### **1框架概述**

基于Axon的应用程序应遵循一种架构模式，该架构基于领域驱动设计（DDD），命令查询责任隔离（CQRS）和事件驱动架构（EDA）的原理。这些原则的结合使基于Axon的应用程序更加健壮，而且可以适应我们业务领域的变化所要求的变化。

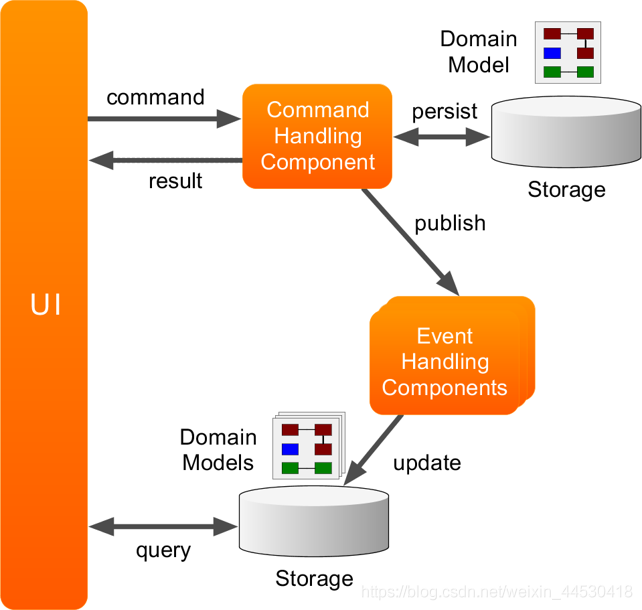
### **1.1.2DDD和CQRS**

域驱动设计（DDD）描述了一种构建软件的方法，该方法非常重视模型的设计，并利用了普适性的语言。域模型是软件的核心，应该正确地构建，同时也需要处理域的基本复杂性。

命令查询责任隔离（CQRS）是一种体系或者说结构模式，它描述了应用程序中处理命令（请求更改应用程序状态）的部分与回答查询部分（请求有关应用程序状态的信息）之间的区别。

在把DDD和CQRS组合在一起时，我们实际上是将一个应用程序划分为多个组件，其中每个组件要么提供有关应用程序状态的信息，要么更改应用程序的状态。每个组件都有一个专注于这些职责的模型。

下图显示了基于Axon的应用程序的典型体系结构。



在这样的体系结构中，UI（或API）可以发送命令以请求更改应用程序的状态。这些命令由命令处理组件处理，该组件使用模型来验证命令并决定要顺带触发哪些边缘行为（如果有）。

这些顺带的边缘行为是通过“事件”引发的命令来产生的。这些事件由采取适当操作的一个或多个事件处理组件来拾取。其中一个典型的操作是更新视图模型，这允许UI呈现应用程序的状态。其他操作可能是向外部组件发送消息，甚至可能通过新命令触发其他副作用。

命令模型和查询模型（也称为视图模型或投影）的分离使这些模型仅专注于应用程序的特定方面。这使得各个单独的模型更容易理解，因此从长期来看更易于维护。

### **1.1.3业务逻辑和基础架构分离**

在基础结构和业务逻辑混在一起时，很容易就会一不留神出现意外的错误。Axon的当务之急是将其进行严格的区分。Axon的设计使以下两点有了明显的区别：你想做的事（如发布的事件）和这件事实际上是怎么做的（如发布事件的实施方法）。

这使Axon可以根据您的特定情况进行配置和调整。更重要的是，它可以将复杂性降至最低。例如，尽管Axon使实现事件源聚合变得容易，但绝不将聚合强制为事件源，Repository接口就完全践行了这一决策。同样，决定通过命令总线发送命令的组件不会去管如何将该消息传输到处理程序。

Axon不仅通过为组件提供清晰的接口来进行这种分离，而且还结合了Configuration API中的基础结构选择，那里的业务逻辑组件与应用程序的基础结构部分是分开配置的。

### **1.1.4显式消息**

Axon强烈建议使用显式消息对象。这意味着基于Axon的应用程序中的每个消息通常将由该应用程序中的特定Java类表示。尽管这样做确实会在编写基于Axon的应用程序时带来一些开销，但它确实具有一些优点：

1.使用显式消息可以更轻松地将它们透明地分发到远程组件。

2.显式消息的使用强调消息设计，事实证明，消息设计对应用程序的长期可维护性很重要。

3.显式消息可以轻松存储以供以后处理

虽然消息传递是Axon的核心概念，但并非所有消息都是一样的。不同的意图需要不同的路由模式。例如，对于某些消息，一个人会期望结果，而其他消息天生就是一劳永逸的。

Axon将消息大致分为三类：

****命令****; 表达改变应用程序状态的意图。命令被路由到单个目的地，并且可以提供响应。

****查询****; 表达对信息的渴望。根据调度策略，查询可能会同时路由到一个或多个目的地。

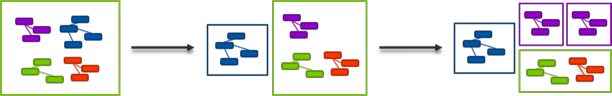
****日志****; 表示发生了相关事件的通知。事件将发布到任何感兴趣的组件，并且不提供任何形式的返回值。

4.位置透明

使用显式消息的最大好处是，相互交互的组件不需要知道其对应对象的位置。实际上，在大多数情况下，发送组件甚至对消息的实际目的地都不感兴趣。我们称此为“位置透明度”。

与将服务放在逻辑URL后面相比，Axon使位置透明性进一步提高。在Axon中，发送消息的组件不需要为该消息指定目的地。根据消息的构造型（命令，查询或事件）和消息携带的有效负载类型来路由消息。Axon使用应用程序的功能自动为邮件找到合适的目的地。

由位置透明组件组成的系统使该系统具有高度的适应性。例如，由单独使用命令，事件和查询进行通信的，分隔良好的组件构成的整体系统可以轻松拆分为单独部署的单元，而不会影响功能



通过位置透明度实现微服务演进

这使Axon非常适合微服务环境。逻辑可以轻松地在已部署的组件之间来回移动，而不会影响整个系统的功能。然后，可以根据该系统每个单独组件的非功能性需求来主要确定逻辑的位置。例如，可以将具有明显不同的性能特征的组件或需要不同发布周期的组件从整体应用程序中分离出来，以减少对该组件进行更改的影响。

### **1.1.5事件回溯**

在许多系统中，事件得到了很多额外的关注。尽管Axon明确承认并非每条消息都是一个事件（还有命令和查询），但事件还是有一些特殊之处。

事件存在价值。当命令和查询触发边缘行为或提供结果时，其价值会显着降低，而事件代表已发生的事情，这对于在事件发生后的很长时间内了解可能很有用。

事件为审核跟踪提供了非常好的粒度。但是，要使审核跟踪具有100％的可靠性，它不仅应作为边缘行为而产生，还必须确保审核跟踪正确反映了所有决策。

事件回溯是一个过程，其中事件不仅作为命令的边缘行为而产生，而且还导致了当前对象的状态。尽管应用程序的当前状态未明确存储在数据库中，但它可以作为一系列事件隐式存储，可用于导出当前状态。收到命令后，将从存储在数据库中的事件中动态得出应用程序的状态，然后确定要应用哪些边缘行为。

实施事件回溯可能非常复杂。Axon提供了使构建命令模型变得非常容易甚至更为自然的方法所必需的API。Axon的测试功能有助于确保正确遵循某些准则和要求。

拥有可靠的回溯不仅加强了系统的可验证性，而且还提供了构建新视图模型，进行数据分析并为机器学习算法提供坚实基础。

## **1.2DDD和CQRS**

Axon很大程度上基于域驱动设计（DDD）和命令查询责任隔离的原则。尽管对这些概念的完整解释超出了本参考指南的范围和意图，但我们确实希望提供有关Axon应用程序上下文中最重要概念的摘要。

### **1.2.1战略概念**

战略概念是相对较高的概念，它们在体系结构级别上影响系统的设计。它提供了设计组件边界及其之间相互作用的概念。尽管它们不直接影响单个Axon应用程序的设计，但此类概念通常会影响此类应用程序的边界。

### **1.2.2领域和子域**

在DDD上下文中，领域的正式定义是：

知识、影响或活动的范围，用户在这个范围内会运行一个以软件为主体的程序。

这个定义可能看起来很模糊，但确实很好地抓住了本质。域基本上是构建软件的环境。该环境由法律，最佳实践，期望，约定等组成，它们定义了什么是重要的，什么不是重要的。

域可能非常大，并且域内的不同区域可能会产生不同的影响。例如，在银行领域，个人银行，企业银行和理财产品销售之间存在明显的区别。

有很多技术可以帮助您探索一个域。[事件风暴](https://www.eventstorming.com/book/)就是其中特别有趣的一种。它具有研讨会的形式，用于快速探索复杂的业务领域。

### **1.2.3模型**

一个模型是：

一种描述领域选定方面的抽象系统，可用于解决与该领域相关的问题；

换句话说，模型捕获了对于我们解决领域内特定问题的重要信息。这个定义本身表明一个应用程序应该包含多个模型，因为每个问题都有一个不同的理想模型来解决。

有关基于Axon的应用程序中模型的某些构造块的更多信息，请参见[战术概念](https://docs.axoniq.io/reference-guide/architecture-overview/ddd-cqrs-concepts" \l "tactical-concepts)。

### **1.2.4有界上下文**

上下文是：

用单词或陈述出现的设置来确定其含义。

换句话说，相同的领域概念对不同的人可能具有不同的含义。

例如，考虑一次飞行的概念。对于乘客而言，飞行是指从飞机起飞到到达目的地之间的时间段。但是，地勤人员关心航班到达登机口的次数，要登上飞机的饭菜，枕头等的数量，这些事情在航班离开登机口后就完成了。对他们来说，出发时间是最后期限，而不是起点。

因此，对于模型和上下文有许多规则：

* 明确定义应用模型的上下文。
* 在团队组织，应用程序特定部分内的用法以及诸如代码库和数据库模式之类的物理表现方面，明确设置边界。
* 使模型在这些范围内严格保持一致，但不要过多关注模型之外的问题

### **1.2.5上下文映射**

有界的上下文永远不会独立出现。来自不同上下文的信息最终将被同步。对这个交互显式建模是很有用的。域驱动设计列出了上下文之间的一些关系，这些关系决定了它们之间的交互方式：

* 伙伴关系（两个环境/团队共同努力建立互动）
* 客户-供应商（具有上游/下游关系的两个团队-上游可以独立于下游团队获得成功）
* 服从关系（上游/下游关系中有两个团队-上游没有动力向下游提供，下游团队也没有努力索要）
* 共享内核（明确共享模型的一部分）
* 分开的方式（将它们切开）
* 反泄露层（下游团队构建一层，通过转换交互来防止上游设计“泄漏”到自己的模型中）

在基于Axon的应用程序中，上下文定义事件在其中携带价值的边界。有些事件可能仅在其发布的上下文中才有价值，而其他事件甚至在外部也可能有价值。发布事件（或任何消息，就此而言）的范围越广，最终有更多的组件耦合到发送者。

### **1.2.6战术概念**

为了构建模型，DDD（在某种程度上还包括CQRS）提供了许多有用的构建块。下面是一些在基于Axon的应用程序中很重要的构建块。

1.2.6.1聚合

聚合是始终保持一致状态（在单个ACID事务内）的实体或实体组。聚合根是聚合中负责维护此一致状态的实体。这使聚合成为在任何基于CQRS的应用程序中实现命令模型的主要构建块。

DDD的正式定义是：

关联对象的群集，出于数据更改的目的，它们被视为一个聚合。外部引用仅限于聚合的一个对外成员，这个成员称为根。整组一致性规则适用于聚合的边界。

在基于CQRS的应用程序中，聚合在命令模型中非常明确地存在，因为这是启动更改的地方。但是，查询模型/映射也是由聚合建立的。但是，通常，查询模型中的聚合要简单得多，因为状态一致性在这些模型中通常不那么严格。

1.2.6.2 Saga

并非每个命令都能在单个原子事务中完全执行。汇款是交易中经常出现的一个非常常见的例子。人们通常认为，绝对必要的原子交易是将钱从一个帐户转移到另一个帐户。好吧，不是。相反，这是完全不可能的。如果资金从银行A上的一个帐户（总计A的实例）转移到银行B上的另一个帐户（总计B的实例），该怎么办？银行A是否获得银行B数据库的锁？如果转帐正在进行中，银行A扣除了这笔金额，但银行B尚未存入，这很奇怪吗？不完全是，它正在进行中。另一方面，如果在将钱存入银行B的帐户时出了点问题，银行A的客户会希望退回他的钱。因此，我们确实希望最终会有某种形式的一致性。另一个示例可能是在下订单后开始的示例。它将确保成功兑换礼品卡后，在另一侧确认订单。

尽管在某些情况下ACID交易不是必需的，甚至是不可能的，但仍需要某种形式的交易管理。通常，这些事务称为BASE事务：基本可用，软状态，最终一致性。与ACID相反，BASE事务无法轻松回滚。要回滚，需要采取补偿措施以还原在事务中发生的任何事情。在礼品卡示例中，如果兑换失败，将拒绝付款。

在CQRS中，可以使用Saga来管理这些BASE事务。它们响应事件并可以调度命令，调用外部应用程序等。在域驱动设计的上下文中，通常将Saga用作不同聚合（或聚合实例）之间的协调机制，以实现最终一致性。

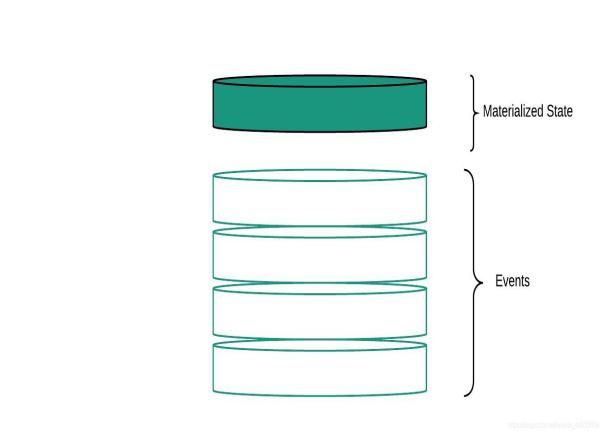
视图模型或投影

在CQRS中，视图模型（也称为投影或查询模型）用于有效地公开有关应用程序状态的信息。与命令模型不同，视图模型着重于数据而不是行为。视图模型通常是为适应特定受众的信息需求而建模的。这些模型应清楚地呈现在目标受众前，以防止“分心”和范围模糊，最终导致可维护性甚至性能的损失。

## **1.3事件回溯**

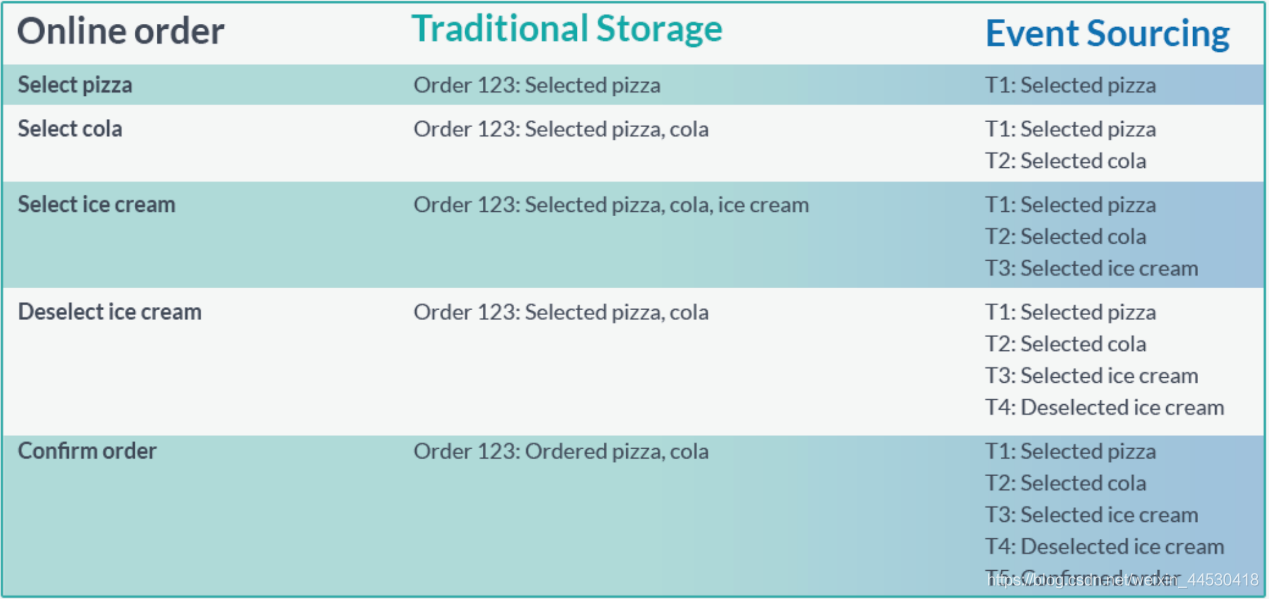
传统的存储应用程序状态的方式，是我们捕获当前状态并将其存储在某些关系数据库或[NoSQL](https://en.wikipedia.org/wiki/NoSQL)数据库中。尽管此方法确实非常简单，但它并未提供一种推断我们如何达到当前状态的方法。当然，有人可能会争辩说，我们可以有一个单独的模型来保存导致当前状态的动作的历史记录，但是除了额外的复杂性之外，这两个模型可能很容易走不同的道路并开始变得不一致

事件回溯是一种通过过去发生的事件的历史记录存储应用程序状态的方法。当前状态是根据事件的全部历史记录重建的，其中每个事件代表我们应用程序中的更改或事实。事件为我们提供了有关应用程序中发生的情况的唯一事实来源。在需要向外部审阅者提供完整审核日志的应用程序中，这特别有益。在某些资源中，当前状态称为实现状态。



事件是构建状态的来源

让我们看一个示例，事件回溯与传统存储有何不同（见图2）。在传统存储系统中，我们仅知道我们已订购比萨饼和可乐。在“事件采购”中，我们看到用户选择了一个比萨饼，一个可乐，一个冰淇淋并取消了一个冰淇淋。传统存储中不存在有关冰淇淋选择/取消选择的信息。借助事件采购，我们可以推断用户为什么取消选择冰淇淋，价格太高或其他原因。关键是我们不会丢失这些信息，我们可以通过各种方式从中受益。稍后我们可以看到用户确认了订单。



### **1.3.1事件存储**

事件回溯需要一个事件存储来存储事件。由于事件是不可修改的（事件是确实发生的且不能被修改的事实），因此应针对附件优化事件存储。事件排序在基于事件的系统中起着非常重要的作用-在我们重构对象状态的过程中，我们希望获得相同的结果。

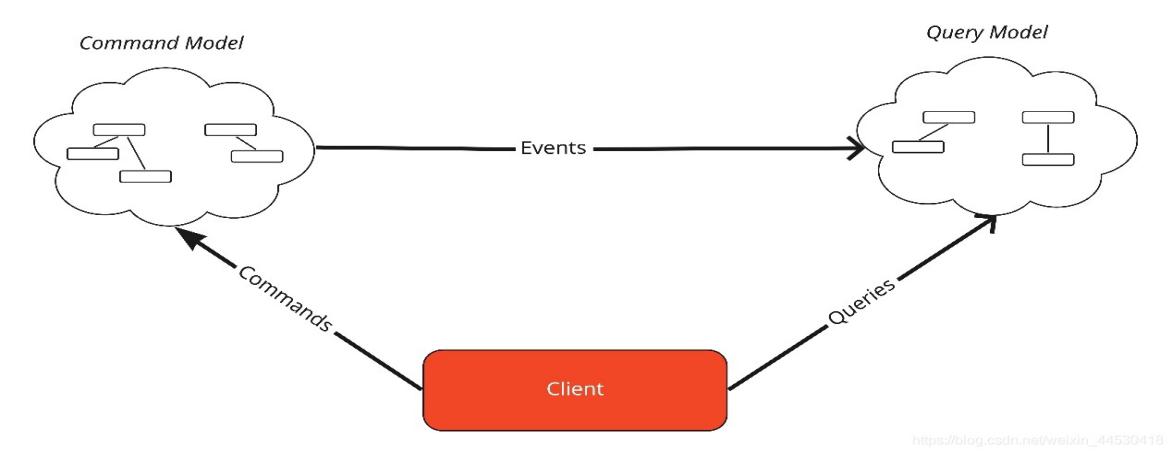
[Axon服务器](https://blog.csdn.net/reference-guide/axon-server-introduction)是在axon中默认的选择，它提供了企业级的专用事件存储，其高度用于存储/检索的事件进行了优化。该服务器有标准版或企业版。

另外，Axon框架提供对RDBMS或NoSQL数据库作为事件存储的支持。

### **1.3.2事件回溯和CQRS**

事件回溯与[CQRS](https://axoniq.io/resources/cqrs)天生的契合。通常，我们不存储基于CQRS的体系结构中的命令模型，而是存储其事件序列。基于这些相同的事件，查询模型将不断更新以包含当前状态的特定表示形式。

与其重建整个命令模型状态（这将是一个漫长的过程），我们将模型分成多个聚合；模型中需要高度一致的部分。聚合中的这种分离使模型更易于推理，更易于更改，更重要的是，它使应用程序更具可伸缩性。



好处

下面列出了事件回溯的好处

* 使审查更自然明了

为了遵守某些规定，软件系统需要提供完整的审核日志。基于事件的系统为我们提供了完整的审核日志，而且我们不必向审阅者提供任何其他信息。根据事件流生成一份报告就行了。

* 有利于算法分析

与我们的应用程序交互的完整历史记录存储在事件存储中。我们可以应用各种机器学习算法来从这些对我们的业务至关重要的交互中提取信息。

③设计灵活性

构建软件系统的敏捷方法要求我们应该能够适应此过程中发生的任何变化。能够从一开始就使用新的业务逻辑重播事件流，这意味着我们不必担心自己做出的决定（除了哪些事件对于存储很重要），我们以后可以随时修复行为。在事件流中引入新的视图意味着向解决方案中添加一个带有事件侦听器的新组件。

④ 时间报告

我们都知道调查生产中发生的事件有多么困难。它需要大量日志来挖掘和推理系统当时的状态。事件回溯提供了一种方法，可以将事件回滚到某个时间点，并在事件发生的状态下调试应用程序。我们不必担心我们设置正确的日志级别还是我们是否记录了所有必要的路径以找出事件。

### **1.3.3结论**

事件回溯是一种避免我们重蹈覆辙的自然的解决方案。事件回溯的主要优势还有它增加了软件系统的可扩展性-当我们发现需要添加的新报告组件时，我们可以将其编写，回滚历史事件并使其运行。这对于零停机的蓝绿部署非常有用。

它还可以使用事件与外部系统集成。在这种情况下，事件是我们的API，外部系统必须了解它们。当然，我们可能不会将所有事件都发布到集成事件中心，而仅将那些重要的事件发布到集成事件中心。

\*如果我们正在应用CQRS（命令查询责任隔离）实践，那么我们也可以重建命令模型和查询模型

[Axon服务器](https://blog.csdn.net/reference-guide/axon-server-introduction)提供了一种简单的方法来启动和规模化事件来源的Java应用程序。

## **1.4事件驱动的微服务**

在设计和创建（事件驱动）微服务系统时，[DDD和CQRS概念](https://docs.axoniq.io/reference-guide/architecture-overview/ddd-cqrs-concepts)一章中描述的[概念](https://docs.axoniq.io/reference-guide/architecture-overview/ddd-cqrs-concepts)非常适用。在本章中，我们将明确列出在此类环境中应用Axon的几种常见策略。

### **1.4.1进化微服务**

在AxonIQ，我们相信系统会逐渐发展为微服务，而不是尝试从头开始构建微服务系统。主要原因是探索合理的上下文边界（请参阅[Bounded Context](https://docs.axoniq.io/reference-guide/architecture-overview/ddd-cqrs-concepts" \l "bounded-context)）和模型需要时间。与在整体系统中相比，在分布式系统中更改这些边界要困难得多。

Axon利用组件的分离并在它们之间使用显式消息传递，这使这些组件的[位置透明](https://docs.axoniq.io/reference-guide/architecture-overview" \l "location-transparency)。与使用服务发现不同，Axon用于消息传递的方法完全不需要组件知道消息的目的地。它们会自动路由到一个广告组件，以宣传处理此类消息的功能。这使得这些系统比“常规”基于微服务的系统更灵活地进行更改。

### **1.4.2应用Axon的策略**

在微服务环境中应用Axon有不同的策略。可以在系统级别采用Axon哲学，并使用Axon构建所有服务。但是，仅在单个应用程序/服务中应用Axon时，它也已经很有用。最后，我们还将讨论在多语言环境中使用Axon时的特定策略。为此，Axon在构建时就考虑了集成。

### **1.4.3基于Axon的微服务**

在系统级别使用Axon时，这意味着有多个服务在运行Axon（或兼容的API），一个人可以最大程度地使用消息传递概念。应用程序可以简单地利用不同的消息总线来发送和接收来自其他组件的消息。在更改组件的部署策略时，这使系统非常灵活。

### **1.4.4仅在单一服务中使用Axon**

在现有的微服务系统中构建单个基于Axon的服务时，您可能希望使用“传统”的休息端点公开您的API。在这种情况下，基于Axon的应用程序将需要一个小的API层，它将REST调用转换为命令，然后将其分派到内部的命令总线。但是请务必考虑到，请求可能不会被一致地路由，并且用于同一聚合的命令可能会被路由到不同的实例。

如果基于Axon的服务已部署了多个实例，则仍可以使用总线的分布式实现来受益，以允许这些实例在它们之间适当地平衡消息处理。

### **1.4.5混合/多语种环境**

实际上，许多基于微服务的系统都在多语言环境中运行。不同的服务将在不同的技术堆栈上运行。在这些环境中，更重要的是要确保适当地保护上下文边界并在适用的情况下提供体面的反腐败层。

所有使用过的技术堆栈都不太可能遵循与Axon应用程序相同的基于消息传递的方法。但是，这并不意味着需要放弃这些概念。您仍然可以从显式消息传递的许多优点中受益。在这样的环境中，反腐败层可以实现为处理命令，事件和查询，并执行对外部服务的其他类型的调用（例如REST调用）的组件。这样，使用显式消息传递的组件就不必担心轮询外部服务是否发生更改，也不必担心由不同类型的API引起的技术挑战。

Axon支持不同类型的连接器，这些连接器允许将事件（在某些情况下还包括其他消息类型）发布到第三方消息代理。默认情况下，Axon将对这些外部事件的格式进行假设，但始终可以覆盖它们。

## **1.5 Axon服务器**

在消息驱动的微服务环境中，服务之间的通信高效，可靠且易于管理和监视非常重要。消息路由不需要任何手动配置，添加新服务应该很容易。

使用Axon Framework，对于应用程序开发人员而言，内部通信是隐藏的。Axon Server在分布式环境中提供了相同的体验。它是一个易于使用，易于管理的平台，可以处理所有事件，命令和查询。

Axon Server知道有关正在交换的消息及其类型的信息：从一种服务发送到一项或多项其他服务的事件，以通知服务某些事情已经发生；命令，发送到一项服务以执行某件事；可能等待结果查询，发送到一个或多个服务以检索信息。

这些消息中的每条消息都需要不同的策略，Axon Server均支持所有策略。

应用程序连接到消息传递平台并注册其功能。一个应用程序可能能够执行一组特定的命令，另一个应用程序可以处理许多查询。同一应用程序可能有多个实例连接到消息传递平台，每个实例具有相同的功能。

### ****1.5.1讯息模式****

客户端或应用程序将请求发送到消息传递平台。平台找到适当的已连接应用程序以将请求发送到该应用程序，然后转发该请求。它接收一个或多个答复，并将其转发给呼叫者。命令总是发送到恰好一个应用程序。相同聚合的命令始终发送到同一应用程序实例，以避免聚合的并发更新出现问题。查询将发送到所有能够回答查询的应用程序。如果同一应用程序有多个实例，则查询仅发送到实例之一。事件则存储在事件存储中，并发送给所有注册的监听器。

### ****1.5.2高可用****

Axon Server可以在群集模式下运行。群集中的每个节点都是活动的，并且应用程序在节点上动态地进行负载均衡。同一应用程序的实例将连接到同一消息传递节点以优化性能。不同应用程序的实例分布在节点上以分散负载。

### ****1.5.3流量控制****

Axon Server控制发送到消息处理程序的消息流。消息处理程序向消息传递平台发送了许多许可，指示消息传递平台可以发送的消息数。一旦处理程序准备好进行更多请求，它就会发送另一条带有大量附加许可的消息。如果没有允许的处理程序，则Axon Server会将消息排队。当仍然有排队的消息断开处理程序时，这些消息将重新路由到另一个处理程序（如果可能）。

### ****1.5.4QoS消息****

客户可以指示其请求的优先级。这样，例如，与在线请求相比，可以以较低的优先级执行源自批处理过程的命令。

### ****1.5.5访问控制****

需要授予应用程序对消息传递平台的访问权限。这避免了随机客户端可以开始向系统发送命令的风险。

### ****1.5.6实施技术****

Axon Server已在Spring Boot的基础上完全用Java开发。它作为单个jar文件分发，其中包含通过着色使用的所有库。

Axon Server的配置信息存储在小型h2数据库中。这包含有关消息传递平台节点和具有访问权限的应用程序的信息。此信息将自动在群集中的节点之间复制。

消息传递平台具有两种类型的接口：

* HTTP
* gRPC（HTTP 2.0）

标准Axon Framework Axon Server客户端和消息传递平台之间的通信使用gRPC。

# **三、消息的概念**

## **3.1消息传递**

消息传递是Axon的核心概念之一。组件之间的所有通信都是使用消息对象完成的。这使这些组件具有位置透明性，以便在必要时能够缩放和分发这些组件。‌

尽管所有这些消息都实现了接口，但是在不同类型的消息及其处理方式之间存在明显的区别。

所有消息都包含有效负载，元数据和唯一标识符。消息的有效负载是消息含义的功能描述。该对象的类名及其携带的数据的组合描述了应用程序对消息的含义。元数据可让您描述发送消息的上下文。例如，您可以存储跟踪信息，以允许跟踪消息的来源或原因。在命令被执行时，您还可以存储信息以描述这个命令的安全上下文。

[Axon编码教程＃2： -核心API](https://youtu.be/vnCxjWZrrk0)

****注意****

*请注意，所有消息都是不可变的。在消息中存储数据实际上意味着在前一条消息的基础上创建一条新消息，并添加了更多信息。这保证了在多线程和分布式环境中可以安全地使用消息。*

命令

命令描述了更改应用程序状态的意图。它们被实现为（最好是只读的）pojo，使用CommandMessage的实现之一包装这些pojo。

命令总是只有一个目的地。虽然发送方不关心哪个组件处理命令或该组件驻留在何处，但了解它的结果可能很有趣。这就是为什么通过命令总线发送的CommandMessage允许返回结果。

日志

事件是描述应用程序中发生的事情的对象。典型的事件来源是聚合。当聚合中发生重要事件时，它将引发一个事件。在Axon Framework中，事件可以是任何对象。强烈建议您确保所有事件都可序列化。‌

调度事件时，Axon将它们封装在EventMessage中。实际使用的消息类型取决于事件的来源。当聚合引发事件时，它被包装在DomainEventMessage中（它扩展了EventMessage）。所有其他事件都封装在EventMessage中。除了常见的消息属性（如唯一标识符）外，EventMessage还包含时间戳。DomainEventMessage还包含引发事件的聚合的类型和标识符。它还包含聚合的事件流中事件的序列号，这允许复制事件的顺序。

****注意****

*即使域事件信息包含对聚合标识符的引用，也应始终将标识符包含在实际事件本身中。域事件信息中的标识符由事件存储器用于存储事件，它并不一定总是为其他目的提供可靠的值‌*

原始的事件对象存储为EventMessage的有效负载。在有效负载旁边，您可以将信息存储在EventMessage的元数据中。元数据的目的是存储有关事件的其他信息，这些信息主要不旨在用作业务信息。审计信息就是一个典型的例子。它使您可以查看在什么情况下引发了事件。例如触发处理的用户帐户，或处理事件的计算机的名称。

****注意****

*通常，您不应基于事件消息的元数据中的信息来进行业务决策。如果真是这样，您可能已经附加了实际上应该属于事件本身的信息。元数据通常用于报告，审核和跟踪。*

尽管没有强制执行，但最好的做法是使域事件不可变，最好是使所有字段都为final，并在构造函数中初始化事件。如果事件构造太麻烦，请考虑使用Builder模式。

****注意****

*尽管域事件从技术上指示状态更改，但您也应该尝试捕获事件中状态的意图。一个好的实践是使用域事件的抽象实现来捕获特定状态已更改的事实，并使用该抽象类的具体子实现来指示更改的意图。例如，您可能有一个抽象类（AddressChangedEvent）和两个实现（ContactMovedEvent和AddressCorrectedEvent），它们捕获了状态更改的意图。一些监听器不在乎意图（例如，数据库更新事件监听器）。这些将监听抽象类型。其他监听器确实关心此意图，这些监听器将侦听具体的子类型（例如，向客户发送地址更改确认电子邮件）。*

在事件总线上调度事件时，需要将事件包装在事件消息中。GenericEventMessage是一个实现，允许您将事件包装在消息中。可以使用构造函数或静态asEventMessage（）方法。后者检查给定参数是否尚未实现消息接口。如果是这样，它要么直接返回（如果它实现EventMessage，），要么使用给定消息的有效负载和元数据返回一个新的GenericEventMessage。如果某个事件被聚合应用（发布），Axon将自动将该事件包装在包含聚合标识符、类型和序列号的DomainEventMessage中。

查询

查询描述了对信息或状态的请求。一个查询可以有多个处理程序。调度查询时，客户端会指示他是从一个查询还是从所有可用查询处理程序中获取结果。

## **3.2 对消息的剖析**

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

消息的每个子接口表示一种特定类型的消息，并定义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，可以有效地创建一个新的消息实例，其中包含一个额外的（或其他）元数据元素。为了仍然能够将消息的两个Java实例视为表示同一个概念性消息，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。

元数据

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该消息生成的消息的信息（如命令处理程序根据传入的命令生成事件）。

在Axon中，元数据表示为字符串到对象的映射。尽管您可以随意添加任何类型的对象作为“元数据”值，但我们强烈建议您坚持使用基本体和字符串（或最多是简单的对象）。在进行结构更改时，元数据具有与有效负载不同的灵活性。

与常规的Map<String,Object>不同，元数据在Axon中是不可变的。变异方法将创建并返回一个新实例，而不是修改现有实例。

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

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

//第一行用指定的键值对创建实例，第二行添加额外的键值对，返回的新实例。

元数据在消息中的行为方式是相似的：

EventMessage eventMessage =

        GenericEventMessage.asEventMessage("myPayload") // 1

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

        .andMetaData(singletonMap("otherKey", "some value")); // 3

/\*第一行创建一个以“myPayload”作为有效负载的EventMessage

第二行的withMetaData用给定的映射替换消息中的任何元数据，在这种情况下java.util.Collections.singletonMap（）用于定义单个条目。

第三行andMetaData将给定映射中的条目添加到消息的元数据中。具有相等键的现有条目将被覆盖。

\*/

注意：元数据还实现了Map<String，Object>，这意味着除了传递单个映射（或任何其他类型的Map），还可以使用MetaData.with（键，值）。但由于元数据不可变，所有的操作都会抛出UnsupportedOperationException。

特定消息的数据

某些类型的消息（message）提供额外的信息（在消息提供的信息之上）。例如，EventMessage（接口扩展消息）还提供时间戳，表示事件生成的时间。除了有效负载和元数据之外，QueryMessage还包含请求组件所期望的响应类型的描述。

## **3.3消息相关**

在消息传递系统中，通常将消息分组或关联起来。在Axon框架中，命令消息可能导致一个或多个事件消息，而查询消息可能会导致一个或多个QueryResponse消息。通常，关联是使用特定的消息属性（所谓的关联标识符）来实现的。

### **3.3.1 相关消息提供者**

Axon框架中的消息使用元数据属性来传输有关消息的元信息。元数据对象的类型为Map<String，object>，并与消息一起传递。为了填充在工作单元内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填充新消息的元数据的工作单元。Axon Framework目前提供了此功能接口的一些实现方式：

消息源提供程序：默认情况下，MessageOriginProvider注册为要使用的相关数据提供程序。它负责将两个值从一个消息传递到另一个消息，即correlationId和traceId。消息的correlationId总是引用它源于的消息的标识符（即父消息）。另一方面，traceId引用启动消息链（即根消息）的消息标识符。创建新消息时，如果父消息中不存在这两个字段，则消息标识符将用于这两个字段。在一个示例中，如果您要处理一个命令消息，而该命令消息又发布了一个事件消息，则事件消息的元数据将基于以下内容填充：

****·****correlationId的命令消息标识符。

****·****命令消息在元数据中是否存在traceId，或者消息的traceId标识符。

**SimpleCorrelationDataProvider**

SimpleCorrelationDataProvider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。必须使用要复制的提供程序的SimpleDataRelativeList调用该构造函数。下面是一个示例，讲述了如何配置它以复制myId和myId2值。

public class Configuration {

    public CorrelationDataProvider customCorrelationDataProvider() {

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

    }

}

**MultiCorrelationDataProvider**

能够组合多个相关数据提供者的效果。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()

            )

        );

    }

}

* 实现自定义相关数据提供程序

如果预定义的提供程序不能满足您的需求，您可以始终实现自己的CorrelationDataProvider。该类必须实现CorrelationDataProvider接口，如下例所示：

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;

    }

}

### **3.3.2配置**

当默认的MessageOriginProvider不足以满足您的用例时，需要在应用程序中注册（自定义）相关数据提供程序。如果您使用的是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springboot自动配置，只需提供一个公开springbean的工厂方法，如果您都需要，则提供多个方法。以下片段显示了一些可能的注册方法：

①Axon配置API

public class Configuration {

public void configuring() {

Configurer configurer = DefaultConfigurer.defaultConfiguration().configureCorrelationDataProviders(

config -> Arrays.asList(

new SimpleCorrelationDataProvider("someKey"),newMessageOriginProvider()

)

);

}

}

***Spring boot配置类***

@Configuration

public class CorrelationDataProviderConfiguration {

// 配置单个CorrelationDataProvider将自动覆盖默认的MessageOriginProvider

@Bean

public CorrelationDataProvider someKeyCorrelationProvider() {

return new SimpleCorrelationDataProvider("someKey");

}

@Bean

public CorrelationDataProvider messageOriginProvider() {

return new MessageOriginProvider();

}}

## **3.4消息拦截**

拦截器有两种不同类型：调度拦截器和处理程序拦截器。在将消息调度到消息处理程序之前，将调用调度拦截器。在那时，甚至可能还不知道该消息存在处理程序。处理程序拦截器在消息处理程序被调用之前被调用。

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命令进行操作。示例是日志记录或身份验证，无论命令类型如何，都可能要执行这些操作。这是使用拦截器完成的。

命令调度拦截器

在命令总线上分派命令时，将调用消息调度拦截器。他们可以通过添加元数据来更改命令消息。他们还可以通过引发异常来阻止命令。这些拦截器总是在分派命令的线程上调用。

让我们创建一个MessageDispatchInterceptor（消息调度拦截器），它分派的每个命令都消息记录在CommandBus上。

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);

  command;

}; } }

通过执行以下操作，我们可以使用CommandBus来注册此调度拦截器：

public CommandBusConfiguration {

​ public class CommandBusConfiguration {

    public CommandBus configureCommandBus() {

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

        commandBus.registerDispatchInterceptor(new MyCommandDispatchInterceptor());

        return commandBus;

    }

}

结构验证

命令没有以正确的格式包含所有必需的信息，则没有必要处理该命令。实际上，应该尽早阻止缺少信息的命令，最好在开始事务处理之前就将其阻塞。因此，拦截器应检查所有传入的命令以获取此类信息。这称为结构验证。

Axon Framework支持基于JSR 303 Bean验证的验证。这样，您便可以使用诸如@NotEmpty和@Pattern的注释来注释命令中的字段。您需要在类路径中包含JSR 303实现（例如Hibernate-Validator）。然后，在命令总线上配置一个BeanValidationInterceptor，它将自动查找并配置您的验证程序实现。您可以根据自己的特定需求对其进行微调也可以用默认设置，使它使用合理的值。

****拦截器设定技巧****

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正前方。在某些情况下，可能需要先放置LoggingInterceptor或AuditingInterceptor，然后紧接着是validating interceptor。

BeanValidationInterceptor还实现了MessageHandlerInterceptor，允许您将其配置为处理程序监听器。

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执行操作。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，它接受两个参数：当前UnitOfWork和一个拦截器链。拦截器链用于继续调度过程。UnitOfWork提供（1）正在处理的消息，（2）提供在（命令）消息处理之前、期间或之后连接逻辑的可能性（有关阶段的更多信息，请参阅工作单元）。

拦截器与上下文处理程序不同，拦截器是在命令调度中调用的。例如，这意味着它们可以根据正在处理的消息将相关数据附加到工作单元。然后，该关联数据将附加到在该工作单元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，请注册一个TransactionManagingInterceptor，该拦截器将配置为使TransactionManager来启动和提交（或回滚）实际事务。

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字段值的命令。如果用户标识不在元数据中，则将引发一个异常，该异常将阻止处理该命令。另外，如果userId的值与axonUser不匹配，我们也不会继续向上走。与调度拦截器不同，处理程序拦截器在命令处理程序的上下文中调用。例如，这意味着他们可以根据正在处理的消息将相关性数据附加到工作单元。然后，此关联数据将附加到在该工作单元的上下文中创建的消息上。

处理程序拦截器通常还用于管理命令处理周围的事务。为此，请注册一个TransactionManagingInterceptor，然后配置一个TransactionManager，以启动和提交（或回滚）实际事务。

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字段值的命令。如果用户标识不在元数据中，则将引发一个异常，该异常将阻止处理该命令。另外，如果userId的值与axonUser不匹配，我们也不会继续执行程序。

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;

    }

}

我们可以使用CommandBus注册处理程序拦截器：

      public class CommandBusConfiguration {

    public CommandBus configureCommandBus() {

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

        commandBus.registerHandlerInterceptor(new MyCommandHandlerInterceptor());

        return commandBus;

    }

}

​

}

****@CommandHandlerInterceptor 注解****

该框架能够将处理程序拦截器作为@CommandHandlerInterceptor注解添加到聚合/实体上。聚合上的方法和“常规”命令处理程序拦截器的区别在于，使用注解，您可以根据给定聚合的当前状态做出决策。带注解的命令处理程序拦截器的一些属性是：

注释可以放在聚合中的实体上。

当命令处理程序位于子实体中时，可以在聚合根级别截获命令。

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。

可以将拦截器链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。

通过使用@CommandHandlerInterceptor注释的commandNamePattern属性，我们可以截获与所提供正则表达式匹配的所有命令。

事件可以从带注释的命令处理程序拦截器应用。

在下面的示例中，我们可以看到一个带@CommandHandlerInterceptor注释的方法，如果命令的状态字段与聚合的状态字段不匹配，则可以阻止命令执行：

    public class GiftCard {

    //..

    private String state;

    //..

    @CommandHandlerInterceptor

    public void intercept(RedeemCardCommand command, InterceptorChain interceptorChain) {

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

            interceptorChain.proceed();

        }

    }

}

请注意，它本质上是@CommandHandlerInterceptor@MessageHandlerInterceptor[此处](https://docs.axoniq.io/reference-guide/axon-framework/messaging-concepts/message-intercepting" \l "messagehandlerinterceptor)描述的更具体的实现。

事件拦截器

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，以对所有事件执行其他操作。这归结为两种类型的消息拦截器：调度拦截器和处理程序拦截器。

事件调度拦截器

发布事件时，将调用注册到事件总线的任何消息调度拦截器。他们可以通过添加元数据来更改事件消息。它们还可以为您提供事件发布的总体日志记录功能。这些拦截器总是在发布事件的线程上调用。

让我们创建一个事件消息调度拦截器，它记录正​​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，我们可以通过执行以下操作，向EventBus分配此调度拦截器：

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执行操作。例如，出于安全原因，拦截器甚至可以完全阻止事件处理。

拦截器必须实现MessageHandlerInterceptor接口。拦截器声明了当前接口的一个handle（）方法和两个参数：拦截链和当前UnitOfWork。拦截器链用于继续调度过程。UnitOfWork提供（1）正在处理的消息，（2）提供在（事件）消息处理之前、期间或之后连接逻辑的可能性（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，这意味着它们可以根据正在处理的消息将相关数据附加到工作单元。然后，该关联数据将附加到在该工作单元上下文中创建的事件消息（event message）。

让我们创建一个消息处理程序拦截器，它只允许处理包含axonuser作为元数据中userId字段值的事件。如果元数据中不存在userId，则将引发一个异常，该异常将阻止事件的处理。如果userId的值与axonUser不匹配，我们也不会继续向上走。对事件消息进行身份验证（如本例所示）是MessageHandlerInterceptor的常规用例。

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;

    }

}

我们可以使用EventProcessor注册处理程序拦截器，：

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****

****CommandBus和QueryBus都可以有处理程序拦截器和调度拦截器，而EventBus与以上两者不同，它只能注册调度拦截器。这是因为EventBus的唯一目的是事件发布/调度，因此它们是注册事件调度拦截器的地方。EventProcessors负责处理事件消息，因此它们是注册事件处理程序拦截器的地方。****

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查询执行操作。示例是日志记录或身份验证，无论查询类型如何，都要执行它。这是使用拦截器完成的。

查询调度拦截器

当在查询总线上调度查询或在查询更新发射器上调度对查询消息的订阅更新时，将调用消息调度拦截器。他们可以通过添加元数据来更改消息。他们还可以通过引发异常来阻止处理程序执行。这些拦截器总是在调度消息的线程上调用。

结构验证

如果查询不包含正确格式的所有必需信息，则没有必要处理查询。实际上，应该尽早阻止缺少必要信息的查询。因此，拦截器应检查所有传入查询中此类信息的可用性。这称为结构验证。

Axon Framework支持基于JSR 303 Bean验证的验证。这样，您便可以使用诸如@NotEmpty和@Pattern的注释来注释查询中的字段。您需要在类路径中包含JSR 303实现（例如Hibernate-Validator）。然后，在查询总线上配置 BeanValidationInterceptor ，它将自动查找并配置您的验证程序实现。当它使用合理的默认值时，您可以根据自己的特定需求对其进行微调。

****拦截器定制技巧****

您想要在无效查询上花费尽可能少的资源。因此，此拦截器通常位于拦截器链的最前面。在某些情况下，可能需要首先放置日志记录或审核拦截器，然后紧跟验证拦截器。

BeanValidationInterceptor同时实现了MessageHandlerInterceptor

，这使得你可以像配置处理程序拦截器一样配置它。

查询处理程序拦截器

消息处理程序拦截器（message handler interceptor）可以在查询处理之前和之后采取措施。出于安全原因，拦截器甚至可以完全阻止查询处理。

拦截器必须实现MessageHandlerInterceptor接口。此接口声明一个handle（）方法，该方法带有两个参数：当前UnitOfWork和一个拦截器链 。拦截器链InterceptorChain将用于继续调度过程。UnitOfWork可为您提供（1）正在处理的消息，以及（2）提供在（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，请参见[工作单元](https://docs.axoniq.io/reference-guide/axon-framework/messaging-concepts/unit-of-work)）。

与程序调度拦截器不同，在查询处理程序的上下文中调用处理程序拦截器。例如，这意味着他们可以根据正在处理的消息将相关性数据附加到工作单元。然后，此关联数据将附加到在该工作单元的上下文中创建的消息上。

@MessageHandlerInterceptor

除了在处理消息的组件上定义整体MessageHandlerInterceptor实例（例如命令，查询或事件）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。添加这种方法可以使您更精细地控制哪些消息处理组件应该做出反应以及它们如何做出反应。

添加@MessageHandlerInterceptor时，会为您提供几个句柄，例如：

* MessageHandlerInterceptor实例与拦截器链一起工作，以决定何时处理链中的其他拦截器。拦截器链是一个可选参数，可以将其添加到拦截方法中以提供相同的控制。如果没有此参数，一旦方法退出，框架将调用InterceptorChain#proceed。
* 您可以定义应处理Message的拦截器的类型。默认情况下，它对任何Message都做出反应。如果需要特定EventMessage的拦截器，则注释上的messageType参数应设置为EventMessage.class。
* 为了更细粒度地控制哪些消息应该对拦截器作出反应，可以指定要处理的消息中包含的payloadType。

以下代码片段显示了使用@MessageHandlerInterceptor注释的一些可能方法：

//简单的@MessageHandlerInterceptor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

    }

}

//定义消息类型的@MessageHandlerInterceptor方法

public class CardSummaryProjection {

    /\*

     \* Some @EventHandler and @QueryHandler annotated methods

     \*/

    @MessageHandlerInterceptor(

        messageType = EventMessage.class,

        payloadType = CardRedeemedEvent.class

    )

    public void intercept(CardRedeemedEvent event) {

        // Add your intercepting logic here based on the

    }

}

//定义InterceptorChain参数的@MessageHandlerInterceptor方法

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

    }

}

在message、payload和InterceptorChain旁边，@MessageHandlerInterceptor注释方法也可以解析其他参数。框架可以解析此类函数的哪些参数取决于拦截器处理的消息类型。有关哪些参数可用于处理的消息的详细信息，请查看此页。

@ExceptionHandler

@MessageHandlerInterceptor也允许更特定版本的拦截函数。即带@ExceptionHandler注释的方法。用@ExceptionHandler注释的函数将被视为一个处理程序拦截器，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，可以作为引发的数据库/服务异常的结果引发更特定于域的异常。您可以引入对所有异常作出反应的@ExceptionHandler，或者指定带注释的方法对哪些异常做出反应，如下所示：

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.

    }

}

**3.5一些注解支持的参数**

本章提供了注释消息处理函数的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一个叫做ParameterResolver的Axon框架内部机制来解析的。如果应该将自定义（或尚未）支持的参数注入到带注释的处理程序中，则可以扩展parameterResolver集。

****支持命令处理的参数****

默认情况下，@CommandHandler注释的方法允许以下参数类型：

※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。

※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。

※MetaData类型的参数将注入CommandMessage的整个元数据。

※UnitOfWork类型的参数获取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，或获得对向其注册的资源的访问权。

※类型为Message或CommandMessage的参数将获得完整的消息，其中包含有效负载和元数据。如果一个方法需要多个元数据字段或包装消息的其他属性，这将非常有用。

※用@MessageIdentifier注释的String类型的参数将解析正在处理的CommandMessage的标识符

※ConflictResolver类型的参数将解析配置的ConflictResolver实例。有关此主题的详细信息，请参阅冲突解决部分。

※InterceptorChain类型的参数将解析CommandMessage的messagehandler拦截器链，此特性应与@CommandHandlerInterceptor注释方法结合使用。关于这方面的更多细节，建议阅读本节。

※如果应用程序在Spring环境中运行，则可以解析任何springbean。请注意，@Qualifier注释可以与此一起使用，以进一步指定应该解析哪个Bean。

****支持查询处理的参数****

默认情况下，@QueryHandler注释方法允许以下参数类型：

※第一个参数是查询消息的有效负载。

※如果@QueryHandler注释明确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，查询名称是查询的有效负载的完全限定类名。

※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。

※MetaData类型的参数将注入QueryMessage的整个元数据。

※UnitOfWork类型的参数获取注入的当前工作单元。允许处理程序在此阶段注册或访问该单元的特定操作。

※类型为Message或QueryMessage的参数将获得完整的消息，包括有效负载和元数据。如果一个方法需要多个元数据字段或包装消息的其他属性，这将非常有用。

※用@MessageIdentifier注释的String类型的参数将解析正在处理的QueryMessage的标识符

※如果应用程序在Spring环境中运行，则可以解析任何springbean。请注意，@Qualifier注释可以与此一起使用，以进一步指定应该解析哪个Bean。

## **3.6处理异常**

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习惯的更具挑战性。尤其是在处理命令或查询时出现的故障，以及打算有返回值的消息时，我们应该意识到如何抛出异常。

### **3.6.1处理程序执行异常**

HandlerExecutionException标记源自消息处理成员的异常。由于事件消息是单向的，因此处理事件时不包含任何返回值。因此，HandlerExecutionException只应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这个异常的更具体的实现，分别是CommandExecutionException和QueryExecutionException。

在分布式应用程序环境中，专用处理程序执行异常的有用性变得更加明显，例如，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，您失去了两个应用程序可以访问相同类的任何确定性，因此对于任何异常类都是如此。为了支持和鼓励这种分离，Axon将生成任何由命令或查询处理导致的异常。

为了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，建议在命令/查询处理失败时抛出CommandExecutionException/QueryExecutionException，其中包含所需的详细信息。这种行为通常可以通过实现拦截器来支持，这些拦截器为您执行异常包装。

### **3.6.2 @ExceptionHandler注释方法**

Axon框架允许使用带@ExceptionHandler注释的方法来提供关于如何对异常做出反应的更细粒度的控制。更具体地说，这是一种专门用于对异常结果作出反应的消息处理程序侦听器。注意，这样的@ExceptionHandler只处理同一类中消息处理函数抛出的异常。有关如何使用此注释的详细信息，请参阅本节。

## **3.7工作单元**

工作单元（UnitOfWork）是Axon框架中的一个重要概念。不过，在大多数情况下，你不太可能直接与它互动。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，例如onPrepareCommit或onCleanup。

你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，无论出于什么原因，你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。如果使用注释支持，则可以将UnitOfWork类型的参数添加到带注释的方法中。在其他位置，您可以通过调用CurrentUnitOfWork.get(). 请注意，如果没有UnitOfWork绑定到当前线程，则此方法将引发异常。使用CurrentUnitOfWork.isStarted（）看看是否有工作单元启动。

访问当前工作单元的一个原因是附加在消息处理过程中多次重用的资源，或者如果创建的资源需要在工作单元完成时清理。在这种情况下，unitOfWork.GetorComputerSource（）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此工作单元期间要执行的操作。

请注意，工作单元只是变更的缓冲区，而不是事务的替代品。尽管所有阶段化的更改都只在提交工作单元时提交，但它的提交不是原子性的。这意味着，当提交失败时，某些更改可能已被持久化，而其他更改则没有持久化。最好使一个命令只包含单个操作。如果你坚持这种做法，每个工作单元将只包含一个单独的动作，这样它就可以安全地按原样使用。如果工作单元中有更多操作，则可以考虑将事务附加到工作单元的提交。使用unitOfWork.onCommit（..）注册在提交工作单元时需要采取的行动。

您的处理程序可能会由于处理消息而引发异常。默认情况下，未经检查的异常将导致UnitOfWork还原所有更改。这可能不会引发你所设置的边缘动作。

Axon Framework提供了一些现成的回滚策略：

RollbackConfigurationType.NEVER -将始终提交 UnitOfWork

RollbackConfigurationType.ANY\_THROWABLE-发生异常时总是回滚

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（但不会包含错误）

使用框架组件处理消息时，工作单元的生命周期将自动为您管理。如果选择不使用这些组件，而是自己处理，则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。它期望在单个线程内进行处理。要在工作单元的上下文中执行任务，只需在新的DefaultUnitOfWork上调用UnitOfWork.execute（可运行）或UnitOfWork.executeWithResult（可调用）。工作单元将在任务完成时启动并提交，如果任务失败则回滚。如果需要更多控制，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...

}

工作单位包含着信息。它总是以要处理的消息开头。工作单元执行executeWithResult（…）后将返回ResultMessage，而实际执行结果则是该ResultMessage的有效负载。如果在消息处理过程中出现问题，我们将得到一个异常的ResultMessage--isException（）方法将返回true，exceptionResult（）将为我们提供实际的可丢弃文件，指出出了什么问题。

一个工作单元有几个生命周期。每次进入一个新阶段时，都会通知监听器。

* 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（单位工程）。随后，消息通常在这个阶段由消息处理程序处理。
* 提交阶段-消息处理完成后，在提交工作单元之前，将调用onPrepareCommit侦听器。如果一个工作单元绑定到一个事务，则调用onCommit侦听器来提交任何支持事务。当提交成功时，将调用afterCommit侦听器。如果提交或之前的任何步骤失败，则调用onRollback侦听器。消息处理程序结果包含在工作单元的ExecutionResult中（如果可用）。
* 清理阶段-释放此工作单元（如锁）所持有的任何资源的阶段。如果嵌套了多个工作单元，则清理阶段将推迟到外部工作单元准备清理。

  可以将消息处理过程视为一个原子过程；它要么完全处理，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将尝试提交在工作单元中注册的操作。

可以将事务绑定到工作单元。许多组件（例如CommandBus和QueryBus的实现以及所有异步处理的EventProcessors）都允许您配置TransactionManager。然后，系统将使用此事务管理器创建事务并将其绑定到用于管理消息流程的工作单元。

当应用程序组件在消息处理的不同阶段（如数据库连接或EntityManager）需要资源时，可以将这些资源附加到UnitOfWork。这个unitOfWork.getResources（）方法允许您访问附加到当前工作单元的资源。有几个helper方法可以直接在工作单元上使用，这使得使用资源更容易。

当嵌套的工作单元需要访问资源时，建议将其注册到根工作单元上，可以使用unitOfWork.root(). 如果一个工作单元是根，这个方法就会返回自己。