DDD FAQ: 领域建模与数据库建模的区别?

SaraQian 关注

FAQ：DDD领域建模与传统数据库建模（E-R建模）的区别?

> 我们技术人员做技术方面设计时，考虑业务逻辑往往是从设计数据库表开始的，因为我们的认知里，所有的软件的作用就是用来做各类业务数据处理的，对各类数据的增删查改就构成了一个有机的软件程序。这种从设计数据模型开始的思路和DDD的设计思想有冲突吗？两者可以和谐统一起来吗?

这是最近被问到的比较经典的问题，所以先抽出来尝试回答一下。（由于目前面对的大圈子还是坚持不懈地拥抱面向对象，所以下面的回答也以面向对象的领域建模为基础）

区别一：动作的分析去哪里了?

当我们需要描述一个对象，它通常涉及几个要素：

- **名词** - 这是一个什么对象 （如果类比到Java：这是一个什么Class）
- **形容词** - 这个对象是怎么样的 （如果类比到Java：这个Class应该有什么属性）
- **动词** - 这个对象能干嘛（如果类比到Java：这个Class需要什么方法）

而当我们做E-R建模的时候，我们需要考虑：

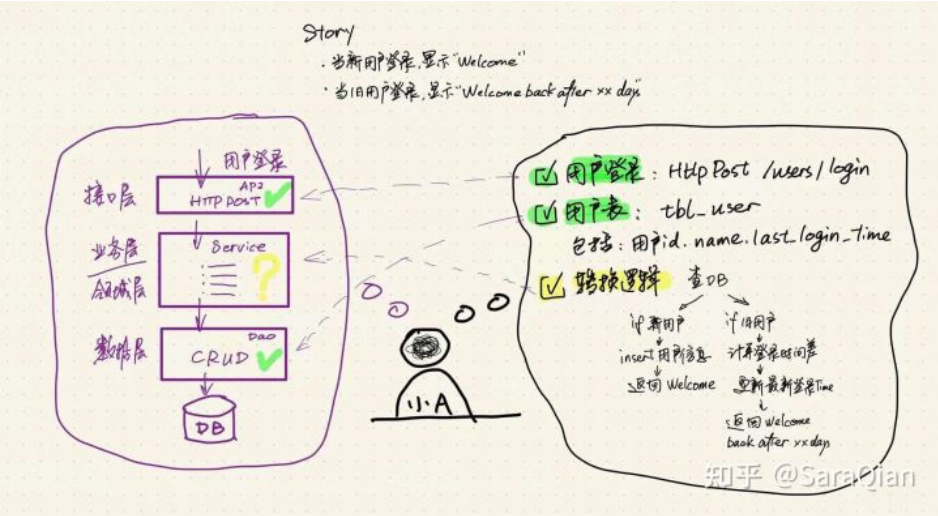
- 需要建什么主表 - 可以类比为对象 （当然也有些表只是为了关系映射）
- 表里面需要什么列 - 可以类比为这对象所需的属性

那，动作去哪里了呢？ER建模里面所谓的动作其实是名词间的关联关系，而不是真正业务上的动

于是又可能有这样的回答：

> 小A：数据库的动作不都是CRUD嘛，这还用说吗？
> 小B：那要不我们把用户动作那些触发的接口动作也列一下就好咯？

好的，那有了接口、有了数据库设计，就真的OK了吗？



设想一下这么一个场景，一个关于用户登陆的需求来了：

- Step1. 比较轻松的就可以分析到触发入口应该提供的API
- Step2. 然后基于ER模型分析得到用户表及基本字段应该也不难
- Step3. 如何从API入口动作转换到用户表的CRUD呢？这复杂又多分岔的转换通常就是边写边想了吧？

是不是忽然发觉了为什么service层总是那么乱的其中一个原因呢？我们总是抓住了两头技术相关的层去分析建模，却忽略了中间更为关键的业务编排和转换逻辑的梳理分析？

区别二：此对象非彼对象

看着上面场景的图，又会发现另一个问题：数据库ER建模出来的其实是DAL数据处理层的模型，而领域模型建的是业务层的模型 - 此模型非彼模型。

对于数据库建模来说，看重的是最后一步 - **“以终为始”**：我不管你中间怎么扭来扭去的转换计算，反正你最后能按照这个数据库模型给我往DB里面塞对东西就好了。所以落到代码分层架构上，基本也都是围绕最后这个数据库模型为核心。

而当我们使用DDD领域建模的时候，更看重的是中间业务那层的建模-**“以中为始”**。比如说：接口针对的是哪个业务实体？一个触发动作进来之后应该有什么后续动作去编排不同的业务实体？每个业务实体里面该有什么自己自治的逻辑和动作？业务实体本身在当前上下文中应该包含什么信息？而另一边厢，它相对没这么关注底层的数据模型，现在大家可能用的关系型数据库，也可能是nosql，自己从领域模型按需映射一下就好了。所以当落到分层架构的时候，更强调的是依赖倒置，以中间领域模型为核心，周边技术相关的模型依赖领域模型进行映射。

这两种思路反应出背后的想法、想解决的问题是不一样的：

- **“以终为始”**： 世间万物都是增删改查，数据才是核心，只要看破中间的转换得到最终数据模型，就是对了
- **“以中为始”**：业务逻辑才是最复杂的、最核心的、又万变不离其中的，要把握的更应该是中间的那层逻辑，而外层存储和集成方式怎么换都不应该影响我中间的业务逻辑。

> 小S：业务建模很重要哦
> 小A：嘿嘿我们的业务很简单的，几个CRUD就完了，没啥好建的
> 小S： Show me your code！
> 打开代码一瞧，几百甚至上千行的流水账式的service层方法比比皆是，那。。敢问这上千行写的是什么东西？

备注：这里更强调的是业务模型和数据库模型以谁为主的问题，在刚诞生的时候它们可能是长一样的，但在后面的演变中基于“谁应该跟随谁”的决定后，就有了不一样的走势。

区别三：读写模型的混沌

继续设想一个场景：

> 模型已经建好了，系统演进，这时候界面来了新需求，需要多展示几个各种格式化转换后字段，另外还需要支持一个综合查询页面或者小报表。
> 小A：我们开始分析这些新的字段、查询信息、报表信息和原来的表/列之间的关系吧
> 小B：巧了！都是一对一的关系，那简单了，咱们就直接把这些字段都加在原来主表里面就好了！（于是一个数据库表从40几个字段变成了100+字段）
> 小A：还没完，因为代码里面都是围绕这个数据库模型处理的，那这个Entity对象是不是也得加对应的属性呢？
> 小B：OK！（一个带着100+属性的Class出现了）

当更聚焦数据库模型的时候，我们强调的是数量级关联关系，如果设计强调的是去除数据冗余的话，尤其当读场景还没有特别的非功能性压力的时候，就很容易就会产生大量的宽表设计，以及一个有很多属性的数据模型对象。

而当进行领域建模的时候，首先在前期分析过程，它其实是忽略的读模型的（具体可以看事件风暴的过程，都是没有查询命令存在的）；其次，它很强调领域/聚合的纯洁性和自治能力，比如说：

- 如果要访问一个聚合内部的某些信息，必须从聚合根开始引用；
- 当涉及到两个不同聚合根之间的编排整合，就需要在应用层进行处理（而非某一聚合内部）；
- 同样的，当界面需要展示不同上下文/聚合信息时候，也会考虑以BFF（Backend-For-Frontend）的方式去整合而不在某一原有聚合里面处理

可见，它宁愿用“加一层”的方式也要保证领域内部的独立自治和纯洁。

回到上面的例子，当使用领域建模的时候，尤其是综合查询和报表这种一看就是跨聚合的场景，很有可能采用的就是独立于原有的领域模型而新建另一套专门的读模型，然后可以从代码的比如应用层（越过domain层）直接进行基于读模型的查询。从而避免一个非常臃肿的领域模型对象。

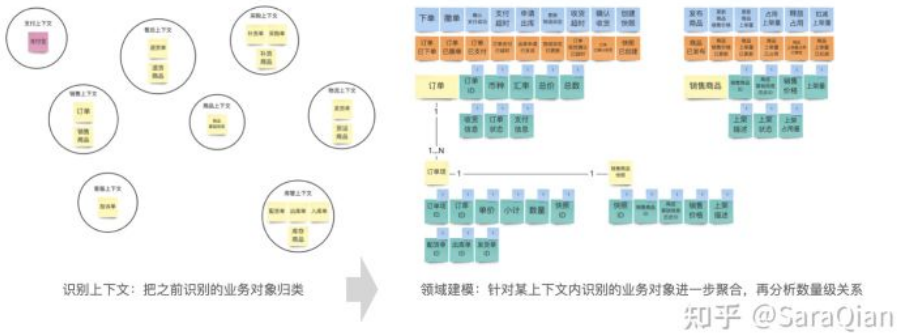
区别四：距离感的缺失

都说“距离产生美”，所以当说到距离感，仿佛透露着一丝文艺范，而程序猿是不是偏偏就缺乏这种对美感的追求？

在数据库建模的时候，分析完应该有什么表、什么字段之后，大家会去分析表与表之间的关系，两者是有关系还是没关系？有的话那是一对一、一对多、还是多对多？然后就根据这些数量级的关系继续去设置外键或者关联表。

而在领域建模的时候，除了这种数量级关系，它还非常看重对象之间的**远近亲疏**。

才是围绕聚合根下面按照数量级关系类似树状枝节地展开。



这种边界感其实是上面提到的领域内独立自治的基础。每圈定一个边界，边界内的东西就需要自己管理好自己，只由根节点对外暴露所需的部分。这也正是架构上常说的“高内聚低耦合”的实现形式。像常说的充血模型，也就是让对象相关的动作更内聚地在所属领地内部自治而已。

而另一边厢，如果我们不考虑远近亲疏，而把每一个名词都当作是平等的，只有数量级关系的时候，我们会发现这种数量级带来的依赖关系可以是延绵不绝的。场景一下，A和B依赖，场景二下，A和C、D依赖，B和D依赖。。等等。所以在做遗留系统改造的时候，数据耦合关系分析往往就是其中的一个切入点。

但事实上，往往难也难在判定这种远近亲疏。

两个名词之间有没有关系以及相互间数量级关系其实是肉眼可见的，但远近关系却往往是相对的。当站在不同的角度看就会得到不一样的结果。应该选择哪里划下这个圈，并不是一个照镜子的工作，而需要分析设计过程中的权衡。

如果我们回看UML等传统建模方式，里面也是有分析这种依赖强弱的。如：

- **依赖关系/Dependency**：这是耦合最小的一种关系，代码实现上，类A要完成某个功能引用了类B，则类A依赖类B
- **关联/Association**：代码实现上，类A成为类B的属性，而属性是一种更为紧密的耦合，更为长久的持有关系
- **聚合/Aggregation**：聚合用来表示集体与个体之间的关联关系。例如班级与学生之间存在聚合关系。聚合关系的类具有不同的生命周期
- **组合/Composition**：用来表示个体与组成部分之间的关联关系，组合关系的类具有相同的生命周期。
- **泛化/Generalization**：继承与实现

虽然百般讨厌UML，但现在在建模的时候有时还是会考虑用一些简化了的UML形式去表达一下这种远近亲疏。

好了，拍脑袋出来的4个区别到此为止，后面想到其他点再继续补充吧～

今天的FAQ愉快地结束咯 ^_^

编辑于 2021-06-07 09:28

领域驱动设计（DDD）　软件架构　工程技术