都在聊DDD, 哪里超越了MVC?

要想深入掌握和了解 DDD 领域驱动设计的核心，那无论如何也绕不开两大较为抽象的概念——“贫血模型”、“充血模型”：

- 贫血模型即事务脚本模式。
- 充血模型即领域模型模式。

- 贫血模型 -

贫血模型最早广泛应用源于EJB2，最强盛时期则是由Spring创造，将：

- “行为”（逻辑、过程）；
- “状态”（数据，对应到语言就是对象成员变量）。

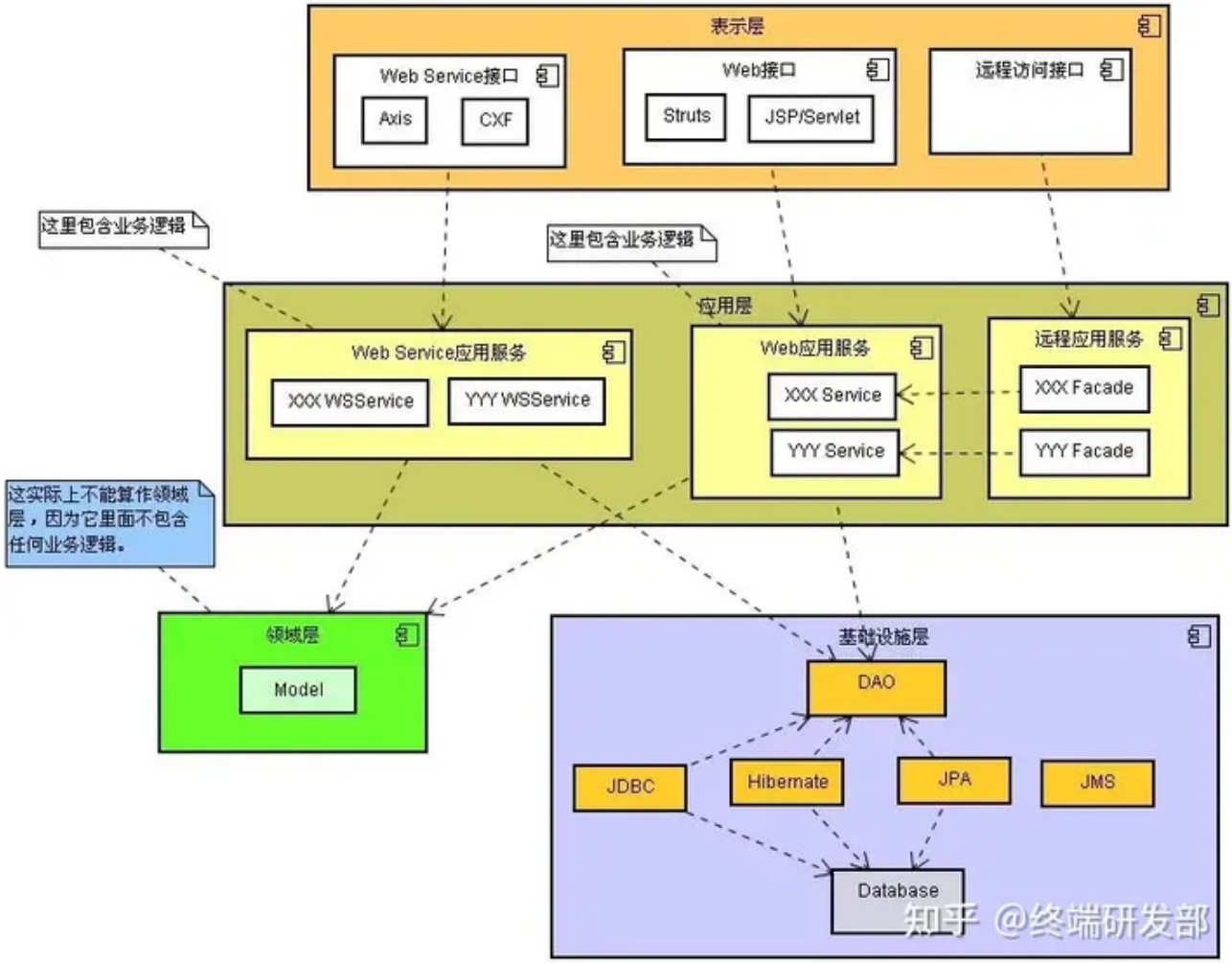
分离到不同的对象中：

- 只有状态的对象就是所谓的“贫血对象”（常称为VO——Value Object）；
- 只有行为的对象就是，我们常见的N层结构中的Logic/Service/Manager层（对应到EJB2中的Stateless Session Bean）。

——曾经Spring的作者Rod Johnson也承认，Spring不过是在沿袭EJB2时代的“事务脚本”，也就是面向过程编程。

贫血领域模型是一个存在已久的反模式，目前仍有许多拥趸者。

Martin Fowler 曾经和 Eric Evans 聊天谈到它时，都觉得这个模型似乎越来越流行了。作为领域模型的推广者，他们觉得这不是一件好事。



贫血领域模型的基本特征是：它第一眼看起来还真像这么回事儿。项目中有许多对象，它们的命名都是根据领域来的。对象之间有着丰富的连接方式，和真正的领域模型非常相似。但当你检视这些对象的行为时，会发现它们基本上没有任何行为，仅仅是一堆getter/setter。

其实，这些对象在设计之初就被定义为只能包含数据，不能加入领域逻辑；逻辑要全部写入一组叫Service的对象中；而Service则构建在领域模型之上，需要使用这些模型来传递数据。

这种反模式的恐怖之处在于：它完全和面向对象设计背道而驰。

面向对象设计主张将数据和行为绑定在一起，而贫血领域模型则更像是一种面向过程设计，Martin Fowler和Eric在Smalltalk时就极力反对这种做法。更糟糕的是，很多人认为这些贫血领域对象是真正的对象，从而彻底误解了面向对象设计的涵义。

如今，面向对象的概念已经传播得很广泛了，而要反对这种贫血领域模型的做法，还需要更多论据。贫血领域模型的根本问题是，它引入了领域模型设计的所有成本，却没有带来任何好处。最主要的成本是将对象映射到数据库中，从而产生了一个O/R（对象关系）映射层。

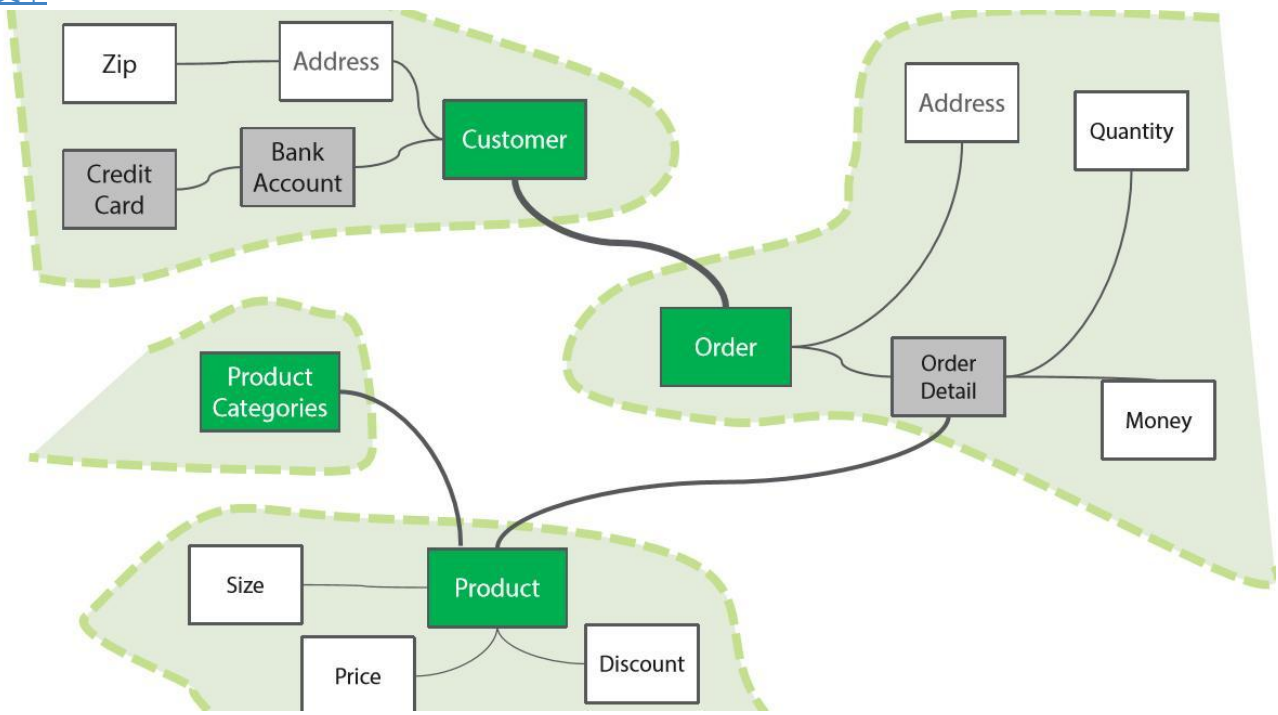
只有当你充分使用了面向对象设计来组织复杂的业务逻辑后，这一成本才能够被抵消。如果将所有行为都写入到Service对象，那最终你会得到一组事务处理脚本，从而错过了领域模型带来的好处。正如martin在企业应用架构模式一书中说到的，领域模型并不一定是最好的工具。

将行为放入领域模型，这点和分层设计（领域层、持久化层、展现层等）并不冲突。因为领域模型中放入的是和领域相关的逻辑——验证、计算、业务规则等。如果你要讨论能否将数据源或展现逻辑放入到领域模型中，这就不在本文论述范围之内了。

一些面向对象专家的观点有时会让人产生疑惑，他们认为的确应该有一个面向过程的服务层。但是，这并不意味着领域模型就不应该包含行为。事实上，service层需要和一组富含行为的领域模型结合使用。

这里有一篇关于DDD写的比较不错的一篇文章：

[终端研发部：什么是DDD（领域驱动设计）？ 这是我见过最容易理解的一篇关于DDD 的文章了920 赞同 · 105 评论文章]



Eric Evans的Domain Driven Design一书中提到：

应用层（即Service层）

描述应用程序所要做的工作，并调度丰富的领域模型来完成它。这个层次的任务是描述业务逻辑，或和其它项目的应用层做交互。这层很薄，不包含任何业务规则或知识，仅用于调度和派发任务给下一层的领域模型。这层没有业务状态，但可以为用户或程序提供任务状态。

领域层（或者叫模型层）

表示业务逻辑、业务场景和规则。该层次会控制和使用业务状态，即使这些状态最终会交由持久化层来存储。总之，该层是软件核心。

服务层很薄——所有重要的业务逻辑都写在领域层。他在服务模式中复述了这一观点：如今人们常犯的错误是不愿花时间将业务逻辑放到合适的领域模型中，从而逐渐形成面向过程的程序设计。

我不清楚为什么这种反模式会那么常见。我怀疑是因为大多数人并没有使用过一个设计良好的领域模型，特别是那些以数据为中心的开发人员。此外，有些技术也会推动这种反模式，比如J2EE的Entity Bean，这会让我更倾向于使用POJO领域模型。

总之，如果你将大部分行为都放置在服务层，那么你就会失去领域模型带来的好处。如果你将所有行为都放在服务层，那你就无可救药了。

优点

简单：

- 对于只有少量业务逻辑的应用来说，使用起来非常自然；
- 开发迅速，易于理解；
- 注意：也不能完全排斥这种方式。

缺点

无法良好的应对复杂逻辑：

- 比如收入确认规则发生变化，例如在4月1号之前签订的合同要使用某规则……
- 和欧洲签订的合同使用另外一个规则……

- 充血模型 -

面向对象设计的本质是：“一个对象是拥有状态和行为的”。

比如一个人：

- 他眼睛什么样鼻子什么样这就是状态；
- 人可以去打游戏或是写程序，这就是行为。

为什么要有一个“人Manager”这样的东西存在去帮人“打游戏”呢？举个简单的J2EE案例，设计一个与用户（User）相关功能。

传统的设计一般是：

- 类：User+UserManager；
- 保存用户调用：userManager.save(User user)。

充血的设计则可能会是：

- 类：User；
- 保存用户调用：user.save()；
- User有一个行为是：保存它自己。

其实它们没有什么特别适用的方向，个人更倾向于总是使用充血模型，因为OOP总是比面向过程编程要有更丰富的语义、更合理的组织、更强的可维护性—当然也更难掌握。

因此实际工程场景中，是否使用，如何使用还依赖于设计者以及团队充血模型设计的理解和把握，因为现在绝大多数J2EE开发者都受贫血模型影响非常深。另外，实际工程场景中使用充血模型，还会碰到很多很多细节问题，其中最大的难关就是“如何设计充血模型”或者说“如何从复杂的业务中分离出恰到好处且包含语义的逻辑放到VO的行为中”。

如果一个对象包含其他对象，那就将职责继续委托下去，由具体的 POJO 执行业务逻辑，将策略模式更加细粒度，而不是写 ifelse。

传统架构对比DDD领域设计

软件架构模式发展到现在可以主要经历了三个阶段：UI+DataBase的两层架构、UI+Service+DataBase的多层SOA架构、分布式微服务架构。

一、传统架构的缺点

在前两种架构中，系统分析、设计和开发往往是独立、分阶段割裂进行的。

1、两层架构是面向数据库的架构，根本没有灵活性。

2、微服务盛行的今天，多层SOA架构已经完全不能满足微服务架构应用的需求，它存在这么一些问题

1. 臃肿的servcie
2. 三层分层后文件的随意组装方式
3. 技术导向分层，导致业务分离，不能快速定位。

比如，在系统建设过程中，我们经常会看到这样的情形：A 负责提出需求，B 负责需求分析，C 负责系统设计，D 负责代码实现，这样的流程很长，经手的人也很多，很容易导致信息丢失。最后，就很容易导致需求、设计与代码实现的不一致，往往到了软件上线后，我们才发现很多功能并不是自己想要的，或者做出来的功能跟自己提出的需求偏差太大。

在这两种模式下，软件无法快速响应需求和业务的迅速变化，最终错失发展良机。此时，分布式微服务的出现就有点恰逢其时的意思了。

二、DDD领域驱动

虽说分布式微服务有这么好的优点，但也不是适合所有的系统，而且也会有许多问题。

微服务的粒度应该多大呀？微服务到底应该如何拆分和设计呢？微服务的边界应该在哪里？这些都是微服务设计要解决的问题，但是很久以来都没有一套系统的理论和方法可以指导微服务的拆分，综合来看，我认为微服务拆分困境产生的根本原因就是不知道业务或者微服务的边界到底在什么地方。换句话说，确定了业务边界和应用边界，这个困境也就迎刃而解了。

DDD 核心思想是通过领域驱动设计方法定义领域模型，从而确定业务和应用边界，保证业务模型与代码模型的一致性。

领域驱动设计是一种以业务为导向的软件设计方法和思路。我们在开发前，通常需要进行大量的业务知识梳理，而后到达软件设计的层面，最后才是开发。而在业务知识梳理的过程中，我们必然会形成某个领域知识，根据领域知识来一步步驱动软件设计，就是领域驱动设计的基本概念。而领域驱动设计的核心就在于建立正确的领域驱动模型。

1、DDD 包括战略设计和战术设计两部分。

a、战略设计主要从业务视角出发，建立业务领域模型，划分领域边界，建立通用语言的限界上下文，限界上下文可以作为微服务设计的参考边界。

b、战术设计则从技术视角出发，侧重于领域模型的技术实现，完成软件开发和落地，包括：聚合根、实体、值对象、领域服务、应用服务和资源库等代码逻辑的设计和实现。

很多 DDD 初学者，学习 DDD 的主要目的，可能是为了开发微服务，因此更看重 DDD 的战术设计实现。殊不知 DDD 是一种从领域建模到微服务落地的全方位的解决方案。

战略设计时构建的领域模型，是微服务设计和开发的输入，它确定了微服务的边界、聚合、代码对象以及服务等关键领域对象。领域模型边界划分得清不清晰，领域对象定义得明不明确，会决定微服务的设计和开发质量。没有领域模型的输入，基于 DDD 的微服务的设计和开发将无从谈起。因此我们不仅要重视战术设计，更要重视战略设计。

2、DDD的优势

1. 接触到需求第一步就是考虑领域模型，而不是将其切割成数据和行为，然后数据用数据库实现，行为使用服务实现，最后造成需求的首肢分离。DDD让你首先考虑的是业务语言，而不是数据。重点不同导致编程世界观不同。
2. DDD可以更加领域模型界限上下文边界快速拆分微服务，实现系统架构适应业务的快速变化，例如：系统的用户量并发量增长得很快，单体应用很快就支持不了，如果我们一开始就采用DDD领域驱动设计，那我们就能很快的把服务拆分成多个微服务，以适应快速增长的用户量。
3. DDD 是一套完整而系统的设计方法，它能带给你从战略设计到战术设计的标准设计过程，使得你的设计思路能够更加清晰，设计过程更加规范。
4. 使用DDD可以降低服务的耦合性，让系统设计更加规范，即使是刚加入团队的新人也可以根据业务快速找到对应的代码模块，降低维护成本。
5. DDD 善于处理与领域相关的拥有高复杂度业务的产品开发，通过它可以建立一个核心而稳定的领域模型，有利于领域知识的传递与传承。
6. DDD 强调团队与领域专家的合作，能够帮助你的团队建立一个沟通良好的氛围，构建一致的架构体系。
7. DDD 的设计思想、原则与模式有助于提高你的架构设计能力。
8. 无论是在新项目中设计微服务，还是将系统从单体架构演进到微服务，都可以遵循 DDD 的架构原则。

3、DDD设计原则

1. 要领域驱动设计，而不是数据驱动设计，也不是界面驱动设计。
2. 要边界清晰的微服务，而不是泥球小单体。
3. 要职能清晰的分层，而不是什么都放的大箩筐。
4. 要做自己能 hold 住的微服务，而不是过度拆分的微服务。

4、微服务拆分需要考虑哪些因素?

理论上一个限界上下文内的领域模型可以被设计为微服务，但是由于领域建模主要从业务视角出发，没有考虑非业务因素，比如需求变更频率、高性能、安全、团队以及技术异构等因素，而这些非业务因素对于领域模型的系统落地也会起到决定性作用，因此在微服务拆分时我们需要重点考虑它们。我列出了以下主要因素供你参考。

4.1、基于领域模型

基于领域模型进行拆分，围绕业务领域按职责单一性、功能完整性拆分。

4.2、基于业务需求变化频率

识别领域模型中的业务需求变动频繁的功能，考虑业务变更频率与相关度，将业务需求变动较高和功能相对稳定的业务进行分离。这是因为需求的经常性变动必然会导致代码的频繁修改和版本发布，这种分离可以有效降低频繁变动的敏态业务对稳态业务的影响。

4.3、基于应用性能

识别领域模型中性能压力较大的功能。因为性能要求高的功能可能会拖累其它功能，在资源要求上也会有区别，为了避免对整体性能和资源的影响，我们可以把在性能方面有较高要求的功能拆分出去。

4.4、基于组织架构和团队规模

除非有意识地优化组织架构，否则微服务的拆分应尽量避免带来团队和组织架构的调整，避免由于功能的重新划分，而增加大量且不必要的团队之间的沟通成本。拆分后的微服务项目团队规模保持在 10 ~ 12 人左右为宜。

4.5、基于安全边界

有特殊安全要求的功能，应从领域模型中拆分独立，避免相互影响。

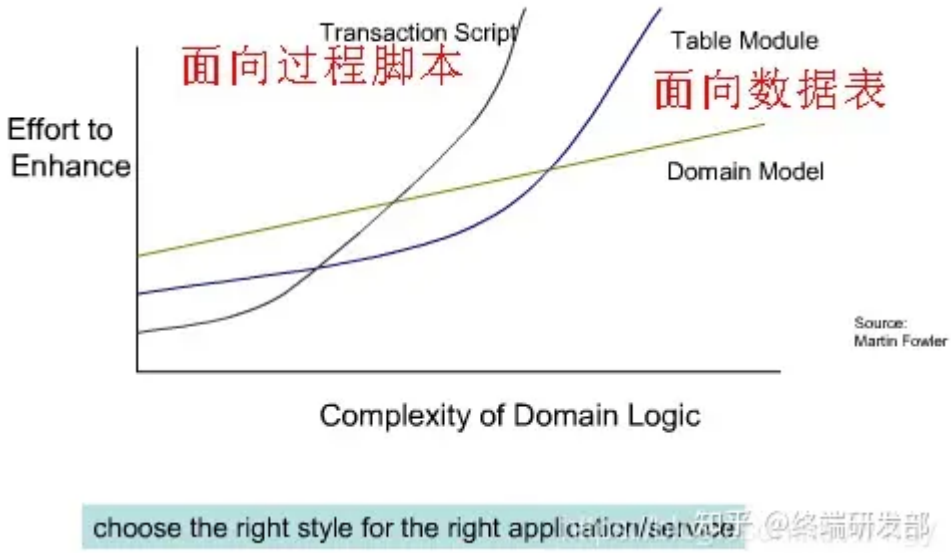
4.6、基于技术异构

领域模型中有些功能虽然在同一个业务域内，但在技术实现时可能会存在较大的差异，也就是说领域模型内部不同的功能存在技术异构的问题。由于业务场景或者技术条件的限制，有的可能用.NET，有的则是Java，有的甚至大数据架构。对于这些存在技术异构的功能，可以考虑按照技术边界进行拆分。

5、其他

效率问题

虽然DDD有这么多明显的优势，但也不是所有的系统都适合采用DDD领域驱动设计。



如上图是两层架构、多层SOA架构与DDD随着业务复杂度的增加在开发成本上的比较，可以看到刚开始DDD的成本是最高的，所以，如果是简单的系统，后续业务不会有太大变化，那么久不适合用DDD，合适才是最好的。

案例介绍

下面通过一个电商业务场景，来介绍如何通过四色模型进行建模，该案例来自InfoQ的文章《运用四色建模法进行领域分析》。

用户故事如下：

现在你是一家在线电子书店的COO。突然有一天，有一位顾客向你投诉，说他订购的书少了一本，并且价钱算错了，他多给了钱。在承诺理赔之前，你需要核对这位顾客说的是否属实。那么这时你需要知道什么样的信息才能做出准确的判断。

简单来说，你需要知道这位顾客订购了哪些书籍、付了多少钱，以及书店到底为这个顾客递送了哪些书籍。不幸的是，由于科技不够发达，你无法直接驾驶时间机器回到从前去亲眼看看发生了什么事。但幸运的是，你并不需要这么做，你只需要看看这位顾客的订单和网银的支付记录，以及你们书店交给EMS的快递单存根，就可以知道这些信息了。

从上面这个故事中我们可以看到：任何的业务事件都会以某种数据的形式留下足迹。我们对于事件的追溯可以通过对数据的追溯来完成。正如在故事中，你无法回到从前去看看到底发生了什么，但是却可以在单据的基础上，一定程度地还原当时事情发生的场景。当把这些数据的足迹按照时间顺序排列起来，我们几乎可以清晰地推测出在过往的一段时间内发生了哪些事情。

为什么这些业务数据具备可追溯性（Tracibility）呢？因为这些数据都是关键业务流程执行的结果。如图7-16所示，比如订单是业务的起点，而快递存根是业务的终点，正是这些数据在支撑运营体系的关键流程的执行结果。

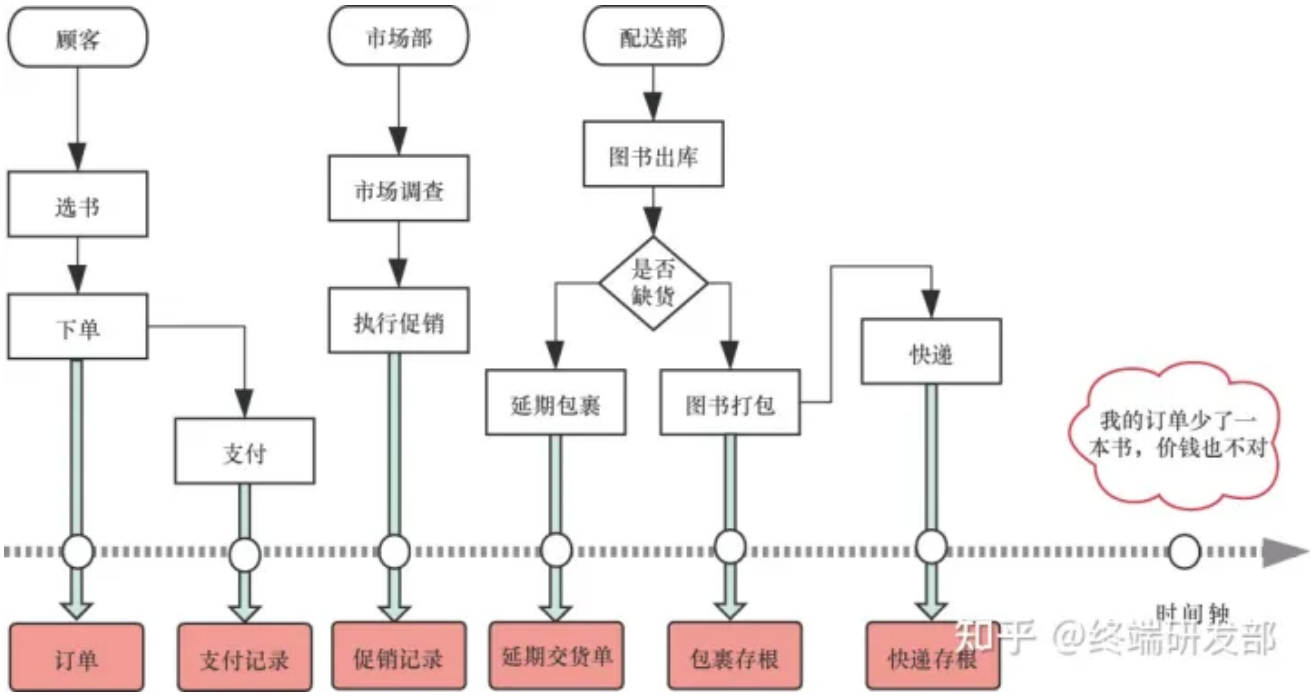


图7-16　在线电子书店的关键业务流程

除了上述例子之外，对于任何一笔正常的经济往来，我们需要知道如下内容。

> 如果我付出一笔资金，那么我的权益是什么？ 如果我收到一笔资金，那么我的义务是什么？

这些问题都需要业务系统捕捉到相应的足迹才能够回答，所以企业的业务系统的主要目的之一，就是记录这些足迹，并将这些足迹形成一条有效的追溯链。

足迹通常都具有一个有意思的特性，**即它们是Moment-interval（要么是“时间时刻”，要么是“时间段”）的。**发现这些业务关键时刻对象就是建模的起点。对这些对象稍加整理，我们就能得到图7-17所示的整个领域模型的骨干。

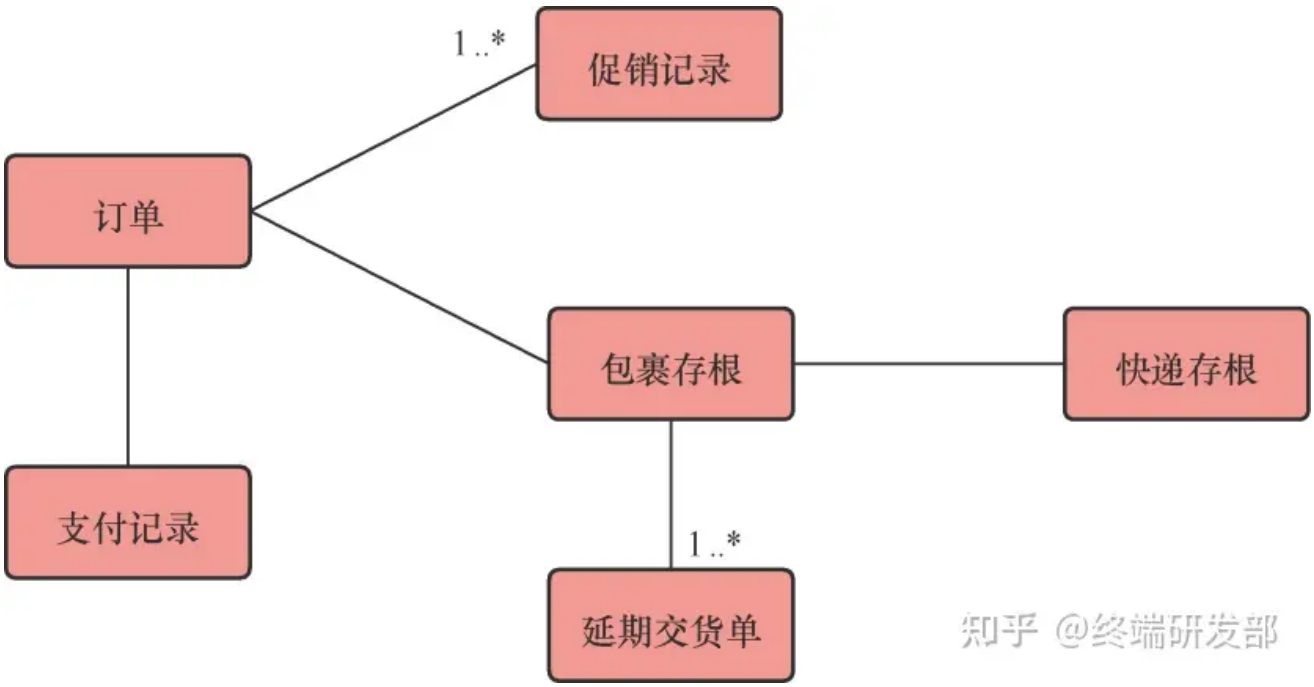


图7-17　在线电子书店的业务关键时刻对象

在得到骨干之后，我们需要丰富这个模型，使它可以更好地描述业务概念。这时我们需要补充一些实体对象，通常实体对象有3类，即人-事-物（Party，Place or Thing），如图7-18所示。

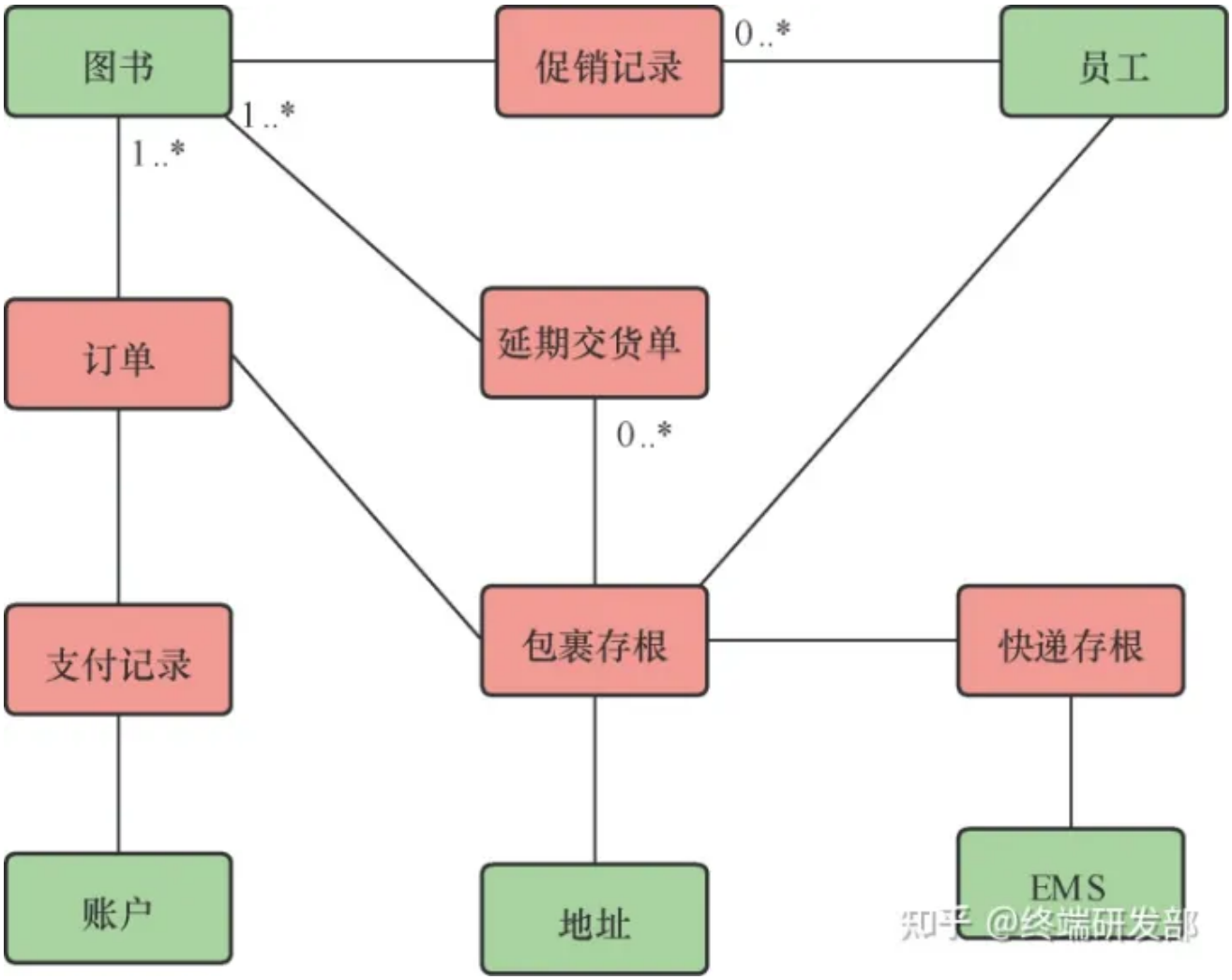


图7-18　在线电子书店的人-事-物对象

在这个基础上，我们可以进一步抽象，将这些实体参与到各种不同的流程中去，这时就需要用到角色（Role），如图7-19所示。

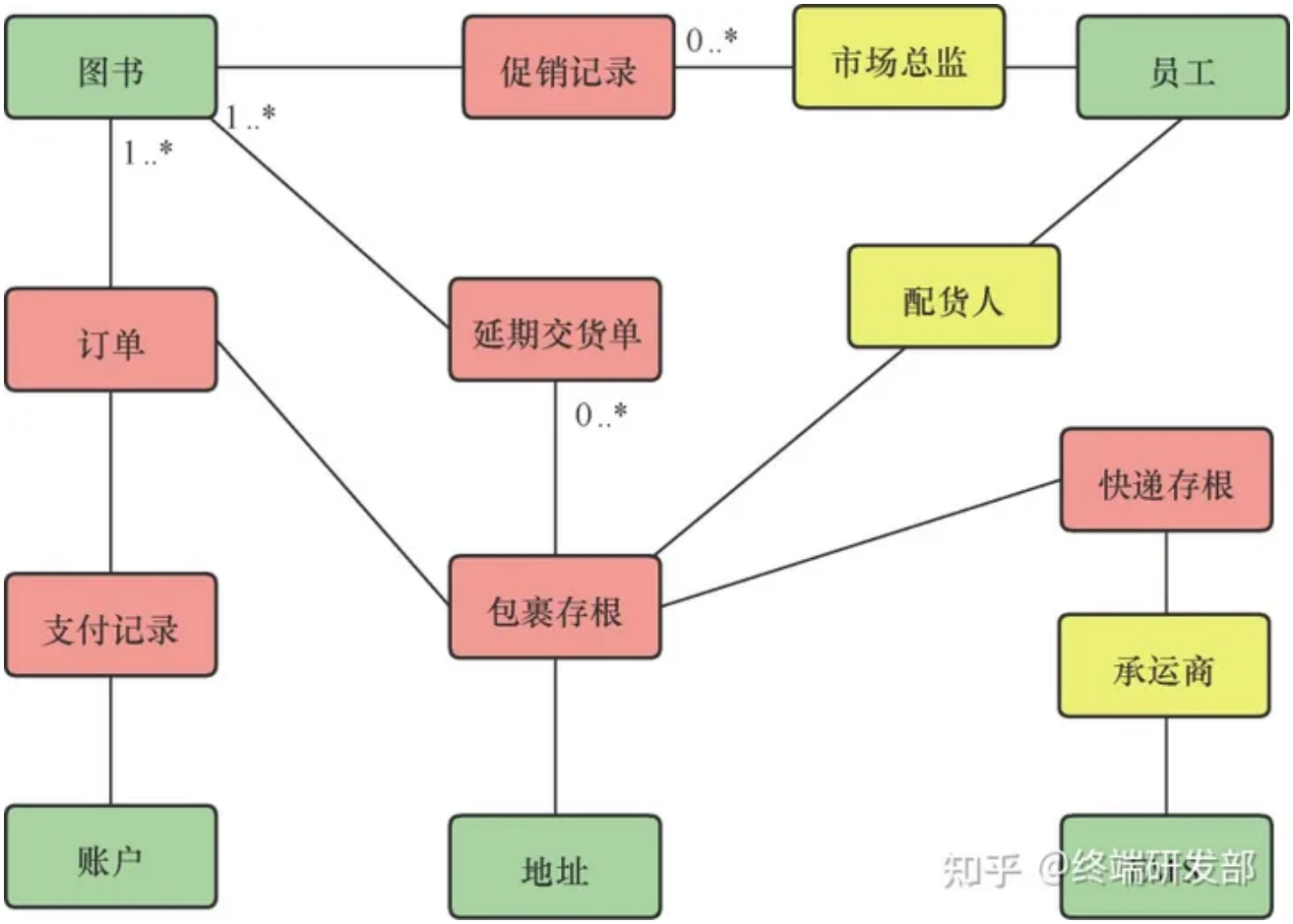


图7-19　在线电子书店的角色对象

最后，把一些需要描述的信息放入描述（Description）对象，如图7-20所示。

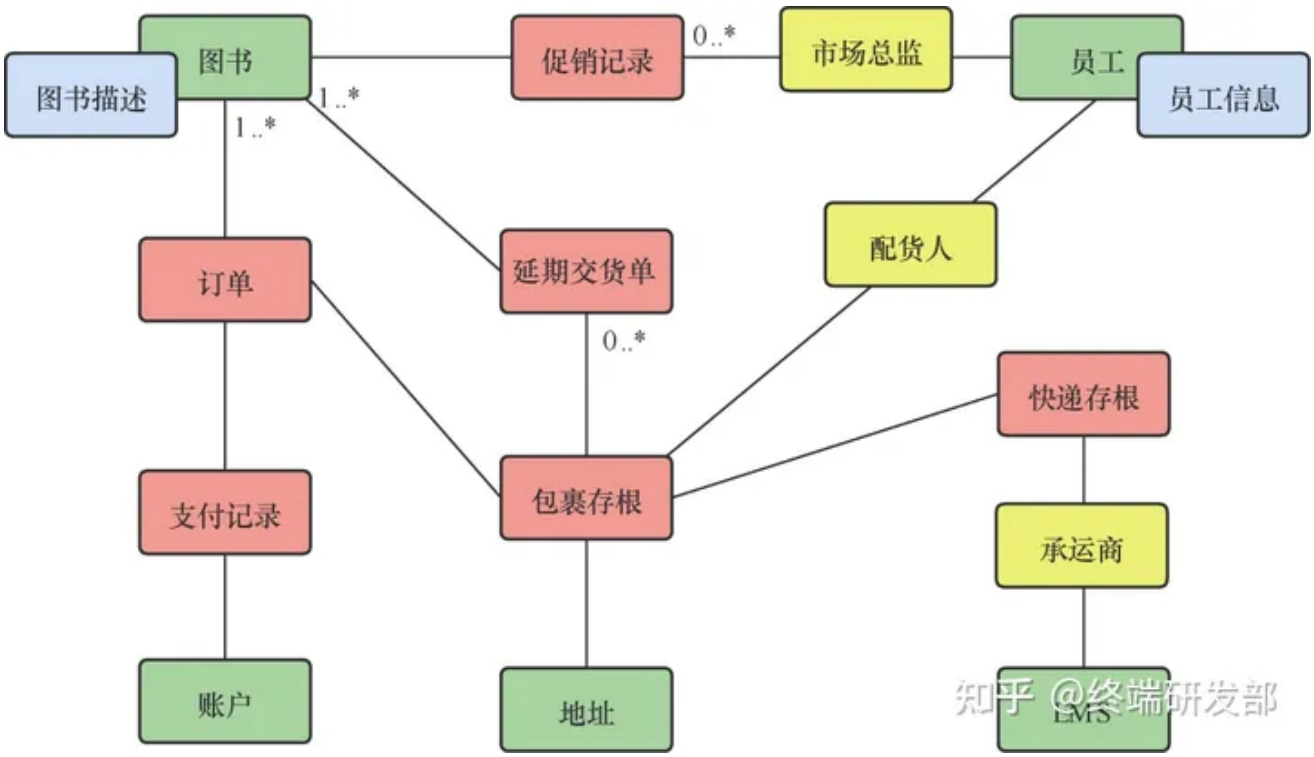


图7-20　电子书店的描述对象

这样，我们就得了应用四色建模方法建立的一套领域模型。简要回顾一下上面的过程，不难发现此次建模的次序和重点。

> （1）首先以满足管理和运营的需要为前提，寻找需要追溯的事件，或者称为关键业务时刻。 （2）根据这些需要追溯，寻找足迹以及相应的关键业务时刻对象。 （3）寻找“关键业务时刻”对象周围的“人-事-物”对象。 （4）从“人-事-物”中抽象出角色。 （5）把一些描述信息用对象补足。

由于在第一步中我们就将管理和运营目标作为建模的出发点，因此整套模型实际上是围绕“如何有效地追踪这些目标”而建立的，这样可以保证模型能够支撑企业的运营。

对设计和开发人员的要求相对较高

DDD 战术设计对设计和开发人员的要求相对较高，实现起来相对复杂。不同企业的研发管理能力和个人开发水平可能会存在差异。尤其对于传统企业而言，在战术设计落地的过程中，可能会存在一定挑战和困难，我建议你和你的公司如果有这方面的想法，就一定要谨慎评估自己的能力，选择最合适的方法落地 DDD。

参考

1. https://blog.csdn.net/w1lgy/article/details/109562193
2. http://juejin.cn/post/6917125801460629518
3. https://blog.csdn.net/u01352789