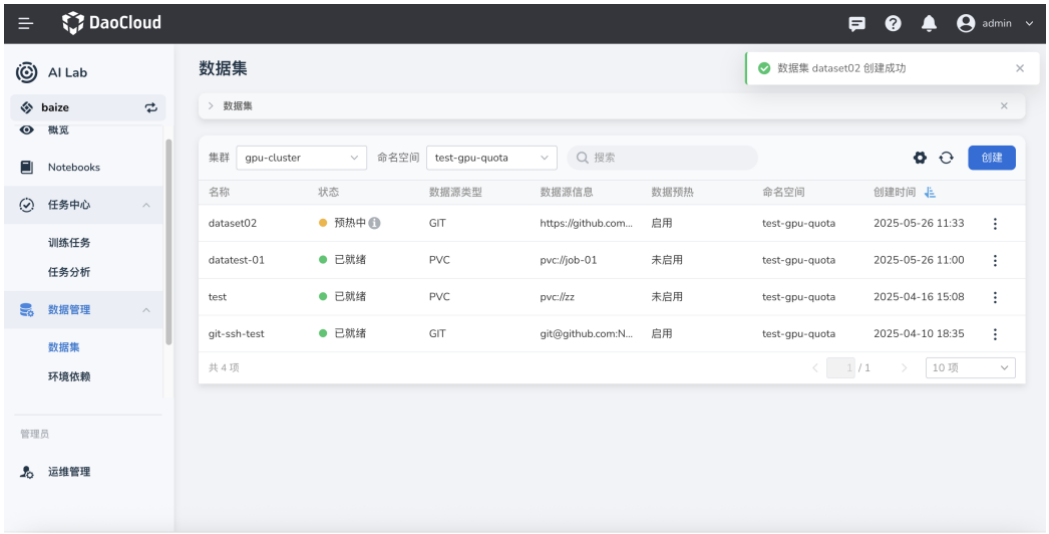

图 78　数据集列表

> 系统自动会在数据集创建成功后，立即进行一次性的数据预加载；在预加载完成之前，数据集不可以使

用。

数据集使用

数据集创建成功后，可以在模型训练、推理等任务中使用。

在 Notebook 中使用

在创建 Notebook 中，可以直接使用数据集；使用方式如下：

- 使用数据集做训练数据挂载
- 使用数据集做代码挂载

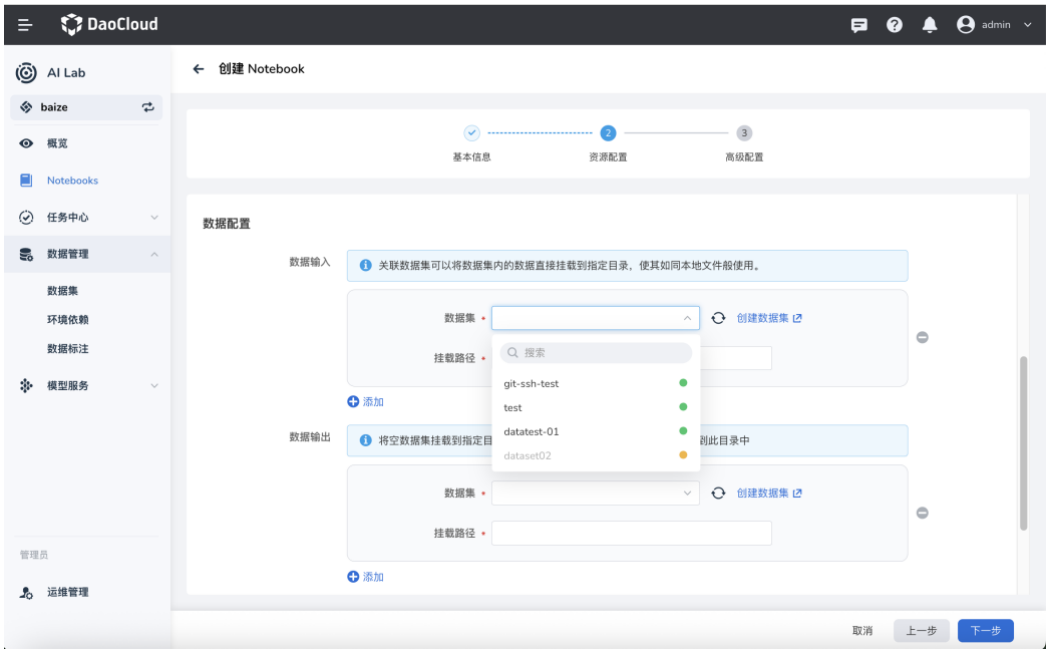

图 79　在 Notebook 中挂载数据集

在训练任务中使用

- 使用数据集指定任务输出
- 使用数据集指定任务输入
- 使用数据集指定 TensorBoard 输出

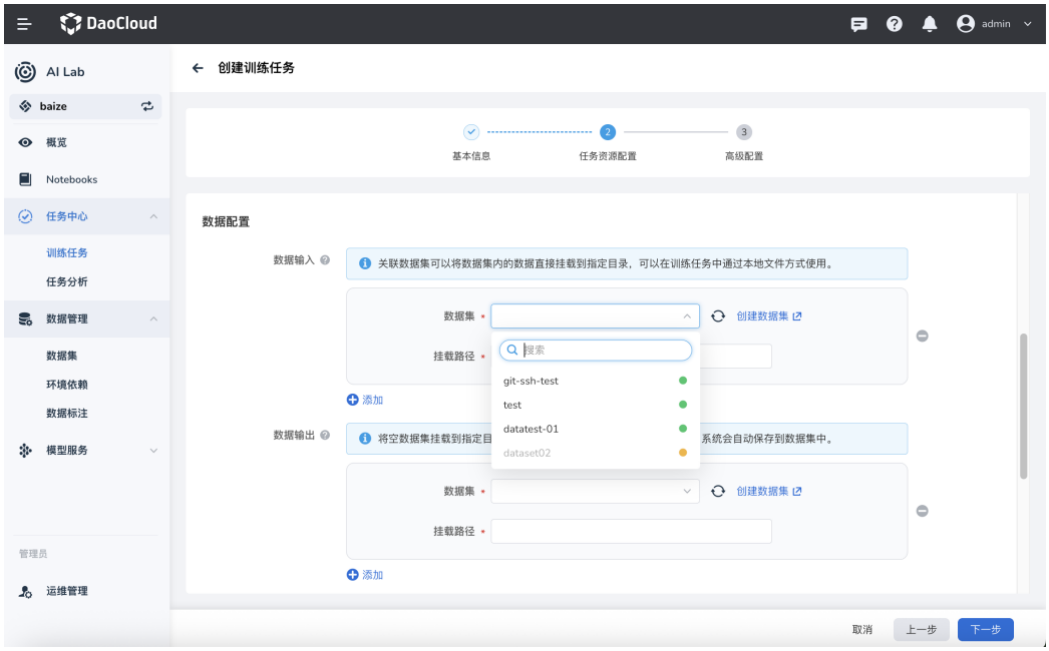

图 80　在训练任务中挂载数据集

在推理服务中使用

- 使用数据集挂载模型

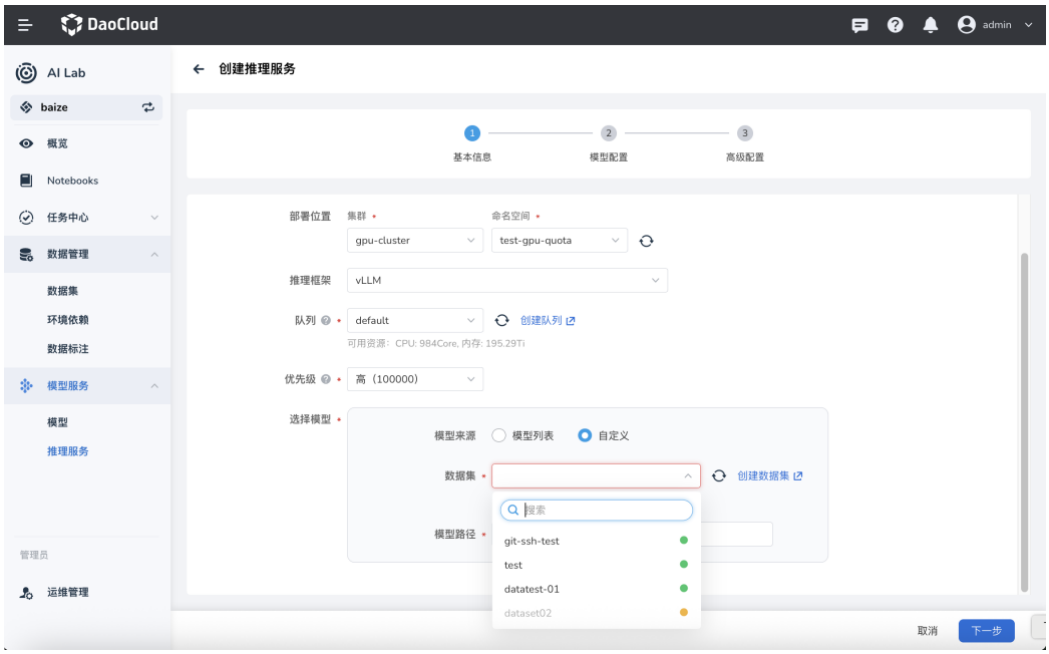

图 81　在推理服务中挂载模型

删除数据集

如果发现数据集冗余、过期或因其他缘故不再需要，可以从数据集列表中删除。

1. 在数据集列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

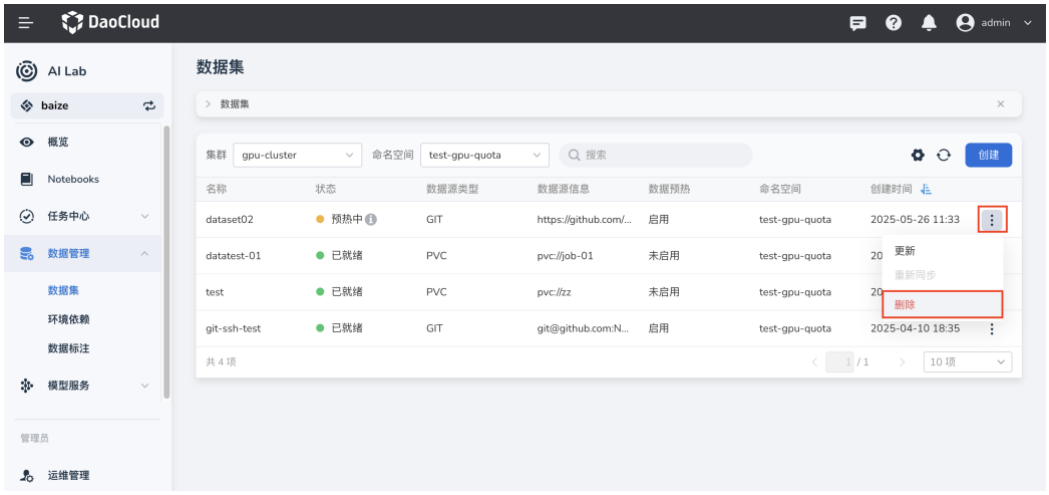

图 82　选择删除数据集

2. 在弹窗中确认要删除的数据集，输入数据集名称后点击 **删除**。

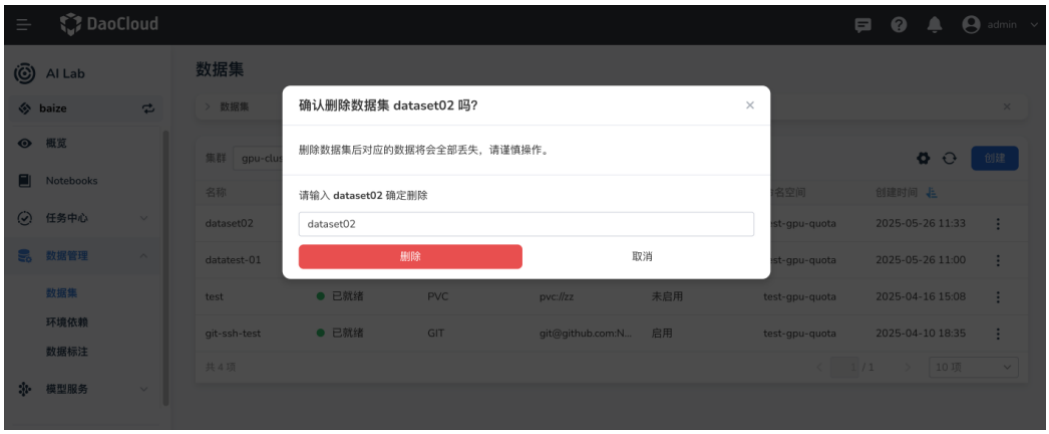

图 83　确认删除数据集

3. 屏幕提示删除成功，该数据集从列表中消失。

> 数据集一旦删除将不可恢复，请谨慎操作。

（二）环境依赖

本节说明如何在 DCE AI Lab 中管理你的环境依赖库，包括环境依赖概述、创建新环境、配置环境与故障排除。

环境依赖概述

传统方式，一般会将 Python 环境依赖在镜像中构建，镜像带有 Python 版本和依赖包的镜像，维护成本较高且更新不方便，往往需要重新构建镜像。

而在 DCE AI Lab 中，用户可以通过 **环境依赖** 模块来管理纯粹的环境依赖，将这部分从镜像中解耦，带来的优势有：

- 一份环境多处使用，同时可以在 Notebook、分布式训练任务、乃至推理服务中使用。
- 更新依赖包更加方便，只需要更新环境依赖即可，无需重新构建镜像。

以下为环境管理的主要组成部分：

- **集群**：选择需要操作的集群。
- **命名空间**：选择命名空间以限定操作范围。
- **环境列表**：展示当前集群和命名空间下的所有环境及其状态。

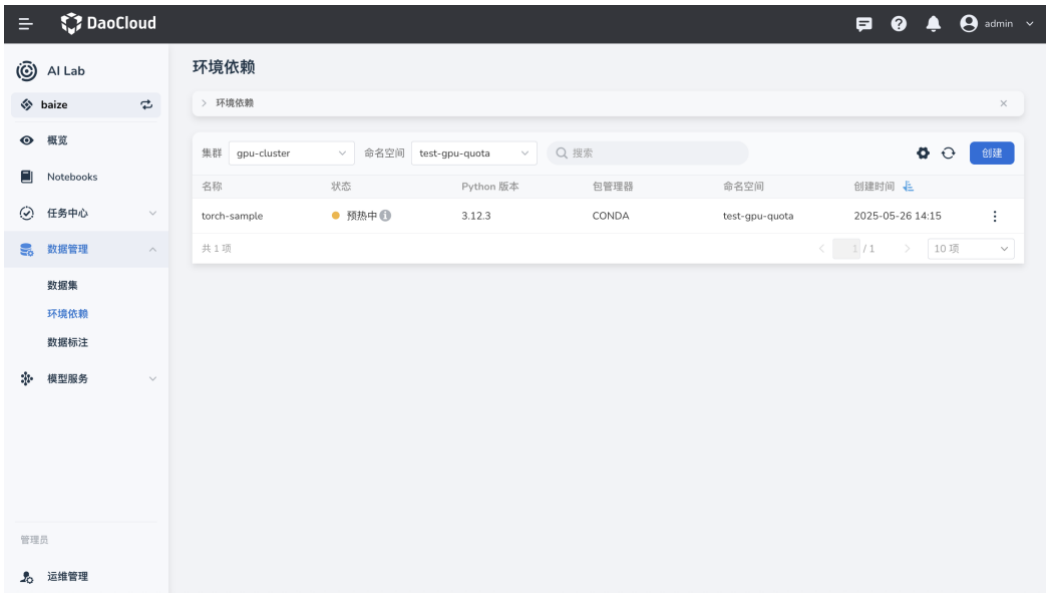

图 84　环境依赖列表

字段	描述	举例值
名称	环境的名称	my-environment
状态	环境当前的状态（正常或失败），新创建环境有一个预热过程，预热成功后即可在其他任务中使用	正常
Python 版本	环境当前的 Python 版本	3.12.3
包管理器	环境当前的包管理工具	CONDA
命名空间	环境当前所处的命名空间	default
创建时间	环境创建的时间	2023-10-01 10:00:00

创建新环境

在 **环境依赖** 界面，点击右上角的 **创建** 按钮，进入创建环境的流程。

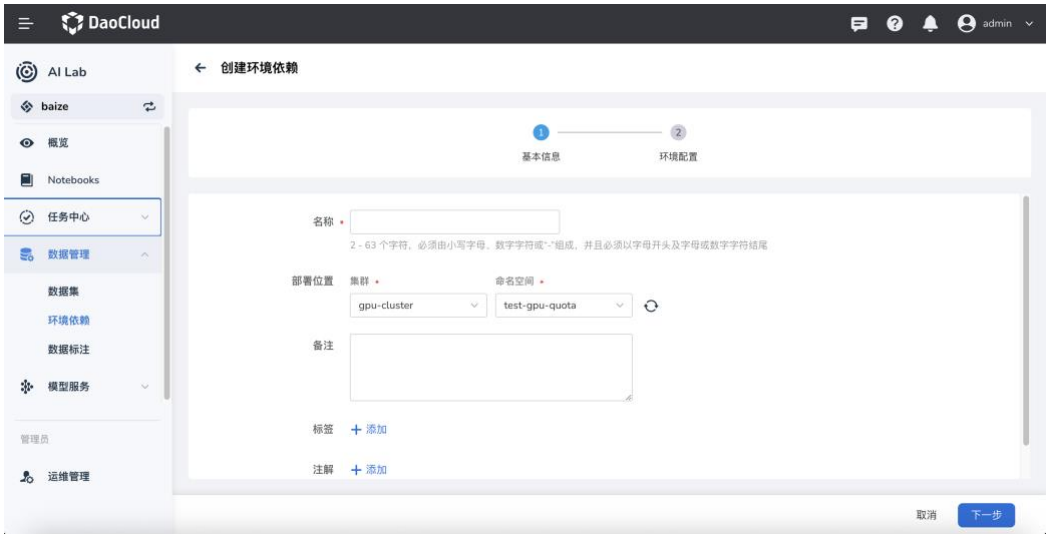

图 85　创建新环境

字段	描述	举例值
名称	输入环境的名称，长度为 2-63 个字符，必须以小写字母、数字开头和结尾。	my-environment
部署位置	**集群**：选择需要部署的集群	gpu-cluster
	命名空间：选择命名空间	default
备注	填写备注信息。	这是一个测试环境
标签	为环境添加标签。	env:test
注解	为环境添加注解。填写完成后，点击 **下一步** 进入环境配置。	注解示例

配置环境

在环境配置步骤中，用户需要配置 Python 版本和依赖包管理工具。

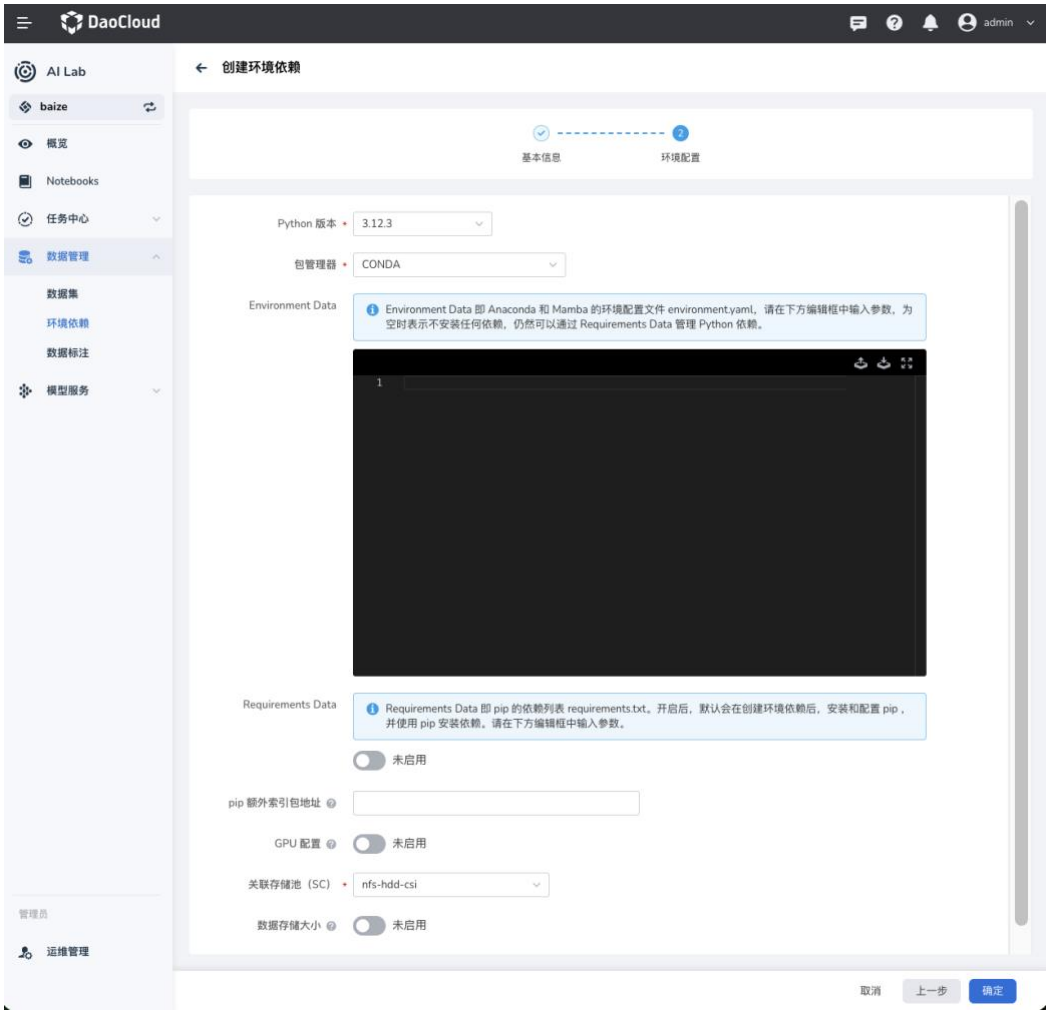

图 86　配置环境

字段	描述	举例值
Python 版本	选择所需的 Python 版本	3.12.3
包管理器	选择包管理工具，可选 PIP 或 CONDA	PIP
Environment Data	如果选择 PIP：在下方编辑器中输入 requirements.txt 格式的依赖包列表。	numpy==1.21.0
	如果选择 CONDA：在下方编辑器中输入 environment.yaml 格式的依赖包列表。	
其他选项	**pip 额外索引地址**：配置 pip 额外的索引地址；适用于企业内部有自己的私有仓库或者 PIP 加速站点。	https://pypi.example.com
	GPU 配置：启用或禁用 GPU 配置；部分涉及到 GPU 的依赖包需要在预加载时配置 GPU 资源。	启用

	关联存储：选择关联的存储配置；环境依赖包会存储在关联存储中。注意：需要使用支持 ReadWriteMany 的存储。	my-storage-config
	数据存储大小：启用或禁用数据存储大小设置；合理评估数据量以避免造成存储资源浪费；如默认不配置，默认不限量（100TB）。	启用
	pip 镜像推荐：在中国大陆环境下，可使用清华 TUNA（https://pypi.tuna.tsinghua.edu.cn/simple）或阿里云（https://mirrors.aliyun.com/pypi/simple）等镜像以提升下载速度。	

配置完成后，点击 **创建** 按钮，系统会自动创建并配置新的 Python 环境。

示例依赖文件

在输入编辑器中可以参考以下模板进行填写。平台会自动写入环境名称和 Python 版本，无需手动指定。

在中国大陆网络环境中，可以优先使用国内的 Conda 镜像源（如清华 TUNA、阿里云等）来提升依赖下载速度。

```
channels:
  - defaults
  - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
dependencies:
  - pytorch=2.2
  - torchvision
  - torchaudio
  - pip
  - pip:
      - numpy==1.24.4
      - matplotlib>=3.8
```

```
torch==2.2.0
torchvision==0.17.0
torchaudio==2.2.0
numpy>=1.24
matplotlib>=3.8
```

故障排除

- 如果环境创建失败：
  - 检查网络连接是否正常。
  - 确认填写的 Python 版本和包管理器配置无误。
  - 确保所选集群和命名空间可用。
- 如果依赖预热失败：
  - 检查 `requirements.txt` 或 `environment.yaml` 文件格式是否正确。

- 确认依赖包名称和版本是否正确无误。如遇到其他问题，请联系平台管理员或查看平台帮助文档获取更多支持。

（三）数据标注

AI Lab 提供数据标注功能，通过为图片、视频、文本、语音等原始数据添加标签或注释，为机器学习模型提供高质量训练数据，从而提升模型的理解能力和准确性。

创建数据标注

1. 在左侧导航栏中点击 **数据管理** -> **数据标注**，点击右侧的 **创建** 按钮。

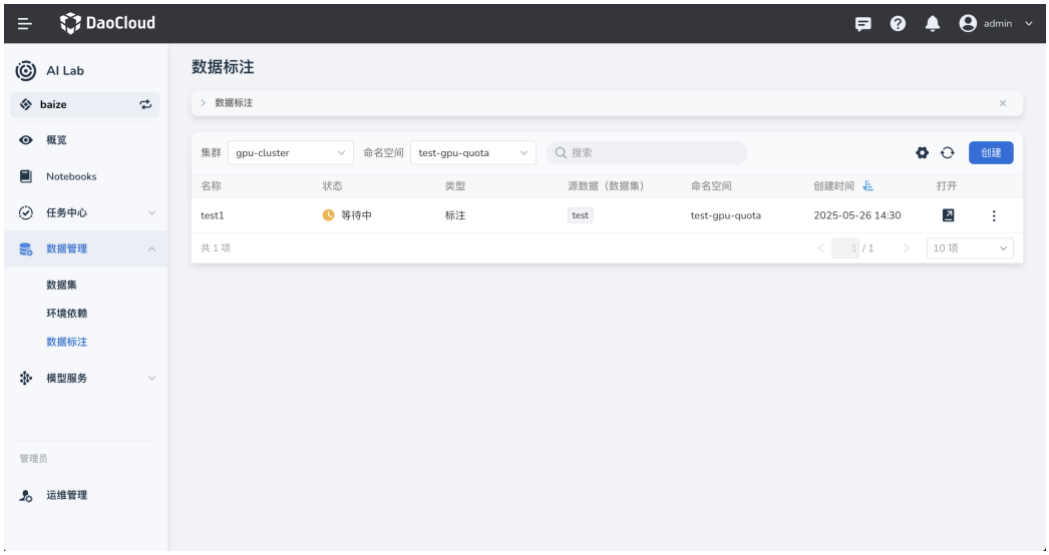

图 87　数据标注列表点击创建

2. 选择数据标注归属的工作集群、命名空间、队列，配置数据标注的源数据、类型和标注结果，然后点击 **确定**。

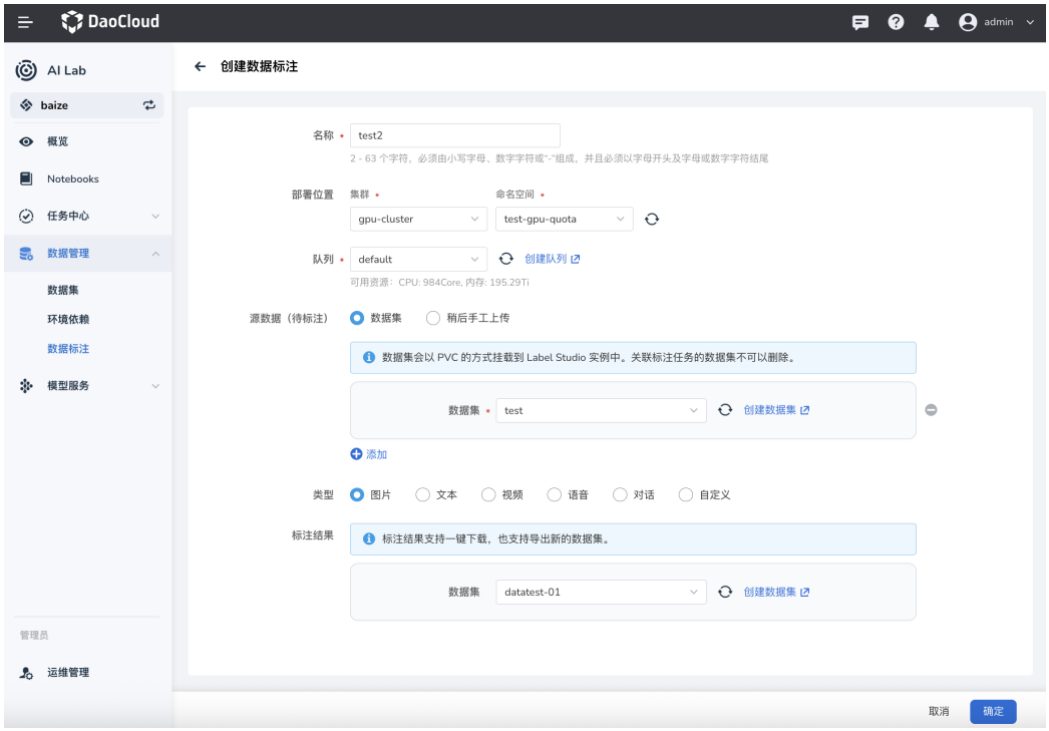

图 88　配置数据标注参数

3. 数据标注创建成功将返回数据标注列表。你可以通过右侧的 ⋮ 执行更多操作。

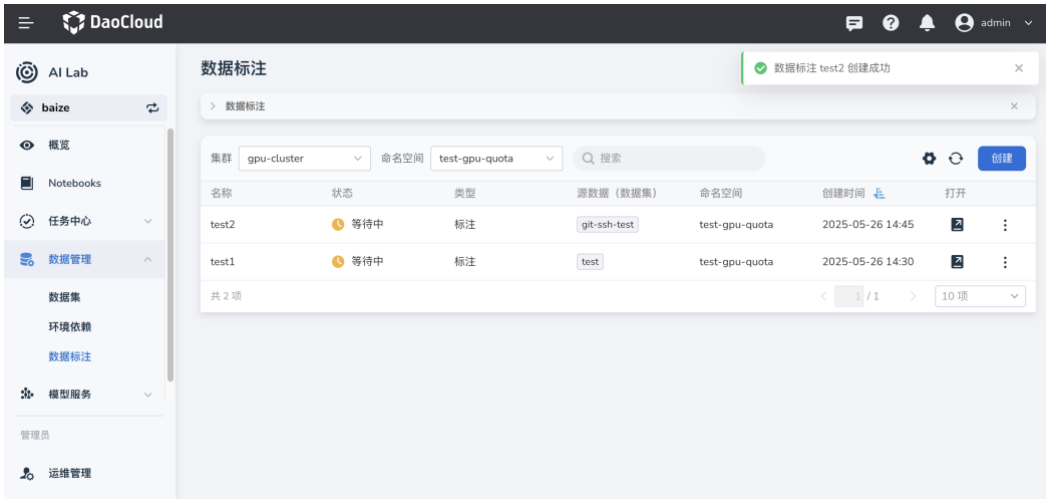

图 89　数据标注列表

下载标注结果

标注结果支持一键下载，在数据标注列表右侧点击 ⋮，在弹出菜单中选择 **下载标注结果**。

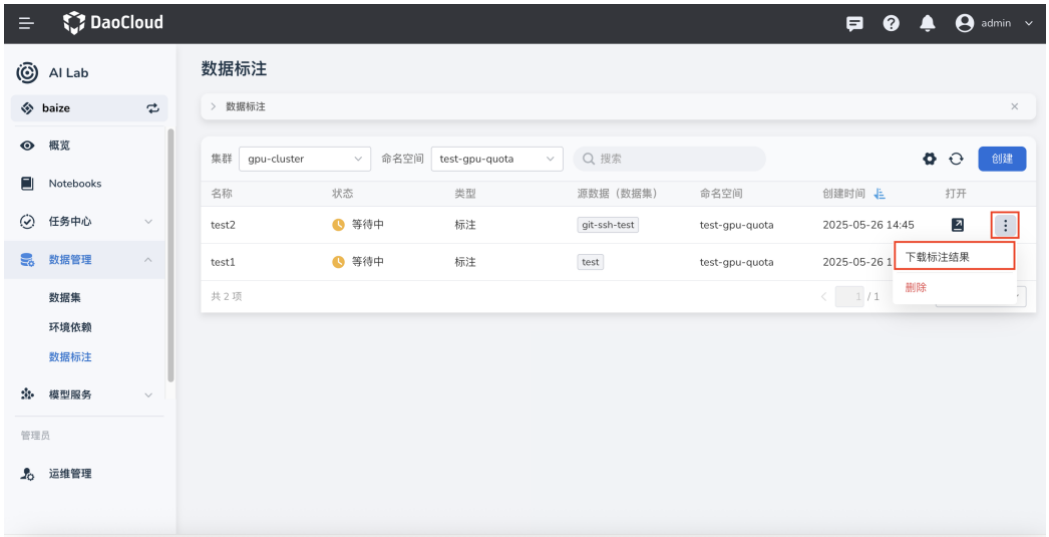

图 90　下载标注结果

删除数据标注

1. 在数据标注列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

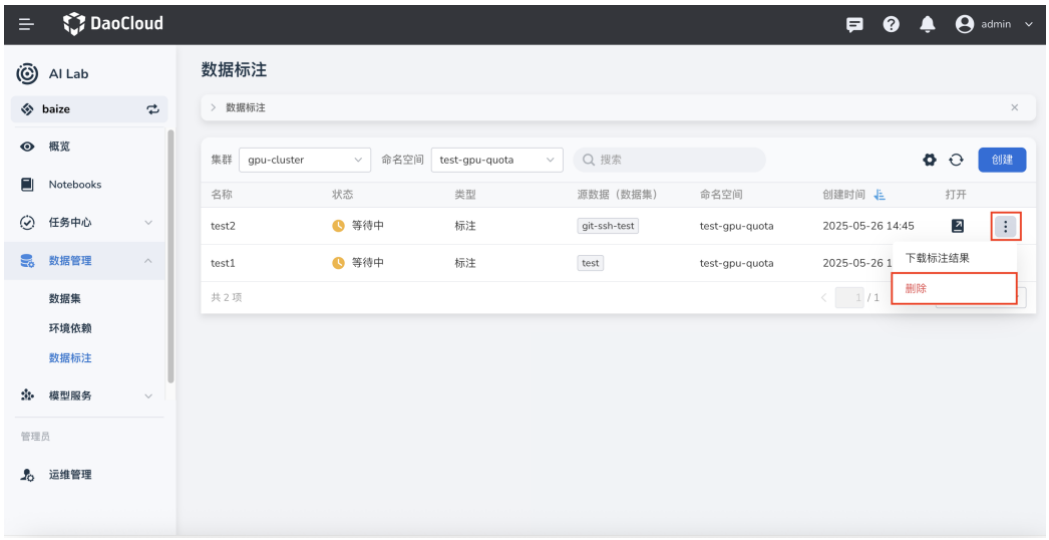

图 91　选择删除数据标注

2. 在弹窗中确认要删除的数据标注后点击 **删除**。

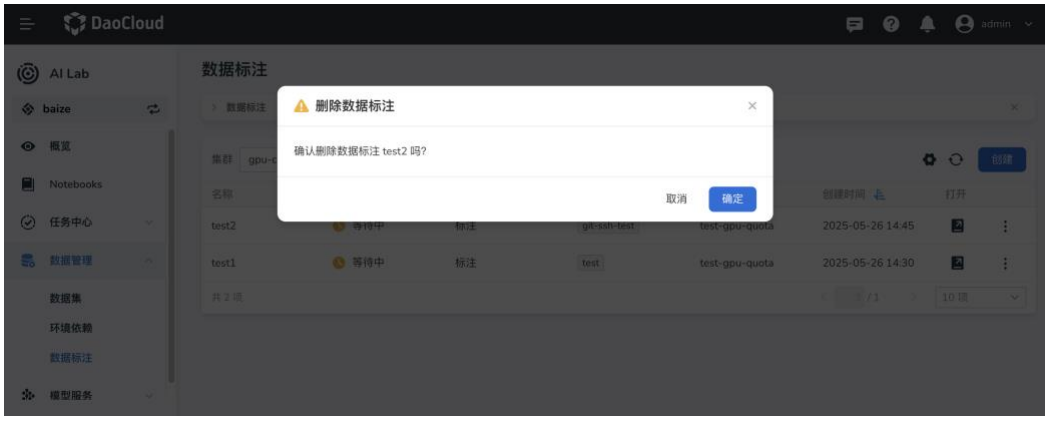

图 92　确认删除数据标注

3. 屏幕提示删除成功，该数据标注从列表中消失。

> 数据标注一旦删除将不可恢复，请谨慎操作。

六、模型服务

模型服务提供模型的创建、版本管理、公开模型下载与部署能力，并支持以 Triton、vLLM 作为推理框架，用户只需简单配置即可快速启动一个高性能的推理服务。

（一）模型列表

AI Lab 模型列表用于展示和管理各种机器学习或深度学习模型，方便用户快速进行数据科学和机器学习实验。您可以在不同集群、命名空间中创建模型和下载公开模型，进行模型管理。

创建模型

1. 在左侧导航栏中点击 **模型服务** -> **模型**，进入模型列表。点击右侧的 **创建** 按钮。

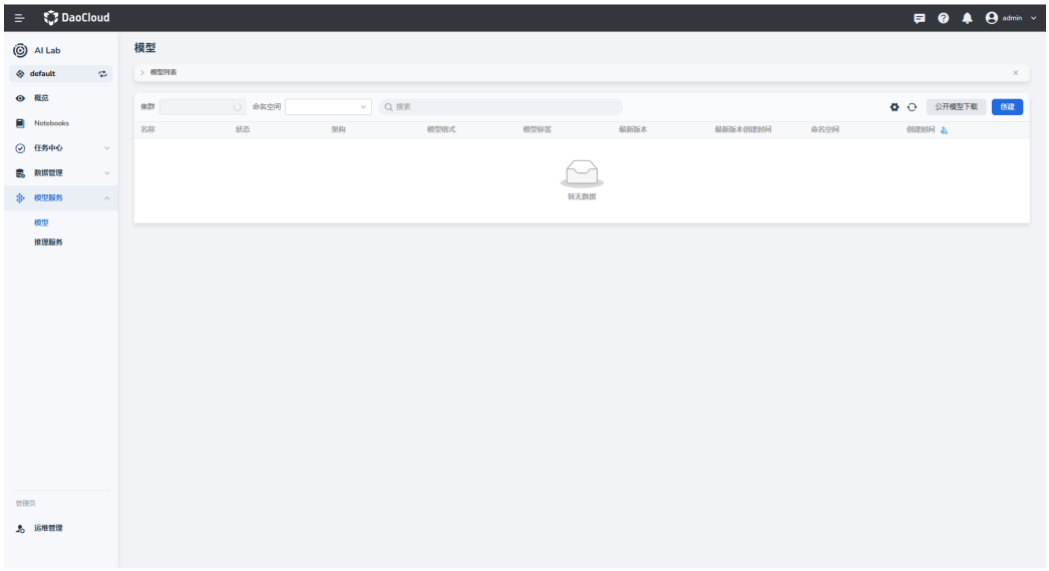

图 93　模型列表点击创建

2. 选择模型部署的集群、命名空间、架构、模型格式、模型标签、模型介绍、模型版本等，然后点击 **确定**。

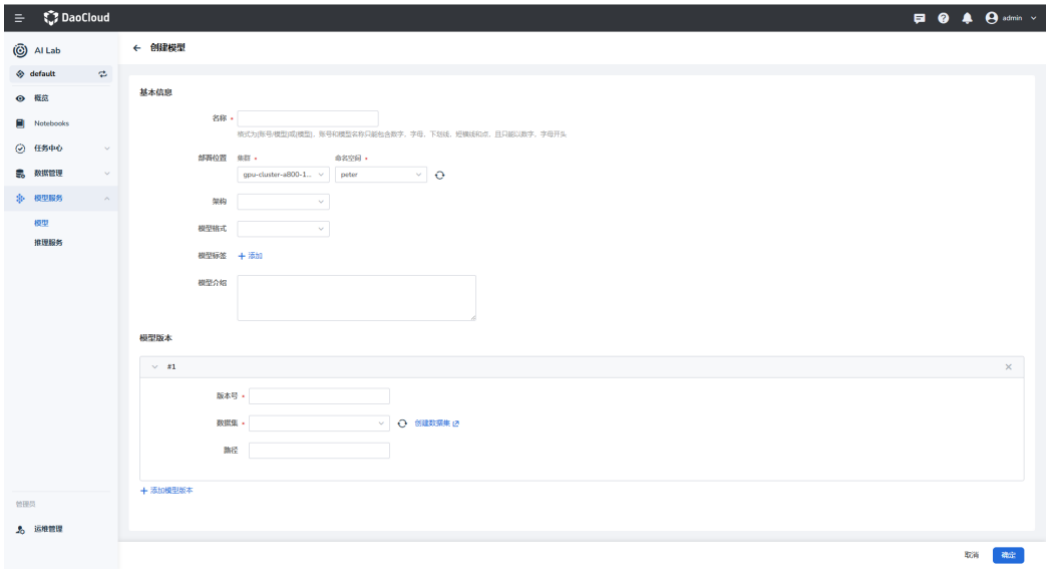

图 94　填写模型参数

目前支持的模型架构有：CNN、RNN、LSTM、TRANSFORMER、MLP、BERT、GAN、RESNET 等，模型格式包括 PYTORCH、TENSORFLOW 和 ONNX 三种格式。

公开模型下载

AI Lab 支持从魔搭社区和 HuggingFace 下载模型，您可以在 ModelScope 模型库和 HuggingFace Model Hub 查看更多模型。

点击模型列表右侧的 **公开模型下载** 按钮，选择 **公开模型来源**，设置模型部署的集群、命名空间、关联存储池、数据储存大小、Endpoint 等，填写 Token 授权访问。然后点击 **确定**。

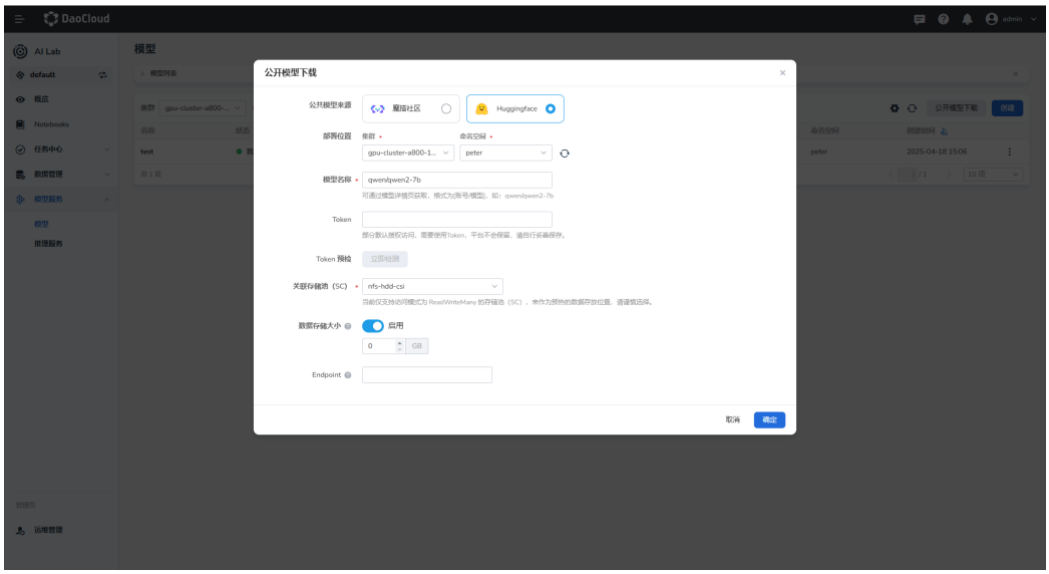

图 95　下载公开模型

模型部署

1. 在模型列表中，点击某个模型右侧的 ⋮，在弹出菜单中选择 **部署**。

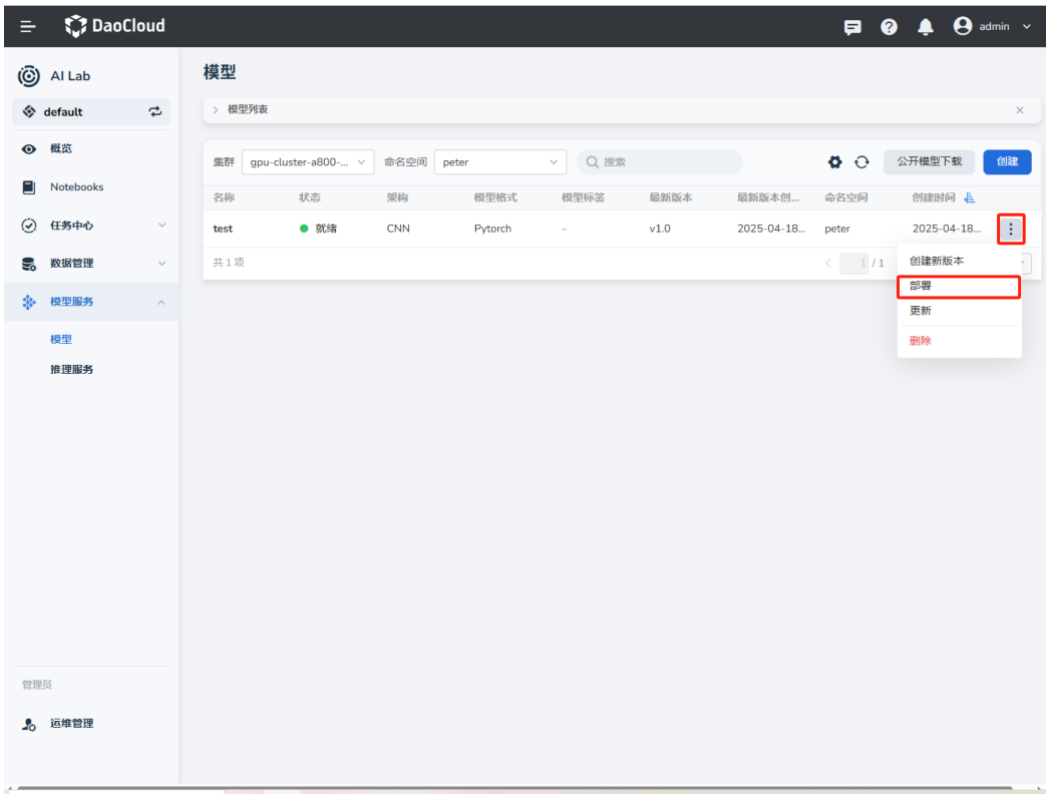

图 96　选择部署模型

2. 跳转到推理服务界面对模型进行部署。

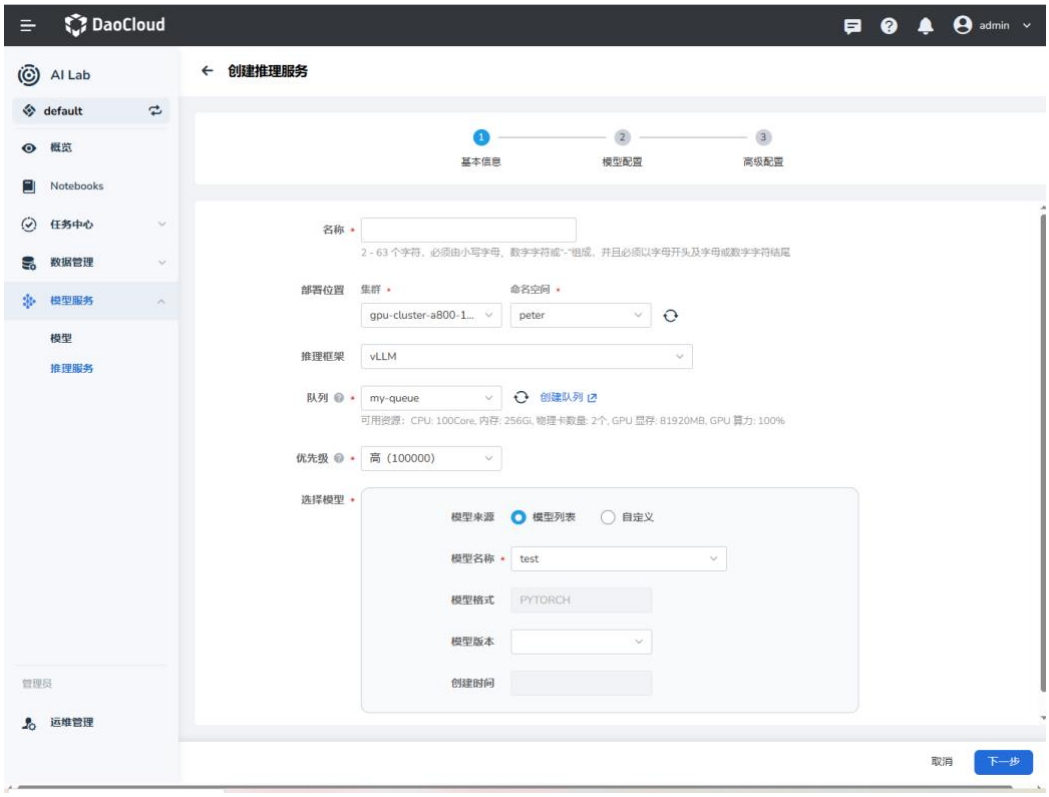

图 97　在推理服务界面部署模型

创建新版本

在模型列表右侧点击 ⋮ -> **创建新版本**，在弹窗中设置新版本模型的版本号、数据集、路径等内容。

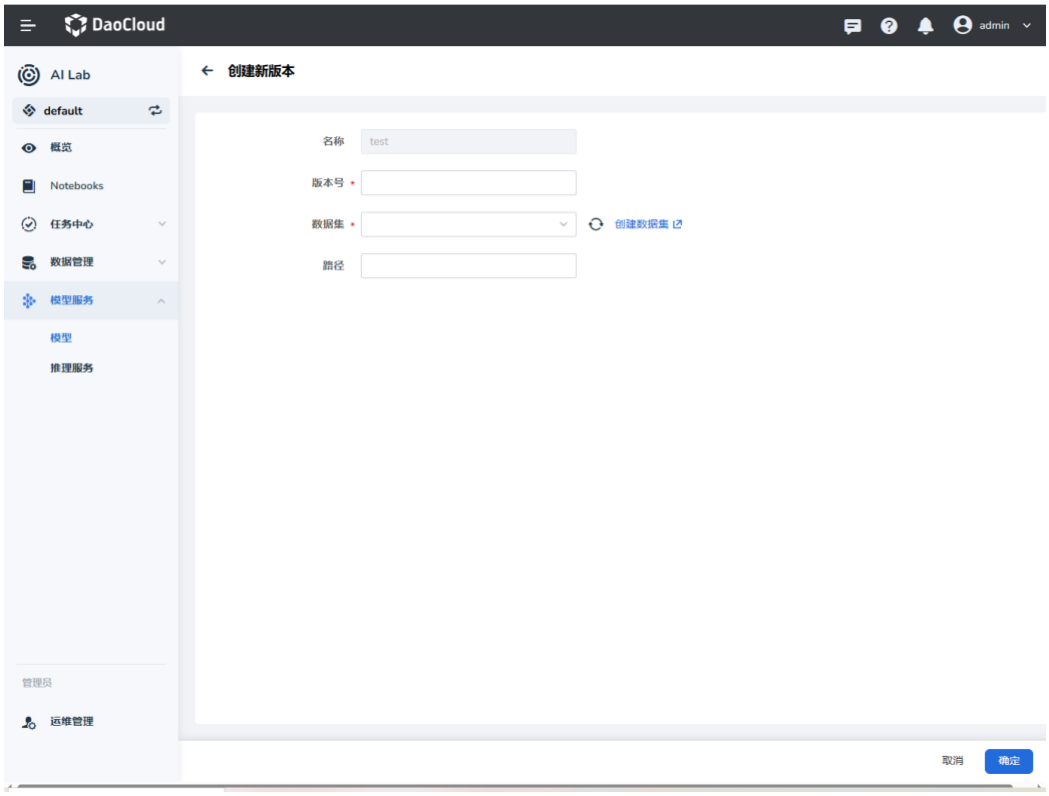

图 98　创建新版本

更新模型

在模型列表右侧点击 ⋮ -> **更新**，更新目标模型。

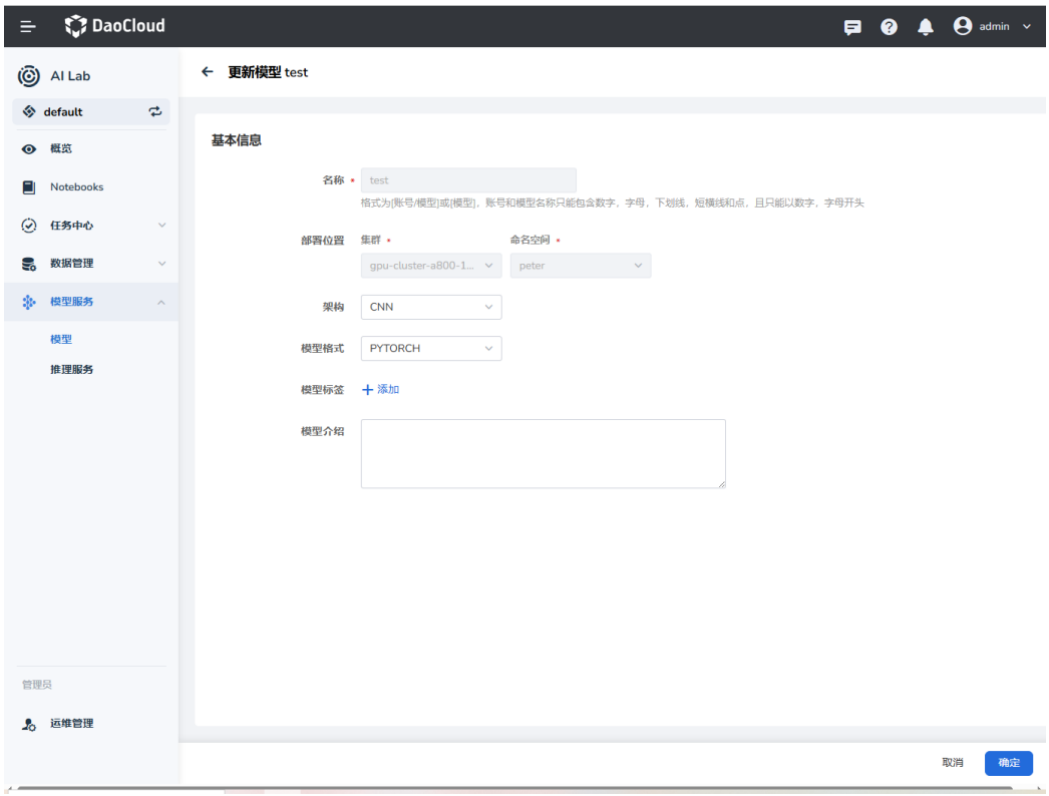

图 99　更新模型

删除模型

在模型列表右侧点击 ⋮ -> **删除**，在弹窗中确认要删除的模型，输入模型名称后点击 **删除**。

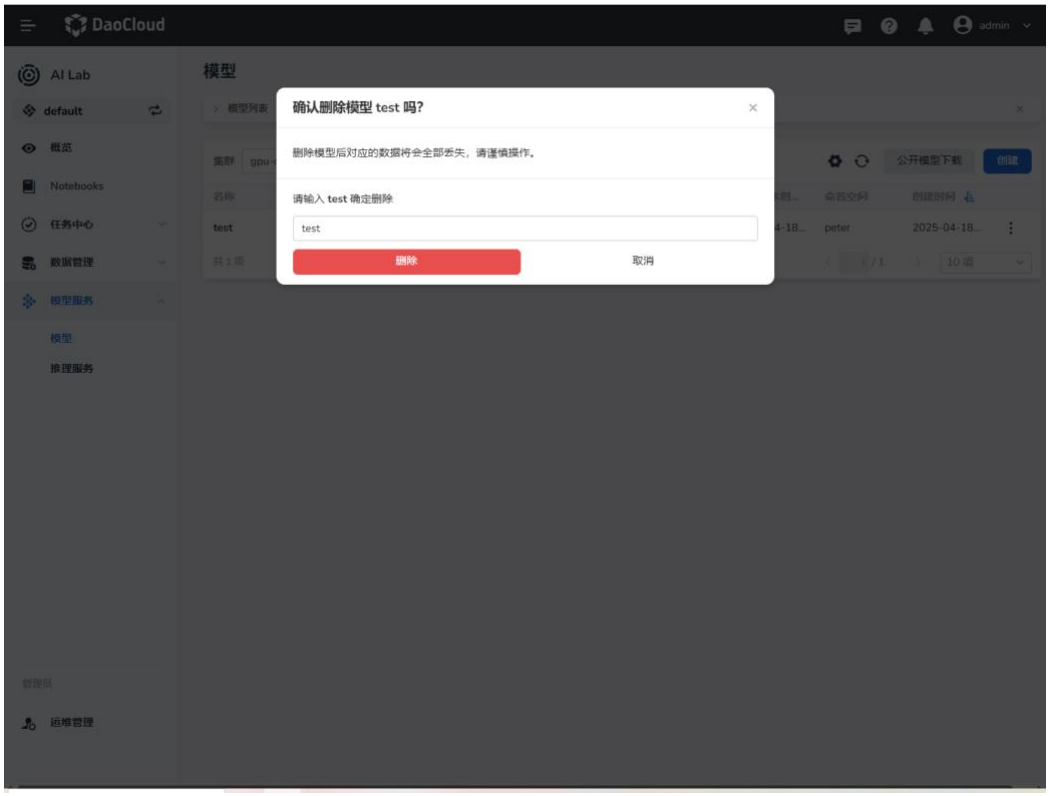

图 100　删除模型

> 模型一旦删除将不可恢复，请谨慎操作。

（二）模型支持情况

随着 AI Lab 的快速迭代，我们已经支持了多种模型的推理服务，您可以在这里看到所支持的模型信息。

- AI Lab v0.3.0 上线了模型推理服务，针对传统的深度学习模型，方便用户可以直接使用 AI Lab 的推理服务，无需关心模型的部署和维护。
- AI Lab v0.6.0 支持了完整版本的 vLLM 推理能力，支持诸多大语言模型，如 `LLama`、`Qwen`、`ChatGLM` 等。

> 推理能力的支持与 AI Lab 的版本有关，请查阅 Release Notes 了解最新版本并及时更新。

您可以在 AI Lab 中使用经过 DCE 验证过的 GPU 类型；更多细节参阅 DCE「GPU 支持矩阵」文档。

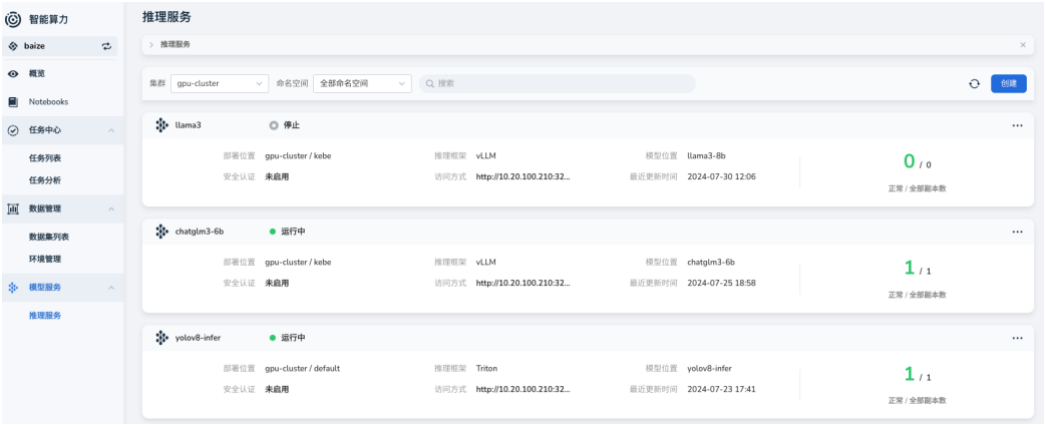

图 101 推理服务创建界面

Triton Inference Server

通过 Triton Inference Server 可以很好的支持传统的深度学习模型，我们目前支持主流的推理后端服务：

Backend	支持模型格式	介绍
pytorch	TorchScript、PyTorch 2.0 格式的模型	triton-inference-server/pytorch_backend
tensorflow	TensorFlow 2.x	triton-inference-server/tensorflow_backend
vLLM (Deprecated)	与 vLLM 一致	支持的模型和 vLLM support Model 一致

使用 Triton 的 Backend vLLM 的方式已被弃用，推荐使用最新支持 vLLM 来部署您的大语言模型。

vLLM

通过 vLLM 我们可以很快的使用大语言模型，您可以在这里看到我们支持的模型列表，这通常和 `vLLM Support Models` 保持一致。

- HuggingFace 模型：我们支持了 HuggingFace 的大部分模型，您可以在 HuggingFace Model Hub 查看更多模型。
- vLLM 支持模型：vLLM 官方文档列出了支持的大语言模型和视觉语言模型。
- 使用 vLLM 支持框架的模型进行微调后的模型。

vLLM 新特性

目前，AI Lab 还支持在使用 vLLM 作为推理工具时的一些新特性：

- 在推理模型时，启用 `Lora Adapter` 来优化模型推理服务。

- 提供兼容 OpenAI 的 OpenAPI 接口，方便用户切换到本地推理服务时，可以低成本的快速切换。

（三）创建 Triton 推理服务

AI Lab 目前提供以 Triton、vLLM 作为推理框架，用户只需简单配置即可快速启动一个高性能的推理服务。

> 使用 Triton 的 Backend vLLM 的方式已被弃用，推荐使用最新支持 vLLM 来部署您的大语言模型。

Triton 介绍

Triton 是由 NVIDIA 开发的一个开源推理服务器，旨在简化机器学习模型的部署和推理服务。它支持多种深度学习框架，包括 TensorFlow、PyTorch 等，使得用户能够轻松管理和部署不同类型的模型。

前提条件

准备模型数据：在数据集管理中纳管模型代码，并保证数据成功预加载，下面以 mnist 手写数字识别的 PyTorch 模型为例。

> 待推理的模型在数据集中需要遵以下目录格式：

```
<model-repository-name>
└──── <model-name>
    └──── <version>
        └──── <model-definition-file>
```

本例中的目录格式为：

```
model-repo
└──── mnist-cnn
    └──── 1
        └──── model.pt
```

创建推理服务

目前已经支持表单创建，可以按界面字段提示进行服务创建。

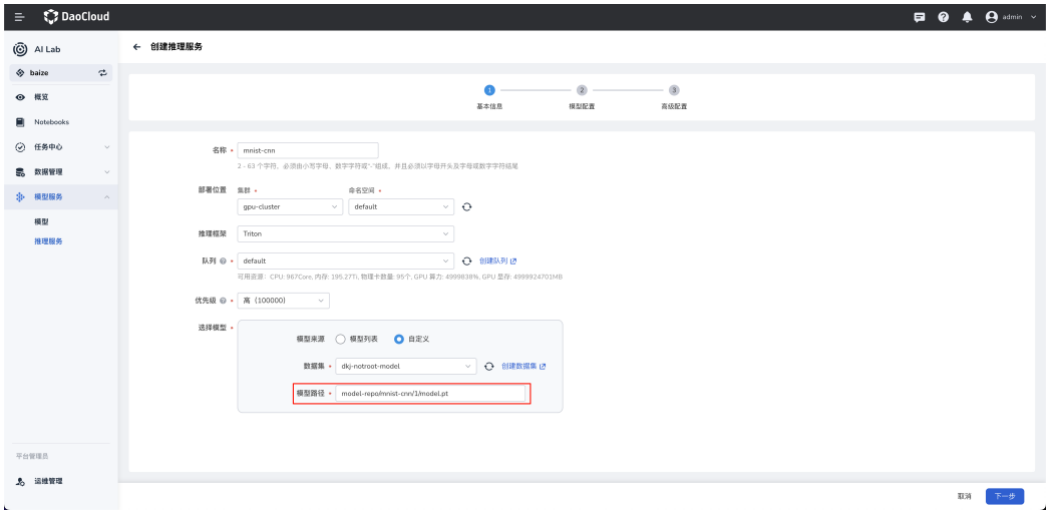

图 102　创建 Triton 推理服务

配置模型路径

模型路径 `model-repo/mnist-cnn/1/model.pt` 需要和数据集中的模型目录格式一致。

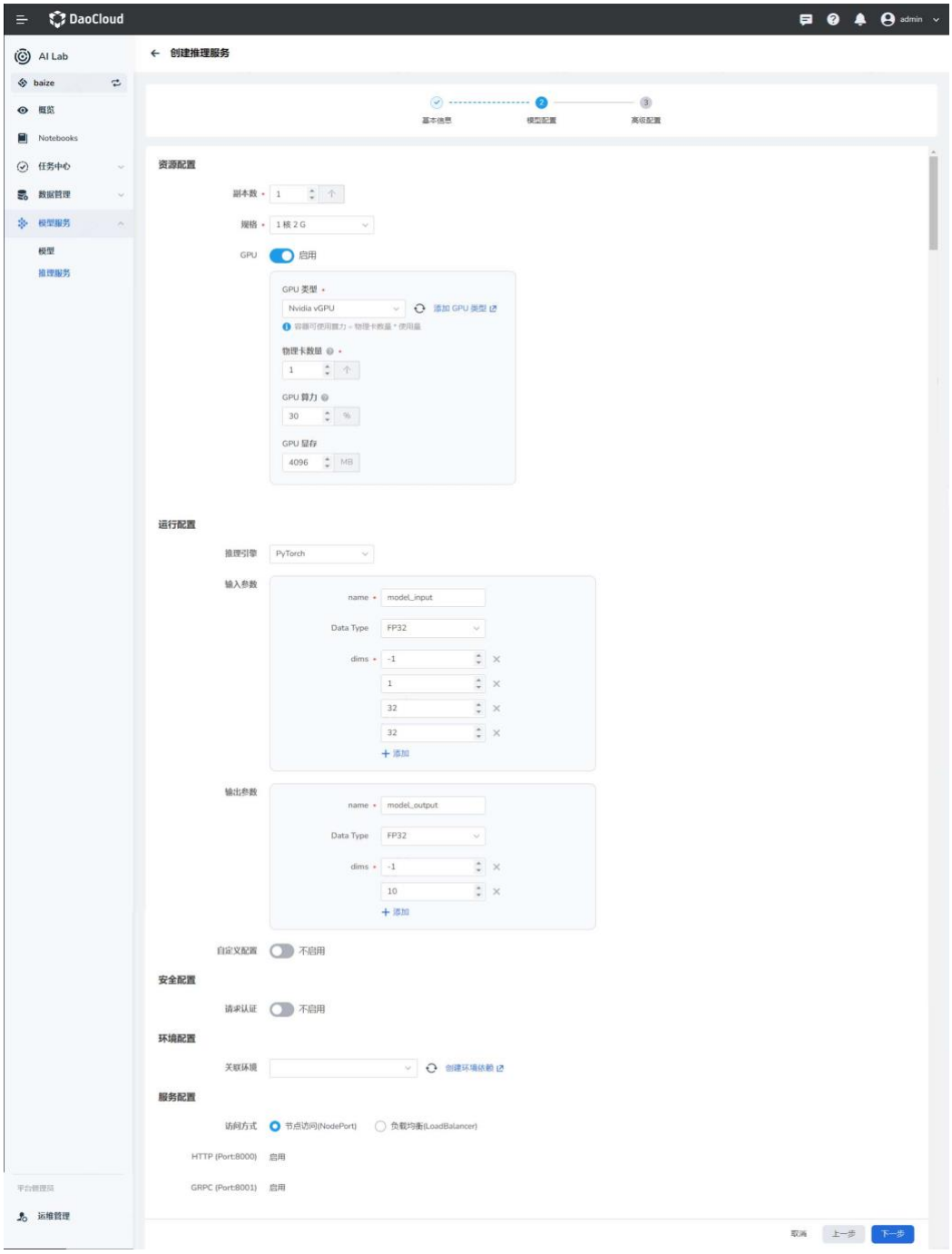

图 103　配置模型路径

配置输入和输出参数

> 输入和输出参数的第一个维度默认为 batchsize 的大小，设置为 -1 可以根据输入的推理数据自动计算 batchsize。参数其余维度和数据类型需要与模型输入匹配。

配置环境

可以导入环境管理中创建的环境作为推理时的运行环境。

高级配置

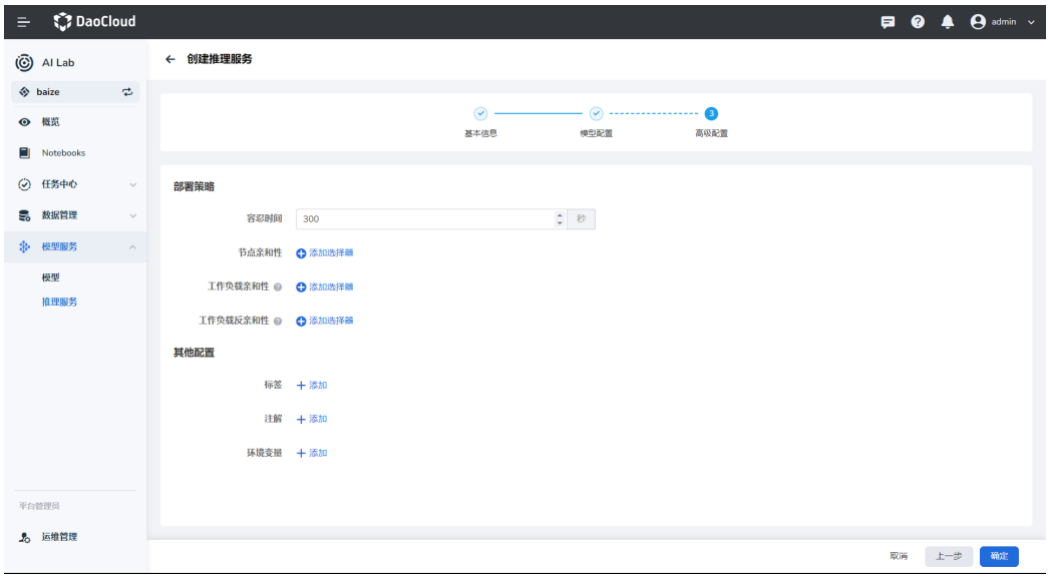

图 104　Triton 推理服务高级配置

配置认证策略

支持 API key 的请求方式认证，用户可以自定义增加认证参数。

亲和性调度

支持根据 GPU 资源等节点配置实现自动化的亲和性调度，同时也方便用户自定义调度策略。

访问

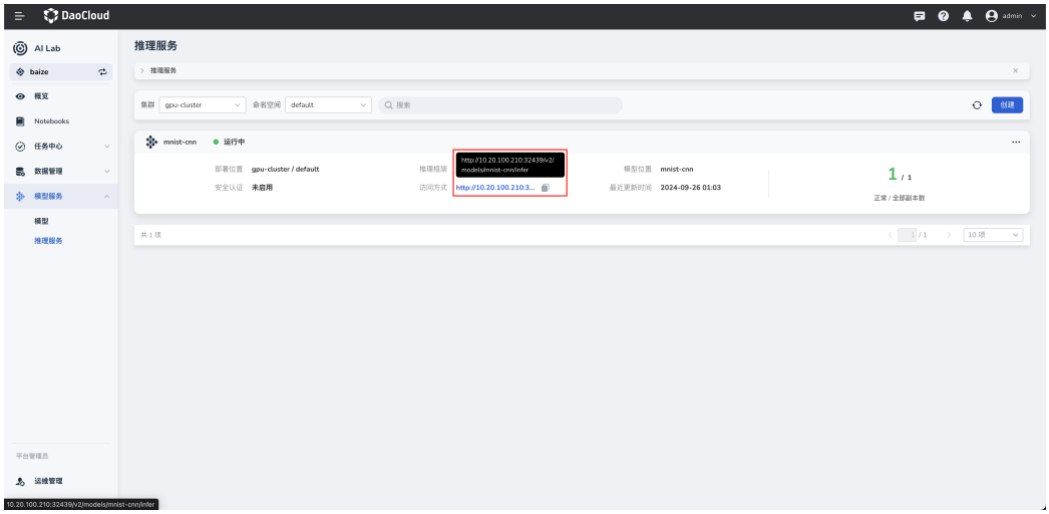

图 105　查看服务访问方式

API 访问

- Triton 提供了一个基于 REST 的 API，允许客户端通过 HTTP POST 请求进行模型推理。

- 客户端可以发送 JSON 格式的请求体，其中包含输入数据和相关的元数据。

HTTP 访问

1. 发送 HTTP POST 请求：使用工具如 `curl` 或 HTTP 客户端库（如 Python 的 `requests` 库）向 Triton Server 发送 POST 请求。

2. 设置 HTTP 头：根据用户配置项自动生成的配置，包含模型输入和输出的元数据。

3. 构建请求体：请求体通常包含要进行推理的输入数据，以及模型特定的元数据。

示例 curl 命令：

```
curl -X POST "http://<ip>:<port>/v2/models/<inference-name>/infer" \
-H "Content-Type: application/json" \
-d '{
  "inputs": [
    {
      "name": "model_input",
      "shape": [1, 1, 32, 32],
      "datatype": "FP32",
      "data": [
        [0.1234, 0.5678, 0.9101, ... ]
      ]
    }
  ]
}'
```

- `<ip>` 是 Triton Inference Server 运行的主机地址。
- `<port>` 是 Triton Inference Server 运行的主机端口号。
- `<inference-name>` 是所创建的推理服务的名称。
- `"name"` 要与模型配置中的输入参数的 `name` 一致。
- `"shape"` 要与模型配置中的输入参数的 `dims` 一致。
- `"datatype"` 要与模型配置中的输入参数的 `Data Type` 一致。
- `"data"` 替换为实际的推理数据。

请注意，上述示例代码需要根据你的具体模型和环境进行调整，输入数据的格式和内容也需要符合模型的要求。

（四）创建 vLLM 推理服务

AI Lab 支持以 vLLM 作为推理服务，提供全部 vLLM 的能力，同时提供了完全适配 OpenAI 接口定义。

vLLM 介绍

vLLM 是一个快速且易于使用的用于推理和服务的库，vLLM 旨在极大地提升实时场景下的语言模型服务的吞吐与内存使用效率。vLLM 在速度、灵活性方面具有以下部分特点：

- 连续批处理传入请求；

- 使用 PagedAttention 高效管理注意力键和值内存；
- 与流行的 HuggingFace 型号无缝集成；
- 兼容 OpenAI 的 API 服务器。

前提条件

准备模型数据：在数据集管理中纳管模型代码，并保证数据成功预加载。

创建推理服务

1. 选择 vLLM 推理框架，选择提前创建好的数据集，填写数据集中模型所在的 路径 信息。本文推理服务的创建使用 ChatGLM3 模型。

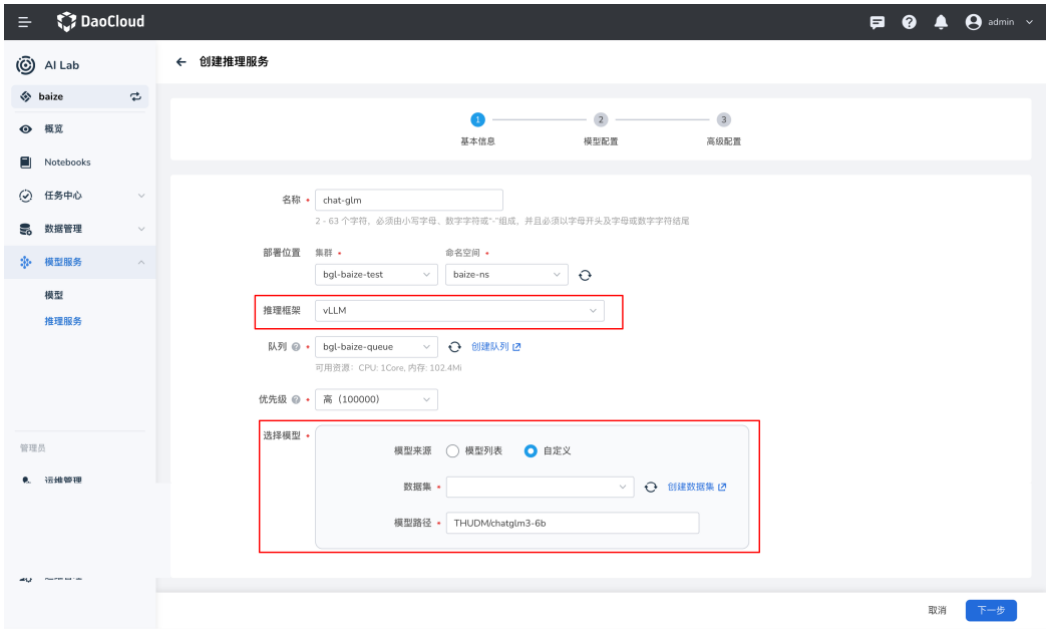

图 106　选择推理框架与模型路径

2. 配置推理服务的资源，并调整推理服务运行的参数。

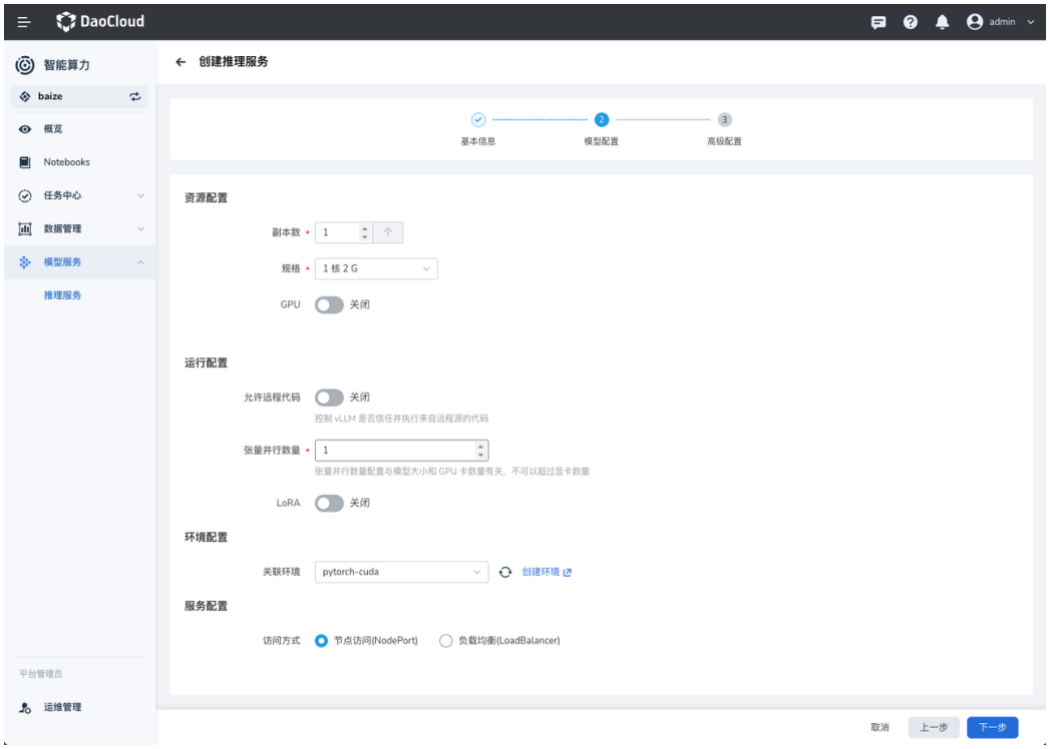

图 107　配置推理服务资源与参数

参数名	描述
GPU 资源	根据模型规模以及集群资源可以为推理配置 GPU 资源。
允许远程代码	控制 vLLM 是否信任并执行来自远程源的代码。
LoRA	**LoRA** 是一种针对深度学习模型的参数高效调整技术。它通过将原始模型参数矩阵分解为低秩矩阵，从而减少参数数量和计算复杂度。 （1）--lora-modules：用来指定特定模块或层进行低秩近似。 （2）max_loras_rank：用来指定 LoRA 模型中每个适配层的最大秩，对于简单的任务，可以选择较小的秩值，而对于复杂任务，可能需要较大的秩值来保证模型性能。 （3）max_loras：表示模型中可以包含的 LoRA 层的最大数量，根据模型大小、推理复杂度等因素自定。 （4）max_cpu_loras：用于指定在 CPU 环境中可以处理的 LoRA 层的最大数。
关联环境	通过选择环境预定义推理时所需的环境依赖。

支持配置 LoRA 参数的模型可参考 vLLM 官方文档中的「支持的模型」列表。

3. 在 **高级配置** 中，支持根据 GPU 资源等节点配置实现自动化的亲和性调度，同时也方便用户自定义调度策略。

验证推理服务

推理服务创建完成之后，点击推理服务名称进入详情，查看 API 调用方法。通过使用 Curl、Python、Nodejs 等方式验证执行结果。

拷贝详情中的 `curl` 命令，并在终端中执行命令发送一条模型推理请求。

图 108　验证推理服务

七、运维管理

运维管理是 IT 运维人员日常监控、管理和优化 IT 资源与工作负载的重要空间。通过运维管理平台，管理员可以直观地掌握集群资源使用情况，监控关键硬件指标，并高效调度任务，从而保证系统稳定性和资源利用率。

（一）运维管理概览

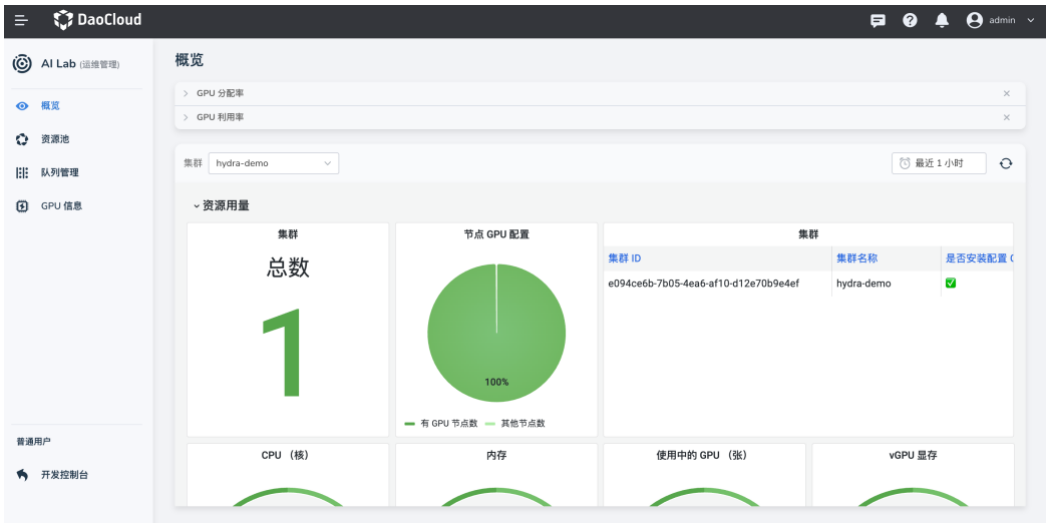

图 109　运维管理概览

为了满足不同运维场景的需求，运维管理模块设计了以下几个核心子页面：

- **概览**：提供集群总体视图，通过大屏展示关键指标，包括节点资源用量、GPU 使用情况、GPU 功率以及 GPU 设备温度等。运维人员可以快速识别集群瓶颈和资源异常，做出及时决策。
- **资源池**：用于定义集群中可用的计算资源对象，包括 CPU、内存和 GPU 等。通过资源池，可以将工作负载与特定节点类型绑定，实现精细化资源分配与管理，从而提高任务调度效率和集群整体性能。

- **队列管理**：用于管理和优化批处理工作负载，通过队列系统对任务进行调度。队列管理可以合理分配资源，平衡高优先级和低优先级任务的执行顺序，从而提高集群任务吞吐量，降低资源空闲率。
- **GPU 信息**：自动化汇总整个平台的 GPU 资源信息，提供详尽的 GPU 设备状态展示。管理员可以查看每块 GPU 的负载统计、功率使用情况、温度及正在运行的任务信息，支持 GPU 资源监控和优化调度。

常见术语

- **GPU 分配率**：表示当前集群内所有未完成任务对 GPU 资源的占用比例。它反映了资源的预占情况，帮助管理员判断资源是否紧张。
- **GPU 利用率**：表示当前集群中所有运行任务的 GPU 实际使用情况。它衡量资源的实际消耗效率，帮助管理员评估任务调度和 GPU 负载分布是否合理。

扩展功能与实践建议

1. **实时监控与告警**：配合概览页面，设置 GPU 温度、功率和利用率的阈值告警，可在异常情况下即时通知运维人员，避免硬件损耗和任务失败。

2. **资源池策略优化**：针对不同类型的任务（短作业、长作业、GPU 密集型任务等），合理划分资源池，提高资源复用率，减少任务排队等待时间。

3. **队列调度优化**：结合任务优先级、资源需求和历史运行时间，设计公平或加权调度策略，以提高任务完成率和集群整体吞吐量。

4. **GPU 资源分析**：定期分析 GPU 利用率与分配率数据，识别低效使用的资源，调整任务分配或迁移策略，实现 GPU 使用最大化。

5. **运维报表**：可生成定期报表，包括资源使用趋势、任务完成情况和 GPU 使用效率，为决策提供数据支撑。

（二）创建资源池

资源池用于定义集群中可用的计算资源，并通过将工作负载与特定节点类型关联，实现细粒度的资源管理。以下是创建资源池的步骤。

1. 进入 **运维管理** 视图，选择左侧菜单 **资源池**，点击资源池列表右上角的 **创建资源池** 按钮。

图 110　点击创建资源池

2. 根据需求，填写表单信息，然后点击 **确定**。

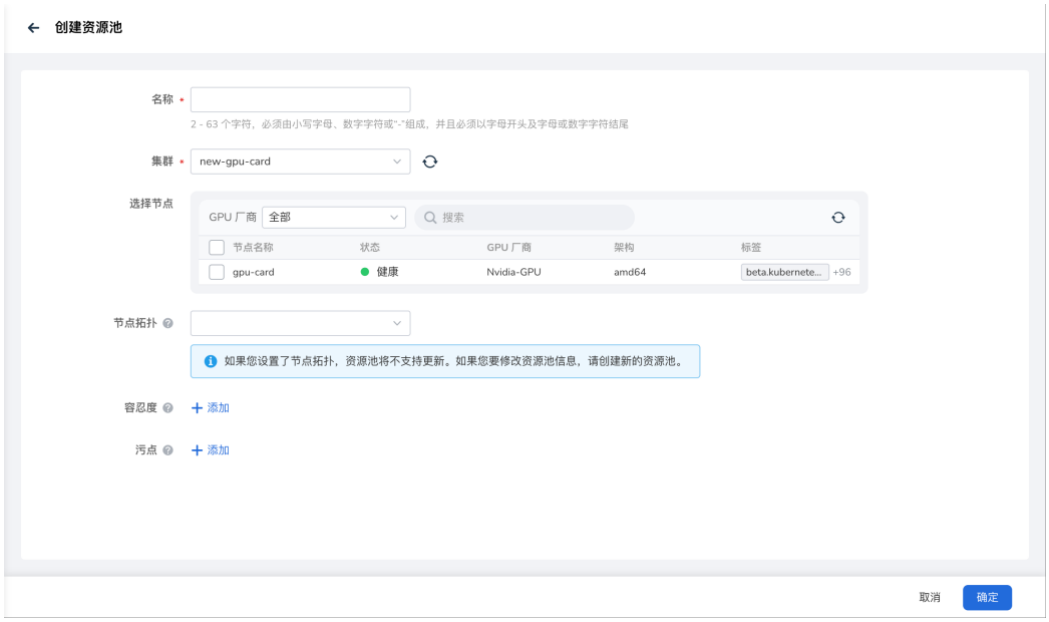

图 111　填写资源池参数

字段	说明
名称	2-63 个字符，必须由小写字母、数字字符或「-」组成，并且必须以字母开头、以字母或数字结尾。
集群	资源池节点所在集群。
选择节点	可以根据 GPU 厂商或节点名称搜索节点。
节点拓扑	描述 Kubernetes 集群中节点在数据中心的物理或逻辑层级结构（如区块、机架、主机）。通过定义拓扑，可优化调度，使 Pod 尽可能运行在最佳网络链路的节点上，从而降低网络延迟，适合 AI/ML 训练或高性能计算任务。
容忍度	允许调度器将带有对应污点的 Pod 调度到节点上。

污点	使节点能够排斥特定类型的 Pod。污点与容忍度结合，可避免 Pod 被分配到不合适的节点。

3. 屏幕提示创建成功，返回资源池管理列表。点击列表右侧的 ⋮，可以执行更新、删除等更多操作。

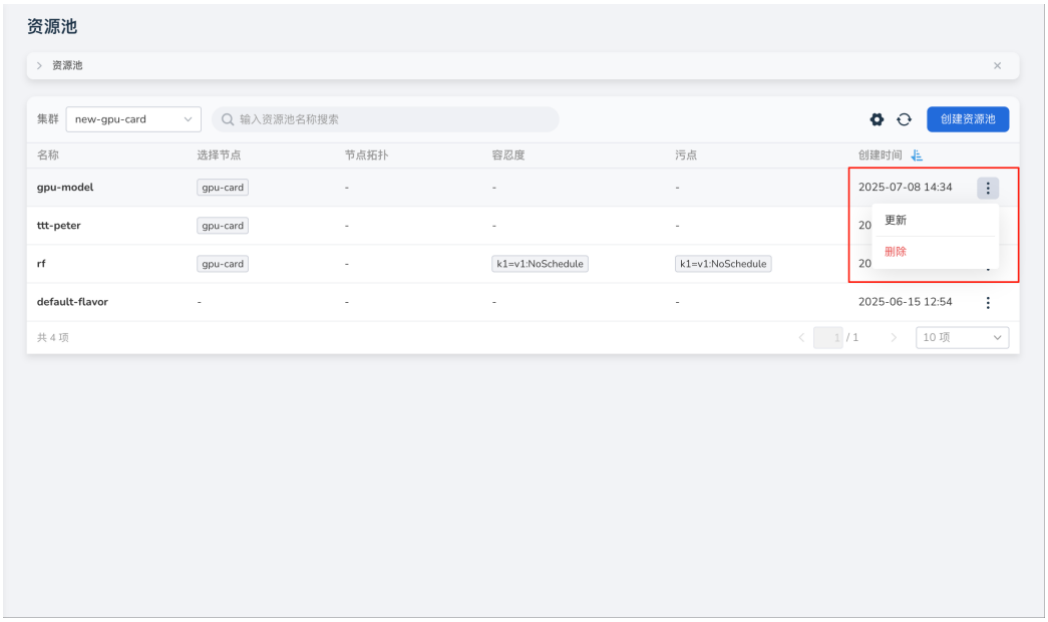

图 112　资源池更多操作

（三）管理资源池

AI Lab 支持对资源池进行更新、删除操作。

更新

> 如果资源池设置了节点拓扑，资源池将不支持更新。如果您要修改资源池信息，请创建新的资源池。

1. 在资源池列表右侧点击 ⋮，在弹出菜单中选择 **更新**。

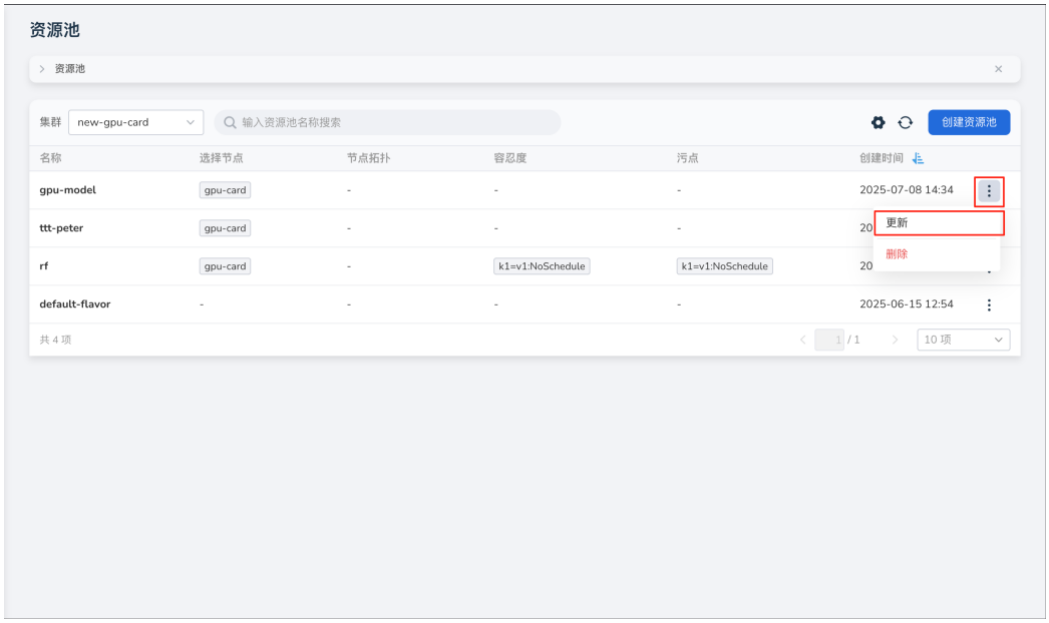

图 113　选择更新资源池

2. 在更新页面，修改配置参数，完成后点击 **确认**，系统自动返回到资源池列表页面。

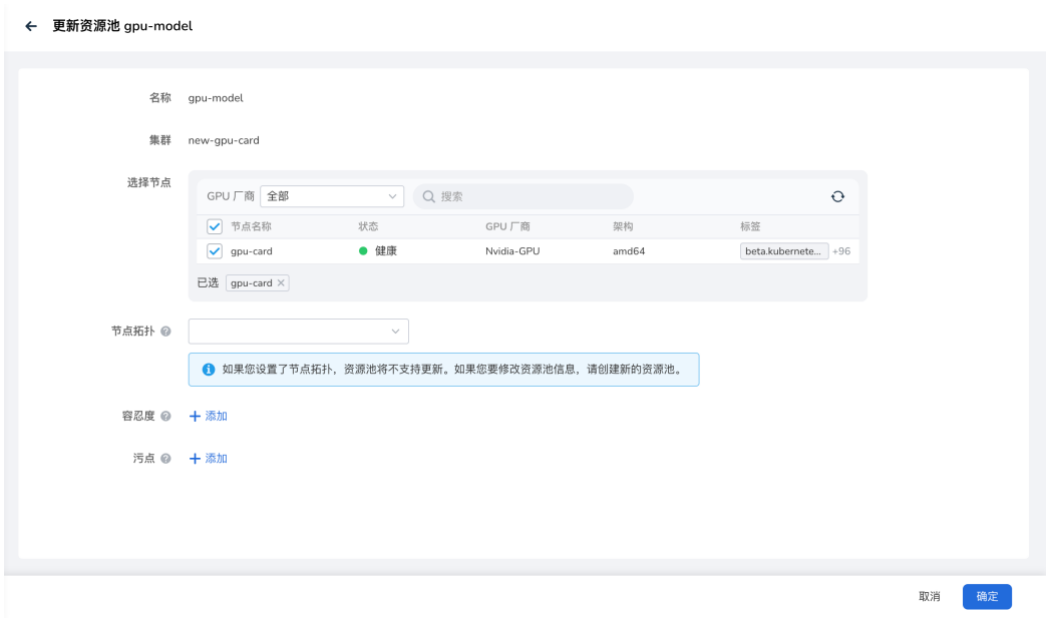

图 114　修改并确认

删除

> 如果您当前的资源池绑定了队列，并且队列下有正在运行的工作负载，请谨慎此操作。

1. 在资源池列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

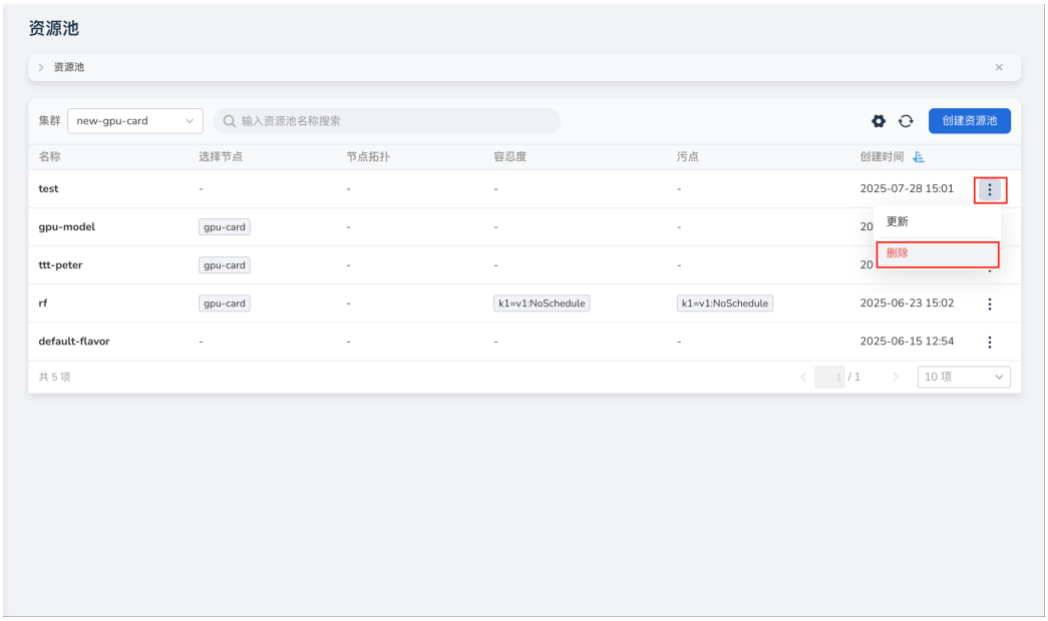

图 115　选择删除资源池

2. 在二次确认删除弹窗中点击 **删除**。

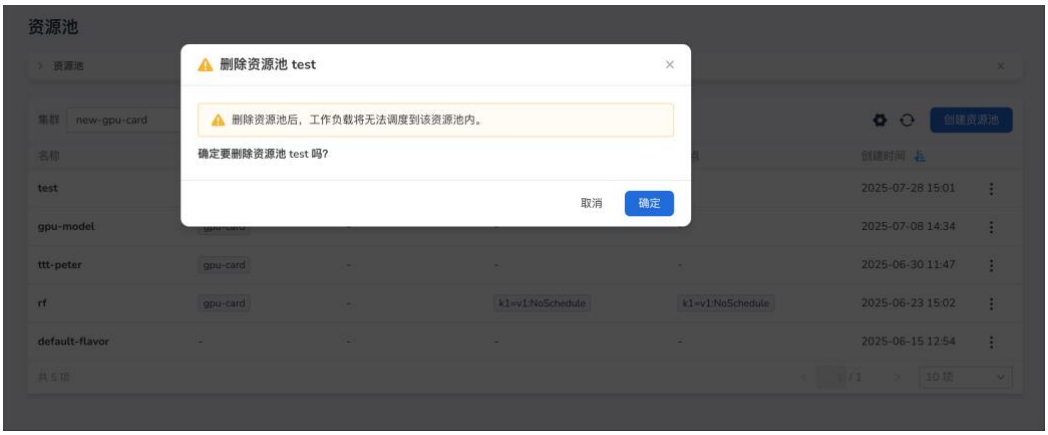

图 116　确认删除资源池

3. 屏幕提示删除成功，该资源池从列表中消失。

（四）创建队列

在运维管理模式中，队列可用于调度和优化批处理工作负载，它可以有效地管理在集群上运行的多个任务，通过队列系统来优化资源利用率。

1. 在左侧导航栏中点击 **队列管理**，点击右侧的 **创建** 按钮。

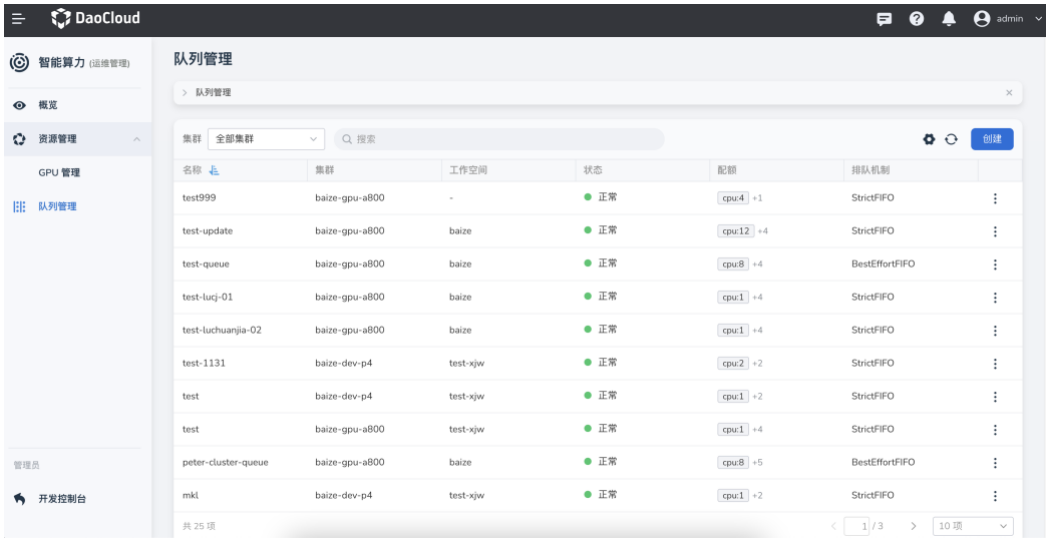

图 117　队列列表点击创建

2. 系统会预先填充基础设置数据，包括要部署的集群、工作空间、排队策略等。调整这些参数后点击 **确定**。

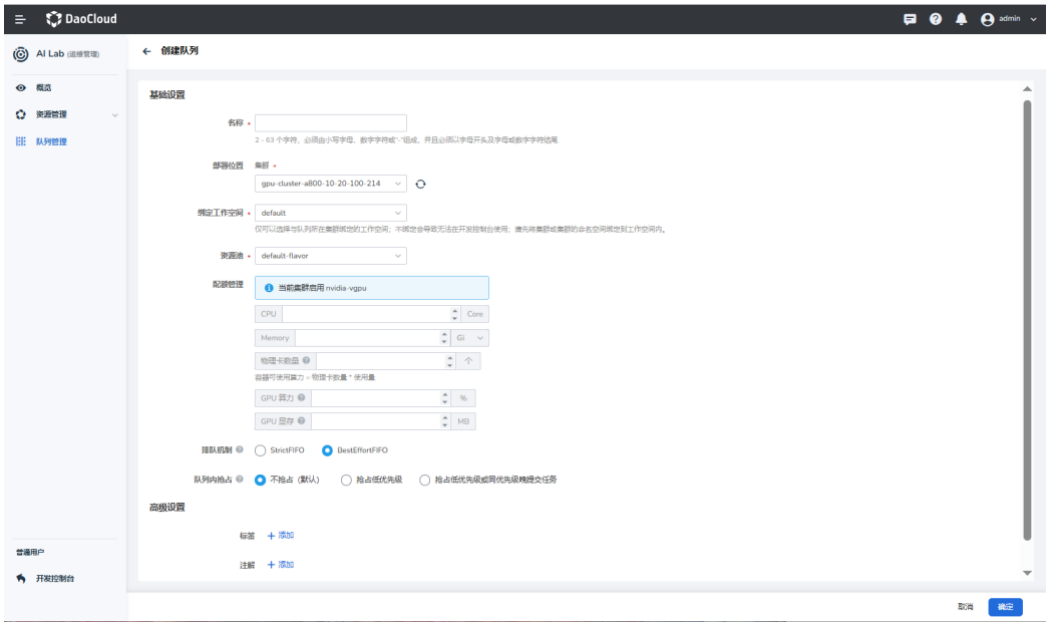

图 118　填写队列参数

3. 屏幕提示创建成功，返回队列管理列表。点击列表右侧的 ⋮，可以执行更新、删除等更多操作。

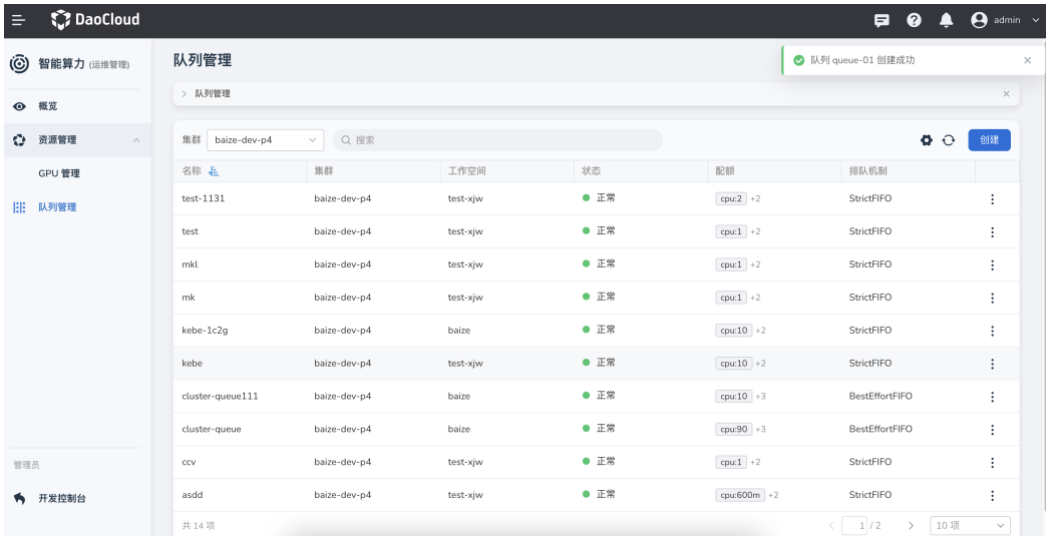

图 119　队列更多操作

（五）删除队列

在运维管理模式中，如果发现队列冗余、过期或因其他缘故不再需要，可以从队列列表中删除。

1. 在队列列表右侧点击 ⋮，在弹出菜单中选择 **删除**。

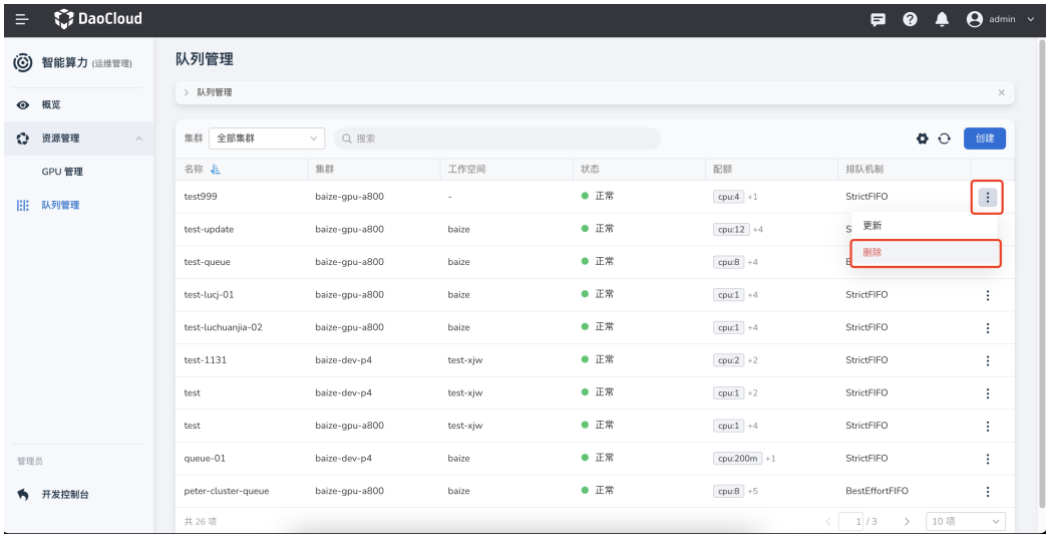

图 120　选择删除队列

2. 在弹窗中确认要删除的队列，输入队列名称后点击 **删除**。

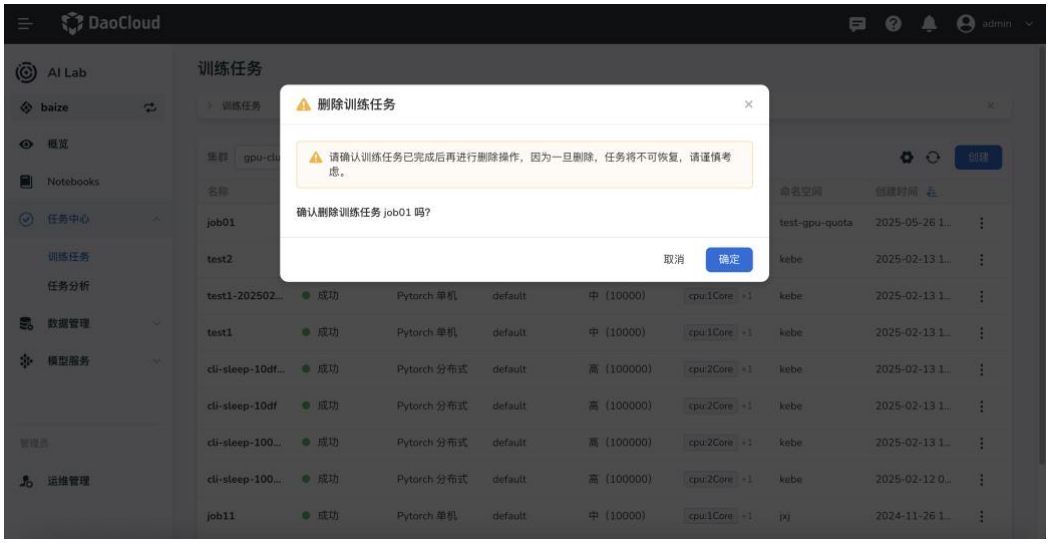

图 121　确认删除队列

3. 屏幕提示删除成功，该队列从列表中消失。

> 队列一旦删除将不可恢复，请谨慎操作。

（六）GPU 信息

自动化汇总整个平台中的 GPU 资源信息，提供详尽的 GPU 设备信息展示，可查看各种 GPU 卡的负载统计和任务运行信息。

进入 **运维管理** 后，点击左侧导航栏的 **GPU 信息**，可以查看 GPU 卡和各种训练任务。

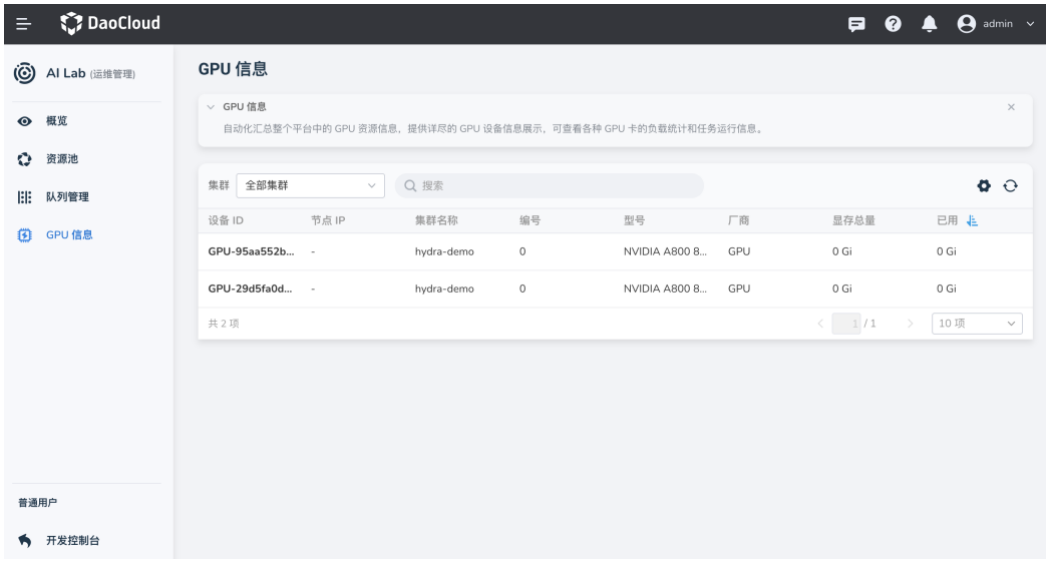

图 122　GPU 信息列表

点击某个 **设备 ID**，可以查看 GPU 的详情信息。

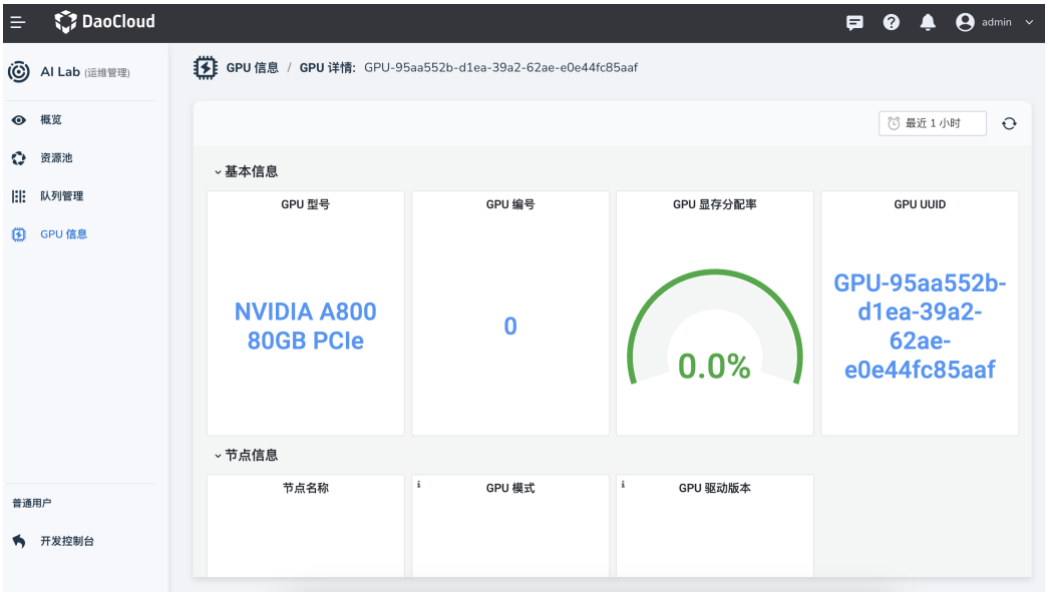

图 123 GPU 详情

八、最佳实践

本章汇总 AI Lab 在实际交付与使用中的典型场景与推荐做法，包括存储与数据预热、镜像维护、调度器扩展、拓扑感知调度、训练稳定性（Checkpoint / DeepSpeed），以及完整的模型微调与数据标注工具部署实践。

（一）部署 NFS 做数据集预热

网络文件系统（NFS）允许远程主机通过网络挂载文件，并像本地文件系统一样进行交互。这使系统管理员能够将资源集中到网络服务器上进行管理。

数据集 是 DCE AI Lab 中的核心数据管理功能，将 MLOps 生命周期中对于数据的依赖统一抽象为数据集；支持用户将各类数据纳管到数据集内，以便训练任务可以直接使用数据集中的数据。

当远端数据不在工作集群内时，数据集提供了自动进行预热的能力，支持 `Git`、`S3`、`HTTP` 等数据提前预热到集群本地。

数据集需要一个支持 `ReadWriteMany` 模式的存储服务对远端数据进行预热，推荐在集群内部署 NFS。本节主要介绍了如何快速部署一个 NFS 服务，并将其添加为集群的「存储类」。

准备工作

- NFS 默认使用节点的存储作为数据缓存点，因此需要确认磁盘本身有足够的磁盘空间。
- 安装方式使用 `Helm` 与 `Kubectl`，请确保已经安装好。

部署过程

一共需要安装几个组件：NFS Server、csi-driver-nfs、StorageClass。

1. 初始化命名空间

所有系统组件会安装到 `nfs` 命名空间内，因此需要先创建此命名空间。

```
kubectl create namespace nfs
```

2. 安装 NFS Server

这里是一个简单的 YAML 部署文件，可以直接使用。

> 注意检查 image:，根据集群所在位置情况，可能需要修改为国内镜像。

```
kind: Service
apiVersion: v1
metadata:
  name: nfs-server
  namespace: nfs
  labels:
    app: nfs-server
spec:
  type: ClusterIP
  selector:
    app: nfs-server
  ports:
    - name: tcp-2049
      port: 2049
      protocol: TCP
    - name: udp-111
      port: 111
      protocol: UDP
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nfs-server
  namespace: nfs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nfs-server
  template:
    metadata:
      name: nfs-server
      labels:
        app: nfs-server
    spec:
      nodeSelector:
        "kubernetes.io/os": linux
      containers:
        - name: nfs-server
          image: itsthenetwork/nfs-server-alpine:latest
          env:
            - name: SHARED_DIRECTORY
              value: "/exports"
          volumeMounts:
            - mountPath: /exports
              name: nfs-vol
          securityContext:
            privileged: true
```

```
          ports:
            - name: tcp-2049
              containerPort: 2049
              protocol: TCP
            - name: udp-111
              containerPort: 111
              protocol: UDP
      volumes:
        - name: nfs-vol
          hostPath:
            path: /nfsdata  # 修改此处以指定另一个路径来存储 NFS 共享数据
            type: DirectoryOrCreate
```

将上述 YAML 保存为 nfs-server.yaml，然后执行以下命令进行部署：

```
kubectl -n nfs apply -f nfs-server.yaml

# 检查部署结果
kubectl -n nfs get pod,svc
```

3. 安装 csi-driver-nfs

安装 csi-driver-nfs 需要使用 Helm，请注意提前安装。

```
# 添加 Helm 仓库
helm repo add csi-driver-nfs
https://mirror.ghproxy.com/https://raw.githubusercontent.com/kubernetes-csi/csi-driver-
nfs/master/charts
helm repo update csi-driver-nfs

# 部署 csi-driver-nfs
# 这里参数主要优化了镜像地址，加速国内下载
helm upgrade --install csi-driver-nfs csi-driver-nfs/csi-driver-nfs \
    --set image.nfs.repository=k8s.m.daocloud.io/sig-storage/nfsplugin \
    --set image.csiProvisioner.repository=k8s.m.daocloud.io/sig-storage/csi-provisioner
\
    --set image.livenessProbe.repository=k8s.m.daocloud.io/sig-storage/livenessprobe \
    --set image.nodeDriverRegistrar.repository=k8s.m.daocloud.io/sig-storage/csi-node-
driver-registrar \
    --namespace nfs \
    --version v4.5.0
```

> csi-nfs-controller 的镜像并未全部支持 helm 参数，需要手工修改 deployment 的 image 字段。将 image: registry.k8s.io 改为 image: k8s.dockerproxy.com 以加速国内下载。

4. 创建 StorageClass

将以下 YAML 保存为 nfs-sc.yaml：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: nfs-server.nfs.svc.cluster.local
  share: /
  # csi.storage.k8s.io/provisioner-secret is only needed for providing mountOptions in
DeleteVolume
  # csi.storage.k8s.io/provisioner-secret-name: "mount-options"
  # csi.storage.k8s.io/provisioner-secret-namespace: "default"
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

```
mountOptions:
  - nfsvers=4.1
```

然后执行以下命令进行部署：

```
kubectl apply -f nfs-sc.yaml
```

测试

创建数据集，并将数据集的 **关联存储类**、预热方式 设置为 NFS，即可将远端数据预热到集群内。

数据集创建成功后，可以看到数据集的状态为 预热中，等待预热完成后即可使用。

常见问题

缺少必要的 NFS 客户端软件 `/sbin/mount`

```
bad option; for several filesystems (e.g. nfs, cifs) you might need a /sbin/mount.<type>
helper program.
```

在运行 Kubernetes 的节点机器上，确保已安装 NFS 客户端：

Ubuntu/Debian

```
sudo apt-get update
sudo apt-get install nfs-common
```

CentOS/RHEL

```
sudo yum install nfs-utils
```

检查 NFS 服务器配置，确保 NFS 服务器正在运行且配置正确。你可以尝试运行以下命令手动挂载来测试：

```
sudo mkdir -p /mnt/test
sudo mount -t nfs <nfs-server>:/nfsdata /mnt/test
```

（二）更新 Notebook 内置镜像

在 Notebook 中，默认提供了多个可用的基础镜像，供开发者选择；大部分情况下，这会满足开发者的使用。

图 124　创建 Notebook 时选择镜像

DaoCloud 提供了一个默认的 Notebook 镜像，包含了所需的任何开发工具和资料。

```
baize/baize-notebook
```

这个 Notebook 里面包含了基础的开发工具，以 `baize-notebook:v0.5.0`（2024 年 5 月 30 日）为例，相关依赖及版本如下：

依赖	版本编号	介绍
Ubuntu	22.04.3	默认 OS
Python	3.11.6	默认 Python 版本
pip	23.3.1	
conda(mamba)	23.3.1	
jupyterlab	3.6.6	JupyterLab 镜像，提供完整的 Notebook 开发体验
codeserver	v4.89.1	主流 Code 开发工具，方便用户使用熟悉的工具进行开发体验
*baizectl	v0.5.0	DaoCloud 内置 CLI 任务管理工具
*SSH	-	支持本地 SSH 直接访问到 Notebook 容器内
*kubectl	v1.27	Kubernetes CLI，可以使用 kubectl 在 Notebook 内管理容器资源

> 随着版本发展，DCE 会主动维护并在每次版本迭代时更新。

但有时用户可能需要自定义镜像，本节介绍了如何更新镜像，并增加到 Notebook 创建界面中进行选择。

构建自定义镜像（仅供参考）

> 注意，构建新镜像 **需要以 baize-notebook 作为基础镜像**，以保证 Notebook 的正常运行。

在构建自定义镜像时，建议先了解 baize-notebook 镜像的 Dockerfile，以便更好地理解如何构建自定义镜像。

baize-notebook 的 Dockerfile

```
ARG BASE_IMG=docker.m.daocloud.io/kubeflownotebookswg/jupyter:v1.8.0

FROM $BASE_IMG

USER root

# install - useful linux packages
RUN export DEBIAN_FRONTEND=noninteractive \
 && apt-get -yq update \
 && apt-get -yq install --no-install-recommends \
    openssh-server git git-lfs bash-completion \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*

# remove default s6 jupyterlab run script
RUN rm -rf /etc/services.d/jupyterlab

# install - useful jupyter plugins
RUN mamba install -n base -y jupyterlab-language-pack-zh-cn \
  && mamba clean --all -y

ARG CODESERVER_VERSION=4.89.1
ARG TARGETARCH

RUN curl -fsSL "https://github.com/coder/code-
server/releases/download/v$CODESERVER_VERSION/code-
server_${CODESERVER_VERSION}_$TARGETARCH.deb" -o /tmp/code-server.deb \
  && dpkg -i /tmp/code-server.deb \
  && rm -f /tmp/code-server.deb

ARG CODESERVER_PYTHON_VERSION=2024.4.1
ARG CODESERVER_JUPYTER_VERSION=2024.3.1
ARG CODESERVER_LANGUAGE_PACK_ZH_CN=1.89.0
ARG CODESERVER_YAML=1.14.0
ARG CODESERVER_DOTENV=1.0.1
ARG CODESERVER_EDITORCONFIG=0.16.6
ARG CODESERVER_TOML=0.19.1
ARG CODESERVER_GITLENS=15.0.4

# configure for code-server extensions
RUN code-server --list-extensions --show-versions \
  && code-server \
    --install-extension MS-CEINTL.vscode-language-pack-zh-
hans@$CODESERVER_LANGUAGE_PACK_ZH_CN \
    --install-extension ms-python.python@$CODESERVER_PYTHON_VERSION \
    --install-extension ms-toolsai.jupyter@$CODESERVER_JUPYTER_VERSION \
    --install-extension redhat.vscode-yaml@$CODESERVER_YAML \
    --install-extension mikestead.dotenv@$CODESERVER_DOTENV \
```

```
    --install-extension EditorConfig.EditorConfig@$CODESERVER_EDITORCONFIG \
    --install-extension tamasfe.even-better-toml@$CODESERVER_TOML \
    --install-extension eamodio.gitlens@$CODESERVER_GITLENS \
    --install-extension catppuccin.catppuccin-vsc-pack \
    --force \
  && code-server --list-extensions --show-versions

# configure for code-server
RUN mkdir -p /home/${NB_USER}/.local/share/code-server/User \
  && chown -R ${NB_USER}:users /home/${NB_USER} \
  && cat <<EOF > /home/${NB_USER}/.local/share/code-server/User/settings.json
{
  "gitlens.showWelcomeOnInstall": false,
  "workbench.colorTheme": "Catppuccin Mocha",
}
EOF

RUN mkdir -p /tmp_home/${NB_USER}/.local/share \
  && mv /home/${NB_USER}/.local/share/code-server /tmp_home/${NB_USER}/.local/share

# set ssh configuration
RUN mkdir -p /run/sshd \
 && chown -R ${NB_USER}:users /etc/ssh \
 && chown -R ${NB_USER}:users /run/sshd \
 && sed -i "/#\?Port/s/^.*$/Port 2222/g" /etc/ssh/sshd_config \
 && sed -i "/#\?PasswordAuthentication/s/^.*$/PasswordAuthentication no/g"
/etc/ssh/sshd_config \
 && sed -i "/#\?PubkeyAuthentication/s/^.*$/PubkeyAuthentication yes/g"
/etc/ssh/sshd_config \
 && rclone_version=v1.65.0 && \
       arch=$(uname -m | sed -E 's/x86_64/amd64/g;s/aarch64/arm64/g') && \
       filename=rclone-${rclone_version}-linux-${arch} && \
       curl -fsSL
https://github.com/rclone/rclone/releases/download/${rclone_version}/${filename}.zip -o
${filename}.zip && \
       unzip ${filename}.zip && mv ${filename}/rclone /usr/local/bin && rm -rf
${filename} ${filename}.zip

# Init mamba
RUN mamba init --system

# init baize-base environment for essential python packages
RUN mamba create -n baize-base -y python \
  && /opt/conda/envs/baize-base/bin/pip install tensorboard \
  && mamba clean --all -y \
  && ln -s /opt/conda/envs/baize-base/bin/tensorboard /usr/local/bin/tensorboard

# prepare baize-runtime-env directory
RUN mkdir -p /opt/baize-runtime-env \
  && chown -R ${NB_USER}:users /opt/baize-runtime-env

ARG APP
ARG PROD_NAME
ARG TARGETOS

COPY out/$TARGETOS/$TARGETARCH/data-loader /usr/local/bin/
COPY out/$TARGETOS/$TARGETARCH/baizectl /usr/local/bin/

RUN chmod +x /usr/local/bin/baizectl /usr/local/bin/data-loader && \
    echo "source /etc/bash_completion" >> /opt/conda/etc/profile.d/conda.sh && \
    echo "source <(baizectl completion bash)" >> /opt/conda/etc/profile.d/conda.sh && \
    echo "source <(kubectl completion bash)" >> /opt/conda/etc/profile.d/conda.sh && \
    echo '[ -f /run/baize-env ] && export $(cat /run/baize-env | xargs)' >>
/opt/conda/etc/profile.d/conda.sh && \
    echo 'alias conda="mamba"' >> /opt/conda/etc/profile.d/conda.sh

USER ${NB_UID}
```

构建你的镜像

```
ARG BASE_IMG=release.daocloud.io/baize/baize-notebook:v0.5.0

FROM $BASE_IMG
USER root

# Do Customization
RUN mamba install -n baize-base -y pytorch torchvision torchaudio cpuonly -c pytorch \
 && mamba install -n baize-base -y tensorflow \
 && mamba clean --all -y

USER ${NB_UID}
```

增加到 Notebook 镜像列表（Helm）

> 注意，必须由平台管理员操作，谨慎变更。

目前，镜像选择器需要通过更新 baize 的 Helm 参数来修改，具体步骤如下：

在 kpanda-global-cluster 全局服务集群的 Helm 应用 列表，找到 baize，进入更新页面，在 YAML 参数中修改 Notebook 镜像：

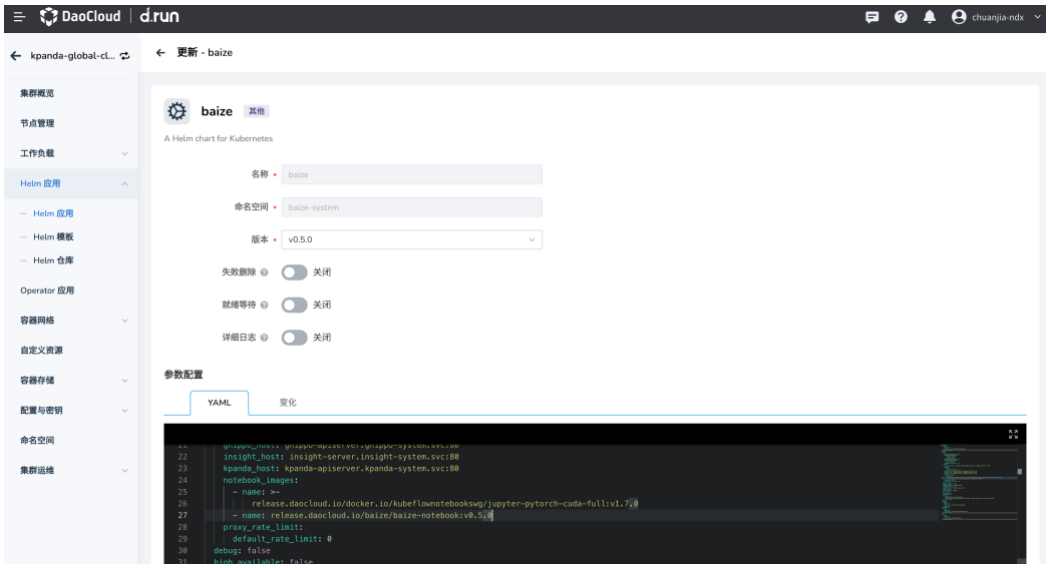

图 125　更新 baize 的 Helm 参数

注意参数修改的路径如下 global.config.notebook_images：

```
...
global:
  ...
  config:
    notebook_images:
      ...
      names: release.daocloud.io/baize/baize-notebook:v0.5.0
      # 在这里增加你的镜像信息
```

更新完成之后，待 Helm 应用重启成功之后，可以在 Notebook 创建界面中的选择镜像看到新的镜像。

（三）增加任务调度器

DCE AI Lab 提供了任务调度器，可以帮助您更好地管理任务，除了提供基础的调度器之外，目前也支持用户自定义调度器。

任务调度器介绍

在 Kubernetes 中，任务调度器负责决定将 Pod 分配到哪个节点上运行。它考虑多种因素，如资源需求、硬件/软件约束、亲和性/反亲和性规则、数据局部性等。

默认调度器是 Kubernetes 集群中的一个核心组件，负责决定将 Pod 分配到哪个节点上运行。让我们深入了解它的工作原理、特性和配置方法。

调度器的工作流程

默认调度器的工作流程可以分为两个主要阶段：过滤（Filtering）和评分（Scoring）。

过滤阶段

调度器会遍历所有节点，排除不满足 Pod 要求的节点，考虑的因素包括：

- 资源需求
- 节点选择器
- 节点亲和性
- 污点和容忍

以上参数，我们可以通过创建任务时的高级配置来设置，如下图所示：

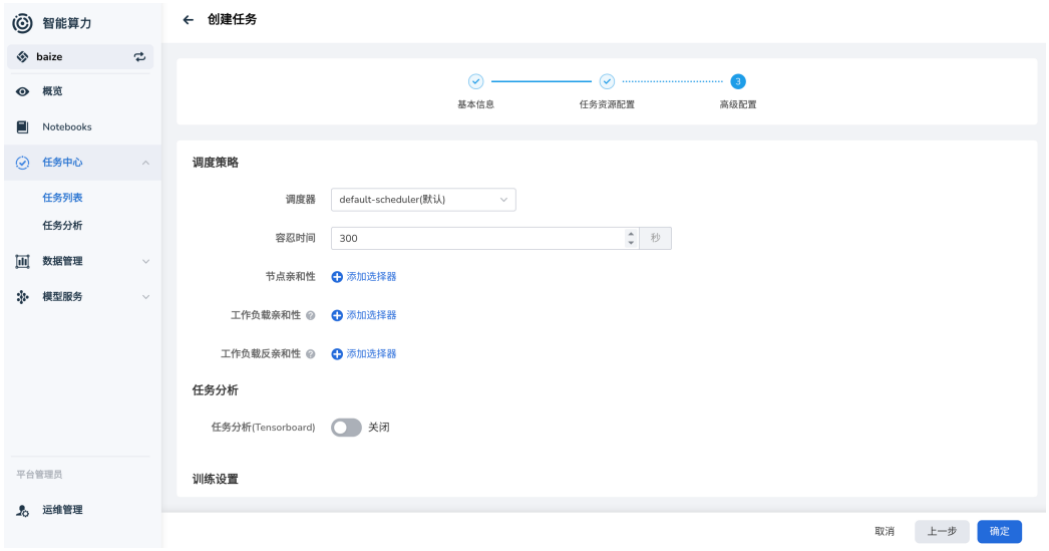

图 126　在任务高级配置中设置调度参数

评分阶段

对通过过滤的节点进行打分，选择得分最高的节点来运行 Pod，考虑因素包括：

- 资源使用率
- Pod 亲和性/反亲和性
- 节点亲和性等

调度器插件

除了基础的一些任务调度能力之外，我们还支持使用 `Scheduler Plugins: Kubernetes SIG Scheduling` 维护的一组调度器插件，包括 `Coscheduling (Gang Scheduling)` 等功能。

部署调度器插件

在工作集群中部署第二调度器插件，请参考 DCE「部署第二调度器插件」文档。

在 AI Lab 中启用调度器插件

> 增加调度器插件若操作不当，可能会影响到整个集群的稳定性，建议在测试环境中进行测试；或者联系我们的技术支持团队。

注意，如果希望在训练任务中使用更多的调度器插件，需要事先手工在工作集群中成功安装，然后在集群中部署 `baize-agent` 时，增加对应的调度器插件配置。

通过容器管理提供的界面 **Helm 应用** 管理能力，可以方便地在集群中部署调度器插件，如下图所示：

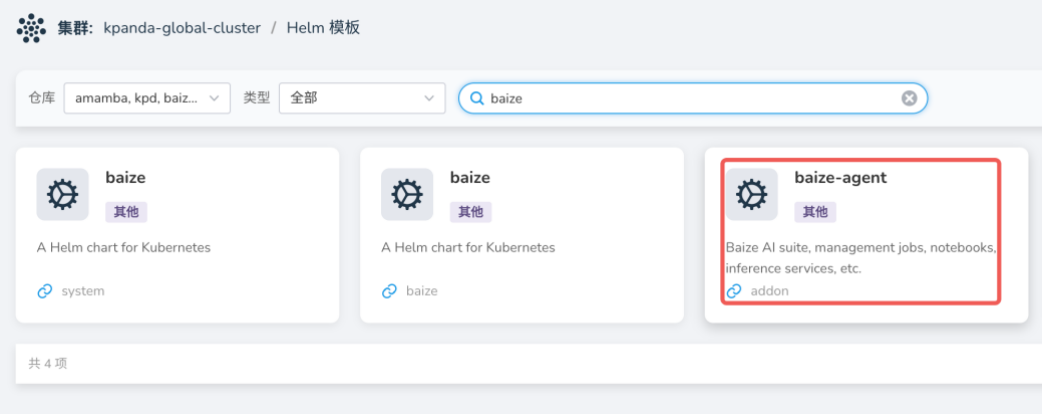

图 127　在 Helm 应用中部署调度器插件

然后，在右上角点击 **安装**，（若已部署了 `baize-agent`，可以到 Helm 应用列表去更新），根据如下图所示的配置，增加调度器。

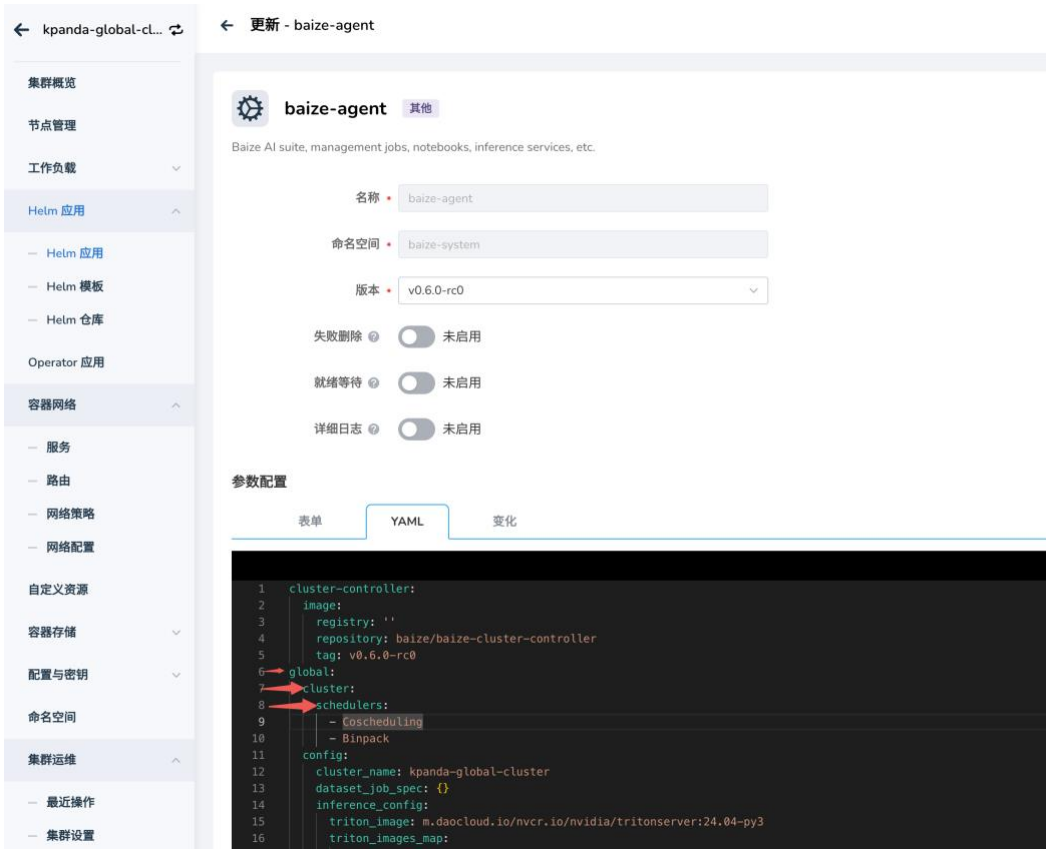

图 128　配置 baize-agent 的调度器参数

注意调度器的参数层级，添加完成后，点击 **确定** 即可。注意以后在更新 `baize-agent` 时，不要遗漏这个配置。

在创建任务时指定调度器

当您在集群中成功部署了对应的调度器插件，并且在 `baize-agent` 也正确增加了对应的调度器配置后，可以在创建任务时，指定调度器。

一切正常的情况下，您可以在调度器下拉框中看到您部署的调度器插件。

图 129　在创建任务时选择调度器

以上，就是我们在 AI Lab 中，为任务增加调度器选项的配置使用说明。

（四）拓扑感知调度训练任务

在人工智能和机器学习领域，模型训练任务，如大型语言模型 LLM 的分布式训练，对计算资源和网络性能提出了极高要求。这些任务通常涉及多个 Pod 之间的频繁通信，例如梯度聚合和数据交换，网络延迟和带宽利用率直接影响训练效率。

在现代数据中心中，复杂的网络拓扑结构（如节点、机架、区块）使得 Pod 的物理位置对通信性能至关重要。Kubernetes 默认调度器主要根据 CPU 和内存等资源进行分配，缺乏对网络拓扑的感知，可能导致 Pod 分布在网络上较远的节点，增加通信开销并延长训练时间。

Kueue 是 Kubernetes 原生的作业队列系统，通过配额管理和队列机制优化批处理作业的资源分配。Baize 基于其拓扑感知调度（Topology Aware Scheduling, TAS）功能，利用数据中心的拓扑信息，将 Pod 智能地调度到网络拓扑上更靠近的节点上，从而减少网络跳跃、提升通信吞吐量。TAS 特别适合模型训练场景，能够显著降低分布式训练中的网络延迟，加速梯度同步和数据传输，为 AI/ML 工作负载提供高效、可靠的调度支持。

本节介绍如何使用拓扑感知调度能力，详细步骤如下：

1. 手动创建 Topology CR

前面提到，数据中心的组织单元具有层次结构，运行在同一组织单元内的 Pod 之间的网络带宽优于运行在不同单元的 Pod。

我们可以通过节点标签来表示数据中心内节点的层次结构，相同层级的节点可以组成一个超节点，一个超节点表示一个网络拓扑性能域，多个超节点通过层级连接，形成树状结构。

Baize 暂不支持通过界面创建节点拓扑，需用户手动在 GPU 集群创建 Topology CR。Yaml 示例如下：

```
apiVersion: kueue.x-k8s.io/v1alpha1
kind: Topology
metadata:
  name: default
spec:
  levels:
    - nodeLabel: shanghai-cube/super-node # 一个超节点表示一个网络拓扑性能域，多个超节点通过层级连接，形成树状结构。

---
apiVersion: kueue.x-k8s.io/v1beta1
kind: ResourceFlavor
metadata:
  name: tas-flavor
spec:
  nodeLabels:
    metax-tech.com/driver.ready: 'true'
  topologyName: default
```

2. 创建资源池

操作流程参考创建资源池，请选择需要调度的节点和已创建好的节点拓扑。

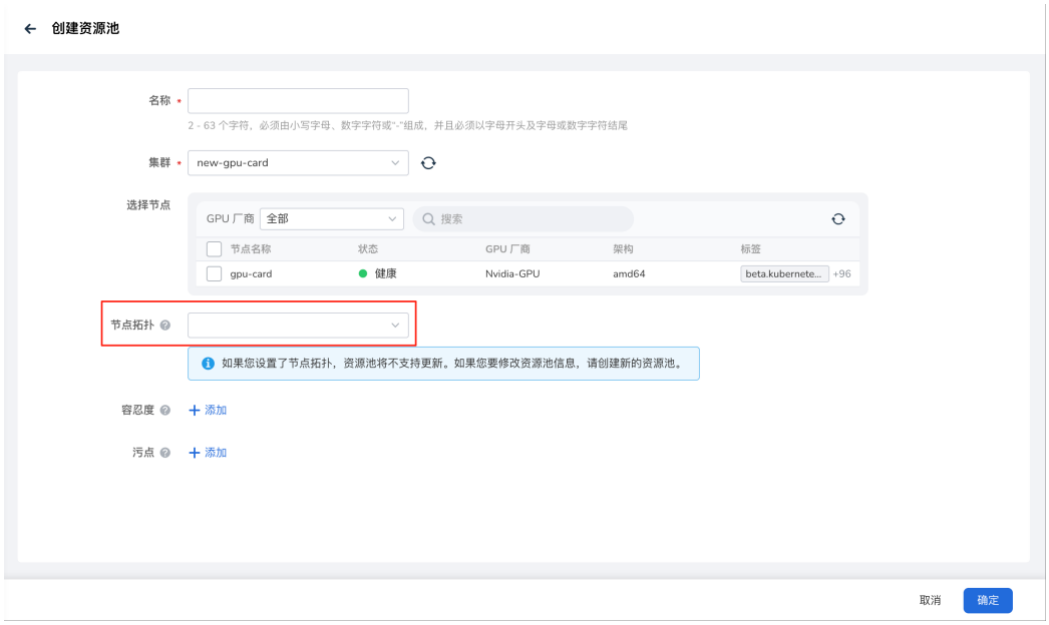

图 130 创建带节点拓扑的资源池

3. 创建队列

操作流程参考创建队列，选择上文创建好的资源池。

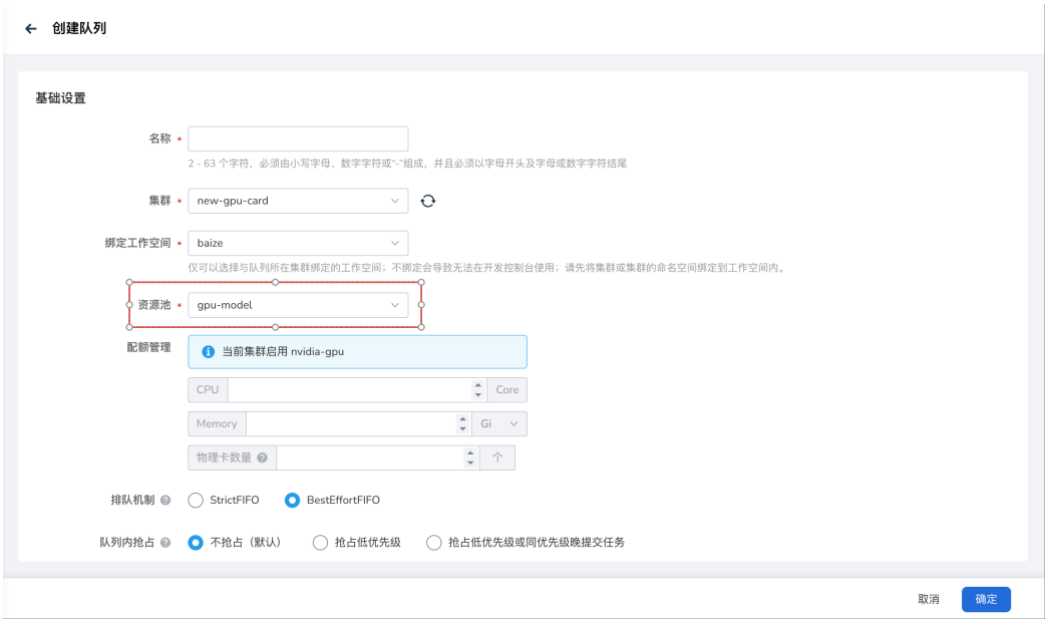

图 131　创建关联资源池的队列

4. 创建训练任务

以上步骤基本完成 TAS 配置工作，接下来就可以在 PodTemplate 级别创建具有 TAS 配置的训练任务，操作流程参考创建训练任务。

训练任务中选择的队列，其资源池包含拓扑信息，会根据 TAS 配置自动化调度。在同一拓扑域内的节点上调度相关联的所有 Pod，是一种偏好而不是要求，默认匹配到拓扑的最低层级，如果 PodSet 无法适应给定的拓扑域，则考虑下一个拓扑级别。

至此，完成训练任务基于拓扑感知调度的所有操作。

（五）Checkpoint 机制及用法

在深度学习的实际场景中，模型训练一般都会持续一段时间，这对分布式训练任务的稳定性和效率提出了更高的要求。而且，在实际训练的过程中，异常中断会导致训练过程中的模型状态丢失，需要重新开始训练，这不仅浪费了时间和资源，这在 LLM 训练中尤为明显，而且也会影响模型的训练效果。

能够在训练过程中保存模型的状态，以便在训练过程中出现异常时能够恢复模型状态，变得至关重要。Checkpoint 就是目前主流的解决方案，本节将介绍 Checkpoint 机制的基本概念和在 PyTorch 和 TensorFlow 中的使用方法。

什么是 Checkpoint？

Checkpoint 是在模型训练过程中保存模型状态的机制。通过定期保存 Checkpoint，可以在以下情况下恢复模型：

- 训练过程中断（如系统崩溃或手动中断）

- 需要在某个训练阶段进行评估
- 希望在不同的实验中复用模型

PyTorch

在 PyTorch 中，`torch.save` 和 `torch.load` 是用于保存和加载模型的基本函数。

PyTorch 保存 Checkpoint

在 PyTorch 中，通常使用 `state_dict` 保存模型的参数。以下是一个简单的示例：

```
import torch
import torch.nn as nn

# 假设我们有一个简单的神经网络
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(10, 2)

    def forward(self, x):
        return self.fc(x)

# 初始化模型和优化器
model = SimpleModel()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# 训练模型...
# 保存 Checkpoint
checkpoint_path = 'model_checkpoint.pth'
torch.save({
    'epoch': 10,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': 0.02,
}, checkpoint_path)
```

PyTorch 恢复 Checkpoint

加载模型时，需要恢复模型参数和优化器状态，并继续训练或推理：

```
# 恢复 Checkpoint
checkpoint = torch.load('model_checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

# 继续训练或推理...
```

- `model_state_dict`：模型参数
- `optimizer_state_dict`：优化器状态
- `epoch`：当前训练轮数
- `loss`：损失值
- `learning_rate`：学习率
- `best_accuracy`：最佳准确率

TensorFlow

TensorFlow 提供了 `tf.train.Checkpoint` 类来管理模型和优化器的保存和恢复。

TensorFlow 保存 Checkpoint

以下是一个在 TensorFlow 中保存 Checkpoint 的示例：

```
import tensorflow as tf

# 假设我们有一个简单的模型
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2, input_shape=(10,))
])
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# 定义 Checkpoint
checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
checkpoint_dir = './checkpoints'
checkpoint_prefix = f'{checkpoint_dir}/ckpt'

# 训练模型...
# 保存 Checkpoint
checkpoint.save(file_prefix=checkpoint_prefix)
```

> 使用 DCE AI Lab 的用户，可以直接将高性能存储挂载为 Checkpoint 目录，以提高 Checkpoint 保存和恢复的速度。

TensorFlow 恢复 Checkpoint

加载 Checkpoint 并恢复模型和优化器状态：

```
# 恢复 Checkpoint
latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
checkpoint.restore(latest_checkpoint)

# 继续训练或推理...
```

TensorFlow 在分布式训练的 Checkpoint 管理

TensorFlow 在分布式训练中管理 Checkpoint 的主要方法如下：

- 使用 `tf.train.Checkpoint` 和 `tf.train.CheckpointManager`：

```
checkpoint = tf.train.Checkpoint(model=model, optimizer=optimizer)
manager = tf.train.CheckpointManager(checkpoint, directory='/tmp/model', max_to_keep=3)
```

- 在分布式策略中保存 Checkpoint：

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    checkpoint = tf.train.Checkpoint(model=model, optimizer=optimizer)
    manager = tf.train.CheckpointManager(checkpoint, directory='/tmp/model',
max_to_keep=3)
```

- 只在主节点（chief worker）保存 Checkpoint：

```
if strategy.cluster_resolver.task_type == 'chief':
```

```
    manager.save()
```

- 使用 MultiWorkerMirroredStrategy 时的特殊处理：

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
with strategy.scope():
    # 模型定义
    ...
    checkpoint = tf.train.Checkpoint(model=model, optimizer=optimizer)
    manager = tf.train.CheckpointManager(checkpoint, '/tmp/model', max_to_keep=3)

def _chief_worker(task_type, task_id):
    return task_type is None or task_type == 'chief' or (task_type == 'worker' and
task_id == 0)

if _chief_worker(strategy.cluster_resolver.task_type,
strategy.cluster_resolver.task_id):
    manager.save()
```

- 使用分布式文件系统：确保所有工作节点都能访问到同一个 Checkpoint 目录，通常使用分布式文件系统如 HDFS 或 GCS。
- 异步保存：使用 `tf.keras.callbacks.ModelCheckpoint` 并设置 `save_freq` 参数可以在训练过程中异步保存 Checkpoint。
- Checkpoint 恢复：

```
status = checkpoint.restore(manager.latest_checkpoint)
status.assert_consumed()  # 确保所有变量都被恢复
```

- 性能优化：
  - 使用 `tf.train.experimental.enable_mixed_precision_graph_rewrite()` 启用混合精度训练
  - 调整保存频率，避免过于频繁的 I/O 操作
  - 考虑使用 `tf.saved_model.save()` 保存整个模型，而不仅仅是权重

注意事项

1. **定期保存**：根据训练时间和资源消耗，决定合适的保存频率。如每个 epoch 或每隔一定的训练步数。

2. **保存多个 Checkpoint**：保留最新的几个 Checkpoint 以防止文件损坏或不适用的情况。

3. **记录元数据**：在 Checkpoint 中保存额外的信息，如 epoch 数、损失值等，以便更好地恢复训练状态。

4. **使用版本控制**：保存不同实验的 Checkpoint，便于对比和复用。

5. **验证和测试**：在训练的不同阶段使用 Checkpoint 进行验证和测试，确保模型性能和稳定性。

Checkpoint 机制在深度学习训练中起到了关键作用。通过合理使用 PyTorch 和 TensorFlow 中的 Checkpoint 功能，可以有效提高训练的可靠性和效率。

（六）提交 DeepSpeed 训练任务

根据 DeepSpeed 官方文档，我们推荐使用修改代码的方式实现。

即使用 `deepspeed.init_distributed()` 代替 `torch.distributed.init_process_group(...)`。然后运行命令使用 `torchrun`，提交为 Pytorch 分布式任务，即可运行 DeepSpeed 任务。

是的，你可以使用 `torchrun` 运行你的 DeepSpeed 训练脚本。`torchrun` 是 PyTorch 提供的一个实用工具，用于分布式训练。你可以结合 `torchrun` 和 DeepSpeed API 来启动你的训练任务。

以下是一个使用 `torchrun` 运行 DeepSpeed 训练脚本的示例：

1. 编写训练脚本：

```
import torch
import deepspeed
from torch.utils.data import DataLoader

# 模型和数据加载
model = YourModel()
train_dataset = YourDataset()
train_dataloader = DataLoader(train_dataset, batch_size=32)

# 配置文件路径
deepspeed_config = "deepspeed_config.json"

# 创建 DeepSpeed 训练引擎
model_engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    model_parameters=model.parameters(),
    config_params=deepspeed_config
)

# 训练循环
for batch in train_dataloader:
    loss = model_engine(batch)
    model_engine.backward(loss)
    model_engine.step()
```

2. 创建 DeepSpeed 配置文件：

```
{
  "train_batch_size": 32,
  "gradient_accumulation_steps": 1,
  "fp16": {
    "enabled": true,
    "loss_scale": 0
  },
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 0.00015,
      "betas": [0.9, 0.999],
      "eps": 1e-08,
```

```
      "weight_decay": 0
    }
  }
}
```

3. 使用 `torchrun` 或者 `baizectl` 运行训练脚本：

```
torchrun train.py
```

通过这种方式，你可以结合 PyTorch 的分布式训练功能和 DeepSpeed 的优化技术，从而实现更高效的训练。您可以在 Notebook 中，使用 `baizectl` 提交命令：

```
baizectl job submit --pytorch --workers 2 -- torchrun train.py
```

（七）使用 AI Lab 微调 ChatGLM3

本文以 `ChatGLM3` 模型为例，演示如何在 DCE AI Lab 中使用 LoRA（Low-Rank Adaptation，低秩自适应）微调 ChatGLM3 模型。Demo 程序来自 ChatGLM3 官方案例。

微调的大致流程为：

数据准备 → 模型调试（Notebook） → 任务训练（UI 或 baizectl） → 推理服务

其中「模型调试」阶段还需要先完成环境准备。

环境依赖

- GPU 显存至少 20GB，推荐使用 RTX4090、NVIDIA A/H 系列显卡
- 可用磁盘空间至少 200GB
- CPU 至少 8 核，推荐 16 核
- 内存 64GB，推荐 128GB

> 在开始体验之前，请检查 DCE 以及 AI Lab 部署正确，GPU 队列资源初始化成功，且算力资源充足。

数据准备

利用 DCE AI Lab 提供的数据集管理功能，快速将微调大模型所需的数据进行预热及持久化，减少因为准备数据导致的 GPU 资源占用，提高资源利用效率。

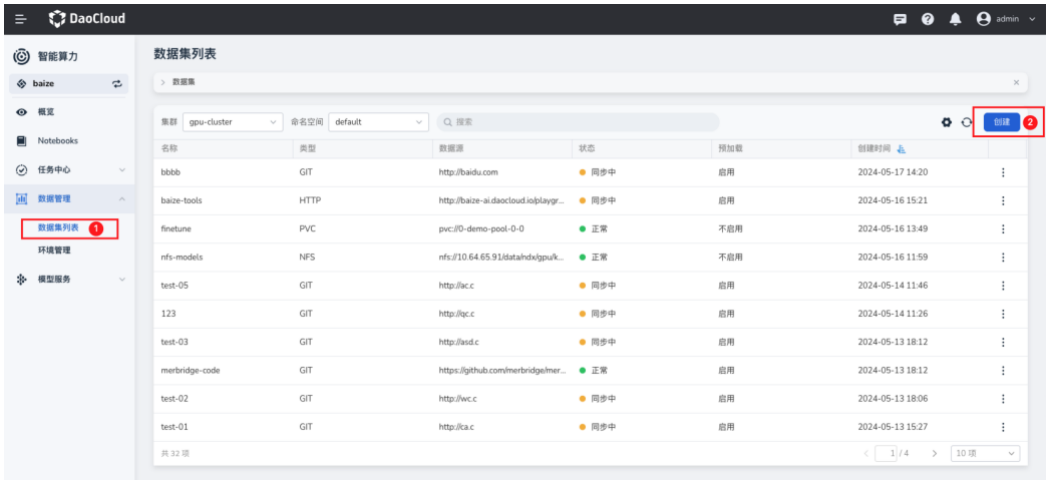

图 132　数据集列表

在数据集列表页面，创建需要的数据资源，这些资源包含了 ChatGLM3 代码，也可以是数据文件，所有这些数据都可以通过数据集列表来统一管理。

代码及模型文件

ChatGLM3 是智谱 AI 和清华大学 KEG 实验室联合发布的对话预训练模型。先拉取 ChatGLM3 代码仓库，下载预训练模型，用于后续的微调任务。

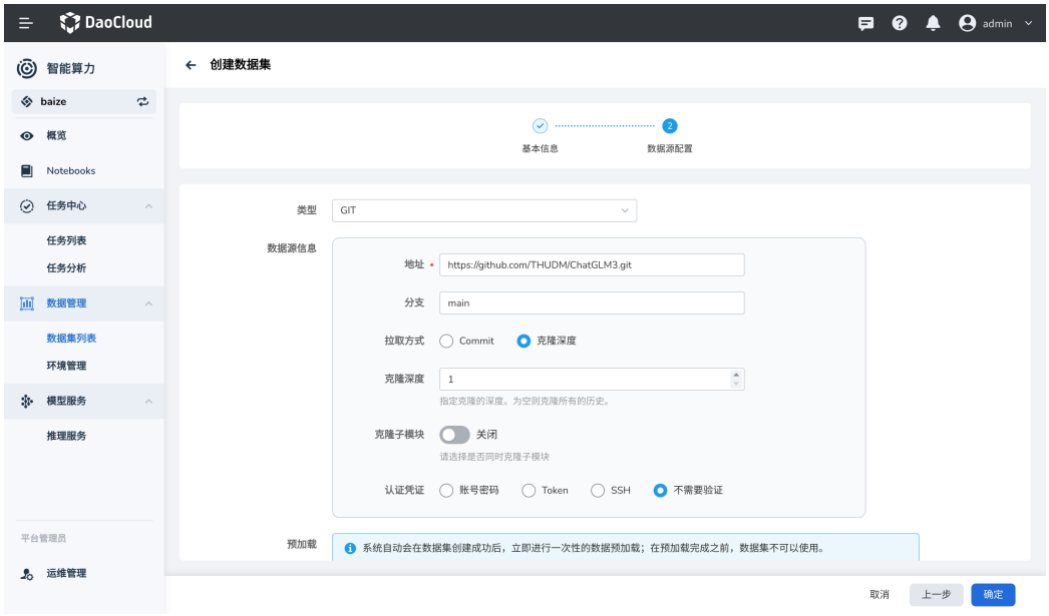

图 133　纳管 ChatGLM3 代码与模型

DCE AI Lab 会在后台进行全自动数据预热，以便后续的任务能够快速访问数据。

AdvertiseGen 数据集

国内数据可以从 Tsinghua Cloud 直接获取，这里使用 HTTP 的数据源方式。注意创建完成后，需要等待数据集预热完成，一般很快，根据您的网络情况而定。

微调输出数据

同时，您需要准备一个空的数据集，用于存放微调任务完成后输出的模型文件，这里创建一个空的数据集，以 PVC 为例。

> 注意需要使用支持 ReadWriteMany 的存储类型，以便后续的任务能够快速访问数据。

环境准备

对于模型开发者来说，准备模型开发需要的 Python 环境依赖是非常重要的，传统做法将环境依赖直接打包到开发工具的镜像中，或者直接在本地环境中安装，但是这样做会导致环境依赖的不一致，而且不利于环境的管理和依赖更新及同步。

DCE AI Lab 提供了环境管理的能力，将 Python 环境依赖包管理和开发工具、任务镜像等进行解耦，解决了依赖管理混乱，环境不一致等问题。

这里使用 DCE AI Lab 提供的环境管理功能，创建 ChatGLM3 微调所需的环境，以备后续使用。

> 1. ChatGLM 仓库内有 requirements.txt 文件，里面包含了 ChatGLM3 微调所需的环境依赖。
> 2. 本次微调没有用到 deepspeed 和 mpi4py 包，建议从 requirements.txt 文件中将其注释掉，否则可能出现包编译不通过的情况。

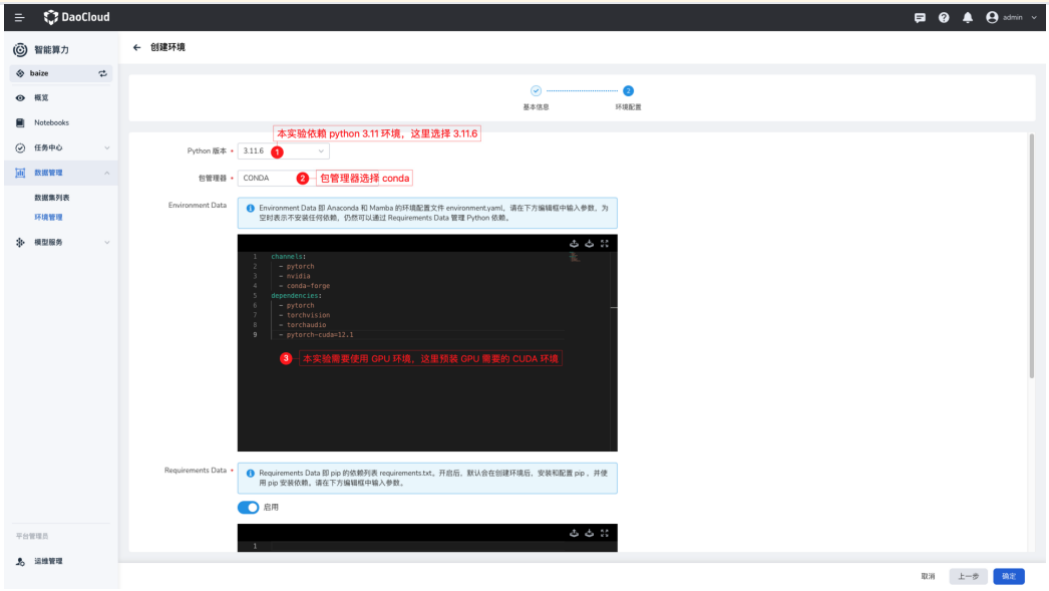

图 134　创建微调所需的环境

在环境管理列表，您可以快速创建一个 Python 环境，并通过简单的表单配置来完成环境的创建；这里需要一个 Python 3.11.x 环境。

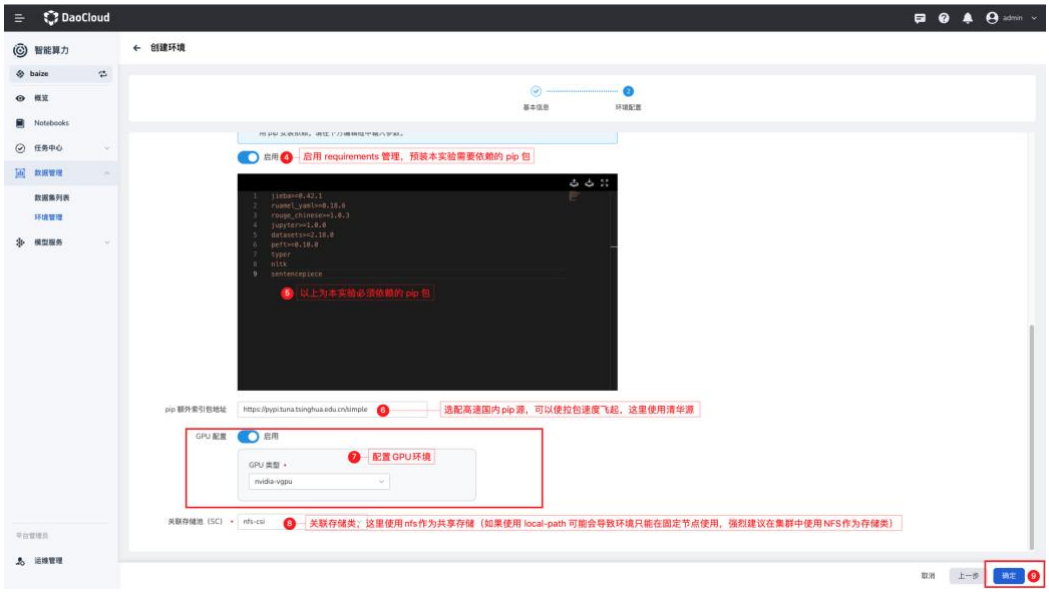

图 135　配置环境

因为本实验需要使用 CUDA，所以在这里需要配置 GPU 资源，用于预热需要资源的依赖库。

创建环境，需要去下载一系列的 Python 依赖，根据您的实际位置不同，可能会有不同的下载速度，这里使用了国内的镜像加速，可以加快下载速度。

使用 Notebook 作为 IDE

DCE AI Lab 提供了 Notebook 作为 IDE 的功能，可以让用户在浏览器中直接编写代码，运行代码，查看代码运行结果，非常适合于数据分析、机器学习、深度学习等领域的开发。您可以使用 AI Lab 提供的 JupyterLab Notebook 来进行 ChatGLM3 的微调任务。

创建 JupyterLab Notebook

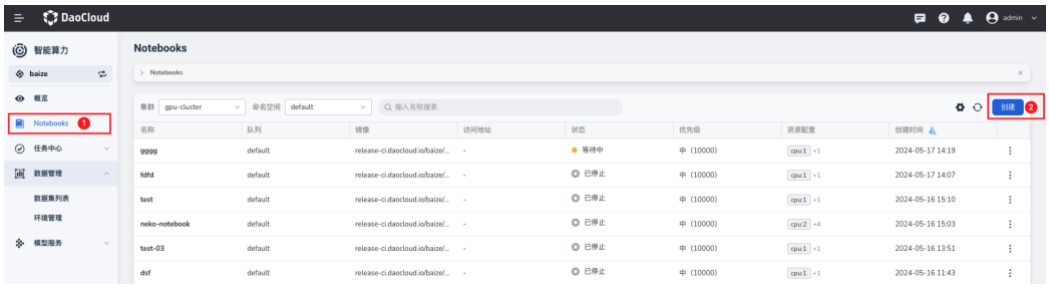

图 136　创建 Notebook

在 Notebook 列表中，可以根据页面操作指引，创建一个 Notebook。注意您需要根据前文提到的资源要求来配置对应的 Notebook 资源参数，避免后续因为资源问题，影响微调过程。

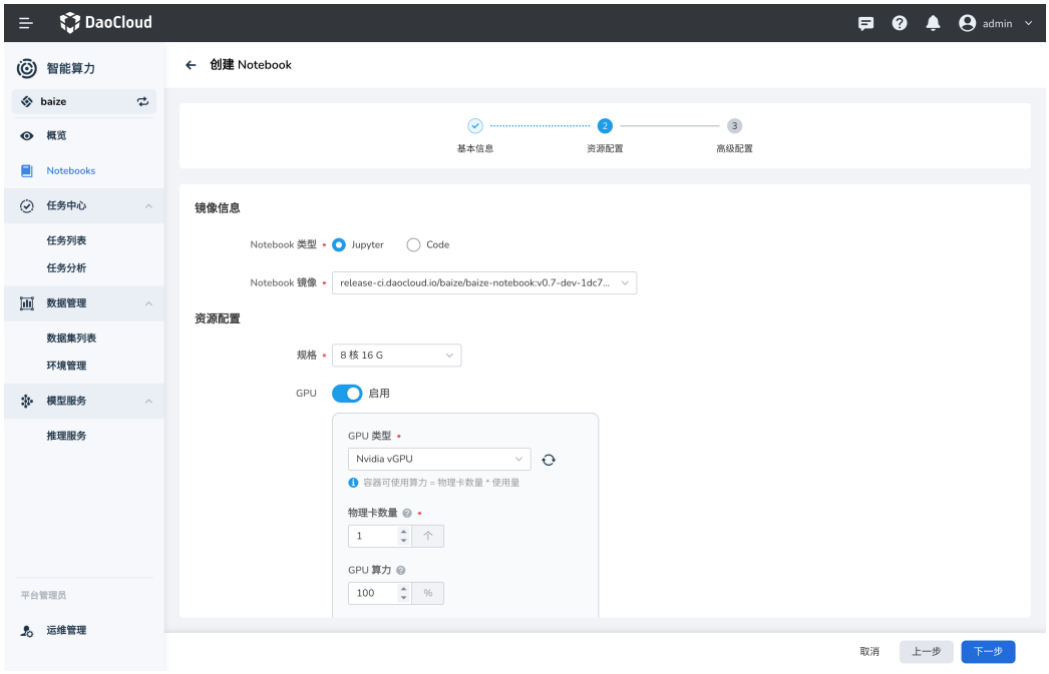

图 137　配置 Notebook 资源

在创建 Notebook 时，可以将之前预加载的模型代码数据集和环境，直接挂载到 Notebook 中，极大节省了数据准备的时间。

挂载数据集和代码

注意：ChatGLM3 的代码文件挂载到了 `/home/jovyan/ChatGLM3` 目录下，同时您也需要将 `AdvertiseGen` 数据集挂载到 `/home/jovyan/ChatGLM3/finetune_demo/data/AdvertiseGen` 目录下，以便后续的微调任务能够访问数据。

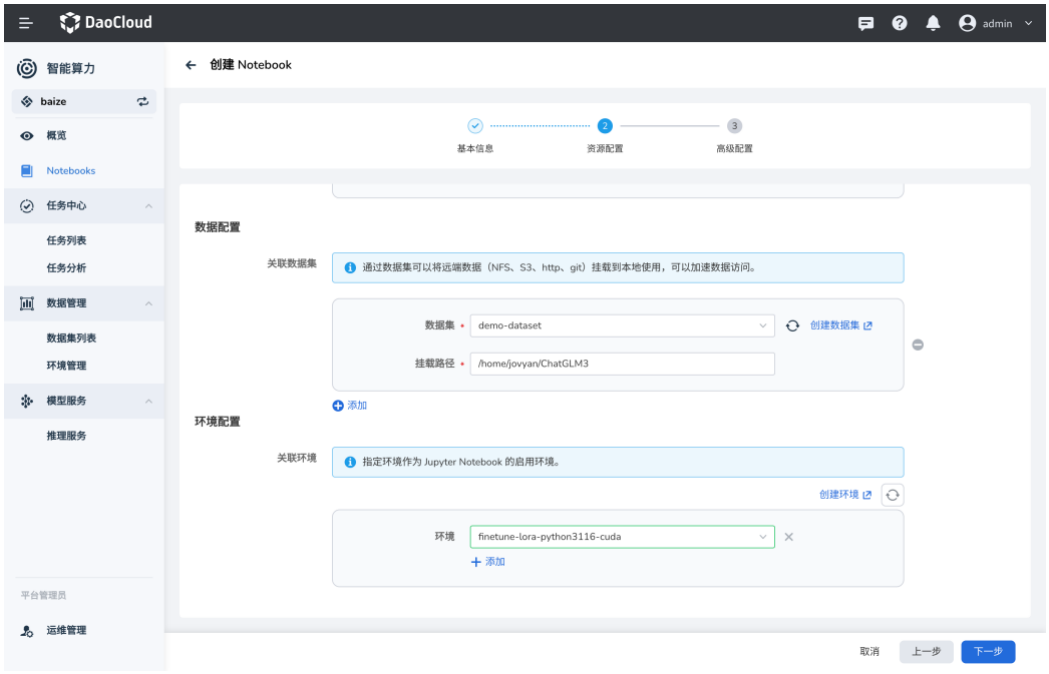

图 138　挂载代码与数据集

挂载 PVC 到模型输出文件夹

本次使用的模型输出位置在 `/home/jovyan/ChatGLM3/finetune_demo/output` 目录下，可以将之前创建的 `PVC` 数据集挂载到这个目录下，这样训练输出的模型就可以保存到数据集中，后续模型推理等任务可以直接访问。

创建完成后，可以看到 Notebook 的界面，您可以直接在 Notebook 中编写代码，运行代码，查看代码运行结果。

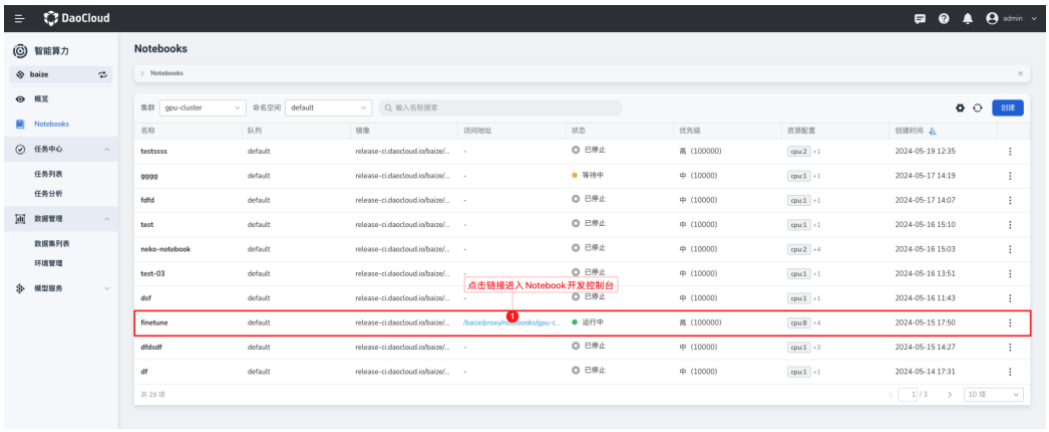

图 139　挂载 PVC 到模型输出目录

微调 ChatGLM3

当您进入到 Notebook 中后，可以在 Notebook 侧边栏会发现有一个 `File Browser` 的选项，可以看到之前挂载的数据集和代码，在这里找到 ChatGLM3 的文件夹。

您可以看到 ChatGLM3 的微调代码在 `finetune_demo` 文件夹中，这里可以直接打开 `lora_finetune.ipynb` 文件，这是 ChatGLM3 的微调代码。

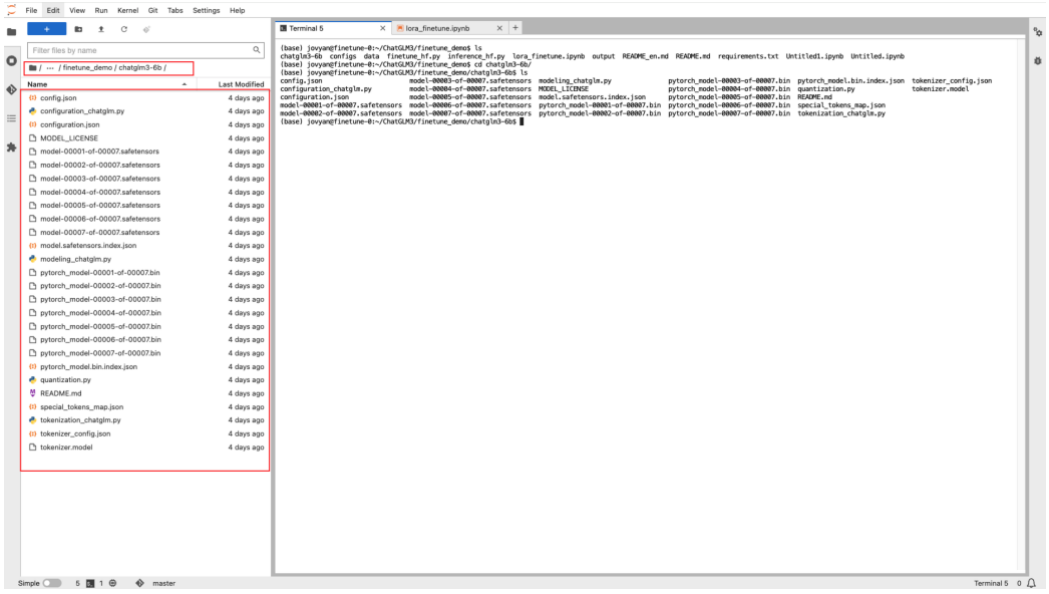

图 140　在文件浏览器中找到微调代码

首先，根据 `README.md` 的说明，您可以了解到整个微调的过程，建议先阅读一遍，确保基础的环境依赖和数据准备工作都已经完成。

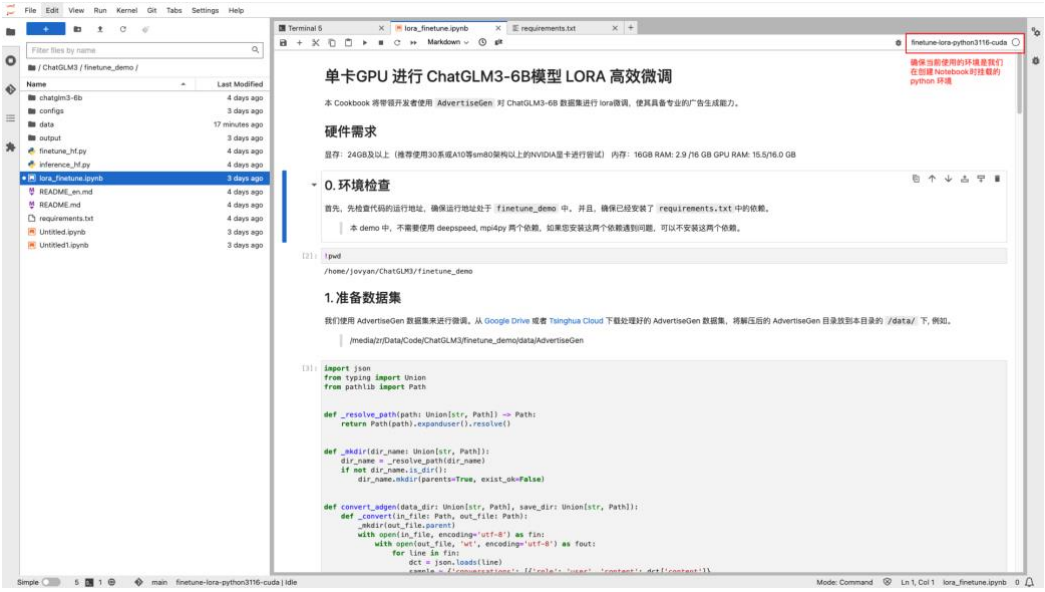

图 141　阅读 README 说明

打开终端，并使用 `conda` 切换到您提前预热的环境中，此环境与 JupyterLab Kernel 保持一致，以便后续的代码运行。

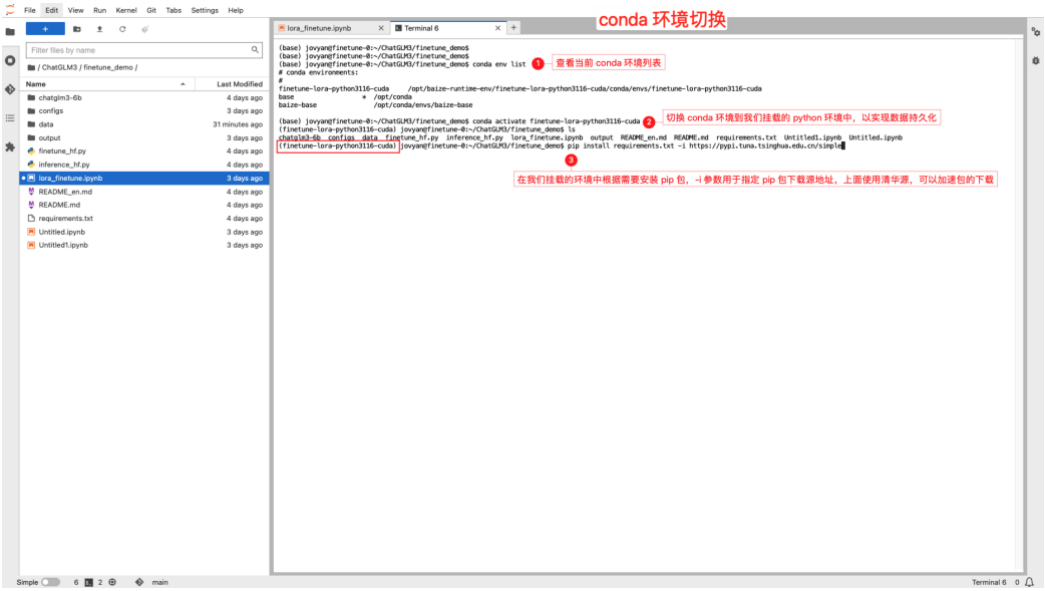

图 142　在终端中切换环境

数据预处理

首先，您需要将 `AdvertiseGen` 数据集进行预处理，对数据进行标准化处理，使其符合 `Lora` 预训练的标准格式要求；这里将处理后的数据保存到 `AdvertiseGen_fix` 文件夹中。

```
import json
```

```
from typing import Union
from pathlib import Path

def _resolve_path(path: Union[str, Path]) -> Path:
    return Path(path).expanduser().resolve()

def _mkdir(dir_name: Union[str, Path]):
    dir_name = _resolve_path(dir_name)
    if not dir_name.is_dir():
        dir_name.mkdir(parents=True, exist_ok=False)

def convert_adgen(data_dir: Union[str, Path], save_dir: Union[str, Path]):
    def _convert(in_file: Path, out_file: Path):
        _mkdir(out_file.parent)
        with open(in_file, encoding='utf-8') as fin:
            with open(out_file, 'wt', encoding='utf-8') as fout:
                for line in fin:
                    dct = json.loads(line)
                    sample = {'conversations': [{'role': 'user', 'content':
dct['content']},
                                                {'role': 'assistant', 'content':
dct['summary']}]}
                    fout.write(json.dumps(sample, ensure_ascii=False) + '\n')

    data_dir = _resolve_path(data_dir)
    save_dir = _resolve_path(save_dir)

    train_file = data_dir / 'train.json'
    if train_file.is_file():
        out_file = save_dir / train_file.relative_to(data_dir)
        _convert(train_file, out_file)

    dev_file = data_dir / 'dev.json'
    if dev_file.is_file():
        out_file = save_dir / dev_file.relative_to(data_dir)
        _convert(dev_file, out_file)

convert_adgen('data/AdvertiseGen', 'data/AdvertiseGen_fix')
```

为了节省调试的时间，您可以将 `/home/jovyan/ChatGLM3/finetune_demo/data/AdvertiseGen_fix/dev.json` 中的数据量缩减到 50 条，这里的数据是 JSON 格式，处理起来也是比较方便的。

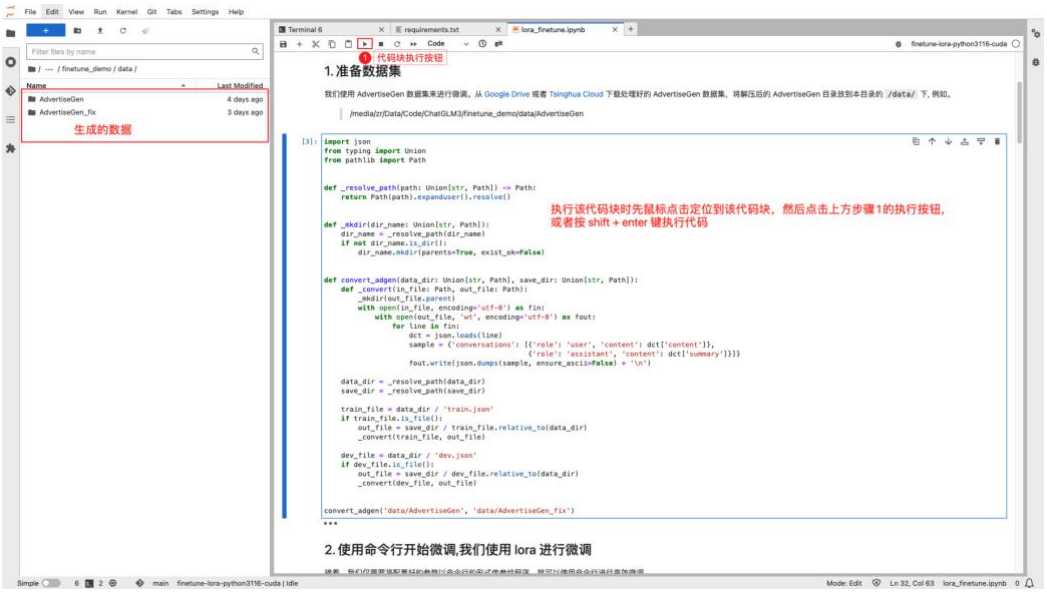

图 143　数据预处理代码与数据

本地 LoRA 微调测试

完成数据的预处理之后，基本上您就可以直接微调测试了，可以在 `/home/jovyan/ChatGLM3/finetune_demo/configs/lora.yaml` 文件中配置微调的参数，一般需要关注的参数基本如下：

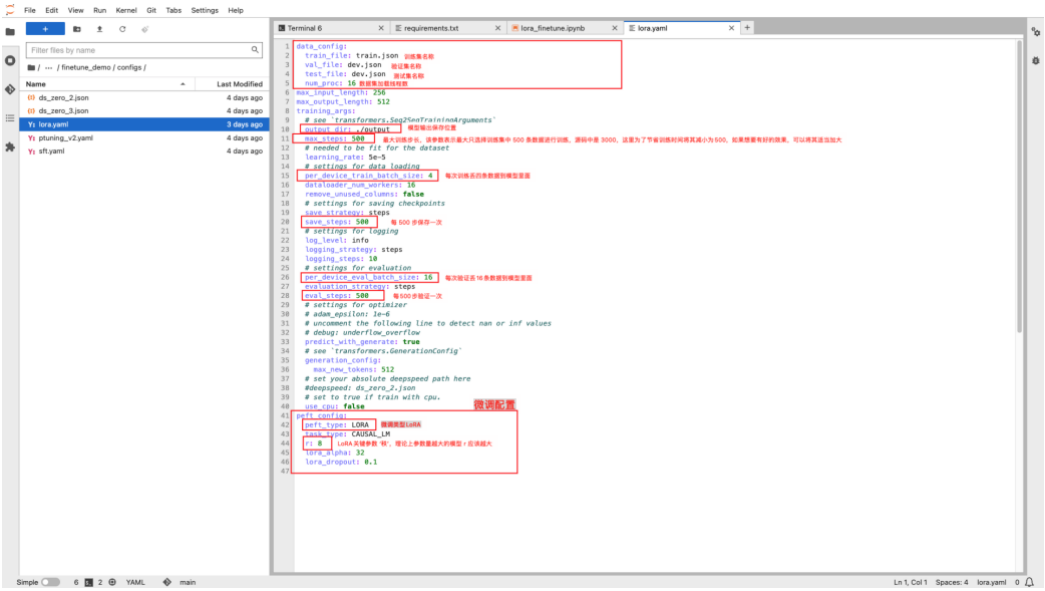

图 144　需要关注的微调参数

新开一个终端窗口，使用如下命令即可进行本地微调测试，请确保参数配置和路径正确：

```
!CUDA_VISIBLE_DEVICES=0 NCCL_P2P_DISABLE="1" NCCL_IB_DISABLE="1" python finetune_hf.py
data/AdvertiseGen_fix  ./chatglm3-6b  configs/lora.yaml
```

在这条命令中，

- `finetune_hf.py` 是 ChatGLM3 代码中的微调脚本
- `data/AdvertiseGen_fix` 是您预处理后的数据集
- `./chatglm3-6b` 是您预训练模型的路径
- `configs/lora.yaml` 是微调的配置文件

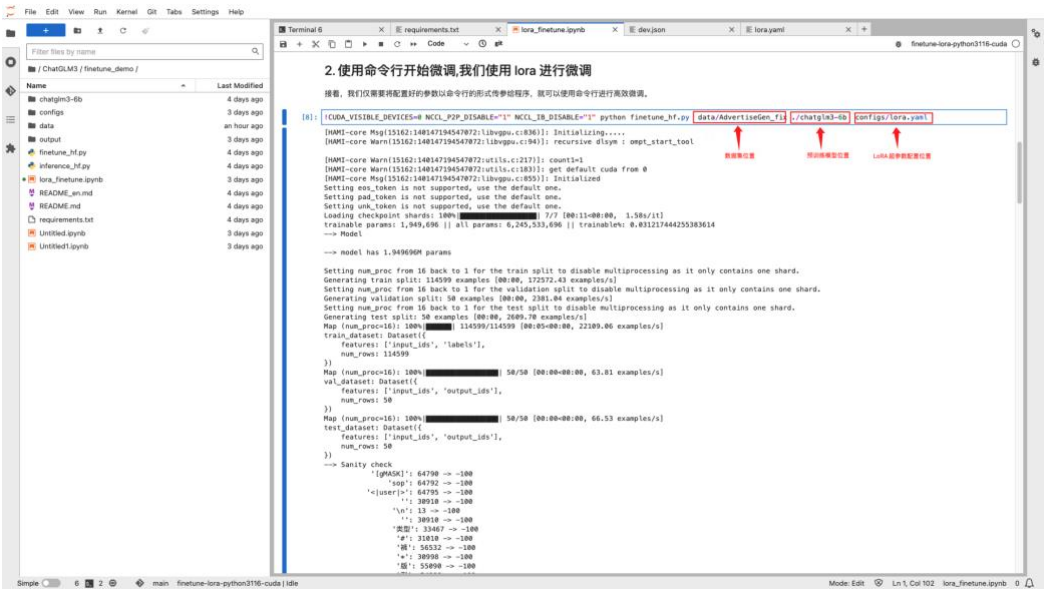

图 145　运行微调命令

微调过程中可以使用 `nvidia-smi` 命令查看 GPU 显存使用情况：

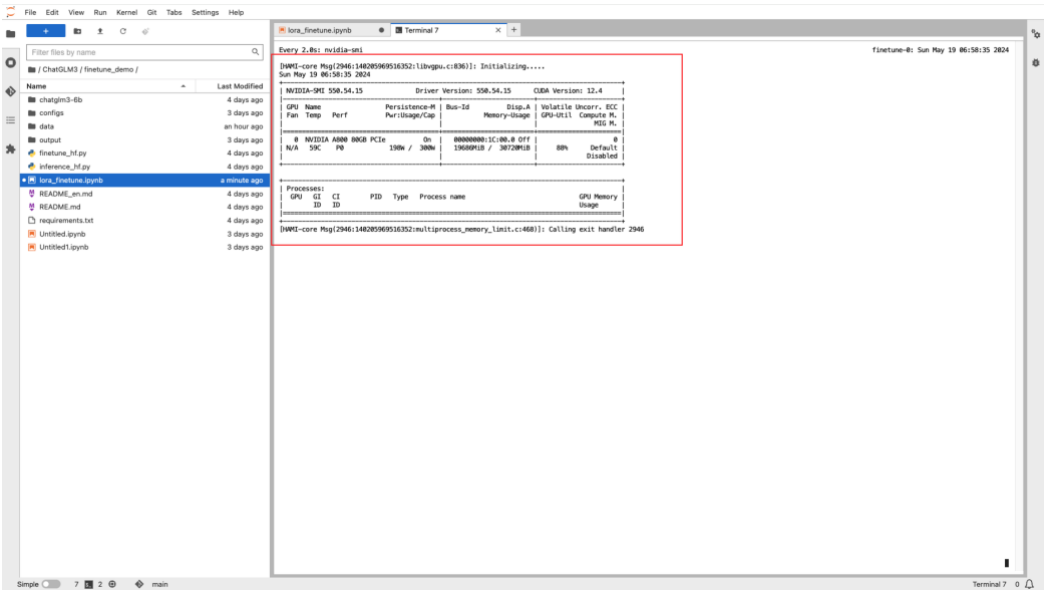

图 146　查看 GPU 显存使用情况

在微调完成后，在 `finetune_demo` 目录下会生成一个 `output` 目录，里面包含了微调的模型文件，这样微调的模型文件就直接保存到您之前创建的 `PVC` 数据集中了。

微调任务提交

在本地微调测试完成后，确保您的代码和数据没有问题，接下来可以将微调任务提交到 AI Lab 中，进行大规模的训练和微调任务。

这也是推荐的模型开发和微调流程，先在本地进行微调测试，确保代码和数据没有问题。

使用界面提交微调任务

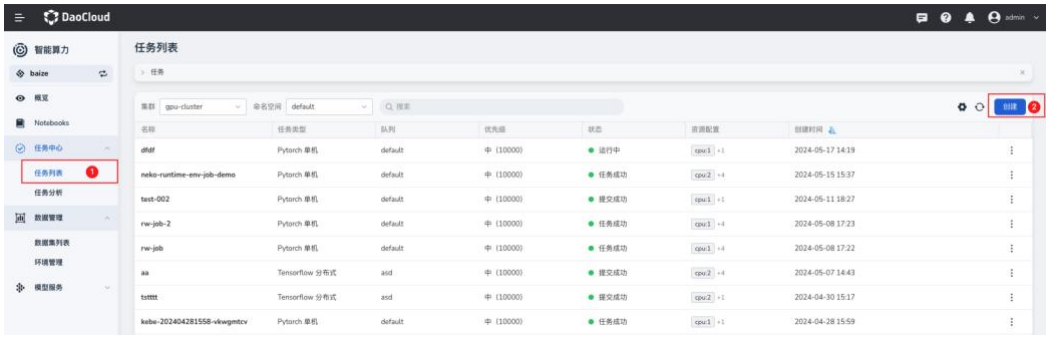

图 147　训练任务列表

这里使用 Pytorch 来创建微调任务，根据您的实际情况，选择需要使用哪个集群的资源，注意需要满足前面资源准备中提及的资源要求。

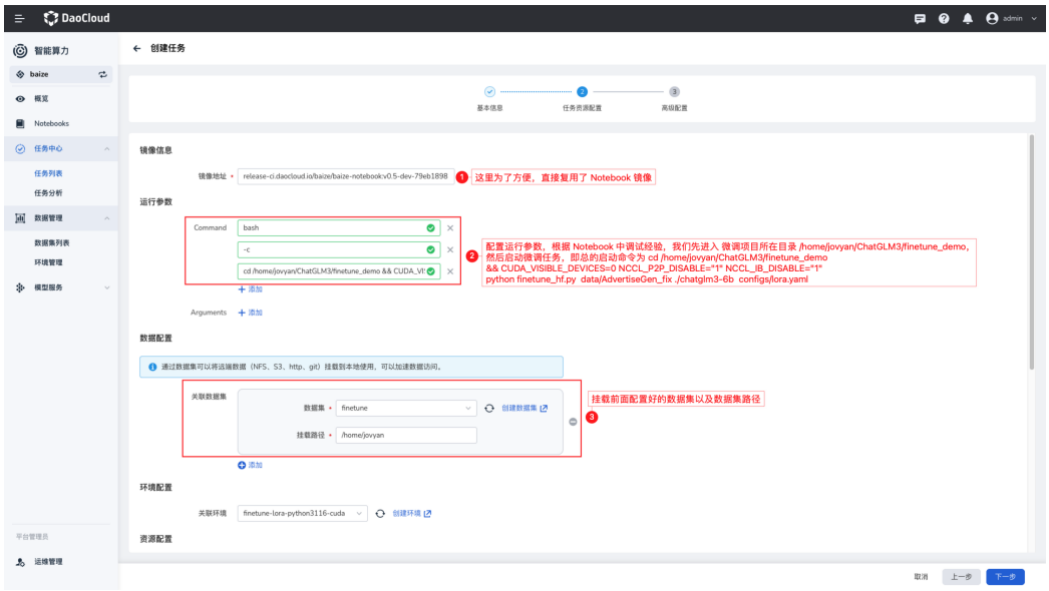

图 148　配置微调任务资源

- 镜像：可直接使用 baizectl 提供的模型镜像。
- 启动命令，根据您在 Notebook 中使用 LoRA 微调的经验，代码文件和数据在 /home/jovyan/ChatGLM3/finetune_demo 目录下，所以您可以直接使用这个路径：

```
bash -c "cd /home/jovyan/ChatGLM3/finetune_demo && CUDA_VISIBLE_DEVICES=0
NCCL_P2P_DISABLE="1" NCCL_IB_DISABLE="1" python finetune_hf.py  data/AdvertiseGen_fix
./chatglm3-6b  configs/lora.yaml"
```

- 挂载环境，这样之前预加载的环境依赖不仅可以在 Notebook 中使用，同时也可以在任务中使用。
- 数据集：直接使用之前预热的数据集
  - 将模型输出路径设置为之前创建的 PVC 数据集
  - 将 AdvertiseGen 数据集挂载到 /home/jovyan/ChatGLM3/finetune_demo/data/AdvertiseGen 目录下

- 配置足够的 GPU 资源，确保微调任务能够正常运行。

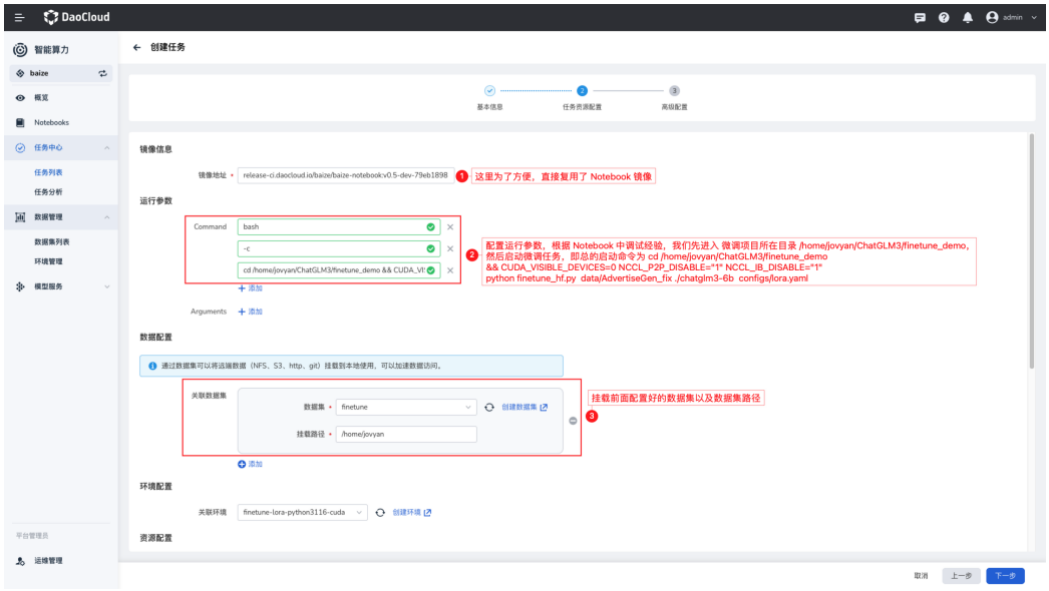

图 149　提交微调任务

查看任务状态

任务成功提交后，您可以在任务列表中实时查看任务的训练进展，这里您可以看到任务的状态、资源使用情况、日志等信息。

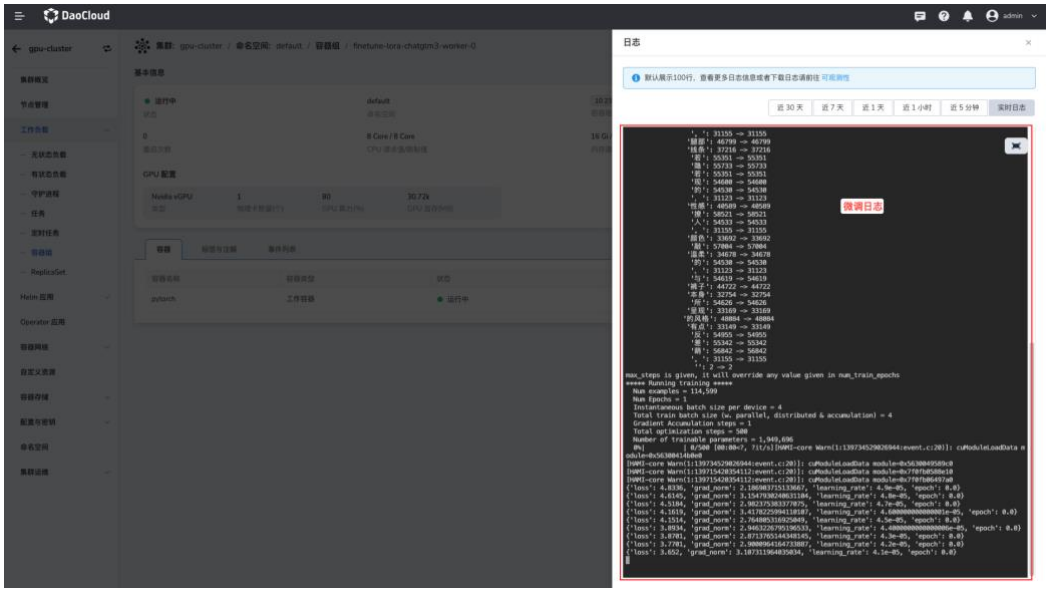

图 150　查看任务日志

任务运行完成后，您可以在数据输出的数据集中查看微调的模型文件，这样就可以使用这个模型文件进行后续的推理任务。

使用 `baizectl` 提交任务

DCE AI Lab 的 Notebook 支持免认证直接使用 `baizectl` 命令行工具，如果您喜欢使用 CLI，那么可以直接使用 `baizectl` 提供的命令行工具，提交任务。

```
baizectl job submit --name finetunel-chatglm3 -t PYTORCH \
    --image release.daocloud.io/baize/baize-notebook:v0.5.0 \
    --priority baize-high-priority \
    --resources cpu=8,memory=16Gi,nvidia.com/gpu=1 \
    --workers 1 \
    --queue default \
    --working-dir /home/jovyan/ChatGLM3 \
    --datasets AdvertiseGen:/home/jovyan/ChatGLM3/finetune_demo/data/AdvertiseGen  \
    --datasets output:/home/jovyan/ChatGLM3/finetune_demo/output  \
    --labels job_type=pytorch \
    --restart-policy on-failure \
    -- bash -c "cd /home/jovyan/ChatGLM3/finetune_demo && CUDA_VISIBLE_DEVICES=0
NCCL_P2P_DISABLE="1" NCCL_IB_DISABLE="1" python finetune_hf.py  data/AdvertiseGen_fix
./chatglm3-6b  configs/lora.yaml"
```

如果希望了解更多 `baizectl` 的使用说明，可以查看 baizectl 命令行工具。

模型推理

在微调任务完成后，您可以使用微调的模型进行推理任务，这里您可以使用 AI Lab 提供的推理服务，将输出后的模型创建为推理服务。

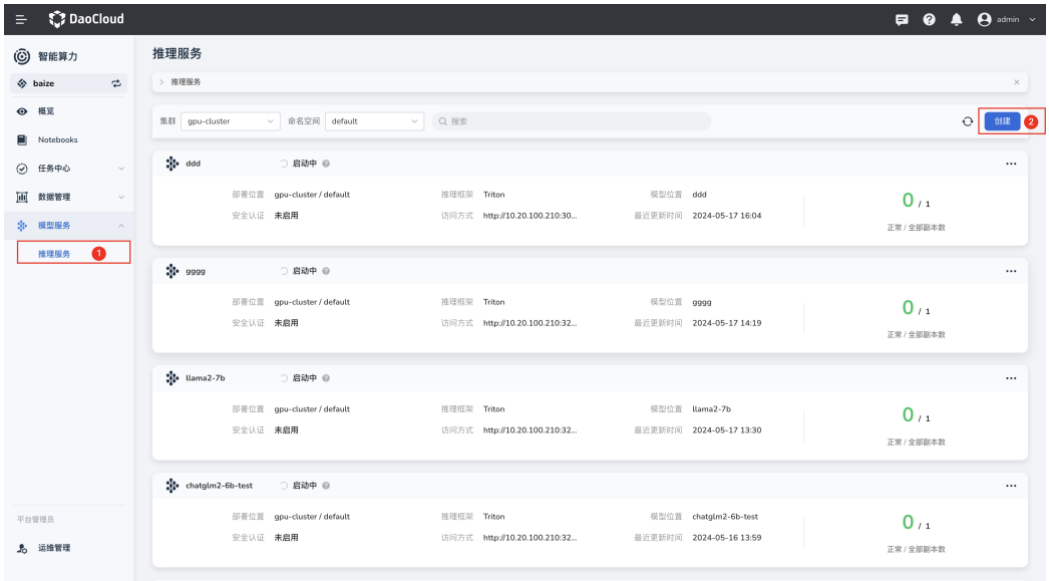

图 151　推理服务列表

在推理服务列表中，您可以创建一个新的推理服务，在选择模型的位置，选择之前推理输出的数据集，并配置模型路径。

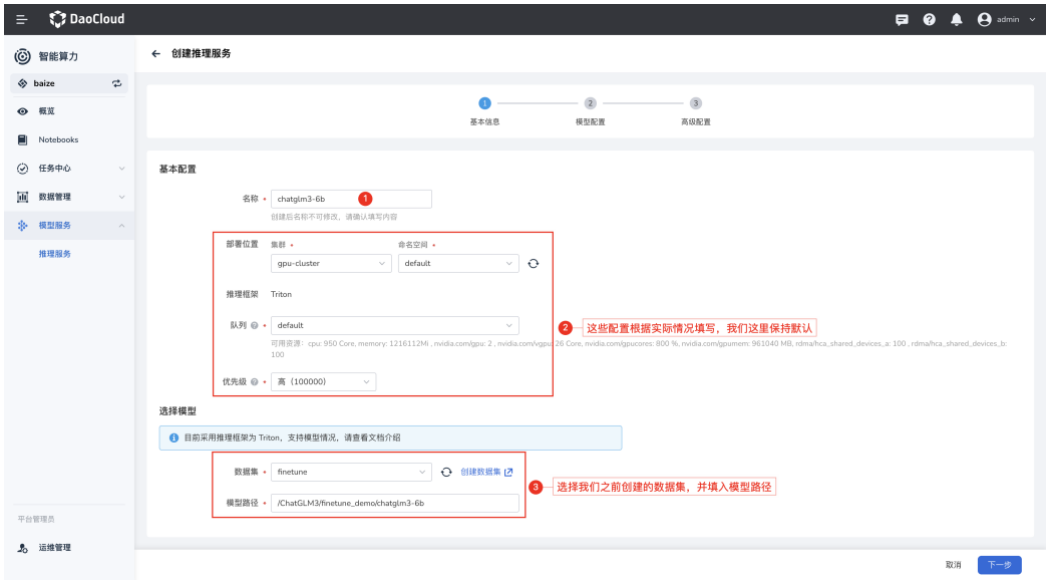

图 152　创建推理服务

有关模型资源要求、推理服务的 GPU 资源要求，需要根据模型的大小和推理的并发量来配置，这里您可以根据之前微调任务的资源配置来配置。

配置模型运行时

配置模型的运行时尤为重要，目前 DCE AI Lab 已经支持 vLLM 作为模型推理服务的运行时，可以直接选择 vLLM。

> vLLM 支持非常丰富的大语言模型，建议访问 vLLM 官方文档了解更多信息，这些模型都可以很方便地在 AI Lab 中使用。

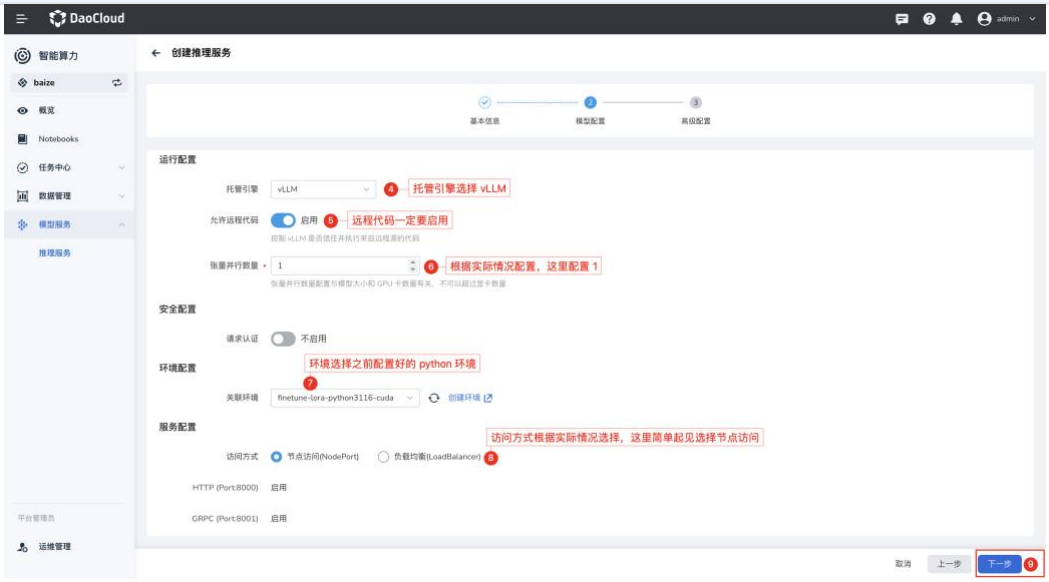

图 153　配置模型运行时

创建完成后，您可以在推理服务列表中看到您创建的推理服务，在模型服务列表，您可以直接获取模型的访问地址。

使用模型服务测试

简单在终端中尝试，使用 `curl` 命令来测试模型服务，这里您可以看到返回的结果，这样就可以使用模型服务进行推理任务了。

```
curl -X POST http://10.20.100.210:31118/v2/models/chatglm3-6b/generate \
  -d '{"text_input": "hello", "stream": false, "sampling_parameters": "{\"temperature\": 0.7, \"top_p\": 0.95, 'max_tokens\": 1024}"}'
```

图 154　验证模型服务

本文以 `ChatGLM3` 为例，带您快速了解和上手 **AI Lab** 的模型微调，使用 `LoRA` 微调了 ChatGLM3 模型。DCE AI Lab 提供了非常丰富的功能，可以帮助模型开发者快速进行模型开发、微调、推理等任务，同时也提供了丰富的 OpenAPI 接口，可以方便地与第三方应用生态进行结合。

（八）部署 Label Studio

> 参阅视频教程：数据标注和数据集使用说明。

Label Studio 是一个开源的数据标注工具，用于各种机器学习和人工智能任务。以下是 Label Studio 的简要介绍：

- 支持图像、音频、视频、文本等多种数据类型的标注
- 可用于目标检测、图像分类、语音转录、命名实体识别等多种任务
- 提供可定制的标注界面
- 支持多种标注格式和导出选项

Label Studio 通过其灵活性和功能丰富性，为数据科学家和机器学习工程师提供了强大的数据标注解决方案。

部署到 DCE

要想在 AI Lab 中使用 Label Studio，需将其部署到全局服务集群，你可以通过 Helm 的方式快速部署。

> 更多部署详情，请参阅 Label Studio 官方文档 Deploy Label Studio on Kubernetes。

1. 打开全局服务集群界面，从左侧导航栏找到 **Helm 应用** -> **Helm 仓库**，选择 **创建仓库** 按钮，填写如下参数：

图 155　创建 Helm 仓库

2. 添加成功后，点击列表右侧的 ⋮，选择 **同步仓库**，稍等片刻后完成同步。（后续更新 Label Studio 也会用到这个同步操作）。

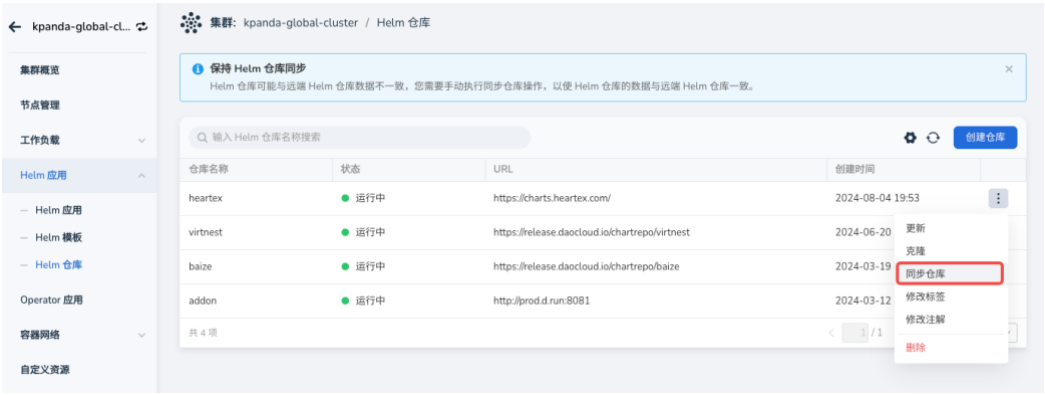

图 156　同步仓库

3. 然后跳转到 **Helm 模板** 页面，你可以搜索找到 `label-studio`，点击卡片。

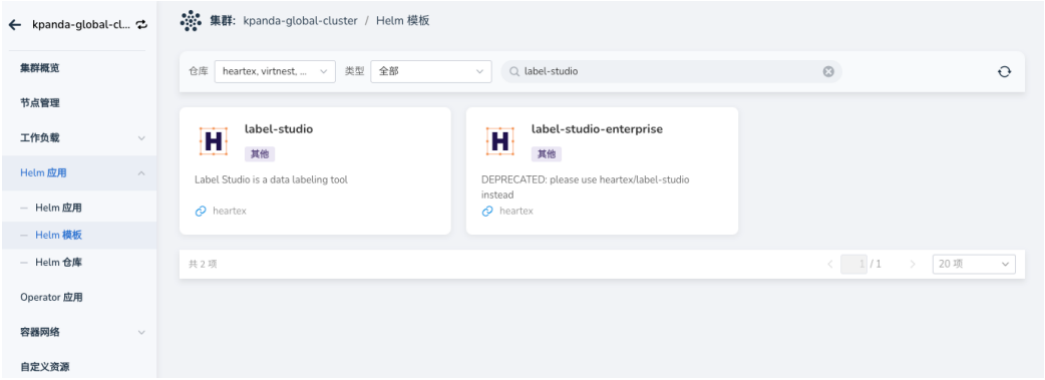

图 157　在 Helm 模板中找到 label-studio

4. 选择最新的版本，如下图配置安装参数，名称为 label-studio，建议创建新的命名空间，配置参数切换到 YAML，根据说明修改其中配置。

```
global:
  image:
    repository: heartexlabs/label-studio   # 如果无法访问 docker.io，在此处配置代理地址
  extraEnvironmentVars:
    LABEL_STUDIO_HOST: https://{DCE_访问地址}/label-studio    # 使用 DCE 的登录地址，请参阅当前网页 URL
    LABEL_STUDIO_USERNAME: {用户邮箱}    # 必须是邮箱，替换为自己的
    LABEL_STUDIO_PASSWORD: {用户密码}
app:
  nginx:
    livenessProbe:
      path: /label-studio/nginx_health
    readinessProbe:
      path: /label-studio/version
```

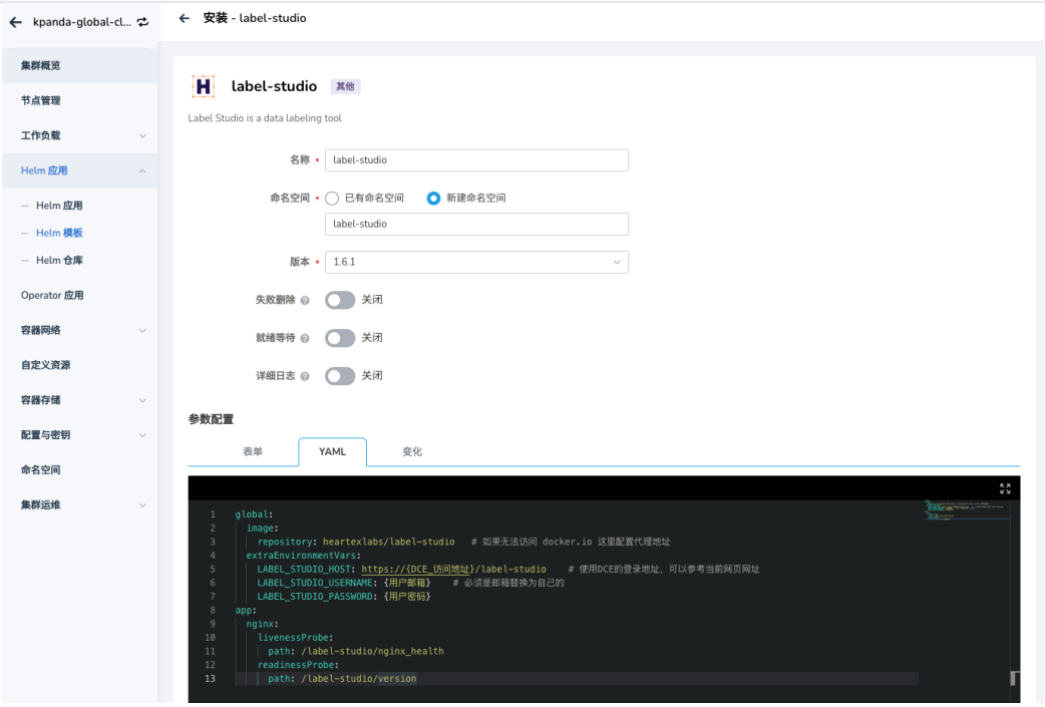

图 158　配置安装参数

至此，完成了 Label Studio 的安装。

> 默认会安装 PostgreSQL 作为数据服务中间件，如果镜像拉取失败，可能是 docker.io 无法访问，注意切换到可用代理即可。

如果你有自己的 PostgreSQL 数据服务中间件，可以使用如下参数配置：

```
global:
  image:
    repository: heartexlabs/label-studio   # 如果无法访问 docker.io，在此处配置代理地址
  extraEnvironmentVars:
    LABEL_STUDIO_HOST: https://{DCE_访问地址}/label-studio    # 使用 DCE 的登录地址，参阅当前网页 URL
    LABEL_STUDIO_USERNAME: {用户邮箱}    # 必须是邮箱，替换为自己的
    LABEL_STUDIO_PASSWORD: {用户密码}
```

```
app:
  nginx:
    livenessProbe:
      path: /label-studio/nginx_health
    readinessProbe:
      path: /label-studio/version
postgresql:
  enabled: false  # 禁用内置的 PostgreSQL
externalPostgresql:
  host: "postgres-postgresql"  # PostgreSQL 地址
  port: 5432
  username: "label_studio"  # PostgreSQL 用户名
  password: "your_label_studio_password"  # PostgreSQL 密码
  database: "label_studio"  # PostgreSQL 数据库名
```

添加 GProduct 到导航栏

如果要添加 Label Studio 到 DCE 导航栏，可以参考 DCE「全局管理 OEM IN」的方式。以下案例是增加到 AI Lab 二级导航的添加方式。

添加代理访问

```
apiVersion: ghippo.io/v1alpha1
kind: GProductProxy
metadata:
  name: label-studio
spec:
  gproduct: label-studio
  proxies:
  - authnCheck: false
    destination:
      host: label-studio-ls-app.label-studio.svc.cluster.local
      port: 80
    match:
      uri:
        prefix: /label-studio
```

添加到 AI Lab

修改 CRD 为 `GProductNavigator` 的 CR `baize`，然后在现有配置中进行如下变更：

```
apiVersion: ghippo.io/v1alpha1
kind: GProductNavigator
metadata:
  annotations:
    meta.helm.sh/release-name: baize
    meta.helm.sh/release-namespace: baize-system
  labels:
    app.kubernetes.io/managed-by: Helm
    gProductName: baize
  name: baize
spec:
  category: cloudnativeai
  gproduct: baize
  iconUrl: ./ui/baize/logo.svg
  isCustom: false
  localizedName:
    en-US: AI Lab
    zh-CN: AI Lab
  menus:
    - iconUrl: ''
      isCustom: false
```

```
      localizedName:
        en-US: AI Lab
        zh-CN: AI Lab
      name: workspace-view
      order: 1
      url: ./baize
      visible: true
    - iconUrl: ''
      isCustom: false
      localizedName:
        en-US: Operator
        zh-CN: 运维管理
      name: admin-view
      order: 1
      url: ./baize/admin
      visible: true
    # 添加开始
    - iconUrl: ''
      localizedName:
        en-US: Data Annotation
        zh-CN: 数据标注
      name: label-studio
      order: 1
      target: blank    # 控制新开页
      url: https://{DCE_访问地址}/label-studio    # 访问地址
      visible: true
    # 添加结束
  name: AI Lab
  order: 10
  url: ./baize
  visible: true
```

添加效果

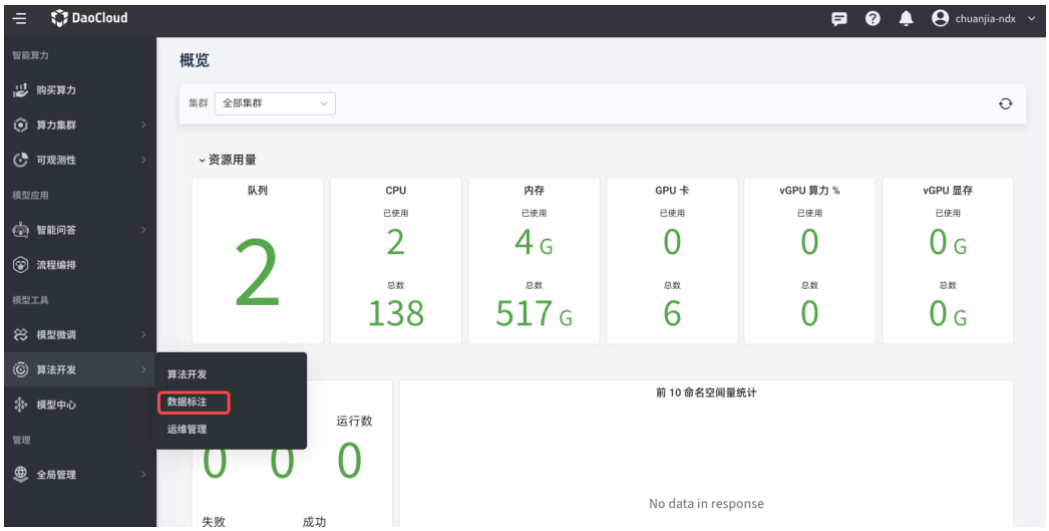

图 159　导航栏添加效果

结语

以上，就是如何添加 Label Studio 并将其作为 AI Lab 的标注组件，通过将标注后的数据添加到 AI Lab 的数据集中，联动算法开发，完善算法开发流程，后续如何使用请关注其他文档参考。

九、故障排查

本章将持续统计和梳理 AI Lab 使用过程可能因环境或操作不规范引起的报错，以及在使用过程中遇到某些报错的问题分析、解决方案。

> 本文档仅适用于 DCE 版本，若遇到 AI Lab 的使用问题，请优先查看此排障手册。

AI Lab 在 DCE 中模块名称 `baize`，提供了一站式的模型训练、推理、模型管理等功能。本章包含以下常见故障案例：

- 集群下拉列表中找不到集群
- Notebook 不受队列配额控制
- 本地队列初始化失败

（一）集群下拉列表中找不到集群

问题现象

在 AI Lab 开发控制台、运维控制台，功能模块的集群搜索条件的下拉列表找不到想要的集群。

问题分析

在 AI Lab 中，集群下拉列表如果缺少了想要的集群，可能是由于以下原因导致的：

- `baize-agent` 未安装或安装不成功，导致 AI Lab 无法获取集群信息。
- 安装 `baize-agent` 未配置集群名称，导致 AI Lab 无法获取集群信息。
- 工作集群内可观测组件异常，导致无法采集集群内的指标信息。

解决办法

`baize-agent` 未安装或安装不成功

AI Lab 有一些基础组件需要在每个工作集群内进行安装，如果工作集群内未安装 `baize-agent` 时，可以在界面上选择安装，可能会导致一些非预期的报错等问题。

所以，为了保障使用体验，可选择的集群范围仅包含了已经成功安装了 `baize-agent` 的集群。

如果是因为 `baize-agent` 未安装或安装失败，则使用 **容器管理** -> **集群管理** -> **Helm 应用** -> **Helm 模板**，找到 `baize-agent` 并安装。

> 此地址快速跳转 https://<dce_host>/kpanda/clusters/<cluster_name>/helm/charts/addon/baize-agent。注意将 <dce_host> 替换为实际的 DCE 控制台地址，<cluster_name> 替换为实际的集群名称。

安装 `baize-agent` 时未配置集群名称

在安装 baize-agent 时，需要注意配置集群的名称，这个名称会用于可观测指标采集，**默认为空，需手工配置**。

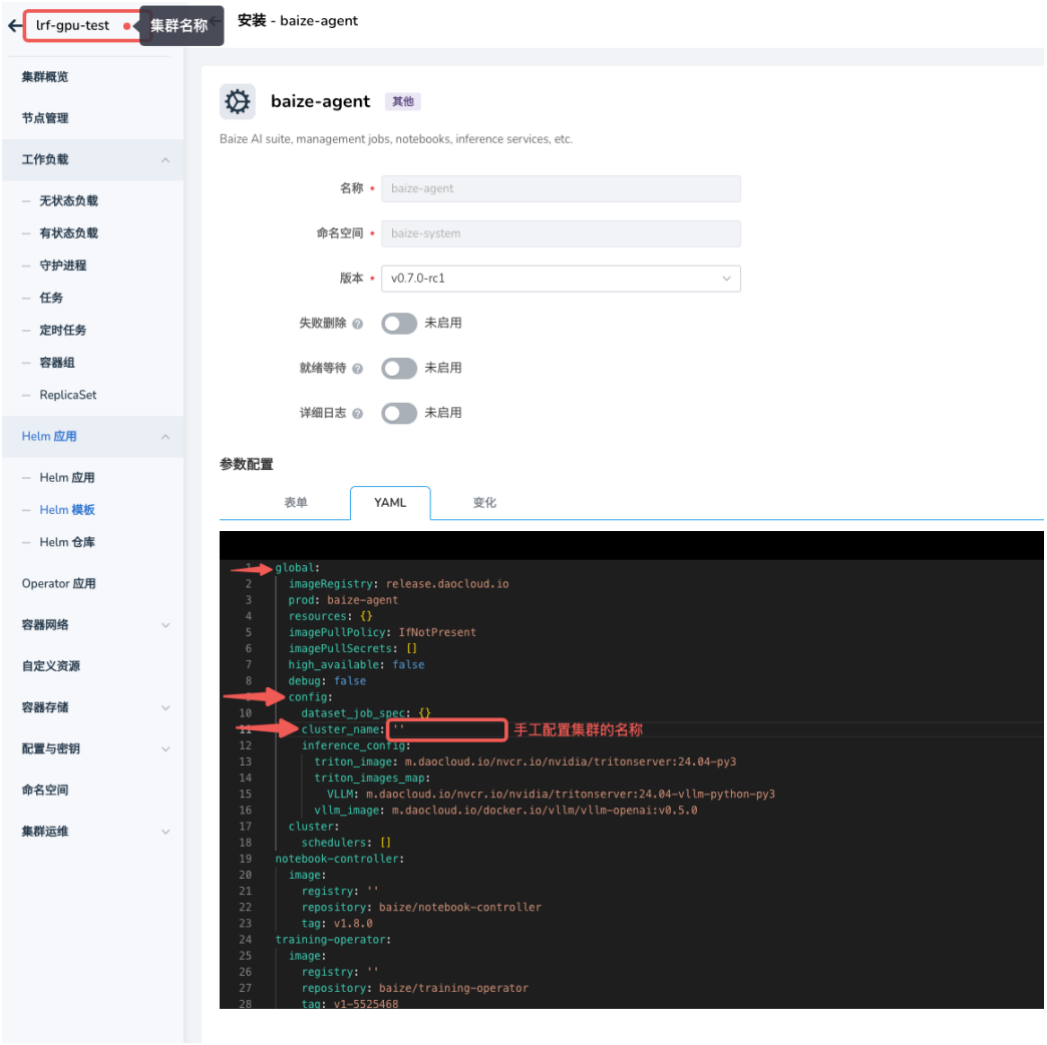

图 160　配置 baize-agent 的集群名称

工作集群内可观测组件异常

如果集群内可观测组件异常，可能会导致 AI Lab 无法获取集群信息，请检查平台的可观测服务是否正常运行及配置。

- 检查全局服务集群内 insight-server 组件是否正常运行
- 检查工作集群内 insight-agent 组件是否正常运行

（二）Notebook 不受队列配额控制

在 AI Lab 中，用户在创建 Notebook 时，发现选择的队列即使资源不足，Notebook 依然可以创建成功。

问题 01：Kubernetes 版本不支持

分析：

AI Lab 中的队列管理能力由 Kueue 提供，Notebook 服务是通过 JupyterHub 提供的。JupyterHub 对 Kubernetes 的版本要求较高，对于低于 v1.27 的版本，即使在 DCE 中设置了队列配额，用户在创建 Notebook 时也选择了配额，但 Notebook 实际也不会受到队列配额的限制。

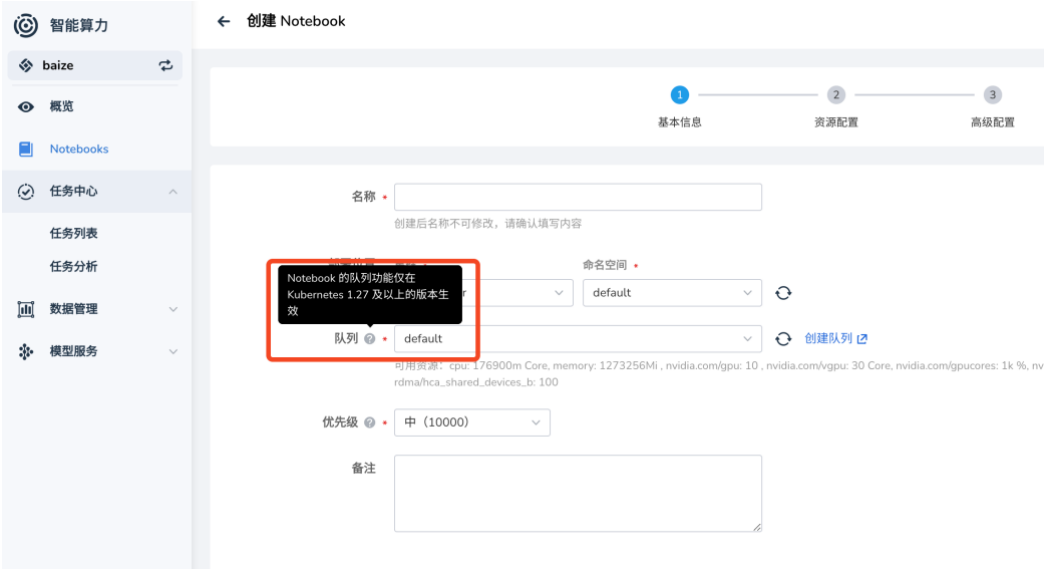

图 161 Kubernetes 版本低于 v1.27 的提示

解决办法：提前规划，生产环境中建议使用 Kubernetes 版本 `v1.27` 以上。

参考资料：Jupyter Notebook Documentation。

问题 02：配置未启用

分析：

当 Kubernetes 集群版本大于 v1.27 时，Notebook 仍无法受到队列配额的限制。

这是因为，Kueue 需要启用对 `enablePlainPod` 支持，才会对 Notebook 服务生效。

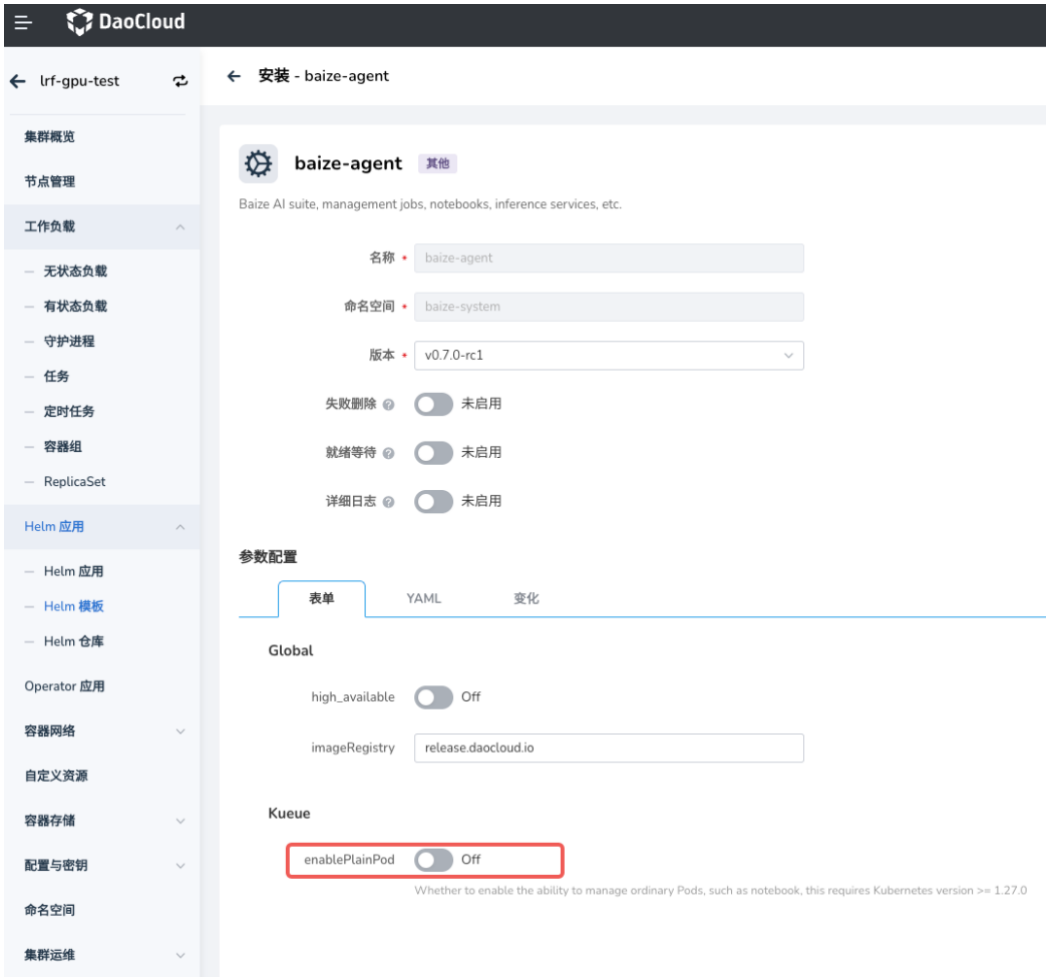

图 162 Kueue 的 enablePlainPod 配置

解决办法：在工作集群中部署 `baize-agent` 时，启用 Kueue 对 `enablePlainPod` 的支持。

参考资料：Run Plain Pods as a Kueue-Managed Job。

（三）本地队列初始化失败

问题现象

在创建 Notebook、训练任务或者推理服务时，当队列是首次在该命名空间使用时，会提示需要一键初始化队列，但是初始化失败。

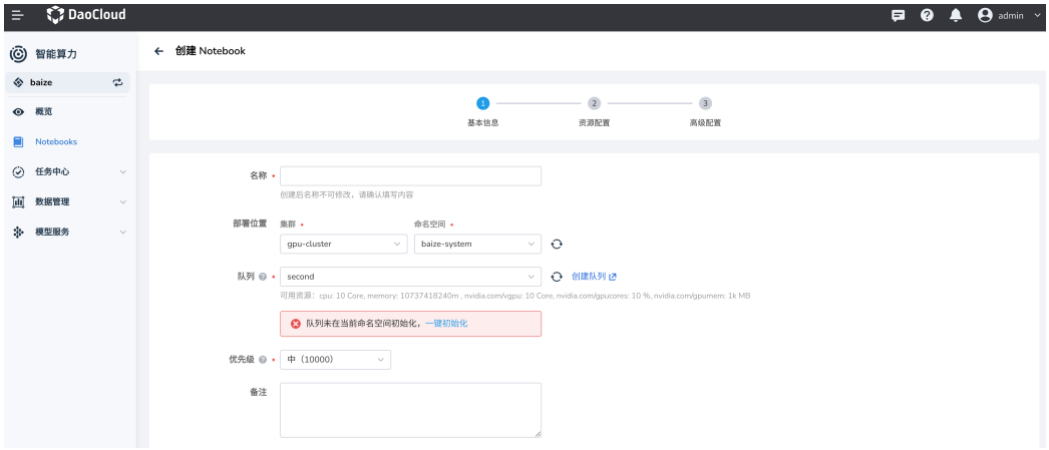

图 163　本地队列初始化失败

问题分析

在 AI Lab 中，队列管理能力由 Kueue 提供，而 Kueue 提供了两种队列管理资源：

- **ClusterQueue** 是集群级别的队列，主要用于管理队列中的资源配额，包含了 CPU、内存、GPU 等资源。
- **LocalQueue** 是命名空间级别的队列，需要指向到一个 ClusterQueue，用于使用队列中的资源分配。

在 AI Lab 中，如果创建服务时，发现指定的命名空间不存在 `LocalQueue`，则会提示需要初始化队列。在极少数情况下，可能由于特殊原因会导致 `LocalQueue` 初始化失败。

解决办法

检查 Kueue 是否正常运行，如果 `kueue-controller-manager` 未运行，可以通过以下命令查看。

```
kubectl get deploy kueue-controller-manager -n baize-system
```

如果 `kueue-controller-manager` 未正常运行，请先修复 Kueue。

参考资料：Kueue 官方文档中的 ClusterQueue、LocalQueue 概念说明。

十、附录

本附录收录 AI Lab 升级到新版本时需要注意的相关事项，以及各版本的 Release Notes，便于您了解版本演进路径和特性变化。

（一）升级注意事项

本页说明将 baize 升级到新版本时需要注意的相关事项。

从 v0.24.1（或更低版本）升级到 v0.25.1

Baize v0.25.1 重构数据空间依赖的组件 dataset，请参考下述步骤进行手动升级。Baize v0.25.1 重构调度感知组件 kueue，请参考下述步骤进行手动升级。

1. 升级 dataset

baize 使用的 dataset 的 crd 从 **datasets.dataset.baize.io** 变为 **datasets.dataset.baizeai.io**，下文将详细阐述如何平滑升级，保留原有数据的办法。

- 旧版本的 crd 的 apiVersion：**datasets.dataset.baize.io**
- 新版本的 crd 的 apiVersion：**datasets.dataset.baizeai.io**

Break change 详情

1. baize 和 hydra 都会使用 BaizeAI/dataset 这个项目来处理 dataset 的调谐。
2. baize 项目里将不再包含 dataset controller 相关的代码。
3. BaizeAI/dataset 将会以 helm 的方式部署在 dataset-system 命名空间里。
4. datasets.dataset.baizeai.io 的 cr 以 **"app.kubernetes.io/part-of"** 为 key 的 label 来区分是由 baize 还是 hydra 创建的。

升级步骤

1. 升级 Global 集群的 baize 的 apiserver，升级后暂时在页面看不到 dataset，因为 apiserver 查询的是新版本的 cr。
2. 停止运行 **baize-agent-cluster-controller** 和 **baize-agent-jobset-controller** 这两个 deployment，保留 **kueue-controller-manager**。

> 目前不要升级 worker 集群的 baize-agent，避免旧版的 crd 被删除。

3. 在 worker 集群安装 BaizeAI/dataset 的 helm 应用，安装完成后，在 **dataset-system** 命名空间里把唯一的 deployment 停止。
4. 这个时候 worker 集群会有两个版本的 crd 并存的情况，这个时候不要创建任何新的 dataset，不论是从页面还是调 API 接口。

```
kubectl get crd | grep dataset
datasets.dataset.baize.io                                 2026-03-05T07:28:15Z
datasets.dataset.baizeai.io                               2026-03-05T09:10:29Z
```

5. 如果是支持预热类型的（非 pvc 和 nfs）的 dataset 执行下面的脚本，第一个参数是命名空间，第二个参数是 dataset 的名字。

> 这个脚本会把 **dataset-datasetname-round-1** 的 job 删除，如果这个 job 和其对应的 pod 的日志有意义的话，先保存。

```
#!/bin/bash

# 检查参数
if [ $# -ne 2 ]; then
    echo "Usage: $0 <namespace> <dataset-name>"
    echo "Example: $0 default my-dataset"
    exit 1
fi

NAMESPACE="$1"
DATASET_NAME="$2"
PVC_NAME="$DATASET_NAME"
JOB_NAME=dataset-"${DATASET_NAME}-round-1"

echo "Namespace: $NAMESPACE"
echo "Dataset Name: $DATASET_NAME"
echo "PVC Name: $PVC_NAME"
echo "Job Name: $JOB_NAME"
echo ""

# ========== 第零步：获取 Dataset 信息并判断类型 ==========
echo "Step 0: Checking Dataset source type..."
DATASET_TYPE=$(kubectl get dataset.dataset.baize.io "$DATASET_NAME" -n "$NAMESPACE" -o
jsonpath='{.spec.source.type}' 2>/dev/null)

if [ -z "$DATASET_TYPE" ]; then
    echo "Error: Failed to get dataset source type"
    exit 1
fi
echo "Dataset source type: $DATASET_TYPE"

# 判断是否需要创建 Job
SKIP_JOB=false
if [ "$DATASET_TYPE" = "NFS" ] || [ "$DATASET_TYPE" = "PVC" ]; then
    SKIP_JOB=true
    echo "Type is NFS or PVC, will skip Job creation"
fi
echo ""

# ========== 第一步：更新 Dataset ==========
echo "Step 1: Updating Dataset..."

# 构建 jq 命令，根据类型决定是否添加 rcloneOp
if [ "$DATASET_TYPE" = "S3" ] || [ "$DATASET_TYPE" = "HTTP" ]; then
    echo "Type is S3 or HTTP, setting rcloneOp to syncMode"
    kubectl get dataset.dataset.baize.io "$DATASET_NAME" -n "$NAMESPACE" -o json | jq '
          del(
            .metadata.resourceVersion,
            .metadata.uid,
            .metadata.creationTimestamp,
            .metadata.managedFields,
            .metadata.generation,
            .metadata.selfLink
          )
          | .apiVersion="dataset.baizeai.io/v1alpha1"
          | .spec.dataSyncRound=1
          | .status.phase="Ready"
          | .status.inProcessingRound=1
          | .metadata.labels["app.kubernetes.io/part-of"] = "baize"
          | .spec.source.options.syncMode = .spec.source.options.rcloneOp
          | del(.spec.source.options.rcloneOp)
        ' | kubectl apply -f -
else
    kubectl get dataset.dataset.baize.io "$DATASET_NAME" -n "$NAMESPACE" -o json | jq '
          del(
            .metadata.resourceVersion,
            .metadata.uid,
            .metadata.creationTimestamp,
```

```
            .metadata.managedFields,
            .metadata.generation,
            .metadata.selfLink
          )
          | .apiVersion="dataset.baizeai.io/v1alpha1"
          | .spec.dataSyncRound=1
          | .status.phase="Ready"
          | .status.inProcessingRound=1
          | .metadata.labels["app.kubernetes.io/part-of"] = "baize"
        ' | kubectl apply -f -
fi

if [ $? -ne 0 ]; then
    echo "Error: Failed to update dataset"
    exit 1
fi
echo "Dataset updated successfully"
echo ""

# 如果类型为 PVC，只执行完第一步就结束
if [ "$DATASET_TYPE" = "PVC" ]; then
    echo "Type is PVC, only updating Dataset. Done."
    exit 0
fi

# 等待 Dataset 创建完成
sleep 10

# ========== 第二步：获取 Dataset 的 UID ==========
echo "Step 2: Getting Dataset UID..."
DATASET_UID=$(kubectl get dataset.dataset.baizeai.io "$DATASET_NAME" -n "$NAMESPACE" -o
jsonpath='{.metadata.uid}')

if [ -z "$DATASET_UID" ]; then
    echo "Error: Failed to get dataset UID"
    exit 1
fi
echo "Dataset UID: $DATASET_UID"
echo ""

# ========== 第三步：更新 PVC ==========
echo "Step 3: Checking PVC before update..."
echo "Current PVC ownerReferences:"
kubectl get pvc "$PVC_NAME" -n "$NAMESPACE" -o json | jq '.metadata.ownerReferences //
"null"'
echo ""

echo "Step 3: Updating PVC..."
kubectl get pvc "$PVC_NAME" -n "$NAMESPACE" -o json | jq --arg uid "$DATASET_UID" --arg
name "$DATASET_NAME" '
  del(.metadata.ownerReferences)
  | .metadata.ownerReferences = [{
      "apiVersion": "dataset.baizeai.io/v1alpha1",
      "kind": "Dataset",
      "name": $name,
      "uid": $uid,
      "controller": true,
      "blockOwnerDeletion": true
    }]
  | .metadata.labels["baize.io/dataset-name"] = $name
  | del(.metadata.labels["dataset.baize.io/name"])
' | kubectl replace -f -

if [ $? -ne 0 ]; then
    echo "Error: Failed to update PVC"
    exit 1
fi
echo "PVC updated successfully"
```

```
echo ""

# ========== 第三步（附加）：如果是 NFS 类型，更新 PV ==========
if [ "$DATASET_TYPE" = "NFS" ]; then
    echo "Step 3.5: Getting PV name from PVC..."
    PV_NAME=$(kubectl get pvc "$PVC_NAME" -n "$NAMESPACE" -o
jsonpath='{.spec.volumeName}')
    if [ -z "$PV_NAME" ]; then
        echo "Error: Failed to get PV name from PVC"
        exit 1
    fi
    echo "PV Name: $PV_NAME"
    echo ""

    echo "Step 3.5: Updating PV..."
    kubectl get pv "$PV_NAME" -o json | jq --arg uid "$DATASET_UID" --arg name
"$DATASET_NAME" '
      del(.metadata.ownerReferences)
      | .metadata.ownerReferences = [{
          "apiVersion": "dataset.baizeai.io/v1alpha1",
          "kind": "Dataset",
          "name": $name,
          "uid": $uid,
          "controller": true,
          "blockOwnerDeletion": true
        }]
      | .metadata.labels["baize.io/dataset-name"] = $name
      | del(.metadata.labels["dataset.baize.io/name"])
    ' | kubectl replace -f -

    if [ $? -ne 0 ]; then
        echo "Error: Failed to update PV"
        exit 1
    fi
    echo "PV updated successfully"
    echo ""
fi

# ========== 第四步：创建 Job（条件判断）==========
if [ "$SKIP_JOB" = true ]; then
    echo "Step 4: Skipping Job creation (type is $DATASET_TYPE)"
else
    echo "Step 4: Creating Job..."
    kubectl delete job "$JOB_NAME" -n "$NAMESPACE" 2>/dev/null || true
    cat <<-EOF | kubectl apply -f -
apiVersion: batch/v1
kind: Job
metadata:
  name: $JOB_NAME
  namespace: $NAMESPACE
  ownerReferences:
  - apiVersion: dataset.baizeai.io/v1alpha1
    kind: Dataset
    name: $DATASET_NAME
    uid: $DATASET_UID
    controller: true
    blockOwnerDeletion: true
spec:
  template:
    spec:
      containers:
      - name: main
        image: 10.6.178.191:5000/demo/busybox:latest
        command: ["sh", "-c", "sleep 1"]
        resources:
          requests:
            memory: "16Mi"
            cpu: "10m"
```

```
          limits:
            memory: "32Mi"
            cpu: "50m"
      restartPolicy: Never
  backoffLimit: 2
EOF

    if [ $? -ne 0 ]; then
        echo "Error: Failed to create Job"
        exit 1
    fi
    echo "Job created successfully"
fi
echo ""

# ========== 验证结果 ==========
echo "=== Verification ==="
echo "Dataset UID: $(kubectl get dataset "$DATASET_NAME" -n "$NAMESPACE" -o
jsonpath='{.metadata.uid}' 2>/dev/null || echo 'N/A')"
echo "PVC Owner UID: $(kubectl get pvc "$PVC_NAME" -n "$NAMESPACE" -o
jsonpath='{.metadata.ownerReferences[0].uid}' 2>/dev/null || echo 'N/A')"
if [ "$SKIP_JOB" = false ]; then
    echo "Job Status:"
    kubectl get job "$JOB_NAME" -n "$NAMESPACE" 2>/dev/null || echo "Job not found"
fi
echo ""
echo "Success! All resources created/updated."
```

6. 所有的 dataset 都用这个脚本执行完成后，可以升级 baize-agent。

7. 确认 baize-agent 升级完成后，可以删除旧版的 cr 及 crd（看需求，也可以留着不删）。

8. 恢复 **dataset-system** 里的 deployment，查看页面上的数据空间里的列表项状态是否正常。

> 如果类型是 NFS，脚本会修改 pvc 及 pvc 对应的 pv 的 label 和 ownerReferences，其他类型只会修改 pvc。

2. 升级 kueue

从 Baize v0.25.1 版本起，kueue 组件从 baize-agent 中独立出来，作为一个单独的插件（addon）部署在 **kueue-system** 命名空间下。目的是为了让 baize 和 hydra 两个系统能够共享使用 kueue 组件。下文将详细介绍升级操作步骤。

1. 升级到 v0.25.1 版本后，执行以下脚本：

> 注意此脚本需要保证所有的 workload 都被 admitted。

```
#!/bin/bash
set -e

# ========================================
# 环境变量配置区 (根据你的实际情况核对)
# ========================================
OLD_NAMESPACE="baize-system"
OLD_RELEASE="baize-agent"
NEW_NAMESPACE="kueue-system"
NEW_RELEASE="kueue"
```

```
BACKUP_DIR="./kueue-migration-backup"
VISIBILITY_ROLE_NAME="kueue-visibility-server-auth-reader"
VISIBILITY_ROLE_NAMESPACE="kube-system"

annotate_release() {
  local resource="$1"

  kubectl annotate "$resource" \
    meta.helm.sh/release-name="$NEW_RELEASE" \
    meta.helm.sh/release-namespace="$NEW_NAMESPACE" \
    --overwrite
}

annotate_release_in_namespace() {
  local kind="$1"
  local name="$2"
  local namespace="$3"

  kubectl annotate "$kind" "$name" -n "$namespace" \
    meta.helm.sh/release-name="$NEW_RELEASE" \
    meta.helm.sh/release-namespace="$NEW_NAMESPACE" \
    --overwrite
}

annotate_release_ignore_error() {
  local resource="$1"

  kubectl annotate "$resource" \
    meta.helm.sh/release-name="$NEW_RELEASE" \
    meta.helm.sh/release-namespace="$NEW_NAMESPACE" \
    --overwrite 2>/dev/null || true
}

annotate_release_in_namespace_ignore_error() {
  local kind="$1"
  local name="$2"
  local namespace="$3"

  kubectl annotate "$kind" "$name" -n "$namespace" \
    meta.helm.sh/release-name="$NEW_RELEASE" \
    meta.helm.sh/release-namespace="$NEW_NAMESPACE" \
    --overwrite 2>/dev/null || true
}

annotate_keep_policy() {
  local resource="$1"

  kubectl annotate "$resource" helm.sh/resource-policy=keep --overwrite
}

echo "🚀 开始：Kueue 资源所有权转移与旧控制器停机..."

# 1. 数据备份
echo "📦 [1/4] 正在备份当前集群中的 Kueue CRD 和实例数据..."
mkdir -p "$BACKUP_DIR"
kubectl get crd -o yaml | grep -A 10 "kueue.x-k8s.io" > "$BACKUP_DIR/kueue-crds-
backup.yaml" || true
kubectl get clusterqueue,localqueue,resourceflavor,workload,topology -A -o yaml >
"$BACKUP_DIR/kueue-data-backup.yaml" 2>/dev/null || true
echo "✅ 备份完成，保存在 ${BACKUP_DIR} 目录下。"

# 2. 转移 CRD 所有权并添加防删保护
echo "🔀 [2/4] 正在将集群级别 CRD 的 Helm 所有权转移给新 Release (${NEW_RELEASE})..."
CRDS=$(kubectl get crd -o name | grep 'kueue.x-k8s.io' || true)
if [ -n "$CRDS" ]; then
  for crd in $CRDS; do
    annotate_release "$crd"
    annotate_keep_policy "$crd"
```

```
    echo "  - 已接管: $crd"
  done
else
  echo "⚠ 未找到任何 Kueue CRD，请确认集群状态。"
fi

# 批量修改 v1beta1 和 v1beta2 的所有权标记
for version in v1beta1 v1beta2; do
  annotate_release "apiservice/${version}.visibility.kueue.x-k8s.io"
  echo "✅ APIService ${version} 已成功转移所有权"
done

# 3. 转移 ClusterRole 和 Binding 所有权
echo "🔀 [3/4] 正在转移 ClusterRole 和 ClusterRoleBinding 的所有权..."
ROLES=$(kubectl get clusterrole,clusterrolebinding -o name | grep 'kueue' || true)
if [ -n "$ROLES" ]; then
  for role in $ROLES; do
    annotate_release "$role"
  done
fi

# 4. 控制面平滑静默
echo "⏸  [4/4] 正在停止旧的 Kueue 控制器并清理 Webhook..."
kubectl scale deployment -l app.kubernetes.io/name=kueue -n "$OLD_NAMESPACE" --replicas=0
|| echo "⚠ 未找到旧 Deployment，可能已停止。"
kubectl delete validatingwebhookconfigurations,mutatingwebhookconfigurations -l
app.kubernetes.io/name=kueue --ignore-not-found

annotate_release_in_namespace rolebinding "$VISIBILITY_ROLE_NAME"
"$VISIBILITY_ROLE_NAMESPACE"

annotate_release_in_namespace_ignore_error role "$VISIBILITY_ROLE_NAME"
"$VISIBILITY_ROLE_NAMESPACE"

# 给所有名字包含 kueue 的 ClusterRole 和 ClusterRoleBinding 加上 keep 策略
for res in $(kubectl get clusterrole,clusterrolebinding,serviceaccount -o name | grep
kueue); do
  annotate_keep_policy "$res"
  echo "🛡️ 已为 $res 添加防删保护"
done
echo ""
echo "🎉 完成！此时集群内的旧控制器已静默，CRD 已安全隔离。"
echo "👉 现有的 Job 不受影响，但新提交的 Job 会短暂 Pending。"
echo "请立即执行: kueue 的安装与 baize-agent 的升级。"
```

2. 进入工作集群的 **Helm 应用** -> **Helm 模板** 页面，找到 kueue 插件并安装。

3. 进入工作集群的 **Helm 应用** -> **Helm 应用** 页面，找到 baize-agent，执行更新操作。

以上操作完成后，kueue 组件即可正常使用。

（二）Release Notes

本页列出 AI Lab 各版本的 Release Notes，便于您了解各版本的演进路径和特性变化。

> 标记为 Beta 的功能更新，在使用时请多注意，如遇问题请及时反馈。

2026-08-31

v0.29.2

- **新增** 数据空间支持通过 SCP、SFTP 方式上传本地数据
- **新增** 支持一个队列可以绑定多个工作空间，方便用户在不同工作空间之间共享队列资源
- **优化** 共享数据空间的产品交互体验
- **优化** 将 kcover 组件迁移至 addon 中
- **修复** 安全漏洞：CVE-2026-33186，CVE-2026-31789，CVE-2024-45337
- **修复** vllm dashboard 展示不准确的问题
- **修复** insight-agent 对接时产生的状态不准确的问题
- **修复** 数据空间标签不正确导致的数据空间可见性异常的问题

2026-06-30

v0.28.0

- **优化** 工作负载、队列 vGPU 产品交互
- **修复** 训练任务预检查阈值不生效的问题
- **修复** 镜像同步问题

2026-05-31

v0.27.0

- **优化** 升级 Kcover 至 0.0.8 版，支持沐曦 Day2 巡检任务及自动污点处理
- **优化** 升级推理服务容错检查功能以支持沐曦最新超节点
- **修复** CVE-2026-35469
- **修复** CVE-2026-29181
- **修复** CVE-2026-4117
- **修复** CVE-2026-41179
- **修复** 推理服务中 secret 不正常的问题

2026-04-30

v0.26.0

- **新增** 下载中心支持下载离线安装包，用于 AI Lab 模块的离线安装或升级
- **优化** Notebook、训练任务、推理服务手动输入镜像时，支持自动解析镜像 tag（前提：镜像地址在仓库存在）
- **修复** CVE-2026-33186 漏洞

2026-03-31

v0.25.1

- **优化** 重构数据空间依赖的组件 dataset，需要手动升级，请参考升级 dataset
- **优化** 重构调度感知组件（kueue），需要手动升级，请参考升级 kueue
- **修复** 创建 notebook ssh 不能正常工作的问题
- **修复** 集成镜像仓库根目录下的镜像时，界面异常无法选择镜像的问题
- **修复** CVE-2026-26958 漏洞
- **修复** 离线包中部分离线镜像缺失的问题
- **修复** baizectl 命令行工具在 notebook 中无法正常工作的问题

2026-01-31

v0.24.1

- **新增** 推理服务支持自定义推理框架
- **新增** Notebook 支持自定义启动命令
- **优化** 升级 vllm 至 v0.12.0
- **优化** 支持 kubeflow trainer v2，同时兼容 v1，训练任务可正常运行
- **修复** Notebook 运行时必须要 root 权限的问题
- **修复** 数据空间更新后，secretRef 字段丢失的问题

2025-12-31

v0.23.1

- **优化** 创建训练任务资源利用率优先和性能优先调度策略添加说明信息
- **修复** 修复概览界面中集群下拉列表不正确的问题
- **修复** 修复 Notebook 安全漏洞 CVE-2025-47914

2025-11-30

v0.23.0

- **新增** 分布式训练任务支持选择资源利用率优先或性能优先调度策略配置
- **新增** 训练任务支持镜像拉取策略配置
- **修复** 推理服务健康检查和就绪检查配置不符合界面操作的问题

2025-10-31

v0.22.0

- **新增** 训练任务支持 checkpoint 文件滚动删除功能

- **新增** 在监控面板中支持显示 GPU 相关指标的 GPU ID
- **新增** 训练任务支持暂停功能
- **新增** 镜像保存的关联日志查看能力
- **优化** 升级 kueue 至 v0.14.1
- **修复** 修复单机训练任务异常添加了 TAS 任务特性的问题

2025-09-30

v0.21.1

- **新增** 推理服务支持自定义镜像
- **新增** 训练任务支持容错预检查，识别任务异常并辅助定位问题，保障训练稳定性
- **优化** 训练任务支持资源池容忍时间设置
- **优化** 资源池 GPU 利用率及显存情况展示

2025-08-31

v0.20.2

- **新增** Notebook 支持开启 Docker in Docker 配置，以便在 Notebook 内执行 Docker 命令。
- **新增** 训练任务支持容忍配置
- **优化** 队列配额交互，更好地支持多型号 GPU
- **优化** GPU 监控面板链接修改为 insight 空间下 GPU 面板链接
- **优化** 资源池节点拓扑提示信息
- **优化** 升级 vLLM 至 v0.10.0 版
- **优化** 升级 Kubesnapshot 至 v0.2.8 版，解决基于 kind 安装报错问题
- **优化** 移除 Bitnami 相关镜像，利用 release 仓库中对应镜像替代
- **修复** 修复 TensorBoard 由于存储制度而不能展示数据的问题

2025-07-31

v0.19.1

- **新增** 训练任务支持拓扑感知调度
- **新增** 训练任务队列支持查看 GPU 型号信息
- **新增** 资源池支持根据 GPU 型号筛选节点
- **新增** 推理服务支持就绪检查和健康检查
- **新增** 推理服务支持共享内存配置
- **新增** HTTP 及 S3 类型数据空间支持预热模式配置
- **优化** 用户自定义镜像交互体验

- **优化** 升级 Kueue 版本至 v0.12.4
- **优化** 移除不必要的 RBAC 权限
- **修复** CVE-2025-22868 安全漏洞

2025-06-30

v0.18.1

- **新增** 资源池管理，可基于集群节点配置资源池
- **新增** 数据空间预热可选清除现有文件或保留现有文件的接口支持
- **优化** 产品逻辑中基本概念数据集为数据空间
- **优化** 升级 vLLM 镜像至 0.9.1 版
- **修复** 安全漏洞 2024-24790

2025-05-31

v0.17.3

- **优化** 升级 vLLM 框架镜像至 0.8.5-post1 版
- **优化** HTTP 及 S3 类型数据集预热时会清除存储中现有文件的逻辑，当前预热数据时不会删除任何文件
- **修复** 更新数据集类型异常的相关问题
- **修复** baize-agent 升级时出现异常 Webhook 逻辑的问题
- **修复** 界面创建推理服务报错的问题
- **修复** Notebook 中 baizectl 工具的资源权限问题
- **修复** 界面创建推理服务报错的问题

2025-04-30

v0.16.1

- **新增** 推理服务支持 vLLM 前置命令和后置命令运行参数配置
- **新增** vLLM 推理框架支持禁用 vLLM 命令，以满足分布式推理需求
- **新增** CRD 支持自定义数据集预热任务的 CPU/内存资源、训练镜像保存的超时时间和推理服务的启动超时时间
- **优化** 裁剪了 Notebook 镜像大小
- **优化** 推理服务，环境变量增加配置示例
- **优化** 创建数据集时，PVC 类型新增「创建 pvc」调整链接
- **优化** 更新推理服务，vLLM 推理框架支持修改运行配置
- **修复** 升级 kube-snapshot 至 v0.7.3 时，解决保存镜像时可能出现异常的问题
- **修复** git 方式数据集的 SSH 模式拉取数据的问题

- **修复** CVE-2025-22872、CVE-2025-30204 安全漏洞

2025-03-31

v0.15.1

- **新增** 训练推理任务选择多 GPU 卡时可对共享内存进行配置
- **新增** vLLM 框架添加 API Key 的能力
- **优化** 训练任务镜像默认选择 Notebook 镜像交互体验
- **优化** 创建数据集前的环境检测，避免集群中没有默认 storageClassName 导致数据集无法运行的情况
- **优化** 为避免使用风险，移除 mamba 的 defaults 的 channel
- **优化** 升级 vLLM 框架镜像至 0.7.3 版
- **修复** 训练任务 RDMA 标签未按预期添加的问题
- **修复** 训练任务无基础配置时导致崩溃的问题
- **修复** CVE-2025-22870，CVE-2024-45337 安全漏洞
- **修复** 由于 Notebook 中依赖版本过新导致标注实例无法正常运行的问题
- **修复** 训练任务 TAS 开启的问题（该功能所依赖的 Kueue 版本尚未正式发布，如需体验该功能请咨询产品研发）
- **修复** 在数据库为 kingbase 以及 postgresql 时产品界面功能异常的问题
- **修复** Triton 框架更新 API Key 功能异常的问题

2025-02-28

v0.14.1

- **新增** 训练任务支持启用 RDMA 配置。
- **新增** 数据集支持添加 HF_ENDPOINT 环境变量。
- **新增** 监控面板添加时间区间选择功能。
- **优化** 升级 vLLM 镜像至 0.7.1 版（支持 DeepSeek）。
- **优化** 升级 Kueue 至 0.10.1 版。
- **修复** 监控面板图例颜色不符合预期的问题。
- **修复** 运维管理概览页面中 GPU 资源图表显示错误问题。

2025-01-31

v0.13.0

- **优化** 默认 vllm 镜像升级至 0.6.6 以提高训练和推理任务的兼容性。
- **修复** 训练任务配置断点续训但是任务详情中仍为未启用状态的问题。
- **修复** 训练任务监控 GPU 使用率始终是 no data 的问题。

- **修复** 不存在默认资源池的情况下界面操作无法进行的问题。

2024-12-31

v0.12.0

- **新增** 支持队列自定义资源池。
- **新增** 沐曦 GPU 监控看板，丰富 GPU 观测指标。
- **修复** 漏洞 CVE-2024-45337, CVE-2024-45338。
- **修复** 无法正常创建数据集问题。

2024-11-30

v0.11.0

- **新增** Notebook、数据集、训练任务 以及 推理服务 状态详情展示，提高异常处理效率。
- **新增** 运维控制台内，队列管理可在队列详情页面查看所有使用了队列的资源。
- **优化** 优化数据集更新交互。

2024-10-31

v0.10.0

功能

- **新增** 训练任务 支持在配置 vGPU 资源时指定使用的显卡类型。
- **新增** 数据集 支持 Huggingface 数据源，可下载其海量模型和数据集。
- **新增** 数据集 支持 Modelscope 数据源，可下载其海量模型和数据集。
- **新增** 支持 数据集 的 **跨命名空间** 引用能力。
- **新增** 推理服务 支持在配置 vGPU 资源时指定使用的卡类型。
- **新增** 运维控制台 GPU 管理模块，支持查看卡级别的监控和指标信息。
- **新增** 适配 沐曦 GPU 卡。

优化

- **优化** 数据集更新界面，提供更多配置更新能力。
- **优化** 调整了 Notebook 的入口位置，提升访问便捷性。

2024-09-30

v0.9.0

> 产品模块名称从 智能算力 升级为 AI Lab。

- **新增** 全新数据管理子模块 数据标注，可管理主流数据类型的数据标注能力。
- **新增** 全新模型管理子模块 模型列表，可快速创建模型，支持模型多版本管理。

- **新增** 数据集 创建时可指定使用 PVC 存储空间大小。
- **新增** 支持 训练任务 一键重启。
- **新增** baize-notebook 基础镜像升级到 v0.9.0。
- **优化** 支持集群异常时，全局提醒同时保证数据可用。

2024-08-31

v0.8.0

- **新增** [Beta] 支持 Notebook 运行中时，手工保存为镜像（依赖镜像仓库模块）。
- **新增** [Beta] 支持 Notebook 关闭时自动保存为镜像（依赖镜像仓库模块）。
- **新增** 支持 Notebook 镜像通过表单选择镜像仓库内的私有镜像。
- **新增** 支持 Notebook 配置 **数据输入、数据输出**，可直接关联数据集。
- **新增** 支持 Notebook 配置以 Root 身份启动。
- **新增** 支持 训练任务 配置 **数据输入、数据输出**，可直接关联数据集。
- **新增** [Beta] 支持 训练任务 支持配置断点续训，自动检测任务故障后自动修复。
- **新增** 支持 训练任务 镜像通过表单选择镜像仓库内的私有镜像。
- **新增** 支持 训练任务 详情增加展示任务参数信息。
- **新增** 环境管理 可查询预热进度，并支持快速调试入口。
- **新增** 支持 推理任务 详情增加服务调用监控。
- **新增** baize-notebook 基础镜像升级到 v0.8.0。

2024-07-31

v0.7.0

- **新增** 支持 数据集 创建数据集后可查询预热进度，并支持快速调试入口。
- **新增** 支持 训练任务 创建 MxNet 单机和分布式任务。
- **新增** 支持 训练任务 创建 MPI 分布式任务。
- **新增** 支持 训练任务 支持默认镜像，统一使用基础镜像。
- **新增** 支持 训练任务 启动命令可直接配置启动脚本。
- **新增** 支持 训练任务 运行参数指定工作目录位置。
- **新增** 支持 推理任务 详情展示 API 调用示例文档。
- **优化** 环境管理 列表展示环境有的包管理器及 Python 版本。

2024-07-10

v0.6.1

- **修复** 创建推理服务时，推理框架选择使用 Triton，托管引擎缺少 vLLM 选项。

2024-06-30

v0.6.0

功能

- **新增** 支持创建 Code 类型的 Notebook，提供原生 VS Code 开发体验。
- **新增** 支持快速复制 Notebook。
- **新增** 支持在选择工作集群时，展示集群的状态信息，当失联或离线时不可选择。
- **新增** 支持创建推理服务时，使用 vLLM 作为推理引擎，暴露原生 vLLM 能力。
- **新增** 支持创建推理服务时，vLLM 支持配置 Lora 推理参数。
- **优化** 创建 Notebook 时，队列优先级默认值调整为 高。

修复

- **修复** Tensorboard 最小资源限制，避免因资源不足导致 Tensorboard 启动失败。
- **修复** 优化任务状态中文描述，避免因状态描述不清晰导致的误解。

2024-05-30

v0.5.0

功能

- **新增** 支持 baizectl 创建任务时同时增加 Tensorboard 分析看板。
- **新增** 支持 Job 绑定 环境管理 中创建的自定义环境。
- **新增** 支持 环境管理 中进行自定义环境配置更新、优化 Python 版本选择器等。
- **新增** 支持 推理服务 详情，查看模型运行时的资源监控看板。
- **新增** 支持 推理服务 绑定 环境管理 中创建的自定义环境。

修复

- **修复** 环境管理中少数情况下 Python 版本提示权限问题情况。
- **修复** 推理服务在异常时不支持停止的问题。

2024-04-30

v0.4.0

功能

- **新增** Notebook 支持本地 SSH 访问，适配多种开发工具，如 Pycharm、VS Code 等。
- **新增** 升级 Notebook 镜像，支持内置 CLI 工具 baizectl，命令行提交和管理任务。
- **新增** Notebook 增加亲和性调度策略配置。
- **新增** 分布式训练任务，可界面化配置 SHM size。
- **新增** 训练任务一键重启功能。

- **新增** 模型训练任务支持自定义指定集群调度器。
- **新增** 训练任务分析工具 Tensorboard 支持，可在 Notebook 与训练任务中一键启动。
- **新增** 队列配额编辑时，提示当前工作空间的共享资源配置。
- **新增** 升级适配 Kueue 版本 v0.6.2。

修复

- **修复** Notebook CRD 偶现配置同步异常问题。
- **修复** Notebook 亲和性配置参数查询接口未返回。

2024-04-01

v0.3.0

- **新增** 发布 Notebook 模块，支持 Jupyter Notebook 等开发工具。
- **新增** 发布任务中心模块，支持多种主流开发框架 Pytorch、Tensorflow、Paddle 任务训练。
- **新增** 发布模型推理服务模块，支持快速部署 Model Serving，支持任意模型算法与大语言模型。
- **新增** 发布数据管理模块，支持接入 S3、NFS、HTTP 及 Git 等主流数据源，并支持自动数据预热。