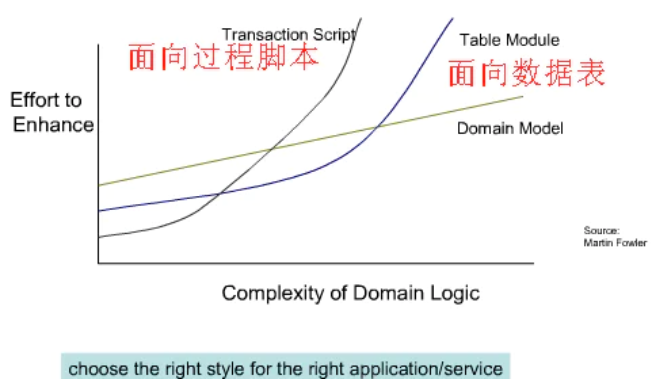
<https://juejin.im/post/5ca16166e51d456708675612>

# **为什么我们需要领域驱动设计**



研究领域驱动设计（后面简称DDD）有半年之多，初识DDD是因为了解何为充血模式，何为贫血模式，进而顺蔓摸瓜触及DDD，初次了解有种相见恨晚的感觉，为什么到现在才了解到有DDD这么个东西，之后，一个伴随我成长的疑惑，在我成长过程中不断致力于去解决，终于在DDD帮助下云拨雾散。   我的疑惑，第一点就是对于我们采用了不知道多久的开发模式，即Controller->Service->Dao，**Service层扮演者上帝类，所有的逻辑都往里塞**，这种模式造成Service层的无限膨胀真的是对的吗，很显然答案是不对的，但事实上很多人对此还是无动于衷，但是究竟要如何改进？以前在大学的时候，自己一直在给自己找外包项目做，曾经对自己放下豪言壮语，多复杂的系统，只要一次性过不要求对其进行维护以及加和改需求，我都能做出来，因为这种开发模式，确实是将CURD表现淋漓尽致，一个资深工程师的代码实现结果，一个菜鸟也同样做到（这里说的是结果，不是过程，然而实际是过程也可能一样，这就很难受了）。第二点，其实也是延续着第一点而来，随着经验的丰富，开始领悟到面向对象和软件工程，那么问题来了，**在这种传统的开发模式下，面向对象究竟体现在哪里，我们学到的设计模式，究竟在哪里展开实践**，而实际上我们往往是挂羊头卖狗肉，羊头是面向对象，狗肉是面向过程。

  下面我将一点点的引导大家进入DDD的思维模式中，将我们习惯的开发方式面向过程开发与DDD所倡导的面向模型开发进行对比，逐步进入面向对象。注意，本篇仅仅告诉大家何为DDD，以及DDD的好处和缺点，具体如何实践将在后面展开探讨。DDD提出了一个所谓的**领域建模**，其实本质是面向对象开发原则中的类的单一职责，我们的第一个问题是**如何处置承担过多的职责的Service上帝类**。一个字很好解决，就是拆，笔者曾经尝试按着类的单一职责的指导，将一个依赖注入过多其他类的一个Service类，可能有十几二十个，读者你可以检查一下自己的项目，这就是坏代码，这带来的好处其实只有功能点封装，并不能解决不让Service类臃肿的问题，他还是会慢慢臃肿起来的，而本质仍是面向过程开发，还有一个问题是，就算是这种做法解决Service类臃肿的问题，他无法解决一个很关键的东西，即这里面的代码，没有体现出业务性。什么叫做没有体现出业务性？比如说激活用户，我们一般会怎么开发？

public class UserService{

public void updateAvailableById(long id,boolean available){

userDao.updateAvailableById(id,available);

}

}复制代码

  这是很典型的面向过程开发的代码，在我们的项目中处处可见，丧失了面向对象中对象的概念，而现如今我们在开发的时候，反而被因为使用了数据库而束缚了我们的思维，我们在接收到需求的第一时刻，在分析设计阶段，往往是以数据表为核心进行分析设计， 也就是根据需求首先得到数据表名和字段，然后培训程序员学会SQL语句如何操作这些数据表，那么程序员为实现数据表的前后顺序操作，必然会将代码写成过程式的风格，**以数据表为核心进行分析设计，代码很难避免不演变成过程式的代码，因为我们的重点放在了操作某张表，以及相关的某个字段**。   这里就是所谓的**贫血模型**了，UserDao是关于用户表的SQL语句的集合，在项目中的表现就是写了一堆SQL语句，UserService则是作为Transaction Script的入口以及夹杂着其他杂七杂八的Service类。而这一整个过程中，user对象都没出现过。 而领域驱动倡导的做法如下。

@Data

public class User{

private Long id;

private Boolean available;

public void activate(){

this.setAvailable(true);

}

}

//领域驱动服务

@Service

public class UserOperationService{

@Autowired

private UserRepository userRepository;

public void activate(User user){

user.activate();

userRepository.save(user);

}

}复制代码

  首先第一点是**类富含行为（即充血模式）**，这才是真正倡导的面向对象开发，业务系统所关注的点是业务功能而不是想着这个功能我们应该如何建表，如何写SQL语句，一直以来我们的思维都因为使用了数据库而被绑架了，一上来就开始建表写代码，代码写的非常冗余，完全是过程式的思考方式，最后导致系统非常难以维护。业务功能只需要明白是激活功能，而不是把user表中available设为true，数据库只是基础设施，我们不应该关注以及知会对应的SQL语句是怎么样的，所以DDD强调的是Repository而不是DAO，Repository翻译为仓库资源，所需即所得，不用去关注其背后，而DAO我们不得不去关心SQL语句如何写。可以前往阅读这篇文章进入更深入的思考[对象和数据库的天然阻抗](https://www.jdon.com/mda/oo-reltaion2.html" \t "_blank)，如果看完之后还不是很明白的话可以在留言区留下疑惑。

  我们不应将原属于领域模型的行为方法等划放在服务中实现，对象不但有属性还有行为将数据和行为封装在一起，并与现实世界的业务对象相映射。回归到我们最初刚学习面向对象教我们的那样，万物皆是类，而不是只有一个上帝类，我们需要多创建有合适逻辑的类，各类具备明确的职责划分，使得逻辑分散到合适对象中。这样的对象就是“充血模型”，而这个过程我们也称之为领域建模，这才是面向对象的真谛啊！

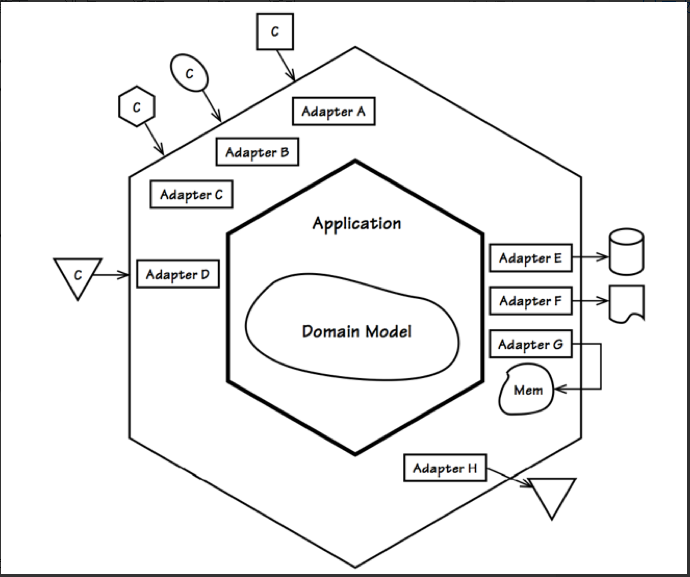
  说到这里，大家可能对以上的示例存在着一点疑惑，1.“怎么这么简单的功能，用DDD的方式下却整的这么复杂？”，2.“你说的不就是要使用ORM框架hibernate吗”。3.“但是一切说到底不都是需要对数据的增删查改吗”

  第一点，DDD为什么称之为复杂软件的设计之道？因为系统够复杂，才能全面展现DDD的功效，业务逻辑站在至高点使得任何业务逻辑的修改扩展都能做到兵来将挡水来土掩，极大的降低了刚接触该系统的同事上手入门的门槛，以及更好的避免存在隐藏BUG（你总会因为不小心改了一行代码，而导致出现了业务异常）。简单的系统功能，只有简单的增删查改，所有的业务逻辑实际只有简单的增加和查询，根本无法提炼出来领域模型，若直接运用DDD的全部理论，实则浪费时间，杀鸡用牛刀。但是没有亘古不变的代码，重构是一直伴随着整个系统的演变，因此建议是即便是简单系统，最好一开始就搭好DDD的样子，后面我会继续出几篇文章，总结我对于DDD理论的实践。

  对于第二点，建议大家前往阅读此篇文章 [领域驱动设计（DDD）的实践经验分享之ORM的思考](https://www.cnblogs.com/netfocus/archive/2010/12/20/1910875.html" \t "_blank)，还是那句话，如果看完还不是很明白的话，请到留言区进行留言。其实目的就是为了避免进行数据建模， 数据模型，数据库技术绑架了面向对象开发的思路，无形中导致开发人员率先输出表设计，而忽略原本的模型设计，进行形成思维习惯，“你这样表不好设计啊”。而Hibernate其实是根据DDD中的JPA理论指导下设计，就是为了让开发人员更加专注于业务逻辑实现，进行领域建模。

  还有一个问题是一直以来很多人都错误的使用了像hibernate这种全ORM框架，建议大家前往阅读此片文章[如何对 JPA 或者 MyBatis 进行技术选型](http://www.spring4all.com/article/391" \t "_blank)

  第三点，初步认识DDD并想展开实践的，确实会存在这个疑惑，DDD难在一点的是，他需要你思维的转变，你需要开始不去关注数据库，不去关心SQL语句， 从数据建模转向领域建模，而这个过程又会将你面向对象过车功能的思维短板暴露出来，很抱歉，长期以来你并没有用面向对象的思维来开发。   Alistair Cockburn 提出了六边形架构，旨在解决了传统的分层架构所带来的问题。**倡导的是屏蔽技术细节，强调业务逻辑，最终目的是实现业务逻辑可重用，组织为一个可重用的自封闭的业务模型，即拷贝不走样**。仔细想想在贫血模式下，最大的不足就是业务逻辑不能重用，业务逻辑没有组织为一个可重用的自封闭的业务模型。



  初步认识领域驱动设计的第一篇文章就讲到这里，后续会继续分享领域驱动设计的概念和实践。走过路过不要错过，您的点赞是给予我写作的最大动力。

作者：plz叫我红领巾  
链接：https://juejin.im/post/5ca16166e51d456708675612  
来源：掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。