浅析 VO、DTO、DO、PO 的概念、区别和用处！！

Java精选 2022-10-07 08:00 发表于北京

收录于合集
#Java基础

69个

由于不同的项目和开发人员有不同的命名习惯，这里我首先对上述的概念进行一个简单描述，名字只是个标识，我们重点关注其概念：

概念：

- **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，这个概念来源于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，但在这里，我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，它跟持久层（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。

模型：

下面以一个时序图建立简单模型来描述上述对象在三层架构应用中的位置

- 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。面试宝典：https://www.yoodb.com

VO与DTO的区别

大家可能会有个疑问（在笔者参与的项目中，很多程序员也有相同的疑惑）：既然DTO是展示层与服务层之间传递数据的对象，为什么还需要一个VO呢？对！对于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。

用一个例子来说明可能会比较容易理解：例如服务层有一个getUser的方法返回一个系统用户，其中有一个属性是gender(性别)，对于服务层来说，它只从语义上定义：1-男性，2-女性，0-未指定，而对于展示层来说，它可能需要用“帅哥”代表男性，用“美女”代表女性，用“秘密”代表未指定。说到这里，可能你还会反驳，在服务层直接就返回“帅哥美女”不就行了吗？

对于大部分应用来说，这不是问题，但设想一下，如果需求允许客户可以定制风格，而不同风格对于“性别”的表现方式不一样，又或者这个服务同时供多个客户端使用（不同门户），而不同的客户端对于表现层的要求有所不同，那么，问题就来了。再者，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。

理论归理论，这到底还是分析设计层面的思维，是否在实现层面必须这样做呢？一刀切的做法往往会得不偿失，下面我马上会分析应用中如何做出正确的选择。

VO与DTO的应用

上面只是用了一个简单的例子来说明VO与DTO在概念上的区别，本节将会告诉你如何在应用中做出正确的选择。

在以下才场景中，我们可以考虑把VO与DTO二合为一（注意：是实现层面）：

- 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）
- 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。

以下场景需要优先考虑VO、DTO并存：

- 上述场景的反面场景
- 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。
- 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

首先是概念上的区别，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，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，正如第一篇系列文章所说，DO不是简单的POJO，它具有领域业务逻辑。

对于DTO和DO的命名规则，请参见：

> https://www.cnblogs.com/qixuejia/p/10789612.html
> https://www.yoodb.com/framework/concurrent-cache.html

DTO与DO的应用

从上一节的例子中，细心的读者可能会发现问题：既然getUser方法返回的UserInfo不应该包含password，那么就不应该存在password这个属性定义，但如果同时有一个createUser的方法，传入的UserInfo需要包含用户的password，怎么办？

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：

- 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。
- DO具有一些不应该让展示层知道的数据。推荐公众号Java精选，回复Java面试，在先刷面试题。
- 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。
- 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）。
- 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，也有一点必须进行说明，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。

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：

- 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。

- 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”。推荐公众 号Java精选，回复Java面试，在先刷面试题。

- 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。

- 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。

DO与PO的应用

- 由于ORM框架的功能非常强大而大行其道，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，但有些问题我们还必须注意：

- 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。

- 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。

- 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。

到目前为止，相信大家都已经比较清晰的了解VO、DTO、DO、PO的概念、区别和实际应用了。通过上面的详细分析，我们还可以总结出一个原则：**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）。**

第一篇系列博文抛砖引玉，大唱领域驱动设计的优势，但其实领域驱动设计在现实环境中还是有种种的限制，需要选择性的使用，正如我在《田七的智慧》博文中提到，我们不能永远的理想化的去选择所谓“最好的设计”，在必要的情况下，我们还是要敢于放弃，因为最合适的设计才是最好的设计。

> 作者：Cat Qi
> cnblogs.com/qixuejia/p/4390086.html

收录于合集 #Java基础 69

上一篇
Java内部类有坑，100%内存泄露！

下一篇
面试官被欺负：new Object()到底占用几个字节？

阅读原文

喜欢此内容的人还喜欢

Mybatis中SQL注入攻击的3种方式，真是防不胜防！
江南一点雨

eCapture旁观者：Android HTTPS明文抓包
榫卯江湖

node应用故障定位顶级技巧—动态追踪技术[Dynamic Trace]
元语言