MIIO/OT Linux 客户端软件架构与接口

Date	Version	Changes
2015/04/20	0.1	Initial Draft
2015/04/22	0.2	Two separate ack channel.
2015/04/30	0.3	Change Channel name
2015/05/20	0.4	Add AP smart config
2015/06/09	0.5	Add mosquitto example
2015/06/19	0.6	1. 配置文件说明 2. 设备软件升级文档 3. 设备电子说明书示例
2015/07/07	0.7	1. 快连出错和重试说明 2. FAQ
2015/08/20	0.8	增加 六，路由器密码修改
2015/09/16	0.9	增加 RAW socket API
2015/09/30	1.0	升级协议细化，增加 4.3 上报升级状态
2015/12/01	1.1	修改 十，设备动态申请 did 和 key
2015/12/17	1.2	设备睡眠接口
2016/04/20	1.3	设备重置步骤说明
2016/05/24	1.4	1. 设备固件升级文档更新，补充静默下载/静默升级相关内容 2. 对命令的应答 json 字符串标准化
2016/07/05	1.5	1. 升级协议细节改动：上报统一使用 props，而不是 prop；增加升级失败的状态上报细节； 2. 版本号规则改为：x.x.x_aaaa； 3. 十二，设备需要实现的其他接口
2017/02/28	1.6	1.增加对隐藏 wifi 的支持 2.增加常用功能接口 （十四，MIIO Client 提供的其他功能接口）
2017/03/30	1.7	1、添加蓝牙快连方式说明

		2、添加移植步骤说明，支持快连时配置时区
2017/04/12	1.8	1、添加配置时区接口说明 2、添加配置 log 级别接口说明
2017/09/15	1.9	1、添加 miio_agent 使用说明
2018/03/07	2.0	为 mosquitto 版本添加如何获取时间等功能接口的说明
2018/10/16	2.1	1、增加 MQTT 版本注意事项 2、增加二维码方式扫描配网使用说明
2019/08/05	2.2	1、设备离线状态通知
2019/09/10	2.3	1、增加设备接入 SPEC
2019/11/12	2.4	1. 使用新版本 SDK 4.0.4 2. 新版本 SDK 支持 tls 功能 3. 更改 rpc id 说明 4. 增加 FAQ
2019/12/19	2.5	1. 增加 FAQ(产品固件对于不支持的 RPC 命令的处理)

一，概述

二，软件架构

三，本地编程接口/通讯接口

1. 消息语义

1.1 属性、事件上报

1.1.1 设备电子说明书（profile）

1.1.2 上报属性变化

1.1.3 上报事件

1.2 云端或手机下达的动作、命令

1.2.1 下达动作命令

1.2.2 获得设备属性

1.3 属性上报或者命令的回应

1.3.1 id 字段

1.3.2 回应具体格式

2. 设备接入 SPEC

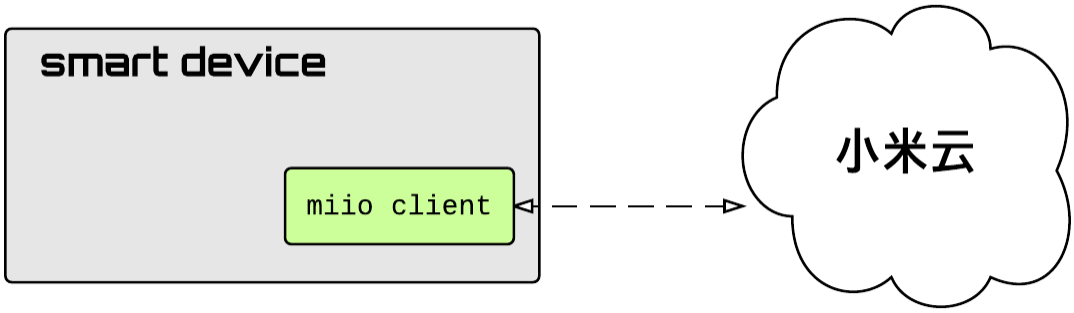
2.1 profile 与 spec 中 method 对应关系
2.2 属性、事件上报
2.2.1 上报属性
2.2.2 上报事件
2.3 云端或手机下达动作、命令
2.3.1 读属性
2.3.2 写属性
2.3.3 执行方法
2.4 code 状态码定义
3. MQTT API
4. RAW socket API
四，设备配置文件说明
1. /etc/miio/device.conf
2. /etc/miio/device.token
3. /etc/miio/wifi.conf
4. /etc/os-release
五，安装和移植
1，怎么安装
2，怎么移植
3，怎么运行
六，快连
1. AP 方式快连
2. 蓝牙方式快连
3. 二维码方式快连
4. 快连出错和重试
七，路由器密码修改
八，设备联网状态广播
九、设备离线状态通知
十，设备固件/软件升级
1. 升级框架
2. 版本号规则
3. 软件版本号获取
4. 详细命令解析
4.0 名词解释
4.1 升级开始

4.2 查询升级状态
4.3 上报升级状态
4.3.1 升级完成后状态上报
4.3.2 升级失败的状态上报
4.4 上报升级进度
4.5 查询升级进度
5. 每次升级多个文件
十一，设备重置
十二，设备动态申请 did 和 key
1. 触发
2. 大致流程
十三，设备 suspend/resume 接口
十四，设备需要实现的其他接口
1. miIO.reboot
2. miIO.restore
十五，设备与小米云通讯（经由 miio 客户端）典型流程
十六，MIIO Client 提供的其他功能接口
一、查询接口：
1、查询本地时间：
2、查询国别域名：
3、查询 miio_client 状态：
二、配置接口
1、配置时区信息
2、配置 log 打印级别
十七，FAQ
十八，附录：
1. 一个较完整的_otc.info 例子：
2. mosquitto 编程例子
例子 1：
例子 2：
3．设备电子说明书示例
设备型号(model)
属性
事件

方法

一，概述

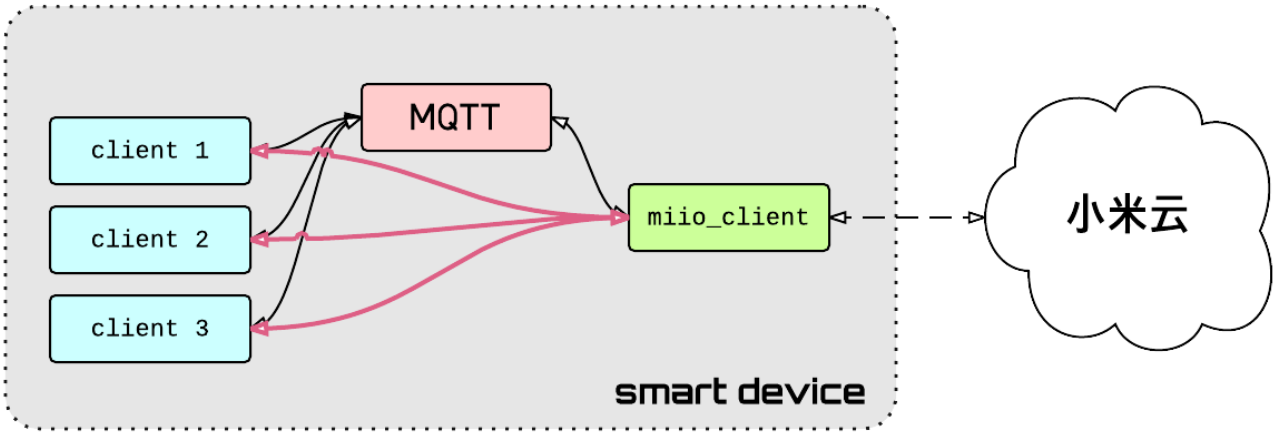
MIIO 客户端软件跑在 Linux 系统之上，连接具体智能家居设备和小米云端。



（图 1）

MIIO 客户端软件以 daemon 的形式在后台运行。miio 客户端通过一定的协议跟小米云通讯，上报必要的信息、事件（上行），和执行云端下达的动作、命令（下行）。miio 客户端保证跟小米云之间的通讯是加密的，可靠的。

二，软件架构



（图 2）

图 2 是本地设备上的软件通讯图。miio_client 有两套 API 跟用户进程通信，用户可以任意选用一种 API。

1. **MQTT API**（如图 2 黑色箭头所示），用户进程通过 MQTT 跟 miio_client 通信，MQTT 能带来一些额外的好处比如 QoS。发送和接收的消息都是文本消息，消息语义下面章节会介绍。
2. **RAW socket API**（如图 2 粉红色箭头所示），用户进程直接跟 miio_client 使用标准的 socket 接口进行通信，端口 54322。发送和接收的消息都是文本消息，消息语义下面章节会介绍。
3. **RAW socket + Agent**（如图 2 粉红色箭头所示），用户进程跟 miio_agent 使用标准的 socket 接口进行通信，端口 54320，miio_agent 负责和 miio_client 进行通信。

之所以引入 Agent 模块，是因为在图 2 中，对于从云端接收到的命令，miio_client 会使用广播的方式发送给所有用户进程，这样对不关心这条命令的进程会造成误唤醒，Agent 模块实现了一个消息订阅和分发的功能，用户进程首先根据需要向 Agent 模块订阅感兴趣的消息，由 Agent 模块负责和 miio_client 的通信。

Agent 是小米工程师自行开发的开源项目，用户可以通过 github 获得：

https://github.com/MiEcosystem/miio_linux_open

对于已经使用 2 的用户，只需要将 socket 端口改为 54320，并且在建立连接之后注册感兴趣的消息，即可使用 Agent 模块的功能。消息注册格式请参考 agent_client.c。

MIIO client 连接小米云。小米云对于本地其他的应用来说是透明的，本地其他的应用只看到 MIIO client。

关于 MQTT 更多的信息，可以参考官方网站：http://mqtt.org/documentation ，我们系统里面集成的是 MQTT 的 C 语言版本实现 mosquitto，更多资料可以参考这里：http://mosquitto.org/ 。

三，本地编程接口/通讯接口

1. 消息语义

无论是使用 MQTT API 还是标准 socket API，发送和接收的消息都是文本字符串消息。字符串流遵循类似于 JSON-RPC 2.0 的格式。http://www.jsonrpc.org/specification

消息流分上行和下行，上行主要是设备属性、事件上报；下行主要是云端或者手机下发的命令。

1.1 属性、事件上报

1.1.1 设备电子说明书（profile）

设备电子说明书由固定的几部分组成：类型、属性、方法、事件、日志。

智能设备在与小米云进行通讯（接受云端下达的方法，汇报属性、事件等）之前，必须得把电子说明书定义清楚，并且输入到小米云后台。每次小米云与设备的通讯都会看是否符合电子说明书的定义。电子说明书必须由小米方输入到小米云后台，请联系小米相应工程师拿到电子说明书模板以及进行提交工作。

设备电子说明书示例请参考附录 8.3。

1.1.2 上报属性变化

上报属性是上行的交互动作，由设备发送给云端。例如：

```
{"method":"prop.power","params":["off"]}
```

```
{
        "method": "prop.power",
       "params": ["off"]
}
```

1.1.3 上报事件

上报事件是上行的交互动作，由设备发送给云端，例如：

```
{"method":"event.lock_broken","params":[1,"ss"]}
```

```
{
        "method": "event.lock_broken",
        "params": [
                1,
                "ss"
        ]
}
```

1.2 云端或手机下达的动作、命令

1.2.1 下达动作命令

动作、命令是下行的交互动作，由 MIIO client（云或手机）发送给设备上其他的进程，例如：

```
{"method":"set_volumn","params":[50]}
```

```
{
        "method": "set_volumn",
        "params": [50]
}
```

1.2.2 获得设备属性

get_prop，这个方法不用在每个设备的 profile 中定义，是系统默认就有的下行命令，用于获取设备某个或某些属性。

1.3 属性上报或者命令的回应

1.3.1 id 字段

如果某个请求 json 字符串（不管是上行还是下行）必须包含有 id 字段
如下所示：

```
{"method":"xxxx","param":"yyyy","id":123}
```

```
{
        "method": "xxxx",
       "param": "yyyy",
        "id": 123
}
```

id 字段（32 位无符号整型数）表示当前会话，必选。

- 回应/答复中需要附加相同 id 名值（发送者构造 id 时需要保证其唯一性）。
- id 尽量从某一个随机数开始，不要每次启动都从 0 开始。

1.3.2 回应具体格式

调用对方 method，若正确，得到可能回应如下：

1. {"result":["ok"],"id":123}
2. {"result":[1,2,3],"id":123}
3. {"result":[1,2,"tmp"],"id":123}
4. {"result":{"tmp_key":"tmp_value"},"id":123}

调用对方 method，若错误，得到回应如下：

```
{"error":{"code":"xxx","message":"xxxx"},"id":123}
```

```
{
        "error": {
                "code": "xxx",
                "message": "xxxx"
        },
        "id": 123
}
```

- 被请求者发还 result 字段，表示已经执行了被要求的 method；

- 被请求者发还 error 字段，表示未能正确执行 method 得到结果（无该服务、超时或其它之类的故障）；
    - code：错误编号，均为负数，小米智能家居内部错误有统一编号，对于厂商返回的错误编号目前限定为-10000。
    - message：错误具体字符串信息；
- error 与 result 互斥；

2. 设备接入 SPEC

设备接入 spec，即设备与小米云通讯的消息中属性、方法和事件符合 spec 协议规范。
MIoT-Spec 协议规范快速入门

2.1 profile 与 spec 中 method 对应关系

profile method	spec method
get_prop	get_properties
set_xxxx	set_properties
event.xxxx	event_occured
prop.xxxx	properties_changed
props	properties_changed
xxxx	action

其他的 json-rpc 命令，如 info、ota 保持不变。属性、方法、事件中的消息均使用 iid 字段，其中：

属性 ID 为 piid = <properties iid>

方法 ID 为 aiid = <actions iid>

事件 ID 为 eiid = <events iid>

siid = <services iid>

2.2 属性、事件上报

上报属性和事件是消息上行，消息由设备发给云端。

2.2.1 上报属性

请求：Device -> Server

```
{
        "id": 123,
```

```
        "method": "properties_changed",
        "params": [
                {
                        "did" : "1234",
                        "siid" : 1,
                        "piid" : 1,
                        "value" : 17
                }
        ]
}
```

成功应答：Server -> Device

```
{
        "id": 123,
        "result": [
                {
                        "did"  : "1234",
                        "siid" : 1,
                        "piid" : 1,
                        "code" : 0
                }
        ]
}
```

2.2.2 上报事件

请求：Device -> Server

```
{
        "id": 123,
        "method": "event_occured",
        "params":{
                "did": "1234",
                "siid": 1,
                "eiid": 1,
                "arguments": [
                        {
                                "piid": 1,
                                "value": 17
                        }
                ]
        }
}
```

成功应答：Server -> Device

```
{
        "id" : 123,
        "result" : {
                "code" : 0
        }
}
```

上报属性、事件失败应答：

```
{
        "id": 123,
        "error": {
                "code": -32600,
                "message": "Invalid Request"
        }
}
```

2.3 云端或手机下达动作、命令

2.3.1 读属性

请求：Server -> Device

```
{
        "id" :  123,
        "method" : "get_properties",
        "params" : [
                {
                        "did" : "1234",
                        "siid" : 1,
                        "piid" : 2
                },
                {
                        "did" : "1234",
                        "siid" : 1,
                        "piid" : 3
                }
        ]
}
```

应答：Device -> Server

```
{
        "id": 123,
        "result": [
                {
```

```
                "did"  : "1234",
                "siid" : 1,
                "piid" : 2,
                "value": 10,
                "code" : 0      // 读属性成功了
        },
        {
                "did" : "1234",
                "siid" : 1,
                "piid": 3,
                "code": -4001   //属性不可读,请根据 ERROR CODE 定义返回
        }
    ]
}
```

2.3.2 写属性

请求：Server -> Device

```
{
    "id": 123,
    "method" : "set_properties",
    "params" : [
        {
                "did"  : "1234",
                "siid" : 1,
                "piid" : 1,
                "value" : 10
        },
        {
                "did"  : "1234",
                "siid" : 1,
                "piid" : 88,
                "value" : "hello"
        }
    ]
}
```

应答：Device -> Server

```
{
    "id": 123,
    "result": [
        {
```

```
                "did"  : "1234",
                "siid" : 1,
                "piid" : 1,
                "code" : 0          // 成功
        },
        {
                "did"  : "1234",
                "siid" : 1,
                "piid" : 88,
                "code" : -4003    //属性不存在,请根据 ERROR CODE 定义返回
        }
    ]
}
```

2.3.3 执行方法

请求：Server -> Device

```
{
    "id": 123,
    "method" : "action",
    "params" : {
        "did" : "1234",
        "siid" : 1,
        "aiid" : 1,
        "in" : [
            {
                "piid" : 1,
                "value": 10
            }
        ]
    }
}
```

成功应答：Device -> Server

```
{
    "id" : 123,
    "result" : {
        "code" : 0,
        "out" : [
            {
                "piid" : 3,
                "value" : 10
            }
        ]
```

```
        }
}
```

失败应答：Device -> Server

```
{
        "id": 123,
        "result" : {
                "code" : -4004
        }
}
```

2.4 code 状态码定义

0	成功
1	接收到请求，但操作还没有完成
-4001	属性不可读
-4002	属性不可写
-4003	属性、方法、事件不存在
-4004	其他内部错误
-4005	属性 value 错误
-4006	方法 in 参数错误
-4007	did 错误

3. MQTT API

如果使用 MQTT API，MQTT 上目前有 6 个频道用于设备上其他进程与 MIIO 客户端通信：

- **miio/report**：频道用于设备（通过某进程）向 MIIO 客户端汇报信息与事件。
- **miio/report_ack**：云端对设备报告（属性、事件或请求）的回应。
- **miio/command**：频道用于小米云或者手机（当手机和设备在同一个局域网时）向设备下达动作命令，再由本地某进程去具体执行。
- **miio/command_ack**：设备对云端或者手机动作命令的回应。
- **local/miio_client_in**：通过这个频道可以利用 miio_client 提供的一些其他功能，比如查询时间等，格式详见第十五节。
- **local/miio_client_out**：miio_client 对请求的回应。

注：厂商使用 MQTT 方式与 miio_client 通信的时候，消息 id 尽量用随机数，因为服务器过滤机制对于同一个 id，一分钟之内只处理一次。

4. RAW socket API

如果使用 RAW socket API，由于 socket 是全双工，上行和下行都通过用户进程与 miio_client 建立的 socket 进行通信。

四，设备配置文件说明

MIIO 用到的配置文件有

- /etc/miio/device.conf
- /etc/miio/device.token
- /etc/miio/wifi.conf
- /etc/os-release

注：配置文件具体路径，厂商可以根据系统实际情况适配，与 helper 脚本使用保持一致即可。

1. /etc/miio/device.conf

设备配置文件，保存着设备独有的信息，例如：

```
# cat /etc/miio/device.conf
# did must be a unsigned int
# key must be a string
#
did=10000
key=EsrtdeaInabcPQM0
mac=8C:BE:BE:AA:bb:59
vendor=coolvendor
# model max len 23
model=coolvendor.prod.v1
```

其中的 did, key, mac 由小米智能家居项目组分配，请联系小米相关负责人。

2. /etc/miio/device.token

设备 token，每次设备重置（reset）的时候生成，主要用于设备跟用户绑定，快连，同一局域网下手机对设备直接控制等。第三方开发者不用关心这个文件。

3. /etc/miio/wifi.conf

设备快连完成之后，WiFi 用户名和密码存放文件。同时作为设备是否完成快连的标志文件存在。
第三方开发者注意：设备重置时，必须删除这个文件。更多细节请参考《AP 方式快连》一章。

4. /etc/os-release

/etc/os-release 是 Linux 标准的软件版本存放文件，我们用它来存放设备固件版本号。更多细节请参考《设备固件升级》一章

五，安装和移植

1，怎么安装

有两种方式安装。假设你拿到的是 miio_linux.le_glibc_armv7-a_cortex-a9_hardvfpv3.tar.gz，你可以解开安装包到一个临时文件夹然后一个一个拷贝你需要的文件到相应的目录：

```
$ mkdir miio_tmp
$ mv miio_linux.le_glibc_armv7-a_cortex-a9_hardvfpv3.tar.gz miio_tmp
$ cd miio_tmp
$ tar zxf miio_linux.le_glibc_armv7-a_cortex-a9_hardvfpv3.tar.gz
```

或者，使用下面的命令自动把所有的动态链接库和可执行文件安装到根目录相应的地方：

```
$ tar -C / -zxf miio_linux.le_glibc_armv7-a_cortex-a9_hardvfpv3.tar.gz
```

你可以随时用下面这个命令查看.tar.gz 里面包含了哪些文件：

```
$ tar -tvf miio_linux.le_glibc_armv7-a_cortex-a9_hardvfpv3.tar.gz
```

注意：如何您的系统中已经包含.tar.gz 中的一些库，就不必把所有库文件都覆盖，只需拷贝对应的库即可，另外可能需要创建一些对应的软链接，命令为：

```
ln -s {target-filename} {symbolic-filename}
```

2，怎么移植

在/etc/miio/目录下面有一个 device.conf 文件，包含每个设备唯一的{did, key, mac}等信息，你需要向小米申请这些信息才能使 miio_client 连上小米云。

在/etc/miio/目录下面有一个 wifi_start.sh，用于系统 wifi 的启动脚本，该脚本目的/功能是在系统启动的时候判断设备是否已经配置过本地路由器的{ssid, passwd}，如果没有，那么该脚本会让 wifi 进入 AP 快连模式；否则会启动 wifi 进入 STA 模式。每个系统都不一样，你需要修改这个脚本使它满足上面描述的功能。并且让它在系统启动的时候运行。

在 wifi_start.sh 里面，启动 AP 快连模式的时候，会设置 AP 的 ssid，这个 ssid 的格式有一定规则，否则手机 app 不能自动发现这个设备，则不能完成快连。ssid 名字规则如下：

假设你的设备 model 是：vendor.camera.v1，那么 ssid 的名字必须是：

```
vendor-camera-v1_miapAABB
```

其中 AABB 是设备 MAC 地址最后两个 bytes，比如 MAC 是 34:17:eb:9b:a7:47，那么 ssid 的名字是：

```
vendor-camera-v1_miapA747
```

同时你需要修改/usr/bin/miio_client_helper.sh，告诉它 wifi_start.sh 脚本的位置。

另外，你还需要修改 miio_client_helper.sh，以使 miio_client 支持时区配置。考虑到不同系统使用的文件系统格式不一样，对于只读文件系统，我们不能动态修改/etc/localtime(glibc)或者/etc/TZ(uclibc)，所以我们这里给出一个通用的方式。

首先您需要创建一个/etc/localtime 或/etc/TZ（根据系统使用的 libc 库选择）到一个实际可读写的位置(YOUR_LINK_TIMEZONE_FILE)的软链接，(这一步需要在制作根文件的时候完成)，然后根据实际使用的 libc 库选择时区配置的路径：

```
GLIBC_TIMEZONE_DIR="/usr/share/zoneinfo"
UCLIBC_TIMEZONE_DIR="/usr/share/zoneinfo/uclibc"
```

假如使用的为 uclibc 库，链接位置为/mnt/TZ，则脚本实际配置如下：

```
YOUR_LINK_TIMEZONE_FILE="/mnt/TZ"
YOUR_TIMEZONE_DIR=$UCLIBC_TIMEZONE_DIR
```

3，怎么运行

把下面这段脚本放入到你系统的启动脚本中：

```
/some/path/wifi_start.sh (or equivalent some other scripts)
/usr/bin/mosquitto -c /etc/mosquitto.conf -d （如果你使用 mqtt 版本的话）
/usr/bin/miio_client -D
/usr/bin/miio_client_helper.sh &
```

六，快连

为了方便用户将设备与 APP 进行绑定，MIIO SDK 提供了以下两种方式。

1. AP 方式快连

系统启动的时候，会调用 wifi_start.sh，这个脚本会检查设备是否已经配置好 wifi（通过检查文件 /etc/miio/wifi.conf）。然后根据 wifi 是否配置好分两条路径。

1. 如果已经配置好，wifi_start.sh 会直接驱动 wifi 进入 STA 模式并且连入网络。在 wifi_start.sh 之后启动的 miio_client 尝试连接小米云。
2. 如果 wifi 没有配置好，wifi_start.sh 会驱动 wifi 进入 AP 模式。在 wifi_start.sh 之后启动的 miio_client 暂时不会跟小米云连接，而是监听特定 UDP 端口（54321）的包。如果收到本地手机的配置请求，miio_client 先发送{did, token, timestamp} 建立信任关系，然后手机把路由器的{ssid，passwd}发给 miio_client，格式为：

```
{"id":xx,"method":"miIO.config_router","params":{"ssid":"xx","passwd":"xx","uid":xx,
"country_domain":"xx","tz":"Asia/Shanghai"}}
{
        "id": xx,
        "method": "miIO.config_router",
        "params": {
                "ssid": "xx",
                "passwd": "xx",
                "uid": xx,
                "country_domain": "xx",
                "tz": "Asia/Shanghai"
        }
}
```

其中，ssid、passwd、uid 为必选项，country_domain 和 tz 可以根据需要进行传递，如果设备在中国大陆，则 country_domain 可以忽略。

miio_client 收到{ssid,passwd}之后把他们写入 WiFi 配置文件/etc/miio/wifi.conf，调用 wifi_start.sh 切换 WiFi 到 STA 模式，开始尝试连接小米云。

2. 蓝牙方式快连

蓝牙方式快连与 AP 方式类似，不同的地方是步骤 2 时，蓝牙方式快连会监听 TCP 端口（54323）的包。

APP 下发的命令格式为：

```
{"id":xx,"method":"local.ble.config_router","params":{"ssid":"xx","passwd":"xx","uid":
xx,"country_domain":"xx","tz":"Asia/Shanghai"}}
{
        "id": xx,
        "method": "local.ble.config_router",
        "params": {
                "ssid": "xx",
                "passwd": "xx",
                "uid": xx,
```

```
                "country_domain": "xx",
                "tz": "Asia/Shanghai"
        }
}
```

3. 二维码方式快连

二维码方式快联与蓝牙方式类似，不同的地方是步骤 2 时，扫描二维码进程（厂商自己写）将解析到的数据，TCP 发送到 54322 端口。

二维码进程发出的命令格式为：

{ "id": xx ,"method": "local.ble.config_router", "params": { "bind_key": "xx", "ssid": "xx", "passwd": "xx", "tz": "Asia\/Shanghai", "country_domain":"" } }

```
{
        "id": xx,
        "method": "local.ble.config_router",
        "params": {
                "bind_key": "xx",
                "ssid": "xx",
                "passwd": "xx",
                "tz": "Asia\/Shanghai",
                "country_domain": ""
        }
}
```

4. 快连出错和重试

手机侧会验证用户输入的{ssid, passwd}，尽量保证没问题（可以正常连接路由器）；同时，设备端 miio_client 也会有出错重试次数（默认是 5 次，每次间隔 3s），重试完如果还是不成功，则回到 AP 快连状态。手机/用户可以重新开始快连。

七，路由器密码修改

智能设备通过链接路由器实现联网功能。如果路由器 ssid 或者密码更改，设备则连不上路由器，需要重置并且重新快连。

如果设备连接的路由器是小米路由器，我们有一个增强用户体验的功能是让小米路由器把新的 ssid 和密码发送给设备，设备则不需要重置和重新快连的过程。

设备要支持这个功能，有两个条件：

1. 如上所说，设备连接的必须是小米路由器；
2. 设备 Linux 系统的 hostname（DHCP hostname）必须符合以下规范：

   DHCP NAME: [model]_miio[数字 DID]

   例如：

   xiaomi-dev-v1_miio11245

注：路由器下发配网命令的时候没有 uid，请在 miio_client_helper.sh 脚本里面将 uid 的更新增加有效值判断后更新。

八，设备联网状态广播

设备启动，快连，联网的时候，会广播一些状态。这些状态信息会广播出来（如果是 mqtt 的接口，通过 miio/command 频道发出来；如果是 socket 接口，通过 socket 发出来）。

1, 设备启动的时候，如果发现没有配置过路由器 ssid 和密码，设备 WiFi 进入 AP 模式，并且广播：
`{'method': 'local.status', 'params': 'wifi_ap_mode'}`

2, 设备收到手机发过来的路由器 ssid 和密码之后，准备连接路由器，广播：
`{'method': 'local.status', 'params': 'wifi_connecting'}`

WiFi 连接成功，广播：
{'method': 'local.status', 'params': 'wifi_connected'}

WiFi 连接失败（比如距离过远），广播：
{'method': 'local.status', 'params': 'wifi_failed'}
（同时，手机会显示快连失败，并且提示用户可能原因）

3, 设备连上小米云之后，广播：
{'method': 'local.status', 'params': 'cloud_connected'}

如果连不上，重试，广播：
{'method': 'local.status', 'params': 'cloud_retry'}

九、设备离线状态通知

1, 当本地网络不通时, miio_client 间隔 60 分钟发送_internal.wifi_reload 消息给 helper 脚本;
当 helper 脚本收到该消息, 建议重新加载 wifi 驱动并重连 wifi(一定要在后台运行, 谨慎评估重新加载 wifi 驱动风险)

2，当本地网络不通时，miio_client 间隔 15 分钟发送_internal.wifi_reconnect 消息给 helper 脚本（如果已发送_internal.wifi_reload， 该消息不发送).

当 helper 脚本收到该消息，建议启动 wifi 重连(一定要在后台运行);

十，设备固件/软件升级

设备固件升级用到的命令有下面几个：

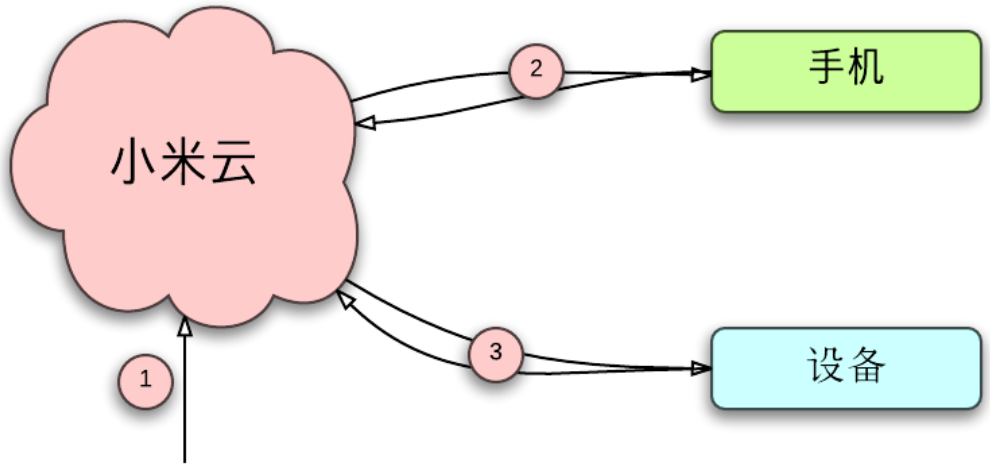
云端下发：

- miIO.ota
- miIO.get_ota_state
- miIO.get_ota_progress

设备属性上报：

- {"method":"props","params":{"ota_state":"xxxx"},"id":123}
- {"method":"props","params":{"ota_progress":xx},"id":123}

步骤如下图所示：



步骤说明：

1. 用户上传固件到小米 IoT 开发者平台；
2. 根据后台对该产品设置的升级模式来选择处理。如果是普通模式，则通知手机有新版本可更新，用户点击升级；如果是静默下载，则后台会分批次自动发送下载命令到设备。
3. ota 命令，带 url 参数；手机通过服务器不停轮询设备 ota_progress；
4. 设备升级完成（重启）后，通过 ota_state 属性上报 idle 状态；

升级状态说明：

idle：空闲态，当升级完成上报该状态给后台

downloading：固件正在下载

downloaded：固件下载完成

wait_install：等待安装。仅用于静默下载模式，下载完成并等待用户安装指令时发送该状态

installing：固件正在安装

installed：固件已经安装成功，一般会重启。

failed：升级失败。失败的消息可以带错误码和错误消息上报给后台，见 下面章节 升级失败的状态上报

busy：设备正在忙。

1. 升级框架

所有接入小米智能家居的设备，软件升级都必须通过小米 IoT 开发者平台（https://iot.mi.com/index.html）系统来做，大致的步骤如下：

1. 合作产商把固件或者新的软件包通过小米 IoT 开发者平台管理页面上传到服务器，产生 URL；
2. 服务器后台通知用户手机，指出他绑定的设备有新的版本；
3. 用户点击同意升级；事件先发送到云后台；
4. 云后台下发升级命令到设备，同时下发的还有新固件或软件包的 URL 以及 md5；
5. 设备收到升级命令和 URL，下载升级固件或者软件包，校验没有错误，执行升级；升级过程中，手机（或者云后台）可能会不停的询问升级进度等信息；
6. 升级完成（并重启）后，通过 ota_state 属性上报 idle 状态。

2. 版本号规则

版本号规则是这样：x.x.x_aaaa, 其中下划线前面的是小米智能家居负责的一些版本号，后面的是其他产商的版本号。

小米智能家居的版本号由小米决定，比如 2.1.1，其他产商的的版本号产商自己决定，但有几点：

1. 不像小米的版本号可以有小数点，产商的版本号现在只能是一个数字，中间不能有点，也不能有下划线等隔开；
2. 必须保证新的版本号比前一个版本号大，即线性增加；
3. 只有 4 个数字；

例子：2.1.1_9527

3. 软件版本号获取

软件版本号建议放在“**/etc/os-release**”中，“key=value”的格式。其中 key 为${VENDOR}_VERSION。

${VENDOR}来自与/etc/miio/device.conf 里面的”vendor“字段，然后全部转成大写。比如：

/etc/miio/device.conf 中，有

```
vendor=xiaomi
```

那么/etc/os-release 中的软件版本号表示为：

```
XIAOMI_VERSION=2.1.1_9527
```

/etc/os-release 是标准的 Linux 系统发布版本号存放文件。比如 Ubuntu14.04 中/etc/os-release 为：

```
$ cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.2 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.2 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

4. 详细命令解析

4.0 名词解释

静默下载：

下载和安装分离，云端推送下载命令，待用户需要升级时，下发安装命令，减少用户等待下载的时间。

静默升级：

升级的过程中，设备不会发出声音或其他动作，以免干扰用户。

二者是两个需求，可以同时存在，也可以单独启用。 后台服务器需要配置设备支持那种模式。

4.1 升级开始

云后台下发升级命令给设备，参数包括新固件或者软件包的 URL，比如：

1. 下载和安装

```
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"dnld
install","mode":"xxxx"}}
```

```
{
        method: "miIO.ota",
        param: {
                app_url: "xxxx_url",
                file_md5: "xxxxxxx",
                proc: "dnld install",
                mode: "xxxx"
        }
}
```

2.下载

只下载，不安装

```
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"dnld"}
}
```

```
{
        method: "miIO.ota",
        param: {
                app_url: "xxxx_url",
                file_md5: "xxxxxxx",
                proc: "dnld"
        }
}
```

3. 安装

安装之前下载的固件，该 md5 与 dnld 下载的文件 md5 一致。

```
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"instal
l","mode":"xxxx"}}
```

```
{
        method: "miIO.ota",
        param: {
                app_url: "xxxx_url",
                file_md5: "xxxxxxx",
                proc: "install",
                mode: "xxxx"
```

```
        }
}
```

参数说明：

- app_url: 新固件地址。
- file_md5: 固件 md5。
- proc：表示固件处理方法，dnld：仅下载，install：仅安装，dnld install：同时下载和安装。如果没有该字段，则固件按照“dnld install”处理。
- mode：字段表示 ota 模式，silent 或者 normal，前者表示静默下载，后者表示正常下载，如果没有该字段，则固件按照 normal 处理。

4. 场景

场景 1、设备支持下载安装和分离

Step1:
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"dnld","mode":"xxxx"}}

仅下载，不安装。

Step2:
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"install","mode":"xxxx"}}

安装之前下载的固件，app_url 必选，原因见下。

场景 2、设备只支持下载同时安装

Step1:
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"dnld","mode":"xxxx"}}

设备不支持，返回错误。

Step2:
{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"install","mode":"xxxx"}}

Or

{"method":"miIO.ota","param":{"app_url":"xxxx_url","file_md5":"xxxxxxx","proc":"dnld install","mode":"xxxx"}}

设备下载固件，并安装，可见对于不支持下载安装分离的设备，安装指令也需要带上 url 这些关键信息。如此，该接口便可以同时兼容两种不同的场景

5. 应答

对这条命令的应答是：

```
{"id":123,"result":["OK"]}
{
        "id": 123,
        "result": ["OK"]
}
```

4.2 查询升级状态

```
{"id":123,"method":"miIO.get_ota_state","params":{}}
{
        "id": 123,
        "method": "miIO.get_ota_state",
        "params": { }
}
```

获取升级状态，可能的应答包括 idle，downloading，dowloaded，installing，wait_install, installed，failed，busy

比如：

```
{"id":123,"result":["downloading"]}
{
        "id": 123,
        "result": ["downloading"]
}
```

4.3 上报升级状态

根据当前的状态，可能上报的状态为 idle，downloading，downloaded，installing，wait_install，installed，failed，busy。

静默下载的过程中，当设备固件下载完成、等待用户发送安装命令的过程中，会上报 wait_install 命令。

4.3.1 升级完成后状态上报

系统升级完之后，应上报 ota_state 属性，值为“idle”，后台根据这个状态，结合_otc.info 上报的版本为新版本，得知升级成功完成，并通知手机端。

在具体的实现中，每次系统重启之后，miio_client 都会发送这个属性。

```
{"id":123,"method":"props","params":{"ota_state":"idle"}}
{
        "id": 123,
        "method": "props",
        "params": {
                "ota_state": "idle"
        }
}
```

4.3.2 升级失败的状态上报

1.上报格式：

{“method”:”props”,”params”:{“ota_state”:“failed|error_code|error msg”},”id”:123}

failed,error_code,error_msg 采用管道符“|”分割，其中 error code 是负整数，范围是-33000~-33100，error msg：错误消息,可选

2.error code 和 error msg 定义

所有设备采用统一的 error code 和 error msg，不能随意修改。如果需要增加新的 error code 和 error msg，必须遵循上述格式

error code	error msg	说明
-33001	down error	下载失败，设备侧无法提供失败的原因，一般是网络连接失败
-33002	dns error	dns 解析失败
-33003	connect error	连接下载服务器失败

-33004	disconnect	下载过程中连接中断
-33005	install error	安装错误，下载已经完成，但是安装的时候报错
-33006	cancel	设备侧取消下载
-33007	low energy	电量低，终止下载
-33020	unknown	未知原因

4.4 上报升级进度

```
{"id":123,"method":"props", "params": {"ota_progress":80}}
{
        "id": 123,
        "method": "props",
        "params": {
                "ota_progress": 80
        }
}
```

4.5 查询升级进度

```
{"id":123,"method":"miIO.get_ota_progress","params":{}}
{
        "id": 123,
        "method": "miIO.get_ota_progress",
        "params": { }
}
```

获取升级进度，可能的应答包括进度 0 - 100。比如：

```
{"id":123,"result":[99]}
{
        "id": 123,
        "result": [99]
}
```

5. 每次升级多个文件

有些设备每次升级包含多个文件，比如 Linux 的设备可能同时需要升级 kernel，rootfs，这时候产商需要把这些升级文件整合在一个文件里（比如放在一个 zip 包里），其他的流程跟上面说的一样。

十一，设备重置

建议设备重置的执行顺序如下：

1. 接到用户（或者云端）重置命令；
2. 删除/etc/miio/wifi.conf；
3. WiFi 进入 AP 模式；
4. Kill miio_client 和 miio_client_helper.sh, 重启这两个程序；
5. miio_client 启动的时候检测到没有 wifi.conf，会删除 device.token（意即设备跟用户解绑）；
6. miio_client 发送状态广播：{'method': 'local.status', 'params': 'wifi_ap_mode'}；
7. 手机可以重新开始快连。

用户重置命令比如物理按键组合；某个物理按键按住 3s 以上等。
云端重置命令指的是从云端下发的 reset 动作（method）命令。

十二，设备动态申请 did 和 key

大部分的设备，did 和 key 都是预先分配的，存储在/etc/miio/device.conf 里面；但是也有部分特殊设备，它们的 did 和 key 需要设备第一次上电之后从云端获取。这一章描述 miio 客户端从服务器获取 did 和 key 的大致流程，以及用户可见的一些接口。

（注：此机制只对特定的品类有效。）

1. 触发

/etc/miio/device.conf 中的 did 如果是 0 或者 1 的话，会触发 miio 客户端向服务器申请 did 和 key。

- did=0：忽略快连过程，设备已经连上网络，直接向服务器申请 did 和 key；
- did=1：需要快连过程，连上网络之后，向服务器申请 did 和 key；

2. 大致流程

did=0 的情形，miio 客户端向服务器提交申请，服务器检查申请是否合理，如果合理会把申请放入队列，统一在当天的某一时候批处理。miio 客户端第二天再向服务器提交申请，服务器返回 did 和 key。

did=1 的情形，miio 客户端快连完成之后，向服务器提交申请，服务器检查申请是否合理，如果合理立即生成并下发 did 和 key。

十三，设备 suspend/resume 接口

假定设备系统中有一个进程或服务行驶 power manager 相关的功能，负责协调系统中所有进程的睡眠和唤醒请求，它是系统进入睡眠的信息汇总和决策者。它跟 miio 客户端的通信如下：

（图）

power_manager 向 miio_client 发送 **prep_suspend** 询问请求。该请求带有一个参数 id, 用来表征这次请求的唯一性。

miio_client 应立刻向 power_manager 回复 **resp_suspend**，回复会带着两个参数：id 和 wakeup_time（相对时间）。id 即是 prep_suspend 命令中送来的请求 id；wakeup_time 为 miio_client 向 power_manager 请求的下次唤醒时间。

wakeup_time 分两种情况：

- 如果 miio_client 当前有需要处理的事情，不能进入睡眠，将 wakeup_time 设置为 0。power_manger 收到这个回包，会在一段时间以后重新询问；
- 如果 miio_client 同意进入休眠，将 wakeup_time 设置为下次需要被唤醒的时间（相对时间，比如 15，单位秒）。power_manager 收到这个回包，会根据 miio_client 请求的 wakeup_time 在指定的时间点唤醒系统。

具体命令格式：

```
{"id":9527,"method":"local.prep_suspend","params":[]}
{
        "id": 9527,
        "method": "local.prep_suspend",
        "params": [ ]
}

{"id":9527,"method":"local.resp_suspend","params":{"wakeup_time":10}}
{
        "id": 9527,
        "method": "local.resp_suspend",
        "params": {
                "wakeup_time": 10
        }
}
```

十四，设备需要实现的其他接口

1. miIO.reboot

设备上的应用层可以实现 reboot 这个 RPC 命令，当远程有调用的时候，保存好必要的信息，然后重启。

2. miIO.restore

设备上的应用层应该实现 restore 这个 RPC 命令。当用户在手机 app 上解绑了一个设备，这个命令就会下发到设备端。设备端应该当作 reset 处理：需要删除 wifi.conf（清除 wifi 密码），device.token，删除绑定关系。

十五，设备与小米云通讯（经由 miio 客户端）典型流程

1. 设备连上小米云；
2. 设备通过特定数据包与小米云同步时间；
3. 设备通过_otc.info 方法将设备环境上报给小米云；
   （以上步骤都由 miio_client 负责完成，设备端其他进程不必关心细节）
4. 设备在检测到 profile 中定义的属性或者事件发生改变的时候，通过 event.xxx 或者 prop.xxx 将改变上报到小米云（xxx 必须是设备 profile 中定义的属性或者事件）
5. 小米云/手机通过 xxx 方法让设备执行某些动作（xxx 是 profile 中定义的设备可以执行的方法）；或者，小米云/手机通过 get_prop/set_prop 从设备获取属性值或者设定设备的属性值。

十六，MIIO Client 提供的其他功能接口

为了方便用户操作，miio_client 提供了一些常用的功能接口，这些接口分为两大类，一类是给用户 socket 客户端提供的查询接口，比如查询网络连接状态、查询本地时间等；另一类是给云端或者移动客户端的配置接口，如配置时区信息、配置 log 级别等。
如果当前这些功能不能满足您的需求，请联系小米工程师协助添加。

这些接口以”method”字段进行区分，统一调用格式为：

```
{"id":12345,"method":"xxx"}
```

```
{
        "id": 12345,
```

```
        "method": "xxx"
}
```


如查询本地时间，格式为：


```
{"id":12345,"method":"local.query_time"}
{
    "id": 12345,
    "method": "local.query_time"
}
```


其中，id 由用户自定义，用于区分不同消息的 index。

一、查询接口：

1、查询本地时间：

method：local.query_time

miio_client 回复消息格式为：


```
{"id":12345,"method":"local.time","params":1488524370}
{
        "id": 12345,
        "method": "local.time",
        "params": 1488524370
}
```


2、查询国别域名：

method：local.query_country

miio_client 回复消息格式为：

存在域名：


```
{"id":12345,"method":"local.country","params":"sg"}
{
        "id": 12345,
        "method": "local.country",
        "params": "sg"
}
```


不存在域名：


```
{"id":12345,"method":"local.country","params":"prc"}
{
        "id": 12345,
        "method": "local.country",
        "params": "prc"
}
```


3、查询 miio_client 状态：

method： local.query_status

miio_client 回复消息格式为：

```
{"id":12345,"method":"local.status","params":"cloud_connected"}
{
        "id": 12345,
        "method": "local.status",
        "params": "cloud_connected"
}
```

其中 params 可能为以下几个值：

```
{
      "device_init",          /* STATE_DEVICE_INIT */
      "didkey_req1",          /* STATE_DIDKEY_REQ1 */
      "didkey_req2",          /* STATE_DIDKEY_REQ2 */
      "didkey_done",          /* STATE_DIDKEY_DONE */
      "token_done",           /* STATE_TOKEN_DONE */
      "ap_mode",              /* STATE_WIFI_AP_MODE */
      "sta_mode",             /* STATE_WIFI_STA_MODE */
      "cloud_trying",         /* STATE_CLOUD_TRYING */
      "cloud_connected",      /* STATE_CLOUD_CONNECTED */
      "cloud_retry"           /* STATE_CLOUD_CONNECT_RETRY */
}
```

二、配置接口

1、配置时区信息

method：miIO.config_tz

```
{"id":12345,"method":"miIO.config_tz","params":{"tz":"Asia/Shanghai"}}
{
        "id": 12345,
        "method": "miIO.config_tz",
        "params": {
                "tz": "Asia/Shanghai"
        }
}
```

配置时区信息要对应修改 helper 脚本，详见第五章怎么移植

2、配置 log 打印级别

method：miIO.config_loglevel

```
{"id":12345,"method":"miIO.config_loglevel","params":{"loglevel":2}}
{
        "id": 12345,
        "method": "miIO.config_loglevel",
        "params": {
                "loglevel": 2
        }
}
```

其中 loglevel 的值为 0~4，分别代表：

```
LOG_ERROR = 0,
LOG_WARNING = 1
LOG_INFO = 2
LOG_DEBUG = 3
LOG_VERBOSE = 4
```

成功返回：

```
"{"id":12345,"resul":["OK"]}"
```

失败返回：

```
"{"id":12345,"resul":["ERROR"]}"
```

十七，FAQ

1. 如何观察 MQTT 上发送和接收的消息？

```
$ mosquitto_sub -h localhost -t \# -v
```

2. 开放平台下载 SDK 无法运行？

应该是 SDK 与厂商使用的 toolchain 不一致，请联系小米使用厂商 toolchain 编译 SDK.

3.Helper 脚本怎么编写？

小米提供 SDK helper 脚本和 wifi start 脚本示例，阅读 helper 脚本使用说明文档，厂商需要根据系统实际情况适配。

4.怎么通过 log 确认 SDK 可以正常工作？

执行 miio_client -v 查看说明文档，保存 log 信息。从 log 中看到 STATE_CLOUD_CONNECTED 表明 SDK 工作正常。

5.配网过程是否支持中文 SSID？

miio client SDK 不会关注 ssid 是否为中文编码，需要米家 APP 和路由器都支持同一种中文编码格式。

6.产品固件对于不支持的 RPC 命令该如何处理？

对来自 miio_client 的 RPC 命令，如果产品固件不支持，需要返回{"id":xxx,"error":{"code":-10000,"message":"user undefined error"}}消息，xxx 为 RPC 的 id。由于有多进程接收 miio_client 的 RPC，产品固件不易支持，故从 4.0.5 版本开始，对产品固件不作此要求，会在 Linux SDK 固件中修复。

十八，附录：

1. 一个较完整的_otc.info 例子：

{"id":2,"method":"_otc.info","params":{"model":"vendor.camera.v1","mac":"AA:BB:CC:01:02:03","token":"71575a337068697372747739735a3938","life":91843,"hw_ver":"unknown","fw_ver":"unknown","ap":{"ssid":"2zy

anlu","bssid":"10:48:b1:c9:bf:92"},"netif":{"localIp":"192.168.1.157","mask":"255.255.255.0","gw":"192.168.1.1"}}}


```
{
        "id": 2,
        "method": "_otc.info",
        "params": {
                "model": "vendor.camera.v1",
                "mac": "AA:BB:CC:01:02:03",
                "token": "71575a337068697372747739735a3938",
                "life": 91843,
                "hw_ver": "unknown",
                "fw_ver": "unknown",
                "ap": {
                        "ssid": "myssid",
                        "bssid": "10:20:b1:c9:bf:92"
                },
                "netif": {
                        "localIp": "192.168.1.157",
                        "mask": "255.255.255.0",
                        "gw": "192.168.1.1"
                }
        }
}
```


2. mosquitto 编程例子

[libmosquitto API documentation]

例子 1：

```
#include <stdio.h>
#include <mosquitto.h>

void my_message_callback(struct mosquitto *mosq, void *userdata, const struct mosquitto_message *message)
{
	if(message->payloadlen){
		printf("%s %s\n", message->topic, message->payload);
	}else{
		printf("%s (null)\n", message->topic);
	}
	fflush(stdout);
}

void my_connect_callback(struct mosquitto *mosq, void *userdata, int result)
{
```

```
	int i;
	if(!result){
		/* Subscribe to broker information topics on successful connect. */
		mosquitto_subscribe(mosq, NULL, "$SYS/#", 2);
	}else{
		fprintf(stderr, "Connect failed\n");
	}
}

void my_subscribe_callback(struct mosquitto *mosq, void *userdata, int mid, int qos_count,
const int *granted_qos)
{
	int i;

	printf("Subscribed (mid: %d): %d", mid, granted_qos[0]);
	for(i=1; i<qos_count; i++){
		printf(", %d", granted_qos[i]);
	}
	printf("\n");
}

void my_log_callback(struct mosquitto *mosq, void *userdata, int level, const char *str)
{
	/* Pring all log messages regardless of level. */
	printf("%s\n", str);
}

int main(int argc, char *argv[])
{
	char id[30];
	int i;
	char *host = "localhost";
	int port = 1883;
	int keepalive = 60;
	bool clean_session = true;
	struct mosquitto *mosq = NULL;

	mosquitto_lib_init();
	mosq = mosquitto_new(id, clean_session, NULL);
	if(!mosq){
		fprintf(stderr, "Error: Out of memory.\n");
		return 1;
	}
	mosquitto_log_callback_set(mosq, my_log_callback);

	mosquitto_connect_callback_set(mosq, my_connect_callback);
	mosquitto_message_callback_set(mosq, my_message_callback);
	mosquitto_subscribe_callback_set(mosq, my_subscribe_callback);

	if(mosquitto_connect(mosq, host, port, keepalive)){
		fprintf(stderr, "Unable to connect.\n");
		return 1;
	}

	while(!mosquitto_loop(mosq, -1, 1)){
```

```
	}
	mosquitto_destroy(mosq);
	mosquitto_lib_cleanup();
	return 0;
}
```

例子 2：

如果你的主程序已经有自己的 event loop，你需要使用的是 mosquitto_loop_misc(), mosquitto_loop_read/write() 来把 mosquitto 相应的处理函数嵌入到你的事件循环中。比如：

```
...
	n = 0;
	while (n >= 0 && !miio.force_exit) {
		int i;

		n = poll(miio.pollfds, miio.count_pollfds, POLL_TIMEOUT);
		if (n < 0) {
			perror("poll");
			continue;
		}
		if (n == 0) {
			/* printf("poll timeout\n"); */
			mosquitto_loop_misc(miio.mosq);

			continue;
		}

		for (i = 0; i < miio.count_pollfds && n > 0; i++) {
			if (miio.pollfds[i].revents & POLLIN) {
				if (miio.pollfds[i].fd == mosq_sock)
					mosquitto_loop_read(miio.mosq, 1);

				n--;
			} else if (miio.pollfds[i].revents & POLLOUT) {
				if (miio.pollfds[i].fd == mosq_sock)
					mosquitto_loop_write(miio.mosq, 1);

				n--;
			}
		}
	}
...
```

3．设备电子说明书示例

设备型号(model)

xiaomi.demo.v1

属性

编号	名称	类型	范围	解释
1	rgb	int	0x00000000~0x00FFFFFF	颜色【可读可写】
2	temperature	int	0~50	摄氏温度【可读不可写】
3	humidity	int	20~90	湿度【可读不可写】

例如：
温度和湿度会每 5 秒钟会上报一次，上报格式如下：

```
{"method":"props","params":{"temperature":25,"humidity":35}}
```

```
{
        "method": "props",
        "params": {
                "temperature": 25,
                "humidity": 35
        }
}
```

云端返回：

```
{"result":["ok"]}
```

```
{
        "result": ["ok"]
}
```

或

```
{"result":["error"]}
```

```
{
        "result": ["error"]
}
```

事件

编号	名称	解释	参数个数	参数说明
1	button_pressed	按钮按下事件	0	--
2	button_long_pressed	按钮长按事件	1	second(int): 被按下的时间

例如：
当按钮被按下时，会上报事件，上报格式如下：

```
{"method":"event.button_pressed","params":[]}
{
        "method": "event.button_pressed",
        "params": [ ]
}
```

云端返回：

```
{"result":["ok"]}
{
        "result": ["ok"]
}
```

或

```
{"result":["error"]}
{
        "result": ["error"]
}
```

当按钮被长按时，会上报事件，上报格式如下：

```
{"method":"event.button_long_pressed","params":[5]}
{
        "method": "event.button_long_pressed",
        "params": [ 5 ]
}
```

云端返回：

```
{"result":["ok"]}
{
        "result": ["ok"]
}
```

或

```
{"result":["error"]}
{
        "result": ["error"]
}
```

方法

编号	名称	解释	参数个数	参数说明
1	get_prop	获取属性值的通用命令	1~N	参数是一个数组，包含 1～N 个要获取的属性名称（字符串），比如： [“rgb”,“temperature”,“humidity”]
2	set_rgb	设置灯的颜色	1	rgb(int): 0xaabbcc
3	get_rgb	获取灯的颜色	0	--
4	get_temperature	获取温度	0	--
5	get_humidity	获取湿度	0	--
6	move	机器人移动	2	direction (string): 方向 distance (int): 距离，单位 cm {“direction”:“north”,“distance”:5}

例如：

云端下发：

```
{"method":"set_rgb","params":[250]}
```

```
{
        "method": "set_rgb",
        "params": [250]
}
```

设备回复：

```
{"result":["ok"]}
```

```
{
        "result": ["ok"]
}
```

或

```
{"error":{"code":-5001,"message":"set failed"}}
```

```
{
        "error": {
                "code": -5001,
                "message": "set failed"
        }
}
```

云端下发：

```
{"method":"get_rgb","params":[]}
```

```
{
        "method": "get_rgb",
        "params": [ ]
}
```

设备回复：


```
{"result":[255]}
```

```
{
        "result": [255]
}
```


或


```
{"error":{"code":-5001,"message":"read failed"}}
```

```
{
        "error": {
                "code": -5001,
                "message": "read failed"
        }
}
```