转载自：soul orbit: <http://r12f.com/posts/sonic-5-syncd-sai-workflow/>

# SONiC学习笔记（五）：Syncd-SAI工作流

（以下内容已经整合进《SONiC入门指南》的 [SAI介绍](https://r12f.com/sonic-book/2-4-sai-intro.html) 和 [Syncd-SAI工作流](https://r12f.com/sonic-book/5-1-syncd-sai-workflow.html) 两节中。）

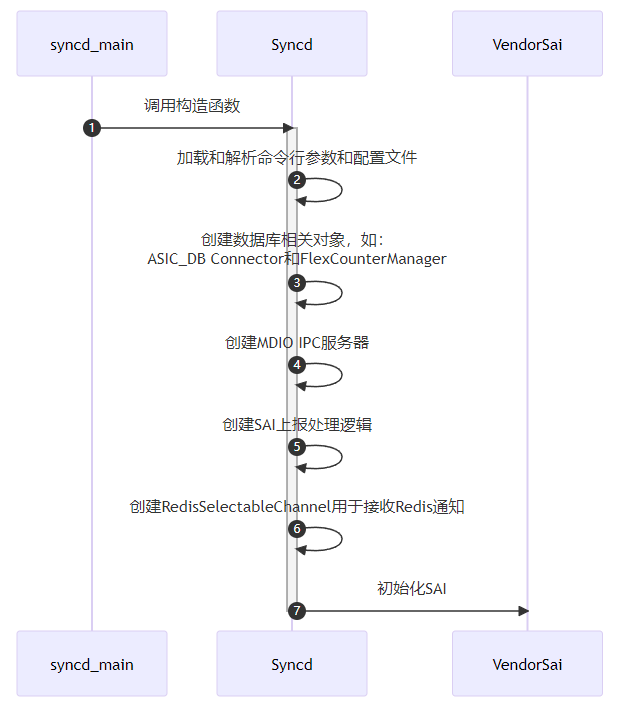
[Syncd容器](http://r12f.com/posts/sonic-2-key-components/" \l "ASIC%E7%AE%A1%E7%90%86%E5%AE%B9%E5%99%A8%EF%BC%9Asyncd)是SONiC中专门负责管理ASIC的容器，其中核心进程syncd负责与Redis数据库沟通，加载SAI并与其交互，以完成ASIC的初始化，配置和状态上报的处理等等。

由于SONiC中大量的工作流最后都需要通过Syncd和SAI来和ASIC进行交互，所以这一部分也就成为了这些工作流的公共部分，所以，在展开其他工作流之前，我们先来看一下Syncd和SAI是如何工作的。

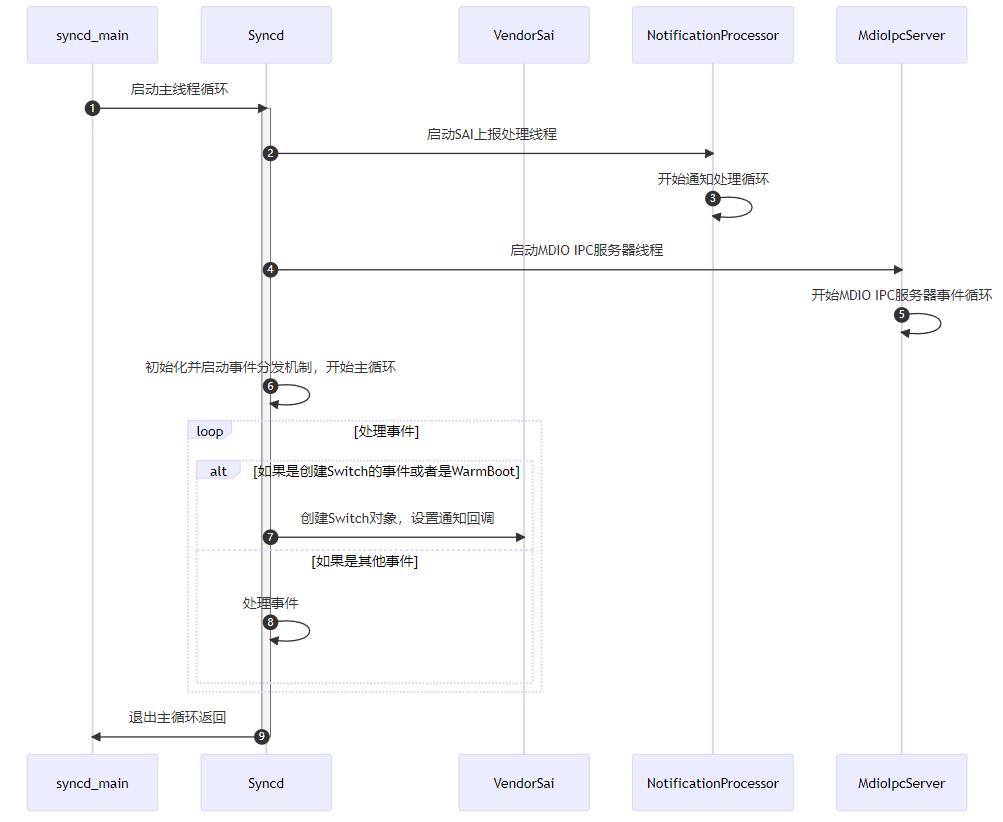
## **1. Syncd启动流程**

syncd进程的入口在syncd\_main.cpp中的syncd\_main函数，其启动的整体流程大致分为两部分。

第一部分是创建各个对象，并进行初始化：



第二个部分是启动主循环，并且处理初始化事件：



然后我们再从代码的角度来更加仔细的看一下这个流程。

### **1.1. syncd\_main函数**

syncd\_main函数本身非常简单，主要逻辑就是创建Syncd对象，然后调用其run方法：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/syncd\_main.cpp* int syncd\_main(int argc, char \*\*argv) {  auto vendorSai = std::make\_shared<VendorSai>();  auto syncd = std::make\_shared<Syncd>(vendorSai, commandLineOptions, isWarmStart);  syncd->run();  return EXIT\_SUCCESS; } |

其中，Syncd对象的构造函数负责初始化Syncd中的各个功能，而run方法则负责启动Syncd的主循环。

### **1.2. Syncd构造函数**

Syncd对象的构造函数负责创建或初始化Syncd中的各个功能，比如用于连接数据库的对象，统计管理，和ASIC通知的处理逻辑等等，其主要代码如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/Syncd.cpp* Syncd::Syncd(  \_In\_ std::shared\_ptr<sairedis::SaiInterface> vendorSai,  \_In\_ std::shared\_ptr<CommandLineOptions> cmd,  \_In\_ bool isWarmStart):  m\_vendorSai(vendorSai),  ... {  ...  *// Load context config*  auto ccc = sairedis::ContextConfigContainer::loadFromFile(m\_commandLineOptions->m\_contextConfig.c\_str());  m\_contextConfig = ccc->get(m\_commandLineOptions->m\_globalContext);  ...  *// Create FlexCounter manager*  m\_manager = std::make\_shared<FlexCounterManager>(m\_vendorSai, m\_contextConfig->m\_dbCounters);  *// Create DB related objects*  m\_dbAsic = std::make\_shared<swss::DBConnector>(m\_contextConfig->m\_dbAsic, 0);  m\_mdioIpcServer = std::make\_shared<MdioIpcServer>(m\_vendorSai, m\_commandLineOptions->m\_globalContext);  m\_selectableChannel = std::make\_shared<sairedis::RedisSelectableChannel>(m\_dbAsic, ASIC\_STATE\_TABLE, REDIS\_TABLE\_GETRESPONSE, TEMP\_PREFIX, modifyRedis);  *// Create notification processor and handler*  m\_notifications = std::make\_shared<RedisNotificationProducer>(m\_contextConfig->m\_dbAsic);  m\_client = std::make\_shared<RedisClient>(m\_dbAsic);  m\_processor = std::make\_shared<NotificationProcessor>(m\_notifications, m\_client, std::bind(&Syncd::syncProcessNotification, this, \_1));  m\_handler = std::make\_shared<NotificationHandler>(m\_processor);  m\_sn.onFdbEvent = std::bind(&NotificationHandler::onFdbEvent, m\_handler.get(), \_1, \_2);  m\_sn.onNatEvent = std::bind(&NotificationHandler::onNatEvent, m\_handler.get(), \_1, \_2);  *// Init many other event handlers here*  m\_handler->setSwitchNotifications(m\_sn.getSwitchNotifications());  ...  *// Initialize SAI*  sai\_status\_t status = vendorSai->initialize(0, &m\_test\_services);  ... } |

### **1.3. SAI的初始化与VendorSai**

为了有一个更加直观的理解，我们拿一小部分代码来展示一下SAI的接口定义和初始化的方法，如下：

|  |
| --- |
| *// File: meta/saimetadata.h* typedef struct \_sai\_apis\_t {  sai\_switch\_api\_t\* switch\_api;  sai\_port\_api\_t\* port\_api;  ... } sai\_apis\_t; *// File: inc/saiswitch.h* typedef struct \_sai\_switch\_api\_t {  sai\_create\_switch\_fn create\_switch;  sai\_remove\_switch\_fn remove\_switch;  sai\_set\_switch\_attribute\_fn set\_switch\_attribute;  sai\_get\_switch\_attribute\_fn get\_switch\_attribute;  ... } sai\_switch\_api\_t; *// File: inc/saiport.h* typedef struct \_sai\_port\_api\_t {  sai\_create\_port\_fn create\_port;  sai\_remove\_port\_fn remove\_port;  sai\_set\_port\_attribute\_fn set\_port\_attribute;  sai\_get\_port\_attribute\_fn get\_port\_attribute;  ... } sai\_port\_api\_t; |

其中，sai\_apis\_t结构体是SAI所有模块的接口的集合，其中每个成员都是一个特定模块的接口列表的指针。我们用sai\_switch\_api\_t来举例，它定义了SAI Switch模块的所有接口，我们在inc/saiswitch.h中可以看到它的定义。同样的，我们在inc/saiport.h中可以看到SAI Port模块的接口定义。

SAI的初始化其实就是想办法获取上面这些函数指针，这样我们就可以通过SAI的接口来操作ASIC了。

参与SAI初始化的主要函数有两个，他们都定义在inc/sai.h中：

* sai\_api\_initialize：初始化SAI
* sai\_api\_query：传入SAI的API的类型，获取对应的接口列表

虽然大部分厂商的SAI实现是闭源的，但是mellanox却开源了自己的SAI实现，所以这里我们可以借助其更加深入的理解SAI是如何工作的。

比如，sai\_api\_initialize函数其实就是简单的设置设置两个全局变量，然后返回SAI\_STATUS\_SUCCESS：

|  |
| --- |
| *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_interfacequery.c* sai\_status\_t sai\_api\_initialize(\_In\_ uint64\_t flags, \_In\_ const sai\_service\_method\_table\_t\* services) {  if (g\_initialized) {  return SAI\_STATUS\_FAILURE;  }  *// Validate parameters here (code emitted)*  memcpy(&g\_mlnx\_services, services, sizeof(g\_mlnx\_services));  g\_initialized = true;  return SAI\_STATUS\_SUCCESS; } |

初始化完成后，我们就可以使用sai\_api\_query函数，通过传入API的类型来查询对应的接口列表，而每一个接口列表其实都是一个全局变量：

|  |
| --- |
| *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_interfacequery.c* sai\_status\_t sai\_api\_query(\_In\_ sai\_api\_t sai\_api\_id, \_Out\_ void\*\* api\_method\_table) {  if (!g\_initialized) {  return SAI\_STATUS\_UNINITIALIZED;  }  ...  return sai\_api\_query\_eth(sai\_api\_id, api\_method\_table); } *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_interfacequery\_eth.c* sai\_status\_t sai\_api\_query\_eth(\_In\_ sai\_api\_t sai\_api\_id, \_Out\_ void\*\* api\_method\_table) {  switch (sai\_api\_id) {  case SAI\_API\_BRIDGE:  \*(const sai\_bridge\_api\_t\*\*)api\_method\_table = &mlnx\_bridge\_api;  return SAI\_STATUS\_SUCCESS;  case SAI\_API\_SWITCH:  \*(const sai\_switch\_api\_t\*\*)api\_method\_table = &mlnx\_switch\_api;  return SAI\_STATUS\_SUCCESS;  ...  default:  if (sai\_api\_id >= (sai\_api\_t)SAI\_API\_EXTENSIONS\_RANGE\_END) {  return SAI\_STATUS\_INVALID\_PARAMETER;  } else {  return SAI\_STATUS\_NOT\_IMPLEMENTED;  }  } } *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_bridge.c* const sai\_bridge\_api\_t mlnx\_bridge\_api = {  mlnx\_create\_bridge,  mlnx\_remove\_bridge,  mlnx\_set\_bridge\_attribute,  mlnx\_get\_bridge\_attribute,  ... }; *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_switch.c* const sai\_switch\_api\_t mlnx\_switch\_api = {  mlnx\_create\_switch,  mlnx\_remove\_switch,  mlnx\_set\_switch\_attribute,  mlnx\_get\_switch\_attribute,  ... }; |

Syncd使用VendorSai来对SAI的所有API进行封装，方便上层调用。其初始化过程也非常直接，基本就是对上面两个函数的直接调用和错误处理，如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/VendorSai.cpp* sai\_status\_t VendorSai::initialize(  \_In\_ uint64\_t flags,  \_In\_ const sai\_service\_method\_table\_t \*service\_method\_table) {  ...  *// Initialize SAI*  memcpy(&m\_service\_method\_table, service\_method\_table, sizeof(m\_service\_method\_table));  auto status = sai\_api\_initialize(flags, service\_method\_table);  *// If SAI is initialized successfully, query all SAI API methods.*  *// sai\_metadata\_api\_query will also update all extern global sai\_\*\_api variables, so we can also use*  *// sai\_metadata\_get\_object\_type\_info to get methods for a specific SAI object type.*  if (status == SAI\_STATUS\_SUCCESS) {  memset(&m\_apis, 0, sizeof(m\_apis));  int failed = sai\_metadata\_apis\_query(sai\_api\_query, &m\_apis);  ...  }  ...  return status; } |

当获取好所有的SAI API之后，我们就可以通过VendorSai对象来调用SAI的API了。当前调用SAI的API方式主要有两种。

第一种是通过sai\_object\_type\_into\_t来调用，它类似于为所有的SAI Object实现了一个虚表，如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/VendorSai.cpp* sai\_status\_t VendorSai::set(  \_In\_ sai\_object\_type\_t objectType,  \_In\_ sai\_object\_id\_t objectId,  \_In\_ const sai\_attribute\_t \*attr) {  ...  auto info = sai\_metadata\_get\_object\_type\_info(objectType);  sai\_object\_meta\_key\_t mk = { .objecttype = objectType, .objectkey = { .key = { .object\_id = objectId } } };  return info->set(&mk, attr); } |

另外一种是通过保存在VendorSai对象中的m\_apis来调用，这种方式更加直接，但是调用前需要先根据SAI Object的类型来调用不同的API。

|  |
| --- |
| sai\_status\_t VendorSai::getStatsExt(  \_In\_ sai\_object\_type\_t object\_type,  \_In\_ sai\_object\_id\_t object\_id,  \_In\_ uint32\_t number\_of\_counters,  \_In\_ const sai\_stat\_id\_t \*counter\_ids,  \_In\_ sai\_stats\_mode\_t mode,  \_Out\_ uint64\_t \*counters) {  sai\_status\_t (\*ptr)(  \_In\_ sai\_object\_id\_t port\_id,  \_In\_ uint32\_t number\_of\_counters,  \_In\_ const sai\_stat\_id\_t \*counter\_ids,  \_In\_ sai\_stats\_mode\_t mode,  \_Out\_ uint64\_t \*counters);  switch ((int)object\_type)  {  case SAI\_OBJECT\_TYPE\_PORT:  ptr = m\_apis.port\_api->get\_port\_stats\_ext;  break;  case SAI\_OBJECT\_TYPE\_ROUTER\_INTERFACE:  ptr = m\_apis.router\_interface\_api->get\_router\_interface\_stats\_ext;  break;  case SAI\_OBJECT\_TYPE\_POLICER:  ptr = m\_apis.policer\_api->get\_policer\_stats\_ext;  break;  ...  default:  SWSS\_LOG\_ERROR("not implemented, FIXME");  return SAI\_STATUS\_FAILURE;  }  return ptr(object\_id, number\_of\_counters, counter\_ids, mode, counters); } |

可以明显看出，第一种调用方式代码要精炼和直观许多。

### **1.4. Syncd主循环**

Syncd的主循环也是使用的SONiC中标准的[事件分发](http://r12f.com/posts/sonic-5-syncd-sai-workflow/4-3-event-polling-and-error-handling.html)机制：在启动时，Syncd会将所有用于事件处理的Selectable对象注册到用于获取事件的Select对象中，然后在主循环中调用Select的select方法，等待事件的发生。核心代码如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/Syncd.cpp* void Syncd::run() {  volatile bool runMainLoop = true;  std::shared\_ptr<swss::Select> s = std::make\_shared<swss::Select>();  onSyncdStart(m\_commandLineOptions->m\_startType == SAI\_START\_TYPE\_WARM\_BOOT);  *// Start notification processing thread*  m\_processor->startNotificationsProcessingThread();  *// Start MDIO threads*  for (auto& sw: m\_switches) { m\_mdioIpcServer->setSwitchId(sw.second->getRid()); }  m\_mdioIpcServer->startMdioThread();  *// Registering selectable for event polling*  s->addSelectable(m\_selectableChannel.get());  s->addSelectable(m\_restartQuery.get());  s->addSelectable(m\_flexCounter.get());  s->addSelectable(m\_flexCounterGroup.get());   *// Main event loop*  while (runMainLoop)  {  swss::Selectable \*sel = NULL;  int result = s->select(&sel);  ...  if (sel == m\_restartQuery.get()) {  *// Handling switch restart event and restart switch here.*  } else if (sel == m\_flexCounter.get()) {  processFlexCounterEvent(\*(swss::ConsumerTable\*)sel);  } else if (sel == m\_flexCounterGroup.get()) {  processFlexCounterGroupEvent(\*(swss::ConsumerTable\*)sel);  } else if (sel == m\_selectableChannel.get()) {  *// Handle redis updates here.*  processEvent(\*m\_selectableChannel.get());  } else {  SWSS\_LOG\_ERROR("select failed: %d", result);  }  ...  }  ... } |

其中，m\_selectableChannel就是主要负责处理Redis数据库中的事件的对象。它使用[ProducerTable / ConsumerTable](http://r12f.com/posts/sonic-5-syncd-sai-workflow/4-2-2-redis-messaging-layer.md" \l "producertable--consumertable)的方式与Redis数据库进行交互，所以，所有orchagent发送过来的操作都会以三元组的形式保存在Redis中的list中，等待Syncd的处理。其核心定义如下：

|  |
| --- |
| *// File: src/sonic-sairedis/meta/RedisSelectableChannel.h* class RedisSelectableChannel: public SelectableChannel {  public:  RedisSelectableChannel(  \_In\_ std::shared\_ptr<swss::DBConnector> dbAsic,  \_In\_ const std::string& asicStateTable,  \_In\_ const std::string& getResponseTable,  \_In\_ const std::string& tempPrefix,  \_In\_ bool modifyRedis);  public: *// SelectableChannel overrides*  virtual bool empty() override;  ...  public: *// Selectable overrides*  virtual int getFd() override;  virtual uint64\_t readData() override;  ...   private:  std::shared\_ptr<swss::DBConnector> m\_dbAsic;  std::shared\_ptr<swss::ConsumerTable> m\_asicState;  std::shared\_ptr<swss::ProducerTable> m\_getResponse;  ... }; |

另外，在主循环启动时，Syncd还会额外启动两个线程：

* 用于接收ASIC上报通知的通知处理线程：m\_processor->startNotificationsProcessingThread();
* 用于处理MDIO通信的MDIO IPC处理线程：m\_mdioIpcServer->startMdioThread();

它们的细节我们在初始化的部分不做过多展开，等后面介绍相关工作流时再来详细介绍。

### **1.5. 创建Switch对象，初始化通知机制**

在主循环启动后，Syncd就会开始调用SAI的API来创建Switch对象，这里的入口有两个，一个是ASIC\_DB收到创建Switch的通知，另外一个是Warm Boot时，Syncd来主动调用，但是创建Switch这一步的内部流程都类似。

在这一步中间，有一个很重要的步骤，就是初始化SAI内部实现中的通知回调，将我们之前已经创建好的通知处理逻辑传递给SAI的实现，比如FDB的事件等等。这些回调函数会被当做Switch的属性（Attributes）通过参数的形式传给SAI的create\_switch方法，SAI的实现会将其保存起来，这样就可以在事件发生时调用回调函数，来通知Syncd了。这里的核心代码如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/Syncd.cpp* sai\_status\_t Syncd::processQuadEvent(  \_In\_ sai\_common\_api\_t api,  \_In\_ const swss::KeyOpFieldsValuesTuple &kco) {  *// Parse event into SAI object*  sai\_object\_meta\_key\_t metaKey;  ...  SaiAttributeList list(metaKey.objecttype, values, false);  sai\_attribute\_t \*attr\_list = list.get\_attr\_list();  uint32\_t attr\_count = list.get\_attr\_count();   *// Update notifications pointers in attribute list*  if (metaKey.objecttype == SAI\_OBJECT\_TYPE\_SWITCH && (api == SAI\_COMMON\_API\_CREATE || api == SAI\_COMMON\_API\_SET))  {  m\_handler->updateNotificationsPointers(attr\_count, attr\_list);  }  if (isInitViewMode())  {  *// ProcessQuadEventInInitViewMode will eventually call into VendorSai, which calls create\_swtich function in SAI.*  sai\_status\_t status = processQuadEventInInitViewMode(metaKey.objecttype, strObjectId, api, attr\_count, attr\_list);  syncUpdateRedisQuadEvent(status, api, kco);  return status;  }  ... }  *// File: src/sonic-sairedis/syncd/NotificationHandler.cpp* void NotificationHandler::updateNotificationsPointers(\_In\_ uint32\_t attr\_count, \_In\_ sai\_attribute\_t \*attr\_list) const {  for (uint32\_t index = 0; index < attr\_count; ++index) {  ...  sai\_attribute\_t &attr = attr\_list[index];  switch (attr.id) {  ...  case SAI\_SWITCH\_ATTR\_SHUTDOWN\_REQUEST\_NOTIFY:  attr.value.ptr = (void\*)m\_switchNotifications.on\_switch\_shutdown\_request;  break;  case SAI\_SWITCH\_ATTR\_FDB\_EVENT\_NOTIFY:  attr.value.ptr = (void\*)m\_switchNotifications.on\_fdb\_event;  break;  ...  }  ...  } } *// File: src/sonic-sairedis/syncd/Syncd.cpp* *// Call stack: processQuadEvent* *// -> processQuadEventInInitViewMode* *// -> processQuadInInitViewModeCreate* *// -> onSwitchCreateInInitViewMode* void Syncd::onSwitchCreateInInitViewMode(\_In\_ sai\_object\_id\_t switchVid, \_In\_ uint32\_t attr\_count, \_In\_ const sai\_attribute\_t \*attr\_list) {  if (m\_switches.find(switchVid) == m\_switches.end()) {  sai\_object\_id\_t switchRid;  sai\_status\_t status;  status = m\_vendorSai->create(SAI\_OBJECT\_TYPE\_SWITCH, &switchRid, 0, attr\_count, attr\_list);  ...  m\_switches[switchVid] = std::make\_shared<SaiSwitch>(switchVid, switchRid, m\_client, m\_translator, m\_vendorSai);  m\_mdioIpcServer->setSwitchId(switchRid);  ...  }  ... } |

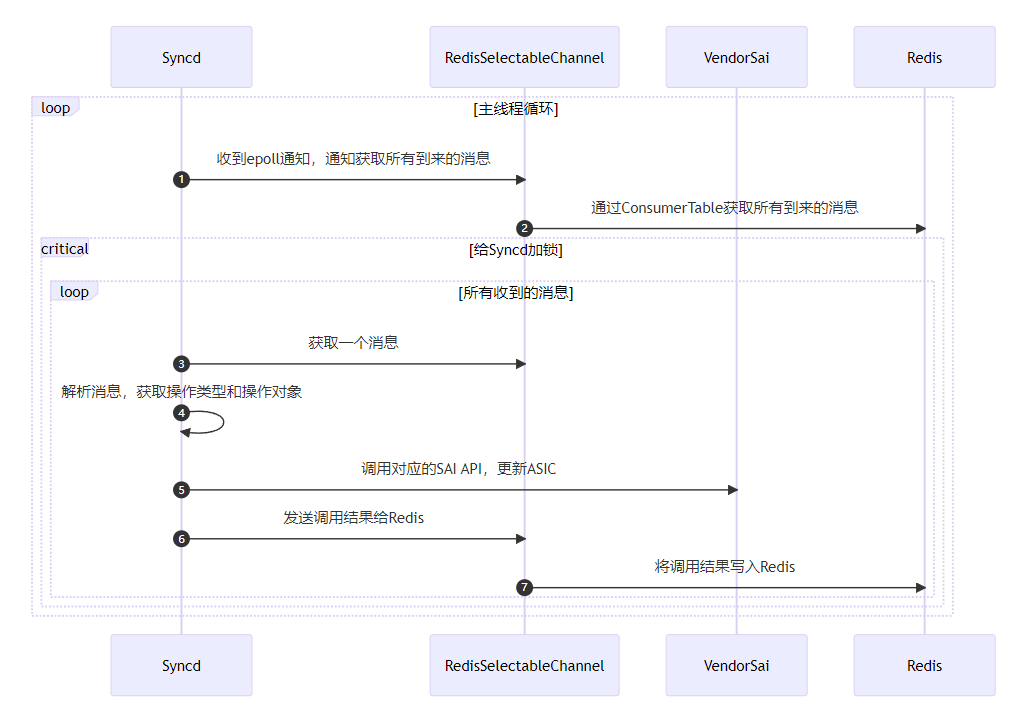
从Mellanox的SAI实现，我们可以看到其具体的保存的方法：

|  |
| --- |
| static sai\_status\_t mlnx\_create\_switch(\_Out\_ sai\_object\_id\_t \* switch\_id,  \_In\_ uint32\_t attr\_count,  \_In\_ const sai\_attribute\_t \*attr\_list) {  ...   status = find\_attrib\_in\_list(attr\_count, attr\_list, SAI\_SWITCH\_ATTR\_SWITCH\_STATE\_CHANGE\_NOTIFY, &attr\_val, &attr\_idx);  if (!SAI\_ERR(status)) {  g\_notification\_callbacks.on\_switch\_state\_change = (sai\_switch\_state\_change\_notification\_fn)attr\_val->ptr;  }  status = find\_attrib\_in\_list(attr\_count, attr\_list, SAI\_SWITCH\_ATTR\_SHUTDOWN\_REQUEST\_NOTIFY, &attr\_val, &attr\_idx);  if (!SAI\_ERR(status)) {  g\_notification\_callbacks.on\_switch\_shutdown\_request =  (sai\_switch\_shutdown\_request\_notification\_fn)attr\_val->ptr;  }  status = find\_attrib\_in\_list(attr\_count, attr\_list, SAI\_SWITCH\_ATTR\_FDB\_EVENT\_NOTIFY, &attr\_val, &attr\_idx);  if (!SAI\_ERR(status)) {  g\_notification\_callbacks.on\_fdb\_event = (sai\_fdb\_event\_notification\_fn)attr\_val->ptr;  }  status = find\_attrib\_in\_list(attr\_count, attr\_list, SAI\_SWITCH\_ATTR\_PORT\_STATE\_CHANGE\_NOTIFY, &attr\_val, &attr\_idx);  if (!SAI\_ERR(status)) {  g\_notification\_callbacks.on\_port\_state\_change = (sai\_port\_state\_change\_notification\_fn)attr\_val->ptr;  }  status = find\_attrib\_in\_list(attr\_count, attr\_list, SAI\_SWITCH\_ATTR\_PACKET\_EVENT\_NOTIFY, &attr\_val, &attr\_idx);  if (!SAI\_ERR(status)) {  g\_notification\_callbacks.on\_packet\_event = (sai\_packet\_event\_notification\_fn)attr\_val->ptr;  }  ... } |

## **2. ASIC状态更新**

ASIC状态更新是Syncd中最重要的工作流之一，当orchagent发现任何变化并开始修改ASIC\_DB时，就会触发该工作流，通过SAI来对ASIC进行更新。在了解了Syncd的主循环之后，理解ASIC状态更新的工作流就很简单了。

所有的步骤都发生在主线程一个线程中，顺序执行，总结成时序图如下：

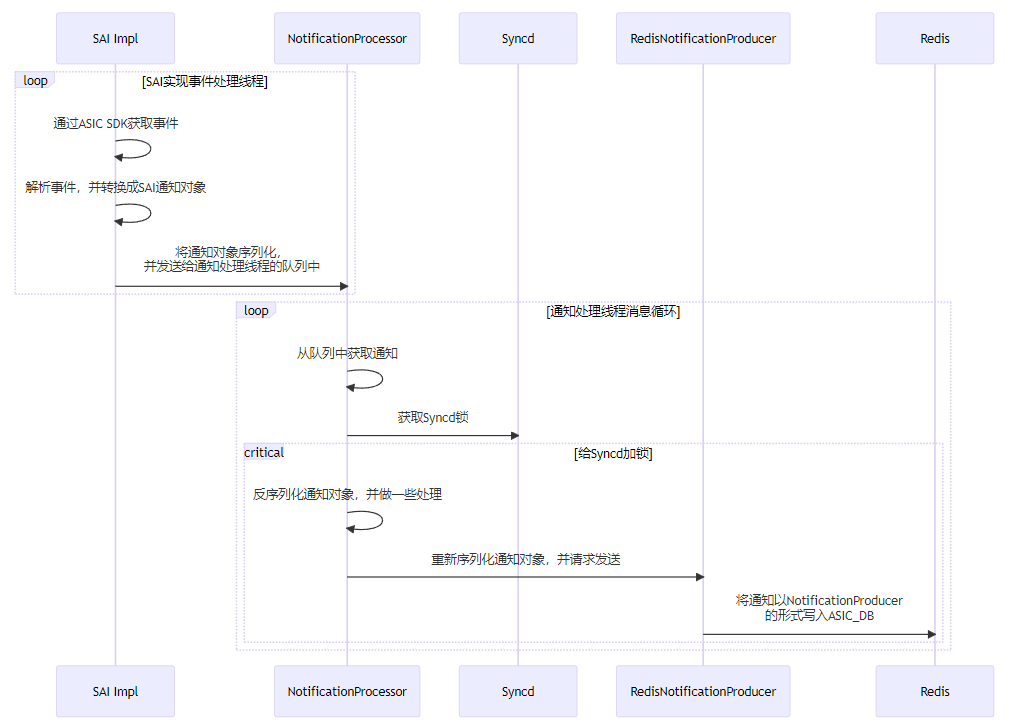
首先，orchagent通过Redis发送过来的操作会被RedisSelectableChannel对象接收，然后在主循环中被处理。当Syncd处理到m\_selectableChannel时，就会调用processEvent方法来处理该操作。这几步的核心代码我们上面介绍主循环时已经介绍过了，这里就不再赘述。

然后，processEvent会根据其中的操作类型，调用对应的SAI的API来对ASIC进行更新。其逻辑是一个巨大的switch-case语句，如下：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/Syncd.cpp* void Syncd::processEvent(\_In\_ sairedis::SelectableChannel& consumer) {  *// Loop all operations in the queue*  std::lock\_guard<std::mutex> lock(m\_mutex);  do {  swss::KeyOpFieldsValuesTuple kco;  consumer.pop(kco, isInitViewMode());  processSingleEvent(kco);  } while (!consumer.empty()); } sai\_status\_t Syncd::processSingleEvent(\_In\_ const swss::KeyOpFieldsValuesTuple &kco) {  auto& op = kfvOp(kco);  ...  if (op == REDIS\_ASIC\_STATE\_COMMAND\_CREATE)  return processQuadEvent(SAI\_COMMON\_API\_CREATE, kco);  if (op == REDIS\_ASIC\_STATE\_COMMAND\_REMOVE)  return processQuadEvent(SAI\_COMMON\_API\_REMOVE, kco);  ... } sai\_status\_t Syncd::processQuadEvent(  \_In\_ sai\_common\_api\_t api,  \_In\_ const swss::KeyOpFieldsValuesTuple &kco) {  *// Parse operation*  const std::string& key = kfvKey(kco);  const std::string& strObjectId = key.substr(key.find(":") + 1);  sai\_object\_meta\_key\_t metaKey;  sai\_deserialize\_object\_meta\_key(key, metaKey);  auto& values = kfvFieldsValues(kco);  SaiAttributeList list(metaKey.objecttype, values, false);  sai\_attribute\_t \*attr\_list = list.get\_attr\_list();  uint32\_t attr\_count = list.get\_attr\_count();  ...  auto info = sai\_metadata\_get\_object\_type\_info(metaKey.objecttype);  *// Process the operation*  sai\_status\_t status;  if (info->isnonobjectid) {  status = processEntry(metaKey, api, attr\_count, attr\_list);  } else {  status = processOid(metaKey.objecttype, strObjectId, api, attr\_count, attr\_list);  }  *// Send response*  if (api == SAI\_COMMON\_API\_GET) {  sai\_object\_id\_t switchVid = VidManager::switchIdQuery(metaKey.objectkey.key.object\_id);  sendGetResponse(metaKey.objecttype, strObjectId, switchVid, status, attr\_count, attr\_list);  ...  } else {  sendApiResponse(api, status);  }  syncUpdateRedisQuadEvent(status, api, kco);  return status; } sai\_status\_t Syncd::processEntry(\_In\_ sai\_object\_meta\_key\_t metaKey, \_In\_ sai\_common\_api\_t api,  \_In\_ uint32\_t attr\_count, \_In\_ sai\_attribute\_t \*attr\_list) {  ...  switch (api)  {  case SAI\_COMMON\_API\_CREATE:  return m\_vendorSai->create(metaKey, SAI\_NULL\_OBJECT\_ID, attr\_count, attr\_list);  case SAI\_COMMON\_API\_REMOVE:  return m\_vendorSai->remove(metaKey);  ...  default:  SWSS\_LOG\_THROW("api %s not supported", sai\_serialize\_common\_api(api).c\_str());  } } |

## **3. ASIC状态变更上报**

反过来，当ASIC状态发生任何变化，或者需要上报数据，它也会通过SAI来通知我们，此时Syncd会监听这些通知，然后通过ASIC\_DB上报给orchagent。其主要工作流如下：



这里我们也来看一下具体的实现。为了更加深入的理解，我们还是借助开源的Mellanox的SAI实现来进行分析。

最开始，SAI的实现需要接受到ASIC的通知，这一步是通过ASIC的SDK来实现的，Mellanox的SAI会创建一个事件处理线程（event\_thread），然后使用select函数来获取并处理ASIC发送过来的通知，核心代码如下：

|  |
| --- |
| *// File: platform/mellanox/mlnx-sai/SAI-Implementation/mlnx\_sai/src/mlnx\_sai\_switch.c* static void event\_thread\_func(void \*context) { #define MAX\_PACKET\_SIZE MAX(g\_resource\_limits.port\_mtu\_max, SX\_HOST\_EVENT\_BUFFER\_SIZE\_MAX)  sx\_status\_t status;  sx\_api\_handle\_t api\_handle;  sx\_user\_channel\_t port\_channel, callback\_channel;  fd\_set descr\_set;  int ret\_val;  sai\_object\_id\_t switch\_id = (sai\_object\_id\_t)context;  sai\_port\_oper\_status\_notification\_t port\_data;  sai\_fdb\_event\_notification\_data\_t \*fdb\_events = NULL;  sai\_attribute\_t \*attr\_list = NULL;  ...  *// Init SDK API*  if (SX\_STATUS\_SUCCESS != (status = sx\_api\_open(sai\_log\_cb, &api\_handle))) {  if (g\_notification\_callbacks.on\_switch\_shutdown\_request) {  g\_notification\_callbacks.on\_switch\_shutdown\_request(switch\_id);  }  return;  }  if (SX\_STATUS\_SUCCESS != (status = sx\_api\_host\_ifc\_open(api\_handle, &port\_channel.channel.fd))) {  goto out;  }  ...  *// Register for port and channel notifications*  port\_channel.type = SX\_USER\_CHANNEL\_TYPE\_FD;  if (SX\_STATUS\_SUCCESS != (status = sx\_api\_host\_ifc\_trap\_id\_register\_set(api\_handle, SX\_ACCESS\_CMD\_REGISTER, DEFAULT\_ETH\_SWID, SX\_TRAP\_ID\_PUDE, &port\_channel))) {  goto out;  }  ...  for (uint32\_t ii = 0; ii < (sizeof(mlnx\_trap\_ids) / sizeof(\*mlnx\_trap\_ids)); ii++) {  status = sx\_api\_host\_ifc\_trap\_id\_register\_set(api\_handle, SX\_ACCESS\_CMD\_REGISTER, DEFAULT\_ETH\_SWID, mlnx\_trap\_ids[ii], &callback\_channel);  }  while (!event\_thread\_asked\_to\_stop) {  FD\_ZERO(&descr\_set);  FD\_SET(port\_channel.channel.fd.fd, &descr\_set);  FD\_SET(callback\_channel.channel.fd.fd, &descr\_set);  ...   ret\_val = select(FD\_SETSIZE, &descr\_set, NULL, NULL, &timeout);  if (ret\_val > 0) {  *// Port state change event*  if (FD\_ISSET(port\_channel.channel.fd.fd, &descr\_set)) {  *// Parse port state event here ...*  if (g\_notification\_callbacks.on\_port\_state\_change) {  g\_notification\_callbacks.on\_port\_state\_change(1, &port\_data);  }  }  if (FD\_ISSET(callback\_channel.channel.fd.fd, &descr\_set)) {  *// Receive notification event.*  packet\_size = MAX\_PACKET\_SIZE;  if (SX\_STATUS\_SUCCESS != (status = sx\_lib\_host\_ifc\_recv(&callback\_channel.channel.fd, p\_packet, &packet\_size, receive\_info))) {  goto out;  }   *// BFD packet event*  if (SX\_TRAP\_ID\_BFD\_PACKET\_EVENT == receive\_info->trap\_id) {  const struct bfd\_packet\_event \*event = (const struct bfd\_packet\_event\*)p\_packet;  *// Parse and check event valid here ...*  status = mlnx\_switch\_bfd\_packet\_handle(event);  continue;  }  *// Same way to handle BFD timeout event, Bulk counter ready event. Emiited.*  *// FDB event and packet event handling*  if (receive\_info->trap\_id == SX\_TRAP\_ID\_FDB\_EVENT) {  trap\_name = "FDB event";  } else if (SAI\_STATUS\_SUCCESS != (status = mlnx\_translate\_sdk\_trap\_to\_sai(receive\_info->trap\_id, &trap\_name, &trap\_oid))) {  continue;  }  if (SX\_TRAP\_ID\_FDB\_EVENT == receive\_info->trap\_id) {  *// Parse FDB events here ...*  if (g\_notification\_callbacks.on\_fdb\_event) {  g\_notification\_callbacks.on\_fdb\_event(event\_count, fdb\_events);  }  continue;  }  *// Packet event handling*  status = mlnx\_get\_hostif\_packet\_data(receive\_info, &attrs\_num, callback\_data);  if (g\_notification\_callbacks.on\_packet\_event) {  g\_notification\_callbacks.on\_packet\_event(switch\_id, packet\_size, p\_packet, attrs\_num, callback\_data);  }  }  }  } out:  ... } |

接下来，我们用FDB事件来举例，当ASIC收到FDB事件，就会被上面的事件处理循环获取到，并调用g\_notification\_callbacks.on\_fdb\_event函数来处理。这个函数接下来就会调用到Syncd初始化时设置好的NotificationHandler::onFdbEvent函数，这个函数会将该事件序列化后，通过消息队列转发给通知处理线程来进行处理：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/NotificationHandler.cpp* void NotificationHandler::onFdbEvent(\_In\_ uint32\_t count, \_In\_ const sai\_fdb\_event\_notification\_data\_t \*data) {  std::string s = sai\_serialize\_fdb\_event\_ntf(count, data);  enqueueNotification(SAI\_SWITCH\_NOTIFICATION\_NAME\_FDB\_EVENT, s); } |

而此时通知处理线程会被唤醒，从消息队列中取出该事件，然后通过Syncd获取到Syncd的锁，再开始处理该通知：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/NotificationProcessor.cpp* void NotificationProcessor::ntf\_process\_function() {  std::mutex ntf\_mutex;  std::unique\_lock<std::mutex> ulock(ntf\_mutex);  while (m\_runThread) {  *// When notification arrives, it will signal this condition variable.*  m\_cv.wait(ulock);  *// Process notifications in the queue.*  swss::KeyOpFieldsValuesTuple item;  while (m\_notificationQueue->tryDequeue(item)) {  processNotification(item);  }  } } *// File: src/sonic-sairedis/syncd/Syncd.cpp* *// Call from NotificationProcessor::processNotification* void Syncd::syncProcessNotification(\_In\_ const swss::KeyOpFieldsValuesTuple& item) {  std::lock\_guard<std::mutex> lock(m\_mutex);  m\_processor->syncProcessNotification(item); } |

接下来就是事件的分发和处理了，syncProcessNotification函数是一系列的if-else语句，根据事件的类型，调用不同的处理函数来处理该事件：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/NotificationProcessor.cpp* void NotificationProcessor::syncProcessNotification( \_In\_ const swss::KeyOpFieldsValuesTuple& item) {  std::string notification = kfvKey(item);  std::string data = kfvOp(item);  if (notification == SAI\_SWITCH\_NOTIFICATION\_NAME\_SWITCH\_STATE\_CHANGE) {  handle\_switch\_state\_change(data);  } else if (notification == SAI\_SWITCH\_NOTIFICATION\_NAME\_FDB\_EVENT) {  handle\_fdb\_event(data);  } else if ...  } else {  SWSS\_LOG\_ERROR("unknown notification: %s", notification.c\_str());  } } |

而每个事件处理函数都类似，他们会对发送过来的事件进行反序列化，然后调用真正的处理逻辑发送通知，比如，fdb事件对应的handle\_fdb\_event函数和process\_on\_fdb\_event：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/NotificationProcessor.cpp* void NotificationProcessor::handle\_fdb\_event(\_In\_ const std::string &data) {  uint32\_t count;  sai\_fdb\_event\_notification\_data\_t \*fdbevent = NULL;  sai\_deserialize\_fdb\_event\_ntf(data, count, &fdbevent);  process\_on\_fdb\_event(count, fdbevent);  sai\_deserialize\_free\_fdb\_event\_ntf(count, fdbevent); } void NotificationProcessor::process\_on\_fdb\_event( \_In\_ uint32\_t count, \_In\_ sai\_fdb\_event\_notification\_data\_t \*data) {  for (uint32\_t i = 0; i < count; i++) {  sai\_fdb\_event\_notification\_data\_t \*fdb = &data[i];  *// Check FDB event notification data here*  fdb->fdb\_entry.switch\_id = m\_translator->translateRidToVid(fdb->fdb\_entry.switch\_id, SAI\_NULL\_OBJECT\_ID);  fdb->fdb\_entry.bv\_id = m\_translator->translateRidToVid(fdb->fdb\_entry.bv\_id, fdb->fdb\_entry.switch\_id, true);  m\_translator->translateRidToVid(SAI\_OBJECT\_TYPE\_FDB\_ENTRY, fdb->fdb\_entry.switch\_id, fdb->attr\_count, fdb->attr, true);  ...  }  *// Send notification*  std::string s = sai\_serialize\_fdb\_event\_ntf(count, data);  sendNotification(SAI\_SWITCH\_NOTIFICATION\_NAME\_FDB\_EVENT, s); } |

具体发送事件的逻辑就非常直接了，最终就是通过[NotificationProducer](http://r12f.com/posts/sonic-5-syncd-sai-workflow/4-2-2-redis-messaging-layer.html" \l "notificationproducer--notificationconsumer)来发送通知到ASIC\_DB中：

|  |
| --- |
| *// File: src/sonic-sairedis/syncd/NotificationProcessor.cpp* void NotificationProcessor::sendNotification(\_In\_ const std::string& op, \_In\_ const std::string& data) {  std::vector<swss::FieldValueTuple> entry;  sendNotification(op, data, entry); } void NotificationProcessor::sendNotification(\_In\_ const std::string& op, \_In\_ const std::string& data, \_In\_ std::vector<swss::FieldValueTuple> entry) {  m\_notifications->send(op, data, entry); } *// File: src/sonic-sairedis/syncd/RedisNotificationProducer.cpp* void RedisNotificationProducer::send(\_In\_ const std::string& op, \_In\_ const std::string& data, \_In\_ const std::vector<swss::FieldValueTuple>& values) {  std::vector<swss::FieldValueTuple> vals = values;  *// The m\_notificationProducer is created in the ctor of RedisNotificationProducer as below:*  *// m\_notificationProducer = std::make\_shared<swss::NotificationProducer>(m\_db.get(), REDIS\_TABLE\_NOTIFICATIONS\_PER\_DB(dbName));*  m\_notificationProducer->send(op, data, vals); } |

到此，Syncd中的通知上报的流程就结束了。

## **4. 参考资料**

[SONiC Architecture](https://github.com/sonic-net/SONiC/wiki/Architecture" \t "http://r12f.com/posts/sonic-5-syncd-sai-workflow/_blank)

[Github repo: sonic-sairedis](https://github.com/sonic-net/sonic-sairedis/" \t "http://r12f.com/posts/sonic-5-syncd-sai-workflow/_blank)

[Github repo: Nvidia (Mellanox) SAI implementation](https://github.com/Mellanox/SAI-Implementation/tree/master" \t "http://r12f.com/posts/sonic-5-syncd-sai-workflow/_blank)