



湖南大學  
HUNAN UNIVERSITY

## 实验名称: Volcano 算力调度系统

成员	工作
束建杰	学习并指导集群与节点的部署，负责 Volcano 调度的使用与管理
张铭昊	负责镜像、Pod、Deployment 的部署至节点，以及算法分发
陈恩展	负责 React UI 前端设计与优化
叶晨新	负责节点 API 设计与前后端交互

# 目录

1	实验目的	3
2	实验环境	3
3	实验内容——Volcano 算力调度系统安装与配置	3
3.1	Docker 安装 . . . . .	4
3.2	kubectll 安装 . . . . .	4
3.3	kind 安装 . . . . .	5
3.4	Kubernetes 集群搭建 . . . . .	5
3.5	Volcano 安装 . . . . .	7
4	实验内容——部署和运行一个深度学习任务	10
4.1	删除旧 k8s 集群并安装新集群与 Volcano . . . . .	10
4.2	初始化节点 . . . . .	12
4.3	常用命令 . . . . .	20
5	实验内容——二次开发：UI 化节点管理工具	22
6	实验总结	40

# 1 实验目的

- 熟悉 Volcano 算力调度系统的架构和工作原理。
- 掌握 Volcano 算力调度系统的安装与配置方法。
- 了解 Volcano 算力调度系统的常用功能和操作流程。
- 通过实际操作，提升对分布式计算资源管理的理解和应用能力。

# 2 实验环境

本实验在本地计算机上搭建 Kubernetes 集群,并在其上安装 Volcano 算力调度系统。实验环境配置如下:

- 操作系统: Ubuntu 22.04 LTS
- Kubernetes 版本: v1.30.0
- Docker 版本: 20.10.7
- kind 版本: v0.23.0
- Helm 版本: v3.11.2
- Volcano 版本: v1.8.0
- 计算机配置: Intel Core i7-10700K CPU @ 3.80GHz, 32GB RAM
- 网络环境: 本地局域网
- 其他工具: kubectl, curl

# 3 实验内容——Volcano 算力调度系统安装与配置

由于实验环境的限制,我们无法获取云服务器,故采用 kind (Kubernetes IN Docker) 在本地搭建 Kubernetes 集群,并在其上安装

Volcano 算力调度系统，完成相关实验任务。

## 3.1 Docker 安装

更新 apt 包索引并安装依赖：

```
1 sudo apt-get update
2 sudo apt-get install -y ca-certificates curl gnupg
```

添加 Docker 官方 GPG 密钥：

```
1 sudo install -m 0755 -d /etc/apt/keyrings
2 curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/
  ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/
  docker.gpg
```

安装 Docker Engine：

```
1 sudo apt-get update
2 sudo apt-get install -y docker-ce docker-ce-cli containerd
  .io docker-buildx-plugin docker-compose-plugin
```

## 3.2 kubectl 安装

kubectl 是 Kubernetes 的命令行工具，用于与 Kubernetes 集群进行交互和管理。

下载 kubectl

```
1 curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
  dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

## 安装 kubectl

```
1 sudo install -o root -g root -m 0755 kubectl /usr/local/  
   bin/kubectl
```

## 3.3 kind 安装

kind 是一个用于在本地运行 Kubernetes 集群的工具，特别适合用于测试和开发环境。

### 下载 kind

```
1 curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.23.0/kind-  
   linux-amd64
```

### 安装 kind

```
1 chmod +x ./kind  
2 sudo mv ./kind /usr/local/bin/kind
```

## 3.4 Kubernetes 集群搭建

### 赋予 kind 创建集群的权限

使用 kind 创建一个包含一个控制平面节点和三个工作节点的 Kubernetes 集群。

在创建集群之前，要先赋予 docker 用户组权限：

```
1 sudo usermod -aG docker $USER  
2 newgrp docker  
3 docker info # 检查是否有权限
```

### 创建集群配置文件

创建一个名为 kind-config.yaml 的配置文件，内容如下：

```

1  kind: Cluster
2  apiVersion: kind.x-k8s.io/v1alpha4
3  nodes:
4  - role: control-plane
5  - role: worker
6  - role: worker
7  - role: worker
8  networking:
9  apiServerAddress: "127.0.0.1"

```

## 创建 Kubernetes 集群

使用以下命令创建 Kubernetes 集群：

```

1  kind create cluster --name mycluster --config kind-config.
    yaml # 注意此处配置文件要修改为自己的路径

```

## 验证集群状态

使用以下命令验证集群是否创建成功：

```

1  kubectl cluster-info --context kind-mycluster
2  kubectl get nodes

```

会看到类似如下输出，表示集群已成功创建并运行：

1	NAME	STATUS	ROLES	AGE
	VERSION			
2	mycluster-control-plane v1.30.0	Ready	control-plane	6m33s
3	mycluster-worker v1.30.0	Ready	<none>	6m11s
4	mycluster-worker2 v1.30.0	Ready	<none>	6m12s
5	mycluster-worker3 v1.30.0	Ready	<none>	6m13s

```

● zmh@zmhvm:~$ kubectl get nodes
NAME                                STATUS    ROLES    AGE   VERSION
mycluster-control-plane            Ready    control-plane   15h   v1.30.0
mycluster-worker                   Ready    <none>         15h   v1.30.0
mycluster-worker2                  Ready    <none>         15h   v1.30.0
mycluster-worker3                  Ready    <none>         15h   v1.30.0

```

## 3.5 Volcano 安装

Volcano 是一个基于 Kubernetes 的高性能计算（HPC）和大规模机器学习（ML）工作负载调度系统。它提供了丰富的调度策略和资源管理功能，适用于需要高效利用计算资源的场景。

### 安装 Helm

Helm 是 Kubernetes 的包管理工具，类似于 Linux 系统中的 apt 或 yum。它简化了 Kubernetes 应用程序的安装和管理过程。

```

1  curl https://raw.githubusercontent.com/helm/helm/main/
    scripts/get-helm-3 | bash
2  helm version

```

### 添加 Volcano Helm 仓库

使用以下命令添加 Volcano 的 Helm 仓库：

```

1  helm repo add volcano-sh https://volcano-sh.github.io/helm
    -charts
2  helm repo update

```

若安装成功，会看到类似如下输出：

```

1  Hang tight while we grab the latest from your chart
    repositories...
2  ...Successfully got an update from the "volcano-sh" chart
    repository
3  Update Complete. Happy Helming!

```

### 部署 Volcano

使用以下命令在 Kubernetes 集群中部署 Volcano：

```

1 kubectl create namespace volcano-system
2 helm install volcano volcano-sh/volcano \
3 --namespace volcano-system \
4 --set basic.enable=true

```

若安装成功，会看到类似如下输出：

```

1 NAME: volcano
2 LAST DEPLOYED: Wed Sep 17 01:40:18 2025
3 NAMESPACE: volcano-system
4 STATUS: deployed
5 REVISION: 1
6 TEST SUITE: None
7 NOTES:
8 Thank you for installing volcano.
9
10 Your release is named volcano.
11
12 For more information on volcano, visit:
13 https://volcano.sh/

```

## 简单验证 Volcano 安装

使用以下命令验证 Volcano 是否安装成功：

```

1 kubectl get pods -n volcano-system

```

若安装成功，会看到类似如下输出：

1	NAME	READY	STATUS	
	RESTARTS AGE			
2	volcano-admission-xxxxxx	1/1	Running	0
	1m			
3	volcano-controller-xxxxxx	1/1	Running	0
	1m			
4	volcano-scheduler-xxxxxx	1/1	Running	0
	1m			

```

zmh@zmhvm:~$ kubectl get pods -n volcano-system
NAME                                READY   STATUS    RESTARTS   AGE
volcano-admission-6496b59554-7rh92  1/1     Running   0           2m8s
volcano-controllers-7c58bf7dc4-xzww6 1/1     Running   0           2m8s
volcano-scheduler-597ccd8f49-2mz5s   1/1     Running   0           2m8s

```



## 提交一个简单的 Volcano 任务

创建一个名为 vcjob.yaml 的配置文件，内容如下：

```
1  apiVersion: batch.volcano.sh/v1alpha1
2  kind: Job
3  metadata:
4    name: test-job
5  spec:
6    minAvailable: 1
7    schedulerName: volcano
8    tasks:
9      - replicas: 2
10       name: "sleep"
11       template:
12         spec:
13           containers:
14             - image: busybox
15               command: ["sleep", "60"]
16               name: sleep-container
17           restartPolicy: Never
```

这个任务定义了一个包含两个副本的任务，每个副本运行一个简单的 sleep 命令，持续 60 秒。

```
1  kubectl apply -f vcjob.yaml # 注意此处配置文件要修改为自己的
    路径
2  kubectl get vcjob
3  kubectl get pods
```

若任务提交成功，会看到 test-job 启动了 2 个 busybox 容器，运行 60 秒。

```
zmh@zmvhvm:~$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
test-job-sleep-0    1/1     Running   0           59s
test-job-sleep-1    1/1     Running   0           59s
```

至此，我们演示了如何在本地使用 kind 搭建 Kubernetes 集群，并在其上安装和配置 Volcano 算力调度系统。通过提交一个简单的任务，我们验证了 Volcano 的基本功能。

## 4 实验内容——部署和运行一个深度学习任务

由于我们是在本地部署的 Kubernetes 集群，资源有限，无法运行大型的深度学习任务。为了演示 Volcano 的调度能力，我们选择了一个相对较小的深度学习任务：MNIST 手写数字识别任务。这个任务使用 TensorFlow 框架，在一个小型的神经网络上训练 MNIST 数据集。

### 4.1 删除旧 k8s 集群并安装新集群与 Volcano

深度学习任务对计算资源的需求较高，默认的 kind 集群配置可能无法满足需求。因此，我们首先删除旧的 k8s 集群，并创建一个包含更多资源的新集群。

```
1 kind delete cluster --name mycluster
```

创建一个新的配置文件 kind-config-large.yaml，内容如下：

```
1 kind: Cluster
2 apiVersion: kind.x-k8s.io/v1alpha4
3 nodes:
4 - role: control-plane
5   kubeadmConfigPatches:
6   - |
7       kind: ClusterConfiguration
8       apiServer:
9       extraArgs:
10         "runtime-config": "api/all=true"
11   extraPortMappings:
12   - containerPort: 30000
13     hostPort: 30000
14   # 限制 master 节点资源
15   kubeadmConfigPatchesJSON6902:
16   - group: kubeadm.k8s.io
17     version: v1beta3
18     kind: InitConfiguration
19     patch: |
20     [
21       {"op": "add", "path": "/nodeRegistration/
22         kubeletExtraArgs/system-reserved", "value":
```

```

22         "cpu=2000m,memory=2Gi"}
23     ]
24     - role: worker
25       extraMounts:
26         - hostPath: /mnt/data
27           containerPath: /data
28       kubeadmConfigPatchesJSON6902:
29         - group: kubeadm.k8s.io
30           version: v1beta3
31           kind: JoinConfiguration
32           patch: |
33             [
34               {"op": "add", "path": "/nodeRegistration/
35                 kubeletExtraArgs/system-reserved", "value":
36                 "cpu=3000m,memory=6Gi"}
37             ]
38     - role: worker
39       extraMounts:
40         - hostPath: /mnt/data
41           containerPath: /data
42       kubeadmConfigPatchesJSON6902:
43         - group: kubeadm.k8s.io
44           version: v1beta3
45           kind: JoinConfiguration
46           patch: |
47             [
48               {"op": "add", "path": "/nodeRegistration/
49                 kubeletExtraArgs/system-reserved", "value":
50                 "cpu=3000m,memory=6Gi"}
51             ]

```

这个配置文件定义了一个包含一个控制平面节点和两个工作节点的集群，注意我们为控制节点分配 2 核 CPU 和 2GB 内存，为每个工作节点分配 3 核 CPU 和 6GB 内存，预留了 2GB 内存用于系统使用。

最后，使用以下命令创建新的 Kubernetes 集群并重新安装 Volcano：

```

1  kind create cluster --name dl-cluster --config kind-config
   .yaml # 注意此处配置文件要修改为自己的路径
2  # 重新安装Volcano

```

```

3 helm repo add volcano-sh https://volcano-sh.github.io/helm
   -charts
4 helm repo update
5 kubectl create namespace volcano-system
6 helm install volcano volcano-sh/volcano -n volcano-system

```

## 4.2 初始化节点

对于工作节点，我们需要进行一些初始化操作，安装必要的软件包和依赖。首先创建一个 docker 镜像，包含我们需要的环境和依赖。

若宿主机想要与工作节点交流，需要暴露 api 端口，api 文件如下：

```

1  import os
2  import shutil
3  import zipfile
4  import subprocess
5  import psutil
6  from fastapi import FastAPI, UploadFile, Form
7  from fastapi.responses import JSONResponse, FileResponse
8  import uvicorn
9  import threading
10 from collections import deque
11
12 app = FastAPI()
13
14 # 工作空间路径
15 WORKSPACE = "/workspace"
16 os.makedirs(WORKSPACE, exist_ok=True)
17
18 # 保存子进程引用
19 process = None
20 log_buffer = deque(maxlen=1000) # 固定长度日志缓冲区
21 log_lock = threading.Lock()    # 线程锁，避免竞争
22
23 def stream_reader(pipe):
24     """后台线程：持续读取子进程输出到 log_buffer"""
25     global log_buffer
26     for line in iter(pipe.readline, b''):

```

```

27         with log_lock:
28             log_buffer.append(line.decode("utf-8"))
29     pipe.close()
30
31 @app.post("/clear-workspace/")
32 async def clear_workspace():
33     """清空 workspace (但是保留main.py、requirements.txt文件
        与data文件夹) """
34     if os.path.exists(WORKSPACE):
35         for item in os.listdir(WORKSPACE):
36             item_path = os.path.join(WORKSPACE, item)
37             if item not in ["main.py", "requirements.txt",
38                             "data"]:
39                 if os.path.isfile(item_path):
40                     os.remove(item_path)
41                 else:
42                     shutil.rmtree(item_path)
43     return {"status": "success", "msg": "Workspace cleared"}
44
45 @app.post("/upload-algo/")
46 async def upload_algo(file: UploadFile):
47     """下发算法: 清空 workspace + 解压 zip 文件"""
48     # 清空 workspace
49     if os.path.exists(WORKSPACE):
50         for item in os.listdir(WORKSPACE):
51             item_path = os.path.join(WORKSPACE, item)
52             if item not in ["main.py", "requirements.txt",
53                             "data"]:
54                 if os.path.isfile(item_path):
55                     os.remove(item_path)
56                 else:
57                     shutil.rmtree(item_path)
58
59     # 保存并解压
60     file_path = os.path.join(WORKSPACE, "algo.zip")
61     with open(file_path, "wb") as f:
62         f.write(await file.read())

```

```

61
62     with zipfile.ZipFile(file_path, "r") as zip_ref:
63         zip_ref.extractall(WORKSPACE)
64
65     os.remove(file_path)
66     return {"status": "success", "msg": "Algorithm
        uploaded and extracted"}
67
68 @app.get("/resource-usage/")
69 async def resource_usage():
70     """返回计算资源利用情况（CPU、内存）"""
71     cpu = psutil.cpu_percent(interval=1)
72     mem = psutil.virtual_memory().percent
73     return {"cpu": cpu, "memory": mem}
74
75 @app.get("/logs/")
76 async def get_logs(lines: int = 50):
77     """获取最近的终端输出（默认返回 50 行）"""
78     with log_lock:
79         logs = list(log_buffer)[-lines:]
80     return {"logs": "".join(logs)}
81
82 @app.post("/exec/")
83 async def exec_command(cmd: str = Form(...)):
84     """执行终端命令"""
85     global process, log_buffer
86     try:
87         # 清空旧日志
88         with log_lock:
89             log_buffer.clear()
90
91         process = subprocess.Popen(
92             cmd,
93             shell=True,
94             cwd=WORKSPACE,
95             stdout=subprocess.PIPE,
96             stderr=subprocess.STDOUT,
97             bufsize=1

```

```

98         )
99
100     # 启动后台线程，实时收集日志
101     threading.Thread(target=stream_reader, args=(
102         process.stdout,), daemon=True).start()
103
104     return {"status": "running", "cmd": cmd}
105 except Exception as e:
106     return JsonResponse(status_code=500, content={"
107         error": str(e)})
108
109 @app.post("/interrupt/")
110 async def interrupt():
111     """中断子进程"""
112     global process
113     if process:
114         process.terminate()
115         process = None
116         return {"status": "terminated"}
117     return {"status": "no process"}
118
119 @app.get("/list-files/")
120 async def list_files(path: str = ""):
121     """获得工作空间某个文件/文件夹索引"""
122     target_path = os.path.join(WORKSPACE, path)
123     if not os.path.exists(target_path):
124         return JsonResponse(status_code=404, content={"
125             error": "Path not found"})
126
127     if os.path.isfile(target_path):
128         return FileResponse(target_path)
129
130     files = os.listdir(target_path)
131     return {"files": files}
132
133 @app.get("/health/")
134 async def health():
135     """返回节点是否存活"""

```

```

133         return {"status": "alive"}
134
135     if __name__ == "__main__":
136         uvicorn.run(app, host="0.0.0.0", port=8000)

```

这个 api 文件实现了以下功能：

- 上传算法文件并解压到工作空间
- 清空工作空间（保留 main.py、requirements.txt 文件与 data 文件夹）
- 获取当前节点的 CPU 和内存使用情况
- 获取当前运行的命令的日志输出
- 执行指定的终端命令
- 中断当前运行的命令
- 列出工作空间中的文件和目录
- 获取指定文件的内容
- 健康检查，确认节点是否存活

由此，dockerfile 文件如下：

```

1  # 选择官方 PyTorch CUDA 镜像作为基础镜像
2  FROM pytorch/pytorch:latest
3
4  # 设置工作目录
5  WORKDIR /workspace
6
7  # 复制 requirements.txt 到容器
8  COPY requirements.txt .
9
10 # 安装 Python 依赖
11 RUN pip install -r requirements.txt
12
13 COPY main.py .

```



```
14
15 CMD ["bash", "-c", "python ../main.py; exec bash"]
```

其中, requirements.txt 文件包含以下内容:

```
1 pillow
2 tqdm
3 torch
4 torchvision
5 fastapi
6 uvicorn
7 psutil
```

这个镜像会在节点开始时就运行 api 服务  
然后运行以下命令构建 docker:

```
1 docker build -f dockerfile -t myenv-image:latest .
```

接下来, 我们需要在每个工作节点上运行这个镜像 (pod), 编写一个 volcano 任务配置文件:

```
1  # =====
2  # 训练节点 (worker1)
3  # =====
4  apiVersion: apps/v1
5  kind: Deployment
6  metadata:
7    name: mnist-train
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       app: mnist-train
13   template:
14     metadata:
15       labels:
16         app: mnist-train
17     spec:
18       nodeSelector:
19         kubernetes.io/hostname: kind-worker      # 绑定到
20         worker1
21     containers:
```

```

21         - name: train-container
22           image: myenv-image:latest
23           imagePullPolicy: IfNotPresent
24           ports:
25             - containerPort: 8000
26           volumeMounts:
27             - name: workspace
28               mountPath: /workspace    # 存放训练资源
29     volumes:
30       - name: workspace
31         emptyDir: {}
32     ---
33     apiVersion: v1
34     kind: Service
35     metadata:
36       name: mnist-train-service
37     spec:
38       selector:
39         app: mnist-train
40       type: NodePort
41       ports:
42         - port: 8000
43           targetPort: 8000
44           nodePort: 30081    # worker1 节点暴露端口
45     ---
46     # =====
47     # 预测节点 (worker2)
48     # =====
49     apiVersion: apps/v1
50     kind: Deployment
51     metadata:
52       name: mnist-predict
53     spec:
54       replicas: 1
55       selector:
56         matchLabels:
57           app: mnist-predict
58     template:

```

```

59     metadata:
60     labels:
61         app: mnist-predict
62     spec:
63     nodeSelector:
64         kubernetes.io/hostname: kind-worker2    # 绑定到
        worker2
65     containers:
66         - name: predict-container
67         image: myenv-image:latest
68         imagePullPolicy: IfNotPresent
69         ports:
70             - containerPort: 8000
71         volumeMounts:
72             - name: workspace
73               mountPath: /workspace    # 存放预测模型和上传图
               片
74     volumes:
75         - name: workspace
76           emptyDir: {}
77 ---
78 apiVersion: v1
79 kind: Service
80 metadata:
81 name: mnist-predict-service
82 spec:
83 selector:
84     app: mnist-predict
85 type: NodePort
86 ports:
87     - port: 8000
88       targetPort: 8000
89     nodePort: 30082    # worker2 节点暴露端口

```

这个配置文件定义了两个 Deployment 和两个 Service，分别用于训练和预测。每个 Deployment 都绑定到特定的工作节点，并运行我们之前创建的 Docker 镜像。

首先将镜像绑定到 kind 集群：

```
1 kind load docker-image myenv-image:latest --name dl-  
   cluster
```

这一步通常需要一些时间，完成后使用以下命令部署任务：

```
1 kubectl apply -f volcano-dl.yaml # 注意此处配置文件要修改  
   为自己的路径  
2 kubectl get pods
```

若部署成功，会看到 mnist-train 和 mnist-predict 两个 Pod 正在运行。

```
znh@znhvm:~/kind_volcano/MNIST$ kubectl get pods  
NAME                                READY   STATUS    RESTARTS   AGE  
mnist-predict-6fdb49f8c9-lgcww      1/1     Running   0           15s  
mnist-train-676f4b855-vj68d        1/1     Running   0           15s
```

至此，我们成功部署了一个简单的深度学习任务，包含训练和预测两个部分。接下来，我们可以通过暴露的端口与这些服务进行交互，上传数据并获取预测结果。

## 4.3 常用命令

这里列举一些常用与节点 api 交互的命令：

```
1 # 清空工作空间  
2 curl -X POST http://<NODE_IP>:<NODE_PORT>/clear-workspace/  
3  
4 # 上传算法文件  
5 curl -X POST "http://<NODE_IP>:<NODE_PORT>/upload-algo/" -  
   F "file=@/path/to/algo.zip"  
6  
7 # 获取资源使用情况  
8 curl http://<NODE_IP>:<NODE_PORT>/resource-usage/  
9  
10 # 获取最近的日志输出  
11 curl http://<NODE_IP>:<NODE_PORT>/logs/?lines=100  
12  
13 # 执行命令  
14 curl -X POST "http://<NODE_IP>:<NODE_PORT>/exec/" -F "cmd=  
   python main.py"
```

```

15
16 # 中断当前命令
17 curl -X POST http://<NODE_IP>:<NODE_PORT>/interrupt/
18
19 # 列出工作空间文件
20 curl http://<NODE_IP>:<NODE_PORT>/list-files/?path=/
    workspace
21
22 # 获取指定文件内容
23 curl http://<NODE_IP>:<NODE_PORT>/list-files/?path=/
    workspace/main.py -O
24
25 # 健康检查
26 curl http://<NODE_IP>:<NODE_PORT>/health/

```

## 一个例子：MNIST 手写数字识别任务

我们以 MNIST 手写数字识别任务为例，演示如何使用上述命令与节点交互，完成从上传算法到获取预测结果的全过程。

首先，我们需要准备好算法文件。假设我们已经编写好了 `train.py` 和 `predict.py` 两个脚本，并有 `mnist` 数据集和一些测试图片。

将数据集与训练脚本打包成一个 `zip` 文件 `train_files.zip`，上传到 `worker1` 节点：

```

1 zip -r train_files.zip train.py mnist
2 curl -X POST "http://<WORKER1_IP>:30081/upload-algo/" -F "
    file=@/path/to/train_files.zip"

```

上传完成后，执行训练命令：

```

1 curl -X POST "http://<WORKER1_IP>:30081/exec/" -F "cmd=
    python train.py"

```

训练过程中，可以实时查看日志输出，监控训练进度：

```

1 curl http://<WORKER1_IP>:30081/logs/?lines=100

```

训练完成后，模型会保存在工作空间中。获取这个模型文件：

```

1 curl http://<WORKER1_IP>:30081/list-files/?path=/workspace
    /model.pth -O

```

接下来，将模型文件、预测脚本和测试图片打包成 predict\_files.zip, 上传到 worker2 节点：

```
1 zip -r predict_files.zip predict.py model.pth test_images
2 curl -X POST "http://<WORKER2_IP>:30082/upload-algo/" -F "
    file=@/path/to/predict_files.zip"
```

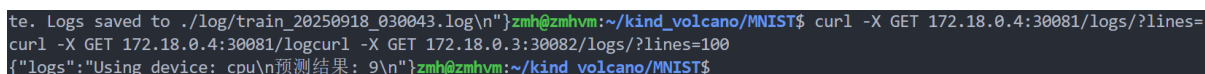
上传完成后，执行预测命令：

```
1 curl -X POST "http://<WORKER2_IP>:30082/exec/" -F "cmd=
    python predict.py"
```

预测过程中，同样可以查看日志输出，确认预测是否成功：

```
1 curl http://<WORKER2_IP>:30082/logs/?lines=100
```

下图是最后一步获取预测结果的截图：



```
te. Logs saved to ./log/train_20250918_030043.log\n"}znh@znhvm:~/kind_volcano/MNIST$ curl -X GET 172.18.0.4:30081/logs/?lines=
curl -X GET 172.18.0.4:30081/logcurl -X GET 172.18.0.3:30082/logs/?lines=100
{"logs":"Using device: cpu\n预测结果: 9\n"}znh@znhvm:~/kind_volcano/MNIST$
```

## 5 实验内容——二次开发：UI 化节点管理工具

对于 ui 化，我们选择 react 框架进行开发。主要功能包括（主要）：

- 节点管理：显示集群中所有节点的状态（在线/离线）、资源使用情况（CPU、内存、GPU 等）。
- 任务管理：列出当前运行的任务，支持任务的提交、暂停、取消等操作。
- 日志查看：实时查看节点和任务的日志输出，方便调试和监控。
- 文件管理：浏览和下载节点上的文件，支持文件的上传和删除。
- 命令执行：在节点上执行自定义命令，并查看执行结果。

我们仍对后端 api 做出一个整合，方便前端调用：

```
1 from fastapi import FastAPI
2 from fastapi import Query
3 from fastapi import Body
```

```

4  from kubernetes import client, config
5  import subprocess
6  import datetime
7
8
9  app = FastAPI()
10
11  @app.get("/k8s-clusters")
12  def get_k8s_clusters():
13      from kubernetes import config
14      contexts, active_context = config.
          list_kube_config_contexts()
15      clusters = [c['name'] for c in contexts]
16      return {"clusters": clusters, "active": active_context
          ['name']}
17
18
19  @app.get("/nodes")
20  def get_nodes():
21      """
22      获取所有节点信息
23      """
24      config.load_kube_config() # 如果在集群外运行
25      v1 = client.CoreV1Api()
26      nodes = []
27      for item in v1.list_node().items:
28          node_info = {
29              "NAME": item.metadata.name,
30              "STATUS": item.status.conditions[-1].type if
                  item.status.conditions else "",
31              "ROLES": item.metadata.labels.get("kubernetes.
                  io/role", ""),
32              "VERSION": item.status.node_info.
                  kubelet_version,
33              "INTERNAL-IP": next((addr.address for addr in
                  item.status.addresses if addr.type == "
                  InternalIP"), ""),
34              "EXTERNAL-IP": next((addr.address for addr in

```

```

        item.status.addresses if addr.type == "
        ExternalIP"), ""),
35     "OS-IMAGE": item.status.node_info.os_image,
36     "KERNEL-VERSION": item.status.node_info.
        kernel_version,
37     "CONTAINER-RUNTIME": item.status.node_info.
        container_runtime_version,
38     }
39     nodes.append(node_info)
40     return {"nodes": nodes}
41
42 @app.get("/cluster-nodes")
43 def get_cluster_nodes(cluster: str = Query(...)):
44     """
45     根据集群名称（context）获取该集群的所有节点信息
46     """
47     contexts, active_context = config.
        list_kube_config_contexts()
48     context_names = [c['name'] for c in contexts]
49     if cluster not in context_names:
50         return {"error": f"集群 {cluster} 不存在"}
51     config.load_kube_config(context=cluster)
52     v1 = client.CoreV1Api()
53     nodes = []
54     for item in v1.list_node().items:
55         node_info = {
56             "NAME": item.metadata.name,
57             "STATUS": item.status.conditions[-1].type if
                item.status.conditions else "",
58             "ROLES": item.metadata.labels.get("kubernetes.
                io/role", ""),
59             "VERSION": item.status.node_info.
                kubelet_version,
60             "INTERNAL-IP": next((addr.address for addr in
                item.status.addresses if addr.type == "
                InternalIP"), ""),
61             "EXTERNAL-IP": next((addr.address for addr in
                item.status.addresses if addr.type == "

```



```

        ExternalIP"), ""),
62         "OS-IMAGE": item.status.node_info.os_image,
63         "KERNEL-VERSION": item.status.node_info.
            kernel_version,
64         "CONTAINER-RUNTIME": item.status.node_info.
            container_runtime_version,
65     }
66     nodes.append(node_info)
67     return {"nodes": nodes}
68
69
70
71
72 # 查询当前主机上的所有 Docker 镜像，返回镜像名:tag、镜像ID
    # 创建时间、大小等信息
73 @app.get("/docker-images")
74 def get_docker_images():
75     try:
76         result = subprocess.run(
77             ["docker", "images", "--format", "{{.
                Repository}}:{{.Tag}} {{.ID}} {{.CreatedAt}}
                {{.Size}}"],
78             capture_output=True, text=True, check=True
79         )
80         images = []
81         for line in result.stdout.strip().split("\n"):
82             parts = line.split()
83             if len(parts) >= 4:
84                 image_info = {
85                     "REPOSITORY:TAG": parts[0],
86                     "IMAGE ID": parts[1],
87                     "CREATED AT": " ".join(parts[2:-1]),
88                     "SIZE": parts[-1]
89                 }
90                 images.append(image_info)
91         return {"images": images}
92     except subprocess.CalledProcessError as e:
93         return {"error": str(e), "output": e.output, "

```

```

        "stderr": e.stderr}

94
95 @app.post("/load-image-to-cluster")
96 def load_image_to_cluster(
97     image: str = Body(..., embed=True),
98     cluster: str = Body(..., embed=True)
99 ):
100     """
101     将本地 Docker 镜像加载到指定 kind 集群
102     """
103     #去掉前置可能存在的kind, 如: kind-dl-cluster变为dl-
        cluster
104     cluster = cluster.replace("kind-", "")
105
106     try:
107         result = subprocess.run(
108             ["kind", "load", "docker-image", image, "--",
                "name", cluster],
109             capture_output=True, text=True, check=True
110         )
111         return {"success": True, "output": result.stdout,
            "stderr": result.stderr}
112     except subprocess.CalledProcessError as e:
113         return {"success": False, "error": str(e), "output":
            e.output, "stderr": e.stderr}
114
115
116
117
118 # 获取所有Pod列表
119 @app.get("/pods")
120 def get_pods():
121     pods = []
122     v1 = client.CoreV1Api()
123     ret = v1.list_pod_for_all_namespaces(watch=False)
124     for item in ret.items:
125         pod_info = {
126             "NAMESPACE": item.metadata.namespace,

```

```

127         "NAME": item.metadata.name,
128         "READY": f"{sum(1 for c in item.status.
            container_statuses if c.ready)}/{len(item.
            status.container_statuses)}" if item.status.
            container_statuses else "0/0",
129         "STATUS": item.status.phase,
130         "RESTARTS": sum(c.restart_count for c in item.
            status.container_statuses) if item.status.
            container_statuses else 0,
131         "AGE": str(item.metadata.creation_timestamp)
132     }
133     pods.append(pod_info)
134     return {"pods": pods}
135
136 # 获取指定Pod的日志
137 @app.get("/pod-logs")
138 def get_pod_logs(
139     pod_name: str = Query(..., description="Pod名称"),
140     namespace: str = Query("default", description="命名空
        间名称"),
141     container: str = Query(None, description="容器名称（如
        果Pod中有多个容器）"),
142     tail_lines: int = Query(100, description="返回的日志行
        数", ge=1, le=5000)
143 ):
144     """
145     获取指定Pod的日志
146     """
147     try:
148         config.load_kube_config() # 如果在集群外运行
149         v1 = client.CoreV1Api()
150
151         # 获取Pod信息，检查是否存在
152         try:
153             pod = v1.read_namespaced_pod(name=pod_name,
                namespace=namespace)
154         except client.exceptions.ApiException as e:
155             if e.status == 404:

```

```

156         return {"error": f"Pod {pod_name} 在命名空  

157             间 {namespace} 中不存在"}
158
159     raise e
160
161     # 如果没有指定容器，但Pod有多个容器，则使用第一个
162     容器
163     if not container and pod.spec.containers and len(
164         pod.spec.containers) > 1:
165         container = pod.spec.containers[0].name
166
167     # 获取日志
168     logs = v1.read_namespaced_pod_log(
169         name=pod_name,
170         namespace=namespace,
171         container=container,
172         tail_lines=tail_lines
173     )
174
175     # 将日志按行分割
176     log_lines = logs.split('\n') if logs else []
177
178     return {
179         "pod": pod_name,
180         "namespace": namespace,
181         "container": container,
182         "log_lines": log_lines,
183         "total_lines": len(log_lines)
184     }
185
186     except client.exceptions.ApiException as e:
187         return {"error": f"API 错误: {str(e)}"}
188     except Exception as e:
189         return {"error": f"获取日志失败: {str(e)}"}
190
191 # 获取所有deployment列表
192 @app.get("/deployments")
193 def get_deployments():

```

```

191     apps_v1 = client.AppsV1Api()
192     deployments = []
193     ret = apps_v1.list_deployment_for_all_namespaces(watch
        =False)
194     for item in ret.items:
195         deploy_info = {
196             "NAMESPACE": item.metadata.namespace,
197             "NAME": item.metadata.name,
198             "READY": f"{item.status.ready_replicas}/{item.
                status.replicas}" if item.status.replicas
                else "0/0",
199             "UP-TO-DATE": item.status.updated_replicas if
                item.status.updated_replicas else 0,
200             "AVAILABLE": item.status.available_replicas if
                item.status.available_replicas else 0,
201             "AGE": str(item.metadata.creation_timestamp)
202         }
203         deployments.append(deploy_info)
204     return {"deployments": deployments}
205
206     # 获取所有 service 列表
207     @app.get("/services")
208     def get_services():
209         services = []
210         v1 = client.CoreV1Api()
211         ret = v1.list_service_for_all_namespaces(watch=False)
212         for item in ret.items:
213             ports = ", ".join([f"{p.port}:{p.node_port}" if p.
                node_port else str(p.port) for p in item.spec.
                ports])
214             svc_info = {
215                 "NAMESPACE": item.metadata.namespace,
216                 "NAME": item.metadata.name,
217                 "TYPE": item.spec.type,
218                 "CLUSTER-IP": item.spec.cluster_ip,
219                 "EXTERNAL-IP": item.status.load_balancer.
                    ingress[0].ip if item.status.load_balancer
                    and item.status.load_balancer.ingress else "

```

```

                ",
220             "PORT(S)": ports,
221             "AGE": str(item.metadata.creation_timestamp)
222         }
223         services.append(svc_info)
224     return {"services": services}
225
226 # 停止全部Pod
227 @app.post("/stop-all-pods")
228 def stop_all_pods():
229     v1 = client.CoreV1Api()
230     ret = v1.list_namespaced_pod(namespace="default",
231                                 watch=False)
232     for item in ret.items:
233         try:
234             v1.delete_namespaced_pod(name=item.metadata.
235                                     name, namespace="default")
236         except Exception as e:
237             return {"error": str(e)}
238     return {"status": "Default namespace pods deletion
239             initiated"}
240
241 # 停止全部deployment
242 @app.post("/stop-all-deployments")
243 def stop_all_deployments():
244     apps_v1 = client.AppsV1Api()
245     ret = apps_v1.list_namespaced_deployment(namespace="
246         default", watch=False)
247     for item in ret.items:
248         try:
249             apps_v1.delete_namespaced_deployment(name=item
250             .metadata.name, namespace="default")
251         except Exception as e:
252             return {"error": str(e)}
253     return {"status": "Default namespace deployments
254             deletion initiated"}
255
256 # 停止全部service

```

```

251 @app.post("/stop-all-services")
252 def stop_all_services():
253     v1 = client.CoreV1Api()
254     ret = v1.list_namespaced_service(namespace="default",
255                                     watch=False)
256     for item in ret.items:
257         if item.metadata.name == "kubernetes":
258             continue # 不删除默认的 kubernetes 服务
259         try:
260             v1.delete_namespaced_service(name=item.
261                                         metadata.name, namespace="default")
262         except Exception as e:
263             return {"error": str(e)}
264     return {"status": "Default namespace services deletion
265             initiated"}
266
267 # 创建vc任务，需要提交yaml文件
268 from fastapi import File, UploadFile
269
270 # 创建vc任务，需要上传yaml文件（multipart/form-data）
271 @app.post("/create-vc-job")
272 def create_vc_job(file: UploadFile = File(...)):
273     import yaml
274     from kubernetes.utils import create_from_dict
275     try:
276         content = file.file.read().decode()
277         docs = list(yaml.safe_load_all(content))
278         created = 0
279         for data in docs:
280             if data:
281                 create_from_dict(client.ApiClient(), data)
282                 created += 1
283         return {"status": f"{created} document(s) created
284                 successfully"}
285     except Exception as e:
286         return {"error": str(e)}
287
288 # 检查节点状态

```

```

285 # 通过节点内部的api端口访问
286 # IP、端口号由用户给出
287 @app.get("/check-node-status")
288 def check_node_status(ip: str = Query(...), port: int =
    Query(...)):
289     import requests
290     try:
291         url = f"http://{ip}:{port}/health/"
292         response = requests.get(url, timeout=5)
293         if response.status_code == 200:
294             return {"status": "reachable", "data":
                response.json()}
295         else:
296             return {"status": "unreachable", "http_status":
                response.status_code}
297     except Exception as e:
298         return {"status": "error", "error": str(e)}
299
300 # 清空workspace
301 # 通过节点内部的api端口访问
302 # IP、端口号由用户给出
303 @app.post("/clear-workspace")
304 def clear_workspace(ip: str = Body(...), port: int = Body
    (...)):
305     import requests
306     try:
307         url = f"http://{ip}:{port}/clear-workspace/"
308         response = requests.post(url, timeout=10)
309         if response.status_code == 200:
310             return {"status": "success", "data": response.
                json()}
311         else:
312             return {"status": "failed", "http_status":
                response.status_code, "data": response.text}
313     except Exception as e:
314         return {"status": "error", "error": str(e)}
315
316 # 下发算法

```



```

317 # 通过节点内部的api端口访问
318 # IP、端口号由用户给出
319 @app.post("/upload-algo")
320 def upload_algo(ip: str = Body(...), port: int = Body(...)
    , file: UploadFile = File(...)):
321     import requests
322     try:
323         url = f"http://{ip}:{port}/upload-algo/"
324         files = {'file': (file.filename, file.file, file.
            content_type)}
325         response = requests.post(url, files=files, timeout
            =30)
326         if response.status_code == 200:
327             return {"status": "success", "data": response.
                json()}
328         else:
329             return {"status": "failed", "http_status":
                response.status_code, "data": response.text}
330     except Exception as e:
331         return {"status": "error", "error": str(e)}
332
333 # 返回节点的计算资源利用情况
334 # 通过节点内部的api端口访问
335 # IP、端口号由用户给出
336 @app.get("/get-resource-usage")
337 def get_resource_usage(ip: str = Query(...), port: int =
    Query(...)):
338     import requests
339     try:
340         url = f"http://{ip}:{port}/resource-usage/"
341         response = requests.get(url, timeout=5)
342         if response.status_code == 200:
343             return {"status": "success", "data": response.
                json()}
344         else:
345             return {"status": "failed", "http_status":
                response.status_code, "data": response.text}
346     except Exception as e:

```

```

347         return {"status": "error", "error": str(e)}
348
349 # 获取最近的终端输出
350 # 通过节点内部的api端口访问
351 # IP、端口号由用户给出
352 @app.get("/get-logs")
353 def get_logs(ip: str = Query(...), port: int = Query(...),
354             lines: int = Query(50)):
355     import requests
356     try:
357         url = f"http://{ip}:{port}/logs/?lines={lines}"
358         response = requests.get(url, timeout=5)
359         if response.status_code == 200:
360             return {"status": "success", "data": response.
361                     json()}
362         else:
363             return {"status": "failed", "http_status":
364                     response.status_code, "data": response.text}
365     except Exception as e:
366         return {"status": "error", "error": str(e)}
367
368 # 执行终端命令
369 # 通过节点内部的api端口访问
370 # IP、端口号由用户给出
371 @app.post("/exec-command")
372 def exec_command(ip: str = Body(...), port: int = Body
373                 (...), cmd: str = Body(...)):
374     import requests
375     try:
376         url = f"http://{ip}:{port}/exec/"
377         data = {'cmd': cmd}
378         response = requests.post(url, data=data, timeout
379                                 =10)
380         if response.status_code == 200:
381             return {"status": "success", "data": response.
382                     json()}
383         else:
384             return {"status": "failed", "http_status":

```

```

        response.status_code, "data": response.text}
379     except Exception as e:
380         return {"status": "error", "error": str(e)}
381
382     # 中断子进程
383     # 通过节点内部的api端口访问
384     # IP、端口号由用户给出
385     @app.post("/interrupt-process")
386     def interrupt_process(ip: str = Body(...), port: int =
        Body(...)):
387         import requests
388         try:
389             url = f"http://{ip}:{port}/interrupt/"
390             response = requests.post(url, timeout=5)
391             if response.status_code == 200:
392                 return {"status": "success", "data": response.
                    json()}
393             else:
394                 return {"status": "failed", "http_status":
                    response.status_code, "data": response.text}
395         except Exception as e:
396             return {"status": "error", "error": str(e)}
397
398     # 获得工作空间某个文件
399     # 或者文件夹内容索引
400     # 通过节点内部的api端口访问
401     # IP、端口号由用户给出
402     '''
403     @app.get("/list-files/")
404     async def list_files(path: str = ""):
405         """获得工作空间某个文件/文件夹索引"""
406         target_path = os.path.join(WORKSPACE, path)
407         if not os.path.exists(target_path):
408             return JSONResponse(status_code=404, content={"
                error": "Path not found"})
409
410         if os.path.isfile(target_path):
411             return FileResponse(target_path)

```

```

412
413     files = os.listdir(target_path)
414     return {"files": files}
415 '''
416 from fastapi.responses import Response
417
418 from fastapi.responses import Response
419
420 @app.get("/list-files")
421 def list_files(ip: str = Query(...), port: int = Query(
    (...), path: str = Query(")):
422     import requests
423     try:
424         url = f"http://{ip}:{port}/list-files/?path={path}"
425
426         response = requests.get(url, timeout=10)
427         if response.status_code == 200:
428             content_type = response.headers.get("content-
429                 type", "")
430             # 如果远程返回的是文件内容，直接返回原始内容
431             if content_type.startswith("application/octet-
432                 stream") or content_type.startswith("text/")
433                 :
434                 return Response(content=response.content,
435                     media_type=content_type)
436             # 如果是 JSON，则解析并返回
437             return {"status": "success", "data": response.
438                 json()}
439         else:
440             return {"status": "failed", "http_status":
441                 response.status_code, "data": response.text}
442     except Exception as e:
443         return {"status": "error", "error": str(e)}
444
445 if __name__ == "__main__":
446     import uvicorn
447     uvicorn.run(app, host="0.0.0.0", port=8500)

```

以下是部分界面截图：

Volcano K8s  
集群管理平台

仪表盘

集群管理

节点管理

Pod管理

部署管理

服务管理

Docker镜像

文件浏览器

当前集群: kind-di-cluster

Volcano K8s 管理平台

清理端口缓存

资源监控

刷新

1  
集群数量

3  
节点数量

15  
Pod数量

4  
服务数量

快速操作

创建 Volcano 任务

资源监控

执行命令

Volcano K8s  
集群管理平台

仪表盘

集群管理

节点管理

Pod管理

部署管理

服务管理

Docker镜像

文件浏览器

当前集群: kind-di-cluster

节点管理

自动刷新

刷新节点

节点资源概览

di-cluster-control-plane

Ready

CPU0.0%

内存0.0%

di-cluster-worker

Ready

CPU1.2%

内存25.3%

di-cluster-worker2

Ready

CPU1.0%

内存25.3%

集群节点

节点名称	状态	CPU使用率	内存使用率	内部IP	操作
di-cluster-control-plane	Ready	0.0%	0.0%	172.18.0.2	详情管理

## Pod 管理

停止所有Pod 刷新

Pod 列表						命名空间: 所有命名空间
命名空间	POD名称	就绪状态	运行状态	重启次数	操作	
default	mnist-predict-6fdb49f8c9-6sstd	1/1	Running	0	日志 删除	
default	mnist-train-676f4b855-htzhz	1/1	Running	0	日志 删除	
kube-system	coredns-7db6d8ff4d-826nr	1/1	Running	1	日志 删除	
kube-system	coredns-7db6d8ff4d-842cv	1/1	Running	1	日志 删除	
kube-system	etcd-dl-cluster-control-plane	1/1	Running	0	日志 删除	
kube-system	kindnet-5l2jb	1/1	Running	1	日志 删除	
kube-system	kindnet-g9wm7	1/1	Running	1	日志 删除	
kube-system	kindnet-tpc6n	1/1	Running	1	日志 删除	
kube-system	kube-apiserver-dl-cluster-control-plane	1/1	Running	0	日志 删除	
kube-system	kube-controller-manager-dl-cluster-control-plane	1/1	Running	3	日志 删除	

## 部署管理

创建部署 刷新 停止所有部署

命名空间	名称	就绪	最新	可用	创建时间	操作
default	mnist-predict	1/1	1	1	2025/9/18 21:55:10	🔍 📄 🗑️
default	mnist-train	1/1	1	1	2025/9/18 21:55:10	🔍 📄 🗑️
kube-system	coredns	2/2	2	2	2025/9/17 22:02:39	🔍 📄 🗑️
local-path-storage	local-path-provisioner	1/1	1	1	2025/9/17 22:02:40	🔍 📄 🗑️

Volcano K8s

集群管理平台

仪表板

集群管理

节点管理

Pod管理

部署管理

服务管理

Docker镜像

文件浏览器

当前集群: kind-di-cluster

服务管理

停止所有服务

刷新

4 总服务数

0 负载均衡器

2 NodePort

2 命名空间

服务列表

命名空间: 所有命名空间

命名空间	服务名称	类型	集群IP	外部IP	端口	创建时间
default	kubernetes	ClusterIP	10.96.0.1	无	443	2025/9/18
default	mnist-predict-service	NodePort	10.96.232.159	无	8000:30082	2025/9/18
default	mnist-train-service	NodePort	10.96.201.139	无	8000:30081	2025/9/18
kube-system	kube-dns	ClusterIP	10.96.0.10	无	53, 53, 9153	2025/9/17

Volcano K8s

集群管理平台

仪表板

集群管理

节点管理

Pod管理

部署管理

服务管理

Docker镜像

文件浏览器

当前集群: kind-di-cluster

Docker 镜像管理

刷新镜像

搜索镜像名称...

共 5 个镜像

加载镜像到集群

选择镜像

选择集群

选择镜像...

选择集群...

加载到集群

本地 Docker 镜像

镜像名称: 标签	镜像ID	创建时间	大小	操作
myenv-image:latest	6779c04b32a9...	2025-09-18 02:53:29 +0000 UTC	7.62GB	选择
<none>:<none>	7a8b83e8a165...	2025-09-18 01:43:02 +0000 UTC	7.62GB	选择
<none>:<none>	7c31fe83ceb0...	2025-09-17 14:34:47 +0000 UTC	7.62GB	选择
<none>:<none>	bd64ba3837d2...	2025-09-17 14:20:29 +0000 UTC	7.62GB	选择



## 6 实验总结

本实验通过使用 kind 和 volcano 在本地搭建了一个简易的深度学习集群环境，并实现了节点的 API 化管理，最后开发了一个 UI 化的节点管理工具。具体总结如下：

- **环境搭建：**通过 kind 快速创建了一个包含多个工作节点的 Kubernetes 集群，并使用 volcano 实现了对深度学习任务的调度和管理。整个过程简便高效，适合本地测试和开发。
- **节点 API 化：**为每个工作节点编写了一个基于 FastAPI 的 API 服务，实现了算法上传、工作空间管理、资源监控、日志查看等功能。这些 API 为后续的自动化管理和任务调度提供了基础。
- **UI 化管理工具：**使用 React 框架开发了一个简单的前端界面，集成了节点管理、任务管理、日志查看等功能。通过调用后端 API，实现了对集群和节点的可视化管理，提升了用户体验。
- **实践与挑战：**在实验过程中，深入理解了 Kubernetes 和 volcano 的基本概念和操作，掌握了容器化应用的部署与管理。同时也遇到了



一些挑战，如网络配置、权限管理等，但通过查阅文档和社区资源，成功解决了这些问题。

总体而言，本实验不仅提升了对容器编排和深度学习任务管理的理解，也锻炼了实际动手能力。未来可以进一步扩展功能，如支持更多类型的资源监控、集成更多的调度策略、实际环境中的部署等。