[DTO 1](#_Toc247914443)

[VO 3](#_Toc247914444)

[BO 4](#_Toc247914445)

[POJO 5](#_Toc247914446)

[DAO 6](#_Toc247914447)

# DTO

　　Data Transfer Object（数据传输对象）

　　您正在设计一个分布式应用程序，为了满足单个客户端请求，您发现自己对一个远程接口发出了多个调用，而这些调用所增加的响应时间超出了可接受的程度。

　　问题

　　如何保留过程调用接口的简单语义，而不受远程通信固有的滞后时间问题的影响？

　　影响因素

　　在与远程对象通信时，请考虑下列需要权衡的因素：

　　远程调用（那些必须跨越网络的调用）速度缓慢。虽然许多远程调用框架可以隐藏进行远程调用的复杂性，但是它们不能消除发生通信所需的步骤。例如，必须先找到远程对象位置，而且建立与远程计算机的连接，然后才能将数据串行化为字节流，然后可能进行加密，最后才能将其传输到远程计算机。

　　在考虑网络性能时，必须同时考虑滞后时间和吞吐量。简单地说，"滞后时间"描述了数据的首字节到达目的地之前所经过的时间。"吞吐量"描述了在某个时间段（例如 1 秒）内通过网络发送的数据字节数。在基于 IP 路由的现代网络（例如 Internet）中，滞后时间可以是比吞吐量更大的因素。这意味着，传输 10 字节数据所用的时间可能几乎等于传输 1,000 字节数据所用的时间。在使用无连接协议（如 HTTP）时，此效果尤其明显。通常，网络速度越快可以使吞吐量得以增加，但是，要减少滞后时间则会更加困难。

　　在设计对象接口时，好的做法是将大量信息隐藏在对象内，并提供一组细粒度方法来访问和操作该信息。"细粒度"意味着每个方法都应该负责单个的、相当小的和基本的功能单位。此方法简化了编程，并提供了对对象内部的更佳抽象，从而增加了重用的可能性。必须根据以下事实对此进行平衡取舍：使用较细粒度的方法意味着需要调用更多的方法才能执行高级别的任务。通常，在同一进程内调用方法时，这些额外函数调用的开销是可接受的；但是，在跨进程和网络边界调用这些方法时，开销可能变得难以接受。

　　避免远程调用中固有的滞后时间问题的最佳方法是进行更少的调用，并让每个调用传递更多的数据。做到这一点的一种方法是，使用长参数列表来声明远程方法。这样，客户端就可以在单个调用中将更多的信息传递给远程组件。但是，这样做会使针对此接口的编程容易出错，因为程序很可能仅按调用语句中的位置来调用外部方法的参数。例如，如果远程方法接受 10 个字符串参数，则开发人员很容易按错误顺序传递参数。编译器将无法检测到这样的错误。

　　长参数列表无助于从远程调用向客户端返回更多的信息，因为大多数的编程语言将方法调用的返回类型限制为单个参数。而巧合的是，在传输大多数数据时通常需要返回较多信息。例如，许多用户接口传输少量的信息，却希望返回大量结果数据。

　　解决方案

　　创建一个数据传输对象 (DTO)，用该对象包含远程调用所需要的所有数据。修改远程方法签名，以便将 DTO 作为单个参数接受，并将单个 DTO 参数返回给客户端。在调用方应用程序收到 DTO 并将其作为本地对象存储之后，应用程序可以分别对 DTO 发出一系列单独的过程调用，而不会引发远程调用开销。Martin Fowler 在 Patterns of Enterprise Application Architecture [Fowler03] 中对此模式进行了说明。

　　下图显示客户端应用程序如何进行一系列远程调用以检索客户名称的各个元素。

　　图 1:没有 DTO 的远程调用

　　DTO 允许远程对象在单个远程调用中将整个客户名称返回给客户端。在此示例中，这样做将使调用次数从 4 次减为 1 次。客户端进行单个调用，然后在本地与 DTO 交互，而不用进行多次远程调用（见图 2）。

　　图 2:通过使用 DTO 减少调用次数

　　DTO 是一组需要跨进程或网络边界传输的聚合数据的简单容器。它不应该包含业务逻辑，并将其行为限制为诸如内部一致性检查和基本验证之类的活动。注意，不要因实现这些方法而导致 DTO 依赖于任何新类。

　　在设计数据传输对象时，您有两种主要选择：使用一般集合；或使用显式的 getter 和 setter 方法创建自定义对象。

　　一般集合的优点是，只需要一个类，就可以在整个应用程序中满足任何数据传输目的。此外，集合类（例如，简单数组或散列图）内置于几乎所有语言库中，因此您根本不必编写新类的代码。对 DTO 使用集合对象的主要缺点是，客户端必须按位置序号（在简单数组的情况下）或元素名称（在键控集合的情况下）访问集合内的字段。此外，集合存储的是同一类型（通常是最一般的 Object 类型）的项目，这可以导致在编译时无法检测到的微妙但致命的编码错误。

　　如果为每个 DTO 创建自定义类，则可以提供与任何其他对象完全一样的、客户端应用程序可访问的强类型对象，这样的对象可以提供编译时检查，并支持代码编辑器功能（如 Microsoft&reg; IntelliSense&reg; 技术）。主要缺点是，如果应用程序发出许多远程调用，则您最终可能必须编写大量类的代码。

　　许多方法试图将这两种方法的优点结合在一起。第一种方法是代码生成技术，该技术可以生成脱离现有元数据（如可扩展标记语言 (XML) 架构）的自定义 DTO 类的源代码。第二种方法是提供更强大的集合，尽管它是一般的集合，但它将关系和数据类型信息与原始数据存储在一起。Microsoft ADO.NET DataSet 支持这两种方法（请参阅在 .NET 中使用 DataSet 实现 Data Transfer Object）。

　　有了 DTO 类以后，需要用数据填充它。大多数情况下，DTO 内的数据来自多个域对象。因为 DTO 没有行为，因此它不能从域对象提取数据。这是对的，因为如果让 DTO 不知道域对象，您就可以在不同的上下文中重用 DTO。同样，您不希望域对象知道 DTO，因为这可能意味着更改 DTO 将要求更改域逻辑中的代码，这将导致大量维护任务。

　　最佳的解决方案是使用 Assembler 模式 [Fowler03]，该模式可以用业务对象创建 DTO 或者相反。Assembler 是 Mapper 模式的专门实例，在 Patterns of Enterprise Application Architecture [Fowler03] 中也提到过它。

　　图 3:使用 Assembler 将数据加载到 DTO 中

　　Assembler 的关键特征是 DTO 和域对象不相互依赖。这就消除了这两种对象的相互影响。不利方面是 Assembler 同时依赖于 DTO 和域对象。对这些类的任何更改都可能导致必须更改 Assembler 类。

　　示例

　　请参阅在 .NET 中使用 DataSet 实现 Data Transfer Object。

　　测试考虑事项

　　DTO 是简单对象，它不应该包含需要测试的任何业务逻辑。但是，您确实需要测试每个 DTO 的数据聚合。每个 DTO 可能需要测试，也可能不需要，这取决于您的序列化机制。如果序列化是框架的一部分，则只需要测试一个 DTO。如果不是这样，请使用一般的反射机制，这样就不需要测试每个 DTO 的序列化。

　　DTO 还对远程函数的可测试性有好处。通过使远程方法的结果能够在对象实例中使用，可以轻松地将此数据传递到测试模块，或将其与所需结果进行比较。

　　安全考虑事项

　　理想情况下，应该先筛选和验证从不可靠的来源获得的数据（如来自 Web 页的用户输入），然后将其置于 DTO 中。通过这样做，就可以认为 DTO 中的数据是相对安全的，从而简化了将来与 DTO 的交互。

　　接收 DTO 的进程和关联用户的安全凭据也是值得注意的。DTO 通常包含从许多不同来源聚集在一起的大量信息。您是否已授权 DTO 的所有用户访问 DTO 所包含的所有信息？确保用户已得到授权的最佳方法是仅使用用户安全凭据所允许的特定数据填充 DTO。努力避免让 DTO 负责自己的安全性。这将增加 DTO 对其他类的依赖数，这意味着必须将这些类部署到使用 DTO 的所有节点。这还会将安全性功能分散到更多类中，从而增大了安全风险，并对灵活性和可维护性产生负面影响。

　　结果上下文

　　Data Transfer Object 具有下列优缺点：

　　优点

　　减少了远程调用次数。通过在单个远程调用中传输更多的数据，应用程序可以减少远程调用次数。

　　提高了性能。远程调用可以使应用程序的运行速度大大降低。减少调用次数是提高性能的最佳方法之一。在大多数方案中，传输大量数据的远程调用所用的时间与仅传输少量数据的调用所用的时间几乎相等。

　　隐藏内部情况。在单个调用中来回传递更多的数据，还可以更有效地将远程应用程序的内部情况隐藏在粗粒度接口的背后。这就是使用 Remote Facade 模式 [Fowler03] 的主要原因。

　　发现业务对象。在一些情况下，定义 DTO 有助于发现有意义的业务对象。在创建用作 DTO 的自定义类时，您通常会注意到作为一组凝聚性信息而显示给用户或另一个系统的元素分组。通常，这些分组用作描述应用程序所处理的业务域的对象的有用原型。

　　可测试性。将所有参数封装到可序列化对象中可以提高可测试性。例如，可以从 XML 文件中读取 DTO，并调用远程函数以测试它们。同样，可以轻松地将结果再序列化为 XML 格式，并将 XML 文档与所需结果进行比较，而不必创建冗长的比较脚本。

　　缺点

　　可能需要太多的类。如果选择了使用强类型的 DTO，则可能必须为每个远程方法创建一个（如果考虑返回值，则为两个）DTO。即使在粗粒度接口中，这也可能导致大量的类。编写如此数量的类的代码并管理这些类会是很困难的。使用自动代码生成可以在一定程度上缓解此问题。

　　增加计算量。如果将服务器上的一种数据格式转换为可以跨网络传输的字节流，并在客户端应用程序内转换回对象格式，可以带来相当大的开销。通常，需要将来自多个源的数据聚合到服务器上的单个 DTO 中。要提高通过网络进行远程调用的效率，必须在任一端执行其他计算，才能聚合和串行化信息。

　　增加编码工作量。可以用一行代码完成将参数传递到方法的操作。使用 DTO 要求实例化新对象，并为每个参数调用 setters 和 getters。编写此代码可能是很乏味的。

# VO

:value object值对象。

　　通常用于业务层之间的数据传递，和PO一样也是仅仅包含数据而已。但应是抽象出的业务对象,可以和表对应,也可以不,这根据业务的需要.

　　模型驱动：把数据跟处理分开，用单独的JavaBean封装数据（不管是请求的，还是返回的，可以在一起封装），把处理类完全孤立成一个框架，用这个模型作为灵魂在驱动整个骨架。

　　特性：血肉分开，使结构更清晰，只能划分更清楚，免得局长管权利有管财务的，弄不好贪污。

　　核心：模型（model）就是指对象Vo（value object）

　　对比：属性驱动--你把一大堆属性搞到你的处理类里面，完全可以。方便而快捷，不过结构清晰度当然就不好了。

说实话，模型驱动的确挺烦人的，开发效率不高，你看着办吧，具体问题具体对待，真的。

# BO

[[编辑本段](http://baike.baidu.com/view/478349.htm?func=retitle)]

BO业务对象(Business Object，BO)

　　是对数据进行检索和处理的组件。是简单的真实世界的软件抽象。业务对象通常位于中间层。

　　业务对象可以在一个应用中自动的加入一个特定的功能来获得增值效应。使知识重用变为可能。比如，如果你要开发一个包含多货币处理的应用，你可以选择使用一个已经开发完成的，包含所有多货币处理功能的业务对象来开始你的开发。使您的开发工作极大的减少。

　　一、业务对象的必要条件

　　1.由状态和行为组成

　　2.表达了来自业务域的一个人，地点，事物或概念

　　3.可以重用

　　二、业务对象的分类

　　1、实体业务对象:

　　表达了一个人，地点，事物或者概念。根据业务中的名词从业务域中提取。如客户，订单，物品。

　　2、过程业务对象:

　　表达应用程序中业务处理过程或者工作流程任务。通常依赖于实体业务对象，是业务的动词。作为应用程序的管理者或者控制者。

　　3、事件业务对象:

　　表达应用程序中由于系统的一些操作造成或产生的一些事件。

　　三、业务对象的抽象与整合

　　业务对象的抽象和整合有何联系呢，或许有人会问我这样的一个问题（以前的我也老想着这个问题），就这个问题我个人觉得,如果脱离业务抽象而想象一个架构体系，那么是一个本末倒置的愚蠢的做法，因为只有做了一个业务的抽象才能根本上满足需求本质，这样才能更实际的充分的得到现实业务现象的抽象才能合理有效的模拟实现的IT系统（一个IT化的过程第一步骤）。

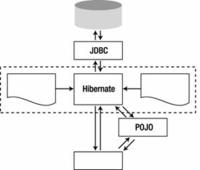
　　整合IT系统面临一个很大问题如何抽象IT系统的交互问题，这个方面IBM采取了消息通信的抽象；它这样做当然有他的道理（也是比较接近现实场景的），但是我在这里顺便提一下我个人的想法，消息其实只是通信和协调的一个实现而已，但是还没有到本质；本质就是通信的协议的定制。我自己采取的就是在底层使用一个会话协议抽象（工作的保密关系不能再细说了，但是我的实践告诉我这样做有很高的架构体系扩展上，大家有机会可以试一试）。

　　所以业务抽象十分重要，只有把握好这一点，你的架构系统将体现更高的架构体系高度。你会发现需求的现象的本质，已经没有太多的需求变动能破坏你的架构----除非业务的本质改变了（但这却仅仅说明我们还没有把握业务的本质）。

　　另一种解释，用于淘汰赛上的赛制。BO=best of，比如BO1,单淘汰制;BO3，三局两胜制;BO5，五局三胜制。

　　还是：由像素软件开发，搜狐运营的网络游戏“刀剑 英雄”的缩写

# POJO

[](http://imgsrc.baidu.com/baike/pic/item/7d98931053392d15203f2e41.jpg)

在Java应用程序中的角色

　　简单的Java对象（Plain Ordinary Java Objects）实际就是普通JavaBeans,使用POJO名称是为了避免和EJB混淆起来, 而且简称比较直接. 其中有一些属性及其getter setter方法的类,有时可以作为value object或dto(Data Transform Object)来使用.当然,如果你有一个简单的运算属性也是可以的,但不允许有业务方法,也不能携带有connection之类的方法。

　　POJO是Plain Ordinary Java Objects的缩写不错，但是它通指没有使用Entity Beans的普通java对象，可以把POJO作为支持业务逻辑的协助类。

　　POJO有一些private的参数作为对象的属性。然后针对每个参数定义了get和set方法作为访问的接口。例如：

　　public class User {

　　private long id;

　　private String name;

　　public void setId(long id) {

　　this.id = id;

　　}

　　public void setName(String name) {

　　this.name=name;

　　}

　　public long getId() {

　　return id;

　　}

　　public String getName() {

　　return name;

　　}

　　}

　　POJO对象有时也被称为Data对象，大量应用于表现现实中的对象。

# DAO

[](http://imgsrc.baidu.com/baike/pic/item/9dc3cf58ae3b5e92800a18bc.jpg)

DAO是Data Access Object数据访问接口，数据访问：顾名思义就是与数据库打交道。夹在业务逻辑与数据库资源中间。

　　在核心J2EE模式中是这样介绍DAO模式的：为了建立一个健壮的J2EE应用，应该将所有对数据源的访问操作抽象封装在一个公共API中。用程序设计的语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储。

　　DAO（数据访问对象）是一种应用程序编程接口（API），存在于微软的Visual Basic中，它允许程序员请求对微软的Access数据库的访问。DAO是微软的第一个面向对象的数据库接口。DAO对象封闭了Access的Jet函数。通过Jet函数，它还可以访问其他的结构化查询语言（SQL）数据库。

　　除去Data Access Object的意思，在光盘刻录方面DAO对应于Disk At Once（一次写盘方式），在使用刻录机刻录镜像文件时可以选择这种方式。

　　一次写盘是单次的写入方式，引导区、数据磁道以及导出区都是一次性写入，一次写完之后光盘就关闭，即便此次写入没有写满整个刻录盘，也无法再写入其它数据。当引导区写入到光盘上时，并没有在该引导区标示出下一个可用的地址，因此光盘就被视为关闭，再也无法写入更多的数据。

　　这种写入模式主要用于光盘的复制，一次完成整张光盘的刻录。其特点是能使复制出来的光盘与源盘毫无二致。DAO写入方式可以轻松完成对于音乐 CD、混合或特殊类型CD-ROM等数据轨之间存在间隙的光盘的复制，且可以确保数据结构与间隙长度都完全相同。值得一提的是，由于DAO写入方式把整张光盘当作一个区段来处理，一些小的失误都有可能导致整张光盘彻底报废，所以它对数据传送的稳定性和驱动器的性能有较高的要求。

　　在Java Web应用中使用DAO，可以将底层数据访问逻辑与业务逻辑分离开来。可以构建为不同数据源提供CRUD操作的DAO类！

PO：  
persistant object持久对象  
  
最形象的理解就是一个PO就是数据库中的一条记录。  
好处是可以把一条记录作为一个对象处理，可以方便的转为其它对象。

BO：  
business object业务对象  
  
主要作用是把业务逻辑封装为一个对象。这个对象可以包括一个或多个其它的对象。  
比如一个简历，有教育经历、工作经历、社会关系等等。  
我们可以把教育经历对应一个PO，工作经历对应一个PO，社会关系对应一个PO。  
建立一个对应简历的BO对象处理简历，每个BO包含这些PO。  
这样处理业务逻辑时，我们就可以针对BO去处理。

VO ：  
value object值对象  
ViewObject表现层对象  
  
主要对应界面显示的数据对象。对于一个WEB页面，或者SWT、SWING的一个界面，用一个VO对象对应整个界面的值。

DTO ：  
Data Transfer Object数据传输对象  
主要用于远程调用等需要大量传输对象的地方。  
比如我们一张表有100个字段，那么对应的PO就有100个属性。  
但是我们界面上只要显示10个字段，  
客户端用WEB service来获取数据，没有必要把整个PO对象传递到客户端，  
这时我们就可以用只有这10个属性的DTO来传递结果到客户端，这样也不会暴露服务端表结构.到达客户端以后，如果用这个对象来对应界面显示，那此时它的身份就转为VO

POJO ：  
plain ordinary java object 简单java对象  
个人感觉POJO是最常见最多变的对象，是一个中间对象，也是我们最常打交道的对象。  
  
一个POJO持久化以后就是PO  
直接用它传递、传递过程中就是DTO  
直接用来对应表示层就是VO

DAO：  
data access object数据访问对象  
这个大家最熟悉，和上面几个O区别最大，基本没有互相转化的可能性和必要.  
主要用来封装对数据库的访问。通过它可以把POJO持久化为PO，用PO组装出来VO、DTO  
  
  
       总结下我认为一个对象究竟是什么O要看具体环境，在不同的层、不同的应用场合，对象的身份也不一样，而且对象身份的转化也是很自然的。就像你对老婆来说就是老公，对父母来说就是子女。设计这些概念的初衷不是为了唬人而是为了更好的理解和处理各种逻辑，让大家能更好的去用面向对象的方式处理问题.  
  
       大家千万不要陷入过度设计，大可不必为了设计而设计一定要在代码中区分各个对象。一句话技术是为应用服务的。  
  
欢迎指正。

