转载自：soul orbit: <http://r12f.com/posts/sonic-7-bgp-workflow-part-2/>

# SONiC学习笔记（七）：BGP工作流（下）—— BGP路由变更下发

（以下内容已经整合进《SONiC入门指南》的 [BGP路由变更下发](https://r12f.com/sonic-book/5-2-2-bgp-route-update-workflow.html) 一节中。）

在上一篇中，我们介绍了BGP路由变更的工作流中从bgpd到fpmsync的部分，本篇我们将继续介绍剩下的BGP路由变更的下发流程。

## **1. SONiC路由变更工作流**

当FRR变更内核路由配置后，SONiC便会收到来自Netlink和FPM的通知，然后进行一系列操作将其下发给ASIC，其主要流程如下：

### 

### **1.1. fpmsyncd更新Redis中的路由配置**

首先，我们从源头看起。fpmsyncd在启动的时候便会开始监听FPM和Netlink的事件，用于接收路由变更消息：

|  |
| --- |
| *// File: src/sonic-swss/fpmsyncd/fpmsyncd.cpp* int main(int argc, char \*\*argv) {  ...  DBConnector db("APPL\_DB", 0);  RedisPipeline pipeline(&db);  RouteSync sync(&pipeline);  *// Register netlink message handler*  NetLink netlink;  netlink.registerGroup(RTNLGRP\_LINK);  NetDispatcher::getInstance().registerMessageHandler(RTM\_NEWROUTE, &sync);  NetDispatcher::getInstance().registerMessageHandler(RTM\_DELROUTE, &sync);  NetDispatcher::getInstance().registerMessageHandler(RTM\_NEWLINK, &sync);  NetDispatcher::getInstance().registerMessageHandler(RTM\_DELLINK, &sync);  rtnl\_route\_read\_protocol\_names(DefaultRtProtoPath);  ...   while (true) {  try {  *// Launching FPM server and wait for zebra to connect.*  FpmLink fpm(&sync);  ...  fpm.accept();  ...  } catch (FpmLink::FpmConnectionClosedException &e) {  *// If connection is closed, keep retrying until it succeeds, before handling any other events.*  cout << "Connection lost, reconnecting..." << endl;  }  ...  } } |

这样，所有的路由变更消息都会以Netlink的形式发送给RouteSync，其中[EVPN Type 5][EVPN]必须以原始消息的形式进行处理，所以会发送给onMsgRaw，其他的消息都会统一的发给处理Netlink的onMsg回调：（关于Netlink如何接收和处理消息，请移步[之前总结的通信机制一篇](http://r12f.com/posts/sonic-4-communication)）

|  |
| --- |
| *// File: src/sonic-swss/fpmsyncd/fpmlink.cpp* *// Called from: FpmLink::readData()* void FpmLink::processFpmMessage(fpm\_msg\_hdr\_t\* hdr) {  size\_t msg\_len = fpm\_msg\_len(hdr);  nlmsghdr \*nl\_hdr = (nlmsghdr \*)fpm\_msg\_data(hdr);  ...  */\* Read all netlink messages inside FPM message \*/*  for (; NLMSG\_OK (nl\_hdr, msg\_len); nl\_hdr = NLMSG\_NEXT(nl\_hdr, msg\_len))  {  */\**  *\* EVPN Type5 Add Routes need to be process in Raw mode as they contain*  *\* RMAC, VLAN and L3VNI information.*  *\* Where as all other route will be using rtnl api to extract information*  *\* from the netlink msg.*  *\*/*  bool isRaw = isRawProcessing(nl\_hdr);  nl\_msg \*msg = nlmsg\_convert(nl\_hdr);  ...  nlmsg\_set\_proto(msg, NETLINK\_ROUTE);  if (isRaw) {  */\* EVPN Type5 Add route processing \*/*  */\* This will call into onRawMsg() \*/*  processRawMsg(nl\_hdr);  } else {  */\* This will call into onMsg() \*/*  NetDispatcher::getInstance().onNetlinkMessage(msg);  }   nlmsg\_free(msg);  } }  void FpmLink::processRawMsg(struct nlmsghdr \*h) {  m\_routesync->onMsgRaw(h); }; |

接着，RouteSync收到路由变更的消息之后，会在onMsg和onMsgRaw中进行判断和分发：

|  |
| --- |
| *// File: src/sonic-swss/fpmsyncd/routesync.cpp* void RouteSync::onMsgRaw(struct nlmsghdr \*h) {  if ((h->nlmsg\_type != RTM\_NEWROUTE) && (h->nlmsg\_type != RTM\_DELROUTE))  return;  ...  onEvpnRouteMsg(h, len); } void RouteSync::onMsg(int nlmsg\_type, struct nl\_object \*obj) {  *// Refill Netlink cache here*  ...  struct rtnl\_route \*route\_obj = (struct rtnl\_route \*)obj;  auto family = rtnl\_route\_get\_family(route\_obj);  if (family == AF\_MPLS) {  onLabelRouteMsg(nlmsg\_type, obj);  return;  }  ...  unsigned int master\_index = rtnl\_route\_get\_table(route\_obj);  char master\_name[IFNAMSIZ] = {0};  if (master\_index) {  */\* If the master device name starts with VNET\_PREFIX, it is a VNET route.*  *The VNET name is exactly the name of the associated master device. \*/*  getIfName(master\_index, master\_name, IFNAMSIZ);  if (string(master\_name).find(VNET\_PREFIX) == 0) {  onVnetRouteMsg(nlmsg\_type, obj, string(master\_name));  }  */\* Otherwise, it is a regular route (include VRF route). \*/*  else {  onRouteMsg(nlmsg\_type, obj, master\_name);  }  } else {  onRouteMsg(nlmsg\_type, obj, NULL);  } } |

从上面的代码中，我们可以看到这里会有四种不同的路由处理入口，这些不同的路由会被最终通过各自的[ProducerStateTable](http://r12f.com/posts/sonic-7-bgp-workflow-part-2/4-2-2-redis-messaging-layer.html" \l "producerstatetable--consumerstatetable)写入到APPL\_DB中的不同的Table中：

| **路由类型** | **处理函数** | **Table** |
| --- | --- | --- |
| MPLS | onLabelRouteMsg | LABLE\_ROUTE\_TABLE |
| Vnet VxLan Tunnel Route | onVnetRouteMsg | VNET\_ROUTE\_TUNNEL\_TABLE |
| 其他Vnet路由 | onVnetRouteMsg | VNET\_ROUTE\_TABLE |
| EVPN Type 5 | onEvpnRouteMsg | ROUTE\_TABLE |
| 普通路由 | onRouteMsg | ROUTE\_TABLE |

这里以普通路由来举例子，其他的函数的实现虽然有所不同，但是主体的思路是一样的：

|  |
| --- |
| *// File: src/sonic-swss/fpmsyncd/routesync.cpp* void RouteSync::onRouteMsg(int nlmsg\_type, struct nl\_object \*obj, char \*vrf) {  *// Parse route info from nl\_object here.*  ...  *// Get nexthop lists*  string gw\_list;  string intf\_list;  string mpls\_list;  getNextHopList(route\_obj, gw\_list, mpls\_list, intf\_list);  ...  *// Build route info here, including protocol, interface, next hops, MPLS, weights etc.*  vector<FieldValueTuple> fvVector;  FieldValueTuple proto("protocol", proto\_str);  FieldValueTuple gw("nexthop", gw\_list);  ...  fvVector.push\_back(proto);  fvVector.push\_back(gw);  ...   *// Push to ROUTE\_TABLE via ProducerStateTable.*  m\_routeTable.set(destipprefix, fvVector);  SWSS\_LOG\_DEBUG("RouteTable set msg: %s %s %s %s", destipprefix, gw\_list.c\_str(), intf\_list.c\_str(), mpls\_list.c\_str());  ... } |

### **1.2. orchagent处理路由配置变化**

接下来，这些路由信息会来到orchagent。在orchagent启动的时候，它会创建好VNetRouteOrch和RouteOrch对象，这两个对象分别用来监听和处理Vnet相关路由和EVPN/普通路由：

|  |
| --- |
| *// File: src/sonic-swss/orchagent/orchdaemon.cpp* bool OrchDaemon::init() {  ...  vector<string> vnet\_tables = { APP\_VNET\_RT\_TABLE\_NAME, APP\_VNET\_RT\_TUNNEL\_TABLE\_NAME };  VNetRouteOrch \*vnet\_rt\_orch = new VNetRouteOrch(m\_applDb, vnet\_tables, vnet\_orch);  ...  const int routeorch\_pri = 5;  vector<table\_name\_with\_pri\_t> route\_tables = {  { APP\_ROUTE\_TABLE\_NAME, routeorch\_pri },  { APP\_LABEL\_ROUTE\_TABLE\_NAME, routeorch\_pri }  };  gRouteOrch = new RouteOrch(m\_applDb, route\_tables, gSwitchOrch, gNeighOrch, gIntfsOrch, vrf\_orch, gFgNhgOrch, gSrv6Orch);  ... } |

所有Orch对象的消息处理入口都是doTask，这里RouteOrch和VNetRouteOrch也不例外，这里我们以RouteOrch为例子，看看它是如何处理路由变化的。

note

|  |
| --- |
| 从`RouteOrch`上，我们可以真切的感受到为什么这些类被命名为`Orch`。`RouteOrch`有2500多行，其中会有和很多其他Orch的交互，以及各种各样的细节…… 代码是相对难读，请大家读的时候一定保持耐心。 |

RouteOrch在处理路由消息的时候有几点需要注意：

* 从上面init函数，我们可以看到RouteOrch不仅会管理普通路由，还会管理MPLS路由，这两种路由的处理逻辑是不一样的，所以在下面的代码中，为了简化，我们只展示普通路由的处理逻辑。
* 因为ProducerStateTable在传递和接受消息的时候都是批量传输的，所以，RouteOrch在处理消息的时候，也是批量处理的。为了支持批量处理，RouteOrch会借用EntityBulker<sai\_route\_api\_t> gRouteBulker将需要改动的SAI路由对象缓存起来，然后在doTask()函数的最后，一次性将这些路由对象的改动应用到SAI中。
* 路由的操作会需要很多其他的信息，比如每个Port的状态，每个Neighbor的状态，每个VRF的状态等等。为了获取这些信息，RouteOrch会与其他的Orch对象进行交互，比如PortOrch，NeighOrch，VRFOrch等等。

|  |
| --- |
| *// File: src/sonic-swss/orchagent/routeorch.cpp* void RouteOrch::doTask(Consumer& consumer) {  *// Calling PortOrch to make sure all ports are ready before processing route messages.*  if (!gPortsOrch->allPortsReady()) { return; }  *// Call doLabelTask() instead, if the incoming messages are from MPLS messages. Otherwise, move on as regular routes.*  ...  */\* Default handling is for ROUTE\_TABLE (regular routes) \*/*  auto it = consumer.m\_toSync.begin();  while (it != consumer.m\_toSync.end()) {  *// Add or remove routes with a route bulker*  while (it != consumer.m\_toSync.end())  {  KeyOpFieldsValuesTuple t = it->second;  *// Parse route operation from the incoming message here.*  string key = kfvKey(t);  string op = kfvOp(t);  ...  *// resync application:*  *// - When routeorch receives 'resync' message (key = "resync", op = "SET"), it marks all current routes as dirty*  *// and waits for 'resync complete' message. For all newly received routes, if they match current dirty routes,*  *// it unmarks them dirty.*  *// - After receiving 'resync complete' (key = "resync", op != "SET") message, it creates all newly added routes*  *// and removes all dirty routes.*  ...  *// Parsing VRF and IP prefix from the incoming message here.*  ...  *// Process regular route operations.*  if (op == SET\_COMMAND)  {  *// Parse and validate route attributes from the incoming message here.*  string ips;  string aliases;  ...  *// If the nexthop\_group is empty, create the next hop group key based on the IPs and aliases.*   *// Otherwise, get the key from the NhgOrch. The result will be stored in the "nhg" variable below.*  NextHopGroupKey& nhg = ctx.nhg;  ...  if (nhg\_index.empty())  {  *// Here the nexthop\_group is empty, so we create the next hop group key based on the IPs and aliases.*  ...  string nhg\_str = "";  if (blackhole) {  nhg = NextHopGroupKey();  } else if (srv6\_nh == true) {  ...  nhg = NextHopGroupKey(nhg\_str, overlay\_nh, srv6\_nh);  } else if (overlay\_nh == false) {  ...  nhg = NextHopGroupKey(nhg\_str, weights);  } else {  ...  nhg = NextHopGroupKey(nhg\_str, overlay\_nh, srv6\_nh);  }  }  else  {  *// Here we have a nexthop\_group, so we get the key from the NhgOrch.*  const NhgBase& nh\_group = getNhg(nhg\_index);  nhg = nh\_group.getNhgKey();  ...  }  ...  *// Now we start to create the SAI route entry.*  if (nhg.getSize() == 1 && nhg.hasIntfNextHop())  {  *// Skip certain routes, such as not valid, directly routes to tun0, linklocal or multicast routes, etc.*  ...  *// Create SAI route entry in addRoute function.*  if (addRoute(ctx, nhg)) it = consumer.m\_toSync.erase(it);  else it++;  }  */\**  *\* Check if the route does not exist or needs to be updated or*  *\* if the route is using a temporary next hop group owned by*  *\* NhgOrch.*  *\*/*  else if (m\_syncdRoutes.find(vrf\_id) == m\_syncdRoutes.end() ||  m\_syncdRoutes.at(vrf\_id).find(ip\_prefix) == m\_syncdRoutes.at(vrf\_id).end() ||  m\_syncdRoutes.at(vrf\_id).at(ip\_prefix) != RouteNhg(nhg, ctx.nhg\_index) ||  gRouteBulker.bulk\_entry\_pending\_removal(route\_entry) ||  ctx.using\_temp\_nhg)  {  if (addRoute(ctx, nhg)) it = consumer.m\_toSync.erase(it);  else it++;  }  ...  }  *// Handle other ops, like DEL\_COMMAND for route deletion, etc.*  ...  }  *// Flush the route bulker, so routes will be written to syncd and ASIC*  gRouteBulker.flush();  *// Go through the bulker results.*  *// Handle SAI failures, update neighbors, counters, send notifications in add/removeRoutePost functions.*  ...    */\* Remove next hop group if the reference count decreases to zero \*/*  ...  } } |

解析完路由操作后，RouteOrch会调用addRoute或者removeRoute函数来创建或者删除路由。这里以添加路由addRoute为例子来继续分析。它的逻辑主要分为几个大部分：

* 从NeighOrch中获取下一跳信息，并检查下一跳是否真的可用。
* 如果是新路由，或者是重新添加正在等待删除的路由，那么就会创建一个新的SAI路由对象
* 如果是已有的路由，那么就更新已有的SAI路由对象

|  |
| --- |
| *// File: src/sonic-swss/orchagent/routeorch.cpp* bool RouteOrch::addRoute(RouteBulkContext& ctx, const NextHopGroupKey &nextHops) {  *// Get nexthop information from NeighOrch.*  *// We also need to check PortOrch for inband port, IntfsOrch to ensure the related interface is created and etc.*  ...  *// Start to sync the SAI route entry.*  sai\_route\_entry\_t route\_entry;  route\_entry.vr\_id = vrf\_id;  route\_entry.switch\_id = gSwitchId;  copy(route\_entry.destination, ipPrefix);  sai\_attribute\_t route\_attr;  auto& object\_statuses = ctx.object\_statuses;  *// Create a new route entry in this case.*  *//*  *// In case the entry is already pending removal in the bulk, it would be removed from m\_syncdRoutes during the bulk call.*  *// Therefore, such entries need to be re-created rather than set attribute.*  if (it\_route == m\_syncdRoutes.at(vrf\_id).end() || gRouteBulker.bulk\_entry\_pending\_removal(route\_entry)) {  if (blackhole) {  route\_attr.id = SAI\_ROUTE\_ENTRY\_ATTR\_PACKET\_ACTION;  route\_attr.value.s32 = SAI\_PACKET\_ACTION\_DROP;  } else {  route\_attr.id = SAI\_ROUTE\_ENTRY\_ATTR\_NEXT\_HOP\_ID;  route\_attr.value.oid = next\_hop\_id;  }  */\* Default SAI\_ROUTE\_ATTR\_PACKET\_ACTION is SAI\_PACKET\_ACTION\_FORWARD \*/*  object\_statuses.emplace\_back();  sai\_status\_t status = gRouteBulker.create\_entry(&object\_statuses.back(), &route\_entry, 1, &route\_attr);  if (status == SAI\_STATUS\_ITEM\_ALREADY\_EXISTS) {  return false;  }  }  *// Update existing route entry in this case.*  else {  *// Set the packet action to forward when there was no next hop (dropped) and not pointing to blackhole.*  if (it\_route->second.nhg\_key.getSize() == 0 && !blackhole) {  route\_attr.id = SAI\_ROUTE\_ENTRY\_ATTR\_PACKET\_ACTION;  route\_attr.value.s32 = SAI\_PACKET\_ACTION\_FORWARD;   object\_statuses.emplace\_back();  gRouteBulker.set\_entry\_attribute(&object\_statuses.back(), &route\_entry, &route\_attr);  }  *// Only 1 case is listed here as an example. Other cases are handled with similar logic by calling set\_entry\_attributes as well.*  ...  }  ... } |

在创建和设置好所有的路由后，RouteOrch会调用gRouteBulker.flush()来将所有的路由写入到ASIC\_DB中。flush()函数很简单，就是将所有的请求分批次进行处理，默认情况下每一批是1000个，这个定义在OrchDaemon中，并通过构造函数传入：

|  |
| --- |
| *// File: src/sonic-swss/orchagent/orchdaemon.cpp* #define DEFAULT\_MAX\_BULK\_SIZE 1000 size\_t gMaxBulkSize = DEFAULT\_MAX\_BULK\_SIZE; *// File: src/sonic-swss/orchagent/bulker.h* template <typename T> class EntityBulker { public:  using Ts = SaiBulkerTraits<T>;  using Te = typename Ts::entry\_t;  ...  void flush()  {  *// Bulk remove entries*  if (!removing\_entries.empty()) {  *// Split into batches of max\_bulk\_size, then call flush. Similar to creating\_entries, so details are omitted.*  std::vector<Te> rs;  ...  flush\_removing\_entries(rs);  removing\_entries.clear();  }  *// Bulk create entries*  if (!creating\_entries.empty()) {  *// Split into batches of max\_bulk\_size, then call flush\_creating\_entries to call SAI batch create API to create*  *// the objects in batch.*  std::vector<Te> rs;  std::vector<sai\_attribute\_t const\*> tss;  std::vector<uint32\_t> cs;    for (auto const& i: creating\_entries) {  sai\_object\_id\_t \*pid = std::get<0>(i);  auto const& attrs = std::get<1>(i);  if (\*pid == SAI\_NULL\_OBJECT\_ID) {  rs.push\_back(pid);  tss.push\_back(attrs.data());  cs.push\_back((uint32\_t)attrs.size());  *// Batch create here.*  if (rs.size() >= max\_bulk\_size) {  flush\_creating\_entries(rs, tss, cs);  }  }  }  flush\_creating\_entries(rs, tss, cs);  creating\_entries.clear();  }  *// Bulk update existing entries*  if (!setting\_entries.empty()) {  *// Split into batches of max\_bulk\_size, then call flush. Similar to creating\_entries, so details are omitted.*  std::vector<Te> rs;  std::vector<sai\_attribute\_t> ts;  std::vector<sai\_status\_t\*> status\_vector;  ...  flush\_setting\_entries(rs, ts, status\_vector);  setting\_entries.clear();  }  }  sai\_status\_t flush\_creating\_entries(  \_Inout\_ std::vector<Te> &rs,  \_Inout\_ std::vector<sai\_attribute\_t const\*> &tss,  \_Inout\_ std::vector<uint32\_t> &cs)  {  ...  *// Call SAI bulk create API*  size\_t count = rs.size();  std::vector<sai\_status\_t> statuses(count);  sai\_status\_t status = (\*create\_entries)((uint32\_t)count, rs.data(), cs.data(), tss.data()  , SAI\_BULK\_OP\_ERROR\_MODE\_IGNORE\_ERROR, statuses.data());  *// Set results back to input entries and clean up the batch below.*  for (size\_t ir = 0; ir < count; ir++) {  auto& entry = rs[ir];  sai\_status\_t \*object\_status = creating\_entries[entry].second;  if (object\_status) {  \*object\_status = statuses[ir];  }  }  rs.clear(); tss.clear(); cs.clear();  return status;  }  *// flush\_removing\_entries and flush\_setting\_entries are similar to flush\_creating\_entries, so we omit them here.*  ... }; |

### **1.3. orchagent中的SAI对象转发**

细心的小伙伴肯定已经发现了奇怪的地方，这里EntityBulker怎么看着像在直接调用SAI API呢？难道它们不应该是在syncd中调用的吗？如果我们对传入EntityBulker的SAI API对象进行跟踪，我们甚至会找到sai\_route\_api\_t就是SAI的接口，而orchagent中还有SAI的初始化代码，如下：

|  |
| --- |
| *// File: src/sonic-sairedis/debian/libsaivs-dev/usr/include/sai/sairoute.h* */\*\**  *\* @brief Router entry methods table retrieved with sai\_api\_query()*  *\*/* typedef struct \_sai\_route\_api\_t {  sai\_create\_route\_entry\_fn create\_route\_entry;  sai\_remove\_route\_entry\_fn remove\_route\_entry;  sai\_set\_route\_entry\_attribute\_fn set\_route\_entry\_attribute;  sai\_get\_route\_entry\_attribute\_fn get\_route\_entry\_attribute;  sai\_bulk\_create\_route\_entry\_fn create\_route\_entries;  sai\_bulk\_remove\_route\_entry\_fn remove\_route\_entries;  sai\_bulk\_set\_route\_entry\_attribute\_fn set\_route\_entries\_attribute;  sai\_bulk\_get\_route\_entry\_attribute\_fn get\_route\_entries\_attribute; } sai\_route\_api\_t; *// File: src/sonic-swss/orchagent/saihelper.cpp* void initSaiApi() {  SWSS\_LOG\_ENTER();  if (ifstream(CONTEXT\_CFG\_FILE))  {  SWSS\_LOG\_NOTICE("Context config file %s exists", CONTEXT\_CFG\_FILE);  gProfileMap[SAI\_REDIS\_KEY\_CONTEXT\_CONFIG] = CONTEXT\_CFG\_FILE;  }  sai\_api\_initialize(0, (const sai\_service\_method\_table\_t \*)&test\_services);  sai\_api\_query(SAI\_API\_SWITCH, (void \*\*)&sai\_switch\_api);  ...  sai\_api\_query(SAI\_API\_NEIGHBOR, (void \*\*)&sai\_neighbor\_api);  sai\_api\_query(SAI\_API\_NEXT\_HOP, (void \*\*)&sai\_next\_hop\_api);  sai\_api\_query(SAI\_API\_NEXT\_HOP\_GROUP, (void \*\*)&sai\_next\_hop\_group\_api);  sai\_api\_query(SAI\_API\_ROUTE, (void \*\*)&sai\_route\_api);  ...  sai\_log\_set(SAI\_API\_SWITCH, SAI\_LOG\_LEVEL\_NOTICE);  ...  sai\_log\_set(SAI\_API\_NEIGHBOR, SAI\_LOG\_LEVEL\_NOTICE);  sai\_log\_set(SAI\_API\_NEXT\_HOP, SAI\_LOG\_LEVEL\_NOTICE);  sai\_log\_set(SAI\_API\_NEXT\_HOP\_GROUP, SAI\_LOG\_LEVEL\_NOTICE);  sai\_log\_set(SAI\_API\_ROUTE, SAI\_LOG\_LEVEL\_NOTICE);  ... } |

相信大家第一次看到这个代码会感觉到非常的困惑。不过别着急，这其实就是orchagent中SAI对象的转发机制。

熟悉RPC的小伙伴一定不会对proxy-stub模式感到陌生 —— 利用统一的接口来定义通信双方调用接口，在调用方实现序列化和发送，然后再接收方实现接收，反序列化与分发。这里SONiC的做法也是类似的：利用SAI API本身作为统一的接口，并实现好序列化和发送功能给orchagent来调用，然后再syncd中实现接收，反序列化与分发功能。

这里，发送端叫做ClientSai，实现在src/sonic-sairedis/lib/ClientSai.\*中。而序列化与反序列化实现在SAI metadata中：src/sonic-sairedis/meta/sai\_serialize.h：

|  |
| --- |
| *// File: src/sonic-sairedis/lib/ClientSai.h* namespace sairedis {  class ClientSai:  public sairedis::SaiInterface  {  ...  }; }  *// File: src/sonic-sairedis/meta/sai\_serialize.h* *// Serialize* std::string sai\_serialize\_route\_entry(\_In\_ const sai\_route\_entry\_t &route\_entry); ... *// Deserialize* void sai\_deserialize\_route\_entry(\_In\_ const std::string& s, \_In\_ sai\_route\_entry\_t &route\_entry); ... |

orchagent在编译的时候，会去链接libsairedis，从而实现调用SAI API时，对SAI对象进行序列化和发送：

|  |
| --- |
| *# File: src/sonic-swss/orchagent/Makefile.am* orchagent\_LDADD = $(LDFLAGS\_ASAN) -lnl-3 -lnl-route-3 -lpthread -lsairedis -lsaimeta -lsaimetadata -lswsscommon -lzmq |

我们这里用Bulk Create作为例子，来看看ClientSai是如何实现序列化和发送的：

|  |
| --- |
| *// File: src/sonic-sairedis/lib/ClientSai.cpp* sai\_status\_t ClientSai::bulkCreate(  \_In\_ sai\_object\_type\_t object\_type,  \_In\_ sai\_object\_id\_t switch\_id,  \_In\_ uint32\_t object\_count,  \_In\_ const uint32\_t \*attr\_count,  \_In\_ const sai\_attribute\_t \*\*attr\_list,  \_In\_ sai\_bulk\_op\_error\_mode\_t mode,  \_Out\_ sai\_object\_id\_t \*object\_id,  \_Out\_ sai\_status\_t \*object\_statuses) {  MUTEX();  REDIS\_CHECK\_API\_INITIALIZED();  std::vector<std::string> serialized\_object\_ids;  *// Server is responsible for generate new OID but for that we need switch ID*  *// to be sent to server as well, so instead of sending empty oids we will*  *// send switch IDs*  for (uint32\_t idx = 0; idx < object\_count; idx++) {  serialized\_object\_ids.emplace\_back(sai\_serialize\_object\_id(switch\_id));  }  auto status = bulkCreate(object\_type, serialized\_object\_ids, attr\_count, attr\_list, mode, object\_statuses);  *// Since user requested create, OID value was created remotely and it was returned in m\_lastCreateOids*  for (uint32\_t idx = 0; idx < object\_count; idx++) {  if (object\_statuses[idx] == SAI\_STATUS\_SUCCESS) {  object\_id[idx] = m\_lastCreateOids.at(idx);  } else {  object\_id[idx] = SAI\_NULL\_OBJECT\_ID;  }  }  return status; } sai\_status\_t ClientSai::bulkCreate(  \_In\_ sai\_object\_type\_t object\_type,  \_In\_ const std::vector<std::string> &serialized\_object\_ids,  \_In\_ const uint32\_t \*attr\_count,  \_In\_ const sai\_attribute\_t \*\*attr\_list,  \_In\_ sai\_bulk\_op\_error\_mode\_t mode,  \_Inout\_ sai\_status\_t \*object\_statuses) {  ...  *// Calling SAI serialize APIs to serialize all objects*  std::string str\_object\_type = sai\_serialize\_object\_type(object\_type);  std::vector<swss::FieldValueTuple> entries;  for (size\_t idx = 0; idx < serialized\_object\_ids.size(); ++idx) {  auto entry = SaiAttributeList::serialize\_attr\_list(object\_type, attr\_count[idx], attr\_list[idx], false);  if (entry.empty()) {  swss::FieldValueTuple null("NULL", "NULL");  entry.push\_back(null);  }   std::string str\_attr = Globals::joinFieldValues(entry);  swss::FieldValueTuple fvtNoStatus(serialized\_object\_ids[idx] , str\_attr);  entries.push\_back(fvtNoStatus);  }  std::string key = str\_object\_type + ":" + std::to\_string(entries.size());  *// Send to syncd via the communication channel.*  m\_communicationChannel->set(key, entries, REDIS\_ASIC\_STATE\_COMMAND\_BULK\_CREATE);  *// Wait for response from syncd.*  return waitForBulkResponse(SAI\_COMMON\_API\_BULK\_CREATE, (uint32\_t)serialized\_object\_ids.size(), object\_statuses); } |

最终，ClientSai会调用m\_communicationChannel->set()，将序列化后的SAI对象发送给syncd。而这个Channel，在202106版本之前，就是[基于Redis的ProducerTable](https://github.com/sonic-net/sonic-sairedis/blob/202106/lib/inc/RedisChannel.h" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)了。可能是基于效率的考虑，从202111版本开始，这个Channel已经更改为[ZMQ](https://github.com/sonic-net/sonic-sairedis/blob/202111/lib/ZeroMQChannel.h" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)了。

|  |
| --- |
| *// File: <https://github.com/sonic-net/sonic-sairedis/blob/202106/lib/inc/RedisChannel.h>* class RedisChannel: public Channel {  ...  */\*\**  *\* @brief Asic state channel.*  *\**  *\* Used to sent commands like create/remove/set/get to syncd.*  *\*/*  std::shared\_ptr<swss::ProducerTable> m\_asicState;   ... }; *// File: src/sonic-sairedis/lib/ClientSai.cpp* sai\_status\_t ClientSai::initialize(  \_In\_ uint64\_t flags,  \_In\_ const sai\_service\_method\_table\_t \*service\_method\_table) {  ...  m\_communicationChannel = std::make\_shared<ZeroMQChannel>(  cc->m\_zmqEndpoint,  cc->m\_zmqNtfEndpoint,  std::bind(&ClientSai::handleNotification, this, \_1, \_2, \_3));  m\_apiInitialized = true;  return SAI\_STATUS\_SUCCESS; } |

关于进程通信的方法，这里就不再赘述了，大家可以参考第四章描述的[进程间的通信机制](http://r12f.com/posts/sonic-4-communication)。

### **1.4. syncd更新ASIC**

最后，当SAI对象生成好并发送给syncd后，syncd会接收，处理，更新ASIC\_DB，最后更新ASIC。这一段的工作流，我们已经在[Syncd-SAI工作流](http://r12f.com/posts/sonic-5-syncd-sai-workflow)中详细介绍过了，这里就不再赘述了，大家可以移步去查看。

## **2. 参考资料**

[SONiC Architecture](https://github.com/sonic-net/SONiC/wiki/Architecture" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)

[Github repo: sonic-swss](https://github.com/sonic-net/sonic-swss" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)

[Github repo: sonic-swss-common](https://github.com/sonic-net/sonic-swss-common" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)

[Github repo: sonic-sairedis](https://github.com/sonic-net/sonic-sairedis/" \t "http://r12f.com/posts/sonic-7-bgp-workflow-part-2/_blank)