# 数据对象

**VO(Value Object) 表现对象 ：控制器传递到视图的对象，没有任何数据操作。**只包含当前视图用户所输入的属性。例如登陆页面存在账号密码两个属性则封装一个对象只存在账号密码属性，，此类对象称之为VO应该与view model等价。

**DTO(Data Transfer Object) 数据传输对象 ：视图传递到控制器的数据接收对象，没有任何数据操作。**只包含当前视图需要展示的属性。例如一张表中的id为自增属性，不需要让用户知道则DTO对象属性包括除id之外表的其他属性。

**VO与DTO对比：**

**VO为视图传递到控制器的数据接收对象**

**public IActionResult Create(StudentDto Dto)**

**DTO为控制器传递到视图的对象**

**return View(Vo)**

**当展示与接受的数据一致时，则VO与DTO一致，此时只需要其中一个即可**

**BO(Business Object) 业务对象 ：主要作用是把业务逻辑封装为一个对象。**

BO接收来自于VO/PO的数据进行逻辑业务操作，转换为相应的PO/DTO。三层中的BLL曾的每一个类实例化之后的对象都属于BO。

**PO(Persistant Object) 持久对象 ：用于表示数据库中的一条记录映射成对象。PO 仅仅用于表示数据，没有任何数据操作。有些地方也称之为MO（**Model Object**）**

var student =

await \_context.Students.AsNoTracking().SingleOrDefaultAsync(m => m.Id == 1);

通过ORM/sql直接读取出来的一条数据映射到某个对象，这个对象为PO

**DAO(Data access object) 数据访问对象 ：用于表示一个数据访问对象。DAO 一般在持久层，完全封装数据库操作，对外暴露的方法使得上层应用不需要关注数据库相关的任何信息。EF中的db。**

顺序：DAO.PO.DTO.BO.VO

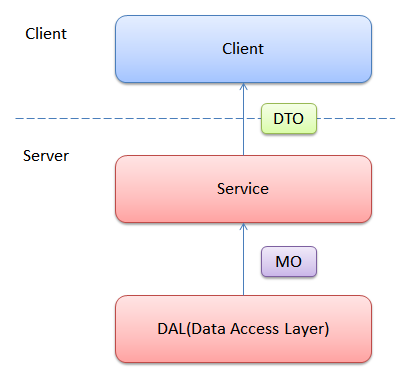
DAO，PO必须存在，如果遇到展示不一致情况下添加DTO,当DTO达到一定数量时添加专门的BO负责逻辑操作（数据验证等），当控制器的接受与传递数据出现不一致时考虑VO

# 具体思路

问题:用于在用户页面上展示的数据和从数据库中取得的数据常常具有较大区别。在这种情况下，我们常常需要向服务端发送多个请求才能将用于在页面中展示的数据凑齐。

一个解决该问题的方法就是根据不同需求使用不同的数据表现形式。在一个服务实现中较为常见的数据表现形式有MO（Model Object，在有些上下文中也被称为PO）和DTO（Data Transfer Object）。MO用来表示从数据库中读取的数据，而DTO则用来表示在网络上所传输的数据。

定义：一类用来访问数据库，在系统中表现数据库中所记录的数据，叫MO，即Model Object；另一类用来在网络中传输，叫DTO，即Data Transfer Object。



使用步骤：

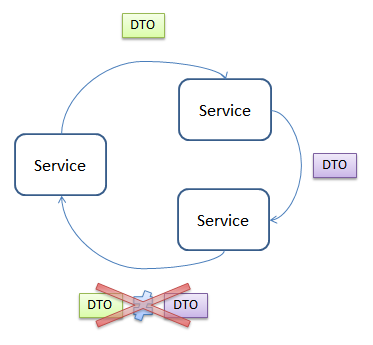
在系统中添加DTO主要有以下几部分工作需要完成：

1. 添加DTO类。
2. 添加从MO到DTO的转化逻辑。
3. 将原本对MO的使用转换为对DTO的使用。

很多插件和框架可以完成 2 3步

优点：尽可能地在向服务端减少发送的请求的情况下拿到完整数据

缺点：一个DTO在依次流转过多个服务的情况。如果在DTO依次传递的过程中使用了不同的DTO表示，那么一个服务所需要的DTO可能和另一个服务中所拥有的DTO并不匹配。这便是DTO反过来会影响到架构设计的一个最简单的例子，却也是DTO管理中最常见的问题，那就是DTO的表现形式过多。如果为所有的不同需求都创建一个DTO，那么一个概念所对应的DTO可能多达5，6种，非常难于管理。这种管理上的困难常常存在于如何指定某个服务所需要使用的DTO种类，以及在更改DTO时需要同时修改一系列DTO的情况中。



解决方案：为了防止DTO由于不同的需求而衍生出过多的种类，服务实现中常常允许DTO中的数据包含一些冗余。

缺点2：DTO中只包含数据，并没有包含任何行为。在服务端的业务逻辑实现以及客户端的页面逻辑中，我们有时需要指定对这些数据的操作逻辑。比如检查年龄是否为负数

解决方案：为了解决这个问题，我们可以在服务端添加一个业务逻辑表现，即BO（Business Object）。在这种情况下，MO将不会直接转化为DTO，而是转化为BO。在所有业务处理完毕并需要将数据发送给客户的时候，BO将转化为DTO以进行传输。