## 【案例】层次的职责与协作关系（图文篇）

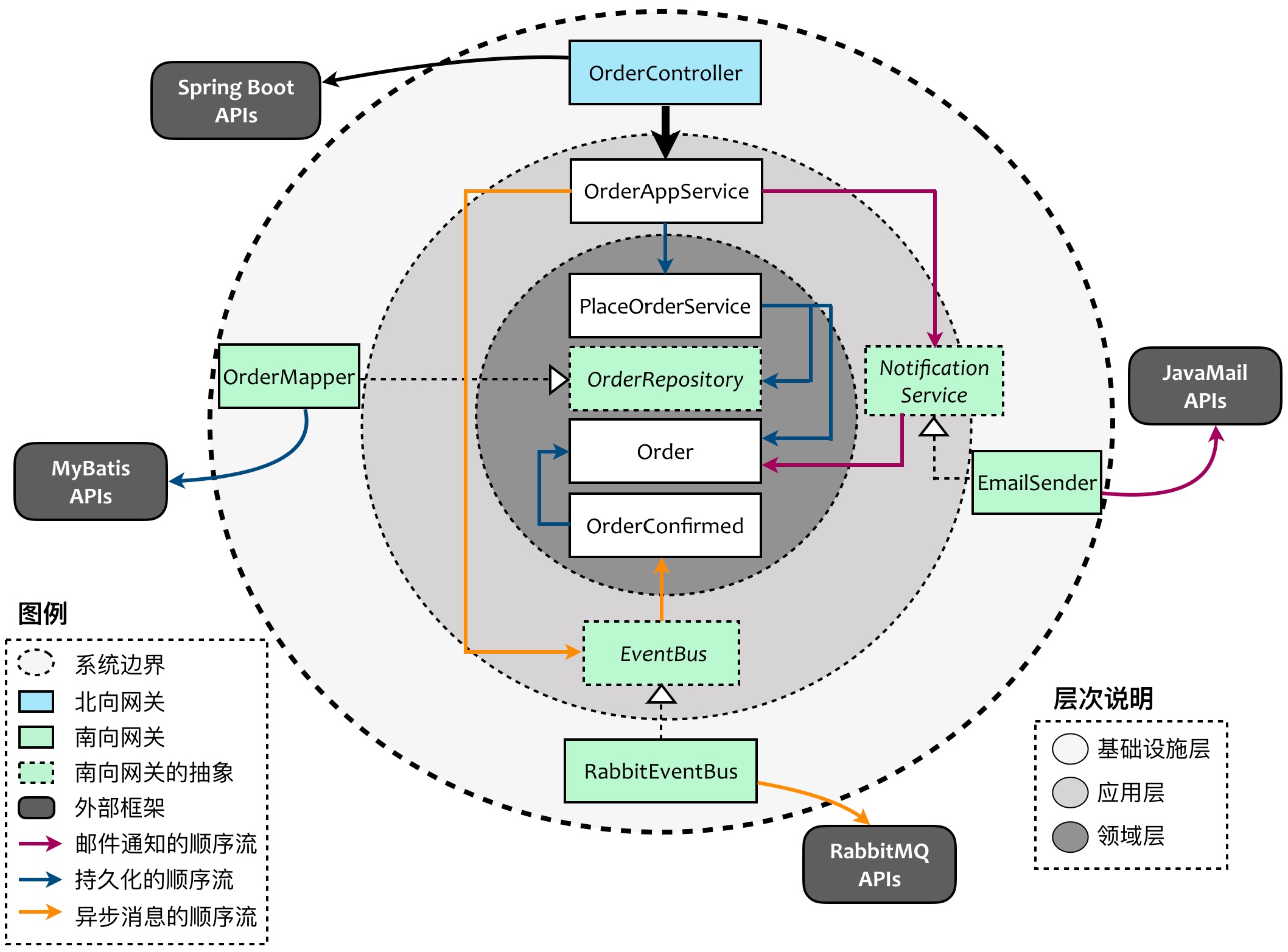
经历多次演进，我们已经初步得到了符合领域驱动设计思想的分层架构，但这种架构仅仅是一种静态的逻辑划分，在实现一个业务用例时，各层之间是怎么协作的，我们似乎还不得而知。辨别这种动态的协作关系，还是应该从职责的角度入手，即必须清楚地理解分层架构中每个逻辑层的职责。

一味的理论讲解可能已经让爱好案例与代码的程序员昏昏欲睡了，何况用纯理论知识来讲解职责与协作，也让我力有未逮。不如通过一个具体案例，来说明层次的职责以及层次之间的协作关系。还是以电商系统的下订单场景为例，在买家提交订单时，除了与订单直接有关的业务之外，还需要执行以下操作。

* 订单数据的持久化：OrderRepository 提供插入订单功能。它属于支撑订单提交业务的基础功能，但将订单持久化到数据库的实现 OrderMapper 并不属于该业务范畴。
* 发送通知邮件：NotificationService 提供通知服务。它属于支撑通知业务的基础功能，但邮件发送的实现 EmailSender 却不属于该业务范畴。
* 异步发送消息给仓储系统：提交订单成功后需要异步发送消息 OrderConfirmed 给仓储系统，这一通信模式是通过消息队列来完成的。EventBus 发送 OrderConfirmed 事件属于支撑订单提交成功的基础功能，但发送该事件到 RabbitMQ 消息队列的 RabbitEventBus 则不属于该业务范畴。

同时，为了用户界面客户端或第三方服务的分布式调用，需要通过 OrderController 暴露 RESTful 服务。它本身不提供任何业务实现，而是通过将请求委派给应用层的 OrderAppService 来完成订单的提交。

下图体现了前述三个流程在各层之间以及系统内外部之间的协作关系。 **注意，在这里我将牵涉到的类型放在了同一个限界上下文中，如果牵涉到多个限界上下文之间的协作，实现会略有不同，对应的代码模型也将有所调整** 。我会在后续内容中深入探讨限界上下文之间的协作对代码模型的影响。



enter image description here

基础设施层的 OrderController 扮演了北向网关的角色，承担了与用户界面层或第三方服务交互的进出口职责。它通过 Spring Boot 来完成对 HTTP 请求的响应、路由和请求/响应消息的序列化与反序列化。它的自有职责仅仅是对请求/响应消息的验证，以及对 OrderAppService 的调用。或许有人会质疑处于后端顶层的控制器为何属于基础设施层？但我认为这样的分配是合理的，因为 Controller 要做的事情与基础设施层所要履行的职责完全匹配，即它提供的是 REST 服务的基础功能。

基础设施层南向网关包括 OrderMapper、EmailSender 和 RabbitEventBus，它们对内为具体的某个业务提供支撑功能，对外则需要借助框架或驱动器访问外部资源。与北向网关不同，对它们的调用由属于内层的应用服务 OrderAppService 发起，因此需要为它们建立抽象来解除内层对外层的依赖。前面已经分析，由于 Repository 提供的方法分属领域逻辑，故而将 OrderMapper 所要实现的接口 OrderRepository 放到核心的领域层。至于 EmailSender 与 RabbitEventBus 各自的抽象 NotificationService 与 EventBus 并未代表领域逻辑，为了不污染领域层的纯洁性，放在应用层似乎更为合理。

无论是北向网关还是南向网关，它们都要与外部资源进行协作，不管是对内/外协议的适配，还是对内部协作对象的封装，本质上它们只做与业务直接有关的基础功能。真正与业务无关的通用基础功能，是与具体某个软件系统无关的，属于更加基础和通用的框架。例如，OrderController 调用的 Spring Boot APIs，EmailSender 调用的 JavaMail APIs、OrderMapper 调用的 MyBatis APIs 以及 RabbitEventBus 调用的 RabbitMQ APIs，都是这样的通用框架。它们是系统代码边界外的外部框架，通常为第三方的开源框架或商业产品；即使是团队自行研发，也不应该属于当前业务系统的代码模型。

我们可以基于这个案例归纳各个层次的职责。

* 领域层：包含 PlaceOrderService、Order、Notification、OrderConfirmed 与抽象的 OrderRepository，封装了纯粹的业务逻辑，不掺杂任何与业务无关的技术实现。
* 应用层：包含 OrderAppService 以及抽象的 EventBus 与 NotificationService，提供对外体现业务价值的统一接口，同时还包含了基础设施功能的抽象接口。
* 基础设施层：包含 OrderMapper、RabbitEventBus 与 EmailSender，为业务实现提供对应的技术功能支撑，但真正的基础设施访问则委派给系统边界之外的外部框架或驱动器。

**注意** ：这里定义了两个分属不同层次的服务，二者极容易混淆。PlaceOrderService 是领域服务，定义在领域层中；OrderAppService 是应用服务，定义在应用层中。这二者的区别属于战术设计的层面，我会在之后的战术设计讲解中深入阐释，我的博客《[如何分辨应用服务与领域服务](http://zhangyi.xyz/how-%20to-identify-application-service/)》也有比较详细的介绍。

OrderController 的实现代码如下所示：

package practiceddd.ecommerce.ordercontext.infrastucture;  
  
import org.springframework.web.bind.annotation.RestController;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.beans.factory.annotation.Autowired;  
  
import practiceddd.ecommerce.ordercontext.infrastructure.message.CreateOrderRequest;  
import practiceddd.ecommerce.ordercontext.application.OrderAppService;  
import practiceddd.ecommerce.ordercontext.domain.Order;  
  
@RestController  
@RequestMapping(value = "/orders/")  
public class OrderController {  
 @Autowired  
 private OrderAppService service;  
  
 @RequestMapping(method = RequestMethod.POST)  
 public void create(@RequestParam(value = "request", required = true) CreateOrderRequest request) {  
 if (request.isInvalid()) {  
 throw new BadRequestException("the request of placing order is invalid.");  
 }  
 Order order = request.toOrder();  
 service.placeOrder(order);  
 }  
}

应用服务 OrderAppService 的代码如下所示：

package practiceddd.ecommerce.ordercontext.application;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.domain.PlaceOrderService;  
import practiceddd.ecommerce.ordercontext.domain.Order;  
import practiceddd.ecommerce.ordercontext.domain.OrderCompleted;  
import practiceddd.ecommerce.ordercontext.domain.Notification;  
import practiceddd.ecommerce.ordercontext.domain.OrderNotification;  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
  
@Serivce  
public class OrderAppService {  
 @Autowired  
 private NotificationService notificationService;  
 @Autowired  
 private EventBus eventBus;  
 @Autowired  
 private PlaceOrderService placeOrderService;  
  
 public void placeOrder(Order order) {  
 try {  
 placeOrderService.execute(order);  
 notificatonService.send(composeNotification(order));  
 eventBus.publish(composeEvent(order));  
 } catch (InvalidOrderException | Exception ex) {  
 throw new ApplicationException(ex.getMessage());  
 }  
 }  
  
 private Notification composeNotification(Order order) {  
 // 组装通知邮件的内容，实现略  
 }  
 private OrderConfirmed composeEvent(Order order) {  
 // 组装订单确认事件的内容，实现略  
 }  
}

既然 OrderAppService 属于应用层的应用服务，它就不应该包含具体的业务逻辑。倘若我们将发送邮件和异步消息发送视为“横切关注点”，那么在应用服务中调用它们是合乎情理的；然而，通过 Order 组装 Notification 与 OrderConfirmed 的职责，却应该放在领域层，因为基于订单去生成邮件内容以及发布事件包含了业务逻辑与规则。 **问题出现！** 由于这两个对象是由领域层生成的对象，我们该 **如何将领域层生成的对象传递给处于它之上的应用层对象？**

有三种解决方案可供选择。

第一种方案是将组装通知邮件与订单确认事件的职责封装到领域层的相关类中，然后在应用层调用这些类的方法，如此可以减少应用层的领域逻辑：

package practiceddd.ecommerce.ordercontext.domain;  
import org.springframework.stereotype.Service;  
  
@Service  
public class NotificationComposer {  
 public Notification compose(Order order) {  
 // 实现略  
 }  
}  
  
package practiceddd.ecommerce.ordercontext.domain;  
import org.springframework.stereotype.Service;  
  
@Service  
public class OrderConfirmedComposer {  
 public OrderConfirmed compose(Order order) {  
 // 实现略  
 }  
}

则应用服务就可以简化为：

package practiceddd.ecommerce.ordercontext.application;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.domain.PlaceOrderService;  
import practiceddd.ecommerce.ordercontext.domain.Order;  
import practiceddd.ecommerce.ordercontext.domain.OrderConfirmed;  
import practiceddd.ecommerce.ordercontext.domain.Notification;  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
import practiceddd.ecommerce.ordercontext.domain.NotificationComposer;  
import practiceddd.ecommerce.ordercontext.domain.OrderConfirmedComposer;  
  
@Service  
public class OrderAppService {  
 @Autowired  
 private NotificationService notificationService;  
 @Autowired  
 private EventBus eventBus;  
 @Autowired  
 private PlaceOrderService placeOrderService;  
 @Autowired  
 private NotificationComposer notificationComposer;  
 @Autowired  
 private OrderConfirmedComposer orderConfirmedComposer;  
  
 public void placeOrder(Order order) {  
 try {  
 placeOrderService.execute(order);  
 notificatonService.send(notificationComposer.compose(order));  
 eventBus.publish(orderConfirmedComposer.compose(order));  
 } catch (InvalidOrderException | Exception ex) {  
 throw new ApplicationException(ex.getMessage());  
 }  
 }  
}

采用这种方案的代码结构如下所示：

ordercontext.infrastructure  
 - OrderController  
 - OrderMapper  
 - EmailSender  
 - RabbitEventBus  
ordercontext.application  
 - OrderAppService  
 - NotificationService  
 - EventBus  
ordercontext.domain  
 - OrderRepository  
 - PlaceOrderService  
 - Order  
 - Notification  
 - OrderConfirmed  
 - NotificationComposer  
 - OrderConfirmedComposer

第二种方案则将“上层对下层的调用”改为“下层对上层的通知”，即前面讲解层之间协作时所谓“自底向上”的通信问题，这就需要在领域层为订单业务定义 OrderEventPublisher 接口。当满足某个条件时，通过它在领域层发布事件，这个事件即所谓“领域事件（Domain Event）”。如果我们将建模的视角切换到以“事件”为中心，则意味着领域服务在下订单完成后，需要分别发布 NotificationComposed 与 OrderConfirmed 事件，并由应用层的 OrderEventHandler 作为各自事件的订阅者。这里的前提是：发送邮件与异步发送通知属于应用逻辑的一部分。

我们需要先在领域层定义发布者接口：

package practiceddd.ecommerce.ordercontext.domain;  
  
public interface OrderEventPublisher {  
 void publish(NotificationComposed event);  
 void publish(OrderConfirmed event);  
}

实现 OrderEventPublisher 接口的类放在应用层：

package practiceddd.ecommerce.ordercontext.application;  
  
import practiceddd.ecommerce.ordercontext.domain.OrderEventPublisher;  
import practiceddd.ecommerce.ordercontext.domain.NotificationComposed;  
import practiceddd.ecommerce.ordercontext.domain.Notification;  
import practiceddd.ecommerce.ordercontext.domain.OrderConfirmed;  
  
public class OrderEventHandler implements OrderEventPublisher {  
 private NotificationService notificationService;  
 private EventBus eventBus;  
  
 public OrderEventHandler(NotificationService notificationService, EventBus eventBus) {  
 this.notificationService = notificationService;  
 this.eventBus = eventBus;  
 }  
  
 public void publish(NotificationComposed event) {  
 notificationService.send(event.notification());  
 }  
  
 public void publish(OrderConfirmed event) {  
 eventBus.publish(event);  
 }  
}

应用层的应用服务则修改为：

package practiceddd.ecommerce.ordercontext.application;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.domain.PlaceOrderService;  
import practiceddd.ecommerce.ordercontext.domain.Order;  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
  
@Service  
public class OrderAppService {  
 @Autowired  
 private PlaceOrderService placeOrderService;  
 @Autowired  
 private NotificationService notificationService;  
 @Autowired  
 private EventBus eventBus;  
  
 public void placeOrder(Order order) {  
 try {  
 placeOrderService.register(new OrderEventHandler(notificationService, eventBus));  
 placeOrderService.execute(order);  
 } catch (InvalidOrderException ex) {  
 throw new ApplicationException(ex.getMessage());  
 }  
 }  
}

领域服务修改为：

package practiceddd.ecommerce.ordercontext.domain;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
  
@Service  
public class PlaceOrderService {  
 @Autowired  
 private OrderRepository orderRepository;  
 @Autowired  
 private NotificationComposer notificationComposer;  
  
 private OrderEventPublisher publisher;  
  
 public void register(OrderEventPublisher publisher) {  
 this.publisher = publisher;  
 }  
  
 public void execute(Order order) {  
 if (!order.isValid()) {  
 throw new InvalidOrderException(String.format("The order with id %s is invalid.", order.id()));  
 }  
 orderRepository.save(order);  
 fireNotificationComposedEvent(order);  
 fireOrderConfirmedEvent(order);  
 }  
  
 private void fireNotificationComposedEvent(Order order) {  
 Notification notification = notificationComposer.compose(order);  
 publisher.publish(new NotificationComposed(notification));  
 }  
 private void fireOrderConfirmedEvent(Order order) {  
 publisher.publish(new OrderConfirmed(order));  
 }  
}

倘若采用这种方案，则代码结构如下所示：

ordercontext.infrastructure  
 - OrderController  
 - OrderMapper  
 - EmailSender  
 - RabbitEventBus  
ordercontext.application  
 - OrderAppService  
 - NotificationService  
 - EventBus  
 - OrderEventHandler  
ordercontext.domain  
 - OrderRepository  
 - PlaceOrderService  
 - NotificationComposer  
 - OrderEventPublisher  
 - Order  
 - OrderConfirmed  
 - NotificationComposed

第三种方案需要重新分配 NotificationService 与 EventBus，将这两个抽象接口放到单独的一个名为 interfaces 的包中，这个 interfaces 包既不属于应用层，又不属于领域层。在后面讲解代码模型时，我会解释这样设计的原因，详细内容请移步阅读后面的章节。

通过这样的职责分配后，业务逻辑发生了转移，发送邮件与异步发送通知的调用不再放到应用服务 OrderAppService 中，而是封装到了 PlaceOrderService 领域服务。这时，应用服务 OrderAppService 的实现也变得更加简单。看起来，修改后的设计似乎更符合领域驱动分层架构对应用层的定义，即“应用层是很薄的一层，不包含业务逻辑”。这里的前提是： **发送邮件与异步发送通知属于业务逻辑的一部分。**

应用服务的定义如下所示：

package practiceddd.ecommerce.ordercontext.application;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.domain.PlaceOrderService;  
import practiceddd.ecommerce.ordercontext.domain.Order;  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
  
@Service  
public class OrderAppService {  
 @Autowired  
 private PlaceOrderService placeOrderService;  
  
 public void placeOrder(Order order) {  
 try {  
 placeOrderService.execute(order);  
 } catch (InvalidOrderException | Exception ex) {  
 throw new ApplicationException(ex.getMessage());  
 }  
 }  
}

不过，领域服务就变得不太纯粹了：

package practiceddd.ecommerce.ordercontext.domain;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import practiceddd.ecommerce.ordercontext.interfaces.NotificationService;  
import practiceddd.ecommerce.ordercontext.interfaces.EventBus;  
import practiceddd.ecommerce.ordercontext.domain.exceptions.InvalidOrderException;  
  
@Service  
public class PlaceOrderService {  
 @Autowired  
 private NotificationService notificationService;  
 @Autowired  
 private EventBus eventBus;  
 @Autowired  
 private OrderRepository orderRepository;  
 @Autowired  
 private NotificationComposer notificationComposer;  
  
 public void execute(Order order) {  
 if (!order.isValid()) {  
 throw new InvalidOrderException(String.format("The order with id %s is invalid.", order.id()));  
 }  
 orderRepository.save(order);  
 notificatonService.send(notificationComposer.compose(order));  
 eventBus.publish(new OrderConfirmed(order));  
 }  
}

代码结构如下所示：

ordercontext.infrastructure  
 - OrderController  
 - OrderMapper  
 - EmailSender  
 - RabbitEventBus  
ordercontext.application  
 - OrderAppService  
ordercontext.interfaces  
 - NotificationService  
 - EventBus  
ordercontext.domain  
 - OrderRepository  
 - PlaceOrderService  
 - Order  
 - OrderConfirmed  
 - Notification  
 - NotificationComposer

这三个方案该如何选择？根本的出发点在于你对业务逻辑和应用逻辑的认知，进而是你对领域服务与应用服务的认知，这些内容，就留待战术设计部分来讨论。由于并不存在绝对完美的正确答案，因此我的建议是在满足功能需求与松散耦合的前提下，请尽量选择 **更简单** 的方案。

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