1 Anatomy of a Message 信息剖析

在Axon中，组件之间的所有通信都是通过显式消息完成的，由消息接口表示。消息由有效负载和元数据组成，有效负载是表示实际功能消息的特定于应用程序的对象，而元数据是描述消息上下文的键值对。

每个消息的子接口代表一个特定类型的消息，并定义了描述该消息的附加信息。与元数据不同，此附加信息定义了正确处理该类型消息所需的信息。

消息是不可变的。这意味着，要添加元数据元素，您可以有效地创建一个新的消息实例，并添加一个(或其他)元数据元素。为了仍然能够将一条消息的两个java实例考虑为表示相同的概念性消息，每个消息都有一个标识符。更改消息的元数据不会更改此标识符。

**Meta Data**

消息的元数据通常描述生成消息的上下文。例如，元数据可能包含有关导致生成此消息的消息的信息(例如，命令处理程序根据传入命令生成事件)。

在Axon中，元数据表示为字符串到对象的映射。虽然您可以随意添加任何类型的对象作为元数据值，但我们强烈建议坚持使用原语和字符串(或者最多是简单对象)。当涉及到结构更改时，元数据不具有与有效负载相同的灵活性

与常规的Map<String,Object>不同，Axon中的元数据是不可变的。改变方法将创建并返回一个新的实例，而不是修改一个现有的实例。

MetaData metaData = MetaData.with("myKey", 42) // 1

.and("otherKey", "some value"); // 2

1 使用指定的键-值对创建元数据实例

2 添加一个额外的键-值对，返回一个元数据的新实例。

消息中的元数据的工作原理类似:

EventMessage eventMessage =

GenericEventMessage.asEventMessage("myPayload") // 1

.withMetaData(singletonMap("myKey", 42)) // 2

.andMetaData(singletonMap("otherKey", "some value")); // 2

1 创建一个以“myPayload”作为有效负载的EventMessage

2 withMetaData用给定的映射替换消息中的任何元数据。在本例中，java.util.Collections.singletonMap()用于定义单个条目。

3 andMetaData将给定映射中的条目添加到消息的元数据中。具有相同键值的现有条目将被覆盖。

注意:关于实现Map<String的元数据，对象>元数据也实现了Map<String，对象>，这意味着除了传递一个singletonMap(或任何其他类型的Map)，你还可以使用元数据。(关键字,值)。由于元数据是不可变的，Map的所有变异操作都会抛出UnsupportedOperationException。

**于消息数据**

某些类型的消息提供额外的信息(在消息提供的信息之上)。例如，EventMessage(一个接口扩展消息)还提供了时间戳，表示事件生成的时间。除了有效负载和元数据外，QueryMessage还携带请求组件所期望的响应类型的描述。

2 Message Correlation 消息相关性

在消息传递系统中，将消息分组或关联在一起是很常见的。在Axon框架中，命令消息可能会产生一个或多个事件消息，而查询消息可能会产生一个或多个QueryResponse消息。通常，相关性是使用特定的消息属性(所谓的相关性标识符)来实现的。

**相关数据提供商**

Axon框架中的消息使用元数据属性来传输关于消息的元信息。元数据对象类型为Map<String，对象>，并随消息传递。为了填充工作单元中生成的新消息的元数据，可以使用所谓的CorrelationDataProvider。它是负责基于此CorrelationDataProvider填充新消息元数据的工作单元。Axon框架目前提供了这个功能接口的两个实现:

**MessageOriginProvider**

默认情况下，将MessageOriginProvider注册为要使用的关联数据提供程序。它负责从一个消息传递到另一个消息的两个值，即correlationId和traceId。消息的correlationId总是引用消息的标识符(即父消息)。另一方面，traceId引用了启动消息链的消息标识符(即根消息)。当创建新消息时，父消息中没有这两个字段时，将使用消息标识符

correlationId的命令消息标识符。

命令消息中的traceId是否存在于元数据中，或者是该消息的traceId标识符。

**SimpleCorrelationDataProvider**

SimpleCorrelationDataProvider被配置为无条件地将指定键的值从一条消息复制到另一条消息的元数据。为此，必须使用应复制的键列表调用SimpleCorrelationDataProvider的构造函数。下面是一个如何配置它来复制myId和myId2值的示例。

public class Configuration {

public CorrelationDataProvider customCorrelationDataProvider() {

return new SimpleCorrelationDataProvider("myId", "myId2");

}

}

MultiCorrelationDataProvider

MultiCorrelationDataProvider能够组合多个相关数据提供程序的效果。为此，必须使用提供程序列表调用MultiCorrelationDataProvider的构造函数，如下面的示例所示:

public class Configuration {

public CorrelationDataProvider customCorrelationDataProviders() {

return new MultiCorrelationDataProvider<CommandMessage<?>>(

Arrays.asList(

new SimpleCorrelationDataProvider("someKey"),

new MessageOriginProvider()

)

);

}

}

**实现自定义关联数据提供程序**

如果预定义的提供程序不能满足您的要求，您可以始终实现自己的CorrelationDataProvider。类必须实现CorrelationDataProvider接口，如下面的示例所示:

public class AuthCorrelationDataProvider implements CorrelationDataProvider {

private final Function<String, String> usernameProvider;

public AuthCorrelationDataProvider(Function<String, String> userProvider) {

this.usernameProvider = userProvider; }

@Override

public Map<String, ?> correlationDataFor(Message<?> message) {

Map<String, Object> correlationData = new HashMap<>();

if (message instanceof CommandMessage<?>) {

if (message.getMetaData().containsKey("authorization")) {

String token = (String) message.getMetaData().get("authorization");

correlationData.put("username", usernameProvider.apply(token));

}

}

return correlationData;

}

}

**Configuration**

当默认的MessageOriginProvider不足以满足您的用例时，(自定义)相关数据提供程序需要在您的应用程序中注册。如果您正在使用Axon配置API，请确保调用configuration# configureCorrelationDataProviders方法来注册相关的数据提供程序。如果您依赖于Spring Boot自动配置，只需提供一个暴露Spring Bean的工厂方法，如果需要多个提供程序，则提供多个。以下代码片段展示了一些可能的注册方法:

public class Configuration {

public void configuring() {

Configurer configurer =

DefaultConfigurer.defaultConfiguration()

.configureCorrelationDataProviders(config -> Arrays.asList(

new SimpleCorrelationDataProvider("someKey"),

new MessageOriginProvider()

));

}

}

@Configuration

public class CorrelationDataProviderConfiguration {

// Configuring a single CorrelationDataProvider will automatically override the default MessageOriginProvider

@Bean

public CorrelationDataProvider someKeyCorrelationProvider() {

return new SimpleCorrelationDataProvider("someKey");

}

@Bean

public CorrelationDataProvider messageOriginProvider() {

return new MessageOriginProvider();

}

}

3 Message Intercepting 消息拦截

有两种不同类型的拦截器:分派拦截器和处理程序拦截器。在将消息分派给消息处理程序之前调用分派拦截器。此时，甚至可能不知道该消息是否存在处理程序。在调用消息处理程序之前调用处理程序拦截器。

**命令拦截器**

使用命令总线的优点之一是能够根据所有传入的命令执行操作。示例是日志记录或身份验证，无论命令类型如何，您都可能希望这样做。这是使用拦截器完成的。

**指挥调度拦截器**

在命令总线上分派命令时，将调用消息分派拦截器。它们能够通过添加元数据来更改命令消息。它们还可以通过抛出异常来阻止命令。这些拦截器总是在分派命令的线程上被调用。

让我们创建一个MessageDispatchInterceptor，它记录在CommandBus上分派的每个命令消息。

public class MyCommandDispatchInterceptor

implements MessageDispatchInterceptor<CommandMessage<?>> {

private static final Logger LOGGER =

LoggerFactory.getLogger(MyCommandDispatchInterceptor.class);

@Override

public BiFunction<Integer, CommandMessage<?>, CommandMessage<?>> handle(List<? extends CommandMessage<?>> messages) {

return (index, command) -> {

LOGGER.info("Dispatching a command {}.", command);

return command;

};

}

}

我们可以用CommandBus注册这个分派拦截器，方法如下:

public class CommandBusConfiguration {

public CommandBus configureCommandBus() {

CommandBus commandBus = SimpleCommandBus.builder().build();

commandBus.registerDispatchInterceptor(new MyCommandDispatchInterceptor());

return commandBus;

}

}

**结构验证**

如果命令不能以正确的格式包含所有必需的信息，那么处理命令就没有意义。事实上，缺乏信息的命令应该尽早被阻塞，最好在事务启动之前就阻塞。因此，拦截器应该检查所有传入的命令，以确定这些信息的可用性。这被称为结构性验证。

Axon框架支持基于JSR 303 Bean的验证。这允许您用@NotEmpty和@Pattern这样的注释注释命令的字段。您需要在类路径中包含JSR 303实现(如Hibernate-Validator)。然后，在您的命令总线上配置BeanValidationInterceptor，它将自动查找并配置您的验证器实现。虽然它使用了合理的默认值，但您可以根据具体需要对它进行微调。

**拦截器命令提示**

您希望在无效命令上花费尽可能少的资源。因此，这个拦截器通常被放置在拦截器链的最前端。在某些情况下，可能需要先放置LoggingInterceptor或AuditingInterceptor，然后紧跟着验证拦截器。

BeanValidationInterceptor还实现了MessageHandlerInterceptor，允许您将其配置为一个处理程序拦截器。

**命令处理程序拦截器**

消息处理程序拦截器可以在命令处理之前和之后采取行动。拦截器甚至可以完全阻止命令处理，例如出于安全原因。

拦截器必须实现MessageHandlerInterceptor接口。这个接口声明了一个方法handle，它带两个参数:当前的UnitOfWork和一个InterceptorChain。InterceptorChain用于继续调度过程。UnitOfWork为您提供了(1)正在处理的消息，(2)提供了在(命令)消息处理之前、期间或之后绑定逻辑的可能性(有关阶段的更多信息，请参阅工作单元)。

与分派拦截器不同，处理程序拦截器是在命令处理程序的上下文中调用的。例如，这意味着它们可以将基于正在处理的消息的相关数据附加到工作单元。然后，该关联数据将被附加到在该工作单元的上下文中创建的消息。

处理程序拦截器通常也用于管理围绕命令处理的事务。为此，注册一个TransactionManagingInterceptor，它又配置了一个TransactionManager来启动和提交(或回滚)实际的事务。

让我们创建一个消息处理程序拦截器，它将只允许处理将axonUser作为元数据中userId字段的值的命令。如果元数据中没有该用户id，则会抛出一个异常，这将阻止命令被处理。此外，如果userId的值与axonUser不匹配，我们也不会继续执行链上的操作。

public class MyCommandHandlerInterceptor

implements MessageHandlerInterceptor<CommandMessage<?>> {

@Override

public Object handle(UnitOfWork<? extends CommandMessage<?>> unitOfWork, InterceptorChain interceptorChain) throws Exception {

CommandMessage<?> command = unitOfWork.getMessage();

String userId = Optional.ofNullable(command.getMetaData().get("userId"))

.map(uId -> (String) uId)

.orElseThrow(IllegalCommandException::new);

if ("axonUser".equals(userId)) {

return interceptorChain.proceed();

}

return null;

}

}

我们可以像这样用命令总线注册处理器拦截器:

public class CommandBusConfiguration {

public CommandBus configureCommandBus() {

CommandBus commandBus = SimpleCommandBus.builder().build();

commandBus.registerHandlerInterceptor(new MyCommandHandlerInterceptor());

return commandBus;

}

}

@CommandHandlerInterceptor Annotation

框架能够在聚合/实体上添加一个处理程序拦截器，作为@CommandHandlerInterceptor带注释的方法。聚合上的方法与“常规”命令处理程序拦截器的区别在于，使用注释方法，您可以根据给定聚合的当前状态做出决策。带注释的命令处理拦截器的一些属性是:

1 注释可以放在聚合中的实体上。

2 可以在聚合根级别上拦截命令，而命令处理程序位于子实体中。

3 可以通过从带注释的命令处理程序拦截器触发异常来阻止命令执行。

4 可以将InterceptorChain定义为命令处理程序拦截器方法的参数，并使用它来控制命令执行。

5 通过使用@CommandHandlerInterceptor注释的commandnameppattern属性，我们可以拦截所有匹配所提供的正则表达式的命令。

6 事件可以从带注释的命令处理程序拦截器应用。

在下面的例子中，我们可以看到一个@CommandHandlerInterceptor注释方法，如果命令的状态字段不匹配聚合的状态字段，它会阻止命令执行:

public class GiftCard {

//..

private String state;

//..

@CommandHandlerInterceptor

public void intercept(RedeemCardCommand command, InterceptorChain interceptorChain) {

if (this.state.equals(command.getState())) {

interceptorChain.proceed();

}

}

}

注意，@CommandHandlerInterceptor本质上是这里描述的@MessageHandlerInterceptor的更具体的实现。

**事件的拦截器**

与命令消息类似，事件消息也可以在发布和处理之前被拦截，以对所有事件执行额外的操作。这可以归结为同样的两种消息拦截器:分派拦截器和处理程序拦截器。

**事件分派拦截器**

在发布事件时，将调用注册到事件总线的任何消息分派拦截器。它们能够通过添加元数据来更改事件消息。它们还可以为发布事件时提供全面的日志记录功能。这些拦截器总是在发布事件的线程上被调用。

让我们创建一个事件消息分派拦截器，它记录在EventBus上发布的每个事件消息。

public class EventLoggingDispatchInterceptor

implements MessageDispatchInterceptor<EventMessage<?>> {

private static final Logger logger =

LoggerFactory.getLogger(EventLoggingDispatchInterceptor.class);

@Override

public BiFunction<Integer, EventMessage<?>, EventMessage<?>> handle(

List<? extends EventMessage<?>> messages) {

return (index, event) -> {

logger.info("Publishing event: [{}].", event);

return event;

};

}

}

然后，我们可以通过以下操作将这个分派拦截器注册到EventBus上:

public class EventBusConfiguration {

public EventBus configureEventBus(EventStorageEngine eventStorageEngine) {

// note that an EventStore is a more specific implementation of an EventBus

EventBus eventBus = EmbeddedEventStore.builder()

.storageEngine(eventStorageEngine)

.build();

eventBus.registerDispatchInterceptor(new EventLoggingDispatchInterceptor());

return eventBus;

}

}

**事件处理程序拦截器**

消息处理程序拦截器可以在事件处理之前和之后采取行动。拦截器甚至可以完全阻塞事件处理，例如出于安全原因。

拦截器必须实现MessageHandlerInterceptor接口。这个接口声明了一个方法handle()，它有两个参数:当前的UnitOfWork和一个InterceptorChain。InterceptorChain用于继续调度过程。UnitOfWork为您提供了(1)正在处理的消息，(2)提供了在(事件)消息处理之前、期间或之后绑定逻辑的可能性(有关阶段的更多信息，请参阅工作单元)。

与分派拦截器不同，处理程序拦截器是在事件处理程序的上下文中调用的。例如，这意味着它们可以将基于正在处理的消息的相关数据附加到工作单元。然后，该关联数据将被附加到在该工作单元的上下文中创建的事件消息。

让我们创建一个消息处理程序拦截器，它只允许处理将axonUser作为元数据中userId字段的值的事件。如果该用户id不在元数据中，将抛出一个异常，这将阻止事件被处理。而且，如果userId的值与axonUser不匹配，我们也不会继续执行链上的操作。验证事件消息(如本例中所示)是MessageHandlerInterceptor的一个常规用例。

public class MyEventHandlerInterceptor

implements MessageHandlerInterceptor<EventMessage<?>> {

@Override

public Object handle(UnitOfWork<? extends EventMessage<?>> unitOfWork,

InterceptorChain interceptorChain) throws Exception {

EventMessage<?> event = unitOfWork.getMessage();

String userId = Optional.ofNullable(event.getMetaData().get("userId"))

.map(uId -> (String) uId)

.orElseThrow(IllegalEventException::new);

if ("axonUser".equals(userId)) {

return interceptorChain.proceed();

}

return null;

}

}

我们可以像这样用EventProcessor注册处理器拦截器:

public class EventProcessorConfiguration {

public void configureEventProcessing(Configurer configurer) {

configurer.eventProcessing()

.registerTrackingEventProcessor("my-tracking-processor")

.registerHandlerInterceptor("my-tracking-processor",

configuration -> new MyEventHandlerInterceptor());

}

}

**拦截器注册**

不同于CommandBus和QueryBus，它们都可以有处理程序拦截器和分派拦截器，EventBus只能注册分派拦截器。这是因为EventBus的唯一目的是事件发布/分派，因此它们是事件分派拦截器注册的地方。事件处理器负责处理事件消息，因此它们是事件处理程序拦截器注册的地方。

**查询拦截器**

使用查询总线的优点之一是能够根据所有传入的查询执行操作。例如日志记录或身份验证，无论查询类型是什么，您都可能希望这样做。这是使用拦截器完成的。

**查询调度拦截器**

当在查询总线上分派查询或在查询更新发射器上分派对查询消息的订阅更新时，将调用消息分派拦截器。它们能够通过添加元数据来更改消息。它们还可以通过抛出异常来阻止处理程序的执行。这些拦截器总是在分派消息的线程上被调用。

**结构验证**

如果查询没有以正确的格式包含所有必需的信息，那么处理查询就没有意义。事实上，缺少信息的查询应该尽早被阻塞。因此，拦截器应该检查所有传入的查询，以确定这些信息的可用性。这被称为结构性验证。

Axon框架支持基于JSR 303 Bean的验证。这允许您用@NotEmpty和@Pattern这样的注释注释查询上的字段。您需要在类路径中包含JSR 303实现(如Hibernate-Validator)。然后，在您的查询总线上配置BeanValidationInterceptor，它将自动查找并配置您的验证器实现。虽然它使用了合理的默认值，但您可以根据具体需要对它进行微调。

**拦截器命令提示**

您希望在无效查询上花费尽可能少的资源。因此，这个拦截器通常被放置在拦截器链的最前端。在某些情况下，可能需要先放置日志或审计拦截器，然后紧跟着验证拦截器。

BeanValidationInterceptor还实现了MessageHandlerInterceptor，允许您将其配置为一个处理程序拦截器。

**查询处理程序拦截器**

消息处理程序拦截器可以在查询处理之前和之后采取行动。拦截器甚至可以完全阻塞查询处理，例如出于安全原因。

拦截器必须实现MessageHandlerInterceptor接口。这个接口声明了一个方法handle，它带两个参数:当前的UnitOfWork和一个InterceptorChain。InterceptorChain用于继续调度过程。UnitOfWork为您提供了(1)正在处理的消息，(2)提供了在(查询)消息处理之前、期间或之后绑定逻辑的可能性(有关阶段的更多信息，请参阅工作单元)。

与分派拦截器不同，处理程序拦截器是在查询处理程序的上下文中调用的。例如，这意味着它们可以将基于正在处理的消息的相关数据附加到工作单元。然后，该关联数据将被附加到在该工作单元的上下文中创建的消息。

**@MessageHandlerInterceptor**

除了在处理消息(例如，命令、查询或事件)的组件上定义总体MessageHandlerInterceptor实例外，还可以为包含处理程序的特定组件定义处理程序拦截器。这可以通过添加处理消息的方法，并结合@MessageHandlerInterceptor注释来实现。添加这样的方法允许您更细粒度地控制哪些消息处理组件应该作出反应，以及这些组件应该如何作出反应。

当添加@MessageHandlerInterceptor时，有几个句柄会给你，比如:

1 MessageHandlerInterceptor实例与InterceptorChain一起工作，以决定何时继续处理链中的其他拦截器。InterceptorChain是一个可选参数，可以添加到intercepting方法中，为您提供相同的控件。如果没有这个参数，一旦该方法退出，框架将调用InterceptorChain#proceed。

2 您可以定义拦截器应该处理的消息类型。默认情况下，它对任何消息实现进行响应。如果需要特定于EventMessage的拦截器，注释上的messageType参数应该设置为EventMessage.class。

3 为了更细粒度地控制哪些消息应该响应拦截器，可以指定要处理的消息中包含的payloadType。

下面的代码片段展示了一些可能使用@MessageHandlerInterceptor注释的方法:

public class CardSummaryProjection {

/\*

\* Some @EventHandler and @QueryHandler annotated methods

\*/

@MessageHandlerInterceptor

public void intercept(Message<?> message) {

// Add your intercepting logic here based on the

}

}

public class CardSummaryProjection {

/\*

\* Some @EventHandler and @QueryHandler annotated methods

\*/

@MessageHandlerInterceptor(messageType = EventMessage.class)

public void intercept(EventMessage<?> eventMessage) {

// Add your intercepting logic here based on the

}

}

public class CardSummaryProjection {

/\*

\* Some @EventHandler and @QueryHandler annotated methods

\*/

@MessageHandlerInterceptor(messageType = QueryMessage.class)

public void intercept(QueryMessage<?, ?> queryMessage,

InterceptorChain interceptorChain) throws Exception {

// Add your intercepting logic before

interceptorChain.proceed();

// or after the InterceptorChain#proceed invocation

}

}

除了消息、有效负载和InterceptorChain之外，一个带@MessageHandlerInterceptor注释的方法也可以解析其他参数。框架可以在这样的函数上解析哪些参数，这取决于拦截器正在处理的消息类型。对于正在处理的消息，哪些参数是可解析的，要了解更多细节，请查看此页面。

**@ExceptionHandler**

@MessageHandlerInterceptor也允许拦截函数的更特定版本。即，一个@ExceptionHandler注释的方法。带有@ExceptionHandler注释的函数将被视为一个处理器拦截器，它只会在出现异常结果时被调用。例如，为此使用带注释的函数允许您抛出一个更特定于域的异常作为抛出数据库/服务异常的结果。你可以引入一个@ExceptionHandler来对所有异常进行响应，或者像这里所示的那样指定带注释的处理程序对哪些异常进行响应:

public class CardSummaryProjection {

/\*

\* Some @EventHandler and @QueryHandler annotated methods

\*/

@ExceptionHandler

public void handle(Exception exception) {

// How you prefer to react to this generic exception,

// for example by throwing a domain specific exception.

}

@ExceptionHandler(resultType = IllegalArgumentException.class)

public void handle(IllegalArgumentException exception) {

// How you prefer to react to the IllegalArgumentException,

// for example by throwing a domain specific exception.

}

}

4 支持带注释的处理程序的参数

本章为带注释的消息处理函数提供了所有可能参数的详尽列表。任何消息处理函数的参数都是通过称为ParameterResolver的Axon框架内部机制来解析的。

如果需要将自定义(或者还没有)支持的参数注入到带注释的处理程序中，则可以扩展ParameterResolvers的集合。关于配置自定义参数解析器的更多细节，我们建议阅读本节。

支持命令处理程序的参数

默认情况下，@CommandHandler注释的方法允许以下参数类型:

1 第一个参数是命令消息的有效负载。

如果@CommandHandler注释显式定义了处理程序可以处理的命令的名称，那么它也可以是Message或CommandMessage类型。

默认情况下，命令名是命令及其负载的完全限定类名。

2 用@MetaDataValue注释的参数将解析为带有注释中指示的键的元数据值。

如果required为false(默认)，则在元数据值不存在时传递null。

如果required为true，则解析器将不匹配并阻止该方法被调用

3 MetaData类型的参数将注入命令消息的全部元数据。

4 类型为UnitOfWork的参数获得注入的当前工作单元。这允许命令处理程序注册要在工作单元的特定阶段执行的操作，或者获得对向其注册的资源的访问权。

5 Message或CommandMessage类型的参数将获得完整的消息，其中包括有效负载和元数据。如果一个方法需要几个元数据字段或包装消息的其他属性，那么这是很有用的。

6 带@MessageIdentifier注释的String类型参数将解析正在处理的CommandMessage的标识符

7 类型为ConflictResolver的参数将解析配置的ConflictResolver实例。有关此主题的详细信息，请参阅冲突解决方案一节。

8 InterceptorChain类型的参数将解析命令消息的MessageHandlerInterceptors链。该特性应该与@CommandHandlerInterceptor带注释的方法一起使用。关于这方面的更多细节，建议阅读本节。

9 如果应用程序在Spring环境中运行，则可以解析任何Spring Bean。注意，@Qualifier注释可以与此结合使用，以进一步指定应该解析哪个Bean。

支持查询处理程序的参数

默认情况下，@QueryHandler注释的方法允许以下参数类型:

1 第一个参数是查询消息的有效负载。

如果@QueryHandler注释显式定义了处理程序可以处理的查询的名称，那么它也可以是Message或QueryMessage类型。

默认情况下，查询名是查询及其负载的完全限定类名。

2 用@MetaDataValue注释的参数将解析为带有注释中指示的键的元数据值。

如果required为false(默认)，则在元数据值不存在时传递null。

如果required为true，则解析器将不匹配，并在元数据值不存在时阻止调用该方法。

3 MetaData类型的参数将注入QueryMessage的整个元数据。

4 类型为UnitOfWork的参数获得注入的当前工作单元。这允许查询处理程序注册要在工作单元的特定阶段执行的操作，或获得对向其注册的资源的访问权。

5 Message类型的参数或QueryMessage将获得完整的消息，其中包括有效负载和元数据。如果一个方法需要几个元数据字段，或者包装消息的其他属性，这是很有用的。

6 带@MessageIdentifier注释的String类型参数将解析正在处理的QueryMessage的标识符

7 如果应用程序在Spring环境中运行，则可以解析任何Spring Bean。注意，@Qualifier注释可以与此结合使用，以进一步指定应该解析哪个Bean。

5 异常处理

在开发软件时，异常处理是一个众所周知的概念。在分布式应用程序环境中处理异常比我们通常所习惯的更具挑战性。特别是当涉及到处理命令或查询、意图具有返回值的消息失败时，我们应该注意如何抛出异常。

异常处理执行连接点

HandlerExecutionException标记了一个来自消息处理成员的异常。由于事件消息是单向的，所以处理事件不包括任何返回值。因此，HandlerExecutionException应该只作为处理命令的异常结果返回。Axon为失败的命令和查询处理提供了该异常的更具体实现，分别是CommandExecutionException和QueryExecutionException。

在分布式应用程序环境中，专用的处理程序执行异常的用途变得更加清晰，例如，在分布式应用程序环境中，有一个专用的应用程序处理命令，另一个应用程序负责查询端。由于应用程序隔离，丧失了两个应用程序可以访问相同类的确定性，因此对任何异常类都适用。为了支持和鼓励这种解耦，Axon将生成命令或查询处理导致的任何异常。

要维护对依赖于分布式场景中抛出的异常类型的条件逻辑的支持，可以在HandlerExecutionException中提供详细信息。因此，当命令/查询处理失败时，建议抛出一个CommandExecutionException / queryexecutionexception，并提供所需的详细信息。这种行为可以通过实现为你执行异常包装的拦截器来支持。

Ex@ExceptionHandler annotated methods 带注释的方法

Axon框架允许使用带@ExceptionHandler注释的方法来提供对如何响应异常的更细粒度控制。更具体地说，这是一种消息处理程序拦截器，专门用于响应异常结果。注意，这样的@ExceptionHandler将只处理同一类中的消息处理函数抛出的异常。有关如何使用该注释的更多细节，请参阅本节。

6 工作单元

工作单位是轴突框架中的一个重要概念。尽管，在大多数情况下，您不太可能直接与它交互。消息的处理被看作是一个单独的单元。工作单元的目的是协调在处理消息(命令、事件或查询)期间执行的操作。组件可以注册要在UnitOfWork的每个阶段执行的操作，比如onPrepareCommit或onCleanup。

您不太可能需要直接访问UnitOfWork。它主要由Axon框架提供的构建块使用。如果您确实需要访问它，无论出于何种原因，有几种方法可以获得它。处理程序通过handle方法中的参数接收工作单元。如果你使用了注释支持，你可以给你的注释方法添加一个类型为UnitOfWork的参数。在其他位置，您可以通过调用CurrentUnitOfWork.get()来检索绑定到当前线程的工作单元。注意，如果没有将UnitOfWork绑定到c中，这个方法将抛出异常

要求访问当前工作单元的一个原因是附加需要在消息处理过程中多次重用的资源，或者创建的资源需要在工作单元完成时清理。在这种情况下，unitOfWork.getOrComputeResource()和生命周期回调方法，如onRollback()、afterCommit()和onCleanup()允许您注册资源并声明在处理这个工作单元期间要采取的操作。

请注意

请注意，工作单元仅仅是变更的缓冲，而不是事务的替换。尽管所有阶段性更改仅在提交工作单元时提交，但它的提交不是原子的。这意味着，当提交失败时，一些更改可能已经持久化，而其他更改则没有。最佳实践规定一个命令永远不应该包含一个以上的操作。如果您坚持这种做法，工作单元将包含单个操作，这样就可以安全地按原样使用。如果在工作单元中有更多的操作，则可以考虑将事务附加到

处理程序可能会在处理消息时抛出异常。默认情况下，未检查的异常将导致UnitOfWork回滚所有更改。因此，预定的副作用被取消了。

Axon框架提供了一些开箱即用的回滚策略:

RollbackConfigurationType。永不献身的人，永远是一心一意的人

RollbackConfigurationType。ANY\_THROWABLE—在发生异常时将总是回滚

RollbackConfigurationType。UNCHECKED\_EXCEPTIONS——将回滚错误和运行时异常

RollbackConfigurationType。RUNTIME\_EXCEPTION—将回滚运行时异常(但不会回滚错误)

当使用框架组件处理消息时，工作单元的生命周期将为您自动管理。如果您选择不使用这些组件，而是自己实现处理，那么您将需要以编程方式启动并提交(或回滚)一个工作单元。

在大多数情况下，DefaultUnitOfWork将为您提供所需的功能。它希望在单个线程中进行处理。要在工作单元的上下语境中执行任务，只需在一个新的DefaultUnitOfWork上调用UnitOfWork.execute(Runnable)或UnitOfWork.executeWithResult(Callable)。工作单元将在任务完成时启动并提交，或者在任务失败时回滚。如果需要更多的控制，您还可以选择手动启动、提交或回滚工作单元。

典型用法如下:

UnitOfWork uow = DefaultUnitOfWork.startAndGet(message);

// then, either use the autocommit approach:

uow.executeWithResult(() -> ... logic here);

// or manually commit or rollback:

try {

// business logic comes here

uow.commit();

} catch (Exception e) {

uow.rollback(e);

// maybe rethrow...

}

请注意

工作单元以消息为中心。它总是以要处理的消息开始。作为工作单元执行的结果(executeWithResult(…))，将返回ResultMessage，而实际的执行结果将是ResultMessage的有效负载。如果在消息处理过程中出现了问题，我们将得到一个异常的ResultMessage——isexception()将返回true，而exceptionResult()将得到实际的可抛出对象，指示出错的地方。

A 个工作单位知道几个阶段。每次它进展到另一个阶段时，都会通知侦听器。

1 活动阶段——这是工作单元开始的地方。工作单元通常在这个阶段注册到当前线程(通过CurrentUnitOfWork.set(UnitOfWork))。随后，该消息通常在此阶段由消息处理程序处理。

提交阶段——在消息处理完成后但在提交工作单元之前，调用onPrepareCommit监听器。如果一个工作单元绑定到一个事务，就会调用onCommit监听器来提交任何支持事务。当提交成功时，将调用afterCommit监听器。如果提交或之前的任何步骤失败，则调用onRollback监听器。消息处理程序结果包含在工作单元的ExecutionResult(如果可用)中。

清理阶段——在此阶段，该工作单元所持有的任何资源(如锁)都将被释放。如果多个工作单元是嵌套的，则清理阶段将推迟到外部工作单元准备好清理为止

消息处理过程可以被认为是一个原子过程;要么完全处理，要么根本不处理。Axon框架使用工作单元来跟踪消息处理程序执行的操作。处理程序完成后，Axon将尝试提交注册到工作单元的操作。

可以将事务绑定到工作单元。许多组件，如CommandBus和QueryBus实现以及所有异步处理事件处理器，都允许您配置TransactionManager。然后，该事务管理器将用于创建要绑定到用于管理消息流程的工作单元的事务。

当应用程序组件在消息处理的不同阶段需要资源时，例如数据库连接或EntityManager，这些资源可以附加到UnitOfWork。getresources()方法允许您访问附加到当前工作单元的资源。工作单元上有几个直接可用的助手方法，使使用资源变得更容易。

当嵌套的工作单元需要能够访问资源时，建议在根工作单元上注册它，可以使用unitOfWork.root()访问根工作单元。如果一个工作单元是根，它将简单地返回自身。

命令

与Axon应用程序相关联的第一类“消息”是命令消息(或简单的命令)。参考指南的这一部分将详细介绍Axon框架提供的帮助促进命令消息开发的功能。

下面给出了各个小节的摘要。

建模 ： Axon框架提供了组件来处理聚合建模问题

命令调度程序/命令处理程序： 使用Axon框架进行命令消息的功能开发

实现 ： 使用Axon框架进行命令消息的非功能性开发

配置 ： 使用Axon框架的命令消息的配置问题

本指南由一个视频教程补充，该视频教程对本节中解释的概念进行了实际演示。该教程可在->轴突编码教程-命令消息

Aggregate

本章将介绍如何实现“聚合”的基础知识。有关什么是聚合的详细信息，请阅读DDD和CQRS概念页面。

Basic Aggregate Structure

聚合是一个常规对象，它包含改变状态的状态和方法。当创建聚合对象时，您实际上是在创建“聚合根”，通常携带整个聚合的名称。在本文中，我们将使用“GiftCard”域，将GiftCard作为集合(根)。默认情况下，Axon会将聚合配置为“事件源”聚合(如这里所述)。今后，我们的基本GiftCard聚合结构将关注事件来源方法:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.eventsourcing.EventSourcingHandler;

import org.axonframework.modelling.command.AggregateIdentifier;

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

public class GiftCard {

@AggregateIdentifier // 1.

private String id;

@CommandHandler // 2.

public GiftCard(IssueCardCommand cmd) {

// 3.

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

}

@EventSourcingHandler // 4.

public void on(CardIssuedEvent evt) {

id = evt.getCardId();

}

// 5.

protected GiftCard() {

}

// omitted command handlers and event sourcing handlers

}

从给定的代码片段中，有几个值得注意的概念，用编号的Java注释标记出来，这些注释指的是下面的项目符号:

1 @AggregateIdentifier是GiftCard聚合的外部引用点。这个字段是一个硬性要求，因为没有它，Axon将不知道给定命令的目标聚合是哪个。注意，这个注释可以放在字段和方法上。

2 一个带@CommandHandler注释的构造函数，或者用不同的方式放置“命令处理构造函数”。这个注释告诉框架给定的构造函数能够处理IssueCardCommand。

带有@CommandHandler注释的函数是放置决策/业务逻辑的地方。

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

4 使用@EventSourcingHandler是什么告诉框架，当聚合“来自它的事件”时，应该调用带注释的函数。

由于所有事件源处理程序的组合将形成聚合，因此所有状态更改都发生在这里。

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带注释的函数使用特定的规则进行解析。

这些规则对于@EventHandler注释的方法是相同的，并且是彻底的

5 无参数的构造函数，Axon需要它。

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聚合实例。

如果不能提供此构造函数，将在加载聚合时导致异常。

消息处理函数的修饰符

事件处理程序方法可以是私有的，只要JVM的安全设置允许Axon框架更改方法的可访问性。这允许您清晰地将聚合的公共API(公开生成事件的方法)与处理事件的内部逻辑分离开来。

大多数IDE都有一个选项，可以忽略带有特定注释的方法的“未使用的私有方法”警告。或者，你可以添加一个@SuppressWarnings("UnusedDeclaration")注释到方法中，以确保你不会发生意外

总生命周期操作Aggregate Lifecycle Operations

在聚合的生命周期中，有几个操作是需要执行的。为此，Axon中的AggregateLifecycle类提供了两个静态函数:

1 apply(对象)和apply(对象，元数据):AggregateLifecycle#apply将在EventBus上发布一个事件消息，这样就知道该消息来自于执行操作的聚合。

可以只提供事件对象，也可以同时提供事件和某些特定元数据。

2 creatennew (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。

3 isLive():检查以验证聚合是否处于'live'状态。

如果一个聚合体已经完成了历史事件的重演，以重现其状态，那么这个聚合体就被认为是“活的”。

如果聚合因此处于事件来源的过程中，aggregatelifcycle . 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。

4 markDeleted():将调用该函数的聚合实例标记为'deleted'。

如果域指定了一个给定的聚合可以被删除/删除/关闭，在此之后它将不再被允许处理任何命令，这是非常有用的。

这个函数应该从@EventSourcingHandler带注释的函数中调用，以确保标记已删除是聚合状态的一部分。

Multi-Entity Aggregates

复杂的业务逻辑通常需要的不仅仅是只有聚合根的聚合所能提供的。在这种情况下，将复杂性分散到聚合体中的许多“实体”上是很重要的。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。

**实体之间的状态**

对于聚合不应该公开状态的规则的一种常见误解是，任何实体都不应该包含任何属性访问器方法。事实并非如此。事实上，如果聚合中的实体向同一聚合中的其他实体公开状态，那么聚合可能会获益良多。但是，建议不要在聚合外部公开状态。

在“GiftCard”域中，在本节中定义了GiftCard聚合根。让我们利用这个领域来引入实体:

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;

// omitted constructors, command and event sourcing handlers

}

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;

}

// omitted command handlers, event sourcing handlers and equals/hashCode

}

与聚合根一样，实体是简单对象，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/实体的字段必须用@AggregateMember注释。此注释告诉Axon，带注释的字段包含一个应检查的消息处理程序类。这个示例显示了Iterable实现上的注释，但它也可以放置在单个对象或映射上。在后一种情况下，映射的值应该包含实体，而键包含一个作为它们的引用的值。注意，这个注释可以放在字段和方法上。

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。用于查找消息应该路由到的实体的有效负载上的属性，默认为@EntityId注释字段的名称。例如，在注释字段transactionId时，命令必须定义一个具有相同名称的属性，这意味着transactionId或getTransactionId()方法必须存在。如果字段的名称和路由属性不同，则可以提供一个值expli

**定义实体类型**

集合或映射的字段声明都应该包含适当的泛型，以允许Axon标识集合或映射中包含的实体类型。如果不可能在声明中添加泛型(例如，因为你使用的自定义实现已经定义了泛型类型)，你必须通过在@AggregateMember注释中指定类型字段来指定实体类型:

>@AggregateMember(type = GiftCardTransaction.class).

**实体中的命令处理**

注释并不局限于聚合根。将所有命令处理程序放置在根中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。如果是这种情况，您可以将@CommandHandler注释放置在一个底层实体的方法上。为了让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被标记为@AggregateMember:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.modelling.command.AggregateIdentifier;

import org.axonframework.modelling.command.AggregateMember;

import org.axonframework.modelling.command.EntityId;

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

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。如果在该实体的传入命令到达时，字段值为null，则抛出异常。如果存在子实体的集合或映射，且没有找到与命令的路由键匹配的实体，Axon将抛出一个IllegalStateException，因为显然聚合在那个时间点无法处理命令。

命令处理程序注意事项

注意，每个命令在聚合中必须只有一个处理程序。这意味着您不能用@CommandHandler注释多个处理相同命令类型的实体(根实体或非根实体)。如果您需要有条件地将命令路由到一个实体，这些实体的父实体应该处理该命令，并根据应用的条件转发它。

字段的运行时类型不必完全是声明的类型。但是，对于@CommandHandler方法，只检查@AggregateMember注释字段的声明类型。

实体中的事件源处理程序

当使用事件源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，该聚合中的每个实体也需要使用事件。Axon为诸如此类的复杂聚合结构提供了开箱即用的事件来源支持。

当一个实体(包括聚合根)应用一个事件时，它首先由聚合根处理，然后通过每个@AggregateMember注释字段向下冒泡到所有包含它的子实体:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.modelling.command.AggregateIdentifier;

import org.axonframework.modelling.command.AggregateMember;

import org.axonframework.modelling.command.EntityId;

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

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

}

上面的代码片段中有两个细节值得一提，用编号的Java注释指出:

实体的创建发生在其父类的事件源处理程序中。

因此，在实体类上不可能像聚合根类那样有一个“命令处理构造函数”。

实体中的事件源处理程序执行验证，检查接收到的事件是否确实属于实体。

这是必要的，因为一个实体实例应用的事件也将由相同类型的任何其他实体实例处理。

第二点描述的情况是可定制的，通过改变@AggregateMember注释的eventForwardingMode:

import org.axonframework.modelling.command.AggregateIdentifier;

import org.axonframework.modelling.command.AggregateMember;

import org.axonframework.modelling.command.ForwardMatchingInstances;

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，事件消息将只在它包含一个字段/getter，该字段匹配实体上的@EntityId注释字段的名称时被转发。这种路由行为可以通过@EntityId注释上的routingKey字段进一步指定，与实体中的路由命令相镜像。其他可以使用的转发模式有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事件。

State Stored Aggregates

在Aggregate主页中，我们已经看到了如何创建由事件来源支持的聚合。换句话说，事件源聚合的存储方法是通过重放构成聚合上的更改的事件。

然而，聚合也可以按原样存储。在执行此操作时，用于保存和加载聚合的存储库是GenericJpaRepository。状态存储的聚合体的结构与事件来源的聚合体稍有不同:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.eventhandling.EventHandler;

import org.axonframework.modelling.command.AggregateIdentifier;

import org.axonframework.modelling.command.AggregateMember;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.OneToMany;

@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.

}

上面的示例显示了从“礼品卡服务”存储的聚合状态。代码片段中的编号注释指出了这里解释的Axon细节:

1 由于聚合存储在JPA存储库中，所以需要用@Entity对类进行注释。

2 聚合根必须声明一个包含聚合标识符的字段。该标识符必须在发布第一个事件时进行初始化。这个标识符字段必须由@AggregateIdentifier标注。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，所以@Id注释是一个硬性要求。

3 这个集合有几个“集合成员”。 因为聚合是按原样存储的，所以应该考虑实体的正确映射。

4 一个带@CommandHandler注释的构造函数，或者用不同的方式放置“命令处理构造函数”。 这个注释告诉框架给定的构造函数能够处理IssueCardCommand。

5 静态AggregateLifecycle#apply(Object…)可以用来发布事件消息。在调用此函数时，所提供的对象将作为EventMessages在其应用的聚合范围内发布。

6 命令处理方法首先决定传入命令在此时是否有效。

7 验证业务逻辑之后，可以调整聚合的状态

8 通过定义一个@EventHandler带注释的方法，聚合中的实体可以侦听聚合发布的事件。 当事件消息在由任何外部处理程序处理之前发布时，将调用这些方法。

9 无参数构造函数，JPA需要它。如果不能提供此构造函数，将在加载聚合时导致异常。

调整命令处理程序中的状态

与事件源聚合不同，状态存储聚合可以将决策制定逻辑与命令处理程序中的状态更改配对。遵循这种范式不会对状态存储聚合产生任何影响，因为没有驱动其状态的事件源处理程序。

从另一个聚合创建聚合

通常，通过发出一个创建命令来实例化一个新的聚合，该创建命令由@CommandHandler带注释的聚合构造函数处理。例如，这些命令可以由简单的REST端点或事件处理组件发布，作为对特定事件的响应。然而，有时域描述从另一个实体创建的某些实体。在这种情况下，从父聚合实例化聚合将更忠实于域。

**Aggregate-from-Aggregate用例**

从“父”聚合创建“子”聚合的最合适场景是，在父聚合的上下文中创建子聚合的决定。例如，如果父聚合包含驱动子创建决策的必要状态，则可以显示它本身。

**如何从另一个聚合创建一个聚合**

让我们假设我们有一个ParentAggregate，在处理某个命令时将决定创建一个子聚合。为了实现这一点，ParentAggregate应该是这样的:

import org.axonframework.commandhandling.CommandHandler;

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

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 command handlers

}

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

在这个场景中，ChildAggregate的实现将类似于以下格式:

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

}

注意，一个ChildAggregateCreatedEvent被显式地应用来通知子聚合已经创建，否则这个信息会被封装在ParentAggregate的SomeParentCommand命令处理程序中。

从事件源处理程序创建聚合?

应该在命令处理程序中而不是在事件源处理程序中创建新聚合。这样做的理由是，当父聚合来自其事件时，您不希望创建新的子聚合，因为这将不受欢迎地创建新的子聚合实例

然而，如果在事件来源处理程序中偶然调用creatennew方法，则会抛出UnsupportedOperationException作为临时解决方案。

Aggregate Polymorphism

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的子类型从超聚合中继承@CommandHandlers、@EventSourcingHandlers和@CommandHandlerInterceptors。基于@AggregateIdentifier，将加载正确的聚合类型并在其上执行命令。让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类型和ClosedLoopGiftCard、OpenLoopGiftCard和RechargeableGiftCard的子类型的多态聚合。如果在Card类中存在处理程序，那么这些处理程序也会在所有聚合中存在。

**当建模一个多态聚合层次结构时，记住这些约束是很重要的:**

1 不允许在抽象聚合上有一个带@CommandHandler注释的构造函数。这样做的基本原理是永远不能创建抽象聚合。

2 在同一层次结构中的不同聚合上拥有相同命令名的创建命令处理程序也被禁止，因为Axon无法派生要调用哪个创建命令处理程序。

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

**Registering aggregate subtypes**

多态聚合层次结构可以通过调用AggregateConfigurer#registerSubtype(Class)来通过AggregateConfigurer注册。请注意，没有注册为子类型的父聚合的子类型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扩展RechargeableGiftCard，则不会拾取它(除非显式地注册为子类型)。

AggregateConfigurer<GiftCard> configurer

= AggregateConfigurer.defaultConfiguration(GiftCard.class)

.withSubtype(OpenLoopGiftCard.class)

.withSubtype(RechargeableGiftCard.class);

**Polymorphic Aggregates in Spring**

如果您正在使用Spring, Axon将根据@Aggregate注释和类层次结构自动检测多态聚合。@Aggregate注释需要放在包含聚合标识符的共享父类上，以及作为该共享父类的潜在实例类型的每个子类上。

冲突解决

明确更改含义的主要优点之一是，您可以更精确地检测冲突的更改。通常，当两个用户(几乎)同时操作相同的数据时，就会发生这些冲突性的更改。假设有两个用户，都在查看数据的特定版本。他们都决定对数据进行更改。它们都将发送一个命令，比如“在这个聚合的版本X上，做那个”，其中X是预期的聚合版本。其中一个将把更改实际应用到预期的版本。其他用户不会。

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看不见的更改发生冲突，而不是简单地拒绝所有传入的命令。

要检测冲突，请将一个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。

Expected Aggregate Version

请注意，如果聚合是用预期的版本加载的，那么ConflictResolver只会包含任何潜在的冲突事件。在命令的字段上使用@TargetAggregateVersion来指示聚合的预期版本。

如果发现与谓词匹配的事件，则抛出异常(detectConflicts的第二个可选参数允许您定义要抛出的异常)。如果没有找到，则继续正常处理。

如果没有进行检测冲突的调用，并且存在潜在的冲突事件，那么@CommandHandler将失败。当提供了预期的版本，但是@CommandHandler方法的参数中没有可用的ConflictResolver时可能会出现这种情况。

Command Dispatchers

命令处理程序页提供了如何在应用程序中处理命令消息的背景信息。调度流程是此类命令消息的起点。Axon提供了两个接口，你可以使用它们将命令发送到你的命令处理程序，它们是:

1 The Command Bus, and

2 The Command Gateway

本页将显示如何以及何时使用命令网关和总线。如何在命令网关和总线实现上配置和细节在这里被讨论

**The Command Bus**

“命令总线”是将命令分派给各自的命令处理程序的机制。因此，正是基础架构组件知道哪个组件可以处理哪个命令。

每个命令总是被发送到一个命令处理程序。如果被分派的命令没有可用的命令处理程序，则抛出NoHandlerForCommandException异常。

CommandBus提供了两个方法来将命令分派到各自的处理程序，分别是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();

}

}

);

}

// omitted class, constructor and result usage

面描述的CommandDispatcher示例了调度命令的几个重要方面和功能:

1 CommandBus接口提供分派命令消息的功能。

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，只要该对象实现了一个合理的toString()函数。

3 GenericCommandMessage#asCommandMessage(对象)方法用于创建CommandMessage。

为了能够在CommandBus上分派一个命令，你需要在CommandMessage中包装你自己的命令对象(例如'命令消息有效负载')。

CommandMessage还允许向命令消息添加元数据。

4 CommandBus#dispatch(CommandMessage)函数将在总线上分发提供的CommandMessage，以便交付给命令处理程序。

如果应用程序对命令的结果不直接感兴趣，可以使用此方法。

5 如果命令处理的结果与您的应用程序相关，则可以提供第二个可选参数CommandCallback。

CommandCallback允许在命令处理完成时通知调度组件。

6 命令回调有一个函数onResult(CommandMessage, commandsultmessage)，该函数在命令处理完成时被调用。

第一个参数是被分派的命令，第二个参数是被分派命令的执行结果。

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

因此,commandBus。调度(commandMessage， (cmdMsg, commandsultmessage) ->{/\*…\*/})也是可能的。

7 commandsultmessage提供了验证命令执行是否异常或成功的API。

如果commandsultmessage # isexception返回true，您可以假定commandsultmessage #exceptionResult将返回一个包含实际异常的可抛出实例。

否则，commandsultmessage #getPayload方法可能为您提供一个实际的结果或null，在这里进一步指定。

**命令回调的考虑**

在使用dispatch(CommandMessage、CommandCallback)的情况下，调用组件可能不会假定回调函数在分派命令的同一线程中被调用。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(在java中定义的)的组合。并发包)和Axon的CommandCallback。或者，考虑使用CommandGateway。

**The Command Gateway**

“命令网关”是一种方便的调度命令的方法。它通过在CommandBus上调度命令时为您抽象某些方面来实现这一点。它使用下面的CommandBus来执行消息的实际分派。

虽然您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引入了几个概念，用编号的注释标记:

1 CommandGateway接口提供分派命令消息的功能。 它通过内部利用CommandBus接口分派消息来做到这一点。

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，只要该对象实现了一个合理的toString()函数。

3 send(Object)函数需要一个参数，即command对象。这是一种分派命令的异步方法。因此，send方法的响应是一个CompletableFuture。这允许在命令结果返回后将后续操作链接起来。

**使用send(对象)时的回调**

CommandGateway#send(Object)方法使用底层的FutureCallback来解除命令调度线程与命令处理线程之间的阻塞。

通过使用sendAndWait方法，也可以实现发送消息的同步方法:

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

1 CommandGateway#sendAndWait(对象)函数接受一个参数，即您的命令对象。它将无限期等待，直到命令调度和处理过程已经解决。这个方法返回的结果可能是成功的，也可能是异常的，下面将解释。

2 如果不希望无限期地等待，可以在命令对象旁边提供与“时间单位”配对的“超时”。这样做将确保命令调度线程等待的时间不会超过指定的时间。如果使用这种方法时命令调度/处理中断或超时，命令结果将为null。在所有其他场景中，结果遵循所引用的方法。

**Command Dispatching Results**

一般来说，调度命令有两种可能的结果:

1 Command handled successfully, and

2 command handled exceptionally

结果在某种程度上取决于调度流程，但更多地取决于命令处理程序的实现。因此，如果带注释的@CommandHandler函数由于某些业务逻辑而抛出异常，则该异常将是分派命令的结果。

命令处理的成功解决故意不应该提供任何返回对象。因此，如果CommandBus/CommandGateway提供了一个响应(直接或通过commandsultmessage)，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，但应该少用。该命令的目的绝不应该是检索值，因为这将表明应该将消息设计为查询消息。例外情况是聚合根的标识符，或者聚合根实例化的实体的标识符。框架在带@CommandHandler注释的聚合构造函数上内置了一个这样的异常。如果“命令处理构造函数”已成功执行，而不是聚合本身

轴突编码教程#5:-连接UI

Command Handlers

**Aggregate Command Handlers**

尽管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该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注释应该处理该命令的方法。带注释的@CommandHandler方法将成为命令消息的命令处理程序，其中命令名称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(RedeemCardCommand cmd)的方法签名将

也可以使用不同的命令名分派命令消息。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，命令对象中携带聚合标识符的属性必须用@TargetAggregateIdentifier标注。注释可以放在命令对象的字段或访问器方法(例如getter)上。

以GiftCard聚合为例，我们可以识别两个命令句柄

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.modelling.command.AggregateIdentifier;

​

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

​

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()));

}

// omitted event sourcing handlers

}

命令对象，IssueCardCommand和RedeemCardCommand，它们的GiftCard句柄有以下格式:

import org.axonframework.modelling.command.TargetAggregateIdentifier;

​

public class IssueCardCommand {

​

@TargetAggregateIdentifier

private final String cardId;

private final Integer amount;

​

public IssueCardCommand(String cardId, Integer amount) {

this.cardId = cardId;

this.amount = amount;

}

// omitted getters, equals/hashCode, toString functions

}

​

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;

}

// omitted getters, equals/hashCode, toString functions

}

这两个命令中的cardId是对GiftCard实例的引用，因此使用@TargetAggregateIdentifier注释。创建聚合实例的命令不需要标识目标聚合标识符，因为目前还不存在聚合。尽管如此，为了保持一致性，还是建议在它们上标注聚合标识符。

如果您更喜欢使用另一种机制来路由命令，则可以通过提供一个定制的CommandTargetResolver来覆盖该行为。这个类应该返回聚合标识符和预期的版本

**聚合创建命令处理程序**

当@CommandHandler注释被放置在一个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。这些命令不需要以特定的聚合实例为目标。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定制的CommandTargetResolver。

但是，无论命令类型如何，只要您通过Axon服务器等分发应用程序，都强烈建议在给定的消息上指定一个路由密钥。@TargetAggregateIdentifier也是这样，但是如果没有值得注释的字段，则应该添加@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，这在命令调度部分进一步指定。

**业务逻辑和状态更改**

在聚合中，有一个特定的位置用于执行业务逻辑验证和聚合状态更改。命令处理程序应决定聚合是否处于正确的状态。如果是，则发布事件。如果没有，则可能忽略该命令或抛出异常，这取决于域的需要。

任何命令处理函数都不应该发生状态改变。事件源处理程序应该是惟一更新聚合状态的方法。如果不能这样做，则意味着聚合将在从它的偶数来源时错过状态更改

聚合测试Fixture将防止命令处理函数中无意的状态更改。因此，建议为任何聚合实现提供完整的测试用例。

**何时处理事件**

聚合所需要的唯一状态是它做出决策所需要的状态。因此，只有当事件类似的状态更改需要用于驱动未来的验证时，才需要处理聚合发布的事件。

Applying Events from Event Sourcing Handlers应用来自事件源处理程序的事件

在某些情况下，特别是当聚合结构增长超过几个实体时，对在同一聚合的其他实体中发布的事件作出反应是更清晰的(多实体聚合在这里有更详细的解释)。但是，由于在重构聚合状态时也会调用事件处理方法，因此必须采取特殊的预防措施。

可以在事件来源处理程序方法中应用()新事件。这使得实体“B”有可能对实体“A”做某事做出反应。当重放历史事件时，Axon将忽略apply()调用。请注意，在从事件源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都收到第一个事件之后才发布给实体。如果需要发布更多的事件，则基于应用inner后实体的状态

**对其他事件的反应**

聚合不能处理来自其他源的事件。这是故意的，因为使用事件源处理程序来重新创建聚合的状态。为此，它只需要自己的事件，因为这些事件表示它的状态更改。

要使聚合对来自其他聚合实例的事件作出反应，应该利用saga或事件处理组件

**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命令处理程序描述了GiftCard聚合:

@CommandHandler带注释的构造函数

@CommandHandler带注释的方法

选项1总是期望是GiftCard聚合的实例化，而选项2期望以现有聚合实例为目标。虽然这可能是默认的，但是可以在命令处理程序上定义创建策略。这可以通过向命令处理程序注释的方法添加@CreationPolicy注释来实现，如下所示:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.modelling.command.CreationPolicy;

import org.axonframework.modelling.command.AggregateCreationPolicy;

​

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) {

// 一个'CreateOrRechargeCardCommand'-handler，它创建一个' GiftCard '聚合，如果它不存在 // 否则，它将更新现有的‘GiftCard’聚合

}

// 省略聚合状态、命令处理逻辑和事件来源处理程序}

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

1 ALWAYS—“ALWAYS”的创建策略将期望实例化聚合。这就像带注释的命令处理程序构造函数一样有效地工作。在不定义返回类型的情况下，将返回创建期间使用的聚合标识符。通过这种方法，可以在聚合标识符旁边返回其他结果。

2 CREATE\_IF\_MISSING—“如果缺失就创建”的创建策略既可以创建聚合，也可以对现有实例进行操作。 该策略应被视为聚合的创建或更新方法。

3 NEVER—“NEVER”的创建策略将在现有的聚合实例上处理。 这与任何常规的带注释的命令处理程序一样有效。

外部命令处理程序

命令处理函数通常直接放置在聚合上(在这里详细描述)。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。消息处理函数，如命令处理函数，可以放置在任何对象上。因此，可以实例化一个“命令处理对象”。

命令处理对象是一个简单的(常规的)对象，它具有@CommandHandler带注释的方法。与聚合不同，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类型的命令:

import org.axonframework.commandhandling.CommandHandler;

import org.axonframework.modelling.command.Repository;

​

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.

}

​

// omitted constructor

}

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

1 GiftCard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，直接访问存储库不是强制性的，而是一种设计选择。

2 要加载预期的GiftCard聚合实例，需要使用存储库#load(String)方法。 所提供的参数应该是聚合标识符。

3 在加载了该聚合之后，应该调用聚合#execute(Consumer)函数来对该聚合执行操作。使用execute函数可以确保聚合生命周期正确启动。

Implementations 实现

命令调度，如调度命令页面所示，有许多优点。首先，有一个对象清楚地描述了客户机的意图。通过记录命令，您可以存储意图和相关数据，以备将来参考。命令处理还使得向远程客户端公开命令处理组件(例如通过web服务)变得很容易。测试也变得简单多了。您可以通过定义开始情况(给定)、要执行的命令(何时)和期望的结果(然后)来定义测试脚本，方法是列出一些偶数最后一个主要的优点是，可以很容易地在同步和异步以及本地和分布式命令处理之间进行切换。

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的目标不是指定一种特定的工作方式，而是支持您按照自己的方式进行工作，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(参见工作单元)，并在该方法完成时对其执行提交或回滚。

下一节将概述与使用Axon设置命令调度基础设施相关的任务

**The Command Gateway**

命令网关是一个方便的命令调度机制接口。虽然您不需要使用网关来分派命令，但这通常是最简单的选择。

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

另一种选择也许是最灵活的。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定义应用程序的接口，并声明自己的(已检查的)业务异常。Axon将在运行时自动生成该接口的实现。

**配置命令网关**

您自定义的命令网关和Axon提供的命令网关至少都需要配置一个命令总线。此外，可以使用RetryScheduler、CommandDispatchInterceptors和CommandCallbacks来配置命令网关。

**RetryScheduler**

当命令执行失败时，RetryScheduler能够调度重试。当命令因显式非瞬态异常而失败时，根本不会进行重试。请注意，只有当由于RuntimeException而导致命令失败时，才会调用重试调度程序。检查过的异常被视为“业务异常”，永远不会触发重试。

**目前有两种实现:**

1 IntervalRetryScheduler将以设置的间隔重试给定的命令，直到成功， 或者发生重试的最大次数。

2 ExponentialBackOffIntervalRetryScheduler以指数间隔重试失败的命令，直到它成功了，或者已经进行了最大次数的重试。

CommandDispatchInterceptor

CommandDispatchInterceptors允许在将命令消息分派到命令总线之前修改命令消息。与命令总线上配置的CommandDispatchInterceptors相反，这些拦截器只在消息通过网关发送时被调用。例如，这些拦截器可以用于将元数据附加到命令或执行验证。

CommandCallback

可以在常规发送时向命令网关提供CommandCallback，指定如何处理命令处理结果。它与CommandMessage和commandsultmessage类一起工作，因此允许对所有通过该网关发送的命令(不管它们的类型是什么)进行一些通用行为。

创建自定义命令网关

Axon允许使用自定义接口作为命令网关。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，而且允许您在需要的地方模拟您的接口，从而使测试变得容易得多。

以下是参数如何影响命令网关的行为:

1 第一个参数预计是要分派的实际命令对象。

2 使用@MetaDataValue注释的参数将其值分配给元数据字段，并将标识符作为注释参数传递

3 MetaData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面参数的元数据，如果它们的键相等。

4 CommandCallback类型的参数将有它们的onResult(CommandMessage<?扩展C >, CommandResultMessage < ?extends R>)在命令被处理后被调用。尽管CommandCallback提供了一种处理命令处理结果的方法，但这并不影响您能否在自定义命令网关上定义返回类型。如果定义了回调和返回类型，那么回调的调用将始终与返回值(或异常)匹配。最后，要知道您可以传入几个CommandCallback实例，它们都将按顺序调用。

5 最后两个参数表示超时，可以是long(或int)和TimeUnit类型。 只要这些参数表明，该方法将最多阻塞。方法如何响应超时取决于在该方法上声明的异常(见下文)。请注意，如果该方法的其他属性完全阻止阻塞，则永远不会发生超时。

方法声明的返回值也会影响它的行为:

1 void返回类型将导致方法立即返回，除非方法上有需要等待的其他指示，比如超时或声明的异常。

2 返回类型Future、CompletionStage和CompletableFuture将导致该方法立即返回。

您可以使用该方法返回的CompletableFuture实例访问命令处理程序的结果。

在该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该方法阻塞，直到结果可用为止。

结果被强制转换为返回类型(如果类型不匹配，会引发ClassCastException)。

例外有以下效果:

1 如果命令处理程序(或拦截器)抛出了该类型的异常，则任何声明的checked异常都将被抛出。

如果抛出一个未声明的检查异常，它将被包装在一个RuntimeException中。

2 当超时发生时，默认行为是从方法返回null。这可以通过声明TimeoutException来更改。如果声明了这个异常，则抛出一个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，中断标志被设置回线程。通过在方法上声明InterruptedException，该行为将被更改为抛出该异常。在抛出异常时删除中断标志，这与java规范一致。

4 其他运行时异常可以在方法上声明，但除了向API用户澄清之外，不会有任何影响。

最后，还可以使用注释:

1 正如parameter部分中指定的那样，参数上的@MetaDataValue注释将该参数的值添加为元数据值。元数据项的键作为参数提供给注释。

2 带@Timeout注释的方法最多阻塞指定的时间。

3 用@Timeout注释的类将导致该类中声明的所有方法最多阻塞指定的时间，除非用它们自己的@Timeout注释或指定超时参数。

public interface MyGateway {

//发射后不管

void sendCommand(MyPayloadType command);

// 方法附加元数据，并将等待结果10秒

@Timeout(value = 10, unit = TimeUnit.SECONDS)

ReturnValue sendCommandAndWaitForAResult(MyPayloadType command,

@MetaDataValue("userId") String userId);

// 在超时时抛出异常的替代方法

@Timeout(value = 20, unit = TimeUnit.SECONDS)

ReturnValue sendCommandAndWaitForAResult(MyPayloadType command)

throws TimeoutException, InterruptedException;

// 这个方法也将等待，调用者决定等待多久

void sendCommandAndWait(MyPayloadType command, long timeout, TimeUnit unit)

throws TimeoutException, InterruptedException;

}

// 配置网关:

CommandGatewayFactory factory = CommandGatewayFactory.builder()

.commandBus(commandBus) .build();

//注意，commandBus可以从配置中获得

//对象返回 `configurer.initialize()`.

MyGateway myGateway = factory.createGateway(MyGateway.class);

**The Command Bus**

命令总线是将命令分派给Axon应用程序中各自的命令处理程序的机制。关于如何使用CommandBus的建议可以在这里找到。在这个框架中存在着几种命令总线，它们具有不同的特征:

**AxonServerCommandBus**

Axon提供了开箱即用的命令总线，即AxonServerCommandBus。它连接到AxonIQ AxonServer服务器以提交和接收命令。

AxonServerCommandBus是一个分布式命令总线。默认情况下，它使用SimpleCommandBus来处理不同JVM上的传入命令。

通过简单地声明对Axon -spring-boot-starter的依赖，Axon将自动配置Axon服务器命令总线:

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

<dependency>

<groupId>org.axonframework</groupId>

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

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

</dependency>

Excluding the Axon Server Connector 不包括Axon服务器连接器

如果排除了axon-server-connector依赖关系，则会退回到“non-axon-server”命令总线选项SimpleCommandBus(见下文)。

SimpleCommandBus

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。处理完命令后，将保存修改的聚合，并在同一线程中发布生成的事件。在大多数场景中，例如web应用程序，这种实现将适合您的需求。

像大多数CommandBus实现一样，simplecommanbus允许配置拦截器。当命令在命令总线上被分派时，CommandDispatchInterceptors被调用。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。更多信息请参见命令拦截器。

由于所有的命令处理都是在同一个线程中完成的，因此这个实现受到JVM边界的限制。这个实现的性能很好，但并不特别。要跨越JVM边界，或充分利用CPU周期，请查看其他CommandBus实现。

@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;

}

Excluding the Axon Server Connector

如果您从axon-spring-boot-starter依赖项中排除了axon-server-connector依赖项，SimpleCommandBus将为您自动配置。

**AsynchronousCommandBus**

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使用一个无限制的缓存线程池。这意味着在分派命令时创建一个线程。完成了命令处理的线程将被重新用于新命令。如果线程在60秒内没有处理某个命令，则会停止线程。

另外，还可以提供一个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，以确保任何等待线程都已正确关闭。要关闭，请调用shutdown()方法。这还将关闭任何提供的Executor实例，如果它实现了ExecutorService接口。

@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;

}

**DisruptorCommandBus**

simplecommanbus具有合理的性能特点。SimpleCommandBus需要锁定以防止多个线程并发地访问同一聚合，这一事实导致了处理开销和锁争用。

DisruptorCommandBus对多线程处理采取了不同的方法。与其让多个线程各自执行同一个进程，不如让多个线程各自处理进程的一部分。DisruptorCommandBus使用了Disruptor，一个用于并发编程的小框架，通过采用不同的多线程方法来实现更好的性能。与在调用线程中进行处理不同，这些任务被移交给两组线程，每组线程负责部分处理。第一组线程将执行命令处理程序，更改聚合的状态。第二组将存储事件并将其发布到事件存储中。

虽然DisruptorCommandBus很容易比simplecommanbus性能好4倍(!)，但还是有一些限制:

1 DisruptorCommandBus只支持事件来源的聚合。这个命令总线也充当由破坏者处理的聚合的存储库。要获得对存储库的引用，请使用createRepository(AggregateFactory)。

2 命令只能导致单个聚合实例中的状态更改。

3 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。

4 命令通常不会导致需要回滚工作单元的故障。当回滚发生时，DisruptorCommandBus不能保证命令按照它们被分派的顺序被处理。此外，它需要重试许多其他命令，导致不必要的计算。

5 在创建新的聚合实例时，更新已创建实例的命令可能不会完全按照提供的顺序执行。一旦创建了聚合，所有命令将完全按照它们被分派的顺序执行。若要确保顺序，请在创建命令上使用回调，以等待聚合被创建。应该不会超过几毫秒。

要构造一个DisruptorCommandBus实例，你需要一个EventStore。此组件在事件总线和事件存储部分进行了解释。

你还可以提供一个DisruptorConfiguration实例，它允许你调整配置，以优化特定环境的性能:

Buffer size - 在环形缓冲区上注册传入命令的槽位数。

更高的值可能会增加吞吐量，但也会导致更高的延迟。一定是2的幂。默认为4096。

ProducerType - 指示条目是由单个线程生成的还是多个线程生成的。默认为多个。

WaitStrategy - 当处理器线程(负责实际处理的三个线程)需要相互等待时使用的策略。

最佳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。

如果低延迟是至关重要的，并且DisruptorCommandBus可以为自己申请核心，你可以使用BusySpinWaitStrategy。

为了使命令总线占用较少的CPU，并允许其他线程进行处理，使用YieldingWaitStrategy。

最后，你可以使用SleepingWaitStrategy和BlockingWaitStrategy来允许其他进程公平地共享CPU。

如果不期望命令总线进行全职处理，则后者是合适的。

默认为BlockingWaitStrategy。

Executor - 设置为DisruptorCommandBus提供线程的执行程序。

此执行程序必须能够提供至少四个线程。

其中三个线程是由DisruptorCommandBus的处理组件声明的。

额外的线程用于调用回调，并在聚合的状态被检测到损坏时安排重试。

默认为一个CachedThreadPool，它提供来自名为“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事件的存储和发布在事务中执行。

InvokerInterceptors -定义在调用过程中使用的CommandHandlerInterceptors。这是调用实际命令处理程序方法的过程。

PublisherInterceptors - 定义将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 - 定义应该回滚工作单元的哪些异常。默认为回滚未检查异常的配置。

RescheduleCommandsOnCorruptState - 指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

CoolingDownPeriod - 指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。Defaults to 1000 (1 second).

Cache - 指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

InvokerThreadCount -指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

PublisherThreadCount - 指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

SerializerThreadCount -指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

Serializer - 指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

@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;

}

**Distributing the Command Bus**

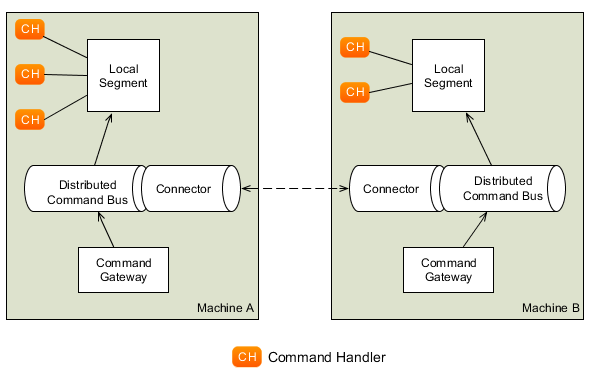
指示是否应该重新调度对已损坏(例如，由于工作单元已回滚)的聚合执行的命令。如果为false，则会调用回调函数的onFailure()方法。如果为true(默认值)，则该命令将被重新调度。

这就是“分发命令总线”概念的由来。分布式命令总线的默认实现是AxonServerCommandBus。它连接到AxonIQ AxonServer服务器以提交和接收命令。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”。

默认情况下，SimpleCommandBus被配置为处理不同JVM上的传入命令。你可以配置AxonServerCommandBus来使用其他命令总线实现来实现这个目的:AsynchronousCommandBus, DisruptorCommandBus。

**DistributedCommandBus**

DistributedCommandBus是分发命令总线(命令)的另一种方法。每个JVM上DistributedCommandBus的每个实例都被称为一个“段”。



分布式命令总线的结构

DistributedCommandBus依赖于两个组件:CommandBusConnector(实现JVM之间的通信协议)和CommandRouter(为每个传入命令选择目的地)。这个路由器根据路由策略计算的路由键定义DistributedCommandBus的哪个片段应该被给出一个'命令。具有相同routing key的两条命令总是被路由到同一个网段，只要网段的数量和配置没有变化。一般情况下，使用目标聚合的标识符作为路由ke

提供的两个实现RoutingStrategy: MetaDataRoutingStrategy,它使用命令消息的元数据属性查找路由的关键,而AnnotationRoutingStrategy使用@TargetAggregateIdentifier注释在命令消息有效负载提取路由的关键。显然，您还可以提供自己的实现。

默认情况下，当不能从命令消息中解析出任何键时，RoutingStrategy实现将抛出异常。这个行为可以通过在MetaDataRoutingStrategy或AnnotationRoutingStrategy的构造函数中提供UnresolvedRoutingKeyPolicy来改变。有三种可能的政策:

1 ERROR -默认值，当路由键不可用时将引发异常

2 RANDOM\_KEY -当一个'路由键无法从命令消息中解析时，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。

3 STATIC\_KEY—对于未解析的路由键，将返回一个静态键(为“未解析”)。这实际上意味着，只要段的配置没有改变，所有这些命令都将被路由到同一段。

您可以选择这些组件的不同风格，它们可以在其中一个扩展模块中使用: SpringCloud or JGroups.

配置分布式命令总线(大部分)可以在不修改配置文件的情况下完成。

首先，需要包含一个Axon分布式命令总线模块的启动器(例如JGroups或Spring Cloud)。

一旦出现这种情况，就需要向应用程序上下文添加单个属性，以启用分布式命令总线:

axon.distributed.enabled=true

有一种设置与所使用的连接器类型无关: axon.distributed.load-factor=100

分布式命令总线的负载因子默认为100。

The Load Factor Explained.

负载因子定义了实例与其他实例相比所承载的负载量。例如，如果您设置了两台机器，它们的负载系数都是100，那么它们都将承担相同的负载。在两台机器上将负载因数增加到200仍然意味着两台机器接收相同的负载。综上所述，负载系数旨在通过将更多的负载分配到速度较快的机器而不是速度较慢的机器来服务于异构应用程序环境。

Configuration 配置

本页面旨在描述配置命令模型的选项套件。

**Aggregate Configuration**

命令模型中的核心概念是实现的聚合。要实例化一个默认聚合配置，您只需执行以下操作:

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureAggregate(GiftCard.class);

}

@Aggregate注释(在org.axonframework.spring中。原型包)触发自动配置来设置必要的组件，以使用带注释的类型作为聚合。注意，只有聚合根需要注释。

Axon将自动将所有带@CommandHandler注释的方法注册到命令总线中，如果没有存储库，则设置一个存储库。

// ...

import org.axonframework.spring.stereotype.Aggregate;

// ...

@Aggregate

public class GiftCard {

@AggregateIdentifier

private String id;

@CommandHandler

public GiftCard(IssueCardCommand cmd) {

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

}

}

**Registering a Command Handler**

通常，命令处理程序函数直接放置在聚合上。当采用这种方法时，简单地像上面描述的那样注册聚合就足够注册它的所有命令处理程序方法了。

然而，外部命令处理程序需要直接注册为命令处理程序，示例如下:

给定以下命令处理程序的存在:

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

}

以下是需要注册GiftCardCommandHandler作为一个命令处理程序:

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

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

或者，可以使用一种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());

重复命令处理函数

正如消息传递概念部分中指定的，命令总是只有一个目的地。这意味着对于任何给定的命令应该只有一个命令处理程序方法。

默认情况下，当注册一个重复的命令处理程序方法时，将保留最后一个注册，并记录一个警告。可以通过指定不同的DuplicateCommandHandlerResolver来调整此行为，如运行时调优部分所述。

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));

}

// omitted constructor

}

**Command Model Repositories**

存储库是提供对聚合的访问的机制。存储库充当了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找到聚合。应该对查询数据库执行任何其他类型的查询。

在Axon框架中，所有存储库都必须实现存储库接口。这个接口规定了三个方法:load(标识符，版本)、load(标识符)和newInstance(factoryMethod)。load方法允许您从存储库加载聚合。可选的version参数用于检测并发的修改(参见冲突解决)。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大多数存储库所需的基本功能。Axon框架区分了保存聚合的当前状态的存储库(请参阅标准存储库)和存储聚合事件的存储库(请参阅事件源存储库)。

注意，Repository接口没有指定delete(identifier)方法。删除聚合是通过从聚合中调用aggregatelifcycle . markdeleted()方法来完成的。与其他迁移一样，删除聚合也是一种状态迁移，唯一的区别是在许多情况下是不可逆转的。你应该在你的聚合上创建你自己的有意义的方法，它将聚合的状态设置为“deleted”。这还允许您注册想要发布的任何事件。

要完全定制所使用的存储库，您可以在应用程序上下文中定义一个。为了让Axon框架将此存储库用于预期的聚合，请在@Aggregate注释上的repository属性中定义存储库的bean名称。或者，指定存储库的bean名称作为聚合的名称(第一个字符小写)，并以repository作为后缀。因此，对于GiftCard类型的类，默认存储库名称是giftcarrepository。如果没有找到具有该名称的bean, Axon将定义EventSourcingRepository(如果没有可用的EventStore，则该方法将失败)。

@Bean

public Repository<GiftCard> repositoryForGiftCard(EventStore eventStore) {

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

}

@Aggregate(repository = "repositoryForGiftCard")

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

注意，这需要对存储库进行完整配置，包括任何SnapshotTriggerDefinition或AggregateFactory，否则可能会自动配置。

//Configurer configurer = DefaultConfigurer.defaultConfiguration()

// .configureAggregate(

// AggregateConfigurer.defaultConfiguration(GiftCard.class)

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

// .eventStore(c.eventStore())

// .build())

// );

**Standard Repositories**

标准存储库存储聚合的实际状态。在每次更改时，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。根据您正在创建的应用程序的类型，这可能是最简单的解决方案。如果是这样，Axon提供一些构建块来帮助您实现这样的存储库。

Axon为标准存储库提供了一种开箱即用的实现:genericjprepository。它期望聚合是一个有效的JPA实体。它配置了一个EntityManagerProvider(它提供了EntityManager来管理实际持久性)和一个指定存储在存储库中聚合的实际类型的类。您还可以传入EventBus，当聚合调用静态aggregatelifcycle .apply()方法时，事件将被发布到哪个EventBus。

您还可以轻松地实现自己的存储库。在这种情况下，最好从抽象的LockingRepository扩展。作为聚合包装器类型，建议使用AnnotatedAggregate。有关示例，请参阅genericjprepository的源代码。

**Event Sourcing Repositories**

能够基于事件重构其状态的聚合根也可以配置为由事件源存储库加载。这些存储库并不存储聚合本身，而是存储聚合生成的一系列事件。基于这些事件，可以在任何时候恢复聚合的状态。

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

**Aggregate Factories**

您还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就可以使用从事件存储中加载的事件对其进行初始化。Axon框架附带了许多AggregateFactory实现，您可以使用它们。如果它们还不够，那么很容易创建自己的实现。

GenericAggregateFactory

GenericAggregateFactory是一种特殊的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类，并声明一个不进行任何初始化的默认无参数构造函数。

GenericAggregateFactory适用于大多数聚合不需要特殊注入非序列化资源的场景。

**SpringPrototypeAggregateFactory**

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

要将依赖项注入到聚合中，您需要在Spring上下文中配置聚合根的原型bean，它还定义了SpringPrototypeAggregateFactory。它使用Spring应用程序上下文来实例化聚合，而不是创建使用构造函数的常规实例。这也将在聚合中注入任何依赖项。

**Implementing your own Aggregate Factory**

在某些情况下，GenericAggregateFactory不能提供您需要的内容。例如，对于不同的场景，您可以有一个抽象聚合类型，具有多个实现(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完成此操作。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。您可以使用此信息选择实现并调用其构造函数。确保该构造函数没有应用任何事件;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提供了一个缓存，如果可用，可以从其中加载聚合。

**Events**

与Axon应用程序相关联的第二种“消息”是事件消息(或简单的事件)。参考指南的这一部分将详细介绍Axon框架提供的帮助促进事件消息开发的功能。

下面给出了各个小节的摘要。

Event Dispatchers / Event Handlers 使用Axon框架开发事件消息的功能

Event Processors / Event Bus & Event Store 使用Axon框架进行事件消息的非功能性开发

Event Serialization Axon框架提供的事件序列化功能的详细信息

Event Versioning Axon框架提供的事件版本控制功能的详细信息

本指南由一个视频教程补充，该视频教程对本节中解释的概念进行了实际演示。该教程可在

Event Dispatchers

事件发布可以来自Axon框架应用程序中的几个位置。一般来说，这些可以分为两大类:

从聚合分派事件

从常规组件分发事件

此页将描述如何从两个位置获取事件总线上的事件消息。有关Axon框架中事件发布和存储实现的更多细节，请阅读本节。

**Dispatching events from an Aggregate从聚合分派事件**

聚合或其实体通常是所有事件消息的起点。事件消息只是一个决定已经做出的通知;成功地解决了处理命令消息的问题。

要从聚合发布事件，需要从聚合实例的生命周期中发布事件。这是强制性的，因为我们希望聚合标识符绑定到事件消息。事件的发生是有秩序的，这也是本质。这是通过向聚合中的每个事件添加序列号来实现的。

AggregateLifecycle提供了一种实现上述目标的简单方法:

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

public class GiftCard {

@CommandHandler

public GiftCard(IssueCardCommand cmd) {

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

}

// omitted state, command and event sourcing handlers

}

AggregateLifecycle#apply(Object)将经历以下几个步骤:

AggregateLifecycle#apply(Object)将经历以下几个步骤:

1 检索聚合的当前范围。

2 聚合的最后一个已知序列号用于设置要发布的事件的序列号。

3 提供的事件有效负载，即对象，将被包装在EventMessage中。EventMessage还将接收上一步中的sequenceNumber，以及聚合它的标识符。

4 事件消息将从现在开始发布。事件将首先发送给聚合中所有感兴趣的事件处理程序。这对于事件溯源是必要的，以相应地更新聚合的状态。

5 在聚合本身处理了事件之后，它将在EventBus上发布。

聚合事件消息中的元数据

AggregateLifecycle还提供了一个apply(对象、元数据)函数。这可以用于附加特定于命令处理程序的元数据。

Dispatching events from a Non-Aggregate 从非聚合对象分派事件

在绝大多数情况下，聚合将通过应用它们来发布事件。然而，偶尔也有必要发布一个事件(可能是从另一个组件内部)，直接发布到事件网关:

private EventGateway eventGateway;

public void dispatchEvent() {

eventGateway.publish(new CardIssuedEvent("cardId", 100, "shopId"));

}

// omitted class and constructor

Event Dispatchers

事件发布可以来自Axon框架应用程序中的几个位置。一般来说，这些可以分为两大类:

从聚合分派事件

从常规组件分发事件

此页将描述如何从两个位置获取事件总线上的事件消息。有关Axon框架中事件发布和存储实现的更多细节，请阅读本节。

**Dispatching events from an Aggregate从聚合分派事件**

聚合或其实体通常是所有事件消息的起点。事件消息只是一个决定已经做出的通知;成功地解决了处理命令消息的问题。

要从聚合发布事件，需要从聚合实例的生命周期中发布事件。这是强制性的，因为我们希望聚合标识符绑定到事件消息。事件的发生是有秩序的，这也是本质。这是通过向聚合中的每个事件添加序列号来实现的。

AggregateLifecycle提供了一种实现上述目标的简单方法:

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

public class GiftCard {

@CommandHandler

public GiftCard(IssueCardCommand cmd) {

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

}

// omitted state, command and event sourcing handlers

}

AggregateLifecycle#apply(Object)将经历以下几个步骤:

AggregateLifecycle#apply(Object)将经历以下几个步骤:

1 检索聚合的当前范围。

2 聚合的最后一个已知序列号用于设置要发布的事件的序列号。

3 提供的事件有效负载，即对象，将被包装在EventMessage中。EventMessage还将接收上一步中的sequenceNumber，以及聚合它的标识符。

4 事件消息将从现在开始发布。事件将首先发送给聚合中所有感兴趣的事件处理程序。这对于事件溯源是必要的，以相应地更新聚合的状态。

5 在聚合本身处理了事件之后，它将在EventBus上发布。

**聚合事件消息中的元数据**

AggregateLifecycle还提供了一个apply(对象、元数据)函数。这可以用于附加特定于命令处理程序的元数据。

**Event Handlers**

在Axon中，一个对象可以声明许多事件处理程序方法，方法是用@EventHandler对它们进行注释。方法声明的参数定义了它将接收哪些事件。

**Axon提供了以下参数类型的开箱即用支持:**

1 第一个参数始终是事件消息的有效负载。如果事件处理程序不需要访问消息的有效负载，则可以在@EventHandler注释上指定预期的有效负载类型。指定后，第一个参数将使用下面指定的规则解析。如果希望将有效负载作为参数传递，则不要在注释上配置有效负载类型。

2 用@MetaDataValue注释的参数将解析为带有注释中指示的键的元数据值。如果required为false(默认)，则在元数据值不存在时传递null。如果required为true，则解析器将不匹配，并在元数据值不存在时阻止调用该方法。

3 MetaData类型的参数将注入EventMessage的整个元数据。

4 用@Timestamp注释的参数，类型为java.time。Instant(或java.time.temporal.Temporal)将解析为EventMessage的时间戳。这是事件产生的时间。

5 带@SequenceNumber注释的参数，类型为java.lang。Long或Long将解析为DomainEventMessage的sequenceNumber。这提供了事件生成的顺序(在事件产生的聚合范围内)。重要的是要注意DomainEventMessage只能来自聚合。因此已经被直接发布在EventBus/EventGateway上的事件不是DomainEventMessage的实现，因此不会解析序列号。

6 可分配给message的参数将注入整个EventMessage(如果消息可分配给该参数)。如果第一个参数的类型是message，那么它有效地匹配任何类型的事件，即使泛型参数建议不匹配。由于类型擦除，轴突无法检测到所期望的参数。在这种情况下，最好声明一个有效负载类型的参数，然后声明一个message类型的参数。

7 TrackingToken类型的参数将使当前TrackingToken与被注入的处理事件相关。

8 当使用Spring和激活Axon配置时(通过包括Axon Spring Boot Starter模块)，任何其他参数都将解析为自动连接bean，如果应用程序上下文中正好有一个可注入的候选对象可用的话。这允许您将资源直接注入@EventHandler带注释的方法。

您可以通过实现ParameterResolverFactory接口并创建一个名为

/META-INF/service/org.axonframework.common.annotation.ParameterResolverFactory 包含实现类的完全限定名的。

在所有情况下，每个监听器实例最多调用一个事件处理程序方法。Axon将使用以下规则搜索要调用的最特定的方法:

1 在类层次结构的实际实例级上(由this.getClass()返回)，所有带注释的方法都会被求值

2 如果找到一个或多个方法，其中所有参数都可以解析为一个值，则选择并调用具有最特定类型的方法

3 如果在类层次结构的这一层没有找到方法，则以同样的方式计算超类型

4 当到达层次结构的顶层时，没有找到合适的事件处理程序，则忽略该事件。

// 假设EventB扩展了EventA

// EventC扩展了EventB

// 并且注册了子侦听器的单个实例

public class TopListener {

@EventHandler

public void handle(EventA event) { }

@EventHandler

public void handle(EventC event) { }

}

public class SubListener extends TopListener {

@EventHandler

public void handle(EventB event) { }

}

在上面的例子中，子监听器的处理方法将被EventB以及EventC的所有实例调用(因为它扩展了EventB)。换句话说，TopListener的处理方法根本不会接收任何对EventC的调用。由于EventA不能赋值给EventB(它是它的超类)，这些将由TopListener中的handler方法处理。

**Registering event handlers**

事件处理组件使用EventProcessingConfigurer定义，可以从全局Axon配置器访问该组件。EventProcessingConfigurer用于配置EventProcessingConfiguration。通常，应用程序将定义一个单独的EventProcessingConfiguration，但更大的模块化应用程序可以选择为每个模块定义一个。

要用@EventHandler方法注册对象，在EventProcessingConfigurer中使用registerEventHandler()方法:

@Component

public class MyEventHandlerClass {

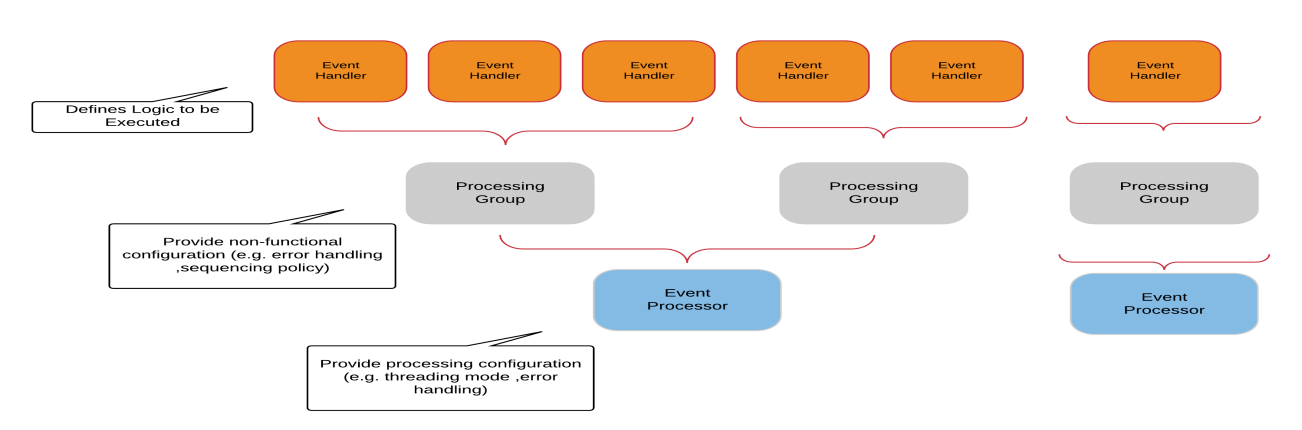
// contains @EventHandler(s)

}

Event Processors

事件处理程序定义接收到事件时要执行的业务逻辑。事件处理器是负责处理该处理的技术方面的组件。它们启动一个工作单元，可能还启动一个事务。但是，它们还确保关联数据可以正确地附加到事件处理期间创建的所有消息。

事件处理程序和事件处理程序的组织表示如下所示。



事件处理器大致有两种形式:订阅和跟踪。订阅事件处理器将自己订阅到事件源，并由发布机制管理的线程调用。另一方面，跟踪事件处理器使用自己管理的线程从源中提取消息。

**将处理程序分配给处理器**

所有处理器都有一个名称，用于跨JVM实例标识一个处理器实例。具有相同名称的两个处理器被认为是同一个处理器的两个实例。

所有事件处理程序都附加到名称为事件处理程序类的包名称的处理程序。

例如，以下类:

org.axonframework.example.eventhandling.MyHandler,

org.axonframework.example.eventhandling.MyOtherHandler, and

org.axonframework.example.eventhandling.module.MyHandler

将触发创建两个处理器:

org.axonframework.example.eventhandling with 2 handlers, and

org.axonframework.example.eventhandling.module with a single handler

配置API允许您配置其他策略，以便将类分配给处理器，或者甚至将特定的处理程序实例分配给特定的处理器。

对单个事件处理程序中的事件处理程序进行排序

要在事件处理程序中对事件处理程序进行排序，注册事件处理程序的顺序(如注册事件处理程序一节中所述)是指导。因此，事件处理程序调用事件处理程序进行事件处理的顺序与它们在配置API中的插入顺序相同。

如果选择Spring作为连接所有内容的机制，则可以通过添加@Order注释显式指定事件处理程序的顺序。这个注释应该放在事件处理程序类的类级别，并且应该提供一个整数值来指定排序。

这并不是说不能对不属于同一事件处理程序的事件处理程序进行排序。

**配置的处理器**

处理器负责处理事件的技术方面，而不管每个事件触发的业务逻辑是什么。然而，“常规”(单例、无状态)事件处理程序的配置方式与saga稍有不同，因为对于这两种处理程序来说，不同的方面都很重要。

**事件处理器**

默认情况下，Axon将使用跟踪事件处理器。可以更改处理程序的分配方式和配置方式。

EventProcessingConfigurer类定义了许多方法，可用于定义如何配置处理器。

1 registerEventProcessorFactory允许您定义一个默认的工厂方法，该方法创建没有明确定义工厂的事件处理器。

2 registerEventProcessor(字符串名称，EventProcessorBuilder builder)定义了用于创建给定名称的处理器的工厂方法。注意，只有选择name作为任何可用事件处理程序bean的处理程序时，才会创建这样的处理程序。

3 registerTrackingEventProcessor(字符串名称)定义使用默认设置将给定名称的处理器配置为跟踪事件处理器。它配置了一个TransactionManager和一个TokenStore，默认情况下它们都是从主配置中获取的。

registerTrackingProcessor(String name, Function<Configuration, StreamableMessageSource<TrackedEventMessage<?>>> source, Function<Configuration, TrackingEventProcessorConfiguration> processorConfiguration)

定义应将具有给定名称的处理器配置为跟踪处理器，并使用给定的trackingeventprocessconfiguration读取多线程的配置设置。

StreamableMessageSource定义了一个事件源，处理器应该从中提取事件。

usingSubscribingEventProcessors()将默认值设置为订阅事件处理器，而不是跟踪事件处理器。

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.eventProcessing(eventProcessingConfigurer -> eventProcessingConfigurer.usingSubscribingEventProcessors())

.eventProcessing(eventProcessingConfigurer -> eventProcessingConfigurer.registerEventHandler(c -> new MyEventHandler()));

**多个事件源**

可以将跟踪事件处理器配置为在处理事件时使用多个源。这对于跨域编译度量很有用，或者只是当事件分布在多个事件存储之间时。

拥有多个源意味着处理器可以在任意给定的瞬间使用多个事件。因此，您可以指定一个比较器在它们之间进行选择。默认实现选择时间戳最长的事件(即等待时间最长的事件)。

多个数据源还意味着需要使用一种策略将跟踪处理器的轮询间隔在多个数据源之间进行划分，以优化事件发现并最小化建立到数据源的昂贵连接的开销。因此，您可以在构建器中使用longPollingSource()方法来选择哪个源来完成大部分轮询。这可以确保一个源消耗大部分轮询间隔，同时还可以间歇性地检查其他源上的事件。默认的longPollingSource是在最后配置的源文件上执行的。

使用其构建器()创建一个MultiStreamableMessageSource，并在调用eventprocessingconfiguration . registertrackingeventprocessor()时将其注册为消息源。

例如:

MultiStreamableMessageSource.builder()

.addMessageSource("eventSourceA", eventSourceA)

.addMessageSource("eventSourceB", eventSourceB)

.longPollingSource("eventSourceA") // Overrides eventSourceB as the longPollingStream

.trackedEventComparator(priorityA) // Where 'priorityA' is a comparator prioritizing events from eventSourceA

.build();

Sagas

**可以更改传奇的配置方式。**

使用SagaConfigurer类配置传奇:

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.eventProcessing(eventProcessingConfigurer ->

eventProcessingConfigurer.registerSaga(

MySaga.class,

sagaConfigurer -> sagaConfigurer.configureSagaStore(c -> sagaStore)

.configureRepository(c -> repository)

.configureSagaManager(c -> manager)

)

);

Error Handling

错误是不可避免的。根据它们发生的地点，您可能想要做出不同的反应。

默认情况下，将记录事件处理程序引发的异常，并继续处理下一个事件。当处理器试图提交事务、更新令牌或在流程的任何其他部分中抛出异常时，该异常将被传播。对于跟踪事件处理器，这意味着处理器将进入错误模式，释放任何令牌，并以递增的间隔(从1秒开始，一直到最大60秒)重新尝试。订阅事件处理器将向提供事件的组件报告发布错误。

要改变这种行为，可以在两个级别上自定义Axon处理异常的方式:

**由事件处理程序方法引发的异常**

默认情况下，这些异常会被记录下来，处理将继续进行到下一个处理程序或消息。

可以为每个处理组配置此行为:

eventProcessingConfigurer.registerDefaultListenerInvocationErrorHandler(conf -> /\* create error handler \*/);

​// or for a specific processing group:

eventProcessingConfigurer.registerListenerInvocationErrorHandler("processingGroup", conf -> /\* create error handler \*/);

很容易实现自定义错误处理行为。要实现的错误处理方法提供了异常、已处理的事件和对正在处理消息的处理程序的引用。您可以选择重试、忽略或重新抛出异常。在后一种情况下，异常将上升到事件处理器级别。

**异常处理过程中**

发生在事件处理程序范围之外的异常，或从事件处理程序的范围溢出的异常，由ErrorHandler处理。默认行为取决于处理器实现:

TrackingEventProcessor将进入错误模式。然后，它将使用递增的回退周期重试处理事件。它将从1秒开始，每次尝试后加倍，直到达到每次尝试60秒的最大等待时间。这个回退时间确保了如果另一个节点能够成功处理事件，那么它将有机会声明处理事件所需的令牌。

SubscribingEventProcessor将拥有异常气泡，直到事件的发布组件，从而允许它相应地处理该异常。

要自定义行为，可以在事件处理程序级别配置错误处理程序。

eventProcessingConfigurer.registerDefaultErrorHandler(conf -> /\* create error handler \*/);

​// or for a specific processor:

eventProcessingConfigurer.registerErrorHandler("processorName", conf -> /\* create error handler \*/);

要实现自定义行为，请实现ErrorHandler的单一方法。根据提供的ErrorContext对象，您可以决定忽略错误、安排重试、执行死信队列传递或重新抛出异常。

**Token Store**

与订阅事件处理器不同，跟踪事件处理器需要一个令牌存储来存储它们的进度。跟踪处理器通过其事件流接收的每个消息都附带一个令牌。这个令牌允许处理器在以后的任何时间点重新打开流，从最后一个事件结束的地方开始。

配置API从全局配置实例中获取令牌存储，以及处理器需要的大多数其他组件。如果没有显式定义令牌存储，则使用InMemoryTokenStore，这在生产环境中是不推荐的。

要配置令牌存储，请使用EventProcessingConfigurer来定义要使用哪个实现。

为所有处理器配置默认令牌存储:

Configurer.eventProcessing().registerTokenStore(conf -> ... create token store ...)

另外，要为特定处理器配置令牌存储，请使用:

Configurer.eventProcessing().registerTokenStore("processorName", conf -> ... create token store ...)`.

注意，您可以在定义处理器的EventProcessingConfiguration中覆盖令牌存储，以便与跟踪处理器一起使用。在可能的情况下，建议使用令牌存储，将令牌存储在事件处理程序更新视图模型的数据库中。这样，对视图模型的更改就可以与更改的令牌一起自动存储。这刚好保证了一次处理语义。

**拆分和合并跟踪令牌**

可以通过拆分段来增加在高负载上处理事件的线程数量，并通过合并段来减少负载时的线程数量，从而调优跟踪事件处理器的性能。

分裂和合并是允许的在运行时，这允许你动态地控制段的数量。这可以通过Axon服务器API或Axon框架完成，使用TrackingEventProcessor中的方法splitSegment(int segmentId)和mergeSegment(int segmentId)，提供你想要分割或合并的段的segmentId。

段选择注意事项

通过使用Axon服务器分割/合并，可以为您选择最合适的分割或合并段。当直接使用Axon框架API时，分割/合并段应该由开发人员自己推导:

分割:为了达到公平的平衡，最好在最大的段上进行分割

合并:为了公平的平衡，理想的合并是在最小的段上执行

**并行处理**

跟踪处理器可以使用多个线程来处理事件流。它们通过声明一个由数字标识的段来实现这一点。通常，单个线程将处理单个段。

可以定义所使用的段的数量。当事件处理器第一次启动时，它可以初始化许多段。这个数字定义了可以同时处理事件的最大线程数。

运行跟踪事件处理器的每个节点都将尝试启动其配置的线程数量，以开始处理事件。TrackingEventProcessor可以为每个线程声明一个段，如果配置的线程数小于配置的段数，则可能无法处理某些事件。因此，建议每个节点上的线程数大于或等于段的总数。

事件处理程序可能对事件的顺序有特定的期望。如果是这种情况，处理器必须确保这些事件按照特定的顺序发送给这些处理程序。Axon使用了测序策略。SequencingPolicy是一个为任何给定消息返回值的函数。如果SequencingPolicy函数对于两个不同的事件消息的返回值相等，则意味着必须按顺序处理这些消息。默认情况下，Axon组件将使用SequentialPerAggregatePolicy，这使得由相同聚合实例发布的事件将按顺序处理。

saga实例永远不会被多个线程并发调用。因此，一个传奇的排序政策是无关紧要的。Axon将确保每个saga实例按照事件总线上发布的顺序接收到需要处理的事件。

并行处理和订阅事件处理器

注意，订阅事件处理器不管理它们自己的线程。因此，不可能配置它们应该如何接收事件。实际上，它们总是以每个聚合的顺序为基础工作，因为这通常是命令处理组件中的并发级别。

DefaultConfigurer.defaultConfiguration()

.eventProcessing(eventProcessingConfigurer -> eventProcessingConfigurer.registerTrackingEventProcessor(

"myProcessor",

c -> c.eventStore(),

c -> c.getComponent(

TrackingEventProcessorConfiguration.class,

() -> TrackingEventProcessorConfiguration.forParallelProcessing(3)

)

)

);

**顺序处理**

即使事件从它们的发布者异步处理，通常也希望按照发布的顺序处理某些事件。在轴突中，这是由测序策略控制的。

SequencingPolicy定义事件是必须顺序处理、并行处理还是两者结合处理。策略返回给定事件的序列标识符。如果策略为两个事件返回相同的标识符，这意味着它们必须由事件处理程序顺序处理。空序列标识符意味着事件可以与任何其他事件并行处理。

**Axon提供了一些常见的策略，您可以使用:**

1 FullConcurrencyPolicy将告诉Axon这个事件处理程序可以并发地处理所有事件。这意味着在需要按特定顺序处理的事件之间没有关系。

2 SequentialPolicy告诉Axon所有事件必须按顺序处理。事件的处理将在前一个事件的处理完成后开始。

3 默认策略是SequentialPerAggregatePolicy。它将强制顺序处理从同一聚合引发的域事件。但是，来自不同聚合的事件可以并发处理。对于从数据库表中的聚合中更新详细信息的事件侦听器来说，这通常是一个合适的策略。

除了这些提供的策略之外，您还可以定义自己的策略。所有策略都必须实现SequencingPolicy接口。这个接口定义了一个方法getSequenceIdentifierFor，它返回给定事件的序列标识符。必须按顺序处理返回相同序列标识符的事件。产生不同序列标识符的事件可以并发处理。如果事件可能与任何其他事件并行处理，策略实现可能会返回null。

**多节点处理**

对于跟踪处理器，处理事件的线程是运行在同一个节点上，还是运行在托管相同(逻辑)跟踪处理器的不同节点上并不重要。当一个具有相同名称的跟踪处理器的两个实例在不同的机器上处于活动状态时，它们被认为是同一个逻辑处理器的两个实例。他们将“竞争”事件流的片段。每个实例将“声明”一个段，防止分配给该段的事件在其他节点上被处理。

TokenStore实例将使用JVM的名称(通常是主机名和进程ID的组合)作为默认的nodeId。这可以在支持多节点处理的TokenStore实现中覆盖。

**分发事件**

Axon服务器默认支持在进程间环境中分发事件。或者，你也可以选择在扩展模块(Spring AMQP, Kafka)中找到的其他组件。

**事件重演**

当您想要重建投影(视图模型)时，重新播放过去的事件会很方便。其思想是从时间的起点开始，并重新调用所有事件处理程序。TrackingEventProcessor支持事件的重放。为了实现这一点，您应该在它上面调用resetTokens()方法。

重要的是，要知道在启动重置时跟踪事件处理器不能处于活动状态。因此，需要先关闭它，然后重置，一旦成功，就可以再次启动它。

通过TrackingEventProcessor初始化重播将打开一个API以进入重播过程。例如，可以定义@ResetHandler，这样您就可以在重置之前做一些准备工作。

让我们看看如何实现跟踪事件处理器的重放。首先，让我们来看一个简单的投影类:

@AllowReplay // 1.

@ProcessingGroup("card-summary")

public class CardSummaryProjection {

//...

@EventHandler

@DisallowReplay // 2. - It is possible to prevent some handlers from being replayed

public void on(CardIssuedEvent event) {

// 此事件处理程序执行一个 "side effect",

// 比如发电子邮件或短信

// 我们也不希望a 重放发生，因此我们禁止此方法重放

}

​

@EventHandler

public void on(CardRedeemedEvent event, ReplayStatus replayStatus /\* 3. \*/) {

我们可以在这里连接ReplayStatus

//事件被传递到我们的处理器作为'REGULAR'事件或

//重放事件

//执行事件处理

}

​

@ResetHandler // 4. - This method will be called before replay starts

public void onReset(ResetContext resetContext) {

// 做预重置逻辑，如清除投影表为a

/ /白纸。给定的resetContext是[可选的]，允许

//指定在什么上下文中执行重置。

}

//...

}

CardSummaryProjection显示了一些值得注意的有趣的事情，当它涉及到“意识到”重置正在进行:

1 可以使用@AllowReplay，它既可以位于整个类上，也可以位于@EventHandler注释的方法上。它定义了当重播传输时是否应该调用给定的类或方法。

2 除了允许重放，@DisallowReplay也可以使用。与@AllowReplay类似，它可以放在类级别和方法上，用于定义在replay传输过程中是否不应该调用类/方法。

3 为了更细粒度地控制重放期间要做什么(不做什么)，可以添加ReplayStatus参数。它是一个附加参数，可以添加到@EventHandler带注释的方法中，允许根据重播是否在传输中执行方法中的条件操作。

4 如果需要执行特定的预重置逻辑，比如清空投影表，则应该使用@ResetHandler注释。该注释只能放置在方法上，允许在必要时添加重置上下文。@ResetHandler中传递的resetContext源自操作，初始化TrackingEventProcessor#resetTokens(R resetContext)方法。resetContext的类型由用户决定。

所有这些就绪之后，我们就可以从TrackingEventProcessor开始重置了。为此，我们需要访问我们想要重置的TrackingEventProcessor。为此，您应该检索主配置中可用的EventProcessingConfiguration。此外，我们可以在这里提供一个可选的reset上下文，并在@ResetHandler中传递:

public class ResetService {

//...

public void reset(Configuration config) {

EventProcessingConfiguration eventProcessingConfig = config.eventProcessingConfiguration();

eventProcessingConfig.eventProcessor("card-summary", TrackingEventProcessor.class)

.ifPresent(processor -> {

processor.shutDown();

processor.resetTokens();

processor.start();

});

}

}

可以提供一个更改侦听器，它可以在重播完成时进行验证。更具体地说，EventTrackerStatusChangeListener可以通过TrackingEventProcessorConfiguration配置。有关更改侦听器的更多详细信息，请参阅监视和度量。

**部分重播**

当重置TrackingEventProcessor时，可以提供一个令牌位置;因此，可以指定从事件日志的哪个点开始重放事件。这将需要使用TrackingEventProcessor#resetTokens(TrackingToken)或TrackingEventProcessor#resetTokens(函数<StreamableMessageSource<TrackedEventMessage<?方法，两者都提供了应该启动重置的TrackingToken。

这里描述了如何自定义跟踪令牌位置。

**自定义跟踪令牌位置**

在Axon 3.3版之前，您只能将TrackingEventProcessor重置为事件流的开头。从3.3版开始，引入了从自定义位置启动TrackingEventProcessor的功能。TrackingEventProcessorConfiguration提供了一个选项，通过andInitialTrackingToken(函数<StreamableMessageSource, TrackingToken>)构建器方法为给定的TrackingEventProcessor设置一个初始令牌。作为令牌生成器函数的输入参数，我们接收到一个StreamableMessageSource，它为我们提供了三种构建令牌的可能性:

从事件流的头部:createHeadToken()。

从事件流的尾部:createTailToken()。

从某个时间点:createTokenAt(即时)和createTokenSince(持续时间)-

创建一个标记来跟踪给定时间后的所有事件。

如果恰好在那个给定时刻发生了一个事件，那么该事件也将被考虑进去。

当然，您可以完全忽略StreamableMessageSource输入参数，并自己创建一个令牌。

下面我们可以看到在“2007-12-03T10:15:30.00Z”上创建TrackingEventProcessorConfiguration的一个示例:

public class Configuration {

​

public TrackingEventProcessorConfiguration customConfiguration() {

return TrackingEventProcessorConfiguration

.forSingleThreadedProcessing()

.andInitialTrackingToken(streamableMessageSource -> streamableMessageSource.createTokenAt(

Instant.parse("2007-12-03T10:15:30.00Z")

));

}

}

Event Bus & Event Store

**Event Bus**

EventBus是将事件分派给订阅的事件处理程序的机制。Axon提供了三种事件总线实现:AxonServerEventStore、EmbeddedEventStore和SimpleEventBus这三种实现都支持订阅和跟踪处理器(参见事件处理器)。但是，AxonServerEventStore和EmbeddedEventStore持久化事件(请参阅事件存储)，这允许您在稍后的阶段重播它们。SimpleEventBus具有不稳定的存储，并且一旦将事件发布到订阅组件，就会立即“忘记”事件。

默认情况下配置了AxonServerEventStore事件总线/存储。

**Event Store**

事件源存储库需要一个事件存储库来存储和加载来自聚合的事件。事件存储提供了事件总线的功能。此外，它将持久化已发布的事件，并能够基于给定的聚合标识符检索以前的事件。

Axon Server as an event store

Axon提供了一个开箱即用的事件存储，即AxonServerEventStore。它连接到AxonIQ AxonServer服务器来存储和检索事件。

通过简单地声明一个依赖Axon -spring-boot-starter, Axon将自动配置事件总线/事件存储:

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

<dependency>

<groupId>org.axonframework</groupId>

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

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

</dependency>

**Excluding the Axon Server Connector**

如果您从axon-spring-boot-starter中排除了axon-server-connector依赖项，那么如果EventStorageEngine的具体实现可用，EmbeddedEventStore将为您自动配置。如果在类路径上检测到JPA或JDBC, JpaEventStorageEngine或JdbcEventStorageEngine将分别自动配置为EventStorageEngine。如果两者都没有，则自动配置退回到SimpleEventBus。

**Embedded event stor**

另外，Axon提供了一个非Axon服务器选项，EmbeddedEventStore。它将事件的实际存储和检索委托给EventStorageEngine。

有多种EventStorageEngine实现可用:

**JpaEventStorageEngine**

JpaEventStorageEngine将事件存储在与jpa兼容的数据源中。JPA事件存储将事件存储在条目中。这些项包含事件的序列化形式，以及存储元数据以快速查找这些项的一些字段。要使用JpaEventStorageEngine，您必须在类路径上有JPA (javax.persistence)注释。

默认情况下，事件存储需要你配置你的持久化上下文(例如在META-INF/persistence.xml文件中定义)来包含类DomainEventEntry和SnapshotEventEntry(这两个类都位于org.axonframework. eventsource .eventstore.jpa包中)。

下面是一个持久化上下文配置的示例配置:

<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

<persistence-unit name="eventStore" transaction-type="RESOURCE\_LOCAL"> (1)

<class>org...eventstore.jpa.DomainEventEntry</class> (2)

<class>org...eventstore.jpa.SnapshotEventEntry</class>

</persistence-unit>

</persistence>

1 在本例中，事件存储有一个特定的持久性单元。但是，您可以选择将第三行添加到任何其他持久性单元配置中。

2 这一行用持久化上下文注册DomainEventEntry (JpaEventStore使用的类)。

唯一键约束考虑在这种情况下，您必须依赖数据库来检测冲突。对事件存储的并发访问将导致键约束违背，因为该表只允许给定聚合和序列号的单个事件。因此，为具有现有序列号的现有聚合插入第二个事件将导致错误。

JpaEventStorageEngine可以检测此错误并将其转换为一个并发异常。但是，每个数据库系统报告这个违规的方式不同。如果您向JpaEventStore注册数据源，它将尝试检测数据库类型，并找出哪些错误代码表示键约束违反。或者，您可以提供一个PersistenceExceptionTranslator实例，它可以告诉给定的异常是否表示键约束违背。

如果没有提供DataSource或PersistenceExceptionTranslator，则按原样抛出数据库驱动程序的异常。

默认情况下，JpaEventStorageEngine需要一个EntityManagerProvider实现，该实现返回EntityManager实例供EventStorageEngine使用。这也允许使用应用程序管理的持久性上下文。提供EntityManager的正确实例是EntityManagerProvider的责任。

EntityManagerProvider有几个可用的实现，每个实现都满足不同的需求。SimpleEntityManagerProvider只返回在构造时提供给它的EntityManager实例。这使得实现成为容器管理上下文的一个简单选项。另外，还有ContainerManagedEntityManagerProvider，它返回默认的持久性上下文，并在JPA事件存储中默认使用。

如果你有一个名为“myPersistenceUnit”的持久化单元，你希望在JpaEventStore中使用，EntityManagerProvider实现可以像这样:

public class MyEntityManagerProvider implements EntityManagerProvider {

private EntityManager entityManager;

@Override

public EntityManager getEntityManager() {

return entityManager;

}

@PersistenceContext(unitName = "myPersistenceUnit")

public void setEntityManager(EntityManager entityManager) {

this.entityManager = entityManager;

}

默认情况下，JPA事件存储在DomainEventEntry和SnapshotEventEntry实体中存储条目。虽然在很多情况下这就足够了，但是您可能会遇到这样的情况:这些实体提供的元数据不够。您还可能希望在不同的表中存储不同聚合类型的事件。

如果是这样，您可以扩展JpaEventStorageEngine。它包含许多受保护的方法，您可以重写这些方法以调整其行为。

警告

请注意，持久性提供程序(如Hibernate)在其EntityManager实现中使用了一级缓存。通常，这意味着查询中使用或返回的所有实体都附加到EntityManager。只有当周围的事务被提交或在事务内部执行显式的“清除”时，它们才会被清除。当查询在事务的上下文中执行时，情况尤其如此。

要解决这个问题，请确保只查询非实体对象。您可以使用JPA的“SELECT new SomeClass(parameters) FROM…”样式查询来解决这个问题。或者，在获取一批事件后调用EntityManager.flush()和EntityManager.clear()。这样做的失败可能导致加载大的事件流时outofmemoryexception。

// The Event store `EmbeddedEventStore` delegates actual storage and retrieval of events to an `EventStorageEngine`.

@Bean

public EmbeddedEventStore eventStore(EventStorageEngine storageEngine,

AxonConfiguration configuration) {

return EmbeddedEventStore.builder()

.storageEngine(storageEngine)

.messageMonitor(configuration.messageMonitor(EventStore.class, "eventStore"))

.build();

}

// The JpaEventStorageEngine stores events in a JPA-compatible data source

@Bean

public EventStorageEngine storageEngine(

Serializer defaultSerializer,

PersistenceExceptionResolver persistenceExceptionResolver,

@Qualifier("eventSerializer") Serializer eventSerializer,

AxonConfiguration configuration,

EntityManagerProvider entityManagerProvider,

TransactionManager transactionManager) {

return JpaEventStorageEngine.builder()

.snapshotSerializer(defaultSerializer)

.upcasterChain(configuration.upcasterChain())

.persistenceExceptionResolver(persistenceExceptionResolver)

.eventSerializer(eventSerializer)

.entityManagerProvider(entityManagerProvider)

.transactionManager(transactionManager)

.build();

}

如果您从axon-spring-boot-starter中排除了axon-server-connector依赖项，那么如果EventStorageEngine的具体实现可用，EmbeddedEventStore将为您自动配置。如果在类路径上检测到JPA或JDBC, JpaEventStorageEngine或JdbcEventStorageEngine将分别自动配置为EventStorageEngine。如果两者都没有，则自动配置退回到SimpleEventBus。

**JdbcEventStorageEngine**

JDBC事件存储引擎使用JDBC连接将事件存储在JDBC兼容的数据存储中。通常，这些都是关系数据库。理论上，任何具有JDBC驱动程序的东西都可以用来支持JdbcEventStorageEngine。

与JPA类似，JDBCEventStorageEngine将事件存储在条目中。默认情况下，每个事件存储在单个条目中，该条目对应于表中的一行。一个表用于事件，另一个表用于快照。

JdbcEventStorageEngine使用ConnectionProvider获取连接。通常，这些连接可以直接从数据源获得。然而，Axon将把这些连接绑定到一个工作单元，以便在一个工作单元中使用单个连接。这可以确保使用单个事务存储所有事件，即使多个工作单元嵌套在同一个线程中也是如此。

// The Event store `EmbeddedEventStore` delegates actual storage and retrieval of events to an `EventStorageEngine`.

@Bean

public EmbeddedEventStore eventStore(EventStorageEngine storageEngine, AxonConfiguration configuration) {

return EmbeddedEventStore.builder()

.storageEngine(storageEngine)

.messageMonitor(configuration.messageMonitor(EventStore.class, "eventStore"))

.build();

}

建议Spring用户使用SpringDataSourceConnectionProvider将来自数据源的连接附加到现有事务。

// EventStorageEngine implementation that uses JDBC to store and fetch events.

@Bean

public JdbcEventStorageEngine eventStorageEngine(ConnectionProvider connectionProvider) {

return JdbcEventStorageEngine.builder()

.connectionProvider(connectionProvider)

.transactionManager(NoTransactionManager.INSTANCE)

.build();

如果您从axon-spring-boot-starter中排除了axon-server-connector依赖项，那么如果EventStorageEngine的具体实现可用，EmbeddedEventStore将为您自动配置。当需要使用JDBC作为事件存储方法时，可以直接提供JdbcEventStorageEngine bean，或者通过公开数据源bean让Axon自动配置它。

**SQL语句可定制性**

数据库在不同的场景下执行的最佳SQL语句之间有一些细微的差异。由于对所有可能性进行优化超出了框架的范围，因此可以调整使用的默认语句。

检查JdbcEventStorageEngineStatements实用程序类中JdbcEventStorageEngine使用的默认语句。此外，org.axonframework. eventsource .eventstore.jdbc。statements包包含一组可调语句。每个语句构建器都可以通过JdbcEventStorageEngine.Builder进行定制

MongoEventStorageEngine

MongoDB是一个基于NoSQL的文档存储。它的可伸缩性特征使其适合用作事件存储。Axon提供了MongoEventStorageEngine，它使用MongoDB作为后台数据库。它包含在Axon Mongo模块中(Maven artifactId Axon - Mongo)。

事件存储在两个单独的集合中:一个用于事件流，一个用于快照。

默认情况下，MongoEventStorageEngine将每个事件存储在单独的文档中。但是，可以更改所使用的存储策略。Axon提供的替代方法是DocumentPerCommitStorageStrategy，它为存储在单个提交(即在同一个DomainEventStream中)中的所有事件创建单个文档。

在单个文档中存储整个提交的优点是，提交是按原子方式存储的。此外，对于任意数量的事件，它只需要一次往返。缺点是在数据库中直接查询事件变得更加困难。例如，重构领域模型时，如果事件包含在“提交文档”中，则很难将它们从一个聚合体“转移”到另一个聚合体。

MongoEventStorageEngine不需要很多配置。它只需要一个对集合的引用来存储事件，然后就可以开始了。对于生产环境，您可能需要再次检查集合上的索引。

// The Event store `EmbeddedEventStore` delegates actual storage and retrieval of events to an `EventStorageEngine`.

@Bean

public EmbeddedEventStore eventStore(EventStorageEngine storageEngine, AxonConfiguration configuration) {

return EmbeddedEventStore.builder()

.storageEngine(storageEngine)

.messageMonitor(configuration.messageMonitor(EventStore.class, "eventStore"))

.build();

}

// The `MongoEventStorageEngine` stores each event in a separate MongoDB document

@Bean

public EventStorageEngine storageEngine(MongoClient client) {

return MongoEventStorageEngine.builder().mongoTemplate(DefaultMongoTemplate.builder().mongoDatabase(client).build()).build();

}

如果您从axon-spring-boot-starter中排除了axon-server-connector依赖项，那么如果EventStorageEngine的具体实现可用，EmbeddedEventStore将为您自动配置。当希望使用Mongo作为事件存储方法时，这意味着提供一个MongoEventStorageEngine bean。

**事件存储工具**

Axon提供了许多事件存储引擎，这些引擎在某些情况下可能很有用。

内存中的事件存储

InMemoryEventStorageEngine将存储的事件保存在内存中。虽然它的性能可能优于其他任何事件存储，但它并不真正适合长期生产使用。但是，它在寿命较短的工具或需要事件存储的测试中非常有用。

// The Event store `EmbeddedEventStore` delegates actual storage and retrieval of events to an `EventStorageEngine`.

@Bean

public EmbeddedEventStore eventStore(EventStorageEngine storageEngine, AxonConfiguration configuration) {

return EmbeddedEventStore.builder()

.storageEngine(storageEngine)

.messageMonitor(configuration.messageMonitor(EventStore.class, "eventStore"))

.build();

}

// The `InMemoryEventStorageEngine` stores each event in memory

@Bean

public EventStorageEngine storageEngine() {

return new InMemoryEventStorageEngine();

}

如果您从axon-spring-boot-starter中排除了axon-server-connector依赖项，那么如果EventStorageEngine的具体实现可用，EmbeddedEventStore将为您自动配置。当希望使用内存事件存储方法时，这意味着提供一个InMemoryEventStorageEngine Bean。

**将多个事件存储合并为一个**

SequenceEventStorageEngine是两个其他事件存储引擎的包装器。当读取时，它从两个事件存储引擎返回事件。追加的事件只追加到第二个事件存储引擎。例如，在出于性能原因使用了两种不同的事件存储实现的情况下，这是很有用的。第一个将是一个更大但更慢的事件存储，而第二个针对快速读写进行了优化。

**存储事件过滤**

FilteringEventStorageEngine允许基于谓词过滤事件。只存储与给定谓词匹配的事件。请注意，使用事件存储作为事件源的事件处理器可能不会接收这些事件，因为它们没有被存储。

**影响序列化过程**

事件存储需要一种方法来序列化事件，以便为存储做好准备。默认情况下，Axon使用XStreamSerializer，它使用XStream将事件序列化为XML。XStream相当快，而且比Java序列化更灵活。此外，XStream序列化的结果是人类可读的。这使得它对于日志记录和调试非常有用。

可以配置XStreamSerializer。您可以定义它应该用于某些包、类甚至字段的别名。别名不仅是缩短可能很长的名称的好方法，还可以在事件的类定义发生变化时使用别名。有关别名的更多信息，请访问XStream网站。

另外，Axon还提供了JacksonSerializer，它使用Jackson将事件序列化为JSON。虽然它产生了更紧凑的序列化形式，但它确实要求类遵循Jackson所要求的约定(或配置)。

您还可以实现自己的序列化器，只需创建一个实现序列化器的类，并配置事件存储以使用该实现而不是默认实现。

You can specify a serializer in your application.properties:

# somewhere in your `application.properties`

axon.serializer.events=jackson # posible values: java, xstream, jackson

另外，你也可以在Spring上下文中显式地定义你的序列化器:

// somewhere in your `@Configuration` class

@Qualifier("eventSerializer")

@Bean

public Serializer eventSerializer() {

return JacksonSerializer.builder().build();

}

**序列化事件vs ' rest'**

可以使用不同的序列化器来存储事件，而不是Axon需要序列化的所有其他对象(如命令、快照、传奇等)。虽然XStreamSerializer能够序列化几乎任何东西，这使它成为一个非常不错的默认值，但它的输出并不总是可以与其他应用程序共享的表单。JacksonSerializer创建了更好的输出，但需要在对象中使用特定的结构来序列化。这种结构通常出现在事件中，使其成为非常合适的事件序列化程序。

如果没有配置显式的eventSerializer，则使用已配置的主序列化器(默认为XStreamSerializer)对事件进行序列化。

Event Versioning

在Axon应用程序的生命周期中，事件通常会改变它们的格式。由于事件被无限期地存储，应用程序应该能够处理一个事件的多个版本。这一章将讨论在创建事件时要记住什么，以便向后(和向前)兼容。它也将解释向上铸造过程。

Event Upcasting

由于软件应用程序不断变化的本质，事件定义很可能也会随着时间而变化。由于事件存储被认为是一个只读和追加的数据源，所以您的应用程序必须能够读取所有事件，而不管它们是何时添加的。这就是向上投射的由来。

upcasting最初是面向对象编程的概念，其中“一个子类在需要时自动转换为它的超类”，upcasting的概念也可以应用于事件来源。upcast是指将事件从原来的结构转化为新的结构。与OOP向上转换不同，事件向上转换不能在完全自动化的情况下完成，因为新事件的结构对旧事件是未知的。必须提供手工编写的上浇注程序来指定如何将旧结构上浇注到新结构。

Upcasters是接受一个版本x的输入事件并输出0个或多个版本x + 1的新事件的类。此外，上行卡斯特是在一个链中处理的，这意味着一个上行卡斯特的输出被发送到下一个的输入。这允许您以增量的方式更新事件，为每个新的事件修订编写upcaster，使它们更小、更独立、更容易理解。

请注意

upcasting最大的好处可能是它允许您进行非破坏性的重构。换句话说，完整的事件历史保持不变。

在本节中，我们将解释如何编写upcaster，描述与Axon一起提供的不同(抽象)upcaster实现，并解释事件的序列化表示如何影响upcaster的编写。

为了允许upcaster查看他们接收的序列化对象的版本，事件存储库存储了一个修订号以及事件的完全限定名。这个修订号是由一个在序列化程序中配置的修正解析器生成的。Axon提供了几个revision - resolver的实现:

1 AnnotationRevisionResolver检查事件有效负载上的@Revision注释。

2 SerialVersionUIDRevisionResolver使用Java序列化API定义的serialVersionUID。

3 FixedValueRevisionResolver总是返回一个预定义的值。这在注入当前应用程序版本时特别有用。它将允许您查看哪个版本的应用程序生成了特定的事件。

4 Maven用户可以使用Maven artifactrevisionresolver自动使用项目版本。它使用项目的groupId和artifactId进行初始化，以获得版本。因为这只能在Maven创建的JAR文件中工作，所以版本不能总是由IDE解析。如果一个版本无法解析，则返回null。

Axon的upcasters不直接使用EventMessage，而是使用intermediateevent表示。IntermediateEventRepresentation提供了检索所有必要字段以构造EventMessage(因此也是upcast的EventMessage)的功能，以及实际的upcast函数。默认情况下，这些向上转换函数只允许调整事件的有效负载、有效负载类型和添加到事件元数据的内容。upcast函数中事件的实际表示可能会根据使用的事件序列化器或期望处理的形式而变化，因此IntermediateEventRepresentation的upcast函数允许选择预期的表示类型。其他字段，例如消息/聚合标识符、聚合类型、时间戳等，不能通过IntermediateEventRepresentation进行调整。调整这些字段并不是上行者的预期工作。因此，所提供的IntermediateEventRepresentation实现并不提供这些选项。

Axon框架中用于事件的基本Upcaster接口工作在一个intermediateevent表示流上，并返回一个intermediateevent表示流。因此，上行转换过程不会直接返回引入的上行转换函数的最终结果，而是通过堆叠intermediateevent表示将每个上行转换函数从一个修订链接到另一个修订。一旦发生了这个过程并从它们中提取了最终结果，即在序列化事件上执行实际的upcasting函数时。

提供了抽象的Upcaster实现

如前所述，Upcaster接口不会向上cast单个事件;它需要一个流<IntermediateEventRepresentation>并返回一个。然而，通常会写一个向上的发送器来调整流中的单个事件。更复杂的上投设置也可以想象。例如，从一个事件到多个事件，或者从一个早期事件中提取状态并将其推入一个后期事件的upcaster。本节描述了当前提供的(抽象的)事件上行转换实现，用户可以通过扩展来添加自己想要的上行转换功能。

1 SingleEventUpcaster -一对一的事件upcaster实现。从这个实现扩展需要实现canUpcast和doUpcast函数，它们分别检查手边的事件是否要进行上转换，以及如果要进行上转换，应该如何进行上转换。这是最可能扩展的实现，因为大多数事件调整都基于自包含的数据，并且是一对一的。

2 EventMultiUpcaster -一对多的事件upcaster实现。它与SingleEventUpcaster基本相同，唯一的例外是doUpcast函数返回一个流，而不是单一的IntermediateEventRepresentation。因此，该upcaster允许您将单个事件转换为多个事件。例如，如果您想要从fat事件中获得更细粒度的事件，这可能会很有用。

3 ContextAwareSingleEventUpcaster——一个一对一的upcaster实现，它可以在过程中存储事件上下文。除了canUpcast和doUpcast之外，感知上下文的upcaster还需要实现buildContext函数，该函数用于实例化通过upcaster的事件之间携带的上下文。canUpcast和doUpcast函数接收上下文作为第二个参数，紧邻IntermediateEventRepresentation。然后可以在向上转换过程中使用上下文从早期事件中提取字段并填充其他事件。因此，它允许您将一个字段从一个事件移动到一个完全不同的事件。

4 ContextAwareEventMultiUpcaster——一对多的upcaster实现，它可以在过程中存储事件上下文。这个抽象实现是EventMultiUpcaster和ContextAwareSingleEventUpcaster的组合，因此服务于保持intermediateevent表示的上下文，并将一个这样的表示向上转换为多个。如果您不仅希望将字段从一个事件复制到另一个事件，而且需要在流程中生成多个新事件，那么此实现是非常有用的。

5 EventTypeUpcaster -一个完整的upcaster实现，用于改变事件类型。EventTypeUpcaster是SingleEventUpcaster的一个实现，带有预定义的canUpcast和doUpcast函数，可以将事件从一种事件类型更改为另一种事件类型。例如，这可以用来轻松地更改事件的类名或包名。要创建EventTypeUpcaster，建议使用

EventTypeUpcaster#from(String expectedPayloadType, expectedRevision) and EventTypeUpcaster.Builder#to(upcastedPayloadType, upcastedRevision) methods.

Writing an Upcaster

下面的Java代码片段将作为一对一upcaster (SingleEventUpcaster)的基本示例。

旧版本的事件:

@Revision("1.0")

public class ComplaintEvent {

private String id;

private String companyName;

// Constructor, getter, setter...

}

新版本的事件:

@Revision("2.0")

public class ComplaintEvent {

private String id;

private String companyName;

private String description; // New field

// Constructor, getter, setter...

}

从1.0版升级到2.0版:

// Upcaster from 1.0 revision to 2.0 revision

public class ComplaintEventUpcaster extends SingleEventUpcaster {

private static SimpleSerializedType targetType =

new SimpleSerializedType(ComplainEvent.class.getTypeName(), "1.0");

@Override

protected boolean canUpcast(IntermediateEventRepresentation

intermediateRepresentation) {

return intermediateRepresentation.getType().equals(targetType);

}

@Override

protected IntermediateEventRepresentation doUpcast(

IntermediateEventRepresentation intermediateRepresentation) {

return intermediateRepresentation.upcastPayload(

new SimpleSerializedType(targetType.getName(), "2.0"),

org.dom4j.Document.class,

document -> {

document.getRootElement()

.addElement("description")

.setText("no complaint description"); // Default value

return document;

}

);

}

}

Spring Boot configuration:

@Configuration

public class AxonConfiguration {

@Bean

public SingleEventUpcaster myUpcaster() {

return new ComplaintEventUpcaster();

}

@Bean

public JpaEventStorageEngine eventStorageEngine(Serializer eventSerializer,

Serializer snapshotSerializer,DataSource dataSource,

SingleEventUpcaster myUpcaster,EntityManagerProvider entityManagerProvider,

TransactionManager transactionManager) throws SQLException {

return JpaEventStorageEngine.builder()

.eventSerializer(eventSerializer)

.snapshotSerializer(snapshotSerializer)

.dataSource(dataSource)

.entityManagerProvider(entityManagerProvider)

.transactionManager(transactionManager)

.upcasterChain(myUpcaster)

.build();

}

}

# Event Serialization

Event Serialization

事件存储需要一种方法来序列化事件，以便为存储做好准备。默认情况下，Axon使用XStreamSerializer，它使用XStream将事件序列化为XML。XStream相当快，而且比Java序列化更灵活。此外，XStream序列化的结果是人类可读的。这使得它对于日志记录和调试非常有用。

可以配置XStreamSerializer。您可以定义它应该用于某些包、类甚至字段的别名。别名不仅是缩短可能很长的名称的好方法，还可以在事件的类定义发生变化时使用别名。有关别名的更多信息，请访问XStream网站。

另外，Axon还提供了JacksonSerializer，它使用Jackson将事件序列化为JSON。虽然它产生了更紧凑的序列化形式，但它确实要求类遵循Jackson所要求的约定(或配置)。

您还可以实现自己的序列化器，只需创建一个实现序列化器的类，并配置事件存储以使用该实现而不是默认实现。

**序列化器实现**

序列化器在Axon框架中有几种风格，用于各种主题。目前，您可以在XStreamSerializer、JacksonSerializer和JavaSerializer之间进行选择，以序列化消息(命令/查询/事件)、令牌、快照和Axon应用程序中的saga。

由于有几个要序列化的对象，通常希望选择哪个序列化器处理哪个对象。为此，Configuration API允许你定义default、消息和事件序列化器，这将导致以下对象序列化分解:

1 事件序列化器负责反/序列化事件消息。事件通常存储在事件存储中很长一段时间。这是选择事件序列化器实现的主要驱动程序。

2 消息序列化器负责反/序列化命令和查询消息(在分布式应用程序设置中使用)。消息在节点之间共享，通常需要互操作和/或紧凑。在选择消息序列化器实现时要考虑到这一点。

3 默认的序列化器负责反/序列化其余部分，包括令牌、快照和传奇。这些对象通常不会在不同的应用程序之间共享，而且这些类中的大多数都不希望拥有一些基于Jackson的序列化器通常需要的getter和setter。像XStream这样灵活的通用序列化程序非常适合这个目的。

默认情况下，所有三种序列化器风格都设置为使用XStreamSerializer, XStreamSerializer在内部使用XStream将对象序列化为XML格式。XML是一种需要序列化的冗长格式，但是XStream的主要优点是几乎可以序列化任何内容。在存储令牌、传奇或快照时，这种冗长通常没有问题，但对于消息(特别是事件)，XML可能由于其序列化的大小而成本太高。因此，出于优化原因，您可以为消息配置不同的序列化器。定制序列化器的另一个非常有效的原因是为了实现不同(Axon)应用程序之间的互操作性，其中接收端可能强制使用特定的序列化格式。

可配置序列化程序之间存在隐式排序。如果未配置事件序列化程序，则消息序列化程序将执行事件反/序列化。反过来，如果没有配置消息序列化器，则默认的序列化器将扮演该角色。

我们使用XStreamSerializer作为默认值，使用JacksonSerializer作为所有消息，请参阅以下示例，了解如何具体配置每个序列化器:

# Possible values for these keys are `default`, `xstream`, `java`, and `jackson`.

axon.serializer.general

axon.serializer.events

axon.serializer.messages

# Possible values for these keys are `default`, `xstream`, `java`, and `jackson`.

axon:

serializer:

general:

events:

messages:

Serializing events vs 'the rest'

可以使用不同的序列化器来存储事件，而不是Axon需要序列化的所有其他对象(如命令、快照、传奇等)。虽然XStreamSerializer能够序列化几乎任何东西，这使它成为一个非常不错的默认值，但它的输出并不总是可以与其他应用程序共享的表单。JacksonSerializer创建了更好的输出，但需要在对象中使用特定的结构来序列化。这种结构通常出现在事件中，使其成为非常合适的事件序列化程序。

使用Configuration API，您可以简单地注册一个事件序列化器，如下所示:

Configurer configurer = ... // initialize

configurer.configureEventSerializer(conf -> /\* create serializer here\*/);

如果没有配置显式的eventSerializer，则使用已配置的主序列化器(默认为XStreamSerializer)对事件进行序列化。

**ContentTypeConverters**

upcaster工作于给定的内容类型(例如dom4j文档)。为了在上移者之间提供额外的灵活性，链接上移者之间的内容类型可能会有所不同。Axon将尝试使用ContentTypeConverters在内容类型之间自动转换。它将搜索从类型x到类型y的最短路径，执行转换并将转换后的值传递给请求的upcaster。出于性能方面的原因，只有当接收上投端的canUpcast方法为true时，才会执行转换。

ContentTypeConverters可能依赖于所使用的序列化器的类型。尝试将byte[]转换为dom4j文档是没有意义的，除非使用一个序列化程序将事件写入为XML。Axon框架将只使用通用内容类型转换器(例如将字符串转换为byte[]或将byte[]转换为InputStream的转换器)和序列化器上配置的用于反序列化消息的转换器。这意味着，如果您使用基于JSON的序列化器，您将能够在特定于JSON的格式之间进行转换。

提示

为了获得最佳性能，请确保同一链(其中一个的输出是另一个的输入)中的所有上移者处理相同的内容类型。

如果Axon没有提供您需要的内容类型转换，您总是可以通过实现ContentTypeConverter接口自己编写一个。

XStreamSerializer支持dom4j以及XOM和XML文档表示。JacksonSerializer支持Jackson的JsonNode。

**序列化器调优**

当序列化过程证明没有达到预期的要求时，可能需要考虑以下几点。

**XStreamSerializer**

XStream是非常可配置和可扩展的。如果只使用普通的XStreamSerializer，那么有一些快速的方法可供选择。XStream允许您为包名和事件类名配置别名。别名通常要短得多(特别是当您的包名很长时)，这使得事件的序列化形式更小。由于我们讨论的是XML，从XML中删除的每个字符都是利润的两倍(一个用于开始标记，一个用于结束标记)。

XStream中一个更高级的主题是创建自定义转换器。默认的基于反射的转换器很简单，但不会生成最紧凑的XML。始终要仔细查看生成的XML，看看是否真的需要所有这些信息来重构原始实例。

尽可能避免使用上脚轮。XStream允许在字段更改名称时使用别名。假设事件的修订0使用了一个名为“clientId”的字段。业务部门更喜欢使用术语“customer”，因此使用名为“customerId”的字段创建了修订1。这可以在XStream中使用字段别名完全配置。您需要配置两个别名，顺序如下:别名“customerId”到“clientId”，别名“customerId”到“customerId”。这将告诉XStream，如果遇到一个名为“customerId”的字段，它将调用相应的XML元素“customerId”(第二个别名覆盖第一个别名)。但是，如果XStream遇到名为“clientId”的XML元素，则它是已知的别名，并将被解析为字段名“customerId”。查看XStream文档获取更多信息。

为了获得最终的性能，您最好完全不使用基于反射的机制。在这种情况下，最明智的做法可能是创建一个定制的序列化机制。DataInputStream和DataOutputStream允许您轻松地将事件的内容写入输出流。ByteArrayOutputStream和ByteArrayInputStream允许对字节数组进行写入和读取。

防止重复序列化

特别是在分布式系统中，事件消息需要在多个场合序列化。Axon的组件意识到了这一点，并支持SerializationAware消息。如果检测到SerializationAware消息，则使用它的方法来序列化对象，而不是简单地将有效负载传递给序列化器。这允许进行性能优化。

当您自己序列化消息时，并且希望从SerializationAware优化中获益，请使用MessageSerializer类来序列化消息的有效负载和元数据。所有优化逻辑都在这个类中实现。更多细节请参阅MessageSerializer的JavaDoc。

**事件的不同序列化程序**

当使用事件源时，序列化的事件可能会持续很长时间。因此，请仔细考虑它们所序列化的格式。考虑为事件配置一个单独的序列化器，并仔细优化它们的存储方式。Jackson生成的JSON格式通常比XStream的XML格式更适合长期使用。有关如何将yourEventSerializer配置为不同的东西的更多信息，请参阅关于serializer的文档。

**仁慈的反序列化**

从序列化程序的角度来看，“宽容”意味着序列化程序可以忽略未知的属性。如果它这样处理要反序列化的格式，那么当它无法在序列化格式中找到给定字段的字段/ setter /构造函数参数时，它就不会失败。

启用宽松的序列化对于容纳不同的消息版本特别有帮助。当使用事件存储时，这种情况会很自然地发生，因为事件的格式会随着时间的推移而改变。但是，如果一个应用程序的几个不同版本同时运行，这也可能发生在命令和查询之间。遇到这种情况的一个场景是使用滚动升级模式来部署新服务。

为了更紧密地适应忽略未知字段的愿望，XStreamSerializer和JacksonSerializer都可以这样启用。如何实现这一点显示在以下代码片段:

public class SerializerConfiguration {

public Serializer buildSerializer() {

return JacksonSerializer.builder()

.lenientDeserialization()

.build();

}

}

Query Processing

将查询作为专用类型的消息处理与遵循cqr时隔离查询模型是一致的。尽管创建查询处理层相当简单，但在应用程序的这一部分中使用Axon框架有很多好处。

通过提供描述查询处理方法的功能(本节将进一步解释)和用于查询消息的专用总线，可以使用常见的消息特性，如拦截器和消息监视。

下一节将概述与配置必要的组件以开始处理Axon应用程序中的查询相关的任务。为此，我们讨论了注册@QueryHandler注释方法的方法，以及在分派查询时提供哪些选项。

**查询类型**

Axon框架区分了三种查询类型，即(1)点到点查询、(2)分散聚集查询和(3)订阅查询。

单击这些链接以获得有关如何实现每种查询类型的详细信息。

**Query Dispatchers**

如何处理查询消息已在查询处理一节中详细介绍。就像任何类型的消息一样，查询在被处理之前必须被分派。为此轴突提供了两个接口:

1 The Query Bus, and 2 The Query Gateway

本页将显示如何以及何时使用查询网关和总线。如何配置和查询网关和总线实现的细节在这里讨论

**查询总线和查询网关**

QueryBus是将查询分派给查询处理程序的机制。使用查询请求名称和查询响应类型的组合注册查询。可以为相同的请求-响应组合注册多个处理程序，这些处理程序可用于实现分散-聚集模式。在分发查询时，客户机必须指示它是想要单个处理程序的响应，还是想要所有处理程序的响应。

QueryGateway是查询调度机制的一个方便接口。虽然您不需要使用网关来分派查询，但这通常是最简单的选择。它为您抽象了某些方面，比如在查询消息中包装查询有效负载的必要性。

无论您选择使用QueryBus还是QueryGateway，它们都提供了几种类型的查询。轴突框架区分了三种类型，即:

1 点对点查询, 2 散集查询, 3 订阅查询

**1 点对点查询**

直接查询表示对单个查询处理程序的查询请求。如果没有找到给定查询的处理程序，则抛出NoHandlerForQueryException。如果注册了多个处理程序，则由查询总线的实现决定实际调用哪个处理程序。在下面的清单中，我们有一个简单的查询处理程序:

@QueryHandler // 1.

public List<String> query(String criteria) {

// return the query result based on given criteria

}

默认情况下，查询的名称是查询负载的完全限定类名(java.lang. class)。在本例中为字符串)。

但是，可以通过声明@QueryHandler注释的queryName属性来覆盖此行为。

如果我们想要查询我们的视图模型，List<String>，我们会这样做:

// 1.

GenericQueryMessage<String, List<String>> query =

new GenericQueryMessage<>("criteria", ResponseTypes.multipleInstancesOf(String.class));

// 2. send a query message and print query response

queryBus.query(query).thenAccept(System.out::println);

1 在构建查询消息时，还可以声明查询名称，

默认情况下，这是查询有效负载的完全限定类名。

2 发送查询的响应是Java CompletableFuture，

这取决于查询总线的类型，可以立即解析。

但是，如果@QueryHandler注释函数的返回类型是CompletableFuture，

无论查询总线的类型是什么，结果都将异步返回。

**2 散集查询**

当您希望来自与您的查询消息匹配的所有查询处理程序的响应时，应该使用分散收集查询的类型。作为对该查询的响应，将返回一个结果流。此流包含成功处理查询的每个处理程序的结果，顺序未指定。如果查询没有处理程序，或者所有处理程序在处理请求时抛出异常，则流为空。

在下面的清单中，我们有两个查询处理程序:

@QueryHandler(queryName = "query")

public List<String> query1(String criteria) {

// return the query result based on given criteria

}

@QueryHandler(queryName = "query")

public List<String> query2(String criteria) {

// return the query result based on given criteria

}

这些查询处理程序可能位于不同的组件中，我们希望同时从这两个组件中获得结果。因此，我们将使用散聚查询，如下所示:

// create a query message

GenericQueryMessage<String, List<String>> query =

new GenericQueryMessage<>("criteria", "query", ResponseTypes.multipleInstancesOf(String.class));

// send a query message and print query response

queryBus.scatterGather(query, 10, TimeUnit.SECONDS)

.map(Message::getPayload)

.flatMap(Collection::stream)

.forEach(System.out::println);

**订阅查询**

订阅查询允许客户机获得它想要查询的模型的初始状态，并在查询的视图模型更改时保持最新状态。简而言之，它是对直接查询的调用，在初始状态改变时可能会更新。要使用对模型的更改来更新订阅，我们将使用Axon提供的QueryUpdateEmitter组件。

让我

们来看看CardSummaryProjection的一个片段:

@QueryHandler

public List<CardSummary> handle(FetchCardSummariesQuery query) {

log.trace("handling {}", query);

TypedQuery<CardSummary> jpaQuery = entityManager.createNamedQuery("CardSummary.fetch", CardSummary.class);

jpaQuery.setParameter("idStartsWith", query.getFilter().getIdStartsWith());

jpaQuery.setFirstResult(query.getOffset());

jpaQuery.setMaxResults(query.getLimit());

return log.exit(jpaQuery.getResultList());

}

这个查询处理程序将向我们提供GiftCard状态列表。一旦我们的GiftCard得到兑换，我们想更新任何对GiftCard更新状态感兴趣的组件。我们将通过使用RedeemedEvt事件处理函数中的QueryUpdateEmitter组件发出更新来实现这一点:

@EventHandler

public void on(RedeemedEvt evt) {

// 1.

CardSummary summary = entityManager.find(CardSummary.class, event.getId());

summary.setRemainingValue(summary.getRemainingValue() - event.getAmount());

// 2.

queryUpdateEmitter.emit(FetchCardSummariesQuery.class,

query -> event.getId().startsWith(query.getFilter().getIdStartsWith()),

summary);

}

1 首先，我们通过更新现有的卡来更新视图模型。

2 如果有一个订阅查询对关于这个特定GiftCard的更新感兴趣，我们就发出一个更新。

发射的第一个参数是查询的类型(在本例中是FetchCardSummariesQuery)

它对应于前面定义的查询处理程序中的查询类型。

第二个参数是一个谓词，它将选择要更新的订阅查询。

在我们的情况下，我们将只更新订阅查询感兴趣的礼品卡已更新。

第三个参数是实际更新，在本例中是卡片汇总。

目前有几个emit方法的重载，您可以查看JavaDoc了解更多细节。

这里要强调的重要一点是，更新是一条消息，有些重载会发生

更新消息作为参数(在本例中，我们只是发送了包装在消息中的有效负载)

例如，我们可以附加元数据。

一旦我们实现了查询处理和发送端，我们就可以发出订阅查询来获取GiftCard的初始状态，并在该GiftCard被兑换后进行更新:

// 1.

commandGateway.sendAndWait(new IssueCmd("gc1", amount));

// 2.

FetchCardSummariesQuery fetchCardSummariesQuery =

new FetchCardSummariesQuery(offset, limit, filter);

// 3.

SubscriptionQueryResult<List<CardSummary>, CardSummary> fetchQueryResult =

queryGateway.subscriptionQuery(

fetchCardSummariesQuery,

ResponseTypes.multipleInstancesOf(CardSummary.class),

ResponseTypes.instanceOf(CardSummary.class));

fetchQueryResult

//4.

.handle(cs -> cs.forEach(System.out::println), System.out::println)

//5.

.doFinally(it -> fetchQueryResult.close());

// 6.

commandGateway.sendAndWait(new RedeemCmd("gc1", amount));

1 发行带有gc1 id和初始金额的礼品卡。

2 创建订阅查询消息以获取礼品卡列表

(初始状态是CardSummary的多个实例)

当id为gc1的GiftCard的状态发生变化时(在本例中，更新意味着兑换了卡片)，就会进行更新。

更新的类型是CardSummary的单个实例。

请注意，更新的类型必须与发射端类型匹配。

3 一旦创建了消息，我们将通过QueryGateway发送它。

我们收到一个包含两个组件的查询结果:一个是initialResult，另一个是updates。

为了达到“反应性”，我们使用Project Reactor的Mono来进行initialResult

以及更新流量。

请注意

一旦发出订阅查询，将排队等待所有更新，直到完成对更新通量的订阅。此行为可防止丢失更新。

请注意

该框架防止发出多个具有相同id的查询消息。如果需要在几个不同的地方更新，则创建一个新的查询消息。

请注意

对于订阅查询的使用，核反应堆依赖关系是强制性的。但是，它是一个编译时依赖项，其他Axon特性不需要它。

1 The SubscriptionQueryResult#handle(Consumer<? super I>, Consumer<? super U>)方法使我们能够一次订阅initialResult和更新。

如果我们想要对结果进行更细粒度的控制，我们可以对查询结果使用initialResult()和updates()方法。

因为即使没有订阅者，queryUpdateEmitter也将继续发出更新，所以当我们不再对接收更新感兴趣时，我们需要通知发送端。如果不这样做，可能会导致挂起不定式流，并最终导致内存泄漏。

一旦我们完成了使用订阅查询，我们需要关闭使用的资源。我们可以在doFinally hook中做到这一点。

作为doFinally钩子的替代方法，还有Flux#using API。这是同义的

到try-with-resource的Java API:

Flux.using( () -> fetchQueryResult,

queryResult -> queryResult.handle(..., ...),

SubscriptionQueryResult::close

);

3 当我们发出RedeemCmd时，投影中的事件处理程序最终会被触发，

这将导致更新的发送。

因为我们用println()方法订阅了更新，所以一旦收到更新，就会打印出来。

Query Handlers

查询的处理归结为一个带注释的处理程序返回查询的响应。本章的目标是描述这样一个@QueryHandler注释的方法是什么样子的，以及描述调用顺序和响应类型选项。对于查询处理程序和QueryBus的配置，建议阅读configuration部分。

**编写查询处理程序**

在Axon中，一个对象可以声明许多查询处理程序方法，方法是用@QueryHandler注释它们。所讨论的对象就是您所说的查询处理程序或查询处理组件。对于查询处理程序方法，第一个声明的参数定义它将接收哪个查询消息对象。

以包含CardSummary查询模型的“礼品卡”域为例，我们可以假设存在一个用于获取单个CardSummary实例的查询消息。让我们定义查询消息的格式如下:

public class FetchCardSummaryQuery {

private final String cardSummaryId;

public FetchCardSummaryQuery(String cardSummaryId) {

this.cardSummaryId = cardSummaryId;

}

// omitted getters, equals/hashCode, toString functions

}

如图所示，我们有一个常规的POJO，它将根据cardSummaryId字段获取CardSummary。这个FetchCardSummaryQuery将被分派给一个将给定消息定义为其第一个声明参数的处理程序。处理程序可能包含在一个负责或访问问题中的CardSummary模型的对象中:

import org.axonframework.queryhandling.QueryHandler;

public class CardSummaryProjection {

private Map<String, CardSummary> cardSummaryStorage;

@QueryHandler // 1.

public CardSummary handle(FetchCardSummaryQuery query) { // 2.

return cardSummaryStorage.get(query.getCardSummaryId());

}

// omitted CardSummary event handlers which update the model

}

从上面的示例中，我们想强调两个关于编写查询处理程序的细节:

1 @QueryHandler注释，它将函数标记为查询处理程序方法。

2 所讨论的方法由返回类型CardSummary(称为查询响应类型)和FetchCardSummaryQuery(查询负载)定义。

存储查询模型

出于本例的目的，我们选择使用常规映射作为存储方法。在现实的系统中，这将被数据库或存储库层的一种形式所取代。

**查询处理程序呼叫顺序**

在所有情况下，每个查询处理实例最多调用一个查询处理程序方法。Axon将使用以下规则搜索要调用的最特定的方法:

1 在类层次结构的实际实例级上(由this.getClass()返回)，所有带注释的方法都会被求值

2 如果找到一个或多个方法，其中所有参数都可以解析为一个值，则选择并调用具有最特定类型的方法

3 如果在类层次结构的这一层没有找到方法，则以同样的方式计算超类型

4 当到达层次结构的顶层时，如果没有找到合适的查询处理程序，则忽略此查询处理实例。

请注意，与命令处理类似，而与事件处理不同，查询处理不考虑查询消息的类层次结构。

// 假设QueryB扩展了QueryA

// and QueryC extends QueryB

// and 注册了一个SubHandler的实例

public class QueryHandler {

@QueryHandler

public MyResult handle(QueryA query) { // Return result }

@QueryHandler

public MyResult handle(QueryB query) { // Return result }

@QueryHandler

public MyResult handle(QueryC query) { // Return result}

}

public class SubQueryHandler extends QueryHandler {

@QueryHandler

public MyResult handleEx(QueryB query) { // Return result }

}

在上面的示例中，对于QueryB和result MyResult的查询将调用SubQueryHandler的处理程序方法。

**查询处理程序返回值**

如本页前面定义的，Axon允许查询处理程序方法有多种返回类型。您应该考虑单个对象和对象集合，还应该考虑通配符或泛型。下面我们分享了框架中支持和测试的所有选项列表。

为了清晰起见，我们在响应类型的单个实例和多个实例之间做了偏移。这符合在分派查询时指定ResponseType的要求，该要求用户声明是需要一个结果还是需要多个结果。Axon将使用这个ResponseType对象来匹配查询与查询处理程序方法，以及查询负载和查询名称。

**支持单实例返回值**

要查询单个对象，应该使用ReponseTypes#instanceOf(Class)方法来创建所需的ResponseType对象。这个“类的实例”ResponseType对象反过来支持以下查询处理程序返回值:

An exact match of Class

A subtype of Class

A generic bound to Class

A Future of Class

A primitive of Class

An Optional of Class

**原始的返回类型**

在常见的对象中，查询也可以返回基本数据类型:

public class QueryHandler {

@QueryHandler

public float handle(QueryA query) { }

}

注意，查询方将检索一个框起来的结果，而不是原始类型。

**支持多个实例返回值**

要查询多个对象，应该使用ReponseTypes#multipleInstancesOf(Class)方法来创建所需的ResponseType对象。这个“多实例类”ResponseType对象反过来支持以下查询处理程序返回值:

**一个数组,其中包含:**

Class

A subtype of Class

A generic bound to Class

可迭代对象或可迭代对象的自定义实现包含:

Class

A subtype Class

A generic bound to Class

A wildcard bound to Class

A Stream of Class

A Future of an Iterable of Class

**不支持的返回值**

以下列表包含了查询时不支持的方法返回值:

1 基本类型的数组

2 给定键和值类型的映射

实现

至于分派查询，如分派查询一节中所述，在实际发送相关的查询消息时，有两种实现。下一节将概述可能的实现，并指出如何使用Axon设置查询调度基础设施。

**查询网关**

查询网关是查询调度机制的一个方便接口。虽然您不需要使用网关来分派查询，但这通常是最简单的选择。

Axon提供了一个QueryGateway接口和DefaultQueryGateway实现。查询网关提供了许多方法，允许您发送查询并以同步、超时或异步方式等待单个或多个结果。查询网关需要配置访问查询总线和QueryDispatchInterceptors的列表(可能为空)。

**Query Bus**

查询总线是将查询分派给查询处理程序的机制。使用查询请求名称和查询响应类型的组合注册查询。可以为相同的请求-响应组合注册多个处理程序，这些处理程序可用于实现分散-聚集模式。在分发查询时，客户机必须指示它是想要单个处理程序的响应，还是想要所有处理程序的响应。

**AxonServerQueryBus**

Axon提供了开箱即用的查询总线AxonServerQueryBus。它连接到AxonIQ AxonServer服务器以发送和接收查询。

AxonServerQueryBus是一个“分布式查询总线”。默认情况下，它使用SimpleQueryBus来处理不同JVM上的传入查询。

通过简单地声明对Axon -spring-boot-starter的依赖，Axon将自动配置Axon服务器查询总线:

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

<dependency>

<groupId>org.axonframework</groupId>

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

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

</dependency>

如果您排除了axon-server-connector依赖项，您将退回到“non-axon-server”查询总线选项，即SimpleQueryBus(参见下面)

**SimpleQueryBus**

SimpleQueryBus在分派查询的线程中直接处理查询。配置一个SimpleQueryBus(而不是AxonServerQueryBus):

@Bean

public SimpleQueryBus queryBus(AxonConfiguration axonConfiguration,

TransactionManager transactionManager) {

return SimpleQueryBus.builder()

.messageMonitor(axonConfiguration.messageMonitor(QueryBus.class, "queryBus"))

.transactionManager(transactionManager)

.errorHandler(axonConfiguration.getComponent(

QueryInvocationErrorHandler.class,

() -> LoggingQueryInvocationErrorHandler.builder().build()

))

.queryUpdateEmitter(axonConfiguration.getComponent(QueryUpdateEmitter.class))

.build();

}

Configuration

此页描述了配置查询处理程序的过程。注意，查询处理程序是一个(单例)对象，包含@QueryHandler注释函数。

**注册查询处理程序**

当您注册一个查询处理程序时，这意味着您正在注册一个包含带注释的查询处理程序的类。在配置过程中接收到这样一个类后，Axon将扫描其内容以找到所有@QueryHandler注释的方法。在注册过程中，以下信息定义了一个给定的查询处理函数:

1 方法的第一个参数是查询负载。

2 方法响应类型是查询的响应类型。

3 注释中的queryName字段的值作为查询的名称(这是可选的，如果没有它，则默认为查询有效负载)。

注意，可以为相同的查询有效负载、响应类型和名称注册多个查询处理程序。此外，在分派查询时，客户机可以指示他/她是想要单个处理程序的结果，还是想要与查询负载、名称和响应类型组合对应的所有处理程序的结果。

以下代码片段指出了如何注册查询处理程序:

假定存在以下查询处理程序:

public class CardSummaryProjection {

@QueryHandler

public CardSummary handle(FetchCardSummaryQuery query) {

CardSummary cardSummary;

// Retrieve CardSummary instance, for example from a repository.

return cardSummary;

}

}

注册一个CardSummaryProjection作为一个查询处理程序需要以下代码:

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

.registerQueryHandler(conf -> new CardSummaryProjection());

或者，可以使用一种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 CardSummaryProjection());

当使用Spring Boot时，只需将查询处理程序指定为bean就足够了:

@Component

public class CardSummaryProjection {

@QueryHandler

public CardSummary handle(FetchCardSummaryQuery query) {

CardSummary cardSummary;

// Retrieve CardSummary instance, for example from a repository.

return cardSummary;

}

}

单个查询处理程序中的相同查询处理方法

一个查询处理程序类目前可以包含几个相同的查询处理方法。然而，实际调用哪个方法的结果是未知的。

请注意，这应该被视为一种非常少见的场景，因为通常相同的查询处理方法会分散在几个查询处理程序上。

Sagas

Axon框架为长期运行的业务事务或传奇提供了一流的支持。参考指南的这一部分将详细介绍Axon框架提供的帮助促进Saga开发的功能‌。

下面给出了各个小节的摘要。

1 实现 使用Axon框架实现saga

2 协会 使用Axon框架管理Saga关联

3 基础设施 使用Axon框架进行传奇的非功能性开发

1 实现

Saga是一种特殊类型的事件监听器:它管理业务事务。一些事务可能运行几天甚至几周，而其他事务在几毫秒内就完成了。在Axon中，Saga的每个实例负责管理单个业务事务。这意味着Saga维护管理该事务所必需的状态，继续执行该事务或采取补偿操作来回滚任何已经采取的操作。通常情况下，与常规的事件监听器相反，一个传奇有起点和终点，都是由事件触发的。虽然长篇故事的起点通常都很清楚，但它的结局可能有多种方式。

在Axon中，saga是定义一个或多个@SagaEventHandler方法的类。与常规事件处理程序不同，一个saga的多个实例可能随时存在。saga由单个事件处理器(跟踪或订阅)管理，该事件处理器专门处理特定saga类型的事件。

Life cycle

单个Saga实例负责管理单个事务。这意味着你需要能够指明一个传奇生命周期的开始和结束。

在一个saga中，事件处理程序是用@SagaEventHandler注释的。如果特定的事件表示事务的开始，则向同一方法添加另一个注释:@StartSaga。此注释将创建一个新的saga，并在发布匹配的事件时调用其事件处理程序方法。

默认情况下，只有在没有合适的现有saga(相同类型)时，才会开始一个新的saga。您还可以通过将@StartSaga注释上的forcennew属性设置为true来强制创建一个新的saga实例。

终结一场传奇有两种方法。如果一个特定的事件总是指示一个传奇的生命周期的结束，那么用@EndSaga注释这个事件处理程序。它的生命周期将在调用处理程序后结束。或者，您可以从saga中调用sagalifcycle .end()来结束生命周期。这可以让你有条件地结束这个传奇。

事件处理

saga中的事件处理与常规事件侦听器相当。这里也适用相同的方法和参数解析规则。不过，有一个主要区别。虽然只有一个事件监听器实例来处理所有传入事件，但一个saga可能存在多个实例，每个实例对不同的事件感兴趣。例如，一个围绕Id为“1”的订单管理事务的saga将不会对有关订单“2”的事件感兴趣，反之亦然。

Axon不是将所有事件发布给所有saga实例(这将完全浪费资源)，而是只发布包含与saga相关属性的事件。这是通过使用AssociationValues完成的。AssociationValue由键和值组成。键表示所使用的标识符类型，例如“orderId”或“order”。该值表示对应的值，即上例中的“1”或“2”。

@SagaEventHandler带注释的方法的计算顺序与@EventHandler方法的计算顺序相同(请参阅带注释的事件处理程序)。如果处理程序方法的参数与传入事件匹配，并且saga与处理程序方法上定义的属性有关联，那么方法就匹配。

@SagaEventHandler注释有两个属性，其中associationProperty是最重要的一个。这是传入事件上的属性名称，应用于查找关联的saga。关联值的键是属性的名称。该值是属性及其getter方法返回的值。

例如，考虑一个带有方法字符串getOrderId()的传入事件，该方法返回"123"。如果接受此事件的方法用@SagaEventHandler注解(associationProperty="orderId")，则此事件将被路由到所有与键为"orderId"和值为"123"的AssociationValue相关联的Sagas。这可能是1，可能多于1，甚至根本没有。

有时，您想要关联的属性的名称并不是您想要使用的关联的名称。例如，您有一个匹配“卖出订单”和“买入订单”的Saga，您可以有一个包含“buyOrderId”和“sellOrderId”的事务对象。如果您希望saga将“sellOrderId”值关联为“orderId”，您可以在@SagaEventHandler注释中定义一个不同的keyName。然后它会变成@SagaEventHandler(associationProperty="sellOrderId"， keyName="orderId")

注入资源

传奇通常不仅仅是维护基于事件的状态。它们与外部组件交互。为此，他们需要访问必要的资源来处理组件通常情况下，这些资源并不是游戏的一部分，这些资源也不应该被保存下来。然而，一旦重构了saga，这些资源必须在事件路由到该实例之前被注入。

为了达到这个目的，有一个源注入器。它被SagaRepository用来向saga注入资源。Axon提供了一个SpringResourceInjector，它用来自应用上下文的资源注入带注释的字段和方法。Axon还提供了一个SimpleResourceInjector，它将已经注册的资源注入到@Inject带注释的方法和字段中。

提示

由于资源不应该与saga一起持久化，所以请确保将transient关键字添加到这些字段中。这将防止序列化机制试图将这些字段的内容写入存储库。在一个saga被反序列化之后，存储库将自动重新注入所需的资源。

SimpleResourceInjector允许注入预先指定的资源集合。它扫描Saga的(setter)方法和字段，以找到带@Inject注释的方法和字段。

当使用Configuration API时，Axon将默认使用ConfigurationResourceInjector。它将注入配置中任何可用的资源。组件像EventBus, EventStore, CommandBus和CommandGateway是默认可用的。你也可以使用configuration . registercomponent()注册你自己的组件。

SpringResourceInjector使用Spring的依赖注入机制将资源注入到Saga中。这意味着，如果需要，您可以使用setter注入或直接字段注入。要注入的方法或字段需要注释，以便Spring将其识别为依赖项，例如使用@Autowired。

**Saga Infrastructure**

事件需要重定向到适当的saga实例。为此，需要一些基础设施类。最重要的组件是SagaManager和SagaRepository。

**Saga Manager**

与处理事件的任何组件一样，处理由事件处理器完成。然而，传奇并不是处理事件的单例实例。它们有需要管理的个体生命周期。

Axon通过带注释的sagamanager支持生命周期管理，它提供给事件处理程序来执行处理程序的实际调用。它使用要管理的saga类型以及一个可以存储和检索该类型saga的SagaRepository进行初始化。一个注释的sagamanager只能管理一个saga类型。

当使用Configuration API时，Axon将为大多数组件使用合理的默认值。但是，强烈建议定义一个SagaStore实现来使用。SagaStore是一种“物理地”存储saga实例的机制。注释的sagarepository(默认值)使用SagaStore来存储和检索需要的Saga实例。

@Saga(sagaStore = "mySagaStore")

public class MySaga {...}

...

// somewhere in configuration

@Bean

public SagaStore mySagaStore() {

return new MongoSagaStore(...); // default is JpaSagaStore

}

Saga repository and saga store

SagaRepository负责存储和检索SagaManager使用的传奇。它能够通过标识符以及关联值来检索特定的saga实例。

然而，这里有一些特殊的要求。由于sagas中的并发处理是一个非常微妙的过程，存储库必须确保对于每个概念上的saga实例(具有相同的标识符)，JVM中只存在一个实例。

Axon提供了带注释的sagarepository实现，它允许查找saga实例，同时保证同一时间只能访问saga的单个实例。它使用一个SagaStore来执行saga实例的实际持久性。

要使用的实现的选择主要取决于应用程序使用的存储引擎。Axon提供了JdbcSagaStore、InMemorySagaStore、JpaSagaStore和MongoSagaStore。

在某些情况下，应用程序可以从缓存saga实例中获益。在这种情况下，有一个CachingSagaStore，它包装了另一个实现来添加缓存行为。注意，CachingSagaStore是一个透写缓存，这意味着保存操作总是立即转发到后备存储，以确保数据安全。

**JpaSagaStore**

JpaSagaStore使用JPA来存储saga的状态和关联值。传奇本身不需要任何JPA注释;Axon将使用序列化器序列化saga(类似于事件序列化，您可以在XStreamSerializer、JacksonSerializer或JavaSerializer之间进行选择，可以通过在应用程序中配置默认的序列化器来设置。有关更多细节，请参见序列化器。

JpaSagaStore配置了EntityManagerProvider，它提供了对要使用的EntityManager实例的访问。这种抽象允许同时使用应用程序管理和容器管理的实体管理器。您还可以选择定义用于序列化Saga实例的序列化器。Axon默认使用XStreamSerializer。

**JdbcSagaStore**

JdbcSagaStore使用纯JDBC存储stage实例及其关联值。与JpaSagaStore类似，saga实例不需要知道它们的存储方式。使用序列化器对它们进行序列化。

JdbcSagaStore使用数据源或ConnectionProvider进行初始化。虽然不是必需的，但在使用ConnectionProvider进行初始化时，建议将实现包装在UnitOfWorkAwareConnectionProviderWrapper中。它将检查已经打开的数据库连接的当前工作单元，以确保工作单元中的所有活动都是在单个连接上完成的。

与JPA不同，JdbcSagaRepository使用纯SQL语句存储和检索信息。这可能意味着某些操作取决于特定于数据库的SQL方言。也有可能某些数据库供应商提供了您想要使用的非标准特性。为此，您可以提供自己的SagaSqlSchema。SagaSqlSchema是一个定义存储库需要在底层数据库上执行的所有操作的接口。它允许您定制为每个操作执行的SQL语句。默认是GenericSagaSqlSchema。其他可用的实现有PostgresSagaSqlSchema, Oracle11SagaSqlSchema和HsqlSagaSchema。

**MongoSagaStore**

MongoSagaStore将saga实例及其关联存储在MongoDB数据库中。mongsagastore将所有的传奇存储在MongoDB数据库的一个集合中。对于每个saga实例，将创建单个文档。

MongoSagaStore还确保在任何时候，单个JVM中只有一个Saga实例存在。这可以确保不会因为并发性问题而丢失状态更改。

monsagastore使用MongoTemplate和可选的序列化程序进行初始化。MongoTemplate提供了一个用于存储传奇的集合的参考。Axon提供了DefaultMongoTemplate，它接受一个MongoClient实例以及用于存储saga的数据库名称和集合名称。数据库名称和集合名称可以省略。在这种情况下，它们默认分别为“axonframework”和“sagas”。

**缓存**

如果使用了数据库支持的saga存储，保存和加载saga实例可能是一个相对昂贵的操作。在同一saga实例在短时间内被多次调用的情况下，缓存对应用程序的性能特别有利。

Axon提供了CachingSagaStore实现。它是一个sagstore包装另一个，它做实际的存储。当加载saga或关联值时，CachingSagaStore将首先查询它的缓存，然后再委托给包装的存储库。在存储信息时，总是委托所有调用，以确保后备存储对saga的状态始终具有一致的视图。

要配置缓存，只需将任何SagaStore打包到CachingSagaStore中。CachingSagaStore的构造函数有三个参数:2. The sagstore to wrap用于关联值3的缓存。用于saga实例的缓存

后两个参数可以指向同一个缓存，也可以指向不同的缓存。这取决于特定应用程序的驱逐要求。

Associations

当一个saga管理跨多个领域概念(如订单、发货、发票等)的事务时，该saga需要与这些概念的实例相关联。一个关联需要两个参数:一个键，它标识关联的类型(订单、发货等)，一个值，它表示该概念的标识符。

将传奇与概念联系在一起可以通过多种方式实现。首先，当调用@StartSaga带注释的事件处理程序时新创建Saga时，它会自动与@SagaEventHandler方法中标识的属性相关联任何其他的关联都可以使用SagaLifecycle创建。关联(字符串键，字符串/数字值)方法。使用SagaLifecycle。方法移除一个特定的关联。

请注意

在Saga中关联域概念的API故意只允许字符串或数字作为标识值，因为存储的关联值条目需要标识符的字符串表示。在API中使用简单的标识符值和简单的字符串表示是设计好的，因为数据库中的字符串列条目使数据库引擎之间的比较更简单。例如，故意不使用associateWith(String, Object)，因为object# toString()调用的结果可能会提供笨拙的标识符。

想象一下，一个围绕订单的交易创建了一个传奇。saga自动与订单关联，因为该方法用@StartSaga注释。saga负责为该订单创建发票，并告诉发货为该订单创建发货。一旦货物到达并支付了发票，交易就完成了，传奇也就结束了

这是这样一个传奇故事的代码:

public class OrderManagementSaga {

private boolean paid = false;

private boolean delivered = false;

@Inject

private transient CommandGateway commandGateway;

@StartSaga

@SagaEventHandler(associationProperty = "orderId")

public void handle(OrderCreatedEvent event) {

// client generated identifiers

ShippingId shipmentId = createShipmentId();

InvoiceId invoiceId = createInvoiceId();

// associate the Saga with these values, before sending the commands

SagaLifecycle.associateWith("shipmentId", shipmentId);

SagaLifecycle.associateWith("invoiceId", invoiceId);

// send the commands

commandGateway.send(new PrepareShippingCommand(...));

commandGateway.send(new CreateInvoiceCommand(...));

}

@SagaEventHandler(associationProperty = "shipmentId")

public void handle(ShippingArrivedEvent event) {

delivered = true;

if (paid) { SagaLifecycle.end(); }

}

@SagaEventHandler(associationProperty = "invoiceId")

public void handle(InvoicePaidEvent event) {

paid = true;

if (delivered) { SagaLifecycle.end(); }

}

}

通过允许客户端生成标识符，saga可以很容易地与一个概念相关联，而不需要请求-响应类型命令。在发布命令之前，我们将事件与这些概念联系起来。这样，我们保证也可以捕获作为该命令一部分生成的事件。一旦发票支付完毕，货物到达，这个故事就结束了。

基础设施

事件需要重定向到适当的saga实例。为此，需要一些基础设施类。最重要的组件是SagaManager和SagaRepository。

**Saga Manager 事件管理器**

与处理事件的任何组件一样，处理由事件处理器完成。然而，传奇并不是处理事件的单例实例。它们有需要管理的个体生命周期。

Axon通过带注释的sagamanager支持生命周期管理，它提供给事件处理程序来执行处理程序的实际调用。它使用要管理的saga类型以及一个可以存储和检索该类型saga的SagaRepository进行初始化。一个注释的sagamanager只能管理一个saga类型。

当使用Configuration API时，Axon将为大多数组件使用合理的默认值。但是，强烈建议定义一个SagaStore实现来使用。SagaStore是一种“物理地”存储saga实例的机制。注释的sagarepository(默认值)使用SagaStore来存储和检索需要的Saga实例。

@Saga(sagaStore = "mySagaStore")

public class MySaga {...}

...

// somewhere in configuration

@Bean

public SagaStore mySagaStore() {

return new MongoSagaStore(...); // default is JpaSagaStore

}

**Saga存储库和Saga存储**

SagaRepository负责存储和检索SagaManager使用的传奇。它能够通过标识符以及关联值来检索特定的saga实例。

然而，这里有一些特殊的要求。由于sagas中的并发处理是一个非常微妙的过程，存储库必须确保对于每个概念上的saga实例(具有相同的标识符)，JVM中只存在一个实例。

Axon提供了带注释的sagarepository实现，它允许查找saga实例，同时保证同一时间只能访问saga的单个实例。它使用一个SagaStore来执行saga实例的实际持久性。

要使用的实现的选择主要取决于应用程序使用的存储引擎。Axon提供了JdbcSagaStore、InMemorySagaStore、JpaSagaStore和MongoSagaStore。

在某些情况下，应用程序可以从缓存saga实例中获益。在这种情况下，有一个CachingSagaStore，它包装了另一个实现来添加缓存行为。注意，CachingSagaStore是一个透写缓存，这意味着保存操作总是立即转发到后备存储，以确保数据安全。

**JpaSagaStore**

JpaSagaStore使用JPA来存储saga的状态和关联值。传奇本身不需要任何JPA注释;Axon将使用序列化器序列化saga(类似于事件序列化，您可以在XStreamSerializer、JacksonSerializer或JavaSerializer之间进行选择，可以通过在应用程序中配置默认的序列化器来设置。有关更多细节，请参见序列化器。

JpaSagaStore配置了EntityManagerProvider，它提供了对要使用的EntityManager实例的访问。这种抽象允许同时使用应用程序管理和容器管理的实体管理器。您还可以选择定义用于序列化Saga实例的序列化器。Axon默认使用XStreamSerializer。

**JdbcSagaStore**

JdbcSagaStore使用纯JDBC存储stage实例及其关联值。与JpaSagaStore类似，saga实例不需要知道它们的存储方式。使用序列化器对它们进行序列化。

JdbcSagaStore使用数据源或ConnectionProvider进行初始化。虽然不是必需的，但在使用ConnectionProvider进行初始化时，建议将实现包装在UnitOfWorkAwareConnectionProviderWrapper中。它将检查已经打开的数据库连接的当前工作单元，以确保工作单元中的所有活动都是在单个连接上完成的。

与JPA不同，JdbcSagaRepository使用纯SQL语句存储和检索信息。这可能意味着某些操作取决于特定于数据库的SQL方言。也有可能某些数据库供应商提供了您想要使用的非标准特性。为此，您可以提供自己的SagaSqlSchema。SagaSqlSchema是一个定义存储库需要在底层数据库上执行的所有操作的接口。它允许您定制为每个操作执行的SQL语句。默认是GenericSagaSqlSchema。其他可用的实现有PostgresSagaSqlSchema, Oracle11SagaSqlSchema和HsqlSagaSchema。

**MongoSagaStore**

MongoSagaStore将saga实例及其关联存储在MongoDB数据库中。mongsagastore将所有的传奇存储在MongoDB数据库的一个集合中。对于每个saga实例，将创建单个文档。

MongoSagaStore还确保在任何时候，单个JVM中只有一个Saga实例存在。这可以确保不会因为并发性问题而丢失状态更改。

monsagastore使用MongoTemplate和可选的序列化程序进行初始化。MongoTemplate提供了一个用于存储传奇的集合的参考。Axon提供了DefaultMongoTemplate，它接受一个MongoClient实例以及用于存储saga的数据库名称和集合名称。数据库名称和集合名称可以省略。在这种情况下，它们默认分别为“axonframework”和“sagas”。

**缓存**

如果使用了数据库支持的saga存储，保存和加载saga实例可能是一个相对昂贵的操作。在同一saga实例在短时间内被多次调用的情况下，缓存对应用程序的性能特别有利。

Axon提供了CachingSagaStore实现。它是一个sagstore包装另一个，它做实际的存储。当加载saga或关联值时，CachingSagaStore将首先查询它的缓存，然后再委托给包装的存储库。在存储信息时，总是委托所有调用，以确保后备存储对saga的状态始终具有一致的视图。

要配置缓存，只需将任何SagaStore打包到CachingSagaStore中。CachingSagaStore的构造函数有三个参数:2. The sagstore to wrap用于关联值3的缓存。用于saga实例的缓存

后两个参数可以指向同一个缓存，也可以指向不同的缓存。这取决于特定应用程序的驱逐要求。

Deadlines

Axon框架中的“截止日期”概念是一种机制，能够在一定时间后执行特定的动作。这次执行的背景是一个汇总或传奇，其中的最后期限被安排。如果截止日期过时，也有可能取消它。‌

参考指南的这一部分将详细介绍Axon框架提供的帮助安排和处理截止日期的功能。‌

下面给出了各个小节的摘要。

经理最后期限 使用期限管理器处理期限

事件调度器 使用事件调度程序处理截止日期

**Deadline Managers**

可以通过saga和聚合来安排截止日期。DeadlineManager组件负责调度截止日期并在截止日期满足时调用@ deadlinehandler。DeadlineManager可以作为一个资源注入。它有两种风格:SimpleDeadlineManager和QuartzDeadlineManager

**Scheduling a Deadline**

可以通过提供一个将被触发的持续时间(或一个将被触发的瞬间)和一个截止日期名称来安排截止日期。

预定的事件或预定的截止日期

与事件调度不同，当触发截止日期时，将不会存储已发布的消息。调度/触发截止日期不涉及EventBus(或EventStore)，因此消息不会被存储。

class DeadlineSchedulingComponent {

void scheduleMyDeadline() {

String deadlineId =

deadlineManager.schedule(Duration.ofMillis(500), "myDeadline");

// For example store the `deadlineId`

}

}

因此，我们会收到一个deadlineId，它可以用来取消截止日期。在大多数情况下，将这个deadlineId存储为聚合/Saga中的一个字段是最方便的。当某一事件意味着先前计划的截止日期已经过时时，取消一个截止日期就可以派上用场了(例如，有一个支付发票的截止日期，但客户支付了金额，这意味着截止日期已经过时，可以取消)。

class DeadlineCancelingComponent {

void cancelMyDeadline(String deadlineId) {

deadlineManager.cancelSchedule("myDeadline", deadlineId);

}

}

注意，除了前面提到的，还有更多的选项可以取消截止日期:

1 cancelAll(String deadlineName) 取消与给定截止日期名匹配的每个预定截止日期。

注意，这样也会取消与名称匹配的其他聚合和/或saga实例的截止日期。

2 cancelAllWithinScope(String deadlineName)

在调用方法的范围内取消与给定的截止日期匹配的预定截止日期。例如，如果该操作在“聚合实例X”内执行，则将使用“聚合实例X”中的作用域描述符取消。

cancelAllWithinScope(String deadlineName, ScopeDescriptor作用域)

取消与给定的截止日期名和作用域描述符匹配的预定截止日期。这允许从不同的作用域通过名称取消最后期限。

如果你需要关于截止日期的上下文数据当截止日期被处理时，你可以附加一个截止日期有效载荷当调度一个截止日期:

class DeadlineSchedulingWithPayloadComponent {

void scheduleMyDeadlineWithPayload() {

String deadlineId = deadlineManager.schedule(

Duration.ofMillis(500), "myDeadline",

new MyDeadlinePayload(/\* some user specific parameters \*/)

);

// For example store the `deadlineId`

}

}

**Handling a Deadline**

我们现在已经了解了如何安排截止日期。当满足预定时间时，将调用对应的@DeadlineHandler。@DeadlineHandler是一个消息处理程序，与Axon中的任何其他处理程序一样——它可以注入存在参数解析器的参数。

期限的范围

在安排截止日期时，要考虑到它被安排的上下文。这意味着计划的截止日期只会在其初始上下文中被触发。因此，任何你希望在met截止日期被调用的@DeadlineHandler注释函数，都必须在被调度的同一聚合/Saga中。

Axon将此上下文称为范围。如果有必要，实现和提供你自己的范围将允许你在你的自定义，“范围”的组件时间表截止日期。

@DeadlineHandler根据截止日期名称和截止日期有效负载匹配。

@DeadlineHandler(deadlineName = "myDeadline")

public void on(MyDeadlinePayload deadlinePayload) {

// handle the deadline

}

如果在@DeadlineHandler中没有定义截止日期的名称，匹配将只根据截止日期有效载荷进行。

@DeadlineHandler

public void on(MyDeadlinePayload deadlinePayload) {

// handle the deadline

}

如果我们计划了一个没有特定负载的截止日期，@DeadlineHandler就不必指定负载。

@DeadlineHandler(deadlineName = "payloadlessDeadline")

public void on() {

// handle the deadline

}

**Using Time In Your Application**

在应用程序需要访问时钟的情况下，它们可以通过访问GenericEventMessage.clock来利用EventMessage中使用的时钟。这个钟是固定的。在运行时使用systemUTC，并在测试期间通过操纵来模拟时间。

public void handle(PublishTime cmd) {

apply(new TimePublishedEvent(GenericEventMessage.clock.instant()));

}

注意，当前时间戳会自动添加到EventMessage中。如果处理程序只需要依赖事件发布的时间戳，它们可以直接访问该时间戳，如处理事件中所述。

事件调度器

本节将继续使用EventScheduler处理截止日期时的建议行动。

为了更好地理解这一点，让我们以一个传奇的场景为例:当事情发生时，让一个传奇采取行动很容易。毕竟，有一个事件可以通知整个事件。但如果你想让你的传奇在什么都没发生的时候做点什么呢?这就是截止日期的作用。对于发票，这通常是几个星期，而信用卡付款的确认应该在几秒钟内发生。

Scheduled Events as Deadlines

在Axon中，可以使用EventScheduler来安排发布事件。以发票为例，你应该希望发票在30天内支付。一个saga将在发送createinvoiccommand之后，安排一个在30天内发布的InvoicePaymentDeadlineExpiredEvent。EventScheduler在调度一个事件后返回一个ScheduleToken。该令牌可用于取消计划，例如，当收到发票的付款时。

Axon提供了三种EventScheduler实现:

1 Pure Java 2 Quartz based 3 Axon Server based

EventScheduler的纯java实现使用ScheduledExecutorService来调度事件发布。尽管这个调度器的定时非常可靠，但它是一个纯粹的内存实现。一旦JVM关闭，所有的调度都会丢失。这使得这种实现不适合长期计划。SimpleEventScheduler需要配置一个EventBus和一个SchedulingExecutorService(请参阅java.util.concurrent上的静态方法。Executors类用于helper方法)。

QuartzEventScheduler是一个更可靠的企业级实现。使用Quartz作为底层调度机制，它提供了更强大的特性，如持久性、集群和失火管理。这意味着事件发布是有保证的。可能有点晚了，但一定会出版的。它需要配置石英调度器和EventBus。还可以设置调度Quartz作业的组的名称，默认为“AxonFramework-Events”

AxonServerEventScheduler使用Axon服务器来调度发布事件。因此，使用Axon服务器作为事件存储解决方案来利用这个事件调度器是很难的。与QuartzEventScheduler一样，AxonServerEventScheduler是EventScheduler接口的可靠和企业级实现。可以通过构建器创建AxonServerEventScheduler，该构建器的唯一需求是AxonServerConnectionManager。

重要的是要注意，QuartzEventScheduler和AxonServerEventScheduler都应该使用事件序列化程序来序列化和反序列化计划的事件。如果调度程序使用的序列化程序与事件存储程序使用的序列化程序不一致，则应该预期会出现异常情况。Quartz实现的序列化器可以通过定义不同的EventJobDataBinder来设置，而Axon服务器实现允许通过构建器直接定义所使用的序列化器。

一个或多个组件将侦听预定的事件。这些组件可能依赖于绑定到调用它们的线程的事务。调度的事件由EventScheduler管理的线程发布。要管理这些线程上的事务，您可以配置事务性管理器或UnitOfWorkFactory来创建事务绑定的工作单元。

**Spring Configuration**

Spring用户可以使用QuartzEventSchedulerFactoryBean或SimpleEventSchedulerFactoryBean来简化配置。它允许您直接设置PlatformTransactionManager。

依赖于Axon服务器的Spring Boot用户不需要定义任何东西。自动配置将为它们自动创建一个AxonServerEventScheduler。

Spring Boot Integration

Axon框架为Spring提供了广泛的支持，但不需要使用Spring才能使用Axon。所有组件都可以通过编程方式配置，不需要在类路径上使用Spring。但是，如果您确实使用Spring，那么使用Spring的注释支持可以简化很多配置。Axon在此基础上提供了Spring启动器，因此您也可以从自动配置中受益。

**Auto-configuration**

到目前为止，Axon的Spring Boot auto-configuration是开始配置Axon组件最简单的选项。通过简单地声明对Axon -spring-boot-starter的依赖，Axon将自动配置基础设施组件(命令总线、事件总线、查询总线)以及运行和存储聚合和saga所需的任何组件。

**JPA & Persistence Contexts**

使用Spring Data-JPA，可以自动配置JPA持久性上下文。Axon的Spring Boot Autoconfiguration模块将确保Axon的JPA实体自动注册到这个默认上下文中。

但是，当显式包含某些包时，例如使用@EntityScan注释，这种自动配置将不再发生。如果您希望使用来自Axon的基于JPA的组件，则需要确保注册了正确的实体。

要注册Axon的JPA实体，需要包含相关的包，如下所述:

1 org.axonframework.eventhandling。tokenstore包含事件使用的tokenstore所需的实体处理器。

2 org.axonframework.modelling.saga.repository.jpa 包含持久化传奇所需的实体

3 org.axonframework.eventsourcing.eventstore.jpa包含JPA事件存储引擎所需的实体，使用关系数据库作为事件存储时。

**Demystifying Axon Spring Boot Starter**

由于很多事情都是在后台进行的，有时很难理解一个注释或仅仅包含一个依赖项是如何支持这么多特性的。

axon-spring-boot-starter在构造启动器时遵循一般的Spring启动约定。它依赖于Axon -spring-boot-autoconfigure，它包含了Axon自动配置的具体实现。当Axon Spring引导应用程序启动时，它会查找一个名为Spring的文件。类路径中的工厂。这个文件位于axon-spring-boot-autoconfigure模块的META-INF目录中:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.axonframework.springboot.autoconfig.MetricsAutoConfiguration,\

org.axonframework.springboot.autoconfig.EventProcessingAutoConfiguration,\

org.axonframework.springboot.autoconfig.AxonAutoConfiguration,\

org.axonframework.springboot.autoconfig.JpaAutoConfiguration,\

org.axonframework.springboot.autoconfig.JpaEventStoreAutoConfiguration,\

org.axonframework.springboot.autoconfig.JdbcAutoConfiguration,\

org.axonframework.springboot.autoconfig.TransactionAutoConfiguration,\

org.axonframework.springboot.autoconfig.NoOpTransactionAutoConfiguration,\

org.axonframework.springboot.autoconfig.InfraConfiguration,\

org.axonframework.springboot.autoconfig.ObjectMapperAutoConfiguration,\

org.axonframework.springboot.autoconfig.AxonServerAutoConfiguration

这个文件映射了Axon Spring boot应用程序将尝试应用的不同配置类。因此，根据这段代码，Spring Boot将尝试应用AxonServerAutoConfiguration、AxonAutoConfiguration、…的所有配置类。

是否应用这些配置类，取决于这些类定义的条件:

1 AxonServerAutoConfiguration将Axon服务器配置为命令总线、查询总线和事件存储的实现。它将在AxonAutoConfiguration之前应用，并且只在org.axonframework.axonserver.connector. connector. org. AxonAutoConfiguration。AxonServerConfiguration类在类路径中可用。可以通过设置Axon .axonserver来禁用Axon服务器自动配置。在.properties/中启用属性为false。yml文件。

2 AxonAutoConfiguration配置命令总线、查询总线、事件存储/事件总线和其他Axon组件的“非Axon服务器”实现。这些组件只有在它们不在Spring应用程序上下文中时才会被初始化，例如。@ConditionalOnMissingBean (EventBus.class)。因为AxonAutoConfiguration将在AxonServerAutoConfiguration之后应用，这些Axon组件将已经在Spring应用程序上下文中，因此Axon服务器的命令总线、查询总线和事件存储/事件总线的实现将获胜。

Axon Spring引导自动配置是不侵入的。它将只定义您还没有在应用程序上下文中显式定义的Spring组件。这允许您通过在@Configuration类中定义自己的bean来完全覆盖自动配置的bean。

特定的轴突(Spring)组件配置将在本指南的以下章节中详细解释。

Reactor

忽略Axon框架架构，您可以注意到，在使用框架的一般系统中，有“消息驱动”、“响应”、“弹性”和“弹性”。根据反应性宣言，反应性系统通常也是如此。

虽然我们可以说轴突框架是一种反应性系统，但我们不能说它是完全反应性的。

响应式编程是一种编写包含异步I/O的软件的方法。异步I/O是一个小的想法，它预示着软件的重大变化。这个想法很简单:通过回收在等待I/O活动时闲置的资源来缓解低效的资源利用。异步I/O反转了正常的设计I/O处理:客户端收到新数据的通知，而不是请求它，这释放了客户端在等待这些通知的同时做其他工作。

就其本质而言，响应式API和Axon非常适合，因为大多数框架的操作都是异步和非阻塞的。因此，为此提供一个专用的扩展是一个合乎逻辑的步骤。为此，我们选择使用Pivotal的Project Reactor来构建这个扩展。Reactor构建在Reactive Streams规范的基础上，是Java enterprise和Spring应用程序的事实上的标准。因此，我们认为它非常适合提供扩展，使Axon更具反应性。

并不是所有的Axon组件都提供了响应式API。我们将逐步引入更多的“反应性”到这个扩展，优先给组件，用户可以受益最大。

要使用Axon Reactor扩展，请确保Axon - Reactor模块在类路径上可用。

**Reactor Gateways**

“响应式网关”提供了围绕命令、查询和事件总线的响应式API包装器。大多数操作与非无功网关的操作类似，只是简单地用单峰或通量替换完整的未来。在某些情况下，API被用于简化通用响应模式的使用。

反应器不允许流中有空值。处理程序返回的任何空值都将映射到Mono#empty()。进入翻译页面

**重新尝试操作**

所有操作都支持Reactor的重试机制:

reactiveQueryGateway。查询(查询,ResponseType.class) .retry (5);

当调用失败时，这个调用将重试发送查询最多5次。

Configuration in Spring Boot

这个扩展可以使用组id org.axonframework.extensions作为Spring启动器依赖添加到您的项目中。反应器和工件id轴突反应器-弹簧启动器。扩展的实现可以在这里找到。

**Reactor Command Gateway**

本节描述ReactorCommandGateway上的方法。

send——在调用者订阅命令结果后发送给定的命令。立即返回。

一个常见的模式是使用REST API发送命令。在这种情况下，建议使用例如WebFlux，并直接返回命令结果Mono到控制器:

class SpringCommandController {

private final ReactorCommandGateway reactiveCommandGateway;

@PostMapping

public Mono<CommandHandlerResponseBody> sendCommand(@RequestBody CommandBody command) {

return reactiveCommandGateway.send(command);

}

}

**从Spring WebFlux控制器发送命令。**

如果命令处理函数返回类型为void, Mono<CommandHandlerResponseBody>应该被替换为Mono< void >

另一个常见的模式是“send and forget”:

class CommandDispatcher {

private final ReactorCommandGateway reactiveCommandGateway;

public void sendAndForget(MyCommand command) {

reactiveCommandGateway.send(command)

.subscribe();

}

}

发送命令并立即返回而不等待结果的函数。

sendAll—此方法使用给定的命令发布者来分派传入的命令。

此操作仅在反应器扩展中可用。使用它连接发送命令的第三方流。

class CommandPublisher {

private final ReactorCommandGateway reactiveCommandGateway;

@PostConstruct

public void startReceivingCommands(Flux<CommandBody> inputStream) {

reactiveCommandGateway.sendAll(inputStream)

.subscribe();

}

}

将外部输入流直接连接到反应堆命令网关。

sendAll操作将一直发送命令，直到输入流被取消。

send和sendAll还没有提供任何反压力。唯一的反压机制是命令将按顺序发送;因此，一旦前一个命令的结果到达。命令的数量从传入流中预取，并存储在缓冲区中以供发送(参见Flux#concatMap)。这减慢了发送速度，但不能保证如果发送太快，订阅者不会被命令压垮

**Reactor Query Gateway**

query——发送给定的查询，期望从单个源获得responseType形式的响应。

class SpringQueryController {

private final ReactorQueryGateway reactiveQueryGateway;

// 查询的Mono返回给Spring控制器。订阅控制被赋予Spring框架。

@GetMapping

public Mono<SomeResponseType> findAll(FindAllQuery query, Class<SomeResponseType> responseType) {

return reactiveQueryGateway.query(query, responseType);

}

}

在Spring REST控制器中使用反应器查询网关的推荐方法。

scatterGather——发送给定的查询，期望在指定的持续时间内从多个源获得responseType形式的响应。

class SpringQueryController {

private final ReactorQueryGateway reactiveQueryGateway;

@GetMapping

public Flux<SomeResponseType> findMany(FindManyQuery query) {

return reactiveQueryGateway.scatterGather(query, SomeResponseType.class, Duration.ofSeconds(5)).take(3);

}

}

发送一个给定的查询，在收到三个结果后或5秒后停止。

**订阅查询**

首先，Axon中用于订阅查询的Reactor API并不新鲜。然而，我们注意到一些常用的模式，例如:将初始结果与单个流中的查询更新连接起来，或者跳过最初的结果。因此，Reactor扩展提供了几种方法来简化这些常见模式的使用。

subscriptionQuery -发送给定的查询，返回初始结果，并保持流增量更新，直到订阅者取消订阅流量。注意，当初始结果的响应类型与增量更新匹配时，应该使用此方法。

注意，当初始结果的响应类型与增量更新匹配时，应该使用此方法。

Flux<ResultType> resultFlux =

reactiveQueryGateway.subscriptionQuery("criteriaQuery", ResultType.class)

上面通过ReactorQueryGateway调用等价于:

class SubscriptionQuerySender {

private final ReactorQueryGateway reactiveQueryGateway;

public Flux<SomeResponseType> sendSubscriptionQuery(

SomeQuery query, Class<SomeResponseType> responseType) {

return reactiveQueryGateway.subscriptionQuery(

query, responseType, responseType)

.flatMapMany(result -> result.initialResult()

.concatWith(result.updates())

.doFinally(signal -> result.close()));

}

}

subscriptionQueryMany—发送给定的查询，返回初始结果，并保持流增量更新，直到订阅者取消订阅流量。

当初始结果包含需要扁平化的响应类型的多个实例时，应该使用此操作。此外，初始响应的响应类型需要与增量更新相匹配。

Flux<ResultType> resultFlux = reactiveQueryGateway.subscriptionQueryMany("criteriaQuery", ResultType.class);

上面通过ReactorQueryGateway调用等价于:

class SubscriptionQuerySender {

private final ReactorQueryGateway reactiveQueryGateway;

public Flux<SomeResponseType> sendSubscriptionQuery(

SomeQuery query, Class<SomeResponseType> responseType) {

return reactiveQueryGateway.subscriptionQuery(

query, ResponseTypes.multipleInstancesOf(responseType),

ResponseTypes.instanceOf(responseType))

.flatMapMany(result -> result.initialResult()

.flatMapMany(Flux::fromIterable)

.concatWith(result.updates())

.doFinally(signal -> result.close()));

}

}

queryUpdates—发送给定的查询并接收增量更新，直到订阅服务器取消订阅该流量。

当订阅者只对更新感兴趣时，可以使用此方法。

Flux<ResultType> updatesOnly =

reactiveQueryGateway.queryUpdates("criteriaQuery", ResultType.class);

上面通过ReactorQueryGateway调用等价于:

class SubscriptionQuerySender {

private final ReactorQueryGateway reactiveQueryGateway;

public Flux<SomeResponseType> sendSubscriptionQuery(

SomeQuery query, Class<SomeResponseType> responseType) {

return reactiveQueryGateway.subscriptionQuery(

query, ResponseTypes.instanceOf(Void.class), responseType)

.flatMapMany(result -> result.updates()

.doFinally(signal -> result.close()));

}

}

在上述方法中，当订阅服务器取消订阅流量后，订阅查询将自动关闭。但是，当使用常规的QueryGateway时，订阅查询需要手动关闭。

**Reactor Event Gateway**

事件网关的反应性变化。为响应式返回类型(如通量)提供支持。

发布——一旦调用方订阅了产生的流量，就发布给定的事件。

此方法返回已发布的事件。请注意，返回的事件可能与用户发布的事件不同，如果已经注册了修改事件的拦截器。

class EventPublisher {

private final ReactorEventGateway reactiveEventGateway;

// Register a dispatch interceptor to modify the event messages

public EventPublisher() {

reactiveEventGateway.registerDispatchInterceptor(

eventMono -> eventMono.map(event -> GenericEventMessage.asEventMessage(

"intercepted" + event.getPayload()))

);

}

public void publishEvent() {

Flux<Object> result = reactiveEventGateway.publish("event");

}

}

dispatcher修改事件的示例，作为结果流量返回给用户。

**Interceptors**

轴突提供了拦截器的概念。反应堆网关允许类似的拦截器逻辑，即ReactorMessageDispatchInterceptor和ReactorResultHandlerInterceptor。

这些拦截器允许我们集中定义将应用于消息流的规则和过滤器。

拦截器将按照它们已经注册到给定组件的顺序被应用。

**Reactor Dispatch Interceptors**

ReactorMessageDispatchInterceptor应该用于集中应用传出消息的规则和验证。注意，ReactorMessageDispatchInterceptor是整个框架中使用的缺省MessageDispatchInterceptor接口的实现。具体接口实现如下:

@FunctionalInterface

public interface ReactorMessageDispatchInterceptor<M extends Message<?>>

extends MessageDispatchInterceptor<M> {

Mono<M> intercept(Mono<M> message);

@Override

default BiFunction<Integer, M, M> handle(List<? extends M> messages) {

return (position, message) -> intercept(Mono.just(message)).block();

}

}

因此，它默认MessageDispatchInterceptor#句柄(List<?扩展M>方法，利用ReactorMessageDispatchInterceptor#intercept(Mono<M>)方法。因此，ReactorMessageDispatchInterceptor也可以配置在普通的轴突网关上。下面是一些如何使用消息分派拦截器的例子:

class ReactorConfiguration {

public void registerDispatchInterceptor(ReactorCommandGateway reactiveGateway) {

reactiveGateway.registerDispatchInterceptor(

msgMono -> msgMono.map(msg -> msg.andMetaData(Collections.singletonMap("key1", "value1")))

);

}

}

向消息的元数据添加键值对的分派拦截器。

class ReactorConfiguration {

public void registerDispatchInterceptor(ReactorEventGateway reactiveGateway) {

reactiveGateway.registerDispatchInterceptor(

msgMono -> msgMono.filterWhen(v -> Mono.subscriberContext()

.filter(ctx-> ctx.hasKey("security"))

.map(ctx->ctx.get("security")))

);

}

}

基于反应堆上下文中的安全标志丢弃消息的调度拦截器。

Reactor Result Handler Interceptors

ReactorResultHandlerInterceptor应该用于集中应用传入消息(也就是结果)的规则和验证。具体接口实现如下:

@FunctionalInterface

public interface ReactorResultHandlerInterceptor<M extends Message<?>, R extends ResultMessage<?>> {

Flux<R> intercept(M message, Flux<R> results);

}

参数是已发送的消息，以及将被拦截的消息的结果流。如果您希望只对特定的消息应用给定的结果规则，message参数可能会很有用。下面是一些如何使用消息结果拦截器的例子:

这种拦截器只能在反应堆扩展中使用。

class ReactorConfiguration {

public void registerResultInterceptor(ReactorQueryGateway reactiveGateway) {

reactiveGateway.registerResultHandlerInterceptor(

(msg, results) -> results.filter(r -> !r.getPayload().equals("blockedPayload"))

);

}

}

结果拦截器，丢弃所有有效载荷与blockedPayload匹配的结果

class ReactorConfiguration {

public void registerResultInterceptor(ReactorQueryGateway reactiveGateway) {

reactiveQueryGateway.registerResultHandlerInterceptor(

(query, results) -> results.flatMap(r -> {

if (r.getPayload().equals("")) {

return Flux.<ResultMessage<?>>error(new RuntimeException("no empty strings allowed"));

} else {

return Flux.just(r);

}

})

);

}

}

结果拦截器，它验证查询结果不包含空字符串。

class ReactorConfiguration {

public void registerResultInterceptor(ReactorQueryGateway reactiveGateway) {

reactiveQueryGateway.registerResultHandlerInterceptor(

(q, results) -> results.filter(it -> !((boolean) q.getQueryName().equals("myBlockedQuery")))

);

}

}

结果拦截器，丢弃queryName匹配myBlockedQuery的所有结果。

class ReactorConfiguration {

public void registerResultInterceptor(ReactorCommandGateway reactiveGateway) {

reactiveGateway.registerResultHandlerInterceptor(

(msg,results) -> results.timeout(Duration.ofSeconds(30))

);

}

}

结果拦截器，将每个消息的结果等待时间限制为30秒。

class ReactorConfiguration {

public void registerResultInterceptor(ReactorCommandGateway reactiveGateway) {

reactiveGateway.registerResultHandlerInterceptor(

(msg,results) -> results.log().take(5)

);

}

}

结果拦截器，它将结果的数量限制为5个条目，并记录所有结果。