**[领域驱动设计(DDD)](https://www.cnblogs.com/Leo_wl/p/3866629.html)**

**阅读目录**

* [**一、领域和子域（Domain/Subdomain）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label0)
* [**二、限界上下文（Bounded Context）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label1)
* [**三、架构风格（Architecture）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label2)
* [**四、**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label3)
* [**五、行为饱满的领域对象**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label4)
* [**六、实体vs值对象（Entity vs Value Object）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label5)
* [**七、聚合（Aggregate）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label6)
* [**八、领域服务（Domain Service）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label7)
* [**九、资源库（Repository）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label8)
* [**十、领域事件（Domain Event）**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label9)

**阅读目录**

* [**领域驱动设计(DDD)实现之路**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_label0)

[**回到目录**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

[**领域驱动设计(DDD)实现之路**](http://www.cnblogs.com/CloudTeng/p/3865411.html)

2004年，当Eric Evans的那本**[《领域驱动设计——软件核心复杂性应对之道》](http://book.douban.com/subject/1629512/)**（后文简称《领域驱动设计》）出版时，我还在念高中，接触到领域驱动设计（DDD）已经是8年后的事情了。那时，我正打算在软件开发之路上更进一步，经同事介绍，我开始接触DDD。

我想，多数有经验的程序开发者都应该听说过DDD，并且尝试过将其应用在自己的项目中。不知你是否遇到过这样的场景：你创建了一个资源库（Repository），但一段时间之后发现这个资源库和传统的DAO越来越像了，你开始反思自己的实现方式是正确的吗？或者，你创建了一个聚合，然后发现这个聚合是如此的庞大，它为什么引用了如此多的对象，难道又是我做错了吗？

其实你并不孤单，我相信多数同仁都曾遇到过相似的问题。前不久，我一个同事给我展示了他在2007年买的那本已经被他韦编三绝过的《领域驱动设计》，他告诉我，读过好几遍后，他依然不知道如何将DDD付诸实践。Eric那本书固然是好，无可否认，但是我们程序员总希望看到一些实际的例子能够切实将DDD落地以指导我们的日常开发。

于是，在Eric的书出版将近10年之后，我们有了**[Vaughn Vernon](http://vaughnvernon.co/)**的**[《实现领域驱动设计》](http://book.douban.com/subject/25844633/)**，作为该书的译者，我有幸通读了本书，受益匪浅，得到的结论是：好的软件就应该是DDD的。



就像在微电子领域有知识产权核（Intellectual Property）一样，DDD将一个软件系统的核心业务功能集中在一个核心域里面，其中包含了实体、值对象、领域服务、资源库和聚合等概念。在此基础上，DDD提出了一套完整的支撑这样的核心领域的基础设施。此时，DDD已经不再是“面向对象进阶”那么简单了，而是演变成了一个系统工程。

所谓领域，即是一个组织的业务开展方式，业务价值便体现在其中。长久以来，我们程序员都是很好的技术型思考者，我们总是擅长从技术的角度来解决项目问题。但是，一个软件系统是否真正可用是通过它所提供的业务价值体现出来的。因此，与其每天钻在那些永远也学不完的技术中，何不将我们的关注点向软件系统所提供的业务价值方向思考思考，这也正是DDD所试图解决的问题。

在DDD中，代码就是设计本身，你不再需要那些繁文缛节的并且永远也无法得到实时更新的设计文档。编码者与领域专家再也不需要翻译才能理解对方所表达的意思。

DDD有战略设计和战术设计之分。战略设计主要从高层“俯视”我们的软件系统，帮助我们精准地划分领域以及处理各个领域之间的关系；而战术设计则从技术实现的层面教会我们如何具体地实施DDD。

**DDD之战略设计**

需要指出的是，DDD绝非一套单纯的技术工具集，但是我所看到的很多程序员却的确是这么认为的，并且也是怀揣着这样的想法来使用DDD的。过于拘泥于技术上的实现将导致DDD-Lite。简单来讲，DDD-Lite将导致劣质的领域对象，因为我们忽略了DDD战略建模所带来的好处。

DDD的战略设计主要包括领域/子域、通用语言、限界上下文和架构风格等概念。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**领域和子域（Domain/Subdomain）**

既然是领域驱动设计，那么我们主要的关注点理所当然应该放在如何设计领域模型上，以及对领域模型的划分。

领域并不是多么高深的概念，比如，一个保险公司的领域中包含了保险单、理赔和再保险等概念；一个电商网站的领域包含了产品名录、订单、发票、库存和物流的概念。这里，我主要讲讲对领域的划分，即将一个大的领域划分成若干个子域。

在日常开发中，我们通常会将一个大型的软件系统拆分成若干个子系统。这种划分有可能是基于架构方面的考虑，也有可能是基于基础设施的。但是在DDD中，我们对系统的划分是基于领域的，也即是基于业务的。

于是，问题也来了：首先，哪些概念应该建模在哪些子系统里面？我们可能会发现一个领域概念建模在子系统A中是可以的，而建模在子系统B中似乎也合乎情理。第二个问题是，各个子系统之间的应该如何集成？有人可能会说，这不简单得就像客户端调用服务端那么简单吗？问题在于，两个系统之间的集成涉及到基础设施和不同领域概念在两个系统之间的翻译，稍不注意，这些概念就会对我们精心创建好的领域模型造成污染。

如何解决？答案是：限界上下文和上下文映射图。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**限界上下文（Bounded Context）**

在一个领域/子域中，我们会创建一个概念上的领域边界，在这个边界中，任何领域对象都只表示特定于该边界内部的确切含义。这样边界便称为限界上下文。限界上下文和领域具有一对一的关系。

举个例子，同样是一本书，在出版阶段和出售阶段所表达的概念是不同的，出版阶段我们主要关注的是出版日期，字数，出版社和印刷厂等概念，而在出售阶段我们则主要关心价格，物流和发票等概念。我们应该怎么办呢，将所有这些概念放在单个Book对象中吗？这不是DDD的做法，DDD有限界上下文将这两个不同的概念区分开来。

从物理上讲，一个限界上下文最终可以是一个DLL(.NET)文件或者JAR(Java)文件，甚至可以是一个命名空间（比如Java的package）中的所有对象。但是，技术本身并不应该用来界分限界上下文。

将一个限界上下文中的所有概念，包括名词、动词和形容词全部集中在一起，我们便为该限界上下文创建了一套通用语言。通用语言是一个团队所有成员交流时所使用的语言，业务分析人员、编码人员和测试人员都应该直接通过通用语言进行交流。

对于上文中提到的各个子域之间的集成问题，其实也是限界上下文之间的集成问题。在集成时，我们主要关心的是领域模型和集成手段之间的关系。比如需要与一个REST资源集成，你需要提供基础设施（比如Spring 中的RestTemplate），但是这些设施并不是你核心领域模型的一部分，你应该怎么办呢？答案是防腐层，该层负责与外部服务提供方打交道，还负责将外部概念翻译成自己的核心领域能够理解的概念。当然，防腐层只是限界上下文之间众多集成方式的一种，另外还有共享内核、开放主机服务等，具体细节请参考《实现领域驱动设计》原书。限界上下文之间的集成关系也可以理解为是领域概念在不同上下文之间的映射关系，因此，限界上下文之间的集成也称为上下文映射图。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**架构风格（Architecture）**

DDD并不要求采用特定的架构风格，因为它是对架构中立的。你可以采用传统的三层式架构，也可以采用REST架构和事件驱动架构等。但是在《实现领域驱动设计》中，作者比较推崇事件驱动架构和六边形（Hexagonal）架构。

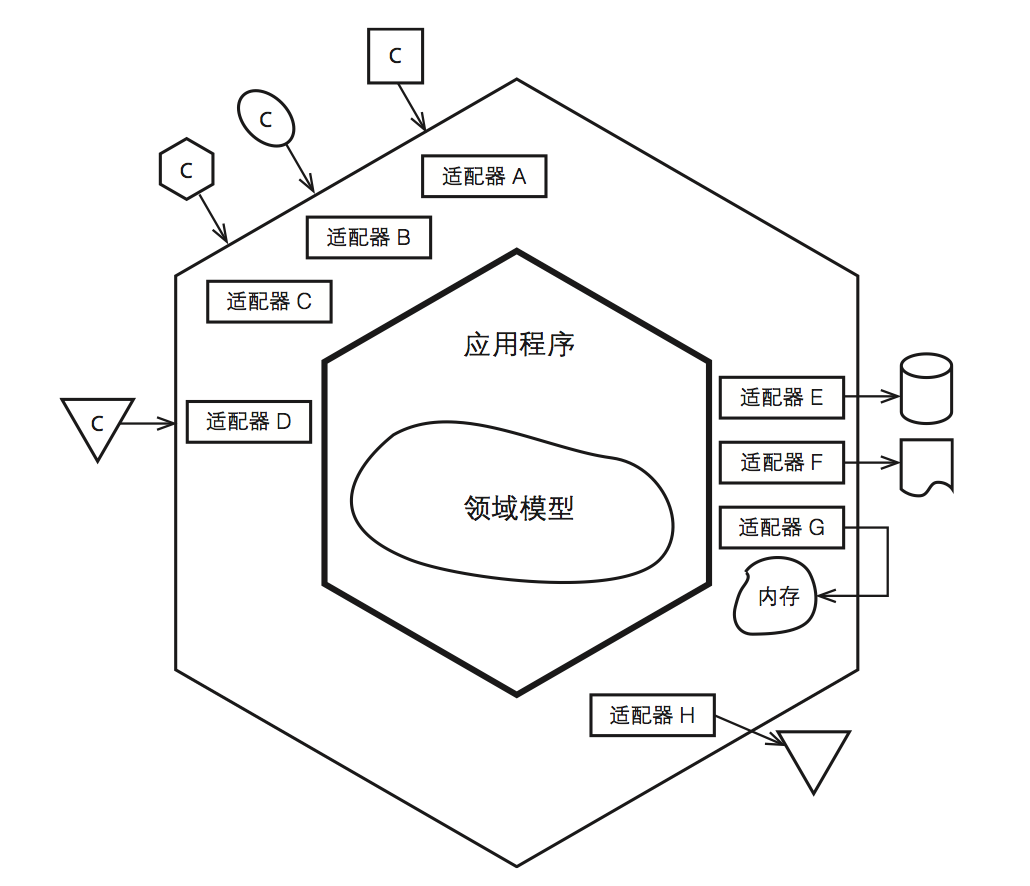
当下，面向接口编程和依赖注入原则已经在颠覆着传统的分层架构，如果再进一步，我们便得到了六边形架构，也称为端口和适配器（Ports and Adapters）。在六边形架构中，已经不存在分层的概念，所有组件都是平等的。这主要得益于软件抽象的好处，即各个组件的之间的交互完全通过接口完成，而不是具体的实现细节。正如Robert C. Martin所说：

**抽象不应该依赖于细节，细节应该依赖于抽象。**

采用六边形架构的系统中存在着很多端口和适配器的组合。端口表示的是一个软件系统的输入和输出，而适配器则是对每一个端口的访问方式。比如，在一个Web应用程序中，HTTP协议可以作为一个端口，它向用户提供HTML页面并且接受用户的表单提交；而Servlet（对于Java而言）或者Spring中的Controller则是相对应于HTTP协议的适配器。再比如，要对数据进行持久化，此时的数据库系统则可看成是一个端口，而访问数据库的Driver则是相应于数据库的适配器。如果要为系统增加新的访问方式，你只需要为该访问方式添加一个相应的端口和适配器即可。

那么，我们的领域模型又如何与端口和适配器进行交互呢？

上文已经提到，软件系统的真正价值在于提供业务功能，我们会将所有的业务功能分解为若干个业务用例，每一次业务用例都表示对软件系统的一次原子操作。所以首先，软件系统中应该存在这样的组件，他们的作用即以业务用例为单位向外界暴露该系统的业务功能。在DDD中，这样的组件称为应用层（Application Layer）。



在有了应用层之后，软件系统和外界的交互便变成了适配器和应用层之间的交互，如上图所示。

从图中可以看出，领域模型位于应用程序的核心部分，外界与领域模型的交互都通过应用层完成，应用层是领域模型的直接客户。然而，应用层中不应该包含有业务逻辑，否则就造成了领域逻辑的泄漏，而应该是很薄的一层，主要起到协调的作用，它所做的只是将业务操作代理给我们的领域模型。同时，如果我们的业务操作有事务需求，那么对于事务的管理应该放在应用层上，因为事务也是以业务用例为单位的。

应用层虽然很薄，但却非常重要，因为软件系统的领域逻辑都是通过它暴露出去的，此时的应用层扮演了系统门面（Facade）的角色。

**DDD之战术设计**

战略设计为我们提供一种高层视野来审视我们的软件系统，而战术设计则将战略设计进行具体化和细节化，它主要关注的是技术层面的实施，也是对我们程序员来得最实在的地方。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**行为饱满的领域对象**

我们希望领域对象能够准确地表达出业务意图，但是多数时候，我们所看到的却是充满getter和setter的领域对象，此时的领域对象已经不是领域对象了，而是Martin Fowler所称之为的**[贫血对象](http://martinfowler.com/bliki/AnemicDomainModel.html)**。

放到Java世界中，多年以来，Java Bean规范都引诱着程序员们以“自然而然又合乎情理”的方式创建着无数的贫血对象，而一些框架也规定对象必须提供getter和setter方法，比如Hibernate的早期版本。那么，贫血对象到底有什么坏处呢？来看一个例子：要修改一个客户（Customer）的邮箱地址，在使用setter方法时为：

**[复制代码](javascript:void(0);)**

复制代码

public class Customer {

private String email;

public void setEmail(String email) {

this.email = email;

}

}

复制代码

**[复制代码](javascript:void(0);)**

虽然以上代码可以完成“修改邮箱地址”的功能，但是当你读到这段代码时，你能否推测出系统中就一定存在着一个“修改邮箱地址”的业务用例呢？

你可能会说，可以在另一个Service类里面创建一个changeCustomerEmail()方法，再在该方法中调用Customer的setEmailAddress()方法，这样业务意图不就明了了吗？问题在于，修改邮箱地址这样的职责本来就应该放在Customer上，而不应该由Service和Customer共同完成。遵循诸如信息封装这样的基本面向对象原则是在实施DDD时最基本的素养。

要创建行为饱满的领域对象并不难，我们需要转变一下思维，将领域对象当做是服务的提供方，而不是数据容器，多思考一个领域对象能够提供哪些行为，而不是数据。

近几年又重新流行起来的函数式编程也能够帮助我们编写更加具有业务表达力的业务代码，比如C#和Java 8都提供了Lambda功能，同时还包括多数动态语言（比如Ruby和Groovy等）。再进一步，我们完全可以通过领域特定语言（DSL）的方式实现领域模型。

笔者曾经设想过这么一个软件系统：它的核心功能完全由一套DSL暴露给外界，所有业务操作都通过这套DSL进行，这个领域的业务规则可以通过一套规则引擎进行配置，于是这套DSL可以像上文提到的知识产权核一样拿到市面上进行销售。此时，我们的核心域被严严实实地封装在这套DSL之内，不容许外界的任何污染。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**实体vs值对象（Entity vs Value Object）**

在一个软件系统中，实体表示那些具有生命周期并且会在其生命周期中发生改变的东西；而值对象则表示起描述性作用的并且可以相互替换的概念。同一个概念，在一个软件系统中被建模成了实体，但是在另一个系统中则有可能是值对象。例如货币，在通常交易中，我们都将它建模成了一个值对象，因为我们花了20元买了一本书，我们只是关心货币的数量而已，而不是关心具体使用了哪一张20元的钞票，也即两张20元的钞票是可以互换的。但是，如果现在中国人民银行需要建立一个系统来管理所有发行的货币，并且希望对每一张货币进行跟踪，那么此时的货币便变成了一个实体，并且具有唯一标识（Identity）。在这个系统中，即便两张钞票都是20元，他们依然表示两个不同的实体。

具体到实现层面，值对象是没有唯一标识的，他的equals()方法（比如在Java语言中）可以用它所包含的描述性属性字段来实现。但是，对于实体而言，equals()方法便只能通过唯一标识来实现了，因为即便两个实体所拥有的状态是一样的，他们依然是不同的实体，就像两个人的名字都叫张三，但是他们却是两个不同的人的个体。

我们发现，多数领域概念都可以建模成值对象，而非实体。值对象就像软件系统中的过客一样，具有“创建后不管”的特征，因此，我们不需要像关心实体那样去关心诸如生命周期和持久化等问题。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**聚合（Aggregate）**

聚合可能是DDD中最难理解的概念 ，之所以称之为聚合，是因为聚合中所包含的对象之间具有密不可分的联系，他们是内聚在一起的。比如一辆汽车（Car）包含了引擎（Engine）、车轮（Wheel）和油箱（Tank）等组件，缺一不可。一个聚合中可以包含多个实体和值对象，因此聚合也被称为根实体。聚合是持久化的基本单位，它和资源库（请参考下文）具有一一对应的关系。

既然聚合可以容纳其他领域对象，那么聚合应该设计得多大呢？这也是设计聚合的难点之一。比如在一个博客（Blog）系统中，一个用户（User）可以创建多个Blog，而一个Blog又可以包含多篇博文（Post）。在建模时，我们通常的做法是在User对象中包含一个Blog的集合，然后在每个Blog中又包含了一个Post的集合。你真的需要这么做吗？如果你需要修改User的基本信息，在加载User时，所有的Blog和Post也需要加载，这将造成很大的性能损耗。诚然，我们可以通过延迟加载的方式解决问题，但是延迟加载只是技术上的实现方式而已。导致上述问题的深层原因其实在我们的设计上，我们发现，User更多的是和认证授权相关的概念，而与Blog关系并不大，因此完全没有必要在User中维护Blog的集合。在将User和Blog分离之后，Blog也和User一样成为了一个聚合，它拥有自己的资源库。问题又来了：既然User和Blog分离了，那么如果需要在Blog中引用User又该怎么办呢？在一个聚合中直接引用另外一个聚合并不是DDD所鼓励的，但是我们可以通过ID的方式引用另外的聚合，比如在Blog中可以维护一个userId的实例变量。

User作为Blog的创建者，可以成为Blog的工厂。放到DDD中，创建Blog的功能也只能由User完成。

综上，对于“创建Blog”的用例，我们可以通过以下方法完成：

**[复制代码](javascript:void(0);)**

复制代码

public class BlogApplicatioinService {

@Transactional

public void createBlog(String blogName, String userId) {

User user = userRepository.userById(userId);

Blog blog = user.createBlog(blogName);

blogRepository.save(blog);

}

}

复制代码

**[复制代码](javascript:void(0);)**

在上例中，业务用例通过BlogApplicationService应用服务完成，在用例方法createBlog()中，首先通过User的资源库得到一个User，然后调用User中的工厂方法createBlog()方法创建一个Blog，最后通过BlogRepository对Blog进行持久化。整个过程构成了一次事务，因此createBlog()方法标记有@Transactional作为事务边界。

使用聚合的首要原则为在一次事务中，最多只能更改一个聚合的状态。如果一次业务操作涉及到了对多个聚合状态的更改，那么应该采用发布领域事件（参考下文）的方式通知相应的聚合。此时的数据一致性便从事务一致性变成了最终一致性（Eventual Consistency）。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**领域服务（Domain Service）**

你是否遇到过这样的问题：想建模一个领域概念，把它放在实体上不合适，把它放在值对象上也不合适，然后你冥思苦想着自己的建模方式是不是出了问题。恭喜你，祝贺你，你的建模手法完全没有问题，只是你还没有接触到领域服务（Domain Service）这个概念，因为领域服务本来就是来处理这种场景的。比如，要对密码进行加密，我们便可以创建一个PasswordEncryptService来专门负责此事。

值得一提的是，领域服务和上文中提到的应用服务是不同的，领域服务是领域模型的一部分，而应用服务不是。应用服务是领域服务的客户，它将领域模型变成对外界可用的软件系统。

领域服务不能滥用，因为如果我们将太多的领域逻辑放在领域服务上，实体和值对象上将变成贫血对象。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**资源库（Repository）**

资源库用于保存和获取聚合对象，在这一点上，资源库与DAO多少有些相似之处。但是，资源库和DAO是存在显著区别的。DAO只是对数据库的一层很薄的封装，而资源库则更加具有领域特征。另外，所有的实体都可以有相应的DAO，但并不是所有的实体都有资源库，只有聚合才有相应的资源库。

资源库分为两种，一种是基于集合的，一种是基于持久化的。顾名思义，基于集合的资源库具有编程语言中集合的特征。举个例子，Java中的List，我们从一个List中取出一个元素，在对该元素进行修改之后，我们并不用显式地将该元素重新保存到List里面。因此，面向集合的资源库并不存在save()方法。比如，对于上文中的User，其资源库可以设计为：

**[复制代码](javascript:void(0);)**

public interface CollectionOrientedUserRepository {

public void add(User user);

public User userById(String userId);

public List allUsers(); public void remove(User user);

}

**[复制代码](javascript:void(0);)**

对于面向持久化的资源库来说，在对聚合进行修改之后，我们需要显式地调用sava()方法将其更新到资源库中。依然是User，此时的资源库如下：

**[复制代码](javascript:void(0);)**

public interface PersistenceOrientedUserRepository {

public void save(User user);

public User userById(String userId);

public List<User> allUsers();

public void remove(User user);

}

**[复制代码](javascript:void(0);)**

在以上两种方式所实现的资源库中，虽然只是将add()方法改成了save()方法，但是在使用的时候却是不一样的。在使用面向集合资源库时，add()方法只是用来将新的聚合加入资源库；而在面向持久化的资源库中，save()方法不仅用于添加新的聚合，还用于显式地更新既有聚合。

[**返回顶部**](https://www.cnblogs.com/Leo_wl/p/3866629.html#_labelTop)

**领域事件（Domain Event）**

在Eric的《领域驱动设计》中并没有提到领域事件，领域事件是最近几年才加入DDD生态系统的。

在传统的软件系统中，对数据一致性的处理都是通过事务完成的，其中包括本地事务和全局事务。但是，DDD的一个重要原则便是一次事务只能更新一个聚合实例。然而，的确存在需要修改多个聚合的业务用例，那么此时我们应该怎么办呢？

另外，在最近流行起来的微服务（Micro Service）的架构中，整个系统被分成了很多个轻量的程序模块，他们之间的数据一致性并不容易通过事务一致性完成，此时我们又该怎么办呢？

在DDD中，领域事件便可以用于处理上述问题，此时最终一致性取代了事务一致性，通过领域事件的方式达到各个组件之间的数据一致性。

领域事件的命名遵循英语中的“名词+动词过去分词”格式，即表示的是先前发生过的一件事情。比如，购买者提交商品订单之后发布OrderSubmitted事件，用户更改邮箱地址之后发布EmailAddressChanged事件。

需要注意的是，既然是领域事件，他们便应该从领域模型中发布。领域事件的最终接收者可以是本限界上下文中的组件，也可以是另一个限界上下文。

领域事件的额外好处在于它可以记录发生在软件系统中所有的重要修改，这样可以很好地支持程序调试和商业智能化。另外，在CQRS架构的软件系统中，领域事件还用于写模型和读模型之间的数据同步。再进一步发展，事件驱动架构可以演变成事件源（Event Sourcing），即对聚合的获取并不是通过加载数据库中的瞬时状态，而是通过重放发生在聚合生命周期中的所有领域事件完成。

**总结**

DDD存在战略设计和战术设计之分，过度地强调DDD的技术性将使我们错过由战略设计带来的好处。因此，在实现DDD时，我们应该将战略设计也放在一个重要的位置加以对待。战略设计帮助我们从一个宏观的角度观察和审视软件系统，其中的限界上下文和上下文映射图帮助我们正确地界分各个子域（系统）。DDD的战术设计则更加侧重于技术实现，它向我们提供了一整套技术工具集，包括实体、值对象、领域服务和资源库等。虽然DDD的概念已经提出近10年了，但是在如何实现DDD上，我们依然有很长的路要走。