GPU和cuda安装问题

2020年7月30日 13:44

一 输入nvidia-smi，出现：

Failed to initialize NVML: Driver/library version mismatch

解决方法：

1 查看当前nvidia显卡驱动版本我们要的是418.39版本

cat /proc/driver/nvidia/version

2查看系统驱动

cat /var/log/dpkg.log | grep nvidia

3 检查驱动和系统版本是否一致

4如果不一致，删除系统驱动

apt-get purge nvidia

apt-get remove nvidia-*

来自 <https://blog.csdn.net/ChaoFeiLi/article/details/108857701>

5 开始安装nvidia,执行脚本bash /soft/cuda_10.1.105_418.39_linux.run

6如果出现module nvidia XXX in use等字样，执行：

lsmod | grep nvidia & lsof /dev/nvidia*

```
root@slave06:~# lsmod | grep nvidia
nvidia_drm             45056  0
nvidia_modeset       1085440  1 nvidia_drm
nvidia              17592320  16 nvidia_modeset
ipmi_msghandler        53248  4 ipmi_devintf,ipmi_si,nvidia,ipmi_ssif
drm_kms_helper        172032  2 ast,nvidia_drm
drm                   401408  6 drm_kms_helper,ast,nvidia_drm,ttm
root@slave06:~# lsof /dev/nvidia*
COMMAND    PID USER   FD   TYPE  DEVICE SIZE/OFF NODE NAME
gnome-she 3198  gdm    9u   CHR 195,255      0t0  532 /dev/nvidiactl
gnome-she 3198  gdm   11u   CHR   195,0      0t0  533 /dev/nvidia0
gnome-she 3198  gdm   12u   CHR   195,0      0t0  533 /dev/nvidia0
gnome-she 3198  gdm   13u   CHR   195,1      0t0  546 /dev/nvidia1
gnome-she 3198  gdm   14u   CHR   195,1      0t0  546 /dev/nvidia1
root@slave06:~# kill 3198
```

杀如上进程。

7 卸载如上出现的所有nvidia模块

```
root@slave06:~# lsof /dev/nvidia*
root@slave06:~# lsmod | grep nvidia
nvidia_drm             45056  0
nvidia_modeset       1085440  1 nvidia_drm
nvidia              17592320  1 nvidia_modeset
ipmi_msghandler        53248  4 ipmi_devintf,ipmi_si,nvidia,ipmi_ssif
drm_kms_helper        172032  2 ast,nvidia_drm
drm                   401408  5 drm_kms_helper,ast,nvidia_drm,ttm
root@slave06:~#  rmmod nvidia_drm
root@slave06:~# rmmod  nvidia_modeset
root@slave06:~# rmmod  nvidia
root@slave06:~# rmmod  ipmi_msghandler
rmmod: ERROR: Module ipmi_msghandler is in use by: ipmi_devintf ipmi_si ipmi_ssif
root@slave06:~# rmod ipmi_devintf

Command 'rmod' not found, did you mean:

  command 'mod' from deb monodoc-base
  command 'jmod' from deb openjdk-11-jdk-headless
  command 'rmmod' from deb kmod
  command 'qmod' from deb gridengine-client
  command 'rmid' from deb openjdk-11-jre-headless
  command 'rmid' from deb openjdk-8-jre-headless
  command 'kmod' from deb kmod

Try: apt install <deb name>

root@slave06:~# rmmod ipmi_devintf
root@slave06:~# rmmod ipmi_si
root@slave06:~# rmmod nvidia
rmmod: ERROR: Module nvidia is not currently loaded
root@slave06:~# rmmod ipmi_ssif
root@slave06:~# lsmod | grep nvidia
root@slave06:~# cd /soft/
```

无

8 如果上述一直删不掉，则执行：

cd /usr/local/cuda-10.1/bin
cuda-uninstaller
nvidia-uninstall

```
To uninstall the CUDA Toolkit, run cuda-uninstaller in /usr/local/cuda-10.1/bin
To uninstall the NVIDIA Driver, run nvidia-uninstall
```

重启再安装新的驱动

```
root@slave06:~# cd /soft/
root@slave06:/soft# ./cuda_10.1.105_418.39_linux.run
 Completed with errors. See log at /var/log/cuda-installer.log for details.
root@slave06:/soft# vim /var/log/cuda-installer.log
root@slave06:/soft# vim /etc/ld.so.conf.d/
root@slave06:/soft# cd /etc/ld.so.conf.d/
root@slave06:/etc/ld.so.conf.d# ll
total 44
drwxr-xr-x   2 root root  4096 Jul 27 12:52 ./
drwxr-xr-x 150 root root 12288 Aug  3 05:45 ../
```

9还是报错，查看错误是：

```
 39671
 39672 [INFO]: Skipping copy. File already exists at: /usr/local/cuda-10.1/lib64
 39673 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libcudart.so.10.1
 39674 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libcudart.so.10.1.105
 39675 [INFO]: Handle ld conf file
 39676 [ERROR]: cuda.conf wasn't handled correctly during upgrade
 39677 [INFO]: cuda-nvrtc
 39678 [INFO]: /usr/bin/lsb_release
 39679
 39680 [INFO]: Skipping copy. File already exists at: /usr/local/cuda-10.1/lib64
 39681 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libnvrtc-builtins.so
 39682 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libnvrtc-builtins.so.10.1
 39683 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libnvrtc-builtins.so.10.1.105
 39684 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libnvrtc.so.10.1
 39685 [INFO]: Installed: /usr/local/cuda-10.1/targets/x86_64-linux/lib/libnvrtc.so.10.1.105
 39686 [INFO]: cuda-nvjpeg
```

检查/etc/ld.config.so.d/下有一个cuda-10-1.conf，好像和这个版本对应不上，但是和其他完好的机器相比，并无异常。所以执行ldconfig。

10查看日志，发现除了这个错误，其他并无ERROR异常，

于是执行nvidia-smi

```
root@slave06:/soft# nvidia-smi
Mon Aug  3 06:18:58 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.39       Driver Version: 418.39       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 208...  On   | 00000000:3D:00.0 Off |                  N/A |
| 29%   29C    P8     1W / 250W |      0MiB / 10989MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce RTX 208...  On   | 00000000:41:00.0 Off |                  N/A |
| 29%   29C    P8     4W / 250W |      0MiB / 10989MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

二 查看GPU型号和架构

nvidia-smi -L

```
root@slave06:/usr/local/cuda/samples# nvidia-smi -L
GPU 0: GeForce RTX 2080 Ti (UUID: GPU-20fbec52-8c31-118f-52a7-8678af956128)
GPU 1: GeForce RTX 2080 Ti (UUID: GPU-014deb77-a5fa-7ea2-eda6-a170cec4f64f)
root@slave06:/usr/local/cuda/samples#
```

三 Hyper Q 让流的kernel也并行

hyperq --nstreams=8

四遇到这种问题：

NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver. Make sure that the latest NVIDIA

发现有驱动，就是起不起来。这样咋办呢，两步：

sudo apt-get install dkms

dkms install -m nvidia -v 418.39

```
    https://microk8s.io/high-availability

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

59 packages can be updated.
2 updates are security updates.

New release '20.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Thu Nov  5 01:44:44 2020 from 172.19.0.179
NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver. Make sure that the latest NVIDIA
driver is installed and running.

mount.nfs: access denied by server while mounting master00:/opt
mount.nfs: access denied by server while mounting master00:/home
root@slave06:~# ls /usr/src | grep nvidia
nvidia-418.39
root@slave06:~# nvcc -v
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
root@slave06:~# apt-get install dkms
Reading package lists... Done
Building dependency tree
Reading state information... Done
dkms is already the newest version (2.3-3ubuntu9.7).
The following packages were automatically installed and are no longer required:
  libatk-wrapper-java libatk-wrapper-java-jni libgif7 libxt-dev
Use 'apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 76 not upgraded.
root@slave06:~# dkms install -m nvidia -v 418.39

Creating symlink /var/lib/dkms/nvidia/418.39/source ->
                 /usr/src/nvidia-418.39

DKMS: add completed.

Kernel preparation unnecessary for this kernel.  Skipping...

Building module:
cleaning build area...
'make' -j32 NV_EXCLUDE_BUILD_MODULES='' KERNEL_UNAME=4.15.0-122-generic IGNORE_CC_MISMATCH='1' modules......
Signing module:
 - /var/lib/dkms/nvidia/418.39/4.15.0-122-generic/x86_64/module/nvidia-modeset.ko
 - /var/lib/dkms/nvidia/418.39/4.15.0-122-generic/x86_64/module/nvidia-uvm.ko
 - /var/lib/dkms/nvidia/418.39/4.15.0-122-generic/x86_64/module/nvidia.ko
```

安装cuda 10.0，选项如下：

```
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 410.48?
(y)es/(n)o/(q)uit: y

Do you want to install the OpenGL libraries?
(y)es/(n)o/(q)uit [ default is yes ]: n

Do you want to run nvidia-xconfig?
This will update the system X configuration file so that the NVIDIA X driver
is used. The pre-existing X configuration file will be backed up.
This option should not be used on systems that require a custom
X configuration, such as systems with multiple GPU vendors.
(y)es/(n)o/(q)uit [ default is no ]:

Install the CUDA 10.0 Toolkit?
(y)es/(n)o/(q)uit: y

Enter Toolkit Location
 [ default is /usr/local/cuda-10.0 ]:

Do you want to install a symbolic link at /usr/local/cuda?
(y)es/(n)o/(q)uit: y

Install the CUDA 10.0 Samples?
(y)es/(n)o/(q)uit: y

Enter CUDA Samples Location
 [ default is /root ]:

Installing the NVIDIA display driver...
```

安装cudnn ,文件在/usr/local下

tar zxvf cudnn-10.0-linux-x64-v7.6.4.38.tgz

sudo cp cuda/include/cudnn.h /usr/local/cuda/include/

sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64/

sudo chmod a+r /usr/local/cuda/include/cudnn.h

sudo chmod a+r /usr/local/cuda/lib64/libcudnn*

查看cudnn版本

cat /usr/local/cuda/include/cudnn.h | grep CUDNN_MAJOR -A 2

GPU相关知识

SIMT 相对于SIMD(Single Instruction, Multiple Data),前者主要采用线程并行的方式，后者主要采用数据并行的方式

即：线程并行，处理不同的数据。

SM：流处理器

SP：执行单元

Docker 常用命令

2020年8月3日 19:31

```
root@slave06:~# docker exec -it aa /bin/bash
Error response from daemon: Container 5c08fbad98add018b4e3a4600e153c607b584f051fd6278443d083d9d7932f5f is not running
root@slave06:~# docker start 534917b1443a
Error response from daemon: could not select device driver "" with capabilities: [[gpu]]
Error: failed to start containers: 534917b1443a
root@slave06:~# docker ps
```

docker启动:systemctl start docker

查看镜像状态:

docker ps -a

启动容器 docker start [CONTAINER ID]

```
ontainerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root     13915  0.0  0.0  13136  1104 pts/2    S+   12:09   0:00 grep --color=auto 534917b1443a
root@slave06:/soft/nvidia-docker-master# kill  -9 12786
-bash: kill: (12786) - No such process
root@slave06:/soft/nvidia-docker-master# docker start 534917b1443a
534917b1443a
root@slave06:/soft/nvidia-docker-master# docker stop 534917b1443a
534917b1443a
root@slave06:/soft/nvidia-docker-master# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
7968bde7bce8        hello-world         "/hello"                 18 minutes ago      Exited (0) 18 minutes ago                       vigilant_curie
534917b1443a        appl/project2:v1.0  "/bin/sh -c 'while t…"   3 days ago          Exited (137) 4 minutes ago                      bb
5c08fbad98ad        appl/project2:v1.0  "/bin/sh -c 'while t…"   6 days ago          Exited (137) 9 hours ago                        aa
5de58fb6d454        hello-world         "/hello"                 6 days ago          Exited (0) 6 days ago                           competent_stonebraker
root@slave06:/soft/nvidia-docker-master# docker start 534917b1443a
534917b1443a
root@slave06:/soft/nvidia-docker-master# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
7968bde7bce8        hello-world         "/hello"                 18 minutes ago      Exited (0) 18 minutes ago                       vigilant_curie
534917b1443a        appl/project2:v1.0  "/bin/sh -c 'while t…"   3 days ago          Up 11 seconds                                   bb
5c08fbad98ad        appl/project2:v1.0  "/bin/sh -c 'while t…"   6 days ago          Exited (137) 9 hours ago                        aa
5de58fb6d454        hello-world         "/hello"                 6 days ago          Exited (0) 6 days ago                           competent_stonebraker
root@slave06:/soft/nvidia-docker-master#
```

启动容器内一个镜像：docker exec -it [容器名称] /bin/bash

```
root@slave06:/soft/nvidia-docker-master# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
7968bde7bce8        hello-world         "/hello"                 21 minutes ago      Exited (0) 21 minutes ago                       vigilant_curie
534917b1443a        appl/project2:v1.0  "/bin/sh -c 'while t…"   3 days ago          Up 3 minutes                                    bb
5c08fbad98ad        appl/project2:v1.0  "/bin/sh -c 'while t…"   6 days ago          Exited (137) 9 hours ago                        aa
5de58fb6d454        hello-world         "/hello"                 6 days ago          Exited (0) 6 days ago                           competent_stonebraker
root@slave06:/soft/nvidia-docker-master# docker exec -it 534917b1443a /bin/bash
root@534917b1443a:/#
root@534917b1443a:/# ll
total 88
drwxr-xr-x   1 root root 4096 Jul 31 03:49 ./
drwxr-xr-x   1 root root 4096 Jul 31 03:49 ../
-rwxr-xr-x   1 root root    0 Jul 31 03:49 .dockerenv*
drwxr-xr-x   1 root root 4096 Jul 31 04:04 bin/
drwxr-xr-x   2 root root 4096 Apr 24  2018 boot/
drwxr-xr-x   5 root root  440 Aug  3 12:15 dev/
drwxr-xr-x   1 root root 4096 Aug  3 12:15 etc/
drwxr-xr-x   2 root root 4096 Apr 24  2018 home/
drwxr-xr-x   1 root root 4096 Apr  3  2019 lib/
drwxr-xr-x   2 root root 4096 Mar  7  2019 lib64/
drwxr-xr-x   2 root root 4096 Mar  7  2019 media/
drwxr-xr-x   2 root root 4096 Mar  7  2019 mnt/
drwxr-xr-x   2 root root 4096 Mar  7  2019 opt/
dr-xr-xr-x 477 root root    0 Aug  3 12:15 proc/
drwx------   1 root root 4096 Aug  3 00:08 root/
drwxr-xr-x   1 root root 4096 Mar 12  2019 run/
drwxr-xr-x   1 root root 4096 Jun 24 08:35 sbin/
drwxr-xr-x   2 root root 4096 Mar  7  2019 srv/
dr-xr-xr-x  13 root root    0 Aug  3 12:15 sys/
drwxrwxrwt   1 root root 4096 Jul 31 09:17 tmp/
drwxr-xr-x   1 root root 4096 Mar  7  2019 usr/
drwxr-xr-x   1 root root 4096 Mar  7  2019 var/
root@534917b1443a:/#
```

关闭docker systemctl stop docker

```
root@slave06:~# docker start 534917b1443a
Error response from daemon: could not select device driver "" with capabilities: [[gpu]]
Error: failed to start containers: 534917b1443a
root@slave06:~# systemctl start docker
root@slave06:~# dockers version

Command 'dockers' not found, did you mean:

  command 'docker' from snap docker (19.03.11)
  command 'dockerd' from deb docker.io
  command 'docker' from deb docker.io
```

杀容器，这个在启动出错时候使用非常管用：

```
drwxr-xr-x  4 root root  4096 Jul  8 20:35 rpm/
root@slave06:/soft/nvidia-docker-master# ps -aux|grep 534917b1443a
root     13786  0.0  0.0 107700  5700 ?        Sl   12:07   0:00 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/534917b14
ontainerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root     13915  0.0  0.0  13136  1104 pts/2    S+   12:09   0:00 grep --color=auto 534917b1443a
root@slave06:/soft/nvidia-docker-master# kill  -9 12786
-bash: kill: (12786) - No such process
root@slave06:/soft/nvidia-docker-master# docker start 534917b1443a
534917b1443a
```

将docker中的内容复制出来

Docker cp bb:/root/app1 ./app1

将内容复制到docker中

docker cp libjdetection_cpu.so bb:/usr/lib/

NFS磁盘阵列操作步骤

2020年8月3日 20:53

一master(Ubuntu18.04)连接磁盘阵列，使用iscsi模式

(1)master上安装open-iscsi

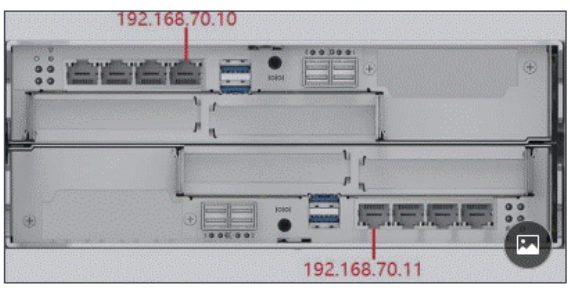
https://packages.debian.org/buster/open-iscsi

(2)查看系统的qin号，这个号码每次重装系统都会变化

```
root@master00:~# cat /etc/iscsi/initiatorname.iscsi
## DO NOT EDIT OR REMOVE THIS FILE!
## If you remove this file, the iSCSI daemon will not start.
## If you change the InitiatorName, existing access control lists
## may reject this initiator.  The InitiatorName must be unique
## for each iSCSI initiator.  Do NOT duplicate iSCSI InitiatorNames.
InitiatorName=iqn.1993-08.org.debian:01:219b5d4f819a
root@master00:~#
```

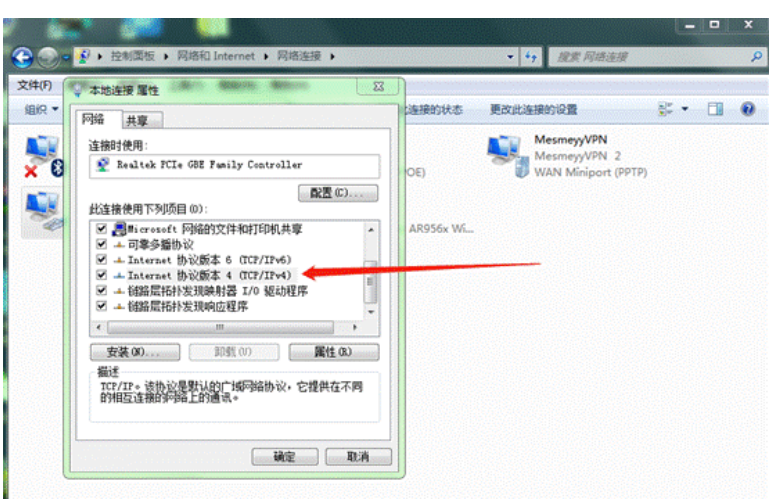
(1) 把qin号传进磁盘阵列存储器

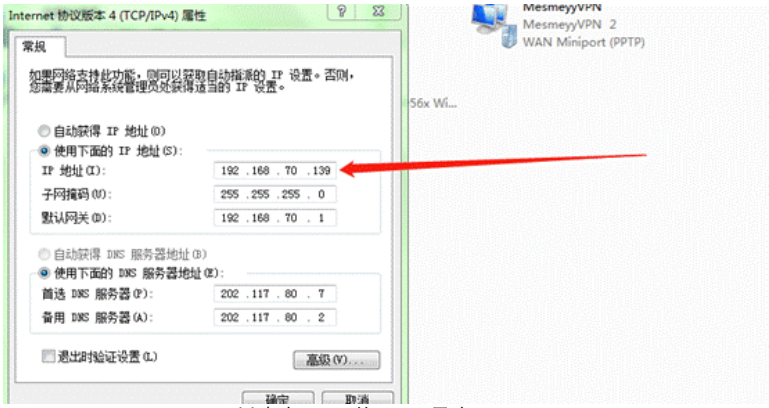
1.找根网线连接你的笔记本和磁盘阵列。连接口如下:



控制器和端口	IPv4 地址	IPv4 子网掩码
控制器 1（上）端口 1（最右）	192.168.70.10	255.255.255.0
控制器 2（下）端口 1（最左）	192.168.70.11	255.255.255.0

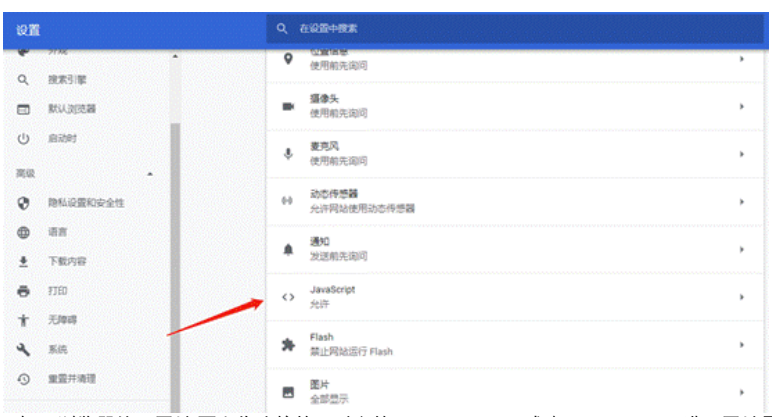
2.改自己笔记本的本地连接IP,使得笔记本和磁盘阵列在同一网段。





192.168.70.139里面，139可以改动。下面的DNS不用动。

3.开始连接，在浏览器端要允许运行jsp程序，谷歌浏览器为例：



4.打开浏览器输入网址,要和你连接的口对应的，192.168.70.10或者192.168.20.11,进入网址界面找到主机项目，添加主机，端口号就是iqn号码。添加完成后，实际上没有激活iqn号，仅仅是添加而已，主机状态是“已降级”，因此还要点击添加主机映射。

5.在master机器端，需要安装 iscsi-initiator 的包，这个包包含iscsiadm命令，之前的open-iscsi已经包含这个命令了。不用再管 iscsi-initiator 了。
发现存储设备 的命令如下，这个命令成功，代表磁盘阵列端已经好了。
root@master00:/etc/iscsi# iscsiadm -m discovery -t st -p 172.19.0.208 172.19.0.208:3260,1 iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1
172.19.0.208:3260,1 iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1

(2) 开始挂载master到磁盘阵列(以下展示Ubuntu18.04下的命令，红帽系主机和Debian系主机命令有差别)。
红帽系列:https://blog.csdn.net/zhongbeida_xue/article/details/70921167
Debian系列：
root@master00:/etc/iscsi# sudo iscsiadm --mode node --targetname iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1 --portal 172.19.0.208:3260 --logout(非必须)

Logging out of session [sid: 1, target: iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1, portal: 172.19.0.208,3260]
Logout of [sid: 1, target: iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1, portal: 172.19.0.208,3260] successful.

root@master00:/etc/iscsi# sudo iscsiadm --mode node --targetname iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1 --portal 172.19.0.208:3260 --login

Logging in to [iface: default, target: iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1, portal: 172.19.0.208,3260] (multiple)
Login to [iface: default, target: iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1, portal: 172.19.0.208,3260] successful.
root@master00:/etc/iscsi# sudo iscsiadm -m session
tcp: [2] 172.19.0.208:3260,4 iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1 (non-flash)

(4) 发现磁盘阵列并挂载

```
root@master00:~# fdisk -l
Disk /dev/loop0: 88.5 MiB, 92778496 bytes, 181208 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/sda: 3.7 TiB, 4000225165312 bytes, 7812939776 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 262144 bytes / 262144 bytes
Disklabel type: gpt
Disk identifier: 62357171-9FE0-4AF3-8B91-32CFA0C5E555

Device       Start        End    Sectors  Size Type
/dev/sda1     2048    1050623    1048576  512M EFI System
/dev/sda2  1050624    2074623    1024000  500M Linux filesystem
/dev/sda3  2074624   69183487   67108864   32G Linux swap
/dev/sda4 69183488 7812937727 7743754240  3.6T Linux filesystem


Disk /dev/sdb: 32.5 TiB, 35723132786688 bytes, 69771743724 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 32768 bytes / 32768 bytes
root@master00:~#
```

挂载：mount /dev/sdb /data
如果之前/data已经有挂载，执行umount -l /data，不要执行rm 。

二 在master上搭建nfs服务，slave使用nfs服务

(1) Master端的工作，
1.准备nfs和rpcbind,nfs依赖rpcbind:
下载安装nfs-kernel-server,其中也包含依赖包的地址:
https://packages.debian.org/buster/nfs-kernel-server
下载安装nfs-common,其中也包含依赖包的地址:
https://packages.debian.org/buster/nfs-common
2.检查rpcbind服务是否开启，一般会自动开机启动的。执行rpcinfo，如果出现以下样子，代表rpcbind启动成功。

```
root@slave06:/etc# rpcinfo
   program version netid     address                service    owner
    100000    4    tcp6      ::.0.111               portmapper superuser
    100000    3    tcp6      ::.0.111               portmapper superuser
    100000    4    udp6      ::.0.111               portmapper superuser
    100000    3    udp6      ::.0.111               portmapper superuser
    100000    4    tcp       0.0.0.0.0.111          portmapper superuser
    100000    3    tcp       0.0.0.0.0.111          portmapper superuser
    100000    2    tcp       0.0.0.0.0.111          portmapper superuser
    100000    4    udp       0.0.0.0.0.111          portmapper superuser
    100000    3    udp       0.0.0.0.0.111          portmapper superuser
    100000    2    udp       0.0.0.0.0.111          portmapper superuser
    100000    4    local     /run/rpcbind.sock      portmapper superuser
    100000    3    local     /run/rpcbind.sock      portmapper superuser
    100024    1    udp       0.0.0.0.221.124        status     112
    100024    1    tcp       0.0.0.0.202.109        status     112
    100024    1    udp6      ::.155.99              status     112
    100024    1    tcp6      ::.150.23              status     112
root@slave06:/etc#
```

3.修改/etc/exports文件：

```
root@master00:/# cd /etc/
root@master00:/etc# cat /etc/exports
/data *(rw,insecure,no_root_squash,sync)
root@master00:/etc#
```

4.root下执行:service nfs-kernel-server start

5.查看目录是否被共享：showmount -e

```
hrm@master00:~$ showmount -e
Export list for master00:
/data *
hrm@master00:~$
```

(2) slave端的工作

1.安装nfs-common及依赖:

https://packages.debian.org/buster/nfs-common

2.执行rpcinfo命令检查rpcbind运行情况(servise rpcbind start)

3.mount -t nfs 172.19.0.200:/data /data

4.设置开机启动

```
ln -fs /lib/systemd/system/rc-local.service /etc/systemd/system/rc-local.service
cd /etc/systemd/system/
vim rc-local.service
下面如果没有【】install】，在最下面添加:
[Install]
WantedBy=multi-user.target
Alias=rc-local.service

cd /etc
touch rc.local
vim rc.local
mount -t nfd 172.19.0.200:/data /data
chmod 755 rc.local
完成!
```

平常用，简要版:

Server:

发现命令:

iscsiadm -m discovery -t st -p 172.19.0.208

登录命令:

iscsiadm -m node -T iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1 -p 172.19.0.208:3260 -l

(注释：iqn.2004-12.com.inspur:mcs.cluster192.168.0.100.node1是上一个命令执行的后半部分结果，ip和端口不变)

挂载命令:

Mount /dev/sdb /data

vim /etc/exports

service nfs-kernel-server start

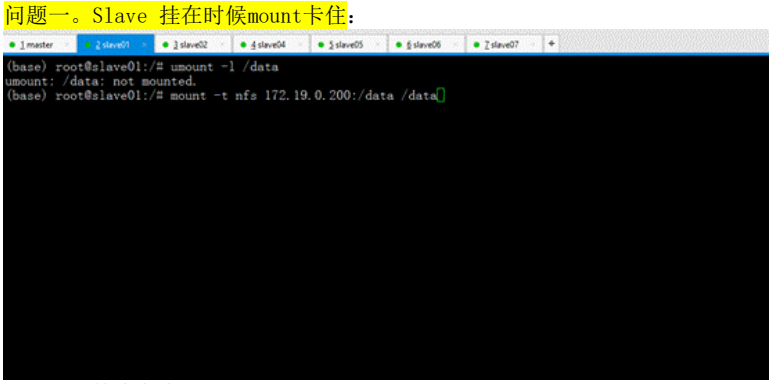
Showmount -e:查看分享目录

Showmount -a:查看各个slave的挂载情况

Client:

查看端口rpcinfo:即使列表没有出现112service服务也不影响。
挂载目录 mount -t nfs 172.19.0.200:/data /data

问题一。Slave 挂在时候mount卡住：



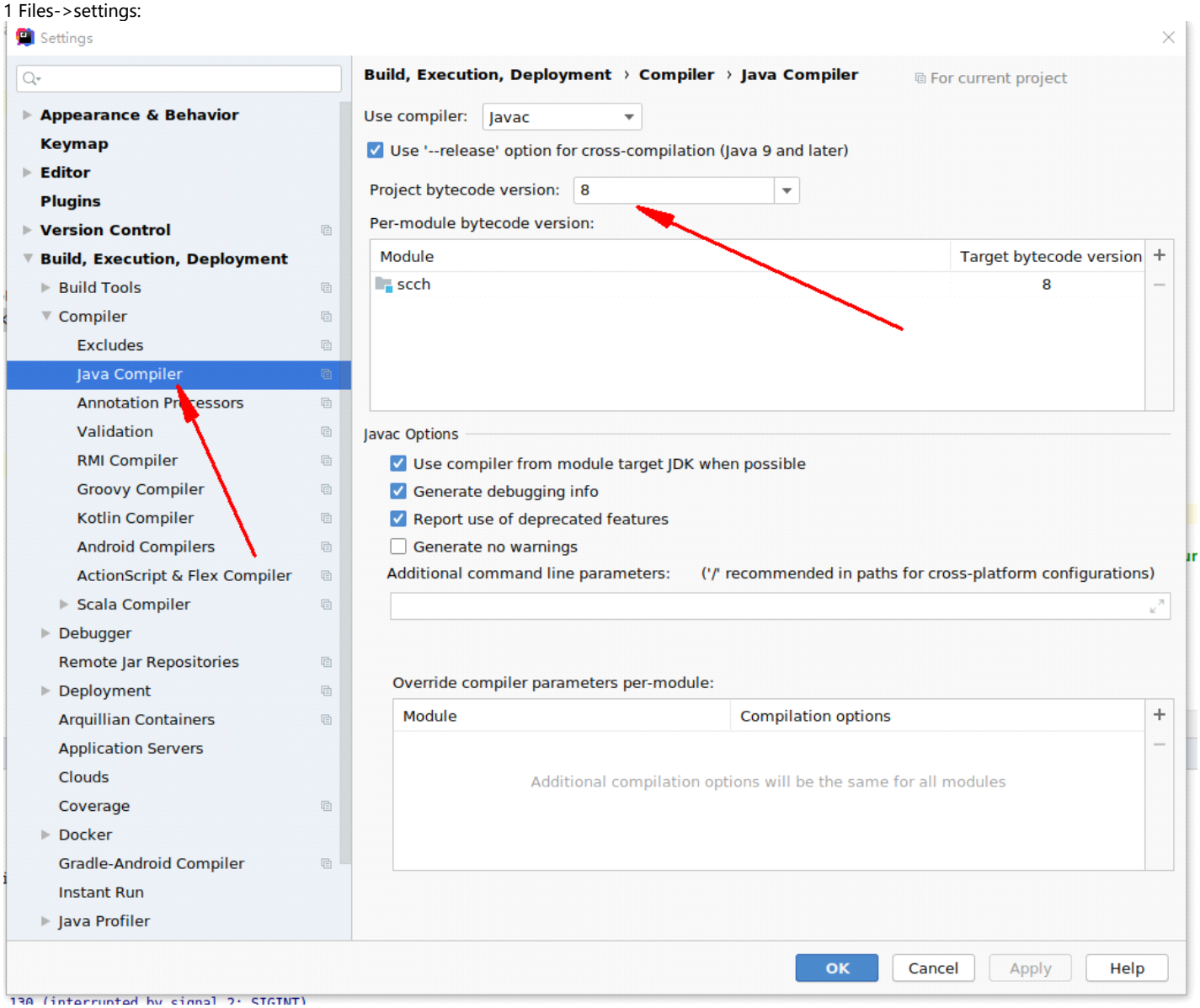
这时候先检查各个slave ：
umount -l /data
再关掉master的nfs ：
service nfs-kernel-server stop
重启nfs服务：
service nfs-kernel-server start
然后挂在slave:
mount -t nfs 172.19.0.200:/data /data

在需要对系统重启的时候，先重启slave后重启master.否则就会因为master的NFS服务先挂而造成slave mount卡主。

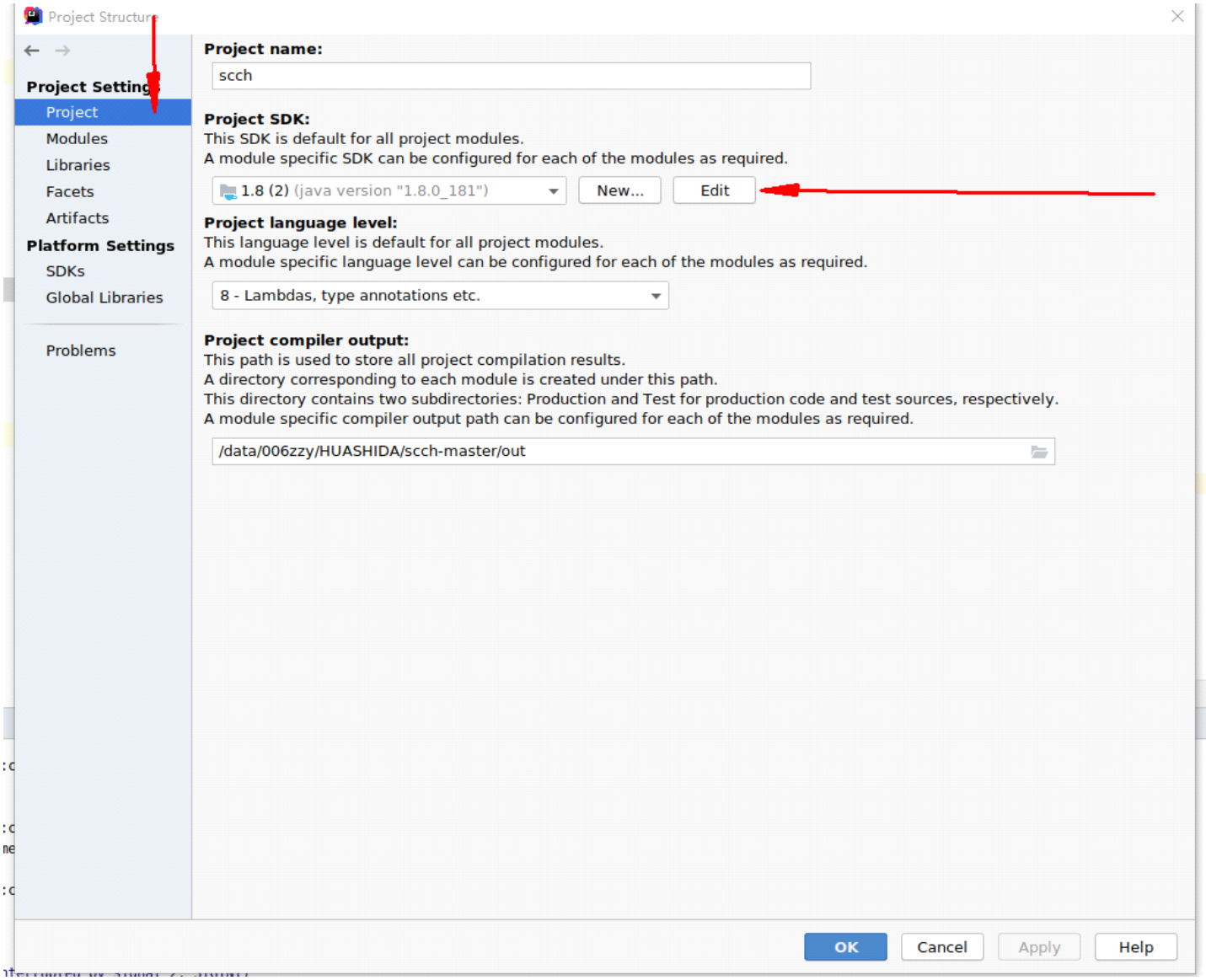
IDEA更改jdk版本的方法

2020年8月9日 11:48

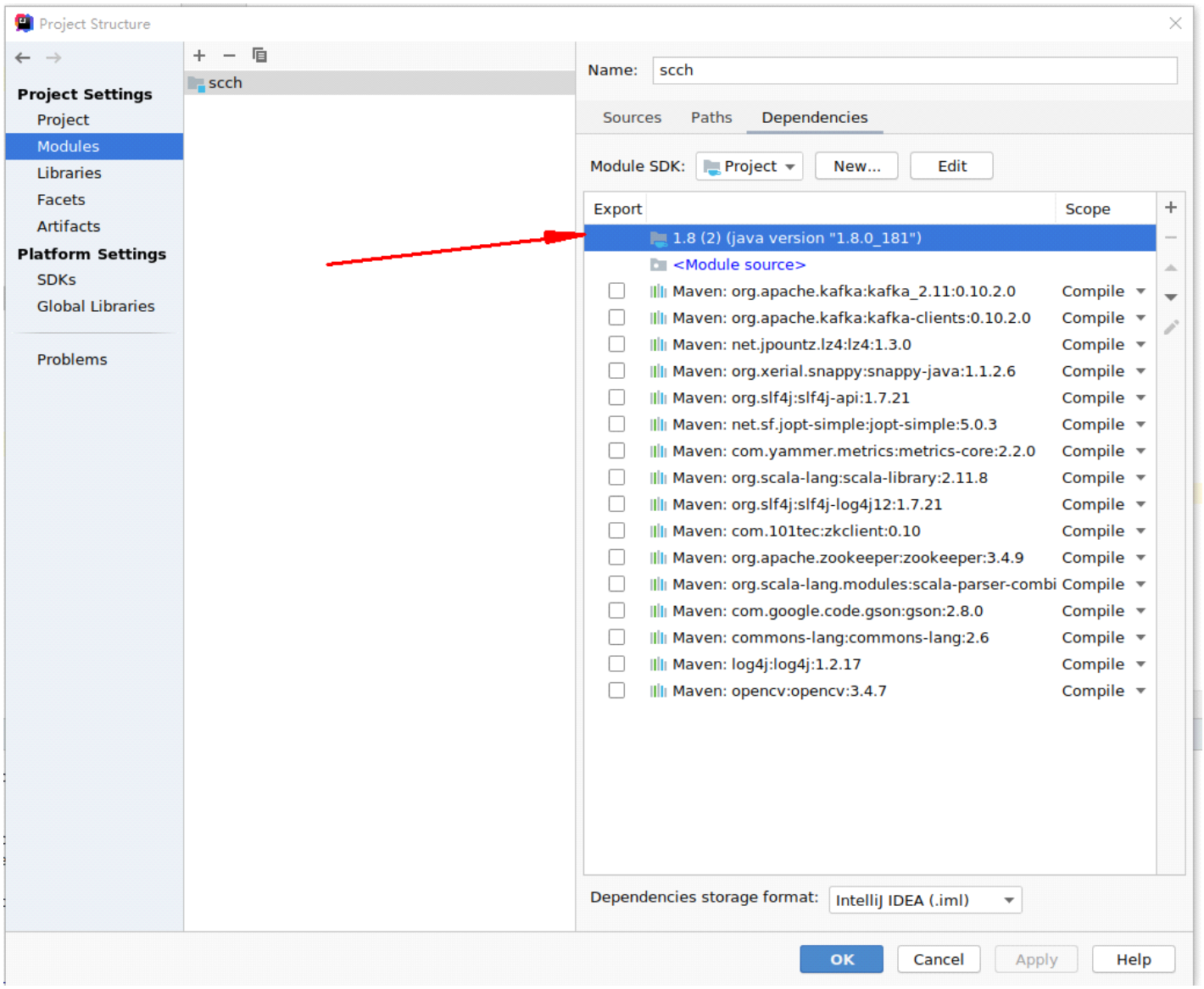
1 Files->settings:



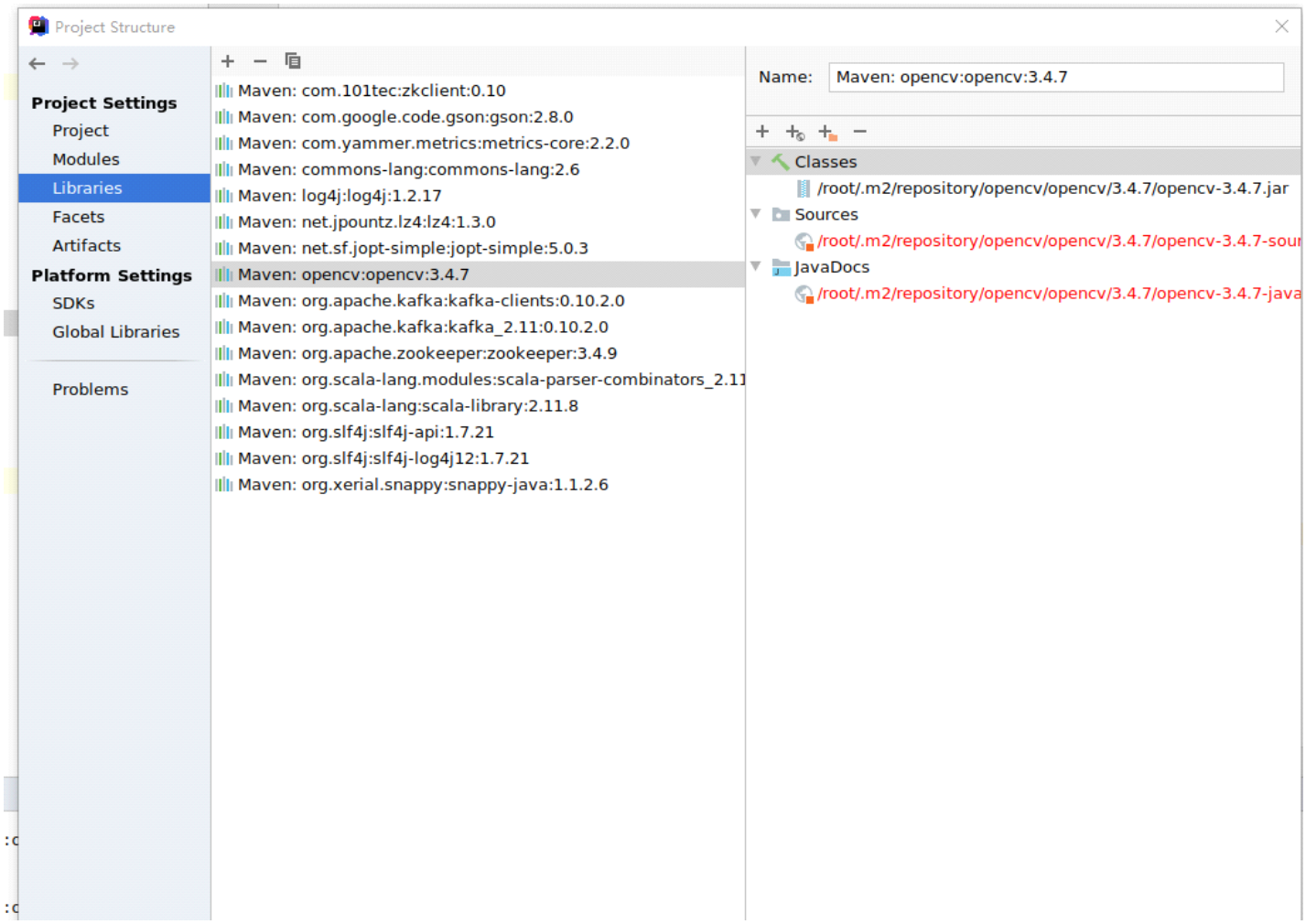
2 Files->project structure->project:



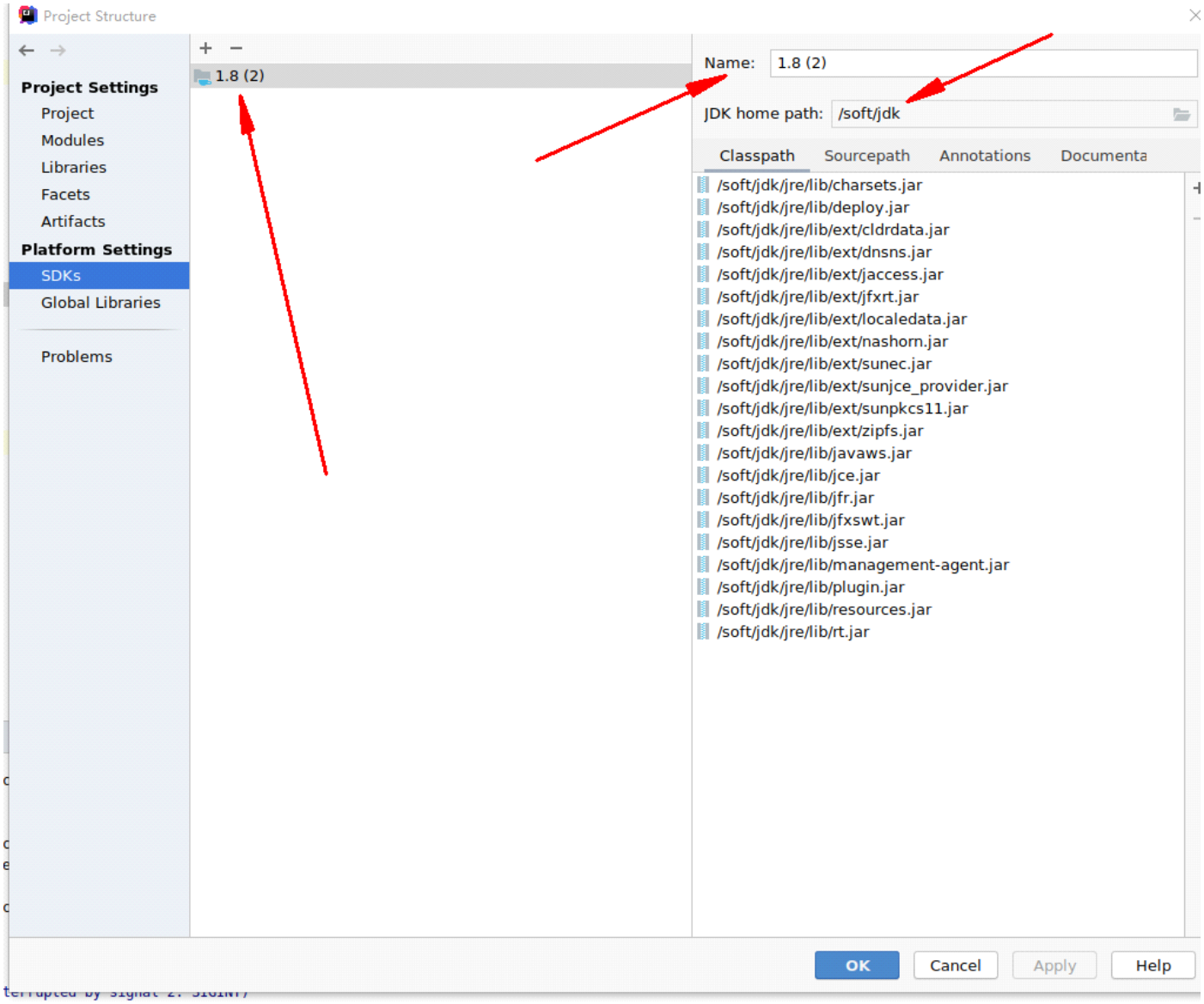
3 Files->project structure->modules:箭头处一定要对得上。



4 Files->project structure->libraries:


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Maven: com.101tec:zkclient:0.10
Maven: com.google.code.gson:gson:2.8.0
Maven: com.yammer.metrics:metrics-core:2.2.0
Maven: commons-lang:commons-lang:2.6
Maven: log4j:log4j:1.2.17
Maven: net.jpountz.lz4:lz4:1.3.0
Maven: net.sf.jopt-simple:jopt-simple:5.0.3
Maven: opencv:opencv:3.4.7
Maven: org.apache.kafka:kafka-clients:0.10.2.0
Maven: org.apache.kafka:kafka_2.11:0.10.2.0
Maven: org.apache.zookeeper:zookeeper:3.4.9
Maven: org.scala-lang.modules:scala-parser-combinators_2.11
Maven: org.scala-lang:scala-library:2.11.8
Maven: org.slf4j:slf4j-api:1.7.21
Maven: org.slf4j:slf4j-log4j12:1.7.21
Maven: org.xerial.snappy:snappy-java:1.1.2.6
Name:
Maven: opencv:opencv:3.4.7
Classes
/root/.m2/repository/opencv/opencv/3.4.7/opencv-3.4.7.jar
Sources
/root/.m2/repository/opencv/opencv/3.4.7/opencv-3.4.7-sour
JavaDocs
/root/.m2/repository/opencv/opencv/3.4.7/opencv-3.4.7-java

5 检查SDK


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
1.8 (2)
Name: 1.8 (2)
JDK home path: /soft/jdk
Classpath
Sourcepath
Annotations
Documenta
/soft/jdk/jre/lib/charsets.jar
/soft/jdk/jre/lib/deploy.jar
/soft/jdk/jre/lib/ext/cldrdata.jar
/soft/jdk/jre/lib/ext/dnsns.jar
/soft/jdk/jre/lib/ext/jaccess.jar
/soft/jdk/jre/lib/ext/jfxrt.jar
/soft/jdk/jre/lib/ext/localedata.jar
/soft/jdk/jre/lib/ext/nashorn.jar
/soft/jdk/jre/lib/ext/sunec.jar
/soft/jdk/jre/lib/ext/sunjce_provider.jar
/soft/jdk/jre/lib/ext/sunpkcs11.jar
/soft/jdk/jre/lib/ext/zipfs.jar
/soft/jdk/jre/lib/javaws.jar
/soft/jdk/jre/lib/jce.jar
/soft/jdk/jre/lib/jfr.jar
/soft/jdk/jre/lib/jfxswt.jar
/soft/jdk/jre/lib/jsse.jar
/soft/jdk/jre/lib/management-agent.jar
/soft/jdk/jre/lib/plugin.jar
/soft/jdk/jre/lib/resources.jar
/soft/jdk/jre/lib/rt.jar
OK
Cancel
Apply
Help

Detection Java被调用的过程

2020年8月13日 14:02

1 root/detection/1018CPUGPUBSV6/test_java_detector_v3.1/runtime/detection/Detector.java

```
        return computeBoxesAndAccByInputBytes(strutWrapperPeer, bytes, outputFile, thresh, hier_thresh, fullscreen, w, h, c);
    }

    public byte[] jpgfile2bytes(String path, int w, int h, int c) {

        return jpg2Bytes(path, w, h, c);
    }


    Detector(String cfgFile, String weightFile, String dataConfig, String labelPath, int batchSize, int gpuIndex, String gpuId) {
        System.out.println("构造方法");
        peer = initialize(cfgFile, weightFile, dataConfig, labelPath, batchSize, gpuIndex, gpuId);
        System.out.println("构造方法结束");
    }

    private native byte[] jpg2Bytes(String path, int w, int h, int c);

    private native BoxesAndAcc[] computeBoxesAndAccByInputBytes(long strutWrapperPeer, byte[] bytes, String outfile, float thresh, float hierThresh, int fullscreen, int w, i

    private native long initialize(String cfgfile, String weightfile, String datacfg, String labelpath, int batchsize, int gpuIndex, String gpuid);


}

class Box {
    float x;
```

2 生成runtime_detection_Detector.h文件

```
/*
 * Class:     runtime_detection_Detector
 * Method:    initialize
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;IILjava/lang/String;)J
 */
JNIEXPORT jlong JNICALL Java_runtime_detection_Detector_initialize
  (JNIEnv *, jobject, jstring, jstring, jstring, jstring, jint, jint, jstring);
```

3 编写runtime_detection_Detector.cpp文件

```
//JNIEXPORT jlong JNICALL Java_src_main_java_operator_Detector_initialize// 3-1: initialize(), return (jlong)peer.
JNIEXPORT jlong JNICALL Java_runtime_detection_Detector_initialize// 3-1: initialize(), return (jlong)peer.
  (JNIEnv *env , jobject obg , jstring cfgfile, jstring weightfile, jstring datacfg, jstring labelpath, jint batchsize , jint gpu_index , jstring gpuid){

        printf("********************************************** This is the 1st GetStringUTFChars...\n");

        //printf("before ********************************************** cfgfile: %s \n", cfgfile);
        char* _cfgfile = (char*)(env)->GetStringUTFChars(  cfgfile, 0 );
        //printf("after  ********************************************** cfgfile: %s \n", _cfgfile);

        //printf("********************************************** This is the 2nd GetStringUTFChars...\n");
        char* _weightfile = (char*)(env)->GetStringUTFChars( weightfile, 0 );

           printf("Java_Detector_initialize in cpp.\n");

        char* gpuids =jstring2char(env, gpuid);


        long startLoad = getCurrentTime();
        network* peer = load_network_test(_cfgfile, _weightfile , batchsize , gpu_index , gpuids);
        long endLoad = getCurrentTime();
       (env)->ReleaseStringUTFChars(  cfgfile, _cfgfile );
```

4这个文件链接的动态库内容在run.sh中，使用的是detectionc的动态库

```
#javac Detector.java
#javah Detector
#export LD_LIBRARY_PATH=/home/kfch/Downloads/0924/0925/test_java_detector_v3.1
export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
#g++ -I./ -I=/opt/jvm/jdk1.8.0_201/include -I=opt/jvm/jdk1.8.0_201/include/linux -L=/opt/jvm/jdk1.8.0_201/lib -fPIC -shared -o libjdetection.so Detector.cpp -L/home/kfch/Down
Detection-v3/test_java_detector_v3.1 -L/usr/local/cuda-10.0/lib64  -ldetection
#g++ -I./ -I=/opt/jdk1.8.0_201/include -L=/opt/jdk1.8.0_201/lib -fPIC -shared -o libjdetection.so operator_detection_yolo__Detector.cpp -L/home/chenkk/Files/Others/Detection-
tector_v3.1 -L/usr/local/cuda/lib64  -ldetection
#/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
g++ -I./ -I/usr/lib/jvm/java-8-openjdk-amd64/include -I/usr/lib/jvm/java-8-openjdk-amd64/include/linux -L/usr/lib/jvm/java-8-openjdk-amd64/lib -fPIC -shared -o libjdetection.s
ction_Detector.cpp -L/home/spark/tools/project2_dynamiclinklib/detection/1018CPUGPUBSV6/test_java_detector_v3.1 -L/usr/local/cuda-10.1/lib64 -ldetection
#javac Detector.java
#java Detector ./cfg/yolov3.cfg ./yolov3.weights 768 576 3 /home/kfch/Downloads/0404Java/Detection-v3/test_java_detector_v3.1/dog_.txt
cp ./libjdetection.so /usr/lib
~
~
~
~
~
~
```

5 在/root/detection/1018CPUGPUBSV6/detection-master-v6/examples/detector.c中实现

```
network *load_network_test(char *cfgfile, char *weightfile, int batchsize, int gpu_index_params, char *gpuid)
{

    //printf("from java batchsize: %d\n", batchsize);
    //printf("from java gpuids: %s\n", gpuid);

    gpu_index = gpu_index_params;

    long load_networkStart = getCurrentTime();
    network *net = load_network(cfgfile, weightfile, 0);
    long load_networkEnd = getCurrentTime();
    printf("load time = %ld \n", load_networkEnd - load_networkStart);
    set_batch_network(net, 1);
    //
    //set_batch_network(net, batchsize);
    //printf("actual net batchsize: %d\n", net->batch);

    return net;
}
```

6，6及以后是加载和检测过程

图解TcpIp

2020年8月19日 15:45

IP地址有层次性，mac地址无层次性：层次性指的是在地址寻找过程中，能否根据地址先分成大类，再分成小类，然后找到目标主机。

分层设备们：
中继器:物理层，延长网络信号，延长电信号或者光信号。中继器链接的通信媒介必须速度相同。
集线器：多端口的中继器。
网桥：将数据帧转发给相邻的其他网络，不限制连接网段个数。
路由器：分担网络负荷，连接不同链路，从这一层开始处理IP。
4-7层路由器：负载均衡服务器。

mesos 配置gpu

2020年8月29日 11:39

```
./mesos-agent --master=192.19.0.205:5050 --work_dir==/soft/mesos-1.6.1/mesos_install/work --containerizers=mesos --isolation="filesystem/linux,cgroups/devices,gpu/nvidia" --nvidia_gpu_devices="0" --resources="gpus:1"
```

CUDA

2020年9月14日 11:09

传说中的CHECK

```
#include <cuda_runtime.h>
#include <stdio.h>

#define CHECK(call)
{
   const cudaError_t error = call;
   if (error != cudaSuccess)
   {
      printf("Error: %s:%d, ", __FILE__, __LINE__);
      printf("code:%d, reason: %s\n", error, cudaGetErrorString(error));
      exit(-10*error);
   }                                                                    \
}

void initialInt(int *ip, int size) {
   for (int i=0; i<size; i++) {
      ip[i] = i;
   }
}

void printMatrix(int *C, const int nx, const int ny) {
   int *ic = C;
   printf("\nMatrix: (%d.%d)\n",nx,ny);
   for (int iy=0; iy<ny; iy++) {
```

矩阵配置标准模板

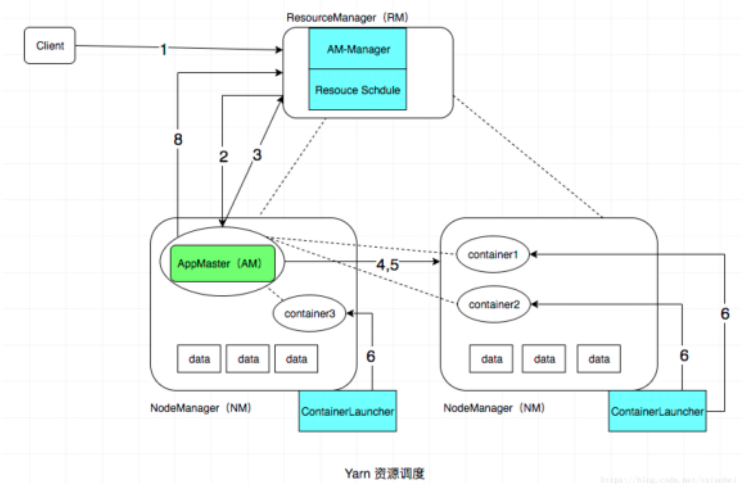
```
int nx = 1<<14;
int ny = 1<<14;
```

```
int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);
```

Yarn的工作过程笔记

2020年11月26日 9:51

Yarn的结构包含ResourceManager和NodeManager两部分：

- ResourceManager：顾名思义资源管理器，主要负责资源管理和调度，ResourceManager主要由两个组件构成：**ApplicationMasterManager**,主要负责两类工作：1.管理监控各个系统的应用，包括启动Applicaiton Master，监控Applicaiton Master运行状态（为了容错）2.跟踪分配给Application Master的进度和状态。**Scheduler**，主要负责分配Container给Applicaiton Master，分配算法有多种（如公平调度等等）可以根据需求不同选择适合的调度策略。
- NodeManager：节点管理器，主要负责维护本节点的资源情况和任务管理。首先NodeManager需要定期向ResourceManager汇报本节点资源使用情况，以便ResourceManager，根据资源使用情况，来分配资源给Application Master(这里的Application 指的是Spark 等应用程序)，其次，需要管理Applicaiton Master提交来的task，比如接收Applicaiton Master 启动或停止task的请求(启动和停止有NodeManager的组件**ContainersLanuncher**完成)。ApplicaitonMaster：用户提交的每个program都会对应一个ApplicationMaster，主要负责监控应用，任务容错（重启失败的task）等。它同时会和ResourceManager和NodeManager有交互，向ResourceManager申请资源，请求NodeManager启动或关闭task
- Container：容器是资源调度的单位，它是内存、cpu、磁盘、和IO的集合。Application Master会给task分配Container，task只能只用分配给它的Container的资源。分配流程为Resource Manager ->Application Master -> task
- 

解释上图的过程：

1.有YarnClient提交program信息打拼ResourceManager，包括（应用代码和应用需要的一切参数和环境信息）
2.ResourceManager收到请求之后，调用ApplicationMasterManager向NodeManager发送请求，申请一个资源（Container），并且要求Container启动ApplicationMaster.
3.ApplicationMaster启动之后，首先注册自己到ResourceManager，然后为自己的Task申请Container，这个过程是轮训的，循环申请资源，ResourceManager收到请求之后，会要求NodeManager分配资源，相当于还是ResourceManager分配的Container，这种资源调度程序可插拔。
4.资源分配完毕之后，Application Master发送请求到NodeManager，启动任务。
5.NodeManager设置Container的运行时环境（jar包，环境变量，任务启动脚本），NodeManager会通过脚本启动任务
6.启动的过程是由NodeManager的ContainerLauncher负责的，ContainerLauncher完成启动任务的工作
7.这一步是在作业执行过程中持续发生的，我用虚线代表，主要包括两类交互，第一，task和Application Master的交互，task会向AM汇报任务状态和进度信息，比如任务启动，停止，状态更新。Applicaiton Master利用这些信息监控task整个执行过程。第二，是NodeManager和ResourceManager的交互，这个过程和任务没有关系，主要是两者之间保持的心跳信息（状态的变化等等）
8.Application Master在检测到作业运行完毕之后，Application Master想Resource Manager 删除自己，并且停止自己执行

Yarn将资源封装抽象为Container，将应用抽象为Application Master,两个关键模型的抽象，实现了对资源和应用的统一管理，进而实现了调度平台和执行引擎的解耦。至于Application怎么样把Container分配给task，怎么样监控task的执行过程，完全是由执行引擎根据自身特性实现的。充分解耦的好处是，Yarn可以同时部署不同的执行引擎，集群不受限制，并且当一个执行引擎升级的时候，不会影响到别的引擎，这对于生产是至关重要的。并且Yarn提供了多种资源调度模式，以满足不同的生产环境。

资源调度：

1 FIFO Scheduler把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中排第一的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。
FIFO Scheduler是最简单也是最容易理解的调度器，也不需要任何配置，但它并不适用于共享集群。缺点：大的应用可能会占用大量资源，导致剩下的资源不足以支撑后面的job，这就导致其它应用被阻塞。
在共享集群中，更适合采用Capacity Scheduler或Fair Scheduler，这两个调度器都允许大任务和小任务在提交时获取指定的资源而不是全部，便于app的并行执行。
2 Capacity Scheduler
将应用挂在了ResourceManager上，且每个应用一开始就自动直到自己要挂在哪个队列上。

一个FIFO叫一个队列，队列内执行FIFO策略，队列间查找最闲队列（一个队列中资源正在运行的任务占用的资源和该队列的资源上限比值，选择最小的队列），然后按照作业优先级 + 提交时间 + 用户资源限制 + 硬件限制 对queue内的任务排序。

3 Fair Scheduler（HRM就是实现的这个方式）

队列内作业公平共享资源，但是会按照缺额由大到小，将作业的优先级排序。

4公平调度代码分析：

其中createSchedulerEventDispatcher()代码如下：

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/ResourceManager.java
  protected EventHandler<SchedulerEvent> createSchedulerEventDispatcher() {
    return new EventDispatcher(this.scheduler, "SchedulerEventDispatcher");
  }
```

其中scheduler对象是根据配置yarn.resourcemanager.yarn.resourcemanager指定的类生成的对象，这里使用org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler。

下面进入FairScheduler的handle方法：

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FairScheduler.java
  public void handle(SchedulerEvent event) {
    switch (event.getType()) {
    case NODE_ADDED://省略这部分逻辑
    case NODE_REMOVED://省略这部分逻辑
    case NODE_UPDATE:
      if (!(event instanceof NodeUpdateSchedulerEvent)) {
        throw new RuntimeException("Unexpected event type: " + event);
      }
      NodeUpdateSchedulerEvent nodeUpdatedEvent = (NodeUpdateSchedulerEvent)event;
      nodeUpdate(nodeUpdatedEvent.getRMNode());
      break;
    case APP_ADDED://省略这部分逻辑
    case APP_REMOVED://省略这部分逻辑
    case NODE_RESOURCE_UPDATE://省略这部分逻辑
    case APP_ATTEMPT_ADDED://省略这部分逻辑
    case APP_ATTEMPT_REMOVED://省略这部分逻辑
    case CONTAINER_EXPIRED://省略这部分逻辑
    default:
      LOG.error("Unknown event arrived at FairScheduler: " + event.toString());
    }
  }
```

由于NodeUpdateSchedulerEvent的事件类型是SchedulerEventType.NODE_UPDATE，这里会进入NODE_UPDATE处理逻辑。

由于NodeUpdateSchedulerEvent的事件类型是SchedulerEventType.NODE_UPDATE，这里会进入NODE_UPDATE处理逻辑。

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FairScheduler.java
  protected void nodeUpdate(RMNode nm) {
    try {
      writeLock.lock();
      long start = getClock().getTime();
      eventLog.log("HEARTBEAT", nm.getHostName());
      super.nodeUpdate(nm);

      FSSchedulerNode fsNode = getFSSchedulerNode(nm.getNodeID());
      attemptScheduling(fsNode);

      long duration = getClock().getTime() - start;
      fsOpDurations.addNodeUpdateDuration(duration);
    } finally {
      writeLock.unlock();
    }
  }
```

上面的代码中，

- 先调用父类的nodeUpdate()方法会进行container状态更新，和NM的状态的更新，这里跳过这部分逻辑。
- 然后获取了FSSchedulerNode的一个实例，并尝试进行调度。下面重点看一下这块逻辑。

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FairScheduler.java
  void attemptScheduling(FSSchedulerNode node) {
    try {
      writeLock.lock();
      if (rmContext.isWorkPreservingRecoveryEnabled() && !rmContext
          .isSchedulerReadyForAllocatingContainers()) {
        return;
      }

      final NodeId nodeID = node.getNodeID();
      if (!nodeTracker.exists(nodeID)) {
        // The node might have just been removed while this thread was waiting
        // on the synchronized lock before it entered this synchronized method
        LOG.info(
            "Skipping scheduling as the node " + nodeID + " has been removed");
        return;
      }

      // Assign new containers...
```

```
      // Assign new containers...
      // 1. 先确认抢占的板分配
      // 2. 再检查有没有reserved预留
      // 3. 最后再进行调度分配新的container

      // Apps may wait for preempted containers
      // We have to satisfy these first to avoid cases, when we preempt
      // a container for A from B and C gets the preempted containers,
      // when C does not qualify for preemption itself.
      assignPreemptedContainers(node);
      FSAppAttempt reservedAppSchedulable = node.getReservedAppSchedulable();
      boolean validReservation = false;
      if (reservedAppSchedulable != null) {
        validReservation = reservedAppSchedulable.assignReservedContainer(node);
      }
      if (!validReservation) {
        // No reservation, schedule at queue which is farthest below fair share
        int assignedContainers = 0;
        Resource assignedResource = Resources.clone(Resources.none());
        Resource maxResourcesToAssign = Resources.multiply(
            node.getUnallocatedResource(), 0.5f);
        while (node.getReservedContainer() == null) {
          //敲黑板，这里是分配的逻辑
          Resource assignment = queueMgr.getRootQueue().assignContainer(node);
          if (assignment.equals(Resources.none())) {
            break;
          }

          assignedContainers++;
          Resources.addTo(assignedResource, assignment);
          if (!shouldContinueAssigning(assignedContainers, maxResourcesToAssign,
              assignedResource)) {
            break;
          }
        }
      }
      updateRootQueueMetrics();
    } finally {
      writeLock.unlock();
    }
  }
```

这里不关注抢占和reserve机制，重点关注分配新container的部分，这里是调用queueMgr，找到RootQueue，然后调用了它的assignContainer(node)方法。

```
//位置：org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSParentQueue.java
  public Resource assignContainer(FSSchedulerNode node) {
    Resource assigned = Resources.none();

    // 超过了max share就直接返回
    if (!assignContainerPreCheck(node)) {
      return assigned;
    }

    // 对所有的子队列进行排序
    // 排序的策略在fair-scheduler.xml里面的配置项defaultQueueSchedulingPolicy决定，这里使用drf
    //
    writeLock.lock();
    try {
      Collections.sort(childQueues, policy.getComparator());
    } finally {
      writeLock.unlock();
    }

    /*
     * 这里释放了写锁，加入了读锁
     * 这样可能会带来两个问题：
     * 1. 新增了一个queue，不影响结果正确性，下次会再处理新queue
     * 2. 删除了一个queue，这个最好处理一下，不过目前没有处理，也还好
     */
    readLock.lock();
    try {
      for (FSQueue child : childQueues) {
        assigned = child.assignContainer(node);
        if (!Resources.equals(assigned, Resources.none())) {
          break;
        }
      }
    } finally {
      readLock.unlock();
    }
    return assigned;
  }
```

关于drf的思想参考：Yarn源码分析4-资源调度排序算法.

这里重点介绍一下读锁内的assignContainer的部分：

这里重点介绍一下读锁内的assignContainer的部分：

- 通过遍历所有孩子节点，递归的调用assignContainer方法。
  - 如果孩子节点是FSParentQueue类型，那么还是递归进入到刚刚才一样的逻辑中。
  - 如果孩子节点是FSLeafQueue类型，那么进入到后面的逻辑：

```
//位置：org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSLeafQueue.java
  public Resource assignContainer(FSSchedulerNode node) {
    Resource assigned = none();
    if (LOG.isDebugEnabled()) {
      LOG.debug("Node " + node.getNodeName() + " offered to queue: " +
          getName() + " fairShare: " + getFairShare());
    }
    // 检查是否超过了max share
    if (!assignContainerPreCheck(node)) {
      return assigned;
    }

    for (FSAppAttempt sched : fetchAppsWithDemand(true)) {
      if (SchedulerAppUtils.isPlaceBlacklisted(sched, node, LOG)) {
        continue;
      }
      assigned = sched.assignContainer(node);
      if (!assigned.equals(none())) {
        if (LOG.isDebugEnabled()) {
          LOG.debug("Assigned container in queue:" + getName() + " " +
              "container:" + assigned);
        }
        break;
      }
    }
    return assigned;
  }
```

上面这块代码，有两个点需要重点关注：

- fetchAppsWithDemand：找到饥饿的app列表，并按照drf的策略进行排序，然后遍历
- FSAppAttempt类的实例sched.assignContainer(node)方法，这里会进行container的分配，下面进入这块逻辑：

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSAppAttempt.java
  public Resource assignContainer(FSSchedulerNode node) {
    if (isOverAMShareLimit()) {
      PendingAsk amAsk = appSchedulingInfo.getNextPendingAsk();
      updateAMDiagnosticMsg(amAsk.getPerAllocationResource(),
          " exceeds maximum AM resource allowed).");
      if (LOG.isDebugEnabled()) {
        LOG.debug("AM resource request: " + amAsk.getPerAllocationResource()
            + " exceeds maximum AM resource allowed, "
            + getQueue().dumpState());
      }
      return Resources.none();
    }
    return assignContainer(node, false);
  }
```

首先判断是否达到了队列中可用于运行AM的资源比例限制，如果没有的话，继续跟进：

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSAppAttempt.java
  private Resource assignContainer(FSSchedulerNode node, boolean reserved) {
    if (LOG.isTraceEnabled()) {
      LOG.trace("Node offered to app: " + getName() + " reserved: " + reserved);
    }

    // 按照priority进行排序
    Collection<SchedulerRequestKey> keysToTry = (reserved) ?
        Collections.singletonList(
            node.getReservedContainer().getReservedSchedulerKey()) :
        getSchedulerKeys();

    // For each priority, see if we can schedule a node local, rack local
    // or off-switch request. Rack of off-switch requests may be delayed
    // (not scheduled) in order to promote better locality.
    try {
```

```
    try {
      writeLock.lock();

      // 按priority从高到低遍历所有的ResourceRequest
      // 如果一个ResourceRequest可以在当前node上分配出来，就进入分配逻辑
      for (SchedulerRequestKey schedulerKey : keysToTry) {
        // 跳过无法在当前node上进行分配的请求。
        // hasContainerForNode()这个函数会分node,rack,any三种情况来考虑是否有合适的container。
        // 并且也会考虑当前node上剩余的资源是否还足够分配
        if (!reserved && !hasContainerForNode(schedulerKey, node)) {
          continue;
        }

        // 调度机会计数加1
        addSchedulingOpportunity(schedulerKey);

        PendingAsk rackLocalPendingAsk = getPendingAsk(schedulerKey,
            node.getRackName());
        PendingAsk nodeLocalPendingAsk = getPendingAsk(schedulerKey,
            node.getNodeName());

        // 如果有node级别的locality请求，并且不支持relaxLocality，那就给个warn。
        if (nodeLocalPendingAsk.getCount() > 0
            && !appSchedulingInfo.canDelayTo(schedulerKey,
            node.getNodeName())) {
          LOG.warn("Relax locality off is not supported on local request: "
              + nodeLocalPendingAsk);
        }

        NodeType allowedLocality;
        if (scheduler.isContinuousSchedulingEnabled()) {
          allowedLocality = getAllowedLocalityLevelByTime(schedulerKey,
              scheduler.getNodeLocalityDelayMs(),
              scheduler.getRackLocalityDelayMs(),
              scheduler.getClock().getTime());
        } else {
          allowedLocality = getAllowedLocalityLevel(schedulerKey,
              scheduler.getNumClusterNodes(),
              scheduler.getNodeLocalityThreshold(),
              scheduler.getRackLocalityThreshold());
        }

        // 如果同时有node和rack级别的请求，就以NODE_LOCAL为参数进入下一步的assignContainer函数，并返回
        if (rackLocalPendingAsk.getCount() > 0
            && nodeLocalPendingAsk.getCount() > 0) {
          if (LOG.isTraceEnabled()) {
            LOG.trace("Assign container on " + node.getNodeName()
                + " node, assignType: NODE_LOCAL" + ", allowedLocality: "
                + allowedLocality + ", priority: " + schedulerKey.getPriority()
```

```
        // 如果同时有node和rack级别的请求，就以NODE_LOCAL为参数进入下一步的assignContainer函数，并返回
        if (rackLocalPendingAsk.getCount() > 0
            && nodeLocalPendingAsk.getCount() > 0) {
          if (LOG.isTraceEnabled()) {
            LOG.trace("Assign container on " + node.getNodeName()
                + " node, assignType: NODE_LOCAL" + ", allowedLocality: "
                + allowedLocality + ", priority: " + schedulerKey.getPriority()
                + ", app attempt id: " + this.attemptId);
          }
          return assignContainer(node, nodeLocalPendingAsk, NodeType.NODE_LOCAL,
              reserved, schedulerKey);
        }

        // 如果错过了上面的node级别，并且还不支持降级到rack级别，那就跳过这次调度机会
        if (!appSchedulingInfo.canDelayTo(schedulerKey, node.getRackName())) {
          continue;
        }

        // 以RACK_LOCAL为参数进入下一步的assignContainer函数，并返回
        if (rackLocalPendingAsk.getCount() > 0
            && (allowedLocality.equals(NodeType.RACK_LOCAL) || allowedLocality
            .equals(NodeType.OFF_SWITCH))) {
          if (LOG.isTraceEnabled()) {
            LOG.trace("Assign container on " + node.getNodeName()
                + " node, assignType: RACK_LOCAL" + ", allowedLocality: "
                + allowedLocality + ", priority: " + schedulerKey.getPriority()
                + ", app attempt id: " + this.attemptId);
          }
          return assignContainer(node, rackLocalPendingAsk, NodeType.RACK_LOCAL,
              reserved, schedulerKey);
        }

        PendingAsk offswitchAsk = getPendingAsk(schedulerKey,
            ResourceRequest.ANY);
        if (!appSchedulingInfo.canDelayTo(schedulerKey, ResourceRequest.ANY)) {
          continue;
        }

        // 以OFF_SWITCH为参数进入下一步的assignContainer函数，并返回
        if (offswitchAsk.getCount() > 0) {
          if (getAppPlacementAllocator(schedulerKey).getUniqueLocationAsks()
              <= 1 || allowedLocality.equals(NodeType.OFF_SWITCH)) {
            if (LOG.isTraceEnabled()) {
              LOG.trace("Assign container on " + node.getNodeName()
                  + " node, assignType: OFF_SWITCH" + ", allowedLocality: "
                  + allowedLocality + ", priority: "
                  + schedulerKey.getPriority()
                  + ", app attempt id: " + this.attemptId);
            }
```

上面的的代码，主要逻辑是按priority从高到低的顺序遍历所有的ResourceRequest，针对每个ResourceRequest，在当前的node上面，找到适合它的locality，并将这个locality传入到下一级的assignContainer()函数中。

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSAppAttempt.java
  private Resource assignContainer(
      FSSchedulerNode node, PendingAsk pendingAsk, NodeType type,
      boolean reserved, SchedulerRequestKey schedulerKey) {

    // 当前的request需要多少资源
    Resource capability = pendingAsk.getPerAllocationResource();

    // 当前这个Node有多少资源
    Resource available = node.getUnallocatedResource();

    Container reservedContainer = null;
    if (reserved) {
      reservedContainer = node.getReservedContainer().getContainer();
    }

    // 资源够分配的
    if (Resources.fitsIn(capability, available)) {
      // 划重点：分配一个container出来
      RMContainer allocatedContainer =
          allocate(type, node, schedulerKey, pendingAsk,
              reservedContainer);
      if (allocatedContainer == null) {
        // Did the application need this resource?
        if (reserved) {
          unreserve(schedulerKey, node);
        }
        return Resources.none();
      }

      // If we had previously made a reservation, delete it
      if (reserved) {
        unreserve(schedulerKey, node);
      }

      // Inform the node
      node.allocateContainer(allocatedContainer);

```

```
      // If not running unmanaged, the first container we allocate is always
      // the AM. Set the amResource for this app and update the leaf queue's AM
      // usage
      if (!isAmRunning() && !getUnmanagedAM()) {
        setAMResource(capability);
        getQueue().addAMResourceUsage(capability);
        setAmRunning(true);
      }

      return capability;
    }

    // 以下逻辑，是资源不够分配的情况，判断一下是否需要reserve
    if (LOG.isDebugEnabled()) {
      LOG.debug("Resource request: " + capability + " exceeds the available"
          + " resources of the node.");
    }

    if (isReservable(capability) && //如果应用比较饥饿，并且请求的资源超过了一定大小
        !node.isPreemptedForApp(this) && //不准备在这个节点进行抢占，否则有可能有多个Reserve
        reserve(pendingAsk.getPerAllocationResource(), node, reservedContainer,
            type, schedulerKey)) {
      updateAMDiagnosticMsg(capability, " exceeds the available resources of "
          + "the node and the request is reserved)");
      if (LOG.isDebugEnabled()) {
        LOG.debug(getName() + "'s resource request is reserved.");
      }
      return FairScheduler.CONTAINER_RESERVED;
    } else {
      updateAMDiagnosticMsg(capability, " exceeds the available resources of "
          + "the node and the request cannot be reserved)");
      if (LOG.isDebugEnabled()) {
        LOG.debug("Couldn't create reservation for app:  " + getName()
            + ", at priority " +  schedulerKey.getPriority());
      }
      return Resources.none();
    }
  }
```

上面代码中，主要是看一下能否分配一个container出来，如果不能话，那么看一下能否进行一次reserve。

这是重点关注分配一个container出来的逻辑：

```
//位置: org/apache/hadoop/yarn/server/resourcemanager/scheduler/fair/FSAppAttempt.java
  public RMContainer allocate(NodeType type, FSSchedulerNode node,
      SchedulerRequestKey schedulerKey, PendingAsk pendingAsk,
      Container reservedContainer) {
    RMContainer rmContainer;
    Container container;

    try {
      writeLock.lock();
      // Update allowed locality level
      NodeType allowed = allowedLocalityLevel.get(schedulerKey);
      if (allowed != null) {
        if (allowed.equals(NodeType.OFF_SWITCH) && (type.equals(
            NodeType.NODE_LOCAL) || type.equals(NodeType.RACK_LOCAL))) {
          this.resetAllowedLocalityLevel(schedulerKey, type);
        } else if (allowed.equals(NodeType.RACK_LOCAL) && type.equals(
            NodeType.NODE_LOCAL)) {
          this.resetAllowedLocalityLevel(schedulerKey, type);
        }
      }

      // Required sanity check - AM can call 'allocate' to update resource
      // request without locking the scheduler, hence we need to check
      if (getOutstandingAsksCount(schedulerKey) <= 0) {
        return null;
      }

      container = reservedContainer;
      if (container == null) {
        // 敲黑板，这里会创建一个container实例出来。
        container = createContainer(node, pendingAsk.getPerAllocationResource(),
            schedulerKey);
      }

      // 下面的逻辑是记录这个新创建出来的container
      // Create RMContainer
      rmContainer = new RMContainerImpl(container, schedulerKey,
          getApplicationAttemptId(), node.getNodeID(),
          appSchedulingInfo.getUser(), rmContext);
      ((RMContainerImpl) rmContainer).setQueueName(this.getQueueName());

```

```
      // 重点关注：这里会把rmContainer记录下来，等待下次AM心跳时，会从这里把分配出来的container带走
      addToNewlyAllocatedContainers(node, rmContainer);
      liveContainers.put(container.getId(), rmContainer);

      // Update consumption and track allocations
      List<ResourceRequest> resourceRequestList = appSchedulingInfo.allocate(
          type, node, schedulerKey, container);
      this.attemptResourceUsage.incUsed(container.getResource());
      getQueue().incUsedResource(container.getResource());

      // Update resource requests related to "request" and store in RMContainer
      ((RMContainerImpl) rmContainer).setResourceRequests(resourceRequestList);

      // Inform the container
      rmContainer.handle(
          new RMContainerEvent(container.getId(), RMContainerEventType.START));

      if (LOG.isDebugEnabled()) {
        LOG.debug("allocate: applicationAttemptId=" + container.getId()
            .getApplicationAttemptId() + " container=" + container.getId()
            + " host=" + container.getNodeId().getHost() + " type=" + type);
      }
      RMAuditLogger.logSuccess(getUser(), AuditConstants.ALLOC_CONTAINER,
          "SchedulerApp", getApplicationId(), container.getId(),
          container.getResource());
    } finally {
      writeLock.unlock();
    }

    return rmContainer;
  }
```

公平调度 容量调度的区别，主要是fair的多余功能：

1.4. Fair Scheduler与Capacity Scheduler区别

- 资源公平共享：在每个队列中，Fair Scheduler可选择按照FIFO、Fair或DRF策略为应用程序分配资源。Fair策略即平均分配，默认情况下，每个队列采用该方式分配资源
- 支持资源抢占：当某个队列中有剩余资源时，调度器会将这些资源共享给其他队列，而当该队列中有新的应用程序提交时，调度器要为它回收资源。为了尽可能降低不必要的计算浪费，调度器采用了先等待再强制回收的策略，即如果等待一段时间后尚有未归还的资源，则会进行资源抢占：从那些超额使用资源的队列中杀死一部分任务，进而释放资源
- 负载均衡：Fair Scheduler提供了一个基于任务数的负载均衡机制，该机制尽可能将系统中的任务均匀分配到各个节点上。此外，用户也可以根据自己的需求设计负载均衡机制
- 调度策略灵活配置：Fiar Scheduler允许管理员为每个队列单独设置调度策略（当前支持FIFO、Fair或DRF三种）
- 提高小应用程序响应时间：由于采用了最大最小公平算法，小作业可以快速获取资源并运行完成

安装k8s

2021年9月11日 16:20

1 【ERROR ImagePull]: failed to pull image registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:v1.8.4: output: Error response from daemon: manifest for registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:v1.8.4 not found: manifest unknown: manifest unknown
原因：外网
执行：
docker images//检查镜像
docker pull coredns/coredns:1.8.4
docker tag coredns/coredns:1.8.4 registry.aliyuncs.com/google_containers/coredns:v1.8.4 $ docker rmi -f coredns/coredns:1.8.4
docker rmi -f coredns/coredns:1.8.4

2 curl 没有https:/重新从官网下载curl
执行./configure --with-ssl

3 kube初始化过程：
kubeadm init --pod-network-cidr 172.16.0.0/16 --image-repository registry.cn-hangzhou.aliyuncs.com/google_containers

4 初始化时，kube端口占用问题/etc/kubernetes/manifests/etcd.yaml already exists：
kubeadm reset

5 初始化时，The HTTP call equal to 'curl -sSL http://localhost:10248/healthz' failed with error: Get "http://localhost:10248/healthz": dial tcp [::1]:10248: connect: connection refused
systemctl daemon-reload
systemctl restart kubelet
再执行初始化

6

Mesos使用GPU

2021年12月15日 10:22

Master

/soft/mesos-1.6.1/mesos_install/etc/mesos/masters文件

```
master00
#slave01
```

Slave文件

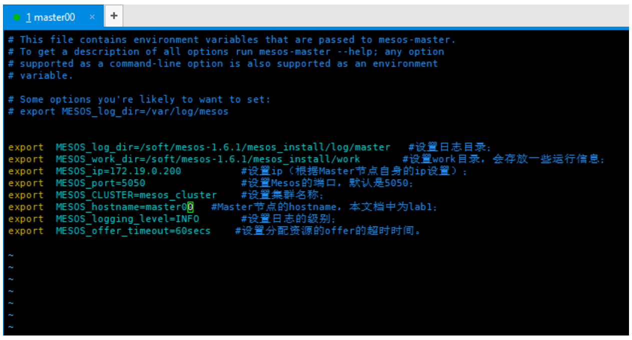
```
1 slave01
2 slave02
3 slave03
4 slave04
5 slave05
6 slave06
7 slave07
8 #master00
```

attention

Agent.sh文件

```
# This file contains environment variables that are passed to mesos-agent.
# To get a description of all options run mesos-agent --help; any option
# supported as a command-line option is also supported as an environment
# variable.

# You must at least set MESOS_master.

# The mesos master URL to contact. Should be host:port for
# non-ZooKeeper based masters, otherwise a zk:// or file:// URL.
export MESOS_master=172.19.0.200:5050

# Other options you're likely to want to set:
# export MESOS_log_dir=/var/log/mesos
# export MESOS_work_dir=/var/run/mesos
# export MESOS_isolation=cgroups

export MESOS_ip=172.19.0.200    #本节点的ip;
export MESOS_port=5051          #设置Mesos的端口，Slave节点一般为5051;
export MESOS_hostname=master00
export MESOS_log_dir=/soft/mesos-1.6.1/mesos_install/log/slave    # 设置节点的日志目录;
export MESOS_work_dir=/soft/mesos-1.6.1/mesos_install/work   #设置节点的work目录。
export MESOS_logging_level=INFO
```

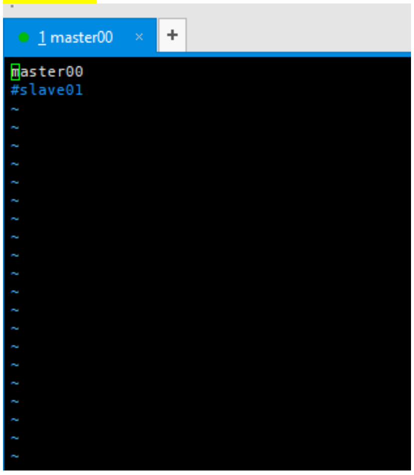
master文件



```
# This file contains environment variables that are passed to mesos-master.
# To get a description of all options run mesos-master --help; any option
# supported as a command-line option is also supported as an environment
# variable.

# Some options you're likely to want to set:
# export MESOS_log_dir=/var/log/mesos


export  MESOS_log_dir=/soft/mesos-1.6.1/mesos_install/log/master   #设置日志目录;
export  MESOS_work_dir=/soft/mesos-1.6.1/mesos_install/work       #设置work目录，会存放一些运行信息;
export  MESOS_ip=172.19.0.200          #设置ip（根据Master节点自身的ip设置）;
export  MESOS_port=5050                #设置Mesos的端口，默认是5050;
export  MESOS_CLUSTER=mesos_cluster    #设置集群名称;
export  MESOS_hostname=master00   #Master节点的hostname，本文档中为lab1;
export  MESOS_logging_level=INFO       #设置日志的级别;
export  MESOS_offer_timeout=60secs    #设置分配资源的offer的超时时间。

~
~
~
~
~
~
~
```

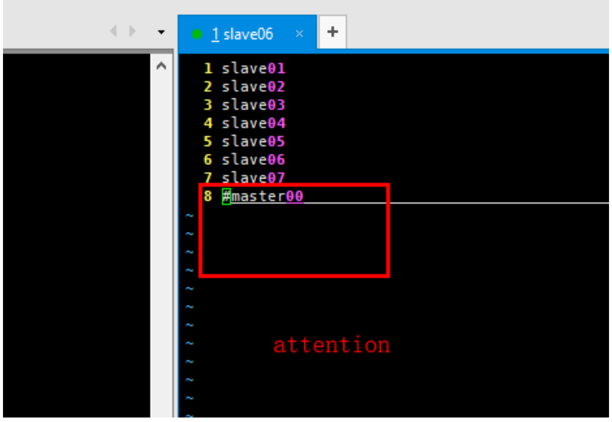
每一个slave

master文件



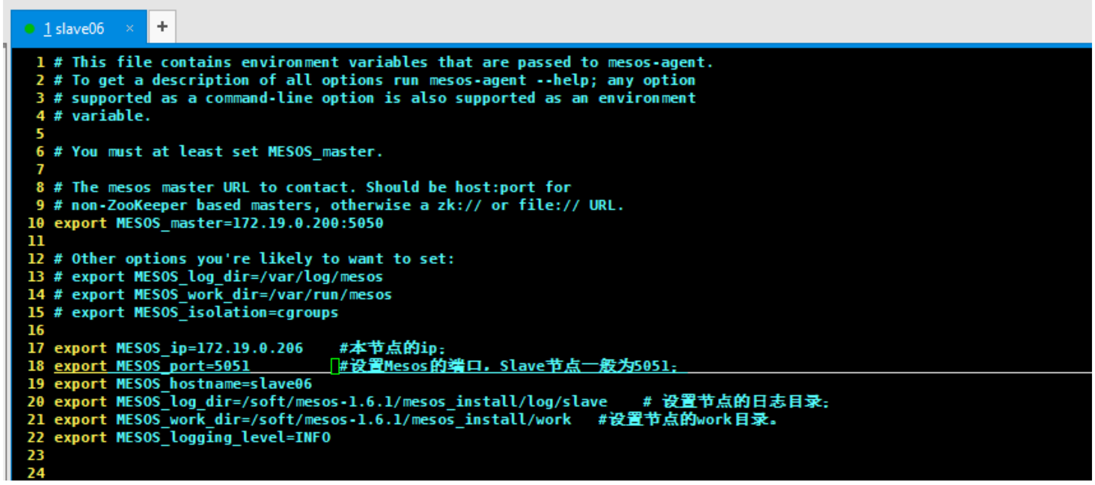
```
master00
#slave01
~
~
~
```

Slave文件



```
slave01
slave02
slave03
slave04
slave05
slave06
slave07
#master00
~
~
```

agent文件



```
# This file contains environment variables that are passed to mesos-agent.
# To get a description of all options run mesos-agent --help; any option
# supported as a command-line option is also supported as an environment
# variable.

# You must at least set MESOS_master.

# The mesos master URL to contact. Should be host:port for
# non-ZooKeeper based masters, otherwise a zk:// or file:// URL.
export MESOS_master=172.19.0.200:5050

# Other options you're likely to want to set:
# export MESOS_log_dir=/var/log/mesos
# export MESOS_work_dir=/var/run/mesos
# export MESOS_isolation=cgroups

export MESOS_ip=172.19.0.206     #本节点的ip;
export MESOS_port=5051           #设置Mesos的端口，Slave节点一般为5051;
export MESOS_hostname=slave06
export MESOS_log_dir=/soft/mesos-1.6.1/mesos_install/log/slave    # 设置节点的日志目录;
export MESOS_work_dir=/soft/mesos-1.6.1/mesos_install/work    #设置节点的work目录。
export MESOS_logging_level=INFO


```

Master.env文件

```
1 slave06

# This file contains environment variables that are passed to mesos-master.
# To get a description of all options run mesos-master --help; any option
# supported as a command-line option is also supported as an environment
# variable.

# Some options you're likely to want to set:
# export MESOS_log_dir=/var/log/mesos

export  MESOS_log_dir=/soft/mesos-1.6.1/mesos_install/log/master   #设置日志目录;
export  MESOS_work_dir=/soft/mesos-1.6.1/mesos_install/work       #设置work目录，会存放一些运行信息;
export  MESOS_ip=172.19.0.206                #设置ip（根据Master节点自身的ip设置）;
export  MESOS_port=5050                #设置Mesos的端口，默认是5050;
export  MESOS_CLUSTER=mesos_cluster    #设置集群名称;
export  MESOS_hostname=slave06    #Master节点的hostname，本文档中为lab1;
export  MESOS_logging_level=INFO       #设置日志的级别;
export  MESOS_offer_timeout=60secs    #设置分配资源的offer的超时时间。
```

系统启动

1 在mesos master执行
mesos-start-cluster.sh

2在mesos slave节点执行
mesos-stop-agent.sh

3 在agent每一个节点继续执行
rm -f /soft/mesos-1.6.1/mesos_install/work/meta/slaves/latest

./sbin/mesos-agent --isolation="filesystem/linux,cgroups/devices,gpu/nvidia" --nvidia_gpu_devices="0" --resources="gpus:1" --work_dir=/soft/mesos-1.6.1/mesos_install/work --master=172.19.0.200:5050
#当高亮部分省略，mesos可以看见系统所有GPU，高亮部分只能看见GPUID=0的GPU设备