细说业务逻辑

前言

记得几个月前，在一次北京博客园俱乐部的活动上，最后一个环节是话题自由讨论。就是提几个话题，然后大家各自加入感兴趣的话题小组，进行自由讨论。当时金色海洋同学提出了一个话题——“什么是业务逻辑”。当时我和大家讨论ASP.NET MVC的相关话题去了，就没能加入“业务逻辑”组的讨论，比较遗憾。

其实，一段时间内，我脑子里对“业务逻辑”的概念也是非常模糊的。但在不断地阅读、思考和实践过程中，这个概念及其相关的内容才在我脑子里渐渐清晰。我想，很多朋友也许也对这个概念不是很了解，所以愿意结合既有资料和自己的思考，总结一篇关于业务逻辑的概述性文章，一则与朋友们分享探讨，二则也是为自己对业务逻辑的学习做一个总结和提升。因为我还不敢说对业务逻辑内涵及外延理解的非常充分，所以文中如有不当之处，还请各位不用客气，尽管批评就好！

内容提要

==========前篇==========

前言

内容提要

3.5、各种架构模式的比较及选择

4、结束语

参考文献

1、我把业务逻辑丢了！——找回丢失的业务逻辑

相信朋友们基本都是软件开发人员。不论身处什么职位，我们的工作都有一个共同的目标——制作软件产品。而所谓的软件产品，一定是在某个领域内去实现某些业务 。如此看来，“业务逻辑”本应和“软件产品”是紧紧绑在一起的，没有业务逻辑，何来软件产品？

但是，我发现一个奇怪的现象，一说业务逻辑，很多人就无法形成清晰地印象。例如，经典的三层架构：表示层、业务逻辑层和数据访问层，一提到表示层或数据访问层，大家脑子里马上能产生出清晰的概念，但一提到业务逻辑层，就有点模糊了，或者完全不知道其是什么，或者有个模糊的轮廓，但对其具体的职责、结构不是很清楚。真是奇了怪了！我们天天和业务逻辑打交道，搞不清业务逻辑是什么。

对于这个奇怪的现象，我思前想后，结合自身的教训（我也曾很长时间搞不清业务逻辑），终于弄清楚了其原因——这和我们接触这个概念的途径和认知结构有莫大关系。

不知道有多少人和我一样，首次接触“业务逻辑”这个概念是从分层架构中的“业务逻辑层”概念开始的，我相信不在少数。事情坏就坏在这里！为了让朋友们直观看清“业务逻辑”的概念是怎么被我们丢掉的，请大家看一个图，这个图展示了很多人对“业务逻辑”的认知过程。

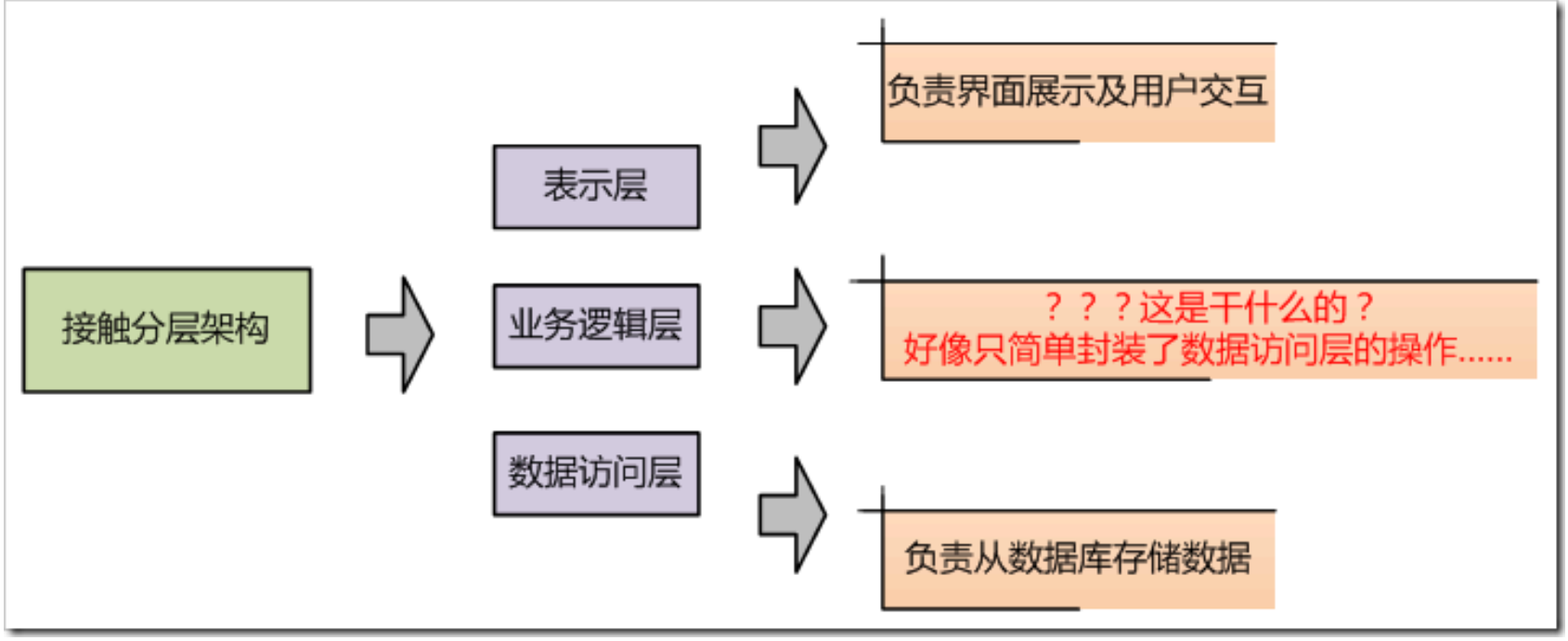

图 1-1、狭义的认知分解过程

如图1-1所示，我们先接触了分层架构，然后对每个层产生了初步的认识。其中，由于表示层和数据访问层的代码职责清晰明确，基本能正确认识。但是，由于我们接触的分层架构的Demo大多业务极其简单，又基本是CRUD操作集中型的业务。所以，我们脑子中就产生了疑问：这个所谓的业务逻辑层是干什么的？怎么就简单封装了一下数据访问层的操作？这有存在的必要吗？由于有了这种“先入为主”的误导，使得很多朋友脑中将“业务逻辑”和“业务逻辑层”两个概念混淆了，始终想不明白这东西到底是什么，做什么用的。再加上很多朋友所看的、所做的系统都是CRUD操作集中型的，就形成了“业务逻辑貌似就是对数据访问操作的简单封装”这一片面概念。

到底这一概念有没有错呢？其实没错，因为在简单的、CRUD操作集中型软件中，业务逻辑基本就是对数据访问简单的封装。但是，无错不代表全面，这是一种狭义的业务逻辑理解，而且是狭义中的狭义。为什么这么说呢？因为我们不但是在“业务逻辑层”这么一个狭义范围内去理解业务逻辑，而且还是CRUD集中型操作这种“非常瘦”的业务逻辑层范围内去理解，所以，可谓是在狭义的基础上的狭义。

当我们把这么一个“狭义中的狭义业务逻辑”与“业务逻辑”等同起来时，误会、迷茫、困惑、不屑就出现了。这就如同，给你一只温顺的哈巴狗，还是病怏怏的、无精打采的小哈巴狗，而你把这只“病怏怏的小哈巴狗”与“狗”的概念等同起来了。那么你一定就会为有人养狗看家和警察养狗当警犬抓坏人而困惑：这东西这么弱小，我一脚就踩死了，怎么弄用来看家和抓坏人呢？进而可能会产生“狗狗无用论”，“狗狗废品”等观念。当然，在现实中，很少

有人只见过小哈巴狗而没见过狼狗等其它狗类，所以，故事中的误会对“狗”一般是不存在的。但在现实中，确实有很多人只见过业务逻辑中的“小哈巴狗”，却没有见过业务逻辑中的“狼狗”、“藏獒”，所以，这种误会在对“业务逻辑”的理解上广泛存在。

那么，广义的情况究竟是怎么样的？请看下图。

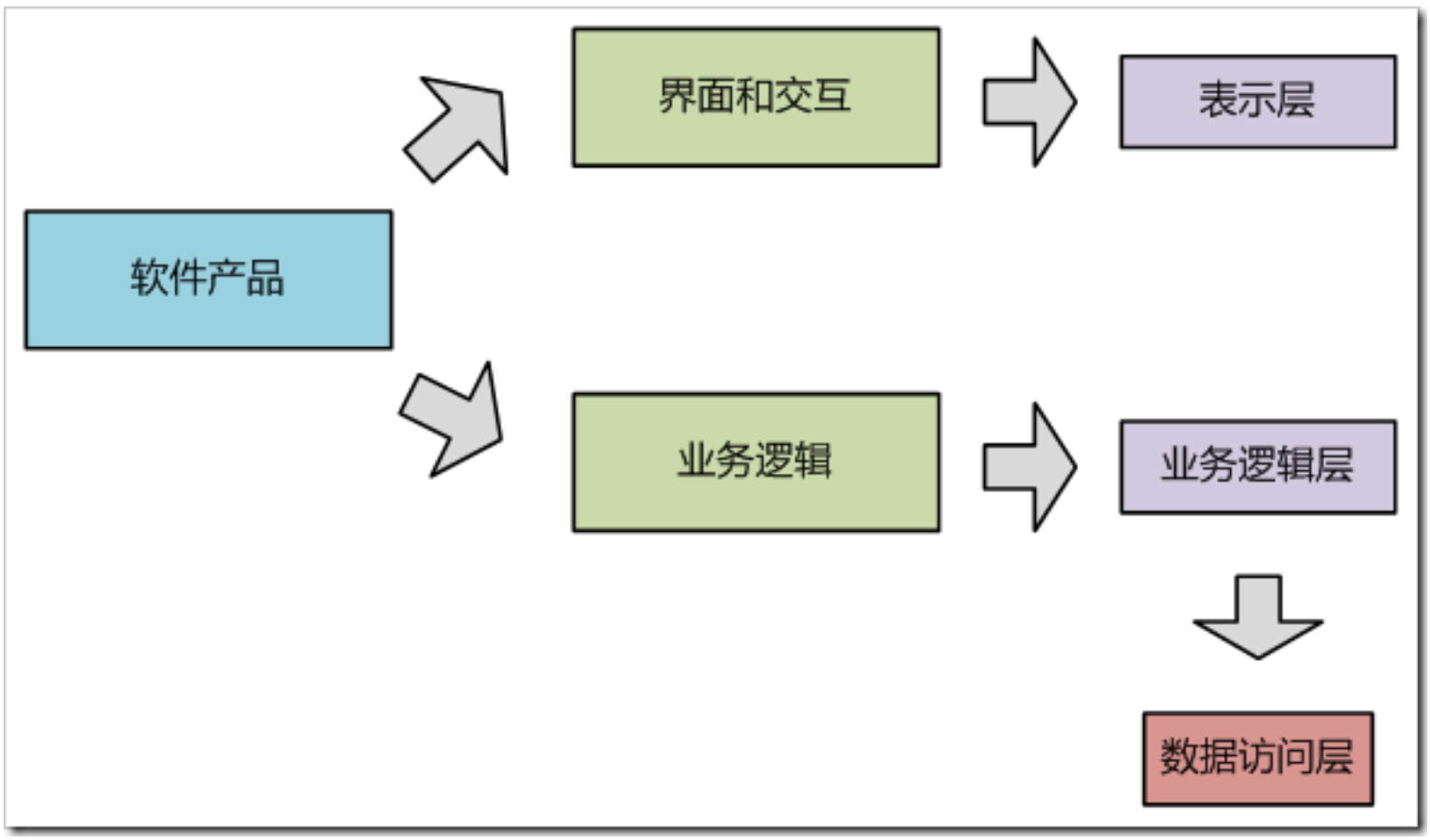

图 1-2、广义的认知分解过程

（注意！凡是不特别说明，下文中所有“数据”一词都指需要持久化的数据，而不包括内存中的临时数据。请各位留心。）

如图1-2所示，广义的认知分解应该是这样的：软件产品都是在某个领域内实现某些特定业务，所以，软件产品天生应该分解为界面交互部分和业务逻辑部分，其中业务逻辑部分是软件产品的核心，它客观存在于软件产品内部，但是无法对使用者产生直观刺激，因此业务逻辑不能与使用者直接交互。而界面交互部分是业务逻辑与使用者进行交流的接口，使用者通过界面交互部分，与业务进行交流，从而使得软件产品发挥其作用。

而在具体实现系统时，界面交互部分演化成表示层，业务逻辑部分演化成业务逻辑层。所以，可以认为，数据访问层不是软件产品自然演化的直接产物，之所以出现数据访问层，是因为某些产品的业务属于“数据操作集中型”业务，为了实现隔离、复用等目的，架构师从业务逻辑中分离出了频繁

使用的数据访问业务，形成了单独的数据访问层。从广义来说，可以认为数据访问隶属于业务逻辑，因为，数据访问操作实际上也是业务逻辑的一部分。

总结一下几个要点：（这几个要中的业务逻辑均指广义业务逻辑）

1）软件产品自然的可分为界面交互部分和业务逻辑部分。

2）从空间结构上看，业务逻辑和数据访问不是并列关系，而是隶属关系——数据访问隶属于业务逻辑。虽然在具体系统实现层面，数据访问层和业务逻辑层是并列存在，但从概念本质层面上分析，两者是隶属关系。

3）从时间结构上看，应该是先有业务逻辑的概念，才有数据访问的概念。业务逻辑衍生自软件本身，数据访问衍生自业务逻辑。

4）因为业务逻辑是软件产品自然的一部分，所以拥有业务逻辑是软件产品的必要条件（读者可以试着举出一个不包含业务逻辑的软件）。但是一个软件可以没有数据访问，如“计算器”、“不带存档的小游戏”等。

利用以上论述要点和认知分解，朋友们可以试试在脑中重新构筑狭义和广义“业务逻辑”的概念。看能不能把我们丢掉的业务逻辑概念找回来。关于业务逻辑更多的细节，将在下文中讨论。

2、细说业务逻辑

2.1、业务逻辑到底是什么

在第一大节里说了那么多，相信各位基本已经形成“业务逻辑”的概念了。如果我在这里再啰嗦什么，我不嫌累各位也要嫌烦了。所以，这里我仅给出两个定义。

广义上的义务逻辑——软件本身固有的一种品性，自然存在于软件产品内部，是软件具有的在某个业务领域内的逻辑，是软件的核心和灵魂。软件产品除界面和交互外的一切都可看作是广义业务逻辑。

狭义上的业务逻辑——等同于分层架构中“业务逻辑层”的职责，是软件中处理与业务相关任务的部分，一般狭义上的业务逻辑不包含数据持久化，而只关注领域内的相关业务。

对于以上两种定义，希望朋友们不要割裂开来看，而 要辩证统一的去看，这样，才能构建一个完整而辩证统一的“业务逻辑”概念。在下文中，将不再明确区分狭义和广义，“业务逻辑”一词将代表两者的辩证统一体。

2.2、业务逻辑的组成结构

业务逻辑作为一个高层次概念，其内在结构也是非常丰富的，下面我们深入其里，去探寻一下业务逻辑都是由哪些更底层的部分构成的。

2.2.1、领域实体（Domain Entity）

通俗的说，领域实体就是这个领域内有哪些东西。例如，银行业领域内有账户、支票、前台营业员等实体；B2C电子商务领域有商品、订单、交易等实体；魔兽世界游戏的领域内有角色、种族、道具、魔法等实体；高等代数领域有矩阵、行列式等实体。

领域实体是某个领域内各种对象的抽象，可以用名词表示（可以是具体名词或抽象名词，甚至动名词，只要其具有名词性），构成了整个业务逻辑的骨骼和静态模型。一般每个领域实体有自己的一些属性和行为。 顺便说一句，领域实体的存在时OOA&D的基础。

在具体的软件系统中，领域实体往往会根据架构的不同有不同的映射存在形式。

其中一种叫做Business Object（BO），即业务对象，某些文献称其为“充血实体类”，这种对象完整抽象了领域内的某个实体，封装了此实体相关属性和行为。在面向对象的设计和架构中，这种实体类很常见。

另一种叫做Data Transfer Object（DTO），某些文献称其为“贫血实体类”，其特点是仅有属性，不存在行为。这种实体类主要负责整体性传递数据。另外，与BO不同的是，DTO可以不抽象领域实体的全部属性，而只根据需要抽象一部分。例如，某个“User”实体存在很多属性，但如果某个方法仅需要其联系方式，可以设计一个DTO，仅有id，email，address，phone等就够了。在面向过程的设计和架构中，这种实体设计比较常见。

2.2.2、业务规则（Business Rules）

业务规则就是某个领域内运作的规则，构成了整个业务逻辑的灵魂和动态模

型。业务规则作用于领域实体，领域实体遵从业务规则进行运作。

如：在银行领域内，“转账时从A账户扣除相应款项，在B账户添加相应款项，并从A账户扣除相应手续费，并通过某些途径通知A和B账户的户主”就是一条规则。需要注意的是，业务规则比较抽象，并不是需求，需求需要具体且无二义性，而业务规则只是抽象的一种描述，例如，通知户主的途径是什么？电子邮件？电话？短信？并没有具体描述，但在规则中有“通知”这一项，因此不能将业务规则等同于需求。

2.2.3、完整性约束（Validation）

领域实体和业务规则构建了业务逻辑的主体，但在这主体之上，还存在着一个限制，这就是完整性约束。

完整性约束是对业务领域中的数据、规则的强制性规定与约束。这种约束是系统正常运转的保证。

如“账户密码不能为空”，“身份证号必须符合具体格式规定”，“转账流程必须具有原子性，A账户扣钱、B账户存钱、A账户扣除手续费、通知户主四项操作必须要么都做，要么都不做”，都是完整性约束。

2.2.4、业务流程及工作流（Business Processes and Workflows）

有了上述三项，业务逻辑还不能正常工作，因为还没有“启动器”和“过程托管器”。设想我们有了各种实体类，它们有各自的属性和行为，也有定义好的业务规则和完整性约束。现在实体类仅仅具有实现业务规则的能力，但它们如何启动并交互协调完成业务规则呢？因此我们需要有东西去触发和协调实体。

业务流程或工作流是启动及托管协调领域实体完成既定规则的过程。例如，“在线订购”是一个业务流程，它包括“用户登录-选择商品-结算-下订单-付款-确认收货”这一系列流程。各个实体如会员、订单、商品等已经包含了完成在线订购必要的行为，但仍需一个流程，才能真正完成业务。

具体到程序中，业务流程也许通过一个方法来实现，这个方法负责启动并协调各个实体类，完成一个流程。

2.3、业务逻辑层职责及相关争议

2.3.1、数据的格式化

关于数据的格式化应该放在业务层进行还是表示层进行一直存在争议。我个人的意见是这样的：

业务层送给表示层的数据应该具备以下要求。1）返回的数据应该完成了所有必要的业务处理和业务计算。 例如，若返回订单信息让表示层展示，会有个必要的数据——订单总额。这个数据需要首先用各个订单项的单价乘以数量，然后加和。那么，这个数据应该在业务层完成计算直接返回，总之不应让表示层进行任何业务处理和计算操作。2）一次性返回所有需要的数据，避免表示层再一个Action里调用多次业务。 打个比方，例如订单中有个“客户姓名”，这个数据不保存在订单表中，而是通过外键关联的，那么，业务层应该将“客户姓名”一并取出返回给表示层。总之，避免表示层在一个Action里多次调用业务层。3）不携带任何格式信息，仅仅是结构良好的纯净数据，如DTO形式。 因为，数据如何展示，是表示层的职责，如何在业务层中返回了过多格式信息，就会造成表示层的修改困难。例如，我曾听说过所里承接的一个实际项目，开始是使用B/S，当时他们的业务层返回的数据全都附带了html代码。后来，客户嫌B/S响应不够迅速（可能是客户公司的网络条件不好），要求改成C/S，当时全傻眼了，貌似几乎修改了整个业务层。那个项目相当庞大，7个子系统，投入200人开发了1年多，想想修改的难度吧。

2.3.2、数据合法性及完整性验证

一般做系统，都避免不了数据验证。上文曾经提到，完整性约束是业务逻辑的一部分。如此看来，数据验证一般应该放在业务层。但是，实际情况并不尽然。个人认为数据验证的方式，目前没有统一标准，可以根据需要放在表示层或业务层。但是，我个人不提倡在“表示层的服务端”放置过多完整性验证。因为，表示层的职责应该仅仅是接收数据并传递给业务层，不应对数据是否合法负责。过多的数据验证，不但令表示层代码臃肿，而且使得表示层职责变得不明确。

可以在“表示层的服务端”放置一些简单的验证，如空值验证，两次输入密码是否一致等，但业务关系紧密的验证，最好放在业务层。甚至有些验证只能在业务层验证，如“当前用户名不能与已有用户名重复”，这种验证需要访问持久化数据，需要由业务层完成。

这里之所以强调“表示层的服务端”，是因为一般在B/S系统中，都会在JavaScript里加入一些基本的数据验证，如空值检查，格式正则匹配等。这主要是为了减轻服务器负担，将大多数显然包含不合法数据的请求拒绝掉，而不发给服务端验证。当然，因为可能会出现JS被屏蔽或黑客恶意攻击行为，所以，所有验证不论JS中是否验证过，服务端（可能是表示层的服务端部分或业务层）一定要再进行验证。

2.3.3、CRUD

CRUD，即常说的增删改查操作。关于CRUD是否是业务层的职责，一直也是争议不断。因为目前并没有权威的定义，所以这里我斗胆说一下我对这个问题的看法。还请大家批判性阅读。

一说到“增删改查”，大家一定会觉得这理所当然是数据访问层的职责。我认为这个理解是对的，但是只对了一半！之所以这么说，是因为“增删改查”有两个层次含义。

第一个层次，是数据访问层次上的。在这个层次上，“增删改查”只是单纯的数据库操作，“增删改查”可以理解为“插入一条记录，删除一条记录，更新一条记录的信息，获取一条或多条记录”四个操作，其意义和着眼点完全是数据访问层面上的，不带有任何业务成分和业务知觉。这个层面上的CRUD应该属于数据访问层的职责。

第二个层次，是业务逻辑层次上的。在这个层次上，“增删改查”是业务领域内实体的变化以及一系列相关反应，“增删改查”可以理解为“领域内新增一个业务实体，领域内去掉一个业务实体，领域内一个业务实体更新了信息，得到领域内一个或多个业务实体的信息”。

两者最大的不同，是业务层面上的增删改查往往不是单纯的增加减少，还包括实体变化后相关的业务流程。下面举个例子：

“添加一个新的订单”——这是一条典型的“增”操作。在数据访问层面上，它的意义是“在表示订单的数据表里增加一条记录”；而在业务逻辑层面上，它的意义除了“领域内多了一个订单实体”外，还可能包括“根据业务规则判断是否是重复下单，根据金额对下订单客户的等级做相应提升、发送Email和短信通知客户等”。可以看到，业务层面上的“增”可能不仅是简单封装一个简单的插入记录，可能还要去做其他数据访问——提升用户等级，以及做一

些非CRUD的业务操作——发送短信通知。

在许多稍微复杂的系统中，业务往往不仅仅是封装了一条数据访问操作，而是还有很多如计算等业务处理，一个业务操作期间可能要多次使用数据访问操作。退一步说，即使某个业务仅仅封装了一条数据访问操作，其意义和层面也是不同的，在数据访问层面，仅仅是多了一条记录，而业务逻辑层面，是领域内多了一个业务实体。也许其本质上都是往数据库插入一条记录，但人类的抽象思维可以将之在不同层面上区分，这也是人类思维层面的一种抽象能力的表现。例如，我们知道太阳升起不过是地球自转使得从背阴面转到了向阳面，但当人们看日出时，很少有人会说“看！我们从背阴面转到向阳面了！”，我们会说“看！日出！”，这就是同一事物的不同层次表现。

2.3.4、存储过程

也许是性能上的诱惑，许多人喜欢在数据库系统中写很复杂的存储过程。这样，许多业务操作就被写到存储过程中去了。我个人建议，除非对性能要求极高，否则最好还是不要用存储过程实现业务。例如，在一般的系统中，某个业务操作可能需要1秒，而是用了存储过程只用0.1秒，看上去存储过程将效率提高了10倍。但对大多数用户来说，1秒和0.1秒的差别并不大，但是这样做的话，业务会变得十分不容易维护。所以，我个人觉得，除非十分必要，还是不要用存储过程实现业务。

3、业务逻辑的架构模式及实现

Martin Fowler在《Patterns of Enterprise Application Architecture》一书中，总结了四种企业应用中业务逻辑的组织方式：Transcation Script，Domain Model，Table Module及Service Layer，另外，本书的第十章“Data Source Architecture Patterns”中包含一种模式——Active Record。结合软件体系结构的近期发展及个人的理解，我更倾向将Active Record归入业务逻辑的组织模式，而Service Layer应该不算做业务逻辑特有的模式，所以，在本文中，将介绍四种模式：Transcation Script，Table Module，Active Record及Domain Model。

3.1、Transction Script

3.1.1、概述

Transction Script（以下简称TS）是一种面向过程的业务逻辑组织方式。这里首先要强调一点，这里的Transction一词与数据库系统中表示“事务”的Transction没有任何联系。TS是将领域中的业务分解为一个个业务过程，每个过程实现一项业务功能，具体到程序中，一个业务过程往往映射到一个方法。TS是完全面向过程的业务组织模式，适合应用于业务逻辑较简单的场合。

应该说，我们见到的绝大多数系统都是以TS组织业务的。例如PetShop及Oxite等经典示例。有时为了方便维护，开发者会将同一领域实体相关的业务方法集中到一个类中，这里虽然用到了领域实体和类的概念，但和面向对象没有任何关系，完全是面向过程的。

当使用TS时，可以不需要数据访问层，而是将数据操作执行代码（如执行SQL或存储过程的代码）直接嵌入在业务方法中，有时为了复用性和维护性可以编写一个helper类封装数据库的操作。当然这并不是说TS不能配合数据访问层使用，但由于应用TS的场合一般业务非常简单，如果配合Repository或 ORM使用，业务逻辑层往往就会变得非常“瘦”，看起来仅仅是对数据访问层的封装。一般在需要支持多数据库的场合，要配合Repository和 Abstract Factory使用。

TS的示意图如下所示：

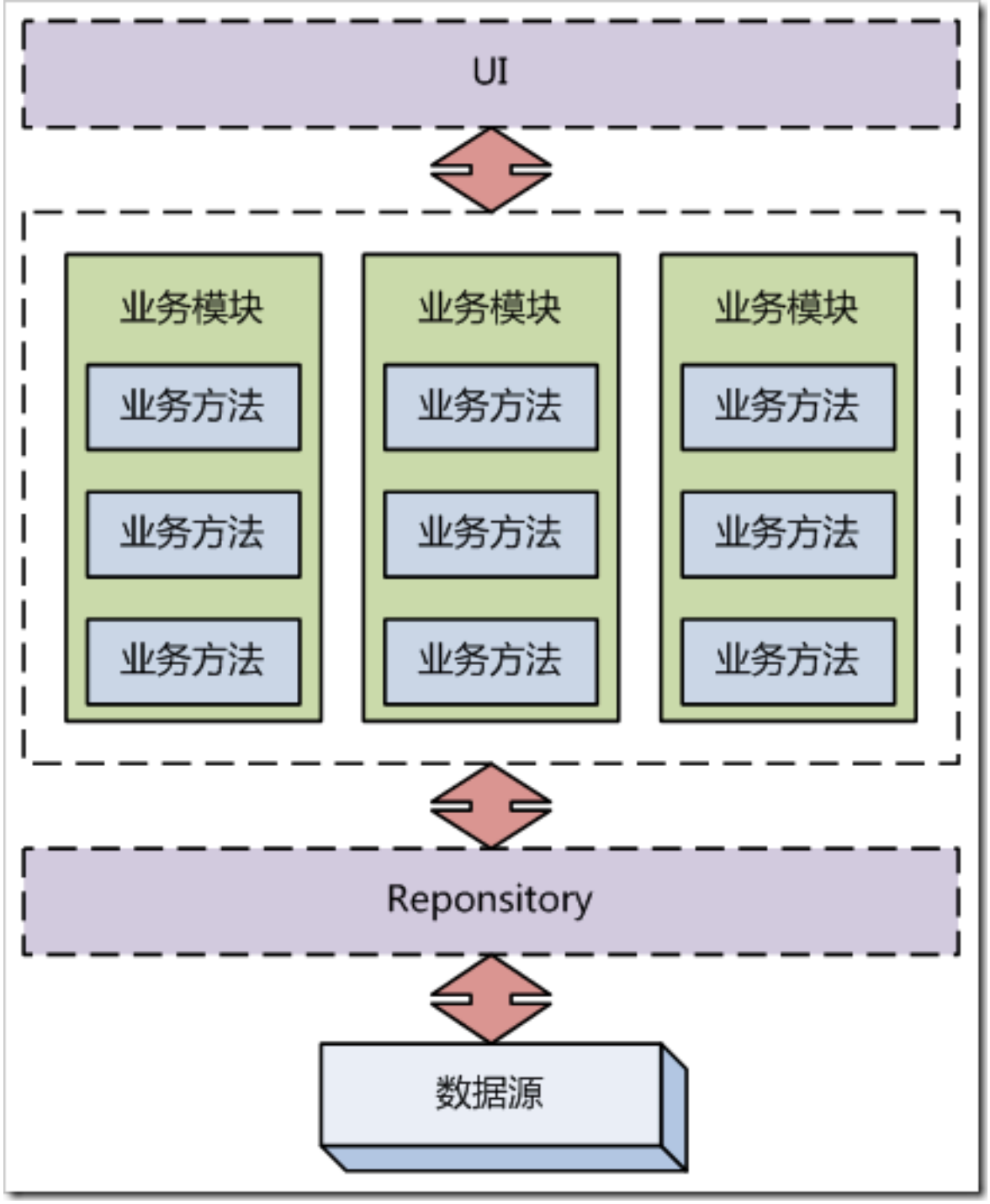

图3-1、 Transcation Script架构示意

可以看到，在TS中，业务层并没有面向对象的东西。也许会用到类，但类只是组织业务方法的模块，每个模块中有一个个业务方法，每个业务方法完成一个业务流程，完全按面向过程结构组织。

3.1.2、分析

- *什么时候可以用TS?*

应该说，如果具备以下条件之一 ，你可以考虑TS：

1）系统业务十分简单直观，并且频繁变动的可能性不大

2）工期很紧，需要尽量压缩设计的时间，尽快投入编码

3）不能熟练掌握和使用OO进行系统的设计与开发

4）厌恶OO，就是喜欢面向过程

- *TS的优点？*

1）设计阶段投入较小，启动耗费低。因为TS较容易掌握，使用起点低，所以使用TS的初期投入较少

2）在业务比较简单直观的情况下，TS结构的代码直观易懂，具有良好的可维护性

- *TS的缺点？*

1）容易造成代码冗余。因为各个业务自行组织流程，所以减少了复用的机会，可能产生重复性代码

2）因为TS天生不适合业务复杂的系统，当系统业务较复杂时，可能会令业务层代码繁杂不堪

3.2、Table Module

3.2.1、概述

Table Module（以下简称TM）同样是一种面向过程的业务逻辑组织方式，与TS不同的是，TM更贴近关系型数据库结构。在TS中，一般使用DTO等进行数据表示和传递，其着眼点一般在单个对象。而TM一般根据数据表组织业务模块，每个模块对应一个表，其中包含了这个表的相应处理。并且在业务层内，使用库-表结构的对象进行数据操作，做到最大限度与数据表的对应。业务组织一般按照面向过程组织。

一般当业务相对简单且业务基本集中在CRUD操作时，可以考虑TM。使用TM意味着使用数据驱动设计。通常自己实现一套库-表结构操作对象的库是难度比较大的，所以一般选用TM时，所使用的平台应该包括这么一套库。如.NET平台上的ADO.net就内置了丰富的库-表操作，DataSet，DataTable，DataAdapter等在TM架构的实现中可以起到非常方便的作用。

使用TM后，一般不需要再配合Reponsitory或ORM，因为此时的业务层也是面向过程和面向关系型结构的，无须映射。

TM的示意图如下：

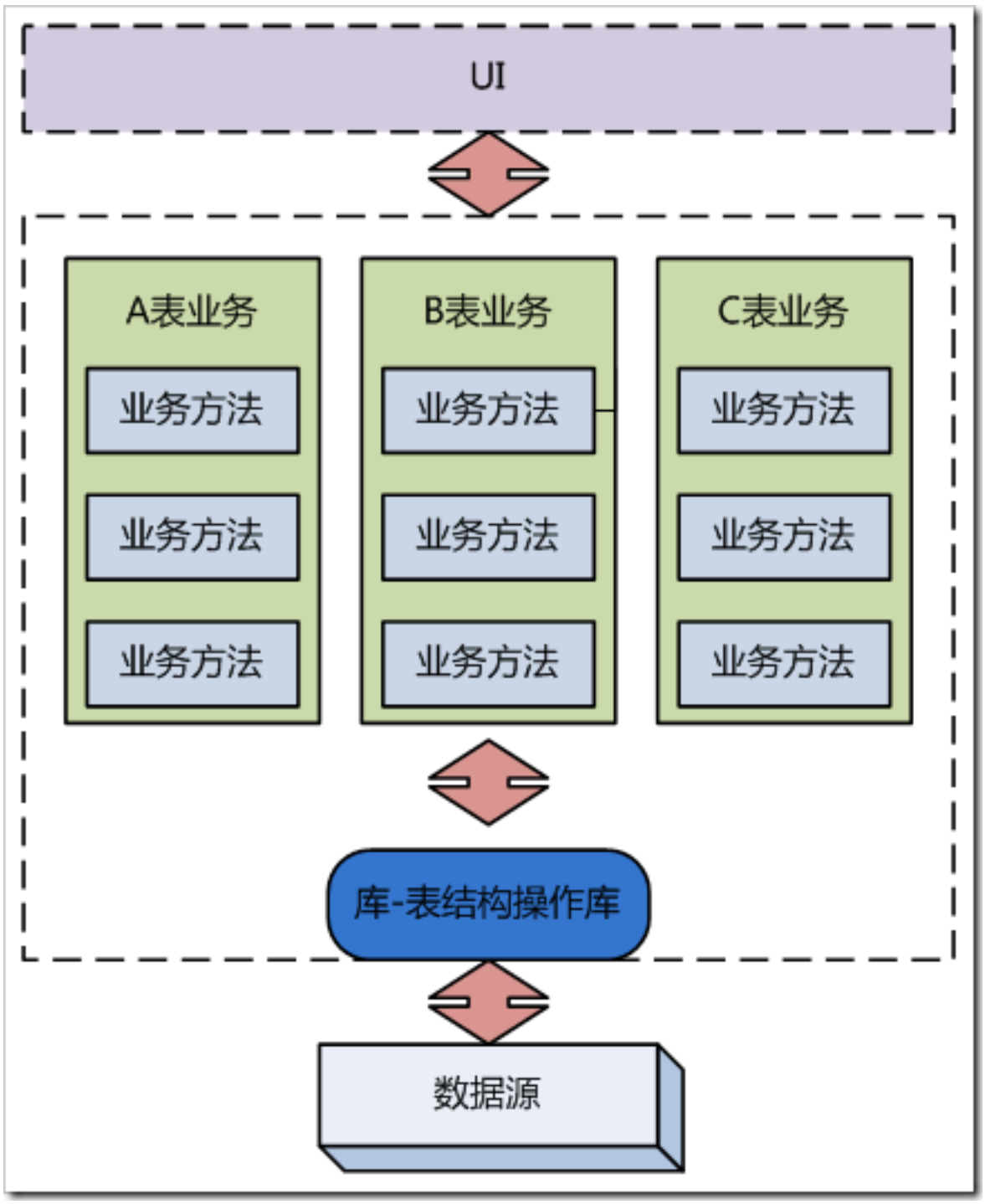

图3-2、Table Module架构示意

在使用TM后，业务代码中往往有各种对象对应数据库中的库、表、记录、字段等元素，并提供类似关系数据库的操作。

3.2.2、分析

- *什么时候可以用TM?*

如果同时 具备以下条件，你可以考虑TM：

1）系统业务较直观，以CRUD操作比较集中

2）整个开发的指导思想是数据驱动

3）所选用的平台有成熟的库-表操作库支持

- *TM的优点?*

1）类似关系数据库的数据操作方式非常直观，使得设计和编写数据操作功能的代码简单高效

- *TM的缺点?*

1）TM需要完全的数据驱动，从业务到UI传递、存放数据都要以表结构形式，造成一定程度上的不灵活

2）当业务并非CRUD集中型操作，特别是领域模型和数据库表模型差异较大时，使用TM组织业务的难度非常大

3.3、Active Record

3.3.1、概述

Active Record（以下简称AR）是一种面向对象的业务逻辑组织方式。AR适用于在业务较简单的情况下，应用面向对象思想进行设计。它的基本思想就是将领域中每个实体抽象出一个业务类（BO），然后，将这个实体的数据和行为封装成类的属性和方法。特别的，将CRUD功能也封装进BO中。也就是说，AR中的BO同时具备业务方法和持久化功能。其本身具有ORM的特性，其内部要处理关系实体间的关联问题。

使用AR时，一般最好有相应框架支持，否则完全手工实现AR有点麻烦。像Castle框架中就有AR功能，Linq to sql也有AR的意思。使用AR后，一般不需要再单独使用数据访问层。

AR的组织架构如下图：

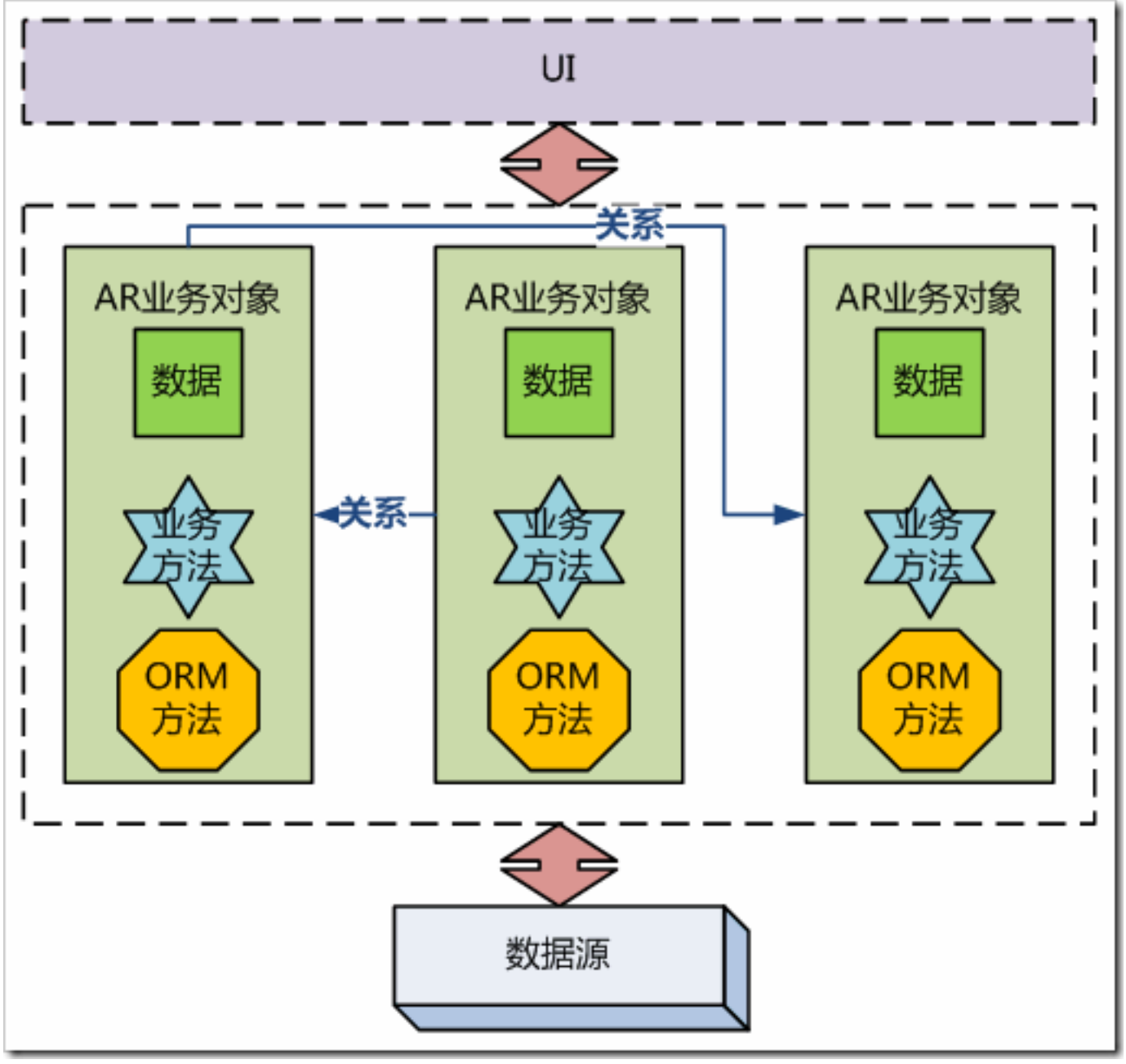

图3-3、Active Record架构示意

从图3-3中可以看出，AR对业务领域进行了一个简单的OO抽象，将各个实体抽象为AR业务对象，AR业务对象内含有数据、业务方法及数据访问相关的ORM方法。另外，AR业务对象要维护实体间简单的一对多和多对多等关系。

3.3.2、分析

- *什么时候可以用AR?*

如果同时 具备以下条件，你可以考虑AR：

1）系统业务较直观

2）想尝试使用或习惯于使用OO进行系统设计与实现

3）平台上有成熟的AR框架可以用

- *AR的优点？*

1）使用OO的方式进行设计与实现，能在一定程度上避免冗余代码问题）

2）使用AR后，与某个实体相关的数据和业务全部集中于AR业务对象中，模块内聚性好，便于维护

3）实践证明，AR结构的业务层编码效率很高

- *AR的缺点？*

1）AR仍需要关注数据之间的关联，在一定程度上带有数据表和影子，没有完全摆脱数据驱动，所以当业务领域和数据库结构差距大时，实施困难

2）AR的CRUD是以个体为粒度的，当进行批量操作时，如一次查数千个数据，如果严格尊从AR就需要生成数千个AR业务对象，这简直是场灾难。所以在有大规模查询的情况下，可以考虑使用TS配合AR

3）如果业务非常复杂，AR将力不从心

3.4、Domain Model

3.4.1、概述

Domain Model（以下简称DM）是一种适合领域驱动和为复杂业务系统组织业务的面向对象业务逻辑组织方式。前面三种架构模式都有一个共同的缺点——不适合业务复杂的系统。那么何为复杂何为简单？很抱歉，我给不出明确答案，而且我估计世界上任何一个人都很难给出标准的无争议答案。因为软件系统中的复杂和简单本身就是一个难以量化的指标，很多时候，只能靠专业人员的经验了。

我个人估计，世界上95%的软件系统其业务难度都不会超出上述三种模式的能力范围，而若你不幸遇到剩下的5%，恐怕目前只有Domain Model能帮你了。Domain Model是一种纯面向对象的业务架构模式，它的核心思想是获取领域中的各种实体抽象，然后完全按照现实领域中的情况去建模和运行。并且业务对象是“持久化无知”的。 关于“持久化无知”下面细讨论。这个模式

十分复杂和难以掌握，但一旦掌握并使用，其能力绝对会超乎你的想象。

下面看一下DM的架构示意图：

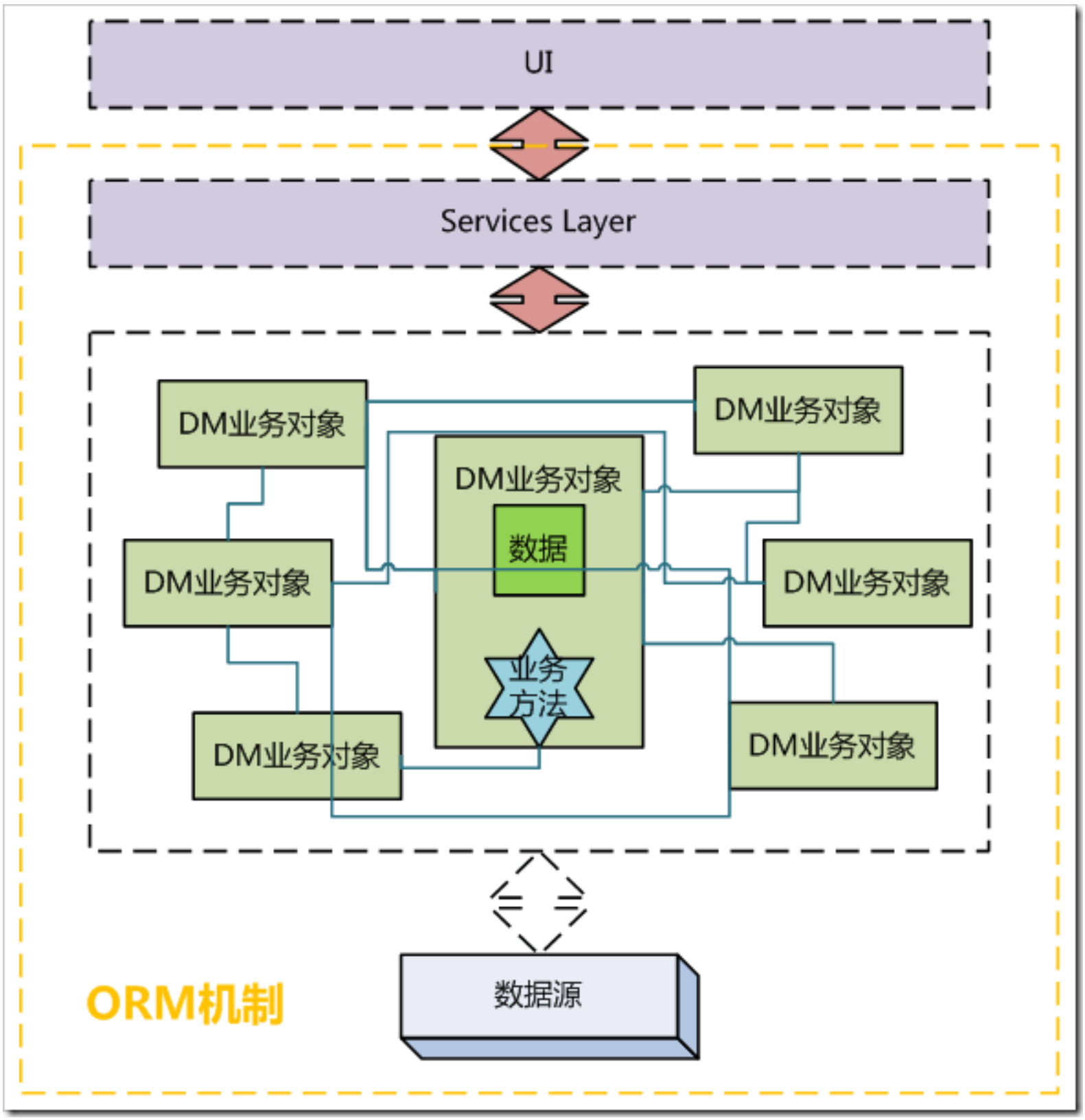

图3-4、Domain Model架构示意

从图3-4中可以看出，DM看上去是个十分纠结的模式，而实际上，它确实很纠结！实际上，我认为如果能熟练掌握并运用DM进行业务逻辑的组织，那这人绝对是架构师中的大师级人物（我目前是做不到）。

还是先结合图示分析一下DM中的要点。

第一，DM中的业务对象是纯业务对象，不含数据访问操作。这个可以和AR中的业务对象对比一下。也就是说，DM中的业务对象是纯业务对象，它们只关注与业务的实现。

第二，DM的组织内部对象多，关系复杂，而这种关系不再只是那种简单的一对一、一对多的关系，而是领域中的各种依赖和关联的抽象，关系类型多，非常复杂。

第三，DM需要业务部分“持久化无知”。所谓持久化无知，指业务部分只需执行业务功能，而不必关系持久化。在使用DM时，必须设计一套ORM机制（注意这里用到了“机制”一词，而不是“框架”或“库”），使得在业务系统运行时，自动在必要的时候执行数据持久化操作。这也是为什么上图数据源和业务层间的箭头是虚线的关系。

上文曾说过，DM要最大程度模拟现实情况。而现实世界和软件世界最大的区别就是现实世界是“内存无限大、永不停机的”，可以把现实世界看成在一个无限大内存里永不停止运行的程序。而软件世界不同，它的内存有限制，我们不能将所有对象都放在内存，而且一旦掉电，它就会停止运行，正因如此，我们才需要持久化机制去配合DM模拟现实世界。为了让业务更接近现实，它必须对持久化过程毫无感觉。而一套持久化机制默默为其营造了一个好似内存无限大、永不停机的环境，因此DM才得以发挥威力。

第四，DM往往需要Services Layer的配合。因为DM内部仅有一个个业务对象，它们互相调用，并没有提供一个友好的接口与UI交互，所以在使用DM时，往往在其上对各种UI需要的服务进行封装（回顾一下Facade模式），形成一个Services Layer，以方便与UI交互。

3.4.2、分析

- *什么时候可以用DM?*

如果同时 具备以下条件，你可以考虑DM:

1）系统业务极为复杂

2）有功底扎实和经验丰富的精通OO的架构及设计师

3）项目经费和时间充足

4）贯彻领域驱动设计

- *DM的优点？*

1）完全的OO思想运用，将使你享受到OO的所有优势

2）应付复杂业务的强力杀手锏。如果DM运用得当，将会使得复杂业务被高效解决

- *DM的缺点？*

1）使用门槛极高，难度极大，如果团队中没有精通OO和系统架构且经验丰富的专家很难实施

2）设计过程极为复杂，可能会导致设计瘫痪

3）如何设计良好的ORM机制辅助DM是一大难题

3.5、各种架构模式的比较及选择

相信看过上文内容后，各位一定对各种业务组织模式及其特点、优劣、应用场景有了清晰地认识，如果我在这里再喋喋不休讨论各种模式的比较及如何选择，难免有侮辱各位智商之嫌O(∩_∩)O~，所以这里我只给大家呈现一幅决策网络图，以期起到一个梳理和归纳总结的作用。

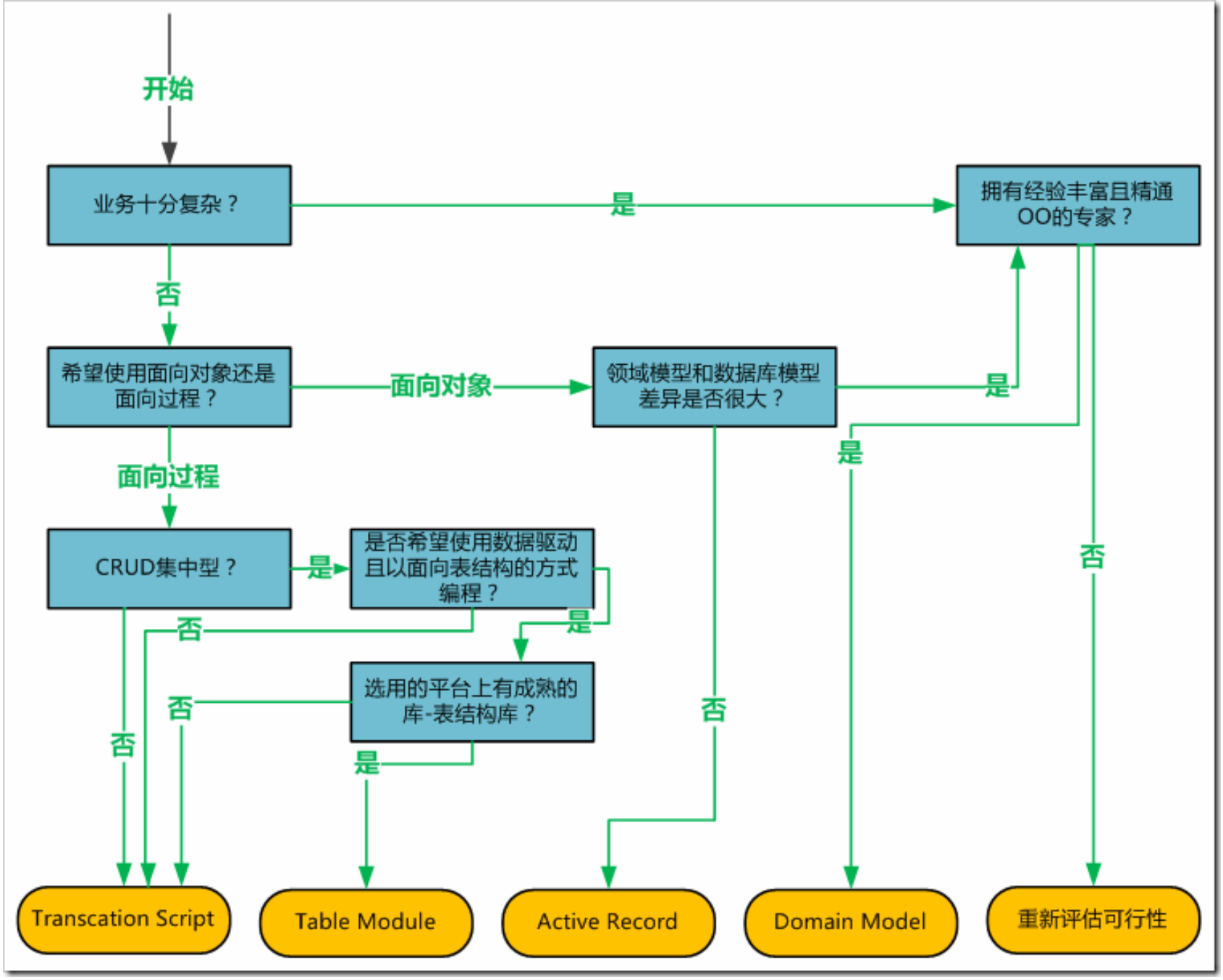

图3-5、业务架构模式决策网络

（郑重声明：图3-5为本人原创，并非摘录自已有文献，因此此图的选型流程仅代表个人意见。由于笔者水平有限，不能保证此图一定合理和正确。因此在实际选型时请多多参考已有文献及咨询相关专家，此图只起总结归纳和探讨作用，不作为任何指导和规范。若因遵循此图选型而给项目带来的任何经济及其他方面损失，笔者不承担任何责任。）

4、结束语

本文通过两篇文章的篇幅，先后介绍了业务逻辑的定义、相关理论及经典的业务逻辑相关的架构模式。本文中阐述了不少已有理论，亦掺杂诸多个人理解及看法。因此请各位在阅读时多进行批判吸收，同时参考以后经典文献及书目综合理解业务逻辑，切勿仅看我一家之言。

另外，由于本文仅仅是综述性文章，不能具名业务逻辑的各个方面，在深度上也基本是浅尝辄止。因此，若希望深入理解业务逻辑，可以看到相关经典

书籍及文献。

参考文献

[1] [意]Dino Esposito, Andrea Saltarello, .NET软件架构之美英文版(原名Microsoft .NET Architecting Application for the Enterprise), 人民邮电出版社, 2009

[2] [美]Martin Fowler, 企业应用架构模式影印版(原名Patterns of Enterprise Application Architecture), 中国电力出版社, 2004

[3] [美]Mclaughlin, Pollice, West, 深入浅出面向对象分析与设计影印版(原名Head First OOA&D), 东南大学出版社, 2007

[4] Google, www.google.com