智物联工业网关（适配器 APRUS）

软件编程手册

V1.1

深圳市智物联网络有限公司

Mixlinker Networks(Shenzhen) Inc.

目　录

引言：适配器与 Lua 脚本

APRUS 适配器的基本行为受脚本控制，适配器的数据采集、边缘计算、上、下行消息等行为都可以根据用户需求在适配器的脚本中进行订制。APRUS 适配器的脚本采用 Lua 语言编写。Lua 是一种小巧、高效、易学习和易使用的脚本语言。

APRUS 适配器的脚本分为两个部分，一个是程序脚本 aprus.lua，另一个是配置脚本 config.lua。程序脚本 aprus.lua 控制适配器使用什么协议采集数据、如何运算、采集和计算的结果如何投递到下一级设备。配置脚本 config.lua 则控制采集的具体参数，比如采集的具体点位、数量、上报名称等等。将配置好的 aprus.lua 和 config.lua 上传到 APRUS，APRUS 就会根据配置的要求进行采集数据。不同的通讯协议，Lua 的配置也有一些小的区别。对于我们已支持的接口和协议，用户只需要根据自己的实际情况在已有的 demo 配置文件上稍加改动即可完成采集功能。不支持但通用的协议可定制开发。

APRUS 主要行为分为两部分，数据采集（CollrReg）和数据上报（CollpReg）。为了提升采集效率，数据采集和数据上报工作在不同的线程：数据采集过程中尽可能的把能连续采集的一次采集回来；数据上报是将已经采集的数据进行必要的数据处理，然后上报到云端。

一、内置的通用 Lua API

1.1. user API

1.1.1. user.setluaver

功能	设置 lua 版本信息
接口描述	setluaver(luaver)
luaver	lua 版本信息<字符串> 例:"LUA_V1.0.0"

1.1.2. user.setdevinfo

功能	设置 lua 所采集设备的信息
接口描述	setdevinfo(devinfo)
devinfo	所采集设备的信息<字符串>例:"Siemens_S7-1200"

1.1.3. user.ipconfig

功能	设置数据上报网口的信息
接口描述	ipconfig(ipmode, inet_addr, netmask,gw, dns)
ipmode	ip 获取方式<字符串> ["none" / "manual"/ "auto"]
inet_addr	ip 地址<字符串> 例:"192.168.1.100"
netmask	子网掩码<字符串> 例:"255.255.255.0"
gw	网关<字符串> 例:"192.168.1.1"
dns	域名服务器<字符串> 例:"114.114.114.114"

1.1.4. user.stop_gardsroute

功能	禁止通过路由模式自动连接服务器，当需指定 mqtt server 时使用
接口描述	stop_gardsroute()

1.1.5. user.waitmsg

功能	消息获取接口，阻塞等待所有事件
接口描述	msg = user.waitmsg()
msg	msg.from<字符串> 来自哪个模块的消息 包括: "mqtt-sys":mqtt 系统消息 "mqtt-msg":mqtt 数据消息 "s7": s7 模块的消息 "modbus":modbus 模块的消息 "mitsufx":mitsufx 模块的消息 "mitsumc":mitsumc 模块的消息 "opcua":opcua 模块的消息

	"devdata":设备属性的消息 "adio":开关模拟量的消息 "dlt645":dlt645 模块的消息 "transfer":透传模块的消息 msg.session 消息的来自哪个 obj msg.code 消息编码

1.1.6. user.setPDcallback

功能	设置掉电事件的 lua 回调函数名
接口描述	setPDcallback(pFunName)
pFunName	回调函数的名称<字符串>

1.1.7. user.ledctrl

功能	设置 led 灯状态
接口描述	ledctrl(led, status, arg)
led	led<字符串>["green" / "red"/ "yellow"/"blue"/"white"/"orange"]
status	状态<字符串>["on" / "off"/ "blink"]
arg	闪烁参数(当 status 为 blink 时有效 代表闪烁次数)<整形>

1.1.8. user.configReport

功能	设置默认上报配置
接口描述	configReport(flag_ReportLua,cycle_ReportLua, flag_ReportCSQ, cycle_ReportCSQ, flag_ReportGPS, cycle_ReportGPS)
flag_ReportLua	是否上报 lua 信息[-1（仅上报一次）/0（不上报）/1（周期上报）]
cycle_ReportLua	lua 信息上报周期
flag_ReportCSQ	是否上报 4G/wifi 信号强度[-1（仅上报一次）/0（不上报）/1（周期上报）]
cycle_ReportCSQ	4G/wifi 信号强度上报周期
flag_ReportGPS	是否上报 GPS 信息[-1（仅上报一次）/0（不上报）/1（周期上报）]
cycle_ReportGPS	GPS 信息上报周期

1.1.9. user.addtimer

功能	设置定时器(一个 Lua 程序中最多可存在 10 个定时器)；详见 APRUS Lua 计时器和定时器配置说明
接口描述	user.addtimer("timerName", Count, Cycle)
"timerName"	定时器的名称为"timerName"（最长 19 字符）
Count	总共需要执行的定时器次数。如果 Count 为-1，则表示定时器将无限次执行。
Cycle	定时器执行的间隔时间，以秒为单位。每隔 Cycle 秒，定时器就会执

	行一次回调函数。

1.1.10. user.closetimer

功能	停止定时器
接口描述	user.closetimer("timerName")
"timerName"	定时器的名称为 timerName

1.1.11. user.newClockTimer

功能	user.newTimer 返回一个计时器实例对象；未复位前，计时器的值会从当前时间开始一直增加下去；一个 LUA 程序可以有多个计时器；可用于延时函数；详见 APRUS Lua 计时器和定时器配置说明
接口描述	myTimer = user.newClockTimer()
myTimer	user.newClockTimer 返回值；计时器实例对象

1.1.12. user.resetClockTimer

功能	复位计时器
接口描述	user.resetClockTimer(myTimer)
myTimer	计时器实例对象；

1.1.13. user.getClockTimer

功能	获取计时器当前计时值
接口描述	num = user.getClockTimer(myTimer)
myTimer	计时器实例对象；
num	计时器当前计时值，单位 ms

1.1.14. user.saveVal

功能	保存数据到文件中，以 json 格式保存；
接口描述	user.saveVal("/data/val_1","Total_Quantity", Total_Quantity)
/data/val_1	字符串，将数据保存到/data/val_1 文件中；
Total_Quantity	字符串，具体的 json 中的 key 值索引；
Total_Quantity	值，具体的 json 中的 val 值；

1.1.15. user.getVal

功能	获取文件中的数据，通过 json 中的 key 索引；
接口描述	Total_Quantity = user.getVal("/data/val_1", "Total_Quantity")
/data/val_1	字符串，从/data/val_1 文件中获取数据；

Total_Quantity	字符串，具体的 json 中的 key 值索引；
Total_Quantity	返回值，具体的 json 中的 val 值；

1.1.16. user.getDev

功能	获取设备数据；若想要获取的参数在下述结构体中不存在，则返回 nil 值
接口描述	net_ver = user.getDev("net", "ver")
net	字符串，参数类型；可选 manage、net、app
ver	字符串，具体的参数；参考如下结构体；

```
struct
{
    char softver[20];        /*软件版本*/
    char devid[20];          /*适配器ID*/
    char devmodel[20];       /*设备模式*/
    char devhwver[12];       /*设备硬件版本*/
    char managever[20];      /*管理程序版本*/
    char sysver[16];         /*文件系统版本*/
    char nettype;            /*网络类型*/
    char isconfig;           /*配置状态*/
    char mmqtt_status;       /*mqtt状态*/
    long long int sendbyte; /*发生数据流量，单位byte*/
    long long int sendn;     /*发生数据次数*/
    int connectn;            /*链接次数*/
    int connecterrn;         /*链接失败次数*/

    struct
    {
        int status;  /*gps状态*/
        double lat;  /*gps参数*/
        double lon;  /*gps参数*/
        double high; /*gps参数*/
        double hdop; /*gps参数*/
    } gps;
} manage;
```

若想获取适配器 ID，可以

devID = user.getDev("manage", "devid")

若想获取 GPS 的状态，可以

gps_status = user.getDev("manage", "gps.status")

若想获取文件系统版本，可以

gps_status = user.getDev("manage", "sysver")

其余的参数依次类推，不再一一列举。

```
struct
{
    char ver[16];    /*网络程序版本*/
    char autotype;  // 识别到的设备网络类型(1: eth0;2:wifi;3:4g;4:eth1)
    char netstatus; // 网络连接的状态(1: 未获取IP地址; 3: 已获取IP地址)
    struct
    {
        char status;    /*4G状态*/
        char csq;       /*4G信号值*/
        char model[20]; // 连接成功后显示为: EC600N
        char imei[32];  // 连接成功后显示imei, 如: 867453055794435
        char ccid[32];  /*4G参数*/
        char info[64];  /*4G参数*/
    } ecm;

    struct
    {
        char status;   /*wifi参数*/
        char ssid[32]; /*wifi用户名*/
        char pwd[32];  /*wifi密码*/
        char ip[16];   /*wifi的IP*/
        char signal;   /*wifi的信号*/
    } wifi;

    struct
    {
        char status; /*网口的状态*/
        char ip[16]; /*网卡IP*/
        char mode;   /*网卡模式*/
    } eth[2];
} net;
```

若想获取网络程序版本，可以
net_ver = user.getDev("net", "ver")
若想获取 ECM(4G)的 IMEI 数据，可以
ecm_imei = user.getDev("net", "ecm.imei")
若想获取网卡的数据（两个网卡索引从 0 开始），可以
eth0_status = user.getDev("net", "eth[0].status")
eth1_status = user.getDev("net", "eth[1].status")

eth0_ip0 = user.getDev("net", "eth[0].ip")
eth1_ip1 = user.getDev("net", "eth[1].ip")
其余的参数依次类推，不再一一列举。

```
struct
{
    char softver[16]; /*mixrtu程序版本*/
    char luadev[48];  /*Lua*/
    char luaver[48];  /*Lua版本*/
    char luapath[48]; // RW:"/home/root/app" ; RO:/sysenv文件中的luapath的值
    char mqttnum;     /*mqtt数量*/
    struct
    {
        char mqttserver[APP_MQTT_HOST]; /*mqtt服务器*/
        int port;                       /*mqtt端口*/
        char user[APP_MQTT_NAME_LEN];   /*mqtt用户名*/
        char pwd[APP_MQTT_PASSWD_LEN];  /*mqtt密码*/
        char status;                    // 0: 未连接服务器; 1: 已连接服务器
        long long int sendbyte;         /*mqtt发送数据流量，单位byte*/
        long long int sendn;            /*mqtt发送次数*/
        int connectn;                   /*mqtt链接次数*/
        int connecterrn;                /*mqtt链接失败次数*/
    } mqtt[MaxCount_mqtt];

    struct
    {
        char protocol[16]; /*协议类型*/
        char status;       /*状态*/
    } collect[MaxCount_collect];
} app;
```

若想获取 mixrtu 程序版本，可以

```
mixrtu_ver = user.getDev("app", "softver")
```

若想获取 Lua 程序版本，可以

```
lua_ver = user.getDev("app", "lua_ver")
```

若想获取 mqtt 服务器，可以

```
mqtt0_svr = user.getDev("app", "mqtt[0].mqttserver")
mqtt1_svr = user.getDev("app", "mqtt[1].mqttserver")


mqtt0_pwd = user.getDev("app", "mqtt[0].pwd")
mqtt1_pwd = user.getDev("app", "mqtt[1].pwd")
```

其余的参数依次类推，不再一一列举。

1.2. DataCenter API

1.2.1. DataCenter.addDB

功能	添加新的数据库
接口描述	DataCenter.addDB(arg)
arg	数据库属性描述<字符串>
例	DataCenter.addDB("dbName="DB_3_100", dStart=0, count=100, dSize=2, dType=2")

dbName	数据库名称<字符串>，用来查找、索引数据库
dStart	数据单元起始编号（Modbus 等需要使用。默认为 0）
count	数据单元数量
dSize	数据单元的大小（占用字节数）
dType	数据单元的类型（暂时无作用）

1.2.2. DataCenter.getDB

功能	**获取数据库指针**
接口描述	pDB = DataCenter.getDB(dbName)
dbName	要查找的数据库名称<字符串>
pDB	返回值<数据库指针>。如果未找到，则返回 NULL

1.2.3. DataCenter.removeDB

功能	**删除数据库**
接口描述	DataCenter.removeDB(dbName)
dbName	要删除的数据库名称<字符串>

1.2.4. DataCenter.removeAllDB

功能	**删除所有数据库**
接口描述	DataCenter.removeAllDB()

1.2.5. DataCenter.debug

功能	**设置数据库 LOG 打印级别**
接口描述	DataCenter.debug(LogLevel)
LogLevel	0x00000101

1.2.6. DataCenter.DBresize

功能	**重设数据库的数据单元数量**
接口描述	DataCenter.DBresize(pDB, count)
pDB	数据库指针
count	数据单元数量（如果新数量比原来的数量小，多余的数值将会丢失）

1.2.7. DataCenter.DBdestroy

功能	**销毁数据库内部的资源**
接口描述	DataCenter.DBdestroy(dbName)
dbName	要销毁的数据库名称<字符串>

1.2.8. DataCenter.DBclear

功能	清空数据库内所有数据
接口描述	DataCenter.DBclear(dbName)
dbName	要清空的数据库名称<字符串>

1.2.9. DataCenter.DBupdateRecord

功能	更新数据库中某个数据单元（附带更新时间）
接口描述	DataCenter.DBupdateRecord(pDB,index,value,time)
pDB	数据库指针
index	数据单元编号
value	新的数值
time	新的时间<Unix 时间戳（u32 类型）>

1.2.10. DataCenter.DBupdateRecords

功能	更新数据库中多个数据单元（附带更新时间）
接口描述	DataCenter.DBupdateRecords(pDB,index,count,pvalue,time)
pDB	数据库指针
index	数据单元编号
count	更新的数据单元数量
pvalue	新的数值指针（必须与数据库中数据同样字长）
time	新的时间<Unix 时间戳（u32 类型）>

1.2.11. DataCenter.DBsetRecord

功能	更新数据库中某个数据单元
接口描述	DataCenter.DBsetRecord(pDB,index,value)
pDB	数据库指针
index	数据单元编号
value	新的数值

1.2.12. DataCenter.DBsetRecords

功能	更新数据库中多个数据单元
接口描述	DataCenter.DBsetRecords(pDB,index,count,pvalue)
pDB	数据库指针
index	数据单元编号
count	更新的数据单元数量
pvalue	新的数值指针（必须与数据库中数据同样字长）

1.2.13. DataCenter.DBgetRecord

功能	获取数据库中某个数据单元的数值
接口描述	ret,val = DataCenter.DBgetRecord(pDB,index)
ret	函数返回值，ret=0 时证明函数运行无错，val 有效
val	数据单元的数值
pDB	数据库指针
index	数据单元编号

1.2.14. DataCenter.DBgetRecords

功能	获取数据库中多个数据单元的数值（暂不支持）
接口描述	ret = DataCenter.DBgetRecords(pDB,index,count,pvalue)
ret	函数返回值，ret=0 时证明函数运行无错，pvalue 内数据有效
pDB	数据库指针
index	需要获取的起始数据单元编号
count	需要获取的数据单元数量
pvalue	输出数值指针，获取的多个数据会被放置到这个数组中。（必须与数据库中数据同样字长）

1.3. mqtt V3 API

1.3.1. mqtt.new

功能	创建 mqtt 实例
接口描述	obj = mqtt.new()
obj	创建并返回的 mqtt 实例对象

1.3.2. mqtt.subscribe

功能	订阅 mqtt 主题消息
接口描述	subscribe(obj,topic)
obj	mqtt 实例对象
topic	订阅的主题<字符串>

1.3.3. mqtt.unsubscribe

功能	取消订阅 mqtt 主题消息
接口描述	unsubscribe(obj,topic)
obj	mqtt 实例对象
topic	订阅的主题<字符串>

1.3.4. mqtt.config

功能	配置 mqtt server 连接信息
接口描述	config(obj, mqttid, serverip, serverport)
obj	mqtt 实例对象
mqttid	mqttid<字符串>，当填 nil 时，则默认使用设备 ID 作为 mqtt id
serverip	mqtt 服务器地址<字符串>
serverport	mqtt 端口号<字符串>

1.3.5. mqtt.publish

功能	发布消息
接口描述	publish(obj, taginfo, topic, payload)
obj	mqtt 实例对象
taginfo	标记消息，用于日志输出查看<字符串>，没有可填 nil
topic	报文 topic<字符串>
payload	报文消息<字符串>

1.3.6. publish_hex

功能	发布十六进制消息
接口描述	publish_hex(obj, taginfo, topic, payload)
obj	mqtt 实例对象
taginfo	标记消息，用于日志输出查看<字符串>，没有可填 nil
topic	报文 topic<字符串>
payload	报文消息<字符串> 例："0e0a0b" 实际发送为 3 字节十六进制数据

1.3.7. mqtt.run

功能	启动 mqtt 实例
接口描述	run(obj)
obj	mqtt 实例对象

1.3.8. mqtt.stop

功能	停止 mqtt 实例
接口描述	stop(obj)
obj	mqtt 实例对象

二、 aprus.Lua 配置样例

2.1. apurs.lua 示例

```
package.cpath="./?.so"
package.path="./?.lua"
cjson = require "cjson"
config = require "config"
require "myModule"

function act_control(m, json)
    for k,v in pairs(json) do
        if k ~= "Act" then
            myModule.write(moduleObj, cjson.encode({["name"] = k, ["val"] = v}))
        end
    end
end

function mqttdata_handle(m, topic, data)
    local json = cjson.decode(data)
    if json.Act == "Control" then
        act_control(m, json)
    end
end

function mqttsys_handle(m, code)
    if code == 0 then
        myModule.stop(moduleObj)
    elseif code == 1 then
        myModule.run(moduleObj)
    end
end

function myModule_handle(obj, name, code, data)
    mqtt.publish(mqtt3Obj, name, "r", data)
end

function myModule_load_collectnodes(obj, nodes)
    for k,v in pairs(nodes)
    do
        myModule.addcnode(obj,cjson.encode(v))
    end
end

function myModule_load_varnodes(obj, nodes)
```

```
    for k,v in pairs(nodes)
    do
        myModule.addvnode(obj,cjson.encode(v))
    end
end

function init()
    mqtt3Obj = mqtt.new()
    mqtt.subscribe(mqtt3Obj, "p2p")
    --mqtt.config(mqtt3Obj, nil, "123.456.789.123", "1883", "user", "mix123")
    user.setluaver(config.AprusX.luaver)
    user.setdevinfo(config.AprusX.devinfo)
    user.ipconfig(config.AprusX.ipmode, config.AprusX.inet_addr, config.AprusX.netmask)
end

function start()
    init()
    moduleObj = myModule.new("myModule")
    myModule.config(moduleObj, cjson.encode(config.myModule.Device))
    myModule_load_collectnodes(moduleObj,config.myModule.ColNode)
    myModule_load_varnodes(moduleObj,config.myModule.VarNode)
    mqtt.run(mqtt3Obj)

    while true do
        local msg = user.waitmsg()
        if msg.from == "mqtt-sys" then
            mqttsys_handle(msg.session, msg.code)
        elseif msg.from == "mqtt-msg" then
            mqttdata_handle(msg.session, msg.topic, msg.payload)
        elseif msg.from == "myModule" then
            myModule_handle(msg.obj, msg.name, msg.code, msg.data)
        end
    end
end

start()
```

(1) 代码中”--”后面的部分为注释语句，仅供解释代码意图，不对程序流程有所干预。

(2) 在源码的开头部分调用需要的模块

require "myModule" --其中 myModule 为用户需要模块名称，比如 modbus 或 s7

(3) 所有函数以 function 开头， end 结尾。代码使用 TAB 缩进对齐。

例：

```
function mqttdata_handle(m, topic, data)
    local json = cjson.decode(data)
    if json.Act == "Control" then
        act_control(m, json)
```

```
    end
end
```

这是一个名叫 mqttdata_handle()的函数，它有三个参数，分别是 m, topic, data。

(4) 最后一行表示调用 start()函数，是整个程序的起始。

向上搜索 function start()，可以找到 start()的函数定义。

(5) 下面将逐行解读 start()函数的每一行语句

```
--函数定义：函数名 start，  无参数
function start()
    --先调用 init()函数进行一些初始化操作。
    init()
    --调用模块的功能 myModule.new()创建一个新的模块实体对象 moduleObj。
    moduleObj = myModule.new("myModule")
    --调用 myModule.config()功能，对模块实体对象 moduleObj 进行配置。
    myModule.config(moduleObj, cjson.encode(config.myModule.Device))
    --调用 myModule_load_collectnodes()函数，配置 moduleObj 的采集节点
    myModule_load_collectnodes(moduleObj,config.myModule.ColNode)
    --调用 myModule_load_varnodes()函数，配置 moduleObj 的上报节点
    myModule_load_varnodes(moduleObj,config.myModule.VarNode)
    --调用 mqtt.run()功能，运行 mqtt 的实体对象 mqtt3Obj
    mqtt.run(mqtt3Obj)

    --循环下面一段代码，反复运行
    while true do
        --接收消息，并把消息保存在本地变量 msg 中
        local msg = user.waitmsg()
        --判断消息 msg 的来源，如果是 mqtt 系统消息“mqtt-sys”
        if msg.from == "mqtt-sys" then
            --调用 mqttsys_handle()函数处理消息的内容
            mqttsys_handle(msg.session, msg.code)
        --判断消息 msg 的来源，如果是 mqtt 数据消息“mqtt-msg”
        elseif msg.from == "mqtt-msg" then
            --调用 mqttdata_handle()函数处理消息的内容
            mqttdata_handle(msg.session, msg.topic, msg.payload)
        --判断消息 msg 的来源，如果是来自模块 myModule 的消息“myModule”
        elseif msg.from == "myModule" then
            --调用 myModule_handle()函数处理消息的内容
            myModule_handle(msg.obj, msg.name, msg.code, msg.data)
        --判断结束
        end
    --循环结束
    end
--start()函数结束
end
```

(6) 初始化函数 init()解析

```
--函数定义：函数名 init，  无参数
function init()
```

```
--调用 mqtt 模块的功能 mqtt.new()创建一个新的模块实体对象 mqtt3Obj 。
mqtt3Obj = mqtt.new()
--调用 mqtt 模块的功能 mqtt.subscribe()配置 mqtt3Obj 的订阅模式为”p2p”

mqtt.subscribe(mqtt3Obj, "p2p")
--如果需要，使用 mqtt.config()配置 mqtt3Obj 连接到用户自己的服务器。（现在此行以“--”开头，表示注释掉了，不起作用）

--mqtt.config(mqtt3Obj, nil, "123.456.789.123", "1883", "user", "mix123")
--设置 lua 版本(版本信息来自于配置文件 config.lua 的 AprusX 字段的 luaver)

user.setluaver(config.AprusX.luaver)
--设置设备信息(设备信息来自于配置文件 config.lua 的 AprusX 字段的 devinfo)

user.setdevinfo(config.AprusX.devinfo)
--设置 Aprus 的 IP 地址(信息来自于配置文件 config.lua 的 AprusX 字段)

user.ipconfig(config.AprusX.ipmode, config.AprusX.inet_addr, config.AprusX.netmask)
--init()函数结束
end
```

2.2. user 全局通用/配置类函数

(1) 本机网络配置方法

```
user.ipconfig(config.AprusX.ipmode,config.AprusX.inet_addr,config.AprusX.netmask)
```

(2) 消息捕获方法　　msg = user.waitmsg()

通过 msg.from 可判断消息来源

```
mqtt-sys:　来自 mqtt 系统消息
mqtt-msg:　来自 mqtt 数据消息
modubs:　　来自 modbus 管理器消息
```

(3) 以下为 msg.from 为 modbus、opcua 、mitsufx、mitsumc、dlt645、s7 等协议时通用

msg.session: 消息会话对象

msg.code：消息号

msg.style_L：变量名称

msg.val_L：变量值

msg.style_E：事件变量名称

msg.val_E：事件变量值

msg.z：条件改变标志位

(4) 以下为 msg.from 为 mqtt 时使用

msg.topic：mqtt 话题

msg.payload：mqtt 消息数据

(5) 以下为 msg.from 为 mqtt-sys 时使用

msg.code：mqtt 事件

0： mqtt 断开

1： mqtt 连接

(6) 设置 Lua 版本信息

user.setLuaver(Luaver)

(7) 设置设备信息

user.setdevinfo(devinfo)

2.3. mqtt 对象操作方法函数

(1) 创建 mqtt 对象实例

mqttobj = mqtt.new()

(2) 配置 mqtt 连接信息 (选填)

mqtt.config(mqttobj, "mqtt_1", "192.168.1.159", "1883")

param 1 mqtt 对象实例

param 2 实例 id

param 3 mqtt 服务器 ip

param 4 mqtt 服务器端口

注：此配置选填 ,如果不填,则通过 gards 服务器自动指定

(3) 话题订阅

mqtt.subscribe(mqttobj, "p2p")

(4) 发布消息

mqtt.publish (mqttobj, reserve, topic, payload)

reserve: 预留参数,必须有,可填 nil

(5) mqtt 运行

mqtt.run(mqttobj)

备注：支持多实例，最大可创建 3 个 mqtt 对象

三、 config.Lua 配置样例(Modbus)

config.Lua 一般由 3 大部分组成：适配器参数（AprusX 段）、协议接口参数（Device 段）和节点信息（node 段）；节点信息又细分为**采集节点（CNode）**和**上报节点（VNode）**。

3.1. 适配器参数：AprusX 字段

适配器参数的功能如下表所示：

序号	模块	说明	备注
1	ipmode	IP 获取方式	默认 Manual，不更改
2	inet_addr	APRUS IP	与对接的设备保持相同网段，且同网段内 IP 不能重复
3	netmask	子网掩码	默认 255.255.255.0，一般不修改，除非相同网段内没有足够的 IP
4	luaver	脚本版本号	每次变更脚本都需更新版本，以便区分不同版本的区别
5	devinfo	设备类型	对接的设备类型

示例：

```
AprusX={
    ipmode="Manual",
    inet_addr="192.168.1.234",
    netmask="255.255.255.0",
    luaver="MAPRUS X.Lua.V030300.R",
    devinfo="Modbus-kyj",
},
```

ipmode: APRUS 的 ip 获取方式

ipmode="Manual"　　--手动设置成固定 IP 地址

ipmode="Auto"　　--自动获取 IP 地址

inet_addr: APRUS 设备 Ip 地址

inet_addr="192.168.1.234"

配置设备的 IP 地址，值为类似"192.168.1.234"的字符串。此 IP 地址需要与对接设备保持在相同的网段内，否则无法与对接设备通过网口通信。

netmask:　子网掩码

netmask="255.255.255.0"。

子网掩码，默认为 255.255.255.0，与客户保持一致，一般情况下均为 255.255.255.0 不变。

devinfo:　设备类型

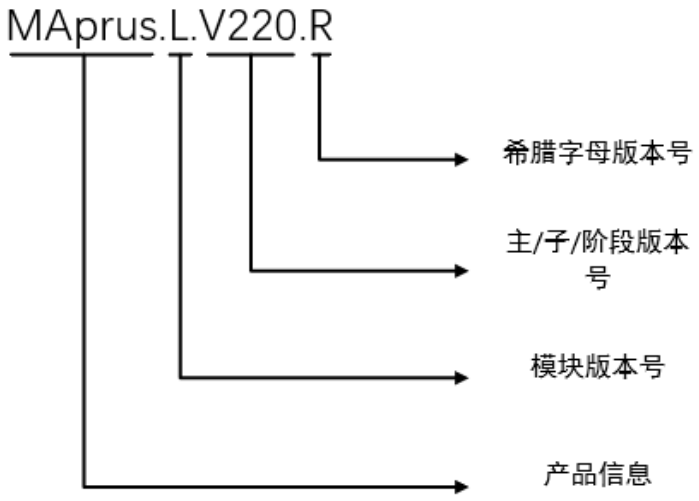
devinfo="Modbus-kyj"。

设备类型，用来标识对接的是何种设备。不同的适配器对接的设备类型不同，比如对接西门子 Modbus-200-SMART，对接西门子 Modbus-300 等等，由于对接的设备类型不同，因此为了方便使用，知道对接的是哪类设备，我们设定了设备类型。

luaver: 脚本版本

luaver="MAPRUS.L.V030005.R"。

当前脚本的版本号。每次更新都需要更改并向上增加版本号；版本号在增加过程中，只需要修改主/子/阶段版本号即可，其他无需修改，主/子/阶段版本号不能为 0，一般情况下如果只更新 config.Lua 的配置无需更新主/子版本号，只要修改阶段版本号就行，版本号一定是要向上叠加，这样才能保证之前的版本有备份，才能自动升级。



3.2. 协议接口：Device 字段

```
Device={
    type="rtu",          -- rtu/tcp
    rate=38400,
    dataBit=8,
    stopBit=1,
    parity="None",       -- None/Odd/Even
```

Modbus 设备信息:

序号	模块	说明	
1	type	Modbus 协议类型	rtu
			tcp
2	rate=38400	波特率	
3	dataBit=8	数据位：5/6/7/8	
4	stopBit=1	停止位：1/2	
5	parity="None"	校验位：None/Odd/Even	

3.3. 节点信息：node 字段

```
node={
    collect={
            {ID=1, reg="1", addr=0, cnt=50},
        },
    variable={
    {ID=1,reg="3",addr=1,dtype="byte",dBit=0,pMode={1,8},dStyle={"L1_3_1_0"} ,
      dOffset={{"+",10}}, dExt={}},
},
```

(1) 采集配置：collect 字段

collect 设置采集配置，采集配置主要有几个参数构成，需要注意的是 Modbus 协议采集时都是通过双字（short）去采集.

采集参数说明：

采集配置	参数	说明	备注
collect	ID	设备 ID	
	reg	寄存器功能码	1/2/3/4/5/6/15/16
	addr	采集地址起始	
	cnt	采集数据长度	

(2) 上报配置：variable 字段

variable 设置上报配置，上报配置时需要把采集到的数据处理成最终实际使用的数据，包括数据处理逻辑，上报逻辑，数据分类等等；由于采集时都是按照双字（short）采集，所以在数据处理的时候要需要注意。

注意：上报配置设置的地址，一定是在采集配置里面有的，否则数据无法成功获取。

上报配置	参数	说明		备注
variable	ID	设备 ID	ID = 1	0~n
	reg	Modbus 功能码	reg=1	1/2/3/4/5/6/15/16
	addr	上报地址	addr=1	需要上报哪个地址设置那个地址
	dtype	变量数据类型	dtype="short"	bit
				byte
				short/ushort（Word 类型即为 short/ushort）
				Int/uint（Double Word 类型即为 int/uint）
				float
	pMode	上报方式	pMode={1,5}	详见附件一
	dOffset	数据偏移	dOffset={{"+",10},{"*",5}}	数据偏移在关于 bit 处理的时候是不需要的，详见附件二
	dBit	位处理	dBit=5	位标识，详见附件三

	dExt	附件报文处理	dExt={0,{0,0},{0,0}}	(选填)改变上报条件限制，当 pMode 为{2,0}时有效，{">", 100} 表示 当原始数据发生改变 并且改变值大于 100 时上报，{"<", 100} 表示 当原始数据发生改变 并且改变值小于 100 时上报，{"=", 100} 表示 当原始数据发生改变 并且改变值等于 100 时上报
	dStyle	KEY	dStyle={"L1_3_0_0"}	自行设置
	len	字符串长度	len=5	当 dtype 为 bytes 时有效，表示字符串长度
	format	字节序配置	format =	选填，当 dtype 为 4 字节数据时有效，详见附件四

附件一

模块		举例	参数 1	参数 2
			上报模式	上报周期
pMode	上报类型	pMode={1,5}	1：周期上报	自行设置上报周期
			2：改变上报	0

附件二

模块		参数 1		参数 2		备注
		偏移参数 1		偏移参数 2		
dOffset	dOffset={{"+",10},{"*",5}}	"+"	N	"+"	N	
		"-"	N	"-"	N	
		"*"	N	"*"	N	
		"/"	N	"/"	N	
		"."	N	"."	N	保留 N 位小数

注意：

1) 如果无偏移计算，则无须填写此字段，或者填写为 dOffset={}。

2) 如果只有单层偏移，那么只需要设置参数 1 即可，假如一个参数需要*10，即 dOffset={{"*",10}}。

3) 如果有双层偏移，那么需要设置参数 1，参数 2，假如一个参数需要*10，然后在+10，即 dOffset={{"*",10},{"+",10}}。

4) 如果上报数据包含小数且需要保留指定的小数位数时，即可对参数 1 进行设置，假如需要保留 3 位小数，即 dOffset={{".",3}}。

附件三

标识	说明
dBit	当 dtype 为 bit 时有效，范围 0~15 标识第几位
	当 dtype 为 byte/ubyte 时有效 范围 0~1 标识 低/高字节

附件四

标识	说明
format	ABCD 为原始采集字节序
	BADC 12 字节对换 34 字节对换
	CDAB 13 字节对换 24 字节对换
	DCBA 14 字节对换 23 字节对换（默认值）