寒武纪 EasyDK 用户手册

发布 *latest*

2022 年 04 月 11 日

目录

1 版权声明 **1**

2 前言 **3**

2.1 版本记录 . . . 3

2.2 更新历史 . . . 3

3 介绍 **7**

4 快速入门 **10**

4.1 构建和运行环境要求 . . . 10

4.2 源码编译 EasyDK . . . 11

4.3 EasyDK 开发示例 . . . 13

4.3.1 开发示例目录结构 . . . 13

4.3.2 编译和运行开发示例 . . . 14

5 模块 **16**

5.1 Cxxutil . . . 16

5.1.1 异常 . . . 16

5.2 Device . . . 17

5.2.1 设备操作 . . . 17

5.2.2 MLU 任务队列 . . . 18

5.3 EasyCodec . . . 18

5.3.1 解码 . . . 18

5.3.2 编码 . . . 20

5.4 EasyInfer . . . 21

5.4.1 内存操作 . . . 21

5.4.2 模型 . . . 22

5.4.3 推理 . . . 23
5.5 EasyBang . . . 23
5.5.1 ResizeConvert 算子 . . . 23
5.5.2 Resize 算子 . . . 24
5.6 EasyTrack . . . 25
5.6.1 FeatureMatch . . . 25

6 推理服务 27
6.1 介绍 . . . 27
6.1.1 基本概念 . . . 27
6.1.1.1 InferServer . . . 28
6.1.1.2 ModelInfo . . . 30
6.1.1.3 Session . . . 30
6.1.1.4 Processor . . . 31
6.1.1.5 InferData . . . 31
6.1.1.6 Package . . . 32
6.1.1.7 Observer . . . 32
6.1.2 功能 . . . 32
6.1.2.1 模型加载与管理 . . . 32
6.1.2.2 推理任务调度 . . . 34
6.1.2.3 后端处理单元（Processor） . . . 35
6.1.2.4 扩展接口（contrib） . . . 39
6.2 编程指南 . . . 41
6.2.1 同步或异步请求 . . . 41
6.2.2 等待处理结束或遗弃任务 . . . 42
6.2.3 自定义数据结构 . . . 42
6.2.4 自定义后端处理单元 . . . 43
6.2.5 特殊用法 . . . 44
6.2.6 性能调优方法 . . . 44
6.3 python 封装 . . . 46
6.3.1 使用说明 . . . 46
6.3.2 编程模型 . . . 48
6.3.2.1 同步推理 . . . 48
6.3.2.2 异步推理 . . . 49
6.3.2.3 仅推理 . . . 52
6.3.3 自定义前后处理 . . . 52

7 示例代码说明 54
7.1 深度学习应用示例 . . . 54
7.2 推理服务示例 . . . 61

7.2.1 推理示例 61
7.2.2 引入新模型示例 63
7.2.3 自定义后处理单元示例 64

8 FAQ **66**
8.1 有没有交叉编译 EasyDK 的指导? 66
8.2 如何将 Log 打印到终端以及如何带颜色显示? 66
8.3 怎么调整 Log 打印等级? 66

1

版权声明

免责声明

中科寒武纪科技股份有限公司（下称“寒武纪”）不代表、担保（明示、暗示或法定的）或保证本文件所含信息，并明示放弃对可销售性、所有权、不侵犯知识产权或特定目的适用性做出任何和所有暗示担保，且寒武纪不承担因应用或使用任何产品或服务而产生的任何责任。寒武纪不应对因下列原因产生的任何违约、损害赔偿、成本或问题承担任何责任：（1）使用寒武纪产品的任何方式违背本指南；或（2）客户产品设计。

责任限制

在任何情况下，寒武纪都不对因使用或无法使用本指南而导致的任何损害（包括但不限于利润损失、业务中断和信息损失等损害）承担责任，即便寒武纪已被告知可能遭受该等损害。尽管客户可能因任何理由遭受任何损害，根据寒武纪的产品销售条款与条件，寒武纪为本指南所述产品对客户承担的总共和累计责任应受到限制。

信息准确性

本文件提供的信息归寒武纪所有，且寒武纪保留不经通知随时对本文件信息或对任何产品和服务做出任何更改的权利。本指南所含信息和本指南所引用寒武纪文档的所有其他信息均“按原样”提供。寒武纪不担保信息、文本、图案、链接或本指南内所含其他项目的准确性或完整性。寒武纪可不经通知随时对本指南或本指南所述产品做出更改，但不承诺更新本指南。

本指南列出的性能测试和等级要使用特定芯片或计算机系统或组件来测量。经该等测试，本指南所示结果反映了寒武纪产品的大概性能。系统硬件或软件设计或配置的任何不同会影响实际性能。如上所述，寒武纪不代表、担保或保证本指南所述产品将适用于任何特定用途。寒武纪不代表或担保测试每种产品

的所有参数。客户全权承担确保产品适合并适用于客户计划的应用以及对应用程序进行必要测试的责任，以期避免应用程序或产品的默认情况。

客户产品设计的脆弱性会影响寒武纪产品的质量和可靠性并导致超出本指南范围的额外或不同的情况和/或要求。

知识产权通知

寒武纪和寒武纪的标志是中科寒武纪科技股份有限公司在中国和其他国家的商标和/或注册商标。其他公司和产品名称应为与其关联的各自公司的商标。

2

前言

2.1 版本记录

表 1: 版本记录

文档名称	寒武纪 EasyDK 用户手册
版本号	Vlatest
作者	Cambricon
修改日期	2022.4.11

2.2 更新历史

- v3.0.0

  更新时间：2022 年 1 月 5 日

  更新内容：

  - 合并 `InferServer` 到 `EasyDK`。
  - 增加 `InferServer` 的 python 接口，支持同步和异步推理。
  - 支持 `InferServer` 处理输出形状可变的模型。
  - 增加 `InferServer::Preprocessor` 预处理器，在预处理函数中处理 batch 数据。

  - 修改 `InferServer::PreprocessorHost` 的 `ProcessFunction` 类型。
  - 修改 `InferServer::Postprocessor` 的 `ProcessFunction` 类型。
  - 增加 `WITH_BACKWARD` 编译选项。
  - 兼容 CNRT 4.x 和 5.x 版本。
  - 优化 `EasyDecode`, 支持 MLU300 系列解码。
  - 优化 `EasyEncode`, 支持 MLU300 系列编码。
  - 支持 `EasyEncode` 从编码器获取 MLU 内存，减少内存拷贝。
  - 修改部分 `EasyEncode::Attr` 参数。
  - 增加三种解码方式示例，支持使用 `EasyDecode`, `FFmpeg` 和 `FFmpeg-MLU` 三种解码方式。
  - 替换示例中的推理部分，使用 `InferServer` 进行推理。
  - 增加以下接口:
    * `EasyEncode::FeedData`
    * `EasyEncode::FeedEos`
    * `EasyEncode::RequestFrame`
  - 废弃接口 `EasyEncode::SendDataCPU`，使用 `EasyEncode::FeedData` 和 `EasyEncode::FeedEos`。
  - 废弃 `EasyEncode::FeedData` 接口的 `integral_frame` 参数

- v2.6.0

  更新时间：2021 年 10 月 11 日

  更新内容：

  - 移除 `easy_plugin` 模块。
  - 优化 `FeatureMatchTrack` 性能。
  - 移除日志颜色。
  - 修复 JPEG 编码未处理错误帧。
  - `EasyInfer` 构造函数不再需要绑定设备。
  - 修正 `MluMemoryOp` 日志中错误的内存大小。
  - 移除以下废弃接口：
    * `EasyDecode::SendData`
    * `EasyInfer::WithRGB0Output`
    * `EasyInfer::WithYUVInput`

- v2.5.0

  更新时间：2021 年 3 月 23 日

  更新内容：

  - 参考 Google log（glog）实现日志系统，替换 glog。
  - 增加 `TimeMark` 用于测量硬件时间。
  - 设置 `cnrtSetDeviceFlag` ，使能驱动代理同步队列，降低 CPU 占用。
  - 移除 `EasyCodec` 模块以下繁琐参数，简化 Codec 模块使用：
    * `EasyDecode::Attr::buf_strategy`
    * `EasyEncode::Attr::crop_config`

    * `EasyEncode::Attr::ir_count`
    * `EasyEncode::Attr::max_mb_per_slice`
    * `EasyEncode::Attr::cabac_init_idc`

- v2.4.0

  更新时间：2020 年 12 月 31 日

  更新内容：

  - 移除 v2.2.0 废弃的 API。
  - 丰富示例代码，支持 rtsp 拉流解码，保存推理结果至本地视频文件等功能。
  - [Early Access] 增加推理服务模块，详见《寒武纪推理服务用户手册》。
  - 提升追踪模块 `FeatureMatchTrace` 精确度。
  - `EasyInfer` 增加异步推理接口。
  - 修复 `MluResizeConvertOp` 未凑齐 batch 时运行算子发生段错误的问题（该 BUG 由 v2.3.0 引入）。

- v2.3.0

  更新时间：2020 年 11 月 30 日

  更新内容：

  - 支持交叉编译 MLU220EDGE 平台。
  - 使用 ShapeEx 代替 Shape，支持任意维度数据。
  - 废弃接口 `ModelLoader::InputShapes` 和 `ModelLoader::OutputShapes`。
  - 增加接口 `ModelLoader::InputShape(uint32_t index)` 和 `ModelLoader::OutputShape(uint32_t index)`。
  - 优化 EasyCodec 实现。

- v2.2.0

  更新时间：2020 年 11 月 5 日

  更新内容：

  - 废弃以下不合理的接口（将于 v2.4.0 版本删除）：
    * `MluContext::(ConfigureForThisThread, ChannelId, SetChannelId)`
    * `MluResize::InvokeOp`
    * `EasyDecode::Create, EasyEncode::Create`
    * `EasyInfer::(Init, Loader, BatchSize)`
    * `ModelLoader::InitLayout`
    * `MluMemoryOp::(SetLoader, Loader, AllocCpuInput, AllocCpuOutput, AllocMluInput, AllocMluOutput, AllocMlu, FreeArrayMlu, MemcpyInputH2D, MemcpyOutputD2H, MemcpyH2D, MemcpyD2H)`
  - 替换示例代码中使用的上述废弃接口。
  - EasyDecode 支持解码 Progressive-JPEG。
  - 增加查询设备数量的接口 `edk::MluContext::GetDeviceNum()`。
  - 修复 MluResizeConvertOp 部分规模不支持的问题。
  - 使用统一的异常类型，详见异常 。

  - 增加 Resize（yuv2yuv）的算子。
  - 重构 MluTaskQueue 为 C++ 风格。

- v2.1.0

  更新时间：2020 年 10 月 26 日

  更新内容：

  - 初始版本。

3

介绍

EasyDK（Cambricon Neuware Easy Development Kit）提供了一套面向 MLU（Machine Learning Unit，寒武纪机器学习单元）设备的高级别接口（C++11 标准），用于面向 MLU 平台快速开发和部署深度学习应用。

EasyDK 包含如下 6 个模块:

- EasyTrack：提供目标追踪的功能。
- EasyBang：提供简易调用 BANG 算子的接口，目前支持的算子包括 `ResizeConvert` 和 `Resize`。
- EasyInfer：提供离线模型推理相关功能。
- EasyCodec：提供支持视频与图片的 MLU 硬件编解码功能。
- Device：提供 MLU 设备 Context 相关操作。
- cxxutil：其他模块用到的部分 cpp 实现。

下图为 EasyDK 的模块结构：

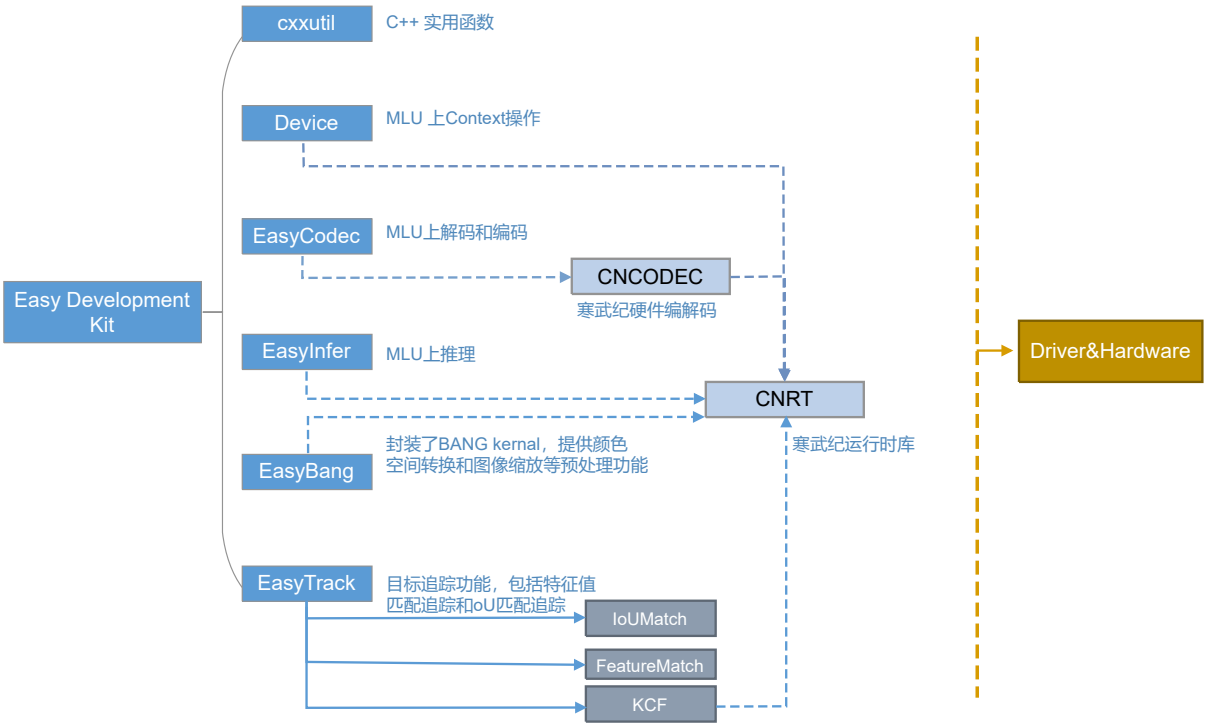

图 1: EasyDK 结构

EasyDK 还包含推理服务组件：提供了一套面向 MLU（Machine Learning Unit，寒武纪机器学习单元）类似服务器的推理接口（C++11 标准），以及模型加载与管理，推理任务调度等功能，极大地简化了面向 MLU 平台高性能深度学习应用的开发和部署工作。

推理服务在寒武纪软件栈中的位置，如下图所示：

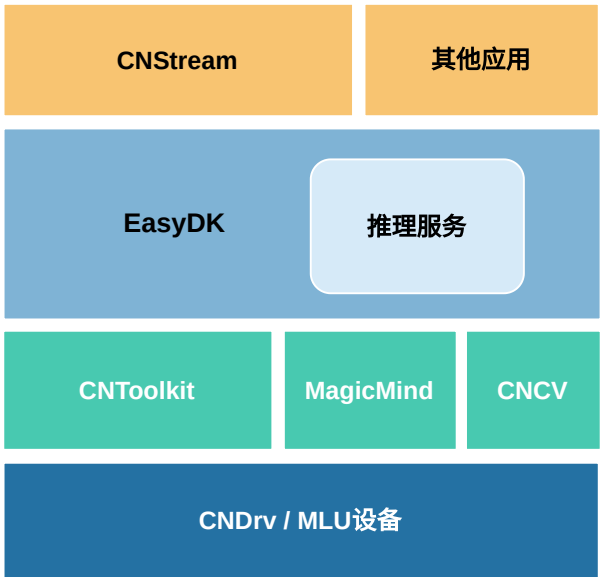

图 2: 寒武纪软件栈

推理服务共包含以下 3 个模块的用户接口:

- Model: 模型加载与管理
- Processor: 可自定义的后端处理单元
- InferServer: 执行推理任务

4

快速入门

4.1 构建和运行环境要求

构建和运行 EasyDK 有如下依赖：

- CMake 2.8.7+
- GCC 4.8.5+
- GLog 0.3.4
- Cambricon Neuware Toolkit >= 1.5.0
- CNCV >= 0.4.0 (optional)
- libcurl-dev (optional)

使用 Magicmind 后端有以下额外依赖：

- Cambricon Neuware Toolkit >= 2.4.2
- Magicmind >= 0.8.2

（Magicmind 运行时库引入对 CNNL、CNNL_extra、CNLight 的依赖，版本要求见 Magicmind 用户手册。）

使用 Mlu300 系列编解码有一下额外依赖：

- Cambricon Neuware Toolkit >= 2.6.2
- CNCodec_v3 >= 0.8.2

测试程序和示例有以下额外依赖：

- OpenCV 2.4.9+

- GFlags 2.1.2
- FFmpeg 2.8 3.4 4.2

4.2 源码编译 EasyDK

EasyDK 仅支持源码编译的方式使用，按如下步骤编译 EasyDK（${EASYDK_DIR}，代表 EasyDK 源码目录）：

1. 创建编译文件夹存储编译结果。

```
cd ${EASYDK_DIR}
mkdir build
```

2. 运行 CMake 配置编译选项，并生成编译指令。
   寒武纪 EasyDK 提供了一个 CMakeLists.txt 描述编译流程，用户可以从 CMake 官网[1] 免费下载和使用 CMake。

[1] http://www.cmake.org/

cmake 选项	范围	默认值	描述
BUILD_SAMPLES	ON / OFF	OFF	编译 sample
BUILD_TESTS	ON / OFF	OFF	编译 test
WITH_CODEC	ON / OFF	ON	编译 EasyCodec
WITH_INFER	ON / OFF	ON	编译 EasyInfer
WITH_TRACKER	ON / OFF	ON	编译 EasyTracker
WITH_BANG	ON / OFF	ON	编译 EasyBang
WITH_BACKWARD	ON / OFF	ON	编译 Backward
WITH_TURBOJPEG	ON / OFF	OFF	编译 turbo-jpeg
ENABLE_KCF	ON / OFF	OFF	Easytrack 支持 KCF
WITH_INFER_SERVER	ON / OFF	ON	编译推理服务
CNIS_WITH_CONTRIB	ON / OFF	ON	编译推理服务 contrib 内容
CNIS_RECORD_PERF	ON / OFF	ON	使能推理服务详细性能信息的测量和记录
CNIS_WITH_CURL	ON / OFF	ON	使能 CURL，支持推理服务从网络下载模型
CNIS_USE_MAGICMIND	ON / OFF	OFF	使能推理服务使用 Magmicmind 作为推理后端
CNIS_WITH_PYTHON_API	ON / OFF	OFF	使能推理服务 python 接口

示例:

```
cd build
cmake ${EASYDK_DIR} \
      -DBUILD_SAMPLES=ON \
      -DBUILD_TESTS=ON
```

3. 运行编译指令。

```
make
```

4. 编译后的库文件存放在 ${EASYDK_DIR}/lib；头文件存放在 ${EASYDK_DIR}/include；推理服务头文件存放在 ${EASYDK_DIR}/infer_server/include。
5. 如果想要交叉编译 EasyDK，则需要事先交叉编译并安装第三方依赖库，并配置 CMAKE_TOOLCHAIN_FILE 文件，以 MLU220-SOM 为例：

```
export NEUWARE_HOME=/your/path/to/neuware
export PATH=$PATH:/your/path/to/cross-compiler/bin
cmake ${EASYDK_DIR} -DCMAKE_FIND_ROOT_PATH=/your/path/to/3rdparty-libraries-
↪install-path -DCMAKE_TOOLCHAIN_FILE=${EASYDK_DIR}/cmake/cross-compile.cmake ␣
↪-DCNIS_WITH_CURL=OFF
```

更多信息可以参考交叉编译。

4.3 EasyDK 开发示例

深度学习应用开发示例为用户提供了离线模型、视频文件、运行脚本以及开发示例代码，帮助用户快速了解如何使用 EasyDK 完成简单的深度学习应用部署。用户可以直接通过脚本运行示例代码，无需修改任何设置。

推理服务开发示例为用户提供了目标检测网络异步推理开发示例代码，帮助用户快速了解如何使用推理服务完成深度学习推理。用户可以通过可执行文件运行示例代码。

4.3.1 开发示例目录结构

深度学习应用和转码开发示例存放于 ${EASYDK_DIR}/samples/ 文件夹下，支持在 MLU270，MLU220 和 MLU370 平台下运行。开发示例包含的文件如下:

- common：OSD、后处理、特征提取、视频解析等示例通用功能。
- stream-app：检测示例。
- classification：分类示例。
- transcode：转码示例。
- CMakelists.txt：CMake 文件，编译示例时使用，用户无需做任何设置和改动。

- cmake 文件夹：存放 CMake 脚本。
- data：运行示例使用的视频文件。

推理服务开发示例存放于 ${EASYDK_DIR}/infer_server/samples/ 文件夹下，支持在 MLU270 和 MLU370 平台下运行。

4.3.2 编译和运行开发示例

1. 编译样例。编译 EasyDK 时打开 BUILD_SAMPLES 选项即可编译示例代码。
2. 运行深度学习应用样例。运行如下命令:

```
EXPORT LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${NEUWARE_HOME}/lib64
pushd samples/stream-app
./run_yolov5_270.sh          # MLU270 平台
./run_ssd_270.sh             # MLU270 平台
./run_yolov3_270.sh          # MLU270 平台
./run_yolov3_220.sh          # MLU220 平台
./run_yolov3_370.sh          # MLU370 平台，需要使能CNIS_USE_MAGICMIND
popd

pushd samples/classification
./run_resnet50_270.sh     # MLU270平台
./run_resnet18_220.sh     # MLU220 平台
./run_resnet50_370.sh     # MLU370 平台，需要使能CNIS_USE_MAGICMIND
```

运行结束后程序会自动退出。

3. 运行转码样例。运行如下命令:

```
EXPORT LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${NEUWARE_HOME}/lib64
pushd samples/transcode
./run_transcode.sh              # 支持MLU270，MLU220和MLU370平台。MLU370␣
↪平台，需要安装寒武纪CNCodec_v3库。
popd
```

4. 运行推理服务样例。运行如下命令：

```
EXPORT LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${NEUWARE_HOME}/lib64
pushd infer_server/samples/
# 运行单路输入demo，预处理使用OpenCV
./bin/cnis_demo  ${video_path}    # MLU270平台。如果使能CNIS_USE_␣
↪MAGICMIND，MLU370平台。
# 运行多路输入demo，预处理方式可选择使用OpenCV或CNCV。
#（预处理方式通过 ``${EASYDK_DIR}/infer_server/samples/multi_stream_demo.cpp``␣
↪USE_CNCV_PREPROC宏控制）。
```

(下页继续)

(续上页)

```
./bin/cnis_multi_stream_demo  ${video_path}   # MLU270平台。如果使能CNIS_USE_
↪MAGICMIND，MLU370平台。
popd
```

运行结束后程序会自动退出。`${video_path}` 代表输入视频路径。

5

模块

本章详细介绍 EasyDK 的各个模块。

5.1 Cxxutil

本模块提供了一些实用功能的封装，包括读写锁（基于 pthread 的简单 C++ 封装）、线程安全队列，统一的异常类型。

5.1.1 异常

EasyDK 中统一采用异常进行错误处理。仅有一种异常类型 `edk::Exception`，继承自 `std::exception`。异常中包含错误码、文件名、行数、函数名，补充信息等内容，调用 `edk::Exception::what()` 将得到格式如下的信息：

```
# print on shell
# |file_name: line    |function    |error_code   |error_message
mlu_context.cpp:172 (BindDevice) INVALID_ARG] channel id should less than 4
```


5.2 Device

本模块提供了 MLU 设备 Context 及队列等相关操作，包含设备初始化、设备绑定、MLU 任务队列等功能封装。

5.2.1 设备操作

设备操作被封装在 MluContext 类中，提供常用的设备管理接口。

在调用任何 MLU 上的接口之前均需调用 `edk::MluContext::BindDevice()`。将当前线程绑定到设备，设备号从 0 开始，取值范围在 [0, edk::MluContext::GetDeviceNum() - 1]。绑定设备后，该线程后续所有 MLU 的接口调用都会在该指定设备上执行。

```
// 获取设备数量
uint32_t dev_num = edk::MluContext::GetDeviceNum();

edk::MluContext ctx;

// 检查是否存在设备0
if (ctx.CheckDeviceId(0)) {
  // 绑定设备
  ctx.SetDeviceId(0);
  ctx.BindDevice();
} else {
  std::cout << "cannot find device 0\n";
  assert(0);
}

// 获取设备平台，可以通过平台运行时动态选择对应的BANG算子和离线模型
// 必须在绑定要查询的目标设备后调用GetCoreVersion
edk::CoreVersion version = ctx.GetCoreVersion();
```

在第一次调用 edk::MluContext::BindDevice() 、edk::MluContext::CheckDeviceId(int) 、edk::MluContext::GetDeviceNum() 、edk::MluContext::GetCoreVersion() 任一接口时，会自动进行设备的初始化。在进程结束，全局静态变量析构时销毁设备。用户在全局静态变量析构的过程中操作 MLU 设备（如释放 MLU 内存等）属于未定义行为（无法保证销毁设备和操作 MLU 设备的顺序）。

5.2.2 MLU 任务队列

`MluTaskQueue` 是对 `cnrtQueue` 的封装，用于在不同的 EasyDK 模块间传递 `cnrtQueue` ，同时不暴露 CNRT 的符号给用户。`MluTaskQueue_t` 是 `std::shared_ptr<MluTaskQueue>` 的别名。

在 MLU 上执行的任务都需要经由任务队列下发给设备，同一个队列中的任务串行，不同队列中的任务并行。

`MluTaskQueue` 提供了 RAII 的任务队列管理，可以在析构时自动释放 `cnrtQueue` 资源。

```
edk::MluTaskQueue_t q = edk::MluTaskQueue::Create();
edk::MluResizeConvertOp rc_op;
// 用户手动指定MLU任务队列
rc_op.SetMluQueue(q);
```

任务队列支持测量硬件时间，可以通过 `MluTaskQueue::PlaceMark()` 添加时间标记在任务两侧，`MluTaskQueue::Count` 计算两个标记中间执行任务的硬件时间。

```
edk::MluTaskQueue_t q = edk::MluTaskQueue::Create();
edk::EasyInfer infer;

// some init process
...

auto start = q->PlaceMark();
infer.RunAsync(input, output, q);
auto end = q->PlaceMark();
q->Sync();

float hw_time = q->Count(start, end);
```

5.3 EasyCodec

5.3.1 解码

EasyDecode 封装了 Mlu200 系列和 Mlu300 系列的硬件解码功能。

对于 Mlu200 系列，封装了 CNCodec 中解码的功能，支持视频和图片解码，格式如下:

- HEVC（H.265）：支持 Main Profile 和 Main10 Profile
- H.264: 支持 Baseline Profile、Main Profile、High Profile 和 High10 Profile
- VP8 所有规格
- VP9 Profile 0 和 VP9 Profile 2

- JPEG

视频支持的最大分辨率为 4096 x 4096，JPEG 支持的最大分辨率为 8192 x 4320。视频解码（H.264、HEVC、VP8、VP9）支持输出 NV12、NV21、I420、P010 格式的数据。JPEG 解码支持输出 YUYV、UYVY、NV12、NV21 格式的数据（硬解码不支持 Progressive JPEG）。

对于 Mlu300 系列，封装了 CNCodec_v3 中解码的功能，支持视频和图片解码，格式如下:

- HEVC（H.265）：支持 Main Profile 和 Main10 Profile
- H.264: 支持 Baseline Profile、Main Profile、High Profile 和 High10 Profile
- VP8 所有规格
- VP9 Profile 0 和 VP9 Profile 2
- JPEG

视频支持的最大分辨率为 8192 x 4320，JPEG 支持的最大分辨率为 16384 x 16384。视频解码（H.264、HEVC、VP9）支持输出 NV12、NV21、Monochrome、I420、P010、I010 格式的数据。视频解码 VP8 仅支持输出 NV12、NV21、Monochrome。JPEG 解码支持输出 NV12、NV21 格式的数据（硬解码不支持 Progressive JPEG）。

软解码 Progressive JPEG 需要依赖 turbo-jpeg 和 libyuv，源码存放于 3rdparty，无需额外安装，编译时打开 CMake 选项 WITH_TURBOJPEG 即可支持。使用方式与硬解码 JPEG 相同。

解码可配置选项:

- 输出数据格式，具体支持格式见上述。
- 输出 buffer 数量，更大的 buffer 数量可以降低解码器因 buffer 不足被阻塞的几率，同时也会增大占用的内存空间。
- 解码器设备号，可指定对应设备解码。
- 解码器输出数据 stride 对齐格式，以适应其他处理流程的要求（如 IPU）。

注解: 解码器仅支持输入完整帧数据进行解码，建议使用 FFmpeg 进行解封装和解析后再送入解码器。

解码流程如下:

1. 设计解码数据回调函数 `DecodeFrameCallback` 和 EOS 回调函数 `DecodeEOSCallback`。
2. 调用 `EasyDecode::New(Attr attr)` 按 attr 参数创建一个解码器实例。
3. 调用 `EasyDecode::FeedData(const CnPacket& packet)` 方法发送数据给解码器解码。
4. 通过设置的解码数据回调函数 `DecodeFrameCallback` 获取解码后的数据，数据使用完毕后调用 `EasyDecode::ReleaseBuffer(uint64_t buf_id)` 通知解码器可以复用这块内存。
5. 在最后一帧发送给解码器后，调用 `EasyDecode::FeedEos()` 告知解码器解码结束。
6. 解码最后一帧结束后，解码器通过设置的 EOS 回调函数 `DecodeEOSCallback` 通知应用程序。
7. 解码结束后，解码器智能指针析构时自动释放解码器资源。

解码中出现不可恢复的错误时，解码器将直接 abort，无法接受输入数据，之后送入数据都返回 false。仍有待处理的解码任务可以重新创建新的解码器实例。

解码示例程序见 samples/common/video_decoder.cpp 。

5.3.2 编码

EasyEncode 封装了 Mlu200 系列和 Mlu300 系列的硬件编码功能。

对于 Mlu200 系列，封装了 CNCodec 中编码的功能，支持视频和图片编码，格式如下:

- HEVC（H.265）Main、Main Still、Main Intra、Main 10 Profile
- H.264 Baseline Profile、Main Profile、High Profile、High 10 Profile
- JPEG

视频支持的最大分辨率为 4096 x 4096，JPEG 支持的最大分辨率为 8192 x 4320。视频编码（H264，HEVC）支持输入 NV12、NV21、I420、RGBA、BGRA、ARGB、ABGR 格式的数据。JPEG 编码支持输入 NV12、NV21 格式的数据。

对于 Mlu300 系列，封装了 CNCodec 中编码的功能，支持视频和图片编码，格式如下:

- HEVC（H.265）: Main Profile、Main Still Profile
- H.264 Baseline Profile、Main Profile、High Profile
- JPEG

视频支持的最大分辨率为 8192 x 8192，JPEG 支持的最大分辨率为 16384 x 16384。视频编码（H264，HEVC）支持输入 NV12、NV21、I420 格式的数据。JPEG 编码支持输入 NV12、NV21 格式的数据。

编码可配置选项:

- 输入数据格式，具体支持格式见上述。
- 编码器设备号，可指定对应设备编码。
- 视频编码支持设置 Profile 和 Level，可配置 GOP length、B 帧数量、GOP 类型、可配置固定的量化参数或量化参数范围进行码率控制。
- JPEG 编码支持配置质量因子控制压缩质量和输出图片大小。

编码流程如下:

1. 设计编码数据回调函数 EncodePacketCallback 和 EOS 回调函数 EncodeEOSCallback。
2. 调用 EasyEncode::New(Attr attr) 按 attr 参数创建一个编码器实例。
3. 如果希望直接使用编码器输入 MLU 内存，调用 RequestFrame(CnFrame* frame) 方法获取编码器分配的输入内存。该内存会存放在 frame->ptrs 变量中。
4. 调用 EasyEncode::SendData(const CnFrame& frame) 方法发送数据给编码器编码。
5. 通过设置的编码数据回调函数 EncodePacketCallback 获取编码后的数据，数据使用完毕后调用 EasyEncode::ReleaseBuffer(uint64_t buf_id) 通知编码器可以复用这块内存。
6. 编码最后一帧结束后，编码器通过设置的 EOS 回调函数 EncodeEOSCallback 通知应用程序。
7. 编码结束后，编码器智能指针析构时自动释放编码器资源。

注解: 输入 CPU 数据时，CnFrame 对象的 device_id 参数必须为负数。输入 MLU 数据时，CnFrame 的

ptrs 必须是通过调用 RequestFrame 方法，从编码器获取的输入内存。

编码中出现不可恢复的错误时，编码器将直接 abort，无法接受输入数据，之后送入数据都返回 false。仍有待处理的编码任务可以重新创建新的编码器实例。

编码示例程序见 tests/src/test_encode.cpp。

5.4 EasyInfer

5.4.1 内存操作

MluMemoryOp 提供 MLU 内存操作的简易封装，支持宿主到设备，设备到宿主，设备到设备的内存拷贝。内存操作有两种接口，一种是不依赖模型的内存操作，仅根据设置的大小操作内存：

```
class MluMemoryOp {
  ...
  // 分配内存
  static void* AllocMlu(size_t nBytes) const;
  // 释放内存
  static void FreeMlu(void *ptr) const;
  // 内存拷贝
  static void MemcpyH2D(void *mlu_dst, void *cpu_src, size_t nBytes)␣
↪const;
  static void MemcpyD2H(void *cpu_dst, void *mlu_src, size_t nBytes)␣
↪const;
  static void MemcpyD2D(void *mlu_dst, void *mlu_src, size_t nBytes)␣
↪const;
  ...
}
```

另一种是依赖模型的内存操作，无需设置内存大小，根据模型的输入输出规模操作内存：

```
class MluMemoryOp {
  ...
  // 设置模型
  void SetModel(std::shared_ptr<ModelLoader> model);

  // 分配内存
  void** AllocCpuInput() const;
  void** AllocCpuOutput() const;
  void** AllocMluInput() const;
```

(下页继续)

(续上页)

```
  void** AllocMluOutput() const;
  // 释放内存
  void FreeCpuInput(void **ptr) const;
  void FreeCpuOutput(void **ptr) const;
  void FreeMluInput(void **ptr) const;
  void FreeMluOutput(void **ptr) const;
  // 内存拷贝
  void MemcpyInputH2D(void **mlu_dst, void **cpu_src) const;
  void MemcpyOutputD2H(void **cpu_dst, void **mlu_src) const;
  ...
}
```

依赖模型的内存操作必须在设置模型后才可以调用，未设置调用将抛出异常。依赖模型的内存拷贝操作还会将模型中固定的设备数据布局转换至用户设置的宿主数据布局（如 NHWC 转 NCHW，float16 转 float32）。

数据布局包含 DataType 和 DimOrder，设置宿主数据布局的方法由 ModelLoader 提供:

```
void ModelLoader::SetCpuInputLayout(DataLayout layout, int data_index);
void ModelLoader::SetCpuOutputLayout(DataLayout layout, int data_index);
```

注解: 宿主指 CPU，设备指 MLU。H2D（Host2Device）代表从宿主拷贝到设备。D2H（Device2Host）代表从设备拷贝到宿主。D2D（Device2Device）代表设备上的内存拷贝。

5.4.2 模型

ModelLoader 提供了易用的模型管理接口，包括模型加载与卸载，模型信息读取等。模型管理采用 RAII 的方式，构造函数加载模型，析构函数卸载模型。支持根据路径从文件中读取模型，和从内存中读取模型两种方式，从内存中读取模型主要用于模型加密的场景，由用户提供解密后的模型数据完成加载。

```
ModelLoader::ModelLoader(const std::string& model_path, const std::string&
↪ function_name);
ModelLoader::ModelLoader(void* mem_ptr, const char* function_name);
```

ModelLoader 在加载模型后会自动读取模型信息并存储，用户可随时获取模型的各种信息，包含输入输出个数，输入输出数据形状等。

5.4.3 推理

EasyInfer 提供了离线模型的推理功能。将 ModelLoader 设置给 EasyInfer 并初始化后即可执行推理任务。推理任务需要提供存储在 MLU 的输入数据和足够存放输出数据的 MLU 内存。使用异步推理任务的结果前需要调用 `MluTaskQueue::Sync` 等待推理任务执行完毕，未同步访问推理结果属于未定义行为。

```
void EasyInfer::Init(std::shared_ptr<ModelLoader> model, int dev_id);
void EasyInfer::Run(void** input, void** output, float* hw_time = nullptr) const;
void EasyInfer::RunAsync(void** input, void** output, MluTaskQueue_t task_queue)␣
↪const;
```

5.5 EasyBang

5.5.1 ResizeConvert 算子

ResizeConvert 是较为通用的图像神经网络前处理，提供 YUV 转 RGB，图像大小缩放，截取 ROI 图像的功能。

- 输入支持 YUV420 2 plane 的 NV12、NV21 格式，输出可指定为 RGBA，BGRA，ARGB，ABGR 格式。
- 输入最大支持宽 7680 像素，缩放最大支持放大 100 倍。同一个 batch 中处理的图像分辨率可以不同。

运行 ResizeConvert 算子需要由任务队列作为设备调度缓存，任务队列可以由用户设置，与其他 MLU 任务共用。用户未设置任务队列时，算子将自动创建新的任务队列。

一般采用 ResizeConvert 算子凑 batch，可以减少内存拷贝，输入是单帧图像，输出是连续内存上的一个 batch，算子直接将输出按偏移写在用户提供的连续内存上。调用过程如下图：

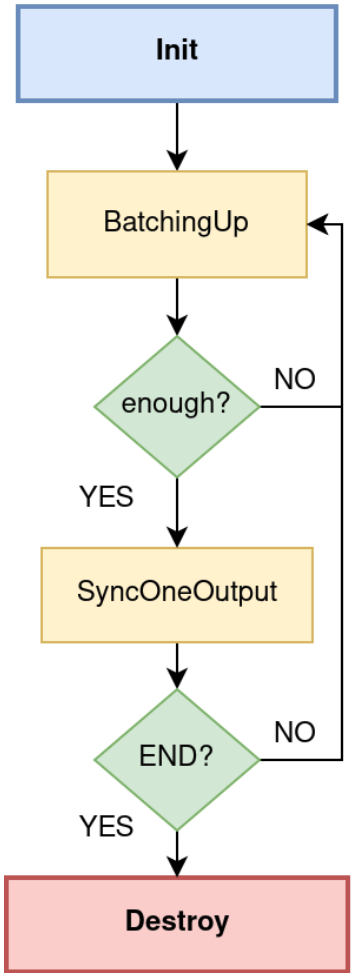

图 1: ResizeConvert 算子调用流程示意图

5.5.2 Resize 算子

Resize 是较为通用的图像处理，仅支持 YUV2YUV 的图像大小缩放，主要用于避免 YUV 系列图像缩放需要先转换至 RGB，缩放后在转换回 YUV。

- 输入支持 YUV420 2 plane 的 NV12、NV21 格式，输出与输入格式相同。
- 输入最大支持宽 7680 像素，缩放最大支持放大 100 倍。一个 Resize 算子实例仅处理一种图像分辨率。

运行 Resize 算子需要由任务队列作为设备调度缓存，任务队列可以由用户设置，与其他 MLU 任务共用。用户未设置任务队列时，算子将自动创建新的任务队列。

一般采用 Resize 算子凑 batch，可以减少内存拷贝，输入是单帧图像，输出是连续内存上的一个 batch，算子直接将输出按偏移写在用户提供的连续内存上。使用方式与 ResizeConvert 算子相同。

5.6 EasyTrack

本版本暂不支持 KCF 追踪算法。

5.6.1 FeatureMatch

FeatureMatchTrack 提供了基于检测结果的多目标追踪功能。输入是出现在某一帧上的全部待追踪检测目标，输出是与输入目标一一对应的追踪目标（DetectObject::detect_id‘标记与之对应的检测目标序号），顺序与输入目标不保证一致，未被确认为 CONFIRMED 的追踪目标 ‘track_id 标为-1。

追踪器对输入的多个检测目标与已知的追踪目标集进行匹配，未匹配到的检测目标将被加入到已知追踪目标集，并标记为 TENTATIVE，处于 TENTATIVE 状态的目标有一帧没有匹配到检测目标就会被删除，连续 n_init 次匹配到就会转为 CONFIRMED 状态，并分配 track_id，CONFIRMED 状态的目标连续 max_age 帧未匹配到检测目标，则认为生命周期结束，被删除。

目标特征由一维 float 向量描述，长度一般是 128，也可以采用其他值，但一个 FeatureMatch 实例中必须使用统一长度的特征组。不同长度的特征向量无法进行相似度运算。本追踪器不提供提取特征的方法，用户可参考 samples/common/feature_extractor.cpp 中提取特征的示例。

注解： 每一次追踪输入视为一次状态更新，只有每一帧都按顺序更新才能保证良好的追踪结果，当前帧无检测目标也需要送空的检测目标数组进行状态更新，跨帧或漏帧将会导致追踪精确度大幅下降。

追踪器可配置选项：

- 特征余弦距离阈值 max_cosine_distance，低于阈值的检测-追踪目标对可以被认为是同一目标。
- 特征集大小 nn_budget，每个缓存的追踪目标最多可保存 nn_budget 组特征，大特征集可稍微提高追踪精度，但也会增大内存占用和运算量。
- IoU 匹配阈值 max_iou_distance，低于阈值的检测-追踪目标对可以被认为是同一目标。
- 已确认追踪目标生命周期 max_age，CONFIRMED 状态的目标连续 max_age 次没有匹配到检测目标则生命周期结束。
- 目标确认次数 n_init，当目标连续 n_init 帧被追踪到时才会被标记为 CONFIRMED。

追踪效果优化方法：

1. 同一物体 track_id 跳变，可能由目标特征值变化大或两帧之间目标位置变化大导致，提升数据源帧率可以有效改善追踪效果，增大特征余弦值距离或 IoU 匹配阈值也可以一定程度上改善追踪效果，但过度增大阈值可能会导致不同物体匹配至同一个目标。max_age 默认值 30，有特征值时一般情况下不会因为偶尔漏检导致同一物体被识别为不同目标。

2. 不同物体匹配为同一个目标，可能由不同目标之间特征值差距过小或多个目标空间距离过近导致，降低特征余弦值距离或 IoU 阈值可以一定程度上改善追踪效果，但过度降低阈值可能会导致目标 track_id 跳变。特征值余弦距离与待追踪的数据集和采用的提取特征算法有关，建议针对待追踪目标数据集进行特征值余弦距离计算验证，设定合适的阈值。
3. 目标相遇时 track_id 交换，一般出现在无特征值输入，仅采用 IoU 匹配的情况下，给检测目标增加特征值可以有效降低 id 交换现象。
4. 物体出现在视界中的时间极短，可能仅有数帧，导致大部分情况下物体处于 TENTATIVE 状态，track_id 都是-1，适当降低 n_init 可以有效改善追踪效果。但是当 n_init 小于 2 时，极有可能出现由检测错误导致生成无效追踪目标，具体取值应根据应用场景和检测精度而定。

6

推理服务

本章描述推理服务。

6.1 介绍

6.1.1 基本概念

本节描述推理服务中所涉及的具体概念。

6.1.1.1 InferServer

其整体架构如下图所示：

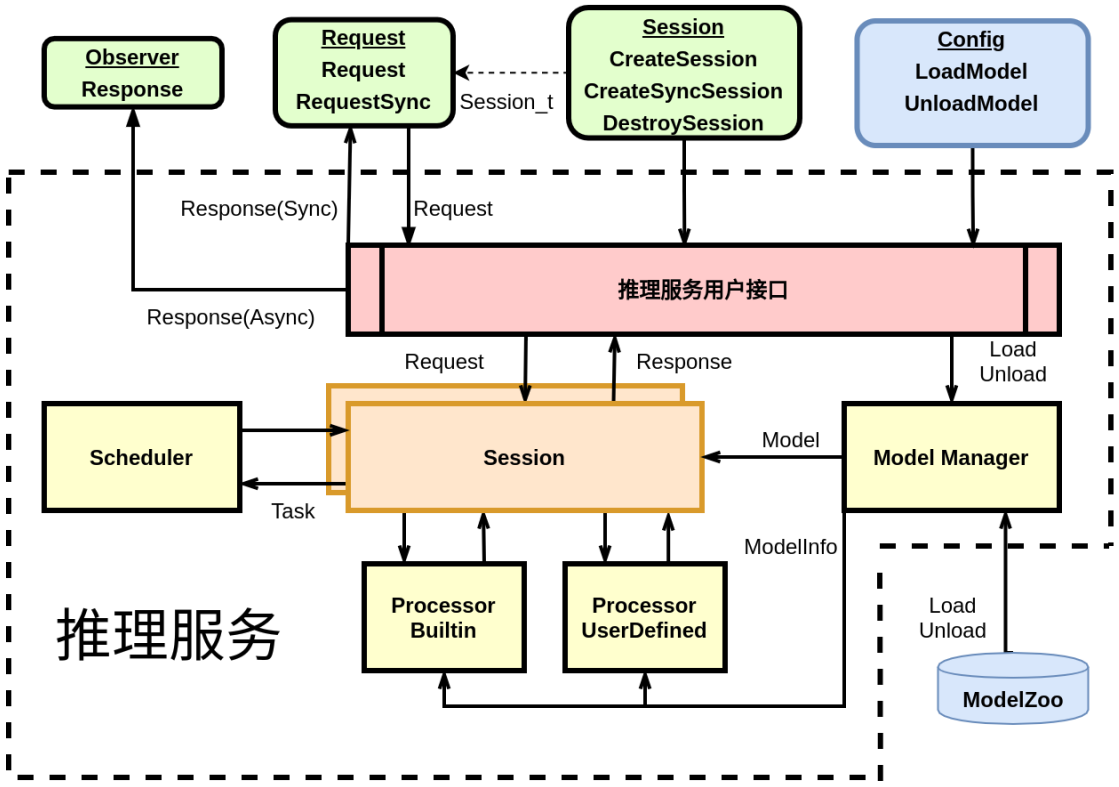

图 1: 推理服务架构

`InferServer` 是推理服务暴露给用户的功能入口，用户通过此入口进行加载或卸载模型（Model）、创建推理节点（Session）、请求推理任务（Request）等操作。推理任务划分为预处理，推理，后处理三个环节，分别交由不同的后端处理单元（Processor）完成。每个推理服务实例维护一个线程池（Scheduler），以处理环节（Task）作为最小调度单元，调度执行在该推理服务实例中运行的所有推理任务。

`InferServer` 使用 pimpl 指针隔离接口与实现，内部保证每个设备上仅有一个 pimpl 实例，同一设备号下创建的 `InferServer` 链接同一个 pimpl 指针，提供对应功能。

```
InferServer s_0(0);
InferServer s_1(1);

// another_s_0 和 s_0 共用同一个任务调度器和相同的推理资源
InferServer another_s_0(0);
```

使用 `InferServer` 异步接口进行推理的步骤如下所示：

1. 加载离线模型 `InferServer::LoadModel()`。
2. 创建异步推理节点 `InferServer::CreateSession()`。
3. 准备输入数据。
4. 提交推理请求 `InferServer::Request()`，推理任务完成后将结果通过 `Observer::Response` 发送给用户。
5. 完成所有推理任务后，释放推理节点 `InferServer::DestroySession()`。

```
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) { ... }
};

bool PreprocFunction(ModelIO*, const InferData&, const ModelInfo*) { ... }

// prepare resources
InferServer server(0);

SessionDesc desc;
desc.name = "sample infer";
desc.model = InferServer::LoadModel(model_path);
// create processors
desc.preproc = PreprocessorHost::Create();
desc.postproc = Postprocessor::Create();
desc.preproc->SetParams("process_function", PreprocFunction);

Session_t session = server.CreateSession(desc, std::make_shared<MyObserver>());

// run inference
// create an input package with tag "stream_0", containing two piece of InferData
auto input = Package::Create(2, "stream_0");
input->data[0].Set<cv::Mat>(image_1);
input->data[1].Set<cv::Mat>(image_2);

server.Request(session, input, nullptr);
// result will be passed to MyObserver::Response after finishing process

// wait until the "stream_0" tagged inference task done
server.WaitTaskDone(session, "stream_0");

// release resources
server.DestroySession(session);
```


6.1.1.2 ModelInfo

ModelInfo 提供了易用的用户侧模型管理和信息读取接口，是各种模型实现的基类。由 InferServer::LoadModel() 加载模型，得到模型基类的智能指针。当所有实例生命周期结束后，模型自动卸载。用户可随时获取模型的各种信息，包含输入输出个数、输入输出数据形状以及 batch_size 等。

6.1.1.3 Session

Session 是推理任务节点的抽象，接收用户的推理请求并完成响应。一个 Session 为一个处理节点，内部的后端处理单元的顺序结构是固定的，处理同一类请求。

使用 InferServer::CreateSession 创建异步 Session，异步 Session 只能使用异步 Request 接口，处理完毕后通过 Observer::Response 发送响应给用户；使用 InferServer::CreateSyncSession 创建同步 Session，同步 Session 只能使用同步 RequestSync 接口，响应作为 RequestSync 的输出参数返回。

```
InferServer s(0);
SessionDesc desc;
/*
 * set desc params
 * ...
 */
Session_t async_session = s.CreateSession(desc, std::make_shared<MyObserver>());
Session_t sync_session = s.CreateSyncSession(desc);

s.Request(async_session, input, user_data);
s.RequestSync(sync_session, input, &status, output);
```

Session 根据传入的参数准备 SessionDesc::engine_num 份推理资源，使每份推理资源可以独立执行任务，达成并行推理。

注解:

模型键值拼接预处理和后处理单元类型名称（modelkey_preproc_postproc）作为 Session 的键值，键值相同的 Session 共用同一簇推理资源。

6.1.1.4 Processor

后端处理单元，负责处理推理任务中的一个环节，由多个 `Processor` 链接起来构成整个推理任务处理流程。

`BaseObject` 基类为 `Processor` 提供设置任意参数的功能。

```
MyProcessor p;
p.SetParams("some int param", 1,
            "some string param", "some string");
int a = p.GetParam<int>("some int param");
const char* b = p.GetParam<const char*>("some string");
```

6.1.1.5 InferData

`InferData` 表示一份推理数据，基于 C++11 编译器实现的 `any` 类使任意类型的推理数据都可以在 `InferData` 中设置。

```
InferData data;
cv::Mat opencv_image;
// 填充数据到InferData
data.Set(opencv_image);
// 获取一份数据的复制
auto image = data.Get<cv::Mat>();
// 获取一份数据的引用
auto& image_ref = data.GetLref<cv::Mat>();
try {
  // 类型不匹配，抛出异常bad_any_cast!
  auto non_sense = data.Get<int>();
} catch (bad_any_cast&) {
  std::cout << "Data stored in any cannot convert to given type!";
}

video::VideoFrame vframe;
// 重新设置data后image_ref非法化
data.Set(vframe);
// cv::imwrite("foo.jpg", image_ref);  // segment fault
auto frame = data.Get<video::VideoFrame>();
auto& frame_ref = data.GetLref<video::VideoFrame>();
```


6.1.1.6 Package

`Package` 是一组推理数据的集合，既是 Request 的输入，也是 Response 的输出。用户通过 Package 可以一次请求多份数据进行处理。

使能 `CNIS_RECORD_PERF` 编译选项时，输出中 `Package::perf` 包含每一个处理环节的性能数据，未使能时为空表。

6.1.1.7 Observer

进行异步请求需要在创建 Session 时，设置 Observer 实例。Session 完成推理任务后，以通知 Observer 的方式完成 Response。

```
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) {
    std::cout << "Get one response\n";
  }
};

InferServer s(0);
SessionDesc desc;
Session_t async_session = s.CreateSession(desc, std::make_shared<MyObserver>());
```

6.1.2 功能

本节详细介绍推理服务的功能。

6.1.2.1 模型加载与管理

推理服务提供模型加载和管理功能，并在全局保有唯一一份模型缓存。

推理服务可以使用两种后端进行推理，CNRT 和 MagicMind，其中 CNRT 仅支持 MLU200 系列平台，MagicMind 仅支持 MLU300 系列平台。两种后端通过编译选项（`CNIS_USE_MAGICMIND`）选择，编译时只能使能一种后端。

当使用 CNRT 后端时，使用 `InferServer::LoadModel(const std::string& pattern1, const std::string& pattern2 = "subnet0")` 加载模型。pattern1 代表模型 URI，pattern2 代表模型中的函数名，一般是 `"subnet0"`；当使用 MagicMind 后端时，使用 `InferServer::LoadModel(const std::string& model_uri, const std::vector<Shape>& in_shapes = {})` model_uri 代表模型模型 URI，当模型输入 shape 可变时，通过 in_shapes 设置模型输入形状。

若使能 CNIS_WITH_CURL 编译选项，上述接口可以从远端下载模型，并加载至模型存储路径（由 InferServer::SetModelDir 设置，默认值为当前目录）。当检测到模型存储路径中已存在同名模型，则跳过下载，直接加载本地模型（请注意不要使用相同的模型名，可能会导致使用错误的模型）。URI 形如 http://some.web.site/foo.graph 或 ../../bar.data 。

当使用 CNRT 后端时，使用 InferServer::LoadModel(void* mem_ptr, const std::string& func_name = "subnet0") 从内存中加载模型，适用于模型加密的场景，由用户对存储的模型解密后交由推理服务进行加载。在使用 MagicMind 后端时，使用 InferServer::LoadModel(void* ptr, size_t size, const std::vector<Shape>& in_shapes = {}) 。

将两个模型相关字符串进行拼接作为键值，对模型进行区分（从内存加载的模型路径是内存地址字符串）。若加载模型时发现缓存中已存在该模型，则直接返回缓存模型。模型缓存存在上限，超出上限自动卸载未在使用的模型。上限默认值是 10，可通过环境变量 CNIS_MODEL_CACHE_LIMIT 更改默认值。支持运行时清除模型缓存，从缓存中清除不会直接卸载模型，仅在无其他模型的智能指针实例时才会卸载模型（确保模型已经没有在使用，避免功能性错误）。

```
#ifdef CNIS_USE_MAGICMIND
// magicmind backend
ModelPtr local_model = InferServer::LoadModel("../../resnet50.model");
ModelPtr net_model = InferServer::LoadModel("http://foo.com/resnet50.model");
// input shape is mutable, set input shape: 4, 416, 416, 3
ModelPtr mutable_input_model = InferServer::LoadModel("../../resnet50.model",
↪{infer_server::Shape({4,416,416,3})});
#else
// cnrt backend
ModelPtr local_model = InferServer::LoadModel("../../resnet50.cambricon", "subnet0
↪");
// use function name "subnet0" as default
ModelPtr local_model = InferServer::LoadModel("../../resnet50.cambricon");
ModelPtr net_model = InferServer::LoadModel("http://bar.com/resnet50.cambricon");
#endif
void *model_mem, *decoded_model_mem;
size_t file_len = ReadFromFile(model_mem, ...);
size_t model_size = DecodeModel(decoded_model_mem, model_mem, file_len, ...);
#ifdef CNIS_USE_MAGICMIND
// magicmind backend
ModelPtr mem_model = InferServer::LoadModel(decoded_model_mem, model_size);
// input shape is mutable. Assume the model has two inputs.
// Set input0 shape: 4, 416, 416, 3 and input2 shape: 4, 256, 256, 3
ModelPtr mutable_input_mem_model =
    InferServer::LoadModel(decoded_model_mem, model_size,
                           {infer_server::Shape({4,416,416,3}), infer_
↪server::Shape({4,256,256,3})});
```

(下页继续)

(续上页)

```
#else
// cnrt backend
ModelPtr mem_model = InferServer::LoadModel(decoded_model_mem);
#endif
```

6.1.2.2 推理任务调度

推理服务对所有请求的推理任务进行调度，以在保证通用性的前提下尽量达到最优性能。推理服务使用三种共同作用的调度方式：批处理、优先级和并行处理。

批处理（Batch）

推理服务提供两种批处理模式，Dynamic 模式（`BatchStrategy::DYNAMIC`）和 Static 模式（`BatchStrategy::STATIC`），在创建 Session 时通过 `SessionDesc::strategy` 指定。

- Dynamic 模式会跨 Request 拼凑批数据，尽可能使用性能最优的批大小进行处理，达到较为理想的吞吐。但由于跨 Request 拼凑数据会存在等待数据集齐的时间，单次 Request 的时延较高。达到设置的 timeout 时间后，即使未集齐批也会进行处理，以避免时延过高。
- Static 模式以用户每次输入的 Request 数据为一批，不跨 Request 拼凑数据，可以达到较为理想的单次 Request 时延。但相较于 Dynamic 模式更难达到性能较优的批大小，总吞吐量较 Dynamic 模式略低。

注解:

目前底层尚未支持带状态的离线模型，待后端支持后，会增量支持 Sequence 模式的批处理策略。

优先级（Priority）

每个设备上的所有推理任务共用同一个调度器。用户可以在创建 Session 时通过 `SessionDesc::priority` 设置优先级，高优先级的 Session 中的任务将会优先被处理。优先级限制为 0~9 的整数，数值越大优先级越高，低于 0 的按 0 处理，高于 9 的按 9 处理。

并行处理（Parallel）

为达到最大性能，通常需要在一个设备上并行执行多组推理任务。若某个 Session 代表的一类推理任务负载较重，可以通过设置 `SessionDesc::engine_num` 增大该类推理任务的并行度，使该 Session 共占用 engine_num * model_core_number 个计算核，和对应份推理资源（内存，模型指令等）。超出上限后，继续增加 engine_num，可能出现由于资源竞争导致的总吞吐下降的情况。

6.1.2.3 后端处理单元（Processor）

推理服务内置三种后端处理单元。

预处理

预处理单元完成输入数据预处理的功能，推理服务内置了通用的预处理单元 Preprocessor 和 PreprocessorHost（在 CPU 侧完成运算）。

预处理单元 Preprocessor

用户提供 Batch 数据预处理的方法 `bool(ModelIO*, const BatchData&, const ModelInfo*)`，通过 `BaseObject::SetParams("process_function", func)` 设置给 `Preprocessor`。由于预处理函数由用户设置，预处理函数的输入是一整个 batch 的数据 `BatchData`，batch 中的每一个数据是可保有任意类型数据的 `InferData`，输出是保有模型的 batch 输入连续 MLU 内存的 `ModelIO` 类型，故支持输入任意类型数据做推理。

注解:

由用户提供的数据预处理方法需要处理一整个 batch 的数据，用户需要自己将预处理结果依次填入模型输入内存中，该内存为 MLU 上的连续内存。

```
InferServer s(0);
SessionDesc desc;
desc.preproc = Preprocessor::Create();
desc.preproc->SetParams("process_function", some_func);
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
```


表 1: 预处理参数表

参数名称	默认值	范围	描述
process_function	nullptr	N/A	用户定义的预处理方法

预处理单元 PreprocessorHost

用户提供单份数据预处理的方法 `bool(ModelIO*, const InferData&, const ModelInfo*)`，通过 `BaseObject::SetParams("process_function", func)` 设置给 `PreprocessorHost`，内置预处理单元内部实现并发对批数据进行任务处理，完成预处理后转换数据摆放（从用户设置的 `SessionDesc::host_input_layout` 转换至模型接受的 layout），拷贝入 MLU，转给推理单元处理。由于预处理函数由用户设置，预处理函数的输入是可保有任意类型数据的 `InferData`，输出是固定类型的 `ModelIO`，故支持输入任意类型数据做推理。

注解:

由用户提供的数据预处理方法仅处理单份数据，假如一个预处理过程（执行一次 Process 方法）的数据包中存在多份数据，多份数据将会被拆分，每份数据分别调用预处理方法并发执行预处理任务。多线程并行执行引入限制：用户需要自己保证预处理方法线程安全。

```
InferServer s(0);
SessionDesc desc;
desc.preproc = PreprocessorHost::Create();
desc.preproc->SetParams("process_function", some_func);
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
```

表 2: 预处理参数表

参数名称	默认值	范围	描述
parallel	4	[1, 16]	处理并行度
process_function	nullptr	N/A	用户定义的预处理方法

推理

推理单元完成推理任务（暂不支持用户设置推理单元，所有 Session 默认使用内置的推理单元）。推理单元接受固定类型的输入，输出固定类型的推理结果（`ModelIO`），输入输出数据均在 MLU 内存上。推理完成后，将推理结果数据转至后处理单元解析。

每个推理单元实例从模型获取一份包含指令和权重数据的 Context。当 engine_num 大于 1 时，同一个模型存在多个推理单元实例，多份 Context 共享部分数据，以避免指令的多份拷贝。

后处理

后处理单元完成推理结果的拷贝和解析，推理服务内置了通用的后处理单元 Postprocessor（在 CPU 侧完成运算）。后处理单元首先将推理结果从 MLU 设备拷贝回 CPU，并转换数据摆放格式（从模型输出的 layout 转换至用户设置的 `SessionDesc::host_output_layout`）。用户提供单份数据后处理的方法 `bool(InferData*, const ModelIO&, const ModelInfo*)`，通过 `BaseObject::SetParams("process_function", func)` 设置给 `Postprocessor`，内置后处理单元内部实现并发，调用用户设置的方法对模型输出的批数据进行解析任务。

注解:

由用户提供的数据后处理方法仅处理单份数据，假如一个后处理过程（执行一次 Process 方法）的数据包中存在多份数据，多份数据将会被拆分，每份数据分别调用后处理方法并发执行后处理任务。多线程并行执行引入限制：用户需要自己保证后处理方法线程安全。

```
InferServer s(0);
SessionDesc desc;
desc.postproc = Postprocessor::Create();
desc.postproc->SetParams("process_function", some_func);
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
```

若用户未设置后处理方法，则仅执行拷贝和转换 layout 操作，后处理单元输出 CPU 上的 `ModelIO` 数据。此种情况下，用户无需在 `SessionDesc` 中设置 postproc，保持默认值 nullptr 将自动创建一个仅执行拷贝操作的 Postprocessor。

很多模型需要额外的数据才能够完成后处理，不同的模型需要的数据不同。`InferData` 中可以设置任意类型的补充数据（`user_data`）供后处理使用。用户在发起 Request 前，通过 `InferData::SetUserData` 设置额外数据：


```
struct Meta {
  int cols, rows;
  ObjTracker* tracker;
};

// create package containing 4 piece of infer data
auto input = Package::Create(4);
for (auto& item : input->data) {
  // read image and set to InferData as inference input
  auto img = GetImage(...);
  item->Set(img);

  // set user data for postprocessor
  Meta meta;
  meta.cols = img.cols;
  meta.rows = img.rows;
  meta.tracker = GetTracker(...);
  item->SetUserData(meta);
}
server->Request(session, input, nullptr);
```

用户在后处理方法中通过 `InferData::GetUserData` 获取该数据：

```
// postprocessor function for one piece of infer data
auto postproc_func = [](InferData* data, const ModelIO& model_out, const␣
↪ModelInfo* model) {
  Meta meta = data->GetUserData<Meta>();
  // 根据额外参数对模型输出后处理
  auto postproc_out = FooPostproc(model_out, meta.cols, meta.rows);
  // 用户侧额外需要的特定逻辑，如给推理目标打上标签，更新追踪器的状态等
  meta.tracker->Update(postproc_out);
  // 设置用户需要在response获取到的输出
  data->Set(postproc_out);
  return true;
};

// 创建session前，设置处理方法给后处理单元
SessionDesc desc;
desc.postproc = Postprocessor::Create();
desc.postproc->SetParams("process_function", postproc_func);
```


表 3: 后处理参数表

参数名称	默认值	范围	描述
parallel	4	[1, 16]	处理并行度
process_function	nullptr	N/A	用户定义的后处理方法

6.1.2.4 扩展接口（contrib）

推理服务提供对图像推理任务的特化接口，简化图像推理过程，其中包括 `VideoFrame` 数据类型，针对该数据类型的 MLU 预处理单元 `PreprocessorMLU`，OpenCV 特化数据类型 `OpencvFrame` ，针对该数据类型的 CPU 预处理仿函数 `OpencvPreproc` ，请求推理任务简化接口的 `VideoInferServer` ，继承自 `InferServer` 。

```
// 一级推理接口
bool Request(Session_t session, const VideoFrame& vframe,
             const std::string& tag, any user_data, int timeout = -1) noexcept;
bool RequestSync(Session_t session, const VideoFrame& vframe, const std::string&
↪tag,
                 Status* status, PackagePtr output, int timeout = -1) noexcept;

// 二级分析接口
bool Request(Session_t session, const VideoFrame& vframe, const std::vector
↪<BoundingBox>& objs,
             const std::string& tag, any user_data, int timeout = -1) noexcept;
bool RequestSync(Session_t session, const VideoFrame& vframe,
                 const std::vector<BoundingBox>& objs, const std::string& tag,
                 Status* status, PackagePtr output, int timeout = -1) noexcept;
```

```
InferServer s(0);
SessionDesc desc;
desc.preproc = PreprocessorHost::Create();
// 使用OpenCV实现的预处理函数
desc.preproc->SetParams("process_function",
↪video::OpencvPreproc::GetFunction(PixelFmt::RGB24));

...

Session_t sync_session = s.CreateSyncSession(desc);
```



```
InferServer s(0);
SessionDesc desc;
// 使用MLU预处理单元
desc.preproc = video::PreprocessorMLU::Create();
// src_format会从VideoFrame中读取，不需要设置
desc.preproc->SetParams("dst_format", video::PixelFmt::RGBA,
                        "preprocess_type", video::PreprocessType::CNCV_PREPROC);

...

Session_t sync_session = s.CreateSyncSession(desc);
```

PreprocessorMLU 单元支持基于 CNCV（寒武纪计算机视觉库），基于 EasyDK 中 EasyBang，基于硬件 scaler（仅 MLU220 支持）的预处理加速。

参数 preprocess_type 选择 CNCV_PREPROC，指定使用 CNCV 的 ResizeConvert 算子和 MeanStd 算子进行预处理。支持 YUV 转至 RGB 家族（3 通道和 4 通道共六种格式），同时图像缩放至指定大小，然后根据用户设置的参数依次进行归一化（除以 255.f），减均值，除方差等处理。当 mean 和 std 未设置或设置为空，且不需要归一化时，预处理输出 uint8 类型的数据；否则，预处理输出模型指定输入类型的数据。

参数 preprocess_type 选择 SCALER，指定使用 Scaler 硬件进行预处理（仅 MLU220 支持）。支持 YUV 转至 RGBA 家族（4 通道共四种格式），同时图像缩放至指定大小。仅支持输出 uint8 类型的数据。

参数 preprocess_type 选择 RESIZE_CONVERT，指定使用 EasyDK 中的 ResizeConvert 算子进行预处理（仅 CNRT 后端支持）。支持 YUV 转至 RGBA 家族（4 通道共四种格式），同时图像缩放至指定大小。仅支持输出 uint8 类型的数据。由于 EasyDK 中的 Bang 算子不再维护，这种预处理方式已经废弃，建议使用功能更全面的 CNCV_PREPROC。

表 4: 扩展预处理参数表

参数名称	默认值	范围	描述
dst_format	N/A	RGB family	dst 图像格式
prepro-cess_type	N/A	CNCV_PREPROC, SCALER, RE-SIZE_CONVERT	预处理后端
mean	{}	std::vector<float>, size = 0 / 3 / 4	分通道均值
std	{}	std::vector<float>, size = 0 / 3 / 4	分通道方差
normalize	false	true、false	数据是否归一化（除以 255.f）
keep_aspect_ratio	false	true、false	图片缩放是否保持宽高比
pad_value	0	0 ~ 255	保持宽高比时 pad 填充值

注解:

参数无默认值即用户必须设置

当 normalize，mean 和 std 同时设置时，先对像素值进行归一化（除以 255.f），再对归一化后的像素值进行减均值除方差（(pixel - mean) / std）。当 mean 为空时，mean = {0, 0, ⋯, 0}；当 std 为空时，std = {1, 1, ⋯, 1}。

6.2 编程指南

6.2.1 同步或异步请求

请求任务（Request）有同步接口和异步接口两种选择。

```
bool Request(Session_t session, PackagePtr input, any user_data, int timeout = -1)␣
↪noexcept;
bool RequestSync(Session_t session, PackagePtr input,
                 Status* status, PackagePtr output, int timeout = -1) noexcept;
```

异步请求中保留了 `user_data` 字段，支持用户设置任意数据，推理任务中不会对该字段进行处理，任务完成后将 Response 返回给用户，完成透传。

为避免数据处理能力不一致导致内存堆积，推理服务对处理任务缓存进行了限制，缓存超出一定数量则进入繁忙状态，Request 和 RequestSync 接口会被阻塞，直至推理服务空闲后再将请求的数据推入队列中。因此同步和异步接口都可以选择设置 timeout，同步和异步接口在等待超时后直接退出，不进行请求。请求后，同步接口阻塞直至得到 Response，若等待过程总时长超过 timeout 设置的值，则将该份 Request 数据标记为 discarded，直接退出。

当 Session 不属于该 InferServer，或等待入队列超时，未发送请求，则接口返回 false，其他情况返回 true。若同步接口发送请求后，等待处理完成超时，则接口返回 true，status 值为 `Status::TIMEOUT`。

`Package::data` 是一个 vector，其中的每一个元素代表一份数据。每次请求可以包含一份数据，或者多份数据（`Package::data.size()`），但请求-响应对与数据份数无关，始终是一份响应对应一份请求。换言之，对异步接口调用时，`Observer::Response` 被调用次数与请求次数一致，且数据一一对应。

异步请求收到响应的顺序与发送请求的顺序保持一致（如请求顺序为 req_0、req_1、req_2, 则响应顺序一定是 resp_0、resp_1、resp_2），以避免有序数据出现紊乱。

6.2.2 等待处理结束或遗弃任务

在某些数据处理正常结束时，如流数据到达 EOS（End Of Stream），如果需等待相同类型数据全部完成后再退出处理或销毁 Session，需在请求时输入 Package 设置 tag（`std::string`），调用 `InferServer::WaitTaskDone(session, tag)`，这样就会在函数中等待数据处理完成，直到最后一个持有该 tag 的请求处理完毕后退出函数，若无持有该 tag 的请求则立即返回。

如需快速停止某些数据处理，遗弃未处理数据，需在请求时输入 Package 设置 tag（`std::string`），在需要快速停止时调用 `InferServer::DiscardTask(session, tag)`，该函数不阻塞，仅将所有持有该 tag 的请求标记为 discarded，然后返回。标记为 discarded 的数据，若在等待处理的缓存中，则直接抛弃，若已在处理，则等待处理完毕后废弃。所有被标记为 discarded 的数据都不会返回给用户。

6.2.3 自定义数据结构

用户可自定义任意数据结构提交给推理服务进行推理，但用户需提供预处理自定义数据结构至模型输入 `ModelIO` 的函数，如下所示：

```
struct FooStruct {
  ...
};

bool Preproc(ModelIO* out, const FooStruct& in, const ModelInfo* model);
bool Bar(ModelIO* out, const InferData& in, const ModelInfo* model) {
  const FooStruct& foo = in.GetLref<FooStruct>();
  return Preproc(out, foo, model);
```

(下页继续)

(续上页)

```
}

InferServer s(0);
SessionDesc desc;
desc.preproc = PreprocessorHost::Create();
desc.preproc->SetParams("process_func", Bar);
...
Session_t session = s.CreateSession(desc, std::make_shared<MyObserver>());

FooStruct foo;
...
PackagePtr pack = Package::Create(1);
pack->data[0]->Set(foo);
s.Request(session, pack, nullptr);
```

6.2.4 自定义后端处理单元

用户可自定义后端处理单元，以满足定制化需求。自定义的后端处理单元 `MyProcessor` 需继承自 `ProcessorForkable<MyProcessor>` （CRTP）以获得 `Create` 和 `Fork` 的能力，并重写 `Init` 和 `Process` 两个纯虚函数。Session 调用虚函数 Fork 复制 Processor 实例，用来并行处理推理任务。参数拷贝自原有 Processor，并调用 `Init` 来初始化。自定义后端处理单元对纯虚函数 Init 的重写需要实现解析参数并初始化推理资源，对纯虚函数 Process 的重写需要实现任务数据的处理。

```
class MyProcessor : public ProcessorForkable<MyProcessor> {
 public:
  MyProcessor() noexcept : ProcessorForkable<MyProcessor>("MyProcessor") { ... }
  Status Init() noexcept override { ... }
  Status Process() noexcept override { ... }
}
```

自定义 Processor 的实现应满足：

- 传递给 `ProcessorForkable` 基类的构造函数一个唯一的类型名称（用于日志打印，区分 Executor，性能数据表键值）。
- `Processor::Init` 函数解析所有参数，并准备好推理资源。
- `Processor::Process` 函数在 Init 后可直接调用，实现无需线程安全，交由调用方（内部友元 `TaskNode`）通过私有方法 `Processor::Lock` 锁住处理资源。

6.2.5 特殊用法

存在多个不同模型，但是输入数据和预处理相同，并行推理的特殊情况，仅执行一次预处理可以极大节省算力开销，用户可以不通过框架单独调用 `Preprocessor`，对数据进行预处理，预处理后直接送入不同的 Session 进行处理（`SessionDesc::preproc` 需设置为 `PreprocessorHost`，且不设置 process_function，将直接跳过 `PreprocessorHost`，由于属于特殊用法，故不支持不设置 preproc 的情况下默认创建空的 `PreprocessorHost`）。

预处理后数据存储在连续的一片内存地址上（不能保证多份预处理的结果在连续的内存地址上，无法凑批处理，故仅支持 Static 模式），设置给 `Package::predict_io` 即可，同时创建数据份数的 data，告知连续数据中共有多少份数据，便于后处理时分解成独立的数据包。使用方式如下所示：

```
ModelIO model_input;
// do preproc in user function, such as:
// Preproc(std::vector<...> input_data, &model_input);
auto pack = Package::Create(data_num);
pack->predict_io.reset(new InferData);
pack->predict_io->Set(model_input);
server->Request(session, pack, user_data);
```

存在模型输入规模过大的情况，使用 MLU 上的 BANG 算子实现后处理可减少拷贝，降低带宽压力。现有 Postprocessor 先拷贝再调用用户设置的后处理接口的逻辑不满足需求，用户可自行实现继承自 ProcessorForkable 的 postproc 模块，接收 Predictor 输出的 MLU 连续数据，后处理后拷贝至 CPU 再自行解批处理。

6.2.6 性能调优方法

仅在打开编译选项 `CNIS_RECORD_PERF` 时进行性能统计，未使能时 `InferServer::GetPerformance` 接口返回空表。使能 `CNIS_RECORD_PERF` 后创建 Session 时设置 `SessionDesc::show_perf` 为 true，则每隔 2 秒自动打印 Session 的性能数据。

1. 每个 Processor 时延计时。
2. 锁等待计时。
3. 批处理等待时间计时，每份批处理数据数量记录。
4. 单次 Request 时延计时，整体吞吐计数。

通过 `std::map<string, LatencyStatistic> GetPerformance(Session_t session)` 获取时延性能信息，其中 `LatencyStatistic` 如下：

```
struct LatencyStatistic {
  // 由于每个批处理中数据量可能不一致，以单份数据为单元计数
  uint32_t unit_cnt;  // 数据单元总数
```

(下页继续)

(续上页)

```
  uint64_t total_time;  // 总时延
  uint32_t max_time;  // 单元最大时延
  uint32_t min_time;  // 单元最小时延
};
```

通过 `ThroughoutStatistic GetThroughout(Session_t session)` 获取吞吐性能信息，其中 `ThroughoutStatistic` 如下：

```
struct ThroughoutStatistic {
  uint32_t request_cnt{0};  // 请求总数
  uint32_t unit_cnt{0};  // 数据单元总数
  float rps{0};  // 每秒请求数
  float ups{0};  // 每秒数据单元数
  float rps_rt{0};  // 实时每秒请求数
  float ups_rt{0};  // 实时每秒数据单元数
};
```

影响性能的参数有如下三个：

1. `BatchStrategy`：DYNAMIC 模式总吞吐较高，单个 Request 时延较长；Static 模式单个 Request 时延较短，总吞吐较低。
2. `SessionDesc::engine_num`：engine_num == card core number / model core number 时可达到最大吞吐，继续增大 engine_num 吞吐基本无提升，或可能下降。engine_num 越大占用的 MLU 内存越多，并行处理能力越强。
3. `SessionDesc::batch_timeout`：仅在 Dynamic 模式下生效，一般应调整至大多数情况都能凑齐一组批处理的时长，然后根据要求的吞吐和时延上下微调。

根据数据调优性能的建议如下：

- 若要降低单帧数据总时延，可以选择使用 Static 策略，或者使用 batchsize 更小的离线模型。
- 若要提高总吞吐量，可以选择使用 Dynamic 策略，或者使用 batchsize 更大的模型。
- 批处理等待时间过长可能表示 timeout 设置时间过长，模型的 batchsize 过大。
- 批处理中数据达不到模型的 batchsize 值，可能表示 timeout 设置时间过短，或模型的 batchsize 过大。
- 单个 Processor 处理时间过长表示可能遇到性能瓶颈，资源占用达到上限（算力、带宽、CPU），或者处理并行度较低，在 engine_num * model_core_number 小于 MLU 单个设备核数的时候，可以适当调大 engine_num，以加快处理速度。
- MagicMind 后端模型默认占用所有 MLU 设备核，engine_num 选 1 时延较低，选 2 吞吐更高，设置更大的 engine_num 没有性能收益。
- 使用 cnmon 获取到 MLU 占用率较低的情况，可能是任务不饱和，确认 engine_num 足够大。若使用的是 CPU 侧的预处理，可以考虑更换成 MLU 上的预处理以加快处理速度，提高 MLU 占用率。

- 为保证 Response 顺序与 Request 顺序一致，`Observer::Response` 对单个 Session 的 Request 在时序上是严格串行执行的，若 `Response` 函数耗时 10ms，则相当于限制该 Session 处理速率上限是 100 个请求/秒。降低 `Response` 耗时或将推理任务分给多个不同的 Session 能够有效地降低该限制，提升性能（使用相同模型的多个 Session 使用同一套推理资源，不会有额外的资源开销）。

6.3 python 封装

InferServer 基于 pybind11 封装了 Python 接口，帮助用户通过 Python 编程语言快速开发实际业务。目前该套接口仅支持 Python 3 以上版本调用。

6.3.1 使用说明

EasyDK 仓库中默认没有编译和安装 InferServer Python 包，通过执行以下命令即完成 InferServer Python 包的安装。

```
cd {EASYDK_DIR}/infer_server/python
python setup.py install
```

`${EASYDK_DIR}` 代表 EasyDK 源码目录。

卸载 InferServer Python 包：

```
pip uninstall cnis -y
```

除此之外，可以通过使能编译选项 `CNIS_WITH_PYTHON_API` 编译 Python API 库。

```
cd {EASYDK_DIR}/;
mkdir build; cd build;
# For mlu200
cmake .. -DCNIS_WITH_PYTHON_API=ON
# For mlu300
# cmake .. -DCNIS_WITH_PYTHON_API=ON -DCNIS_USE_MAGICMIND=ON
make -j4
```

注解： 如果此前通过 setup.py 安装过 InferServer Python 包，需要将其卸载后再进行编译。否则默认使用安装的 cnis 包。

编译及执行 InferServer Python 接口单元测试：

通过执行以下命令，编译单元测试，

```
cd {EASYDK_DIR}/;
mkdir build; cd build;
# For mlu200
cmake .. -DCNIS_WITH_PYTHON_API=ON -DBUILD_TESTS=ON
# For mlu300
# cmake .. -DCNIS_WITH_PYTHON_API=ON -DCNIS_USE_MAGICMIND=ON -DBUILD_TESTS=ON
```

执行单元测试依赖于 pytest，Numpy 和 OpenCV，安装依赖：

```
pip install -U pytest
cd {EASYDK_DIR}/infer_server/python
pip install -r requirements.txt
```

注解： pytest 要求 Python 版本高于 3.6。

另外，提供三个示例应用：

示例依赖于 Numpy 和 OpenCV，通过执行以下命令安装依赖：

```
cd {EASYDK_DIR}/infer_server/python
pip install -r requirements.txt
```

- cnis_sync_demo.py，该应用演示了如何创建推理服务并请求同步推理。前处理使用 OpenCV，后处理使用 Python 侧自定义的后处理。
- cnis_async_demo.py，该应用演示了如何创建推理服务并请求异步推理。前处理使用 CNCV（寒武纪计算机视觉库，详情参考《寒武纪计算机视觉库用户手册》）算子，后处理使用 C++ 侧自定义的后处理。
- cnis_infer_only_demo.py，该应用演示了使用推理服务进行推理，前后处理在业务层 Python 代码中进行。

注解： 假如碰到未能找到 cnis 动态库错误，则需要将 Python 虚拟环境中 cnis 动态库路径添加到 `LD_LIBRARY_PATH`。

6.3.2 编程模型

6.3.2.1 同步推理

1. 创建 InferServer

```
# Create InferServer.
infer_server = InferServer(dev_id=0)
```

2. 创建 SessionDesc，通过该对象描述 Session

```
# Create SessionDesc.
session_desc = SessionDesc()
session_desc.name = "test_session_sync"
session_desc.engine_num = 1
session_desc.strategy = BatchStrategy.STATIC

# Load model (ssd mlu270 model).
model_path = "http://video.cambricon.com/models/MLU270/Primary_Detector/ssd/vgg16_
↪ssd_b4c4_bgra_mlu270.cambricon"
session_desc.model = infer_server.load_model(model_path)

# Set preprocessing.
# Use Opencv.
session_desc.preproc = PreprocessorHost()
session_desc.set_preproc_func(OpencvPreproc(dst_fmt=VideoPixelFmt.BGRA, keep_
↪aspect_ratio=False).execute)

# Set postprocessing.
# Custom postprocessing.
class CustomPostprocess(Postprocess, threshold):
  def __init__(self):
    super().__init__()
    self.threshold = threshold
  def execute_func(self, result, model_output, model_info):
    # Do postprocessing.
    return True

session_desc.postproc = Postprocessor()
session_desc.set_postproc_func(CustomPostprocess(0.6).execute)
```

3. 创建同步 Session


```
session = infer_server.create_sync_session(session_desc)
```

4. 准备输入数据

由于我们使用 OpencvPreproc 作为前处理，因此我们的输入数据类型必须为 OpencvFrame。

```
# Create a video_frame.
import cv2
cv_frame = OpencvFrame()
img = cv2.imread("your_image.jpg")
cv_frame.img = img
cv_frame.fmt = VideoPixelFmt.BGR24

# Create package with one frame and set cv_frame to it.
tag = "stream_0"
input_pak = Package(1, tag)
input_pak.data[0].set(cv_frame)
```

5. 准备输出数据

```
output_pak = Package(1)
```

6. 请求同步推理

```
status = Status.SUCCESS
ret = infer_server.request_sync(session, input_pak, status, output_pak,␣
↪timeout=20000)
```

7. 销毁 Session

```
# Destroy Session.
infer_server.destroy_session(session)
```

6.3.2.2 异步推理

1. 创建 InferServer

```
# Create InferServer.
infer_server = InferServer(dev_id=0)
```

2. 创建 SessionDesc，通过该对象描述 Session


```
# Create SessionDesc.
session_desc = SessionDesc()
session_desc.name = "test_session_async"
session_desc.engine_num = 1
session_desc.strategy = BatchStrategy.DYNAMIC

# Load model (yolov3 mlu270 model).
model_path = "http://video.cambricon.com/models/MLU270/yolov3_b4c4_argb_mlu270.
↪cambricon"
session_desc.model = infer_server.load_model(model_path)

# Set preprocessing.
# Use CNCV MLU ResizeConvert operator.
session_desc.preproc = VideoPreprocessorMLU()
session_desc.set_preproc_params(VideoPixelFmt.ARGB, VideoPreprocessType.CNCV_
↪PREPROC, keep_aspect_ratio=True)

# Set postprocessing.
# The custom postprocessing PostprocYolov3 is written in C++ code as an sample.
session_desc.postproc = Postprocessor()
session_desc.set_postproc_func(PostprocYolov3(0.5).execute)
```

3. 定义一个观察者，用于接收输出数据

```
class MyObserver(Observer):
  def __init(self):
    super().__init__()
  def response_func(self, status, data, user_data):
    # Process outputs here.

obs = MyObserver()
```

4. 创建异步 Session

```
session = infer_server.create_session(session_desc, obs)
```

5. 准备输入数据

由于我们使用 VideoPreprocessorMLU 前处理，因此我们的输入数据类型必须为 VideoFrame，并且在 VideoFrame 中填入 MLU 数据帧。

```
# Create package with one frame and set video_frame to it.
```

(下页继续)

(续上页)

```
# Create a video_frame.
video_frame = VideoFrame()
video_frame.plane_num = 2
video_frame.format = VideoPixelFmt.NV12
video_frame.width = 1280
video_frame.height = 720
video_frame.stride = [video_frame.width, video_frame.width]
# Create an input image.
import numpy as np
img_y = np.zeros(video_frame.width * video_frame.height)
img_uv = np.zeros(video_frame.width * video_frame.height / 2)
# Create mlu buffer and copy image to it.
mlu_buffer_y = Buffer(video_frame.width * video_frame.height, 0)
mlu_buffer_uv = Buffer(int(video_frame.width * video_frame.height / 2), 0)
mlu_buffer_y.copy_from(img_y)
mlu_buffer_uv.copy_from(img_uv)
# Set buffer to video_frame.
video_frame.set_plane(0, mlu_buffer_y)
video_frame.set_plane(1, mlu_buffer_uv)

tag = "stream_0"
input_pak = Package(1, tag)
# We could set user data of the input package if it is needed by the custom␣
↪postproc.
# In PostprocYolov3, image_width and image_height are needed to calculate scaling␣
↪ratio.
input_pak.data[0].set_user_data({"image_width": w, "image_height": h})
input_pak.data[0].set(video_frame)
```

6. 请求异步推理

```
ret = infer_server.request(session, input_pak, {"user_data":"test"}, timeout=20000)
```

7. 等待所有任务结束后，销毁 Session

```
# Wait all tasks done.
infer_server.wait_task_done(session, tag)
# Destroy Session.
infer_server.destroy_session(session)
```


6.3.2.3 仅推理

支持使用默认前后处理，InferServer 内部仅进行推理。

- 不设置 SessionDesc::preproc 时，使用默认前处理。将输入数据拷贝至 InferServer 指定的神经网络输入内存中。
- 不设置 SessionDesc::postproc 时，使用默认后处理。返回神经网络输出的原始数据。
- 输入数据为经过预处理后，符合神经网络输入的数据，数据类型必须为 numpy array。

```
# Prepare input data.
img = cv2.imread("your_image.jpg")
resized_img = cv2.resize(img, (width, height))
bgra_img = cv2.cvtColor(resized_img, cv2.COLOR_BGR2BGRA)

tag = "stream_0"
input_pak = Package(1, tag)
input_pak.data[0].set(bgra_img)
```

- 输出数据为神经网络输出，用户需要在业务层 Python 代码完成后处理。

```
# Get output data.
for data in output_pak.data:
  model_io = data.get_model_io()
  buffer = model_io.buffers[0]
  data = buffer.data(model_io.shapes[0], session_desc.model.output_layout(0))
```

6.3.3 自定义前后处理

InferServer 内置了 SSD 和 Yolov3 神经网络的后处理代码。同时也支持 Python 语言创建自定义前后处理逻辑并关联至 Session 中。虽然 InferServer 框架为了提高处理效率一直并行调度前后处理，但是由于 Python GIL(全局解释器锁) 的存在，前后处理的 python 代码段都会被串行执行，大大降低整体的处理效率。因此对性能敏感场景请谨慎使用该功能，建议 1. 使用 C++ 编写神经网络自定义前后处理代码；2. 在业务层 Python 代码完成预处理，然后喂给推理服务，推理服务则返回推理后原始数据。

1. 首先继承前后处理的基类。在 execute_func 函数中实现前后处理逻辑。处理成功后 execute_func 函数需要返回 True，但如果处理失败则需要返回 False。

```
class CustomPreprocess(Preprocess):
  def __init__(self):
    super().__init__()


  def execute_func(self, model_input, input_data, model):
```

(下页继续)

(续上页)

```
    # Do preprocessing.
    return True

class CustomPostprocess(Postprocess):
  def __init__(self):
    super().__init__()

  def execute_func(self, result, model_output, model):
    # Doing postprocessing.
    return True
```

2. 将前后处理基类的 execute 函数设置给 SessionDesc。execute 函数会调用自定义的前后处理子类的 execute_func 函数。

```
session_desc.preproc = PreprocessorHost()
session_desc.set_preproc_func(CustomPreprocess().execute)

session_desc.postproc = Postprocessor()
session_desc.set_postproc_func(CustomPostprocess().execute)
```


7

示例代码说明

7.1 深度学习应用示例

```
#include <math.h>
#include <string.h>

#include <cstdlib>
#include <atomic>
#include <future>
#include <iostream>
#include <queue>

#include "device/mlu_context.h"
#include "cnis/contrib/video_helper.h"
#include "cnis/infer_server.h"
#include "cnis/processor.h"
#include "easycodec/easy_decode.h"
#include "postprocess/postproc.h"

std::unique_ptr<edk::EasyDecode> g_decoder;
std::queue<edk::CnFrame> g_frames;
static std::mutex g_mut;
static std::condition_variable g_cond;
```

(下页继续)

(续上页)

```
static std::atomic<bool> g_receive_eos{false};
static std::atomic<bool> g_running{true};

static std::unique_ptr<infer_server::InferServer> g_infer_server;
static infer_server::Session_t g_session;

constexpr const char *gmodel_path_220 =
    "http://video.cambricon.com/models/MLU220/resnet18_b4c4_bgra_mlu220.cambricon";
constexpr const char *gmodel_path_270 =
    "http://video.cambricon.com/models/MLU270/resnet50_b16c16_bgra_mlu270.cambricon
↪";
constexpr const char *gmodel_path_370 =
    "http://video.cambricon.com/models/MLU370/resnet50_nhwc_tfu_0.8.2_uint8_int8_
↪fp16.model";

// The decode frame callback
void OnDecodeFrame(const edk::CnFrame &info) {
  std::unique_lock<std::mutex> lk(g_mut);
  g_frames.push(info);
  g_cond.notify_one();
}

// The eos frame callback
void OnEos() {
  std::unique_lock<std::mutex> lk(g_mut);
  g_receive_eos = true;
  g_cond.notify_one();
}

bool DecodeLoop() {
  // Init decoder
  edk::EasyDecode::Attr attr;
  attr.frame_geometry.w = 1920;
  attr.frame_geometry.h = 1080;
  // choose codec type from H264/H265/JPEG
  attr.codec_type = edk::CodecType::JPEG;
  attr.pixel_format = edk::PixelFmt::NV12;
  attr.dev_id = 0;
  attr.frame_callback = OnDecodeFrame;
  attr.eos_callback = OnEos;
  attr.silent = false;
```

(下页继续)

(续上页)

```
  attr.input_buffer_num = 6;
  attr.output_buffer_num = 6;
  g_decoder = edk::EasyDecode::New(attr);

  // For example, read a jpeg file
  std::string img_path = "your_image.jpg";
  FILE *fp = fopen(img_path.c_str(), "rb");
  if (!fp) {
    std::cerr << "img_path not exist, path: " << img_path << "\n";
    return false;
  }
  fseek(fp, 0, SEEK_END);
  uint64_t len = ftell(fp);
  rewind(fp);
  void* data = malloc(len);
  memset(data, 0, len);
  fread(data, 1, len ,fp);
  fclose(fp);

  // Decode loop
  uint64_t frame_index = 0;
  uint64_t frame_cnt = 10;
  while ((frame_cnt--) > 0 && g_running) {
    edk::CnPacket pkt;
    pkt.data = data;
    pkt.length = len;
    pkt.pts = frame_index++;

    // Feed data to codec
    bool ret = g_decoder->FeedData(pkt);
    if (!ret) { return false; }
  }
  free(data);
  // Feed eos (end of stream)
  g_decoder->FeedEos();
  return true;
}

bool InitInfer() {
  g_infer_server.reset(new infer_server::InferServer(0));
  infer_server::SessionDesc desc;
```

(下页继续)

(续上页)

```
  desc.strategy = infer_server::BatchStrategy::STATIC;
  desc.engine_num = 1;
  desc.priority = 0;
  desc.show_perf = true;
  desc.name = "infer session";

  edk::MluContext context;
  context.SetDeviceId(0);
  context.BindDevice();
  auto version = context.GetCoreVersion();
  std::string func_name = "subnet0";
  std::string model_path;
  if (version == edk::CoreVersion::MLU220) {
    model_path = gmodel_path_220;
  } else if (version == edk::CoreVersion::MLU270) {
    model_path = gmodel_path_270;
  } else if (version == edk::CoreVersion::MLU370) {
    model_path = gmodel_path_370;
  } else {
    std::cerr << "Unsupported core version: "
              << static_cast<int>(version) << std::endl;
    return false;
  }

  // Load offline model
  #ifdef CNIS_USE_MAGICMIND
  desc.model = infer_server::InferServer::LoadModel(model_path);
  #else
  desc.model = infer_server::InferServer::LoadModel(model_path, func_name);
  #endif
  // set preproc and postproc
  desc.preproc = infer_server::video::PreprocessorMLU::Create();
  desc.postproc = infer_server::Postprocessor::Create();

  // Use CNCV preproc
  #ifdef CNIS_USE_MAGICMIND
  desc.preproc->SetParams("preprocess_type", infer_
↪server::video::PreprocessType::CNCV_PREPROC,
      "src_format", infer_server::video::PixelFmt::NV12,
      "dst_format", infer_server::video::PixelFmt::RGB24);
```

(下页继续)

(续上页)

```
  #else
  desc.preproc->SetParams(
      "preprocess_type", infer_server::video::PreprocessType::CNCV_PREPROC,
      "src_format", infer_server::video::PixelFmt::NV12,
      "dst_format", infer_server::video::PixelFmt::BGRA);
  #endif
  // PostprocClassification is declared at samples/common/postprocess/postproc.h
  desc.postproc->SetParams("process_function",
      infer_server::Postprocessor::ProcessFunction(PostprocClassification(0.2)));

  g_session = g_infer_server->CreateSyncSession(desc);
  if (g_session) {
    return true;
  } else {
    return false;
  }
}

bool Process(edk::CnFrame frame) {
  // Prepare input
  infer_server::video::VideoFrame vframe;
  vframe.plane_num = frame.n_planes;
  vframe.format = infer_server::video::PixelFmt::NV12;
  vframe.width = frame.width;
  vframe.height = frame.height;

  for (size_t plane_idx = 0; plane_idx < vframe.plane_num; ++plane_idx) {
    vframe.stride[plane_idx] = frame.strides[plane_idx];
    uint32_t plane_bytes = vframe.height * vframe.stride[plane_idx];
    if (plane_idx == 1) plane_bytes = std::ceil(1.0 * plane_bytes / 2);
    infer_server::Buffer mlu_buffer(frame.ptrs[plane_idx], plane_bytes, nullptr,␣
↪0);
    vframe.plane[plane_idx] = mlu_buffer;
  }
  infer_server::PackagePtr in = infer_server::Package::Create(1);
  in->data[0]->Set(std::move(vframe));
  infer_server::PackagePtr out = infer_server::Package::Create(1);
  infer_server::Status status = infer_server::Status::SUCCESS;
  bool ret = g_infer_server->RequestSync(g_session, std::move(in), &status, out);
  if (!ret || status != infer_server::Status::SUCCESS) {
    g_decoder->ReleaseBuffer(frame.buf_id);
```

(下页继续)

(续上页)

```
    std::cerr << "Request sending data to infer server failed. Status:"
              << static_cast<int>(status) << std::endl;
    return false;
  }

  // Release codec buffer
  g_decoder->ReleaseBuffer(frame.buf_id);

  // print result
  const std::vector<DetectObject>& postproc_results =
      out->data[0]->GetLref<std::vector<DetectObject>>();
  std::cout << "----- Classification Result:\n";
  int show_number = 2;
  for (auto& obj : postproc_results) {
    std::cout << "[Object] label: " << obj.label << " score: " << obj.score << "\n
↪";
    if (!(--show_number)) break;
  }
  std::cout << "-----------------------------------\n" << std::endl;
  return true;
}

bool RunLoop() {
  while (true) {
    std::unique_lock<std::mutex> lk(g_mut);
    g_cond.wait(lk, []() { return !g_frames.empty() || !g_running; });
    if (!g_running) return false;
    edk::CnFrame frame = g_frames.front();
    g_frames.pop();
    lk.unlock();

    if (!Process(std::move(frame))) return false;

    lk.lock();
    if (g_frames.size() == 0 && g_receive_eos.load()) { break; }
    lk.unlock();
  }
  return true;
}

void Destroy() {
```

(下页继续)

(续上页)

```
  if (g_decoder) {
    g_decoder.reset();
  }
  if (g_infer_server && g_session) {
    g_infer_server->DestroySession(g_session);
    g_infer_server.reset();
  }
}

int main() {
  bool ret = true;
  // Init InferServer
  if (!InitInfer()) {
    std::cout << "InitInfer failed!!" << std::endl;
    return -1;
  }

  // Start a thread to do inference
  std::future<bool> process_loop_return = std::async(std::launch::async, RunLoop);

  // Set mlu environment
  edk::MluContext context;
  context.SetDeviceId(0);
  context.BindDevice();

  // Decode loop
  ret = DecodeLoop();
  if (!ret) {
    std::unique_lock<std::mutex> lk(g_mut);
    g_running = false;
    g_cond.notify_one();
  }

  // Wait all done
  ret = process_loop_return.get();
  if (ret) {
    std::cout << "\n>>>>>>>>>>>>>>>>> Run succeed!! >>>>>>>>>>>>>>>>>" <<␣
↪std::endl;
  } else {
    std::cout << "\n>>>>>>>>>>>>>>>>> Run failed!! >>>>>>>>>>>>>>>>>" << std::endl;
  }
```

(下页继续)

(续上页)

```
  Destroy();
  return 0;
}
```

完整流程示例代码见路径：easydk/samples

- samples/stream-app 集成了视频解码、图像预处理（颜色空间转换和缩小图像）、神经网络推理、追踪、显示等功能，实现了一个以 SSD 网络为基础的目标检测应用程序。
- samples/classification 集成了视频解码、图像预处理（颜色空间转换和缩小图像）、神经网络推理、显示等功能，实现了一个以 ResNet50 网络为基础的目标识别应用程序。

7.2 推理服务示例

7.2.1 推理示例

使用 `contrib/PreprocessorMLU` 的推理示例如下，示例包含同步接口和异步接口的使用。

```
namespace infer_server {
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) {
    std::cout << "Get one response\n";
  }
};

// Load offline model
#ifdef CNIS_USE_MAGICMIND
ModelPtr model = InferServer::LoadModel(model_path);
#else
ModelPtr model = InferServer::LoadModel(model_path, func_name);
#endif

// Create session
SessionDesc desc;
desc.model = model;
desc.preproc = video::PreprocessorMLU::Create();
desc.postproc = Postprocessor::Create();
desc.batch_timeout = 200;
desc.priority = 1;
desc.engine_num = 1;
#ifdef CNIS_USE_MAGICMIND
```

(下页继续)

(续上页)

```
desc.preproc->SetParams("preprocess_type", video::PreprocessType::CNCV_PREPROC,
                        "dst_format", video::PixelFmt::RGB24,
                        "keep_aspect_ratio", true,
                        "mean", {104.f, 117.f, 123.f},
                        "std", {1.f, 1.f, 1.f},
                        "normalize", false);
#else
desc.preproc->SetParams("preprocess_type", video::PreprocessType::CNCV_PREPROC,
                        "dst_format", video::PixelFmt::RGBA);
#endif
int device_id = 0;
InferServer s(device_id);

// Create session
#if ASYNC
Session_t session = s.CreateSession(desc, std::make_shared<MyObserver>());
#else
Session_t session = s.CreateSyncSession(desc);
#endif

// Prepare input data with tag "stream_0"
// for wait or discard task with specified tag
auto input = Package::Create(1, "stream_0");
video::VideoFrame frame;
input->data[0]->Set(frame);

#if ASYNC
// run inference async
s.Request(session, input, nullptr);
#else
// run inference
auto out = std::make_shared<Package>();
Status stat;
s.RequestSync(session, input, &stat, out);
#endif

s.DestroySession(session);
}  // namespace infer_server
```


7.2.2 引入新模型示例

引入新模型时，模型的运行没有区别，加载模型部分无需改动。若新模型无法使用 contrib/PreprocessorMLU 内置的 CNCV 或 Scaler 完成预处理，仅需换用 PreprocessorHost ，并设置自实现的预处理函数即可。

```
namespace infer_server {
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) {
    std::cout << "Get one response\n";
  }
};

// Load offline model
#ifdef CNIS_USE_MAGICMIND
ModelPtr model = InferServer::LoadModel(new_model_path);
#else
ModelPtr model = InferServer::LoadModel(new_model_path, func_name);
#endif

auto preprocess_func =
    [](ModelIO* model_input, const InferData& data, const ModelInfo* model) {...};

// Create session
int device_id = 0;
InferServer s(device_id);
SessionDesc desc;
desc.model = model;
desc.preproc = PreprocessorHost::Create();
desc.preproc->SetParams<Preprocessor::ProcessFunction>("process_func", preprocess_
↪func);
desc.postproc = Postprocessor::Create();
desc.batch_timeout = 200;
desc.priority = 1;
desc.engine_num = 1;

// Create session
#ifdef ASYNC
Session_t session = s.CreateSession(desc, std::make_shared<MyObserver>());
#else
Session_t session = s.CreateSyncSession(desc);
#endif
```

(下页继续)

(续上页)

```
// Prepare input data
auto input = std::make_shared<Package>();
input->data.push_back(std::make_shared<InferData>());
MyData my_data;
input->data[0]->Set(my_data);
// For wait or discard task with specified tag
input->tag = "stream-id XX";

#ifdef ASYNC
// run inference async
s.Request(session, input, nullptr);
#else
// run inference
auto out = std::make_shared<Package>();
Status stat;
s.RequestSync(session, input, &stat, out);
#endif

s.DestroySession(session);
}  // namespace infer_server
```

7.2.3 自定义后处理单元示例

该示例展示了如何自定义后处理单元。模型输出数据以连续 batch 数据的形式存在，例如 | unit_1 unit_2 unit3 unit_4 |，该后处理单元简单地将模型输出数据拆分为分散的 unit，| unit_1 | unit_2 | unit_3 | unit_4 |，设置给输出的 `InferData`。

```
class MyPostprocessor : public ProcessorForkable<MyPostprocessor> {
public:
 MyPostprocessor() : ProcessorForkable<MyPostprocessor>("MyPostprocessor") {}
 ~MyPostprocessor() {}

 Status Init() noexcept override {
   // 检查是否所有必需参数已设置
   constexpr const char* params[] = {"model_info", "device_id"};
   for (auto p : params) {
     if (!HaveParam(p)) {
       LOG(ERROR) << p << " has not been set!";
       return Status::INVALID_PARAM;
```

(下页继续)

(续上页)

```
    }
  }
  // 读取参数
  try {
    model_ = GetParam<ModelPtr>("model_info");
    dev_id_ = GetParam<int>("device_id");
  } catch (bad_any_cast&) {
    LOG(ERROR) << "Unmatched data type";
    return Status::WRONG_TYPE;
  }
  return Status::SUCCESS;
 }

 Status Process(PackagePtr pack) noexcept override {
  // 检查输入，后处理单元的输入package中predict_io应该包含模型输出数据
  if (!pack->predict_io || !pack->predict_io->HasValue()) return Status::ERROR_
↪BACKEND;

  auto output = pack->predict_io->Get<ModelIO>();
  auto& shape = output.shapes[0];
  for (uint32_t idx = 0; idx < pack->data.size(); ++idx) {
    // 将整个batch的模型输出数据逐份拷贝到分离的内存上
    auto buf_size = output.buffers[0].MemorySize() / shape[0];
    Buffer buf(buf_size, dev_id_);
    buf.CopyFrom(output.buffers[0](idx * shape.DataCount()), buf_size);
    pack->data[idx]->Set(buf);
  }
  return Status::SUCCESS;
 }

private:
 ModelPtr model_;
 int dev_id_;
};
```


8

FAQ

8.1 有没有交叉编译 EasyDK 的指导?

可下载 edge 编译压缩包 http://video.cambricon.com/models/edge.tar.gz，解压后按照 README 文档提供的步骤进行编译。

8.2 如何将 Log 打印到终端以及如何带颜色显示?

可以通过环境变量 GLOG_alsologtostderr 或者命令行参数 alsologtostderr 将日志打印到终端。可以通过环境变量 GLOG_colorlogtostderr 或者命令行参数 colorlogtostderr 使不同等级的日志显示为不同的颜色。

8.3 怎么调整 Log 打印等级?

可以通过环境变量 GLOG_minloglevel 或者命令行参数 minloglevel 调整日志输出等级，范围 [0-3]，数字越小输出的 log 内容越多。* 0 代表打印 LOG(INFO) * 1 代表打印 LOG(WARNING) * 2 代表打印 LOG(ERROR) * 3 代表打印 LOG(FATAL)

另外，如果希望打印更多日志信息，在 minloglevel=0 的前提下，设置环境变量 GLOG_v 或者命令行参数 v 调整自定义日志等级，范围 [1-5]，数字越大输出的 log 内容越多。

