



Tengine 用户手册

文档版本 2.1

发布日期 2019-11-08

OPEN AI LAB

变更记录

日期	版本	说明	作者
2019-03-05	0.1	初版	Rui Zhang
2019-04-15	0.2	合并之前文档	Bin Hu
2019-06-20	1.0	翻译成中文版本	Bin Hu
2019-06-25	1.1	Linux 交叉编译配置更改	Bin Hu
2019-06-27	1.2	Tengine 模型转换工具使用方法	Bin Hu
2019-09-17	2.0	目录结构及内容调整 添加 Tengine AAR 添加 PyTengine	CMeng
2019-09-25	2.1	Tengine 编译说明更新	JieJunZeng
2019-11-07	2.2	Tengine 编译说明更新 增加 dump node 说明	JieJunZeng CMeng

目录

1	简介	9
1.1	TENGINE 框架	10
1.2	TENGINE 软件发布包.....	12
2	基本操作	13
2.1	通用指令	13
2.1.1	设置使用的 CPU 内核	13
2.1.2	设置 kernel mode	14
2.2	算子使用多线程运行.....	15
2.2.1	Pooling 多线程模式.....	15
2.2.2	FC 多线程模式	15
2.3	支持 CPU 和 GPU 的异构计算.....	15
3	如何转换自己的模型	17
3.1	转换为 TENGINE 模型.....	17
3.1.1	工具所在路径	17
3.1.2	使用方法	18
3.2	TENGINE 模型量化工具.....	18
3.2.1	工具所在路径	18
3.2.2	使用方法	18
4	高阶使用	19
4.1	添加用户自定义 KERNEL	19
4.2	添加自定义算子.....	24
4.2.1	以 Scale Operator 为例	25
4.3	添加自定义 CPU 设备	27
4.3.1	通过环境变量配置默认 CPU 设备.....	27
4.3.2	预定义的 CPU	27
4.3.3	手动设置 CPU	28
4.4	NPU 使用简介.....	29

4.4.1	RKNN.....	29
4.4.2	NNIE	30
4.4.3	AIPU.....	30
4.5	内存加载模型	31
4.6	DUMP 中间节点数据	32
5	调试技巧	33
5.1	设置 TENGINE 的 LOG 级别和输出函数	33
5.1.1	通过环境变量设置	33
5.1.2	通过 API 设置	33
5.2	开启 PROF_TIME	34
5.2.1	打开方式	34
5.3	GDB 工具的使用	35
6	测试例程	35
6.1	测试数据和模型	35
6.2	编译	35
6.3	运行	36
7	TENGINE AAR	36
7.1	TENGINE AAR API	36
7.2	TENGINE AAR 的使用	38
8	PYTEngine	38
8.1	安装	39
8.1.1	使用源码安装	39
8.1.2	使用 whl 安装包安装	40
8.2	PYTEngine API 介绍	41
8.3	常数	41
8.4	API 介绍	43
8.4.1	Tengine 模块的 API	43
8.4.2	Graph 模块的 API	43
8.4.3	Node 模块 API	44
8.4.4	Tensor 模块 API	45
8.4.5	模块 DeviceAPI	46

8.4.6	context 模块 API	46
8.5	TENGINE	46
8.5.1	__init__	46
8.5.2	__del__	47
8.5.3	__version__	47
8.5.4	requestTengineVersion	47
8.5.5	setDefaultDevice	47
8.5.6	__errno__	47
8.5.7	log_lever	48
8.5.8	log_print	48
8.5.9	plugin.add	48
8.5.10	plugin.remove	48
8.5.11	plugin.__len__	49
8.5.12	plugin.__getitem__	49
8.6	GRAPH 的相关操作	49
8.6.1	__init__	49
8.6.2	__del__	49
8.6.3	setLayout	50
8.6.4	setNode	50
8.6.5	__add__	50
8.6.6	getInputNodeNumber	50
8.6.7	getInputNodeByIdx	50
8.6.8	getOutputNodeNumber	51
8.6.9	getOutputNodeByIdx	51
8.6.10	getOutputTensor	51
8.6.11	getInputTensor	51
8.6.12	getNodeNumber	52
8.6.13	getNodeByIdx	52
8.6.14	getNodeByName	52
8.6.15	setDevice	53
8.6.16	doPerfStat	53
8.6.17	getPerfStat	53
8.6.18	dump	53
8.6.19	setAttr	53
8.6.20	getAttr	54
8.6.21	setGdMethod	54
8.6.22	preRun	54
8.6.23	run	54
8.6.24	wait	55
8.6.25	postRun	55
8.6.26	__status__	55
8.6.27	setEvent	55
8.7	NODE	56
8.7.1	__init__	56
8.7.2	__name__	56
8.7.3	__op__	56

8.7.4	__del__	56
8.7.5	getInputTensorByIdx	56
8.7.6	getOutputTensorByIdx	57
8.7.7	setInputTensorByIdx	57
8.7.8	setOutputTensorByIdx	57
8.7.9	getOutputNumber	57
8.7.10	getInputNumber	58
8.7.11	addAttr	58
8.7.12	getAttr	58
8.7.13	setDevice	58
8.7.14	getDevice	59
8.7.15	dump	59
8.7.16	getDumpBuf	59
8.7.17	setKernel	59
8.7.18	removeKernel	59
8.8	TENSOR	60
8.8.1	__init__	60
8.8.2	__name__	60
8.8.3	__del__	60
8.8.4	shape	60
8.8.5	设置 shape	61
8.8.6	buffer 大小	61
8.8.7	getbuffer	61
8.8.8	buffer	61
8.8.9	getData	61
8.8.10	setData	62
8.8.11	dtype	62
8.8.12	dtype 设置	62
8.8.13	setQuantParam	62
8.8.14	getQuantParam	62
8.9	DEVICE	63
8.9.1	__init__	63
8.9.2	__del__	63
8.9.3	policy	63
8.9.4	setAttr	63
8.9.5	getAttr	64
8.10	CONTEXT	64
8.10.1	__init__	64
8.10.2	__del__	64
8.10.3	getDevNumber	65
8.10.4	getDevByIdx	65
8.10.5	addDev	65
8.10.6	rmDev	65
8.10.7	setAttr	65
8.10.8	getAttr	66
8.11	测试	66

8.11.1	运行 SqueezeNet 测试程序	66
8.11.2	运行 MobileNet 测试程序	67
9	编译 TENGINE 源码	67
9.1	在 X86(UBUNTU)本地编译	67
9.1.1	安装依赖的库文件	68
9.1.2	编译	68
9.1.3	测试	68
9.2	在 ARM(LINUX)中编译源代码	69
9.2.1	获取源代码	69
9.2.2	编译 Armv7	69
9.2.3	编译 Armv8	69
9.2.4	测试	70
9.3	在 X86(UBUNTU)对 ARM LINUX 交叉编译	70
9.3.1	获取源代码	70
9.3.2	安装 rootfs	71
9.3.3	安装交叉编译链	71
9.3.4	编译 Arm64	71
9.3.5	编译 Arm32	71
9.3.6	测试	72
9.4	在 X86 LINUX 平台为 ANDROID 交叉编译	72
9.4.1	Tengine 源代码	72
9.4.2	下载 Android ndk	73
9.4.3	解压	73
9.4.4	设置依赖项库文件路径	73
9.4.5	编译 tengine	73
9.4.6	在电脑上安装 adb, adb_driver	74
9.4.7	在安卓上测试 Tengine 的 Demo	74
9.5	编译样例	75
9.6	编译 TENGINE-MODULE	76
9.6.1	安装依赖的库文件	76
9.6.2	编辑 configuration 文件	76

9.6.3	编译.....	77
9.7	其它编译选项说明.....	78
9.7.1	配置 open blas 库.....	78
9.7.2	所有选项说明.....	78
10	TENGINE C API 介绍.....	80
10.1	数据类型和数据结构.....	80
10.1.1	MAX_SHAPE_DIM_NUM.....	80
10.1.2	Tensor 的数据类型.....	80
10.1.3	数据布局格式.....	81
10.1.4	Tensor 的类型.....	81
10.1.5	Node 转储操作类型.....	82
10.1.6	Graph 性能统计动作类型.....	83
10.1.7	Log 级别.....	83
10.1.8	Graph 执行事件类型.....	84
10.1.9	Graph 执行状态类型.....	84
10.1.10	设备运行策略.....	85
10.1.11	上下文.....	86
10.1.12	Graph 句柄.....	86
10.1.13	Tensor 句柄.....	87
10.1.14	Node 句柄.....	87
10.1.15	事件回调函数.....	88
10.1.16	Log 输出函数.....	88
10.1.17	性能分析记录结构体.....	89
10.1.18	用户自定义 Tensor.....	89
10.1.19	用户自定义 Kernel.....	91
10.2	C API 介绍.....	96
10.2.1	Tengine 创建与释放相关 API.....	99
10.2.2	graph 相关操作.....	101
10.2.3	node 相关操作.....	108
10.2.4	kernel 相关操作.....	118
10.2.5	The tensor operation.....	119
10.2.6	与 graph 运行相关 API.....	126

10.2.7	与设备相关操作	130
10.2.8	与 context 相关部分	138
10.2.9	其他的 API	142
10.3	错误码	145

OPENAI LAB

1 简介

Tengine 是 **OPEN AI LAB** 针对前端智能设备开发的软件开发包，核心部分是一个轻量级，模块化，高性能的 AI 推断引擎，并支持用 CPU、GPU、DSP、NPU 作为硬件加速计算资源异构加速。

Tengine 解决了深度神经网络模型在端侧设备上推理的问题，涵盖了深度神经网络模型的优化，推理和异构调度计算。Tengine 具有通用，开放，高性能等特点。

通用性体现在 Tengine 支持 Tensorflow/Tensorflow-Lite, MXNet, Caffe, ONNX 等主流网络模型格式，并且可以支持 NHWC/NCHW 两种数据排布。同时，还支持 CNN, RNN/LSTM, GAN 等常用网络。

开放性指的 Tengine 对外提供了高度可扩展的接口，用户可以很方便的在 Tengine 代码库之外，开发和扩展自己需要的功能。用户可以通过 Serializer 接口，定义自己的专有模型格式的加载和存储，以及实现模型加密的功能。用户可以通过 Operator 接口，定义自己特有的算。同样，用户还可以通过 Driver 接口，实现自己专有的设备接入 Tengine。

高性能是指 Tengine 对于端侧计算平台，做了大量定制和优化，从而实现在设备上高效运行神经网络。CPU 上核心运算的代码都是针对微架构手工优化的汇编代码，把 ARM CPU 的算力发挥到极致。针对不同参数和形状的卷积，高效实现了多种优化算法，保证了任意形状卷积推理的高性能。

同时，Tengine 还完美支持各种非 CPU 的计算部件，既包括 GPU 和 DSP 等需要编程的计算单元，还包括 Arm 中国 AIPU，华为海思 NNIE，瑞芯微 RKNN 等硬件加速器，并支持和 CPU 的异构计算。

1.1 Tengine 框架

Tengine 的推理过程涵盖了模型的加载解析，格式转换，计算图的调度和优化，在异构后端上高效运行。

在模型加载阶段，来自不同训练框架的模型，通过各自的序列化模块，统一解析和转换成 Tengine 在内存的计算图表示。对于 Tensorflow 模型，因为算子的粒度和灵活性很高，在 Tensorflow 的序列化模块中，还对 Tensorflow 的计算图，做了裁减，合并和优化，来实现统一的计算图表示。

此外，对于高级用户，还可以将网络模型的参数提取出来，通过 Tengine C API 接口来直接构造计算图，从而省去修改或者实现一个序列化模块的工作。Tengine 还提供一个 Python 的 API 包，来帮助用户快速实现这个功能。

在内存的计算图构造完成之后，下一步的工作是进行设备的调度：确定计算图的各个节点在哪个设备上运行。按照预定义的优先级，各个设备会依次拿到计算图。每个设备的驱动程序，会遍历计算图的计算节点，并根据自己设备的特点和计算节点的具体参数，依靠一定的策略，计算出自己设备运行该节点的优先级。如果该节点的绑定优先级低于当前设备计算的优先级，说明当前设备更加适合运行该节点。当前设备驱动会将该节点的绑定设备设为当前设备，并修改绑定的优先级。

Tengine 还支持手工绑定设备策略，包括图级别和节点级别：既可以指定整个图跑在特定设备上，也可以指定某些特定的节点跑在特定的设备上。这样给用户提供了最大的灵活性和控制权。

在各个计算节点的运行设备确定之后，计算图按各个节点绑定的设备，被切割成多个子图，每个子图的所有节点都运行在一个设备上。

再接下来是准备执行阶段。这一阶段的主要任务是，在各个设备上计算子图的优化和为网络运行分配资源。计算子图的优化主要是算子的融合，例如，把 Batch Normalization 和 Convolution 融合：Convolution 的 weight 等于 weight 乘 alpha，而 bias 等于 bias 乘 alpha 再加 beta。对于 ReLU/ReLU6 等不带参数的 Activation 操作，也可以融入

Convolution。分配运行时资源主要是分配运行时需要的内存，这样，可以提前规划好内存使用和复用，减少推理时需要的内存大小，另外，在推理时不必来回分配和释放内存，提高了运行的速度。

在图执行时，Tengine 会调度所有输入数据都准备好的计算子图，到绑定的设备上运行。当一个子图执行完成时，会更新所有依赖于该子图的其他计算子图的状态。如果之前处于等待状态的计算子图的所有依赖子图都已经计算完成，该子图会被调度执行。

Tengine 实现了一系列高效高性能的计算算子。例如，对于矩阵乘，这一基本运算，Tengine 就针对不同平台，做了微架构级别的汇编优化。

当所有的计算子图都执行完毕，就可以通过计算图的输出 Tensor 取到计算结果。Tengine 同时提供了同步和异步两种方式来获取计算的结果。

下图是 Tengine 的架构图，Tengine 对外提供了清晰的接口定义，方便用户扩展自定义的功能：

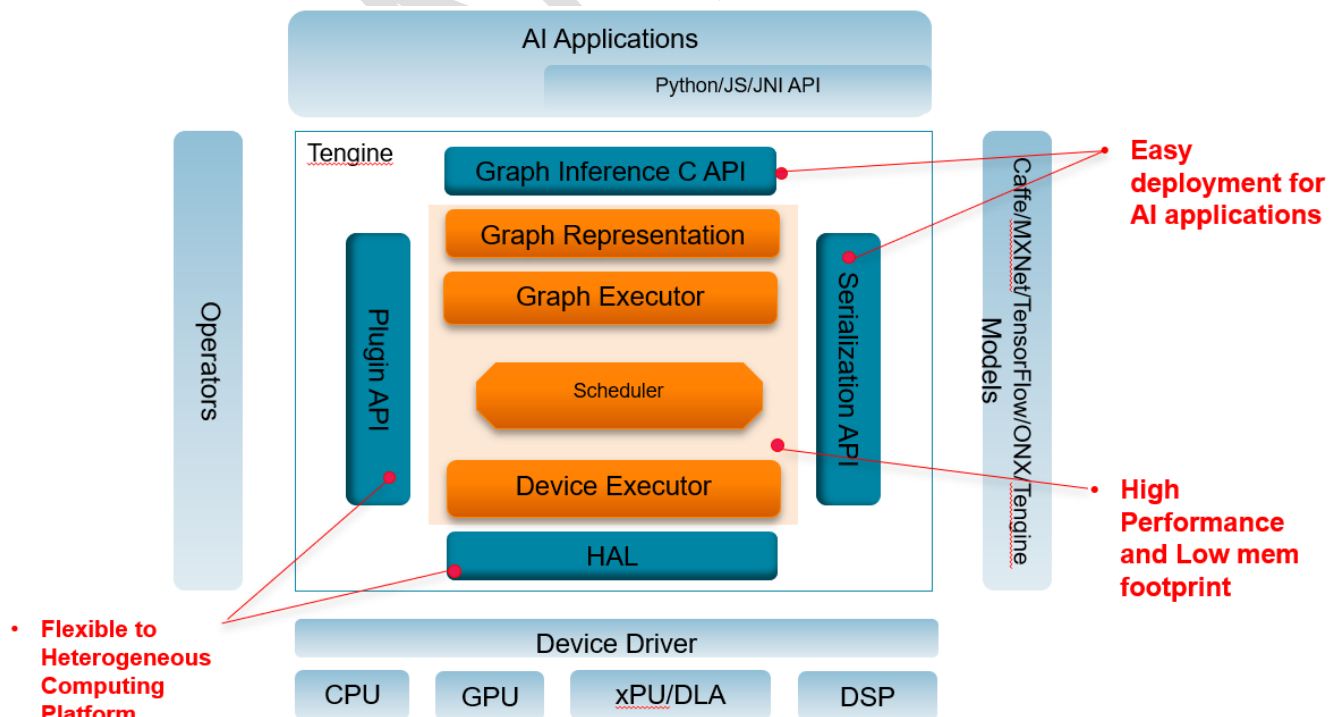


Figure 1 Tengine 框架

- **Graph Inference C API:** 提供给应用开发的 C API 接口。在 C API 接口之上，还可以构造出 python, Java, JavaScript 等接口。
- **Serialization API:** 是一个持久化的接口，以动态库的形式提供不同 AI 框架存储格式的加载，比如 Caffe/MXNet 等。客户也可以定制自己的格式，并实现对应的 serializer 模块来加载到系统中。
- **Plugin API:** 是为 Operator 的定义接口，为用户提供自定义 DL 操作的接口。
- **HAL:** HAL 层定义了和硬件的接口。不同类型的硬件，通过 HAL 层将自己的能力上报给 Tengine。Tengine 通过 HAL 调用硬件的计算资源以及和硬件进行数据交互。

1.2 Tengine 软件发布包

Tengine 商用版有二进制和源码两种发布方式。源码方式可以方便客户扩展功能。

Tengine 软件发布包主要包括数据、文档、预编译库、源码、插件、工具等目录。目录结构示例如下：

```
.
|-- data
|   |-- images
|   `-- models
|-- doc
|   |-- Tengine\ Release\ Note\ V1.0.pdf
|   |-- Tengine_Technical_Spec.pdf
|   `-- Tengine_User_Manual.pdf
|-- pre-built
|   |-- android_arm64
|   |-- linux_arm64
|   `-- toolchain.readme
|-- source
|   `-- classification
|-- tengine-plugin
|   `-- RKNN
`-- tools
    |-- auth_tools
    `-- x86
```

各个目录简要说明：

- Data: 存放随版本发布的 example 运行需要的模型和图像等资源。
- Doc: 版本相关文档。
- Pre-built: 预编译的二进制库, 根据版本情况分为 Android/Linux 或者 arm32/arm64 等版本。
- Source: 随版本发布的 example, 或者 tengine 源码。
- Tengine-plugin: tengine 插件, 包括 rknnplugin、nnirplugin、aipuplugin 等。
- Tools: 仅提供 x86 64bit 的版本工具。包括模型转换工具, 模型量化工具。

2 基本操作

2.1 通用指令

2.1.1 设置使用的 CPU 内核

目前 CPU 大多数是多核架构, 根据业务设计不同的性能的 SOC 的组合。为了能够根据业务合理分配 CPU 资源, Tengine 支持设置使用的 CPU 内核。虽然多核可以提升计算性能, 但也可能引入性能不稳定的问题。因为调度或者模型本身问题, 加速比与理想值很有可能不一致。

设置 CPU 内核有两种不同的方法:

1. 使用环境变量的方法设置
2. 在代码中调用 API 的方法设置。

默认使用的 CPU 核心是 SOC 中功能最强的核。(例如, 如果 SOC 为四核, 一个是双核 A72, 另一个是双核 A53, Tengine 默认将在双核 A72 上运行; 再比如, SOC 是四核 A53, Tengine 默认将在四核上运行)。

1) 环境变量

如果只在一个 CPU 核心上运行 demo，可以设置环境变量为（我们不建议将 tengine 设置在 0 核，容易与系统运行资源发生竞争），示例命令如下：

```
export TENGINE_CPU_LIST=3
```

如果要在双核上运行 demo，可以设置环境变量为：

```
export TENGINE_CPU_LIST=2,3
```

如果要在四核上运行 demo，可以设置环境变量为：

```
export TENGINE_CPU_LIST=0,1,2,3
```

2) 在代码中调用 API 方法

```
set_default_cpu(const int* cpu_list, int cpu_number)
```

注意：该函数必须在 init_tengine() 之前被调用，并且引用如下的头文件：

```
#include "cpu_device.h"
```

更多说明可以参考 4.3 添加自定义 CPU 设备。

2.1.2 设置 kernel mode

普通的量化推理采用混合精度计算，中间数据依然采用浮点进行缓存。在不影响模型准确率的情况下，模型内部可以全部采用整形类型进行计算和存储，可有效减少内存带宽，同时提高计算速度，减小资源消耗。不过这种实时量化的方法对有些模型会带来一定的精度损失。对于安防等对识别精度要求特别高的场景，这种精度损失是不可以接受的。Tengine 量化训练工具对模型进行量化训练使得模型精度保持不变。

kernel mode 将定义在线量化方法。有 3 种 kernel mode：FP32 模式、FP16 模式和 INT8 模式。默认为 FP32 模式。并且，有两种配置方法。

1) 使用环境变量进行配置

```
export KERNEL_MODE=0    //Float32 模式
export KERNEL_MODE=1    //Float16 模式
export KERNEL_MODE=2    //Int8 模式
```

通过一下方式查看变量是否设置成功

```
echo $KERNEL_MODE
```

2) 在代码中使用相关 API 进行配置

```
int val=1;          // val 表示 kernel 模式, 0 表示 FP32, 1 表示 FP16, 2 表示 INT8
set_graph_attr(graph, "kernel_mode", &val, sizeof(val))
```

5. 设置运行 graph 的设备

```
set_graph_device(graph, device_name)
```

2.2 算子使用多线程运行

个别算子在多核运行的情况下可以进一步设置多线程模式, 从而提升计算效率。同样有可能会引入性能不稳定的问题。

2.2.1 Pooling 多线程模式

在设置了多核运行的情况下, 该设置才会生效。代码如下:

```
int val=1;
set_graph_attr(graph, "pooling_mt", &val, sizeof(val))
```

2.2.2 FC 多线程模式

在设置了多核运行的情况下, 该设置才会生效。代码如下:

```
int val=1;
set_graph_attr(graph, "fc_mt", &val, sizeof(val))
```

2.3 支持 CPU 和 GPU 的异构计算

Tengine 的 GPU/CPU 异构调度的原理是:根据算子对计算图 (Graph) 进行切分, 切分的子图 (SubGraph) 再通过调度器分配给相应的 device。由于 GPU 的编程较复杂, 我们优先支持神经网络中的常用算子(例如:CONV,POOL,FC 等), 而对于某些网络中特有的算子 (例如检测网络 SSD 中的 PRIORBOX 等), 我们将这些算子分配给 CPU 进行计算。

下面我们将演示如何调用 Tengine 进行 GPU/CPU 异构调度,进行检测网络 MobilenetSSD 的推理加速。

我们的测试环境是 Linux, 测试平台是开发板 Firefly-RK3399:

```
GPU: Mali-T860
CPU: dual-core Cortex-A72 + quad-core Cortex-A53
```

Tengine 是通过调用 Arm Compute Library (ACL) 进行 GPU 加速, 我们使用的 ACL 版本为 19.02。

```
git clone https://github.com/ARM-software/ComputeLibrary.git
git checkout v18.05
scons Werror=1 -j4 debug=0 asserts=1 neon=0 opencv=1 embed_kernels=1 os=linux
arch=arm64-v8a
```

为了发挥 GPU 的最高性能, 我们需要设置 GPU 的频率到最高频率:

```
sudo su
echo "performance" >/sys/devices/platform/ff9a0000.gpu/devfreq/ff9a0000.gpu/governor
cat /sys/devices/platform/ff9a0000.gpu/devfreq/ff9a0000.gpu/cur_freq
```

显示的 GPU 频率应该是 800000000。

git clone Tengine 项目:

```
git clone https://github.com/OAID/Tengine.git
cp makefile.config.example makefile.config
```

在配置文件中打开开关 CONFIG_ACL_GPU=y, 并指定 ACL 路径:

```
CONFIG_ACL_GPU=y
ACL_ROOT=/home/firefly/ComputeLibrary
```

编译:

```
make -j4
make install
```

下载 MobilenetSSD 模型, 可以从 [Tengine models](链接:

<https://pan.baidu.com/s/1Ar9334MPeIV1eq4pM1eI-Q>) (psw: hhgc) 下载模型~/tengine/models/ 路径

下:

```
cd example/mobilenet_ssd
cmake -DTENGINE_DIR=/home/firefly/tengine
make
```

执行时需要设置一些环境变量：

```
export GPU_CONCAT=0          # 禁用 GPU 运行 concat，避免 CPU 和 GPU 之间频繁的数据传输
export ACL_FP16=1            # 启用 GPU fp16 模式
export ACL_NCHW=1            # 设置 ACL 的数据排布格式
export REPEAT_COUNT=100      # 重复运行 mssd，获取平均时间
taskset 0x1 ./MSSD -d acl_openc1 # -d acl_openc1 表示使用 GPU
...
```

设置 GPU_CONCAT=0，是为了避免 GPU/CPU 在 concat 这一层在 GPU/CPU 频繁来回传输数据造成性能损失；

设置 ACL_FP16=1，是支持 GPU 用 float16 的数据格式进行推理计算；

设置 REPEAT_COUNT=100，是让算法重复执行 100 次，取平均时间作为性能数据；

设置 ACL_NHWC=1，是设置 ACL 数据的排布格式为 NHWC

taskset 0x1 用于绑定 CPU0（1A53）执行程序；

执行的时候需要加 -d acl_openc1 来打开使用 gpu 的开关

3 如何转换自己的模型

Tengine 支持在线转换模型，即支持直接解析 Caffe、Onnx、Mxnet、Tensorflow/Tf-Lite 等模型。也支持离线转换模型，即使用 tengine 提供的工具转成 tmfile 模型。推荐使用离线转换模型。

3.1 转换为 Tengine 模型

3.1.1 工具所在路径

在发布包的 tools/x86 下看到转换工具 convert_model_to_tm。

3.1.2 使用方法

convert_model_to_tm 可以方便的将 Caffe、Onnx、Mxnet、Tensorflow 或 Tf-lite 模型转换成 Tengine 模型。目前仅提供 Linux x86 64bit 平台的工具。

命令格式为：

```
[Usage]: ./install/tool/convert_model_to_tm [-h] [-f file_format] [-p proto_file] [-m model_file] [-o output_tmfile]
```

file_format 可以指定：caffe、caffe_single、onnx、mxnet、tensorflow、tflite 当输入只有一个模型文件时，不需要指定“-p”选项，只指定“-m”选项即可。“-h”是“help”选项。

下面是将 Caffe 模型和 Tensorflow 模型（squeezenet）转成 Tengine 模型的使用示例：

```
cd ~/tengine
./install/tool/convert_model_to_tm -f caffe -p models/sqz.prototxt -m
models/squeezenet_v1.1.caffemodel -o models/squeezenet.tmfile
./install/tool/convert_model_to_tm -f tensorflow -m models/squeezenet.pb -o
models/squeezenet_tf.tmfile
```

3.2 Tengine 模型量化工具

CNN 模型属于同一层的参数值会分布在一个较小的区间内，记下这个最小值和最大值，比如采用 8 位数进行量化，可以把同一层的所有参数都线性映射到区间 $[-127, 127]$ 之间的 255 个 8 位整数中的最接近的一个数。

Tengine 提供将 tengine 模型进行量化的工具。

3.2.1 工具所在路径

PACKER_ROOT/tools/x86/quant_tm_model

3.2.2 使用方法

目前仅提供 Linux x86 64bit 平台的工具。

你可以使用以下命令运行量化工具。

命令行：

```
[Usage]: ./install/tool/quant_tm_model [-h] [-q quant_mode] [-i input_tmfile] [-o output_tmfile]
```

详细参数说明：

- quant_mode 0 -- fp16, 1 -- int8
- input_tmfile The tengine FP32 mode
- output_tmfile The tengine FP16 mode or INT8 mode

4 高阶使用

4.1 添加用户自定义 kernel

如果 Tengine 的商业版本中不包含源代码，可以使用 Tengine C API 添加用户自定义算子。下面展示如何自定义 Convolution 算子，替换 Tengine 中的 Convolution 算子并在设备上运行。

1) 首先，初始化 tengine，并创建一个 conv 图形。

```
int c, h, w;

c = 3;
h = 16;
w = 16;

init_tengine();

graph_t graph = create_conv_graph(c, h, w);
```

2) 声明自定义 Kernel 的 prerun 函数

```
int my_prerun(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[],
int input_num, struct custom_kernel_tensor* outputs[], int output_num, int
dynamic_shape)
{
    std::cout << __FUNCTION__ << " called\n";
    std::cout << "input_num " << input_num << "\n";
```

```

    std::cout << "output_num " << output_num << "\n";

    return 0;
}

```

3) 声明自定义 Kernel 的 run 函数

```

int my_run(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[], int
input_num,
           struct custom_kernel_tensor* outputs[], int output_num)
{
    std::cout << __FUNCTION__ << " called\n";

    my_conv_param* param = ( my_conv_param* )ops->kernel_param;

    struct custom_kernel_tensor* output = outputs[0];
    struct custom_kernel_tensor* input = inputs[0];
    struct custom_kernel_tensor* weight = inputs[1];
    struct custom_kernel_tensor* bias = nullptr;

    if(input_num > 2)
        bias = inputs[2];

    int n = output->dim[0];
    int out_c = output->dim[1];
    int out_h = output->dim[2];
    int out_w = output->dim[3];
    int stride_h = param->stride_h;
    int stride_w = param->stride_w;
    int pad_h0 = param->pad_h0;
    int pad_w0 = param->pad_w0;

    int kernel_h = param->kernel_h;
    int kernel_w = param->kernel_w;

    int in_c = input->dim[1];
    int in_h = input->dim[2];
    int in_w = input->dim[3];

    float* src_ptr = ( float* )input->data;
    float* dst_ptr = ( float* )output->data;

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < out_c; j++)
        {

```

```

float bias_data = 0;

if(bias)
{
    float* bias_ptr = ( float* )bias->data;
    bias_data = bias_ptr[j];
}

for(int h = 0; h < out_h; h++)
    for(int w = 0; w < out_w; w++)
    {
        float* weight_ptr = ((( float* )weight->data) + j * in_c *
kernel_h * kernel_w);
        float sum = 0;

        for(int c = 0; c < in_c; c++)
        {
            int start_h = h * stride_h - pad_h0;
            int start_w = w * stride_w - pad_w0;
            float* src_hw = src_ptr + c * in_h * in_w;
            float* weight_base = weight_ptr + c * kernel_h * kernel_w;

            /* one slide window */
            for(int k_h = 0; k_h < kernel_h; k_h++)
            {
                int src_h = start_h + k_h;

                if(src_h < 0 || src_h >= in_h)
                    continue;

                for(int k_w = 0; k_w < kernel_w; k_w++)
                {
                    int src_w = start_w + k_w;

                    if(src_w < 0 || src_w >= in_w)
                        continue;

                    float wt = weight_base[k_h * kernel_w + k_w];
                    float src = src_hw[src_h * in_w + src_w];

                    sum += wt * src;
                }
            }

            sum += bias_data;

```

```

        dst_ptr[h * out_w + w] = sum;
    }

    dst_ptr += out_h * out_w;
}

src_ptr += in_c * in_h * in_w;
}

return 0;
}

```

4) 声明自定义 Kernel 的 postrun 函数

```

int my_postrun(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[],
int input_num,
        struct custom_kernel_tensor* outputs[], int output_num)
{
    std::cout << __FUNCTION__ << " called\n";
    std::cout << "input_num " << input_num << "\n";
    std::cout << "output_num " << output_num << "\n";

    return 0;
}

```

5) 声明自定义 Kernel 的释放函数

```

void my_release(struct custom_kernel_ops* ops)
{
    std::cout << __FUNCTION__ << " called\n";
}

```

6) 初始化用户自定义 kernel

```

void init_custom_kernel(struct custom_kernel_ops* ops)
{
    ops->kernel_name = "custom_conv";
    ops->op = "Convolution";
    ops->force = 1;

    ops->infer_shape = nullptr;
    ops->inplace_info = nullptr;
    ops->bind = nullptr;
}

```

```
ops->reshape = nullptr;
ops->prerun = my_prerun;
ops->run = my_run;
ops->postrun = my_postrun;
ops->release = my_release;
ops->kernel_param = &conv_param;
ops->kernel_param_size = sizeof(conv_param);
}
```

在 main 函数中调用

```
init_custom_kernel(&conv_custom_kernel);
```

7) 声明填充给自定义 Kernel 函数

```
void fill_custom_kernel_param(node_t node, struct custom_kernel_ops* ops)
{
    my_conv_param* param = ( my_conv_param* )ops->kernel_param;

    get_node_attr_int(node, "kernel_h", &param->kernel_h);
    get_node_attr_int(node, "kernel_w", &param->kernel_w);
    get_node_attr_int(node, "stride_h", &param->stride_h);
    get_node_attr_int(node, "stride_w", &param->stride_w);
    get_node_attr_int(node, "dilation_h", &param->dilation_h);
    get_node_attr_int(node, "dilation_w", &param->dilation_w);
    get_node_attr_int(node, "pad_h", &param->pad_h0);
    get_node_attr_int(node, "pad_w", &param->pad_w0);
    get_node_attr_int(node, "output_channel", &param->out_channel);
    get_node_attr_int(node, "group", &param->group);

    param->pad_h1 = param->pad_h0;
    param->pad_w1 = param->pad_w0;
}
```

8) 获取 conv 节点，并将其参数填充给自定义 Kernel

```
node_t my_node = get_graph_node(graph, "conv");

fill_custom_kernel_param(my_node, &conv_custom_kernel);
```

9) 设置自定义 Kernel 运行的设备


```
if(set_custom_kernel(my_node, "ANY_DEVICE", &conv_custom_kernel) < 0)
{
    std::cerr << "set_custom_kernel failed: ERRNO: " << get_tengine_errno() << "\n";
    return 1;
}
```

10) 预运行 Graph

```
prerun_graph(graph)
```

11) 设置 Input 数据, weight 数据, bias 数据

代码略。

12) 运行 Graph

```
run_graph(graph)
```

13) 获取输出数据

代码略。

14) 释放资源等

代码略。

4.2 添加自定义算子

如果商业版本中包含 Tengine 源代码, 可以参考以下内容在 Tengine 源码中添加新的 operator。

要添加新的 operator, 你需要:

- 1) 在目录 [Tengine/operator/](#) 中注册新的 operator。

Operator schema 为 operator 的功能定义了一个接口, 该接口独立于 operator 的实现。operator schema 需要定义:

- op 名称
- op 输入与输出

- op 参数
- infershape 函数：用于预运行(Prerun)阶段的张量形状推断。
- 2) 在目录 [Tengine/executor/](#)中实现 operator。

这是 operator 的具体实现。对于不同的参数（例如，conv1x1、conv3x3）或架构（例如，armv8/armv7）可以有多个实现。

4.2.1 以 Scale Operator 为例

在这个部分，我们以 Scale operator 为例，如何实现为 Tengine 添加自定义 operator。

Step 1: 创建 Operator 头文件

operators 分为两种类型：

1. 没有内部参数(parameters)的 Operator: `template <typename T> OperatorNoParam`
2. 有内部参数(parameters)的 Operator: `template <typename T, typename P> OperatorWithParam`

scale operator 是带有参数的运算符

- 创建 ScaleParam 文件 [operator/include/operator/scale_param.hpp](#):

```
struct ScaleParam {
    int    axis;
    int    num_axes;
    int    bias_term;

    DECLARE_PARSER_STRUCTURE(ScaleParam) {
        DECLARE_PARSER_ENTRY(axis);
        DECLARE_PARSER_ENTRY(num_axes);
        DECLARE_PARSER_ENTRY(bias_term);
    };
};
```

- 创建 Scale 文件 [operator/include/operator/scale.hpp](#):

```
class Scale: public OperatorWithParam<Scale,ScaleParam> {
public:

    Scale() { name_="Scale"; }
    Scale(const Scale&)= default;
```

```

~Scale() {}

void SetSchema(void) override;
};

```

Step 2: 创建 Operator Cpp 文件

在文件 [operator/operator/scale.cpp](#) 中设置 operator schema:

```

void Scale::SetSchema(void)
{
    Input({"input:float32", "gamma:float32", "bias:float32"})
    .Output({"output:float32"})
    .SetAttr("axis", 1)
    .SetAttr("num_axes", 1)
    .SetAttr("bias_term", 0)
    .SetDoc(R"DOC(Scale: only caffe flavor scale)DOC");
}

```

在文件 [operator/operator/Makefile](#) 中加入 `obj-y+=scale.o`。

Step 3: 在 Operator Plugin 中注册 Operator

在文件 [operator/plugin/init.cpp](#) 中加入对 scale operator 的注册内容:

```
RegisterOp<Scale>("Scale");
```

Step 4: 在 Executor 文件夹中实现算法

在文件 [executor/operator/common/scale.cpp](#) 中添加 scale operator 的实现, 并在函数

`RegisterScaleNodeExec` 中注册该实现:

```

namespace ScaleImpl
{
    struct ScaleOps: public NodeOps
    {
        bool Run(Node * node)
        {
            // your implementation
        }
    };
}

using namespace ScaleImpl;
void RegisterScaleNodeExec(void)
{
    ScaleOps * ops=new ScaleOps();

    NodeOpsRegistryManager::RegisterOPIplementor("common",

```

```
        "Scale",ops);
    }
```

在编译文件 [executor/operator/common/Makefile](#) 中加入 `obj-y+=scale.o`。

Step 5: 在 Executor Plugin 中注册实现

在文件 [executor//operator/plugin/init.cpp](#) 中加入 `RegisterScale_NodeExec`:

```
extern void RegisterScale_NodeExec(void);
...
RegisterScale_NodeExec();
```

4.3 添加自定义 CPU 设备

目前 CPU 大多数是多核架构，根据业务设计不同的性能的 SOC 的组合。为了能够根据业务合理分配 CPU 资源，Tengine 支持通过添加自定义 CPU 设备来分配 CPU 资源。

Tengine 可以自动探测 CPU 作为运行设备，也可以手动创建一个新的 CPU 设备。

4.3.1 通过环境变量配置默认 CPU 设备

Tengine 支持通过环境变量配置模型 CPU 设备。配置方式如下：

```
export TENGINE_CPU_LIST=0,1,2,3,4
```

4.3.2 预定义的 CPU

Tengine 预定义了一些 CPU。用户可以通过接口 `get_predefined_cpu()` 获取。

- **rk3399** **4xA53 + 2xA72**
- **a63** **4xA53**
- **rk3288** **4xA17**
- **r40** **4xA7**
- **kirin960** **4xA53 + 4xA73**
- **apq8096** **4xA72**

4.3.3 手动设置 CPU

下面展示如何为 CPU(4XA55)创建一个设备。

1. 创建新的 CPU 信息。

cpu_cluster 和 cpu_info 的声明在 cpu_device.h 头文件中。

```
struct cpu_cluster my_cluster = {4, 1520, &nbsp;CPU_A55, &nbsp;ARCH_ARMV8,  
                                32 << 10, 52 << 10, {0, 1, 2, 3}};  
struct cpu_info my_soc = {"my_soc", "my_board", 1, 0, &my_cluster, -1, NULL};
```

2. 创建设备

在 init_tengine()之后调用以下代码。

```
int&nbsp;cpu_list={0,1,2,3};  
set_online_cpu(&my_soc,&nbsp;cpu_list, 4);  
create_cpu_device("my_device", &my_soc);
```

3. 设置默认设备

```
set_default_device("my_device");
```

4. 运行

Tengine 可以在创建的设备上运行。

```
prerun_graph(graph);  
run_graph(graph,1);
```

4.4 NPU 使用简介

Tengine 框架支持 RKNN、NNIE、AIPU 等 NPU 的调用。这些 NPU 通过插件的方式在 tengine 框架中扩展。后续更多的 NPU 的支持也采取这种方式。

NPU 插件以二进制方式提供，比如 libxxxplugin.so。在 tengine 初始化后，通过动态库加载的方式加载 libxxxplugin.so。后面就可以创建图像加载模型并进行推理了。

4.4.1 RKNN

- 模型准备

通过 rknn-toolkit 工具转换好 RKNN 模型。

- API 调试步骤

初始化 tengine 后，动态库加载 librknplugin.so。

```
if (load_tengine_plugin("rknnplugin","librknnplugin.so","rknn_plugin_init") != 0 )
{
    std::cout << "load rknn plugin faield.\n";
}
```

然后创建 rk3399pro 图形。model_file 及为前面准备好的 rknn 模型文件的路径。

```
context_t rknn_context = nullptr;
graph_t graph = create_graph(rknn_context, "rk3399pro", model_file);
```

后面即可设置输入数据并运行图形。

```
set_tensor_buffer(input_tensor, img.data, len);
prerun_graph(graph);
run_graph(graph,1);
```

4.4.2 NNIE

- 模型准备

通过 NNIE 配套模型转换工具转换好 NNIE 模型。

- API 调试步骤

初始化 tengine 后，加载动态库 libnnieplugin.so。

```
if (load_tengine_plugin("nnieplugin", "libnnieplugin.so", "nnie_plugin_init") != 0 )
{
    std::cout << "load nnie plugin failed.\n";
}
```

然后创建 nnie 图形。model_file 及为前面准备好的 nnie 模型文件的路径。

```
context_t nnie_context = nullptr;
graph_t graph = create_graph(nnie_context, "nnie", model_file);
```

后面即可设置输入数据并运行图形。

```
set_tensor_buffer(input_tensor, img.data, len);
prerun_graph(graph);
run_graph(graph, 1);
```

4.4.3 AIPU

- 模型准备

获取 AIPU 模型。目前未开放转换工具。

- API 调试步骤

初始化 tengine 后，加载动态库 libaipuplugin.so。

```
if(load_tengine_plugin("aipuplugin", "libaipuplugin.so", "aipu_plugin_init") != 0 )
```

```
{
    std::cout << "load nnie plugin faield.\n";
}
```

然后创建 AIPU 图形。model_file 及为前面准备好的 AIPU 模型文件的路径。

```
context_t aipu_context = nullptr;
graph_t graph = create_graph(aipu_context, "aipu", model_file);
```

设置输入数据并运行图形。

```
set_tensor_buffer(input_tensor, img.data, len);
prerun_graph(graph);
run_graph(graph, 1);
```

4.5 内存加载模型

很多场景下模型的保护是个很重要的问题，加密传输，端侧解密后，数据存储在内存中。Tengine 支持从内存加载模型。主要参考 create_graph API。在该 API 的第二个参数模型格式字符串后面追加“:m”，后面参数紧跟内存地址参数和内存大小参数。Caffe 模型需要两块内存。

- 首先获取模型在内存中的指针和大小

```
int text_size, bin_size;
void *text_mem, *bin_mem;

if(!get_file_mem_size(text_file, &text_mem, &text_size))
{
    std::cerr << "cannot load " << text_file << "\n";
    return -1;
}

if(!get_file_mem_size(model_file, &bin_mem, &bin_size))
{
    std::cerr << "cannot load " << model_file << "\n";
    return -1;
}
```


- 从内存加载 caffe 模型。

```
graph_t graph = create_graph(nullptr, "caffe:m", ( const char* )text_mem, text_size,
bin_mem, bin_size);
```

4.6 Dump 中间节点数据

某些场景下用户需要获取运行图中间某个节点的数据，比如人脸检测的特征值。下面介绍获取中间节点数据的方法。

- 首先创建 Graph 后，通过名称获取中间节点

```
graph_t graph = create_graph(nullptr, "caffe", text_file, model_file);
.. .. .

node_t pool_node = get_graph_node(graph, "pool1");
```

- 预运行 Graph

```
prerun_graph(graph);
```

- 打开节点的 dump 功能，启动节点 dump

```
do_node_dump(pool_node, NODE_DUMP_ACTION_ENABLE);
do_node_dump(pool_node, NODE_DUMP_ACTION_START);
```

- 运行 Graph

```
run_graph(graph, 1);
```

- 获取节点 dump 数据

```
void** pool_buf;
int buf_size = 10;
pool_buf = ( void** )malloc(buf_size * sizeof(void*));
int pool_size = get_node_dump_buffer(pool_node, pool_buf, buf_size);

for(int i = 0; i < pool_size; i++)
{
```

```
struct tensor_dump_header* header = ( struct tensor_dump_header* )pool_buf[i];
std::cout << "dim: ";

for(int j = 0; j < header->dim_number; j++)
    std::cout << header->dim[j] << " ";
    std::cout << " data: " << header->data << " from device: " <<
get_node_device(pool_node) << "\n";
}
```

5 调试技巧

5.1 设置 tengine 的 log 级别和输出函数

Tengine 支持设置 Log 级别。可以通过设置环境变量和 API 两种方式设置。

5.1.1 通过环境变量设置

```
export TENGINE_LOG_LEVEL=7
```

详细定义如下：

```
enum log_level
{
    LOG_EMERG,    // 0
    LOG_ALERT,    // 1
    LOG_CRIT,     // 2
    LOG_ERR,      // 3
    LOG_WARNING,  // 4
    LOG_NOTICE,   // 5
    LOG_INFO,     // 6
    LOG_DEBUG     // 7
};
```

5.1.2 通过 API 设置

需要区分 Linux 和 Android 两种环境。

Linux 环境下日志直接输出到终端，一般仅设置 Log 级别：

```
set_log_level(LOG_DEBUG);
```

Android 环境下因为 Android 的日志是要输出到 Logcat 中的，所以先声明 Log 输出函数：

```
void tengine_core_log_print_t(const char* log){
    LOGE("TENGINE CORE LOG:%s\n",log);
}
```

然后设置 log 级别和 Log 输出函数。

```
set_log_level(LOG_DEBUG);
set_log_output(tengine_core_log_print_t);
```

这样才能够在 Logcat 中看到 Tengine 的日志。

5.2 开启 PROF_TIME

在模型的性能评估时会用到该环境变量。该功能开启后，会在模型退出运行时打印出模型的整个运行状况，包括以下内容：

- 模型中所有 OP 运行时间。从高到底排序。
- 模型中每一层的运行时间，输入数据结构，输出数据结构，kernel 结构等。从高到底排序。

打印输出示例如下：

```
=== /home/cmeng/tengine/tengine_model_test/models/squeezenet_int8.tmfile: time stats by operator: ===
total time: 1739305 us, repeat 1
PER RUN: time 1739305 us on 775.50 Mfops, RATE: 445.86 Mfops
0: Dropout used 2 us (0.00%)
1: Concat used 9974 us (0.57%)
2: Softmax used 50 us (0.00%)
3: Convolution used 1718946 us (98.83%)
4: Pooling used 10333 us (0.59%)
```

实际还有更详细的打印，这里不再举例。必要情况下请将详细输出 log 发送给我们分析。

5.2.1 打开方式

设置环境变量：

```
export PROF_TIME=1
```

5.3 gdb 工具的使用

Linux 环境下使用 gdb, Android 环境下使用 gdbserver, 这里不再详细介绍。

6 测试例程

商业版本中已经根据版本需求预编译好演示 Classify 程序。

6.1 测试数据和模型

在\${Tengine_ROOT}/data/文件夹中存放了测试数据和模型

```
data
|-- images
|   |-- cat.jpg
|-- models
|   |-- squeezenet.tmfile
|   |-- synset_words.txt
```

2 directories, 3 files

6.2 编译

- 修改文件'source/classification/linux_build.sh', 设置正确的 TENGINE_DIR。
- 按照如下步骤编译。

```
cd source/classification
mkdir build
cd build
../linux_build.sh
make
```

6.3 运行

- 声明 LD_LIBRARY_PATH, 将 tengine 库和其他依赖库加入其中。
- 进入到可执行程序的目录:

```
cd ${Tengine_ROOT}/pre-built/linux_arm32/bin
```

- 测试图像:

```
./Classify -f tengine -m ../../data/models/squeezenet.tmfile -  
l ../../data/models/synset_words -i ../../data/images/cat.jpg
```

更多信息参考'source/classification/readme.txt'。

7 Tengine AAR

为方便 android 用户开发集成, 直接使用 java 开发 app 而不依赖 C/C++, Tengine 提供 AAR 开发包。

7.1 Tengine AAR API

Tengine AAR 现在封装了一些常用的接口, 后续会根据需求将更多 tengine c api 接口封装进来。

目前已经实现的 AAR API 列表见下表。

API 名称	描述
init_tengine	Tengine 初始化
release_tengine	释放 tengine
get_tengine_version	获取版本号
create_graph	创建 Graph
destroy_graph	销毁 Graph
get_graph_input_tensor	获取 Graph 的输入 Tensor

get_graph_input_node_number	获取 Graph 的数据节点数量
get_graph_input_node	获取 Graph 的数据节点
get_graph_output_tensor	获取 Graph 的输出 Tensor
get_graph_output_node_number	获取 Graph 的数据节点数量
get_graph_output_node	获取 Graph 的数据节点
get_node_output_number	获取节点的输出个数
get_node_output_tensor	获取节点的数据 tensor
release_graph_node	释放节点
get_graph_tensor	获取Graph的Tensor
release_graph_tensor	释放Tensor
get_tensor_shape	获取Tensor的形状
get_tensor_buffer	获取Tensor的buffer
set_tensor_shape	设置Tensor的形状
get_tensor_buffer_size	获取Tensor的Buffer大小
set_tensor_bufer	设置Tensor的Buffer
prerun_graph	预运行Graph
run_graph	运行Graph
wait_graph	等待Graph运行结束
postrun_graph	停止运行Graph
get_tengine_errno	获取错误码

更多 API 的详细定义请参考 Tengine C API。

7.2 Tengine AAR 的使用

Tengine AAR 现在只能在 Android Studio 中使用并不支持 Eclipse。在 Android Studio 使用步骤如下。

1. 在 Android Studio 中创建一个支持 native 代码的项目。
2. 获取 tengine aar 包 tengine.aar。
3. 把 tengine.aar 包拷贝到 app/libs 目录下。
4. 修改 app/build.gradle 文件按以下修改:
 - a. 黄色是要添加的内容

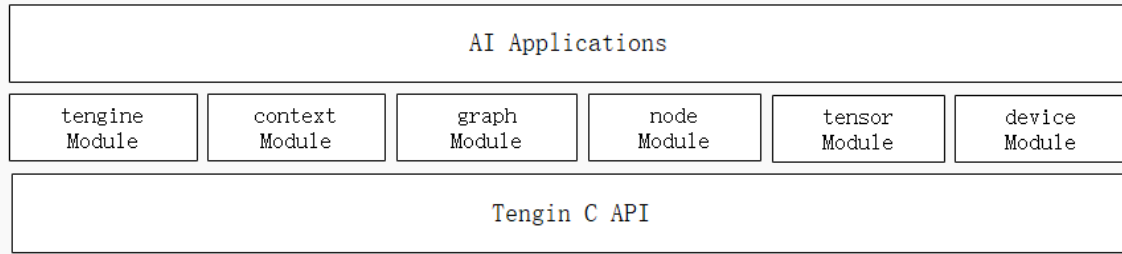
```
dependencies{  
    ...  
    compile(name: 'tengine', ext: 'aar')  
}
```

- b. minSdkVersion 如果不是 21 请改为 21。
 - c. 修改 ndk 配置为: ndk{ abiFilters "armeabi-v7a" , " arm64-v8a" }
5. 在代码中通过 com.openailab.jni.tengine.xxxx 调用 aar api。

注意：以上是基于 tengine 的 aar 包名为 tengine.aar 修改的步骤，比如 tengine-release.aar 那么需要把对应为 tengine 改为 tengine-release。

8 PyTengine

PyTengine 是针对 OPEN AI LAB 推出的 Tengine 的一组 Python 接口，方便用户既可以使用 Python 下的库进行快速开发，同时又可以享受 Tengine 带来的硬件加速效果。



PyTengine 结构

PyTengine 由六个模块组成：Tengine/Context/Graph/Node/Tensor/Device

Tengine: 提供错误反馈、log 设置、plugin 管理、模块初始化及释放等

Context: 提供 Context 的创建及管理方法

Graph: 提供 Graph 的创建及管理方法，可用于导入已保存模型

Node: 提供 Node 相关的操作及属性

Tensor: 提供 Tensor 相关的输入/输出及其他操作

Device: 提供 Device 的创建、销毁及模式和属性设置等操作

8.1 安装

PyTengine 有两种安装方式，但是都依赖 Tengine C 库。

8.1.1 使用源码安装

商用版本发布包中获取源码包，或者联系我们获取。

1) 安装

进入 PyTengine 目录，查看源码结构为：setup.py 文件和 tengine 目录及其下面文件

运行安装指令：

```
python setup.py install
```

2) 配置 Tengine 环境

pytengine 是建立在 Tengine 环境下的 python 接口，所以使用 pytengine 前需要配置 tengine 的库路径，指令为：

```
export LD_LIBRARY_PATH={TENGINE_ROOT}:$LD_LIBRARY_PATH
```

3) 检查是否安装成功

进入 python 环境下：运行

```
from tengine import tg
```

如果没有报错，则安装完成

4) 卸载

如果想要卸载 PyTengine，运行下面指令

```
pip uninstall pytengine
```

8.1.2 使用 whl 安装包安装

如果没有获得源代码而是获得了安装包，可以运行以下命令安装：

```
pip install pytengine-0.9.1-py2-none-any.whl
```

8.2 PyTengine API 介绍

8.3 常数

`/* tensor 的数据类型 */`

```
(tg.TENGINE_DT_FP32,  
tg.TENGINE_DT_FP16,  
tg.TENGINE_DT_INT8,  
tg.TENGINE_DT_UINT8,  
tg.TENGINE_DT_INT32,  
tg.TENGINE_DT_INT16) = map(int, range(6))
```

`/* layout 格式 */`

```
(tg.TENGINE_LAYOUT_NCHW,  
tg.TENGINE_LAYOUT_NHWC) = map(int, range(2))
```

`/* tensor type: the content changed or not during inference */`

```
(tg.TENSOR_TYPE_UNKNOWN,  
tg.TENSOR_TYPE_VAR,  
tg.TENSOR_TYPE_CONST,  
tg.TENSOR_TYPE_INPUT,  
tg.TENSOR_TYPE_DEP) = map(int, range(5))
```

`/* node 转储操作定义 */`

```
(tg.NODE_DUMP_ACTION_DISABLE,  
tg.NODE_DUMP_ACTION_ENABLE,  
tg.NODE_DUMP_ACTION_START,  
tg.NODE_DUMP_ACTION_STOP,  
tg.NODE_DUMP_ACTION_GET) = map(int, range(5))
```

`/* graph 性能动作定义 */`

```
(tg.GRAPH_PERF_STAT_DISABLE,  
tg.GRAPH_PERF_STAT_ENABLE,  
tg.GRAPH_PERF_STAT_STOP,  
tg.GRAPH_PERF_STAT_START,
```

```
tg.GRAPH_PERF_STAT_RESET,
tg.GRAPH_PERF_STAT_GET) = map(int, range(6))
```

```
/* quant mode */
(tg.TENGINE_QUANT_FP16,
tg.TENGINE_QUANT_INT8,
tg.TENGINE_QUANT_UINT8) = map(int, range(3))
```

```
/* 日志级别定义 */
```

```
(tg.LOG_EMERG,
tg.LOG_ALERT,
tg.LOG_CRIT,
tg.LOG_ERR,
tg.LOG_WARNING,
tg.LOG_NOTICE,
tg.LOG_INFO,
tg.LOG_DEBUG
) = map(int, range(8))
```

```
/* todo: should add suspend? */
```

```
(
    tg.GRAPH_STAT_CREATED,
    tg.GRAPH_STAT_READY,
    tg.GRAPH_STAT_RUNNING,
    tg.GRAPH_STAT_DONE,
    tg.GRAPH_STAT_ERROR
) = map(int, range(5))
```

```
/* graph_exec_event */
```

```
(
    tg.GRAPH_EXEC_START,
    tg.GRAPH_EXEC_SUSPEND,
    tg.GRAPH_EXEC_RESUME,
    tg.GRAPH_EXEC_ABORT,
    tg.GRAPH_EXEC_DONE
) = map(int, range(5))
```

```
/* device_policy */
```

```
(
```

```
tg.DEFAULT_POLICY,
tg.LATENCY_POLICY,
tg.LOW_POWER_POLICY
) = map(int,range(3))
```

8.4 API 介绍

8.4.1 Tengine 模块的 API

<code>__init__</code>	初始化tengine
<code>__del__</code>	释放tengine资源
<code>__version__</code>	获取tengine版本号
<code>requestTengineVersion</code>	检查tengine版本号是否支持
<code>setDefaultDevice</code>	设置默认设备
<code>getDefaultDevice</code>	获取默认设备

8.4.2 Graph 模块的 API

<code>__init__</code>	创建graph
<code>__del__</code>	销毁graph
<code>__add__</code>	将多个graph合并为一个
<code>save</code>	保存graph到文件
<code>setLayout</code>	设置graph的布局类型 (layout)
<code>setNode</code>	设置graph的输入节点及输出节点
<code>getInputNodeNumber</code>	获取graph的输入节点 (node) 个数
<code>getInputNodeByIdx</code>	通过ID获取graph的输入节点 (node)
<code>getOuptNodeByIdx</code>	通过ID获取graph的输出节点 (node)

<i>getOutputTensor</i>	通过ID获取graph的输出节点的张量
<i>getInputTensor</i>	通过ID获取graph的输入节点的张量
<i>doPerfStat</i>	启用或禁用性能统计
<i>getPerfStat</i>	获取性能统计信息
<i>dump</i>	dump图的结构
<i>setAttr</i>	设置graph的参数
<i>getAttr</i>	获取graph的参数
<i>setGdMethod</i>	设置graph的梯度下降方法
<i>getTensorByName</i>	通过名称获取graph的张量
<i>preRun</i>	graph的预运行
<i>Run</i>	运行graph
<i>wait</i>	等待graph
<i>postRun</i>	停止运行graph并释放graph占据的资源

8.4.3 Node 模块 API

<i>__init__</i>	创建graph的节点(node)
<i>__del__</i>	释放graph的节点 (node)
<i>__name__</i>	获取节点 (node) 名称
<i>__op__</i>	获取节点 (node) 的操作 (operation)
<i>getInputTensorByIdx</i>	通过ID获取节点 (node) 的输入张量 (tensor)
<i>getOutputTensorByIdx</i>	通过ID获取节点 (node) 的输出张量 (tensor)
<i>setInputTensorByIdx</i>	设置节点 (node) 的输入张量 (tensor)
<i>setOutputTensorByIdx</i>	设置节点 (node) 的输出张量 (tensor)

<i>getOutputNumber</i>	获取节点 (node) 的输出张量 (tensor) 个数
<i>getInputNumber</i>	获取节点 (node) 的输入张量 (tensor) 个数
<i>setAttr</i>	设置 (不存在的添加) 节点 (node) 参数(整型、浮点型、字符串类型)
<i>getAttr</i>	获取节点 (node) 的参数 (整型、浮点型、字符串型)
<i>setKernel</i>	设置用户内核(kernel)
<i>removeKernel</i>	移出用户内核 (kernel)
<i>setDevice</i>	设置node的执行设备
<i>getDevice</i>	获取node的执行设备
<i>dump</i>	使用节点的dump功能
<i>getDumpBuf</i>	获取节点的dump数据

8.4.4 Tensor 模块 API

<i>__init__</i>	创建graph的张量
<i>__del__</i>	释放graph的张量
<i>__name__</i>	获取张量(tensor)的名称
<i>__len__</i>	获取张量(tensor)的缓存大小
<i>shape</i>	设置 (获取) 张量 (tensor) 的形状 (shape)
<i>getbuffer</i>	获取张量 (tensor) 的缓存 (buffer)
<i>buf</i>	设置 (获取) 张量 (tensor) 的缓存
<i>getData</i>	获取张量 (tensor) 的数据 (data)

<i>setData</i>	设置张量 (tensor) 的数据 (data)
<i>dtype</i>	获取 (设置) 张量的数据类型
<i>setQuantParam</i>	设置tensor的量化参数
<i>getQuantParam</i>	获取tensor的量化参数

8.4.5 模块 DeviceAPI

<i>__init__</i>	创建设备
<i>__del__</i>	销毁设备
<i>policy</i>	获取 (设置) 设备策略

8.4.6 context 模块 API

<i>__init__</i>	创建context
<i>__del__</i>	销毁context
<i>getDevNumber</i>	获取context的设备数量
<i>getDevByIdx</i>	获取context设备名称
<i>addDev</i>	将设备添加到context中
<i>rmDev</i>	将设备移除出context

8.5 Tengine

8.5.1 *__init__*

使用	<i>from tengine import tg</i>
介绍	tg会自动加载libtengine.so库并调用init_tengine初始化,

	对tengine库进行初始化
--	----------------

8.5.2 __del__

使用	<code>exit()</code> or <code>del tg</code>
介绍	退出python环境或介绍python文件运行, 自动调用 <code>release_tengine()</code> ,对tengine库进行资源释放操作

8.5.3 __version__

使用	<code>tg.__version__</code>
介绍	获取Tengine库的版本信息
返回值	版本的字符串

8.5.4 requestTengineVersion

使用	<code>tg.requestTengineVersion</code>
介绍	初始化Tengine后, 通过该接口获取对版本的兼容情况

8.5.5 setDefaultDevice

使用	<code>tg.setDefaultDevice(dev_name)</code>	
介绍	设置默认运行设备	
参数	Dev_name	设备名称

8.5.6 __errno__

使用	<code>tg.__errno__</code>
介绍	获取错误码

8.5.7 log_lever

使用	<code>tg.log_lever = tg.LOG_EMERG</code>
介绍	获取错误码
参数	<code>tg.LOG_EMERG</code> 、 <code>tg.LOG_ALERT</code> and etc

8.5.8 log_print

使用	<code>tg.log_print = func</code>	
介绍	设置log输出函数	
参数	Func	输出方法: <code>def func(arg):</code> <code>print(arg)</code>

8.5.9 plugin.add

使用	<code>tg.plugin.add(name, fname, init_func)</code>	
介绍	加载tengine插件	
参数	Name	插件的名称
	Fname	文件的名称
	Init_func	初始化函数名

8.5.10 plugin.remove

使用	<code>tg.plugin.remove(name, del_func)</code>	
介绍	移除tengine插件	
参数	Name	插件的名称
	Del_name	Release的函数名

8.5.11 plugin.__len__

使用	<code>len(tg.plugin)</code>
介绍	获取tengine的插件数量

8.5.12 plugin.__getitem__

使用	<code>tg.plugin[idx]</code>	
介绍	获取tengine的插件名称	
参数	Idx	插件编号 (≥ 0)

8.6 graph 的相关操作

8.6.1 __init__

使用	<code>graph = tg.Graph(context,model_format,file_name,*kargs)</code>	
介绍	创建graph	
参数	Context	Context的对象
	Model_format	采用模型的类型, 比如 "tengine" , "mxnet" , "caffe" 等
	File_name	文件名
	...	其他参数

8.6.2 __del__

使用	<code>del graph</code>
介绍	销毁graph

8.6.3 setLayout

使用	<code>graph.setLayout(tg.TENGINE_LAYOUT_NCHW)</code>
介绍	设置graph的layout
参数	Graph的layout类型: tg.TENGINE_LAYOUT_NCHW、tg.TENGINE_LAYOUT_NHWC

8.6.4 setNode

使用	<code>graph.setNode(input_nodes=[],output_nodes=[])</code>	
介绍	设置graph的输入、输出节点	
参数	Input_nodes	输入节点数组
	Output_nodes	输出节点数组

8.6.5 __add__

使用	<code>graph=graph1+graph2</code>
介绍	将多个graph合并为一个graph

8.6.6 getInputNodeNumber

使用	<code>graph.getInputNodeNumber()</code>
介绍	获取graph的输入节点 (node) 数目
参数	无
返回值	>0: 节点数目; -1: 错误

8.6.7 getInputNodeByIdx

使用	<code>node = graph.getInputNodebyIdx(idx)</code>
----	--

介绍	通过ID获取graph的输入节点 (node)	
参数	Idx	Node下标 (≥ 0)
返回值	Node对象	

8.6.8 getOutputNodeNumber

使用	<code>graph.getOutputNodeNumber()</code>	
介绍	获取graph的输出节点 (node) 个数	
参数	无	
返回值	>0 :node编号; -1 : 失败	

8.6.9 getOutputNodeByIdx

使用	<code>node = graph.getOutputNodeByIdx(idx)</code>	
介绍	通过ID获取graph的输出节点 (node)	
参数	Idx	Node下标 (≥ 0)
返回值	Node对象	

8.6.10 getOutputTensor

使用	<code>tensor=graph.getOutputTensor(node_idx,idx)</code>	
介绍	按ID获取graph的输出张量 (tensor)	
参数	Node_idx	Node下标 (≥ 0)
	Idx	Tensor下标 (≥ 0)
返回值	Tensor对象	

8.6.11 getInputTensor

使用	<code>tensor=graph.getInputTensor(node_idx,idx)</code>	
----	--	--

介绍	按ID获取graph的输入张量	
参数	Node_idx	Node下标(>=0)
	idx	Tensor下标 (>=0)
返回值	Tensor对象	

8.6.12 getNodeNumber

使用	<code>graph.getNodeNumber()</code>	
介绍	获取graph的node数目	
参数	无	
返回值	>=0: node的数目; -1: 失败	

8.6.13 getNodeByIdx

使用	<code>node = graph.getNodeByIdx(idx)</code> 或 <code>node = graph[idx]</code>	
介绍	通过下标获取graph的node节点	
参数	Idx	Node的下标 (>=0)
返回值	Node对象	

8.6.14 getNodeByName

使用	<code>node = graph.getNodeByName(name)</code>	
介绍	获取graph的node	
参数	Name	Node名称
返回	Node对象	

8.6.15 setDevice

使用	<code>node = graph.setDevice (dev_name)</code>	
介绍	设置node运行设备	
参数	Dev_name	Device名称

8.6.16 doPerfStat

使用	<code>graph.doPerfStat(tg. GRAPH_PERF_STAT_DISABLE)</code>	
介绍	启用或禁用性能统计	
参数	Action	Tg. GRAPH_PERF_STAT_DISABLE etc ...

8.6.17 getPerfStat

使用	<code>info = graph.getPerfStat(size)</code>	
介绍	获取性能统计信息	
参数	Size	数据的大小

8.6.18 dump

使用	<code>graph.dump()</code>	
介绍	dump图的结构	

8.6.19 setAttr

使用	<code>graph.setAttr(attr_name,obj)</code>	
----	---	--

介绍	设置graph的参数	
参数	Attr_name	设置参数名称
	Obj	需要设置的参数（整型、浮点型、字符型的数据或列表）

8.6.20 getAttr

使用	<i>graph.getAttr(attr_name)</i>	
介绍	获取graph的参数	
参数	attr_name	参数名称
返回值	Buf列表	

8.6.21 setGdMethod

使用	<i>graph.setGdMethod(method,...)</i>	
介绍	设置图的梯度下降方法	
参数	Method	梯度下降函数
	...	可变参数

8.6.22 preRun

使用	<i>graph.preRun()</i>
介绍	准备运行graph，并准备资源

8.6.23 run

使用	<i>graph.run(block=1)</i>
介绍	运行graph，执行推理，可以反复多次调用

参数	Block	1: 阻塞; 0: 非阻塞
----	-------	---------------

8.6.24 wait

使用	<code>graph.wait(try=1)</code>	
介绍	等待graph运行的结果	
参数	try	1:检查状态, 然后返回 0: 立即返回

8.6.25 postRun

使用	<code>graph.postRun()</code>	
介绍	停止运行graph并释放graph占据资源	

8.6.26 __status__

使用	<code>graph.__status__</code>	
介绍	获取graph的执行状态	
返回值	Graph的运行状态	

8.6.27 setEvent

使用	<code>graph.setEvent(event, cb_func, cb_arg)</code>	
介绍	设置graph的事件回调函数	
参数	Event	事件类型
	Cb_func	回调函数, 类似于 def func(graph,value,arg): ... return 1

	Cb_arg	回调参数
--	--------	------

8.7 Node

8.7.1 __init__

使用	<code>node = tg.Node(graph,node_name,op_name)</code>	
介绍	创建graph的node	
参数	Graph	Graph对象
	Node_name	Node的名称
	Op_name	操作名称

8.7.2 __name__

使用	<code>node.__name__</code>
介绍	获取node的名称

8.7.3 __op__

使用	<code>node.__op__</code>
介绍	获取node使用的操作名称

8.7.4 __del__

使用	<code>del node</code>
介绍	释放graph的node

8.7.5 getInputTensorByIdx

使用	<code>tensor=node.getInputTensorByIdx(idx)</code>
介绍	获取node的输入张量 (tensor)

参数	Idx	Input的下标
返回值	Tensor对象	

8.7.6 getOutputTensorByIdx

使用	<i>tensor=node.getOuputTensorByIdx(idx)</i>	
介绍	获取node的输出张量	
参数	Idx	Output的下标
返回值	Tensor对象	

8.7.7 setInputTensorByIdx

使用	<i>node.setInputTensorByIdx(idx, tensor)</i>	
介绍	设置node的输入张量	
参数	idx	Input的下标 (>=0)
	Tensor	Tensor 对象

8.7.8 setOutputTensorByIdx

使用	<i>node.setOutputTensorByIdx(idx, tensor)</i>	
参数	idx	Input的下标 (>=0)
	Tensor	Tensor 对象

8.7.9 getOutputNumber

使用	<i>node.getOutputNumber()</i>	
介绍	获取tensor的输出张量的个数	
参数	无	
返回值	>0:输出张量个数; -1:失败	

8.7.10 getInputNumber

使用	<code>node.getInputNumber()</code>
介绍	获取tensor的输入张量个数
参数	无
返回值	>0:输入张量个数; -1: 失败

8.7.11 addAttr

使用	<code>node.addAttr(attr_name, attr)</code>	
介绍	添加node属性	
参数	attr_name	属性名称
	attr	属性数值（整型、浮点型或字符串或list）

8.7.12 getAttr

使用	<code>node.getAttr(attr)</code>	
介绍	获取node的属性值	
参数	attr	属性名称
返回值	属性值，（整型、浮点型、字符串型或者list）	

8.7.13 setDevice

使用	<code>node.setDevice(dev_name)</code>	
介绍	获取node的属性值	
参数	Dev_name	设备名称

8.7.14 getDevice

使用	<code>dev = node.getDevice()</code>
介绍	获取node的属性值
参数	无
返回值	设备的名称

8.7.15 dump

使用	<code>node.dump()</code>
介绍	设置节点的dump功能

8.7.16 getDumpBuf

使用	<code>buf = node.getDumpBuf(size)</code>	
介绍	获取节点的dump数据缓冲区	
参数	size	数组大小
返回值	返回节点的缓冲区buf	

8.7.17 setKernel

使用	<code>node.setKernel(dev_name, kernel_ops)</code>	
介绍	设置自定义kernel	
参数	Dev_name	设备名称
	Kernel_ops	Custom_kernel_ops对象

8.7.18 removeKernel

使用	<code>node.removeKernel(dev_name)</code>
介绍	设置自定义kernel

参数	Dev_name	设备名称
----	----------	------

8.8 Tensor

8.8.1 __init__

使用	<pre>tensor = tg.Tensor(graph, tensor_name, data_type)</pre>	
介绍	创建graph的tensor	
参数	Graph	Graph对象
	Tensor_name	Tensor的名称
	Data_type	数据类型

8.8.2 __name__

使用	<pre>tensor.__name__</pre>
介绍	获取tensor的名称

8.8.3 __del__

使用	<pre>del tensor</pre>
介绍	释放graph的tensor

8.8.4 shape

使用	<pre>dims = tensor.shape</pre>
介绍	获取tensor的shape

8.8.5 设置 shape

使用	<code>tensor.shape = [1,3,12,12]</code>
介绍	设置tensor的shape

8.8.6 buffer 大小

使用	<code>length = len(tensor)</code>
介绍	获取tensor的buffer大小

8.8.7 getbuffer

使用	<code>data = tensor.buf (或者 data = tensor.getbuffer(type=float))</code>
介绍	获取tensor的buffer

8.8.8 buffer

使用	<code>tensor.buf = datalist</code>	
介绍	设置tensor的buffer	
	Datalist	可以是整型、浮点型或字符串，列表等格式的数据

8.8.9 getData

使用	<code>data = tensor.getData()</code>
介绍	设置tensor的数据

8.8.10 setData

使用	<code>tensor.setData(data)</code>	
介绍	设置tensor的数据	
参数	Data	可以是整型、浮点型、字符型的数据或列表

8.8.11 dtype

使用	<code>type = tensor.dtype</code>	
介绍	获取tensor的数据类型	
返回值	返回一个DType对象	

8.8.12 dtype 设置

使用	<code>tensor.dtype = tg. TENGINE_DT_FP32</code>	
介绍	设置tensor的数据类型	

8.8.13 setQuantParam

使用	<code>tensor.setQuantParam(scale,zero_point,number)</code>	
介绍	设置tensor的量化参数	
参数	Scale	Float类型的数组
	Zero_point	起点指针
	number	数据大小

8.8.14 getQuantParam

使用	<code>Scale,zero_point = tensor.getQuantParam(number)</code>	
----	--	--

介绍	获取tensor的量化参数
参数	数据大小
返回值	返回scale数组和zero_point的数组

8.9 Device

8.9.1 __init__

使用	<code>device = tg.Device(driver_name, dev_name)</code>	
介绍	创建设备	
参数	Driver_name	驱动名称
	Dev_name	设备名称

8.9.2 __del__

使用	<code>del device</code>
介绍	销毁设备

8.9.3 policy

使用	设置: <code>device.policy = tg.DEFAULT_POLICY</code> 获取: <code>policy = device.policy</code>
介绍	设置或获取设备策略

8.9.4 setAttr

使用	<code>device.setAttr(item, obj)</code>	
介绍	设置设备参数	
参数	Item	参数名称

	Obj	设置参数（整型、浮点型、字符型数据或列表）
--	-----	-----------------------

8.9.5 getAttr

使用	<i>attr = device.getAttr(item, size)</i>	
介绍	获取设备参数	
参数	Item	参数名称
	Size	参数返回值大小
返回值	参数buf	

8.10 context

8.10.1 __init__

使用	<i>context = tg.Context(name, empty)</i>	
介绍	创建context	
参数	Name	Context名称
	empty	1:没有为该设备分配可用的设备; 0: 所有声明的设备都将添加到context中

8.10.2 __del__

使用	del context
介绍	销毁context

8.10.3 getDevNumber

使用	<code>context.getDevNumber()</code>
介绍	获取context设备编号
参数	无
返回值	Context使用的设备编号

8.10.4 getDevByIdx

使用	<code>name = context.getDevByIdx(idx)</code>
介绍	获取设备名称
参数	Idx索引
返回值	设备名称

8.10.5 addDev

使用	<code>context.addDev(dev_name)</code>
介绍	将设备添加到context中
参数	Dev_name 设备名称

8.10.6 rmDev

使用	<code>context.rmDev(dev_name)</code>
介绍	将设备移出context
参数	Dev_name 设备名称

8.10.7 setAttr

使用	<code>context.setAttr(attr,obj)</code>
介绍	设置context参数

参数	Attr	参数名称
	Obj	待设置参数

8.10.8 getAttr

使用	<i>buf = context.getAttr(attr, size)</i>	
介绍	获取context属性	
参数	Attr	参数名称
	size	参数所占的值得大小
返回值	返回attr的buffer	

8.11 测试

8.11.1 运行 SqueezeNet 测试程序

- 1) 修改测试文件中用到的模型及图片路径为用户自己的路径
- 2) 执行指令：

```
python bench_sqz.py
```

输出信息：

```
REPEAT COUNT= 100
Repeat [100] time 0.0262046790123 s per RUN. used 2.62046790123 s
----->
(0.27631011605262756, ' - "', 'n02123045 tabby, tabby cat\n')
(0.2672532796859741, ' - "', 'n02123159 tiger cat\n')
(0.17660656571388245, ' - "', 'n02119789 kit fox, Vulpes macrotis\n')
(0.08265203982591629, ' - "', 'n02124075 Egyptian cat\n')
(0.07774369418621063, ' - "', 'n02085620 Chihuahua\n')
----->
```

8.11.2 运行 MobileNet 测试程序

- 1) 修改测试文件中用到的模型及图片路径为用户当前路径
- 2) 执行指令：

```
python bench_mobilenet.py
```

输出信息：

```
run-time library version: 1.3.2-github
REPEAT COUNT= 100
Repeat [100] time 0.0655942893028 s per RUN. used 6.55942893028 s
8.59759235382 - n02123159 tiger cat

7.95499801636 - n02119022 red fox, Vulpes vulpes

7.86789274216 - n02119789 kit fox, Vulpes macrotis

7.42741203308 - n02113023 Pembroke, Pembroke Welsh corgi

6.36464977264 - n02123045 tabby, tabby cat

ALL TEST DONE
```

9 编译 Tengine 源码

Tengine 源码包括开源版本和商业版本，理论上商业版本的 tengine 支持的算子更多。以下仅介绍 Tengine 源码商业版本的编译方式。开源版本请参考 [Tengine Github](#)。

9.1 在 X86(ubuntu)本地编译

一般情况下，在服务器对算法模型进行评测时才会用到这种编译方式。这种情况下，商业版本的 hclcpu 库需要单独提供。请联系我们获取。

在开始编译 Tengine 之前，你需要确认你已经安装了 git, g++, make 和一些其他工具。如果没有，你可以通过如下指令安装：

```
sudo apt install make g++
```

如果你已经安装了这些工具，你可以跳过该步骤。

9.1.1 安装依赖的库文件

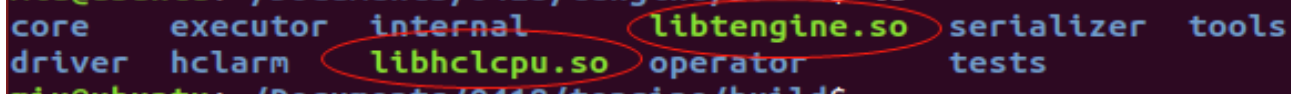
```
sudo apt install libopenblas-dev
```

9.1.2 编译

进入 tengine 目录，然后使用 bash 脚本编译 tengine 项目。

```
cd ~/tengine
bash linux_build.sh defaultconfig/x86_linux_native.config
```

如果在编译完成之后 build 目录下存在 libtengine.so 和 libhclcpu.so 文件，说明编译过程成功完成。



```
core      executor internal  libtengine.so  serializer  tools
driver    hclarm    libhclcpu.so  operator       tests
```

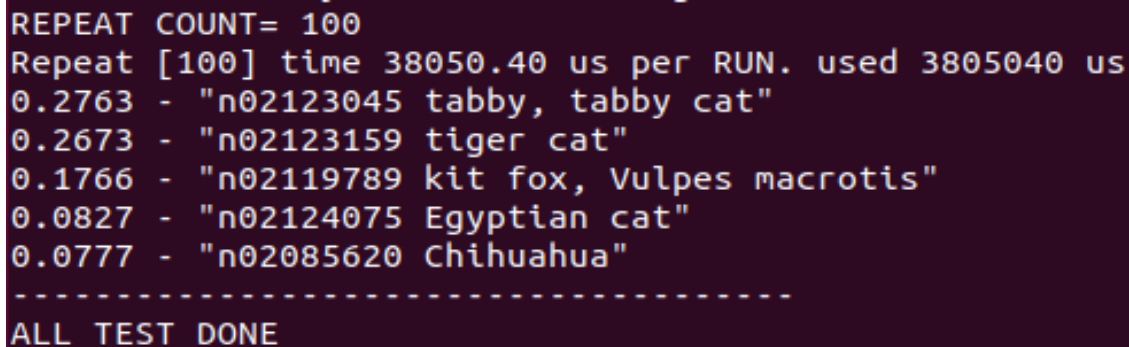
9.1.3 测试

编译完成之后，相关目录下有测试程序：SqueezeNet。你可以使用如下指令执行相关测试：

1) 运行 bench_sqz 测试程序

```
cd ~/tengine
./build/benchmark/bin/bench_sqz
```

输出信息：



```
REPEAT COUNT= 100
Repeat [100] time 38050.40 us per RUN. used 3805040 us
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox, Vulpes macrotis"
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"
-----
ALL TEST DONE
```

9.2 在 ARM(Linux)中编译源代码

目前 Tengine 商业版本支持 ARM-v7, ARM-v8 , 更多信息可以参考《Tengine 产品规格书》。

该章节将详细描述在 arm 本地编译源代码的步骤。

9.2.1 获取源代码

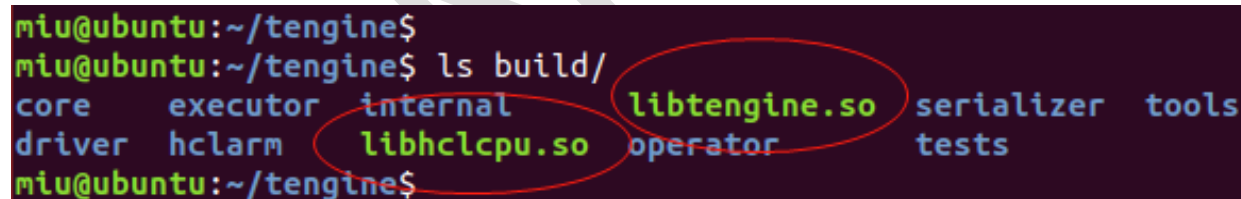
请在版本发布包中获取源代码。

9.2.2 编译 Armv7

进入 tengine 路径然后使用 bash 脚本 进行编译。

```
cd ~/tengine
bash linux_build.sh defaultconfig/arm32_linux_native.config
```

等待一段时间 , 如果在编译完成之后 build 目录下存在 libtengine.so 文件, 说明编译过程成功完成。



```
miu@ubuntu:~/tengine$
miu@ubuntu:~/tengine$ ls build/
core      executor  internal  libtengine.so  serializer  tools
driver    hclarm    libhclcpu.so  operator      tests
```

9.2.3 编译 Armv8

进入 tengine 路径然后使用 bash 脚本 进行编译。

```
cd ~/tengine
bash linux_build.sh defaultconfig/arm64_linux_native.config
```

等待一段时间 , 如果在编译完成之后 build 目录下存在 libtengine.so 文件, 说明编译过程成功完成。

```
miu@ubuntu:~/tengine$  
miu@ubuntu:~/tengine$ ls build/  
core      executor  internal  libtengine.so  serializer  tools  
driver    hclarm    libhclcpu.so  operator      tests
```

9.2.4 测试

Tengine 在 test 路径下提供了一些样例程序，通过运行样例程序与得到的结果，你可以判断你的 tengine 是否成功编译。

有两个测试示例来显示测试用例：

- 运行 SqueezeNet

```
cd ~/tengine  
./build/benchmark/bin/bench_sqz
```

输出信息：

```
REPEAT COUNT= 100  
Repeat [100] time 38050.40 us per RUN. used 3805040 us  
0.2763 - "n02123045 tabby, tabby cat"  
0.2673 - "n02123159 tiger cat"  
0.1766 - "n02119789 kit fox, Vulpes macrotis"  
0.0827 - "n02124075 Egyptian cat"  
0.0777 - "n02085620 Chihuahua"  
-----  
ALL TEST DONE
```

9.3 在 X86(Ubuntu)对 ARM Linux 交叉编译

Tengine 目前支持对 ARM64 与 ARM32 在 X86 Linux(Ubuntu)进行交叉编译，你可以按照如下步骤进行编译。

9.3.1 获取源代码

请在版本发布包中获取源代码。

9.3.2 安装 rootfs

进入 tengine 目录下的 systemroot。

```
cd ~/tengine/sysroot/
sudo apt install make
sudo apt install multistrap
```

Arm64:

```
make ubuntu
```

Arm32:

```
make ubuntu32
```

9.3.3 安装交叉编译链

Arm64:

```
sudo apt install g++-aarch64-linux-gnu
```

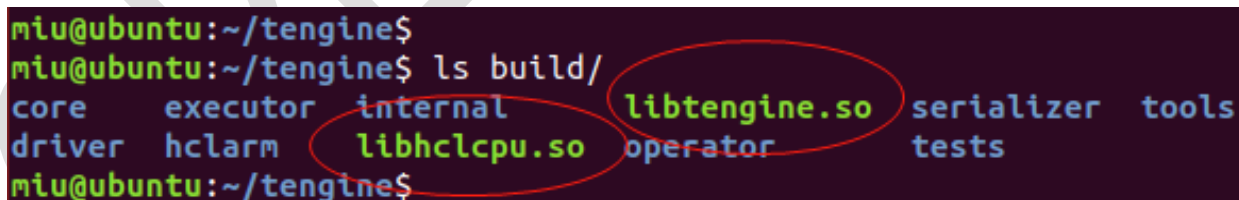
Arm32:

```
sudo apt install g++-arm-linux-gnueabi
```

9.3.4 编译 Arm64

```
cd ~/tengine/
bash linux_build.sh defaultconfig/arm64_linux_cross.config
```

等待一段时间，如果在编译完成之后 build 目录下存在 libtengine.so 文件，说明编译过程成功完成。



```
miu@ubuntu:~/tengine$
miu@ubuntu:~/tengine$ ls build/
core      executor  internal  libtengine.so  serializer  tools
driver    hclarm    libhclcpu.so  operator        tests
```

9.3.5 编译 Arm32

```
bash linux_build.sh defaultconfig/arm32_linux_cross.config
```

等待一段时间，如果在编译完成之后 build 目录下存在 libtengine.so 文件，说明编译过程成功完成。


```
miu@ubuntu:~/tengine$
miu@ubuntu:~/tengine$ ls build/
core      executor  internal  libtengine.so  serializer  tools
driver    hclarm    libhclcpu.so  operator      tests
```

9.3.6 测试

无错误地编译完成后，通过指令打包该文件。

```
make install
```

将文件传输到目标板，用你的目标 IP 和路径替换下文的红色文字(加个 scp 教程链接)

```
scp -r ./install eaidk@192.168.88.52:~/test
scp -r ./models/ eaidk@192.168.88.52:~/test
scp -r ./tests/images/ eaidk@192.168.88.52:~/test
```

登录目标板，进入交叉编译文件的测试目录。

```
cd ~/test
export LD_LIBRARY_PATH=.
mv install/lib/lib*.so .
./install/benchmark/bench_sqz
```

提示：你也可以仅传送应用程序。把它们放在同一个目录下。

(说明提前) 如果你得到一个提示为 GLIBCXX 版本问题的错误，如下图显示：

```
quest@firefly:~/test$ ./bench_mobilenet
./bench_mobilenet: /usr/lib/aarch64-linux-gnu/libstdc++.so.6: version `GLIBCXX_3.4.22' not found (required by ./libtengine.so)
quest@firefly:~/test$
```

你可以通过一下指令解决该问题：

```
sudo apt-get install libstdc++6
```

9.4 在 X86 Linux 平台为 Android 交叉编译

9.4.1 Tengine 源代码

请在版本发布包中获取源代码。

9.4.2 下载 Android ndk

从链接下载以下文件 http://ftp.openailab.net.cn/Tengine_android_build/

```
android-ndk-r16-linux-x86_64.zip
```

注意： 另一个备份链接：

[Tengine Android build](https://pan.baidu.com/s/1RPHK_ji0LIL3ztjUa893Yg#list/path=%2F) (链接：https://pan.baidu.com/s/1RPHK_ji0LIL3ztjUa893Yg#list/path=%2F)密

钥：*ka6a*:

9.4.3 解压

```
unzip android-ndk-r16-linux-x86_64.zip
```

9.4.4 设置依赖项库文件路径

我们需要更新依赖文件路径 NDK PATH 和 ARCH_TYPE。

```
vi ~/tengine/example_config/arm_android_cross.config
```

- 如果你想要编译 android for armv7:

设置 ARCH_TYPE:**Armv7**

- 如果你想要编译 android for armv8:

设置 ARCH_TYPE:**Armv8**

并且改变如下路径 ANDROID_NDK 到之前存储的位置，如下所示：

```
ANDROID_NDK: /home/usr/android-ndk-r16b
```

以下是一个针对 ARMv8 的配置文件样例：

```
ANDROID_NDK: /home/bhu/workshop/android/android-ndk-r16b
ARCH_TYPE:Armv8
```

9.4.5 编译 tengine

```
cd ~/tengine
./android_build.sh example_config/arm_android_cross.config
```

9.4.6 在电脑上安装 adb, adb_driver

请按照你的安卓设备的相关手册来做。

9.4.7 在安卓上测试 Tengine 的 Demo

9.4.7.1 准备文件

```
cd ~/tengine
./android_pack.sh example_config/arm_android_cross.config
cp -rf ./models ./android_pack
cp -rf ./install/benchmark ./android_pack
```

9.4.7.2 将包传输到 Android 设备

将以下文件放置到 ~/tengine/android_pack

```
adb push ./android_pack /data/local/tmp
```

9.4.7.3 运行

```
adb root
adb shell
cd /data/local/tmp/
chmod u+x Classify
export LD_LIBRARY_PATH=.
./benchmark/bench_sqz
```

输出信息:

```
REPEAT COUNT= 100
Repeat [100] time 38050.40 us per RUN. used 3805040 us
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox, Vulpes macrotis"
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"
-----
ALL TEST DONE
```

9.5 编译样例

9.5.1.1 设置 *TENGINE_DIR*.

```
cd ~/tengine/examples
```

- 如果你想要编译 android for armv7:

```
vi android_build_armv8.sh
```

请在 example/android_build_armv7.sh 中添加正确的 blas 路径，确保安装目录位于 *TENGINE_DIR* 中。

如下为文件截图，红色文字应该替换为实际的目录。

```
#!/bin/bash

export ANDROID_NDK=/home/usr/android-ndk-r16b

cmake -DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.cmake \
      -DANDROID_ABI="arm64-v8a" \
      -DANDROID_PLATFORM=android-21 \
      -DANDROID_STL=c++_shared \
      -DTENGINE_DIR=/home/usr/tengine \
      ..
```

- 如果你想要使用 OpenBlas 运行 Tengine:

请在 example/android_build_armv7.sh 或 example/android_build_armv8.sh 等，中添加正确的 blas 路径: `-DBLAS_DIR=/home/usr/openblas020_android`

9.5.1.2 编译 demo 样例

```
cd ~/tengine/examples
mkdir build
cd build
```

- 如果你想要编译 android for armv7:

```
../android_build_armv7.sh
```

- 如果你想要编译 android for armv8:

```
../android_build_armv8.sh  
make -j4
```

9.6 编译 tengine-module

9.6.1 安装依赖的库文件

```
sudo apt install protobuf  
sudo apt install libopencv-dev
```

注意：tengine 本身是不依赖于 opencv 和 protobuf 库文件，但是 tengine-module 依赖所以编译前需要安装库文件。

9.6.2 编辑 configuration 文件

使用编辑器打开 example_config/x86_build_tools.config 文件(以 vim 为例子)

```
vim example_config/x86_build_tools.config
```

- 如果你想要编译 serializer:

请设置 BUILD_SERIALIZER=y 以及正确的库路径

```
#The following option is used for setting compile flag  
CONFIG_OPT_CFLAGS=-O2  
  
#The following option is used for building serializer or not.  
#option value [y/n]  
BUILD_SERIALIZER=y  
  
#The following two options are used for setting protobuf only if turning on the BUILD_SERIALIZER option  
PROTOBUF_LIB_PATH=/protobuf/path/include  
PROTOBUF_INCLUDE_PATH=/protobuf/path/lib
```

- 如果你想要编译 ACL:

请设置 BUILD_ACL=y 以及正确的库路径

```
#The following option is used for building acl or not. [y/n]
#option value [y/n]
BUILD_ACL=n

#The following option are used for setting acl only if turning on the BUILD_ACL option
ACL_ROOT=/acl/path

#The following option is used for building conver tools only support linux_x86
#option value [y/n]
BUILD_TOOLS=y
```

- 如果你想要生成 tengine 相关工具:

请设置 BUILD_TOOLS=y。注意这个选项只对 x86 生效

```
#The following option are used for setting acl only if turning on the BUILD_ACL option
ACL_ROOT=/acl/path

#The following option is used for building conver tools only support linux_x86
#option value [y/n]
BUILD_TOOLS=y

#The following option is used for building with open blas
#option value [y/n]
```

9.6.3 编译

进入 tengine 路径然后使用 bash 脚本 进行编译。

```
cd ~/tengine
bash linux_build.sh example_config/x86_build_tools.config
```

等待一段时间，如果在编译完成之后 install 目录下存在 convert_tools 目录如果开启了 BUILD_TOOLS 选项；lib 目录下存在 libcaffe-serializer.so、libdarknet-serializer.so、libmxnet-

serializer.so、libtensorflow-serializer.so、libonnx-serializer.so、libtf-lite-serializer.so 如果开启了选项 BUILD_SERIALIZER,如果开始了 BUILD_ACL 选项会存在 libacl_openccl.so。

9.7 其它编译选项说明

9.7.1 配置 open blas 库

打开任意配置文件，这里以 x86_build_tools.config 为例。

```
vim default_config/x86_build_tools.config
```

配置项如下：

```
#BUILD_TOOLS=y

#The following option is used for building with open blas
OPEN_BLAS=y

#The following two options are used for setting openblas only if turning on the OPEN_BLAS option
#OPENBLAS_LIB_PATH=/openblas/path/include
#OPENBLAS_INCLUDE_PATH=/openblas/path/lib
```

9.7.2 所有选项说明

linux 交叉编译工具链指定

```
#The following option is used for setting cross compiler path
#EMBEDDED_CROSS_PATH=/opt/toolchain/path
```

```
# cross compile for ARM64
# CROSS_COMPILE=aarch64-linux-gnu-
# cross compile for ARM32
# CROSS_COMPILE=arm-linux-gnueabihf-
```

设置 android ndk 路径 以及 api-level

```
#The following option is used for android ndk path
ANDROID_NDK=/root/work/git/android_libs/android-ndk-r16
```

```
#The following option is used for android api level
```

```
API_LEVEL=22
```

```
# 设置目标 arch
#The following option is used for system architecture type.
#option value [x86/Arm32/Arm64]
ARCH_TYPE=x86
```

```
# 启用编译选项
CONFIG_OPT_CFLAGS = -O2
```

```
# 编译 tengine-module 中的 serializer
#The following option is used for building serializer or not.
#option value [y/n]
BUILD_SERIALIZER=n

#The following two options are used for setting protobuf only if turning on the
BUILD_SERIALIZER option
PROTOBUF_LIB_PATH=/protobuf/path/include
PROTOBUF_INCLUDE_PATH=/protobuf/path/lib
```

```
#编译 tengine-module 中的 acl
#The following option is used for building acl or not. [y/n]
#option value [y/n]
BUILD_ACL=n

#The following option are used for setting acl only if turning on the BUILD_ACL option
ACL_ROOT=/acl/path
```

```
# 是否编译 tengine-module 中的工具
#The following option is used for building conver tools only support linux_x86
#option value [y/n]
#BUILD_TOOLS=y
```


10 Tengine C API 介绍

10.1 数据类型和数据结构

Tengine C API 相关的数据类型和数据结构定义如下：

10.1.1 MAX_SHAPE_DIM_NUM

【说明】

定义 shape 最大维度。

【定义】

#define	MAX_SHAPE_DIM_NUM	4
----------------	-------------------	---

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.2 Tensor 的数据类型

【说明】

定义 Tensor 的数据类型。

【定义】

#define	TENGINE_DT_FP32	0
#define	TENGINE_DT_FP16	1
#define	TENGINE_DT_INT8	2
#define	TENGINE_DT_UINT8	3
#define	TENGINE_DT_INT32	4

#define	TENGINE_DT_INT16	5
----------------	------------------	---

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.3 数据布局格式

【说明】

定义数据的布局格式。

【定义】

#define	TENGINE_LAYOUT_NCHW	0
#define	TENGINE_LAYOUT_NHWC	1

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.4 Tensor 的类型

【说明】

定义 Tensor 类型。

【定义】

#define	TENSOR_TYPE_UNKNOWN	0
----------------	---------------------	---

#define	TENSOR_TYPE_VAR	1
#define	TENSOR_TYPE_CONST	2
#define	TENSOR_TYPE_INPUT	3
#define	TENSOR_TYPE_DEP	4

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.5 Node 转储操作类型

【说明】

定义节点转储类型。

【定义】

#define	NODE_DUMP_ACTION_DISABLE	0
#define	NODE_DUMP_ACTION_ENABLE	1
#define	NODE_DUMP_ACTION_START	2
#define	NODE_DUMP_ACTION_STOP	3
#define	NODE_DUMP_ACTION_GET	4

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.6 Graph 性能统计动作类型

【说明】

定义 Graph 性能统计动作类型。

【定义】

#define	GRAPH_PERF_STAT_DISABLE	0
#define	GRAPH_PERF_STAT_ENABLE	1
#define	GRAPH_PERF_STAT_STOP	2
#define	GRAPH_PERF_STAT_START	3
#define	GRAPH_PERF_STAT_RESET	4
#define	GRAPH_PERF_STAT_GET	5

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

- do_graph_perf_stat
- get_graph_perf_stat

10.1.7 Log 级别

【说明】

定义 Log 级别。

【定义】

```
enum log_level
{
    LOG_EMERG,
    LOG_ALERT,
    LOG_CRIT,
    LOG_ERR,
    LOG_WARNING,
    LOG_NOTICE,
    LOG_INFO,
    LOG_DEBUG
}
```

```
};
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

- set_log_level

10.1.8 Graph 执行事件类型

【说明】

定义 Graph 执行事件类型。

【定义】

```
enum graph_exec_event
{
    GRAPH_EXEC_START,
    GRAPH_EXEC_SUSPEND,
    GRAPH_EXEC_RESUME,
    GRAPH_EXEC_ABORT,
    GRAPH_EXEC_DONE
};
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.9 Graph 执行状态类型

【说明】

定义 Graph 执行状态类型。

【定义】

```
enum graph_exec_stat
{
    GRAPH_STAT_CREATED,
    GRAPH_STAT_READY,
    GRAPH_STAT_RUNNING,
    GRAPH_STAT_DONE,
    GRAPH_STAT_ERROR
};
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.10 设备运行策略

【说明】

定义设备运行策略。

【定义】

```
enum device_policy
{
    DEFAULT_POLICY,
    LATENCY_POLICY,
    LOW_POWER_POLICY
};
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.11 上下文

【说明】

定义上下文。

【定义】

```
typedef void * context_t;
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.12 Graph 句柄

【说明】

定义 Graph 句柄。

【定义】

```
typedef void * graph_t;
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.13 Tensor 句柄

【说明】

定义 Tensor 句柄。

【定义】

```
typedef void * tensor_t;
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.14 Node 句柄

【说明】

定义 Node 句柄。

【定义】

```
typedef void * node_t;
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.15 事件回调函数

【说明】

定义事件回调函数。

【定义】

```
typedef int (*event_handler_t)(graph_t, int, void* arg);
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.16 Log 输出函数

【说明】

定义 Log 输出函数。

【定义】

```
typedef void (*log_print_t)(const char*);
```

【成员】

无。

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.17 性能分析记录结构体

【说明】

定义性能分析记录结构体。

【定义】

```
struct perf_info
{
    const char* name;           /* node 名称 */
    const char* dev_name;       /* 设备名称 */
    uint32_t count;
    uint32_t min;
    uint32_t max;
    uint64_t total_time;        /* us 或 cycle, 取决于设备 */
    uint32_t base;              /* 1ms second time number */
};
```

【成员】

略。

【注意事项】

无。

【相关数据类型及接口】

- do_graph_perf_stat
- get_graph_perf_stat

10.1.18 用户自定义 Tensor

【说明】

定义用户自定义 Tensor 结构体。

【定义】

```
struct custom_kernel_tensor
{
    int dim[MAX_SHAPE_DIM_NUM]; /* 形状的数组 */
    int dim_num;                 /* valid entry number */
    int element_num;
```

```

    int element_size;           /* 由数据类型(data_type)决定 */
    int data_type;
    int dev_type;               /* 表示张量属于 CPU/GPU ... */
    int layout_type;           /* NCHW 类型或者 NHWC 类型 */

    /* quant info */
    int quant_type;             /* int8, int16 or int32 */
    float* scale;
    int* zero_point;
    int* quant_number;

    void* data;                 /* pointer to host memory (virtual address) */
    void* dev_mem;              /* refers to device memory block */
    void* mapped_mem;          /* the mapped dress for device memory block */
};

```

【成员】

成员名称	描述
int dim[MAX_SHAPE_DIM_NUM]	形状数组
int dim_num	形状数组中数据个数
int element_num	要素个数
int element_size	要素大小
int data_type	数据类型
int dev_type	设备类型, 表示张量属于 CPU/GPU ...
int layout_type	布局类型, NCHW / NHWC 类型
int quant_type	量化类型, int8, int16 或者 int32
float* scale	系数
int* zero_point	零点
int* quant_number	量化参数
void* data	数据指针
void* dev_mem	设备中的数据块指针
void* mapped_mem	映射地址

【注意事项】

无。

【相关数据类型及接口】

无。

10.1.19 用户自定义 Kernel

【说明】

定义用户自定义 Kernel 结构体。

【定义】

```
struct custom_kernel_ops
{
    const char* kernel_name;    /* kernel 名称 */
    const char* op;             /* 要实现的 op 的名称 */
    int force;                  /* if not set, when bind() failed,
                               try to use other kernel implementations*/
    void* kernel_param;         /* 用于内核实现的函数 */
    int kernel_param_size;

    /*!
     * @brief 根据输入形状生成输出形状
     *
     * 如果未实现，请将其设置为 NULL。
     *
     * @param [in] ops: 自定义 kernel 的指针。
     * @param [in] inputs[]: 指向输入张量形状的指针数组。
     *
     * 形状为 MAX_SHAPE_DIM_NUM 元素，
     * 值为 0 的元素表示形状的结尾。
     * @param [in] input_num: 输入张量的数目。
     * @param [out] outputs[]: 指向已分配内存的输出张量形状的指针数组。
     * @param [in] output_num: 输出张量的数目
     * @param [in] layout: 图形布局为 NHWC 或 NCHW
     *
     * @return 0: success, -1: fail.
     */
};
```

```

int (*infer_shape)(struct custom_kernel_ops* ops, const int* inputs[],
                    int input_num, int* outputs[],int output_num, int layout);

/*!
 * @brief Get the inplace input tensor index for an output tensor.
 *
 * @param [in] ops: 自定义内核的指针。
 * @param [in] output_idx: 自定义内核输出的索引。
 *
 * @return the inplace input tensor index for an output tensor.
 *         if the output tensor is not an inplace one, return -1.
 */
int (*inplace_info)(struct custom_kernel_ops* ops, int output_idx); // 可选项

/*!
 * @brief 检查内核是否可以处理输入和输出形状。
 *
 * @param [in] ops: 自定义 kernel 的指针。
 * @param [in] inputs[]: 用于输入的自定义 kernel 的 tensor
 * @param [in] input_num: input tensors 的数目
 * @param [in] outputs[]: 自定义 kernel 的 output tensor
 * @param [in] output_num: output tensors 的数目
 *
 * @return 0 if the input and output are supported
 *         otherwise, return -1.
 *
 * notes: If not implemented, set it NULL, which means always return 0.
 */
int (*bind)( struct custom_kernel_ops* ops,
              const struct custom_kernel_tensor* inputs[], int input_num,
              const struct custom_kernel_tensor* outputs[], int output_num);

/*!
 * @brief 预运行 graph.
 *
 *         dynamic_shape 意味着 input_num 为零时不是异常情况。
 *
 * @param [in] ops: 自定义 kernel 的指针。
 * @param [in] inputs[]: 自定义 kernel 的 input tensor
 * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
 * @param [in] outputs[]: 自定义 kernel 的 output tensor

```

```

* @param [in] output_num: 自定义 kernel 的 output tensor 的数目
* @param [in] dynamic_shape: It is not an abnormal case when input_num is
zero.? ?
*
* @return 0: success, -1: fail.
*/
int (*prerun)(struct custom_kernel_ops* ops,
               struct custom_kernel_tensor* inputs[], int input_num,
               struct custom_kernel_tensor* outputs[], int output_num,
               int dynamic_shape);

/*!
* @brief Reshape the graph.
*
* @param [in] ops: 自定义 kernel 的指针.
* @param [in] inputs[]: 自定义 kernel 的 input tensor
* @param [in] input_num: 自定义 kernel 的 input tensor 的数目
* @param [in] outputs[]: 自定义 kernel 的 output tensor
* @param [in] output_num: 自定义 kernel 的 output tensor 的数目
*
* @return 0: success, -1: fail.
*
* notes: 当输入形状更改时, 将被调用它。
*        调用 prerun()后, 需要回收并重新分配运行时
*        资源取决于输入形状
*/
int (*reshape)(struct custom_kernel_ops* ops,
                struct custom_kernel_tensor* inputs[], int input_num,
                struct custom_kernel_tensor* outputs[], int output_num);

/*!
* @brief 运行 graph.
*
* @param [in] ops: 自定义 kernel 的指针.
* @param [in] inputs[]: 自定义 kernel 的 input tensor
* @param [in] input_num: 自定义 kernel 的 input tensor 的数目
* @param [in] outputs[]: 自定义 kernel 的 output tensor
* @param [in] output_num: 自定义 kernel 的 output tensor 的数目
*
* @return 0: success, -1: fail.
*/

```

```

int (*run)(struct custom_kernel_ops* ops,
           struct custom_kernel_tensor* inputs[], int input_num,
           struct custom_kernel_tensor* outputs[], int output_num);

/*!
 * @brief 暂停 graph 并释放 graph 运行时使用的资源。
 *
 * @param [in] ops: 自定义 kernel 的指针。
 * @param [in] inputs[]: 自定义 kernel 的 input tensor
 * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
 * @param [in] outputs[]: 自定义 kernel 的 output tensor
 * @param [in] output_num: 自定义 kernel 的 output tensor 的数目
 *
 * @return 0: success, -1: fail.
 */
int (*postrun)(struct custom_kernel_ops* ops,
               struct custom_kernel_tensor* inputs[], int input_num,
               struct custom_kernel_tensor* outputs[], int output_num);

/*!
 * @brief 释放分配给此 ops 实现的资源。
 *
 * @param [in] ops: 自定义 kernel 的指针。
 *
 * @return None.
 */
void (*release)(struct custom_kernel_ops* ops);
};

```

【成员】

成员名称	描述
const char* kernel_name	Kernel 名称
const char* op	OP 名称
int force;	是否强制使用
void* kernel_param	参数指针
int kernel_param_size	参数大小

int (*infer_shape)(struct custom_kernel_ops* ops, const int* inputs[], int input_num, int* outputs[],int output_num, int layout)	形状推理函数
int (*inplace_info)(struct custom_kernel_ops* ops, int output_idx)	为一个输出 Tensor 获取其输入 Tensor 的 Index, 可选。
int (*bind)(struct custom_kernel_ops* ops, const struct custom_kernel_tensor* inputs[], int input_num, const struct custom_kernel_tensor* outputs[], int output_num);	绑定函数
int (*prerun)(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[], int input_num, struct custom_kernel_tensor* outputs[], int output_num, int dynamic_shape);	预运行函数
int (*reshape)(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[], int input_num, struct custom_kernel_tensor* outputs[], int output_num);	形状切换函数
int (*run)(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[], int input_num, struct custom_kernel_tensor* outputs[], int output_num);	运行函数
int (*postrun)(struct custom_kernel_ops* ops, struct custom_kernel_tensor* inputs[], int input_num, struct custom_kernel_tensor* outputs[], int output_num);	停止运行函数
void (*release)(struct custom_kernel_ops* ops);	释放函数

【注意事项】

无。

【相关数据类型及接口】

- set_custom_kernel

10.2 C API 介绍

Tengine 支持的主要 APIs 是：

- `init_tengine`：初始化 tengine.
- `release_tengine`：释放 tengine 资源
- `get_tengine_version`：获取 tengine 版本号
- `request_tengine_version`：检查 tengine 版本号是否支持
- `create_graph`：创建 graph
- `save_graph`：保存 graph 到文件
- `set_graph_layout`：设置 graph 的布局类型(layout)
- `set_graph_input_node`：设置 graph 的输入节点(node)
- `set_graph_output_node`：设置 graph 的输出节点(node)
- `merge_graph`：将多个 graph 合并为一个 graph
- `destroy_graph`：销毁 graph
- `get_graph_input_node_number`：获取 graph 的输入节点(node)的个数
- `get_graph_input_node`：通过 ID 获取 graph 的输入节点(node)
- `get_graph_output_node_number`：获取 graph 的输出节点(node)的个数
- `get_graph_output_node`：通过 ID 获取 graph 的输出节点(node)
- `get_graph_output_tensor`：通过 ID 获取 graph 的输出节点的张量(tensor)
- `get_graph_input_tensor`：通过 ID 获取 graph 的输入节点的张量(tensor)
- `create_graph_node`：创建 graph 的节点(node)
- `get_graph_node`：获取节点(node)句柄
- `get_node_name`：获取节点(node)的名称
- `get_node_op`：获取节点(node)的操作(operation)
- `release_graph_node`：释放 graph 的节点(node)
- `get_node_input_tensor`：通过 ID 获取节点(node)的输入张量(tensor)

- `get_node_output_tensor`: 通过 ID 获取节点(node)的输出张量(tensor)
- `set_node_input_tensor`: 设置节点(node)的输入张量(tensor)
- `set_node_output_tensor`: 设置节点(node)的输出张量(tensor)
- `get_node_output_number`: 获取节点(node)的输出张量(tensor)个数
- `get_node_input_number`: 获取节点(node)的输入张量(tensor)个数
- `add_node_attr`: 添加节点(node)参数(attribute)
- `get_node_attr_int`: 获取节点(node)的整型参数
- `get_node_attr_float`: 获取节点(node)的浮点型参数
- `get_node_attr_pointer`: 获取节点(node)的指针型参数
- `get_node_attr_generic`: 获取节点(node)的通用型参数
- `set_node_attr_int`: 设置节点(node)的整型参数
- `set_node_attr_float`: 设置节点(node)的浮点型参数
- `set_node_attr_pointer`: 设置节点(node)的指针型参数
- `set_node_attr_generic`: 设置节点(node)的通用型参数
- `set_custom_kernel`: 设置设置用户内核(kernel)
- `remove_custom_kernel`: 移除用户内核(kernel)
- `create_graph_tensor`: 创建 graph 的张量(tensor)
- `get_graph_tensor`: 通过名称获取 graph 的张量(tensor)
- `get_tensor_name`: 获取张量(tensor)的名称
- `release_graph_tensor`: 释放 graph 的张量(tensor)
- `get_tensor_shape`: 获取张量(tensor)的形状(shape)
- `set_tensor_shape`: 设置张量(tensor)的形状(shape)
- `get_tensor_buffer_size`: 获取张量(tensor)的缓存大小
- `get_tensor_buffer`: 获取张量(tensor)的缓存(buffer)
- `set_tensor_buffer`: 设置张量(tensor)的缓存(buffer)
- `get_tensor_data`: 获取张量(tensor)的数据(data)

- `set_tensor_data`: 设置张量(tensor)的数据(data)
- `get_tensor_data_type`: 获取张量(tensor)的数据类型
- `set_tensor_data_type`: 设置张量(tensor)的数据类型
- `set_tensor_quant_param`: 设置 tensor 的量化参数
- `get_tensor_quant_param`: 获取 tensor 的量化参数
- `set_graph_attr`: 设置 graph 的参数
- `get_graph_attr`: 获取 graph 的参数
- `set_graph_gd_method`: 设置 graph 的梯度下降方法
- `prerun_graph`: graph 的预运行
- `run_graph`: 运行 graph
- `wait_graph`: 等待 graph
- `postrun_graph`: 停止运行 graph 并释放 graph 占据的资源
- `get_graph_exec_status`: 获取 graph 的执行状态
- `set_graph_event_hook`: 设置 graph 的事件回调钩子
- `set_default_device`: 设置默认设备
- `set_graph_device`: 设置 graph 的执行设备
- `set_node_device`: 设置 node 的执行设备
- `get_node_device`: 获取 node 的执行设备
- `do_node_dump`: 使能节点的 dump 功能
- `get_node_dump_buffer`: 获取节点的 dump 数据
- `do_graph_perf_stat`: 启用或禁用性能统计
- `get_graph_perf_stat`: 获取性能统计信息
- `get_device_number`: 获取设备编号
- `get_device_name`: 按 index 获取设备名称
- `get_default_device`: 获取默认设备
- `create_device`: 创建设备

- `destroy_device`: 销毁设备
- `set_device_policy`: 设置设备策略
- `get_device_policy`: 获取设备策略
- `get_device_attr`: 获取设备参数
- `set_device_attr`: 设置设备参数
- `create_context`: 创建 context
- `destroy_context`: 销毁 context
- `get_context_device_number`: 获取 context 设备数量
- `get_context_device_name`: 获取 context 设备名称
- `add_context_device`: 将设备添加到 context 中
- `remove_context_device`: 将设备移除出 context
- `set_context_attr`: 设置 context 属性
- `get_context_attr`: 获取 context 属性
- `get_tengine_errno`: 获取错误数量
- `set_log_level`: 设置 log 级别
- `set_log_output`: 设置日志打印功能
- `dump_graph`: dump 图的结构
- `load_tengine_plugin`: 加载 Tengine 插件
- `unload_tengine_plugin`: 移除 Tengine 插件
- `get_tengine_plugin_number`: 获取 Tengine 插件数量
- `get_tengine_plugin_name`: 通过索引获取插件的名称

10.2.1 Tengine 创建与释放相关 API

10.2.1.1 `init_tengine`

名称	<code>init_tengine</code>
----	---------------------------

介绍	使用 tengine 库前，首先需要执行 init_tengine(), 对 tengine 库进行初始化	
参数	无	
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int init_tengine(void);	

10.2.1.2 release_tengine

名称	release_tengine	
介绍	执行 release_tengine(), 对 tengine 库进行资源释放操作	
参数	无	
返回值	无	
原型	void release_tengine(void);	

10.2.1.3 get_tengine_version

名称	get_tengine_version	
介绍	获取 Tengine 库的版本信息	
参数	无	
返回值	const char *	版本的字符串指针
原型	const char * get_tengine_version(void);	

10.2.1.4 request_tengine_version

名称	request_tengine_version	
介绍	初始化 Tengine 后，通过该接口获取对版本的兼容情况	
参数	const char * version	(I) 版本号字符串的指针
返回值	int	1：表示支持 0：表示不支持
原型	int request_tengine_version(const char * version);	

10.2.2 graph 相关操作

10.2.2.1 create_graph

名称	create_graph	
介绍	创建 graph	
参数	context_t context	(I) context 的句柄
	const char * model_format	(I) 采用模型的类型，比如“tengine”，“mxnet”，“caffe”，“onnx”，“tensorflow”等
	const char * file_name	(I) 文件名称
	...	(I) 其他参数
返回值	graph_t	Graph 句柄，如果为 NULL 表示失败
原型	graph_t create_graph(context_t context, const char * model_format, const char * file_name,...);	

10.2.2.2 save_graph

名称	save_graph	
介绍	保存 graph	
参数	graph_t graph	(I) graph 的句柄
	const char * model_format	(I) 采用模型的类型，比如“tengine”，“mxnet”，“caffe”，“onnx”，“tensorflow”等
	const char * file_name	(I) 文件名称
	...	(I) 其他参数
返回值	int	0：表示成功 -1：表示失败
原型	int save_graph(graph_t graph, const char * model_format, const char * file_name, ...)	

10.2.2.3 set_graph_layout

名称	set_graph_layout	
介绍	设置 graph 的 layout	
参数	graph_t graph	(I) graph 的句柄
	int layout_type	(I) Graph 的 layout 类型，比如： ENGINE_LAYOUT_NCHW ENGINE_LAYOUT_NHWC
返回值	int	0：表示成功 -1：表示失败

原型	<code>int set_graph_layout(graph_t graph, int layout_type);</code>
-----------	--

10.2.2.4 set_graph_input_node

名称	set_graph_input_node	
介绍	设置 graph 的 input node	
参数	graph_t graph	(I) graph 的句柄
	const char * input_nodes[]	(I) 输入节点数组的指针, char 类型
	int input_number	(I) 输入节点的数量
返回值	int	0 : 表示成功 -1 : 表示失败
原型	<code>int set_graph_input_node(graph_t graph, const char * input_nodes[], int input_number);</code>	

10.2.2.5 set_graph_output_node

名称	set_graph_output_node	
介绍	设置 graph 的 output node	
参数	graph_t graph	(I) graph 的句柄
	const char * output_nodes[]	(I) 输出节点数组的指针, char 类型
	int output_number	(I) 输出节点的数量

返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_graph_output_node(graph_t graph, const char * output_nodes[], int output_number);	

10.2.2.6 merge_graph

名称	merge_graph	
介绍	将多个 graph 合并为一个 graph	
参数	int graph_num	(I) graph 句柄的个数
	graph_t graph0	(I) graph 句柄 0
	graph_t graph1	(I) graph 句柄 1
	...	(I) 其他参数
返回值	graph_t	graph 句柄, 如果为 NULL 表示失败
原型	graph_t merge_graph(int graph_num , graph_t graph0, graph_t graph1, ...);	

10.2.2.7 destroy_graph

名称	destroy_graph	
介绍	销毁 graph	
参数	graph_t graph	(I) graph 句柄
返回值	int	0 : 表示成功 -1 : 表示失败

原型	<code>int destroy_graph(graph_t graph);</code>
----	--

10.2.2.8 get_graph_input_node_number

名称	<code>get_graph_input_node_number</code>	
介绍	获取 graph 的输入节点(node)数目	
参数	<code>graph_t graph</code>	(I) graph 句柄
返回值	<code>int</code>	>0 : node 个数 -1 : failure
原型	<code>int get_graph_input_node_number(graph_t graph);</code>	

10.2.2.9 get_graph_input_node

名称	<code>get_graph_input_node</code>	
介绍	通过 ID 获取 graph 的输入节点(node)	
参数	<code>graph_t graph</code>	(I) graph 句柄
	<code>int idx</code>	(I) Node 下标(>=0)
返回值	<code>node_t</code>	node 句柄, 如果为 NULL 表示失败
原型	<code>node_t get_graph_input_node(graph_t graph, int idx);</code>	

10.2.2.10 get_graph_output_node_number

名称	<code>get_graph_output_node_number</code>
介绍	获取 graph 的输出节点(node)个数

参数	graph_t graph	(I) graph 句柄
返回值	int	>0 : node 编号 -1 : 失败
原型	int get_graph_output_node_number(graph_t graph);	

10.2.2.11 get_graph_output_node

名称	get_graph_output_node	
介绍	通过 ID 获取 graph 的输出节点(node)	
参数	graph_t graph	(I) graph 句柄
	int idx	(I) Node 下标(>=0)
返回值	node_t	node 句柄, 如果为 NULL 表示失败
原型	node_t get_graph_output_node(graph_t graph, int idx);	

10.2.2.12 get_graph_output_tensor

名称	get_graph_output_tensor	
介绍	按 ID 获取 graph 的输出张量(tensor)	
参数	graph_t graph	(I) graph 句柄
	int output_node_idx	(I) Node 下标(>=0)
	int tensor_idx	(I) tensor 下标 (>=0)
返回值	tensor_t	tensor 句柄, 如果为 NULL 表示失败

原型	tensor_t get_graph_output_tensor(graph_t graph, int output_node_idx, int tensor_idx);
----	---

10.2.2.13 get_graph_input_tensor

名称	get_graph_input_tensor	
介绍	按 ID 获取 graph 的输入张量(tensor)	
参数	graph_t graph	(I) graph 句柄
	int input_node_idx	(I) Node 下标(>=0)
	int tensor_idx	(I) tensor 下标 (>=0)
返回值	tensor_t	tensor 句柄, 如果为 NULL 表示失败
原型	tensor_t get_graph_input_tensor(graph_t graph, int input_node_idx, int tensor_idx);	

10.2.2.14 get_graph_node_number

名称	get_graph_node_number	
介绍	获取 graph 的 node 数目	
参数	graph_t graph	(I) graph 句柄
返回值	int	>=0 : 表示成功 -1 : 表示失败
原型	int get_graph_node_number(graph_t graph);	

10.2.2.15 get_graph_node_by_idx

名称	get_graph_node_by_idx	
介绍	通过下标获取 graph 的 node	
参数	graph_t graph	(I) graph 句柄
	int node_idx	(I) node 的下标
返回值	node_t	node 句柄, 如果为 NULL 表示失败
原型	node_t get_graph_node_by_idx(graph_t graph, int node_idx);	

10.2.3 node 相关操作

10.2.3.1 create_graph_node

名称	create_graph_node	
介绍	创建 graph 的 node	
参数	graph_t graph	(I) graph 句柄
	const char * node_name	(I) node 的名称
	const char * op_name	(I) 操作名称
返回值	node_t	node 句柄, 如果为 NULL 表示失败
原型	node_t create_graph_node(graph_t graph, const char * node_name, const char * op_name);	

10.2.3.2 get_graph_node

名称	get_graph_node	
介绍	获取 graph 的 node 的句柄	
参数	graph_t graph	(I) graph 句柄
	const char * node_name	(I) node 的名称
返回值	node_t	node 句柄, 如果为 NULL 表示失败
原型	node_t get_graph_node(graph_t graph, const char * node_name);	

10.2.3.3 get_node_name

名称	get_node_name	
介绍	获取 node 的名称	
参数	node_t node	(I) node 句柄
返回值	const char *	node 句柄名称指针, 如果为 NULL 表示失败
原型	const char * get_node_name(node_t node);	

10.2.3.4 get_node_op

名称	get_node_op	
介绍	获取 node 使用的操作(op)	
参数	node_t node	(I) node 句柄

返回值	const char *	node 的 op 名称指针, 如果为 NULL 表示失败
原型	const char * get_node_op(node_t node);	

10.2.3.5 release_graph_node

名称	release_graph_node	
介绍	释放 graph 的 node	
参数	node_t node	(I) node 句柄
返回值	无	
原型	void release_graph_node(node_t node);	

10.2.3.6 get_node_input_tensor

名称	get_node_input_tensor	
介绍	获取 node 的输入张量(tensor)	
参数	node_t node	(I) node 句柄
	int input_idx	(I) Input 的下标 (>=0)
返回值	tensor_t	tensor 的句柄, 如果为 NULL 表示失败
原型	tensor_t get_node_input_tensor(node_t node, int input_idx);	

10.2.3.7 get_node_output_tensor

名称	get_node_output_tensor
-----------	------------------------

介绍	获取 node 的输出张量(tensor)	
参数	node_t node	(I) node 句柄
	int output_idx	(I) Output 的下标 (>=0)
返回值	tensor_t	tensor 的句柄, 如果为 NULL 表示失败
原型	tensor_t get_node_output_tensor(node_t node, int output_idx);	

10.2.3.8 set_node_input_tensor

名称	set_node_input_tensor	
介绍	设置 node 的输入张量(tensor)	
参数	node_t node	(I) node 句柄
	int input_idx	(I) Input 的下标 (>=0)
	tensor_t tensor	(I) tensor 的句柄
返回值	int	0 : 表示设置成功 -1 : 表示设置失败
原型	int set_node_input_tensor(node_t node, int input_idx, tensor_t tensor);	

10.2.3.9 set_node_output_tensor

名称	set_node_output_tensor	
介绍	设置 node 的输出张量(tensor)	
参数	node_t node	(I) node 句柄

	int output_idx	(I) Input 的下标 (≥ 0)
	tensor_t tensor	(I) tensor 的句柄
	int tensor_type	(I) tensor 类型
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_node_output_tensor(node_t node, int output_idx, tensor_t tensor, int tensor_type);	

10.2.3.10 get_node_output_number

名称	get_node_output_number	
介绍	获取 node 的输出张量(tensor)的个数	
参数	node_t node	(I) node 句柄
返回值	int	>0 : 输出张量(tensor)的个数 -1 : 表示失败
原型	int get_node_output_number(node_t node);	

10.2.3.11 get_node_input_number

名称	get_node_input_number	
介绍	获取 node 中输入张量(tensor)的个数	
参数	node_t node	(I) node 句柄
返回值	int	>0 : 表示输入张量(tensor)的个数

	-1 : 表示失败
原型	int get_node_input_number(node_t node);

10.2.3.12 add_node_attr

名称	add_node_attr	
介绍	添加 node 属性	
参数	node_t node	(I) node 句柄
	const char* attr_name	(I) 属性名称
	const char* type_name	(I) 属性类型名称
	int size	(I) 属性大小
返回值	int	0 : success -1 : failure
原型	int add_node_attr(node_t node, const char *attr_name, const char* type_name, int size);	

10.2.3.13 get_node_attr_int

名称	get_node_attr_int	
介绍	获取 node 的 int 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称

	int * attr_val	(O) int 参数返回值指针
返回值	int	0 : 表示获取成功 -1 : 表示失败
原型	int get_node_attr_int(node_t node, const char * attr_name, int * attr_val);	

10.2.3.14 get_node_attr_float

名称	get_node_attr_float	
介绍	获取 node 的 float 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	float * attr_val	(O) float 参数返回值的指针
返回值	int	0 : 表示获取成功 -1 : 表示失败
原型	int get_node_attr_float(node_t node, const char * attr_name, float * attr_val);	

10.2.3.15 get_node_attr_pointer

名称	get_node_attr_pointer	
介绍	获取 node 的指针类型参数	
参数	node_t node	(I) node 句柄

	const char * attr_name	(I) 参数名称
	void * attr_val	(O) pointer 参数的指针
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int get_node_attr_pointer(node_t node, const char * attr_name, void * attr_val);	

10.2.3.16 get_node_attr_generic

名称	get_node_attr_generic	
介绍	获取 node 的通用参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const char * type_name	(I) 参数类型名称
	void * buf	(O) 数据的 buffer
	int size	(I) 数据的 buffer 的尺寸
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int get_node_attr_generic(node_t node, const char * attr_name, const char* type_name, void * buf, int size);	

10.2.3.17 set_node_attr_int

名称	set_node_attr_int
----	-------------------

介绍	设置 node 的 int 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const int * attr_val	(I) int 参数返回值的指针
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int set_node_attr_int(node_t node, const char * attr_name, const int *attr_val);	

10.2.3.18 set_node_attr_float

名称	set_node_attr_float	
介绍	设置 node 的 float 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const float * attr_val	(I) float 型返回值的指针
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int set_node_attr_float(node_t node, const char * attr_name, const float *attr_val);	

10.2.3.19 set_node_attr_pointer

名称	set_node_attr_pointer
-----------	-----------------------

介绍	设置 node 的指针类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const void * attr_val	(I) 指针类型参数的指针
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int set_node_attr_pointer(node_t node, const char * attr_name, const void *attr_val);	

10.2.3.20 set_node_attr_generic

名称	set_node_attr_generic	
介绍	设置 node 的通用参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const char* type_name	(I) 类型名称
	const void * buf	(I) 数据的 buffer
	int size	(I) 数据的 buffer 的尺寸
返回值	int	0 : 表示获取成功 -1 : 表示获取失败
原型	int set_node_attr_generic(node_t node, const char * attr_name, const char * type_name, const void * buf, int size);	

10.2.4 kernel 相关操作

10.2.4.1 set_custom_kernel

名称	set_custom_kernel	
介绍	设置自定义 kernel	
参数	node_t node	(I) node 句柄
	const char * dev_name	(I) 设备名称
	struct custom_kernel_ops * kernel_ops	(I) 内核操作结构体指针
返回值	int	0 : 表示设置成功 -1 : 表示设置失败
原型	int set_custom_kernel(node_t node, const char * dev_name, struct custom_kernel_ops * kernel_ops);	

10.2.4.2 remove_custom_kernel

名称	remove_custom_kernel	
介绍	移除自定义 kernel	
参数	node_t node	(I) node 句柄
	const char * dev_name	(I) 设备名称
返回值	int	0 : 表示设置成功 -1 : 表示设置失败
原型	int remove_custom_kernel(node_t node, const char * dev_name);	

10.2.5 The tensor operation

10.2.5.1 create_graph_tensor

名称	create_graph_tensor	
介绍	创建 graph 的 tensor	
参数	graph_t graph	(I) graph 的句柄
	const char * tensor_name	(I) tensor 的名称
	int data_type	(I) 数据类型
返回值	tensor_t	tensor 的句柄, 如果为 NULL 表示失败
原型	tensor_t create_graph_tensor(graph_t graph, const char * tensor_name, int data_type);	

10.2.5.2 get_graph_tensor

名称	get_graph_tensor	
介绍	获取 graph 的 tensor	
参数	graph_t graph	(I) graph 的句柄
	const char * tensor_name	(I) tensor 的名称
返回值	tensor_t	tensor 的句柄, 如果为 NULL 表示失败
原型	tensor_t get_graph_tensor(graph_t graph, const char * tensor_name);	

10.2.5.3 get_tensor_name

名称	get_tensor_name	
介绍	获取 tensor 的名称	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	const char *	tensor 名称, 如果为 NULL 表示失败
原型	const char * get_tensor_name(tensor_t tensor);	

10.2.5.4 release_graph_tensor

名称	release_graph_tensor	
介绍	释放 graph 的 tensor	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	无	
原型	void release_graph_tensor(tensor_t tensor);	

10.2.5.5 get_tensor_shape

名称	get_tensor_shape	
介绍	获取 tensor 的 shape	
参数	tensor_t tensor	(I) tensor 的句柄
	int dims[]	(O) shape 的数组
	int dim_number	(I) 数组的尺寸

返回值	int	0 : 表示成功 -1 : 表示失败
原型	int get_tensor_shape(tensor_t tensor, int dims[], int dim_number);	

10.2.5.6 set_tensor_shape

名称	set_tensor_shape	
介绍	设置 tensor 的 shape	
参数	tensor_t tensor	(I) tensor 的句柄
	const int dims[]	(I) shape 的数组
	int dim_number	(I) 数组的尺寸
返回值	int	0 : 表示设置成功 -1 : 表示设置失败
原型	int set_tensor_shape(tensor_t tensor, const int dims[], int dim_number);	

10.2.5.7 get_tensor_buffer_size

名称	get_tensor_buffer_size	
介绍	获取 tensor 的 buffer 的大小	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	Int	>0 : buffer size -1 : failure

原型	<code>int get_tensor_buffer_size(tensor_t tensor);</code>
----	---

10.2.5.8 get_tensor_buffer

名称	get_tensor_buffer	
介绍	获取 tensor 的 buffer	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	void *	Buffer 的指针, 如果为 NULL 表示失败
原型	<code>void * get_tensor_buffer(tensor_t tensor);</code>	

10.2.5.9 set_tensor_buffer

名称	set_tensor_buffer	
介绍	设置 tensor 的 buffer	
参数	tensor_t tensor	(I) tensor 的句柄
	void * buffer	(I) Buffer 的指针
	int buffer_size	(I) 数据 buffer 的尺寸
返回值	int	0 : 表示成功 -1 : 表示失败
原型	<code>int set_tensor_buffer(tensor_t tensor, void * buffer, int buffer_size);</code>	

10.2.5.10 get_tensor_data

名称	get_tensor_data	
介绍	获取 tensor 的 data	
参数	tensor_t tensor	(I) tensor 的句柄
	void * output_data	(O) 输出 buffer 的指针
	int data_size	(I) 输出数据 buffer 的大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int get_tensor_data(tensor_t tensor, void * output_data, int data_size);	

10.2.5.11 set_tensor_data

名称	set_tensor_data	
介绍	设置 tensor 的数据	
参数	tensor_t tensor	(I) tensor 的句柄
	const void * input_data	(I) input 数据 buffer 的指针
	int data_size	(I) Input 数据的尺寸
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_tensor_data(tensor_t tensor, const void * input_data, int data_size);	

10.2.5.12 get_tensor_data_type

名称	get_tensor_data_type	
介绍	获取 tensor 的数据类型	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	int	Tensor 类型。比如： ENGINE_DT_FP32、 ENGINE_DT_FP16、 ENGINE_DT_IN8, 等等 -1：失败
原型	int get_tensor_data_type(tensor_t tensor);	

10.2.5.13 set_tensor_data_type

名称	set_tensor_data_type	
介绍	设置 tensor 的数据类型	
参数	tensor_t tensor	(I) tensor 的句柄
	int data_type	(I) 数据类型, 比如： ENGINE_DT_FP32、 ENGINE_DT_FP16、 ENGINE_DT_IN8, 等等
返回值	int	0: success

		-1 : failure
原型	int set_tensor_data_type(tensor_t tensor, int data_type);	

10.2.5.14 set_tensor_quant_param

名称	set_tensor_quant_param	
介绍	设置 tensor 的量化参数	
参数	tensor_t tensor	(I) tensor 的句柄
	const float *scale	(I) scale 指针
	const int *zero_point	(I) 起点指针
	int number	数据大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_tensor_quant_param(tensor_t tensor, const float *scale, const int *zero_point, int number);	

10.2.5.15 get_tensor_quant_param

名称	get_tensor_quant_param	
介绍	获取 tensor 的量化参数	
参数	tensor_t tensor	(I) 张量句柄
	float *scale	(I) scale 指针

	int *zero_point	(I) 起点指针
	int number	(I) 数据大小
返回值	int	0 : 表示成功 -1 : 表示失败

10.2.6 与 graph 运行相关 API

10.2.6.1 set_graph_attr

名称	set_graph_attr	
介绍	设置 graph 的参数	
参数	graph_t graph	(I) graph 的句柄
	const char * attr_name	(I) 要设置的参数名称
	const void * buf	(I) 设置的参数的指针
	int size	(I) 数据的尺寸
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_graph_attr(graph_t graph, const char * attr_name, const void * buf, int size);	

10.2.6.2 get_graph_attr

名称	get_graph_attr
介绍	获取 graph 的参数

参数	graph_t graph	(I) graph 的句柄
	const char * attr_name	(I) 要获取的参数名称
	void * buf	(I) 要获取参数的指针
	int size	(I) 数据的大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int get_graph_attr(graph_t graph, const char * attr_name, void * buf, int size);	

10.2.6.3 set_graph_gd_method

名称	set_graph_gd_method	
介绍	设置图的梯度下降方法	
参数	graph_t graph	(I) graph 的句柄
	int gd_method	(I) 梯度下降函数
	...	(I) 可变参数
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_graph_gd_method(graph_t graph, int gd_method, ...);	

10.2.6.4 prerun_graph

名称	prerun_graph
----	--------------

介绍	准备运行 graph, 并准备资源	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int prerun_graph(graph_t graph);	

10.2.6.5 run_graph

名称	run_graph	
介绍	运行 graph, 执行推理, 可以反复多次调用	
参数	graph_t graph	(I) graph 的句柄
	int block	1: 阻塞. 需要配置 GRAPH_DONE 函数 0: 非阻塞: 需要调用 wait_graph()以获取结果
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int run_graph(graph_t graph, int block);	

10.2.6.6 wait_graph

名称	wait_graph	
介绍	等待 graph 运行的结果	
参数	graph_t graph	(I) graph 的句柄

	int try_wait	1: 检查状态, 然后返回 0: 立即返回
返回值	int	1: graph 运行完成 0: 未运行完成, 需要再次尝试
原型	int wait_graph(graph_t graph, int try_wait);	

10.2.6.7 postrun_graph

名称	postrun_graph	
介绍	停止运行 graph 并释放 graph 占据的资源	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0: 表示成功 -1: 表示失败
原型	int postrun_graph(graph_t graph);	

10.2.6.8 get_graph_exec_status

名称	get_graph_exec_status	
介绍	获取 graph 的执行状态	
参数	graph_t graph	(I) graph 的句柄
返回值	int	graph 的运行状态, 见 graph_exec_stat
原型	int get_graph_exec_status(graph_t graph);	

10.2.6.9 set_graph_event_hook

名称	set_graph_event_hook	
介绍	设置 graph 的事件回调函数	
参数	graph_t graph	(I) graph 的句柄
	int event	(I) 事件类型
	event_handler_t cb_func	(I) 事件回调函数指针
	void * cb_arg	(I) 回调参数
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_graph_event_hook(graph_t graph, int event, event_handler_t cb_func, void * cb_arg);	

10.2.7 与设备相关操作

10.2.7.1 set_default_device

名称	set_default_device	
介绍	设置默认运行设备	
参数	const char * device	(I) 运行的设备名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_default_device(const char * device);	

10.2.7.2 set_graph_device

名称	set_graph_device	
介绍	设置 graph 运行设备	
参数	graph_t graph	(I) graph 的句柄
	const char * dev_name	(I) 设备的名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_graph_device(graph_t graph, const char * dev_name);	

10.2.7.3 set_node_device

名称	set_node_device	
介绍	设置 node 运行设备	
参数	node_t node	(I) node 的句柄
	const char * dev_name	(I) 设备的名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_node_device(node_t node, const char * dev_name);	

10.2.7.4 get_node_device

名称	get_node_device
----	-----------------

介绍	获取 node 运行设备	
参数	node_t node	(I) node 的句柄
返回值	const char *	设备的名称: NULL 意味着没有可利用的设备
原型	const char * get_node_device(node_t node);	

10.2.7.5 do_node_dump

名称	do_node_dump	
介绍	设置节点的 dump 功能	
参数	node_t node	(I) 节点句柄
	int action	(I) 1 使能, 0 不使能
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int do_node_dump(node_t node, int action);	

10.2.7.6 get_node_dump_buffer

名称	get_node_dump_buffer	
介绍	获取节点的 dump 数据缓存区。dump 数据缓冲区的具体定义, 由产生 dump 的设备决定。	
参数	node_t node	(I) 节点句柄

	void ** buf	(O) 用于接收 dump 数据块指针的指针数组
	int buf_size	(I) 指针数组大小
返回值	int	指针数组中有效指针的个数。如果等于 buf_size, 则存在有数据块没有返回的可能性
原型	int get_node_dump_buffer(node_t node, void **buf, int buf_size);	

10.2.7.7 do_graph_perf_stat

名称	do_graph_perf_stat	
介绍	启用或禁用性能统计	
参数	graph_t graph	(I) graph 的句柄
	int action	(I) 0 停止, 1 开始, 2 重新计数
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int do_graph_perf_stat(graph_t graph, int action);	

10.2.7.8 get_graph_perf_stat

名称	get_graph_perf_stat	
介绍	获取性能统计信息	
参数	graph_t graph	(I) graph 的句柄

	perf_info ** buf	(O) 指向(输出性能信息的指针)的指针
	int buf_size	(I) 数据的大小
返回值	int	>=0: 表示数据记录的个数 -1 : 表示失败
原型	int get_graph_perf_stat(graph_t graph, struct perf_info ** buf, int buf_size);	

10.2.7.9 get_device_number

名称	get_device_number	
介绍	获取设备编号	
参数	无	
返回值	int	设备编号
原型	int get_device_number(void);	

10.2.7.10 get_device_name

名称	get_device_name	
介绍	按 index 获取设备名称	
参数	int idx	(I) 设备编号(>=0)
返回值	const char *	设备的名称: NULL 意味着失败
原型	const char * get_device_name(int idx);	

10.2.7.11 get_default_device

名称	get_default_device	
介绍	获取默认使用设备名称	
参数	无	
返回值	const char *	设备的名称： NULL 意味着失败
原型	const char * get_default_device(void);	

10.2.7.12 create_device

名称	create_device	
介绍	创建设备	
参数	const char * driver_name	驱动名称
	const char * dev_name	(I) 设备名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int create_device(const char * driver_name, const char * dev_name);	

10.2.7.13 destroy_device

名称	destroy_device	
介绍	销毁设备	
参数	const char * driver_name	(I) 驱动名称

	const char * dev_name	(I) 设备名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int destroy_device(const char * driver_name, const char * dev_name);	

10.2.7.14 set_device_policy

名称	set_device_policy	
介绍	设置设备策略	
参数	const char * driver_name	(I) 驱动名称
	device_policy policy	(I) 政策
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_device_policy(const char * device_name, device_policy policy);	

10.2.7.15 get_device_policy

名称	get_device_policy	
介绍	获取设备策略	
参数	const char * driver_name	(I) 驱动名称
返回值	int	>=0: 策略值 -1 : 表示失败

原型	<code>int get_device_policy(const char * device_name);</code>
-----------	---

10.2.7.16 get_device_attr

名称	get_device_attr	
介绍	获取设备参数	
参数	const char * driver_name	(I) 驱动名称
	const char * attr_name	(I) 参数名称
	void * val	(O) 参数返回结果指针
	int size	(I) 参数返回值大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int get_device_attr(const char * device_name, const char * attr_name, void * val, int size);	

10.2.7.17 set_device_attr

名称	set_device_attr	
介绍	设置设备参数	
参数	const char * driver_name	(I) 驱动名称
	const char * attr_name	(I) 参数名称
	void * val	(I) 参数指针
	int size	(I) 参数大小

返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_device_attr(const char * device_name, const char * attr_name, void * val, int size);	

10.2.8 与 context 相关部分

10.2.8.1 create_context

名称	create_context	
介绍	创建 context	
参数	const char * context_name	(I) context 的名称
	int empty_context	(I) 1: 没有为该设备分配可用的设备 0: 所有声明的设备都将添加到 context 中
返回值	context_t	Context 的句柄, 返回 NULL 表示创建失败
原型	context_t create_context(const char * context_name, int empty_context);	

10.2.8.2 destroy_context

名称	destroy_context	
介绍	销毁 context	
参数	context_t context	(I) context 的句柄
返回值	无	

原型	<code>void destroy_context(context_t context);</code>
----	---

10.2.8.3 get_context_device_number

名称	<code>get_context_device_number</code>	
介绍	获取 context 设备编号	
参数	<code>context_t context</code>	(I) context 的句柄
返回值	<code>int</code>	context 使用的设备编号
原型	<code>int get_context_device_number(context_t context);</code>	

10.2.8.4 get_context_device_name

名称	<code>get_context_device_name</code>	
介绍	获取 context 设备名称	
参数	<code>context_t context</code>	(I) context 的句柄
	<code>int idx</code>	(I) 索引
返回值	<code>const char *</code>	设备名称
原型	<code>const char * get_context_device_name(context_t context, int idx);</code>	

10.2.8.5 add_context_device

名称	<code>add_context_device</code>
介绍	将设备添加到 context 中

参数	context_t context	(I) context 的句柄
	const char * dev_name	(I) 设备名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int add_context_device(context_t context, const char * dev_name);	

10.2.8.6 remove_context_device

名称	remove_context_device	
介绍	将设备移除出 context	
参数	context_t context	(I) context 的句柄
	const char * dev_name	(I) device 名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int remove_context_device(context_t context, const char * dev_name);	

10.2.8.7 set_context_attr

名称	set_context_attr	
介绍	设置 context 参数	
参数	context_t context	(I) context 的句柄

	const char * attr_name	(I) 参数名称
	const void * val	(I) 设置参数值的指针
	int val_size	(I) 参数的值所占的大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int set_context_attr(context_t context, const char * attr_name, const void * val, int val_size);	

10.2.8.8 get_context_attr

名称	get_context_attr	
介绍	获取 context 属性	
参数	context_t context	(I) context 的句柄
	const char * attr_name	(I) 参数名称
	void * val	(O) 参数返回值的指针
	int val_size	(I) 参数的值所占的大小
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int get_context_attr(context_t context, const char * attr_name, void * val, int val_size);	

10.2.9 其他的 API

10.2.9.1 get_tengine_errno

名称	get_tengine_errno	
介绍	获取错误码	
参数	无	
返回值	int	tengine 错误码。定义遵循 glibc
原型	int get_tengine_errno(void);	

10.2.9.2 set_log_level

名称	set_log_level	
介绍	设置 log 级别	
参数	log_level level	(I) 上下文的句柄
返回值	无	
原型	void set_log_level(enum log_level level);	

10.2.9.3 set_log_output

名称	set_log_output	
介绍	设置 log 输出函数	
参数	log_print_t func	(I) 输出函数的指针

返回值	无	
原型	void set_log_output(log_print_t func);	

10.2.9.4 dump_graph

名称	dump_graph	
介绍	dump 图的结构	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0 : 表示成功 -1 : 表示失败
原型	void dump_graph(graph_t graph);	

10.2.9.5 load_tengine_plugin

名称	load_tengine_plugin	
介绍	加载 Tengine 插件	
参数	const char * plugin_name	(I) 插件的名称
	const char * fname	(I) 文件的名称
	const char * init_func_name	(I) 初始化函数的名称
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int load_tengine_plugin(const char * plugin_name, const char * fname, const char * init_func_name);	

10.2.9.6 unload_tengine_plugin

名称	unload_tengine_plugin	
介绍	移除 Tengine 插件	
参数	const char * plugin_name	(I) 插件的名称
	const char * rel_func_name	(I) release 函数的指针
返回值	int	0 : 表示成功 -1 : 表示失败
原型	int unload_tengine_plugin(const char * plugin_name, const char * rel_func_name);	

10.2.9.7 get_tengine_plugin_number

名称	get_tengine_plugin_number	
介绍	获取 Tengine 插件的编号	
参数	无	
返回值	int	插件数量
原型	int get_tengine_plugin_number(void);	

10.2.9.8 get_tengine_plugin_name

名称	get_tengine_plugin_name	
介绍	获取 Tengine 插件的名称	
参数	int idx	(I) 插件的编号(>=0)

返回值	const char *	插件名称
原型	const char * get_tengine_plugin_name(int idx);	

10.3 错误码

Tengine C API 错误码如下表所示。

错误代码	描述
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Arg list too long
8	Exec format error
9	Bad file number
10	No child processes

更多错误码信息请参考 Linux Error Code 标准定义。