# **五、事件**

## **5.1事件调度**

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

1. 从聚合中调度事件， 2 从常规组件调度事件

本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，请阅读以下内容节.事件调度。

## **5.1.1从聚合中调度事件**

聚合或它的实体通常是所有事件消息的起点。事件消息仅仅是一个决定已经做出的通知；一个成功处理命令消息的解决方案。

要从聚合中发布事件，需要从聚合实例的生命周期开始执行此操作。这是必需的，因为我们希望将聚合标识符绑定到事件消息。同样重要的是，事件的发生是有序的。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应用（Object）将经历多个步骤：

①将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提供的事件负载对象将包装在EventMessage中。EventMessage还将接收上一步中的sequenceNumber，以及它的标识符的聚合。

④事件消息将从此处发布。事件将首先发送到聚合中感兴趣的所有事件处理程序。这是事件回溯所必需的，以便相应地更新聚合的状态。

⑤在聚合本身处理了事件之后，它将被发布到EventBus上。

*聚合事件消息中的元数据*

*AggregateLifecycle还提供apply（对象、元数据）函数。这可用于附加命令处理程序特定的元数据。*

## **5.1.2从非聚合调度事件**

在绝大多数情况下，聚合将通过应用它们来发布事件。但是，有时有必要将事件（可能从另一个组件中）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

## **5.2事件处理程序（Event handler）**

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注释它们。方法的声明参数定义它将接收哪些事件。

Axon为以下参数类型提供现成的支持：

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注释上指定预期的负载类型。指定后，将使用下面指定的规则解析第一个参数。如果希望负载作为参数传递，请不要在注释上配置负载类型。
2. 用@MetaDataValue注释的参数将解析为带有注释所示键的元数据值。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，则解析程序将不匹配，并且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MetaData。
4. 用@Timestamp注释的参数，类型为java.time.Instant（或java.time.temporal.Temporal）将解析为事件消息的时间戳。这是生成事件的时间。
5. 用@SequenceNumber和类型“”批注的参数java.lang.Long文件或long将解析为DomainEventMessage的sequenceNumber。这提供了事件生成的顺序（在它起源的聚合范围内）。需要注意的是，DomainEventMessage只能来自聚合。因此，直接在EventBus/EventGateway上发布的事件不是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，那么它将有效地匹配任何类型的事件，即使泛型参数会建议使用其他类型。由于类型擦除，Axon无法检测预期的参数。在这种情况下，最好声明一个有效负载类型的参数，然后再声明一个message类型的参数。
7. TrackingToken类型的参数将注入与已处理事件相关的当前TrackingToken。
8. 当使用Spring并激活Axon配置时（通过包括Axon-Spring引导启动模块），如果应用程序上下文中只有一个可注入的候选对象可用，那么任何其他参数都将解析为autowired bean。这允许您将资源直接注入@EventHandler带注释的方法中。

通过实现ParameterResolverFactory接口并创建名为/META-INF/service的文件，可以配置其他ParameterResolver/org.axonframework.common.annotation.ParameterResolverFactory包含实现类的完全限定名。

在所有情况下，每个侦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具体的方法：

1. 在类层次结构的实际实例级别上（由这个.getClass（）），对所有带注释的方法进行求值
2. 如果找到一个或多个方法，其中的所有参数都可以解析为一个值，那么将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个级别上找不到方法，则以相同的方式计算超类型
4. 当到达层次结构的顶层，并且没有找到合适的事件处理程序时，将忽略该事件。

// assume EventB extends EventA

// and EventC extends EventB

// and a single instance of SubListener is registered

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，SubListener的处理程序方法将为EventB和EventC的所有实例调用（因为它扩展了EventB）。换句话说，TopListener的处理程序方法根本不会接收任何EventC调用。由于EventA不可分配给EventB（它是它的超类），因此这些将由TopListener中的handler方法处理。

### **5.2.1注册事件处理程序**

事件处理组件是使用EventProcessingConfigurer定义的，可以从全局Axon Configurer访问它。EventProcessingConfigurer用于配置EventProcessingConfiguration。通常，应用程序将定义单个事件处理配置，但更大的模块化应用程序可能会选择为每个模块定义一个。

要使用@EventHandler方法注册对象，请在EventProcessingConfigurer上使用registerEventHandler（）方法：

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.registerEventHandler(conf -> new MyEventHandlerClass()));

或者配置事件处理器：

Configurer configurer = DefaultConfigurer.defaultConfiguration()

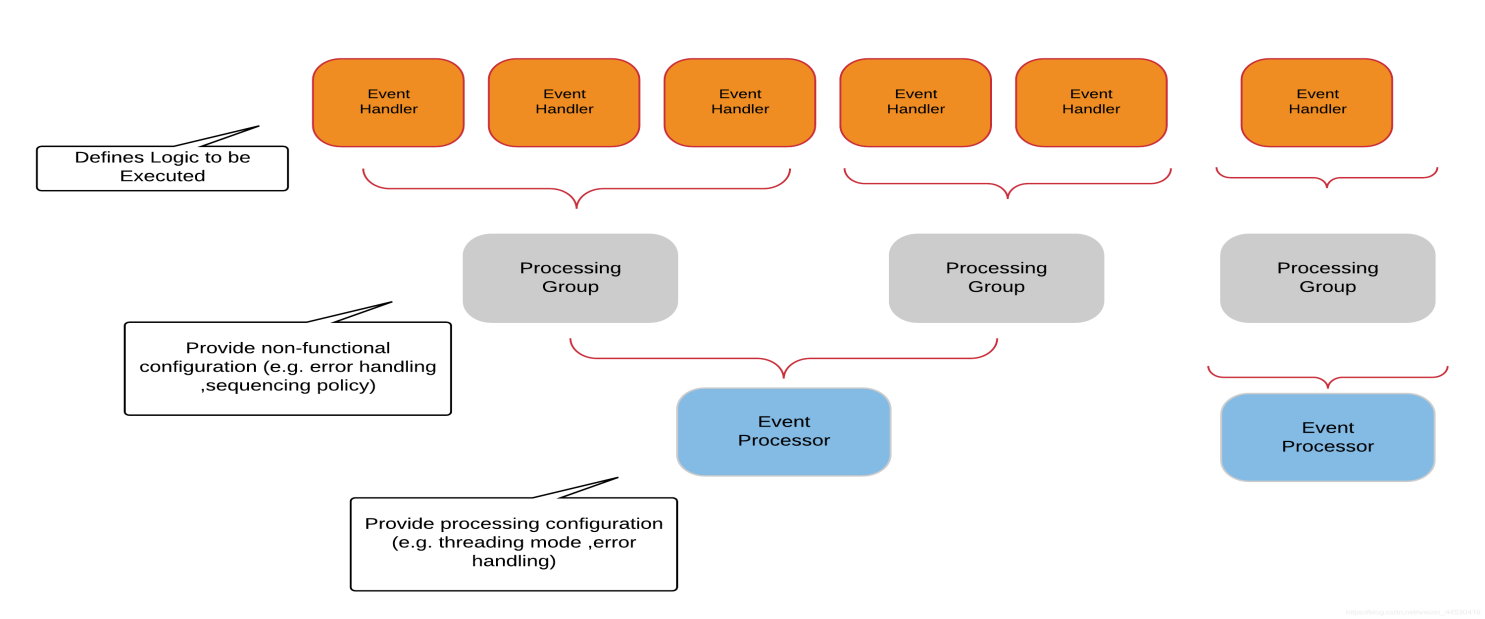
.eventProcessing(eventProcessingConfigurer -> eventProcessingConfigurer

.registerEventHandler(conf -> new MyEventHandlerClass()));

## **5.3事件处理器（event processor）**

事件处理程序定义接收事件时要执行的业务逻辑。事件处理器是处理该处理的技术方面的组件。他们开始一个工作单元，可能还有一个事务。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。

下面描述了事件处理器和事件处理程序的组织结构。



事件处理器大致有两种形式：订阅和跟踪。订阅事件处理器自己订阅事件源，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它自己管理的线程从源获取消息。

### **5.3.1将处理程序分配给处理器**

一个JVM实例标识所有处理器的一个实例。具有相同名称的两个处理器被视为同一处理器的两个实例。所有事件处理程序都附加到一个处理器，其名称是事件处理程序类的包名称。

例如，以下类：

org.axonframework.example.eventhandling.MyHandler,

org.axonframework.example.eventhandling.MyOtherHandler，和

org.axonframework.example.eventhandling.module.MyHandler

将触发两个处理器的创建：

org.axonframework.example.eventhandling（有2个处理程序），以及

org.axonframework.example.eventhandling.module（只有一个处理程序）

配置API允许您配置将类分配给处理器的其他策略，甚至可以将特定的处理程序实例分配给特定的处理器。

### **5.3.2在单个事件处理器中对事件处理程序排序**

要对事件处理器中的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”部分中所述）是指导。因此，事件处理器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。此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顺序。

并不是说不可能注册不属于同一事件处理器一部分的事件处理程序。

### **5.3.3配置处理器**

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。但是，“常规”（单例、无状态）事件处理程序的配置方式与sagas略有不同，因为不同的方面对这两种类型的处理程序都很重要。

5.3.3.1事件处理器

默认情况下，Axon将使用跟踪事件处理器（Tracking Event Processors）。可以更改处理程序的分配方式和处理器的配置方式。

Axon部分：

EventProcessingConfigurer类定义了许多方法，可用于定义处理器的配置方式。

1. registerEventProcessorFactory允许您定义一个默认工厂方法，该方法创建尚未定义显式工厂的事件处理器。
2. registerEventProcessor（字符串名称，EventProcessorBuilder builder）定义用于创建具有给定名称的处理器的工厂方法。请注意，只有选择name作为任何可用事件处理程序bean的处理器时，才会创建这样的处理器。
3. registerTrackingEventProcessor（字符串名称）定义应使用默认设置将具有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，默认情况下都是从主配置获取的。
4. registerTrackingProcessor（字符串名称，函数<配置，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TrackingEventProcessorConfiguration读取多线程的配置设置。StreamableMessageSource定义了一个事件源，此处理器应从中提取事件。
5. usingSubscriptingEventProcessors（）将默认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 ->

ventProcessingConfigurer.usingSubscribingEventProcessors())

.eventProcessing(eventProcessingConfigurer ->

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

);

**springBoot部分**

// Default all processors to subscribing mode.

@Autowired

public void configure(EventProcessingConfigurer config) {

config.usingSubscribingEventProcessors();

}

在某些方面，事件处理器也可以应用程序.属性.

axon.eventhandling.processors.name.mode=subscribing

axon.eventhandling.processors.name.source=eventBus

如果事件处理器的名称包含句点，请使用映射表示法：

axon.eventhandling.processors[name].mode=subscribing

axon.eventhandling.processors[name].source=eventBus

5.3.3.2多个事件源

您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，或者仅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在任何给定时刻都可以选择使用多个事件。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早的事件（即等待时间最长的事件）。

多个源还意味着跟踪处理器的轮询间隔需要在源之间进行划分，使用一种策略来优化事件发现，并将建立到数据源的昂贵连接的开销降到最低。因此，您可以使用构建器中的longPollingSource（）方法选择对哪个源执行大多数轮询。这样可以确保一个源占用大部分轮询间隔，同时也间歇性地检查其他源上的事件。默认的longPollingSource是在最后配置的源上完成的。

Axon部分：

使用其builder（）创建一个MultiStreamableMessageSource，并在调用事件时将其注册为消息源ProcessingConfigurer.registerTrackingEventProcessor().

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();

Springboot部分：

@Bean

public MultiStreamableMessageSource multiStreamableMessageSource(EventStore eventSourceA, EventStore eventStoreB) {

return 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();

}

@Autowired

public void configure(EventProcessingConfigurer config, MultiStreamableMessageSource multiStreamableMessageSource) {

config.registerTrackingEventProcessor("NameOfEventProcessor", c -> multiStreamableMessageSource);

}

5.3.3.3 saga

你也可以改变saga的配置。

Axon部分：

使用SagaConfigurer来配置saga：

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.eventProcessing(eventProcessingConfigurer ->

eventProcessingConfigurer.registerSaga(MySaga.class,

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

.configureRepository(c -> repository)

.configureSagaManager(c -> manager)

)

);

**Springboot部分：**

用于操作sagas的基础设施组件的配置由@Saga注释触发（在org.axonframework.spring.stereotype包）。Axon将配置一个sagamanger和SagaRepository。SagaRepository将使用上下文中可用的SagaStore（如果找到JPA，则默认为JPASagaStore）来实际存储sagas。

要为sagas使用不同的SagaStore，请提供SagaStore的bean名称，以便在每个@Saga注释的SagaStore属性中使用。

Sagas将从应用程序上下文注入资源。注意，这并不意味着使用Spring注入来注入这些资源。@Autowired和@javax.inject.inject.注入注释可以用来划分依赖关系，但它们是由Axon通过在字段和方法上查找这些注释来注入的。尚不支持构造函数注入。

为了优化sagas的配置，可以定义一个自定义sagaconfigurationbean。对于带注释的saga类，Axon将尝试为该saga查找配置。它通过检查具有特定名称的SagaConfiguration类型的bean来实现。对于名为MySaga的saga类，Axon查找的bean是mySagaConfiguration。如果没有找到这样的bean，它将基于可用组件创建一个配置。

如果一个SagaConfiguration实例存在于带注释的saga中，那么该配置将用于检索和注册该类型saga的组件。如果SagaConfiguration bean没有如上所述命名，则saga可能会注册两次。然后它将接收两份事件。为了防止这种情况发生，您可以使用@Saga注释指定SagaConfiguration的bean名称：

@Autowired

public void configureSagaEventProcessing(EventProcessingConfigurer config) {

config.registerTokenStore("MySagaProcessor", c -> new MySagaCustomTokenStore())

}

### **5.3.4 错误处理**

错误是不可避免的。根据它们发生的地点，你可能需要做出不同的反应。

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。当处理器尝试提交事务、更新令牌或在进程的任何其他部分中引发异常时，将传播异常。对于跟踪事件处理器，这意味着处理器将进入错误模式，释放所有令牌并以增量间隔（从1秒开始，最多60秒）重试。订阅事件处理器将向提供事件的组件报告发布错误。

要更改此行为，您可以在两个级别定义Axon如何处理异常：

5.3.4.1事件处理程序方法引发的异常

默认情况下，会记录这些异常，并继续处理下一个处理程序或消息。

可以为每个处理组配置此行为：

Axon部分：

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

// or for a specific processing group:

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

**springboot部分**

@Autowired

public void configure(EventProcessingConfigurer config) {

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

// or for a specific processing group:

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

}

实现自定义错误处理行为很容易。要实现的错误处理方法提供异常、已处理的事件以及对处理消息的处理程序的引用。您可以选择重试、忽略或重新引发异常。在后一种情况下，异常将冒泡到事件处理器级别。

5.3.4.2处理过程中出现异常

发生在事件处理程序范围之外的异常，或从事件处理程序范围内冒出的异常由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，它将使用增量退避期重试处理事件。它将从1秒开始并在每次尝试后加倍，直到每次尝试的最大等待时间达到60秒。此退避时间确保如果另一个节点能够成功地处理事件，它将有机会声明处理该事件所需的令牌。

订阅EventProcessor将把异常冒泡到事件的发布组件，从而允许它相应地处理它。

您可以在事件处理程序级别配置错误处理程序。

Axon部分

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

// or for a specific processor:

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

springboot部分

@Autowired

public void configure(EventProcessingConfigurer config) {

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

// or for a specific processing group:

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

}

要实现自定义行为，请实现ErrorHandler的单个方法。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。

### **5.3.5Token存储器**

与订阅事件处理器不同，跟踪事件处理器需要一个token存储器来存储它们的进度。跟踪处理器通过其事件流接收到的每条消息都有一个token。此token允许处理器在以后的任何时候重新打开流，从上次事件停止的地方恢复。

配置API从全局配置实例中获取令牌存储以及处理器所需的大多数其他组件。如果没有显式定义token存储，则使用InMemoryTokenStore，这在生产中不建议使用。

Axon部分

要配置令牌存储，请使用EventProcessingConfigurer定义要使用的实现。

要为所有处理器配置默认令牌存储，请执行以下操作：

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

或者，要为特定处理器配置token存储，请使用：

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

Springboot部分

默认的令牌库实现是根据Spring Boot中可用的依赖项定义的，顺序如下：

①如果定义了任何令牌存储bean，则使用该bean

②否则，如果EntityManager可用，则定义JpaTokenStore。

③否则，如果定义了数据源，则会创建JdbcTokenStore

④最后，使用InMemoryToken存储

要覆盖TokenStore，可以在Spring@Configuration类中定义bean：

@Bean

public TokenStore myCustomTokenStore() {

return new MyCustomTokenStore();

}

或者，插入EventProcessingConfigurer，它允许更细粒度的定制：

@Autowired

public void configure(EventProcessingConfigurer epConfig) {

epConfig.registerTokenStore(conf -> new MyCustomTokenStore());

// or, to define one for a single processor:

epConfig.registerTokenStore("processorName", conf -> new MyCustomTokenStore());

}

请注意，您可以在定义该处理器的EventProcessingConfiguration中重写token存储以与跟踪处理器一起使用。在可能的情况下，建议使用token存储区，该存储区将token存储在事件处理程序更新视图模型的同一数据库中。这样，对视图模型的更改可以与更改的token一起以原子方式存储。这保证了只处理一次语义。

### **5.3.6*分支*和合并跟踪token**

可以通过*分支*来增加在高负载下处理事件的线程数，以及通过合并段来减少负载时的线程数，来调整跟踪事件处理器的性能。

在运行时允许*分支*和合并，这允许您动态控制段的数量。这可以通过Axon服务器API或通过Axon框架使用TrackingEventProcessor的splitSegment（int segmentId）和mergeSegment（int segmentId）方法来完成，方法是提供要*分支*或合并的段的segmentId。

*段选择注意事项*

*通过使用Axon Server进行拆分/合并，可以为您选择最适合拆分或合并的段。当直接使用Axon Framework API时，需要拆分/合并的段应该由开发人员自己推导：*

*分支：为了达到公平平衡，最好在最大的细分市场上进行分割*

*合并：理想情况下，在最小合并段上执行公平合并*

### **5.3.7 并行处理**

跟踪处理器可以使用多个线程来处理一个事件流。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。

可以定义使用的段数。当事件处理器第一次启动时，它可以初始化许多段。此数字定义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大量线程以开始处理事件。

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特定的顺序发送到这些处理程序。Axon为此使用了Sequencing策略。SequencingPolicy是一个函数，它为任何给定的消息返回一个值。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，这使得同一个聚合实例发布的事件将按顺序处理。

saga实例从不由多个线程并发调用。因此，一个saga的排序策略是无关紧要的。Axon将确保每个saga实例按照在事件总线上发布的顺序接收需要处理的事件。

*并行处理和订阅事件处理器*

*请注意，订阅事件处理器不管理自己的线程。因此，不可能配置它们应该如何接收事件。实际上，它们将始终按顺序逐个聚合工作，因为这通常是命令处理组件中的并发级别。*

Axon部分配置：

DefaultConfigurer.defaultConfiguration()

.eventProcessing(eventProcessingConfigurer -> eventProcessingConfigurer.registerTrackingEventProcessor(

"myProcessor",

c -> c.eventStore(),

c -> c.getComponent(

TrackingEventProcessorConfiguration.class,

() -> TrackingEventProcessorConfiguration.forParallelProcessing(3)

)

)

);

Springboot部分

您可以配置线程数（在这个实例上）以及处理器应该定义的初始段数（如果还没有可用的话）。

axon.eventhandling.processors.name.mode=tracking

# Sets the number of maximum number threads to start on this node

axon.eventhandling.processors.name.threadCount=2

# Sets the initial number of segments (i.e. defines the maximum number of overall threads)

axon.eventhandling.processors.name.initialSegmentCount=4

5.3.7.1顺序处理

即使事件是从其发布者异步处理的，通常也需要按照事件的发布顺序来处理某些事件。在Axon中，这是由SequencingPolicy控制的。

SequencingPolicy定义事件是必须按顺序处理、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等的标识符，这意味着事件处理程序必须按顺序处理它们。空序列标识符表示事件可以与任何其他事件并行处理。

Axon提供了许多可以使用的通用策略：

1. FullConcurrencyPolicy将告诉Axon此事件处理程序可以并发处理所有事件。这意味着需要按特定顺序处理的事件之间没有关系。
2. SequentialPolicy告诉Axon所有事件都必须按顺序处理。事件的处理将在上一个事件的处理完成后开始。
3. 默认策略是SequentialPerAggregatePolicy。它将强制按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更新详细信息的事件侦听器，这通常是一个合适的策略。

除了这些提供的策略，您还可以定义自己的策略。所有策略必须实现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。对于返回等序列标识符的事件，必须按顺序处理。可以同时处理产生不同序列标识符的事件。如果事件可以与任何其他事件并行处理，则策略实现可能返回null。

eventProcesingConfigurer.registerSequencingPolicy("processingGroup", conf -> /\* define policy \*/);

// or, to change the default:

eventProcesingConfigurer.registerDefaultSequencingPolicy(conf -> /\* define policy \*/);

5.3.7.2多节点处理

对于跟踪处理器，处理事件的线程是在同一个节点上运行，还是在托管同一（逻辑）跟踪处理器的不同节点上运行并不重要。当具有相同名称的跟踪处理器的两个实例在不同的计算机上处于活动状态时，它们被视为同一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令牌库实现中重写。

### **5.3.8分发事件**

默认情况下，Axon服务器支持在进程间环境中分发事件。

或者，您可以选择在扩展模块（springamqp，Kafka）中可以找到的其他组件。

### **5.3.9重播事件**

在需要重建映射（视图模型）的情况下，回放过去的事件非常有用。其思想是从头开始，重新调用所有事件处理程序。TrackingEventProcessor支持事件回放。为了实现这一点，应该对其调用resetTokens（）方法。

重要的是要知道，当启动重置时，跟踪事件处理器不能处于活动状态。因此，需要先关闭它，然后重置它，一旦成功，之后可以重新启动。

通过TrackingEventProcessor启动重播将打开一个API以进入重播过程。例如，可以定义一个@ResetHandler，这样就可以在重置之前做一些准备。

让我们看看如何实现跟踪事件处理器的重放。首先，我们将看到一个简单的投影类：

@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) {

// This event handler performs a "side effect",

// like sending an e-mail or a sms.

// Neither, is something we want to reoccur when a

// replay happens, hence we disallow this method

// to be replayed

}

@EventHandler

public void on(CardRedeemedEvent event, ReplayStatus replayStatus

/\* 3. \*/) {

// We can wire a ReplayStatus here so we can see whether this

// event is delivered to our handler as a 'REGULAR' event or

// a 'REPLAY' event

// Perform event handling

}

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

public void onReset(ResetContext resetContext) {

// Do pre-reset logic, like clearing out the projection table for a

// clean slate. The given resetContext is [optional], allowing the

// user to specify in what context a reset was executed.

}

//...

}

CardSummaryProjection显示了“意识到”正在进行重置时需要注意的几个有趣的事情：

1. 可以使用@AllowReplay，它位于整个类或@EventHandler注释的方法上。它定义在重播传输过程中是否应调用给定的类或方法。
2. 除了允许重播之外，还可以使用@DisallowReplay。与@AllowReplay类似，它可以放在类级别和方法上，用于定义在传输重播时是否不应调用类/方法。
3. 要对重播期间的操作（或不能做什么）进行更细粒度的控制，可以添加ReplayStatus参数。它是一个附加参数，可以添加到@EventHandler注释的方法中，允许根据重播是否在传输中执行条件操作。
4. 如果需要执行某些预重置逻辑，例如清除投影表，则应使用@ResetHandler注释。此批注只能放置在方法上，以便在必要时添加重置上下文。@ResetHandler中传递的resetContext源自启动TrackingEventProcessor#resetTokens（R resetContext）方法的操作。resetContext的类型由用户决定。

所有这些就绪后，我们就可以从TrackingEventProcessor启动重置了。为此，我们需要访问要重置的TrackingEventProcessor。为此，您应该检索主配置中可用的EventProcessingConfiguration。另外，在这里我们可以提供一个可选的重置上下文在@ResetHandler中传递：

Axon部分

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();

});

}

}

Springboot部分

public class ResetService {

//...

public <R> void resetWithContext(Configuration config, R resetContext) {

EventProcessingConfiguration eventProcessingConfig = config.eventProcessingConfiguration();

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

.ifPresent(processor -> {

processor.shutDown();

processor.resetTokens(resetContext);

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，可以通过TrackingEventProcessorConfiguration配置EventTrackerStatusChangeListener。有关更改侦听器的更多细节，请参阅监视和度量。

*部分重播*

*可以提供重置TrackingEventProcessor时使用的令牌位置，从而指定从事件日志中的哪个点开始重放事件。这将需要使用TrackingEventProcessor重置令牌（TrackingToken）或TrackingEventProcessor重置令牌（函数<StreamableMessageSource<TrackedEventMessage<？>>，TrackingToken>）方法，这两个方法都提供应从中开始重置的TrackingToken。*

*这里描述了如何自定义跟踪令牌位置。*

5.3.10自定义跟踪token位置

在Axon 3.3版之前，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提供选项，通过andInitialTrackingToken（Function<StreamableMessageSource，TrackingToken>）生成器方法为给定的TrackingEventProcessor设置初始令牌。作为令牌生成器函数的输入参数，我们收到一个StreamableMessageSource，它为我们提供了三种构建令牌的可能性：

1. 从事件流的头部：createHeadToken（）。
2. 从事件流的尾部：createTailToken（）。
3. 从某个时间点开始：createTokenAt（即时）和createTokenSince（持续时间）-创建跟踪给定时间后所有事件的令牌。如果在给定的时间有一个事件，它也会被考虑在内。

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

下面我们可以看到一个在“2007-12-03T10:15:30.00Z”上使用初始令牌创建TrackingEventProcessorConfiguration的示例：

public class Configuration {

public TrackingEventProcessorConfiguration customConfiguration() {

return TrackingEventProcessorConfiguration

.forSingleThreadedProcessing()

.andInitialTrackingToken(streamableMessageSource -> streamableMessageSource.createTokenAt(

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

));}}

## **5.4 事件总线和事件存储**

### **5.4.1事件总线**

EventBus是将事件分派给订阅的事件处理程序的机制。Axon提供了事件总线的三种实现：AxonServerEventStore、EmbeddedEventStore和SimpleEventBus。这三种实现都支持订阅和跟踪处理器（请参阅事件处理器）。但是，AxonServerEventStore和EmbeddedEventStore持久化事件（请参阅事件存储），这允许您在稍后阶段重放它们。SimpleEventBus有一个易失性存储，一旦事件被发布到订阅的组件上，就会“忘记”事件。

默认情况下配置了AxonServerEventStore事件总线/存储。

### **5.4.2事件存储**

事件源存储库需要一个事件存储库来存储和加载聚合中的事件。事件存储提供事件总线的功能。此外，它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。

5.4.2.1作为事件存储的Axon服务器

Axon提供了一个现成的事件存储，即AxonServerEventStore。它连接到axoniqaxonserver服务器来存储和检索事件。

Axon部分

依赖：

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

<dependency>

<groupId>org.axonframework</groupId>

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

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

</dependency>

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-configuration</artifactId>

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

</dependency>

配置代码

// Returns a Configurer instance with default components configured.

// `AxonServerEventStore` is configured as Event Store by default.

Configurer configurer = DefaultConfigurer.defaultConfiguration();

Springboot部分

通过简单地声明对axon spring boot starter的依赖，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>

*排除Axon服务器连接器*

*如果您从axon spring boot starter中排除axon服务器连接器依赖项，则EmbeddedEventStore将为您自动配置，前提是EventStorageEngine的具体实现可用。如果在JDBC上分别配置了jpa或JDBC，则将分别自动配置JpaEventStorageEngine或JdbcEventStorageEngine。如果两者都不存在，自动配置将返回到SimpleEventBus。*

5.4.2.2嵌入式事件存储

或者，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：

1. JPAEventstorage引擎

JpaEventStorageEngine将事件存储在与JPA兼容的数据源中。JPA事件存储在条目中存储事件。这些条目包含事件的序列化形式，以及存储元数据以快速查找这些条目的某些字段。要使用JpaEventStorageEngine，必须具有JPA(javax.持久性)类路径上的注释。

默认情况下，事件存储需要您配置持久性上下文（例如，按照META-INF中的定义/持久性.xml文件）来包含类DomainEventEntry和SnapshotEventEntry（这两个类都位于org.axonframework.eventsourceing.事件存储.jpa包）。

以下是持久性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. 这行代码用持久性上下文注册DomainEventEntry（JpaEventStore使用的类）。

*唯一关键约束考虑*

*Axon使用锁定来防止两个线程访问同一个聚合。但是，如果您有多个jvm使用同一个数据库，这对您没有帮助。在这种情况下，您必须依赖数据库来检测冲突。并发访问事件存储将导致键约束冲突，因为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可以检测到此错误并将其转换为ConcurrencyException。但是，每个数据库系统报告这种冲突的方式不同。如果您向JpaEventStore注册数据源，它将尝试检测数据库的类型，并找出哪些错误代码表示键约束冲突。或者，可以提供PersistenceExceptionTranslator实例，该实例可以判断给定的异常是否表示键约束冲突。*

*如果没有提供DataSource或PersistenceExceptionTranslator，则会按原样引发来自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EventStorageEngine要使用的EntityManager实例。这也允许使用应用程序管理的持久性上下文。EntityManagerProvider负责提供EntityManager的正确实例。

有几种EntityManagerProvider的实现可供选择，每种实现都可以满足不同的需要。

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，默认情况下由JPA事件存储使用。

如果您有一个名为“myPersistenceUnit”的持久性单元希望在JpaEventStore中使用，那么EntityManagerProvider实现可以如下所示：

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事件存储将条目存储在DomainEventEntry和SnapshotEventEntry实体中。虽然这在许多情况下已经足够了，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。还可能需要在不同的表中存储不同聚合类型的事件。

如果是这样，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来调整其行为。

*警告*

*请注意，持久性提供者（如Hibernate）在其EntityManager实现中使用一级缓存。通常，这意味着查询中使用或返回的所有实体都附加到EntityManager。只有在提交了周围的事务或在事务内部执行了显式的“清除”时，才会清除它们。尤其是在事务上下文中执行查询时。*

*要解决此问题，请确保以独占方式查询非实体对象。您可以使用JPA的“selectnewsomeclass（parameters）FROM…”样式的查询来解决这个问题。或者，打电话实体管理器.flush（）和实体管理器清除（）获取一批事件后。否则，在加载大型事件流时，可能会导致OutOfMemoryException。*

Axon配置部分

// Returns a Configurer instance which has JPA components configured,

// such as a JPA based Event Store `JpaEventStorageEngine`, a //`JpaTokenStore` and `JpaSagaStore`.

Configurer configurer = DefaultConfigurer.jpaConfiguration(entityManagerProvider, transactionManager);

**Springboot部分**

// 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();

}

*排除Axon服务器连接器*

*如果您从axon spring boot starter中排除axon服务器连接器依赖项，则EmbeddedEventStore将为您自动配置，前提是EventStorageEngine的具体实现可用。如果在JDBC上分别配置了jdaeventsageo或JDBC，则将分别检测到jdaeventsageengine或JDBC。如果两者都不存在，自动配置将返回到SimpleEventBus。*

5.4.2.3 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，每个事件都存储在一个条目中，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来获取连接。通常，这些连接可以直接从数据源获得。但是，Axon会将这些连接绑定到一个工作单元，因此在一个工作单元中使用一个连接。这可以确保使用单个事务来存储所有事件，即使多个工作单元嵌套在同一个线程中。

Axon部分：  
// Returns a Configurer instance with default components configured.

// We explicitly set `JdbcEventStorageEngine` as desired engine for //Embedded Event Store.

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureEmbeddedEventStore(

c -> JdbcEventStorageEngine.builder()

.connectionProvider(connectionProvider)

.transactionManager(NoTransactionManager.INSTANCE)

.build()

);

S*QL语句可定制性*

*数据库在不同场景下执行的最佳SQL语句之间有细微的偏差。由于为所有可能性进行优化超出了框架的范围，所以可以调整正在使用的默认语句。*

*检查JdbcEventStorageEngineStatements实用程序类中JdbcEventStorageEngine使用的默认语句。更进一步说org.axonframework.eventsourceing.eventstore.jdbc.statements包包含可调整语句集。这些语句生成器中的每一个都可以通过Jd进行定制bcEventStorageEngine.Builder.*

Springboot部分

// 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();

}

Springboot方面

// 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();

}

*使用Spring的数据源提供程序*

*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();

排除Axon服务器连接器

如果您从axon spring boot starter中排除axon服务器连接器依赖项，则EmbeddedEventStore将为您自动配置，前提是EventStorageEngine的具体实现可用。当需要使用JDBC作为事件存储方法时，可以直接提供jdbceventstoragengine bean，或者通过公开数据源bean让Axon自动配置它。

5.4.2.4 MongoEvents存储引擎

MongoDB是一个基于文档的NoSQL存储。它的可伸缩性特性使它适合用作事件存储。Axon提供了MongoEventStorageEngine，它使用MongoDB作为后台数据库。它包含在Axon Mongo模块（Maven artifactId Axon Mongo）中。

事件存储在两个独立的集合中：一个用于事件流，另一个用于快照。

默认情况下，MongoEventStorageEngine将每个事件存储在单独的文档中。但是，可以更改使用的存储策略。Axon提供的另一种选择是DocumentPerCommitStorageStrategy，它为存储在一次提交（即在同一个DomainEventStream中）的所有事件创建一个文档。

将整个提交存储在单个文档中的优点是提交是以原子方式存储的。此外，对于任何数量的事件，它只需要一次往返。缺点是直接在数据库中查询事件变得更加困难。例如，重构域模型时，如果事件包含在“提交文档”中，则很难将事件从一个聚合“传输”到另一个聚合。

MongoEventStorageEngine不需要很多配置。它只需要引用存储事件的集合，就可以开始了。对于生产环境，您可能需要仔细检查集合上的索引。

Axon部分

// Returns a Configurer instance with default components configured.

// We explicitly set `MongoEventStorageEngine` as desired engine for //Embedded Event Store.

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureEmbeddedEventStore(

c -> MongoEventStorageEngine.builder().mongoTemplate(mongoTemplate).build()

);

Springboot方面

// 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();

}

排除Axon服务器连接器

如果您从axon spring boot starter中排除axon服务器连接器依赖项，则EmbeddedEventStore将为您自动配置，前提是EventStorageEngine的具体实现可用。当需要使用Mongo作为事件存储方法时，这意味着提供一个MongoEventStorageEngine bean。

### **5.4.3事件存储实用程序**

Axon提供了许多事件存储引擎，这些引擎在某些情况下可能有用。

5.4.3.1内存内事件存储

InMemoryEventStorageEngine将存储的事件保存在内存中。虽然它可能比其他任何一个事件存储都要好，但它并不是真正意义上的长期生产使用。然而，它在需要事件存储的短期工具或测试中非常有用。

Axon部分

// Returns a Configurer instance with default components configured.

// We explicitly set `InMemoryEventStorageEngine` as desired engine //for Embedded Event Store.

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureEmbeddedEventStore(c -> new InMemoryEventStorageEngine());

**Springboot部分**

// 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();

}

*排除Axon服务器连接器*

*如果您从axon spring boot starter中排除axon服务器连接器依赖项，则EmbeddedEventStore将为您自动配置，前提是EventStorageEngine的具体实现可用。当需要使用内存中的事件存储方法时，这意味着提供一个InMemoryEventStorageEngine Bean。*

5.4.3.2将多个事件存储合并为一个

SequenceEventStorageEngine是另外两个事件存储引擎的包装器。读取时，它从两个事件存储引擎返回事件。附加事件仅附加到第二个事件存储引擎。例如，当出于性能原因使用两种不同的事件存储实现时，这很有用。第一个将是一个更大但速度较慢的事件存储，而第二个是为快速读写而优化的。

5.4.3.3筛选存储的事件

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无法接收这些事件，因为它们没有被存储。

### **5.4.4影响序列化过程**

事件存储需要一种序列化事件的方法来为存储做准备。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，比Java序列化更灵活。此外，XStream序列化的结果是可读的。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。除了缩短可能很长的名称的好方法外，别名还可以在事件的类定义更改时使用。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。虽然它生成了一个更紧凑的序列化表单，但它确实要求类遵守Jackson所要求的约定（或配置）。

您也可以实现自己的序列化程序，只需创建一个实现序列化程序的类，并将事件存储配置为使用该实现而不是默认实现。

Axon部分

// Returns a Configurer instance with default components configured.

// We explicitly set `JacksonSerializer` as desired event serializer.

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureEventSerializer(c -> JacksonSerializer.builder().build());

Springboot部分

你可以在你的application.properties中指定序列化工具：

# somewhere in your `application.properties`

axon.serializer.events=jackson

# posible values: java, xstream, jackson

然后你可以在你的spring上下文中显式的定义序列化器

// somewhere in your `@Configuration` class

@Qualifier("eventSerializer")

@Bean

public Serializer eventSerializer() {

return JacksonSerializer.builder().build();

}

序列化器对比：

与Axon需要序列化的所有其他对象（如命令、快照、saga等）相比，可以使用不同的序列化程序来存储事件。虽然XStreamSerializer几乎可以序列化任何东西，这使得它成为一个非常不错的默认值，但它的输出并不总是一个便于与其他应用程序共享的表单。JacksonSerializer可以创建更好的输出，但是需要对象中的某种结构来序列化。此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。

如果未配置显式事件序列化程序，则使用已配置的主序列化程序（默认为XStreamSerializer）序列化事件。

## **5.5 事件版本设置**

在Axon应用程序的生命周期中，事件通常会改变其格式。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。本章将讨论在创建事件时要记住什么，以便向后（和向前）兼容。它也将解释向上转型的过程。

### **5.5.1事件上转**

由于软件应用程序的性质不断变化，事件定义很可能也会随着时间的推移而改变。由于事件存储被视为只读取和附加的数据源，因此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。这就是向上转型的原因。

最初是面向对象编程的概念，其中“子类在需要时自动转换为其超类”，上转的概念也可以应用于事件溯源。上转事件意味着将其从原始结构转换为新结构。与OOP上推不同，事件上转不能完全自动化，因为新事件的结构对于旧事件是未知的。必须提供手动编写的向上转换程序，以指定如何将旧结构上移到新结构。

向上转换程序是接受修订版x的一个输入事件并输出修订版x+1的零个或多个新事件的类。此外，上转是在一个链中处理的，这意味着一个上转的输出被发送到下一个上转的输入。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一个向上转换程序，使它们变得更小、独立且易于理解。

*注意*

*上转的最大好处可能是它允许您进行非破坏性重构。换句话说，完整的事件历史保持完整。*

在这一节中，我们将解释如何编写一个上转，描述Axon附带的上转的不同（抽象）实现，并解释事件的序列化表示如何影响上转的编写。

为了让升级者看到他们接收到的序列化对象的版本，事件存储区存储了一个修订号和事件的完全限定名。此修订号由在序列化程序中配置的修订解析程序生成。Axon提供了几种RevisionResolver的实现：

1. AnnotationRevisionResolver检查事件负载上的@Revision注释。
2. SerialVersionUIDRevisionResolver使用Java序列化API定义的serialVersionUID。
3. FixedValueRevisionResolver始终返回预定义的值。这在注入当前应用程序版本时特别有用。它将允许您查看应用程序的哪个版本生成了特定事件。
4. Maven用户可以使用MavenArtifactRevisionResolver自动使用项目版本。它是使用项目的groupId和artifactId初始化的，以获取的版本。因为这只适用于Maven创建的JAR文件，所以版本不能总是由IDE解析。如果无法解析版本，则返回null。

Axon的向上转换程序不直接使用EventMessage，而是使用中间的VentRepresentation。IntermediateEventRepresentation提供了检索所有必需字段的功能，以构建EventMessage（因此也包括upcasted EventMessage），以及实际的上推函数。默认情况下，这些上推函数只允许调整事件的有效载荷、有效载荷类型和添加到事件元数据中。upcast函数中事件的实际表示形式可能会因所使用的事件序列化程序或要使用的所需形式而有所不同，因此IntermediateEventRepresentation的上溯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）不可由IntermediateEventRepresentation调整。调整这些字段不是上变频器的预期工作。因此，所提供的IntermediateEventRepresentation实现不提供这些选项。

Axon框架中事件的基本Upcaster接口作用于IntermediateeVentRepresentation流，并返回IntermediateEventRepresentations流。因此，向上转换过程不会直接返回引入的上推函数的最终结果，而是通过堆叠中间eVentRepresentation将每个上推函数从一个修订版链接到另一个修订版。一旦这个过程发生并从中提取最终结果，即对序列化事件执行实际的上推函数。

### **5.5.2提供了抽象的上转实现**

如前所述，上转接口不向上转换单个事件；它需要流<IntermediateEventRepresentation>并返回一个。然而，通常编写一个向上转换程序来调整流外的单个事件。更复杂的上转设置也是可以想象的。例如，从一个事件到多个事件，或者一个上转程序，它从一个先前的事件中提取状态，然后将其推送到下一个事件中。本节描述了当前提供的（抽象）事件向上转换程序的实现，用户可以扩展这些实现来添加自己所需的向上转换功能。

1. SingleEventUpcaster—一个事件升级程序的一对一实现。从这个实现扩展需要实现canUpcast和doUpcast函数，这两个函数分别检查手头的事件是否要升级，如果是，应该如何升级。这很可能是扩展的实现，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-一个事件向上caster的一对多实现。除了doUpcast函数返回流而不是单个intermediateventrepresentation之外，它基本上与单个eventupcaster相同。因此，这个向上转换程序允许您将单个事件转换为多个事件。例如，如果您发现希望从fat事件中获得更细粒度的事件，那么这可能会很有用。
3. ContextawResingleEventUpcaster—upcaster的一对一实现，它可以存储过程中事件的上下文。在canUpcast和doUpcast旁边，上下文感知的upcast需要一个实现buildContext函数的函数，该函数用于实例化在经过upcaster的事件之间传递的上下文。canUpcast和doUpcast函数接收上下文作为第二个参数，位于IntermediateEventRepresentation旁边。然后，可以在升级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，它允许您将字段从一个事件移动到完全不同的事件。
4. ContextawreeventMultiUpcaster—一对多的upcaster实现，它可以存储过程中事件的上下文。这个抽象的实现是EventMultiUpcaster和contextawresingleventurepcaster的组合，因此服务于保持IntermediateEventRepresentations的上下文以及将一个这样的表示向上转换为多个这样的表示的目标。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，则此实现非常有用。
5. EventTypeUpcaster—一个完整的upcaster实现，专门用于更改事件类型。EventTypeUpcaster是SingleEventUpcaster的实现，它具有预定义的canUpcast和doUpcast函数，可以将事件从一种事件类型更改为另一种类型。例如，这可以用来轻松地更改事件的类或包名称。要创建EventTypeUpcaster，建议使用EventTypeUpcaster from（String expectedPayloadType，expectedRevision）和打字机升级版（upcastedPayloadType，upcastedRevision）方法。

5.5.2.1 写一个上转

下面的Java片段将作为一对一上转（SingleEventUpcaster）的基本示例。

事件的旧版本：

@Revision("1.0")

public class ComplaintEvent {

private String id;

private String companyName;

// Constructor, getter, setter...

}

事件的新版

@Revision("2.0")

public class ComplaintEvent {

private String id;

private String companyName;

private String description; // New field

// Constructor, getter, setter...

}

从1.0版本到2.0版本的上转：

// 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;

}

);

}

}

Springboot的设置

@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();

}

}

## **5.6事件序列化器**

事件存储需要一种序列化事件的方法来为存储做准备。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，比Java序列化更灵活。此外，XStream序列化的结果是可读的。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。除了缩短可能很长的名称的好方法外，别名还可以在事件的类定义更改时使用。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。虽然它生成了一个更紧凑的序列化表单，但它确实要求类遵守Jackson所要求的约定（或配置）。

您也可以实现自己的序列化程序，只需创建一个实现序列化程序的类，并将事件存储配置为使用该实现而不是默认实现。

### **5.6.1序列化程序实现**

序列化程序在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Axon应用程序中的消息（命令/查询/事件）、令牌、快照和saga。

由于要序列化多个对象，通常需要选择哪个序列化程序处理哪个对象。为此，配置API允许您定义默认、消息和事件序列化程序，这将导致以下对象序列化中断：

1. 事件序列化程序负责反序列化事件消息。事件通常长时间存储在事件存储中。这是选择事件序列化程序实现的主要驱动程序。
2. 消息序列化程序负责对命令和查询消息（在分布式应用程序设置中使用）进行反序列化。消息在节点之间共享，通常需要具有互操作性和/或紧凑性。在选择消息序列化程序实现时，请考虑这一点。
3. 默认的序列化程序负责反序列化其余部分，即令牌、快照和saga。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具有某些getter和setter，而这些getter和setter通常是基于Jackson的序列化程序所必需的。像XStream这样的灵活的通用序列化程序非常适合于此目的。

默认情况下，所有三种序列化程序风格都设置为使用XStreamSerializer，它在内部使用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的主要优点是能够序列化几乎任何内容。当存储令牌、saga或快照时，这种冗长通常是可以的，但是对于消息（尤其是事件），由于XML的序列化大小，它的成本可能太高。因此，出于优化的原因，您可以为消息配置不同的序列化程序。定制序列化程序的另一个非常有效的原因是实现不同（Axon）应用程序之间的互操作性，其中接收端可能强制执行特定的序列化格式。

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，则事件反/序列化将由消息序列化程序执行。反过来，如果没有配置消息序列化程序，则默认序列化程序将担任该角色。

请参阅以下示例，了解如何具体配置每个序列化程序，如果我们将XStreamSerializer用作默认值，并将JacksonSerializer用于所有消息：

Axon部分：

public class SerializerConfiguration {

public void serializerConfiguration(Configurer configurer) {

// By default we want the XStreamSerializer

XStreamSerializer defaultSerializer = XStreamSerializer.defaultSerializer();

// But for all our messages we'd prefer the JacksonSerializer due to //JSON its smaller format

JacksonSerializer messageSerializer = JacksonSerializer.defaultSerializer();

configurer.configureSerializer(configuration -> defaultSerializer)

.configureMessageSerializer(configuration -> messageSerializer)

.configureEventSerializer(configuration -> messageSerializer);

}

}

Springboot的properties文件配置

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

# and `jackson`.

axon.serializer.general

axon.serializer.events

axon.serializer.messages

springboot的yml文件配置

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

# and `jackson`.

axon:

serializer:

general:

events:

messages:

### **5.6.2事件序列化器对比**

与Axon需要序列化的所有其他对象（如命令、快照、saga等）相比，可以使用不同的序列化程序来存储事件。虽然XStreamSerializer几乎可以序列化任何东西，这使得它成为一个非常不错的默认值，但它的输出并不总是一个便于与其他应用程序共享的表单。JacksonSerializer可以创建更好的输出，但是需要对象中的某种结构来序列化。此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。

使用配置API，您可以简单地注册事件序列化程序，如下所示：

Configurer configurer = ... // initialize

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

如果未配置显式事件序列化程序，则使用已配置的主序列化程序（默认为XStreamSerializer）序列化事件。

### **5.6.3内容类型转换器**

向上转换程序处理给定的内容类型（例如dom4j文档）。为了在上传者之间提供额外的灵活性，链接的上传者之间的内容类型可能会有所不同。Axon将尝试使用ContentTypeConverter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到请求的上caster。出于性能原因，只有在接收上变频器上的canUpcast方法生成true时才会执行转换。

ContentTypeConverter可能取决于使用的序列化程序的类型。尝试将字节[]转换为dom4j文档没有任何意义，除非使用了以XML形式编写事件的序列化程序。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[]或将字节[]转换为InputStream的转换器）以及在序列化程序上配置的转换器，这些转换器将用于反序列化消息。这意味着，如果使用基于JSON的序列化程序，就可以在JSON特定格式之间进行转换。

*小贴士*

*要获得最佳性能，请确保同一链（其中一个输出是另一个输入）中的所有升级程序都在同一个内容类型上工作。*

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

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

### **5.6.4序列化程序调整**

当序列化过程证明没有达到预期的水平时，可以考虑几件事。

①XStreamSerializer

XStream是非常可配置和可扩展的。如果您只使用简单的XStreamSerializer，就有一些立竿见影的好机会。允许您配置XStream的类名和别名。别名通常要短得多（尤其是当包名较长时），使事件的序列化形式更小。因为我们讨论的是XML，从XML中删除的每个字符都能获得两倍收益（一个用于开始标记，一个用于结束标记）。

XStream中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，但不能生成最紧凑的XML。一定要仔细查看生成的XML，看看是否真的需要所有的信息来重建原始实例。

尽可能避免使用上转。XStream允许在字段更改名称时使用别名。想象一下事件的修订版0，它使用了一个名为“clientId”的字段。业务部门更喜欢使用“客户”一词，因此修订版1是用一个名为“customerId”的字段创建的。这可以完全在XStream中使用字段别名进行配置。您需要按以下顺序配置两个别名：将“customerId”别名为“clientId”，然后将“customerId”别名为“customerId”。这将告诉XStream，如果遇到名为“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遇到一个名为“clientId”的XML元素，则它是一个已知别名，并将被解析为字段名“customerId”。有关更多信息，请查看XStream文档。

为了获得最终的性能，最好不要使用基于反射的机制。在这种情况下，创建自定义序列化机制可能是最明智的做法。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。

②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，并且支持序列化消息。如果检测到序列化感知消息，则使用其方法序列化对象，而不是简单地将负载传递给序列化程序。这允许性能优化。

当您自己序列化消息并希望从SerializationAware优化中获益时，请使用MessageSerializer类来序列化消息的有效负载和元数据。所有优化逻辑都在该类中实现。有关更多详细信息，请参阅MessageSerializer的JavaDoc。

③事件的不同序列化程序

使用事件源时，序列化事件可能会持续很长时间。因此，请仔细考虑它们序列化到的格式。考虑为事件配置单独的序列化程序，仔细优化事件的存储方式。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EventSerializer配置为其他类型的详细信息，请查看有关序列化程序的文档。

④宽松反序列化

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属性。如果它处理的是要反序列化的格式，那么当它无法在序列化格式中为给定字段找到字段/setter/constructor参数时，它不会失败。

启用宽松序列化对于适应不同的消息版本特别有用。当使用事件存储时，这种情况自然会发生，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多个不同版本同时运行，则在命令和查询之间也可能发生这种情况。当您使用滚动升级模式来部署新服务时，会遇到这种情况。

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。下面的代码片段显示了如何实现这一点：

XstreamSerializer：

public class SerializerConfiguration {

public Serializer buildSerializer() {

return XStreamSerializer.builder()

.lenientDeserialization()

.build();

}

}

JacksonSerializer：  
public class SerializerConfiguration {

public Serializer buildSerializer() {

return JacksonSerializer.builder()

.lenientDeserialization()

.build();

}

}