批量计算

最佳实践

最佳实践

使用 Docker 镜像构建 App

批量计算提供了 App 功能，可以使用虚拟机（VM）镜像来定制运行环境，也可以使用 Docker 镜像，本文将介绍如何使用 Docker 镜像创建 App 和提交 App 作业。

背景

如果您的作业使用了 ISV 提供的软件或算法，可以考虑将其封装在 Docker 镜像中，再使用 App 设置作业的模板（包括资源类型和运行环境），这样一来，提交作业时只需提供输入和输出信息即可。当软件或算法有更新时，只需要更新 Docker 镜像，比如通过 Docker 镜像的 Tag 来标识不同的版本号，修改 App 中 Docker 镜像的版本号即可完成运行环境的更新。

1. 准备 App 的 Docker 镜像

A) 制作 Docker 镜像

根据自己的需求，用户可以使用官方镜像仓库中的镜像作为基础镜像，安装需要的软件或算法，制作成 Docker 镜像，完成运行环境的定制；制作镜像有两种方法：

- 使用 Dockfile 制作镜像
- 使用容器快速制作镜像

具体制作方法可参考用户指南中的 Docker 镜像制作。

> **建议**：在制作 Docker 镜像时，最好带上 Tag，后续版本有更新时，只需要更新 Tag 即可。

B) 本地调试Docker镜像

Docker 镜像制作完成以后，可以参考用户指南中的 Docker 本地调试相关章节进行本地调试，确保 Docker 镜像在 BatchCompute 的环境下可以正常使用。

C) 推送到镜像仓库

可以将制作好的 Docker 镜像推送到 OSS 的镜像仓库。具体方法请参考用户指南中 Docker镜像上传到 OSS 的详细描述 。

2. 创建 App

BatchCompute提供了 API、SDK、控制台等三种方式创建 App，下面以控制台和 Python SDK 为例，分别介绍如何使用 Docker 镜像创建 App。

A) 使用控制台创建 App

假如 Docker 镜像被推送到 OSS 镜像仓库的路径为oss://demo-bucket/dockers/，镜像名称为 localhost:5000/demodockerimage:0.1。

如上图所示，在创建 App 时，选择镜像类型为 Docker，填写 Docker 镜像的名称，以及 OSS Registry 的路径。关于控制台如何创建 App 的其他参数详情，请参考用户指南中创建 App 的描述，这里不再赘述。

B) 使用 SDK 创建 App

使用 Python SDK 创建 App 时，参考如下的形式：

```
#encoding=utf-8
import sys
from batchcompute import Client, ClientError
from batchcompute import CN_BEIJING as REGION
from batchcompute.resources import (
JobDescription, TaskDescription, DAG, AutoCluster, GroupDescription, ClusterDescription, AppDescription
)
ACCESS_KEY_ID='xxxx' # 填写您的 ACCESS_KEY_ID
ACCESS_KEY_SECRET='xxxx' # 填写您的 ACCESS_KEY_SECRET
def main():
try:
client = Client(REGION, ACCESS_KEY_ID, ACCESS_KEY_SECRET)
app_desc = {
"Name":"Docker-app-demo",
"Daemonize":False,
"Docker":{
"Image":"localhost:5000/demodockerimage:0.1",
"RegistryOSSPath":"oss://demo-bucket/dockers/"
},
"CommandLine":"python test.py",

#其他参数这里不详细展示
```

```
}
appName = client.create_app(app_desc).Name
print('App created: %s' % appName)
except ClientError, e:
print (e.get_status_code(), e.get_code(), e.get_requestid(), e.get_msg())
if __name__ == '__main__':
sys.exit(main())
```

如上面的实例代码所示，在AppDescription中填写 Docker 信息的Image和RegistryOSSPath。其他参数请参考用户指南中的创建示例。

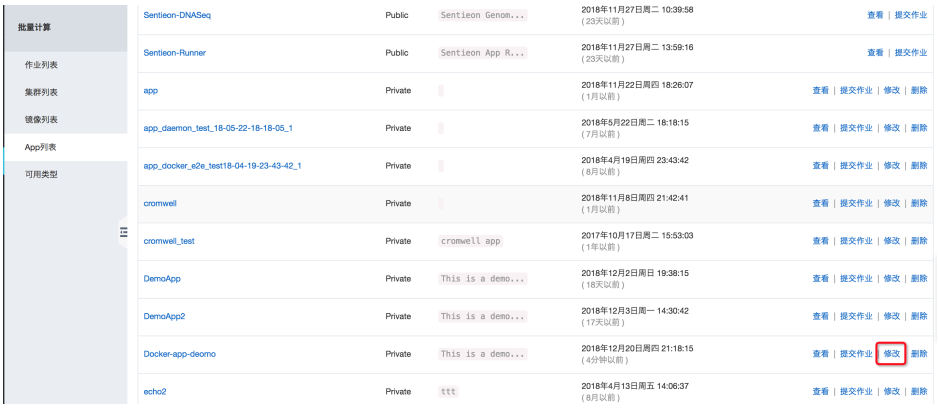
3. 提交 App 作业

提交作业时，不再涉及 Docker 相关的信息，具体方法请参考用户指南中提交 App 作业的描述。

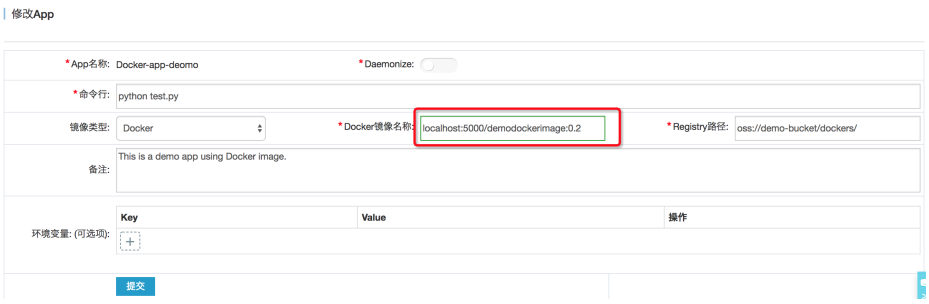
4. Docker 镜像更新

假如 App 中使用的 ISV 提供的软件或算法有更新，您只需要更新 Docker 镜像，并用 Tag 标识版本。然后更新 App 信息中的 Docker 镜像名称就可以。

A) 使用控制台更新



如上图所示，在 App 列表中找到需要更新的 App，点击**修改**按钮进入 App 的修改页面。



如上图所示，在修改页面，修改 App 的 **Docker 镜像名称**后，点击**提交**即可完成 App 的更新。

B) 使用 SDK 更新

使用 Python SDK 来更新 App 的 Docker 信息可参考如下示例：

```
#encoding=utf-8
import sys
from batchcompute import Client, ClientError
from batchcompute import CN_BEIJING as REGION
from batchcompute.resources import (
JobDescription, TaskDescription, DAG, AutoCluster, GroupDescription, ClusterDescription, AppDescription
)
ACCESS_KEY_ID='xxxx' # 填写您的 ACCESS_KEY_ID
ACCESS_KEY_SECRET='xxxx' # 填写您的 ACCESS_KEY_SECRET
def main():
try:
client = Client(REGION, ACCESS_KEY_ID, ACCESS_KEY_SECRET)
app_desc = {
"Name":"Docker-app-demo",
"Daemonize":False,
"Docker":{
"Image":"localhost:5000/demodockerimage:0.2",
"RegistryOSSPath":"oss://demo-bucket/dockers/"
},
"CommandLine":"python test.py",
"EnvVars": {}
}

res = client.modify_app("Docker-app-demo", app_desc)
print res

except ClientError, e:
print (e.get_status_code(), e.get_code(), e.get_requestid(), e.get_msg())

if __name__ == '__main__':
sys.exit(main())
```

对于简单的修改 Docker 版本号的情况，推荐使用控制台，操作更简单。

Blender渲染App最佳实践

本篇主要是介绍如何将渲染软件 Blender 创建成 BatchCompute 的 App，并通过此 App 提交 Blender 渲染作业。

Blender 是目前最流行的一款开源的跨平台全能三维动画制作软件，提供从建模、动画、材质、渲染、到音频处理、视频剪辑等一系列动画短片制作解决方案。 具体介绍可以看这里：https://www.blender.org/features/。

1. 准备工作

(1) 开通服务

- 开通批量计算服务（BatchCompute）: https://help.aliyun.com/document_detail/127644.html
- 开通对象存储服务（OSS）: https://oss.console.aliyun.com
- 开通MNS服务: https://mns.console.aliyun.com
- 开通容器服务: https://cr.console.aliyun.com

如果已经开通，请忽略此步骤。

(2) 地域的选择

本篇例子所有阿里云服务都需要使用相同的地域。

本篇例子使用地域： 华南 1 (深圳)

(3) 准备OSS Bucket

- 请到OSS控制台 创建一个Bucket。

本篇例子假设创建的 bucket 名称为：blender-demo, 地域在华南 1 (深圳)。

> 注意: 使用批量计算时，地域需要和 OSS bucket 的地域相同。
>
> 注意: 实际操作时，需要将例子中的bucket 名称修改为您自己创建的真实的bucket名称。

2. 制作 Blender Docker 镜像

(1) 创建一个Dockerfile文件

文件名：Dockerfile， 内容如下:

```
FROM ubuntu:latest

MAINTAINER your-name<your-email>

# 更新源
RUN apt update

# 清除缓存
RUN apt autoclean

# 安装
RUN apt install python python-pip curl pulseaudio blender -y
```

```
# 启动时运行这个命令
CMD ["/bin/bash"]
```

(2) build

```
docker build -t ubuntu-blender ./
```

等待完成，然后使用下面的命令查看是否有 ubuntu-blender

```
docker images
```

(3) check

```
docker run -t ubuntu-blender blender -v
```

显示：

Blender 2.79 (sub 0)

记住此版本信息，下面要用到。

3. Docker镜像上传

您需要将 ubuntu-blender 上传到 BatchCompute 支持Registry。

BatchCompute支持2种Registry：阿里云的 CR（Container Registry）和阿里云的OSS。

选择一种即可，推荐第一种: CR。

如何上传，请参考这2篇文档：

docker 镜像上传到CR

docker 镜像上传到OSS

假设已经上传到CR(地域：华南1-深圳)，名称为: registry.cn-shenzhen.aliyuncs.com/batchcompute_test/blender:1.0

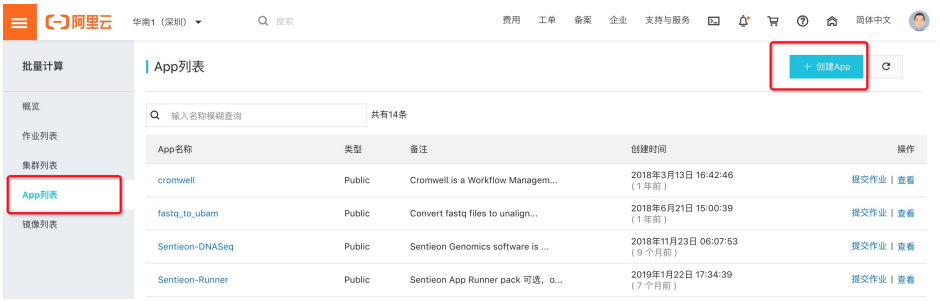
4. 创建 App

BatchCompute 提交作业，需要配置很多参数。BatchCompute 提供的 App 模板机制，让用户很方便预设参数默认值，提交作业时，只需填写少量参数即可。

下面我们来创建一个 Blender 渲染 App。

(1) 开始创建 App

打开批量计算控制台: https://batchcompute.console.aliyun.com



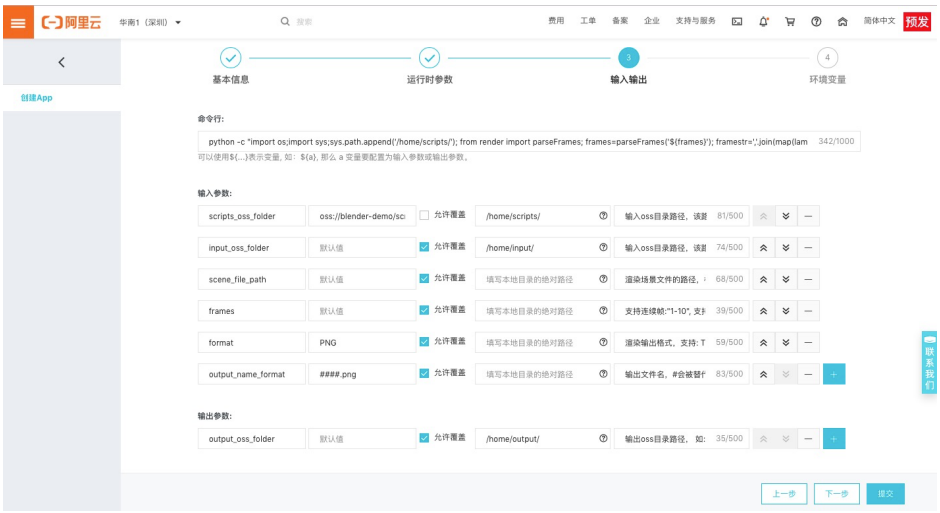
填写基本信息

Docker 镜像名称，填写您已经上传到CR的镜像名称，如： registry.cn-shenzhen.aliyuncs.com/batchcompute_test/blender:1.0

运行时参数

只需修改 实例类型为 8核16GB规格，其他的默认即可。

(2) 命令行和参数配置



命令行填写:

```
python -c "import os;import sys;sys.path.append('/home/scripts/'); from framer import parseFrames; frames=parseFrames('${frames}'); framestr=','.join(map(lambda x:str(x), frames)); s='blender -b /home/input/${scene_file_path} -o /home/output/result/${output_name_format} -F ${format} -f %s' % framestr; print('exec: %s' % s); os.system(s);"
```

${..} 都是变量，可以作为输入和输出参数。在使用此App提交作业的时候，传入的参数将替换掉这些变量。

参考文档 Blender 2.79 命令行参数

输入参数

注意: 实际操作时，需要将例子中的bucket 名称修改为您自己创建的真实的bucket名称。

名称	默认值	允许覆盖	本地目录绝对路径	备注
scripts_oss_folder	oss://blender-demo/scripts/	否	/home/scripts/	输入oss目录路径，该路径将挂载到虚拟机的/home/scripts/，应该包含要渲染的 framer.py 文件, 如: oss://bucket/scripts/
input_oss_folder		是	/home/input/	输入oss目录路径，该路径将挂载到虚拟机的/home/input/，应该包含要渲染的.blend文件, 如: oss://bucket/input/
scene_file_path		是		渲染场景文件的路径，相对于input_oss_folder的目录路径，如：a.blend 或者 folder_name/a.blend
frames		是		支持连续帧:”1-10”，支持多帧(逗号隔开,无空格):”1,3,5-10”
format	PNG	是		渲染输出格式，支持: TGA,RAWTGA,JPEG,IRIS,IRIZ,AVIRAW,AVIJPEG,PNG,BMP
output_name_format	####.png	是		输出文件名，#会被替代为帧序号，不足位补零。举例: test_###.png 变成 test_001.png，可以在前面加目录名: test/test_###.png

输出参数

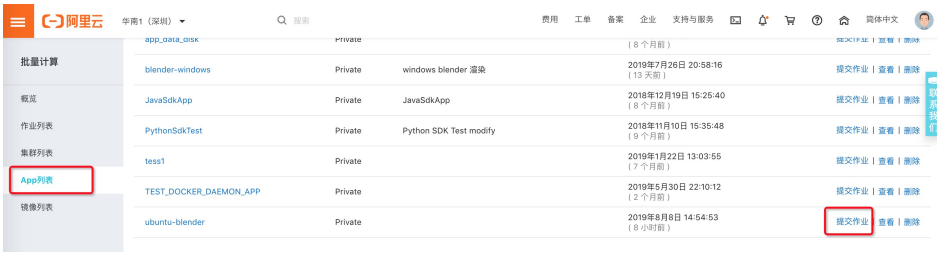
名称	默认值	允许覆盖	本地目录绝对路径	备注
output_oss_folder		是	/home/output/	输出oss目录路径，如: oss://bucket/output/。

环境变量

环境变量可以不用配置， 直接提交即可。

4. 提交渲染作业

在App列表中可以看到已经创建好的 ubuntu-blender， 点击”提交作业”。



(1) 准备工作

在提交作业前，还有一些准备工作。

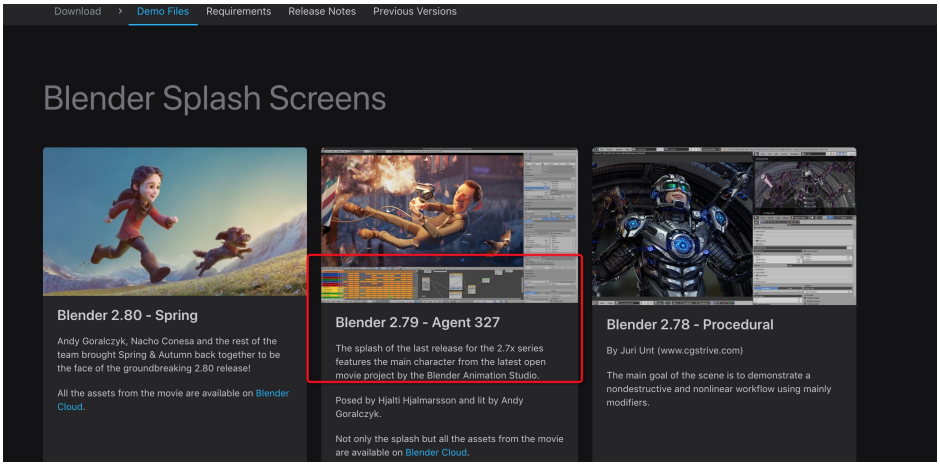
手动上传分帧器

分帧器python代码（见附录），上传到您的OSS目录下，比如: oss://blender-demo/scripts/framer.py

手动上传blender场景文件

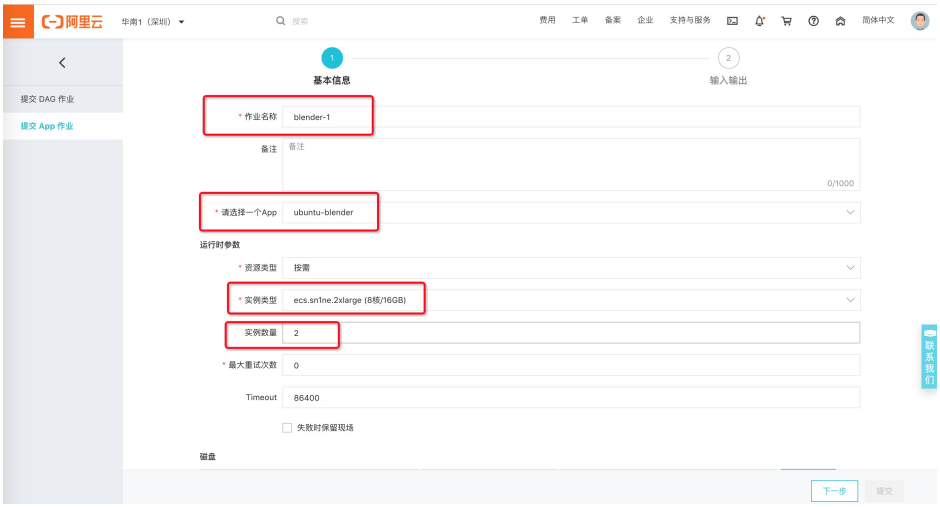
Blender 官网提供了好多 demo 文件： https://www.blender.org/download/demo-files/

本例子需要下载 2.79 版本（注意：要和镜像中安装的Blender版本相同。不同版本的可能渲染不出来）



素材下载后，解压得到目录： splash279/ 将整个目录上传到 oss://blender-demo/input/ 下面，即：oss://blender-demo/input/splash279/。

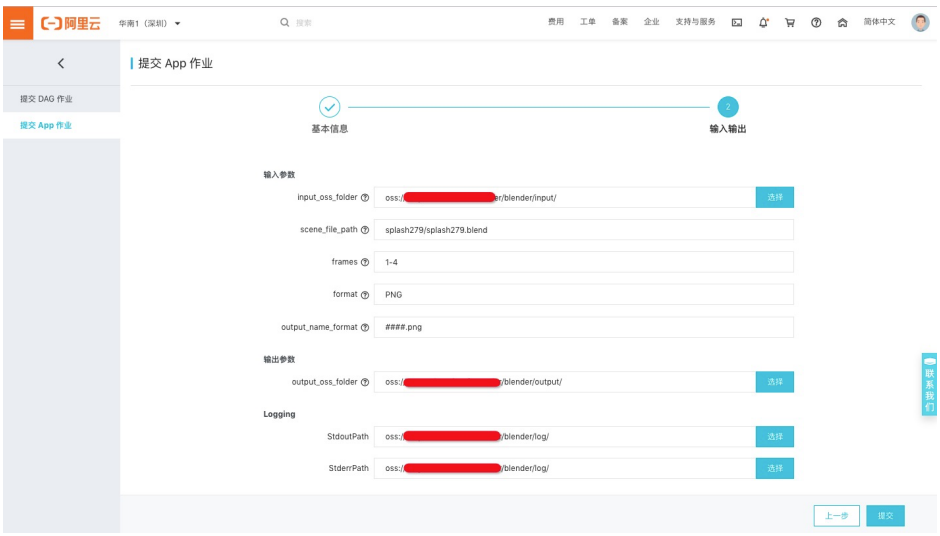
(2) 开始提交作业



- 实例类型要选大一点的，比如: 8核16GB。
- 实例数量本例子填 2 个。

(3) 参数配置

注意: 实际操作时，需要将例子中的 bucket 名称修改为您自己创建的真实的bucket名称。



输入:

参数	值	说明
input_oss_folder	oss://blender-demo/input/	场景文件所在OSS目录
scene_file_path	splash279/splash279.blend	场景文件名
frames	1-4	渲染1到4帧
format	PNG	渲染输出格式，默认即可
output_name_format	####.png	渲染输出文件名，默认即可

- scripts_oss_folder 设置了默认值，且不允许覆盖，可以不用填。

输出:

参数	值	说明
input_oss_folder	oss://blender-demo/output/	输出OSS目录, 渲染结果图片将保存到此目录的 result/ 子目录下

Loggin(日志目录配置):

参数	值	说明
StdoutPath	oss://blender-demo/log/	stdout日志输出到此
StderrPath	oss://blender-demo/log/	stderr日志输出到此

填好后点击提交即可。

5. 查看作业状态和结果

(1) 查看作业状态

(2) 查看结果

oss://blender-demo/output/result/

(3) 渲染时长和实例规格参考

实例规格	节点数	渲染帧数	时长
ecs.sn1ne.2xlarge(8核16GB)	2	1-4	9-12分钟
ecs.sn1ne.4xlarge (16核/32GB)	2	1-4	4-6分钟

6. 附录

分帧器代码(python):

framer.py:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
import os
import math
import sys
import re

NOTHING_TO_DO = 'Nothing to do, exit'

def _calcRange(a,b, id, step):
start = min(id * step + a, b)
end = min((id+1) * step + a-1, b)
return (start, end)

def _parseContinuedFrames(render_frames, total_nodes, id=None, return_type='list'):
'''
解析连续帧, 如: 1-10
'''
[a,b]=render_frames.split('-')
a=int(a)
b=int(b)
#print(a,b)
step = int(math.ceil((b-a+1)*1.0/total_nodes))

#print('step:', step)
mod = (b-a+1) % total_nodes
#print('mod:', mod)

if mod==0 or id < mod:
(start, end) = _calcRange(a,b, id, step)
#print('--->',start, end)
return (start, end) if return_type!='list' else range(start, end+1)
else:
a1 = step * mod + a
#print('less', a1, b, id)
(start, end) = _calcRange(a1 ,b, id-mod, step-1)

#print('--->',start, end)
return (start, end) if return_type!='list' else range(start, end+1)

def _parseIntermittentFrames(render_frames, total_nodes, id=None):
'''
解析不连续帧, 如: 1,3,8-10,21
'''
a1=render_frames.split(',')
a2=[]
for n in a1:
a=n.split('-')
a2.append(range(int(a[0]),int(a[1])+1) if len(a)==2 else [int(a[0])])

a3=[]
for n in a2:
a3=a3+n
#print('a3',a3)

step = int(math.ceil(len(a3)*1.0/total_nodes))
#print('step',step)
```

```
mod = len(a3) % total_nodes
#print('mod:', mod)

if mod==0 or id < mod:
(start, end) = _calcRange(0, len(a3)-1, id, step)
#print(start, end)
a4= a3[start: end+1]
#print('--->', a4)
return a4
else:
#print('less', step * mod , len(a3)-1, id)
(start, end) = _calcRange( step * mod ,len(a3)-1, id-mod, step-1)
if start > len(a3)-1:
print(NOTHING_TO_DO)
sys.exit(0)
#print(start, end)
a4= a3[start: end+1]
#print('--->', a4)
return a4


def parseFrames(render_frames, return_type='list', id=None, total_nodes=None):
'''
@param render_frames {string}: 需要渲染的总帧数列表范围，可以用"-"表示范围，不连续的帧可以使用","隔开, 如: 1,3,5-10
@param return_type {string}: 取值范围[list,range]。 list样例: [1,2,3], range样例: (1,3)。
注意: render_frames包含","时有效，强制为list。
@param id, 节点ID，从0开始。 正式环境不要填写，将从环境变量 BATCH_COMPUTE_DAG_INSTANCE_ID 中取得。
@param total_nodes, 总共的节点个数。正式环境不要填写，将从环境变量 BATCH_COMPUTE_DAG_INSTANCE_COUNT 中取得。
'''

if id==None:
id=os.environ['BATCH_COMPUTE_DAG_INSTANCE_ID']
if type(id)==str:
id = int(id)

if total_nodes==None:
total_nodes = os.environ['BATCH_COMPUTE_DAG_INSTANCE_COUNT']
if type(total_nodes)==str:
total_nodes = int(total_nodes)

if re.match(r'^(\d+)\-(\d+)$',render_frames):
# 1-2
# continued frames
return _parseContinuedFrames(render_frames, total_nodes, id, return_type)

else:
# intermittent frames
return _parseIntermittentFrames(render_frames, total_nodes, id)
```

3ds Max DAG作业最佳实践

1. 准备工作

1.1. 选择区域

所有阿里云服务都需要使用相同的地域。

1.2. 开通服务

- 开通批量计算服务（BatchCompute）；
- 开通对象存储服务（OSS）。

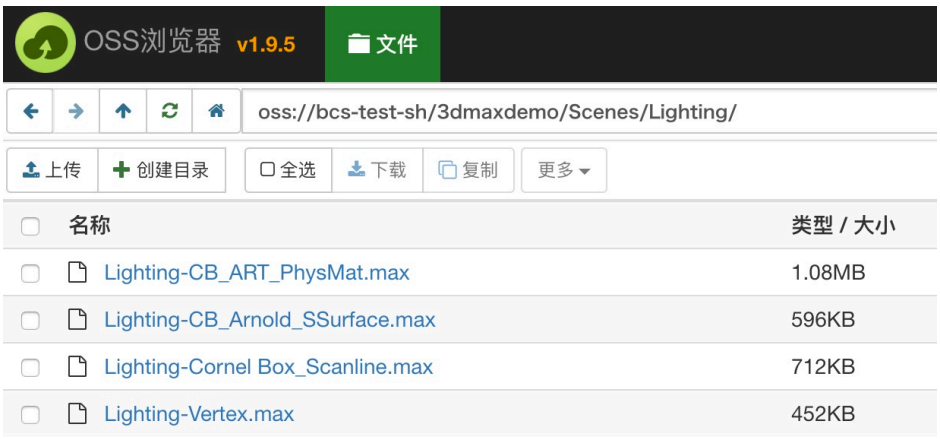
1.3. 制作镜像

制作镜像具体步骤请参考**集群镜像**, 请严格按文档的步骤创建镜像。镜像制作完成后，通过以下方式可以获取到对应的镜像信息。

1.4. 上传素材

可以下载 3ds Max 官方提供的**免费素材包**进行测试。

通过 OSSBrowser 工具将渲染素材到指定的 OSS bucket 中，如下图：



1.5. 安装批量计算 SDK

在需要提交作业的机器上，安装批量计算 SDK 库；已经安装请忽略。Linux 安装执行如下命令；Windows 平台请参考文档。

```
pip install batchcompute
```

2. 编写work脚本

work.py

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
import os
import math
import sys
import re
import argparse
NOTHING_TO_DO = 'Nothing to do, exit'

def _calcRange(a,b, id, step):
start = min(id * step + a, b)
end = min((id+1) * step + a-1, b)
return (start, end)

def _parseContinuedFrames(render_frames, total_nodes, id=None, return_type='list'):
'''
解析连续帧, 如: 1-10
'''
[a,b]=render_frames.split('-')
a=int(a)
b=int(b)
#print(a,b)
step = int(math.ceil((b-a+1)*1.0/total_nodes))
#print('step:', step)
```

```
mod = (b-a+1) % total_nodes
#print('mod:', mod)
if mod==0 or id < mod:
(start, end) = _calcRange(a,b, id, step)
#print('--->',start, end)
return (start, end) if return_type!='list' else range(start, end+1)
else:
a1 = step * mod + a
#print('less', a1, b, id)
(start, end) = _calcRange(a1 ,b, id-mod, step-1)
#print('--->',start, end)
return (start, end) if return_type!='list' else range(start, end+1)

def _parseIntermittentFrames(render_frames, total_nodes, id=None):
'''
解析不连续帧, 如: 1,3,8-10,21
'''
a1=render_frames.split(',')
a2=[]
for n in a1:
a=n.split('-')
a2.append(range(int(a[0]),int(a[1])+1) if len(a)==2 else [int(a[0])])
a3=[]
for n in a2:
a3=a3+n
#print('a3',a3)
step = int(math.ceil(len(a3)*1.0/total_nodes))
#print('step',step)
mod = len(a3) % total_nodes
#print('mod:', mod)
if mod==0 or id < mod:
(start, end) = _calcRange(0, len(a3)-1, id, step)
#print(start, end)
a4= a3[start: end+1]
#print('--->', a4)
return a4
else:
#print('less', step * mod , len(a3)-1, id)
(start, end) = _calcRange( step * mod ,len(a3)-1, id-mod, step-1)
if start > len(a3)-1:
print(NOTHING_TO_DO)
sys.exit(0)
#print(start, end)
a4= a3[start: end+1]
#print('--->', a4)
return a4
def parseFrames(render_frames, return_type='list', id=None, total_nodes=None):
'''
@param render_frames {string}: 需要渲染的总帧数列表范围，可以用"-"表示范围，不连续的帧可以使用","隔开, 如: 1,3,5-
10
@param return_type {string}: 取值范围[list,range]。 list样例: [1,2,3], range样例: (1,3)。
注意: render_frames包含","时有效，强制为list。
@param id, 节点ID，从0开始。 正式环境不要填写，将从环境变量 BATCH_COMPUTE_DAG_INSTANCE_ID 中取得。
@param total_nodes, 总共的节点个数。正式环境不要填写，将从环境变量 BATCH_COMPUTE_DAG_INSTANCE_COUNT
中取得。
'''
```

```
if id==None:
id=os.environ['BATCH_COMPUTE_DAG_INSTANCE_ID']
if type(id)==str:
id = int(id)
if total_nodes==None:
total_nodes = os.environ['BATCH_COMPUTE_DAG_INSTANCE_COUNT']
if type(total_nodes)==str:
total_nodes = int(total_nodes)
if re.match(r'^(\d+)\-(\d+)$',render_frames):
# 1-2
# continued frames
return _parseContinuedFrames(render_frames, total_nodes, id, return_type)
else:
# intermittent frames
return _parseIntermittentFrames(render_frames, total_nodes, id)

if __name__ == "__main__":
parser = argparse.ArgumentParser(
formatter_class = argparse.ArgumentDefaultsHelpFormatter,
description = 'python scripyt for 3dmax dag job',
usage='render3Dmax.py <positional argument> [<args>]',
)

parser.add_argument('-s', '--scene_file', action='store', type=str, required=True, help = 'the name of the file with
.max subffix .')
parser.add_argument('-i', '--input', action='store', type=str, required=True, help = 'the oss dir of the scene_file, eg:
xxx.max.')
parser.add_argument('-o', '--output', action='store', type=str, required=True, help = 'the oss of dir the result file to
upload .')
parser.add_argument('-f', '--frames', action='store', type=str, required=True, help = 'the frames to be renderd, eg:
"1-10".')
parser.add_argument('-t', '--retType', action='store', type=str, default="test.jpg", help = 'the tye of the render
result,eg. xxx.jpg/xxx.png.')
args = parser.parse_args()

frames=parseFrames(args.frames)
framestr='-'.join(map(lambda x:str(x), frames))

s = "cd \"C:\\Program Files\\Autodesk\\3ds Max 2018\\\" && "
s +='3dsmaxcmd.exe -o="%s%s" -frames=%s "%s\\%s"' % (args.output, args.retType, framestr, args.input,
args.scene_file)
print("exec: %s" % s)

rc = os.system(s)
sys.exit(rc>>8)
```

注意：

- work.py 只需要被上传到 OSS bucket中不需要手动执行；各项参数通过作业提交脚本进行传递；
- work.py 的112 行需要根据镜像制作过程中 3ds MAX 的位置做对应替换；
- work.py 的 scene_file 参数表示场景文件；如 Lighting-CB_Arnold_SSurface.max；
- work.py 的 input 参数表示素材映射到 VM 中的位置，如: D；
- work.py 的 output 参数表示渲染结果输出的本地路径；如 C:\tmp\；
- work.py 的 frames 参数表示渲染的帧数，如: 1；

- work.py 的 retType 参数表示素材映射到 VM 中的位置，如: test.jpg；渲染结束后如果是多帧，则每帧的名称为test000.jpg，test001.jpg等。

3. 编写作业提交脚本

test.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from batchcompute import Client, ClientError
from batchcompute.resources import (
ClusterDescription, GroupDescription, Configs, Networks, VPC,
JobDescription, TaskDescription, DAG,Mounts,
AutoCluster,Disks,Notification,
)
import time
import argparse

from batchcompute import CN_SHANGHAI as REGION #需要根据 region 做适配
access_key_id = "xxxx" # your access key id
access_key_secret = "xxxx" # your access key secret

instance_type = "ecs.g5.4xlarge" # instance type #需要根据 业务需要 做适配

image_id = "m-xxx"

workossPath = "oss://xxxxx/work/work.py"

client = Client(REGION, access_key_id, access_key_secret)

def getAutoClusterDesc(InstanceCount):
auto_desc = AutoCluster()

auto_desc.ECSImageId = image_id

#任务失败保留环境，程序调试阶段设置。环境保留费用会继续产生请注意及时手动清除环境任务失败保留环境，
# 程序调试阶段设置。环境保留费用会继续产生请注意及时手动清除环境
auto_desc.ReserveOnFail = False

# 实例规格
auto_desc.InstanceType = instance_type
```

```
#case3 按量
auto_desc.ResourceType = "OnDemand"

#Configs
configs = Configs()
#Configs.Networks
networks = Networks()
vpc = VPC()

# CidrBlock和VpcId 都传入，必须保证VpcId的CidrBlock 和传入的CidrBlock保持一致
vpc.CidrBlock = '172.26.0.0/16'
# vpc.VpcId = "vpc-8vbfxdyhx9p2flummuwmq"

networks.VPC = vpc
configs.Networks = networks

# 设置系统盘type(cloud_efficiency/cloud_ssd)以及size(单位GB)
configs.add_system_disk(size=40, type_='cloud_efficiency')

#设置数据盘type(必须和系统盘type保持一致) size(单位GB) 挂载点
# case1 linux环境
# configs.add_data_disk(size=40, type_='cloud_efficiency', mount_point='/path/to/mount/')

# 设置节点个数
configs.InstanceCount = InstanceCount
auto_desc.Configs = configs
return auto_desc

def getTaskDesc(inputOssPath, outputossPath, scene_file, frames, retType, clusterId, InstanceCount):
taskDesc = TaskDescription()

timestamp = time.strftime("%Y_%m_%d_%H_%M_%S", time.localtime())
inputLocalPath = "D:"
outputLocalPath = "C:\\\\tmp\\\\" + timestamp + "\\\\"
outputossBase = outputossPath + timestamp + "/"
stdoutOssPath = outputossBase + "stdout/" #your stdout oss path
stderrOssPath = outputossBase + "stderr/" #your stderr oss path
outputossret = outputossBase + "ret/"

taskDesc.InputMapping = {inputOssPath: inputLocalPath}
taskDesc.OutputMapping = {outputLocalPath: outputossret}

taskDesc.Parameters.InputMappingConfig.Lock = True

# 设置程序的标准输出地址，程序中的print打印会实时上传到指定的oss地址
taskDesc.Parameters.StdoutRedirectPath = stdoutOssPath

# 设置程序的标准错误输出地址，程序抛出的异常错误会实时上传到指定的oss地址
taskDesc.Parameters.StderrRedirectPath = stderrOssPath

#触发程序运行的命令行
# PackagePath存放commandLine中的可执行文件或者二进制包
taskDesc.Parameters.Command.PackagePath = workossPath
taskDesc.Parameters.Command.CommandLine = "python work.py -i %s -o %s -s %s -f %s -t %s" % (inputLocalPath,
outputLocalPath, scene_file, frames, retType)
```

```
# 设置任务的超时时间
taskDesc.Timeout = 86400

# 设置任务所需实例个数
taskDesc.InstanceCount = InstanceCount

# 设置任务失败后重试次数
taskDesc.MaxRetryCount = 3

if clusterId:
# 采用固定集群提交作业
taskDesc.ClusterId = clusterId
else:
#采用auto集群提交作业
taskDesc.AutoCluster = getAutoClusterDesc(InstanceCount)

return taskDesc


def getDagJobDesc(inputOssPath, outputossPath, scene_file, frames, retType, clusterId = None, instanceNum = 1):
job_desc = JobDescription()
dag_desc = DAG()

job_desc.Name = "testBatch"
job_desc.Description = "test 3dMAX job"
job_desc.Priority = 1

# 任务失败
job_desc.JobFailOnInstanceFail = False

# 作业运行成功后户自动会被立即释放掉
job_desc.AutoRelease = False
job_desc.Type = "DAG"

render = getTaskDesc(inputOssPath, outputossPath, scene_file, frames, retType, clusterId, instanceNum)

# 添加任务
dag_desc.add_task('render', render)

job_desc.DAG = dag_desc
return job_desc

if __name__ == "__main__":
parser = argparse.ArgumentParser(
formatter_class = argparse.ArgumentDefaultsHelpFormatter,
description = 'python scripyt for 3dmax dag job',
usage='render3Dmax.py <positional argument> [<args>]',
)

parser.add_argument('-n','--instanceNum', action='store',type = int, default = 1,help = 'the parell instance num .')
parser.add_argument('-s', '--scene_file', action='store', type=str, required=True, help = 'the name of the file with
.max subffix .')
parser.add_argument('-i', '--inputoss', action='store', type=str, required=True, help = 'the oss dir of the scene_file,
eg: xxx.max.')
parser.add_argument('-o', '--outputoss', action='store', type=str, required=True, help = 'the oss of dir the result file
to upload .')
```

```
parser.add_argument('-f', '--frames', action='store', type=str, required=True, help = 'the frames to be renderd, eg: "1-10".')
parser.add_argument('-t', '--retType', action='store', type=str, default = "test.jpg", help = 'the tye of the render result,eg. xxx.jpg/xxx.png.')
parser.add_argument('-c', '--clusterId', action='store', type=str, default=None, help = 'the clusterId to be render .')

args = parser.parse_args()

try:
job_desc = getDagJobDesc(args.inputoss, args.outputoss, args.scene_file, args.frames,args.retType, args.clusterId, args.instanceNum)
# print job_desc
job_id = client.create_job(job_desc).Id
print('job created: %s' % job_id)
except ClientError,e:
print (e.get_status_code(), e.get_code(), e.get_requestid(), e.get_msg())
```

注意：

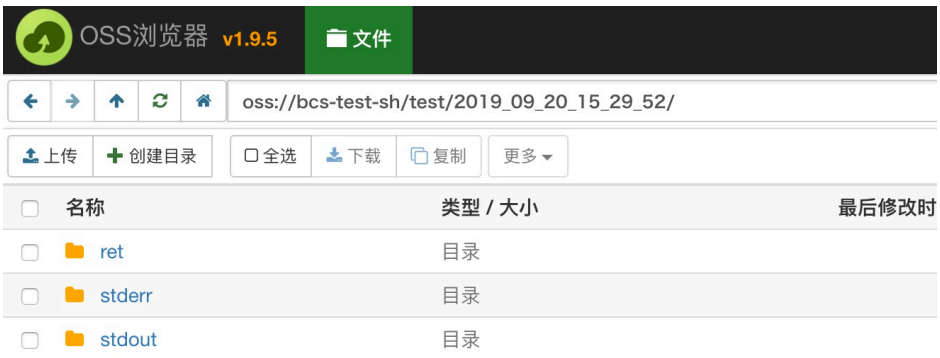
- 代码中 12~20 行 需要根据做适配，如 AK 信息需要填写账号对应的AK信息；镜像Id 就是1.3 中制作的镜像 Id；workosspath 是步骤 2 work.py 在oss上的位置；
- 参数 instanceNum 表示 当前渲染作业需要几个节点参与，默认是1个节点；若是设置为多个节点，work.py 会自动做均分；
- 参数 scene_file 表示需要渲染的场景文件，传给 work.py；
- 参数 inputoss 表示 素材上传到 OSS 上的位置，也即1.4 中的 OSS 位置；
- 参数 outputoss 表示最终结果上传到 Oss 上的位置；
- 参数 frames 表示需要渲染的场景文件的帧数，传给 work.py；3ds MAX 不支持隔帧渲染，只能是连续帧，如1-10；
- 参数 retType 表示需要渲染渲染结果名称，传给 work.py，默认是 test.jpg，则最终得到test000.jpg
- 参数 clusterId 表示采用固定集群做渲染时，固定集群的Id。

4. 提交作业

根据以上示例文档，执行以下命令：

```
python test.py -s Lighting-CB_Arnold_SSurface.max -i oss://bcs-test-sh/3dmaxdemo/Scenes/Lighting/ -o oss://bcs-test-sh/test/ -f 1-1 -t 123.jpg
```

示例运行结果：



云渲染管理系统

简介

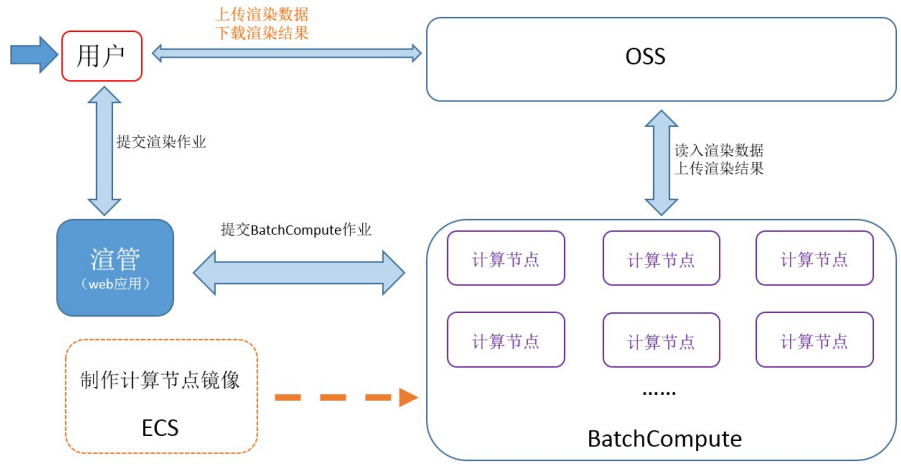
云渲染管理系统（Render Manager 简称渲管）是一个开源的 web 应用，可以帮助用户轻松搭建阿里云上的私有渲染系统，直接调用海量计算资源，一键管控集群规模，在加速渲染任务的同时省去自建集群的烦恼。

渲管建立在阿里云 BatchCompute 、OSS 和 ECS 的三个云产品基础之上的。详细介绍请参考官网，在使用渲管前，请确保已开通此三产品。

- BatchCompute 是阿里云上的批量计算服务，可以帮助用户进行大规模并行计算。
- OSS 是阿里云上的对象存储服务，可以存储海量数据。

- ECS 是阿里云上的云服务器，极易运维和操作,可以方便的制作系统镜像。

渲管与这三个云产品的关系如下图



1. 使用流程

A) 制作计算节点镜像

根据所要使用的区域，创建 ECS 按量云服务器，在云服务器中安装所需的渲染软件；保存为自定义镜像，并将镜像共享给账号1190847048572539，详见计算节点 镜像制作 章节。

B) 上传数据到OSS

将渲染所需要的数据上传到对应区域的OSS，并保持上传前的目录结构。

C) 启动渲管

在 ECS 控制台创建实例（短期使用，选择按量即可），镜像选择镜像市场中的rendermanager（也可以使用渲管安装包进行部署，详见 操作手册 部署章节）。

D) 配置渲管

登录渲管页面 https://ip/rm/login, 配置完基本信息后（AccessKeys 和 OSS bucket），在镜像管理页中添加上面制作的计算节点镜像 ID，并对该计算节点镜像配置渲染命令行。

E) 创建项目

在渲管的项目管理页面创建项目，指定 OSS 的数据映射规则（也称 OSS 挂载，在计算节点启动的时候，OSS 上的数据会被挂载到节点的本地路径），选择计算节点镜像 ID，OSS 的输出路径（用于保存渲染结果），计算节点中的临时输出路径。

F) 集群的创建和管理

在集群管理页面可以按需创建集群，指定计算节点使用的镜像 ID，节点类型和节点数量等信息。

G) 提交渲染作业

在项目页里提交渲染作业，要指定目的集群、渲染的帧范围以及节点数量等信息。提交完作业后，可实时查看渲染日志以及节点 CPU 使用率等信息。

> 使用 AutoCluter 时，BatchCompute 将按作业的规模自动生成集群，使用 AutoCluster 需要指定计算节点类型等配置。

快速开始

BatchCompute 提供了测试用的计算节点镜像（windows server 2008，ID：m-wz9du0xaa1pag4ylwzsu），它预装了 blender 渲染软件。使用 blender 制作一个小场景的 演示视频 已上传 OSS（测试时，需下载并上传到您的 OSS bucket）。

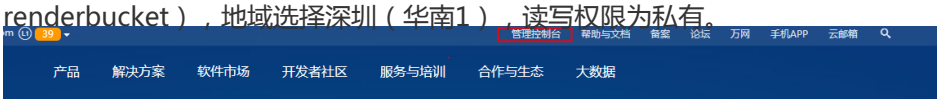
> 实际生产时，请根据需求制作合适的计算节点镜像。

1. 准备工作

- 注册阿里云账号并开通 OSS、ECS 和 BatchCompute 服务。
- 创建AccessKey。账号信息->AccessKeys->创建 Access Key，记录 Access Key 信息。

2. 渲染示例

A) 创建 OSS bucket阿里云官网->管理控制台->对象存储 OSS->创建 bucket（例如，名字为 renderbucket），地域选择深圳（华南1），读写权限为私有。



3. 获取blender场景并上传到您的 OSS bucket

- 在浏览器输入 http://openrm.oss-cn-qingdao.aliyuncs.com/blender/monkey/cube.blend 。
- 下载示例场景文件（BatchCompute 提供的测试场景），在 OSS 控制台创建目录结构 blender/monkey，然后在该目录下上传文件，文件路径为 oss://renderbucket/blender/monkey/cube.blend。

4. 启动rendermanager

A) 阿里云官网->管理控制台->云服务器 ECS->创建实例

- 选择按量付费，然后在镜像市场应用开发分类中搜索 rendermanager 镜像，使用 rendermanager 镜像并按下图配置购买，可适当提高带宽。

  使用按量付费要求用户账户至少有 100 块金额，对于地域没有要求，看 ECS 实际售卖库存情况而定。

B) 购买后，点击进入管理控制台，在实例列表中可看到刚才启动的云主机（创建会有延迟，请刷新几次）。

5. 登入渲管页面

在本地浏览器输入 https://ecs_instance_ip/rm/login，ecs_instance_ip 为 ECS 实例的公网 IP（由于使用了 https，请在浏览器页面授权信任）。初始账号密码为：

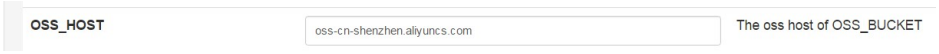
- rm_admin
- rm_admin@123

> 生产系统，请一定更改账号和密码。

6. 配置渲管

A) 登录后，点击右上角的配置可进入配置页面，填入 SECURITY_ID，SECURITY_KEY， OSS_BUCEKET 三个字段的值，SECURITY_ID 和 SECURITY_KEY 即上面准备工作中获取的 AccessKey 信息。

B) 设置 OSS_HOST 为 oss-cn-shenzhen.aliyuncs.com；REGION 的选择主要和计算节点的镜像归属有关，必须和计算节点镜像归属 REGION 保持一致；本例采用的官方计算节点镜像（该镜像部署在深圳 REGION）所以此处设置在深圳 REGION 。

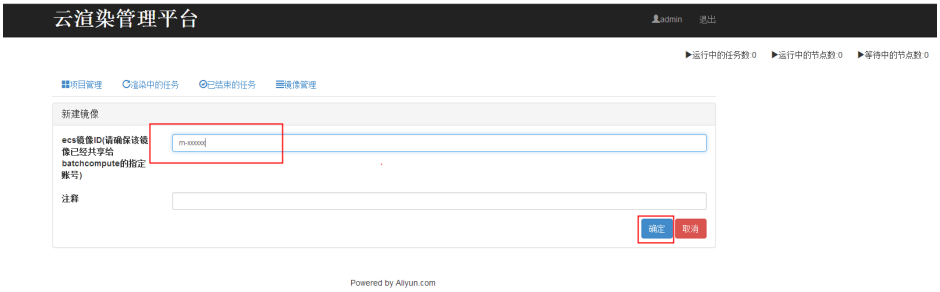


C) 设置 BATCHCOMPUTE_REGION 为 cn-shenzhen；设置深圳 REGION 原因同上。

D) 点击保存。

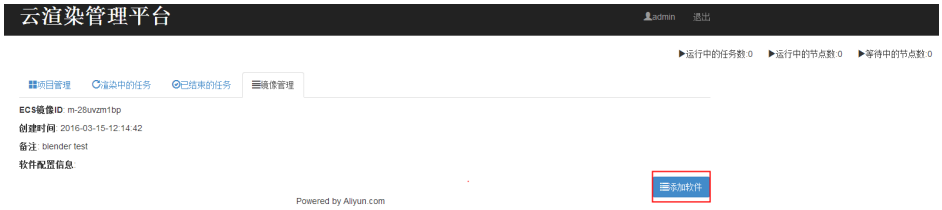
7. 添加计算节点镜像

镜像管理->添加计算节点镜像，ECS 镜像 ID：m-wz9du0xaa1pag4ylwzsu（BatchCompute 提供的公用计算节点镜像，实际生产，需要用户制作所需要的计算节点镜像，具体制作流程请参考 操作手册）。



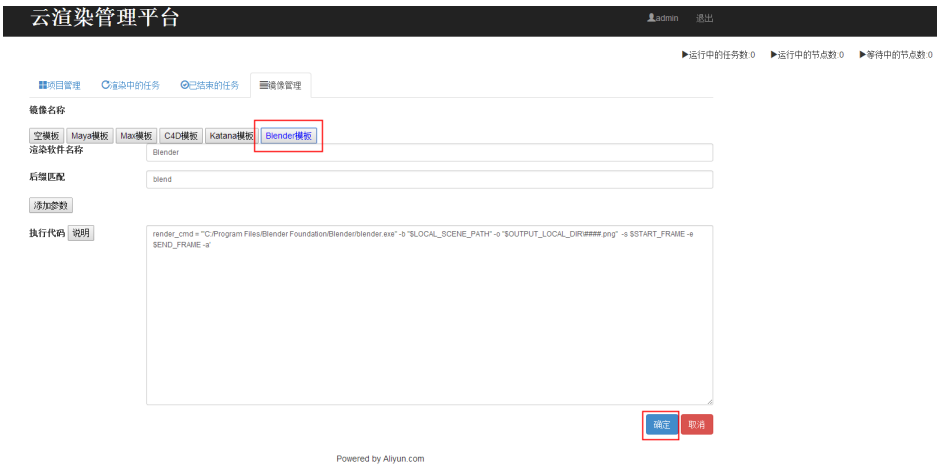
8. 配置渲染软件信息

A) 镜像管理->软件配置。



B) 添加软件。

C) 选择 blender 模板并确定，执行 render_cmd 渲染命令。

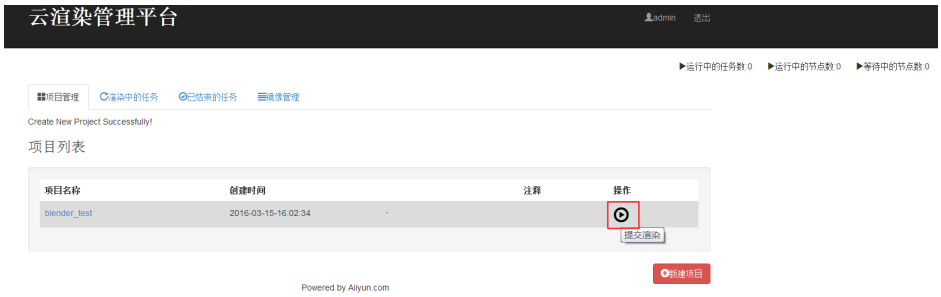


9. 创建项目

A) 项目管理->新建项目。 B) 填入需要映射的 OSS 路径数量(本例只映射一个OSS路径)，并点击确认。

C) 填入项目名称: blender_test。D) 镜像选择上面创建的镜像。E) OSS 映射中的选择/输入路径为 /renderbucket/blender/。F) OSS 映射的目的地为盘符 G: (本例中使用的镜像系统为 Windows2008 server)。G) OSS 输出目录填写为 /renderbucket/rm_test/output/。H) 虚拟机中的输出目录填写为 C:\render_output\，该路径用于渲染节点中临时存放渲染结果，并且该目录里的渲染结果会被传输到 OSS 上输出目录里。I) 确定提交。

10. 提交渲染任务



A) 项目管理->提交渲染。

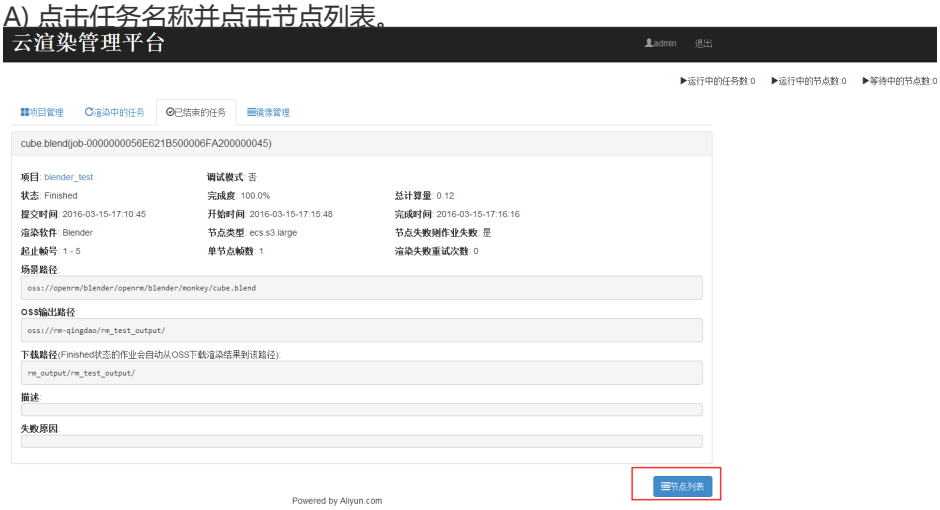
B) 选择场景所在的 OSS 路径前缀。

C) 选择项目根目录， 直到场景文件cube.blend，选中 monkey 文件夹；可以看到页面下部出现场景选择，勾选场景，选择渲染软件，填入渲染起止帧 1~5，并点击提交渲染按钮。

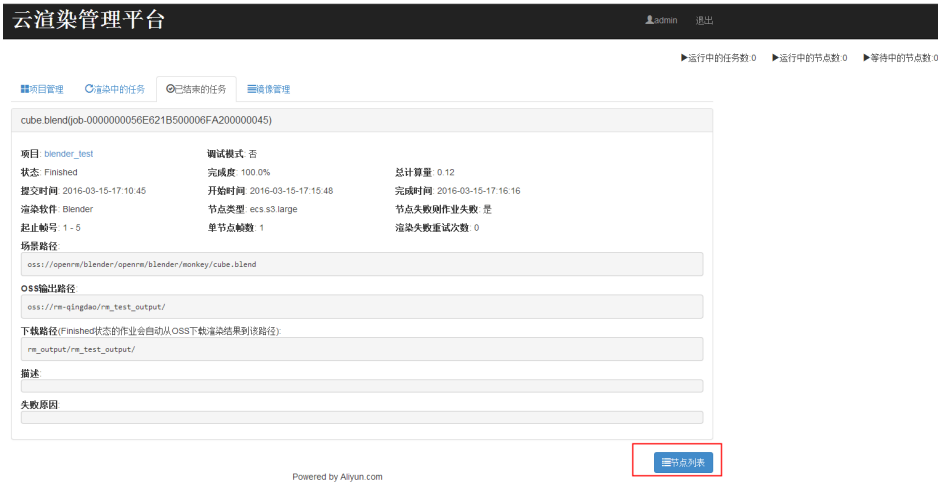
D) 选择渲染中的任务，可查看刚才提交的作业。

11. 查看渲染日志

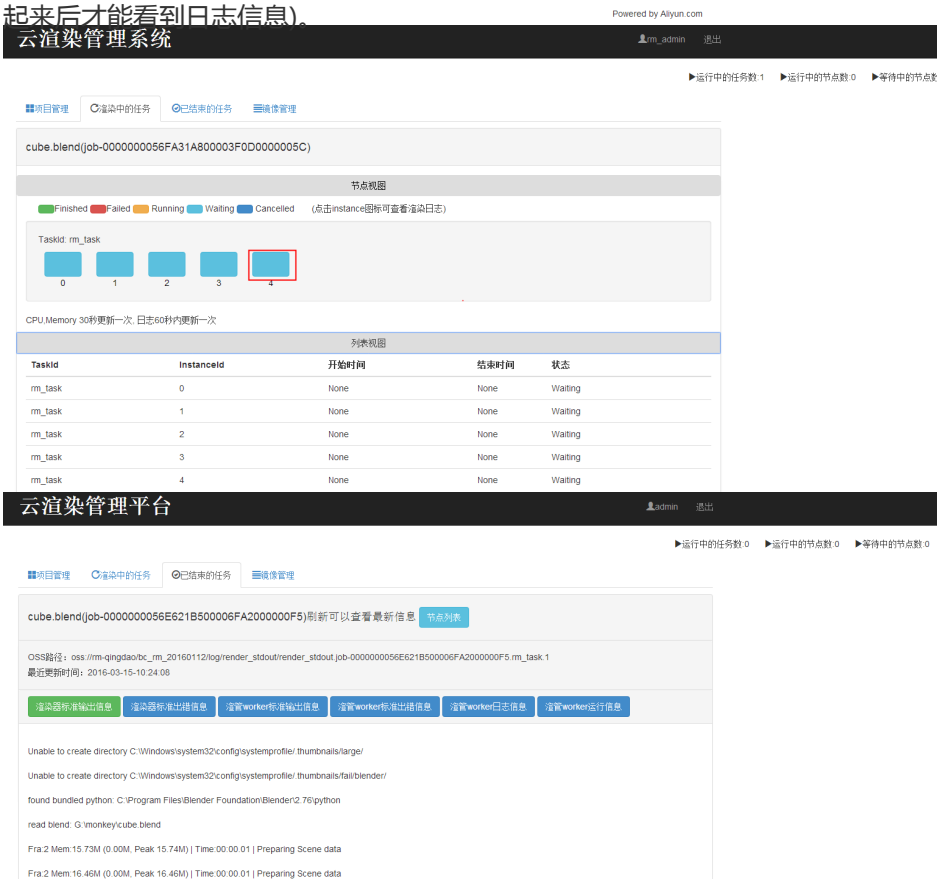
A) 点击任务名称并点击节点列表。



B) 点击想查看的节点，可以看到渲染器和渲管 worker 的各种日志、标准输出以及标准出错信息(计算节点运行



起来后才能看到日志信息)。

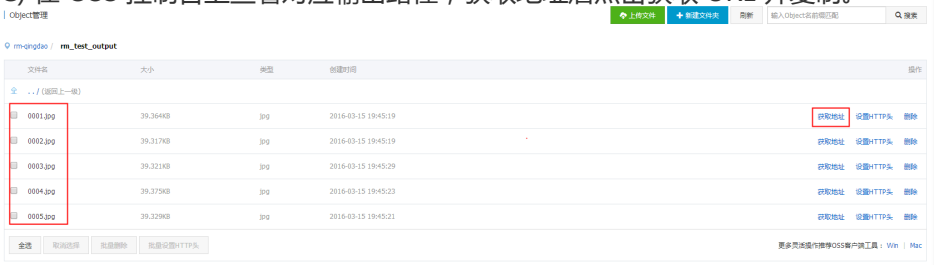


12. 查看渲染结果

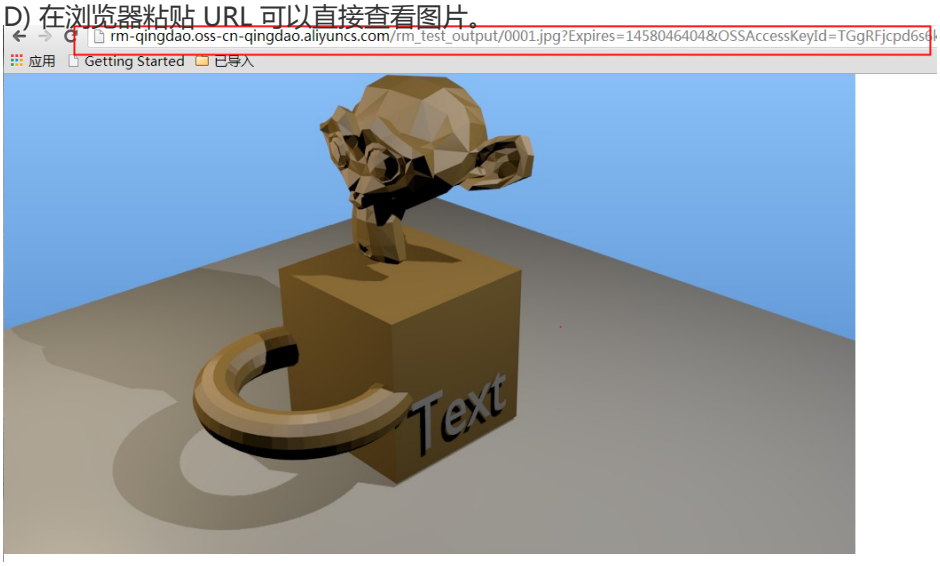
A) 等待作业结束后，在已结束的任务中可以可以看到任务状态为 Finished。

B) 点击任务名称，可以查看 OSS 上的输出路径。

C) 在 OSS 控制台上查看对应输出路径，获取地址后点击获取 URL 并复制。



D) 在浏览器粘贴 URL，可以直接查看图片。



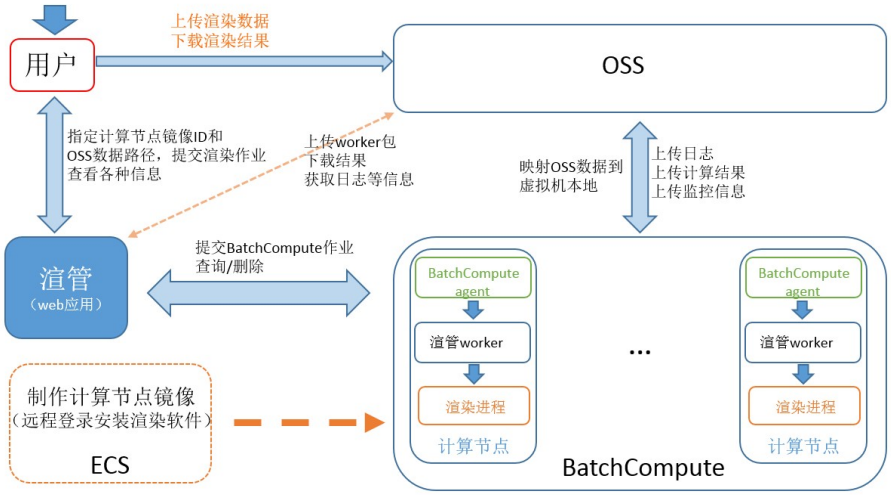
E) 恭喜您已跑通云上的 Blender 渲染测试。

操作手册

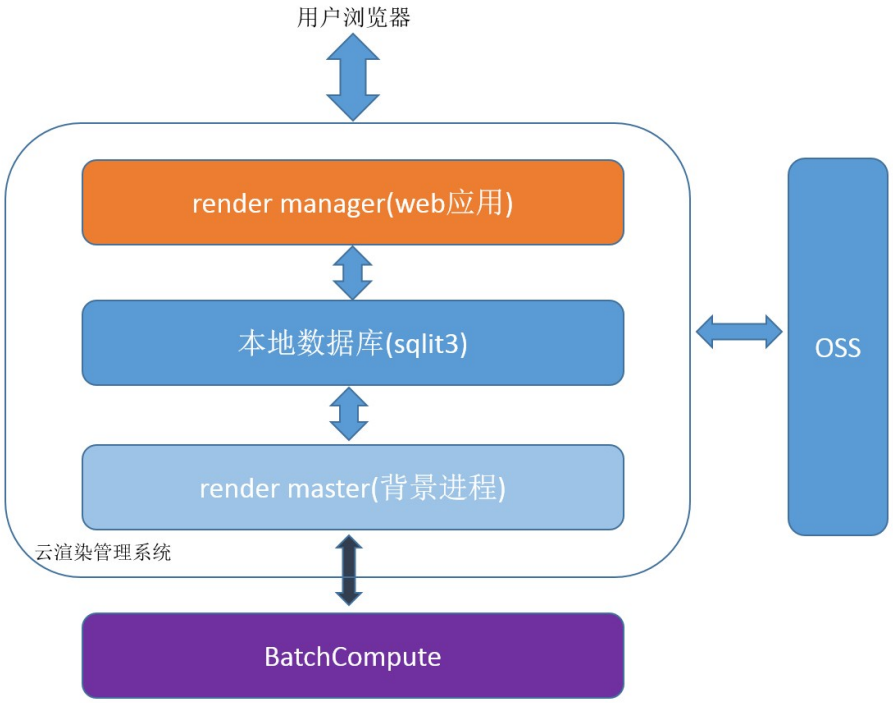
1. 渲管系统结构

A) 渲管与各云产品的详细关系

渲管与各云产品的依赖如下图所示。



B) 渲管系统内部结构



渲管系统由如下 3 部分组成

：

- render manager: 基于 flask 框架开发web 应用，主要负责和用户进行人机交互，接收用户请求。
- render master：后台背景进程，根据人机交互的结果进行作业提交以调度。

- 本地数据库：主要存放用户提交的渲管请求，待渲管任务结束后自动删除该信息。

2. 渲管的部署

在阿里云云市场有已安装了渲管的 ECS 镜像免费售卖，在启动 ECS 实例时，将镜像指定为镜像市场中的 rendermanager，启动即可使用。

A) 获取渲管镜像

官方渲管镜像：RenderManager 镜像，创建 ECS 实例时，选择镜像市场，直接搜索以上关键字即可获取。自定义渲管镜像：基础镜像建议采用 Ubuntu 14.04 64 位，按照以下步骤安装渲管系统。

```
# 安装 flask
sudo apt-get install python-flask -y
# 安装 uwsgi
sudo apt-get install uwsgi uwsgi-plugin-python -y
# 安装 nginx
sudo apt-get install nginx –y
# 修改 nginx 配置，在 http 模块里添加新的 server
#
# server {
# listen 1314; #listen port
# server_name localchost;
# location / {
# include uwsgi_params;
# uwsgi_pass 0.0.0.0:8818;#this must be same app_config.xml
# }
# }
#
vi /etc/nginx/nginx.conf
# 启动 nginx 或重启
nginx
# 获取最新版渲管
wget http://openrm.oss-cn-qingdao.aliyuncs.com/render_manager_release/latest/rm.tar.gz
# 解压
tar –xf rm.tar.gz
# x.x.x 为版本号
cd rm-x.x.x
# 指定安装目录部署
python deploy.py /root/rm_install/
# 启动
cd /root/rm_install/rm_install_s && python rm_cmd.py start
# 登陆渲管 http://installed_machine_ip:1314/rm/login
# 初始账号： rm_admin 密码： rm_admin@123
# 若监听在公网，建议采用https
```

B) 开通 ECS 实例

请指定某 ECS 实例部署渲管系统，配置参数，请参考创建 Linux 实例

- 公网 IP 地址选择分配。

- 镜像市场: RenderManager 或者自定义镜像
- 设置密码

3. 渲管系统升级

页面右上角的版本信息中可以查看是否有可升级的新版本，第一次使用渲管前，建议升级到最新版本后再使用渲管（每次只能升级到下一版本，所以升级后请查看是否已是最新版本）。

4. 渲管系统配置

配置页面里有渲管系统的各种系统设置。第一次使用渲管时，必须设置SECURITY_ID，SECURITY_KEY，OSS_BUCKET 三个值，不然渲管无法使用。

- SECURITY_ID 和 SECURITY_KEY 即阿里云账号的 AccessKeys 信息，可以在阿里云官网控制台创建。
- OSS_BUCKET 可以在 OSS 的控制台创建，用于存储渲管自身的 worker 包已经渲染数据。

渲管默认使用青岛（华北1）区域，如果使用其他区域的 BatchCompute，请修改配置中的 OSS_HOST(OSS_BUCKET 必须与 OSS_HOST 属于同一个region)与 BATCHCOMPUTE_REGION，每个 REGION 的 OSS_HOST 也可以工单咨询获取。 区域的选择和计算节点的镜像区域保持一致，若计算节点镜像在深圳区域，则渲管的区域信息也必须是深圳，同时 OSS BUCKET 也必须是该 REGION 下的 BUCKET；若使用批量计算官方提供的计算节点镜像则需要选择深圳 REGION。

其他配置项，请参考页面上的说明。

5. OSS数据上传

提交渲染作业前，一定要将渲染用到的数据上传 OSS，在计算节点启动后再上传的数据将不能在计算节点中访问到。

由于 OSS 页面控制台上传数据有大小限制，所以上传数据建议使用 OSS 的 命令行工具（类 linux系统）、windows 客户端或者 MAC 客户端 。

参考 更多 OSS工具 。

6. 计算节点镜像制作

渲染客户如希望定制计算节点镜像，请参考：自定义镜像。

7. 计算节点镜像管理

A) 添加计算节点镜像

在镜像管理页面，可以添加计算节点镜像 ID。

B) 给计算节点镜像配置渲染软件信息

在添加完计算节点镜像 ID 后，在镜像信息页面可以点击添加软件并配置软件信息。

- 在配置软件信息时，需要填入渲染软件的名称，渲染文件的后缀（用于识别渲染文件）以及执行代码。
- 执行代码（要求 python 语法）会在渲管 worker 中执行，render_cmd 变量即渲染时的命令行，命令行应根据实际安装的渲染软件来填写，比如渲染软件的路径，以及一些参数。渲管中的模板只是个示例，实际使用需要微调。

渲管已经预定义了一些变量和函数，在执行代码中可以调用这些变量和函数，例如$CPU在执行期会被替换成实际的cpu核数，$START_FRAME在执行期会被替换成起始帧号。

如果想增加自定义参数，可以选择添加参数，添加的自定义参数会需要在提交作业时填入。关于所有的可用变量可在软件配置页面点击查看。

$OUTPUT_LOCAL_DIR这个变量即创建项目时配置的节点内临时输出路径，渲染的输出结果应该放在该路径下（大部分渲染器都支持在命令行中指定输出路径），在渲染结束后该目录下的数据会被传输到 OSS。

8. 项目管理

A) 项目创建

创建项目时需要指定 OSS 数据映射，计算节点镜像，虚拟机内的临时输出路径，OSS 输出路径。

i. 计算节点镜像

创建项目时选择的计算节点镜像（需要先在镜像管理页面添加计算节点镜像）是提交 AutoCluster 作业时使用的镜像，如果提交作业时指定了集群（在集群管理页面可以创建）则作业直接跑在所指定的集群中。

ii. OSS数据映射

OSS 数据映射（或者称 OSS 数据挂载），可以将 OSS 上的数据映射到计算节点的本地路径（windows 是盘符），一个作业中的所有计算节点可以共享访问到相同的数据。OSS 数据挂载有如下功能或限制：

1. 映射的目的路径必须根据计算节点镜像实际的操作系统类型进行填写，否则会导致挂载失败，windows 只能映射到盘符（例 G:），linux 必须是绝对路径。
2. 可共享读取访问 OSS 上的数据。
3. 不支持修改 OSS 上已存在的文件和文件夹名称。
4. 选择 WriteSupport 后，支持本地（挂载路径下）文件和文件夹的创建，以及新建文件的修改。

5. 挂载的本地路径里的改动只是本计算节点可见，不会同步到 OSS。
6. 在 Windows 系统中，在挂载时刻已存在的文件夹中创建的文件或文件夹将不支持删除操作，linux 系统可以。
7. 选择 LockSupport 后，将可以使用文件锁功能（只影响 windows）。
8. OSS 数据挂载会有分布式cache（集群内），所以在大规模并发读取数据时性能较好（能达到 10MB~30MB，200 台并发，读取 20G 数据）。
9. OSS 路径必须以’/’ 结尾。

iii. OSS 输出目录与临时本地输出目录

渲染作业结束时，计算节点中的临时输出目录中的数据将会被传输到 OSS 输出目录中。临时输出路径格式必须与节点的操作系统类型对应，不然会出错。

B) 提交渲染任务

选择目的集群和场景所在的 OSS 路径前缀后进入提交的详细页面，选中场景文件的上一级目录，可以被提交渲染的场景文件则会被列出，勾选想要渲染的文件，选择配置的渲染软件和起止帧，即可提交渲染作业。

可指定节点数量，如果指定集群，并发数量上限是集群的节点数上限。填入的起止帧会均匀的分布在各个计算

节点被渲染。

任务结束后可以在OSS上查看输出结果，如果开启自动下载（配置页面设置），渲管会在任务结束后将OSS上的输出结果下载到渲管部署的机器上。

C) 渲染日志

在节点列表页面，点击节点可以查看各种日志，渲管 worker 日志里都是渲管系统 worker 的日志，里面可以查看该计算节点中运行的实际渲染命令行。

渲染器标准输出和渲染器标准输出里的日志，就是渲染软件的输出日志。

9. 调试

新启动的渲管需要进行配置，并进行调试然后再提交大规模的渲染任务。

配置完，应该先提交1帧测试任务，查看错误日志（渲管 worker 日志和渲染器标准输出）调整渲染软件配置（主要是修改渲染命令行），走通全流程并确认结果没有问题后才进行正式生产渲染。

当作业失败的时候可以在作业信息中查看失败原因项。

建议创建一个集群然后将作业提交到该集群进行调试（AutoCluster 的作业需要启停计算节点，比较费时）。

10. 集群管理

在集群管理页面可以创建自定义集群，需要选择所需的计算节点镜像 ID，节点的实例类型（BatchCompute 的不同区域可能支持的实例类型和磁盘类型不同，详细可以提工单咨询）。

磁盘类型和磁盘大小（根据实际制作的计算节点镜像的磁盘大小选择，选择过小会导致无法启动计算节点）。

创建好的集群可以动态调整节点数量，甚至调整数量到 0。

常见问题

Q：我有大量渲染作业，但是波峰波谷明显，有什么好建议？

A： 使用自定义集群，可长期维持在一定数量，满足日常的渲染需求，当波峰来临时，可以提交 AutoCluster 任务或者调高集群规模（波峰过去调低数量），省钱又省力。

Q：制作完场景后我要上传哪些数据到 OSS？

A： 场景文件，还有场景引用了的贴图、素材及渲染中使用的其他数据，建议在制作场景时所有使用的数据和场景文件都在一个目录里，这样上传一个目录即可。要保证在镜像中访问数据的路径同制作场景时相同，有些渲染软件也可设置素材路径。

Q：我的计算节点可以连接公网么？

A： 目前 BatchCompute 启动的计算节点只有内网 IP，无法连接公网，但同一个作业里的计算节点可以互相连通。

Q：渲染软件需要连接lisence server怎么办？

A： 由于 batchcompute 启动的渲染节点是无公网 IP 的虚拟机，所以对于需要连接lisence server 的渲染软件，可以直接将 lisence server 做在镜像里，这样每个计算节点都会有一个 lisence server。

Q：我想一个阿里云账号部署多个渲管怎么办？

A： 在配置中将 RENDER_FLAG 设置成不同的值，千万不要使用同一个 RENDER_FLAG 部署多份渲管实例，会出错的。

Q：我的作业跑的时间超出1天怎么办？

A： 向 batchcompute 客服提工单增加 timeout 的 quota，并且修改配置中 RENDER_TIMEOUT 值。

Q：提交的作业失败了，渲染器标准输出为空，怎么办？

A： 在节点日志页面，查看 worker 运行信息以及其它几个日志信息，相信能找到蛛丝马迹。

Q：我制作的场景使用的很多贴图分布在各个路径，渲染时如何办？

A: 上传数据到 OSS 时，保持目录结构，在数据映射时填好前缀（可能需要多个映射），尽量保证在计算节点中看到的渲染数据文件结构与制作时一样。

Q：我制作的场景使用了远程的文件怎么办（windows）？

A： 制作镜像时，将远程 nas 的名字设置成本地机器的别名，在执行代码中执行命令将目标文件夹共享，如果

数据小，也可以直接将数据制作进镜像，并共享。

Q：我的数据分布在多个盘符（windows）里怎么办？

A： 在创建项目时，OSS数据映射项，直接映射多个盘符。

Q：虚拟机内临时输出路径必须在C盘下么（windows）？

A: 是的，虚拟机只有一个 C 盘（默认 40G）。

Q：系统盘40G不够大怎么办？

A： 在制作计算节点镜像时可以使用更大的系统盘，在使用该计算节点镜像创建集群时也需要选择足够大的磁盘容量，但使用超过 40G 的磁盘，BatchCompute 可能会收取少量费用。

Q：我想节点并发数量大于 100 怎么办？

A： 提工单给 BatchCompute 修改配额，并在渲管配置页面修改 MAX_NODE_NUM。

Q：对于集群和 AutoCluster 有什么使用建议么？

A: 看场景。AutoCluster 类型的作业每个节点都要经历启停（启停时间在分钟级别），对于运行时间很短的任务比较不划算，而且可能因为资源紧张而等待，大量小任务建议创建集群进行渲染。对等待时间有要求的用户也应该使用自定义集群，这样提交任务到该集群，马上就可以运行，但 AutoCluster 的任务不用担心集群利用率的问题。

Q：我是程序员，我可以改代码么？

A：渲管是开源的（apache 2.0），想怎么改怎么改，请记得贡献回社区哦。

Cromwell 工作流引擎支持

Cromwell 是 Broad Institute 开发的工作流管理系统，当前已获得阿里云批量计算服务的支持。通过 Cromwell 可以将 WDL 描述的 workflow 转化为批量计算的作业（Job）运行。用户将为作业运行时实际消耗的计算和存储资源付费，不需要支付资源之外的附加费用。本文将介绍如何使用 Cromwell 在阿里云批量计算服务上运行工作流。

1. 准备工作

A) 开通批量计算服务

要使用批量计算服务，请根据官方文档里面的指导开通批量计算和其依赖的相关服务，如OSS等。

> 注意：创建 OSS Bucket 的区域,需要和使用批量计算的区域一致。

B) 下载 Cromwell

Cromwell 官方下载

> 注意：为了确保所有的特性可用，建议下载45及之后的最新版本。

C) 开通 ECS 作为 Cromwell server

当前批量计算提供了 Cromwell server 的 ECS 镜像，用户可以用此镜像开通一台 ECS 作为 server。镜像中提供了 Cromwell 官网要求的基本配置和常用软件。在此镜像中，Cromwell 的工作目录位于 /home/cromwell，上一步下载的 Crowwell jar 包可以放置在 /home/cromwell/cromwell 目录下。

> 注意：用户也可以自己按照 Cromwell 官方的要求自己搭建 Cromwell server， 上面的镜像只是提供了方便的方式，不是强制要求。

2. 使用 Cromwell

配置文件

Cromwell 运行的配置文件，包括：

- Cromwell 公共配置。
- 批量计算相关配置，包含了批量计算作为后端需要的存储、计算等资源配置。

关于配置参数的详细介绍请参考 Cromwell 官方文档。如下是一个批量计算配置文件的例子 bcs.conf:

```
include required(classpath("application"))


database {
profile = "slick.jdbc.MySQLProfile$"
db {
driver = "com.mysql.jdbc.Driver"
url =
"jdbc:mysql://localhost/db_cromwell?rewriteBatchedStatements=true&useSSL=false&allowPublicKeyRetrieval=true
```

```
"
user = "user_cromwell"
#Your mysql password
password = ""
connectionTimeout = 5000
}
}

workflow-options {
workflow-log-dir = "/home/cromwell/cromwell/logs/"
}

call-caching {
# Allows re-use of existing results for jobs you've already run
# (default: false)
enabled = false

# Whether to invalidate a cache result forever if we cannot reuse them. Disable this if you expect some cache copies
# to fail for external reasons which should not invalidate the cache (e.g. auth differences between users):
# (default: true)
invalidate-bad-cache-results = true
}

docker {
hash-lookup {
enabled = false
# Set this to match your available quota against the Google Container Engine API
#gcr-api-queries-per-100-seconds = 1000

# Time in minutes before an entry expires from the docker hashes cache and needs to be fetched again
#cache-entry-ttl = "20 minutes"

# Maximum number of elements to be kept in the cache. If the limit is reached, old elements will be removed from
the cache
#cache-size = 200

# How should docker hashes be looked up. Possible values are "local" and "remote"
# "local": Lookup hashes on the local docker daemon using the cli
# "remote": Lookup hashes on docker hub and gcr
method = "remote"
#method = "local"
alibabacloudcr {
num-threads = 5
#aliyun CR credentials
auth {
#endpoint = "cr.cn-shanghai.aliyuncs.com"
access-id = ""
access-key = ""
}
}
}
}

engine {
filesystems {
oss {
```

```
auth {
endpoint = "oss-cn-shanghai.aliyuncs.com"
access-id = ""
access-key = ""
}
}
}
}

backend {
default = "BCS"

providers {
BCS {
actor-factory = "cromwell.backend.impl.bcs.BcsBackendLifecycleActorFactory"
config {
root = "oss://your-bucket/cromwell_dir"
region = "cn-shanghai"
access-id = ""
access-key = ""

filesystems {
oss {
auth {
endpoint = "oss-cn-shanghai.aliyuncs.com"
access-id = ""
access-key = ""
}

caching {
# When a cache hit is found, the following duplication strategy will be followed to use the cached outputs
# Possible values: "copy", "reference". Defaults to "copy"
# "copy": Copy the output files
# "reference": DO NOT copy the output files but point to the original output files instead.
# Will still make sure than all the original output files exist and are accessible before
# going forward with the cache hit.
duplication-strategy = "reference"
}
}
}

default-runtime-attributes {
failOnStderr: false
continueOnReturnCode: 0
autoReleaseJob: false
cluster: "OnDemand ecs.sn1.medium img-ubuntu-vpc"
#cluster: cls-6kihku8blloidu3s1t0006
vpc: "192.168.0.0/16"
}
}
}
}
}
```

如果使用前面章节中的镜像开通 ECS 作为 Cromwell server，配置文件位于

/home/cromwell/cromwell/bcs_sample.conf，只需要填写自己的配置即可使用 Cromwell。

注意：Cromwell 可以在公网环境（如本地服务器、配置了公网 IP 的阿里云 ECS 等）运行，也可以在阿里云 VPC 环境下运行。在 VPC 环境下使用时，有如下几处要修改为 VPC 内网下的配置：

- OSS 的内网 endpoint：
  - engine.filesystems.oss.auth.endpoint = "oss-cn-shanghai-internal.aliyuncs.com"
  - backend.providers.BCS.config.filesystems.oss.auth.endpoint = "oss-cn-shanghai-internal.aliyuncs.com"
- 添加批量计算的内网 endpoint:
  - backend.providers.BCS.config.user-defined-region = "cn-shanghai-vpc"
  - backend.providers.BCS.config.user-defined-domain = "batchcompute-vpc.cn-shanghai.aliyuncs.com"
- 添加容器镜像服务的内网 endpoint：
  - docker.hash-lookup.alibabacloudcr.auth.endpoint = "cr-vpc.cn-shanghai.aliyuncs.com"

运行模式

Cromwell支持两种模式：

- run 模式
- server 模式

关于两种模式的详细描述，请参考 Cromwell 官网文档。下面重点介绍这两种模式下如何使用批量计算。

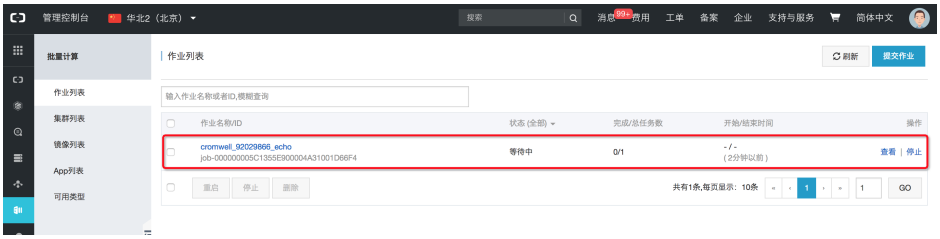
A) run模式

run模式适用于本地运行一个单独的 WDL 文件描述的工作流，命令行如下：java -Dconfig.file=bcs.conf -jar cromwell.jar run echo.wdl --inputs echo.inputs

- WDL 文件：描述详细的工作流。工作流中每个 task 对应批量计算的一个作业（Job）。
- inputs文件：是 WDL 中定义的工作流的输入信息inputs 文件是用来描述 WDL 文件中定义的工作流及其 task 的输入文件。如下所示：

```
{
"workflow_name.task_name.input1": "xxxxxx"
}
```

运行成功后，WDL 文件中描述的工作流中的一个 task 会作为批量计算的一个作业（Job）来提交。此时登录批量计算的控制台就可以看到当前的 Job 状态。



当 workflow 中所有的 task 对应的作业运行完成后，工作流运行完成。

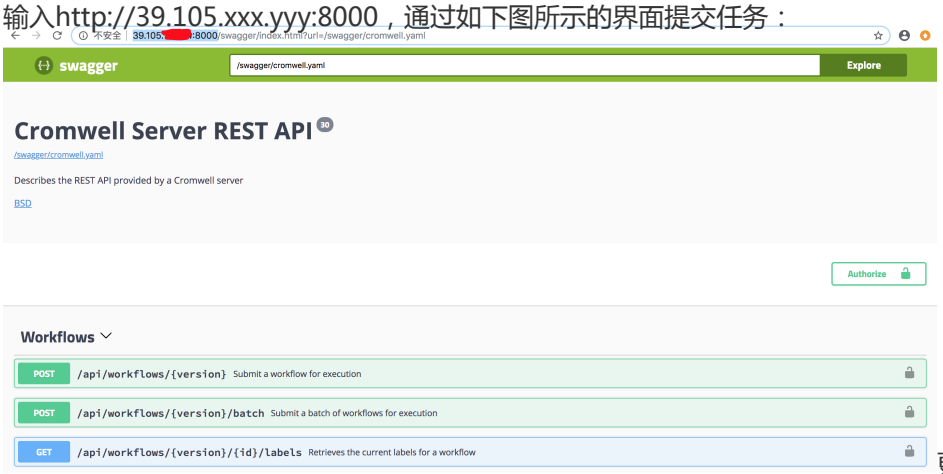
B) server 模式

启动 server

相比 run 模式一次运行只能处理一个 WDL 文件，server 模式可以并行处理多个 WDL 文件。关于 server 模式的更多信息，请参考 Cromwell 官方文档。可以采用如下命令行启动 server：java -Dconfig.file=bsc.conf -jar cromwell.jar serverserver 启动成功后，就可以接收来自 client 的工作流处理请求。下面分别介绍如何使用 API 和 CLI 的方式向 server 提交工作流。

使用 API 提交工作流

server 启动后，可以通过浏览器访问 Cromwell Server，比如 Server 的 IP 为39.105.xxx.yyy，则在浏览器中输入http://39.105.xxx.yyy:8000，通过如下图所示的界面提交任务：



更多API接口及用法，请参考 Cromwell 官网文档。

使用 CLI 提交工作流[推荐]

除了可以使用 API 提交工作流以外，Cromwell 官方还提供了一个开源的 CLI 命令行工具 widder。可以使用如下的命令提交一个工作流：

```
python widdler.py run echo.wdl echo.inputs -o bcs_workflow_tag:tagxxx -S localhost
```

其中-o key:value是用于设置option，批量计算提供了 bcs_workflow_tag:tagxxx 选项，用于配置作业输出目录的tag（下一节查看运行结果中会介绍）。

如果使用前面章节中的镜像开通 ECS 作为 Cromwell server，镜像中已经安装了 widdler，位于 /home/cromwell/widdler。可以使用如下的命令提交工作流：

```
widdler run echo.wdl echo.inputs -o bcs_workflow_tag:tagxxx -S localhost
```

更多命令用法可使用widdler -h命令查看，或参考官方文档。

3. 查看运行结果

工作流运行结束后，输出结果被上传到了配置文件或 WDL 中定义的 OSS 路径下。在OSS路径上面的目录结构如下：

如上图所示，在配置文件中的config.root目录下有如下输出目录：

- 第一层：workflowname 工作流的名称
- 第二层：通过上一节中 CLI 命令的-o设置的目录tag
- 第三层：workflow id，每次运行会生成一个
- 第四层：workflow 中每个 task 的运行输出，比如上图中的 workflow 15e45adf-6dc7-4727-850c-

89545faf81b0 有两个 task，每个task对应的目录命名是call-taskname，目录中包含三部分内容：

- 批量计算的日志，包括 bcs-stdout 和 bcs-stderr
- 当前 task 的输出，比如图中的 output1/output2 等
- 当前 task 执行的 stdout 和 stderr

4. 使用建议

在使用过程中，关于 BCS 的配置，有如下的建议供参考：

使用集群

批量计算提供了两种使用集群的方式：

- 自动集群
- 固定集群

A) 自动集群

在config配置文件中指定默认的资源类型、实例类型以及镜像类型，在提交批量计算 Job 时就会使用这些配置自动创建集群，比如：

```
default-runtime-attributes {
cluster : "OnDemand ecs.sn1ne.large img-ubuntu-vpc"
}
```

如果在某些 workflow 中不使用默认集群配置，也可以通过inputs文件中指定 workflow 中某个 task 的对应的批量计算的集群配置（将 cluster_config 作为 task 的一个输入），比如：

```
{
"workflow_name.task_name.cluster_config": "OnDemand ecs.sn2ne.8xlarge img-ubuntu-vpc"
}
```

然后在 task 中重新设置运行配置:

```
task task_demo {
String cluster_config

runtime {
cluster: cluster_config
}
}
```

就会覆盖默认配置，使用新的配置信息创建集群。

B) 固定集群

使用自动集群时，需要创建新集群，会有一个等待集群的时间。如果对于启动时间有要求，或者有了大量的作业提交，可以考虑使用固定集群。比如：

```
default-runtime-attributes {
cluster : "cls-xxxxxxxxxx"
}
```

注意：使用固定集群时，如果使用完毕，请及时释放集群，否则集群中的实例会持续收费。

Cromwell Server 配置建议

- 大压力作业时，建议使用较高配置的机器作为 Cromwell Server，比如ecs.sn1ne.8xlarge等32核64GB的机器。
- 大压力作业时，修改 Cromwell Server 的最大打开文件数。比如在ubuntu下可以通过修改/etc/security/limits.conf文件，比如修改最大文件数为100万：

```
root soft nofile 1000000
root hard nofile 1000000
* soft nofile 1000000
* hard nofile 1000000
```

- 确认 Cromwell Server 有配置数据库，防止作业信息丢失。
- 设置 bcs.conf 里面的并发作业数，比如 system.max-concurrent-workflows = 1000

开通批量计算相关配额

如果有大压力场景，可能需要联系批量计算服务开通对应的配额，比如：

- 一个用户所有作业的数量（包括完成的、运行的、等待的等多种状态下）；
- 同时运行的作业的集群的数量（包括固定集群和自动集群）；

使用 NAS

使用 NAS 时要注意以下几点：

- NAS 必须在 VPC 内使用，要求添加挂载点时，必须指定 VPC；
- 所以要求在 runtime 中必须包含：
    - VPC 信息
    - mounts 信息

下面的例子可供参考：

```
runtime {
cluster: cluster_config
mounts: "nas://1f****04-xkv88.cn-beijing.nas.aliyuncs.com:/ /mnt/ true"
vpc: "192.168.0.0/16 vpc-2zexxxxxxxx1hxirm"
}
```

高级特性支持

Glob

Cromwell 支持使用 glob 来指定工作流中多个文件作为 task 的输出，比如：

```
task globber {
command <<<
for i in `seq 1 5`
do
mkdir out-$i
echo globbing is my number $i best hobby out-$i/$i.txt
done
>>>
output {
Array[File] outFiles = glob("out-*/*.txt")
}
}

workflow test {
call globber
}
```

当 task 执行结束时，通过 glob 指定的多个文件会作为输出，上传到 OSS 上。

Call Caching

Call Caching 是 Cromwell 提供的高级特性，如果检测到工作流中某个 task (对应一个批量计算的 job)和之前已经执行过的某个 task 具有相同的输入和运行时等条件，则不需要再执行，直接取之前的运行结果，这样可以为客户节省时间和费用。一个常见的场景是如果一个工作流有 n 个 task，当执行到中间某一个 task 时由于某些原因失败了，排除了错误之后，再次提交这个工作流运行后，Cromwell 判断如果满足条件，则已经完成的几个 task 不需要重新执行，只需要从出错的 task 开始继续运行。

配置 Call Caching

要在 BCS 后端情况下使用 Call Caching 特性，需要如下配置项：

```
database {
profile = "slick.jdbc.MySQLProfile$"
db {
driver = "com.mysql.jdbc.Driver"
```

```
url = "jdbc:mysql://localhost/db_cromwell?rewriteBatchedStatements=true&useSSL=false"
user = "user_cromwell"
password = "xxxxx"
connectionTimeout = 5000
}
}

call-caching {
# Allows re-use of existing results for jobs you have already run
# (default: false)
enabled = true

# Whether to invalidate a cache result forever if we cannot reuse them. Disable this if you expect some cache copies
# to fail for external reasons which should not invalidate the cache (e.g. auth differences between users):
# (default: true)
invalidate-bad-cache-results = true
}

docker {
hash-lookup {

enabled = true

# How should docker hashes be looked up. Possible values are local and remote
# local: Lookup hashes on the local docker daemon using the cli
# remote: Lookup hashes on alibab cloud Container Registry
method = remote
alibabacloudcr {
num-threads = 10
auth {
access-id = "xxxx"
access-key = "yyyy"
}
}
}
}

engine {
filesystems {
oss {
auth {
endpoint = "oss-cn-shanghai.aliyuncs.com"
access-id = "xxxx"
access-key = "yyyy"
}
}
}
}

backend {
default = "BCS"

providers {
BCS {
actor-factory = "cromwell.backend.impl.bcs.BcsBackendLifecycleActorFactory"
config {
```

```
#其他配置省略

filesystems {
oss {
auth {
endpoint = "oss-cn-shanghai.aliyuncs.com"
access-id = "xxxx"
access-key = "yyyy"
}
caching {
# When a cache hit is found, the following duplication strategy will be followed to use the cached outputs
# Possible values: copy, reference. Defaults to copy
# copy: Copy the output files
# reference: DO NOT copy the output files but point to the original output files instead.
# Will still make sure than all the original output files exist and are accessible before
# going forward with the cache hit.
duplication-strategy = "reference"
}
}
}

default-runtime-attributes {
failOnStderr: false
continueOnReturnCode: 0
cluster: "OnDemand ecs.sn1.medium img-ubuntu-vpc"
vpc: "192.168.0.0/16"
}
}
}
}
}
```

- database 配置：Cromwell 将 workflow 的执行元数据存储在数据库中，所以需要添加数据库配置，详细情况参考Cromwell 官网指导。
- call-caching 配置：Call Caching 的开关配置等；
- docker.hash-lookup 配置： 设置 Hash 查找开关及阿里云 CR 等信息，用于查找镜像的 Hash 值。
- backend.providers.BCS.config.filesystems.oss.caching 配置：设置 Call Caching命中后，使用原来输出的方式，批量计算在这里支持 reference 模式，不需要拷贝原有的结果，节省时间和成本。

命中条件

使用批量计算作为后端时，Cromwell 通过如下条件判断一个 task 是否需要重新执行：

条件	解释
inputs	task 的输入，比如 OSS 上的样本文件
command	task 定义中的命令行
continueOnReturnCode	公共运行时参数，可以继续执行的返回码
docker	公共运行时参数，后端的Docker配置
failOnStderr	公共运行时参数，stderr非空时是否失败

imageId	批量计算后端运行时参数，标识作业运行的 ECS 镜像，如果使用的官方镜像如img-ubuntu-vpc可不用填写此项
userData	批量计算后端，用户自定义数据

如果一个 task 的上述参数未发生改变，Cromwell 会判定为不需要执行的 task，直接获取上次执行的结果，并继续工作流的执行。

SGE集群支持

批量计算支持自动化搭建 Sun Grid Engine（SGE）集群，批量计算使用的是 CentOS 自带的 SGE 版本，请参考 SGE 。

批量计算提供了名为 BatchCompute SGE 的公共镜像，使用该镜像可快速、可靠的构建 SGE 集群，具体的流程如下：

1. 获取 BatchCompute SGE 镜像

请在云市场 搜索关键字 BatchCompute SGE 了解该镜像，它完全免费使用，使用流程请参考 如何通过镜像创建实例 。

2. 自定义镜像（可选）

本步骤可选，如对镜像没有特殊需求，可直接进入下一步。如果需要在此系统镜像基础上安装软件，必须基于 BatchCompute SGE 制作自定义镜像，请参考 自定义镜像 。

> - 必须在 BatchCompute SGE 镜像基础上制作新镜像。
> - 制作镜像过程中，请务必不要执行任何有关 SGE 和 bcc 工具的命令，并且不要更新 python。

3. 准备 SGE Master 节点

请指定某 ECS VM 作为 SGE 系统的 Master 节点，它负责管理整个集群，也可以充当提交作业的节点。如果采用自定义镜像，在启动 VM 时要选用自定义镜像，否则选用 BatchCompute SGE 镜像。

配置参数，请参考 创建 Linux 实例。

> 由于 Master 节点需要长期稳定运行，建议在启动 VM 时选用包年包月的付费方式；如果是测试，建议使用按量方式。

详细步骤如下:

A) 创建 VPC 和交换机

如果您需要使用已经存在的 VPC ，可以跳过这一步。

打开 ECS 官方控制台，点击专有网络 VPC 进入 VPC 控制台，然后点击”专有网络”菜单。

创建专有网络。在本示例中，设置专用网络 CIDR 为 192.168.0.0/16，而交换机 CIDR 为 192.168.0.0/24。

- 创建交换机。

B) 购买 ECS VM

- 点击刚才创建的“专有网络”，然后点击“交换机”进入交换机列表，再点击“创建实例”进入创建 ECS 实例页面。
- 公网 IP 地址选择分配。
- 选择安全组，创建专有网络时自动创建了一个安全组，这里只有一个安全组可选。
- 实例规格至少2核4GB。
- 镜像市场: BatchCompute SGE。
- 设置密码。
- 配置参考。
- 请注意创建实例时实例的名称不能修改

3. 启动 SGE 集群

批量计算提供了命令行工具 bccluster(bcc) 来帮助您管理 SGE 集群，该工具预装到 BatchCompute SGE 镜像中。

A) 登录 Master

使用 ssh 命令登录到 Master 节点，务必使用 root 用户。

```
ssh root@<外网IP>
```

然后，输入购买 ECS 时设置的密码.

B) bccluster 命令登录

bccluster（bcc） 工具用来管理 SGE 集群，包括启动、扩容和停止等操作。如果第一次登入 Master 节点，请先更新 bccluster 工具，然后执行以下命令来配置 region, accessKeyId 和 accessKeySecret。其中的 region

必须与 Master 虚拟机所在的 region 相同。

```
pip install -U batchcompute-sge #如果命令执行出错，重试该命令就可以了。
bcc login <region> <accessKeyId> <accessKeySecret>
```

- 该命令只需要第一次登入 Master 节点时执行。
- AccessKey 对应的子账号，要被授予 BatchCompute 全部权限 和 ECS 查询权限，以及 AuthorizeSecurityGroup 和 RevokeSecurityGroup 两个 ECS 写操作 API 的权限。请打开 RAM 控制台 点击“用户管理”菜单，选择相应的子用户进行授权。
- region 参考列表 。
- 执行 bcc login 命令出现如下错误说明 ECS 的名称被修改过，需要删除 ECS 实例重新创建实例然后进行登录操作。

```
fix ip hostname binding...
fixed
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0    22    0    22    0     0   2804      0 --:--:-- --:--:-- --:--:--  5500

  ERROR: Can not found master_security_group_id by i-2zeic9ggky8iuudm5jy9
```

C) start 集群

启动worker节点。

```
bcc start -n 2 -t ecs.sn2.medium -i img-sge --vpc_cidr_block=192.168.1.0/24
```

参数:

- -n 表示启动多少台 worker 节点。
- -t 表示 worker 节点使用哪种实例类型, bcc t命令可以列举可用的实例类型。
- -i 表示 worker 节点使用哪个镜像（可以指定系统镜像 ID：img-sge，或者自定义镜像 ID）。
- —vpc_cidr_block 指定集群网段, 请参考 如何选择网段 。

运行完该命令, 启动指令提交成功, 因为 worker 节点启动有一段时间, 还不能立即使用该集群，需要等待一段时间。

SGE 集群只能运行在 vpc 网络中，因此必须指定 —vpc_cidr_block；cidr_block必须在创建master ECS实例设置的CIDR范围内，如本例创建master ecs时选的vpc cidr为 192.168.0.0/16，所以cidr_block可选范围在192.168.0.0/16-192.168.0.0/24

D) 查看批量计算集群状态

```
bcc status
```

该命令可以查看集群状态, worker 节点启动情况等。

E) 查看 SGE 集群状态

```
qhost
```

尝试运行qhost命令，看看 SGE 集群是否完全启动。

4. 释放(删除) SGE 集群

如果不再使用 worker 节点，请使用 stop 命令停止所有的 worker 节点。如果 master 节点也不再使用，可以通过控制台删除掉 master 节点。

```
bcc stop
```

注意：必须先停止 worker 节点，然后才能释放 master。

5. 如何启动 NAS 挂载的 SGE 集群

使用 bcc 工具可以轻松挂载 NAS，示例如下：

```
bcc start -n 2 -t ecs.sn1.medium -i img-sge --vpc_cidr_block=192.168.1.0/24 -m nas://a/b/c:/home/nas/
```

注意：如何在 VPC 里面创建 NAS 文件系统，请参考 创建文件系统 和 添加挂载点 。

6. 启动多种实例类型的集群（多个group）

运行 bcc start 命令时, 增加 option: —group_num 4 # 表示创建 4 个 group。

```
bcc start -n 2 -t ecs.sn2.medium -i img-sge --vpc_cidr_block=192.168.1.0/24 option: --group_num 4
```

- group 名称分别为: default，group1，group2，group3，并且 group1，group2，group3 的 node 数量都是 0。
- 通过 bcc update 命令批量修改所有 group 的 instanceType 或者只修改某个 group 的 instanceType ；具体参考bcc update --help。
- 通过 bcc resize 命令某一个 group 的 node 数量；具体参考bcc resize --help。

注意： group 个数在 start 后不能变更，如需变更 group 数量，必须 stop 集群后后再重新 start。

7. 如何创建包年包月的 SGE 集群

请按照前面的步骤，启动一个 SGE master 节点，然后登入并初始化 bcc 工具（更新并且 login）；登录到阿里云工单系统提交工单联系运维工程师做包年包月集群的配置处理。

注意：包年包月集群创建后不支持删除，只能等到包月时间点到之后才能释放集群；测试场景建议使用按量使用模式，待准备工作完成后再开通包月集群；bcc start 命令不支持创建包年包月的 SGE 集群。

A）控制台创建包年包月的集群

1. 登入批量计算的控制台，选择您的 master 节点所在的区域，在“集群列表”页面中，点击“创建集群”。
2. 填写必选字段，其中的“镜像 ID”需要选择为“sge(官网提供)”（如果您是自定义的镜像，那么需要选择您自定义的镜像 ID），资源类型选中“包月”。
3. 填写可选字段。
    - Bootstrap：/usr/local/bin/sge_bootstrap，必须为该值。
    - 增加 2 个环境变量。SGE_MASTER_IP_ADDRESS：对应 master 所在的 IP 地址，SGE_MASTER_HOST_NAME：对应 master 所在的 hostname。
    - VpcId：对应 master 所在 VPC。
    - CidrBlock：指定一个 CidrBlock，注意不能与该 VPC 中已有的地址段相冲突。
    - 如果需要挂载 NAS，那么需要增加“Mounts”选项。
4. 点击“提交”，创建集群。
5. 集群创建成功后，进入该集群的页面。
6. 点击“创建预付费实例”，在新的页面中，选择“项目”，“集群”，“实例组”，点击“立即购买”，再点击“去支付”，“确认支付”。

注意，在上面第 6 步中，一定要确认“项目”，“集群”，“实例组”这三个选项。

B）命令行 attach 该集群

在命令行中，执行如下命令：

```
bcc attach <your_cluster_id>
```

执行成功后，就完成了包年包月的 SGE 集群的创建。

SGE集群1.1版本

批量计算支持自动化搭建 Sun Grid Engine（SGE）集群，批量计算使用的是 CentOS 自带的 SGE 版本，请参考 SGE 。

批量计算提供了名为 BatchCompute SGE 的公共镜像，使用该镜像可快速、可靠的构建 SGE 集群，具体的流程如下：

1. 获取 BatchCompute SGE 镜像

请在云市场 搜索关键字 BatchCompute SGE 了解该镜像，它完全免费使用，使用流程请参考 如何通过镜像创建实例 。

2. 自定义镜像（可选）

本步骤可选，如对镜像没有特殊需求，可直接进入下一步。如果需要在此系统镜像基础上安装软件，必须基于 BatchCompute SGE 制作自定义镜像， 制作过程请参考 自定义镜像 。

- 必须在 BatchCompute SGE V1.1 版本镜像基础上制作新镜像。
- BatchCompute SGE V1.1 版本在原有支持命令行创建 SGE 集群的基础上，推出控制台一键创建 SGE 集群。无需用户通过命令行创建、扩容，以及删除 SGE集群操作。
- 制作镜像过程中，请务必不要执行任何有关 SGE 的命令，并且不要更新 python 。
- 镜像制作完成后需要注册给 BatchCompute 。

3. 控制台创建 SGE 集群

3.1. 设置集群名称和镜像

登录到 BatchCompute 控制台，确定集群所在的 Region 点击创建 创建集群 按钮，准备集群创建。

选择 创建 SGE 集群，若采用系统镜像则选择 sge-centos-vpc-x64(官网提供) ；若是采用自定义镜像则选择注册的自定义镜像。设置完成后进行下一步：

3.2. 设置组信息

根据业务需求配置 SgeMaster 的实例类型 和 镜像 ID。

- SGE work节点 支持设置多个组；组间实例类型、实例个数以及镜像ID 可以互不相同。
- SGE 集群内所有 work 节点都可以在 Master 节点通过 ssh hostname 进行免密登录。
- SGE 集群内所有 work 节点之间网络互通，不支持免密登录。
- SGE Master 属于单独的一个组，实例类型支持和 work 不同，组内节点个数有且只有一个

3.3. 设置挂载信息

根据需求配置数据盘信息，NAS/OSS 挂载信息。

- 若添加了 OSS 挂载到本地，则只支持 OSS 的读操作。
- 若写数据到 OSS 映射到 VM 本地路径上，则数据无法上传到 OSS 对象中，节点重启后数据丢失。

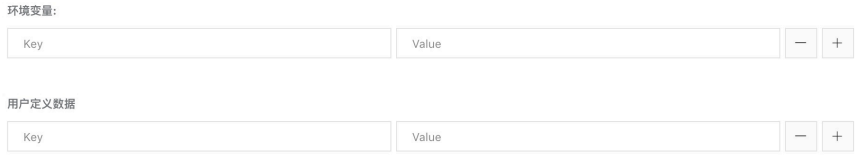
3.4. 设置网络信息

可以将网络设置到指定的 VPC ；也可以采用默认网络设置配置集群

- 若挂载有 NAS 时，网络设置必须和 NAS 保持在同一个 VPC 内；否则无法正常挂载 NAS 。
- SGE 集群只支持 VPC 网络。

3.5. 设置环境变量

根据业务需要进行环境变量配置操作



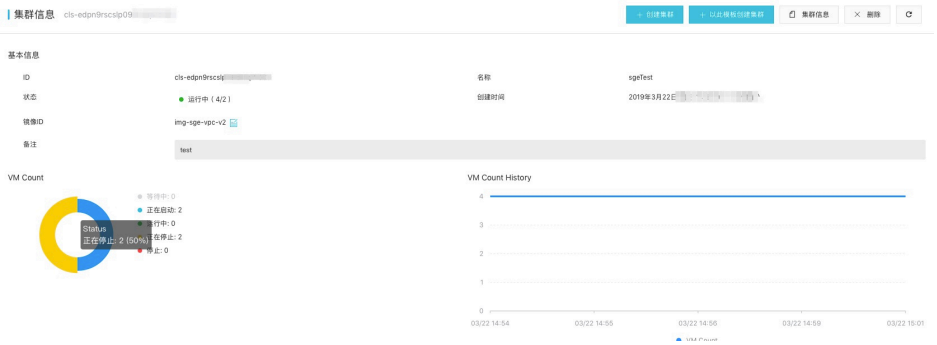
3.6. 提交创建操作

设置完成后提交集群创建即可。提交成功后可以看到集群处于初始化状态。

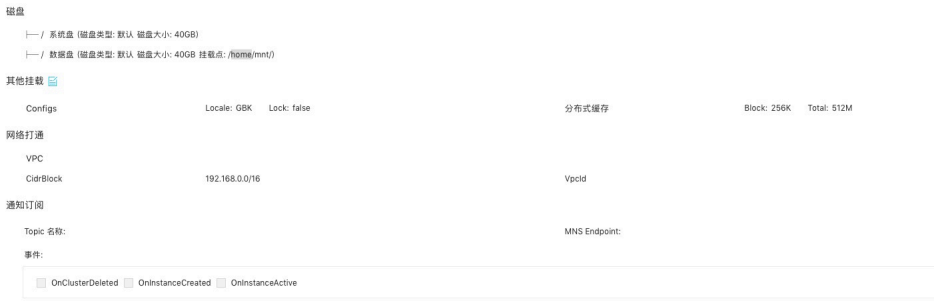
4. SGE 集群查看

在集群列表页面，点击 “查看” 可以进入 SGE 集群的详细信息页面

4.1. 集群状态显示



4.2. 集群挂载显示



4.3. 集群实例组显示

展示各个组内实例的类型、个数以及镜像信息。

- SGE 集群 支持按组做扩容或者缩容操作
- 支持按组展开组内实例列表信息，查看实例在 VPC 内的 IP以及登录密码信息；

4.4. 集群实例列表显示

该页面显示 实例ID、名称、hostname 以及 机器IP 登录密码等信息。

- 密码信息获取关闭密码隐藏功能方可获取。
- 支持采用 VNC 登录方式登录到实例节点

4.4. 集群操作日志显示

显示集群的历史操作信息

5. SGE 集群扩容缩容

在 BatchCompute 控制台，找到指定的 SGE 集群。进入到集群详细信息标签页，在对应的实例组中直接修改期望的实例个数，点击“修改”即可。

- Master 组不支持进行扩容或者缩容操作

6. SGE 集群删除

在 BatchCompute 控制台，找到指定的 SGE 集群。进入到集群详细信息标签页，点击“删除”按钮，即可删除对应的集群。

7. 登录 SGE Master 节点

在 BatchCompute 控制台，找到指定的 SGE 集群。进入到详细信息标签页，在对应的实例组“sgeMasterGroup”中查看实例列表信息，可以获取 Master 节点的公网 IP 以及登录密码信息。

使用 ssh 命令登录到 Master 节点，务必使用 root 用户。

```
ssh root@<外网IP>
```

进入Master 节点后，通过 SGE 相关命令对集群进行配置提交作业操作。

> - 集群启动需要一定时间，进入 Master 后执行 SGE 命令出现无法执行，请稍等片刻后重试即可。

GATK支持

GATK 软件分析流程由阿里云和 Broad Institute 合作提供。Broad Institute 提供的 GATK 流程最佳实践用 工作流定义语言（WDL）编写，通过批量计算集成的 Cromwell 工作流引擎解析执行。用户将为作业运行时实际消耗的计算和存储资源付费，不需要支付资源之外的附加费用。

Broad Institute GATK 网站和论坛为 GATK 工具和 WDL 提供了更完整的背景信息，文档和支持。

如果需要执行用 WDL 编写的通用工作流程，请参考 cromwell 工作流引擎和 WDL 支持的 APP 。

1. 准备

A) 使用 OSS 存储

要在批量计算上运行 GATK，输入、输出文件都需要保存在 OSS。所以，需要先开通 OSS 并创建好 Bucket。

注意：创建 Bucket 的区域,需要和运行批量计算的 GATK 区域一致。

B) 安装 batchcompute-cli 命令行工具

```
pip install batchcompute-cli
```

安装完成后，还需要 配置 。

注意：当前最佳实践中使用的 GATK 相关软件版本信息如下：

- GATK: 4.0.0.0
- picard: 2.13.2
- genomes-in-the-cloud: 2.3.0-1501082129

2. 快速运行

本示例中，运行 Broad Institute 提供的 GATK4 版本全基因分析流程，该流程分为两步：

- 第一步为 gatk4-data-processing 。
- 第二步为 gatk4-germline-snps-indels 。

在配置好 bcs 工具后，执行如下命令：

```
bcs gen ./demo -t gatk
cd demo/gatk4-data-processing
sh main.sh # 运行gatk4-data-processing 流程
cd ../gatk4-germline-snps-indels
sh main.sh # 运行gatk4-germline-snps-indels 流程
```

这样您就在批量计算上运行了以上两个 GATK4 流程。

3. 命令详解

A) 生成示例

执行如下命令生成示例：

```
bcs gen ./demo -t gatk
```

它将生成以下目录结构:

```
demo
```

```
|-- Readme.md
|-- gatk4-data-processing
| |-- main.sh
| |-- src
| |-- LICENSE
| |-- README.md
| |-- generic.batchcompute-papi.options.json
| |-- processing-for-variant-discovery-gatk4.hg38.wgs.inputs.json
| |-- processing-for-variant-discovery-gatk4.hg38.wgs.inputs.30x.json
| |-- processing-for-variant-discovery-gatk4.wdl
|-- gatk4-germline-snps-indels
|-- main.sh
|-- src
|-- LICENSE
|-- README.md
|-- generic.batchcompute-papi.options.json
|-- haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.json
|-- haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.30x.json
|-- haplotypecaller-gvcf-gatk4.wdl
```

- gatk4-data-processing 目录中包括了运行 gatk4-data-processing 流程所需的所有配置和脚本。
- gatk4-germline-snps-indels 目录中包括了运行 gatk4-germline-snps-indels 流程所需的所有配置和脚本。
- 每个目录下面的 main.sh 脚本封装了使用 bcs 工具提交作业的命令。
- src 目录下面包括了工作流实现代码。

B) 运行 gatk4-data-processing 流程

进入 demo/gatk4-data-processing 目录，运行 main.sh，该文件内容如下：

```
#!/bin/bash

# bcs asub cromwell -h for more

bcs asub cromwell gatk-job\
--config ClassicNetwork=false\
--input_from_file_WDL src/processing-for-variant-discovery-gatk4.wdl\
--input_from_file_WORKFLOW_INPUTS src/processing-for-variant-discovery-gatk4.hg38.wgs.inputs.json\
--input_from_file_WORKFLOW_OPTIONS src/generic.batchcompute-papi.options.json\
--input_WORKING_DIR oss://demo-bucket/cli/gatk4_worker_dir/\
--output_OUTPUTS_DIR oss://demo-bucket/cli/gatk4_outputs/\
-t ecs.sn1.large -d cloud_efficiency
```

其中，部分参数描述为：

- input_from_file_WDL： WDL 流程描述文件路径。
- input_from_file_WORKFLOW_INPUTS：WDL 流程输入文件。
- input_from_file_WORKFLOW_OPTIONS：WDL 流程选项文件。
- input_WORKING_DIR：OSS上的目录，用来存储 WDL 流程中各个步骤生成的文件，bcs 会自动给您生成一个默认的路径。

- output_OUTPUTS_DIR：OSS 上的目录，用来存储 WDL 流程结束后生成的 metadata 文件，bcs 会自动给您生成一个默认的路径。

其他参数，请参考 bcs asub -h 命令。

如果希望使用此流程来运行自己的数据，需要修改 src/processing-for-variant-discovery-gatk4.hg38.wgs.inputs.json 文件中的 PreProcessingForVariantDiscovery_GATK4.flowcell_unmapped_bams_list 参数，指定存储在 OSS 上的 ubam 文件。

注意：该示例中的流程输入文件不是 FASTQ 格式，而是 unaligned BAM 文件。

C) 运行 gatk4-germline-snps-indels 流程

该流程的运行与 gatk4-data-processing 流程类似，参考上述章节。

- 如果希望使用此流程来运行自己的数据，需要修改 src/haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.json 文件中的 HaplotypeCallerGvcf_GATK4.input_bam 参数，修改为 gatk4-data-processing 流程输出的 bam 文件路径。
- 将 HaplotypeCallerGvcf_GATK4.input_bam_index 参数修改为相应的索引文件路径。

4. 作业状态查询与日志

在提交作业后，如果看到以下信息，说明提交成功

```
Job created: job-0000000059DC658400006822000001E3
```

job-0000000059DC658400006822000001E3 即是当次提交作业的 ID。

查看作业状态:

```
bcs j  # 获取作业列表
bcs j job-0000000059DC658400006822000001E3 # 查看作业详情
```

查看作业日志:

```
bcs log job-0000000059DC658400006822000001E3
```

5. 验证结果

查看 OSS 空间中的输出数据：

```
bcs o ls oss://demo-bucket/cli/gatk4_worker_dir/
```

查看 metadata 文件：

```
bcs o ls oss://demo-bucket/cli/gatk4_outputs/
```

6. 如何分析 30X 的全基因组数据

A) 生成配置文件

执行上述步骤生成本示例时，会同时生成一个适用 30X 全基因组数据分析的配置：

- processing-for-variant-discovery-gatk4.hg38.wgs.inputs.30x.json
- haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.30x.json

B) 修改 processing-for-variant-discovery-gatk4 配置文件

为分析 30X 样本，需要将 processing-for-variant-discovery-gatk4.hg38.wgs.inputs.30x.json 文件中的 PreProcessingForVariantDiscovery_GATK4.flowcell_unmapped_bams_list 参数改为OSS 文件路径，该文件包括了需要分析的 30X 样本在 OSS 上的路径列表。

> 注意，30X 数据样本，格式为 unaligned BAM 文件。

C) 修改 gatk4-data-processing 流程文件

找到 gatk4-data-processing 流程的 main.sh 文件，将其中的 --input_from_file_WORKFLOW_INPUTS 参数，修改为 src/processing-for-variant-discovery-gatk4.hg38.wgs.inputs.30x.json，加上 --timeout 172800 参数，并提交作业。

D) 修改 haplotypecaller-gvcf-gatk4 配置文件

- 将 haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.30x.json 中的 HaplotypeCallerGvcf_GATK4.input_bam 参数修改为gatk4-data-processing 流程输出的 bam 文件路径。
- 将 HaplotypeCallerGvcf_GATK4.input_bam_index 参数修改为相应的索引文件路径。

E) 修改 gatk4-germline-snps-indels 流程文件

找到 gatk4-germline-snps-indels 流程的 main.sh，将其中的 --input_from_file_WORKFLOW_INPUTS 参数修改为 src/haplotypecaller-gvcf-gatk4.hg38.wgs.inputs.30x.json，加上 --timeout 172800 参数，并最后提交作业。

> 如遇到 QuotaExhausted 错误，请通过工单调整 Quota。

Cromwell 工作流管理系统

Call Caching 用法详解

背景

- Cromwell 的 Call Caching 功能如何开启和关闭？
- 在一些场景下，提交工作流时不想使用 Call Caching，需要无条件执行，该如何设置？
- 工作流重新提交后，有一些 task 预期不需要重新执行，但依然执行了，Call Caching 疑似没有生效，怎么查看原因？

本篇文档将对 Call Caching 的使用做一个详细的介绍，包括功能的开启和关闭、如何通过查看元数据的方式，确认 Call Caching 未生效的原因等。

Call Caching 设置

配置文件中设置全局 Call Caching 开关状态

如果要使用 Cromwell 的 Call Caching 功能，需要在 Server 的配置文件中设置：

```
call-caching {
# Allows re-use of existing results for jobs you have already run
# (default: false)
enabled = true
# Whether to invalidate a cache result forever if we cannot reuse them. Disable this if you expect some cache copies
# to fail for external reasons which should not invalidate the cache (e.g. auth differences between users):
# (default: true)
invalidate-bad-cache-results = true
}
```

call-caching.enabled 是 Call Caching 功能的开关，可以按照自己的需求开启和关闭。

在 Option 中设置单个 Workflow 是否使用 Call Caching

在 Call Caching 功能全局开启的状态下，提交工作流时，可以通过携带如下两个 option 选项设置本次执行是

否使用 Call Caching:

```
{
"write_to_cache": true,
"read_from_cache": true
}
```

- write_to_cache: 表示本次 workflow 执行结果是否写入 Cache，实际上就是是否给后面的工作流复用。默认是 **true。**
- read_from_cache: 表示本次 workflow 执行是否从 Cache 中读取之前的结果，也就是是否复用以前的结果，默认是 **true**，如果设置为 false，表示本次执行不使用 Call Caching，强制执行。

查看元数据

工作流执行时，每一个 task 的每一个 call（对应批量计算的一个作业）都会有 metadata，记录了这个步骤的运行过程，当然也包括 Call Caching 的详细信息，通过下面的命令可以查询一个工作流的 metadata:

```
widdler query -m [WorkflowId]
```

在元数据信息中找到对应的 task 的详细信息，比如：

```
{
"callRoot": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/53cfd3fc-e9d5-4431-83ec-be6c51ab9365/call-HaplotypeCaller/shard-10",
"inputs": {
"gatk_path": "/gatk/gatk",
"ref_fasta": "oss://genomics-public-data-shanghai/broad-references/hg38/v0/Homo_sapiens_assembly38.fasta",
"cluster_config": "OnDemand ecs.sn2ne.xlarge img-ubuntu-vpc",
"input_bam_index": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/cf55a2d1-572c-4490-8edf-07656802a79b/call-GatherBamFiles/NA12878.hg38.ready.bam.bai",
"output_filename": "NA12878.hg38.vcf.gz",
"contamination": null,
"ref_fasta_index": "oss://genomics-public-data-shanghai/broad-references/hg38/v0/Homo_sapiens_assembly38.fasta.fai",
"ref_dict": "oss://genomics-public-data-shanghai/broad-references/hg38/v0/Homo_sapiens_assembly38.dict",
"interval_list": "/home/data/GATK_human_genome_resource_bundle/hg38_from_GCP/hg38_wgs_scattered_calling_intervals/temp_0047_of_50/scattered.interval_list",
"input_bam": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/cf55a2d1-572c-4490-8edf-07656802a79b/call-GatherBamFiles/NA12878.hg38.ready.bam.bam",
"docker_image": "registry.cn-shanghai.aliyuncs.com/wgs_poc/poc:4.0.10.1"
},
"returnCode": 0,
"callCaching": {
"allowResultReuse": true,
"hashes": {
"output expression": {
"File output_vcf_index": "A162250CB6F52CC32CB75F5C5793E8BB",
"File output_vcf": "7FD061EEA1D3C63912D7B5FB1F3C5218"
```

```
},
"runtime attribute": {
"userData": "N/A",
"docker": "F323AFFA030FBB5B352C60BD7D615255",
"failOnStderr": "68934A3E9455FA72420237EB05902327",
"imageId": "N/A",
"continueOnReturnCode": "CFCD208495D565EF66E7DFF9F98764DA"
},
"output count": "C81E728D9D4C2F636F067F89CC14862C",
"input count": "D3D9446802A44259755D38E6D163E820",
"command template": "9104DF40289AB292A52C2A753FBF58D2",
"input": {
"File interval_list": "04dc2cb895d13a40657d5e2aa7d31e8c",
"String output_filename": "2B77B986117FC94D088273AD4D592964",
"File ref_fasta": "9A513FB0533F04ED87AE9CB6281DC19B-400",
"File input_bam_index": "D7CA83047E1B6B8269DF095F637621FE-1",
"String gatk_path": "EB83BBB666B0660B076106408FFC0A9B",
"String docker_image": "0981A914F6271269D58AA49FD18A6C13",
"String cluster_config": "B4563EC1789E5EB82B3076D362E6D88F",
"File ref_dict": "3884C62EB0E53FA92459ED9BFF133AE6",
"File input_bam": "9C0AC9A52F5640AA06A0EBCE6A97DF51-301",
"File ref_fasta_index": "F76371B113734A56CDE236BC0372DE0A"
},
"backend name": "AE9178757DD2A29CF80C1F5B9F34882E"
},
"effectiveCallCachingMode": "ReadAndWriteCache",
"hit": false,
"result": "Cache Miss"
},
"stderr": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/53cfd3fc-e9d5-4431-83ec-
be6c51ab9365/call-HaplotypeCaller/shard-10/stderr",
"shardIndex": 10,
"stdout": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/53cfd3fc-e9d5-4431-83ec-
be6c51ab9365/call-HaplotypeCaller/shard-10/stdout",
"outputs": {
"output_vcf": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/53cfd3fc-e9d5-4431-83ec-
be6c51ab9365/call-HaplotypeCaller/shard-10/NA12878.hg38.vcf.gz",
"output_vcf_index": "oss://gene-test/cromwell_test/GATK4_VariantDiscovery_pipeline_hg38/53cfd3fc-e9d5-4431-
83ec-be6c51ab9365/call-HaplotypeCaller/shard-10/NA12878.hg38.vcf.gz.tbi"
},
"commandLine": "set -e\n\n /gatk/gatk --java-options \"-Xmx4g -Xmx4g\" \\\n HaplotypeCaller \\\n -R
/cromwell_inputs/73a7571e/Homo_sapiens_assembly38.fasta \\\n -I
/cromwell_inputs/02f1b5ca/NA12878.hg38.ready.bam.bam \\\n -L
/home/data/GATK_human_genome_resource_bundle/hg38_from_GCP/hg38_wgs_scattered_calling_intervals/temp_0
047_of_50/scattered.interval_list \\\n -O NA12878.hg38.vcf.gz \\\n -contamination 0",
"attempt": 1,
"jobId": "job-000000005DB051A800006F970001CAC8",
"start": "2019-10-25T02:38:03.522Z",
"backendStatus": "Finished",
"runtimeAttributes": {
"cluster": "Right(AutoClusterConfiguration(OnDemand,ecs.sn2ne.xlarge,img-ubuntu-vpc,None,None,None))",
"continueOnReturnCode": "0",
"failOnStderr": "false",
"vpc": "BcsVpcConfiguration(Some(10.20.200.0/24),Some(vpc-uf61zj30k0ebuen0xi7ci))",
"mounts": "BcsInputMount(Right(nas://10.20.66.4:/data/ali_yun_test/),Left(/home/data),true)",
"docker": "BcsDockerWithoutPath(registry.cn-shanghai.aliyuncs.com/wgs_poc/poc:4.0.10.1)",
```

```
"autoReleaseJob": "false",
"maxRetries": "0"
},
"executionStatus": "Done",
"end": "2019-10-25T03:22:23.481Z",
"executionEvents": [
{
"endTime": "2019-10-25T03:22:21.626Z",
"description": "RunningJob",
"startTime": "2019-10-25T02:38:03.645Z"
},
{
"endTime": "2019-10-25T03:22:22.481Z",
"description": "UpdatingCallCache",
"startTime": "2019-10-25T03:22:21.626Z"
},
{
"endTime": "2019-10-25T02:38:03.645Z",
"description": "CallCacheReading",
"startTime": "2019-10-25T02:38:03.643Z"
},
{
"endTime": "2019-10-25T02:38:03.522Z",
"description": "Pending",
"startTime": "2019-10-25T02:38:03.522Z"
},
{
"endTime": "2019-10-25T02:38:03.542Z",
"description": "WaitingForValueStore",
"startTime": "2019-10-25T02:38:03.542Z"
},
{
"endTime": "2019-10-25T03:22:23.481Z",
"description": "UpdatingJobStore",
"startTime": "2019-10-25T03:22:22.481Z"
},
{
"endTime": "2019-10-25T02:38:03.643Z",
"description": "PreparingJob",
"startTime": "2019-10-25T02:38:03.542Z"
},
{
"endTime": "2019-10-25T02:38:03.542Z",
"description": "RequestingExecutionToken",
"startTime": "2019-10-25T02:38:03.522Z"
}
],
"backend": "BCS"
}
```


在上面的元数据中，有一项 callCaching，主要记录了如下信息：

- allowResultReuse：是否允许其他工作流复用。
  - 如果当前工作流设置了不允许写入 Cache，则不可以复用
  - 如果当前工作流设置了允许写入 Cache，则只有任务执行成功，才允许复用

- hashes：当前任务的输入、输出、运行时等参数的 hash 记录，用于比对两次运行条件是否一样。
- effectiveCallCachingMode：Call Caching 的模式，比如是否从 Cache 中读取，或者是否写入 Cache 等。
- hit：当前任务在 Cache 是否命中。
- result：当前任务在 Cache 中命中的详情，比如哪个工作流的哪个 task 的哪个 shard。

综合上面的解释，我们看到实例中的这个 call, 是 GATK4_VariantDiscovery_pipeline_hg38 这个工作流的 HaplotypeCaller 这个 task 的10号 shard，Call Cache 情况如下：

- 未在 Cache 中命中，完整的执行了一次
- 执行成功，可以允许后的流程复用

Call Caching 未生效问题排查

如果遇到不符合预期的 task，可以通过如下步骤排查原因：

- 查看当前 workflow 重新执行的 task 的 Call Caching 元数据
    - 如果当前 task 的 Call Caching 的模式是不使用Cache（可能是提交作业时设置了不使用 Call Caching 的选项），则不会去利用之前的结果，确实会强制重新执行，是符合预期的
    - 如果当前 task 未命中 Cache，则需要查看之前的 workflow, 进一步确认未命中的原因
- 查看之前的 workflow 的 task 的 CalCaching 元数据，确认之前的 task 是否执行成功，是否可以复用
    - 如果之前的 task 的 不允许复用，可能是执行失败了，或者虽然执行成功，但 Cache 模式设置的不写入 Cache，即不允许复用
    - 如果之前的 task 允许复用，但未命中，则需要比较两次的 hash 记录，可能是由于 Call Caching 相关的参数变化引起的

使用 docker-compose 一键启停 Cromwell

背景

Cromwell server 的启动需要以下组件配合：

- 启动 Mysql 的 docker 容器作为 Crowmell 的持久化数据库，包括配置用户名，密码等
- 填写 Cromwell 配置文件，包括 BCS 后端配置及数据库等配置
- 使用 Cromwell 的 jar 包，启动 server

实际上 Cromwell 除了发布 jar 包，也会发布对应的 docker 镜像，我们可以考虑使用 docker-compose来简

化以上步骤。docker-compose 是 Docker 官方的开源项目，其定位是定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）。

使用docker-compose 可以将容器化的 Cromwell 和 Mysql 两个 service 拉起，作为一个应用来运行。再配合脚本来简化配置，可以将 Cromwell 的服务做成一键启停。

开通 ECS 作为 Crowmell server

首先使用 Cromwell server 镜像开通一台 ECS，ssh 登入机器后，可以运行目录下的cromwell-server.sh，进行Cromwell Server的管理：

```
# ./cromwell-server.sh
cromwell-server.sh - init cromwell config and start/stop service
Usage: cromwell-server.sh [options...] [init/start/stop/status]
Options:
--id=STRING Access id
--key=STRING Access key
--root=STRING Oss root for cromwell, e.g: oss://my-bucket/cromwell/
--instance=STRING default runtime: instance type [ecs.sn1.medium]
--image=STRING default runtime: image id [img-ubuntu-vpc]
```

第一次配置与启动服务

初次使用，需要做一些初始配置，可以使用下面的命令完成一键初始化与启动：

```
./cromwell-server.sh init --id=LTAI8xxxxx --key=vVGZVE8qUNjxxxxxxxx --root=oss://gtx-wgs-demo/cromwell/
```

上面的命令完成了以下配置：

- --id: 批量计算的 Access Id
- --key: 批量计算的 Access Key
- --root: Crowmell 运行时在 OSS 上的工作根目录
- --instance: Cromwell 默认运行时参数，实例类型
- --image: Cromwell 默认运行时参数，镜像ID执行完以上命令后，会根据 Crowmell 配置文件模板生成配置文件，并通过 docker-compose 启动 Cromwell server，并在后台运行。

服务启动后，就可以通过镜像中的命令行工具 widdler 执行工作流的提交：

```
cd /home/cromwell/cromwell/
widdler run echo.wdl inputs.json -o bcs_workflow_tag:test_echo
```

停止服务

使用下面的命令可以一键停止服务：

```
./cromwell-server stop
```

再次启动服务

在已经完成配置的情况下，使用下面的命令，可以完成服务启动：

```
./cromwell-server start
```

重新配置并启动服务

如果需要修改配置，在服务停止的情况下，再次使用 init 命令可以完成新配置重新启动：

```
./cromwell-server.sh init --id=LTAI8xxxxx --key=vVGZVE8qUNjxxxxxxxx --root=oss://gtx-wgs-demo/cromwell/
```

使用option文件设置默认运行时参数

在 Crowmell 的配置文件中，可以设置每个 backend 的默认运行时参数 default-runtime-attibutes，也可以在提交工作流时通过 option 覆盖原有设置。

所以如果您在提交工作流时用到了数据盘、NAS等，都可以在 option 文件中设置：

```
{
"default_runtime_attributes": {
"vpc": "192.168.0.0/24",
"autoReleaseJob": true,
"mounts": "nas://1f****04-xkv88.cn-beijing.nas.aliyuncs.com:/ /mnt/ true",
"dataDisk": "cloud_ssd 250 /home/mount/"
},
"bcs_workflow_tag": "Tagxxx",
"read_from_cache": true
}
```

使用 widdler 命令行的 -O （大写的O）参数提交 option 文件:

```
widdler run echo.wdl inputs.json -O options.json
```

cromwell-server运维

在 Cromwell Server 配置完成后，如何快捷的进行提交、停止工作流、流程失败后如何快速定位问题以及工作流完成后如何如何快速查看运行日志、查看工作流的 Metrics 信息、工作流产生的费用等手段，这些问题就变成了 server 运维工作的基本诉求。

Cromwell 以 server 的方式运行后是支持通过 API 接口获取以上信息的，但是有二次开发的工作量；而 widdler 是针对 Cromwell Server API 接口开发的命令行工具，通过 widdler 命令行工具减少运维成本。

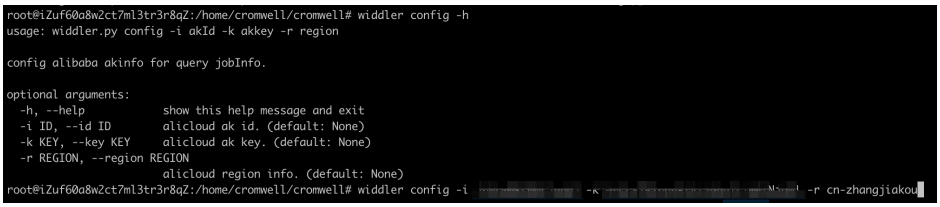
本文主要介绍阿里云对 widdler 工具的扩展，通过 widdler 工具查询指定工作流的作业运行状态、后端引擎的运行时信息、费用查询、问题定位调查以及子任务日志查看等功能。

1. widdler 安装

widdler 默认在阿里云批量计算提供的 Cromwell server 镜像中安装。可以直接使用，无需做安装操作。

2. 配置 widdler

由于涉及到个人阿里云运行数据的查询，需要在使用之前设置对应账号的 AK 信息、以及后端执行引擎所部署的region信息。



命令格式：

```
widdler config -i id -k key -r cn-zhangjiakou
```

3. 校验 WDL

提交工作流之前对 WDL 做语法校验，排除部分低级问题；减少后续提交工作流后由于低级问题导致的流程失败。

```
root@iZuf60a8w2ct7ml3tr3r8qZ:/home/cromwell/cromwell# widdler validate echo.wdl inputs.json
No errors found in echo.wdl
null
root@iZuf60a8w2ct7ml3tr3r8qZ:/home/cromwell/cromwell#
```

命令格式：

```
widdler validate echo.wdl inputs.json
```

4. 提交工作流

```
root@iZuf60a8w2ct7ml3tr3r8qZ:/home/cromwell/cromwell# widdler run -h
usage: widdler.py run <wdl file> <json file> [<args>]

Submit a WDL & JSON for execution on a Cromwell VM.

positional arguments:
  wdl                   Path to the WDL to be executed.
  json                  Path the json inputs file.

optional arguments:
  -h, --help            show this help message and exit
  -O OPTION, --option OPTION
                        Path to the json option file. (default: None)
  -v, --validate        Validate WDL inputs in json file. (default: False)
  -l LABEL, --label LABEL
                        A key:value pair to assign. May be used multiple
                        times. (default: None)
  -m, --monitor         Monitor the workflow and receive an e-mail
                        notification when it terminates. (default: False)
  -i INTERVAL, --interval INTERVAL
                        If --monitor is selected, the amount of time in
                        seconds to elapse between status checks. (default: 30)
  -o EXTRA_OPTIONS, --extra_options EXTRA_OPTIONS
                        Additional workflow options to pass to Cromwell.
                        Specify as k:v pairs. May be specified multipletimes
                        for multiple options. See
                        https://github.com/broadinstitute/cromwell#workflow-
                        optionsfor available options. (default: None)
  -V, --verbose         If selected, widdler will write the current status to
                        STDOUT until completion while monitoring. (default:
                        False)
  -n, --no_notify       When selected, disable widdler e-mail notification of
                        workflow completion. (default: False)
  -d DEPENDENCIES, --dependencies DEPENDENCIES
                        A zip file containing one or more WDL files that the
                        main WDL imports. (default: None)
  -b BUCKET, --bucket BUCKET
                        Name of bucket where files were uploaded. Default is
                        None (default: None)
  -D, --disable_caching
                        Don't used cached data. (default: False)
  -S {localhost}, --server {localhost}
                        Choose a cromwell server from ['localhost'] (default:
                        localhost)
root@iZuf60a8w2ct7ml3tr3r8qZ:/home/cromwell/cromwell# widdler run echo.wdl inputs.json -l test
```

命令格式：

```
widdler run echo.wdl inputs.json -l test
```

其中：test 为 label，可以根据样本进行打标签，后续可以按 label 做过滤。

5. 终止工作流

命令格式：

```
widdler abort workflowId
```

```
root@iZuf60a8w2ct7ml3tr3r8qZ:~# widdler abort -h
usage: widdler.py abort <workflow id>

Abort a submitted workflow.

positional arguments:
  workflow_id           workflow id of workflow to abort.

optional arguments:
  -h, --help            show this help message and exit
  -S {localhost}, --server {localhost}
                        Choose a cromwell server from ['localhost'] (default:
                        localhost)
root@iZuf60a8w2ct7ml3tr3r8qZ:~#
```

6. 获取工作流

6.1 获取工作流列表

命令格式：

```
widdler query
```

```
root@iZuf60a8w2ct7ml3tr3r8qZ:~# widdler query -h
usage: widdler.py query <workflow id> [<args>]

Query cromwell for information on the submitted workflow.

positional arguments:
  workflow_id           workflow id for workflow execution of interest.
                        (default: None)

optional arguments:
  -h, --help            show this help message and exit
  -s, --status          Print status for workflow to stdout (default: False)
  -m, --metadata        Print metadata for workflow to stdout (default: False)
  -l, --logs            Print logs for workflow to stdout (default: False)
  -u USERNAME, --username USERNAME
                        Owner of workflows to query. (default: root)
  -L LABEL, --label LABEL
                        Query status of all workflows with specific label(s).
                        (default: None)
  -d DAYS, --days DAYS  Last n days to query. (default: 7)
  -S {localhost}, --server {localhost}
                        Choose a cromwell server from ['localhost'] (default:
                        localhost)
  -f {Running,Submitted,QueuedInCromwell,Failed,Aborted,Succeeded}, --filter {Running,Submitted,QueuedInCromwell,Failed,Aborted,Succeeded}
                        Filter by a workflow status from those listed above.
                        May be specified more than once. (default: None)
  -a, --all             Query for all users. (default: False)
root@iZuf60a8w2ct7ml3tr3r8qZ:~#
```

其中：默认获取当前 user 7 天内的工作流信息；可以根据 user label等信息来筛选工作流信息；其他使用方法参考 help 信息。

6.2 获取工作流 Meta

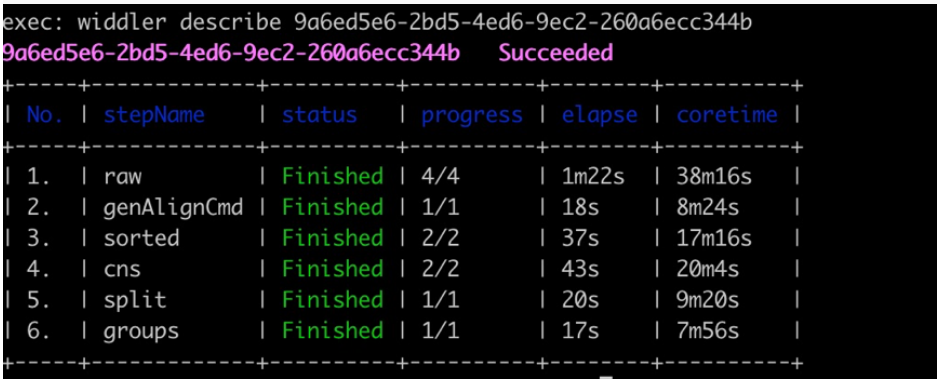
命令格式：

```
widdler query workflowId
```

7. 获取工作流运行状态

命令格式：

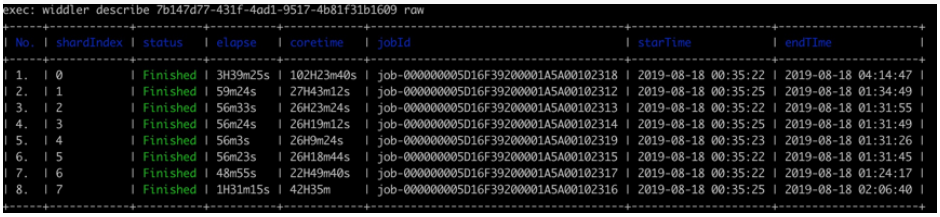
```
widdler describe workflowId
```



其中：” stepName” 表示工作流的某个自步骤的名称；” status” 表示当前步骤的运行状态”成功、失败、运行中” ；” progress” 表示当前步骤的进度，如” 4/4” 表示当前步骤存在4个子任务全部执行完成; 若是” 2/4” 则表示当前步骤存在4个任务，已经完成2个。” elapse”表示当前步骤的所有子任务执行总耗时时间(不包括机器的启动时间); “coretime”表示当前步骤所有子任务消耗的核时时间(不包括机器的启动时间)。

命令格式：

```
widdler describe workflowId -t stepName
```

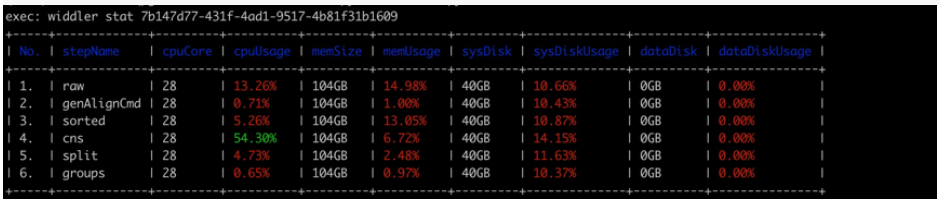


shardIndex 是 Cromwell 将每个 task 的子任务按 shardIndex 做索引，对应的是批量计算的一个作业。通过该命令可以看到指定 task 对应的统计信息。

8. 获取工作流统计信息

命令格式：

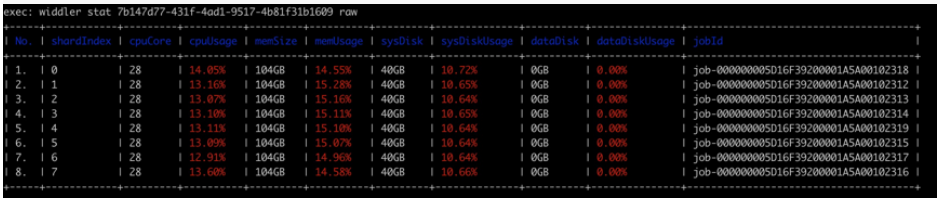
```
widdler stat workflowId
```



其中：” cpuCore” 表示当前步骤中使用对应实例的 CPU 核数，” cpuUsage” 表示当前步骤所有任务从开始到当前（若当前任务结束状态则表示从开始到结束）的 CPU 平均利用率；” memSize” 表示当前步骤中使用对应实例的内存大小，” memUsage” 表示当前步骤中所有任务从开始到当前的MEM平均利用率；” sysDisk” 表示当前步骤实例的系统盘大小（默认 40GB），” sysDiskUsage” 表示当前步骤的所有任务在当前时间点的磁盘平均利用率；” dataDisk” 表示实例的数据盘大小（默认没有），” dataDiskUsage” 表示当前步骤的所有任务在当前时间点的磁盘平均利用率。

命令格式：

```
widdler stat workflowId -t stepName
```

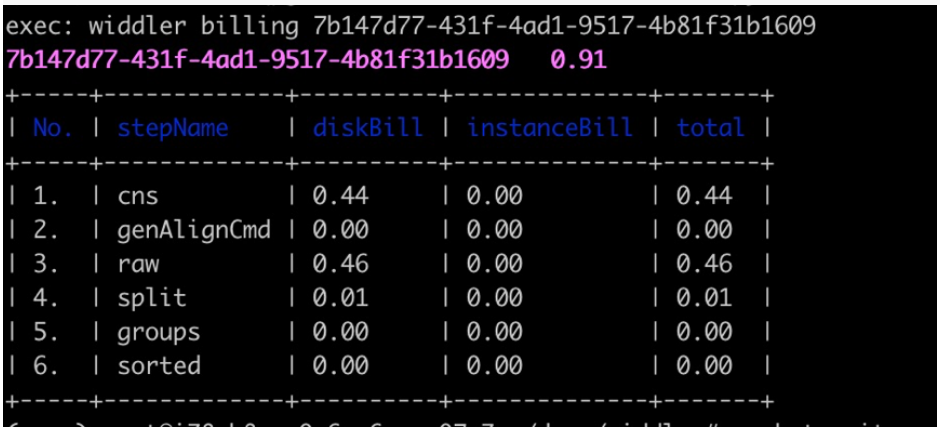


查询某个步骤对应的 Metrics 信息；可能某个步骤存在多个 scatter，那么每个 scatter 运行情况如何，则可以通过本命令获取到。

9. 获取工作流费用

命令格式：

```
widdler billing workflowId
```



10. 获取工作流运行日志

命令格式：

```
widdler log workflowId
```

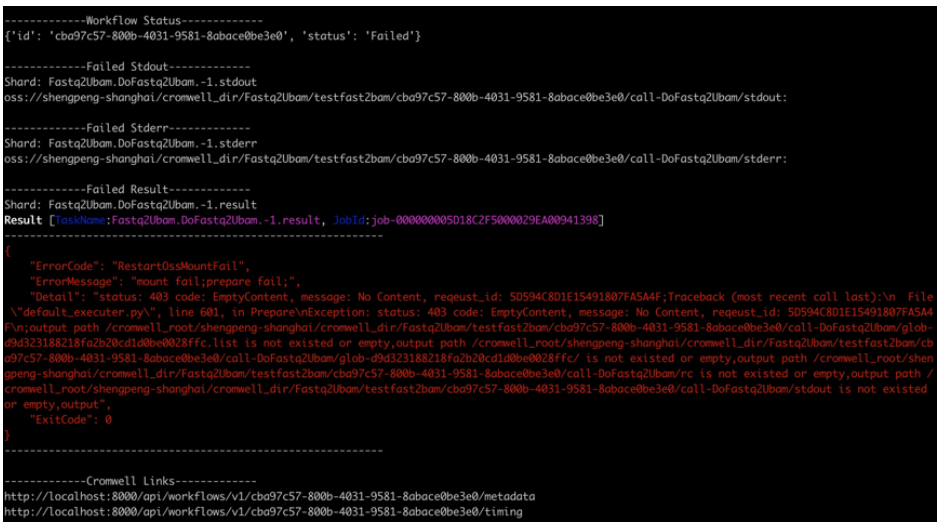
通过 log 查询命令，可以查看工作流的实际运行情况，执行过程是否符合预期可以通过该命令做到一键查看。stdout 以及 stderr 日志小于 1MB 的，会直接在屏幕上显示出来；超过 1MB 的需要借助 OSS 工具查看。

11. 工作流问题定位

命令格式：

```
widdler explain workflowId
```

通过该命令可以一键查询工作流失败的原因，展示出现问题的步骤，输出该步骤的对应失败任务的 stdout 以及 stderr 信息，快速排查问题。



- 更多其他功能请参考 widdler 的帮助信息

AttachCluster最佳实践

0 背景

AttachCluster作业是批量计算最新推出的作业类型。它结合了固定集群作业和AutoCluster作业的优势，既能自动管理集群生命周期，弹性伸缩资源，又能使用分布式缓存节省资源。本文的目的在于介绍在阿里云批量计算服务上运行AttachCluster作业。

1 准备工作

1.1 开通阿里云批量计算服务

要使用批量计算服务，请根据官方文档里面的指导开通批量计算和其依赖的相关服务，如OSS等。

1.2 升级Python SDK

若您未安装批量计算Python SDK，请您参照安装方法安装该SDK。如果您检查已经安装之后，请您参照Python SDK升级方法， 升级批量计算Python SDK至最新版。

2 创建集群

AttachCluster作业首次使用时，需要创建一个集群，创建方法可参考官方文档 。该集群对配置没有特殊需求

，实例数可设置为0。以下是创建集群的Python源代码。

```
import time
import random
import string
import batchcompute
from batchcompute import CN_SHENZHEN as REGION
from batchcompute import Client, ClientError
from batchcompute.resources import (
JobDescription, TaskDescription, DAG,
GroupDescription, ClusterDescription,
Configs, Networks, VPC, Classic, Mounts, Notification, Topic
)
ACCESS_KEY_ID = 'Your Access Key Id'
ACCESS_KEY_SECRET = 'Your Access Key Secret'
IMAGE_ID = 'img-ubuntu'
INSTANCE_TYPE = 'ecs.sn2ne.large'
client = Client(REGION, ACCESS_KEY_ID, ACCESS_KEY_SECRET)
def create_cluster(idempotent_token=''):
try:
# Cluster description.
cluster_desc = ClusterDescription()
cluster_desc.Name = "test-cluster"
cluster_desc.Description = "demo"
cluster_desc.ImageId = IMAGE_ID
cluster_desc.InstanceType = INSTANCE_TYPE
#Group description
group_desc1 = GroupDescription()
group_desc1.DesiredVMCount = 4
group_desc1.InstanceType = 'ecs.sn1ne.large' #user group special instance type
group_desc1.ResourceType = 'OnDemand'
cluster_desc.add_group('group1', group_desc1)
#cluster_desc.add_group('group2', group_desc2)
#Configs
configs = Configs()
#Configs.Disks
configs.add_system_disk(50, 'cloud_efficiency')
configs.add_data_disk(500, 'cloud_efficiency', '/home/my-data-disk')
#Configs.Networks
networks = Networks()
vpc = VPC()
vpc.CidrBlock = '192.168.0.0/16'
#vpc.VpcId = 'vpc-xxxxx'
networks.VPC = vpc
configs.Networks = networks
cluster_desc.Configs = configs
print cluster_desc
rsp = client.create_cluster(cluster_desc, idempotent_token)

# get cluster id for attach cluster job
return rsp.Id

except ClientError, e:
print (e.get_status_code(), e.get_code(), e.get_requestid(), e.get_msg())
return ""
```

```
if __name__ == '__main__':
#Not Use idempotent token
cluster_id = create_cluster()
print cluster_id
```

3 创建作业

在创建作业的时候需要步骤2中的集群Id，填入task的AutoCluster的ClusterId字段中。以下是创建作业的Python源代码。

```
from batchcompute import Client, ClientError
from batchcompute import CN_ZHANGJIAKOU as REGION
from batchcompute.resources import (
ClusterDescription, GroupDescription, Configs, Networks, VPC,
JobDescription, TaskDescription, DAG,Mounts,
AutoCluster,Disks,Notification,
)
access_key_id = "" # your access key id
access_key_secret = "" # your access key secret
image_id = "m-8vbd8lo9xxxx" # the id of a image created before,镜像需要确保已经注册给批量计算
instance_type = "ecs.sn1.medium" # instance type
inputOssPath = "oss://xxx/input/" # your input oss path
outputOssPath = "oss://xxx/output/" #your output oss path
stdoutOssPath = "oss://xxx/log/stdout/" #your stdout oss path
stderrOssPath = "oss://xxx/log/stderr/" #your stderr oss path
def getAutoClusterDesc():
auto_desc = AutoCluster()
# attach cluster这里里填入上一步创建的集群Id
auto_desc.ClusterId = cls-xxxxx

auto_desc.ECSImageId = image_id
auto_desc.ReserveOnFail = False
# 实例规格
auto_desc.InstanceType = instance_type
#case1 设置上限价格的竞价实例;
# auto_desc.ResourceType = "Spot"
# auto_desc.SpotStrategy = "SpotWithPriceLimit"
# auto_desc.SpotPriceLimit = 0.5
#case2 系统自动出价，最高按量付费价格
# auto_desc.ResourceType = "Spot"
# auto_desc.SpotStrategy = "SpotAsPriceGo"
#case3 按量
auto_desc.ResourceType = "OnDemand"
#Configs
configs = Configs()
#Configs.Networks
networks = Networks()
vpc = VPC()
#case1 只给CidrBlock
vpc.CidrBlock = '192.168.0.0/16'
#case2 CidrBlock和VpcId 都传入，必须保证VpcId的CidrBlock 和传入的CidrBlock保持一致
# vpc.CidrBlock = '172.26.0.0/16'
# vpc.VpcId = "vpc-8vbfxdyhxxxx"
```

```
networks.VPC = vpc
configs.Networks = networks
# 设置系统盘type(cloud_efficiency/cloud_ssd)以及size(单位GB)
configs.add_system_disk(size=40, type_='cloud_efficiency')
#设置数据盘type(必须和系统盘type保持一致) size(单位GB) 挂载点
# case1 linux环境
# configs.add_data_disk(size=40, type_='cloud_efficiency', mount_point='/path/to/mount/')
# case2 windows环境
# configs.add_data_disk(size=40, type_='cloud_efficiency', mount_point='E:')
# 设置节点个数
configs.InstanceCount = 1
auto_desc.Configs = configs
return auto_desc

def getDagJobDesc(clusterId = None):
job_desc = JobDescription()
dag_desc = DAG()
mounts_desc = Mounts()
job_desc.Name = "testBatchSdkJob"
job_desc.Description = "test job"
job_desc.Priority = 1
# 订阅job完成或者失败事件
noti_desc = Notification()
noti_desc.Topic['Name'] = "test-topic"
noti_desc.Topic['Endpoint'] = "http://[UserId].mns.[Region].aliyuncs.com/"
noti_desc.Topic['Events'] = ["OnJobFinished", "OnJobFailed"]
# job_desc.Notification = noti_desc
job_desc.JobFailOnInstanceFail = False
# 作业运行成功后户自动会被立即释放掉
job_desc.AutoRelease = False
job_desc.Type = "DAG"
echo_task = TaskDescription()
# echo_task.InputMapping = {"oss://xxx/input/": "/home/test/input/",
# "oss://xxx/test/file": "/home/test/test/file"}
echo_task.InputMapping = {inputOssPath: "/home/test/input/"}
echo_task.OutputMapping = {"/home/test/output/":outputOssPath}
#触发程序运行的命令行
#case1 执行linux命令行
echo_task.Parameters.Command.CommandLine = "/bin/bash -c 'echo BatchcomputeService'"
#case2 执行Windows CMD.exe
# echo_task.Parameters.Command.CommandLine = "cmd /c 'echo BatchcomputeService'"
#case3 输入可执行文件
# PackagePath存放commandLine中的可执行文件或者二进制包
# echo_task.Parameters.Command.PackagePath = "oss://xxx/package/test.sh"
# echo_task.Parameters.Command.CommandLine = "sh test.sh"
# 设置程序运行过程中相关环境变量信息
echo_task.Parameters.Command.EnvVars["key1"] = "value1"
echo_task.Parameters.Command.EnvVars["key2"] = "value2"
# 设置程序的标准输出地址，程序中的print打印会实时上传到指定的oss地址
echo_task.Parameters.StdoutRedirectPath = stdoutOssPath
# 设置程序的标准错误输出地址，程序抛出的异常错误会实时上传到指定的oss地址
echo_task.Parameters.StderrRedirectPath = stderrOssPath
# 设置任务的超时时间
echo_task.Timeout = 600
# 设置任务所需实例个数
# 环境变量BATCH_COMPUTE_INSTANCE_ID为0到InstanceCount-1
```

```
# 在执行程序中访问BATCH_COMPUTE_INSTANCE_ID，实现数据访问的切片实现单任务并发执行
echo_task.InstanceCount = 1
# 设置任务失败后重试次数
echo_task.MaxRetryCount = 0
# NAS数据挂载
#采用NAS时必须保证网络和NAS在同一个VPC内
nasMountEntry = {
"Source": "nas://xxxx.nas.aliyuncs.com:/",
"Destination": "/home/mnt/",
"WriteSupport":True,
}
mounts_desc.add_entry(nasMountEntry)
mounts_desc.Locale = "utf-8"
mounts_desc.Lock = False
# echo_task.Mounts = mounts_desc

# attach cluster作业该集群字段设置为空
echo_task.ClusterId = ""

echo_task.AutoCluster = getAutoClusterDesc()
# 添加任务
dag_desc.add_task('echoTask', echo_task)
# 可以设置多个task,每个task可以根据需求进行设置各项参数
# dag_desc.add_task('echoTask2', echo_task)
# Dependencies设置多个task之间的依赖关系,echoTask2依赖echoTask;echoTask3依赖echoTask2
# dag_desc.Dependencies = {"echoTask":["echoTask2"], "echoTask2":["echoTask3"]}
job_desc.DAG = dag_desc
return job_desc

if __name__ == "__main__":
client = Client(REGION, access_key_id, access_key_secret)
try:
job_desc = getDagJobDesc()
job_id = client.create_job(job_desc).Id
print('job created: %s' % job_id)
except ClientError,e:
print (e.get_status_code(), e.get_code(), e.get_requestid(), e.get_msg())
```

AttachCluster作业创建已经完成。

GTX_FPGA最佳实践

gtx-fpga

介绍

GTX-FPGA产品是由未来实验室 GTX-Laboratory开发的全基因组分析加速工具，采用CPU和FPGA协同工作的异构加速技术，利用各自的特性进行基因数据的高性能计算。可以将30X的全基因组数据分析时间从30小时缩短至30分钟；将100X全外显子数据分析时间从6小时缩短至5分钟完成。

GTX-FPGA 分析主要包括：index(构建索引)、align(基因组对比)、 vc(突变检测)、wgs(整合，将alin 和 vc 整合到一起，下文中的 GTX one也是指该步骤)等步骤。

本文主要介绍如果通过阿里云批量计算直接使用 GTX-FPGA 产品，实现全基因组数据分析、全外显子数据分析作业一键式运行。

使用约束

- GTX-FPGA 产品目前只支持阿里云 F3 型 ECS 实例类型。同时每个实例类型需要配置一定容量的 SSD 数据盘，容量大小和fasta大小有关；其中 align 需要的磁盘大小是 2 个 fastq 文件大小的和再乘以 2（例如：需要计算的 fastq1 是 40G，fastq2 是 42G, 需要的数据盘空间大小是 164G）；wgs需要的计算空间大小是 fasta 文件大小的8倍（例如human30x.fasta数据大小是 3.4G,则需要的数据盘大小是 252G）。针对人类基因组数据盘大小可以采用下文中 demo 示例的设置默认值。
- GTX-FPGA 产品目前只支持 **北京** 区域测试。
- GTX-FPGA 产品目前处于公测阶段，公测阶段 GTX-FPGA 产品不收取费用，只收取作业所需要的实例以及相关存储费用。

前置条件

- 登录阿里云，并确保账号余额大于100元，以便体验完整分析流程。
- 开通批量计算服务，用于执行分析任务。
- 开通OSS对象存储, 用于上传用户自己的测序数据，保存分析结果。创建bucket，例如 gtx-wgs-demo
- 查看或者创建的AccessKey, 如果您使用的是**子账号**，请确认具有以上批量计算和OSS的产品使用权限，参考快速开始文档。复制AccessKey ID（如LTAI8xxxxx), Access Key Secret(如 vVGZVE8qUNjxxxxxxxx) 备用。

使用说明

GTX-FPGA 支持WDL模式运行以及DAG作业模式运行。

1 GTX 命令格式

指令	参数简称	参数说明
index	-f	强行覆盖已有的索引文件
	-h	打印帮助文档
	-m	指定中间临时文件的路径 默认是/ssd-cache
	- disable-gtx-index	不创建gtx索引文档
	--disable-bwa-index	不创建bwa的索引文档
	--enable-bwa2-index	创建mem2的索引文档
align	-o	输出bam的文件
	-R	read group 的头部信息 默认是'@RG\\tID:foo\\tSM:bar'\n"
	-A	匹配(match)得分，默认为1匹配(match) 得分，默认为1
	-B	错配(mismatch)罚分，默认为4
	-o	gap open罚分，默认为6
	-E	gap extension罚分，默认为1
	-t	开启线程数量，默认 32（一体机环境下为最佳性能）
	--bwa	加此参数的比对结果准确性可以 和BWA-mem的结果媲美
	--disable-mark-duplicate	不做去重处理
wgs	-o	输出vcf的文件
	-b	输出bam的文件
	-R	read group 的头部信息 默认是'@RG\\tID:foo\\tSM:bar'\n"
	-A	匹配(match)得分，默认为1匹配(match) 得分，默认为1
	-B	错配(mismatch)罚分，默认为4
	-o	gap open罚分，默认为6
	-E	gap extension罚分，默认为1
	-t	开启线程数量，默认 32（一体机环境下为最佳性能）
	-L	指定一个染色体（eg.chr1:1-200）或者多个 染色体(bed文件)进行计算
	-g	输出gvcf格式文件
	--bwa	加此参数的比对结果准确性可以
	--disable-mark-duplicate	不做去重处理
	--metrics	输出去重过程中的相关参数
vc	-o	输出vcf的文件
	-r	fasta文件
	-i	输入排序和去重以后的bam文件
	-t	开启线程数量，默认 32（一体机环境下为最佳性能）
	-L	指定一个染色体（eg.chr1:1-200）或者多个 染色体(bed文件)进行计算
	-g	输出gvcf格式文件
	--gtz-rbin1	这个参数代表当输入文件fastq1是gtz 文件时，需要用到的rbin进行解压计算
	--gtz-rbin2	文件时，需要用到的rbin进行解压计算rbin文件 说请参考gtz的官方文档

2 WDL模式运行

WDL 模式使用方式请参考文档

3 DAG作业模式

3.1 示例脚本

下载 DAG 作业示例代码。

其中：

genGtxIndexCmd 则是对应 GTX 的建索引命令；命令使用方法可以参考代码中帮助信息。
genGtxWgsCmd 则是对应 GTX one的命令；命令使用方法可以参考代码中帮助信息。
genGtxAlignCmd 则是对应 GTX 基因组对比命令；命令使用方法可以参考代码中帮助信息。
genGtxVcCmd 则是对应 GTX 突变检测命令；命令使用方法可以参考代码中帮助信息。

- 可以自定义以上步骤中每项 GTX 参数，也可以按默认值来执行。
- 建索引操作是非必选项目，本 demo 示例默认索引构建完成；若需要构建索引在执行脚本时需要增加参数(isNeedIndex)描述。
- read_group_header 可以通过命令行传入也可以使用默认值。
- 示例代码默认运行 GTX one流程，一次性执行对比以及编译检测流程；若需要按分步骤执行则需要设置对应参数。
- 更新批量计算python SDK到最新版本 “pip install —upgrade batchcompute”。

3.1 执行命令

```
python test.py --reference oss://xxx/ref/hg19.fa --fastq1 oss://xxx/input/human30x_10m_1.fastq --fastq2 oss://xxxx/_input/human30x_10m_2.fastq --output oss://xxx/testoutput/
```

3.2 执行结果

名称	类型 / 大小	最后修改时间
ret	目录	
stderr	目录	
stdout	目录	

名称	类型 / 大小	最后修改时间
hg19.fa.mapping_metrics.txt	384B	2020-01-17 00:37:22
hg19.fa.markdup_metrics.txt	203B	2020-01-17 00:39:42
hg19.fa.vcf	11.51MB	2020-01-17 00:39:42