转载自：soul orbit: <http://r12f.com/posts/sonic-6-bgp-workflow-part-1/>

# SONiC学习笔记（六）：BGP工作流（上）—— 命令实现与FRR

以下内容已经整合进《SONiC入门指南》的 [BGP工作流](https://r12f.com/sonic-book/5-2-bgp-workflow.html) ，[BGP命令实现](https://r12f.com/sonic-book/5-2-1-bgp-command-impl.html)和[BGP路由变更下发](https://r12f.com/sonic-book/5-2-2-bgp-route-update-workflow.html) 三节中。）

[BGP](https://datatracker.ietf.org/doc/html/rfc4271" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)可能是交换机里面最常用，最重要，或者线上使用的最多的功能了。这一篇，我们就来深入的看一下BGP相关的工作流。

## **1. BGP相关进程**

SONiC使用[FRRouting](https://frrouting.org/" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)作为BGP的实现，用于负责BGP的协议处理。FRRouting是一个开源的路由软件，支持多种路由协议，包括BGP，OSPF，IS-IS，RIP，PIM，LDP等等。当FRR发布新版本后，SONiC会将其同步到[SONiC的FRR实现仓库：sonic-frr](https://github.com/sonic-net/sonic-frr" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)中，每一个版本都对应这一个分支，比如frr/8.2。

FRR主要由两个大部分组成，第一个部分是各个协议的实现，这些进程的名字都叫做\*d，而当它们收到路由更新的通知的时候，就会告诉第二个部分，也就是zebra进程，然后zebra进程会进行选路，并将最优的路由信息同步到kernel中，其主体结构如下图所示：

|  |
| --- |
| +----+ +----+ +-----+ +----+ +----+ +----+ +-----+ |bgpd| |ripd| |ospfd| |ldpd| |pbrd| |pimd| |.....| +----+ +----+ +-----+ +----+ +----+ +----+ +-----+  | | | | | | | +----v-------v--------v-------v-------v-------v--------v | | | Zebra | | | +------------------------------------------------------+  | | |  | | | +------v------+ +---------v--------+ +------v------+ | | | | | | | \*NIX Kernel | | Remote dataplane | | ........... | | | | | | | +-------------+ +------------------+ +-------------+ |

在SONiC中，这些FRR的进程都跑在bgp的容器中。另外，为了将FRR和Redis连接起来，SONiC在bgp容器中还会运行一个叫做fpgsyncd的进程（Forwarding Plane Manager syncd），它的主要功能是监听kernel的路由更新，然后将其同步到APP\_DB中。但是因为这个进程不是FRR的一部分，所以它的实现被放在了[sonic-swss](https://github.com/sonic-net/sonic-swss" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)仓库中。

## **2. BGP命令实现**

由于BGP是使用FRR来实现的，所以自然而然的，show命令会将直接请求转发给FRR的vtysh，核心代码如下：

|  |
| --- |
| *## file: src/sonic-utilities/show/bgp\_frr\_v4.py* *## 'summary' subcommand ("show ip bgp summary")* @bgp.command() @multi\_asic\_util.multi\_asic\_click\_options def summary(namespace, display):  bgp\_summary = bgp\_util.get\_bgp\_summary\_from\_all\_bgp\_instances(  constants.IPV4, namespace, display)  bgp\_util.display\_bgp\_summary(bgp\_summary=bgp\_summary, af=constants.IPV4)  *## file: src/sonic-utilities/utilities\_common/bgp\_util.py* def get\_bgp\_summary\_from\_all\_bgp\_instances(af, namespace, display):  *# IPv6 case is emitted here for simplicity*  vtysh\_cmd = "show ip bgp summary json"    for ns in device.get\_ns\_list\_based\_on\_options():  cmd\_output = run\_bgp\_show\_command(vtysh\_cmd, ns)  def run\_bgp\_command(vtysh\_cmd, bgp\_namespace=multi\_asic.DEFAULT\_NAMESPACE, vtysh\_shell\_cmd=constants.VTYSH\_COMMAND):  cmd = ['sudo', vtysh\_shell\_cmd] + bgp\_instance\_id + ['-c', vtysh\_cmd]  output, ret = clicommon.run\_command(cmd, return\_cmd=True) |

这里，我们也可以通过直接运行vtysh来进行验证：

|  |
| --- |
| root@7260cx3:/etc/sonic/frr*# which vtysh* /usr/bin/vtysh  root@7260cx3:/etc/sonic/frr*# vtysh*  Hello, this is FRRouting (version 7.5.1-sonic). Copyright 1996-2005 Kunihiro Ishiguro, et al.  7260cx3*# show ip bgp summary*  IPv4 Unicast Summary: BGP router identifier 10.1.0.32, local AS number 65100 vrf-id 0 BGP table version 6410 RIB entries 12809, using 2402 KiB of memory Peers 4, using 85 KiB of memory Peer groups 4, using 256 bytes of memory  Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd PfxSnt 10.0.0.57 4 64600 3702 3704 0 0 0 08:15:03 6401 6406 10.0.0.59 4 64600 3702 3704 0 0 0 08:15:03 6401 6406 10.0.0.61 4 64600 3705 3702 0 0 0 08:15:03 6401 6406 10.0.0.63 4 64600 3702 3702 0 0 0 08:15:03 6401 6406  Total number of neighbors 4 |

而config命令则是通过直接操作CONFIG\_DB来实现的，核心代码如下：

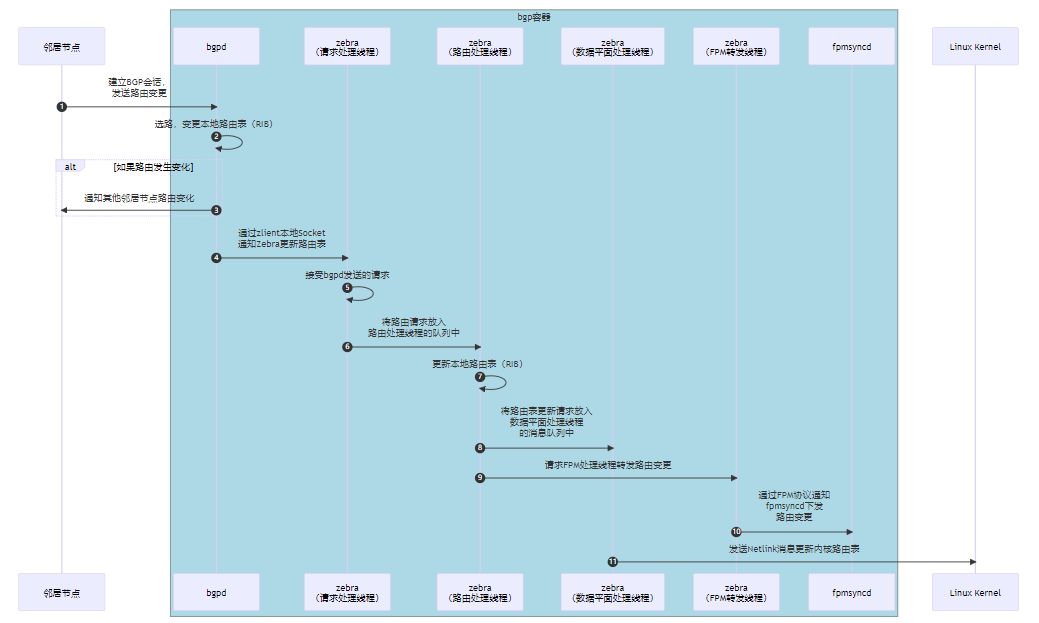
|  |
| --- |
| *## file: src/sonic-utilities/config/main.py*  @bgp.group(cls=clicommon.AbbreviationGroup) def remove():  "Remove BGP neighbor configuration from the device"  pass  @remove.command('neighbor') @click.argument('neighbor\_ip\_or\_hostname', metavar='<neighbor\_ip\_or\_hostname>', required=True) def remove\_neighbor(neighbor\_ip\_or\_hostname):  """Deletes BGP neighbor configuration of given hostname or ip from devices  User can specify either internal or external BGP neighbor to remove  """  namespaces = [DEFAULT\_NAMESPACE]  removed\_neighbor = False  ...   *# Connect to CONFIG\_DB in linux host (in case of single ASIC) or CONFIG\_DB in all the*  *# namespaces (in case of multi ASIC) and do the sepcified "action" on the BGP neighbor(s)*  for namespace in namespaces:  config\_db = ConfigDBConnector(use\_unix\_socket\_path=True, namespace=namespace)  config\_db.connect()  if \_remove\_bgp\_neighbor\_config(config\_db, neighbor\_ip\_or\_hostname):  removed\_neighbor = True  ... |

## **3. BGP路由变更下发**

路由变更几乎是SONiC中最重要的工作流，它的整个流程从bgpd进程开始，到最终通过SAI到达ASIC芯片，中间参与的进程较多，流程也较为复杂，但是弄清楚之后，我们就可以很好的理解SONiC的设计思想，并且举一反三的理解其他配置下发的工作流了。所以这一节，我们就一起来深入的分析一下它的整体流程。

为了方便我们理解和从代码层面来展示，我们把这个流程分成两个大块来介绍，分别是FRR是如何处理路由变化的，和SONiC的路由变更工作流以及它是如何与FRR进行整合的。

### **3.1. FRR处理路由变更**



关于FRR的实现，这里更多的是从代码的角度来阐述其工作流的过程，而不是其对BGP的实现细节，如果想要了解FRR的BGP实现细节，可以参考[官方文档](https://docs.frrouting.org/en/latest/bgp.html" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)。

#### **3.1.1. bgpd处理路由变更**

bgpd是FRR中专门用来处理BGP会话的进程，它会开放TCP 179端口与邻居节点建立BGP连接，并处理路由表的更新请求。当路由发生变化后，FRR也会通过它来通知其他邻居节点。

请求来到bgpd之后，它会首先来到它的io线程：bgp\_io。顾名思义，bgpd中的网络读写工作都是在这个线程上完成的：

|  |
| --- |
| *// File: src/sonic-frr/frr/bgpd/bgp\_io.c* static int bgp\_process\_reads(struct thread \*thread) {  ...   while (more) {  *// Read packets here*  ...    *// If we have more than 1 complete packet, mark it and process it later.*  if (ringbuf\_remain(ibw) >= pktsize) {  ...  added\_pkt = true;  } else break;  }  ...   if (added\_pkt)  thread\_add\_event(bm->master, bgp\_process\_packet, peer, 0, &peer->t\_process\_packet);   return 0; } |

当数据包读完后，bgpd会将其发送到主线程进行路由处理。在这里，bgpd会根据数据包的类型进行分发，其中路由更新的请求会交给bpg\_update\_receive来进行解析：

|  |
| --- |
| *// File: src/sonic-frr/frr/bgpd/bgp\_packet.c* int bgp\_process\_packet(struct thread \*thread) {  ...  unsigned int processed = 0;  while (processed < rpkt\_quanta\_old) {  uint8\_t type = 0;  bgp\_size\_t size;  ...   */\* read in the packet length and type \*/*  size = stream\_getw(peer->curr);  type = stream\_getc(peer->curr);  size -= BGP\_HEADER\_SIZE;   switch (type) {  case BGP\_MSG\_OPEN:  ...  break;  case BGP\_MSG\_UPDATE:  ...  mprc = bgp\_update\_receive(peer, size);  ...  break;  ... }  *// Process BGP UPDATE message for peer.* static int bgp\_update\_receive(struct peer \*peer, bgp\_size\_t size) {  struct stream \*s;  struct attr attr;  struct bgp\_nlri nlris[NLRI\_TYPE\_MAX];  ...   *// Parse attributes and NLRI*  memset(&attr, 0, sizeof(struct attr));  attr.label\_index = BGP\_INVALID\_LABEL\_INDEX;  attr.label = MPLS\_INVALID\_LABEL;  ...   memset(&nlris, 0, sizeof(nlris));  ...   if ((!update\_len && !withdraw\_len && nlris[NLRI\_MP\_UPDATE].length == 0)  || (attr\_parse\_ret == BGP\_ATTR\_PARSE\_EOR)) {  *// More parsing here*  ...   if (afi && peer->afc[afi][safi]) {  struct vrf \*vrf = vrf\_lookup\_by\_id(peer->bgp->vrf\_id);   */\* End-of-RIB received \*/*  if (!CHECK\_FLAG(peer->af\_sflags[afi][safi], PEER\_STATUS\_EOR\_RECEIVED)) {  ...  if (gr\_info->eor\_required == gr\_info->eor\_received) {  ...  */\* Best path selection \*/*  if (bgp\_best\_path\_select\_defer( peer->bgp, afi, safi) < 0)  return BGP\_Stop;  }  }  ...  }  }  ...   return Receive\_UPDATE\_message; } |

然后，bgpd会开始检查是否出现更优的路径，并更新自己的本地路由表（RIB，Routing Information Base）：

|  |
| --- |
| *// File: src/sonic-frr/frr/bgpd/bgp\_route.c* */\* Process the routes with the flag BGP\_NODE\_SELECT\_DEFER set \*/* int bgp\_best\_path\_select\_defer(struct bgp \*bgp, afi\_t afi, safi\_t safi) {  struct bgp\_dest \*dest;  int cnt = 0;  struct afi\_safi\_info \*thread\_info;  ...   */\* Process the route list \*/*  for (dest = bgp\_table\_top(bgp->rib[afi][safi]);  dest && bgp->gr\_info[afi][safi].gr\_deferred != 0;  dest = bgp\_route\_next(dest))  {  ...  bgp\_process\_main\_one(bgp, dest, afi, safi);  ...  }  ...   return 0; }  static void bgp\_process\_main\_one(struct bgp \*bgp, struct bgp\_dest \*dest, afi\_t afi, safi\_t safi) {  struct bgp\_path\_info \*new\_select;  struct bgp\_path\_info \*old\_select;  struct bgp\_path\_info\_pair old\_and\_new;  ...   const struct prefix \*p = bgp\_dest\_get\_prefix(dest);  ...   */\* Best path selection. \*/*  bgp\_best\_selection(bgp, dest, &bgp->maxpaths[afi][safi], &old\_and\_new, afi, safi);  old\_select = old\_and\_new.old;  new\_select = old\_and\_new.new;  ...   */\* FIB update. \*/*  if (bgp\_fibupd\_safi(safi) && (bgp->inst\_type != BGP\_INSTANCE\_TYPE\_VIEW)  && !bgp\_option\_check(BGP\_OPT\_NO\_FIB)) {   if (new\_select && new\_select->type == ZEBRA\_ROUTE\_BGP  && (new\_select->sub\_type == BGP\_ROUTE\_NORMAL  || new\_select->sub\_type == BGP\_ROUTE\_AGGREGATE  || new\_select->sub\_type == BGP\_ROUTE\_IMPORTED)) {  ...   if (old\_select && is\_route\_parent\_evpn(old\_select))  bgp\_zebra\_withdraw(p, old\_select, bgp, safi);   bgp\_zebra\_announce(dest, p, new\_select, bgp, afi, safi);  } else {  */\* Withdraw the route from the kernel. \*/*  ...  }  }   */\* EVPN route injection and clean up \*/*  ...   UNSET\_FLAG(dest->flags, BGP\_NODE\_PROCESS\_SCHEDULED);  return; } |

最后，bgp\_zebra\_announce会通过zclient通知zebra更新内核路由表。

|  |
| --- |
| *// File: src/sonic-frr/frr/bgpd/bgp\_zebra.c* void bgp\_zebra\_announce(struct bgp\_node \*rn, struct prefix \*p, struct bgp\_path\_info \*info, struct bgp \*bgp, afi\_t afi, safi\_t safi) {  ...  zclient\_route\_send(valid\_nh\_count ? ZEBRA\_ROUTE\_ADD : ZEBRA\_ROUTE\_DELETE, zclient, &api); } |

zclient使用本地socket与zebra通信，并且提供一系列的回调函数用于接收zebra的通知，核心代码如下：

|  |
| --- |
| *// File: src/sonic-frr/frr/bgpd/bgp\_zebra.c* void bgp\_zebra\_init(struct thread\_master \*master, unsigned short instance) {  zclient\_num\_connects = 0;   */\* Set default values. \*/*  zclient = zclient\_new(master, &zclient\_options\_default);  zclient\_init(zclient, ZEBRA\_ROUTE\_BGP, 0, &bgpd\_privs);  zclient->zebra\_connected = bgp\_zebra\_connected;  zclient->router\_id\_update = bgp\_router\_id\_update;  zclient->interface\_add = bgp\_interface\_add;  zclient->interface\_delete = bgp\_interface\_delete;  zclient->interface\_address\_add = bgp\_interface\_address\_add;  ... }  int zclient\_socket\_connect(struct zclient \*zclient) {  int sock;  int ret;   sock = socket(zclient\_addr.ss\_family, SOCK\_STREAM, 0);  ...   */\* Connect to zebra. \*/*  ret = connect(sock, (struct sockaddr \*)&zclient\_addr, zclient\_addr\_len);  ...   zclient->sock = sock;  return sock; } |

在bgpd容器中，我们可以在/run/frr目录下找到zebra通信使用的socket文件来进行简单的验证：

|  |
| --- |
| root@7260cx3:/run/frr*# ls -l* total 12 ... srwx------ 1 frr frr 0 Jun 16 09:16 zserv.api |

#### **3.1.2. zebra更新路由表**

由于FRR支持的路由协议很多，如果每个路由协议处理进程都单独的对内核进行操作则必然会产生冲突，很难协调合作，所以FRR使用一个单独的进程用于和所有的路由协议处理进程进行沟通，整合好信息之后统一的进行内核的路由表更新，这个进程就是zebra。

在zebra中，内核的更新发生在一个独立的数据面处理线程中：dplane\_thread。所有的请求都会通过zclient发送给zebra，经过处理之后，最后转发给dplane\_thread来处理，这样路由的处理就是有序的了，也就不会产生冲突了。

zebra启动时，会将所有的请求处理函数进行注册，当请求到来时，就可以根据请求的类型调用相应的处理函数了，核心代码如下：

|  |
| --- |
| *// File: src/sonic-frr/frr/zebra/zapi\_msg.c* void (\*zserv\_handlers[])(ZAPI\_HANDLER\_ARGS) = {  [ZEBRA\_ROUTER\_ID\_ADD] = zread\_router\_id\_add,  [ZEBRA\_ROUTER\_ID\_DELETE] = zread\_router\_id\_delete,  [ZEBRA\_INTERFACE\_ADD] = zread\_interface\_add,  [ZEBRA\_INTERFACE\_DELETE] = zread\_interface\_delete,  [ZEBRA\_ROUTE\_ADD] = zread\_route\_add,  [ZEBRA\_ROUTE\_DELETE] = zread\_route\_del,  [ZEBRA\_REDISTRIBUTE\_ADD] = zebra\_redistribute\_add,  [ZEBRA\_REDISTRIBUTE\_DELETE] = zebra\_redistribute\_delete,  ... |

我们这里拿添加路由zread\_route\_add作为例子，来继续分析后续的流程。从以下代码我们可以看到，当新的路由到来后，zebra会开始查看并更新自己内部的路由表：

|  |
| --- |
| *// File: src/sonic-frr/frr/zebra/zapi\_msg.c* static void zread\_route\_add(ZAPI\_HANDLER\_ARGS) {  struct stream \*s;  struct route\_entry \*re;  struct nexthop\_group \*ng = NULL;  struct nhg\_hash\_entry nhe;  ...   *// Decode zclient request*  s = msg;  if (zapi\_route\_decode(s, &api) < 0) {  return;  }  ...   *// Allocate new route entry.*  re = XCALLOC(MTYPE\_RE, sizeof(struct route\_entry));  re->type = api.type;  re->instance = api.instance;  ...    *// Init nexthop entry, if we have an id, then add route.*  if (!re->nhe\_id) {  zebra\_nhe\_init(&nhe, afi, ng->nexthop);  nhe.nhg.nexthop = ng->nexthop;  nhe.backup\_info = bnhg;  }  ret = rib\_add\_multipath\_nhe(afi, api.safi, &api.prefix, src\_p, re, &nhe);   *// Update stats. IPv6 is emitted here for simplicity.*  if (ret > 0) client->v4\_route\_add\_cnt++;  else if (ret < 0) client->v4\_route\_upd8\_cnt++; }  *// File: src/sonic-frr/frr/zebra/zebra\_rib.c* int rib\_add\_multipath\_nhe(afi\_t afi, safi\_t safi, struct prefix \*p,  struct prefix\_ipv6 \*src\_p, struct route\_entry \*re,  struct nhg\_hash\_entry \*re\_nhe) {  struct nhg\_hash\_entry \*nhe = NULL;  struct route\_table \*table;  struct route\_node \*rn;  int ret = 0;  ...   */\* Find table and nexthop entry \*/*  table = zebra\_vrf\_get\_table\_with\_table\_id(afi, safi, re->vrf\_id, re->table);  if (re->nhe\_id > 0) nhe = zebra\_nhg\_lookup\_id(re->nhe\_id);  else nhe = zebra\_nhg\_rib\_find\_nhe(re\_nhe, afi);   */\* Attach the re to the nhe's nexthop group. \*/*  route\_entry\_update\_nhe(re, nhe);   */\* Make it sure prefixlen is applied to the prefix. \*/*  */\* Set default distance by route type. \*/*  ...   */\* Lookup route node.\*/*  rn = srcdest\_rnode\_get(table, p, src\_p);  ...   */\* If this route is kernel/connected route, notify the dataplane to update kernel route table. \*/*  if (RIB\_SYSTEM\_ROUTE(re)) {  dplane\_sys\_route\_add(rn, re);  }   */\* Link new re to node. \*/*  SET\_FLAG(re->status, ROUTE\_ENTRY\_CHANGED);  rib\_addnode(rn, re, 1);   */\* Clean up \*/*  ...  return ret; } |

rib\_addnode会将这个路由添加请求转发给rib的处理线程，并由它顺序的进行处理：

|  |
| --- |
| static void rib\_addnode(struct route\_node \*rn, struct route\_entry \*re, int process) {  ...  rib\_link(rn, re, process); } static void rib\_link(struct route\_node \*rn, struct route\_entry \*re, int process) {  rib\_dest\_t \*dest = rib\_dest\_from\_rnode(rn);  if (!dest) dest = zebra\_rib\_create\_dest(rn);  re\_list\_add\_head(&dest->routes, re);  ...   if (process) rib\_queue\_add(rn); } |

请求会来到RIB的处理线程：rib\_process，并由它来进行进一步的选路，然后将最优的路由添加到zebra的内部路由表（RIB）中：

|  |
| --- |
| */\* Core function for processing routing information base. \*/* static void rib\_process(struct route\_node \*rn) {  struct route\_entry \*re;  struct route\_entry \*next;  struct route\_entry \*old\_selected = NULL;  struct route\_entry \*new\_selected = NULL;  struct route\_entry \*old\_fib = NULL;  struct route\_entry \*new\_fib = NULL;  struct route\_entry \*best = NULL;  rib\_dest\_t \*dest;  ...   dest = rib\_dest\_from\_rnode(rn);  old\_fib = dest->selected\_fib;  ...   */\* Check every route entry and select the best route. \*/*  RNODE\_FOREACH\_RE\_SAFE (rn, re, next) {  ...   if (CHECK\_FLAG(re->flags, ZEBRA\_FLAG\_FIB\_OVERRIDE)) {  best = rib\_choose\_best(new\_fib, re);  if (new\_fib && best != new\_fib)  UNSET\_FLAG(new\_fib->status, ROUTE\_ENTRY\_CHANGED);  new\_fib = best;  } else {  best = rib\_choose\_best(new\_selected, re);  if (new\_selected && best != new\_selected)  UNSET\_FLAG(new\_selected->status, ROUTE\_ENTRY\_CHANGED);  new\_selected = best;  }   if (best != re)  UNSET\_FLAG(re->status, ROUTE\_ENTRY\_CHANGED);  } */\* RNODE\_FOREACH\_RE \*/*  ...   */\* Update fib according to selection results \*/*  if (new\_fib && old\_fib)  rib\_process\_update\_fib(zvrf, rn, old\_fib, new\_fib);  else if (new\_fib)  rib\_process\_add\_fib(zvrf, rn, new\_fib);  else if (old\_fib)  rib\_process\_del\_fib(zvrf, rn, old\_fib);   */\* Remove all RE entries queued for removal \*/*  */\* Check if the dest can be deleted now. \*/*  ... } |

对于新的路由，会调用rib\_process\_add\_fib来将其添加到zebra的内部路由表中，然后通知dplane进行内核路由表的更新：

|  |
| --- |
| static void rib\_process\_add\_fib(struct zebra\_vrf \*zvrf, struct route\_node \*rn, struct route\_entry \*new) {  hook\_call(rib\_update, rn, "new route selected");  ...   */\* If labeled-unicast route, install transit LSP. \*/*  if (zebra\_rib\_labeled\_unicast(new))  zebra\_mpls\_lsp\_install(zvrf, rn, new);   rib\_install\_kernel(rn, new, NULL);  UNSET\_FLAG(new->status, ROUTE\_ENTRY\_CHANGED); }  void rib\_install\_kernel(struct route\_node \*rn, struct route\_entry \*re,  struct route\_entry \*old) {  struct rib\_table\_info \*info = srcdest\_rnode\_table\_info(rn);  enum zebra\_dplane\_result ret;  rib\_dest\_t \*dest = rib\_dest\_from\_rnode(rn);  ...   */\* Install the resolved nexthop object first. \*/*  zebra\_nhg\_install\_kernel(re->nhe);   */\* If this is a replace to a new RE let the originator of the RE know that they've lost \*/*  if (old && (old != re) && (old->type != re->type))  zsend\_route\_notify\_owner(rn, old, ZAPI\_ROUTE\_BETTER\_ADMIN\_WON, info->afi, info->safi);   */\* Update fib selection \*/*  dest->selected\_fib = re;   */\* Make sure we update the FPM any time we send new information to the kernel. \*/*  hook\_call(rib\_update, rn, "installing in kernel");   */\* Send add or update \*/*  if (old) ret = dplane\_route\_update(rn, re, old);  else ret = dplane\_route\_add(rn, re);  ... } |

这里有两个重要的操作，一个自然是调用dplane\_route\_\*函数来进行内核的路由表更新，另一个则是出现了两次的hook\_call，fpm的钩子函数就是挂在这个地方，用来接收并转发路由表的更新通知。这里我们一个一个来看：

##### **3.1.2.1. dplane更新内核路由表**

首先是dplane的dplane\_route\_\*函数，它们的做的事情都一样：把请求打包，然后放入dplane\_thread的消息队列中，并不会做任何实质的操作：

|  |
| --- |
| *// File: src/sonic-frr/frr/zebra/zebra\_dplane.c* enum zebra\_dplane\_result dplane\_route\_add(struct route\_node \*rn, struct route\_entry \*re) {  return dplane\_route\_update\_internal(rn, re, NULL, DPLANE\_OP\_ROUTE\_INSTALL); }  enum zebra\_dplane\_result dplane\_route\_update(struct route\_node \*rn, struct route\_entry \*re, struct route\_entry \*old\_re) {  return dplane\_route\_update\_internal(rn, re, old\_re, DPLANE\_OP\_ROUTE\_UPDATE); }  enum zebra\_dplane\_result dplane\_sys\_route\_add(struct route\_node \*rn, struct route\_entry \*re) {  return dplane\_route\_update\_internal(rn, re, NULL, DPLANE\_OP\_SYS\_ROUTE\_ADD); }  static enum zebra\_dplane\_result dplane\_route\_update\_internal(struct route\_node \*rn, struct route\_entry \*re, struct route\_entry \*old\_re, enum dplane\_op\_e op) {  enum zebra\_dplane\_result result = ZEBRA\_DPLANE\_REQUEST\_FAILURE;  int ret = EINVAL;   */\* Create and init context \*/*  struct zebra\_dplane\_ctx \*ctx = ...;   */\* Enqueue context for processing \*/*  ret = dplane\_route\_enqueue(ctx);   */\* Update counter \*/*  atomic\_fetch\_add\_explicit(&zdplane\_info.dg\_routes\_in, 1, memory\_order\_relaxed);   if (ret == AOK)  result = ZEBRA\_DPLANE\_REQUEST\_QUEUED;   return result; } |

然后，我们就来到了数据面处理线程dplane\_thread，其消息循环很简单，就是从队列中一个个取出消息，然后通过调用其处理函数：

|  |
| --- |
| *// File: src/sonic-frr/frr/zebra/zebra\_dplane.c* static int dplane\_thread\_loop(struct thread \*event) {  ...   while (prov) {  ...   */\* Process work here \*/*  (\*prov->dp\_fp)(prov);   */\* Check for zebra shutdown \*/*  */\* Dequeue completed work from the provider \*/*  ...   */\* Locate next provider \*/*  DPLANE\_LOCK();  prov = TAILQ\_NEXT(prov, dp\_prov\_link);  DPLANE\_UNLOCK();  } } |

默认情况下，dplane\_thread会使用kernel\_dplane\_process\_func来进行消息的处理，内部会根据请求的类型对内核的操作进行分发：

|  |
| --- |
| static int kernel\_dplane\_process\_func(struct zebra\_dplane\_provider \*prov) {  enum zebra\_dplane\_result res;  struct zebra\_dplane\_ctx \*ctx;  int counter, limit;  limit = dplane\_provider\_get\_work\_limit(prov);   for (counter = 0; counter < limit; counter++) {  ctx = dplane\_provider\_dequeue\_in\_ctx(prov);  if (ctx == NULL) break;   */\* A previous provider plugin may have asked to skip the kernel update. \*/*  if (dplane\_ctx\_is\_skip\_kernel(ctx)) {  res = ZEBRA\_DPLANE\_REQUEST\_SUCCESS;  goto skip\_one;  }   */\* Dispatch to appropriate kernel-facing apis \*/*  switch (dplane\_ctx\_get\_op(ctx)) {  case DPLANE\_OP\_ROUTE\_INSTALL:  case DPLANE\_OP\_ROUTE\_UPDATE:  case DPLANE\_OP\_ROUTE\_DELETE:  res = kernel\_dplane\_route\_update(ctx);  break;  ...  }  ...  }  ... }  static enum zebra\_dplane\_result kernel\_dplane\_route\_update(struct zebra\_dplane\_ctx \*ctx) {  enum zebra\_dplane\_result res;  */\* Call into the synchronous kernel-facing code here \*/*  res = kernel\_route\_update(ctx);  return res; } |

而kernel\_route\_update则是真正的内核操作了，它会通过netlink来通知内核路由更新：

|  |
| --- |
| *// File: src/sonic-frr/frr/zebra/rt\_netlink.c* *// Update or delete a prefix from the kernel, using info from a dataplane context.* enum zebra\_dplane\_result kernel\_route\_update(struct zebra\_dplane\_ctx \*ctx) {  int cmd, ret;  const struct prefix \*p = dplane\_ctx\_get\_dest(ctx);  struct nexthop \*nexthop;   if (dplane\_ctx\_get\_op(ctx) == DPLANE\_OP\_ROUTE\_DELETE) {  cmd = RTM\_DELROUTE;  } else if (dplane\_ctx\_get\_op(ctx) == DPLANE\_OP\_ROUTE\_INSTALL) {  cmd = RTM\_NEWROUTE;  } else if (dplane\_ctx\_get\_op(ctx) == DPLANE\_OP\_ROUTE\_UPDATE) {  cmd = RTM\_NEWROUTE;  }   if (!RSYSTEM\_ROUTE(dplane\_ctx\_get\_type(ctx)))  ret = netlink\_route\_multipath(cmd, ctx);  ...   return (ret == 0 ? ZEBRA\_DPLANE\_REQUEST\_SUCCESS : ZEBRA\_DPLANE\_REQUEST\_FAILURE); }  *// Routing table change via netlink interface, using a dataplane context object* static int netlink\_route\_multipath(int cmd, struct zebra\_dplane\_ctx \*ctx) {  *// Build netlink request.*  struct {  struct nlmsghdr n;  struct rtmsg r;  char buf[NL\_PKT\_BUF\_SIZE];  } req;   req.n.nlmsg\_len = NLMSG\_LENGTH(sizeof(struct rtmsg));     req.n.nlmsg\_flags = NLM\_F\_CREATE | NLM\_F\_REQUEST;     ...   */\* Talk to netlink socket. \*/*  return netlink\_talk\_info(netlink\_talk\_filter, &req.n, dplane\_ctx\_get\_ns(ctx), 0); } |

##### **3.1.2.2. FPM路由更新转发**

FPM（Forwarding Plane Manager）是FRR中用于通知其他进程路由变更的协议，其主要逻辑代码在src/sonic-frr/frr/zebra/zebra\_fpm.c中。它默认有两套协议实现：protobuf和netlink，SONiC就是使用的是netlink协议。

上面我们已经提到，它通过钩子函数实现，监听RIB中的路由变化，并通过本地Socket转发给其他的进程。这个钩子会在启动的时候就注册好，其中和我们现在看的最相关的就是rib\_update钩子了，如下所示：

|  |
| --- |
| static int zebra\_fpm\_module\_init(void) {  hook\_register(rib\_update, zfpm\_trigger\_update);  hook\_register(zebra\_rmac\_update, zfpm\_trigger\_rmac\_update);  hook\_register(frr\_late\_init, zfpm\_init);  hook\_register(frr\_early\_fini, zfpm\_fini);  return 0; }  FRR\_MODULE\_SETUP(.name = "zebra\_fpm", .version = FRR\_VERSION,  .description = "zebra FPM (Forwarding Plane Manager) module",  .init = zebra\_fpm\_module\_init, ); |

当rib\_update钩子被调用时，zfpm\_trigger\_update函数会被调用，它会将路由变更信息再次放入fpm的转发队列中，并触发写操作：

|  |
| --- |
| static int zfpm\_trigger\_update(struct route\_node \*rn, const char \*reason) {  rib\_dest\_t \*dest;  ...   *// Queue the update request*  dest = rib\_dest\_from\_rnode(rn);  SET\_FLAG(dest->flags, RIB\_DEST\_UPDATE\_FPM);  TAILQ\_INSERT\_TAIL(&zfpm\_g->dest\_q, dest, fpm\_q\_entries);  ...   zfpm\_write\_on();  return 0; }  static inline void zfpm\_write\_on(void) {  thread\_add\_write(zfpm\_g->master, zfpm\_write\_cb, 0, zfpm\_g->sock, &zfpm\_g->t\_write); } |

这个写操作的回调就会将其从队列中取出，并转换成FPM的消息格式，然后通过本地Socket转发给其他进程：

|  |
| --- |
| static int zfpm\_write\_cb(struct thread \*thread) {  struct stream \*s;   do {  int bytes\_to\_write, bytes\_written;  s = zfpm\_g->obuf;   *// Convert route info to buffer here.*  if (stream\_empty(s)) zfpm\_build\_updates();   *// Write to socket until we don' have anything to write or cannot write anymore (partial write).*  bytes\_to\_write = stream\_get\_endp(s) - stream\_get\_getp(s);  bytes\_written = write(zfpm\_g->sock, stream\_pnt(s), bytes\_to\_write);  ...  } while (1);   if (zfpm\_writes\_pending()) zfpm\_write\_on();  return 0; }  static void zfpm\_build\_updates(void) {  struct stream \*s = zfpm\_g->obuf;  do {  */\* Stop processing the queues if zfpm\_g->obuf is full or we do not have more updates to process \*/*  if (zfpm\_build\_mac\_updates() == FPM\_WRITE\_STOP) break;  if (zfpm\_build\_route\_updates() == FPM\_WRITE\_STOP) break;  } while (zfpm\_updates\_pending()); } |

到此，FRR的工作就完成了。

## **4. 小结**

我们这一篇大概总结了SONiC和BGP相关的进程，BGP命令的实现思路，和它是如何通过FRR实现BGP路由的。下次，我们会继续介绍SONiC中FRR之后的BGP变更工作流。

## **5. 参考资料**

[SONiC Architecture](https://github.com/sonic-net/SONiC/wiki/Architecture" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[Github repo: sonic-swss](https://github.com/sonic-net/sonic-swss" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[Github repo: sonic-swss-common](https://github.com/sonic-net/sonic-swss-common" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[Github repo: sonic-frr](https://github.com/sonic-net/sonic-frr" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[Github repo: sonic-utilities](https://github.com/sonic-net/sonic-utilities" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[RFC 4271: A Border Gateway Protocol 4 (BGP-4)](https://datatracker.ietf.org/doc/html/rfc4271" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)

[FRRouting](https://frrouting.org/" \t "http://r12f.com/posts/sonic-6-bgp-workflow-part-1/_blank)