**领域驱动设计(domain driven design)战略篇之一 战略 & Bounded Context**

2018年08月04日 15:57:37 [even\_he](https://me.csdn.net/abchywabc" \t "_blank) 阅读数：230

之前的文章主要从战术层面的角度介绍了ddd。在岛国也被称为轻量级ddd。它提供了一些概念如aggregate, entity, domain event和一些设计模式如repository, specification来帮助我们建模和设计。各种战术还有能够扩展的地方，有机会还会再写下去。不过从这篇文章开始会写一写ddd战略方面的知识。

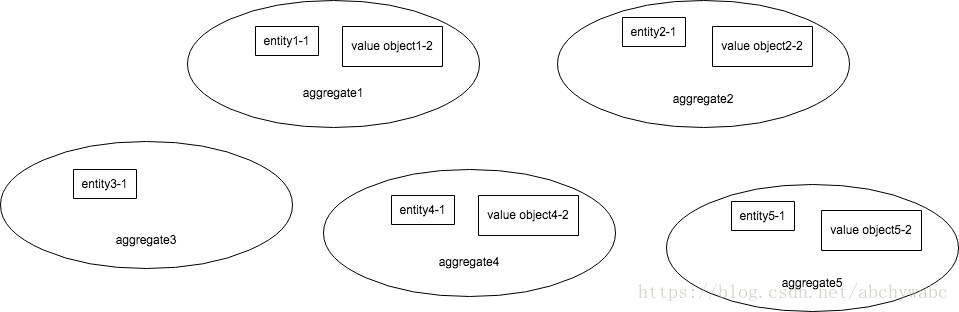
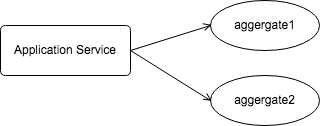
**战略 vs 战术**

究竟什么是战略与战术？他们有什么区别？   
目测这两个词都来源于战争（自己的理解），战术是偏微观的策略，目的是取得某场战斗或者战役的胜利。诱敌深入，敌进我退之类的可能都属于战术吧。而战略是偏宏观的策略，目的是赢得一场战争，它所关注的不限于军事方面，可能资源的调配，甚至会牵涉到外交等。   
扯远了～在领域驱动设计[domain driven design]中战术与战略的概念亦是如此。战术是微观层面的。在之前的文章中的例子中，基本都是为了解决某个十分具体问题而进行的模型设计。这些设计会反应具体的细节。比如一个如何去识别一个entity,一个entity会有什么样的行为。而这些设计会直接反映到我们的代码上。

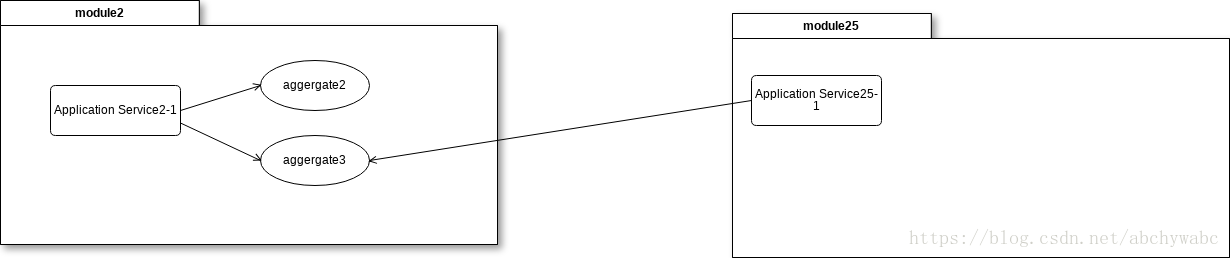
**为什么需要战略**

从很抽象的概念上来说，我们应该不难理解为什么我们需要战略。战略与战术他们需要解决的是不同层面的问题。当我们谈宏观问题时，自然是需要战略的。但话是这么说，什么从程序开发的角度来说，什么才是宏观问题？什么又是微观问题？这个好像很难用语言来具体地定义。   
假设我们完全不考虑宏观问题。我们想象（想象…）一下只用ddd中介绍的战术知识来进行设计。如今互联网产品的行业可能比较现实的制造产品的方案会是，先做proto type，然后做mvp(minimum viable product 只包含核心功能的产品)，然后在mvp的技术上进行后续开发。

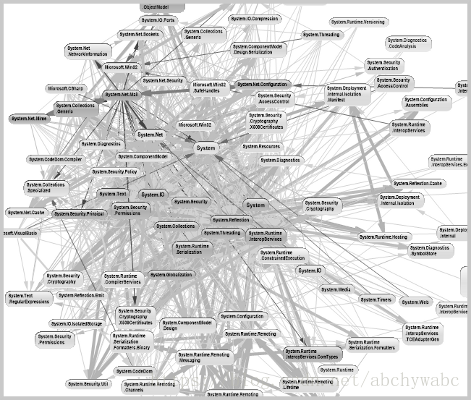
**mvp**

当我们在开发mvp的阶段时，产品的需求可能相对简单。我们分析需求，业务逻辑，然后设计能够描述这些业务的模型。可能10个到20个左右的aggregate就能解决问题。画个简图来表示我们定义的aggregate和其中的entity, value object。   
  
这个时候你的application层的application service可能不需要与很多aggregate交互。   


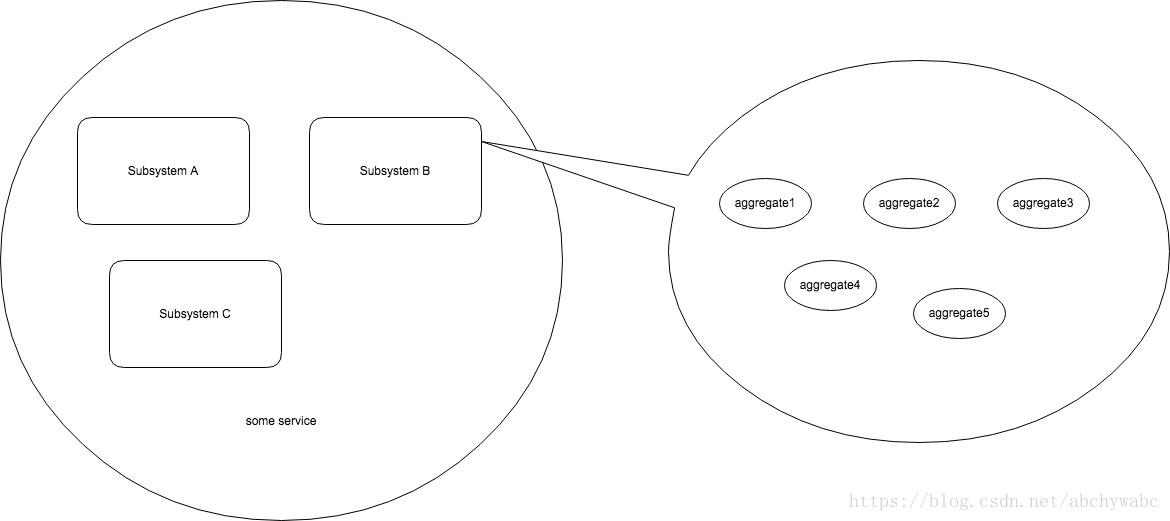
**产品进化**

随着产品经理对产品的认识更加深入，我们需要追加更多的开发，为了应对新的需求，我们要调整现有的模型，也需要增加新的模型来应对。某个时点，我们可以能已经到达了像图中的一个状况。我们已经无法再在一张图中精确描述aggregate中的元素。   
   
而此时，我们的application层的中的一些application service可能已经不得不与非常多的aggregate进行交互。   
   
当application service包含了很多的aggregate操作时，我们可能[不知不觉](https://www.baidu.com/s?wd=%E4%B8%8D%E7%9F%A5%E4%B8%8D%E8%A7%89&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)增加了各个模型间的耦合度，同时也冒着把业务逻辑写进application service的风险。理想情况下，我们希望所有的业务逻辑都在domain层，而application service是很薄的。   
另外随着模型量的增加，我们对模型之间的关系也会慢慢模糊。再加上如果是在多人项目中，程序员之间交流会有极限。对于系统的认知里为下降，各种“神奇现象“会开始发生。如下图，模块2里的aggregate3，在设计之初可能设想只有模块2里的东西会对aggregate3进行操作，但项目大了，参与开发的人多了，之前完全没有想到的不知名模块25也对aggregate3进行了操作。对模块25的修改可能影响到了模块2。   
  
注: 没有专门写文章说明module(模块)这个概念。如果你是java工程师，就把它当成package就行。

**进化的究极体—-big ball of mud**

当产品的复杂度不断增加，而我们有没有去控制控制这种复杂的话，我们的系统会成为ddd中称作big ball of mud（大泥球）的东西。   
   
这个说法应该是比较形象的，系统的各种模块已经耦合在一起，很难直观地，整体地理解这个系统的构成。当我们对它无法整体地理解它，[各种各样](https://www.baidu.com/s?wd=%E5%90%84%E7%A7%8D%E5%90%84%E6%A0%B7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的误判就会发生，系统的各种行为变得难以捉摸。对于程序员来说，在这样的系统基础上开发会是个噩梦。

**如何解决这种问题？**

前面描述的方法哪里有问题？我们把注意力只集中在了小范围的设计上，我没有对全局进行把控。可能我们在各个模块的层面上，设计是合理的，当把视野放宽时，系统设计却变得很混乱。   
很显然，除了战术层面的aggregate, module这样定义边界的概念，我们还需要一个抽象度更高，范围更广的概念来帮助我们把系统进行切分。其实这也符合一般的系统设计的思路，   
在传说中的waterfall模式时代，我们会把需求定得十分明确，把尽可能地具体到每个细节（至少岛国就是这种状况。各种纸质设计文档，一审再审，一改再改，最后文档叠起来的高度会不亚于“上海中心“）。但即使在那种设计难以应对变化的年代，在程序设计时也不会一下子就设计到最细小的部分。还是会先分大的区块。比如一个服务，我们需要用多少个子系统来组成它，如何分割子系统，子系统之间如何交互。这写就是战略层面上要考虑的问题。而在子系统中我们会在进行更细节部分的设计。如下图所示，先将服务分成A, B, C三个子系统。其中的子系统B考虑它应该包含什么样的模块。   
  
在[领域驱动设计中]提出了bounded context这个概念。直译过来时”边界确定的上下文”。这是一个帮助我们进行系统粗粒度切分的概念。

**Bounded Context**

个人认为Bounded Context强调的是边界这个概念。这个思想有一个前提，万能的模型是做不到的。我们建模是为了描述或解决现实的问题。而现实问题是复杂的，我们无法建立一个模型是面面俱到，一般我们只会截取自己所关注的一个切面，对它进行建模。

**同一个概念，不同模型**

比如在一个管理户籍时，我们只关注一个人的姓名，身份证号，住址。在写简历时，我们会关注一个人拥有的技能与工作经验。而在¥%@时，我们会关注一个人的性格等。在现实生活中同样的一个人的概念，它十分复杂，拥有很多的侧面，我们在建模时不会指望构建一个完美的模型，它能够复刻一个人。即使有这样的模型，它肯定也是极度复杂的（在本人目前的认知下），也很有可能超过了我们可以理解的范围。   
因此，模型在某一个bounded context中，它是有限的，仅描述它所关注的部分。按上面的例子来说，在一个户籍管理系统中，Person可能就是下面的一个类

class Person (val identification: Identification,

var firstName: String,

var lastName: String,

var address: String) {

}

* 1
* 2
* 3
* 4
* 5
* 6

而在一个求职服务里，Person会是这样的

class Person (val id : Long,

val userName: String,

var emailAddress: String,

var resumeId: ResumeId){

}

class Resume(val id: ResumeId,

var education: String,

var professionalExperience: String

var skills: List<Skill>) {

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

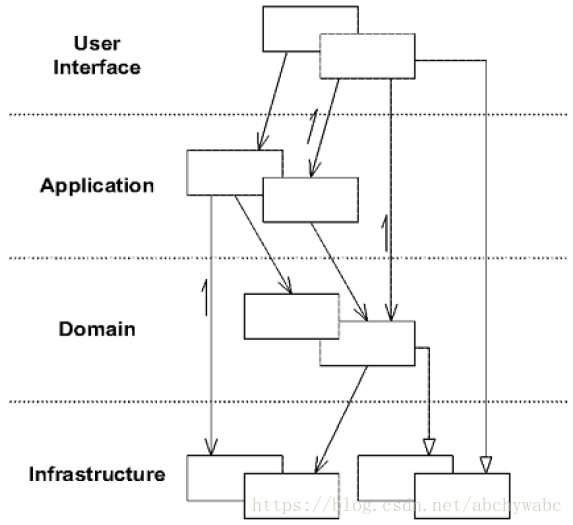
尽管这两个Person在现实中是相关的，即使他们共同存在于某一个平台，在他们各自的bounded context中，他们是相对独立的，可以想像，这两个Person拥有的行为也会是不同的。

**不同的语境，不同的概念**

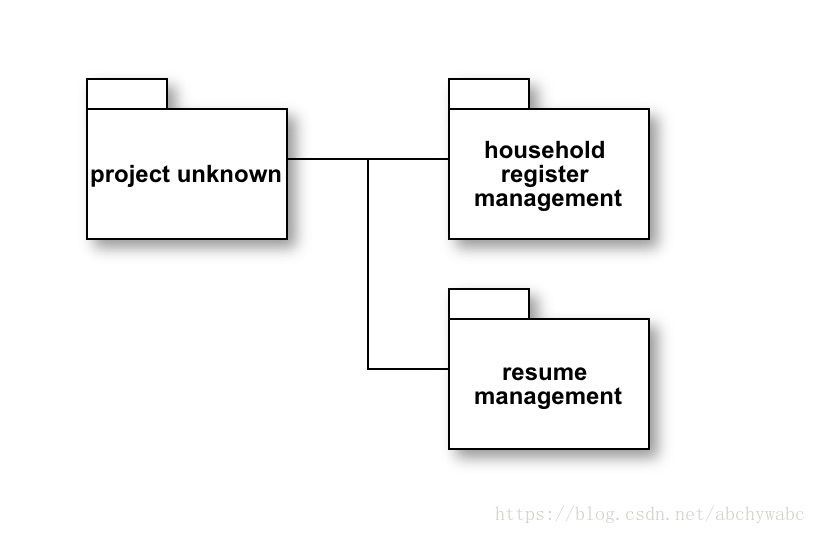
在英语中Account的这个词有不同的意思，一个表示银行的账户，一个表示账号。Account在银行业务，与网络业务的bounded context里会是完全不同的概念。   
我们必须认识到这种思路和自然语言是不同的。在自然语言里，一个词可能是多义的，意义也可能是宽泛的。Bounded Context这个概念要求我们在Context中的模型必须是单义的，意义相对狭义的。当一个Bounded Context中的一个模型类开了Bounded Context,它的意义与行为会发生改变。

Bounded Context让我必须认清我们建模时所应该关注的地方。明确我们所关心的点后，我们能更有目的性地建模。同时把不应该在某一个Context关注的东西放到另一个Context。再次提醒！同一个概念，可以出现在在不同的Bounded Context，但是他们会以不同的模型来展现。

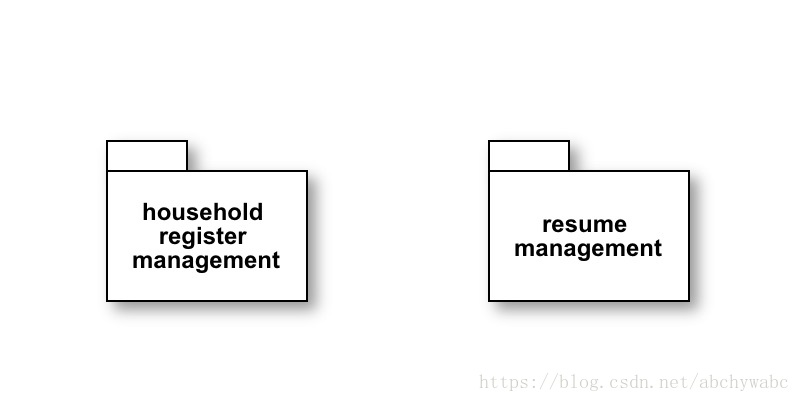
**如何实现Bounded Context**

很遗憾的是，[领域驱动设计]一书中并没有给出操作层面实现Bounded Context的方法。这也导致了Bounded Context的实现成为了一个众说纷纭的话题。这里列举一些实现的方式。   
首先Bounded Context并不是一个层，不是要在presentation, application, domain, infrastructure的4层架构中在加一个层。   


**namespace级别的bounded context**

如果是java的画，这个就是通过包(package)来构建bounded context。   


**project级别的bounded context**

   
project又是一个很有歧义的词，如果你是java工程师，那project就是代码层面的一个叫project的单位。使用如果你使用intellij开发的话，它会被称为module。总之他们有各自的代码，如果不使用build工具，project A是无法引用project B的代码的。

**micro-service级别的bounded context**

这个可能不必赘述了把。用一个微服务来实现一个context。

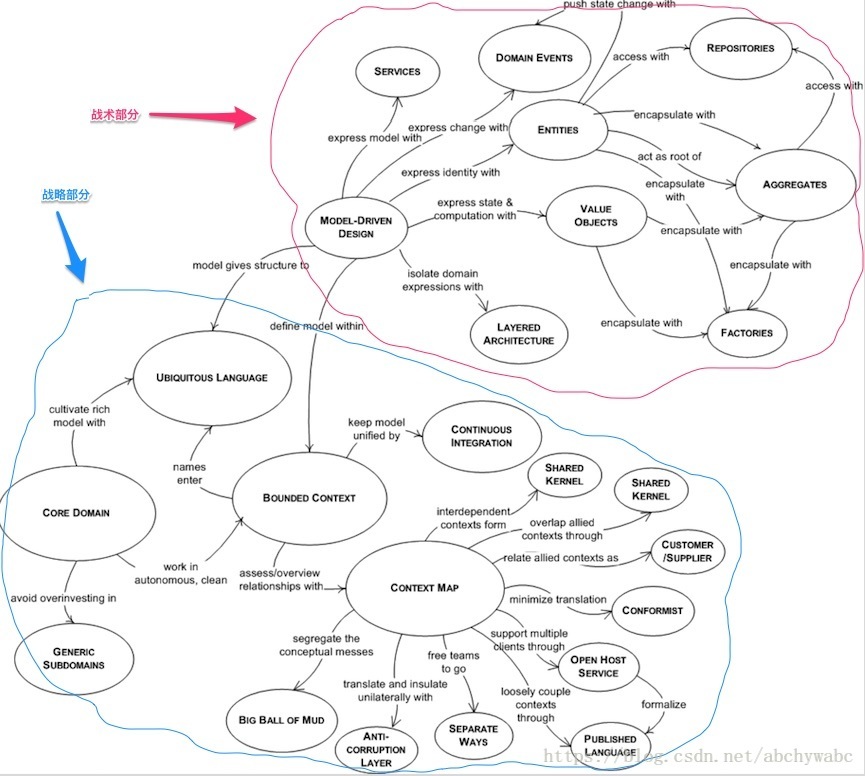
**比较**

从边界的强度来说   
namespace < project < micro-service   
使用namespace来实现的话，如果编程语言不对namespace之间类的引用有限制的话，这种边界定义是很弱，很容易打破边界（有意无意）。对越界（package A是否引用package B的类）的检查可能需要人力，从长期来看成本很大。   
project级别的话，基本不会有代码层面的越界。但是不同不同的project A, project B可能引用了同一个数据库，这可能会造成隐性的越界。比如A，B中的模块要对同一张数据表进行修改。但在A并不知道有B的存在。A所拥有的数据会在A不知情的情况被修改。   
micro-service的边界是最牢固的。如果你不开发接口，基本上做不到越界。   
边界初期投入上来说   
namespace < project < micro-service   
这个[显而易见](https://www.baidu.com/s?wd=%E6%98%BE%E8%80%8C%E6%98%93%E8%A7%81&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)。如果你选择micro-service,意味着你选择了一个复杂的系统架构，你可能还需要一大堆配套的框架。而如果是namespace的话投入基本就是0。

**如何实现bounded context是否是该尽早决定的**

很显然如何实现bounded context没有一个完美的方案（可能本人没有找到）。无论选择那个选项，都必须做出牺牲，这就成了令人烦恼的权衡问题。   
个人觉得，如要考虑的有如下几个要素   
1. 对边界的认识是清晰明确   
2. 团队技术实力   
3. 项目规模   
在这几个要素都十分明确的情况下，我们会有足够的信息作出选择。   
但如果在边界还不是很清晰，对domain的理解也不成熟，就直接用micro service进行划分是比较危险的。因为一刀割下去割得太彻底，之后发现割错了也没办法缝合了。你可能觉得那只要把需求理解清楚后，把边界明确定义不就行了。很可惜现实情况是需求也是在变的，我们很在初期就把握所有的需求。   
所以如果在各个要素都还不明确的情况下，比较实际的做法可能是，一开始选择边界强度低的实现方法，随着对domain理解的深入，明确边界后再过度到强度高的实现（micro service）。不过这种方法有一个隐性前提，边界分割方法的更改可以低成本地实现。如果monolithic的服务能够轻易地变换成微服务，那就不用逼我们在早期冒险做决定。这需要借助其他的一些手段。比如由尽量高的自动测试覆盖率等。

**总结**

这次讲了ddd的战术与战略的区别，战术帮我们优化局部，战略帮我们把握全局。   
介绍了bounded context的这个概念以及实现的方式。那究竟如何能把一个系统分解成bounded context呢？之后的战略篇文章会进行说明。   
网上发现了一张不错的图，总结了在ddd中，战略和战术都有那些概念。战略部分的概念之后会进行讲解。   


参考资料   
[BoundedContext](https://martinfowler.com/bliki/BoundedContext.html)

[**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。

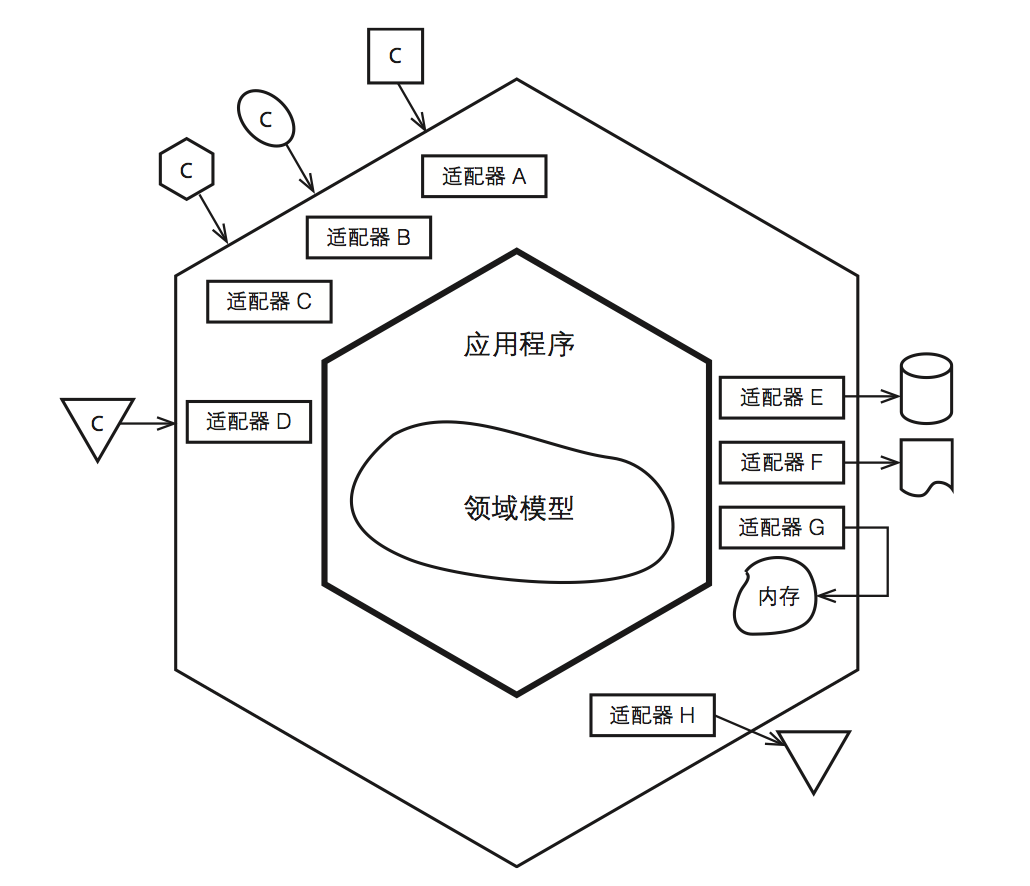
当下，面向接口编程和依赖注入原则已经在颠覆着传统的分层架构，如果再进一步，我们便得到了六边形架构，也称为端口和适配器（Ports and Adapters）。在六边形架构中，已经不存在分层的概念，所有组件都是平等的。这主要得益于软件抽象的好处，即各个组件的之间的交互完全通过接口完成，而不是具体的实现细节。正如Robert C. Martin所说：

**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。

[**IDDD 实现领域驱动设计－由贫血导致的失忆症**](https://www.cnblogs.com/xishuai/p/iddd-anemia-model-ubiquitous-language.html)

**啰嗦几句**

年前的时候，在和 [netfocus](http://www.cnblogs.com/netfocus/) 兄，以及对 [DDD](http://www.cnblogs.com/cate/ddd/) 感兴趣园友的探讨过程中，我发现自己有很多不足的地方，对 DDD 的了解也只是皮毛而已，代码写的少，DDD 的基本概念也不是很清楚，空有一腔热爱之情是做不了事的，后来我就多写技术代码，也记录了很多的技术问题，这让我收获很多，.NET 开源等等一系列的事件，也让我们 .NET 技术阵营看到了一丝希望。

后来，在探讨的过程中，有很多我不知道的概念被讨论，比如 CQRS、六边形架构、事件溯源等等，我对这些概念是一窍不通的，像六边形架构，我只知道六边形有六个边（莫笑），这让我意识到，你只了解经典 DDD 架构，会让你自己陷入一些困境，有时候不是你自己的设计问题，而是你的眼界被遮掩住了，你需要去探寻自己视野之外的东西，这样才会有所进步。

其实，学习 DDD 最好的方式，就是用最真实的实际案例去运用，在运用的过程中，去发现问题并进行探讨学习，这样虽然会很艰辛，但收获也是巨大的，除此之外，你还会发现另一个问题，就像在建高楼大厦的时候，虽然楼房的设计是世界最高水平，但是地基打不稳，空有一张设计图纸又有什么用呢？

读《实现领域驱动设计》这本书，其实在很早的时候就计划好了，之前也读了两三章，大概是写《[三个问题思考实体和值对象](http://www.cnblogs.com/xishuai/p/ddd-entity-value-object.html)》这篇博文的时候，读了下实体、值对象和仓储章节，因为是带着问题读的，所以并没有很深入，只是想可以尽快从书中找到自己的答案。

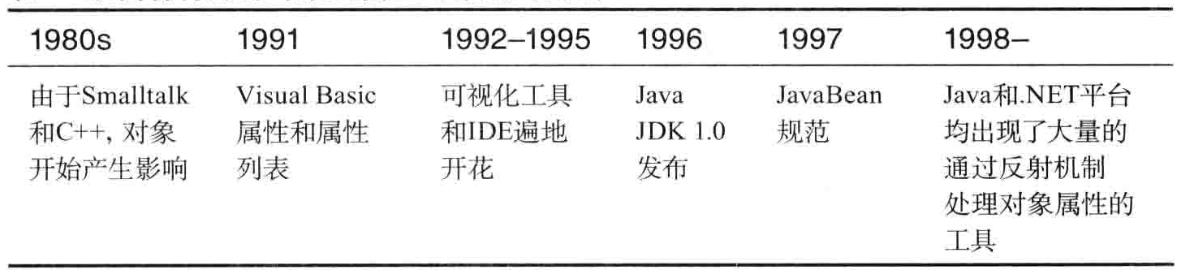
昨天晚上，我大概读了第一章《DDD 入门》的前半部分，有很多内容我觉得还是蛮有意思的，我希望可以把这些东西记录下来，以防备自己的“健忘症”。

**由贫血导致的失忆症**

先来看书中提到的两个病例测试：

1. 你的领域对象中是不是主要是些共有的 getter 和 setter 方法，并且几乎没有业务逻辑，或者完全没有业务逻辑－对象嘛，主要就是用来容纳属性值的？
2. 软件组件经常使用的领域对象是否包含了系统主要的业务逻辑，并且多数情况下你需要调用那些 getter 和 setter？你可能会将这样的客户代码称为服务层（Service Layer）或者应用层（Application Layer）代码，也或者，如果这描述的是你的用户界面，请回答“Yes”，然后好好反省一下，告诫自己一定不要再这么做了。

第一个问题是领域对象的定义，第二个问题是领域对象的调用，你的回答是什么？一个 Yes、一个 No？如果你是这样的回答，作者给你这样的分析：你可能是在自欺或者患上了由贫血症导致的神经系统紊乱。哈哈，作者还蛮调皮的，回归正题，考虑这两个问题的时候，你可以和你正在做的项目进行对比考虑，是不是对你产生了一些共鸣呢？有人可能会说：唉呀妈呀，这不是我“万能”三层架构里面的 Model 层和 BLL 层嘛？如果你这么想的话，对你的最终确认结果是：先生，你患上了贫血症，而且还“贫”的不轻呢。



上面是从富有行为对象到贫血对象的时间线，凡事都有存在的理由，像贫血对象也是，它也是由多种因素导致并演化而来的，在作者叙述的这一部分内容中，我觉得主要概括为两个因素：Microsoft Visual Basic 开发方式和早期 ORM 暴露共有属性，ORM 暴露共有属性这个我不是很懂，但是 Microsoft Visual Basic 开发方式对我还是蛮有影响的，记得在上大学的时候，老师讲 Web Forms 和 Windows Forms 的课程，都是一拖一个控件，然后再设置控件的属性，这样一个项目基本就完成了，从那时候开始，“属性”的概念就慢慢培养起来了，做一个项目之前，会先把一系列的 Model 属性设计好，按照需求下面就是对这些属性值的修改，最后就是把这些 Model 保存的数据库中，过程就是这么个过程，有错吗？没有，但是呢，好像建设一栋摩天大楼的设计不应该这么简单吧？我们看下面的代码（PDF 文件，不能复制，只能纯手打）：

public void saveCustomer(

String customerId,

String customerFirstName, String customerLastName,

String streetAddress1, String streetAddress2,

String city, String stateOrProvince,

String postalCode, String country,

String homePhone, String mobilePhone,

String primaryEmailAddress, String secondaryEmailAddress) {

Customer customer = customerDao.readCustomer(customerId);

if (customer == null) {

customer = new Customer();

customer.setCustomerId(customerId);

}

customer.setCustomerFirstName(customerFirstName);

customer.setCustomerLastName(customerLastName);

customer.setStreetAddress1(streetAddress1);

customer.setStreetAddress2(streetAddress2);

customer.setCity(city);

customer.setStateOrProvince(stateOrProvince);

customer.setPostalCode(postalCode);

customer.setCountry(country);

customer.setHomePhone(homePhone);

customer.setMobilePhone(mobilePhone);

customer.setPrimaryEmailAddress(primaryEmailAddress);

customer.setSecondaryEmailAddress (secondaryEmailAddress);

customerDao.saveCustomer(customer);

}

当时，看到这个 saveCustomer 方法中的代码，我哈哈大笑了三声，笑的不是别人，而是我自己，因为我之前写过比这个 saveCustomer 方法还多的代码，那个看起来更加臃肿，之前开发的是快递业务系统，一个表多的话有近上百个字段，那修改这个表的属性，就是像上面的代码一样，不同的是，我的比这个更多，一坨一坨的。比如上面，不管是地址变了没变，你都是使用的 saveCustomer，那这个方法到底是什么含义呢？你也说不清楚，因为它看上去是那么的“万能”，不过，也确实如此。因为你说不清一个方法的具体作用，这样导致的结果就是失忆症，原因是由贫血模型产生。

举个例子，有一天，业务人员告诉 DBA（业务实际掌握人），要去掉 Customer 中的一个属性，然后 DBA 就在 Customer 表中，把这个属性对应的字段去掉了，但是 DBA 并没有告知你，因为他觉得没必要（你又不懂业务），但是，你发现项目突然报错了，然后你就各种排查，最后发现是 saveCustomer 方法里面抛出的异常，然后你就开始一个一个比较 Customer 模型属性和 saveCustomer 表字段，发现原来是少了一个字段，然后，你就和 DBA 干了起来。。。

以上纯属虚构，如有雷同，那就雷同吧，针对 saveCustomer 出现的问题，作者简要总结了下：

1. saveCustomer() 业务意图不明确。
2. 方法的实现本身增加了潜在的复杂性。
3. Customer 领域对象根本就不是对象，而只一个数据持有器（data holder）。

以上的三大问题，就是导致“失忆症”发生的根本原因。

**Ubiquitous Language－通用语言**

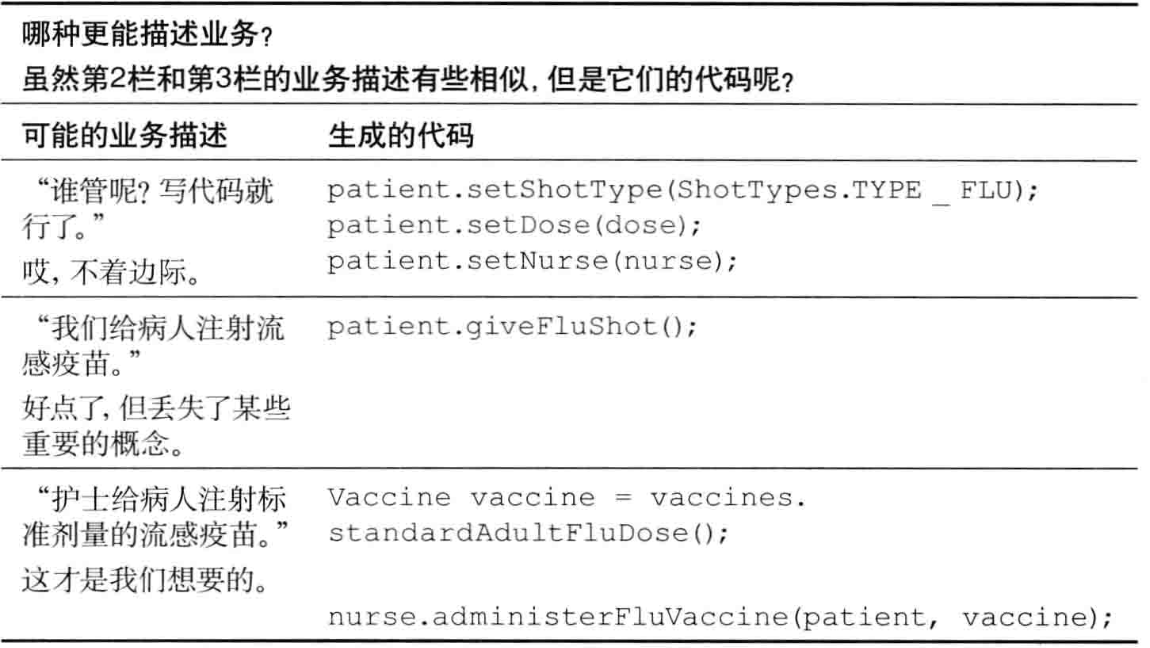
在领域驱动设计中，通用语言是非常重要的一个概念，在书中的第一章节中，作者也反复提到这个概念，并进行了详细解释，我之前认为通用语言就是代码，领域专家和开发人员都可以看懂的代码，但这种理解是片面的，领域专家是业务专家，他又不是开发人员，怎么能看懂代码呢？来看几个对话：

1. 很明显，通用语言是一种业务语言。  
   抱歉，不是。
2. 通用语言必须采用工业标准术语。  
   不完全是。.
3. 通用语言是领域专家专用的。  
   对不起，不是。
4. 通用语言是团队自己创建的公用语言，团队中同时包含领域专家和软件开发人员。  
   对了。

在理解通用语言之间，还有个问题也容易混淆，至少我是这样的，那就是设计和实现的区别，有人就说了：很简单啊，这有什么好混淆的，设计就是我们画的业务流程图或者是 UML，实现就是代码。仔细一想，好像也确实是这样，但是在领域驱动设计中，领域模型的设计是通过与领域专家进行讨论确定的，画的各种设计图，并不是领域驱动的设计，而只是我们建设讨论的一种方式而已，那设计是什么？设计其实就是代码，代码就是设计，所以，在领域模型的设计中，不要把设计和实现的概念区分开，他们其实是一个概念而已。

在上面对话的第四点中，通用语言是团队自己创建的公用语言，什么意思呢？公用的意思，就是所表达的内容领域专家和开发人员都懂，语言其实不是说话的语言，中文？英文？都不是，也不是代码语言，它其实是沟通的一种方式，大家都可以理解的一种方式，一个团队有一个属于自己的公用语言，范围是仅限于团队内容，可能这个公用语言在其他团队就不适用了，也就说，它是团队成员自己创建的，当然也不是一下就可以创建出来的，是一步一步进行完善，需要每一个领域专家和开发人员的参与。

不管怎么理解，公用语言概念中，有一点是非常重要的，那就是沟通，可能说多了不好理解，作者就举了一个示例：



你会发现，业务描述的不同，最后实现的代码就会千差万别，也就是说开发人员和领域专家的沟通很重要，当然开发人员的理解能力也很重要，很多的方方面面，就组成了通用语言的概念。

如果你不知道怎么理解通用语言，你可以尝试用一个最小业务用例去实现并理解，比如，修改客户名称业务用例，你可以先把这个业务用例中所涉及的概念抽离出来，比如客户、客户名称，修改客户名称，需要首先找到这个具体的客户，当然，可能会有一些限制操作，但不管怎样，“修改客户名称”这个业务所表达的结果，就是这个客户的名称要被修改，所以你要实现修改客户名称这个操作，可能实现的代码就是下面这样：

public void changeCustomerPersonalName(

String customerId,

String customerName) {

Customer customer = customerRepository.customerOfId(customerId);

if (customer == null) {

throw new IllegalStateException("Customer does not exist.");

}

customer.changePersonalName(customerName);

}

上面的实现代码和之前的 saveCustomer 代码，很明显的区别，首先，你修改客户名称，如果你使用的是 saveCustomer，你需要在方法参数中，传递一大堆的 null，而且整个方法内部充满了一些没必要的操作，而且你把 saveCustomer 方法描述给领域专家听，我想他们肯定也会不知所云，相反，changeCustomerPersonalName 就是通用语言的一种表达方式。

一个业务用例，从一开始的讨论，到最后的实现，整个过程中所涉及的方方面面，其实都可以理解为通用语言的表现，关于通用语言的界定问题，作者还提到几点：

1. 通用语言在团队范围内使用，并且只表达一个单一的领域模型。
2. 只有当团队工作在一个独立的限界上下文中时，通用语言才是“通用”的。
3. “通用语言”并不表示全企业、全公司或者全球性的万能的领域语言。
4. 每个限界上下文都有自己的通用语言，而有时语言间的术语可能有重叠的地方。
5. 。。。

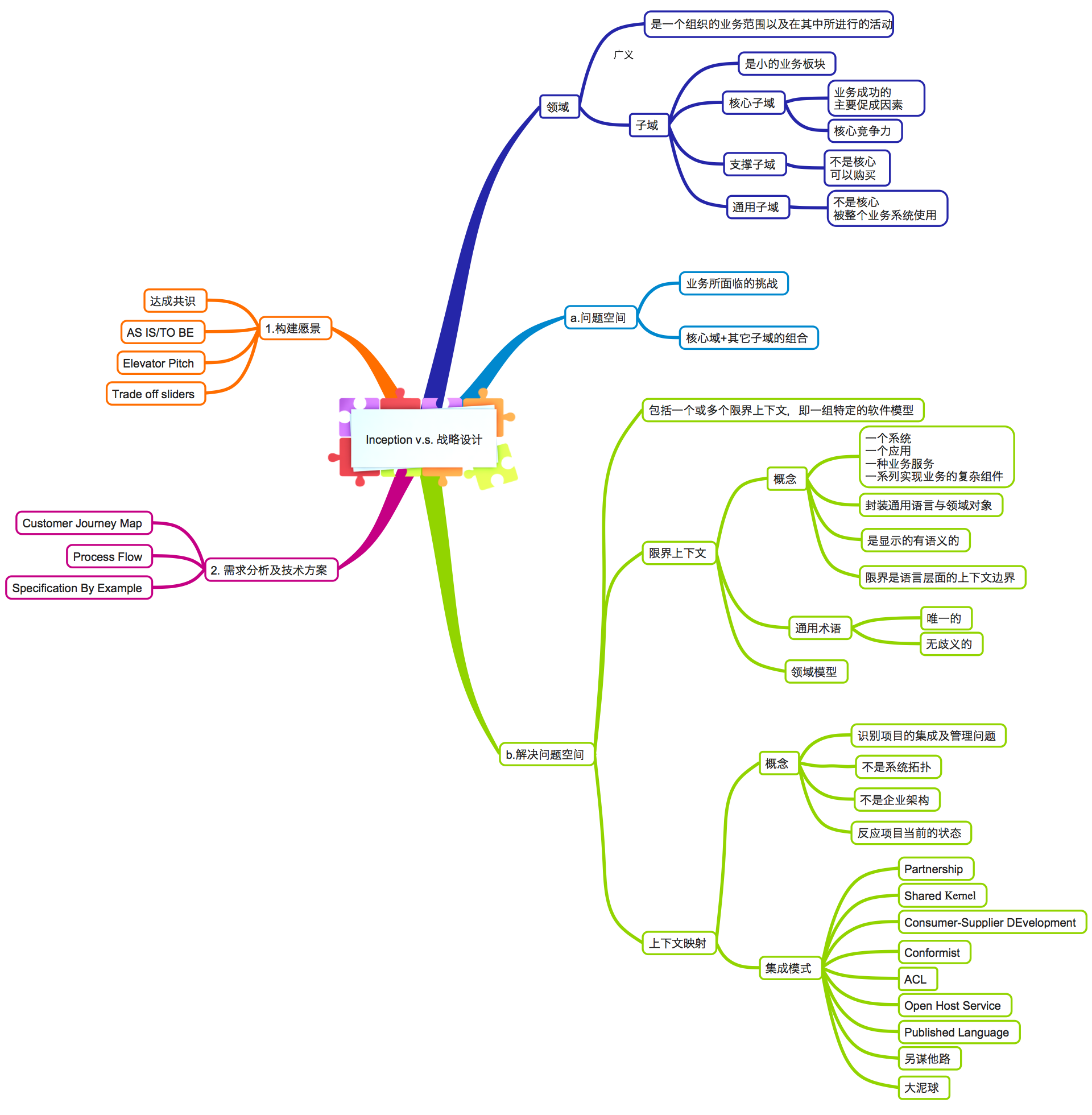
以上几点警示你，通用语言有一定的界定，并不是所有团队，也不是一个项目，而是一个单一的领域模型或者一个独立的界定上下文，你可以把它理解为一个领域模型或者一个独立界定上下文的具体表现，或者称之为过程体现。

以上只是简单的概念整理，并没有一些实际意义，具体的体会只能在实践中更加深刻，就记录到这！

[**IDDD 实现领域驱动设计－理解限界上下文**](https://www.cnblogs.com/xishuai/p/iddd-bounded-context.html)

上一篇：《[IDDD 实现领域驱动设计－理解领域和子域](http://www.cnblogs.com/xishuai/p/iddd-domain-and-subdomain.html)》

《实现领域驱动设计》前两章内容，基本上读完了，和《领域驱动设计》不同的是，它把很多的概念都放在前面进行讲述了，比如领域精炼、界限上下文等等，在《领域驱动设计》中，是很靠后的内容，不过这样也好，可以让你从一个大局的视角去看待问题，由广到细的思路学习，我觉得也蛮好的。另外，随着一点一点的学习，你会发现，领域驱动设计越来越有意思了，有很多“新鲜”的东西等待发现。



一张很重要的图（无意间搜到），引自：《[Implementing DDD Reading - Strategic Design](http://xianjing.github.io/blog/2014/04/18/implementing-ddd-reading-strategic-design/)》

**战略建模（Strategic Modeling）和战术建模（Tactical Modeling）**

战略建模和战术建模，其实是《实现领域驱动设计》最前面的内容，位于《如何使用本书》部分，当时看的时候并没有很注意，但在前两章的内容中，发现有很多这样的字眼：“团队有人花额外的时间去了解战术模式、团队采用的是战略模式的建模方式。。。”，这就不得不让你回过头看下，什么是战略建模和战术建模？其实，关于这两点，作者并没有很准确的进行定义，只是分别描述了这两点内容的关键字，我们来总结一下：

* 战略建模：界限上下文（Bounded Context）、上下文映射图（Context Mapping）。
* 战术建模：聚合（Aggregate）、实体（Entity）、值对象（Value Objects）、资源库（Repository）、领域服务（Domain Services）、领域事件（Domain Events）、模块（Modules）。

像聚合、实体、值对象等，都可以称之为战术建模的工具，战略建模和战术建模的区别，你可以从字面上进行理解，战略的意思，就是从大局出发，是一种运筹帷幄的感觉，那为什么和界限上下文有关呢？在《[理解领域和子域](http://www.cnblogs.com/xishuai/p/iddd-domain-and-subdomain.html)》中，有一张很重要的图，领域是业务系统的全部，其中包含核心域、子域和通用子域，相对应的就是限界上下文，你可以把某一块的领域和限界上下文进行映射，他们都是通用语言的一种表述，在项目之初，领域专家和开发人员的工作就是探讨限界上下文的划定，这个非常重要，如果限界上下文的划定有问题，那么将来战术建模的进行将“一塌糊涂”，就像作者一个例子一样，团队成员将用户和权限限界上下文划到具体的子域中实现，最后导致了一系列的问题，后来，团队发现问题后，将用户和权限限界上下文重新定义为身份和安全限界上下文，并划分到通用子域中，最后的效果显而易见，避免了很多问题的发生，也增加了业务系统的灵活性。

如果你注意的话，会发现上面说的只是“纸面”上的探讨，也就是说都没有进行实施，所以才称之为战略建模，而战术建模可以理解为战略建模的实现，前提是界限上下文都已经划定好，并确定无误。

**问题空间（Problem Space）和解决方案空间（Solution Space）**

和战略建模、战术建模一样，又是一个概念性的问题，在问题空间中，我们思考的是业务所面临的问题和挑战，而在解决方案空间中，我们思考的是如何实现软件以解决这些业务挑战。

具体什么意思呢？其实，问题空间和战略建模的概念有些类似，但只是思考的方式类似，他们是两个不同的概念，在上面图中，问题空间包括两部分：业务所面临的挑战、核心域+其他子域的组合，注意其中并不包含限定上下文的划分，领域专家和开发人员在探讨领域的设计中，首先，就是对问题空间的探讨，用来确定核心域和其他子域，并列出业务系统中可能会存在的一些问题。

在上面图中，解决方案空间包含的内容很多，它是什么的解决方案？其实就是针对问题空间的解决方案，当问题空间被确定下来后，我们就会对核心域以及其他子域进行探讨和实施，然后在其中划分出很多的限界上下文，并用软件的方式进行实现。

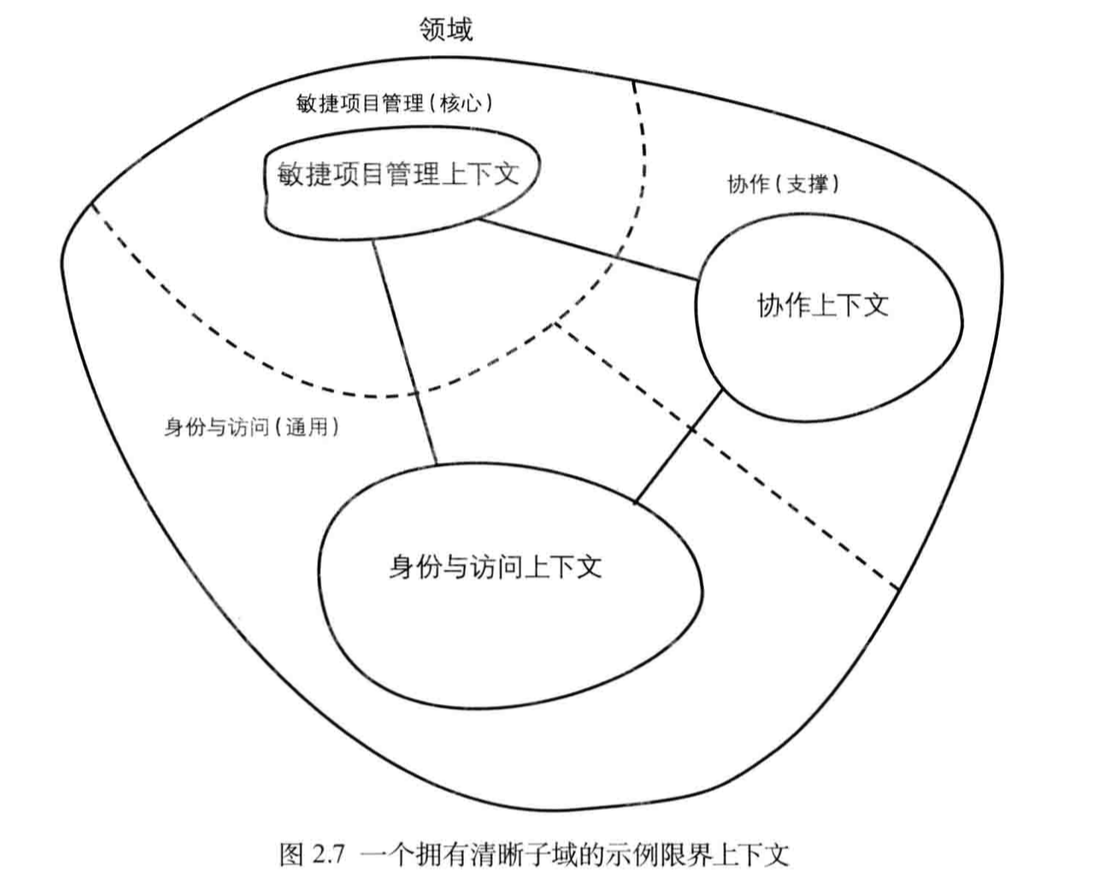
如果这样进行思考，你会发现，问题空间和解决方案空间对应于战略建模和战术建模，他们之间是有一些相似处，比如一个是探讨、战略，一个是实施、实现，但还是有些不同，比如界限上下文是战略建模中的概念，对应与问题空间和解决方案空间，界限上下文却是解决方案空间中的的概念，可以说问题空间和解决方案空间涵盖的东西很多，像战术建模就可以看作是解决方案空间实施的一种手段。

问题空间和解决方案空间，你可以不把它看作是领域驱动设计中的概念，因为在原著《领域驱动设计》中并没有这些概念，并不是说没有就不重要，在实现领域驱动设计中，还是非常重要的，你可以把它看作是一种思考的方式，就像你切一个西瓜，横切、竖切、还是直接用拳头爆掉，这些方式都可以，不管怎么实施，只要最后能吃到西瓜就行。对于领域专家和开发人员所建立的通用语言，到底该如何沟通，或者相互直接如何表达？我觉得探讨问题空间和解决方案空间，是一个很好的方式，你可以把他们看作是切西瓜的“刀”，很锋利，也高效。

**理解限界上下文（Bounded Context）**

上面的四点概念，在领域驱动设计的时候，可以不必了解，因为它只是实现领域驱动设计的一种概念方式，理解它也只不过可以让你少走些弯路，你完全可以按照自己的方式去实现，当然，偏离了大道，也怨不得别人。

限界上下文的概念很重要，我之前在做消息项目的时候，不是很了解这个概念，只是隐约记得什么限定上下文、界限上下文，然后就是实体、值对象和领域服务了，其实最准确的名字是限界上下文，限的意思就是划分、规定，界就是界限、或者一个边界，上下文就是业务的整个流程，总的来说，可以称限界上下文为业务流程在一个划定的界限中，我们知道，业务的描述是通过通用语言来表述的，限界上下文和通用语言的关系就是：在一个特定的限界上下文只使用一套通用语言，并且保证它的清晰性和简洁性。

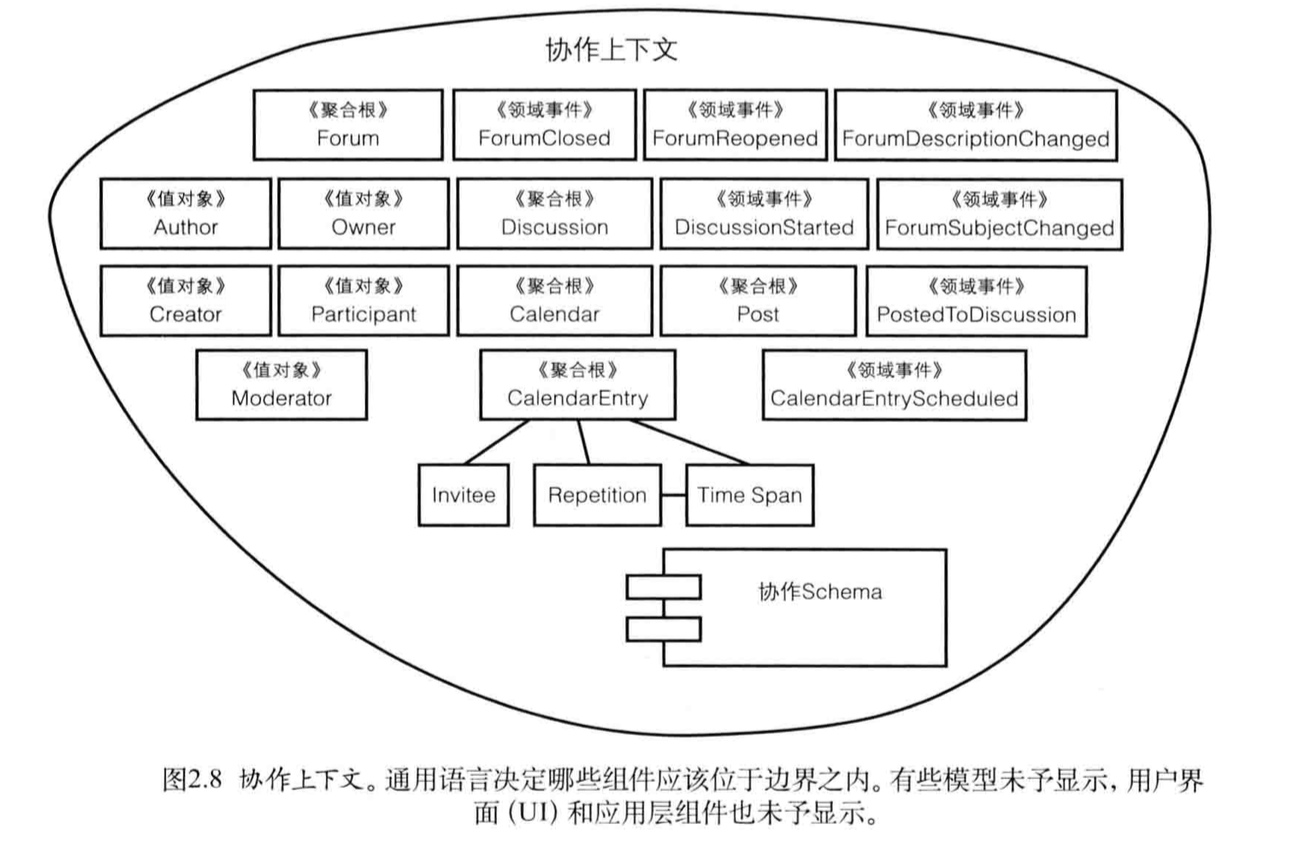


上面的图来自《实现领域驱动设计》，这个图我们可以和上一篇进行对比下，在之前的团队开发中，是把身份与访问上下文划分到协作上下文中了，并导致了一系列的问题，协作上下文包含的内容有论坛、博客、及时消息、留言板等，但这些都不是核心域，核心域是敏捷项目管理，也就是一开始说的那个简单业务用例：待定项提交到冲刺中，协作上下文只不过是支撑子域，它的作用就是用来支撑敏捷项目管理上下文的，可以这样说，如果协作上下文出现了问题，并不影响这个项目的运行，顶多是影响某一模块的运行，比如待定项提交到冲刺中，这个业务操作完成后，会有一个消息通知，协作上下文出现了问题，消息通知发不出去，但是待定项是可以提交到冲刺中的，因为这两个业务操作分别处于不同的限界上下文中，也可以这样说，对于敏捷项目管理上下文，协作上下文是可以替换的。

那限界上下文和子域有什么关系呢？在上面图中，可以看到是一一对应的，比如通用子域对应于身份与访问上下文，但其实并不是这样，请注意那个虚线，虚线表示的意思是核心域和子域的界限，但界限中很多都是空白的，比如通用子域除了包含身份与访问上下文，还可以包含消息与通知上下文、日志记录上下文等等，同样，支撑子域也是如此。

我记得我在开发消息项目的时候，在领域层只有一个 MessageManager.Domain 项目，并且项目下有很多的文件夹，比如 Entity、Domain Service 等等，然后我就认为这个 Domain 项目，是整个消息项目的核心，并且，如果我再开发一个新的项目的时候，我也会这样做，这样有什么问题呢？好像没什么问题，因为对于消息项目，业务场景很简单，Domain 项目所代表的是整个领域层，也就是上面图中整个的概念，其实这种命名是有问题的，实体、值对象和领域服务等概念，是存在于一定的限界上下文中，而不是整个领域概念，也就是说，我当时在设计 Domain 项目的时候，就完全没有把限界上下文设计好，暴露出来最明显的一个问题，就是 Domain 项目中包含有 User 实体的概念，你明白了吧，我和作者描述的那个团队开发都犯了同一个问题。

我们再来看一张图：



上面是协作上下文所包含的内容，你可以看到有好多的聚合根、领域对象等等，对于协作上下文的开发，IDDD 作者的做法是，新建一个程序集项目，也就是我们所说的类库项目，这个每个限定上下文都互不影响，而不是像我那样包含在一个 Domain 项目中，分开开发更新也方便，如果限界上下文足够复杂，比如上面的协定上下文，包含的聚合根太多，我们也可以进行细分。还有个问题是，比如用户的概念，在博客、论坛、日历等场景中，所表达的概念是不同的，那我们的身份与访问上下文该如何进行设计，还有就是协定上下文中的用户概念改如何进行设计，这是一个很重要的问题，如果是我的话，我以前肯定会把用户的概念放在协定上下文中进行开发，因为消息项目我就是这么干的，但这样造成的问题也是很严重的。

对于上面所描述的问题，我们来分析一下，不管在博客、论坛、日历等场景中，用户的概念是唯一的，也就是说它必须是唯一标识的，不能有两个同样的用户同时存在，这是首要基本条件，还有就是，用户的一些基本属性，比如用户名、邮箱、密码等等，这些在不同的场景中都是可以确定的，也都是同样存在的，对待这些共有属性，我们可以抽离出来，除了属性之外，还有一些业务操作也是公用的，比如身份验证操作，我们也同样抽离出来，对于这些抽离出来的属性和操作，我们应该在哪边进行实现？该如何实现？是在协定上下文中吗？不是，我们应该把这些用户属性和操作放在身份与访问上下文中，并进行隔离实现，为什么要进行隔离？因为身份与访问上下文是在通用子域中，也就是说并不是在支撑子域中，通用子域和核心域、其他支撑子域都有联系，也就是说，不要把协定上下文中所包含的独有用户概念，放到身份与访问上下文中进行开发，如果这样做，那么身份与访问上下文就不是通用子域了，而变成了协定上下文的一个附属上下文。

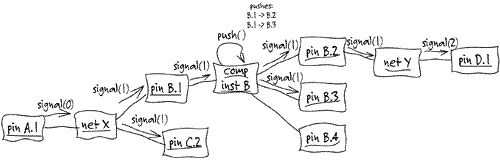
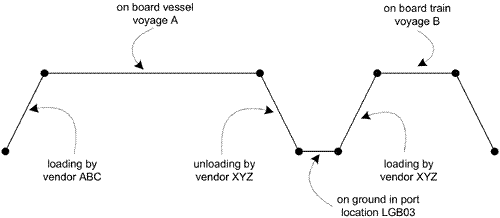
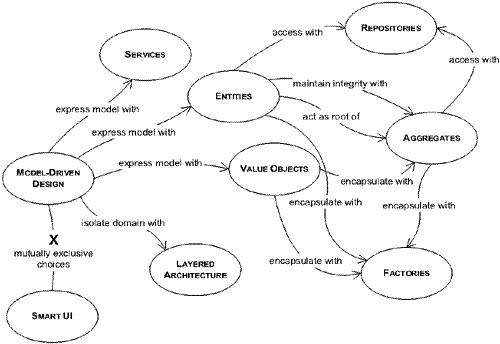
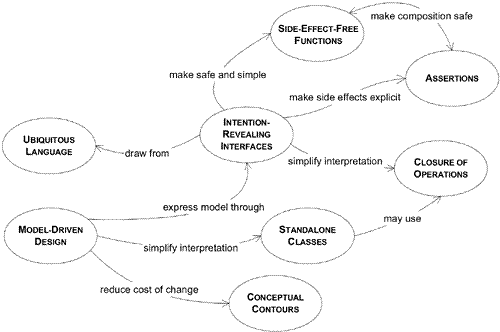
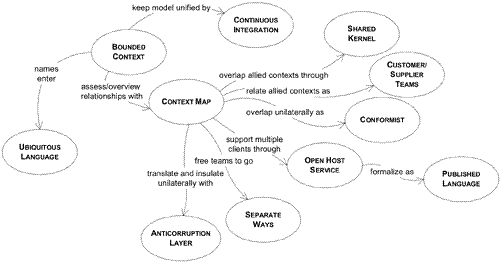
一个模型应该要与一个上下文相适应，上下文可能是指一段代码，也可能是指特定团队的工作，如果一个模型是在一次头脑风暴会议上诞生的，那么它的上下文就可能会限制在这些讨论的范围中，在有特定意义的模型中，不管模型的上下文是什么，必须要说明模型中的术语是什么意思。以上是《领域驱动设计》中，关于模型和上下文的内容，注意，上面所说的上下文并不是限界上下文，上面所说的上下文可以是一段通用语言的表述，也可以是一段代码，如果概括的话，可以认为是限界上下文的一部分。

关于限界上下文，我只是了解冰山一角，有太多的内容需要进行探讨学习，但不可否认，限界上下文是领域驱动设计中，最重要的概念之一，可以称之为最重要的首要概念，因为它是领域驱动设计的开始，自己肚里知识有限，我希望后面可以再次对这部分内容进行补充，最后，引用《领域驱动设计》中的一段描述：

* **细胞膜不仅能把细胞内部和外部区分开来，而且还能决定通过的物质。**

有人会说，你搞这么多的概念有什么用？还不如直接实践来的有用，但有时候，你会发现，实践是建立在一定基础之上的。

[**领域驱动设计(Domain-Driven Design)**](https://www.cnblogs.com/tommyli/archive/2008/03/24/1119420.html)

**Part I: Putting the Domain Model to Work**    领域驱动讲求将领域模型作为领域专家、分析人员、开发人员之间交流沟通的核心。传统的瀑布模型方式下，缺乏有效的反馈机制，在链路上领域知识以不同的表现 形式进行传递，知识的丢失容易造成需求与实现之间的断层。传统的迭代方式下，软件产品的优秀程度取决于开发者对领域知识的兴趣和掌握程度。  
 **Chapter One. Crunching Knowledge**  
    领域模型是领域专家和分析人员互相沉淀知识的一个工具，它帮助分析人员理解领域知识，也为领域专家提供一个规范的表达形式，有条有理的描绘领域知识，分 析、解决领域问题。另外，领域模型也是开发团队知识沉淀的一种方式，帮助开发人员了解他所从事的特定领域，提高建模技能。  
  
**Chapter Two. Communication and the Use of Language**  
    领域模型其实是一种语言，领域专家与分析人员、开发人员之间交流的通用语言。  
    一开始，分析人员与领域专家需要对这个通用语言达成一致，双方能熟练的运用领域模型描述问题，表达、分析、处理问题。  
    1. 领域模型不是图，图只是让核心、关键的概念清晰的呈现出来。图的表达能力有限，模型必须配备描述*（需求采集会议中的口头描述，或文档中的文字描述）*，将图形所代表的意义，以及图形中没有呈现出来的规则、断言、细节进行补充，才能完整地表述需求。  
    2. 领域模型的UML或者类UML图不能太细太完整，否则过于庞大的模型会干扰人的思维，阻碍对主要部分，或者复杂逻辑的梳理。业务总是被切分成一个个片断进行分析，在每一个片断里，画出几个主要的对象和交互逻辑，细节的部分用文字记录、描述。  
    3. 领域模型中不应当出现设计、技术方面的术语，也不应当出现开发人员不理解的业务术语。  
      
    图：讨论过程中手绘领域对象模型  
       
    说明性模型：最好不使用UML，从而与领域对象模型明显区别开。  
      
    图：Explanatory Models  
  
    关于领域模型表达方法*（UML图？Visio图？）*的选择问题，标准的形式是使用UML，但如果领域专家或开发人员根本无法对UML图例形成清晰的概念*（将UML图映射到领域对象、逻辑，或代码实现）*， 将注定DDD方式应用失败。因为没有掌握语言就无法使用它正确交流。第三节中强调建模人员要直接与开发过程接触，重要的一个作用是确认开发人员对领域模型 这个语言是否正确理解，确保代码实现保持了建模者要表达的意图。因此DDD的一个关键思想是建模人员自始至终维护领域模型这个语言，以及它表达的内容，向 团队的各个角色*（领域专家、开发人员等）*诠释各个建模元素的含义，让各个角色掌握这个语言，运用它来表达、实现实际需求。这是分析人员最关键的职责，贯穿整个过程。  
    领域逻辑的表达不要受UML图的约束，否则可能是问题所在，适当的描述就可以很好的解决这个问题。  
    领域模型是语言，所以表达的准确性很重要。对英语系国家，在英语理解、英语词汇提取方面很自然，可惜Rose、PowerDesinger等都无法象QQ一样，在用户昵称和备注之间切换显示。所以项目需要花精力克服这个障碍。  
  
**Chapter Three. Binding Model and Implementation**  
    整个模型庞大关系错综复杂，会导致无法实现，模型应当清晰，拥有明显的边界而分出子模型。  
    分析模型独立于设计模型方式不可取，领域模型需要同步反映领域业务、分析、设计、编码实现。  
    建模人员必须直接与开发人员、与代码接触，把握代码实现过程中模型的约束，接受实现层面对模型的反馈。开发人员必须一定程度的参与模型讨论，与领域专家接触。  
 **Part II: The Building Blocks of a Model-Driven Design**  
      
    图：A navigation map of the language of MODEL-DRIVEN DESIGN  
  
    DDD跟Smart UI模式冲突。如何在界面快速的为Customer添加一个属性、定制一张报表，这些不是DDD需要考虑的。  
  
**Chapter Four. Isolating the Domain**  
    使用分层方式将领域模型隔离出来。  
    基础结构层：例如ORM、消息服务、日志服务、安全服务等。领域层：纯粹的领域逻辑处理。应用层：协调、组织领域层、基础结构层，构造完整的业务操作，例如事务控制等应当在应用层完成。  
  
**Chapter Five. A Model Expressed in Software**  
    关键因素：Entity、Value Object、Service。  
    1. 关联：关联设计的考虑*（方向的设计等）*，简化关联的实现。NHibernate基于主映射端做级联更新。  
    2. Entity：An object defined primarily by its identity is called an ENTITY，所以Hibernate在Identity映射、生成方式上提供强力支持，而对组合主建映射比较弱。  
    3. Value Object：基于值是否可改变，是否可以共享考虑。Hibernate中数据类型的IsMutable属性是对这种情况的处理。  
    4. Service：常见的形式如Manager等，特点是只做事情，无状态，不是实体和值对象固有的。注意区分领域服务和其它层服务，前者处理领域逻辑。  
    5. 模块：好的模块划分提高系统清晰度，复杂的划分导致领域实现的分散甚至无法实现，丧失DDD的优势。  
  
**Chapter Six. The Life Cycle of a Domain Object**  
    1. Aggregates. Aggregate root and boundary, reference issues, 实现生命周期、加锁范围等控制.  
    复杂的关联导航会导致实现上陷入泥潭，丧失对领域焦点的把握能力，什么时候使用关联实现内聚，什么时候使用查询实现解耦，需要良好的把握。  
    2. Factories. 工厂并非领域对象，只是一种设计构造，但它担任的是领域层的职责。从数据库加载聚合的各个部分，构造整个聚合对象，也是一种创建过程。  
    3. Repositories. 也是设计构造，主要职责是CRUD，以及使用各种属性进行query的方法。Hibernate中Session的一个重要角色，就是作为全局通用repository基础*（当然还有Unit of Work等其它职责）*。它还不能完全算是repository，因为repository的目的，其一是封装数据库检索技术的繁杂性导致对领域模型的关注重点发生偏移，其二是防止领域逻辑封装的泄漏，Session的HQL、SQL Query等违背了第二点。对每个聚合体*（例如采购订单）*设计一个repository类，其实就是poor model中Impl/Manager类增删改查部分的逻辑。  
    Repository与factory的区别，repository主要职责是查询，它将对象创建、加载委托给factory。  
  
**Chapter Seven. Using the Language: An Extended Example**  
    完整的示例，贯穿讲解整个第二部分的内容：Isolating the Domain, Distinguishing ENTITIES and VALUE Objects, Designing Associations, AGGREGATE Boundaries, Selecting REPOSITORIES, Walking Through Use Cases, Object Creation (Factory), Refactoring, MODULES等等。  
  
**Part III: Refactoring Toward Deeper Insight**  
    Deep models: 经过多次迭代重构之后，抽象层次较深，表现力较强的模型。  
    Supple Design: 柔性设计，不断重构的产物。  
  
**Chapter Eight. Breakthrough**  
    这个例子excellent，06年刚好碰到过完全类似的情况。一个项目中客户对一个功能模块提出了挑战性的需求，经过仔细的分析，我发现应当从核心领域 对象中再分离一个概念出来，不仅完美的解决当时的需求，之前10多家客户实施过程中的相关问题都能够非常优雅的解决。尽管最终的表现形式是设计上，你的类 结构、关系不同了，但最关键的是领域模型上的突破，是基于对领域的深层理解消化后，在业务分析上的突破。一般它会给领域模型带来一些新的名词概念、系统功 能模块，甚至是操作流程，但这些领域专家极为认同，更能应对复杂的领域业务。  
    Don't become paralyzed trying to bring about a breakthrough. 没有什么标准、定律指导我们去完成某一个具体业务的分析、优化设计，参考成熟优秀的系统，参考象BPML、WfMC这样有限的参考模型和 《Analysis Patters》，是一个捷径，但很多时候这些参考是空白的。基于DDD思想，在不断迭代重构的过程中加深对领域知识的理解沉淀，为突破创造条件，是一个 不错的实践型方法。  
  
**Chapter Nine. Making Implicit Concepts Explicit**  
    开发人员必须敏锐的捕捉隐含概念：仔细聆听团队的表达用语，研究设计中不协调的地方以及专家看似矛盾的说法，查阅领域相关的文献，尽量挖掘隐含概念。  
  
**Chapter Ten. Supple Design**  
    柔性设计是隐含概念的补充，隐含概念挖掘出来后你的手头已经拥有了足够的materials，将这些材料组织成灵活的设计形式就是柔性设计范畴。要防止柔性设计变成overengineering，避免过分的抽象等间接层次的出现。  
      
    图：Some patterns that contribute to supple design  
  
    1. Intention-Revealing Interfaces: 名字即表达用意，主要在接口*（公共对象的public方法）*设计上，不要让写client的人对接口方法名摸不着头脑。是应该好好学英语呢还是应该开发个中文语言？  
    2. Side -Effect-Free Functions: 边际效应/副作用 是状态的记录、改变造成，象deep copy*（Hibernate等ORM中就用的很多）*就是为了避免side effect。  
    3. Assertions: design by contract的产物，就是在方法的组织设计上，避免出现理解上的多义性。通常情况下大家都是通过注释来进行说明，但DDD讲究代码就是对领域模型的一种描述。  
    4. Conceptual Contours: 粒度、边界的精细设计优化问题。  
    5. Standalone Classes: 降低关联依赖。  
    6. Closure of Operations: 指返回值跟操作对象类型相同，主要用在值对象上，例如实数相乘结果仍然是实数，xslt转换xml后结果可以仍然是xml。在开源框架中这种模式经常可以见到。  
    7. Declarative Design: 声明性设计的运用比较广泛，例如Hibernate使用xml配置映射关系，这就是一种声明，框架使用这个声明实现对象与数据库记录的转换操作，其它的例如流程引擎、一些MVC框架等等。  
    一般通过反射或者代码生成实现，缺点：a) 表达能力的不足可能使模型受限，开发束手束脚；b) 代码生成的破坏性可能扰乱迭代过程。  
    其它的形式：Domain-Specific Language, From the Ground Up*（一些函数编成方法）*等。  
  
**Chapter Eleven. Applying Analysis Patterns**  
    实例说明accounting model分析模式的使用。  
  
**Chapter Twelve. Relating Design Patterns to the Model**  
    说明领域模式和设计模式的区别：领域模式为领域问题提供模型，设计模式侧重于代码实现层面。  
    拿Strategy, Composite模式示例了一下。  
  
**Chapter Thirteen. Refactoring Toward Deeper Insight**  
    低级的重构是觉得代码实现不合理，refactory一下。深层次的重构指领域模型层面，不断与领域专家对话，查看领域文献，参考分析模式等成熟模型。  
    They see the risk of changing code and the cost of developer time to make a change; but what's harder to see is the risk of keeping an awkward design and the cost of working around that design. 这句话说的很好。  
  
**Part IV: Strategic Design**  
    把整个系统的设计图画在一面墙上，估计没有几个人能看懂；协作开发过程中，边界上的约束、断言等清晰程度，影响着开发过程控制和质量。因此在较高的层次，一些策略型的措施必不可少。  
  
**Chapter Fourteen. Maintaining Model Integrity**  
      
    图：A navigation map for model integrity patterns  
  
    作者蜻蜓点水的提到下面这样一些开发场景，如何控制开发方式，确保模型的完整性，提供一点参考：  
    1. Bounded Context: 两个团队协作开发，模型的重叠/交互部分，边界上下文是什么？两个团队都清楚吗？  
    duplicate concepts: 同一个概念在两个团队中存在不同的实现。false cognates: 对某个东西，两个团队以为理解是一致的，实际上存在分歧/偏差而没有被察觉。  
    2. Continuous Integration: 不是直到最后期限才进行集成，协作开发过程中分步骤地进行集成、测试，降低风险。但它会带来一些额外的成本开销。  
    3. Context Map: 透过边界的代码复用、数据功能的集成是一种风险，应当使用一个转换实现，提供一个全局的上下文映射视图和文档说明，集中管理，避免边界上下文的混乱。  
    4. Shared Kernel: 将两个团队模型重叠部分的一些子集作为共享部分维护。  
    5. Customer/Supplier Development Teams: 协作团队之间存在上、下游关系，典型的一些情况是底层框架开发团队与具体应用开发团队之间。  
    6. Conformist: 上下游团队不是同一个部门，或者由不同的人领导。  
    7. Anticorruption Layer: 与外部系统集成时隔离层的设计。  
    8. Separate Ways: 尽量避免集成。  
    9. Open Host Service: 过多的子系统集成不能都使用转换来实现，应当提供标准的协议/服务。  
    10. Published Language: 两个系统间的集成，实用任何一方的模型可能都不是理想的，采用一种标准/规范的交互协议可能是更好的选择。  
    11. Unifying an Elephant: 多个模型的集成问题。  
  
**Chapter Fifteen. Distillation**  
  
**Chapter Sixteen. Large-Scale Structure**  
  
**Chapter Seventeen. Bringing the Strategy Together**  
  
    精髓思想在前面几章中，后面的实在看不下去了。前面描述的是DDD思想，实例说明，以及一些方法论上的东西。后面部分侧重在一些实现、项目管理过程控制层 面，尽管是作者的一些宝贵经验之谈，但这种发散性的解说应用于实际项目状况是有待推敲的事情了。更何况很多情况，大家使用不同的设计思想、开发模式时，都 有遇到和处理过。

[**IDDD 实现领域驱动设计－理解领域和子域**](https://www.cnblogs.com/xishuai/p/iddd-domain-and-subdomain.html)

上一篇：《[IDDD 实现领域驱动设计－一个简单业务用例的回顾和理解](http://www.cnblogs.com/xishuai/p/iddd-domain-experts-and-specific-case-and-tdd.html)》

在《实现领域驱动设计》第二章的前半部分内容中，提到领域和子域的概念，并且作者把这两者又进行了细致的区分，其实在《领域驱动设计》书中，也有进行详细说明，只不过是在第十五章《精炼》中，章节比较靠后，我先是读了《实现领域驱动设计》这部分的内容，但读完之后，完全没有任何的感觉，或者说我自己和作者没有产生一些共鸣，也记不起来自己到底读了什么内容，但是在读《领域驱动设计》对应这部分内容的时候，我觉得有些内容是我想要的，也产生了一些共鸣，这让我对之前短消息项目也有了一些新的思考，我觉得还是蛮有价值的，下面是自己的一些理解。

一张很重要的图，引自：《实现领域驱动设计》

**Domain 领域**

首先，领域（Domain）不是领域模型（Domain Model），不要把他们两个画等号，从字面上进行理解，“领”的意思，可以理解为领土、领地或属于的某一区域，但不管它包含的是什么，它总是有一个界限，也就是说这个界限要进行明确，如果不进行明确，就会产生一些麻烦，比如“领地纠纷”等等，这个界限其实就可以看作是“领”的意思，“域”的意思，就是一个方面的具体称谓，加上“领”，领域的意思，其实就是明确某一方面的称谓。可能有点晕，我们开发某一套业务系统，比如快递行业的业务系统，那么这个快递业务系统之内的所有业务都包含在领域中，这个领域称之为快递业务领域，其中可能有很多的子领域，或者是业务模块，但都属于这个领域之内，如果超出这个领域之外，那么业务模块将不包含在快递业务领域之内了。在上一篇中有说到领域专家，其实，就是对某一领域精通的人，关键词是某一领域，而不是所有领域，强调的是界限的重要性。

从上面图中可以看出，业务领域中所有东西构成了这个业务的领域概念，也就是说它是唯一的，你开发什么业务系统，首先需要明确的是，你业务系统的领域是什么？如果连这个都确定不了，那么领域驱动设计也就没有再进行下去的必要了。让我自己来考验自己一下，之前开发的消息项目（MessageManager），这个消息项目的领域是什么？我想你应该会和我一样，脑海中首先想到的应该就是消息领域（Message Domain），其实我觉得这个答案没什么问题，消息项目不围绕消息领域开展，那围绕什么开展呢，你可能会有一些疑问，比如消息项目中会有一些用户模块，用户模块不应该属于用户领域吗？如果消息项目只有一个消息领域，那它们俩不相违背吗？其实这个答案，可以从上面的图中找到，消息领域是一个大的概念，它包含了这个消息项目中所有的业务领域概念，消息项目中的用户模块，只不过是消息领域的内部的一部分而已，不要被消息领域中的“消息”字眼所迷惑。

对于开发者来说，理解领域的概念是有一些歧异的，比如消息领域，我们开发者该怎么去描述它，或者表达它呢？我想你应该和我一样，首先，新建一个 Message.Domain 类库项目，然后把所有的业务操作都在这个类库中进行实现，实现完成之后，你告诉领域专家，说这个 Message.Domain 项目就是消息领域，仔细一想，好像也确实有些道理，对于开发者来说，消息领域的表达就是项目代码的实现，这样做也无可厚非。但是，有一点非常重要，在消息领域确定的过程中，不是只有开发人员的参与，最重要的还要有领域专家的参与和讨论，其实，对消息领域的最理解的人，不是开发者自己，而是领域专家，消息领域的表达也不只是代码表现这么简单，而是之前提到的通用语言，消息领域在开发者和领域专家之间的传递和确定，这个介质其实就是通用语言，之前有说到通用语言是开发者和领域专家共同创建的，它的表现形式可以是一个白板，也可以是一堆文件，又或者是一个项目代码等等，但不论是什么东西，它能在开发者和领域专家正确传递领域所包含的业务概念即可。

在上面图中，领域中有一个核心域（Core Domain）的概念，核心域是什么？它是领域中最重要的一块，你可以把它看作是人体中的心脏，也就是说是最核心的东西，开发者和领域专家花费最多的精力都是在它上面，一个业务系统一般有且只有一个核心域，但是一般核心域的精炼工作是需要花费很多的时间和精力，而且也很容易出错，如果业务系统中核心域的精炼出现了问题，那么这个业务系统注定是失败的，因为当一个核心域确定下来之后，开发者和领域专家剩余的工作，都是围绕着核心域进行展开的，比如一个人得了心脏病，而医生在诊断的时候，却认为是肝脏出了问题，然后他就对这个病人的肝脏做了手术，可想而知后果会怎样。。。这个看似简单的问题，但其实中间也蕴含了一些重要东西，首先，我们可以把它抽离出来，有三个人物：病人、诊断医生和手术医生，再想一下，我们业务系统开发，也有三个重要对象：业务系统、领域专家和开发者，然后，你再对它们进行对比下，就可以得出一些东西了，诊断医生是最了解病人的，由他来确定病人得了什么病，就好比领域专家把业务系统的核心域精炼出来，手术医生只不是实施者，用的是手术刀，而开发者用的是代码，但和看病不一样的是，核心域的精炼是领域专家和开发者共同探讨决定的，这就避免了诊断医生和手术医生所产生一些不必要的“冲突”，说了这么多，总而言之，核心域的精炼很重要，需要领域专家和开发者共同参与。

回到我们的消息领域上，那在消息领域中，核心域是什么呢？我的个人想法是发消息业务操作，这是消息领域中最重要的一个业务操作，消息的存在就是为了传递信息，在现实生活中，对于消息相类似的就是寄信，对于写信的我来说，我其实就是寄信的领域专家，为什么？因为我自己就是用户，对于寄信用户来说，我写信的目的就是让收件人可以收到我的信，具体这封信是怎么寄出去的，公路、铁路、飞机等等，这些我都不关心，我只关心的是现在这封信有没有到收件人手里。按照这个思路进行理解，对于消息领域来说，领域专家所能描述出来的核心域描述就是：发消息，这个和之前我们提到的待定项提交到冲刺一样，最简单的业务描述就是这样，至于消息发送限制、或者发送之后要不要邮件通知，这都是发消息具体的内部实现，对于消息领域以后的业务变化来说，变的也只是核心域的内部而已，我们只需要改变具体的实现就可以了，有点以不变应万变的意思，不变的是核心域，变的是核心域的内部实现。

描述这么多，可能有点晕，只需要记住两个概念就可以了，领域和核心域，具体的深入理解，只有运用后才可以真正体会到。

**SubDomain 子域**

什么是子域（SubDomain）呢？理解子域的概念，必须和核心域对应起来，子域也是领域的一部分，只不过它的重要性没有核心域那么大，它们之间的关系，你可以看作是人体器官中，心脏和其他器官的关系，在《领域驱动设计》中，其实并没有子域的概念，而是通用子域（Common Subdomain），而在《实现领域驱动设计》中，作者把子域拆分成了支撑子域（Generic Subdomain）和通用子域（Common Subdomain）。对于领域来说，除了核心域，用来支撑核心域的子域，就可以称之为支撑子域，在整个领域中，可以被公用的子域，称之为通用子域，通用子域还有一个理解是，在某一个业务领域中，它可能是被看作是通用领域，但是在另外一个业务领域中，它可能就被看作是核心域了，举个例子，比如在团购业务系统中，地图服务可能就被看作是通用领域，而对于地图服务商来说，毫无疑问，地图服务将是他们的核心域。

这些概念性的东西可能理解起来很费劲，我们再回到消息领域中，在上面分析中，我们已经确定了“发消息”是核心域，插一句，有人说，那回复消息是什么啊？难道有两个核心域？其实发消息就包含回复消息，因为回复消息也是发消息的一种信息，回到正题上，我们在消息领域中精炼一下支撑子域和通用子域，我现在可以想到的就是消息验证服务领域，就是在发消息之前对发件人、收件人、以及对消息内容进行验证，就好比你去邮局寄一封信，工作人员需要验证一下收发件地址一样，消息验证服务领域用来支撑发消息核心域，所以可以把它单独精炼出来进行探讨，还有就是，对于支撑子域来说，在整个领域中，可能会有多个，但大部分都是围绕核心域进行展开的，这也就是“支撑”的具体含义吧，说到这，我现在脑子里面有闪现一个，那就是消息发送之后的通知服务，这个也可以看作是支撑子域，需要记住的是，支撑子域虽然没有核心域那么重要，但它也是领域的一种，也是非常重要的，不要完全忽略它。

说了支撑子域，再来分析一下，消息领域中的通用子域，其实就是用户领域，用户的概念会贯穿整个消息领域，因为一切都是用户进行操作完成相关业务，用户领域的概念其实和上面说到的地图服务领域是一样的，在用户业务系统中，用户领域就不是通用子域了，而是核心域，对于非用户业务系统来说，用户领域在其他业务系统中，都可以被看作是通用子域。可能还有一点内容容易造成误解，就是消息领域中，“用户”的概念是很重要的，发消息是用户进行发送，那用户领域是不是应该被看作是核心域呢？我记得当时在开发消息项目的时候，曾经就把发消息业务操作放在了用户模型中，认为用户才能发消息啊，那如果是这样的理解思路，所有的业务系统中的业务操作，都应该是放在用户模型中，因为只有用户才能进行这些操作，但好像并不是这么回事，我们有点过于“面向对象”了，应用软件中和现实生活中的用户概念是不太一样的。

在业务领域中，对于通用子域来说，是比较容易精炼的，但是对于支撑子域来说，精炼它是有些难度的，难点就在于它和核心域的区分，有时候，我们可能会从核心域中剥离支撑子域，当然，最重要的，还是领域专家和开发者之间如何确立通用语言并与之沟通。

啰里八嗦的，就记录到这！