ALIBABA CLOUD

阿里云

容器服务Kubernetes版

解决方案

文档版本：20201029

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.AI解决方案

1.1. 概述

基于阿里云强大计算能力和开源Kubeflow，ACK深度学习解决方案提供一个低门槛、开放、端到端的深度学习服务平台，将深度学习能力转化为服务API，加速与业务应用的集成。ACK深度学习解决方案方便数据和算法工程师利用阿里云的资源执行数据准备、模型开发、模型训练、评估和预测等任务。

特性

ACK深度学习解决方案具备以下特性：

- 简单：降低构建和管理深度学习平台的门槛。
- 高性能：提升CPU、GPU等异构计算资源的性能和使用效率。
- 开放：支持TensorFlow、PyTorch等多种主流深度学习框架，支持使用定制的环境，所有方案中的工具全部开源。
- 全周期：提供基于阿里云强大服务体系构建端到端深度学习任务流程的最佳实践。
- 服务化：支持深度学习能力服务化，与云上应用的轻松集成。
- 多用户：支持数据科学家团队的协同合作。

1.2. 环境准备

1.2.1. 通过组件安装最新版的Arena

Arena是基于Kubernetes的机器学习轻量级解决方案，支持数据准备，模型开发，模型训练，模型预测的完整生命周期，提升数据科学家工作效率。同时和阿里云的基础云服务深度集成，支持GPU共享、CPFS等服务，可以运行阿里云优化的深度学习框架，最大化使用阿里云异构设备的性能和成本的效益。

前提条件

- 创建包含GPU的Kubernetes集群，请参见Kubernetes GPU集群支持GPU调度。
- 集群节点可以访问公网，请参见通过公网访问集群API Server。

操作步骤

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面，选择目标集群，并在目标集群右侧**操作**列下，选择**更多 > 系统组件管理**。
4. 找到组件**ack-arena**，单击**安装**。

配置Arena客户端

如果您使用的是专有版集群，以SSH方式登录到专有版集群的管理节点上，然后直接执行**arena**命令。有关使用SSH方式登录集群具体步骤，请参见通过SSH访问Kubernetes集群。

如果您使用的是托管版集群，需使用kubectl命令行工具先连接集群，以确保*kubeconfig*文件在机器上的正确位置是*$HOME/.kube/config*，具体步骤请参见通过kubectl连接Kubernetes集群。然后配置Arena客户端，具体操作步骤如下。

> **说明** 您可以通过执行命令 `kubectl get nodes` 判断*kubeconfig*文件是否正确配置。

1. 下载Arena客户端。
   - Linux版
   - Mac版
2. 解压安装包。
   - 对于Linux版系统，执行以下命令解压安装包。

     ```
     tar -xvf arena-installer-0.5.0-e22162d-linux-amd64.tar.gz
     ```

   - 对于Mac版系统，执行以下命令解压安装包。

     ```
     tar -xvf arena-installer-0.5.0-e22162d-darwin-amd64.tar.gz
     ```

3. 执行以下命令安装Arena。

   ```
   cd arena-installer
   ./install.sh
   ```

4. （可选）安装自动补齐软件。安装自动补齐软件支持命令参数自动提示功能，帮助您避免记忆命令行。
   - 执行以下命令安装CentOS或Alibaba Cloud Linux 2。

     ```
     yum install bash-completion -y
     ```

   - 执行以下命令安装Debian或Ubuntu。

     ```
     apt-get install bash-completion
     ```

   - 执行以下命令安装MacOS。

     ```
     brew install bash-completion@2
     ```

5. 执行以下命令在*profile*文件中增加自动补齐功能。
   - Linux

     ```
     echo "source <(arena completion bash)" >> ~/.bashrc
     chmod u+x ~/.bashrc
     ```

   - MacOS

     ```
     echo "source $(brew --prefix)/etc/profile.d/bash_completion.sh" >> ~/.bashrc
     ```

     在命令行终端通过**Tab**键即可自动补齐命令。

验证Arena

您可以执行以下步骤验证Arena是否正常工作。

1. 执行以下命令检查集群的可用GPU资源。

   ```
   arena top node
   ```

   看到输出为节点和GPU卡的信息，代表输出正确。

```
NAME                IPADDRESS     ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx7  192.1xx.x.xx7 <none>  ready  8        0
cn-huhehaote.192.1xx.x.xx8  192.168.0.118 <none>  ready  8        0
cn-huhehaote.192.1xx.x.xx9  192.168.0.119 <none>  ready  8        0
cn-huhehaote.192.1xx.x.xx0  192.168.0.120 <none>  ready  8        0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
0/32 (0%)
```

2. 通过**arena**提交一个训练作业， 看到任务被成功提交。

```
arena submit tf \
    --name=firstjob \
    --gpus=1 \
    --image=registry.cn-hangzhou.aliyuncs.com/tensorflow-samples/tf-mnist-standalone:gpu \
    "python /app/main.py"
```

```
configmap/firstjob-tfjob created
configmap/firstjob-tfjob labeled
tfjob.kubeflow.org/firstjob created
INFO[0001] The Job firstjob has been submitted successfully
INFO[0001] You can run `arena get firstjob --type tfjob` to check the job status
```

3. 执行 `arena list` 命令列出所有作业。输出如下。

```
NAME    STATUS   TRAINER  AGE  NODE
firstjob  RUNNING  TFJOB   5s   192.1xx.x.xxx
```

4. 执行以下命令查看提交作业的状态。

```
arena get firstjob
```

```
STATUS: SUCCEEDED
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 52s
NAME    STATUS    TRAINER  AGE  INSTANCE       NODE
firstjob  SUCCEEDED  TFJOB   14m  firstjob-chief-0  192.168.0.118
```

5. 执行以下命令查看作业日志。

```
arena logs --tail=10 firstjob
```

```
Accuracy at step 910: 0.9694
Accuracy at step 920: 0.9687
Accuracy at step 930: 0.9676
Accuracy at step 940: 0.9678
Accuracy at step 950: 0.9704
Accuracy at step 960: 0.9692
Accuracy at step 970: 0.9721
Accuracy at step 980: 0.9696
Accuracy at step 990: 0.9675
Adding run metadata for 999
```

1.2.2. 配置NAS共享存储

阿里云文件存储NAS（Network Attached Storage）是面向阿里云ECS实例、E-HPC和ACK等计算节点的文件存储服务。阿里云NAS服务具有无缝集成，共享访问，安全控制等特性，非常适合跨多个ECS、E-HPC或ACK实例部署的应用程序访问相同数据来源的应用场景。文本介绍如何配置NAS共享存储。

背景信息

为了保留数据科学家的工作内容，或者读取同一份训练数据。建议您配置共享存储卷，并挂载到Arena提交作业的运行环境中，确保数据科学家的工作内容（代码、数据）得以保留，不会随着容器删除而丢失。

在团队开发中，建议分配一个共享的存储池，让数据和代码能够在团队中共享。在Arena提交作业使用--data参数时，如果您声明了配置共享存储，以及要挂载到运行环境的路径，共享存储将会被挂载到您指定的目录中，这样提交作业就可以复用这部分数据或者代码。

在Kubernetes中，通过存储卷（PV）和存储声明（PVC）描述存储对象。 作为集群管理员，分配环境时您需要为每个数据科学家创建属于自己的存储声明。例如用户A和用户B，存储声明的后端可以挂载到相同的NAS或者CPFS，但是必须指定不同的子目录，保证他们工作环境隔离。

步骤一：创建NAS实例

有关创建NAS实例的具体操作步骤，请参见创建通用型NAS文件系统。

> **说明**
> 创建NAS实例的参数配置说明如下：
> - 文件系统类型设置为**通用型**。
> - 地域设置为和ACK集群对应的地域。
> - 协议类型设置为NFS。

步骤二：配置挂载点

有关配置挂载点的具体操作步骤，请参见添加挂载点。

> **说明**
> 添加挂载点的参数配置说明如下：
> - 挂载点类型设置为**专有网络**。
> - VPC网络和交换机设置为和ACK集群一致的VPC和交换机。

创建成功后，在**挂载点**页面，悬浮鼠标至，查看NAS实例的挂载地址。

步骤三：配置ACK集群的存储卷（PV）和存储声明（PVC）

1. 登录容器服务管理控制台。
2. 创建存储卷（PV）。创建存储卷的具体步骤请参见创建PV。

   > **说明** 创建存储卷配置参数时，设置**挂载点域名**为**选择挂载点**，并选择NAS的挂载点地址。

3. 创建存储声明（PVC）。创建存储声明的具体步骤请参见创建PVC。
   完成创建后，在存储声明 (PVC) 的列表中，可以看到刚刚创建的存储声明 (PVC) 实例。

步骤四：给PVC填充数据

因为Kubernetes集群通过PVC访问各种共享数据（也就是本文在第一步中创建的NAS实例），所以您仅需要给PVC实例对应的NAS实例上填充数据即可。

1. 执行以下命令登录ACK任意一台ECS节点。

   ```
   ssh root@39.10x.xx.xx
   ```

   > **说明** ECS节点VPC网络需和创建的NAS实例VPC一致。

2. 在命令行终端，粘贴并执行NAS挂载命令。NAS挂载命令表示将NAS实例挂载到当前登录的ECS节点下的*/mnt*路径中。

   

   ```
   sudo mount -t nfs -o xxxx.nas.aliyuncs.com:/ /mnt
   ```

3. 执行以下命令，通过挂载的这个目录为NAS实例创建两个目录*tf_data/*和*pytorch_data/*，分别存放TF mnist和Pytorch mnist的训练数据。

   ```
   cd /mnt/
   mkdir tf_data/
   mkdir pytorch_data/
   ```

4. 执行以下命令下载TF mnist的数据集。

   ```
   cd tf_data
   git clone https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git
   mv tensorflow-sample-code/data/* ./ && rm -rf tensorflow-sample-code
   ```

5. 执行以下命令下载Pytorch mnist的数据集。

   ```
   cd pytorch_data
   git clone https://code.aliyun.com/370272561/mnist-pytorch.git
   mv mnist-pytorch/MNIST ./ && rm -rf mnist-pytorch
   ```

1.2.3. 配置CPFS共享存储

阿里云文件存储CPFS（Cloud Paralleled File System）是一种并行文件系统。CPFS的数据存储在集群中的多个数据节点，并可由多个客户端同时访问，从而能够为大型高性能计算机集群提供高IOPS、高吞吐、低时延的数据存储服务。文本介绍如何配置CPFS共享存储。

背景信息

为了保留数据科学家的工作内容，或者读取同一份训练数据。建议您配置共享存储卷，并挂载到Arena提交作业的运行环境中，确保数据科学家的工作内容（代码、数据）得以保留，不会随着容器删除而丢失。

在团队开发中，建议分配一个共享的存储池，让数据和代码能够在团队中共享。在Arena提交作业使用--data参数时，如果您声明了配置共享存储，以及要挂载到运行环境的路径，共享存储将会被挂载到您指定的目录中，这样提交作业就可以复用这部分数据或者代码。

在Kubernetes中，通过存储卷（PV）和存储声明（PVC）描述存储对象。 作为集群管理员，分配环境时您需要为每个数据科学家创建属于自己的存储声明。例如用户A和用户B，存储声明的后端可以挂载到相同的NAS或者CPFS，但是必须指定不同的子目录，保证他们工作环境隔离。

步骤一：创建CPFS实例

有关创建CPFS实例的具体操作步骤，请参见创建文件系统。

> **说明**
> 创建CPFS实例的参数配置，例如地域、专有网络、虚拟交换机，需和ACK集群的参数配置一致。

步骤二：配置挂载点

1. 登录NAS控制台。
2. 在控制台左侧导航栏，选择**文件系统 > 文件系统列表**。
3. 在目标CPFS实例右侧的**操作**列下，单击**管理**。
   在**基本信息**页签，查看文件系统ID和挂载点信息。

步骤三：配置ACK集群的存储卷（PV）和存储声明（PVC）

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**工作负载**。
5. 在**无状态（Deployment）**页面，单击右上角**使用模板创建**。设置集群名称和命名空间、设置示例模板为Resource-PersistentVolumeClaim。
   模板内容如下。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-cpfs
  labels:
    alicloud-pvname: pv-cpfs
spec:
  capacity:
    storage: 500Gi
  accessModes:
    - ReadWriteMany
  flexVolume:
    driver: "alicloud/cpfs"
    options:
      server: "cpfs-xxxxx.cn-huhehaote.cpfs.nas.aliyuncs.com@tcp:/08fba53c"
      fileSystem: "cpfs-08fba53c"
      subPath: "tf_data"

---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pv-cpfs
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Gi
  selector:
    matchLabels:
      alicloud-pvname: pv-cpfs
```

> **说明**
> PV和PVC的配置说明如下：
>
> - server：填写步骤二中获取的CPFS挂载点信息。
> - fileSystem：填写CPFS文件系统ID。
> - subPath：填写tf_data。

6. 单击**创建**。

通过YAML模板成功创建存储卷（PV）和存储声明（PVC）后，在**存储卷**列表和**存储声明**列表分别看到创建好的CPFS存储卷（PV）和存储声明（PVC）。

1.3. 训练

1.3.1. TensorFlow单机训练

文本展示如何使用Arena提交TensorFlow的单机训练作业，并通过TensorBoard可视化查看训练作业。

前提条件

- 创建包含GPU的Kubernetes集群。
- 集群节点可以访问公网。
- 已经安装最新版的Arena。
- 已给集群配置了Arena使用的PVC，并且PVC已填充本文使用的数据集，详情请参见配置NAS共享存储（或者配置CPFS共享存储）。

背景信息

本文示例从Git URL下载源代码，数据集放在共享存储系统（基于NAS的PV和PVC）中。示例假设您已经获得了一个名称为**training-data**的PVC实例（一个共享存储），里面存在一个目录*tf_data*，存放了示例所使用的数据集。

操作步骤

1. 执行以下命令检查可用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0       0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0       0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0       0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2       0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2       0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2       0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
0/6 (0%)
```

由上看出，有3个包含GPU的节点可用于运行训练作业。

2. 执行`arena submit tfjob/tf [--flag]`命令提交TensorFlow作业。通过以下代码示例提交一个单机单卡的TensorFlow作业。

```
 arena submit tf \
   --name=tf-git \
   --working-dir=/root \
   --gpus=1 \
   --image=tensorflow/tensorflow:1.5.0-devel-gpu \
   --sync-mode=git \
   --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
   --data=training-data:/mnist_data \
   --tensorboard \
   --logdir=/mnist_data/tf_data/logs \
   "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --log_dir /mnist_data/tf_data/logs  --data_dir /mnist_data/tf_data/"
```

```
configmap/tf-git-tfjob created
configmap/tf-git-tfjob labeled
service/tf-git-tensorboard created
deployment.apps/tf-git-tensorboard created
tfjob.kubeflow.org/tf-git created
INFO[0000] The Job tf-git has been submitted successfully
INFO[0000] You can run `arena get tf-git --type tfjob` to check the job status
```

参数解释如下表。

参数	是否必选	解释	默认值
--name	必选	指定提交的作业名字，全局唯一，不能重复。	无
--working-dir	可选	指定当前执行命令所在的目录。	/root
--gpus	可选	指定作业Worker节点需要使用的GPU卡数。	0
--image	必选	指定训练环境的镜像地址。	无
--sync-mode	可选	同步代码的模式，您可以指定git、rsync。本文使用git模式。	无
--sync-source	可选	同步代码的仓库地址，需要和--sync-mode一起使用，本文示例使用git模式，该参数可以为任何github项目地址。阿里云code项目地址等支持git的代码托管地址。项目代码将会被下载到--working-dir下的*code/*目录中。本文示例即为：*/root/code/tensorflow-sample-code*。	无

参数	是否必选	解释	默认值
--data	可选	挂载共享存储卷PVC到运行环境中。它由两部分组成，通过分号 : 分割。冒号左侧是您已经准备好的PVC名称。您可以通过 arena data list 查看当前集群可用的PVC列表；分号右侧是您想将PVC的挂载到运行环境中的路径，也是您训练代码要读取数据的本地路径。这样通过挂载的方式，您的代码就可以访问PVC的数据。 **说明** 执行 arena data list 查看本文示例当前集群可用的PVC列表。 NAME ACCESSMODE DESCRIPTION OWNER AGE training-data ReadWriteMany 35m 如果没有可用的PVC，您可创建PVC。详情请参见配置NAS共享存储。	无
--tensorboard	可选	为训练任务开启一个TensoBoard服务，用作数据可视化，您可以结合--logdir指定TensorBoard要读取的event路径。不指定该参数，则不开启TensorBoard服务。	无
--logdir	可选	需要结合--tensorboard一起使用，该参数表示TensorBoard需要读取event数据的路径。	/training_logs

注意 如果您使用非公开Git代码库，则可以使用以下命令。

```
arena submit tf \
    ...
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
    --env=GIT_SYNC_USERNAME=yourname \
    --env=GIT_SYNC_PASSWORD=yourpwd \
    "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py
```

arena命令使用git-sync同步源代码。您可以设置在git-sync项目中定义的环境变量。

3. 执行以下命令查看当前通过Arena提交的所有作业。

```
arena list
```

```
NAME   STATUS   TRAINER  AGE  NODE
tf-git  RUNNING  TFJOB   20s  192.1xx.x.xx
```

4. 执行以下命令检查作业所使用的GPU资源。

```
arena top job
```

```
NAME   GPU(Requests)  GPU(Allocated)  STATUS   TRAINER  AGE  NODE
tf-git  1           1           RUNNING  tfjob    1m   192.1xx.x.xx


Total Allocated GPUs of Training Job:
1

Total Requested GPUs of Training Job:
1
```

5. 执行以下命令检查集群所使用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS     ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        1
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
1/6 (16%)
```

6. 执行以下命令获取作业详情。

```
arena get tf-git
```

```
STATUS: SUCCEEDED
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 1m

NAME   STATUS    TRAINER  AGE  INSTANCE      NODE
tf-git  SUCCEEDED  TFJOB   18m  tf-git-chief-0  192.16x.x.xx

Your tensorboard will be available on:
http://192.16x.x.xx:31619
```

> **说明** 本文示例因为开启TensorBoard，在上述作业详情中最后两行，可以看到TensorBoard的web访问地址；如果没有开启TensorBoard，最后两行信息不存在。

7. 通过浏览器查看TensorBoard。从上述步骤的作业详情中，可以看到TensorBoard的web服务地址。由于集群为远端部署，因此需要利用sshuttle代理才能在您的电脑中通过浏览器查看训练可视化的信息。使用sshuttle代理示例代码如下。

```
# you can install sshuttle==0.74 in your mac with python2.7
pip install sshuttle==0.74
# 0/0 -> 0.0.0.0/0
sshuttle -r root@39.104.xx.xxx  0/0
```

> **说明** 39.104.xx.xxx为ACK集群对外暴露的公网IP地址。此外，您还需检查您的安全组是否开启了22端口，一般默认开启。

将步骤6中获取的TensorBoard的web服务地址http://192.16x.x.xx:31619，拷贝至浏览器地址栏，显示效果如下图。

tensorboard

8. 执行以下命令获取作业日志信息。

```
arena logs tf-git
```

```
WARNING:tensorflow:From code/tensorflow-sample-code/tfjob/docker/mnist/main.py:120: softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:
...
Accuracy at step 9970: 0.9834
Accuracy at step 9980: 0.9828
Accuracy at step 9990: 0.9816
Adding run metadata for 9999
Total Train-accuracy=0.9816
```

您还可以通过命令 `arena logs $job_name -f` 实时查看作业的日志输出，通过命令 `arena logs $job_name -t N` 查看尾部N行的日志，以及通过 `arena logs --help` 查询更多参数使用情况。
查看尾部N行的日志示例代码如下。

```
arena logs tf-git -t 5
```

```
Accuracy at step 9970: 0.9834
Accuracy at step 9980: 0.9828
Accuracy at step 9990: 0.9816
Adding run metadata for 9999
Total Train-accuracy=0.9816
```

1.3.2. TensorFlow分布式训练

文本展示如何使用Arena提交TensorFlow基于PS-Worker模式的分布式训练作业，并通过TensorBoard可视化查看训练作业。

前提条件

- 创建包含GPU的Kubernetes集群。
- 集群节点可以访问公网。
- 已经安装最新版的Arena。
- 已给集群配置了Arena使用的PVC，并且PVC已填充本文使用的数据集，详情请参见配置NAS共享存储（或者配置CPFS共享存储）。

背景信息

本文示例从Git URL下载源代码，数据集放在共享存储系统（基于NAS的PV和PVC）中。示例假设您已经获得了一个名称为**training-data**的PVC实例（一个共享存储），里面存在一个目录*tf_data*，存放了示例所使用的数据集。

操作步骤

1. 执行以下命令检查可用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
0/6 (0%)
```

由上看出，有3个包含GPU的节点可用于运行训练作业。

2. 执行 `arena submit tfjob/tf [--flag] command` 形式命令提交TensorFlow作业。通过以下代码示例提交PS-Worker模式下的TensorFlow分布式作业，它包含1个PS（Parameter Server）节点，2个Worker节点。

```
  arena submit tf --name=tf-dist       \
        --working-dir=/root \
        --gpus=1          \
        --workers=2          \
        --worker-image=tensorflow/tensorflow:1.5.0-devel-gpu  \
        --sync-mode=git \
        --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
        --ps=1          \
        --ps-image=tensorflow/tensorflow:1.5.0-devel   \
        --data=training-data:/mnist_data \
        --tensorboard \
        --logdir=/mnist_data/tf_data/logs \
        "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --log_dir /mnist_data/tf_d
ata/logs  --data_dir /mnist_data/tf_data/"
```

```
configmap/tf-dist-tfjob created
configmap/tf-dist-tfjob labeled
service/tf-dist-tensorboard created
deployment.apps/tf-dist-tensorboard created
tfjob.kubeflow.org/tf-dist created
INFO[0000] The Job tf-dist has been submitted successfully
INFO[0000] You can run `arena get tf-dist --type tfjob` to check the job status
```

参数解释如下表。

参数	是否必选	解释	默认值
--name	必选	指定提交的作业名字，全局唯一，不能重复。	无
--working-dir	可选	指定当前执行命令所在的目录。	/root
--gpus	可选	指定作业Worker节点需要使用的GPU有卡数。	0
--workers	可选	指定作业Worker节点的数量。	1
--image	如果不单独指定--worker-image、--ps-image，则必选。	指定训练环境的镜像地址。如果不指定--worker-image或者--ps-image，则Worker节点和PS节点都使用该镜像地址。	无
--worker-image	如果不指定--image，则必选。	指定作业Worker节点需要使用的镜像地址。如果--image同时出现，则覆盖--image。	无
--sync-mode	可选	同步代码的模式，您可以指定git、rsync。本文使用git模式。	无
--sync-source	可选	同步代码的仓库地址，需要和--sync-mode一起使用。本文示例使用git模式，该参数可以为任何github项目地址。阿里云code项目地址等支持git的代码托管地址。项目代码将会被下载到--working-dir下的*code/*目录中。本文示例即为：*/root/code/tensorflow-sample-code*。	无
--ps	分布式作业必选	指定参数服务器（Parameter Server）节点数。	0
--ps-image	如果不指定--image，则必选。	指定PS节点的镜像地址。如果--image同时出现，则覆盖--image。	无

参数	是否必选	解释	默认值
--data	可选	挂载共享存储卷PVC到运行环境中。它由两部分组成，通过分号（ : ）分割。分号左侧是您已经准备好的PVC名称。您可以通过命令 arena data list 查看当前集群可用的PVC列表；分号右侧是您想将PVC的挂载到运行环境中的路径，也是您训练代码要读取数据的本地路径。这样通过挂载的方式，您的代码就可以访问PVC的数据。 **说明** 执行 arena data list 查看本文示例当前集群可用的PVC列表。 NAME ACCESSMODE DESCRIPTION OWNER AGE training-data ReadWriteMany 35m 如果没有可用的PVC，您可创建PVC。详情请参见配置NAS共享存储（或者配置CPFS共享存储）。	无
--tensorboard	可选	为训练任务开启一个TensoBoard服务，用作数据可视化，您可以结合--logdir指定TensorBoard要读取的event路径。不指定该参数，则不开启TensorBoard服务。	无
--logdir	可选	需要结合--tensorboard一起使用，该参数表示TensorBoard需要读取event数据的路径。	/training_logs

注意 如果您使用非公开Git代码库，则可以使用以下命令。

```
arena submit tf \
    ...
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
    --env=GIT_SYNC_USERNAME=yourname \
    --env=GIT_SYNC_PASSWORD=yourpwd \
    "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py
```

arena命令使用git-sync同步源代码。您可以设置在git-sync项目中定义的环境变量。

3. 执行以下命令查看当前通过Arena提交的所有作业。

```
arena list
```

```
NAME    STATUS    TRAINER  AGE  NODE
tf-dist  RUNNING   TFJOB   58s  192.1xx.x.xx
tf-git   SUCCEEDED TFJOB   2h   N/A
```

4. 执行以下命令检查作业所使用的GPU资源。

```
arena top job
```

```
NAME    GPU(Requests)  GPU(Allocated)  STATUS    TRAINER  AGE  NODE
tf-dist  2          2           RUNNING   tfjob   1m   192.1xx.x.x
tf-git   1          0           SUCCEEDED tfjob   2h   N/A


Total Allocated GPUs of Training Job:
2

Total Requested GPUs of Training Job:
3
```

5. 执行以下命令检查集群所使用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx <none>  ready   2        1
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx <none>  ready   2        1
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx <none>  ready   2        0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
2/6 (33%)
```

6. 执行以下命令获取作业详情。

```
arena get tf-dist
```

```
STATUS: RUNNING
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 1m

NAME    STATUS   TRAINER  AGE  INSTANCE        NODE
tf-dist  RUNNING  TFJOB   1m   tf-dist-ps-0     192.1xx.x.xx
tf-dist  RUNNING  TFJOB   1m   tf-dist-worker-0  192.1xx.x.xx
tf-dist  RUNNING  TFJOB   1m   tf-dist-worker-1  192.1xx.x.xx

Your tensorboard will be available on:
http://192.1xx.x.xx:31870
```

> **说明** 本文示例因为开启TensorBoard，在上述作业详情中最后两行，可以看到TensorBoard的web访问地址；如果没有开启TensorBoard，最后两行信息不存在。

7. 通过浏览器查看TensorBoard。从上述步骤的作业详情中，可以看到TensorBoard的web服务地址。由于集群为远端部署，因此需要使用sshuttle代理才能在您的电脑中通过浏览器查看训练可视化的信息。使用sshuttle代理示例代码如下。

```
# you can install sshuttle==0.74 in your mac with python2.7
pip install sshuttle==0.74
# 0/0 -> 0.0.0.0/0
sshuttle -r root@39.104.xx.xxx  0/0
```

> **说明** 39.104.xx.xxx为ACK集群对外暴露的公网IP地址。此外，您还需检查您的安全组是否开启了22端口，一般默认开启。

将步骤6中获取的TensorBoard的web服务地址http://192.1xx.x.xx:31870，拷贝至浏览器地址栏，显示效果如下图。

8. 执行以下命令获取作业日志信息。

```
arena logs tf-dist
```

```
WARNING:tensorflow:From code/tensorflow-sample-code/tfjob/docker/mnist/main.py:120: softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:
...
Accuracy at step 960: 0.9691
Accuracy at step 970: 0.9677
Accuracy at step 980: 0.9687
Accuracy at step 990: 0.968
Adding run metadata for 999
Total Train-accuracy=0.968
```

使用上述命令获取作业日志信息时，默认输出worker-0的节点日志。如果需要查看分布式训练任务中的某个节点日志，可以先查看作业详情获取作业的节点列表，然后使用命令 `arena logs $job_name -i $instance_name` 查看具体实例的日志。
示例代码如下。

```
arena get tf-dist
# 输出:
STATUS: SUCCEEDED
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 1m

NAME    STATUS    TRAINER  AGE  INSTANCE        NODE
tf-dist  SUCCEEDED  TFJOB   5m   tf-dist-ps-0    192.16x.x.xx
tf-dist  SUCCEEDED  TFJOB   5m   tf-dist-worker-0  192.16x.x.xx
tf-dist  SUCCEEDED  TFJOB   5m   tf-dist-worker-1  192.16x.x.xx

Your tensorboard will be available on:
http://192.16x.x.xx:31870
```

```
arena logs tf-dist -i tf-dist-worker-1
# 输出:
WARNING:tensorflow:From code/tensorflow-sample-code/tfjob/docker/mnist/main.py:120: softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:
...
Accuracy at step 970: 0.9676
Accuracy at step 980: 0.968
Accuracy at step 990: 0.967
Adding run metadata for 999
Total Train-accuracy=0.967
```

您还可以通过命令 `arena logs $job_name -f` 实时查看作业的日志输出，通过命令 `arena logs $job_name -t N` 查看尾部N行的日志，以及通过 `arena logs --help` 查询更多参数使用情况。
查看尾部N行的日志示例代码如下。

```
arena logs tf-dist -t 5
```

```
Accuracy at step 9970: 0.9834
Accuracy at step 9980: 0.9828
Accuracy at step 9990: 0.9816
Adding run metadata for 9999
Total Train-accuracy=0.9816
```

1.3.3. PyTorch单机训练

文本展示如何使用Arena提交PyTorch的单机训练作业，并通过TensorBoard可视化查看训练作业。

前提条件

- 创建包含GPU的Kubernetes集群。
- 集群节点可以访问公网。
- 已经安装最新版的Arena。
- 已给集群配置了Arena使用的PVC，并且PVC已填充本文使用的数据集，详情请参见配置NAS共享存储（或者配置CPFS共享存储）。

背景信息

本文示例从Git URL下载源代码，数据集放在共享存储系统（基于NAS的PV和PVC）中。示例假设您已经获得了一个名称为**training-data**的PVC实例（一个共享存储），里面存在一个目录*pytorch_data*，其存放了示例所使用的数据集。

操作步骤

1. 执行以下命令检查可用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE    STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0         0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0         0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0         0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2         0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2         0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2         0
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
0/6 (0%)
```

由上看出，有3个包含GPU的节点可用于运行训练作业。

2. 执行 `arena submit tfjob/tf [--flag] command` 形式命令提交PyTorch作业。通过以下示例代码提交一个单机单卡的PyTorch作业。

```
 arena submit pytorch \
   --name=pytorch-git \
   --gpus=1 \
   --working-dir=/root \
   --image=registry.cn-huhehaote.aliyuncs.com/lumo/pytorch-with-tensorboard:1.5.1-cuda10.1-cudnn7-runtime \
   --sync-mode=git \
   --sync-source=https://code.aliyun.com/370272561/mnist-pytorch.git \
   --data=training-data:/mnist_data \
   --tensorboard \
   --logdir=/mnist_data/pytorch_data/logs \
   "python /root/code/mnist-pytorch/mnist.py --epochs 10 --backend nccl --dir /mnist_data/pytorch_data/logs --data /mnist_data/pytorch_data/"
```

```
configmap/pytorch-git-pytorchjob created
configmap/pytorch-git-pytorchjob labeled
service/pytorch-git-tensorboard created
deployment.apps/pytorch-git-tensorboard created
pytorchjob.kubeflow.org/pytorch-git created
INFO[0000] The Job pytorch-git has been submitted successfully
INFO[0000] You can run `arena get pytorch-git --type pytorchjob` to check the job status
```

参数解释如下表。

参数	是否必选	解释	默认值
--name	必选	指定提交的作业名字，全局唯一，不能重复。	无
--working-dir	可选	指定当前执行命令所在的目录。	/root
--gpus	可选	指定作业Worker节点需要使用的GPU有卡数。	0
--image	必选	指定训练环境的镜像地址。	无
--sync-mode	可选	同步代码的模式，您可以指定git、rsync。本文使用git模式。	无
--sync-source	可选	同步代码的仓库地址，需要和--sync-mode一起使用，本文示例使用git模式，该参数可以为任何github项目地址。阿里云code项目地址等支持git的代码托管地址。项目代码将会被下载到--working-dir下的*code/*目录中。本文示例即为：*/root/code/mnist-pytorch*。	无

参数	是否必选	解释	默认值
--data	可选	挂载共享存储卷PVC到运行环境中。它由两部分组成，通过分号 : 分割。分号左侧是您已经准备好的PVC名称。您可以通过 arena data list 查看当前集群可用的PVC列表；分号右侧是您想将PVC的挂载到运行环境中的路径，也是您训练代码要读取数据的本地路径。这样通过挂载的方式，您的代码就可以访问PVC的数据。 **说明** 执行 arena data list 查看本文示例当前集群可用的PVC列表。 NAME ACCESSMODE DESCRIPTION OWNER AGE training-data ReadWriteMany 35m 如果没有可用的PVC，您可创建PVC。详情请参见配置NAS共享存储（或者配置CPFS共享存储）。	无
--tensorboard	可选	为训练任务开启一个TensoBoard服务，用作数据可视化，您可以结合--logdir指定TensorBoard要读取的event路径。不指定该参数，则不开启TensorBoard服务。	无
--logdir	可选	需要结合--tensorboard一起使用，该参数表示TensorBoard需要读取event数据的路径。	/training_logs

> 注意 如果您使用非公开Git代码库，则可以使用以下命令。
>
> ```
> arena --loglevel info submit pytorch \
>     ...
>     --sync-mode=git \
>     --sync-source=https://code.aliyun.com/370272561/mnist-pytorch.git \
>     --env=GIT_SYNC_USERNAME=yourname \
>     --env=GIT_SYNC_PASSWORD=yourpwd \
>     "python /root/code/mnist-pytorch/mnist.py --backend gloo"
> ```
>
> arena命令使用git-sync同步源代码。您可以设置在git-sync项目中定义的环境变量。

3. 执行以下命令查看当前通过Arena提交的所有作业。

```
arena list
```

```
NAME       STATUS    TRAINER    AGE  NODE
pytorch-git  RUNNING   PYTORCHJOB  19s  192.1xx.x.xx
tf-dist     SUCCEEDED  TFJOB      13h  N/A
tf-git      SUCCEEDED  TFJOB      16h  N/A
```

4. 执行以下命令检查作业所使用的GPU资源。

```
arena top job
```

```
NAME       GPU(Requests)  GPU(Allocated)  STATUS    TRAINER    AGE  NODE
tf-dist     2           0           SUCCEEDED  tfjob      13h  N/A
tf-git      1           0           SUCCEEDED  tfjob      16h  N/A
pytorch-git  1           1           RUNNING    pytorchjob  25s  192.1xx.x.xx


Total Allocated GPUs of Training Job:
1

Total Requested GPUs of Training Job:
4
```

5. 执行以下命令检查集群所使用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        1
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
1/6 (16%)
```

6. 执行以下命令获取作业详情。

```
arena get pytorch-git
```

```
STATUS: SUCCEEDED
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 2m

NAME        STATUS     TRAINER     AGE  INSTANCE           NODE
pytorch-git  SUCCEEDED  PYTORCHJOB  3m   pytorch-git-master-0  192.16x.x.xx

Your tensorboard will be available on:
http://192.16x.x.xx:30171
```

> **说明** 本文示例因为开启TensorBoard，在上述作业详情中最后两行，可以看到TensorBoard的web访问地址；如果没有开启TensorBoard，最后两行信息不存在。

7. 通过浏览器查看TensorBoard。从上述步骤的作业详情中，可以看到TensorBoard的web服务地址。由于集群为远端部署，因此需要利用sshuttle代理才能在您的电脑中通过浏览器查看训练可视化的信息。使用sshuttle代理示例代码如下。

```
# you can install sshuttle==0.74 in your mac with python2.7
pip install sshuttle==0.74
# 0/0 -> 0.0.0.0/0
sshuttle -r root@39.104.xx.xxx  0/0
```

> **说明** 39.104.xx.xxx为ACK集群对外暴露的公网IP地址。此外，您还需检查您的安全组是否开启了22端口，一般默认开启。

将步骤6中获取的TensorBoard的web服务地址http://192.16x.x.xx:30171，拷贝至浏览器地址栏，显示效果如下图。

pytorch单机

> **说明** 本文PyTorch示例代码，默认每10次epoch写入event信息。如果您修改了--epochs参数，请修改为10的整数倍，否则无法在TensorBoard上看到数据。

8. 执行以下命令获取作业日志信息。

```
arena logs pytorch-git
```

```
WORLD_SIZE: 1, CURRENT_RANK: 0
args: Namespace(backend='nccl', batch_size=64, data='/mnist_data/data', dir='/mnist_data/logs', epochs=1, log_interval=10, lr=0.01, momentum=0.5, no_cuda=False, save_model=False, seed=1, test_batch_size=1000)
Using CUDA
...
Train Epoch: 10 [58240/60000 (97%)] loss=0.0128
Train Epoch: 10 [58880/60000 (98%)] loss=0.0098
Train Epoch: 10 [59520/60000 (99%)] loss=0.0051

accuracy=0.9904
```

您还可以通过命令 `arena logs $job_name -f` 实时查看作业的日志输出，通过命令 `arena logs $job_name -t N` 查看尾部N行的日志，以及通过 `arena logs --help` 查询更多参数使用情况。
查看尾部N行的日志示例代码如下。

```
arena logs pytorch-git -t 5
```

```
Train Epoch: 10 [58880/60000 (98%)] loss=0.0098
Train Epoch: 10 [59520/60000 (99%)] loss=0.0051

accuracy=0.9904
```

1.3.4. PyTorch分布式训练

文本展示如何使用Arena提交PyTorch的分布式训练作业，并通过TensorBoard可视化查看训练作业。

前提条件

- 创建包含GPU的Kubernetes集群。
- 集群节点可以访问公网。
- 已经安装最新版的Arena。
- 已给集群配置了Arena使用的PVC，并且PVC已填充本文使用的数据集，详情请参见配置NAS共享存储（或者配置CPFS共享存储）。

背景信息

本文示例从Git URL下载源代码，数据集放在共享存储系统（基于NAS的PV和PVC）中。示例假设您已经获得

了一个名称为training-data的PVC实例（一个共享存储），里面存在一个目录*pytorch_data*，存放了示例所使用的数据集。

操作步骤

1. 执行以下命令检查可用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE    STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.16x.x.xx  192.1xx.x.xx  <none>  ready   2        0
-----------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
0/6 (0%)
```

由上看出，有3个包含GPU的节点可用于运行训练作业。

2. 执行 `arena submit tfjob/tf [--flag] command` 形式命令提交PyTorch多机作业。通过以下代码示例提交一个3机1卡的PyTorch训练作业。

```
arena submit pytorch \
    --name=pytorch-dist \
    --gpus=1 \
    --workers=3 \
    --working-dir=/root \
    --image=registry.cn-huhehaote.aliyuncs.com/lumo/pytorch-with-tensorboard:1.5.1-cuda10.1-cudnn7-runtime \
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/370272561/mnist-pytorch.git \
    --data=training-data:/mnist_data \
    --tensorboard \
    --logdir=/mnist_data/pytorch_data/logs \
    "python /root/code/mnist-pytorch/mnist.py --epochs 10 --backend nccl --dir /mnist_data/pytorch_data/logs --data /mnist_data/pytorch_data/"
```

```
configmap/pytorch-dist-pytorchjob created
configmap/pytorch-dist-pytorchjob labeled
service/pytorch-dist-tensorboard created
deployment.apps/pytorch-dist-tensorboard created
pytorchjob.kubeflow.org/pytorch-dist created
INFO[0000] The Job pytorch-dist has been submitted successfully
INFO[0000] You can run `arena get pytorch-dist --type pytorchjob` to check the job status
```

说明 PyTorch的分布式作业较单机作业多了一个--workers参数，它表明参数分布式作业训练的所有节点数。一个分布式作业有多个节点，节点名称的组成结构：【job_name】-【role_name】-【index】。

- 【job_name】为您指定作业的名称。
- 【role_name】表名这个节点的角色，这里包含一个特殊的节点Master节点，对应PyTorch分布式作业中rank-0节点，该节点的实例名带有master关键字，例如pytorch-dist-master-0，并且一个作业仅有一个实例名带有master字样的节点。
- 其他非rank-0的节点角色即为Worker节点，他们的rank号即为各自对应节点名称中【index】+1；每个节点都注入了RANK环境变量，因此，您也可以通过获取该环境变量拿到当前节点对应的rank号。

参数解释如下表。

参数	是否必选	解释	默认值
--name	必选	指定提交的作业名字，全局唯一，不能重复。	无
--working-dir	可选	指定当前执行命令所在的目录。	/root
--gpus	可选	指定作业Worker节点需要使用的GPU有卡数。	0
--workers	分布式作业必选	指定作业需要的节点数，它设置的值包含Master节点。例如设置3，表明该作业包含1个Master节点和2个Worker节点。	0
--image	必选	指定训练环境的镜像地址。	无
--worker-image	如果不指定--image，则必选。	指定作业Worker节点需要使用的镜像地址。如果--image同时出现，则覆盖--image。	无
--sync-mode	可选	同步代码的模式，您可以指定git、rsync。本文使用git模式。	无

参数	是否必选	解释	默认值
--sync-source	可选	同步代码的仓库地址，需要和--sync-mode一起使用。本文示例使用git模式，该参数可以为任何github项目地址。阿里云code项目地址等支持git的代码托管地址。项目代码将会被下载到--working-dir下的*code/*目录中。本文示例即为：*/root/code/mnist-pytorch*。	无
--data	可选	挂载共享存储卷PVC到运行环境中。它由两部分组成，通过分号（ : ）分割。分号左侧是您已经准备好的PVC名称。您可以通过命令 arena data list 查看当前集群可用的PVC列表；分号右侧是您想将PVC的挂载到运行环境中的路径，也是您训练代码要读取数据的本地路径。这样通过挂载的方式，您的代码就可以访问PVC的数据。 **说明** 执行 arena data list 查看本文示例当前集群可用的PVC列表。 NAME ACCESSMODE DESCRIPTION OWNER AGE training-data ReadWriteMany 35m 如果没有可用的PVC，您可创建PVC。详情请参见配置NAS共享存储（或者配置CPFS共享存储）。	无
--tensorboard	可选	为训练任务开启一个TensoBoard服务，用作数据可视化，您可以结合--logdir指定TensorBoard要读取的event路径。不指定该参数，则不开启TensorBoard服务。	无
--logdir	可选	需要结合--tensorboard一起使用，该参数表示TensorBoard需要读取event数据的路径。	/training_logs

> 注意 如果您使用非公开Git代码库，则可以使用以下命令。
>
> ```
> arena --loglevel info submit pytorch \
>     ...
>     --sync-mode=git \
>     --sync-source=https://code.aliyun.com/370272561/mnist-pytorch.git \
>     --env=GIT_SYNC_USERNAME=yourname \
>     --env=GIT_SYNC_PASSWORD=yourpwd \
>     "python /root/code/mnist-pytorch/mnist.py --backend gloo"
> ```
>
> arena命令使用git-sync同步源代码。您可以设置在git-sync项目中定义的环境变量。

3. 执行以下命令查看当前通过Arena提交的所有作业。

```
arena list
```

```
NAME         STATUS    TRAINER    AGE  NODE
pytorch-dist  RUNNING   PYTORCHJOB 21s  192.16x.x.xx
pytorch-git   SUCCEEDED PYTORCHJOB 46m  N/A
tf-dist      SUCCEEDED TFJOB     14h  N/A
tf-git       SUCCEEDED TFJOB     17h  N/A
```

4. 执行以下命令检查作业所使用的GPU资源。

```
arena top job
```

```
NAME         GPU(Requests)  GPU(Allocated)  STATUS    TRAINER    AGE  NODE
pytorch-dist  3              3               RUNNING   pytorchjob  29s  192.16x.x.xx
tf-dist      2              0               SUCCEEDED tfjob      14h  N/A
tf-git       1              0               SUCCEEDED tfjob      17h  N/A
pytorch-git   1              0               SUCCEEDED pytorchjob  46m  N/A


Total Allocated GPUs of Training Job:
3

Total Requested GPUs of Training Job:
7
```

5. 执行以下命令检查集群所使用的GPU资源。

```
arena top node
```

```
NAME                IPADDRESS    ROLE   STATUS  GPU(Total)  GPU(Allocated)
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  master  ready   0        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        2
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        0
cn-huhehaote.192.1xx.x.xx  192.1xx.x.xx  <none>  ready   2        1
-----------------------------------------------------------------------------------------
Allocated/Total GPUs In Cluster:
3/6 (50%)
```

6. 执行以下命令获取作业详情。

```
arena get pytorch-dist
```

```
STATUS: RUNNING
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 57s

NAME        STATUS   TRAINER    AGE  INSTANCE           NODE
pytorch-dist  RUNNING  PYTORCHJOB  57s  pytorch-dist-master-0  192.168.0.33
pytorch-dist  RUNNING  PYTORCHJOB  57s  pytorch-dist-worker-0  192.168.0.31
pytorch-dist  RUNNING  PYTORCHJOB  57s  pytorch-dist-worker-1  192.168.0.31

Your tensorboard will be available on:
http://192.16x.x.xx:30131
```

> **说明** 本文示例因为开启TensorBoard，在上述作业详情中最后两行，可以看到TensorBoard的web访问地址；如果没有开启TensorBoard，最后两行信息不存在。

通过以上命令，可以看到该作业包含一个Master节点（ rank-0节点）和2个Worker节点（非rank-0节点），它们共同参与整个训练。

7. 通过浏览器查看TensorBoard。从上述步骤的作业详情中，可以看到TensorBoard的web服务地址。由于集群为远端部署，因此需要使用sshuttle代理才能在您的电脑中通过浏览器查看训练可视化的信息。使用sshuttle代理示例代码如下。

```
# you can install sshuttle==0.74 in your mac with python2.7
pip install sshuttle==0.74
# 0/0 -> 0.0.0.0/0
sshuttle -r root@39.104.xx.xxx  0/0
```

> **说明** 39.104.xx.xxx为ACK集群对外暴露的公网IP地址。此外，您还需检查您的安全组是否开启了22端口，一般默认开启。

将步骤6中获取的TensorBoard的web服务地址http://192.1xx.x.xx:31870，拷贝至浏览器地址栏，显示效果如下图。

PyTorch

> **说明** 本文PyTorch示例代码，默认每10次epoch写入envent信息。如果您修改了--epochs参数，请修改为10的整数倍，否则无法在TensorBoard上看到数据。

8. 执行以下命令获取作业日志信息。

```
arena logs pytorch-dist
```

```
WORLD_SIZE: 3, CURRENT_RANK: 0
args: Namespace(backend='nccl', batch_size=64, data='/mnist_data/pytorch_data/', dir='/mnist_data/
pytorch_data/logs', epochs=10, log_interval=10, lr=0.01, momentum=0.5, no_cuda=False, save_model=F
alse, seed=1, test_batch_size=1000)
Using CUDA
...
Train Epoch: 10 [57600/60000 (96%)] loss=0.0026
Train Epoch: 10 [58240/60000 (97%)] loss=0.0101
Train Epoch: 10 [58880/60000 (98%)] loss=0.0106
Train Epoch: 10 [59520/60000 (99%)] loss=0.0051


accuracy=0.9904
```

使用上述命令获取作业日志信息时，默认输出rank-0的节点（带有master字样的节点）日志。如果需要查看分布式训练任务中的某个节点日志，可以先查看作业详情获取作业的节点列表，然后使用命令 `arena logs $job_name -i $instance_name` 查看具体实例的日志。
示例代码如下。

```
arena get pytorch-dist
# 输出:
STATUS: SUCCEEDED
NAMESPACE: default
PRIORITY: N/A
TRAINING DURATION: 3m

NAME        STATUS    TRAINER    AGE  INSTANCE           NODE
pytorch-dist  SUCCEEDED  PYTORCHJOB  4m   pytorch-dist-master-0  192.16x.x.xx
pytorch-dist  SUCCEEDED  PYTORCHJOB  4m   pytorch-dist-worker-0  192.16x.x.xx
pytorch-dist  SUCCEEDED  PYTORCHJOB  4m   pytorch-dist-worker-1  192.16x.x.xx

Your tensorboard will be available on:
http://192.16x.x.xx:30131
```

```
arena logs pytorch-dist -i pytorch-dist-worker-0
# 输出:
WORLD_SIZE: 3, CURRENT_RANK: 1
args: Namespace(backend='nccl', batch_size=64, data='/mnist_data/pytorch_data/', dir='/mnist_data/
pytorch_data/logs', epochs=10, log_interval=10, lr=0.01, momentum=0.5, no_cuda=False, save_model=F
alse, seed=1, test_batch_size=1000)
Using CUDA
...
Train Epoch: 10 [57600/60000 (96%)] loss=0.0026
Train Epoch: 10 [58240/60000 (97%)] loss=0.0101
Train Epoch: 10 [58880/60000 (98%)] loss=0.0106
Train Epoch: 10 [59520/60000 (99%)] loss=0.0051

accuracy=0.9904
```

您还可以通过命令 `arena logs $job_name -f` 实时查看作业的日志输出，通过命令 `arena logs $job_name -t N` 查看尾部N行的日志，以及通过 `arena logs --help` 查询更多参数使用情况。
查看尾部N行的日志示例代码如下。

```
arena logs pytorch-dist -t 5
```

```
Train Epoch: 10 [58880/60000 (98%)] loss=0.0106
Train Epoch: 10 [59520/60000 (99%)] loss=0.0051

accuracy=0.9904
```

1.4. Arena在多用户场景下的最佳实践

本文通过实现五个目标任务举例说明如何在多用户场景下使用Arena。

前提条件

请确保您已完成以下操作：

- 创建一个ACK集群。详情请参见创建Kubernetes托管版集群。
- 在ACK集群同VPC下，创建一个操作系统为Linux的ECS实例。具体操作步骤请参见使用向导创建实例。本示例中，ECS实例被称为Client机器。Client机器作为Arena的工作站，提交作业至ACK集群。
- 安装最新版本Arena。具体操作步骤请参见通过组件安装最新版的Arena。

背景信息

当多个开发人员在一个公司或大团体下使用Arena进行工作时，为了有效管理，您可能需要对这些人员进行小组划分，且每个小组需要彼此隔离。这样在同一个集群内，小组就是您分配、隔离资源和权限的基本单元。

通常您需要将整个集群的资源（GPU、CPU、MEM）根据具体需求划分给每个组，并且给组内成员分配不同权限，以及提供各自独立的Arena使用环境。其中权限包括：用户对于作业的可见、可操作权限，用户的作业对特定数据的读写权限。

配置多用户使用Arena的工作环境图

本文示例中的ACK集群和Client机器的节点信息如下表所示。

主机名（假名）	角色	IP地址	GPU卡数	CPU核数	MEM
client01	Client	10.0.0.97（私有）39.98.xxx.xxx（公）	0	2	8 GiB
master01	Master	10.0.0.91（私有）	0	4	8 GiB
master02	Master	10.0.0.92（私有）	0	4	8 GiB
master03	Master	10.0.0.93（私有）	0	4	8 GiB
worker01	Worker	10.0.0.94（私有）	1	4	30 GiB
worker02	Worker	10.0.0.95（私有）	1	4	30 GiB
worker03	Worker	10.0.0.96（私有）	1	4	30 GiB

说明 如无特殊说明的话，本文的操作均为集群管理员（admin）操作，操作节点为Client机器。

目标任务

本文最佳实践的示例中将达成以下五个目标任务：

- 目标一：您需要为当前ACK集群创建两个分组dev1和dev2，并且为这两个分组各添加一个用户bob和tom。
- 目标二：每个用户只能通过自己的账号和密码登录到Client机器，使用自己环境下的Arena。
- 目标三：bob和tom只对自己提交的作业可见，以及进行管理。
- 目标四：按组划分Worker节点GPU、CPU、 MEM资源 （注：仅Worker节点的计算资源才能被Arena作业使用）。
- 目标五：创建组内共享的数据卷，和组与组之间全局共享的数据卷。

数据配置表

组名	用户	GPU	CPU	MEM	共享数据卷
dev1	bob	1	不限制	不限制	*dev1-public*和*department1-public-dev1*
dev2	tom	2	8	60 GiB	*dev2-public*和*department1-public-dev2*

> **说明** *department1-public-dev1*和*department1-public-dev2*数据卷指向的是同一份数据；组dev1和dev2为所有用户共享；*dev1-public*和*dev2-public*分别对应分组dev1、dev2用户独享的数据。

步骤一：创建和管理AI平台的用户和组

为了安全起见，不建议您直接登录ACK集群的Master节点安装使用Arena以及对集群进行操作，因此建议您在与ACK集群同一个VPC下创建ECS实例（Client机器）。通过配置*KubeConfig*文件，使用ECS实例节点对ACK集群进行访问。

1. 创建Client机器上的用户和组。
    i. 通过kubectl连接ACK集群。使用**kubectl**命令连接ACK集群时，您需要安装kubectl客户端工具和配置供集群管理员admin操作ACK集群的KubeConfig文件。有关具体的操作步骤，请参见通过kubectl连接Kubernetes集群。

    > **说明** 要求kubectl的版本大于或等于1.10。

ii. 执行以下代码在Client机器上创建对应的Linux UID和GID。集群管理员需要为bob、tom以及分组dev1、dev2，在Client机器上创建对应的Linux UID和GID。通过Linux自身的账号系统机制，实现目标二，即每个用户只能通过自己的账号和密码登录到Client机器，使用自己环境下的Arena。

```
# Create Linux groups, users.
groupadd -g 10001 dev1
groupadd -g 10002 dev2
adduser -u 20001 -s /bin/bash -G dev1 -m bob
adduser -u 20002 -s /bin/bash -G dev2 -m tom

#设置bob登录操作AI平台的Linux节点的密码。
passwd bob
#设置tom登录操作AI平台的Linux节点的密码。
passwd tom
```

2. 创建ACK集群中的用户（服务账号）和组（命名空间）。
提交给AI平台的作业都将在ACK集群中运行。在ACK中，作业的拥有者以服务账号（ServiceAccount）区分，作业运行的环境以名字空间（Namespace）区分。运维管理员admin将创建ServiceAccount和Namespace，并保证与在Client机器上创建的用户和组对应。可以将Namespace对应组，ServiceAccount对应用户。
admin以root身份登录Client机器，并确保拥有操作集群的权限（通过kubectl连接ACK集群步骤中完成）。执行以下操作：

```
# 创建分组dev1对应的Namespace。
kubectl create namespace dev1

# 创建分组dev2对应的Namespace。
kubectl create namespace dev2

# 创建用户bob对应的serviceaccount。
kubectl create serviceaccount bob -n dev1

# 创建用户tom对应的serviceaccount。
kubectl create serviceaccount tom -n dev2

# 可以查看创建的组（Namespace）和用户（ServiceAccount）。
kubectl get serviceaccount -n dev1
kubectl get serviceaccount -n dev2
```

步骤二：为用户配置Arena的使用环境

1. 安装Arena。集群管理员在Client机器上安装Arena。首先集群管理员以root身份登录到Client机器，下载社区最新Arena发行的release安装包，然后解压、执行安装包中*install.sh*脚本。有关具体的操作步骤，请参见通过组件安装最新版的Arena。

> **说明** 在同一台Linux操作机上，您只需安装一份Arena工具。管理员通过为每位用户配置各自的配置文件，从而实现隔离，便于用户以不同的权限使用Arena工具。

2. 为用户创建Arena配置文件。为了使不同用户以各自的身份和权限正确操作AI平台集群（ACK集群），首先您需要创建不同用户用于Arena连接ACK集群的环境配置文件（即创建在各自ServiceAccount下使用的KubeConfig文件）。
   集群管理员以root身份登录到Client机器上，使用附录提供的脚本*createKubeConfig.sh*为每位用户生成各自的KubeConfig配置文件。执行以下代码：

```
# 赋予生成KubeConfig脚本工具执行权限。
chmod +x createKubeConfig.sh

# 为组dev1下的bob用户创建专门的KubeConfig文件。
./createKubeConfig.sh bob -n dev1

# 为组dev2下的tom用户创建专门的KubeConfig文件。
./createKubeConfig.sh tom -n dev2

# 把生成各自用户的Kubernetes环境配置文件放到各自用户的home目录下。
mkdir -p /home/bob/.kube/ && cp bob.config /home/bob/.kube/config
mkdir -p /home/tom/.kube/ && cp tom.config /home/tom/.kube/config
```

   完成上述操作后，用户使用自己的账号登录Client机器后，就可以使用Arena自动连接正确的集群环境。至此，本文示例已完成目标一和目标二。

步骤三：为用户配置Arena的权限

1. 在ACK集群中为每个组（Namespace），按需创建组内的角色（Role）。
   集群管理员可以为每个组（Namespace）在ACK集群的具体操作权限配置不同的角色（Role）。一个角色（Role）代表组（Namespace）内一系列具体权限规则的集合。有关ACK集群中角色（Role）的定义方法，请参见Using RBAC Authorization。

   i. 创建权限定义文件。
      因为需要保证组dev1下用户bob和组dev2下用户tom的作业彼此不可见，并且独立管理各自的作业，所以需要给组dev1和dev2创建最小权限，并绑定到对应的用户上（即用户bob和tom对应的ServiceAccount）。
      有关组dev1的权限定义，请参见附录文件*dev1_roles.yaml*；有关组dev2的权限定义，请参见附录文件*dev2_roles.yaml*。

ii. 创建好权限定义文件*dev1_roles.yaml*和*dev2_roles.yaml*后，集群管理员通过执行以下命令，使之在ACK集群中生效。

```
kubectl apply -f dev1_roles.yaml


kubectl apply -f dev2_roles.yaml
```

集群管理员可以通过以下命令查看集群中不同Namespace下的角色。

```
kubectl get role -n dev1
kubectl get role -n dev2
```

如出现下图所示内容，即完成了角色创建操作。

2. 为用户配置在集群内的权限（赋权）。
组的角色创建完成后，我们需要给用户绑定这个角色，使之作用在组里的成员上，即赋权给用户。集群管理员通过将组（Namespace）内的某些角色（Role）与用户（ServiceAccount）进行绑定，来为每位用户在组内赋权。
通过角色绑定，集群管理员可以为用户赋予多个组内的角色权限。这既包括用户所属的组内权限，也包括其他组内的权限。这样集群管理员就可以动态地管理任何用户在任何组内的权限，只需要更新用户与角色的对应关系即可。
ACK中通过定义RoleBinding和ClusterRoleBinding对象来声明用户（ServiceAccount）在不同组（Namespaces）或者整个集群所有组中可以扮演的角色而获取权限。ACK中角色绑定（RoleBinding和ClusterRoleBinding）的定义方法可以参见官方的文档Using RBAC Authorization。
本文以目标三为例，仅仅需要将用户bob和tom分别绑定各自组的权限即可（步骤三中创建的角色）。
操作步骤如下：

i. 集群管理员以root身份登录Client机器后，通过kubectl执行附录中的文件*bob_rolebindings.yaml*和*tom_rolebindings.yaml*即可完成用户和组角色的绑定，即用户赋权。

```
kubectl apply -f bob_rolebindings.yaml
kubectl apply -f tom_rolebindings.yaml
```

ii. 集群管理员通过以下命令查看集群中不同Namespace下对不同用户的角色赋权情况。

```
kubectl get rolebinding -n dev1
kubectl get rolebinding -n dev2
```

如出现下图所示内容，即完成了权限绑定操作。

至此，本文示例已完成了上述的前三个目标任务。

步骤四：配置用户组资源配额

AI平台通过ACK统一管理所有集群资源。为了保证不同组和用户安全、有效地使用集群资源，需要为小组分配资源配额。AI平台会在用户提交作业时，自动检查用户有权限运行作业的组配额使用情况。如果作业需求量超过配额上限，则提交会被拒绝。

在ACK中资源配额（ResourceQuota）的作用域是命名空间（Namespace），即对应本文中的用户组。ResourceQuota支持的资源类型很多，既包括CPU、Memory，也包括Nvidia GPU这类扩展资源。ResourceQuota也能够限制一个Namespace中容器及其它Kubernetes对象的使用量。有关更详细的说明，请参见Resource Quotas。

根据目标四中的描述，本文示例按照组划分当前集群的GPU、CPU、MEM等资源。示例假设划分的配置如数据配置表所示。本文示例将分配组dev1一张GPU卡，不划分CPU和MEM，即可以使用整个集群的CPU和MEM。分配组dev2两张GPU卡，八核CPU和60 GiB MEM。具体的操作步骤如下：

1. 集群管理员登录Client机器后，使用附录中*dev1_quota.yaml*和*dev2_quota.yaml*文件定义组资源配额。执行以下命令使之在集群中生效。

   ```
   kubectl apply -f dev1_quota.yaml
   kubectl apply -f dev2_quota.yaml
   ```

   创建后，集群管理员可以通过以下命令查看ResourceQuata是否生效，并可以看到定义的配额以及组内已经分配掉的资源。

   ```
   # 查看组dev1下的资源配额。
   kubectl get resourcequotas -n dev1

   # 查看组dev2下的资源配额。
   kubectl get resourcequotas -n dev2

   # 查看组dev1资源配额的使用情况。
   kubectl describe resourcequotas dev1-compute-resources -n dev1

   # 查看组dev2资源配额的使用情况。
   kubectl describe resourcequotas dev2-compute-resources -n dev2
   ```

   至此，本文完成了上述的目标四的按照组划分ACK集群计算资源。

步骤五：配置多级别共享存储NAS

根据用户组织内对数据共享的控制规则，可以为AI平台内的组和用户分别配置带有相应权限控制的数据卷挂载，方便用户间安全地共享数据。

本文示例目标五需要创建两类共享数据卷。一类用于存放全局的共享数据，所有组内成员都可以访问和使用，另一类仅仅用于各个小组内共享。本文假设共享数据卷的需求如数据配置表所示。四个数据卷*dev1-public*、*dev2-public*、*department1-public-dev1*、*department1-public-dev2*中的*department1-public-dev1*和*department1-public-dev2*指向NAS盘的同一个目录，组dev1和dev2共享这一份数据。*dev1-public*、*dev2-public*分别指向组dev1和组dev2所属NAS盘的不同目录，组dev1 、dev2独享这份数据。具体操作如下：

1. 创建NAS实例。在阿里云NAS控制台开通、购买NAS示例，并创建挂载点。有关详细步骤，请参见配置NAS共享存储。
2. 配置ACK集群的存储卷（PV）和存储声明（PVC）。

i. 创建PV。分别创建4个PV。有关创建PV的具体操作步骤，请参见创建PV。*department1-public-dev1*和*department1-public-dev2*是用于dev1和dev2两组分别共享部门department1的数据。*dev1-public*，*dev2-public*是用于dev1和dev2各自组内共享数据。创建PV的参数配置如下图所示。

> **说明** 设置挂载点时，选择上步骤创建NAS实例时创建的挂载点。

ii. 创建PVC。使用上步创建的PV，分别创建PVC。有关创建PVC的具体操作步骤，请参见创建PVC。完成创建PVC后，可以看到在命名空间dev1下给dev1组分配部门和小组两级共享的NAS数据卷：*department1-public-dev1*和*dev1-public*。在命名空间dev2下给dev2组分配部门和小组两级共享的NAS数据卷：*department1-public-dev2*和*dev2-public*。

3. 检查数据卷的配置。集群管理员以root身份登录Client机器，执行以下命令查看为不同组分配的NAS数据情况。

```
# 查看组dev1可以使用的数据卷。
arena data list -n dev1


# 查看组dev2可以使用的数据卷。
arena data list -n dev2
```

至此，本文已经完成了所有目标。接下来本文将会以bob和tom账户进行登录使用Arena。

步骤六：模拟多用户操作

用户bob

1. 登录Client机器，执行以下命令查看当前可以用共享数据卷。

```
# 使用bob账号登录Client机器
ssh bob@39.98.xxx.xx


# 通过arena data list查看当前用户可用共享数据卷。
arena data list
```

2. 执行以下命令提交一个1张GPU卡的训练作业。

```
arena submit tf \
      --name=tf-git-bob-01 \
      --gpus=1 \
      --image=tensorflow/tensorflow:1.5.0-devel-gpu \
      --sync-mode=git \
      --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
      "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --max_steps 10000 --data_dir
=code/tensorflow-sample-code/data"
```

3. 执行以下命令列出当前用户提交的作业。

```
arena list
```

4. 执行以下命令再提交一个一张GPU卡的训练作业。

```
arena submit tf \
    --name=tf-git-bob-02 \
    --gpus=1 \
    --image=tensorflow/tensorflow:1.5.0-devel-gpu \
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
    "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --max_steps 10000 --data_dir
=code/tensorflow-sample-code/data"
```

由于本文示例只给dev1组分配了一张GPU卡，所以预期这个作业不会被调度起来。

可以看到，虽然整个集群还有剩余资源，但是由于bob所在的组只被分配了一张GPU卡资源，并且当前已经有了一个作业占用了资源，所以bob后续的提交的作业就会被暂停。

用户tom

1. 执行以下命令登录Client机器，查看当前可以用共享数据卷。

```
# 使用tom账号登录Client机器。
ssh tom@39.98.xxx.xx

# 通过arena data list查看当前用户可用共享数据卷。
arena data list
```

2. 执行以下命令列出当前用户提交的作业。

```
arena list
```

此时看不到bob提交的作业。

3. 执行以下命令提交一个1张GPU卡的训练作业。

```
arena submit tf \
    --name=tf-git-tom-01 \
    --gpus=1 \
    --chief-cpu=2 \
    --chief-memory=10Gi \
    --image=tensorflow/tensorflow:1.5.0-devel-gpu \
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
    "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --max_steps 10000 --data_dir
=code/tensorflow-sample-code/data"
```

说明 由于示例分配了组dev2的GPU、CPU、MEM资源，因此在dev2组里的用户提交作业的时候需要明确指定GPU、CPU、MEM的使用资源。

4. 执行以下命令再次提交一个1张GPU卡的训练作业。

```
arena submit tf \
    --name=tf-git-tom-02 \
    --gpus=1 \
    --chief-cpu=2 \
    --chief-memory=10Gi \
    --image=tensorflow/tensorflow:1.5.0-devel-gpu \
    --sync-mode=git \
    --sync-source=https://code.aliyun.com/xiaozhou/tensorflow-sample-code.git \
    "python code/tensorflow-sample-code/tfjob/docker/mnist/main.py --max_steps 10000 --data_dir
=code/tensorflow-sample-code/data"
```

5. 执行以下命令列出当前用户提交的作业。

```
arena list
```

执行结果

从上述操作可以看到，目前分别位于两个组的用户bob、tom可以通过自己的账号登录Client机器独立地使用Arena，并通过Arena查看和使用用户当前所在组可访问的存储资源和计算资源，以及管理各自的作业。

操作视频

附录

下载tools.tar.gz的命令如下：

```
wget https://lumo-package.oss-cn-beijing.aliyuncs.com/tools.tar.gz
```

2.大数据解决方案

2.1. ACK中运行Spark工作负载

2.1.1. ACK中运行Apache Spark概述

Apache Spark是一个在数据分析领域广泛使用的开源项目，它常被应用于众所周知的大数据和机器学习工作负载中。从Apache Spark 2.3.0版本开始，您可以在Kubernetes上运行和管理Spark资源。本文介绍在ACK上运行Apache Spark的优势及涉及的产品介绍。

在ACK上运行Apache Spark工作负载的优势

Kubernetes是一个开源容器管理系统，可以提供应用发布、运维、扩缩等机制。容器服务Kubernetes版ACK（Alibaba Cloud Container Service for Kubernetes）是全球首批通过Kubernetes一致性认证的服务平台，提供高性能可伸缩的容器应用管理服务，支持企业级Kubernetes容器化应用的生命周期管理。简化集群的搭建和扩容等运维工作，整合阿里云虚拟化、存储、网络和安全能力，打造云端最佳的Kubernetes容器化应用运行环境。您可以在ACK上运行微服务、批量处理、机器学习等各种工作负载。

在ACK上运行Spark应用具有以下优势：

- 通过把Spark应用和依赖项打包成容器，您可享受容器的各种优点，解决Hadoop版本不匹配和兼容性问题，还可以给容器镜像打上标签控制版本。如果需要测试不同版本的Spark或者依赖项的话，选择对应的版本即可。
- 重用Kubernetes生态的各种组件，比如监控、日志。把Spark工作负载部署在已有的的Kubernetes基础设施中，能够快速开始工作，大大减少运维成本。
- 支持多租户，可利用Kubernetes的Namespace和ResourceQuota做用户粒度的资源调度，利用Kubernetes的节点选择机制保证Spark工作负载得到专用的资源。另外，由于Driver Pods创建Executor Pods，您可以用Kubernetes Service Account控制权限，利用Role或者Cluster Role定义细粒度访问权限，安全地运行工作负载，避免受其他工作负载影响。
- 把Spark和管理数据生命周期的应用运行在同一个集群中，可以使用单个编排机制构建端到端生命周期的解决方案，并能复制到其他区域部署，甚至是在私有化环境部署。

在ACK上运行Apache Spark工作负载涉及的工具

- **Spark on Kubernetes Operator**
  Spark on Kubernetes Operator帮助您在Kubernetes上像其他工作负载一样用通用的方式方便地运行Spark应用。它使用Kubernetes custom resource来配置、运行Spark应用，并展现其状态。您需要Spark 2.3及以上的版本来支持Kubernetes调度。
- **Alluxio**
  Alluxio一个面向基于云的数据分析和人工智能的开源的数据编排技术。 它为数据驱动型应用和存储系统构建了桥梁，将数据从存储层移动到距离数据驱动型应用更近的位置从而能够更容易被访问。 这还使得应用程序能够通过一个公共接口连接到许多存储系统。Alluxio内存至上的层次化架构使得数据的访问速度能比现有方案快几个数量级。
  在大数据生态系统中，Alluxio位于数据驱动框架或应用（例如Apache Spark、Presto、Tensorflow、Apache HBase、Apache Hive、Apache Flink）和各种持久化存储系统（例如Amazon S3、Google Cloud Storage、OpenStack Swift、HDFS、GlusterFS、IBM Cleversafe、EMC ECS、Ceph、NFS 、Minio、Alibaba OSS）之间。 Alluxio统一了存储在这些不同存储系统中的数据，为其上层数据驱动型应用提供统一的客户端API和全局命名空间。

  Spark on ACK的大数据解决方案将会采用Alluxio通过缓存的方式加速Spark访问持久化存储系统中的数据。

- **TPC-DS Benchmark**
  TPC-DS由第三方社区创建和维护，是性能压测，协助确定解决方案的工业标准。这个测试集包含对大数据集的统计、报表生成、联机查询、数据挖掘等功能的复杂应用，测试用的数据和值是有倾斜的，与真实数据一致。可以说TPC-DS是与真实场景非常接近的一个测试集，也是难度较大的一个测试集。
  TPC-DS包含104个Query，覆盖了SQL 2003的大部分标准，有99条压测Query，其中的4条Query各有2个变体（14,23,24,39），最后还有一个 `s_max` Query进行全量扫描和最大的一些表的聚合。
  TPC-DS基准测试有以下几个主要特点：
  - 遵循SQL 2003的语法标准，SQL案例比较复杂。
  - 分析的数据量大，并且测试案例是在回答真实的商业问题。
  - 测试案例中包含各种业务模型（例如分析报告型，迭代式的联机分析型，数据挖掘型等）。
  - 几乎所有的测试案例都有很高的IO负载和CPU计算需求。

  ACK中运行Apache Spark的大数据解决方案采用TPC-DS来评测Spark在ACK上的性能。

相关文档

- 搭建测试环境
- 开发测试代码
- 在ACK上运行Spark Benchmark
- 分析测试结果
- 定位排查问题

2.1.2. 搭建测试环境

在ACK上运行Spark作业，并用Alluxio进行分布式加速，可以通过ACK Spark Operator来简化提交作业的操作。同时ACK也提供了Spark History Server用来记录作业历史数据，方便排查问题。本文介绍如何在ACK上搭建Spark运行环境。

背景信息

在ACK上搭建Spark运行的环境包括：

- 创建ACK集群
- 创建OSS存储空间
- 安装ack-spark-operator
- 安装ack-spark-history-server
- 安装Alluxio

创建ACK集群

有关创建ACK集群的具体操作，请参见创建Kubernetes托管版集群。

> **注意**
> 配置集群参数时，您需注意以下内容：
> - 设置Worker节点的实例规格时，选择**大数据网络增强型**的ecs.d1ne.6xlarge规格，并且设置节点数量为20。
> - 每个ecs.d1ne.6xlarge节点实例自带12块5 TB的HDD数据盘。您需要对这12个数据盘进行分区格式化挂载。有关操作的详情，请参见分区格式化大于2 TiB数据盘。
> - 格式化并挂载完成后，执行 `df -h` 可以下图挂载详情的信息。
> - */mnt*目录下的12个文件路径会在Alluxio中用到。当集群节点多时，手工挂载数据盘的操作十分繁琐，ACK简化了操作方式。详情请参见通过LVM数据卷管理本地存储。

挂载详情

挂载详情

创建OSS存储空间

您需要创建一个OSS存储空间（Bucket）用来存放TPC-DS生成的数据、测试结果和测试过程中的日志等。本文示例中设置Bucket名称为*cloudnativeai*。有关创建OSS Bucket的具体操作步骤，请参见创建存储空间。

安装ack-spark-operator

通过安装**ack-spark-operator**组件，您可以使用ACK Spark Operator简化提交作业的操作。

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，选择**市场 > 应用目录**。
3. 在**应用目录**页面，找到并单击**ack-spark-operator**。
4. 在**应用目录 - ack-spark-operator**页面右侧，单击**创建**。

安装ack-spark-history-server

ACK Spark History Server通过记录Spark执行任务过程中的日志和事件信息，并提供UI界面，帮助排查问题。

在创建**ack-spark-history-server**组件时，您需在**参数**页签配置OSS相关的信息，用于存储Spark历史数据。有关创建**ack-spark-history-server**详细步骤，请参见安装ack-spark-operator。

配置OSS信息如下。

```
oss:
  enableOSS: false
  # Please input your accessKeyId
  alibabaCloudAccessKeyId: ""
  # Please input your accessKeySecret
  alibabaCloudAccessKeySecret: ""
  # oss bucket endpoint such as oss-cn-beijing.aliyuncs.com
  alibabaCloudOSSEndpoint: ""
  # oss file path such as oss://bucket-name/path
  eventsDir: "oss://cloudnativeai/spark/spark-events"
```

安装完成后，执行以下命令查看是否安装成功。

```
kubectl get service ack-spark-history-server -n {YOUR-NAMESPACE}
```

安装Alluxio

在ACK中，您需要通过**Helm**命令安装Alluxio。

1. 首先执行以下命令下载Allxuio。

   ```
   wget http://kubeflow.oss-cn-beijing.aliyuncs.com/alluxio-0.6.8.tgz
   tar -xvf alluxio-0.6.8.tgz
   ```

2. 在Alluxio同级目录下新建并配置*config.yaml*文件。
   有关完整的配置信息，请参见config.yaml。

关键配置如下：

- 其中以下AccessKey、endpoint、挂载的UFS等信息需要替换成您创建的OSS相关信息。

```
# Site properties for all the components
properties:
  fs.oss.accessKeyId: YOUR-ACCESS-KEY-ID
  fs.oss.accessKeySecret: YOUR-ACCESS-KEY-SECRET
  fs.oss.endpoint: oss-cn-beijing-internal.aliyuncs.com
  alluxio.master.mount.table.root.ufs: oss://cloudnativeai/
  alluxio.master.persistence.blacklist: .staging,_temporary
  alluxio.security.stale.channel.purge.interval: 365d
  alluxio.user.metrics.collection.enabled: 'true'
  alluxio.user.short.circuit.enabled: 'true'
  alluxio.user.file.write.tier.default: 1
  alluxio.user.block.size.bytes.default: 64MB #default 64MB
  alluxio.user.file.writetype.default: CACHE_THROUGH
  alluxio.user.file.metadata.load.type: ONCE
  alluxio.user.file.readtype.default: CACHE
  #alluxio.worker.allocator.class: alluxio.worker.block.allocator.MaxFreeAllocator
  alluxio.worker.allocator.class: alluxio.worker.block.allocator.RoundRobinAllocator
  alluxio.worker.file.buffer.size: 128MB
  alluxio.worker.evictor.class: alluxio.worker.block.evictor.LRUEvictor
  alluxio.job.master.client.threads: 5000
  alluxio.job.worker.threadpool.size: 300
```

- tieredstore中的mediumtype、path对应ACK Worker节点中挂载的数据盘。

```
tieredstore:
  levels:
    - level: 0
      alias: HDD
      mediumtype: HDD-0,HDD-1,HDD-2,HDD-3,HDD-4,HDD-5,HDD-6,HDD-7,HDD-8,HDD-9,HDD-10,HDD-11
      path: /mnt/disk1,/mnt/disk2,/mnt/disk3,/mnt/disk4,/mnt/disk5,/mnt/disk6,/mnt/disk7,/mnt/disk8,/mnt/disk9,/mnt/disk10,/mnt/disk11,/mnt/disk12
      type: hostPath
      quota: 1024G,1024G,1024G,1024G,1024G,1024G,1024G,1024G,1024G,1024G,1024G,1024G
      high: 0.95
      low: 0.7
```

3. 为ACK集群的Worker节点设置alluxio=true的标签。有关添加标签的具体操作步骤，请参见管理节点标签。
4. 执行以下**Helm**命令安装Alluxio。

```
kubectl create namespace alluxio
helm install -f config.yaml -n alluxio alluxio alluxio
```

5. 执行以下命令查看是否成功安装Alluxio。

```
kubectl get pod -n alluxio
```

6. 执行以下命令进入alluxio-admin，查看是否成功挂载数据盘。

```
kubectl exec -it alluxio-master-0 -n alluxio -- /bin/bash
./bin/alluxio fsadmin report capacity
```

如果能看到每个Worker节点上都有挂载的数据盘，说明Alluxio安装配置成功。

```
Alluxio
```

后续步骤

开发测试代码

2.1.3. 开发测试代码

在进行Spark Benchmark前，您需要编写通过TPC-DS生成测试数据和执行Spark SQL的代码。本文介绍如何编写代码以及制作镜像。

前提条件

搭建测试环境

背景信息

为了方便测试，本文已经提供了预制镜像（registry.cn-beijing.aliyuncs.com/yukong/ack-spark-benchmark:1.0.0），您可以直接使用。有关镜像的完整代码，请参见benchmark-for-spark。

准备工作

测试代码依赖Databricks两个工具：一个是TPC-DS测试包，另一个是测试数据生成工具tpcds-kit。

1. 执行以下命令打包TPC-DS依赖JAR文件。

```
git clone https://github.com/databricks/spark-sql-perf.git
sbt package
```

生成JAR包*spark-sql-perf_2.11-0.5.1-SNAPSHOT*，作为测试项目的依赖。

2. 执行以下命令编译tpcds-kit。

```
git clone https://github.com/davies/tpcds-kit.git
yum install gcc gcc-c++ bison flex cmake ncurses-devel
cd tpcds-kit/tools
cp Makefile.suite Makefile  #复制Makefile.suite为Makefile。
make
#验证
./dsqgen --help
```

编译后生成二进制可执行程序，本文示例主要依赖两个：dsdgen（数据生成）和dsqgen（查询生

成）。

编写代码

1. 使用以下代码创建并配置*DataGeneration.scala*文件，用来生成数据。

```
package com.aliyun.spark.benchmark.tpcds

import com.databricks.spark.sql.perf.tpcds.TPCDSTables
import org.apache.log4j.{Level, LogManager}
import org.apache.spark.sql.SparkSession

import scala.util.Try

object DataGeneration {
  def main(args: Array[String]) {
    val tpcdsDataDir = args(0)
    val dsdgenDir = args(1)
    val format = Try(args(2).toString).getOrElse("parquet")
    val scaleFactor = Try(args(3).toString).getOrElse("1")
    val genPartitions = Try(args(4).toInt).getOrElse(100)
    val partitionTables = Try(args(5).toBoolean).getOrElse(false)
    val clusterByPartitionColumns = Try(args(6).toBoolean).getOrElse(false)
    val onlyWarn = Try(args(7).toBoolean).getOrElse(false)

    println(s"DATA DIR is $tpcdsDataDir")
    println(s"Tools dsdgen executable located in $dsdgenDir")
    println(s"Scale factor is $scaleFactor GB")

    val spark = SparkSession
      .builder
      .appName(s"TPCDS Generate Data $scaleFactor GB")
      .getOrCreate()

    if (onlyWarn) {
      println(s"Only WARN")
      LogManager.getLogger("org").setLevel(Level.WARN)
    }

    val tables = new TPCDSTables(spark.sqlContext,
      dsdgenDir = dsdgenDir,
      scaleFactor = scaleFactor,
      useDoubleForDecimal = false,
```

```
    useStringForDate = false)

  println(s"Generating TPCDS data")

  tables.genData(
    location = tpcdsDataDir,
    format = format,
    overwrite = true, // overwrite the data that is already there
    partitionTables = partitionTables,  // create the partitioned fact tables
    clusterByPartitionColumns = clusterByPartitionColumns, // shuffle to get partitions coalesced into sin
gle files.
    filterOutNullPartitionValues = false, // true to filter out the partition with NULL key value
    tableFilter = "", // "" means generate all tables
    numPartitions = genPartitions) // how many dsdgen partitions to run - number of input tasks.

  println(s"Data generated at $tpcdsDataDir")

  spark.stop()
 }
}
```

2. 使用以下代码创建并配置*BenchmarkSQL.scala*文件，用来查询数据。

```
package com.aliyun.spark.benchmark.tpcds

import com.databricks.spark.sql.perf.tpcds.{TPCDS, TPCDSTables}
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions.col
import org.apache.log4j.{Level, LogManager}
import scala.util.Try

object BenchmarkSQL {
 def main(args: Array[String]) {
  val tpcdsDataDir = args(0)
  val resultLocation = args(1)
  val dsdgenDir = args(2)
  val format = Try(args(3).toString).getOrElse("parquet")
  val scaleFactor = Try(args(4).toString).getOrElse("1")
  val iterations = args(5).toInt
  val optimizeQueries = Try(args(6).toBoolean).getOrElse(false)
```

```
  val filterQueries = Try(args(7).toString).getOrElse("")
  val onlyWarn = Try(args(8).toBoolean).getOrElse(false)

  val databaseName = "tpcds_db"
  val timeout = 24*60*60

  println(s"DATA DIR is $tpcdsDataDir")

  val spark = SparkSession
    .builder
    .appName(s"TPCDS SQL Benchmark $scaleFactor GB")
    .getOrCreate()

  if (onlyWarn) {
    println(s"Only WARN")
    LogManager.getLogger("org").setLevel(Level.WARN)
  }

  val tables = new TPCDSTables(spark.sqlContext,
    dsdgenDir = dsdgenDir,
    scaleFactor = scaleFactor,
    useDoubleForDecimal = false,
    useStringForDate = false)

  if (optimizeQueries) {
    Try {
      spark.sql(s"create database $databaseName")
    }
    tables.createExternalTables(tpcdsDataDir, format, databaseName, overwrite = true, discoverPartition
s = true)
    tables.analyzeTables(databaseName, analyzeColumns = true)
    spark.conf.set("spark.sql.cbo.enabled", "true")
  } else {
    tables.createTemporaryTables(tpcdsDataDir, format)
  }

  val tpcds = new TPCDS(spark.sqlContext)

  var query_filter : Seq[String] = Seq()
  if (!filterQueries.isEmpty) {
    println(s"Running only queries: $filterQueries")
```

```
    query_filter = filterQueries.split(",").toSeq
  }

  val filtered_queries = query_filter match {
    case Seq() => tpcds.tpcds2_4Queries
    case _ => tpcds.tpcds2_4Queries.filter(q => query_filter.contains(q.name))
  }

  // Start experiment
  val experiment = tpcds.runExperiment(
    filtered_queries,
    iterations = iterations,
    resultLocation = resultLocation,
    forkThread = true)

  experiment.waitForFinish(timeout)

  // Collect general results
  val resultPath = experiment.resultPath
  println(s"Reading result at $resultPath")
  val specificResultTable = spark.read.json(resultPath)
  specificResultTable.show()

  // Summarize results
  val result = specificResultTable
    .withColumn("result", explode(col("results")))
    .withColumn("executionSeconds", col("result.executionTime")/1000)
    .withColumn("queryName", col("result.name"))
  result.select("iteration", "queryName", "executionSeconds").show()
  println(s"Final results at $resultPath")

  val aggResults = result.groupBy("queryName").agg(
    callUDF("percentile", col("executionSeconds").cast("long"), lit(0.5)).as('medianRuntimeSeconds),
    callUDF("min", col("executionSeconds").cast("long")).as('minRuntimeSeconds),
    callUDF("max", col("executionSeconds").cast("long")).as('maxRuntimeSeconds)
  ).orderBy(col("queryName"))
  aggResults.repartition(1).write.csv(s"$resultPath/summary.csv")
  aggResults.show(105)

  spark.stop()
}
```

```
}
```

镜像制作

测试代码编译成JAR后，可以和依赖的其他JAR包一起，制作成镜像供测试使用。Dockerfile文件如下。

```
FROM registry.cn-hangzhou.aliyuncs.com/acs/spark:ack-2.4.5-f757ab6
RUN mkdir -p /opt/spark/jars
RUN mkdir -p /tmp/tpcds-kit
COPY ./target/scala-2.11/ack-spark-benchmark-assembly-0.1.jar /opt/spark/jars/
COPY ./lib/*.jar /opt/spark/jars/
COPY ./tpcds-kit/tools.tar.gz /tmp/tpcds-kit/
RUN cd /tmp/tpcds-kit/ && tar -xzvf tools.tar.gz
```

后续步骤

在ACK上运行Spark Benchmark

2.1.4. 在ACK上运行Spark Benchmark

准备好测试环境和镜像后，您就可以在ACK上运行Spark Benchmark作业。本文介绍如何生成测试数据和利用测试数据进行Benchmark，以及通过Alluxio进行分布式缓存加速的方法。

前提条件

开发测试代码

测试说明

本文测试先生成1 TB数据，然后分别测试Spark直接从OSS读取数据，和通过Alluxio冷、热缓存做加速的三种情况。

生成1 TB数据

执行DataGeneration.scala生成1 TB数据，并将数据存在OSS上，然后Spark SQL查询任务会用到这些数据。具体操作步骤如下。

1. 使用以下模板创建*tpcds-data-generator.yaml*文件。

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: tpcds-data-generation
  namespace: default
spec:
  type: Scala
  image: registry.cn-beijing.aliyuncs.com/yukong/ack-spark-benchmark:1.0.0
  sparkVersion: 2.4.5
  mainClass: com.aliyun.spark.benchmark.tpcds.DataGeneration
  mainApplicationFile: "local:///opt/spark/jars/ack-spark-benchmark-assembly-0.1.jar"
  mode: cluster
```

```
  arguments:
    # TPC-DS data localtion
    - "oss://cloudnativeai/spark/data/tpc-ds-data/1000g"
    # Path to kit in the docker image
    - "/tmp/tpcds-kit/tools"
    # Data Format
    - "parquet"
    # Scale factor (in GB)
    - "100000"
    # Generate data num partitions
    - "100"
    # Create the partitioned fact tables
    - "false"
    # Shuffle to get partitions coalesced into single files.
    - "false"
    # Logging set to WARN
    - "true"
  hadoopConf:
    # OSS
    "fs.oss.impl": "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem"
    "fs.oss.endpoint": "oss-cn-beijing-internal.aliyuncs.com"
    "fs.oss.accessKeyId": "YOUR-ACCESS-KEY-ID"
    "fs.oss.accessKeySecret": "YOUR-ACCESS-KEY-SECRET"
  sparkConf:
    "spark.kubernetes.allocation.batch.size": "100"
    "spark.sql.adaptive.enabled": "true"
    "spark.eventLog.enabled": "true"
    "spark.eventLog.dir": "oss://cloudnativeai/spark/spark-events"
  driver:
    cores: 6
    memory: "20480m"
    serviceAccount: spark
  executor:
    instances: 20
    cores: 8
    memory: "61440m"
    memoryOverhead: 2g
  restartPolicy:
    type: Never
```

2. 执行以下命令生成数据。

```
kubectl apply -f tpcds-data-generator.yaml
```

执行Benchmark任务

查询任务分三次：

1. 第一次直接用Spark读取OSS上的1 TB数据，运行Benchmark。
2. 第二次利用Alluxio做分布式缓存，OSS上的数据会先加载到Alluxio中，Spark从Alluxio中读取缓存数据运行Benchmark。
3. 第三次修改第二次任务中的OSS结果存放路径，此时Alluxio中的缓存数据还在，然后重新运行Benchmark。

通过这三次任务对比，可以看到使用Alluxio缓存加速后，有较大的性能提升。具体操作步骤如下。

1. 从OSS中读取数据，运行Benchmark。
   i. 使用以下模板创建并部署*tpcds-benchmark.yaml*文件。

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: tpcds-benchmark
  namespace: default
spec:
  type: Scala
  mode: cluster
  image: registry.cn-beijing.aliyuncs.com/yukong/ack-spark-benchmark:1.0.0
  imagePullPolicy: Always
  sparkVersion: 2.4.5
  mainClass: com.aliyun.spark.benchmark.tpcds.BenchmarkSQL
  mainApplicationFile: "local:///opt/spark/jars/ack-spark-benchmark-assembly-0.1.jar"
  arguments:
    # TPC-DS data localtion
    - "oss://cloudnativeai/spark/data/tpc-ds-data/1000g"
    # results location
    - "oss://cloudnativeai/spark/result/tpcds-benchmark-result-1000g"
    # Path to kit in the docker image
    - "/tmp/tpcds-kit/tools"
    # Data Format
    - "parquet"
    # Scale factor (in GB)
    - "1000"
    # Number of iterations
    - "1"
    # Optimize queries
    - "false"
```

```
  # Filter queries, will run all if empty - "q70-v2.4,q82-v2.4,q64-v2.4"
  - ""
  # Logging set to WARN
  - "true"
hostNetwork: true
dnsPolicy: ClusterFirstWithHostNet
restartPolicy:
  type: Never
timeToLiveSeconds: 86400
hadoopConf:
  # OSS
  "fs.oss.impl": "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem"
  "fs.oss.endpoint": "oss-cn-beijing-internal.aliyuncs.com"
  "fs.oss.accessKeyId": "YOUR-ACCESS-KEY-ID"
  "fs.oss.accessKeySecret": "YOUR-ACCESS-KEY-SECRET"
sparkConf:
  "spark.kubernetes.allocation.batch.size": "200"
  "spark.sql.adaptive.join.enabled": "true"
  "spark.eventLog.enabled": "true"
  "spark.eventLog.dir": "oss://cloudnativeai/spark/spark-events"
driver:
  cores: 5
  memory: "20480m"
  labels:
    version: 2.4.5
    spark-app: spark-tpcds
    role: driver
  serviceAccount: spark
executor:
  cores: 7
  instances: 20
  memory: "20480m"
  memoryOverhead: "8g"
  labels:
    version: 2.4.5
    role: executor
```

ii. 执行以下命令，运行Benchmark任务。

```
kubectl apply -f tpcds-benchmark.yaml
```

2. 用Alluxio冷缓存，运行Benchmark。

i. 使用以下模板创建并部署*tpcds-benchmark-with-alluxio.yaml*文件。

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: tpcds-benchmark-sql
  namespace: default
spec:
  type: Scala
  mode: cluster
  image: registry.cn-beijing.aliyuncs.com/yukong/ack-spark-benchmark:1.0.0
  imagePullPolicy: Always
  sparkVersion: 2.4.5
  mainClass: com.aliyun.spark.benchmark.tpcds.BenchmarkSQL
  mainApplicationFile: "local:///opt/spark/jars/ack-spark-benchmark-assembly-0.1.jar"
  arguments:
    # TPC-DS data localtion
    - "alluxio://alluxio-master-0.alluxio.svc.cluster.local:19998/spark/data/tpc-ds-data/1000g"
    # results location
    - "oss://cloudnativeai/spark/result/tpcds-benchmark-result-1000g-alluxio"
    # Path to kit in the docker image
    - "/tmp/tpcds-kit/tools"
    # Data Format
    - "parquet"
    # Scale factor (in GB)
    - "1000"
    # Number of iterations
    - "1"
    # Optimize queries
    - "false"
    # Filter queries, will run all if empty - "q70-v2.4,q82-v2.4,q64-v2.4"
    - ""
    # Logging set to WARN
    - "true"
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
  restartPolicy:
    type: Never
  timeToLiveSeconds: 86400
  hadoopConf:
    # OSS
```

```
  "fs.oss.impl": "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem"
  "fs.oss.endpoint": "oss-cn-beijing-internal.aliyuncs.com"
  "fs.oss.accessKeyId": "YOUR-ACCESS-KEY-ID"
  "fs.oss.accessKeySecret": "YOUR-ACCESS-KEY-SECRET"
sparkConf:
  "spark.kubernetes.allocation.batch.size": "200"
  "spark.sql.adaptive.join.enabled": "true"
  "spark.eventLog.enabled": "true"
  "spark.eventLog.dir": "oss://cloudnativeai/spark/spark-events"
volumes:
  - name: "spark-local-dir-1"
    hostPath:
      path: "/mnt/disk1"
      type: Directory
  - name: "spark-local-dir-2"
    hostPath:
      path: "/mnt/disk2"
      type: Directory
  - name: "spark-local-dir-3"
    hostPath:
      path: "/mnt/disk3"
      type: Directory
  - name: "spark-local-dir-4"
    hostPath:
      path: "/mnt/disk4"
      type: Directory
  - name: "spark-local-dir-5"
    hostPath:
      path: "/mnt/disk5"
      type: Directory
  - name: "spark-local-dir-6"
    hostPath:
      path: "/mnt/disk6"
      type: Directory
  - name: "spark-local-dir-7"
    hostPath:
      path: "/mnt/disk7"
      type: Directory
  - name: "spark-local-dir-8"
    hostPath:
      path: "/mnt/disk8"
```

```
      type: Directory
  - name: "spark-local-dir-9"
    hostPath:
      path: "/mnt/disk9"
      type: Directory
  - name: "spark-local-dir-10"
    hostPath:
      path: "/mnt/disk10"
      type: Directory
  - name: "spark-local-dir-11"
    hostPath:
      path: "/mnt/disk11"
      type: Directory
  - name: "spark-local-dir-12"
    hostPath:
      path: "/mnt/disk12"
      type: Directory
driver:
  cores: 5
  memory: "20480m"
  labels:
    version: 2.4.5
    spark-app: spark-tpcds
    role: driver
  serviceAccount: spark
executor:
  cores: 7
  instances: 20
  memory: "20480m"
  memoryOverhead: "8g"
  labels:
    version: 2.4.5
    role: executor
  volumeMounts:
    - name: "spark-local-dir-1"
      mountPath: "/mnt/disk1"
    - name: "spark-local-dir-2"
      mountPath: "/mnt/disk2"
    - name: "spark-local-dir-3"
      mountPath: "/mnt/disk3"
    - name: "spark-local-dir-4"
```

```
    mountPath: "/mnt/disk4"
  - name: "spark-local-dir-5"
    mountPath: "/mnt/disk5"
  - name: "spark-local-dir-6"
    mountPath: "/mnt/disk6"
  - name: "spark-local-dir-7"
    mountPath: "/mnt/disk7"
  - name: "spark-local-dir-8"
    mountPath: "/mnt/disk8"
  - name: "spark-local-dir-9"
    mountPath: "/mnt/disk9"
  - name: "spark-local-dir-10"
    mountPath: "/mnt/disk10"
  - name: "spark-local-dir-11"
    mountPath: "/mnt/disk11"
  - name: "spark-local-dir-12"
    mountPath: "/mnt/disk12"
```

ii. 执行以下命令，开始通过Alluxio缓存加速的Benchmark任务。

```
kubectl apply -f tpcds-benchmark-with-alluxio.yaml
```

3. 用Alluxio热缓存，运行Benchmark。
   在第一次通过Alluxio做缓存加速时，Benchmark会从OSS读取数据，但是缓存在Alluxio中，所以读取速度比较慢。在第一次使用Alluxio缓存加速Benchmark测试完后，您可以再测试几次对比效果。

后续步骤

分析测试结果

2.1.5. 分析测试结果

本文分析Spark在ACK上运行1 TB数据的Spark SQL作业和在采用Alluxio分布式缓存加速后的性能对比。

前提条件

在ACK上运行Spark Benchmark

硬件配置

ACK集群配置说明如下表。

集群类型	ACK标准专有集群

集群类型	ACK标准专有集群
ECS实例	• ECS规格：ecs.d1ne.6xlarge • Alibaba Cloud Linux 2.1903 • CPU：24核 • 内存：96 GB • 数据盘：5500 GB，HDD类型
Worker节点个数	20

软件配置

- 软件版本
  - Spark version：2.4.5
  - Alluxio version：2.3.0
- Spark配置说明

配置	参数
spark.driver.cores	5
spark.driver.memory (MB)	20480
spark.executor.cores	7
spark.executor.memory (MB)	20480
spark.executor.instances	20

测试结果

1 TB数据单并发Benchmark如下表。

Benchmark	104 Query总耗时（Min）
Spark with OSS	180
Spark with Alluxio Cold	145
Spark with Alluxio Warm	137

其中每条SQL耗时对比如下图。

结果分析

从测试结果中可以看到，使用Alluxio做缓存加速后，整体性能较直接从OSS读取数据有了较大提升。Alluxio初次测试时需要从OSS加载数据到缓存，性能会稍差。如后面再测试，ACK中运行Spark作业性能有会提升。

后续步骤

定位排查问题

2.1.6. 定位排查问题

在测试过程中，您会发现错误参数配置容易导致Spark作业异常情况出现。本文介绍如何通过Spark UI和ACK Spark History Server查看作业执行情况和排查问题。

前提条件

分析测试结果

访问Spark UI

任务运行过程中，可以通过Spark UI实时看到每个SQL的执行情况。操作步骤如下。

1. 执行命令**kubectl get services**。

   其中**tpcds-query-runner-with-alluxio-ui-svc**即Spark UI对应的Service。

2. 执行以下命令在本地访问Spark UI页面。

   ```
   kubectl port-forward service/tpcds-query-runner-with-alluxio-ui-svc 4040:4040
   ```

   输出以下内容。

3. 在浏览器中输入*localhost:4040*查看Spark UI中的任务执行情况。

访问ACK Spark History Server

任务结束后，如果想看到历史数据，可以采用以下操作方式。

1. 执行以下命令，获取SparkApplication任务ID。

   ```
   kubectl get sparkapplication tpcds-query-runner-with-alluxio -o yaml
   ```

   sparkApplicationId信息显示如下图。

2. 执行以下命令获取ACK Spark History Server的endpoint。

   ```
   kubectl get service ack-spark-history-server
   ```

3. 在浏览器中输入**EXTERNAL-IP**类型IP（39.XX.XX.XXX）和端口。您可以看到所有Spark任务的历史数据，再通过步骤一中获取的**sparkApplicationId**信息找到对应记录，即可查看作业执行情况和排查问题。

2.1.7. 通过LVM数据卷管理本地存储

在大数据场景下，选用ecs.d1ne.6xlarge规格的ECS实例运行Spark作业。每个实例自带12块5 TB的HDD数据盘，需要对这12个数据盘进行手动分区格式化和挂载。当节点数量多时，手动分区格式化和挂载是一个十分繁琐和耗时的操作。本文介绍如何通过LVM数据卷对数据盘格式化和挂载进行简化。

前提条件

- 创建Kubernetes托管版集群。

- 集群Worker节点的数据盘设备未经过手动格式化和挂载。

部署LVM插件

步骤一：使用以下模板部署LVM CSI Plugin。 ❯
步骤二：使用以下模板部署LVM CSI Provisioner。 ❯
步骤三：使用以下模板部署Node Storage Manager。 ❯
步骤四：使用以下模板部署ConfigMap。 ❯

配置PVC

1. 使用以下模板配置StorageClass。

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: csi-local-imm
parameters:
 fsType: ext4
 lvmType: striping
 vgName: volumegroup1
 volumeType: LVM
provisioner: localplugin.csi.alibabacloud.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

2. 使用以下模板配置PVC。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: lvm-pvc-im
spec:
 accessModes:
 - ReadWriteOnce
 resources:
  requests:
   storage: 12288Gi
 storageClassName: csi-local-imm
```

配置Alluxio

使用以下模板配置Alluxio。

```
# The Alluxio Open Foundation licenses this work under the Apache License, version 2.0
# (the "License"). You may not use this work except in compliance with the License, which is
```

```
# available at www.apache.org/licenses/LICENSE-2.0
#
# This software is distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
# either express or implied, as more fully set forth in the License.
#
# See the NOTICE file distributed with this work for information regarding copyright ownership.
#

# This should not be modified in the usual case.
fullnameOverride: alluxio



## Common ##

# Docker Image
image: registry-vpc.cn-beijing.aliyuncs.com/alluxio/alluxio
imageTag: 2.3.0
imagePullPolicy: IfNotPresent

# Security Context
user: 0
group: 0
fsGroup: 0

# Site properties for all the components
properties:
 fs.oss.accessKeyId: YOUR-ACCESS-KEY-ID
 fs.oss.accessKeySecret: YOUR-ACCESS-KEY-SECRET
 fs.oss.endpoint: oss-cn-beijing-internal.aliyuncs.com
 alluxio.master.mount.table.root.ufs: oss://cloudnativeai/
 alluxio.master.persistence.blacklist: .staging,_temporary
 alluxio.security.stale.channel.purge.interval: 365d
 alluxio.user.metrics.collection.enabled: 'true'
 alluxio.user.short.circuit.enabled: 'true'
 alluxio.user.file.write.tier.default: 1
 alluxio.user.block.size.bytes.default: 64MB #default 64MB
 alluxio.user.file.writetype.default: CACHE_THROUGH
 alluxio.user.file.metadata.load.type: ONCE
 alluxio.user.file.readtype.default: CACHE
 #alluxio.worker.allocator.class: alluxio.worker.block.allocator.MaxFreeAllocator
 alluxio.worker.allocator.class: alluxio.worker.block.allocator.RoundRobinAllocator
```

```
  alluxio.worker.file.buffer.size: 128MB
  alluxio.worker.evictor.class: alluxio.worker.block.evictor.LRUEvictor
  alluxio.job.master.client.threads: 5000
  alluxio.job.worker.threadpool.size: 300

# Recommended JVM Heap options for running in Docker
# Ref: https://developers.redhat.com/blog/2017/03/14/java-inside-docker/
# These JVM options are common to all Alluxio services
# jvmOptions:
#   - "-XX:+UnlockExperimentalVMOptions"
#   - "-XX:+UseCGroupMemoryLimitForHeap"
#   - "-XX:MaxRAMFraction=2"

# Mount Persistent Volumes to all components
# mounts:
# - name: <persistentVolume claimName>
#   path: <mountPath>

# Use labels to run Alluxio on a subset of the K8s nodes

## Master ##

master:
  count: 1 # Controls the number of StatefulSets. For multiMaster mode increase this to >1.
  replicas: 1 # Controls #replicas in a StatefulSet and should not be modified in the usual case.
  enableLivenessProbe: false
  enableReadinessProbe: false
  args: # Arguments to Docker entrypoint
    - master-only
    - --no-format
  # Properties for the master component
  properties:
  # Example: use ROCKS DB instead of Heap
  # alluxio.master.metastore: ROCKS
  # alluxio.master.metastore.dir: /metastore
  resources:
    # The default xmx is 8G
    limits:
      cpu: "4"
      memory: "8G"
```

```
    requests:
      cpu: "1"
      memory: "1G"
  ports:
    embedded: 19200
    rpc: 19998
    web: 19999
  hostPID: true
  hostNetwork: true
  # dnsPolicy will be ClusterFirstWithHostNet if hostNetwork: true
  # and ClusterFirst if hostNetwork: false
  # You can specify dnsPolicy here to override this inference
  # dnsPolicy: ClusterFirst
  # JVM options specific to the master container
  jvmOptions:
  nodeSelector:
    alluxio: 'true'

jobMaster:
  args:
    - job-master
  # Properties for the jobMaster component
  enableLivenessProbe: false
  enableReadinessProbe: false
  properties:
  resources:
    limits:
      cpu: "4"
      memory: "8G"
    requests:
      cpu: "1"
      memory: "1G"
  ports:
    embedded: 20003
    rpc: 20001
    web: 20002
  # JVM options specific to the jobMaster container
  jvmOptions:

# Alluxio supports journal type of UFS and EMBEDDED
# UFS journal with HDFS example
```

```
# journal:
#   type: "UFS"
#   folder: "hdfs://{$hostname}:{$hostport}/journal"
# EMBEDDED journal to /journal example
# journal:
#   type: "EMBEDDED"
#   folder: "/journal"
journal:
 type: "UFS" # "UFS" or "EMBEDDED"
 ufsType: "local" # Ignored if type is "EMBEDDED". "local" or "HDFS"
 folder: "/journal" # Master journal folder
 # volumeType controls the type of journal volume.
 # It can be "persistentVolumeClaim" or "emptyDir"
 volumeType: emptyDir
 size: 1Gi
 # Attributes to use when the journal is persistentVolumeClaim
 storageClass: "standard"
 accessModes:
  - ReadWriteOnce
 # Attributes to use when the journal is emptyDir
 medium: ""
 # Configuration for journal formatting job
 format:
  runFormat: false # Change to true to format journal
  job:
   activeDeadlineSeconds: 30
   ttlSecondsAfterFinished: 10
  resources:
   limits:
    cpu: "4"
    memory: "8G"
   requests:
    cpu: "1"
    memory: "1G"


# You can enable metastore to use ROCKS DB instead of Heap
# metastore:
#   volumeType: persistentVolumeClaim # Options: "persistentVolumeClaim" or "emptyDir"
#   size: 1Gi
#   mountPath: /metastore
```

```
# # Attributes to use when the metastore is persistentVolumeClaim
#   storageClass: "standard"
#   accessModes:
#    - ReadWriteOnce
# # Attributes to use when the metastore is emptyDir
#   medium: ""


## Worker ##

worker:
 args:
  - worker-only
  - --no-format
 enableLivenessProbe: false
 enableReadinessProbe: false
 # Properties for the worker component
 properties:
 resources:
  limits:
   cpu: "4"
   memory: "4G"
  requests:
   cpu: "1"
   memory: "2G"
 ports:
  rpc: 29999
  web: 30000
 hostPID: true
 hostNetwork: true
 # dnsPolicy will be ClusterFirstWithHostNet if hostNetwork: true
 # and ClusterFirst if hostNetwork: false
 # You can specify dnsPolicy here to override this inference
 # dnsPolicy: ClusterFirst
 # JVM options specific to the worker container
 jvmOptions:
 nodeSelector:
  alluxio: 'true'

jobWorker:
```

```
  args:
    - job-worker
  enableLivenessProbe: false
  enableReadinessProbe: false
  # Properties for the jobWorker component
  properties:
  resources:
    limits:
      cpu: "4"
      memory: "4G"
    requests:
      cpu: "1"
      memory: "1G"
  ports:
    rpc: 30001
    data: 30002
    web: 30003
  # JVM options specific to the jobWorker container
  jvmOptions:

# Tiered Storage
# emptyDir example
#   - level: 0
#     alias: MEM
#     mediumtype: MEM
#     path: /dev/shm
#     type: emptyDir
#     quota: 1G
#
# hostPath example
#   - level: 0
#     alias: MEM
#     mediumtype: MEM
#     path: /dev/shm
#     type: hostPath
#     quota: 1G
#
# persistentVolumeClaim example
#   - level: 1
#     alias: SSD
#     mediumtype: SSD
```

```
#    type: persistentVolumeClaim
#    name: alluxio-ssd
#    path: /dev/ssd
#    quota: 10G
#
# multi-part mediumtype example
#  - level: 1
#    alias: SSD,HDD
#    mediumtype: SSD,HDD
#    type: persistentVolumeClaim
#    name: alluxio-ssd,alluxio-hdd
#    path: /dev/ssd,/dev/hdd
#    quota: 10G,10G
tieredstore:
  levels:
    - level: 0
      alias: HDD
      mediumtype: HDD-0
      path: /mnt/disk1
      type: persistentVolumeClaim
      name: lvm-pvc-im
      quota: 12000G
      high: 0.95
      low: 0.7

# Short circuit related properties
shortCircuit:
  enabled: true
  # The policy for short circuit can be "local" or "uuid",
  # local means the cache directory is in the same mount namespace,
  # uuid means interact with domain socket
  policy: uuid
  # volumeType controls the type of shortCircuit volume.
  # It can be "persistentVolumeClaim" or "hostPath"
  volumeType: hostPath
  size: 1Mi
  # Attributes to use if the domain socket volume is PVC
  pvcName: alluxio-worker-domain-socket
  accessModes:
    - ReadWriteOnce
  storageClass: standard
```

```
  storageClass: standard
  # Attributes to use if the domain socket volume is hostPath
  hostPath: "/tmp/alluxio-domain" # The hostPath directory to use


## FUSE ##

fuse:
  image: registry-vpc.cn-beijing.aliyuncs.com/alluxio/alluxio-fuse
  imageTag: 2.3.0
  imagePullPolicy: IfNotPresent
  # Change both to true to deploy FUSE
  enabled: false
  clientEnabled: false
  # Properties for the jobWorker component
  properties:
  # Customize the MaxDirectMemorySize
  # These options are specific to the FUSE daemon
  jvmOptions:
    - "-XX:MaxDirectMemorySize=2g"
  hostNetwork: true
  hostPID: true
  dnsPolicy: ClusterFirstWithHostNet
  user: 0
  group: 0
  fsGroup: 0
  args:
    - fuse
    - --fuse-opts=allow_other
  # Mount path in the host
  mountPath: /mnt/alluxio-fuse
  resources:
    requests:
      cpu: "0.5"
      memory: "1G"
    limits:
      cpu: "4"
      memory: "4G"
  nodeSelector:
    alluxio: 'true'
```

```
##  Secrets ##

# Format: (<name>:<mount path under /secrets/>):
# secrets:
#   master: # Shared by master and jobMaster containers
#     alluxio-hdfs-config: hdfsConfig
#   worker: # Shared by worker and jobWorker containers
#     alluxio-hdfs-config: hdfsConfig
```

注意

- 需使用上述配置的PVC配置tieredstore。

```
tieredstore:
  levels:
    - level: 0
      alias: HDD
      mediumtype: HDD-0
      path: /mnt/disk1
      type: persistentVolumeClaim
      name: lvm-pvc-im
      quota: 12000G
      high: 0.95
      low: 0.7
```

- Alluxio需要和PVC在同一个namspace下。

相关文档

- LVM数据卷