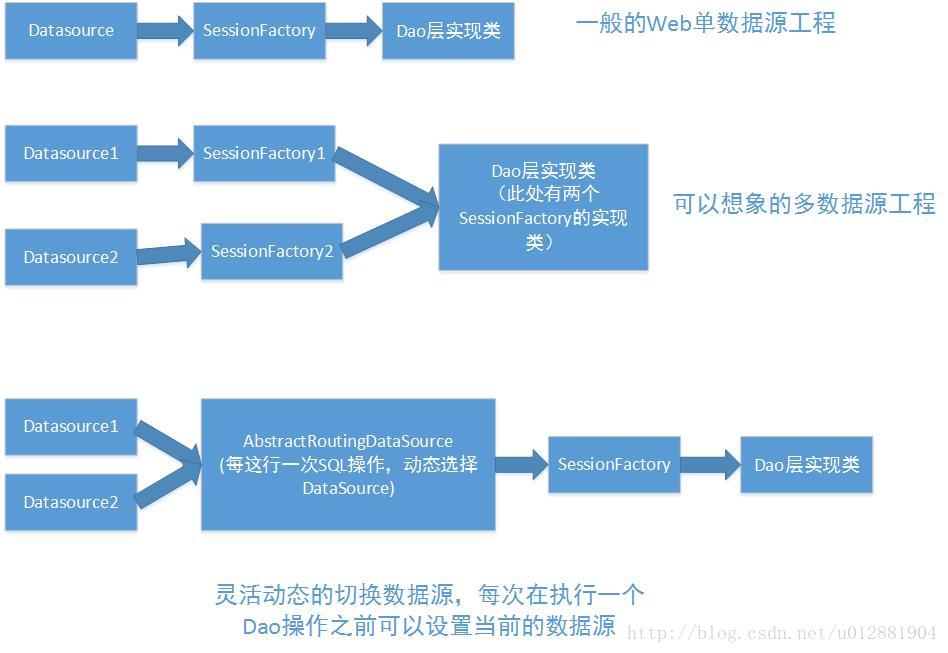
## 一、数据源动态切换

在viid中，是在对接案事件时用到了数据源的动态切换。

对于单数据源、多数据源，看下图：



**\***单数据源的场景：对于大多数的一般业务，这样的配置就可以满足需求

\*多数据源：

1、多数据源多sessionFactory的配置方式，即使用多个DataSource然后创建多个SessionFactory，在使用Dao层的时候通过不同的SessionFactory进行处理。这样的入侵性比较明显，一般的情况都是使用继承HibernateSupportDao进行了封装的处理，如果多个SessionFactory这样处理就比较麻烦

2、使用AbstractRoutingDataSource的实现类，进行灵活的切换，可以通过AOP或者手动编程设置当前的DataSource，不用修改继承了HibernateDaoSupport的实现类。

在VIID中就是编写了AbstractRoutingDataSource的实现类，通过AOP的方式设置当前的DataSource

1. 编写AbstractRoutingDataSource的实现类

**public class** VdsDataSourceController **extends** AbstractRoutingDataSource {  
 @Override  
 **protected** Object determineCurrentLookupKey() {  
 **return** DataSourceAOP.*getDataSource*();  
 }  
}

1. 在配置文件中会根据方法determineCurrentLookupKey返回的值，即DataSourceAOP.*getDataSource*()的值，选择不同的数据源

<**bean id="caseOperation" class="com.hikvision.viid.common.antelope.mvc.oracle.util.VdsDataSourceController"**>  
 <**property name="targetDataSources"**>  
 <**map key-type="java.lang.String"**>  
 <**entry key="1" value-ref="vdsOracleDataSource"**/>  
 <**entry key="0" value-ref="dataSource"**/>  
 </**map**>  
 </**property**></**bean**>

1. 而DataSourceAOP.getDataSource()的值，是通过AOP的方式执行setDataSource方法设置的

<**aop:config**>  
 <**aop:aspect id="dataSourceAspectOracle" ref="dataSourceAOP"** >  
 <**aop:pointcut id="dataAop" expression="execution(\* com.hikvision.viid..service.impl.CaseManagerService.\*(..))"**/>  
 <**aop:before method="setDataSource" pointcut-ref="dataAop"** />  
 </**aop:aspect**>  
</**aop:config**>

@Component(**"dataSourceAOP"**)  
**public class** DataSourceAOP {  
  
 **private static** ThreadLocal<String> *dataSoueceKey* = **new** ThreadLocal<>();  
  
 **private static** String *caseOpration* = ObjectUtil.*objToInteger*(KmsConfig.*getPropConfigValue*(AntelopeConstant.***CASE\_OPERATION***)) > 0 ?  
 KmsConfig.*getPropConfigValue*(AntelopeConstant.***CASE\_OPERATION***) : **"0"**;  
  
 **public static** String getDataSource() {  
 **if**(*dataSoueceKey*.get() == **null**){  
 **return "0"**;  
 }  
 **return** *dataSoueceKey*.get();  
 }  
  
 **public static void** setDataSource() {  
 *dataSoueceKey*.set(*caseOpration*);  
 }  
  
 **public static void** clearDataSource() {  
 *dataSoueceKey*.remove();  
 }  
}

其中的caseOpration读取的是配置文件.properties中的值

1. 这样就可以为SessionFactory配置动态指定的数据源了

<**bean class="org.springframework.orm.hibernate5.LocalSessionFactoryBean" id="vdsOracleSessionFactory"**>  
 <**property name="dataSource" ref="caseOperation"**/>  
  
 *<!-- 主动扫描包内实体对象 -->* <**property name="packagesToScan"**>  
 <**list**>  
 <**value**>com.hikvision.\*\*.domain</**value**>  
 </**list**>  
 </**property**>  
 <**property name="configLocations"**>  
 <**list**>  
 <**value**>classpath:hibernate-oracle.cfg.xml</**value**>  
 </**list**>  
 </**property**>  
</**bean**>

1. 然后就可以通过继承HibernateDaoSupport，为其配置sessionFactory即可

**public abstract class** HibernateUtil<T> **extends** HibernateDaoSupport {  
  
 @Resource(name = **"vdsOracleSessionFactory"**)  
 **public void** setSessionFactoryEx(SessionFactory sessionFactory) {  
 **super**.setSessionFactory(sessionFactory);  
 }

**public void** delete(T entity) {  
 **try** {  
 **this**.getHibernateTemplate().delete(entity);  
 } **catch** (Exception e) {  
 ***logger***.error(e);  
 }  
 }

……

}

在其他与数据库操作相关的DAO层使用时，只需要继承HibernateUtil<T>即可，例如：

**public class** CaseClueDAO **extends** HibernateUtil<CaseClueInfo>

## 二、利用反射调用方法

在VIID中专门编写了一个类，用于作为方法参数（传递方法进去），目的是通过反射的方式执行其方法。

例如以下方法：

consumer.consume(**topic**, MethodObject.*function*(**new** Object() {  
 @SuppressWarnings(**"unused"**)  
 **public void** consume( ) {  
 ……  
 }  
}));

其中，MethodObject就是为了作为方法参数传递方法进去而存在的类

**public class** MethodObject {  
   
 **private** Object **target**;  
 **private** String **method\_name**;

**public static** MethodObject function(Object target, Object... objects) {  
 MethodObject mo = **new** MethodObject(target);  
 **return** mo;  
 }

**public** Object invoke(Object... objects) {  
 Class clazz = **target**.getClass();  
 **try** {  
 Method[] ms = clazz.getDeclaredMethods();  
   
 Method targetMethod = **null**;  
 **if** (**method\_name** == **null** && ms.**length** == 1) {  
 targetMethod = ms[0];  
 } **else if** (**method\_name** != **null** && ms.**length** >= 1) {  
 **for** (Method m : ms) {  
 **if** (**method\_name**.equals(m.getName())) {  
 targetMethod = m;  
 **break**;  
 }  
 }  
   
 } **else** {  
 **return null**;  
 }  
 **if**(**null** == targetMethod)  
 {  
 **return null**;  
 }  
 targetMethod.setAccessible(**true**);  
 **return** targetMethod.invoke(**target**, objects);  
 } **catch** (Exception e) {  
 **throw new** RuntimeException(e);  
 }  
}

}

而之所以调用到了反射指向的方法，还是因为在consume方法中调用了methodObject的invoke方法。

**public void** consume(String topicName, MethodObject methodObject) {  
  
 ConcurrentMap<String, Acknowledgment> ackMap = **new** ConcurrentHashMap<>();  
 **container**.getContainerProperties().setMessageListener(**new** AcknowledgingMessageListener<String, String>() {  
 @Override  
 **public void** onMessage(ConsumerRecord<String, String> record, Acknowledgment acknowledgment) {  
 ackMap.put(String.*valueOf*(record.partition()),acknowledgment);  
 methodObject.invoke(record.value(), ackMap);  
 }  
 });

……  
 **container**.start();  
}

这里之所以使用反射的方式，是因为对于不同的消费者要对消息做不同的操作，而它们都是同一个类Consumer的实例。为了让其执行满足各自需求的方法，所以采用了这样的方式。

## 三、对于请求结果的缓存

Viid中采集设备的数据信息需要调用其他平台系统获得，而且往往被不止一次的查询，为了提高性能，使用了Guava的Cache

Guava Cache与ConcurrentMap的区别：

他们很相似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管LoadingCache 不回收元素，它也是很有用的，因为它会自动加载缓存。

有两种创建方式，通过CacheLoader或者Callable，在viid中是CacheLoader的方式

在viid中的使用：

**private static** LoadingCache<String,List< Ape>> *APE\_MAP* = CacheBuilder.*newBuilder*().expireAfterWrite(60, TimeUnit.***MINUTES***).build(**new** MyCacheLoader());

**private static class** MyCacheLoader **extends** CacheLoader<String,List<Ape> > {  
 @Override  
 **public** List<Ape> load(String key) **throws** Exception {  
 **return null**;  
 }  
}

其中的expireAfterWrite方法表示缓存项在给定时间内没有被写访问（创建或覆盖），则回收。

同时，通过方法定时的向缓存中更新值

@Scheduled(fixedDelay = 40 \* 60 \* 1000)  
**public void** execute() {  
 */\*定时将人脸设备和车辆设备存到内存里面\*/* List<Ape> fasApeList = CrossingInfoSyncUtils.*syncDeviceInfoFromFas*(**true**);  
 *//过滤设备* nonstandardApesFilter(fasApeList,**"人脸"**);  
 **int** fasStandardSize = fasApeList.size();  
 **if**(fasStandardSize > 0) {  
 *APE\_MAP*.put(*FAS\_APES*, fasApeList);  
 }

……

}

这样，在获取采集设备的时候就不需要每次都发起请求了，通过APE\_MAP.get(apeType);就可以

## 四、关于自定义Content-Type的解析

依据1400标准，viid在发送接收http请求响应时，其Content-Type为自定义的application/viid+json;charset=UTF-8

可以通过RequestMapping中produces属性可以设置返回数据的类型以及编码。但是必须要和@ResponseBody注解一起使用才可以，不加@ResponseBody注解相当于按照和返回String同名jsp页面解析自然就会报错。

@RequestMapping(value = **"/Subscribes"**, produces = **"application/VIID+JSON;charset=UTF-8"**, method = RequestMethod.***POST***)  
**public** ResponseStatusListObject createSubscribes(@RequestBody String params, HttpServletRequest request, **boolean**... isTest) {  
 ……;

}

然后设置json转换器

<**mvc:annotation-driven**>  
 <**mvc:message-converters register-defaults="true"**>  
 *<!--json处理 -->* <**bean class="org.springframework.http.converter.StringHttpMessageConverter"**></**bean**>  
 <**bean id="mappingJacksonHttpMessageConverter" class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"**>  
 <**property name="supportedMediaTypes"**>   
 <**list**>   
 <**value**>application/json;charset=UTF-8</**value**>  
 <**value**>application/VIID+JSON;charset=UTF-8</**value**>  
 <**value**>text/html;charset=UTF-8</**value**>   
 </**list**>   
 </**property**>   
 <**property name="objectMapper" ref="customObjectMapper"**></**property**>  
 </**bean**>  
 </**mvc:message-converters**>  
</**mvc:annotation-driven**>

其他，为了在返回结果中不显示为null的字段，也可以通过注解的方式

@JsonIgnoreProperties({**"ResponseStatus"**})  
@JsonInclude(JsonInclude.Include.***NON\_NULL***)  
**public class** ResponseStatusListObject **extends** ResponseStatusWarpper {  
 @JsonProperty(**"ResponseStatusListObject"**)  
 **protected** ResponseStatusObject **responseStatusList**;  
 @JsonIgnore  
 **public** ResponseStatusObject getResponseStatusList() {  
 **return responseStatusList**;  
 }  
 @JsonIgnore  
 **public void** setResponseStatusList(ResponseStatusObject responseStatusList) {  
 **this**.**responseStatusList** = responseStatusList;  
 }  
}

## 五、详细的日志记录

VIID作为中间层面的网关，涉及到的上下游平台很多，记录的日志必须详细且各个类目要有区分，才可以发挥日志的作用。

Viid为不同的类目，比如车辆、人脸、人员等都编写了自己的java类，在需要添加日志时调用相应类的静态方法即可。

如：

**public class** FaceSendLog {  
 **private static final** Logger ***LOG*** = LoggerFactory.*getLogger*(FaceSendLog.**class**);  
 **public static void** printLog(String message, **boolean**... isDebug) {  
 **if** (isDebug.**length** > 0 && isDebug[0]) {  
 ***LOG***.debug(message);  
 } **else** {  
 ***LOG***.info(message);  
 }  
 }  
}

*###输出异常信息到单独的日志文件###***log4j.logger.com.hikvision.viid.common.utils.log.FaceSendLog**=**INFO, SUBSCRIBE\_FACE  
log4j.appender.SUBSCRIBE\_FACE** = **org.apache.log4j.RollingFileAppender  
log4j.appender.SUBSCRIBE\_FACE.File** = **${catalina.home}/logs/subscribe/viid3\_log4j\_SUBSCRIBE\_FACE.log  
log4j.appender.SUBSCRIBE\_FACE.MaxFileSize** = **512KB  
log4j.appender.SUBSCRIBE\_FACE.MaxBackupIndex** = **5  
log4j.appender.SUBSCRIBE\_FACE.Threshold** = **INFO  
log4j.appender.SUBSCRIBE\_FACE.layout** = **org.apache.log4j.PatternLayout  
log4j.appender.SUBSCRIBE\_FACE.layout.ConversionPattern** =**%d{yyyy-MM-dd HH:mm:ss.SSS} - %m%n**

而且，对于调试、定位问题时要显示的日志和线上环境的日志，可以通过log4j.properties文件中INFO为DEBUG进行控制。因为在代码中调试时要显示的日志内容，调用的是printLog(message, true)