Axon入门系列

<http://edisonxu.com/2017/03/23/hello-cqrs.html>

<https://github.com/EdisonXu/sbs-axon>

1. ：CQRS基本概念

在研究微服务的过程中，跨服务的操作处理，尤其是带有事务性需要统一commit或rollback的，是比较麻烦的。本系列记录了我在研究这一过程中的心得体会。

本篇主要就以下几个问题进行介绍：

微服务中的一个大难题

DDD中的几个基本概念

什么是EventSourcing？

什么是CQRS？

EventSourcing和CQRS的关系？

CQRS/ES怎么解决微服务的难题？

微服务中的一个大难题

微服务架构已经热了有两年了，而且目测会越来越热，除非有更高级的架构出现。相关解释和说明，网上一搜一大堆，我这里就不重复了。一句话概括：

微服务将原来的N个模块，或者说服务，按照适当的边界，从单节点划分成一整个分布式系统中的若干节点上。

原来服务间的交互直接代码级调用，现在则需要通过以下几种方式调用：

SOA请求

RPC调用

ED(EventDriven)事件驱动

前面两种就比较类似，都属于直接调用，好处明显，缺点是请求者必须知道被请求方的地址。现在一般会提供额外的机制，如服务注册、发现等，来提供动态地址，实现负载和动态路由。目前大多数微服务框架都走的这条路子，如当下十分火热的SpringCloud等。

事件驱动的方式，把请求者与被请求者的绑定关系解耦了，但是需要额外提供一个消息队列，请求者直接把消息发送到队列，被请求者监听队列，在获取到与自己有关系的事件时进行处理。主要缺点主要有二：

1） 调用链不再直观；

2） 高度依赖队列本身的性能和可靠性；

但无论是哪种方式，都使得传统架构下的事务无法再起到原先的作用了。

事务的作用主要有二：

统一结果，要么都成功，要么都失败

并发时保证原子性操作

在传统架构下，无论是DB还是框架所提供的事务操作，都是基于同线/进程的。在微服务所处的分布式框架下，业务操作变成跨进程、跨节点，只能自行实现，而由于节点通信状态的不确定性、节点间生命周期的不统一等，把实现分布式事务的难度提高了很多。

这就是微服务中的一个大难题。

统一结果，要么都成功，要么都失败

并发时保证原子性操作

在传统架构下，无论是DB还是框架所提供的事务操作，都是基于同线/进程的。在微服务所处的分布式框架下，业务操作变成跨进程、跨节点，只能自行实现，而由于节点通信状态的不确定性、节点间生命周期的不统一等，把实现分布式事务的难度提高了很多。

这就是微服务中的一个大难题。

DDD中的几个基本概念

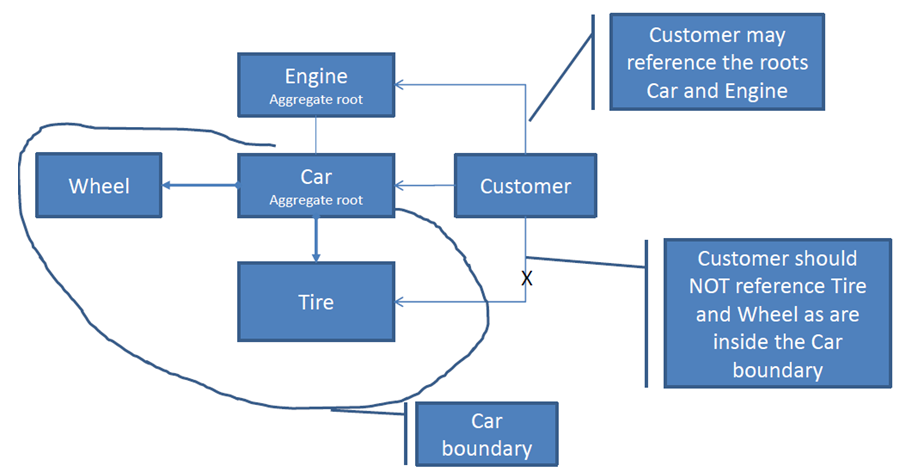
在进一步深入前，必须要了解几个基本概念。这些基本概念在EventSourcing和CQRS中都会用到。

**Aggregate**

聚合。这个词或许听起来有点陌生，用集合或者组合就好理解点。

A DDD aggregate is a cluster of domain objects that can be treated as a single unit.

—— Martin Fowler



车、轮子、轮胎构成了一个聚合。其中车是聚合根(AggregateRoot)

Aggregate有两大特征： 明确的边界、AggregateRoot

具体来说，Aggregate存在于两种形式：

一个单独的对象；

一组相互有关联的对象，其中一个作为ROOT，外部只能通过AggregateRoot对这组对象进行交互；

这里Customer不能直接访问Car下面的Tire，只能通过聚合根Car来访问。

什么是EventSourcing?

不保存对象的最新状态，而是保存对象产生的所有事件。通过事件回溯(Event Sourcing, ES)得到对象最新的状态

以前我们是在每次对象参与完一个业务动作后把对象的最新状态持久化保存到数据库中，也就是说我们的数据库中的数据是反映了对象的当前最新的状态。而事件溯源则相反，不是保存对象的最新状态，而是保存这个对象所经历的每个事件，所有的由对象产生的事件会按照时间先后顺序有序的存放在数据库中。当我们需要这个对象的最新状态时，只要先创建一个空的对象，然后把和改对象相关的所有事件按照发生的先后顺序从先到后全部应用一遍即可。这个过程就是事件回溯。

因为一个事件就是表示一个事实，事实是不能被磨灭或修改的，所以ES中的事件本身是不可修改的(Immutable)，不会有DELETE或UPDATE操作。

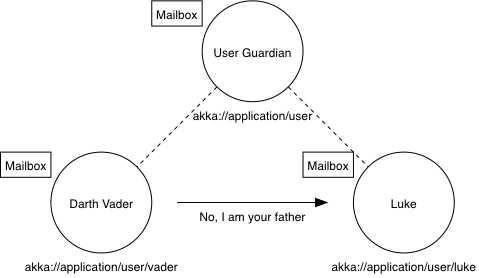
ES很明显先天就会有个问题——由于不停的记录Event，回溯获得对象最新状态所需花的时间会与事件的数量成正比，当数据量大了以后，获取最新状态的时间也相对的比较长。

而在很多的逻辑操作中，进行“写”前一般会需要“读”来做校验，所以ES架构的系统中一般会在内存中维护一份对象的最新状态，在启动时进行”预热”，读取所有持久化的事件进行回溯。这样在读对象——也就是Aggregate的最新状态时，就不会因为慢影响性能。

同时，也可以根据一些策略，把一部分的Event合集所产生的状态作为一个snapshot，下次直接从该snapshot开始回溯。既然需要读，就不可避免的遇到并发问题。EventSourcing要求对回溯的操作必须是原子性的，具体实现可参照Actor模型。

Actor Model

ActorModel的核心思想是与对象的交互不会直接调用，而是通过发消息。如下图：



每一个Actor都有一个Mailbox，它收到的所有的消息都会先放入Mailbox中，然后Actor内部单线程处理Mailbox中的消息。从而保证对同一个Actor的任何消息的处理，都是线性的，无并发冲突。整个系统中，有很多的Actor，每个Actor都在处理自己Mailbox中的消息，Actor之间通过发消息来通信。

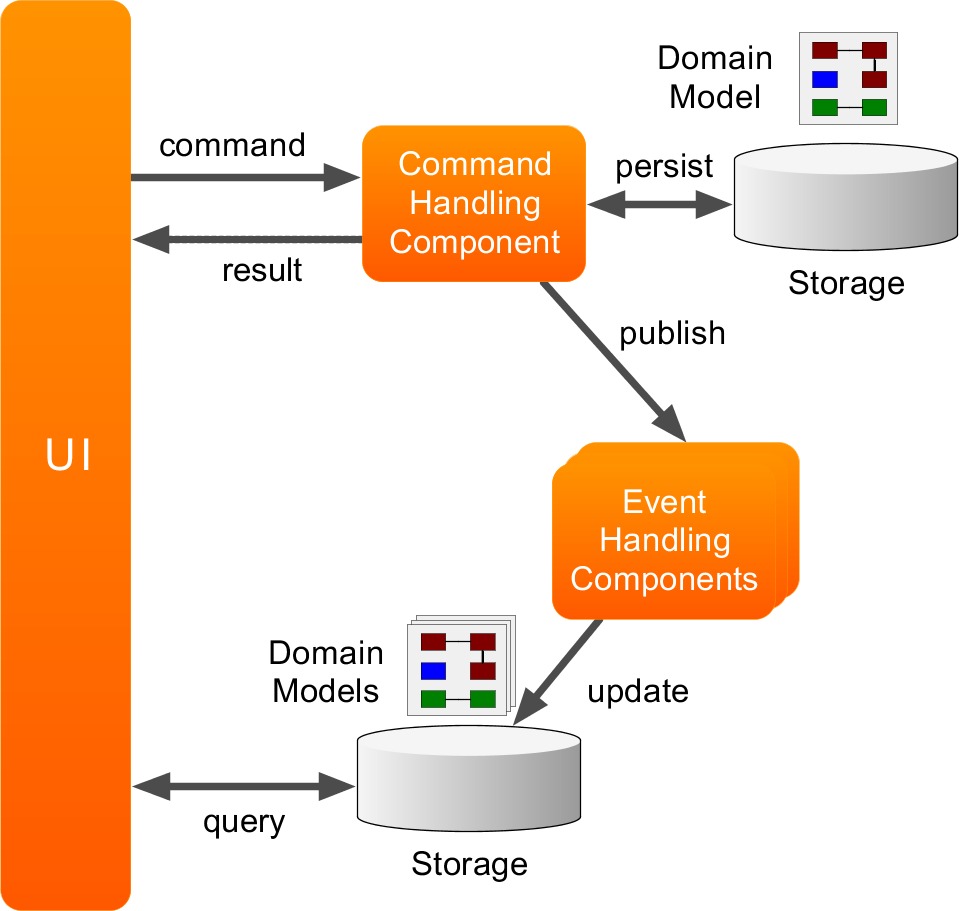
Akka框架就是实现Actor模型的并行开发框架。Actor作为DDD聚合根，最新状态是在内存中。Actor的状态修改是由事件驱动的，事件被持久化起来，然后通过Event Sourcing的技术，还原特定Actor的最新状态到内存。

另外，还有Eventuate，两者的作者是同一人，如果对Akka和Eventuate的区别感兴趣的话，可以参照我翻译的一篇文章(译)Akka Persistence和Eventuate的对比。

什么是CQRS?

CQRS 架构全称是Command Query Responsibility Segregation，即命令查询职责分离，名词本身最早应该是Greg Young提出来的，但是概念却很早就有了。

本质上，CQRS也是一种读写分离的机制，架构图如下：



CQRS把整个系统划分成两块：

**Command Side 写的一边**

接收外部所有的Insert、Update、Delete命令，转化为Command，每一个Command修改一个Aggregate的状态。Command Side的命令通常不需要返回数据。注意：这种“写”操作过程中，可能会涉及“读”，因为要做校验，这时可直接在这一边进行读操作，而不需要再到Query Side去。

**Query Side 读的一边**

接受所有查询请求，直接返回数据。

由于C端与Q端的分离，两端各有一个自己的Repository，可根据不同的特性选取不同的产品，比如C端用RMDB，而Q端选用读取速度更快的NoSQL产品。

CQRS适用的场景

使用了CQRS架构，由于读写之间会有延迟，就意味着系统的一致性模型为最终一致性(Eventual Consistency)，所以CQRS架构一般用于读比写大很多的场景。

*注意：*

CQRS并不像SOA、EDA(EventDrivenArchitecture)属于顶级架构，它有自己的局限性，并不适合于一切场景。有些天然适合于CRUD的系统，在评估CQRS所带来的好处与坏处后，认为利大于弊再选取CQRS。所以，通常CQRS只作为一个大系统中某部分功能实现时使用。

EventSourcing和CQRS的关系

从前面的介绍，应该可以发现两者其实并没有直接的关系，但是EventSourcing天然适合CQRS架构的C端的实现。

CQRS/ES整合在一起的架构，优缺点如下：

**优点**

记录了数据变化的完整过程，便于BI分析

可以有效解决线上的数据问题，重演一遍，就可以找到问题所在

可以随时将系统还原到任何一个时间点

正确的实施后，天然适合并发场景

**缺点**

事件数量巨大，造成存储端成本上升

通过回溯重演获取Aggregate状态时，如果相关事件过多，需要提前“预热”

事件本身的内容被重构后，必须兼容以前的事件

事件驱动对传统思维的改变，需要适应

实施门槛高，需要成熟框架或中间件支撑

CQRS/ES怎么解决微服务的难题？

我们先把实现微服务事务中的主要难点列出来，然后看用CQRS/ES是怎么一一解决的。

**必须自己实现事务的统一commit和rollback；**

这个是无论哪一种方式，都必须面对的问题。完全逃不掉。在DDD中有一个叫Saga的概念，专门用于统理这种复杂交互业务的，CQRS/ES架构下，由于本身就是最终一致性，所以都实现了Saga，可以使用该机制来做微服务下的transaction治理。

**请求幂等**

请求发送后，由于各种原因，未能收到正确响应，而被请求端已经正确执行了操作。如果这时重发请求，则会造成重复操作。

CQRS/ES架构下通过AggregateRootId、Version、CommandId三种标识来识别相同command，目前的开源框架都实现了幂等支持。

**并发**

单点上，CQRS/ES中按事件的先来后到严格执行，内存中Aggregate的状态由单一线程原子操作进行改变。

多节点上，通过EventStore的broker机制，毫秒级将事件复制到其他节点，保证同步性，同时支持版本回退。(Eventuate)

CQRS/ES如何与微服务架构结合

结合的方式很简单，就是把合适的服务变成CQRS/ES架构，然后提供一个统一的分布式消息队列。

每个服务自己内部用的C或Q的Storage完全可以不同，但C端的Storage尽量使用同一个，例如MongoDB、Cansandra这种本身就是HA的，以保证可用性。同时也可以避免大数据分析导数据时需要从不同的库导。

目前，相对成熟的CQRS/ES可用框架有：

AxonFramework

Akka Persistence

Eventuate

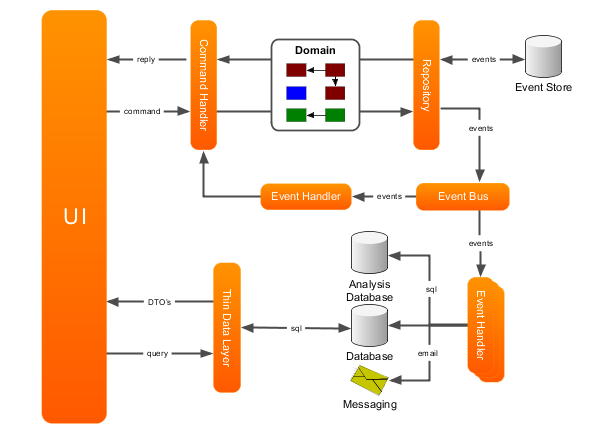
ENode

Confluent

1. ：Hello,Axon3

简介

AxonFramework是一个基于事件驱动的轻量级CQRS框架，既支持直接持久化Aggreaget状态，也支持采用EventSourcing，使用AxonFramework的应用架构如下



Aggregate

显然，在这个例子中，我们要实现一个Aggregate是银行账户，定义如下

public class BankAccount {

@AggregateIdentifier

private AccountId accountId;

private String accountName;

private BigDecimal balance;

}

结合前文DDD概念中关于Aggregate的介绍，每个Aggregate都有自己独立的全局唯一的标识符，@AggregateIdentifier即是这个唯一标识的标志，例子中就是银行的AccountId。一个AggregateIdentifier必须：

实现equal和hashCode方法，因为它会被拿来与其他标识对比

实现toString方法，其结果也应该是全局唯一的

实现Serializable接口以表明可序列化

Command

在CQRS模式下，所有的“写”操作，都是发送Command来操作。Axon中Command可以是任意的POJO类，由于axon是基于事件驱动的架构，Command类处理时会被axon封装成一个CommandMessage。

篇幅问题，我这里省略了getter/setter方法，但是，如果使用Jackson做序列化器，必须实现空参构造器和提供所有field的getter方法！

Event

Event是系统中发生任何改变时产生的事件类，典型的event就是对Aggregate状态的修改。与Command一样，Event可以是任何POJO，axon也会把Event自动封装成EventMessage，其中如果是Aggregate发送出来的Event，会被封装成DomainEventMessage。通常来说，Event最好是可序列化的。那么对应到本例，显然有两个Event：

AccountCreatedEvent

一样，省略了gettter/setter，注意序列化器对构造器和getter的要求。

CommandHandler

axon使用@CommandHandler注解来标明用来处理Command的方法，配置时会把这些CommandHandler统一加载管理，与其对应的Command形成KV键值对。在Aggregate实现BankAccount里面加入CommandHandler如下：

@CommandHandler

public BankAccount(CreateAccountCommand command){

apply(new AccountCreatedEvent(command.getAccountId(), command.getAccountName(), command.getAmount()));

}

@CommandHandler

public void handle(WithdrawMoneyCommand command){

apply(new MoneyWithdrawnEvent(command.getAccountId(), command.getAmount()));

}

这里不做其他事，只简单的产生Event并使用提供的静态方法apply把Event发送出去。

值得一提的是，这里用一个构造器来接受CreateAccountCommand，至于有什么特殊，这里卖个关子，文章最后见分晓。

EventHandler

专门用来处理Event的方法，用@EventHandler标明或使用EventHandlingConfiguration去注册。在BankAccount内加入：

@EventHandler

public void on(AccountCreatedEvent event){

this.accountId = event.getAccountId();

this.accountName = event.getAccountName();

this.balance = new BigDecimal(event.getAmount());

LOGGER.info("Account {} is created with balance {}", accountId, this.balance);

}

@EventHandler

public void on(MoneyWithdrawnEvent event){

BigDecimal result = this.balance.subtract(new BigDecimal(event.getAmount()));

if(result.compareTo(BigDecimal.ZERO)<0)

LOGGER.error("Cannot withdraw more money than the balance!");

else {

this.balance = result;

LOGGER.info("Withdraw {} from account {}, balance result: {}", event.getAmount(), accountId, balance);

}

}

配置

现在基本内容都有了，只差最后一步，对axon进行配置。Axon启动最少要指定如下几个模块：

**CommandBus**

CommandBus是用来分发Command到对应CommandHandler的机制。每一个Command只会发送到一个CommandHandler去，当有多个CommandHandler去订阅一个CommandMessage时，最后一个覆盖前面所有。

Axon内置了四种CommandBus：

**1 SimpleCommandBus**

默认，直接在发送线程里去执行command handler，执行后保存Aggregate状态和发送事件也都在同一个线程上，适用于大多数情况。

**2 AsynchrounousCommandBus**

默认使用一个CachedThreadPool来起一个新线程去处理command。CachedThreadPool线程调用时，会检查是否有可用的线程，没有则创建。闲置线程60s后自动关闭。也可以通过config指定其他的线程池来采用不同的线程调度策略。

**3 DisruptorCommandBus**

适用于多线程场景。SimpleCommandBus在遇到多线程调用时，为了保证aggregate的状态，必须要加锁，这样就降低了效率。DisruptorCommandBus用了开源的并发处理框架Disruptor，用两组线程来处理多线程场景，一组用于执行command handler去更新aggregate的状态，一组用于存储和发送所产生的event到EventStore。

但是DisruptorCommandBus有以下的限制：

仅支持Event Sourced Aggregates

一个Command只能改变一个Aggregate的状态。

当使用Cache的时候，一个identifier只能对应一个aggregate，即不允许两个不同类型的aggregate拥有同一个identifier

所处理的Command不能导致UnitOfWork的rollback，因为DisruptorCommandBus无法保证rollback时按照dispatch的顺序来处理。

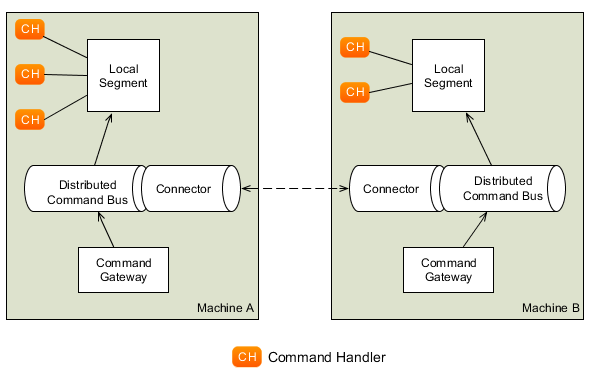
用于更新Aggregate的command只能按照dispatch的顺序执行，无法指定顺序。

**4 DisruptorCommandBus**

可以使用DisruptorConfiguration来配置，它提供了一些进一步优化的参数。

DistributedCommandBus 不像其他CommandBus，DistributedCommandBus并不调用任何command handler，它只是在不同JVM的commandbus之间建立一个“桥梁”。每个JVM上的DistributedCommandBus被称为“Segment”。

DistributedCommandBus需要指定路由规则和具体的connector，这两个东东具体实现由distributed-commandbus-xxx模块提供。



EventBus

EventBus用于把event发送到subscribe它的各个handler去。Axon提供了两种EventBus的实现，都支持订阅和跟踪：

1 SimpleEventBus 默认的EventBus，不持久化event，一旦发送到消费者去，就会销毁。

2 EmbeddedEventStore 可以持久化event，以便以后replay。

Repository

即Aggregate的持久化方式。Axon内置了两种

1 Standard Repositories 代表是GenericJpaRepository，直接把Aggregate的最新状态存到db去。

2 Event Sourcing Repositories 并不直接保存Aggregate的最新状态，而是保存对Aggregate造成影响的所有Event，通过Event回溯来恢复Aggregate状态

EventStorageEngine

提供event在底层storage读写的机制，内置了若干种：

1 InMemoryEventStorageEngine 存储到内存中

2 JpaEventStorageEngine 使用JPA进行存储

3 JdbcEventStorageEngine 使用jdbc

4 MongoEventStorageEngine 使用Mongodb存储event

Serializer

由于是事件驱动框架，序列化器必不可少。Axon内置了三种：XStreamSerializer, JavaSerializer, JacksonSerializer，默认是XStreamSerializer，使用XStream来做序列化，理论上比Java自带的序列化器要快。

public class Application {

private static final Logger LOGGER = getLogger(Application.class);

public static void main(String args[]){

Configuration config = DefaultConfigurer.defaultConfiguration()

.configureAggregate(BankAccount.class)

.configureEmbeddedEventStore(c -> new InMemoryEventStorageEngine())

.buildConfiguration();

config.start();

AccountId id = new AccountId();

config.commandGateway().send(new CreateAccountCommand(id, "MyAccount",1000));

config.commandGateway().send(new WithdrawMoneyCommand(id, 500));

config.commandGateway().send(new WithdrawMoneyCommand(id, 500));

}

}

Axon提供了DefaultConfigurer来帮助我们做一些基本配置，所以我们只需要简单的做Aggregate的注册和指定一个EventStorageEngine。这里因为是测试，用了InMemoryEventStorageEngine。CommandGateway是对CommandBus的一个封装，更加方便的来发送Command。

本文完整代码

https://github.com/EdisonXu/sbs-axon/tree/master/lesson-1

前面说用一个构造器来接受CreateAccountCommand，有什么特殊地方。这里涉及到一个问题，就是Aggregate在Repository的创建。

Axon中，打开@Aggregate注解的定义会发现里面其实定义了一个repository。

/\*\*

\* Selects the name of the AggregateRepository bean. If left empty a new repository is created. In that case the

\* name of the repository will be based on the simple name of the aggregate's class.

\*/

String repository() default "";

Axon其实会为每一个Aggregate对应一个AggregateRepository，如果不额外指定，会使用给定的StorageEngine对应的Repository。

通常情况下，如果要在Repository里面保存Aggregate，需要执行repository.newInstance(()->new BankAccount())，但如果直接提供了构造器接受command，那么axon在执行这个command，如CreateAccountCommand时，会自动帮你做一个newInstance的操作。

另外，有人会说，为什么要把CommandHandler、EventHandler放到Aggregate内部，能不能放到外面单独用一个类。答案是当然可以。

Axon会自动扫描带有@CommandHandler,@EventHandler的方法，加载到KV值中。

并没有明确规定说这些方法一定得放在Aggregate内部或外部，不过一般应该把仅涉及当前Aggregate状态变化的，放到Aggregate内部处理，如果牵扯到其他复杂逻辑，如查询其他Aggregate做判断等，则最好是另起一个handler类。

1. ：Axon使用Jpa存储Aggregate状态

上一篇里，介绍了Axon的基本概念，并且做了一个最简单的hello例子。本篇将更进一步，完成两个小目标：

集成SpringBoot；

使用Standard Repository来存储Aggregate的最新状态。

**3. 使用Spring进行配置**

@Configuration

@EnableAxon

public class JpaConfig {

private static final Logger LOGGER = getLogger(JpaConfig.class);

@Autowired

private PlatformTransactionManager transactionManager;

@Bean

public EventStorageEngine eventStorageEngine(){

return new InMemoryEventStorageEngine();

}

@Bean

public TransactionManager axonTransactionManager() {

return new SpringTransactionManager(transactionManager);

}

@Bean

public EventBus eventBus(){ return new SimpleEventBus(); }

@Bean

public CommandBus commandBus() {

SimpleCommandBus commandBus =

new SimpleCommandBus(axonTransactionManager(), NoOpMessageMonitor.INSTANCE);

//commandBus.registerHandlerInterceptor(transactionManagingInterceptor());

return commandBus;

}

@Bean

public TransactionManagingInterceptor transactionManagingInterceptor(){

return new TransactionManagingInterceptor(new SpringTransactionManager(transactionManager));

}

@Bean

public EntityManagerProvider entityManagerProvider() {

return new ContainerManagedEntityManagerProvider();

}

@Bean

public Repository<BankAccount> accountRepository(){

return new GenericJpaRepository<BankAccount>(entityManagerProvider(),BankAccount.class, eventBus());

}

}

@EnableAxon会启用SpringAxonAutoConfigurer，后者会自动把上线文里的关键配置模块注入到Axon的config中。但这个注解未来会被替代，所以推荐使用方式为引入axon-spring-boot-autoconfigure包。下一篇文章就会介绍如何使用autoconfigure进行配置。

在本例中，我们把Event保存在内存中，所以指定EventStoreEngine为InMemoryEventStorageEngine。

前一篇说过，Axon会给每一个Aggregate创建一个AggregateRepositoryBean，来指定每一个Aggregate的实际Repository。这里我们直接声明BankAccount对应的Repository为一个GenericJpaRepository,来直接保存Aggregate的状态。GenericJpaRepository要求提供一个EntityManagerProvider，该Provider会提供具体的EntityManager来管理持久化。

值得注意的是，CommandBus在初始化时，需要提供一个TransactionManager，如果直接调用SimpleCommandBus的无参构造器，默认是NoTransactionManager.INSTANCE。本例测试时把几个command放在一个线程里串行执行，如果不提供TransactionManager，那么最终withdraw会失败。

提供TransactionManager的方式有两种：

1 如上例中直接构造器中指定；

2 注册一个TransactionManagingInterceptor；

**4. 把Aggregate加上JPA的标准Entity注解**

@Aggregate(repository = "accountRepository")

@Entity

public class BankAccount {

@AggregateIdentifier

private AccountId accountId;

@Id

public String getAccountId() { return accountId.toString(); }

@Column

public String getAccountName() { return accountName; }

@Column

public BigDecimal getBalance() { return balance; }

}

repository = “accountRepository”指定了该Aggregate对应的Repository的Bean名字，即在JpaConfig中定义的那一个。

JPA要求Entity必须有一个ID，GenericJpaRepository默认使用String作为EntityId的类型，而这里并没有直接用String，将会在存储时报

java.lang.IllegalArgumentException: Provided id of the wrong type for class com.edi.learn.axon.aggregates.BankAccount.

Expected: class com.edi.learn.axon.domain.AccountId, got class java.lang.String

解决方法是把@Id，@Column加在getter方法上。

**5. 配置controller接受请求并发送command**

@RestController

@RequestMapping("/bank")

public class BankAccountController {

@RequestMapping(method = RequestMethod.POST)

public void create() {

LOGGER.info("start");

AccountId id = new AccountId();

LOGGER.debug("Account id: {}", id.toString());

commandGateway.send(new CreateAccountCommand(id, "MyAccount",1000));

commandGateway.send(new WithdrawMoneyCommand(id, 500));

commandGateway.send(new WithdrawMoneyCommand(id, 300));

commandGateway.send(new CreateAccountCommand(id, "MyAccount", 1000));

commandGateway.send(new WithdrawMoneyCommand(id, 500));

}

}

我这里是为了偷懒，直接一个post请求就可以执行一堆操作。有心者可以改下，接受参数，根据参数发送command。

**6. 启动类**

@SpringBootApplication

@ComponentScan(basePackages = {"com.edi.learn"})

public class Application {

public static void main(String args[]){

SpringApplication.run(Application.class, args);

}

}

唯一需要注意的是，如果Application类不在JpaConfig包路径的前面，JpaConfig讲不会被Spring扫描注册到上下文中，需要指定包路径。

启动后，在http://localhost:8080/bank 发送一个POST请求，就可以看到log

17:53:47.099 [http-nio-8080-exec-1] INFO c.e.l.a.c.aggregates.BankAccount - Account 2fabef76-80bc-4dfc-8f21-4b68c5969fa5 is created with balance 1000

可以看到故意发送的第二个CreateAccountCommand时，由于id相同，提示创建失败。

进一步取钱时，因余额不足报错 。

本文源码：https://github.com/EdisonXu/sbs-axon/tree/master/lesson-2

1. Axon使用EventSourcing和AutoConfigure

继上一篇集成SpringBoot后，本篇将继续完成小目标：

使用EventSourcing

使用AutoConfigure配置Axon

前一篇中看到配置Axon即便在Spring中也是比较麻烦的，好在Axon提供了spring-boot-autoconfigure，提供了Spring下的一些默认配置，极大地方便了我们的工作。

启用也是非常方便的，在上一篇的基础上，我们只需要干三件事即可达成目标：

1 引入spring-boot-autoconfigure

2 删除JpaConfig类

3 去除BankAccount中的Entity声明

由于提供的application.properties里关于数据库的配置信息本身就是符合SpringDatasource定义的，所以，SpringBoot在检测到该配置后自动启用JPA。

spring-boot-autoconfigure中AxonAutoConfiguration类帮我们提供了最常用的CommandBus、EventBus、EventStorageEngine、Serializer、EventStore等，所以可以直接运行了。

在该类中有一段

@ConditionalOnBean(EntityManagerFactory.class)

@RegisterDefaultEntities(packages = {"org.axonframework.eventsourcing.eventstore.jpa",

"org.axonframework.eventhandling.tokenstore",

"org.axonframework.eventhandling.saga.repository.jpa"})

@Configuration

public static class JpaConfiguration {

@ConditionalOnMissingBean

@Bean

public EventStorageEngine eventStorageEngine(EntityManagerProvider entityManagerProvider,

TransactionManager transactionManager) {

return new JpaEventStorageEngine(entityManagerProvider, transactionManager);

}

@ConditionalOnMissingBean

@Bean

public EntityManagerProvider entityManagerProvider() {

return new ContainerManagedEntityManagerProvider();

}

@ConditionalOnMissingBean

@Bean

public TokenStore tokenStore(Serializer serializer, EntityManagerProvider entityManagerProvider) {

return new JpaTokenStore(entityManagerProvider, serializer);

}

@ConditionalOnMissingBean(SagaStore.class)

@Bean

public JpaSagaStore sagaStore(Serializer serializer, EntityManagerProvider entityManagerProvider) {

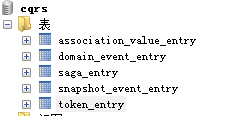
return new JpaSagaStore(serializer, entityManagerProvider);

}

}

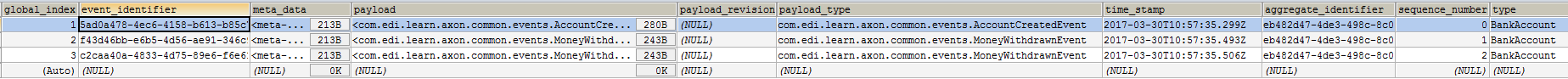
所以，当我们提供了JPA相关配置，以及mysql-connector后，这些Bean也会被启用，可以看到里面默认的EventStoreEngine就是JpaEventStorageEngine。

执行后，我们可以看到数据库中创建了如下表



其中domain\_event\_entry就是用来保存对Aggregate状态造成改变的所有Event的表。如果不做特别声明，所有Event都会记录在这张表里。

表内容



其中，比较重要的字段有：

pay\_load Event的具体内容

pay\_load\_type Event的类型，Axon在ES(Event Sourcing)时会通过这个反射出来原来的Java class

time\_stamp 该Event发生的时间

aggregate\_identifier event所对应Aggregate的唯一标识，在ES时，只有相同identifier的event才会一起回溯

sequence\_number 同一Aggregate对应的event发生的序列号，回溯时严格按照该顺序

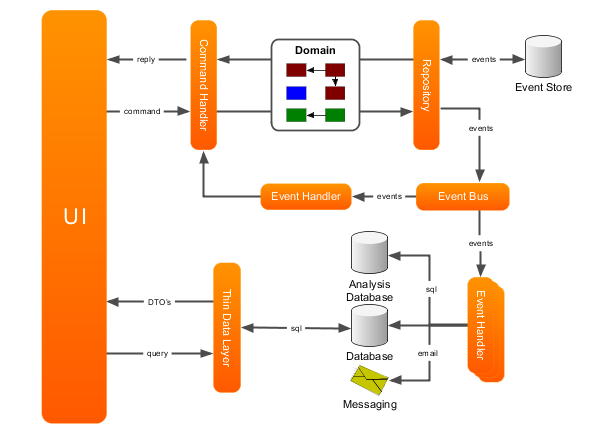
值得注意的是，在使用EventSourcing时，由于Aggregate本身的状态是通过ES获得的，所以所有对于Aggregate状态变化的动作一定都是放在@EventHandler里的，否则将会造成状态丢失。

预告一下，基本介绍已经完毕，下一篇开始，进入复杂的实现。

1. 第一个正式Axon例子

设计

回顾一下使用AxonFramework应用的架构



Command端Repository和Query端的Database是解耦的，完全可以使用不同的持久化技术，我们来尝试用MongoDB做Command端的Repository，而MySQL做Query的数据库。

例子描述

我们尝试完成一个简单的case：后台人员创建商品，用户选定若干商品后下单购买。

**商品定义**：Product(id, name, stock, price)

商品创建流程：

CreateProductCommand -> new ProductAggregate instance -> ProductCreatedEvent

**订单定义**： Order(id, username, payment, products)

订单创建流程：

CreateOrderCommand -> new OrderAggregate instance -> OrderCreatedEvent

创建商品时，我们只接收商品ID，去查询商品的具体信息，这样来学习如何在handler内去查询Aggregate。

Command端实现

Command端实现与前面几篇文章基本一致，需要定义Aggregate、Command，然后提供配置即可。

**Aggregate**

ProductAggregate

@Aggregate

public class ProductAggregate {

private static final Logger LOGGER = getLogger(ProductAggregate.class);

@AggregateIdentifier

private String id;

private String name;

private int stock;

private long price;

public ProductAggregate() {

}

@CommandHandler

public ProductAggregate(CreateProductCommand command) {

apply(new ProductCreatedEvent(command.getId(),command.getName(),command.getPrice(),command.getStock()));

}

@EventHandler

public void on(ProductCreatedEvent event){

this.id = event.getId();

this.name = event.getName();

this.price = event.getPrice();

this.stock = event.getStock();

LOGGER.debug("Product [{}] {} {}x{} is created.", id,name,price,stock);

}

// getter and setter

......

}

OrderAggregate

@Aggregate

public class OrderAggregate {

@AggregateIdentifier

private OrderId id;

private String username;

private double payment;

@AggregateMember

private Map<String, OrderProduct> products;

public OrderAggregate(){}

*public OrderAggregate(OrderId id, String username, Map<String, OrderProduct> products) {*

*apply(new OrderCreatedEvent(id, username, products));*

*}*

@EventHandler

public void on(OrderCreatedEvent event){

this.id = event.getOrderId();

this.username = event.getUsername();

this.products = event.getProducts();

computePrice();

}

private void computePrice() {

products.forEach((id, product) -> {

payment += product.getPrice() \* product.getAmount();

});

}

/\*\*

\* Divided 100 here because of the transformation of accuracy

\*

\* @return

\*/

public double getPayment() {

return payment/100;

}

public void addProduct(OrderProduct product){

this.products.put(product.getId(), product);

payment += product.getPrice() \* product.getAmount();

}

public void removeProduct(String productId){

OrderProduct product = this.products.remove(productId);

payment = payment - product.getPrice() \* product.getAmount();

}

}

这里，我并没有像ProductAggregate一样，把CreateOrderCommand放到OrderAggregate的构造器中去处理，原因是在创建订单时，由于需要知道商品的单价，所以要根据商品id查询商品信息，因为涉及到了其他Aggregate操作，特地单独创建一个OrderHandler来处理。

@Component

public class OrderHandler {

@CommandHandler

public void handle(CreateOrderCommand command) throws Exception {

Map<String, OrderProduct> products = new HashMap<>();

command.getProducts().forEach((productId,number)->{

LOGGER.debug("Loading product information with productId: {}",productId);

Aggregate<ProductAggregate> aggregate = productRepository.load(productId);

products.put(productId,

new OrderProduct(productId,

aggregate.invoke(productAggregate -> productAggregate.getName()),

aggregate.invoke(productAggregate -> productAggregate.getPrice()),

number));

});

repository.newInstance(() -> new OrderAggregate(command.getOrderId(), command.getUsername(), products));

}

}

如果查看org.axonframework.commandhandling.model.Repository<T>接口的定义，会发现里面只有三个方法：

public interface Repository<T> {

/\*\*

\* 使用给定的唯一标识符加载聚合。加载聚合时不进行版本检查，表示不检查并发访问

\*

\* @param aggregateIdentifier The identifier of the aggregate to load

\* @return The aggregate root with the given identifier.

\* @throws AggregateNotFoundException if aggregate with given id cannot be found

\*/

Aggregate<T> load(String aggregateIdentifier);

/\*\*

\* 使用给定的唯一标识符加载聚合。

\*

\* @param aggregateIdentifier The identifier of the aggregate to load

\* @param expectedVersion The expected version of the loaded aggregate

\* @return The aggregate root with the given identifier.

\* @throws AggregateNotFoundException if aggregate with given id cannot be found

\*/

Aggregate<T> load(String aggregateIdentifier, Long expectedVersion);

/\*\*

\* 使用给定的{@code factoryMethod}为聚合创建一个新的托管实例来实例化聚合的根。

\*

\* @param factoryMethod The method to create the aggregate's root instance

\* @return an Aggregate instance describing the aggregate's state

\* @throws Exception when the factoryMethod throws an exception

\*/

Aggregate<T> newInstance(Callable<T> factoryMethod) throws Exception;

}

有人会疑惑了，为什么没有Delete和Update？

先说update，这个Repository其实是对Aggregate的操作，EventSourcing中对Aggregate所有的变化都是通过Event来实现的，所以在调用apply(EventMessage)时，Event就已经被持久化了，EventHandler在处理该Event时，就已经实现了对Aggregate的update。

而Delete没有，很简单，EventSourcing脱胎于现实概念，你见过现实生活中把一个事物真正“delete”掉吗？估计得使用高能量子炮把东西轰成原子吧。

所以，只会有一个把这个Aggregate标为失效的标志，Axon中，在Aggregate内部可以直接调用markDeleted()来表示这个Aggregate被“delete”掉了，其实只是不能被load出来罢了。

由于Repository默认返回的是同一类型Aggregate，所以我们取属性就没那么简单了，只能通过invoke来调用get方法。是不是觉得很麻烦？因为其实CQRS压根不推荐直接从Repository直接query Aggregate来查询，而是调用Query端。

**Command**

command的实现因为都是POJO我就不贴代码了，可以直接看源码。

这里写一下基于SpringWeb的Controller类（引入spring-boot-starter-web包），以创建Product为例

@RestController

@RequestMapping("/product")

public class ProductController {

@Autowired

private CommandGateway commandGateway;

@RequestMapping(value = "/{id}", method = RequestMethod.POST)

public void create(@PathVariable(value = "id") String id,

@RequestParam(value = "name", required = true) String name,

@RequestParam(value = "price", required = true) long price,

@RequestParam(value = "stock",required = true) int stock,

HttpServletResponse response) {

try {

// multiply 100 on the price to avoid float number

CreateProductCommand command = new CreateProductCommand(id,name,price\*100,stock);

commandGateway.sendAndWait(command);

response.setStatus(HttpServletResponse.SC\_CREATED);// Set up the 201 CREATED response

return;

} catch (CommandExecutionException cex) {

LOGGER.warn("Add Command FAILED with Message: {}", cex.getMessage());

response.setStatus(HttpServletResponse.SC\_BAD\_REQUEST);

if (null != cex.getCause()) {

if (cex.getCause() instanceof ConcurrencyException) {

LOGGER.warn("A duplicate product with the same ID [{}] already exists.", id);

response.setStatus(HttpServletResponse.SC\_CONFLICT);

}

}

}

}

}

CommandGateway提供了四种发送Comman的方法：

1 send(command, CommandCallback) 发送command，根据执行结果调用CommandCallback中的onSuccess或onFailure方法

2 sendAndWait(command) 发送完command，等待执行完成并返回结果

3 sendAndWait(command, timeout, TimeUnit) 这个好理解，比上面多了一个超时

4 send(command) 该方法返回一个CompletableFuture，不用等待command的执行，立刻返回。结果通过future获取。

**Repository**

由于我们要使用axon-mongo，而非默认的jpa，所以必须得手动指定两个Aggregate的Repository，以其中一个为例：

@Configuration

public class ProductConfig {

@Autowired

private EventStore eventStore;

@Bean

@Scope("prototype")

public ProductAggregate productAggregate(){

return new ProductAggregate();

}

@Bean

public AggregateFactory<ProductAggregate> productAggregateAggregateFactory(){

SpringPrototypeAggregateFactory<ProductAggregate> aggregateFactory

= new SpringPrototypeAggregateFactory<>();

aggregateFactory.setPrototypeBeanName("productAggregate");

return aggregateFactory;

}

@Bean

public Repository<ProductAggregate> productAggregateRepository(){

EventSourcingRepository<ProductAggregate> repository = new EventSourcingRepository<ProductAggregate>(

productAggregateAggregateFactory(),

eventStore

);

return repository;

}

}

使用EventSourcingRepository，必须指定一个AggregateFactory用来反射生成Aggregate的，所以我们这里定义了Aggregate的prototype，并把它注册到AggregateFactory中去。

这样在系统启动时，读取历史Event进行ES还原时，就可以真实再现Aggregate的状态。

**配置**

前面使用MySQL作为EventStorage是不是感到不爽，那么我们通过引入axon-mongo依赖，使用MongoDB来做EventStorage。

pom的修改我就不写了，着重看下相关配置

先是修改application.property

通过Spring提供的@Value注解在具体的Configuration类里读取。

@Configuration

public class CommandRepositoryConfiguration {

@Value("${mongodb.url}")

private String mongoUrl;

@Value("${mongodb.dbname}")

private String mongoDbName;

@Value("${mongodb.events.collection.name}")

private String eventsCollectionName;

@Value("${mongodb.events.snapshot.collection.name}")

private String snapshotCollectionName;

@Bean

public Serializer axonJsonSerializer() { return new JacksonSerializer(); }

@Bean

public EventStorageEngine eventStorageEngine(){

return new MongoEventStorageEngine(

axonJsonSerializer(),null, axonMongoTemplate(), new DocumentPerEventStorageStrategy());

}

@Bean(name = "axonMongoTemplate")

public MongoTemplate axonMongoTemplate() {

MongoTemplate template = new DefaultMongoTemplate(mongoClient(), mongoDbName, eventsCollectionName, snapshotCollectionName);

return template;

}

@Bean

public MongoClient mongoClient(){

MongoFactory mongoFactory = new MongoFactory();

mongoFactory.setMongoAddresses(Arrays.asList(new ServerAddress(mongoUrl)));

return mongoFactory.createMongo();

}

}

用Jacson做序列化器，MongoClient提供了具体连接实现，MongoTemplate指定了db名称、存放event的collection名称、存放snapshot的collection名称。（snapshot的概念以后再解释）中间一个参数是做不同版本Event间兼容的，我们先留null。

EventStorageEngine指定MongoEventStorageEngine，spring-boot-autoconfigure中的AxonAutoConfiguration就会帮你把它注入到Axon的配置器中。

这里指的注意的是，使用Jackson做序列化器时，对应的entity的所有需要持久化的field必须都有public getter方法，因为Jackson在反射时默认只读public修饰符的field，否则就会报

com.fasterxml.jackson.databind.JsonMappingException: No serializer found for class com.edi.learn.axon.common.domain.OrderId and no properties discovered to create BeanSerializer (to avoid exception, disable SerializationFeature.FAIL\_ON\_EMPTY\_BEANS) (through reference chain: com.edi.learn.axon.common.events.OrderCreatedEvent[“orderId”])

错误。如果确实不想写，那么在Entity的class声明前加上@JsonAutoDetect(fieldVisibility=JsonAutoDetect.Visibility.ANY)

到此，Command端的实现已基本完成（Event我没写，因为与前文类似），那么我们来看看**Query端。**

Query端实现

AxonFramework的Query端其实并没有特别的，我们只需要实现一些EventHandler来处理Command端产生的事件，来更新Query端的数据库就行了。

这里我就使用JPA的MySQL实现，spring提供了spring-boot-starter-data-rest，为JPA Repository增加了HateOas风格的REST接口，非常简单，非常方便，堪称无脑。

先定义三个Entity

@Entity

public class ProductEntry {

public ProductEntry() {

}

public ProductEntry(String id, String name, long price, int stock) {

}

// getter & setter

......

}

@Entity

public class OrderEntry {

@Id

private String id;

@Column

private String username;

@Column

private double payment;

@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)

@JoinColumn(name = "order\_id")

@MapKey(name = "id")

private Map<String, OrderProductEntry> products;

public OrderEntry() {

}

public OrderEntry(String id, String username, Map<String, OrderProductEntry> products) {

}

// getter & setter

......

}

@Entity

public class OrderProductEntry {

@Id

@GeneratedValue

private Long jpaId;

private String id;

public OrderProductEntry() {

}

public OrderProductEntry(String id, String name, long price, int amount) {

}

}

比较简单，唯一需要注意的是ProductEntry和OrderEntry之间的一对多关系。

然后为它们创建两个Repository

@RepositoryRestResource(collectionResourceRel = "orders", path = "orders")

public interface OrderEntryRepository extends PagingAndSortingRepository<OrderEntry, String> {}

@RepositoryRestResource(collectionResourceRel = "products", path = "products")

public interface ProductEntryRepository extends PagingAndSortingRepository<ProductEntry, String> {}

是不是很简单？最后定义handler，为省篇幅，我只写一个

@Component

public class OrderEventHandler {

@EventHandler

public void on(OrderCreatedEvent event){

Map<String, OrderProductEntry> map = new HashMap<>();

event.getProducts().forEach((id, product)->{

map.put(id,

new OrderProductEntry(

product.getId(),

product.getName(),

product.getPrice(),

product.getAmount()));

});

OrderEntry order = new OrderEntry(event.getOrderId().toString(), event.getUsername(), map);

repository.save(order);

}

}

**启动类**

由于我们使用了axon提供的MongoEventStorageEngine，其内部也使用了JPA，所以我们在启动类还需要把Axon帮我们转Entity的一些类也加到EntityScan中去

@SpringBootApplication

@ComponentScan(basePackages = {"com.edi.learn"})

@EntityScan(basePackages = {"com.edi.learn",

"org.axonframework.eventsourcing.eventstore.jpa",

"org.axonframework.eventhandling.saga.repository.jpa",

"org.axonframework.eventhandling.tokenstore.jpa"})

@EnableJpaRepositories(basePackages = {"com.edi.learn.axon.query"})

public class Application {

private static final Logger LOGGER = getLogger(Application.class);

public static void main(String args[]){

SpringApplication.run(Application.class, args);

}

}

启动后，用POST发送请求http://127.0.0.1:8080/product/1?name=ttt&price=10&stock=100 ，查询mongoDB：

> use axon

> show collections

events

snapshots

system.indexes

> db.events.find().pretty()

{

"\_id" : ObjectId("58dd181073bc0c0fb86d895e"),

"aggregateIdentifier" : "1",

"type" : "ProductAggregate",

"sequenceNumber" : NumberLong(0),

"serializedPayload" : "{\"id\":\"1\",\"name\":\"ttt\",\"price\":1000,\"stock\":100}",

"timestamp" : "2017-03-30T14:37:04.075Z",

"payloadType" : "com.edi.learn.axon.common.events.ProductCreatedEvent",

"payloadRevision" : null,

"serializedMetaData" : "{\"traceId\":\"4a298ed4-0d53-402a-ae6b-d79cc5e193bf\",\"correlationId\":\"4a298ed4-0d53-402a-ae6b-d79cc5e193bf\"}",

"eventIdentifier" : "500f3a8f-7c02-4e8e-bb9c-7b676224ce5c"

}

可以看到生成的EventMessage，与前篇文章中MySQL表里内容基本一致。

再去看下MySQL库的product\_entry表，有记录

1. Saga的使用

在上一篇里面，我们正式的使用了CQRS模式完成了AXON的第一个真正的例子，但是细心的朋友会发现一个问题，创建订单时并没有检查商品库存。

库存是否足够直接回导致订单状态的成功与否，在并发时可能还会出现超卖。当库存不足时还需要回滚订单，所以这里出现了复杂的跨Aggregate事务问题。

Saga就是为解决这里复杂流程而生的。

Saga

Saga 这个名词最早是由Hector Garcia-Molina和Kenneth Salem写的Sagas这篇论文里提出来的，但其实Saga并不是什么新事物，在我们传统的系统设计中，它有个更熟悉的名字——“ProcessManager”，只是换了个马甲，还是干同样的事——组合一组逻辑处理复杂流程。

但它与我们平常理解的“ProgressManager”又有不同，它的提出，最早是是为了解决分布式系统中长时间运行事务(long-running business process)的问题，把单一的transaction按照步骤分成一组若干子transaction，通过补偿机制实现最终一致性。

举个例子，在一个交易环节中有下单支付两个步骤，如果是传统方式，两个步骤在一个事务里，统一成功或回滚，然而如果支付时间很长，那么就会导致第一步，即下单这里所占用的资源被长时间锁定，可能会对系统可用性造成影响。如果用Saga来实现，那么下单是一个独立事务，下单的事务先提交，提交成功后开始支付的事务，如果支付成功，则支付的事务也提交，整个流程就算完成，但是如果支付事务执行失败，那么支付需要回滚，因为这时下单事务已经提交，则需要对下单操作进行补偿操作（可能是回滚，也可能是变成新状态）。

可以看到Saga是牺牲了数据的强一致性，实现最终一致性。

Saga的概念使得强一致性的分布式事务不再是唯一的解决方案，通过保证事务中每一步都可以一个补偿机制，在发生错误后执行补偿事务来保证系统的可用性和最终一致性。

在CQRS中，我们尽量遵从“聚合尽量设计的小，且一次修改只修改一个聚合”的原则（与OO中高内聚，低耦合的原则相同），所以当我们需要完成一个复杂流程时，就可能涉及到对多个Aggregate状态的改变，我们就可以把整个过程管理统一放到Saga来定义。

设计

把我们的订单创建流程修改成以下：

创建Command和Event

在上一篇例子的基础上，创建如下Command和Event

-ReserveProductCommand (orderId, productId, number)

-RollbackReservationCommand (orderId, productId, number)

-ConfirmOrderCommand (orderId)

-RollbackOrderCommand (orderId)

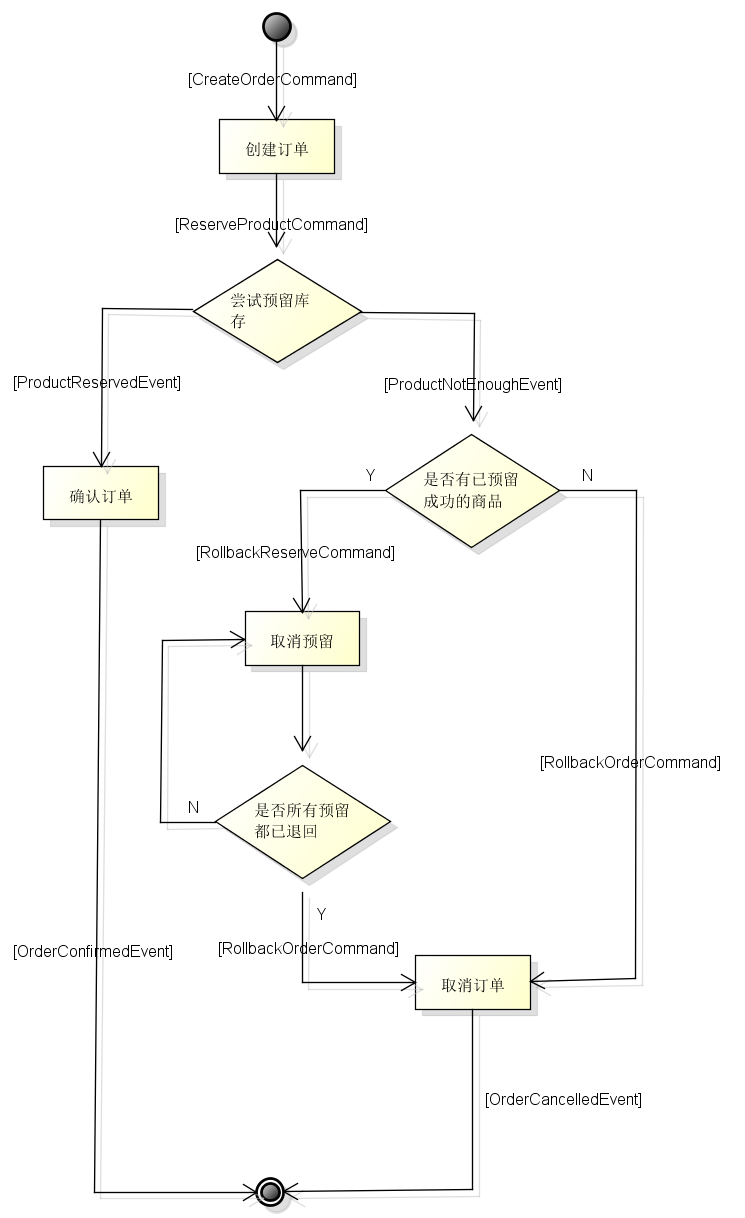
-ProductReservedEvent (orderId, productId, number)

-ProductNotEnoughEvent (orderId, productId)

-OrderCancelledEvent (orderId)

-OrderConfirmedEvent (orderId)

都是POJO，这里我就不放代码了。具体可以去源代码看。



创建Saga

@Saga

public class OrderSaga {

private static final Logger LOGGER = getLogger(OrderSaga.class);

private OrderId orderIdentifier;

private Map<String, OrderProduct> toReserve;

private Map<String, OrderProduct> toRollback;

private int toReserveNumber;

private boolean needRollback;

@Autowired

private transient CommandGateway commandGateway;

@StartSaga

@SagaEventHandler(associationProperty = "orderId")

public void handle(OrderCreatedEvent event){

this.orderIdentifier = event.getOrderId();

this.toReserve = event.getProducts();

toRollback = new HashMap<>();

toReserveNumber = toReserve.size();

event.getProducts().forEach((id,product)->{

ReserveProductCommand command = new ReserveProductCommand(orderIdentifier, id, product.getAmount());

commandGateway.send(command);

});

}

@SagaEventHandler(associationProperty = "orderId")

public void handle(ProductNotEnoughEvent event){

LOGGER.info("No enough item to buy");

toReserveNumber--;

needRollback=true;

if(toReserveNumber==0)

tryFinish();

}

private void tryFinish() {

if(needRollback){

toReserve.forEach((id, product)->{

if(!product.isReserved())

return;

toRollback.put(id, product);

commandGateway.send(new RollbackReservationCommand(orderIdentifier, id, product.getAmount()));

});

if(toRollback.isEmpty())

commandGateway.send(new RollbackOrderCommand(orderIdentifier));

return;

}

commandGateway.send(new ConfirmOrderCommand(orderIdentifier));

}

@SagaEventHandler(associationProperty = "orderId")

public void handle(ReserveCancelledEvent event){

toRollback.remove(event.getProductId());

if(toRollback.isEmpty())

commandGateway.send(new RollbackOrderCommand(event.getOrderId()));

}

@SagaEventHandler(associationProperty = "id", keyName = "orderId")

@EndSaga

public void handle(OrderCancelledEvent event) throws OrderCreateFailedException {

LOGGER.info("Order {} is cancelled", event.getId());

// throw exception here will not cause the onFailure() method in the command callback

//throw new OrderCreateFailedException("Not enough product to reserve!");

}

@SagaEventHandler(associationProperty = "orderId")

public void handle(ProductReservedEvent event){

OrderProduct reservedProduct = toReserve.get(event.getProductId());

reservedProduct.setReserved(true);

toReserveNumber--;

if(toReserveNumber ==0)

tryFinish();

}

@SagaEventHandler(associationProperty = "id", keyName = "orderId")

@EndSaga

public void handle(OrderConfirmedEvent event){

LOGGER.info("Order {} is confirmed", event.getId());

}

}

Saga的启动和结束

Axon中通过@Saga注解标识Saga。Saga有起点和终点，必须以@StartSaga和@EndSaga区分清楚。一个Saga的起点可能只有一个，但终点可能有好几个，对应流程的不同结果。

默认情况下，只有在找不到同类型已存在的Saga instance时，才会创建一个新的Saga。但是可以通过更改@StartSaga中的forceNew为true让它每次都新建一个。

只有当@EndSaga对应的方法被顺利执行，Saga才会结束，但也可以直接从Saga内部调用end()方法强制结束。

EventHandling

Saga通过@SagaEventHandler注解来标明EventHandler，与普通EventHandler基本一致，唯一的不同是，普通的EventHandler会接受所有对应的Event，而Saga的EventHandler只处理与其关联过的Event。

当被注解@StartSaga的方法调用时，axon默认会根据当前@SagaEventHandler中的associationProperty去找Event中的field，然后把它的值与当前Saga进行关联，类似<saga\_id,<key,value>>这种形式。

一旦产生关联，该Saga在遇到同一Event时，只会处理<key,value>与已关联值完全一致的Event。例如，有两个OrderCreatedEvent，我们定义associationProperty ="orderId"，两个event的orderId分别为1、2，当Saga创建时接受了orderId=1的OrderCreatedEvent后，值为2的Event它就不再处理了。

也可以在Saga内直接调用associateWith(String key, String/Number value)来做这个关联。例如，

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

associateWith("shipmentId", shipmentId);

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) { end(); }

}

@SagaEventHandler(associationProperty = "invoiceId")

public void handle(InvoicePaidEvent event) {

paid = true;

if (delivered) { end(); }

}

// ...

}

有时我们可能并不想直接使用Event里field的名称作为associationProperty的值，可以使用keyName来对应field名称。

Saga是靠Event驱动的，但有时command发出去了，并没有在规定时间内收到预期的Event怎么办？Saga提供了EventScheduler，通过Java内置的scheduler或Quarz，定时自动发送一个Event到这个Saga。

Saga的执行是在独立的线程里，所以我们无法通过commandgateway的sendAndWait方法等到其返回值或捕获异常。

Saga Store

由于Sage在处理过程中也存在中间状态，而Saga的一些业务流程可能会执行很长时间，比如好几天，那么万一系统重启Saga的状态就丢失了，所以Saga也需要能够通过ES恢复，即指定一个SagaStore。

SagaStore与EventStore的使用除了名字外，基本没有任何区别，也内置了InMemory,JPA,jdbc,Mongo四种实现这里我就不多叙述了。

注意！当持久化Saga时，对于注入的资源field，如CommandGateway，一定要加上transient修饰符，这样Serializer才不会去序列化这个field。当Saga从Repository读出来的时候，会自动注入相关的资源。

只需要显示的提供一个SagaStore的配置就可以了。当启用JPA时，默认会启动JpaSagaStore。我们这里使用MongoSagaStore，修改AxonConfiguration如下：

@Configuration

public class AxonConfiguration {

.....

@Bean

public SagaStore sagaStore(){

org.axonframework.mongo.eventhandling.saga.repository.MongoTemplate mongoTemplate =

new org.axonframework.mongo.eventhandling.saga.repository.DefaultMongoTemplate(mongoClient(), mongoDbName, "sagas");

return new MongoSagaStore(mongoTemplate, axonJsonSerializer());

}

}

在@StartSaga执行后，会把当前Saga插入到指定的SagaStore中，当@EndSaga执行时，axon会自动的从SagaStore中删除该Saga。

修改Handler

由于ReserveProductCommand和RollbackReservationCommand是需要查找原ProductAggregate的，所以单独创建一个ProductHandler

ProductHandler

@Component

public class ProductHandler {

private static final Logger LOGGER = getLogger(ProductHandler.class);

@Autowired

private Repository<ProductAggregate> repository;

@CommandHandler

public void on(ReserveProductCommand command){

Aggregate<ProductAggregate> aggregate = repository.load(command.getProductId());

aggregate.execute(aggregateRoot->aggregateRoot.reserve(command.getOrderId(), command.getNumber()));

}

@CommandHandler

public void on(RollbackReservationCommand command){

Aggregate<ProductAggregate> aggregate = repository.load(command.getProductId());

aggregate.execute(aggregateRoot->aggregateRoot.cancellReserve(command.getOrderId(), command.getNumber()));

}

}

修改ProductAggregate，增加对应的方法和handler

ProductAggregate

@Aggregate

public class ProductAggregate {

public void reserve(OrderId orderId, int amount){

if(stock>=amount) {

apply(new ProductReservedEvent(orderId, id, amount));

}else

apply(new ProductNotEnoughEvent(orderId, id));

}

public void cancellReserve(OrderId orderId, int amount){

apply(new ReserveCancelledEvent(orderId, id, stock));

}

@EventHandler

public void on(ProductReservedEvent event){

int oriStock = stock;

stock = stock - event.getAmount();

LOGGER.info("Product {} stock change {} -> {}", id, oriStock, stock);

}

@EventHandler

public void on(ReserveCancelledEvent event){

stock +=event.getAmount();

LOGGER.info("Reservation rollback, product {} stock changed to {}", id, stock);

}

}

Order这边对应也要修改Aggregate和handler

OrderHandler

@Component

public class OrderHandler {

@CommandHandler

public void handle(RollbackOrderCommand command){

Aggregate<OrderAggregate> aggregate = repository.load(command.getOrderId().getIdentifier());

aggregate.execute(aggregateRoot->aggregateRoot.delete());

}

@CommandHandler

public void handle(ConfirmOrderCommand command){

Aggregate<OrderAggregate> aggregate = repository.load(command.getId().getIdentifier());

aggregate.execute(aggregateRoot->aggregateRoot.confirm());

}

}

OrderAggregate

@Aggregate

public class OrderAggregate {

private String state="processing"; // 增加一个属性订单状态

......

@EventHandler

public void on(OrderConfirmedEvent event){

this.state = "confirmed";

}

@EventHandler

public void on(OrderCancelledEvent event){

this.state = "deleted";

markDeleted();

}

}

启动测试

其他地方基本没有什么改动，为方便起见，我把Query端也改成MongoDB了，方法比较简单，就引入spring-boot-starter-data-mongodb包，启动类里将@EnableJpaRepositories改成@EnableMongoRepositories，然后把Queyr端的Entry类包含在Scan的范围内就好了。

@SpringBootApplication

@ComponentScan(basePackages = {"com.edi.learn"})

@EntityScan(basePackages = {"com.edi.learn",

"org.axonframework.eventsourcing.eventstore.jpa",

"org.axonframework.eventhandling.saga.repository.jpa",

"org.axonframework.eventhandling.tokenstore.jpa"})

@EnableMongoRepositories(basePackages = {"com.edi.learn"})

public class Application {

private static final Logger LOGGER = getLogger(Application.class);

public static void main(String args[]){

SpringApplication.run(Application.class, args);

}

}

执行后，

POST请求到http://127.0.0.1:8080/product/1?name=ttt&price=10&stock=100 创建商品；

POST如下JSON到http://127.0.0.1:8080/order 来创建订单

{

"username":"Edison",

"products":[{

"id":1,

"number":90

}]

}

再创建一次

可以看到控制台打印

09:39:10.648 [http-nio-8080-exec-1] DEBUG c.e.l.a.c.w.c.ProductController - Adding Product [1] 'ttt' 10x100

09:39:10.675 [http-nio-8080-exec-1] DEBUG c.e.l.a.c.a.ProductAggregate - Product [1] ttt 1000x100 is created.

09:39:10.853 [http-nio-8080-exec-1] DEBUG c.e.l.a.q.h.ProductEventHandler - repository data is updated

09:39:21.640 [http-nio-8080-exec-3] DEBUG c.e.l.a.c.a.ProductAggregate - Product [1] ttt 1000x100 is created.

09:39:21.681 [http-nio-8080-exec-3] INFO c.e.l.a.c.a.ProductAggregate - Product 1 stock change 100 -> 10

09:39:21.823 [http-nio-8080-exec-3] INFO c.e.l.axon.command.saga.OrderSaga - Order 8706dbaf-4511-4b01-b6c5-e24bec3f10a9 is confirmed

09:42:35.255 [http-nio-8080-exec-5] DEBUG c.e.l.a.c.handlers.OrderHandler - Loading product information with productId: 1

09:42:35.259 [http-nio-8080-exec-5] DEBUG c.e.l.a.c.a.ProductAggregate - Product [1] ttt 1000x100 is created.

09:42:35.263 [http-nio-8080-exec-5] INFO c.e.l.a.c.a.ProductAggregate - Product 1 stock change 100 -> 10

09:42:35.301 [http-nio-8080-exec-5] INFO c.e.l.axon.command.saga.OrderSaga - No enough item to buy

09:42:35.313 [http-nio-8080-exec-5] INFO c.e.l.axon.command.saga.OrderSaga - Order 6baba5e9-1173-48a8-ab98-cd51691ba9f5 is cancelled

重启程序，再创建一次订单后发送GET请求到http://127.0.0.1:8080/orders 查询订单

{

"\_embedded": {

"orders": [

{

"username": "Edison",

"payment": 0,

"status": "confirmed",

"products": {

"1": {

"name": "ttt",

"price": 1000,

"amount": 90

}

},

"\_links": {

"self": {

"href": "http://localhost:8080/orders/8706dbaf-4511-4b01-b6c5-e24bec3f10a9"

},

"orderEntry": {

"href": "http://localhost:8080/orders/8706dbaf-4511-4b01-b6c5-e24bec3f10a9"

}

}

},

{

"username": "Edison",

"payment": 0,

"status": "cancelled",

"products": {

"1": {

"name": "ttt",

"price": 1000,

"amount": 90

}

},

"\_links": {

"self": {

"href": "http://localhost:8080/orders/6baba5e9-1173-48a8-ab98-cd51691ba9f5"

},

"orderEntry": {

"href": "http://localhost:8080/orders/6baba5e9-1173-48a8-ab98-cd51691ba9f5"

}

}

},

{

"username": "Edison",

"payment": 0,

"status": "cancelled",

"products": {

"1": {

"name": "ttt",

"price": 1000,

"amount": 90

}

},

"\_links": {

"self": {

"href": "http://localhost:8080/orders/27a829af-cda1-43f4-af37-fbc597fe5f6f"

},

"orderEntry": {

"href": "http://localhost:8080/orders/27a829af-cda1-43f4-af37-fbc597fe5f6f"

}

}

}

]

},

"\_links": {

"self": {

"href": "http://localhost:8080/orders"

},

"profile": {

"href": "http://localhost:8080/profile/orders"

}

},

"page": {

"size": 20,

"totalElements": 3,

"totalPages": 1,

"number": 0

}

}

很明显看到只有第一个订单状态为’confirmed’，其他两个都是’cancelled’。重启后，Aggregate自动回溯后，对库存的判断也是正确的。

再做个小实验，我们修改OrderSaga，强制在确认订单时让线程sleep一段时间，然后去MongoDB里查看Saga信息

@SagaEventHandler(associationProperty = "id", keyName = "orderId")

@EndSaga

public void handle(OrderConfirmedEvent event) throws InterruptedException {

LOGGER.info("Order {} is confirmed", event.getId());

Thread.sleep(10000);

}

> db.sagas.find().pretty()

{

"\_id" : ObjectId("58df074d73bc0c10f4008eff"),

"sagaType" : "com.edi.learn.axon.command.saga.OrderSaga",

"sagaIdentifier" : "08a371f5-9d9a-48a7-b46e-9b8e86b8897b",

"serializedSaga" : BinData(0,"e30="),

"associations" : [

{

"key" : "orderId",

"value" : "5111a55e-1ddd-4434-aab8-635c004fc1eb"

}

]

}

看到我们的关联值了吧

1. DistributeCommand和DistributeEvent
2. AxonFramework与SpringCloud的整合