互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 2 章 组织领域逻辑

领域逻辑的组织可以分为三种主要的模式：事务脚本、领域模型以及表模块。

保存领域逻辑最简单的方法是使用事务脚本。简单地说，事务脚本是这样一个过程：从表示层获得输入、进行校验和计算处理、将数据存储到数据库中以及调用其他系统的操作等。然后，该过程将更多的数据返回给表示层，中间可能要进行大量的计算来组织和整理返回值。基本的组织方式是让每个过程对应用户可能做的一个动作。所以，我们可以将这一模式想像成一个动作或业务事务的脚本。该脚本不必一定是单个的内嵌过程，还可以分离成不同的子例程，这些子例程可以在不同的事务脚本之间共享。但是，每一个动作还是由一个过程来驱动。例如，一个零售系统可能会有结账、将商品放到购物车、显示交货状态以及其他一些事务脚本。

事务脚本具有如下几个优点：

- 它是一个大多数开发者都能理解的简单过程模型。
- 它能够与一个使用行数据入口或表数据入口的简单数据源层很好地协作。
- 设定事务边界的方法显而易见：一个事务始于其脚本的打开，终于其脚本的关闭。很容易用工具在幕后设定事务边界。

但是，事务脚本也存在许多缺点，当领域逻辑的复杂性增加时这些缺点就会凸现。当若干个事务需要做相似的动作时，通常使多个脚本中包含某些相同的代码。通过将这些代码提取出来组织成公共的子例程可以部分消除这种情况，但在许多时候，消除副本仍比较棘手，而检测副本则更为困难。这使得应用程序没有清晰的结构，变成了一张由许多程序组成的极度杂乱无章的网。

当然，复杂逻辑情况的处理必然要引入对象，解决前述问题的面向对象方法就是使用领域模型。我们建立一个应用领域的模型，至少在开始的时候主要围绕领域中的名词来组织。例如一个租赁系统会有租约、资产等类。进行校验和计算的逻辑会置于领域模型中，因此发货对象可能会包含计算一次运输费用的逻辑。可能还有其他例程也完成计算账单的功能，但它实际上是调用领域模型中的已有方法来实现的。

用领域模型而不是事务脚本正是面向对象的程序员所极力鼓吹的“理论体系转换”的精髓。在领域模型中，不再是由一个过程来控制用户某一动作的逻辑，而是由每一个对象都承担一部分相关逻辑。如果不习惯领域模型，开始学习使用它时会充满挫折感，为了找到行为在哪里，你会从一个对象冲到另一个对象。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

用一个简单的例子来说明这两种模式的区别比较困难，但是在关于模式的讨论中，我还是选择了一个简单的领域逻辑片断，试图通过用这两种方法建模来说明它们的不同之处。最易体现出区别的是这两种方法的顺序图（见图2-1和图2-2）。核心的问题是：对于同一给定的合同，不同种类的产品有不同的收入确认算法（关于应用背景，请参见第9章）。计算收入确认的方法中必须先确定给定合同中产品的种类，然后应用正确的算法，之后再创建相应的收入确认对象来保存计算结果。（为简单起见，我省略了与数据库的交互问题。）

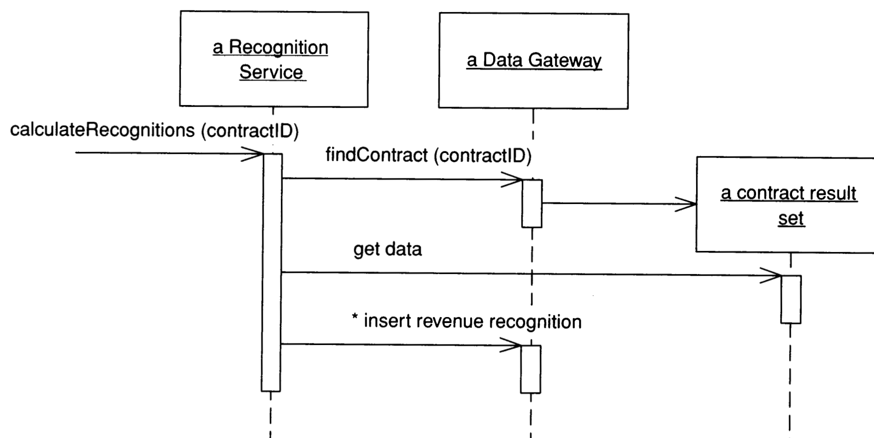

图2-1　使用事务脚本计算收入确认

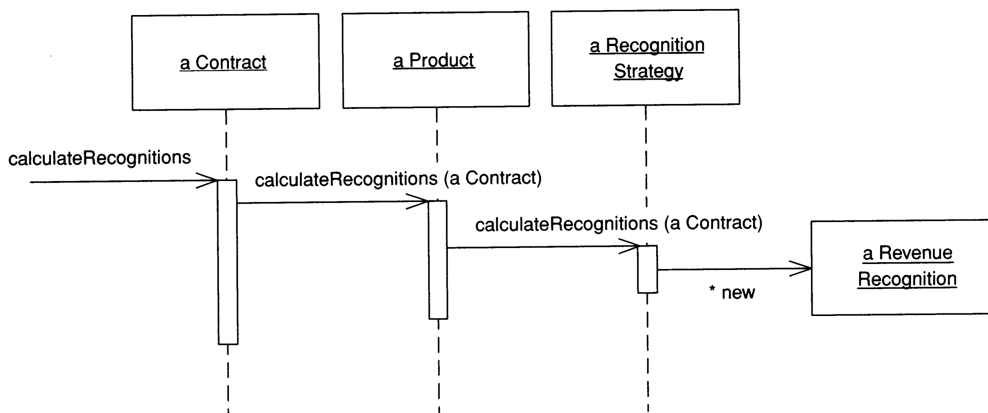

图2-2　使用领域模型计算收入确认

在图2-1中，事务脚本中的`calculateRecognitions`方法完成了所有的工作。底层对象只有一些表数据入口，它们仅仅完成将数据传送给事务脚本的任务。

反之，图2-2中有多个对象，它们每个都向前传递一部分行为给另一个对象，直至策略对象创建了结果。

领域模型的价值在于你一旦掌握了它，就可运用许多现成的技术来较好地组织日趋复杂的领域逻辑。例如，当增加新的收入确认算法时，只需增加相应的新策略对象即可。而使用事务

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

脚本则需要在脚本的判断逻辑中增加许多新的条件。如果你也成为了像我一样坚定的面向对象信徒，你甚至会在相当简单的案例中也宁愿采用领域模型。

领域模型的开销来自使用上的复杂性和数据源层的复杂性。刚接触复杂对象模型的人需要时间来适应复杂的领域模型。通常，开发者要在采用这一模式的项目上工作数月后才能转变他们的思维方式。但是，一旦习惯了领域模型，一般就可以在将来很好地运用它，从而受益终生。当然，的确存在不少开发者，他们似乎总也无法适应这种转变。

即使成功地完成了这一转变，还需要面对将领域模型映射到数据库的问题。运用领域模型越充分，当你将它映射到关系数据库时就越复杂（通常使用数据映射器）。复杂的数据源层就好似一笔固定资产投资——你需要付出相当多的钱（如果你想买）或时间（如果你想自己建造）才能得到一个良好的数据源层，但一旦拥有了它，就可以利用它完成许多工作。

第三种组织领域逻辑的模式是表模块。这一模式乍看起来与领域模型很相似，它们都有合同、产品和收入确认类。关键的区别在于领域模型对数据库中每一个合同都有一个相应合同类的实例，而表模块只有一个公共的合同类实例。表模块设计成与记录集一起工作，因此，在一个用来处理合同的表模块中，客户需要首先对数据库进行查询以生成一个记录集，然后以记录集为参数创建一个合同对象。客户可以调用合同对象的方法来完成各种操作（见图2-3）。如果客户要对某个指定的合同进行操作，它就必须在调用方法时附加该合同的ID。

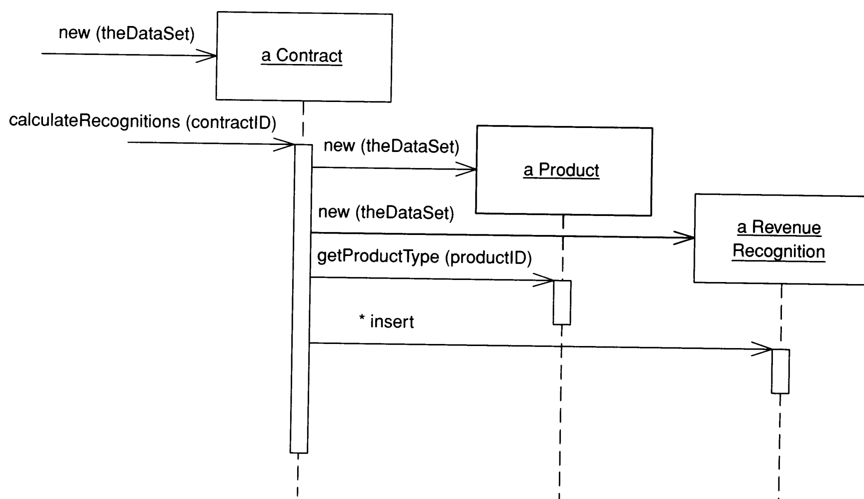

图2-3 使用表模块计算收入确认

在许多方面，表模块是事务脚本和领域模型的一个中间地带。它围绕表而非直接围绕过程来组织领域逻辑，提供了更多的结构，而且更容易发现和移除冗余代码。但是，你无法应用许多在领域模型中可以使用的组织细粒度逻辑结构的技术，例如继承、策略和其他面向对象的设计模式。

表模块最大的优点在于其与软件架构中已有部分的衔接。许多GUI环境在设计时都假定其将与SQL 查询的返回结果协同工作，这些结果是以记录集的方式组织的。表模块也工作在记录集之上，因此你可以很容易对数据库进行一次查询，然后在表模块内对返回结果进行操作，再把操作完成后的数据传给GUI显示。你也可以在将用户界面中修改的数据回传到数据库时，使用表

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

模块来完成计算和校验。许多平台都使用这种开发网格，尤其是微软的COM和.NET。

2.1 抉择

那么，这三种模式应该怎样选择呢？要做出抉择并不容易，在很大程度上取决于领域逻辑的复杂度。图2-4是一个用PowerPoint示意的不精确图表，它能指导你进行选择。虽然我并不欣赏这种非量化的表示方法，但它有助于可视化我对这三种模式区别的理解。当领域逻辑很简单时，领域模型并不合适，因为要透彻理解这一模式需要很大代价，而且数据源层的复杂性也会在开发中增加许多工作量，这些努力此时不会得到回报。然而，当领域逻辑的复杂度增加时，除领域模型以外的其他方法就都不再适用，因为它们增加新功能的困难程度会随系统复杂度的增加而呈指数增长。

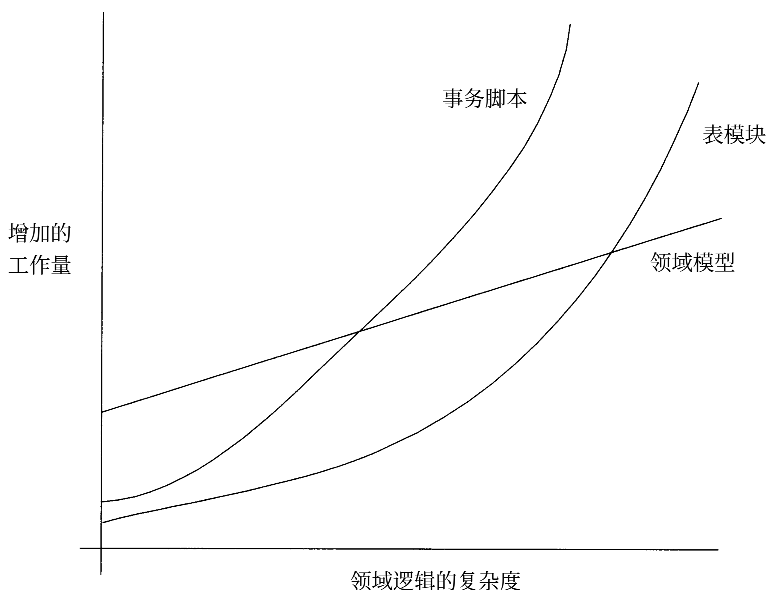

图2-4　对不同的领域逻辑组织方式，领域逻辑的复杂度和工作量之间的关系示意

当然，你的问题是要确定应用在*x*轴的哪个位置。好消息是只要你的领域逻辑复杂度大于7.42，你就应当使用领域模型。坏消息是没有人知道如何测定领域逻辑的复杂度。因此，事实上你所能做的只是向那些有经验的开发者请教，他们能对需求进行早期分析并做出正确的判断。

某些因素会对图中的曲线做一些修正。一个熟悉领域模型的开发小组能降低使用这一模式的固有开销，但由于数据源层的复杂性，它不会降低到与其他模式相同的起点。但是，开发小组的经验越丰富，我越倾向于使用领域模型。

是否应用表模块很大程度上取决于环境对通用记录集结构的支持。如果开发环境拥有大量基于记录集的工具（如.NET或Visual Studio），则表模块就很有吸引力。事实上，我找不出在.NET环境下使用事务脚本的理由。当然，如果没有特定的基于记录集的工具，我就不会纠缠于表模块。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

一旦你做出了抉择，虽然你的决定不是绝对不可更改的，但中途的改变也会很棘手。因此，事先花费一些时间来进行思考和选择是值得的。如果在开发过程中才发现你的选择是错误的，且你原来的选择是事务脚本，那么不要犹豫，你应立即改用领域模型。而如果你原来的模式是领域模型，中途转而采用事务脚本，这通常不太值得，除非能简化你的数据源层。

这三种模式并不互相排斥。事实上，使用事务脚本来处理一部分领域逻辑，同时使用表模块或领域模型来处理剩下的部分，这也是很常见的。

2.2 服务层

处理领域逻辑的常见方法是将领域层再细分成两层。服务层独立出来，置于底层的领域模型或表模块之上。通常只有使用领域模型或表模块时才会这样细分，因为仅使用事务脚本的领域层并不复杂，没有必要再单独设服务层。表现逻辑与领域层的交互完全通过服务层，就好像应用程序的API一样。

在提供一个清晰的API的同时，服务层也是放置事务控制和安全等功能的好场所。这样做可以使你获得一个简单的、包含了服务层所有方法并描述了其事务和安全特征的模型。通常，通过一个独立的特性文件来描述这一模型，但.NET中的属性值提供了一个直接在代码中进行描述的好方法。

如果设立了服务层，在其中置入行为的多少是一个至关重要的决定。最小化情况下，服务层只是一个外观，所有实际的行为都在下层的对象中，服务层所做的只是将上层调用传递到更低层。在这种情况下，服务层提供一个更易于使用的API，因为它的方法通常根据用例来组织。此时，它也提供一个很方便的切入点，用来增加事务封装和安全检查等功能。

另一个极端则是将大多数业务逻辑都以事务脚本的形式置于服务层中。下层的领域对象变得极为简单。如果下层是领域模型，则其中的对象与数据库一一对应，因而此时你就可以使用诸如活动记录等较简单的数据源层。

以上二者的折中是一个行为的混合体：控制器－实体风格（controller-entity style）。这一术语来源受到了［Jacobson et al.］的强烈影响。此处要点在于：将单个事务或用例所特有的逻辑置于事务脚本之中，它们通常被称为控制器或服务。有许多不同的控制器，如模型 - 视图 - 控制器中的输入控制器和我们稍后将会接触到的应用控制器。所以在此处我使用术语“用例控制器”（use-case controller）。可供多个用例调用的行为访问那些被称为实体的领域对象。

虽然控制器－实体方法很常用，但我一直不是很欣赏它。与事务脚本一样，用例控制器容易产生代码副本。我认为，如果你确实决定完全选用领域模型，就应当彻底贯彻这一模式，使它在应用程序中占主导地位。一个例外是你已经从采用了行数据入口的事务脚本开始了，那么可以将冗余行为移到行数据入口中，这样就将它转变成一个使用活动记录的简单领域模型。但是我不会一开始就这样做。我只会用这种办法来改进已有的存在缺陷的设计。

我并不是说绝不应设计包含了业务逻辑的服务层对象，而是说未必一定要将它们组成一个固定的层。过程化的服务层对象有时对细分逻辑是很有用的，但我偏向于只在必要时才使用这种对象，而不是将它们组织成一个架构中的层来使用。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

因此，我的建议是：如果你确实需要，尽可能使用最小化的服务层。我通常首先假定并不需要这么一层，然后当发现应用确实需要时才增加。但是，我也知道许多优秀的软件设计者总是使用一个包含了合理数量逻辑的服务层，因此读者也不要拘束于我的观点。Randy Stafford在复杂服务层的应用方面有许多成功的案例，这也是我为什么请他执笔本书的服务层模式部分的原因。