VPP 源码分析

Version <vpp-18.01>

Revision History

Date	Version	Description	Author
2018-04-21	vpp-18.01	VPP 源码分析	陈漂评

目　录

1. 架构介绍

1.1 线程模型

vpp 采用多线程模型，按照功能来划分当前线程有四类：hqos、stats、worker 和 main。

介绍每个核的 runtime 结构，如何同步

介绍同步锁机制

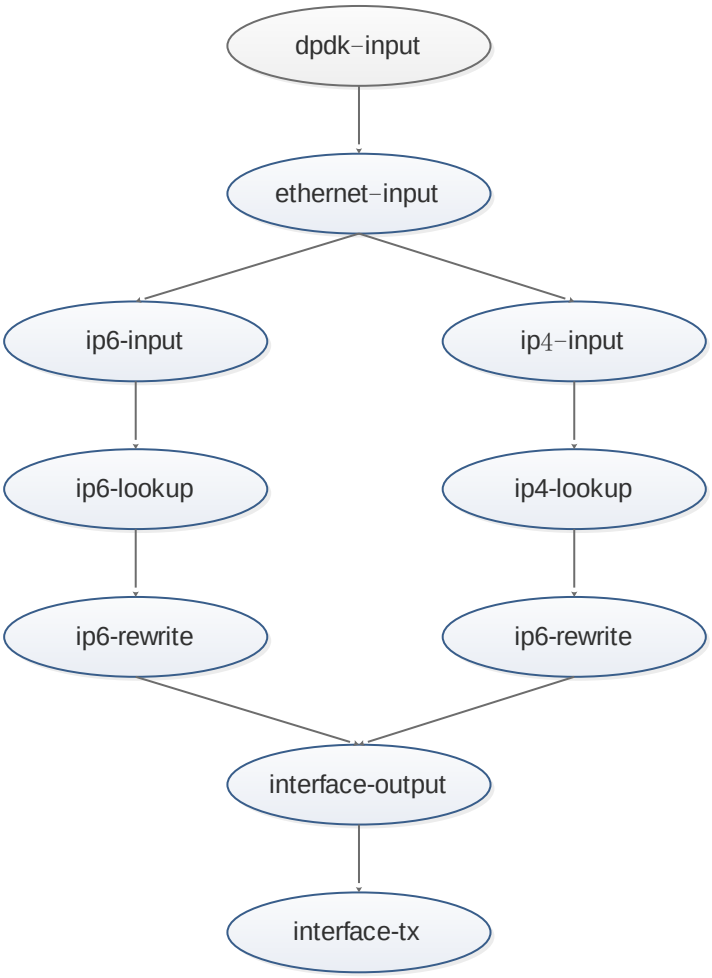
线程的初始化

线程和 vm 的关系

（待完善）

1.2 node 机制

vpp 把一个包的处理流程划分成多个阶段，每个阶段只做一部分处理工作，报文经过所有的阶段处理后完成报文的处理。一个阶段称为 node（中文叫做节点），即一个报文从某个口收上来后会经过一系列的 node 处理，然后从某个口发送出去。如下图所示：



按照上图，ipv4 报文经过的 node 依次为：dpdk-input→ip4-input→ip4-lookup→ip4-rewrite→interface-output→interface-tx；ipv6 报文经过的 node 依次为：dpdk-input→ip6-input→ip6-lookup→ip6-rewrite→interface-output→interface-tx。这里只是为了说明 vpp 的报文处理流程，暂时不关系这些 node 都对报文做了什么处理。因此，理解 vpp 的转发框架的重点在于理解 node 的结构、node 之间的关系等机制。那么，一个 node 处理完报文以后怎么知道扔给哪个 node 处理呢？可以根据报文的类型来决定，如 ethernet-input 把 ipv4 和 ipv6 的报文分别送给 ip4-input 和 ip6-input 节点去处理，也可以是根据预定义的规则来决定往哪个 node 走。总之，往哪个 node 走由当前处理报文的 node 说了算。但是一般情况下只会把报文发送给自己的孩子节点处理，但并不是说不能发送给非孩子节点。

1.2.1 node 图

如上图所示所有的 node 组成一张有向图，我们按照树的叫法，把相邻的 node 之间的关系称为父节点和孩子节点。比如上图中 dpdk-input 是 ethernet-input 的父节点，而 ethernet-input 是 dpdk-input 孩子节点。任意一个节点可以有多个孩子节点，也可以有多个父节点，比如 ethernet-input 有两个孩子节点 ip4-input 和 ip6-input，interface-output 有两个父节点 ip4-rewrite 和 ip6-rewrite。节点之间还有另外一种关系，叫做兄弟关系，兄弟节点之间具有相同的孩子节点。

1.2.2 *node 的类型*

node 的类型按其在 node 图中的位置以及自身的特点分为 INTERNAL、INPUT、PRE_INPUT、PROCESS 等四种类型：

INTERNAL：为 node 图中的 node；

INPUT：为 node 图提供报文输入的 node，这类 node 在收包线程中每次循环都被调用，比如收包 node，每次循环都被调用以从网卡收包；

PRE_INPUT：为在 INPUT 类型的 node 之前需要被调用的 node；

PROCESS：为可以被挂起和重新被执行的 node。

1.2.3 *node 的注册*

node 的注册，即在 node 图中添加一个新的 node，vpp 提供宏 VLIB_REGISTER_NODE 用来注册 node，下面以 ip4-frag 这个 node 为例说明 node 的注册需要填写的内容：

```
VLIB_REGISTER_NODE (ip4_frag_node) = {
  .function = ip4_frag,
  .name = IP4_FRAG_NODE_NAME,
  .vector_size = sizeof (u32),
  .format_trace = format_ip_frag_trace,
  .type = VLIB_NODE_TYPE_INTERNAL,

  .n_errors = IP_FRAG_N_ERROR,
  .error_strings = ip4_frag_error_strings,

  .n_next_nodes = IP4_FRAG_N_NEXT,
  .next_nodes = {
    [IP4_FRAG_NEXT_IP4_LOOKUP] = "ip4-lookup",
    [IP4_FRAG_NEXT_IP6_LOOKUP] = "ip6-lookup",
    [IP4_FRAG_NEXT_ICMP_ERROR] = "ip4-icmp-error",
    [IP4_FRAG_NEXT_DROP] = "ip4-drop"
  },
};
```

function：为该 node 对应的执行函数，每个 node 都需要提供一个 function 来做具体的事情；

name：node 的名字；

type：node 的类型；

vector_size：存放一个包需要的字节数

format_trace：是一个函数，show trace 命令会调用每个 node 的 format_trace 函数来输出改 node 的信息，每个 node 自己实现该函数，自己决定在输入 show trace 时需要输出的和该 node 相关的信息；

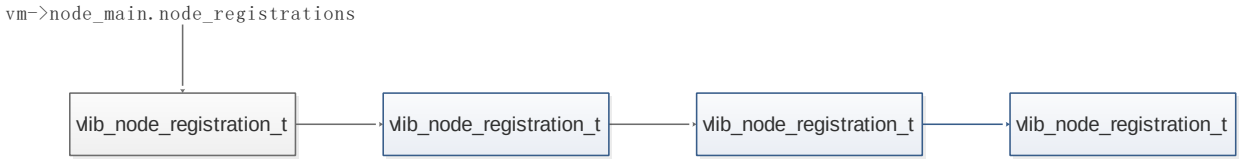
n_errors、error_strings：这两个主要是为了实现错误统计计数而存在的，n_errors 为错误统计个数，error_strings 为具体的错误信息，是一个字符串数组；

n_next_nodes、next_nodes：孩子节点的信息，n_next_nodes 为孩子节点的个数、next_nodes 为具体

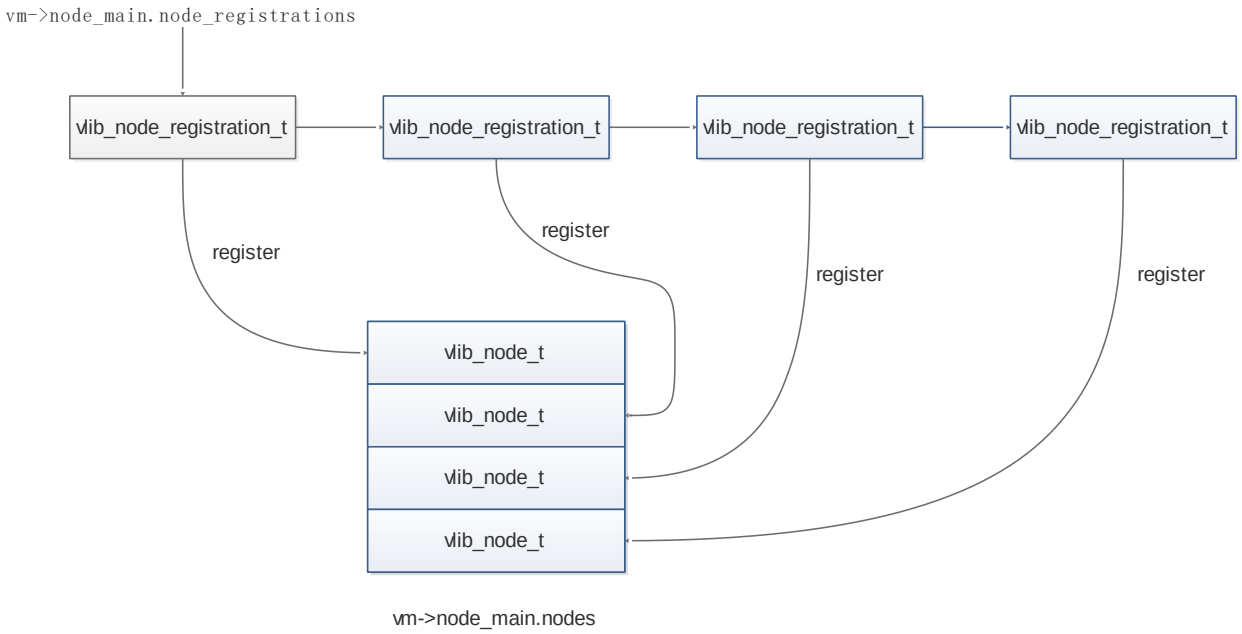
的孩子节点的名称，是一个字符串数组。再来看看 VLIB_REGISTER_NODE 这个宏的实现：

```
#define VLIB_REGISTER_NODE(x,...)
    __VA_ARGS__ vlib_node_registration_t x;
static void __vlib_add_node_registration_##x (void)
    __attribute__((__constructor__)) ;
static void __vlib_add_node_registration_##x (void)
{
    vlib_main_t * vm = vlib_get_main();
    x.next_registration = vm->node_main.node_registrations;
    vm->node_main.node_registrations = &x;
}
__VA_ARGS__ vlib_node_registration_t x
```

可知，该宏展开就是一个名为__vlib_add_node_registration_##x 的函数，该函数就是把类型为 vlib_node_registration_t 的实例 x 添加到 vm->node_main.node_registrations 链表上，vlib_node_registration_t 结构正是 node 的注册结构。需要注意的是该函数有 __attribute__((__constructor__))修饰，为 GNU C 的函数属性，表示该函数为构造函数，在 main 函数之前被执行。最终所有 node 的注册结构由 vm->node_main.node_registrations 这个链表串起来：



总结一下，所谓 node 的注册是在 main 函数执行之前通过宏 VLIB_REGISTER_NODE 只是定义并初始化一个类型为 vlib_node_registration_t 的结构，并把该结构插入到链表 vm->node_main.node_registrations 上。其实 node 的注册还未完成，还需要在 main 函数里再做进一步的“注册”，main 函数里的注册则是把以上建立好的链表中的所有 vlib_node_registration_t 结构分配 vlib_node_t 结构。vlib_node_registration_t 只是注册时存放用户提供的 node 相关信息的结构，而 vlib_node_t 才是真正的 node 对应的结构，转流程中访问的也是 vlib_node_t 结构。如下图所示：



vm->node_main.nodes 是一个 vector 结构（见 vector 分析章节，这里先认为 vector 就是一个数组）。在分析 vm->node_main.nodes 的初始化源码之前有必要先了解一下 vlib_node_registration_t 和 vlib_node_t 两个结构中每个字段的含义，如下表所示，建议第一次看的时候先大概浏览一下，后面结合具体的代码流程再返回来查阅。

了解 node 相关结构之后再看下 node 的注册代码流程：

```
void
vlib_register_all_static_nodes (vlib_main_t * vm)
{
    vlib_node_registration_t *r;

    static char *null_node_error_strings[] =
    {
        "blackholed packets",
    };

    static vlib_node_registration_t null_node_reg =
    {
        .function = null_node_fn,
        .vector_size = sizeof (u32),
        .name = "null-node",
        .n_errors = 1,
        .error_strings = null_node_error_strings,
    };

    /* make sure that node index 0 is not used by
       real node */
    register_node (vm, &null_node_reg);

    r = vm->node_main.node_registrations;
    while (r)
    {
        register_node (vm, r);
        r = r->next_registration;
    }
}
```

由上面的代码可知，先定义一个名为“null-node”的 node，并调用 register_node 函数注册该 node，最后遍历 vm->node_main.node_registrations 调用 register_node 函数注册该链表上的所有 node。再来 register_node 函数的实现，由于该函数稍微有点长，我们分段来分析：

```
static void
register_node (vlib_main_t * vm, vlib_node_registration_t * r)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_node_t *n;
    u32 page_size = clib_mem_get_page_size ();
    int i;

    if (CLIB_DEBUG > 0)
    {
        /* Default (0) type should match INTERNAL. */
        vlib_node_t zero = { 0 };
        ASSERT (VLIB_NODE_TYPE_INTERNAL == zero.type);
    }

    ASSERT (r->function != 0);

    n = clib_mem_alloc_no_fail (sizeof (n[0]));
    memset (n, 0, sizeof (n[0]));
    n->index = vec_len (nm->nodes);

    vec_add1 (nm->nodes, n);
```

这段主要是为 node 分配内存、和 index，并把 node 的指针加入 vm->node_main.nodes 数组中，注意加入数组的是 node 的指针，不是 node 结构，可以看到 vm->node_main.nodes 是一个二级指针。函数 clib_mem_alloc_no_fail 是给 node 分配一块 node 结构大小的内存，并保证不会失败，这个函数的实现在内存管理模块来分析。index 就是该 node 在 vm->node_main.nodes 数组中的索引。

```
  /* Name is always a vector so it can be formatted with %v. */
  if (clib_mem_is_heap_object (vec_header (r->name, 0)))
      n->name = vec_dup ((u8 *) r->name);
  else
      n->name = format (0, "%s", r->name);

  if (!nm->node_by_name)
      nm->node_by_name = hash_create_vec ( /* size */ 32,
                                          sizeof (n->name[0]), sizeof (uword));

  /* Node names must be unique. */
  {
      vlib_node_t *o = vlib_get_node_by_name (vm, n->name);
      if (o)
          clib_error ("more than one node named `%v'", n->name);
  }

  hash_set (nm->node_by_name, n->name, n->index);
```

以 node 的名称作为 key，index 作为 value 加入 hash 表 node_by_name 中，第一个 node 注册的时候，该 hash 表还没有创建，需要调用 hash_create_vec 来创建。

```
    n->sibling_of = r->sibling_of;
    if (r->sibling_of && r->n_next_nodes > 0)
        clib_error ("sibling node should not have any next nodes `%v'", n->name);

    if (r->type == VLIB_NODE_TYPE_INTERNAL)
        ASSERT (r->vector_size > 0);

#define _(f) n->f = r->f

    _(type);
    _(flags);
    _(state);
    _(scalar_size);
    _(vector_size);
    _(format_buffer);
    _(unformat_buffer);
    _(format_trace);
    _(validate_frame);
```

复制 vlib_node_registration_t 结构的各个字段到 vlib_node_t 结构里，sibling_of 字段表示这个 node 和该字段指向的 node 为兄弟关系，所以本 node 不能有孩子节点（兄弟关系的 node 拥有相同的孩子节点，所以兄弟节点之间只有一个兄弟节点有孩子节点）。

```
/* Register error counters. */
vlib_register_errors (vm, n->index, r->n_errors, r->error_strings);
node_elog_init (vm, n->index);
```

这两句分别是 error 和 elog 的初始化，这里先不详细介绍，留到专门的章节去介绍。

```
_(runtime_data_bytes);
if (r->runtime_data_bytes > 0)
{
    vec_resize (n->runtime_data, r->runtime_data_bytes);
    if (r->runtime_data)
        clib_memcpy (n->runtime_data, r->runtime_data, r->runtime_data_bytes);
}
```

runtime_data 可以理解为指向存放和该 node 相关的私有数据的结构的指针，runtime_data_bytes 表示该私有结构的长度。node 执行时可以读取该结构的信息，根据这些信息对报文做相应的处理。该结构里的数据由用户提供，这里只需要把它拷贝到 vlib_node_t 结构里即可。

```
vec_resize (n->next_node_names, r->n_next_nodes);
for (i = 0; i < r->n_next_nodes; i++)
    n->next_node_names[i] = r->next_nodes[i];

vec_validate_init_empty (n->next_nodes, r->n_next_nodes - 1, ~0);
vec_validate (n->n_vectors_by_next_node, r->n_next_nodes - 1);

n->owner_node_index = n->owner_next_index = ~0;
```

注意，数组 next_node_names 中的元素是指针类型，指向孩子节点的名称。n->next_nodes、n->n_vectors_by_next_node 这两个数组也是用来存放孩子节点信息的，这里只是初始化他们的长度为孩子节点的个数，具体存放什么信息在节点图章节再介绍。owner_node_index 记录的是当前往该 node 发包的父节点的 index，owner_next_index 记录，这两个字段初始值为~0。

接下来就是 PROCESS 类型的 node（以下简称 process）的特殊处理了，这里先大概介绍一下 process，详细的内容在单独的章节来介绍。process 和一般的 node 相比具有以下特点：

1、 只能在 thread0 上执行，即在 main 核上执行，不能在 worker 线程上执行，多个 process 共享 main 的时钟。

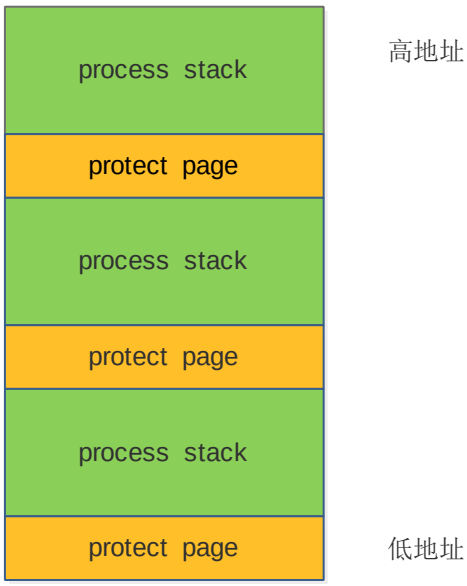
2、 能被挂起和恢复执行，恢复执行的时候要从被挂起的地方接着执行，因此需要给 process 创建单独的栈用来在被挂起时保存现场，被恢复执行时恢复现场。

process 可以简单得理解成操作系统中进程的概念，对于操作系统而言，其上面跑的进程都具有独立的栈空间，都被分时得调度。

```
        if (n->type == VLIB_NODE_TYPE_PROCESS)
        {
            vlib_process_t *p;
            uword log2_n_stack_bytes;

            log2_n_stack_bytes = clib_max (r->process_log2_n_stack_bytes, 15);

#ifdef CLIB_UNIX
            if ((page_size > (4 << 10)) && log2_n_stack_bytes < 19)
            {
                if ((1 << log2_n_stack_bytes) <= page_size)
                    log2_n_stack_bytes = min_log2 (page_size) + 1;
                else
                    log2_n_stack_bytes++;
            }
#endif

            p = clib_mem_alloc_aligned_at_offset
                (sizeof (p[0]) + (1 << log2_n_stack_bytes),
                 STACK_ALIGN, STRUCT_OFFSET_OF (vlib_process_t, stack),
            if (p == 0)
                clib_panic ("failed to allocate process stack (%d bytes)",
                            1 << log2_n_stack_bytes);

            memset (p, 0, sizeof (p[0]));
            p->log2_n_stack_bytes = log2_n_stack_bytes;

            n->runtime_index = vec_len (nm->processes);
            vec_add1 (nm->processes, p);
            p->stack[0] = VLIB_PROCESS_STACK_MAGIC;
            rt = &p->node_runtime;

#ifdef CLIB_UNIX
            if (mprotect (p->stack, page_size, PROT_READ) < 0)
                clib_unix_warning ("process stack");
#endif

        }
```

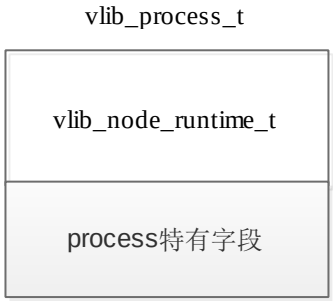
这段代码正是给 PROCESS 类型的 node 分配栈空间，PROCESS 类型的 node 用 vlib_process_t 结构来描述其相关的特性，该结构主要保存着 process 的调度信息，包括运行状态和保存的现场信息等。process 的栈空间大小可以在注册时通过 process_log2_n_stack_bytes 来指定，最小值和默认值都是 2^15 字节，即 32kB。以上两端 CLIB_UNIX 包含的内容意思是如果在 unix 环境则需要在栈底添加一个页大小的保护空间，也通常说的保护页。当栈溢出时就会访问到该保护页，而该保护页是没有

读写权限的，因此会被系统捕获到栈溢出的异常。另外，向栈底写入 magic 值 VLIB_PROCESS_STACK_MAGIC，用于来判断栈是否被写穿。



process 的栈空间是紧跟着 vlib_process_t 结构之后的，由最后一个成员 stack[0]指向，数组 vm->node_main.processes 存放指向所有 process 的指针。和 process 一样，其他的 node 也需要一个结构来保存运行时的数据，比如该 node 对应的执行函数、当前的状态、异常信息等。用来保存这些数据的结构叫做 vlib_node_runtime_t ，vlib_node_runtime_t 结构如下表描述：

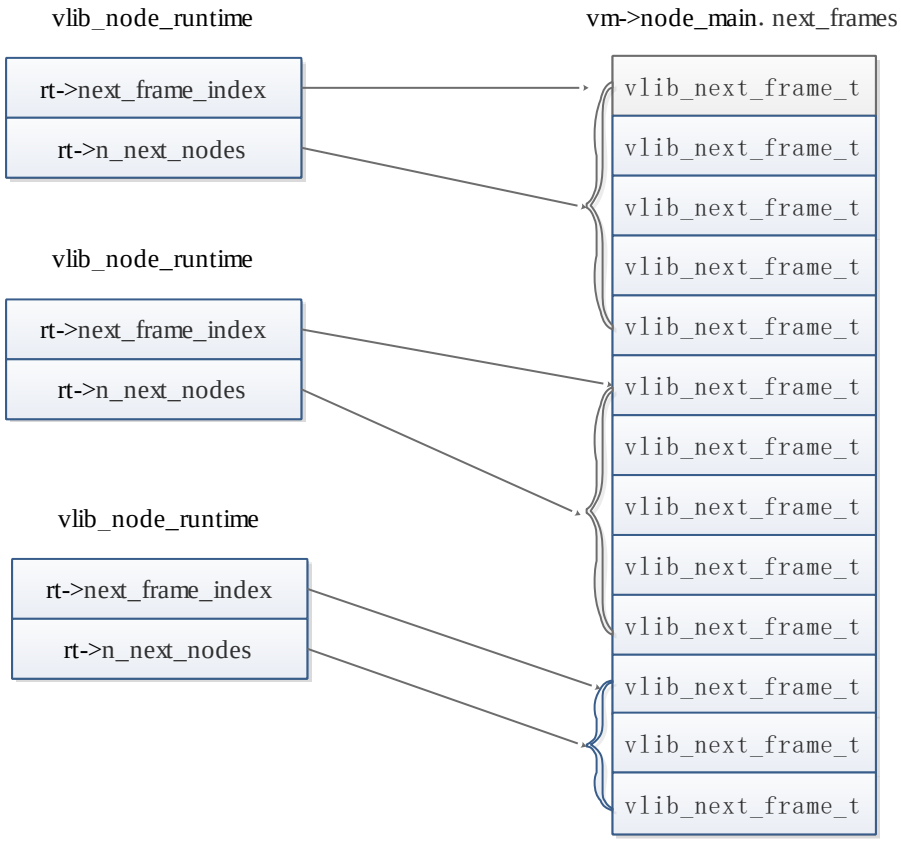
process 除了需要保存一般 node 的数据外，还需要保存现场等额外的数据。因此实现上，process 相关的数据保存在 vlib_process_t 结构中，在 vlib_process_t 结构中包含 vlib_node_runtime_t 结构：



process 的 vlib_node_runtime_t 结构嵌套在 vlib_process_t 结构中，而 vlib_process_t 结构从全局的数组 vm->node_main.processes 中分配。一般 node 的 vlib_node_runtime_t 从二维数组 vm->node_main. nodes_by_type[]中分配，第一维是 node 的类型。不管是从 vm->node_main.processes 还是从 vm->node_main. nodes_by_type[]中分配的 vlib_process_t 或 vlib_node_runtime_t 结构，其索引都保存在 vlib_node_t 结构的 runtime_index 字段里，即 runtime_index 字段对于 process 就是 vlib_process_t 结构的索引，对于一般 node 就是 vlib_node_runtime_t 结构的索引。

```
rt->function = n->function;
rt->flags = n->flags;
rt->state = n->state;
rt->node_index = n->index;

rt->n_next_nodes = r->n_next_nodes;
rt->next_frame_index = vec_len (nm->next_frames);

vec_resize (nm->next_frames, rt->n_next_nodes);
for (i = 0; i < rt->n_next_nodes; i++)
    vlib_next_frame_init (nm->next_frames + rt->next_frame_index + i);

vec_resize (rt->errors, r->n_errors);
for (i = 0; i < vec_len (rt->errors); i++)
    rt->errors[i] = vlib_error_set (n->index, i);

STATIC_ASSERT_SIZEOF (vlib_node_runtime_t, 128);
ASSERT (vec_len (n->runtime_data) <= VLIB_NODE_RUNTIME_DATA_SIZE);

if (vec_len (n->runtime_data) > 0)
    clib_memcpy (rt->runtime_data, n->runtime_data,
                 vec_len (n->runtime_data));

vec_free (n->runtime_data);
```

分配 vlib_node_runtime_t 结构之后对其进行初始化，function、flags、state、node_index、runtime_data 字段直接从 vlib_node_t 结构中拷贝。其中 runtime_data 就是上面所提到的 node 运行时的私有数据保存结构，直接从 vlib_node_t 中拷贝到 vlib_node_runtime_t 结构中即可，node 运行时访问的是 vlib_node_runtime_t 中的 runtime_data，而 vlib_node_t 中的 runtime_data 只是暂时保存而已，因此，拷贝完之后就被释放掉了。vlib_node_runtime_t 结构中的 next_frame_index 字段和 errors 字段需要根据孩子节点个数和 error 个数来初始化，next_frame_index 是 vlib_next_frame_t 结构的索引，vlib_next_frame_t 结构是为每个孩子节点分配的用来描述送给孩子节点处理 frame 的结构，细节在 node 的调度流程里描述。vlib_next_frame_t 结构是从全局的 vm->node_main. next_frames 数组中分配的， rt->next_frame_index 是本 node 的第一个 vlib_next_frame_t 结构在 vm->node_main. next_frames 数组中位置，n_next_nodes 字段是 vlib_node_runtime_t 结构的个数。如下图描述：



rt->errors 用于存放错误统计计数信息，这里不介绍了，在 vlib 章节再介绍。

1.2.4 *node 图的初始化*

node 图的初始化并不是创建一张真正的有向图，而是要根据已经注册的 node 建立所有 node 之间的关联关系，比如父子关系和兄弟关系，以便报文处理流程中能够正确得将报文交付给一系列的 node 来处理。代码上对应的函数为 vlib_node_main_init，该函数逻辑也比较清晰，主要分成四段，第一段处理兄弟节点；第二段处理孩子节点；第三段处理父亲节点；最后一段处理结构校验。下面进行源码详细分析：

```
nm->frame_size_hash = hash_create (0, sizeof (uword));
nm->flags |= VLIB_NODE_MAIN_RUNTIME_STARTED;
```

创建哈希表 vm->node_main.frame_size_hash，该哈希表用来存放回收的 vlib_frame_t 结构以便对 vlib_frame_t 结构的分配和释放进行管理，将在 node 的调度流程里详细描述。在 vm->node_main. Flags 打上标志位 VLIB_NODE_MAIN_RUNTIME_STARTED，表示 node 图已经初始化过了。

```
vlib_node_t *n, *sib;
uword si;

for (ni = 0; ni < vec_len (nm->nodes); ni++)
{
    n = vec_elt (nm->nodes, ni);

    if (!n->sibling_of)
        continue;

    sib = vlib_get_node_by_name (vm, (u8 *) n->sibling_of);
    if (!sib)
    {
        error = clib_error_create ("sibling `%s' not found for node `%v'",
                                   n->sibling_of, n->name);
        goto done;
    }

    clib_bitmap_foreach (si, sib->sibling_bitmap, ({
        vlib_node_t * m = vec_elt (nm->nodes, si);

        m->sibling_bitmap = clib_bitmap_ori (m->sibling_bitmap, n->index);
        n->sibling_bitmap = clib_bitmap_ori (n->sibling_bitmap, si);
    }));

    sib->sibling_bitmap = clib_bitmap_ori (sib->sibling_bitmap, n->index);
    n->sibling_bitmap = clib_bitmap_ori (n->sibling_bitmap, sib->index);
}
```

遍历所有节点，判断每个节点是否有兄弟节点，如果节点有兄弟节点则遍历兄弟节点的所有兄弟节点，并把兄弟节点的所有兄弟节点的 index 加到自己的位图 sibling_bitmap 中，同时把自己的 index 加到兄弟节点的所有兄弟节点的位图 sibling_bitmap 中，即兄弟的兄弟也是兄弟（类似于和兄弟的兄弟相互加下微信）位图 sibling_bitmap 记录一个 node 的所有兄弟节点的 index。最后把兄弟节点的 index 加到自己的位图 sibling_bitmap 中，把自己 index 加到兄弟节点的位图 sibling_bitmap 中。

再来看第二段代码：

```
for (ni = 0; ni < vec_len (nm->nodes); ni++)
{
    uword i;

    n = vec_elt (nm->nodes, ni);

    for (i = 0; i < vec_len (n->next_node_names); i++)
    {
        char *a = n->next_node_names[i];

        if (!a)
            continue;

        if (~0 == vlib_node_add_named_next_with_slot (vm, n->index, a, i))
        {
            error = clib_error_create
                    ("node `%v' refers to unknown node `%s'", n->name, a);
            goto done;
        }
    }

    vec_free (n->next_node_names);
}
```

第一层循环是遍历所有的 node，第二层循环是 node 的所有孩子节点，并调用函数 vlib_node_add_named_next_with_slot 把所有孩子节点添加到该 node 下，其实就是为 node 的每个孩子节点分配资源，并建立父子关系，所有 node 之间的父子关系便是一张 node 图。

vlib_node_add_named_next_with_slot 这个函数实现细节先跳过，先往下分析 vlib_node_main_init 函数的实现再回过头来看 vlib_node_add_named_next_with_slot。

```
for (ni = 0; ni < vec_len (nm->nodes); ni++)
{
    vlib_node_t *n_next;
    uword i;

    n = vec_elt (nm->nodes, ni);

    for (i = 0; i < vec_len (n->next_nodes); i++)
    {
        if (n->next_nodes[i] >= vec_len (nm->nodes))
            continue;

        n_next = vec_elt (nm->nodes, n->next_nodes[i]);
        n_next->prev_node_bitmap =
            clib_bitmap_ori (n_next->prev_node_bitmap, n->index);
    }
}
```

和上段代码一样两个循环，第一个循环是遍历所有的 node，第二个循环是遍历 node 的所有孩子节点，并把 node 的 index 加到孩子所有孩子节点的位图 prev_node_bitmap 中。即 prev_node_bitmap 记录的是所有父亲节点的 index。其中，n->next_nodes 是节点 n 的所有孩子节点的 index，n->next_nodes[i] >= vec_len (nm->nodes)这个判断的意思是如果孩子节点的 index 比 node 的个数还要

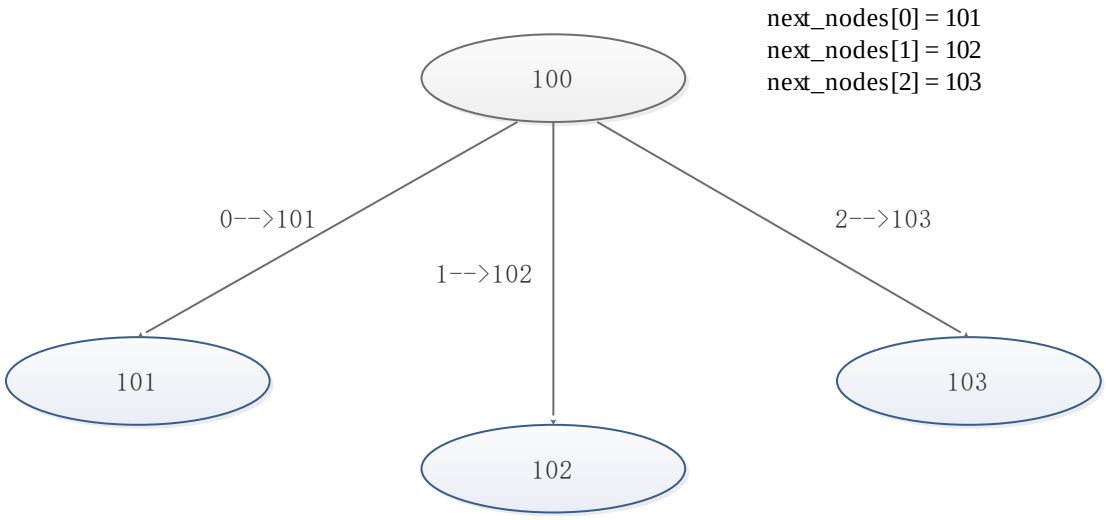
大，说明这个孩子节点的 index 是非法的，即这个孩子节点不存在，所以跳过。

```
vlib_next_frame_t *nf;
vlib_node_runtime_t *r;
vlib_node_t *next;
uword i;

vec_foreach (r, nm->nodes_by_type[VLIB_NODE_TYPE_INTERNAL])
{
    if (r->n_next_nodes == 0)
        continue;

    n = vlib_get_node (vm, r->node_index);
    nf = vec_elt_at_index (nm->next_frames, r->next_frame_index);

    for (i = 0; i < vec_len (n->next_nodes); i++)
    {
        next = vlib_get_node (vm, n->next_nodes[i]);

        ASSERT (nf[i].node_runtime_index == next->runtime_index);

        nf[i].flags = 0;
        if (next->flags & VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH)
            nf[i].flags |= VLIB_FRAME_NO_FREE_AFTER_DISPATCH;
    }
}
```

遍历所有类型为 INTERNAL 的 node，检查没有 node 的所有孩子节点对应的 vlib_next_frame_t 结构里的 node_runtime_index 字段的合法性，该字段存放的是孩子节点的 vlib_node_runtime_t 结构的索引。最后判断孩子节点是否有 VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH 标志位，有的话在对应的 vlib_next_frame_t 结构上打上与之对应的 VLIB_FRAME_NO_FREE_AFTER_DISPATCH 标志位。该标志位的意思是该 node 被调度之后不回收相应的 vlib_frame_t 结构。再来看以上第二段代码分析中遗留的函数 vlib_node_add_named_next_with_slot：

```
uword
vlib_node_add_named_next_with_slot (vlib_main_t * vm,
                                    uword node, char *name, uword slot)
{
    vlib_node_main_t *nm;
    vlib_node_t *n, *n_next;

    nm = &vm->node_main;
    n = vlib_get_node (vm, node);

    n_next = vlib_get_node_by_name (vm, (u8 *) name);
    if (!n_next)
    {
        if (nm->flags & VLIB_NODE_MAIN_RUNTIME_STARTED)
            return ~0;

        if (slot == ~0)
            slot = clib_max (vec_len (n->next_node_names),
                             vec_len (n->next_nodes));
        vec_validate (n->next_node_names, slot);
        n->next_node_names[slot] = name;
        return slot;
    }

    return vlib_node_add_next_with_slot (vm, node, n_next->index, slot);
}
```

参数 node 是父亲节点的 index，name 是孩子节点的名称，slot 是第几个孩子。这个函数的主要功能是把孩子节点的 index 添加到自己的 next_nodes 数组里并为孩子节点创建 vlib_pending_frame_t 结构，其实该数组就是一个映射表，把第几个孩子到该孩子节点的 index 做了一个映射。当需要把报文发送给孩子节点的时候通过这个映射表来找到孩子节点的 index，如下图所示，节点 100 有单个孩子节点，第 0 个孩子的节点 index 为 101，第一个孩子的节点 index 为 102，第二个孩子的节点 index 为 103：



由上面的分析可知，该函数是在 node 图初始化函数里被调用的，但该函数也可以在 node 图初始化之前被调用，如果是后者，允许添加一个不存在的孩子节点，因为该孩子节点可能还没来得及创建出来呢，因此把它先加到父节点的 next_node_names 数组的末尾。但是如果是前者，即 node 图已经初始化了，则孩子节点必须存在，否则就算加到 next_node_names 没被机会被初始化了。接着看函数 vlib_node_add_next_with_slot 的实现：

```
uword
vlib_node_add_next_with_slot (vlib_main_t * vm,
                              uword node_index,
                              uword next_node_index, uword slot)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_node_t *node, *next;
    uword *p;

    node = vec_elt (nm->nodes, node_index);
    next = vec_elt (nm->nodes, next_node_index);

    /* Runtime has to be initialized. */
    ASSERT (nm->flags & VLIB_NODE_MAIN_RUNTIME_STARTED);

    if ((p = hash_get (node->next_slot_by_node, next_node_index)))
    {
        /* Next already exists: slot must match. */
        if (slot != ~0)
            ASSERT (slot == p[0]);
        return p[0];
    }

    if (slot == ~0)
        slot = vec_len (node->next_nodes);
```

哈希表 next_slot_by_node 和上面介绍的数组 next_nodes 的功能是相反的，next_slot_by_node 是通过孩子节点的 index 找到该孩子节点是第几个孩子，而数组 next_nodes 则是查到第几个孩子节点的 index。next_nodes 是转发流程时使用，而 next_slot_by_node 则是用来判断孩子节点是否重复添加，转发流程并不会使用。以上代码段的意思是先查找孩子节点是否添加过，如果已经添加过了，直接返回该孩子节点是第几个孩子。参数 slot 指定添加的孩子节点想作为第几个孩子，如果该孩子节点已经存在但是和 slot 指定的不是相等，则返回失败。如果不是以上情况则可以正常添加，如果没有指定 slot，则当前孩子节点的个数作为 slot。

```
    vec_validate_init_empty (node->next_nodes, slot, ~0);
    vec_validate (node->n_vectors_by_next_node, slot);

    node->next_nodes[slot] = next_node_index;
    hash_set (node->next_slot_by_node, next_node_index, slot);

    vlib_node_runtime_update (vm, node_index, slot);
```

数组 n_vectors_by_next_node 用来记录本节点往每个孩子节点发包的个数，把孩子节点的 index 加到数组 next_nodes 中 slot 位置，把孩子节点的 index 作为 key，slot 作为 value 添加到哈希表 next_slot_by_node，到这里建立第几个孩子和孩子节点的 index 之间的双向映射关系结束了。先往下分析本函数剩余的代码再回来分析函数 vlib_node_runtime_update。

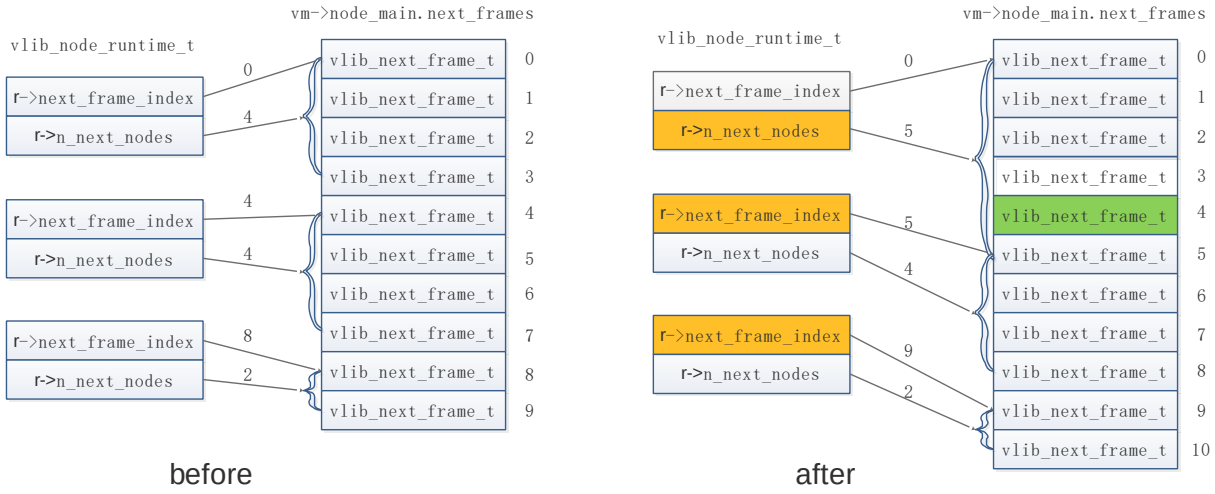
```
next->prev_node_bitmap = clib_bitmap_ori (next->prev_node_bitmap,
                         node_index);
{
    uword sib_node_index, sib_slot;
    vlib_node_t *sib_node;
    clib_bitmap_foreach (sib_node_index, node->sibling_bitmap, ({
        sib_node = vec_elt (nm->nodes, sib_node_index);
        if (sib_node != node)
        {
            sib_slot = vlib_node_add_next_with_slot (vm, sib_node_index, next_node_index, slot);
            ASSERT (sib_slot == slot);
        }
    }));
}

return slot;
```

把父节点的 index 加到孩子节点的位图 prev_node_bitmap 中，最后遍历父节点的所有兄弟节点，递归调用 vlib_node_add_next_with_slot 函数把孩子节点添加到所有所有兄弟节点的孩子节点中。

再次说明一下函数 vlib_node_add_named_next_with_slot 的任务主要有两个：一是建立第几个孩子和孩子节点的 index 之间的双向映射，也就是数组 next_nodes 和哈希表 next_slot_by_node；而是为孩子节点创建运行时需要的数据结构 vlib_next_frame_t 和 vlib_pending_frame_t。第一步已经完成，接下来看第二步，即函数 vlib_node_runtime_update 的实现：

```
vlib_worker_thread_barrier_sync (vm);

node = vec_elt (nm->nodes, node_index);
r = vlib_node_get_runtime (vm, node_index);

n_insert = vec_len (node->next_nodes) - r->n_next_nodes;
if (n_insert > 0)
{
    i = r->next_frame_index + r->n_next_nodes;
    vec_insert (nm->next_frames, n_insert, i);

    for (j = 0; j < n_insert; j++)
        vlib_next_frame_init (nm->next_frames + i + j);

    for (j = 0; j < vec_len (nm->nodes); j++)
    {
        s = vlib_node_get_runtime (vm, j);
        if (j != node_index && s->next_frame_index >= i)
            s->next_frame_index += n_insert;
    }

    vec_foreach (pf, nm->pending_frames)
    {
        if (pf->next_frame_index != VLIB_PENDING_FRAME_NO_NEXT_FRAME
                && pf->next_frame_index >= i)
            pf->next_frame_index += n_insert;
    }

    pool_foreach (pf, nm->suspended_process_frames, ({
        if (pf->next_frame_index != ~0 && pf->next_frame_index >= i)
        pf->next_frame_index += n_insert;
    }));

    r->n_next_nodes = vec_len (node->next_nodes);
}
```

vlib_next_frame_t 结构在分析 node 注册过程的时候已经介绍过了，这里用一张图来说明上面这段代码的意思：



添加一个孩子节点时要从 vm->node_main.next_frames 里为该孩子节点 vlib_next_frame_t 结构，同时包保证一个 node 的所有孩子节点对应的 vlib_next_frame_t 结构在数组 vm->node_main.next_frames 里是连续的。上图绿色表示插入的 vlib_next_frame_t 结构，黄色表示插入 vlib_next_frame_t 结构后变化的值。数组 vm->node_main. pending_frames 里存放的是 vlib_pending_frame_t 结构，该结构描述一个正在准备被调度的 node 和发送给该 node 去处理的一批报文。因此，数组 vm->node_main. pending_frames 存放的是即将被调度的 node 的信息，这里不详细介绍 vlib_pending_frame_t 结构，留到下一节 node 的调度来详细介绍。而数组 vm->node_main. suspended_process_frames 存放的是 process 的 vlib_pending_frame_t 结构，为什么分开存放，是因为 process 的调度和非 proces 节点的调度是分开的。

```
next_node = vlib_get_node (vm, node->next_nodes[next_index]);
nf = nm->next_frames + r->next_frame_index + next_index;
nf->node_runtime_index = next_node->runtime_index;

vlib_worker_thread_node_runtime_update ();
```

最后把 vlib_node_runtime_t 结构的 index 写入 vlib_next_frame_t 结构中的 node_runtime_index 字段，调用 vlib_worker_thread_node_runtime_update 设置 runtime 需要同步到 worker 线程的标志位。至此，node 图的初始化流程分析结束。

1.2.5 node 的调度

由于 process 本身的特点，process 和其他类型的 node 的调度方式不一样，实现上为两个不同的函数，分别为：dispatch_process 和 dispatch_node，但 node 使用统一的状态，只是各自有独立的 flags。

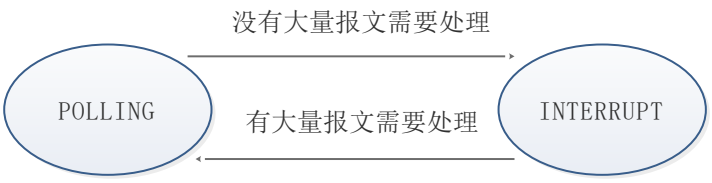
node 的状态：

POLLING：轮询方式，每次循环都会被调度

INTERRUPT：中断方式，只有收到中断信号之后才会被调度

DISABLED：只有非 INTERNAL 类型的 node 才会设置该状态，如果设置了 DISABLE 状态表示不再会被调度

POLLING 和 INTERRUPT 的思想和操作系统管理中 CPU 处理外设请求的方式类似，分别对应轮询方式和中断方式，轮询方式下 CPU 定时去读取外设的请求，如果有请求则处理，没有则等待下次轮询；中断方式下 CPU 不会去询问外设是否有请求需要处理，而是当外设有请求需要 CPU 来处理的时候，通过向 CPU 发出中断信号，让 CPU 暂停当前的事情来处理自己的请求。轮询方式的优势是处理请求效率高实现简单，缺点是浪费 CPU 的时钟；而中断方式则相反，即处理请求效率低，但 CPU 利用率高。为了充分发挥这种模式的优势，当 node 有较大量的报文需要处理时使用 POLLING 方式，否则使用 INTERRUPT 方式。并且两种方式可以根据当前处理的报文数量进行动态切换：



1.2.5.1 PRE_INPUT 和 INPUT 类型的 node 调度

PRE_INPUT 和 INPUT 类型的 node 为 INTERNAL 和 PROCESS 的 node 提供数据输入。PRE_INPUT 类型的 node 主要是为 PROCESS 类型的 node 提供数据，而 INPUT 类型的 node 则是为 INTERNAL 类型的 node 提供报文。INTERNAL 类型的 node 只负责处理报文就好了。所以，PRE_INPUT 类型的 node 只会在 main 线程上被调度，INPUT 类型的 node 可以在 main 线程和 worker 线程上被调度。

```
if (is_main)
    vec_foreach (n, nm->nodes_by_type[VLIB_NODE_TYPE_PRE_INPUT])
        cpu_time_now = dispatch_node(vm, n, VLIB_NODE_TYPE_PRE_INPUT,
                VLIB_NODE_STATE_POLLING,
                0, cpu_time_now);

vec_foreach (n, nm->nodes_by_type[VLIB_NODE_TYPE_INPUT])
    cpu_time_now = dispatch_node(vm, n, VLIB_NODE_TYPE_INPUT,
            VLIB_NODE_STATE_POLLING,
            0, cpu_time_now);
```

分别遍历所有的 PRE_INPUT 和 INPUT 类型的 node，调用函数 dispatch_node 来进行 node 的调度工作，详细看 dispatch_node 函数的实现：

```
static_always_inline u64 dispatch_node(vlib_main_t * vm,
        vlib_node_runtime_t * node, vlib_node_type_t type,
        vlib_node_state_t dispatch_state, vlib_frame_t * frame,
        u64 last_time_stamp) {
    uword n, v;
    u64 t;
    vlib_node_main_t *nm = &vm->node_main;
    vlib_next_frame_t *nf;

    if (CLIB_DEBUG > 0) {
        vlib_node_t *n = vlib_get_node(vm, node->node_index);
        ASSERT(n->type == type);
    }

    if (type != VLIB_NODE_TYPE_INTERNAL && node->state != dispatch_state) {
        ASSERT(type != VLIB_NODE_TYPE_INTERNAL);
        return last_time_stamp;
    }
```

参数 node 是本次调度的 node 的 vlib_node_runtime_t 结构，应该叫 rt 更合适；参数 type 是 node 的类型；参数 dispatch_state 是 node 的状态；参数 frame 是存放报文的结构，PRE_INPUT 和 INPUT 类型的 node 该参数为 0；参数 last_time_stamp 是上次调度完成的时间戳。第一段是确保这个 node 的类型和当前调度的类型是一致的。dispatch_state 函数是 PRE_INPUT、INPUT 和 INTERNAL 类型的 node 的调度函数，第二段判断如果不是 INTERNAL 类型的 node，则判断 node 的状态是不是当前调

度的状态如果不是则直接返回，比如 node 状态可能为 INTERRUPT，当前调度的状态为 POLLING，则不在此时调度该 node，而是调度 INTERRUPT 状态的地方调度。一般来说 INTERNAL 类型的 node 只负责处理报文，所以状态一直未 POLLING，不会切换成 INTERRUPT，所以 type!= VLIB_NODE_TYPE_INTERNAL 只判断 PRE_INPUT 和 INPUT 类型的 node。顺便地，第二个 if 语句里的 ASSERT 真是多此一举。

```
if ((type == VLIB_NODE_TYPE_PRE_INPUT || type == VLIB_NODE_TYPE_INPUT)
        && dispatch_state != VLIB_NODE_STATE_INTERRUPT) {
    u32 c = node->input_main_loops_per_call;
    if (c) {
        node->input_main_loops_per_call = c - 1;
        return last_time_stamp;
    }
}
```

对于 PRE_INPUT 和 INPUT 类型的 node 而言，有的 node 可能不需要调度得很平凡，比如输入数据很慢或者实时性要求不是很高的 node，没必要每次循环都被调度，这样可以减少 main 核的时钟开销。实现上通过字段 node->input_main_loops_per_call 来调整 node 的调度频率。该字段的含义是多少次循环才调度该 node 一次，其值由 node 自己设置，node 可以根据自己的需要在 node 对应的函数里动态来调整该值。这里判断该值如果大于 0 则减一后返回，不进行该 node 的调度，直到减为 0 后才调度该 node。

```
if (node->n_next_nodes > 0) {
    nf = vec_elt_at_index(nm->next_frames, node->next_frame_index);
    CLIB_PREFETCH(nf, 4 * sizeof(nf[0]), WRITE);
}
```

如果该 node 有孩子节点，则预取一下第一个孩子对应的 vlib_next_frame_t 结构，有可能会往该孩子发包，往该孩子发包就会用到对应的 vlib_next_frame_t 结构，先提前预取到 cache 里性能更好。

```
vm->cpu_time_last_node_dispatch = last_time_stamp;

if (1) {
    vlib_main_t *stat_vm;

    stat_vm = vm;

    vlib_elog_main_loop_event(vm, node->node_index, last_time_stamp,
            frame ? frame->n_vectors : 0, 0);

    if (VLIB_BUFFER_TRACE_TRAJECTORY && frame) {
        int i;
        u32 *from;
        from = vlib_frame_vector_args(frame);
        for (i = 0; i < frame->n_vectors; i++) {
            vlib_buffer_t *b = vlib_get_buffer(vm, from[i]);
            add_trajectory_trace(b, node->node_index);
        }
        n = node->function(vm, node, frame);
    } else
        n = node->function(vm, node, frame);

    t = clib_cpu_time_now();
    vlib_elog_main_loop_event(vm, node->node_index, t, n, 1);

    vm->main_loop_vectors_processed += n;
    vm->main_loop_nodes_processed += n > 0;

    v = vlib_node_runtime_update_stats(stat_vm, node, 1, n, t - last_time_stamp);
```

这里先不介绍 stats、elog 等内容，留到 vlib 去介绍。如果 frame 不为 0，即有报文需要本 node 处理，且宏 VLIB_BUFFER_TRACE_TRAJECTORY 开启，则调用 add_trajectory_trace 处理每个报文。add_trajectory_trace 函数的默认处理是把当前 node 的 index 写入报文的私有字段中去，这样报文经过的所有 node 的 index 都能查到，调试时方便跟踪报文经过的 node。最后调用 node 的 function 函数完成 node 的调度。上面介绍过 node 的三种状态，其中 POLLING 和 INTERRUPT 状态是可以转换的，一下代码正式实现这两个状态的转换：

```
if ((dispatch_state == VLIB_NODE_STATE_INTERRUPT)
        || (dispatch_state == VLIB_NODE_STATE_POLLING
                && (node->flags & VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE))) {
    if ((dispatch_state == VLIB_NODE_STATE_INTERRUPT && v >= nm->polling_threshold_vector_length)
            && !(node->flags & VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE)) {
        vlib_node_t *n = vlib_get_node(vm, node->node_index);
        n->state = VLIB_NODE_STATE_POLLING;
        node->state = VLIB_NODE_STATE_POLLING;
        node->flags &= ~VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE;
        node->flags |= VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE;
    } else if (dispatch_state == VLIB_NODE_STATE_POLLING
            && v <= nm->interrupt_threshold_vector_length) {
        vlib_node_t *n = vlib_get_node(vm, node->node_index);
        if (node->flags & VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE) {
            n->state = VLIB_NODE_STATE_INTERRUPT;
            node->state = VLIB_NODE_STATE_INTERRUPT;
            node->flags &= ~VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE;
        } else {
            node->flags |= VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE;
        }
    }
}
```

这里为了方便讲解，把 elog 相关的代码先剪掉了。之前这两个状态的含义已经介绍过了，这里不再重复介绍了。如果当前为 INTERRUPT 状态，或者虽然为 POLLING 状态，但是 node 打上标志位

VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE，则需要判断该 node 是否满足切换成 POLLING 状态的条件，满足条件为 main 循环 2^VLIB_LOG2_MAIN_LOOPS_PER_STATS_UPDATE（默认为 7）次该 node 处理的报文的个数超过 nm->polling_threshold_vector_length（默认为 10），也就是说 main 循环 128 次后该 node 处理的报文个数超过 10 个，则认为 node 有大量的报文需要处理，需要切换 node 的状态成 POLLING 来提高该 node 的报文处理效率。如果满足条件，则把该 node 的 vlib_node_t 和 vlib_node_runtime_t 结构里的 state 改成 POLLING，并在 vlib_node_runtime_t 结构的 flags 字段设置标志位 VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE，同时清除标志位 VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE。

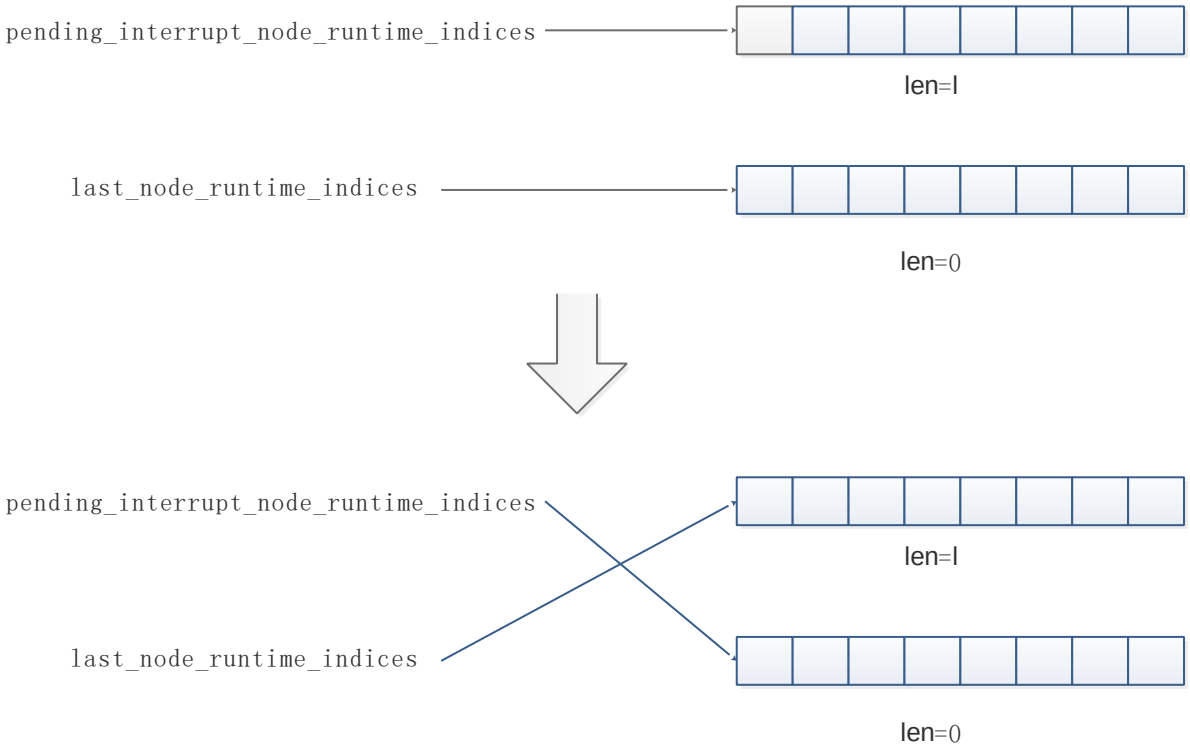
VLIB_NODE_FLAG_SWITCH_FROM_INTERRUPT_TO_POLLING_MODE 表示从 INTERRUPT 状态切换成 POLLING 状态，

VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE 表示即将从 POLLING 状态切换成 INTERRUPT，或从 POLLING 状态切换成 INTERRUPT。为什么说即将呢？因为从 INTERRUPT 状态切换成 POLLING 状态是直接切换的，但是从 POLLING 状态切换成 INTERRUPT 不是直接切换的，而是先打上 VLIB_NODE_FLAG_SWITCH_FROM_POLLING_TO_INTERRUPT_MODE 标志位，等到下次调度再切换成 INTERRUPT，else if 里的代码正式这个意思。可能是因为 POLLING 状态需要高效处理，所以从 INTERRUPT 状态切换成 POLLING 状态无需犹豫，而从 POLLING 状态切换成 INTERRUPT 状态需要慎重考虑吧。那么 INTERRUPT 状态的 node 在什么地方以及如何调度的呢？

```
uword l = _vec_len(nm->pending_interrupt_node_runtime_indices);
uword i;
if (l > 0) {
    u32 *tmp;

    if (!is_main)
        clib_spinlock_lock(&nm->pending_interrupt_lock);

    tmp = nm->pending_interrupt_node_runtime_indices;
    nm->pending_interrupt_node_runtime_indices = last_node_runtime_indices;
    last_node_runtime_indices = tmp;
    _vec_len (last_node_runtime_indices) = 0;

    if (!is_main)
        clib_spinlock_unlock(&nm->pending_interrupt_lock);

    for (i = 0; i < l; i++) {
        n = vec_elt_at_index(nm->nodes_by_type[VLIB_NODE_TYPE_INPUT],
                last_node_runtime_indices[i]);
        cpu_time_now = dispatch_node(vm, n, VLIB_NODE_TYPE_INPUT,
                VLIB_NODE_STATE_INTERRUPT, 0, cpu_time_now);
    }
}
```

数组 nm->pending_interrupt_node_runtime_indices 用来存放需要通过中断方式来调度的 node 的 index。

类似操作系统对外设的管理，当外设需要 CPU 处理数据的时候需要通过中断的方式来告诉 CPU，类似的当 INTERRUPT 状态的 node 需要 main 或者 worker 线程调度它的时候就往该数组里写入自己的 node 的 index 即可。考虑到调度该数组的 node 的过程中有可能还有其他 node 加入到该数组中，因此，先用一个临时数组来和该数组交换，如果调度过程中有 node 加入该数组，则加入到临时数组，等下次循环再来调度新加入的 node，好处是减少拷贝。



INPUT 类型的 node 可以在 main 线程或者 worker 线程上跑，如果是在 worker 线程上跑，由于 worker 线程有多个，所以需要加锁，但是 nm->pending_interrupt_node_runtime_indices 和 last_node_runtime_indices 都是线程本地变量，所以理论上应该不需要加锁，但这里加锁了，为什么呢？最后，遍历数组中的所有 node 调用函数 dispatch_node 进行追个调度执行，函数的第四个参数为 VLIB_NODE_STATE_INTERRUPT 表示调度的是 INTERRUPT 状态的节点。该函数上面已经分析过了。

1.2.5.2 INTERNAL 类型的 node 调度

上节分析了 PRE_INPUT 和 INPUT 类型的 node 的调度流程，PRE_INPUT 和 INPUT 类型的 node 是给 INTERNAL 类型的 node 产生输入数据的 node，如 INPUT 类型的 dpdk-input 节点，该节点会从网卡收包，然后发送给 INTERNAL 类型的节点去处理。也就是说 INTERNAL 类型的 node 不像 PRE_INPUT 和 INPUT 类型的 node 一样每次循环都会被调度，而是 INPUT 类型的节点或者其他 INTERNAL 类型的 node 执行之后给它发送报文，让它去处理的时候它才会被调度到，这是 INTERNAL 类型的 node 的一个特点。那么，在介绍 INTERNAL 类型的 node 的调度过程之前，我

们需要了解 INPUT 类型的 node 是如何把报文发送给 INTERNAL 类型的 node 的。整个过程可以分成以下几步：

1、 确定把报文发送给第几个孩子节点

确定把报文送给哪个孩子节点处理，是由该 node 的业务逻辑决定的，通常有两种方式：一是通过解析报文的内容来决定，比如 ethernet-input 节点通过解析报文是 ipv4 还是 ipv6 报文把报文发送给 ip4-input 或 ip6-input 去处理；而是通过查找转发表来决定，比如 ip4-lookup 通过查找路由表来决定把报文发送给 ip4-arp 还是 ip4-rewrite 节点来处理。

2、 找到该孩子节点对应的用于存放报文的结构 vlib_frame_t

根据前面 node 图的初始化章节的分析可知，确定把报文发送给第几个孩子节点之后，就可以获取该孩子节点对应的 vlib_next_frame_t 结构了。vlib_next_frame_t 结构中的 frame_index 字段指向 vlib_frame_t 结构，vlib_frame_t 结构是存放报文的地方。

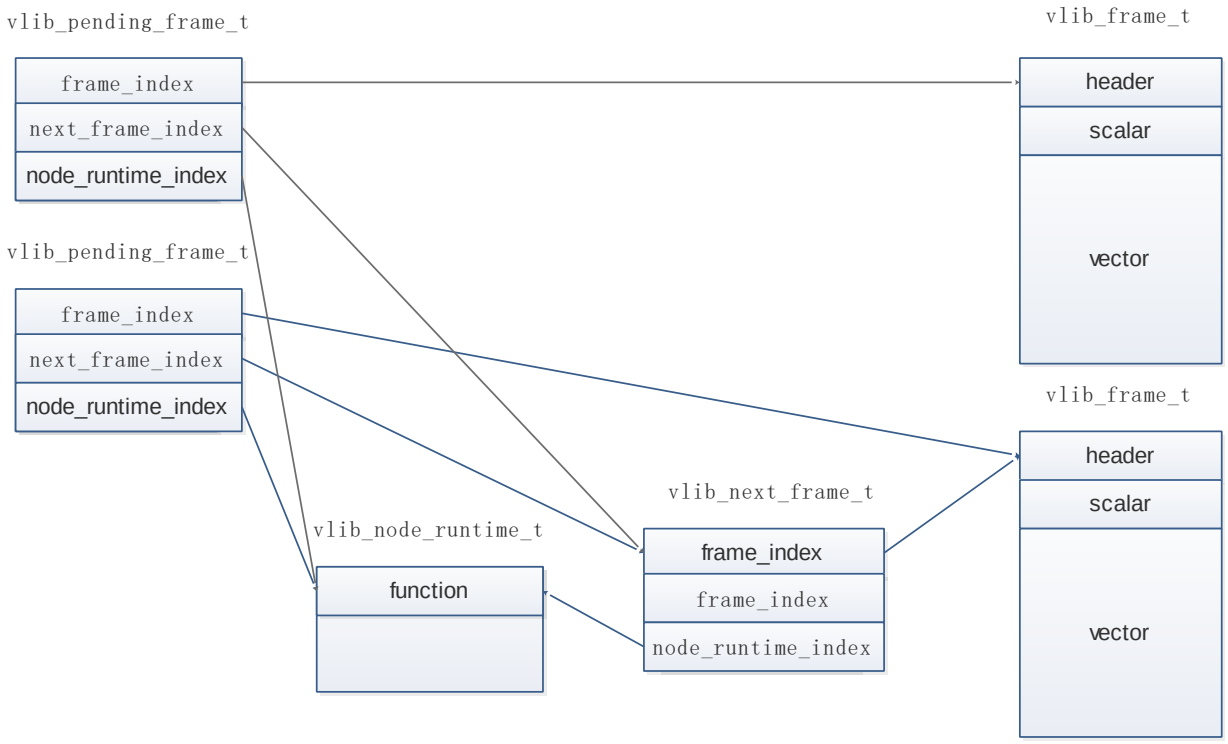
3、 把报文放入 vlib_frame_t 结构

vlib_frame_t 结构最后一个字段为可变数组，初始化的时候会分配一块内存，用于存放报文的索引。

4、 创建 vlib_pending_frame_t 结构，并把它加入数组 vm->node_main. pending_frames 等待调度

vlib_pending_frame_t 结构记录报文所在的结构 vlib_next_frame_t 的 index，以及处理这些报文的 node 的 vlib_node_runtime_t 结构的索引，这样通过 vlib_pending_frame_t 结构里面的信息就可以把报文分发给指定的 node 处理了。

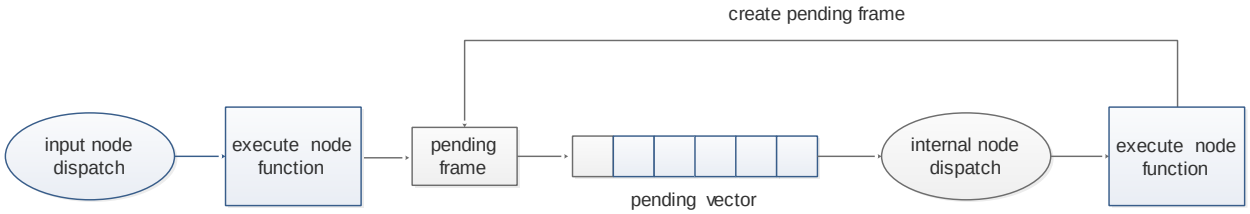
以上过程中涉及到的数据结构，以及他们之间的关系如下图所示：



vlib_pending_frame_t 是为调度 INTERNAL 类型的 node 提供的结构体，main 或者 worker 线程在主

循环里一直遍历数组 vm->node_main. pending_frames，当数组不为空时则获取数组里的 vlib_pending_frame_t 结构，调用函数 dispatch_pending_node 进行 INTERNAL 类型的 node 的调度。vlib_node_runtime_t 在 noe 图初始化的时候已经介绍过了，它是保存 node 运行时需要的数据的结构，保存的数据主要有 node 的执行函数、node 的状态，node 运行时的私有数据等。vlib_frame_t 是正真存放报文索引的结构，字段 n_vectors 表示当前存放的报文的个数。那么 vlib_next_frame_t 是干嘛的呢？能不能不要这个结构呢？由于一个 vlib_frame_t 结构可存放的报文的个数是有线的（默认是 256），如果给孩子节点发送的报文超过 256 个怎么办呢？那就需要多个 vlib_frame_t 结构来存放，也就是说给孩子节点发送报文的时候可能需要网多个 vlib_frame_t 结构里放报文。那么，如何知道应该往哪个 vlib_frame_t 结构里放呢？一种思路是使用一个结构来记录当前应该往哪个 vlib_frame_t 结构放报文，这个 vlib_frame_t 结构如果放满了需要重新分配一个新的 vlib_frame_t 结构，并更新记录。这就是 vlib_next_frame_t 结构所做的事情了。该结构每个孩子节点一一对应，其主要字段是 frame_index 和 node_runtime_index。

整个流程如下图所示：



从以上分析可知，其实把处理完之后的报文发送给孩子节点处理的流程是：首先要找到孩子节点对应的 vlib_next_frame_t 结构，通过 vlib_next_frame_t 结构里的 frame_index 找到 vlib_frame_t 结构；把报文的索引放入 vlib_frame_t 结构里；最后创建 vlib_pending_frame_t 结构，该结构里的 frame_index 字段指向存放报文的 vlib_frame_t 结构，node_runtime_index 字段指向孩子节点对应的 vlib_node_runtime_t 结构，并把创建 vlib_pending_frame_t 结构加入等待被调度的数组 vm->node_main. pending_frames 中；main 或者 worker 线程在主循环里遍历该数组，取出 vlib_pending_frame_t 结构，调用函数 dispatch_pending_node 进行调度。接下来分析一下整个流程对应的源码，首先是找到孩子节点对应的 vlib_next_frame_t 结构，对应的代码是宏 vlib_get_next_frame：

```
#define vlib_get_next_frame(vm,node,next_index,vectors,n_vectors_left)  \
  vlib_get_next_frame_macro (vm, node, next_index,                      \
                             vectors, n_vectors_left,                   \
                             /* alloc new frame */ 0)
```

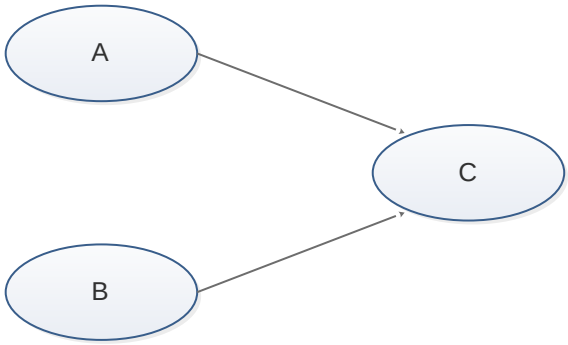
宏 vlib_get_next_frame 又是宏 vlib_get_next_frame_macro

```
#define vlib_get_next_frame_macro(vm,node,next_index,vectors,n_vectors_left,alloc_new_frame) \
do {                                                                    \
  vlib_frame_t * _f                                                     \
    = vlib_get_next_frame_internal ((vm), (node), (next_index),     \
                    (alloc_new_frame));                                 \
  u32 _n = _f->n_vectors;                                               \
  (vectors) = vlib_frame_vector_args (_f) + _n * sizeof ((vectors)[0]); \
  (n_vectors_left) = VLIB_FRAME_SIZE - _n;                              \
} while (0)
```

调用函数 vlib_get_next_frame_internal 获取 vlib_frame_t 结构，从函数名称上看像是获取 vlib_next_frame_t 结构，实际上返回的是 vlib_frame_t 结构，一步到位了。最后是计算 vlib_frame_t 结构中下一个存放报文的位置，以及可存放的报文个数。继续分析函数 vlib_get_next_frame_internal

```
vlib_frame_t *
vlib_get_next_frame_internal(vlib_main_t * vm, vlib_node_runtime_t * node,
        u32 next_index, u32 allocate_new_next_frame) {
    vlib_frame_t *f;
    vlib_next_frame_t *nf;
    u32 n_used;

    nf = vlib_node_runtime_get_next_frame(vm, node, next_index);

    if (PREDICT_FALSE(!(nf->flags & VLIB_FRAME_OWNER)))
        vlib_next_frame_change_ownership(vm, node, next_index);
```

参数 node 是当前真正在运行的节点的 vlib_node_runtime_t 结构，next_index 是第几个孩子节点，即 slot。调用函数 vlib_node_runtime_get_next_frame 获取第 next_index 个孩子节点对应的 vlib_next_frame_t 结构。在解析 VLIB_FRAME_OWNER 标志位是干嘛的之前，先来看下以下例子：



假设节点 A 和 B 同时往 C 发包，按照之前 node 图初始化和上面的分析可知，A 到 C 节点有一个 vlib_next_frame_t 结构，B 到 C 节点也有 vlib_next_frame_t 结构，因此，A 和 B 同时往 C 发的包分别放在两个 ~~vlib_frame_t~~ 结构中，最终会创建两个 vlib_pending_frame_t 结构放到等待调度的 vm->node_main. pending_frames 数组中，也就说 C 会被调度两次。那么，如果 A 和 B 同时发的 C 的包加起来少于一个 vlib_frame_t 结构能够存放的包，是不是可以把两个 vlib_frame_t 结构合并成一个呢？这样就会减少 C 被调度的次数，从而提高处理性能。VLIB_FRAME_OWNER 标志位就是为了这个目的而设计的，当往孩子节点发包时会在该孩子节点对应的 vlib_next_frame_t 结构上打上 VLIB_FRAME_OWNER 标志位，表示有人正在往这个 vlib_next_frame_t 结构里发包，同时把往这个结构发包的节点的 index 和该节点往第几个孩子发包记录到孩子节点的 owner_node_index 和

owner_next_index 字段中，表示当前是哪个节点在往第几个孩子节点的 vlib_next_frame_t 结构发包。这样如果另外一个节点也在网这个节点发包，那它就可以从孩子节点的 owner_node_index 和 owner_next_index 字段获取同一个 vlib_next_frame_t 结构了，从而实现两个节点同时往一个节点发包时发到一个 vlib_next_frame_t 结构里。来看下代码是怎么实现的：

```
static void vlib_next_frame_change_ownership(vlib_main_t * vm,
        vlib_node_runtime_t * node_runtime, u32 next_index) {
    vlib_node_main_t *nm = &vm->node_main;
    vlib_next_frame_t *next_frame;
    vlib_node_t *node, *next_node;

    node = vec_elt(nm->nodes, node_runtime->node_index);

    ASSERT(node->type == VLIB_NODE_TYPE_INTERNAL
        || node->type == VLIB_NODE_TYPE_INPUT
        || node->type == VLIB_NODE_TYPE_PROCESS);

    ASSERT(vec_len (node->next_nodes) == node_runtime->n_next_nodes);

    next_frame = vlib_node_runtime_get_next_frame(vm, node_runtime, next_index);
    next_node = vec_elt(nm->nodes, node->next_nodes[next_index]);
```

获取一下 vlib_next_frame_t 结构和孩子节点对应的 vlib_node_t 结构，往下才是重点：

```
if (next_node->owner_node_index != VLIB_INVALID_NODE_INDEX) {
    vlib_next_frame_t *owner_next_frame;
    vlib_next_frame_t tmp;

    owner_next_frame = vlib_node_get_next_frame(vm,
            next_node->owner_node_index, next_node->owner_next_index);

    tmp = owner_next_frame[0];
    owner_next_frame[0] = next_frame[0];
    next_frame[0] = tmp;

    if (next_frame->flags & VLIB_FRAME_PENDING) {
        vlib_pending_frame_t *p;
        if (next_frame->frame_index != ~0) {
            vec_foreach (p, nm->pending_frames)
            {
                if (p->frame_index == next_frame->frame_index) {
                    p->next_frame_index = next_frame
                            - vm->node_main.next_frames;
                }
            }
        }
    }
} else {
    next_frame->flags |= VLIB_FRAME_OWNER;
}

next_node->owner_node_index = node->index;
next_node->owner_next_index = next_index;
```

next_node->owner_node_index 不为~0 表示有人往节点 next_node 发过包，则通过节点的 owner_node_index 和 owner_next_index 字段找到上次往该节点发包的 vlib_next_frame_t 结构。把当

前 vlib_next_frame_t 结构中的内容和找到的 vlib_next_frame_t 结构的内容交换（交换内容会使原来 vlib_next_frame_t 结构中的 frame_index 指向一个新的 vlib_frame_t 结构，不过不会有任何影响，因为新的 vlib_frame_t 结构里放的包都是往同一个孩子节点的包），这样就可以往找到的 vlib_frame_t 结构发包了。如果该 vlib_next_frame_t 结构已经创建了 vlib_pending_frame_t 结构，即已经加入等待调度的 vm->node_main. pending_frames 数组中了。由于 vlib_pending_frame_t 结构中有字段 next_frame_index，所以需要更新成当前 vlib_next_frame_t 结构的 index。

next_node->owner_node_index 为~0 则表示是第一个往该孩子节点发包的人，直接在 vlib_next_frame_t 结构上打上 VLIB_FRAME_OWNER 标志位。最后，孩子节点的 owner_node_index 和 owner_next_index 字段成当前节点的 index，和孩子节点是当前节点的第几个孩子，下个往该孩子发包的 node 用根据这两个字段来找到 vlib_next_frame_t 结构。往下看：

```
if (PREDICT_FALSE(!(nf->flags & VLIB_FRAME_IS_ALLOCATED))) {
    nf->frame_index = vlib_frame_alloc(vm, node, next_index);
    nf->flags |= VLIB_FRAME_IS_ALLOCATED;
}
```

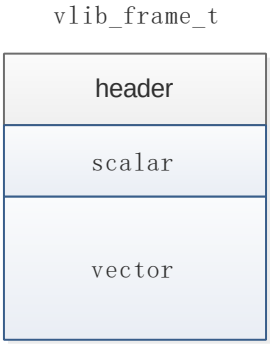
vlib_next_frame_t 结构初始化的时候 frame_index 的值为~0，也就是还没有分配 vlib_frame_t 结构，标志位 VLIB_FRAME_IS_ALLOCATED 表示是否分配过 vlib_frame_t 结构。如果没有打上该标志位则调用函数 vlib_frame_alloc 来分配 vlib_frame_t 结构。

```
static u32 vlib_frame_alloc(vlib_main_t * vm,
        vlib_node_runtime_t * from_node_runtime, u32 to_next_index) {
    vlib_node_t *from_node;

    from_node = vlib_get_node(vm, from_node_runtime->node_index);
    ASSERT(to_next_index < vec_len (from_node->next_nodes));

    return vlib_frame_alloc_to_node(vm, from_node->next_nodes[to_next_index], 0);
}
```

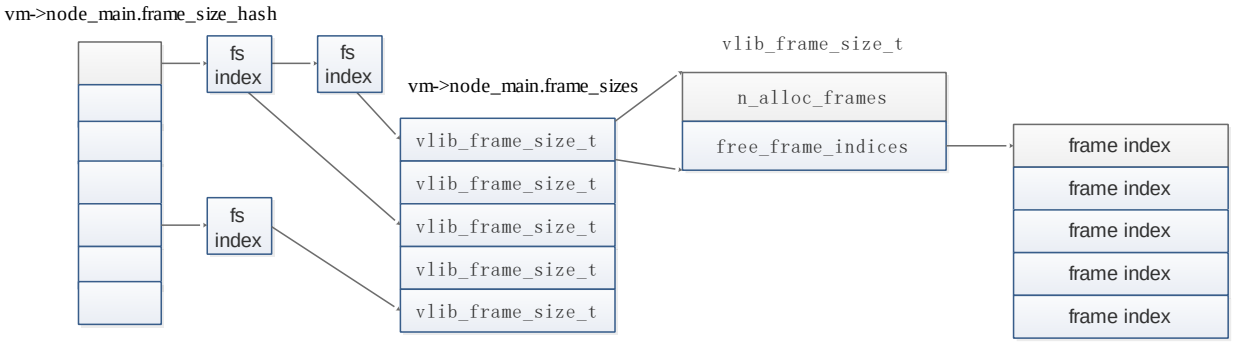
获取当前节点的 vlib_node_t 结构，从 vlib_node_t 结构的数组 next_nodes 找到第 to_next_index 个孩子节点的 index 作为参数调用函数 vlib_frame_alloc_to_node 来分配 vlib_frame_t 结构。

```
static u32 vlib_frame_alloc_to_node(vlib_main_t * vm, u32 to_node_index,
        u32 frame_flags) {
    vlib_node_main_t *nm = &vm->node_main;
    vlib_frame_size_t *fs;
    vlib_node_t *to_node;
    vlib_frame_t *f;
    u32 fi, l, n, scalar_size, vector_size;

    to_node = vlib_get_node(vm, to_node_index);

    scalar_size = to_node->scalar_size;
    vector_size = to_node->vector_size;

    fs = get_frame_size_info(nm, scalar_size, vector_size);
    n = vlib_frame_bytes(scalar_size, vector_size);
```

再来看下 vlib_frame_t 结构：



vlib_frame_t 结构分为三段：header、scalar、vector，header 是 vlib_frame_t 结构的描述字段；scalar 是 vlib_frame_t 结构的标量空间，目前不使用 scalar，可以认为 scalar 段不存在；vector 是报文索引存放的地方，可认为就是一个数组，数组里的元素就是报文的索引。vlib_frame_t 结构的分配和回收使用三个数据结构来管理，分别为： vlib_frame_size_t 结构、哈希表 vm->node_main.frame_size_hash 和数组 vm->node_main.frame_sizes。vlib_frame_size_t 结构有且仅有两个字段 n_alloc_frames 和数组 free_frame_indices。n_alloc_frames 用来记录已经分配的 vlib_frame_t 结构的个数，数组 free_frame_indices 用里存放释放的 vlib_frame_t 结构的索引。哈希表 vm->node_main.frame_size_hash 的 key 是 scalar 的字节数左移 16 位加上每个报文索引所占的字段数，value 是 vlib_frame_size_t 结构的索引。分配 vlib_frame_t 结构的过程为：先在哈希表 vm->node_main.frame_size_hash 中查找 vlib_frame_size_t 结构的索引，如果没找到则从数组 vm->node_main.frame_sizes 分配一个 vlib_frame_size_t 结构，并把它的索引加入哈希表 vm->node_main.frame_size_hash 中。再看 vlib_frame_size_t 结构里的 free_frame_indices 有没有空闲的 vlib_frame_t 结构的索引，如果有则直接从 free_frame_indices 分配，没有则调用函数 clib_mem_alloc_aligned_no_fail 为 vlib_frame_t 结构分配一块内存。如下图所示：



对应的代码为：

```
if ((l = vec_len(fs->free_frame_indices)) > 0) {
    fi = fs->free_frame_indices[l - 1];
    f = vlib_get_frame_no_check(vm, fi);
    _vec_len (fs->free_frame_indices) = l - 1;
} else {
    f = clib_mem_alloc_aligned_no_fail(n, VLIB_FRAME_ALIGN);
    fi = vlib_frame_index_no_check(vm, f);
}
```

其中，函数 get_frame_size_info 的功能是获取或分配 vlib_frame_size_t 结构。函数 vlib_frame_index_no_check 是计算 vlib_frame_t 结构对应的 index，计算的方法是 vlib_frame_t 结构的地址减去堆地址左移 6 位（一个 cache line 的大小）。由于 frame index 的类型是 u32，因此 vlib_frame_t 结构相对堆的偏移不能超过 4G，左右 6 位可以把这个限制扩大到不能超过 256G，但是要求 vlib_frame_t 结构必须按照 64 字节对齐。再来看 vlib_frame_t 结构占用内存大小的计算，函数 vlib_frame_bytes：

```
always_inline u32 vlib_frame_bytes(u32 n_scalar_bytes, u32 n_vector_bytes) {
    u32 n_bytes;

    n_bytes = vlib_frame_vector_byte_offset(n_scalar_bytes);

#define VLIB_FRAME_SIZE_EXTRA 4
    n_bytes += (VLIB_FRAME_SIZE + VLIB_FRAME_SIZE_EXTRA) * n_vector_bytes;

#define VLIB_FRAME_MAGIC (0xabadc0ed)
    n_bytes += sizeof(u32);

    n_bytes = round_pow2(n_bytes, CLIB_CACHE_LINE_BYTES);

    return n_bytes;
}
```

参数 n_scalar_bytes 是 scalar 段占用的内存大小，参数 n_vector_bytes 是每个报文占用的内存大小。总的内存=header+scalar 段长度+vector 段长度+四个字节的 magic，最后再按照 cache line 对齐。

```
if (CLIB_DEBUG > 0)
{
    memset(f, 0xfe, n);
}

{
    u32 *magic;
    magic = vlib_frame_find_magic(f, to_node);
    *magic = VLIB_FRAME_MAGIC;
}

f->flags = VLIB_FRAME_IS_ALLOCATED | frame_flags;
f->n_vectors = 0;
f->scalar_size = scalar_size;
f->vector_size = vector_size;

fs->n_alloc_frames += 1;
```

给 vlib_frame_t 结构的各字段赋值。再回到函数 vlib_get_next_frame_internal：

```
f = vlib_get_frame(vm, nf->frame_index);

if ((nf->flags & VLIB_FRAME_PENDING) && !(f->flags & VLIB_FRAME_PENDING)) {
    nf->flags &= ~VLIB_FRAME_PENDING;
    f->n_vectors = 0;
}
```

如果 nf->flags 有标志位 VLIB_FRAME_PENDING，则说明上次 nf 对应的孩子节点被调度过，也就是说往 nf->frame_index 对应的 vlib_frame_t 结构发过包，如果 f->flags 没有标志位 VLIB_FRAME_PENDING，则说明上次调度已经完成，即 vlib_frame_t 结构里的包已经被全部处理过了。因此，需要清空 f->n_vectors 的值来清空 vlib_frame_t 结构里的报文，同时清掉 nf->flags 上的标志位 VLIB_FRAME_PENDING，表示当前没有等待被调度。

```
n_used = f->n_vectors;
if (n_used >= VLIB_FRAME_SIZE || (allocate_new_next_frame && n_used > 0)) {
    if (!(nf->flags & VLIB_FRAME_NO_FREE_AFTER_DISPATCH)) {
        vlib_frame_t *f_old = vlib_get_frame(vm, nf->frame_index);
        f_old->flags |= VLIB_FRAME_FREE_AFTER_DISPATCH;
    }

    nf->frame_index = vlib_frame_alloc(vm, node, next_index);
    f = vlib_get_frame(vm, nf->frame_index);
    n_used = f->n_vectors;
}
```

如果 vlib_frame_t 结构里存放报文索引的空间已满，或者参数 allocate_new_next_frame 为 1 表示强制分配新的 vlib_frame_t 结构且当前的 vlib_frame_t 结构里有包，则需要重新分配新的 vlib_frame_t 结构。重新分配新的 vlib_frame_t 结构后，老的 vlib_frame_t 结构被调度完成后要不要释放取决于 nf->flags 是否有 VLIB_FRAME_NO_FREE_AFTER_DISPATCH 标志位，如果有则不需要释放，否

则需要释放，该标志位注册节点时可以指定。如果需要分配新的 vlib_frame_t 结构，则分配后要把 vlib_next_frame_t 结构里的 frame_index 字段更新成新的值（如果老的 vlib_frame_t 结构不释放是不是就泄露了呢？因为永远访问不到了）。

```
if (CLIB_DEBUG > 0) {
    validate_frame_magic(vm, f, vlib_get_node(vm, node->node_index),
            next_index);
}
```

判断 vlib_frame_t 结构里的 magic 值是否正确。至此，vlib_frame_t 结构已经分配完成，此时 node 的 function 函数里就可以把需要发送给孩子节点的报文放到 vlib_frame_t 结构里了，这个过程在所有 node 的 funtion 函数里都能看到，类似以下的代码段：

```
vlib_get_next_frame (vm, node, next_index, to_next, n_left_to_next);
/*other code*/
to_next[0] = bi0;
to_next++;
n_left_to_next--;
```

按照上面的分析，下一步应该是创建 vlib_pending_frame_t 结构，并把它加入数组 vm->node_main.pending_frames 等待调度了。对应的代码为函数 vlib_put_next_frame：

```
void vlib_put_next_frame(vlib_main_t * vm, vlib_node_runtime_t * r,
        u32 next_index, u32 n_vectors_left) {
    vlib_node_main_t *nm = &vm->node_main;
    vlib_next_frame_t *nf;
    vlib_frame_t *f;
    u32 n_vectors_in_frame;

    if (vm->buffer_main->callbacks_registered == 0 && CLIB_DEBUG > 0)
        vlib_put_next_frame_validate(vm, r, next_index, n_vectors_left);

    nf = vlib_node_runtime_get_next_frame(vm, r, next_index);
    f = vlib_get_frame(vm, nf->frame_index);

    if (CLIB_DEBUG > 0) {
        vlib_node_t *node = vlib_get_node(vm, r->node_index);
        validate_frame_magic(vm, f, node, next_index);
    }
```

参数 r 是 node 运行时的 vlib_node_runtime_t 结构，next_index 第几个孩子节点，n_vectors_left 是 vlib_frame_t 中剩余的报文存放个数。函数 vlib_put_next_frame_validate 的功能是调用 node 的 validate_frame 函数来校验 vlib_frame_t 和合法性。validate_frame 可以在注册 node 的时候指定，关于 validate_frame 函数这里不详细介绍了。获取 vlib_next_frame_t 结构 nf，获取 vlib_frame_t 结构 f。

```
ASSERT(n_vectors_left <= VLIB_FRAME_SIZE);
n_vectors_in_frame = VLIB_FRAME_SIZE - n_vectors_left;

f->n_vectors = n_vectors_in_frame;
```

计算并更新 vlib_frame_t 结构中当前存放的报文个数，即最大可存放数减去剩余个数。

```
if (PREDICT_TRUE(n_vectors_in_frame > 0)) {
    vlib_pending_frame_t *p;
    u32 v0, v1;

    r->cached_next_index = next_index;

    if (!(f->flags & VLIB_FRAME_PENDING)) {
        __attribute__ ((unused)) vlib_node_t *node;
        vlib_node_t *next_node;
        vlib_node_runtime_t *next_runtime;

        node = vlib_get_node(vm, r->node_index);
        next_node = vlib_get_next_node(vm, r->node_index, next_index);
        next_runtime = vlib_node_get_runtime(vm, next_node->index);

        vec_add2(nm->pending_frames, p, 1);

        p->frame_index = nf->frame_index;
        p->node_runtime_index = nf->node_runtime_index;
        p->next_frame_index = nf - nm->next_frames;
        nf->flags |= VLIB_FRAME_PENDING;
        f->flags |= VLIB_FRAME_PENDING;
```

如果 vlib_frame_t 中有报文，并且该 vlib_frame_t 还未创建 vlib_pending_frame_t 结构，则需要为该 vlib_frame_t 创建 vlib_pending_frame_t 结构 p，并给 p 赋值后加入 vm->node_main. pending_frames 等待调度。设置 nf 和 f 的 flags 字段上打上标志位 VLIB_FRAME_PENDING 表示正在等待被调度。

```
if (0 && r->thread_index != next_runtime->thread_index) {
    nf->frame_index = ~0;
    nf->flags &= ~(VLIB_FRAME_PENDING | VLIB_FRAME_IS_ALLOCATED);
}
```

这段代码的意思是如果当前节点和孩子节点的在不同的核上需要把 nf->frame_index 设置为~0，以便下次重新分配新的 vlib_frame_t 结构。不过这是不可能的，因为按照当前的实现每个核都有一份 vlib_node_runtime_t 结构，所以 if(0)，暂时当这段代码是没有意义的。

```
nf->flags |= (nf->flags & VLIB_NODE_FLAG_TRACE)
        | (r->flags & VLIB_NODE_FLAG_TRACE);

v0 = nf->vectors_since_last_overflow;
v1 = v0 + n_vectors_in_frame;
nf->vectors_since_last_overflow = v1;
if (PREDICT_FALSE(v1 < v0)) {
    vlib_node_t *node = vlib_get_node(vm, r->node_index);
    vec_elt (node->n_vectors_by_next_node, next_index)+= v0;
}
```

在 nf 和 f 的 flags 字段上设置 VLIB_NODE_FLAG_TRACE 标志位，表示可以被 trace，trace 的内容在 vlib 章节介绍。最后，在 nf 上统计处理的报文个数，如果 vectors_since_last_overflow 翻转则统计到 node->n_vectors_by_next_node 上。至此，INTERNAL 类型节点调度前的准备工作已经分析完成。

接下来分析 INTERNAL 类型的 node 调度。

```
for (i = 0; i < _vec_len(nm->pending_frames); i++)
    cpu_time_now = dispatch_pending_node(vm, i, cpu_time_now);
_vec_len (nm->pending_frames) = 0;
```

这段代码在 main 或者 worker 线程主循环里，遍历 vm->node_main. pending_frames 数组调用函数 dispatch_pending_node 等待调度的 node，调度完成所有 node 后清空 vm->node_main. pending_frames 数组。

```
static u64 dispatch_pending_node(vlib_main_t * vm, uword pending_frame_index,
        u64 last_time_stamp) {
    vlib_node_main_t *nm = &vm->node_main;
    vlib_frame_t *f;
    vlib_next_frame_t *nf, nf_dummy;
    vlib_node_runtime_t *n;
    u32 restore_frame_index;
    vlib_pending_frame_t *p;

    p = nm->pending_frames + pending_frame_index;

    n = vec_elt_at_index(nm->nodes_by_type[VLIB_NODE_TYPE_INTERNAL],
            p->node_runtime_index);
```

参数 pending_frame_index 是 vlib_pending_frame_t 结构的 index，参数 last_time_stamp 是上次调度的时间戳。根据 pending_frame_index 获取 vlib_pending_frame_t 结构 p，根据 p->node_runtime_index 获取 vlib_node_runtime_t 结构 n。

```
f = vlib_get_frame(vm, p->frame_index);
if (p->next_frame_index == VLIB_PENDING_FRAME_NO_NEXT_FRAME) {
    nf = &nf_dummy;
    nf->flags = f->flags & VLIB_NODE_FLAG_TRACE;
    nf->frame_index = ~p->frame_index;
} else
    nf = vec_elt_at_index(nm->next_frames, p->next_frame_index);

ASSERT(f->flags & VLIB_FRAME_IS_ALLOCATED);
```

p->next_frame_index 为 VLIB_PENDING_FRAME_NO_NEXT_FRAME 表示该没有 vlib_next_frame_t 结构。也就说这个 vlib_pending_frame_t 结构并不是通过上面描述的流程来创建的，上面描述流程为：查找孩子节点对应的 vlib_next_frame_t 结构、创建或获取 vlib_frame_t 结构、把报文放入 vlib_frame_t 结构、创建 vlib_pending_frame_t 结构。而是直接创建 vlib_frame_t 结构、把报文放入 vlib_frame_t 结构、创建 vlib_pending_frame_t 结构。什么情况需要这么传递报文呢？答案是当把报文发送给非孩子节点的时候会这么传递报文，因为，目标节点不是孩子节点所以没有 vlib_next_frame_t 结构，因此只能跳过查找 vlib_next_frame_t 结构这一步，对应的代码是函数 vlib_put_frame_to_node。如果没有 vlib_next_frame_t 结构，则先把给 nf 赋值一个临时的，因为为了代码流程统一下面的处理代码需要用到 vlib_next_frame_t 结构。并把 f->flags 上的标志位 VLIB_NODE_FLAG_TRACE 赋给 nf->flags（如果有的话），这里就能解释为啥 nf 和 f 都有 flags 字

段了。如果有 vlib_next_frame_t 结构，则直接根据 p->next_frame_index 获取 vlib_next_frame_t 结构 nf。此时 f->flags 应该打上标志位 VLIB_FRAME_IS_ALLOCATED 表示已经分配了 vlib_frame_t 结构。

```
restore_frame_index = ~0;
if (nf->frame_index == p->frame_index) {
    nf->frame_index = ~0;
    nf->flags &= ~VLIB_FRAME_IS_ALLOCATED;
    if (!(n->flags & VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH))
        restore_frame_index = p->frame_index;
}
```

从上面的分析流程可知，一个孩子节点对应一个 vlib_next_frame_t 结构，但是一个 vlib_next_frame_t 结构可能对应多个 vlib_frame_t 结构（当一个 vlib_frame_t 结构已满不够用时会申请一个新的 vlib_frame_t 结构，此时会更新 nf->frame_index 为新的 vlib_frame_t 结构的 index）。当 nf->frame_index 和 p->frame_index 相等时，说明 nf->frame_index 指向的时正在 pending，的 vlib_frame_t 结构。这个时候不能再往该 vlib_frame_t 结构里放报文了，因此在调用的 node 的 function 之前之前先把 nf->frame_index 置~0，并清空 VLIB_FRAME_IS_ALLOCATED 标志位，如果在 node 的 function 里需要往 nf 对应的孩子节点发包的话，就会申请一个新的 vlib_frame_t 结构，因此不会影响当前的 vlib_frame_t 结构。如果没有打上 VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH 标志位，保存 frame_index 以便调度完成后判断是否需要释放该 frame_index 对应的 vlib_frame_t 结构。

```
ASSERT(f->flags & VLIB_FRAME_PENDING);
ASSERT(f->n_vectors > 0);
```

确保 f 是 pengding 的，f 里是有报文的

```
n->flags &= ~VLIB_NODE_FLAG_TRACE;
n->flags |= (nf->flags & VLIB_FRAME_TRACE) ? VLIB_NODE_FLAG_TRACE : 0;
nf->flags &= ~VLIB_FRAME_TRACE;
```

把 nf 上的 VLIB_FRAME_TRACE 标志位设置到 n 上（如果有的话），清空 nf 上的 VLIB_FRAME_TRACE 标志位，此时 nf 对 pending 的 vlib_frame_t 已经没用了，可以被重新使用了。

```
last_time_stamp = dispatch_node(vm, n, VLIB_NODE_TYPE_INTERNAL,
        VLIB_NODE_STATE_POLLING, f, last_time_stamp);

f->flags &= ~VLIB_FRAME_PENDING;
```

调用函数 dispatch_node 去执行 n 对应的 node 的 function 函数，dispatch_node 函数在 PRE_INPUT 和 INPUT 类型的 node 调度章节已经介绍过了。最后清楚 f 的 VLIB_FRAME_PENDING 标志位表示该 f 不再试 pending 的了。

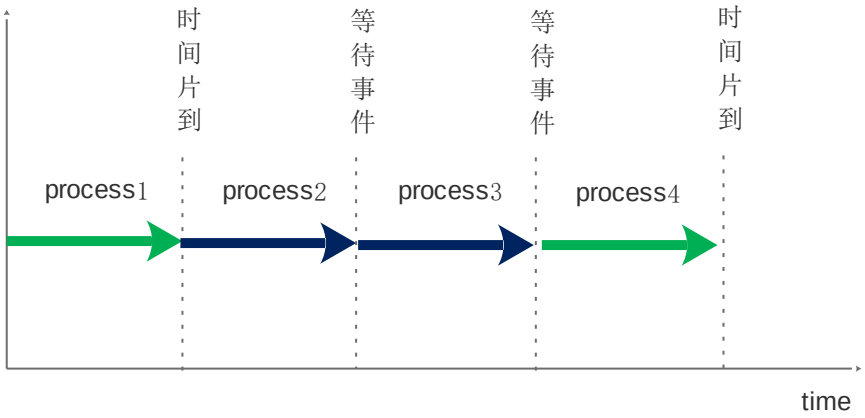
```
if (restore_frame_index != ~0) {
    ASSERT(!(f->flags & VLIB_FRAME_FREE_AFTER_DISPATCH));

    p = nm->pending_frames + pending_frame_index;

    nf = vec_elt_at_index(nm->next_frames, p->next_frame_index);
    nf->flags |= VLIB_FRAME_IS_ALLOCATED;

    if (~0 == nf->frame_index) {
        nf->frame_index = restore_frame_index;
        f->n_vectors = 0;
    } else {
        vlib_frame_free(vm, n, f);
    }
} else {
    if (f->flags & VLIB_FRAME_FREE_AFTER_DISPATCH) {
        ASSERT(!(n->flags & VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH));
        vlib_frame_free(vm, n, f);
    }
}

return last_time_stamp;
```

restore_frame_index 不为~0 的含义是：nf->frame_index 指向的 vlib_frame_t 结构就是当前处理完的 vlib_frame_t 结构，这种情况要么是 nf 和 f 是一一对应的，要么是 nf 对应多个 f，如果是后者则根据 n 有没有 VLIB_NODE_FLAG_FRAME_NO_FREE_AFTER_DISPATCH 标志位来决定是否需要释放其他的 f，但是当前的 f 不能被释放，因此，f 不可能有 VLIB_FRAME_FREE_AFTER_DISPATCH 标志位（只释放老的 f，在 vlib_get_next_frame_internal 函数里分析过）。在带掉用 dispatch_node 之前，清空了 nf 里的 VLIB_FRAME_IS_ALLOCATED 标志位和 frame_index，如果 nf->frame_index 为~0，说明在 dispatch_node 函数里该 nf 没有被用到，也就是说在 dispatch_node 函数里没有人给 nf 对应的孩子节点发包，这个时候可以把 nf->frame_index 恢复成当前的 f 的 index 了。相反则说明在 dispatch_node 函数里有人给 nf 对应的孩子节点发包了，这个时候肯定生成新的 vlib_frame_t，当前的 f 就变成老的，此时需要把它释放掉。

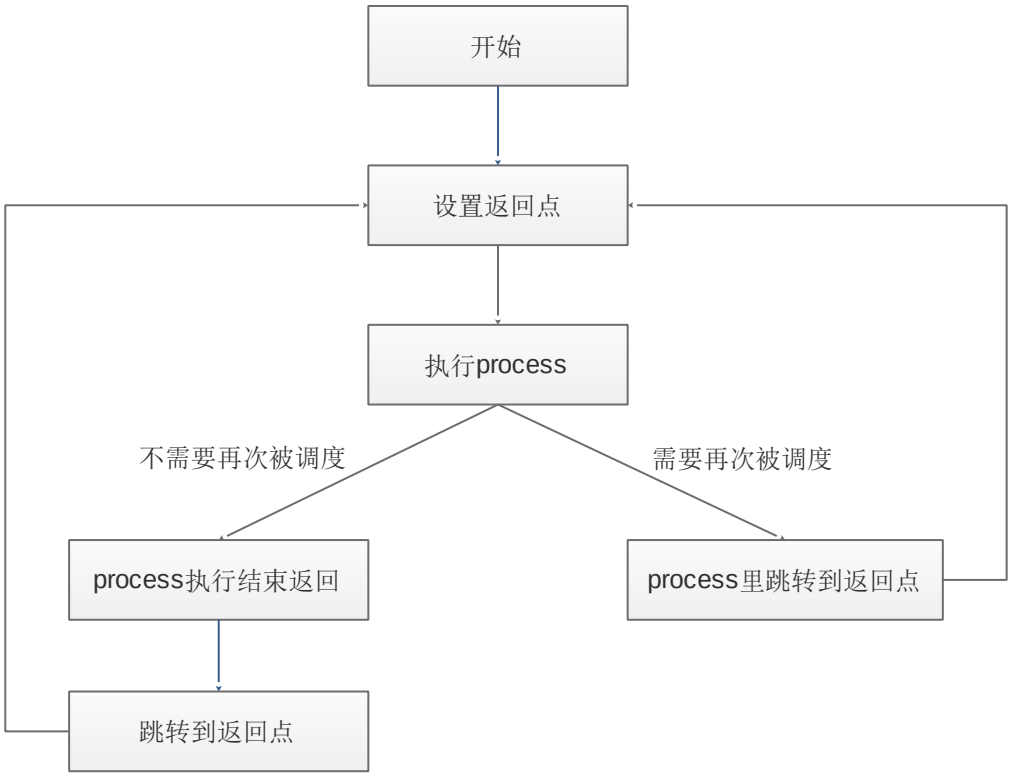
restore_frame_index 为~0 的含义和以上相反：：nf->frame_index 指向的 vlib_frame_t 结构不是当前的 f，说明当前的 f 是一个老的 vlib_frame_t 结构，此时 f 上的 VLIB_FRAME_FREE_AFTER_DISPATCH 决定是否需要释放该 f。至此，INTERNAL 类型的 node 调度分析完毕。

1.2.5.3 PROCESS 类型的 node 调度

process 的特点之前也介绍过，process 只能在 main 线程上执行，可以被挂起和恢复执行，所有的 process 共享 main 核的时钟。如下图所示：



既然所有的 process 共享 main 核的时钟，那么每个 process 执行的时间应该怎么来控制呢？答案是由 process 自己来控制，每个 process 每次被调度的时候要么把事情处理完成后让出 main 核，要么执行固定的时间片后让出 main 核。此外，和内核对进程的调度一样，当 process 从挂起到恢复执行的时候还需要从挂起的地方继续往后执行，而不是从头开始执行 process 的 function。这就需要 process 在挂起之前保存现场，恢复执行的时候恢复现场。那么，process 挂起是怎么实现的呢？也就是说当 process 时间片到或者需要让出 main 核等待事件的时候，应该怎么退出执行呢？如果是直接返回，那么下次再次被调度的时候怎么跳转到被挂起的地方呢？答案是先保存现场（保存当前堆栈信息），再通过跳转指令跳出 process 的执行函数，调到哪儿呢？在执行 process 之前需要先设置一个“返回点”，让 process 跳出自己的执行函数时跳到这个“返回点”，而设置“返回点”其实也是保存现场。还有一种情况是 process 可能只会执行一次，执行完之后不需要再次被调度了，则不需要保存现场再跳转到“返回点”了，直接从执行函数返回就行。如下图所示：



不需要再次被调度的 process 就没有挂起一说了，所以我们所说的挂起指的是在将来某时间点会被恢复执行的 process。那么，怎么恢复挂起的 process 呢？挂起的 process 要么是时间片已用完等待，时间片用完挂起的 process 要告诉 main 核恢复执行的时间点，即 process 自己挂起之前要设置好被重新调度执行的时间，时间一到 main 核要调度该 process 以恢复执行。毫无疑问，这个是通过定时器来实现的。还有一种情况是等待事件而将自己挂起，这种情况只有等待的事件发生了该 process 才会被调度执行，当所等待的事件发生时告诉 main 核即可。

了解了这些原理之后再来看看代码的实现，实现上分为两部分对应两个函数：

1、 第一次调度所有的 process 执行的代码：dispatch_process 函数

```
if (is_main)
{
    uword i;
    nm->current_process_index = ~0;
    for (i = 0; i < vec_len(nm->processes); i++)
        cpu_time_now = dispatch_process(vm, nm->processes[i], 0,
                                        cpu_time_now);
}
```

在 vlib_main_or_worker_loop 函数里，如果是 main 线程则遍历所有的 process，并调用函数 dispatch_process 执行每个 process。再看 dispatch_process 的实现：

```
static u64 dispatch_process(vlib_main_t * vm, vlib_process_t * p,
                            vlib_frame_t * f, u64 last_time_stamp)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_node_runtime_t *node_runtime = &p->node_runtime;
    vlib_node_t *node = vlib_get_node(vm, node_runtime->node_index);
    u64 t;
    uword n_vectors, is_suspend;

    if (node->state != VLIB_NODE_STATE_POLLING
            || (p->flags
                & (VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK
                   | VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT)))
        return last_time_stamp;

    p->flags |= VLIB_PROCESS_IS_RUNNING;
```

process 也是 node，和其他类型的 node 一样 state 字段表示 node 的状态，node 的状态有，这个在前面已经介绍过了。对于 process 而言只会判断是不是 POLLING 状态，如果不是则不需要调度。标志位 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK 和 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT 分别表示 process 正在等待时钟或者等待事件，这里是第一次调度所有的 process，如果 process 存在着两个标志位中的一个说明该 process 已经被调度过了，因此不再调度了。标志位 VLIB_PROCESS_IS_RUNNING 表示 process 正在执行，即正在被调度，process 在执行之前打上该标志位，执行结束后清除该标志位。

```
    t = last_time_stamp;
    vlib_elog_main_loop_event(vm, node_runtime->node_index, t,
                              f ? f->n_vectors : 0, 0);

    nm->current_process_index = node->runtime_index;

    n_vectors = vlib_process_startup(vm, p, f);

    nm->current_process_index = ~0;
```

在执行 process 之前，在 main 核对应的结构里记录当前 node 对应的 vlib_node_runtime_t 结构的 index，表示当前 main 线程正在调度哪个 process，process 执行之后清除该 index。process 的执行是调用函数 vlib_process_startup 来完成的，再看下该函数的实现：

```
static_always_inline uword vlib_process_startup(vlib_main_t * vm,
        vlib_process_t * p, vlib_frame_t * f)
{
    vlib_process_bootstrap_args_t a;
    uword r;

    a.vm = vm;
    a.process = p;
    a.frame = f;

    r = clib_setjmp(&p->return_longjmp, VLIB_PROCESS_RETURN_LONGJMP_RETURN);
    if (r == VLIB_PROCESS_RETURN_LONGJMP_RETURN)
        r = clib_calljmp(vlib_process_bootstrap, pointer_to_uword(&a),
                         (void *) p->stack + (1 << p->log2_n_stack_bytes));

    return r;
}
```

函数 clib_setjmp 的功能是保存现场，也就是上述所介绍的设置“返回点”，把保存当前寄存器的值保存到 process 的 return_longjmp 字段，后续可以根据 return_longjmp 里的值调用函数 clib_calljmp 跳转到当前的位置，返回值为第二个参数，这里是宏 VLIB_PROCESS_RETURN_LONGJMP_RETURN。如果是调用函数 clib_calljmp 跳转到当前设置的“返回点”，也会从函数 clib_setjmp 返回，此时返回值为函数 clib_calljmp 的第二个参数 VLIB_PROCESS_RETURN_LONGJMP_SUSPEND。

VLIB_PROCESS_RETURN_LONGJMP_RETURN 表示是从设置“返回点”返回的，而宏 VLIB_PROCESS_RETURN_LONGJMP_SUSPEND 则是当 process 把自己挂起时，从 process 内部返回，用来和第一种情况区分。函数 clib_calljmp 的功能是以第三个参数 stack 为栈，调用第一个参数 func 指定的函数，第二个参数 func_arg 为函数 func 的参数，即函数 vlib_process_bootstrap 的堆栈空间为 process 初始化的时候分配好的栈空间，函数 vlib_process_bootstrap 的主要功能是执行 process 的 function 函数。

```
static uword vlib_process_bootstrap(uword _a)
{
    vlib_process_bootstrap_args_t *a;
    vlib_main_t *vm;
    vlib_node_runtime_t *node;
    vlib_frame_t *f;
    vlib_process_t *p;
    uword n;

    a = uword_to_pointer(_a, vlib_process_bootstrap_args_t *);

    vm = a->vm;
    p = a->process;
    f = a->frame;
    node = &p->node_runtime;

    n = node->function(vm, node, f);

    ASSERT(vlib_process_stack_is_valid(p));

    clib_longjmp(&p->return_longjmp, n);

    return n;
}
```

上面介绍过，process 的特点是执行时间片到或者等待事件的时把自己挂起，或者干脆只执行一次，以后再也不会执行，如果是前者，在 process 里就会直接跳转到刚刚通过函数 clib_setjmp 设置的地方，不会从 node->function 处返回，也就是从函数 clib_setjmp 返回，但此时函数 clib_setjmp 的返回值不是 VLIB_PROCESS_RETURN_LONGJMP_RETURN 了，而是由 process 调用 clib_calljmp 跳转回来时传的第二个参数；如果是后者，则直接从 node->function 处返回，表示该 process 已经执行完毕，不会再被调度执行了。此时，也要跳转到前面函数 clib_setjmp 设置的返回点。既然 process 已经执行完成了，为什么不直接从函数 vlib_process_bootstrap 返回呢，而要跳转到函数 clib_setjmp 设置的返回点呢？原因是函数 vlib_process_bootstrap 的堆栈是 process 的 stack 指向的内存。如果直接从函数 vlib_process_bootstrap 返回，则无法切换成 vlib_process_startup 的堆栈。函数 clib_longjmp 的功能是跳转到前面通过函数 clib_setjmp 设置的“返回点”，第一个参数 save 为前面通过函数 clib_setjmp 设置的“返回点”的寄存器信息，第二个参数 return_value 是跳转到“返回点”时从函数 clib_setjmp 返回时的返回值，如果是因为 process 被挂起而发生跳转，则返回值由 process 指定，如果是 process 执行完成而发成跳转，则返回值为 process 的 function 的返回值。再回来接着看 dispatch_process：

```
ASSERT(n_vectors != VLIB_PROCESS_RETURN_LONGJMP_RETURN);
is_suspend = n_vectors == VLIB_PROCESS_RETURN_LONGJMP_SUSPEND;
if (is_suspend)
{
    vlib_pending_frame_t *pf;

    n_vectors = 0;
    pool_get(nm->suspended_process_frames, pf);
    pf->node_runtime_index = node->runtime_index;
    pf->frame_index = f ? vlib_frame_index(vm, f) : ~0;
    pf->next_frame_index = ~0;

    p->n_suspends += 1;
    p->suspended_process_frame_index = pf - nm->suspended_process_frames;

    if (p->flags & VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK)
    {
        TWT (tw_timer_wheel) * tw =
            (TWT (tw_timer_wheel) *) nm->timing_wheel;
        p->stop_timer_handle =
            TW (tw_timer_start)(tw,
                                vlib_timing_wheel_data_set_suspended_process(
                                    node->runtime_index) ,0 , p->resume_clock_interval);
    }
}
else
{
    p->flags &= ~VLIB_PROCESS_IS_RUNNING;
}
```

函数 vlib_process_startup 的返回要么是 process 执行完毕 function 的返回值，要么是 process 挂起的返回值 VLIB_PROCESS_RETURN_LONGJMP_SUSPEND，不可能为 VLIB_PROCESS_RETURN_LONGJMP_RETURN。如果是因为 process 被挂起而返回，则需要判断是否为因为等待时钟而挂起，如果是因为等待时钟而挂起（process 的 flags 字段应该打上 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK 标志位），需要把该 process 加入定时器，定时器到时后重新执行该 process。当 process 需要等待时钟时，挂起前除了需要打上 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK 之外，还要告诉 main 线程下次恢复执行的时钟数，是通过 process 的 resume_clock_interval 字段设置的。另外，dispatch_process 函数的第三个参数是 vlib_frame_t 结构，也就是说通过该参数把报文传递给 process 处理，不过当前的代码里没有看到哪个 process 是需要通过 vlib_frame_t 结构传递报文给它处理的。既然可以处理报文，和其他类型的 node 一样，也需要创建一个 vlib_pending_frame_t 结构 pf，由于 process 没有 vlib_next_frame_t 结构，所以 next_frame_index 字段赋值为~0。pf 是从 vm->node_main. suspended_process_frames 这个 pool 里分配的。如果 process 不是因为挂起而返回的，则说明该 process 已经执行完成，清除 VLIB_PROCESS_IS_RUNNING 标志位。elog 和统计计数相关内容这里不做介绍。到这里我们介绍了 process 第一次被调度的详细流程，接下来介绍 process 挂起后再恢复执行的详细流程。

2、 恢复挂起的 process 的代码：dispatch_suspended_process 函数

需要恢复执行的 process 有两种原因：一是等待的时钟已经到时（即定时器到期），二是等待的事件已经发生。上面我们介绍过 process 等待时钟的时候会把自己加入定时器，让定时器到

时后调度它重新执行，对应的代码在 main 线程主循环里：

```
nm->data_from_advancing_timing_wheel =
    TW (tw_timer_expire_timers_vec)((TWT (tw_timer_wheel) *) nm->timing_wheel,
        vlib_time_now(vm), nm->data_from_advancing_timing_wheel);
```

上面代码的意思是获取所有超时的 process，即数组 data_from_advancing_timing_wheel 中存放的是所有超时的 process 的 index。Process 等待的事件发生时则是通过 vlib_process_signal_event_helper 函数向该数组里 data_from_advancing_timing_wheel 加入自己的 index 来实现恢复执行的，这个函数后面再做介绍。也就是说，不管是因为等待的时钟到来还是因为等待的事件发生了，都需要把自己的 index 加入数组 data_from_advancing_timing_wheel 才能实现恢复执行。

```
if (PREDICT_FALSE(_vec_len (nm->data_from_advancing_timing_wheel) > 0))
{
    uword i;

    processes_timing_wheel_data:
    for (i = 0; i < _vec_len(nm->data_from_advancing_timing_wheel); i++)
    {
        u32 d = nm->data_from_advancing_timing_wheel[i];
        u32 di = vlib_timing_wheel_data_get_index(d);

        if (vlib_timing_wheel_data_is_timed_event(d))
        {
            vlib_signal_timed_event_data_t *te = pool_elt_at_index(
                    nm->signal_timed_event_data_pool, di);
            vlib_node_t *n = vlib_get_node(vm,
                                           te->process_node_index);
            vlib_process_t *p = vec_elt(nm->processes,
                                        n->runtime_index);
            void *data;
            data = vlib_process_signal_event_helper(nm, n, p,
                                                    te->event_type_index, te->n_data_elts,
                                                    te->n_data_elt_bytes);
            if (te->n_data_bytes < sizeof(te->inline_event_data))
                clib_memcpy(data, te->inline_event_data,
                            te->n_data_bytes);
            else
            {
                clib_memcpy(data, te->event_data_as_vector,
                            te->n_data_bytes);
                vec_free(te->event_data_as_vector);
            }
            pool_put(nm->signal_timed_event_data_pool, te);
        }
        else
        {
            cpu_time_now = clib_cpu_time_now();
            cpu_time_now = dispatch_suspended_process(vm, di,
                                                      cpu_time_now);
        }
    }
    _vec_len (nm->data_from_advancing_timing_wheel) = 0;
}
```

如果数组 data_from_advancing_timing_wheel 不为空，则说明有 process 需要恢复执行，遍历该数组中的所有 process，调用 vlib_timing_wheel_data_get_index 获取 process 的 index，其实加入数组 data_from_advancing_timing_wheel 中的不是 process 的 index，而是 process 的 index 左移一位再或上 0 或 1，1 表示该 process 等待的是时间事件（timed even），1 当前没用，所以最后一位总是 0。如果等待的是 timed even，则就算时间到了，也不是立即被恢复执行，而是调用

vlib_process_signal_event_helper 函数重新加入数组 data_from_advancing_timing_wheel，让其他 process 优先执行？由于这种情况从代码上看当前没有 process 使用，这里不做过多的介绍。如果等待的不是 timed even，则直接调用 dispatch_suspended_process 来恢复 process 的执行。

```
static u64 dispatch_suspended_process(vlib_main_t * vm, uword process_in
                                      u64 last_time_stamp)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_node_runtime_t *node_runtime;
    vlib_node_t *node;
    vlib_frame_t *f;
    vlib_process_t *p;
    vlib_pending_frame_t *pf;
    u64 t, n_vectors, is_suspend;

    t = last_time_stamp;

    p = vec_elt(nm->processes, process_index);
    if (PREDICT_FALSE(!(p->flags & VLIB_PROCESS_IS_RUNNING)))
        return last_time_stamp;

    ASSERT(p->flags & (VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK
        | VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT));

    pf = pool_elt_at_index(nm->suspended_process_frames,
                           p->suspended_process_frame_index);

    node_runtime = &p->node_runtime;
    node = vlib_get_node(vm, node_runtime->node_index);
    f = pf->frame_index != ~0 ? vlib_get_frame(vm, pf->frame_index) : 0;

    vlib_elog_main_loop_event(vm, node_runtime->node_index, t,
                              f ? f->n_vectors : 0, 0);
```

根据 process 的 index 获取 vlib_process_t 结构，如果没有 VLIB_PROCESS_IS_RUNNING 标志位表示该 process 已经执行完成或者，没有被第一次调度过，不需要恢复执行。需要恢复执行的 process 应该是在等待时钟或者等待事件，即有 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK 或 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT 标志位。获取 vlib_node_runtime_t、vlib_node_t 和 vlib_pending_frame_t 结构，并进行 elog 操作。

```
nm->current_process_index = node->runtime_index;

n_vectors = vlib_process_resume(p);
t = clib_cpu_time_now();

nm->current_process_index = ~0;
```

调用 vlib_process_resume 恢复 process 执行，在恢复执行之前记录当前正在执行的 process 的 index，执行完成后清除该记录。再来看 vlib_process_resume：

```
static_always_inline uword vlib_process_resume(vlib_process_t * p)
{
    uword r;
    p->flags &= ~(VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK
                  | VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT
                  | VLIB_PROCESS_RESUME_PENDING);
    r = clib_setjmp(&p->return_longjmp, VLIB_PROCESS_RETURN_LONGJMP_RETURN);
    if (r == VLIB_PROCESS_RETURN_LONGJMP_RETURN)
        clib_longjmp(&p->resume_longjmp, VLIB_PROCESS_RESUME_LONGJMP_RESUME);
    return r;
}
```

process 的恢复执行实际上和前面介绍的跳转是一样，即调用 clib_longjmp 跳转到 process 挂起之前通过函数 clib_setjmp 设置的“返回点”，process 挂起前保存的寄存器信息在 pprocess 的 resume_longjmp 字段中，clib_longjmp 的第二个参数为 VLIB_PROCESS_RESUME_LONGJMP_RESUME 表示恢复 process 的执行。在跳转到 process 设置的“返回点”之前也需要设置一个“返回点”，以便该 process 返回时能够跳转到当前位置，并且切换线程的栈为当前线程使用的栈。因为，在 process 里线程使用的栈是 process 的 stack 字段指向的内存。

```
is_suspend = n_vectors == VLIB_PROCESS_RETURN_LONGJMP_SUSPEND;
if (is_suspend)
{
    n_vectors = 0;
    p->n_suspends += 1;
    if (p->flags & VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK)
    {
        p->stop_timer_handle =
            TW (tw_timer_start)((TWT (tw_timer_wheel) *) nm->timing_wheel,
                                vlib_timing_wheel_data_set_suspended_process(
                                    node->runtime_index),0 ,
                                    p->resume_clock_interval);
    }
}
else
{
    p->flags &= ~VLIB_PROCESS_IS_RUNNING;
    p->suspended_process_frame_index = ~0;
    pool_put(nm->suspended_process_frames, pf);
}
```

和第一次调度一样，process 恢复执行之后返回也有两种情况：一是 process 又再次挂起（大多数是这种情况），此时需要判断挂起原因是否为等待时钟，如果是则重新加入定时器等待超时再次被调度，否则什么都不做；二是 process 执行结束，此时清除 VLIB_PROCESS_IS_RUNNING 标志位，释放 vlib_pending_frame_t 结构。至此，process 的第一次调度和恢复执行介绍完毕，还剩下两个问题：一是 process 里是怎么把自己挂起的？二是当 process 等待的事件发生时，如何把 process 的 index 加入数组 data_from_advancing_timing_wheel 以便被恢复执行，即函数 vlib_process_signal_event 实现。至于第一个问题通过上面的介绍后应该很好理解，其实 process 的挂起是通过调用函数 clib_setjmp 设置“返回点”之后，再调用函数 clib_longjmp 跳转到 process 调度前设置的“返回

点”来实现的。而挂起的原因有等待事件和等待时钟，分别对应的函数是 vlib_process_wait_for_event 和 vlib_process_wait_for_event_or_clock。

介绍事件机制之前，先来了解一下几个相关的数据结构：

每个 process 都可以定义自己的事件类型，如命令行 process 的事件类型为：

```
typedef enum
{
  UNIX_CLI_PROCESS_EVENT_READ_READY,
  UNIX_CLI_PROCESS_EVENT_QUIT,
} unix_cli_process_event_type_t;
```

每种事件类型对应一个任意数值，可以是从 0 开始也可以从任意数开始。向 process 发送事件的时候要指定事件的类型，以及需要该 process 处理的该事件对应的数据。事件数据的存储结构为数组 pending_event_data_by_type_index，下标是事件类型的值。由于事件类型可以指定定义且其值可以为任意值，因此需要一张映射表来把自定义的所有事件的值映射成从 0 开始的连续的值，这里用的映射表为哈希表 event_type_index_by_type_opaque，key 是自定义的事件的值，value 为从 event_type_pool 池中分配的 elt 的索引，这个索引作为数组 pending_event_data_by_type_index 的下标（后面称为事件类型索引），elt 存放的自定义的事件的值。同时，为了快速查找当前 process 有是否有时间或者有哪些事件需要处理，使用一个位图 non_empty_event_type_bitmap 来记录当前需要处理事件，事件类型索引对应的 bit 置 1 时表示该事件等待被处理，否则不是。

向某个 process 发送事件的过称为：

1、 先在哈希表 event_type_index_by_type_opaque 中查找该事件对应的事件类型索引，如果找不到则从 event_type_pool 池分配一个索引作为该事件的事件类型索引

2、 以事件类型索引为下标，把事件对应的数据加入数组 pending_event_data_by_type_index 中

3、 在位图 non_empty_event_type_bitmap 中设置事件类型索引对应的 bit 为 1

来看下对应的代码，对应的函数为 vlib_process_signal_event：

```
always_inline void
vlib_process_signal_event (vlib_main_t * vm,
                           uword node_index, uword type_opaque, uword data)
{
    uword *d = vlib_process_signal_event_data (vm, node_index, type_opaque,
                1 , sizeof (uword));
    d[0] = data;
}
```

参数 node_index 为 process 对应的 node 的 index，type_opaque 为自定义的事件的类型，data 为该事件对应的数据，直接调用 vlib_process_signal_event_data 函数发送事件，返回值 d 为存放事件数据的位置，直接把 data 写进去即可。

```
always_inline void *
vlib_process_signal_event_data (vlib_main_t * vm,
                                uword node_index,
                                uword type_opaque,
                                uword n_data_elts, uword n_data_elt_bytes)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_node_t *n = vlib_get_node (vm, node_index);
    vlib_process_t *p = vec_elt (nm->processes, n->runtime_index);
    uword *h, t;

    ASSERT (vlib_get_thread_index () == 0);

    h = hash_get (p->event_type_index_by_type_opaque, type_opaque);
    if (!h)
    {
        vlib_process_event_type_t *et =
            vlib_process_new_event_type (p, type_opaque);
        t = et - p->event_type_pool;
        hash_set (p->event_type_index_by_type_opaque, type_opaque, t);
    }
    else
        t = h[0];

    return vlib_process_signal_event_helper (nm, n, p, t, n_data_elts,
            n_data_elt_bytes);
}
```

这段代码的意思是根据事件类型查找事件类型索引，如果找不到则调用函数 vlib_process_new_event_type 创建一个事件类型索引，该函数比较简单，这里就不做介绍了。接着调用 vlib_process_signal_event_helper 完成发送事件的操作。

```
always_inline void *
vlib_process_signal_event_helper (vlib_node_main_t * nm,
                                  vlib_node_t * n,
                                  vlib_process_t * p,
                                  uword t,
                                  uword n_data_elts, uword n_data_elt_bytes)
{
    uword p_flags, add_to_pending, delete_from_wheel;
    void *data_to_be_written_by_caller;

    ASSERT (!pool_is_free_index (p->event_type_pool, t));

    vec_validate (p->pending_event_data_by_type_index, t);

    {
        void *data_vec = p->pending_event_data_by_type_index[t];
        uword l;

        if (!data_vec && vec_len (nm->recycled_event_data_vectors))
        {
            data_vec = vec_pop (nm->recycled_event_data_vectors);
            _vec_len (data_vec) = 0;
        }

        l = vec_len (data_vec);

        data_vec = _vec_resize (data_vec, n_data_elts,
                                (l + n_data_elts) * n_data_elt_bytes,0,  0);

        p->pending_event_data_by_type_index[t] = data_vec;
        data_to_be_written_by_caller = data_vec + l * n_data_elt_bytes;
    }
```

参数 n 为 process 对应的 node 结构，p 为 process 结构，t 为事件类型索引，n_data_elts 为事件数据个数，n_data_elt_bytes 为每个事件数据的大小。根据事件类型索引在数组 pending_event_data_by_type_index 中查找当前该事件有没有正在等待被处理的数据，如果有直接把本次该事件数据添加到数组 pending_event_data_by_type_index 之后即可，否则从回收数组 recycled_event_data_vectors 中分配一个 vector 用来存放本次的事件数据，当事件被 process 处理后，存放对应的数据的空间释放到 recycled_event_data_vectors 里，以便下次快速分配使用。可见，在 process 处理上次的事件之前可以多次向 process 发送相同的事件，且事件对应的数据可以相同也可以不相同。data_to_be_written_by_caller 指向本次事件数据存放的位置。

```
p->non_empty_event_type_bitmap =
    clib_bitmap_ori (p->non_empty_event_type_bitmap, t);
```

在位图 non_empty_event_type_bitmap 中把事件类型索引对应的 bit 位设置为 1 表示该事件正在等待被处理。

```
p_flags = p->flags;

add_to_pending = (p_flags & VLIB_PROCESS_RESUME_PENDING) == 0;

delete_from_wheel = 0;
if (p_flags & VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK)
{
    if (p_flags & VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT)
        delete_from_wheel = 1;
    else
        add_to_pending = 0;
}

add_to_pending &= nm->current_process_index != n->runtime_index;

if (add_to_pending)
{
    u32 x = vlib_timing_wheel_data_set_suspended_process (n->runtime_index);
    p->flags = p_flags | VLIB_PROCESS_RESUME_PENDING;
    vec_add1 (nm->data_from_advancing_timing_wheel, x);
    if (delete_from_wheel)
        TW (tw_timer_stop) ((TWT (tw_timer_wheel) *) nm->timing_wheel,
                            p->stop_timer_handle);
}

return data_to_be_written_by_caller;
```

标志位 VLIB_PROCESS_RESUME_PENDING 表示 process 正在等待被恢复执行，如果该 process 还未打上该标志位，说明还没有加入等待调度的数组 data_from_advancing_timing_wheel 中，需要加入该数组。如果该 process 上次挂起时因为等待时钟和事件，即只要等待的时钟到达或者等待的事件发生就要被恢复执行。而此时，等待的事件已经发生了，则需要从定时器中把它删除，不再等待时钟了，因为等待的事件已经到来；如果该 process 上次挂起仅仅是因为等待时钟，而当前则是给该 process 发送事件，和 process 挂起的原因不匹配，则不需要把它加入等待调度的数组 data_from_advancing_timing_wheel 中。最后，返回事件数据存放的位置 data_to_be_written_by_caller。

process 等待事件的实现，对应的函数为 vlib_process_wait_for_event：

```
always_inline uword *
vlib_process_wait_for_event (vlib_main_t * vm)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_process_t *p;
    uword r;

    p = vec_elt (nm->processes, nm->current_process_index);
    if (clib_bitmap_is_zero (p->non_empty_event_type_bitmap))
    {
        p->flags |= VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT;
        r = clib_setjmp (&p->resume_longjmp, VLIB_PROCESS_RESUME_LONGJMP_SUSPEND);
        if (r == VLIB_PROCESS_RESUME_LONGJMP_SUSPEND)
        {
            clib_longjmp (&p->return_longjmp,VLIB_PROCESS_RETURN_LONGJMP_SUSPEND);
        }
    }

    return p->non_empty_event_type_bitmap;
}
```

判断事件位图 non_empty_event_type_bitmap 是否为空，如果不为空则表示有事件需要处理，直接返回该事件位图 non_empty_event_type_bitmap。否则表示没有事件可以处理，此时需要把自己挂起等待事件发生，先调用 clib_setjmp 设置事件发生后恢复执行时的“返回点”，再调用 clib_longjmp 跳转到调度该 process 之前设置的“返回点”，即跳出该 process 的执行回到 main 的主循环中。设置标志位 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT 表示是因为等待事件而挂起的。

process 等待事件或时钟的实现，对应的函数为 vlib_process_wait_for_event_or_clock：

```
always_inline f64
vlib_process_wait_for_event_or_clock (vlib_main_t * vm, f64 dt)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_process_t *p;
    f64 wakeup_time;
    uword r;

    p = vec_elt (nm->processes, nm->current_process_index);

    if (vlib_process_suspend_time_is_zero (dt)
            || !clib_bitmap_is_zero (p->non_empty_event_type_bitmap))
        return dt;

    wakeup_time = vlib_time_now (vm) + dt;

    p->flags |= (VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT
                 | VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK);

    r = clib_setjmp (&p->resume_longjmp, VLIB_PROCESS_RESUME_LONGJMP_SUSPEND);
    if (r == VLIB_PROCESS_RESUME_LONGJMP_SUSPEND)
    {
        p->resume_clock_interval = dt * 1e5;
        clib_longjmp (&p->return_longjmp, VLIB_PROCESS_RETURN_LONGJMP_SUSPEND);
    }

    return wakeup_time - vlib_time_now (vm);
}
```

参数 dt 表示挂起的时间，单位为秒，函数 vlib_process_suspend_time_is_zero 判断挂起的时间是

否小于 10us，挂起的时间以 10us 为最小单位，如果太短则可能还没来及调度其他的 process 就到时间了，这里判断如果挂起时间小于 10us 则挂起时间太短不需要挂起。判断事件位图 non_empty_event_type_bitmap 是否为空，如果不为空表示有事件需要处理，直接返回。打上 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_EVENT 和 VLIB_PROCESS_IS_SUSPENDED_WAITING_FOR_CLOCK 标志位表示 process 是因为等待时钟和事件而挂起的，如果等待的时钟或者事件之一发生都需要恢复执行该 process。外次，还需要设置 resume_clock_interval 字段，该字段是挂起的时间，单位是 10us，由于 dt 的单位为 s，所以需要乘以 1e5。

process 获取事件的实现，对应的函数为 vlib_process_get_events：

```
always_inline uword
vlib_process_get_events (vlib_main_t * vm, uword ** data_vector)
{
    vlib_node_main_t *nm = &vm->node_main;
    vlib_process_t *p;
    vlib_process_event_type_t *et;
    uword r, t, l;

    p = vec_elt (nm->processes, nm->current_process_index);

    t = clib_bitmap_first_set (p->non_empty_event_type_bitmap);
    if (t == ~0)
        return t;

    p->non_empty_event_type_bitmap =
        clib_bitmap_andnoti (p->non_empty_event_type_bitmap, t);

    l = _vec_len (p->pending_event_data_by_type_index[t]);
    if (data_vector)
        vec_add (*data_vector, p->pending_event_data_by_type_index[t], l);
    _vec_len (p->pending_event_data_by_type_index[t]) = 0;

    et = pool_elt_at_index (p->event_type_pool, t);

    r = et->opaque;

    vlib_process_maybe_free_event_type (p, t);

    return r;
}
```

先判断图 non_empty_event_type_bitmap 中为否有事件需要处理，如果没有则直接返回。从事件位图中获取需要第一个需要处理的事件的索引（可见，事件的处理顺序跟事件发生的时间并没有关系，而是跟事件的索引有关），同时清空该事件在位图 non_empty_event_type_bitmap 中对应的 bit 位。根据事件类型索引，从数组 pending_event_data_by_type_index 获取事件对应的数据，并拷贝到参数 data_vector 中，从 event_type_pool 中获取事件对应的自定义类型。如果是 one time 事件（one time 事件只的是只发生一次的事件，后面不会再发生了，因此也不需要再为该事件保留索引了，由于这种事件目前机会用不到，这里先不做过多的介绍了），则还需要释放

从 event_type_pool 中分配的结构，即事件的索引。

2. 编译

3. plugin

4. vlib

4.1 error

4.2 elog

5. vppinfra

5.1 heap

6. api

通过 vpp 提供的 api 可以对 vpp 的转发表项进行添加、修改、删除和查看等操作。一般调用 api 方称为 client 端，vpp 为 server 端，client 和 server 一般为两个独立的进程。api 的实现有两种方式：共享内存和 socket 方式，对应的代码分别为 vlibmemory 和 vlibsocket。共享内存方式适合于 client 和 server 进程在同一台机器上部署的情况，而 socket 方式适合于 client 和 server 部署在不同的机器上的情况。

6.1 api 的配置

```
api-segment {
        prefix
        uid
        gid
        baseva
        global-size
        global-pvt-heap-size
        api-pvt-heap-size
        api-size
}
```

prefix：共享内存映射的前缀（文件名）

uid：对应 linux 用户权限的用户 id

gid：对应 linux 用户权限用户组 id

baseva：共享内存的基址

global-size：全局内存区的大小

global-pvt-heap-size：全局内存区中堆的大小

api-size：api 内存区的大小

api-pvt-heap-size：api 内存区中堆的大小

```
socksrv {
        socket-name
}
```

6.2 vlibmomery

为了

6.3 vlibsocket

7. feature

8. vnet

8.1 fib

8.2 arp

9. dpdk

10. cli

11. test