ouyangxibao RP 1
2019-06-28 发布

VPP API机制分析（上）

VPP除了使用命令行进行配置外，还可以使用API进行配置。VPP不仅支持c语言的API，还支持python，java，lua等高级语言的API，非常适合自动化部署。

API简单使用案例

VPP提供了一个VAT客户端程序，用于进行简单的API测试。可执行文件位于：

- debug版本：vpp源码路径/build-root/build-vpp_debug-native/vpp/bin/vpp_api_test
- release版本：vpp源码路径/build-root/build-vpp-native/vpp/bin/vpp_api_test

启动vpp与vat程序

```
sudo systemctl start vpp
cd vpp源码路径/build-root/build-vpp-native/vpp/bin/
sudo vpp_api_test
load_one_plugin:68: Loaded plugin: /usr/lib/x86_64-linux-gnu/vpp_api_test_plugins/memif_t
......
load_one_plugin:68: Loaded plugin: /usr/lib/x86_64-linux-gnu/vpp_api_test_plugins/mactime
vat#
vat#
```

简单使用

- [ ] **help**命令列出所有的命令

```
vat#
vat# help
Help is available for the following:
acl_add_replace
......
```

- [ ] **help + 具体命令** 列出指定命令的帮助信息

```
vat# help acl_del
usage: acl_del <acl-idx>
vat#
```

- [ ] **quit** 或者 **q** 命令退出vat

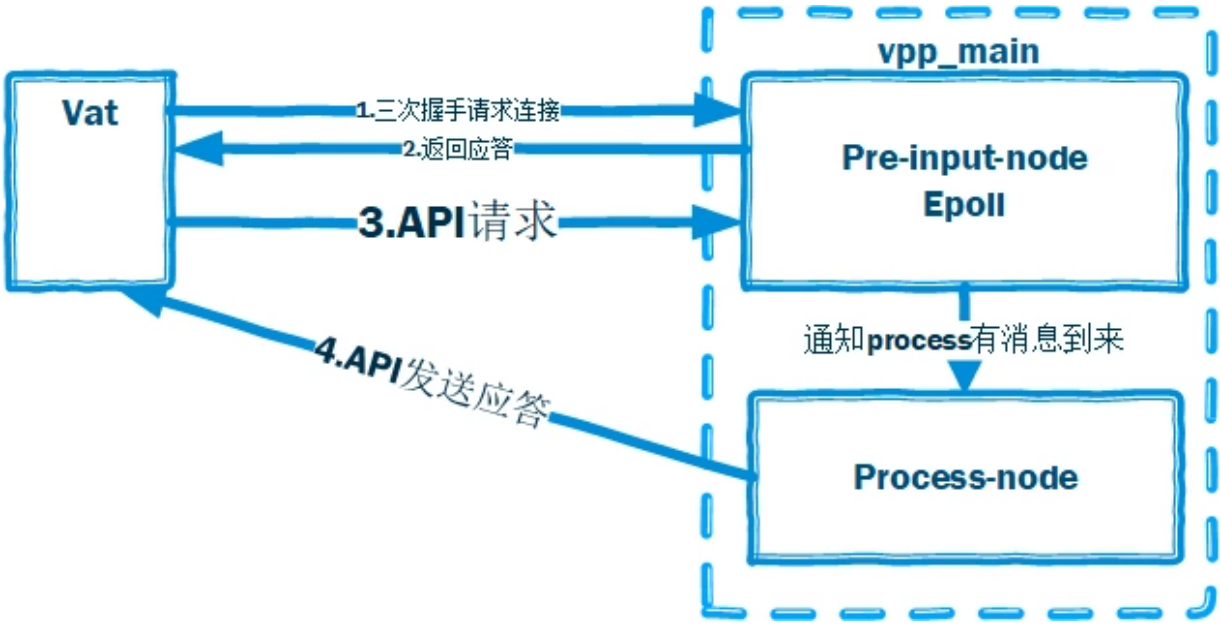
```
vat# quit
admin@ubuntu:~/vpp/build-root/build-vpp-native/vpp/bin$
```

- [ ] **show_version** 显示VPP版本

```
vat# show_version
        program: vpe
        version: 19.08-rc0~65-g3b62e29c3
     build date: Sat Apr 20 13:38:27 CST 2019
build directory: /home/admin/vpp
vat#
```

VPP API交互过程

vpp的api是CS模型。VPP是服务器端，监听客户端连接请求，处理客户端发送的api请求，返回应答。如下图所示：



如上图所示，客户端VAT向VPP请求连接，VPP返回新连接。连接建立之后，VAT发送API请求，VPP处理请求返回应答。VPP支持unix套接字或者inet套接字。VPP支持多个客户端同时进行请求。VPP支持使用共享内存进行数据交换，当客户端和服务器端都在同一个节点上的时候，可以选择使用共享内存进行message交换(3,和4交互过程可以选择共享内存交换也可以选择套接字交换)，提高通信速率。

VPP API编写说明

VPP API命名规则

vpp一共支持三种类型的API：

- [ ] **request/reply**
  这种类型的规则，客户端发送一个请求，VPP发送一个应答。应答消息命名为：method_name + _reply。
- [ ] **Dump/Detail**
  客户端发送一个bulk请求，VPP发送多个details消息。一个dump/detail请求最终会以一个"control ping block "结束。请求方法命名规则为： method + _dump。 应答方法命名规则为method + _details。该类型消息主要用来请求一个表消息，比如FIB表包含多个表项，可以使用一个该消息获取所有的FIB表项信息。
- [ ] **Events**
  客户端可以通过该类型API向服务器注册一些获取异步消息通知。比如获取interface的状态变化消息，周期性统计消息等。这类API通常以"want_"字段作为前缀，比如："want_interface_events"。

客户端发送的消息会包含一个'client_index'字段，该字段对服务器端是透明的，相当于一个‘cookie’，用于区分不同的客户端。

- [ ] **命名**
  - Reply/Request 方法. 请求: *名字* 应答: *名字*+_reply
  - Dump/Detail 方法. 请求: *名字*+_dump Reply: *名字*+_details
  - Event 注册方法: 请求: want_+*名字* Reply: want_+*名字*+_reply

VPP API编写步骤(这部分内容来自于网络)

注：这部分内容来自于网络 [](https://blog.51cto.com/zhangc...

添加一个新的二进制控制层的API，涉及两部分内容，一部分是客户端，另一个就是服务器端。我们以acl插件的api为例进行说明。

添加一个api需要修改三个文件。代码路径是vpp/src/plugins/acl下
acl.api -- vat 与vpp 通信的结构体定义
acl_test.c -- vat使用(客户端)
acl.c -- vpp使用(服务器端)

1.acl.api

acl.api中定义vat与vpp通信的结构体，然后由vppapigen文件处理，最终生成acl.api.h的头文件。两边都包含这个头文件，这样vat与vpp就使用了相同的结构体通信了。 我们看一下acl.api中的定义:

```
/** \brief Delete an ACL
    @param client_index - opaque cookie to identify the sender
    @param context - sender context, to match reply w/ request
    @param acl_index - ACL index to delete
*/

autoreply manual_print define acl_del
{
  u32 client_index;  //系统使用
  u32 context;       //系统使用
  u32 acl_index;     //通过命令acl_del <acl-idx>输入的acl-idx
};
```

这个结构体的定义由3个关键字（autoreply 、manual_print、define ）加上名称再加成员构成，最终会被转化为:

```
typedef VL_API_PACKED(struct _vl_api_acl_del {
    u16 _vl_msg_id;
    u32 client_index;
    u32 context;
    u32 acl_index;
}) vl_api_acl_del_t;

typedef VL_API_PACKED(struct _vl_api_acl_del_reply {
    u16 _vl_msg_id;
    u32 context;
    i32 retval;
}) vl_api_acl_del_reply_t;
```

这样就可以用使用vl_api_acl_del_t与vl_api_acl_del_reply这个结构体通信了。
具体说一下每个部分:

关键字分析

- 关键字 autoreply
  在这里需要先提一下reply
  正常情况下，vat发送一个请求消息，然后等待一个reply消息。所以xxx_reply_t结构是不可少的，可以自己写，也可自动生成。
  而这个关键字表示了自动生成xxx_reply_t结构体，但是自动生成的结构体只有默认的参数

_vl_msg_id，context，retval。如上所示vl_api_acl_del_reply_t。
这个转换的函数实现如下。

```
void autoreply (void *np_arg)
{
static u8 *s;
node_t *np = (node_t *)np_arg;
int i;

vec_reset_length (s);

s = format (0, " define %s_reply\n", (char *)(np->data[0]));
s = format (s, "{\n");
s = format (s, "    u32 context;\n");
s = format (s, "    i32 retval;\n");
s = format (s, "};\n");

for (i = 0; i < vec_len (s); i++)
    clib_fifo_add1 (push_input_fifo, s[i]);
}
```

- 关键字manual_print
  xxx_print函数是用来打印消息结构体内容的。默认情况下会自动生成。如果你想自己来实现，就需要加入这个关键字，然后在模块路径下的manual_fns.h中实现。

```
static inline void *
vl_api_acl_del_t_print (vl_api_macip_acl_del_t * a, void *handle)
{
  u8 *s;

  s = format (0, "SCRIPT: acl_del %d ",
              clib_host_to_net_u32 (a->acl_index));

  PRINT_S;
  return handle;
}
```

- 关键字define
  define 关键字是转化的关键。每个定义都要加上。
- 其他关键字
  还有一些其他的关键字，大家对比一下定义与生成的结果也基本都能看出来，在这里就不赘述了。

2.acl_test.c

这个文件是vat使用。

有三件事要做，1.写cli的help 2.写函数 3.函数加载

2.1写cli的help

```
#define foreach_vpe_api_msg \
_(acl_del, "<acl-idx>")
```

2.2写函数

我们需要写两个函数

api_acl_del与vl_api_acl_del_reply_t_handler

这两个函数是配合使用的，来一个一个看

- api_acl_del

  这个函数是与cli直接关联，命令输入后就调用的就是这个函数.

```
static int api_acl_del (vat_main_t * vam)
{
    unformat_input_t * i = vam->input;
    //这个结构体就是在acl.api中定义的消息传递结构体
    vl_api_acl_del_t * mp;
    u32 acl_index = ~0;
    int ret;

    //解析字符串，跟vpp的命令行解析一样
    if (!unformat (i, "%d", &acl_index)) {
      errmsg ("missing acl index\n");
      return -99;
    }

    //给mp分配内存，然后填写要传递的值
    /* Construct the API message */
    M(ACL_DEL, mp);
    mp->acl_index = ntohl(acl_index);

    /* send it... */
    S(mp);

    /* Wait for a reply... */
    W (ret);
```

在这里把这几个宏的实现也贴一下。对应一下，就能看明白了。

```
/* M: construct, but don't yet send a message */
#define M(T, mp)                                                \
```

```
do {                                                            \
    vam->result_ready = 0;                                      \
    mp = vl_msg_api_alloc_as_if_client(sizeof(*mp));            \
    memset (mp, 0, sizeof (*mp));                               \
    mp->_vl_msg_id = ntohs (VL_API_##T+__plugin_msg_base);      \
    mp->client_index = vam->my_client_index;                    \
} while(0);

/* S: send a message */
#define S(mp) (vl_msg_api_send_shmem (vam->vl_input_queue, (u8 *)&mp))

/* W: wait for results, with timeout */
#define W(ret)                  \
do {                                            \
    f64 timeout = vat_time_now (vam) + 1.0;     \
    ret = -99;                                  \
                                                \
    while (vat_time_now (vam) < timeout) {      \
        if (vam->result_ready == 1) {           \
            ret = vam->retval;                  \
            break;                              \
```

- vl_api_acl_del_reply_t_handler
  这个函数是在vpp回复消息后，clinet接收回应的函数。
  由于大多数都一样，就直接用宏来实现了。

```
#define foreach_standard_reply_retval_handler   \
_(acl_del_reply)

#define _(n)                                            \
    static void vl_api_##n##_t_handler                  \
    (vl_api_##n##_t * mp)                               \
    {                                                   \
        vat_main_t * vam = acl_test_main.vat_main;   \
        i32 retval = ntohl(mp->retval);                 \
        if (vam->async_mode) {                          \
            vam->async_errors += (retval < 0);          \
        } else {                                        \
            vam->retval = retval;                       \
            vam->result_ready = 1;                      \
        }                                               \
    }
foreach_standard_reply_retval_handler;
#undef _
```

注上面这个宏只是定义多个xxx_reply_retval_handler函数

- api_acl_del和acl_del_reply函数的关系
  这两个函数是在不同的线程，在宏W中，是在等待result_ready被置位；而result_ready 就是在acl_del_reply中被置位的。

从下面信息可以看出VAT有两个线程：

```
admin@ubuntu:~/vpp$ pstree -p `pidof vpp_api_test`
vpp_api_test(25810)---{vpp_api_test}(25811)
admin@ubuntu:~/vpp$
```

也可以从如下的gdb信息可以看出vat有一个线程进行应答消息的处理。

```
(gdb) info thread
  Id   Target Id         Frame
  1    Thread 0x7f3bb05d2b80 (LWP 25810) "vpp_api_test" vat_time_now (
    vam=vam@entry=0x5575ebcdc800 <vat_main>)
    at /home/jd/vpp/src/vat/api_format.c:141
* 2    Thread 0x7f3ba6d09700 (LWP 25811) "vpp_api_test" 0x00007f3ba5cec470 in vl_api_ac
    at /home/jd/vpp/src/plugins/acl/acl_test.c:91
(gdb) bt
#0  0x00007f3ba5cec470 in vl_api_acl_del_reply_t_handler (mp=0x13004dee8)
    at /home/jd/vpp/src/plugins/acl/acl_test.c:91
#1  0x00007f3bb01c2251 in msg_handler_internal (free_it=1, do_it=1,
    trace_it=<optimized out>, the_msg=0x13004dee8,
    am=0x7f3bb03ca420 <api_main>)
    at /home/jd/vpp/src/vlibapi/api_shared.c:425
#2  vl_msg_api_handler (the_msg=0x13004dee8)
    at /home/jd/vpp/src/vlibapi/api_shared.c:559
#3  0x00007f3bb01c321a in vl_msg_api_queue_handler (
    q=q@entry=0x1301c69c0) at /home/jd/vpp/src/vlibapi/api_shared.c:770
#4  0x00007f3bb01bc07e in rx_thread_fn (arg=<optimized out>)
    at /home/jd/vpp/src/vlibmemory/memory_client.c:94
#5  0x00007f3baf6b86db in start_thread (arg=0x7f3ba6d09700)
    at pthread_create.c:463
#6  0x00007f3baf3e188f in clone ()
```

2.3加载函数

需要把写的两个函数挂载上。只需要在对应的宏下按格式写就好了。

2.3.1 在宏中添加定义

- api_acl_del
  在foreach_vpe_api_msg宏下定义
  其实这个在“2.1写cli的help”中已经写过，就不用再写了

```
/*
 * List of messages that the api test plugin sends,
 * and that the data plane plugin processes
 */
```

```
#define foreach_vpe_api_msg
_(acl_del, "<acl-idx>") \
```

- vl_api_acl_del_reply_t_handler
  在foreach_vpe_api_reply_msg宏下定义

```
/*
* Table of message reply handlers, must include boilerplate handlers
* we just generated
*/
#define foreach_vpe_api_reply_msg                                       \
_(ACL_DEL_REPLY, acl_del_reply)
```

2.3.2 函数挂载

上一节提到的宏，都是在acl_vat_api_hookup这个函数中使用的，我们不需要做任何修改。

```
static
void acl_vat_api_hookup (vat_main_t *vam)
{
    acl_test_main_t * sm = &acl_test_main;
    /* Hook up handlers for replies from the data plane plug-in */
#define _(N,n)                                                  \
    vl_msg_api_set_handlers((VL_API_##N + sm->msg_id_base),     \
                            #n,                                 \
                            vl_api_##n##_t_handler,             \
                            vl_noop_handler,                    \
                            vl_api_##n##_t_endian,              \
                            vl_api_##n##_t_print,               \
                            sizeof(vl_api_##n##_t), 1);
    foreach_vpe_api_reply_msg;
#undef _

    /* API messages we can send */
#define _(n,h) hash_set_mem (vam->function_by_name, #n, api_##n);
    foreach_vpe_api_msg;
#undef _

    /* Help strings */
#define _(n,h) hash_set_mem (vam->help_by_name, #n, h);
    foreach_vpe_api_msg;
```

3.acl.c

这个文件是vpp使用，它用来接收vat(vpp-api-test)发送的消息，然后处理，最后回应给vat。
我们需要写对应的函数，然后挂上就可以了

- **写宏**

  只需要在这个宏里，把函数添加进去即可

```
/* List of message types that this plugin understands */

#define foreach_acl_plugin_api_msg      \
_(ACL_DEL, acl_del)
```

- **写函数**

  这里的函数也得遵循格式，vl_api_xxx_t_handler，函数如下所示

```
static void
vl_api_acl_del_t_handler (vl_api_acl_del_t * mp)
{
  acl_main_t *am = &acl_main;
  //这个结构体就是在acl.api中定义的消息应答传递结构体，用于给VAT发送应答消息
  vl_api_acl_del_reply_t *rmp;
  int rv;

  //mp中就是VAT发送来的结构体，我们可以从中取得配置的acl_index使用。然后调用相应的处理函数。
  rv = acl_del_list (ntohl (mp->acl_index));

  //这里是消息处理完毕后的应答消息，VAT会在那里等待回应。也是通过共享内存的方式来通信。
  //如果需要在回应消息里传递参数，可以使用另一个宏 ---  REPLY_MACRO2
  REPLY_MACRO (VL_API_ACL_DEL_REPLY);
}
```

- **挂钩子**

  使用定义的宏来挂载函数。这里也不用做任何的改变。

```
/* Set up the API message handling tables */
static clib_error_t *
acl_plugin_api_hookup (vlib_main_t * vm)
{
acl_main_t *am = &acl_main;
#define _(N,n)                                                  \
vl_msg_api_set_handlers((VL_API_##N + am->msg_id_base),     \
                        #n,                     \
                        vl_api_##n##_t_handler,             \
                        vl_noop_handler,                    \
                        vl_api_##n##_t_endian,              \
                        vl_api_##n##_t_print,               \
                        sizeof(vl_api_##n##_t), 1);
foreach_acl_plugin_api_msg;
#undef _
```

```
return 0;
}
```

参考链接

[](https://wiki.fd.io/view/VPP/A...

[](https://wiki.fd.io/view/VPP/H...

你可能感兴趣的

- **KVM halt-polling机制分析** 云计算
- **Koa源码分析（三） -- middleware机制的实现** raulzuo node.js
- **MySQL InnoDB锁机制全面解析分享** leeon mysql innodb
- **详细深入分析 Java ClassLoader 工作机制** zhisheng classloader jvm intellij-idea java
- **Log42j 源代码分析：plugin（插件）机制** xingpingz log4j2 java
- **浏览器缓存机制分析** juan26 前端 缓存
- **android源码分析-深入消息机制** 机械面条 c++ java
- **对Koa-middleware实现机制的分析** ssssyoki co koa.js koa-compose javascript

评论