这篇文章是[软件架构编年史](https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/" \t "_blank)([译](https://www.jianshu.com/p/b477b2cc6cfa" \t "_blank))的一部分，这部编年史由[一系列关于软件架构的文章](https://herbertograca.com/category/development/series/software-architecture/" \t "_blank)组成。在这一系列文章中，我将写下我对软件架构的学习和思考，以及我是如何运用这些知识的。如果你阅读了这个系列中之前的文章，本篇文章的的内容将更有意义。

如果我们的应用以数据为中心，比如，仅实现基本的 CRUD 操作而把业务流程(例如，哪些数据需要修改，应按什么顺序修改)留给用户；其优点是用户可以在无需改变应用的情况下改变业务流程。而另一方面，这意味着所有用户都需要了解所有使用应用可以执行的业务流程的全部细节，当我们的流程不那么简单并且需要许多人都去理解它们时，这是一个大问题。

以数据为中心的应用对业务流程一无所知，因此领域不能使用任何动词，除了修改原始数据以外不能做任何事。它变成了徒有其表的数据模型抽象。流程都在使用应用的用户脑袋里，甚至只能在他们屏幕周围贴着的便利贴上找到。

一个有效的能真正发挥作用的应用的目标应该是通过捕捉用户的意图将他们从“流程”的负担中解放，让应用可以处理行为，而不仅仅只是简单地存储数据。

CQRS 就是这样一些技术概念演化的结果，它们一起帮助应用更准确地反映领域，同时还要克服常见的技术限制。

# 命令查询分离

如 Martin Fowler 所述，“命令查询分离”这个术语由 Bertrand Meyer 在他的“[Object Oriented Software Construction](https://www.amazon.com/gp/product/0136291554?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0136291554" \t "_blank)” (1988)一书中提出。这本书据说是 OO 早期最有影响力的著作之一。

Meier 为这样一条原则辩护，**我们不应该使用既能修改数据也能返回数据的方法**。这样我们就有了两种类型的方法：

1. **查询**：返回数据但不修改数据，因此没有副作用；
2. **命令**：修改数据但不返回数据。

换句话说，访问不应该改变答案而做事不应该给出答案，这样也遵守了单一职责原则。

然而，有一些模式是这条规则的例外，Martin Fowler 又说，传统的队列和堆栈的实现在弹出一个元素时，即改变了队列/堆栈也返回了移除的元素。

# 命令模式

命令模式的主要思想就是让我们远离数据为中心的应用，向具备领域知识和应用流程知识的以流程为中心的应用迈进。

事实上，这意味着用户不需要按顺序分别执行“CreateUser”、“ActivateUser”、“SendUserCreatedEmail”三个操作，只需要简单地执行一个“RegisterUser”命令，就可以将上面三个操作作为一个封装好的业务流程执行。

一个更有意思的例子是使用表单来修改一个客户的数据。假设我们可以使用表单来修改客户的名字、地址和电话号码，以及设置他是否是优先客户。我们还假设客户只有支付了账单才可以成为优先客户。在一个 CRUD 应用中，我们在收到数据之后，可以检查客户是否支付了账单，还可以接受或是拒绝数据修改请求。然而，这却是两个不同的业务流程：即便是客户没有支付账单，他也能成功地修改名字、地址和电话号码。使用命令模式之后，我们就能在代码中清晰地区别它们，创建两个代表不同业务流程的命令：一个用来改变客户数据，而另一个用来升级用户的优先状态，两个流程都由同一个 UI 界面触发。

在修改数据时为我们提供正确的粒度和意图。这就是命令的全部。—— Udi Dahan 2009, [Clarified CQRS](http://udidahan.com/2009/12/09/clarified-cqrs/" \t "_blank)

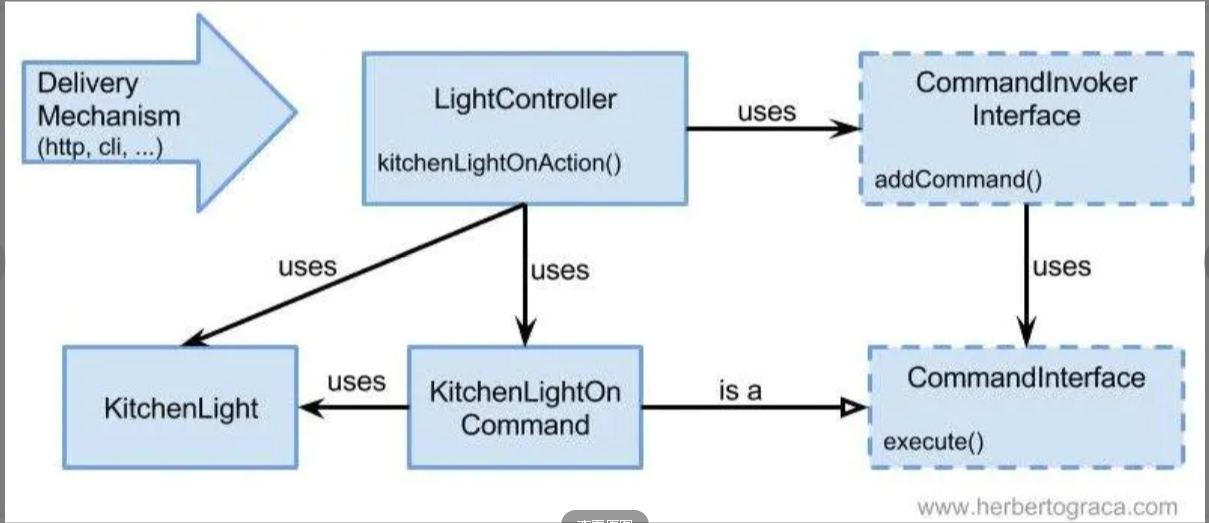
可是，还是有一点要记得，并不是说不能有“CreateUser”这样的简单命令。CRUD 的用例可以和带着意图的代表着复杂业务流程的用例完美共存，重要的是别误用。

技术上来说，如[Head First Design Patterns](https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124" \t "_blank) 所述，命令模式会将执行一个动作或者一系列动作所需的所有信息都封装起来。当我们需要在同一个地方以同样的方式执行一些不同的业务流程(命令)时这特别有用，因此它们需要同样的接口。例如，所有命令都有同样的execute()方法，这样在某个时刻，任何命令都可以被触发，不管到底是哪个命令。这也能让任何业务流程(命令)可以被放到队列中在合适的时候执行，同步或异步都行。

[Head First Design Patterns](https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124" \t "_blank) 一书给出的例子是屋子里的灯的遥控器。接下来我也会使用同样的例子，尽管我会指出它的不足之处。

那么，假设我们有一个控制屋子里的灯的遥控器，上面有一个按钮可以打开厨房里的灯，还有一个按钮关掉它们。每个按钮都代表着一个我们可以发给房屋灯光系统的命令。

下图是这个系统一种可能的设计：



这个一个朴素的设计，当然，它甚至不用考虑 DIC 我也完全用不到 UML。但我希望它能表达我的意思，所以我们来看看上面这幅图：作为对来自传达机制的输入的反映，LightController会使用参数为CommandInvoker的构造方法实例化并触发一个特定的控制器动作kitchenLightOnAction。这个动作将实例化正确的灯KitchenLight，还会实例化正确的命令KitchenLightOnCommand，把灯对象作为构造方法的参数传递它。然后命令会被交给CommandInvoker在某个时刻执行。要关灯的话，我们得创建另外的动作和命令，但设计基本是一样的。

这样我们就有了一个开灯的命令和一个关灯的命令。如果我们要将它们的功率设置为 50% 呢？我们再创建一条命令！如果我们要将它们的功率设置为 25% 和 75 % 呢？我们创建更多的命令！如果我们不用按钮而是用调光器将灯花设置成任意值呢？我们没办法创建无限多的命令！！！

这时的**实现问题**是：虽然命令中的逻辑一样，但是数据(功率的百分比)每次都不一样。所以我们应该创建一个命令，它的逻辑不变，仅仅是执行时的数据不同，但我们就会面临一个问题，接口execute()方法不接受参数。如果让它接受参数，那么将破坏整个命令的最重要的技术思路(将执行业务流程所需的所有信息都封装起来，而不用知道将要执行的到底是哪一个流程)。

当然，我们可以将数据传递给命令的构造方法来绕过这个问题，但并不优雅。实际上这是一个非正常的手段，因为数据不是对象之所以存在的必要信息，数据是它执行某段逻辑是需要的信息。因此，这些数据是方法的依赖而非对象的依赖。

我们还可以使用原生的语言结构[译：？？]来绕过这个问题，但还是不够优雅。

# 命令总线

要解决命令模式的这个限制，我们能做的就是应用最古老的 OO 原则：**将变化的部分和不变的部分分开**。

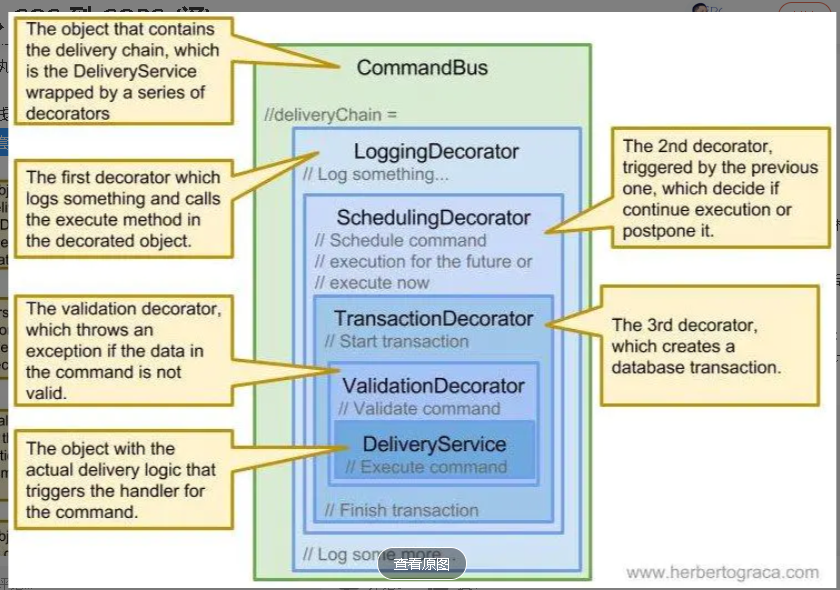
这个例子中变化的数据不变的是命令中执行的逻辑，所以我们可以将它们分成两个类。一个是用来存放数据的简单 DTO(我们称之为**命令**)，另一个存放要执行的逻辑(我们称之为**处理器**)，它拥有一个用来触发逻辑执行的方法execute(CommandInterface $command): void。CommandInvoker也将演化，它将可以接收命令并找出能够处理该命令的处理器。我们称之为**命令总线**。

用户界面的模式还可以进一步修改，许多命令不需要立即处理，它们可以放到队列中异步地执行。这种方式有一些优点能让系统更健壮：

* 响应可以更快地返回给用户，因为不用等着命令立即执行；
* 如果因为系统缺陷(如出现问题或者数据库下线) 导致命令失败，用户可能根本不会意识到。当问题解决后命令可以简单地进行重放。

在一个集中的地方处理需要执行(触发处理器)的逻辑，还会带来一个好处：我们可以在一个地方为所有处理器增加执行前后的逻辑。例如，我们可以在命令数据传给处理器之前进行校验，或者我们可以用数据库事务包装处理器的执行逻辑，或者让命令总线支持复杂的队列操作和异步的命令/处理器执行。

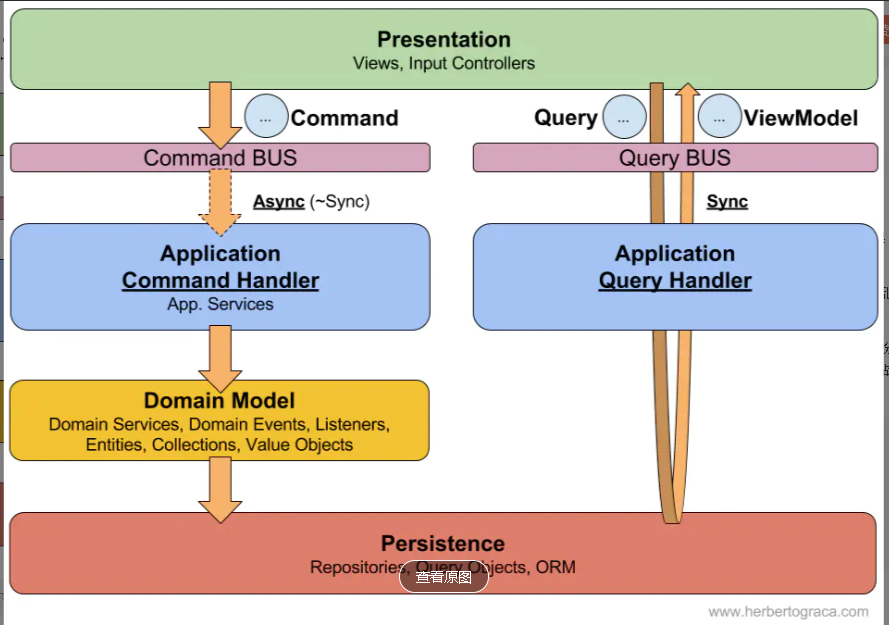
命令总线一般会使用包装着它的装饰器(或者已经包装了该装饰器的装饰器)来实现这个目标，类似俄罗斯套娃的结构.



这样我们可以创建自己的装饰器，可以配置命令总线(可能是第三方的)由哪些装饰器按照何种顺序组成，在命令总线中加入我们的定制功能。如果我们需要队列，我们就增加一个管理命令队列的装饰器。如果我们没有使用支持事务的数据库，我们就不需要用装饰器将处理器的执行器包装在数据库事务中。以此类推。

# 命令查询职责分离

将 CQS、命令和命令总线的概念组合在一起，我们最终得到了 CQRS。CQRS 可以用不同的方式实现，也可以不同程度地实现，也许只用了命令端，也许不会使用命令总线。为了保持完整性，下面的图代表了我所认为的全套 CQRS 实现：



## 查询端

依照 CQS，查询端只返回数据，完全不会修改它。由于我们不会尝试在这些数据上执行业务逻辑，我们不需要业务对象(如实体)，所以我们不需要 ORM 来填充实体，也不需要获取填充实体所需的全部数据。我们只需要查询原始数据展现给用户，并且只用查询展现给用户的模板所需的数据！

这立即就可以提升性能：查询数据时无需穿过业务逻辑层，我们直接查询刚好够用的数据。

这种拆分还可能带来的优化是数据存储完全会被拆分成两个独立的数据存储：一个专为写优化，另一个专为读优化。例如，如果我们使用关系型数据库管理系统：

* 读操作不需要任何数据完整性校验，也完全不需要外键约束，因为数据完整性的校验在写入数据存储是已经完成。所以**我们可以去掉读库的数据完整性约束**。
* 我们还能使用**刚好包含每个模板需要的数据的数据库视图**，让查询变得简单，变得更快(尽管我们要在模板变化时保持视图与之同步，而这会增加系统的复杂性) 。

这一点上，如果每个模板我们都有专门的数据库视图与之对应来简化查询，为什么我们还需要使用关系型数据库管理系统来做读取呢？！也许我们可以**使用文档存储来做读取**，比如 Mongo DB 甚至 Redis，它们要更快一些。也许可行，也许不行，我只是觉得如果应用在读取端出现性能问题的话这值得考虑。

查询本身可以使用返回一组供模板使用的数据的查询对象来实现，或者我们可以使用更成熟的方案，例如查询总线，它接收一个模板名字，使用一个查询对象查询数据并返回该模板需要的 ViewModel 实例。

这种方法可以解决 [Greg Young](https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf" \t "_blank) 提出的一些问题：

* 大量的存储库读方法常常还要包含分页和排序信息；
* 为了构造 DTO，Getter 暴露了领域对象的内部状态；
* \*在读取用例上使用预取路径，因为它们需要更多由 ORM 加载的数据；
* 构建 DTO 需要加载多个聚合根，导致对数据模型的非最优查询。另外，DTO 的构建操作还会导致聚合边界变得模糊；
* 不过，最大的问题是查询的优化极度困难：因为查询是针对对象模型的操作然后被转换成数据模型，比如 ORM，这些查询的优化可能非常困难。

## 命令端

如前所述，使用命令之后，我们将应用由以数据为中心的设计变成了围绕行为的设计，这和 DDD 完全一致。

将读取操作从处理命令的代码和领域中去掉之后，Greg Young 提出的问题也就不复存在：

* 领域对象突然不再需要暴露内部状态了；
* 除了GetById之外，资源库几乎没有任何查询方法；
* 聚合的边界将更聚焦于行为。

实体间“一对多”和“多对多”的关系会严重的影响 ORM 的性能。好消息是我们在处理命令时很少会需要这些关系，它们大多数时候只会在查询中用到，而我们已经把查询从命令的处理中移走了，所以我们可以移除这些实体关系。这里我所说的并不是关系型数据库管理系统中表之间的关系，这些外键约束依然应该存在于写库中，我指的是在 ORM 级别配置的实体间的连接。

我们真的需要在客户实体中保留订单集合吗？我们需要在哪条命令中浏览这个集合？实际上，到底有什么样的命令会需要一对多关系？如果一对多关系是这种情况，那么多对多关系绝对也是一样的。我的意思是，大多数命令都只包含一两个 ID。—— Udi Dahan 2009, [Clarified CQRS](http://udidahan.com/2009/12/09/clarified-cqrs/" \t "_blank)

按照和查询端的一样的思路，如果复杂查询用不上写入端，我们能用序列化实体的文档或键值存储来代替关系型数据库管理系统吗？也许可行，也许不行，我只是觉得如果应用在写入端出现性能问题的话这值得考虑。

### 业务处理事件

命令处理完之后，如果成功，处理器会触发一个事件将发生的事情通知到应用的其它部分。事件应该和按触发它的命令一样，只是应该以过去时态命名，这是它的命名规则。

# 总结

使用 CQRS 之后，我们就能够把读模型和写模型完全分开，让我们可以优化读操作和写操作。除了性能提升，它还让代码库更清晰简洁，更能体现出领域，更易维护。

同样，这全部都是封装、低耦合、高内聚和单一责任原则的体现。

然而，请记住，尽管 CQRS 提供了一种设计风格和一些技术解决方案，可以使应用非常健壮，但这并不意味着所有应用都应该以这种方式构建：我们应该在需要的时候使用我们需要的东西。

# 引用来源

(我认为最有价值的条目都**加粗**了。)

1994 – Gamma, Helm, Johnson, Vlissides – [Design Patterns: Elements of Reusable Object-Oriented Software](https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented-ebook/dp/B000SEIBB8" \t "_blank)  
1999 – Bala Paranj – [Java Tip 68: Learn how to implement the Command pattern in Java](http://www.javaworld.com/article/2077569/core-java/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html" \t "_blank)  
2004 – Eric Freeman, Elisabeth Robson – [Head First Design Patterns](https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124" \t "_blank)  
2005 – Martin Fowler – [Command Query Separation](https://martinfowler.com/bliki/CommandQuerySeparation.html" \t "_blank)  
**2009 – Udi Dahan –** [Clarified CQRS](http://udidahan.com/2009/12/09/clarified-cqrs/" \t "_blank)  
2010 – Greg Young – [CQRS, Task Based UIs, Event Sourcing agh!](http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/" \t "_blank)  
2010 – Greg Young – [CQRS Documents](https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf" \t "_blank)  
2010 – Udi Dahan – [Race Conditions Don’t Exist](http://udidahan.com/2010/08/31/race-conditions-dont-exist/" \t "_blank)  
2011 – Martin Fowler – [CQRS](https://martinfowler.com/bliki/CQRS.html" \t "_blank)  
2011 – Udi Dahan – [When to avoid CQRS](http://udidahan.com/2011/04/22/when-to-avoid-cqrs/" \t "_blank)  
**2014 – Greg Young –** [CQRS and Event Sourcing – Code on the Beach 2014](https://www.youtube.com/watch?v=JHGkaShoyNs" \t "_blank)  
2015 – Matthias Noback – [Responsibilities of the command bus](https://php-and-symfony.matthiasnoback.nl/2015/01/responsibilities-of-the-command-bus/" \t "_blank)  
2017 – Martin Fowler – [What do you mean by “Event-Driven”?](https://martinfowler.com/articles/201701-event-driven.html" \t "_blank)  
2017\* – Doug Gale – [Command Pattern](http://wiki.c2.com/?CommandPattern" \t "_blank)  
2017\* – Wikipedia – [Command Pattern](https://en.wikipedia.org/wiki/Command_pattern" \t "_blank)

作者：qinyu  
链接：https://www.jianshu.com/p/ab3843093903  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。