快速开始

第一步：集成

平台支持：

平台	SDK 及兼容性	SDK 及 Demo
iOS	Xcode 9.3+, iOS 9.0+	填写 申请表 进行申请，完成申请后，相关工作人员将联系您进行需求沟通，并提供对应 SDK 及 Demo

将IoTVideoSDK集成到您的项目中并配置工程依赖，就可以完成SDK的集成工作。

> 详情请参见 【集成指南】

第二步：接入准备

开始使用 SDK 前，我们还需要获取`accessId`和`accessToken`，获取方式如下：

- **accessId**： 指外部访问 IoT Video 云平台的唯一性身份标识
- **accessToken**： 登录成功后 IoT Video 云服务器返回的`accessToken`。

1. 获取 accessId

用户自有账号体系可以采用云对接的方式实现账户体系相关业务，详情请参见 终端用户注册。

2. 获取accessToken

用户自有账号体系可以采用云对接的方式实现账户体系相关业务，详情请参见 终端用户接入授权。

第三步：SDK初始化

1、初始化

在`AppDelegate`中的`application:didFinishLaunchingWithOptions:`调用如下初始化方法：

```
import IoTVideo

IoTVideo.sharedInstance.setup(launchOptions: launchOptions)
```

2、注册

账号注册成功后可获取到`accessId`，登录成功后可获取到`accessToken`，调用sdk注册接口：

```
import IoTVideo

IoTVideo.sharedInstance.register(withAccessId: "********", accessToken:
"********")
```

⚠️注意：对设备的操作都依赖于`accessId`和`accessToken`的加密校验，非法参数将无法操作设备。

第四步：配网

通过SDK初始化 我们已经可以正常使用SDK，现在我们为设备配置上网环境。

1.设备联网

设备配网模块用来为设备配置上网环境，目前支持以下配网方式:

- 有线配网
- 扫码配网
- AP配网

⚠️注意：并非任意设备都支持以上所有配网方式，具体支持的配网方式由硬件和固件版本决定。

> 详见【设备配网】

2.设备绑定

设备绑定具体操作请参见 终端用户绑定设备接口 进行设备绑定。

绑定成功后，获取到订阅token,需调用命令使IoTVideo SDK订阅该设备：

```
import IoTVideo.IVNetConfig

IVNetConfig.subscribeDevice(withToken: "********", deviceId: deviceId)
```

第五步：监控

使用内置的多媒体模块可以轻松实现设备监控。

```
import IoTVideo

// 1.创建监控播放器
let monitorPlayer = IVMonitorPlayer(deviceId: device.deviceID)
// 2.设置播放器代理（回调）
monitorPlayer.delegate = self
// 3.添加播放器渲染图层
videoView.insertSubview(monitorPlayer.videoView!, at: 0)
monitorPlayer.videoView?.frame = videoView.bounds
// 4.预连接，获取流媒体头信息
monitorPlayer.prepare()
// 5.开始播放，启动推拉流、渲染模块
```

```
monitorPlayer.play()
// 6.开启/关闭语音对讲（只支持MonitorPlayer/LivePlayer）
monitorPlayer.startTalk()
monitorPlayer.stopTalk()
// 7.停止播放，断开连接
monitorPlayer.stop()
```

> 详见【多媒体】

第六步：消息管理

```
import IoTVideo.IVMessageMgr

// 设备ID的字符串
let deviceId = dev.deviceId
// 模型路径的字符串
let path = "ProWritable._logLevel"
// 模型参数的字符串
let json = "{\"setVal\":0}"

// 1.读取属性
IVMessageMgr.sharedInstance.readProperty(ofDeviceId: deviceId, path: path) {
(json, error) in
    // do something here
}

// 2.设置属性
IVMessageMgr.sharedInstance.writeProperty(ofDevice:deviceId, path: path, json:
json) { (json, error) in
    // do something here
}

// 3.执行动作
// 模型路径的字符串
let actionPath = "Action.cameraOn"
// 模型参数的字符串
let actionJson = "{\"ctlVal\":1}"

IVMessageMgr.sharedInstance.takeAction(ofDevice: deviceId, path: actionPath,
json: actionJson) { (json, error) in
    // do something here
}
```

> 详见【消息管理】

集成指南

本节主要介绍如何快速地将IoTVideoSDK集成到您的项目中，按照如下步骤进行配置，就可以完成 SDK 的集成工 作。

开发环境要求

- Xcode 9.3+
- iOS 9.0+

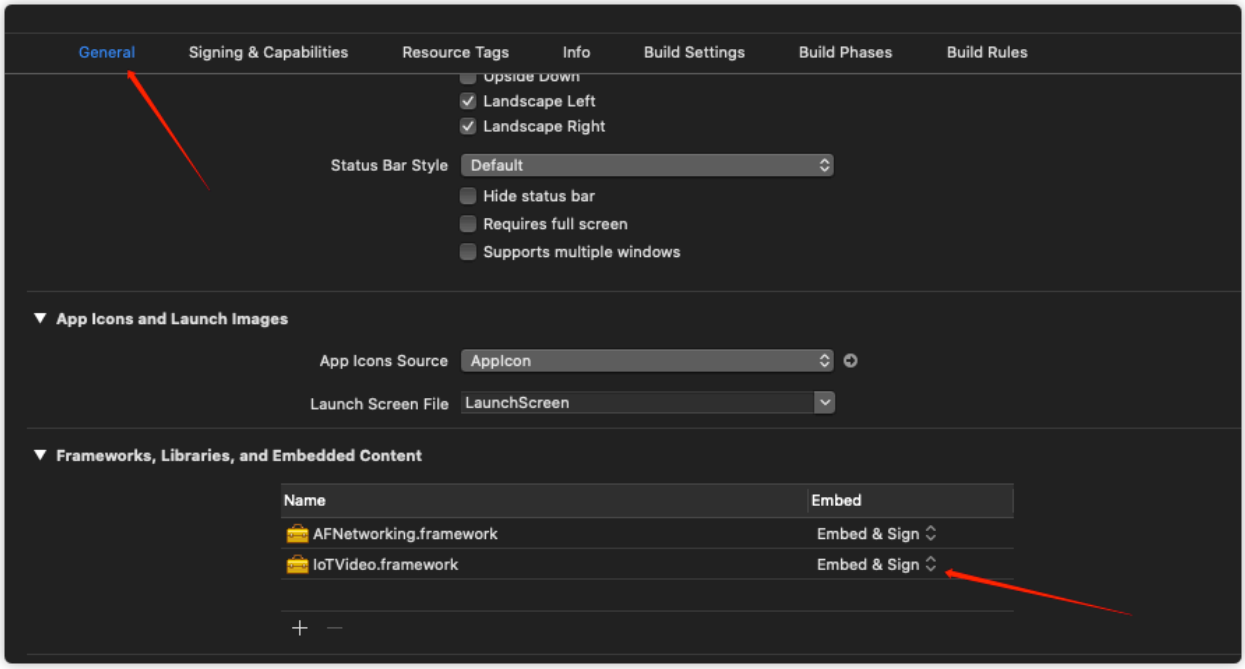
集成 SDK

1. 登录 物联网智能视频服务控制台 进行申请，申请完成后，相关工作人员将联系您进行需求沟通，并提供对应 SDK 及 Demo。

2. 将下载并解压得到的IoTVideo相关Framework添加到工程中

- IoTVideo.framework //核心库
- IVVAS.framework //增值服务相关

⚠️注意： 需要设置TATGETS -> Build Phases -> Embed Frameworks为 Embed & sign，或者Xcode11后可在General -> Frameworks,Libraries,and Embedded Content设置Embed&Sign



3. 添加系统的 Framework

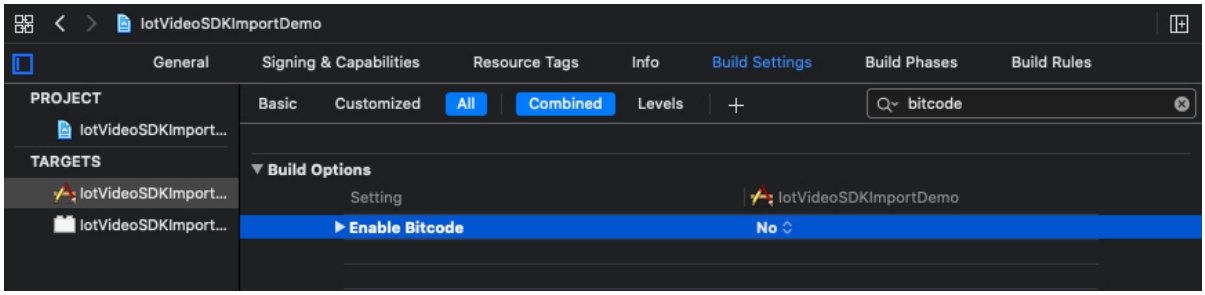
- AudioToolbox.framework
- VideoToolbox.framework
- CoreMedia.framework
- libz.tbd

4. 添加其他第三方库依赖

- AFNNetWorking 3.0+ //*添加方法参考GitHub*

5. 其他设置

- 关闭bitcode： TARGETS -> Build Settings -> Build Options -> Enable Bitcode -> NO



- 设置APP权限,在info.plist中加入下方代码

```
<key>NSCameraUsageDescription</key>
<string>访问相机</string>
<key>NSMicrophoneUsageDescription</key>
<string>访问麦克风</string>
<key>NSPhotoLibraryAddUsageDescription</key>
<string>访问相册</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>访问相册</string>
```

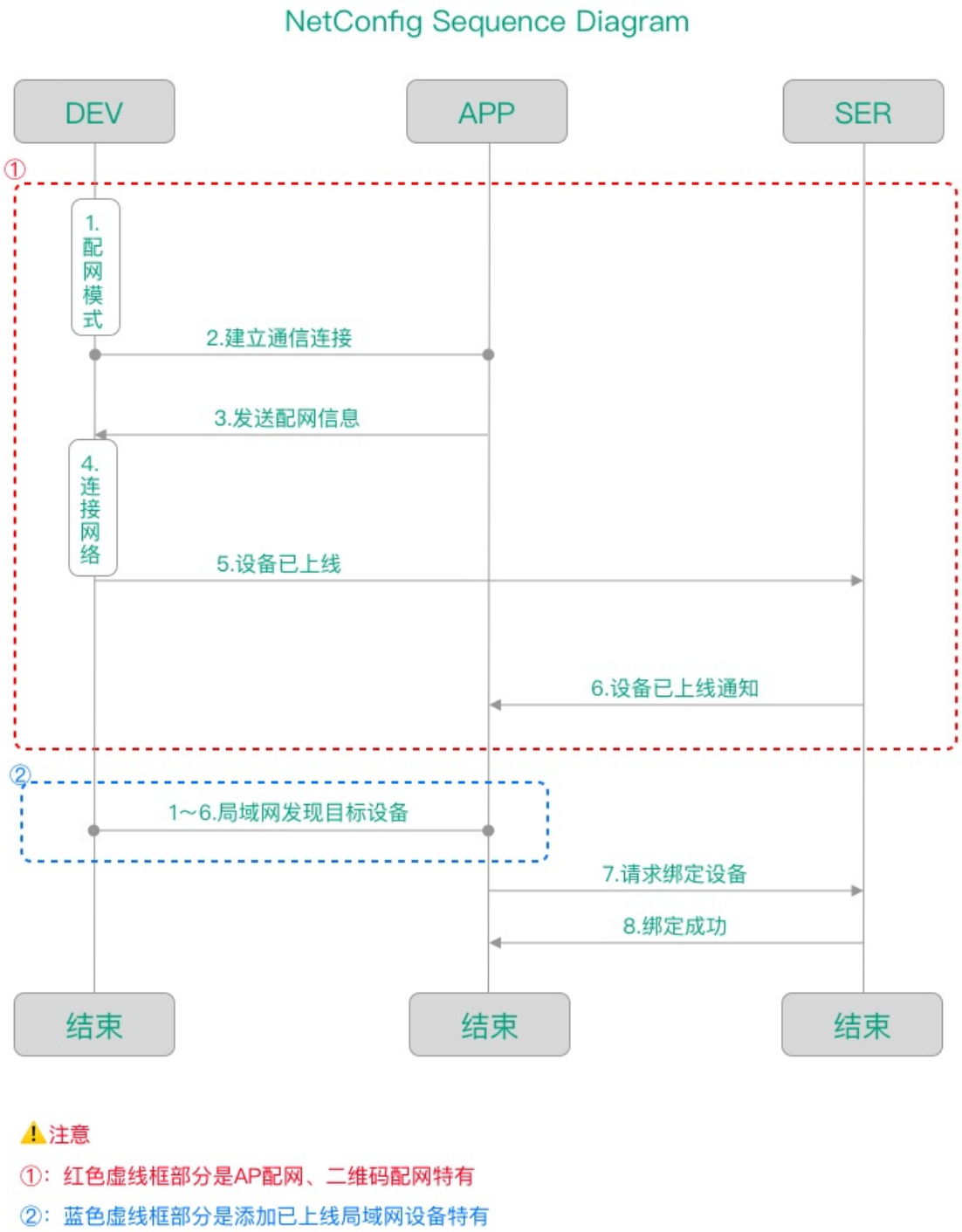
设备配网

设备配网模块用来为设备配置上网环境，目前支持以下配网方式:

- 有线配网
- 扫码配网
- AP配网

⚠️注意：并非任意设备都支持以上所有配网方式，具体支持的配网方式由硬件和固件版本决定。

通用配网流程



有线配网（添加已在线局域网设备，需设备硬件支持）

部分设备可通过自带网口使用有线上网，省去了配网环节，APP可通过局域网搜索到目标设备，使用设备ID向服务器发起绑定请求。
流程大致如下：

> 1. APP连接到与设备同一网络下的Wi-Fi
> 2. APP搜索到目标设备，取得目标设备ID
> 3. APP向服务器发起绑定目标设备的请求
> 4. 账户绑定设备成功
> 5. 订阅该设备
> 6. 配网结束

AP配网（需设备硬件支持）

AP配网原理是APP连接设备发射的热点，使设备与APP处于同一局域网，并在局域网下实现信息传递。
流程大致如下：

> 1. 设备复位进入配网模式并发射Wi-Fi热点
> 2. APP连接设备的热点（进入局域网）
> 3. APP向设备发送配网信息（Wi-Fi信息）
> 4. 设备收到配网信息并连接指定网络
> 5. 设备上线并向服务器注册
> 6. APP收到设备已上线通知
> 7. APP向服务器发起绑定目标设备的请求
> 8. 账户绑定设备成功
> 9. 订阅该设备
> 10. 配网结束

二维码配网

二维码配网原理是APP使用配网信息生成二维码，设备通过摄像头扫描二维码获取配网信息。
流程大致如下：

> 1. 设备复位进入配网模式，摄像头开始扫描二维码
> 2. APP使用配网信息生成二维码
> 3. 用户使用设备扫描二维码
> 4. 设备获取配网信息并连接指定网络
> 5. 设备上线并向服务器注册
> 6. APP收到设备已上线通知
> 7. APP向服务器发起绑定目标设备的请求
> 8. 账户绑定设备成功
> 9. 订阅该设备
> 10. 配网结束

使用示例

1.有线配网

```
import IoTVideo.IVNetConfig

// 1.获取局域网设备列表
let deviceList: [IVLANDevice] = IVNetConfig.lan.getDeviceList()

// 2.取得目标设备
let dev = deviceList[0]

// 3.绑定设备

// 4.订阅设备
IVNetConfig.subscribeDevice(withToken: "********", deviceId: deviceId)
```

设备绑定具体操作请参见 终端用户绑定设备接口 进行设备绑定。

2.AP配网

```
import IoTVideo.IVNetConfig

// 1.向服务器请求配网ID

// 2.连接设备热点

// 3.发送配网信息
IVNetConfig.lan.sendWifiName("********", wifiPwd: "********", toDevice:
"********") { (success, error) in
    if success {
        // 发送成功
    } else {
        // 发送失败
    }
}

// 4.等待设备上线通知...

// 5.绑定设备

// 6.订阅设备
IVNetConfig.subscribeDevice(withToken: "********", deviceId: deviceId)
```

设备绑定具体操作请参见 终端用户绑定设备接口 进行设备绑定。

3.二维码配网

接入方自定义传递给设备的数据格式，可使用内置工具类生成二维码，也可自行生成二维码

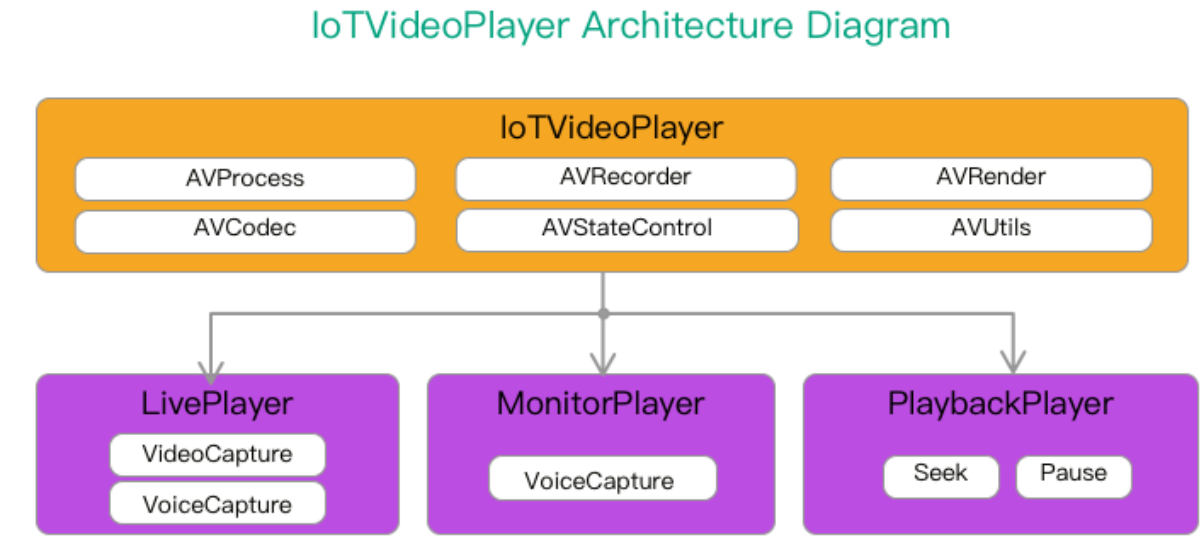
```
import IoTVideo.IVNetConfig

// 1.生成二维码图片

IVNetConfig.qrCode.createQRCode(withWifiName: wifiName, wifipwd: wifiPwd,
qrSize: size, completionHandler: { (image, error)  in
     // get the image
 })

 //或使用下方API进行额外的语言及时区的传输
 //langugae: 语言 默认 0 中文 可参考 IV_QR_CODE_LANGUAGE 枚举同时需要固件支持设置多语言
 //timeZone: 时区（单位分钟） 默认 480 即北京时区28800/60
IVNetConfig.qrCode.createQRCode(withWifiName: wifiName, wifipwd: wifiPwd,
language:0, timeZone: 480, qrSize: size, completionHandler: { (image, error)
 in
     // get the image
```

```
 })

// 2.用户使用设备扫描二维码....

// 3.等待设备上线通知...

// 4.绑定设备

// 5.订阅设备
IVNetConfig.subscribeDevice(withToken: "********", deviceId: deviceId)
```

设备绑定具体操作请参见 终端用户绑定设备接口 进行设备绑定。

多媒体

多媒体模块为SDK提供音视频能力，包含实时监控、实时音视频通话、远程回放、录像、截图等功能。



播放器核心(IVPlayer)

IVPlayer是整个多媒体模块的核心，主要负责以下流程控制：

- 音视频通道建立
- 音视频流的推拉
- 协议解析
- 封装和解封装
- 音视频编解码
- 音视频同步
- 音视频渲染
- 音视频录制
- 播放状态控制

其中，*音视频编解码* 和 *音视频渲染* 流程允许开发者自定义实现（⚠️*播放器已内置实现，不推荐自定义实现*）

监控播放器(MonitorPlayer)

MonitorPlayer是基于IVPlayer派生的监控播放器，主要增加以下功能：

- 语音对讲

音视频通话播放器(LivePlayer)

LivePlayer是基于IVPlayer派生的音视频通话播放器，主要增加以下功能：

- 语音对讲
- 双向视频

回放播放器(PlaybackPlayer)

PlaybackPlayer是基于IVPlayer派生的回放播放器，主要增加以下功能：

- 暂停/恢复
- 跳至指定位置播放

播放器功能对比

功能	监控播放器	回放播放器	音视频通话
视频播放	✓	✓	✓
音频播放	✓	✓	✓
暂停/恢复	x	✓	x
跳至指定位置播放	x	✓	x
总时长	x	✓	x
当前播放进度	x	✓	x
播放器状态变更通知	✓	✓	✓
静音	✓	✓	✓
画面缩放模式设置	✓	✓	✓
播放器截图	✓	✓	✓
边播边录	✓	✓	✓
对讲	✓	✓	✓
分辨率切换	✓	x	x
双向视频	x	x	✓

使用示例

1.创建播放器实例

```
import IoTVideo

// 监控播放器
let monitorPlayer = IVMonitorPlayer(deviceId: device.deviceID)
// 音视频通话播放器
let livePlayer = IVLivePlayer(deviceId: device.deviceID)
// 回放播放器
let playbackPlayer = IVPlaybackPlayer(deviceId: device.deviceID, playbackItem:
item, seekToTime: time)
```

⚠️*注意：以下使用`xxxxPlayer`泛指支持该功能的播放器*

2.设置播放器代理（回调）

```
xxxxPlayer.delegate = self
```

3.添加摄像头预览图层(只支持LivePlayer)

```
previewView.layer.addSublayer(livePlayer.previewLayer)
livePlayer.previewLayer.frame = previewView.bounds
```

4.添加播放器渲染图层

```
videoView.insertSubview(xxxxPlayer.videoView!, at: 0)
xxxxPlayer.videoView?.frame    = videoView.bounds
```

5.预连接（可选），获取流媒体头信息

```
xxxxPlayer.prepare()
```

6.开始播放，启动推拉流、渲染模块

```
xxxxPlayer.play()
```

7.开启/关闭语音对讲（只支持MonitorPlayer/LivePlayer）

```
xxxxPlayer.startTalk()
xxxxPlayer.stopTalk()
```

8.开启/切换/关闭摄像头（只支持LivePlayer）

```
//打开摄像头
livePlayer.openCamera()
//切换摄像头
livePlayer.switchCamera()
//关闭摄像头
livePlayer.closeCamera()
```

9.指定时间播放(只支持PlaybackPlayer)

```
playbackPlayer.seek(toTime: time, playbackItem: item)
```

10.暂停/恢复播放(只支持PlaybackPlayer)

```
//暂停
playbackPlayer.pause()
//恢复
playbackPlayer.resume()
```

11.停止播放，断开连接

```
xxxxPlayer.stop()
```

高级功能

> ⚠️注意：音视频编解码及渲染已默认由核心播放器实现。如无必要，无需另行实现。

自定义音视频编解码

可选实现并赋值给核心播放器（IVPlayer）中的以下音视频编解码器即可自定义编解码器：

```
/// 视频解码
open var videoDecoder: IVVideoDecodable?
/// 视频编码
open var videoEncoder: IVVideoEncodable?
/// 音频解码
open var audioDecoder: IVAudioDecodable?
/// 音频编码
open var audioEncoder: IVAudioEncodable?
```

自定义音视频渲染

- 获取PCM音频数据

```
/// 获取PCM音频数据，建议由音频播放单元驱动（例如在playbackCallback中调用该方法）
/// @param aframe [IN][OUT]用于接收音频帧数据
/// @note aframe入参时 `aframe->data`不可为NULL，aframe->size`不可为0；
/// @return [YES]成功，[NO]失败
open func getAudioFrame(_ aframe: UnsafeMutablePointer<IVAudioFrame>) -> Bool
```

- 获取YUV视频数据

> 方式一：通过回调获取视频帧

```
/// 视频渲染器
/// 视频渲染器，默认为内置渲染器。 ⚠️如无必要 请勿修改
/// 仅当`syncAudio=YES`时有效。
/// @see 参见`IVVideoRenderable`
open var videoView: (UIView & IVVideoRenderable)?
```

> 方式二：手动获取视频帧

```
/// 获取YUV视频数据
/// 仅当`syncAudio=NO`时有效。
/// @param vframe [IN][OUT]用于接收视频帧数据
/// @note vframe入参时 `vframe->data[i]`不可为NULL，vframe->linesize[i]`不可为0；
/// @return [YES]成功，[NO]失败
open func getVideoFrame(_ vframe: UnsafeMutablePointer<IVVideoFrame>) -> Bool
```

消息管理

消息管理模块负责APP与设备、服务器之间的消息传递，主要包含以下功能：

- 在线消息回调
  - 接收到事件消息（Event）：告警、分享、系统通知
  - 接收到状态消息（ProReadonly）
- 控制/操作设备（Action）
- 设置设备参数（ProWritable）
- 获取设备状态（ProReadonly）
- 获取设备参数（ProWritable）
- 自定义消息透传 (Data)

使用示例

1.状态和事件消息通知

```
import IoTVideo.IVMessageMgr
```

```
class MyViewController: UIViewController, IVMessageDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        // 设置消息代理
        IVMessageMgr.sharedInstance.delegate = self
    }

    // MARK: - IVMessageDelegate

    // 接收到事件消息（Event）：  告警、分享、系统通知
    func didReceiveEvent(_ event: String, topic: String) {
        // do something here
    }

    // 接收到状态消息（ProReadonly）
    func didUpdateProperty(_ json: String, path: String, deviceId: String) {
        // do something here
    }
}
```

3.读取属性

```
import IoTVideo.IVMessageMgr

// 设备ID的字符串
let deviceId = dev.deviceId
// 模型路径的字符串
let path = "ProWritable._logLevel"

IVMessageMgr.sharedInstance.readProperty(ofDevice: deviceId, path: path) {
(json, error) in
    // do something here
}
```

4.设置属性

```
import IoTVideo.IVMessageMgr

// 设备ID的字符串
let deviceId = dev.deviceId
// 模型路径的字符串
let path = "ProWritable._logLevel"
// 模型参数的字符串
let json = "{\"setVal\":0}"

IVMessageMgr.sharedInstance.writeProperty(ofDevice: deviceId, path: path, json:
json) { (json, error) in
    // do something here
}
```

5.执行动作

```
import IoTVideo.IVMessageMgr

// 设备ID的字符串
let deviceId = dev.deviceId
// 模型路径的字符串
let path = "Action.cameraOn"
// 模型参数的字符串
let json = "{\"ctlVal\":1}"

IVMessageMgr.sharedInstance.takeAction(ofDevice: deviceId, path: path, json:
json) { (json, error) in
    // do something here
}
```

高级功能

1.透传数据给设备

```
/// 透传数据给设备（无数据回传）
/// 使用在不需要数据回传的场景，如发送控制指令
/// @note 完成回调条件：收到ACK 或 消息超时
/// @param deviceId 设备ID
/// @param data 数据内容
/// @param completionHandler 完成回调
func sendData(toDevice deviceId: String, data: Data, withoutResponse
completionHandler: IVMsgDataCallback? = nil)


/// 透传数据给设备（有数据回传）
/// 使用在预期有数据回传的场景，如获取信息
/// @note 完成回调条件：收到ACK错误、消息超时 或 有数据回传
```

```
/// @param deviceId 设备ID
/// @param data 数据内容
/// @param completionHandler 完成回调
func sendData(toDevice deviceId: String, data: Data, withResponse
completionHandler: IVMsgDataCallback? = nil)


/// 透传数据给设备
/// 可使用在需要数据回传的场景，如获取信息
/// @note 可以等待有数据回传时才完成回调，如忽略数据回传可简单使用
`sendDataToDevice:data:completionHandler:`代替。
/// @param deviceId 设备ID
/// @param data 数据内容
/// @param timeout 自定义超时时间，默认超时时间可使用@c `IVMsgTimeoutAuto`
/// @param expectResponse 【YES】预期有数据回传 ； 【NO】忽略数据回传
/// @param completionHandler 完成回调
func sendData(toDevice deviceId: String, data: Data, timeout: TimeInterval,
expectResponse: Bool, completionHandler: IVMsgDataCallback? = nil)
```

2.透传数据给服务器

```
/// 透传数据给服务器
/// @param url 服务器路径
/// @param data 数据内容
/// @param completionHandler 完成回调
func sendData(toServer url: String, data: Data?, completionHandler:
IVMsgDataCallback? = nil)


/// 透传数据给服务器
/// @param url 服务器路径
/// @param data 数据内容
/// @param timeout 超时时间
/// @param completionHandler 完成回调
func sendData(toServer url: String, data: Data?, timeout: TimeInterval,
completionHandler: IVMsgDataCallback? = nil)
```