命令查询职责分离（CQRS）浅析

2017302580098

姚晓璐

摘要：命令查询职责分离模式（Command Query Responsibility Segregation，CQRS）从业务上分离修改和查询的行为，从而使得逻辑更加清晰，便于对不同部分进行针对性的优化。本文通过其产生原因、基本结构、适用场景等角度来简单介绍 CQRS。

关键字：CQRS 模式 浅析

1 绪论

在以前的管理系统中，命令（Command，通常用来更新数据，操作 DB）和查询（Query）通常使用的是在数据访问层中 Repository 中的实体对象（这些对象是对 DB 中表的映射），这些实体有可能是 SQLServer 中的一行数据或者多个表。

通常对 DB 执行的增、删、改、查（CRUD）都是针对的系统的实体对象。如通过数据访问层获取数据，然后通过数据传输对象 DTO 传给表现层。或者，用户需要更新数据，通过 DTO 对象将数据传给 Model，然后通过数据访问层写回数据库，系统中的所有交互都是和数据查询和存储有关，可以认为是数据驱动（Data-Driven）的。对于一些比较简单的系统，使用这种 CRUD 的设计方式能够满足要求。特别是通过一些代码生成工具及 ORM 等能够非常方便快速的实现功能。

但是传统的 CRUD 方法有一些问题：

使用同一个对象实体来进行数据库读写可能会太粗糙，大多数情况下，比如编辑的时候可能只需要更新个别字段，但是却需要将整个对象都穿进去，有些字段其实是不需要更新的。在查询的时候在表现层可能只需要个别字段，但是需要查询和返回整个实体对象。

使用同一实体对象对同一数据进行读写操作的时候，可能会遇到资源竞争的情况，经常要处理的锁的问题，在写入数据的时候，需要加锁。读取数据的时候需要判断是否允许脏读。这样使得系统的逻辑性和复杂性增加，并且会对系统吞吐量的增长会产生影响。

同步的，直接与数据库进行交互在大数据量同时访问的情况下可能会影响性能和响应性，并且可能会产生性能瓶颈。

由于同一实体对象都会在读写操作中用到，所以对于安全和权限的管理会变得比较复杂。

这里面很重要的一个问题是，系统中的读写频率比，是偏向读，还是偏向写，就如同一般的数据结构在查找和修改上时间复杂度不一样，在设计系统的结构时也需要考虑这样的问题。解决方法就是我们经常用到的对数据库进行读写分离。让主数据库处理事务性的增（Insert）、删（Delete）、改（Update）操作，让从数据库处理查询（Select）操作，数据库复制被用来将事务性操作导致的变更同步到集群中的从数据库。这只是从 DB 角度处理了读写分离，但是从业务或者系统上面读和写仍然是存放在一起的。他们都是用的同一个实体对象。

要从业务上将读和写分离，就需要使用命令查询职责分离模式。

2 CQRS 简介

其基本思想在于，任何一个对象的方法可以分为两大类：命令（Command）：不返回任何结果（void），但会改变对象的状态；查询（Query）：返回结果，但是不会改变对象的状态，对系统没有副作用。

CQRS 使用分离的接口将数据查询操作（Queries）和数据修改操作（Commands）分离开来，这也意味着在查询和更新过程中使用的数据模型也是不一样的。这样读和写逻辑就隔离开来了。

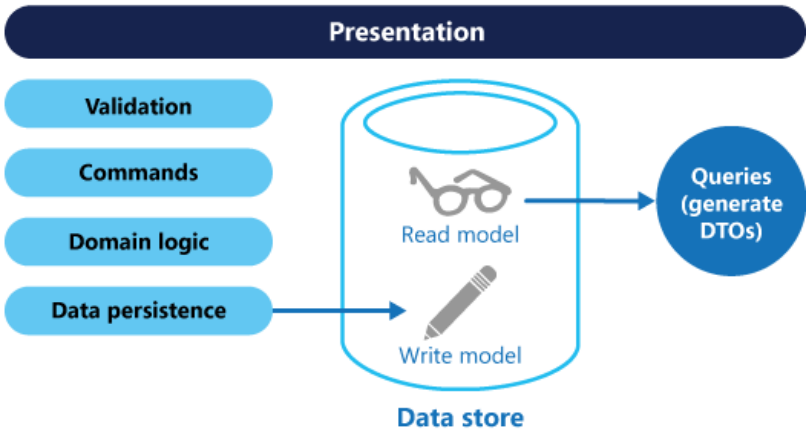

图 2.1 基本的 CQRS 模型

使用 CQRS 分离了读写职责之后，可以对数据进行读写分离操作来改进性能，可扩展性和安全。如图 2.2 所示。

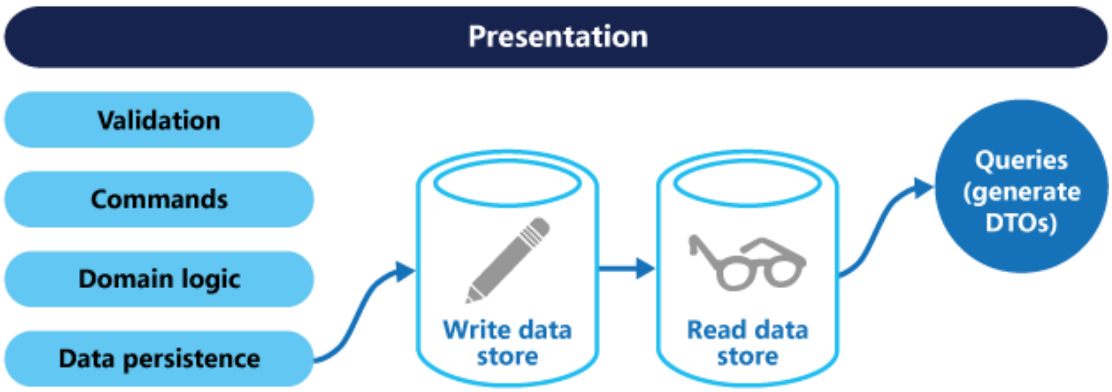

图 2.2 读写分离的 CQRS 模型

主数据库处理 CUD，从库处理 R，从库的的结构可以和主库的结构完全一样，也可以不一样，从库主要用来进行只读的查询操作。在数量上从库的个数也可以根据查询的规模进行扩展，在业务逻辑上，也可以根据专题从主库中划分出不同的从库。从库也可以实现成 ReportingDatabase，根据查询的业务需求，从主库中抽取一些必要的数据生成一系列查询报表来存储。

3 CQRS 适用场景

当在业务逻辑层有很多操作需要相同的实体或者对象进行操作的时候。CQRS 使得我们可以对读和写定义不同的实体和方法，从而可以减少或者避免对某一方面的更改造成冲突

对于一些基于任务的用户交互系统，通常这类系统会引导用户通过一系列复杂的步骤和操作，通常会需要一些复杂的领域模型，并且整个团队已经熟悉领域驱动设计技术。写模型有很多和业务逻辑相关的命令操作的堆，输入验证，业务逻辑验证来保证数据的一致性。读模型没有业务逻辑以及验证堆，仅仅是返回 DTO 对象为视图模型提供数据。读模型最终和写模型相一致。

适用于一些需要对查询性能和写入性能分开进行优化的系统，尤其是读/写比非常高的系统，横向扩展是必须的。比如，在很多系统中读操作的请求时远大于写操作。为适应这种场景，可以考虑将写模型抽离出来单独扩展，而将写模型运行在一个或者少数几个实例上。少量的写模型实例能够减少合并冲突发生的情况

适用于一些团队中，一些有经验的开发者可以关注复杂的领域模型，这些用到写操作，而另一些经验较少的开发者可以关注用户界面上的读模型。

对于系统在将来会随着时间不段演化，有可能会包含不同版本的模型，或者业务规则经常变化的系统

需要和其他系统整合，特别是需要和事件溯源 Event Sourcing 进行整合的系统，这样子系统的临时异常不会影响整个系统的其他部分。

但是在以下场景中，可能不适宜使用 CQRS：

领域模型或者业务逻辑比较简单，这种情况下使用 CQRS 会把系统搞复杂。

对于简单的，CRUD 模式的用户界面以及与之相关的数据访问操作已经足够的话，没必要使用 CQRS，这些都是一个简单的对数据进行增删改查。

不适合在整个系统中到处使用该模式。在整个数据管理场景中的特定模块中 CQRS 可能比较有用。但是在有些地方使用 CQRS 会增加系统不必要的复杂性。

4 总结

CQRS 是一种思想很简单清晰的设计模式，它通过在业务上分离操作和查询来使得系统具有更好的可扩展性及性能，使得能够对系统的不同部分进行扩展和优化。在 CQRS 中，所有的涉及到对 DB 的操作都是通过发送 Command，然后特定的 Command 触发对应事件来完成操作，这个过程是异步的，并且所有涉及

到对系统的变更行为都包含在具体的事件中，结合 Eventing Source 模式，可以记录下所有的事件，而不是以往的某一点的数据信息，这些信息可以作为系统的操作日志，可以来对系统进行回退或者重放。

5 参考文献

① Introduction to CQRS，http://www.codeproject.com/Articles/555855/Introduction-to-CQRS.

② CQRS，http://martinfowler.com/bliki/CQRS.html.

③ CQRS Journey，http://msdn.microsoft.com/en-us/library/jj554200.aspx.

④ Command and Query Responsibility Segregation (CQRS) Pattern, http://msdn.microsoft.com/en-us/library/dn568103.aspx.

⑤ Event Sourcing Pattern, http://msdn.microsoft.com/en-us/library/dn589792.aspx.