## 应用服务

Eric Evans 为运用领域驱动设计的系统架构划定了层次，在领域层和展现层之间引入了应用层（Application Layer）：“应用层要尽量简单，不包含业务规则或者知识，而只为下一层（指领域层）中的领域对象协调任务，分配工作，使它们互相协作。”我在讲解领域驱动架构的演进时，则认为领域层提供了细粒度的领域模型对象，不利于它的客户端调用。因此，“基于 KISS（Keep It Simple and Stupid）原则或最小知识原则，我们希望调用者了解的知识越少越好，调用变得越简单越好，这就需要引入一个间接的层来封装。这就是应用层存在的主要意义。”

### 应用服务的本质

应用服务是外观模式（Facade Pattern）的体现。经典著作《设计模式》定义了外观模式的意图：“为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。”这恰与引入应用服务的作用不谋而合。使用外观模式的场景主要包括：

* 当你要为一个复杂子系统提供一个简单接口时
* 客户程序与抽象类的实现部分之间存在着很大的依赖性
* 当你需要构建一个层次结构的子系统时，使用外观模式定义子系统中每层的入口点

这三个场景恰好说明了应用服务的本质。对外，应用服务为外部调用者提供了一个简单统一的接口，该接口为一个完整的用例场景提供了自给自足的功能，使得调用者无需求助于别的接口就能满足业务需求。对内，应用服务自身并不包含任何领域逻辑，仅负责协调领域模型对象，通过它们的领域能力来组合完成一个完整的应用目标。应用服务作为应用外观，仅仅是领域层的一个入口点，通过它可以降低客户程序与领域层实现之间的依赖。作为领域模型对象的包装，它自身不应该包含任何领域逻辑。由此可得到应用服务设计的第一条准则： **不包含领域逻辑的业务服务应被定义为应用服务。**

如果参考 Robert Martin 提出的整洁架构思想，领域驱动分层架构的应用层可对应整洁架构内核中的用例（Use Case）层。不过，领域驱动设计强调应用服务虽然对外表现了应用业务逻辑（Application Business Rule），但达成应用目标的实现逻辑需要分配给领域层的领域模型对象。

无论六边形架构还是整洁架构，都认为是网关（即六边形架构中的适配器）打通了内部领域核心与外部资源和框架的通道。网关封装了外部资源访问与框架依赖的实现逻辑，属于外部的基础设施层。北向网关属于外部依赖内部，南向网关则相反，属于内部依赖外部。因此，要让南向网关满足整洁架构思想，避免内部的领域逻辑依赖于外部的基础设施，就需要为南向网关引入抽象和依赖注入。

在领域驱动设计中，属于南向网关的资源库，其抽象常被视为领域层的一部分；不止于此，整个“南向网关”的抽象其实亦可视为组成领域层的一部分，例如访问第三方服务的 HttpClient，发送通知的抽象服务接口。考虑到分层与模块之间的关系，我在《领域驱动战略设计》中，给出了与领域驱动设计思想对应的代码模型。在这个代码模型中，我将网关分为了 interfaces 与 gateways 两个包，前者仅定义了网关的抽象，后者则提供对应的实现。对应到分层架构，网关的抽象归属于领域层，网关的实现归属于基础设施层。

在考虑业务逻辑与具体技术实现之间的协作时，可以将南向网关的抽象既注入到领域服务或应用服务。领域服务与南向网关抽象之间的协作关系属于同层之间的依赖，应用服务与南向网关抽象之间的协作属于外层调用内层，二者都没有违背整洁架构思想。这意味着，领域逻辑与技术实现的隔离和结合既可以在领域层完成，也可以在应用层完成；那么，应用服务除了能对细粒度的领域逻辑进行包装之外，它还能提供其余什么价值呢？

一个完整的业务用例场景，多数时候不仅限于领域逻辑，也不仅限于访问数据库或者其他第三方服务，往往还需要和如下逻辑进行协作：

* 消息验证
* 错误处理
* 监控
* 事务
* 认证与授权
* ……

《领域驱动设计模式、原理与实践》一书将以上内容视为 **基础架构问题** 。这些关注点与具体的领域逻辑无关，且在整个系统中，会作为重用模块被诸多服务调用。调用时，这些关注点是与领域逻辑交织在一起的，因此这些关注点都属于 **横切关注点** 。

从面向切面编程（Aspect-Oriented Programming，AOP）的角度看，所谓“横切关注点”就是那些在职责上是内聚的，但在使用上又会散布在所有对象层次中，且与所散布到的对象的核心功能毫无关系的关注点。与“横切关注点”对应的是“核心关注点”，就是与系统业务有关的领域逻辑。例如，订单业务是核心关注点，提交订单时的事务管理以及日志记录则是横切关注点：

public class OrderAppService {  
 @Service  
 private PlacingOrderService placingOrderService;  
  
 // 事务为横切关注点  
 @Transactional(propagation=Propagation.REQUIRED)   
 public void placeOrder(Order order) {   
 try {  
 orderService.execute(order);  
 } catch (InvalidOrderException ex | Exception ex) {  
 // 日志为横切关注点  
 logger.error(ex.getMessage());  
 // ApplicationException 派生自 RuntimeException，事务会在抛出该异常时回滚  
 throw new ApplicationException("failed to place order", ex);  
 }  
 }  
}

横切关注点与具体的业务无关，它与核心关注点在逻辑上应该是分离的。为保证领域逻辑的纯粹性，应尽量避免将横切关注点放在领域模型对象中。于是，应用服务就成了与横切关注点协作的最佳位置。由此，可以得到应用服务设计的第二条原则： **与横切关注点协作的服务应被定义为应用服务。**

### 应用服务与领域服务的选择

如前所述，应用服务不应该包含任何领域逻辑，同时，它又将作为一个外观服务，负责封装多个领域模型对象之间的协作。那么，将多个领域行为组合起来的协调行为，究竟算不算是领域逻辑呢？例如，对于“下订单”用例而言，如果我们在各自的领域对象中定义了如下行为：

* 验证订单是否有效
* 提交订单
* 移除购物车中已购商品
* 发送邮件通知买家

这些行为的组合正好满足了“下订单”这个完整用例的需求，同时也为了保证客户调用的简便性，我们需要协调这四个领域行为。这一协调行为牵涉到不同的领域对象，因此只能定义为服务。那么，这个服务应该是应用服务，还是领域服务？

《领域驱动设计模式、原理与实践》一书将这种封装认为是 **与领域的交互** 。该书作者给出了一个判断标准：

决定一系列交互是否属于领域的一种方式是，提出“这种情况总是会出现吗？”或者“这些步骤无法分开吗？”的问题。如果答案是肯定的，那么这看起来就是一个领域策略，因为那些步骤总是必须一起发生。然而，如果那些步骤可以用若干方式重新组合，那么可能它就不是一个领域概念。

我想，这一判断标准是基于“任务编制”得出的结论。如果领域逻辑的步骤必须一起发生，就说明这些逻辑不存在“任务编制”的可能，因为它们在本质上是一个整体，只是基于单一职责原则与分治原则，需要进行分解，做到对象的各司其职而已。如果领域步骤可以用若干方式重新组合，就意味着可以有多种方式进行“任务编制”。因此，任务编制逻辑就属于应用逻辑的范畴，编制的每个任务则属于领域逻辑的范畴，前者由应用服务来承担，后者由领域模型对象来承担。

Eric Evans 用另一种玄而又玄的说法印证了该判断标准： **“应用服务是协调者，它们只是负责提问，而不负责回答，回答是领域层的工作。”** 注意，对所谓“提问”和“回答”的理解，要站在一个完整用例场景的高度来阐释。当客户端发来请求要执行一个完整的用例场景时，作为协调者的应用服务只负责安排任务，至于任务该怎么做，就是领域模型对象要完成的工作。这实际上是业务价值（Why）与业务功能（What）之间的关系。对于一个用例场景，需要为参与者提供业务价值，该价值由应用服务提供；要实现这一业务价值，需要若干业务功能按照某种顺序进行组合，组合的顺序就是编制，编制的业务功能就是回答问题的领域模型对象。

要基于这一标准对应用服务与领域服务做出正确判断，更多地还是依靠你对设计的感觉。因为价值与功能在不同的层次会产生一种层层递进的递归关系。例如下订单是业务价值，验证订单就是实现该业务价值的业务功能；然而再进一层，又可以将验证订单视为业务价值，而将验证订单的配送地址有效性作为实现该业务价值的业务功能。至于前面提到的“任务编制”，其实也存在歧义，即使在领域服务中，也存在任务编制的可能，这实际取决于你对任务层次的定位。这还真是剪不断理还乱了。

让我们回归本质，回到对“领域”这个词的理解。在领域驱动设计这个大背景下，领域其实与软件系统服务的行业有关，如金融行业、制造行业、医疗行业、教育行业等。在领域驱动设计的战略阶段，又将整个系统的领域分解为核心领域与子领域，它们解决的是不同的问题域。在解决方案域，应用服务和领域服务都属于一个具体的限界上下文，它们又必然映射到问题域中某一个子领域上。由此可得到一个推论：领域逻辑就是对应子领域包含的业务知识和业务规则，应用逻辑则是为了完成完整用例而包含的除领域逻辑之外的其他业务逻辑，包括作为基础架构问题的横切关注点，也可能包含对非领域知识相关的处理逻辑，如对输入、输出格式的转换等。

Eric Evans 用银行转账的案例来讲解应用逻辑与领域逻辑的差异。他说：“资金转账在银行领域语言中是一项有意义的操作，而且它涉及基本的业务逻辑。”这就说明资金转账属于领域逻辑。至于应用服务该做什么，他又说道：“如果银行应用程序可以把我们的交易进行转换并导出到一个电子表格文件中，以便进行分析，那么这个导出操作就是应用服务。‘文件格式’在银行领域中是没有意义的，它也不涉及业务规则。”

因此，到底选择应用服务还是领域服务，就看它的实现中到底是应用逻辑的范畴，还是领域逻辑的范畴。一个简单的判断标准在于这段代码蕴含的知识是否与它所处的限界上下文要解决的问题域直接有关？如此说来，针对“下订单”用例而言，在前面列出的四个领域行为中，只有“发送邮件”与购买子领域没有关系，因此可考虑将其作为要编制的任务放到应用服务中。如此推导出来的订单应用服务实现为：

public class OrderAppService {  
 @Service  
 private PlacingOrderService placingOrderService;  
  
 // 此时将 NotificationService 视为基础设施服务  
 @Service  
 private NotificationService notificationService;  
  
 // 事务为横切关注点  
 @Transactional(propagation=Propagation.REQUIRED)   
 public void placeOrder(Order order) {   
 try {  
 orderService.execute(order);  
 notificationService.send(notificationComposer.compose(order));  
 } catch (InvalidOrderException ex | Exception ex) {  
 // 日志为横切关注点  
 logger.error(ex.getMessage());  
 // ApplicationException 派生自 RuntimeException，事务会在抛出该异常时回滚  
 throw new ApplicationException("failed to place order", ex);  
 }  
 }  
}

即使如此，应用逻辑与领域逻辑的边界线依旧微妙难分。

我注意到《领域驱动设计》中的两段描述。其一：

很多领域服务或应用服务是在实体和值对象的基础上建立起来的，它们的行为类似于将领域的一些潜在功能组织起来以 **执行某种任务的脚本** 。实体和值对象往往由于粒度过细而无法提供对领域层功能的便捷访问。

其二：

在大型系统中，中等粒度的、无状态的服务更容易被复用，因为它们在简单的接口背后封装了重要的功能。……由于应用层负责对领域对象的行为进行协调，因此 **细粒度的领域对象可能会把领域层的知识泄露到应用层中** 。这产生的结果是应用层不得不处理复杂的、细致的交互，从而使得领域知识蔓延到应用层或用户界面代码当中，而领域层会丢失这些知识。 **明智地引入领域服务有助于在应用层和领域层之间保持一条明确的界限。**

综合这两段话，我们可以隐约探索到分辨应用服务与领域服务的真相。第一段提到“实体和值对象往往由于粒度过细而无法提供对领域层功能的便捷访问”，第二段又提到“细粒度的领域对象可能会把领域层的知识泄露到应用层中”，无论从隐藏细节的角度，还是从便捷访问的角度，在领域层，领域服务都成了当仁不让的最佳选择。

而在第一段中，Eric Evans 又说应用服务和领域服务都是“执行某种任务的脚本”。任务脚本可以理解为对任务的编制，只是应用服务和领域服务处理的任务层级并不相同罢了。再结合第二段的最后一句“明智地引入领域服务有助于在应用层和领域层之间保持一条明确的界限”，我们有理由得到如下结论：

* 细粒度的领域对象包括实体、值对象以及领域服务，但为了避免领域层知识泄漏到应用层中，应在领域层定义中等粒度的领域服务，它的实现可以认为是对细粒度领域服务、聚合的任务编制
* 理想状态下，应用服务应该只与中等粒度的领域服务协作，它对任务的编制，实则就是对领域服务的编制

若同意这一结论， **说明应用服务中只能包含两部分内容：领域服务、横切关注点。** 如此设计自然逃脱不了僵化的嫌疑，但殊不知我是在 **为设计做减法** 。若设计者能够充分辨别应用逻辑与领域逻辑之间的差别，突破这一约束也未尝不可。一旦你拥有了足够丰富的设计知识和设计经验，就意味着你可以正确地做出适合当前场景的设计决策与判断。若无法做到，不妨从一些相对固化的简单原则开始做起，这算是从新手到专家所必须经历的成长过程。

### 影响应用服务的因素

一旦对应用服务的设计进行了约束，要分辨应用服务和领域服务的区别就变得容易了许多。然而，软件设计就是这样，当你因为某种干扰因素而做出一种设计决策时，在消除了这一干扰因素的同时，另外一些原来不曾显现的干扰因素又可能浮现出来。既然应用服务的实现代码只能包含横切关注点，也只能与领域层的领域服务协作，那就需要我们对横切关注点做出正确判断，同时还需要明确领域服务的设计粒度。

#### 横切关注点的判断

要判断一个服务是否为应用服务，需要明确什么是“横切关注点”。前面已经明确给出了“横切关注点”的定义，但是，在判断横切关注点以及整合横切关注点时，除了前面提到的事务、监控、身份验证与授权没有争议之外，社区对如下关注点普遍存在困惑与纠结。

**日志**

毫无疑问，日志属于横切关注点的范畴。然而，倘若将日志功能仅仅放在应用层，又可能无法准确详细地记录操作行为与错误信息。很多语言都提供了基础的日志框架，将日志混杂在领域对象中，会影响领域的纯粹性，也带来了系统与日志框架的耦合，除非采用 AOP 的方式。目前看来，这是一种编码取舍，即倾向于代码的纯粹性，还是代码的高质量。我个人更看重代码的质量，尤其是丰富的日志内容有助于运维排错，因此可考虑将作为横切关注点之一的日志功能放在领域服务中，算是上述应用服务边界定义的特例。

当然，这个划分并非排他性的。在应用服务中，同样需要调用日志功能，只是记录的信息与粒度和领域服务不尽相同罢了。

**验证**

如果是验证外部客户传递过来的消息，例如对 RESTful 服务的 Request 请求的验证，则该验证功能属于横切关注点，对它的调用就应该放在应用服务（亦可考虑由远程服务自己承担）。如果验证逻辑属于一种业务规则，例如验证订单有效性，就应该将验证逻辑放在领域层，以便于领域模型对象调用。

**异常处理**

与领域逻辑有关的错误与异常，应该以自定义异常形式表达业务含义，并被定义在领域层。此外，如果该异常表达了业务含义，为了保证业务的健壮性，可在领域层中将异常定义为受控异常（Checked Exception）。由于该异常与业务有关，即使被定义在方法接口中，也不存在异常对接口的污染，即可以将异常视为接口契约的一部分。但是，在领域服务中，不应该将与业务无关的受控异常定义在领域服务的方法中，否则就会导致业务逻辑与技术实现的混合。

在应用层，应尽可能保证应用服务的通用性，因而需要在应用服务中捕获与业务有关的自定义异常，然后将其转换为标准格式的异常之后再抛出。例如，可统一定义为应用层的标准异常 ApplicationException，然后在 message 或 cause 中包含具体的业务含义。因此，针对异常处理，只有这部分与业务无关的处理与转换功能，才属于横切关注点的范畴，并放在应用层，其余异常处理逻辑都属于领域层。

**基础设施服务**

除了上述纠结的横切关注点之外，我们还要注意基础设施服务与横切关注点之间的区别。在领域驱动设计中，基础设施服务作为技术服务，被定义为网关。从代码实现的角度考虑，南向网关代表了一个内聚的技术实现，可以被抽象为接口；横切关注点则是一些钩子方法，会在领域行为方法的前后被执行，因此难以抽象为接口。显然，基础设施服务就像提供的其他基础功能一般，可以很容易被重用，而横切关注点由于会和领域逻辑纠缠在一起，很难剥离出单独的横切关注点代码，除非采用面向切面编程。

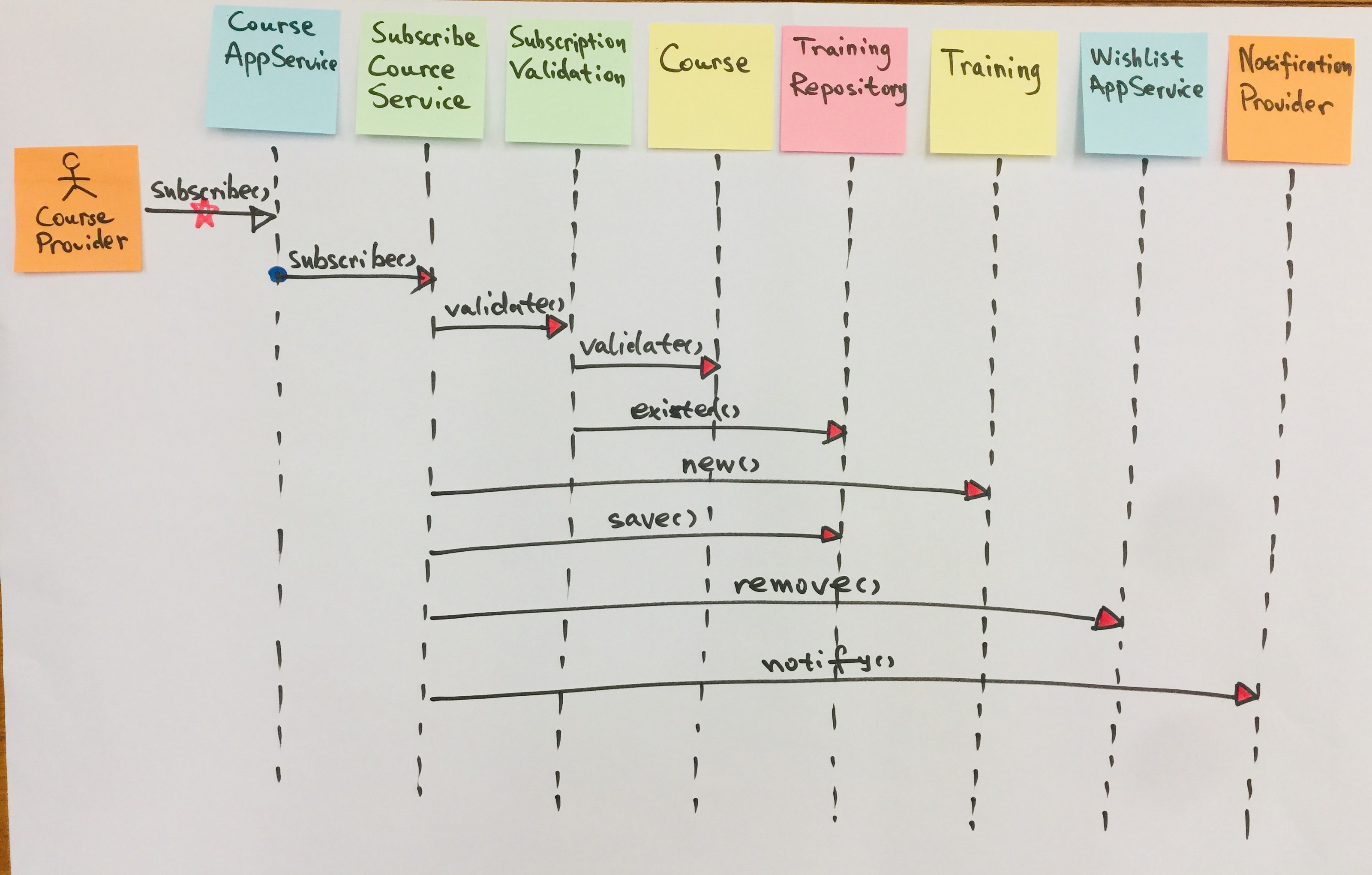
遵循应用服务的设计原则，它除了和领域服务进行协作之外，就只是包含了横切关注点，这就说明应用服务甚至都不应该依赖于提供基础设施服务的南向网关。这样的设计约束充分保证了应用服务的简单性。因此，只要判断某个逻辑属于基础设施服务，就应该首先考虑与领域服务协作，而非应用服务。例如，邮件通知服务就属于典型的基础设施服务。既然如此，针对订单应用服务的实现，就应该将通知服务转移到 PlacingOrderService 领域服务中。事实上，在前面修改后的订单应用服务代码中，代码 notificationComposer.compose(order) 放在应用服务中本身也不太合适，因为将订单内容转换为邮件通知内容，更像是领域逻辑，而非应用逻辑。

#### 领域服务的设计粒度

在领域层中，为了保证聚合内部实体与值对象的纯粹性，我们将与外部资源抽象之间的协作推给了领域服务；为了避免出现贫血模型和过程式的事务脚本，我们要求定义带有动词的领域服务，使得领域服务在正确表达领域行为特征的同时，粒度也变得更细。这时的领域服务其本质更像是一个函数，没有状态，单一职责，体现的是领域逻辑的行为特征。

但是，在面向对象设计中，粒度大小与简单设计需要平衡。若要二者兼得，需要在细粒度对象之上再引入一层封装：一边是纷繁的实现细节，一边是干净利落的接口。这正是引入 **中等粒度领域服务** 的由来。中等粒度的领域服务实质上是对更细粒度的领域模型对象之间的流程编制，它的主要作用在于协调多个领域对象，尤其是多个细粒度领域服务之间的协作。

还记得《理解领域模型》一节给出的“订阅课程”业务场景的案例吗？当时我以可视化的时序图方式给出了各个对象角色之间的协作关系：



79fc89bc-2ae8-4074-b9b5-4c09f2046b6c.jpg

显然，图中蓝色的应用服务 CouseAppService 划定了一条远程服务与领域层之间的界限，使得远程服务无需了解课程订阅领域逻辑的实现细节。课程与期望列表属于两个不同的限界上下文，但它们又都处于同一个进程边界内，因此它们之间的协作通过应用服务来完成。领域服务 SubscriptionValidation 仅仅实现了对订阅的验证功能。它是一个细粒度服务，部分验证的逻辑委派给了 Course 聚合，避免了贫血模型。持久化与邮件通知都属于基础设施服务，分别由资源库和邮件通知南向网关完成。

领域服务 SubscribeCourseService 并没有履行具体的业务职责，它只是将多个领域对象组合起来，进行业务流程的编制。观察时序图，你会发现由该服务发出的方法调用是最多的。这就是所谓的 **中等粒度领域服务** 。它在应用层和领域层之间划定了一条明确的界限，也使得应用服务 CouseAppService 得偿所愿，成为一个没有领域逻辑的外观服务。采用时序图的可视化方式，可以观察应用服务发起的调用，即图中涂为深蓝色的地方。很明显，应用服务发起的调用越少，包含领域逻辑的可能性就越小。

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