NetSDK_JAVA 编程指导手册

（智能楼宇分册）

V2.0.1

前言

概述

欢迎使用 NetSDK_JAVA（以下简称 SDK）编程指导手册。

SDK 是软件开发者在开发网络硬盘录像机、网络视频服务器、网络摄像机、网络球机和智能设备等产品监控联网应用时的开发套件。

本文档描述了智能楼宇产品的通用业务涉及的 SDK 接口以及调用流程。

基础核心业务流程（例如初始化、登录、普通报警和智能报警等）详见《NetSDK 编程指导手册》。

本文档提供的示例代码仅为演示接口调用方法，不保证能直接拷贝编译。

读者对象

使用 SDK 的软件开发工程师、项目经理和产品经理。

符号约定

在本文档中可能出现下列标志，代表的含义如下。

符号	说明
窍门	表示能帮助您解决某个问题或节省您的时间。
说明	表示是正文的附加信息，是对正文的强调和补充。

修订记录

版本号	修订内容	发布日期
V2.0.1	新增智能报警事件	2025.07
V2.0.0	前言补充	2025.02
V1.0.0	首次发布。	2023.12

名词解释

以下对本文档中使用的专业名词分别说明，帮助您更好的理解各个业务功能。

名词	说明
防区	报警输入通道可接收外部探测的信号，每一路报警输入通道即成为一个防区。
布防和撤防	● 布防：设备已布防的防区处于警戒状态，接收外部信号、处理、记录和转发。 ● 撤防：设备已撤防的防区不会接收外部报警信号、也不会处理、记录和转发。
旁路	在设备处于布防时，某个防区对外部探测器仍然监听，也会做记录，但不会转发给用户。在设备撤防后，原来旁路的防区会转变成正常状态，当再次布防时，该防区布防成功。
消警	设备发出报警后，通常会执行一些联动操作，比如蜂鸣、报警键盘提示等，这些操作往往会持续一段时间，消警是使这些联动操作提前结束。
即时防区	在设备布防状态下，该防区触发报警时，立刻记录并转发报警信号。
延时防区	防区类型为延时防区时，可设置进入延时或者退出延时。 进入延时，即用户在延时时间内从非布防区域进入布防区域时产生报警，但不联动报警，延时时间结束后，如果没撤防，此时会联动报警；如果撤防了，就不会联动报警。设置退出延时后，设备会在退出延时结束后再进入布防状态。
24 小时防区	一经配置成 24 小时防区，立即生效，且布撤防动作对其无效，可应用于火警等场景。
情景模式	报警主机现有情景模式两种，分别为“外出模式”和“在家模式”，个模式会有相关的配置，用户选择哪种情景模式，即可使相应配置生效。
在家模式/外出模式	当情景模式切换到“在家模式”或“外出模式”时，会使该情景模式下预设的防区布防，其余防区变为旁路状态。
隔离	一种对入侵报警探测回路的设置，处于此状态的入侵报警探测回路不能通告报警。此状态一直保持到人为复位。
模拟量报警通道(模拟量防区)	设备有多个报警输入通道，接收外部探测的信号。通道类型为模拟量时，则称为模拟量报警通道，可连接模拟量探测器，用于采集模拟量数据。
胁迫卡	门禁卡的一种，用户被胁迫时使用胁迫卡开门，门禁系统识别出是胁迫卡刷卡，并发出报警信号。

目录

第 1 章 产品概述

1.1 概述

本文档主要介绍 SDK 接口参考信息，包括主要功能、接口函数和回调函数。

主要功能包括：通用功能、报警主机、门禁等功能。

根据环境不同，开发包包含的文件会不同，具体如下所示。

- Windows 开发包所包含的文件如下：

表1-1 开发包包括的文件

库类型	库文件名称	库文件说明
功能库	dhnetsdk.h	头文件
	dhnetsdk.dll	库文件
	avnetsdk.dll	库文件
	dhconfigsdk.h	头文件
	dhconfigsdk.dll	库文件
播放（编码解码）辅助库	dhplay.dll	播放库
	StreamConvertor.dll	转码库

- Linux 开发包所包含的文件如下：

表1-2 开发包包括的文件

库类型	库文件名称	库文件说明
功能库	dhnetsdk.h	头文件
	libdhnetsdk.so	库文件
	libavnetsdk.so	库文件
	dhconfigsdk.h	头文件
	libdhconfigsdk.so	库文件

说明

- SDK 的功能库和配置库是必备库。
- 功能库是设备网络 SDK 的主体，主要用于网络客户端与各类产品之间的通讯交互，负责远程控制、查询、配置及码流数据的获取和处理等。
- 配置库针对配置功能的结构体进行打包和解析。
- 推荐使用播放库进行码流解析和播放。
- 辅助库用于预览、回放、对讲等功能的音视频码流解码以及本地音频采集。

1.2 适用性

1.2.1 适用系统

- 推荐内存：不低于 512 MB。
- SDK 支持的系统：
  - ◇ Windows

    Windows 10/Windows 8.1/Windows 7/vista/XP/2000 以及 Windows Server 2008/2003。

- Linux
  Red Hat/SUSE 等通用 Linux 系统。

1.2.2 支持设备

- 门禁设备（一代设备）
  - DH-ASC1201C-D
  - DH-ASC1202B-D、DH-ASC1202B-S、DH-ASC1202C-D、DH-ASC1202C-S
  - DH-ASC1204B-S、DH-ASC1204C-D、DH-ASC1204C-S
  - DH-ASC1208C-S
  - DH-ASI1201A、DH-ASI1201A-D、DH-ASI1201E-D、DH-ASI1201E
  - DH-ASI1212A(V2)、DH-ASI1212A-C(V2)、DH-ASI1212A-D(V2)、DH-ASI1212D、DH-ASI1212D-D
  - DHI-ASC1201B-D、DHI-ASC1201C-D
  - DHI-ASC1202B-D、DHI-ASC1202B-S、DHI-ASC1202C-D、DHI-ASC1202C-S
  - DHI-ASC1204B-S、DHI-ASC1204C-D、DHI-ASC1204C-S
  - DHI-ASC1208C-S
  - DHI-ASI1201A、DHI-ASI1201A-D、DHI-ASI1201E-D、DHI-ASI1201E
  - DHI-ASI1212A(V2)、DHI-ASI1212A-D(V2)、DHI-ASI1212D、DHI-ASI1212D-D
  - ASC1201B-D、ASC1201C-D
  - ASC1202B-S、ASC1202B-D、ASC1202C-S、ASC1202C-D
  - ASC1204B-S、ASC1204C-S、ASC1204C-D
  - ASC1208C-S
  - ASI1201A、ASI1201A-D、ASI1201E、ASI1201E-D
  - ASI1212A(V2)、ASI1212A-D(V2)、ASI1212D、ASI1212D-D
- 门禁设备（二代设备）
  - DH-ASI4213Y
  - DH-ASI4214Y
  - DH-ASI7213X、DH-ASI7213X-C、DH-ASI7213Y、DH-ASI7213Y-V3
  - DH-ASI7214X、DH-ASI7214X-C、DH-ASI7214Y、DH-ASI7214Y-V3
  - DH-ASI7223X-A、DH-ASI7223Y-A、DH-ASI7223Y-A-V3
  - DH-ASI8213Y(V2)、DH-ASI8213Y-C(V2)、DH-ASI8213Y-V3
  - DH-ASI8214Y、DH-ASI8214Y(V2)、DH-ASI8214Y-C(V2)、DH-ASI8214Y-V3
  - DH-ASI8215Y、DH-ASI8215Y(V2)、DH-ASI8215Y-V3
  - DH-ASI8223Y(V2)、DH-ASI8223Y-A(V2)、DH-ASI8223Y、DH-ASI8233Y-A-V3
  - DHI-ASI1202M、DHI-ASI1202M-D
  - DHI-ASI4213Y、DHI-ASI4214Y
  - DHI-ASI7213X、DHI-ASI7213Y、DHI-ASI7213Y-D、DHI-ASI7213Y-V3
  - DHI-ASI7214X、DHI-ASI7214Y、DHI-ASI7214Y-D、DHI-ASI7214Y-V3
  - DHI-ASI7223X-A、DHI-ASI7223Y-A、DHI-ASI7223Y-A-V3
  - DHI-ASI8213Y-V3
  - DHI-ASI8214Y、DHI-ASI8214Y(V2)、DHI-ASI8214Y-V3
  - DHI-ASI8223Y、ASI8223Y(V2)、DHI-ASI8223Y-A(V2)、DHI-ASI8223Y-A-V3
  - ASI1202M、ASI1202M-D
  - ASI7213X、ASI7213Y-D、ASI7213Y-V3
  - ASI7214X、ASI7214Y、ASI7214Y-D、ASI7214Y-V3
  - ASI7223X-A、ASI7223Y-A、ASI7223Y-A-V3

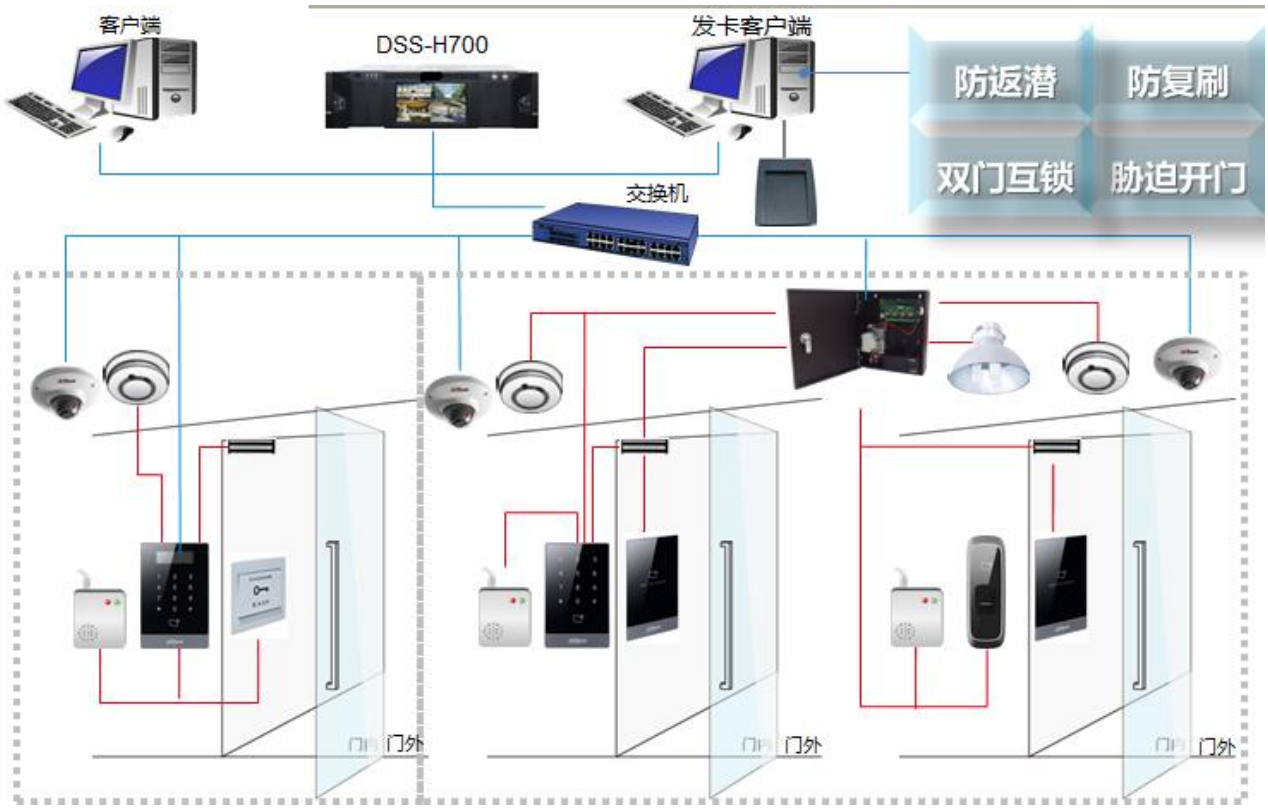
- ◇ ASI8213Y-V3
- ◇ ASI8214Y、ASI8214Y(V2)、ASI8214Y-V3
- ◇ ASI8223Y、ASI8223Y(V2)、ASI8223Y-A(V2)、ASI8223Y-A-V3

- 可视对讲设备
  - ◇ VTA8111A
  - ◇ VTO1210B-X、VTO1210C-X
  - ◇ VTO1220B
  - ◇ VTO2000A、VTO2111D
  - ◇ VTO6210B、VTO6100C
  - ◇ VTO9231D、VTO9241D
  - ◇ VTH1510CH、VTH1510A、VTH1550CH
  - ◇ VTH5221D、VTH5241D
  - ◇ VTS1500A、VTS5420B、VTS8240B、VTS8420B
  - ◇ VTT201、VTT2610C
- 报警主机
  - ◇ ARC2008C、ARC2008C-G、ARC2016C、ARC2016C-G、ARC5408C、ARC5408C-C、ARC5808C、ARC5808C-C、ARC9016C、ARC9016C-G
  - ◇ DH-ARC2008C、DH-ARC2008C-G、DH-ARC2016C、DH-ARC2016C-G、DH-ARC5408C、DH-ARC5408C-C、DH-ARC5408C-E、DH-ARC5808C、DH-ARC5808C-C、DH-ARC5808C-E、DH-ARC9016C、DH-ARC9016C-G、
  - ◇ DHI-ARC2008C、DHI-ARC2008C-G、DHI-ARC2016C、DHI-ARC2016C-G、DHI-ARC5808C、DHI-ARC5808C-C、DHI-ARC5408C、DHI-ARC5408C-C、DHI-ARC9016C、DHI-ARC9016C-G、
  - ◇ ARC2008C、ARC2008C-G、ARC2016C、ARC2016C-G、ARC5408C、ARC5408C-C、ARC5408C-E、ARC5808C-C、ARC5808C、ARC5808C-E、ARC9016C、ARC9016C-G

1.3 应用场景

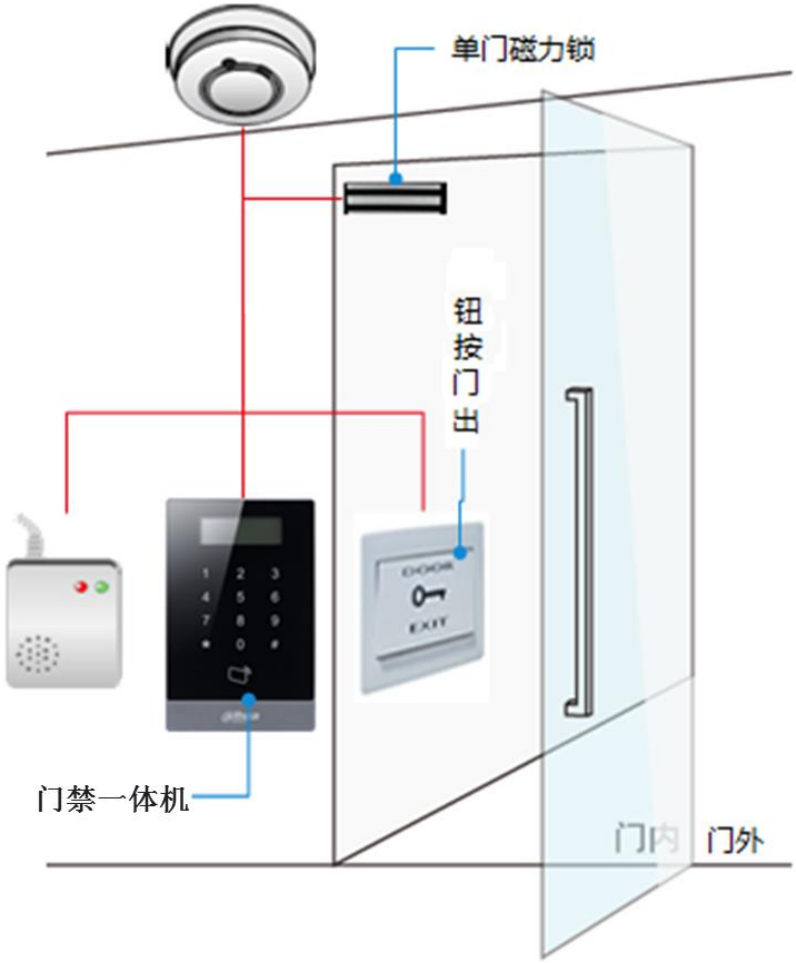
- 典型场景。

图1-1 典型场景



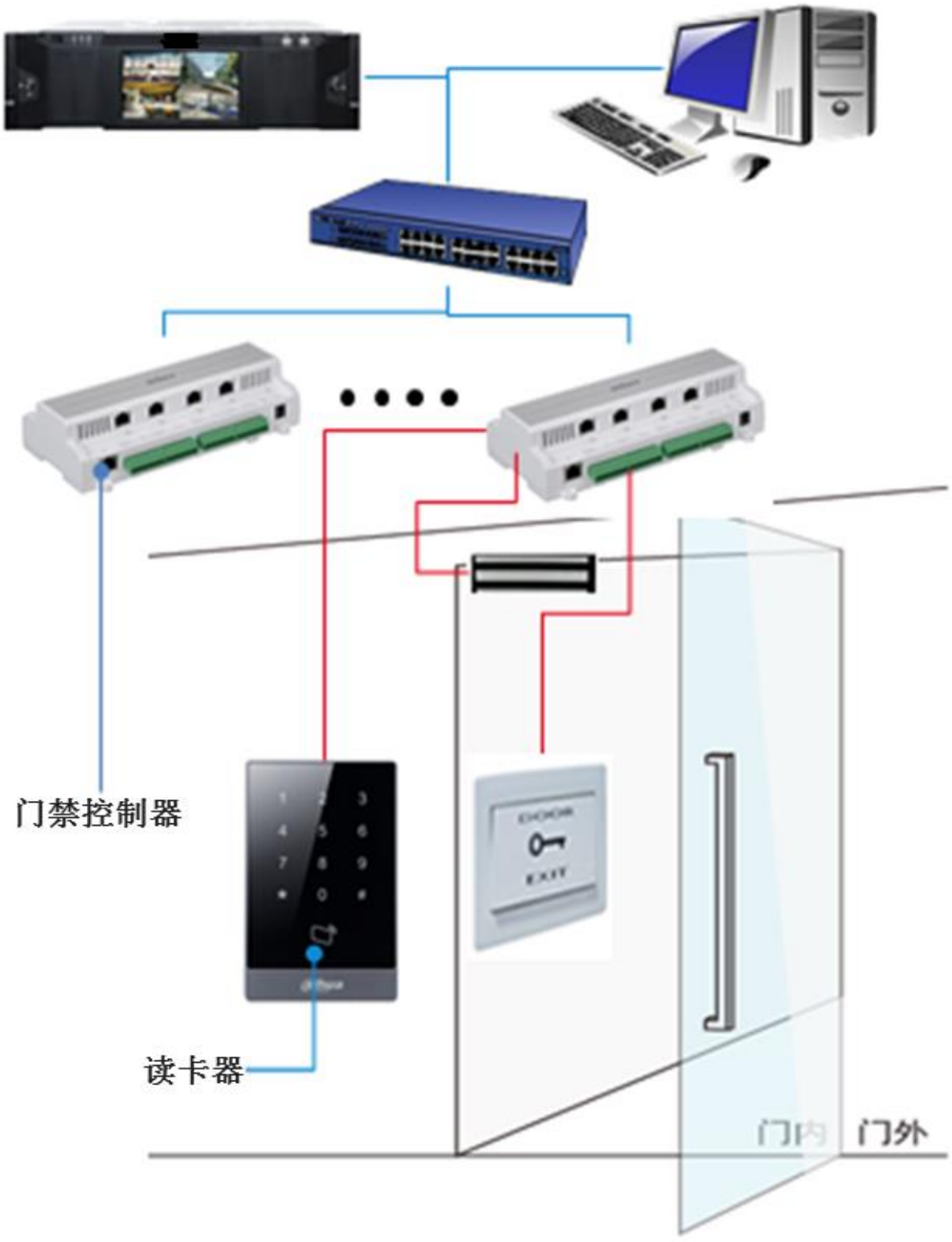
- 微型门禁适用于小型办公。

图1-2 微型门禁



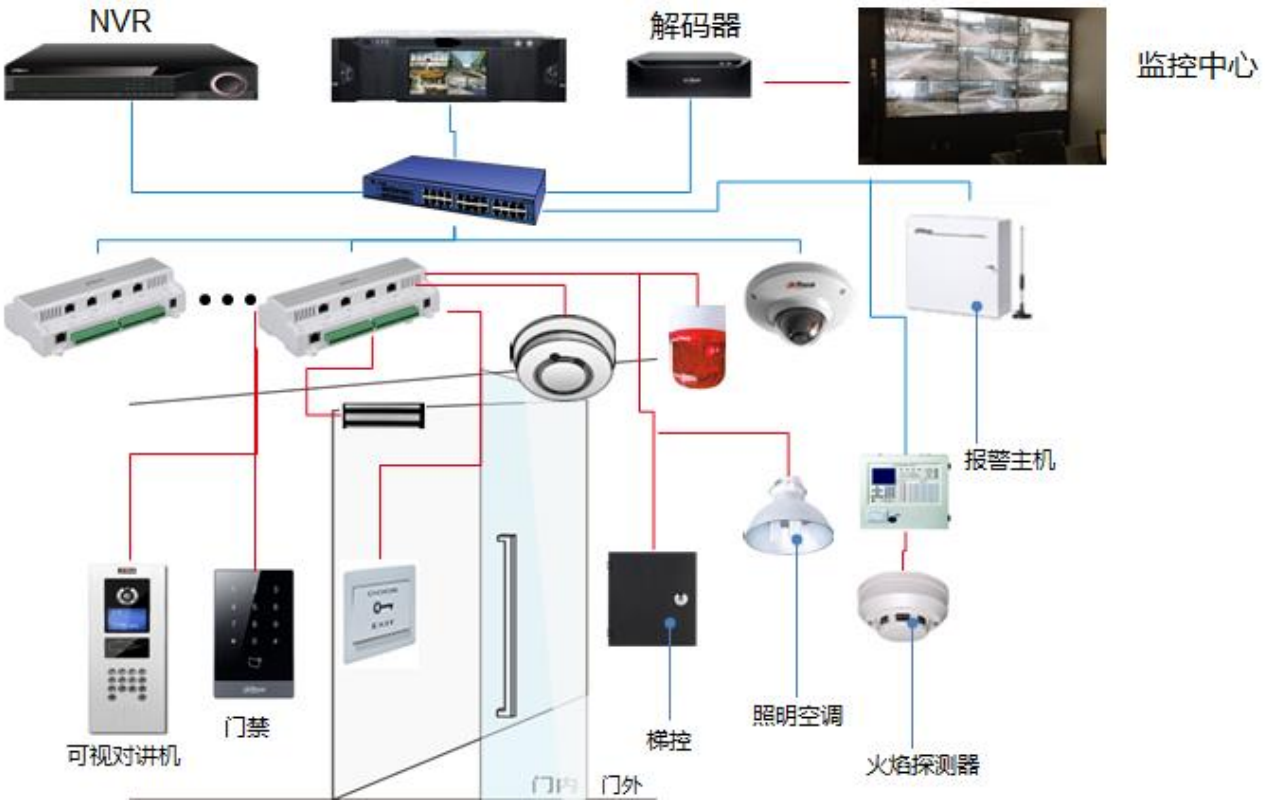
- 网络型门禁适用于中小型以上智能楼宇项目及金库和监所项目。

图1-3 网络型门禁



- 增强型门禁。

图1-4 增强型门禁



第 2 章 主要功能

2.1 通用

2.1.1 SDK 初始化

2.1.1.1 简介

初始化是 SDK 进行各种业务的第一步。初始化本身不包含监控业务，但会设置一些影响全局业务的参数。

- SDK 的初始化将会占用一定的内存。
- 同一个进程内，只有第一次初始化有效。
- 使用完毕后需要调用 CLIENT_Cleanup 释放资源。

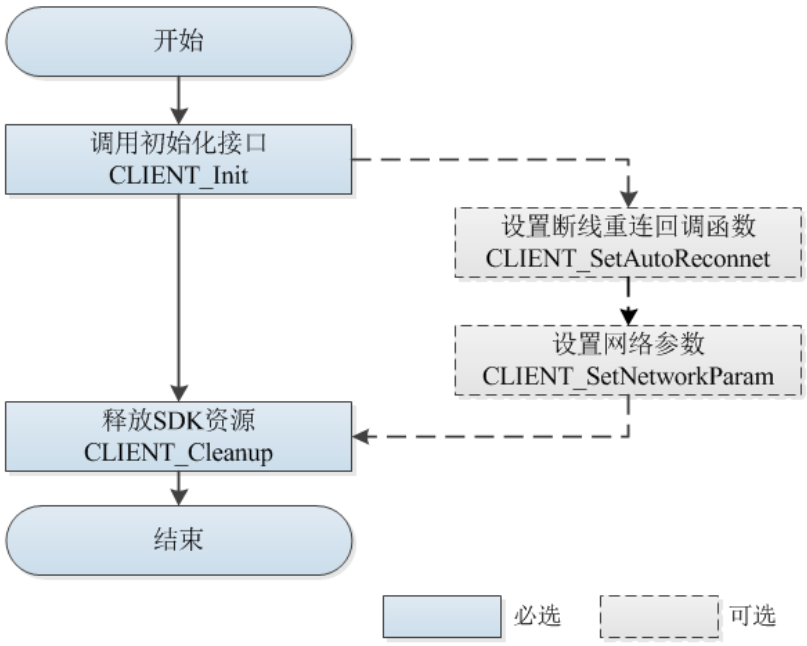
2.1.1.2 接口总览

表2-1 SDK 初始化接口说明

接口	说明
CLIENT_Init	SDK 初始化接口
CLIENT_Cleanup	SDK 清理接口
CLIENT_SetAutoReconnect	设置断线重连回调接口
CLIENT_SetNetworkParam	设置登录网络环境接口

2.1.1.3 流程说明

图2-1 SDK 初始化业务流程



流程说明

步骤1　调用 CLIENT_Init 完成 SDK 初始化流程。

步骤2　（可选）调用 CLIENT_SetAutoReconnect 设置断线重连回调函数，设置后 SDK 内部断线自动重连。

步骤3　（可选）调用 CLIENT_SetNetworkParam 设置网络登录参数，参数中包含登录设备超时时间和尝试次数。

步骤4　SDK 所有功能使用完后，调用 CLIENT_Cleanup 释放 SDK 资源。

注意事项

- SDK 的 CLIENT_Init 和 CLIENT_Cleanup 接口需成对调用，支持单线程多次成对调用，但建议全局调用一次。
- 初始化：CLIENT_Init 接口内部多次调用时，仅在内部用做计数，不会重复申请资源。
- 清理：CLIENT_Cleanup 接口内会清理所有已开启的业务，如登录、实时预览和报警订阅等。
- 断线重连：SDK 可以设置断线重连功能，当遇到一些特殊情况（例如断网、断电等）设备断线时，在 SDK 内部会定时持续不断地进行登录操作，直至成功登录设备。断线重连后可以恢复实时预览和录像回放业务，其他业务无法恢复。
- 加载动态库：如果加载动态库遇到报错如“Unable to load library './wrongpath/libs/win64/dhnetsdk': 找不到指定的模块”，通常是路径不匹配，需要根据报错信息调整动态库的位置或者修改代码。此问题多见于打包整个工程为 jar 包提供给其他项目时。由于此问题跟平台和工程使用方式相关，不能一概而论，需要具体分析。比如在 linux 平台下直接使用工程可以通过以下方式将动态库路径加载到动态库搜索路径中。
  1.在终端输入：export　LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/XXX　当前终端生效。
  2.修改~/.bashrc 或~/.bash_profile，最后一行添加 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/XXX，保存之后，使用 source .bashrc 执行该文件，当前用户生效。
  3.修改/etc/profile，添加内容如第 2 条，同样保存后用 source 执行该文件，所有用户生效。

2.1.1.4 示例代码

```
/**
 * 登录接口实现
 * 主要有 ：初始化、登录、登出功能
 */
public class LoginModule {

    public static NetSDKLib netsdk             = NetSDKLib.NETSDK_INSTANCE;
    public static NetSDKLib configsdk    = NetSDKLib.CONFIG_INSTANCE;

    // 登录句柄
    public static LLong m_hLoginHandle = new LLong(0);

    private static boolean bInit       = false;
```

```
    private static boolean bLogopen = false;

    //初始化
    public static boolean init(NetSDKLib.fDisConnect disConnect, NetSDKLib.fHaveReConnect haveReConnect) {
        bInit = netsdk.CLIENT_Init(disConnect, null);
        if(!bInit) {
            System.out.println("Initialize SDK failed");
            return false;
        }

        //打开日志，可选
        NetSDKLib.LOG_SET_PRINT_INFO setLog = new NetSDKLib.LOG_SET_PRINT_INFO();
        File path = new File("./sdklog/");
        if (!path.exists()) {
            path.mkdir();
        }
        String logPath = path.getAbsoluteFile().getParent() + "\\sdklog\\" + ToolKits.getDate() + ".log";
        setLog.nPrintStrategy = 0;
        setLog.bSetFilePath = 1;
        System.arraycopy(logPath.getBytes(), 0, setLog.szLogFilePath, 0, logPath.getBytes().length);
        System.out.println(logPath);
        setLog.bSetPrintStrategy = 1;
        bLogopen = netsdk.CLIENT_LogOpen(setLog);
        if(!bLogopen ) {
            System.err.println("Failed to open NetSDK log");
        }

        // 设置断线重连回调接口，设置过断线重连成功回调函数后，当设备出现断线情况，SDK内部会自动进行重连操作
        // 此操作为可选操作，但建议用户进行设置
        netsdk.CLIENT_SetAutoReconnect(haveReConnect, null);

        //设置登录超时时间和尝试次数，可选
        int waitTime = 5000; //登录请求响应超时时间设置为 5S
        int tryTimes = 1;      //登录时尝试建立链接 1 次
        netsdk.CLIENT_SetConnectTime(waitTime, tryTimes);
```

```
        // 设置更多网络参数，NET_PARAM 的 nWaittime，nConnectTryNum 成员与
CLIENT_SetConnectTime
        // 接口设置的登录设备超时时间和尝试次数意义相同,可选
        NetSDKLib.NET_PARAM netParam = new NetSDKLib.NET_PARAM();
        netParam.nConnectTime = 10000;          // 登录时尝试建立链接的超时时间
        netParam.nGetConnInfoTime = 3000;     // 设置子连接的超时时间
        netsdk.CLIENT_SetNetworkParam(netParam);

        return true;
    }

    //清除环境
    public static void cleanup() {
        if(bLogopen) {
            netsdk.CLIENT_LogClose();
        }

        if(bInit) {
            netsdk.CLIENT_Cleanup();
        }
    }
}
```

2.1.2 设备登录

2.1.2.1 简介

设备登录，即用户鉴权，是进行其他业务的前提。

用户登录设备产生唯一的登录 ID，其他功能的 SDK 接口需要传入登录 ID 才可执行。登出设备后，登录 ID 失效。

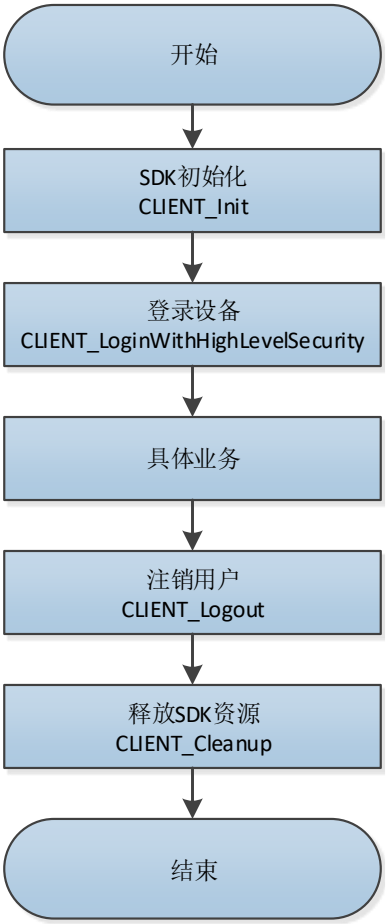
2.1.2.2 接口总览

表2-2 设备登录接口说明

接口	说明
CLIENT_LoginWithHighLevelSecurity	高安全级别登录接口。 **说明** CLINET_LoginEx2 仍然可以使用，但存在安全风险，所以强烈推荐使用最新接口 CLIENT_LoginWithHighLevelSecurity 登录设备。
CLIENT_Logout	登出接口。

2.1.2.3 流程说明

图2-2 登录业务流程



流程说明

步骤1　调用 CLIENT_Init 完成 SDK 初始化流程。
步骤2　调用 CLIENT_LoginWithHighLevelSecurity 登录设备。
步骤3　登录成功后，用户可以实现需要的业务功能。
步骤4　业务使用完后，调用 CLIENT_Logout 登出设备。
步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 释放 SDK 资源。

注意事项

- 登录句柄：登录成功时接口返回值非 0（即句柄可能小于 0，也属于登录成功）；同一设备登录多次，每次的登录句柄不一样。如果无特殊业务，建议只登录一次，登录的句柄可以重复用于其他各种业务。
- 登出：接口内部会释放登录会话中已打开的业务，但建议用户不要依赖登出接口的清理功能。例如打开预览后，在不需要使用预览时，用户应该调用结束预览的接口。
- 登录与登出配对使用，登录会消耗一定的内存和 socket 信息，在登出后释放资源。
- 登录失败：建议通过登录接口的 error 参数（登录错误码）初步排查。常见错误码如表 2-3 所示。
- 多设备登录：SDK 初始化后，可以登录多台设备，但是相应的登录句柄、登录信息需要调整。

表2-3 常见错误码

error 错误码	对应的含义
1	密码不正确。
2	用户名不存在。
3	登录超时。
4	账号已登录。
5	账号已被锁定。
6	账号被列为禁止名单。
7	资源不足，设备系统忙。
8	子连接失败。
9	主连接失败。
10	超过最大用户连接数。
11	缺少 avnetsdk 或 avnetsdk 的依赖库。
12	设备未插入 U 盘或 U 盘信息错误。
13	客户端 IP 地址没有登录权限。

2.1.2.4 示例代码

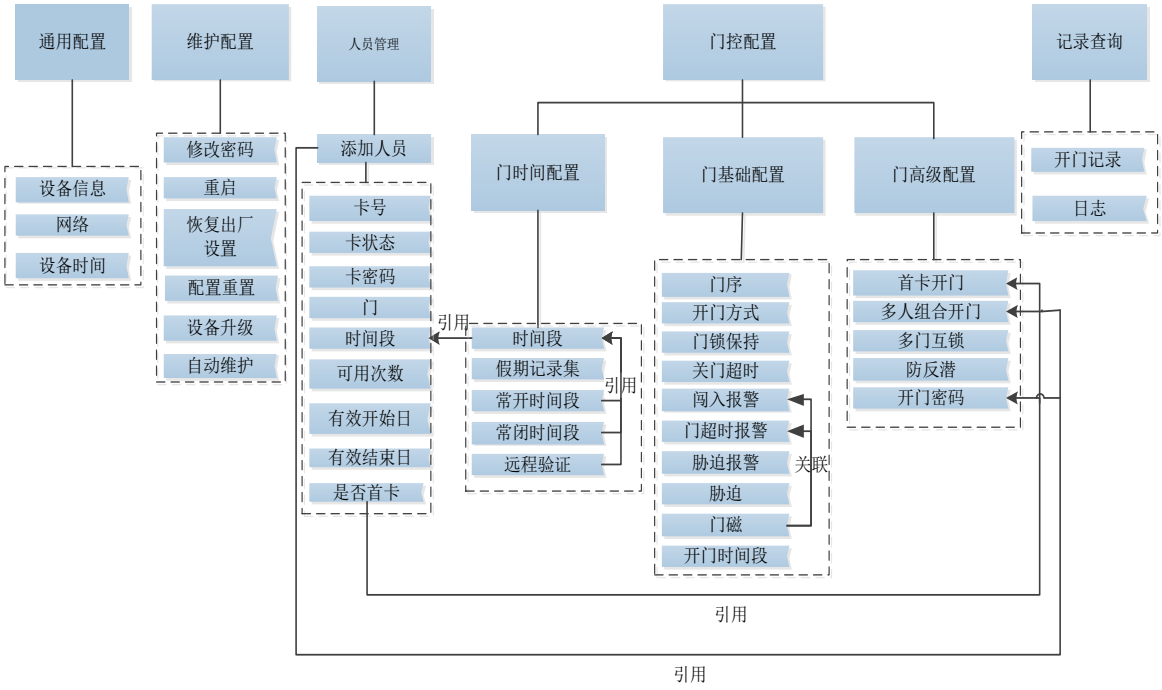
```
public class LoginModule {

    public static NetSDKLib netsdk            = NetSDKLib.NETSDK_INSTANCE;
    public static NetSDKLib configsdk    = NetSDKLib.CONFIG_INSTANCE;

    //SDK 初始化，SDK 清理省略

    // 设备信息
    public static NetSDKLib.NET_DEVICEINFO_Ex m_stDeviceInfo = new NetSDKLib.NET_DEVICEINFO_Ex();

    //登录句柄
    public static LLong m_hLoginHandle = new LLong(0);

    //登录设备
    public static boolean login(String m_strIp, int m_nPort, String m_strUser, String m_strPassword) {
    //入参
NET_IN_LOGIN_WITH_HIGHLEVEL_SECURITY pstInParam=
new NET_IN_LOGIN_WITH_HIGHLEVEL_SECURITY();
pstInParam. szIP= m_strIp;
pstInParam.nport= m_nPort;
pstInParam.szUserName= m_strUser;
pstInParam.szPassword= m_strPassword;
```

```
//出参
NET_OUT_LOGIN_WITH_HIGHLEVEL_SECURITY pstOutParam=
new NET_OUT_LOGIN_WITH_HIGHLEVEL_SECURITY();
    m_hLoginHandle = netsdk.CLIENT_LoginWithHighLevelSecurity(NET_IN_LOGIN_WITH_HIGHLEVEL_SECURITY pstInParam, NET_OUT_LOGIN_WITH_HIGHLEVEL_SECURITY pstOutParam);

if(m_hLoginHandle.longValue() == 0) {
            System.err.printf("Login Device[%s] Port[%d]Failed. %s\n", m_strIp, m_nPort, ToolKits.getErrorCodePrint());
        } else {
            System.out.println("Login Success ");
        }

        return m_hLoginHandle.longValue() == 0? false:true;
    }

    //登出设备
    public static boolean logout() {
        if(m_hLoginHandle.longValue() == 0) {
            return false;
        }

        boolean bRet = netsdk.CLIENT_Logout(m_hLoginHandle);
        if(bRet) {
            m_hLoginHandle.setValue(0);
        }

        return bRet;
    }
}
```

2.2 门禁控制器/指纹一体机（一代）

门禁控制器/指纹一体机（一代）功能调用关系

图2-3 功能调用关系



功能调用关系中引用和关联的含义如下：

- 引用：箭头终点指向的功能引用箭头起点指向的功能。
- 关联：箭头起始的功能是否能够正常使用，与箭头终点指向的功能配置相关。

2.2.1 门禁控制

2.2.1.1 简介

控制门禁的打开和关闭，获取门磁状态。不需要人员信息，直接远程控制门进行打开和关闭。

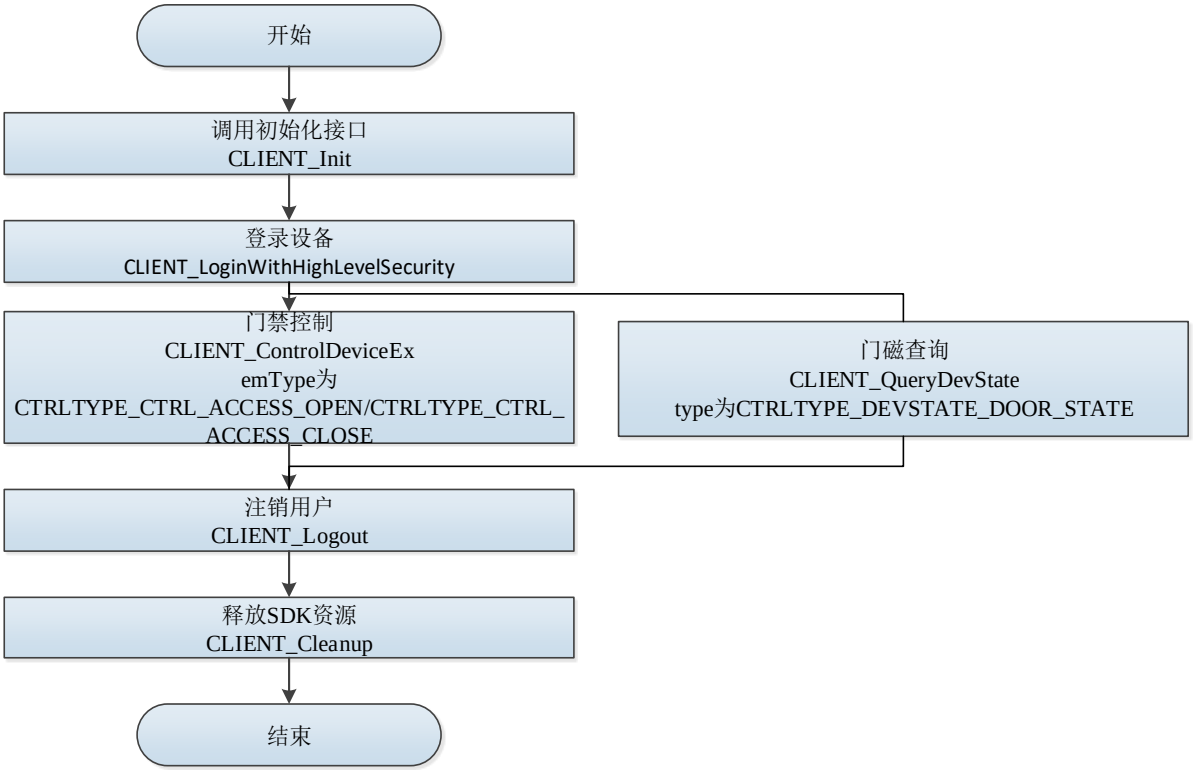
2.2.1.2 接口总览

表2-4 门禁控制接口说明

接口	说明
CLIENT_ControlDeviceEx	设备控制扩展接口。
CLIENT_QueryDevState	状态查询接口。

2.2.1.3 流程说明

图2-4 门禁控制业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_ControlDeviceEx 函数来控制门禁。

- 打开门禁：**emType** 值为 **CTRLTYPE _CTRL_ACCESS_OPEN**。
- 关闭门禁：**emType** 值为 **CTRLTYPE _CTRL_ACCESS_CLOSE**。

步骤4　业务实现，调用 CLIENT_QueryDevState 来实现门磁查询。

- Type：CTRLTYPE _DEVSTATE_DOOR_STATE
- pBuf：NET_DOOR_STATUS_INFO

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.1.4 示例代码

```
/**
 * 关门
 */
public void closeDoor() {
    final NetSDKLib.NET_CTRL_ACCESS_CLOSE close = new
NetSDKLib.NET_CTRL_ACCESS_CLOSE();
    close.nChannelID = 0; // 对应的门编号 - 如何开全部的门
    close.write();
```

```
        boolean result = netsdkApi.CLIENT_ControlDeviceEx(loginHandle,
                                            NetSDKLib.CtrlType.CTRLTYPE_CTRL_ACCESS_CLOSE,
                                            close.getPointer(),
                                            null,
                                            5000);
        close.read();
        if (!result) {
            System.err.println("close error: 0x" + Long.toHexString(netsdkApi.CLIENT_GetLastError()));
        }
    }

    /**
     * 查询门（开、关）状态
     */
    public void queryDoorStatus() {
        int cmd = NetSDKLib.NET_DEVSTATE_DOOR_STATE;
        NetSDKLib.NET_DOOR_STATUS_INFO doorStatus = new
    NetSDKLib.NET_DOOR_STATUS_INFO();
        IntByReference retLenByReference = new IntByReference(0);

        doorStatus.write();
        boolean bRet = netsdkApi.CLIENT_QueryDevState(loginHandle,
                                                      cmd,
                                                      doorStatus.getPointer(),
                                                      doorStatus.size(),
                                                      retLenByReference,
                                                      3000);
        doorStatus.read();
        if (!bRet) {
            System.err.println("Failed to queryDoorStatus. Error Code 0x"
                    + Integer.toHexString(netsdkApi.CLIENT_GetLastError()));
            return;
        }

        String stateType[] = {"未知", "门打开", "门关闭", "门异常打开"};
        System.out.println("doorStatus -> Channel: " + doorStatus.nChannel
                    + " type: " + stateType[doorStatus.emStateType]);
    }
```

2.2.2 报警事件

2.2.2.1 简介

事件获取，即用户调用 SDK 接口，SDK 主动连接设备，并向设备订阅事件，包括开门事件、报警事件，设备发生事件立即发送给 SDK。若要停止接收设备事件，则需要停止订阅。

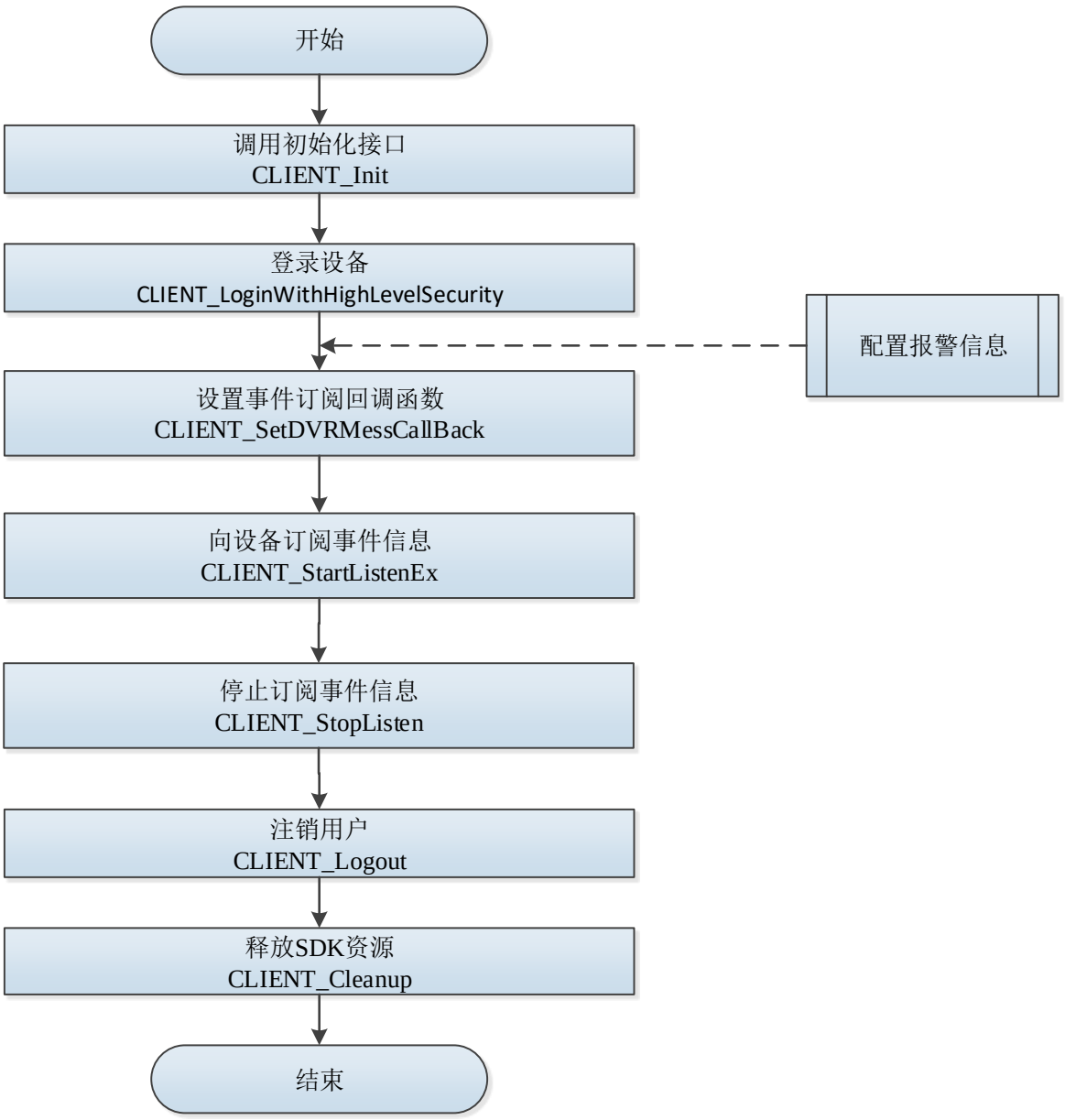
2.2.2.2 接口总览

表2-5 报警事件接口说明

接口	说明
CLIENT_StartListenEx	向设备订阅报警。
CLIENT_SetDVRMessCallBack	设置设备消息回调函数，用来得到设备当前状态信息，与调用顺序无关，SDK 默认不回调，此回调函数必须先调用报警消息订阅接口 CLIENT_StartListen 或 CLIENT_StartListenEx 才有效。
CLIENT_StopListen	停止订阅。

2.2.2.3 流程说明

图2-5 报警事件业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。
步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。
步骤3　进行报警布防配置（如果报警布防已经配置完成，则可省略）。
步骤4　设置报警回调函数 CLIENT_SetDVRMessCallBack。
步骤5　调用 CLIENT_StartListenEx 函数向设备订阅报警信息。
步骤6　整个报警上传过程结束后还需要停止订阅报警接口 CLIENT_StopListen。
步骤7　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。
步骤8　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.2.4 示例代码

```
/**
 * 订阅报警信息
 */
public void startListen() {
    // 设置报警回调函数
    netsdk.CLIENT_SetDVRMessCallBack(fAlarmDataCB.getCallBack(), null);

    // 订阅报警
    boolean bRet = netsdk.CLIENT_StartListenEx(m_hLoginHandle);
    if (!bRet) {
        System.err.println("订阅报警失败! LastError = 0x%x\n" + netsdk.CLIENT_GetLastError());
    }
    else {
        System.out.println("订阅报警成功.");
    }
}

/**
 * 取消订阅报警信息
 */
public void stopListen() {
    // 停止订阅报警
    boolean bRet = netsdk.CLIENT_StopListen(m_hLoginHandle);
    if (bRet) {
        System.out.println("取消订阅报警信息.");
    }
}
/**
 * 报警信息回调函数原形,建议写成单例模式
 */
private static class fAlarmDataCB implements NetSDKLib.fMessCallBack{
    private fAlarmDataCB(){}
```

```
    private static class fAlarmDataCBHolder {
        private static fAlarmDataCB callback = new fAlarmDataCB();
    }

    public static fAlarmDataCB getCallBack() {
        return fAlarmDataCB.fAlarmDataCBHolder.callback;
    }

    public boolean invoke(int lCommand, NetSDKLib.LLong lLoginID, Pointer pStuEvent, int dwBufLen,
String strDeviceIP, NativeLong nDevicePort, Pointer dwUser){
        switch (lCommand)
        {
            case NetSDKLib.NET_ALARM_TALKING_INVITE:    {  // 设备请求对方发起对讲事件(对
应结构体  ALARM_TALKING_INVITE_INFO)
                System.out.println("设备请求对方发起对讲事件");
                NetSDKLib.ALARM_TALKING_INVITE_INFO msg = new
NetSDKLib.ALARM_TALKING_INVITE_INFO();
                ToolKits.GetPointerData(pStuEvent, msg);
                System.out.println("emCaller :" + msg.emCaller);
                System.out.println("szCallID :" + new String(msg.szCallID).trim());
                System.out.println("nLevel :" + msg.nLevel);

                /**
                 RealUTC 是否有效，bRealUTC 为 TRUE 时，用 RealUTC，否则用 stuTime 字段
                 */
                int bRealUTC
                        = msg.bRealUTC;
                System.out.println("bRealUTC :" + bRealUTC);
                if(bRealUTC==1){
                    System.out.println("RealUTC :" + msg.RealUTC.toStringTime());
                }else {
                    System.out.println("stuTime :" + msg.stuTime.toStringTime());

                }

                NetSDKLib.TALKINGINVITE_REMOTEDEVICEINFO stuRemoteDeviceInfo
                        = msg.stuRemoteDeviceInfo;
                int emProtocol = stuRemoteDeviceInfo.emProtocol;

                System.out.println("emProtocol:" + emProtocol);


                try {
                    System.out.println("szIP:" +     new String(stuRemoteDeviceInfo.szIP,encode));

                    System.out.println("nPort:" +    stuRemoteDeviceInfo.nPort);
```

```
                    System.out.println("szUser:" +     new
String(stuRemoteDeviceInfo.szUser,encode));

                    System.out.println("szPassword:" +     new
String(stuRemoteDeviceInfo.szPassword,encode));

                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }


                break;
            }
            case NetSDKLib.NET_ALARM_TALKING_HANGUP   :{ // 设备主动挂断对讲事件(对应结
构体 ALARM_TALKING_HANGUP_INFO)
                System.out.println("设备主动挂断对讲事件");
                ALARM_TALKING_HANGUP_INFO msg=new ALARM_TALKING_HANGUP_INFO();
                ToolKits.GetPointerData(pStuEvent, msg);

                try {
                    System.out.println("szRoomNo :" + new String(msg.szRoomNo,encode));
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                /**
                 RealUTC 是否有效，bRealUTC 为 TRUE 时，用 RealUTC，否则用 stuTime 字段
                 */
                int bRealUTC
                        = msg.bRealUTC;
                System.out.println("bRealUTC :" + bRealUTC);
                if(bRealUTC==1){
                    System.out.println("RealUTC :" + msg.RealUTC.toStringTime());

                }else {
                    System.out.println("stuTime :" + msg.stuTime.toStringTime());

                }


                byte[] szCaller = msg.szCaller;
                try {
                    System.out.println("szCaller :" + new String(szCaller,encode));
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                break;
```

```
            }

            default:
                System.out.println("事件："+lCommand);

                break;
        }
        return true;
    }
}
```

2.2.3 智能报警带图事件

2.2.3.1 简介

智能订阅，即智能设备对实时码流进行分析，当检测到预先设定好的事件时，将事件发送给用户。智能事件有交通违章、停车场有无车位等事件。

智能订阅实现方式为 SDK 主动连接设备，并向设备订阅智能事件功能，设备检测到智能事件立即发送给 SDK。

当前支持的智能订阅事件参见 NetSDKLib.java 中以 EVENT_IVS_开头的常量，包含了常规的交通占道、车辆违规等事件。

2.2.3.2 接口总览

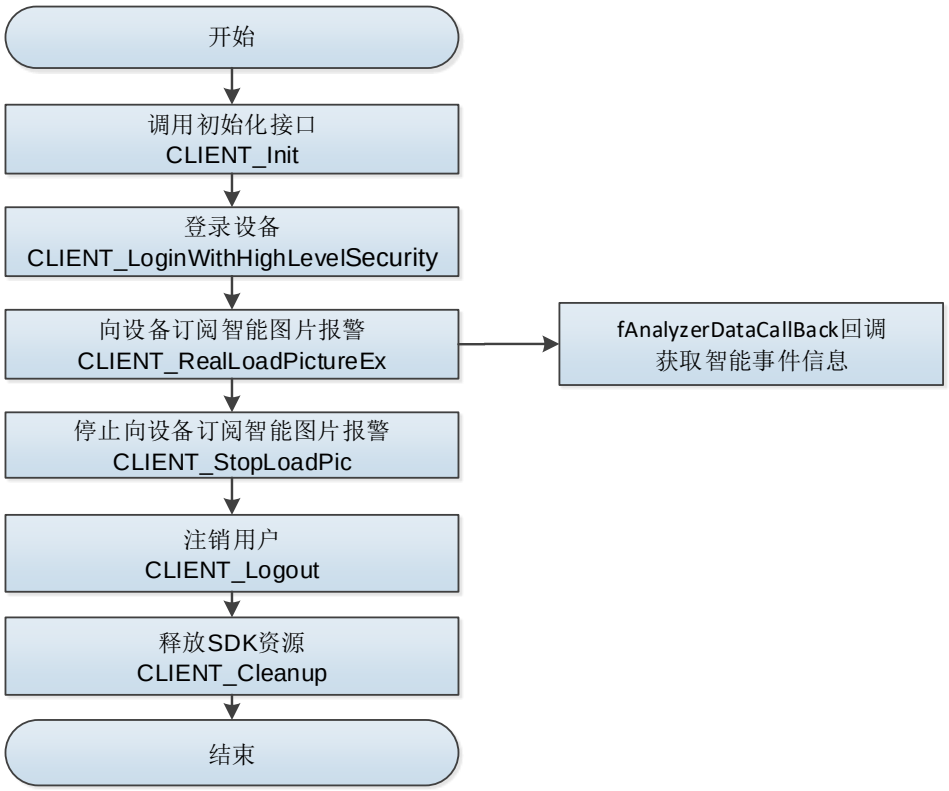
表2-6 智能交通时间上报的接口信息

接口	说明
CLIENT_RealLoadPictureEx	订阅智能事件
CLIENT_StopLoadPic	取消订阅智能事件
fAnalyzerDataCallBack	用于回调获取智能事件的信息

2.2.3.3 流程说明

智能订阅事件上报流程如图 2-6 所示。

图2-6 智能订阅事件上报业务流程



流程说明

步骤1　调用 CLIENT_Init 函数完成 SDK 初始化流程。
步骤2　初始化成功后，调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。
步骤3　调用 CLIENT_RealLoadPictureEx 函数向设备订阅智能事件。
步骤4　订阅成功后设备上报的智能交通事件通过 fAnalyzerDataCallBack 回调函数获取智能事件并通知用户。
步骤5　智能事件上报功能使用完毕后，调用 CLIENT_StopLoadPic 函数停止订阅智能事件。
步骤6　业务使用完后，调用 CLIENT_Logout 函数登出设备。
步骤7　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

注意事项

- 订阅事件类型：如果需要同时上报不同智能事件时，支持订阅所有智能事件（EVENT_IVS_ALL）；也支持订阅单个智能事件。
- 设置是否接收图片：由于某些设备所在网络环境是 3G 或 4G 网络，当 SDK 连接设备时，如不需要接收图片可以把 CLIENT_ RealLoadPictureEx 接口中 bNeedPicFile 参数设置为 False，只接收智能交通事件信息，不带图片。
- 通过通道号传-1 进行全通道订阅。部分智能交通类产品不支持全通道订阅。若传-1 订阅失败，请尝试单通道订阅。

2.2.3.4 示例代码

```
//本实例将用门禁事件举例
//省略 SDK 初始化以及门禁设备登录
```

```
    // 登录句柄
    private NetSDKLib.LLong loginHandle = new NetSDKLib.LLong(0);
    // 智能事件订阅句柄
    private NetSDKLib.LLong m_attachHandle = new NetSDKLib.LLong(0);
    // 监听
    /**
     * 订阅智能任务
     */
    public void AttachEventRealLoadPic() {
        // 先退订，设备不会对重复订阅作校验，重复订阅后会有重复的事件返回
        this.DetachEventRealLoadPic();
        // 需要图片
        int bNeedPicture = 1;
        //EVENT_IVS_ALL 表示订阅所有智能事件
        m_attachHandle = netsdk.CLIENT_RealLoadPictureEx(loginHandle, channelId,
EVENT_IVS_ALL, bNeedPicture, AnalyzerDataCB.getInstance(), null, null);
        /**
         * // EVENT_IVS_WORKCLOTHES_DETECT 安全帽检测事件
         * // EVENT_IVS_SMOKING_DETECT 吸烟检测事件
         **/
        if (m_attachHandle.longValue() != 0) {
            System.out.printf("Chn[%d] CLIENT_RealLoadPictureEx Success\n", channelId);
        } else {
            System.out.printf("Ch[%d] CLIENT_RealLoadPictureEx Failed!LastError = %s\n",
channelId,
                    ToolKits.getErrorCode());
        }
    }


    /**
     * 报警事件（智能）回调
     */
    private static class AnalyzerDataCB implements NetSDKLib.fAnalyzerDataCallBack {
        private final File picturePath;
        private static AnalyzerDataCB instance;

        private AnalyzerDataCB() {
            picturePath = new File("./AnalyzerPicture/");
            if (!picturePath.exists()) {
```

```
                picturePath.mkdirs();
            }
        }

        public static AnalyzerDataCB getInstance() {
            if (instance == null) {
                synchronized (AnalyzerDataCB.class) {
                    if (instance == null) {
                        instance = new AnalyzerDataCB();
                    }
                }
            }
            return instance;
        }

        @Override
        public int invoke(NetSDKLib.LLong lAnalyzerHandle, int dwAlarmType, Pointer pAlarmInfo, Pointer pBuffer, int dwBufSize,
                          Pointer dwUser, int nSequence, Pointer reserved) {
            if (lAnalyzerHandle == null || lAnalyzerHandle.longValue() == 0) {
                return -1;
            }

            switch (dwAlarmType) {
                case EVENT_IVS_WORKCLOTHES_DETECT: // 安全帽检测事件
                {
                    NetSDKLib.DEV_EVENT_WORKCLOTHES_DETECT_INFO msg = new NetSDKLib.DEV_EVENT_WORKCLOTHES_DETECT_INFO();
                    ToolKits.GetPointerData(pAlarmInfo, msg);
                    if (msg.stuSceneImage != null && msg.stuSceneImage.nLength > 0) {
                        String bigPicture = picturePath + "\\" + System.currentTimeMillis() + ".jpg";
                        ToolKits.savePicture(pBuffer, msg.stuSceneImage.nOffSet, msg.stuSceneImage.nLength, bigPicture);
                        if (msg.stuHumanImage != null && msg.stuHumanImage.nLength > 0) {
                            String smallPicture = picturePath + "\\" + System.currentTimeMillis() + "small.jpg";
                            ToolKits.savePicture(pBuffer, msg.stuHumanImage.nOffSet, msg.stuHumanImage.nLength, smallPicture);
                        }
```

```
                }
                System.out.println(" 安全帽检测事件(UTC)：" + msg.UTC + " 通道号:" + msg.nChannelID);
                break;
            }
            case NetSDKLib.EVENT_IVS_SMOKING_DETECT：{    // 吸烟检测事件(对应 DEV_EVENT_SMOKING_DETECT_INFO)
                System.out.printf("吸烟检测事件");
                DEV_EVENT_SMOKING_DETECT_INFO msg = new DEV_EVENT_SMOKING_DETECT_INFO();
                ToolKits.GetPointerData(pAlarmInfo, msg);
                String Picture = picturePath + "\\" + "smoking_" + System.currentTimeMillis() + ".jpg";

                if (dwBufSize > 0) {
                    ToolKits.savePicture(pBuffer, msg.stuImageInfo[0].nOffset, msg.stuImageInfo[0].nLength, Picture);
                }
                System.out.println("吸烟检测事件 时间(UTC)：" + msg.UTC + " 开始时间:" + msg.stuObject.stuStartTime + " 结束时间:"
                        + msg.stuObject.stuEndTime);
                break;
            }
            case EVENT_IVS_GRANARY_TRANS_ACTION_DETECTION:{ //粮面异动检测事件上报(对应 NET_DEV_EVENT_GRANARY_TRANS_ACTION_DETECTION_INFO)
                System.out.println("粮面异动检测事件上报");
                NET_DEV_EVENT_GRANARY_TRANS_ACTION_DETECTION_INFO msg = new NET_DEV_EVENT_GRANARY_TRANS_ACTION_DETECTION_INFO();
                ToolKits.GetPointerData(pAlarmInfo, msg);
                System.out.println("粮面异动检测事件上报 时间(stuUTC)：" + msg.stuUTC);
                System.out.println("物体列表消息实际数量（nObjectsCount）：" + msg.nObjectsCount);
                break;
            }
            case EVENT_IVS_REGION_PROPORTION_DETECTION:{ // 区域占比检测事件(对应 NET_DEV_EVENT_REGION_PROPORTION_DETECTION_INFO)
                System.out.println("区域占比检测事件");
                NET_DEV_EVENT_REGION_PROPORTION_DETECTION_INFO msg = new NET_DEV_EVENT_REGION_PROPORTION_DETECTION_INFO();
                ToolKits.GetPointerData(pAlarmInfo, msg);
                System.out.println("区域占比检测事件上报 时间(stuUTC)：" +
```

```
msg.stuUTC);
                        System.out.println("物体列表消息实际数量（nObjectsCount）：" +
msg.nObjectsCount);
                        break;
                    }
                    default:
                        System.out.println("其他事件----------"+ dwAlarmType);
                        break;
                }
                return 0;
            }
        }
        /**
         * 停止侦听智能事件
         */
        public void DetachEventRealLoadPic() {
            if (m_attachHandle.longValue() != 0) {
                netsdk.CLIENT_StopLoadPic(m_attachHandle);
                System.out.println("CLIENT_StopLoadPic Success");
            }
        }
```

2.2.4 设备信息查看

2.2.4.1 能力集查询

2.2.4.1.1 简介

设备信息查看，即用户通过 SDK 下发命令给门禁设备，来获取设备的能力集。

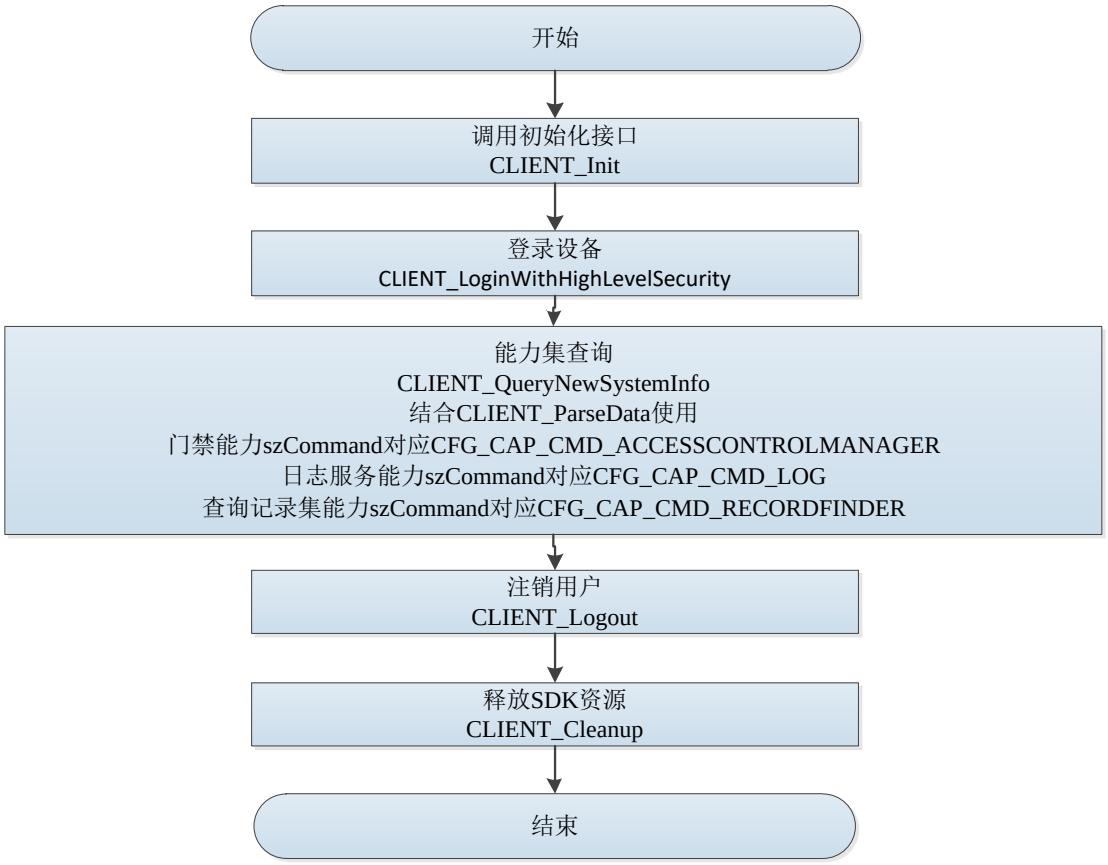
2.2.4.1.2 接口总览

表2-7 能力集查询接口说明

接口	说明
CLIENT_QueryNewSystemInfo	查询系统能力信息（日志、记录集、门控能力等）。
CLIENT_ParseData	解析查询到的配置信息。

2.2.4.1.3 流程说明

图2-7 设备信息查看流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_QueryNewSystemInfo 函数，结合 CLIENT_ParseData，来查询门禁能力集。

表2-8 szCommand 取值对应含义和结构体

szCommand	对应含义	szOutBuffer
CFG_CAP_CMD_ACCESSCONTROLMANAGER	门禁能力	CFG_CAP_ACCESSCONTROL
CFG_CAP_CMD_LOG	获取日志服务能力	CFG_CAP_LOG
CFG_CAP_CMD_RECORDFINDER	获取查询记录集能力	CFG_CAP_RECORDFINDER_INFO

步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.4.1.4 示例代码

```
/**
 * 优化 CLIENT_QueryNewSystemInfo 接口获取视频分析能力集
 * 1. 结构体 CFG_CAP_ANALYSE_INFO 的字节长度 54874632,结构体构建对象和解析时非常耗时,采取计算字节和偏移方式，获取对应字段信息
```

```
 */
public void queryNewSystemInfoOpt() {
    //结构体 CFG_CAP_ANALYSE_INFO 的字节长度 54874632,结构体构建对象和解析时非常耗时，采取计算字节和偏移方式，获取对应字段信息
    //long len = new CFG_CAP_ANALYSE_INFO().size(); len = 54874632
    byte[] data = new byte[54874632];
    IntByReference error = new IntByReference(0);
    boolean result = netsdk.CLIENT_QueryNewSystemInfo(m_hLoginHandle,
EM_NEW_QUERY_SYSTEM_INFO.CFG_CAP_CMD_VIDEOANALYSE.getValue(), 0, data,
data.length, error, 5000);
    if (!result) {
        System.out.println("query system info failed.error is" + ENUMERROR.getErrorMessage());
    }
    //解析能力信息
    Pointer pointer = new Memory(data.length);
    result =
configsdk.CLIENT_ParseData(EM_NEW_QUERY_SYSTEM_INFO.CFG_CAP_CMD_VIDEOANALYSE
.getValue(), data, pointer, data.length, null);
    if (!result) {
        System.out.println("parse system info failed.error is " + ENUMERROR.getErrorMessage());
    }
    CFG_CAP_ANALYSE_INFO_OPT info = new CFG_CAP_ANALYSE_INFO_OPT();
    //结构体 CFG_CAP_ANALYSE_INFO 字段 nSupportedData 之前字节偏移 0,到字段 szSceneName 字节偏移 4100
    info.getPointer().write(0, pointer.getByteArray(0, 4100), 0, info.size());
    info.read();
    System.out.println("支持场景个数:"+info.nSupportedSceneNum);
    System.out.println("支持的场景列表:");// 枚举，参考@{ @link EM_SCENE_TYPE}
    MaxNameByteArrInfo[] szSceneName = info.szSceneName;//判断是否具有枚举的值 FaceAnalysis 人脸分析
    String value = EM_SCENE_TYPE.getNoteByValue(27);//value = FaceAnalysis 人脸分析
    for (int i = 0; i < info.nSupportedSceneNum; i++) {
        //System.out.println(new String(szSceneName[i].name).trim());
        if(value.trim().equals(new String(szSceneName[i].name).trim())) {
            System.out.println("IPC 有目标识别能");
        }
    }
}
```

2.2.4.2 设备版本、MAC 查看

2.2.4.2.1 简介

设备版本、MAC 查看，即用户通过 SDK 下发命令给门禁设备，来获取设备的序列号、版本号、Mac 地址等内容。

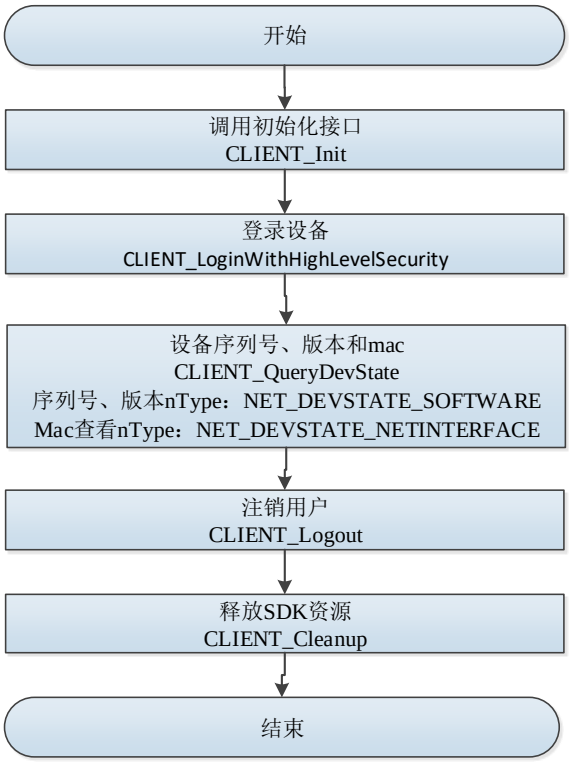
2.2.4.2.2 接口总览

表2-9 设备版本和 MAC 查看接口说明

接口	说明
CLIENT_QueryDevState	查询设备状态（查询序列号、软件版本、编译时间、Mac 地址）。

2.2.4.2.3 流程说明

图2-8 设备信息查看业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。
步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。
步骤3　调用 CLIENT_QueryDevState 函数，来查询门禁设备的序列号、版本和 mac 信息。
步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。
步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.4.2.4 示例代码

```
/**
 * 获取设备软件版本
 *
 */
public void QueryDevDeviceVersionStateTest() {
    NetSDKLib.NETDEV_VERSION_INFO info = new NetSDKLib.NETDEV_VERSION_INFO();
    info.write();
    boolean bRet = netSdk.CLIENT_QueryDevState(loginHandle, NET_DEVSTATE_SOFTWARE,
info.getPointer(), info.size(), new IntByReference(0), 3000);
```

```
        if (!bRet) {
            System.err.println("QueryDevState DEV STATE of SOFTWARE failed: " +
ToolKits.getErrorCode());
            return;
        }
        info.read();
        System.out.println("QueryDevState DEV STATE of SOFTWARE succeed");

        System.out.println("szSoftWareVersion 软件版本: " + new String(info.szSoftWareVersion).trim());
        System.out.println("szDevSerialNo 序列号: " + new String(info.szDevSerialNo).trim());
        int buildData = info.dwSoftwareBuildDate;
        int day = buildData & 0xff;
        buildData >>= 8;
        int month = buildData & 0xff;
        int year = buildData >> 8;
        System.out.println("BuildData 编译日期: " + year + "-" + month + "-" + day);
}

/**
 * 获取所有移动网络接口
 */
public void getNetAppMobileInterface() {
        // 入参
        NET_IN_NETAPP_GET_MOBILE_INTERFACE pstuIn = new
NET_IN_NETAPP_GET_MOBILE_INTERFACE();
        // 出参
        NET_OUT_NETAPP_GET_MOBILE_INTERFACE pstuOut = new
NET_OUT_NETAPP_GET_MOBILE_INTERFACE();

        pstuIn.write();
        pstuOut.write();
        boolean bRet = netsdk.CLIENT_RPC_NetApp(m_hLoginHandle,
EM_RPC_NETAPP_TYPE.EM_PRC_NETAPP_TYPE_GET_MOBILE_INTERFACE.getId(),
pstuIn.getPointer(), pstuOut.getPointer(), 3000);
        if(bRet) {
            pstuOut.read();
            System.out.println("网络接口有效个数:"+pstuOut.nInterfaceNum);
            System.out.println("移动网络接口信息");
            NETDEV_NETINTERFACE_INFO[] stuInterface = pstuOut.stuInterface;
            for (int i = 0; i < pstuOut.nInterfaceNum; i++) {
                int a = i+1;
                System.out.println("-----第"+a+"个----");
                System.out.println("是否有效:"+stuInterface[i].bValid);
                try {
                    System.out.println("网口名称:"+new String(stuInterface[i].szName,encode));
                    System.out.println("实际 3G 支持的网络模式个
数:"+stuInterface[i].nSupportedModeNum);
```

```
                    SupportedModeByteArr[] szSupportedModes= stuInterface[i].szSupportedModes;
                    String prt = "";
                    for (int j = 0; j < stuInterface[i].nSupportedModeNum; j++) {
                        if(j == (stuInterface[i].nSupportedModeNum-1) ) {
                            prt += new String
(szSupportedModes[j].supportedModeByteArr,encode).trim();
                        }else {
                            prt += new String
(szSupportedModes[j].supportedModeByteArr,encode).trim() +"、";
                        }
                    }
                    System.out.println("3G 支持的网络模式:"+prt);

                    System.out.println("国际移动用户识别码:"+new String(stuInterface[i].szIMEI,encode));
                    System.out.println("集成电路卡识别码即 SIM 卡卡号:"+new
String(stuInterface[i].szICCID,encode));
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }

            }

        } else {
            System.err.println("getNetAppMobileInterface Failed!" + ToolKits.getErrorCode());
        }
}
```

2.2.5 网络配置

2.2.5.1 IP 配置

2.2.5.1.1 简介

IP 配置，即用户通过调用 SDK 接口，对设备的 IP 等信息进行获取和配置，包含 IP 地址、子网掩码、默认网关等信息。

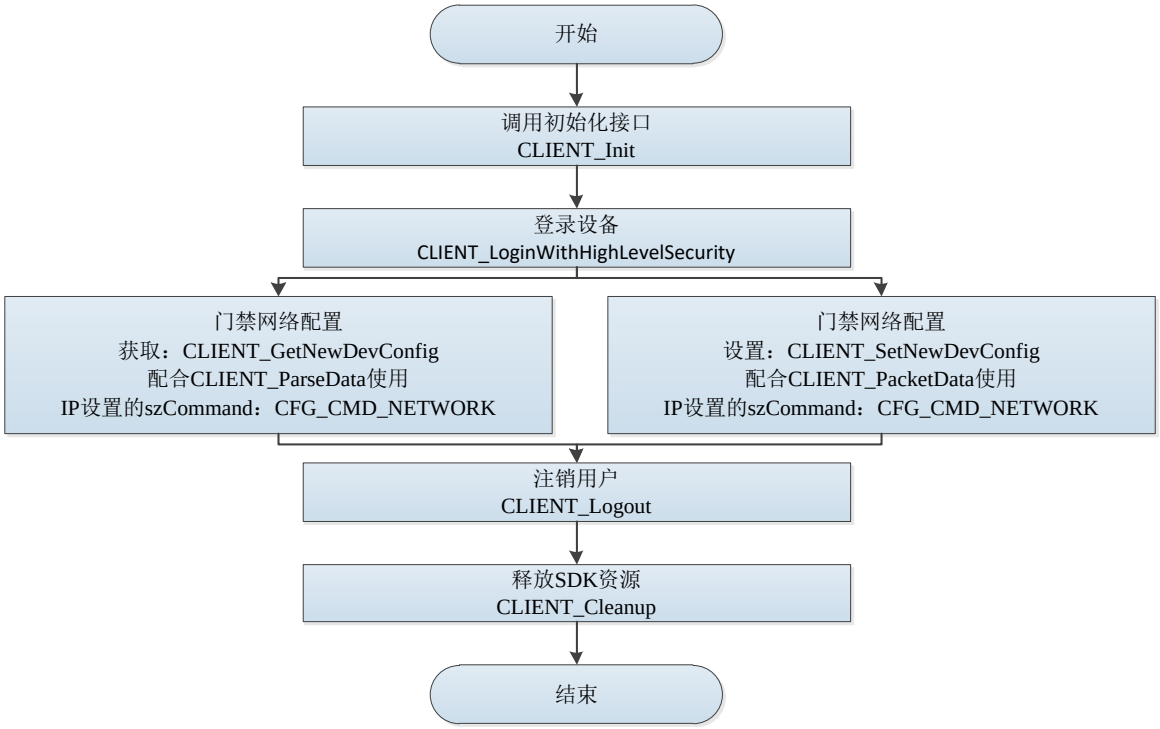
2.2.5.1.2 接口总览

表2-10 IP 配置接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息
CLIENT_ParseData	解析查询到的配置信息
CLIENT_SetNewDevConfig	设置配置信息
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式

2.2.5.1.3 流程说明

图2-9 IP 配置业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁 IP 网络配置。

- szCommand：CFG_CMD_NETWORK。
- pBuf：CFG_NETWORK_INFO。

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁 IP 网络配置。

- szCommand：CFG_CMD_NETWORK。
- pBuf：CFG_NETWORK_INFO。

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.5.1.4 示例代码

```
// 网络配置
public void getSetNetWorkInfo() {
    String command = NetSDKLib.CFG_CMD_NETWORK;
    int channel = 0;
    CFG_NETWORK_INFO network = new CFG_NETWORK_INFO();

    // 获取
    if (ToolKits.GetDevConfig(m_hLoginHandle, channel, command, network)) {
        System.out.println("主机名称 :" + new String(network.szHostName).trim());
```

```
            for (int i = 0; i < network.nInterfaceNum; i++) {
                System.out.println("网络接口名称 : " + new String(network.stuInterfaces[i].szName).trim());
                System.out.println("ip 地址 : " + new String(network.stuInterfaces[i].szIP).trim());
                System.out.println("子网掩码 : " + new
String(network.stuInterfaces[i].szSubnetMask).trim());
                System.out.println("默认网关 : " + new
String(network.stuInterfaces[i].szDefGateway).trim());
                System.out.println(
                        "DNS 服务器地址 : " + new
String(network.stuInterfaces[i].szDnsServersArr[0].szDnsServers).trim()
                                + "\n" + new
String(network.stuInterfaces[i].szDnsServersArr[1].szDnsServers).trim());
                System.out.println("MAC 地址 : " + new
String(network.stuInterfaces[i].szMacAddress).trim());
            }

            if (ToolKits.SetDevConfig(m_hLoginHandle, channel, command, network)) {
                System.out.println("Set NETWORK Succeed!");
            } else {
                System.err.println("Set NETWORK Failed!" + netsdk.CLIENT_GetLastError());
            }

        } else {
            System.err.println("Get NETWORK Failed!" + netsdk.CLIENT_GetLastError());
        }
}
```

2.2.5.2 主动注册配置

2.2.5.2.1 简介

主动注册配置，即用户通过调用 SDK 接口，对设备的主动注册信息进行配置，包括主动注册使能、设备 ID、服务器信息等。

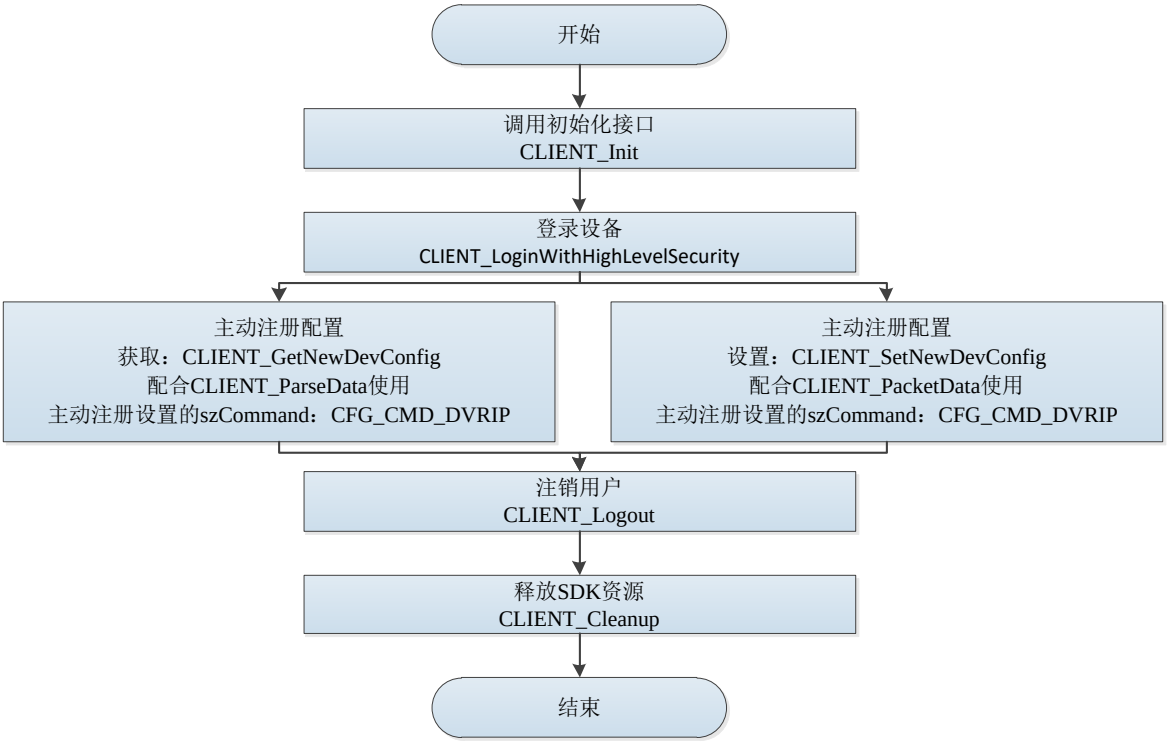
2.2.5.2.2 接口总览

表2-11 主动注册配置接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.5.2.3 流程说明

图2-10 主动注册配置业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　主动注册配置。

- 调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询主动注册配置。
  - ◇ szCommand：CFG_CMD_DVRIP。
  - ◇ pBuf：CFG_DVRIP_INFO。
- 调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置主动注册配置。
  - ◇ szCommand：CFG_CMD_DVRIP。
  - ◇ pBuf：CFG_DVRIP_INFO。

步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.5.2.4 示例代码

```
// 网络协议配置
public void getSetDVRIPInfo() {
    String command = NetSDKLib.CFG_CMD_DVRIP;
    int channel = 0;
    CFG_DVRIP_INFO dvrIp = new CFG_DVRIP_INFO();

    // 获取
    if (ToolKits.GetDevConfig(m_hLoginHandle, channel, command, dvrIp)) {
```

```
            System.out.println("TCP 服务端口 : " + dvrIp.nTcpPort);
            System.out.println("SSL 服务端口 :" + dvrIp.nSSLPort);
            System.out.println("UDP 服务端口 :" + dvrIp.nUDPPort);
            System.out.println("组播端口号 :" + dvrIp.nMCASTPort);

            // 设置，在获取的基础上设置
            if (ToolKits.SetDevConfig(m_hLoginHandle, channel, command, dvrIp)) {
                System.out.println("Set DVRIP Succeed!");
            } else {
                System.err.println("Set DVRIP Failed!" + netsdk.CLIENT_GetLastError());
            }
        } else {
            System.err.println("Get DVRIP Failed!" + netsdk.CLIENT_GetLastError());
        }
    }
```

2.2.6 设备时间设置

2.2.6.1 设备时间设置

2.2.6.1.1 简介

设备时间设置，即用户通过调用 SDK 接口，对设备时间进行获取和设置的操作。

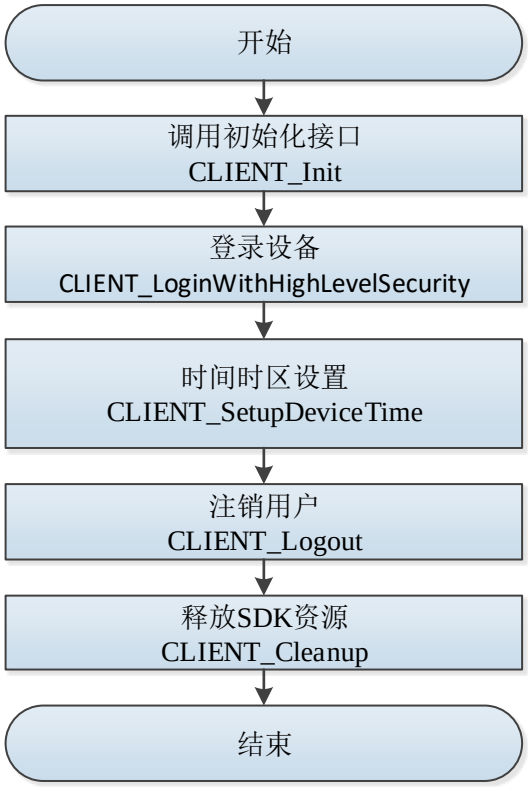
2.2.6.1.2 接口总览

表2-12 时间设置接口说明

接口	说明
CLIENT_SetupDeviceTime	设置设备当前时间。

2.2.6.1.3 流程说明

图2-11 时间设置业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。
步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。
步骤3　调用 CLIENT_SetupDeviceTime 函数来设置门禁时间信息。
步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。
步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.6.1.4 示例代码

```
/**
 * 时间同步
 */
public void setupDeviceTime() {
    SimpleDateFormat simpleDate = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String date = simpleDate.format(new java.util.Date());

    String[] dateTime = date.split(" ");
    String[] mDate1 = dateTime[0].split("-");
    String[] mDate2 = dateTime[1].split(":");

    NetSDKLib.NET_TIME pDeviceTime = new NetSDKLib.NET_TIME();
    pDeviceTime.dwYear = Integer.parseInt(mDate1[0]);
    pDeviceTime.dwMonth = Integer.parseInt(mDate1[1]);
    pDeviceTime.dwDay = Integer.parseInt(mDate1[2]);
    pDeviceTime.dwHour = Integer.parseInt(mDate2[0]);
```

```
        pDeviceTime.dwMinute = Integer.parseInt(mDate2[1]);
        pDeviceTime.dwSecond = Integer.parseInt(mDate2[2]);

        if(netsdkApi.CLIENT_SetupDeviceTime(loginHandle, pDeviceTime)) {
            System.out.println("同步时间成功!");
        } else {
            System.out.println("同步时间失败" + ToolKits.getErrorCode());
        }
    }
```

2.2.6.2 NTP 服务器和时区配置

2.2.6.2.1 简介

NTP 服务器和时区配置，即用户通过调用 SDK 接口，对 NTP 服务器和时区进行获取和配置。

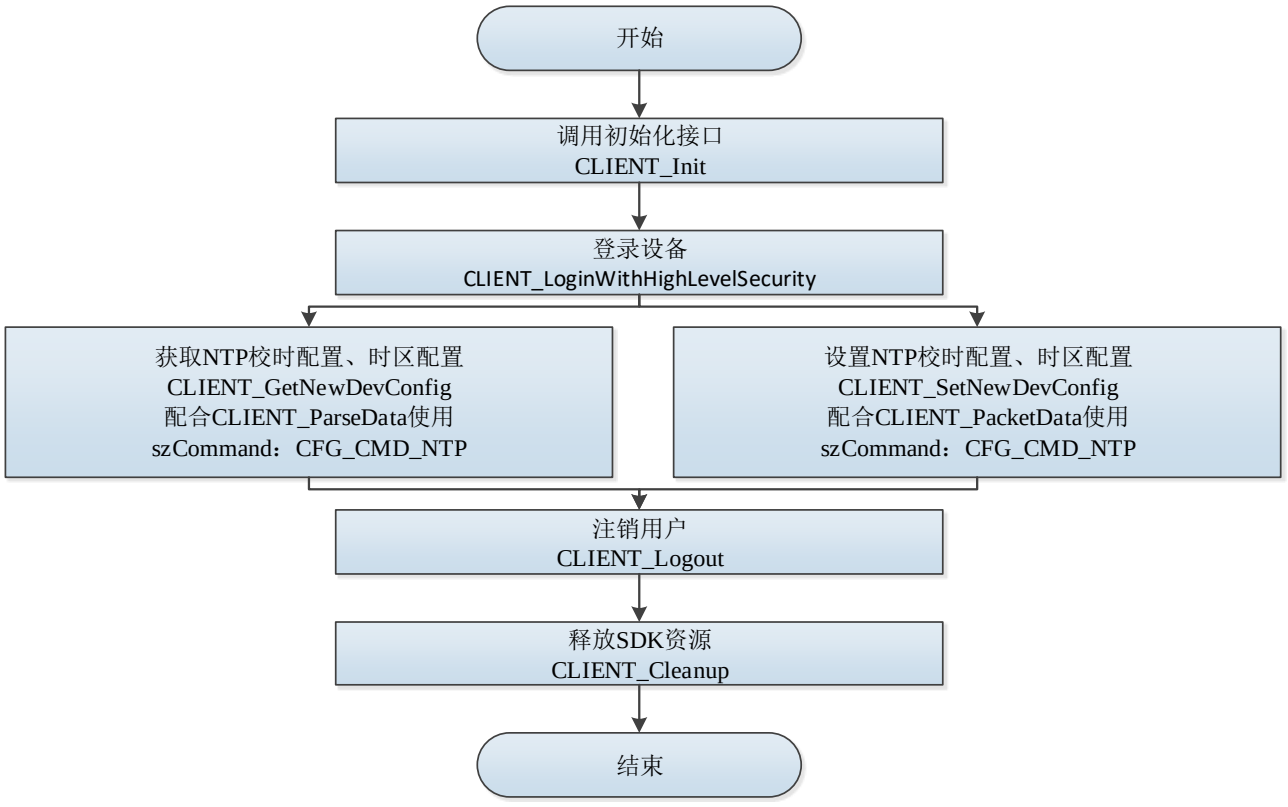
2.2.6.2.2 接口总览

表2-13 NTP 服务器和时区接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.6.2.3 流程说明

图2-12 NTP 校时业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁 NTP 校时配置、时区配置。

- szCommand：CFG_CMD_NTP。
- pBuf：CFG_NTP_INFO。

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁 NTP 校时配置、时区配置。

- szCommand：CFG_CMD_NTP。
- pBuf：CFG_NTP_INFO。

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.6.2.4 示例代码

```
// 时区同步
public void SetNTP() {
    NetSDKLib.CFG_NTP_INFO ntpInfo = new NetSDKLib.CFG_NTP_INFO();
    if (ToolKits.GetDevConfig(loginHandle, -1, NetSDKLib.CFG_CMD_NTP, ntpInfo)) {
        System.out.println("bEnable : " + ntpInfo.bEnable);
        System.out.println("emTimeZoneType : " + ntpInfo.emTimeZoneType);
        System.out.println("szTimeZoneDesc : " + new String(ntpInfo.szTimeZoneDesc).trim());
    }

    // 设置
    ntpInfo.bEnable = 1;
    ntpInfo.emTimeZoneType = NetSDKLib.EM_CFG_TIME_ZONE_TYPE.EM_CFG_TIME_ZONE_0;
    if (ToolKits.SetDevConfig(loginHandle, -1, NetSDKLib.CFG_CMD_NTP, ntpInfo)) {
        System.out.println("SetNTP Succeed!");
    }
}
```

2.2.7 人员管理

2.2.7.1.1 简介

人员信息，即用户通过调用 SDK，可以对门禁设备的人员信息字段（包含：编号、姓名、人脸、卡、指纹、密码、用户权限、时段、假日计划、用户类型等）进行增删查改的操作。

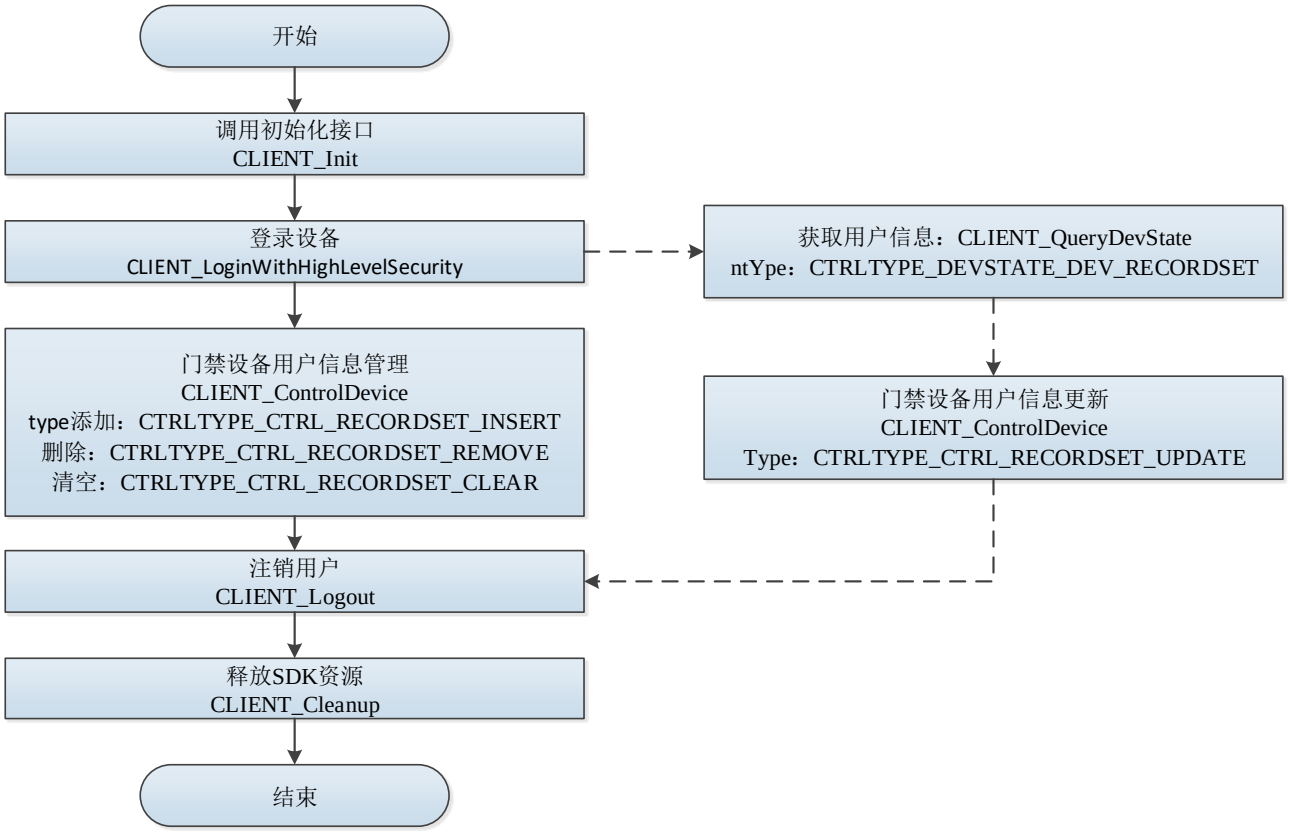
2.2.7.1.2 接口总览

表2-14 人员信息接口说明

接口	说明
CLIENT_ControlDevice	控制设备。
CLIENT_QueryDevState	查询设备状态。

2.2.7.1.3 流程说明

图2-13 用户信息管理业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_ControlDevice 函数来对用户信息进行操作。

表2-15 type 对应含义和结构体

Type	对应含义	emType	Param
● CTRLTYPE_CTRL_RECORDSET_INSERT ● CTRLTYPE_CTRL_RECORDSET_INSERTEX	添加用户信息	NET_RECORD_ACCESSCTLCARD	● NET_CTRL_RECORDSET_INSERT_PARAM ● NET_RECORDSET_ACCESS_CTL_CARD
CTRLTYPE_CTRL_RECORDSET_REMOVE	删除用户信息	NET_RECORD_ACCESSCTLCARD	● NET_CTRL_RECORDSET_PARAM ● NET_RECORDSET_ACCESS_CTL_CARD
CTRLTYPE_CTRL_RECORDSET_CLEAR	清空用户信息	NET_RECORD_ACCESSCTLCARD	NET_CTRL_RECORDSET_PARAM

步骤4　先调用 CLIENT_QueryDevState 接口来获取用户信息。

表2-16 type 对应含义和结构体

Type	对应含义	emType	Param
NET_DEVSTATE_DEV_RECORDSET	获取用户信息	NET_RECORD_ACCESSCTLCARD	NET_CTRL_RECORDSET_PARAM NET_RECORDSET_ACCESS_CTL_CARD

步骤5　再调用 CLIENT_ControlDevice 函数更新用户信息。

表2-17 type 对应含义和结构体

Type	对应含义	emType	Param
● CTRLTYPE_CTRL_RECORDSET_UPDATE ● CTRLTYPE_CTRL_RECORDSET_UPDATEEX	更新用户信息	NET_RECORD_ACCESSCTLCARD	● NET_CTRL_RECORDSET_PARAM ● NET_RECORDSET_ACCESS_CTL_CARD

步骤6　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤7　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.7.1.4 注意事项

- 卡号：人员卡号。
- 卡类型：当设置为胁迫卡时，与卡绑定的人员通过卡密码、开门密码、指纹开门时，会触发胁迫报警 。
- 卡密码：此密码适用于卡+密码开门模式。
- 时间段：选择配置时间段对应序号。如果未设置序号，请到“时间段”配置中进行配置，具体方法请参见“2.2.9.1 时段配置”。
- 开门密码：设置此密码后，无需刷卡直接输入密码即可开门，详细介绍请参见“2.2.10.4 开门密码”。
- 有效次数：仅来宾用户支持设置此字段。
- 是否为首卡：根据实际情况选择。首卡信息的配置方法，请参见“2.2.10.1 分时段、首卡开门”。

2.2.7.1.5 示例代码

```
/**
 * 插入密码
 */
public void insertPassword() {

    // 密码的编号, 支持 500 个, 不重复
    final String userId = "1";

    // 开门密码
    final String openDoorPassword = "888887";

    NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD accessInsert = new
NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD();

```

```
        System.arraycopy(userId.getBytes(), 0, accessInsert.szUserID,
                0, userId.getBytes().length);
        System.arraycopy(openDoorPassword.getBytes(), 0, accessInsert.szDoorOpenPwd,
                0, openDoorPassword.getBytes().length);

        /// 以下字段可以固定，目前设备做了限制必须要带
        accessInsert.nDoorNum = 2; // 门个数 表示双门控制器
        accessInsert.sznDoors[0] = 0; // 表示第一个门有权限
        accessInsert.sznDoors[1] = 1; // 表示第二个门有权限
        accessInsert.nTimeSectionNum = 2; // 与门数对应
        accessInsert.nTimeSectionIndex[0] = 255; // 表示第一个门全天有效
        accessInsert.nTimeSectionIndex[1] = 255; // 表示第二个门全天有效

        NetSDKLib.NET_CTRL_RECORDSET_INSERT_PARAM insert = new
NetSDKLib.NET_CTRL_RECORDSET_INSERT_PARAM();
        insert.stuCtrlRecordSetInfo.emType =
NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;      // 记录集信息类型
        insert.stuCtrlRecordSetInfo.pBuf = accessInsert.getPointer();

        accessInsert.write();
        insert.write();
        boolean success = netSdk.CLIENT_ControlDevice(loginHandle,
                NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_INSERT, insert.getPointer(),
5000);
        insert.read();
        accessInsert.read();

        if(!success) {
            System.err.println("insert password failed. 0x" +
Long.toHexString(netSdk.CLIENT_GetLastError()));
            return;
        }

        System.out.println("Password nRecNo : " + insert.stuCtrlRecordSetResult.nRecNo);
        passwordRecordNo = insert.stuCtrlRecordSetResult.nRecNo;
}

/**
 * 更新密码
 */
public void updatePassword() {

        NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD accessUpdate = new
NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD();
        accessUpdate.nRecNo = passwordRecordNo; // 需要修改的记录集编号,由插入获得

        /// 密码编号，必填否则更新密码不起作用
```

```
        final String userId = String.valueOf(accessUpdate.nRecNo);
        System.arraycopy(userId.getBytes(), 0, accessUpdate.szUserID,
                0, userId.getBytes().length);

        // 新的开门密码
        final String newPassord = "333333";
        System.arraycopy(newPassord.getBytes(), 0,
                accessUpdate.szDoorOpenPwd, 0, newPassord.getBytes().length);

        /// 以下字段可以固定，目前设备做了限制必须要带
        accessUpdate.nDoorNum = 2; // 门个数 表示双门控制器
        accessUpdate.sznDoors[0] = 0; // 表示第一个门有权限
        accessUpdate.sznDoors[1] = 1; // 表示第二个门有权限
        accessUpdate.nTimeSectionNum = 2; // 与门数对应
        accessUpdate.nTimeSectionIndex[0] = 255; // 表示第一个门全天有效
        accessUpdate.nTimeSectionIndex[1] = 255; // 表示第二个门全天有效

        NetSDKLib.NET_CTRL_RECORDSET_PARAM update = new
NetSDKLib.NET_CTRL_RECORDSET_PARAM();
        update.emType = NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;
// 记录集信息类型
        update.pBuf = accessUpdate.getPointer();

        accessUpdate.write();
        update.write();
        boolean result = netSdk.CLIENT_ControlDevice(loginHandle,
                NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_UPDATE, update.getPointer(),
5000);
        update.read();
        accessUpdate.read();
        if (!result) {
            System.err.println("update password failed. 0x" +
Long.toHexString(netSdk.CLIENT_GetLastError()));
        }else {
            System.out.println("update password success");
        }
}

/**
 * 全部删除
 */
public void alldeleteOperate() {
        int type = NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_CLEAR;
        NetSDKLib.NET_CTRL_RECORDSET_PARAM param = new
NetSDKLib.NET_CTRL_RECORDSET_PARAM();
        if (flg) {
            param.emType = NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_TRAFFICREDLIST;
```

```
    } else {
        param.emType =
NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_TRAFFICBLACKLIST;
    }
    param.write();
    boolean zRet = netsdk.CLIENT_ControlDevice(m_hLoginHandle, type, param.getPointer(), 5000);
    if (zRet) {
        System.out.println("全部删除成功");
    } else {
        System.err.println("全部删除失败" + String.format("0x%x", netsdk.CLIENT_GetLastError()));
    }
}
    public static boolean queryRecordState(SdkStructure condition) {

        int emType = getRecordType(condition);
        if (emType == EM_NET_RECORD_TYPE.NET_RECORD_UNKNOWN) {
            System.err.println("the input query condition SdkStructure [" + condition.getClass() + "]
invalid!");
            return false;
        }

        NetSDKLib.NET_CTRL_RECORDSET_PARAM record = new
NetSDKLib.NET_CTRL_RECORDSET_PARAM();
        record.emType = emType;
        record.pBuf = condition.getPointer();

        IntByReference intRetLen = new IntByReference();

        condition.write();
        record.write();
        if (!netsdkApi.CLIENT_QueryDevState(loginHandle,
NetSDKLib.NET_DEVSTATE_DEV_RECORDSET,
                record.getPointer(), record.size(), intRetLen, 3000)) {
            return false;
        }

        record.read();
        condition.read();

        output(condition);

        return true;
    }

    private static void output(SdkStructure record) { // 具体输出在上层定义

        if (record instanceof NET_RECORDSET_ACCESS_CTL_CARDREC) {
```

```
                printRecord((NET_RECORDSET_ACCESS_CTL_CARDREC)record);

            }else if(record instanceof NET_RECORDSET_ACCESS_CTL_CARD) {

                printRecord((NET_RECORDSET_ACCESS_CTL_CARD)record);

            }
        }

        public static int getRecordNo() {
            return nRecordNo;
        }

        private static int getRecordType(SdkStructure object) {
            int type = EM_NET_RECORD_TYPE.NET_RECORD_UNKNOWN;

            if (object instanceof NET_RECORDSET_ACCESS_CTL_CARD
                    || object instanceof FIND_RECORD_ACCESSCTLCARD_CONDITION) {

                type = EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLCARD;

            }else if (object instanceof NET_RECORD_ACCESSQRCODE_INFO) {

                type = EM_NET_RECORD_TYPE.NET_RECORD_ACCESSQRCODE;

            }else if (object instanceof NET_RECORDSET_ACCESS_CTL_PWD) {

                type = EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;

            }else if (object instanceof NET_RECORDSET_ACCESS_CTL_CARDREC
                    || object instanceof FIND_RECORD_ACCESSCTLCARDREC_CONDITION_EX) {

                type = EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLCARDREC_EX;

            }

            return type;
        }
}
```

2.2.8 门配置

2.2.8.1 简介

门配置信息，即用户通过调用 SDK 接口，对门禁设备的门配置进行获取和设置的操作，包括开

门模式、门锁保持时间、关门超时时长、假期时间段编号、开门时间段及报警使能项等。

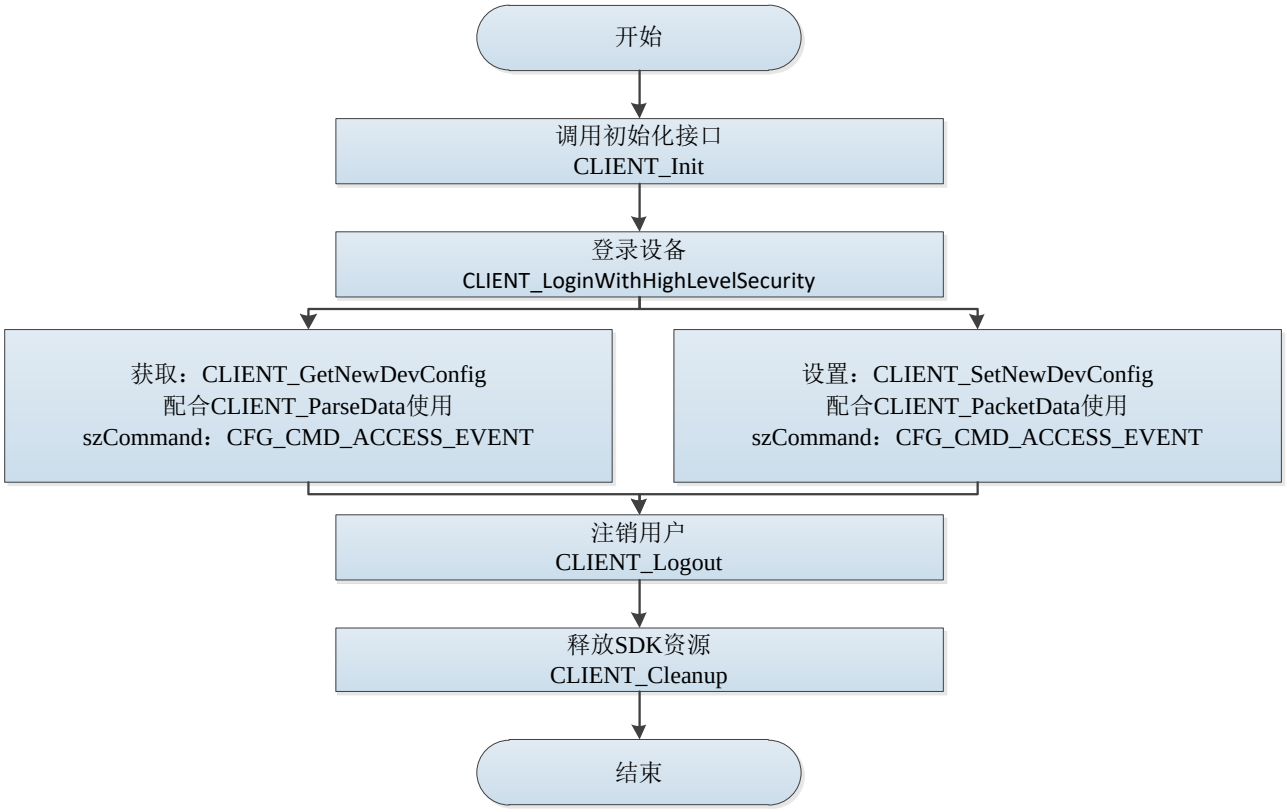
2.2.8.2 接口总览

表2-18 门配置信息接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.8.3 流程说明

图2-14 门配置信息业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁门配置信息。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-19 CFG_ACCESS_EVENT_INFO 参数说明

CFG_ACCESS_EVENT_INFO 参数	对应含义
emState	门状态
nUnlockHoldInterval	开门时长

CFG_ACCESS_EVENT_INFO 参数	对应含义
nCloseTimeout	关门超时时长
emDoorOpenMethod	开门模式
bDuressAlarmEnable	胁迫
bBreakInAlarmEnable	闯入报警使能
bRepeatEnterAlarm	重复进入报警使能
abDoorNotClosedAlarmEnable	互锁报警使能
abSensorEnable	门磁使能

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁门配置信息。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-20 CFG_ACCESS_EVENT_INFO 参数说明

CFG_ACCESS_EVENT_INFO 参数	对应含义
emState	门状态
nUnlockHoldInterval	开门时长
nCloseTimeout	关门超时时长
emDoorOpenMethod	开门模式
bDuressAlarmEnable	胁迫
bBreakInAlarmEnable	闯入报警使能
bRepeatEnterAlarm	重复进入报警使能
abDoorNotClosedAlarmEnable	互锁报警使能
abSensorEnable	门磁使能

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.8.4 注意事项

- 开启闯入报警、门未关报警功能时，需要同时开启门磁功能，才能实现闯入报警、门未关报警功能。
- 常开、常闭、远程验证引用的序号，通过“时间段”进行设置，具体设置方法请参见“2.2.9.1 时段配置”。

2.2.8.5 示例代码

```
// 门禁事件配置
public void AccessConfig() {
    // 获取
    String szCommand = NetSDKLib.CFG_CMD_ACCESS_EVENT;
    int nChn = 0;  // 通道
    CFG_ACCESS_EVENT_INFO access = new CFG_ACCESS_EVENT_INFO();  //
m_stDeviceInfo.byChanNum 为设备通道数

    if (ToolKits.GetDevConfig(loginHandle, nChn, szCommand, access)) {
        System.out.println("门禁通道名称:" + new String(access.szChannelName).trim());
        System.out.println("首卡使能:" + access.stuFirstEnterInfo.bEnable);    // 0-false; 1-true
```

```
            System.out.println("首卡权限验证通过后的门禁状态:" + access.stuFirstEnterInfo.emStatus); //
状态参考枚举   CFG_ACCESS_FIRSTENTER_STATUS
            System.out.println("需要首卡验证的时间段，值为通道号:" +
access.stuFirstEnterInfo.nTimeIndex);
        }

        // 设置
        boolean bRet = ToolKits.SetDevConfig(loginHandle, nChn, szCommand, access);
        if (bRet) {
            System.out.println("Set Succeed!");
        }
}
```

2.2.9 门时间配置

2.2.9.1 时段配置

2.2.9.1.1 简介

时段配置信息，即用户通过调用 SDK 接口，对门禁设备的门时间段进行获取和设置的操作。此模板的配置不能直接对设备生效，需要被其他功能模块调用。

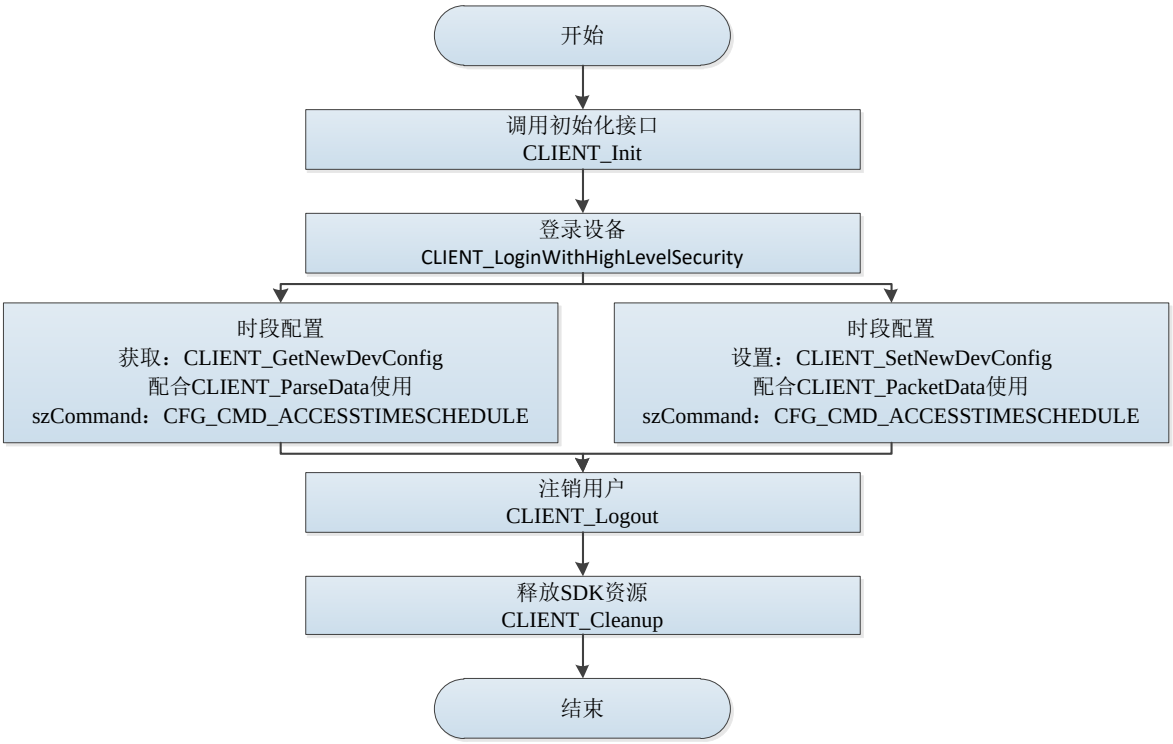
2.2.9.1.2 接口总览

表2-21 时段接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.9.1.3 流程说明

图2-15 时段业务配置流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁时段信息。

- szCommand：CFG_CMD_ACCESSTIMESCHEDULE。
- pBuf：CFG_ACCESS_TIMESCHEDULE_INFO。

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁时段信息。

- szCommand：CFG_CMD_ACCESSTIMESCHEDULE。
- pBuf：CFG_ACCESS_TIMESCHEDULE_INFO。

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.9.1.4 示例代码

```
/**
 * 门禁刷卡时间段设置
 */
public void setAccessTimeSchedule() {
    CFG_ACCESS_TIMESCHEDULE_INFO msg = new CFG_ACCESS_TIMESCHEDULE_INFO();

    String strCmd = NetSDKLib.CFG_CMD_ACCESSTIMESCHEDULE;
    int nChannel = 120; // 通道号

    // 获取
    if(ToolKits.GetDevConfig(loginHandle, nChannel, strCmd, msg)) {
        System.out.println("Enable:" + msg.bEnable);
```

```
            try {
                System.out.println("自定义名称:" + new String(msg.szName, "GBK").trim());
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }

            for(int i = 0; i < 7; i++) {
                for(int j = 0; j < 4; j++) {
                    System.out.println("dwRecordMask:" +
msg.stuTimeWeekDay[i].stuTimeSection[j].dwRecordMask);
                    System.out.println(msg.stuTimeWeekDay[i].stuTimeSection[j].startTime() + "-" +
                                        msg.stuTimeWeekDay[i].stuTimeSection[j].endTime() + "\n");
                }
            }

            // 设置
            if(ToolKits.SetDevConfig(loginHandle, nChannel, strCmd, msg)) {
                System.out.println("Set AccessTimeSchedule Succeed!");
            } else {
                System.err.println("Set AccessTimeSchedule Failed!" + ToolKits.getErrorCode());
            }
        } else {
            System.err.println("Get AccessTimeSchedule Failed!" + ToolKits.getErrorCode());
        }
    }
```

2.2.9.2 常开常闭时间段配置

2.2.9.2.1 简介

常开常闭时间段配置，即用户通过调用 SDK 接口，对门禁设备的时间段配置进行获取和设置的操作，这里包括常开/常闭时段配置、远程验证时间段配置等信息

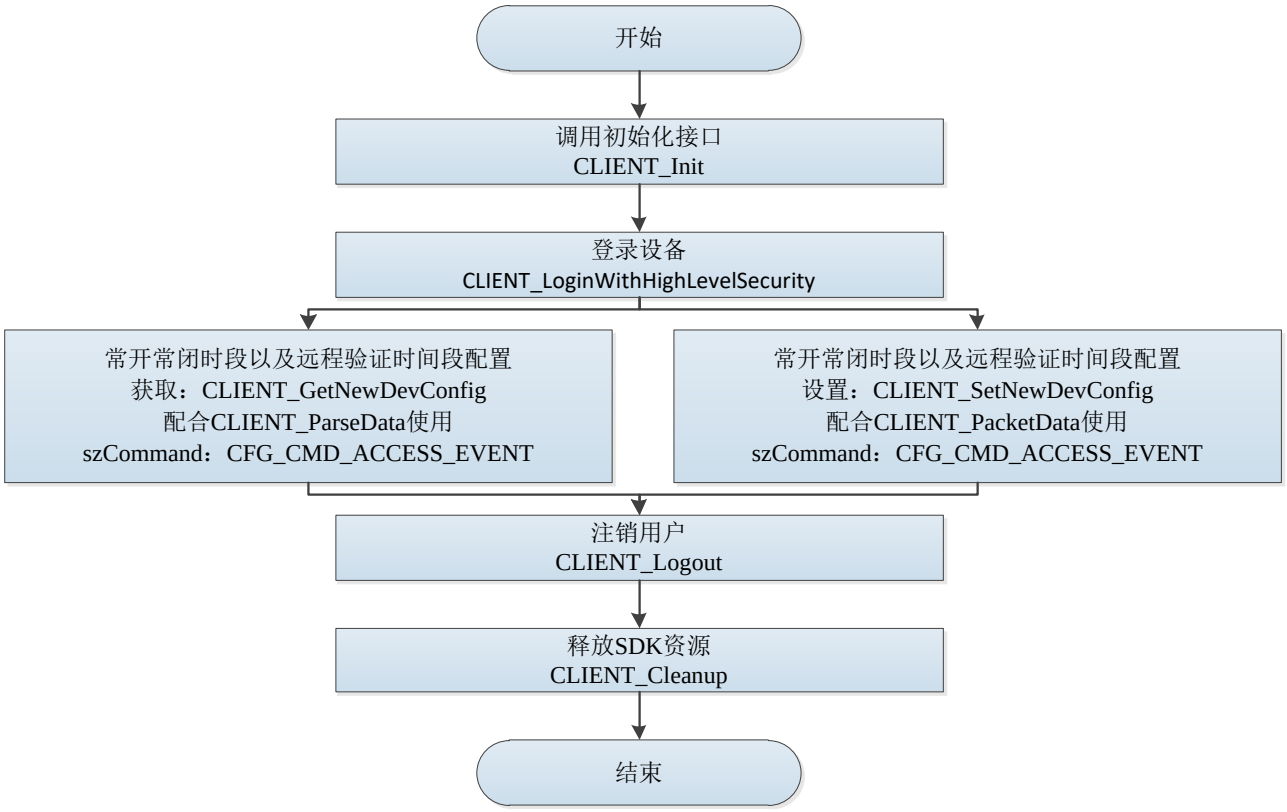
2.2.9.2.2 接口总览

表2-22 常开常闭时间段配置接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.9.2.3 流程说明

图2-16 常开常闭时间段配置流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁常开常闭时段配置信息以及远程验证时间段。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-23 CFG_ACCESS_EVENT_INFO 参数说明

CFG_ACCESS_EVENT_INFO 参数	对应含义
nOpenAlwaysTimeIndex	常开时间段配置
nCloseAlwaysTimeIndex	常闭时间段配置
stuAutoRemoteCheck	远程验证时间段

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁常开常闭时段配置信息以及远程验证时间段。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-24 CFG_ACCESS_EVENT_INFO 参数说明

CFG_ACCESS_EVENT_INFO 参数	对应含义
nOpenAlwaysTimeIndex	常开时间段配置
nCloseAlwaysTimeIndex	常闭时间段配置
stuAutoRemoteCheck	远程验证时间段

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.9.2.4 注意事项

常开、常闭、远程验证引用的序号，通过“时间段”进行设置，具体设置方法请参见“2.2.9.1 时段配置”。

2.2.9.2.5 示例代码

```
/**
 * 门禁刷卡时间段设置
 */
public void setAccessTimeSchedule() {
    CFG_ACCESS_TIMESCHEDULE_INFO msg = new CFG_ACCESS_TIMESCHEDULE_INFO();

    String strCmd = NetSDKLib.CFG_CMD_ACCESSTIMESCHEDULE;
    int nChannel = 120; // 通道号

    // 获取
    if(ToolKits.GetDevConfig(loginHandle, nChannel, strCmd, msg)) {
        System.out.println("Enable:" + msg.bEnable);
        try {
            System.out.println("自定义名称:" + new String(msg.szName, "GBK").trim());
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }

        for(int i = 0; i < 7; i++) {
            for(int j = 0; j < 4; j++) {
                System.out.println("dwRecordMask:" +
msg.stuTimeWeekDay[i].stuTimeSection[j].dwRecordMask);
                System.out.println(msg.stuTimeWeekDay[i].stuTimeSection[j].startTime() + "-" +
                                        msg.stuTimeWeekDay[i].stuTimeSection[j].endTime() + "\n");
            }
        }

        // 设置
        if(ToolKits.SetDevConfig(loginHandle, nChannel, strCmd, msg)) {
            System.out.println("Set AccessTimeSchedule Succeed!");
        } else {
            System.err.println("Set AccessTimeSchedule Failed!" + ToolKits.getErrorCode());
        }
    } else {
        System.err.println("Get AccessTimeSchedule Failed!" + ToolKits.getErrorCode());
    }
}
```

2.2.10 门高级配置

2.2.10.1 分时段、首卡开门

2.2.10.1.1 简介

分时段、首卡开门，即用户通过调用 SDK 接口，对门禁设备的分时段、首卡、首用户开门配置进行获取和设置的操作。

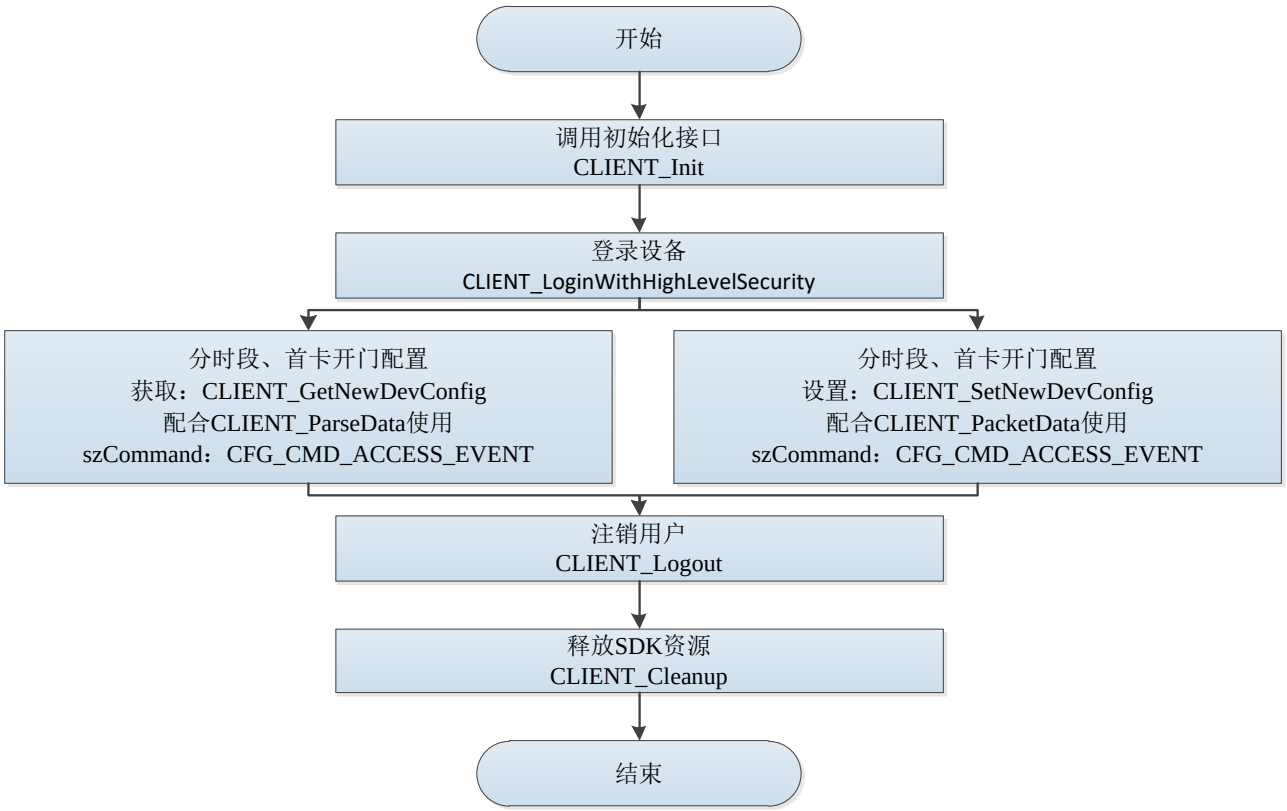
2.2.10.1.2 接口总览

表2-25 分时段和首卡开门接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.10.1.3 流程说明

图2-17 分时段和首卡开门业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁分时段、

首卡开门配置信息。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-26 CFG_ACCESS_EVENT_INFO 结构体参数说明

参数	含义
stuDoorTimeSection	分时段开门配置
stuFirstEnterInfo	首用户/卡开门配置

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁分时段、首卡开门配置信息。

- szCommand：CFG_CMD_ACCESS_EVENT。
- pBuf：CFG_ACCESS_EVENT_INFO。

表2-27 _ACCESS_EVENT_INFO 结构体参数说明

参数	含义
stuDoorTimeSection	分时段开门配置
stuFirstEnterInfo	首用户/卡开门配置

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.10.1.4 注意事项

- 首卡引用的用户 ID 为卡号。
- 为实现首卡开门功能，添加该用户 ID 对应人员到设备时，需要选择“首卡”，否则，首卡开门功能无法使用。

2.2.10.1.5 示例代码

```
// 门禁事件配置
public void AccessConfig() {
    // 获取
    String szCommand = NetSDKLib.CFG_CMD_ACCESS_EVENT;
    int nChn = 0; // 通道
    CFG_ACCESS_EVENT_INFO access = new CFG_ACCESS_EVENT_INFO(); //
m_stDeviceInfo.byChanNum 为设备通道数

    if (ToolKits.GetDevConfig(loginHandle, nChn, szCommand, access)) {
        System.out.println("门禁通道名称:"
                + new String(access.szChannelName).trim());
        System.out.println("首卡使能:" + access.stuFirstEnterInfo.bEnable); // 0-false;
                                                                    // 1-true
        System.out.println("首卡权限验证通过后的门禁状态:"
                + access.stuFirstEnterInfo.emStatus); // 状态参考枚举
                                                      //
CFG_ACCESS_FIRSTENTER_STATUS
        System.out.println("需要首卡验证的时间段，值为通道号:"
                + access.stuFirstEnterInfo.nTimeIndex);

        System.out.println("  当前门采集状态:" + access.emReadCardState);
    }
```

```
        // 设置
//          access.emReadCardState = EM_CFG_CARD_STATE.EM_CFG_CARD_STATE_SWIPE;   //
门禁刷卡
        access.emReadCardState = EM_CFG_CARD_STATE.EM_CFG_CARD_STATE_COLLECTION;
         // 门禁采集卡
//          access.emReadCardState = EM_CFG_CARD_STATE.EM_CFG_CARD_STATE_UNKNOWN;
        // 退出读卡状态

        boolean bRet = ToolKits.SetDevConfig(loginHandle, nChn, szCommand,
                access);
        if (bRet) {
            System.out.println("Set Succeed!");
        }
}
```

2.2.10.2 多人组合开门

2.2.10.2.1 简介

多人组合开门，即用户通过调用 SDK 接口，对门禁设备的多人组合开门配置信息进行获取和设置的操作。

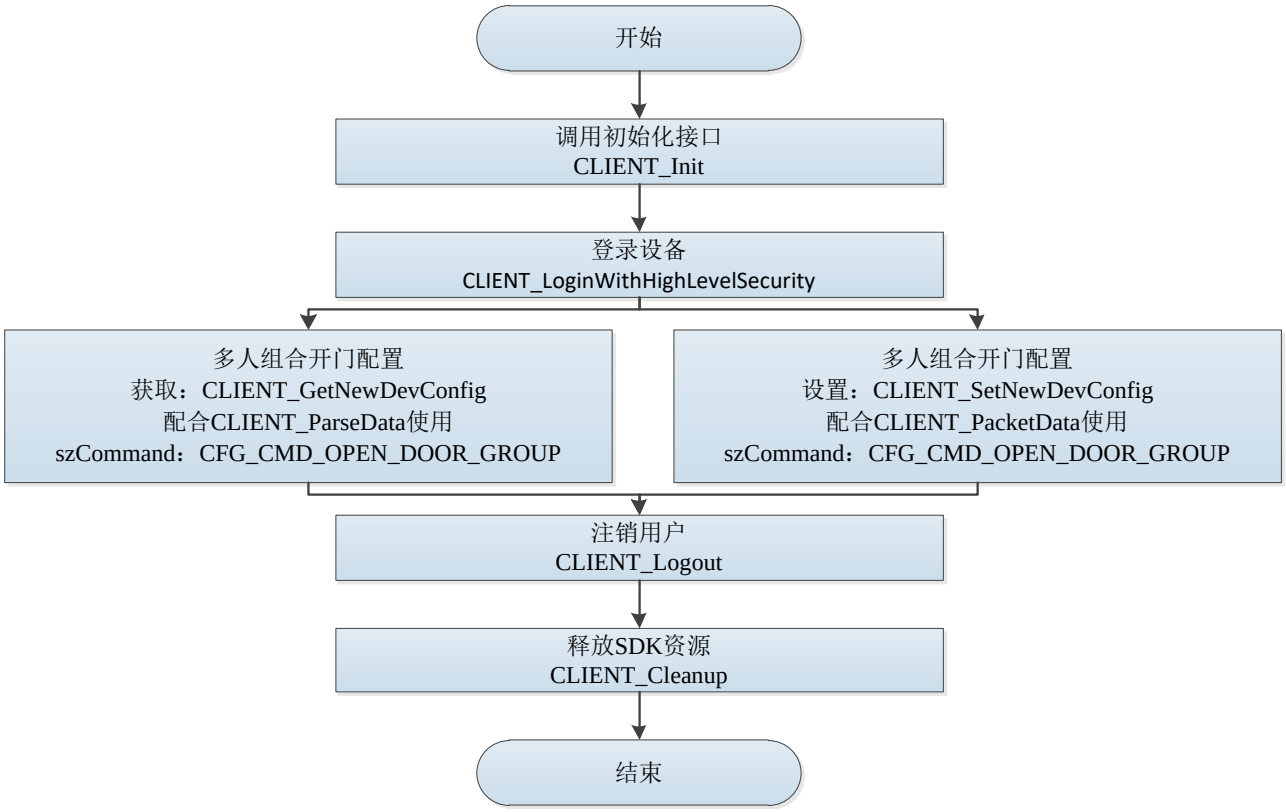
2.2.10.2.2 接口总览

表2-28 多人组合开门接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。
CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.10.2.3 流程说明

图2-18 多人组合开门业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁多人组合开门配置信息。

- szCommand：CFG_CMD_OPEN_DOOR_GROUP。
- pBuf：CFG_OPEN_DOOR_GROUP_INFO。

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁多人组合开门配置信息。

- szCommand：CFG_CMD_OPEN_DOOR_GROUP。
- pBuf：CFG_OPEN_DOOR_GROUP_INFO。

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.10.2.4 注意事项

- 配置多人组合开门功能之前，需要先将人员添加到设备。
- 组合序号：将人员进行分组，1 个门最多可以配置 4 个人员组。
- 人员组：序号组内的人员。1 个组内人员人数最多为 50 人。人员需要提前添加到设备。
- 有效人数：每个组内的有效人数≤组内当前人数，1 个门的有效人员总数≤5 人。
- 设置人员组开门方式：仅支持卡、指纹，二选一。

2.2.10.2.5 示例代码

```
/**
 * 多人多开门方式组合配置
 * @param nChannel       通道号
 * @param msg              多人多开门方式组合配置
 */
public boolean MoreOpenDoor(int nChannel,CFG_OPEN_DOOR_GROUP_INFO msg){
    // 获取
    String szCommand = NetSDKLib.CFG_CMD_OPEN_DOOR_GROUP;
    if (!ToolKits.GetDevConfig(loginHandle, nChannel, szCommand, msg)) {
            System.err.println("Get more open door Failed.");
            return false;
    }
    System.out.println("有效组合数" + msg.nGroup + " 用户数目 " + msg.stuGroupInfo.length );

    //设置
    msg.stuGroupInfo[0].stuGroupDetail[0].emMethod=1;
    msg.stuGroupInfo[0].stuGroupDetail[1].emMethod=1;
    System.arraycopy("123".getBytes(), 0, msg.stuGroupInfo[0].stuGroupDetail[0].szUserID, 0,
"123".getBytes().length);
    System.arraycopy("234".getBytes(), 0, msg.stuGroupInfo[0].stuGroupDetail[1].szUserID, 0,
"234".getBytes().length);
    for(int i=0;i<msg.stuGroupInfo.length;i++){
        msg.stuGroupInfo[i].nUserCount=2;
        msg.stuGroupInfo[i].nGroupNum=2;
        msg.stuGroupInfo[i].bGroupDetailEx=true;
    }
    if(!ToolKits.SetDevConfig(loginHandle, nChannel, szCommand, msg)){
        System.err.println("Set more open door Failed!");
        return false;
    }
    System.out.println("Set more open door Succeed!");
    return true;
}
```

2.2.10.3 多门互锁

2.2.10.3.1 简介

多门互锁配置，即用户通过调用 SDK 接口，对门禁设备的多门互锁配置进行获取和设置的操作。

2.2.10.3.2 接口总览

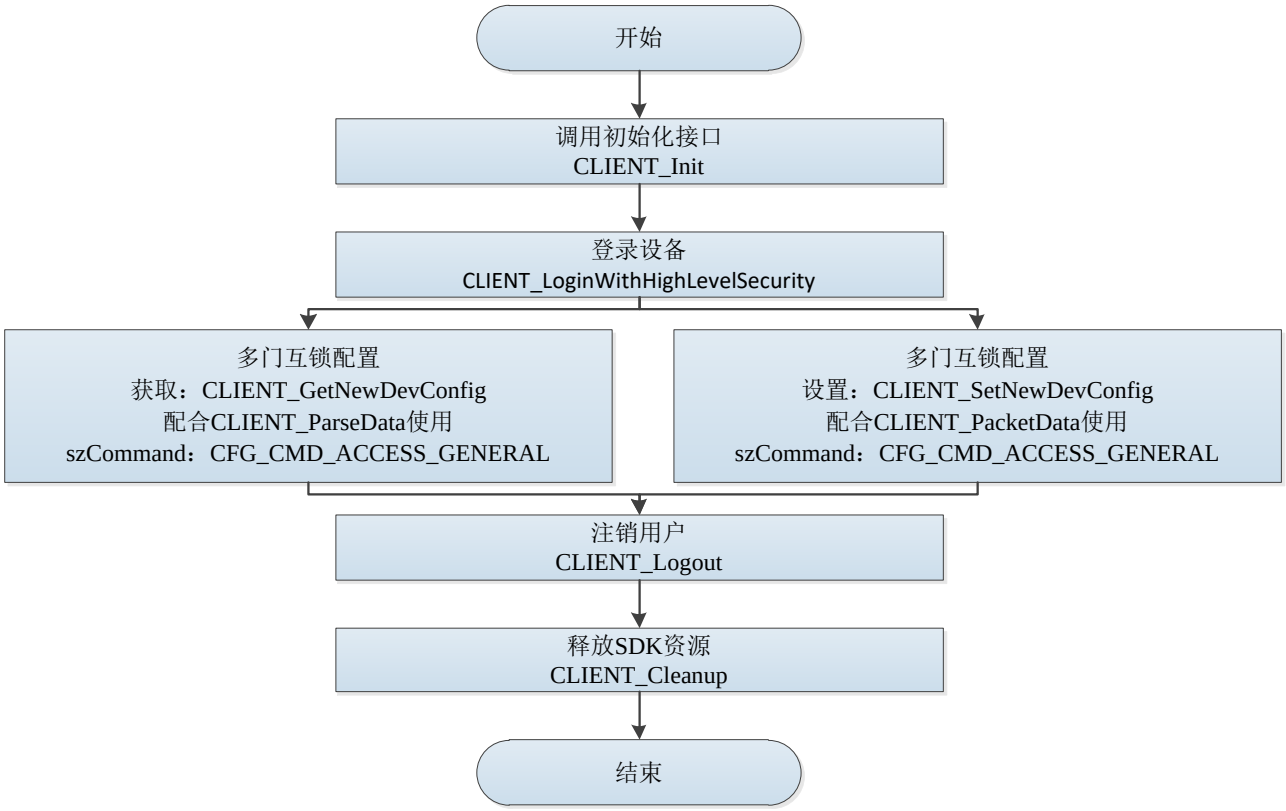
表2-29 多门互锁接口说明

接口	说明
CLIENT_GetNewDevConfig	查询配置信息。
CLIENT_ParseData	解析查询到的配置信息。
CLIENT_SetNewDevConfig	设置配置信息。

CLIENT_PacketData	将需要设置的配置信息，打包成字符串格式。

2.2.10.3.3 流程说明

图2-19 多门互锁配置业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetNewDevConfig 函数，结合 CLIENT_ParseData 来查询门禁多门互锁配置信息。

- szCommand：CFG_CMD_ACCESS_GENERAL。
- pBuf：CFG_ACCESS_GENERAL_INFO。

步骤4　调用 CLIENT_SetNewDevConfig 函数，结合 CLIENT_PacketData 来设置门禁多门互锁配置信息。

- szCommand：CFG_CMD_ACCESS_GENERAL。
- pBuf：CFG_ACCESS_GENERAL_INFO。

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.10.3.4 注意事项

1 个设备只能支持 1 种多门互锁方案。

2.2.10.3.5 示例代码

```
/** 多门互锁配置的获取与下发 */
  public void ABLockInfo() {
    // 对于配置的下发，建议先获取再修改配置项下发，可以避免漏掉或误修改其他无关配置项
```

```
    CFG_ACCESS_GENERAL_INFO inParam = new CFG_ACCESS_GENERAL_INFO();
    inParam =
        (CFG_ACCESS_GENERAL_INFO)
            configModule.getNewConfig(loginHandler, CFG_CMD_ACCESS_GENERAL, inParam, 0,
3000);
    System.out.println("修改前,多门互锁使能:" + inParam.stuABLockInfo.bEnable + ",配置为:");
    for (int i = 0; i < inParam.stuABLockInfo.nDoors; i++) {
      for (int j = 0; j < inParam.stuABLockInfo.stuDoors[i].nDoor; j++) {
        // 互锁信息
        System.out.println(i + "," + j + ",door:" + inParam.stuABLockInfo.stuDoors[i].anDoor[j]);
      }
    }
    // 修改多门互锁配置
    inParam.abABLockInfo = 1;
    inParam.stuABLockInfo.bEnable = !inParam.stuABLockInfo.bEnable;
    inParam.stuABLockInfo.nDoors = 1;
    int doors = 2;
    inParam.stuABLockInfo.stuDoors[0].nDoor = doors;
    inParam.stuABLockInfo.stuDoors[0].anDoor[0] = 1;
    inParam.stuABLockInfo.stuDoors[0].anDoor[1] = 2;

    inParam.stuABLockInfo.stuDoors[1].nDoor = doors;
    inParam.stuABLockInfo.stuDoors[1].anDoor[0] = 0;
    inParam.stuABLockInfo.stuDoors[1].anDoor[1] = 3;
    boolean result =
        configModule.setNewConfig(loginHandler, CFG_CMD_ACCESS_GENERAL, inParam, 0,
3000);
    // 下发配置成功,则重新获取配置信息,打印
    if (result) {
      inParam =
          (CFG_ACCESS_GENERAL_INFO)
              configModule.getNewConfig(loginHandler, CFG_CMD_ACCESS_GENERAL, inParam,
3, 3000);
      System.out.println("修改后,多门互锁使能:" + inParam.stuABLockInfo.bEnable + ",配置为:");
    }
  }
```

2.2.10.4 开门密码

2.2.10.4.1 简介

开门密码，即用户通过调用 SDK 接口，对门禁设备开门密码进行增删查改等操作。

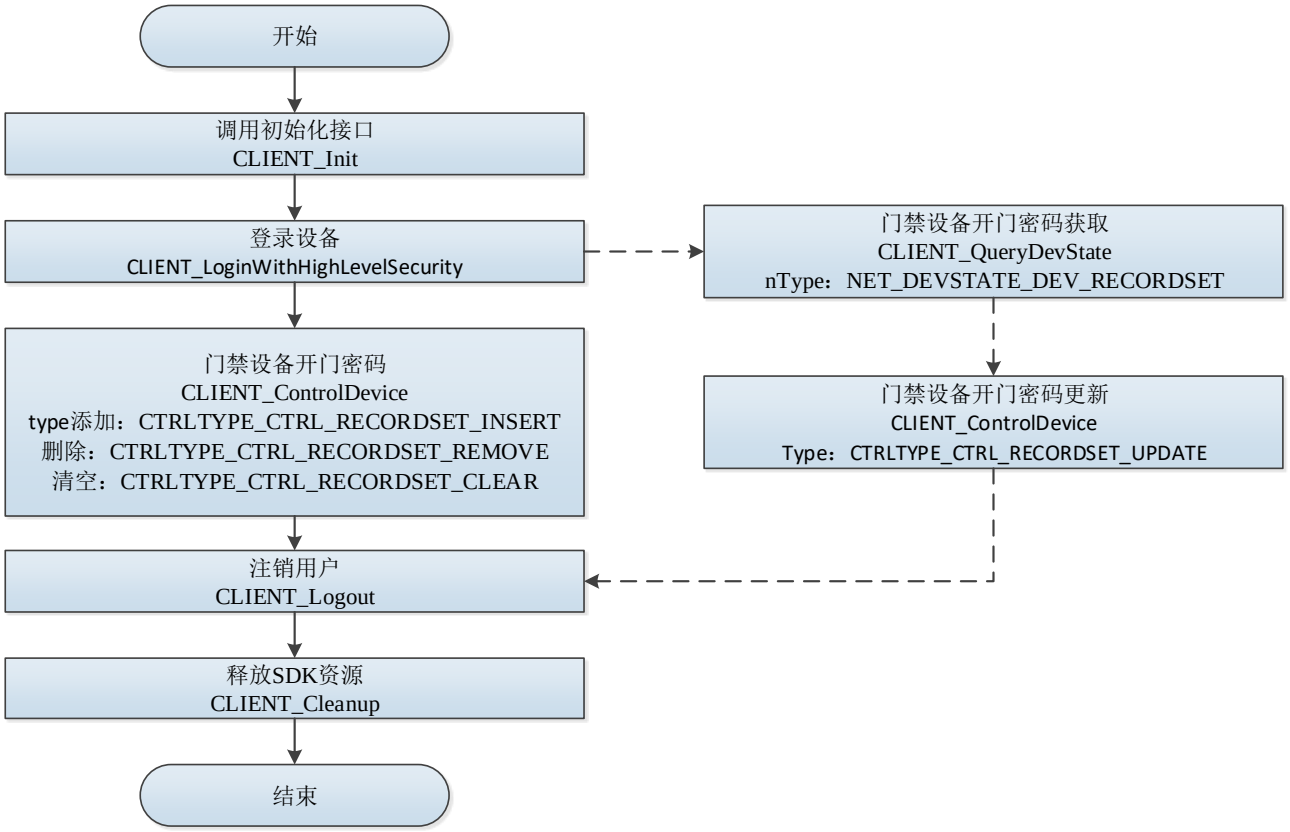
2.2.10.4.2 接口总览

表2-30 开门密码接口说明

接口	说明
CLIENT_ControlDevice	设备控制。

2.2.10.4.3 流程说明

图2-20 开门密码配置流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_ControlDevice 函数来对开门密码信息进行操作。

表2-31 Type 对应操作和结构体

type	含义	emType	Param
CTRLTYPE_CTRL_RECORDSET_INSERT	添加开门密码	NET_RECORD_ACCESSCTLPWD	NET_CTRL_RECORDSET_INSERT_PARA NET_RECORDSET_ACCESS_CTL_PWD
CTRLTYPE_CTRL_RECORDSET_REMOVE	删除开门密码	NET_RECORD_ACCESSCTLPWD	NET_CTRL_RECORDSET_PARAM NET_RECORDSET_ACCESS_CTL_PWD
CTRLTYPE_CTRL_RECORDSET_CLEAR	清空开门密码	NET_RECORD_ACCESSCTLPWD	NET_CTRL_RECORDSET_PARAM

步骤4　先调用 CLIENT_QueryDevState 接口来获取开门密码信息。

表2-32 Type 对应操作和结构体

type	含义	emType	Param
NET_DEVSTATE_DEV_RECORDSET	获取开门密码	NET_RECORD_ACCESSCTLPWD	NET_CTRL_RECORDSET_PARAM NET_RECORDSET_ACCESS_CTL_PWD

步骤5　再调用 CLIENT_ControlDevice 函数更新开门密码信息。

表2-33 Type 对应操作和结构体

type	含义	emType	Param
CTRLTYPE_CTRL_RECORDSET_UPDATE	获取开门密码	NET_RECORD_ACCESSCTLPWD	NET_CTRL_RECORDSET_PARAM NET_RECORDSET_ACCESS_CTL_PWD

步骤6　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤7　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.10.4.4 注意事项

- 配置多人组合开门功能之前，需要先将人员添加到设备。
- 用户编号：人员卡号。

2.2.10.4.5 示例代码

```
/**
 * 插入密码成功后返回的编号, 用于后续的更新、删除等操作
 */
private int passwordRecordNo = 0;

/**
 * 插入密码
 */
public void insertPassword() {

    // 密码的编号, 支持 500 个, 不重复
    final String userId = "1";

    // 开门密码
    final String openDoorPassword = "888887";

    NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD accessInsert = new
NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD();

    System.arraycopy(userId.getBytes(), 0, accessInsert.szUserID,
            0, userId.getBytes().length);
    System.arraycopy(openDoorPassword.getBytes(), 0, accessInsert.szDoorOpenPwd,
            0, openDoorPassword.getBytes().length);

    /// 以下字段可以固定, 目前设备做了限制必须要带
```

```
        accessInsert.nDoorNum = 2; // 门个数 表示双门控制器
        accessInsert.sznDoors[0] = 0; // 表示第一个门有权限
        accessInsert.sznDoors[1] = 1; // 表示第二个门有权限
        accessInsert.nTimeSectionNum = 2; // 与门数对应
        accessInsert.nTimeSectionIndex[0] = 255; // 表示第一个门全天有效
        accessInsert.nTimeSectionIndex[1] = 255; // 表示第二个门全天有效

        NetSDKLib.NET_CTRL_RECORDSET_INSERT_PARAM insert = new
NetSDKLib.NET_CTRL_RECORDSET_INSERT_PARAM();
        insert.stuCtrlRecordSetInfo.emType =
NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;      // 记录集信息类型
        insert.stuCtrlRecordSetInfo.pBuf = accessInsert.getPointer();

        accessInsert.write();
        insert.write();
        boolean success = netSdk.CLIENT_ControlDevice(loginHandle,
                NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_INSERT, insert.getPointer(),
5000);
        insert.read();
        accessInsert.read();

        if(!success) {
            System.err.println("insert password failed. 0x" +
Long.toHexString(netSdk.CLIENT_GetLastError()));
            return;
        }

        System.out.println("Password nRecNo : " + insert.stuCtrlRecordSetResult.nRecNo);
        passwordRecordNo = insert.stuCtrlRecordSetResult.nRecNo;
}

/**
 * 更新密码
 */
public void updatePassword() {

        NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD accessUpdate = new
NetSDKLib.NET_RECORDSET_ACCESS_CTL_PWD();
        accessUpdate.nRecNo = passwordRecordNo; // 需要修改的记录集编号,由插入获得

        /// 密码编号，必填否则更新密码不起作用
        final String userId = String.valueOf(accessUpdate.nRecNo);
        System.arraycopy(userId.getBytes(), 0, accessUpdate.szUserID,
                0, userId.getBytes().length);

        // 新的开门密码
        final String newPassord = "333333";
```

```
        System.arraycopy(newPassord.getBytes(), 0,
                accessUpdate.szDoorOpenPwd, 0, newPassord.getBytes().length);

        /// 以下字段可以固定，目前设备做了限制必须要带
        accessUpdate.nDoorNum = 2; // 门个数 表示双门控制器
        accessUpdate.sznDoors[0] = 0; // 表示第一个门有权限
        accessUpdate.sznDoors[1] = 1; // 表示第二个门有权限
        accessUpdate.nTimeSectionNum = 2; // 与门数对应
        accessUpdate.nTimeSectionIndex[0] = 255; // 表示第一个门全天有效
        accessUpdate.nTimeSectionIndex[1] = 255; // 表示第二个门全天有效

        NetSDKLib.NET_CTRL_RECORDSET_PARAM update = new
NetSDKLib.NET_CTRL_RECORDSET_PARAM();
        update.emType = NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;
// 记录集信息类型
        update.pBuf = accessUpdate.getPointer();

        accessUpdate.write();
        update.write();
        boolean result = netSdk.CLIENT_ControlDevice(loginHandle,
                NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_UPDATE, update.getPointer(),
5000);
        update.read();
        accessUpdate.read();
        if (!result) {
            System.err.println("update password failed. 0x" +
Long.toHexString(netSdk.CLIENT_GetLastError()));
        }else {
            System.out.println("update password success");
        }
}


/**
 * 删除密码
 */
public void deletePassword() {
        NetSDKLib.NET_CTRL_RECORDSET_PARAM remove = new
NetSDKLib.NET_CTRL_RECORDSET_PARAM();
        remove.emType = NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLPWD;
        remove.pBuf = new IntByReference(passwordRecordNo).getPointer();

        remove.write();
        boolean result = netSdk.CLIENT_ControlDevice(loginHandle,
                NetSDKLib.CtrlType.CTRLTYPE_CTRL_RECORDSET_REMOVE, remove.getPointer(),
5000);
        remove.read();
```

```
        if(!result){
            System.err.println(" remove pawssword failed. 0x" +
Long.toHexString(netSdk.CLIENT_GetLastError()));
        }else {
            System.out.println("remove password success");
        }
}
```

2.2.11 记录查询

2.2.11.1 开门记录

2.2.11.1.1 简介

开门记录查询，即用户调用 SDK 接口对门禁设备的开门进行查询。查询可以设置查询条件，查询条目数。

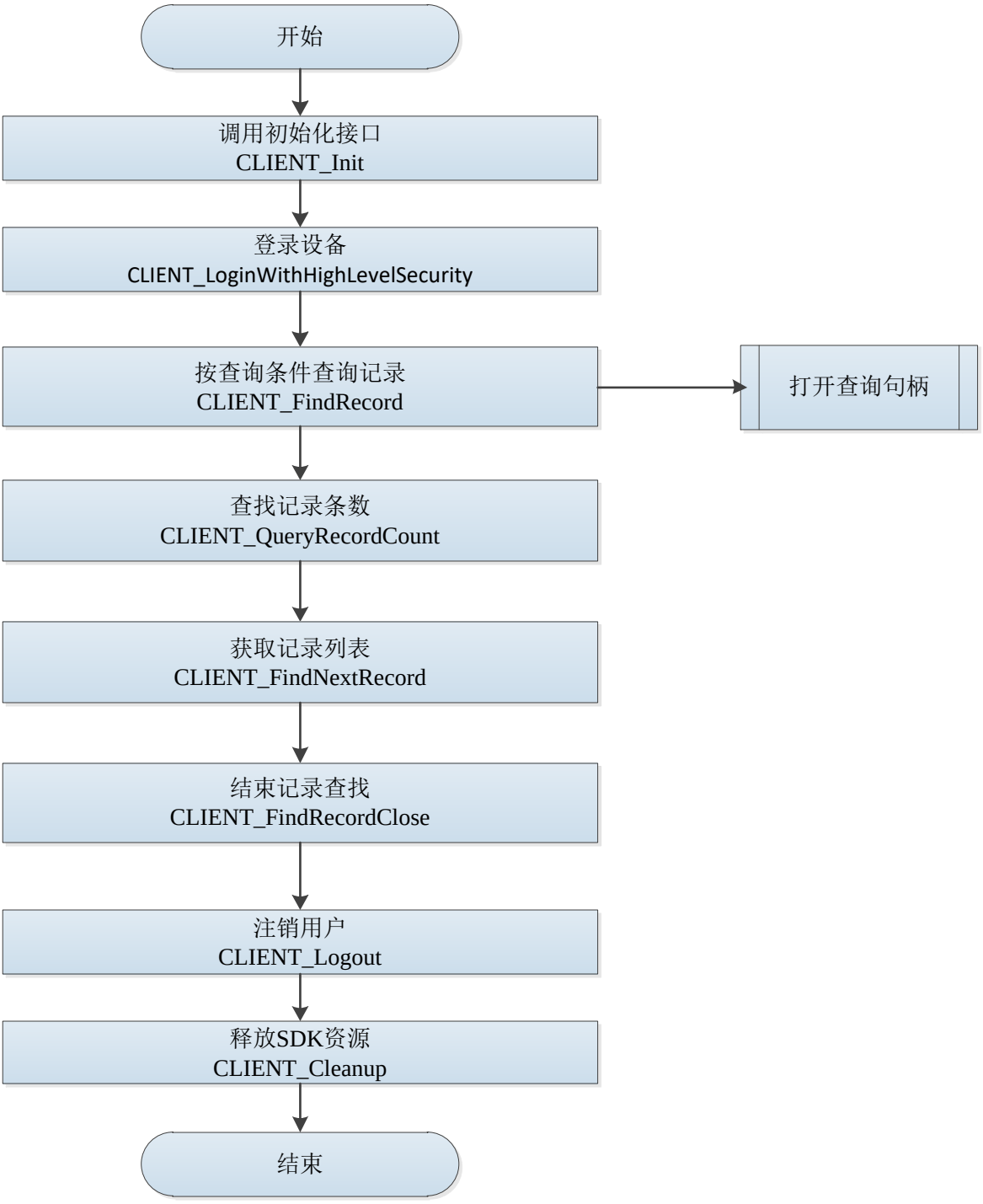
2.2.11.1.2 接口总览

表2-34 记录查询接口说明

接口	说明
CLIENT_QueryRecordCount	查找记录条数
CLIENT_FindRecord	按查询条件查询记录
CLIENT_FindNextRecord	查找记录:nFilecount:需要查询的条数，返回值为媒体文件条数 返回值小于 nFilecount 则相应时间段内的文件查询完毕
CLIENT_FindRecordClose	结束记录查询

2.2.11.1.3 流程说明

图2-21 记录查询业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　先调用 CLIENT_FindRecord 函数获得查询句柄。

emType 开门记录：NET_RECORD_ACCESSCTLCARDREC。

步骤4　再调用 CLIENT_QueryRecordCount 函数查找记录条数。

步骤5　再调用 CLIENT_FindNextRecord 函数获取记录列表。

步骤6　查询完毕可以调用 CLIENT_FindRecordClose 关闭查询句柄。

步骤7　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤8　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.11.1.4 示例代码

```
/// 根据卡号查询门禁刷卡记录
public void findAccessRecordByCardNo() {
    ///查询条件
    NetSDKLib.FIND_RECORD_ACCESSCTLCARDREC_CONDITION_EX recordCondition = new
NetSDKLib.FIND_RECORD_ACCESSCTLCARDREC_CONDITION_EX();
    recordCondition.bCardNoEnable = 1; //启用卡号查询
    String cardNo = "FB0DCF65";
    System.arraycopy(cardNo.getBytes(), 0, recordCondition.szCardNo, 0, cardNo.getBytes().length);

    ///CLIENT_FindRecord 入参
    NetSDKLib.NET_IN_FIND_RECORD_PARAM stuFindInParam = new
NetSDKLib.NET_IN_FIND_RECORD_PARAM();
    stuFindInParam.emType =
NetSDKLib.EM_NET_RECORD_TYPE.NET_RECORD_ACCESSCTLCARDREC_EX;
    stuFindInParam.pQueryCondition = recordCondition.getPointer();

    ///CLIENT_FindRecord 出参
    NetSDKLib.NET_OUT_FIND_RECORD_PARAM stuFindOutParam = new
NetSDKLib.NET_OUT_FIND_RECORD_PARAM();

    recordCondition.write();
    if (netsdkApi.CLIENT_FindRecord(loginHandle, stuFindInParam, stuFindOutParam, 5000)) {
        recordCondition.read();
        System.out.println("FindRecord Succeed" + "\n" + "FindHandle :" +
stuFindOutParam.lFindeHandle);

        int count = 0;   //循环的次数
        int nFindCount = 0;
        int nRecordCount = 10;   // 每次查询的个数
        ///门禁刷卡记录记录集信息
        NetSDKLib.NET_RECORDSET_ACCESS_CTL_CARDREC[] pstRecord = new
NetSDKLib.NET_RECORDSET_ACCESS_CTL_CARDREC[nRecordCount];
        for (int i = 0; i < nRecordCount; i++) {
            pstRecord[i] = new NetSDKLib.NET_RECORDSET_ACCESS_CTL_CARDREC();
        }

        ///CLIENT_FindNextRecord 入参
        NetSDKLib.NET_IN_FIND_NEXT_RECORD_PARAM stuFindNextInParam = new
NetSDKLib.NET_IN_FIND_NEXT_RECORD_PARAM();
        stuFindNextInParam.lFindeHandle = stuFindOutParam.lFindeHandle;
        stuFindNextInParam.nFileCount = nRecordCount;   //想查询的记录条数

        ///CLIENT_FindNextRecord 出参
```

```
        NetSDKLib.NET_OUT_FIND_NEXT_RECORD_PARAM stuFindNextOutParam = new
NetSDKLib.NET_OUT_FIND_NEXT_RECORD_PARAM();
        stuFindNextOutParam.nMaxRecordNum = nRecordCount;
        stuFindNextOutParam.pRecordList = new Memory(pstRecord[0].dwSize * nRecordCount);
        stuFindNextOutParam.pRecordList.clear(pstRecord[0].dwSize * nRecordCount);

        ToolKits.SetStructArrToPointerData(pstRecord, stuFindNextOutParam.pRecordList);      //将数
组内存拷贝给 Pointer 指针

        while (true) {   //循环查询
            if (netsdkApi.CLIENT_FindNextRecord(stuFindNextInParam, stuFindNextOutParam,
5000)) {
                ToolKits.GetPointerDataToStructArr(stuFindNextOutParam.pRecordList, pstRecord);

                for (int i = 0; i < stuFindNextOutParam.nRetRecordNum; i++) {
                    nFindCount = i + count * nRecordCount;

                    System.out.println("[" + nFindCount + "]刷卡时间:" +
pstRecord[i].stuTime.toStringTime());
                    System.out.println("[" + nFindCount + "]用户 ID:" + new
String(pstRecord[i].szUserID).trim());
                    System.out.println("[" + nFindCount + "]卡号:" + new
String(pstRecord[i].szCardNo).trim());
                    System.out.println("[" + nFindCount + "]门号:" + pstRecord[i].nDoor);
                    if (pstRecord[i].emDirection == 1) {
                        System.out.println("[" + nFindCount + "]开门方向: 进门");
                    } else if (pstRecord[i].emDirection == 2) {
                        System.out.println("[" + nFindCount + "]开门方向: 出门");
                    }
                }

                if (stuFindNextOutParam.nRetRecordNum < nRecordCount) {
                    break;
                } else {
                    count++;
                }
            } else {
                System.err.println("FindNextRecord Failed" + netsdkApi.CLIENT_GetLastError());
                break;
            }
        }
        netsdkApi.CLIENT_FindRecordClose(stuFindOutParam.lFindeHandle);
    } else {
        System.err.println("Can Not Find This Record" + String.format("0x%x",
netsdkApi.CLIENT_GetLastError()));
    }
}
```

2.2.11.2 设备日志

2.2.11.2.1 简介

设备日志，即用户通过调用 SDK 接口，可以对门禁设备的操作日志进行查询，查询可以指定日志类型，可以分页查询，可以指定查询数目等信息。

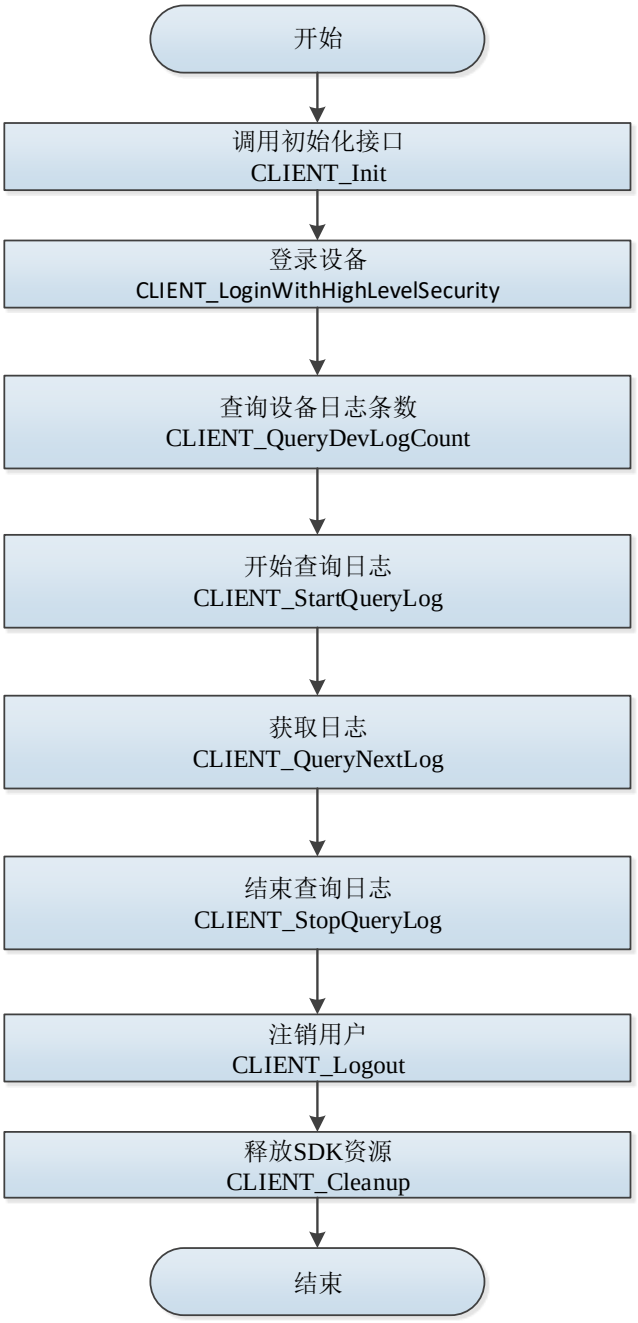
2.2.11.2.2 接口总览

表2-35 设备日志接口说明

接口	说明
CLIENT_QueryDevLogCount	查询设备日志条数。
CLIENT_StartQueryLog	开始查询日志。
CLIENT_QueryNextLog	获取日志。
CLIENT_StopQueryLog	结束查询日志。

2.2.11.2.3 流程说明

图2-22 设备日志业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_QueryDevLogCount 函数设置查询日志条数。

步骤4　调用 CLIENT_StartQueryLog 函数开始查询日志信息。

- pInParam：NET_IN_START_QUERYLOG。
- pOutParam：NET_OUT_START_QUERYLOG。

步骤5　调用 CLIENT_QueryNextLog 函数获取日志信息。

- pInParam：NET_IN_QUERYNEXTLOG。
- pOutParam：NET_OUT_QUERYNEXTLOG。

步骤6　调用 CLIENT_StopQueryLog 函数停止日志查询。

步骤7　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤8　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.2.11.2.4 示例代码

```
/**
 * 查询设备日志信息
 */
public void queryDeviceLog() {
    NET_IN_START_QUERYLOG stIn = new NET_IN_START_QUERYLOG();
    // 查询日志类型
    stIn.emLogType = 0;
    // 查询的时间段
    stIn.stuStartTime = new NET_TIME("2023/1/6/0/0/0");
    stIn.stuEndTime = new NET_TIME("2023/6/6/20/0/0");
    // 按时间升序
    stIn.emResultOrder = 1;
    stIn.write();
    NET_OUT_START_QUERYLOG stOut = new NET_OUT_START_QUERYLOG();
    stOut.write();

    LLong lLogID = netsdk.CLIENT_StartQueryLog(m_hLoginHandle, stIn.getPointer(),
stOut.getPointer(), 5000);
    stIn.read();
    stOut.read();
    if (lLogID.longValue() == 0) {
        System.err.println("CLIENT_StartQueryLog Failed!" + ToolKits.getErrorCode());
        return;
    }

    int num = 1024;//某段时间内，日志数量太多,分批查询不建议取太大值出现内存问题
    // 初始化
    NET_LOG_INFO[] info = new NET_LOG_INFO[num];
    for (int i = 0; i < info.length; i++) {
        info[i] = new NET_LOG_INFO();
    }

    NET_IN_QUERYNEXTLOG stIn1 = new NET_IN_QUERYNEXTLOG();
    // 需要查询的日志条数
    stIn1.nGetCount = num;
    stIn1.write();

    NET_OUT_QUERYNEXTLOG stOut1 = new NET_OUT_QUERYNEXTLOG();
    stOut1.nMaxCount = num;
    stOut1.pstuLogInfo = new Memory(info[0].dwSize * num);
    stOut1.pstuLogInfo.clear(info[0].dwSize * num);
    // 将 native 数据初始化
    ToolKits.SetStructArrToPointerData(info, stOut1.pstuLogInfo);
```

```
        stOut1.write();
        boolean flg1 = netsdk.CLIENT_QueryNextLog(lLogID, stIn1.getPointer(), stOut1.getPointer(), 5000);
        stIn1.read();
        stOut1.read();
        if (!flg1) {
            System.err.println("CLIENT_QueryNextLog Failed!" + ToolKits.getErrorCode());
            return;
        }
        // 实际返回日志条数
        int nRetCount = stOut1.nRetCount;
        System.out.println("实际返回日志条数: " + nRetCount);
        // 将 native 数据转为 java 数据
        ToolKits.GetPointerDataToStructArr(stOut1.pstuLogInfo, info);
        try {
            for (int i = 0; i < nRetCount; i++) {
                NET_LOG_INFO log = info[i];
                System.out.println("时间: " + log.stuTime.toStringTime()); //某段时间内，日志数量太多,分批查询时，获取批次中日志最大的时间，作为下次的开始时间
                System.out.println("操作者: " + new String(log.szUserName, encode));
                System.out.println("类型: " + new String(log.szLogType, encode));
                System.out.println("日志信息: " + new String(log.stuLogMsg.szLogMessage, encode));
            }
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        System.out.println("实际返回日志条数: " + nRetCount);
        // 手动释放内存
        long peer = Pointer.nativeValue(stOut1.pstuLogInfo);
        Native.free(peer);
        Pointer.nativeValue(stOut1.pstuLogInfo, 0);

        boolean flg2 = netsdk.CLIENT_StopQueryLog(lLogID);
        if (!flg2) {
            System.err.println("CLIENT_StopQueryLog Failed!" + ToolKits.getErrorCode());
            return;
        }
    }
```

2.2.12 门禁事件

2.2.12.1 简介

门禁设备开门时，上报开门相关的事件信息，包括事件、开门方式、开门人员对应信息等。

2.2.12.2 流程说明

本章节只介绍针对具体事件的回调处理。事件的订阅和接收流程，请参见“2.2.3 智能报警带图事件”。

2.2.12.3 枚举与结构体

- 事件对应的枚举值：EVENT_IVS_ACCESS_CTL
- 事件对应的结构体：DEV_EVENT_ACCESS_CTL_INFO

2.2.12.4 示例代码

```
private class AnalyzerDataCB implements NetSDKLib.fAnalyzerDataCallBack {
        private BufferedImage gateBufferedImage = null;

        public int invoke(LLong lAnalyzerHandle, int dwAlarmType,
                          Pointer pAlarmInfo, Pointer pBuffer, int dwBufSize,
                          Pointer dwUser, int nSequence, Pointer reserved)
        {
            if (lAnalyzerHandle.longValue() == 0 || pAlarmInfo == null) {
                return -1;
            }

            File path = new File("./GateSnapPicture/");
            if (!path.exists()) {
                path.mkdir();
            }

            ///< 门禁事件
            if(dwAlarmType == NetSDKLib.EVENT_IVS_ACCESS_CTL) {
                DEV_EVENT_ACCESS_CTL_INFO msg = new DEV_EVENT_ACCESS_CTL_INFO();
                ToolKits.GetPointerData(pAlarmInfo, msg);

                // 保存图片，获取图片缓存
                String snapPicPath = path + "\\" + System.currentTimeMillis() + "GateSnapPicture.jpg";  // 保存图片地址
                byte[] buffer = pBuffer.getByteArray(0, dwBufSize);
            ByteArrayInputStream byteArrInputGlobal = new ByteArrayInputStream(buffer);

                try {
                    gateBufferedImage = ImageIO.read(byteArrInputGlobal);
```

```
                    if(gateBufferedImage != null) {
                        ImageIO.write(gateBufferedImage, "jpg", new File(snapPicPath));
                    }
                } catch (IOException e2) {
                    e2.printStackTrace();
                }

                // 图片以及门禁信息界面显示
                EventQueue eventQueue = Toolkit.getDefaultToolkit().getSystemEventQueue();
                if (eventQueue != null) {
                eventQueue.postEvent( new AccessEvent(target,gateBufferedImage,msg));
                }
            }

            return 0;
        }
```

2.2.13 人证比对事件

2.2.13.1 简介

将检测到的人员信息与证件信息比较，检测人员是否匹配。

2.2.13.2 流程说明

本章节只介绍针对具体事件的回调处理。事件的订阅和接收流程，请参见“2.2.3 智能报警带图事件”。

2.2.13.3 枚举与结构体

- 事件对应的枚举值：EVENT_IVS_CITIZEN_PICTURE_COMPARE
- 事件对应的结构体：DEV_EVENT_CITIZEN_PICTURE_COMPARE_INFO

2.2.13.4 示例代码

```
/* 智能报警事件回调 */
  public static class fAnalyzerDataCB implements NetSDKLib.fAnalyzerDataCallBack {
      private BufferedImage snapBufferedImage = null;
      private BufferedImage idBufferedImage = null;

      private fAnalyzerDataCB() {}
```

```
private static class fAnalyzerDataCBHolder {
    private static final fAnalyzerDataCB instance = new fAnalyzerDataCB();
}
public static fAnalyzerDataCB getInstance() {
    return fAnalyzerDataCBHolder.instance;
}

@Override
public int invoke(LLong lAnalyzerHandle, int dwAlarmType,
        Pointer pAlarmInfo, Pointer pBuffer, int dwBufSize,
        Pointer dwUser, int nSequence, Pointer reserved) {
    if(pAlarmInfo == null) {
        return 0;
    }

    File path = new File("./CitizenCompare/");
    if (!path.exists()) {
        path.mkdir();
    }

    switch(dwAlarmType)
    {
        case NetSDKLib.EVENT_IVS_CITIZEN_PICTURE_COMPARE:
        //人证比对事件
        {
            DEV_EVENT_CITIZEN_PICTURE_COMPARE_INFO msg = new
            DEV_EVENT_CITIZEN_PICTURE_COMPARE_INFO();
            ToolKits.GetPointerData(pAlarmInfo, msg);

            try {
                System.out.println("事件发生时间：" + msg.stuUTC.toString());
                System.out.println("事件名称:" + new String(msg.szName,
                "GBK").trim());

                // 人证比对结果,相似度大于等于阈值认为比对成功，1-表示成功，0-
                表示失败
                System.out.println("比对结果:" + msg.bCompareResult);

                System.out.println("两张图片的相似度:" + msg.nSimilarity);
```

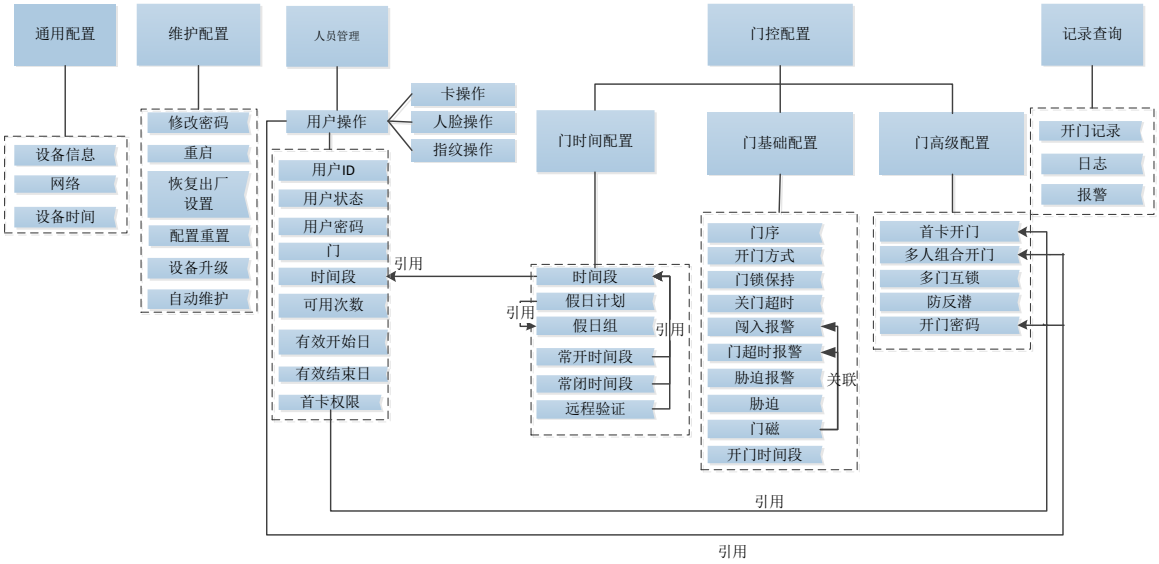
```
                System.out.println("检测阈值:" + msg.nThreshold);

                if (msg.emSex == 1) {
                    System.out.println("性别:男");
                }else if (msg.emSex == 2){
                    System.out.println("性别:女");
                }else {
                    System.out.println("性别:未知");
                }



                System.out.println("居民姓名:" + new String(msg.szCitizen,
                "GBK").trim());
                System.out.println("住址:" + new String(msg.szAddress,
                "GBK").trim());
                System.out.println("身份证号:" + new String(msg.szNumber).trim());
                System.out.println("签发机关:" + new String(msg.szAuthority,
                "GBK").trim());

                System.out.println("出生日期:" + msg.stuBirth.toStringTimeEx());
                System.out.println("有效起始日期:" +
                msg.stuValidityStart.toStringTimeEx());
                if (msg.bLongTimeValidFlag == 1) {
                    System.out.println("有效截止日期：永久");
                }else{
                    System.out.println("有效截止日期:"+
                msg.stuValidityEnd.toStringTimeEx());
                }
                System.out.println("IC卡号：" + new String(msg.szCardNo,
                "GBK").trim());
            } catch (Exception e) {
                e.printStackTrace();
            }

            // 拍摄照片
                String strFileName = path + "\\" + System.currentTimeMillis() +
            "citizen_snap.jpg";
                    byte[] snapBuffer =
             pBuffer.getByteArray(msg.stuImageInfo[0].dwOffSet,
             msg.stuImageInfo[0].dwFileLenth);
```

```
                    ByteArrayInputStream snapArrayInputStream = new
                ByteArrayInputStream(snapBuffer);
                try {
                    snapBufferedImage = ImageIO.read(snapArrayInputStream);
                    if(snapBufferedImage == null) {
                        return 0;
                    }
                    ImageIO.write(snapBufferedImage, "jpg", new File(strFileName));

                } catch (IOException e) {
                    e.printStackTrace();
                }

                // 身份证照片
                strFileName = path + "\\" + System.currentTimeMillis() + "citizen_id.jpg";
                    byte[] idBuffer =
                pBuffer.getByteArray(msg.stuImageInfo[1].dwOffSet,
                msg.stuImageInfo[1].dwFileLenth);
                    ByteArrayInputStream idArrayInputStream = new
                ByteArrayInputStream(idBuffer);
                try {
                    idBufferedImage = ImageIO.read(idArrayInputStream);
                    if(idBufferedImage == null) {
                        return 0;
                    }
                    ImageIO.write(idBufferedImage, "jpg", new File(strFileName));
                } catch (IOException e) {
                    e.printStackTrace();
                }

                    break;
                }
                default:
                    break;
            }

            return 0;
        }
    }
```

2.3 门禁控制器/人脸一体机（二代）

门禁控制器/人脸一体机（二代）功能调用关系

图2-23 功能调用关系



功能调用关系中引用和关联的含义如下：

- 引用：箭头终点指向的功能引用箭头起点指向的功能。
- 关联：箭头起始的功能是否能够正常使用，与箭头终点指向的功能配置相关。

2.3.1 门禁控制

请参见“2.2.1 门禁控制”。

2.3.2 报警事件

请参见“2.2.2 报警事件”。

2.3.3 智能报警带图事件

请参见“2.2.3 智能报警带图事件”。

2.3.4 设备信息查看

2.3.4.1 能力集查询

2.3.4.1.1 简介

设备信息查看，即用户通过 SDK 下发命令给门禁设备，来获取设备的能力集。

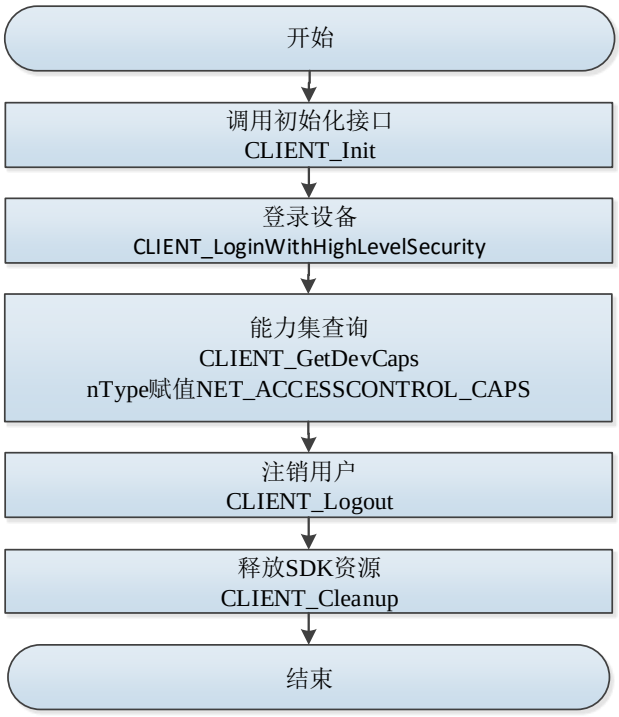
2.3.4.1.2 接口总览

表2-36 能力集查询接口说明

接口	说明
CLIENT_GetDevCaps	获取门禁能力（包括门禁、用户、卡、人脸、指纹能力）。

2.3.4.1.3 流程说明

图2-24 设备信息查看流程图



流程说明

步骤9　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤10　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤11　调用 CLIENT_GetDevCaps 函数，nType 赋值为 NET_ACCESSCONTROL_CAPS，可以获取门禁能力。

步骤12　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤13　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.4.1.4 示例代码

```
    public void    accesscontrolCaps(){
//      获取门禁能力, pInBuf = NET_IN_AC_CAPS*, pOutBuf = NET_OUT_AC_CAPS*
        int nType = NetSDKLib.NET_ACCESSCONTROL_CAPS;


        // 入参
        NET_IN_AC_CAPS stIn = new NET_IN_AC_CAPS();
```

```
        // 出参
        NET_OUT_AC_CAPS stOut = new NET_OUT_AC_CAPS();

        stIn.write();
        stOut.write();

        boolean bRet = netsdk.CLIENT_GetDevCaps(m_hLoginHandle, nType, stIn.getPointer(),
stOut.getPointer(), 3000);
        if(bRet) {
            stOut.read();
            /**
             ACCaps 能力集
             */
            NET_AC_CAPS stuACCaps
                    = stOut.stuACCaps;
            System.out.println("ACCaps 能力集");

            System.out.println("nChannels:"+stuACCaps.nChannels);
            /**
             是否支持门禁报警日志记录在记录集中
             */
            int bSupAccessControlAlarmRecord
                    = stuACCaps.bSupAccessControlAlarmRecord;
            System.out.println("bSupAccessControlAlarmRecord:"+bSupAccessControlAlarmRecord);

            /**
             AccessControlCustomPassword 记录集中密码的保存方式,0:明文,默认值 0, 1:MD5
             */
            int nCustomPasswordEncryption
                    = stuACCaps.nCustomPasswordEncryption;
            System.out.println("nCustomPasswordEncryption:"+nCustomPasswordEncryption);
            /**
             是否支持信息功能,0:未知，默认,1:不支持, 2:支持
             */
            int nSupportFingerPrint
                    = stuACCaps.nSupportFingerPrint;
            System.out.println("nSupportFingerPrint:"+nSupportFingerPrint);

            /**
             user 操作能力集
```

```
 */
System.out.println("user 操作能力集");

NET_ACCESS_USER_CAPS stuUserCaps=   stOut.stuUserCaps;
/**
 每次下发的最大数量
 */
int nMaxInsertRate = stuUserCaps.nMaxInsertRate;
System.out.println("nMaxInsertRate:"+nMaxInsertRate);

/**
 用户数量上限
 */
int nMaxUsers
        = stuUserCaps.nMaxUsers;
System.out.println("nMaxUsers:"+nMaxUsers);
/**
 每个用户可以记录的最大信息数量
 */
int nMaxFingerPrintsPerUser
        = stuUserCaps.nMaxFingerPrintsPerUser;
System.out.println("nMaxFingerPrintsPerUser:"+nMaxFingerPrintsPerUser);
/**
 每个用户可以记录的最大卡片数量
 */
int nMaxCardsPerUser
        = stuUserCaps.nMaxCardsPerUser;
System.out.println("nMaxCardsPerUser:"+nMaxCardsPerUser);

/**
 card 操作能力集
 */
NET_ACCESS_CARD_CAPS stuCardCaps
        = stOut.stuCardCaps;
System.out.println("card 操作能力集");

/**
 每次下发的最大数量
 */
int nMaxInsertRate1
```

```
            = stuCardCaps.nMaxInsertRate;
System.out.println("nMaxInsertRate1:"+nMaxInsertRate1);

/**
 卡片数量上限
 */
int nMaxCards
            = stuCardCaps.nMaxCards;
System.out.println("nMaxCards:"+nMaxCards);


/**
 信息操作能力集
 */
NET_ACCESS_FINGERPRINT_CAPS stuFingerprintCaps
            = stOut.stuFingerprintCaps;

System.out.println("信息操作能力集");

/**
 每次下发的最大数量
 */
int nMaxInsertRate2
            = stuFingerprintCaps.nMaxInsertRate;
System.out.println("nMaxInsertRate2:"+nMaxInsertRate2);

/**
 单信息数据的最大字节数
 */
int nMaxFingerprintSize = stuFingerprintCaps.nMaxFingerprintSize;
System.out.println("nMaxFingerprintSize:"+nMaxFingerprintSize);

/**
 信息数量上限
 */
int nMaxFingerprint = stuFingerprintCaps.nMaxFingerprint;

System.out.println("nMaxFingerprint:"+nMaxFingerprint);
```

```
/**
 目标操作能力集
 */
NET_ACCESS_FACE_CAPS stuFaceCaps
        = stOut.stuFaceCaps;

System.out.println("目标操作能力集");
/**
 每次下发的最大数量
 */
int nMaxInsertRate3
        = stuFaceCaps.nMaxInsertRate;
System.out.println("nMaxInsertRate3:"+nMaxInsertRate3);

/**
 目标存储上限
 */
int nMaxFace
        = stuFaceCaps.nMaxFace;
System.out.println("nMaxFace:"+nMaxFace);

/**
 目标识别类型，0:白光 1:红外
 */
int nRecognitionType
        = stuFaceCaps.nRecognitionType;
System.out.println("nRecognitionType:"+nRecognitionType);
/**
 目标识别算法，0:未知 1:华 2:商 3:依 4:汉 5:火
 */
int nRecognitionAlgorithm
        = stuFaceCaps.nRecognitionAlgorithm;
System.out.println("nRecognitionAlgorithm:"+nRecognitionAlgorithm);

/**
 眼睛相关能力集
 */
NET_ACCESS_IRIS_CAPS stuIrisCaps
        = stOut.stuIrisCaps;
```

```
System.out.println("眼睛相关能力集");

/**
 每次最大插入量
 */
int nMaxInsertRate4
        = stuIrisCaps.nMaxInsertRate;
System.out.println("nMaxInsertRate4:"+nMaxInsertRate4);

/**
 眼睛信息图片最小尺寸,单位 KB
 */
int nMinIrisPhotoSize
        = stuIrisCaps.nMinIrisPhotoSize;
System.out.println("nMinIrisPhotoSize:"+nMinIrisPhotoSize);

/**
 眼睛信息图片最大尺寸，单位 KB
 */
int nMaxIrisPhotoSize
        = stuIrisCaps.nMaxIrisPhotoSize;
System.out.println("nMaxIrisPhotoSize:"+nMaxIrisPhotoSize);

/**
 每个用户最多支持多少组
 */
int nMaxIrisGroup
        = stuIrisCaps.nMaxIrisGroup;
System.out.println("nMaxIrisGroup:"+nMaxIrisGroup);

/**
 眼睛识别算法提供标识，0 未知，1 华
 */
int nRecognitionAlgorithmVender
        = stuIrisCaps.nRecognitionAlgorithmVender;
System.out.println("nRecognitionAlgorithmVender:"+nRecognitionAlgorithmVender);

/**
 算法(模型)版本号,如果版本号有多位，按 Major/Minor 从高到低每 8bit 表示一个版本 如
1.5.2 表示成 0x00010502
```

```
             */
            int nRecognitionVersion
                        = stuIrisCaps.nRecognitionVersion;
            System.out.println("nRecognitionVersion:"+nRecognitionVersion);

            /**
             眼睛信息存储上限
             */
            int nMaxIrisesCount
                        = stuIrisCaps.nMaxIrisesCount;
            System.out.println("nMaxIrisesCount:"+nMaxIrisesCount);



        } else {
            System.err.println("GetDevCaps Failed!" + ToolKits.getErrorCode());
        }
    }
```

2.3.4.2 设备版本、MAC 查看

请参见“2.2.4.2 设备版本、MAC 查看”。

2.3.5 网络配置

请参见“2.2.5 网络配置”。

2.3.6 设备时间设置

请参见“2.2.6 设备时间设置”。

2.3.7 人员管理

2.3.7.1 用户管理

2.3.7.1.1 简介

用户通过调用 SDK，可以对门禁设备的用户信息字段（包含：用户 ID、人员名称、类型、状态、身份证号码、有效时间段、假日计划、权限等）进行增删查的操作。

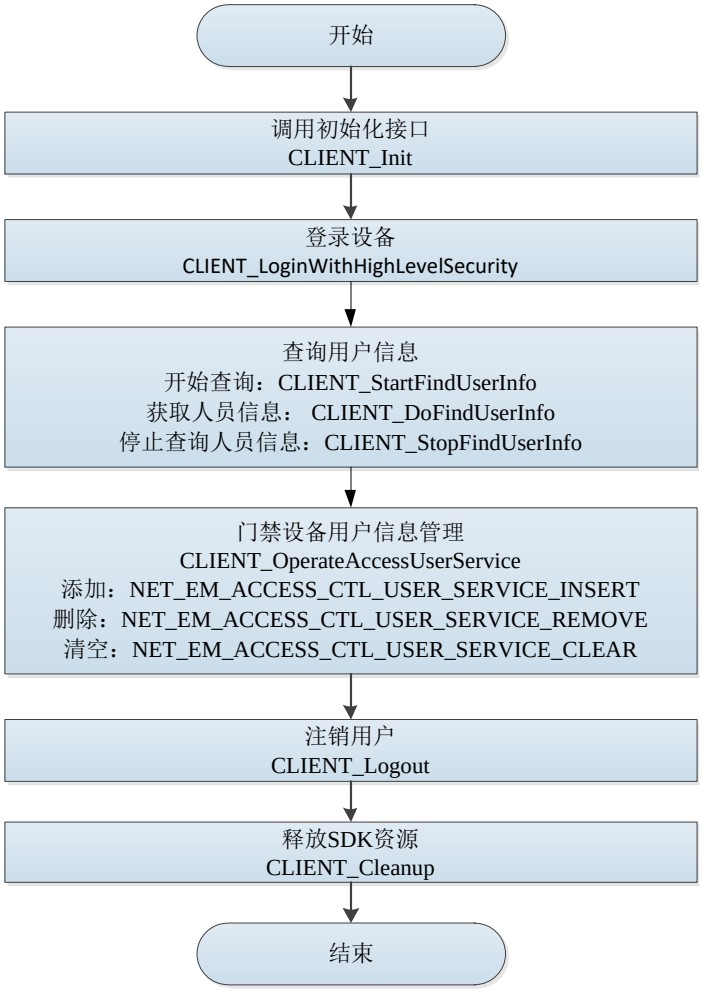
2.3.7.1.2 接口总览

表2-37 用户信息接口说明

接口	说明
CLIENT_OperateAccessUserService	门禁用户信息管理接口
CLIENT_StartFindUserInfo	开始查询用户信息
CLIENT_DoFindUserInfo	获取用户信息
CLIENT_StopFindUserInfo	停止查询用户信息

2.3.7.1.3 流程说明

图2-25 用户信息管理业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。
步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。
步骤3　调用 CLIENT_StartFindUserInfo 开始查询用户信息。
步骤4　调用 CLIENT_DoFindUserInfo 获取用户信息。
步骤5　调用 CLIENT_StopFindUserInfo 停止查询用户信息。
步骤6　调用 CLIENT_OperateAccessUserService 函数来对用户信息进行添加、删除、清空操

作。

步骤7　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤8　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.7.1.4 示例代码

```
/**
 * 根据用户 ID 获取用户信息
 */
public void operateAccessUserService(){

    String[] userIDs = {"3"};

    // 获取的用户个数
    int nMaxNum = userIDs.length;

    // ////////////////////////// 以下固定写法
    // //////////////////////////////////
    // 用户操作类型
    // 获取用户
    int emtype =
NetSDKLib.NET_EM_ACCESS_CTL_USER_SERVICE.NET_EM_ACCESS_CTL_USER_SERVICE_G
ET;

    /**
     * 用户信息数组
     */
    // 先初始化用户信息数组
    NetSDKLib.NET_ACCESS_USER_INFO[] users = new
NetSDKLib.NET_ACCESS_USER_INFO[nMaxNum];


    // 初始化返回的失败信息数组
    NetSDKLib.FAIL_CODE[] failCodes = new NetSDKLib.FAIL_CODE[nMaxNum];

    for (int i = 0; i < nMaxNum; i++) {
        NetSDKLib.NET_ACCESS_USER_INFO info      = new
NetSDKLib.NET_ACCESS_USER_INFO();
        int size
                = new NET_FLOORS_INFO().size();

        Pointer floors =new Memory(size);

        floors.clear(size);

        info.pstuFloorsEx2=floors;

        NET_ACCESS_USER_INFO_EX pstuUserInfoEx=
```

```
                    new NET_ACCESS_USER_INFO_EX();

            Pointer pstuUserInfo
                    = new Memory(pstuUserInfoEx.size());

            pstuUserInfo.clear(pstuUserInfoEx.size());

            info.pstuUserInfoEx=pstuUserInfo;

            users[i]=info;

            failCodes[i] = new NetSDKLib.FAIL_CODE();
        }

        /**
         * 入参 NET_IN_ACCESS_USER_SERVICE_GET
         */
        NetSDKLib.NET_IN_ACCESS_USER_SERVICE_GET stIn = new
NetSDKLib.NET_IN_ACCESS_USER_SERVICE_GET();
        // 用户 ID 个数
        stIn.nUserNum = userIDs.length;

        // 用户 ID
        for (int i = 0; i < userIDs.length; i++) {
            System.arraycopy(userIDs[i].getBytes(), 0,
                    stIn.szUserIDs[i].szUserID, 0, userIDs[i].getBytes().length);
        }

        /**
         * 出参   NET_OUT_ACCESS_USER_SERVICE_GET
         */
        NetSDKLib.NET_OUT_ACCESS_USER_SERVICE_GET stOut = new
NetSDKLib.NET_OUT_ACCESS_USER_SERVICE_GET();

        stOut.nMaxRetNum = nMaxNum;

        stOut.pUserInfo = new Memory(users[0].size() * nMaxNum); // 申请内存
        stOut.pUserInfo.clear(users[0].size() * nMaxNum);

        stOut.pFailCode = new Memory(failCodes[0].size() * nMaxNum); // 申请内存
        stOut.pFailCode.clear(failCodes[0].size() * nMaxNum);

        ToolKits.SetStructArrToPointerData(users, stOut.pUserInfo);


        ToolKits.SetStructArrToPointerData(failCodes, stOut.pFailCode);
```

```
        stIn.write();
        stOut.write();


        if (netSdk.CLIENT_OperateAccessUserService(loginHandle, emtype,
                stIn.getPointer(), stOut.getPointer(), 3000)) {
            // 将指针转为具体的信息
            ToolKits.GetPointerDataToStructArr(stOut.pUserInfo, users);
            ToolKits.GetPointerDataToStructArr(stOut.pFailCode, failCodes);

            /**
             * 打印具体的信息
             */
            for (int i = 0; i < nMaxNum; i++) {
                try {
                    System.out.println("[" + i + "]用户名："
                            + new String(users[i].szName, "GBK").trim());
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                System.out.println("[" + i + "]密码："
                        + new String(users[i].szPsw).trim());
                System.out.println("[" + i + "]查询用户结果："
                        + failCodes[i].nFailCode);
            }
        } else {
            System.err.println("查询用户失败, " + ToolKits.getErrorCode());
        }
    }
```

2.3.7.2 卡片管理

2.3.7.2.1 简介

用户通过调用 SDK，可以对门禁设备的卡片信息字段（包含：卡号、用户 ID、卡类型等）进行增删查改的操作。

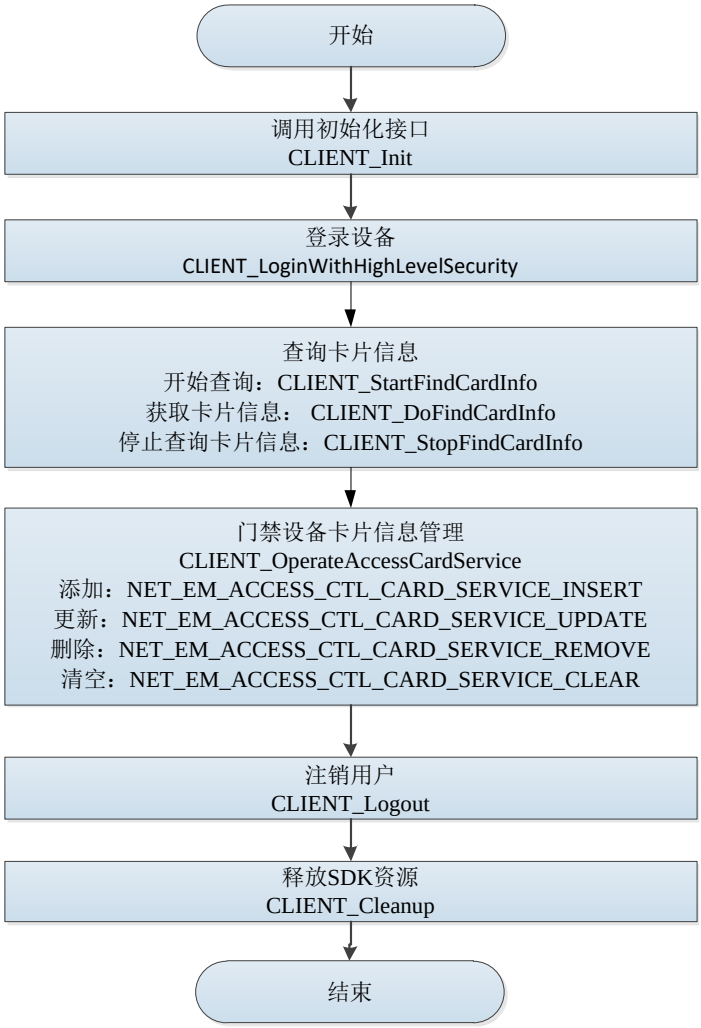
2.3.7.2.2 接口总览

表2-38 卡片信息接口说明

接口	说明
CLIENT_OperateAccessCardService	门禁卡片信息管理接口
CLIENT_StartFindCardInfo	开始查询卡片信息
CLIENT_DoFindCardInfo	获取卡片信息
CLIENT_StopFindCardInfo	停止查询卡片信息

2.3.7.2.3 流程说明

图2-26 卡片信息管理业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_StartFindCardInfo 开始查询卡片信息。

步骤4　调用 CLIENT_DoFindCardInfo 获取卡片信息。

步骤5　调用 CLIENT_StopFindCardInfo 停止查询卡片信息。

步骤6　调用 CLIENT_OperateAccessCardService 函数来对卡片信息进行添加、更新、删除、清空操作。

步骤7　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤8　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.7.2.4 示例代码

```
/**
 * 查询所有卡信息
 */
```

```
public void queryAllCard() {
//          String userId = "1011";
        // ////////////////////////////////////////////////////////////////////////////////////////////
        /**
         * 入参
         */
        NET_IN_CARDINFO_START_FIND stInFind = new NET_IN_CARDINFO_START_FIND();
        // 用户 ID, 为空或者不填，查询所有用户的所有卡
//          System.arraycopy(userId.getBytes(), 0, stInFind.szUserID, 0,
//                  userId.getBytes().length);

        /**
         * 出参
         */
        NET_OUT_CARDINFO_START_FIND stOutFind = new NET_OUT_CARDINFO_START_FIND();

        LLong lFindHandle = netsdkApi.CLIENT_StartFindCardInfo(loginHandle,
                stInFind, stOutFind, TIME_OUT);

        if (lFindHandle.longValue() == 0) {
            return;
        }

        System.out.println("符合查询条件的总数:" + stOutFind.nTotalCount);

        if (stOutFind.nTotalCount <= 0) {
            return;
        }
        // ////////////////////////////////////////////////////////////////////////////////////////////
        // 起始序号
        int startNo = 0;

        // 每次查询的个数
        int nFindCount = stOutFind.nCapNum == 0 ? 5 : stOutFind.nCapNum;

        while (true) {

            NET_ACCESS_CARD_INFO[] cardInfos = new NET_ACCESS_CARD_INFO[nFindCount];
            for (int i = 0; i < nFindCount; i++) {
                cardInfos[i] = new NET_ACCESS_CARD_INFO();
                cardInfos[i].bUserIDEx = 1;
            }

            /**
             * 入参
             */
            NET_IN_CARDINFO_DO_FIND stInDoFind = new NET_IN_CARDINFO_DO_FIND();
```

```
        // 起始序号
        stInDoFind.nStartNo = startNo;

        // 本次查询的条数
        stInDoFind.nCount = nFindCount;

        /**
         * 出参
         */
        NET_OUT_CARDINFO_DO_FIND stOutDoFind = new NET_OUT_CARDINFO_DO_FIND();
        stOutDoFind.nMaxNum = nFindCount;

        stOutDoFind.pstuInfo = new Memory(cardInfos[0].size() * nFindCount);
        stOutDoFind.pstuInfo.clear(cardInfos[0].size() * nFindCount);

        ToolKits.SetStructArrToPointerData(cardInfos, stOutDoFind.pstuInfo);

        if (netsdkApi.CLIENT_DoFindCardInfo(lFindHandle, stInDoFind,
                stOutDoFind, TIME_OUT)) {
            if (stOutDoFind.nRetNum <= 0) {
                return;
            }

            ToolKits.GetPointerDataToStructArr(stOutDoFind.pstuInfo,
                    cardInfos);

            for (int i = 0; i < stOutDoFind.nRetNum; i++) {
                System.out.println("[" + (startNo + i) + "]用户 ID："
                        + new String(cardInfos[i].szUserID).trim());
                System.out.println("[" + (startNo + i) + "]卡号："
                        + new String(cardInfos[i].szCardNo).trim());
                System.out.println("[" + (startNo + i) + "]卡类型："
                        + cardInfos[i].emType + "\n");
            }
        }

        if (stOutDoFind.nRetNum < nFindCount) {
            break;
        } else {
            startNo += nFindCount;
        }
    }

    // ////////////////////////////////////////////////////////////////////////////////////////
    // 停止查找
    if (lFindHandle.longValue() != 0) {
        netsdkApi.CLIENT_StopFindCardInfo(lFindHandle);
```

```
            lFindHandle.setValue(0);
        }
    }
```

2.3.7.3 人脸管理

2.3.7.3.1 简介

用户通过调用 SDK，可以对门禁设备的人脸信息字段（包含：用户 ID、人脸图片数据等）进行增删查改的操作。

2.3.7.3.2 接口总览

表2-39 人脸信息接口说明

接口	说明
CLIENT_OperateAccessFaceService	门禁人脸信息管理接口

2.3.7.3.3 流程说明

图2-27 人脸信息管理业务流程

流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_OperateAccessFaceService 函数来对人脸信息进行添加、获取、更新、删除操作。

步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.7.3.4 示例代码

```
/**
 * 获取人脸信息
 */
public void getFace() {
    String[] userIDs = { "1","2","3" };
//    String[] userIDs = { "3423" };
    // 获取人脸的用户最大个数
    int nMaxCount = userIDs.length;

    // ////////////////////// 每个用户的人脸信息初始化 //////////////////////
    NetSDKLib.NET_ACCESS_FACE_INFO[] faces = new
NetSDKLib.NET_ACCESS_FACE_INFO[nMaxCount];
    for (int i = 0; i < faces.length; i++) {
        faces[i] = new NetSDKLib.NET_ACCESS_FACE_INFO();

        // 根据每个用户的人脸图片的实际个数申请内存，最多 5 张照片

        faces[i].nFacePhoto = 1; // 每个用户图片个数

        // 对每张照片申请内存
        faces[i].nInFacePhotoLen[0] = 200 * 1024;
        faces[i].pFacePhotos[0].pFacePhoto = new Memory(200 * 1024); // 人脸照片数据,大小不超过
200K
        faces[i].pFacePhotos[0].pFacePhoto.clear(200 * 1024);
    }

    // 初始化
    NetSDKLib.FAIL_CODE[] failCodes = new NetSDKLib.FAIL_CODE[nMaxCount];
    for (int i = 0; i < failCodes.length; i++) {
        failCodes[i] = new NetSDKLib.FAIL_CODE();
    }

    // 人脸操作类型
    // 获取人脸信息
    int emtype =
NetSDKLib.NET_EM_ACCESS_CTL_FACE_SERVICE.NET_EM_ACCESS_CTL_FACE_SERVICE_G
ET;

    /**
     * 入参
     */
```

```
    NetSDKLib.NET_IN_ACCESS_FACE_SERVICE_GET stIn = new
NetSDKLib.NET_IN_ACCESS_FACE_SERVICE_GET();
    stIn.nUserNum = nMaxCount;
    for (int i = 0; i < nMaxCount; i++) {
        System.arraycopy(userIDs[i].getBytes(), 0,
                stIn.szUserIDs[i].szUserID, 0, userIDs[i].getBytes().length);
    }

    /**
     * 出参 NET_OUT_ACCESS_FACE_SERVICE_GET
     */


    NetSDKLib.NET_OUT_ACCESS_FACE_SERVICE_GET stOut = new
NetSDKLib.NET_OUT_ACCESS_FACE_SERVICE_GET();
    stOut.nMaxRetNum = nMaxCount;

    stOut.pFaceInfo = new Memory(faces[0].size() * nMaxCount);
    stOut.pFaceInfo.clear(faces[0].size() * nMaxCount);

    stOut.pFailCode = new Memory(failCodes[0].size() * nMaxCount);
    stOut.pFailCode.clear(failCodes[0].size() * nMaxCount);

    ToolKits.SetStructArrToPointerData(faces, stOut.pFaceInfo);
    ToolKits.SetStructArrToPointerData(failCodes, stOut.pFailCode);

    stIn.write();
    stOut.write();
    if (netSdk.CLIENT_OperateAccessFaceService(loginHandle, emtype,
            stIn.getPointer(), stOut.getPointer(), 3000)) {
        // 将获取到的结果信息转成具体的结构体
        ToolKits.GetPointerDataToStructArr(stOut.pFaceInfo, faces);
        ToolKits.GetPointerDataToStructArr(stOut.pFailCode, failCodes);

        // 打印具体信息
        // nMaxCount 几个用户
        for (int i = 0; i < nMaxCount; i++) {
            System.out.println("[" + i + "]用户 ID : "
                    + new String(faces[i].szUserID).trim());

            int nFacePhoto = faces[i].nFacePhoto;
            System.out.println("nFacePhoto:"+nFacePhoto);


            int nFaceData = faces[i].nFaceData;
            System.out.println("nFaceData:"+nFaceData);
            NetSDKLib.FACEDATA[] szFaceDatas = faces[i].szFaceDatas;
```

```
                for(int n=0;n<nFaceData;n++){
                    NetSDKLib.FACEDATA szFaceData = szFaceDatas[i];

                    byte[] szFaceData1 = szFaceData.szFaceData;
                   new DefaultAnalyseTaskResultCallBack().fileOut(szFaceData1);
                }


                System.out.println("[" + i + "]获取人脸结果 : "
                        + failCodes[i].nFailCode);
            }
        } else {
            System.err.println("获取人脸失败, " + ToolKits.getErrorCode());
        }

        stIn.read();
        stOut.read();
    }
```

2.3.7.4 指纹管理

2.3.7.4.1 简介

用户通过调用 SDK，可以对门禁设备的指纹信息字段（包含：用户 ID、指纹数据包、胁迫指纹序号等）进行增删查改的操作。

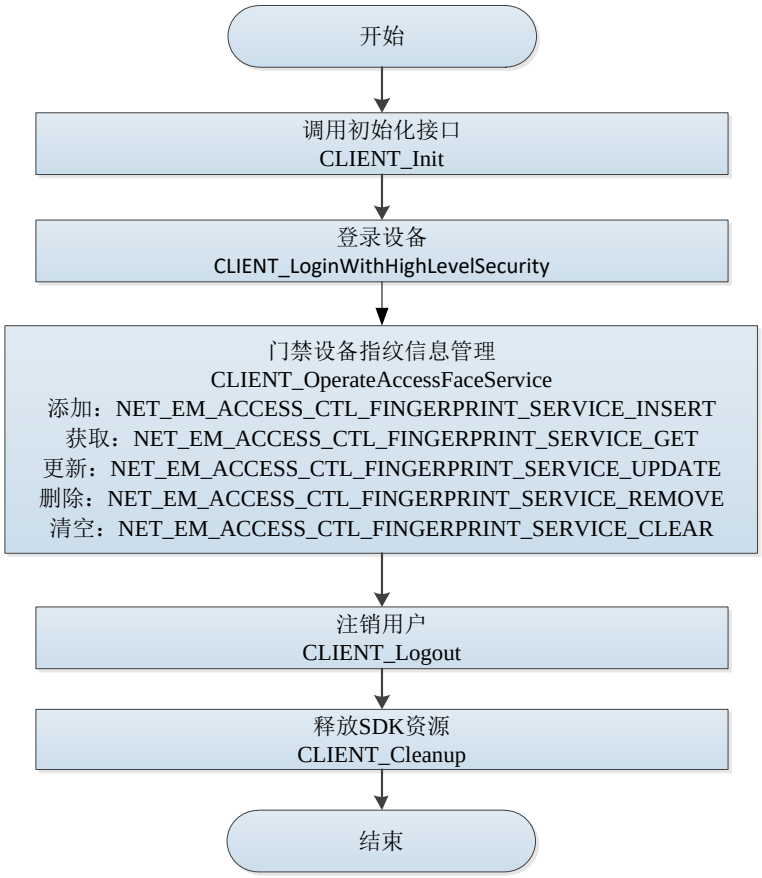
2.3.7.4.2 接口总览

表2-40 指纹信息接口说明

接口	说明
CLIENT_OperateAccessFingerprintService	指纹信息管理接口

2.3.7.4.3 流程说明

图2-28 指纹信息管理业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_OperateAccessFingerprintService 函数来对指纹信息进行添加、获取、更新、删除、清空操作。

步骤4　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤5　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.7.4.4 示例代码

```
/************************************************************************************************
 * 信息操作：添加/修改/删除/获取/清空 一次只能针对一个用户添加/修改/删除信息操作，并只能对应一个信息， 每个用户最多添加 3 个信息
 *
 * 如果需要给操作一个用户的信息，需要循环操作 例： 接口调用： 用户 ID：1011 + 一个信息数据 再次接口调用 用户 ID：1011 +
 * 一个信息数据
 ************************************************************************************************/
/**
```

```
 * 根据用户 ID 添加信息(一个用户 ID 对应一个信息)
 */
public void addFingerprint() {
    String userId = "1011";

    String finger = "xTxpAASFsk3+8hoDh4ky0ghH2oR7hjp658Wp"
            + "Q4eJQyEQBdoFgIerDuhFuAQGhsr6BvQpguuF804HMjpFaIUjOQ"
            + "AJyUSGiAwICEupSGSFnA3gS7nF2IUUVe62SYVXhcSBoJG5iGSGvJqI"
            + "lZsRkYhksjBNnUmjh4VOEFEpizyGVUnol4oQQohVft6sOQdWh93lzarniBeJK"
            +
"4EANhtCKoqEmuA0C4M+im2+/zPJAzCKrcX5OipCtIpF9g6DqQOth32CAA8bUiiGdbog0wqKWofGYtx
wecgQhsZi"
            +
"ONMO3VqH5p7b8kmGiooaSQfF/MEWilKZ+Dgeg4iJuwYQBgyDJ4uEMfg3/kOsi53VB/++RTaLbf74N/u
FY4NzPegFyQNfg3Ot6EXawda"
            +
"CO63/fhpAWIS0YehJyoRSg9SR+IfYRFCCbQXnxcrBR4RtTeCJykPKgrU5/rYagr6EvbIOeDhCOoWt5d
B/yI"
            +
"dDgd3FB8f4BEGDpf3oR/oDQYImAQfJ+kM+iy5x+HXMRUaJznnQtCrGNoteifk4CkUohZaWyIepBTGEd
sLX/9qBHoZO0pD"
            +
"JGog6iybq+Tn3x1yH7vrbco2HN4R+8s//+EIhhbcW18f3RZqGnx4LfhvXO4qvOvlB+0qQhpc9/L5MVlJH8
78zUyMvjxF2YSIlVRJUamE"
            +
"lNH8nRUGD8niPUp/xI6f28/czRPMPMfH2QvIvQSNEcyNB/zIyRSM/8zXzUvMzYfNFQ/IzUh9CNSf1X//x
L0U6MohGebIWlDNWhRZi/xIVgkVP+Hhn"
            +
"9f9RAUWhRi8eafk+QXSytDwtFjGCklxSMTK8pNZYFILYkrkkYmS30qhUEcKqtNswRwKFkq8QQSHHk
RBtESNRkfhdEbK9ovsllVKMoYYdMtGdgSM0"
            + "ERE1cnIOQQTYoLJkIQJ0gJsNFFTugetWEhME5zESEnLokpsbFFS+kYVfVLGbs"
            + "E4qQTaYgcEOIRPEkfQRJGDhkWsOEmXWoLdpRKKNoA5tRENBoM"
            +
"5GEFEkosISZBLUkKlXIHbdkGRRAQHRoAwLDw4NFwgJChglKxEoKiYaGQcSBDIbIy0pNBQWFVVs";

    // 初始化信息数据
    NET_ACCESS_FINGERPRINT_INFO fingerprint = new NET_ACCESS_FINGERPRINT_INFO();

    /**
     * 添加信息数据
     */
    // 用户 ID
```

```
    System.arraycopy(userId.getBytes(), 0, fingerprint.szUserID, 0,
            userId.getBytes().length);

    // 将字符串转为信息数据
    byte[] fingerPrintBuffer = Base64Util.getDecoder().decode(finger);

    // 单个信息长度
    fingerprint.nPacketLen = fingerPrintBuffer.length;

    // 信息包个数
    fingerprint.nPacketNum = 1;

    // 信息数据
    // 先申请内存
    fingerprint.szFingerPrintInfo = new Memory(fingerPrintBuffer.length);
    fingerprint.szFingerPrintInfo.clear(fingerPrintBuffer.length);

    fingerprint.szFingerPrintInfo.write(0, fingerPrintBuffer, 0,
            fingerPrintBuffer.length);

    // ////////////////////////////////////////////////////////////////////////////////

    // 初始化
    FAIL_CODE failCode = new FAIL_CODE();

    // 信息操作类型
    // 添加信息信息
    int emtype =
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE.NET_EM_ACCESS_CTL_FINGERPRINT_SERVI
CE_INSERT;

    /**
     * 入参
     */
    NET_IN_ACCESS_FINGERPRINT_SERVICE_INSERT stIn = new
NET_IN_ACCESS_FINGERPRINT_SERVICE_INSERT();
    stIn.nFpNum = 1;

    stIn.pFingerPrintInfo = fingerprint.getPointer();
```

```
        /**
         * 出参
         */
        NET_OUT_ACCESS_FINGERPRINT_SERVICE_INSERT stOut = new
NET_OUT_ACCESS_FINGERPRINT_SERVICE_INSERT();
        stOut.nMaxRetNum = 1;

        stOut.pFailCode = failCode.getPointer();

        stIn.write();
        stOut.write();
        fingerprint.write();
        failCode.write();
        if (netsdkApi.CLIENT_OperateAccessFingerprintService(loginHandle,
                emtype, stIn.getPointer(), stOut.getPointer(), TIME_OUT)) {

            // 打印具体信息
            System.out.println("添加信息结果 : " + failCode.nFailCode);
        } else {
            System.err.println("添加信息失败, " + getErrorCode());
        }

        fingerprint.read();
        failCode.read();
        stIn.read();
        stOut.read();
}

/**
 * 根据用户 ID 修改信息(一个用户 ID 对应一个信息)
 */
public void modifyFingerprint() {
        String userId = "1011";

        String finger = "xTxpAASFsk3+8hoDh4ky0ghH2oR7hjp658Wp"
                + "Q4eJQyEQBdoFgIerDuhFuAQGhsr6BvQpguuF804HMjpFaIUjOQ"
                + "AJyUSGiAwlCEupSGSFnA3gS7nF2IUUVe62SYVXhcSBoJG5iGSGvJqI"
                + "lZsRkYhksjBNnUmjh4VOEFEpizyGVUnol4oQQohVft6sOQdWh93lzarniBeJK"
                +
"4EANhtCKoqEmuA0C4M+im2+/zPJAzCKrcX5OipCtIpF9g6DqQOth32CAA8bUiiGdbog0wqKWofGYtx
```

```
wecgQhsZi"
                +
"ONMO3VqH5p7b8kmGiooaSQfF/MEWilKZ+Dgeg4iJuwYQBgyDJ4uEMfg3/kOsi53VB/++RTaLbf74N/uFY4NzPegFyQNfg3Ot6EXawda"
                +
"CO63/fhpAWIS0YehJyoRSg9SR+IfYRFCCbQXnxcrBR4RtTeCJykPKgrU5/rYagr6EvbIOeDhCOoWt5dB/yI"
                +
"dDgd3FB8f4BEGDpf3oR/oDQYImAQfJ+kM+iy5x+HXMRUaJznnQtCrGNoteifk4CkUohZaWyIepBTGEdsLX/9qBHoZO0pD"
                +
"JGog6iybq+Tn3x1yH7vrbco2HN4R+8s//+EIhhbcW18f3RZqGnx4LfhvXO4qvOvlB+0qQhpc9/L5MVlJH878zUyMvjxF2YSIlVRJUamE"
                +
"lNH8nRUGD8niPUp/xI6f28/czRPMPMfH2QvIvQSNEcyNB/zIyRSM/8zXzUvMzYfNFQ/IzUh9CNSf1X//xL0U6MohGebIWlDNWhRZi/xIVgkVP+Hhn"
                +
"9f9RAUWhRi8eafk+QXSytDwtFjGCklxSMTK8pNZYFILYkrkkYmS30qhUEcKqtNswRwKFkq8QQSHHkRBtESNRkfhdEbK9ovsIlVKMoYYdMtGdgSM0"
                + "ERE1cnIOQQTYoLJkIQJ0gJsNFFTugetWEhME5zESEnLokpsbFFS+kYVfVLGbs"
                + "E4qQTaYgcEOIRPEkfQRJGDhkWsOEmXWoLdpRKKNoA5tRENBoM"
                +
"5GEFEkosISZBLUkKlXIHbdkGRRAQHRoAwLDw4NFwgJChglKxEoKiYaGQcSBDIbIy0pNBQWFVVs";

    // 初始化信息数据
    NET_ACCESS_FINGERPRINT_INFO fingerprint = new NET_ACCESS_FINGERPRINT_INFO();

    /**
     * 添加信息数据
     */
    // 用户 ID
    System.arraycopy(userId.getBytes(), 0, fingerprint.szUserID, 0,
                userId.getBytes().length);

    // 将字符串转为信息数据
    byte[] fingerPrintBuffer = Base64Util.getDecoder().decode(finger);

    // 单个信息长度
    fingerprint.nPacketLen = fingerPrintBuffer.length;

    // 信息包个数
    fingerprint.nPacketNum = 1;
```

```
    // 信息数据
    // 先申请内存
    fingerprint.szFingerPrintInfo = new Memory(fingerPrintBuffer.length);
    fingerprint.szFingerPrintInfo.clear(fingerPrintBuffer.length);

    fingerprint.szFingerPrintInfo.write(0, fingerPrintBuffer, 0,
            fingerPrintBuffer.length);

    // ////////////////////////////////////////////////////////////////////////////////
    // ////////////////////////////////////////////////////////////////////////////////

    // 初始化
    FAIL_CODE failCode = new FAIL_CODE();

    // 信息操作类型
    // 修改信息信息
    int emtype =
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE.NET_EM_ACCESS_CTL_FINGERPRINT_SERVI
CE_UPDATE;

    /**
     * 入参
     */
    NET_IN_ACCESS_FINGERPRINT_SERVICE_UPDATE stIn = new
NET_IN_ACCESS_FINGERPRINT_SERVICE_UPDATE();
    stIn.nFpNum = 1;

    stIn.pFingerPrintInfo = fingerprint.getPointer();

    /**
     * 出参
     */
    NET_OUT_ACCESS_FINGERPRINT_SERVICE_UPDATE stOut = new
NET_OUT_ACCESS_FINGERPRINT_SERVICE_UPDATE();
    stOut.nMaxRetNum = 1;

    stOut.pFailCode = failCode.getPointer();

    stIn.write();
```

```
        stOut.write();
        fingerprint.write();
        failCode.write();
        if (netsdkApi.CLIENT_OperateAccessFingerprintService(loginHandle,
                emtype, stIn.getPointer(), stOut.getPointer(), TIME_OUT)) {

            // 打印具体信息
            System.out.println("修改信息结果 : " + failCode.nFailCode);
        } else {
            System.err.println("修改信息失败, " + getErrorCode());
        }

        fingerprint.read();
        failCode.read();
        stIn.read();
        stOut.read();
    }

    /**
     * 根据用户 ID 获取单个信息(一个用户 ID 对应一个信息)
     */
    public void getFingerprint() {
        String userId = "1011";

        // 信息操作类型
        // 获取信息信息
        int emtype =
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE.NET_EM_ACCESS_CTL_FINGERPRINT_SERVI
CE_GET;

        /**
         * 入参
         */
        NET_IN_ACCESS_FINGERPRINT_SERVICE_GET stIn = new
NET_IN_ACCESS_FINGERPRINT_SERVICE_GET();
        // 用户 ID
        System.arraycopy(userId.getBytes(), 0, stIn.szUserID, 0,
                userId.getBytes().length);

        /**
```

```
     * 出参
     */
    NET_OUT_ACCESS_FINGERPRINT_SERVICE_GET stOut = new
NET_OUT_ACCESS_FINGERPRINT_SERVICE_GET();
    // 接受信息数据的缓存的最大长度
    stOut.nMaxFingerDataLength = 1024;

    stOut.pbyFingerData = new Memory(1024);
    stOut.pbyFingerData.clear(1024);

    stIn.write();
    stOut.write();
    if (netsdkApi.CLIENT_OperateAccessFingerprintService(loginHandle,
            emtype, stIn.getPointer(), stOut.getPointer(), TIME_OUT)) {
        // 需要在此处，才能获取到具体信息
        stIn.read();
        stOut.read();

        byte[] buffer = stOut.pbyFingerData.getByteArray(0,
                stOut.nRetFingerDataLength);

        // 将获取到的信息转成没有乱码的字符串
        String figerStr = Base64Util.getEncoder().encodeToString(buffer);

        System.out.println("获取到的信息数据：" + figerStr);
    } else {
        System.err.println("获取信息失败, " + getErrorCode());
    }
}

/**
 * 根据用户 ID 删除信息
 */
public void deleteFingerprint() {
    String userID = "1011";

    // 初始化
    FAIL_CODE failCode = new FAIL_CODE();

    // 信息操作类型
```

```
    // 删除信息信息
    int emtype =
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE.NET_EM_ACCESS_CTL_FINGERPRINT_SERVI
CE_REMOVE;

    /**
     * 入参
     */
    NET_IN_ACCESS_FINGERPRINT_SERVICE_REMOVE stIn = new
NET_IN_ACCESS_FINGERPRINT_SERVICE_REMOVE();
    stIn.nUserNum = 1;
    System.arraycopy(userID.getBytes(), 0, stIn.szUserIDs[0].szUserID, 0,
            userID.getBytes().length);

    /**
     * 出参
     */
    NET_OUT_ACCESS_FINGERPRINT_SERVICE_REMOVE stOut = new
NET_OUT_ACCESS_FINGERPRINT_SERVICE_REMOVE();
    stOut.nMaxRetNum = 1;

    stOut.pFailCode = failCode.getPointer();

    stIn.write();
    stOut.write();
    if (netsdkApi.CLIENT_OperateAccessFingerprintService(loginHandle,
            emtype, stIn.getPointer(), stOut.getPointer(), TIME_OUT)) {

        // 打印具体信息
        System.out.println("删除信息结果 : " + failCode.nFailCode);

    } else {
        System.err.println("删除信息失败, " + getErrorCode());
    }

    stIn.read();
    stOut.read();
}

/**
```

```
 * 清空所有信息
 */
public void clearFingerprint() {
    // 信息操作类型
    // 清空信息信息
    int emtype =
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE.NET_EM_ACCESS_CTL_FINGERPRINT_SERVI
CE_CLEAR;

    /**
     * 入参
     */
    NET_IN_ACCESS_FINGERPRINT_SERVICE_CLEAR stIn = new
NET_IN_ACCESS_FINGERPRINT_SERVICE_CLEAR();

    /**
     * 出参
     */
    NET_OUT_ACCESS_FINGERPRINT_SERVICE_CLEAR stOut = new
NET_OUT_ACCESS_FINGERPRINT_SERVICE_CLEAR();

    stIn.write();
    stOut.write();
    if (netsdkApi.CLIENT_OperateAccessFingerprintService(loginHandle,
            emtype, stIn.getPointer(), stOut.getPointer(), TIME_OUT)) {
        System.out.println("清空信息成功 ！ ");
    } else {
        System.err.println("清空信息失败, " + getErrorCode());
    }

    stIn.read();
    stOut.read();
}
```

2.3.8 门配置

请参见“2.2.8 门配置”。

2.3.9 门时间管理

2.3.9.1 时段配置

请参见“2.2.9.1 时段配置”。

2.3.9.2 常开常闭时间段配置

请参见“2.2.9.2 常开常闭时间段配置”。

2.3.9.3 假日组

2.3.9.3.1 简介

假日组，即用户通过调用 SDK 接口对设备假日组进行配置，包含假日组名称、开始结束时间、组使能等信息。

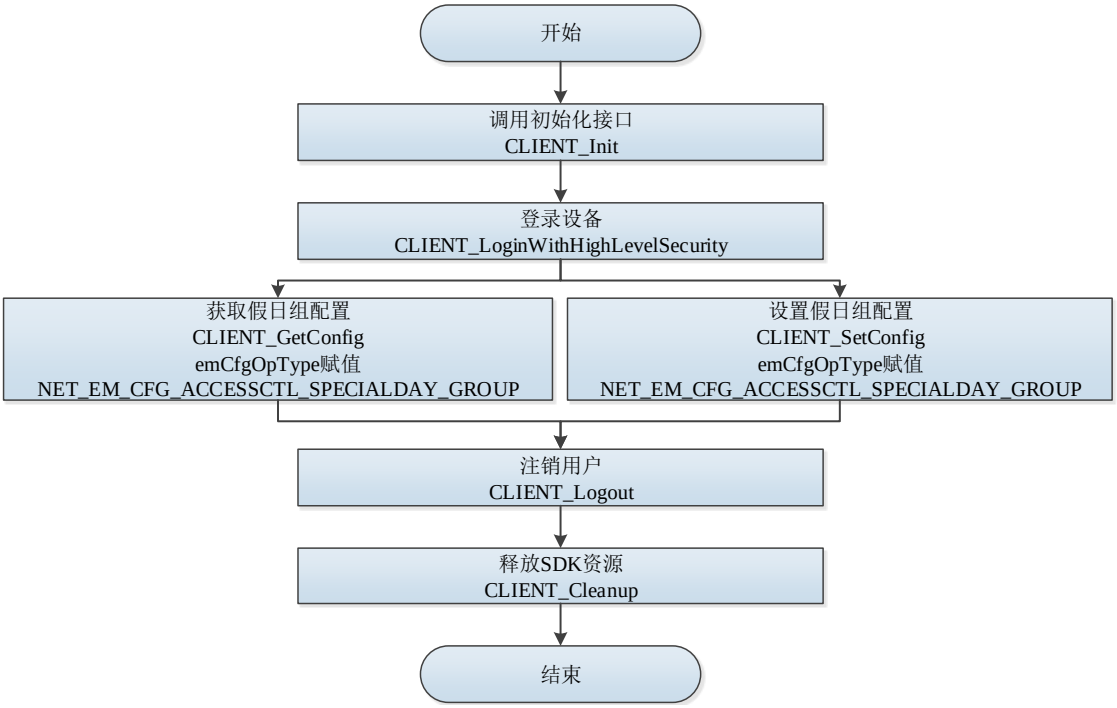
2.3.9.3.2 接口总览

表2-41 假日组接口说明

接口	说明
CLIENT_GetConfig	查询配置信息。
CLIENT_SetConfig	设置配置信息。

2.3.9.3.3 流程说明

图2-29 假日组业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetConfig 函数来查询门禁假日组配置信息。

表2-42 获取假日组信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szOutBuffer	dwOutBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP	获取假日组信息	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO 结构体尺寸

步骤4　调用 CLIENT_SetConfig 函数来设置门禁假日组配置信息。

表2-43 设置假日组信息 mCfgOpType 参数赋值说明

emCfgOpType	描述	szInBuffer	dwInBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP	设置假日组信息	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO 结构体尺寸

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.9.3.4 示例代码

```
/**
 * 下发假日组配置
 *
```

```
 * @return 下发是否成功
 */
public boolean setSpecialDayGroup() {
   NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO config = new
NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO();
   config.bGroupEnable = true;
   config.nSpeciaday = 2;
   // byte[]赋值请使用 System.arraycopy()
   System.arraycopy("test".getBytes(), 0, config.szGroupName, 0, "test".getBytes().length);
   System.arraycopy(
         "test1".getBytes(), 0, config.stuSpeciaday[0].szDayName, 0, "test1".getBytes().length);
   config.stuSpeciaday[0].stuStartTime = new NET_TIME(2020, 10, 19, 0, 0, 0);
   config.stuSpeciaday[0].stuEndTime = new NET_TIME(2020, 10, 21, 23, 59, 59);
   config.stuSpeciaday[1].szDayName = "test2".getBytes();
   System.arraycopy(
         "test2".getBytes(), 0, config.stuSpeciaday[1].szDayName, 0, "test2".getBytes().length);
   config.stuSpeciaday[1].stuStartTime = new NET_TIME(2020, 10, 22, 10, 10, 10);
   config.stuSpeciaday[1].stuEndTime = new NET_TIME(2020, 10, 23, 12, 0, 0);
   return configModule.setConfig(
         loginHandler,
NET_EM_CFG_OPERATE_TYPE.NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP, config, 0);
}

/** 获取假日组配置 */
public void getSpecialDayGroup() {
   NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO config = new
NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO();
   config =
         (NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO)
               configModule.getConfig(
                     loginHandler,

NET_EM_CFG_OPERATE_TYPE.NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP,
                     config,
                     0);
   if (config != null) {
      System.out.println(config.toString());
   }
}
```

2.3.9.4 假日计划

2.3.9.4.1 简介

假日计划，即用户通过调用 SDK 接口对设备假日计划进行配置，包含假日计划名称、使能、时间段、有效的门通道等信息。

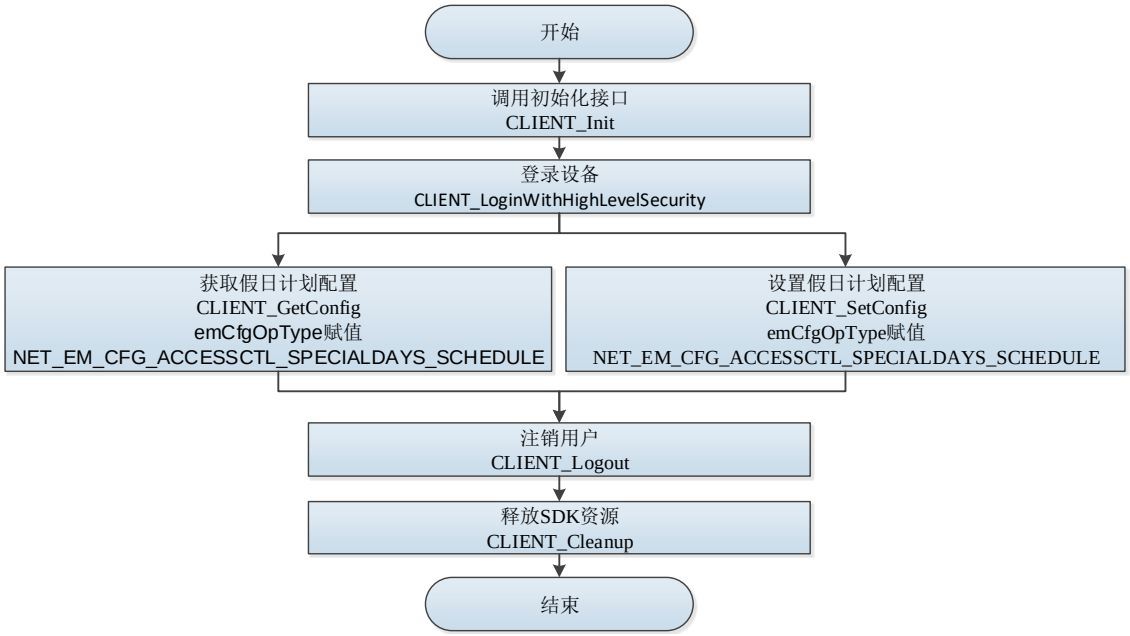
2.3.9.4.2 接口总览

表2-44 假日计划接口说明

接口	说明
CLIENT_GetConfig	查询配置信息。
CLIENT_SetConfig	设置配置信息。

2.3.9.4.3 流程说明

图2-30 假日计划业务流程



流程说明

步骤1　调用 CLIENT_Init 函数，完成 SDK 初始化流程。

步骤2　调用 CLIENT_LoginWithHighLevelSecurity 函数登录设备。

步骤3　调用 CLIENT_GetConfig 函数来查询门禁假日计划配置信息。

表2-45 获取假日计划信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szOutBuffer	dwOutBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE	获取假日计划信息	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO 结构体尺寸

步骤4　调用 CLIENT_SetConfig 函数来设置门禁假日计划配置信息。

表2-46 设置假日计划信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szInBuffer	dwInBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE	设置假日计划信息	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO 结构体尺寸

步骤5　业务执行完成之后，调用 CLIENT_Logout 函数登出设备。

步骤6　SDK 功能使用完后，调用 CLIENT_Cleanup 函数释放 SDK 资源。

2.3.9.4.4 示例代码

```
/** 下发假日计划 */
public void setSpecialDaysSchedule() {
    NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO config =
            new NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO();
    System.arraycopy(
            "scheduleName".getBytes(), 0, config.szSchduleName, 0, "scheduleName".getBytes().length);
    // 计划使能
    config.bSchdule = true;
    // 假日组的下标,NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO 中 stuSpeciaday 的下标
    config.nGroupNo = 1;
    // 时间段
    config.nTimeSection = 2;
    config.stuTimeSection[0].setTime(1, 5, 0, 0, 14, 0, 0);
    config.stuTimeSection[1].setTime(0, 15, 0, 0, 21, 0, 0);
    // 生效的门数量
    config.nDoorNum = 1;
    // 生效的门
    config.nDoors[0] = 1;
    configModule.setConfig(
            loginHandler,
NET_EM_CFG_OPERATE_TYPE.NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE, config,
0);
}

/** 获取假日计划 */
public void getSpecialDaysSchedule() {
    NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO config =
            new NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO();
    config.nGroupNo = 1;
    config =
            (NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO)
                    configModule.getConfig(
                            loginHandler,

NET_EM_CFG_OPERATE_TYPE.NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE,
                            config,
                            0);
    if (config != null) {
        System.out.println(config.toString());
    }
}
```

2.3.10 门高级配置

请参见“2.2.10 门高级配置”。

2.3.11 记录查询

2.3.11.1 开门记录

请参见“2.2.11.1 开门记录”。

2.3.11.2 设备日志

请参见“2.2.11.2 设备日志”。

2.3.12 门禁事件

请参见“2.2.12 门禁事件”。

2.3.13 人证比对事件

请参见“2.2.13 人证比对事件”。

第 3 章 接口函数

3.1 通用接口

3.1.1 SDK 初始化

3.1.1.1 SDK 初始化 CLIENT_Init

表3-1 SDK 初始化说明书

选项	说明	
描述	对整个 SDK 进行初始化	
函数	public boolean CLIENT_Init(Callback cbDisConnect, Pointer dwUser);	
参数	[in]cbDisConnect	断线回调函数
	[in]dwUser	断线回调函数的用户参数
返回值	成功返回 TRUE，败返回 FALSE	
说明	● 调用网络 SDK 其他函数的前提 ● 回调函数设置成 NULL 时，设备断线后不会回调给用户	

3.1.1.2 SDK 清理 CLIENT_Cleanup

表3-2 SDK 清理说明

选项	说明
描述	清理 SDK
函数	public void CLIENT_Cleanup();
参数	无
返回值	无
说明	SDK 清理接口，在结束前最后调用

3.1.1.3 设置断线重连回调函数 CLIENT_SetAutoReconnect

表3-3 设置断线重连回调函数说明

选项	说明
描述	设置自动重连回调函数

选项	说明	
函数	public void CLIENT_SetAutoReconnect(Callback cbAutoConnect, Pointer dwUser);	
参数	[in]cbAutoConnect	断线重连回调函数
	[in]dwUser	断线重连回调函数的用户参数
返回值	无	
说明	设置断线重连回调接口。如果回调函数设置为 NULL，则不自动重连	

3.1.1.4 设置网络参数 CLIENT_SetNetworkParam

表3-4 设备网络参数说明

选项	说明	
描述	设置网络环境相关参数	
函数	public void CLIENT_SetNetworkParam(NET_PARAM pNetParam);	
参数	[in]pNetParam	网络延迟、重连次数、缓存大小等参数
返回值	无	
说明	可根据实际网络环境，调整参数	

3.1.2 设备登录

3.1.2.1 用户登录设备 CLIENT_LoginWithHighLevelSecurity

表3-5 用户登录设备说明

选项	说明	
描述	用户登录设备	
函数	Public LLong CLIENT_LoginWithHighLevelSecurity(NET_IN_LOGIN_WITH_HIGHLEVEL_SECURITY pstInParam, NET_OUT_LOGIN_WITH_HIGHLEVEL_SECURITY pstOutParam);	
参数	[in] pstInParam	登录参数包括 IP、端口、用户名、密码、登录模式等
	[out] pstOutParam	设备登录输出参数包括设备信息、错误码
返回值	● 成功返回非 0 ● 失败返回 0	
说明	高安全级别登录接口。 **说明** CLINET_LoginEx2 仍然可以使用，但存在安全风险，所以强烈推荐使用最新接口 CLIENT_LoginWithHighLevelSecurity 登录设备。	

参数 pstOutParam 中 error 的错误码及含义说明，请参见表 3-6。

表3-6 参数 error 的错误码及含义

error 的错误码	对应的含义
1	密码不正确
2	用户名不存在
3	登录超时

error 的错误码	对应的含义
4	账号已登录
5	账号已被锁定
6	账号被列为禁止名单
7	资源不足，设备系统忙
8	子连接失败
9	主连接失败
10	超过最大用户连接数
11	缺少 avnetsdk 或 avnetsdk 的依赖库
12	设备未插入 U 盘或 U 盘信息错误
13	客户端 IP 地址没有登录权限

3.1.2.2 用户登出设备 CLIENT_Logout

表3-7 用户登出设备说明

选项	说明	
描述	用户登出设备	
函数	public boolean CLIENT_Logout(LLong lLoginID);	
参数	[in]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.1.3 设备控制

3.1.3.1 设备控制 CLIENT_ControlDeviceEx

表3-8 设备控制说明

选项	说明	
描述	设备控制	
函数	public boolean CLIENT_ControlDeviceEx(LLong lLoginID, int emType, Pointer pInBuf, Pointer pOutBuf, int nWaitTime);	
参数	[in]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
	[in]emType	控制类型
	[in]pInBuf	输入参数，因 emType 不同而不同
	[out]pOutBuf	输出参数，默认为 NULL，对于有些 emType 有相应输出结构体
	[in]waittime	超时时间，默认 1000ms，可根据需要自行设置
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

emType、pInBuf 和 pOutBuf 的对照关系，请参见表 3-9。

表3-9 emType、pInBuf 和 pOutBuf 对照关系

emType	描述	pInBuf	pOutBuf
CTRLTYPE_CTRL_ARMED_EX	布撤防	CTRL_ARM_DISARM_PARAM	NULL
CTRLTYPE_CTRL_SET_BYPASS	设置旁路功能	NET_CTRL_SET_BYPASS	NULL
CTRLTYPE_CTRL_ACCESS_OPEN	门禁控制-开门	NET_CTRL_ACCESS_OPEN	NULL
CTRLTYPE_CTRL_ACCESS_CLOSE	门禁控制-关门	NET_CTRL_ACCESS_CLOSE	NULL

3.2 智能订阅

3.2.1 开始智能事件订阅 CLIENT_RealLoadPictureEx

表3-10 开始智能事件订阅 CLIENT_RealLoadPictureEx

选项	说明	
描述	开始智能事件订阅	
方法	public LLong CLIENT_RealLoadPictureEx(LLong lLoginID, int nChannelID, int dwAlarmType, int bNeedPicFile, StdCallCallback cbAnalyzerData, Pointer dwUser, Pointer Reserved);	
参数	[in]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
	[in]nChannelID	设备通道号(从 0 开始)
	[in]dwAlarmType	订阅报警事件类型
	[in]bNeedPicFile	是否订阅图片文件
	[in]cbAnalyzerData	智能事件回调函数
	[in]dwUser	用户自定义数据类型
	[in]Reserved	保留字段
返回值	成功返回 LLONG 类型订阅句柄，失败返回 0	
说明	接口返回失败，请使用 CLIENT_GetLastError 获取错误码	

智能报警事件类型说明请参见表 3-11。

表3-11 智能事件类型说明

dwAlarmType 宏定义	宏定义值	含义	回调 pAlarmInfo 对应结构体
EVENT_IVS_ALL	0x00000001	所有事件	无
EVENT_IVS_CROSSFENCEDETECTION	0x0000011F	穿越围栏	DEV_EVENT_CROSSFENCEDETECTION_INFO

dwAlarmType 宏定义	宏定义值	含义	回调 pAlarmInfo 对应结构体
EVENT_IVS_CROSSLINEDETECTION	0x00000002	绊线入侵	DEV_EVENT_CROSSLINE_INFO
EVENT_IVS_CROSSREGIONDETECTION	0x00000003	区域入侵	DEV_EVENT_CROSSREGION_INFO
EVENT_IVS_LEFTDETECTION	0x00000005	物品遗留	DEV_EVENT_LEFT_INFO
EVENT_IVS_PRESERVATION	0x00000008	物品保全	DEV_EVENT_PRESERVATION_INFO
EVENT_IVS_TAKENAWAYDETECTION	0x00000115	物品搬移	DEV_EVENT_TAKENAWAYDETECTION_INFO
EVENT_IVS_WANDERDETECTION	0x00000007	徘徊事件	DEV_EVENT_WANDER_INFO
EVENT_IVS_VIDEOABNORMALDETECTION	0x00000013	视频异常	DEV_EVENT_VIDEOABNORMALDETECTION_INFO
EVENT_IVS_AUDIO_ABNORMALDETECTION	0x00000126	声音异常	DEV_EVENT_IVS_AUDIO_ABNORMALDETECTION_INFO
EVENT_IVS_CLIMBDETECTION	0x00000128	攀高检测	DEV_EVENT_IVS_CLIMB_INFO
EVENT_IVS_FIGHTDETECTION	0x0000000E	斗殴检测	DEV_EVENT_FLOWSTAT_INFO
EVENT_IVS_LEAVEDETECTION	0x00000129	离岗检测	DEV_EVENT_IVS_LEAVE_INFO
EVENT_IVS_PSRISEDETECTION	0x0000011E	起身检测	DEV_EVENT_PSRISEDETECTION_INFO
EVENT_IVS_PASTEDETECTION	0x00000004	非法黏贴物贴条检测	DEV_EVENT_PASTE_INFO

3.2.2 停止智能事件订阅 CLIENT_StopLoadPic

表3-12 停止智能事件订阅 CLIENT_StopLoadPic

选项	说明	
描述	停止智能事件订阅	
方法	public boolean CLIENT_StopLoadPic(LLong lAnalyzerHandle);	
参数	[in]lAnalyzerHandle	智能事件订阅句柄
返回值	BOOL 类型 ● 成功：TRUE ● 失败：FALSE	
说明	接口返回失败，请使用 CLIENT_GetLastError 获取错误码	

3.3 门禁控制器/指纹一体机（一代）

3.3.1 门禁控制

关于门控制接口的详细介绍，请参见“3.1.3.1 设备控制 CLIENT_ControlDeviceEx”。

关于门磁状态接口的详细介绍，请参见“3.3.2.4 查询设备状态 CLIENT_QueryDevState”。

3.3.2 设备信息查看

3.3.2.1 查询系统能力信息 CLIENT_QueryNewSystemInfo

表3-13 查询系统能力信息说明

选项	说明	
描述	查询系统能力信息，按字符串格式	
函数	public boolean CLIENT_QueryNewSystemInfo(LLong lLoginID, String szCommand, int nChannelID, byte[] szOutBuffer, int dwOutBufferSize, IntByReference error, int waittime);	
参数	[in]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
	[in] szCommand	命令参数，详细介绍请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”
	[in] nChannelID	通道号
	[out] szOutBuffer	接收的协议缓冲区
	[in] dwOutBufferSize	接收的总字节数(单位字节)
	[out] error	错误号
	[in]waittime	超时时间，默认 1000ms，可根据需要自行设置
返回值	成功返回 TRUE，失败返回 FALSE	
说明	获取到的信息按照字符串格式，各个字符串包含的信息由 CLIENT_ParseData 解析	

表3-14 参数 error 的错误码及含义

error 的错误码	对应的含义
0	成功
1	失败
2	数据不合法
3	暂时无法设置
4	没有权限

3.3.2.2 解析查询到的配置信息 CLIENT_ParseData

表3-15 解析查询到的配置信息说明

选项	说明	
描述	解析查询到的配置信息	
函数	public boolean CLIENT_ParseData(String szCommand, byte[] szInBuffer, Pointer lpOutBuffer, int dwOutBufferSize, Pointer pReserved);	
参数	[in] szCommand	命令参数，详细介绍请参见表 3-16
	[in] szInBuffer	输入缓冲，字符配置缓冲
	[out] lpOutBuffer	输出缓冲，结构体类型请参见表 3-16
	[in] dwOutBufferSize	输出缓冲的大小
	[in] pReserved	保留参数
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

表3-16 szCommand、查询类型和对应结构体的对照关系

szCommand	查询类型	对应结构体
CFG_CAP_CMD_ACCESSCONTROLMANAGER	门禁能力	CFG_CAP_ACCESSCONTROL
CFG_CMD_NETWORK	IP 配置	CFG_NETWORK_INFO
CFG_CMD_DVRIP	主动注册配置	CFG_DVRIP_INFO
CFG_CMD_NTP	NTP 校时	CFG_NTP_INFO
CFG_CMD_ACCESS_EVENT	门禁事件配置（门配置信息、常开常闭时段配置、分手段、首卡开门配置）	CFG_ACCESS_EVENT_INFO
CFG_CMD_ACCESSTIMESCHEDULE	门禁刷卡时段（时段配置）	CFG_ACCESS_TIMESCHEDULE_INFO
CFG_CMD_OPEN_DOOR_GROUP	多人组合开门配置	CFG_OPEN_DOOR_GROUP_INFO
CFG_CMD_ACCESS_GENERAL	门禁基本配置（多门互锁）	CFG_ACCESS_GENERAL_INFO
CFG_CMD_OPEN_DOOR_ROUTE	开门路线集合，或称防反潜路线配置	CFG_OPEN_DOOR_ROUTE_INFO

3.3.2.3 获取设备能力 CLIENT_GetDevCaps

表3-17 获取设备能力说明

选项	说明
描述	获取设备能力

选项	说明	
函数	public boolean CLIENT_GetDevCaps(LLong lLoginID, int nType, Pointer pInBuf, Pointer pOutBuf, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] nType	设备类型 控制参数根据 type 不同而不同
	[in] pInBuf	获取设备能力（入参）
	[out] pOutBuf	获取设备能力（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

nType、pInBuf 和 pOutBuf 的对照关系，请参见表 3-18。

表3-18 nType、pInBuf 和 pOutBuf 对照关系

nType	描述	pInBuf	pOutBuf
NET_FACEINFO_CAPS	获得人脸门禁控制器能力集	NET_IN_GET_FACEINFO_CAPS	NET_OUT_GET_FACEINFO_CAPS

3.3.2.4 查询设备状态 CLIENT_QueryDevState

表3-19 查询设备状态说明

选项	说明	
描述	获取前端设备的当前工作状态	
函数	public boolean CLIENT_QueryDevState(LLong lLoginID, int nType, Pointer pBuf, int nBufLen, IntByReference pRetLen, int waittime);	
参数	[in] lLoginID	登录句柄
	[in] nType	设备类型 控制参数,根据 type 不同而不同
	[out] pBuf	输出参数，用于接收查询返回的数据的缓存。根据查询类型的不同，返回数据的数据结构也不同
	[in] nBufLen	缓存长度，单位字节
	[in] waittime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

nType、查询类型和结构体的对应关系，请参见表 3-20。

表3-20 nType、查询类型和结构体的对应关系

nType	描述	pBuf
NET_DEVSTATE_SOFTWARE	查询设备软件版本信息	NETDEV_VERSION_INFO
NET_DEVSTATE_NETINTERFACE	查询网络接口信息	NETDEV_NETINTERFACE_INFO
NET_DEVSTATE_DEV_RECORDSET	查询设备记录集信息	NET_CTRL_RECORDSET_PARAM
NET_DEVSTATE_DOOR_STATE	查询门禁状态（门磁）	NET_DOOR_STATUS_INFO

3.3.3 网络配置

3.3.3.1 IP 配置

3.3.3.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig

表3-21 查询配置信息说明

选项	说明	
描述	获取配置，按照字符串格式	
函数	public boolean CLIENT_GetNewDevConfig(LLong lLoginID, String szCommand, int nChannelID, byte[] szOutBuffer, int dwOutBufferSize, IntByReference error, int waiitime, Pointer pReserved);	
参数	[in] lLoginID	登录句柄
	[in] szCommand	命令参数，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”
	[in] nChannelID	通道号
	[out]szOutBuffer	输出缓冲
	[in] dwOutBufferSize	输出缓冲大小
	[out] error	错误码
	[in] waittime	等待超时时间
返回值	成功返回 TRUE，失败返回 FALSE	

选项	说明
说明	获取配置，按照字符串格式，各个字符串包含的信息由 CLIENT_ParseData 解析

表3-22 参数 error 的错误码及含义说明

error 的错误码	对应的含义
0	成功
1	失败
2	数据不合法
3	暂时无法设置
4	没有权限

3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig

表3-23 设置配置信息说明

选项	说明	
描述	获取配置，按照字符串格式	
函数	public boolean CLIENT_SetNewDevConfig(LLong lLoginID, String szCommand, int nChannelID, byte[] szInBuffer, int dwInBufferSize, IntByReference error, IntByReference restart, int waittime);	
参数	[in] lLoginID	登录句柄
	[in] szCommand	命令参数信息，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”
	[in] nChannelID	通道号
	[in] szInBuffer	输出缓冲
	[in] dwInBufferSize	输出缓冲大小
	[out] error	错误码
	[out] restart	配置设置后是否需要重启设备，1 表示需要重启，0 表示不需要重启
	[in] waittime	等待超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	设置配置，按照字符串格式，各个字符串包含的信息由 CLIENT_PacketData 组包	

表3-24 参数 error 的错误码及含义说明

error 的错误码	对应的含义
0	成功
1	失败
2	数据不合法
3	暂时无法设置
4	没有权限

3.3.3.1.4 打包字符串格式 CLIENT_PacketData

表3-25 打包字符串格式说明

选项	说明	
描述	将需要设置的配置信息，打包成字符串格式	
函数	public boolean CLIENT_PacketData(String szCommand, Pointer lpInBuffer, int dwInBufferSize, byte[] szOutBuffer, int dwOutBufferSize);	
参数	[out] szCommand	命令参数，具体请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”
	[in] lpInBuffer	输入缓冲，结构体类型请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”
	[in] dwInBufferSize	输出缓冲的大小
	[out] szOutBuffer	输出缓冲
	[in] dwOutBufferSize	输出缓冲大小
返回值	成功返回 TRUE，失败返回 FALSE	
说明	此接口配合 CLIENT_SetNewDevConfig 使用，使用 CLIENT_PacketData 后，将打包的信息通过 CLIENT_SetNewDevConfig 设置到设备上	

3.3.3.2 主动注册配置

3.3.3.2.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.3.2.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.3.2.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.3.2.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.4 时间设置

3.3.4.1 时间设置

表3-26 时间设置说明

选项	说明	
描述	设置设备当前时间	
函数	public boolean CLIENT_SetupDeviceTime(LLong lLoginID, NET_TIME pDeviceTime);	
参数	[in] lLoginID	登录句柄
	[in] pDeviceTime	设置的设备时间指针
返回值	成功返回 TRUE，失败返回 FALSE	
说明	应用于系统校时时更改当前前端设备系统时间与本机系统时间同步	

3.3.4.2 NTP 校时、时区配置

3.3.4.2.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.4.2.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.4.2.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.4.2.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.4.3 夏令时配置

3.3.4.3.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.4.3.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息

CLIENT_GetNewDevConfig”。

3.3.4.3.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.4.3.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.5 人员管理

3.3.5.1 人员信息字段集合

请参见“3.3.2.4 查询设备状态 CLIENT_QueryDevState”。

3.3.6 门配置

3.3.6.1 门配置信息

3.3.6.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.6.1.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.6.1.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.6.1.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.7 门时间配置

3.3.7.1 时段配置

3.3.7.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.7.1.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.7.1.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.7.1.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.7.2 常开常闭时段配置

3.3.7.2.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.7.2.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.7.2.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.7.2.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.8 门高级配置

3.3.8.1 分时段、首卡开门

3.3.8.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.8.1.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.8.1.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.8.1.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.8.2 多人组合开门

3.3.8.2.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.8.2.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.8.2.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.8.2.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.8.3 多门互锁

3.3.8.3.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.3.8.3.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.3.8.3.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.3.8.3.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.3.8.4 设备日志

3.3.8.4.1 查询设备日志条数 CLIENT_QueryDevLogCount

表3-27 查询设备日志条数说明

选项	说明	
描述	查询设备日志条数	
函数	public boolean CLIENT_QueryDevLogCount(LLong lLoginID, Pointer pInParam, Pointer pOutParam, int waittime);	
参数	[in]lLoginID	设备登录句柄
	[in] pInParam	查询日志的参数，详细介绍请参见 NET_IN_GETCOUNT_LOG_PARAM
	[out] pOutParam	返回日志条数，详细介绍请参见 NET_OUT_GETCOUNT_LOG_PARAM
	[in] waittime	查询超时时间
返回值	返回查询日志条数	
说明	无	

3.3.8.4.2 开始查询日志 CLIENT_StartQueryLog

表3-28 开始查询日志说明

选项	说明
描述	开始查询设备日志

选项	说明	
函数	public LLong CLIENT_StartQueryLog(LLong lLoginID, Pointer pInParam, Pointer pOutParam, int nWaitTime);	
参数	[in]lLoginID	设备登录句柄
	[in] pInParam	开始查询日志的参数，详细介绍请参见 NET_IN_START_QUERYLOG
	[out] pOutParam	开始查询日志输出参数，详细介绍请参见 NET_OUT_START_QUERYLOG
	[in] nWaitTime	查询超时时间
返回值	成功返回查询句柄，失败返回 0	
说明	无	

3.3.8.4.3 获取日志 CLIENT_QueryNextLog

表3-29 获取日志说明

选项	说明	
描述	获取日志	
函数	public boolean CLIENT_QueryNextLog(LLong lLogID, Pointer pInParam, Pointer pOutParam, int nWaitTime);	
参数	[in] lLogID	查询日志句柄
	[in] pInParam	获取日志的输入参数，详细介绍请参见 NET_IN_QUERYNEXTLOG
	[out] pOutParam	获取日志的输出参数，详细介绍请参见 NET_OUT_QUERYNEXTLOG
	[in] nWaitTime	查询超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.3.8.4.4 结束查询日志 CLIENT_StopQueryLog

表3-30 结束查询日志说明

选项	说明	
描述	停止查询设备日志	
函数	public boolean CLIENT_StopQueryLog(LLong lLogID);	
参数	[in] lLogID	查询日志句柄
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.3.9 记录查询

3.3.9.1 开门记录

3.3.9.1.1 查找记录条数 CLIENT_QueryRecordCount

表3-31 查询记录条数说明

选项	说明	
描述	查找记录条数	
函数	public boolean CLIENT_QueryRecordCount(NET_IN_QUEYT_RECORD_COUNT_PARAM pInParam, NET_OUT_QUEYT_RECORD_COUNT_PARAM pOutParam, int waittime);	
参数	[in] pInParam	查询记录条数入参，pInParam ->lFindeHandle 为 CLIENT_FindRecord 的 pOutParam-> lFindeHandle
	[out] pOutParam	查询记录条数出参，pOutParam ->nRecordCount 为记录个数
	[in] waittime	查询超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	调用本接口之前应先调用 CLIENT_FindRecord 以打开查询句柄	

3.3.9.1.2 按查询条件查询记录 CLIENT_FindRecord

表3-32 按查询条件查询记录说明

选项	说明	
描述	按查询条件查询记录	
函数	public boolean CLIENT_FindRecord(LLong lLoginID, NET_IN_FIND_RECORD_PARAM pInParam, NET_OUT_FIND_RECORD_PARAM pOutParam, int waittime);	
参数	[in]lLoginID	设备登录句柄
	[in] pInParam	查询日志的参数，参数赋值说明请参见表 3-33
	[out] pOutParam	查询日志的出参，返回查询句柄
	[in] waittime	等待超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	可以先调用本接口获得查询句柄，再调用 CLIENT_FindNextRecord 函数获取记录列表，查询完毕可以调用 CLIENT_FindRecordClose 关闭查询句柄	

表3-33 查询开门记录入参数说明

pInParam 结构体	赋值	说明
emType	NET_RECORD_ACCESSCTLCARDREC_EX	用于开门记录查询

3.3.9.1.3 查找记录 CLIENT_FindNextRecord

表3-34 查找记录说明

选项	说明	
描述	查找记录:nFilecount:需要查询的条数,返回值为媒体文件条数返回值小于 nFilecount 则相应时间段内的文件查询完毕	
函数	public boolean CLIENT_FindNextRecord(NET_IN_FIND_NEXT_RECORD_PARAM pInParam, NET_OUT_FIND_NEXT_RECORD_PARAM pOutParam, int waittime);	
参数	[in] pInParam	查询记录入参，pInParam ->lFindeHandle 为 CLIENT_FindRecord 的 pOutParam-> lFindeHandle
	[out] pOutParam	查询记录出参返回记录信息
	[in] waittime	等待超时时间
返回值	1：成功取回一条记录，0：记录已取完，-1：参数出错	
说明	无	

表3-35 查询开门记录出参说明

pOutParam 结构体	赋值	说明
pRecordList	NET_RECORDSET_ACCESS_CTL_CARDREC	用于开门记录查询

3.3.9.1.4 结束记录查询 CLIENT_FindRecordClose

表3-36 结束记录查询说明

选项	说明	
描述	结束记录查询	
函数	public boolean CLIENT_FindRecordClose(LLong lFindHandle);	
参数	[in] lFindHandle	CLIENT_FindRecord 的返回值
返回值	成功返回 TRUE，失败返回 FALSE	
说明	调用 CLIENT_FindRecord 打开查询句柄，查询完毕后应调用本函数以关闭查询句柄	

3.4 门禁控制器/人脸一体机（二代）

3.4.1 门禁控制

关于门控制接口的详细介绍，请参见“3.1.3.1 设备控制 CLIENT_ControlDeviceEx”。

关于门磁状态接口的详细介绍，请参见“3.3.2.4 查询设备状态 CLIENT_QueryDevState”。

3.4.2 设备信息查看

3.4.2.1 获取设备能力 CLIENT_GetDevCaps

表3-37 获取设备能力说明

选项	说明	
描述	获取设备能力	
函数	public boolean CLIENT_GetDevCaps(LLong lLoginID, int nType, Pointer pInBuf, Pointer pOutBuf, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] nType	设备类型 控制参数根据 type 不同而不同
	[in] pInBuf	获取设备能力（入参）
	[out] pOutBuf	获取设备能力（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

nType、pInBuf 和 pOutBuf 的对照关系，请参见表 3-18。

表3-38 nType、pInBuf 和 pOutBuf 对照关系

nType	描述	pInBuf	pOutBuf
NET_ACCESSCONTROL_CAPS	获取门禁能力	NET_IN_AC_CAPS	NET_OUT_AC_CAPS

3.4.2.2 查询设备状态 CLIENT_QueryDevState

关于 CLIENT_QueryDevState 详细介绍，请参见“3.3.2.4 查询设备状态 CLIENT_QueryDevState”。

3.4.3 网络配置

请参见“3.3.3 网络配置”。

3.4.4 时间设置

请参见“3.3.4 时间设置”。

3.4.5 人员管理

3.4.5.1 用户管理

3.4.5.1.1 门禁用户信息管理接口 CLIENT_OperateAccessUserService

表3-39 门禁用户信息管理接口说明

选项	说明	
描述	门禁人员信息管理接口	
函数	public boolean CLIENT_OperateAccessUserService(LLong lLoginID, int emtype, Pointer pstInParam, Pointer pstOutParam, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] emtype	用户信息操作类型
	[in] pInBuf	用户信息管理（入参）
	[out] pOutBuf	用户信息管理（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

emtype、pInBuf 和 pOutBuf 的对照关系，请参见表 3-40。

表3-40 nType、pInBuf 和 pOutBuf 对照关系

emtype	描述	pInBuf	pOutBuf
NET_EM_ACCESS_CTL_USER_SERVICE_INSERT	添加用户信息	NET_IN_ACCESS_USER_SERVICE_INSERT	NET_OUT_ACCESS_USER_SERVICE_INSERT
NET_EM_ACCESS_CTL_USER_SERVICE_REMOVE	删除用户信息	NET_IN_ACCESS_USER_SERVICE_REMOVE	NET_OUT_ACCESS_USER_SERVICE_REMOVE
NET_EM_ACCESS_CTL_USER_SERVICE_CLEAR	清空所有用户信息	NET_IN_ACCESS_USER_SERVICE_CLEAR	NET_OUT_ACCESS_USER_SERVICE_CLEAR

3.4.5.1.2 开始查询人员信息接口 CLIENT_StartFindUserInfo

表3-41 开始查询人员信息接口说明

选项	说明
描述	开始查询人员信息接口

选项	说明	
函数	public LLong CLIENT_StartFindUserInfo(LLong lLoginID, NET_IN_USERINFO_START_FIND pstIn, NET_OUT_USERINFO_START_FIND pstOut, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] pstIn	开始查询人员信息接口（入参）
	[out] pstOut	开始查询人员信息接口（出参）
	[in] nWaitTime	超时时间
返回值	成功返回查询句柄，失败返回 0	
说明	无	

3.4.5.1.3 获取人员信息接口 CLIENT_DoFindUserInfo

表3-42 获取人员信息接口说明

选项	说明	
描述	获取人员信息接口	
函数	public boolean CLIENT_DoFindUserInfo(LLong lFindHandle, NET_IN_USERINFO_DO_FIND pstIn, NET_OUT_USERINFO_DO_FIND pstOut, int nWaitTime);	
参数	[in] lFindHandle	CLIENT_StartFindUserInfo 返回值
	[in] pstIn	获取人员信息接口（入参）
	[out] pstOut	获取人员信息接口（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.4.5.1.4 停止查询人员信息接口 CLIENT_StopFindUserInfo

表3-43 停止查询人员信息接口说明

选项	说明	
描述	停止查询人员信息接口	
函数	public boolean CLIENT_StopFindUserInfo(LLong lFindHandle);	
参数	[in] lFindHandle	CLIENT_StartFindUserInfo 返回值
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.4.5.2 卡片管理

3.4.5.2.1 门禁卡片信息管理接口 CLIENT_OperateAccessCardService

表3-44 门禁卡片信息管理接口说明

选项	说明	
描述	门禁卡片信息管理接口	
函数	public boolean CLIENT_OperateAccessCardService(LLong lLoginID, int emtype, Pointer pstInParam, Pointer pstOutParam, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] emtype	卡片信息操作类型
	[in] pInBuf	卡片信息管理（入参）
	[out] pOutBuf	卡片信息管理（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

emtype、pInBuf 和 pOutBuf 的对照关系，请参见表 3-18。

表3-45 nType、pInBuf 和 pOutBuf 对照关系

emtype	描述	pInBuf	pOutBuf
NET_EM_ACCESS_CTL_CARD_SERVICE_INSERT	添加卡片信息	NET_IN_ACCESS_CARD_SERVICE_INSERT	NET_OUT_ACCESS_CARD_SERVICE_INSERT
NET_EM_ACCESS_CTL_CARD_SERVICE_REMOVE	删除卡片信息	NET_IN_ACCESS_CARD_SERVICE_REMOVE	NET_OUT_ACCESS_CARD_SERVICE_REMOVE
NET_EM_ACCESS_CTL_CARD_SERVICE_CLEAR	清空所有卡片信息	NET_IN_ACCESS_CARD_SERVICE_CLEAR	NET_OUT_ACCESS_CARD_SERVICE_CLEAR

3.4.5.2.2 开始查询卡片信息接口 CLIENT_StartFindCardInfo

表3-46 开始查询卡片信息接口说明

选项	说明	
描述	开始查询卡片信息接口	
函数	public LLong CLIENT_StartFindCardInfo(LLong lLoginID, NET_IN_CARDINFO_START_FIND pstIn, NET_OUT_CARDINFO_START_FIND pstOut, int nWaitTime);	
参数	[in] lLoginID	登录句柄

选项	说明	
	[in] pstIn	开始查询卡片信息接口（入参）
	[out] pstOut	开始查询卡片信息接口（出参）
	[in] nWaitTime	超时时间
返回值	成功返回查询句柄，失败返回 0	
说明	无	

3.4.5.2.3 获取卡片信息接口 CLIENT_DoFindCardInfo

表3-47 获取卡片信息接口说明

选项	说明	
描述	获取卡片信息接口	
函数	public boolean CLIENT_DoFindCardInfo(LLong lFindHandle, NET_IN_CARDINFO_DO_FIND pstIn, NET_OUT_CARDINFO_DO_FIND pstOut, int nWaitTime);	
参数	[in] lFindHandle	CLIENT_StartFindCardInfo 返回值
	[in] pstIn	获取卡片信息接口（入参）
	[out] pstOut	获取卡片信息接口（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.4.5.2.4 停止查询卡片信息接口 CLIENT_StopFindUserInfo

表3-48 停止查询卡片信息接口说明

选项	说明	
描述	停止查询卡片信息接口	
函数	public boolean CLIENT_StopFindCardInfo(LLong lFindHandle);	
参数	[in] lFindHandle	CLIENT_StartFindCardInf 返回值
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

3.4.5.3 人脸管理

3.4.5.3.1 门禁人脸信息管理接口 CLIENT_OperateAccessFaceService

表3-49 门禁人脸信息管理接口说明

选项	说明
描述	门禁人脸信息管理接口

选项	说明	
函数	public boolean CLIENT_OperateAccessFaceService(LLong lLoginID, int emtype, Pointer pstInParam, Pointer pstOutParam, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] emtype	人脸信息操作类型
	[in] pInBuf	人脸信息管理（入参）
	[out] pOutBuf	人脸信息管理（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

emtype、pInBuf 和 pOutBuf 的对照关系，请参见表 3-50。

表3-50 nType、pInBuf 和 pOutBuf 对照关系

emtype	描述	pInBuf	pOutBuf
NET_EM_ACCESS_CTL_FACE_SERVICE_INSERT	添加人脸信息	NET_IN_ACCESS_FACE_SERVICE_INSERT	NET_OUT_ACCESS_FACE_SERVICE_INSERT
NET_EM_ACCESS_CTL_FACE_SERVICE_GET	获取人脸信息	NET_IN_ACCESS_FACE_SERVICE_GET	NET_OUT_ACCESS_FACE_SERVICE_GET
NET_EM_ACCESS_CTL_FACE_SERVICE_UPDATE	更新人脸信息	NET_IN_ACCESS_FACE_SERVICE_UPDATE	NET_OUT_ACCESS_FACE_SERVICE_UPDATE
NET_EM_ACCESS_CTL_FACE_SERVICE_REMOVE	删除人脸信息	NET_IN_ACCESS_FACE_SERVICE_REMOVE	NET_OUT_ACCESS_FACE_SERVICE_REMOVE
NET_EM_ACCESS_CTL_FACE_SERVICE_CLEAR	清空人脸信息	NET_IN_ACCESS_FACE_SERVICE_CLEAR	NET_OUT_ACCESS_FACE_SERVICE_CLEAR

3.4.5.4 指纹管理

3.4.5.4.1 门禁指纹信息管理接口 CLIENT_OperateAccessFingerprintService

表3-51 门禁指纹信息管理接口说明

选项	说明
描述	门禁指纹信息管理接口

选项	说明	
函数	public boolean CLIENT_OperateAccessFingerprintService(LLong lLoginID, int emtype, Pointer pstInParam, Pointer pstOutParam, int nWaitTime);	
参数	[in] lLoginID	登录句柄
	[in] emtype	指纹信息操作类型
	[in] pInBuf	指纹信息管理（入参）
	[out] pOutBuf	指纹信息管理（出参）
	[in] nWaitTime	超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

emtype、pInBuf 和 pOutBuf 的对照关系，请参见表 3-52。

表3-52 nType、pInBuf 和 pOutBuf 对照关系

emtype	描述	pInBuf	pOutBuf
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE_INSERT	添加指纹信息	NET_IN_ACCESS_FINGERPRINT_SERVICE_INSERT	NET_OUT_ACCESS_FINGERPRINT_SERVICE_INSERT
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE_GET	获取指纹信息	NET_IN_ACCESS_FINGERPRINT_SERVICE_GET	NET_OUT_ACCESS_FINGERPRINT_SERVICE_GET
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE_UPDATE	更新指纹信息	NET_IN_ACCESS_FINGERPRINT_SERVICE_UPDATE	NET_OUT_ACCESS_FINGERPRINT_SERVICE_UPDATE
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE_REMOVE	删除指纹信息	NET_IN_ACCESS_FINGERPRINT_SERVICE_REMOVE	NET_OUT_ACCESS_FINGERPRINT_SERVICE_REMOVE
NET_EM_ACCESS_CTL_FINGERPRINT_SERVICE_CLEAR	清空指纹信息	NET_IN_ACCESS_FINGERPRINT_SERVICE_CLEAR	NET_OUT_ACCESS_FINGERPRINT_SERVICE_CLEAR

3.4.6 门配置

3.4.6.1 门配置信息

3.4.6.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.4.6.1.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.4.6.1.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.4.6.1.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.4.7 门时间配置

3.4.7.1 时段配置

3.4.7.1.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.4.7.1.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.4.7.1.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.4.7.1.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.4.7.2 常开常闭时段配置

3.4.7.2.1 解析查询到的配置信息 CLIENT_ParseData

关于 CLIENT_ParseData 的详细介绍，请参见“3.3.2.2 解析查询到的配置信息 CLIENT_ParseData”。

3.4.7.2.2 查询配置信息 CLIENT_GetNewDevConfig

关于 CLIENT_GetNewDevConfig 的详细介绍，请参见“3.3.3.1.2 查询配置信息 CLIENT_GetNewDevConfig”。

3.4.7.2.3 设置配置信息 CLIENT_SetNewDevConfig

关于 CLIENT_SetNewDevConfig 的详细介绍，请参见“3.3.3.1.3 设置配置信息 CLIENT_SetNewDevConfig”。

3.4.7.2.4 打包字符串格式 CLIENT_PacketData

关于 CLIENT_PacketData 的详细介绍，请参见“3.3.3.1.4 打包字符串格式 CLIENT_PacketData”。

3.4.7.3 假日组

3.4.7.3.1 获取假日组接口 CLIENT_GetConfig

表3-53 获取假日组接口说明

选项	说明	
描述	获取假日组接口	
函数	public boolean CLIENT_GetConfig(LLong lLoginID, int emCfgOpType, int nChannelID, Pointer szOutBuffer, int dwOutBufferSize, int waittime, Pointer reserve);	
参数	[in] lLoginID	登录句柄
	[in] emCfgOpType	设置配置信息的类型 假日组配置： NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP
	[in] nChannelID	通道号
	[out] szOutBuffer	获取配置信息的缓存地址
	[in] dwOutBufferSize	缓存地址大小
	[in] waittime	超时时间
	[in] reserve	保留参数
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

表3-54 获取假日组信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szOutBuffer	dwOutBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP	获取假日组信息	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO 结构体尺寸

3.4.7.3.2 设置假日组接口 CLIENT_SetConfig

表3-55 设置假日组接口说明

选项	说明
描述	设置假日组接口

选项	说明	
函数	public boolean CLIENT_SetConfig(LLong lLoginID, int emCfgOpType, int nChannelID, Pointer szInBuffer, int dwInBufferSize, int waittime, IntByReference restart, Pointer reserve);	
参数	[in] lLoginID	登录句柄
	[in] emCfgOpType	设置配置的类型 假日组配置： NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP
	[in] nChannelID	通道号
	[in] szInBuffer	配置的缓存地址
	[in] dwInBufferSize	缓存地址大小
	[in] waittime	超时时间
	[in] restart	是否需要重启
	[in] reserve	保留参数
返回值	成功返回 TRUE，失败返回 FALSE	
说明	无	

表3-56 设置假日组信息 mCfgOpType 参数赋值说明

emCfgOpType	描述	szInBuffer	dwInBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAY_GROUP	设置假日组信息	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO	NET_CFG_ACCESSCTL_SPECIALDAY_GROUP_INFO 结构体尺寸

3.4.7.4 假日计划

关于接口的详细介绍，请参见“3.4.7.3 假日组”。

表3-57 获取假日计划信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szOutBuffer	dwOutBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE	获取计划信息	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO 结构体尺寸

表3-58 设置假日计划信息 emCfgOpType 参数赋值说明

emCfgOpType	描述	szInBuffer	dwInBufferSize
NET_EM_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE	设置假日计划信息	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO	NET_CFG_ACCESSCTL_SPECIALDAYS_SCHEDULE_INFO 结构体尺寸

3.4.8 门高级配置

请参见“3.3.8 门高级配置”。

3.4.9 记录查询

3.4.9.1 开门记录

请参见“3.3.9.1 开门记录”。

3.4.9.2 报警记录

3.4.9.2.1 查找记录条数 CLIENT_QueryRecordCount

表3-59 查询记录条数说明

选项	说明	
描述	查找记录条数	
函数	public boolean CLIENT_QueryRecordCount(NET_IN_QUEYT_RECORD_COUNT_PARAM pInParam, NET_OUT_QUEYT_RECORD_COUNT_PARAM pOutParam, int waittime);	
参数	[in] pInParam	查询记录条数入参，pInParam ->lFindeHandle 为 CLIENT_FindRecord 的 pOutParam-> lFindeHandle
	[out] pOutParam	查询记录条数出参，pOutParam ->nRecordCount 为记录个数
	[in] waittime	查询超时时间
返回值	成功返回 TRUE，失败返回 FALSE	
说明	调用本接口之前应先调用 CLIENT_FindRecord 以打开查询句柄	

3.4.9.2.2 按查询条件查询记录 CLIENT_FindRecord

表3-60 按查询条件查询记录说明

选项	说明	
描述	按查询条件查询记录	
函数	public boolean CLIENT_FindRecord(LLong lLoginID, NET_IN_FIND_RECORD_PARAM pInParam, NET_OUT_FIND_RECORD_PARAM pOutParam, int waittime);	
参数	[in]lLoginID	设备登录句柄
	[in] pInParam	查询日志的参数，参数赋值说明请参见表 3-33
	[out] pOutParam	查询日志的出参，返回查询句柄
	[in] waittime	等待超时时间

选项	说明
返回值	成功返回 TRUE，失败返回 FALSE
说明	可以先调用本接口获得查询句柄，再调用 CLIENT_FindNextRecord 函数获取记录列表，查询完毕可以调用 CLIENT_FindRecordClose 关闭查询句柄

表3-61 查询开门记录入参数说明

pInParam 结构体	赋值	说明
emType	NET_RECORD_ACCESS_ALARMRECORD	用于报警记录查询

3.4.9.2.3 查找记录 CLIENT_FindNextRecord

表3-62 查找记录说明

选项	说明	
描述	查找记录:nFilecount:需要查询的条数，返回值为媒体文件条数返回值小于 nFilecount 则相应时间段内的文件查询完毕	
函数	public boolean CLIENT_FindNextRecord(NET_IN_FIND_NEXT_RECORD_PARAM pInParam, NET_OUT_FIND_NEXT_RECORD_PARAM pOutParam, int waittime);	
参数	[in] pInParam	查询记录入参，pInParam ->lFindeHandle 为 CLIENT_FindRecord 的 pOutParam-> lFindeHandle
	[out] pOutParam	查询记录出参返回记录信息
	[in] waittime	等待超时时间
返回值	1：成功取回一条记录，0：记录已取完，-1：参数出错	
说明	无	

表3-63 查询开门记录出参说明

pOutParam 结构体	赋值	说明
pRecordList	NET_RECORD_ACCESS_ALARMRECORD_INFO	用于报警记录查询

3.4.9.2.4 结束记录查询 CLIENT_FindRecordClose

表3-64 结束记录查询说明

选项	说明	
描述	结束记录查询	
函数	public boolean CLIENT_FindRecordClose(LLong lFindHandle);	
参数	[in] lFindHandle	CLIENT_FindRecord 的返回值
返回值	成功返回 TRUE，失败返回 FALSE	
说明	调用 CLIENT_FindRecord 打开查询句柄，查询完毕后应调用本函数以关闭查询句柄	

第 4 章 回调函数

4.1 断线回调函数 fDisConnect

表4-1 断线回调函数说明

选项	说明	
描述	断线回调函数	

函数	public interface fDisConnect extends StdCallCallback {
	public void invoke(
	LLong lLoginID,
	String pchDVRIP, int nDVRPort,
	Pointer dwUser);
	}	

参数	[out]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
	[out]pchDVRIP	断线的设备 IP
	[out]nDVRPort	断线的设备端口
	[out]dwUser	回调函数的用户参数
返回值	无	
说明	无	

4.2 断线重连回调函数 fHaveReConnect

表4-2 断线重连回调函数说明

选项	说明	
描述	断线重连回调函数	

函数	public interface fHaveReConnect extends StdCallCallback {
	public void invoke(
	LLong lLoginID,
	String pchDVRIP, int nDVRPort,
	Pointer dwUser);
	}	

参数	[out]lLoginID	CLIENT_LoginWithHighLevelSecurity 的返回值
	[out]pchDVRIP	断线后重连成功的设备 IP
	[out]nDVRPort	断线后重连成功的设备端口
	[out]dwUser	回调函数的用户参数
返回值	无	
说明	无	

4.3 实时预览数据回调函数 fRealDataCallBackEx2

表4-3 实时预览数据回调函数说明

选项	说明	
描述	实时预览数据回调函数	

函数	public interface fRealDataCallBackEx2 extends SDKCallback{
	void invoke(
	LLong lRealHandle,
	int dwDataType,
	Pointer pBuffer,
	int dwBufSize,
	LLong param,
	Pointer dwUser);
	}	

参数	[out]lRealHandle	CLIENT_RealPlayEx 的返回值
	[out]dwDataType	数据类型 ● 0 表示原始数据 ● 1 表示带有帧信息的数据 ● 2 表示 YUV 数据 ● 3 表示 PCM 音频数据
	[out]pBuffer	预览数据块地址
	[out]dwBufSize	预览数据块的长度，单位：字节
	[out]param	回调数据参数结构体，dwDataType 值不同类型不同。 ● dwDataType 为 0 时，param 为空指针 ● dwDataType 为 1 时，param 为 tagVideoFrameParam 结构体指针 ● dwDataType 为 2 时，param 为 tagCBYUVDataParam 结构体指针 ● dwDataType 为 3 时，param 为 tagCBPCMDataParam 结构体指针
	[out]dwUser	回调函数的用户参数
返回值	无	
说明	无	

4.4 音频数据回调函数 pfAudioDataCallBack

表4-4 音频数据回调函数说明

选项	说明
描述	语音对讲的音频数据回调函数

选项	说明	

函数	public interface pfAudioDataCallBack extends StdCallCallback {
	public void invoke(
	LLong lTalkHandle,
	Pointer pDataBuf, int dwBufSize,
	byte byAudioFlag, Pointer dwUser);
	}	

参数	[out]lTalkHandle	CLIENT_StartTalkEx 的返回值
	[out]pDataBuf	音频数据块地址
	[out]dwBufSize	音频数据块的长度，单位：字节
	[out]byAudioFlag	数据类型标志 ● 0 表示来自本地采集 ● 1 表示来自设备发送
	[out]dwUser	回调函数的用户参数
返回值	无	
说明	无	

4.5 报警回调函数 fMessCallBack

表4-5 报警回调函数说明

选项	说明	
描述	报警回调函数	

函数	public interface fMessCallBack extends SDKCallback{
	public boolean invoke(
	int lCommand,
	LLong lLoginID,
	Pointer pStuEvent,
	int dwBufLen,
	String strDeviceIP,
	NativeLong nDevicePort,
	Pointer dwUser);
	}	

参数	[out]lCommand	报警类型，具体请参见表 4-6
	[out]lLoginID	登录接口返回值
	[out]pBuf	接收报警数据的缓存，根据调用的侦听接口和 lCommand 值不同，填充的数据不同
	[out]dwBufLen	pBuf 的长度，单位：字节
	[out]pchDVRIP	设备 IP
	[out]nDVRPort	端口
	[out]dwUser	用户自定义数据
返回值	成功返回 TRUE，失败返回 FALSE	
说明	一般在应用程序初始化时调用设置回调，在回调函数中根据不同的设备 ID 和命令值做出不同的处理	

表4-6 报警类型和结构体对应关系

报警所属业务	报警类型名称	lCommand	pBuf
报警主机	本地报警事件	NET_ALARM_ALARM_EX2	ALARM_ALARM_INFO_EX2
	电源故障事件	NET_ALARM_POWERFAULT	ALARM_POWERFAULT_INFO
	防拆事件	NET_ALARM_CHASSISINTRUDED	ALARM_CHASSISINTRUDED_INFO
	扩展报警输入通道事件	NET_ALARM_ALARMEXTENDED	ALARM_ALARMEXTENDED_INFO
	紧急事件	NET_URGENCY_ALARM_EX	数据为 16 个字节数组，每个字节表示一个通道状态 ● 1 为有报警 ● 0 为无报警
	蓄电池低电压事件	NET_ALARM_BATTERYLOWPOWER	ALARM_BATTERYLOWPOWER_INFO
	设备邀请平台对讲事件	NET_ALARM_TALKING_INVITE	ALARM_TALKING_INVITE_INFO
	设备布防模式变化事件	NET_ALARM_ARMMODE_CHANGE_EVENT	ALARM_ARMMODE_CHANGE_INFO
	防区旁路状态变化事件	NET_ALARM_BYPASSMODE_CHANGE_EVENT	ALARM_BYPASSMODE_CHANGE_INFO
	报警输入源信号事件	NET_ALARM_INPUT_SOURCE_SIGNAL	ALARM_INPUT_SOURCE_SIGNAL_INFO
	消警事件	NET_ALARM_ALARMCLEAR	ALARM_ALARMCLEAR_INFO
	子系统状态改变事件	NET_ALARM_SUBSYSTEM_STATE_CHANGE	ALARM_SUBSYSTEM_STATE_CHANGE_INFO
	扩展模块掉线事件	NET_ALARM_MODULE_LOST	ALARM_MODULE_LOST_INFO
	PSTN 掉线事件	NET_ALARM_PSTN_BREAK_LINE	ALARM_PSTN_BREAK_LINE_INFO
	模拟量报警事件	NET_ALARM_ANALOG_PULSE	ALARM_ANALOGPULSE_INFO
	报警传输事件	NET_ALARM_PROFILE_ALARM_TRANSMIT	ALARM_PROFILE_ALARM_TRANSMIT_INFO
	无线设备低电量报警事件	NET_ALARM_WIRELESSDEV_LOWPOWER	ALARM_WIRELESSDEV_LOWPOWER_INFO
	防区布撤防状态改变事件	NET_ALARM_DEFENCE_ARMMODE_CHANGE	ALARM_DEFENCE_ARMMODECHANGE_INFO
	子系统布撤防状态改变事件	NET_ALARM_SUBSYSTEM_ARMMODE_CHANGE	ALARM_SUBSYSTEM_ARMMODECHANGE_INFO
	探测器异常报警	NET_ALARM_SENSOR_ABNORMAL	ALARM_SENSOR_ABNORMAL_INFO
	病人活动状态报警事件	NET_ALARM_PATIENTDETECTION	ALARM_PATIENTDETECTION_INFO

报警所属业务	报警类型名称	lCommand	pBuf
门禁	门禁事件	NET_ALARM_ACCESS_CTL_EVENT	ALARM_ACCESS_CTL_EVENT_INFO
	门禁未关事件详细信息	NET_ALARM_ACCESS_CTL_NOT_CLOSE	ALARM_ACCESS_CTL_NOT_CLOSE_INFO
	闯入事件详细信息	NET_ALARM_ACCESS_CTL_BREAK_IN	ALARM_ACCESS_CTL_BREAK_IN_INFO
	反复进入事件详细信息	NET_ALARM_ACCESS_CTL_REPEAT_ENTER	ALARM_ACCESS_CTL_REPEAT_ENTER_INFO
	恶意开门事件	NET_ALARM_ACCESS_CTL_MALICIOUS	ALARM_ACCESS_CTL_MALICIOUS
	胁迫卡刷卡事件详细信息	NET_ALARM_ACCESS_CTL_DURESS	ALARM_ACCESS_CTL_DURESS_INFO
	多人组合开门事件	NET_ALARM_OPENDOORGROUP	ALARM_OPEN_DOOR_GROUP_INFO
	防拆事件	NET_ALARM_CHASSISINTRUDED	ALARM_CHASSISINTRUDED_INFO
	本地报警事件	NET_ALARM_ALARM_EX2	ALARM_ALARM_INFO_EX2
	门禁状态事件	NET_ALARM_ACCESS_CTL_STATUS	ALARM_ACCESS_CTL_STATUS_INFO
	锁舌报警	NET_ALARM_ACCESS_CTL_STATUS	ALARM_ACCESS_CTL_STATUS_INFO
	获取指纹事件	NET_ALARM_FINGER_PRINT	ALARM_CAPTURE_FINGER_PRINT_INFO
可视对讲	直连情况下，呼叫无答应事件	NET_ALARM_CALL_NO_ANSWERED	NET_ALARM_CALL_NO_ANSWERED_INFO
	手机号码上报事件	NET_ALARM_TELEPHONE_CHECK	ALARM_TELEPHONE_CHECK_INFO
	VTS 状态上报	NET_ALARM_VTSTATE_UPDATE	ALARM_VTSTATE_UPDATE_INFO
	VTO 目标检测	NET_ALARM_ACCESSIDENTIFY	NET_ALARM_ACCESSIDENTIFY
	设备请求对方发起对讲事件	NET_ALARM_TALKING_INVITE	ALARM_TALKING_INVITE_INFO
	设备取消对讲请求事件	NET_ALARM_TALKING_IGNORE_INVITE	ALARM_TALKING_IGNORE_INVITE_INFO
	设备主动挂断对讲事件	NET_ALARM_TALKING_HANGUP	ALARM_TALKING_HANGUP_INFO
	雷达监测超速报警事件	NET_ALARM_RADAR_HIGH_SPEED	ALARM_RADAR_HIGH_SPEED_INFO

4.6 升级进度回调函数 fUpgradeCallBackEx

表4-7 升级进度回调函数说明

选项	说明	
描述	升级进度回调函数	

函数	public interface fUpgradeCallBackEx extends SDKCallback {
	public void invoke(
	LLong lLoginID,
	LLong lUpgradechannel,
	int nTotalSize,
	int nSendSize,
	Pointer dwUserData);
	}	

参数	[out]lLoginID	登录接口返回值
	[out] lUpgradechannel	CLIENT_StartUpgradeEx2 返回的升级句柄 ID
	[out] nTotalSize	升级文件的总长度(单位:字节)
	[out] nSendSize	已发送的文件长度(单位:字节)，为-1 时，表示发送升级文件结束
	[out]dwUser	用户自定义数据
返回值	无	
说明	升级设备程序回调函数原形支持 G 以上升级文件 nTotalSize = 0, nSendSize = -1 表示升级完成 nTotalSize = 0, nSendSize = -2 表示升级出错 nTotalSize = 0, nSendSize = -3 用户没有权限升级 nTotalSize = 0, nSendSize = -4 升级程序版本过低 nTotalSize = -1, nSendSize = XX 表示升级进度 nTotalSize = XX, nSendSize = XX 表示升级文件发送进度	

4.7 智能事件回调函数 fAnalyzerDataCallBack

表4-8 智能事件回调函数 fAnalyzerDataCallBack

选项	说明	
描述	远程设备状态回调函数	

函数	public interface fAnalyzerDataCallBack extends Callback {
	public int invoke(LLong lAnalyzerHandle, int dwAlarmType, Pointer pAlarmInfo, Pointer pBuffer, int dwBufSize, Pointer dwUser, int nSequence, Pointer reserved);
	}	

参数	[out]lAnalyzerHandle	CLIENT_RealLoadPictureEx 返回值
	[out]dwEventType	智能事件类型
	[out] pAlarmInfo	事件信息缓存
	[out]pBuffer	图片缓存
	[out]dwBufSize	图片缓存大小
	[out]dwUser	用户数据
	[out]nSequence	序列号
	[out]reserved	保留
返回值	无	
说明	订阅远程设备智能事件后，如果前端设备有相关的智能事件，会上报发生事件的相关信息	

附录1 法律声明

商标声明

- HDMI™ HIGH-DEFINITION MULTIMEDIA INTERFACE：本声明适用所有产品。如本产品使用 HDMI 技术，词语 HDMI、HDMI High-Definition Multimedia Interface（高清晰度多媒体接口）、HDMI 商业外观和 HDMI 徽标均为 HDMI Licensing Administrator, Inc.的商标或注册商标。本产品已经获得 HDMI Licensing Administrator, Inc.授权使用 HDMI 技术。
- VGA 是 IBM 公司的商标。
- Windows 标识和 Windows 是微软公司的商标或注册商标。
- 在本文档中可能提及的其他商标或公司的名称，由其各自所有者拥有。

责任声明

- 在适用法律允许的范围内，在任何情况下，本公司都不对因本文档中相关内容及描述的产品而产生任何特殊的、附随的、间接的、继发性的损害进行赔偿，也不对任何利润、数据、商誉、文档丢失或预期节约的损失进行赔偿。
- 本文档中描述的产品均“按照现状”提供，除非适用法律要求，本公司对文档中的所有内容不提供任何明示或暗示的保证，包括但不限于适销性、质量满意度、适合特定目的、不侵犯第三方权利等保证。

隐私保护提醒

您安装了我们的产品，您可能会采集人脸、指纹、车牌等个人信息。在使用产品过程中，您需要遵守所在地区或国家的隐私保护法律法规要求，保障他人的合法权益。如，提供清晰、可见的标牌，告知相关权利人视频监控区域的存在，并提供相应的联系方式。

关于本文档

- 本文档供多个型号产品使用，产品外观和功能请以实物为准。
- 如果不按照本文档中的指导进行操作而造成的任何损失由使用方自己承担。
- 本文档会实时根据相关地区的法律法规更新内容，具体请参见产品的纸质、电子光盘、二维码或官网，如果纸质与电子档内容不一致，请以电子档为准。
- 本公司保留随时修改本文档中任何信息的权利，修改的内容将会在本文档的新版本中加入，恕不另行通知。
- 本文档可能包含技术上不准确的地方、或与产品功能及操作不相符的地方、或印刷错误，以公司最终解释为准。
- 如果获取到的 PDF 文档无法打开，请使用最新版本或最主流的阅读工具。

附录2 网络安全建议

保障设备基本网络安全的必须措施：

1. **使用复杂密码**

   请参考如下建议进行密码设置：

   - 长度不小于 8 个字符。
   - 至少包含两种字符类型，字符类型包括大小写字母、数字和符号。
   - 不包含账户名称或账户名称的倒序。
   - 不要使用连续字符，如 123、abc 等。
   - 不要使用重叠字符，如 111、aaa 等。

2. **及时更新固件和客户端软件**

   - 按科技行业的标准作业规范，设备（如 NVR、DVR 和 IP 摄像机等）的固件需要及时更新至最新版本，以保证设备具有最新的功能和安全性。设备接入公网情况下，建议开启在线升级自动检测功能，便于及时获知厂商发布的固件更新信息。
   - 建议您下载和使用最新版本客户端软件。

增强设备网络安全的建议措施：

1. **物理防护**

   建议您对设备（尤其是存储类设备）进行物理防护，比如将设备放置在专用机房、机柜，并做好门禁权限和钥匙管理，防止未经授权的人员进行破坏硬件、外接设备（例如 U 盘、串口）等物理接触行为。

2. **定期修改密码**

   建议您定期修改密码，以降低被猜测或破解的风险。

3. **及时设置、更新密码重置信息**

   设备支持密码重置功能，为了降低该功能被攻击者利用的风险，请您及时设置密码重置相关信息，包含预留手机号/邮箱、密保问题，如有信息变更，请及时修改。设置密保问题时，建议不要使用容易猜测的答案。

4. **开启账户锁定**

   出厂默认开启账户锁定功能，建议您保持开启状态，以保护账户安全。在攻击者多次密码尝试失败后，其对应账户及源 IP 将会被锁定。

5. **更改 HTTP 及其他服务默认端口**

   建议您将 HTTP 及其他服务默认端口更改为 1024～65535 间的任意端口，以减小被攻击者猜测服务端口的风险。

6. **使能 HTTPS**

   建议您开启 HTTPS，通过安全的通道访问 Web 服务。

7. **MAC 地址绑定**

   建议您在设备端将其网关设备的 IP 与 MAC 地址进行绑定，以降低 ARP 欺骗风险。

8. **合理分配账户及权限**

   根据业务和管理需要，合理新增用户，并合理为其分配最小权限集合。

9. **关闭非必需服务，使用安全的模式**

   如果没有需要，建议您关闭 SNMP、SMTP、UPnP 等功能，以降低设备面临的风险。

   如果有需要，强烈建议您使用安全的模式，包括但不限于：

   - SNMP：选择 SNMP v3，并设置复杂的加密密码和鉴权密码。
   - SMTP：选择 TLS 方式接入邮箱服务器。
   - FTP：选择 SFTP，并设置复杂密码。
   - AP 热点：选择 WPA2-PSK 加密模式，并设置复杂密码。

10. 音视频加密传输

如果您的音视频数据包含重要或敏感内容，建议启用加密传输功能，以降低音视频数据传输过程中被窃取的风险。

11. 安全审计

- 查看在线用户：建议您不定期查看在线用户，识别是否有非法用户登录。
- 查看设备日志：通过查看日志，可以获知尝试登录设备的 IP 信息，以及已登录用户的关键操作信息。

12. 网络日志

由于设备存储容量限制，日志存储能力有限，如果您需要长期保存日志，建议您启用网络日志功能，确保关键日志同步至网络日志服务器，便于问题回溯。

13. 安全网络环境的搭建

为了更好地保障设备的安全性，降低网络安全风险，建议您：

- 关闭路由器端口映射功能，避免外部网络直接访问路由器内网设备的服务。
- 根据实际网络需要，对网络进行划区隔离：若两个子网间没有通信需求，建议使用 VLAN、网闸等方式对其进行网络分割，达到网络隔离效果。
- 建立 802.1x 接入认证体系，以降低非法终端接入专网的风险。
- 开启设备 IP/MAC 地址过滤功能，限制允许访问设备的主机范围。