**4.1建模**

**4.1.1聚合**

**4.1.1.1基本聚合结构**

聚合是一个常规对象，它包含状态和更改该状态的方法。创建聚合对象时，实际上是在创建“聚合根”，通常包含整个聚合的名称。下面给出一个例子，我们将构造“礼品卡”域，它将GiftCard作为聚合（根）。默认情况下，Axon将您的聚合配置为“事件源”聚合（如下所述）。然后，我们的基本礼品卡聚合结构将主要用于活动采购方法：

***@Aggregate***

@Profile("command")

public class GiftCard {

private static final Logger logger = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

***@AggregateIdentifier***

private String giftCardId;

private int remainingValue;

***@CommandHandler***

public GiftCard(IssueCmd cmd) {

logger.debug("handling {}", cmd);

if (cmd.getAmount() <= 0) {

throw new IllegalArgumentException("amount <= 0");

}

***apply(new IssuedEvt(cmd.getId(), cmd.getAmount()));***

}

@CommandHandler

public void handle(RedeemCmd cmd) {

logger.debug("handling {}", cmd);

if (cmd.getAmount() <= 0) {

throw new IllegalArgumentException("amount <= 0");

}

if (cmd.getAmount() > remainingValue) {

throw new IllegalStateException("amount > remaining value");

}

apply(new RedeemedEvt(giftCardId, cmd.getAmount()));

}

***@EventSourcingHandler***

public void on(***IssuedEvt*** evt) {

logger.debug("applying {}", evt);

giftCardId = evt.getId();

remainingValue = evt.getAmount();

logger.debug("new remaining value: {}", remainingValue);

}

***public GiftCard()*** {

// Required by Axon

logger.debug("Empty constructor invoked");

}

}

在给定的代码片段中有两个值得注意的概念，它们用编号的Java注释标记，使用这些注释应注意以下要点

①@AggregateIdentifier是指向GiftCard聚合的外部引用点。这个字段是一个硬要求，因为没有它，Axon将不知道给定命令的目标聚合。请注意，此注释可以放置在字段和方法上。

②@CommandHandler带注释的构造函数，或者以不同的方式将“command handling constructor”放在一起。此注释告诉框架给定的构造函数能够处理IssueCardCommand。@CommandHandler注释函数是放置决策/业务逻辑的地方。

③静态AggregateLifecycle.apply（Object…）是应该发布事件消息时使用的。调用此函数时，所提供的对象将在其应用的聚合范围内作为EventMessages发布。

④使用@EventSourcingHandler告诉框架，当聚合是“源于其事件”时，应该调用带注释的函数。由于所有事件源处理程序组合在一起将形成聚合，因此所有状态更改都在这里发生。请注意，聚合标识符必须在聚合发布的第一个事件的@EventSourcingHandler中设置。这通常是创建事件。最后，@EventSourcingHandler注释的函数是使用特定规则解析的。这些规则对于@EventHandler注释的方法是相同的，并在带注释的事件处理程序中进行了详细说明。

⑤一个无参构造方法是Axon所必需的。Axon框架使用此构造函数在使用过去的事件初始化它之前创建一个空的聚合实例。未提供此构造函数将导致加载聚合时出现异常。

4.1.1.2聚合生命周期操作

在一个聚合的生命周期中，需要执行一些操作。为此，Axon中的AggregateLifecycle类提供了两个静态函数：

①apply（Object）和apply（Object，MetaData）：AggregateLifecycle#app将在EventBus上发布一条事件消息，以便知道它是由执行操作的聚合发出的。可以只提供事件对象，也可以同时提供事件和某些特定的元数据。

②createNew（Class，Callable）：作为处理命令的结果实例化一个新的聚合。 阅读本文了解更多细节。

③isLive（）：检查以验证聚合是否处于“存活”状态。如果一个聚合体完成了历史事件的回放以重现其状态，那么它被认为是“活的”。如果聚合因此处于事件源的过程中，则AggregateLifecycle.isLive()调用将返回false。使用这个isLive（）方法，您可以执行只有在处理新生成的事件时才应该执行的活动。

④markDeleted（）：将调用函数的聚合实例标记为“deleted”。

如果域指定可以删除/删除/关闭给定的聚合，则非常有用，在此之后，它将不再被允许处理任何命令。应该从@EventSourcingHandler带注释的函数调用此函数，以确保标记为已删除是该聚合状态的一部分。

### **4.1.2多实体聚合**

只有聚合根的聚合所能提供的内容通常满足不了复杂的业务逻辑。在这种情况下，我们要将复杂性分散到聚合中的多个“实体”上。在本章中，我们将讨论在聚合中创建实体的细节，以及它们如何处理消息。

*实体间的状态*

*我们一般认为聚合不应公开状态，对这个观点的一种常见误解是，任何实体都不应包含任何属性访问器方法。事实并非如此。实际上，如果聚合中的实体向同一聚合中的其他实体公开状态，这对聚合的构建是有利的。但是，建议不要把状态公开到聚合之外。*

在“礼品卡”域中，本节定义了GiftCard聚合根。让我们利用此域来引入实体：

import org.axonframework.modelling.command.AggregateIdentifier;

import org.axonframework.modelling.command.AggregateMember;

import org.axonframework.modelling.command.EntityId;

public class GiftCard {

@AggregateIdentifier

private String id;

@AggregateMember // 1.

private List<GiftCardTransaction> transactions = new ArrayList<>();

private int remainingValue;

// 省略了构造函数、命令和事件源处理程序

}

public class GiftCardTransaction {

@EntityId // 2.

private String transactionId;

private int transactionValue;

private boolean reimbursed = false;

public GiftCardTransaction(String transactionId, int transactionValue) {

this.transactionId = transactionId;

this.transactionValue = transactionValue;

}

public String getTransactionId() {

return transactionId;

}

// 省略了命令处理程序，事件来源处理程序和equals/hashCode

}

与聚合根一样，实体是简单的对象，如新的GiftCardTransaction实体所示。上面的代码片段显示了多实体聚合的两个重要概念：

①声明子实体的字段必须用@AggregateMember注释，这个注释表明这个字段包含一个被检查的Axon类。此示例显示了Iterable实现上的注释，但也可以将其放置在单个对象或映射上。在后一种情况下，映射的值应包含实体，而键包含用作其引用的值。请注意，此注释可以放置在字段和方法上。

②@EntityId注释，指定实体的标识字段。需要能够将命令（或事件）消息路由到正确的实体实例。负载上用于查找消息应路由到的实体的属性默认为@EntityId注释字段的名称。例如，在注释字段transactionId时，命令必须定义具有相同名称的属性，这意味着必须存在transactionId或getTransactionId（）方法。如果字段名和路由属性不同，可以使用@EntityId（routingKey=“customRoutingProperty”）显式提供一个值。如果此注释将是子实体集合或映射的一部分，则此注释在实体实现上是必需的。请注意，此注释可以放置在字段和方法上。

*定义实体类型*

*集合或映射的字段声明都应包含适当的泛型，以允许Axon标识集合或映射中包含的实体类型。如果无法在声明中添加泛型（例如，因为您使用的是已定义泛型类型的自定义实现），则必须通过在@AggregateMember注释中指定类型字段来指定实体类型：*

*>@AggregateMember(type = GiftCardTransaction.class).*

4.1.2.1实体中的命令处理

@CommandHandler注释不限于聚合根。将所有命令处理程序放在根目录下有时会导致聚合根上有大量方法，而其中许多方法只是将调用转发给底层实体之一。如果是这样，您可以将@CommandHandler注释放在底层实体的某个方法上。要使Axon找到这些带注释的方法，在聚合根中声明实体的字段必须标记为@AggregateMember:

public class GiftCard {

@AggregateIdentifier

private String id;

@AggregateMember

private List<GiftCardTransaction> transactions = new ArrayList<>();

private int remainingValue;

// omitted constructors, command and event sourcing handlers

}

public class GiftCardTransaction {

@EntityId

private String transactionId;

private int transactionValue;

private boolean reimbursed = false;

public GiftCardTransaction(String transactionId, int transactionValue) {

this.transactionId = transactionId;

this.transactionValue = transactionValue;

}

@CommandHandler

public void handle(ReimburseCardCommand cmd) {

if (reimbursed) {

throw new IllegalStateException("Transaction already reimbursed");

}

apply(new CardReimbursedEvent(cmd.getCardId(), transactionId, transactionValue));

}

// omitted getter, event sourcing handler and equals/hashCode

}

注意，对于命令处理程序，只检查注释字段的声明类型。如果字段值在该实体的传入命令到达时为null，则引发异常。如果存在子实体的集合或映射，并且找不到与命令的路由相匹配的实体，则Axon将抛出IllegalStateException，因为显然聚合无法在那时处理该命令。

*命令处理程序注意事项*

*请注意，每个命令在聚合中必须正好有一个处理程序。这意味着您不能用处理同一命令类型的@CommandHandler注释多个实体（根实体或非实体）。如果需要有条件地将命令路由到实体，则应该是这些实体的父级处理该命令，并根据适用的条件转发该命令。字段的运行时类型不必一定要声明。但是对于@CommandHandler注释的方法，只检查@AggregateMember注释字段的声明类型。*

4.1.2.2实体中的事件回溯处理

当使用事件回溯作为存储聚合的机制时，不仅聚合根需要使用事件来触发状态转换，而且该聚合中的每个实体也需要使用事件。Axon提供了对事件回溯的支持，比如这些现成的复杂聚合结构。

当实体（包括聚合根）应用事件时，它首先由聚合根处理，然后通过每个@AggregateMember注释的字段向下冒泡到其包含的所有子实体：

public class GiftCard {

@AggregateIdentifier

private String id;

@AggregateMember

private List<GiftCardTransaction> transactions = new ArrayList<>();

@CommandHandler

public void handle(RedeemCardCommand cmd) {

// Some decision making logic

apply(new CardRedeemedEvent(id, cmd.getTransactionId(), cmd.getAmount()));

}

@EventSourcingHandler

public void on(CardRedeemedEvent evt) { / 1.

transactions.add(new GiftCardTransaction(evt.getTransactionId(), evt.getAmount()));

}

// omitted constructors, command and event sourcing handlers

}

public class GiftCardTransaction {

@EntityId

private String transactionId;

private int transactionValue;

private boolean reimbursed = false;

public GiftCardTransaction(String transactionId, int transactionValue) {

this.transactionId = transactionId;

this.transactionValue = transactionValue;

}

@CommandHandler

public void handle(ReimburseCardCommand cmd) {

if (reimbursed) {

throw new IllegalStateException("Transaction already reimbursed");

}

apply(new CardReimbursedEvent(cmd.getCardId(), transactionId, transactionValue));

}

@EventSourcingHandler

public void on(CardReimbursedEvent event) {// 2.

if (transactionId.equals(event.getTransactionId())) {

reimbursed = true;

}

}

// omitted getter and equals/hashCode

}

上面的片段中有两个细节值得一提：

①实体的创建在其父实体的事件源处理程序中进行。因此，在实体类上不可能像对聚合根一样有“命令处理构造函数”。

②实体中的事件源处理程序执行验证检查，检查接收到的事件是否实际属于实体。这是必要的，因为一个实体实例应用的事件也将由同一类型的任何其他实体实例处理。

通过更改@AggregateMember注释上的eventForwardingMode，可以自定义第二点中描述的情况：

public class GiftCard {

@AggregateIdentifier

private String id;

@AggregateMember(eventForwardingMode = ForwardMatchingInstances.class)

private List<GiftCardTransaction> transactions = new ArrayList<>();

// omitted constructors, command and event sourcing handlers

}

通过将eventForwardingMode设置为ForwardMatchingInstances，只有当事件消息包含与实体上@EntityId注释字段名称匹配的字段/getter时，才会转发该消息。这种路由行为可以通过@EntityId注释上的routingKey字段进一步指定，这与实体中的路由命令的行为类似。其他可以使用的转发模式是ForwardAll（默认）和ForwardNone，它们分别将所有事件转发给所有实体或根本不转发任何事件。

### **4.1.3 使用聚合来存储状态**

在Aggregate主页中，我们看到了如何创建由事件回溯重置的聚合。换言之，事件回溯聚合的存储方法是重放构成聚合更改的事件。

但是，聚合也可以按原样存储。执行此操作时，用于保存和加载聚合的存储库是GenericJpaRepository。状态存储聚合的结构与事件源聚合稍有不同：

@Entity // 1.

public class GiftCard {

@Id // 2.

@AggregateIdentifier

private String id;

// 3.

@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)

@JoinColumn(name = "giftCardId")

@AggregateMember

private List<GiftCardTransaction> transactions = new ArrayList<>();

private int remainingValue;

@CommandHandler // 4.

public GiftCard(IssueCardCommand cmd) {

if (cmd.getAmount() <= 0) {

throw new IllegalArgumentException("amount <= 0");

}

id = cmd.getCardId();

remainingValue = cmd.getAmount();

// 5.

apply(new CardIssuedEvent(cmd.getCardId(), cmd.getAmount()));

}

@CommandHandler

public void handle(RedeemCardCommand cmd) {

// 6.

if (cmd.getAmount() <= 0) {

throw new IllegalArgumentException("amount <= 0");

}

if (cmd.getAmount() > remainingValue) {

throw new IllegalStateException("amount > remaining value");

}

if (transactions.stream().map(GiftCardTransaction::getTransactionId).

anyMatch(cmd.getTransactionId()::equals)) {

throw new IllegalStateException("TransactionId must be unique");

}

// 7.

remainingValue -= cmd.getAmount();

transactions.add(new GiftCardTransaction(id, cmd.getTransactionId(), cmd.getAmount()));

apply(new CardRedeemedEvent(id, cmd.getTransactionId(), cmd.getAmount()));

}

@EventHandler // 8.

protected void on(CardReimbursedEvent event) {

this.remainingValue += event.getAmount();

}

protected GiftCard() { } // 9.

}

上面的尝试显示了实现“礼品卡服务”的状态存储聚合。代码片段中编号的注释指出了Axon的具体细节，在这里进行解释：

①由于聚合存储在JPA存储库中，因此需要用@Entity对类进行注释。

②聚合根必须声明包含聚合标识符的字段。最迟必须在发布第一个事件时初始化此标识符。此标识符字段必须由@AggregateIdentifier注释进行注释。当使用JPA存储聚合时，Axon知道使用JPA提供的@Id注释。因为聚合是一个实体，@Id注释用在这里是一个很难懂的要求。

③此聚合有多个“聚合成员”。由于聚合是按原样存储的，因此应该考虑实体的正确映射。

④@CommandHandler带注释的构造函数，或者以不同的方式将“command handling constructor”放在一起。此注释告诉框架给定的构造函数能够处理IssueCardCommand。

⑤静态AggregateLifecycle.apply(Object...)可用于发布事件消息。调用此函数时，所提供的对象将在其应用的聚合范围内作为EventMessages发布。

⑥命令处理方法将首先决定传入的命令此时是否有效。

⑦在验证业务逻辑之后，可以调整聚合的状态

⑧通过定义@EventHandler带注释的方法，聚合中的实体可以监听聚合发布的事件。当事件消息在任何外部处理程序处理之前发布时，将调用这些方法。

⑨JPA要求一个无参数构造函数，未能提供此构造函数将导致加载聚合时出现异常。

*在命令处理程序中调整状态*

*与事件源聚合不同，状态存储聚合可以将决策逻辑与命令处理程序中的状态更改配对。在这个事件驱动模式中没有存储状态处理程序的结果。*

### **4.1.4从一个聚合创建另一个聚合**

通常，实例化一个新的聚合是通过发出一个由@CommandHandler注释的聚合构造函数处理的创建命令来完成的。例如，作为对某个事件的反应，这些命令可以由简单的REST端点或事件处理组件发布。然而，有时域描述了从另一个实体创建的某些实体。在这时从它的父聚合实例化一个子聚合会更贴近实际领域的情况。

*来自聚合用例的聚合*

*从“父”聚合创建“子”聚合的最合适的场景是，创建子聚合的决策位于父聚合的上下文中。例如，如果父级聚合包含可以驱动此子级创建决策的必要内容，则可以显示它自己。*

4.1.4.1如何从另一个聚合创建聚合

假设我们有一个ParentAggregate，在处理某个命令时将决定创建一个ChildAggregate。为此，ParentAggregate将如下所示：

public class ParentAggregate {

@CommandHandler

public void handle(SomeParentCommand command) {

createNew( ChildAggregate.class, () -> new ChildAggregate(

/\* provide required constructor parameters if applicable \*/)

);

}

// omitted no-op constructor, event sourcing handlers and other commandhandlers

}

AggregateLifecycle#createNew（Class<T>，Callable<T>）是实例化另一个聚合（如ChildAggregate）作为处理命令的反应的关键。createNew方法的第一个参数是要创建的聚合的类。第二个参数是factory方法，它期望结果是与给定类型相同的对象。

在这种情况下，ChildAggregate实现类似于以下格式：

import static org.axonframework.modelling.command.AggregateLifecycle.apply;

public class ChildAggregate {

public ChildAggregate(String aggregateId) {

apply(new ChildAggregateCreatedEvent(aggregateId));

}

// omitted no-op constructor, command and event sourcing handlers

}

请注意，ChildAggregateCreatedEvent显式地用于申明ChildAggregate已创建，而此信息将包含在ParentAggregate的SomeParentCommand命令处理程序中。

*从事件源处理程序创建聚合？*

*新聚合的创建应该在命令处理程序中完成，而不是在事件回溯处理程序中完成。这背后的基本原理是，当父聚合源于其事件时，您不希望创建新的子聚合，因为这将创建新的子聚合实例*

*但是，如果createNew方法在事件回溯处理程序中意外调用，则作为权宜之计将抛出UnsupportedOperationException。*

### **4.1.5聚合的多态**

在某些情况下，在聚合结构中具有多态层次结构是有益的。多态聚合层次结构中子类型从超级聚合继承@CommandHandlers、@EventSourcingHandlers和@CommandHandlerInterceptors。通过@AggregateIdentifier，系统将加载正确的聚合类型并对其执行命令。我们来看看下面的例子：

public abstract class Card {}

public class GiftCard extends Card {}

public class ClosedLoopGiftCard extends GiftCard {}

public class OpenLoopGiftCard extends GiftCard {}

public class RechargeableGiftCard extends ClosedLoopGiftCard {}

我们可以将这个结构定义为GiftCard类型的多态聚合，以及ClosedLoopGiftCard、OpenLoopGiftCard和RechargeableGiftCard的子类。如果Card类上存在处理程序，那么这些处理程序也将出现在所有聚合上。              在建模多态聚合层次结构时，请记住以下约束条件：

①不允许在抽象聚合上使用@CommandHandler注释构造函数。这样做的理由是，抽象的聚合永远无法创建。

②同样禁止在同一层次结构中的不同聚合上使用相同命令名的创建命令处理程序，因为Axon无法派生出要调用哪一个。

③在多态聚合层次结构中，不允许有多个@AggregateIdentifier和@AggregateVersion注释字段。

4.1.5.1注册聚合子类型

多态聚合层次结构可以通过aggregateConfiguer去调用aggregateConfiguer#registerSubtype（Class）来注册。请注意，未注册为子类型的父聚合的子级将自动注册为子类型。在下面的示例中ClosedLoopGiftCard被传递地注册为GiftCard的子类型。但是，如果定义了LimitedRechargeableGiftCard extends RecharableGiftcard，则不会提取它（除非显式注册为子类型）。

AggregateConfigurer<GiftCard> configurer = AggregateConfigurer.defaultConfiguration(GiftCard.class)

.withSubtype(OpenLoopGiftCard.class)

.withSubtype(RechargeableGiftCard.class);

*spring多态聚合*

*如果您使用的是Spring，多态层次结构将根据@Aggregate注释和类层次结构自动检测。*

### **4.1.6解决冲突**

明确变更含义的主要优点之一是可以更精确地检测冲突的变更。通常，当两个用户（几乎）同时对同一数据执行操作时，会发生这些冲突的更改。假设两个用户都在查看特定版本的数据。他们都决定对数据进行修改。它们都将发送一个命令，比如“在这个聚合的版本X上，do that”，其中X是聚合的预期版本。其中一个将实际应用于预期版本的更改。另一个用户不会。当聚合被另一个进程修改时，您可以检查用户的意图是否与任何未看到的更改冲突，而不是简单地拒绝所有传入的命令。

若要检测冲突，请将ConflictResolver类型的参数传递给聚合的@CommandHandler方法。此接口提供detectConflicts方法，允许您定义在执行特定类型的命令时被视为冲突的事件类型。

预期的聚合版本

请注意，如果使用预期版本加载了聚合，则冲突解决程序将仅包含任何潜在冲突事件。在命令的字段上使用@TargetAggregateVersion来指示聚合的预期版本。如果找到与谓词匹配的事件，则抛出异常（detectConflicts的可选第二个参数允许您定义要引发的异常）。如果没有找到，处理将继续正常进行。              如果没有调用detectConflicts，并且存在潜在的冲突事件，@CommandHandler将失效。如果提供了预期的版本，但@CommandHandler方法的参数中没有冲突解决程序，则可能会出现这种情况。