目录

Takia DDD 是一个基于 Spring 扩展所得一个支持领域建模的轻量级插件，主要基于 Spring 对 DDD 开发过程中的 In-Memory 和 DomainEvents 机制缺失的问题进行完善，使其能够更好的支持 DDD 建模。

一.In-memory建模

DDD 的 In-Memory 方法，将系统设计由数据库模式移植到内存中，即让系统设计围绕内存中的领域对象进行设计，领域模型主要分为实体（Entity），值对象（VO），仓储（Repository）和服务（Service）等，在简单的情况下，仓储对象可以对应 SSH 中的 Dao 组件，服务可以与 SSH 中的服务对象对应，而实体和值对象跟 SSH 模型不同，Hibernate 的 model 是贫血模型，在这里我们叫 Pojo 对象，专门用于数据存储。 虽然实体和值对象跟 Pojo 之间存在对应关系,因为他们最后也将随着系统的处理被持久化到数据库,但是在 DDD 中，实体和值对象都是内存模型，意味着长期在内存里面存在，而不是像 Pojo 一样随着 hibernate 的 session 创建和销毁,并且 DDD 的实体模型是半充血模型，除了 pojo 对应的数据以外还存在很多额外的行为方法（主要是领域事件和数据处理，后面讲述），所以领域对象的存放就必须需要在内存中开辟缓存来支持 In-memory 机制。

二.领域缓存的使用 （CDD）

在 Takia 中有两种方式来支持模型缓存，1 是直接使用 ModelContainer 接口来实现领域对象缓存和领域对象（实体领域对象）的创建，ModelContainer 的默认实现为 DefaultCachingModelContainer 类，ModelContainer 的对象引用已经集成在 webbiz 的 Controller 基础类中，可以直接引用，其中几个比较关键的方法说明如下：

void addModel(ModelKey modelKey, Object model)：往缓存中放入模型对象，ModelKey 为缓存 Key，由模型的类和 ID 构成

void addModel(ModelKey modelKey, Object model, boolean enhance)：同上面方法，enhance 参数为 true，表示自动把实体的依赖关系进行注入

Object getModel(ModelKey modelKey)：根据一个 key 从缓存中获取模型对象，如果没找到返回 null

Object getModel(ModelKey modelKey, ModelLoader modelLoader)：同上面方法，如果没找到模型对象，则调用 modelLoader 对象进行加载

Object getModel(ModelKey modelKey, ModelLoader modelLoader, boolean required)：同上面方法，required 为 true 表示 modelLoader 不能返回空对象

Object removeModel(ModelKey modelKey)：根据 key 从缓存中删除一个模型对象

void identifiersToModels(MutablePagedList<Object> pagedList, Class<?> modelClass, ModelLoader modelLoader)：查询用接口，将 id 列表转换为实际的对象列表

<T> List<T> identifiersToModels(List<Object> identifiers, Class<T> modelClass, ModelLoader modelLoader)：同上面方法一个意思，只是没有分页信息

<T> T makeModel(Class<T> modelClass)：创建一个指定类型的模型对象

<T> T enhanceModel(T model)：将一个对象从普通对象增强为领域对象，增加事件支持和依赖关系注入

除了直接使用 ModelContainer，在 Takia 中使用缓存还有一种 AOP 方式如下：

```
@Introduce("domainCache")
public class XxxService {
    @Around //这个方法返回的 domain 对象将会被自动缓存到 ModelContainer 里面
    Public DomainA loadDomain(Long id) {
        //根据 ID 加载 DomainA 的对象
    }
}
```

注：该模式下缓存的删除需要通过 ModelContainer 手动进行

当然，Takia 同时也支持 Spring 的注解缓存：

```
public class XxxService {
    @Cacheable(value="modelCache", key="Model"+DomaiA.class.getName()+id 的值)
    //注意 key 的模式
    //这个方法返回的 domain 对象将会被自动缓存到 ModelContainer 里面
    public DomainA get(Long id) {
        //根据 ID 加载 DomainA 的对象
    }
    @CacheEvict(value="modelCache", key="Model"+DomaiA.class.getName()+id 的值)
    //注意 key 的模式
    //这个方法返回的 domain 对象将会被自动缓存到 ModelContainer 里面
    public void delete(Long id) {
```

```
        //根据 ID 删除 DomainA 的对象记录
    }
}
```

三.领域事件的支持（EDA）

Takia 对领域事件的支持比较强大，领域事件的出现不但有效的弥补了系统不同层的组件耦合问题，而异步事件处理方式对系统吞吐量的提高有着很大的帮助。Takia 通过领域消息（DomainMessage）模型实现基于注解的事件触发机制（有关事件的设计思想可以参考 Observer 和 Producer-Consumer 设计模式），在一般开发模式下，系统处理是从表现层到业务层再到持久层，而引入领域事件以后，处理过程变成从表现层到领域层，领域对象触发相关的业务事件到业务接收层再到仓储层进行数据库处理，所以事件消息可以看成领域对象与服务的纽带，而他们之间又没有直接的耦合关系，Takia 的消息模型底层分别独立支持 JdkFuture 和 Disruptor 机制，Future 模式是一个简单的线程池和可阻塞 Future，支持同步和异步方式，而 Disruptor 是一个完全异步的并发组件，具有较高的吞吐性能。他们两者使用上没有本质区别，下面先对开发中涉及到的几个类进行介绍:

@Send 注释消息发布者（Message-Publisher），里面有四个属性：value 表示主题名，即产生的事件名称，action 属性只能在 future 模式下使用，表示所调用接受者的方法，asyn 属性是一个同步异步开关，只能在 future 模式下使用，type 属性是消息模式选择，目前只可以设置 disruptor 和 future 两个值，默认为 disruptor。

在 future 模式下只支持 1：1 的消息发布接收模式，Spring 的 @Component，@Service 注解的组件均可以接收到@Send 发出的消息

在 disruptor 模式下为了支持 1：N，消息接收者多出了两个注解：

@OnEvent 注释在接收消息的方法上面，value 属性跟@Send 的 value 属性对应

@Consumer 注释在接受者的类上面，value 属性跟@Send 的 value 属性对应

DomainMessage 类，是消息实体，在消息的发布者与接受者之间传递输入参数和结果，发送者通过 DomainMessage 的 eventSource 属性来设置事件源对象，产生一个消息可以直接使用构造函数：

new DomainMessage(eventSource)

这个 eventSource 对象可以在接收者中获取到，并通过他来进行业务处理，处理结果，接收者同样可以通过 DomainMessage 进行设置如下：

DomainMessage.setEventResult(result)

消息处理完后，发送者可以通过 DomainMessage.getEventResult()获

取到接收者的处理结果。

Jdk Future 和 Disruptor 下的消息机制开发举例：

Jdk Future 模式下（1:1 模式）：

消息发送者（Message Publisher）：

```
@Introduce(“message”)
public class FutureDomainEvents {
 @Send(value = “futureChannel”, action=”onEvent1”, asyn=true/* 异步模式 */, type=”future”/*消息模式*/)
 public DomainMessage sendAsynMessage(Object eventSource) {
        return new DomainMessage(eventSource);
 }

@Send(value = “futureChannel”/*接受者名*/, action=”onEvent2”, asyn=false/*同步模式*/, type=”future”)
 public DomainMessage sendSynMessage(Object eventSource) {
   return new DomainMessage(eventSource);
 }

@Send(value = “futureChannel2”, type=”future”)
 public DomainMessage sendAsynMessage(Object eventSource) {
   return new DomainMessage(eventSource);
 }
}
```

Jdk Future 模式下的消息接受者：

```
@Component(“futureChannel”)
public class ChannelListener {
public void onEvent1(DomainMessage message) {
  // do process with message
}
```

```
public void onEvent2(DomainMessage message) {
  // do process with message
}
}

@Component("futureChannel2")
// @Send 没有指定 action 属性时 listener 必须实现 MessageListener 接口
public class ChannelListener2 implements MessageListener {
  @Override
  public void action(DomainMessage message) throws Exception {
     // do process with message
  }
}
```

Disruptor 模式下（高并发异步模式，1：N）：

消息发送者（Message Publisher）：

```
@Introduce("message")
public class DisruptorDomainEvents {
 @Send("topicName")  // disruptor 模式下只有 value 属性起作用
 public DomainMessage sendAsynMessage(Object eventSource) {
        return new DomainMessage(eventSource);
 }
}
```

消息接收者支持以下方式：

1) @Consumer 注解方式

```
@Consumer("topicName")
public class ConsumerListener1 implements DomainEventListener {
   @Override
   public void onEvent(DisruptorEvent event, Boolean endOfBatch) throws Exception {
```

```
        // do process with message
        // DomainMessage object can be retrieved by event.getDomainMessage()
    }
}
```

2) @OnEvent 注解方式，支持以下四种参数形式的方法 Dispatch

```
@Component
public class ConsumerListener2 {
    @OnEvent("topicName")
    public void onEvent(DisruptorEvent event, Boolean endOfBatch) throws Exception {
        // do process with message
    }

@OnEvent("topicName")
    public void onEvent(DisruptorEvent event) throws Exception {
        // do process with message
    }

@OnEvent("topicName")
    public void onEvent(DomainMessage message) throws Exception {
        // do process with message
    }

@OnEvent("topicName")
    public Object onEvent(Object eventSource) throws Exception {
        // do process with event source and
        // return process result, which will be auto filled in DomainMessage
    }
}
```

3) 若没有找到 Consumer 则会扫描@Component 组件的名字，作为 fallback：

```
@Component(“topicName”)
public class ConsumerListenerFallback implements DomainEventListener {
    @Override
    public void onEvent(DisruptorEvent event, Boolean endOfBatch) throws Exception {
        // do process with message
    }
}
```

注：在这两种模式下，若消息未找到接收组件，均会抛出异常报错，终止该消息的 publish 过程！

四、Ehcache和Memcache配置

Takia 默认使用 EhCache 作为缓存组件，也支持 Memcache 方式，两者可以透明切换，均使用 Spring Cache 接口进行交互。

4.1 Ehcache配置

EhCache 模式下的 Spring 上下文配置如下：

```
<cache:annotation-driven cache-manager="cacheManager"/>
<context:component-scan base-package="com.reeham.component.ddd" />
<context:annotation-config />
<bean id="cacheManagerFactory"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
    p:configLocation="classpath:/config/ehcache.xml"/>
<bean id="cacheManager"
class="org.springframework.cache.ehcache.EhCacheCacheManager"
    p:cacheManager-ref="cacheManagerFactory"/>
```

在 ehcache.xml 文件中需要配置一个名为 modelCache 的缓存，示例如下：

```
<cache name="modelCache" maxElementsInMemory="10000"
    maxElementsOnDisk="1000" eternal="false" overflowToDisk="true"
    diskSpoolBufferSizeMB="20" timeToIdleSeconds="300" timeToLiveSeconds="600"
    memoryStoreEvictionPolicy="LRU" />
```

4.2 Memcache配置

Memcache 模式下的 Spring 上下文配置如下：

```
<cache:annotation-driven cache-manager="cacheManagerWrapper"/>
<context:component-scan base-package="com.reeham.component.ddd" />
<context:annotation-config />
```

```
    <bean id="memcachedClientBuilder"
class="net.rubyeye.xmemcached.XMemcachedClientBuilder">
        <!-- XMemcachedClientBuilder have two arguments.First is server
list,and second is weights array. -->
        <constructor-arg>
            <list>
                <bean class="java.net.InetSocketAddress">
                    <constructor-arg>
                        <value>127.0.0.1</value>
                    </constructor-arg>
                    <constructor-arg>
                        <value>11211</value>
                    </constructor-arg>
                </bean>
                <!--  <bean class="java.net.InetSocketAddress">
                    <constructor-arg>
                        <value>localhost</value>
                    </constructor-arg>
                    <constructor-arg>
                        <value>12001</value>
                    </constructor-arg>
                </bean> -->
            </list>
        </constructor-arg>
        <constructor-arg>
            <list>
                <value>1</value>
                <!-- <value>2</value> -->
            </list>
        </constructor-arg>
        <!-- <property name="authInfoMap">
            <map>
                <entry key-ref="server1">
                    <bean class="net.rubyeye.xmemcached.auth.AuthInfo"
                        factory-method="typical">
                        <constructor-arg index="0">
                            <value>cacheuser</value>
                        </constructor-arg>
                        <constructor-arg index="1">
                            <value>123456</value>
                        </constructor-arg>
                    </bean>
                </entry>
            </map>
```

```
        </property> -->
        <property name="connectionPoolSize" value="2"></property>
        <property name="commandFactory">
            <!-- <bean
class="net.rubyeye.xmemcached.command.TextCommandFactory"></bean> -->
            <bean
class="net.rubyeye.xmemcached.command.BinaryCommandFactory"></bean>
        </property>
        <property name="sessionLocator">
            <bean
class="net.rubyeye.xmemcached.impl.KetamaMemcachedSessionLocator"></bean>
        </property>
        <property name="transcoder">
            <bean
class="net.rubyeye.xmemcached.transcoders.SerializingTranscoder" />
        </property>
    </bean>
    <!-- Use factory bean to build memcached client -->
    <bean id="memcachedClient3" factory-bean="memcachedClientBuilder"
        factory-method="build" destroy-method="shutdown" />

    <bean id="cacheManagerWrapper"
class="com.reeham.component.ddd.cache.wrapper.CacheManagerWrapper">
        <property name="cacheManager">
            <bean
class="com.reeham.component.ddd.cache.memcache.MemcacheCacheManager">
                <property name="client" ref="memcachedClient3" />
            </bean>
        </property>
        <property name="valuePostProcessors">
            <list>
                <bean
class="com.reeham.component.ddd.model.cache.ModelAwareValuePostProcessor">
                    <property name="supportedCacheNames">
                       <list>
                          <value>modelCache</value>
                       </list>
                    </property>
                    <property name="recursiveReinject" value ="true">
                    </property>
                </bean>
            </list>
        </property>
    </bean>
```

```
    <bean id="modelCache"
class="com.reeham.component.ddd.cache.memcache.MemcacheCache">
        <property name="expiry" value="600" />
    </bean>
```

注： 在 Memcache 模式下，为了提升缓存性能，domain 对象的关联对象尽量精简，在设计时，把 lazy-load 的属性，和容器中存在的组件依赖最好都设置成 transient 字段，takia 可以在缓存序列化时，自动透明的剥离和恢复这些数据。示例如下：

```
    @Model
    public class User {
        private String name;
        private String password;
        @Resource   //will be auto-restored by takia
        private transient domainEvents;
        private transient Collection<Role> roles ;
        // lazy-load aware method
        public Collection<Role> getRoles() {
            if (roles == null) {
                DomainMessage message = domainEvents.loadRolesByUser(this);
                this.roles = (Collections<Role>)message.getEventResult ();
             }
             return roles;
        }
        // other domain methods
    }
```