
MegFlow

MegEngine

2022 年 03 月 04 日

1	Building with docker	3
2	Building from Source	5
3	aarch64 源码编译	9
4	Build on win10	11
5	创建 rtsp 流地址	13
6	Run in 15 minutes	17
7	模型和测试数据下载	21
8	megflow_quickstart	23
9	串联检测和分类	29
10	批量推理和 Pipeline 级测试	33
11	视频结果 Web 可视化	37
12	Config	41
13	Python Plugins	49
14	生成 MegEngine 模型	51
15	FAQ	55
16	如何 Debug 常见问题	57
17	如何提交代码	59

Please select a specific version of the document in the lower left corner of the page.

docker build 方式能够“可复现地”生成运行环境、减少依赖缺失的痛苦

1.1 下载模型包

docker 运行方式，建议把模型包下好，解压备用。[下载地址](#)

1.2 编译 Docker 镜像

```
$ cd MegFlow
$ docker build -t megflow -f Dockerfile.github-dev .
```

稍等一段时间（取决于网络和 CPU）镜像构建完成并做了基础自测

注意：不要移动 Dockerfile 文件的位置。受 EAR 约束，MegFlow 无法提供现成的 docker 镜像，需要自己 build 出来，这个过程用了相对路径。

```
$ docker images
REPOSITORY          TAG             IMAGE ID         CREATED          SIZE
megflow             latest         c65e37e1df6c    18 hours ago    5.05GB
```

直接用 `${IMAGE ID}` 进入开始跑应用，挂载上之前下载好的模型包

```
$ docker run -p 18081:8081 -p 18082:8082 -v ${DOWNLOAD_MODEL_PATH}:/megflow-runspace/  
↪flow-python/examples/models -i -t c65e37e1df6c /bin/bash
```

1.3 Python Built-in Applications

接下来开始运行好玩的 Python 应用

- 猫猫围栏运行手册
 - 图片注册猫猫
 - 部署视频围栏，注册的猫离开围栏时会发通知
 - 未注册的不会提示
- 电梯电瓶车告警
 - 电梯里看到电瓶车立即报警
- *quickstart*
 - 问答式创建自己的应用
- 视频实时超分

2.1 一、安装依赖

2.1.1 安装 Rust

```
$ sudo apt install curl
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

成功后, cargo 应该可以正常执行

```
$ cargo --version
cargo 1.56.0 (4ed5d137b 2021-10-04)
```

如果不成功, 提示 Command 'cargo' not found, 可以按照提示加载一下环境变量 (重新连接或打开终端也可以):

```
source $HOME/.cargo/env
```

cargo 是 Rust 的包管理器兼编译辅助工具。类似 Java maven/go pkg/C++ CMake 的角色, 更易使用。

2.1.2 安装 python3.x (推荐 conda)

打开 [miniconda 官网](#) 下载 miniconda 安装包, 修改权限并安装。

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod a+x Miniconda3-latest-Linux-x86_64.sh
$ ./Miniconda3-latest-Linux-x86_64.sh
```

安装时接受 conda 修改默认.bashrc 环境变量 (zsh 用户还需自行修改.zshrc 中的 conda initialize 配置)。成功后 conda 可正常运行

```
$ conda --version
conda 4.10.3
```

创建一个 Python3.x (这里以 3.8 为例) 的环境, 激活。

```
$ conda create --name py38 python=3.8
$ conda activate py38
```

2.1.3 安装基础依赖库

megflow 的构建需要一些基础依赖库, 下面以 ubuntu 系统为例:

```
$ sudo apt update
$ sudo apt install -y wget yasm clang git build-essential
$ sudo apt install -y libssl-dev
$ sudo apt install -y pkg-config --fix-missing
```

2.2 二、编译

编译并安装 Python 包到当前用户下

```
$ git clone --recursive https://github.com/MegEngine/MegFlow --depth=1
$ cd MegFlow/flow-python
$ python3 setup.py install --user
```

编译成功后, 在 Python import megflow 正常。

2.3 三、Python “开机自检”

```
$ cd examples
$ megflow_run -p logical_test
```

`logical_test` 是 `examples` 下最基础的计算图测试用例，运行能正常结束表示 MegFlow 编译成功、基本语义无问题。

`megflow_run` 是计算图的实现。编译完成之后不再需要 `cargo` 和 `Rust`，使用者只需要

- `import megflow` 成功
- `megflow_run -h` 正常

2.4 四、编译选项

aarch64 源码编译方式和 源码编译 相同，此处只说明差异。

3.1 环境差异

如果是华为鲲鹏 ARM 服务器 CentOS 系统，gcc 需要 ≥ 7.5 版本，系统默认的 aarch64-redhat-linux-gcc 4.8.5 缺 `__ARM_NEON` 会导致大量异常。

```
$ yum install -y centos-release-scl
$ yum install -y devtoolset-8-gcc devtoolset-8-gcc-c++
$ source /opt/rh/devtoolset-8/enable
$ gcc --version
gcc (GCC) 8.3.1 20190311 (Red Hat 8.3.1-3)
...
```

3.2 软件差异

conda 建议使用 archiconda，目前 (2021.12.06) miniconda aarch64 官网版本在 KhadasVIM3/JetsonNao 上均会崩溃。archiconda 安装：

```
$ wget https://github.com/Archiconda/build-tools/releases/download/0.2.3/Archiconda3-
→0.2.3-Linux-aarch64.sh
$ chmod +x Archiconda3-0.2.3-Linux-aarch64.sh && ./Archiconda3-0.2.3-Linux-aarch64.sh
```


4.1 下载模型包

docker 运行方式，建议把模型包下好，解压备用。下载地址

4.2 安装 wsl2

安装文档 已经非常详细，核心是安装 Linux 内核更新包。完成后第 6 步中的 Linux 分发应该可以正常运行。

4.3 安装 docker

下载 windows docker 客户端 并安装。docker 依赖 wsl2，Docker Desktop 启动正常没有报 fail 即可。

4.4 安装 git

下载安装 git 客户端 并运行 Git Bash。

```
$ pwd
/c/Users/username
$ cd /d # 切换到合适的盘符
$ git clone https://github.com/MegEngine/MegFlow
```

(下页继续)

(续上页)

```
...  
$ cd MegFlow  
$ docker build -t megflow .  
... # 等待镜像完成，取决于网络和 CPU
```

注意：**不要移动 Dockerfile 文件的位置**。受 EAR 约束，MegFlow 无法提供现成的 docker 镜像，需要自己 build 出来，这个过程用了相对路径。

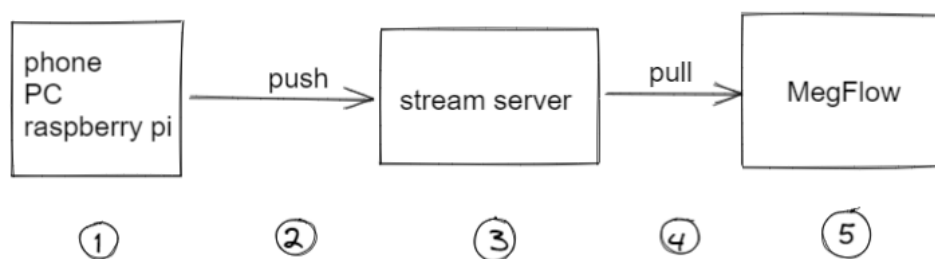
4.5 运行

```
$ docker images  
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE  
megflow             latest      c65e37e1df6c    18 hours ago    5.05GB
```

直接用 `#{IMAGE ID}` 进入开始跑应用，挂载上之前下载好的模型包

```
$ docker run -p 18081:8081 -p 18082:8082 -v #{DOWNLOAD_MODEL_PATH}:/megflow-runspace/  
↪flow-python/examples/models -i -t c65e37e1df6c /bin/bash
```


5.1 视频解析概要



workflow

1. 前端采集。录制的视频需要用 `ffmpeg` 转到 `.ts/.h264/.h265` 格式，**不能直接推.mp4**
2. 推流。`ffmpeg` 命令发数据包给流服务器
3. 流服务器。`github` 有许多开源实现

本文使用 ``rtsp-simple-server`` 作为样例

1. MegFlow 拉流

2. MegFlow 解码、解析

本文只说明 1~3 如何推流。

5.2 配置流服务器

下载 `rtsp-simple-server` 并启动

```
$ wget https://github.com/aler9/rtsp-simple-server/releases/download/v0.17.2/rtsp-
→simple-server_v0.17.2_linux_amd64.tar.gz
$
$ tar xvf rtsp-simple-server_v0.17.2_linux_amd64.tar.gz && ./rtsp-simple-server
...
2021/08/19 18:08:00 I [0/0] [RTSP] TCP listener opened on :8554
...
```

5.3 笔记本推本地视频文件

如果是手机录制的 `.mp4`，先转成 `.ts`。移除音频（可选）。

```
$ ffmpeg -i test.mp4 -s 640x480 -q:v 2 -vcodec copy -an test.ts
```

推 `.ts` 文件上去

```
$ ffmpeg -re -stream_loop -1 -i test.ts -c copy -f rtsp rtsp://127.0.0.1:8554/test
```

5.4 笔记本推本地摄像头

```
ffmpeg -framerate 25 -video_size 640x480 -i /dev/video0 -vcodec h264 -f rtsp rtsp://
→127.0.0.1:8554/test
```

参数说明 | 选项 | 含义 || -l | -l -framerate | 每秒帧数量 (FPS) || -video_size | 采集视频宽高 || -i | 设备文件描述符 || -f | format 格式, 如 `rtsp/flv` 等 || -q:v | 图像品质, 2 表示最好 || -an | 移除音频 || -vcodec copy | 拷贝流 |

5.5 树莓派实时推流

1. 参照官方文档，推流前检查摄像头是否正常运行
2. ffmpeg 推流

```
ffmpeg -framerate 25 -video_size 640x480 -i /dev/video0 -vcodec h264 -f rtsp rtsp://  
↪127.0.0.1:8001/test
```

1. 常见问题
 - ffmpeg 进程被 kill: Out of memory: killed process: pid GPU 显存不足导致，树莓派最多支持设置 512M 显存
 - USB camera 推流卡顿、帧率低。应该换树莓派专用 camera

5.6 检查流地址是否可用

打开 VLC 媒体播放器 - “网络串流”，地址输入 “rtsp://127.0.0.1:8554/test” 正常播放即可。

Run in 15 minutes

6.1 安装 python3.x (推荐 conda)

打开 [miniconda 官网](#) 下载 miniconda 安装包，修改权限并安装。

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod a+x Miniconda3-latest-Linux-x86_64.sh
$ ./Miniconda3-latest-Linux-x86_64.sh
```

安装时接受 conda 修改默认.bashrc 环境变量 (zsh 用户还需自行修改.zshrc 中的 conda initialize 配置)。成功后 conda 可正常运行

```
$ conda --version
conda 4.10.3
```

创建一个 Python3.x (这里以 3.8 为例) 的环境，激活。

```
$ conda create --name py38 python=3.8
$ conda activate py38
```

6.2 安装 Prebuilt 包

从 [MegFlow release](#) 下载对应 python 版本的.whl 包，安装

```
$ python3 -m pip install megflow-0.1.0-py38-none-linux_x86_64.whl --force-reinstall
```

完成后应该可以 import megflow

```
$ python3
Python 3.8.3 (default, May 19 2020, 18:47:26)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import megflow
```

.whl 提供了 megflow_run 命令，某些环境可能要 export PATH=~/.local/bin/:\${PATH}

```
$ apt install build-essential -y
$ megflow_run -h
megflow_run 1.0
megvii
...
```

6.3 Python “开机自检”

6.3.1 下载 MegFlow 源码（需要使用到 flow-python/examples 中文件）

```
$ git clone https://github.com/MegEngine/MegFlow.git
```

6.3.2 运行“开机自检”

```
$ cd ${MegFlow_PATH}/flow-python/examples # 这行必须
$ megflow_run -p logical_test
```

logical_test 是 examples 下最基础的计算图测试用例，运行能正常结束表示 MegFlow 编译成功、基本语义无问题。目前版本输出为 logical_test pass 即为正确

megflow_run 是计算图的实现。使用者不需要关心 Rust/cargo，只需要

- import megflow 成功
- megflow_run -h 正常

工作原理：megflow 仅是一层接口，由 megflow_run “注入” 建图/调度/优化等实现。

6.4 Python Built-in Applications

接下来开始运行好玩的 Python 应用

- 猫猫围栏运行手册
 - 图片注册猫猫
 - 部署视频围栏，注册的猫离开围栏时会发通知
 - 未注册的不会提示
- 电梯电瓶车告警
 - 电梯里看到电瓶车立即报警
- *quickstart*
 - 问答式创建自己的应用
- 视频实时超分

CHAPTER 7

模型和测试数据下载

取最新的 models_xxx.zip, 解压、软链为 examples/models

```
$ wget ${URL}/models.zip
$ cd flow-python/examples/application
$ ln -s ${DOWNLOAD_DIR}/models models
```

如果有 MegFlow-models repo, 可以直接

```
$ cd MegFlow-models
$ git-lfs update
$ git lfs pull
```


8.1 简介

本文介绍如何使用 megflow_quickstart **问答式** 创建应用。

目前支持 4 种用法：

- modelserving。单模型图片服务
- 图片 pipeline 服务
- 视频 pipeline
- 自定义模板

8.2 单模型服务

假设模型使用 megengine 格式且 input tensor 只有一个

```
$ megflow_quickstart
...
Welcome to MegFlow quickstart utility.
Please enter values for the following settings (just press Enter to accept a default.
↳value, if one is given in brackets).
> Enter the root fullpath for the project. [megflow-app]
megflow-app
```

(下页继续)

(续上页)

```

> Enter project type, modelserving/image/video/custom? [modelserving]
modelserving
📄 fetching remote template, please wait...
> Enter model input tensor name. [data]
data
> Enter model fullpath. [model.mge]
model.mge
📄 Project created, read ${PROJECT_dir}/README.md to run it.

```

quickstart 会依次问几个问题，并且提供默认值：

- 项目路径
- 服务类型，这里用 `modelserving`
- `input tensor` 名称，这里用 `data`
- 模型所在路径。阅读此文档生成 `megengine` 模型

正常会提示项目创建成功，阅读 `${PROJECT_dir}/README.md` 即可运行。

```

$ cd megflow-app
$ ./requires.sh # 安装 Python 依赖
$ cd ..
$ megflow_run -p megflow-app/config.toml -p megflow-app # 运行服务
...
# 浏览器打开 127.0.0.1:8080/docs

```

对于可恢复的错误（如模板拉取失败），quickstart 会提醒重试，对应 emoji 是 📄

8.3 图片/视频服务

```

$ megflow_quickstart
...
Welcome to MegFlow quickstart utility.
Please enter values for the following settings (just press Enter to accept a default_
↩value, if one is given in brackets).
> Enter the root fullpath for the project. [megflow-app]
megflow-app
> Enter project type, modelserving/image/video/custom? [modelserving]
image
📄 fetching remote template, please wait...
📄 Project created, read ${PROJECT_dir}/README.md to run it.

```

图片/视频创建的项目只有服务框架，可以用 `megflow_run` 直接运行，不含具体业务功能。

8.4 自定义模板

quickstart 工作原理:

- 拉取 github 上对应分支
- 检查分支里的 placeholder
- 让用户填写 placeholder 对应内容
- 替换 placeholder

此流程同样可用于自定义 repo 和分支, quickstart 提供了 `--git` 参数

```
$ megflow_quickstart --git https://github.com/user/repo
...
> Enter project type, modelserving/image/video/custom? [modelserving]
custom
...
```

custom 选项会问以下问题:

- 模型路径
- 类型
- 分支名称
- 如有 placeholder, 应该替换成什么

placeholder 使用的正则匹配是

```
$ cat flow-quickstart/main.rs
...
    let re = Regex::new(r"##[_\a-zA-Z0-9]*##").unwrap();
...
```

8.5 MegFlow 服务使用方式

8.5.1 WebUI 图片

浏览器打开对应端口 (例如 `http://127.0.0.1:8080/docs`), 选择一张图 “try it out” 即可。

8.5.2 WebUI 视频

浏览器打开端口服务（例如 `http://127.0.0.1:8080/docs`）

- 参照如何生成 `rtsp`，提供一个 `rtsp` 流地址
- 或者给 `.mp4` 文件的绝对路径（文件和 8080 服务在同一台机器上）

8.5.3 命令行方式

图片服务

```
$ curl http://127.0.0.1:8080/analyze/image_name -X POST --header "Content-Type:image/
↪*" --data-binary @test.jpeg
```

`image_name` 是用户自定义参数，用在需要 POST 内容的场景。这里随便填即可；`test.jpeg` 是测试图片

视频服务

```
$ curl -X POST 'http://127.0.0.1:8085/start/rtsp%3A%2F%2F127.0.0.1%3A8554%2Ftest1.ts
↪' # start rtsp://127.0.0.1:8554/test1.ts
start stream whose id is 2%
$ curl 'http://127.0.0.1:8085/list' # list all stream
[{"id":1,"url":"rtsp://10.122.101.175:8554/test1.ts"}, {"id":0,"url":"rtsp://10.122.
↪101.175:8554/test1.ts"}]%`
```

路径中的 `%2F`、`%3A` 是 URL 的转义字符

8.5.4 Python Client 方式

图片 client 代码

```
import requests
import cv2

def test():
    ip = 'localhost'
    port = '8084'
    url = 'http://{ip}:{port}/analyze/any_content'.format(ip, port)
    img = cv2.imread("./test.jpg")
    _, data = cv2.imencode(".jpg", img)
    data = data.tobytes()

    headers = {'Content-Length': '%d' % len(data), 'Content-Type': 'image/*'}
    res = requests.post(url, data=data, headers=headers)
```

(下页继续)

(续上页)

```
print(res.content)

if __name__ == "__main__":
    test()
```

视频 client 代码

```
import requests
import urllib

def test():
    ip = 'localhost'
    port = '8085'
    video_path = 'rtsp://127.0.0.1:8554/vehicle.ts'
    video_path = urllib.parse.quote(video_path, safe='')
    url = 'http://{}:{}/start/{}'.format(ip, port, video_path)

    res = requests.post(url)
    ret = res.content
    print(ret)

if __name__ == "__main__":
    test()
```

8.5.5 其他语言

rweb/Swagger 提供了 http RESTful API 描述文件，例如在 <http://127.0.0.1:8084/openapi.json>。swagger_codegen 可用描述文件生成 java/go 等语言的调用代码。更多教程见 [swagger codegen tutorial](#)。

串联检测和分类

本文将在 `tutorial01 modelserving` 的基础上扩展计算图：先检测、再扣图分类。对外提供视频解析服务。完整的代码在 `flow-python/examples/simple_det_classify` 目录。

9.1 移除分类预处理

见 `生成带预处理的模型`

9.2 准备检测模型

这里直接用现成的 YOLOX mge 模型。复用 `cat_finder` 的检测 或者从 YOLOX 官网 下载最新版。

9.3 配置计算图

`flow-python/examples` 增加 `simple_det_classify/video_cpu.toml`

```
$ cat flow-python/examples/simple_det_classify/video_cpu.toml  
  
main = "tutorial_02"  
  
# 重资源结点要先声明
```

(下页继续)

```
[[nodes]]
name = "det"
ty = "Detect"
model = "yolox-s"
conf = 0.25
nms = 0.45
tsize = 640
path = "models/simple_det_classify_models/yolox_s.mge"
interval = 5
visualize = 1
device = "cpu"
device_id = 0

[[nodes]]
name = "classify"
ty = "Classify"
path = "models/simple_det_classify_models/resnet18_preproc_inside.mge"
device = "cpu"
device_id = 0

[[graphs]]
name = "subgraph"
inputs = [{ name = "inp", cap = 16, ports = ["det:inp"] }]
outputs = [{ name = "out", cap = 16, ports = ["classify:out"] }]
# 描述连接关系
connections = [
  { cap = 16, ports = ["det:out", "classify:inp"] },
]

...
# ty 改成 VdieoServer
[[graphs.nodes]]
name = "source"
ty = "VideoServer"
port = 8085

...
```

想对上一期的配置，需要关注 3 点：

- 视频流中的重资源结点，需要声明在 [[graphs]] 之外，因为多路视频需要复用这个结点。如果每一路都要启一个 det 结点，资源会爆掉
- connections 不再是空白，因为两个结点要描述连接关系

- Server 类型改成 VideoServer, 告诉 UI 是要处理视频的

9.4 运行测试

运行服务

```
$ cd flow-python/examples
$ megflow_run -c simple_det_classify/video_cpu.toml -p simple_det_classify
```

9.5 资源

此文档描述 graph toml 定义

此文档描述 Python node 接口定义

批量推理和 Pipeline 级测试

本文将在 `tutorial02` 的基础上扩展功能：动态 `batching` 测试 QPS 提升。

10.1 分类模型支持动态 batch

resnet 的 `dump` 需要支持多 batch 输入，样例 `dump_resnet.py`

10.2 分类用 `batch_recv` 接口

新的 `classify.py` 改成这样：

```
$ cat flow-python/examples/simple_det_classify/classify.py
...
def exec(self):
    # batching
    (envelopes, _) = self.inp.batch_recv(self.batch_size, self.timeout)

    if len(envelopes) == 0:
        return
...

```

这里 `batch_recv` 的参数列表

然后在 Python 层合并 data，调 `inference_batch`

```
data = np.concatenate(crops)
types = self._model.inference_batch(data)
```

10.3 Pipeline 级测试

MegFlow 支持直接输入图片集/视频列表做测试，不需要 http 服务。使用方自行实现 Validation 结点，集成进 CI 做正确性/性能测试。

10.3.1 图片集测试

以 `simple_classification image_test` 为例

```
...
    [[graphs.nodes]]
    name = "source"
    ty = "ImageInput"
    urls = ["/mnt/data/user/image/", "/home/test_data_dir/"]
...
```

pipeline 建图等不变，新增了一种 source 叫做 ImageInput，调用方填 urls 做图片目录列表。

运行方法不变

```
$ megflow_run -c simple_classification/image_test.toml -p simple_classification
```

10.3.2 视频列表测试

以 `simple_det_classify video_test` 为例：

```
...
    [[graphs.nodes]]
    name = "source"
    ty = "VideoInput"
    repeat = 1
    urls = ["rtsp://127.0.0.1:8554/test.ts", "/mnt/data/file.mp4"]
...
```

建图同样不变，新增了 VideoInput 结点，参数列表

使用方法不变

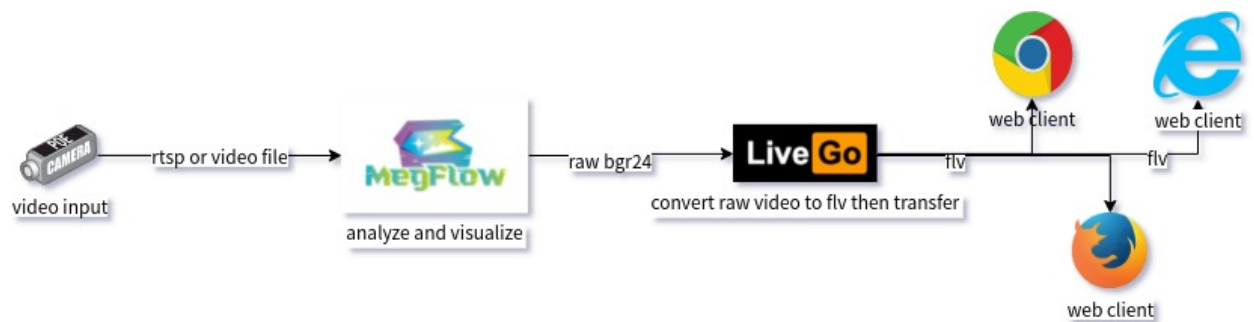
```
$ megflow_run -c simple_det_classify/video_test.toml -p simple_det_classify
```


视频结果 Web 可视化

11.1 依赖

依赖 `ffmpeg` 命令，请确保已经安装并能正常运行。

11.2 部署图



- 1) video input. 可以是 rtsp server 或者视频文件，直接传绝对路径
- 2) MegFlow. 解析视频，把渲染后的结果以 bgr24 格式发给下游
- 3) LiveGo 用于把 raw_video 转为 flv 格式。引入 LiveGo 的原因：
 - 浏览器无法直接播放 rtsp 流，推荐方案是 HLS 或 http-flv
 - 播放端可能不止一个，需要流量转发服务

- 生产环境中存储/点播功能是刚需，MegFlow 不具备此功能，应由其他服务完成

11.3 操作指南

1) 下载启动 LiveGo

```
$ wget https://github.com/gwuhaolin/livego/releases/download/0.0.15/livego_0.0.15_
→linux_amd64.tar.gz && tar xvf livego_0.0.15_linux_amd64.tar.gz && ./livego &
...
INFO[2021-10-27T15:44:40+08:00] HLS server enable....
INFO[2021-10-27T15:44:40+08:00] RTMP Listen On :1935
INFO[2021-10-27T15:44:40+08:00] HTTP-API listen On :8090
INFO[2021-10-27T15:44:40+08:00] HTTP-FLV listen On :7001
...
```

2) 自测视频推流、播放正常

假设测试视频是 `demo.flv` (建议长度超过 30 秒)，使用 `push_video.py` 推流到 LiveGo

```
$ cd ${MegFlow_dir}/flow-python/examples/misc/visualize_client
$ python3 push_video.py
```

浏览器打开 `index.html`，应能正常播放。

3) 运行猫猫围栏可视化配置

猫猫围栏环境设置见 `cat_finder README`。运行起 `video` 基础版后，能够运行 `video` 可视化版本

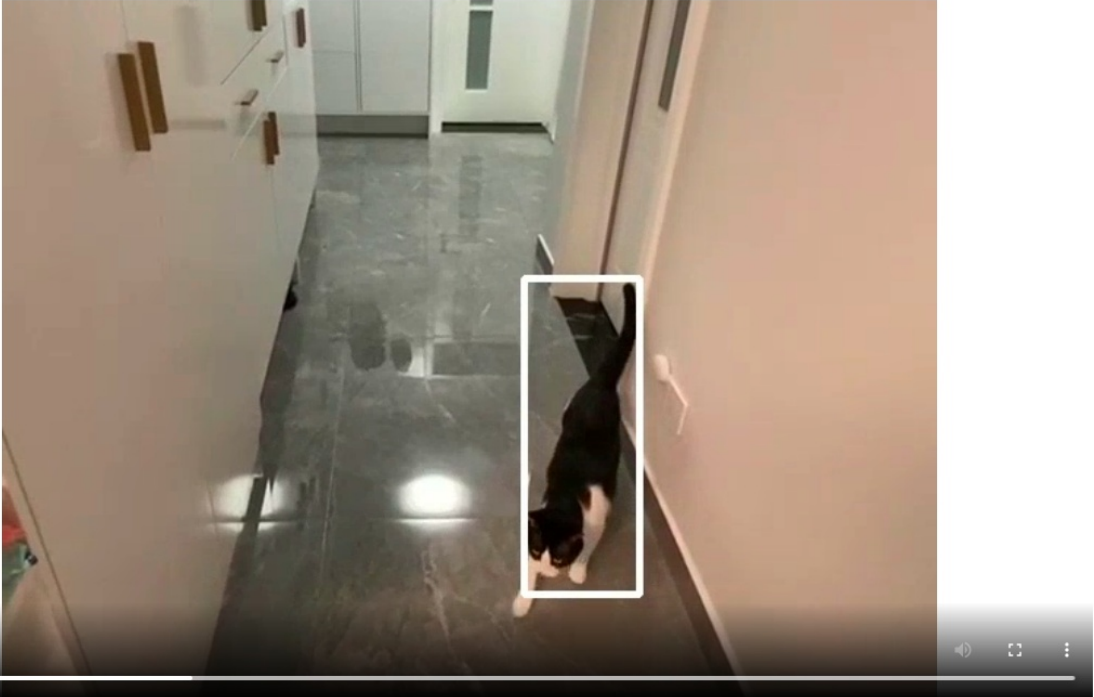
```
$ cd ${MegFlow_dir}/flow-python/examples
$ megflow_run -c cat_finder/video_visualize.toml -p cat_finder
...
```

浏览器打开 8002 端口，提供 `rtsp` 地址或视频文件绝对路径，try it out。

打开 `index.html`，应能播放可视化结果。

Stream URL:

isLive withCredentials hasAudio hasVideo



```

[FLVDemuxer] > Parsed onMetaData
[FLVDemuxer] > Parsed AVCDecoderConfigurationRecord
[MSEController] > Received Initialization Segment, mimeType: video/mp4;codecs=avc1.42c01e
[MSEController] > MediaSource onSourceEnded
[MSEController] > MediaSource onSourceOpen
[FLVDemuxer] > Parsed onMetaData
[FLVDemuxer] > Parsed AVCDecoderConfigurationRecord
[MSEController] > Received Initialization Segment, mimeType: video/mp4;codecs=avc1.42c01e
[MSEController] > MediaSource onSourceEnded

```

4) 注意事项和常见问题

播放地址

注意 index.html 播放地址不能有多余的 ‘/’，例如 `http://10.199.1.100:7001/live/megflow-test.flv` 是无法播放的。

视频长度

测试视频建议超过 30s。至少有 2 个 I 帧，否则影响 flv 转换。

视频尺寸

建议测试视频为标准 480p/720p/1080p。

跨域问题

web client 修改自 `flv.js`。如果不用 index.html、直接浏览器打开原始 demo，需自行解决跨域问题。

MegFlow 的建图描述文件使用 toml 格式。toml 注重人类可读性，学习难度约等于 markdown，看完下面 2 个例子大约就会写了。

12.1 图片范例

举个栗子，cat_finder/image_gpu.toml:

```
// 完整的计算图的名字，叫啥都行
main = "cat_finder_image"

[[graphs]]
name = "subgraph"
↳                                     // 子图类型，叫啥都行。子图 == 自己写的业务
inputs = [{ name = "inp", cap = 16, ports = ["det:inp"] }] //↳
↳子图入口在 det 节点的 inp 端口，队列长度是 16
outputs = [{ name = "out", cap = 16, ports = ["redis_proxy:out"] }] // 子图出口是↳
↳redis_proxy 的 out 端口
connections = [
↳                                     // 构图，A 的输出，连到 B 的输入
  { cap = 16, ports = ["det:out", "reid:inp"] }, // det 输出接到↳
↳reid 输入
  { cap = 16, ports = ["reid:out", "redis_proxy:inp"] }, // reid 输出送给 redis_
↳proxy
```

(下页继续)

```

]

[[graphs.nodes]]           // 子图节点声明。注意 4 个空格，表示和 [[graphs]]_
↳的层次关系
  name = "det"              // 找 examples/xxx/ 下面的 det.py 或者 det_
↳package
  ty = "Detect"             // 这个 node 执行 Detect class
  model = "yolox-nano"      // 自定义参数，yolox 模型构造需要
  conf = 0.25               // yolox 需要的 det 阈值
  nms = 0.45                // yolox 需要的 nms 阈值
  tsize = 640               // yolox 需要的 inference size
  path = "models/yolox_nano.pkl" // yolox 模型相对路径

[[graphs.nodes]]           // reid 和 det 同理
  name = "reid"
  ty = "Reid"
  thres = 1300              // 里面用拉普拉斯计算图像清晰度，阈值写的 1300
  path = "models/aligned_reid.pkl"

[[graphs.nodes]]           // 纯业务逻辑，name 和 ty 字段是必选，其他看需求
  name = "redis_proxy"
  ty = "RedisProxy"
  ip = "127.0.0.1"          // redis ip 地址
  port = "6379"             // redis port
  mode = "save"             // 控制此节点存特征进 redis
  prefix = "feature."       // redis key 的前缀，不然整个 redis 库乱乱的

[[graphs]]                 // 描述完整运行的计算图
name = "cat_finder_image"
connections = [
  { cap = 16, ports = ["source:out", "destination:inp"] }, // source 就是 swagger _
↳这类 service，接收视频或图片用的。收到的数据（图像、视频、extra 字段）送进子图
  { cap = 16, ports = ["source:inp", "destination:out"] } // 子图处理完，把结果发给_
↳http server
]

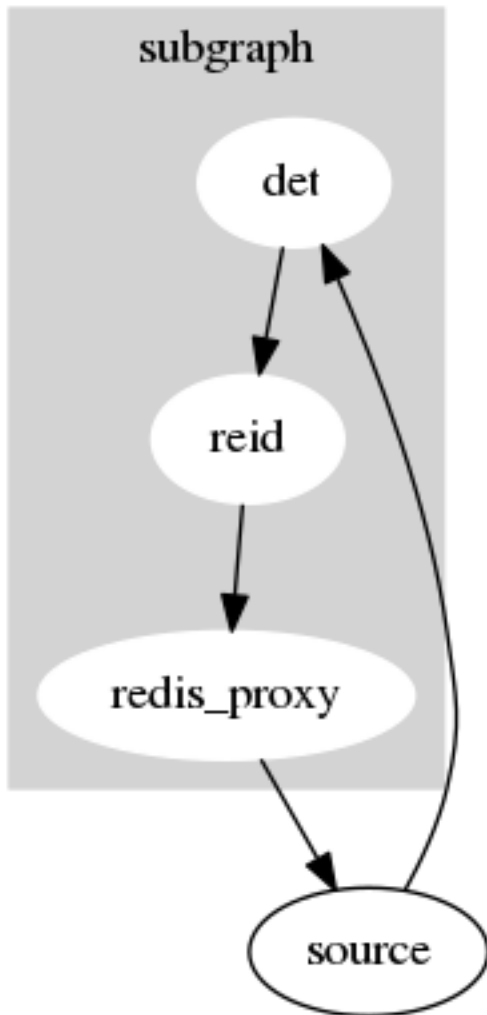
[[graphs.nodes]]           // http server 的配置
  name = "source"
  ty = "ImageServer" // 这是个图片服务
  port = 8081          // 端口号
  response = "json" // response Content-Type 用 application/json。不写就是默认 _
↳image/jpeg

```

(续上页)

```
[[graphs.nodes]] // dest 是自己写的业务子图
name = "destination"
ty = "subgraph" // 子图类型。没事儿改它干啥
```

对应可视化的计算图：



12.2 视频范例

视频和图片的唯一区别：**重资源的节点要在子图之外声明。**

MegFlow 设计上支持不同视频流跑不同业务。每创建 1 路视频就会创建 1 个子图，40 路视频就是 40 个。det/reid 这种很重的节点自然要被复用。

以 `cat_finder/video_gpu.toml` 为例：

```

main = "cat_finder_video"           // 完整计算图名字

[[nodes]]                          // det 节点在子图外声明
name = "det"
ty = "Detect"
model = "yolox-tiny"
conf = 0.25
nms = 0.45
tsize = 640
path = "models/yolox_tiny.pkl"
interval = 5

[[nodes]]                          // ReID 节点同样在外部声明，被共享
name = "reid_video"
ty = "ReIDVideo"
path = "models/aligned_reid.pkl"

[[nodes]]                          // redis_proxy 使用了连接池，也可以看作“重资源”
name = "redis_proxy"
ty = "RedisProxy"
ip = "127.0.0.1"
port = "6379"
mode = "search"
prefix = "feature."

[[graphs]]
name = "subgraph"
inputs = [{ name = "inp", cap = 16, ports = ["det:inp"] }]
outputs = [{ name = "out", cap = 16, ports = ["redis_proxy:out"] }]
connections = [ // 描述连接关系
  { cap = 16, ports = ["det:out", "track:inp"] },
  { cap = 16, ports = ["track:out", "shaper:inp"] },
  { cap = 16, ports = ["shaper:out", "reid_video:inp"] },
  { cap = 16, ports = ["reid_video:out", "redis_proxy:inp"] },
]

[[graphs.nodes]]                  // 给每个检测目标，赋予唯一的 ID
name = "track"
ty = "Track"

[[graphs.nodes]]                  // 业务逻辑：ID 结束后，给个最优结果
name = "shaper"
ty = "Shaper"
mode = "BEST"                     // 结果类型，目前只支持 "BEST" 最优

```

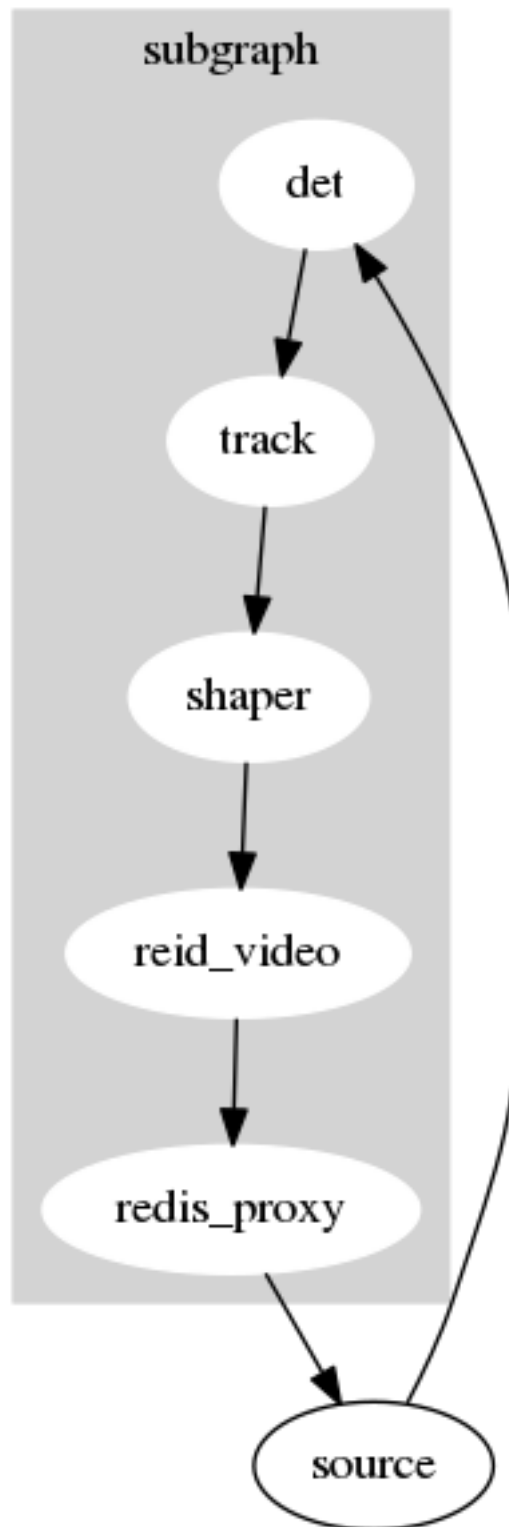
(下页继续)

(续上页)

```
[[graphs]]
name = "cat_finder_video"
connections = [
  { cap = 16, ports = ["source:out", "destination:inp"] },
  { cap = 16, ports = ["source:inp", "destination:out"] }
]

[[graphs.nodes]]
name = "source"
ty = "VideoServer" // 视频服务
port = 8082        // 8081 做注册, 8082 做解析

[[graphs.nodes]]
name = "destination"
ty = "subgraph"
```



视频解析服务可视化结果和图片接近：

12.3 完整定义

TL;DR

严谨的描述文件定义如下:

```
// 节点间的连接channel
struct Connection {
    cap: usize, // channel容量
    ports: Vec<String>, // 连接的节点端口, 格式是`节点名:端口名[:标签]`
}
// 有名channel
struct NamedConn {
    name: String, // channel的名字
    cap: usize, // channel容量
    ports: Vec<String>, // 连接的节点端口, 格式是`节点名:端口名[:标签]`
}
// 节点定义
struct Node {
    name: String, // 节点名
    ty: String, // 节点类型
    cloned: usize, // 表示并行度, 默认值为1
    res: Vec<String>, // 引用的资源名字列表
    ... // 其他参数, 会被透传到节点的构造函数中
}
// 资源定义
struct Resource {
    name: String, // 资源名字
    ty: String, // 资源类型
    ... // 其他参数, 会被透传到资源的构造函数中
}

struct Graph {
    name: String, // 图的名字
    resources: Vec<Resource> // 资源声明, 生命周期与该图绑定
    nodes: Vec<Node>, // 节点声明, 生命周期与该图绑定
    inputs: Vec<NamedConn>, // 图输入声明
    outputs: Vec<NamedConn>, // 图输出声明
    connections: Vec<Connection>, // 节点间连接声明
}

struct Config {
    resources: Vec<Resource> // 全局共享资源, 生命周期与整个应用绑定
    nodes: Vec<Node>, // 全局共享节点, 生命周期与整个应用绑定
    graphs: Vec<Graph>, // 图声明
}
```

(下页继续)

(续上页)

```
main: String, // 主图名字, 及应用的进入点
}
```

从一个最简单的例子开始

```
import megflow
@megflow.register(name="alias", inputs=["inp"], outputs=["out"])
class Node:
    def __init__(self, name, args):
        pass
    def exec(self):
        envelope = self.inp.recv()
        msg = dowith(envelope.msg)
        self.out.send(envelope.repack(msg))
```

这其中有三部分内容: `register` 装饰器, `Node` 的构造函数, `Node` 的执行函数

1. `register` 装饰器

- `name`: 若指定, 则 `register` 所修饰插件重命名为 `name`, 默认为 `register` 所修饰类的类名
- `inputs`: `Node` 的输入列表, 每个输入 `input` 都可以在 `exec` 方法中, 通过 `self.input` 访问,
- `outputs`: `Node` 的输出列表, 每个输出 `output` 都可以在 `exec` 方法中, 通过 `self.output` 访问
- `exclusive`: 默认为 `False`, 调度模型是一个 `thread local` 的协程调度器, 若为 `True`, 则将该任务安排到线程池中

2. `Node` 的构造函数

- `name`: 即参数文件中 Node 的 `name` 字段
- `args`: 即参数文件中 Node 的剩余参数字段

3. Node 的执行函数

- 一个 python 插件的执行方法必须是命名为 `exec` 的零参成员方法
- 对于在参数文件中该插件引用的资源 `resource`, 可以在 `exec` 方法中, 通过 `self.resource` 访问
- 通过输入的 `recv` 方法取得输入消息, 输入消息是 `Envelope` 对象, 其 `msg` 成员即开发者真正需要读写的消息实例
- `Envelope` 语义为在图中流转的消息的相关信息, 由于这些信息需要在图中被传递, 所以开发者应该保持消息与 `Envelope` 的对应关系
- 若一个 `Envelope` 携带的消息被拆分为多个消息, 或者转换为另一个消息, 应该通过 `Envelope` 的 `repack` 方法, 将 `Envelope` 与消息关联起来
- 通过输出的 `send` 方法发送输出消息, 输出消息是 `Envelope` 对象

MegFlow 也提供了一系列异步工具

1. `yield_now()`, 让出当前任务的执行权
2. `sleep(dur)`, 使当前任务沉睡 `dur` 毫秒
3. `join(tasks)`, `tasks` 参数是一个函数列表, `join` 堵塞直到 `tasks` 中的函数都执行完毕
4. `create_future(callback)`, `callback` 参数是一个函数, 默认值为 `None`, `create_future` 返回一个 (`Future`, `Waker`) 对象
 - `Future::wait`, 堵塞直到 `Waker::wake` 被调用, 返回 `Waker::wake(result)` 传入的 `result` 参数

生成 MegEngine 模型

14.1 生成不带预处理的模型

Github MegEngine models 有现成的 imagenet 预训模型。这里把模型 dump 成.mge。

新增 dump.py, 按 [1, 3, 224, 224] 尺寸 trace 模型, 打开推理优化选项, 保存为 model.mge。

```
$ git clone https://github.com/MegEngine/models
$ cd models
$ export PYTHONPATH=${PWD}:${PYTHONPATH}
$ cd official/vision/classification/resnet
$ python3 dump.py
$ ls -lah model.mge
...
```

dump.py 已经 PR 到 MegEngine/models 分类模型目录

```
$ cat dump.py
...
    data = mge.Tensor(np.random.random((1, 3, 224, 224))) # 准备一个样例输入

@jit.trace(capture_as_const=True)
def pred_func(data):
    outputs = model(data) # trace 每个 opr 的 shape
    return outputs
```

(下页继续)

```

pred_func(data)
pred_func.dump( # 保存模型
    graph_name,
    arg_names=["data"],
    optimize_for_inference=True, # 打开推理优化选项
    enable_fuse_conv_bias_nonlinearity=True, # 打开 fuse conv+bias+ReLU pass_
    →推理更快
)
...

```

14.2 模型单测

开发模型的推理封装，对外提供功能内聚的接口。调用方传入一张或多张图片、直接获取结果，尽量避免关心内部实现（如用何种 backbone、预处理是什么、后处理是什么）。

```

$ cat flow-python/examples/application/simple_classification/lite.py
...
def inference(self, mat):
    img = self.preprocess(mat, input_size=(224,224), scale_im = False, mean=[103.
    →530, 116.280, 123.675], std=[57.375, 57.120, 58.395])

    # 设置输入
    inp_data =self.net.get_io_tensor("data")
    inp_data.set_data_by_share(img)

    # 推理
    self.net.forward()
    self.net.wait()

    # 取输出
    output_keys = self.net.get_all_output_name()
    output = self.net.get_io_tensor(output_keys[0]).to_numpy()
    return np.argmax(output[0])
...
$ python3 lite.py --model model.mge --path test.jpg # 测试
2021-09-14 11:45:02.406 | INFO      | __main__:<module>:81 - 285

```

285 是分类模型最后一层的 argmax，对应含义需要查 imagenet 数据集分类表，这里是“Egyptian cat”（下标从 0 开始）。

14.3 生成带预处理的模型

MegEngine 除了不需要转模型，还能消除预处理。我们修改 `dump_resnet.py` 把预处理从 SDK/业务代码提到模型内。这样的好处是：**划清工程和算法的边界**，预处理本来就应该由 `scientist` 维护，每次只需要 `release mge` 文件，减少交接内容

```
...
@jit.trace(capture_as_const=True)
def pred_func(data):
    out = data.astype(np.float32)

    output_h, output_w = 224, 224
    # resize
    print(shape)
    M = mge.tensor(np.array([[1,0,0], [0,1,0], [0,0,1]], dtype=np.float32))
    M_shape = F.concat([data.shape[0],M.shape])
    M = F.broadcast_to(M, M_shape)
    out = F.vision.warp_perspective(out, M, (output_h, output_w), format='NHWC')
    # mean
    _mean = mge.Tensor(np.array([103.530, 116.280, 123.675], dtype=np.float32))
    out = F.sub(out, _mean)
    # div
    _div = mge.Tensor(np.array([57.375, 57.120, 58.395], dtype=np.float32))
    out = F.div(out, _div)
    # dimshuffile
    out = F.transpose(out, (0,3,1,2))

    outputs = model(out)
    return outputs
...
```

具体实现是在 `trace inference` 里增加预处理动作，`fuse opr` 优化加速的事情交给 MegEngine 即可。更多 `cv` 操作参照 [MegEngine API 文档](#)。

因为推理输入变成了 BGR，所以 `dump` 模型的时候参数也应该跟着变

```
$ python3 dump.py -a resnet18 -s 1 224 224 3
```


CHAPTER 15

FAQ

Q: `megflow_run -p logical_test` 无法运行怎么办?

A1: 如果报错 `message: "No such file or directory" }`, 确认是否 `cd flow-python/examples`

A2: 确认安装了 `python` 组件, 即 `cd flow-python` 再 `python3 setup.py install --user`
还不行就提 `issue`。

Q: 视频流跑一路没事。跑多个, 内存爆了/显存爆了/模型加载多次是因为啥?

A: 参照 `cat_finder/video.toml`, 把涉模型的 `nodes` 移到 `[[graphs]]` 上面, 让多个子图来共享。

每启动一个视频流就会启一个子图, 如果 `nodes` 放到 `[[graphs]]` 里, 20 路视频就会创建 20 套 `nodes`。

Q: 如何修改服务端口号, 8080 已被占用?

A: 以 `cat_finder` 为例, 端口配置在 `image.toml` 的 `port` 中。

Q: 如何让 `ImageServer` 返回 `json`, 而不是渲染后的图?

A: `ImageServer` 默认返回 `envelope.msg["data"]` 图像。如果要返回 `json` 需要改两处:

- `image.toml` 的配置里增加 `response = "json"`
 - 最终结果用 `self.out.send(envelope.repack(json.dumps(results)))` 发送出去
-

框架既不知道哪些字段可序列化，也不知道要序列化那几个字段，因此需要调用方序列化成 `str`。代码可参照 `examaples/cat_finder/redis_proxy.py`。

Q: 视频流为什么无法停止，调用了 `stop` 接口还是在处理？

A: 调用 `stop` 之后队列的 `push/put` 接口已经被关闭了，不能追加新的，但之前解好的帧还在队列里。需要把遗留的处理完、依次停止子图节点才完全结束。流不会调用 `stop` 即刻停止，实际上有延迟。

如何 Debug 常见问题

一、megflow_run 无法启动服务，直接 core dump 报错退出

如果“Python 开机自检”的 megflow_run -p logical_test 能够正常结束，排查方向应该是 Python import error。调试方法举例

```
$ gdb --args ./megflow_run -c electric_bicycle/electric_bicycle_cpu.toml -p_
↳electric_bicycle
...
illegal instruction
...
```

可以看到 crash 发生在哪个 import

17.1 一、fork 分支

在浏览器中打开 [MegFlow](https://github.com/MegFlow), fork 到自己的 repositories, 例如

```
https://github.com/user/MegFlow
```

clone 项目到本地, 添加官方 remote 并 fetch:

```
$ git clone https://github.com/user/MegFlow && cd MegFlow
$ git remote add megvii https://github.com/MegEngine/MegFlow
$ git fetch megvii
```

对于 git clone 下来的项目, 它现在有两个 remote, 分别是 origin 和 megvii

```
$ git remote -v
origin https://github.com/user/MegFlow (fetch)
origin https://github.com/user/MegFlow (push)
megvii https://github.com/MegEngine/MegFlow (fetch)
megvii https://github.com/MegEngine/MegFlow (push)
```

origin 指向你 fork 的仓库地址; remote 即官方 repo。可以基于不同的 remote 创建和提交分支。

例如切换到官方 master 分支, 并基于此创建自己的分支 (命名尽量言简意赅。一个分支只做一件事, 方便 review 和 revert)

```
$ git checkout MegEngine/master
$ git checkout -b my-awesome-branch
```

或创建分支时指定基于官方 master 分支：

```
$ git checkout -b fix-typo-in-document MegFlow/master
```

`git fetch` 是从远程获取最新代码到本地。如果是第二次 `pr MegFlow` `git fetch megvii` 开始即可，不需要 `git remote add megvii`，也不需要修改 `github.com/user/MegFlow`。

17.2 二、代码习惯

为了增加沟通效率，reviewer 一般要求 contributor 遵从以下规则

- 不能随意增删空行
- `tab` 替换为 4 个空格
- 若是新增功能或平台，需有对应测试用例
- 文档放到 `docs` 目录下，中文用 `.zh.md` 做后缀；英文直接用 `.md` 后缀

开发完成后提交到自己的 repository

```
$ git commit -a
$ git push origin my-awesome-branch
```

推荐使用 `commitizen` 或 `gitlint` 等工具格式化 commit message，方便事后检索海量提交记录

17.3 三、代码提交

浏览器中打开 [MegFlow pulls](#)，此时应有此分支 `pr` 提示，点击 `Compare & pull request`

- 标题**必须**是英文。未完成的分支应以 `WIP:` 开头，例如 `WIP: fix-typo`
- 正文宜包含以下内容，中英不限
 - 内容概述和实现方式
 - 功能或性能测试
 - 测试结果

回到浏览器签署 CLA，所有 CI 测试通过后通知 reviewer merge 此分支。

打包成 Python .whl

18.1 作用

打成 whl 包，使用方直接安装即可，不再需要编译。

18.2 执行

现在使用 Dockerfile 生成各 python 版本.whl

```
$ cd ${MegFlow_dir}
$ # 构造开发环境，安装依赖。已执行过 docker 编译可以跳过此步骤
$ docker build -t megflow -f Dockerfile.github-dev .
$ # 创建结果目录
$ mkdir dist
$ # docker 打包 whl
$ # https://stackoverflow.com/questions/33377022/how-to-copy-files-from-dockerfile-to-
  ↪ host
$ DOCKER_BUILDKIT=1 docker build -f Dockerfile.github-release --output dist .
```

注意 COPY to host 需要：

- Docker 19.03 以上版本
- 需要 DOCKER_BUILDKIT 环境变量
- 需要 -output 参数