PO、VO、BO、DTO、POJO、DAO之间的关系

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ava对象
POJO是最参见最多变的对象，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：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.

POJO = pure old java object or plain ordinary java object or what ever.
PO = persisent object 持久对象
就是说在一些Object/Relation Mapping工具中，能够做到维护数据库表记录的persisent object完全是一个符合Java Bean规范的纯Java对象，没有增加别的属性和方法。全都是这样子的：

```
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;
        }
}
```

首先要区别持久对象和POJO。
持久对象实际上必须对应数据库中的entity，所以和POJO有所区别。比如说POJO是由new创建，由GC回收。但是持久对象是insert数据库创建，由数据库delete删除的。基本上持久对象生命周期和数据库密切相关。另外持久对象往往只能存在一个数据库Connection之中，Connnection关闭以后，持久对象就不存在了，而POJO只要不被GC回收，总是存在的。
由于存在诸多差别，因此持久对象PO(Persistent Object)在代码上肯定和POJO不同，起码PO相对于POJO会增加一些用来管理数据库entity状态的属性和方法。而ORM追求的目标就是要PO在使用上尽量和POJO一致，对于程序员来说，他们可以把PO当做POJO来用，而感觉不到PO的存在。

JDO的实现方法是这样的：
1、编写POJO
2、编译POJO
3、使用JDO的一个专门工具，叫做Enhancer，一般是一个命令行程序，手工运行，或者在ant脚本里面运行，对POJO的class文件处理一下，把POJO替换成同名的PO。
4、在运行期运行的实际上是PO，而不是POJO。

该方法有点类似于JSP，JSP也是在编译期被转换成Servlet来运行的，在运行期实际上运行的是Servlet，而不是JSP。

Hibernate的实现方法比较先进：
1、编写POJO
2、编译POJO
3、直接运行，在运行期，由Hibernate的CGLIB动态把POJO转换为PO。
由此可以看出Hibernate是在运行期把POJO的字节码转换为PO的，而JDO是在编译期转换的。一般认为JDO的方式效率会稍高，毕竟是编译期转换嘛。但是Hibernate的作者Gavin King说CGLIB的效率非常之高，运行期的PO的字节码生成速度非常之快，效率损失几乎可以忽略不计。
实际上运行期生成PO的好处非常大，这样对于程序员来说，是无法接触到PO的，PO对他们来说完全透明。可以更加自由的以POJO的概念操纵PO。另外由于是运行期生成PO，所以可以支持增量编译，增量调试。而JDO则无法做到这一点。实际上已经有很多人在抱怨JDO的编译期Enhancer问题了，而据说JBossDO将采用运行期生成PO字节码，而不采用编译期生成PO字节码。
另外一个相关的问题是，不同的JDO产品的Enhancer生成的PO字节码可能会有所不同，可能会影响在JDO产品之间的可移植性，这一点有点类似EJB的可移植性难题。

由这个问题另外引出一个JDO的缺陷。
由于JDO的PO状态管理方式，所以当你在程序里面get/set的时候，实际上不是从PO的实例中取values，而是从JDO StateManager中取出来，所以一旦PM关闭，PO就不能进行存取了。
在JDO中，也可以通过一些办法使得PO可以在PM外面使用，比如说定义PO是transient的，但是该PO在PM关闭后就没有PO identity了。无法进行跨PM的状态管理。
而Hibernate是从PO实例中取values的，所以即使Session关闭，也一样可以get/set，可以进行跨Session的状态管理。

在分多层的应用中，由于持久层和业务层和web层都是分开的，此时Hibernate的PO完全可以当做一个POJO来用，也就是当做一个VO，在各层间自由传递，而不用去管Session是开还是关。如果你把这个POJO序列化的话，甚至可以用在分布式环境中。（不适合lazy loading的情况）

但是JDO的PO在PM关闭后就不能再用了，所以必须在PM关闭前把PO拷贝一份VO，把VO传递给业务层和web层使用。在非分布式环境中，也可以使用ThreadLocal模式确保PM始终是打开状态，来避免每次必须进行PO到VO的拷贝操作。但是不管怎么说，这总是权宜之计，不如Hibernate的功能强。