



# TDL\_SDK 开发文档

Version: 2.0

Release date: 2025-3-31

©2025 北京晶视智能科技有限公司  
本文件所含信息归北京晶视智能科技有限公司所有。  
未经授权，严禁全部或部分复制或披露该等信息。

# 目录

<b>1</b>	<b>TDL_SDK 简介</b>	<b>2</b>
1.1	TDL_SDK 总体结构	2
1.2	framework 模块	3
1.3	components 组件	4
1.4	其他	5
<b>2</b>	<b>TDL_SDK 环境搭建</b>	<b>7</b>
2.1	代码拉取及整体编译	7
2.2	TDL_SDK 编译方式	7
2.3	cmodel 模式	8
<b>3</b>	<b>TDL_SDK 模型列表</b>	<b>9</b>
3.1	检测类模型列表	9
3.2	人脸识别类模型列表	10
3.3	人脸属性和特征点类模型列表	10
3.4	图像分类模型列表	10
3.5	声音分类模型列表	11
3.6	关键点识别类模型列表	11
3.7	线识别类模型列表	11
3.8	车牌识别类模型列表	11
3.9	分割类模型列表	12
3.10	特征值提取类模型列表	12
<b>4</b>	<b>TDL_SDK 结构体参考</b>	<b>13</b>
4.1	TDLDataTypeE	13
4.2	TDLBox	14
4.3	TDLFeature	14
4.4	TDLPoints	15
4.5	TDLLandmarkInfo	15
4.6	TDLObjectInfo	16
4.7	TDLObject	16
4.8	TDLFaceInfo	17
4.9	TDLFace	18
4.10	TDLClassInfo	19
4.11	TDLClass	19
4.12	TDLKeypointInfo	19
4.13	TDLKeypoint	20
4.14	TDLSegmentation	20
4.15	TDLInstanceSegInfo	21
4.16	TDLInstanceSeg	22
4.17	TDLLanePoint	22

4.18	TDLLane	23
4.19	TDLDepthLogits	23
4.20	TDLTracker	24
4.21	TDLocr	24
4.22	TDLSnapshotInfo	25
4.23	TDLCaptureInfo	25
4.24	TDLObjectCountingInfo	26
4.25	TDLIspMeta	27
<b>5</b>	<b>TDL SDK API 参考</b>	<b>28</b>
5.1	句柄	28
5.2	TDL_CreateHandle	28
5.3	TDL_CreateHandleEx	29
5.4	TDL_DestroyHandle	29
5.5	TDL_DestroyHandleEx	29
5.6	TDL_WrapVPSSFrame	30
5.7	TDL_ReadImage	30
5.8	TDL_ReadBin	30
5.9	TDL_DestroyImage	31
5.10	TDL_OpenModel	31
5.11	TDL_OpenModelFromBuffer	32
5.12	TDL_CloseModel	32
5.13	TDL_Detection	33
5.14	TDL_FaceDetection	33
5.15	TDL_FaceAttribute	34
5.16	TDL_FaceLandmark	34
5.17	TDL_Classification	35
5.18	TDL_InstanceSegmentation	35
5.19	TDL_SemanticSegmentation	36
5.20	TDL_FeatureExtraction	36
5.21	TDL_LaneDetection	37
5.22	TDL_Tracking	37
5.23	TDL_SetSingleObjectTracking	38
5.24	TDL_SingleObjectTracking	38
5.25	TDL_CharacterRecognition	39
5.26	TDL_LoadModelConfig	39
5.27	TDL_SetModelDir	40
5.28	TDL_SetModelThreshold	40
5.29	TDL_IspClassification	41
5.30	TDL_Keypoint	41
5.31	TDL_DetectionKeypoint	42
5.32	TDL_IntrusionDetection	42
5.33	TDL_MotionDetection	43
5.34	TDL_APP_Init	43
5.35	TDL_APP_SetFrame	44
5.36	TDL_APP_Capture	44
5.37	TDL_APP_ObjectCounting	45
5.38	TDL_APP_ObjectCountingSetLine	45
5.39	TDL_WrapImage	46
5.40	TDL_LLMApiCall	46

<b>6</b>	<b>TDL_SDK 模型部署方式</b>	<b>48</b>
6.1	现有模型类添加新模型文件 . . . . .	48
6.2	集成新的模型类型 . . . . .	48
<b>7</b>	<b>TDL_SDK c sample 使用方式</b>	<b>49</b>
7.1	sample_character_recognition . . . . .	49
7.2	sample_classification . . . . .	49
7.3	sample_detect_keypoints . . . . .	50
7.4	sample_keypoints . . . . .	51
7.5	sample_face_attribute . . . . .	51
7.6	sample_face_detection . . . . .	52
7.7	sample_feature_extraction . . . . .	52
7.8	sample_face_landmark . . . . .	53
7.9	sample_instance_segmentation . . . . .	53
7.10	sample_lane_detection . . . . .	54
7.11	sample_object_detection . . . . .	54
7.12	sample_pose . . . . .	55
7.13	sample_semantic_segmentation . . . . .	56
7.14	sample_tracking . . . . .	57
7.15	sample_face_recognition . . . . .	57
7.16	sample_licence_recognition . . . . .	58
7.17	sample_vi_detection . . . . .	58
7.18	sample_vi_face_pet_cap . . . . .	59
7.19	sample_vi_consumer_counting . . . . .	60
7.20	sample_vi_cross_detection . . . . .	60
7.21	sample_vi_single_object_tracking . . . . .	61
7.22	sample_motion_detection . . . . .	61
7.23	sample_intrusion_detection . . . . .	61
<b>8</b>	<b>常见问题</b>	<b>63</b>
8.1	模型打开失败类问题 . . . . .	63

## 修订记录

Date	Description	Owner
2025/03/28	初稿	张思意
2025/03/31	添加 API 定义	刘俊斐
2025/04/02	添加模型列表和结构体定义	郑鑫烨
2025/08/29	更新模型列表和结构体定义	郑鑫烨
2025/09/30	添加环境搭建，模型部署方式和 sample 使用方式	郑鑫烨

# 1 TDL\_SDK 简介

TDL\_SDK(Turnkey Deep Learning SDK) 是一个基于算能芯片产品的开箱即用深度学习算法 SDK, 致力于为用户提供跨平台(端、边)、简单易用、资源节约、性能高效的算法库及应用。它基于模块化设计, 同类型功能模块抽象出基类, 遵循高内聚低耦合, 仅对外暴露基类, 不暴露具体实现; 同时具有可扩展性和可维护性。

## 1.1 TDL\_SDK 总体结构

```
tld_sdk/
├── CMakeLists.txt # 根 CMake 构建脚本
├── build_tld_sdk.sh # SDK 构建脚本
├── clang-format.sh # 代码格式化脚本
├── clang-tidy.sh # 代码静态分析脚本
├── .clang-format # Clang 格式化配置
├── .clang-tidy # Clang 静态分析配置
├── cmake/ # CMake 相关模块
│   ├── opencv.cmake # OpenCV 依赖查找
│   ├── middleware.cmake # 多媒体件依赖查找
│   ├── mlir.cmake # MLIR 依赖查找
│   └── thirdparty.cmake # 第三方库依赖查找
├── configs/ # 模型参数配置文件
├── docs/ # 文档目录
│   ├── README.md # 文档说明
│   ├── LICENSE # 许可证文件
│   ├── getting_started/ # 入门指南
│   ├── developer_guide/ # 开发者文档
│   ├── api_reference/ # API 参考手册
│   └── images/ # 文档图片资源
├── include/ # 对外导出的头文件
│   ├── framework/ # 框架层 API
│   ├── components/ # 组件 API
│   ├── nn/ # 神经网络模型 API
│   ├── app/ # app类 API
│   ├── pipeline/ # pipeline类 API
│   └── c_apis/ # C 语言 API
├── src/ # 内部实现
│   ├── framework/ # 框架层实现
│   │   ├── common/ # 公共工具实现
│   │   ├── image/ # 图像处理实现
│   │   └── memory/ # 内存管理实现
```

(下页继续)

(续上页)

```

├── model/ # 模型实现
├── net/ # 神经网络实现
├── preprocess/ # 预处理实现
├── tensor/ # 张量处理实现
├── utils/ # 通用工具实现
├── components/ # 组件实现
│   ├── cv/ # 视觉相关的检测类实现
│   ├── encoder/ # 编码相关功能实现
│   ├── ive/ # ive图像处理功能实现
│   ├── llm/ # 大模型类实现
│   ├── matcher/ # 特征匹配功能的实现
│   ├── network/ # 网络组件的实现
│   ├── nn/ # 神经网络模型实现
│   ├── tracker/ # 目标跟踪实现
│   ├── snapshot/ # 抓拍实现
│   └── video_decoder/ # 相机和解码实现
├── c_apis/ # C API 封装
├── python/ # Python 绑定
├── sample/ # 示例代码
│   ├── cpp/ # C++ 示例
│   ├── c/ # C 示例
│   └── python/ # Python 示例
├── evaluation/ # 性能评估
├── tool/ # 工具集
├── toolchain/ # 工具链
├── scripts/ # 脚本
└── README.md # 项目说明

```

## 1.2 framework 模块

为实现跨平台模型推理的统一框架，基于此框架部署的模型，可以在多种硬件平台运行。该框架包括的模块如下：

1. common
  - 公共定义和工具，包括错误码、日志、配置等
  - 提供跨模块使用的通用功能
2. image
  - 图像类的抽象封装，支持多种图像格式和数据类型
  - 提供图像读取、转换、处理等基础功能
  - 支持 OpenCV、VPSS 等多种后端实现
3. memory
  - 内存池类的抽象封装，用于高效内存管理
  - 支持多种内存类型（系统内存、设备内存等）
  - 提供内存分配、释放和复用机制

#### 4. model

- 模型类的抽象封装，用于加载和运行神经网络模型
- 支持多种模型格式（ONNX、TensorFlow、PyTorch 等）
- 提供模型推理和优化功能

#### 5. net

- 神经网络类接口的封装，用于推理结果

#### 6. preprocess

- 预处理类的抽象封装，用于图像预处理
- 支持多种预处理操作（缩放、裁剪、归一化等）
- 提供 OpenCV、VPSS 等多种后端实现

#### 7. tensor

- 张量类的抽象封装，用于表示神经网络模型的输入输出
- 支持多种数据类型和内存布局
- 提供张量操作和转换功能

#### 8. utils

- 工具类接口的封装，包括有定时计数，图像对齐和计算等工具

## 1.3 components 组件

具体算法相关的组件，包括：

#### 1. cv

- 视觉相关的检测类实现
- 已支持入侵检测，运动检测和遮挡检测

#### 2. encoder

- 编码类接口的实现
- 支持 rtsp 出流，实时查看算法效果

#### 3. ive

- ive 图像处理功能实现
- 使用 tpu 进行计算

#### 4. llm

- 大模型实现
- 支持 qwen 和 qwen2-VL

#### 5. matcher

- 特征匹配功能的实现



- 支持使用 cpu 或 tpu 进行计算
- 6. network
  - 网络功能的实现
- 7. nn
  - 各种神经网络模型实现，如目标检测、人脸检测、车牌识别等
  - 提供模型加载、推理和结果解析功能
- 8. snapshot
  - 抓拍功能的实现
- 9. track
  - 目标跟踪算法实现
  - 支持多种跟踪算法（KCF、SORT、DeepSORT 等）
- 10. video\_decoder
  - 相机和解码功能实现
  - 支持多种相机接口和解码格式

## 1.4 其他

1. c\_apis: C 语言 API 封装，提供跨语言调用支持
  - 提供与 C++ API 功能对应的 C 接口
  - 支持 C 语言应用程序集成
  - 提供内存管理和错误处理机制
2. sample: 示例代码，展示 SDK 的使用方法
  - C++ 语言示例，展示框架层和组件层的使用方法
  - C 语言示例，展示 C API 的使用方法
  - Python 语言示例，展示 Python 绑定的使用方法
3. evaluation: 性能评估，用于评估 SDK 的性能
  - 提供性能测试和基准测试功能
  - 支持多种性能指标（吞吐量、延迟、内存使用等）
  - 提供性能分析和优化建议
4. tool: 工具集，提供开发和调试支持
  - 模型转换工具
  - 性能分析工具
  - 调试和日志工具
5. toolchain: 工具链，提供编译和构建支持

- 交叉编译工具链
- 依赖库和头文件
- 构建脚本和配置

6. scripts: 脚本，提供自动化支持

- 构建脚本
- 测试脚本
- 部署脚本

# 2 TDL\_SDK 环境搭建

## 2.1 代码拉取及整体编译

TDL\_SDK 除了 cmodel 模式外，都需要依赖 sophpi sdk 才能正常运行，sophpi sdk 的获取方式和编译方式在如下网址中：

```
https://github.com/sophgo/sophpi
```

请先按照网址中的教程拉取代码，TDL\_SDK 也将随着 sophpi sdk 被下载下来。在进行编译之前，需执行 `export TPU_REL=1`，这样才能编译到 TDL\_SDK 以及其他相关库，具体的编译流程如下：

```
export TPU_REL=1
source build/envsetup_soc.sh
defconfig sg2002_wevb_riscv64_sd //这里需要选择对应的板端型号
clean_all
build_all //编译所有组件，包括TDL_SDK
```

第一次编译时，一定要完整执行如上的步骤。

## 2.2 TDL\_SDK 编译方式

进行完上述步骤后，就可以仅编译 TDL\_SDK，提供了两种编译方式

1. 使用 makefile 定义的伪目标编译

```
build_tdl_sdk
```

如果编译失败先使用 `clean_tdl_sdk` 将旧的编译产物清除掉再重新编译。

2. 使用 `build_tdl_sdk.sh` 脚本进行编译

```
cd tdl_sdk
./build_tdl_sdk.sh all
```

如果编译失败先使用 `./build_tdl_sdk.sh clean` 将旧的编译产物清除掉再重新编译。

`build_tdl_sdk.sh` 提供了一些编译配置供选择，具体如下：

```
./build_tdl_sdk.sh sample //仅编译sample, 在仅改动sample文件夹下内容时使用, 节省编译时间
./build_tdl_sdk.sh static //编译的sample为静态文件, 注意static这个参数仅影响sample, TDL_
→SDK依旧会编译出静态库和动态库
./build_tdl_sdk.sh debug //编译时引入debug信息, 仅调试时使用
```

## 2.3 cmodel 模式

TDL\_SDK 支持 cmodel 模式, 可以支持在 PC 端模拟芯片的运算逻辑, 使得 sample 能够直接在 PC 端模拟芯片端运行测试。cmodel 使用方式如下:

### 1. 下载第三方依赖库

```
cd tdl_sdk
./scripts/download_thirdparty.sh
```

### 2. 获取编译依赖

```
./scripts/extract_cvitek_tpu_sdk.sh
```

### 3. 编译 TDL\_SDK

```
./build_tdl_sdk.sh CMODEL_CVITEK
```

如果编译失败先使用 `./build_tdl_sdk.sh clean` 将旧的编译产物清除掉再重新编译。

# 3 TDL\_SDK 模型列表

## 3.1 检测类模型列表

模型名称	注释
TDL_MODEL_MBV2_DET_PERSON	人体检测模型 (0:person)
TDL_MODEL_YOLOV8N_DET_HAND	手部检测模型 (0:hand)
TDL_MODEL_YOLOV8N_DET_PET_PERSON	宠物与人检测模型 (0: 猫, 1: 狗, 2: 人)
TDL_MODEL_YOLOV8N_DET_PERSON_VEHICLE	人与车辆检测模型 (0: 车, 1: 公交, 2: 卡车, 3: 骑摩托车者, 4: 人, 5: 自行车, 6: 摩托车)
TDL_MODEL_YOLOV8N_DET_HAND_FACE_PERSON	手、脸与人检测模型 (0: 手, 1: 脸, 2: 人)
TDL_MODEL_YOLOV8N_DET_HEAD_PERSON	人头检测模型 (0: 人, 1: 头)
TDL_MODEL_YOLOV8N_DET_HEAD_HARDHAT	头部与安全帽检测模型 (0: 头, 1: 安全帽)
TDL_MODEL_YOLOV8N_DET_FIRE_SMOKE	火与烟检测模型 (0: 火, 1: 烟)
TDL_MODEL_YOLOV8N_DET_FIRE	火检测模型 (0: 火)
TDL_MODEL_YOLOV8N_DET_HEAD_SHOULDER	头肩检测模型 (0: 头肩)
TDL_MODEL_YOLOV8N_DET_LICENSE_PLATE	车牌检测模型 (0: 车牌)
TDL_MODEL_YOLOV8N_DET_TRAFFIC_LIGHT	交通信号灯检测模型 (0: 红, 1: 黄, 2: 绿, 3: 关闭, 4: 等待)
TDL_MODEL_YOLOV8N_DET_MONITOR_PERSON	人体检测模型 (0:person)
TDL_MODEL_YOLOV5_DET_COCO80	YOLOV5 COCO80 检测模型
TDL_MODEL_YOLOV6_DET_COCO80	YOLOV6 COCO80 检测模型
TDL_MODEL_YOLOV7_DET_COCO80	YOLOV7 COCO80 检测模型
TDL_MODEL_YOLOV8_DET_COCO80	YOLOV8 COCO80 检测模型
TDL_MODEL_YOLOV10_DET_COCO80	YOLOV10 COCO80 检测模型
TDL_MODEL_PPYOLOE_DET_COCO80	PPYOLOE COCO80 检测模型
TDL_MODEL_YOLOX_DET_COCO80	YOLOX COCO80 检测模型

## 3.2 人脸识别类模型列表

模型名称	注释
TDL_MODEL_SCRFD_DET_FACE	SCRFD 人脸检测模型 (0: 人脸 + 关键点)
TDL_MODEL_RETINA_DET_FACE	RETINA 人脸检测模型
TDL_MODEL_RETINA_DET_FACE_IR	RETINA 红外人脸检测模型

## 3.3 人脸属性和特征点类模型列表

模型名称	注释
TDL_MODEL_KEYPOINT_FACE_V2	5 个关键点 + 模糊评分的人脸检测模型
TDL_MODEL_CLS_ATTRIBUTE_GENDER _AGE_GLASS	人脸属性分类模型 (年龄, 性别, 眼镜)
TDL_MODEL_CLS_ATTRIBUTE_GENDER _AGE_GLASS_MASK	人脸属性分类模型 (年龄, 性别, 眼镜, 口罩)
TDL_MODEL_CLS_ATTRIBUTE_GENDER _AGE_GLASS_EMOTION	人脸属性分类模型 (年龄, 性别, 眼镜, 情绪)

## 3.4 图像分类模型列表

模型名称	注释
TDL_MODEL_CLS_MASK	口罩检测模型 (0: 戴口罩, 1: 不戴口罩)
TDL_MODEL_CLS_RGBLIVENESS	活体检测模型 (0: 活体, 1: 伪造)
TDL_MODEL_CLS_ISP_SCENE	ISP 场景分类模型 ( “0: 雪, 1: 雾, 2: 逆光, 3: 草, 4: 普通场景” )
TDL_MODEL_CLS_HAND_GESTURE	手势分类模型 (0: 拳头, 1: 五指, 2: 无, 3: 二)
TDL_MODEL_CLS_KEYPOINT_HAND _GESTURE	手势关键点分类模型 (0: 拳头, 1: 五指, 2: 四指, 3: 无, 4: 好, 5: 一, 6: 三, 7: 三 2, 8: 二)

## 3.5 声音分类模型列表

模型名称	注释
TDL_MODEL_CLS_SOUND_BABAY_CRY	婴儿哭声分类模型 (0: 背景, 1: 哭声)
TDL_MODEL_CLS_SOUND_COMMAND_NIHAOSHIYUN	命令声音分类模型 (0: 背景, 1: nihaoshiyun)
TDL_MODEL_CLS_SOUND_COMMAND_NIHAOSUANNENG	命令声音分类模型 (0: 背景, 1: nihaosuan-neng)
TDL_MODEL_CLS_SOUND_COMMAND_XIAOAIXIAOAI	命令声音分类模型 (0: 背景, 1: xiaoaixiaoai)
TDL_MODEL_CLS_SOUND_COMMAND	命令声音分类模型 (0: 背景, 1: 命令 1, 2: 命令 2)

## 3.6 关键点识别类模型列表

模型名称	注释
TDL_MODEL_KEYPOINT_LICENSE_PLATE	车牌关键点检测模型
TDL_MODEL_KEYPOINT_HAND	手部关键点检测模型
TDL_MODEL_KEYPOINT_YOLOV8POSE_PERSON17	人体 17 个关键点检测模型
TDL_MODEL_KEYPOINT_SIMCC_PERSON17	SIMCC 17 个关键点检测模型

## 3.7 线识别类模型列表

模型名称	注释
TDL_MODEL_LSTR_DET_LANE	车道检测模型

## 3.8 车牌识别类模型列表

模型名称	注释
TDL_MODEL_RECOGNITION_LICENSE_PLATE	车牌识别模型

## 3.9 分割类模型列表

TDL_MODEL_YOLOV8_SEG_COCO80	YOLOV8 COCO80 分割模型
TDL_MODEL_SEG_PERSON_FACE_VEHICLE	人、脸与车辆分割模型 (0: 背景, 1: 人, 2: 脸, 3: 车辆, 4: 车牌)
TDL_MODEL_SEG_MOTION	动作分割模型 (0: 静态, 2: 过渡, 3: 运动)

## 3.10 特征值提取类模型列表

TDL_MODEL_FEATURE_IMG	图像特征提取模型
TDL_MODEL_IMG_FEATURE_CLIP	CLIP 图像嵌入模型
TDL_MODEL_TEXT_FEATURE_CLIP	CLIP 文本嵌入模型
TDL_MODEL_FEATURE_CVIFACE	Cviface 256 维特征提取模型
TDL_MODEL_FEATURE_BMFACE_R34	ResNet34 512 维特征提取模型
TDL_MODEL_FEATURE_BMFACE_R50	ResNet50 512 维特征提取模型



# 4 TDL\_SDK 结构体参考

## 4.1 TDLDataTypeE

### 【说明】

数据类型枚举类

### 【定义】

```
typedef enum {  
    TDL_TYPE_INT8 = 0, /**< Equals to int8_t. */  
    TDL_TYPE_UINT8, /**< Equals to uint8_t. */  
    TDL_TYPE_INT16, /**< Equals to int16_t. */  
    TDL_TYPE_UINT16, /**< Equals to uint16_t. */  
    TDL_TYPE_INT32, /**< Equals to int32_t. */  
    TDL_TYPE_UINT32, /**< Equals to uint32_t. */  
    TDL_TYPE_BF16, /**< Equals to bf17. */  
    TDL_TYPE_FP16, /**< Equals to fp16. */  
    TDL_TYPE_FP32, /**< Equals to fp32. */  
    TDL_TYPE_UNKOWN /**< Equals to unkown. */  
} TDLDataTypeE;
```

### 【成员】

数据类型枚举类	注释
TDL_TYPE_INT8	有符号 8 位整数
TDL_TYPE_UINT8	无符号 8 位整数
TDL_TYPE_INT16	有符号 16 位整数
TDL_TYPE_UINT16	无符号 16 位整数
TDL_TYPE_INT32	有符号 32 位整数
TDL_TYPE_UINT32	无符号 32 位整数
TDL_TYPE_BF16	16 位浮点数 (1 位符号, 8 位指数和 7 位尾数)
TDL_TYPE_FP16	16 位浮点数 (1 位符号, 5 位指数和 10 位尾数)
FTDL_TYPE_FP32	32 位浮点数

## 4.2 TDLBox

### 【说明】

box 的坐标数据

### 【定义】

```
typedef struct {  
    float x1;  
    float y1;  
    float x2;  
    float y2;  
} TDLBox;
```

### 【成员】

数据类型枚举类	注释
x1	box 左上角 x 的坐标
y1	box 左上角 y 的坐标
x2	box 右下角 x 的坐标
y2	box 右下角 y 的坐标

## 4.3 TDLFeature

### 【说明】

特征值数据

### 【定义】

```
typedef struct {  
    int8_t *ptr;  
    uint32_t size;  
    TDLDataTypeE type;  
} TDLFeature;
```

### 【成员】

数据类型枚举类	注释
ptr	特征值数据
size	数据大小
type	数据类型

## 4.4 TDLPoints

### 【说明】

坐标队列数据

### 【定义】

```
typedef struct {  
    float *x;  
    float *y;  
    uint32_t size;  
    float score;  
} TDLPoints;
```

### 【成员】

数据类型枚举类	注释
x	坐标队列的 x 数据
y	坐标队列的 y 数据
size	坐标队列的大小
score	分数

## 4.5 TDLLandmarkInfo

### 【说明】

特征点信息

### 【定义】

```
typedef struct {  
    float x;  
    float y;  
    float score;  
} TDLLandmarkInfo;
```

### 【成员】

数据类型枚举类	注释
x	特征点的 x 坐标
y	特征点的 y 坐标
score	分数

## 4.6 TDLObjectInfo

### 【说明】

目标检测信息

### 【定义】

```
typedef struct {  
    char name[128];  
    TDLBox box;  
    float score;  
    int class_id;  
    uint32_t landmark_size;  
    TDLLandmarkInfo *landmark_properity;  
    TDLObjectTypeE obj_type;  
} TDLObjectInfo;
```

### 【成员】

数据类型枚举类	注释
name	目标检测的命名
box	目标检测的框图信息
score	目标检测的分数
class_id	目标检测的类别 id
landmark_size	目标检测的特征点大小
TDLLandmarkInfo	目标检测的特征点信息
obj_type	目标检测的类型

## 4.7 TDLObject

### 【说明】

目标检测数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    uint32_t width;  
    uint32_t height;  
  
    TDLObjectInfo *info;  
} TDLObject;
```

### 【成员】

数据类型枚举类	注释
size	目标检测的个数
width	目标检测图像的宽度
height	目标检测图像的高度
info	目标检测信息

## 4.8 TDLFaceInfo

### 【说明】

人脸信息

### 【定义】

```
typedef struct {  
    char name[128];  
    float score;  
    uint64_t track_id;  
    TDLBox box;  
    TDLPoints landmarks;  
    TDLFeature feature;  
  
    float gender_score;  
    float glass_score;  
    float age;  
    float liveness_score;  
    float hardhat_score;  
    float mask_score;  
  
    float recog_score;  
    float face_quality;  
    float pose_score;  
    float blurness;  
} TDLFaceInfo;
```

### 【成员】

数据类型枚举类	注释
name	人脸的姓名
score	人脸的分数
track_id	人脸的追踪 id
box	人脸的 box 信息
landmarks	人脸的特征点
feature	人脸的特征值
gender_score	人脸的性别分数
glass_score	人脸是否带眼镜
age	人脸的年龄
liveness_score	人脸的活体分数
hardhat_score	人脸的是否带安全帽分数
recog_score	人脸的识别率分数
face_quality	人脸的质量分数
pose_score	人脸的姿态分数
blurness	人脸的模糊度

## 4.9 TDLFace

### 【说明】

人脸数据

### 【定义】

```
typedef struct {
    uint32_t size;
    uint32_t width;
    uint32_t height;
    TDLFaceInfo *info;
} TDLFace;
```

### 【成员】

数据类型枚举类	注释
size	人脸的个数
width	人脸图像的宽度
height	人脸图像的高度
info	人脸信息

## 4.10 TDLClassInfo

### 【说明】

分类信息

### 【定义】

```
typedef struct {  
    int32_t class_id;  
    float score;  
} TDLClassInfo;
```

### 【成员】

数据类型枚举类	注释
class_id	分类的类别
score	分类的分数

## 4.11 TDLClass

### 【说明】

分类数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    TDLClassInfo *info;  
} TDLClass;
```

### 【成员】

数据类型枚举类	注释
size	分类的个数
info	分类信息

## 4.12 TDLKeypointInfo

### 【说明】

关键点信息

### 【定义】

```
typedef struct {
    float x;
    float y;
    float score;
} TDLKeypointInfo;
```

**【成员】**

数据类型枚举类	注释
x	关键点的 x 坐标
y	关键点的 y 坐标
score	关键点的分数

## 4.13 TDLKeypoint

**【说明】**

关键点数据

**【定义】**

```
typedef struct {
    uint32_t size;
    uint32_t width;
    uint32_t height;
    TDLKeypointInfo *info;
} TDLKeypoint;
```

**【成员】**

数据类型枚举类	注释
size	关键点的个数
width	图像的宽度
height	图像的高度
info	关键点信息

## 4.14 TDLSegmentation

**【说明】**

语义分割数据

**【定义】**

```
typedef struct {
    uint32_t width;
    uint32_t height;
```

(下页继续)



(续上页)

```

uint32_t output_width;
uint32_t output_height;
uint8_t *class_id;
uint8_t *class_conf;
} TDLSegmentation;

```

**【成员】**

数据类型枚举类	注释
width	图像的宽度
height	图像的高度
output_width	输出图像的宽度
output_height	输出图像的高度
class_id	分类的类别
class_conf	分类的坐标信息

## 4.15 TDLInstanceSegInfo

**【说明】**

实例分割信息

**【定义】**

```

typedef struct {
    uint8_t *mask;
    float *mask_point;
    uint32_t mask_point_size;
    TDLObjectInfo *obj_info;
} TDLInstanceSegInfo;

```

**【成员】**

数据类型枚举类	注释
mask	实例分割的 mask 队列
mask_point	实例分割的 mask_point 队列
mask_point_size	实例分割的 point 个数
obj_info	实例分割的目标检测信息

## 4.16 TDLInstanceSeg

### 【说明】

实例分割数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    uint32_t width;  
    uint32_t height;  
    uint32_t mask_width;  
    uint32_t mask_height;  
    TDLInstanceSegInfo *info;  
} TDLInstanceSeg;
```

### 【成员】

数据类型枚举类	注释
size	实例分割的个数
width	图像的宽度
height	图像的高度
mask_width	mask 的宽度
mask_height	mask 的高度
info	实例分割信息

## 4.17 TDLLanePoint

### 【说明】

线检测的坐标点

### 【定义】

```
typedef struct {  
    float x[2];  
    float y[2];  
    float score;  
} TDLLanePoint;
```

### 【成员】

数据类型枚举类	注释
x	x 坐标队列
y	y 坐标队列
score	线检测的分数

## 4.18 TDLLane

### 【说明】

线检测数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    uint32_t width;  
    uint32_t height;  
    TDLLanePoint *lane;  
    int lane_state;  
} TDLLane;
```

### 【成员】

数据类型枚举类	注释
size	线检测的个数
width	图像的宽度
height	图像的高度
lane	线检测坐标点
lane_state	线条状态

## 4.19 TDLDepthLogits

### 【说明】

深度估计数据，目前暂未实现对应 C 接口

### 【定义】

```
typedef struct {  
    int w;  
    int h;  
    int8_t *int_logits;  
} TDLDepthLogits;
```

### 【成员】

数据类型枚举类	注释
w	图像的宽度
h	图像的高度
int_logits	深度估计信息

## 4.20 TDLTracker

### 【说明】

追踪数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    uint64_t id;  
    TDLBox bbox;  
    int out_num;  
} TDLTracker;
```

### 【成员】

数据类型枚举类	注释
size	追踪目标的个数
id	追踪目标的 id
bbox	追踪目标的 box
out_num	追踪目标的小时次数

## 4.21 TDLOcr

### 【说明】

文本识别数据

### 【定义】

```
typedef struct {  
    uint32_t size;  
    char* text_info;  
} TDLOcr;
```

### 【成员】

数据类型枚举类	注释
size	文本识别的个数
text_info	文本识别的信息

## 4.22 TDLSnapshotInfo

### 【说明】

抓拍信息

### 【定义】

```
typedef struct {  
    float quality;  
    uint64_t snapshot_frame_id;  
    uint64_t track_id;  
    bool male;  
    bool glass;  
    uint8_t age;  
    uint8_t emotion;  
    TDLImage object_image;  
} TDLSnapshotInfo;
```

### 【成员】

数据类型枚举类	注释
quality	抓拍的质量
snapshot_frame_id	帧 id
track_id	追踪 id
male	性别分数
glass	眼镜分数
age	年龄分数
emotion	情绪状态
object_image	抓拍的 image 对象

## 4.23 TDLCaptureInfo

### 【说明】

抓拍参数结构体

### 【定义】

```
typedef struct {  
    uint32_t snapshot_size;  
    uint64_t frame_id;  
    uint32_t frame_width;  
    uint32_t frame_height;  
    TDLFace face_meta;  
    TDLObject person_meta;  
    TDLObject pet_meta;  
    TDLTracker track_meta;  
    TDLSnapshotInfo *snapshot_info;  
    TDLFeature *features;
```

(下页继续)

(续上页)

```
TDLImage image;
} TDLCaptureInfo;
```

**【成员】**

数据类型枚举类	注释
snapshot_size	抓拍的个数
frame_id	帧 id
frame_width	帧宽度
frame_height	帧长度
face_meta	人脸数据
person_meta	行人数据
pet_meta	宠物数据
track_meta	追踪数据
snapshot_info	抓拍信息
features	特征值信息
image	用于检测的原始图像

## 4.24 TDLObjectCountingInfo

**【说明】**

客流量统计信息

**【定义】**

```
typedef struct {
    uint64_t frame_id;
    uint32_t frame_width;
    uint32_t frame_height;
    uint32_t enter_num;
    uint32_t miss_num;
    int counting_line[4];
    TDLObject object_meta;
    TDLImage image;
} TDLObjectCountingInfo;
```

**【成员】**

数据类型枚举类	注释
frame_id	帧 id
frame_width	帧宽度
frame_height	帧长度
enter_num	进入的数量
miss_num	出去的数量
counting_line	穿越线，通常位于关键通道口
object_meta	目标物体数据
image	用于检测的原始图像

## 4.25 TDLIspMeta

### 【说明】

isp 数据

### 【定义】

```
typedef struct {  
    float awb[3]; // rgain, ggain, bgain  
    float ccm[9]; // rgb[3][3]  
    float blc;  
} TDLIspMeta;
```

### 【成员】

数据类型枚举类	注释
awb	白平衡信息
ccm	颜色信息
blc	黑电平信息

# 5 TDL\_SDK API 参考

## 5.1 句柄

### 【语法】

```
typedef void *TDLHandle;  
typedef void *TDLHandleEx;  
typedef void *TDLImage;
```

### 【描述】

TDL\_SDK 句柄, TDLHandle 是核心操作句柄, TDLHandleEx 是扩展操作句柄, TDLImage 是图像数据抽象句柄。

## 5.2 TDL\_CreateHandle

### 【语法】

```
TDLHandle TDL_CreateHandle(const int32_t tpu_device_id);
```

### 【描述】

创建一个 TDLHandle 对象, 用于 tdl api 的管理, 在使用核心库 (libtdl\_core.a 或 libtdl\_core.so) 中的 api 时使用。

### 【参数】

	数据类型	参数名称	说明
输入	const int32_t	tpu_device_id	指定 TPU 设备的 ID, 默认设备 ID 为 0, 目前暂未开放其他 ID 的使用



## 5.3 TDL\_CreateHandleEx

### 【语法】

```
TDLHandleEx TDL_CreateHandleEx(const int32_t tpu_device_id);
```

### 【描述】

创建一个 TDLHandleEx 对象，用于 tdl 拓展 api 的管理，在使用拓展库（libtdl\_ex.a 或 libtdl\_ex.so）中的 api 时使用。

### 【参数】

	数据类型	参数名称	说明
输入	const int32_t	tpu_device_id	指定 TPU 设备的 ID，默认设备 ID 为 0，目前暂未开放其他 ID 的使用

## 5.4 TDL\_DestroyHandle

### 【语法】

```
int32_t TDL_DestroyHandle(TDLHandle handle);
```

### 【描述】

销毁一个 TDLHandle 对象，在使用核心库（libtdl\_core.a 或 libtdl\_core.so）中的 api 时使用。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	需要销毁的 TDLHandle 对象

## 5.5 TDL\_DestroyHandleEx

### 【语法】

```
int32_t TDL_DestroyHandleEx(TDLHandleEx handle);
```

### 【描述】

销毁一个 TDLHandleEx 对象，在使用拓展库（libtdl\_ex.a 或 libtdl\_ex.so）中的 api 时使用。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandleEx	handle	需要销毁的 TDLHandleEx 对象

## 5.6 TDL\_WrapVPSSFrame

### 【语法】

```
TDLImage TDL_WrapVPSSFrame(void *vpss_frame, bool own_memory);
```

### 【描述】

包装一个 VPSS 帧为 TDLImage 对象，这里的 frame 可以通过多媒体接口获得的 frame（例如 CVI\_VPSS\_GetChnFrame）；也可以是自己封装的一个 frame，示例如下

```
VIDEO_FRAME_INFO_S Frame;
memset(&Frame, 0, sizeof(VIDEO_FRAME_INFO_S));
Frame.stVFrame.pu8VirAddr[0] = buffer; // 数据的虚拟地址
Frame.stVFrame.u32Height = 1;           // 数据的高度，示例为一维数据，所以这里为1
Frame.stVFrame.u32Width = u32BufferSize; // 数据的宽度，示例为一维数据，这里就是数据量

TDLImage image = TDL_WrapVPSSFrame((void *)&Frame, false); // 调用接口创建图像
```

### 【参数】

	数据类型	参数名称	说明
输入	void*	vpss_frame	需要包装的 VPSS 帧
输入	bool	own_memory	是否拥有内存所有权，该功能暂未实现，默认为 false

## 5.7 TDL\_ReadImage

```
TDLImage TDL_ReadImage(const char *path);
```

### 【描述】

读取一张图片为 TDLImage 对象，支持 jpg, png, bmp, jp2, sr 和 tiff 等常见格式的输入。

### 【参数】

	数据类型	参数名称	说明
输入	const char*	path	图片路径

## 5.8 TDL\_ReadBin

### 【语法】

```
TDLImage TDL_ReadBin(const char *path, int count, TDLDataTypeE data_type);
```

### 【描述】

读取文件内容为 TDLImage 对象。

## 【参数】

	数据类型	参数名称	说明
输入	const char*	path	bin 文件路径
输入	int	count	文件中数据量
输入	TDLDataTypeE	data_type	输入数据类型

## 5.9 TDL\_DestroyImage

## 【语法】

```
int32_t TDL_DestroyImage(TDLImage image_handle);
```

## 【描述】

销毁一个 TDLImage 对象，在结束对一个 TDLImage 对象的使用时，一定要调用此接口释放内存。

## 【参数】

	数据类型	参数名称	说明
输入	TDLImage	image_handle	需要销毁的 TDLImage 对象

## 5.10 TDL\_OpenModel

## 【语法】

```
int32_t TDL_OpenModel(TDLHandle handle,  
                      const TDLModel model_id,  
                      const char *model_path,  
                      const char *model_config_json);
```

## 【描述】

加载指定类型的模型到 TDLHandle 对象中，对于参数 model\_config\_json，若使用 TDL\_LoadModelConfig 加载后可以传入 NULL；不使用 TDL\_LoadModelConfig 加载，大部分专有模型也可以传入 NULL，此时会使用算法类内部的默认配置，部分通用模型如特征提取、声音指令等需要传入模型配置信息，可以参考 configs/model/model\_config.json。

## 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	const char*	model_path	模型文件路径
输入	const char*	model_config_json	模型配置文件的路径，文件路径在 tdl_sdk/install/CV184X/configs/model/ 中，如果模型不需要设定特别的参数，可以为 NULL

## 5.11 TDL\_OpenModelFromBuffer

### 【语法】

```
int32_t TDL_OpenModelFromBuffer(TDLHandle handle,
                                const TDLModel model_id,
                                const uint8_t *model_buffer,
                                uint32_t model_buffer_size,
                                const char *model_config_json);
```

### 【描述】

加载指定类型的模型到 TDLHandle 对象中，使用地址来传参。对于参数 model\_config\_json，若使用 TDL\_LoadModelConfig 加载后可以传入 NULL；不使用 TDL\_LoadModelConfig 加载，大部分专有模型也可以传入 NULL，此时会使用算法类内部的默认配置，部分通用模型如特征提取、声音指令等需要传入模型配置信息，可以参考 configs/model/model\_config.json。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	const uint8_t*	model_buffer	模型文件 buffer
输入	uint32_t	model_buffer_size	模型文件 buffer 大小
输入	const char*	model_config_json	模型配置文件的路径，文件路径在 tdl_sdk/install/CV184X/configs/model/ 中，如果模型不需要设定特别的参数，可以为 NULL

## 5.12 TDL\_CloseModel

### 【语法】

```
int32_t TDL_CloseModel(TDLHandle handle,
                       const TDLModel model_id);
```

### 【描述】

卸载指定类型的模型并释放相关资源。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举

## 5.13 TDL\_Detection

**【语法】**

```
int32_t TDL_Detection(TDLHandle handle,
                      const TDLModel model_id,
                      TDLImage image_handle,
                      TDLObject *object_meta);
```

**【描述】**

执行指定模型的推理检测，并返回检测结果元数据，详细实例可以参考 tdl\_sdk/sample/c/sample\_object\_detection.c。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLObject*	object_meta	输出检测结果元数据

## 5.14 TDL\_FaceDetection

**【语法】**

```
int32_t TDL_FaceDetection(TDLHandle handle,
                          const TDLModel model_id,
                          TDLImage image_handle,
                          TDLFace *face_meta);
```

**【描述】**

执行人脸检测并返回人脸检测结果元数据，详细实例可以参考 tdl\_sdk/sample/c/sample\_face\_detection.c。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLFace*	face_meta	输出人脸检测结果元数据

## 5.15 TDL\_FaceAttribute

### 【语法】

```
int32_t TDL_FaceAttribute(TDLHandle handle,  
                           const TDLModel model_id,  
                           TDLImage image_handle,  
                           TDLFace *face_meta);
```

### 【描述】

执行人脸属性分析，需配合已检测到的人脸框进行特征分析，该模型的内部会对图像进行裁剪以提高精度，因此在调用此接口前，最好先调用 TDL\_FaceDetection 以获取人脸框图，详细实例可以参考 tdl\_sdk/sample/c/sample\_face\_attribute.c，如果 face\_meta 中没有框图数据，则不进行裁剪操作。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输入/输出	TDLFace*	face_meta	输入人脸检测结果，输出补充属性信息

## 5.16 TDL\_FaceLandmark

### 【语法】

```
int32_t TDL_FaceLandmark(TDLHandle handle,  
                           const TDLModel model_id,  
                           TDLImage image_handle,  
                           TDLImage *crop_image_handle,  
                           TDLFace *face_meta);
```

### 【描述】

执行人脸关键点检测，在已有的人脸检测结果上补充关键点坐标，该模型的内部会对图像进行裁剪以提高精度，因此在调用此接口前，最好先调用 TDL\_FaceDetection 以获取人脸框图，详细实例可以参考 tdl\_sdk/sample/c/sample\_face\_recognition.c，如果 face\_meta 中没有框图数据，则不进行裁剪操作。

## 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLImage	crop_image_handle	TDLImage 对象, 裁剪后的图像, 为 NULL 时不生效
输入/输出	TDLFace*	face_meta	输入人脸检测结果, 输出补充关键点坐标

## 5.17 TDL\_Classification

## 【语法】

```
int32_t TDL_Classification(TDLHandle handle,
                           const TDLModel model_id,
                           TDLImage image_handle,
                           TDLClassInfo *class_info);
```

## 【描述】

执行通用分类识别, 包含有活体识别, 语音识别和手势识别等, 具体可以参考 tdl\_sdk/sample/c/sample\_classification.c。

## 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLClassInfo*	class_info	输出分类结果

## 5.18 TDL\_InstanceSegmentation

## 【语法】

```
int32_t TDL_InstanceSegmentation(TDLHandle handle,
                                  const TDLModel model_id,
                                  TDLImage image_handle,
                                  TDLInstanceSeg *inst_seg_meta);
```

## 【描述】

执行实例分割 (Instance Segmentation), 检测图像中每个独立目标的像素级轮廓, 具体可以参考 tdl\_sdk/sample/c/sample\_instance\_segmentation.c。

## 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLInstanceSeg*	inst_seg_meta	输出实例分割结果（包含 mask 和 bbox）

## 5.19 TDL\_SemanticSegmentation

### 【语法】

```
int32_t TDL_SemanticSegmentation(TDLHandle handle,
                                  const TDLModel model_id,
                                  TDLImage image_handle,
                                  TDLSegmentation *seg_meta);
```

### 【描述】

执行语义分割（Semantic Segmentation），对图像进行像素级分类，具体可以参考 `tdl_sdk/sample/c/sample_semantic_segmentation.c`。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLSegmentation*	seg_meta	输出分割结果（类别标签图）

## 5.20 TDL\_FeatureExtraction

### 【语法】

```
int32_t TDL_FeatureExtraction(TDLHandle handle,
                               const TDLModel model_id,
                               TDLImage image_handle,
                               TDLFeature *feature_meta);
```

### 【描述】

提取图像的深度特征向量，为了提高检测精度，输入的 image 图像最好是经过裁剪或对齐等方式处理后的图像，可以参考 `tdl_sdk/sample/c/sample_face_recognition.c`。

### 【参数】



	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLFeature*	feature_meta	输出特征向量

## 5.21 TDL\_LaneDetection

### 【语法】

```
int32_t TDL_LaneDetection(TDLHandle handle,  
                           const TDLModel model_id,  
                           TDLImage image_handle,  
                           TDLLane *lane_meta);
```

### 【描述】

检测道路车道线及其属性，具体可以参考 `tdl_sdk/sample/c/sample_lane_detection.c`。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLLane*	lane_meta	输出车道线坐标及属性

## 5.22 TDL\_Tracking

### 【语法】

```
int32_t TDL_Tracking(TDLHandle handle,  
                      const TDLModel model_id,  
                      TDLImage image_handle,  
                      TDLObject *object_meta,  
                      TDLTracker *tracker_meta);
```

### 【描述】

多目标跟踪，基于检测结果进行跨帧目标关联，具体可以参考 `tdl_sdk/sample/c/sample_tracking.c`。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输入/输出	TDLObject*	object_meta	输入检测结果，输出补充跟踪 ID
输出	TDLTracker*	tracker_meta	输出跟踪器状态信息

## 5.23 TDL\_SetSingleObjectTracking

### 【语法】

```
int32_t TDL_SetSingleObjectTracking(TDLHandle handle,
                                     TDLImage image_handle,
                                     TDLObject *object_meta,
                                     int *set_values,
                                     int size);
```

### 【描述】

单目追踪设置追踪目标，根据场景要和 TDL\_Detection 配合使用，具体可以参考 tdl\_sdk/sample/c/camera/sample\_vi\_single\_object\_tracking.c。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	TDLImage	image_handle	TDLImage 对象
输入	TDLObject*	object_meta	当前帧检测结果
输入/输出	int*	set_values	追踪目标。支持以下 3 种形式：1. 传入目标框坐标 (x1, y1, x2, y2)；2. 传入图像中某个点的位置 (x, y)，(此时 object_meta size 不能为 0)；3. 传入 object_meta 中某个目标的索引，(此时 object_meta size 不能为 0)
输入	int	size	set_values 元素个数 (只能为 1 或 2 或 4)

## 5.24 TDL\_SingleObjectTracking

### 【语法】

```
int32_t TDL_SingleObjectTracking(TDLHandle handle,
                                  TDLImage image_handle,
                                  TDLTracker *track_meta,
                                  uint64_t frame_id);
```

**【描述】**

执行单目追踪，具体可以参考 `tdl_sdk/sample/c/camera/sample_vi_single_object_tracking.c`。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	TDLImage	image_handle	TDLImage 对象
输入	TDLTracker*	track_meta	追踪结果
输入	uint64_t	frame_id	帧 id

## 5.25 TDL\_CharacterRecognition

**【语法】**

```
int32_t TDL_CharacterRecognition(TDLHandle handle,
                                const TDLModel model_id,
                                TDLImage image_handle,
                                TDLOcr *char_meta);
```

**【描述】**

字符识别，支持文本检测与识别，为了提高检测精度，输入的 image 图像最好是经过裁剪或对齐等方式处理后的图像，可以参考 `tdl_sdk/sample/c/sample_licence_recognition.c`。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	模型类型枚举
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLOcr*	char_meta	输出识别结果（文本内容和位置）

## 5.26 TDL\_LoadModelConfig

**【语法】**

```
int32_t TDL_LoadModelConfig(TDLHandle handle,
                             const char *model_config_json_path);
```

**【描述】**

加载模型配置信息，加载后可以仅通过模型 id 去打开模型。

**【参数】**

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	model_config_json	模型配置文件路径, 如果为 NULL, 默认使用 configs/model/model_config.json

## 5.27 TDL\_SetModelDir

### 【语法】

```
int32_t TDL_SetModelDir(TDLHandle handle,
                        const char *model_dir);
```

### 【描述】

设置模型文件夹路径, 该文件夹下的子文件夹必须要以 cv181x 等平台名称命名, 在子文件夹中放置模型文件。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	model_dir	为 tdl_models 仓库路径 (下面各平台的子文件夹)

## 5.28 TDL\_SetModelThreshold

### 【语法】

```
int32_t TDL_SetModelThreshold(TDLHandle handle,
                              const TDLModel model_id,
                              float threshold);
```

### 【描述】

设置模型的阈值, 设置范围为 (0, 1)。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	要设置的模型类型枚举值
输入	float	threshold	模型的阈值

## 5.29 TDL\_IspClassification

### 【语法】

```
int32_t TDL_IspClassification(TDLHandle handle,
                              const TDLModel model_id,
                              TDLImage image_handle,
                              TDLIspMeta *isp_meta,
                              TDLClass *class_info);
```

### 【描述】

执行 ISP 图像分类任务。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	指定目标分类模型类型枚举值
输入	TDLImage	image_handle	TDLImage 对象
输入	TDLIspMeta*	isp_meta	输入参数，包含 isp 相关的数据
输出	TDLClass*	class_info	输出参数，存储目标分类结果

## 5.30 TDL\_Keypoint

### 【语法】

```
int32_t TDL_Keypoint(TDLHandle handle,
                     const TDLModel model_id,
                     TDLImage image_handle,
                     TDLKeypoint *keypoint_meta);
```

### 【描述】

执行关键点检测任务，为了提高检测精度，输入的 image 图像最好是经过裁剪处理后的图像，具体例子可以参考 `tdl_sdk/sample/c/sample_keypoints.c`

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	指定关键点检测模型类型枚举值
输入	TDLImage	image_handle	TDLImage 对象
输出	TDLKeypoint*	keypoint_meta	输出参数，存储检测到的关键点坐标及置信度

## 5.31 TDL\_DetectionKeypoint

### 【语法】

```
int32_t TDL_DetectionKeypoint(TDLHandle handle,
                              const TDLModel model_id,
                              TDLImage image_handle,
                              TDLObject *object_meta,
                              TDLImage *crop_image_handle);
```

### 【描述】

执行关键点检测任务（根据目标的坐标进行裁剪后再执行关键点检测），如果 object\_meta 中包含框图信息，则 api 内部会对 image 图像进行裁剪；因此为了提高图像精度，在使用该 api 前最好先调用 TDL\_Detection 以获得目标框图信息。具体可以参考 tdl\_sdk/sample/c/sample\_detect\_keypoints.c。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const TDLModel	model_id	指定关键点检测模型类型枚举值
输入	TDLImage	image_handle	TDLImage 对象
输入/输出	TDLObject*	object_meta	输出参数，存储检测到的关键点坐标及置信度
输出	TDLImage*	crop_image_handle	保存裁剪后的图像，以便后续操作，如果传入 NULL 则不保存

## 5.32 TDL\_IntrusionDetection

### 【语法】

```
int32_t TDL_IntrusionDetection(TDLHandle handle,
                                TDLPoints *regions,
                                TDLBox *box,
                                bool *is_intrusion);
```

### 【描述】

执行入侵检测，具体可以参考 tdl\_sdk/sample/c/sample\_intrusion\_detection.c。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	TDLPoints*	regions	背景区域点集数组
输入	TDLBox*	box	检测区域 bbox
输出	bool*	is_intrusion	输出参数，存储入侵检测结果

## 5.33 TDL\_MotionDetection

### 【语法】

```
int32_t TDL_MotionDetection(TDLHandle handle,
                             TDLImage background,
                             TDLImage detect_image,
                             TDLObject *roi,
                             uint8_t threshold,
                             double min_area,
                             TDLObject *obj_meta);
```

### 【描述】

执行移动侦测任务，具体可以参考 `tdl_sdk/sample/c/sample_motion_detection.c`。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	TDLImage	background	背景图像
输入	TDLImage	detect_image	检测图像
输入	TDLObject*	roi	检测区域
输入	uint8_t	threshold	阈值
输入	double	min_area	最小面积
输出	TDLObject*	obj_meta	输出参数，存储检测结果

## 5.34 TDL\_APP\_Init

### 【语法】

```
int32_t TDL_APP_Init(TDLHandle handle,
                     const char *task,
                     const char *config_file,
                     char ***channel_names,
                     uint8_t *channel_size);
```

### 【描述】

初始化 APP 任务，执行抓拍，客流量计数等复杂的场景，模型配置参数主要在 `config_file` 中配置，具体例子可以参考 `tdl_sdk/sample/c/camera/sample_vi_face_pet_cap.c`。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	task	APP 任务名称
输入	const char*	config_file	APP 的 json 配置文件路径, tdl 提供了模板, 路径在 tdl_sdk/sample/c/config
输出	char***	channel_names	每一路视频流的名称信息
输出	uint8_t*	channel_size	视频流的路数

## 5.35 TDL\_APP\_SetFrame

### 【语法】

```
int32_t TDL_APP_SetFrame(TDLHandle handle,
                          const char *channel_name,
                          TDLImage image_handle,
                          uint64_t frame_id,
                          int buffer_size);
```

### 【描述】

往 APP 场景送帧。

### 【参数】

	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	channel_name	当前 channel 的名称
输入	TDLImage	image_handle	TDLImage 对象
输入	uint64_t	frame_id	当前 TDLImage 对象的 frame id
输入	int	buffer_size	推理线程缓存的帧数

## 5.36 TDL\_APP\_Capture

### 【语法】

```
int32_t TDL_APP_Capture(TDLHandle handle,
                         const char *channel_name,
                         TDLCaptureInfo *capture_info);
```

### 【描述】

执行抓拍任务, 可以抓拍人脸, 宠物等目标, 主要依据配置文件决定, 配置文件在 TDL\_APP\_Init 中调用。

### 【参数】



	数据类型	参数名称	说明
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	channel_name	当前 channel 的名称
输出	TDLCaptureInfo*	capture_info	抓拍结果

## 5.37 TDL\_APP\_ObjectCounting

### 【语法】

```
int32_t TDL_APP_ObjectCounting(TDLHandle handle,  
                                const char *channel_name,  
                                TDLObjectCountingInfo *object_counting_info);
```

### 【描述】

执行客流统计 (TDL\_APP\_Init task 为 consumer\_counting) 或越界检测任务 (TDL\_APP\_Init task 为 cross\_detection)

### 【参数】

	数据类型	参数名称	描述
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	channel_name	当前通道名称
输出	TDLObjectCountingInfo*	object_counting_info	统计/检测结果

## 5.38 TDL\_APP\_ObjectCountingSetLine

### 【语法】

```
int32_t TDL_APP_ObjectCountingSetLine(TDLHandle handle,  
                                        const char *channel_name, int x1,  
                                        int y1, int x2, int y2, int mode);
```

### 【描述】

客流统计或越界检测运行过程中重新设置画线位置。

### 【参数】

	数据类型	参数名称	描述
输入	TDLHandle	handle	TDLHandle 对象
输入	const char*	channel_name	当前通道名称
输入	int	x1	端点 1 横坐标
输入	int	y1	端点 1 纵坐标
输入	int	x2	端点 2 横坐标
输入	int	y2	端点 2 纵坐标
输入	int	mode	对于客流统计: mode 为 0 时, 对于竖直线, 从左到右为进入, 对于非竖直线, 从上到下为进入, mode 为 1 相反。对于越界检测: mode 为 0 时, 对于竖直线, 从左到右为越过, 对于非竖直线, 从上到下为越过, mode 为 1 相反, mode 为 2 双向检测

## 5.39 TDL\_WrapImage

### 【语法】

```
int32_t TDL_WrapImage(TDLImage image,  
                      void *frame);
```

### 【描述】

将 TDLImage 包装为 VIDEO\_FRAME\_INFO\_S 对象。

### 【参数】

	数据类型	参数名称	描述
输入	TDLImage	image	TDLImage 对象
输出	VIDEO_FRAME_INFO_S*	frame	输出参数, 存储包装后的帧信息

## 5.40 TDL\_LLMApiCall

### 【语法】

```
int32_t TDL_LLMApiCall(TDLHandle handle, const char *client_type,  
                      const char *method_name, const char *params_json,  
                      char *result_buf, size_t buf_size)
```

### 【描述】

向指定的 LLM 客户端发起调用请求, 具体例子可以参考 tdl\_sdk/sample/c/sample\_llm\_api.c。

### 【参数】

	数据类型	参数名称	描述
输入	TDLHandle	handle	TDL_CreateHandle 返回的上下文句柄
输入	const char*	client_type	LLM 客户端类型
输入	const char*	method_name	调用的接口方法名
输入	const char*	params_json	请求体的 json 字符串
输入	size_t	buf_size	result_buf 可用空间大小
输出	char*	result_buf	存放调用返回的 JSON 结果或错误信息

# 6 TDL\_SDK 模型部署方式

借助 TDL\_SDK 框架，可以使新模型集成变得更加简单，框架可以帮助完成如下工作：

- 图像预处理，用户只需配置预处理参数，框架内会自动调用预处理设备实现预处理
- 模型推理，无需编写任何推理相关代码
- 内存管理，框架内会自动管理内存，用户无需担心内存泄漏问题

## 6.1 现有模型类添加新模型文件

现有模型在已在第三章中展示，如需添加新的模型文件，只是需要在模型工厂中添加对应的新模型 ID 方便调用，操作步骤如下：

1. 在 `tdl_sdk/install/CV184X/include/nn/tdl_model_list.h` 中添加新模型的 `model_id`；
2. 在 `tdl_sdk/src/components/nn/tdl_model_factory.cpp` 中添加新模型的创建函数；
3. 在 `tdl_sdk/configs/model/model_factory.json` 中添加新模型的配置信息。

## 6.2 集成新的模型类型

1. 在 `tdl_sdk/src/components/nn` 目录下，根据新模型的任务类型，选择合适的文件夹，假如没有匹配的，就新建一个该任务类型的文件夹；
2. 在文件夹内添加新模型的头文件和源文件，头文件派生自 `tdl_sdk/include/framework/model/base_model.hpp`，源文件实现函数 `tdl_sdk/include/framework/model/base_model.hpp`；
3. 在 `tdl_sdk/include/nn/tdl_model_list.h` 创建新的模型 ID；
4. 在 `tdl_sdk/src/components/nn/tdl_model_factory.cpp` 中添加新模型的创建函数；
5. 在 `tdl_sdk/configs/model/model_factory.json` 中添加新模型的配置信息；
6. YOLO 系列模型编译可以参考文档《YOLO 系列开发指南》，其他类型模型，直接按照标准流程进行编译。

# 7 TDL\_SDK c sample 使用方式

以下例子均以 cv181x 的板端为例，使用模型请更新实际情况替换。

## 7.1 sample\_character\_recognition

用于车牌识别场景，支持输入车牌图片，输出识别到的车牌字符文本信息。运行方式如下：

```
./sample_character_recognition --m ./cv181x/recognition_license_plate_24_96_MIX_cv181x.  
→cvimodel -i ./license_plate_keypoints_0.jpg  
// -m 输入模型 -i 输入图像
```

输入图像为一张车牌的图片，运行结果将会在串口打印 txt info: 闽 D999PN



## 7.2 sample\_classification

用于声音分类和图像分类场景，支持输入音频 bin 文件或图片文件，输出其类别信息和置信度分数。音频识别的运行方式如下：

```
./sample_classification -m ./cv181x/baby_cry_cnn10_188_40_INT8_cv181x.cvimodel -b ./test_  
→inputs/laugh_1_m4a_3_1.bin -r 16000 -t 3  
// -m 输入模型 -b 输入音频bin文件 -r 采样率 -t 音频时长
```

输入一段音频数据，以及音频数据的采样率和时长，例程将会对音频进行识别，并输出其分数。

图片识别的运行方式如下：

```
./sample_classification -m ./cv181x/cls_hand_gesture_128_128_INT8_cv181x.cvimodel -i ./test_  
→inputs/hand_two.jpg  
// -m 输入模型 -i 输入图像
```

这里以手势识别为例，输入一张手势图像，将会输出手势以及他的分数。具体的串口打印如下

```
pred_lable: 0, score = 0.964321
```

## 7.3 sample\_detect\_keypoints

用于多目标关键点检测场景，支持输入单张图片，输出手部、车牌或人体姿态的关键点坐标信息。运行方式如下：

```
./sample_detect_keypoints -m./cv181x/hand_detection_yolov8n_mv3_050_INT8_cv181x.cvimodel,.  
→/cv181x/keypoint_hand_128_128_INT8_cv181x.cvimodel -i ./test_inputs/hand.jpg -o out_d.jpg,  
→out_c.jpg  
// -m 输入模型，这个sample需要输入两个模型，分别是检测类模型和关键点类的模型  
// -i 输入图像  
// -o 输出图像，可以不输入。输出两张图像，一张是检测画框的原图，一张是裁剪后的图片
```

这里输入一张手部图像，算法将会识别它的关键点，这里的关键点为比例系数，乘上对应的长宽即为坐标点。

```
obj_meta id: 0, [x, y]: 0.667969, 0.796875  
obj_meta id: 0, [x, y]: 0.726562, 0.683594  
obj_meta id: 0, [x, y]: 0.683594, 0.570312  
obj_meta id: 0, [x, y]: 0.578125, 0.531250  
obj_meta id: 0, [x, y]: 0.484375, 0.523438  
obj_meta id: 0, [x, y]: 0.578125, 0.451172  
obj_meta id: 0, [x, y]: 0.546875, 0.314453  
obj_meta id: 0, [x, y]: 0.539062, 0.225586  
obj_meta id: 0, [x, y]: 0.523438, 0.153320  
obj_meta id: 0, [x, y]: 0.484375, 0.500000  
obj_meta id: 0, [x, y]: 0.345703, 0.386719  
obj_meta id: 0, [x, y]: 0.250000, 0.314453  
obj_meta id: 0, [x, y]: 0.168945, 0.257812  
obj_meta id: 0, [x, y]: 0.427734, 0.570312  
obj_meta id: 0, [x, y]: 0.322266, 0.500000  
obj_meta id: 0, [x, y]: 0.427734, 0.539062  
obj_meta id: 0, [x, y]: 0.500000, 0.578125  
obj_meta id: 0, [x, y]: 0.386719, 0.652344  
obj_meta id: 0, [x, y]: 0.330078, 0.597656  
obj_meta id: 0, [x, y]: 0.417969, 0.605469  
obj_meta id: 0, [x, y]: 0.484375, 0.628906
```

依据关键点对图像进行标记的图片如下：



## 7.4 sample\_keypoints

用于多类型关键点检测，支持输入单张图片，输出手部/车牌/人体姿态的关键点坐标信息及可视化标注图像，与 sample\_detect\_keypoints 不同的是，这个接口需要输入一张已经裁剪后的图像。运行方式如下：

```
./sample_keypoints -m ./cv181x/keypoint_hand_128_128_INT8_cv181x.cvimodel -i ./test_inputs/  
→hand.jpg -o out.jpg  
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张手部图像，算法将会识别它的关键点，依据关键点对图像进行标记的图片如下：



## 7.5 sample\_face\_attribute

用于人脸检测与属性分析场景，支持输入单张图片，输出人脸位置及性别、年龄、眼镜、口罩等属性信息。运行方式如下：

```
./sample_face_attribute -m ./cv181x/scrfd_500m_bnkps_432_768.cvimodel,./cv181x/face_attribute_  
→cls.cvimodel -i ./test_inputs/face.jpg  
// -m 输入模型，这个sample需要输入两个模型，分别是人脸检测和人脸属性识别  
// -i 输入图像
```

这里输入一张人脸图像，算法将识别人脸的各个属性输入图像和结果如下所示：



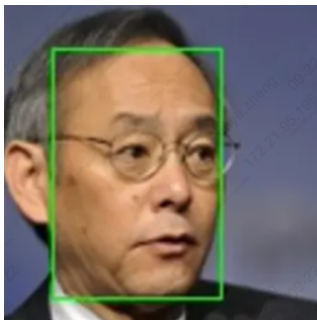
```
gender score: 0.992188, age score: 0.640625, glass score: 0.992188, mask score: 0.000000  
Gender:Male  
Age:Male  
Glass:Yes  
Mask:no
```

## 7.6 sample\_face\_detection

用于通用人脸检测，支持输入单张图片，输出带有人脸检测框标记的结果图像，并框输出人脸框的左上角的 xy 坐标和右下角的 xy 坐标，和置信度分数。运行方式如下：

```
./sample_face_detection -i ./test_inputs/face.jpg -m ./cv181x/scrfd_det_face_432_768_INT8_  
→cv181x.cvimodel -o ./test_inputs/face_det.jpg  
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张人脸图片，依据输出框图对人脸进行标记



## 7.7 sample\_feature\_extraction

用于人脸比对场景，支持输入两张人脸图片，输出其相似度分数及中间处理结果图像。运行方式如下：

```
./sample_face_landmark -m ./cv181x/keypoint_face_v2_64_64_INT8_cv181x.cvimodel -i ./test_  
→inputs/face.jpg  
// -m 输入模型 -i 输入图像
```

这里输入两个图像，用于特征提取后的对比，输入两个一样的图像时，相似度为 1。



```
similarity is 1.000000
```

## 7.8 sample\_face\_landmark

用于人脸关键点检测场景，输出图像的模糊度得分以及五个人脸关键点坐标点。运行方式如下：

```
./sample_face_landmark -m ./cv181x/keypoint_face_v2_64_64_INT8_cv181x.cvimodel -i ./test_
→inputs/face.jpg
// -m 输入模型 -i 输入图像
```

依据输入图像，串口将输出关键点信息

```
landmarks id : 0, landmarks x : 43.531250, landmarks y : 51.625000
landmarks id : 1, landmarks x : 73.500000, landmarks y : 51.625000
landmarks id : 2, landmarks x : 63.875000, landmarks y : 65.187500
landmarks id : 3, landmarks x : 45.937500, landmarks y : 82.250000
landmarks id : 4, landmarks x : 68.687500, landmarks y : 81.375000
```

## 7.9 sample\_instance\_segmentation

用于实例分割场景，输入一张图片数据，输出各个实例的坐标框和轮廓坐标点。运行方式如下：

```
./sample_instance_segmentation -i ./test_inputs/coco.jpg -o ./test_inputs/coco_o.jpg -m ./cv181x/
→segmentation_yolov8n_640_640_INT8_cv181x.cvimodel
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张图片，算法将会对其进行分割



## 7.10 sample\_lane\_detection

用于车道线检测场景，支持输入道路图像，输出检测到的车道线坐标信息。运行方式如下：

```
./sample_lane_detection -m ./cv181x/lstr_det_lane_360_640_MIX_cv181x.cvimodel -i ./test_
→inputs/result.jpg -o ./out.jpg
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张车道线的图片，串口将会打印车道线信息：

```
lane 0
0: 506.925018 432.000031
1: 362.094147 576.000000
lane 1
0: 815.652710 432.000031
1: 1038.046509 576.000000
```

依据结果对图像进行标记：



## 7.11 sample\_object\_detection

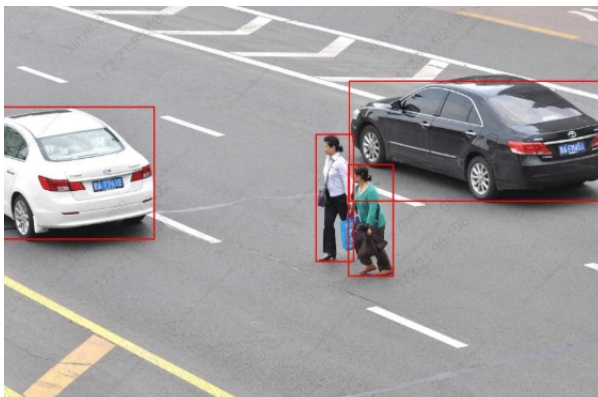
用于通用目标检测 sample，输出目标框的左上角的 xy 坐标和右下角的 xy 坐标，类别和置信度，具体适配哪些模型可参看 sample\_object\_detection.c 中的 get\_od\_model\_info 进行参看。运行方式如下：

```
./sample_object_detection -m ./cv181x/coco80_detection_yolov10n_640_640_INT8_cv181x.
→cvimodel -i ./test_inputs/coco.jpg -o ./out.jpg
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张图片，串口将会打印识别到的物体类别及框图

```
obj_meta_index : 0, class_id : 0, score : 0.943986, boxx : [2.527201 327.880676 226.529968 507.681152]
obj_meta_index : 1, class_id : 0, score : 0.918415, boxx : [740.277466 1285.450562 171.033188 427.
→008545]
obj_meta_index : 2, class_id : 4, score : 0.850404, boxx : [738.621460 834.782410 348.960632 586.
→001221]
obj_meta_index : 3, class_id : 4, score : 0.800627, boxx : [669.198853 750.732361 284.652740 555.
→387817]
```

依据结果对图像进行标记：



## 7.12 sample\_pose

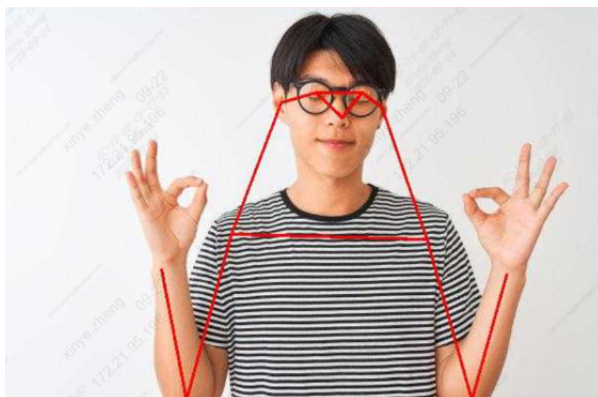
用于人体姿态估计，支持输入单张图片，输出 17 个人体关键点坐标及骨架连接信息，并生成带关键点和骨架标注的可视化结果图。运行方式如下：

```
./sample_pose -m ./cv181x/keypoint_yolov8pose_person17_384_640_INT8_cv181x.cvimodel -i ./
→test_inputs/person.jpg -o ./test_inputs/person_out.jpg
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张图片，串口将会姿态的坐标点和分数

```
obj_meta_index : 0, class_id : 0[79270.104763], score : 0.934758, bbox : [45.138302 249.390228 102.
→585884 354.314575]
pose : 0: 142.950653 160.520309 0.980407
pose : 1: 149.029053 148.363525 0.924847
pose : 2: 124.715469 148.363525 0.962615
pose : 3: 161.185822 154.441910 0.573295
pose : 4: 100.401848 166.598709 0.814048
pose : 5: 197.656219 221.304276 0.962615
pose : 6: 82.166672 245.617874 0.986373
pose : 7: 228.048203 324.637054 0.802613
pose : 8: 63.931477 336.793823 0.924847
pose : 9: 136.872238 342.872223 0.778173
pose : 10: 88.245079 257.774658 0.901571
pose : 11: 191.577805 379.342621 0.573295
pose : 12: 112.558662 391.499390 0.676617
pose : 13: 161.185822 403.656189 0.070180
pose : 14: 100.401848 397.577789 0.098429
pose : 15: 100.401848 391.499390 0.012670
pose : 16: 106.480255 367.185822 0.015762
```

依据结果对图像进行标记：

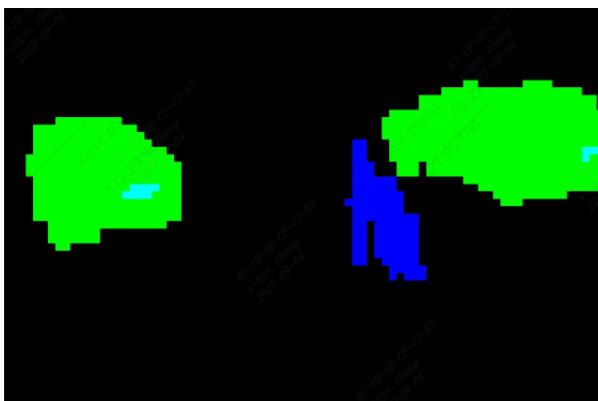


### 7.13 sample semantic segmentation

用于语义分割场景，输入一张图片数据，输出各个像素点的类别信息。运行方式如下：

```
./sample_semantic_segmentation -m ./cv181x/topformer_seg_person_face_vehicle_384_640_INT8_
↪cv181x.cvimodel -i ./test_inputs/topformer.jpg -o out.jpg
// -m 输入模型 -i 输入图像 -o 输出图像，可以不输入
```

这里输入一张图片，串口将会打印图像各个像素图点对应的类别

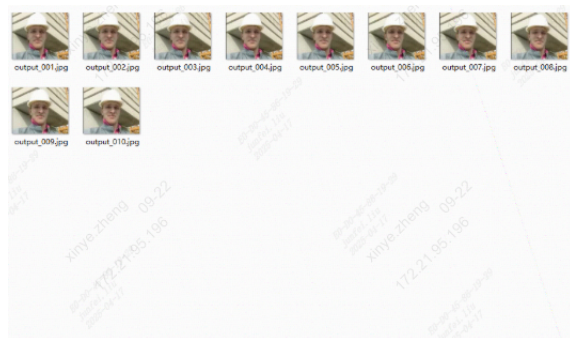
[illegible]

## 7.14 sample\_tracking

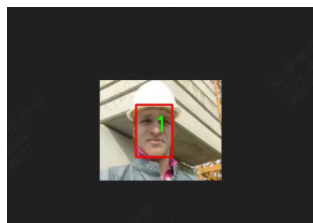
对输入的图片组进行追踪。运行方式如下：

```
./sample_tracking -m ./cv181x/scrfd_det_face_432_768_INT8_cv181x.cvimodel,./cv181x/mbv2_
→det_person_512_896_INT8_cv181x.cvimodel -i input -o output
// -m 输入模型，这个sample需要输入两个个模型，分别是人脸检测和行人检测
// -i 输入图像文件夹，文件夹下包含了一组图片，图片需要按照xxx_001.jpg, xxx_002.
→jpg这样的顺序排列，如下图
// -o 输出图像文件夹
```

输入的图片组示例：



输入的图片组示例：



## 7.15 sample\_face\_recognition

识别人脸并进行特征匹配，人脸检测-> 人脸特征点提取-> 人脸特征值提取-> 特征点比较。运行方式如下：

```
./sample_face_recognition -m ./cv181x/scrfd_det_face_432_768_INT8_cv181x.cvimodel,./cv181x/
→keypoint_face_v2_64_64_INT8_cv181x.cvimodel,./cv181x/feature_cviface_112_112_INT8_
→cv181x.cvimodel -i ./test_inputs/face.jpg,./test_inputs/face.jpg -c ./model_factory.json
// -m 输入模型，这个sample需要输入三个模型，分别是人脸检测，人脸特征点，人脸特征值
// -i 输入图像，这个sample需要输入两张图像，用于提取特征值后的人脸比对
// -c config参数，记录了模型的参数设置
```

输入两张一样的图像，得到的特征值相似度一致

```
similarity is 1.000000
```



## 7.16 sample\_licence\_recognition

识别车牌，检测-> 关键点-> 识别，支持一张图片多个车牌。运行方式如下：

```
./sample_licence_recognition -m ./cv181x/yolov8n_det_license_plate_384_640_INT8_cv181x.  
→cvimodel,./cv181x/keypoint_license_plate_64_128_INT8_cv181x.cvimodel,./cv181x/recognition_  
→license_plate_24_96_MIX_cv181x.cvimodel -i ./test_inputs/out_vertical_cv.jpg  
// -m 输入模型，这个sample需要输入三个模型，分别是车牌检测，车牌关键点，车牌识别  
// -i 输入图像
```

这里输入一张包含两个车牌的图像：



串口将有如下打印：

```
id = 0, txt info: 闽D999PN  
id = 1, txt info: 闽D999PN
```

## 7.17 sample\_vi\_detection

连接摄像头进行物体识别。将 sensor\_cfg.ini(sensor 的配置文件可以联系开发者获取) 放置在/mnt/data 目录下，并使用下列命令运行：

```
./sample_vi_detection -m ./cv181x/yolov8n_det_hand_384_640_INT8_cv181x.cvimodel -c 0  
// -m 输入模型，这里的例子为手部检测  
// -c vi的chn，单sensor的情况下为0  
// -f config参数，记录了模型的参数设置，可以不输入
```

这里以手部检测为例，将会打印出检测到的手部的信息

```
obj_meta_index : 0, class_id : 0, score : 0.861538, bbox : [260.416077 601.567139 103.850037 312.  
→489624]
```

## 7.18 sample\_vi\_face\_pet\_cap

连接摄像头进行人脸宠物抓拍，如果有输入-g 的参数，则会进行特征点识别，如果有-o 的参数，则将抓拍的图像保存下来。

```
//json文件解析，在使用的过程中，model_dir和model_config一定要根据实际情况修改
{
  "model_dir": "/mnt/sd/models/", //
  →模型的路径，后面会自动匹配cv181x, cv184x等，因此实际的模型路径为/mnt/sd/models/
  →cv181x，这里一定要将模型先放置在cv181x（依据板端类型修改）的文件夹下，再放置在model_
  →dir的路径下
  "model_config": "/mnt/sd/configs/model/model_factory.json", //模型的参数文件
  "frame_buffer_size": 1,
  "pipelines": [
    {
      "name": "face_pet_cap",
      "nodes": {
        "object_detection_node": {
          "config_thresh": 0.5
        },
        "track_node": {
          "fuse_track": true
        },
        "snapshot_node": { //抓拍参数
          "snapshot_interval": 5,
          "min_snapshot_size": 40,
          "crop_size_min": 128,
          "crop_size_max": 256,
          "snapshot_quality_threshold": 0,
          "crop_square": true
        }
      }
    }
  ]
}
```

gallery 文件夹在 tdl 中不提供，需要自己生成，注意必须是 0.bin, 1.bin……这样的格式。特征值可以使用 TDL\_FeatureExtraction 获取（参数上述的 TDL\_FeatureExtraction），也可以在本 sample 中的 TDLCaptureInfo 结构体中获取，在 TDL\_APP\_Capture 后，如果识别到人脸，则 TDLCaptureInfo 中的 TDLFeature 参数中的 prt 会记录特征值，将其保存为文件，下图为参考

```

int ret = TDL_APP_Capture(pstArgs->tdl_handle, pstArgs->channel_names[i],
                          &capture_info);

if (ret == 1) {
    continue;
} else if (ret == 2) {
    to_exit = true;
    break;
} else if (ret != 0) {
    printf("TDL_APP_Capture failed with %x\n", ret);
    goto exit0;
}

if (fwm_diff > 30) {
    printf("detect person size: %d, pet size: %d\n",
          capture_info.person_meta.size, capture_info.pet_meta.size);

    for (uint32_t j = 0; j < capture_info.snapshot_size; j++) {
        printf("snapshot[%d]: male:%d,glass:%d,age:%d,emotion:%s\n", j,
              capture_info.snapshot_info[j].male,
              capture_info.snapshot_info[j].glass,
              capture_info.snapshot_info[j].age,
              emotionStr[capture_info.snapshot_info[j].emotion]);
    }

    if (capture_info.features->ptr != NULL) {
        FILE *fp = fopen(path, "wb");
        fwrite(capture_info.features->ptr, 1, capture_info.features->size, fp);
        fclose(fp);
        system("sync");
    }
}

```

将 sensor\_cfg.ini(sensor 的配置文件可以联系开发者获取) 放置在/mnt/data 目录下, 并使用下列命令运行

```

./sample_vi_face_pet_cap -c ./config/face_pet_cap_app.json -v 0 -g ./ipcamera/gallery/ -o out
// -m 配置json文件, 记录了要使用哪些功能
// -v vi的chn, 单sensor的情况下为0
// -g 特征值数据库, 可以不输入
// -o 输出文件夹, 可以不输入

```

## 7.19 sample\_vi\_consumer\_counting

连接摄像头进行客流量统计。运行方式如下:

```
./sample_vi_consumer_counting -c ./config/consumer_counting_app_vi.json -v 0
```

输出结果如下:

```

enter: 0, miss: 0
+++++ frame:124, infer time:40.00, fps: 25.00

```

## 7.20 sample\_vi\_cross\_detection

连接摄像头进行入侵检测。运行方式如下:

```
./sample_vi_cross_detection -c ./config/cross_detection_app_vi.json -v 0
```

输出结果如下:

```

cross num: 0
+++++ frame:62, infer time:40.00, fps: 25.00

```



## 7.21 sample\_vi\_single\_object\_tracking

连接摄像头进行单目追踪，开始运行程序时 TDL\_Detection 对目标进行检测。当需要对某个物体或某块区域进行追踪时，按下 I 或 i 即可输入参数。可以输入 box 的 id (范围为 0 - bj\_meta.size), box 的坐标 (x1,y1,x2,y2) 或者是一个坐标点 (x,y); 输入完成后对目标进行追踪，当目标丢失超过 5 秒后，恢复 TDL\_Detection 检测阶段。运行方式如下：

```
./sample_vi_single_object_tracking -d ./cv181x/yolov8n_det_person_vehicle_384_640_INT8_  
→cv181x.cvimodel -s ./cv181x/tracking_feartrack_128_128_256_256_INT8_cv181x.cvimodel  
// -d 检测类模型 -s 追踪类模型
```

输出结果如下：

```
Usage: input i or I to start tracking .....  
Enter bbox x1,y1,x2,y2 or a point x,y or bbox index to track:  
0 //追踪id为0的目标  
values[0] = 0  
The target has been lost for more than 5 seconds, switching to detection state //  
→当目标丢失超过5s时，回到检测状态  
Usage: input i or I to start tracking .....
```

## 7.22 sample\_motion\_detection

移动检测，输入一张背景图像和检测图像。运行方式如下：

```
./sample_motion_detection ./test_inputs/ir.jpg ./test_inputs/ir.jpg
```

输出结果如下：

```
Running motion detection with images:  
Background: ./test_inputs/ir.jpg  
Detect: ./test_inputs/ir.jpg  
Setting ROI regions...  
Begin motion detection...  
Detection completed
```

## 7.23 sample\_intrusion\_detection

入侵检测。运行方式如下：

```
./sample_intrusion_detection
```

输出结果如下：

```
=====矩形区域测试=====  
===== 矩形区域 =====  
点数: 4
```

(下页继续)

(续上页)

点 0: (100.00, 100.00)

点 1: (300.00, 100.00)

点 2: (300.00, 200.00)

点 3: (100.00, 200.00)

内部边界框检测: 入侵

外部边界框检测: 未入侵

=====凹多边形区域测试=====

===== 凹多边形区域 =====

点数: 6

点 0: (100.00, 100.00)

点 1: (200.00, 50.00)

点 2: (300.00, 100.00)

点 3: (250.00, 150.00)

点 4: (200.00, 120.00)

点 5: (150.00, 150.00)

内部边界框检测: 入侵

外部边界框检测: 未入侵

测试完成!

# 8 常见问题

## 8.1 模型打开失败类问题

### 1. 缺乏 model\_config.json

```
[tdl_model_factory.cpp:80] [I] input model config file is empty, load model config from /configs/model/  
→model_factory.json  
[tdl_model_factory.cpp:87] [E] model config file not found: /configs/model/model_factory.json  
[tdl_model_factory.cpp:238] [E] model config not found for model type: YOLOV8N_DET_PERSON_  
→VEHICLE  
[tdl_model_factory.cpp:248] [I] getModelInstance model_type:YOLOV8N_DET_PERSON_VEHICLE  
[tdl_model_factory.cpp:526] [I] createObjectDetectionModel success,model type:5,category:0  
[tdl_model_factory.cpp:831] [I] model_path: ./cv184x/yolov8n_det_person_vehicle_384_640_INT8_  
→cv181x.cvimodel  
[tdl_model_factory.cpp:238] [E] model config not found for model type: TRACKING_FEARTRACK  
[tdl_model_factory.cpp:248] [I] getModelInstance model_type:TRACKING_FEARTRACK  
[tdl_model_factory.cpp:831] [I] model_path: ./tracking_feartrack_128_128_256_256_INT8_cv181x.  
→cvimodel  
[base_model.cpp:159] [E] mean or std size is not 3  
[base_model.cpp:42] [E] Net setup failed
```

可以看出参数缺乏 mean 和 std, 这是因为在调用 TDL\_OpenModel 时没有传参 model\_config.json, 大部分专有模型是不需要传入的; 部分通用模型如特征提取、声音指令等需要传入模型配置信息, 可以参考 configs/model/model\_config.json。