## 事件溯源模式

### 事件溯源模式

事件溯源（Event Sourcing）模式是针对事件范式提供的设计模式，通过事件风暴识别到的领域事件与聚合将成为领域设计模型的核心要素。事件溯源模式与传统领域驱动设计模式的最大区别在于 **对聚合生命周期的管理** 。资源库在管理聚合生命周期时，会直接针对聚合内的实体与值对象执行持久化操作，而事件溯源则将聚合以一系列事件的方式进行持久化。因为领域事件记录的就是聚合状态的变化，如果能够将每次状态变化产生的领域事件记录下来，就相当于记录了聚合生命周期每一步“成长的脚印”。此时，持久化的事件就成为了一个自由的“时空穿梭机”，随时可以根据需求通过重放（Replaying）回到任意时刻的聚合对象。

《微服务架构设计模式》总结了事件溯源的优点和缺点：

事件溯源有几个重要的好处。例如，它保留了聚合的历史记录，这对于实现审计和监管的功能非常有帮助。它可靠地发布领域事件，这在微服务架构中特别有用。事件溯源也有弊端。它有一定的学习曲线，因为这是一种完全不同的业务逻辑开发方式。此外，查询事件存储库通常很困难，这需要你使用 CQRS 模式。

事件溯源模式的首要原则是“事件永远是不变的”，因此对事件的持久化就变得非常简单，无论发生了什么样的事件，在持久化时都是追加操作。这就好似在 GitHub 上提交代码，每次提交都会在提交日志上增加一条记录。因此，我们在理解事件溯源模式时，可以把握两个关键原则：

* 聚合的每次状态变化，都是一个事件的发生
* 事件是不变的，以追加方式记录事件，形成事件日志

由于事件溯源模式运用在限界上下文的边界之内，它所操作的事件属于领域设计模型的一部分。若要准确说明，应称呼其为“领域事件”，以区分于发布者—订阅者模式操作的“应用事件”。

#### 领域事件的定义

既然事件溯源以追加形式持久化领域事件，就可以不受聚合持久化实现机制的限制，例如对象与关系之间的阻抗不匹配，复杂的数据一致性问题，聚合历史记录存储等。 **事件溯源持久化的不是聚合，而是由聚合状态变化产生的领域事件，这种持久化方式称之为事件存储（Event Store）。** 事件存储会建立一张事件表，记录下事件的 ID、类型、关联聚合和事件的内容，以及产生事件时的时间戳。其中，事件内容将作为重建聚合的数据来源。由于事件表需要支持各种类型的领域事件，意味着事件内容需要存储不同结构的数据值，因此通常选择 JSON 格式的字符串。例如 IssueCreated 事件：

{  
 "eventId": "111",  
 "eventType": "IssueCreated",  
 "aggregateType": "Issue",  
 "aggregateId": "100",  
 "eventPayload": {  
 "issueId": "100",  
 "title": "Global Consent Management",  
 "description": "Manage global consent for customer",  
 "label": "STORY",  
 "iterationId": "111",  
 "points": 5  
 },  
 "createdTimestamp": "2019-08-30 12:10:11 756"   
}

只要保证 eventPayload 的内容为可解析的标准格式，IssueCreated 事件也可存储在关系数据库中，通过 eventType、aggregateType 和 aggregateId 可以确定事件以及该事件对应的聚合，重建聚合的数据则来自 eventPayload 的值。显然，我们需要结合具体的领域场景来定义领域事件。领域事件包含的值必须是订阅方需要了解的信息，例如 IssueCreated 事件会创建一张任务卡，如果事件没有提供该任务的 title、description 等值，就无法通过这些值重建 Issue 聚合对象。显然，事件溯源操作的领域事件主要是为了追溯状态变更，并可以根据存储的事件来重建聚合。这与发布者—订阅者模式引入事件的目的大相径庭。

#### 聚合的创建与更新

要实现事件溯源，需要执行的操作（或职责）包括：

* 处理命令
* 发布事件
* 存储事件
* 查询事件
* 创建以及重建聚合

虽然事件溯源采用了和传统领域驱动设计不同的建模范式和设计模式，但仍然需要遵守领域驱动设计的根本原则： **保证领域模型的纯粹性** 。如果结合事件风暴来理解事件溯源，相与协作的对象包括：领域事件、决策命令和聚合，同时，决策命令包含的信息则为读模型。由于事件溯源采用了事件存储模式，因此它与发布者—订阅者模式不同，实际上并不会真正发布事件到消息队列或者事件总线。事件溯源的所谓“发布事件”实则为创建并存储事件。

如果我们将决策命令、读模型、领域事件和聚合皆视为领域设计模型的一部分，为了保证领域模型的纯粹性，就必须将存储事件和查询事件的职责交给事件存储。与场景驱动设计相似，领域服务承担了协作这些领域模型对象实现领域场景的职责，并由它与抽象的 EventStore 协作。为了让领域服务知道该如何存储事件，聚合在处理了决策命令之后，需要将生成的领域事件返回给领域服务。聚合仅负责创建领域事件，领域服务通过调用 EventStore 存储领域事件。

初次创建聚合实例时，聚合还未产生任何一次状态的变更，不需要重建聚合。因此，聚合的创建操作与更新操作的流程并不相同，实现事件溯源时需区分对待。创建聚合需要执行如下活动：

* 创建一个新的聚合实例
* 聚合实例接收命令生成领域事件
* 运用生成的领域事件改变聚合状态
* 存储生成的领域事件

例如，要创建一张新的问题卡片。在领域层，首先由领域服务接收决策命令，由其统筹安排：

public class CreatingIssueService {  
 private EventStore eventStore;  
  
 public void execute(CreateIssue command) {  
 Issue issue = Issue.newInstance();  
 List<DomainEvent> events = issue.process(command);  
 eventStore.save(events);  
 }  
}

领域服务首先会调用聚合的工厂方法创建一个新的聚合，然后调用该聚合实例的 process(command) 方法处理创建 Issue 的决策命令。聚合的 process(command) 方法首先会验证命令有效性，然后根据命令执行领域逻辑，再生成新的领域事件。在返回领域事件之前，会调用 apply(event) 方法更改聚合的状态：

public class Issue extends AggregateRoot<Issue> {  
 public List<DomainEvent> process(CreateIssue command) {  
 try {  
 command.validate();  
 IssueCreated event = new IssueCreated(command.issueDetail());  
 apply(event);  
 return Collections.singletonList(event);  
 } catch (InvalidCommandException ex) {  
 logger.warn(ex.getMessage());  
 return Collections.emptyList();  
 }   
 }  
  
 public void apply(IssueCreated event) {  
 this.state = IssueState.CREATED;  
 }  
}

process(command) 方法并不负责修改聚合的状态，它将这一职责交给了单独定义的 apply(event) 方法，然后它会调用该方法。之所以要单独定义 apply(event) 方法，是为了聚合的重建。在重建聚合时，通过遍历该聚合发生的所有领域事件，再调用这一单独定义的 apply(event) 方法，完成对聚合实例的状态变更。如此的设计，就能够重用运用事件逻辑的逻辑，同时保证聚合状态变更的一致性，真实地体现了状态变更的历史。

IssueCreated 事件是不可变的，故而大体可以视 process(command) 方法是一个没有副作用的 **纯函数（pure function）** 。此为状态变迁的本质特征，即聚合从一个状态（事件）变迁到一个新的状态（事件），而非真正修改聚合本身的状态值。这也正是我认为事件范式与函数范式更为契合的原因所在。

聚合处理了命令并返回领域事件后，领域服务会通过它依赖的 EventStore 存储这些领域事件。事件的存储既可以认为是对外部资源的依赖，也可以认为是一种副作用。显然，将存储事件的职责转移给领域服务，既符合面向对象尽量将依赖向外推的设计原则，也符合函数编程将副作用往外推的设计原则。遵循这一原则设计的聚合，能很好地支持单元测试的编写。

更新聚合需要执行如下活动：

* 从事件存储加载聚合对应的事件
* 创建一个新的聚合实例
* 遍历加载的事件，完成对聚合的重建
* 聚合实例接收命令生成领域事件
* 运用生成的领域事件改变聚合状态
* 存储生成的领域事件

例如，要将刚才创建好的 Issue 分配给团队成员，就可以发送命令 AssignIssue 给领域服务：

public class AssigningIssueService {  
 private EventStore eventStore;  
  
 public void execute(AssignIssue command) {  
 Issue issue = Issue.newInstance();  
 List<DomainEvent> events = eventStore.findBy(command.aggregateId());   
 issue.applyEvents(events);  
 List<DomainEvent> events = issue.process(command); // 注意process方法内部会apply新的领域事件  
 eventStore.save(events);  
 }  
}

领域服务通过 EventStore 与命令传递过来的聚合 ID 获得该聚合的历史事件，然后针对新生的聚合进行生命状态的重建，这就相当于重新执行了一遍历史上曾经执行过的领域行为，使得当前聚合恢复到接受本次命令之前的正确状态，然后处理当前决策命令，生成事件并存储。

#### 快照

聚合的生命周期各有长短。例如 Issue 的生命周期就相对简短，一旦该问题被标记为完成，几乎就可以认为具有该身份标识的 Issue 已经寿终正寝。除了极少数的 Issue 需要被 ReOpen 之外，该聚合不会再发布新的领域事件了。有的聚合则不同，或许聚合变化的频率不高，但它的生命周期相当漫长，例如账户 Account，就可能随着时间的推移，积累大量的领域事件。当一个聚合的历史领域事件变得越来越多时，如前所述的加载事件以及重建聚合的执行效率就会越来越低。

在事件溯源中，通常通过“快照”形式来解决此问题。

使用快照时，通常会定期将聚合以 JSON 格式持久化到聚合快照表（Snapshots）中。注意，快照表持久化的是当前时间戳的聚合数据，而非事件数据。故而快照表记录了聚合类型、聚合 ID 和聚合的内容，当然也包括持久化快照时的时间戳。创建聚合时，可直接根据聚合 ID 从快照表中获取聚合的内容，然后利用反序列化直接创建聚合实例，如此即可让聚合实例直接从某个时间戳“带着记忆重生”，省去了从初生到快照时间戳的重建过程。由于快照内容并不一定是最新的聚合值，因而还需要运用快照时间戳之后的领域事件，才能快速而正确地恢复到当前状态：

public class AssigningIssueService {  
 private EventStore eventStore;  
 private SnapshotRepository snapshotRepo;  
  
 public void execute(AssignIssue command) {  
 // 利用快照重建聚合  
 Snapshot snapshot = snapshotRepo.snapshotOf(command.aggregateId());  
 Issue issue = snapshot.rebuildTo(Issue.getClass());  
  
 // 获得快照时间戳之后的领域事件  
 List<DomainEvent> events = eventStore.findBy(command.aggregateId(), snapshot.createdTimestamp());  
 issue.applyEvents(events);  
  
 List<DomainEvent> events = issue.process(command);  
 eventStore.save(events);  
 }  
}

#### 面向聚合的事件溯源

事件溯源其实有两个不同的视角。一个视角面向事件，另一个视角可以面向聚合。前述代码中，无论是获取事件、存储事件或者运用事件，其目的还是为了操作聚合。例如，获取事件是为了实例化或者重建一个聚合实例；存储事件虽然是针对事件的持久化，但最终目的却是为了将来对聚合的重建，因此也可同等视为聚合的持久化；至于运用事件，就是为了正确地变更聚合的状态，相当于更新聚合。因此，在领域层，我们可以通过聚合资源库来封装事件溯源与事件存储的底层机制。如此，既可以简化领域服务的逻辑，又可以帮助代码的阅读者更加直观地理解领域逻辑。仍以 Issue 为例，可定义 IssueRepository 类：

public class IssueRepository {  
 private EventStore eventStore;  
 private SnapshotRepository snapshotRepo;  
  
 // 查询聚合  
 public Issue issueOf(IssueId issueId) {  
 Snapshot snapshot = snapshotRepo.snapshotOf(issueId);  
 Issue issue = snapshot.rebuildTo(Issue.getClass());  
  
 List<DomainEvent> events = eventStore.findBy(command.aggregateId(), snapshot.createdTimestamp());  
 issue.applyEvents(events);  
  
 return issue;  
 }  
  
 // 新建聚合  
 public void add(CreateIssue command) {  
 Issue issue = Issue.newInstance();  
 processCommandThenSave(issue, command);  
 }   
  
 // 更新聚合  
 public void update(AssignIssue command) {  
 Issue issue = issueOf(command.issueId());  
 processCommandThenSave(issue, command);   
 }  
  
 private void processCommandThenSave(Issue issue, DecisionCommand command) {  
 List<DomainEvent> events = issue.process(command);  
 eventStore.save(events);  
 }   
}

定义了这样一个面向聚合的资源库后，事件溯源的细节就被隔离在资源库内，领域服务操作聚合就像对象范式的实现一样，不同之处在于领域服务接收的仍然是决策命令。这时的领域服务就从承担领域行为的职责蜕变为对决策命令的分发，由于它封装的领域逻辑非常简单，因此可以为一个聚合定义一个领域服务：

public class IssueService {  
 private IssueRepository issueRepo;  
  
 public void execute(CreateIssue command) {  
 issueRepo.add(command);  
 }  
  
 public void execute(AssignIssue command) {  
 issueRepo.update(command);  
 }  
}

领域服务的职责变成了对命令的分发。在事件范式中，我们甚至可以将领域服务命名为 IssueCommandDispatcher，使其命名更加名副其实。

#### 聚合查询的改进

通过 IssueRepository 的实现，可以看出事件溯源在聚合查询功能上存在的限制：它仅仅支持基于主键的查询，这是由事件存储机制决定的。要提高对聚合的查询能力，唯一有效的解决方案就是在存储事件的同时存储聚合。

单独存储的聚合与领域事件无关，它是根据领域模型对象的数据结构进行设计和管理的，可以满足复杂的聚合查询请求。显然，存储的事件由于真实地反应了聚合状态的变迁，故而用于满足客户端的命令请求；存储的聚合则依照对象范式的聚合对象进行设计，通过 ORM 框架就能满足对聚合的高级查询请求。事件与聚合的分离，意味着命令与查询的分离，实际上就是 CQRS（Command Query Responsibility Segregation，命令查询职责分离）模式的设计初衷。

CQRS 在架构设计上有许多变化，我会在课程的第五部分《融合：战略设计与战术设计》进行深入讲解。这里仅提供事件溯源模式下如何解决查询聚合局限性的方案：在事件溯源的基础上，分别实现事件的存储与聚合的存储，前者用于体现聚合的历史状态，后者用于体现聚合的当前状态。

整个系统架构的领域模型被分为命令与查询两部分。命令请求下的领域模型采用事件溯源模式，聚合负责命令的处理与事件的创建；查询请求下的领域模型采用查询视图模式，直接查询业务数据库，获得它所存储的聚合信息。问题在于：命令端如何做到 **及时可靠一致** 地将事件的最新状态反映给查询端的业务数据库？

根据设计者对事件存储和聚合存储的态度，存在两种迥然不同的解决方案：本地式和分布式。

在领域驱动战略设计的指导下，一个聚合产生的所有领域事件和聚合应处于同一个限界上下文。因此，可以选择将事件存储与聚合存储放在同一个数据库，如此即可保证事件存储与聚合存储的事务强一致性。存储事件时，同时将更新后的聚合持久化。既然数据库已经存储了聚合的最新状态，就无需通过事件存储来重建聚合，但领域逻辑的处理模式仍然体现为命令—事件的状态迁移形式。至于查询，就与事件无关了，可以直接查询聚合所在的数据库。如此，可修改资源库的实现，如 IssueRepository：

public class IssueRepository {  
 private EventStore eventStore;  
 private AggregateRepository<Issue> repo;  
  
 public Issue issueOf(IssueId issueId) {  
 return repo.findBy(issueId);  
 }  
  
 public List<Issue> allIssues() {  
 return repo.findAll();  
 }  
  
 public void add(CreateIssue command) {  
 Issue issue = Issue.newInstance();  
 processCommandThenSave(issue, command);  
 }   
  
 public void update(AssignIssue command) {  
 Issue issue = issueOf(command.issueId());  
 processCommandThenSave(issue, command);   
 }  
  
 private void processCommandThenSave(Issue issue, DecisionCommand command) {  
 List<DomainEvent> events = issue.process(command);  
 eventStore.save(events);  
 repo.save(issue);  
 }   
}

这一方案的优势在于事件存储和聚合存储都在本地数据库，通过本地事务即可保证数据存储的一致性，且在支持事件追溯与审计的同时，还能避免重建聚合带来的性能影响。与不变的事件不同，聚合会被更新，因此它的持久化要比事件存储更加复杂，既然在本地已经存储了聚合对象，引入事件溯源的价值就没有这么明显了。由于事件与聚合存储在一个强一致的事务范围内，事件的异步非阻塞特性也未曾凸显出来。

如果事件存储和聚合存储不在同一个数据库中，就需要将事件的最新状态反映给存储在业务数据库中的聚合，方法是通过 **发布或轮询事件** 来搭建事件存储与聚合存储之间沟通的桥梁。此时的事件起到了通知状态变更的作用。

若采用事件发布的机制，由于事件模型与聚合模型之间属于跨进程的分布式通信，因此需要引入消息中间件作为事件的传输通道。这就相当于在事件溯源模式中引入了发布者—订阅者模式。要通知状态变更，可以直接将领域事件视为应用事件进行发布，也可以将领域事件转换为耦合度更低的应用事件。

事件存储端作为发布者，当聚合接收到决策命令请求后生成领域事件，然后将领域事件或转换后的应用事件发布到诸如 Kafka、RabbitMQ 之类的消息中间件。发布事件的同时还需存储领域事件，以支持事件的溯源。聚合存储端作为订阅者，会订阅它关心的事件，借由事件携带的数据创建或更新业务数据库中的聚合。由于事件消息的发布是异步的，处理命令请求和存储聚合数据的功能又分布在不同的进程，就能更快地响应客户端发送来的命令请求，提高整个系统的响应能力。

如果不需要实时发布事件，则可以定时轮询存储到事件表中事件，获取未曾发布的新事件发布到消息中间件。为了避免事件的重复发布，可以在事件表中增加一个 published 列，用于判断该事件消息是否已经发布。一旦成功发布了该事件消息，就需要更新事件表中的 published 标记为 true。

无论是发布还是轮询事件，都需要考虑分布式事务的一致性问题，事务范围要协调的操作包括：

* 存储领域事件（针对发布事件）
* 发送事件消息
* 更新聚合状态
* 更新事件表标记（针对轮询事件）

虽然在一个事务范围内要协调的操作较多，但要保证数据的一致性也没有想象的那么棘手。首先，事件的存储与聚合的更新并不要求强一致性，尤其对于命令端而言，选择了这样一种模式，意味着你已经接受了执行命令请求时的异步非实时能力。如果选择实时发布事件，为了避免存储领域事件与发送事件消息之间的不一致性，我们可以考虑在事件存储成功之后，再发送事件消息。由于领域事件是不变的，存储事件皆以追加方式进行，故而无需对数据行加锁来控制并发，这使得领域事件的存储操作相对高效。

许多消息中间件都可以保证消息投递做到“至少一次（at least once）”，那么在事件的订阅方，只要保证更新聚合状态操作的幂等性，就能避免重复消费事件消息，变相地做到了“恰好一次（exactly once）”。更新聚合状态的操作包括创建、更新和删除，除了创建操作，其余操作本身就是幂等的。

由于创建聚合的事件消息中包含了聚合的 ID，因此在创建聚合时，只需要判断业务数据库是否已经存在该聚合 ID，若已存在，则证明该事件消息已被消费过，此时应忽略该事件消息，避免重复创建。当然，我们也可以在事件订阅方引入事件发送历史表。由于该历史表可以和聚合所在的业务数据表放在同一个数据库，可保证二者的事务强一致性，也能避免事件消息的重复消费。

针对轮询事件，由于消息中间件保证了事件消息的成功投递，就无需等待事件消息发送的结果，立即更新事件表标记。即使更新标记的操作有可能出现错误，只要能保证事件的订阅者遵循了幂等性，避免了事件消息的重复消费，就可以降低一致性要求。即使事件表标记的更新未曾满足一致性，也不会产生负面影响。

要保证数据的最终一致性，剩下的工作就是如何保证聚合状态的成功更新。在确保了事件消息已成功投递之后，对聚合状态更新的操作已经由分布式事务的协调“降低”为对本地数据库的访问操作。许多消息中间件都可以缓存甚至持久化队列中的事件，在设置了合理的保存时间后，倘若事件的订阅者处理失败，还可通过重试机制来提高更新操作的成功率。

即使如此，要保证 100% 的操作都满足了事务的最终一致性，仍然很难。倘若发布的事件不止一个消费者订阅，事务的一致性问题会变得更加复杂。若业务场景对一致性的要求极高，要么就只能采用本地式的方案，要么就考虑引入诸如 TCC 模式、可靠消息传递以及 Saga 模式等分布式事务模式来实现最终一致性。在当今的软件系统架构中，分布式事务是一个永恒且艰难的话题，若需要深入了解分布式事务，建议阅读与此主题相关的技术文档或技术书籍。

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持