DDD——入门

〇、本章学习路线

- 了解DDD可以为你的项目和团队带来哪些好处
- 如何确定你的项目是否适合采用DDD
- 了解DDD的常见替代方案和它们将导致问题的原因
- 学习DDD的基础
- 学习如何向你的管理层、领域专家和技术成员推销DDD
- 了解使用DDD时所面临的挑战
- 看看一个正在学习采用DDD的团队是如何工作的

一、概述

- 领域驱动设计（DDD）作为一种**软件开发方法**，可以帮助我们设计出高质量的、能够准确表达业务意图的**软件模型**。
- 你应该期待从DDD中得到什么呢?

> 首先，DDD不应该是一个仪式性的过程，更不应该成为你项目进度的阻碍。
> 其次，DDD同时提供了**战略上**和**战术上**的建模工具来帮助我们设计高质量的软件模型。

- DDD首先**不是关于技术的**，而是关于讨论、聆听、理解、发现和业务价值的，而这些都是为了将知识集中起来，构建**集中化的业务知识体系**。
- 在实践DDD的过程中，你最好将那些**不怎么使用技术语言的人（业务领域专家）**加进自己的团队，此时你得仔细地聆听他们，还应该尊重他们的观点，并且相信他们比你了解得更多。
- 谁才是领域专家?

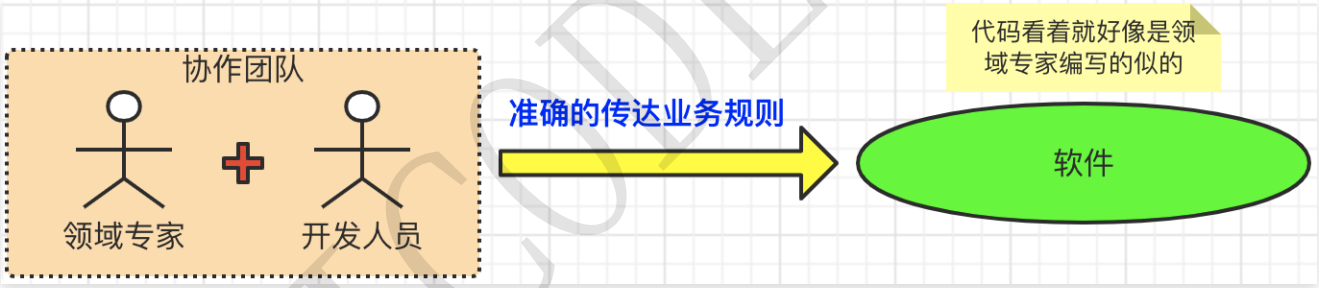
> 领域专家并不是一个职位，他可以是精通业务的任何人——销售、产品经理、软件开发人员.....

- 什么是领域模型?

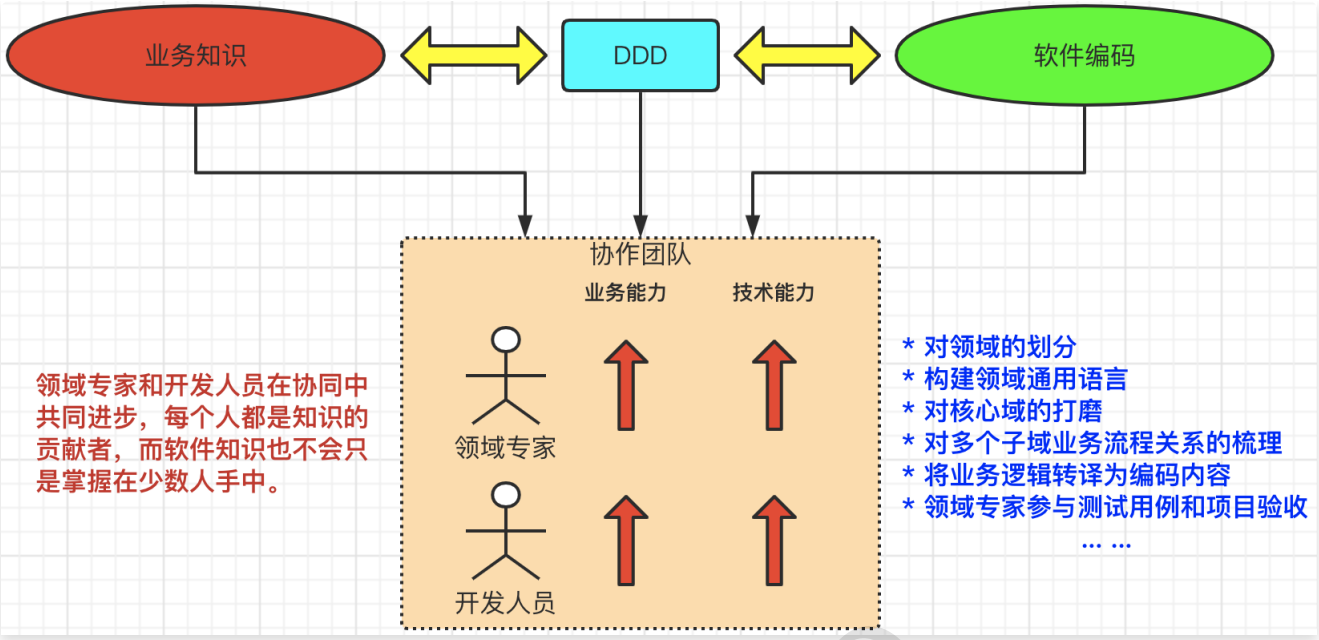
> 领域模型是关于某个特定业务领域的**软件模型**。通常，领域模型通过对象模型来实现（对象=数据+行为），并且表达了准确的业务含义。

二、为什么我们需要DDD

- 使**领域专家和开发者一起工作，将他们组成一个密切协作的团队**，这样开发出来的软件**能够准确地传达业务规则**（代码就好像是领域专家编写出来似的）。



- 可以**帮助业务人员自我提高**。没有任何一个领域专家或者管理者敢说他对业务已经了如指掌了，业务知识也需要一个长期的学习过程。在**DDD中，每个人都在学习**，同时**每个人又是知识的贡献者**。关键在于对知识的集中，因为**这样可以确保软件知识并不只是掌握在少数人手中**。以免开发人员编码好久，还依然不知道自己涉猎的业务到底是什么。



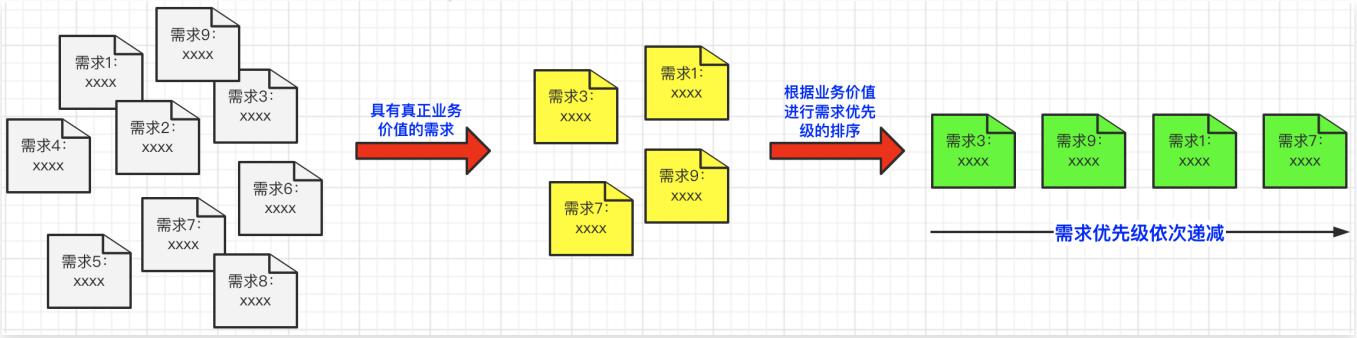
- 在领域专家、开发者和软件本身之间不存在“翻译”，意思是当大家都使用**相同的语言**（领域语言）进行交流时，每个人都能听懂他人所说的话。**设计就是代码，代码就是设计**。设计是关于软件如何工作的，最好的编码设计来自于多次试验，这得益于敏捷的发现过程。
- DDD同时提供了**战略设计**和**战术设计**两种方式。

> **战略设计**：帮助我们理解哪些投入是最重要的；哪些既有软件资产是可以重新拿来使用的；哪些人应该被加入到团队中。
> **战术设计**：帮助我们创建DDD模型中各个部件。

2.1> 难以捉摸的业务价值

- **首先，针对业务价值方面**

> 面对繁杂的业务需求和团队中良莠不齐的技术人员，如何确定**哪些需求可以真正的传递业务价值**？如何去**针对业务价值高低进行需求优先级的排序**？

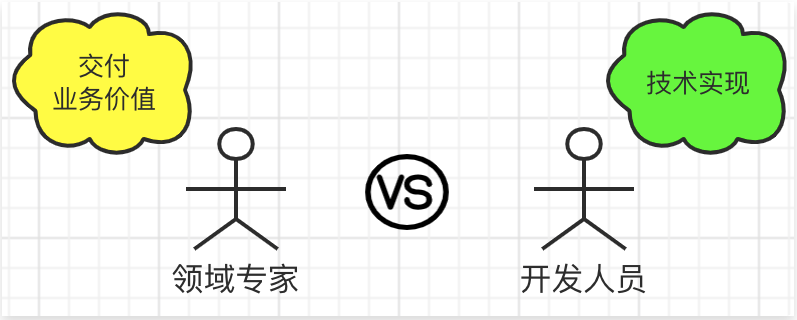


- **其次，针对领域专家与开发人员之间的关系**

> 在开发过程中，**最大的鸿沟之一就存在于领域专家和开发人员之间**。领域专家关注交付业务价值上，而研发人员关注在技术实现上。即便让他们一同工作，他们之间的协作也是表面的，这时候，在所开

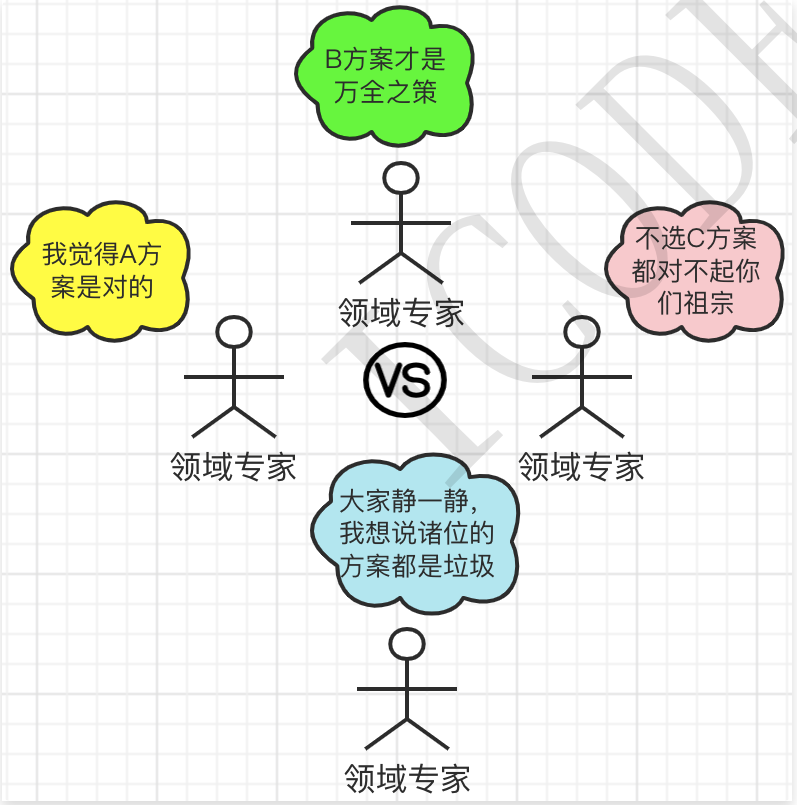
发的软件中便产生了如下的一种映射：

1. 将业务人员所想的映射到开发者所理解的。这样一来，软件便不能完全反映出领域专家的思维模型。
2. 随着时间的推移，这种鸿沟将增加软件的开发成本，而随着开发者转到其他项目或者离职，本应该驻留在软件中的领域知识也就丢失了。



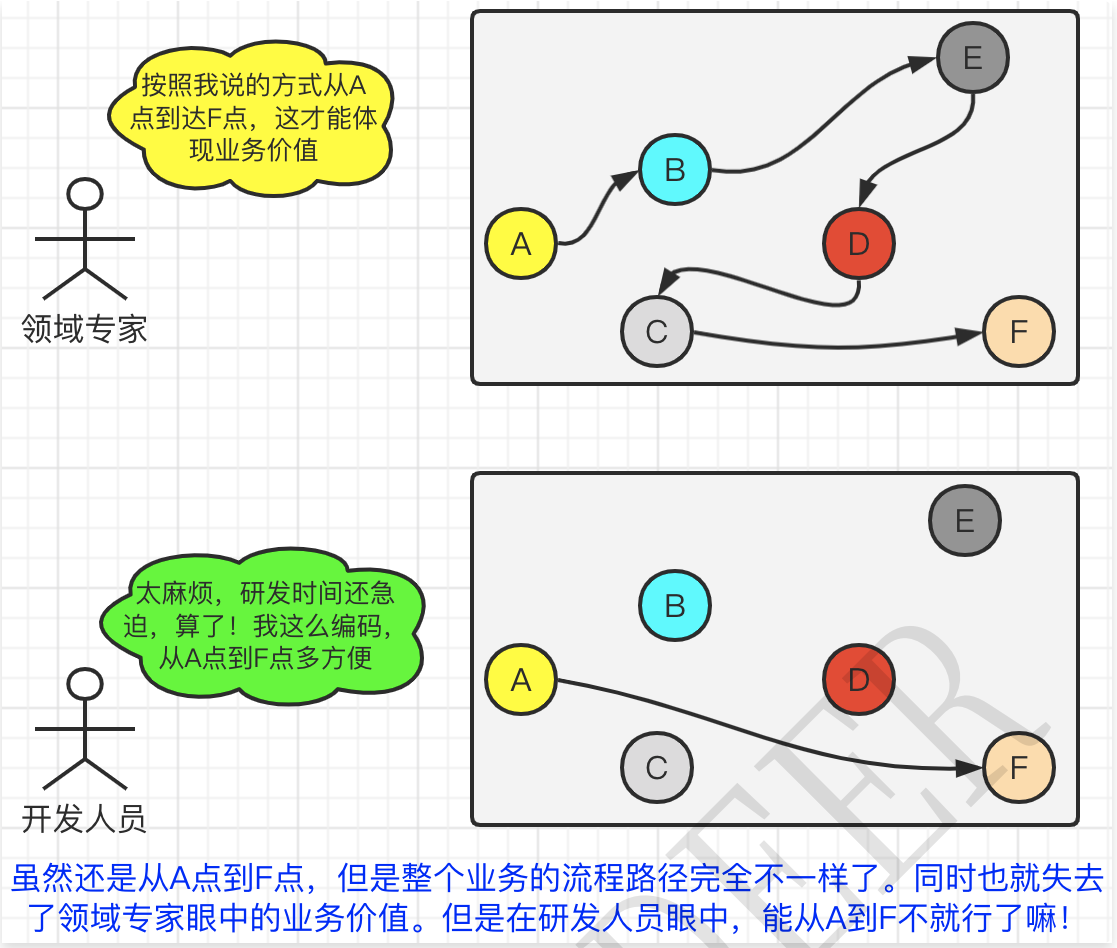
- **第三，针对多个领域专家之间的关系**

当多个领域专家之间存在分歧的时候，他们各抒己见的对业务需求进行影响，而没有一个“地方”可以让他们坐下来统一分歧，然后“一致对外”。在这种分歧的情况下，会很大程度的导致相互矛盾的软件模型。



- **最后，软件的技术实现可能错误地改变软件原有的业务规则**

需求交流的时候，大家彼此都没什么问题。但是，随着**开发过程中的很多不确定因素**，比如：临时插入紧急事情导致研发计划被打乱、研发时间大大超出合理范围、技术上无法对需求进行落地等等。**那么最终交付的产品功能，有可能就跟需求交流时的设想大相径庭了**。



虽然还是从A点到F点，但是整个业务的流程路径完全不一样了。同时也就失去了领域专家眼中的业务价值。但是在研发人员眼中，能从A到F不就行了嘛！

2.2> DDD如何帮助我们

- 那面对上面描述的诸多问题，DDD是否可以帮助我们。DDD作为一种软件开发方法，主要关注以下三个方面：

  - 首先，DDD将领域专家和开发人员**聚集到一起**，这样所开发的软件能够反映出领域专家的思维模型。这不意味着我们将精力都花在了对“真实世界”的建模上，而是**交付最具业务价值**的软件。领域专家和开发人员一起创建一套适用于领域建模的**通用语言**，所有成员后续都会使用通用语言进行交流。并且通用语言也有助于促使原本存在分歧的领域专家们达成一致意见，也会**增加团队凝聚力**。
  - 其次，DDD关注业务战略。DDD的战略设计用于**清楚地界分不同的系统和业务关注点，这样可以保护每个业务层面的服务**。更进一步，这将指引我们如何实现**面向服务架构**或者**业务驱动架构**。
  - 最后，通过使用战术设计建模工具，DDD满足了软件真正的技术需求。

- 在使用DDD时，我们首先希望**将它应用在最重要的业务场景下**，对于那些可以轻易替换的软件来说，你是不会有所投入的。正因如此，我们将这样的模型命名为**核心域**（Core Domain），而那些相对次要的称为**支撑子域**（Supporting Subdomain）。

2.3> 项目是否适合采用DDD

- **条件1**：如果你的软件是**以数据为中心，仅仅/几乎通过对数据库执行CRUD操作**就可以满足业务需求的话，那么你并不需要DDD。
- **条件2**：如果你的系统**只有25到30个业务操作**（而并非25到30个拥有多个方法的服务接口），这意味着你的程序中**不会多于30个用户故事**（user story）并且**仅包含少量简单的业务逻辑**，那么你并不需要DDD。
- **条件3**：如果我们当前的软件并不复杂，但是如果**有暗示说明系统以后的发展是复杂的**，那么我们就可以采用DDD了。
- **条件4**：软件的功能会**在接下来的几年里不断的迭代变化**，而你并不能预期这些变化只是简单的改变，那么采用DDD吧。
- **条件5**：你**不了解软件所要处理的领域**。你的团队中也没有人曾经从事过领域的开发工作。此时你需要和领域专家一起工作了，那么，采用DDD吧。

2.4> 贫血领域对象

- 什么是贫血领域对象?

> - 领域对象中主要是一些公有的getter和setter方法，并且在这些方法中，几乎没有业务逻辑，而仅仅是用来赋值或者获取属性值。
> - 针对领域对象的业务逻辑并不在领域对象中，而是被封装到了**服务层**（Service Layer）或者**应用层**（Application Layer）。

- 贫血领域对象是不好的，因为**它根本就不是领域对象**，而只是将关系型数据库中的模型映射到了对象上而已。
- 最初是在窗口设计领域上，将getter和setter变得过于流行了。而我们熟知的Java Bean标准最早是用来辅助Java的可视化设计工具的。此外，市场上几乎大部分框架都要求对象暴露公有属性。这样一来，开发者们只能被动地接收那些贫血对象。于是我们便到了“到处都是贫血对象”的地步。
- 我们以一个贫血对象为例，下面是其使用的例子：

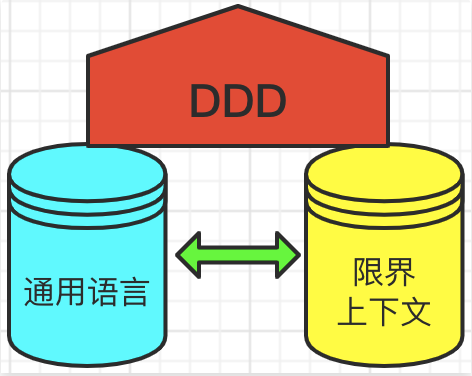
```
@Transaction
public void saveCustomer(String customerId,
                         String customerFirstName,
                         String customerLastName,
                         String streetAddress1,
                         String streetAddress2,
                         String city,
                         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);
}
```

【解释】

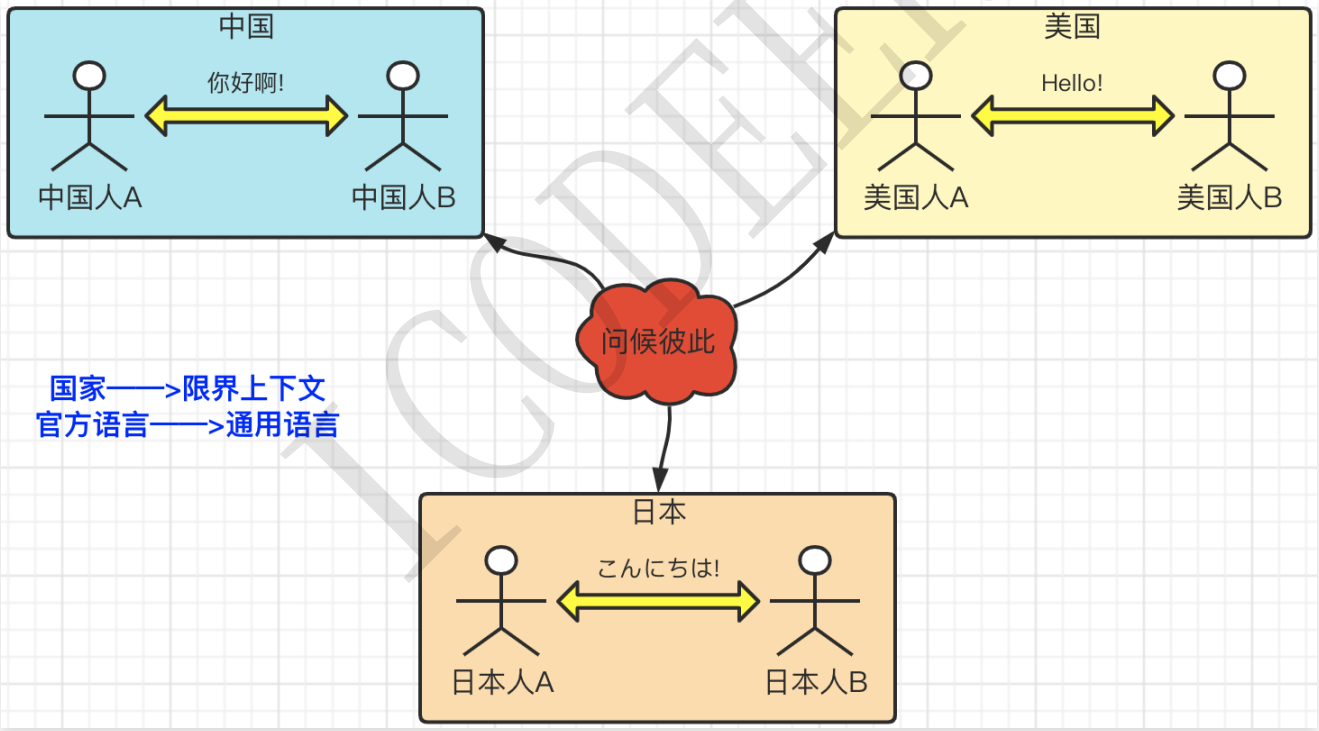
> - 以上代码很通用，可以根据入参去创建或修改customer实例。但是，上面的代码能够表达什么业务逻辑呢？我们通过上面的代码，并不知道saveCustomer()这个方法的业务场景是什么。为什么一开始会创建这个方法？这个方法的本来意图是什么呢？还是这个方法本来就是用来满足不同业务需求的？我从哪里可以知道创建这个方法的“鼻祖”程序员当时为什么创建这个方法呢？
> - 由于贫血模型不包含领域相关业务逻辑，所以通过贫血模型，我们根本无法解答上面的问题。此时我们就需要结合服务层（Service Layer）或者应用层（Application Layer）的代码，找到调用saveCustomer()这个方法的所有相关代码，再去逐一向上梳理业务流程或含义。要搞明白这里面的缘由，可能需要花上几个小时甚至几天的时间。

三、如何DDD——通用语言

- 在DDD中，**通用语言**和**限界上下文**，同时构成了DDD的两大支柱，并且它们是相辅相成的。



- 可以将**限界上下文**看成是整个应用程序之内的概念性边界。这个边界之内的每种领域术语、词组或句子——也就是**通用语言**，都有确定的上下文含义。通用语言是团队共享的语言，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。自然地，**领域专家对通用语言有很大的影响**，因为他最了解业务。



- 通用语言更多地是关于业务本身如何思考和运作的。它就像其他语言一样，也会随着时间推移而**不断演化改变**。
- 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。限界上下文和通用语言是**一对一的关系**。
- 限界上下文是一个相对**较小的概念**，它刚好能够容纳下一个独立的业务领域锁使用的通用语言。
- 当多个限界上下文需要“打交道”时，我们可以通过**上下文映射图**对这些限界上下文进行集成。
- 我们以“护士给病人注射标准剂量的流感疫苗”为例，看看如何通过代码可以表现出实际的业务流程：

代码编写方式	可能的业务描述	生成的代码

代码和业务脱节，只是“写代码”	“管它业务是什么，代码这东西，写就完事儿了”	patient.setShotType(TYPE_FLU); // 给病人设置针剂种类
		patient.setDose(dose); // 给病人分配剂量
		patient.setNurse(nurse); // 给病人分配护士

太笼统，丢失业务细节和概念	“我们给病人注射了流感疫苗”	patient.giveFluShot();

代码可以表现业务，Good!	“护士给病人注射标准剂量的流感疫苗”	Vaccine vaccine = vaccines.standardAdultFluDose(); // 获得标准剂量流感疫苗
		nurse.injectFluVaccine(patient, vaccine); // 护士给病人注射疫苗

- 你该如何掌握通用语言呢?

> - 同时绘制**物理模型图**和**概念模型图**，并标以名字和行为。
> - 创建一个包含**简单定义的术语表**。包括好的和不好的，并注明好与不好的原因。
> - 此后，改进之后的通用语言将**反映到系统的源代码中**。
> - 由于团队交流和代码才是对通用语言的持续表达，你应该试着**抛弃那些模型图、术语表和文档**。虽然这并不是DDD所要求的，但是这样做的确很实用，因为我们**很难将项目文档和软件系统保持同步**。

- 有了以上的认知，我们便可以重新设计saveCustomer()方法了，将其修改后能够反映出他所支持的业务操作，即：通过代码来反映业务。

```
public interface Customer {
        public void changePersonalName(String firstName, String lastName);
        public void postalAddress(PostalAddress postalAddress);
        public void relocateTo(PostalAddress changedPostalAddress);
        public void changeHomeTelephone(Telephone telephone);
        public void disconnectHomeTelephone();
        public void changeMobileTelephone(Telephone telephone);
        public void disconnectMobileTelephone();
        public void primaryEmailAddress(EmailAddress emailAddress);
        public void secondaryEmailAddress(EmailAddress emailAddress);
    }
```

【解释】

> 通过上面的方式，我们可以体现出Customer是支持哪些业务操作的。而哪些业务操作它是不支持的。

- 另外，我们还需要知道，对领域模型的修改也将导致对应用层的修改。**每一个应用层的方法都对应着一个单一的用例流**，而不像上面的saveCustomer方法那样，我们使用了同一个方法来处理多个用例流。如下所示：

```
@Transactional
public void changeCustomerPersonalName(String customerId, String customerFirstName, String customerLastName){
    Customer customer = customerRepository.customerOfId(customerId);
    if (customer == null){
        throw new IllegalStateException ("Customer does not exist.");
    }
    customer.changePersonalName(customerFirstName, customerLastName);
}
```

四、使用DDD的业务价值/如何向老板推销DDD

- 无论我们使用什么技术，目的都应该是**提供业务价值**。如果我们提供的技术方案比其他方案更能够产生业务价值，那么我们的业务能力也将增强。下面我们就来看看DDD所带来的一些非常理想化的业务价值。
- 1> 你获得了一个非常有用的领域模型

> DDD**强调将精力花在对业务最有价值的东西上**。我们并不过度建模，而是**关注业务的核心域**。当我们将关注点放在自己的业务和别人业务的区别上时，我们便能更好地理解自己的任务所在，同时我们将更具竞争优势。

- 2> 你的业务得到了更精确的定义和理解

> 随着业务模型的不断改善，人们对业务的理解也将更加深刻。**在团队讨论的过程中，一些业务细节被不断地暴露出来，这些细节有助于掌握业务价值**。

- 3> 领域专家可以为软件设计做出贡献

> 不同的领域专家由于彼此工作经验不同，工作经历也不同，对领域的认知也会出现分歧，但是，当领域专家们作为一个团队成员时，他们最终还是会达成一致意见，并且开发者和领域专家共同使用一套通用语言进行交流，**领域专家将业务知识传递给开发者**。即使开发者离职或转岗，**此时培训和工作交接也将变得更加简单**——领域专家、剩下的开发者、新成员都可以继续使用通用语言进行交流。

- 4> 更好的用户体验

> 用户体验可以更好的反映出领域模型的好坏，当用户需要经过培训去理解和使用产品的时候，说明产品本身的设计是有问题的。
> 当用户体验是按照领域专家心中的模型来设计时，**软件本身便能对用户起到培训作用，而不需要业务人员来提供培训**。效率提高了，培训减少了——这就是业务价值。

- 5> 清晰的模型边界

> 通过深入的理解项目的限界上下文，从而可以明确目标而随之产生高效的解决方案，而不是技术团队将精力单纯地放在编码和算法上。

- 6> 更好的企业架构

> 将不同的领域业务拆分为不同的**限界上下文**，并通过**上下文映射图**来集成不同的限界上下文，我们可以更全面的了解整个企业的架构。

- 7> 敏捷、迭代式和持续建模

> DDD并不是画模型图，而是**将领域专家的思维模型转化成有用的业务模型**。更不是创建一个真实世界的模型，而是模仿现实。

- 8> 使用战略和战术新工具

> **限界上下文**为团队创建了一个建模边界。在单个限界上下文中团队成员共享一套**通用语言**。不同的团队有时各自负责一个限界上下文，此时可以使用**上下文映射图**在战略层面上对限界上下文进行界分和集成。在某个建模边界内部，团队将使用战术建模工具，如：
>
> - **聚合**（Aggregate）
> - **实体**（Entity）
> - **值对象**（Value Object）
> - **领域服务**（Domain Service）
> - **领域事件**（Domain Event）
> - ……

五、实施DDD所面临的常见挑战

5.1> 通用语言方面的挑战

- 我们需要花费大量的时间和精力去思考业务价值、研究业务概念并和领域专家交流沟通。在一次次的重建、推倒、再重建的磨砺中提炼通用语言和核心业务价值。

5.2> 领域专家方面的挑战

- 引入领域专家也绝非易事，但是无论多么困难，这都是我们必须要做的。如果我们找不到领域专家，我们根本无法对这个领域有深入的理解。
- 当找到领域专家之后，开发人员应该主动与领域专家交流并仔细聆听，然后将你们的谈话转化成软件代码。

5.3> 开发人员方面的挑战

- 作为开发者，技术思想是必须要具备的，但是关于业务层面的思想，往往是很欠缺的。在开发过程中，我们要切忌沉迷在技术上面的炫技，而是要着重思考如何通过技术而为业务服务。

六、领域对象开发流程

- 在你开发一个新的领域对象时，比如实例或值对象，你可以采用如下步骤进行：

> - 首先，编写**测试代码**以模拟客户代码是如何使用该领域对象。（在开发人员的帮助下，领域专家通过阅读测试代码来检验领域对象是否满足业务需求）
> - 其次，创建该**领域对象**以使测试代码能够编译通过。
> - 第三，同时对测试和领域对象**进行重构**，直到测试代码能够正确地模拟客户代码，同时领域对象拥有能够表明业务行为的方法签名。

> - 第四，实现领域对象的行为，直到测试通过为止，再对实现代码进行重构。
> - 最后，向你的团队成员展示代码，包括领域专家，以保证领域对象能够正确地反映通用语言。

吾尝终日而思矣，不如须臾之所学也；
吾尝跂而望矣，不如登高之博见也。
登高而招，臂非加长也，而见者远；
顺风而呼，声非加疾也，而闻者彰。
假舆马者，非利足也，而致千里；
假舟楫者，非能水也，而绝江河。
君子生非异也，善假于物也。
------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“爪哇缪斯” ~\(^o^)/~ 「干货分享，每天更新」