

0：SDK初始化

**public** **static** **final** String ***APP\_KEY*** = "4E70159C5A3533C842ECFEED65333DB9";//每个客户唯一的标识

**public** **static** **final** String ***irDeviceId*** = "xxx";

//在App的入口进行初始化, 在Application中初始化也可以

//按红外设备授权收费的客户，需要传递自己设备唯一的标识***irDeviceId*** ，使用KookongSDK.init(Context context, String key, String irDeviceId);

//irDeviceId是必须传入的，初始化之后sdk首次上传酷控服务器，酷控后台服务器会自动记录，总设备数量自动减一，

//此irDeviceId一般用红外设备的mac地址或者sn号之类的的唯一值做标识，这样也方便排产。

**boolean** result = KookongSDK.*init*(**this**, ***APP\_KEY***, ***irDeviceId***);

/\*

\* 服务器返回错误代码含义

\* 代码 意义

\* 0,"code 1" secret 错误

\* 0,"code 2" secret 不存在

\* 0,"code 3" 客户已被禁用

\* 0,"code 4" 客户已超期

\* 0,"code 6" 选择运营商超出数量限制

\* 0,"code 7" 无访问权限

\* 0,"code 8" 试用数超出限制

\* 0,"code 9" 设备数超出限制

\* 0,"code 10" 单台设备下载红外超出限制

\*/

1: 

**case** R.id.*setupflow\_get\_areaid*:

//获取指定城市的AreaId,输入的地点可由定位信息获得

/\* *getAreaId原型*

//**void** *getAreaId* (String province, String city, String //district, **final** IRequestResult<Integer> requestResult)

\*/

//注意 同一个省级别的运营商都是一样的所以获取运营商的时候同一个省内选//择随意一个地级市区就可以了。

KookongSDK.*getAreaId*("北京市", "北京市", "海淀区", **new** IRequestResult<Integer>()

{

@Override

**public** **void** onSuccess(String msg, Integer result) {

Logger.*d*("AreaId is : " + result);

}

@Override

**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

**break**;

//AreaId是代表县区的ID，可由这个ID获取运营商列表。

运行结果：



2: 

**case** R.id.*setupflow\_get\_operators*:

//获取指定AreaId下的运营商列表也就是spId

/\*getOperaters*原型*

//**void** getOperaters(**int** areaid, **final** IRequestResult<SpList> //requestResult)

\*/

//注意返回的type=0的是普通机顶盒，type=1的是IPTV，两者获取//remoteId的方式不同。

//注意 同一个省级别的运营商都是一样的所以获取运营商的时候同一个省内选//择随意一个地级市区就可以了。

KookongSDK.*getOperaters*(110108, **new** IRequestResult<SpList>()

{

@Override

**public** **void** onSuccess(String msg, SpList result) {

List<Sp> sps = result.spList;

**for** (**int** i = 0; i < sps.size(); i++) {

Logger.*d*("The sp is " + sps.get(i).spName);

}

}

@Override

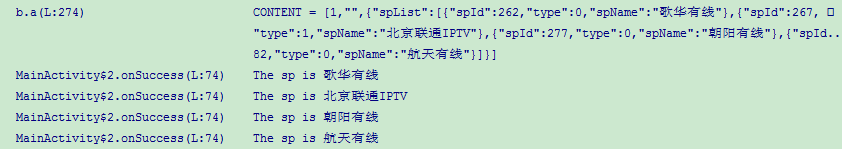
**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：



3: 

**case** R.id.*setupflow\_get\_stb\_remotes*:

//获取指定运营商, 指定AreaId下的遥控器id, brandid传0，

//也就是获取机顶盒的remoteId列表

/\* *getAllRemoteIds原型*

// **void** getAllRemoteIds(**int** deviceTypeId, **int** brandid, **int** //spId, **int** areaId, **final** IRequestResult<RemoteList> //requestResult)

\*/

//注意运营商是IPTV要单独用后面的getIPTV接扣不能使用这个接口。

KookongSDK.*getAllRemoteIds*(Device.*STB*, 0, 262, 110108, **new** IRequestResult<RemoteList>()

{

@Override

**public** **void** onSuccess(String msg, RemoteList result) {

List<Integer> remoteids = result.rids;

String res = Arrays.*toString*(remoteids.toArray());

Logger.*d*("remoteids: " + res);

}

@Override

**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：



4：

**case** R.id.***setupflow\_test\_remotedata***://获取的是某套红外码部分按键，仅限按红外设备授权的客户（初始化时需要传递的deviceId），如果初始化时没有传deviceId，请求接口时会提示参数错误

//对码流程获取rid = 4162 和 10687的 红外码, 批量获取红外码的方式是逗号隔开

KookongSDK.*testIRDataById*("4162,10687", Device.***STB***, **new** IRequestResult<IrDataList>()

{

@Override

**public** **void** onSuccess(String msg, IrDataList result) {

List<IrData> irDatas = result.getIrDataList();

**for** (**int** i = 0; i < irDatas.size(); i++) {

Logger.*d*("The rid is " + irDatas.get(i).rid);

}

}

@Override

**public** **void** onFail(Integer errorCode, String msg) {

//按红外设备授权的客户，才会用到这个指

**if** (errorCode == AppConst.***CUSTOMER\_DEVICE\_NUM\_LIMIT***) {//设备总数超过了授权的额度

msg = "设备总数超过了授权的额度";

}

TipsUtil.*toast*(MainActivity.**this**, msg);

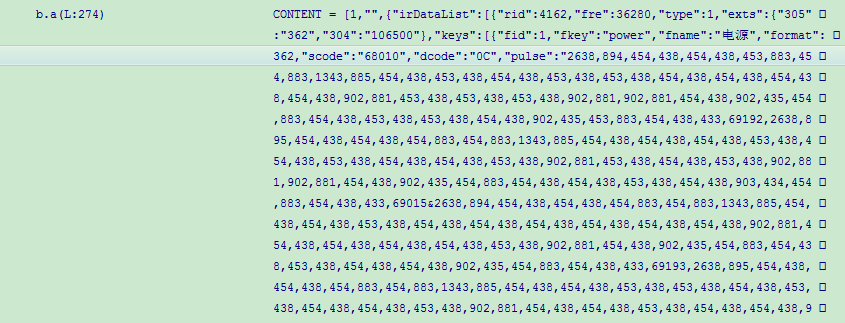
}

});

});

运行结果：会有remoteId，fre频率，type模式（1代表是固定值，2代表是大部分空调码需要手机SDK来解码），keys按键信息，pulse红外时间数码等。其中这个fid就是电源键的按定义键值为1。

这是遥控器对码，服务器返回的是酷控app主面板的按键，用于对码测试，对码成功后获取完整红外按键数据请使用KookongSDK.*getIRDataById()方法*



5：

**case** R.id.***setupflow\_get\_remotedata***://这是获取完整红外码的方法

//获取rid = 4162 和 10687的 红外码, 批量获取红外码的方式是逗号隔开

KookongSDK.*getIRDataById*("4162", Device.***STB***, **new** IRequestResult<IrDataList>()

{

@Override

**public** **void** onSuccess(String msg, IrDataList result) {

List<IrData> irDatas = result.getIrDataList();

**for** (**int** i = 0; i < irDatas.size(); i++) {

Logger.*d*("The rid is " + irDatas.get(i).rid);

}

}

@Override

**public** **void** onFail(Integer errorCode, String msg) {

//按红外设备授权的客户，才会用到这两个值

**if**(errorCode==AppConst.***CUSTOMER\_DEVICE\_REMOTE\_NUM\_LIMIT***){//同一个设备下载遥控器超过了50套限制

msg = "下载的遥控器超过了套数限制";

}**else** **if**(errorCode==AppConst.***CUSTOMER\_DEVICE\_NUM\_LIMIT***){//设备总数超过了授权的额度

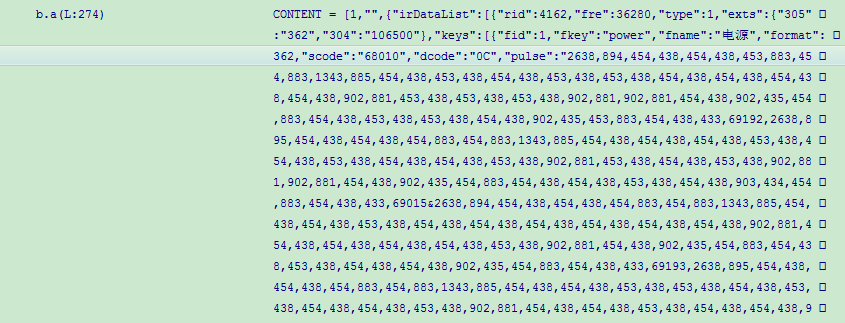
msg="设备总数超过了授权的额度";

}

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});运行结果：会有remoteId，fre频率，type模式（1代表是固定值，2代表是大部分空调码需要手机SDK来解码），keys按键信息，pulse红外时间数码等。其中这个fid就是电源键的按定义键值为1。



6: 

**case** R.id.*setupflow\_get\_remotedata\_allparse*://特殊客户调用的下载空调红外码（有状态的空调，IrData.type=2）

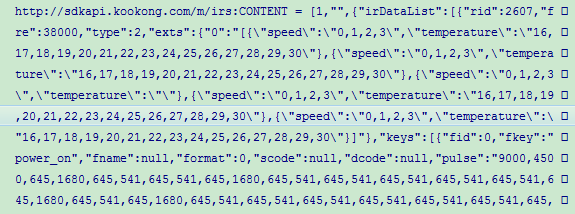
//获取rid = 2607的 红外码, 批量获取红外码的方式是逗号隔开

KookongSDK.*getNoStateIRDataById*("2607", Device.*AC*,**true**, **new** IRequestResult<IrDataList>()

//注意全解的状态里 发完开机后要再发一组模式温度风量的状态才能同步空调状态。因为

//开机码里的模式风量温度只包含了一组。所以要同步下状态。

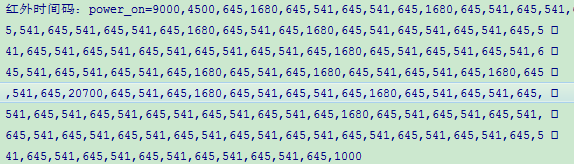
运行结果：会得到这套码的模式，温度，风量3种状态组合全解的红外码，可以脱离SDK使用，rid就是remoteId，fre是频率，exts字段对应的是5个数组，每个数组里从下标0,1,2,3,4分别代表制冷，制热，自动，送风，除湿。如果没有这个模式或者这个模式没有温度，风量则{}里就会是空。



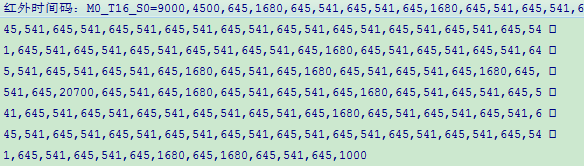
之后就是此套码含有的模式(exts的下标)0：制冷，1：制热，2：自动，3：送风，4除湿，后面是每种模式含有的风速和温度。风速0：自动，1：低，2：中，3：高。然后就是每种组合状态的按键参数代表一个按键，



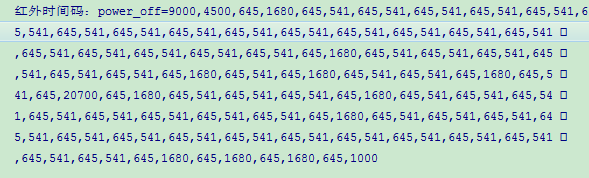
power\_on：开机



M0\_T16\_S0：制冷模式，16度，自动风量



power\_off：关机



7: 

**case** R.id.*setupflow\_get\_iptv*:

//根据spid 获取IPTV列表

/\* getFilterIRData函数原型 spId在获取运营商列表时会得到

// **public** **static** **void** getIPTV(**final** **int** spId, **final** //IRequestResult<StbList> requestResult)

\*/

//IPTV只能用这种获取remoteId，如果用普通机顶盒的方式获取结果可能是错误的

KookongSDK.*getIPTV*(267, **new** IRequestResult<StbList>()

{

@Override

**public** **void** onSuccess(String msg, StbList result) {

List<Stb> stbs = result.stbList;

**for** (**int** i = 0; i < stbs.size(); i++) {

Logger.*d*("The Stb is " + stbs.get(i).bname);

}

}

@Override

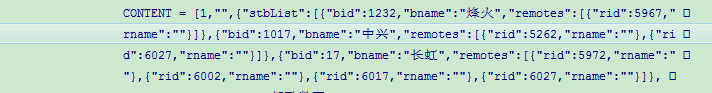
**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：stblist 机顶盒列表 bid是品牌id，rid就是remoteId



8：:

**case** R.id.*setupflow\_searchstb*:

//关键词是为 "华" 的 机顶盒列表

/\* *searchSTB函数原型*

//**public** **static** **void** searchSTB(**final** String name, **final** **int** //areaid, **final** IRequestResult<StbList> requestResult) {

\*/

KookongSDK.*searchSTB*("华", 110108, **new** IRequestResult<StbList>()

{

@Override

**public** **void** onSuccess(String msg, StbList result) {

List<Stb> stbs = result.stbList;

**for** (**int** i = 0; i < stbs.size(); i++) {

Logger.*d*("The Stb is " + stbs.get(i).bname);

}

}

@Override

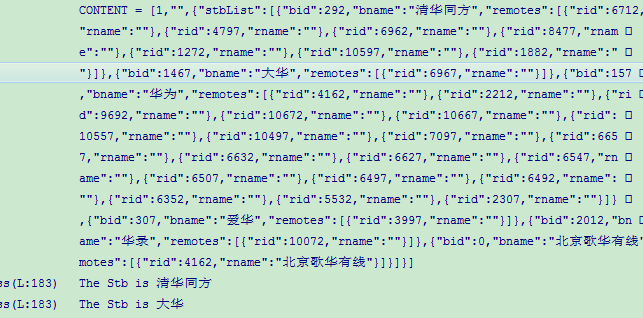
**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：



9: 

**case** R.id.*setupflow\_get\_brandList*:

//获取电视机, 空调等, (除STB以外) 的设备品牌列表

/\* *getBrandListFromNet函数原型*

*//* **public** **static** **void** getBrandListFromNet(**int** deviceTypeId, //**final** IRequestResult<BrandList> requestResult)

\*/

/\* DeviceId如下

public int STB = 1;

public int TV = 2;

public int BOX = 3;

public int DVD = 4;

public int AC = 5;

public int PROJECTOR = 6;

public int AV = 7;

public int FAN = 8;

public int SLR\_CAMERA = 9;

public int LIGHT = 10;

\*/

KookongSDK.*getBrandListFromNet*(Device.*TV*, **new** IRequestResult<BrandList>()

{

@Override

**public** **void** onSuccess(String msg, BrandList result) {

List<Brand> stbs = result.brandList;

**for** (**int** i = 0; i < stbs.size(); i++) {

Logger.*d*("The Brand is " + stbs.get(i).cname);

}

}

@Override

**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：brandList有brandId品牌id，首位大写字母，中文和英文名称

前十个就是常用品牌。



10：

**case** R.id.*setupflow\_get\_ir\_by\_brand*:

//指定brand下的电视机所有的红外码的id spid和areaid都传0

//获取指定设备类型(机顶盒是spId和areaid,所以这里不是机顶盒的获取方//式，机顶盒的红外码都是按区域划分和品牌关系不大),指定品牌下的红外码

/\* *getAllRemoteIds函数原型*DeviceId 品牌id，spid和areaid都传0

//**public** **static** **void** getAllRemoteIds(**int** deviceTypeId, **int** //brandid, **int** spId, **int** areaId, **final** //RequestResult<RemoteList> requestResult)

\*/

KookongSDK.*getAllRemoteIds*(Device.*TV*, 967, 0, 0, **new** IRequestResult<RemoteList>()

{

@Override

**public** **void** onSuccess(String msg, RemoteList result) {

List<Integer> remoteids = result.rids;

String res = Arrays.*toString*(remoteids.toArray());

Logger.*d*("tv remoteids: " + res);

}

@Override

**public** **void** onFail(String msg) {

TipsUtil.*toast*(MainActivity.**this**, msg);

}

});

运行结果：



11: 



空调面板:

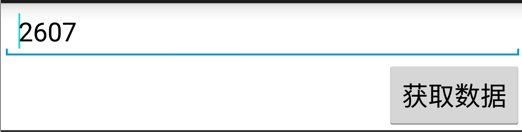
非空调设备的码值都是可以直接使用的。

空调设备则需要此SDK解码。

注意一小部分的空调是不带状态的固定值和电视等设备一样，

不需要解码，所以这一小部分可以直接使用。

1：空调初始化，输入remoteId后点击获取数据，这一套码就可以使用了。

 KookongSDK.*getIRDataById*(rid, **new** IRequestResult<IrDataList>()

{

@Override

**public** **void** onSuccess(String msg, IrDataList result) {

List<IrData> irDatas = result.getIrDataList();

mIrData = irDatas.get(0);

mKKACManager.initIRData(mIrData.rid, mIrData.exts, mIrData.keys);//根据空外数据初始化空调解码器

String acState = DataStoreUtil.*i*().getString("AC\_STATE", "");//获取以前保存过的空调状态

mKKACManager.setACStateV2FromString(acState);

updateACDisplay();//刷新空调面板

mACPannel.setVisibility(View.*VISIBLE*);

}

@Override

**public** **void** onFail(String msg) {

mACPannel.setVisibility(View.*INVISIBLE*);

}

});

2：按键配合状态mKKACManager解码后得到高低电平数组。

/\*\* 发送解析出来的红外码 \*/

**private** **void** sendIr() {

**int**[] patternsInArray = mKKACManager.getACIRPatternIntArray();//这些码可以直接给ConsumerIR发送出去

Log.*d*("IRPattern", Arrays.*toString*(patternsInArray));

}

3：空调主要有以下几个状态：

1. 电源状态：开机，关机
2. 模式状态：制冷，制热，自动，送风，除湿(有些空调没有其中的某个)
3. 温度状态：16-30度(有些空调会没有中的某个)
4. 风量状态：自动，低，中，高(有些空调会没有中的某个)
5. 风向状态：扫风，固定风向(会有N个风向) (有些空调会没有中的某个)
6. **public** **static** **final** **byte** *AC\_MODE\_COOL* = 0;//制冷
7. **public** **static** **final** **byte** *AC\_MODE\_HEAT* = 1;//制热
8. **public** **static** **final** **byte** *AC\_MODE\_AUTO* = 2;//自动
9. **public** **static** **final** **byte** *AC\_MODE\_FAN* = 3; //送风
10. **public** **static** **final** **byte** *AC\_MODE\_DRY* = 4; //除湿
11. **public** **static** **final** **byte** *AC\_WIND\_SPEED\_AUTO* = 0;

//自动风量

1. **public** **static** **final** **byte** *AC\_WIND\_SPEED\_LOW* = 1;

//小风量

1. **public** **static** **final** **byte** *AC\_WIND\_SPEED\_MEDIUM* = 2;

//中风量

1. **public** **static** **final** **byte** *AC\_WIND\_SPEED\_HIGH* = 3;

//高风量

1. **public** **static** **final** **byte** *AC\_POWER\_ON* = 0; //开机
2. **public** **static** **final** **byte** *AC\_POWER\_OFF* = 1; //关机
3. **public** **static** **final** **byte** *AC\_UD\_WIND\_DIRECT\_SWING* = 0; //上下风风向

4：退出SDK后，会保存各个状态 如下所示。

一般的, App需要像空调的遥控器一样, 保存以前操作的空调的状态. 这时候需要先从本地读取保存过的空调的状态(字符串类型, 可以保存到数据库或者文件中), 然后设置到ACManager中：mKKACManager.setACState(acState);

Save AC state: {"curModel":{"curTmp":26,"curWindSpeed":0,"highTmp":30,"lowTmp":16,"modelType":0,"tempCanControl":true,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["0","1","2","3"]},"curPowerState":1,"curUDDirect":0,"curUDDirectType":"UDDIRECT\_FULL","modelList":[{"curTmp":26,"curWindSpeed":0,"highTmp":30,"lowTmp":16,"modelType":0,"tempCanControl":true,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["0","1","2","3"]},{"curTmp":20,"curWindSpeed":0,"highTmp":30,"lowTmp":16,"modelType":1,"tempCanControl":true,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["0","1","2","3"]},{"curTmp":-1,"curWindSpeed":0,"highTmp":30,"lowTmp":16,"modelType":2,"tempCanControl":false,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["0","1","2","3"]},{"curTmp":24,"curWindSpeed":0,"highTmp":30,"lowTmp":16,"modelType":3,"tempCanControl":true,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["0","1","2","3"]},{"curTmp":-1,"curWindSpeed":1,"highTmp":30,"lowTmp":16,"modelType":4,"tempCanControl":false,"tmpList":["16","17","18","19","20","21","22","23","24","25","26","27","28","29","30"],"windSpeedCanControl":true,"windSpeedList":["1"]}],"udWindDirectList":[1]}

## 空调设备SDK：

空调的红外码有两种：

1-有状态的空调（一般遥控器上有个显示屏，发送时把显示屏上显示的全部状态：如温度26度、制冷模式、自动风速等相关状态生成波形，再发送出去，此为组合码）

2-无状态的空调（遥控器上没有显示屏，跟电视遥控器差不多，摁一个按键就发送一个键的码，此类为波形码）

如何判断红外码是哪中空调？

通过IrData类中的type字段区分：type=1为波形码，也就是无状态的空调；type=2为组合码，也就是有状态的空调，这种需要单独处理。

无状态空调的使用跟电视等设备是一样的在就不做描述，主要讲下有状态空调的API。

类定义：

KKACManagerV2：空调状态控制管理器，提供了接口操作空调的状态及生成波形发送出去

ACConstants：空调的功能按键定义

IrData：红外数据

### 接口定义：

KKACManagerV2提供的接口：

**public** **void** initIRData(**int** remoteId, HashMap<Integer, String> exts, ArrayList<IrKey> keys)

参数：

remoteId：红外码的Id，对应IrData.rid

Exts：空调的按键参数，对应IrData.exts

Keys：空调的扩展按键，对应IrData.keys

接口说明：

传入空调红外码的数据，初始化空调的解码器

**public** **void** setACStateV2FromString(String state)

参数：

State：保存的空调状态

接口说明：

将上次保存的空调状态同步到KKACManagerV2中，首次传空即可KKACManagerV2会根据空调红外数据生成默认状态。注意：一个空调对应一个state，即使添加了remoteId相同的两个空调设备，他们的state也要分开管理。

**public** String getACStateV2InString()

接口返回：

保存的空调状态

接口说明：

将空调目前的状态生成json返回，调用此接口的时机是空调最后一次操作后，客户保存此状态，下次打开空调时调用setACStateV2FromString将状态还原。

**public** **boolean** stateIsEmpty()

接口返回：

True：状态为空，false：状态不为空

接口说明：

判断KKACManagerV2管理的状态是否为空，如果为空，空调将不能做任何操作。

**public** **void** changePowerState()

接口说明：

更改空调的开关机状态，空调默认是关机，调用此接口后就是开机了，再调用又更改为关机。

**public** **int** getPowerState()

接口返回：

Int：空调的开关机状态

接口说明:

返回当前空调的开关机状态，值对应ACConstants中



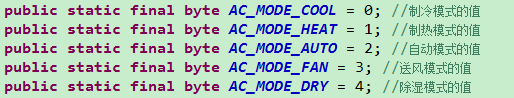
**public** **int** getCurModelType()

接口返回：

Int：空调模式

接口说明：

返回当前空调的模式，总共为五种模式：制冷、制热、自动、送风、除湿，值对应ACConstants中



**public** **void** changeACModel()

接口说明：

更改空调的模式状态，顺序更改：制冷->制热->自动->送风->除湿，部分空调可能只具备其中的几个模式。

**public** **boolean** isContainsTargetModel(**int** modelType)

参数：

modelType：对应ACConstants的模式的值

接口返回：

True:具备指定的模式，false：不具备指定的模式

接口说明：

传入指定模式的类型，查看此空调是否具备指定的模式

**public** **boolean** changeACTargetModel(**int** modelType)

参数：

modelType：对应ACConstants的模式的值

接口返回：

设置结果

接口说明：

根据传入的模式类型，设置为指定的模式，如果具备该模式则设置成功，反之失败。

**public** **boolean** isTempCanControl()

接口返回：

True:温度可调节，false：温度不可调节

接口说明：

判断当前模式下温度是否可以增加一度或减小一度，有些模式下空调的温度按温度加减是不会更改的。

**public** **int** getCurTemp()

接口返回：

空调温度值

接口说明：

获取当前模式下空调的温度状态，如果当前模式下温度不可调节，值为-1

**public** **boolean** setTargetTemp(**int** temp)

参数：

Temp：指定的温度

接口返回：

设置结果

接口说明：

根据传入的值设置空调的温度，设置成功返回true，如果当前模式下温度不能调节或传入的值不再温度范围内，则设置失败

**public** **int** increaseTmp()

接口返回：

增加后的温度值

接口说明：

温度增加一度，当增加到最大温度后，再调用此接口仍然返回最大值，如果当前模式下温度不可调节，值为-1

**public** **int** decreaseTmp()

接口返回：

减小后的温度值

接口说明：

温度减小一度，当减小到最小温度后，再调用此接口仍然返回最小值，如果当前模式下温度不可调节，值为-1

**public** **int** getMaxTemp()

接口返回：

温度最大值

接口说明：

返回当前模式下的温度最大值，温度默认范围为16-30，有些空调的最小和最大值会有所不同。如果当前模式下温度不可调节，值为-1

**public** **int** getMinTemp()

接口返回：

温度最小值

接口说明：

返回当前模式下的温度最小值，温度默认范围为16-30，有些空调的最小和最大值会有所不同。如果当前模式下温度不可调节，值为-1

**public** **boolean** isWindSpeedCanControl()

接口返回：

True:风速可调节，false:风速不可调节

接口说明:

判断当前模式下风速是否可以调节，空调在某些模式下，风速调节是没有作用的

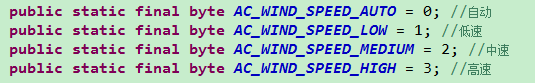
**public** **int** getCurWindSpeed()

接口返回：

空调的风速状态

接口说明：

返回当前空调的风速状态，返回值对应ACConstants中，如果当前模式下，风速不可调节，值为-1



**public** **int** changeWindSpeed()

接口返回：

更改后的风速的状态值

接口说明：

更改空调的风速状态，顺序更换：自动->低风速->中风速->高风速，某些模式下只能更改为部分风速，如果当前模式下风速不可调节，值为-1

**public** **boolean** setTargetWindSpeed(**int** targetSpeed)

参数：

设置的风速

接口返回：

设置结果

接口说明：

根据传入的值设置空调的风速，设置成功返回true，如果当前模式下风速不能调节或不在风速范围内，则设置失败

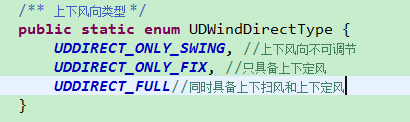
**public** UDWindDirectType getCurUDDirectType()

接口返回：

空调的上下风向类型

接口说明：

返回空调的上下风向类型：上下风向不可调节、只具备上下定风、上下扫风和上下定风都具备，值对应如下：



**public** **int** getCurUDDirect()

接口返回：

空调的上下风向状态

接口说明：

返回当前空调的上下风向状态，值为0是上下扫风，值大于0时为上下定风的不同风向，值为-1时为上下风向不能调节。不同的空调，具备的上下风向是不同。

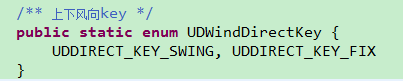
**public** **void** changeUDWindDirect(UDWindDirectKey udKey)

参数：

Udkey：设置上下风向的扫风还是定风

接口说明：

传入设置的是上下扫风还是上下定风，上下扫风传值UDWindDirectKey.UDDIRECT\_KEY\_SWING，上下定风传值UDWindDirectKey.UDDIRECT\_KEY\_FIX。如果当前风向状态已经是扫风，再传入UDWinDirectKey.UDDIRECT\_KEY\_SWING，风向就改为了上下定风；如果上下定风有几个不同的方向值，调用此接口会循环更改为不同的方向值。



**public** **boolean** setTargetUDWindDirect(**int** windDirect)

参数：

windDirect:风向值

接口返回：

设置结果

接口说明：

传入值设置空调的风向状态，值为0时为上下扫风，其他风向值时为上下定风，设置成功返回true

**public** List<Integer> getUDWindDirectList()

接口返回：

空调具备的风向集合

接口说明：

返回空调具备的风向集合

**public** String getACIRPattern()

接口返回：

波形码的字符串

接口说明：

将空调当前的状态生成对应的波形码，逗号分隔的字符串

**public** **int**[] getACIRPatternIntArray()

接口返回：

波形码的int数组

接口说明：

将空调当前的状态生成对应的波形码，int数组

**public** **void** onResume()

接口说明：

初始化空调的解码器，当空调面板进入后台再次打开时，资源可以被释放，调用此接口重新初始化解码器。

**public** **void** onPause()

接口说明：

空调退出时，调用此接口释放资源

其他接口暂不用关心，目前不会使用到，具体的实现请参考酷控提供的Demo示例。