# **六、查询端**

## **6.1查询处理**

当遵循CQRS时，将查询作为一种专用的消息类型处理与分离查询模型是一致的。虽然创建查询处理层相当简单，但是在应用程序的这一部分使用Axon框架有很多好处。

通过提供描述查询处理方法的功能（在本节中进一步解释）和查询消息的专用总线，可以使用诸如拦截器和消息监视之类的常见消息特性。

下一节将概述与配置必要组件以开始在Axon应用程序中处理查询相关的任务。为此，讨论了注册@QueryHandler注释方法的方法，以及在调度查询时提供了哪些选项。

*查询类型*

*Axon框架区分了三种类型的查询，即：*

*（1）点对点查询，（2）分散收集查询，和（3）订阅查询。*

## **6.2 查询适配器**

如何处理查询消息在查询处理部分有更详细的介绍。查询必须被调度，就像任何类型的消息一样，才能被处理。为此，Axon提供了两个接口：查询总线以及查询网关

本页将显示如何以及何时使用查询网关和总线。这里讨论了如何配置查询网关和总线实现以及具体细节。

### **6.2.1查询总线和查询网关**

QueryBus是将查询分派到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。可以为同一请求-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。在分派查询时，客户机必须指出它是希望从单个处理程序还是从所有处理程序获得响应。

QueryGateway是查询调度机制的一个方便接口。虽然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负载的必要性。

无论您选择使用QueryBus还是QueryGateway，两者都提供了几种类型的查询。Axon框架区分了三种类型，即：点对点查询，分散-聚集查询，以及订阅查询。

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来决定实际调用哪个处理程序。在下面的列表中，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(在我们的案例中是java.lang.String）。

但是，可以通过声明@QueryHandler注释的queryName属性来重写此行为。

如果我们想查询视图模型List<String>，我们可以这样做：

// 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，也可以声明查询名称，默认情况下，这是查询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，可以立即进行解析。但是，如果@QueryHandler注释函数的返回类型是CompletableFuture，不管查询总线的类型如何，结果都将异步返回。

②分散-聚集查询

当您需要来自与查询消息匹配的所有查询处理程序的响应时，分散聚集查询就是要使用的类型。作为对该查询的响应，将返回结果流。此流包含每个成功处理查询的处理程序的结果，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。

在下面的列表中，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，我们希望从这两个组件中获取结果。因此，我们将使用分散聚集查询，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);

③订阅查询

订阅查询允许客户机获得它要查询的模型的初始状态，并在所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，当初始状态发生变化时，可以对其进行更新。为了用对模型的更改来更新订阅，我们将使用Axon提供的QueryUpdateEmitter组件。

让我们看看CardSummary投影中的一个片段：

@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为我们提供GiftCard状态的列表。一旦我们的礼品卡被兑换，我们想更新任何对礼品卡的更新状态感兴趣的组件。我们将通过在redemedevt事件的事件处理程序函数中使用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，我们通过更新现有卡来更新视图模型。
2. 如果有订阅查询对有关此特定礼品卡的更新感兴趣，我们将发出更新。emission的第一个参数是查询的类型（在本例中为fetchcardsumeriesquery）先前定义的查询处理程序中的查询类型相对应。第二个参数是一个谓词，它将选择要更新的订阅查询。在本例中，我们将只更新对已更新的GiftCard感兴趣的订阅查询。第三个参数是实际更新，在我们的例子中是卡片摘要。有几个emit方法的重载存在，可以查看JavaDoc了解更多细节。这里需要强调的重要一点是，更新是一条消息，并且一些重载会把更新消息作为一个参数（在我们的例子中，我们只发送了包装在消息中的有效负载）例如这使我们能够附加元数据。

一旦我们实现了查询处理和发出端，我们就可以发出订阅查询以获取礼品卡的初始状态，并在兑换此礼品卡后进行更新：

// 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));

发行带有gc1 id和金额初始值的礼品卡。

创建订阅查询消息以获取礼品卡列表（此初始状态是CardSummary的多个实例）一旦id为gc1的礼品卡的状态发生更改，则需要更新（在我们的例子中，更新意味着该卡被兑换）。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侧的类型匹配。

一旦消息被创建，我们就通过QueryGateway发送它。另一个更新结果是一个结果。为了达到“反应性”，我们使用项目反应器的Mono作为初始结果和流量更新。

注意： 一旦发出订阅查询，所有更新都将排队，直到对更新流量的订阅完成。此行为可防止更新丢失。

注意： 该框架防止发出具有相同id的多个查询消息。如果需要在多个不同的位置更新，请创建一个新的查询消息。

注意：reactor core依赖项对于订阅查询的使用是必需的。但是，它是一个编译时依赖项，其他Axon特性不需要它。

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

如果我们希望对结果进行更精细的控制，可以对查询结果使用initialResult（）和updates（）方法。

* 因为queryUpdateEmitter即使在没有订户的情况下也会继续发出更新，所以我们需要在不再对接收更新感兴趣时通知发出方。否则，可能会导致无限流挂起，并最终导致内存泄漏。完成订阅查询后，我们需要关闭已使用的资源。我们可以用doFinally hook。作为doFinally钩子的一种替代方法，还有一种使用API的Flux。这与使用资源Java API是一样的：

Flux.using( () -> fetchQueryResult,queryResult ->

queryResult.handle(..., ...),SubscriptionQueryResult::close);

* 当我们发出一个RedembleCmd时，我们在投影中的事件处理程序最终将被触发，这将导致发出更新。因为我们使用println（）方法订阅了更新，所以一旦收到更新，就会将其打印出来。

## **6.3 查询处理器**

  查询的处理归结为返回查询响应的带注释的处理程序。本章的目标是描述这样一个@QueryHandler注释方法的外观，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，建议阅读配置部分。

### **6.3.1编写查询处理程序**

在Axon中，对象可以声明许多查询处理程序方法，方法是用@QueryHandler注释对它们进行注释。所讨论的对象是您将称为查询处理程序或查询处理组件的对象。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“Gift Card”域为例，我们可以假设有一条查询消息来获取单个CardSummary实例。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

}

如上所示，我们有一个常规的POJO，它将根据cardSummaryId字段获取cardSummaryId。此FetchCardSummaryQuery将被调度到一个处理程序，该处理程序将给定消息定义为其第一个声明的参数。处理程序可能包含在一个对象中，该对象负责或访问所讨论的CardSummary模型：

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

}

在上面的示例中，我们希望在编写查询处理程序时强调两个细节：

@QueryHandler注释，它将函数标记为查询处理程序方法。

有问题的方法由返回类型CardSummary（称为查询响应类型）和FetchCardSummaryQuery（查询负载）定义。

*存储查询模型*

*在本例中，我们选择使用常规映射作为存储方法。在实际的系统中，这将被数据库或存储库层的形式所取代。*

### **6.3.2查询处理程序调用顺序**

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具体的方法：

* 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
* 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
* 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
* 当到达层次结构的顶层，并且没有找到合适的查询处理程序时，将忽略此查询处理实例。

注意，与命令处理类似，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

// QueryC扩展了QueryB

// 注册一个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，将为查询QueryB调用SubQueryHandler的处理程序方法，结果MyResult QueryHandler的处理程序方法被调用用于查询QueryA和QueryC和result MyResult。

### **6.3.3查询处理程序返回值**

Axon允许查询处理程序方法的返回类型很多，如本页前面定义的那样。您应该考虑单个对象和对象集合，同时考虑通配符或泛型。下面我们共享一个列表，列出了框架中支持和测试的所有选项。

为了清楚起见，我们在响应类型的单个实例和多个实例之间进行偏差。这是在发送查询时指定ResponseType的要求，该查询要求用户在需要单个结果或多个结果时声明。Axon将使用这个ResponseType对象，将查询与查询处理程序方法匹配，并与查询有效负载和查询名称一起匹配。

6.3.3.1支持的单实例返回值

要查询单个对象，应该使用ReponseTypes\instanceOf（Class）方法创建所需的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：

1. 完全匹配的类
2. 类的子类型
3. 绑定到类的泛型
4. 类的future
5. 类的原语
6. 一个可选的类

*基本返回类型*

*在常用对象中，查询也可以返回基本数据类型：*

public class QueryHandler {

@QueryHandler

public float handle(QueryA query) { }

}

请注意，查询方将检索装箱结果而不是基元类型。

6.3.3.2支持多个实例返回值

要查询多个对象，应使用ReponseTypes\multipleInstancesOf（Class）方法创建所需的ResponseType对象。此“类的多个实例”ResponseType对象反过来支持以下查询处理程序返回值：

包含以下内容的数组：类/类的子类型/类的泛型绑定

Iterable或Iterable的自定义实现，包含：

1. 类
2. 子类
3. 类的泛型绑定
4. 绑定到类的通配符
5. 类的流
6. 类的future

6.3.3.3不支持的返回值

以下是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

## **6.4实现**

在调度查询时，如分派查询一节所述，在实际发送查询消息时有两种实现。下面将概述可能的实现，并指出如何使用Axon设置查询调度基础设施。

### **6.4.1查询网关**

查询网关是查询调度机制的一个方便接口。虽然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QueryGateway接口和DefaultQueryGateway实现。查询网关提供了许多方法，允许您发送一个查询并等待单个或多个结果，无论是同步的、超时的还是异步的。查询网关需要配置为对查询总线的访问权和QueryDispatchInterceptor列表（可能为空）。

### **6.4.2查询总线**

查询总线是将查询分派到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。可以为同一请求-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。在分派查询时，客户机必须指出它是希望从单个处理程序还是从所有处理程序获得响应。

6.4.2.1 AxonServer查询总线

Axon提供了一个现成的查询总线，即AxonServerQueryBus。它连接到axoniqaxonserver服务器来发送和接收查询。

AxonServerQueryBus是一种“分布式查询总线”。默认情况下，它使用SimpleQueryBus处理不同JVM上的传入查询。

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>

配置应用：

// Returns a Configurer instance with default components configured.

// `AxonServerQueryBus` is configured as Query Bus by default.

Configurer configurer = DefaultConfigurer.defaultConfiguration();

**Springboot部分：**

还是只需要引入axon-spring-boot-starter依赖，springboot就能自动进行配置。

*排除Axon服务器连接器*

*如果排除了axon服务器连接器依赖项，则将回退到“非axon服务器”查询总线选项SimpleQueryBus（请参见下文）*

6.4.2.2简单查询总线

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

Axon部分：

Configurer configurer = DefaultConfigurer.defaultConfiguration()

.configureQueryBus(c -> SimpleQueryBus.builder()

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

.messageMonitor(c.messageMonitor(SimpleQueryBus.class, "queryBus"))

.build()

);

**Springboot部分：**

@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();

}

## **6.5 配置**

本页描述配置查询处理程序的过程。注意，查询处理程序是包含@QueryHandler注释函数的（单例）对象。

### **6.5.1注册查询处理程序**

注册查询处理程序时，这意味着您注册的类包含带注释的查询处理程序。在配置过程中接收到这样的类后，Axon将扫描其内容以查找所有@QueryHandler注释的方法。在注册过程中，以下信息定义了给定的查询处理函数：

* 第一个参数是方法的有效负载。
* 方法响应类型是查询的响应类型。
* 注释中queryName字段的值作为查询的名称（这是可选的，如果没有它，将默认为查询负载）。

请注意，可以为同一查询负载、响应类型和名称注册多个查询处理程序。此外，在调度查询时，客户机可以指示他/她是希望从单个处理程序获得结果，还是希望从与查询负载、名称和响应类型组合对应的所有处理程序获得结果。

以下代码片段指出如何注册查询处理程序：

Axon部分

假定存在以下处理程序：

public class CardSummaryProjection {

@QueryHandler

public CardSummary handle(FetchCardSummaryQuery query) {

CardSummary cardSummary;

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

return cardSummary;

}

}

以下代码是将CardSummaryProjection注册为Query Handler所需的：

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

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

或者，可以使用更通用的方法在组件中注册所有类型的消息处理程序：

Configurer axonConfigurer = DefaultConfigurer.defaultConfiguration()

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

*在单个查询处理程序中使用相同的查询处理方法*

*查询处理程序类当前可以包含多个相同的查询处理方法。但是，实际调用哪个方法的结果尚未指定。*

*请注意，这应该被视为一种非常罕见的场景，因为通常相同的查询处理方法将分布在多个查询处理程序上。*