## **4.2命令适配器**

命令处理程序页面提供了如何在应用程序中处理命令消息的背景。调度过程是此类命令消息的起点。Axon提供了两个接口，可用于将命令发送到命令处理程序，即：commandBus，以及commandGateway。本页将显示如何以及何时使用gateway和bus命令。这里讨论了如何配置命令网关和总线实现以及具体细节

### **4.2.1命令总线**

“命令总线”是一种将命令发送到各自的命令处理程序的机制。因此，基础结构组件知道哪个组件可以处理哪个命令。每个命令始终只发送到一个命令处理程序。如果调度的命令没有可用的命令处理程序，则抛出NoHandlerForCommandException异常。CommandBus提供了两种方法来将命令分派到各自的处理程序，即dispatch（CommandMessage）和dispatch（CommandMessage，CommandCallback）方法：

private CommandBus commandBus; // 1.

public void dispatchCommands() {

String cardId = UUID.randomUUID().toString(); // 2.// 3. & 4.

commandBus.dispatch(

GenericCommandMessage.asCommandMessage(

new IssueCardCommand(cardId, 100, "shopId"))

);// 5. & 6.

commandBus.dispatch(

GenericCommandMessage.asCommandMessage(

new IssueCardCommand(cardId, 100, "shopId")),

(CommandCallback<IssueCardCommand, String>) (cmdMsg, cmdResultMsg) -> {// 7.

if (cmdResultMsg.isExceptional()) {

Throwable throwable = cmdResultMsg.exceptionResult();

} else {

String commandResult = cmdResultMsg.getPayload();

}

}

);

}

上面描述的CommandDispatcher举例说明了调度命令的几个重要方面和功能：

CommandBus接口提供发送命令消息的功能。

根据最佳实践，聚合标识符初始化为随机唯一标识符的字符串。

类型化标识符对象也是可能的，只要该对象实现一个合理的toString（）函数。

GenericCommandMessage.asCommandMessage（Object）方法用于创建CommandMessage。

为了能够在CommandBus上发送命令，您需要将您自己的命令对象（例如“commandmessagepayload”）包装在CommandMessage中。

CommandMessage还允许向命令消息添加元数据。

CommandBus的dispatch（CommandMessage）函数将在总线上分派所提供的CommandMessage，以便传递给命令处理程序。

如果应用程序对命令的结果不直接感兴趣，则可以使用此方法dispatch（CommandMessage）。

如果命令处理的结果与应用程序相关，则可以提供可选的第二个参数CommandCallback。

CommandCallback允许在命令处理完成时通知调度组件。

命令回调有一个函数onResult（CommandMessage，CommandResultMessage），该函数在命令处理完成后调用。

// 7 第一个参数是调度命令，而第二个参数是调度命令的执行结果。

最后，CommandCallback是一个“函数接口”，因为onResult是它唯一的方法。

像这样的，命令总线调度（commandMessage，（cmdMsg，commandResultMessage）->{/\*。。。\*/}）也是可能的。

CommandResultMessage提供API来验证命令执行是否异常或成功。

如果CommandResultMessage IsException返回true，则可以假定CommandResultMessage exceptionResult将返回包含实际异常的可丢弃实例。

否则，CommandResultMessage.getPayload方法可能会为您提供实际结果或null，如这里进一步指定的那样。

命令回调注意事项

在使用dispatch（CommandMessage，CommandCallback）的情况下，调用组件可能不会假定回调是在调度命令的同一线程中调用的。如果调用线程在继续之前依赖于结果，则可以使用FutureCallback。FutureCallback是Future的组合（定义见java.并发包）和Axon的CommandCallback。或者，考虑使用CommandGateway。

### **4.2.2命令网关**

“命令网关”是一种方便的调度命令的方法。它通过在CommandBus上发送命令时抽象某些方面来实现。它使用下面的CommandBus来执行消息的实际调度。

虽然不需要使用网关来分派命令，但这通常是最简单的选择。

CommandGateway接口可以分为两组方法，即send和sendAndWait：

private CommandGateway commandGateway; // 1.

public void sendCommand() {

String cardId = UUID.randomUUID().toString(); // 2.// 3.

CompletableFuture<String> futureResult =

commandGateway.send(new IssueCardCommand(cardId, 100, "shopId"));

}

// omitted class, constructor and result usage

如上所示的send API引入了两个概念，并用编号的注释进行了标记：

①CommandGateway接口，提供发送命令消息的功能。它通过内部利用CommandBus接口调度消息来实现这一点。

②根据最佳实践，聚合标识符初始化为随机唯一标识符的字符串。类型化标识符对象也是可能的，只要该对象实现一个合理的toString（）函数。

③send（Object）函数需要一个参数command Object。这是一种异步的命令调度方法。因此，send方法的响应是一个CompletableFuture。这允许在返回命令结果后链接后续操作。

使用send（Object）时回调

CommandGateway.send（Object）方法在后台使用FutureCallback从命令处理线程中解除对命令调度线程的阻塞。

通过使用sendAndWait方法，也可以实现同步发送消息的方法：

private CommandGateway commandGateway;

public void sendCommandAndWaitOnResult() {

IssueCardCommand commandPayload = new IssueCardCommand(UUID.randomUUID().toString(), 100, "shopId");// 1.

String result = commandGateway.sendAndWait(commandPayload); // 2.

result = commandGateway.sendAndWait(commandPayload, 1000, TimeUnit.MILLISECONDS);

}

// omitted class, constructor and result usage

①CommandGateway#sendAndWait（Object）函数接受一个参数，即command对象。它将无限期地等待，直到命令调度和处理过程得到解决。此方法返回的结果可以是成功的，也可以是异常的，这里将对此进行解释。

②如果不希望无限期等待，则可以在command对象旁边提供与“time unit”成对出现的“timeout”。这样做可以确保命令调度线程的等待时间不会超过指定的时间。如果使用此方法时命令调度/处理被中断或达到超时，则命令结果将为空。

在所有其他场景中，结果都遵循引用的方法。

### **4.2.3指挥调度结果**

一般来说，调度命令有两种可能的结果：

1.命令处理成功，和 2.命令处理异常

结果在某种程度上取决于调度过程，但更取决于命令处理程序的实现。因此，如果@CommandHandler带注释的函数由于某些业务逻辑而引发异常，则该异常将是调度命令的结果。

故意成功解析命令处理不应提供任何返回对象。因此，如果CommandBus/CommandGateway（直接或通过CommandResultMessage）提供响应，那么您应该假定成功的命令处理结果返回null。

虽然可以从命令处理程序返回结果，但应该尽量少用。该命令的目的绝不应该是检索值，因为这将表明该消息应设计为查询消息。例外情况是聚合根的标识符，或聚合根已实例化的实体的标识符。该框架在一个聚合的@CommandHandler注释构造函数上内置了一个这样的异常。如果“command handling constructor”成功执行，而不是聚合本身，则返回@AggregateIdentifier注释字段的值。

## **4.3 命令处理器**

### **4.3.1聚合命令处理程序**

虽然命令处理程序可以放置在常规组件中，但建议直接在包含处理此命令的状态的聚合上定义命令处理程序。

要在聚合中定义命令处理程序，只需用@CommandHandler注释应该处理命令的方法。带注释的@CommandHandler方法将成为命令消息的命令处理程序，其中命令名与该方法的第一个参数的完全限定类名匹配。因此，用@CommandHandler注释的void handle（redemercardcommand cmd）的方法标志将是redemercardcommand命令消息的命令处理程序。

命令消息也可以用不同的命令名进行调度。为了能够正确地处理它们，可以在@CommandHandler注释中指定字符串commandName值。

为了让Axon知道聚合类型的哪个实例应该处理命令消息，命令对象中包含聚合标识符的属性必须用@TargetAggregateIdentifier进行注释。注释可以放在命令对象中的字段或访问器方法（例如getter）上。

以GiftCard聚合为例，我们可以在聚合上标识两个命令处理程序：

public class GiftCard {

@AggregateIdentifier

private String id;

private int remainingValue;

@CommandHandler

public GiftCard(IssueCardCommand cmd) {

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

}

@CommandHandler

public void handle(RedeemCardCommand cmd) {

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

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

}

if (cmd.getAmount() > remainingValue) {

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

}

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

}

}

GiftCard句柄的Command对象IssueCardCommand和reviecardcommand具有以下格式：

public class IssueCardCommand {

@TargetAggregateIdentifier

private final String cardId;

private final Integer amount;

public IssueCardCommand(String cardId, Integer amount) {

this.cardId = cardId;

this.amount = amount;

}

}

public class RedeemCardCommand {

@TargetAggregateIdentifier

private final String cardId;

private final String transactionId;

private final Integer amount;

public RedeemCardCommand(String cardId, String transactionId, Integer amount) {

this.cardId = cardId;

this.transactionId = transactionId;

this.amount = amount;

}

}

这两个命令中的cardid都是对GiftCard实例的引用，因此使用@targetaggregatieIdentifier注释进行注释。创建聚合实例的命令不需要标识目标聚合标识符，因为还没有聚合存在。尽管如此，为了保持一致性，还是建议在它们上注释聚合标识符。

如果希望使用其他机制路由命令，则可以通过提供自定义CommandTargetResolver来重写该行为。此类应根据给定命令返回聚合标识符和预期版本（如果有）。

**聚合创建命令处理程序**

当@CommandHandler注释放在聚合的构造函数上时，相应的命令将创建该聚合的新实例并将其添加到存储库中。这些命令不需要针对特定的聚合实例。因此，这些命令不需要任何@TargetAggregateIdentifier或@TargetAggregateVersion注释，也不需要为这些命令调用自定义CommandTargetResolver。

但是，不管命令的类型如何，只需通过例如Axon服务器分发应用程序，强烈建议在给定的消息上指定路由键。@TargetAggregateIdentifier是这样加倍的，但是如果没有值得注释的字段，则应添加@RoutingKey注释，以确保可以路由命令。此外，可以配置不同的路由策略，如在命令调度部分中进一步指定的那样。

### **4.3.2业务逻辑和状态变化**

在聚合中，有一个特定的位置来执行业务逻辑验证和聚合状态更改。命令处理程序应该决定聚合是否处于正确的状态。如果是，则发布一个事件。否则，可能会忽略该命令或引发异常，具体取决于域的需要。

任何命令处理函数中都不应发生状态更改。事件源处理程序应该是更新聚合状态的唯一方法。如果不这样做，就意味着它的状态会发生变化。

聚合测试设备将防止命令处理功能中的意外状态更改。因此，建议提供全面的测试案例。

何时处理事件

聚合需要的唯一状态是它需要做出决策的状态。因此，只有在需要事件类似的状态更改以驱动将来的验证时，才需要处理由聚合发布的事件。

### **4.3.3从事件回溯处理程序应用事件**

在某些情况下，尤其是当聚合结构超出了几个实体时，对同一聚合的其他实体中发布的事件作出反应会更为清晰（这里将更详细地解释多实体聚合）。但是，由于事件处理方法也在重构聚合状态时被调用，因此必须采取特殊的预防措施。

可以在事件回溯处理程序方法内部应用（）新事件。这使得实体“B”可以应用事件来响应实体“A”正在做的事情。在获取给定聚合时，Axon将在重放历史事件时忽略apply（）调用。请注意，在从事件回溯处理程序发布事件消息的场景中，内部apply（）调用的事件仅在所有实体接收到第一个事件之后才发布到实体。如果需要根据应用内部事件后实体的状态发布更多事件，请使用apply（…）.和enapply（…）。

对其他事件的反应

聚合本身无法处理来自其他源的事件，这是特意为之，因为事件源处理程序用于重新创建聚合的状态。为此，它只需要它自己的事件，因为这些事件代表它的状态变化。

为了使聚合对来自其他聚合实例的事件做出反应，应该利用saga或事件处理组件

### **4.3.4聚合命令处理程序创建策略**

到目前为止，我们已经用大约两种类型的命令处理程序描述了GiftCard聚合：

1.@CommandHandler带注释的构造函数

2.@CommandHandler带注释的方法

选项1将始终期望是GiftCard聚合的实例化，而选项2期望针对现有的聚合实例。虽然这可能是默认值，但可以选择在命令处理程序上定义创建策略。这可以通过将@CreationPolicy注释添加到命令处理程序注释的方法来实现，如下所示：

public class GiftCard {

public GiftCard() {// Required no-op constructor}

@CommandHandler

@CreationPolicy(AggregateCreationPolicy.ALWAYS)

public void handle(IssueCardCommand cmd) {

// An `IssueCardCommand`-handler which will create a `GiftCard` aggregate

}

@CommandHandler

@CreationPolicy(AggregateCreationPolicy.CREATE\_IF\_MISSING)

public void handle(CreateOrRechargeCardCommand cmd) {

// A 'CreateOrRechargeCardCommand'-handler which creates a `GiftCard` aggregate if it did not exist

// Otherwise, it will update an existing `GiftCard` aggregate.

}

}

如上所示，@CreationPolicy注释需要声明AggregateCreationPolicy。此枚举有以下可用选项：

①ALWAYS-创建策略“ALWAYS”将期望实例化聚合。这实际上就像命令处理程序注释的构造函数。在不定义返回类型的情况下，将返回创建过程中使用的聚合标识符。通过这种方法，可以在聚合标识符旁边返回其他结果。

②CREATE\_IF\_MISSING-创建策略“CREATE IF MISSING”可以创建聚合，也可以对现有实例执行操作。此策略应视为创建或更新聚合的方法。

③NEVER-创建策略“将永远不会在现有策略的创建”上处理“聚合”实例。这与任何常规的命令处理程序注释方法一样有效。

### **4.3.5外部命令处理器**

命令处理函数通常直接放在聚合上（如这里更详细的描述）。但是，在某些情况下，不可能也不希望将命令直接路由到聚合实例。但是，消息处理函数，如命令处理程序，可以放在任何对象上。因此，可以实例化“命令处理对象”。

命令处理对象是一个简单（常规）对象，它有@CommandHandler注释的方法。与聚合不同的是，命令处理对象只有一个实例，它处理它在其方法中声明的所有命令类型：

public class GiftCardCommandHandler { // 1.

private final Repository<GiftCard> giftCardRepository;

@CommandHandler

public void handle(RedeemCardCommand cmd) {

giftCardRepository.load(cmd.getCardId()) // 2.

.execute(giftCard -> giftCard.handle(cmd)); // 3.

}

}

在上面的代码片段中，我们决定不再在GiftCard上直接处理rewemcardcommand。相反，我们手动加载GiftCard并在其上执行所需的方法：

①礼品卡聚合的存储库，用于检索和存储聚合。如果@CommandHandler方法直接放在聚合上，Axon将自动知道调用存储库来加载给定的实例。因此，直接访问存储库并不是一个强制性的选择。

②要加载预期的GiftCard聚合实例，将使用Repository.load（String）方法。提供的参数应为聚合标识符。

③在加载该聚合之后，应该调用Aggregate.execute（Consumer）函数来对聚合执行操作。使用execute函数确保正确启动聚合生命周期。

## **4.4 部分实现方法**

## **4.4 部分实现方法**

如调度命令页面所示，命令调度有许多优点。首先，只有一个对象可以清楚地描述客户机的意图。通过记录命令，可以存储意图和相关数据，以备将来参考。命令处理还可以使您很容易地通过web服务向远程客户机公开命令处理组件。测试也变得容易得多。您可以通过列出一些事件和命令来定义测试脚本，方法是定义启动情况（给定）、要执行的命令（何时）和预期结果（之后）（有关更多信息，请参阅测试）。最后一个主要优点是，在同步和异步以及本地与分布式命令处理之间切换非常容易。

这并不意味着使用显式命令对象进行命令调度是唯一的方法。Axon的目标不是规定一种特定的工作方式，而是支持您按照自己的方式来做，同时提供最佳实践作为默认行为。仍然可以使用可以调用的服务层来执行命令。该方法只需要启动一个工作单元（请参阅工作单元），并在该方法完成后对其执行提交或回滚。

接下来的部分将概述与使用Axon框架建立命令调度基础结构相关的任务。

### **4.4.1命令网关**

命令网关是一个面向命令调度机制的方便接口。虽然不需要使用网关来分派命令，但这通常是最简单的选择。

有两种方法可以使用命令网关。第一种方法是使用CommandGateway接口和Axon提供的DefaultCommandGateway实现。commandgateway提供了许多方法，允许您发送命令并以同步、超时或异步方式等待结果。

另一种选择可能是最灵活的。您可以使用CommandGatewayFactory将几乎任何接口转换为命令网关。这允许您使用强类型和声明自己（选中的）业务异常来定义应用程序的接口。Axon将在运行时为该接口自动生成一个实现。

4.4.1.1配置命令网关

您的自定义命令网关和Axon提供的网关至少都需要配置一个命令总线。此外，命令网关可以配置RetryScheduler、CommandDispatchInterceptors和CommandCallbacks。

4.4.1.1.1调度重试

RetryScheduler能够在命令执行失败时调度重试。当命令由于显式非暂时性的异常而失败时，根本不会重试。请注意，只有当命令因运行时异常而失败时，才会调用重试计划程序。已检查的异常被视为“业务异常”，不会触发重试。

目前有两种实现：

IntervalRetryScheduler将按设置的间隔重试给定的命令，直到成功为止，或者已达到最大重试次数。

ExponentialBackOffIntervalRetryScheduler将以指数后退间隔重试失败的命令，直到它成功，或者已经进行了最大的重试次数。

4.4.1.1.2指挥调度拦截器

CommandDispatchInterceptors允许在将命令消息发送到命令总线之前对其进行修改。与在命令总线上配置的CommandDispatchInterceptors不同，这些拦截器只在消息通过此网关发送时被调用。例如，这些拦截器可用于将元数据附加到命令或执行验证。

4.4.1.1.3命令回调

命令回调可以在常规发送时提供给命令网关，指定如何处理某个命令处理的结果。它与CommandMessage和CommandResultMessage类一起工作，因此通过此网关发送的所有命令的某些通用行为，不管其类型如何都会被允许。

4.4.1.2创建自定义的命令网关

Axon允许自定义接口用作命令网关。接口中声明的每个方法的行为都基于参数类型、返回类型和声明的异常。使用这个网关不仅方便，而且通过允许您在需要的地方模拟您的接口，它使测试更加容易。

以下是影响命令网关的行为的参数：

①第一个参数应该是要分派的实际命令对象。

②用@MetaDataValue注释的参数将把它们的值赋给元数据字段，标识符作为注释参数传递

③元数据类型的参数将与CommandMessage上的元数据合并。后一个参数定义的元数据将覆盖前一个参数的元数据（如果它们的键相等）。

④CommandCallback类型的参数将使onResult（CommandMessage<？扩展C>，CommandResultMessage<？扩展R>）方法在处理命令后调用。尽管CommandCallback提供了一种处理命令处理结果的方法，但这并不影响您是否可以在自定义命令网关上定义返回类型。如果同时定义了回调和返回类型，则回调的调用将始终与返回值（或异常）匹配。最后，要知道您可能传入几个CommandCallback实例，这些实例都将按顺序调用。

⑤最后两个参数表示超时，可以是long（或int）和TimeUnit类型。该方法将阻塞这些参数指示的时间。方法对超时的反应取决于方法上声明的异常（见下文）。请注意，如果方法的其他属性完全阻止阻塞，则永远不会发生超时。

方法的声明返回值也会影响其行为：

①void返回类型将导致该方法立即返回，除非该方法上有其他需要等待的指示，例如超时或声明的异常。

②Future、CompletionStage和CompletableFuture的返回类型将导致该方法立即返回。可以使用从方法返回的CompletableFuture实例访问命令处理程序的结果。方法上声明的异常和超时将被忽略。

③任何其他返回类型都会导致方法阻塞，直到结果可用。结果强制转换为返回类型（如果类型不匹配，则导致ClassCastException）。

例外情况具有以下效果：

①如果命令处理程序（或拦截器）引发了该类型的异常，则将引发任何已声明的已检查异常。如果抛出未声明的已检查异常，则它将被封装在CommandExecutionException中，该异常是RuntimeException。

②当发生超时时，默认行为是从方法返回null。这可以通过声明TimeoutException来更改。如果声明了此异常，则会引发TimeoutException。

③当线程在等待结果时被中断时，默认行为是返回null。在这种情况下，中断标志在线程上被设置回去。通过在方法上声明InterruptedException，此中断行为将改为引发该异常。当抛出异常时，中断标志被移除，这与java规范一致。

④可以在方法上声明其他运行时异常，但除了向API用户澄清之外，不会产生任何效果。

最后，还可以使用注释：

①正如在parameter部分中指定的，参数的@MetaDataValue注释将把该参数的值作为元数据值添加。元数据项的键作为注释的参数提供。

②用@Timeout注释的方法最多将阻塞指定的时间量。如果方法声明超时参数，则忽略此注释。

③用@Timeout注释的类将导致在该类中声明的所有方法最多阻塞指定的时间量，除非使用自己的@Timeout注释或指定Timeout参数。

### **4.4.2命令总线**

命令总线是在Axon应用程序中将命令发送到各自的命令处理程序的机制。在这里可以找到如何使用总线的建议。该框架中存在几种不同特点的命令总线：

**（1）AxonServer命令总线**

Axon提供了一个现成的命令总线，即AxonServerCommandBus。它连接到axoniqaxonserver服务器来提交和接收命令。AxonServerCommandBus是一种分布式命令总线。默认情况下，它使用SimpleCommandBus处理不同JVM上的传入命令。

※Axon配置API

依赖：

<!--somewhere in the POM file-->

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-server-connector</artifactId>

<version>${axon.version}</version>

</dependency>

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-configuration</artifactId>

<version>${axon.version}</version>

</dependency>

配置应用

// 返回配置器实例，其中包含已配置的默认组件。

//`AxonServerCommandBus`默认配置为命令总线。

Configurer configurer = DefaultConfigurer.defaultConfiguration();

※Springboot自动配置

通过声明axon-spring-boot-starter依赖，spring可以自动配置Axon Server Command Bus：

依赖

<!--somewhere in the POM file-->

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-spring-boot-starter</artifactId>

<version>${axon.version}</version>

</dependency>

排除Axon服务器连接器

如果要排除服务器依赖项，请参阅下面的“非服务器连接”命令。

**（2）SimpleCommand总线**

顾名思义，SimpleCommandBus是最简单的实现。它在分派命令的线程中直接处理命令。处理命令后，将保存修改后的聚合，并在同一线程中发布生成的事件。在大多数情况下，例如web应用程序，这种实现将满足您的需要。

与大多数CommandBus实现一样，SimpleCommandBus允许配置拦截器。当命令在命令总线上调度时，将调用CommandDispatchInterceptors。CommandHandlerInterceptors在实际的命令处理程序方法之前被调用，允许您修改或阻止命令。有关详细信息，请参见命令拦截器。

由于所有命令处理都是在同一个线程中完成的，所以这个实现仅限于JVM的边界。这个实现的性能是好的，但不是特别的。要跨越JVM边界，或者要最大限度地利用CPU周期，请查看其他CommandBus实现。

※axon配置

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureCommandBus( c->SimpleCommandBus.builder()

.transactionManager(c.getComponent(TransactionManager.class))

.messageMonitor(c.messageMonitor(SimpleCommandBus.class,"commandBus"))

.build()

);

※springboot配置

@Bean

public SimpleCommandBus commandBus(TransactionManager txManager, AxonConfiguration axonConfiguration) {

SimpleCommandBus commandBus =

SimpleCommandBus.builder()

.transactionManager(txManager)

.messageMonitor(axonConfiguration.messageMonitor(CommandBus.class,"commandBus"))

.build();

commandBus.registerHandlerInterceptor(

new CorrelationDataInterceptor<>(axonConfiguration.correlationDataProviders())

);

return commandBus;

}

*排除Axon服务器连接器*

如果从axon spring boot starter依赖项中排除axon服务器连接器依赖项，那么SimpleCommandBus将为您自动配置。

（3）异步命令总线

顾名思义，AsynchronousCommandBus实现从分派命令的线程异步执行命令。它使用一个执行器在不同的线程上执行实际的处理逻辑。

默认情况下，AsynchronousCommandBus使用无边界缓存线程池。这意味着在调度命令时会创建一个线程。已处理完命令的线程将重新用于新命令。如果线程在60秒内未处理命令，则停止线程。或者，可以提供一个Executor实例来配置不同的线程策略。

请注意，在停止应用程序时，应该关闭AsynchronousCommandBus，以确保所有等待的线程都已正确关闭。要关闭，请调用shutdown（）方法。如果实现了ExecutorService接口，这也将关闭任何提供的Executor实例。

※Axon配置API

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureCommandBus(c -> AsynchronousCommandBus.builder().transactionManager(c.getComponent(TransactionManager.class))

.messageMonitor(c.messageMonitor(AsynchronousCommandBus.class, "commandBus"))

.build());

※springboot自动配置

@Bean

public AsynchronousCommandBus commandBus(TransactionManager txManager, AxonConfiguration axonConfiguration) {

AsynchronousCommandBus commandBus =AsynchronousCommandBus.builder()

.transactionManager(txManager)

.messageMonitor(axonConfiguration.messageMonitor(AsynchronousCommandBus.class,"commandBus"))

.build();

commandBus.registerHandlerInterceptor(new CorrelationDataInterceptor<>(axonConfiguration.correlationDataProviders()));

return commandBus;

}

（4）DisruptorCommandBus

SimpleCommandBus具有合理的性能特性。SimpleCommandBus需要锁定以防止多个线程同时访问同一个聚合，这会导致处理开销和锁争用。

DisruptorCommandBus对多线程处理采用了不同的方法。不同于每个线程都执行同一个进程，而是有多个线程，每个线程负责进程的一部分。DisruptorCommandBus使用Disruptor，一个用于并发编程的小框架，通过对多线程采用不同的方法来实现更好的性能。不是在调用线程中进行处理，而是将任务交给两组线程，每个线程负责处理的一部分。第一组线程将执行命令处理程序，更改聚合的状态。第二个组将存储事件并将其发布到事件存储。

而DisruptorCommandBus很容易比SimpleCommandBus高出4倍（！），但有一些限制：

DisruptorCommandBus只支持事件源聚合。这个命令总线还可以作为中断程序处理的聚合的存储库。要获取对存储库的引用，请使用createRepository（AggregateFactory）。

命令只能导致单个聚合实例中的状态更改。

当使用缓存时，它只允许对给定标识符进行单个聚合。这意味着不可能有两个具有相同标识符的不同类型的聚合。

命令通常不能导致需要回滚工作单元的故障。当发生回滚时，DisruptorCommandBus不能保证命令按照它们被调度的顺序进行处理。此外，它还需要重试许多其他命令，从而导致不必要的计算。

在创建新的聚合实例时，更新该已创建实例的命令可能并非都按所提供的顺序进行。一旦创建了聚合，所有命令都将按照它们被调度的顺序执行。要确保顺序，请在creating命令上使用回调来等待正在创建的聚合。不会超过几毫秒。

要构建DisruptorCommandBus实例，您需要一个EventStore。此组件在事件总线和事件存储部分中进行了说明。

或者，您可以提供DisruptorConfiguration实例，该实例允许您调整配置以优化特定环境的性能：

缓冲区大小-环形缓冲区上用于注册传入命令的插槽数。较高的值可能会增加吞吐量，但也会导致更高的延迟。必须总是2的幂。默认为4096。

ProducerType-指示条目是由单个线程生成还是由多个线程生成。默认为多个。

WaitStrategy-当处理器线程（负责实际处理的三个线程）需要彼此等待时使用的策略。最佳的等待策略取决于机器中可用的内核数量以及正在运行的其他进程的数量。如果低延迟非常重要，并且DisruptorCommandBus可能会为自己声明核心，那么可以使用BusySpinWaitStrategy。要使命令总线占用更少的CPU并允许其他线程进行处理，请使用YieldingWaitStrategy。最后，您可以使用SleepingWaitStrategy和BlockingWaitStrategy来允许其他进程公平地共享CPU。如果命令总线不需要全时处理，则后者适用。默认为BlockingWaitStrategy。

Executor-设置为DisruptorCommandBus提供线程的执行器。这个执行器必须能够提供至少四个线程。DisruptorCommandBus的处理组件声明了其中三个线程。额外的线程用于调用回调和调度重试，以防检测到聚合的状态已损坏。默认为CachedThreadPool，该池提供来自名为“DisruptorCommandBus”的线程组的线程。

TransactionManager—定义事务管理器，该事务管理器应确保在事务中执行事件的存储和发布。

InvokerInterceptors-定义调用过程中要使用的CommandHandlerInterceptors。这是调用实际命令处理程序方法的进程。

PublisherInterceptors-定义要在发布过程中使用的CommandHandlerInterceptors。这是存储和发布生成的事件的过程。

RollbackConfiguration-定义工作单元应该回滚的异常。默认为对未检查异常回滚的配置。

RescheduleCommandsOnCorruptState-指示是否应重新调度针对已损坏的聚合（例如，由于工作单元已回滚）而执行的命令。如果为false，则回调onFailure（）方法。如果为true（默认值），将改为重新调度命令。

CoolingDownPeriod-设置为确保所有命令都已处理而等待的秒数。在冷却期间，不接受新命令，但会处理现有命令，并在必要时重新调度。冷却期确保线程可用于重新调度命令和调用回调。默认值为1000（1秒）。

缓存-设置缓存，该缓存存储已从事件存储重建的聚合实例。缓存用于存储中断程序未在活动使用中的聚合实例。

InvokerThreadCount—分配给命令处理程序调用的线程数。最好是设备核心数量的一半。

PublisherThreadCount-用于发布事件的线程数。一个好的初始值应设为核心数量的一半，如果在I/O上花费大量的时间，这个起点可能会增加。

SerializerThreadCount—用于预序列化事件的线程数。默认值为1，但如果未配置序列化程序，则忽略此值。

序列化程序-要执行预序列化的序列化程序。配置序列化程序后，DisruptorCommandBus将在支持序列化的消息中包装所有生成的事件。

在将有效负载和元数据发布到事件存储之前，将附加它们的序列化形式。

Axon配置API

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureCommandBus(c ->DisruptorCommandBus.builder()

.transactionManager(c.getComponent(TransactionManager.class))

.messageMonitor(c.messageMonitor(DisruptorCommandBus.class, "commandBus"))

.bufferSize(4096)

.build()

);

Springboot自动配置

@Bean

public DisruptorCommandBus commandBus(TransactionManager txManager, AxonConfiguration axonConfiguration) {

DisruptorCommandBus commandBus = DisruptorCommandBus.builder()

.transactionManager(txManager)

.messageMonitor(axonConfiguration.messageMonitor(DisruptorCommandBus.class, "commandBus"))

.build();

commandBus.registerHandlerInterceptor(

new CorrelationDataInterceptor<>(axonConfiguration.correlationDataProviders()));

return commandBus;

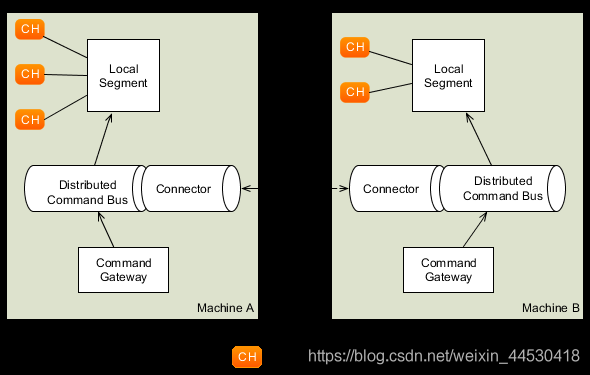
}

4.4.2.1分配命令总线

有时，您希望不同jvm中的多个命令总线实例充当一个实例。在一个JVM的命令总线上调度的命令应该无缝地传输到另一个JVM中的命令处理程序，同时将任何结果发送回。这就是“分布式命令总线”概念的由来。分布式命令总线的默认实现是AxonServerCommandBus。它连接到axoniqaxonserver服务器来提交和接收命令。与其他CommandBus实现不同，AxonServerCommandBus根本不调用任何处理程序。它所做的只是在不同JVM上的命令总线实现之间形成一个“桥梁”。默认情况下，SimpleCommandBus配置为处理不同JVM上的传入命令。您可以将AxonServerCommandBus配置为使用其他命令总线实现：AsynchronousCommandBus、DisruptorCommandBus。

4.4.2.2分布式命令总线

DistributedCommandBus是分发命令总线（commands）的另一种方法。每个JVM上的DistributedCommandBus实例称为“段”。



以上是分布式命令总线的框架，DistributedCommandBus依赖于两个组件：CommandBusConnector实现JVM之间的通信协议，CommandRouter为每个传入的命令选择一个目标。这个路由器根据路由策略计算出的路由密钥，定义了应该给分布式commandbus的哪一段分配命令。具有相同路由键的两个命令将始终路由到同一段，只要段的数量和配置没有更改。通常，目标聚合的标识符用作路由密钥。

Axon提供了两个RoutingStrategy的实现：MetaDataRoutingStrategy，它使用命令消息中的元数据属性来查找路由键；AnnotationRoutingStrategy，它使用命令消息有效负载上的@targetaggregatieIdentifier注释来提取路由密钥。显然，您还可以自己实现。

默认情况下，当无法从命令消息解析任何键时，RoutingStrategy实现将抛出异常。可以通过在MetaDataRoutingStrategy或AnnotationRoutingStrategy的构造函数中提供未解析的RoutingKeyPolicy来更改此行为。有三种可能的策略：

ERROR-默认值，当路由键不可用时将引发异常

RANDOM\_KEY-当无法从命令消息解析“routing KEY”时，将返回一个随机值。这实际上意味着这些命令将被路由到命令总线的随机段。

STATIC\_KEY-将为未解析的路由密钥返回一个静态密钥（“未解析”）。这实际上意味着，只要段的配置不变，所有这些命令都将路由到同一段。

您可以选择其中一个扩展模块中提供的此组件的不同风格实现：

SpringCloud或

JGroup。

配置分布式命令总线（大多数情况下）可以在不修改配置文件的情况下完成。

首先，需要包括一个Axon分布式命令总线模块的启动程序（例如JGroups或springcloud）。一旦存在，就需要向应用程序上下文添加一个属性，以启用分布式命令总线：axon.distributed.enabled=true，有一个配置独立于连接器的类型存在：axon.distributed.load-factor=100，这表示分布式命令总线的默认负载系数为100.

负载系数的解释

负载系数定义了实例与其他实例相比将承载的负载量。例如，如果您设置了两台机器，两台机器的负载系数均为100，则两台机器都将承载相同数量的负载。将两台机器的负载系数增加到200仍然意味着两台机器接收到相同的负载量。最后，负载因子旨在为异构应用程序环境提供服务，从而将更多的负载分配给速度更快的机器，而不是分配给速度较慢的机器。

## **4.5配置**

本页旨在描述用于配置命令模型的一套选项。

### **4.5.1聚合配置**

命令模型中的核心概念是实现的聚合。要实例化默认聚合配置，只需执行以下操作：

**Axon配置API**

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureAggregate(GiftCard.class);

**Springboot配置**

@Aggregate注释（在org.axonframework.spring.stereotype package）触发自动配置以设置必要的组件，以将带注释的类型用作聚合。请注意，只需要对聚合根进行注释。

Axon将用命令总线自动注册所有带@CommandHandler注释的方法，如果没有，则设置一个存储库。

@Aggregate

public class GiftCard {

@AggregateIdentifier

private String id;

@CommandHandler

public GiftCard(IssueCardCommand cmd) {

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

}

}

### **4.5.2注册命令处理器**

通常，命令处理程序函数直接放在聚合上。采用这种方法时，只需按上述方法注册聚合就足以注册其所有命令处理程序方法。

但是，外部命令处理程序需要直接注册为命令处理程序，如下例所示：

**Axon配置API**

假定存在以下命令处理程序：

public class GiftCardCommandHandler {

private final Repository<GiftCard> giftCardRepository;

@CommandHandler

public void handle(RedeemCardCommand cmd) {

giftCardRepository.load(cmd.getCardId())

.execute(giftCard -> giftCard.handle(cmd));

}

// omitted constructor

}

要将GiftCardCommandHandler注册为命令处理程序，需要执行以下操作：

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

.registerCommandHandler(conf -> new GiftCardCommandHandler());

或者，可以使用更通用的方法在组件中注册所有类型的消息处理程序：

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

.registerMessageHandler(conf -> new GiftCardCommandHandler());

重复的命令处理函数

正如在“消息传递概念”部分中指定的，命令始终只有一个目的地。这意味着对于任何给定的命令都应该只有一个命令处理程序方法。默认情况下，当注册重复的命令处理程序方法时，将保留最后一次注册并记录警告。可以通过指定不同的DuplicateCommandHandlerResolver来调整此行为，如运行时优化部分中所述。

**当使用Spring Boot时，只需将命令处理程序指定为bean就足够了:**

@Component

public class GiftCardCommandHandler {

private final Repository<GiftCard> giftCardRepository;

@CommandHandler

public void handle(RedeemCardCommand cmd) {

giftCardRepository.load(cmd.getCardId()) .execute(giftCard -> giftCard.handle(cmd));

}}

### **4.5.3命令模型存储库**

存储库是提供对聚合的访问的机制。存储库充当通向用于持久化数据的实际存储机制的网关。在CQRS中，存储库只需要能够根据其唯一标识符查找聚合。任何其他类型的查询都应该针对查询数据库执行。

在Axon框架中，所有存储库都必须实现存储库接口。该接口规定了三种方法：load（identifier，version）、load（identifier）和newInstance（factoryMethod）。load方法允许您从存储库加载聚合。可选版本参数用于检测并发修改（请参阅冲突解决）。newInstance用于在存储库中注册新创建的聚合。

根据您的底层持久性存储和审计需求，有许多基本实现提供了大多数存储库所需的基本功能。Axon框架区分了保存聚合当前状态的存储库（请参阅标准存储库）和存储聚合事件的存储库（请参阅事件回溯存储库）。

请注意，Repository接口没有指定delete（identifier）方法。通过调用AggregateLifecycle.markDeleted（）聚合中的方法。删除聚合与其他任何聚合一样都是一种状态迁移，唯一的区别是在许多情况下它是不可逆的。您应该在聚合上创建自己有意义的方法，将聚合的状态设置为“已删除”。这还允许您注册任何要发布的事件。

Axon配置

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureAggregate(

AggregateConfigurer.defaultConfiguration(GiftCard.class)

.configureRepository(c -> EventSourcingRepository.builder(GiftCard.class)

.eventStore(c.eventStore())

.build())

);

Springboot自动装配

要完全自定义所使用的存储库，可以在应用程序上下文中定义一个。要使Axon框架将此存储库用于预期的聚合，请在@aggregate注释的repository属性中定义存储库的bean名称。或者，将存储库的bean名称指定为聚合的名称（第一个字符小写），后缀为repository。因此，对于GiftCard类型的类，默认的存储库名称是giftCardRepository。如果找不到具有该名称的bean，那么Axon将定义一个EventSourcingRepository（如果没有EventStore可用，则失败）。

@Bean

public Repository<GiftCard> repositoryForGiftCard(EventStore eventStore) {

return EventSourcingRepository.builder(GiftCard.class).eventStore(eventStore).build();

}

@Aggregate(repository = "repositoryForGiftCard")

public class GiftCard { /\*...\*/ }

请注意，这需要对存储库进行完整配置，包括任何可能已自动配置的SnapshotTriggerDefinition或AggregateFactory。

4.5.3.1标准存储库

标准存储库存储聚合的实际状态。每次更改后，新状态将覆盖旧状态。这使得应用程序的查询组件可以使用命令组件也使用的相同信息。这可能是最简单的解决方案，这取决于您正在创建的应用程序的类型。如果是这样，Axon提供了一些构建块来帮助您实现这样一个存储库。

Axon为标准存储库提供了一个现成的实现：GenericJpaRepository。它期望聚合是一个有效的JPA实体。它配置了一个EntityManagerProvider（提供EntityManager来管理实际的持久性），以及一个指定存储在存储库中的实际聚合类型的类。当聚合调用static时，还将传递事件要发布到的EventBusAggregateLifecycle.apply（）方法。

您还可以轻松地实现自己的存储库。在这种情况下，最好从抽象锁库扩展。作为聚合包装类型，建议使用AnnotatedAggregate。请参阅GenericJpaRepository的源代码以获取示例。

4.5.3.2事件回溯存储库

能够基于事件重构其状态的聚合根也可以配置为由事件回溯存储库加载。这些存储库不存储聚合本身，而是存储由聚合生成的一系列事件。根据这些事件，可以随时恢复聚合的状态。

EventSourcingRepository实现提供了Axon框架中任何事件回溯存储库所需的基本功能。它依赖于EventStore（参见EventStore实现），它抽象了事件的实际存储机制。

### **4.5.4聚合工厂**

或者，您可以提供一个聚合工厂。AggregateFactory指定如何创建聚合实例。一旦创建了聚合，EventSourcingRepository就可以使用它从事件存储中加载的事件来初始化它。Axon框架附带了许多可以使用的AggregateFactory实现。如果它们还不够，也很容易创建自己的实现。

4.5.4.1通用聚合工厂

GenericAggregateFactory是一个特殊的AggregateFactory实现，可用于任何类型的事件源聚合根。GenericAggregateFactory创建存储库管理的聚合类型的实例。聚合类必须是非抽象的，并声明一个根本不初始化的默认无参数构造函数。

GenericAggregateFactory适用于聚合不需要特殊注入不可序列化资源的大多数场景。

4.5.4.2 Spring原型聚合工厂

根据您的体系结构选择，使用Spring将依赖项注入到聚合中可能会很有用。例如，可以将查询存储库注入聚合中，以确保某些值的存在（或不存在）。

要将依赖项注入聚合，您需要在Spring上下文中配置聚合根的原型bean，该bean还定义SpringPrototypeAggregateFactory。它没有使用构造函数创建常规实例，而是使用Spring应用程序上下文实例化聚合。这还将在聚合中注入任何依赖项。

4.5.4.3实现自己的聚合工厂

在某些情况下，GenericAggregateFactory只是不能提供您需要的东西。例如，您可以有一个抽象聚合类型，该类型具有针对不同场景的多个实现（例如PublicUserAccount和BackOfficeAccount都扩展了一个帐户）。您不必为每个聚合创建不同的存储库，而是可以使用单个存储库，并配置一个能够识别不同实现的AggregateFactory。

聚合工厂所做的大部分工作是创建未初始化的聚合实例。它必须使用给定的聚合标识符和流中的第一个事件来执行此操作。通常，此事件是一个创建事件，其中包含有关预期聚合类型的提示。您可以使用此信息来选择实现并调用其构造函数。请确保该构造函数未应用任何事件；聚合必须未初始化。

与直接加载简单存储库实现的聚合相比，基于事件初始化聚合可能是一项耗时的工作。CachingEventSourcingRepository提供一个缓存，可以从中加载聚合（如果可用）。