## 【实践】EAS 系统的代码模型

在领域驱动战略设计的指导下，一个系统的逻辑架构应分为两个层次：系统层次与限界上下文层次。系统层次定义了整个系统所有限界上下文都可能调用的领域内核与基础设施公共组件，然后再以限界上下文为边界，结合领域复杂度决定每个限界上下文的逻辑架构。如果将限界上下文定义为微服务，还需要考虑在零共享架构下，如何实现限界上下文之间的通信与集成。

在 EAS 的战略设计阶段，识别了 EAS 的限界上下文以及上下文映射之后，就可以初步确定系统的整体架构。具体内容请参阅《领域驱动战略设计实践》第 34 章《实践：EAS 的整体架构》。在编码实践上，从一开始，各个限界上下文的开发团队都需要就这一架构达成一致认识，然后形成 EAS 系统和各自限界上下文的代码模型。

建立统一的代码模型固然重要，但更重要的是让团队每位成员认识到这一代码模型的设计原因，即明确每一逻辑层的职责、模块的划分原理、类的分配规范以及层与层之间的依赖关系，否则，再清晰的代码模型也会随着功能增加与时间推移而逐渐腐化，最后达至无可挽回的境地。要一直保证代码模型的清晰，需要遵从整洁架构的设计思想，理解领域驱动设计分层架构的意义，遵守领域驱动设计的纪律，而在编码落地时，则需要遵循领域模型驱动设计的过程。

### 面向领域场景的领域模型驱动设计

领域模型驱动的设计与编码是两条不同的主线。设计的过程可以结合服务模型驱动设计与领域模型驱动设计。在确定了限界上下文之后，可以通过上下文映射确定每个限界上下文需要暴露的服务，并为其定义服务接口。然后自顶向下，从远程服务到应用服务，然后结合领域场景进行场景驱动设计，确定组成领域场景的任务与角色构造型。编码的过程则反其道而行之，以划分了组合任务与原子任务的领域场景为基础，选择为承担原子任务的聚合编写单元测试，然后自下而上开展测试驱动开发，从领域层的聚合到领域服务，最后到应用服务与远程服务，交汇于服务接口的设计与实现。

整个设计与编码的过程都要围绕着领域场景进行。一个领域场景对外暴露了远程服务接口和应用服务接口，对内，形成了领域服务、聚合、资源库及其他南向网关之间的协作。在面向领域场景开始编码实现时，需要时刻谨记 **以领域来驱动开发，抑制即刻编写基础设施代码的冲动** 。如此，即可“强迫”开发人员尝试去理解业务逻辑，设计领域模型对象，并在统一语言的指导下编写代码。到实现应用服务这一交汇点时，才去考虑如何将资源库的实现注入到应用服务，如何实现事务和其他横切关注点，并且编写集成测试来验证整体实现是否满足领域场景的要求。

#### **领域场景的设计方向**

以 EAS 培训上下文的“提名候选人”领域场景为例。设计的方向是从领域分析建模到领域设计建模，依次获得的产出物包括：

* 体现了领域概念的领域分析模型
* 识别了角色构造型的领域设计模型
* 领域场景的用户故事
* 分解的任务与时序图脚本

领域分析模型产生于项目开始的先启阶段，也可以在迭代过程中召集整个特性团队就现有需求开展领域分析建模，从而获得限界上下文的领域分析模型。领域设计模型在迭代阶段获得，主要的参与者是特性团队的开发人员。同时，特性团队的需求分析人员与测试人员开始编写用户故事。开发人员领取用户故事后，开始分解任务，编写时序图脚本。编写好的时序图脚本可以附在用户故事之后，作为领域模型的一部分，通过需求管理工具管理起来。培训上下文“提名候选人”领域场景的用户故事如 [GitHub 的 Issue](https://github.com/agiledon/eas-ddd/issues/14) 所示。

#### **领域场景的编码方向**

在领域实现建模阶段，首先针对不访问外部资源的原子任务进行测试驱动开发，以获得聚合（包括聚合内实体、值对象）的测试代码与产品代码。待该领域场景的实体与值对象在单元测试的保护下实现了各自功能后，再以此为基础对组合任务进行测试驱动开发，从而驱动出领域服务的测试代码与产品代码。

对于“提名候选人”领域场景，在完成原子任务与组合任务的编写后，之前拆分的任务完成情况如下所示：

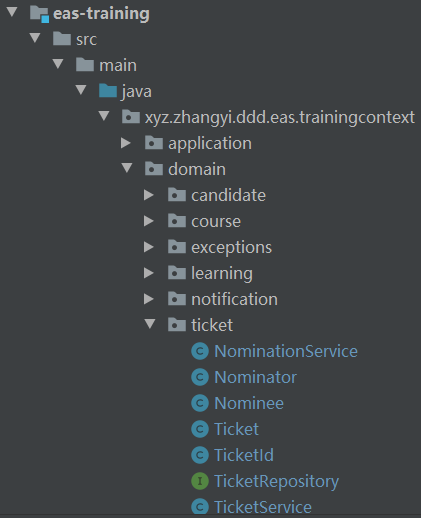
* 提名候选人（领域场景）
  + ~~提名候选人~~
    - ~~确定候选人是否已经参加过该课程~~
      * 获取该培训对应的课程
      * 确定课程学习记录是否有该候选人
    - ~~如果未参加，则提名候选人~~
      * 获得培训票
      * ~~提名~~
      * 保存票的状态
      * 添加票的历史记录
      * 将获得票的员工移出候选人名单
    - ~~发送提名通知~~
      * 获取通知邮件模板
      * ~~组装提名通知内容~~
      * 发送通知

在任务列表中，未完成的原子任务皆与外部资源有关，由角色构造型的 Repository 或 Client 承担，它们的接口定义可以通过为领域服务编写单元测试时驱动出来。

#### 领域层的代码模型

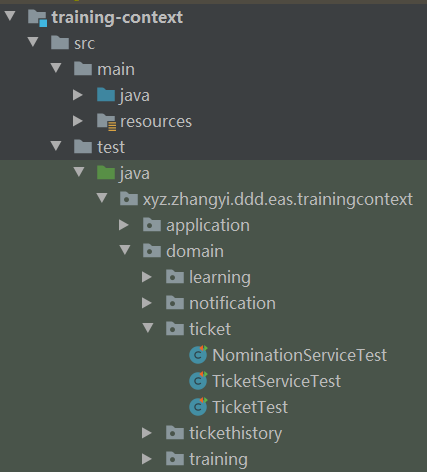
在编写代码的过程中，要保证定义的类与接口遵循代码模型对模块、包、命名空间的划分。原则上，当前限界上下文的领域模型对象都定义在 domain 包里。在进一步对 domain 包进行划分时， **千万不要按照领域驱动设计的设计要素类别进行划分** ——将领域服务、实体、值对象分门别类放在一起的做法是绝对错误的！包或模块的划分应依据 **变化的方向** ，这一划分原则满足“高内聚低耦合”原则。换言之，当前限界上下文的所有领域服务并非高内聚的，实体、值对象同样如此；但是，领域设计模型定义的每个聚合却应当是高内聚的，若非如此，只能说明聚合的设计存在问题。

因此，在编写领域层代码时，应根据领域设计建模获得的设计模型，按照聚合对 domain 包进行划分，确定领域模型对象的命名空间，如下所示：



da660fbb-d25e-4314-9834-0ac4cb065333.png

上图中的 candidate、course、learning、ticket 等命名空间，正是之前设计建模时识别出来的聚合。领域层的测试代码模型与之对应：



841e75f8-a588-41ca-b9df-d7969743c08c.png

#### **应用服务的编码实现**

在完成一个领域场景的领域层代码实现之后，将在应用层的应用服务交汇。一方面，需要根据服务定义，确定应用服务的接口与消息契约对象，并实现应用服务，然后由此向上（向外）实现基础设施层北向网关的远程服务；另一方面，需要为领域服务提供资源库的实现，以及其他需要访问外部资源的南向网关的代码逻辑。

在编写应用服务时，需要考虑：

* 应用服务的测试为集成测试：需要通过 setup 与 teardown 准备和清除测试数据，并准备运行集成测试的环境
* 依赖管理：考虑应用服务、领域服务、资源库之间的依赖管理，确定依赖注入（DI）框架
* 消息契约对象的定义：需要结合对外暴露的远程服务接口定义消息契约对象
* 横切关注点的结合：包括事务、异常处理等横切关注点的实现与集成
* 南向网关的实现：考虑资源库和其他访问外部资源的网关接口的实现，包括框架和技术选型

“提名候选人”的应用服务 NominationAppService 实现如下：

@Service  
@EnableTransactionManagement  
public class NominationAppService {  
 @Autowired  
 private NominationService nominationService;  
  
 @Transactional(rollbackFor = ApplicationException.class)  
 public void nominate(NominationRequest nominationRequest) {  
 if (Objects.isNull(nominationRequest)) {  
 throw new ApplicationValidationException("nomination request can not be null");  
 }  
 try {  
 nominationService.nominate(  
 nominationRequest.getTicketId(),  
 nominationRequest.getTrainingId(),  
 nominationRequest.toCandidate(),  
 nominationRequest.toNominator());  
 } catch (DomainException ex) {  
 throw new ApplicationDomainException(ex.getMessage(), ex);  
 } catch (Exception ex) {  
 throw new ApplicationInfrastructureException("Infrastructure Error", ex);  
 }  
 }  
}

我选择了 Spring 作为依赖注入的框架，事务处理采用声明式事务。应用层异常统一定义为 ApplicationException 类型。它是一个抽象类，具有三个异常子类：

* ApplicationDomainException：因为领域逻辑错误导致的异常
* ApplicationValidationException：因为输入参数验证错误导致的异常
* ApplicationInfrastructureException：因为基础设施访问错误导致的异常

在 EAS 系统中，我为异常划分了层次。领域层的所有自定义异常都派生自 DomainException 超类，应用层在定义了超类的同时，仅规定了三种具体的异常子类，这些异常子类的类别统一了 REST 服务要求返回的状态码。至于基础设施层，则不需要考虑，因为基础设施代码抛出的异常属于基础设施框架。

异常的划分方式体现了分层架构对异常的考虑。领域层通过自定义异常体现了丰富多彩的领域校验逻辑与错误消息，到了应用层，又保证了异常的统一性。异常分层机制确保了代码的健壮性与简单性。领域层作为整洁架构的内部核心，无需关注基础设施层抛出的系统异常，而是将自定义异常视为领域逻辑的一部分。在编写领域层的代码时，对异常的态度为“只抛出，不捕获”。任何异常带来的健壮性隐患，都交给了外层的应用服务。应用服务对待异常的态度迥然不同，采用了“捕获底层异常，抛出应用异常”的设计原则。

应用服务接口的消息契约对象负责消息契约与领域模型的转换。若转换行为包含了业务逻辑，需要编写单元测试去覆盖它，甚至可采用测试驱动开发的过程，尤其当引入了装配器（Assembler）时，更需如此。消息契约对象的结构是领域驱动设计上下文映射模式中发布语言（Published Language）的体现，它同时作为应用服务与远程服务的参数和返回值。要支持远程服务，则消息契约对象需要支持序列化与反序列化。一些序列化框架会通过反射调用对象的构造函数与 getter/setter 访问器，故而消息契约对象的定义应遵循 Java Bean 规范。

为应用服务编写集成测试时，至少需要考虑两个测试用例：正常执行完成的用例与抛出异常需要事务回滚的用例。如下所示：

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("/spring-mybatis.xml")  
public class NominationAppServiceIT {  
 @Autowired  
 private TrainingRepository trainingRepository;  
 @Autowired  
 private TicketRepository ticketRepository;  
 @Autowired  
 private ValidDateRepository validDateRepository;  
 @Autowired  
 private TicketHistoryRepository ticketHistoryRepository;  
  
 @Autowired  
 private NominationAppService nominationAppService;  
  
 @Before  
 public void setup() {  
 training = createTraining();  
 ticket = createTicket();  
 validDate = createValidDate();  
  
 // clean dirty data;  
 trainingRepository.remove(training);  
 ticketRepository.remove(ticket);  
 validDateRepository.remove(validDate);  
 ticketHistoryRepository.deleteBy(ticketId);  
  
 // prepare new data;  
 trainingRepository.add(this.training);  
 ticketRepository.add(ticket);  
 validDateRepository.add(validDate);  
 }  
  
 @Test  
 public void should\_nominate\_candidate\_to\_nominee() {  
 // given  
 NominationRequest nominationRequest = createNominationRequest();  
  
 // when  
 nominationAppService.nominate(nominationRequest);  
  
 // then  
 Optional<Ticket> optionalAvailableTicket = ticketRepository.ticketOf(ticketId, Available);  
 assertThat(optionalAvailableTicket.isPresent()).isFalse();  
  
 Optional<Ticket> optionalConfirmedTicket = ticketRepository.ticketOf(ticketId, TicketStatus.WaitForConfirm);  
 assertThat(optionalConfirmedTicket.isPresent()).isTrue();  
 Ticket ticket = optionalConfirmedTicket.get();  
 assertThat(ticket.id()).isEqualTo(ticketId);  
 assertThat(ticket.trainingId()).isEqualTo(trainingId);  
 assertThat(ticket.status()).isEqualTo(TicketStatus.WaitForConfirm);  
 assertThat(ticket.nomineeId()).isEqualTo(candidateId);  
  
 Optional<TicketHistory> optionalTicketHistory = ticketHistoryRepository.latest(ticketId);  
 assertThat(optionalTicketHistory.isPresent()).isTrue();  
 TicketHistory ticketHistory = optionalTicketHistory.get();  
 assertThat(ticketHistory.ticketId()).isEqualTo(ticketId);  
 assertThat(ticketHistory.getStateTransit()).isEqualTo(StateTransit.from(Available).to(WaitForConfirm));  
 }  
  
 @Test  
 public void should\_rollback\_if\_DomainException\_had\_been\_thrown() {  
 // given  
 NominationRequest nominationRequest = createNominationRequest();  
  
 // removing valid date in order to throw DomainException  
 validDateRepository.remove(validDate);  
  
 // when  
 try {  
 nominationAppService.nominate(nominationRequest);  
 } catch (ApplicationException e) {  
 // then  
 Optional<Ticket> optionalAvailableTicket = ticketRepository.ticketOf(ticketId, Available);  
 assertThat(optionalAvailableTicket.isPresent()).isTrue();  
 Ticket ticket = optionalAvailableTicket.get();  
 assertThat(ticket.id()).isEqualTo(ticketId);  
 assertThat(ticket.trainingId()).isEqualTo(trainingId);  
 assertThat(ticket.status()).isEqualTo(Available);  
 assertThat(ticket.nomineeId()).isEqualTo(null);  
 }  
 }  
}

NominationAppService 的测试类本应该仅依赖于被测应用服务。之所以引入了 TrainingRepository 等资源库的依赖，是为了给集成测试准备和清除数据所用。系统由 flywaydb 管理数据库版本与数据迁移，但集成测试需要的数据不在此列，需要由测试提供数据；更何况集成测试会被反复运行，每个测试用例需要的数据都是彼此独立的。

数据的清除本该由 JUnit 的 teardown 钩子方法负责；不过，在运行集成测试之后，通常需要手工查询数据库以了解被测方法执行之后的数据结果，如果在测试方法执行后通过 teardown 清除了数据，就无法查看执行后的结果了。为避免此种情形，可以将数据的清除挪到准备数据之前。如上测试代码所示，清除数据与准备数据的实现都放到了 setup 钩子方法中。

在编写事务回滚的测试用例时，可以故意营造抛出异常的情况，如上测试方法，我故意通过 ValidDateRepository 删除了提名场景需要的有效日期，导致 DomainException 异常抛出。应用服务在捕获该领域异常后，统一抛出了 ApplicationException，因此事务回滚标记的异常类型为 ApplicationException：

@Transactional(rollbackFor = ApplicationException.class)  
public void nominate(NominationRequest nominationRequest) throws ApplicationException {}

#### **资源库的编码实现**

EAS 的数据库为 MySQL 关系数据库，应选择 ORM 框架实现资源库。这里，我选择了 MyBatis，并采用配置方式定义了 Mapper，如此可减少该框架对 Repository 接口的侵入。虽然 MyBatis 建议将数据访问对象定义为 XXXMapper，但这里我沿用了领域驱动设计的资源库模式，定义为资源库接口，如：

package xyz.zhangyi.ddd.eas.trainingcontext.domain.tickethistory;  
  
import org.apache.ibatis.annotations.Mapper;  
import org.springframework.stereotype.Repository;  
import java.util.Optional;  
  
import xyz.zhangyi.ddd.eas.trainingcontext.domain.ticket.TicketId;  
  
@Mapper  
@Repository  
public interface TicketHistoryRepository {  
 Optional<TicketHistory> latest(TicketId ticketId);  
 void add(TicketHistory ticketHistory);  
 void deleteBy(TicketId ticketId);  
}

它对应的 mapper 配置文件如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >  
<mapper namespace="xyz.zhangyi.ddd.eas.trainingcontext.domain.tickethistory.TicketHistoryRepository" >  
 <resultMap id="ticketHistoryResult" type="TicketHistory" >  
 <id column="id" property="id" jdbcType="VARCHAR"/>  
 <result column="ticketId" property="ticketId.value" jdbcType="VARCHAR" />  
 <result column="operationType" property="operationType" jdbcType="VARCHAR" />  
 <result column="operatedAt" property="operatedAt" jdbcType="TIMESTAMP" />  
 <association property="owner" javaType="TicketOwner">  
 <constructor>  
 <arg column="ownerId" jdbcType="VARCHAR" javaType="String"/>  
 <arg column="ownerType" jdbcType="VARCHAR" javaType="TicketOwnerType" />  
 </constructor>  
 </association>  
 <association property="stateTransit" javaType="StateTransit">  
 <constructor>  
 <arg column="fromStatus" jdbcType="VARCHAR" javaType="TicketStatus" />  
 <arg column="toStatus" jdbcType="VARCHAR" javaType="TicketStatus" />  
 </constructor>  
 </association>  
 <association property="operatedBy" javaType="Operator">  
 <constructor>  
 <arg column="operatorId" jdbcType="VARCHAR" javaType="String" />  
 <arg column="operatorName" jdbcType="VARCHAR" javaType="String" />  
 </constructor>  
 </association>  
 </resultMap>  
  
 <select id="latest" parameterType="TicketId" resultMap="ticketHistoryResult">  
 select  
 id, ticketId, ownerId, ownerType, fromStatus, toStatus, operationType, operatorId, operatorName, operatedAt  
 from ticket\_history  
 where ticketId = #{ticketId} and operatedAt = (select max(operatedAt) from ticket\_history where ticketId = #{ticketId})  
 </select>  
  
 <insert id="add" parameterType="TicketHistory">  
 insert into ticket\_history  
 (id, ticketId, ownerId, ownerType, fromStatus, toStatus, operationType, operatorId, operatorName, operatedAt)  
 values  
 (  
 #{id},  
 #{ticketId}, #{ticketOwner.employeeId}, #{ticketOwner.ownerType},  
 #{stateTransit.from}, #{stateTransit.to}, #{operationType},  
 #{operatedBy.operatorId}, #{operatedBy.name}, #{operatedAt}  
 )  
 </insert>  
  
 <delete id="deleteBy" parameterType="TicketId">  
 delete from ticket\_history where ticketId = #{ticketId}  
 </delete>  
</mapper>

应用服务的一个公开方法对应了一个完整的领域场景，为其编写集成测试时，需要该领域场景各个任务的工作都已准备完毕。结合场景驱动设计与测试驱动开发，领域服务与聚合已经在应用服务之前实现，资源库或其他南向网关对象的接口定义也已确定，但它们的实现却不曾验证。为此，可以考虑在实现应用服务之前，先为南向网关对象的实现编写集成测试。例如，为 TicketHistoryRepository 编写的集成测试如下：

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("/spring-mybatis.xml")  
public class TicketHistoryRepositoryIT {  
 @Autowired  
 private TicketHistoryRepository ticketHistoryRepository;  
 private final TicketId ticketId = TicketId.from("18e38931-822e-4012-a16e-ac65dfc56f8a");  
  
 @Before  
 public void setup() {  
 ticketHistoryRepository.deleteBy(ticketId);  
  
 StateTransit availableToWaitForConfirm = from(Available).to(WaitForConfirm);  
 LocalDateTime oldTime = LocalDateTime.of(2020, 1, 1, 12, 0, 0);  
 TicketHistory oldHistory = createTicketHistory(availableToWaitForConfirm, oldTime);  
 ticketHistoryRepository.add(oldHistory);  
  
 StateTransit toConfirm = from(WaitForConfirm).to(Confirm);  
 LocalDateTime newTime = LocalDateTime.of(2020, 1, 1, 13, 0, 0);  
 TicketHistory newHistory = createTicketHistory(toConfirm, newTime);  
 ticketHistoryRepository.add(newHistory);  
 }  
  
 @Test  
 public void should\_return\_latest\_one() {  
 Optional<TicketHistory> latest = ticketHistoryRepository.latest(ticketId);  
  
 assertThat(latest.isPresent()).isTrue();  
 assertThat(latest.get().getStateTransit()).isEqualTo(from(WaitForConfirm).to(Confirm));  
 }  
}

考虑到集成测试需要准备测试环境，执行效率也要低于单元测试，故而需要将单元测试和集成测试分为两个不同的构建阶段。

#### **远程服务的编码实现**

在实现了应用服务之后，继续逆流而上，编写作为北向网关的远程服务。如果是定义 REST 服务，需要遵循 REST 服务接口的设计原则。例如 TicketResource 的实现：

@RestController  
@RequestMapping("/tickets")  
public class TicketResource {  
 private Logger logger = Logger.getLogger(TicketResource.class.getName());  
  
 @Autowired  
 private NominationAppService nominationAppService;  
  
 @PutMapping  
 public ResponseEntity<?> nominate(@RequestBody NominationRequest nominationRequest) {  
 if (Objects.isNull(nominationRequest)) {  
 logger.log(Level.WARNING,"Nomination Request is Null.");  
 return new ResponseEntity<>(HttpStatus.BAD\_REQUEST);  
 }  
 try {  
 nominationAppService.nominate(nominationRequest);  
 return new ResponseEntity<>(HttpStatus.ACCEPTED);  
 } catch (ApplicationException e) {  
 logger.log(Level.SEVERE, "Exception raised by nominate REST Call.", e);  
 return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  
 }  
 }  
}

虽然服务接口定义并不相同，选择的 HTTP 动词也不相同，但这仅仅是接口定义的差异，每个 REST 资源类服务方法的实现却是大同小异的，即执行对应应用服务的方法，捕获异常，根据执行结果返回带有不同状态码的值。为了避免繁琐代码的编写，应用层定义的应用异常类别就派上了用场，利用 catch 捕获不同类型的应用异常，就可以实现相似的执行逻辑。为此，我在 eas-core 模块中定义了一个 Resources 辅助类：

public class Resources {  
 private static Logger logger = Logger.getLogger(Resources.class.getName());  
  
 private Resources(String requestType) {  
 this.requestType = requestType;  
 }  
  
 private String requestType;  
 private HttpStatus successfulStatus;  
 private HttpStatus errorStatus;  
 private HttpStatus failedStatus;  
  
 public static Resources with(String requestType) {  
 return new Resources(requestType);  
 }  
  
 public Resources onSuccess(HttpStatus status) {  
 this.successfulStatus = status;  
 return this;  
 }  
  
 public Resources onError(HttpStatus status) {  
 this.errorStatus = status;  
 return this;  
 }  
  
 public Resources onFailed(HttpStatus status) {  
 this.failedStatus = status;  
 return this;  
 }  
  
 public <T> ResponseEntity<T> execute(Supplier<T> supplier) {  
 try {  
 T entity = supplier.get();  
 return new ResponseEntity<>(entity, successfulStatus);  
 } catch (ApplicationValidationException ex) {  
 logger.log(Level.WARNING, String.format("The request of %s is invalid", requestType));  
 return new ResponseEntity<>(errorStatus);  
 } catch (ApplicationDomainException ex) {  
 logger.log(Level.WARNING, String.format("Exception raised %s REST Call", requestType));  
 return new ResponseEntity<>(failedStatus);  
 } catch (ApplicationInfrastructureException ex) {  
 logger.log(Level.SEVERE, String.format("Fatal exception raised %s REST Call", requestType));  
 return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  
 }  
 }  
  
 public ResponseEntity<?> execute(Runnable runnable) {  
 try {  
 runnable.run();  
 return new ResponseEntity<>(successfulStatus);  
 } catch (ApplicationValidationException ex) {  
 logger.log(Level.WARNING, String.format("The request of %s is invalid", requestType));  
 return new ResponseEntity<>(errorStatus);  
 } catch (ApplicationDomainException ex) {  
 logger.log(Level.WARNING, String.format("Exception raised %s REST Call", requestType));  
 return new ResponseEntity<>(failedStatus);  
 } catch (ApplicationInfrastructureException ex) {  
 logger.log(Level.SEVERE, String.format("Fatal exception raised %s REST Call", requestType));  
 return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  
 }  
 }  
}

execute() 方法的不同重载对应于是否返回响应消息对象的场景。不同异常类别对应的状态码由调用者传入。为了有效地记录日志信息，需要由调用者提供本服务请求的描述。在引入 Resources 类后，TicketResource 的服务实现为：

@RestController  
@RequestMapping("/tickets")  
public class TicketResource {  
 private Logger logger = Logger.getLogger(TicketResource.class.getName());  
  
 @Autowired  
 private NominationAppService nominationAppService;  
  
 @PutMapping  
 public ResponseEntity<?> nominate(@RequestBody NominationRequest nominationRequest) {  
 return Resources.with("nominate ticket")  
 .onSuccess(ACCEPTED)  
 .onError(BAD\_REQUEST)  
 .onFailed(INTERNAL\_SERVER\_ERROR)  
 .execute(() -> nominationAppService.nominate(nominationRequest));  
 }  
}

而 TrainingResource 的实现则为：

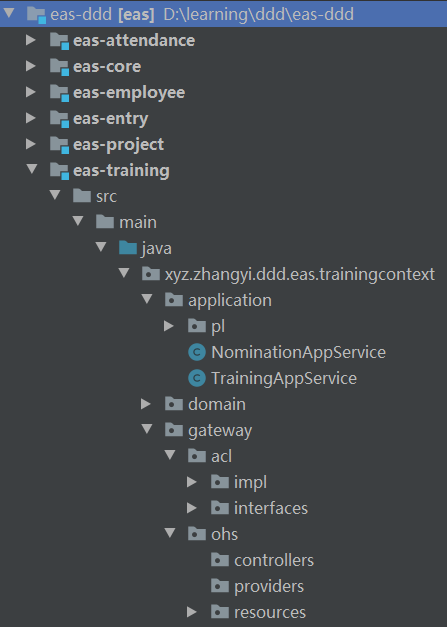
@RestController  
@RequestMapping("/trainings")  
public class TrainingResource {  
 private Logger logger = Logger.getLogger(TrainingResource.class.getName());  
  
 @Autowired  
 private TrainingAppService trainingAppService;  
  
 @GetMapping(value = "/{id}")  
 public ResponseEntity<TrainingResponse> findBy(@PathVariable String id) {  
 return Resources.with("find training by id")  
 .onSuccess(HttpStatus.OK)  
 .onError(HttpStatus.BAD\_REQUEST)  
 .onFailed(HttpStatus.NOT\_FOUND)  
 .execute(() -> trainingAppService.trainingOf(id));  
 }  
}

显然经过这样的重构，可以有效地规避远程服务代码不必要的相似代码重复。

为了保证远程服务的正确性，应考虑为远程服务编写集成测试或契约测试。若选择 Spring Boot 作为 REST 框架，可利用 Spring Boot 提供的测试沙箱 spring-boot-starter-test 为远程服务编写集成测试，或者选择 [Pact](https://docs.pact.io/) 之类的测试框架为其编写消费者驱动的契约测试（Consumer-Driven Contract Test）。如果要面向前端定义控制器（Controller），还可考虑引入 [GraphQL](https://graphql.org/) 定义服务，这些服务为前端组成了 BFF（Backend For Frontend）服务。此外，还可以引入 [Swagger](https://swagger.io/) 为这些远程服务定义 API 文档。

### EAS 系统的代码模型

应用服务与消息契约对象定义在应用层，远程服务虽然处于后端分层架构的顶层，但其本质仍然是基础设施层的北向网关。在定义代码模型时，可以根据分层架构的要素划分模块或包，也可以根据领域驱动设计的模式来划分。EAS 系统的代码模型如下图所示：



d40a0752-6879-4423-82ee-6adeea9014f0.png

以下是对代码模型的详细说明：

\* eas-ddd：项目名称为 EAS  
 \* eas-training：以项目名称为前缀，命名限界上下文对应的模块  
 \* eas.trainingcontext：限界上下文的命名空间，以 context 为后缀  
 \* application：应用层  
 \* pl：即 Published Language 的缩写，该命名空间下的类为消息契约对象，也可以认为是 DTO，乃开发主机服务的发布语言  
 \* domain：领域层，其内部按照聚合边界进行命名空间划分，每个聚合内的实体、值对象以及它对应的领域服务和资源库接口都定义在同一个聚合内部  
 \* gateway：即基础设施层，包含了北向网关和南向网关  
 \* acl：南向网关，Anti-Corruption Layer 的缩写，作为防腐层，需要将接口和实现分离  
 \* interfaces：除 Repository 之外的所有南向网关接口定义  
 \* impl：包含了 Repository 实现的所有南向网关的实现  
 \* ohs：北向网关，Open Host Service 的缩写，皆为远程服务，根据服务的不同可以分为 resources、controllers、providers 以及事件的 publishers

EAS 即使作为一个单体架构，仍然需要清晰地为每个限界上下文定义单独的模块，其中，eas-core 作为共享内核，包含了系统层次的领域内核与基础设施公共组件。EAS 项目的 pom 文件体现了这些模块的定义：

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>xyz.zhangyi.ddd</groupId>  
 <artifactId>eas</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>pom</packaging>  
  
 <modules>  
 <module>eas-core</module>  
 <module>eas-employee</module>  
 <module>eas-attendance</module>  
 <module>eas-project</module>  
 <module>eas-training</module>  
 <module>eas-entry</module>  
 </modules>  
</project>

eas-entry 是整个系统的主程序入口，它仅仅定义了一个 EasApplication 类：

package xyz.zhangyi.ddd.eas;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.transaction.annotation.EnableTransactionManagement;  
  
@SpringBootApplication  
@EnableTransactionManagement  
public class EasApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(EasApplication.class, args);  
 }  
}

通过它可以为整个系统启动一个服务。Spring Boot 需要的配置也定义在 eas-entry 模块的 resource  
文件夹下。该入口加载的所有远程服务均定义在各个限界上下文的内部，保证了每个限界上下文的架构完整性。

正如我在 5-10《领域驱动设计的精髓》总结的边界层次，限界上下文的边界要高于分层的边界，体现在代码模型中，应该是先有限界上下文的模块，再有限界上下文内部的分层。若需要将逻辑分层也定义为模块，这些层次的模块应作为限界上下文模块的子模块。 **如下的代码模型需要竭力避免：**

* application
  + trainingcontext
  + ticketcontext
  + …
* domain
  + trainingcontext
  + ticketcontext
  + …
* gateway
  + acl
    - impl
      * persistence
        + trainingcontext
        + ticketcontext
        + …

只要保证了限界上下文边界在分层边界之上，就清晰地维护了整个系统的内外层次。当我们需要将一个单体架构迁移到微服务架构时，就能降低架构的迁移成本。事实上，若遵循这里建议的代码模型，你会发现：两种迥然不同的架构风格其实拥有完全相同的代码模型。执行架构迁移时，影响到的仅仅包含：

* 与单体架构不同，需要为每个微服务提供一个主程序入口，即去掉 eas-entry 模块，为每个限界上下文（微服务）定义一个 Application 类
* 修改 gateway的实现，将进程内的通信改为跨进程通信
* 修改数据库的配置文件，让 DB 的 url 指向不同的数据库
* 调整应用层的事务处理机制，考虑使用分布式柔性事务

以上修改皆不影响领域层代码，包括领域层的产品代码与测试代码的已有实现。领域层代码作为整洁架构分层的内核，体现了它一如既往的稳定性。

### EAS 的设计与开发流程

到此为止，我们实现了 EAS 系统相关限界上下文从聚合内的实体与值对象到领域服务、应用服务和远程服务的编码实现。毋庸置疑， **面向场景的领域模型驱动设计过程，是一个有着清晰而固化的软件开发流程。**

领域分析建模使用了一种有形的模型语言将无形的软件需求呈现出来，跨过了从现实世界到模型世界的鸿沟；领域设计建模则从整体出发，细节入手，在限界上下文、领域层和聚合的边界控制下对领域分析模型进行分解，形成一个个作用不同的“原子”构件；到领域实现建模时，再用编程语言赋予这些“原子”构件活动和运行的能力，并将它们组装起来，在测试的保护下，缝合成天衣无缝的整体，最后以外部服务的形式暴露给消费者。EAS 的整体案例体现了领域驱动战术设计的全过程。

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