## 设计概念的统一语言

毫无疑问，领域驱动设计引入了一套自成体系的设计概念：限界上下文、应用服务、领域服务、聚合、实体、值对象、领域事件以及资源库和工厂。这些设计概念又与其它设计方法的设计概念互为参考和引用，再糅合不同团队不同企业不同领域的设计实践，就产生了更多的设计概念。诸多概念纠缠不清，说法不一，理解不同，就会形成认知上的混乱，干扰整个团队对领域驱动设计的理解。既然领域驱动设计强调为领域逻辑建立统一语言，我们不妨也为这些设计概念定义一套“统一语言”，通过理解的一致保证交流的畅通，确保架构和设计方案的统一性。

### 设计术语的统一

当我们在讨论领域驱动设计时，不止要谈到领域驱动设计固有的设计概念，结合开发语言和开发平台的设计实践，又会有其他设计概念穿插其中，它们之间的关系并非正交的，解决的问题和思考的角度都不太一致，许多设计概念更有其历史渊源，却又在提出之后或者被滥用，或者被错用，到了最后已经失去了它本来的面目。因此，我们需要揭开这些设计术语的历史迷雾，理解其本真的概念，然后再确定它的统一语言。

#### POJO对象

[POJO（Plain Old Java Object）](https://www.martinfowler.com/bliki/POJO.html)的概念来自 Martin Fowler、Rebecca Parsons 和 Josh MacKenzie 在 2000 年一次大会的讨论。它的本质含义是指一个常规的 Java 对象，不受任何框架、平台的约束和限制。除了遵守 Java 语法之外，它不应该继承预先设定的类、实现预先设定的接口或者包含预先指定的注解。可以认为，如果一个模块定义的对象皆为 POJO，那么除了依赖 JDK 之外，它不会依赖任何框架或平台。在 .NET 框架中，借助这个概念，也提出了 POCO（Plain Old CLR Object）的概念。

Martin Fowler 等人之所以提出 POJO，是因为他们看到了 **使用 POJO 封装业务逻辑的益处** ，而在 2000 年那个时代，恰恰是 EJB 开始流行的时代，受到 EJB 规范的限制，Java 开发人员更愿意使用 Entity Bean，而 Entity Bean 却是与 EJB 强耦合的。

一些人错误地将 Entity Bean 理解为仅具有持久化能力的 Java 对象，实际并非如此。即使 EJB 规范也认为 Entity Bean 可以包含复杂的业务逻辑，例如 [Oracle 的官方网站](https://docs.oracle.com/cd/A91202_01/901_doc/java.901/a90188/entity.htm#1028116)对 Entity Bean 的定义就包括：

* 管理持久化数据
* 通过主键形成唯一标识
* 引入依赖对象执行复杂逻辑

当 Entity Bean 还封装了复杂的业务逻辑时，带来的危害更多。由于定义一个 Entity Bean 类需要继承自 javax.ejb.EntityBean（基于 EJB 3.0 之前的规范），这就使得业务逻辑与 EJB 框架紧耦合，不利于对业务逻辑的测试、部署与运行。这也正是 Rod Johnson 要提出抛开 EJB 进行 J2EE 开发的原因。当然，Entity Bean 与 EJB 框架紧耦合的为人诟病，主要是针对 EJB 3.0 之前的版本，随着 Spring 与 Hibernate 等轻量级框架出来之后，EJB 也开始向轻量级方向发展，通过大量使用标注来降低 EJB 对 Java 类的侵入性。

既然 Entity Bean 也可以封装业务逻辑，针对它提出的 POJO 自然也可以封装业务逻辑。如前所述，Martin Fowler 等人看到的是“使用 POJO 封装业务逻辑的益处”，这就说明 **POJO 对象并非只有 getter/setter 的贫血对象** ，它的主要特征不在于它究竟定义了什么样的成员，而在于它作为一个常规的 Java 对象，并不依赖于除语言之外的任何框架。当然，它的目的不在于数据传输，也不在于数据持久化，它其实是一种设计模式。

#### Java Bean

严格讲来，Java Bean 其实是一种 Java 开发规范。一个 Java Bean 类必须同时满足以下三个条件：

* 类必须是具体的、公共的
* 具有无参构造函数
* 提供一致性设计模式的公共方法将内部字段暴露为成员属性，即为内部字段提供规范的 get 和 set 方法

认真解读这三个条件，你会发现它们都是为支持反射访问类成员而准备的前置条件，包括创建 Java Bean 实例和操作内部字段。只要遵循 Java Bean 规范，就可以采用完全统一的一套代码实现对 Java Bean 的访问。这一规范并没有提及业务方法的定义，这是因为规范无法对公开的方法做出任何一致性的限制。这意味着框架使用 Java Bean，看重的其实是该对象携带的数据，并能通过反射访问对象的字段值。例如，JSP 对 Java Bean 的使用：

<jsp:useBean id="student" class="com.dddsample.javabeans.Student">   
 <jsp:setProperty name="student" property="firstName" value="Bill"/>  
 <jsp:setProperty name="student" property="lastName" value="Gates"/>  
 <jsp:setProperty name="student" property="age" value="20"/>  
</jsp:useBean>

JSP 标签中使用的 Student 类就是一个 Java Bean。如果该类的定义没有遵循 Java Bean 规范，JSP 就可能无法实例化 Student 对象，无法设置 firstName 等字段值。

至于 Session Bean、Entity Bean 和 Message Driven Bean 则是 Enterprise Java Bean 的三个分类，它们都是 Java Bean，但 EJB 对它们又有框架的约束，例如 Session Bean 需要继承自 javax.ejb.SessionBean，Entity Bean 需要继承自 javax.ejb.EntityBean。

追本溯源，可发现 **POJO 与 Java Bean 并没有任何关系** 。一个 POJO 如果遵循了 Java Bean 的设计规范，可以成为一个 Java Bean，但并不意味着 POJO 一定是 Java Bean。反过来，一个 Java Bean 如果没有依赖任何框架，也可以认为是一个 POJO。但是，Enterprise Java Bean 一定不是一个 POJO。POJO 可以封装业务逻辑，Java Bean 也没有限制它不能封装业务逻辑。一个提供了丰富领域逻辑的 Java 对象，如果它同时又遵循了 Java Bean 的设计规范，也可以认为是一个 Java Bean。

#### 贫血模型

贫血模型准确地说，应该被称之为“[贫血领域模型（Anemic Domain Model）](https://www.martinfowler.com/bliki/AnemicDomainModel.html)”，因为该术语主要用于领域模型这个语境，来自 Martin Fowler 的创造。从贫血这个词可知，这样的一种领域模型必然是不健康的，它违背了面向对象设计的关键原则，即“数据与行为应该封装在一起”。在领域驱动设计中，如果一个实体或值对象除了内部字段之外只有一系列的 getter/setter 方法，就成为了贫血对象。

关于贫血领域模型的坏处，我在本书已经阐述了很多，例如它会影响对象之间的协作方式，它违背了“迪米特法则”与“信息专家模式”，它会导致“特性依恋”坏味道的产生，最后，它其实破坏了封装，导致领域服务形成一种事务脚本的实现。

与贫血领域模型相对的是富领域模型（Rich Domain Model），也就是封装了领域逻辑的领域模型。这样的领域模型才符合面向对象设计思想。当我们采用对象范式进行领域设计建模时，实体与值对象都应该建立为富领域模型。富领域模型就是 Martin Fowler 在《企业应用架构模式》中定义的领域模型模式。作为一种[领域逻辑模式（Domain Logic Pattern）](https://martinfowler.com/eaaCatalog/index.html)，它与事务脚本（Transaction Script）、表模块（Table Module）属于不同的表达领域逻辑的模式。倘若遵循这一模式的定义，即默认为 **领域模型就应该是富领域模型** ，而贫血领域模型会导致事务脚本，本不应该将这样的模型称为领域模型。

当我们在讨论领域模型时，发现更有好事者在贫血模型的基础上衍生出各种与“血”有关的各种模型，统计下来，除了 Martin Fowler 提出的贫血模型之外，还包括失血模型、充血模型与胀血模型。我将这些模型戏称为“X 血模型”。我个人并不赞成社区制造出这么多的模型，太多的概念反而为造成概念的混乱不清。实际上，造成贫血模型的原因是不恰当的职责分配。如果我们能够按照合理的职责分配原则来设计领域模型，就无所谓这些 X 血模型了。只要职责分配合理，有可能领域模型中的一个类确乎没有定义具有领域逻辑的行为，那也只能说明该领域概念确实不具有领域逻辑，不应当称之为贫血对象。

我还看到[有的人错误的理解或者误用了“贫血模型”的定义](https://blog.csdn.net/Dancen/article/details/7270784)，将只有字段和 getter/setter 方法的类称之为“失血模型”，而将 Martin Fowler 提出的富领域模型称之为“贫血模型”。这一说法绝对是“谬种流传”。顾名思义，贫血（Anemic）这个词代表着不健康，贫血模型当然就意指不健康的模型。如果采用这篇文章的定义，Martin Fowler 所推崇的富领域模型反倒成了不健康的贫血模型（虽然该文作者未必认为贫血模型不健康），该何其无辜啊！因此，在这些“X 血模型”中，我们必须坚定果断地去掉失血模型，并让贫血模型回到本初的含义。

去掉错误的失血模型，一般认为充血模型其实就是 Martin Fowler 提出的富领域模型。我总觉得“充血”这个词仍然带有不健康的隐含意义，故而不愿意使用这一模式名称，更不用说更加惊悚的“胀血模型”了。这种胀血模型违背了单一职责原则，将与该领域概念相关的所有逻辑都放到了领域模型对象中，包括对数据访问对象或资源库的依赖以及对事务、授权等横切关注点的调用。在领域驱动设计的语境中，这相当于让一个实体类承担了聚合、领域服务以及应用服务的职责，明显有悖于领域驱动设计乃至面向对象的设计原则。

有的观点认为混入了持久化能力的领域模型属于充血模型，这更进一步混淆了 X 血模型的边界。实际上，Martin Fowler 将这种对象称之为“[活动记录（Active Record）](https://martinfowler.com/eaaCatalog/activeRecord.html)”，它属于数据源架构模式（Data Source Architectural Patterns）中的一种。这种设计方式是领域驱动设计努力避免的，如果每个实体都混入了持久化能力，就会丢失聚合的边界保护作用，资源库也就失去了存在的价值。

还有人混淆了领域模型与 POJO 的概念，认为贫血模型对象就是一个 POJO，殊不知这二者根本就是两个迥然不同的维度。POJO 关注类的定义是否纯粹，领域模型关注对领域逻辑的表达与封装。即使是一个只有 getter/setter 方法的贫血模型对象，只要它依赖了任何外部框架，例如标记了 javax.persistence.Entity 标注，它也不属于一个 POJO。事实上，Dubbo 服务化最佳实践给出的建议——“服务参数及返回值建议使用 POJO 对象，即通过 setter、getter 方法表示属性的对象”，对 POJO 的描述是不正确的，因为 Dubbo 服务的输入参数与返回值需要支持序列化，不符合 POJO 的定义。

由此可以看出，定义种类繁多的模式会让人“乱花渐欲迷人眼”，随着信息的多次传递，就会迷失它们本来的面目。我们需要做减法，在领域驱动战术设计中，只要遵循领域驱动设计的原则定义了实体、值对象、领域服务和应用服务，就不用考虑这些模式，只需把握一条：避免设计出贫血领域模型！

#### 诸多 XO

在分层架构的约束下，在职责分离的指引下，一个软件系统需要定义各种各样的对象，在分层架构中承担了不同的职责，又彼此协作，共同响应系统外部的各种请求，执行业务逻辑，并让整个软件系统能够真正地跑起来。然而，若没有真正理解这些对象在架构中扮演的角色和承担的职责，就会导致误用和滥用，效果反而适得其反。因此，有必要在领域驱动设计的方法体系下，将各式各样的对象进行一次梳理，形成一套统一语言。由于这些对象皆以 O 结尾，故而戏称为 XO 对象。

这些 XO 对象包括：

* DTO（Data Transfer Object，数据传输对象）：DTO 用于进程间传递数据，远程服务接口的输入参数与返回值都可以认为是一个 DTO。我个人又根据调用者的不同，将其分为视图模型对象与消息契约对象。DTO 必须支持序列化，同时它通常应该设计为一个 Java Bean，即定义为公开的类，具有默认构造函数和 getter/setter 方法。这样就有利于一些框架通过反射来创建与组装 DTO 对象。DTO 还应该是一个贫血对象，因为它的目的是为了传输数据，没有必要定义封装逻辑的方法。
* VO（View Object，视图对象）：视图对象其实是 DTO 的一种，即我提到的视图模型对象。它遵循 MVC 模式，为前端视图提供数据，即 MVC 中的模型对象。视图对象可能仅传输视图需要呈现的数据，也可能为了满足前端UI的可配置，由后端传递与视图元素相关的属性值，如视图元素的位置、大小乃至颜色等样式信息。在微服务架构下，往往由 BFF（Backend For Frontend）层的控制器操作这样的 VO。注意，有人将领域驱动设计中的值对象（Value Object）也简称为 VO，注意不要混淆这两个概念。
* BO（Business Object，业务对象）：这是一个非常宽泛的定义，在软件系统中，它的定义来自于分层架构的定义，即数据访问层、业务逻辑层与 UI 呈现层，BO 属于定义在业务逻辑层中封装了业务逻辑的对象。在领域驱动设计中，可以认为就是领域层的领域对象。为避免混淆，我建议不要在领域驱动设计中使用该概念。
* DO（Domain Object，领域对象）：领域驱动设计将业务逻辑层分解为应用层和领域层，业务对象在领域层中就变成了领域对象。在领域驱动设计中，更准确的说法是领域模型对象。通常，领域模型对象包括实体、值对象、领域服务与领域事件。有时候，领域模型对象单指组成聚合的实体与值对象。宽泛地讲，只要表达了现实世界的领域概念，或者封装了领域行为逻辑，都可以认为是领域模型对象。
* PO（Persistence Object，持久化对象）：对象字段持有的数据需要被持久化到数据表中，但不要由此认为 PO 就只能定义字段以及对应的 getter/setter 方法，这回让 PO 变成一个贫血对象。前面讨论的富领域模型对象也可以成为 PO，二者并不矛盾。只要领域模型对象持有数据，就可以持久化，它拥有的领域行为方法并不影响持久化的实现。由于需要配置持久化框架对象与关系表之间的映射，往往需要以某种形式在 PO 中展现这种映射关系，由于这种映射关系是 ORM 框架定义的，因此一个 PO 往往不是一个 POJO。
* DAO（Data Access Object，数据访问对象）：DAO 对 PO 进行持久化，实现对数据的访问。由于领域驱动设计引入了聚合边界，并力求领域模型与数据模型之间的分离，且引入了资源库（Repository）来实现对聚合的生命周期管理，因此在领域驱动设计中，不再使用 DAO 这个概念。

通过对以上概念的历史追寻与本质分析，我们基本上理清了这些概念的含义与用途。在归纳到领域驱动设计这个方法体系中，我们可以得出如下统一语言：

* 领域模型对象包含实体、值对象、领域服务与领域事件，有时候也可以单指组成聚合的实体与值对象。
* 领域模型必须是富领域模型。
* 远程服务与应用服务接口的输入参数和返回值定义为 DTO，根据客户端的不同，可以分为视图模型对象与消息契约对象。
* 领域模型对象中的实体与值对象同时也可以作为 PO。
* 只有资源库对象，没有 DAO 对象。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持