> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

《手册》关于分层模型部分的规约如下：

> 【参考】分层领域模型规约
> DO (Data Object): 此对象与数据库表结构一一对应，通过 DAO 层向上传输数据源对象。
>
> DTO (Data Transfer Object): 数据传输对象，Service 或 Manager 向外传输的对象。
>
> BO (Business Object): 业务对象，由 Service 层输出的封装业务逻辑的对象。
>
> AO (Application Object): 应用对象，在 Web 层与 Service 层之间抽象的复用对象模型，极为贴近展示层，复用度不高。
>
> VO (View Object): 显示层对象，通常是 Web 向模板渲染引擎层传输的对象。Query: 数据查询对象，各层接收上层的查询请求。
>
> 注意超过 2 个参数的查询封装，禁止使用 Map 类来传输。

那么我们需要思考以下几个问题：

- 为什么需要这些分层领域模型？
- 实际开发中每种分层领域模型都会用到吗？

本小节我们将重点分析和解答这些问题。

2.1 常见的分层模型有哪些？含义是什么？

学习和工作经常会接触到分层领域模型，如 DO、BO、DTO、VO 等。其中 DO、BO、DTO、AO、Query 在《手册》给出了一些解释，这里给出一些补充。

DTO (Data Transger Object) 为数据传输对象，通常将底层的数据聚合传给外部系统，它通常用作 Service 和 Manager 层向上层返回的对象。需要注意的是：如果作为分布式服务的参数或返回对象，通常要实现序列化接口。

Param 为查询参数对象，适用于各层，通常用作接受前端参数对象。Param 和 Query 的出现是为了避免使用 `Map` 作为接收参数的对象。

BO (Bussiness Object) 即业务对象。该对象中通常包含业务逻辑。此对象在实际使用中有不同的理解，有的团队采用领域驱动设计，BO 含有属性和方法（具体可参考领域驱动设计的相关图书）；有的团队将 BO 当做 Service 返回给上层的 “专用 DTO” 使用；而有的团队则当做 Service 层内保存中间信息数据的 “DTO” 或者上下文对象来使用（本文采用这种理解）。

比如 BO 中可以保存中间状态，放一些逻辑等，这些并不适合放在 `DTO` 中：

```
@Data
public class ItemBO {

    private Boolean isOnSell;

    private Boolean hasStock;

    private Boolean hasSensitiveWords;

    public Boolean isLegal() {
        if (isOnSell == null || hasStock == null || hasSensitiveWords == null) {
            return false;
        }
```

```
        return isOnSell && hasStock && (!hasSensitiveWords);
    }
}
```

VO (View Object) 为视图对象，通常作为控制层通过 JSON 返回给前端然后前端渲染或者加载页面模板在后端进行填充。

AO (Application Object) 应用对象。通常用在控制层和服务层之间。有些团队会将前端查询的属性和保存的属性几乎一致的对象封装为 AO，如读取用户属性传给前端，用户在前端编辑了用户属性后传回后端。这种用法将 AO 用作 Param 和 VO 或 Param 和 DTO 的组合。

2.2 为什么要有分层领域模型?

还有的朋友查询参数喜欢通过 `Map` 或者 `JSONObject` 来封装。有些朋友可能会认为这么多模型没有必要，因为通常各层模型的属性基本相同，而且各种类型的分层模型对象转换非常麻烦。

使用不同的分层领域模型能够让程序更加健壮、更容易拓展，可以降低系统各层的耦合度。

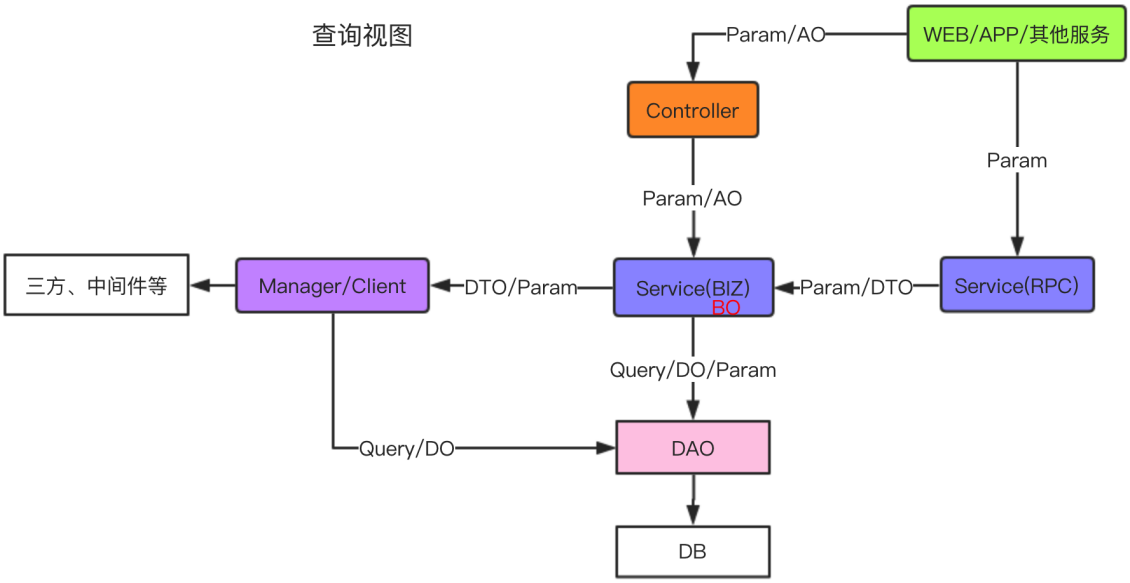
分层模型的优势只有在系统较大时才体现得更加明显。设想一下如果我们不想定义 DTO 和 VO，直接将 DO 用到数据访问层、服务层、控制层和外部访问接口上。此时该表删除或则修改一个字段，DO 必须同步修改，这种修改将会影响到各层，这并不符合高内聚低耦合的原则。通过定义不同的 DTO 可以控制对不同系统暴露不同的属性，通过属性映射还可以实现具体的字段名称的隐藏。不同业务使用不同的模型，当一个业务发生变更需要修改字段时，不需要考虑对其它业务的影响，如果使用同一个对象则可能因为 “不敢乱改” 而产生很多不优雅的兼容性行为。

如果我们不愿意定义 Param 对象，使用 Map 来接收前端的参数，获取时如果采用 JSON 反序列化，则可能出现上一节所讲到的反序列化类型丢失问题。如果我们不使用 Query 对象而是 `Map` 对象来封装 DAO 的参数，设置和获取的 `key` 很可能因为粗心导致设置和获取时的 key 不一致而出现 BUG。

讲完了概念和优势，大家可能会认为文字描述有些抽象，接下来通过查询和返回两个视角为大家展示实际项目中的一种常见的用法（贫血模型）。

3.1 查询视图

我们先从请求访问的视角去了解不同分层数据模型在实际项目中一种常见用法。

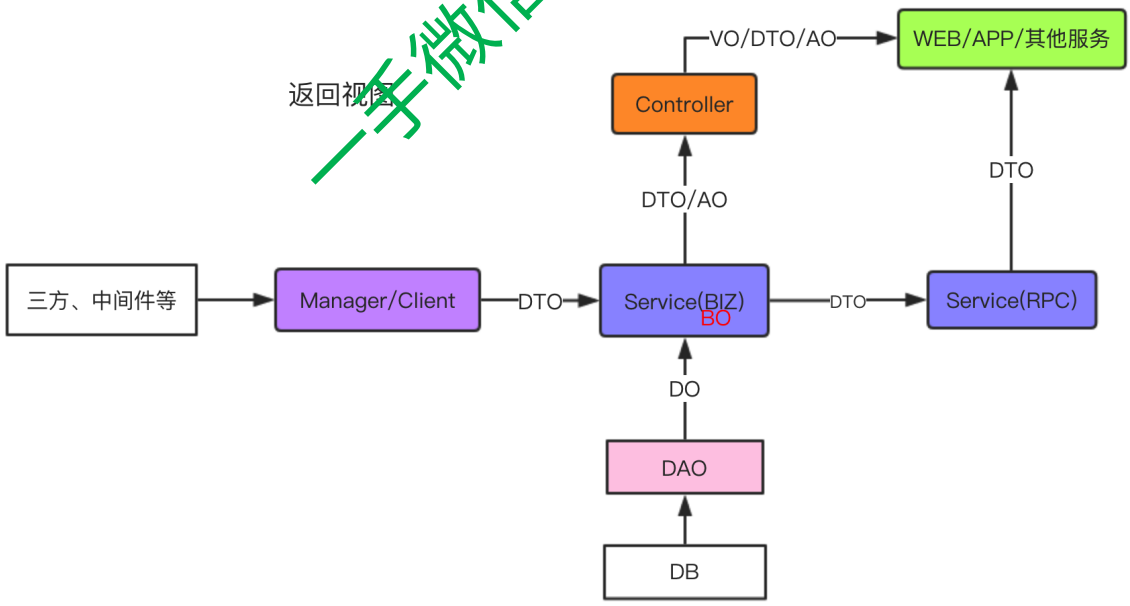


前端或者其它服务将 `Param` 对象作为参数传给控制层或者对外服务接口，然后调用内部的服务类，服务类内部的中间数据和这些数据相关的逻辑可以封装为 `BO` ，比如根据 `BO` 多个属性判断是否符合某个条件。

如果查询数据则封装为 `Query` 对象作为参数，如果需要查询其它依赖，则可以封装 `Param` 对象作为参数去查询。`DAO` 层一般插入和更新的参数对象使用 `DO` 或 `Param`，查询参数一般使用 `Query`，删除参数一般使用 `Param`。

3.2 返回视图

接下来我们从数据返回的视角去了解分层领域模型在实际项目中的一种常见用法：



数据访问层通常将数据封装为 `DO` 对象传给 `Service` 层，`Manager` 或 `Client` 层往往将查询结果封装为 `DTO` 传给 `Service` 层。

通常内部服务层通过 `DTO` 往外传输数据。`Controller` 通常将 `DTO` 组装为前端需要的 `VO` 或者直接将 `DTO` 外传 。

RPC 服务接口将 `DTO` 直接返回或者重新封装为新的 `DTO` 返回给外部服务。

另外即使同一个接口，但是一个对内使用，一个对外暴露，尽量使用不同接口，定义不同的参数和返回值，从而避免因为修改内部或外部的数据结构而导致另外一个受到影响，这也是单一职责原则的要求。

也有部分团队 RPC 的请求和响应参数都通过 DTO 来承载，通过 `XXRequestDTO` 和 `XXResponseDTO` 来表示。

实践分层领域模型能够提高项目的健壮性、可拓展性和可维护性，降低了系统内部各层的耦合度。

上面只是给出一种参考，很多团队对部分分层模型的理解会有差异，实际的使用过程中根据自己团队的规模可以适当变通。比如有很多团队项目并不是特别大，为了降低复杂度，只用到了 `DTO` 、 `VO` 、 `DO` 三种分层领域模型。

最后对分层领域模型的规约这里进行补充：

【参考】不提倡在 DTO 中写逻辑，强制不要在 RPC 返回对象的 DTO 中封装逻辑。

有些团队的个别成员会将根据成员属性作判断的一些函数写到 DTO 中，最奇葩的是该逻辑还主要供内部系统业务层使用。

如:

```
public class xxDTO{




 public static boolean  canXXX(){

 }

}
```


这样造成系统的耦合性非常强。

如果对方用到了这个函数，未来此函数的内部逻辑必须发生变化，未必能及时通知对方升级，容易造成 BUG。

即使耗费了成本找到了使用方，为了你的功能，让别人被迫升级版本重新上线也是非常不专业的事情。

显然这样做不合理。

> 试想一下今天 A 部门告诉你他们因某个功能被迫修改了某个 RPC 返回值 DTO 的某个方法，你们用到没有？用到升级一下哈...
>
> 然后 B 部门的人明天告诉你同样的话，然后 C 部门，然后...
>
> 你会不会崩溃？

建议如果需要在内部业务中写对实体相关的逻辑，可以考虑封装到工具类 / 帮助类中。

本节主要讲分层模型的目的和优势以及在实际开发中的常见用法。给大家一个参考，让大家能够在开发时知道哪些模型应该放到哪一层。

下一节将讲述不同的分层领域模型之间的转换的正确姿势。

在实际项目开发中，不同的分层领域模型之间通常需要转换，你是如何转换的?