AirM2M

Luat 脚本开发指南 V2.7

合宙开源 OpenLuat QQ 讨论群：201848376

百度云盘： https://pan.baidu.com/s/1eSxFHrs

开源社区：bbs.openluat.com

GitHub：https://github.com/airm2m-open/Luat_Air200

开发套件淘宝购买：

https://luat.taobao.com/

https://openluat.taobao.com/

合宙--》Luat--》发烧友--》客户--》产品

目 录

1. Lua 简介

Lua 是一个小巧的脚本语言。脚本语言不需要事先编译，直接可以运行（其实是在运行的时候进行解释翻译）。该语言的设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 由标准 C 编写而成，代码简洁优美，几乎在所有操作系统和平台上都可以解释，运行。

一个完整的 Lua 解释器不过 200k，在目前所有脚本引擎中，Lua 的速度是最快的。这一切都决定了 Lua 是作为嵌入式脚本的最佳选择。

Lua 使用者分为三大类：使用 Lua 嵌入到其他应用中的、独立使用 Lua 的、将 Lua 和 C 混合使用的。

第一：很多人使用 Lua 嵌入到其他应用程序。

我司模块中支持 Lua 程序开发和应用就属于这种情况。用户将自行开发的 lua 应用程序嵌入在我模块基础软件 Lod 之中。

第二：作为一种独立运行的语言，Lua 也是很有用的，主要用于文本处理或者只运行一次的小程序。这种应用 Lua 主要使用它的标准库来实现，标准库提供模式匹配和其它一些字串处理的功能。我们可以这样认为：Lua 是文本处理领域的嵌入式语言。

第三：还有一些使用者使用其他语言开发，把 Lua 当作库使用。这些人大多使用 C 语言开发，但使用 Lua 建立简单灵活易于使用的接口。

2. Luat 开源 GPRS 模块简介

Luat = Lua + AT, Luat 是合宙（AirM2M）推出的物联网开源架构，依托于通信模块做简易快捷的开发，目前支持的模块有两款：Air200 和 Air810。其中， Air200 是一款 GPRS 模块； Air810 是一款支持 GPRS+北斗 GPS 的二合一模块。

3. 模块基础软件和 Lua 程序之间的关系

Lua 脚本是内嵌在模块基础软件(简称 Lod)中运行的，Lod 中有支持 Lua 运行的环境，lua 脚本就在这个环境中运行。脚本实现功能是通过 API（对 AT 命令进行了封装）实现的。

在 Air 模块内部，Lua 发出 AT 命令，并通过虚拟的 uart.ATC 口和 Lod 之间进行 AT 命令的交互。即 Lua 发出 AT 命令，Lod 接收后进行解析并返回 AT 命令运行结果，不需要上位机（一般是单片机）通过物理串口给模块发 AT 命令，这样就节省了单片机的花费。

这样 Air 模块的物理串口就可以挪作他用，比如连接 GPS 芯片，或 PC 工具软件。Air 模块还有一

个 DEBUG UART 口，可以连接 PC 工具软件等。

模块基础软件和 Lua 脚本（上层软件）的关系示意图如下：

图1　模块基础软件和 Lua 脚本程序的关系示意图

4. 合宙 Lua 项目开发和调试

4.1 Luat 开源架构简介

合宙用 Luat 架构开发的 GPRS 模块相关的产品和项目已经做了开源，地址是：
https://github.com/airm2m-open/

Luat_Air 系列模块开源代码架构如下：

底层软件（也叫模块基础软件，位于/core）用 C 语言开发完成，支撑 Lua 的运行。

上层软件用 Lua 脚本语言来开发实现，位于/script。

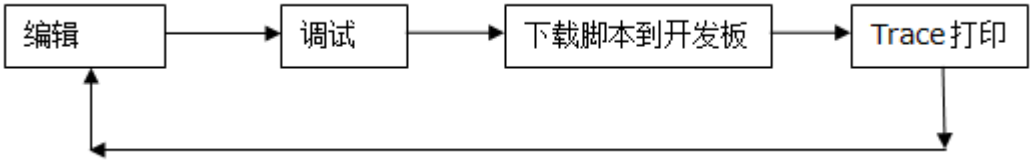
我司提供的开源 lua 脚本代码 script 中，/demo 里是各个功能的示例程序，/product/小蛮 GPS 定位器 是一个完整的定位器代码。/lib 下是 demo、product 以及所有用户代码都需要调用的库文件。

有兴趣的开发者，可以 clone 一份/luat 的所有内容到自己的本地电脑（为方便表述，以开源包来指代），然后以本章所介绍的一般开发步骤进行开发调试。

本文档第二页也同时给出了其他下载地址。对不习惯使用 git 的二次开发用户，也可以到我司开源 QQ 群群文件、百度网盘、开源论坛中下载我司的开源资料包。

4.2 Luat IDE 开发环境的安装和使用

Luat IDE 是 Luat 专用的集成开发环境，集编辑脚本、调试脚本、下载脚本、trace 打印功能于一体，与 Luat 系列开发板一起构成 Luat 集成开发的完整环境。



具体，请点开：Luat IDE 的安装和使用

5. Lua 语法简介

5.1 词法约定

变量是以字母(letter)或者下划线开头的字母、下划线、数字序列。请不要使用下划线加大写字母的标示符，因为 Lua 的保留字也是这样的。

以下字符为 Lua 的保留字，不能当作变量使用。

and	break	do	else	elseif
end	false	for	function	if
in	local	nil	not	or
repeat	return	then	true	until
while				

注意：Lua 是大小写敏感的。 如 and 是保留字，但是 And 和 AND 则不是 Lua 的保留字。

在 SciTE 中输入保留字的时候，会自动变为蓝色粗黑体。

5.2 代码规范

♦ 书写规范

Lua 的多条语句之间并不要求任何分隔符，如 C 语言的分号(;)，其中换行符也同样不能起到语句分隔的作用。因此下面的写法均是合法的。如：

```
a = 1
b = a * 2

a = 1;
b = a * 2;

a = 1; b = a * 2;
a = 1  b = a * 2
```

♦ 注释

Lua 代码中的注释分为 2 种：

一种是单行注释，如：

```
-- This is a single line comment
```

另一种是多行注释，如：

```
--[[
```

```
This is a multi-lines comment
--]]
```

5.3 变量

Lua 变量分为**全局**变量和**局部**变量。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，全局变量在整个文件中起作用。访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil.

```
print(b)                                --> nil
b = 10
print(b)                                -->10
```

如果你想删除一个全局变量，只需要将变量赋值为 nil

```
b = nil
print(b)                                --> nil
```

局部变量通过 local 来声明。与全局变量不同，局部变量只在被声明的那个代码块内有效。代码块：指一个控制结构内，一个函数体，或者一个 chunk（变量被声明的那个文件或者文本串）。

例如：

```
local m = 9

if m <= 10 then
    local m = 5                         -- 这个局部变量 m 的作用范围是 then 和 end 之间
    print(m)
end
```

运行结果是 5.

```
local m = 9

if m <= 10 then
    print(m)                            -- 这个 m 是一开始声明的局部变量 m
end
```

运行结果是 9.

另外 Lua 是动态类型语言，声明变量的时候不要类型定义。变量的类型和当前所赋值的类型是一致的。我们可以通过 type 函数获得变量的或值的类型信息，该类型信息将以字符串的形式返回。

例如：

```
local k = 1                             -- k 是局部变量
print(type(k))                          --> number

k = "today is Friday"                   -- 将一个字符串赋值给 k
print(type(k))                          --> string
```

```
k = {1,2}                          -- 将一个表赋值给 k
print(type(k))                     --> table

local d                            -- d 是局部变量，声明的时候未赋值
print(type(d))                     --> nil
d = 11                             -- d 后来被赋值
print(type(d))                     --> number

h = 9                              -- h 是全局变量
print(type(h))                     --> number
```

5.4 值和类型

Lua 中的值有 8 种类型：nil、boolean、number、string、userdata、function、thread 和 table。

- **nil 型**

Lua 中特殊的类型，他只有一个值：nil，它的主要功能是由于区别其他任何值。一个全局变量没有被赋值以前默认值为 nil；给全局变量负 nil 可以删除该变量。Lua 将 nil 用于表示一种"无效值"的情况。

- **boolean 型**

该类型有两个可选值：false 和 true。但要注意 Lua 中所有的值都可以作为条件。在 Lua 中只有当值是 false 和 nil 时才视为"假"，其它值均视为真，如数字零和空字符串，这一点和 C 语言是不同的。

- **number 型**

表示实数。由于资源所限，我司提供的模块 Lua 开发平台不支持小数和浮点型的 number 类型，目前只支持整数。

- **string 型**

指字符的序列。lua 是 8 位字节，所以字符串可以包含任何数值字符，包括嵌入的 0。这意味着你可以存储任意的二进制数据在一个字符串里。

例如：

```
m = "abc"
n = "\97\98\99"                    -- "a"的数值是 97(decimal)，"b"是 98，"c"是 99
print(m,n)                         --> abc   abc
print("\09798")                    --> a98
```

string 和其他对象一样，Lua 自动进行内存分配和释放，一个 string 可以只包含一个字母也可以包含一本书，Lua 可以高效的处理长字符串，1M 的 string 在 Lua 中是很常见的。可以使用单引号或者双引号把一串字符括起来表示字符串：

```
a = "hello"
b = 'world'
```

为了风格统一，最好使用一种。例如一直使用双引号来表示字符串。如果双引号之间有单引号，系统会自动识别此单引号；如果双引号中之间还有双引号，则里面的双引号需要用转义字符\来区别：

例如：

```
print("one line\nnext line\n\"in quotes\", 'in quotes'")
```

运行结果是：

```
one line
next line
"in quotes", 'in quotes'
```

另外记住： Lua 中的字符串是恒定不变的。String.sub 函数以及 Lua 中其他的字符串操作函数都不会改变字符串的值，而是返回一个新的字符串。一个常见的错误是：

```
string.sub(s, 2, -2)
```

认为上面的这个函数会改变字符串 s 的值。其实不会。

如果你想修改一个字符串变量的值，你必须将变量赋给一个新的字符串：

```
s = string.sub(s, 2, -2)
```

- **table 型**

table 是 Lua 中唯一的数据结构，其他语言所提供的数据结构，如：数组、矩阵、链表、队列等，Lua 都是通过 table 来很好地实现。

Lua 中 table 的键(key)可以为任意类型(nil 除外)。当通过整数下标访问 table 中元素时，即是数组。此外，table 没有固定的大小，可以动态的添加任意数量的元素到一个 table 中。例如：

```
local t = {32,"jeep", name = "cherry"}
local p = {["name"] = "peach"}
print(t[1])                          --> 32
print(t[2])                          --> jeep
print(t["name"],t.name)              --> cherry   cherry
print(p["name"],p.name)              --> peach   peach
```

- **function 型**

函数是第一类值，意味着函数可以存储在变量中，可以作为函数的参数，也可以作为函数的返回值。这个特性给了语言很大的灵活性

- **userdata 型**

userdata 可以将 C 数据存放在 Lua 变量中，userdata 在 Lua 中除了赋值和相等比较外没有预定义的操作。userdata 用来描述应用程序或者使用 C 实现的库创建的新类型。

5.5 表达式

Lua 中的表达式包括数字、字符串、一元和二元操作符、函数调用。还可以是非传统的表构造。

- **算数表达式**

算数表达式是算数操作符及其操作对象所组成的表达式。Lua 中算数操作符的操作对象是实数。

Lua 中的算数操作符包括：

二元的算数操作符： +-*/^(指数)%(取模)

一元的算数操作符：　-(负号)

♦　**关系表达式**

由关系操作符及其操作对象所组成的表达式就是关系表达式。所有关系表达式的结果均为 true 或 false。
Lua 支持的关系操作符有：>、<、>=、<=、==、~=。
有几点需注意：
==和~=这两个操作符可以应用于两个任意类型的值。
如果两个值的类型不同，Lua 就认为他们不等。nil 值与其自身相等。例如：

```
print("0" == 0)                    -->false
print(nil == false)                -->false
```

特别地，tables、userdata、functions 是通过引用进行比较的。也就是说，只有当他们引用同一个对象时，才视为相等。例如：

```
a = {x=1,y=2}
b = {x=1,y=2}
c = a

print(a==c)                        -->true
print(b==c)                        -->false
```

Lua 比较数字按传统的数字大小进行，比较字符串按字母的顺序进行，但是字母顺序依赖于本地环境。
例如：

```
print( 3 < 25)                     -->true
print( “3” <"25")                  -->false (alphabetical order!)
```

把字符串或数字用>、<、>=、<= 符与不同类型的值比较时，会报错。
例如：

```
print( 3 <“25”)                    --> attempt to compare number with string
print(5>=false)                 -->attempt to compare boolean with number
```

♦　**逻辑表达式**

用逻辑运算符将关系表达式或逻辑量连接起来的有意义的式子称为逻辑表达式。
逻辑运算符有 3 个：　and or not
逻辑运算符认为 false 和 nil 是假（false），其他为真，0 也是 true。
and 和 or 的运算结果不一定是 true 和 false，而是和它的两个操作数相关。

```
a and b                            --  如果 a 为 false，则返回 a，否则返回 b
a or b                             --  如果 a 为 true，则返回 a，否则返回 b
```

例如：

```
print(4 and 5)                     --> 5
print(nil and 13)                  --> nil
print(false and 13)                --> false
print(4 or 5)                      --> 4
print(false or 5)                  --> 5
print(true and false)              --> false
print(true or false)               --> true
```

♦ **字符串连接**

字符串运算符是 .. 。

字符串连接，如果操作数为数字，Lua 将数字转成字符串。例如：

```
print (“hello” .. “everyone”)          -->hello everyone
print ( 2 .. “apples”)                 --> 2   apples
s = 2 .. 3
print(s , type(s))                     --> 23     string
```

♦ **表的构造**

表（table）构造器是用于创建和初始化表的表达式。其中最简单的构造器是空构造器{}，用于创建空表。

```
b = {x = 0, y = 1,"Monday", 109}
print(b[1],b[2],b.x,b.y)          -->Monday 1090    1
```

5.6 函数

函数有两种用途：

1.完成指定的任务，这种情况下函数作为调用语句使用。例如：

```
print(8*9,“star” )                     --> 72   star
print(os.date())                       -->07/30/14 15:17:36 调用函数的时候，如果参数列表为空，必须使用()表明是函数调用
```

2.计算并返回值，这种情况下函数作为赋值语句的表达式使用。例如：

```
l = string.len(“1234567”)
print(l)                               --> 7
```

♦ **Lua 函数实参和形参的匹配与赋值语句类似，多余部分被忽略，缺少部分用 nil 补足**

```
function f(a, b)
    return a and b
end
c = f(3)d = f(3,4)e= f(3,4,5)
print(c, d, e)                         --> nil   4   4
```

♦ **Lua 函数支持返回多个结果值**

例如：有三个函数定义

```
function foo0 () end                   -- returns no results
function foo1 () return 'a' end        -- returns 1 result
function foo2 () return 'a','b' end    -- returns 2 results
```

1. 当调用作为表达式最后一个参数或者仅有一个参数时，根据变量个数函数尽可能多地返回多个值，不足补 nil，超出舍去。

```
x,y = foo2()                           -- x='a', y='b'
```

```
x = foo2()                    -- x='a', 'b' is discarded
x,y,z = 10,foo2()             -- x=10, y='a', z='b'
x,y = foo0()                  -- x=nil, y=nil
x,y = foo1()                  -- x='a', y=nil
x,y,z = foo2()                -- x='a', y='b', z=nil
```

2. 其他情况下，函数调用仅返回第一个值（如果没有返回值为 nil）

```
x,y = foo2(), 20              --   x='a', y=20
x,y = foo0(), 20, 30          --   x='nil', y=20, 30 is discarded
print(foo2(), 1)              --> a 1
print(foo2() .. "x")          --> ax
a = {foo0(), foo2(), 4}       -- a[1] = nil, a[2] = 'a', a[3] = 4
```

♦ **函数的可变参数**

Lua 函数可以接受可变数目的参数，在函数形参中用三点（...）表示函数有可变的参数。Lua 将函数的可变参数放在一个叫 arg 的表中，除了参数以外，arg 表中还有一个域 n 表示参数的个数。

在函数参数中，固定参数和可变参数可以一起声明，但是固定参数一定要在变长参数之前声明。

```
function test(arg1,arg2,...)
    ...
end
```

5.7 基本语法

♦ 赋值语句

Lua 中的赋值语句和其它编程语言基本相同，唯一的差别是 Lua 支持"多重赋值"。

例如：

```
local x,y = "test", 12        -- "test" 赋值给 x，12 赋值给 y
```

♦ 局部变量和块

可以用 local 来定义局部变量，例如：

```
local a = "china"
```

local 是保留字，表示该变量是局部变量。和全局变量不同的是，局部变量的作用范围仅限于其所在的程序块。Lua 中的程序可以为控制结构的执行体、函数执行体或者是一个程序块。举例：

```
local x =12

if x >10 then
    local x = 0
    print("x=",x)             -- 打印结果是： x= 0 ，而不是 12
end
```

♦ 控制语句

1） if 语句

if 语句有 3 种结构：

```
if condition then
    statements
end
```

```
if condition then
    statements
else
    statements
end

if condition1 then
    statements
elseif condition2 then
    statements
…                                   -- 很多个 elseif
    statements
end
```

2） while 语句

while 语句语法如下：

```
while condition do
    statements
end
```

3） repeat 语句

repeat 语句语法如下：

```
repeat
    statements
until condition
```

4） break 和 continue

break 跳出内层循环，continue 不会跳出循环，但会结束本次判断。

6. Lua 模块应用程序开发架构介绍

我们现在已经搭了一个架构，将各个功能放在各个 module 中处理，称为库函数（lib）。自开发用户需要自己创建一个 main.lua 主程序，运行也是从 main 开始，main.lua 是入口文件，且只能存在一个 main.lua。脚本在 sys.run（）中循环运行。所以， sys 模块的 run（）是主架构，这个主架构是用消息机制实现的。用户拿到这个架构可以需要根据自己的需求增加或修改对应的消息处理程序即可实现所需的功能。

6.1 main.lua 主程序

```
require"lcd"
require"keypad"
require"idle"

sys.init()

sys.run()
```

Main.lua 是主程序模块，里面一般包括：
Require 程序 -- 用来加载使用到的 module（以.lua 为后缀。包括 lib 中的 module 和自己写的 module）
sys.init() -- 初始化，ATC 口，物理 UART 口，GPIO，I2C 口等的初始化在这里进行，只执行一次
sys.run() -- 主循环程序，是个无限循环，放在 sys.lua 这个 module 里

6.2 主架构 run()详解

1）该架构是用消息机制实现的。
运行框架基于消息处理机制，消息根据来源分为两种：内部消息和外部消息。

内部消息：lua 脚本调用本文件 dispatch 接口产生的消息，消息存储在 qmsg 表中。
外部消息：底层 core 软件产生的消息，lua 脚本通过 rtos.receive 接口读取这些外部消息。

消息根据类型，也有两种：table 型和 Number 型。Number 型使用比较频繁。

Number 类型的消息目前分为：

- 定时器 timeout 消息（msg.id=rtos.MSG_TIMER=1）
- 串口消息(msg.id = rtos.MSG_UART_RXDATA=2)

又分为虚拟 AT 口（msg.uart_id = uart.ATC）和串口（msg.uart_id = 串口号）

- 键盘消息（msg.id=rtos.MSG_KEYPAD=3）
- 中断消息（msg.id=rtos.MSG_INT=4）
- 电源管理消息（msg.id=rtos.MSG_PMD=5）。

当电池在位状态、电池电压百分比、电池电压、充电器在位状态、充电状态任何一个发生-- 变化，就会

上报该消息。

msg.present （电池在位状态，boolean 型，true 或 false）
msg.level （百分比 0-100，number 型）
msg.voltage （电池电压，number 型）
msg.charger （充电器在位状态，boolean 型，true 或 false）
msg.state （充电状态，number 型，0-不在充电 1-充电中 2-充电停止）

2）各个消息处理程序使用前需要用 regmsg 程序注册到 handlers 表中（除了定时器 timeout 消息和物理串口消息以外）。这样代码改动影响面小，当业务流程有修改的时候，只需要修改具体的消息处理程序的内容即可。

3）run() 流程详细介绍。

```
function run()
    local msg,msgpara

    while true do
        --处理内部消息
        runqmsg()
        --阻塞读取外部消息
        msg,msgpara = rtos.receive(rtos.INF_TIMEOUT)

        --电池电量为0%，用户应用脚本中没有定义“低电关机处理程序”，并且没有启动自动关机定时器
        if not lprfun and not lpring and type(msg) == "table" and msg.id == rtos.MSG_PMD and msg.level == 0 then
            --启动自动关机定时器，60秒后关机
            lpring = true
            timer_start(rtos.poweroff,60000,"r1")
        end

        --外部消息为table类型
        if type(msg) == "table" then
            --定时器类型消息
            if msg.id == rtos.MSG_TIMER then
                timerfnc(msg.timer_id)
            --AT命令的虚拟串口数据接收消息
            elseif msg.id == rtos.MSG_UART_RXDATA and msg.uart_id == uart.ATC then
                handlers.atc()
            else
                --物理串口数据接收消息
                if msg.id == rtos.MSG_UART_RXDATA then
                    if uartprocs[msg.uart_id] ~= nil then
                        uartprocs[msg.uart_id]()
                    else
                        handlers[msg.id](msg)
                    end
                --其他消息（音频消息、充电管理消息、按键消息等）
                else
                    handlers[msg.id](msg)
                end
            end
        --外部消息非table类型
        else
            --定时器类型消息
            if msg == rtos.MSG_TIMER then
                timerfnc(msgpara)
            --串口数据接收消息
            elseif msg == rtos.MSG_UART_RXDATA then
                --AT命令的虚拟串口
                if msgpara == uart.ATC then
                    handlers.atc()
                --物理串口
                else
                    if uartprocs[msgpara] ~= nil then
                        uartprocs[msgpara]()
                    else
                        handlers[msg](msg,msgpara)
                    end
                end
            end
        end
        --打印lua脚本程序占用的内存，单位是K字节
        --print("mem:",base.collectgarbage("count"))
    end
end
```

在 ril.lua 中我们已经用 sys.regmsg("atc",atcreader)把 ATC 口的处理程序注册好了。

如果需要电源管理，可以单独写个应用模块，比如取名为 power.lua，并在里面注册 rtos.MSG_PMD 消息处理程序，并定义相应的处理程序。

如果需要 GPIO 中断，也可以单独写个应用模块，并在里面注册 rtos.MSG_INT 消息的处理程序，并定义相应的处理程序。

6.3 lib 库中的各个模块

另外我们按照功能划分了各个模块（module），里面有该功能常用的程序和处理，也是以.lua 做后缀，当做 lib 库来使用。目前 lib 有以下几个模块：

sys.lua	系统模块，核心模块，里面有常用的系统功能
ril.lua	处理 AT 命令的核心模块
sms.lua	处理短信的命令的模块
link.lua	处理 IP 链路和数据收发命令的模块
cc.lua	处理呼叫控制命令的模块
audio.lua	处理音频相关的命令的模块
pm.lua	处理休眠和唤醒相关的命令的模块
pb.lua	处理电话本相关的命令的模块
net.lua	处理网络相关的命令的模块
update.lua	处理脚本升级功能的模块
common.lua	一些经常用到的通用程序
misc.lua	一些时间处理函数和常用 AT 命令的处理
gps.lua	GPS 相关的库函数
agps.lua	agps 相关的库函数
fly.lua	飞行模式相关的库函数
bluetooth.lua	蓝牙相关的库函数
dbg.lua	模块的错误调试信息相关的库函数
maths.lua	里面是平方根函数
patch.lua	Lua 源码功能中的补丁
mqtt.lua	mqtt 协议管理模块

6.4 应用模块

模块脚本发布给客户的时候只有 lib 库，应用程序模块需要客户根据需求自己来编写。

6.5 模块之间的调用关系

1）需要使用这些模块的时候，只需要在调用模块中用 require 语句加载该模块即可。例如：在 a.lua 中想调用 b.lua，只需要在 a.lua 中 require”b” 。这样，如果需要修改某处功能的时候，只需要修改相应的 module，从而保持代码的简洁和模块化。

2）调用模块可以是 main.lua，也可以是各个分模块（包括 lib 中的模块和应用模块）。但是被调用模块

不可以是 main.lua，只能是各个分模块。

3）在被调模块加载后，可以在调用模块中使用被调模块的全局变量和全局程序，而不能使用被调模块的局部变量和局部程序。各个分模块内以 local 定义的变量和程序是局部程序；而未以 local 定义而直接赋值的变量是全局变量，直接用 function 定义的程序是全局程序。

例如，idle.lua 如果需要使用到 keypad.lua 中的一个全局程序 prockey，此时需要在 idle.lua 开头写上 local keypad = require"keypad" ，并在需要使用的时候用 keypad.prockey 调用即可；如果想调用 keypad.lua 中的全局变量 a，调用方法是 keypad.a。

4) 不可以循环 require。
比如 A 模块 require B 模块，B 模块 require C 模块，C 模块不可以再 require A 模块。

5) 客户自己也能根据业务需求自己编制应用 module。但是请尽量不要改动 lib 库中的 module。

7. 程序注册

在使用相关的程序做各种处理的时候，需要先将程序注册下，否则无法使用。这样做的好处是可以将同类型程序集中放置一个表中，方便管理和修改。除了上面提及的 regmsg( )外， 还有 regurc ( )，regapp( ) ，regnotify( )，regrsp( )，reguart( )。

7.1 regmsg()用来注册相应的消息处理程序

regmsg()这个程序放在 sys.lua 这个库文件中。我司提供的 lua 主架构 sys.run() 目前能够处理的消息中，除定时器消息(定时器函数直接用 sys.timer_start 来启动)和物理串口消息（用 reguart 来注册）外，其余外部消息，例如 AT 命令的虚拟串口数据接收消息、音频消息、充电管理消息、按键消息等，均需要用 regmsg 注册相应的消息处理函数。

- 串口消息
  又分为虚拟 AT 口（msg.uart_id = uart.ATC）和物理串口（msg.uart_id = 串口号）消息。
  其中虚拟 AT 口的消息已经缺省注册了，不需要用户再注册；
  物理串口消息处理函数使用前需要用 reguart 来注册。

- 键盘消息（msg.id=rtos.MSG_KEYPAD=3）
  需要用 regmsg()来注册处理函数

- 中断消息（msg.id=rtos.MSG_INT=4）
  需要用 regmsg()来注册处理函数

- 电源管理消息（msg.id=rtos.MSG_PMD=5）
  需要用 regmsg()来注册处理函数

- 音频消息（msg.id=rtos.MSG_AUDIO）
  需要用 regmsg()来注册处理函数

```
function regmsg(id,handler)
    handlers[id] = handler
end
```

regmsg 处理流程见下面的图：

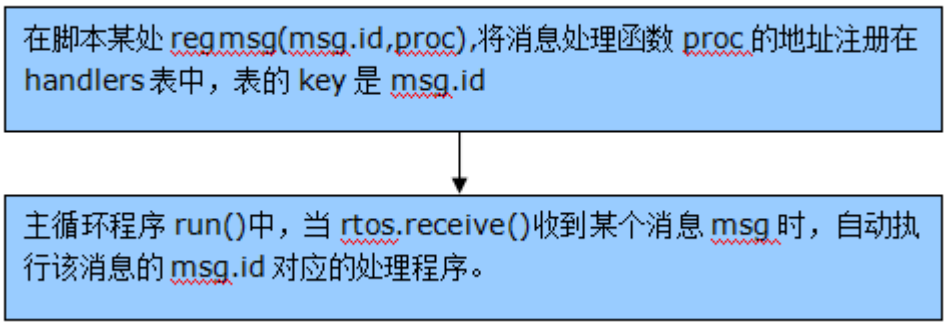


图2　regmsg() 流程图

使用方法举例：

例 1:在应用模块中加入键盘处理程序 keymsg（msg）：

第一步：编制键盘消息处理程序

```
local function keymsg(msg)
    if msg.pressed then
        键盘按下时的处理程序
    else
        键盘弹起时的处理程序
    end
end
```

第二步：注册该处理程序

```
sys.regmsg(rtos.MSG_KEYPAD,keymsg)
```

这样，当 rtos.receive（）收到键盘消息 msg（msg.id=rtos.MSG_KEYPAD=3）时，通过 sys.run()的分发，自动进入 keymsg（msg）这个 function 中进行处理。

例 2:在应用模块中加入 GPIO 中断处理程序

第一步：编制 GPIO 中断处理程序

假设：

gsensor 连接的是 GPIO5

机械式震动传感器连接的是 GPIO3

GPS 连接的是 GPO1

```
IO.gsensor = pio.P0_5
IO.shake = pio.P0_3
IO.gps = pio.P1_1
local function gpio_int（msg）
    --产生下降沿脉冲中断
    if msg.int_id == cpu.INT_GPIO_NEGEDGE then
```

```
        --如果产生中断的 GPIO 是 gsensor 引脚
        if msg.int_resnum == IO.gsensor then
        --如果产生中断的 GPIO 是机械式振动传感器引脚

        elseif msg.int_resnum == IO.shake then

        --如果产生中断的 GPO 是 GPS 引脚
        elseif msg.int_resnum == IO.gps then

        end
    --产生上升沿脉冲中断
    elseif id == cpu.INT_GPIO_POSEDGE then

    end
end
```

第二步：注册该处理程序

```
sys.regmsg(rtos.MSG_INT,gpio_int)
```

例 3：在应用模块中加入 PMD（电源管理）消息处理程序

第一步：编制电源管理消息处理程序

```
local function BatManage(msg)
    print("msg.voltage,msg.charger,msg.state = ",msg.voltage,msg.charger,msg.state)
    DealCharger(msg)
    DealVolt(msg)
end
```

第二步：注册该处理程序

```
sys.regmsg(rtos.MSG_PMD,BatManage)
```

7.2 regapp()用来注册应用程序

对 AT 命令的查询结果或某些事件的处理结果，会以 dispatch（消息 id, 参数 1,参数 2,….）的形式来通知，如果想在 app 中对该消息 id 及对应的参数（参数 1,参数 2,….）进行进一步处理，需要事先以 regapp 方式注册该 app 程序，注册后，每次 dispatch 相应的消息 id，就自动会进入相应的 app 程序对 dispatch 出来的消息 id 和参数进行处理。

***注：dispatch 接口会在第 7 章详细介绍**

regapp 的详细流程详见下图：

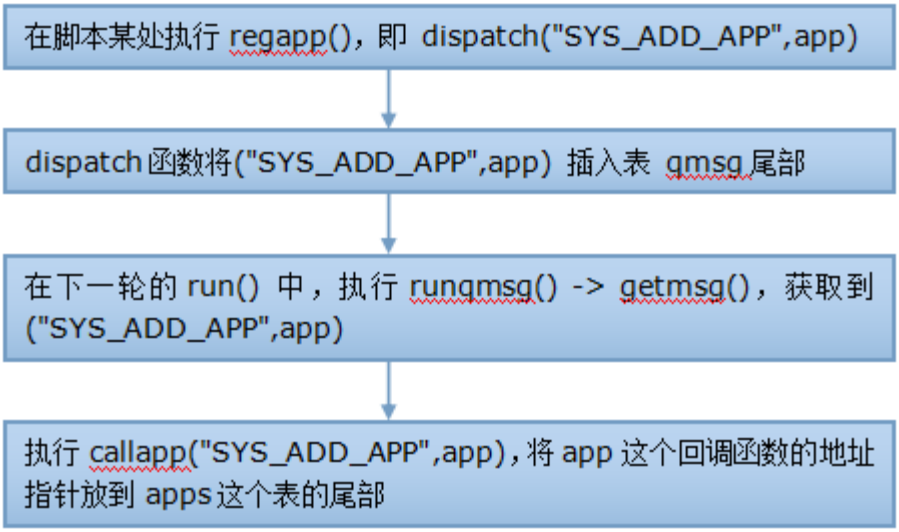


图3　regapp() 流程图

regapp 有 2 种注册方式：

1）　以表的形式来注册

local table = { 消息 id1 = app1, 消息 id2 = app2, …… , 消息 idn = appn }

sys.regapp（table）

这种以表的方式注册 app 程序，一次可以注册多个 app。

使用方法举例：

例：注册多个 dispatch 消息处理程序

第一步：填写注册表

在本例中：

长按键消息（MMI_KEYPAD_LPRESS）和短按键消息（MMI_KEYPAD_SPRESS）由客户编写用程序 dispatch 出来；

通话相关的消息(CALL_CONNECTED,CALL_DISCONNECTED,CALL_INCOMING)由库文件 cc.lua dispatch 出来

```
local idleapp = {
    MMI_KEYPAD_LPRESS = LongPresskey,      --长按键处理
    MMI_KEYPAD_SPRESS = ShortPresskey,     --短按键处理
    CALL_CONNECTED = connect,              --通话接通
    CALL_DISCONNECTED = disconnect,        --通话挂断
    CALL_INCOMING = incall                 --来电话
}
```

第二步：注册该表

```
sys.regapp(idleapp)
```

2） 以 app 的形式注册

sys.regapp（app,消息 id）

以 app 形式注册的程序，一次只可以注册一个 app，但是消息 id 可以不止一个。

使用方法举例：

例：当网络注册状态改变时，闪灯随之改变

第一步：编制闪灯变化程序

```
local function light(id,data)
    -- 当 GSM 网络注册状态发生变化时闪灯发生变化
    if id == "NET_STATE_CHANGED" then
        if data == "REGISTERED" then
            GPIO_ontime(SLOW,ioChgNet)          --蓝灯慢闪
        else
            GPIO_ontime(SLOW,ioChgingNoNet)     --红灯慢闪
        end
    -- 当充电状态发生变化时的闪灯也相应变化
    elseif id == “CHARGE_STATE_CHANGED” then
        if data == “CHARGING” then
            GPIO_ontime(FAST,ioChgingNoNet)     --红灯快闪
        else
            GPIO_on(ioChgNet)                   --蓝灯常亮
        end
    end
end
```

第二步：注册 light 程序

```
sys.regapp(light,"NET_STATE_CHANGED",” CHARGE_STATE_CHANGED”)
```

7.3 regurc()用来注册某些 URC 相应的处理程序

URC=unsolicited result code ，即非请求式上报命令，也就是主动上报的命令。

如果用户需要在应用模块中自己处理感兴趣的 URC，需要用 regurc() 程序来注册 URC 处理程序来达到目的。regurc() 程序放在 ril.lua 中。

```
function regurc(prefix,handler)
    urctable[prefix] = handler
end
```

regurc 处理流程见下面的图：

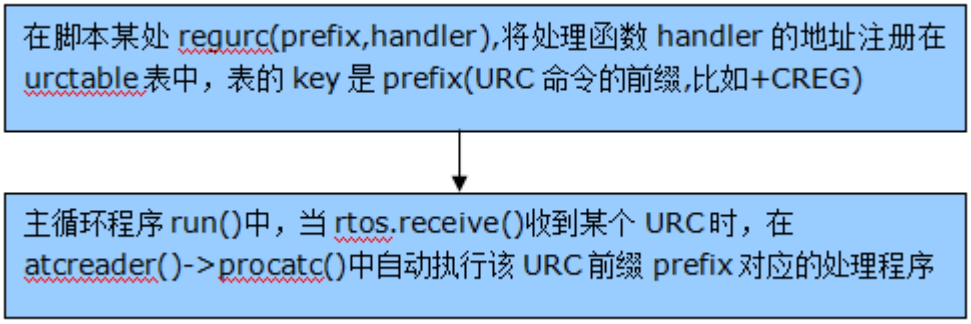


图4　regurc() 流程图

使用方法举例：

例：自己处理 URC

第一步：根据需求，自己编制 URC 的处理程序

```
local function IdleUrc(data,prefix)
    if prefix == "+CPIN" then
        local p = smatch(data,"NOT%s*INSERTED",string.len(prefix)+1)
        if p then
            pbapp.SetSim(false)
        end
        local p = smatch(data,"READY",string.len(prefix)+1)
        if p then
            pbapp.SetSim(true)
        end
    end
end
```

第二步：注册该程序

```
ril.regurc("+CPIN",IdleUrc)
```

7.4　regrsp()用来注册 AT 命令处理程序

当用户发送 AT 命令时，一般是需要这些命令的返回结果的，AT 命令返回结果的处理程序 app 是通过 regrsp()函数注册在 rsptable 表中的。这样当发送 AT 命令时，会自动执行该处理程序处理 AT 命令返回结果。

function regrsp(head,fnc,typ) 放在 ril.lua 这个库文件中。
head 是 AT 命令的头，fnc 是处理函数，typ 是该命令的类型（cmd type: 0:no reuslt 1:number 2:sline 3:mline 4:string 10:spec）。

例如：AT+CHFA?这个获取音频通道命令的 head 是+CHFA? ，fnc 是 AT 命令返回结果处理程序，由客户自己来编写，typ 是 2。

对 AT 命令的返回结果的处理，一般有两种情况：
1）一些常用的 AT 命令的处理程序，库文件已经缺省用 regrsp 注册了。此时处理程序会把 AT 命令结果 dispatch 出来，后续再用 regapp 注册的 app 再做进一步处理（适用于异步处理或前后有因果关系的情况）

2）如果 AT 命令在 lib 库文件中没有被 regrsp 注册过，则需要用 regrsp()注册相应的处理程序。此时用户可以在这些处理程序中直接处理 AT 命令返回结果（适用于同步处理），也可以用 1）的异步处理方式。

regrsp 处理流程见下面的图：

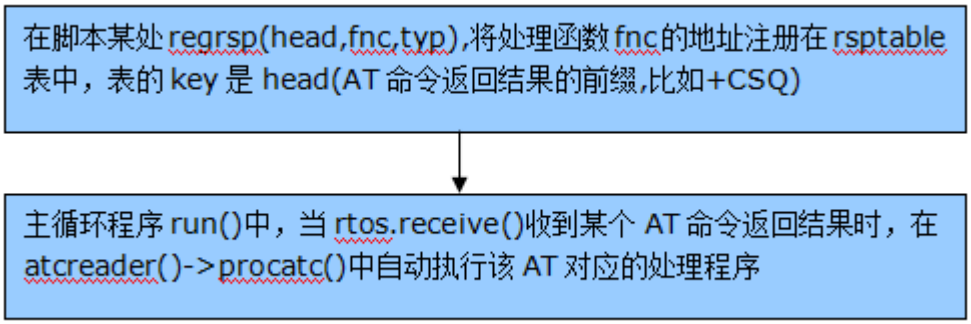


图5　regrsp() 流程图

使用方法举例：
例 1：使用 AT+CPAS?命令获取手机状态
第一步：编制 rsp 程序来处理 AT+CPAS?命令的返回结果

```
local sta
local function rsp(cmd,success,response,intermediate)
    if cmd == "AT+CPAS?" then
        sta = string.match(intermediate,”+CPAS:%s*(%d)”)
    end
end
```

第二步：注册该处理程序

```
ril.regrsp("+CPAS?",rsp,2)
```

7.5　reguart()用来注册 uart 口的数据处理程序

串口(包括物理串口和 debug uart 口)使用之前需要先用 reguart（）程序来注册处理程序。

reguart() 放在 sys.lua 这个库文件中。

```
function reguart(id,fnc)
    uartprocs[id] = fnc
end
```

现在的端口 id 分配如下：
物理端口 1：id = 1
物理端口 2：id = 2
debug 口： id = 3

reguart 处理流程见下面的图：

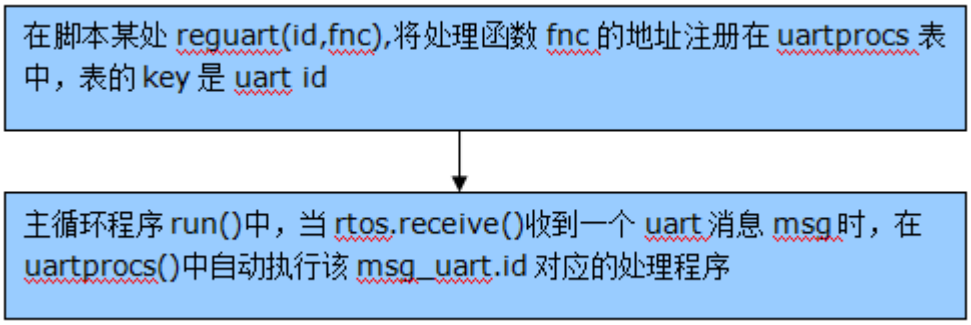


图6 reguart() 流程图

举例说明：如何使用 debug uart 口与上位机通讯
第一步：设置 debug uart 口的通讯特性
uart.setup(3,921600,8,uart.PAR_NONE,uart.STOP_1,2) （debug uart 口波特率必须是 921600 ，mode=2，使用 ID=0xA2 数据透传）

第二步：编写 debug uart 的输入输出处理程序

```
local function huartreader()
    local s

    while true do
        s = uart.read(3,"*l",0)

        if string.len(s) ~= 0 then
            cmddealer(s)
        else
            break
        end
    end
end
```

第三步：注册该处理程序

```
sys.reguart(3,huartreader)
```

8. 各模块详细介绍

全局函数是可以被外部模块调用的，而局部函数不可以被别的模块调用。所以本章将对各个分模块（.lib 文件）中的全局函数接口做详细介绍。

全局函数的调用方法，前面已经讲过：先 require 相关的 lib，然后以库名.函数名的方法进行调用。

8.1 sys 模块详解

sys.lua 这个 module 中除了主框架程序 run（）外，还有些常用的程序：

- 开启定时器

程序名	timer_start（fnc,ms,…）	
功能	开启一个定时器	
输入参数	fnc	function 型。定时器时间 ms 到时后，自动调用 fnc 处理程序
	ms	number 型。该参数设置定时器的时间，以 ms（毫秒）为单位
	…	不定参数，fnc 的参数
返回值	number 型。本次开启定时器分配的 uniquetid。uniquetid 具有独一性，每次调用开启定时器程序 timer_start 的时候分配。	

- 开启循环定时器

程序名	timer_loop_start（fnc,ms,…）	
功能	开启一个循环定时器	
输入参数	fnc	function 型。回调函数。定时器时间 ms 到时后，自动调用 fnc 处理程序
	ms	number 型。该参数设置定时器的时间，以 ms（毫秒）为单位
	…	不定参数，fnc 的参数
返回值	number 型。本次开启定时器分配的 uniquetid。uniquetid 具有独一性，每次调用开启定时器程序 timer_loop_start 的时候分配。	

- 关闭定时器

程序名	timer_stop（val,…）	
功能	结束一个定时器	
输入参数	val	可以为 timer_id，对应上面的 uniquetid，number 型。也可以为 function 类型，对应上面的 fnc
	…	不定参数，fnc 的参数
返回值	无	
举例	例 1：开启定时器，并用 uniquetid 的方式停止定时器 local sms = require”sms” local function sendsms（num,data） sms.send(num,data) end	

	--3 秒后发送短信，收件人号码是 10086，内容是”12345” local tid = timer_start(sendsms,3000,”10086”,”123456”) 如果定时器到时前不想发送了，可以停止定时器 timer_stop(tid)
	例 2：开启定时器，并用 function 的方式停止定时器 还是上面的例子 timer_start(sendsms,3000, ”10086”,”123456”) timer_stop(sendsms, ”10086”,”123456”)

- 关闭某个回调函数标记的所有定时器

程序名	timer_stop_all（fnc）	
功能	关闭某个回调函数标记的所有定时器	
输入参数	fnc	开启定时器时的回调函数，无论开启定时器时有没有传入自定义可变参数
返回值	无	

- 判断定时器是否 active 状态

程序名	timer_is_active (val,...)	
功能	判断某个定时器是否激活状态	
输入参数	val	可以为 timer_id，对应上面的 uniquetid，number 型。也可以为 function 类型，对应上面的 fnc
	...	不定参数，fnc 的参数
返回值	开启返回 true，否则 false	

- 系统初始化

程序名	init(mode, lprfnc)	
功能	系统初始化。ATC 口(即虚拟 AT 命令口)初始化	
输入参数	mode	整数型。充电开机是否启动 GSM 协议栈，1 不启动，否则启动。mode=1 的模式下，充电的时候不会自动开机
	lprfnc	用户应用脚本中定义的“低电关机处理函数”，如果有函数名，则低电时，本文件中的 run 接口不会执行任何动作，否则，会延时 1 分钟自动关机
返回值	无	
举例	例 1：设备充电不开机，按键的时候才开机 在 main.lua 中， sys.init(1) if rtos.poweron_reason() ~= rtos.POWERON_CHARGER then sys.poweron() end sys.run()	

- 消息处理程序注册

程序名	regmsg（id,handler）	
功能	将某类消息的处理程序 handler 注册到一个表 handlers 中 注册“除定时器消息、物理串口消息外的其他外部消息（例如 AT 命令的虚拟串口数据接收消息、音频消息、充电管理消息、按键消息等）”的处理函数	
输入参数	id	消息类型，整数型
	handler	该类消息的处理程序
返回值	无	
举例	详细请参见 regmsg()用来注册相应的消息处理程序	

- app 程序注册

程序名	regapp（...）	
功能	目前支持 2 种注册方式： 1）将一个消息和对应的应用处理程序注册到 apps 表中 2）以表的形式注册到 apps 表中。表中的每个元素 key 是消息，value 是对应的消息处理程序	
输入参数	...	不定参数，有两种方式： ● 以函数方式注册的 app，例如： regapp(fncname,"MSG1","MSG2","MSG3") ● 以 table 方式注册的 app，例如： regapp({MSG1=fnc1,MSG2=fnc2,MSG3=fnc3})
返回值	无	
举例	详细请参见 regapp()用来注册应用程序	

- app 程序解注册

程序名	deregapp（id）	
功能	将 id 这个子程序或表从 apps 表中删除	
输入参数	id	app 的 id，id 共有两种方式，一种是函数名，另一种是 table 名
返回值	无	

- 物理串口处理程序注册

程序名	reguart(id,fnc)	
功能	注册物理串口的数据接收处理函数	
输入参数	id	整数型。物理串口号
	fnc	function 型。处理串口数据的程序
返回值	无	
举例	详细请参见 reguart()用来注册 uart 口的数据处理程序	

- 消息分发

程序名	dispatch(...)	
功能	消息分发	
输入参数	...	不定参数，但其中第一个参数是消息 id
返回值	无	
举例	详细请参见消息分发函数 dispatch 详细介绍	

- 获取工作模式

程序名	getworkmode()
功能	获取工作模式
输入参数	无
返回值	当前工作模式，0- SIMPLE_MODE；1-FULL_MODE： --SIMPLE_MODE：简单模式，默认不会开启“每一分钟产生一个内部消息”、“定时查询 csq”、“定时查询 ceng”的功能 --FULL_MODE：完整模式，默认会开启“每一分钟产生一个内部消息”、“定时查询 csq”、“定时查询 ceng”的功能

- 设置工作模式

程序名	setworkmode(v)	
功能	设置工作模式	
输入参数	v	工作模式。0- SIMPLE_MODE；1-FULL_MODE。
返回值	成功返回 true，否则返回 nil	

- 开启或关闭 trace

程序名	opntrace(v)	
功能	开启或者关闭 print 的打印输出功能	
输入参数	v	false 或 nil 为关闭，其余为开启
返回值	成功返回 true，否则返回 nil	

- 主循环程序

程序名	run()
功能	主架构程序。一般情况下，这个程序是必须要有，而且必须要写在 main.lua 里。
输入参数	无
返回值	无
举例	详细请参见主架构 run()详解

8.2 ril 模块详解

- 发送 AT 命令

程序名	request(cmd,arg,onrsp,delay)	
功能	发送 AT 命令。其实是将发送的 AT 命令插入到 AT 命令队列中。队列中的 AT 命令会按 FIFO 的顺序及时执行	
输入参数	cmd	需要发送的 AT 命令，字符串型
	arg	跟该命令相关的参数，字符串型
	onrsp	对该命令的回复结果进行处理的回调函数，function 型。用 regrsp 也可以注册 AT 命令的回复结果进行处理的回调函数。onrsp 的优先级高于 regrsp 注册的回调函数，前者存在时，后者不起作用。onrsp 用于对回复结果做特殊处理。
	delay	延时 delay 毫秒后，才发送此 AT 命令
返回值	无	
举例	发送数据的函数需要调用到 ril.lua 中的 request 函数： local ril = require"ril" local req = ril.request function send(id,data) req(string.format("AT+CIPSEND=%d,%d",id,string.len(data)),data) return true end	

- 注册 urc 处理程序

程序名	regurc(prefix,handler)	
功能	注册某个 urc 的处理函数	
输入参数	prefix	urc 前缀，最前面的连续字符串，包含+、大写字符、数字的组合，比如”+CREG”，字符串型
	handler	URC 处理程序，function 型
返回值	无	
举例	详细请参见 regurc()用来注册某些 URC 相应的处理程序	

- 解注册 urc 处理程序

程序名	deregurc(prefix)	
功能	解注册某个 urc 的处理函数，将 URC 处理程序从 urctable 表解注册	
输入参数	prefix	同 regurc
返回值	无	

- 注册 rsp 程序
  (rsp 是对 AT 命令响应的处理程序)

程序名	regrsp(head,fnc,typ)	
功能	注册某个 AT 命令应答的处理函数	
输入参数	head	此应答对应的 AT 命令头，去掉了最前面的 AT 两个字符。字符串型。
	fnc	对 AT 命令返回结果的处理程序
	typ	AT 命令的应答类型，取值范围为： 0-NORESULT,1-NUMBERIC,2-SLINE,3-MLINE,4-STRING,10-SPECIAL
	formt	typ 为 STRING 时，进一步定义 STRING 中的详细格式
返回值	无	
举例	详细请参见 regrsp()用来注册 AT 命令处理程序	

8.3 link 模块详解

- 关闭 IP 网络

程序名	shut()
功能	关闭 IP 网络
输入参数	无
返回值	无

8.4 socket 模块详解

- 创建 socket 并且连接服务器

程序名	connect(idx,prot,addr,port,rsp,rcv,cause)	
功能	创建 socket（如果 socket 不存在），并且连接服务器	
输入参数	idx	number 类型，socket id，如果使用了 mqtt 模块，取值范围是 1、2、3；如果没使用 mqtt 模块，取值范围是 1、2、3、4、5。[必选]
	prot	string 类型，传输层协议，目前仅支持"TCP"和"UDP"
	addr	string 类型，服务器地址，支持 IP 地址和域名
	port	number 类型，服务器端口号
	rsp	function 类型，socket 的状态处理函数
	rcv	function 类型，socket 的数据接收处理函数
	cause	暂时无用，后续扩展使用
返回值	true 表示成功调用了连接接口(连接结果会有异步消息通知到 socket 状态处理函数中)，false 表示没有成功调用连接接口	

- 断开一个 socket 连接

程序名	disconnect(idx,cause)	
功能	断开一个 socket 连接	
输入参数	idx	number 类型，socket id，如果使用了 mqtt 模块，取值范围是 1、2、3；如果没使用 mqtt 模块，取值范围是 1、2、3、4、5。[必选]

	cause	暂时无用，后续扩展使用
返回值	true 表示成功调用了断开接口（断开结果会有异步消息通知到 socket 的状态处理函数中），false 表示没有成功调用断开接口	

- 发送数据

程序名	send(idx,data,para,pos,ins)	
功能	发送数据	
输入参数	idx	number 类型，socket id，如果使用了 mqtt 模块，取值范围是 1、2、3；如果没使用 mqtt 模块，取值范围是 1、2、3、4、5。[必选]
	data	要发送的数据
	para	发送的参数
	pos	暂时无用，后续扩展使用
	ins	暂时无用，后续扩展使用
返回值	true 表示成功调用了发送接口(发送结果会有异步消息通知到 socket 状态处理函数中)，false 表示没有成功调用发送接口	

8.5 sms 模块详解

我司提供的 sms.lua 模块，短信缺省格式为 TEXT 型，编码格式是 UCS2，使用默认的存储方式（一般默认为 SIM）。

- 获取短信初始化状态

程序名	getsmstate()
功能	获取短信初始化状态
输入参数	无
返回值	boolean 型 true：初始化完成；false：初始化未完成

- 发送短信

程序名	send(num,data,cb,idx)	
功能	发送短信	
输入参数	num	发送短信的目的号码，字符串型
	data	短信内容， GB2312 编码的字符串
	cb	短信发送结果异步返回时使用的回调函数，可选
	idx	插入短信发送缓冲表的位置，可选，默认是插入末尾
返回值	boolean 型，true 表示调用接口成功（并不是短信发送成功，短信发送结果，通过 sendcnf 返回，如果有 cb，会通知 cb 函数）；或 返回 false，表示调用接口失败	

- 注册新短信的用户处理函数

程序名	regnewsmscb(cb)	
功能	注册新短信的用户处理函数	
输入参数	cb	用户处理函数名
返回值	无	

8.6 cc 模块详解

- 拨打语音电话

程序名	dial(number,delay)	
功能	拨打语音电话	
输入参数	number	拨打的电话号码，字符串型
	delay	延时 delay 毫秒后，才发送 at 命令呼叫，默认不延时，即此参数暂时未启用
返回值	返回 boolean 型，true（已经拨出电话）或 false（未能拨出电话）	

- 挂断电话

程序名	hangup()
功能	主动挂断所有当前语音电话
输入参数	无
返回值	无

- 接听电话

程序名	accept()
功能	接听当前打入的语音电话
输入参数	无
返回值	无

- 注册一个或者多个消息的用户回调

程序名	regcb(evt1,cb1,...)	
功能	注册一个或者多个消息的用户回调函数	
输入参数	evt1	消息类型，目前仅支持: "READY","INCOMING","CONNECTED","DISCONNECTED"
	cb1	消息对应的用户回调函数
	...	evt 和 cb 成对出现
返回值	无	

8.7 net 模块详解

- 获取当前网络注册状态

程序名	getstate()
功能	获取当前 GSM 网络注册状态
输入参数	无
返回值	网络注册状态，字符串型： INIT(刚开机的状态) UNREGISTER（没有注册网络） REGISTERED（已经注册了网络）

- 获取当前小区的 mcc

程序名	getmcc()
功能	获取当前小区的 mcc（移动国家代码）
输入参数	无
返回值	当前小区的 mcc，如果还没有注册 GSM 网络，则返回 sim 卡的 mcc

- 获取当前小区的 mnc

程序名	getmnc()
功能	获取当前小区的 mnc（移动网络代码）
输入参数	无
返回值	当前小区的 mnc，如果还没有注册 GSM 网络，则返回 sim 卡的 mnc

- 获取当前小区的位置区号 LAC

程序名	getlac()
功能	获取当前 GSM 网络当前的位置区号
输入参数	无
返回值	当前位置区 LAC，16 进制字符串，例如"18be"，如果还没有注册 GSM 网络，则返回""

- 获取当前小区号 CI

程序名	getci()
功能	获取当前 GSM 网络当前的小区号
输入参数	无
返回值	小区号 CI， 16 进制字符串，例如"93e1"，如果还没有注册 GSM 网络，则返回""

- 获取当前 rssi 值

程序名	getrssi()
功能	获取当前 GSM 网络接收信号强度
输入参数	无
返回值	rssi 值，0~31

- 获取 TA 值

程序名	getta()	
功能	获取 TA 值	
输入参数	无	
返回值	TA 值	

- 周期性查询信号强度

程序名	startcsqtimer()	
功能	启动“信号强度查询”定时器 这个启动是有选择性的，不是飞行模式 并且打开了查询开关 或者 工作模式为完整模式	
输入参数	无	
返回值	无	

- 设置信号强度查询周期

程序名	setcsqqueryperiod(period)	
功能	设置信号强度查询周期，并开始周期性信号查询	
输入参数	period	信号强度查询周期，整数型，单位为毫秒
返回值	无	

8.8 pm 模块详解

- 唤醒模块

程序名	wake(tag)	
功能	将模块唤醒	
输入参数	tag	module 名，字符串型。tag 可以是 lib 中的库文件，比如 cc，net 等，也可以是用户自己编写的应用 module。引入 tag 的目的是只要存在任何一个模块唤醒，则整个模块都不睡眠。
返回值	无	

- 将模块休眠

程序名	sleep(tag)	
功能	将模块休眠	
输入参数	tag	同 wake(tag)
返回值	无	

8.9 common 模块详解

- 将 UCS2 码转成 ASCII 码

程序名	ucs2toascii(inum)
功能	将 UCS2 码字符串转成 ASCII 码字符串，例如"0031003200330034" -> "1234"

输入参数	inum	待转换字符串
返回值	转换后的字符串	

- 将号码（ASCII 字符串）转换成 UCS2 码字符串

程序名	nstrToUcs2Hex(inum)	
功能	将号码（ASCII 字符串）转换成 UCS2 码 16 进制字符串，仅支持数字和+	
输入参数	inum	待转换字符串
返回值	字符串型	
举例	“+1234” -> “002B0031003200330034”	

- ASCII 字符串 转化为 BCD 编码

程序名	numtobcdnum(num)	
功能	号码 ASCII 字符串 转化为 BCD 编码格式字符串，仅支持数字和+	
输入参数	num	待转换字符串
返回值	字符串型，转换后的 BDC 码	
举例	"+8618126324567" -> 91688121364265f7 （表示第 1 个字节是 0x91，第 2 个字节为 0x68，……）	

- BCD 编码转化为 ASCII 字符串

程序名	bcdnumtonum(num)	
功能	BCD 编码格式字符串转化为 ASCII 字符串 ，仅支持数字和+	
输入参数	num	待转换字符串
返回值	字符串型，转换后的 BDC 码	
举例	91688121364265f7 （表示第 1 个字节是 0x91，第 2 个字节为 0x68，……） -> "+8618126324567"	

- 十六进制串转成可见字符串

程序名	binstohexs(bins,s)	
功能	十六进制串转成可见字符串	
输入参数	bins	十六进制串
	s	转换后，每两个字节之间的分隔符，默认没有分隔符
返回值	可见的字符串	
举例	（122B5699）hex –> “122B5699”	

- 十六进制可见字符串转换为 ASCII 字符串

程序名	hexstobins(hexs)	
功能	十六进制可见字符串转换为 ASCII 字符串	
输入参数	hexs	十六进制可见字符串，字符串型
返回值	ASCII 字符串，字符串型	

举例	"2B3132" -> “+12”

- UCS2 编码的字符串转换为 GB2312 编码的字符串

程序名	ucs2togb2312(ucs2s)	
功能	UCS2 编码的小端编码字符串转换为 GB2312 编码的字符串	
输入参数	ucs2s	UCS2 小端编码的字符串，字符串型
返回值	GB2312 编码的字符串，字符串型	

- GB2312 编码的字符串转换为 UCS2 编码的字符串

程序名	gb2312toucs2(gb2312s)	
功能	GB2312 编码的字符串转换为 UCS2 小端编码的字符串	
输入参数	gb2312s	GB2312 编码的字符串，字符串型
返回值	UCS2 小端编码的字符串，字符串型	

- UCS2BE(大端在前的 UCS2)编码的字符串转换为 GB2312 编码的字符串

程序名	ucs2betogb2312(ucs2s)	
功能	UCS2BE 大端编码的字符串转换为 GB2312 编码的字符串	
输入参数	ucs2s	UCS2BE 编码的字符串，字符串型
返回值	GB2312 编码的字符串，字符串型	

- GB2312 编码的字符串转换为 UCS2BE(大端在前的 UCS2)编码的字符串

程序名	gb2312toucs2be(gb2312s)	
功能	GB2312 编码的字符串转换为 UCS2BE(大端在前的 UCS2)编码的字符串	
输入参数	b2312s	GB2312 编码的字符串，字符串型
返回值	UCS2BE 编码的字符串，字符串型	

8.10 pb 模块详解

- 查找电话本的一笔记录

程序名	find(name)	
功能	查找电话本的一笔记录	
输入参数	name	姓名，字符串型
返回值	boolean 型 true：表示查找这个动作已经发起 false：表示查找这个动作未成功发起	

- 读取电话本的一笔记录

程序名	read(index)	
功能	读取电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
返回值	boolean 型 true：表示读取这个动作已经发起 false：表示读取这个动作未成功发起	

- 写电话本的一笔记录

程序名	writeitem(index,name,num)	
功能	写入电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
	name	姓名，字符串型
	num	号码，字符串型
返回值	boolean 型 true：表示写入一笔电话本记录这个动作已经发起 false：表示写入这个动作未成功发起	

- 删除电话本的一笔记录

程序名	deleteitem(index)	
功能	删除电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
返回值	boolean 型 true：表示删除一笔电话本记录这个动作已经发起 false：表示删除这个动作未成功发起	

8.11 audio 模块详解

- 播放声音文件

程序名	play(name)	
功能	播放声音文件	
输入参数	name	声音文件名称。 声音文件由 luaDB 下载工具缺省下载到模块的/ldata 目录中，调用的时候形式如： “/ldata/filename” filename 是以.amr 或 .mp3 或 .midi 为后缀的音频文件的名字
返回值	无	

- 停止播放声音文件

程序名	stop(priority,typ,path,vol,cb,dup,duprd)	
功能	停止播放当前正在播放的声音文件	
输入参数	priority	number 类型，必选参数，音频优先级，数值越大，优先级越高
	typ	string 类型，必选参数，音频类型，目前仅支持"FILE"、"TTS"、"RECORD"
	path	必选参数，音频文件路径，跟 typ 有关： typ 为"FILE"时：string 类型，表示音频文件路径 typ 为"TTS"时：string 类型，表示要播放数据的 UCS2 十六进制字符串 typ 为"RECORD"时：number 类型，表示录音 ID
	vol	number 类型，可选参数，播放音量，取值范围 audiocore.VOL0 到 audiocore.VOL7
	cb	function 类型，可选参数，音频播放结束或者出错时的回调函数，回调时包含一个参数：0 表示播放成功结束；1 表示播放出错；2 表示播放优先级不够，没有播放
	dup	bool 类型，可选参数，是否循环播放，true 循环，false 或者 nil 不循环
	duprd	number 类型，可选参数，播放间隔(单位毫秒)，dup 为 true 时，此值才有意义
返回值	调用成功返回 true，否则返回 nil	

- 设置听筒或喇叭的音量

程序名	setspeakervol(vol)	
功能	设置听筒或喇叭的音量	
输入参数	vol	音量等级，取值范围为 audio.VOL0 到 audio.VOL7，audio.VOL0 为静音
返回值	无	

- 查询听筒或喇叭的音量

程序名	getspeakervol()
功能	查询听筒或喇叭的音量
输入参数	无
返回值	音量等级，整数型，同 setspeakervol 中 vol 定义

- 设置 mic 音量

程序名	setmicrophonegain(vol)	
功能	设置 mic 音量	
	vol	mic 音量，整数型，取值范围为 audio.MIC_VOL0 到 audio.MIC_VOL15，audio.MIC_VOL0 为静音
返回值	无	

- 查询 mic 音量

程序名	getmicrophonegain()
功能	查询 mic 音量
输入参数	无
返回值	mic 音量，整数型，取值同 setmicrophonegain 中 vol

- 停止音频播放

程序名	stop()
功能	停止音频播放
输入参数	无
返回值	如果可以成功同步停止，返回 true，否则返回 nil

8.12 misc 模块详解

- 设置当前时钟

程序名	setclock(t,rspfunc)	
功能	设置系统时间	
输入参数	t	是一个表，键值有 year,month,day,hour,min,sec， 取值均为字符型，其中 year 是 4 位字符的字符串，其余是 1~2 位字符的字符串 格式举例： {year=2017,month=2,day=14,hour=14,min=2,sec=58}
	rspfunc	设置时钟时所开启的功能，function 型。如果不用，该参数可以为空
返回值	无	

- 获取当前时间串

程序名	getclockstr()
功能	获取系统时间字符串
输入参数	无
返回值	当前时间，字符串型。形式如：”YYMMDDHHMMSS”,年月日时分秒均为 2 位字符
举例	获取时间为：170214141602，表示：17 年 2 月 14 日 14 时 16 分 02 秒

- 判断当前时间是星期几

程序名	getweek()
功能	判断当前是星期几
输入参数	无
返回值	星期几，number 型。1-7 分别对应周一到周日。

- 获取当前日期和时间

程序名	getclock()
功能	获取当前日期和时间
输入参数	无
返回值	当前日期和时间，以表的形式，该表键值有 year,month,day,hour,min,sec， 取值均为字符型，其中 year 是 4 位字符的字符串，其余是 1~2 位字符的字符串
举例	返回值例如：{year=2017,month=2,day=14,hour=14,min=19,sec=23}

- 设置模块 IMEI

程序名	setimei(s,cb)	
功能	设置模块 IMEI 如果传入了 cb，则设置 IMEI 后不会自动重启，用户必须自己保证设置成功后，调用 sys.restart 或者 dbg.restart 接口进行软重启; 如果没有传入 cb，则设置成功后软件会自动重启	
输入参数	s	新 IMEI
	cb	设置 IMEI 后调用的回调函数，调用时会将设置 IMEI 的结果传出去，true 表示设置成功，false 或者 nil 表示失败。 例如:设置 IMEI 成功后，调用 cb 的形式就是 cb(true)；设置 IMEI 失败后，调用 cb 的形式就是 cb(false)
返回值	无	

- 获取模块 IMEI 号码

程序名	getimei()
功能	获取模块 IMEI 号码
输入参数	无
返回值	模块 IMEI 号，字符串型。如果未获取到返回""

- 获取 VBAT 的电池电压

程序名	getvbatvolt()
功能	获取 VBAT 的电池电压
输入参数	无
返回值	VBAT 的电池电压。number 类型，单位毫伏

8.13 gps 模块详解

- 初始化 gps

程序名	init(ionum,dir,edge,period,id,baud,databits,parity,stopbits,apgspwronupd)
功能	初始化 gps

输入参数	ionum	GPS 供电的 GPIO 口的端口号。 pio.P0_0 - pio.P0_31 表示 GPIO_0 - GPIO_31 pio.P1_0 - pio.P1_9 表示 GPO_0 - GPO_9
	dir	此参数没用（为了兼容之前的代码，不能去掉），随便写
	edge	boolean 型，true 表示 GPIO 输出高电平供电，false 或者 nil 表示 GPIO 输出低电平供电
	period	读 gps 通讯端口的周期。整数型，单位是毫秒 gps 通讯端口目前只支持 uart 口 建议 1000 毫秒读取一次
	id	gps uart 口的端口号，number 型。1 表示串口 1,2 表示串口 2
	baud	gps uart 口的波特率，number 型
	databits	gps uart 口的数据位，number 型
	parity	gps uart 口的校验位，number 型，有三种取值： uart.PAR_EVEN, uart.PAR_ODD 或 uart.PAR_NONE
	stopbits	gps uart 口的停止位，整数型，有三种取值： uart.STOP_1 (for 1 stop bit), uart.STOP_1_5 (for 1.5 stop bits) 或 uart.STOP_2 (for 2 stop bits)
	apgspwronupd	是否允许开机就执行 AGPS 功能
返回值	无	

- 打开一个 gps”应用”

程序名	open(mode,para)	
功能	打开一个 gps”应用”	
输入参数	mode	GPS 工作模式
	para	para.cause：“GPS 应用”标记 para.val：GPS 开启最大时长 para.cb：回调函数
返回值	无	

- 关闭一个“GPS 应用”

程序名	close(mode,para)	
功能	关闭一个“GPS 应用”	
输入参数	mode	GPS 工作模式
	para	para.cause：“GPS 应用”标记 para.val：GPS 开启最大时长 para.cb：回调函数
返回值	无	

- 判断一个“GPS 应用”是否处于激活状态

程序名	isactive(mode,para)
功能	判断一个“GPS 应用”是否处于激活状态

输入参数	mode	GPS 工作模式
	para	para.cause：“GPS 应用”标记 para.val：GPS 开启最大时长 para.cb：回调函数
返回值	无	

- 获取终端经纬度位置信息

程序名	getgpslocation(format)	
功能	获取终端经纬度位置信息	
输入参数	format	经纬度的类型，整数型。目前支持 2 个取值： GPS_DEGREES：度。例如：纬度为 XX.YYYYY，则为 XX.YYYYY 度 GPS_DEGREES_MINUTES：度分。例如纬度为 XX.YYYYY，则 XX 为度，YY.YYY 为分。 默认为度
返回值	位置信息。字符串。格式如：E/W,long,N/S,lati E/W:东经或西经 long：经度 N/S:北纬或南纬 lati：纬度	
举例	返回值举例：“E,121.5259850,N,31.2356616” 如果没有经纬度格式为"E,,N,"	

- 获取终端速度

程序名	getgpsspd()
功能	获取由 gps 侦测到的终端速度
输入参数	无
返回值	终端速度，number 型

- 获取终端方位角

程序名	getgpscog()
功能	获取由 gps 侦测到的终端方位角
输入参数	无
返回值	终端速度，number 型。取值：0~359.9

- 获取高度

程序名	getaltitude()
功能	获取高度
输入参数	无
返回值	高度

- 判断 gps 是否已经打开

程序名	isopen()
功能	判断 gps 是否已经打开
输入参数	无
返回值	Boolean 型 true：表示 gps 已经打开 false：表示 gps 未打开

- 判断终端 gps 是否已经成功定位

程序名	isfix()
功能	判断终端 gps 是否已经成功定位
输入参数	无
返回值	Boolean 型 true：表示 gps 已经成功定位 false：表示 gps 未定位

8.14 mqtt 模块详解

- 创建一个 mqtt client

程序名	create(prot,host,port)	
功能	创建一个 mqtt client	
输入参数	prot	string 类型，传输层协议，仅支持"TCP"和"UDP"[必选]
	host	string 类型，服务器地址，支持域名和 IP 地址[必选]
	port	number 类型，服务器端口[必选]
返回值	无	

- 配置遗嘱参数

程序名	tmqtt:configwill(flg,qos,retain,topic,payload)	
功能	配置遗嘱参数	
输入参数	flg	number 类型，遗嘱标志，仅支持 0 和 1
	qos	number 类型，服务器端发布遗嘱消息的服务质量等级，仅支持 0,1,2
	retain	number 类型，遗嘱保留标志，仅支持 0 和 1
	topic	string 类型，服务器端发布遗嘱消息的主题，仅支持 ascii 字符
	payload	string 类型，服务器端发布遗嘱消息的载荷，仅支持 ascii 字符
返回值	无	

- 连接 mqtt 服务器

程序名	tmqtt:connect(clientid,keepalive,user,password,connectedcb,connecterrcb)	
功能	连接 mqtt 服务器	
输入参数	clientid	string 类型，client identifier，仅支持 ascii 字符[必选]
	keepalive	number 类型，保活时间，单位秒[可选，默认 600]
	user	string 类型，用户名，仅支持 ascii 字符[可选，默认""]

	password	string 类型，密码，仅支持 ascii 字符[可选，默认""]
	connectcb	function 类型，连接成功的回调函数[可选]
	connecterrcb	function 类型，连接失败的回调函数[可选]
返回值	无	

- 发布一条消息

程序名	tmqtt:publish(topic,payload,qos,ackcb,usertag)	
功能	发布一条消息	
输入参数	topic	string 类型，消息主题，仅支持 ascii 字符[必选]
	payload	二进制数据，消息负载[必选]
	qos	number 类型，服务质量等级，仅支持 0 和 1[可选，默认 0]
	ackcb	function 类型，qos 为 1 时表示收到 PUBACK 的回调函数，qos 为 0 时消息发送结果的回调函数[可选]
	usertag	string 类型，用户回调函数 ackcb 用到的第一个参数[可选]
返回值	无	

- 订阅主题

程序名	tmqtt:subscribe(topics,ackcb,usertag)	
功能	订阅主题	
输入参数	topics	table 类型，一个或者多个主题，主题名仅支持 ascii 字符，质量等级仅支持 0 和 1，形式如： {{topic="/topic1",qos=质量等级}, {topic="/topic2",qos=质量等级}, ...}[必选]
	ackcb	function 类型，表示收到 SUBACK 的回调函数[可选]
	usertag	string 类型，用户回调函数 ackcb 用到的第一个参数[可选]
返回值	无	

- 注册事件的回调函数

程序名	tmqtt:regevtcb(evtcbs)	
功能	注册事件的回调函数	
输入参数	evtcbs	一对或者多对 evt 和 cb，格式为{evt=cb,...}，evt 取值如下： "MESSAGE"：表示从服务器收到消息，调用 cb 时，格式为 cb(topic,payload,qos)
返回值	无	

9. 消息分发函数 dispatch 详细介绍

在程序任何地方 dispatch 出来的消息和参数，会先放入内部消息队列 qmsg 中，然后通过 sys.run()中的 runqmsg()函数自动在 apps 表中查找并执行对应的回调处理函数（在本文前面的章节中已经提到过，对应的回调处理函数是通过 regapp 注册的，注册形式有 table 和 function 两种）。

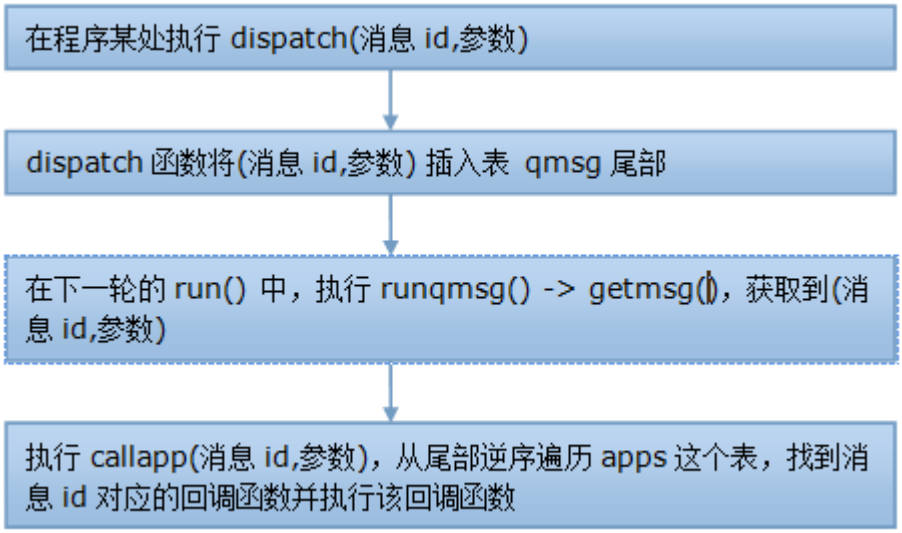


图 77　　dispatch 过程流程图

有一点需要注意，在应用程序中，同一个消息 id 可以对应不同的处理回调函数，比如需要广播的一些消息。这种情况下如果要把所有的回调函数都能执行到，需要在所有回调函数中需要返回的地方都要 return true，因为在任意回调函数中遇到 return false 的情况，就会退出查找并执行所有回调函数这个过程，导致其余的回调执行不到。

在本章中将着重介绍 lib 库各个 module 中 dispatch 出来的消息和相关的参数，以方便客户的使用。

9.1　cc 模块中的 dispatch

dispatch("CALL_CONNECTED")	当 Lua 从 ATC 口收到 CONNECT 这个 URC 时，此时会 dispatch 这个消息出来，表示通话接通
dispatch("CALL_READY")	当 Lua 从 ATC 口收到 CALL READY 这个 URC 时，此时会 dispatch 这个消息出来，表示已经注册网络，可以发起电话呼叫了
dispatch("CALL_DISCONNECTED",reason)	当电话被挂断时（包括主动挂断和对方挂断），或无法接通时，会 dispatch 这样一个消息 消息 id：CALL_DISCONNECTED； 参数 reason：挂断原因，string 型
dispatch("CALL_INCOMING",number)	当有电话呼入时，会 dispatch 这个消息出来告知，并把呼入号码一同告知

	消息 id：CALL_INCOMING 参数 number：呼入号码，string 型

9.2 net 模块中的 dispatch

dispatch("NET_STATE_CHANGED",s)	当网络注册状态发生改变时，会 dispatch 这个消息出来告知，并将网络状态一同告知 消息 id：NET_STATE_CHANGED 参数 s：网络当前的状态，string 型（INIT/EGISTERED/UNREGISTER）
dispatch("GSM_SIGNAL_REPORT_IND", success,rssi)	当收到信号查询命令 AT+CSQ 的返回结果时，会 dispatch 此消息告知接收信号强度。 rssi：接收信号强度，number 型

9.3 sms 模块中的 dispatch

dispatch("SMS_READ_CNF",success,num, data,pos,t,name)	得到读一条短信 AT+CMGR=pos 这个命令的返回结果时会 dispatch 这个消息，告知读短信的结果。 success：查询这个动作是否成功，true/false，boolean 型 num：发件人号码，number 型 data：短信内容，string 型 pos：短信在存储空间的位置，string 型 name：发件人姓名，string 型
dispatch("SMS_DELETE_CNF",success)	得到删除一条短信 AT+CMGD=pos 这条 AT 命令的返回结果后，会 dispatch 这个消息出来，告知删除短信是否成功。 success：删除短信这个动作是否成功，true/false，boolean 型
dispatch("SMS_SEND_CNF",success)	用 AT+CMGS 这个 AT 命令发送完一条短信，并得到该命令的返回消息后，会 dispatch 这个消息，告知短信发送情况 success：发送短信这个动作是否成功，true/false，boolean 型
dispatch("SMS_READY")	收到 SMS READY 这个 URC 后，会 dispatch 这个消息出来，告知短信初始化已经完成
dispatch("SMS_NEW_MSG_IND",pos)	收到+CMTI：pos 这个 URC 后，会 dispatch 这个消息出来，告知收到了一条新短信 pos：短信在存储空间的位置，string 型

9.4 pb 模块中的 dispatch

dispatch("PB_FIND_CNF",success,index,n,name)	当收到查找电话本记录 AT+CPBF 这个 AT 命令的返回结果时会 dispatch 这么一条消息，告知查询结果。 success：查找电话本记录这个动作是否成功，true/false，boolean 型 index：查到的电话本记录的索引，string 型 n：电话号码，string 型 name：名字，string 型
dispatch("PB_READ_CNF",success,index,n,name)	当收到读取一条电话本记录 AT+CPBR 这个 AT 命令的返回结果时会 dispatch 这么一条消息，告知读取结果。 success：读取电话本记录这个动作是否成功，true/false，boolean 型 index：查到的电话本记录的索引，string 型 n：电话号码，string 型 name：名字，string 型
dispatch("CPBS_READ_CNF",success,storage,used,total)	当收到查询电话本存储类型 AT+CPBS?这个 AT 命令的返回结果时，会 dispatch 这样一条消息，以告知查询结果 success：查询这个动作是否成功，true/false，boolean 型 storage：存储类型（SM/ME/VM/ON 等），string 型 used：已用的记录个数，number 型 total：总共可存储的记录个数，number 型

9.5 audio 模块中的 dispatch

dispatch("AUDIO_DTMF_DETECT",dtmf)	当收到+DTMFDET:这个 URC，会 dispatch 这个消息出来，以告知检测到 DTMF 或单频音，并将 DTMF 或单频音的值一并告知 dtmf：DTMF 或单频音，string 型，取值范围：0~9,A~D,*,#,1000Hz 单频音,1400Hz 单频音,2300Hz 单频音
dispatch("AUDIO_PLAY_END_IND")	当用 audio.play 播放完音频文件时，会 dispatch 这个消息以告知成功播放完毕
dispatch("AUDIO_PLAY_ERROR_IND")	当用 audio.play 播放完音频文件时，会 dispatch 这个消息以告知播放失败
dispatch("SPEAKER_VOLUME_SET_CNF",success)	用 AT+CLVL 设置下行音频通道（SPEAKER/RECEIVER）的音量并得到返回结果

	时，会 dispatch 这个消息出来以告知设置成功或失败 success：设置音量这个动作是否成功，true/false，boolean 型
dispatch("AUDIO_CHANNEL_SET_CNF", success)	用 AT+CHFA 设置音频通道并得到返回结果时，会 dispatch 这个消息出来以告知设置成功或失败 success：设置音频通道这个动作是否成功，true/false，boolean 型
dispatch("MICROPHONE_GAIN_SET_CNF",success)	用 AT+CMIC 设置上行通道（MIC）的增益并得到返回结果时，会 dispatch 这个消息出来以告知设置成功或失败 success：设置 MIC 音量这个动作是否成功，true/false，boolean 型

9.6 update 模块中的 dispatch

dispatch("UP_EVT","UP_PROGRESS_IND",packid *100/total)	
dispatch("UP_EVT","UP_END_IND",succ)	
dispatch("UP_EVT","NEW_VER_IND",upselcb)	

9.7 misc 模块中的 dispatch

sys.dispatch("SIM_IND","RDY")	当收到+CPIN: READY 这个 URC 时，会 dispatch 这个消息出来以告知 SIM 处于在位状态
sys.dispatch("SIM_IND","NIST")	当收到+CPIN: NOT INSERTED 这个 URC 时，会 dispatch 这个消息出来以告知 SIM 处于不在位状态

10. 快速入手

本章从一个常用的例子入手，简要介绍根据框架结构和库文件编写应用程序的方法。

举一个例子：开机后检验网络注册状态，如果注册了网络就发起一个语音呼叫，接通后挂断呼叫

第 1 步：编写 Lua 脚本

Luat 电话功能脚本示例

第 2 步：用脚本下载工具将脚本下载到模块中

点此下载脚本下载工具 ，下载完成后将开发板或模块重新上电。

第 3 步：查看脚本 trace
借助 trace 工具查看脚本中的打印语句输出的内容，可以对脚本进行查错，并以此为根据修改代码。

11. 例程详解

点开以下帖子，查看各种例程：
Luat 如何用 Lua 脚本实现各种功能（文字版）

Luat 如何用 Lua 脚本实现各种功能（视频版）

12. 程序调试环境

Air200 lua 调试环境包括真实调试环境和模拟调试环境 2 种。

12.1 真实调试环境

真实调试环境，请参考本文中 合宙 Lua 项目开发和调试的一般步骤 。

12.2 PC 模拟调试环境

PC 模拟调试环境由我司开发提供，使用方法请参考 luat 模拟器。