转载自：soul orbit: <http://r12f.com/posts/sonic-4-communication/>

# SONiC学习笔记（四）：通信机制

在大概了解了[各个服务](http://r12f.com/posts/sonic-2-key-components/)和[源码仓库](http://r12f.com/posts/sonic-3-code-repos/)之后，相信大家已经对可以开始自如的浏览SONiC的源码了，所以这一篇文章中，我们就来看看SONiC中最常用的组件 - 通信机制。

SONiC中主要的通信机制有两种：与内核的通信和基于Redis的服务间的通信。所有的实现都在[sonic-swss-common](https://github.com/sonic-net/sonic-swss-common" \t "http://r12f.com/posts/sonic-4-communication/_blank)这个repo中的common目录下。我们这里就来看看这两种通信机制的实现吧！

## **1. 与内核的通信**

### **1.1. 命令行调用**

SONiC中的与内核通信最简单的方式就是命令行调用了，其实现放在[common/exec.h](https://github.com/sonic-net/sonic-swss-common/blob/master/common/exec.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)文件下，且十分简单，接口如下：

|  |
| --- |
| *// File: common/exec.h* *// Namespace: swss* int exec(const std::string &cmd, std::string &stdout); |

其中，cmd是要执行的命令，stdout是命令执行的输出。这里的exec函数是一个同步调用，调用者会一直阻塞，直到命令执行完毕。其内部通过调用popen函数来创建子进程，并且通过fgets函数来获取输出。不过，****虽然这个函数返回了输出，但是基本上并没有人使用****，而只是通过返回值来判断是否成功，甚至连错误log中都不会写入输出的结果。

这个函数虽然粗暴，但是使用广泛，特别是在各个\*mgrd服务中，比如portmgrd中就用它来设置每一个Port的状态等等。

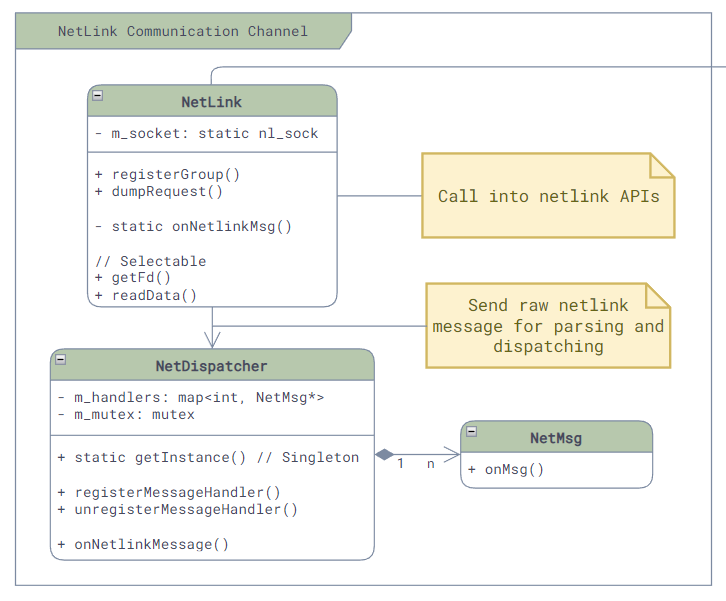
|  |
| --- |
| *// File: sonic-swss - cfgmgr/portmgr.cpp* bool PortMgr::setPortAdminStatus(const string &alias, const bool up) {  stringstream cmd;  string res, cmd\_str;  *// ip link set dev <port\_name> [up|down]*  cmd << IP\_CMD << " link set dev " << shellquote(alias) << (up ? " up" : " down");  cmd\_str = cmd.str();  int ret = swss::exec(cmd\_str, res);  *// ...* |

****为什么说命令行调用是一种通信机制呢****？原因是当\*mgrd服务调用exec函数对系统进行的修改，会触发下面马上会提到的netlink事件，从而通知其他服务进行相应的修改，比如\*syncd，这样就间接的构成了一种通信。所以这里我们把命令行调用看作一种通信机制能帮助我们以后更好的理解SONiC的各种工作流。

### **1.2. Netlink**

Netlink是Linux内核中用于内核与用户空间进程之间的一种基于消息的通信机制。它通过套接字接口和自定义的协议族来实现，可以用来传递各种类型的内核消息，包括网络设备状态、路由表更新、防火墙规则变化、系统资源使用情况等等。而SONiC的\*sync服务就大量使用了Netlink的机制来监听系统中网络设备的变化，并将最新的状态同步到Redis中，并通知其他服务进行相应的修改。

Netlink的实现主要在这几个文件中：[common/netmsg.\*](https://github.com/sonic-net/sonic-swss-common/blob/master/common/netmsg.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)、[common/netlink.\*](https://github.com/sonic-net/sonic-swss-common/blob/master/common/netlink.h" \t "http://r12f.com/posts/sonic-4-communication/_blank) 和 [common/netdispatcher.\*](https://github.com/sonic-net/sonic-swss-common/blob/master/common/netdispatcher.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)，具体类图如下：



其中：

* ****Netlink****：封装了Netlink的套接字接口，提供了Netlink消息的接口和接收消息的回调。
* ****NetDispatcher****：它是一个单例，提供了Handler注册的接口。当Netlink类接收到原始的消息后，就会调用NetDispatcher将其解析成nl\_onject，并根据消息的类型调用相应的Handler。
* ****NetMsg****：Netlink消息Handler的基类，仅提供了onMsg的接口，其中没有实现。

举一个例子，当portsyncd启动的时候，它会创建一个Netlink对象，用来监听Link相关的状态变化，并且会实现NetMsg的接口，对Link相关的消息进行处理。具体的实现如下：

|  |
| --- |
| *// File: sonic-swss - portsyncd/portsyncd.cpp* int main(int argc, char \*\*argv) {  *// ...*  *// Create Netlink object to listen to link messages*  NetLink netlink;  netlink.registerGroup(RTNLGRP\_LINK);  *// Here SONiC request a fulldump of current state, so that it can get the current state of all links*  netlink.dumpRequest(RTM\_GETLINK);   cout << "Listen to link messages..." << endl;  *// ...*  *// Register handler for link messages*  LinkSync sync(&appl\_db, &state\_db);  NetDispatcher::getInstance().registerMessageHandler(RTM\_NEWLINK, &sync);  NetDispatcher::getInstance().registerMessageHandler(RTM\_DELLINK, &sync);  *// ...* } |

上面的LinkSync，就是一个NetMsg的实现，它实现了onMsg接口，用来处理Link相关的消息：

|  |
| --- |
| *// File: sonic-swss - portsyncd/linksync.h* class LinkSync : public NetMsg { public:  LinkSync(DBConnector \*appl\_db, DBConnector \*state\_db);  *// NetMsg interface*  virtual void onMsg(int nlmsg\_type, struct nl\_object \*obj);  *// ...* }; *// File: sonic-swss - portsyncd/linksync.cpp* void LinkSync::onMsg(int nlmsg\_type, struct nl\_object \*obj) {  *// ...*  *// Write link state to Redis DB*  FieldValueTuple fv("oper\_status", oper ? "up" : "down");  vector<FieldValueTuple> fvs;  fvs.push\_back(fv);  m\_stateMgmtPortTable.set(key, fvs);  *// ...* } |

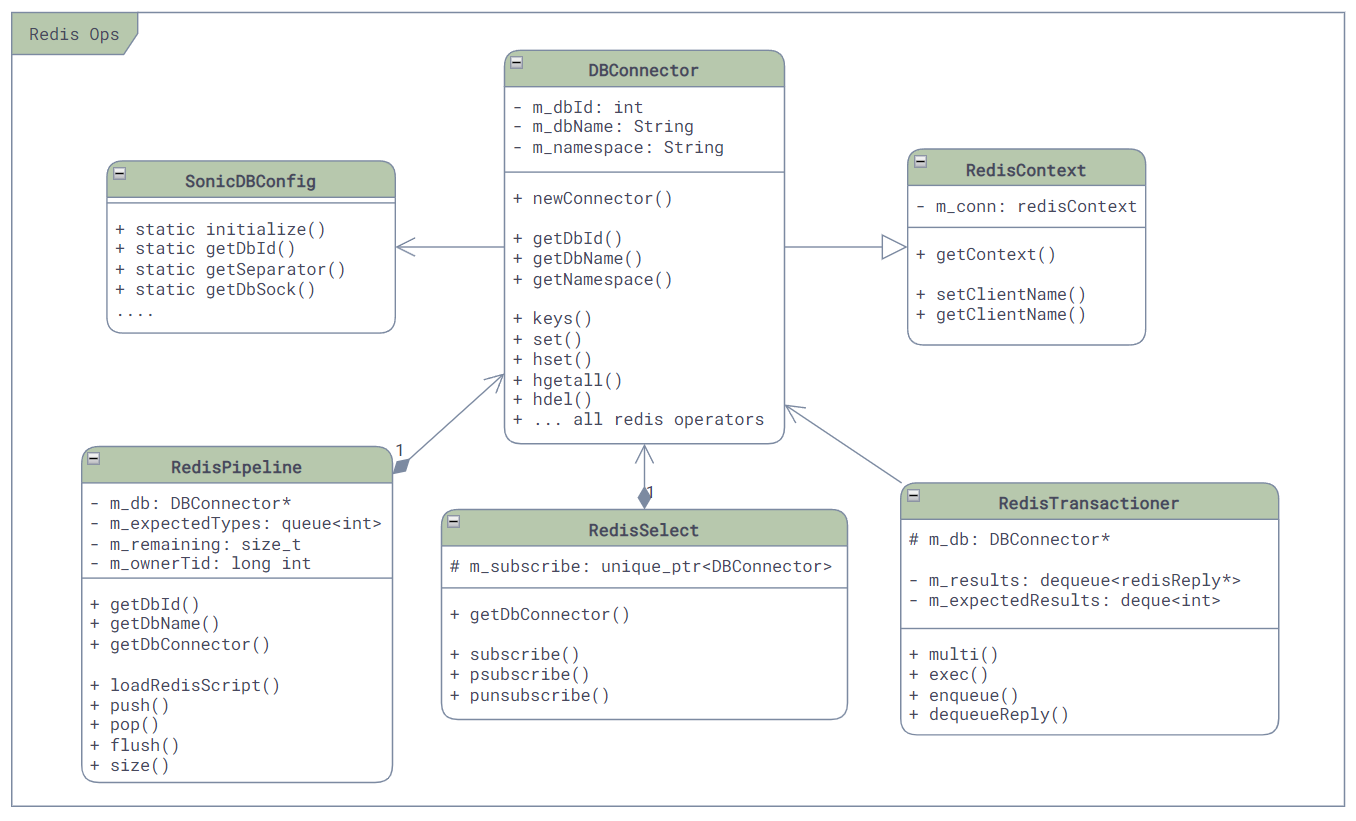
## **2. 基于Redis的服务间通信**

除了和内核的通信接口，SONiC还在Redis的基础上提供了一组PubSub的服务间通信的接口，我们这里就自底向上的来看看这些接口的实现吧！

（虽然SONiC的repo中并没有将以下的代码放在不同的目录下，但是为了方便建立一个清晰的模型，这里我还是会将他们按照层次分开来描述。）

### **2.1. Redis数据库操作层**

第一层，也是最底层，是Redis的数据库操作层，封装了各种基本命令，比如，DB的连接，命令的执行，事件通知的回调接口等等。具体的类图如下：



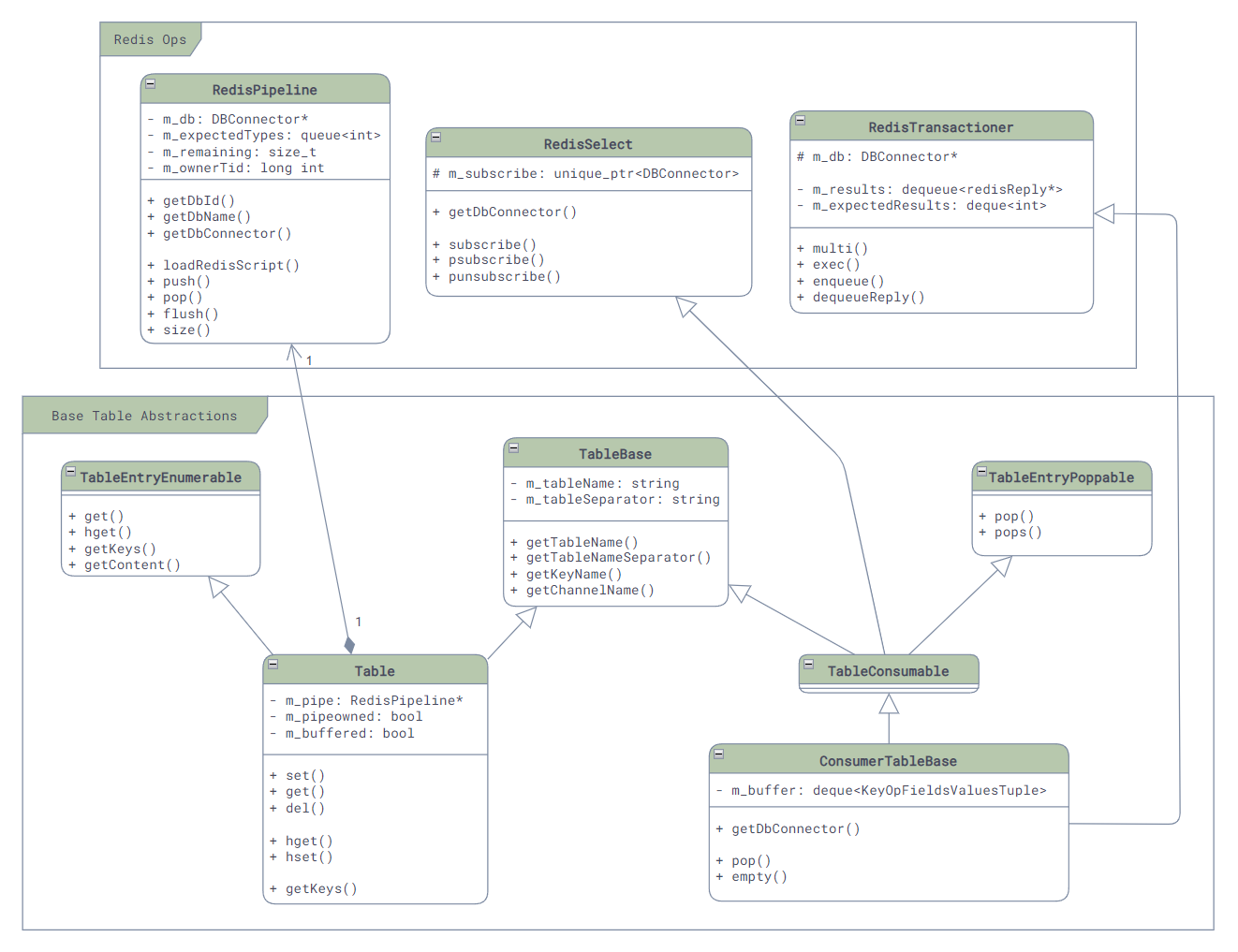
其中：

* **[RedisContext](https://github.com/sonic-net/sonic-swss-common/blob/master/common/dbconnector.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：封装并保持着与Redis的连接，当其销毁时会将其连接关闭。
* **[DBConnector](https://github.com/sonic-net/sonic-swss-common/blob/master/common/dbconnector.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：封装了所有的底层使用到的Redis的命令，比如SET、GET、DEL等等。
* **[RedisTransactioner](https://github.com/sonic-net/sonic-swss-common/blob/master/common/redistran.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：封装了Redis的事务操作，用于在一个事务中执行多个命令，比如MULTI、EXEC等等。
* **[RedisPipeline](https://github.com/sonic-net/sonic-swss-common/blob/master/common/redispipeline.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：封装了hiredis的redisAppendFormattedCommand API，提供了一个类似队列的异步的执行Redis命令的接口（虽然大部分使用方法依然是同步的）。它也是少有的对SCRIPT LOAD命令进行了封装的类，用于在Redis中加载Lua脚本实现存储过程。SONiC中绝大部分需要执行Lua脚本的类，都会使用这个类来进行加载和调用。
* **[RedisSelect](https://github.com/sonic-net/sonic-swss-common/blob/master/common/redisselect.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：它实现了Selectable的接口，用来支持基于epoll的事件通知机制（Event Polling）。主要是在我们收到了Redis的回复，用来触发epoll进行回调（我们最后会更详细的介绍）。
* **[SonicDBConfig](https://github.com/sonic-net/sonic-swss-common/blob/master/common/dbconnector.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：这个类是一个“静态类”，它主要实现了SONiC DB的配置文件的读取和解析。其他的数据库操作类，如果需要任何的配置信息，都会通过这个类来获取。

### **2.2. 表（Table）抽象层**

在Redis数据库操作层之上，便是SONiC自己利用Redis中间的Key建立的表（Table）的抽象了，因为每一个Redis的Key的格式都是<table-name><separator><key-name>，所以SONiC在访问数据库时需要对其进行一次转换（没有印象的小伙伴可以移步[我之前的博客了解更多的信息](http://r12f.com/posts/sonic-2-key-components/" \l "%E6%95%B0%E6%8D%AE%E5%BA%93)）。

相关类的主要类图如下：



其中关键的类有三个：

* **[TableBase](https://github.com/sonic-net/sonic-swss-common/blob/master/common/table.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：这个类是所有表的基类，它主要封装了表的基本信息，如表的名字，Redis Key的打包，每个表发生修改时用于通信的Channel的名字，等等。
* **[Table](https://github.com/sonic-net/sonic-swss-common/blob/master/common/table.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：这个类就是对于每个表增删改查的封装了，里面包含了表的名称和分隔符，这样就可以在调用时构造最终的key了。
* **[ConsumerTableBase](https://github.com/sonic-net/sonic-swss-common/blob/master/common/consumertablebase.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)**：这个类是各种SubscriptionTable的基类，里面主要是封装了一个简单的队列和其pop操作（对，只有pop，没有push），用来给上层调用。

### **2.3. 通信层**

在表抽象层之上，便是SONiC的通信层了，由于需求的不同，这一层中提供了四种不同的PubSub的封装，用于服务间的通信。

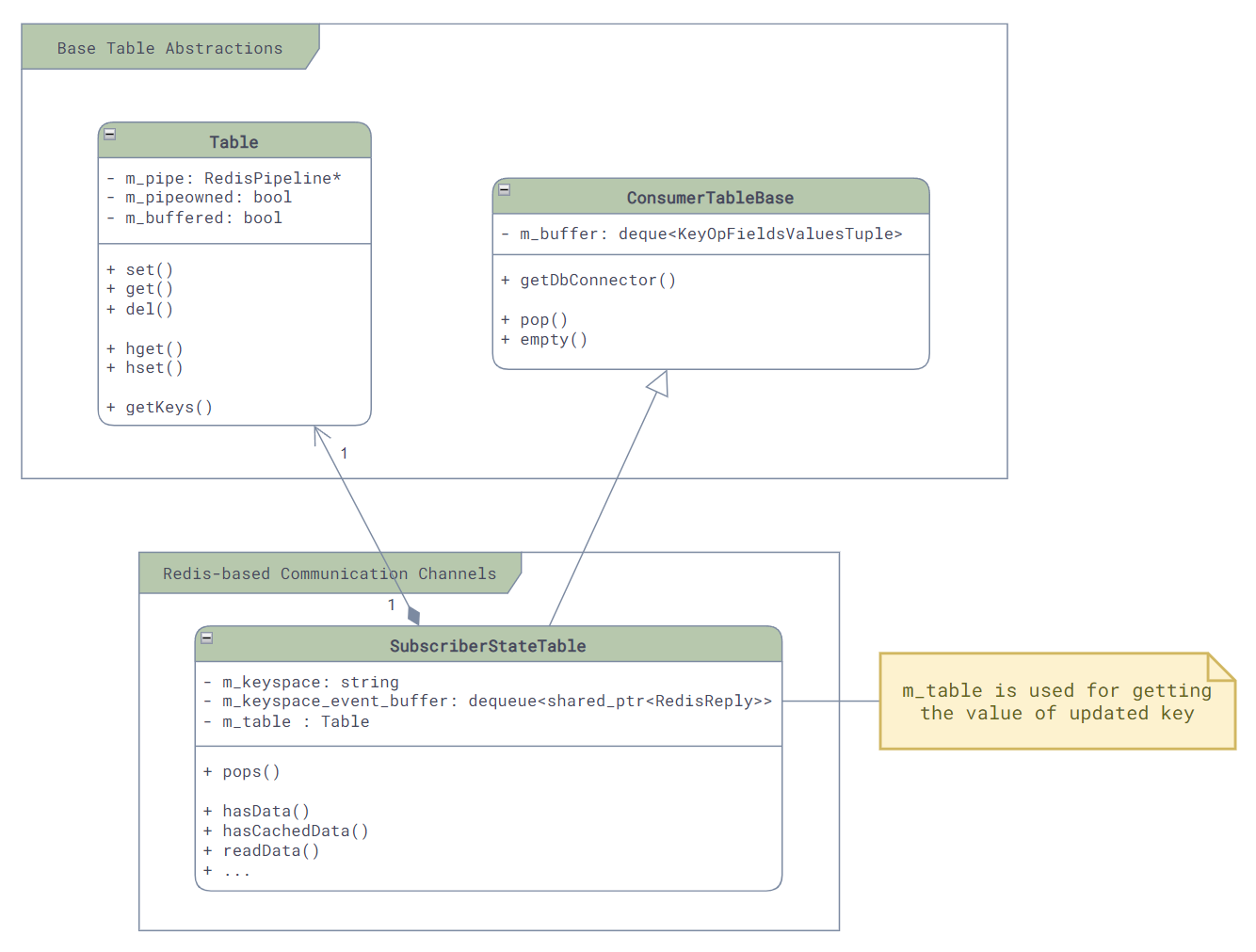
#### **2.3.1. SubscriberStateTable**

最直接的就是[SubscriberStateTable](https://github.com/sonic-net/sonic-swss-common/blob/master/common/subscriberstatetable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)了。

它的原理是利用Redis数据库中自带的keyspace消息通知机制 [[6]](https://redis.io/docs/manual/keyspace-notifications/" \t "http://r12f.com/posts/sonic-4-communication/_blank) —— 当数据库中的任何一个key对应的值发生了变化，就会触发Redis发送两个keyspace的事件通知，一个是\_\_keyspace@<db-id>\_\_:<key>下的<op>事件，一个是\_\_keyspace@<db-id>\_\_:<op>下的<key>>事件，比如，在数据库0中删除了一个key，那么就会触发两个事件通知：

|  |
| --- |
| PUBLISH \_\_keyspace@0\_\_:foo del PUBLISH \_\_keyevent@0\_\_:del foo |

而SubscriberStateTable就是监听了第一个事件通知，然后调用相应的回调函数。和其直接相关的主要的类的类图如下，这里可以看到它继承了ConsumerTableBase，因为它是Redis的消息的Consumer：



在初始化时，我们可以看到它是如何订阅Redis的事件通知的：

|  |
| --- |
| *// File: sonic-swss-common - common/subscriberstatetable.cpp* SubscriberStateTable::SubscriberStateTable(DBConnector \*db, const string &tableName, int popBatchSize, int pri)  : ConsumerTableBase(db, tableName, popBatchSize, pri), m\_table(db, tableName) {  m\_keyspace = "\_\_keyspace@";  m\_keyspace += to\_string(db->getDbId()) + "\_\_:" + tableName + m\_table.getTableNameSeparator() + "\*";  psubscribe(m\_db, m\_keyspace);  *// ...* |

其事件接收和分发主要由两个函数负责：

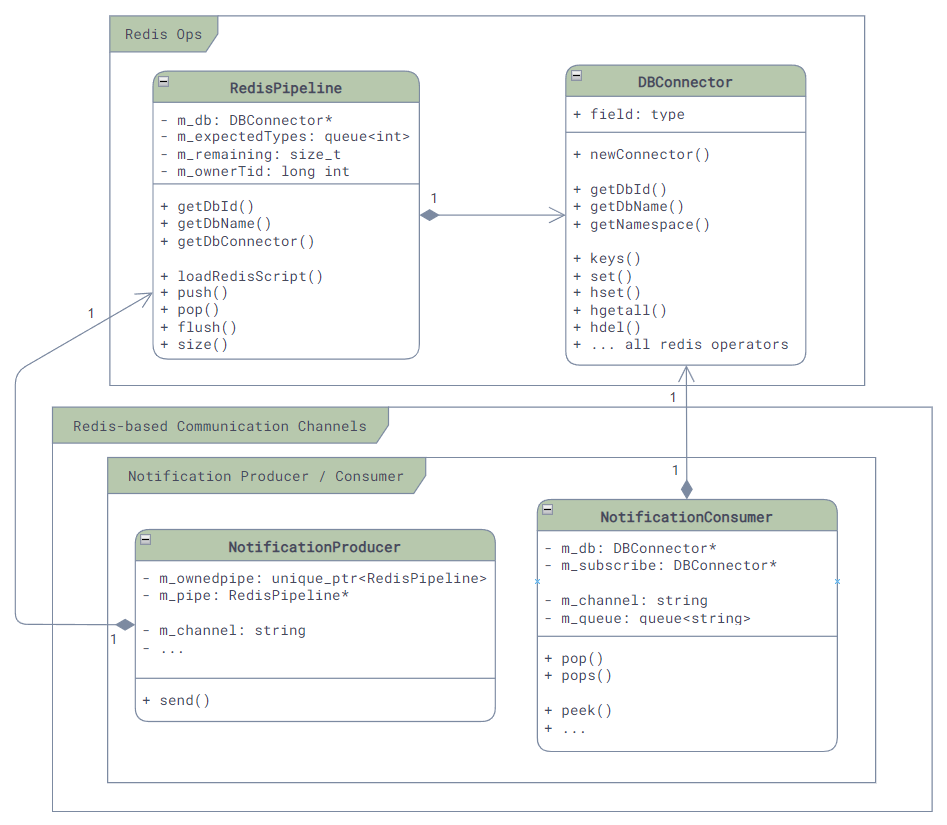
* readData()负责将redis中待读取的事件读取出来，并放入ConsumerTableBase中的队列中
* pops()：负责将队列中的原始事件取出来，并且进行解析，然后通过函数参数传递给调用方

|  |
| --- |
| *// File: sonic-swss-common - common/subscriberstatetable.cpp* uint64\_t SubscriberStateTable::readData() {  *// ...*  reply = nullptr;  int status;  do {  status = redisGetReplyFromReader(m\_subscribe->getContext(), reinterpret\_cast<void\*\*>(&reply));  if(reply != nullptr && status == REDIS\_OK) {  m\_keyspace\_event\_buffer.emplace\_back(make\_shared<RedisReply>(reply));  }  } while(reply != nullptr && status == REDIS\_OK);  *// ...*  return 0; } void SubscriberStateTable::pops(deque<KeyOpFieldsValuesTuple> &vkco, const string& */\*prefix\*/*) {  vkco.clear();  *// ...*  *// Pop from m\_keyspace\_event\_buffer, which is filled by readData()*  while (auto event = popEventBuffer()) {  KeyOpFieldsValuesTuple kco;  *// Parsing here ...*  vkco.push\_back(kco);  }  m\_keyspace\_event\_buffer.clear(); } |

#### **2.3.2. NotificationProducer / NotificationConsumer**

既然说到消息通信，我们很容易就会联想到消息队列，而这就是我们的第二种通信方式 —— [NotificationProducer](https://github.com/sonic-net/sonic-swss-common/blob/master/common/notificationproducer.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)和[NotificationConsumer](https://github.com/sonic-net/sonic-swss-common/blob/master/common/notificationconsumer.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)。

这种通信方式通过Redis的自带的PubSub来实现，主要是对PUBLISH和SUBSCRIBE命令的包装，很有限的应用在最简单的通知型的场景中，比如orchagent中的timeout check, restart check之类，非传递用户配置和数据的场景：



这种通信模式下，消息的发送方Producer，主要会做两件事情：一是将消息打包成JSON格式，二是调用Redis的PUBLISH命令将消息发送出去。而且由于PUBLISH命令只能携带一个消息，所以请求中的op和data字段会被放在values的最前面，然后再调用buildJson函数将其打包成一个JSON数组的格式：

|  |
| --- |
| int64\_t swss::NotificationProducer::send(const std::string &op, const std::string &data, std::vector<FieldValueTuple> &values) {  *// Pack the op and data into values array, then pack everything into a JSON string as the message*  FieldValueTuple opdata(op, data);  values.insert(values.begin(), opdata);  std::string msg = JSon::buildJson(values);  values.erase(values.begin());  *// Publish message to Redis channel*  RedisCommand command;  command.format("PUBLISH %s %s", m\_channel.c\_str(), msg.c\_str());  *// ...*  RedisReply reply = m\_pipe->push(command);  reply.checkReplyType(REDIS\_REPLY\_INTEGER);  return reply.getReply<long long int>(); } |

接收方则是利用SUBSCRIBE命令来接收所有的通知：

|  |
| --- |
| void swss::NotificationConsumer::subscribe() {  *// ...*  m\_subscribe = new DBConnector(m\_db->getDbId(),  m\_db->getContext()->unix\_sock.path,  NOTIFICATION\_SUBSCRIBE\_TIMEOUT);  *// ...*  *// Subscribe to Redis channel*  std::string s = "SUBSCRIBE " + m\_channel;  RedisReply r(m\_subscribe, s, REDIS\_REPLY\_ARRAY); } |

#### **2.3.3. ProducerTable / ConsumerTable**

我们可以看到NotificationProducer/Consumer实现简单粗暴，但是由于API的限制 [[8]](https://redis.io/docs/reference/clients/" \t "http://r12f.com/posts/sonic-4-communication/_blank)，它并不适合用来传递数据，所以，SONiC中提供了一种和它非常接近的另外一种基于消息队列的通信机制 —— [ProducerTable](https://github.com/sonic-net/sonic-swss-common/blob/master/common/producertable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)和[ConsumerTable](https://github.com/sonic-net/sonic-swss-common/blob/master/common/consumertable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)。

这种通信方式通过Redis的List来实现，和Notification不同的地方在于，发布给Channel中的消息非常的简单（单字符"G"），所有的数据都存储在List中，从而解决了Notification中消息大小限制的问题。在SONiC中，它主要用在FlexCounter，syncd服务和ASIC\_DB中：

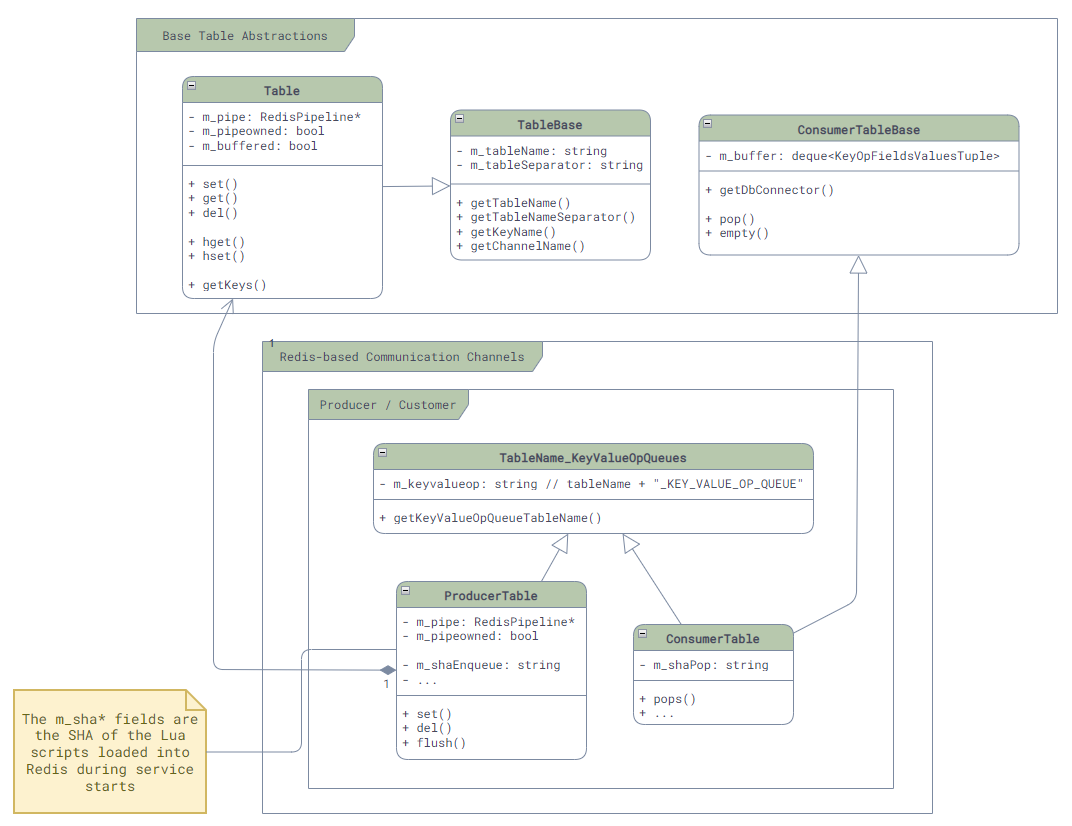
****消息格式****：每条消息都是一个（Key, FieldValuePairs, Op）的三元组，如果用JSON来表达这个消息，那么它的格式如下：（这里的Key是Table中数据的Key，被操作的数据是[Hash](https://redis.io/docs/data-types/hashes/" \t "http://r12f.com/posts/sonic-4-communication/_blank)，所以Field就是Hash中的Field，Value就是Hash中的Value了，也就是说一个消息可以对很多个Field进行操作）

|  |
| --- |
| [ "Key", "[\"Field1\", \"Value1\", \"Field2", \"Value2\", ...]", "Op" ] |

* ****Enqueue****：ProducerTable通过Lua脚本将消息三元组原子的写入消息队列中（Key = <table-name>\_KEY\_VALUE\_OP\_QUEUE，并且发布更新通知到特定的Channel（Key = <table-name>\_CHANNEL）中。
* ****Pop****：ConsumerTable也通过Lua脚本从消息队列中原子的读取消息三元组，并****在读取过程中****将其中请求的改动真正的写入到数据库中。

****注意****：Redis中Lua脚本和MULTI/EXEC的原子性和通常说的数据库ACID中的原子性（Atomicity）不同，Redis中的原子性其实更接近于ACID中的隔离性（Isolation），他保证Lua脚本中所有的命令在执行的时候不会有其他的命令执行，但是并不保证Lua脚本中的所有命令都会执行成功，比如，如果Lua脚本中的第二个命令执行失败了，那么第一个命令依然会被提交，只是后面的命令就不会继续执行了。更多的细节可以参考Redis官方文档 [[4]](https://developer.redis.com/develop/java/spring/rate-limiting/fixed-window/reactive-lua/" \t "http://r12f.com/posts/sonic-4-communication/_blank) [[5]](https://redis.io/docs/manual/transactions/" \t "http://r12f.com/posts/sonic-4-communication/_blank)。

其主要类图如下，这里我们可以看到在ProducerTable中的m\_shaEnqueue和ConsumerTable中的m\_shaPop，它们就是上面我们提到的这两个Lua脚本在加载时获得的SHA了，而之后我们就可以使用Redis的EVALSHA命令对他们进行原子的调用了：



ProducerTable的核心逻辑如下，我们可以看到对Values的JSON打包，和使用EVALSHA来进行Lua脚本的调用：

|  |
| --- |
| *// File: sonic-swss-common - common/producertable.cpp* ProducerTable::ProducerTable(RedisPipeline \*pipeline, const string &tableName, bool buffered)  *// ...* {  string luaEnque =  "redis.call('LPUSH', KEYS[1], ARGV[1], ARGV[2], ARGV[3]);"  "redis.call('PUBLISH', KEYS[2], ARGV[4]);";  m\_shaEnque = m\_pipe->loadRedisScript(luaEnque); } void ProducerTable::set(const string &key, const vector<FieldValueTuple> &values, const string &op, const string &prefix) {  enqueueDbChange(key, JSon::buildJson(values), "S" + op, prefix); } void ProducerTable::del(const string &key, const string &op, const string &prefix) {  enqueueDbChange(key, "{}", "D" + op, prefix); } void ProducerTable::enqueueDbChange(const string &key, const string &value, const string &op, const string& */\* prefix \*/*) {  RedisCommand command;  command.format(  "EVALSHA %s 2 %s %s %s %s %s %s",  m\_shaEnque.c\_str(),  getKeyValueOpQueueTableName().c\_str(),  getChannelName(m\_pipe->getDbId()).c\_str(),  key.c\_str(),  value.c\_str(),  op.c\_str(),  "G");  m\_pipe->push(command, REDIS\_REPLY\_NIL); } |

而另一侧的ConsumerTable就稍稍复杂一点，因为其支持的op类型很多，所以逻辑都写在了一个单独的文件中（common/consumer\_table\_pops.lua），我们这里就不贴代码了，有兴趣的同学可以自己去看看。

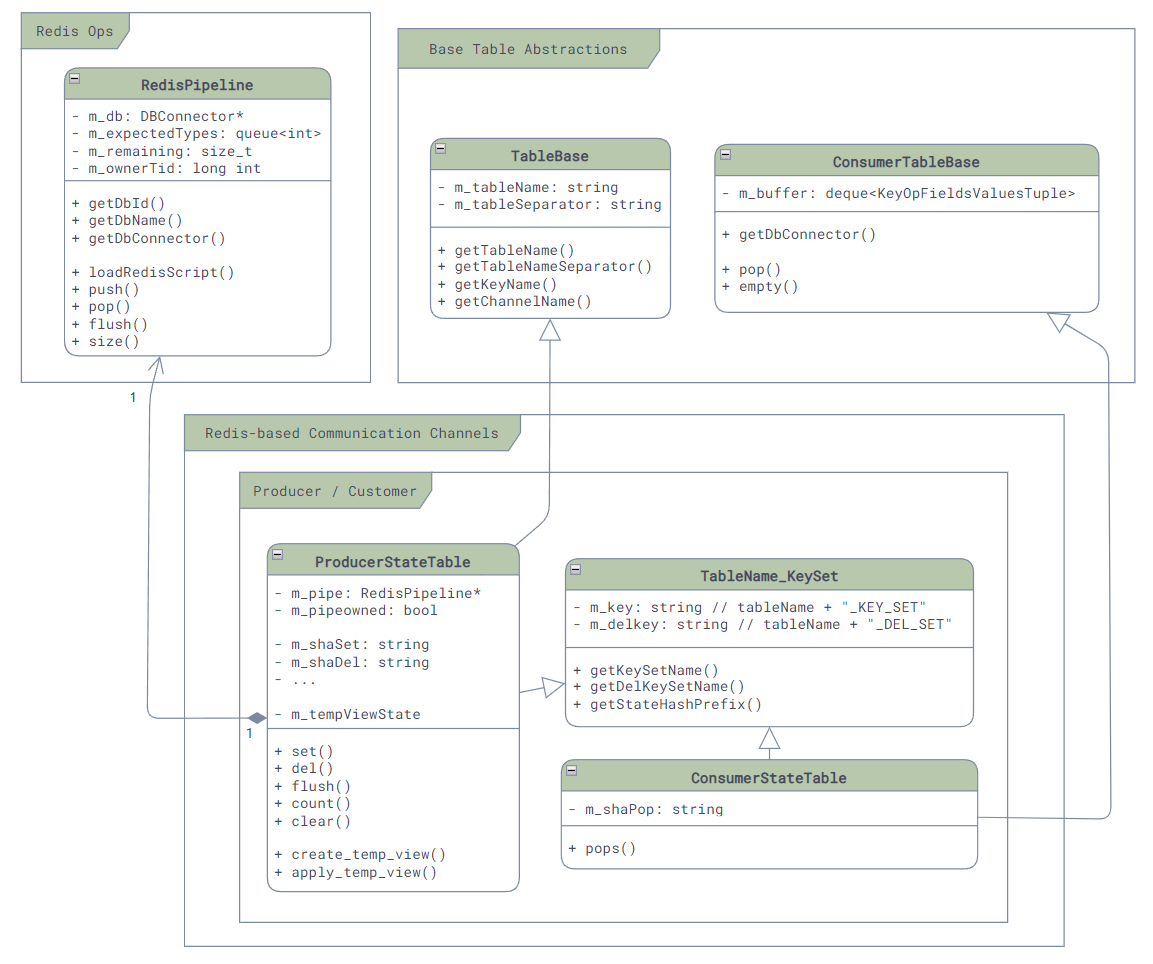
|  |
| --- |
| *// File: sonic-swss-common - common/consumertable.cpp* ConsumerTable::ConsumerTable(DBConnector \*db, const string &tableName, int popBatchSize, int pri)  : ConsumerTableBase(db, tableName, popBatchSize, pri)  , TableName\_KeyValueOpQueues(tableName)  , m\_modifyRedis(true) {  std::string luaScript = loadLuaScript("consumer\_table\_pops.lua");  m\_shaPop = loadRedisScript(db, luaScript);  *// ...* } void ConsumerTable::pops(deque<KeyOpFieldsValuesTuple> &vkco, const string &prefix) {  *// Note that here we are processing the messages in bulk with POP\_BATCH\_SIZE!*  RedisCommand command;  command.format(  "EVALSHA %s 2 %s %s %d %d",  m\_shaPop.c\_str(),  getKeyValueOpQueueTableName().c\_str(),  (prefix+getTableName()).c\_str(),  POP\_BATCH\_SIZE,  RedisReply r(m\_db, command, REDIS\_REPLY\_ARRAY);  vkco.clear();  *// Parse and pack the messages in bulk*  *// ...* } |

#### **2.3.4. ProducerStateTable / ConsumerStateTable**

Producer/ConsumerTable虽然直观，而且保序，但是它一个消息只能处理一个Key，并且还需要JSON的序列化，然而很多时候我们并用不到保序的功能，反而更需要更大的吞吐量，所以为了优化性能，SONiC就引入了第四种通信方式，也是最常用的通信方式：[ProducerStateTable](https://github.com/sonic-net/sonic-swss-common/blob/master/common/producerstatetable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)和[ConsumerStateTable](https://github.com/sonic-net/sonic-swss-common/blob/master/common/consumertatetable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)。

与ProducerTable不同，ProducerStateTable使用Hash的方式来存储消息，而不是List。这样虽然不能保证消息的顺序，但是却可以很好的提升性能！首先，我们省下了JSON的序列化的开销，其次，对于同一个Key下的相同的Field如果被变更多次，那么只需要保留最后一次的变更，这样就将关于这个Key的所有变更消息就合并成了一条，减少了很多不必要的消息处理。

Producer/ConsumerStateTable的底层实现相比于Producer/ConsumerTable也更加复杂一些。其相关联的类的主要类图如下，这里我们依然可以看到它的实现是通过EVALSHA调用Lua脚本来实现的，m\_shaSet和m\_shaDel就是用来存放修改和发送消息的，而另一边m\_shaPop就是用来获取消息的：



在传递消息时：

* 首先，每个消息会被存放成两个部分：一个是KEY\_SET，用来保存当前有哪些Key发生了修改，它以Set的形式存放在<table-name\_KEY\_SET>的key下，另一个是所有被修改的Key的内容，它以Hash的形式存放在\_<redis-key-name>的key下。
* 然后，消息存放之后Producer如果发现是新的Key，那么就是调用PUBLISH命令，来通知<table-name>\_CHANNEL@<db-id>Channel，有新的Key出现了。

|  |
| --- |
| *// File: sonic-swss-common - common/producerstatetable.cpp* ProducerStateTable::ProducerStateTable(RedisPipeline \*pipeline, const string &tableName, bool buffered)  : TableBase(tableName, SonicDBConfig::getSeparator(pipeline->getDBConnector()))  , TableName\_KeySet(tableName)  *// ...* {  string luaSet =  "local added = redis.call('SADD', KEYS[2], ARGV[2])\n"  "for i = 0, #KEYS - 3 do\n"  " redis.call('HSET', KEYS[3 + i], ARGV[3 + i \* 2], ARGV[4 + i \* 2])\n"  "end\n"  " if added > 0 then \n"  " redis.call('PUBLISH', KEYS[1], ARGV[1])\n"  "end\n";  m\_shaSet = m\_pipe->loadRedisScript(luaSet); |

* 最后，Consumer会通过SUBSCRIBE命令来订阅<table-name>\_CHANNEL@<db-id>Channel，一旦有新的消息到来，就会使用Lua脚本调用HGETALL命令来获取所有的Key，并将其中的值读取出来并真正的写入到数据库中去。

|  |
| --- |
| ConsumerStateTable::ConsumerStateTable(DBConnector \*db, const std::string &tableName, int popBatchSize, int pri)  : ConsumerTableBase(db, tableName, popBatchSize, pri)  , TableName\_KeySet(tableName) {  std::string luaScript = loadLuaScript("consumer\_state\_table\_pops.lua");  m\_shaPop = loadRedisScript(db, luaScript);  *// ...*  subscribe(m\_db, getChannelName(m\_db->getDbId()));  *// ...* |

为了方便理解，我们这里举一个例子：启用Port Ethernet0：

首先，我们在命令行下调用config interface startup Ethernet0来启用Ethernet0，这会导致portmgrd通过ProducerStateTable向APP\_DB发送状态更新消息，如下：

|  |
| --- |
| EVALSHA "<hash-of-set-lua>" "6" "PORT\_TABLE\_CHANNEL@0" "PORT\_TABLE\_KEY\_SET"   "\_PORT\_TABLE:Ethernet0" "\_PORT\_TABLE:Ethernet0" "\_PORT\_TABLE:Ethernet0" "\_PORT\_TABLE:Ethernet0" "G"  "Ethernet0" "alias" "Ethernet5/1" "index" "5" "lanes" "9,10,11,12" "speed" "40000" |

这个命令会在其中调用如下的命令来创建和发布消息：

|  |
| --- |
| SADD "PORT\_TABLE\_KEY\_SET" "\_PORT\_TABLE:Ethernet0" HSET "\_PORT\_TABLE:Ethernet0" "alias" "Ethernet5/1" HSET "\_PORT\_TABLE:Ethernet0" "index" "5" HSET "\_PORT\_TABLE:Ethernet0" "lanes" "9,10,11,12" HSET "\_PORT\_TABLE:Ethernet0" "speed" "40000" PUBLISH "PORT\_TABLE\_CHANNEL@0" "\_PORT\_TABLE:Ethernet0" |

所以最终这个消息会在APPL\_DB中被存放成如下的形式：

|  |
| --- |
| PORT\_TABLE\_KEY\_SET:  \_PORT\_TABLE:Ethernet0 \_PORT\_TABLE:Ethernet0:  alias: Ethernet5/1  index: 5  lanes: 9,10,11,12  speed: 40000 |

当ConsumerStateTable收到消息后，也会调用EVALSHA命令来执行Lua脚本，如下：

|  |
| --- |
| EVALSHA "<hash-of-pop-lua>" "3" "PORT\_TABLE\_KEY\_SET" "PORT\_TABLE:" "PORT\_TABLE\_DEL\_SET" "8192" "\_" |

和Producer类似，这个脚本会执行如下命令，将PORT\_TABLE\_KEY\_SET中的key，也就是\_PORT\_TABLE:Ethernet0读取出来，然后再将其对应的Hash读取出来，并更新到PORT\_TABLE:Ethernet0去，同时将\_PORT\_TABLE:Ethernet0从数据库和PORT\_TABLE\_KEY\_SET中删除。

|  |
| --- |
| SPOP "PORT\_TABLE\_KEY\_SET" "\_PORT\_TABLE:Ethernet0" HGETALL "\_PORT\_TABLE:Ethernet0" HSET "PORT\_TABLE:Ethernet0" "alias" "Ethernet5/1" HSET "PORT\_TABLE:Ethernet0" "index" "5" HSET "PORT\_TABLE:Ethernet0" "lanes" "9,10,11,12" HSET "PORT\_TABLE:Ethernet0" "speed" "40000" DEL "\_PORT\_TABLE:Ethernet0" |

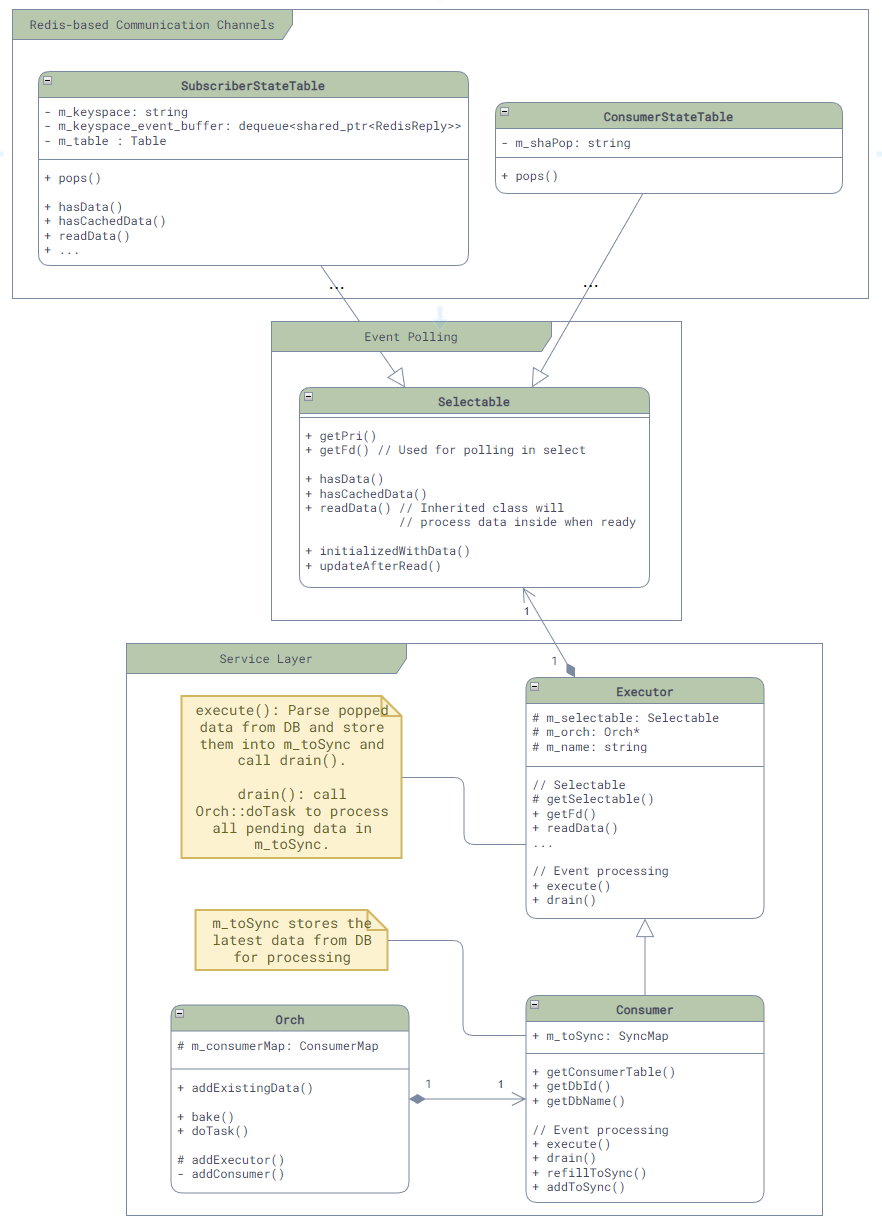
到这里，数据的更新才算是完成了。

### **2.4. 服务层：Orch**

最后，为了方便各个服务使用，SONiC还在通信层上进行了更进一步的封装，为各个服务提供了一个基类：[Orch](https://github.com/sonic-net/sonic-swss/blob/master/src/orchagent/orch.hcommon/consumertatetable.h" \t "http://r12f.com/posts/sonic-4-communication/_blank)。

由于有了上面这些封装，Orch中关于消息通信的封装就相对简单了，主要的类图如下：

注意：由于这一层是服务层，所以其代码是在sonic-swss的仓库中，而不是sonic-swss。这个类中除了消息通信的封装以外，还提供了很多和服务实现相关的公共函数，比如，日志文件等等。



可以看到，Orch主要是封装了SubscriberStateTable和ConsumerStateTable来简化和统一消息的订阅，核心代码非常简单，就是根据不同的数据库创建不同的Consumer，如下：

|  |
| --- |
| void Orch::addConsumer(DBConnector \*db, string tableName, int pri) {  if (db->getDbId() == CONFIG\_DB || db->getDbId() == STATE\_DB || db->getDbId() == CHASSIS\_APP\_DB) {  addExecutor(new Consumer(new SubscriberStateTable(db, tableName, TableConsumable::DEFAULT\_POP\_BATCH\_SIZE, pri), this, tableName));  } else {  addExecutor(new Consumer(new ConsumerStateTable(db, tableName, gBatchSize, pri), this, tableName));  } } |

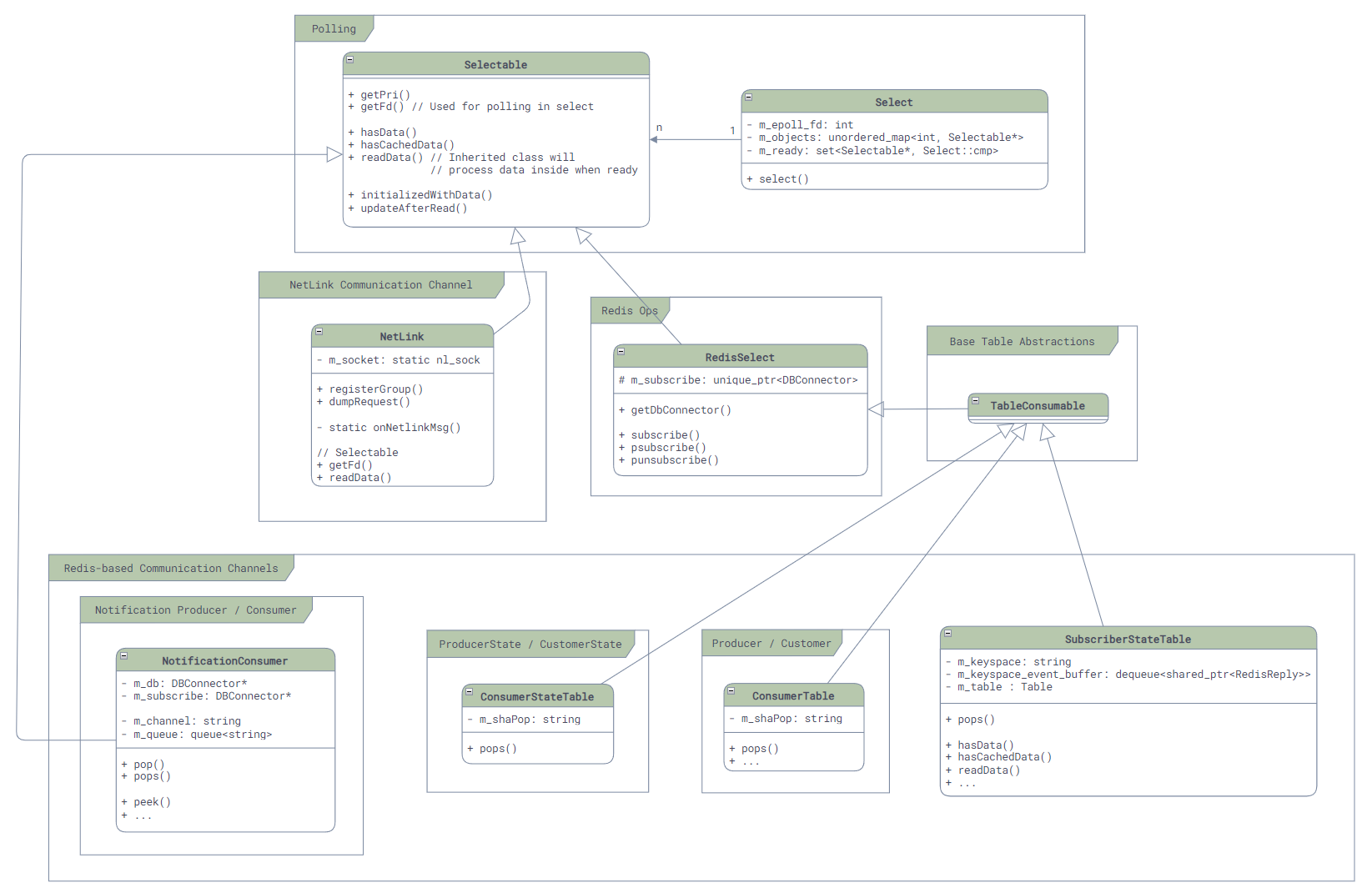
## **3. 基于epoll的事件分发机制**

好了，我们现在已经了解了所有关于消息通信机制的上层封装了，关于消息通信，还有一个很重要的就是消息的检测和分发，我们这里就也来看看吧！

和很多的Linux服务一样，SONiC底层使用了epoll作为事件分发机制：

* 所有需要支持事件分发的类都需要继承Selectable类，并实现两个最核心的函数：int getFd();（用于返回epoll能用来监听事件的fd）和uint64\_t readData()（用于在监听到事件到来之后进行读取）。而对于一般服务而言，这个fd就是redis通信使用的fd，所以getFd()函数的调用，都会被最终转发到Redis的库中。
* 所有需要参与事件分发的对象，都需要注册到Select类中，这个类会将所有的Selectable对象的fd注册到epoll中，并在事件到来时调用Selectable的readData()函数。

其类图如下：



在Select类中，我们可以很容易的找到其最核心的代码，实现也非常的简单：

|  |
| --- |
| int Select::poll\_descriptors(Selectable \*\*c, unsigned int timeout, bool interrupt\_on\_signal = false) {  int sz\_selectables = static\_cast<int>(m\_objects.size());  std::vector<struct epoll\_event> events(sz\_selectables);  int ret;  while(true) {  ret = ::epoll\_wait(m\_epoll\_fd, events.data(), sz\_selectables, timeout);  *// ...*  }  *// ...*  for (int i = 0; i < ret; ++i)  {  int fd = events[i].data.fd;  Selectable\* sel = m\_objects[fd];   sel->readData();  *// error handling here ...*  m\_ready.insert(sel);  }  while (!m\_ready.empty())  {  auto sel = \*m\_ready.begin();  m\_ready.erase(sel);  *// After update callback ...*  return Select::OBJECT;  }  return Select::TIMEOUT; } |

然而，问题来了…… 回调呢？我们上面提过，readData()只是把消息读出来放在一个待处理队列中，并不会真正的处理消息，真正的消息处理需要调用pops()函数，将消息拿出来处理，所以什么地方会调用每一个上层封装的消息处理呢？

这里我们还是找到我们的老朋友portmgrd的main函数，从下面简化的代码中，我们可以看到和一般的Event Loop实现不同，SONiC中，最后的事件处理不是通过回调来实现的，而是需要最外层的Event Loop来主动调用完成：

|  |
| --- |
| int main(int argc, char \*\*argv) {  *// ...*  *// Create PortMgr, which implements Orch interface.*  PortMgr portmgr(&cfgDb, &appDb, &stateDb, cfg\_port\_tables);  vector<Orch \*> cfgOrchList = {&portmgr};  *// Create Select object for event loop and add PortMgr to it.*  swss::Select s;  for (Orch \*o : cfgOrchList) {  s.addSelectables(o->getSelectables());  }  *// Event loop*  while (true)  {  Selectable \*sel;  int ret;  *// When anyone of the selectables gets signaled, select() will call*  *// into readData() and fetch all events, then return.*  ret = s.select(&sel, SELECT\_TIMEOUT);  *// ...*  *// Then, we call into execute() explicitly to process all events.*  auto \*c = (Executor \*)sel;  c->execute();  }  return -1; } |

## **4. 错误处理**

关于Event Loop我们还有最后一个问题，那就是错误处理，比如，如果Redis的命令执行出错了，连接断开了，故障了等等的情况下，我们的服务会发生什么呢？

从代码上来看，SONiC中的错误处理是非常简单的，就是直接抛出异常（比如，获取命令执行结果的代码，如下），然后在Event Loop中捕获异常，打印日志，接着继续执行。关于异常和错误的种类及其原因，在代码里面并没有看到用于统计和Telemetry的代码，所以监控上说是比较薄弱的。另外还需要考虑数据出错的场景，比如数据库写到一半突然断开导致的脏数据，不过简单的重启相关的\*syncd和\*mgrd服务可能可以解决此类问题，因为启动时会进行全量同步。

|  |
| --- |
| RedisReply::RedisReply(RedisContext \*ctx, const RedisCommand& command) {  int rc = redisAppendFormattedCommand(ctx->getContext(), command.c\_str(), command.length());  if (rc != REDIS\_OK)  {  *// The only reason of error is REDIS\_ERR\_OOM (Out of memory)*  *// ref: <https://github.com/redis/hiredis/blob/master/hiredis.c>*  throw bad\_alloc();  }  rc = redisGetReply(ctx->getContext(), (void\*\*)&m\_reply);  if (rc != REDIS\_OK)  {  throw RedisError("Failed to redisGetReply with " + string(command.c\_str()), ctx->getContext());  }  guard([&]{checkReply();}, command.c\_str()); } |

## **5. 总结**

好了，这就是所有和SONiC中内核和服务间通信相关的所有内容了，之后有时间，我们再来深入的看看一些有特色的工作流或者模块吧！

## **6. 参考资料**

* [1] [SONiC Architecture](https://github.com/sonic-net/SONiC/wiki/Architecture" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [2] [SONiC SWSS Code Repo](https://github.com/sonic-net/sonic-swss" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [3] [SONiC SWSS Common Code Repo](https://github.com/sonic-net/sonic-swss-common" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [4] [Redis - Atomicity with Lua](https://developer.redis.com/develop/java/spring/rate-limiting/fixed-window/reactive-lua/" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [5] [Redis - Transactions](https://redis.io/docs/manual/transactions/" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [6] [Redis - Keyspace Notifications](https://redis.io/docs/manual/keyspace-notifications/" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [7] [Redis - Hashes](https://redis.io/docs/data-types/hashes/" \t "http://r12f.com/posts/sonic-4-communication/_blank)
* [8] [Redis - Client Handling](https://redis.io/docs/reference/clients/" \t "http://r12f.com/posts/sonic-4-communication/_blank)