1. 增加接口

在conf\serviceConfig\vmInstance.xml增加配置

<message>

<name>org.zstack.header.vm.APIGetVmInstanceModifyTimesMsg</name>

</message>

* 1. 拦截器

所有消息先经过拦截器，

全局拦截器：继承GlobalApiMessageInterceptor

类级别拦截器，配置在vmInstance.xml顶端

消息拦截器：配置在<message>中

**ApiMediatorImpl**会处理大多数消息

* 1. 子service

<**message**>  
 <**name**>org.zstack.header.vm.APIQueryVmNicMsg</**name**>  
 <**serviceId**>query</**serviceId**>  
</**message**>

QueryService由serach包处理

1. 统一入口：公共Controller接收消息

云总线CloudBusImpl2.java分发消息（serv.handleMessage(msg)）

1. 按照serviceId丢进对应service类

VmInstanceManagerImpl

1. Service类分发消息

@Override  
@MessageSafe  
**public void** handleMessage(Message msg) {  
 **if** (msg **instanceof** APIMessage) {  
 handleApiMessage((APIMessage) msg);  
 } **else** {  
 handleLocalMessage(msg);  
 }  
}

1. Service类处理消息并返回信息

APIGetVmInstanceModifyTimesReply reply = **new** APIGetVmInstanceModifyTimesReply();

**｛do something｝**

**bus**.reply(msg, reply);//1.所有查询类接口的**同步**响应reply

**bus**.send();//2.系统内部再次调用api消息

APIDeleteVmStaticIpEvent evt = **new** APIDeleteVmStaticIpEvent(msg.getId());  
**new** StaticIpOperator().deleteStaticIpByVmUuidAndL3Uuid(**self**.getUuid(), msg.getL3NetworkUuid());  
**bus**.publish(evt);//3.异步，前端轮寻获取结果

1. dao层逻辑

AO对应数据库表

Eo对应数据库表，EO支持软删除，EO表里要有deleted字段，通过该字段实现软删除

VO可以是视图，也可以是table，不需要做软删除可以直接用vo

zstack API消息

Interceptor

extensions

级联Cascade

tracker

消息 － 所有操作都是消息 action

reply 回复

event 事件回复，可以订阅

API消息权限（权限检查只限API消息 在继承关系上有 APIMessage）

\* @SuppressCredentialCheck 抑制凭证检查, 该注解的消息不检查凭证

\* @InnerCredentialCheck

\* @Action 检查凭证

\* 没有注解的 默认onlyAdmin使用

\* 用户使用权限要policy决定

API消息流转 注意：不只有API消息，其他发往非api.portal队列的消息，其他服务直接处理。

所有API －》api.portal队列，该队列服务ApiMediatorImpl-dispatchMessage－》ApiMessageProcessorImpl-process

通过 ApiMessageProcessorImpl 中 process处理，api消息 首先 apiParamValidation 然后调用该消息的所有 intercept，然后通过bus.route(msg) 路由消息到各服务队列，各服务handle 消息处理，返回处理结果到返回队列。

API消息的Interceptors定义架构为 serviceConfigFolders目录为服务消息下配置文件。id为服务ID，也是要进一步分发到下面的消息队列的id前缀。其中定义了该配置文件中消息的服务ID，intercept 等。

整个拦截器链为：

1. 本配置文件中全局拦截器(通常开头配置ApiMessageInterceptor)

2. 本配置文件中自有拦截器(消息单独指定ApiMessageInterceptor

3. 服务全局拦截器GlobalApiMessageInterceptor.class 通过pluginRgty.getExtensionList(GlobalApiMessageInterceptor.class) 获取。

其中GlobalApiMessageInterceptor.class根据接口getMessageClassToIntercept来判断是拦截全部消息，还是指定消息。拦截位置根据getPosition实现指定分为 InterceptorPosition.SYSTEM，InterceptorPosition.FRONT，InterceptorPosition.END 3个位置。

api消息拦截器的执行顺序为：

1. GlobalApiMessageInterceptor 位置为：InterceptorPosition.SYSTEM

2. GlobalApiMessageInterceptor 位置为：InterceptorPosition.FRONT

3. 本配置文件中全局拦截器＋ 本配置文件中自有拦截器

4. 应用全局拦截器InterceptorPosition.END

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

用户权限策略

身份认证负责提供访问控制（ access control）. 系统中定义了账户（account）, 用户（user）, 角色策略（policy）等概念.

账户（Account）

人们需要创建作为拥有所有他们资源的根身份认证（root identity）的账户, 才能对资源实施操作. 有两种账户类型: 管理员（admin）和普通用户（normal）.

管理员账户拥有不受限的权限, 通常由系统管理管理员拥有. 普通用户账户, 只能对部分功能有操作的权限, 普通用户账户是由平台注册的，从而使他们可以操作他们的资源.

API被分为了管理员权限（admin-only）API和非管理员权限（non-admin）API两类。

管理员账户和普通账户的 账户下都可以创建用户， 管理员账户下的用户通常为我们后台运维用户。

API 权限分配：

\* @SuppressCredentialCheck 抑制凭证检查, 该注解的消息不检查凭证

InnerCredentialCheck

\* @Action 检查凭证

\* 没有注解的 默认只有onlyAdmin账户自己使用，其下用户不能使用

\* 用户使用权限要角色策略 policy决定

public @interface Action {  boolean adminOnly() default false; //只Admin 只有一个账户超级管理员  boolean accountOnly() default false; //只帐户  String category(); // api分类  String[] names() default {}; // {"read”, “update”, “create”, “delete”, }  boolean accountControl() default false; // 暂时可以不用，数据库中配置的 }

用户需要要赋予角色才能拥有该角色的权限。

角色的权限策略声明采用如下结构

public static class PolicyStatement {

private StatementEffect effect; // Allow 或 Deny

private List<String> actions;

}

例如

{

"actions": [

".\*:read",

"instance:APICreateInstanceMsg"

],

"effect": "Allow"

}

[{"effect":"Allow","actions":["identity:APIUpdateUserMsg"]}]

{"actions": [".\*:read","instance:APICreateInstanceMsg"],"effect": "Allow"}

actions\*\*是一个用于匹配一个或多个标识的动作字符串（action strings）组成的列表. API标识是唯一标识一个API的字符串，其格式为:

api\_category:api\_name 动作字符串可以是唯一识别单个API的完整的标识如 instance:APICreateVmInstanceMsg, 也可以是一个匹配很多个API的正则表达式（regular expression）, 例如, instance:.\* 会和所有 instance 类别下的API匹配。

api\_name 也可以是 read, update, create, delete 等 根据Action注解names指定

大多数的API只有一个api\_category:api\_name格式的标识; 但有些API拥有多个标识，因此可以使用正则表达式来匹配这些API.

effect 指示API调用匹配一个操作字符串时对应的决定是什么, 允许（Allow）还是拒绝（Deny）.

灰鸽

特别关注

关于事务主意事项

如果方法中需要事务支持，请加入 @Transactional 如：

@Transactional

private List<HostVO> methodTransactional() {

｝

但要注意保存数据的写法

dbf.updateAndRefresh

dbf.persistAndRefresh

dbf.persist

dbf.remove

等方法 会另外嵌套事务（见@Transactional(propagation = Propagation.REQUIRES\_NEW)）所以 保存多个对象时不要使用。

正确使用的方法如下：

dbf.getEntityManager().persist

dbf.getEntityManager().remove

dbf.getEntityManager().merge

方法操作对象。

举例如下：

@Transactional

private List<HostVO> method1() {

dbf.persist(user1)

dbf.persist(user2)

｝

@Transactional

private List<HostVO> method2() {

dbf.getEntityManager()persist(user1)

dbf.getEntityManager().persist(user2)

｝

method2 在同一个事务中，支持回滚。

method1 其实每个persist都在不通的子事务中无法回滚

灰鸽

特别关注

15:24:46

我们框架中常用查询写法如下：

1.

方法中条件查询优先使用 针对对象的查询这个方法很强大。

查询整个对象

SimpleQuery<UserVO> q = dbf.createQuery(UserVO.class);

List<UserVO> users = q.list();

也可以 查询其中的1个列

SimpleQuery<UserVO> q = dbf.createQuery(UserVO.class);

q.select(UserVO\_.uuid);

List<String> ids = q.listValue();

for (String id : ids) {

}

也可以其中几个列

SimpleQuery<UserVO> q = dbf.createQuery(UserVO.class);

q.select(UserVO\_.id, UserVO\_.name);

List<Tuple> ts = q.listTuple();

for (Tuple t : ts) {

t.get(0)

t.get(1)

}

2.

若不能确定对象，需要复杂关联条件的。

String sql = "select u from UserVO u where u.accountUuid in (select a.uuid from AccountVO a where a.name in (:names))";

TypedQuery<UserVO> q = dbf.getEntityManager().createQuery(sql, AccountVO.class);

q.setParameter("names", names);

List<UserVO> users = q.getResultList();

选出1列

也可以选出部分列

String sql = "select uuid from UserVO";

TypedQuery<String> q = dbf.getEntityManager().createQuery(sql, String.class);

List<String> ts = q.getResultList();

也可以选出部分列

String sql = "select uuid, accountUuid from UserVO";

TypedQuery<Tuple> q = dbf.getEntityManager().createQuery(sql, Tuple.class);

List<Tuple> ts = q.getResultList();

for (Tuple t : ts) {

uuid = t.get(0, String.class);

accountUuid = t.get(1, String.class)

}

3.

sql更新例子,该方法可能用的比较少，我们主要针对对象操纵

String dsql = "delete from SessionVO s where CURRENT\_TIMESTAMP >= s.expiredDate";

Query dq = dbf.getEntityManager().createQuery(dsql);

dq.executeUpdate();

4.

当然 hibernate也支持其它用法，以上为常用的方法。优先顺序为 1 中方式。

5.

特别注意：

查询 中 杜绝使用 ＋拼接 sql 语句参数值的情况，防止注入攻击。

比如 "select uuid from UserVO where uuid='" + id + "'" ; 要坚决杜绝。

要使用 setParameter 注入参数

"select uuid from UserVO where uuid = :id"

q.setParameter("id", id);

StringBuilder sbSql = new StringBuilder();

sbSql.append("select cvr from ClusterVlanRangeVO cvr where cvr.clusterUuid = :clusterUuid ");

sbSql.append(" and (cvr.vlanStart >=:vlanStart and cvr.vlanStart <=:vlanEnd or cvr.vlanEnd >=:vlanStart and cvr.vlanEnd <= :vlanEnd) ");

if(uuid!=null&&!"".equals(uuid)){

sbSql.append(" and cvr.uuid!=:uuid");

}

TypedQuery<ClusterVlanRangeVO> q = dbf.getEntityManager().createQuery(sbSql.toString(), ClusterVlanRangeVO.class);

q.setParameter("clusterUuid", clusterUuid);

q.setParameter("vlanStart", start);

q.setParameter("vlanEnd", end);

if(uuid!=null&&!"".equals(uuid)){

q.setParameter("uuid", uuid);

}

List<ClusterVlanRangeVO> cvr = q.getResultList();

if(cvr!=null&&cvr.size()>0){

return true;

}

return false;