360 智能路由器插件开发指南

版本 1.0_20160711

目录

● 概述

本文详细描述了基于 360 OS 的设备端插件程序开发方法。

开发者需要具备 TCP/IP 网络协议方面的知识、Linux 环境下 C 语言插件程序开发经验、熟悉 JSON 数据封装格式、熟悉 HTML/JavaScript。

名词解释：

JSON：JavaScript Object Notation

CGI：Common Gateway Interface

API：Application Programming Interface

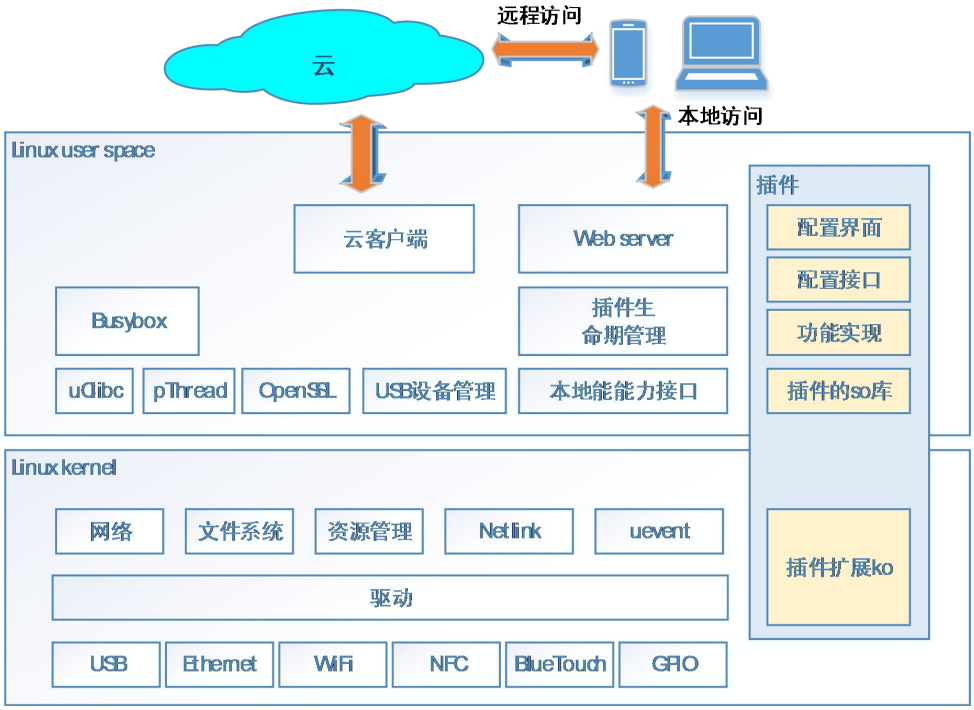
插件：运行于 360 智能路由器中的插件程序

1· 插件与 Linux 插件的关系

除支持 Linux 方式的插件开发外，还提供了更多的针对网络方面的 API。分为以下类别

类别	说明
协议控制	http 访问控制、DNS 访问控制
故障处理	在设备异常情况发生时做出处理
网络相关	配置路由、QoS、限速等网络功能
通知	用于通知系统中随时发生的事件，例如新的存储设备装载了，网络用户下线消息通知
无线	控制无线接口工作模式，获取周围无线主机信息等
组操作	时间组、网址组、用户组操作
存储相关	与存储相关的 API，例如获取系统主分区信息，遍历系统中的所有存储设备等
设备信息	获取设备相关信息，例如设备绑定用户清单，设备的 ID 号等

2·插件架构



360 OS 中插件在系统中的位置

上图中，右侧“插件”框内的部份是网关插件的部分。其中“功能实现”是必须的。

若要开发具有网关本地 web 配置界面的插件，必须具有配置界面、配置接口、功能实现部分；

若仅需要云端 web 配置功能，配置界面仍然需要开发，但它安装在云端，网关中的插件需要配置接口、功能实现部分；

如果插件的所有功能仅通过专用的插件进行配置（不使用 web 技术），例如仅使用手机 APP 进行控制，则无需具有配置界面部分，需要具有配置接口和功能实现部分。

插件自有 so 动态链接库和内核 ko 模块部分是可选的。支持插件自带 so 库和 ko 模块。注意：360 安全路由器 mini 不支持内核模块 ko 文件。

3·插件的启动和退出

插件必须在进程的主线程中申请 360 OS 资源，即调用以 register 为前缀的 API，在插件程序工作期间，主线程保持存在。为避免资源泄漏，插件的主线程结束时，需要释放申请的 360 OS 资源。

插件程序通过注册 SIGTERM 的信号处理函数，如：signal(SIGTERM, sig_func),在 sig_func 函数中通过调用 unregister 为前缀的 API 去释放申请的资源。

4·插件程序配置方式

插件通过 web 方式进行配置。包括插件的启动、停止，获取插件的状态、配置插件参数等。有三种配置方式：本地配置方式、插件中心配置方式，以及两者都支持的配置方式。

360 OS 设计时已确保这两种开发方式的一致性。

4.1 本地配置方式

为支持该配置方式，设备上需要有插件的前端配置网页和 CGI 程序，插件安装后，在路由器配置页面中的“第三方扩展工具”菜单下，将会出现新插件的图标，点击后跳转到插件的配置页面。

在插件开发中，开发者会首先使用该方式来完成插件开发和调试。如果希望让更多的用户使用你的插件，那么需要使用下一种方式：把你的插件提交到插件中心审核。

4.2 插件中心配置方式

该配置方式只会用到设备端的 CGI，前端网页安装在插件中心服务器中。通过插件中心 web 版或移动终端插件来配置插件。

注意，只有通过插件中心发布的插件才能通过插件中心方式进行配置。

5·安装开发工具

访问 360 智能路由器官方网站：luyouw.360.cn 下载开发工具，在 Linux 主机上解包

/opt/tar jxvf srouter_P0_P1_XXXX.tar.bz2　XXXX 为版本号

解包后，生成以下目录

./srouter_ P0_P1_XXXX/include　　header 文件目录

./srouter_ P0_P1_XXXX/linux　　内核头文件，用于开发内核驱动

./srouter_ P0_P1_XXXX/lib　　库目录

./srouter_ P0_P1_XXXX/utility　　工具目录

./srouter_ P0_P1_XXXX/doc　　文档目录，包括本开发指南

./srouter_ P0_P1_XXXX/example　　例程

./srouter_ P0_P1_XXXX/toolchain　　交叉编译工具目录

./srouter_ P0_P1_XXXX/env-rtk.sh　　环境配置脚本，360 安全路由 P1、mini 使用

source ./env-rtk.sh 方式运行

6·如何使用 API

- 本文介绍的 API 需要引用 srouter.h 头文件，连接时需要使用 libnorouter.so。
  链接参数：
  LDFLAGS=-L${SROUTER_PATH}/lib -lnorouter
  ${SROUTER_PATH}是 SDK 安装目录

  为了处理 HTTP 请求，CGI 程序还需要引用 cgi.h，连接时需要使用 libcgi.so。
  LDFLAGS=-L${SROUTER_PATH}/lib –lcgi –lnorouter

- 为解决新固件运行老版本插件的问题，在许多 C 结构的最前面增加了 uint32_t size_of_struct 或 SizeOfStruct 成员，在使用前需要用宏：NOS_STRUCT_INIT 来初始化它。

7·插件程序的目录结构

按以下目录结构组织你的插件：
./插件名称/app.json　插件的基本信息，见后面说明
./插件名称/APPSIGN.png、APPSIGN_b.png、APPSIGN_w.png
插件的图标文件，APPSIGN 是指插件的名称。
例如信号调节插件的图标文件：power_progress_b.png 为

是点开插件配置界面时的图标，分辨率为 75px * 75px

power_progress.png 为

在插件清单中看到的图标，分辨率为 64px * 64px

power_progress_w.png 为

当鼠标移到插件图标上时显示的图片，分辨率为 64px * 64px

./插件名称/config　　插件缺省参数文件，由插件定义内容
./插件名称/bin/　　此目录存放可执行程序、内核模块文件、动态链接库
./插件名称/webs/　　此目录存放 cgi 程序、html、css、javascript 等网页文件

插件程序打包后的扩展名为 opk，可以通过设备本地 web 配置页面安装它。

app.json 文件说明：

该文件是插件基本信息文件，JSON 格式，包括以下对象，插件程序需要填写每一个对象的值(注意值不可换行)：

"appsign": 插件名称，必须是独一无二的，不与其它插件同名，ASCII 字符

“appname”：插件显示名称，显示在手机和本地配置页面中的名称，UTF8 编码

"version": 插件版本号，三段点分十进制数。例如：“1.0.1”

“dep_version": 对基本系统的版本依赖要求，基本系统版本号必大于该版本才能安装

"description": 插件简介

"bin_start": 插件的启动程序文件名，该文件必须位于./bin 目录下

"author": 作者名称

"URL": 官方网站

"maintainer"：维护者

"email"：维护者的电邮

"icon"：插件图标，相对路径

"configtype"：配置类型，0:无需配置，1:原生配置界面(手机 APP 本地实现)，2:H5 配置，对于 H5 配置的插件，配置页路径为 http://<router_ip>/app/<appsign>/webs/index.html

"app_type": 插件程序类型，本地配置″local″，”remote”，”local&remote”

"rom_use": 插件程序需要占用的 rom 空间，单位 KB

"ram_use": 插件程序运行时需要的 ram 空间，单位 KB

8·插件程序的参数

插件程序通常需要临时保存和永久保存数据，SDK 提供了 get_tmp_path 和 get_config_path 两个 API 分别获取插件的临时数据保存路径和永久数据保存路径，

详见 API 部分“存储相关 API”章节。

临时数据是指网关重启后会被清除的数据；永久保存数据是指网关掉电后仍然持续保存的数据。

临时存储路径下不应该保存大量的数据，系统在内存紧张时，将清理临时存储目录，对占用较大临时存储空间的数据将被强制清除。插件的临时目录存储内容不应超过 100KB，需控制存储在临时目录中日志文件的大小。

插件开发者应使用 API 返回的路径来保存自己的数据，使用其它路径将不能通过插件中心审查，也不保证固件升级后仍能正常读写。

当用户通过恢复插件参数时，将删除临时和永久保存路径下的所有文件，并复制插件目录中的 config 文件到永久数据保存目录。

插件升级时，也不会删除临时和永久保存路径下的内容。

9·使用插件自己的动态链接库

当固件中没有你所需要的动态链接库时，可以将你需要的动态链接库文件复制到插件的 bin 目录中。在启动插件时，以及在 web server 执行 CGI 程序时，会将插件的 bin 目录加入到 LD_LIBRARY_PATH 环境变量中，让你的程序可以正确找到相应的动态链接库文件。

360 安全路由器 P1、5G 固件中包括以下动态链接库：

libpthread、libresolv、libm、libuClibc、libssl、libcrypt、libcrypto、libnl、libnsl、librt

360 安全路由器 mini 固件中包括以下动态链接库：

libpthread、libresolv、libm、libuClibc、libcrypt、libcrypto、libnl、libnsl、librt

固件中支持的动态链库可 telnet 到路由器上，查看/lib 目录中的文件。

10· 使用 C++开发插件

因不同的 360 安全路由器所具有的资源不同，360 安全路由器 P1、5G 固件中包含 C++库，360 安全路由器 mini 不包含 C++库。

11·将插件安装到路由器中

有两种插件程序安装方式：

- 通过设备的 web 配置页面安装

  插件通过编译打包后，就能通过设备本地配置页面安装。通过浏览器访问

luyou.360.cn（或 LAN 口 IP 地址）。首先找到“开发者模式”，将其中“第三方插件开关”置为 ON

再找到“第三方扩展工具”，点击“添加插件”，选择插件的 opk 文件，上传路由器后完成安装。

注：

请慎用，第三方插件调试功能为插件开发者专属功能

安装未经过官方审核的第三方插件可能会导致用户信息泄露，甚至可能出现路由器基本功能不稳定的现象，在安装过非官方审核的第三方插件后，本公司将不再对路由器进行保修。

在通电情况下，长按 Reset 键 3~5 秒，恢复出厂设置，可清空第三方插件。

- 通过插件中心安装

  提交插件中心，并通过审核的插件，可通过“360 路由器卫士”手机端中“扩展工具”界面，右上角“+”进入到插件中心。

● 插件 API

1· API 的返回值

API 调用出错时，将返回负值。错误码定义如下：

```
enum {
    ERR_FAILURE = -1,        /* 一般错误，如参数不合法 */
    ERR_GROUP = -2,          /* 用户组 ID 非法*/
    ERR_NO_RESOURCE = -3,    /* 资源不可用*/
    ERR_RULE_FULL = -4,      /* 规则已满*/
    ERR_NO_PERMISSION = -5, /* 无权限*/
    ERR_NO_MEM = -6,         /* 内存分配失败 */
    ERR_NAME_EXIST = - 7,    /* 名称已存在*/
    ERR_NON_EXIST = -8,      /* 规则不存在 */
    ERR_DATA_ERR = -9,       /* 参数错误 */
    ERR_EXIST = -10,         /* 规则已经存在 */
    ERR_NOT_SUPPORT = -11,   /* 不支持 */
};
```

2. 用户组

用户组是内网用户的集合，可以是 IP 地址集合或用户帐号集合。系统支持 256 个用户组，系统保留了前 50 个用户组，用于向插件提供预定义的用户分类，例如有线用户组、无线用户组，插件能引用这些组来进行规则设置。详见用户组 API 说明。

- **系统预留用户组**

系统预留 50 个用户组（ID 为 0 至 49），插件程序可以引用，但不能操作它们。在主机上下线时，主机依据所属的类别自动加入或退出相应系统用户组。目前定义的系统用户组如下表：

用户组编号	名称	解释
0(UGRP_ALL)	全部用户组	所有上网用户都属于该组
1(UGRP_IPMAC)	MAC 绑定用户组	进行了 IP、ARP 协议的源 MAC 地址绑定操作的用户
2(UGRP_IPMAC_IVS)	MAC 未绑定用户组	未进行 IP、ARP 协议的源 MAC 地址绑定操作的用户

3(UGRP_WIRE)	有线用户组	通过有线接口连接到 360 智能路由器的用户
4(UGRP_WIFI)	无线用户组	通过无线口连接到 360 智能路由器的用户
5(UGRP_WIFI_1)	无线网络一用户组(主 AP)	通过特定无线网络（SSID）连接的用户
6(UGRP_WIFI_2)	无线网络二用户组(副 AP)	
7(UGRP_WIFI_3)	无线网络三用户组	
8(UGRP_WIFI_4)	无线网络四用户组	
9(UGRP_WIFI_IPMAC)	无线 MAC 绑定用户组	通过无线连接，并且绑定了 IP、ARP 协议源 MAC 地址的用户。适用于动态获取 IP 或静态 IP 用户
10 (UGRP_WIFI_IPMAC_IVS)	无线 MAC 未绑定用户组	通过无线连接，并且未绑定 IP、ARP 协议源 MAC 地址的用户。适用于动态获取 IP 或静态 IP 用户
11 (UGRP_WIRE_IPMAC)	有线 MAC 绑定用户组	通过有线连接，并且绑定了 IP、ARP 协议源 MAC 地址的用户。适用于动态获取 IP 或静态 IP 用户
12 (UGRP_WIRE_IPMAC_IVS)	有线 MAC 未绑定用户组	通过有线连接，并且未绑定 IP、ARP 协议源 MAC 地址的用户。适用于动态获取 IP 或静态 IP 用户
13(UGRP_PPPOE)	PPPoE 用户组	通过 360 智能路由器中的 PPPoE 服务器接入的内网用户
14(UGRP_PPTP)	PPTP 用户组	所有通过 PPTP 连接到 360 智能路由器的用户
15(UGRP_L2TP)	L2TP 用户组	所有通过 L2TP 连接到 360 智能路由器的用户

- **注册新用户组**

```
struct ip_range {
        struct in_addr start;  /* 起始 IP */
        struct in_addr end;    /* 结束 IP */
};

/* 新增 IP 地址用户组 */
int register_user_group(char *name, int nr,
        struct  ip_range *addr);
/* 替换用户组为 IP 地址用户组 */
```

```
int replace_user_group(int id, char *name, int nr,
        struct ip_range *addr);
/* 新增 MAC 地址用户组 */
int register_user_group_by_mac(char *name, int nr,
          char (*mac)[6]);
/* 替换用户组为 MAC 地址用户组
int replace_user_group_by_mac(int id, char *name, int nr,
          char (*mac)[6])
```

参数:

name[in]: 组名，在系统中必须唯一，否则返回 ERR_NAME_EXIST;

nr[in]: addr 指针指向的数组成员数量，每个用户组最多支持 1000 个 IP 段;

addr[in]: 添加到组中的地址列表，每一个成员是 IP 地址范围;

id[in]: 用于 replace_user_group，是添加新组时得到的组 ID;

返回值:

>=0 调用成功，其值为组号。

<0 注册失败，返回错误码

举例:

1、以 IP 地址注册一个新用户，之后替换这个用户组的内容和名字

```
struct ip_range addr[2];
int id;
addr[0].start.s_addr = htonl(0xc0a80101);
addr[0].end.s_addr = htonl(0xc0a80105);
addr[1].start.s_addr = htonl(0xc0a80110);
addr[1]. end.s_addr = htonl(0xc0a80115);
id = register_user_group(“test”, 2, addr);
/*test 用户组中有 2 段 IP*/
id = replace_user_group(id, “test2”, 1, addr);
/*修改后，用户组名字变成了 test2，id 没有变化，
用户组中只有 addr[0]中的 IP 范围了*/
```

2、以 MAC 地址注册一个新用户组

```
char  mac[2][6] ;
int id;
mac[0][0] =80; mac[0][1] =11; mac[0][2] =22;
mac[0][3] =33; mac[0][4] =44; mac[0][5] =55;
mac[1][0] =90; mac[1][1] =12; mac[1][2] =23;
mac[1][3] =34; mac[1][4] =45; mac[1][5] =56;
id = register_user_group_by_mac (“test”, 2, mac);
/*注意：示例为了清晰展示程序逻辑，代码未判断调用返回值 */
```

- **引用用户组**

用户组数据类型：user_group_mask_t

定义：

typedef unsigned long user_group_mask_t[BITS_TO_LONGS(256)]; /*支持 256 位*/

该数据类型是位掩码，用来选择系统中的多个用户组。目前系统最大支持 256 个用户组。

为方便对位掩码数据结构的操作，提供了一系列二进制位掩码操作函数。

- **获取当前系统中的组信息**

四类系统定义的组都使用以下接口获取当前存在的有效组信息，以便引用它们。

```
struct  rule_comm {
    int id;        /* 组 ID */
    char name[32];   /* 组名 */
};
int dump_group_all(int mid, struct rule_comm *res)
```

参数:

mid[in]: 查询的组类型，有四类:

GRP_USER: 用户组
GRP_TIME: 时间组
GRP_URL: URL 网址组
GRP_DNS: DNS 域名组

res[out]: 输出查询结果，调用者需要事先分配足够的内存，内存大小取决于被查询组类型在系统中支持的最大数量，再乘以 sizeof(struct rule_comm)。

返回值:

>0: 指示 res 指向的数组中有效信息数目;
=0: 表示该组类型没有存在的组;
<0: mid 参数错误

举例:

读取当前的用户组

```
struct rule_comm grp[UGRP_MX];
int nr;
int i;
nr = dump_group_all(GRP_USER, grp);
/*返回 nr 个可用用户组*/
for (i = 0; i < nr; i++)
    printf(“group id is %d, name is %s\n”, grp[i].id, grp[i].name);
```

- **删除组**

用户组、时间组、URL 网址组、DNS 域名组使用相同的删除接口：

```
int unregister_group(int mid, int id)
```

参数:

mid: 为被删除组的类型，有四类:

GRP_USER: 用户组
GRP_TIME: 时间组
GRP_URL: URL 网址组

GRP_DNS: DNS 域名组

id: 将要删除的组 ID

返回值:

>=0 删除成功，<0 删除失败

举例:

1、先注册用户组，然后删除这个用户组

```
struct ip_range addr;
int id;
addr.start.s_addr = htonl(0xc0a80101);
addr.end.s_addr = htonl(0xc0a80105);
id = register_user_group(“test”, 1, &addr);
/*ok，删除这个用户组*/
unregister_group(GRP_USER, id);
```

3·URL 网址组

网络地址组是 URL 组成的集合。每组最大支持 1000 个 URL 地址。系统支持 256 个 URL 地址组。

网址白名单和黑名单是特殊的网络地址组，白名单中的地址能被用户无条件访问，即使用户当前没有进行认证。黑名单是阻止用户访问的地址。

```
#define URL_NAME_LEN 64
```

URL 网址组是 URL 地址的集合。每个 URL 地址最大长度为 URL_NAME_LEN 字节。每个组最多 1000 个成员。每一个 URL 网址是匹配关键字，它能匹配 http 请求中 URL 的 host 部分完整的连续分段内容。

例如：abc.def.ghi.com 中，abc、def、ghi、com 是单个完整分段，abc.def、def.ghi、abc.def.ghi 等是连续的完整分段。URL 关键字匹配只能匹配这些完整的连续分段。本例中，可以匹配关键字 def.ghi，但不能匹配 ef.gh。下一节中的 DNS 网址组也是这样工作。

URL 网址组用于 http 控制功能的 API。

```
/* 新增 URL 网址组 */
int register_url_group(char *name, int nr, char (*url)[URL_NAME_LEN]);
/* 替换已存在组的所有成员 */
int replace_url_group(int id, char *name, int nr, char (*url)[URL_NAME_LEN]);
```

参数:

name[in]: 组名，名称唯一。否则返回 ERR_NAME_EXIST。

nr[in]: url 字符串数组的个数。

url[in]：URL 地址字符串数组。

返回值：

>= 0 时代表注册成功，其值为组 id。

当<0 时，表示注册失败。

举例：

注册一个网址组

```
char url[3][ URL_NAME_LEN] = {“360.cn”, “www.sina.com.cn”, “baidu”};
int id;
id = register_url_group (“test”, 3, url);
```

4·URL 网址过滤策略组

每一个 URL 网址策略组可以包含多个 URL 网址的多种过滤策略，策略组支持灵活的过滤策略，注意：白名单 URL 网址组将不受此功能控制。

- http 策略组注册接口

  int register_http_filter_group(const char *name, int16_t priority);

  参数：

  name：组名，名称必须唯一，最大 16 个字符（包括结尾的空字符）

  priority：优先级，值越小优先级越高

  返回值：

  >= 0 时代表注册成功，其值为组 id。

  当<0 时，表示注册失败。

  NOTES:目前最大支持 128 个组

- http filter 策略组注销接口

  int unregister_http_filter_group(uint16_t group_id);

  参数：

  group_id：调用 register_http_filter_group 时返回的组 id

  返回值：

  = 0 时代表注销成功。

  当<0 时，表示注销失败。

  NOTES:注销成功时自动删除归属于该组的过滤规则

5·http 访问控制

本章节内容提供了丰富的 http 协议控制能力，便于开发者构建商业用途的插件，

例如 portal 认证。

5.1 网址黑、白名单

网址黑名单：阻止该名单中的访问。
网址白名单：任何情况下用户都能访问的网址。
按用户组设置网址黑、白名单，规则注册 API：

/* 将 URL 组加入到特定用户组的黑名单或白名单 */
int register_url_filter_by_url_group(user_group_mask_t gmask，int type，int url_gid)
/* 将指定 URL 加入到特定用户组的黑名单或白名单 */
int register_url_filter (user_group_mask_t gmask，int type，char *url)

参数:

gmask：用户组掩码
type：0 时为网址白名单，1 时为黑名单
url_gid：网址组 id
url：特定网址

返回值:

> 0 时代表注册成功，其值为规则 id 号。当<=0 时，表示注册失败。

规则注销 API:

```
int unregister_url_filter(int id);
```

参数:

id: 为 register_url_filter 或 register_url_filter_by_url_group 成功时的返回值

返回值:

0: 成功; <0: 失败

举例:

1、让 IPMAC 未绑定用户和 PPPOE 用户无条件访问 baidu.com

```
user_group_mask_t gmask ={0,};
int id;
igd_set_bit(UGRP_PPPOE, gmask);
igd_set_bit(UGRP_IPMAC_IVS, gmask);
id = register_url_filter (gmask，0，" baidu.com");
```

1、让所有用户无条件访问 baidu.com,sina.com,qq,163

```
char url[4][ URL_NAME_LEN] = {"baidu.com", "sina.com", "qq", "163"};
int urlid;
int id;
```

```
/*多个网址，注册到一个网址组中*/
urlid = register_url_group (“test”, 4, url);
/*注册网址白名单 */
id = register_url_filter_by_url_group(NULL，0，urlid);
/*接下来，你可以调用 replace_url_group 对放行的 URL 做改变*/
```

5.2 HTTP 高级过滤策略

本节提供更加强大的 HTTP 过滤策略，满足商业插件对路由器的功能需求。调用高级过滤策略 API，需要使用 URL 过滤策略组 ID。

白名单 URL 不受此 API 影响

- 添加 URL 黑名单

  int register_http_filter_reset_rule_by_group(uint16_t group_id, const char *url)

  参　　数：
  group_id: 组 id，通过 register_http_filter_group 函数获取
  url:　指定的网址
  返　　回　　值：
  = 0 时代表添加成功
  当<0 时，表示添加失败。

- 应答内网用户对指定的 url 链接的访问

  int register_http_filter_fake_response_rule_by_group(uint16_t group_id, const char *url, uint16_t status_code, const char *response)

  参　　数：
  group_id: 组 id，通过 register_http_filter_group 函数获取
  url:　指定的网址
  status_code:　HTTP 响应状态码，当 response 为非 NULL 时，忽略该参数
  response: 当 response 为非 NULL 时，response 指向用户自定义的 HTTP 响应，最大长度为 512 个字符
  NOTES: 如果用户自定义 HTTP 响应，其中 response 应为一个完整的 HTTP 响应

  返回值:
  为 0 时代表添加成功
  当<0 时，表示添加失败。

  举例:

```
 //自定义 HTTP 302 响应,将 www.xxx.com 重定向到 www.yyy.com
const char *response = "HTTP/1.1 302 Found\r\n"
                "Server: Apache/2.2.31\r\n"
                "Connection: close\r\n"
                "Location: http://www.yyy.com\r\n"
```

"Content-Type: text/html; charset=utf-8\r\n\r\n"

register_http_filter_fake_response_rule_by_group(0, "www.xxx.com", 0, response)；

//回复 www.xxx.com HTTP/1.1 403

register_http_filter_fake_response_rule_by_group(0, "www.xxx.com", 403, NULL)；

- 对指定的 url 链接数据包做文本替换

int register_http_filter_replace_rule_by_group(uint16_t group_id, const char *url, const struct u_text_replace *original, int o_num, const struct u_text_replace *reply, int r_num)

struct u_text_replace 定义如下：

```
struct u_text_replace {
    uint32_t size_of_struct;
    uint16_t type;
    uint16_t flags;
    unsigned char match_len;
    unsigned char replace_len;
    void *match_data;
    void *replace_data;
};
```

type: 文本替换类型，目前支持如下类型

```
enum {
        HTTP_FILTER_TYPE_UNSPEC,
        HTTP_FILTER_TYPE_REPLACE,
        HTTP_FILTER_TYPE_DELETE,
        HTTP_FILTER_TYPE_INSERT,
};
```

HTTP_FILTER_TYPE_REPLACE: 替换模式，替换指定的子串

HTTP_FILTER_TYPE_DELETE: 删除模式，删除指定的子串

HTTP_FILTER_TYPE_INSERT: 插入模式，在命中的子串附近插入指定的文本串，默认插入在特征串的后面，可以在 flags 中指定 HTTP_FILTER_MATCH_INSERT_BEFORE 来插入到特征串的前面。

flags: 标志，目前支持如下标志

```
#define HTTP_FILTER_MATCH_REPEAT                (1<<0)
#define HTTP_FILTER_MATCH_INSERT_BEFORE         (1<<1)
```

HTTP_FILTER_MATCH_REPEAT: 对当前报文段匹配所有的子串

HTTP_FILTER_MATCH_INSERT_BEFORE: 插入模式类型下有效，表示匹配到子串后，文本数据插入子串前（默认是插入子串后）

match_len：模式串的长度

replace_len：替换/插入文本串的长度，删除模式下忽略该字段

match_data：指向模式文本串，模式文本串支持通配符?和*

replace_data：指向替换/插入文本串，删除模式下忽略该字段

参　　数：

group_id: 组 id，通过 register_http_filter_group 函数获取

url:　指定的网址

original:　指向初始方向的数据替换指针，表示对初始方向的数据包做文本替换

o_num：original 数组长度

reply：指向响应方向的数据替换指针，表示对响应方向的数据包做文本替换

r_num：reply 数组长度

NOTES: original 和 reply 不能同时为空

返　　回　　值：

为 0 时代表添加成功

当<0 时，表示添加失败。

举例:

```
struct u_text_replace org;
struct u_text_replace rep;
//删除初始方向的 Accept-Encoding 首部
org.type = HTTP_FILTER_TYPE_DELETE;
org.flags = 0;
org.match_len = strlen("Accept-Encoding:*\r\n");
org.match_data = "Accept-Encoding:*\r\n";

//在响应方向插入数据,HTTP/1.1 后插入 " test "
rep.type = HTTP_FILTER_TYPE_INSERT;
rep.flags = 0;
rep.match_len = strlen("HTTP/1.1");
rep.match_data = "HTTP/1.1";
rep.replace_len = strlen(" test ");
rep.replace_data = " test ";
register_http_filter_replace_rule_by_group(0, "www.xxx.com", &org, 1, &rep, 1);
```

5.3 Web 认证

针对商业 WIFI 常用的 Web 认证功能，360 智能路由 SDK 提供相关解决方案。

该组 API 能针对内网中指定的用户组，在使用 http（tcp:80）访问互联网的特定网址时，将用户的访问重定向到指定的 URL，这个 URL 可以位于本 360 智能路由器，也可以是 internet 上的服务器。同时，禁止目标用户任何 internet 访问。系统支持多个 web 认证插件，系统将依次进行跳转。

插件可选择跳转到新 URL 时附带的参数：用户设备的 IP 地址、用户设备的 MAC 地址、路由器 http 服务器的 IP 和端口，路由器 LAN 口 MAC 地址，路由器 WAN 口 IP 地址，认证随机数，用户原本访问的 URL。

- **规则注册 API**

```
/* 使用关键字过滤规则的 API */
int register_http_ctrl (user_group_mask_t gmask, int type, char *
        url_pattern,
            char *redir_url, struct redirect_url *data)
/* 使用网址组的过滤规则的 API */
int register_http_ctrl_by_url_group(user_group_mask_t gmask, int type,
            int url_gid, char *redir_url, struct redirect_url
            *data)
/* 仅对指定用户在访问关键网址时重定向一次 */
int register_http_ctrl_once(struct in_addr addr, int type, char
        *url_pattern,
            char *redir_url, struct redirect_url *data)
/* 仅对指定用户访问特定 URL 组时重定向一次 */
int register_http_ctrl_once_by_url_group(struct in_addr addr, int type,
        int url_gid,
            char *redir_url, struct redirect_url *data)
```

参数：

gmask[in]: 规则的目标用户组

type[in]: 操作类型，使用 HTTP_CTRL_TYPE_前缀的宏：

HTTP_CTRL_TYPE_WEBAUTH web 认证跳转

HTTP_CTRL_TYPE_REDIRECT URL 重定向

url_pattern[in]: 关键字过滤规则

对 URL 的 host 中包含该关键字的请求进行操作。url_pattern 可以为 NULL，则匹配所有网址 。

host 中包含的一个或多个连续的完整分段，需要与 url_pattern 完全相同才匹配成功。例如：用户访问 abc.def.com/xxx/xxx.html 时，host 为 abc.def.com，若 url_pattern 为 def 或为 abc.def，则匹配成功；url_pattern 为 ab，则匹配失败。

url_gid[in]: 网址组过滤规则

匹配指定网址组的访问请求

redir_url[in]: type 为 HTTP_CTRL_TYPE_WEBAUTH 或 HTTP_CTRL_TYPE_ REDIRECT 时为跳转的 URL

data[in]：扩展参数

redirect_url 结构定义如下：

```
struct redirect_url {
    uint32_t size_of_struct;
    int interval; /* 跳转间隔，单位秒*/
    int times;  /*跳转次数*/
    int prio;  /* 规则优先级,值越小优先级越高，范围>=0 */
    int islocal; /*页面在路由器本地*/
    unsigned long flags[BITS_TO_LONGS(64)]; /*系统参数标识
*/
    char args[256]; /* 参数*/
};
```

data 可以为 NULL，此时，interval=0，times=0，islocal=0，prio=0，flags=0，args[0]=0。

redirect_url 结构成员的含义：

interval：为跳转间隔，单位秒。为 0 时，代表无间隔。

times：为跳转次数。为 0 时，代表无限次。

prio：为规则匹配优先级，数值越小优先级越高

flags：位掩码，用于选择跳转时附带的参数：

```
/* 用户设备的 IP 地址参数，缺省参数名 pcip */
#define URL_ARGS_PCIP          0
/* 用户设备的 MAC 地址，缺省参数名 pcmac */
#define URL_ARGS_PCMAC         1
/* LAN 内主机访问路由器 http 服务器 IP 和端口，关键字
rgw*/
#define URL_ARGS_RGW           2
/* 路由器 LAN MAC 地址，缺省参数名 rmac */
#define URL_ARGS_RMAC          3
/* 路由器 WAN IP 地址，缺省参数名 rip */
#define URL_ARGS_RIP           4
/* 认证随机数，缺省参数名 magic */
#define URL_ARGS_MAGIC         5
/* 用户设备原本访问的网址，缺省参数名 url */
#define URL_ARGS_URL           6
/* 用户来自无线网络的 SSID，缺省参数名 ssid */
#define URL_ARGS_SSID          7
```

例如：设置 flags 为 ((1<< URL_ARGS_PCIP) | 1 << (URL_ARGS_PCMAC)) 时，跳转后会在原 url 后添加参数?pcip=xxx&pcmac=xxx。

附带参数的名称可以用 replace_http_ctrl_args 函数替换。

args：插件指定的跳转时附加的参数。

strlen(args)=0 则不带参数。对于 web 认证重定向，系统会自动附加参数 rid=xx，xx 为插件调用 register_http_ctrl 的返回值。

islocal：跳转的目标地址的位置。

位于本 360 智能路由器时为 1，同时，redir_url 使用相对路径，不需要 host，只需要 URL，系统会自动加上设备的 IP 和端口号

Internet 网址时为 0，redir_url 使用绝对路径

当 data 为 NULL 时，设置 interval=0, times=0, local_url=0, prio=0，flags=0，strlen(args)=0.

addr[in]：用于对指定用户进行重定向的 API，它是用户的 IP 地址。

返回值： > 0 时代表注册成功，其则是规则 ID。当 id<=0 时，表示注册失败。

重定向一次 API：通常用于通知功能，只能对特定 IP 主机进行操作，重定向一次后规则自动失效。

认证随机数：用于在 web 认证时防止虚假服务器和重放攻击。当 360 智能路由器进行跳转时，自动生成 16 字节随机数。认证服务器可根据该数字来生成认证签名，再发回到插件，插件验证签名合法后，设置用户为已认证状态。

举例:

1、对无线副 AP 用户注册 WEB 认证，跳转到路由器本地的 webauth.htm，并使用系统参数 IP、MAC、SSID, 并使用自定义参数 city=&chengdu&id=028

```
struct redirect_url rd={ 0, };
user_group_mask_t gmask={ 0, };
NOS_STRUCT_INIT(&rd);
 rd.islocal = 1;
 igd_set_bit(URL_ARGS_PCMAC, rd.flags);
 igd_set_bit(URL_ARGS_PCIP, rd.flags);
 igd_set_bit(URL_ARGS_SSID, rd.flags);
strcpy(rd.args, "&city=chengdu&id=028");
 igd_set_bit(UGRP_WIFI_2, gmask);
register_http_ctrl(gmask, HTTP_CTRL_TYPE_WEBAUTH, NULL, “webauth.htm”, &rd);
/* 用户在开启网页时会跳转到 192.168.1.1/webauth.htm?pcip=192.168.1.3&pcmac=80:00:00:00:01:00&ssid=360WIFI&city=chengdu&id=028*/
```

2、同例子 1，跳转到远程页面 www.mywebauth.cn/login.htm。

```
struct redirect_url rd={ 0，};
user_group_mask_t gmask={ 0, };
NOS_STRUCT_INIT(&rd);
 igd_set_bit(URL_ARGS_PCMAC, rd.flags);
 igd_set_bit(URL_ARGS_PCIP, rd.flags);
 igd_set_bit(URL_ARGS_SSID, rd.flags);
strcpy(rd.args, "&city=chengdu&id=028");
 igd_set_bit(UGRP_WIFI_2,  gmask);
register_http_ctrl(gmask, HTTP_CTRL_TYPE_WEBAUTH,
NULL,” www.mywebauth.cn/login.htm” , &rd);
/*注意在使用远程 WEB 认证时，需要首先调用网址白名单对网址
www.mywebauth.cn 放行，否则页面会无法打开*/
```

- **定制附带参数的名称**

在 redirect_url.flags 中指定的跳转附加参数的名称可以用以下函数进行替换：

int replace_http_ctrl_args(int id,char *url[URL_ARGS_MX])

参数：

id：调用 register_http_ctrl_前缀函数的返回值

url：新的名称数组，URL_ARGS_MX 为 64，插件能设置指定参数的名称，名称不得大于 32 字节。例如设置路由器 WAN 口 IP 时：url[URL_ARGS_RIP]=” wanip” 。对不需要修改的参数设置为 NULL。

返回值：成功返回 0，失败小于 0

举例:

1、对 MAC 地址未绑定用户进行 WEB 认证，跳转到路由器本地的 webauth.htm，

并使用系统参数 IP、MAC、SSID, 并修改系统关键字 IP->” devip” , MAC->” devmac” ,SSID->” myssid”

```
struct redirect_url rd={ 0, };
user_group_mask_t gmask={ 0, };
char *url[URL_ARGS_MX] = { NULL, };
int id;
NOS_STRUCT_INIT(&rd);
rd.islocal = 1;
igd_set_bit(URL_ARGS_PCIP, rd.flags);
igd_set_bit(URL_ARGS_PCMAC, rd.flags);
igd_set_bit(URL_ARGS_SSID, rd.flags);
igd_set_bit(UGRP_IPMAC_IVS,  gmask);
id    =    register_http_ctrl(gmask,HTTP_CTRL_TYPE_WEBAUTH,    NULL,
 “webauth.htm” ,  &rd);
url[URL_ARGS_PCIP] =  “devip” ;
```

```
url[URL_ARGS_PCMAC] = “devmac”;
url[URL_ARGS_SSID] = “myssid”;
replace_http_ctrl_args (id, url);
/*用户在开启网页时会跳转到 192.168.1.1/webauth.htm? devip
=192.168.1.3&devmac=80:00:00:00:00:01&myssid=360WIFI*/
```

- **变更用户状态**

```
int http_ctrl_action(struct in_addr addr,  int id,  int action)
```

参数:

addr: 内网主机 IP 地址

id: 为 register_http_ctrl、register_http_ctrl_by_url_group 等成功时的返回值。

action: 动作，可使用宏 IGD_ACTION_ADD、IGD_ACTION_DEL、IGD_ACTION_CLEAN

IGD_ACTION_ADD: 设置主机不再匹配此规则。

IGD_ACTION_DEL: 设置主机需要匹配此规则。

IGD_ACTION_CLEAN: 清除所有 ADD 操作的主机 ，规则重新适用于指定的用户组，此时 addr 可以为任意值。

返回值:

0 成功，<0 代表参数错误

注意: 必须在调用 register_http_ctrl_xxxxx 函数的进程中调用该函数。

举例:

对所有用户进行 WEB 认证，跳转到路由器本地的 webauth.htm。

192.168.1.2 认证后，对此 IP 放行

```
struct in_addr addr;
struct redirect_url rd={ 0，};
int id;
NOS_STRUCT_INIT(&rd);
rd.islocal = 1;
id    =    register_http_ctrl(NULL,HTTP_CTRL_TYPE_WEBAUTH,    NULL,
“webauth.htm”,  &rd);
/*192.168.1.2 认证后，让系统对此放行*/
addr.s_addr=htonl(0xc0a80102);
http_ctrl_action(addr, id,  IGD_ACTION_ADD);
/*现在 192.168.1.2 可以上网了，过一段时间后让 192.168.1.2 重新认证*/
http_ctrl_action(addr,  id,  IGD_ACTION_DEL);
```

- **注销 http 控制规则**

使用以下 API 注销 http 控制规则：

```
int unregister_http_ctrl(int id);
```

参数：

Id ： register_http_ctrl 、 register_http_ctrl_by_url_group 、 register_http_ctrl_once、register_http_ctrl_once_by_url_group 成功时的返回值。

返回值：

0:成功；<0：失败

举例:

对所有用户进行 WEB 认证，跳转到 www.mywifi.cn/webauth.htm。

```
int id;
id=register_http_ctrl(NULL,HTTP_CTRL_TYPE_WEBAUTH,NULL,
 “www.mywifi.cn/webauth.htm”,  NULL);
 unregister_http_ctrl(id);
```

- **让特定用户组跳过网址重定向、WEB 认证规则**

int http_ctrl_skip(user_group_mask_t gmask, int id)

参数：

gmask：用户组掩码

id：为 register_http_ctrl 或 register_http_ctrl_once 成功时的返回值。

成功后代表用户组 gmask 跳过匹配此 id 规则。

返回值：0 成功，<0 代表失败。

举例：

对所有用户进行 WEB 认证，跳转到 www.mywifi.cn/webauth.htm。新建一个用户组，把所有认证通过 IP 放入此用户组，并对这个用户组放行。

```
user_group_mask_t gmask={0, };
struct in_addr addr[2];
int id;
int uid;
id=register_http_ctrl(NULL,HTTP_CTRL_TYPE_WEBAUTH,NULL,
 “www.mywifi.cn/webauth.htm”,  NULL);

 /* 现在 192.168.1.2 认证通过了*/
 addr[0].s_addr=htonl(0xc0a80102);
 uid = register_user_group(“pass”, 1, addr);
 igd_set_bit(uid, gmask);
 /*对用户组放行,调用后，整个用户组都无需认证*/
 http_ctrl_skip(gmask,  id);
 /*现在 192.168.1.3 也认证通过了，只需要把 192.168.1.3 加入
    之前的用户组即可放行*/
 addr[1].s_addr=htonl(0xc0a80103);
 replace_user_group(uid, “pass”, 2,addr);
```

5.4 取消 360 智能路由器中特定 URL 路径的访问认证

当用户访问 360 智能路由器中的页面时，作为安全考虑，会提示用户输入密码。在一些插件中，不希望出现这种现象，例如网址重定向到 360 智能路由器中的页面时，用户通常希望直接展示页面内容，而不需要用户输入密码。该 API 可以将插件专有的目录设置为无需认证即可访问。

规则注册 API：

```
int register_noauth_url(char *url);
```

参数:

url: 取消认证的网址。它以插件的 webs 目录为根，该参数只能设置插件 webs 目录以及之下的子目录。例如附录中 helloworld 示例中，取消了插件的 webs 目录认证，url 参数为 NULL 或“”。

返回值:

> 0 时代表注册成功，其值是规则 ID。当<=0 时，表示注册失败。

规则注销 API:

```
int unregister_noauth_url(int id);
```

参数:

id: 为 register_noauth_url 成功时的返回值。

返回值:

0: 成功，<0: 失败

举例:

对 APP 的 images 和 js 目录放行，且放行 webauth.html。

```
int id1;
int id2;
int id3;
id1 = register_noauth_url( “images/” );
id2 = register_noauth_url( “js/” );
id2 = register_noauth_url( “webauth.html” );
/*现在取消 images 的放行*/
unregister_noauth_url(id1);
```

6·网络接口

6.1 查询、设置 WAN 口配置

6.1.1 查询 WAN 口配置

```
struct nos_wan_cfg {
    uint32_t size_of_struct;                /* Size of this structure */
    int wanid; /* current support: 1 */
    int     connect_type;               /*     WAN_CONN_TYPE_PPPOE     ,
```

```
WAN_CONN_TYPE_STATIC, WAN_CONN_TYPE_DHCP */
    int mtu;
    int nat_mode; /* do nat when !0 */
    char mac[6];
    char user[32]; /*  valid when pppoe */
    char passwd[32]; /* valid when pppoe */
    char server_name[32];/* valid when pppoe */
    char ac_name[32]; /* valid when pppoe */
    struct in_addr ip; /* valid when static ip */
    struct in_addr mask;/* valid when static ip */
    struct in_addr gw;/* valid when static ip */
    struct in_addr auto_dns[IF_DNS_MX];
    struct in_addr manu_dns[IF_DNS_MX];
};
```

int get_wan_config(struct nos_wan_cfg *wan)

参数：

wan[out]：用于保存查询的 WAN 口配置，调用前需要用 NOS_STRUCT_INIT 初始化

返回值：

ERR_NOT_SUPPORT　固件不支持该版本的参数

ERR_FAILURE　函数调用失败。

0 函数调用成功。

6.2 查询设置 LAN 口配置

6.2.1 查询 LAN 口配置

```
struct nos_lan_cfg {
    uint32_t size_of_struct;
    unsigned char mac[6];
    struct in_addr ip;
    struct in_addr mask;
};
```

int get_lan_config(struct nos_lan_cfg *lan)

参数：

lan[out]:用于保存查询的 LAN 口配置参数，调用前需使用 NOS_STRUCT_INIT 初始化。

返回值：

ERR_NOT_SUPPORT　固件不支持该版本的参数

ERR_FAILURE　函数调用失败。

0 函数调用成功。

7·带宽控制

7.1 主机限速

规则注册:

int register_rate_limit(user_group_mask_t gmask, int up_rate,
int down_rate, int conn);

参数:

up_rate：上传速率，单位(KB/s)，0 为不限

down_rate：下载速率，单位(KB/s)，0 为不限

conn：主机最大连接数。

注册成功后，代表对用户组中的每一个 IP 设置此限制值。

返回值:

为 id。id >= 0 时代表注册成功。当 id<0 时，表示注册失败。

注销接口:

int unregister_ rate_limit(int id);

参数:

id 为 register_rate_limit 成功时的返回值

返回值:

0 成功，<0 失败

举例：

对副 AP 限速上行 10K，下行 200K，连接数 300

```
user_group_mask_t gmask = { 0,};
int id;
igd_set_bit(UGRP_WIFI_2, gmask);
id = register_rate_limit(gmask, 10, 200, 300);
```

8·连接网络用户信息

8.1 获取内网主机信息

读取内网 IP 主机信息，可读取该主机 MAC 地址、设备名、所属用户组、NAT session 数量、当前速度、总上行/下行字节数、在线时长信息。

```
int dump_host_info(struct in_addr addr, struct host_info *flow)
```

参数:

addr[in]: 需要查询的 IP 主机地址

flow[out]: 指向存储返回信息的空间，host_info 结构定义如下:

```
struct host_info {
    uint32_t size_of_struct;
    struct in_addr addr;
    unsigned char mac[ETH_ALEN];
    char name[32]; /* host name */
    char manu_name[32]; /* manu host name */
    char dev_label[32];
    user_group_mask_t grp; /* host belongs to which grp bitmap */
    int conn_cnt; /* number of NAT session */
    int up_speed; /*  Byte per second*/
    int down_speed;/* Byte per second*/
    __u64 up_bytes; /*  total upload bytes */
    __u64 down_bytes; /* total download bytes */
    unsigned long up_pkts; /*  upload pkts */
    unsigned long down_pkts; /* download pkts */
    __u64 magic; /* magic number for web auth */
    __u32 second; /*  online second */
    int os_type;/*linux、windows...*/
    int speed_mx[IP_CT_DIR_MAX]; /* peak speed value */
    int device_label; /* device manufacturer */
};
```

返回值 :

< 0 时，查询错误，为 ERR_NON_EXIST 时，查询的主机不存在。

=0 时，成功。

举例:

```
struct host_info info = { { 0,},};
struct in_addr addr;
NOS_STRUCT_INIT(&info);
addr.s_addr=htonl(0xc0a80102);
dump_host_info(addr, &info);
```

8.2 获取内网在线主机信息

```
int dump_host_alive (struct host_info *info);
```

参数:

info[out]: 返回当前在线主机，info 为 HOST_MX 个数组

返回值 :

< 0 时，查询错误。

>=0 时，成功，返回当前在线主机数目。

举例：

```
struct host_info info[HOST_MX];
int nr;
int i;
memset(info, 0, sizeof(info));
for (i=0;i<HOST_MX;i++)
    NOS_STRUCT_INIT(&info[i]);
nr=dump_host_alive(info);
for (i = 0;  i< nr; i ++)
    printf( “ip is %s\n”, inet_ntoa(info[i].addr));
```

9·存储相关接口

9.1 获得 RAM 内存信息

```
int get_raminfo(unsigned long *ramtotal, unsigned long *ramfree)
```

参数：

ramtotal[out]:总内存大小

ramfree[out]:可用内存大小

返回值：

< 0 函数调用失败

>=0 函数调用成功。

9.2 获取插件临时存储路径

获取插件临时数据保存路径，网关重启或掉电，数据将丢失。插件不应保存大于 100KB 的数据在临时存储路径中。在系统存储不足时，将强制清理。

int get_tmp_path (int len,char *tmppath)

参数：

len[in]：tmppath 指向内存的大小

tmppath[out]：指向用于存储返回临时存储路径的内存空间

返回值：

0：成功

ERR_FAILURE：失败

9.3 获取插件永久存储路径

获取插件永久数据保存路径，网关重启或掉电，数据不会丢失。

int get_config_path(int len,char *cfgpath)

参数：

len[in]：cfgpath 指向内存的大小

cfgpath[out]：指向用于存储返回永久存储路径的内存空间

返回值：

0：成功

ERR_FAILURE：失败

10· 系统信息

10.1 获取设备唯一硬件标识

每台 360 智能路由器具有唯一的硬件标识，可利用它来绑定插件的业务授权。

int get_device_hardware_id (void *hwid)

参数：

hwid[out]：指向 16 字节空间，若调用成功，将填写 16 字节硬件唯一 ID。

返回值：0：成功，-1：设备不支持硬件 ID 号

10.2 获取系统启动时间

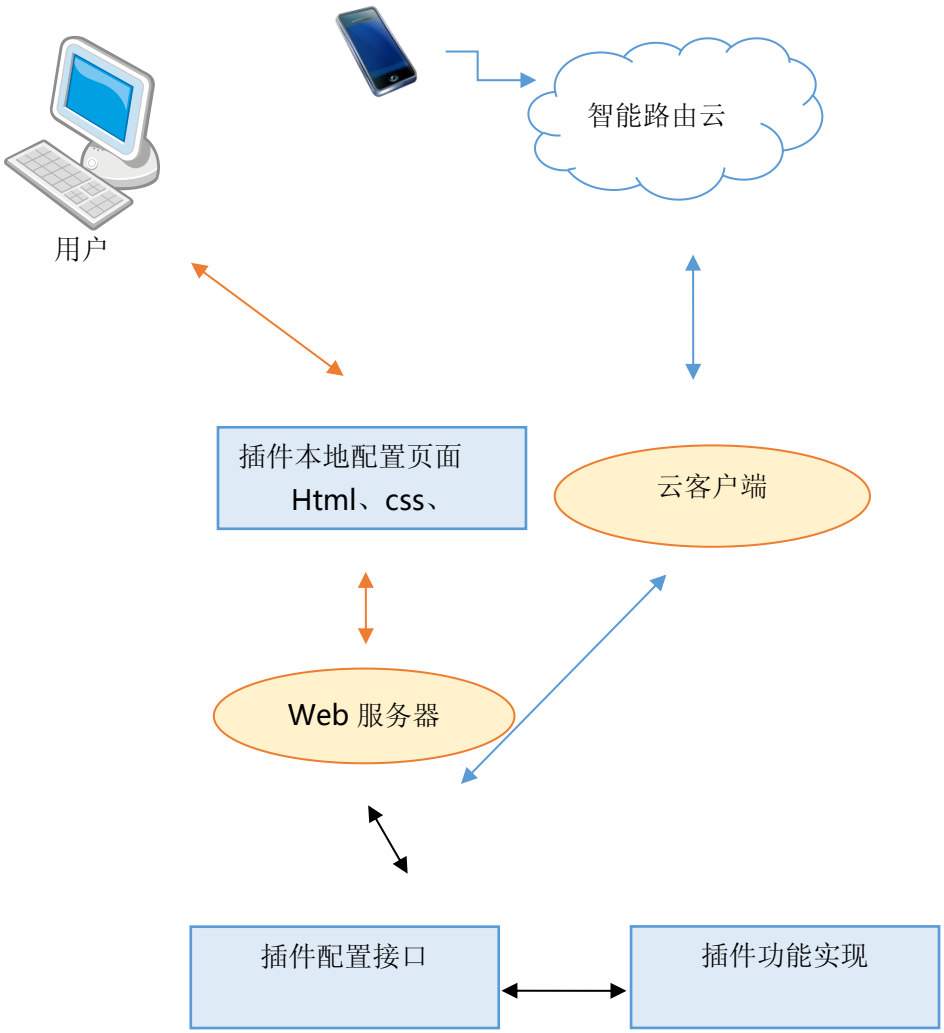
系统启动时间为系统启动后，从 0 开始计时，单调递增，精度为秒。可以使用此时间来计时。

unsigned long get_system_uptime(void)

返回值：

>=0 时，成功，表示系统运行了多少秒

● 插件配置界面和接口



智能路由器用户配置路径

上图中，三个方框的部分，需要由智能路由器插件开发者完成。用户在配置插件时，有三种方式：

- 用浏览器登录网关本地配置界面；
  若仅支持此方式，插件需要具有本地配置页面；
- 用浏览器登录云端访问配置界面；
  若仅支持此方式，插件需要在网关上调试配置页面（开发同上一种方式），在插件发布到插件中心时，将自动将配置界面安装在云端；
- 手机 APP 远程管理方式
  “插件中心”使用上一种方式对插件进行远程配置，即需要插件中心安装了插件的配置界面。

第三方手机客户端也可不使用 web 方式，直接调用云的开放接口，完成对网关的远程配置。若仅需要支持此方式，无需开发插件配置界面。

1· 开发插件配置界面

网页目录结构

/app/插件名称/webs/插件的网页目录

/app/插件名称/webs/index.html 插件的 PC 版页面起始文件

/app/插件名称/webs/index_m.html 插件的手机版页面起始文件

注意：页面内必须使用相对路径。

2·开发插件配置接口（CGI）

插件配置界面以 http post 方式提交数据给插件配置接口（下文简称为 CGI），CGI 响应内容以 json 格式返回。

CGI 程序是独立的可执行文件，它的名称固定为 app.cgi。在 app.cgi 中能处理多个 CGI 接口的请求。

Web 服务器从 http get 请求中的 URL 中提取出本次操作的插件名称。插件名称包含在 URL 的/app/之后，例如：/app/system/webs/set_wireless.cgi 中，system 是插件的名称。Web 服务器会自动执行 system 插件下的 app.cgi 程序，在 app.cgi 中会得到本次调用的接口名称，即 set_wireless.cgi，并执行相应的函数来处理 CGI 参数，例如 username=John 和 set_wireless=0。CGI 程序再与完成实际操作的插件进行通讯，完成设置动作，最后 CGI 向前端返回结果。

CGI 与插件间的通讯方式，系统中没有作规定，可以使用 linux 提供的各种进程间通讯机制。

2.1 360 OS 对 CGI 开发的支持

开发 CGI 程序，需要用到 cgi.so 和 cgi.h。

它们包含了一些常用的函数。CGI 程序的 main 函数定义在 cgi.so 中，所以在

开发 CGI 时，不需要在自己的程序中包含 main 函数，只需链接 cgi.so 即可。

在 cgi.so 的 main 函数中，会首先得到被请求的 CGI 接口名称，例如前例中的 set_wireless.cgi，从 CGI 开发者定义的 CGI 接口表中，找到相应的函数入口，并调用它来处理参数。这个入口表为 struct IGD_CGIMAP_ENTRY 数组。

定义如下：

```
struct IGD_CGIMAP_ENTRY {
        char *name;
        IGD_CGI_HANDLER   handler;
        int  user_perm_flag;
```

2.2 定义插件的 CGI 接口

即定义一个 struct IGD_CGIMAP_ENTRY 数组，数组的名称必须是 IGD_CGI_FUN_MAP，每个成员有三部分组成：

11. 第一部分是字符串，它是 CGI 接口名称，必须以“.cgi”结尾，例如：set_wireless.cgi；
12. 第二部分是函数指针，它是处理该 CGI 接口的函数入口；
13. 第三部分是权限设定，可以使用宏 PERM_ALL 和 PERM_LE_ADMIN，PERM_ALL 表示该接口允许所有用户调用，即本地 web 登录和插件中心远程操作，PERM_LE_ADMIN 只允许本地 web 登录使用该接口。

数组以第一部分为 NULL 的成员结束。

例如：

```
struct  IGD_CGIMAP_ENTRY   IGD_CGI_FUN_MAP[]= {
    {"ip_dns_args_dump.cgi",  ip_dns_args_dump_cgi,  PERM_ALL},
    {"ip_dns_args_set.cgi",  ip_dns_args_set_cgi,  PERM_LE_ADMIN},
    {"ip_dns_set.cgi",  ip_dns_set_cgi,  PERM_LE_ADMIN},
    {"ip_dns_show.cgi",  ip_dns_show_cgi,  PERM_ALL},
    {"ip_dns_clean.cgi",  ip_dns_clean_cgi,  PERM_LE_ADMIN},
    {"ip_dns_del.cgi",  ip_dns_del_cgi,  PERM_LE_ADMIN},
   {NULL,}
};
```

2.3 编写 CGI 处理函数

CGI 处理函数需要完成对 CGI 参数的处理，控制插件程序完成用户请求的操作，最后返回操作结果。

CGI 处理程序的原型是：

```
typedef int (*IGD_CGI_HANDLER)(struct httpd * hp, struct httpform * form,
  char ** filetext);
```

第一个参数：不再使用
第二个参数定义是：

```
struct httpform {
        node* head;   /* 参数链表头 */
        struct  in_addr  user_ip;    /* 用户的地址  */
};
```

其中 node 定义为：

```
typedef struct _node
{
   entrytype   entry;    /* 其中一个参数  */
   struct _node*   next;   /* 下一个参数 */
} node;
typedef struct
{
   char *name;    /* 请求参数名称 */
   char *value;    /*  参数的值 */
} entrytype;
```

第三个参数：不再使用
返回值：FP_OK：成功
FP_ERR：失败

- 取得 CGI 参数

通过函数的第二个参数，可以取得客户的 IP 地址和 CGI 参数。开发者可以调用 get_form_value 函数来取得某一个参数的值：

```
char * get_form_value(struct httpform * form,  char * name)
/* form [in]:  CGI 函数的第二个参数
  name[in]:  需要获取的参数名称
  RETURN:  指向参数值的指针，返回 NULL 为未找到相应参数  */
```

例如： set_wireless=get_form_value (form,” set_wireless”);

set_wireless 将指向字符串” 0” 。函数中要自行转换参数的类型。

- 调用插件程序

CGI 程序仅是 web 与插件之间沟通的桥梁，需要由插件程序来执行实际动作，并保存配置参数。

系统没有规定 CGI 程序与插件之间的通讯方式，需要开发者自行设计。

- 返回结果

CGI 程序调用插件程序之后，会得到操作结果。时常包含较多的信息。这些信息需要发送给 web 前端，并显示给用户。CGI 程序负责将这些信息封装为 json 格式。

SDK 提供一些宏用于简化 JSON 封装过程，首先需要定义在宏中使用到的 char *临时变量，它是所有宏的第一个参数。

/* 初始化 json buff，对 buf 进行初始化 */

json_buf_set(buf)

/* 初始化 json buff，并开始构建 JSON 对象开始，它将输出’{ ‘ */

json_object_start (buf)

/* JSON 对象开始，它将输出”{ “ */

json_object_start_mid(buf)

/* JSON 对象结束，它将判断 JSON 串最后一个字符是否是”,”，

如果是，它将更换为”},”

json_object_end(buf)

/* 封装 json 名称/值对，key 是名称，fmt 是参数的格式，args 是参数 */

json_value (buf,key,fmt,args…)

例如：

char uname[]=” John”;

json_value (ptr, “username”, “%s”, uname);

结果：” username”:” John”,

/* 封装 json 数字 */

json_number(buf,num)

例如：json_number(ptr,123)

结果：123,

```
/* 用于封装多个对象，例如数组成员，index 为对象的序号，
    当 index=0 时它将输出 “{ “，index>0 时输出”,{ “
例如：{ “name”:” John”},{ “name”:” Tom”} */
json_object_begin_start(buf,index)

/* 用于封装多个对象时输出 ‘}’ 符号 */
json_object_begin_end(buf)

/* JSON 数组开始，它将输出”[ “ */
json_array_start(buf)

/* JSON 数组结束，它将输出”]” */
json_array_end(buf)

/* 仅封装 JSON 名称/值对中的名称 */
json_name(buf,name)
例如：json_name (ptr,” array”)
结果：” array”:
```

注意：

CGI 返回串中，不允许存在名为 errorcode 的 json 对象。

● 附录

1. 示例一：Hello,World!

本示例的目标是对每一位新上线的用户展示一次 Hello,World!页面。源码位于 SDK 中 example/helloworld 目录。

文件说明如下：

文件名	说明

hello.c	插件主程序，启动时完成必要的资源申请；通知插件管理程序插件正常启动；等待与 CGI 和系统通讯；收到退出通知时退出程序
hello.h	插件头文件
cgi.c	CGI 程序
cgicomm.c	CGI 与插件主程序通讯使用的代码
cgicomm.h	CGI 与插件主程序通讯使用的代码
Makefile	Makefile 文件，完成 C 文件编译、插件打包，生成 helloworld.opk
helloworld/	插件的打包目录
helloworld/app.json	插件描述文件
helloworld/helloworld.png helloworld_b.png helloworld_w.png	helloworld 插件的图标文件
helloworld/bin	存放二进制执行程序的目录，helloworld 执行程序将放在这里
helloworld/webs	网页和 CGI 程序存放目录，app.cgi 将放在这个目录
helloworld/webs/index.html	配置页面文件

Helloworld 程序的编译方法：

进入 SDK 目录：

source ./env-rtk.sh

cd example/helloworld

make

生成了 helloworld.opk 文件

2. 示例二：网页用户认证

这是一个 web 认证的示例程序。管理员通过配置界面填写 URL 地址,则用户的访问会重定向到填写的 URL，填写的 URL 需是 Internet 服务器上。

用户认证的页面。当认证成功之后，示例程序会撤销重定向，用户则可正常访问 Internet。

源码位于 example/webauth 目录中。

流程说明

- 程序启动

程序启动之后，读取保存的设置参数，根据参数值决定是否需要注册 web 认证的功能。接着等待接受 CGI 处理程序发送的消息。

- 管理员配置。

  管理员通过插件配置界面填写 URL，URL 需是 Internet 服务器上的用户认证页面。页面发送 CGI 请求(auth_info_set.cgi)，把数据传输给路由器插件。

  处理函数(webauth_app.c 中的 auth_info_set 函数)。此函数根据管理员的配置，设置或者撤销重定向，同时保存配置。重定向的 URL 附带的参数，包括 ip、magic、用户原来访问的网址。通过调用 register_http_ctrl 函数来完成认证重定向。

  无需路由器的 web 认证的 CGI 请求(auth_info_check.cgi)，则需调用 register_noauth_url 函数放行。

- 用户认证

  管理员设置重定向成功以后，用户访问 Internet 都会重定向到指定的认证服务器。用户需输入用户名及密码，进行用户认证。若认证成功，Internet 服务器上相关处理程序(示例是通过网页做 MD5 计算)需把用户名、密码、ip、magic、私有密钥 key 组成的字符串计算 MD5。再通过页面把用户名、密码、ip、magic、MD5 加密数据发送 CGI 请求(auth_info_check.cgi)，把数据传输给 app 处理函数(webauth_app.c 中的 auth_info_check 函数)。此函数通过传递的用户名、密码、ip、magic 与私用的密钥 key(需与服务器的私有密钥一致)组成的字符串使用 MD5 加密。加密得到的数据与传递的 MD5 加密数据进行比较，若一致则撤销重定向，跳转到用户原来访问的网址。撤销重定向调用 http_ctrl_action 函数，需传入用户主机的 IP，即用户主机通过认证，可以正常访问 Internet。

3. 位掩码操作函数

360 智能路由器提供的位掩码操作函数定义在 igd_lib.h，以下介绍最常用的一部分，其它内容请参考 igd_lib.h。

3.1 位掩码初始化

```
/* 把所有二进制位初始化为 0 */
    static inline int igd_init_bit(int nr, unsigned long *bit)
```

参数：

nr[in]：bit 参数指向的位掩码的二进制位数
bit[out]：需要初始化的位掩码
返回值：
0：成功

3.2 设置位掩码中的一位

/* 将位掩码中的一位设置为 1 */
static inline int igd_set_bit(int nr, unsigned long *bit)
参数：
nr[in]：设置 bit 中第 nr 位为 1
bit[in/out]：指向位掩码
返回值：
0：成功

3.3 清除位掩码中的一位

/* 将位掩码中的一位清除为 0 */
static inline int igd_clear_bit(int nr, unsigned long *bit)
参数：
nr[in]：设置 bit 中第 nr 位为 0
bit[in/out]：指向位掩码
返回值：
0：成功

3.4 测试位掩码中的一位

/* 返回掩码中指定位的值 */
static inline int igd_test_bit(int nr, unsigned long *bit)
参数：
nr[in]：要测试的位
bit[in]：指向位掩码
返回值：
1：该位为 1，0:该位为 0