深入浅出Event Sourcing和CQRS

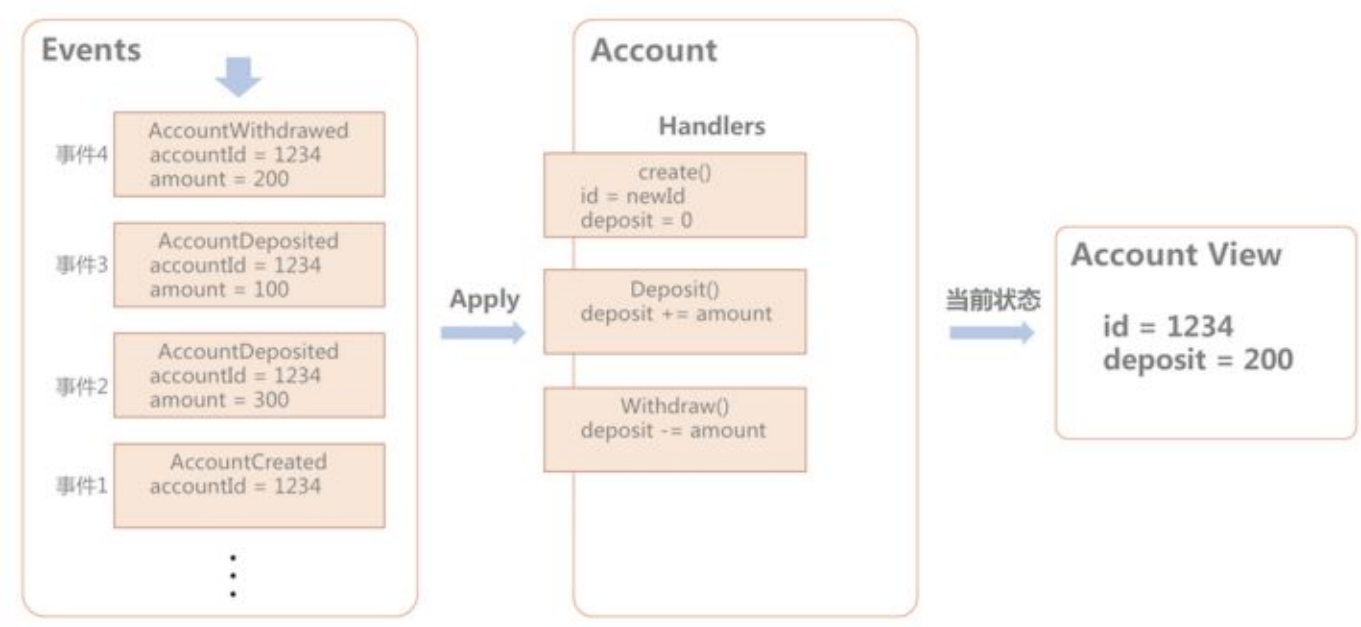
慕课网
已认证的官方帐号

Event Sourcing也叫事件溯源，是这些年另一个越来越流行的概念，是大神Martin Fowler提出的一种架构模式。简单来说，它有几个特点：

- 整个系统以事件为驱动，所有业务都由事件驱动来完成。
- 事件是一等公民，系统的数据以事件为基础，事件要保存在某种存储上。
- 业务数据只是一些由事件产生的视图，不一定要保存到数据库中。

什么是Event Sourcing

这么说可能还是比较难以理解，我们来举个栗子。这是一个账户余额管理的例子：



在这个图中，中间的是我们的账户对象，它有几个时间处理函数 `create()`，`deposit()`，`withdraw()`，分别用于处理新建账户、账户存款和取款的操作。

左边的就是一个个的事件，它是一个事件的流，根据用户请求或者从其他地方产生。在这里例子当中，有3个事件：`AccountCreated`，`AccountDeposited`，`AccountWithdrawed`，分别相当于账户创建的事件，存款的事件和取款的事件。当这些事件产生的时候，我们会触发上面的Account对象的相应的处理函数。

右边的就是这个Account对象处理完左边的4个事件以后，最新的数据状态。具体的处理过程就是：

1. 系统产生一个新建账户的事件 `AccountCreated`， `Account` 对象来处理这个事件，事件里面的Id是1234，系统先尝试着找id是1234的账户，发现没有，于是新建一个 `Account` 对象，并在它上面调用 `create()` 的处理函数，也就是初始化了id和余额。
2. 然后，有一个 `AccountDeposited` 的事件，对应的 `Account` 对象的id是1234，系统找到之前创建的对象，在它上面应用 `deposit()` 处理函数，也就是增加的余额的操作，更新了账户余额。
3. 系统又收到一个存款事件，跟上面一样，又更新了一次余额。
4. 收到一个取款事件，还是找到id是1234的账户，在它上面调用 `withdraw()` 的处理函数，进行取款操作，更新余额。
5. 最后，1234这个账户的余额是200，也就是右边的数据状态。

同时，上面的这些事件需要持久保存在数据库或其他地方，而account的数据状态却不需要保存，我们只是在需要获得account当前的数据状态的时候，通过这个account相关的事件，调用他们的处理函数，重新生成当前状态。当然，每次都这样调用处理函数势必会造成资源的浪费，因为它需要从数据库中取得所有这个account的事件，然后依次调用处理函数。所以一般我们可以把这个account的最新状态，以一种视图的方式保存在数据库中。

上面这个方式和过程，就是我们说的Event Sourcing，也就是以事件为源的处理模式。

Event Sourcing的构成

通过上面的例子，我们理解了Event Sourcing(事件溯源)，下面我们再来看看Event Sourcing包含哪些部分。

聚合对象

在上面的例子中， `Account` 对象就是一个聚合对象，它里面包含账户的基本信息，也包含了对账户操作时的处理方法，也就是几个事件处理函数。了解领域驱动设计的人这时候应该就想到，这个Account对象其实就是一个领域模型，Account这个领域模型需要的业务操作，由它自己提供。

每个聚合对象都有一个Id，用于唯一标识这个对象，所以系统中不同的账户就会有不同的Account对象。

Event Store

我们说了，在Event Sourcing模式当中，所有的事件都是要保存到数据库（或其他存储，下面就直接说数据库了）中的，这个存储就叫Event Store。

每个事件应该也包含一个它要处理的聚合对象的id，以及事件的顺序，查询的时候就是根据聚合对象的id从数据库中找到相关的事件，并按照生成的事件或序号排序。

Event Store除了提供事件数据的存储、查询功能以外，还可以提供事件的重现等功能。事件的重现，就是将截止某一个时间的所有事件取出来，调用他的处理函数，生成当时那个时间点的业务状态。所以在重现之前，如果我们的业务数据的状态，通过视图的形式保存到了数据库中，我们需要先清除相应的数据。正是由于Event Sourcing模式的这个以事件为源的特性，所以我们才有可能提供这样的历史重现的功能。

聚合资源库

一般情况下，我们的聚合对象的数据状态是不会保存在数据库当中的。每当系统要获得某一个账户的数据的时候，都是从Event Store当中取出所有相关聚合对象的事件，然后依次的调用这些事件的处理方法，“聚合”出该领域对象最新的数据状态。这个，就是聚合资源库需要提供的功能。

视图

上面我们也说了，如果每次都重新“聚合”出对象，获取当前的状态，会浪费很多资源。所以，我们可以在某个事件发生的时候，将这个聚合对象的最新数据状态，写到一个表中，这个表可以叫做物化视图。

查询

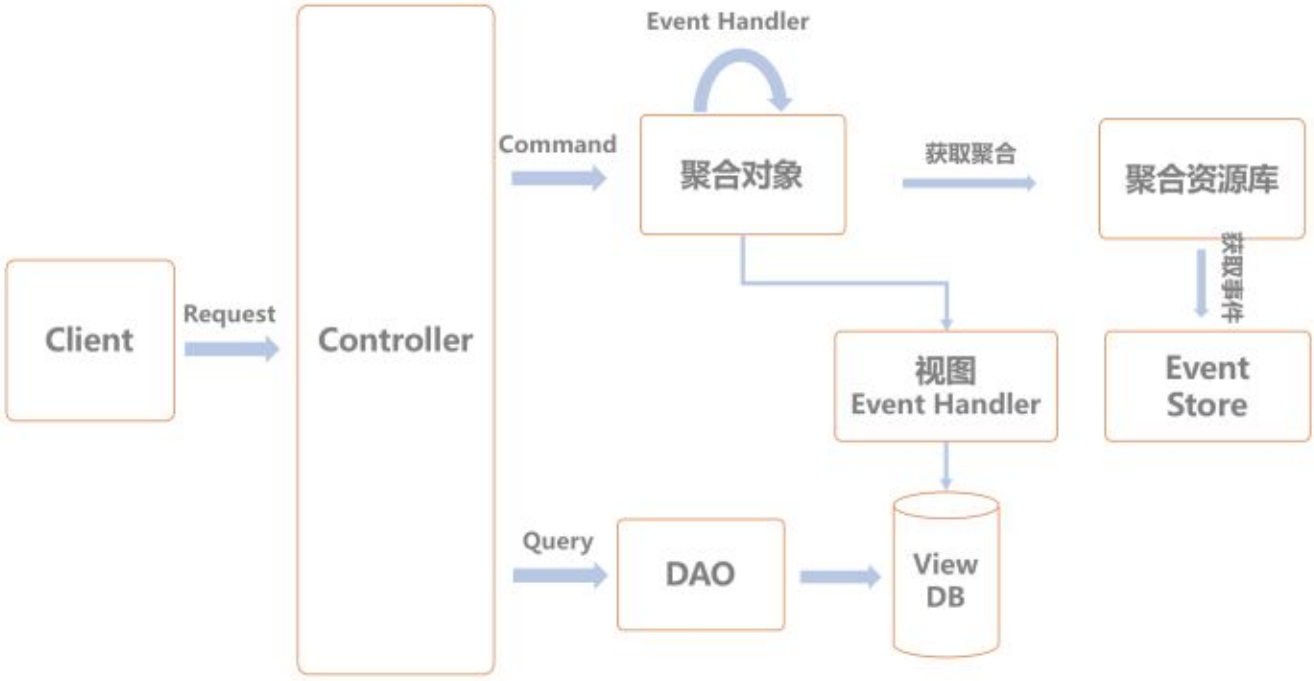
由于我们提供了专门的视图表，将聚合对象的最新状态保存在数据库中，那我们在查询的时候，可以通过该物化视图去查询，而不是通过聚合对象的资源库去查询。

Event Sourcing与CQRS

CQRS，是 Command Query Responsibility Segregation的缩写，也就是通常所说的读写隔离。在上面，我们说，为了性能考虑，将聚合对象的数据状态用物化视图的形式保存，可以用于数据的查询操作，也就是我们把数据的更新与查询的流程隔离开来。我们通过事件来更新聚合对象的数据状态，同时由另一个处理器处理相同的事件，来更新物化视图的数据。

所以，Event Sourcing与CQRS有着天然的联系，所以也经常会有人把他们放在一起讨论。实际上，CQRS是在使用Event Sourcing模式以后，又使用了物化视图的情况下，所产生的额外的好处。

下图就是使用Event Sourcing好CQRS模式以后的一个简单的流程图：



1. 对于Command类型的请求（需要修改数据），web层会走通过Event Sourcing更新聚合对象的流程，这时会有一个Event Handler的处理类监听相应事件，更新物化视图。
2. 对于Query类型的请求，web层会通过相应的DAO获取数据返回。

Event Sourcing的优点与缺点

Event Sourcing之所以会越来越受到关注，是因为它的一些优点：

1. 方便进行溯源与历史重现
   这个“溯源”的意思是，我们可以通过对保存的事件的分析，知道现在的系统的状态，是怎么一步一步的变成这样的。这在一个大型的业务复杂的应用系统里尤为有用。如果没有使用Event Sourcing模式，即使我们使用完备的log机制，提供log查询分析，也很难追溯数据的变化过程。此外，我们还可以根据历史的事件，重新生成所有的业务状态数据。我们甚至可以指定我要生成到具体某一时刻的状态。这就好像我们可以自由的穿梭在我们的系统运行过程当中，查看任何一个时间点的状态。
2. 方便Bug的修复
   由于我们可以重现历史，所以当发现一个bug以后，我们可以在修复完以后，直接重新聚合我们的业务数据，修复我们的数据。如果使用传统的设计方法，我们就需要通过SQL或者写一段程

序，去手动的修改数据库，以使它达到正常的状态。如果这个bug存在的时间较长，牵扯的数据较多，这将会是一个非常麻烦的事情。

同时，由于我们可以通过事件分析业务的过程，这也经常能帮助我们发现问题。

3. 能提供非常好的性能

   在Event Sourcing模式下，事件数据的保存是一个一直新增的写表操作，没有更新。这在很多情况下都能够提供一个非常好的写的性能，让系统的接收事件的吞吐量可以很高。

   然后聚合对象根据事件的聚合Id，获取所有相关的事件，“聚合”出对象，调用业务方法。但是它的结果又不需要写数据库，这里面只有一个读操作，其他的操作都是在内存中。

   最后，由另一个Event Handler处理这些事件，更新物化视图的表。在更新数据的时候，我们只需要锁记录，所以这个更新的过程也可以很快。

   通过这种模式，我们的系统的压力最终基本上都落在数据库上，整个系统里不会有太多锁和等待（只有并发的在同一个聚合对象上处理，才会等待，例如用户同时发了多个请求进行存款取款操作），就可以提供非常好的吞吐量。

4. 方便使用数据分析等系统

   这个就很明显了，用了Event Sourcing模式，我们的数据就是事件，我们只需要在现有的事件是加上需要的处理方法，来做数据分析；或者将event直接发送到某个分析系统。我们就不需要为了做数据分析，再在系统里定义各种事件，发送事件等。

当然，它也有一些缺点：

1. 开发思维的转变

   最大的难点，估计就是对开发人员的思维方式的转变。使用Event Sourcing模式，需要我们从设计角度，使用一定的领域驱动设计的方法，从开发角度，我们又需要使用基于事件的响应式编程思维。对于习惯了传统的面向对象的程序员来说，这都是一个不小的挑战。

2. 没有成熟完善的框架

   我们开发Java的应用，现在绝大多数情况下都会使用Spring，也有很大一部分使用Spring MVC（或Spring Boot）。不管怎么样，都是基于Spring这个框架家族进行开发。但是，对于Event Sourcing模式来说，还没有一个大而一统的框架，既能提高很好的Event Sourcing模式的实现，又能被广泛接受，最好还能有一些厂商提高商业服务，保证整个生态的良性发展。

3. 事件的结构的改变

   使用Event Sourcing模式，还有一个问题就是事件结构的改变。由于业务的变化，我们设计的事件，在结构上可能有一些改变，可能需要添加一些数据，或者删除一些数据。那么这时候，想要进行方才说的“历史重现”就会有问题。这时我们就需要通过某种方式给他提供兼容。

4. 从领域模型角度设计系统，而不是以数据库表为基础设计

   这其实不算是一个缺点，但是由于领域驱动设计并不是广泛使用的软件设计方式，很多开发人员对此不了解，相应的设计和开发方式也不熟悉，所以这也成为使用Event Sourcing模式开发需要解决的问题。

   领域驱动设计从业务分析出发，从领域模型设计着手设计一个系统，而在设计一个基于Event Sourcing模式的系统时，我们往往也要用到领域模型设计的一些方法，从领域模型设计开始，设计聚合对象和它的业务（事件），以及处理方法（Event Handler）。通过领域驱动设计，对复

杂的应用系统往往能提供更好的设计。但是，这种设计方式又和我们常用的设计方法不一致，有一定的学习和实践成本。

基于Event Sourcing的分布式系统

如果要开发一个基于Event Sourcing模式的分布式系统，最简单的方式就是用2个服务分别提供读和写的功能。写服务接收Command请求，触发聚合对象上的处理函数，更新聚合数据。然后把这个事件发送到一个MQ队列上。读服务监听这个队列，获取事件，更新相应的物化视图的数据。同时所有的Query请求都由读服务处理并返回。

对于写服务，它的数据只有事件，是一个流式的写操作，还有基于索引的查询。对于读服务，我们又可以部署多个应用，进一步提供数据查询的性能。可以看到，通过这么一个简单的读写分离，我们就能大大提高系统的性能。

什么时候使用Event Sourcing

使用Event Sourcing有它的优点也有缺点，那么什么时候该使用Event Sourcing模式呢？

1. 首先是系统类型，如果你的系统有大量的CRUD，也就是增删改查类型的业务，那么就不适合使用Event Sourcing模式。Event Sourcing模式比较适用于有复杂业务的应用系统。
2. 如果你或你的团队里面有DDD（领域驱动设计）相关的人员，那么你应该优先考虑使用Event Sourcing。
3. 如果对你的系统来说，业务数据产生的过程比结果更重要，或者说更有意义，那就应该使用Event Sourcing。你可以使用Event Sourcing的事件数据来分析数据产生的过程，解决bug，也可以用来分析用户的行为。
4. 如果你需要系统提供业务状态的历史版本，例如一个内容管理系统，如果我想针对内容实现版本管理，版本回退等操作，那就应该使用Event Sourcing。

大家可以关注一下课程：《分布式事务实践 解决数据一致性》

作者：大漠风

链接：imooc.com/article/40858

来源：慕课网

编辑于 2018-07-11

供应链管理 分布式系统 数据库