## 第一节 DDD领域驱动概述领域驱动设计简述

基本原理

\* 1） DDD领域驱动产生的原因

在所有的软件开发设计中，我们要遵循的核心就是要设计的软件一定是一服务于某个领域。

所以我们的软件存在的意义就是解决业务问题和其他现实世界的问题，如果没有这些其他领域的知识，就不能解决这些问题。比如如果不懂银行金融方面的知识，我们就无法设计出一款银行金融系统。这是就需要相关领域知识来帮助我们。

当我们将领域驱动和软件设计结合，就由此产生

2） DDD领域驱动基本原理

本身原理也是它产生的原因，他是设计师、领域专家和开发人员的集体工作。

3） 微服务关联

在微服务中，我们最核心的地方就是对业务的拆分处理，根据业务的耦合度进行拆分或者聚合。

而DDD领域驱动本身也就是领域专家、设计人员等对领域业务信息进行组织、将其分成较小的部分

，从逻辑上对他们进行分组并创建模型。每个模块可以单独处理。

这样两者就不谋而合。

） 好处

DDD是开发人员和设计人员团结合作，同时从开发人员收到的反馈保持设计与时俱进的能力

，也解决敏捷开发和瀑布式开发所带来某些的局限性，让代码和设计都具有可维护性

关于领域驱动设计中的几个概念VO、DTO、DO、PO

概念解释

VO（View Object）：视图对象，用于展示层，它的作用是把某个指定页面（或组件）的所有数据封装起来。

DTO（Data Transfer Object）：数据传输对象，这个概念源于J2EE的设计模式，原来的目的是为了EJB的分布式应用提供粗粒度的数据实体，以减少分布式调用的次数，从而提高分布式调用的性能和降低网络负载.在这里，泛指用于展示层与服务层之间的数据传输对象。

DO（Domain Object）：领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。

PO（Persistent Object）：持久化对象，跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么，数据表中的每个字段（或若干个）就对应PO的一个（或若干个）属性。

2）在项目中的使用场景：

以一个请求时序图建立简单模型来描述上述对象在三层架构应用中出现叙述

用户发出请求（可能是填写表单），表单的数据在展示层被匹配为VO。

展示层把VO转换为服务层对应方法所要求的DTO，传送给服务层。

服务层首先根据DTO的数据构造（或重建）一个DO，调用DO的业务方法完成具体业务。

服务层把DO转换为持久层对应的PO（可以使用ORM工具，也可以不用），调用持久层的持久化方法，把PO传递给它，完成持久化操作。

对于一个逆向操作，如读取数据，也是用类似的方式转换和传递，略。

3） VO与DTO的区别

绝大部分的应用场景来说，DTO和VO的属性值基本是一致的，而且他们通常都是POJO，因此没必要多此一举，但不要忘记这是实现层面的思维，对于设计层面来说，概念上还是应该存在VO和DTO，因为两者有着本质的区别，DTO代表服务层需要接收的数据和返回的数据，而VO代表展示层需要显示的数据。

举例说明，用户的性别坑DTO记录为0、1，而VO要根据客户端不同该站是可能是男女、也可能是帅哥美女。回到设计层面上分析，从职责单一原则来看，服务层只负责业务，与具体的表现形式无关，因此，它返回的DTO，不应该出现与表现形式的耦合。

4） VO与DTO的应用

1. 考虑把VO与DTO二合为一（注意：是实现层面）：

2. 当需求非常清晰稳定，而且客户端很明确只有一个的时候，没有必要把VO和DTO区分开来，这时候VO可以退隐，用一个DTO即可，为什么是VO退隐而不是DTO？回到设计层面，服务层的职责依然不应该与展示层耦合，所以，对于前面的例子，你很容易理解，DTO对于“性别”来说，依然不能用“帅哥美女”，这个转换应该依赖于页面的脚本（如JavaScript）或其他机制（JSTL、EL、CSS）

3. 即使客户端可以进行定制，或者存在多个不同的客户端，如果客户端能够用某种技术（脚本或其他机制）实现转换，同样可以让VO退隐

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

以下场景优先考虑VO、DTO并存：

1. 上述场景的反面场景

2. 因为某种技术原因，比如某个框架（如Flex）提供自动把POJO转换为UI中某些Field时，可以考虑在实现层面定义出VO，这个权衡完全取决于使用框架的自动转换能力带来的开发和维护效率提升与设计多一个VO所多做的事情带来的开发和维护效率的下降之间的比对。

3. 如果页面出现一个“大视图”，而组成这个大视图的所有数据需要调用多个服务，返回多个DTO来组装（当然，这同样可以通过服务层提供一次性返回一个大视图的DTO来取代，但在服务层提供一个这样的方法是否合适，需要在设计层面进行权衡）。

DTO与DO的区别

概念上的区别，DTO是展示层和服务层之间的数据传输对象（可以认为是两者之间的协议），而DO是对现实世界各种业务角色的抽象，这就引出了两者在数据上的区别，例如UserInfo和User（DTO和DO的命名规则），对于一个getUser方法来说，本质上它永远不应该返回用户的密码，因此UserInfo至少比User少一个password的数据。而在领域驱动设计中，DO不是简单的POJO，它具有领域业务逻辑。

DTO与DO的应用

在设计层面，展示层向服务层传递的DTO与服务层返回给展示层的DTO在概念上是不同的，但在实现层面，我们通常很少会这样做（定义两个UserInfo，甚至更多），因为这样做并不见得很明智，我们完全可以设计一个完全兼容的DTO，在服务层接收数据的时候，不该由展示层设置的属性（如订单的总价应该由其单价、数量、折扣等决定），无论展示层是否设置，服务层都一概忽略，而在服务层返回数据时，不该返回的数据（如用户密码），就不设置对应的属性。

对于DO来说，还有一点需要说明：为什么不在服务层中直接返回DO呢？这样可以省去DTO的编码和转换工作，原因如下：

1. 两者在本质上的区别可能导致彼此并不一一对应，一个DTO可能对应多个DO，反之亦然，甚至两者存在多对多的关系。

2. DO具有一些不应该让展示层知道的数据

3. DO具有业务方法，如果直接把DO传递给展示层，展示层的代码就可以绕过服务层直接调用它不应该访问的操作，对于基于AOP拦截服务层来进行访问控制的机制来说，这问题尤为突出，而在展示层调用DO的业务方法也会因为事务的问题，让事务难以控制。

4. 对于某些ORM框架（如Hibernate）来说，通常会使用“延迟加载”技术，如果直接把DO暴露给展示层，对于大部分情况，展示层不在事务范围之内（Open session in view在大部分情况下不是一种值得推崇的设计），如果其尝试在Session关闭的情况下获取一个未加载的关联对象，会出现运行时异常（对于Hibernate来说，就是LazyInitiliaztionException）。

5. 从设计层面来说，展示层依赖于服务层，服务层依赖于领域层，如果把DO暴露出去，就会导致展示层直接依赖于领域层，这虽然依然是单向依赖，但这种跨层依赖会导致不必要的耦合。

对于DTO来说，也有一点须进行说明，就是DTO应该是一个“扁平的二维对象”，举个例子来说明：如果User会关联若干个其他实体（例如Address、Account、Region等），那么getUser()返回的UserInfo，是否就需要把其关联的对象的DTO都一并返回呢？如果这样的话，必然导致数据传输量的大增，对于分布式应用来说，由于涉及数据在网络上的传输、序列化和反序列化，这种设计更不可接受。如果getUser除了要返回User的基本信息外，还需要返回一个AccountId、AccountName、RegionId、RegionName，那么，请把这些属性定义到UserInfo中，把一个“立体”的对象树“压扁”成一个“扁平的二维对象”，笔者目前参与的项目是一个分布式系统，该系统不管三七二十一，把一个对象的所有关联对象都转换为相同结构的DTO对象树并返回，导致性能非常的慢。

7） DO与PO的区别

DO和PO在绝大部分情况下是一一对应的，PO是只含有get/set方法的POJO，但某些场景还是能反映出两者在概念上存在本质的区别：

1 DO在某些场景下不需要进行显式的持久化，例如利用策略模式设计的商品折扣策略，会衍生出折扣策略的接口和不同折扣策略实现类，这些折扣策略实现类可以算是DO，

但它们只驻留在静态内存，不需要持久化到持久层，因此，这类DO是不存在对应的PO的。

2 同样的道理，某些场景下，PO也没有对应的DO，例如老师Teacher和学生Student存在多对多的关系，在关系数据库中，这种关系需要表现为一个中间表，也就对应有一个TeacherAndStudentPO的PO，但这个PO在业务领域没有任何现实的意义，它完全不能与任何DO对应上。这里要特别声明，并不是所有多对多关系都没有业务含义，这跟具体业务场景有关，例如：两个PO之间的关系会影响具体业务，并且这种关系存在多种类型，那么这种多对多关系也应该表现为一个DO，又如：“角色”与“资源”之间存在多对多关系，而这种关系很明显会表现为一个DO——“权限”。

3 某些情况下，为了某种持久化策略或者性能的考虑，一个PO可能对应多个DO，反之亦然。例如客户Customer有其联系信息Contacts，这里是两个一对一关系的DO，但可能出于性能的考虑（极端情况，权作举例），为了减少数据库的连接查询操作，把Customer和Contacts两个DO数据合并到一张数据表中。反过来，如果一本图书Book，有一个属性是封面cover，但该属性是一副图片的二进制数据，而某些查询操作不希望把cover一并加载，从而减轻磁盘IO开销，同时假设ORM框架不支持属性级别的延迟加载，那么就需要考虑把cover独立到一张数据表中去，这样就形成一个DO对应对个PO的情况。

4 PO的某些属性值对于DO没有任何意义，这些属性值可能是为了解决某些持久化策略而存在的数据，例如为了实现“乐观锁”，PO存在一个version的属性，这个version对于DO来说是没有任何业务意义的，它不应该在DO中存在。同理，DO中也可能存在不需要持久化的属性。

8） DO与PO的应用

ORM框架的功能非常强，而且JavaEE也推出了JPA规范，现在的业务应用开发，基本上不需要区分DO与PO，PO完全可以通过JPA，Hibernate Annotations/hbm隐藏在DO之中。虽然如此，但有些问题还必须注意：

1 对于DO中不需要持久化的属性，需要通过ORM显式的声明，如：在JPA中，可以利用@Transient声明。

2 对于PO中为了某种持久化策略而存在的属性，例如version，由于DO、PO合并了，必须在DO中声明，但由于这个属性对DO是没有任何业务意义的，需要让该属性对外隐藏起来，最常见的做法是把该属性的get/set方法私有化，甚至不提供get/set方法，但对于Hibernate来说，这需要特别注意，由于Hibernate从数据库读取数据转换为DO时，是利用反射机制先调用DO的空参数构造函数构造DO实例，然后再利用JavaBean的规范反射出set方法来为每个属性设值，如果不显式声明set方法，或把set方法设置为private，都会导致Hibernate无法初始化DO，从而出现运行时异常，可行的做法是把属性的set方法设置为protected。

3 对于一个DO对应多个PO，或者一个PO对应多个DO的场景，以及属性级别的延迟加载，Hibernate都提供了很好的支持，请参考Hibnate的相关资料。

总结出一个原则：分析设计层面和实现层面完全是两个独立的层面，即使实现层面通过某种技术手段可以把两个完全独立的概念合二为一，在分析设计层面，我们仍然（至少在头脑中）需要把概念上独立的东西清晰的区分开来，这个原则对于做好分析设计非常重要（工具越先进，往往会让我们越麻木）

补充

服务

一些行为会影响对象，使用面向对象编程可以将他们附加在对象上，这就是所谓的服务。

服务在就技术框架中是很常见的，这些都用在DDD领域层。

聚合

聚合领域模式和对象的生命周期相关，并定义所有权和边界。

用户订餐的中,聚合的一个简单的例子就是客户。客户可以接触到外部的对象，并且

其跟对象包含内部的对象的地址、联系方式。当发出请求，就可以吧内部对象的信息诸如地址信息

传递给外部对象。

存储库

保存领域对象的生命周期、比如数据库或者缓存等

主要参考《DDD领域驱动设计》 《java微服务》 博文http://www.cnblogs.com/qixuejia/p/4390086.html

## 第二节 DDD领域部分DDD补充部分

统一建模语言UML，为方便设计人员、领域专家和程序员都能看懂的一种通用语言

多层架构是DDD的一个通用解决方案，他包含四个层面

1. 表示层或者用户界面层（ui层）

2. 应用层

3. 领域层

4. 基础设施层

一般只有领域层负责领域模型，其他层为其服务，这设计的好处是将领域相关的代码分离开来

具体每层的含义不再赘述。

本节考虑在编写代码时，使用DDD领域驱动的理念设计部分code

实体的实现

对于第一个接口，可以提供一个实体所需的抽象类或者接口

public abstract class EntityT {

T id;

String name;

}

基于此抽象可以实现一个抽象的基础实体类

public abstract class BaseEntityT extends EntityT{

private T id;

public BaseEntity(T id, String name){

super.id = id;

super.name = name;

}

//.... get set 方法 以及其他相关代码

}

基于前面的抽象类创建restaurant实体，我们开发的是餐馆订座系统，所以Table是领域模型中

另一个重要的实体，这里由于业务，我们可以采用聚合模式，restaurant会作为一个根来工作，

而Table将是restaurant实体中的内部实体。因此Table实体始终都可以使用restaurant来访问。

下面作为示范，添加了基本属性

/\*\*

\* @author xzg

\*/

public class Table extends BaseEntityBigInteger {

private int capacity;

public Table(BigInteger id, String name, int capacity) {

super(id, name);

this.capacity = capacity;

}

public void setCapacity(int capacity) {

this.capacity = capacity;

}

public int getCapacity() {

return capacity;

}

}

现在我们实现restaurant聚合实体

\*\*

\* @author xzg

\*/

public class Restaurant extends BaseEntityString {

private ListTable tables = new ArrayList();

public Restaurant(String id, String name,ListTable tables) {

super(id, name);

this.tables = tables;

}

public ListTable getTables() {

return tables;

}

public void setTables(ListTable tables) {

this.tables = tables;

}

}

接下来存储库设计

作为开始我们先创建两接口Repository 和 ReadOnlyRepository。ReadOnlyRepository作为制度

操作的抽象，Repository 将作为所有类型的接口

/\*\*

\* @author xzg

\*/

public interface ReadOnlyRepositoryTE,T {

boolean contain(T id);

Entity get(T id);

CollectionTE getAll();

}

基于只读接口我再添加额外的操作

/\*\*

\* @author xzg

\*/

public interface RepositoryTE,T extends ReadOnlyRepository {

void add(TE entity);

void remove(T id);

void update(TE entity);

}

前面定义的抽象层可以使用合适的方式使用，上面属于存储服务的公共抽象层，具体不同存储业务，还需要我们根据具体去添加定义

下面提供餐馆服务的存储服务接口以及相应的实现类

/\*\*

\* @author xzg

\*/

public interface RestaurantRepositoryRestaurant,String extends RepositoryRestaurant,String {

boolean cotainsName(String name);

}

下面实现：

\*\*

\* @author xzg

\* @这里为方便

\* 使用内存作为存储，实际生产请使用数据库库或者缓存

\*/

public class InMemRestaurantRepository implements RestaurantRepositoryRestaurant,String {

private MapString,Restaurant entities;

public InMemRestaurantRepository(){

entities = new HashMap();

}

@Override

public boolean cotainsName(String name) {

return entities.containsKey(name);

}

@Override

public void add(Restaurant entity) {

entities.put(entity.getName(),entity);

}

@Override

public void remove(String id) {

if(entities.containsKey(id))

entities.remove(id);

}

@Override

public void update(Restaurant entity) {

if(entities.containsKey(entity.getName()))

entities.put(entity.getName(),entity);

}

@Override

public boolean contains(Object id) {

throw new UnsupportedOperationException("not support yet");

}

@Override

public Entity get(Object id) {

throw new UnsupportedOperationException("not support yet");

}

@Override

public Collection getAll() {

return entities.values();

}

}

服务层的设计实现

和前面的存储库服务一样我们把领域服务也抽象出，主抽象服务和只读抽象服务

/\*\*

\* @author xzg

\*/

public abstract class ReadOnlyBaseServiceTE,T {

private RepositoryTE,T repository;

ReadOnlyBaseService(RepositoryTE,T repository){

this.repository = repository;

}

//....

}

同理我们使用一个base服务去实现这个只读的服务,这里使用依赖注入的方式将存储服务注入到

领域服务中去。

/\*\*

\* @author xzg

\*/

public abstract class BaseServiceTE,T extends ReadOnlyBaseServiceTE,T {

private RepositoryTE,T \_repository;

BaseService(RepositoryTE,T repository){

super(repository);

\_repository = repository;

}

public void add(TE entity) throws Exception{

\_repository.add(entity);

}

public CollectionTE getAll(){

return \_repository.getAll();

}

}

有了基础的服务层，我们的具体领域服务也就可以根据自身业务需求设计

/\*\*

\* @author xzg

\*/

public class RestaurantService extends BaseServiceRestaurant, BigInteger {

private RestaurantRepositoryRestaurant,String restaurantRepository;

public RestaurantService(RestaurantRepository restaurantRepository){

super(restaurantRepository);

this.restaurantRepository = restaurantRepository;

}

@Override

public void add(Restaurant restaurant) throws Exception{

if(restaurantRepository.cotainsName(restaurant.getName()))

throw new Exception("name is repeat");

if(StringUtils.isEmpty(restaurant.getName()))

throw new NullPointerException("name is null");

super.add(restaurant);

}

//...others

}

总结：以上通过一个简单的订餐的领域，通过DDD分层设计。之后的篇章将会实现具体的业务。

## 微服务基本方案