

Orbbec SDK C++接口使用指南-v1.0

目录

Orbbec SDK C++接口使用指南-v1.0	1
1 简述	3
1.1 目的	3
1.2 术语	3
2. Orbbec SDK 数据流获取	3
2.1 创建 Context	3
2.2 注册设备断连回调	4
2.3 获取设备列表	4
2.4 打开设备	4
2.5 获取设备信息	4
2.6 创建 Pipeline	5
2.7 获取数据流	5
2.7.1 StreamProfile	5
2.7.2 获取 Depth 流	5
2.7.3 获取 IR 流	6
2.7.4 获取 Color 流	8
2.7.5 获取 Imu 数据	9
3 参数配置	10
3.1 参数配置步骤	10
3.1.1 参数类型	10
3.1.2 参数权限	11
3.1.3 最大值、最小值、步长、默认值	11
3.1.4 参数设置和获取	11
3.1.5 综合示例	12
4、设置彩色曝光时间。	12
3.2 常用参数配置	12
3.2.1 获取序列号	12
3.2.2 获取相机内外参	12
3.2.3 设置 Depth 参数	13
1. 硬件 D2D	14
2. 软件 D2D	14
1. 设置 Depth 最小值	15
2. 设置 Depth 最大值	15
1. 软件滤波开关	16
2. 软件滤波参数设置	16
3.2.4 设置 IR 参数	16
3.2.5 设置 Color 参数	18
3.2.6 设备重启	20
3.2.7 激光开关	20
3.2.8 LDP 开关	20
3.2.9 获取 LDP 近距离保护测量值	20
3.2.10 获取 LDP 保护状态	20
3.2.11 泛光灯开关	21
3.2.12 设备授时	21
3.2.12 网络设备 IP 地址修改设置	21
4 D2C	21
5 点云	22
5.1 Depth 点云	22
2、创建点云对象	22
3、获取相机内参	22
4、获取点云数据	22
5.2 RGBD 点云	23
2、创建点云对象	23
3、获取相机内参	23
4、获取点云数据	23
6 录像和回放	23
6.1 录像	23
6.2 录像回放	24
7 日志管理	24
7.1 设置日志等级	24
7.2 设置日志输出到控制台	25
7.3 设置日志输出到文件	25

7.4 设置日志回调输出	25
8 配置文件介绍	25
8.1 日志段	25
8.2 内存管理	26
8.3 Pipiline 参数配置	26
8.4 device 参数配置	27
9 多机同步	28
10 异常处理	28
11 结束	28

1 简述

1.1 目的

为了使用户正确高效在项目中快速接入 Orbbec SDK，防止在使用 Orbbec SDK 相关 API 接口的过程中由于不规范的调用而引起其他问题，故拟定此文档来规范 API 的调用。该文档参考 Orbbec SDK 1.8 ， 重点介绍 Orbbec SDK 中常用的功能。

1.2 术语

名称	说明
Firmware	3D 相机的固件
UnPack	USB 数据打包传输时，如 12bit 打包，在 SDK 解包，并转换到 16bit。主要目的是在 usb2.0 下，打包传输节省带宽。
Soft Filter	软件滤波是指使用软件算法对采集到的传感器数据进行处理，以去除噪声、平滑图像的过程。
视差	视差就是从有一定距离的两个点上观察同一个目标所产生的方向差异
D2D(视差转深度)	视差转深度是一种图像处理技术，用于将视差信息转换为深度信息。
硬件 D2D	视差转深度在设备内部实现，不占用上位机的算力
软件 D2D	视差转深度 在 Orbbec SDK 中实现
Depth 点云	Depth 点云，三维世界坐标系下的点的坐标，通过 Depth 相机的内参，可以将 Depth 转换成点云
RGBD 点云	叠加 RGB 信息的点云
D2C	Depth to Color ，这是一个对深度图像执行逐像素几何变换的功能，其结果是我们将深度图像通过 D2C 变换与其对应的彩色图像进行空间对齐,使得我们可以通过彩色图像中任意一个像素的图像坐标位置,在 D2C 后的深度图像中通过相同的图像坐标位置即可定位到此彩色像素所对应的深度信息。我们在 D2C 后生成一个与目标彩色图像相同大小的深度图像，图像内容是在彩色相机坐标系统下的深度数据。换句话说，重建了一个使用彩色相机的原点和尺寸“拍摄”的深度图像，其中每个像素都与彩色相机的对应像素坐标匹配
硬件 D2C	硬件 D2C 是指通过相机内部完成 D2C 功能，相机直接输出 D2C 后的结果
软件 D2C	通过 SDK 在上位机端完成 D2C 运算
帧汇聚(FrameSet)	将 Depth、IR、Color 帧合成 Frameset，通过 pipeline 调用
LDP	激光近距离保护

2. Orbbec SDK 数据流获取

2.1 创建 Context

接口描述	Context : Orbbec SDK 运行时的管理类，负责 SDK 资源申请和释放。具备枚举设备、监控设备重连以及启用多设备同步等功能的能力。
------	----------------------------------------------------------------------------------

头文件	#include "libobsensor/ObSensor.hpp"
具体实现	ob::Context ctx;

2.2 注册设备断连回调

接口描述	设置设备断连的回调函数，参数 callback 在设备插入和拔出时会触发，可以根据返回的新增设备列表和移除设备列表来管理设备。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	<pre>// register device callback ctx.setDeviceChangedCallback([](std::shared_ptr<ob::DeviceList> removedList, std::shared_ptr<ob::DeviceList> addedList) { });</pre>

2.3 获取设备列表

接口描述	主动查询已枚举的设备列表，返回 std::shared_ptr<DeviceList> ，指向设备列表类的智能指针。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();

2.4 打开设备

接口描述	获取指向设备(3D 相机)的智能指针 std::shared_ptr<Device> device ，获取设备有多种方式： 方式 1：通过指定枚举设备列表的索引号，获取指定设备。 方式 2：通过指定枚举设备列表的 SN 序列号，获取指定设备。 方式 3：通过指定枚举设备列表的 UID，获取指定设备。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	<pre>//方式 1： 从设备列表中获取设备，比如获取索引 0 的设备 std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDevice(0); //方式 2： 通过 SN 序列号，获取指定设备 //从设备列表中获取设备，比如获 SN 号为“AE4M73D0040”的设备，SN 存在于 USB 设 //备描述符中，如果 USB 设备描述符没有配置 SN 是无法获取指定 SN 的设备的。 std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDeviceBySN("AE4M73D0040"); //方式 3： 通过设备 UID，获取指定设备。 //从设备列表中获取设备，比如获 uid 为“NDSG3958LKHY45”的设备。 std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDeviceByUid("NDSG3958LKHY45");</pre>

2.5 获取设备信息

接口描述	获取指向设备(RGBD)相机的信息 std::shared_ptr<DeviceInfo> ，可以查询设备的 Pid、Vid 以及序列号等相关信息。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	auto deviceInfo = device->getDeviceInfo();

2.6 创建 Pipeline

接口描述	Pipeline 用于单设备或者多设备操作，需要通过 DeviceList 获取指定设备，并通过这个接口将设备和管道进行绑定。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDevice(0); std::shared_ptr<ob::Pipeline> pipeline(new ob::Pipeline(device));

2.7 获取数据流

在使用 ob::Pipeline(ob_device) 创建完成管道后，就可以通过它获取 Depth、IR、Color 中的指定数据流了。

2.7.1 StreamProfile

StreamProfile 是从 StreamProfileList 中获取的播放配置，Pipeline 开流和 Sensor 开流时需要设置 StreamProfile，Pipeline 和 Sensor 都提供了获取 StreamProfileList 的方法。

Pipeline	// 源码：include/libobsensor/hpp/Pipeline.hpp std::shared_ptr<StreamProfileList> getStreamProfileList(ObSensorType sensorType);
Sensor	// 源码：include/libobsensor/hpp/Sensor.hpp const std::shared_ptr<StreamProfileList> getStreamProfileList();

2.7.2 获取 Depth 流

接口描述	1. 通过 ob::Pipeline 获取 3D 相机的所有 Depth Profile 列表，包括流的分辨率、帧率和帧格式，返回 std::vector<std::shared_ptr<StreamProfile>>。 2. 获取想要输出的 Depth Profile。 3. 创建 ob::Config 来配置 ob::Pipeline 要启用的 Depth 流。 4. 使用 ob::Config 启动 ob::Pipeline 获取 Depth 流。 5. Depth 流的读取：分为同步阻塞和异步回调两种方式，选择其中一种即可。
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	1. 配置 Depth //枚举设备并使用第 1 个设备创建 Pipeline std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDevice(0); std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device)); //获取 3D 相机的所有 Depth Profile 列表 auto profiles = pipe->getStreamProfileList(OB_SENSOR_DEPTH); // 获取想要输出 Depth Profile std::shared_ptr<ob::VideoStreamProfile> depthProfile = nullptr; depthProfile = profiles->getVideoStreamProfile(640, 400, OB_FORMAT_Y16, 30); if(depthProfile != nullptr){ //创建 ob::Config 配置文件来配置 ob::Pipeline 要启用的流： std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>(); config->enableStream(depthProfile); } 2.1 同步方式获取 Depth 流 //启动 ob::Pipeline 获取 Depth 流。 pipe.start(config); bool readFrame = true; while(readFrame) { //通过同步阻塞模式等待最多 100 毫秒的时间获取帧集。 auto frameSet = pipe.waitForFrames(100); if(frameSet == nullptr) {

	<pre> continue; } //从帧集中获取 Depth 数据帧 auto depthFrame = frameSet->depthFrame(); //如果 Depth 帧不为空指针，进行业务处理。 if(depthFrame){ /*业务逻辑处理*/ //获取 Depth 帧的宽 uint32_t width = depthFrame->width(); //获取 Depth 帧的高 uint32_t height = depthFrame->height(); //获取 Depth 数据 uint16_t *data = (uint16_t *)depthFrame->data(); //获取帧数据流大小 uint32_t dataSize = depthFrame->dataSize(); //获取 Depth 帧硬件时间戳 uint64_t timeStamp = depthFrame->timeStamp(); //获取 Depth 帧系统时间戳 uint64_t sysTimeStamp = depthFrame->systemTimeStamp(); } }</pre> <h3>2.2 异步方式获取 Depth 流</h3> <pre>ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) { //获取 Depth 数据帧 auto depthFrame = frameSet->depthFrame(); //如果 Depth 帧不为空指针，进行业务处理。 if(depthFrame) { /*业务逻辑处理*/ } }; //设置 callback、ob::Config 启动 ob::Pipeline 获取 Depth 流。 holder->pipeline->start(config, callback);</pre> <h4>1. 停止 pipeline</h4> <pre>//停止 ob::Pipeline，停止 Depth 流读取 pipe.stop();</pre>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.7.3 获取 IR 流

接口描述	<p>1. 通过 ob::Pipeline 获取 3D 相机的所有 IR Profile 列表，包括流的分辨率、帧率和帧格式，返回 std::vector<std::shared_ptr<StreamProfile>>;</p> <p>2. 获取想要输出 IR Profile。;</p> <p>3. 创建 ob::Config 配置文件来配置 ob::Pipeline 要启用的 IR 流;</p> <p>4. 使用 ob::Config 启动 ob::Pipeline 获取 IR 流;</p> <p>5. IR 流的读取：分为同步阻塞和异步回调两种方式，选择其中一种即可。</p>
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	<h4>1. 配置 IR</h4> <pre>//枚举设备并使用第 1 个设备创建 Pipeline std::shared_ptr<DeviceList> devices = ctx.queryDeviceList(); auto device = devices->getDevice(0); std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device)); //获取 3D 相机的所有 IR Profile 列表 auto profiles = pipe->getStreamProfileList(OB_SENSOR_IR); //获取想要输出 IR Profile</pre>

```
std::shared_ptr<ob::VideoStreamProfile> irProfile = nullptr;
irProfile = profiles->getVideoStreamProfile(640, 400, OB_FORMAT_Y10, 30);

if(irProfile != nullptr){
    //创建 ob::Config 配置文件来配置 ob::Pipeline 要启用的流:
    std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();
    config->enableStream(irProfile);
}
```

2. 获取 IR 流

2.1 同步获取 IR 流

```
//启动 ob::Pipeline 获取 IR 流。
pipe.start(config);

bool readFrame = true;
while(readFrame) {
    //通过同步阻塞模式等待最多 100 毫秒的时间获取帧集。
    auto frameSet = pipe.waitForFrames(100);
    if(frameSet == nullptr) {
        continue;
    }

    //从帧集中获取 IR 数据帧
    auto irFrame = frameSet->irFrame();
    //如果 ir 帧不为空指针，进行业务处理。
    if(irFrame){
        /*业务逻辑处理*/
        //获取 IR 帧的宽
        uint32_t width = irFrame->width();
        //获取 IR 帧的高
        uint32_t height = irFrame->height();

        //获取 IR 帧数据
        uint16_t *data = (uint16_t *)irFrame->data();
        //获取帧数据流大小
        uint32_t dataSize = irFrame->dataSize();
        //获取 IR 帧硬件时间戳
        uint64_t timeStamp = irFrame->timeStamp();
        //获取 IR 帧系统时间戳
        uint64_t sysTimeStamp = irFrame->systemTimeStamp();
    }
}
```

2.2 异步获取 IR 流

```
ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) {

    //获取 ir 数据帧
    auto irFrame = frameSet->irFrame();
    //如果 ir 帧不为空指针，进行业务处理。
    if(irFrame) {
        /*业务逻辑处理*/
    }
};

//设置 callback、ob::Config 启动 ob::Pipeline 获取 IR 流。
holder->pipeline->start(config, callback);
```

3. 停止 Pipeline

```
//停止 ob::Pipeline，停止 IR 流读取
pipe.stop();
```

注：

1、双目相机，通过 `pipe->getStreamProfileList(OB_SENSOR_IR)`，默认开启的是左 IR，如果要获取右 IR 的图像，需要下发设置 IR 左右切换的命令。
参考参数配置章节。

2、Gemini 2 XL 支持同时输出左右 IR，请参考 DoubleInfraredViewer.cpp 例子。

2.7.4 获取 Color 流

接口描述	<div>1. 通过 ob::Pipeline 获取 3D 相机所有 Color Profile 列表，包括流的分辨率、帧率和帧格式，返回 std::vector<std::shared_ptr<StreamProfile>>;</div> <div>2. 获取想要输出的 Color Profile;</div> <div>3. 创建 ob::Config 配置文件来配置 ob::Pipeline 要启用 Color;</div> <div>4. 使用 ob::Config 启动 ob::Pipeline 获取 Color 流;</div> <div>5. Color 流的读取：分为同步阻塞和异步回调两种方式，选择其中一种即可。</div>
头文件	<div>#include "libobsensor/ObSensor.hpp"</div>
具体实现	<div><div>1. 配置 Color</div><div>//枚举设备并使用第 1 个设备创建 Pipeline</div><div>std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();</div><div>auto device = devices->getDevice(0);</div><div>std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device));</div><div> </div><div>//获取 3D 相机的所有 Color Profile 列表</div><div>auto profiles = pipe->getStreamProfileList(OB_SENSOR_COLOR);</div><div> </div><div>// 获取想要输出 Color 流配置，数据格式(OB_FORMAT_RGB)根据需求可选</div><div>std::shared_ptr<ob::VideoStreamProfile> colorProfile = nullptr;</div><div>colorProfile = profiles->getVideoStreamProfile(640, 480, OB_FORMAT_RGB, 30);</div><div> </div><div>if(colorProfile != nullptr){</div><div> //创建 ob::Config 配置文件来配置 ob::Pipeline 要启用的流;</div><div> std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();</div><div> config->enableStream(colorProfile);</div><div>}</div><div> </div><div>2. 获取 Color 流</div><div>2.1 同步获取 Color 流</div><div>//启动 ob::Pipeline 获取 Color 流。</div><div>pipe.start(config);</div><div> </div><div>bool readFrame = true;</div><div>while(readFrame) {</div><div> //通过同步阻塞模式等待最多 100 毫秒的时间获取帧集。</div><div> auto frameSet = pipe.waitForFrames(100);</div><div> if(frameSet == nullptr) {</div><div> continue;</div><div> }</div><div> </div><div> //从帧集中获取 Color 数据帧</div><div> auto colorFrame = frameSet->colorFrame();</div><div> //如果 Color 帧不为空指针，进行业务处理。</div><div> if(colorFrame){</div><div> /*业务逻辑处理*/</div><div> //获取 Color 帧的宽</div><div> uint32_t width = colorFrame->width();</div><div> //获取 Color 帧的高</div><div> uint32_t height = colorFrame->height();</div><div> </div><div> //获取 Color 数据</div><div> uint8_t *data = (uint8_t *)colorFrame->data();</div><div> //获取帧数据流大小</div><div> uint32_t dataSize = colorFrame->dataSize();</div><div> //获取 Color 帧硬件时间戳</div><div> uint64_t timeStamp = colorFrame->timeStamp();</div><div> //获取 Color 帧系统时间戳</div><div> uint64_t sysTimeStamp = colorFrame->systemTimeStamp();</div><div> }</div><div>}</div><div> </div><div>2.2 异步获取 Color 流</div></div>

	<pre>ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) { //获取 Color 数据帧 auto colorFrame = frameSet->colorFrame(); //如果 color 帧不为空指针，进行业务处理。 if(colorFrame) { /*业务逻辑处理*/ } }; //设置 callback、ob::Config 启动 ob::Pipeline 获取 IR 流。 holder->pipeline->start(config, callback); 3. 停止 Pipeline //停止 ob::Pipeline，停止 IR 流读取 pipe.stop();</pre>	
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--

2.7.5 获取 Imu 数据

IMU 包含加速度计和陀螺仪，加速度计和陀螺仪的采样率是关联的，如果同时开启，那么实际输出频率以最后设置的采样率为准。示例代码，请参考 [ImuReader Sample]。

加速度计的单位： m/s^2 。

陀螺仪单位： rad/s 。

注：

- 1. 加速度计和 陀螺仪 是独立输出的，如果需要同时获取到这 2 种数据，需要用户在应用程序中使用时间戳匹配。
- 2. Imu sample 是获取默认的加速度计和陀螺仪的配置，如果要设置量程和频率，参考代码如下：

2.7.5.1 陀螺仪(Gyro)

接口描述	<div>1. 获取陀螺仪 Sensor。</div> <div>2. 获取想要输出的陀螺仪的配置</div> <div>3. 开流。</div> <div>4. 通过回调函数获取陀螺仪数据。</div> <div>5. 停流</div>
头文件	#include "libobsensor/ObSensor.hpp"
具体实现	<div>1. 设置帧回调函数</div> <pre>ob::FrameCallback gyroFrameCallback = [](std::shared_ptr<ob::Frame> frame) { uint64_t timeStamp = frame->timeStamp(); auto gyroFrame = frame->as<ob::GyroFrame>(); OBGyroValue value = gyroFrame->value(); std::cout << "Gyro Frame: {tsp = " << timeStamp << "， temperature = " << gyroFrame->temperature() << "， data[" << value.x << "， " << value.y << "， " << value.z << "]rad/s" << std::endl; }</pre> <div>2. 开流</div> <pre>// 获取陀螺仪 Sensor，注意：找不到陀螺仪 Sensor 时会 throw exception。 auto gyroSensor = device->getSensor(OB_SENSOR_GYRO); if(gyroSensor) { // 查询支持的播放配置列表 auto profiles = gyroSensor->getStreamProfileList(); // 设置陀螺仪的量程和采样率，每个设备支持的量程和采样率请参考产品手则 auto profile = profiles->getGyroStreamProfile(OB_GYRO_FS_1000dps, OB_SAMPLE_RATE_200_HZ); gyroSensor->start(profile, gyroFrameCallback); } else { std::cout << "Get Gyro Sensor failed !" << std::endl; }</pre>

	<div><div>3. 停流</div><div>// 关闭数据流，释放资源</div><div>gyroSensor->stop();</div><div>gyroSensor = nullptr;</div></div>

2.7.5.2 加速度计 (Accel)

接口描述	<div>1. 获取加速度计 Sensor。</div> <div>2. 获取想要输出的加速度计的配置。</div> <div>3. 开流。</div> <div>4. 通过回调函数获取加速度计数据。</div> <div>5. 停流。</div>
头文件	<div>#include "libobsensor/ObSensor.hpp"</div>
具体实现	<div><div>1.设置帧回调函数</div><div>ob::FrameCallback accelFrameCallback = [](std::shared_ptr<ob::Frame> frame) {</div><div>auto timeStamp = frame->timeStamp();</div><div>auto accelFrame = frame->as<ob::AccelFrame>();</div><div>auto value = accelFrame->value();</div><div>std::cout << "Accel Frame: {tsp = " << timeStamp</div><div><< "， temperature = "<< accelFrame->temperature()</div><div><< ", data["<< value.x << ", " << value.y << ", " << value.z</div><div><< "]m/s^2"</div><div><< std::endl;</div><div>}</div></div> <div><div>2.开流</div><div>// 获取加速度计 Sensor，注意：找不到 Sensor 时会 throw exception。</div><div>auto accelSensor = device->getSensor(OB_SENSOR_ACCEL);</div><div>if(accelSensor) {</div><div>// 查询支持的播放配置列表</div><div>auto profiles = accelSensor->getStreamProfileList();</div><div>//设置加速度计的量程和采样率，每个设备支持的量程和采样率请参考产品手则</div><div>auto profile = profiles->getAccelStreamProfile(OB_ACCEL_FS_2g,</div><div>OB_SAMPLE_RATE_200_HZ);</div><div>accelSensor->start(profile, accelFrameCallback);</div><div>}</div><div>else {</div><div>std::cout << "Get accel Sensor failed !" << std::endl;</div><div>}</div></div> <div><div>3.停流</div><div>// 关闭数据流，释放资源</div><div>accelSensor->stop();</div><div>accelSensor = nullptr;</div></div>

3 参数配置

3.1 参数配置步骤

3.1.1 参数类型

int	整数型，后缀为 INT
float	浮点型，后缀为 FLOAT
bool	布尔型，后缀为 BOOL，范围只有 true 和 false

struct	结构类型，以 OB_STRUCT 为前缀，结构类型的参数比较复杂，SDK 通常把结构类型参数的操作封装为 Device 方法，降低调用复杂度；例如获取设备信息 ob_device_info。
--------	-------------------------------------------------------------------------------------------------

设备参数在【include/libobsensor/h/Property.h】中定义，类型为 enum，类型名称为【C++接口】OBPropertyID；每个设备参数对应一个 enum 枚举成员。

例如：

```
// Color AE 使能开关
OB_PROP_COLOR_AUTO_EXPOSURE_BOOL = 2000,
```

3.1.2 参数权限

调用设备参数时需要判断设备是否支持该指令操作，设备是否支持该参数或者支持使用该设备参数进行读写，有些设备参数是只读的，不允许写入，有些允许读写。

```
typedef enum {
    OB_PERMISSION_DENY      = 0,    /**< no permission */
    OB_PERMISSION_READ      = 1,    /**< can read */
    OB_PERMISSION_WRITE     = 2,    /**< can write */
    OB_PERMISSION_READ_WRITE = 3,    /**< can read and write */
    OB_PERMISSION_ANY       = 255,  /**< any situation above */
} OBPermissionType, ob_permission_type;
```

```
举例如下：
if (device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_READ_WRITE)) {
    std::cout<< "Support read and write OB_PROP_COLOR_EXPOSURE_INT"<<std::endl;
} else {
    std::cout<< "Not support read and write OB_PROP_COLOR_EXPOSURE_INT"<<std::endl;
}
```

3.1.3 最大值、最小值、步长、默认值

对于 int 类型、float 类型的部分参数存在最大值、最小值、步长的限制，参数设置时，先获取参数的范围和步长值，校验参数的合法性，避免因为不符合限制条件而导致调用失败。

概念	概念说明
参数范围	int 类型、float 类型的部分参数存在最大值、最小值、步长的限制条件，参数合法上下界包括最大值和最小值；bool 类型只有 true 和 false，没有最大值、最小值和步长。
最大值	设置的参数值的合理范围包括最大值，如果设置的参数值大于最大值，那么设置时会出现错误。
最小值	设置的参数值的合理范围包括最小值，如果设置的参数值小于最小值，那么设置时会出现错误。
步长	如果参数存在步长，那么在最大值最小值的闭合区间内，待设置的新参数值必须满足[(新参数值 - 最小值)%步长=0]，即设置的值必须满足步长倍数。
默认值	设备正常启动后参数的默认值

```
typedef struct {
    int32_t cur;    /**< Current value
    int32_t max;    /**< Maximum value
    int32_t min;    /**< Minimum value
    int32_t step;   /**< Step value
    int32_t def;    /**< Default value
} OBIntPropertyRange, ob_int_property_range;
```

说明：OBIntPropertyRange 定义在头文件：include/libobsensor/h/ObTypes.h

```
举例：
// 查询彩色曝光值范围和默认值、步长
OBIntPropeertyRange range = device->geIntPropertyRange( OB_PROP_COLOR_EXPOSURE_INT);
```

3.1.4 参数设置和获取

读取设备参数，权限要求：OB_PERMISSION_READ 或者 OB_PERMISSION_READ_WRITE。

参数获取举例如下：

```
//获取彩色相机的曝光值
uint32_t value = device->getIntProperty(OB_PROP_COLOR_EXPOSURE_INT);
```

设备参数设置，权限要求：OB_PERMISSION_WRITE 或者 OB_PERMISSION_READ_WRITE。
参数设置举例如下：

```
// 设置曝光时间 30000 微秒
device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, 30000);
```

3.1.5 综合示例

综合示例提供了设置彩色曝光时间的方法，在设置或获取参数之前需要注意点：

- 1、设置的参数是否支持。
- 2、参数是否有读写权限。
- 3、校验设置的参数的合法性，请参考【最大值、最小值、步长、默认值】小节。
- 4、设置彩色曝光时间。

```
//1. 检查参数是否支持写
if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
    //2. 关闭彩色 AE，此处可以忽略权限检查，支持彩色曝光值设置时一定支持 AE 开关
    device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
    //3. 设置彩色相机曝光时间
    int32_t exposure = 2000;
    //4. 获取参数的范围和步长
    OBIntPropertyRange range =device->getIntPropertyRange(OB_PROP_COLOR_EXPOSURE_INT);
    //5. 检验参数的合法性
    if(exposure >=range.min && exposure <= range.max && ((exposure -range.min)%range.step == 0)) {
        //6. 设置曝光时间
        device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, exposure);
    }
}
```

3.2 常用参数配置

3.1 章节介绍了参数配置的步骤，在本章节中主要介绍常用的参数配置，为了简单起见，不再介绍参数是否支持、参数的合法性检测，直接介绍常用参数的设置和获取。

3.2.1 获取序列号

获取设备 SN，首先通过 pipeline，获取 device，然后通过 device 获取 deviceInfo，从 deviceInfo 中获取 SN。同样的方法，可以从 DeviceInfo 里面获取设备名称、Pid、vid、固件版本等信息。

```
ob::Pipeline pipe;

auto device = pipe.getDevice();

auto deviceInfo = device->getDeviceInfo();

std::string serialNumber = deviceInfo->serialNumber();
```

3.2.2 获取相机内外参

3.2.2.1 通过 Device 获取相机参数

通过 Device 获取的内参，是从设备中获取的原始内参，这个内参是没做 D2C 的，非专业人员建议从 pipeline 获取参数。

应用场景：在只开 Depth 不要求 D2C 对齐时，可以从 Device 获取内参，Device 获取的内参是一个列表，要根据分辨率，选择和开流分辨率一致的那组内参。

```
ob::Pipeline pipe;
auto device = pipe.getDevice();
auto cameraParamList = device->getCalibrationCameraParamList();
//打印内参信息
```

```
for(int i = 0; i < cameraParamList->count(); i++) {
    OBCameraParam obParam = cameraParamList->getCameraParam(i);
    std::cout << "fx:" << obParam.depthIntrinsic.fx << " fy: " << obParam.depthIntrinsic.fy << " cx:" << obParam.depthIntrinsic.cx
        << " cy: " << obParam.depthIntrinsic.cy
        << " width: " << obParam.depthIntrinsic.width << " height:" << obParam.depthIntrinsic.height << std::endl;

}
```

3.2.2.2 通过 Pipeline 获取相机参数

通过 Pipeline 获取相机内参，可以单独开 depth 流，获取 depth 的内参，参考 DepthViewer 的 Sample 获取 Depth 流，也可以获取 D2C 后的内参，参考 SyncAlignViewer 的 sample 获取 Depth 流和 Color 流。

注意：通过 Pipeline 获取相机参数，一定要在开流后获取。

```
ob::Pipeline pipe;
//参考 DepthViewer 和 SyncAlignViewer 的 sample 开流
pipe.start(config);
auto camera_param = pipe.getCameraParam();
```

3.2.3 设置 Depth 参数

3.2.3.1 设置 Depth AE

```
//设置 Depth 自动曝光
if(device->isPropertySupported(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {

    bool isOpen = device->getBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL);
    if(device->isPropertySupported(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {
        device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, isOpen);
    }

    isOpen = device->getIntProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL);
    std::cout << "isOpen:" << isOpen << std::endl;
}
```

3.2.3.2 设置 Depth 曝光

设置 Depth 曝光，前置条件，需要关掉 Depth AE。对于结构光模组，设置 Depth 曝光等同于设置 IR 的曝光。

```
//关掉 AE
device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, false);

if(device->isPropertySupported(OB_PROP_DEPTH_EXPOSURE_INT, OB_PERMISSION_READ)) {
    //获取 Depth 曝光值
    int32_t exposure = device->getIntProperty(OB_PROP_DEPTH_EXPOSURE_INT);
    if(device->isPropertySupported(OB_PROP_DEPTH_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
        //设置 Depth 曝光值
        device->setIntProperty(OB_PROP_DEPTH_EXPOSURE_INT, exposure / 2);
    }
}
```

3.2.3.3 设置 Depth 增益

设置 Depth 增益，前置条件，需要关掉 Depth AE。对于结构光模组，设置 Depth 增益等同于设置 IR 的增益。

```
//关掉 AE
device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, false);

if(device->isPropertySupported(OB_PROP_DEPTH_GAIN_INT, OB_PERMISSION_READ)) {
    //获取 Depth 增益值
    int32_t gain = device->getIntProperty(OB_PROP_DEPTH_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_DEPTH_GAIN_INT, OB_PERMISSION_WRITE)) {
        //设置 Depth 增益值
        device->setIntProperty(OB_PROP_DEPTH_GAIN_INT, gain + 1000);
    }
}
```

```
    }  
}
```

3.2.3.4 设置 Depth 镜像

```
//判断接口是否支持  
if(device->isPropertySupported(OB_PROP_DEPTH_MIRROR_BOOL, OB_PERMISSION_READ)) {  
    //获取当前的镜像  
    bool isOpen = device->getBoolProperty(OB_PROP_DEPTH_MIRROR_BOOL);  
    if(device->isPropertySupported(OB_PROP_DEPTH_MIRROR_BOOL, OB_PERMISSION_WRITE)) {  
        device->setBoolProperty(OB_PROP_DEPTH_MIRROR_BOOL, !isOpen);  
    }  
}
```

3.2.3.5 设置 Depth 上下翻转

```
if(device->isPropertySupported(OB_PROP_DEPTH_FLIP_BOOL, OB_PERMISSION_WRITE)) {  
    //true: depth 上下翻转; false: 不翻转  
    device->setBoolProperty(OB_PROP_DEPTH_FLIP_BOOL,true);  
}
```

3.2.3.6 设置 Depth 旋转

Gemini2 系列的设备，支持 Depth 旋转，支持 0 度、90 度、180 度、270 度旋转，默认为 0 度。

```
//旋转参数  
typedef enum {  
    OB_ROTATE_DEGREE_0    = 0,    ///< Rotate 0  
    OB_ROTATE_DEGREE_90   = 90,    ///< Rotate 90  
    OB_ROTATE_DEGREE_180  = 180,   ///< Rotate 180  
    OB_ROTATE_DEGREE_270  = 270,   ///< Rotate 270  
} ob_rotate_degree_type,OBRotateDegreeType;  
  
if(device->isPropertySupported(OB_PROP_DEPTH_ROTATE_INT, OB_PERMISSION_WRITE)) {  
    //设置旋转的角度，参数 OBRotateDegreeType  
    device->setIntProperty(OB_PROP_DEPTH_ROTATE_INT, OB_ROTATE_DEGREE_90);  
}
```

3.2.3.7 视差转 Depth(D2D)

1. 硬件 D2D

硬件 D2D 指的是视差转 Depth 在模组内部实现，如 Gemini 2、Gemini 2 L、Astra 2、Gemini 2 XL 模组支持，调用前先判断是否支持该接口属性功能。

```
if(device->isPropertySupported(OB_PROP_DISPARITY_TO_DEPTH_BOOL, OB_PERMISSION_WRITE)) {  
    //参数：true 打开硬件 D2D，false 关闭硬件 D2D  
    device->setBoolProperty(OB_PROP_DISPARITY_TO_DEPTH_BOOL, true);  
}
```

2. 软件 D2D

软件 D2D 指的是视差转 Depth 在 SDK 内部实现。软件 D2D 除 TOF 模组(如 Femto Mega 和 Femto Bolt)外，其它模组都支持。
调用前先判断是否支持该接口属性功能。

```
if(device->isPropertySupported(OB_PROP_SDK_DISPARITY_TO_DEPTH_BOOL, OB_PERMISSION_WRITE)) {  
    //参数：true 打开软件 D2D，false 关闭软件 D2D  
    device->setBoolProperty(OB_PROP_SDK_DISPARITY_TO_DEPTH_BOOL, true);  
}
```


3.2.3.8 设置 Depth 的最小值和最大值

设置 Depth 的最小值和最大值，超出这个范围的 Depth 都赋值为 0，可以用来过滤噪点。

1. 设置 Depth 最小值

```
if(device->isPropertySupported(OB_PROP_MIN_DEPTH_INT, OB_PERMISSION_WRITE)) {
    //设置 Depth 的最小值(单位 mm)
    device->setIntProperty(OB_PROP_MIN_DEPTH_INT, 100);
}
```

2. 设置 Depth 最大值

```
if(device->isPropertySupported(OB_PROP_MAX_DEPTH_INT, OB_PERMISSION_WRITE)) {
    //设置 Depth 的最大值(单位 mm)
    device->setIntProperty(OB_PROP_MAX_DEPTH_INT, 10000);
}
```

3.2.3.9 设置 Depth 的单位(精度)

即深度数据帧的像素值单位，修改深度单位可以调整深度测量精度。例如对于一个距离为 1000mm 的物体，如果将单位设置成 0.2mm，输出的深度数据帧的像素数值将为 5000。（因为 5000 乘以 0.2mm 得到 1000mm）。以此类推，如果将单位设置为 0.1mm，像素数值将为 10000。

```
typedef enum {
    OB_PRECISION_1MM,    /**< 1mm */
    OB_PRECISION_0MM8,   /**< 0.8mm */
    OB_PRECISION_0MM4,   /**< 0.4mm */
    OB_PRECISION_0MM1,   /**< 0.1mm */
    OB_PRECISION_0MM2,   /**< 0.2mm */
    OB_PRECISION_0MM5,   /**< 0.5mm */
    OB_PRECISION_0MM05,  /**< 0.05mm */
    OB_PRECISION_UNKNOWN,
    OB_PRECISION_COUNT,
} OBDepthPrecisionLevel,
```

Gemini2、Gemini2 L、Astra2、Gemini2 XL 、Dabai DCL 支持深度精度

```
if(device->isPropertySupported(OB_PROP_DEPTH_PRECISION_LEVEL_INT, OB_PERMISSION_WRITE)) {
    /设置 Depth 单位 0.1mm
    device->setIntProperty(OB_PROP_DEPTH_PRECISION_LEVEL_INT, OB_PRECISION_0MM1);
}
```

注：

- 1. OrbbecSDK 1.7.5 以后默认输出的 Depth 单位为 1mm。
- 2. Depth 数据 带单位信息，可以从 Depth 帧的接口里面获取 Depth 单位，接口如下：

```
auto frameSet = pipe.waitForFrames(100);
if(frameSet == nullptr) {
    continue;
}

//获取 Depth 帧数据
auto depthFrame = frameSet->depthFrame();
//获取 Depth 单位
float depthUnit = depthFrame->getValueScale();
```

3.2.3.10 相机深度工作模式

深度模式是相机深度工作模式的简称，深度工作模式是指一个模组可以通过参数设置让深度相机切换到不同的应用场景，例如高精度场景、远距离场景、低功耗场景；每个深度应用场景相互独立，并且切换深度模式后不需要重启设备即可使用。

设备启动或者重启后的当前工作模式为默认深度模式。如果应用需要的深度模式不是默认深度模式，那么每次设备启动后都需要切换到目标的深度模式。首次连接设备时，都应该检查当前深度模式是否为目标深度模式。

Gemini2、Gemini2 L、Astra2 、Dabai DCL 、Gemini2 XL 相机 有不同的 Depth 模式。不同的 Depth 模式相机的功耗、支持的分辨率、帧率及 Depth 检测的有效距离不

一样，具体参考产品规格书。

如 Gemini 2 设备切换到【Binned Sparse Default】模式方法如下：

```
device->switchDepthWorkMode("Binned Sparse Default");
```

注：

- 1.切换深度工作模式请参考：(DepthWorkMode sample)

3.2.3.11 软件滤波

软件滤波用来过滤 Depth 的噪点。

注： 需要在软件滤波开关开启的条件下设置滤波参数，滤波参数在 SDK 内部根据分辨率率已经有默认值，一般用户不需要设置，只要开启软件滤波即可。

1. 软件滤波开关

```
if(device->isPropertySupported(OB_PROP_DEPTH_SOFT_FILTER_BOOL, OB_PERMISSION_WRITE)) {  
  
    //参数：true 开启软件滤波，false 关闭软件滤波  
  
    device->setBoolProperty(OB_PROP_DEPTH_SOFT_FILTER_BOOL, true);  
  
}
```

2. 软件滤波参数设置

设置软件滤波 maxdiff 参数，默认不需要调用，除非特殊的需求调节，需要相关技术人员给出设定参数。

```
if(device->isPropertySupported(OB_PROP_DEPTH_MAX_DIFF_INT, OB_PERMISSION_WRITE)) {  
  
    device->setIntProperty(OB_PROP_DEPTH_MAX_DIFF_INT, 16);  
  
}
```

设置软件滤波 maxSpeckleSize 参数，默认不需要调用，除非特殊的需求调节，需要相关技术人员给出设定参数。

```
if(device->isPropertySupported(OB_PROP_DEPTH_MAX_SPECKLE_SIZE_INT, OB_PERMISSION_WRITE)) {  
  
    device->setIntProperty(OB_PROP_DEPTH_MAX_SPECKLE_SIZE_INT, 480);  
  
}
```

3.2.4 设置 IR 参数

3.2.4.1 设置 IR AE

```
if(device->isPropertySupported(OB_PROP_IR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {  
    bool isOpen = device->getBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL);  
    if(device->isPropertySupported(OB_PROP_IR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {  
        //true: 开启 AE, false: 关掉 AE  
        device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, !isOpen);  
    }  
}
```

3.2.4.2 设置 IR 曝光

设置 IR 曝光，前置条件，需要关掉 IR AE。

```
//首先关掉 AE  
device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, false);  
//获取和设置曝光值  
if(device->isPropertySupported(OB_PROP_IR_EXPOSURE_INT, OB_PERMISSION_READ)) {  
    //获取 IR 曝光值
```

```
int32_t exposure = device->getIntProperty(OB_PROP_IR_EXPOSURE_INT);
if(device->isPropertySupported(OB_PROP_IR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
    //设置 IR 曝光值
    device->setIntProperty(OB_PROP_IR_EXPOSURE_INT, exposure / 2);
}
}
```

3.2.4.3 设置 IR 增益

设置 IR 增益，前置条件，需要关掉 IR AE。

//首先关掉 AE

```
device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_IR_GAIN_INT, OB_PERMISSION_READ)) {
    //获取 IR 增益值
    int32_t gain = device->getIntProperty(OB_PROP_IR_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_IR_GAIN_INT, OB_PERMISSION_WRITE)) {
        //设置 IR 增益值
        device->setIntProperty(OB_PROP_IR_GAIN_INT, gain+1000);
    }
}
```

3.2.4.4 设置 IR 镜像

```
if(device->isPropertySupported(OB_PROP_IR_MIRROR_BOOL, OB_PERMISSION_READ)) {
    //获取 IR 镜像
    bool isMirror = device->getIntProperty(OB_PROP_IR_MIRROR_BOOL);
    if(device->isPropertySupported(OB_PROP_IR_MIRROR_BOOL, OB_PERMISSION_WRITE)) {
        //IR 镜像设置
        device->setIntProperty(OB_PROP_IR_MIRROR_BOOL, !isMirror);
    }
}
```

3.2.4.5 设置 IR 上下翻转

```
if(device->isPropertySupported(OB_PROP_IR_FLIP_BOOL, OB_PERMISSION_WRITE)) {
    //true:上下翻转; false: 不翻转
    device->setBoolProperty(OB_PROP_IR_FLIP_BOOL,true);
}
```

3.2.4.6 设置 IR 旋转

Gemini 2 系列的设备，支持 Depth 旋转，支持 0 度、90 度、180 度、270 度旋转，默认 0 度不旋转。

//旋转参数

```
typedef enum {
    OB_ROTATE_DEGREE_0    = 0,    ///< Rotate 0
    OB_ROTATE_DEGREE_90   = 90,    ///< Rotate 90
    OB_ROTATE_DEGREE_180  = 180,   ///< Rotate 180
    OB_ROTATE_DEGREE_270  = 270,   ///< Rotate 270
} ob_rotate_degree_type,OBRotateDegreeType;
```

```
if(device->isPropertySupported(OB_PROP_IR_ROTATE_INT, OB_PERMISSION_WRITE)) {
    //设置旋转的角度参数： OBRotateDegreeType
    device->setIntProperty(OB_PROP_IR_ROTATE_INT, OB_ROTATE_DEGREE_90);
}
```

3.2.4.7 设置 IR 左右切换

//判断是否支持切换左右 IR 通道

```
if(pipe.getDevice()->isPropertySupported(OB_PROP_IR_CHANNEL_DATA_SOURCE_INT, OB_PERMISSION_READ_WRITE)) {
    // Gemini 2 和 Gemini 2 L 产品支持 SENSOR_IR 选择 sensor 输出，0 是左 IR，1 是右 IR。
    int32_t dataSource = 0;
    pipe.getDevice()->setIntProperty(OB_PROP_IR_CHANNEL_DATA_SOURCE_INT, dataSource);
}
```

3.2.5 设置 Color 参数

彩色相机可以设置的参数如下：自动 AE、曝光、增益、自动白平衡、色温、亮度、锐度、Gamma、饱和度、色调、电力线频率等参数。 大多数应用场景使用默认彩色相机参数，就能满足需求。

3.2.5.1 设置 Color AE

```
//设置 AE
if(device->isPropertySupported(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {
    if(device->isPropertySupported(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {
        //true: 表示开启 AE,false: 关闭 AE
        device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, true);
    }
}
```

3.2.5.2 设置 Color 曝光

设置 Color 曝光，前置条件，需要关掉 Color AE。

```
//关闭 AE
device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_READ)) {
    //获取 Color 相机曝光值
    int32_t exposure = device->getIntProperty(OB_PROP_COLOR_EXPOSURE_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
        //设置 Color 相机曝光值
        device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, exposure / 2);
    }
}
```

3.2.5.3 设置 Color 增益

设置 Color 增益，前置条件，需要关掉 Color AE。

```
//关闭 AE
device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_GAIN_INT, OB_PERMISSION_READ)) {
    //获取 Color 相机增益值
    int32_t gain = device->getIntProperty(OB_PROP_COLOR_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_GAIN_INT, OB_PERMISSION_WRITE)) {
        //设置 Color 相机增益值
        device->setIntProperty(OB_PROP_COLOR_GAIN_INT, gain + 10);
    }
}
```

3.2.5.4 设置 Color 镜像

```
//设置 Color 镜像
if(device->isPropertySupported(OB_PROP_COLOR_MIRROR_BOOL, OB_PERMISSION_READ)) {
    if(device->isPropertySupported(OB_PROP_COLOR_MIRROR_BOOL, OB_PERMISSION_WRITE)) {
        //参数: true 镜像, false 非镜像
        device->setBoolProperty(OB_PROP_COLOR_MIRROR_BOOL, true);
    }
}
```

3.2.5.5 设置 Color 上下翻转

```
if(device->isPropertySupported(OB_PROP_COLOR_FLIP_BOOL, OB_PERMISSION_WRITE)) {
    //true: 表示上下翻转; false: 表示不翻转
    device->setBoolProperty(OB_PROP_COLOR_FLIP_BOOL, true);
}
```

```
}
```

3.2.5.6 设置 Color 旋转

```
//旋转参数
typedef enum {
    OB_ROTATE_DEGREE_0    = 0,    ///< Rotate 0
    OB_ROTATE_DEGREE_90   = 90,    ///< Rotate 90
    OB_ROTATE_DEGREE_180  = 180,   ///< Rotate 180
    OB_ROTATE_DEGREE_270  = 270,   ///< Rotate 270
} ob_rotate_degree_type,OBRotateDegreeType;

if(device->isPropertySupported(OB_PROP_COLOR_ROTATE_INT, OB_PERMISSION_WRITE)) {
    //设置旋转的角度
    device->setIntProperty(OB_PROP_COLOR_ROTATE_INT, OB_ROTATE_DEGREE_90);
}
```

3.2.5.7 设置 Color 自动白平衡(AWB)

```
if(device->isPropertySupported(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, OB_PERMISSION_READ)) {

    bool isOpen = device->getBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL);
    if(device->isPropertySupported(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, OB_PERMISSION_WRITE)) {
        // true: 表示开 AWB; false: 关闭 AWB
        device->setBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, isOpen);
    }
}
```

3.2.5.8 设置 Color 白平衡参数

设置白平衡参数，需要先关闭自动白平衡。

```
//关掉白平衡
device->setBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_WHITE_BALANCE_INT, OB_PERMISSION_READ)) {
    int32_t colorWhiteBalance = device->getIntProperty(OB_PROP_COLOR_WHITE_BALANCE_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_WHITE_BALANCE_INT, OB_PERMISSION_WRITE)) {
        //设置 Color 白平衡参数,参数需要满足范围和步长限制，否则会抛异常
        device->setIntProperty(OB_PROP_DEPTH_EXPOSURE_INT, colorWhiteBalance +1 );
    }
}
```

3.2.5.9 设置 Color 亮度

```
if(device->isPropertySupported(OB_PROP_COLOR_BRIGHTNESS_INT, OB_PERMISSION_WRITE)) {
    OBIntPropertyRange range = device->getIntPropertyRange(OB_PROP_COLOR_BRIGHTNESS_INT);
    int32_t brightness = 50
    //校验参数
    if(brightness >= range.min && brightness <=range.max && (brightness - range.min) % range.step == 0) {
        device->setIntProperty(OB_PROP_COLOR_BRIGHTNESS_INT, brightness);
    }
}
```

锐度 (OB_PROP_COLOR_SHARPNESS_INT)、
Gamma (OB_PROP_COLOR_GAMMA_INT)、
饱和度 (OB_PROP_COLOR_SATURATION_INT)、
色调 (OB_PROP_COLOR_HUE_INT)、
自动曝光优先 (OB_PROP_COLOR_AUTO_EXPOSURE_PRIORITY_INT)、
背光补偿 (OB_PROP_COLOR_BACKLIGHT_COMPENSATION_INT)、
对比度 (OB_PROP_COLOR_CONTRAST_INT) 的设置方法和亮度一样，不具体介绍。

3.2.5.10 设置 Color 电力线频率

需要根据不同国家和地区的电力线频率自行设置 50Hz 或 60Hz，设置电力线频率是防止彩色图像闪烁。

```
typedef enum {
    OB_POWER_LINE_FREQ_MODE_CLOSE = 0,    ///< Close
    OB_POWER_LINE_FREQ_MODE_50HZ  = 1,    ///< 50Hz
    OB_POWER_LINE_FREQ_MODE_60HZ  = 2,    ///< 60Hz
} ob_power_line_freq_mode,
    OBPowLineFreqMode;

if(device->isPropertySupported(OB_PROP_COLOR_POWER_LINE_FREQUENCY_INT, OB_PERMISSION_WRITE)) {
    int powerFreq = OB_POWER_LINE_FREQ_MODE_50HZ;
    device->setIntProperty(OB_PROP_COLOR_BRIGHTNESS_INT, powerFreq);
}
```

3.2.6 设备重启

```
device->reboot();
```

3.2.7 激光开关

激光开关用来打开或关闭激光。

```
if(device->isPropertySupported(OB_PROP_LASER_BOOL, OB_PERMISSION_WRITE)) {
    //true: 表示打开激光; false:表示关闭激光
    device->setBoolProperty(OB_PROP_LASER_BOOL, true);
}
```

3.2.8 LDP 开关

LDP 测距模块可以对靠近物体进行检测。当 LDP 打开后，在相机规定距离范围内检测障碍物后，系统会降低激光能级，直到关闭激光，以实现激光保护；当 LDP 关闭后此功能失效。

```
if(device->isPropertySupported(OB_PROP_LDP_BOOL, OB_PERMISSION_WRITE)) {
    //true:表示打开 LDP; false: 表示关闭 LDP
    device->setBoolProperty(OB_PROP_LDP_BOOL, true);
}
```

3.2.9 获取 LDP 近距离保护测量值

LDP 开关处于使能状态时，获取 LDP 近距离保护测量值。

```
注： 这个接口需要在激光打开的条件下，开 Depth 或 IR 流 后调用，才有效。
if(device->isPropertySupported(OB_PROP_LDP_MEASURE_DISTANCE_INT, OB_PERMISSION_READ)) {
    //返回值表示 LDP 测量值，单位 mm。
    int32_t value = device->getIntProperty(OB_PROP_LDP_MEASURE_DISTANCE_INT);
}
```

3.2.10 获取 LDP 保护状态

LDP 开关处于使能状态时，获取 LDP 保护状态，如果 LDP 处于保护状态，激光能级会降低，当激光能级降到 0 时，激光会关掉。

注： 这个接口需要在激光打开的条件下，开 Depth 或 IR 流 后调用，才有效。

```
if(device->isPropertySupported(OB_PROP_LDP_STATUS_BOOL, OB_PERMISSION_READ)) {
    //true: 表示 LDP 处于保护状态; false: 表示 LDP 处于正常未保护状态
    bool status = device->getBoolProperty(OB_PROP_LDP_STATUS_BOOL);
}
```

3.2.11 泛光灯开关

注：需要模组支持泛光灯

```
if(device->isPropertySupported(OB_PROP_FLOOD_BOOL, OB_PERMISSION_WRITE)) {
    // true: 表示打开 ir flood; false:表示关闭 ir flood
    bool status = device->setBoolProperty(OB_PROP_FLOOD_BOOL, true);
}
```

3.2.12 设备授时

```
//多设备授时，参数 repeatInterval 表示授时周期，单位毫秒（ms），例如每半小时授时一次：repeatInterval = 30*1000*1000（ms）
context->enableDeviceClockSync(uint64_t repeatInterval);

//单设备授时，需要周期性的调用，例如 60 分钟调用一次。
device->timerSyncWithHost();
```

3.2.12 网络设备 IP 地址修改设置

```
//Device 修改设备网络 IP 配置，OB_STRUCT_DEVICE_IP_ADDR_CONFIG
OBNetIpConfig config;
device->setStructuredData(OB_STRUCT_DEVICE_IP_ADDR_CONFIG, &config, sizeof(OBNetIpConfig));
```

```
/**
 * @brief 网络设备 IP 地址配置（IPv4）
 */
typedef struct {
    /**
     * @brief DHCP status
     *
     * @note 0: static IP; 1: DHCP
     */
    uint16_t dhcp;

    /**
     * @brief IP 地址（IPv4，大端序：192.168.1.10，address[0] = 192, address[1] = 168, address[2] = 1, address[3] = 10）
     */
    uint8_t address[4];

    /**
     * @brief 子网掩码（大端序）
     */
    uint8_t mask[4];

    /**
     * @brief 网关（大端序）
     */
    uint8_t gateway[4];
} OBNetIpConfig, ob_net_ip_config, DEVICE_IP_ADDR_CONFIG;
```

4 D2C

Depth to Color（D2C），这是一个对深度图像执行逐像素几何变换的功能。其执行结果是我们通过 D2C 变换，将深度图像与其对应的目标彩色图像进行空间对齐，使得我们可以通过彩色图像中任意一个像素的图像坐标位置，在 D2C 后的深度图像中找到相同的图像坐标位置，即可定位到此彩色像素所对应的深度信息。我们在 D2C 后生成一个与目标彩色图像相同大小的深度图像，图像内容是在彩色相机坐标系统下的深度数据。换句话说，D2C 功能重建了一个使用彩色相机的原点和尺寸“拍摄”的深度图像，其中每个像素都与彩色相机的对应像素坐标匹配。

D2C 有硬件和软件两种实现方式，其中硬件 D2C 在模组内部实现，不额外消耗上位机计算资源。软件 D2C 以 SDK 软件库的方式在上位机端实现，需要额外消耗上位机计算资源。在上位机预留算力较少的情况下，推荐使用硬件 D2C；如果上位机预留算力充足，且对 D2C 效果要求较高，则建议使用软件 D2C。

使用 D2C 功能需要开启 Color 和 Depth 数据流，具体可以参考 Orbbec SDK 中 Examples 路径下的【SyncAlignViewer】示例。


```
typedef enum {
    ALIGN_DISABLE,      // 关闭 D2C
    ALIGN_D2C_HW_MODE, // 硬件 D2C
    ALIGN_D2C_SW_MODE, // 软件 D2C
} OBAlignMode,    ob_align_mode;

// 举例：设置硬件 D2C
config->setAlignMode(ALIGN_D2C_HW_MODE);

pipe.start(config);
```

注： 如何判断选择的 Depth 分辨率和 Color 分辨率支持硬件 D2C 对齐？

代码如下：

```
ob::Pipeline pipe;
auto          profiles = pipe.getStreamProfileList(OB_SENSOR_COLOR);

//使用 1920x1080 分辨率举例
auto colorProfile = profiles->getVideoStreamProfile(1920, 1080, OB_FORMAT_MJPEG, 30);
//获取 colorProfile 对应支持的 depthProfile 列表, 硬件 D2C: ALIGN_D2C_HW_MODE
auto depthProfileList = pipe.getD2CDepthProfileList(colorProfile, ALIGN_D2C_HW_MODE);
auto count            = depthProfileList->count();
for(size_t i = 0; i < count; i++) {
    auto depthProfile = depthProfileList->getProfile(i)->as<ob::VideoStreamProfile>();

    int width  = depthProfile->width();
    int height = depthProfile->height();
    int format = depthProfile->format();
    int fps    = depthProfile->fps();

    //打印支持 D2C 的 Depth 数据流信息，包括图像的宽和高
    std::cout << "width: " << width << " height: " << height << " format: " << format << " fps:" << fps << std::endl;
}
```

同样，如何判断选择的 Depth 分辨率和 Color 分辨率支持软件 D2C 对齐？ 只需要将以上代码中的 ALIGN_D2C_HW_MODE 改为 ALIGN_D2C_SW_MODE 即可。

5 点云

点云的 C++ Sample 参考：PointCloud。其中包含两种点云的获取方式：仅包含三维信息的 Depth 点云，以及包含三维信息和 RGB 纹理信息的 RGBD 点云。

5.1 Depth 点云

- 1、开启 Depth 流：参考获取 Depth 流的 Sample，开启对应分辨率的 Depth 流。
- 2、创建点云对象

```
ob::PointCloudFilter pointCloud;
```

- 3、获取相机内参

```
auto cameraParam = pipeline.getCameraParam();
pointCloud.setCameraParam(cameraParam);
```

- 4、获取点云数据

```
//通过同步阻塞模式等待帧，最多等待 100ms
auto frameset = pipeline.waitForFrames(100);

//获取 Depth 单位
auto depthValueScale = frameset->depthFrame()->getValueScale();

//将 Depth 单位设置到点云 Filter
pointCloud.setPositionDataScaled(depthValueScale);

//设置点云的类型，设置成输出 Depth 点云
```



```
pointCloud.setCreatePointFormat(OB_FORMAT_POINT);
//调用点云 Filter 生成点云数据
std::shared_ptr<ob::Frame> frame = pointCloud.process(frameset);
//将点云数据保存成.ply 文件
savePointsToPly(frame, "DepthPoints.ply");
```

5.2 RGBD 点云

- 1、开启 Depth 和 Color 数据流，并开启 D2C 对齐（硬件或软件均可），具体请参考 D2C 章节。
- 2、创建点云对象

```
ob::PointCloudFilter pointCloud;
```

- 3、获取相机内参

```
auto cameraParam = pipeline.getCameraParam();
pointCloud.setCameraParam(cameraParam);
```

- 4、获取点云数据

```
//通过同步阻塞模式等待帧，最多等待 100ms
auto frameset = pipeline.waitForFrames(100);

//获取 Depth 单位
auto depthValueScale = frameset->depthFrame()->getValueScale();

//将 Depth 单位设置到点云 Filter
pointCloud.setPositionDataScaled(depthValueScale);

//设置点云的类型，设置成输出 RGBD 点云
pointCloud.setCreatePointFormat(OB_FORMAT_RGB_POINT);
//调用点云 Filter 生成点云数据
std::shared_ptr<ob::Frame> frame = pointCloud.process(frameset);
//将点云数据保存成.ply 文件
saveRGBPointsToPly(frame, "RGBPoints.ply");
```

6 录像和回放

Orbbec SDK 录像的文件格式是 `rosbag`，但是 Depth 录像在 SDK 内部会做了无损压缩，所以 Orbbec SDK 录制的文件，只能调用 Orbbec SDK 提供的回放接口，进行回放，不支持只用标准 ROS 回放。

6.1 录像

支持录制视频帧数据，如果录制时开启了 D2C，那么录制的文件也是 D2C 对齐的数据。录像的 sample，请参考：Record Sample，下面以 Femto Mega 录制 D2C 后的 Depth 和 Color 为例，介绍录像功能。其它产品的录像功能和 Femto Mega 一样，请参考产品手册修改分辨率。

注：如果要录制 D2C 的后的数据，需要确保开启的 Depth 和 Color 流的分辨率 能支持对齐。请参考 D2C 章节。

```
ob::Pipeline pipe;

//1. Depth 配置
std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();
auto depthProfiles = pipe.getStreamProfileList(OB_SENSOR_DEPTH);
auto depthProfile = depthProfiles->getVideoStreamProfile(640, 576, OB_FORMAT_Y16, 30);
//2. Color 配置
auto colorProfiles = pipe.getStreamProfileList(OB_SENSOR_COLOR);
auto colorProfile = colorProfiles->getVideoStreamProfile(1920, 1080, OB_FORMAT_RGB, 30);
//3. 使能 Depth 流
config->enableStream(depthProfile);
//4. 使能 Color 流
config->enableStream(colorProfile);

//5. 开启 D2C
```

```
config->setAlignMode(ALIGN_D2C_HW_MODE);
//6. 开流
pipe.start(config);

//7. 开启录像
pipe.startRecord("./OrbbecPipeline.bag");

while(1)
{
//等待录像完成，参考 Record Sample
}

//8. 停止录像
pipe.stopRecord();
//9. 停流
pipe.stop();
```

6.2 录像回放

录像回放请参考：Playback Sample，该 sample 也支持回放 OrbbecViewer 工具录制的录像文件，回放 OrbbecViewer 录制的文件时，注意将录像文件名改为 Playback Sample 的文件名。

7 日志管理

日志管理的每个接口都带有日志等级的参数，日志以最后调用的接口设置的日志等级为准。

设置全局日志等级：setLoggerSeverity 函数，能同时控制输出到控制台、输出到文件、输出到日志回调函数的日志等级。

以下三个输出日志的方式，是独立的，相互之间设置不影响。

- 1、 设置日志输出到控制台 (setLoggerToConsole 函数)。
- 2、 设置输出到文件(setLoggerToConsole 函数)
- 3、 设置日志从回调函数输出(setLoggerToCallback 函数)

7.1 设置日志等级

```
typedef enum {
    OB_LOG_SEVERITY_DEBUG, /**< debug */
    OB_LOG_SEVERITY_INFO,  /**< information */
    OB_LOG_SEVERITY_WARN,  /**< warning */
    OB_LOG_SEVERITY_ERROR, /**< error */
    OB_LOG_SEVERITY_FATAL, /**< fatal error */
    OB_LOG_SEVERITY_OFF    /**< off (close LOG) */
} OBlogSeverity,

ob::Context context;
//设置 Log 输出的错误等级
context.setLoggerSeverity(OB_LOG_SEVERITY_DEBUG);
```

这个是一个全局的接口，设置这个接口，可以影响到 输出到控制台、输出到文件、输出到日志回调函数的日志等级。如果初始时调用 setLoggerSeverity 函数设置了日志等级，后面又调用了其他的接口设置日志等级，以后面设置的日志等级为准。

7.2 设置日志输出到控制台

```
ob::Context context;
//设置 Log 输出到终端控制台
context.setLoggerToConsole(OB_LOG_SEVERITY_DEBUG);
```

注： 只会影响输出到控制台的日志等级

7.3 设置日志输出到文件

```
ob::Context context;
//设置 Log 输出到文件
context.setLoggerToFile(OB_LOG_SEVERITY_DEBUG);
```

注： 只会影响输出到文件的日志等级

7.4 设置日志回调输出

```
#include "libobsensor/hpp/Context.hpp"

//日志输出回调函数
ob::Context::LogCallback logCallback = [] (OBLogSeverity severity, const char *logMsg)
{
    std::cout << "LogSeverity: " << static_cast<int> (severity) << std::endl;
    std::cout << "log: " << logMsg << std::endl;
};

int main(int argc, char **argv)
{
    ob::Context context;
    //设置 Log 以回调函数的形式输出，方便用户在自己的应用中管理日志
    context.setLoggerToCallback(OB_LOG_SEVERITY_DEBUG, logCallback);
    .....
}
```

注： 只会影响输出到回调函数的日志等级

8 配置文件介绍

以下内容介绍 Orbbec SDK 配置文件 (OrbbecSDKConfig_v1.0.xml) 。

8.1 日志段

```
<Log>

<!--日志输出等级，可选值: 0-DEBUG, 1-INFO, 2-WARN, 3-ERROR, 4-FATAL, 5-OFF-->

<!--全局日志等级-->

<LogLevel>0</LogLevel>

<!--输出到文件的日志等级 -->

<FileLogLevel>0</FileLogLevel>

<!--输出到控制台的日志等级 -->

<ConsoleLogLevel>1</ConsoleLogLevel>

<!--默认日志输出文件的路径： 如果没配置， Windows/Linux:  "./Log"; Android: "/sdcard/Orbbec/Log"-->
```

```
<!--<OutputDir>./log</OutputDir-->

<!--默认输出文件的大小，单位: MB-->

<MaxFileSize>100</MaxFileSize>

<!--默认输出文件的个数，循环覆盖-->

<MaxFileNum>3</MaxFileNum>

<!--日志异步输出: true-enable, false-disable (default)-->

<Async>>false</Async>

</Log>
```

8.2 内存管理

```
<Memory>

<!--使能内存池，避免造成内存碎片问题. true-enable, false-disable-->

<EnableMemoryPool> true </EnableMemoryPool>

<!--设置数据帧最大的内存 Size, 单位: MB, 最小内存 100MB-->

<MaxFrameBufferSize> 2048 </MaxFrameBufferSize>

<!--Pipeline 中帧缓冲队列的 Size，如设置成 10 表示最大缓冲 10 帧 -->

<PipelineFrameQueueSize>10</PipelineFrameQueueSize>

<!--内部帧处理单元的帧缓冲队列的 Size，如设置成 10 表示最大缓冲 10 帧 ->

<FrameProcessingBlockQueueSize>10</FrameProcessingBlockQueueSize>

</Memory>
```

8.3 Pipeline 参数配置

```
<Pipeline>

  <Stream>

    <Depth>

      <!--true: 使用 device 字段默认的流配置开流，false: 使用特定流配置开流 -->

      <UseDefaultStreamProfile>true</UseDefaultStreamProfile>

    </Depth>

    <Color>

      <!--true:使用 device 字段默认的流配置开流，false: 使用特定流配置开流 -->

      <UseDefaultStreamProfile>true</UseDefaultStreamProfile>

    </Color>

    <!--如果要开启其他流，请参考以上配置进行配置，如：开启 Ir 流 -->

    <!--IR -->

    <!--true:使用 device 字段默认的流配置开流，false: 使用特定流配置开流 -->

    <UseDefaultStreamProfile>true</UseDefaultStreamProfile>

  <!-- /IR -->

  </Stream>

  <!--D2C 对齐(0: 不对齐;1:硬件对齐;2:软件对齐) 注意：需要硬件模组支持-->

  <AlignMode>0</AlignMode>

  <!-- Pipeline 开启多路流时，是否开启帧同步，注意：需要硬件模组支持帧同步->

  <FrameSync>>false</FrameSync>

</Pipeline>
```

8.4 device 参数配置

以 Astra+为例介绍 Device 配置，其它模组的配置，请参考对应的模组配置段。

```
<Device>

  <EnumerateNetDevice>false</EnumerateNetDevice>

  <!--The device corresponding to Astra adv is astra+-->

  <AstraAdv>

    <Depth>

      <!--默认的图像宽-->

      <Width>640</Width>

      <!--默认的图像高-->

      <Height>480</Height>

      <!--默认的帧率-->

      <FPS>30</FPS>

      <!--默认的深度格式，模组支持的格式可以参考 OrbbecViewer-->

      <Format>Y16</Format>

      <!--开流失败，是否重试, 0:不重试, >=1 重试的次数-->

    <StreamFailedRetry>0</StreamFailedRetry>

  </Depth>

  <Color>

    <!--默认的图像宽-->

    <Width>640</Width>

    <!--默认的图像高-->

    <Height>480</Height>

    <!--默认的帧率-->

    <FPS>30</FPS>

    <!--默认的帧格式，模组支持的格式可以参考 OrbbecViewer -->

    <Format>MJPG</Format>

    <!--开流失败，是否重试, 0:不重试, >=1 重试的次数-->

  <StreamFailedRetry>0</StreamFailedRetry>

</Color>

  <IR>

    <!--默认的图像宽-->

    <Width>640</Width>

    <!--默认的图像高-->

    <Height>480</Height>

    <!--默认的帧率-->

    <FPS>30</FPS>

    <!--默认的帧格式，模组支持的格式可以参考 OrbbecViewer -->

    <Format>Y16</Format>

    <!--开流失败，是否重试, 0:不重试, >=1 重试的次数-->

  <StreamFailedRetry>0</StreamFailedRetry>

</IR>

</AstraAdv>
```

9 多机同步

多机同步请参考多机同步文档。
[多机同步使用操作说明文档 - 3D 视觉 AI 开放平台 - 站点标题 \(orbbec.com.cn\)](#)

10 异常处理

调用 OrbbecSDK 接口发生错误时会通过 exception 通知调用者，开发者务必处理这些异常信息。所有接口调用都要 try-catch 并处理 Error。
//C++ 异常处理类：(源码：include/libobsensor/hpp/Error.hpp)

```
namespace ob {  
  
class OB_EXTENSION_API Error {  
  
public:  
  
    const char *getMessage() const noexcept;  
  
    OBExceptionType getExceptionType() const noexcept;  
  
    const char *getName() const noexcept;  
  
    const char *getArgs() const noexcept;  
  
};  
}
```

示例：构建 Context

```
int main(void) {  
    try {  
        // Create a Context.  
        ob::Context ctx;  
  
        catch(ob::Error &e) {  
            std::cerr << "function:" << e.getName() << "\nargs:" << e.getArgs() << "\nmessage:" << e.getMessage() << "\ntype:" << e.getExceptionType() << std::endl;  
            exit(-1);  
        }  
  
        return 0;  
    }  
}
```

11 结束

《Orbbec SDK C++接口使用规范》会根据 Orbbec SDK 的变更不断的修改和优化。