JoyLink2.0.11
京东智能协议组

修订记录：

版本号	修订人	修订日期	修订描述
V 2.0.0		2017.4.19	整理协议文档
V 2.0.5		2018.01.10	增加 IP 代理网关
V 2.0.6		2018.03.04	修改代理认证报文和回应
V 2.0.10		2018.04.25	增加判断设备是否处在待激活状态
V 2.0.11		2018.04.26	将安全章节单独提出并整理

目录

1. 关键词及约定

1.1. 关键词

关键词	描述
ECDH	算法标准：secp160r1
AES	算法标准：AES128 CBC/PKCS#5 padding
feedid	从云端申请并写入设备的唯一 ID 值，不超过 32 字节，设备唯一标识
productuuid	产品类的标识码，6 字节固定长数字和字母组合。是云端系统生成的产品标识码。
accesskey	从云端获取的 key，与 feedid 有一一对应，在设备向云端认证时使用
localkey	APP 根据 accesskey 生成，用于局域网控制。
sessionkey	设备与云端生成的对话密钥，用于广域网与设备端的通讯。

1.2. 约定

crc16 校验和：本协议采用的数据包校验和算法。其实现如下所示：

```
uint16_t CRC16(const uint8_t * buffer, uint32_t size)
{
    uint16_t crc = 0xFFFF;
    if (NULL != buffer && size > 0){
        while (size--) {
            crc = (crc >> 8) | (crc << 8);
            crc ^= *buffer++;
            crc ^= ((unsigned char) crc) >> 4;
            crc ^= crc << 12;
            crc ^= (crc & 0xFF) << 5;
        }
    }
    return crc;
}
```

2. 安全

2.1. 概述

系统安全包括云，App，第三方厂商，设备等四个角色。包括云端与设备互认，授权，报文加密等。
具体方案参见《Joylink 安全方案》。
推荐厂商使用方案中的硬件芯片加密方案。

2.2. 加密方式

方式 0：无加密方式
明文发送数据，可选在 OPT 区间携带发送方的 PublicKey。

方式 1：静态 AES 密钥方式
利用全局共享，预置在程序中的静态密钥加密 Payload 区数据：
AES 算法标准：AES128 CBC/PKCS#5 padding
此加密方式只是形式上看不出明文，实际上并不安全，仅用于传输安全性要求不高的数据。
KEY：” \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 “
IV：”0123456789abcdef“

方式 2：ECDH 协商密钥方式
利用 ECDH 协商临时密钥，实现方式如下：
UDP 返回在线列表由原来的 feedid、productuuid、mac 地址加入设备的 pubKey
机密数据交互时：使用 ECDH 协商的 Sharedkey 进行 AES 加密。
ECDH 算法标准：secp160r1，AES 算法标准：AES128 CBC/PKCS#5 padding
Key = Sharedkey 的前 16 字节（0-15 字节）
IV= Sharedkey 的后 16 字节（16-31 字节）

opt 区段（见章节：升级封装格式）填充数据发送方的 pubkey

加密计算流程：
AES(ECDH(srcPubkey+selfPriKey),控制报文)

回应:

AES(ECDH(srcPubkey+selfPriKey),响应报文)

方式 3：动态 AES 密钥方式
利用预共享的 AES Key 进行加密，实现方式如下：
双方通讯时，使用 AES128 CBC/PKCS#5 padding 对报文进行加密。
AES Key = Key 的前 16 字节（0-15 字节）
IV= Key 的后 16 字节（16-31 字节）
本地控制带上时间戳，与上次相同的直接忽略，如果设备有 RTC 和时间同步，可以设计 1min 作为合法的时间范围（可选，不强制要求）。

加密计算流程：
AES(Key,控制报文)
AES Key=Key 的前 16 字节
Iv=Key 的后 16 字节

回应:
AES(Accesskey,响应 JSON)

3. 角色说明

Joylink2.0 协议组成的网络，按照角色可以把智能硬件分为以下三种：

普通设备：即一般的可连网设备，这样的智能硬件可以通过基站、路由器直接连入因特网，本身具有 IP 地址。

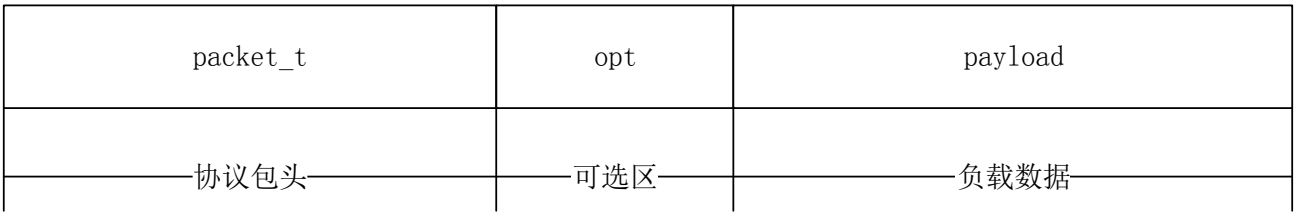
网关设备：这类设备不仅自身有 IP 地址，可以连入因特网；同时代理如 ZigBee、BlueTooth、433 等不具有独立 IP 地址的设备接入网络。

子设备：即不具有独立 IP，不能直接连入因特网，需要依赖网关设备与其它设备或组件通讯。

在以上几种设备之外，还有以下两种系统角色：
控制终端（APP）：与用户产生交互的控制端，指令的发起方，同时也是信息的查询窗口。
云端：提供后台服务、提供广域网连接的具有公网 IP 的服务器端。
本协议是描述以上智能硬件设备与主控、云端如何组成系统，之间如何通讯，如何管理的应用层协议。

4. 数据格式

4.1. 数据包格式



4.2. 包头

```
typedef struct {
    unsigned int        magic;
    unsigned short      optlen;
    unsigned short      payloadlen;
    unsigned char       version;
    unsigned char       type;
    unsigned char       total;
    unsigned char       index;
    unsigned char       enctype;
   unsigned char        reserved;
    unsigned short      crc;
}packet_t;
```

类型值	描述
magic	局域网为 0X123455BB，广域网为 0X123455CC
optlen	opt 区间长度
payloadlen	数据区长度
version	版本号从 1 开始
type	数据包类型
total	需要分包时提供总包数
index	需要分包时提供包序号
enctype	数据包加密类型
reserved	保留
crc	头部之外报文的 crc16 校验

4.3. 可选区

opt 为 option 的简写，是一个可选区。
opt 部分根据不同的指令，其包括的数据意义不同。在大多数控制设备指令时该区也可以为空。在设备发现及设备授权时该区的内容为发送方的 public key。

4.4. 负载区

在本协议中 payload 区的数据格式有明确的定义。各字段的意义也有明确的定义，但根据不同的指令
包字段有所不同。

4.5. 包类型

类型说明表

类型值	**描述**
1	设备发现
2	设备授权
3	数据交互（JSON）
4	数据交互（脚本）
7	OTA 升级
8	OTA 升级状态主动上报
9	云端设备认证
10	心跳
11	云端指令
12	数据上报
13	定时任务
14	结果上报
15	设备主动拉取信息
16	云菜谱上报
17	设备 model_code 上报
102	子设备授权
103	局域网子设备数据交互（JSON）
104	局域网子设备交互（脚本）
105	增加子设备
110	子设备心跳
111	子设备云端指令
112	子设备数据上报
113	子设备解绑
200	云端下发视频推流地址请求
201	设备端对视频推流地址响应

300	增加代理子设备
301	删除代理子设备
311	代理认证
312	代理心跳
313	代理快照
320	控制代理设备

4.6. 加密方式

加密方式	描述
0	不加密
1	静态 AES 密钥
2	ECDH 协商密钥
3	动态 AES 密钥

4.7. biz_code 码

在局域网或者广域网对设备控制时，如果采用脚本方式（即所有的指令已在云端或 APP 采用 lua 脚本端转换完成）虽然采用了 packet_t 结构中的 type 字段标明包的类型，但仍需要在 payload 包中的 biz_code 字段来区分不同的控制指令，其说明如下表：

biz_code 值表

biz_code 值	业务类型
1002	控制请求
102	控制响应
1004	获取设备快照请求
104	获取设备快照响应
1050	任务（云菜谱）下发
150	任务（云菜谱）下发响应
1060	子设备解绑请求
160	子设备解绑响应
1090	查询设备是否支持定时任务请求
190	查询设备是否支持定时任务响应
1091	增加定时任务请求
191	增加定时任务响应
1902	修改定时任务请求
192	修改定时任务响应
1903	删除定时任务请求
193	删除定时任务响应
1904	查询定时任务请求
194	查询定时任务响应
1905	停止定时任务请求
195	停止定时任务响应
1906	重启定时任务请求

196	重启定时任务响应
1907	上报定时任务执行结果请求
197	上报定时任务执行结果响应
1908	上报新增定时任务请求
198	上报新增定时任务响应
1909	上报删除定时任务请求
199	上报删除定时任务响应

5. 局域网交互

5.1. 设备发现 type=1

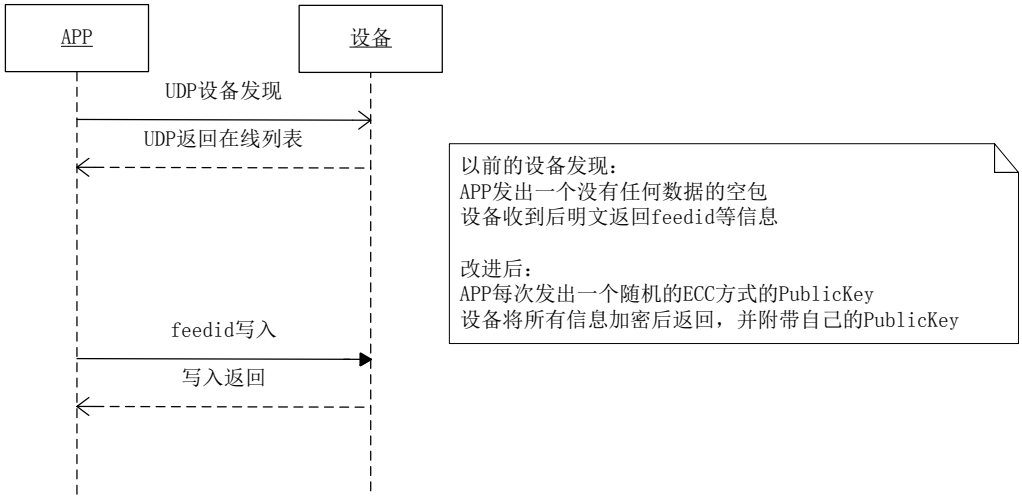


图 5-1 设备发现流程

图 4.1 为设备发现过程。通讯端口为设备监听 80，或者监听备份端口 4320。当初的设计基于局域网可信，没有使用任何加密手段。现在引入一个随机生成 ECC KeyPair 的过程，APP 和设备都必须生成自己的 KeyPair，对外提供自己的 PublicKey。

报文格式具体如下：

请求（APP-->设备）|（OPT=PublicKey，Payload=无加密）：

payload:

```
{
    “cmd”:2
    “productuuid”:”XXXXXX”,
    “status”:1
}
```

响应（设备-->APP）|（OPT=NULL，Payload=无加密）：

```
{
    “mac”:”XX-XX-XX-XX-XX-XX”,
    “productuuid”:”XXXXXX”,
    “CID”: “011c022b”,
    “firmwareVersion”: “aae7_001”,
```

```
“modelCode”: “a7485566bbeeccd8a7485566b9ee9cd8a74855665be2ccd8”,
“lancon”:1,
“trantype”: 1,
“feedid”:”X…X”,
“devkey”:”XXX…XXX’”,
“dbg”:{ “ver”:1,”rssi”:1},
“opt”:{ “type”:1,”std”:1},
"devtype"：0
"subdev":[
      {
          "mac":"XXX…XXX",
          "productuuid":"XXXXXX",
          “CID”: “011c022b”,
          “firmwareVersion”: “aae7_001”,
          “modelCode”: “a7485566bbeeccd8a7485566b9ee9cd8a74855665be2ccd8”,
          “lancon”:1,
          “trantype”: 1,
          "feedid":"X…X",
          “devkey”:”XXX…XXX’”,
          "dbg":{"ver":1,"rssi":1}，
          "devtype"：2
          "state":1,
          "protocol":1,
      }
 ],
"d_idt":{
        "t":1,
        "d_p":"xx",
        "d_s":"xxx",
        "d_r":"xxxx",
        "d_as":"xxx",
        "f_s":"xxxx",
        "f_p":"xxxx"
  }
}
```

注意：只有设备进入待激活状态时在设备发现响应报文中携带 "d_idt":{}信息。

失败时不响应任何信息

设备发现响应个字段描述

字段	描述
mac	mac 地址,也可传输其他形式的厂家唯一标志
productuuid	开发者中心分配的设备品类编码
CID	16 进制数，4 字节，转成 ascii 码为 8byte 字符，设备品类编码
firmwareVersion	固件编号，数字、字线与_的组合，8 个字符长度
modelCode	型号码，最长不超过 64 个字符
lancon	0:局域网不可控, 1:局域网可控
trantype	0:不翻译, 1:Lua, 2:Js
feedid	不超过 32 字节字符串，新设备无 feedid 则返回空串”“，有则返回实际值
devkey	设备 publickey, 42 字节 Hexstring，设备首次上电时随机创建
dbg	调试信息：京东固件版本号，信号强度 0-100（100 最强）
opt	服务器写入的扩展项目,设备存储起来
devtype	0：普通设备,1：网关类设备，2：子设备
state	0：扫描所有设备，1:扫描等待接入设备，2：扫描已接入设备
protocol	0：Wifi , 1:Zigbee, 2: Bluetooth , 3:433

简写	全称	类型
idt	identification	string
a_r	app random	string
d_idt	device identification	string
t	type	int
d_p	device public key	string
d_s	device signature	string
d_r	device random	string
d_as	app random signature	string
f_s	factory signature	string
f_p	factory public key	string
c_idt	cloud identification	string
cloud_sig	cloud signature	string

5.2. 设备授权 type=2

授权之前用户要对设备进行物理操作，类似通过按键或者其他方式操作设备以确定使设备进入到待激活状态。只有设备进入到待激活状态，设备才响应此指令。

请求（APP-->设备）|（OPT=PublicKey，Payload=ECDH 协商密钥加密）：

[时间戳]　//4 字节 int 类型时间戳，小端

{

　“data”:{

```
        “feedid”: “1234567890123456”,
        “accesskey”: “25e35c9691ee91296facd5d28a4822bc”,
        “localkey“: “25e35c9691ee91296facd5d28a4822bc”,
        “productuuid”: “XXXXXX”,
        “server”: [“live.smart.jd.com:2001”],
        “joylink_server” : [“live.smart.jd.com:6001”],
         “tcpaes”: [ “live.smart.jd.com:2014”],
        “lancon”:1,
        “opt”:{ “type”:1,”std”:1},
        "c_idt":{
                "cloud_sig":"xxxxxxxxxxx"
        }
    }
}
```

响应（设备-->APP）|（OPT=NULL，Payload=无加密）：

```
{
    “code”:0,
    “msg”:”success”
}
```

设备授权请求报文字段描述

字段	描述
feedid	云端生成的设备唯一表示，需要持久存储
accesskey	云端分配的设备连接云后认证需要的秘钥
localkey	APP 根据 accesskey 生成的，用于 APP 局域网控制设备
productuuid	开发者后台分配的产品品类
server	ssl 连接的服务器地址及端口
joylink_server	tcp 通讯地址及端口号，joylinkV1 版本之上协议云端通讯地址
tcpaes	tcp 连接地址及端口号，AES 加密
lancon	0:局域网不可控, 1:局域网可控
opt	服务器写入的扩展项目,由服务器确定
cloud_sig	云端生成的激活用的签名信息

5.3. JSON 方式控制 type=3

请求（APP-->设备）|（OPT=NULL，Payload=LocalKey 密钥加密）：

[时间戳]　//4 字节 int 类型时间戳，小端在前

payload:

```
{
```

```
    “cmd”:?,      (数字:根据实际情况选择,见下文)
    “data”:{}     (APP 传的控制指令)
}

data 有两种:
1:获取设备快照
{
    “cmd”:4,
    “data”:{}     (获取快照时传入空对象)
}
2.设备控制
{
    “cmd”:5,
    “data”:{
        “streams”: [
            { “stream_id”: “switch”, “current_value”: “1” }
        ],
        "snapshot": [      //控制时下发全量快照
            {"stream_id":"switch", "current_value":"0"}
        ]
    }
}

响应（设备-->APP）|（OPT=NULL，Payload=Local Key 密钥加密）：
[时间戳]  //4 字节 int 类型时间戳，小端
{
  “code”:0,
    “streams”:[
            {
                “stream_id”: “switch”,
                “current_value”: 1
            },
            {
                “stream_id”: “light”,
                “current_value”: “on”
            }
    ]
```

}

失败时响应：

```
{
    “code”:1,
    “msg”:”err reason”
}
```

5.4. 脚本方式控制 type=4

请求：（APP-->设备）|（OPT=不加密，Payload=LocalKey 密钥加密）：

payload:

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;       //  控制 seq
    unsigned uint8    cmd[];
}control;
```

注意：cmd[]； 脚本解析后的数据，数据长度可计算出来。

模块将收到的 cmd 数据直接透传到控制板

响应：（设备--> APP）|（OPT=不加密，Payload=LocalKey 密钥加密）

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;             //  同控制时的 serial
    unsigned char     resp[];
}control_resp;
```

注意：resp 内容是设备控制响应结果+快照

等待控制板响应超时时间：(500ms)。

5.5. 子设备添加 type=105

当采用扫描子设备信息（二维码）方式时，应当由 APP 主动发送增加子设备报文，此报文中包含需要增加的子设备信息。发送这个报文时应当保证网关设备已同在局域网中并可以正常工作。

请求（APP-->子设备）|（OPT=NULL，Payload=采用网关的 localkey 加密）：

[时间戳]　//4 字节 int 类型时间戳，小端
{
　　“cmd”:6
　　“data”:{
　　　　“mac”: “XXX…XXX”，
　　　　“productuuid”: "XXXXXX"，
　　　　“protocol”: 1
　　}
}
响应（设备-->APP）|（OPT=NULL，Payload=采用网关的 localkey 加密）：
[时间戳]　//4 字节 int 类型时间戳，小端
{
　　“code”:0,
　　“msg”:”success”，
　　“productuuid”: "XXXXXX"，
　　“mac”: “XXX…XXX”
}

5.6. 子设备授权 type=102

授权之前用户要对网关设备进行物理操作，类似通过按键或者其他方式操作设备以确定使网关设备进入到接受激活状态。只有网关设备进入到待激活状态，设备才响应此指令。

子设备绑定必须通过云端完成。发送该报文，即同意子设备入网，同时授权。

请求（APP-->子设备）|（OPT=PublicKey，Payload=ECDH 协商密钥加密,子设备授权用的是网关的 PublicKey，协商密钥也是 APP 与网关协商的）：
[时间戳]　//4 字节 int 类型时间戳，小端
{
　　“data”:{
　　　　“feedid”: “1234567890123456”,
　　　　“accesskey”: “25e35c9691ee91296facd5d28a4822bc”,
　　　　“localkey”: “25e35c9691ee91296facd5d28a4822bc”,
　　　　“server”: [“live.smart.jd.com:2001”],
　　　　“joylink_server” : [“live.smart.jd.com:6001”],
　　　　“tcpaes”: [“live.smart.jd.com:2014”],
　　　　“lancon”:1,
　　　　“opt”:{ “type”:1,”std”:1},

“productuuid”:”XXXXXX”,
“mac”: “XXX…XXX”
}
}
响应（设备-->APP）|（OPT=NULL，Payload=无加密）：
{
“code”:0,
“msg”: ”success”,
“productuuid”:”XXXXXX”,
“mac”: “XXX…XXX”
}

5.7. 子设备控制(JSON 方式)type=103

请求（APP-->设备）|（OPT=NULL，Payload=LocalKey 密钥加密）：
payload:
[时间戳] //4 字节 int 类型时间戳，小端
{
“cmd”:?, (数字:根据实际情况选择,见下文)
“feedid”:”XXXXXXX”,
“data”:{} (APP 传的控制指令)
}
这里的 data 有两种:
1:获取设备快照
{
“cmd”:4,
“feedid”:”XXXXXXX”,
“data”:{} (获取快照时传入空对象)
}
2.设备控制
{
“cmd”:5,
“feedid”:”XXXXXXX”,
“data”:{
“streams”: [
{ “stream_id”: “switch”, “current_value”: “1” }
],
"snapshot": [//控制时下发全量快照

```
                {"current_value":"0","stream_id":"switch"}
        ]
    }
}
```

响应（设备-->APP）|（OPT=NULL，Payload=LocalKey 密钥加密）：

```
[时间戳]   //4 字节 int 类型时间戳，小端
{
    “code”:0,
    “feedid”:”XXXXXXX”,
    “data”:[
          {
              “stream_id”: “switch”,
              “current_value”: 1
          },
          {
              “stream_id”: “light”,
              “current_value”: “on”
          }
    ]
}
```

失败时响应：

```
[时间戳]   //4 字节 int 类型时间戳，小端
{
    “code”:1,
    “msg”:”err reason”,
    “feedid”:”XXXXXXX”,
}
```

5.8. 子设备控制(脚本方式） type=104

请求：（APP-->子设备）|（OPT=不加密, Payload=采用网关的 LocalKey 密钥加密）.

payload:

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;            //  控制 seq
```

```
    unsigned char     feedid[32],       // 子设备 feedid
    unsigned char     cmd[];
}subdev_control;
```

注意：cmd 在网关 localkey 加密之前先用子设备的 localkey 加密

响应：（设备--> APP）|（OPT=不加密，Payload=采用网关的 LocalKey 密钥加密）

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;           // 同控制时的 serial
    unsigned char     feedid[32],       // 子设备 feedid
    unsigned char     resp[];
 } subdev_control_resp;
```

注意：resp 在网关 localkey 加密之前先用子设备的 localkey 加密

等待子设备响应超时时间：(500ms)。

6. 广域网

APP 从 joylink SDK 中获取对应的 feedid 是否在线，如果在线的话就将控制请求提交给 SDK，SDK 直接通过 UDP 协议透传给设备，设备执行后原路返回通知 APP，同时异步告知云端。

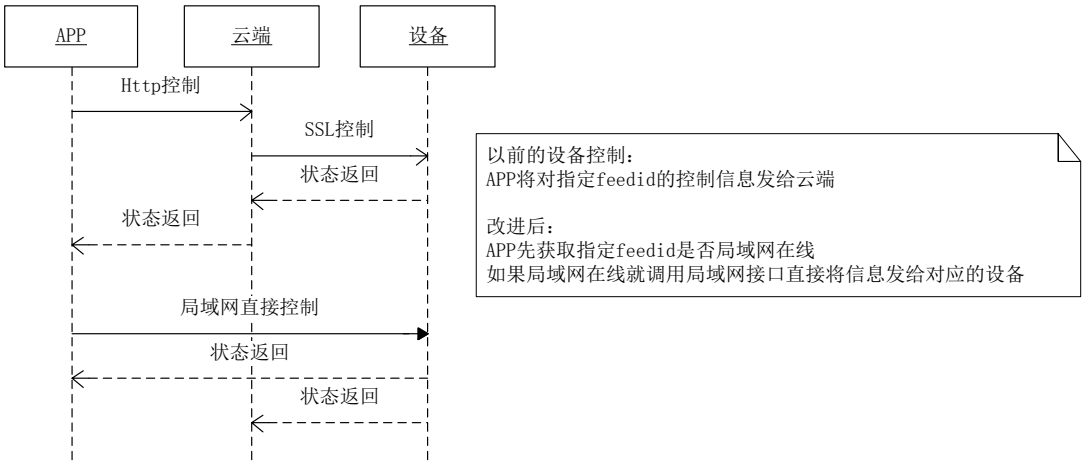


图 6-1 数据通信流程

设备上电后，向云端发送认证请求，请求数据包的头部(package_t)和 OPT 部不加密，数据体(payload)使用 access key 进行 AES 加密。

云端在验证设备合法性后，会随机生成一个 session key 并下发给设备。如果设备认证不合法即断开和设备的连接。

设备认证成功并和云端建立长连接，之后心跳、控制、数据上报等数据包体改为使用 session key 进行 AES 加解密。

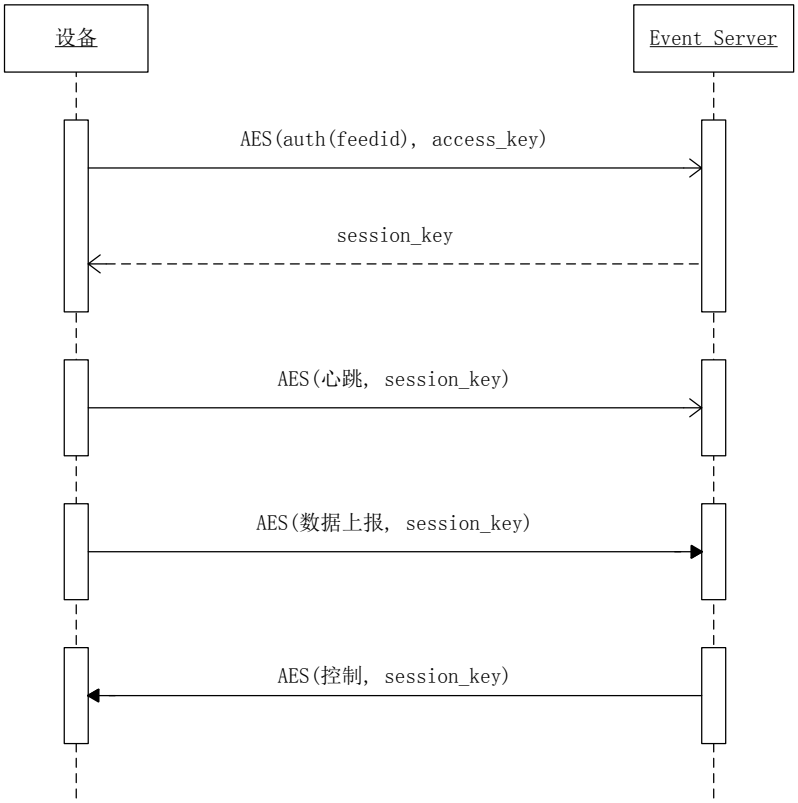


图 6-2 设备和云端交互流程

6.1. 设备认证请求 type=9

数据可选区（OPT）

```
typedef struct {
    unsigned char     feed_id[];        // 不定长字符串  是否需要改成 32 定长
    unsigned int      random_num;    // 设备生产的随机数字
}opt;
```

云端认证响应 type=9

```
typedef struct {
    unsigned int      random_num;    // 云端生产的随机数字
} opt;
```

数据包体（payload）

设备认证请求 type=9

```
typedef struct {
```

```
    unsigned int    timestamp;              // 1970 年 1 月 1 日  00:00:00 至今的秒数
    unsigned int    random_num;             //  设备生产的随机数字
}auth;
```

云端认证响应 type=9

```
typedef struct {
    unsigned int      timestamp;            //   UTC 时间
    unsigned int      random_num;           //  云端生产的随机数字
    unsigned char     session_key[32];      // key + iv，二进制非 ASCII
}auth_resp;
```

注：设备认证请求的 OPT 部和 payload 部的随机数需要一致。云端通过验证随机数是否一致来验证数据包的合法性。设备和云端连接断开后或连续 3 次未收到服务器回复，需要重新走认证流程。

注：云端的 OPT 部和 payload 部的随机数是一致的。设备应该通过验证随机数是否一致来验证数据包的合法性。

子设备不向云端进行认证，授权完成后，网关连入云端，网关认证完成后，子设备不再认证。对子设备的远程管理加密密钥采用网关协商的 session_key 密钥。

时间戳是小端，linux 平台下系统自动转换为小端序，厂商需根据开发环境来判断是否要转化。
另外厂商需要根据认证响应中的时间戳来更新设备时间，务必每次协议包中的时间戳都是正确的。

6.2. 设备心跳请求 type=10

```
typedef struct {
    unsigned int      timestamp;            //设备时间
    unsigned short    version;              //  固件版本序号，需和后台录入一致
    unsigned short    rssi;                 // 0-100(%)取值
}heartbeat;
```

注：设备心跳间隔最长 60 秒

云端心跳响应 type=10

```
typedef struct {
    unsigned int      timestamp;        //  服务器当前时间
```

```
    unsigned int      code;              // 错误码
}heartbeat_resp;
```

6.3. 设备数据上报请求 type=12

```
typedef struct {
    unsigned int      timestamp;
    unsigned char     data[];            //需要脚本解析的数据,变长
}dataupload;
```

注：模块认证成功后需立刻向云端上报一次快照。未上报快照的设备可能无法控制。

云端数据上报响应 type=12

```
typedef struct {
    unsigned int timestamp;
    unsigned int code;                   // 错误码
}heartbeat_resp;
```

6.4. 云端控制请求 type=11

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;       // 控制 seq
    unsigned char     cmd[];        //脚本解析后的数据
}control_t;
```

设备控制响应 type=11

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
    unsigned char     resp[];       //设备控制响应结果+快照
}control_resp_t;
```

6.5. 云端获取设备快照 type=11

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
    unsigned char     cmd[];        //脚本解析后的数据，命令为空
}getSnapshot;
```

获取设备快照响应 type=11

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    unsigned char       resp[];
}getSnapshot_resp;
```

unsigned char resp[]的内容：

成功时响应：

```
{
    “code”:0,
     “streams”:[
            {
                “stream_id”: “switch”,
                “current_value”: 1
            },
           {
                “stream_id”: “light”,
                “current_value”: “on”
            }
     ]
}
```

失败时响应：

```
{
    “code”:1, （0 成功，其他失败|失败信息会透传到云端）
    “msg”:”err reason”
}
```

6.6. 云菜谱下发 type=11

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    unsigned char       cmd[];              //脚本解析后的数据;
}sendTask;
```

云菜谱下发响应 type=11

```
typedef struct {
    unsigned int        timestamp;
```

```
    unsigned int       biz_code;
    unsigned int       serial;
    unsigned char      resp[];
}sendTask_resp;
```

6.7. 云菜谱上报 type=16

```
typedef struct {
    unsigned int       timestamp;
    unsigned int       serial;
    unsigned char      report[];          //脚本解析后的数据;
}reportTask;

report 内容：
{
  "task_id":"123",                        //菜谱 id
  "subTask":[
    {
      "id":"23",                          //步骤 id
      "index":"2",                        //顺序 id
      "result":"0",                       //执行结果，0 成功，1 失败 , -1 未执行 ,
                                          //2 投放果料，3 执行剩余 5 分钟
      "startTime":"2015-05-07 16:03:53", //步骤开始时间
      "endTime":"2015-05-07 16:03:53" //步骤结束时间
    },
    {
      "id":"23",                          //步骤 id
      "index":"2",                        //顺序 id
      "result":"0",                       //执行结果，0 成功，1 失败 , -1 未执行
      "startTime":"2015-05-07 16:03:53",      //步骤开始时间
      "endTime":"2015-05-07 16:03:53"         //步骤结束时间
    }
  ]
}

云菜谱上报响应 type=16
typedef struct {
```

```
    unsigned int      timestamp;
    unsigned int      serial;
    unsigned int      code;
}reportTask_resp;
```

6.8. 子设备心跳请求 type=110

```
typedef struct {
    unsigned short    version;        // 固件版本
    unsigned short    rssi;           // 0-100(%)取值， -1 代表无意义
    unsigned char     feedid[32];
    unsigned char     state;          //0:设备在线，1：设备离线，
                                      //2：设备解绑，3：未知状态
} subdev_info;

typedef struct {
    unsigned int      timestamp;      //设备时间
    unsigned short    version;        // 固件版本
    unsigned short    rssi;           // 0-100(%)取值
    unsigned short    num;            //在线的子设备数量
    subdev_info       sdevinfo[];     //子设备的信息
} dev_heartbeat;
```

注：设备心跳间隔最长 60 秒

子设备云端心跳响应 type=110

```
typedef struct {
     unsigned int     timestamp;      // 服务器当前时间
     unsigned int     code;           // 错误码
} dev_heartbeat_resp;
```

6.9. 子设备数据上报请求 type=112

```
typedef struct {
    unsigned int      timestamp;
    unsigned char     feedid[32];
    unsigned char     data[];  //需要脚本解析的数据变长,该数据应用子设备的 accesskey
加密
} subdev_dataupload;
```

注：模块认证成功后需立刻向云端上报一次快照。未上报快照的设备可能无法控制。

子设备云端数据上报响应 type=112

```
typedef struct {
     unsigned int       timestamp;
     unsigned char      feedid[32];
     unsigned int       code;
} subdev_heartbeat_resp;
```

6.10. 子设备云端控制请求 type=111

```
typedef struct {
     unsigned int       timestamp;
     unsigned int       biz_code;
     unsigned int       serial;
     unsigned char      feedid[32];
     unsigned char      cmd[];
} subdev_control;
```

注：cmd[]; 脚本解析后的数据 //该数据应用子设备的 accesskey 加密

子设备设备控制响应 type=111

```
typedef struct {
     unsigned int       timestamp;
     unsigned int       biz_code;
     unsigned int       serial;
     unsigned char      feedid[32];
     unsigned char      resp[];
} subdev_control_resp;
```

注意: esp[]; 设备控制响应结果+快照 该数据应用子设备的 accesskey 加密

6.11. 子设备云端获取设备快照 type=111

```
typedef struct {
     unsigned int       timestamp;
     unsigned int       biz_code;
     unsigned int       serial;
     unsigned char      feedid[32];
     unsigned char      cmd[];
} subdev_getSnapshot;
```

注意：cmd[]; //脚本解析后的数据 //该数据应用子设备的 accesskey 加密

子设备获取设备快照响应 type=111

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    unsigned char       feedid[32];
    unsigned char       resp[];
} subdev_getSnapshot_resp;
```

注意：resp[];该数据应用子设备的 accesskey 加密

6.12. 子设备解绑请求 type=113

<云端—>网关>

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    unsigned short      num;                //子设备数量
    subdev_info         info[];             //子设备的信息
}subdev_delete_cmd;
```

子设备解绑响应 type=113

<网关—>云端>

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    unsigned short      num;        //子设备数量
    subdev_info         info[];     //子设备的信息
} subdev_delete_resp;
```

6.13. 云端下发视频播放地址请求 type=200

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;         //  控制 seq
```

```
    unsigned char     cmd[];        // 见下面报文格式
}control;
```

设备对云端下发视频播放地址响应 type=201

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
    unsigned char     resp[];       //见下面报文格式
}control_resp
```

视频设备控制 cmd 请求报文格式	{
	"control": [
	{
	"stream_id": " push_stream",
	"current_value":
	"{\"resolution\":240,\"address\":\"rtsp://jdserver.com\"}"
	}
	],
	"feed_id": 10000,
	"attribute": {
	"serial": 87020
	}
	}

视频设备控制 resp 响应报文格式	{
	"result": 0,
	"control_resp": "control_success",
	"resolution": [
	240,
	360,
	480
	],
	"attribute": {
	"serial": 87020
	},
	"device": {
	"feed_id": "1294902345",
	"access_key": "xxxxxxx"
	}
	}

视频设备控制字段详细说明：

字段	是否必要	说明
feed_id	是	设备在京东智能云的编号，唯一
access_key	是	设备秘钥

stream_id	是	功能标识,例如“推视频”的标识为” push_stream”
current_value	是	功能标识需要达到的状态值
resolution	是	视频分辨率分为:240,360,480 对应流畅，标清，高清
resolution	是	设备返回中的 resolution 表示设备中支持分辨率列表
address	是	京东 rtsp 流媒体服务器推送地址
control	是	本次需要控制的功能标识及其状态值
attribute	是	本次控制属性，设备端不需要解析，只需原封不动添加到响应包中，里面目前只有“serial”一项，以后有可能增加
result	是	设备对本次控制命令的响应结果：0 标识成功，1 认证失败,-2 无认证信息,-3 设备不在线,-4 不支持,-5 其他
control_resp	是	设备受控的文字描述，例如 “控制成功”，或失败时的原因，例如“设备异常”。此字段信息最终会返回给 app，“京东微联”在控制失败时，会将此信息显示给用户。
streams	是	如果设备受控成功后，需将此时的所有 stream_id 的当前值返回

6.14. 设备主动拉取信息 type=15

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      serial
    unsigned int      length;
    char info[];      //json 格式，具体要拉取的信息 streamid 号
}info_fetch_req;
Json 举例：
{
  "type":[
    "weather",
    "air"
  ]
}
云端反馈拉取信息 type=15
typedef struct {
    unsigned int      timestamp;
    unsigned int      serial
    unsigned int      ength;   //json 串的长度
    int               code;    //0:成功，其他值:失败码
    char              info[];  //json 格式，具体要反馈的信息
}info_fetch_rsp;
Json 举例：
```

```
{
   "result":[
      {
         "type":"weather",
         "value":{
            "city_name":"北京",
            "district":"北京",
            "proven":"北京",
            "city_code":"101020100",
            "publish_time":"yyyy-MM-dd'T'HH: mm: ssZ",
          }
      },
      {
         "type":"air",
         "value":{
            "pm25":"6",
            "aqi":"20",
            "publish_time":"yyyy-MM-dd'T'HH: mm: ssZ"
         }
      }
   ]
}
```

6.15. 设备上报 model_code type=17

设备上报 type=17

```
typedef struct {
    unsigned int        timestamp;
    char*                model_codes;
}model_code;
```

model_codes 是 json 格式的字符串，考虑到有网关了设备的上报子设备的需求，并且这个信息只上报一次，所以用 json 格式字符串，以数组的方式上报。

```
{
     "model_codes":[
                    {"feedid":"123456","model_code":"model code val aa"},
                    {"feedid":"654321","model_code":"model code val bb"}
                    ]
}
```

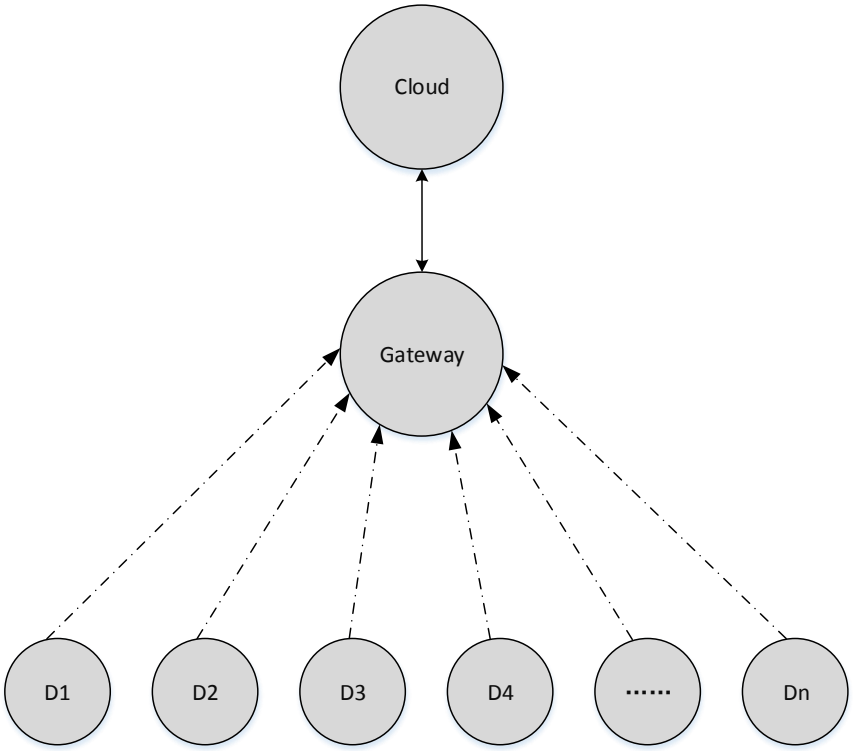
model_code 字符串最长 64Byte。

云端响应 type=17

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        code;                    //错误码
} model_code _resp;
```

注：设备与云端建立长连接，心跳正常后（避免与影响认证时候快照的上报时间），只上报一次。

7. IP 类型设备网关代理



图：代理网关构建的网络方案

在网关代理的方案中，主要的优点是代理设备直接由代理网关代理所有原来与云端的通讯过程（包括心跳、状态上报、设备控制）。本方案希望做到设备端基本不需要开发改动，基于现有的机制及相关协议 IP 代理网关及云端通讯机制可能需要做出相应的改动。

这个方案的主要流程是：

1）代理网关作为一个硬件/软件必须在物联网后台合法注册，并经过微联流程授权与云端建立长连接。

2）需要被代理的设备需要二次合法授权，把之前指定的云端服务器长连接端地址改成本地代理网关的地址（可以是 IP 或者域名，在 APP 端扫描到该类设备可以不出现在设备列表中，当需要代理网关设置时展现给用户）。

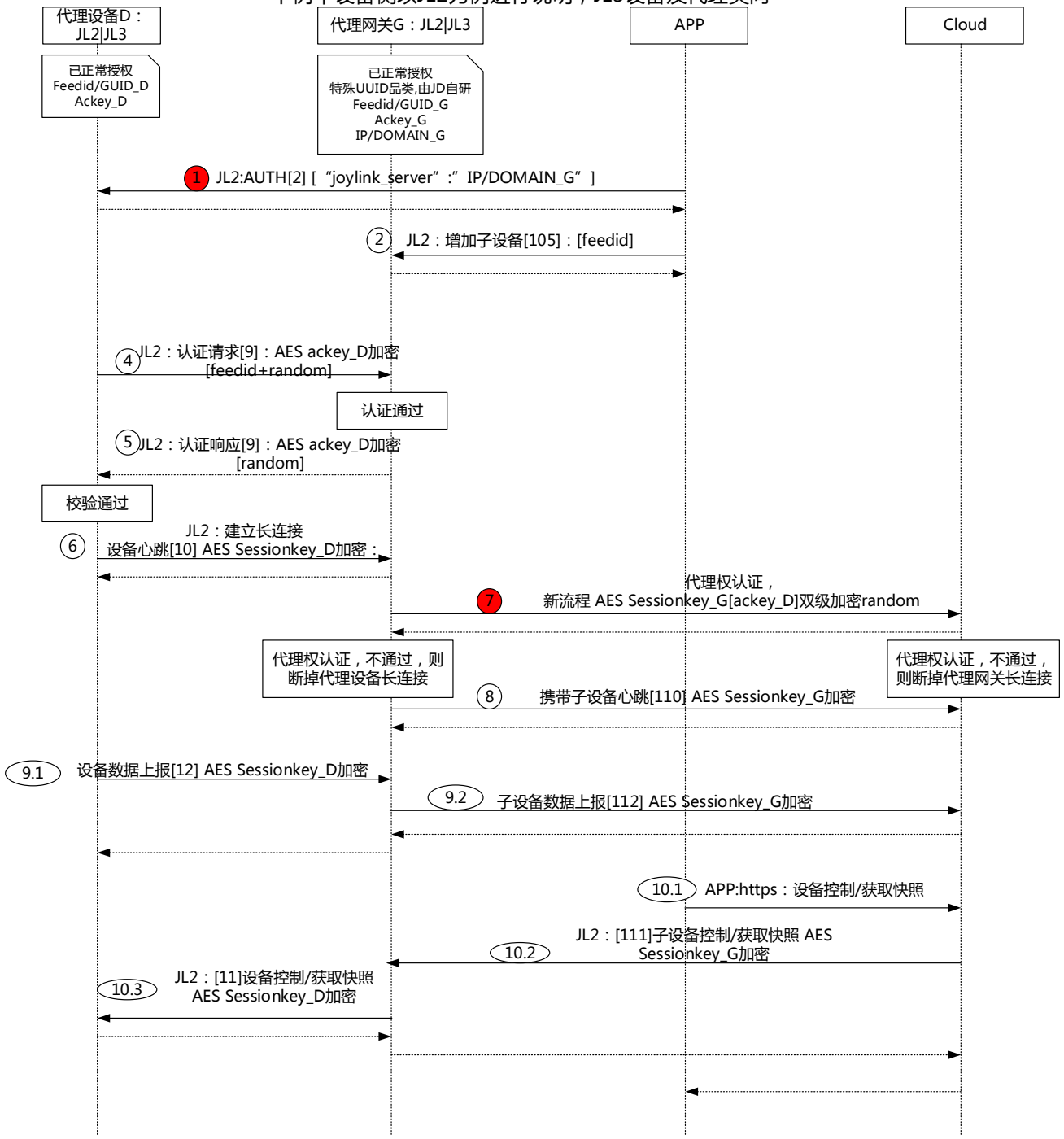
3）设备向网关进行双向认证，通过后维护与之前云端类似的长连接 session（设备侧应不需要做出发改变）。

4）设备长连接心跳、云端控制指令、向云端的快照信息，实际会经由代理网关转发。

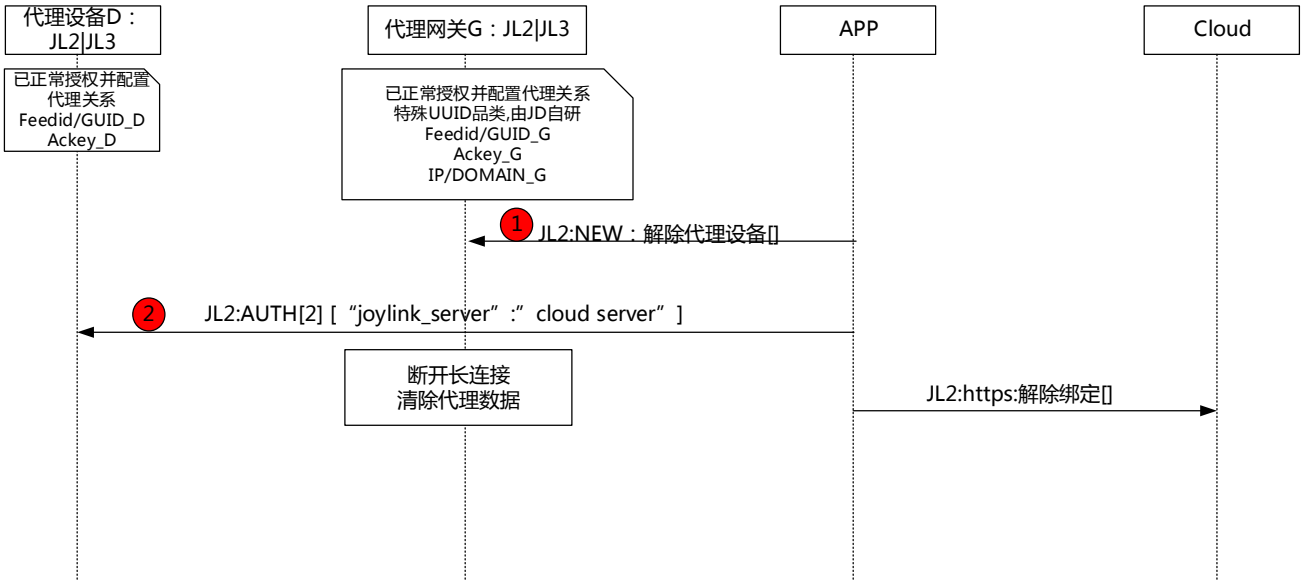
5）设备被设置成普通模式时恢复直连云端。

6）网关模型可以是一个硬体也可以是一个软体。

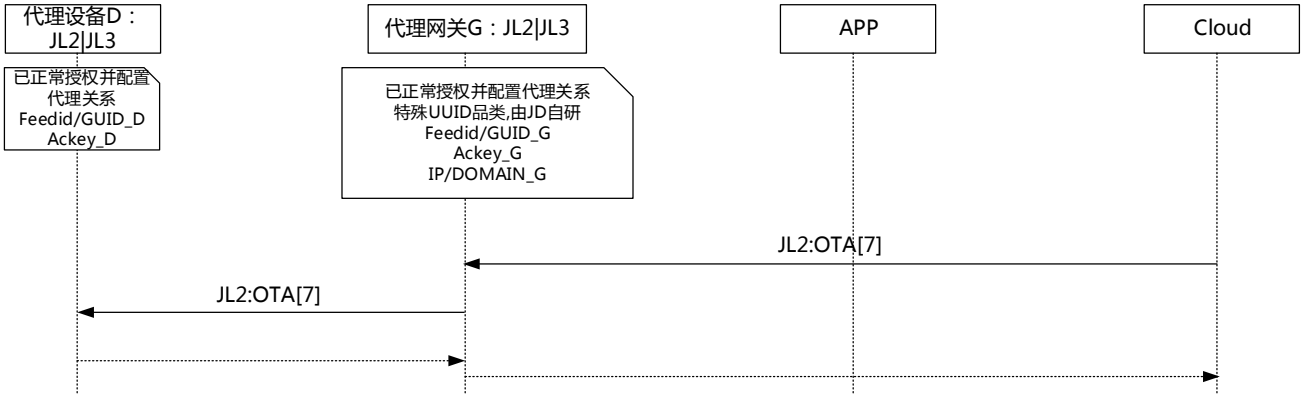
网关向下代理局域网通讯以设备协议为主，代理网关对外通讯采用需要同时支持
JL2和JL3协议对外接口，不同的设备采用不同的协议代理
下例中设备侧以JL2为例进行说明，JL3设备及代理类同



图：代理设备设置代理及主要工作时序图



图：代理设备解除代理关系



图：代理设备的 OTA

代理网关的心跳，如果有代理设备，就放在代理心跳中，如果没有代理设备就走普通设备的心跳逻辑。

代理设备的其他交互协议例如:快照上报，云端控制等全部遵守普通设备的协议。

7.1. 增加代理子设备（type=200，APP->代理网关）

请求（APP-->代理网关）|（OPT=NULL，Payload=LocalKey 密钥加密）：

[时间戳]　//4 字节 int 类型时间戳，小端

[

{"feedid":"12312312312313"，"ackey":"12312312315555"},
{"feedid":"12312312312314"，"ackey":"12312312315545"}
]

响应（设备-->APP）|（OPT=NULL，Payload=无加密）：
{
“code”:0,
“msg”: ”success”
}

7.2. 删除代理子设备（type=201，APP->代理网关）

请求（APP-->代理网关）|（OPT=NULL，Payload=LocalKey 密钥加密）：
[时间戳] //4 字节 int 类型时间戳，小端

[
{"feedid":"12312312312313"},
{"feedid":"12312312312313"}
]

响应（设备-->APP）|（OPT=NULL，Payload=无加密）：
{
“code”:0,
“msg”: ”success”
}

7.3. 代理认证（ type 211，代理网关->云端）

网关向云端查询是否可以代理 feedid 指定的设备

typedef struct {
unsigned int timestamp;
char* agent_devs;
}agent_auth_req;
agent_devs 是 json 格式的字符串，以数组的方式申请。
[
{"feedid":"12312312312313", "random":aaaa , "cloud_auth_hexstr":"aaaaxx"},
{"feedid":"12312312312313", "random":bbbb, "cloud_auth_hexstr":"bbbbxx"}

]

注意: cloud_auth_hexstr type 中的内容是 type = 9， 广域网认证的 palyload 内容的转 16 进制字符串。

```
typedef struct {
    unsigned int  timestamp;
    unsigned int  code;                          //错误码  0  成功，  -1  失败。
    char*      devs;
} agent_auth _resp;
```

如果成功 devs 中是空，如果失败，devs 是 json 格式的字符串，以数组的方式, 内容是出错了的设备的 feedid。

```
[
{"feedid":"12312312312313"},
{"feedid":"12312312312313"}
]
```

注：cloud_auth_hexstr 的值是经过被代理设备 access_key 加密。代理网关代理报文用 session_key 加密。

注：代理网关不验证是否有代理权，只通过每次验证要代理设备完成云端验证即可。

7.4. 代理心跳上报(type 212, 代理网关->云端)

代理网关发送代理设备的心跳给云端。

```
typedef struct {
    unsigned int        timestamp;
    char*                agent_devs;
}agent_hb_req;
```

agent_devs 是 json 格式的字符串，以数组的方式申请。

```
[
{"feedid":"12312312312313", "version":3, "rssi":4},
{"feedid":"12312312312314", "version":5, "rssi":4}
]
```

注意：version， rssi 数据类型与 type=10， 设备心跳请求一致。

```
typedef struct {
```

```
    unsigned int        timestamp;
    unsigned int        code;                    //错误码 0 成功， -1 失败、
} agent_hb_resp;
```

7.5. 代理快照上报(type 213，代理网关->云端)

代理网关发送代理设备的快照给云端。

```
typedef struct {
    unsigned int        timestamp;
    char*               agent_snaps;
}agent_snaps_req;
```

agent_snaps 是 json 格式的字符串，以数组的方式申请。

```
[
{
 "feedid":"12312312312313",
 "snap_hex_str":"xxccc"
},
{
 "feedid":"12312312312314",
 "snap_hex_str":"xxccc"}
]
```

注意: snap_hex_str 内容是设备数据上报请求 type=12 上报 palyload 内容的转 16 进制字符串。

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        code;                    //错误码 0 成功，-1 失败。
} agent_snaps_resp;
```

7.6. 代理设备控制(type 220，云端->代理网关)

报文参考 云端控制请求 type=11 和回应， 代理控制和回应只是在原报文前添加 char[32],的 feedid。

云端控制请求 type=11:

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
```

```
    unsigned int      serial;       // 控制 seq
    unsigned char     cmd[];        //脚本解析后的数据
}control_t;

设备控制响应 type=11
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
    unsigned char     resp[];       //设备控制响应结果+快照
}control_resp_t;

代理控制：
typedef struct {
    char feedid[32];
    control_t cloud_ctr;
}agent_control_t

代理控制回应:
typedef struct {
    char feedid[32];
    control_resp_t ctr_rsp;
}agent_control_resp_t
```

7.7. 代理云端获取设备快照（type 220，云端->代理网关）

报文与代理设备控制(type 220，云端->代理网关) 报文相同，只是其中 biz_code 不同是获取快照的 biz_code。

7.8. OTA

普通设备报文有 feedid，并且此命令使用频率较低，所以代理网关只是做路由，云端与被代理设备透传。

7.9. Mode code

普通设备的 mode code 上报报文有 feedid，并且此命令使用频率较低，代理网关直接将代理设备的 Mode code 报文发送给云端。

8. 子设备管理

什么是子设备？

在 joylink 协议中，设备本身具有 IPV4 或 IPV6 的地址，本身能直接连入互联网，这类设备为普通设备。存在着这样一类设备，它们没有 IPV4/IPV6 地址，接入互联网需要通过网关设备代理，这类设备统称为子设备。这类的设备有 Bluetooth 类、ZigBee 类、433M 类。本章节的描述即是以对此类设备的管理说明。

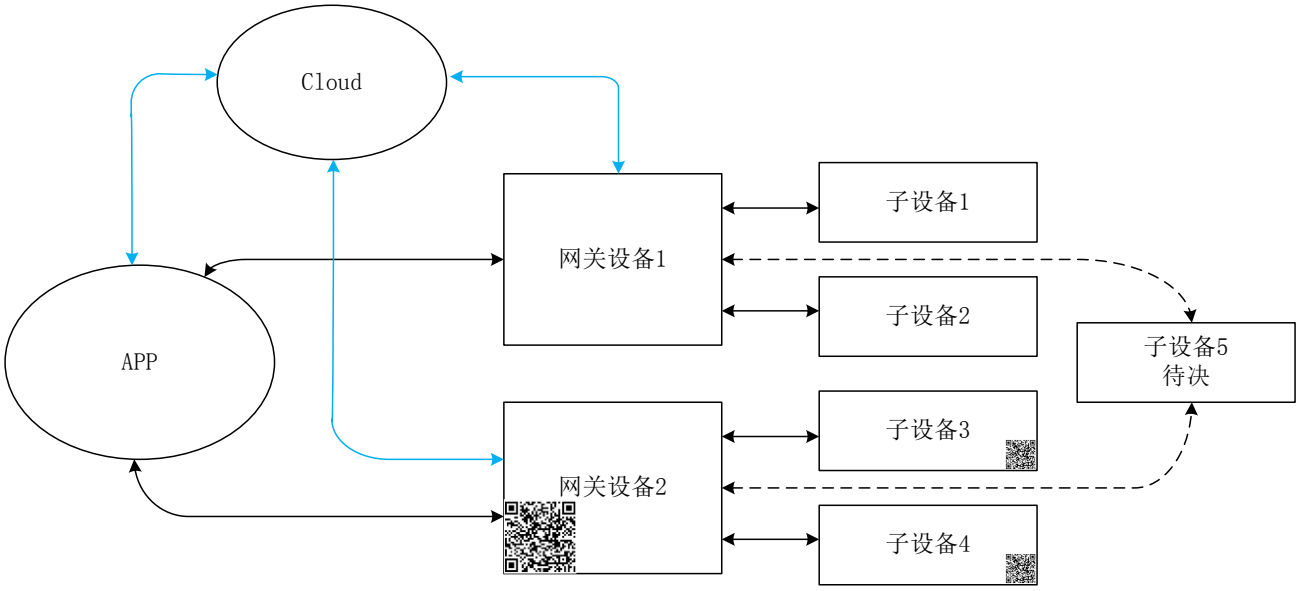


图 8-1 子设备管理系统图

8.1. 增加子设备

子设备入网的前提是：网关类设备必须已经与 APP[控制端]处于一个局域网下，即网关至少已经完成一键配置。在扫码入网方式时，网关设备必须是已经存在于 APP[控制端]的设备列表中。

子设备通常上电后应该向默认的网络/网关发送入网请求，当需要较高的安全性时应该有一个机制触发子设备发送入网请求，这种请求只在一个时间内有效。且在已经绑定到一个网络之后，子设备应具备复位功能，以便重新入网。

单向信道类的子设备仅能采用扫码入网的方式。

8.1.1. 采用设备发现报文方式

子设备只有增加子设备，如果对子设备授权则网关可以子设备接入。对于不允许入网的设备不做处理。

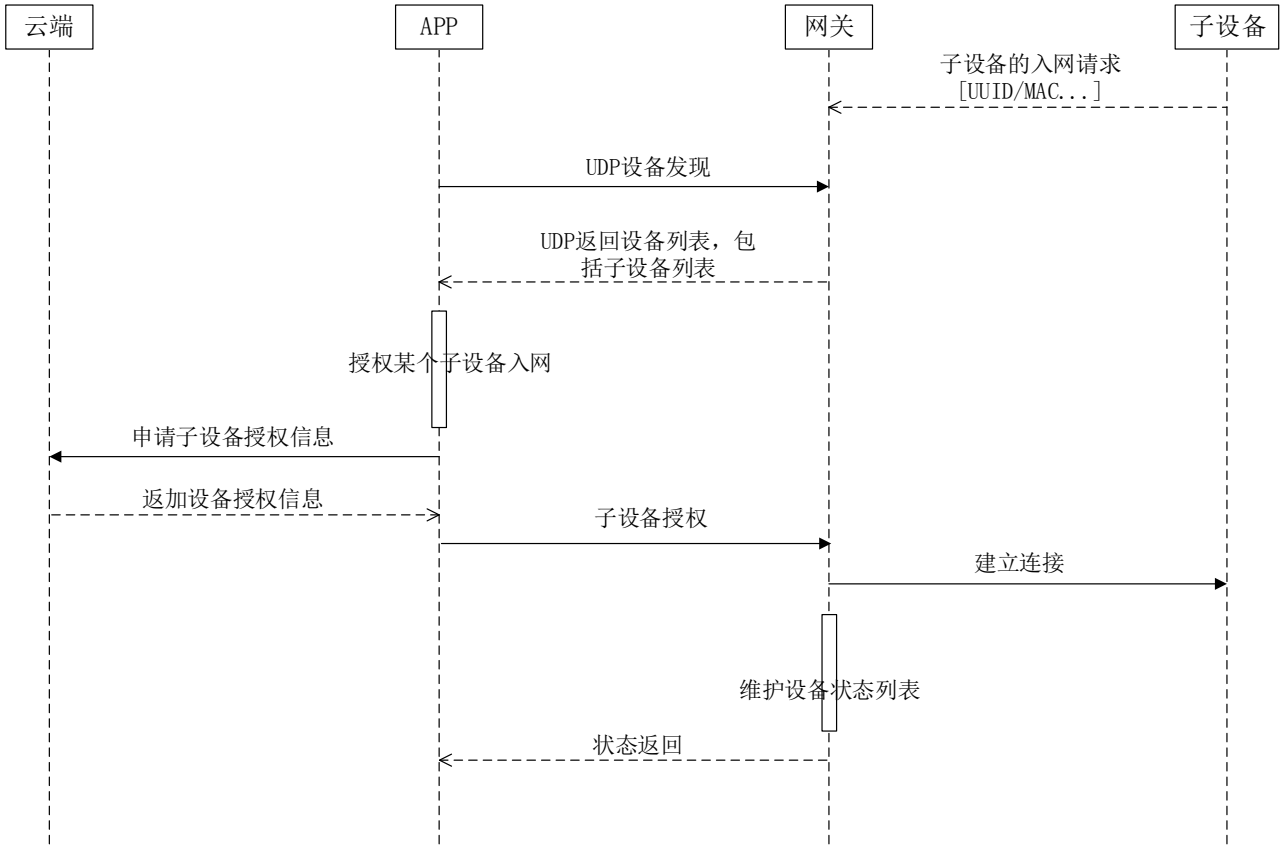


图 8-2 设备发现添加子设备

【子设备绑定状态返回给云端】

（1）当一个没有入网的子设备上电后，或者被触发后应当向默认的网关发送入网请求，网关能够探测到所有请求入网的子设备，并收集子设备的请求信息。

（2）采用该方式，增加子设备需要由 APP 端向网关发送一个设备发现报文，当网关接收到该消息后，应当向 APP 发送设备在线列表，该设备列表可能包括网关自己的设备信息，同时要发送子设备的在线列表信息，这些信息是在网关的设备返回列表中增加的。

（3）当 APP 接收到该信息，如图 5.3 所示，如果用户选择增加一个子设备在一个指定的网关下，需要发送子设备授权报文，该报文包括子设备的硬件信息。当网关收到指令后，与子设备建立连建，并维护子设备状态信息。给 APP 返回状态。

（4）增加子设备时的数据加密：设备发现与响应报文 payload 无加密。子设备授权报文采用加密方式 2，即 ECDH 协商密钥方式。其协商密钥是 APP 与设备之间的协商密钥。

（5）网关只能在子设备授权或增加时接受子设备的连接请求。

8.1.2. 采用扫码入网方式

单向信道类的子设备仅能采用扫码入网的方式。

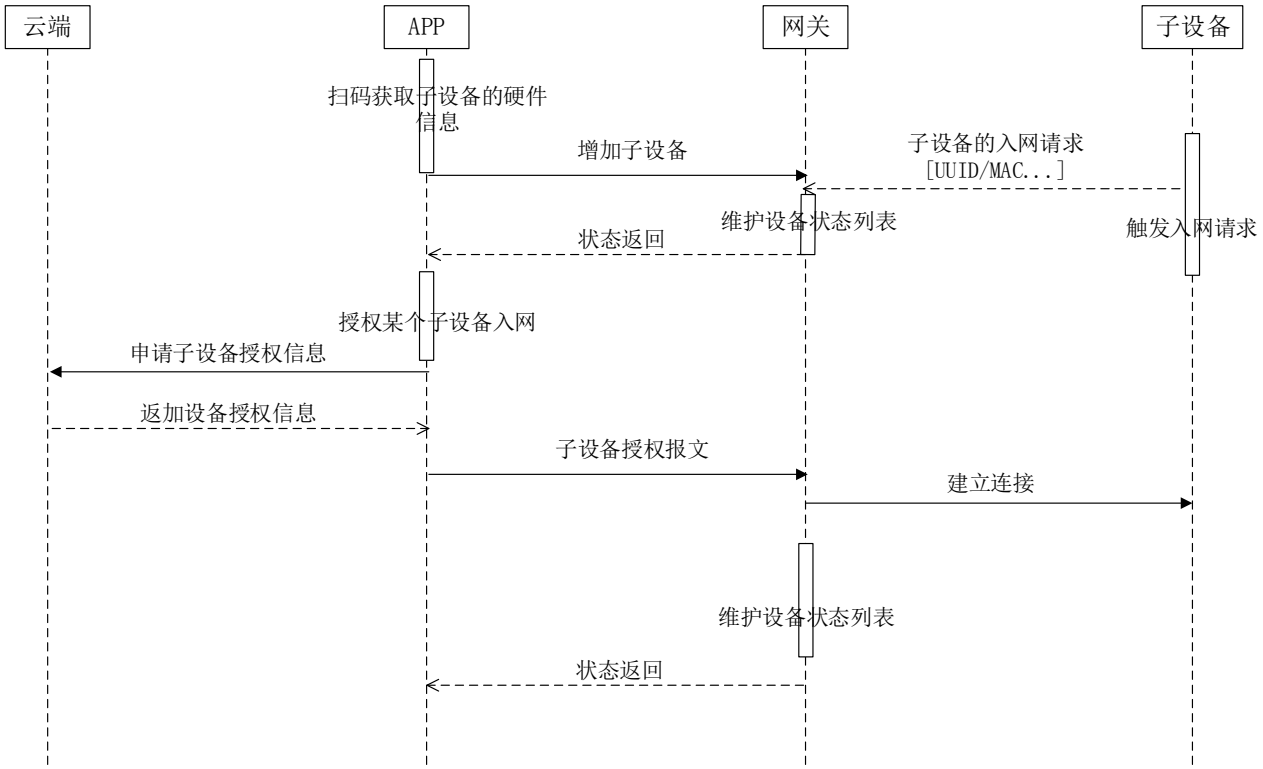


图 8-3 扫码入网流程

采用扫码方式入网，需要在子设备上贴有二维码，此二维码需要包括产品的 MAC 地址，productuuid 等信息。APP 增加该子设备时，扫描该二维码，再发送增加子设备报文，（该方式下可以在下发增加子设备报文后再触发子设备入网请求），在（5~10 秒）内如果增加子设备成功，则应该发送子设备授权报文。收到授权报文网关应更新子设备信息，并返回状态。

8.2. 子设备授权

针对子设备的授权，整体思想是子设备具有独立的 feedid 以及 accesskey，localkey。其中 feedid 是有用的信息，用来在授权之后标志子设备。由于网关与 app 端的通讯采用网

关的 localkey 加密，与云端的通讯采用 session_key 通讯。但在控制过程中子设备的 localkey 是有用的，子设备控制过程中的 cmd[],resp[]的内容要求先用 localkey 加密，然后把 opt 及 payload 部分再用网关的 localkey 或者 session_key 加密。子设备的硬件信息、状态、授权信息由网关维护。APP 在授权子设备时可以携带子设备的信息向云端申请 feedid 以及 accesskey，并采用局域网设备授权报文发送给网关。请注意针对子设备的授权采用的是 APP 与网关之前的 sharekey 加密。

子设备并没有独立的 ECC 的公钥与私钥，因此也没有与 APP 之间的协商密钥。

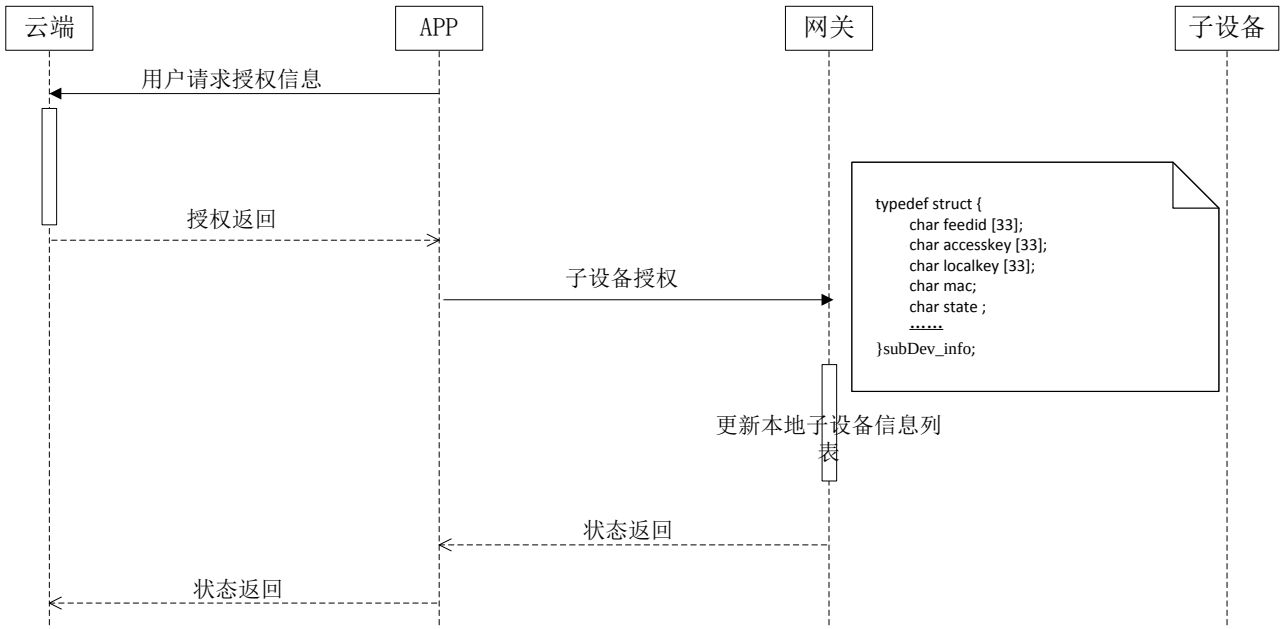


图 8-4 子设备授权

8.3. 子设备局域网控制

在局域网环境中对子设备的控制采用子设备局域网交互报文。

8.4. 子设备广域网控制

网关设备向云端认证，认证通过之后，得到云端给予的 session_key,子设备与云端的通讯共享此 session_key。即网关与云端建立一个会话，子设备在此会话上通过协议与云端通讯。子设备与云端的通讯采用子设备广域网报文.

8.5. 子设备的心跳

子设备心跳包由网关设备代发，一次子设备心跳包中可以包括所有的子设备信息。

8.6. 子设备解绑

为了保证云端、网关、APP 部分的数据保持一致，对子设备的解绑要求网关必须在线。

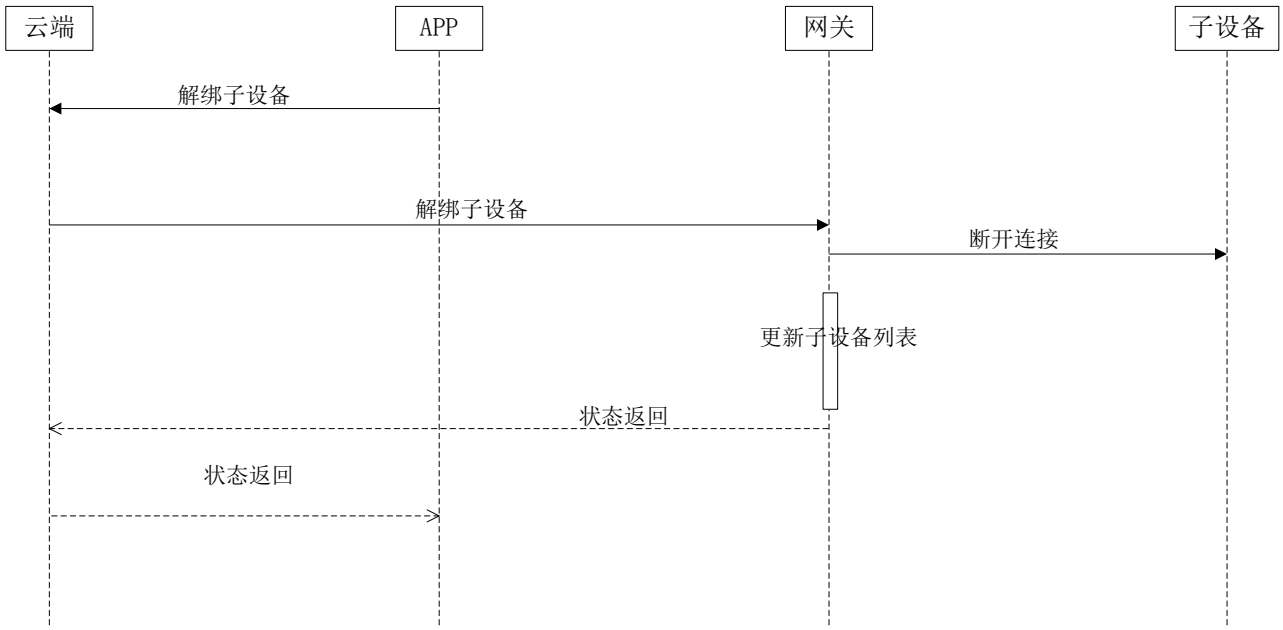


图 8-5 子设备解绑

9. 升级

9.1. 固件提交

开发者可以在云端提交测试过的固件版本。固件对应相应的产品。固件在提交后，产生两个值：
固件的序号。数字值，由云端自动增加，每提交一次序号加 1。
固件的版本名称：
版本名称建议的格式如下：
1.1.1.20151020_release
注意：厂商必须严格按照以上格式填写版本号。
主版本号（最大 3 位数字）：当功能模块有较大的变动，比如增加模块或是整体架构发生变化。
次版本号（最大 3 位数字）：相对于主版本号而言，次版本号的升级对应的只是局部的变动，但该局部的变动造成程序和以前版本不能兼容。
修订版本号（最大 3 位数字）：一般是 Bug 的修复或是一些小的变动或是一些功能的扩充。
日期版本号（年月日的组合如：20151020）：固件的修改日期。
希腊字母版本号：此部分必须是 release,即固件必须是经过严格测试且功能完整的。

产品（模块）端收到升级指令从固件资源端下载固件采用 http 协议。

9.2. 固件验证

云端应生成固件的 CRC32 校验值，下发升级指令时，该应携带发送给设备。（下载完毕和拷贝完毕后都必须对固件进行 CRC32 校验）
Crc32 实现如下所示：
CRC32=X32+X26+X23+X22+X16+X12+X11+X10+X8+X7+X5+X4+X2+X+1

```
uint32_t crc32_table[256];
int make_crc32_table()
{
    uint32_t c;
    int i = 0;
    int bit = 0;

    for(i = 0; i < 256; i++)
    {
```

```
            c    = (uint32_t)i;
            for(bit = 0; bit < 8; bit++)
            {
                  if(c&1)
                        c = (c >> 1)^(0xEDB88320);
                  else
                        c =    c >> 1;
            }
            crc32_table[i] = c;
      }
}

uint32_t make_crc(uint32_t crc, unsigned char *string, uint32_t size)
{
      while(size--)
            crc = (crc >> 8)^(crc32_table[(crc ^ *string++)&0xff]);

      return crc;
}
```

9.3. 用户参与升级

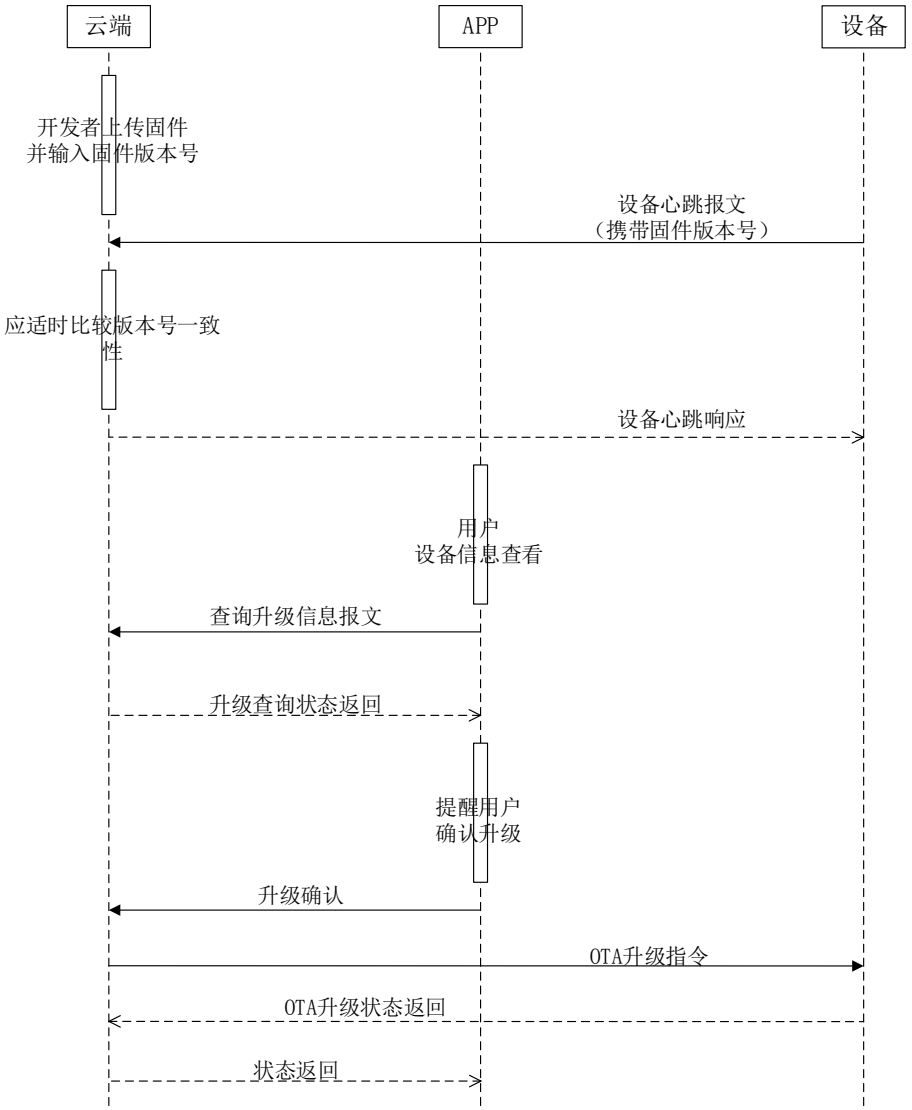


图 9-1 由用户确认的升级过程

9.4. 用户无参与升级

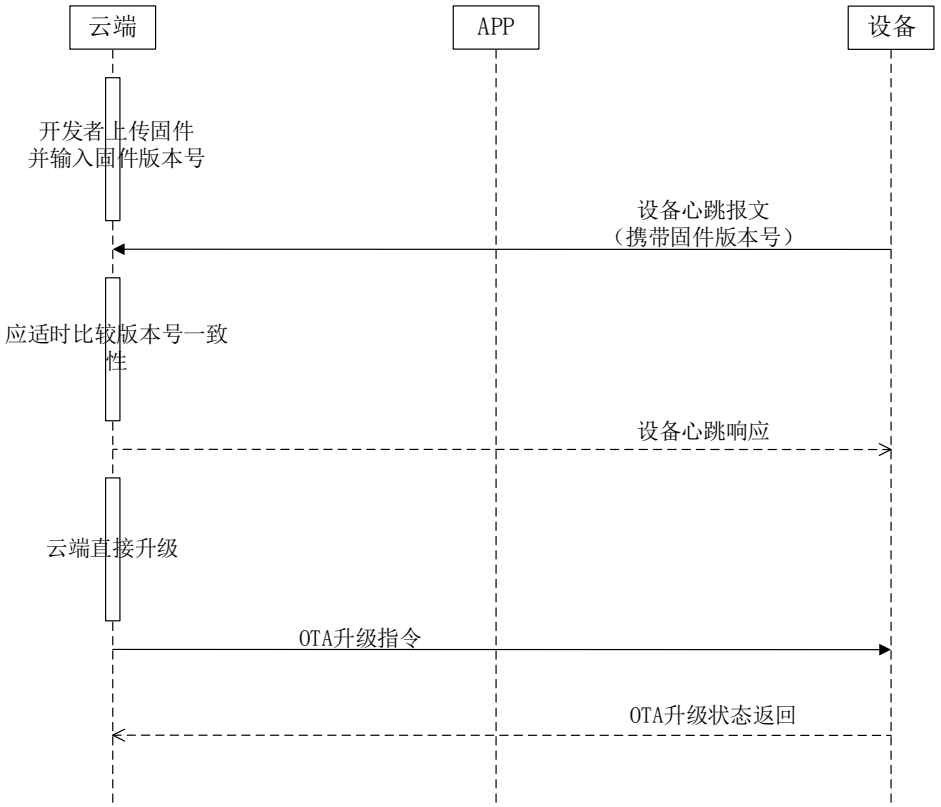


图 9-2 由云端发起的升级过程

9.5. 升级信息查询

在用户进入设备的详细信息页面时，应当主动查看当前设备的固件版本是否需要升级，如果有可升级资源，应当提醒用户。

升级指令 type=7

请求：（Cloud-->设备）|（OPT=无数据不加密，Payload=session_key 密钥加密）：

[时间戳] //4 字节 int 类型时间戳，小端

```
{
    "cmd":"ota",
    "serial":1,
    "data":{
     "feedid":"1234567890123456",
     "productuuid":"XXXXXX",
    "version":1,                                    //用户输入或系统生成的固件版本序号
     "versionname":"1.2.1.20151030_release",  //用户输入的固件版本名
```

```
    “crc32”:12345678,                    //CRC32 计算出的固件校验值，32bit 无符号整型，
    "url":"******"                       //固件存贮的资源地址，http
    }
}
```

响应：（设备-->Cloud）|（OPT=无数据不加密，Payload=session_key 密钥加密）：

```
[时间戳]  //4 字节 int 类型时间戳，小端在前
{
    "code":0,
    "serial":1,                  //控制指令序号
    "msg":"success"
}
```

9.6. 升级状态上报 type=8

请求：（设备--> Cloud）|（OPT=无数据不加密，Payload=session_key 密钥加密）：

报文可在升级状态发生变化时上报。(设备如果超过 10 分钟没有向云端上报升级状态，云端会默认设备升级失败，建议设备考虑此因素)

```
[时间戳]  //4 字节 int 类型时间戳，小端
{
    "cmd":"otastat",
    "data":{
        "feedid":"1234567890123456",
        "productuuid":"XXXXXX",
        "status":1,
        "status_desc":"固件下载中",
        "progress":100
    }
}
```

响应：（Cloud -->设备）|（OPT=无数据不加密，Payload=session_key 密钥加密）：

```
[时间戳]  //4 字节 int 类型时间戳，小端
{
    "code":0,
    "msg":"success",
}
```

字段	说明	备注
status	设备升级时的当前状态	0：固件下载中； 1：固件安装中； 2：固件升级完成；（如果需要重启，最好重启后再上报） 3：固件升级失败；(无论下载失败还是安装失败，都不能影响设备正常使用，最好在 status_desc 字段中注明失败原因)
progress	当前状态的进度	数字标识，百分比的分子，代表当前操作的进度。

10. 定时 type = 13

请求：（APP/云端-->设备）|（OPT=不加密，Payload=采用设备的 LocalKey（APP），SessionKey（云端）密钥加密）.

```
typedef struct {
    int         len;
    char        cmd[];   //设备的二进制指令,云端使用 LUA 脚本将 JSON 转化成二进制格式
}cmd_t;

typedef struct {
    int         len;
    char        num;
    cmd_t     cmds[];
}cmd_trank_t;

typedef struct {
    char len;
    char exp[];
}time_exp_t;

typedef struct {
    char feedid[32];
    char name[128];
    int id;
    char state;                    //0: 停止 1：开始
    char type;                     //0：用户设置定时   1：遥控器设置定时   2：控制面板
设置定时
    time_exp_t exp;
    cmd_trank_t c_trank;
}base_task_t;

typedef struct {
    int id;
    char state;                    //0: 停止 1：开始
    char type;                     //0：用户设置定时   1：遥控器设置定时   2：控制面板
设置定时
```

```
    time_exp_t exp;
    cmd_trank_t c_trank;
}simplified_base_task_t;

typedef struct {
    char feedid[32];
    char name[128];
    int id;
    char state;              //0: 停止  1：开始
    char type;               //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定
时
    time_exp_t start_exp;
    time_exp_t stop_exp;
     char repeat_flag;       // 0:单次   1:重复
    char period_type;        //0:周   1:月
    char period[31];         //0:本天不重复   1:本天重复。 如果为周则只 0~6 有效
    cmd_trank_t start_cmd_trank;
    cmd_trank_t stop_cmd_trank;
}period_task_t;

typedef struct {
    int id;
    char state;   //0: 停止  1：开始
    char type;   //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定时
    time_exp_t start_exp;
    time_exp_t stop_exp;
    char repeat_flag;   // 0:单次   1:重复
     char period_type;   //0:周   1:月
    char period[31]; //0:本天不重复   1:本天重复。 如果为周则只 0~6 有效
    cmd_trank_t start_cmd_trank;
    cmd_trank_t stop_cmd_trank;
}simplified_period_task_t;

typedef struct {
char feedid[32];
    char name[128];
    int id;
```

```
    char state;   //0: 停止  1：开始
    char type;   //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定时
    int action_interval;    //执行持续时间
    int sleep_interval;      //空闲持续时间
    int times;    //循环次数
    cmd_trank_t action_c_trank;
    cmd_trank_t sleep_c_trank;
}cycle_task_t;

typedef struct {
    int id;
    char state;   //0: 停止  1：开始
    char type;   //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定时
    int action_interval;    //执行持续时间
     int sleep_interval;      //空闲持续时间
     int times;    //循环次数
    cmd_trank_t action_c_trank;
    cmd_trank_t sleep_c_trank;
} simplified_cycle_task_t;

typedef struct {
char feedid[32];
    char name[128];
    int id;
    char state;   //0: 停止  1：开始
    char type;   //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定时
    int interval;    //倒计时时间
    cmd_trank_t action_c_trank;
    cmd_trank_t stop_c_trank;
}countdown_task_t;

typedef struct {
    int id;
    char state;   //0: 停止  1：开始
    char type;   //0：用户设置定时   1：遥控器设置定时   2：控制面板设置定时
    int interval;    //倒计时时间
    cmd_trank_t action_c_trank;
```

```
    cmd_trank_t stop_c_trank;
} simplified_countdown_task_t;

typedef union {
    base_task_t base_task;
    period_task_t period_task;
    cycle_task_t cycle_task;
    countdown_task_t countdown_task;
} task_u;

typedef union {
    simplified_base_task_t simplified_base_task;
    simplified_period_task_t simplified_period_task;
    simplified_cycle_task_t simplified_cycle_task;
    simplified_countdown_task_t simplified_countdown_task;
} simplified_task_u;

typedef struct{
     int   mode;     // 0:基本本地定时，1:时间段定时，2:循环定时，3:倒计时定时
    int version;
    task_u type;
} task_t;

typedef struct{
    int   mode;     // 0:基本本地定时，  1:时间段定时，    2:循环定时，    3:倒计时定
    int version;
    simplified_task_u simplified_type;
} simplified_task_t;
```

字段 version,在增加定时任务时 version 为 0，以后每次修改定时任务 version 加 1。
在上报定时快照时云端要比较各个定时的 version，如果不一致则需要使用修改定时任务达到设备端快照和云端快照数据的一致。

```
typedef struct {
    unsigned int  timestamp;
    unsigned int  biz_code;
    unsigned int  serial;
     int task_num;
    task_t tasks[];
```

}time_task_req_t;

```
typedef struct {
    unsigned int  timestamp;
    unsigned int  biz_code;
    unsigned int  serial;
     int task_num;
    char feedid[32];
    simplified_task_t simplified_tasks[];
} simplified_time_task_req_t;
```

响应：（设备--> APP/云端）|（OPT=不加密，Payload=采用设备的 LocalKey（APP），SessionKey（云端）密钥加密）

```
typedef struct {
    unsigned int    timestamp;
    unsigned int    biz_code;
    unsigned int    serial;                 //同下发的 serial
    int code;
} time_task_rsp_t;
```

time_exp 格式：秒_分_时_日_月_周_年

1: 如果某列没有明确值, 则以*代替

2: 如果需要按某一固定时间周期执行， 则在列值前加上/

3: 某一列有多值， 则以,分开

4: 举例如下

编号	任务类型	表达式	说明
1	定时执行一次	0_1_22_26_2_*_2015	2015 年的 2 月 26 号 22 点 01 分执行
2	每周定时执行	0_1_22_*_*_1,3,5_*	每周 1,周 3，周 5 的 22 点 01 分执行
3	每月几号定时执行	0_1_22__11,15_*_*_*	每月 11,15 号 22 点 01 分执行
4	每天定时执行	0_1_22_*_*_*_*	每天 22 点 01 分执行
5	间隔几分钟几个小时几天执行	/2_*_*_*_*_*_* *_/2_*_*_*_*_* *_*_/2_*_*_*_* *_*_*_/2_*_*_* *_*_*_*_*_/2	每隔 2 秒钟 每隔 2 分钟 每隔 2 小时 每隔 2 天 每隔 2 年

biz_code 说明：

功能	请求时 biz_code	响应时 biz_code	备注
查询是否具备定时功能	1090	190	云端/app 请求， 设备响应
增加	1091	191	云端/app 请求， 设备响应
修改	1092	192	云端/app 请求， 设备响应
删除	1093	193	云端/app 请求， 设备响应
查询	1094	194	云端/app 请求， 设备响应
停止	1095	195	云端/app 请求， 设备响应
重启	1096	196	云端/app 请求， 设备响应
上报执行结果	1097	197	设备请求， 云端/app 响应
上报新增定时任务	1098	198	设备请求， 云端/app 响应
上报删除定时任务	1099	199	设备请求， 云端/app 响应
上报修改定时任务	10100	1100	设备请求， 云端/app 响应
上报定时快照	10101	1101	设备请求， 云端/app 响应

等待响应超时时间：(500ms)。

查询是否支持定时功能　biz_code = 1090

请求的报文格式：

```
typedef struct {
    unsigned int  timestamp;
    unsigned int  biz_code;
    unsigned int  serial;
}time_task_query_t;
```

回应报文格式：

参考 time_task_rsp_t 报文内容

rsp 报文字段说明

字段	说明	备注
code	0 表示支持定时 非零　表示不支持	

10.1. 增加定时任务 biz_code = 1091

请求的报文格式：

参考 time_task_req_t 报文内容。

回应报文格式：

参考 time_task_rsp_t 报文内容。

当下发多个定时任务时，均返回一个 Code. 即设备端对于所有的定时任务，要么全部成功，要么全部失败。

rsp 报文字段说明

字段	说明	备注
code	0 表示成功 非零 表示不成功	

10.2. 删除定时任务 biz_code = 1093

请求的报文格式：

```
typedef struct {
unsigned int      timestamp;
unsigned int      biz_code;
unsigned int      serial;
int               ids_num;
int               ids[];
}time_task_ids_req_t;
```

如果 ids_num == 0,为 all,不影响还未上报的定时任务。

响应报文格式：

```
typedef struct {
unsigned int      timestamp;
unsigned int      biz_code;
unsigned int      serial;
int               code;   // code 为 0，表示删除成功；为非 0，表示有删除不成功的
int               ids_num;
int               ids[];
}time_task_ids_rsp_t;
```

code 为非时， ids 字段填入没有删除成功的 id 列表。

10.3. 修改定时任务 biz_code = 1092

请求的报文格式：

参考 time_task_req_t 报文内容。

回应报文格式：

参考 time_task_rsp_t 报文内容。

id 不能修改，其余哪个字段不为空修改哪项。

10.4. 查询定时任务 biz_code = 1094

请求的报文格式：

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    int                 ids_num;
    int                 ids[];
}time_task_ids_req_t;
```

如果传递的 ids_num 为 0 时,表示查询所有的任务， 这种情况不影响还未上报的定时任务。

回应报文格式:

参考 simplified_time_task_req_t 报文内容。

注意：设备新增定时但是没有 id 的不上报。

10.5. 停止定时任务 biz_code = 1095

请求的报文格式：

```
typedef struct {
    unsigned int        timestamp;
    unsigned int        biz_code;
    unsigned int        serial;
    int                 ids_num;
    int                 ids[];
}time_task_ids_req_t;
```

如果传递的 ids_num 为 0 时,表示停止所有的任务，但是不影响还未上报的定时任务。

响应报文格式：

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
    int               code;
    int               ids_num;
    int               ids[];
}time_task_ids_rsp_t;
```

如果 code 非 0， ids 字段填入没有停止成功的 id 列表。

rsp 报文字段说明

字段	说明	备注
code	0 表示停止成功 非零 表示没有停止成功	
ids_num	Id 的个数	

10.6. 重启定时任务 biz_code = 1096

请求的报文格式：

```
typedef struct {
     unsigned int      timestamp;
     unsigned int      biz_code;
    unsigned int      serial;
    int               ids_num;
    int               ids[];
}time_task_ids_req_t;
```

请求报文字段说明

响应报文格式：如果 code 非 0， ids 字段填入没有重启成功的 id 列表。

```
typedef struct {
    unsigned int      timestamp;
    unsigned int      biz_code;
    unsigned int      serial;
```

```
    int             code;
    int             ids_num;
    int             ids[];
}time_task_ids_rsp_t;
```

rsp 报文字段说明

字段	说明	备注
code	0 表示重启成功 非零 表示没有重启成功	
ids_num	Id 的个数	只有 code 为非 0 时值有效

10.7. 上报执行结果 biz_code = 1097 （设备主动上报到云端）

请求（设备主动上报到云端）的报文格式：

```
typedef struct {
    unsigned int    timestamp;
    unsigned int    biz_code;
    unsigned int    serial;
    int num;
    struct {
            int         id;             //定时的 id
            char        step;           //1：第一步   2：第二步   3：第三步
            int         code;
    }result[];
}time_task_result_report_t;
```

响应（云端反馈给设备）的报文格式：

```
typedef struct {
    unsigned int    timestamp;
    unsigned int    biz_code;
    unsigned int    serial;
    int             num;
    struct {
        int         id;         //定时的 id
        char        step;       //1：第一步 2：第二步  3：第三步
    }received[];
}time_task_result_report_rsp_t;
```

如果设备没有收到哪条 task 的响应，则需要重新上报。

报文字段说明

字段	说明	备注
code	0 表示执行成功 非零 表示没有执行成功	云端给出非成功的具体 code 值

10.8. 上报增加定时 biz_code = 1098 （设备主动上报到云端）

请求报文格式同增加定时的报文。
设备新增的定时，每次只上报一个，上报时 id 是空的，云端返回新分配的 id，设备记录。

响应的报文格式

```
typedef struct {
    unsigned int    timestamp;
    unsigned int    biz_code;
    unsigned int    serial;              //同上报的 serial
    int             code；
    int             id;
} time_task_report_new_rsp_t;
```

10.9. 上报删除定时 biz_code = 1099 （设备主动上报到云端）

请求和响应的报文格式同删除定时的报文。

10.10. 上报修改定时 biz_code = 10100 （设备主动上报到云端）

请求和响应的报文格式同更新定时的报文。

10.11. 上报定时快照 biz_code = 10101 （设备主动上报到云端）

设备上报定时快照的时机有六个：
设备连入网络;
增加定时任务成功;
删除定时任务成功;
修改定时任务成功；
停止定时任务成功；

重启定时任务成功。
对于上报增加定时成功、上报删除定时成功、上报修改定时成功这三个时机是否上报定时快照，协议不做硬性要求。
上述六个时机出现之后，设备可以等待几秒钟再执行上报定时快照动作，这样规定是为了方便云端和设备端实现，具体几秒钟各个产品可以根据自己实际需要确定，协议不做硬性规定。
云端在上报定时快照成功之后，需要对比定时快照所描述的设备端定时任务列表和云端数据库中的定时任务列表。如果设备端多了某定时任务那么云端必须通过“删除定时任务”删除之；如果设备端少了某定时任务那么云端必须通过“增加定时任务”增加之；如果设备端某定时任务的内容与云端该定时任务的内容不一致，那么云端必须通过“修改定时任务”修改之。通过这种方式使得设备端的定时任务列表数据与云端的定时任务列表数据保持一致，以云端的定时任务列表数据为准。

请求（设备上报到云端）的报文格式：

参考 simplified_time_task_req_t 报文内容。

回应（云端反馈给设备）报文格式：
参考 time_task_ids_rsp_t 报文内容。int ids[]存放的是云端收到的快照中各个 task 的 id。设备比对这些 id 和自己上报上去的是否一致，如不一致需要重新上传设备定时快照。

11. 状态上报 type=14

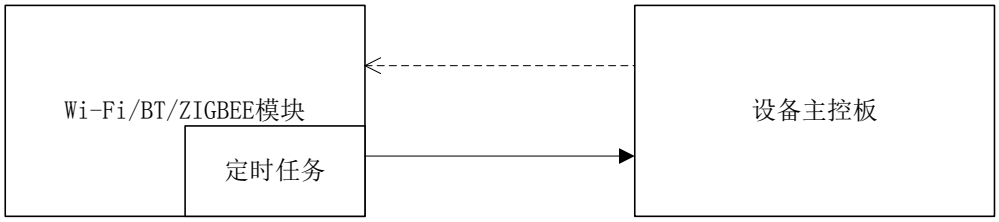
通过云端对设备的控制，及控制及响应信息直接经过云端，可以存贮在云端。
局域网控制时，控制及响应信息存贮在 APP 上，APP 适当的时机可以把此类信息同步至云端。

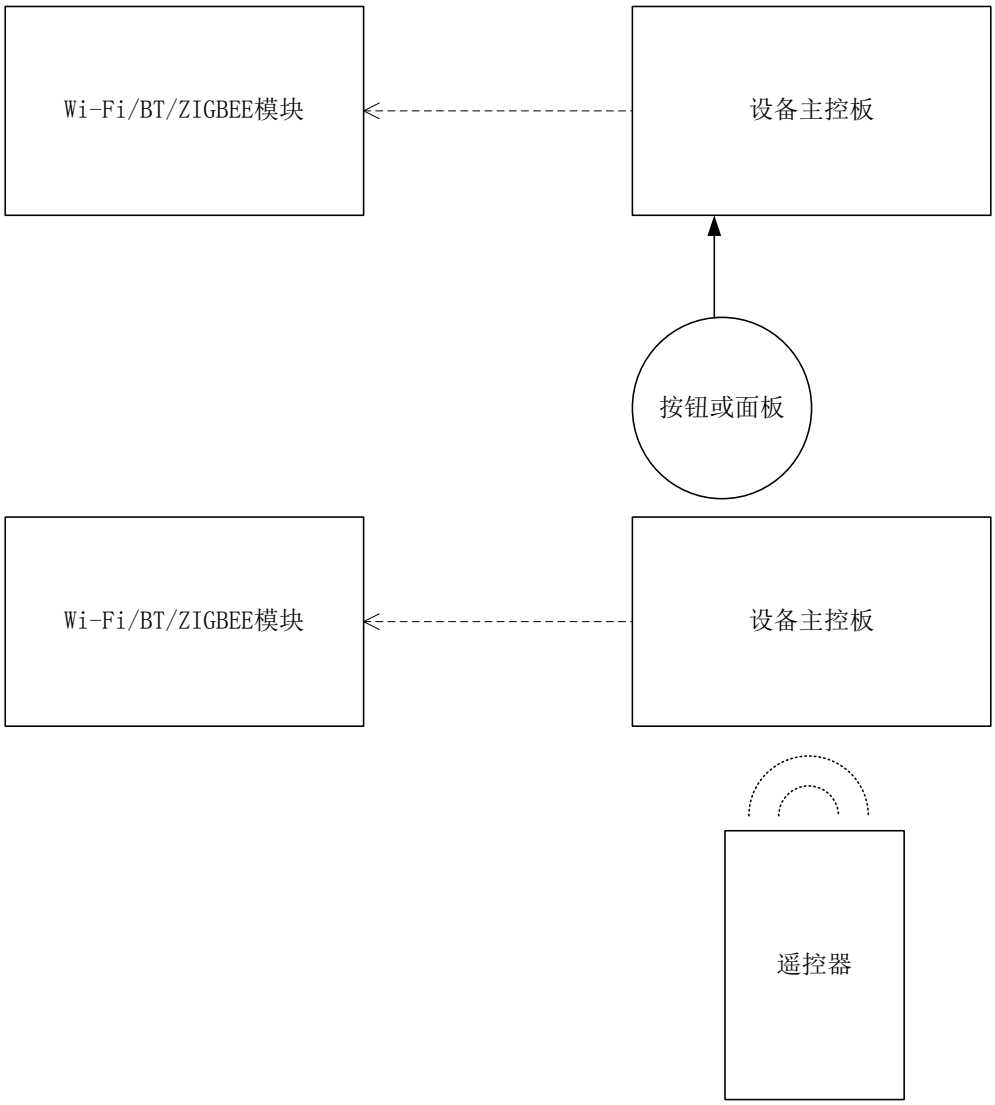
在实际的使用中对设备的控制方式还有其它方式如：
通过设备自带的面板或按钮控制。
通过设备的遥控器控制。

以上汇总为三种类型
通过 Joylink 协议的 APP 控制
通过设备自带的面板或按钮控制。
通过设备的遥控器控制。

另外，对于定时类任务等不是即时得到控制结果的情况，及执行状态是否成功也需要返回。

以上可以统称为 APP 在没有进行即时控制的情况下，设备执行了动作。在这种情况下，我们建议无论控制指令是否来自于模块方，其执行动作的响应都应该通知模块。





模块端视能力存贮多条异步的控制响应，至少应是一条。在模块能与云端建立联接时报该消息上报云端。

注：模块端应与主控板之间约定控制的类型传输，否则只能区分定时任务与其它异步控制方式。

```
typedef struct {
    unsigned int      timestamp;//消息上报时间
    unsigned int      biz_code;
    unsigned int      serial;
    unsigned char     feedid[32];//设备的唯一标号
     unsigned char    controltype;//0:定时任务，1：面板或遥控器控制
    unsigned int      tasktimestamp;//消息执行时间
    char id[32];      //任务 ID,面板或遥控器控制时可为 0；
```

```
    unsigned char       resp[]; //执行的结果
} async_task_response;
```

12. 获取 product_uuid

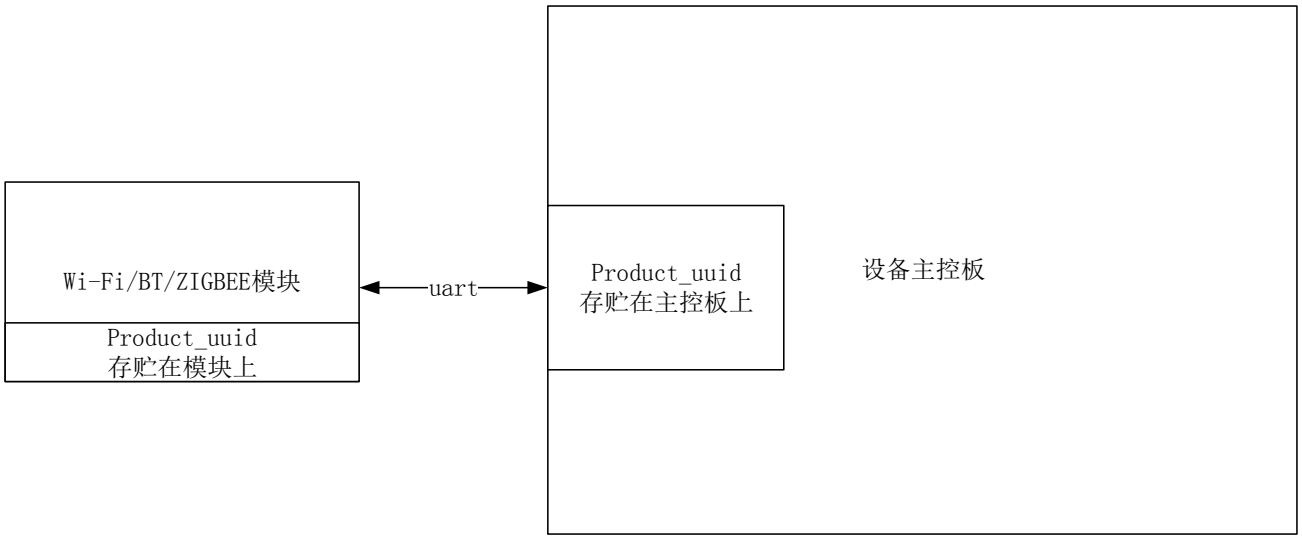


图 12-1 获取控制板图表 uuid

在目前的实际应用中，很多情况下无线模块并不是与主控板集成在一起的，而是如上图所示，在设备内部通常内置一个无线模块，而无线模块与主控板之间能常采用 UART 物理连接，通过 Mobile 的 APP 把指令发送给无线模块，该指令又通过 UART 传给主控板完成控制，控制的响应是一个逆向的过程。

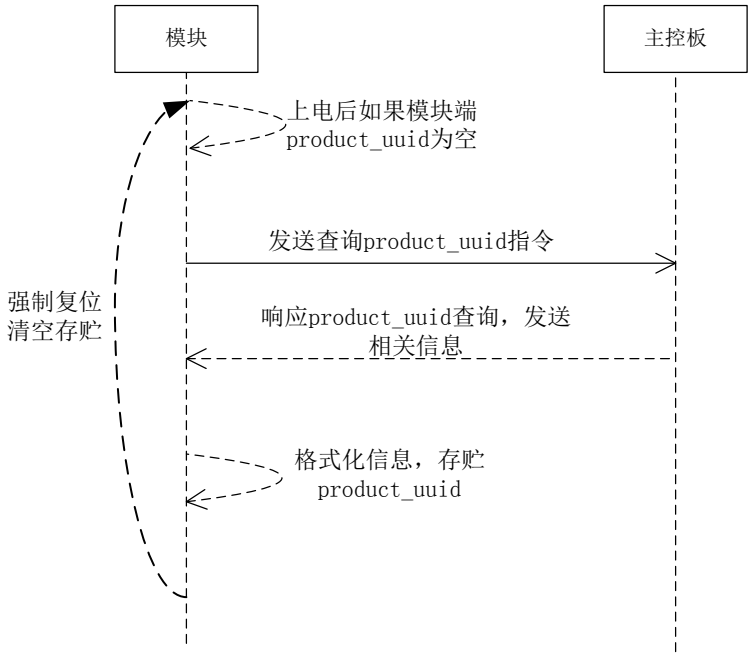
通常设备上电通知以及设备发现过程中向 APP 端发送的报文中需要携带 product_uuid 用来标识某类、某批次产品。相同的产品及 product_uuid 是相同的。

product_uuid 在本协议中通常是长度为 6 个 byte 的字符串，如：XDFSSD

product_uuid 在目前通常是存贮在模块中的，即在生产某一类、某一批次的产品时需要在所有的模块中预先写入 product_uuid。这样的做法导致的一个问题是，不同品类及批次的产品即使采用相同的无线模块也需要根据不同的应用而写入不同的内容。另外，主控板上通常是存贮着 product_uuid 的相关信息的（其格式也许与协议约定的有所不同）。为了统一无线模块端的内容，利用主控板的信息获取 product_uuid，特提出以下应用方案。

注：当无线块模需要通过 UART 从主控板获取 product_uuid 时需要特别约定查询及响应指令，这种二进制的指令建议模块商与主控板厂家直接约定，因为本协议指定一种二进制指定有可能与主控制的正常控制及响应指令是冲突的。

建议的流程如下：



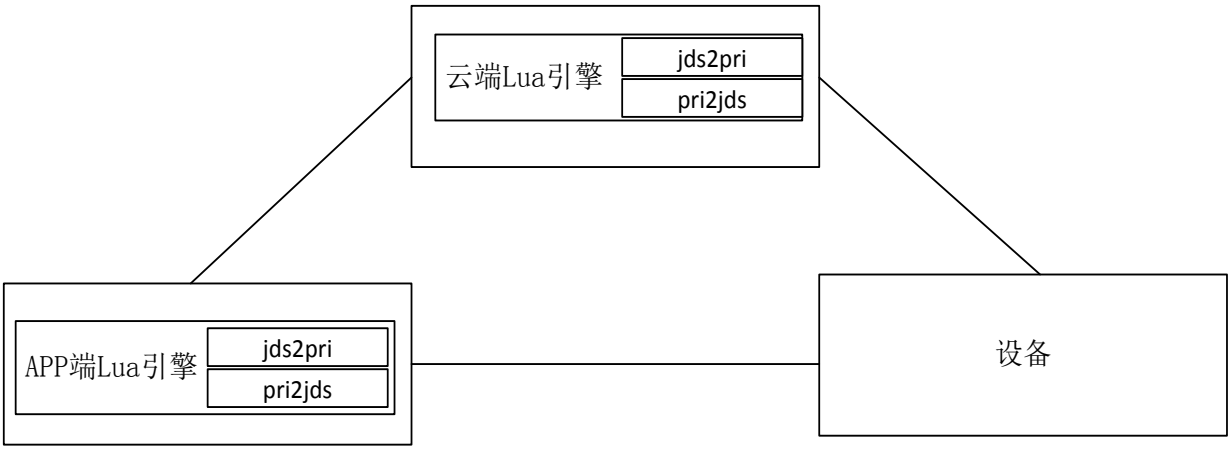
图表 12-1 获取 UUID 流程

13. Lua 脚本规范

考虑到两个因素，其一，各厂家不同的设计理念，不同的设备厂家其设备控制指令集在相同的品类中不同。其二，设备本身的原始指令集往往是二进制的，不适合应用层解析展现。Joylink 协议应用层如前面章面所述对控制指令的格式有一定的约束，采用脚本转换方案时，应用层上行与下行指令的格式是 Json 的，其指令主体为：

```
“data”:{
    “streams”: [
        { “stream_id”: “switch”, “current_value”: “1” }
    ],
    "snapshot": [
        {"stream_id":"switch"， "current_value":"0"}
    ]
}
```

因此应用层标准指令会需要转换成各设备厂家自己的指令集。



图表 13-1 lua 转换指令流程示意

如图所示：在 lua 引擎的脚本中必须要实现 jds2pri 函数，该函数用来实现应用层协议转设备私有指令集。必须实现 pri2jds 函数，该函数实现私有指令转应用层指令协议。这两个函数的说明在本章后面章节描述。

13.1. Lua 引擎内置 cjson

joylink 采用的 Lua 引擎是经过优化的，在性能及功能上有所改变。内置了 cjson 指令。如下面的代码所示 cjson 即是内置的。Lua 脚本可能用到的与 json 相关的函数如下：

cjson.decode(cmd) ：cmd 为 json 对像，函数返回 table。

示例代码：

```
function decode( cmd)
        local tb
        if cjson == nil then
            cjson = (require 'JSON')
            tb = cjson:decode(cmd)
        else
            tb = cjson.decode(cmd)
        end
        return tb
end
```

注：cjson = (require 'JSON')为本地调试用代码。在脚本中也建议调用。

将 JD 标准指令转换为设备（模块）私有指令

函数定义:

```
function    jds2pri( bizcode, cmd )
—厂商自身实现
end
```

函数功能说明：

将 JD 标准命令字符串解析成设备（模块）的私有协议字节数组。

输入参数 2 个：

bizcode： int 类型，业务编码，见 3.6 章节。

cmd： 需要转换的 json 字符串

返回值 3 个：

1: int 类型 error code [0:success；-1：fail]

2: int 类型，转换完成的设备私有指令字符串长度

3: 字符串，转换完成的私有指令

将设备（模块）的私有命令转换为 JD 标准命令 JSON 字符串

函数定义:

```
function pri2jds( bizcode, length, bin )
—由厂商自身实现
```

end

函数说明：
将设备（模块）的私有协议解析为 JD 标准的命令 json 字符串

输入参数 3 个：

biz_code： int 类型。无法确定 biz_code 的产品，biz_code 可填入 0

length： int 类型，传入的字节数组长度

bin ： 字节数组，内容为私有指令

返回值 3 个：

1：int 类型， error code,。[0:success；-1：fail]

2：json 字符串，转换完成的标准命令字符串

3：int 类型，业务编码，见 3.6 章节。

标准 JSON 字符串示例

设备控制请求

```
{
"streams": [{"current_value":"1","stream_id":"switch"}], // 控制命令
    "snapshot": [{"current_value":"0","stream_id":"switch"}] // 当前快照，包含全属性
}
```

设备控制响应

```
{
    "code": 0, // 响应结果
    "streams": [   // 快照
        {
            "current_value": "1",
            "stream_id": "switch"
        }
    ],
    "msg": "控制成功"
}
```

获取设备快照请求

无

获取设备快照响应

```
{
    "code": 0,
    "streams": [
        {
            "current_value": 24,
            "stream_id": "time"
        },
        {
            "current_value": 0,
            "stream_id": "schedule"
        },
        {
            "current_value": 1,
            "stream_id": "switch"
        },
        {
            "current_value": 3,
            "stream_id": "workstatus"
        },
        {
            "current_value": 1,
            "stream_id": "type"
        }
    ]
}
```

设备数据上报

同【获取设备快照响应】

云菜谱下发

```
{
    "at": 1441458319881,    // 时间
    "code": 1050,
    "feed_id": 144093466370504020,
    "subTask": [
        {
            "id": 11760,    // 菜谱步骤 ID
            "index": 1,     // 执行顺序(step)
            "streams": [
                {
```

```
                "current_value": "160",
                "stream_id": "pretemperature"
            },
            {
                "current_value": "5",
                "stream_id": "pretime"
            },
            {
                "current_value": "1",
                "stream_id": "waitime"
            },
            {
                "current_value": "0",
                "stream_id": "upheat"
            },
            {
                "current_value": "0",
                "stream_id": "downheat"
            },
            {
                "current_value": "0",
                "stream_id": "voice"
            }
        ]
    },
    {
        "id": 11761,   // 菜谱步骤 ID
        "index": 2,
        "streams": [
            {
                "current_value": "160",
                "stream_id": "temperature"
            },
            {
                "current_value": "30",
                "stream_id": "time"
            },
            {
```

```
                        "current_value": "0",
                        "stream_id": "upheat"
                    },
                    {
                        "current_value": "0",
                        "stream_id": "downheat"
                    },
                    {
                        "current_value": "0",
                        "stream_id": "voice"
                    }
                ]
            }
        ],
        "task_id": 46     // 菜谱 ID
}
```

云菜谱下发响应

```
{
    "code": 0, // 响应结果
    "msg": "控制成功"
}
```

14. 硬件错误码

在模块与主控板分离式设计时，当模块端给主控板时，主控板返回的错误码。

错误码	含义
0	成功
1	参数错误
2	设备拒绝访问

15. 响应码定义

响应码	说明
-1	认证失败
-2	无认证信息
-3	设备不在线
-4	不支持
0	成功
1	连接不上服务器
2	设备拒绝访问
3	定时操作失败
4	修改设备属性错误
5	设备授权失败
6	固件升级失败
7	数据上报失败
8	联动操作失败
9	子设备操作失败
10	主控超时
11	主控失去连接
12	脚本校验错误
13	内存错误
14	脚本执行错误
15	权限不足
16	已被绑定，请联系管理员
17	已达上限
18	版本不兼容
19	子设备不存在
20	子设备离线
21	子设备无响应
22	正在下载
23	文件损坏
24	写入失败
100	不支持的操作