# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/043\_在架构图合并简化后的开发流程介绍/笔记.docx**

《043\_订单系统的业务架构设计》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计 -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

两个架构图就可以了：业务架构+技术架构，完全画到一个图里去，在一个图里，展现出来你的系统所具备的所有的服务，业务模块，业务功能，微服务框架、中间件系统、基础设施、API网关、前端、安全架构（二级域名、HTTPS、DDoS、云防火墙、SLB），系统架构图，业务，技术

一张图，可以轻而易举的看到你的系统是长什么样子的

业内，一般来说，都是把业务架构+技术架构，合并为一张图的

生产架构图，网络拓扑图，服务器、SLB、网络交换机、网线、机房、数据库、Web Server、缓存，生产架构的拓扑图，画出来 -> 生产机器文档，各个服务的服务器、中间件的服务器，所有生产环境的机器列表都写一个文档出来

文档里分为多个环境：开发环境、联调环境、QA测试环境、预发布环境、生产环境，每个环境里这个系统部署的时候，服务器的地址，中间件的地址，全部都写清楚，域名，端口号，如何访问，各个环境的地址都写清楚

详细设计（领域模型、数据库表、接口、接口的运行流程）

代码

各个环境的测试和部署

迭代需求

上述各个环节，在每次需求迭代的时候，全部是保持一致的

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/040\_DDD业务建模的回顾/笔记.docx**

3616118202

《040\_订单系统的业务架构设计（1）》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计 -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/042\_三架构图合并为一张架构图和一份机器文档的介绍/笔记.docx**

3616118202

《042\_订单系统的业务架构设计》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计 -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

两个架构图就可以了：业务架构+技术架构，完全画到一个图里去，在一个图里，展现出来你的系统所具备的所有的服务，业务模块，业务功能，微服务框架、中间件系统、基础设施、API网关、前端、安全架构（二级域名、HTTPS、DDoS、云防火墙、SLB），系统架构图，业务，技术

一张图，可以轻而易举的看到你的系统是长什么样子的

业内，一般来说，都是把业务架构+技术架构，合并为一张图的

生产架构图，网络拓扑图，服务器、SLB、网络交换机、网线、机房、数据库、Web Server、缓存，生产架构的拓扑图，画出来 -> 生产机器文档，各个服务的服务器、中间件的服务器，所有生产环境的机器列表都写一个文档出来

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/046\_订单系统的系统架构图设计（3）/笔记.docx**

3616118202

《046\_订单系统的系统架构图设计（3）》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计（DDD） -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

两个架构图就可以了：业务架构+技术架构，完全画到一个图里去，在一个图里，展现出来你的系统所具备的所有的服务，业务模块，业务功能，微服务框架、中间件系统、基础设施、API网关、前端、安全架构（二级域名、HTTPS、DDoS、云防火墙、SLB），系统架构图，业务，技术

一张图，可以轻而易举的看到你的系统是长什么样子的

业内，一般来说，都是把业务架构+技术架构，合并为一张图的

生产架构图，网络拓扑图，服务器、SLB、网络交换机、网线、机房、数据库、Web Server、缓存，生产架构的拓扑图，画出来 -> 生产机器文档，各个服务的服务器、中间件的服务器，所有生产环境的机器列表都写一个文档出来

文档里分为多个环境：开发环境、联调环境、QA测试环境、预发布环境、生产环境，每个环境里这个系统部署的时候，服务器的地址，中间件的地址，全部都写清楚，域名，端口号，如何访问，各个环境的地址都写清楚

详细设计（领域模型、数据库表、接口、接口的运行流程）

代码

各个环境的测试和部署

迭代需求

上述各个环节，在每次需求迭代的时候，全部是保持一致的

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/045\_订单系统的系统架构图设计（2）/笔记.docx**

3616118202

《045\_订单系统的系统架构图设计（2）》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计（DDD） -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

两个架构图就可以了：业务架构+技术架构，完全画到一个图里去，在一个图里，展现出来你的系统所具备的所有的服务，业务模块，业务功能，微服务框架、中间件系统、基础设施、API网关、前端、安全架构（二级域名、HTTPS、DDoS、云防火墙、SLB），系统架构图，业务，技术

一张图，可以轻而易举的看到你的系统是长什么样子的

业内，一般来说，都是把业务架构+技术架构，合并为一张图的

生产架构图，网络拓扑图，服务器、SLB、网络交换机、网线、机房、数据库、Web Server、缓存，生产架构的拓扑图，画出来 -> 生产机器文档，各个服务的服务器、中间件的服务器，所有生产环境的机器列表都写一个文档出来

文档里分为多个环境：开发环境、联调环境、QA测试环境、预发布环境、生产环境，每个环境里这个系统部署的时候，服务器的地址，中间件的地址，全部都写清楚，域名，端口号，如何访问，各个环境的地址都写清楚

详细设计（领域模型、数据库表、接口、接口的运行流程）

代码

各个环境的测试和部署

迭代需求

上述各个环节，在每次需求迭代的时候，全部是保持一致的

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/044\_订单系统的系统架构图设计/笔记.docx**

3616118202

《044\_订单系统的系统架构图设计》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计（DDD） -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

两个架构图就可以了：业务架构+技术架构，完全画到一个图里去，在一个图里，展现出来你的系统所具备的所有的服务，业务模块，业务功能，微服务框架、中间件系统、基础设施、API网关、前端、安全架构（二级域名、HTTPS、DDoS、云防火墙、SLB），系统架构图，业务，技术

一张图，可以轻而易举的看到你的系统是长什么样子的

业内，一般来说，都是把业务架构+技术架构，合并为一张图的

生产架构图，网络拓扑图，服务器、SLB、网络交换机、网线、机房、数据库、Web Server、缓存，生产架构的拓扑图，画出来 -> 生产机器文档，各个服务的服务器、中间件的服务器，所有生产环境的机器列表都写一个文档出来

文档里分为多个环境：开发环境、联调环境、QA测试环境、预发布环境、生产环境，每个环境里这个系统部署的时候，服务器的地址，中间件的地址，全部都写清楚，域名，端口号，如何访问，各个环境的地址都写清楚

详细设计（领域模型、数据库表、接口、接口的运行流程）

代码

各个环境的测试和部署

迭代需求

上述各个环节，在每次需求迭代的时候，全部是保持一致的

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/041\_对系统进行技术设计的三架构的讲解/笔记.docx**

《041\_订单系统的业务架构设计》

需求分析 -> 业务建模 -> 架构设计/概要设计 -> 详细设计 -> 开发 -> 测试 -> 部署 -> 监控 -> 维护

业务建模：DDD战略建模，领域的识别 -> 子域的分拆 -> （模块的划分） -> 通用语言的设计 -> 上下文的映射

站在的工程师的角度，对业务进行了建模，就完全知道你未来要实现哪些业务功能，有哪些业务模块，如何进行业务协作，不同的子域之间如何集成和协作，最终呈现出来的，提供给终端用户的是一个什么样的东西

业务建模完成以后的产出成果：子域的模块划分文档、子域的通用语言文档、业务流程图、上下文映射图

技术层面的设计：业务架构、技术架构、生产架构

业务架构：完全就可以从业务建模的成果文档中进行一个提炼，就是纯粹从业务功能的角度去考虑你的系统应该划分为哪些模块，包含哪些功能，跟其他的业务系统是如何进行对接配合的，抛弃掉具体的技术，就纯粹是业务层面来展现你的系统的架构

技术架构：里面不用写具体的业务功能，系统，服务，跟其他的系统如何交互，用了哪些开发框架，以及哪些中间件系统，有哪些具体的技术方案，技术架构有哪些设计

生产架构：具体部署之后，系统如何部署，服务如何部署，部署几台机器，机器的配置，中间件需要的配置，域名，安全，DNS，HTTPS，防火墙，SLB，要考虑一系列的生产环境是如何进行部署的

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/027\_采用DDD思想为订单域设计一套通用语言（7）/笔记.docx**

3616118202

《027\_采用DDD思想为订单域设计一套通用语言（7）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/024\_采用DDD思想为订单域设计一套通用语言（4）/笔记.docx**

《024\_采用DDD思想为订单域设计一套通用语言（4）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目对应商品的原价：Price

待确认订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/023\_采用DDD思想为订单域设计一套通用语言（3）/笔记.docx**

3616118202

《023\_采用DDD思想为订单域设计一套通用语言（3）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

计算待确认订单的优惠价格：CalculateOrderPromotionPrice

待确认订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的优惠价格：OrderPromotionPrice

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/029\_采用DDD思想为订单域设计一套通用语言（9）/笔记.docx**

3616118202

《029\_采用DDD思想为订单域设计一套通用语言（9）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

订单履约系统：OrderFulfillmentSystem

履约订单：fulfilOrder

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/026\_采用DDD思想为订单域设计一套通用语言（6）/笔记.docx**

《026\_采用DDD思想为订单域设计一套通用语言（6）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目对应商品的原价：Price

待确认订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/025\_采用DDD思想为订单域设计一套通用语言（5）/笔记.docx**

《025\_采用DDD思想为订单域设计一套通用语言（5）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目对应商品的原价：Price

待确认订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

订单条目：OrderItem

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/028\_采用DDD思想为订单域设计一套通用语言（8）/笔记.docx**

《028\_采用DDD思想为订单域设计一套通用语言（8）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

订单履约系统：OrderFulfillmentSystem

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/022\_采用DDD思想为订单域设计一套通用语言（2）/笔记.docx**

《022\_采用DDD思想为订单域设计一套通用语言（2）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/021~029资料/021\_采用DDD思想为订单域设计一套通用语言（1）/笔记.docx**

3616118202

《021\_采用DDD思想为订单域设计一套通用语言（1）》

DDD落地实战的过程都梳理了一遍，当我们进行一个DDD落地实战的时候，大部分的同学手头能够上的项目仅仅是你们公司的一个业务领域里的一个子域里的几个module，应该是你负责的一两个服务跟其他人负责的几个服务加起来，才能形成一个完整的系统，属于这个业务领域里的一个子域

技术积累、架构设计、代码落地、平台实战（额外付费）

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/037\_为订单域进行DDD上下文映射（8）/笔记.docx**

3616118202

《037\_为订单域进行DDD上下文映射（8）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了，阿里云，每个产品都会提供API出来，但是你一定是他的尊奉者

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/034\_为订单域进行DDD上下文映射（5）/笔记.docx**

3616118202

《034\_为订单域进行DDD上下文映射（5）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/035\_为订单域进行DDD上下文映射（6）/笔记.docx**

《035\_为订单域进行DDD上下文映射（6）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/030\_为订单域进行DDD上下文映射（1）/笔记.docx**

3616118202

《030\_为订单域进行DDD上下文映射（1）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游是D，上游是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/039\_为订单域进行DDD上下文映射（10）/笔记.docx**

《039\_为订单域进行DDD上下文映射（10）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了，阿里云，每个产品都会提供API出来，但是你一定是他的尊奉者

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON、HTML之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

除了基于HTTP RESTful接口+JSON数据格式来进行实现，还有一种实现方式就是基于消息机制来实现，RocketMQ/Kafkaba/RabbitMQ发布消息，消息的格式就是JSON，此时他也是一种开放主机服务，JSON就是他的发布语言

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

基于业务领域、子域（module）、通用语言、上下文映射图，把系统的业务架构设计出来，技术架构，生产架构

下下周开始，进入具体的订单系统内部的领域建模了，涉及到大量的DDD的知识

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/038\_为订单域进行DDD上下文映射（9）/笔记.docx**

《038\_为订单域进行DDD上下文映射（9）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了，阿里云，每个产品都会提供API出来，但是你一定是他的尊奉者

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON、HTML之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

除了基于HTTP RESTful接口+JSON数据格式来进行实现，还有一种实现方式就是基于消息机制来实现，RocketMQ/Kafkaba/RabbitMQ发布消息，消息的格式就是JSON，此时他也是一种开放主机服务，JSON就是他的发布语言

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/031\_为订单域进行DDD上下文映射（2）/笔记.docx**

3616118202

《031\_为订单域进行DDD上下文映射（2）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游是D，上游是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/036\_为订单域进行DDD上下文映射（7）/笔记.docx**

《036\_为订单域进行DDD上下文映射（7）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了，阿里云，每个产品都会提供API出来，但是你一定是他的尊奉者

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/033\_为订单域进行DDD上下文映射（4）/笔记.docx**

《033\_为订单域进行DDD上下文映射（4）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游（Downstream）是D，上游（Upstream）是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方（Customer-Supplier）的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/032\_为订单域进行DDD上下文映射（3）/笔记.docx**

3616118202

《032\_为订单域进行DDD上下文映射（3）》

公司的一块业务，整个就应该是一个业务领域，自己负责的系统是里面的哪个子域，可能你负责的服务/系统仅仅是子域里的一部分而已，更小的公司，他自己就一个业务领域，就几个子域

需求分析 -> 理解业务 -> 业务领域 -> 各个子域（module） -> 自己子域的通用语言（你应该跟子域里的其他人一起合作） -> 上下文映射（上下游关系）

module：识别出来一个子域里的业务模块，创建订单模块、订单支付模块、订单取消模块、订单退款模块，再说吧，要把一个订单系统拆分为微服务的话，下单服务、订单支付服务、订单调度服务、订单售后服务

上下文的映射，就是梳理和确定你的这个域和其他域之间的关系，包括上下游的关系，协作关系，在这个环节应该是没有技术问题介入的，都是业务上的问题

在完成了需求分析以及理解完业务之后，战略建模，业务建模，顶层建模，代表的是你作为一个技术，理解完了业务以后，用DDD战略建模作为一个工具手段，表达出来你对业务的理解

上下游的关系就是说：下游去调用上游，上游是提供接口的，一般是这么个意思，下游是D，上游是U

合作关系（Partnership）：你和另外一个系统是紧密耦合的，你们俩必须一起合作，一起开发，一起实现某个业务需求，一起成功，一起失败，一荣俱荣，一损俱损，如果有需求变更，必须要一起配合调整接口和逻辑

客户方-供应方的映射关系，就是说你可以跟你要调用的其他系统进行协商，让他们提供你需要的API，这个事儿他们可能会答应，可能不会答应，有协商的余地，这就是客户方-供应方的映射关系

尊奉者关系，就是说，你要调用的系统绝对是不可能为你开发任何的新接口的，他们就暴露这些接口出来，你爱用不用，这时候你就得尊奉他了

其实我们的电商系统里，你订单域和库存域、履约域，都是这样子强耦合的，必须一起配合完成电商的主业务逻辑，包括下单、锁定库存、执行履约，等等

但是和支付域，会员域这些域本质都是通过领域事件进行集成的，比较的松耦合，支付域发布订单支付事件，订单域处理，会员域处理，这里都不用协商，也不涉及调用，完全是松耦合的，说白了就是，人家发布什么事件，你就只能收听到什么事件，你不能去提要求，那就是尊奉者关系了

但是如果你能跟人家商量商量，意思是说你发生啥啥的时候，能不能发布个什么事件，那就是客户方-供应方的关系了，不过一般对于我们的场景来说，那些事件都是固定的，你就是固定消费就行了，不需要去协商，所以就是尊奉者的关系

但是订单域和库存域、履约域，必须是合作关系，强耦合，因为要一起实现电商里的下单和支付两个核心业务流程

共享内核（Shared Kernel）：两个子域之间共同引用一个jar包，里面有共享的模型，模型修改必须两者一致同意，这种一般不会发生，发生也是因为遗留原因

防腐层（Anticorruption Layer，ACL）：如果你是合作、客户方-供应方、共享内核之类的关系，一般大家都是有商量的，一般集成的时候从一个子域的模型转换到自己的子域的模型，这个叫做翻译，一般是比较容易的，但是如果是尊奉者之类的关系，下游系统得做一个防腐层，把上游系统提供的模型进行翻译，避免你跟他耦合在一起，就可以防腐

开放主机服务（Open Host Service，OHS）和发布语言（Published Language，PL，类似XML、JSON之类的）：这个一般都是指的对内的话，就是一些中间件，或者一些通用的系统，直接提供开放的HTTP协议的RESTful接口，你直接调用就行了；对外的话，一般就是一个系统暴露出去的开放接口，现在一般来说，你用一些SaaS类的系统，比如知识付费了，物流快递了，电子签名了，类似这种系统，都是SaaS的，都会提供开放接口出来给你调用，一般跟防腐层一起用

隔离模式（Separate Way）：两个子域完全没关系，就是隔离开来的

大泥球（Big Ball of Mud）：多个子域的模型混合再一起，这就是大泥球

这里要说的一点是，一般基于消息的解耦，都是OHS+PL，他基于发布消息形成了一个开放主机服务，至于消息的具体数据格式，就是PL，一般是xml或者json，都行，也可能是序列化过的PB、HTML之类的，反正就是数据格式

他发布出来消息，然后下游的人就可以接收到了，做一个ACL防腐层就行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/014\_再谈DDD的各种域以及如何落地实战（4）/笔记.docx**

3616118202

《014\_再谈DDD的各种域以及如何落地实战（4）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/012\_再谈DDD的各种域以及如何落地实战（2）/笔记.docx**

《012\_再谈DDD的各种域以及如何落地实战（2）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块，然后拆分为N个服务交给不同的P6去负责

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/018\_再谈DDD的各种域以及如何落地实战（8）/笔记.docx**

3616118202

《018\_再谈DDD的各种域以及如何落地实战（8）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

你最好是站在P9级的一个视角，甚至是P10级的视角，识别出来你所在的是公司的哪个业务，领域，你自己应该去进行一个领域里的子域划分，每个子域的类型是什么；你是负责哪个子域的，子域内的限界上下文的通用语言都有哪些，上下文的映射，具体的细化的领域对象级的建模是如何来做的

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就应该做上述的事情，自己划分领域内的子域以及子域的类型，需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言，再去设计对应的上下文的映射关系，我认为，这些东西应该是代表了技术对业务的一种理解，以及技术和业务人员之间达成的共识

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构（从业务角度描述你的系统架构，包含哪些业务模块，每个业务模块里有哪些业务功能，还有业务流程）和技术架构（服务拆分、技术框架、中间件、基础平台、第三方系统、数据库、Web容器，包含了很多技术方案），包括生产架构（需要多少机器，机器的配置，中间件的配置和容量，负载均衡，域名，安全，跟第三方系统的对接）

设计完了，就可以进行技术评审，过一遍上述的那套东西，确保都是按照通用语言来走的，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/016\_再谈DDD的各种域以及如何落地实战（6）/笔记.docx**

3616118202

《016\_再谈DDD的各种域以及如何落地实战（6）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

你最好是站在P9级的一个视角，甚至是P10级的视角，识别出来你所在的是公司的哪个业务，领域，你自己应该去进行一个领域里的子域划分，每个子域的类型是什么；你是负责哪个子域的，子域内的限界上下文的通用语言都有哪些，上下文的映射，具体的细化的领域对象级的建模是如何来做的

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就应该做上述的事情，自己划分领域内的子域以及子域的类型，需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言，再去设计对应的上下文的映射关系，我认为，这些东西应该是代表了技术对业务的一种理解，以及技术和业务人员之间达成的共识

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/015\_再谈DDD的各种域以及如何落地实战（5）/笔记.docx**

《015\_再谈DDD的各种域以及如何落地实战（5）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/011\_再谈DDD的各种域以及如何落地实战（1）/笔记.docx**

《011\_再谈DDD的各种域以及如何落地实战（1）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个业务领域里的一个模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块，然后拆分为N个服务交给不同的P6去负责

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/017\_再谈DDD的各种域以及如何落地实战（7）/笔记.docx**

《017\_再谈DDD的各种域以及如何落地实战（7）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

你最好是站在P9级的一个视角，甚至是P10级的视角，识别出来你所在的是公司的哪个业务，领域，你自己应该去进行一个领域里的子域划分，每个子域的类型是什么；你是负责哪个子域的，子域内的限界上下文的通用语言都有哪些，上下文的映射，具体的细化的领域对象级的建模是如何来做的

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就应该做上述的事情，自己划分领域内的子域以及子域的类型，需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言，再去设计对应的上下文的映射关系，我认为，这些东西应该是代表了技术对业务的一种理解，以及技术和业务人员之间达成的共识

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构（从业务角度描述你的系统架构，包含哪些业务模块，每个业务模块里有哪些业务功能）和技术架构了，包括生产架构

完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/020\_两个新概念：问题空间和解决方案空间/笔记.docx**

3616118202

《020\_两个新概念：问题空间和解决方案空间》

对DDD应该是有了一个完整的体系化的认知，结合我们的软件工程的流程和规范，是如何融入进去的，如何落地到你的项目开发里去，大家应该都很清晰明了了，包括你自己做DDD的实战应该去怎么思考，怎么做，出去面试应该怎么说

问题空间指的是你要解决的核心业务问题，比如说如何让用户通过互联网买到商品？这就是一个核心的问题空间，往往一个问题空间会包含多个子域；还有一个解决方案空间，是这样的一个概念，比如你有了很多子域，然后针对子域会设计限界上下文，里面有对应的通用语言，领域对象模型

这个大家没问题吧？

所以假设问题空间是包含了多个子域，那么解决方案空间核心是包含了多个限界上下文，因为限界上下文里的通用语言、领域对象模型，都是用于解决业务问题的，所以解决方案空间里会包含多个限界上下文，就这么个意思

因为有的时候一个子域会包含多个限界上下文，一个限界上下文是圈定了一套有业务边界的通用语言和领域模型，也可能一个限界上下文居然横跨了多个子域，但是这都非常非常的不好，我再说一遍，绝对不要这么搞

下周的任务：引入订单系统项目的业务背景，确定子域，设计订单上下文中的通用语言，假设是一个小型电商公司，订单域就是一个子域，就你带了几个人来负责，设计订单域里的通用语言

下下周的内容：深入学习上下文映射，考虑你的子域如何跟其他子域进行映射，技术上如何进行上下文集成

过一遍订单域的业务架构、技术架构、生产架构，模拟设计一遍

对订单域进行实体、值对象、领域事件、领域服务、资源库、工厂、用户界面、应用服务、数据库、接口的具体的设计，代码级落地来走一下，结合真实的电商云平台去进行DDD里的业务细节的完善，给大家提供真实的平台和接口，保证你的真实的DDD建模的订单系统可以部署到平台去运行，让你看到真实的效果

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/013\_再谈DDD的各种域以及如何落地实战（3）/笔记.docx**

《013\_再谈DDD的各种域以及如何落地实战（3）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构和技术架构了，包括生产架构，完成技术评审，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/011~020资料/019\_再谈DDD的各种域以及如何落地实战（9）/笔记.docx**

3616118202

《019\_再谈DDD的各种域以及如何落地实战（9）》

领域、子域（核心域、支撑子域、通用子域）、限界上下文、通用语言、上下文映射，战略建模的内容，领域建模，战术建模（实体、值对象、领域事件、领域服务、资源库、模块、应用服务、用户界面），领域对象建模

之前讲过这些域的概念，其实对于这些域，并没有一个特别固定死的定义，因为根据不同情况，他们可能都有所不同，但是我推荐的比较好的一个划分方式是这样的，首先，领域是什么？领域应该是你们公司的一块业务，就是一个完整的业务

举个例子，有一个巨头公司，他现在有B2C的电商业务，B2B的电商业务，C2C的电商业务，还有社区团购业务，生鲜电商业务，外卖业务，团购到店业务，等等，假设一个巨头有这么多的业务，那么往往建议说是这样子的，就是公司的某一块具体的业务，你可以认为他就是一个领域

比如说一些FM音频类的APP，音频业务、陪玩业务、社交业务、电商业务、知识付费，任何公司只要具备一定的规模，其实都会包含多种业务，公司里的一个业务，就把他确定为一个领域

所以说，针对任何一个公司，如果你要确定这个公司有哪些领域，就直接看他有几块业务，通常来说建议是一块业务就对应了一个领域

软件公司，智慧城市、智慧交通、智慧物流、金融科技，每一块业务就属于一个领域

如果说一个公司只有一块业务，一般这个公司的总人数肯定再100人以内，甚至就是几十个人的一个规模，起步阶段的小型创业公司，他就只有一个领域

然后在领域里可以继续具体的细分，拆分为很多的子域，每一个子域都是这个领域里的一个业务模块，比如B2C里就包含了店铺、商家、用户、商品、订单、库存、物流、营销、客服、支付、财务、权限等各个子域，其他的业务领域也都是同理的

一个领域到底可以拆分为哪些子域呢？这个不好直接下定义，不过一般来说，对于任何一个领域，往往拆分子域都是很容易的，很容易就可以识别出来这个领域内部包含了哪些业务部门，业务功能，业务模块，业务范围

子域又可以进一步具体的划分不同的类型，分为核心域、支撑子域、通用子域

核心域，我们认为就是支撑这块业务核心业务流程的子域；支撑子域，就是支撑业务流程，但是主要是辅助的，没那么重要的一个子域，可能是自己开发的，也可能是第三方采购来的，比如财务和客服；通用子域，很多时候可能是第三方采购来的，当然也可能是自己开发的，通用子域往往具有普遍性，可能适用于全公司，比如说OA、权限之类的

比如某大厂的一个P8或者是P9级的，团队里的大Leadre，会吩咐手下的P7或者P6说，你们有时间的话，尽可能去探索一下DDD，自己把DDD运用到自己的系统里去，手下的人就纷纷开始自己去探索

所以我之前强调了N多遍，一般DDD正确的玩法，应该是从P8以上，甚至P9级的大技术leader，针对公司的某一块业务，从上往下开始推DDD，因为划分子域一定是从这个级别开始进行的，P9级的业务线大领导对领域进行子域的划分

他自己负责的那块业务，就可以认为是一个领域，然后里面拆分N多子域，不同的子域就对应了一个业务模块，然后针对子域再进行类型的划分，看谁是核心域，谁是支撑子域，谁是通用子域

然后再把这些子域交给不同的团队，每个团队负责N个子域，最后每个子域交给一个P6去负责，正常来说应该是这样的，也有时候一个子域交给一个P7，P7再在子域内部划分不同的业务模块（module），然后拆分为N个服务交给不同的P6去负责

引入了一个新的DDD概念，module，模块，子域可以在逻辑上继续细分为多个module

交易域，独立的子域，但是负责交易域的可能有100个人，是一个P9在带，此时你可以考虑在这个子域里继续拆分成N多个module，可能会拆成20~30个module，每个module都对应了一个服务，把服务分配给底下的各个team，每个team拿到一些module，服务去维护，这就是微服务的概念

子域对应的限界上下文的通用语言的建模和设计，往往得是这个子域的负责人去推动和设计，子域的划分往往是P9级的人去负责和推动的，子域的限界上下文和通用语言的建模，上下文之间的映射和集成，往往是P7~P8级的人去负责和推动的

P6拿到自己负责的子域里的module，也就是服务，进行细化的领域对象级的建模，完事儿以后就按照建模去开发代码

所以说，一般来讲，如果你就是一个P7或者P6，你要玩儿DDD，最好的情况下，就是强行将你手头负责的一个或者多个系统/服务，认为他们每个都是所属大的业务领域里的一个或者几个子域，同时明确你的这些子域站在整个业务领域的角度来说，是支撑核心业务流程的核心域，还是非核心锦上添花的支撑子域，或者是支撑全公司所有业务的通用子域

你最好是站在P9级的一个视角，甚至是P10级的视角，识别出来你所在的是公司的哪个业务，领域，你自己应该去进行一个领域里的子域划分，每个子域的类型是什么；你是负责哪个子域的，子域内的限界上下文的通用语言都有哪些，上下文的映射，具体的细化的领域对象级的建模是如何来做的

然后一般对你们来说，往往都是默认自己手头负责的这个系统，或者几个系统，直接就是一个子域，然后就是按照上周强调的，在需求评审完毕之后，你就应该做上述的事情，自己划分领域内的子域以及子域的类型，需要针对你自己的这个子域对应的限界上下文去找产品、运营、用户达成一个共识的通用语言，再去设计对应的上下文的映射关系，我认为，这些东西应该是代表了技术对业务的一种理解，以及技术和业务人员之间达成的共识

一般我的建议是一个子域就对应一个限界上下文，这是最好的一种做法，所以正是因为这样，通常来说不建议子域划分的过于小，因为划分的过于小会导致每个子域都有一套通用语言，这也不太合适，举个例子，假设你每个子域就是一个P6带一个小弟来负责，那么等于是就这个P6和小弟设计了一套通用语言，别人都不知道，也不是太合适

往往我比较倾向于建议的是一个P7带的一个10人到20人的团队负责了一个或者多个子域，然后一个子域起码是有好几个人都在里面参与和工作的，他们几个人可以一起为子域确定一个限界上下文，然后设定一套通用语言

也许你一个子域有好几个人在里面一起负责，但是其实一个子域完全是可以拆分为多个服务的，每个人负责一个服务这种，都是没问题的

当然，其实在DDD理论里，其实有时候一个子域可能会对应多个限界上下文，也可能一个限界上下文会横跨多个子域，其实我觉得这都不太好，容易导致一定的混乱，最简单清晰的方式，就是一个子域就一个限界上下文，在里面设计一套通用语言

完事儿了再考虑你的这个所谓的子域如何对接其他人负责的系统，之所以说是系统，是因为其他人可能压根儿就没用DDD！你只能默认认为他们是一个一个的子域，大家人家根本没按DDD那套来，你可能暴露出去的还是以往的普通接口

然后通用语言和上下文映射设计好之后，就可以基于通用语言这套模型去设计业务架构（从业务角度描述你的系统架构，包含哪些业务模块，每个业务模块里有哪些业务功能，还有业务流程）和技术架构（服务拆分、技术框架、中间件、基础平台、第三方系统、数据库、Web容器，包含了很多技术方案），包括生产架构（需要多少机器，机器的配置，中间件的配置和容量，负载均衡，域名，安全，跟第三方系统的对接）

设计完了，就可以进行技术评审，过一遍上述的那套东西，确保都是按照通用语言来走的，接着就是进行细化的详细技术设计了，此时就可以进行DDD战术建模完成细化的业务模型和流程的建模，以及接口，数据库的设计

此时的重点就是考虑如何在里面进行战术建模（更恰当的说法是，领域对象建模，把内部的领域建模做好就行了），当然稍微复杂点的情况来说，可能你可以考虑按照他的业务功能把他拆分为几个微服务，那么要考虑每个服务里包含哪些领域对象

一般来说，在业务架构里，就会根据DDD理论把一个子域划分为多个模块，module的概念，然后在技术架构里，一个module就是一个服务，所谓的微服务应该是对应着DDD里的一个module的概念

当然也可能是一个子域一个系统，多人协作，里面划分module，就每个人负责一个module，单块系统开发也可能的

然后应该每个人分配到一个module对应的 服务，设计服务对应的module里的领域对象模型，根据通用语言来设计，包括领域服务，领域事件，用户界面，应用服务，数据库表，接口定义，等等一系列的东西

完事儿了大家进行详细设计的评审，评审的时候可以参照系统的核心功能流程来走查，判断各个服务里的领域对象配合能否完成所有的业务流程，评审通过后，就可以进行具体的开发环节了

要是时间过于紧张，那就仅仅设计个接口，然后跳过详细设计和领域对象建模，直接进入coding，coding过程中进行领域对象建模，数据库设计，当然这样可能搞的才匆忙了，建议有时间最好还是把领域对象建模和详细设计做一下

但是有一个问题，作为一个团队的leader而言，很有必要的一点，就是你要亲自确保每个服务每个人的设计都是符合通用语言的，都是符合DDD标准的，有的人就没完全按照这个标准来

最后就是开发、测试、部署就行了

开发的时候，严格按照详细设计里的那些东西写代码，代码基本是完全还原业务模型和业务流程的，和文档都是一一对应起来，极为的漂亮

严格的按照这套流程来，绝对不可能说是solo单打独斗，一定是团队里leader和资深的负责人一起共同把关每一个需求的设计和评审，确保每个人都是按照DDD这套东西在走的，甚至包括代码review都要去审查是否按照DDD来走

确保代码的质量

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/062\_再次梳理订单系统的业务模型和流程（7）/笔记.docx**

《062\_再次梳理订单系统的业务模型和流程（7）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，再加上运费，此时就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：UnconfirmedOrder

待确认订单条目：UnconfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：PromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：UnconfirmedOrderPrice

DDD：面向业务来进行开发

用户在订单确认页面上，对每个订单条目都可以选择不同的促销活动，此时可以通过订单系统的价格计算的功能，完成待确认订单的价格的计算，用户确认每个订单条目使用的促销活动以及订单条目的优惠价格，确认好收货地址以及计算好的运费，确认订单的总价格，以及发票抬头等可选的信息

接着可以点击提交订单的按钮，把确认好的订单提交到订单系统

订单系统接收到一个订单之后，此时首先会把订单以及订单条目都录入到数据库里去，同时设置订单的状态为“待付款“，接着会调用库存系统的锁定库存的功能，把该订单要购买的各个商品的库存都进行锁定

库存系统在锁定订单要购买的各个商品的库存的时候，可能会发现库存不足，此时有可能会返回库存不足的提示给订单系统，同时商品库存的锁定就会失败，订单系统就需要把之前录入的订单进行事务回滚，同时提醒用户提交订单失败，库存不足

提交订单：SubmitOrder

订单：Order

订单编号：OrderCode

用户编号：UserId

收件人：Consignee

收件地址：DeliveryAddress

联系电话：PhoneNumber

运费：Freight

订单原价：OrderAmount

订单优惠金额：DiscountAmount

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单状态：OrderStatus

订单条目：OrderItem

订单编号：OrderCode

商品SKU编号：SkuCode

促销活动编号：PromotionActivityCode

商品原价：Price

优惠价格：PromotionPrice

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

锁定库存成功：StockLocked

库存不足：StockNotEnough

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/057\_再次梳理订单系统的业务模型和流程（2）/笔记.docx**

3616118202

《057\_再次梳理订单系统的业务模型和流程（2）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/056\_再次梳理订单系统的业务模型和流程（1）/笔记.docx**

《056\_再次梳理订单系统的业务模型和流程（1）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

订单履约系统：OrderFulfillmentSystem

履约订单：fulfilOrder

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/059\_再次梳理订单系统的业务模型和流程（4）/笔记.docx**

3616118202

《059\_再次梳理订单系统的业务模型和流程（4）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，再加上运费，此时就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：UnconfirmedOrder

待确认订单条目：UnconfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：PromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：UnconfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/063\_再次梳理订单系统的业务模型和流程（8）/笔记.docx**

《063\_再次梳理订单系统的业务模型和流程（8）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，再加上运费，此时就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：UnconfirmedOrder

待确认订单条目：UnconfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：PromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：UnconfirmedOrderPrice

DDD：面向业务来进行开发

用户在订单确认页面上，对每个订单条目都可以选择不同的促销活动，此时可以通过订单系统的价格计算的功能，完成待确认订单的价格的计算，用户确认每个订单条目使用的促销活动以及订单条目的优惠价格，确认好收货地址以及计算好的运费，确认订单的总价格，以及发票抬头等可选的信息

接着可以点击提交订单的按钮，把确认好的订单提交到订单系统

订单系统接收到一个订单之后，此时首先会把订单以及订单条目都录入到数据库里去，同时设置订单的状态为“待付款“，接着会调用库存系统的锁定库存的功能，把该订单要购买的各个商品的库存都进行锁定

库存系统在锁定订单要购买的各个商品的库存的时候，可能会发现库存不足，此时有可能会返回库存不足的提示给订单系统，同时商品库存的锁定就会失败，订单系统就需要把之前录入的订单进行事务回滚，同时提醒用户提交订单失败，库存不足

提交订单：SubmitOrder

订单：Order

订单编号：OrderCode

用户编号：UserId

收件人：Consignee

收件地址：DeliveryAddress

联系电话：PhoneNumber

运费：Freight

订单原价：OrderAmount

订单优惠金额：DiscountAmount

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单状态：OrderStatus

订单条目：OrderItem

订单编号：OrderCode

商品SKU编号：SkuCode

促销活动编号：PromotionActivityCode

商品原价：Price

优惠价格：PromotionPrice

创建订单：CreateOrder

用户在提交订单之后，直接会跳转到支付界面，此时会进行支付，完成支付之后，第三方支付平台会发送回调的请求到支付系统，通知支付系统订单的支付已经完成了，支付系统会发布一个订单支付成功的事件，订单系统可以订阅和获取到这个事件

订单系统需要对订单的状态进行流转，将订单的状态修改为“待发货“，同时调用库存系统对订单的商品库存进行扣减，再调用履约系统，对订单的发货进行履约

如果没有立即进行支付，此时用户可以查询自己的订单列表，还可以查看订单具体的详情，也可以在这里对未支付的订单进行支付

查询订单列表：QueryOrderList

查看订单详情：ViewOrderDetails

库存系统：InventorySystem

锁定库存：LockStock

锁定库存成功：StockLocked

库存不足：StockNotEnough

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

履约系统：FulfillmentSystem

履约订单：fulfilOrder

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/060\_再次梳理订单系统的业务模型和流程（5）/笔记.docx**

《060\_再次梳理订单系统的业务模型和流程（5）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，再加上运费，此时就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：UnconfirmedOrder

待确认订单条目：UnconfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：PromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：UnconfirmedOrderPrice

DDD：面向业务来进行开发

用户在订单确认页面上，对每个订单条目都可以选择不同的促销活动，此时可以通过订单系统的价格计算的功能，完成待确认订单的价格的计算，用户确认每个订单条目使用的促销活动以及订单条目的优惠价格，确认好收货地址以及计算好的运费，确认订单的总价格，以及发票抬头等可选的信息

接着可以点击提交订单的按钮，把确认好的订单提交到订单系统

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/061\_再次梳理订单系统的业务模型和流程（6）/笔记.docx**

《061\_再次梳理订单系统的业务模型和流程（6）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，再加上运费，此时就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：UnconfirmedOrder

待确认订单条目：UnconfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：PromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：UnconfirmedOrderPrice

DDD：面向业务来进行开发

用户在订单确认页面上，对每个订单条目都可以选择不同的促销活动，此时可以通过订单系统的价格计算的功能，完成待确认订单的价格的计算，用户确认每个订单条目使用的促销活动以及订单条目的优惠价格，确认好收货地址以及计算好的运费，确认订单的总价格，以及发票抬头等可选的信息

接着可以点击提交订单的按钮，把确认好的订单提交到订单系统

订单系统接收到一个订单之后，此时首先会把订单以及订单条目都录入到数据库里去，同时设置订单的状态为“待付款“，接着会调用库存系统的锁定库存的功能，把该订单要购买的各个商品的库存都进行锁定

库存系统在锁定订单要购买的各个商品的库存的时候，可能会发现库存不足，此时有可能会返回库存不足的提示给订单系统，同时商品库存的锁定就会失败，订单系统就需要把之前录入的订单进行事务回滚，同时提醒用户提交订单失败，库存不足

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/055\_在理解业务模型和技术架构的基础上进行建模/笔记.docx**

3616118202

《055\_在理解业务模型和技术架构的基础上进行建模》

产品需求，领域（业务），子域（业务里的一块而已，子系统），业务模型（通用语言，modules），业务+技术的架构（modules -> 微服务），团队内部分配，每个人负责自己的服务内部的领域建模

实体（属性+行为） -> 值对象 -> 领域服务 -> 领域事件 -> 聚合 -> 工厂 -> 资源库 -> modules -> 上下文集成 -> 用户界面 -> 应用服务

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/055~063资料/058\_再次梳理订单系统的业务模型和流程（3）/笔记.docx**

3616118202

《058\_再次梳理订单系统的业务模型和流程（3）》

我们之前初步的梳理过业务流程，通过流程的梳理，把通用语言给确定了出来，当时我们仅仅记录了通用语言这一套术语，但是背后的业务流程并没有记录下来，所以现在得重新的梳理一下

技术架构图，真的很简单

写英文，我们就写中文，你得先明确的把你的系统的业务功能和业务流程，都写出来，在这个过程中，你就会提取很多的名词、动词、形容词出来，作为你的通用语言，通用语言和业务流程，其实是相辅相成

业务模型 -> 如果你把你的业务流程给画出来，把里面的术语都画在流程图里，此时他就是一整套的业务模型

订单系统可以计算待确认订单的价格，每个待确认订单中都包含多个订单条目

此时需要将每个订单条目都计算一下价格，对每个订单条目都需要根据用户选择的促销活动来计算价格，一个订单条目有可能绑定了一个促销活动，也可能没有使用任何的促销活动，用户有选择的权利

订单条目中包含了一个商品，商品有自己的原价，订单条目绑定的促销活动中包含了促销规则，此时需要根据促销规则来计算商品的优惠价格，进而计算出订单条目的优惠价格

将每个订单条目的优惠价格累加起来，就能得到待确认订单的整体的优惠价格

每一个术语，中文+英文+含义

订单系统：OrderSystem

计算订单价格：CalculateOrderPrice

待确认订单：ToBeConfirmedOrder

待确认订单条目：ToBeConfirmedOrderItem

促销活动：PromotionActivity

促销规则：PromotionRule

待确认订单条目对应的商品：Product

待确认订单条目/订单条目对应商品的原价：Price

待确认订单条目/订单条目的优惠价格：OrderItemPromotionPrice

待确认订单的运费：Freight

待确认订单的总价格：ToBeConfirmedOrderPrice

提交订单：SubmitOrder

订单：Order

订单编号：Order

订单编号：OrderSerialNumber

用户编号：UserID

收件人：Consignee

收件地址：DeliveryAddress

联系电话：ContactPhoneNumber

订单总金额：OrderAmount

订单优惠金额：DiscountAmount

运费：Freight

订单应付金额：PayableAmount

支付方式：PayType

发票抬头：InvoiceTitle

订单备注：OrderComment

订单条目：OrderItem

订单编号：OrderSerialNumber

商品SKU编号：ProductSkuCode

创建订单：CreateOrder

库存系统：InventorySystem

锁定库存：LockStock

扣减库存：DeductStock

支付系统：PaySystem

订单支付成功事件：OrderPayedEvent

订单状态：OrderStatus

履约系统：FulfillmentSystem

履约订单：fulfilOrder

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/050\_领域建模应该怎么一步一步完成？（4）/笔记.docx**

《050\_领域建模应该怎么一步一步完成？（4）》

战略建模：划分子域、设计通用语言、上下文映射图

架构设计，做了一下，技术设计，业务+技术，柔和在一个图里来设计，也挺好的，简便易读，看一张图，业务+技术，都能看到了

详细设计，数据库表结构、缓存数据结构、ES映射结构、接口设计、业务流程图

领域建模，领域对象的建模

在以前的详细设计里，完全不去care你的代码是怎么写的，类、成员变量、方法，类与类之间的交互是怎么做的

写出来的代码是什么样子的，可以说，完全没有任何的保障，不同的程序员写出来的代码，1000个哈姆雷特，每个程序员先不说自己的代码写的如何，看到别人的代码，都会说，怎么这么烂

产出的代码，可读性很差，如果要交接给别人，很难，别人一读，都觉得烂

详细设计里融入DDD的领域建模，确保你的代码完全按照通用语言去写，所有人的代码都按照达成一致的通用语言去写

实体/值对象/聚合 + 领域服务 + 工厂 + 资源库 + 用户界面 + 应用服务 + 基础设施，都建模出来，这套东西完全符合你的通用语言的，指导你的代码里的类、变量、方法应该如何去写，代码完全符合通用语言，还原业务模型+流程

在此基础之上，还原了业务模型+流程的代码里，再融入了各种你的技术架构/设计层面的一些东西，ES、mongodb、HBase，各种牛叉的技术方案，都融合在了你的这套领域模型的建模和设计里面

完全符合你的通用语言的设计和技术架构的设计，完全的把业务+技术的落地到代码级别的建模，都设计了出来

写代码，无脑写，完全照搬你的领域建模+技术架构/方案的一些东西，可能要注意很多技术上的细节

如果有人接手你的代码，先看子域划分，通用语言，业务就完全理解了，技术架构的设计，业务套到技术上设计出来什么样子的，领域建模，再看代码，O了，代码特别的清晰，完全还原业务和技术

先建模实体、值对象，就是说，你的这个系统/服务里，应该包含哪些东西，里面大致有哪些数据，也就是成员变量了，接着你可以根据你要对外提供的业务功能，把实体的行为都建模出来，然后各个实体的行为串联，能够完成业务流程/功能

理论上来说，大致而言，你设计的一堆实体/值对象，基于他们的行为+属性，就可以把你的服务/系统要实现的业务功能/流程，都可以实现了，走查，把你要对外提供的所有功能，或者你自己要具备的一些功能，全部放到你的实体模型里去，看看他们的属性和行为，是否能够满足和实现

业务层面的功能，技术上的架构，大体上来说，应该都可以实现的

在建模行为的这个过程中，你肯定会发现有些行为不适合放某个具体的实体里，那就建模出来领域服务这个东西，里面就是纯行为的一些东西，他会负责找很多的实体完成一个业务流程/逻辑

在这个过程中，你可能还需要建模出来领域事件这个概念

如果说你的子域很大，对应了几十人甚至百人的团队，那就应该拆分为很多的module，就是模块，不同的小团队/个人，负责某个或者某几个module，每个module包含了一部分的领域模型，这个module的概念其实在通用语言设计的时候就可以明确下来

这样其实微服务的架构下，一个服务可以对应一个module

此时，你建模好本module内的模型之后，需要建模出来你跟其他module的模型进行交互的部分，可能是行为交互，也可能是领域事件的交互

除此之外，你还需要在你的module内，把一些领域模型设计为一个聚合，针对聚合和模型，还需要有对应的工厂的设计，以及负责持久化模型的资源库的设计

再然后你要考虑，如果你要负责跟其他的限界上下文进行集成，也就是和其他子域对应的系统进行集成，你具体是如何进行集成的，考虑好这个之后就需要考虑你的用户界面、应用服务的设计

另外就是，在具体设计的过程中，建议就把接口、数据库、还有在你的技术架构/技术方案设计之下，各种技术流程、中间件交互逻辑、技术细节，都包含在里面设计出来，最好不要仅考虑业务方面，要把技术层面考虑进去

大家可以这么理解，子域划分、通用语言设计、限界上下文的映射和集成，是纯业务的，技术架构设计是业务+技术的，之前讲过，可以包含在一个图里，业务架构+技术架构+技术方案，都包含在里面了，业务上是反应出你的DDD战略建模的

然后详细设计，就是把DDD战术建模+技术细节放在一起全部设计出来，就是说，照着这套东西，直接咔嚓咔嚓就可以写代码了，这个并不是老外的DDD理论推荐的，他们的DDD理论就是纯DDD业务建模

但是我推荐的是，把DDD建模和软件工程流程规范融合在一起，而且具体的揉到概要设计和详细设计里去，之前其实主要是偏技术的设计，现在就是业务+技术，都揉到一起去了，以后你掌握了这套思路，那么其实你做任何系统都可以是这个思路了

任何系统，一定是纯业务、技术+业务的概要设计、技术+业务的详细设计，最后照着详细设计直接撸代码就行了，你的每个类怎么写，包含什么，行为的具体逻辑是什么，接口，分层，数据层如何处理，等等

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/047\_领域建模应该怎么一步一步完成？（1）/笔记.docx**

《047\_领域建模应该怎么一步一步完成？》

战略建模：划分子域、设计通用语言、上下文映射图

架构设计，做了一下，技术设计，业务+技术，柔和在一个图里来设计，也挺好的，简便易读，看一张图，业务+技术，都能看到了

详细设计，数据库表结构、缓存数据结构、ES映射结构、接口设计、业务流程图

领域建模，领域对象的建模

在以前的详细设计里，完全不去care你的代码是怎么写的，类、成员变量、方法，类与类之间的交互是怎么做的

写出来的代码是什么样子的，可以说，完全没有任何的保障，不同的程序员写出来的代码，1000个哈姆雷特，每个程序员先不说自己的代码写的如何，看到别人的代码，都会说，怎么这么烂

产出的代码，可读性很差，如果要交接给别人，很难，别人一读，都觉得烂

详细设计里融入DDD的领域建模，确保你的代码完全按照通用语言去写，所有人的代码都按照达成一致的通用语言去写

先建模实体、值对象，就是说，你的这个系统/服务里，应该包含哪些东西，里面大致有哪些数据，也就是成员变量了，接着你可以根据你要对外提供的业务功能，把实体的行为都建模出来，然后各个实体的行为串联，能够完成业务流程/功能

在建模行为的这个过程中，你肯定会发现有些行为不适合放某个具体的实体里，那就建模出来领域服务这个东西，里面就是纯行为的一些东西，他会负责找很多的实体完成一个业务流程/逻辑

在这个过程中，你可能还需要建模出来领域事件这个概念

如果说你的子域很大，对应了几十人甚至百人的团队，那就应该拆分为很多的module，就是模块，不同的小团队/个人，负责某个或者某几个module，每个module包含了一部分的领域模型，这个module的概念其实在通用语言设计的时候就可以明确下来

这样其实微服务的架构下，一个服务可以对应一个module

此时，你建模好本module内的模型之后，需要建模出来你跟其他module的模型进行交互的部分，可能是行为交互，也可能是领域事件的交互

除此之外，你还需要在你的module内，把一些领域模型设计为一个聚合，针对聚合和模型，还需要有对应的工厂的设计，以及负责持久化模型的资源库的设计

再然后你要考虑，如果你要负责跟其他的限界上下文进行集成，也就是和其他子域对应的系统进行集成，你具体是如何进行集成的，考虑好这个之后就需要考虑你的用户界面、应用服务的设计

另外就是，在具体设计的过程中，建议就把接口、数据库、还有在你的技术架构/技术方案设计之下，各种技术流程、中间件交互逻辑、技术细节，都包含在里面设计出来，最好不要仅考虑业务方面，要把技术层面考虑进去

大家可以这么理解，子域划分、通用语言设计、限界上下文的映射和集成，是纯业务的，技术架构设计是业务+技术的，之前讲过，可以包含在一个图里，业务架构+技术架构+技术方案，都包含在里面了，业务上是反应出你的DDD战略建模的

然后详细设计，就是把DDD战术建模+技术细节放在一起全部设计出来，就是说，照着这套东西，直接咔嚓咔嚓就可以写代码了，这个并不是老外的DDD理论推荐的，他们的DDD理论就是纯DDD业务建模

但是我推荐的是，把DDD建模和软件工程流程规范融合在一起，而且具体的揉到概要设计和详细设计里去，之前其实主要是偏技术的设计，现在就是业务+技术，都揉到一起去了，以后你掌握了这套思路，那么其实你做任何系统都可以是这个思路了

任何系统，一定是纯业务、技术+业务的概要设计、技术+业务的详细设计，最后照着详细设计直接撸代码就行了，你的每个类怎么写，包含什么，行为的具体逻辑是什么，接口，分层，数据层如何处理，等等

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/051\_领域建模应该怎么一步一步完成？（5）/笔记.docx**

《051\_领域建模应该怎么一步一步完成？（5）》

战略建模：划分子域、设计通用语言、上下文映射图

架构设计，做了一下，技术设计，业务+技术，柔和在一个图里来设计，也挺好的，简便易读，看一张图，业务+技术，都能看到了

详细设计，数据库表结构、缓存数据结构、ES映射结构、接口设计、业务流程图

领域建模，领域对象的建模

在以前的详细设计里，完全不去care你的代码是怎么写的，类、成员变量、方法，类与类之间的交互是怎么做的

写出来的代码是什么样子的，可以说，完全没有任何的保障，不同的程序员写出来的代码，1000个哈姆雷特，每个程序员先不说自己的代码写的如何，看到别人的代码，都会说，怎么这么烂

产出的代码，可读性很差，如果要交接给别人，很难，别人一读，都觉得烂

详细设计里融入DDD的领域建模，确保你的代码完全按照通用语言去写，所有人的代码都按照达成一致的通用语言去写

实体/值对象/聚合 + 领域服务 + 工厂 + 资源库 + 用户界面 + 应用服务 + 基础设施，都建模出来，这套东西完全符合你的通用语言的，指导你的代码里的类、变量、方法应该如何去写，代码完全符合通用语言，还原业务模型+流程

在此基础之上，还原了业务模型+流程的代码里，再融入了各种你的技术架构/设计层面的一些东西，ES、mongodb、HBase，各种牛叉的技术方案，都融合在了你的这套领域模型的建模和设计里面

完全符合你的通用语言的设计和技术架构的设计，完全的把业务+技术的落地到代码级别的建模，都设计了出来

写代码，无脑写，完全照搬你的领域建模+技术架构/方案的一些东西，可能要注意很多技术上的细节

如果有人接手你的代码，先看子域划分，通用语言，业务就完全理解了，技术架构的设计，业务套到技术上设计出来什么样子的，领域建模，再看代码，O了，代码特别的清晰，完全还原业务和技术

先建模实体、值对象，就是说，你的这个系统/服务里，应该包含哪些东西，里面大致有哪些数据，也就是成员变量了，接着你可以根据你要对外提供的业务功能，把实体的行为都建模出来，然后各个实体的行为串联，能够完成业务流程/功能

理论上来说，大致而言，你设计的一堆实体/值对象，基于他们的行为+属性，就可以把你的服务/系统要实现的业务功能/流程，都可以实现了，走查，把你要对外提供的所有功能，或者你自己要具备的一些功能，全部放到你的实体模型里去，看看他们的属性和行为，是否能够满足和实现

业务层面的功能，技术上的架构，大体上来说，应该都可以实现的

在建模行为的这个过程中，你肯定会发现有些行为不适合放某个具体的实体里，那就建模出来领域服务这个东西，里面就是纯行为的一些东西，他会负责找很多的实体完成一个业务流程/逻辑

在这个过程中，你可能还需要建模出来领域事件这个概念

如果说你的子域很大，对应了几十人甚至百人的团队，那就应该拆分为很多的module，就是模块，不同的小团队/个人，负责某个或者某几个module，每个module包含了一部分的领域模型，这个module的概念其实在通用语言设计的时候就可以明确下来

这样其实微服务的架构下，一个服务可以对应一个module

此时，你建模好本module内的模型之后，需要建模出来你跟其他module的模型进行交互的部分，可能是行为交互，也可能是领域事件的交互

除此之外，你还需要在你的module内，把一些领域模型设计为一个聚合，针对聚合和模型，还需要有对应的工厂的设计，以及负责持久化模型的资源库的设计

再然后你要考虑，如果你要负责跟其他的限界上下文进行集成，也就是和其他子域对应的系统进行集成，你具体是如何进行集成的，考虑好这个之后就需要考虑你的用户界面、应用服务的设计

另外就是，在具体设计的过程中，建议就把接口、数据库、还有在你的技术架构/技术方案设计之下，各种技术流程、中间件交互逻辑、技术细节，都包含在里面设计出来，最好不要仅考虑业务方面，要把技术层面考虑进去

建模，就完全反应了业务+技术两个层面所有的东西，都体现在代码上了，一定要做一个详细设计的技术评审，找其他组员和leader来仔细走查你的建模的东西，按照你的业务功能，技术架构，反复走查，看能否实现所有的业务+技术，是否有漏洞

写代码就咔咔咔照着写就可以了

大家可以这么理解，子域划分、通用语言设计、限界上下文的映射和集成，是纯业务的，技术架构设计是业务+技术的，之前讲过，可以包含在一个图里，业务架构+技术架构+技术方案，都包含在里面了，业务上是反应出你的DDD战略建模的

然后详细设计，就是把DDD战术建模+技术细节放在一起全部设计出来，就是说，照着这套东西，直接咔嚓咔嚓就可以写代码了，这个并不是老外的DDD理论推荐的，他们的DDD理论就是纯DDD业务建模

但是我推荐的是，把DDD建模和软件工程流程规范融合在一起，而且具体的揉到概要设计和详细设计里去，之前其实主要是偏技术的设计，现在就是业务+技术，都揉到一起去了，以后你掌握了这套思路，那么其实你做任何系统都可以是这个思路了

任何系统，一定是纯业务、技术+业务的概要设计、技术+业务的详细设计，最后照着详细设计直接撸代码就行了，你的每个类怎么写，包含什么，行为的具体逻辑是什么，接口，分层，数据层如何处理，等等

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/053\_关于实体唯一标识建模的方方面面（1）/笔记.docx**

《053\_关于实体唯一标识建模的方方面面（1）》

海量数据部分，分布式唯一id生成的方案

1、用户输入的标识

可以允许用户输入一个实体唯一标识，举个例子，比如说用户，用户自己的用户名，可以让他自己输入，对不对，然后设定一些规则来限制，同时系统去重保证一定是唯一的，类似于利用数据的唯一键或者主键来保证唯一

不过一般能允许用户手动输入的唯一标识，还真的不多，我能想到的就是系统全局唯一的、不可重复的用户名！

2、UUID、snowflake、业务属性组合等标识

对于实体唯一标识，也可以用UUID来生成的

// 提前剧透：OrderRepository是负责实体数据存储的资源库，接口

// MyBatis是接口具体的实现类的持久化技术方式

// 在这个资源库里应该包含了对实体的持久化的各种操作

// 生成唯一标识就是其中的一种操作

public class MyBatisOrderRepository implements OrderRepository {

public OrderId nextIdentity() {

String rawOrderId = UUID.xxx();

OrderId orderId = new OrderId(rawOrderId);

return orderId;

}

}

OrderId orderId = orderRepository.nextIdentity();

Date orderCreationDate = orderId.creationDate();

其实这块内容，直接对应了我们在海量数据课程里讲解的唯一id生成机制，除了UUID之外，还可以使用snowflake算法，可以使用美团开源的leaf生成snowflake，那么此时上述的nextIdentity逻辑就调整一下就行了

业务属性+时间戳，来生成的

3、基于数据生成的唯一标识

在海量数据课程里重点讲解了flickr公司开源的方案，基于数据库自增序号段，然后内存里来维护序号段的自增和发号，大家具体可以看一下那个课程，美团开源的leaf也支持这种模式的，但是我不太推荐这样子，对性能和稳定性都没那么好，还是snowflake吧

当然，如果你根本不需要什么高并发之类的机制，那么此时你直接在mysql里对你的这个业务表弄一个对应的seq表，比如说order\_seq表，然后初始化里面的nextval字段的值为0，后续每次你要获取一个新的id，就update order\_seq set nextval=LAST\_INSERT\_ID(nextval+1)，再select last\_insert\_id() ，就可以拿到自增id了

不管怎么说，还是觉得有点怪怪的

4、最自然的唯一标识生成方式

如果一定要我说，你在对某个实体获取自增id的时候，就在数据库里先insert一条这个数据，所有字段都给默认值就可以了，然后就可以通过mybatis的select语句的配置，拿到自增的id了

接着对实体进行持久化的时候，就可以根据唯一标识id进行update操作不就可以了

还有就是取消对外提供的nextIdentity方法，而是仅仅提供持久化实体的方法，首次持久化就是一个insert，此时自然就会产生唯一标识了，你后续就不就得了

5、实体标识和委派标识

有的时候吧，你的实体的标识并不是数据库表中的逻辑id主键，而是自己生成的，比如常见的就是订单编号，用户名之类的东西，然后此时对应数据库里的主键的id字段的，被称之为委派标识，是ORM框架insert之后数据库自动生成的

实际上对你来说，在实体中虽然有这么个东西，但是他仅仅是个普通的属性，根本不会用在代码里的，如果你的实体里要包含这种东西，一般的做法是设计一个抽象父类包含这个属性，所有实体继承这个抽象父类就行了

public abstract class IdentifiedDomainObject implements Serializable {

private long id = -1;

public IdentifiedDomainObject() {

super();

}

protected long id() {

return this.id;

}

protected void setId(long id) {

this.id = id;

}

}

6、到底如何建模合适的实体标识

领域标识说实话，一般不是这种无意义的数据库主键，当然有的时候也可能是，但是如果有业务含义，最好是用业务含义来设计唯一标识，比如说利用一定的唯一算法生成的订单编号才是订单最贴切的实体标识，此时数据库主键就是一个委派标识，id需要隐藏起来不能使用的

还有的时候，实体标识是多个属性组成的，此时他们在数据库里会被设计为联合唯一索引

当然也有的时候，你实在是找不出什么业务含义的东西来建模实体标识，那你就简单的使用数据库自增的主键就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/049\_领域建模应该怎么一步一步完成？（3）/笔记.docx**

《049\_领域建模应该怎么一步一步完成？（3）》

战略建模：划分子域、设计通用语言、上下文映射图

架构设计，做了一下，技术设计，业务+技术，柔和在一个图里来设计，也挺好的，简便易读，看一张图，业务+技术，都能看到了

详细设计，数据库表结构、缓存数据结构、ES映射结构、接口设计、业务流程图

领域建模，领域对象的建模

在以前的详细设计里，完全不去care你的代码是怎么写的，类、成员变量、方法，类与类之间的交互是怎么做的

写出来的代码是什么样子的，可以说，完全没有任何的保障，不同的程序员写出来的代码，1000个哈姆雷特，每个程序员先不说自己的代码写的如何，看到别人的代码，都会说，怎么这么烂

产出的代码，可读性很差，如果要交接给别人，很难，别人一读，都觉得烂

详细设计里融入DDD的领域建模，确保你的代码完全按照通用语言去写，所有人的代码都按照达成一致的通用语言去写

实体/值对象/聚合 + 领域服务 + 工厂 + 资源库 + 用户界面 + 应用服务 + 基础设施，都建模出来，这套东西完全符合你的通用语言的，指导你的代码里的类、变量、方法应该如何去写，代码完全符合通用语言，还原业务模型+流程

在此基础之上，还原了业务模型+流程的代码里，再融入了各种你的技术架构/设计层面的一些东西，ES、mongodb、HBase，各种牛叉的技术方案，都融合在了你的这套领域模型的建模和设计里面

完全符合你的通用语言的设计和技术架构的设计，完全的把业务+技术的落地到代码级别的建模，都设计了出来

写代码，无脑写，完全照搬你的领域建模+技术架构/方案的一些东西，可能要注意很多技术上的细节

如果有人接手你的代码，先看子域划分，通用语言，业务就完全理解了，技术架构的设计，业务套到技术上设计出来什么样子的，领域建模，再看代码，O了，代码特别的清晰，完全还原业务和技术

先建模实体、值对象，就是说，你的这个系统/服务里，应该包含哪些东西，里面大致有哪些数据，也就是成员变量了，接着你可以根据你要对外提供的业务功能，把实体的行为都建模出来，然后各个实体的行为串联，能够完成业务流程/功能

理论上来说，大致而言，你设计的一堆实体/值对象，基于他们的行为+属性，就可以把你的服务/系统要实现的业务功能/流程，都可以实现了，走查，把你要对外提供的所有功能，或者你自己要具备的一些功能，全部放到你的实体模型里去，看看他们的属性和行为，是否能够满足和实现

在建模行为的这个过程中，你肯定会发现有些行为不适合放某个具体的实体里，那就建模出来领域服务这个东西，里面就是纯行为的一些东西，他会负责找很多的实体完成一个业务流程/逻辑

在这个过程中，你可能还需要建模出来领域事件这个概念

如果说你的子域很大，对应了几十人甚至百人的团队，那就应该拆分为很多的module，就是模块，不同的小团队/个人，负责某个或者某几个module，每个module包含了一部分的领域模型，这个module的概念其实在通用语言设计的时候就可以明确下来

这样其实微服务的架构下，一个服务可以对应一个module

此时，你建模好本module内的模型之后，需要建模出来你跟其他module的模型进行交互的部分，可能是行为交互，也可能是领域事件的交互

除此之外，你还需要在你的module内，把一些领域模型设计为一个聚合，针对聚合和模型，还需要有对应的工厂的设计，以及负责持久化模型的资源库的设计

再然后你要考虑，如果你要负责跟其他的限界上下文进行集成，也就是和其他子域对应的系统进行集成，你具体是如何进行集成的，考虑好这个之后就需要考虑你的用户界面、应用服务的设计

另外就是，在具体设计的过程中，建议就把接口、数据库、还有在你的技术架构/技术方案设计之下，各种技术流程、中间件交互逻辑、技术细节，都包含在里面设计出来，最好不要仅考虑业务方面，要把技术层面考虑进去

大家可以这么理解，子域划分、通用语言设计、限界上下文的映射和集成，是纯业务的，技术架构设计是业务+技术的，之前讲过，可以包含在一个图里，业务架构+技术架构+技术方案，都包含在里面了，业务上是反应出你的DDD战略建模的

然后详细设计，就是把DDD战术建模+技术细节放在一起全部设计出来，就是说，照着这套东西，直接咔嚓咔嚓就可以写代码了，这个并不是老外的DDD理论推荐的，他们的DDD理论就是纯DDD业务建模

但是我推荐的是，把DDD建模和软件工程流程规范融合在一起，而且具体的揉到概要设计和详细设计里去，之前其实主要是偏技术的设计，现在就是业务+技术，都揉到一起去了，以后你掌握了这套思路，那么其实你做任何系统都可以是这个思路了

任何系统，一定是纯业务、技术+业务的概要设计、技术+业务的详细设计，最后照着详细设计直接撸代码就行了，你的每个类怎么写，包含什么，行为的具体逻辑是什么，接口，分层，数据层如何处理，等等

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/048\_领域建模应该怎么一步一步完成？（2）/笔记.docx**

《048\_领域建模应该怎么一步一步完成？（2）》

战略建模：划分子域、设计通用语言、上下文映射图

架构设计，做了一下，技术设计，业务+技术，柔和在一个图里来设计，也挺好的，简便易读，看一张图，业务+技术，都能看到了

详细设计，数据库表结构、缓存数据结构、ES映射结构、接口设计、业务流程图

领域建模，领域对象的建模

在以前的详细设计里，完全不去care你的代码是怎么写的，类、成员变量、方法，类与类之间的交互是怎么做的

写出来的代码是什么样子的，可以说，完全没有任何的保障，不同的程序员写出来的代码，1000个哈姆雷特，每个程序员先不说自己的代码写的如何，看到别人的代码，都会说，怎么这么烂

产出的代码，可读性很差，如果要交接给别人，很难，别人一读，都觉得烂

详细设计里融入DDD的领域建模，确保你的代码完全按照通用语言去写，所有人的代码都按照达成一致的通用语言去写

实体/值对象/聚合 + 领域服务 + 工厂 + 资源库 + 用户界面 + 应用服务 + 基础设施，都建模出来，这套东西完全符合你的通用语言的，指导你的代码里的类、变量、方法应该如何去写，代码完全符合通用语言，还原业务模型+流程

在此基础之上，还原了业务模型+流程的代码里，再融入了各种你的技术架构/设计层面的一些东西，ES、mongodb、HBase，各种牛叉的技术方案，都融合在了你的这套领域模型的建模和设计里面

完全符合你的通用语言的设计和技术架构的设计，完全的把业务+技术的落地到代码级别的建模，都设计了出来

写代码，无脑写，完全照搬你的领域建模+技术架构/方案的一些东西，可能要注意很多技术上的细节

如果有人接手你的代码，先看子域划分，通用语言，业务就完全理解了，技术架构的设计，业务套到技术上设计出来什么样子的，领域建模，再看代码，O了，代码特别的清晰，完全还原业务和技术

先建模实体、值对象，就是说，你的这个系统/服务里，应该包含哪些东西，里面大致有哪些数据，也就是成员变量了，接着你可以根据你要对外提供的业务功能，把实体的行为都建模出来，然后各个实体的行为串联，能够完成业务流程/功能

在建模行为的这个过程中，你肯定会发现有些行为不适合放某个具体的实体里，那就建模出来领域服务这个东西，里面就是纯行为的一些东西，他会负责找很多的实体完成一个业务流程/逻辑

在这个过程中，你可能还需要建模出来领域事件这个概念

如果说你的子域很大，对应了几十人甚至百人的团队，那就应该拆分为很多的module，就是模块，不同的小团队/个人，负责某个或者某几个module，每个module包含了一部分的领域模型，这个module的概念其实在通用语言设计的时候就可以明确下来

这样其实微服务的架构下，一个服务可以对应一个module

此时，你建模好本module内的模型之后，需要建模出来你跟其他module的模型进行交互的部分，可能是行为交互，也可能是领域事件的交互

除此之外，你还需要在你的module内，把一些领域模型设计为一个聚合，针对聚合和模型，还需要有对应的工厂的设计，以及负责持久化模型的资源库的设计

再然后你要考虑，如果你要负责跟其他的限界上下文进行集成，也就是和其他子域对应的系统进行集成，你具体是如何进行集成的，考虑好这个之后就需要考虑你的用户界面、应用服务的设计

另外就是，在具体设计的过程中，建议就把接口、数据库、还有在你的技术架构/技术方案设计之下，各种技术流程、中间件交互逻辑、技术细节，都包含在里面设计出来，最好不要仅考虑业务方面，要把技术层面考虑进去

大家可以这么理解，子域划分、通用语言设计、限界上下文的映射和集成，是纯业务的，技术架构设计是业务+技术的，之前讲过，可以包含在一个图里，业务架构+技术架构+技术方案，都包含在里面了，业务上是反应出你的DDD战略建模的

然后详细设计，就是把DDD战术建模+技术细节放在一起全部设计出来，就是说，照着这套东西，直接咔嚓咔嚓就可以写代码了，这个并不是老外的DDD理论推荐的，他们的DDD理论就是纯DDD业务建模

但是我推荐的是，把DDD建模和软件工程流程规范融合在一起，而且具体的揉到概要设计和详细设计里去，之前其实主要是偏技术的设计，现在就是业务+技术，都揉到一起去了，以后你掌握了这套思路，那么其实你做任何系统都可以是这个思路了

任何系统，一定是纯业务、技术+业务的概要设计、技术+业务的详细设计，最后照着详细设计直接撸代码就行了，你的每个类怎么写，包含什么，行为的具体逻辑是什么，接口，分层，数据层如何处理，等等

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/054\_关于实体唯一标识建模的方方面面（2）/笔记.docx**

《054\_关于实体唯一标识建模的方方面面（2）》

海量数据部分，分布式唯一id生成的方案

1、用户输入的标识

可以允许用户输入一个实体唯一标识，举个例子，比如说用户，用户自己的用户名，可以让他自己输入，对不对，然后设定一些规则来限制，同时系统去重保证一定是唯一的，类似于利用数据的唯一键或者主键来保证唯一

不过一般能允许用户手动输入的唯一标识，还真的不多，我能想到的就是系统全局唯一的、不可重复的用户名！

2、UUID、snowflake、业务属性组合等标识

对于实体唯一标识，也可以用UUID来生成的

// 提前剧透：OrderRepository是负责实体数据存储的资源库，接口

// MyBatis是接口具体的实现类的持久化技术方式

// 在这个资源库里应该包含了对实体的持久化的各种操作

// 生成唯一标识就是其中的一种操作

public class MyBatisOrderRepository implements OrderRepository {

public OrderId nextIdentity() {

String rawOrderId = UUID.xxx();

OrderId orderId = new OrderId(rawOrderId);

return orderId;

}

}

OrderId orderId = orderRepository.nextIdentity();

Date orderCreationDate = orderId.creationDate();

其实这块内容，直接对应了我们在海量数据课程里讲解的唯一id生成机制，除了UUID之外，还可以使用snowflake算法，可以使用美团开源的leaf生成snowflake，那么此时上述的nextIdentity逻辑就调整一下就行了

业务属性+时间戳，来生成的

3、基于数据生成的唯一标识

在海量数据课程里重点讲解了flickr公司开源的方案，基于数据库自增序号段，然后内存里来维护序号段的自增和发号，大家具体可以看一下那个课程，美团开源的leaf也支持这种模式的，但是我不太推荐这样子，对性能和稳定性都没那么好，还是snowflake吧

当然，如果你根本不需要什么高并发之类的机制，那么此时你直接在mysql里对你的这个业务表弄一个对应的seq表，比如说order\_seq表，然后初始化里面的nextval字段的值为0，后续每次你要获取一个新的id，就update order\_seq set nextval=LAST\_INSERT\_ID(nextval+1)，再select last\_insert\_id() ，就可以拿到自增id了

不管怎么说，还是觉得有点怪怪的

4、最自然的唯一标识生成方式

如果一定要我说，你在对某个实体获取自增id的时候，就在数据库里先insert一条这个数据，所有字段都给默认值就可以了，然后就可以通过mybatis的select语句的配置，拿到自增的id了

接着对实体进行持久化的时候，就可以根据唯一标识id进行update操作不就可以了

还有就是取消对外提供的nextIdentity方法，而是仅仅提供持久化实体的方法，首次持久化就是一个insert，此时自然就会产生唯一标识了，你后续就不就得了

5、实体标识和委派标识

有的时候吧，你的实体的标识并不是数据库表中的逻辑id主键，而是自己生成的，比如常见的就是订单编号，用户名之类的东西，然后此时对应数据库里的主键的id字段的，被称之为委派标识，是ORM框架insert之后数据库自动生成的

实际上对你来说，在实体中虽然有这么个东西，但是他仅仅是个普通的属性，根本不会用在代码里的，如果你的实体里要包含这种东西，一般的做法是设计一个抽象父类包含这个属性，所有实体继承这个抽象父类就行了

public abstract class IdentifiedDomainObject implements Serializable {

private long id = -1;

public IdentifiedDomainObject() {

super();

}

protected long id() {

return this.id;

}

protected void setId(long id) {

this.id = id;

}

}

6、到底如何建模合适的实体标识

领域标识说实话，一般不是这种无意义的数据库主键，当然有的时候也可能是，但是如果有业务含义，最好是用业务含义来设计唯一标识，比如说利用一定的唯一算法生成的订单编号才是订单最贴切的实体标识，此时数据库主键就是一个委派标识，id需要隐藏起来不能使用的

用户id + snowflake生成的唯一id，组合成了这一个订单编号，一串长的数字

还有的时候，实体标识是多个属性组成的，此时他们在数据库里会被设计为联合唯一索引

当然也有的时候，你实在是找不出什么业务含义的东西来建模实体标识，那你就简单的使用数据库自增的主键就行了

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/047~054资料/052\_对实体和值对象的关系和用法进行浅析/笔记.docx**

《052\_对实体和值对象的关系和用法进行浅析》

实体+值对象，按照业务功能倒推他们的行为和属性，领域服务，领域事件，module，工厂+资源库，上下文集成，用户界面+应用服务

大家一定注意，任何一个实体都是有唯一标识的，是唯一的一个实体，比如一个订单、一个用户、一个快递单、一个发货单、一辆货车，等等，而且实体一定是可变的，里面的数据可以变化的

值对象，一定是没有唯一标识的，他是不可变的，他就是包含了多个数据，这些数据都是不可变的，组成了一个纯粹的数据值对象

一般来说，都会把实体的唯一标识建模为一个值对象，值对象里包含了这个实体的标识数据，而且一定是不可变的

public class Order {

private OrderId orderId;

}

还是那句话，刚开始你先别建模行为和详细的数据属性，先把一个一个的实体、标识、值对象都识别和建模出来，然后再慢慢建模里面的行为和详细数据属性

这周我们先从DDD理论层面，分析分析，这个实体的唯一标识一般都是怎么设计的

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/010\_订单系统DDD建模项目的讲解思路/笔记.docx**

《010\_订单系统DDD建模项目的讲解思路》

对DDD做一个练习，充分的理解业务，跟业务专家，充分的沟通，理解业务，理解里面的名词术语，理解业务流程，你可以尝试自己去定义一套通用语言，拿着通用语言跟你的业务方去商量一下，看看正确不正确

想用DDD玩儿一下实践，一般最好是新系统，最好可以把DDD的思路实践一遍

1、需求评审

2、业务建模（战略建模）：领域/子域，限界上下文，通用语言

3、业务架构设计（上下文映射），技术架构设计（上下文集成架构）

4、技术评审

5、详细设计：~~领域对象建模、数据库设计~~、接口设计（必选）

6、编码开发：实体，值对象，领域服务，领域事件，模块，聚合，工厂，资源库，集成限界上下文，用户界面，应用服务

7、测试环节

8、交付部署

电商云平台，讲解真实的订单业务，完善你的DDD建模的订单系统里的某个服务，有真实的电商平台环境可以部署，验收，检验你的成果

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/004\_DDD的核心价值：让业务和技术有相同的理解（1）/笔记.docx**

3616118202

《004\_DDD的核心价值：让业务和技术有相同的理解（1）》

你作为技术，你一定不能自己随便定义代码里的任何一个类名、方法名、参数名、属性名、变量名、接口名、数据库名，刚开始你不能随便去定义，产品经理找业务方去聊需求，聊完了以后出需求文档，找你来聊

你就直接出技术设计，设计review的事儿，跟产品经理、业务方也好，是没有关系的，粗略的，业务架构和技术架构，表结构的设计和接口名的设计，接口的业务流程的图，其他的就没有

直接找技术团队里的leader和其他成员进行技术设计的review，业务模型，架构设计，技术选型，方案设计

一定会导致具体的工程师在写代码的时候，会自己胡乱的定义类名、方法名、参数名、属性名、变量名、接口名、数据库名，进而导致，他开发出来的代码可能是屎，但是也能实现需求里的功能，也能通过测试，也能交付软件

通用语言，是DDD里最最核心的东西，他其实跟技术无关，完全是业务，业务和技术必须在一起，一起制定一套通用语言出来，这个极为的重要，包括通用语言的中文术语以及英文术语，全部要统一制定，规定术语的含义

通用语言是限定在限界上下文里面的，在一个限界上下文里面，才是一套通用语言，通常我们建议是一个限界上下文对应一个系统，而不是所谓的一个服务而已，否则你不可能为每个服务都创建一套通用语言吧？

通常都是一个系统对应一个小团队，几个人到十来人不等，甚至二三十人也有可能，然后他们都用一套通用语言，对应一个限界上下文

我们认为，如果你要用DDD，最核心的一点改变，就是你要大量的跟你的产品经理，你做出的系统的用户，比如C端用户，或者运营，公司内部其他人，管理层，其他团队，去进行交流，大量的交流，才能把业务真的理解清晰

然后才能让产品、用户和你，站在一起，一起定义一套通用语言出来

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/005\_DDD的核心价值：让业务和技术有相同的理解（2）/笔记.docx**

《005\_DDD的核心价值：让业务和技术有相同的理解（2）》

你作为技术，你一定不能自己随便定义代码里的任何一个类名、方法名、参数名、属性名、变量名、接口名、数据库名，刚开始你不能随便去定义，产品经理找业务方去聊需求，聊完了以后出需求文档，找你来聊

需求文档一般会产品经理找技术团队来进行需求评审，技术来听产品讲解一下系统和功能，以及需求，听懂，这里你就会初步的积累这个系统对应的业务知识

业务建模，针对你们team负责的这个系统去进行通用语言的设计，你们技术团队应该找到业务方，比如说产品、用户、运营，大家一起开会讨论，一起设计这个系统 （限界上下文）的通用语言 -> 战略建模

你就直接出技术设计，设计review的事儿，跟产品经理、业务方也好，是没有关系的，粗略的，业务架构和技术架构，表结构的设计和接口名的设计，接口的业务流程的图，其他的就没有

直接找技术团队里的leader和其他成员进行技术设计的review，业务模型，架构设计，技术选型，方案设计

~~一定会导致具体的工程师在写代码的时候，会自己胡乱的定义类名、方法名、参数名、属性名、变量名、接口名、数据库名，进而导致，他开发出来的代码可能是屎，但是也能实现需求里的功能，也能通过测试，也能交付软件~~

写代码之前，还必须大家都进行细化的领域对象的建模，实体、值对象、领域服务、领域事件、行为、数据，全都建模出来，包括实体之间互动的业务流程，这些东西其实都应该设计出来，全部都是根据通用语言来的

落地写代码，搭建工程，用的技术架构都是之前确定好的技术架构，但是里面具体的类名、方法名、参数名、属性名、变量名、接口名、数据库名，全部都是根据领域对象建模出来的东西来做的，通用语言

通用语言，是DDD里最最核心的东西，他其实跟技术无关，完全是业务，业务和技术必须在一起，一起制定一套通用语言出来，这个极为的重要，包括通用语言的中文术语以及英文术语，全部要统一制定，规定术语的含义

通用语言是限定在限界上下文里面的，在一个限界上下文里面，才是一套通用语言，通常我们建议是一个限界上下文对应一个系统，而不是所谓的一个服务而已，否则你不可能为每个服务都创建一套通用语言吧？

通常都是一个系统对应一个小团队，几个人到十来人不等，甚至二三十人也有可能，然后他们都用一套通用语言，对应一个限界上下文

我们认为，如果你要用DDD，最核心的一点改变，就是你要大量的跟你的产品经理，你做出的系统的用户，比如C端用户，或者运营，公司内部其他人，管理层，其他团队，去进行交流，大量的交流，才能把业务真的理解清晰

然后才能让产品、用户和你，站在一起，一起定义一套通用语言出来

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/002\_再谈没有DDD时困境：严重的交流隔阂/笔记.docx**

3616118202

《002\_再谈没有DDD时困境：严重的交流隔阂》

类名、方法名、参数名、变量名，可能全都是乱写的，完全没任何的逻辑，设计，业务，在里面，都是工程师根据自己的喜好乱写的，最后会导致大部分的系统，在很多人一起维护了一两年以后，基本都成屎

三层架构，代码乱写，大部分系统很容易变成屎，技术跟业务之间术语不统一，交流不通常；不同技术团队之间交流不通畅；新技术人员入职，理解困难；维护一个软件，将成为极其令人恶心的事情

早期如果没有DDD，全靠技术高手自行根据业务充分建模，搭建易于理解的软件模型，可以参考我们的中间件课程，分布式服务注册中心，我在里面就是充分进行业务建模，组件划分，清晰的系统运行流程

甚至开发到一半，进行模型优化重构，一切都是为了软件模型易于理解

但是这种技术高手，在国内，不多，大部分人都容易把软件弄到最后跟屎一样

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/006\_DDD的核心价值：易于理解的软件模型和代码（1）/笔记.docx**

《006\_DDD的核心价值：易于理解的软件模型和代码（1）》

你的软件代码，完全符合领域对象的建模，完全符合通用语言，通用语言又是技术、产品、用户全都达成一致，一起定义和理解的一套通用语言，包括中文名词、英文名词、含义，都是大家达成统一的

你作为技术，当你去跟产品，业务，去聊的时候，大家都是用同一套通用语言来聊的，术语和流程，完全都是一致的，沟通无障碍的；老技术离职，新技术，第一件事情，先看产品的需求文档，先理解产品是干什么的，看通用语言的建模，看代码，eady，very easy，所有的名字，完全符合通用语言

根据制定好的通用语言去细化领域对象的建模，包括数据和行为，流程，全部进行符合业务的建模，一旦做好了，那么软件模型极为的清晰漂亮，代码组件划分的极为清晰，业务流程就跟真实的业务流程一样

举个例子

现在我们想要用代码实现一个功能，还是之前的那个销售顾问给学员开通标准体验课程包的场景，如果用DDD的模式来建模，会是什么样的呢？

SaleConsultant saleConsultant = 通过构造器/工厂模式去还原出销售顾问的实体;

Student student = 通过构造器/工厂模式去还原出学生的实体;

CoursePackage standardExperienceCoursePackage = CoursePackageFactory.createStandardExperienceCoursePackage(); // 构建标注体验课程包，值对象

saleConsultant.setupCoursePackage(student, standardExperiencec=CoursePackage);

上面的代码，对比一下三层架构和面向数据库模式的那个代码看看，这段代码就是DDD里应该有的代码，建模通用语言，然后进行领域对象的建模，建模出实体、值对象、行为，完全还原业务流程，代码极为的清晰和漂亮

再来一个代码级的例子

如果你有很多的业务场景，可能需要对你的学员数据进行各种场景下的信息修改，比如针对他的姓名、邮寄地址、电话、邮箱等信息进行修改，但是可能是在不同的场景下进行修改的，此时你是提供一个简单的updateStudent方法吗？

那肯定不行，那同样会导致失忆症

所以说，可以设计一个Student实体，在里面加入各种业务方法

public class Student {

public void changePersonalName(String personalName);

public void relocateTo(String newPostalAddress);

public void changeMobileTelephone(String new mobileTelephone);

}

这些都可以让我们的代码极为的清晰3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/003\_没有DDD时的贫血对象是如何导致软件失忆的！/笔记.docx**

《003\_没有DDD时的贫血对象是如何导致软件失忆的！》

技术对系统的理解，跟业务对系统的理解，可能不太一致，业务是站在业务的角度去说他理解的术语和流程，技术是站在屎一样的代码的基础上，去说他理解的术语和流程完全是不一致的

我曾经是接手过很多屎一样的代码的，完全就没按照业务模型来建模和写流程，名词都很恶心，耐心慢慢的读代码，甚至对代码进行大量的接口测试，debug，看那个代码如何运行的，理解这个软件对外提供的功能是如何通过屎一样的代码来实现的

过程很漫长，会导致极低的效率，代码量越大，就导致软件越是难以维护

重点看看所谓的贫血对象是如何导致失忆的

比如说我们代码里有一个Service，里面传入一个Domain对象，完了你在里面可能就对这个Domain对象根据id查出来他原本的数据，接着就修改里面的字段值，再修改这个对象，可是这么个方法，到底有谁调用？到底是用来干什么的？在哪个业务场景下用了这个修改的行为？

你都记得这些吗？不记得，不记得的话，那如果一个新人来了，面对这个方法还不是一脸懵逼？一个简单的修改方法都能让你失忆，不知道这个方法是用来干什么的，用在哪个业务场景下，那那些复杂点的方法呢？

这就是贫血模型的失忆症

举个稍微具体点的例子好了，代码级告诉你

比如有一个案例场景，是说在线教育公司里，销售顾问给学员开通了一个标准体验课程包，软件里要实现这么一个功能，你的代码会怎么写？我来写一个大家平时都会写的代码，在三层架构和面向数据库的模式下，大家都是这么写的：

CourseService {

public void createCoursePackage(这里有一大堆的参数，课程包类型参数) {

CoursePackage coursePackage = new CoursePackage();

coursePackage.setType(课程包类型参数);

coursePackage.setStudentId(xxx);

coursePackageDAO.add(coursePackage);

}

}

得了，上面那段代码，你完全就不知道他是干什么用的，具体他是用来创建什么样的课程包，在什么业务场景下，谁创建的，你根本看代码都不知道，一切的一切，都是依靠调用他的人去传入参数的

但是问题是，上面的Service方法可能在N多个业务场景下，被多个Controller方法去调用，传入不同的课程包参数，那你是不是就失忆了？这段代码会搞的你很懵逼的

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/009\_本周小作业：对你负责的系统梳理通用语言/笔记.docx**

《009\_本周小作业：对你负责的系统梳理通用语言》

对DDD做一个练习，充分的理解业务，跟业务专家，充分的沟通，理解业务，理解里面的名词术语，理解业务流程，你可以尝试自己去定义一套通用语言，拿着通用语言跟你的业务方去商量一下，看看正确不正确

想用DDD玩儿一下实践，一般最好是新系统，最好可以把DDD的思路实践一遍

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/007\_DDD的核心价值：易于理解的软件模型和代码（2）/笔记.docx**

3616118202

《007\_DDD的核心价值：易于理解的软件模型和代码（2）》

你的软件代码，完全符合领域对象的建模，完全符合通用语言，通用语言又是技术、产品、用户全都达成一致，一起定义和理解的一套通用语言，包括中文名词、英文名词、含义，都是大家达成统一的

你作为技术，当你去跟产品，业务，去聊的时候，大家都是用同一套通用语言来聊的，术语和流程，完全都是一致的，沟通无障碍的；老技术离职，新技术，第一件事情，先看产品的需求文档，先理解产品是干什么的，看通用语言的建模，看代码，eady，very easy，所有的名字，完全符合通用语言

根据制定好的通用语言去细化领域对象的建模，包括数据和行为，流程，全部进行符合业务的建模，一旦做好了，那么软件模型极为的清晰漂亮，代码组件划分的极为清晰，业务流程就跟真实的业务流程一样

举个例子

现在我们想要用代码实现一个功能，还是之前的那个销售顾问给学员开通标准体验课程包的场景，如果用DDD的模式来建模，会是什么样的呢？

SaleConsultant saleConsultant = 通过构造器/工厂模式/Repostiory组件去还原出销售顾问的实体;

Student student = 通过构造器/工厂模式/Repository组件去还原出学生的实体;

CoursePackage standardExperienceCoursePackage = CoursePackage.createStandardExperienceCoursePackage(); // 构建标注体验课程包，值对象

saleConsultant.setupCoursePackage(student, standardExperienceCoursePackage);

上面的代码，对比一下三层架构和面向数据库模式的那个代码看看，这段代码就是DDD里应该有的代码，建模通用语言，然后进行领域对象的建模，建模出实体、值对象、行为，完全还原业务流程，代码极为的清晰和漂亮

再来一个代码级的例子

如果你有很多的业务场景，可能需要对你的学员数据进行各种场景下的信息修改，比如针对他的姓名、邮寄地址、电话、邮箱等信息进行修改，但是可能是在不同的场景下进行修改的，此时你是提供一个简单的updateStudent方法吗？

那肯定不行，那同样会导致失忆症

所以说，可以设计一个Student实体，在里面加入各种业务方法

public class Student {

public void changePersonalName(String personalName);

public void relocateTo(String newPostalAddress);

public void disconnectMobileTelephone();

}

这些都可以让我们的代码极为的清晰

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/001\_再谈我们用烂了的三层架构和面向数据库的开发/笔记.docx**

3616118202

《001\_再谈我们用烂了的三层架构和面向数据库的开发》

DDD的一个项目，这是一个完整的项目，我们会用完整的DDD理论和方法作为驱动，把DDD完整理论体系都穿插到一个订单系统DDD建模的案例里面，用DDD把订单系统给做一遍

三层架构和面向数据库的开发，就已经没有问题了

核心是面向数据库的开发，用的是面向过程的开发模式（三层架构），说实话，完全没必要用Java语言来开发这种东西，面向对象的，脚本语言，php，python，都是一样的，Java没有看到他的优势

业务架构，技术架构，设计接口，设计数据库的表，类是不用设计的，设计一下每个接口的执行流程（画一些图），先创建一个什么数据，然后校验一个什么东西，再修改一个什么数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/001~010资料/008\_小规模的CRUD系统或者服务，没必要用DDD！/笔记.docx**

《008\_小规模的CRUD系统或者服务，没必要用DDD！》

不用DDD会有多么的恶心；用了DDD会有多么的愉快和清爽

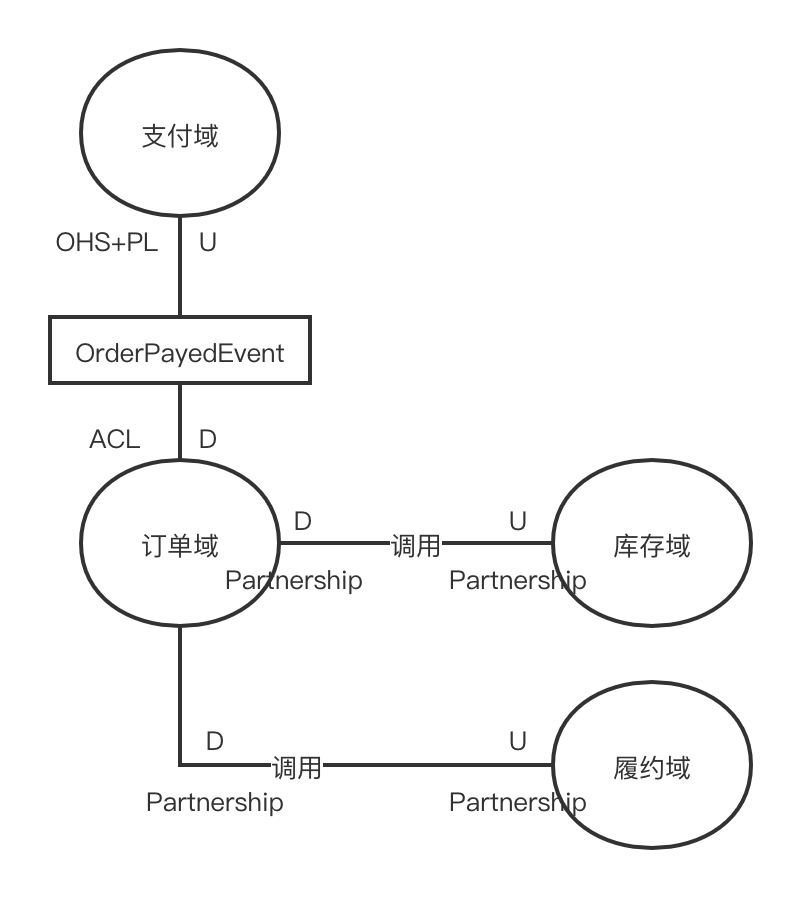
实体，Entity，create、update、query、delete

这里必须给大家澄清一点，那种小规模的CRUD系统或者服务，其实根本没必要用DDD，也就是说一个小系统，或者一个小服务，里面就那么几个接口，或者10来个接口，内部都是一些简单的controller->service->dao的crud操作，那么这种其实根本没必要用所谓的DDD来做

之前大家已经对DDD有一点感觉了，都知道，有时候强行上DDD其实没什么必要，因为简单的crud其实即使你用了DDD，也没有什么太明显的效果

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/030~039资料/订单域上下文映射图.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/20\_基于DDD设计交易核心之复杂订单系统unzip/040~046资料/订单系统架构图.png**

