第 1 部分 创建领域模型

第 1 章

领域驱动设计的概念

软件开发需要领域专家与开发人员协作完成，然而隔行如隔山，交流问题往往是软件开发的最大障碍。领域驱动设计所要解决的问题就是让领域专家和开发人员在一起工作时尽量无障碍地交流，业务核心可以使用代码完整准确地描述，这也是领域驱动设计的吸引力所在。本章首先概要介绍领域驱动设计，包括所要解决的问题、所包含的内容以及学习和使用过程中的难点；然后给出学习和使用领域驱动设计的一些体会；最后简单介绍本书的结构，包括选择的示例、使用的开发环境以及使用的数据和示例代码。

1.1 软件的复杂性

软件，特别是应用软件，是某种特定业务的计算机代码实现，为用户解决特定的业务问题。软件的开发过程，就是将业务问题映射为计算机代码的过程，这个过程涉及多种角色的人员——最终用户、产品经理、需求分析人员、软件开发人员等。不管这个过程如何组织，也不管这个过程中采用什么形式的信息交换方法，最终都是要由开发人员编写代码形成可以运行的软件。所以开发人员在开发软件（特别是应用软件）时需要同时面对两种复杂性：业务复杂性和技术复杂性。一方面，需要理解各种业务问题，将这些问题用软件技术来解决；另一方面，需要解决存储、数据交换以及性能、安全、可靠性等诸多技术问题，这两方面的问题常常交织在一起。更糟糕的是，这些问题随着项目的进行还会发生变化——几乎没有哪个软件项目的需求是不发生变化的。这两种复杂性是天然存在的——从小项目到大项目都有，只是规模越大的项目，复杂性越高。

软件的业务复杂性有两方面的因素：一是软件开发者对于软件所涉及的领域不熟悉，所谓“隔行如隔山”，对领域中的用户来说是简单的问题，对开发人员来说却是复杂的问题；二是业务本身就比较复杂，很多业务只有行业专家才能说得清楚。可以这样说，软件开发人员对于软件所要解决的业务问题通常是外行，软件使用者对于构建软件的技术通常也是外行，而软件通常必须由这两种外行合作完成。外行和外行之间的沟通往往是困难的，经常只能将软件的原型或半成品作为交流的

媒介，然后经过一遍又一遍的修改和迭代逐渐统一认识，在此过程中消耗时间、经费、彼此的信任和耐心。

软件的技术复杂性也有两方面的因素：一是技术本身就比较复杂，应用软件构建的系统涉及的方面很多，例如数据库、中间件、安全等，软件开发人员需要使用这些技术来解决业务问题，然而软件开发人员不一定是所有技术的专家，经常是在实际项目中遇到特定问题时再有针对性地学习；二是软件技术发展速度很快，上半年的技术下半年可能就过时了，经常是“老革命遇到新问题”。

软件的业务复杂性和技术复杂性纠缠在一起，业务因素混杂在技术解决方案之中，当一个业务要素发生变化时，往往会影响到系统的多个方面，这使得软件的开发和维护变得困难。此外，还有一种因素会让这个过程变得更加困难，那就是“变化”。变化有两方面的含义：一是已知业务规则的调整，比如审批流程的改变、税率的变化、健康宝弹窗规则的调整等；二是软件系统（特别是应用软件系统）存在深度和广度上的扩展需求。对于软件而言，变化是必然的，如何应对变化是在软件设计之初就需要考虑的问题。变化本身作为一种潜在的需求，大大增加了软件的复杂性。

1.2　领域驱动设计简介

为了解决软件开发过程中的各种问题，软件界的专家们提出了很多理论和方法，总结了许多最佳实践。其中有些实践是将业务模型化，抽象成各种信息模型，比如数据模型、面向对象模型、工作流模型等，这些模型试图将业务复杂性抽取出来，与实现技术分离。根据这些模型形成代码的过程就是后来所谓的模型驱动开发方法，这些开发方法一直在实践中使用，却没有明确的名称和系统的总结。到了 2003 年，Eric Evans 在 *Domain-Driven Design: Tackling Complexity in the Heart of Software*（领域驱动设计——软件核心复杂性应对之道）一书中提出了领域驱动设计（Domain-Driven Design，简称 DDD），以模式的形式总结了这种开发方法。按 Martin Flower 的说法，Eric Evans 的贡献在于定义了一套用于讨论模型驱动设计方法的词汇，并确定了关键的概念元素。领域驱动设计与模式一样，是先有实践，然后进行总结，将实践中已经存在的“无名的特质”发掘出来，以模式的形式进行描述和命名，形成可供开发人员使用的设计方法。

由于领域驱动设计具有强大的吸引力，因此越来越多的人员投入其中，使之能够快速推广和发展。在发展过程中，衍生出来的概念越来越多，所涵盖的范围越来越广泛，但带来的副作用是原本的基本概念反而不太受关注了，而恰恰是这些基本概念在实践中最有价值。本书将重点放在领域驱动设计的基本概念上，通过实践，使读者对这些概念有深入的理解。

这里首先简单介绍一下领域驱动设计的几个核心概念，这些概念在后面会详细解释。

1.2.1　限界上下文

限界上下文是领域驱动设计引入的重要概念，也可能是最重要的概念。限界上下文的概念不仅可以用在领域驱动设计中，在任何软件分析设计方法中都可以使用，具有很高的实用价值。

在实际项目中，很多术语在不同的业务中的含义是不一样的，比如“成本”，在生产部门只是指原材料等实际生产中发生的费用，而在财务核算时，还要分摊销售、研发等费用，两个部门所指单位产品的成本构成是不一样的。限界上下文解决了这个问题：使用限界上下文规定术语的适用范

围，避免不同上下文中术语的二义性。在生产上下文中的“成本”和在“财务上下文”中的“成本”是两个领域概念，在交流语言中（通用语言）需要进行区分，在软件中需要分别进行建模。如果这两种“成本”在某个上下文中有一定的关系，就需要通过映射反映出这种关系。

限界上下文的目的在于限定范围，在此范围之内，所有的业务概念只有一种含义，而在此范围之外，字面上相同的业务概念可能的含义是不同的。业务概念在领域驱动设计中用领域模型表示，领域模型也是领域驱动设计的核心，而限界上下文规定了领域模型适用的边界。在限界上下文指定的边界范围内，领域模型与通用语言中的概念是一致的，没有二义性。

术语与上下文关系的例子

领域驱动设计的术语本身就是上下文的一个好的例子，当使用领域驱动设计时，所说的“实体”指的是有唯一标识的对象，不是“实体-关系”模型中的实体，所说的“聚合”是指若干关联的实体和值对象，不是统一建模语言（UML）中的“聚合”关系。
还有一个常见的术语是“容器”，读者看到这个词想到的肯定不是锅碗瓢盆，但“容器”到底指的是什么？在不同的上下文中，容器的含义大相径庭，可能是 Docker 中的虚拟化容器，也可能是依赖注入中的组件容器，还可能是集合、列表等的对象容器。

不同的限界上下文之间可能存在某种关系，这些关系可以采用上下文映射（Context Map）进行描述。如果不同的上下文中的模型发生重叠，就需要慎重处理，模型的重叠意味着不同的限界上下文之间存在强耦合关系，首先要考虑的是上下文的划分是否合理，接下来要考虑的是将重叠的模型分离出来，作为共享内核独立存在。

限界上下文中不只包括领域模型，还包括领域模型驱动产生的其他部分，例如，由领域模型驱动产生的数据库结构、用户界面等。这些不是领域模型的一部分，但与领域模型密切相关，同样存在于限界上下文中。

为什么说限界上下文可以应用在任何开发方法中？因为不管使用什么开发方法，都需要对目标系统进行某种形式的分解，而分解结果是否合理，往往不易验证。限界上下文恰好为系统分解提供了依据——业务概念的唯一性。不同的开发方法对系统的划分有不同的术语，在领域驱动设计中，这部分工作称为“战略设计”。

1.2.2 战略设计

领域驱动设计中将应用系统的整体规划或者整体设计称为“战略设计”，这是与在限界上下文内进行领域模型开发的“战术设计”相对应的。

“战略设计”首先要在明确系统业务目标的前提下，使用领域、子域的概念对系统进行分析。子域可以分为核心子域、支撑子域和通用子域。领域和子域是问题空间，对应的解决方案就是限界上下文。一个限界上下文中通常包含一个子域，限界上下文之间的关系使用上下文映射进行描述。

“战略设计”所规划的限界上下文可以独立完成，不是所有限界上下文都必须采用领域驱动设计的方法实现。对于通用限界上下文，可以采用购买第三方产品或使用成熟的开源项目等方式。对于数据维护类的限界上下文，可以使用数据驱动开发完成基本的 CRUD 功能。只有涉及核心业务、包含核心子域的限界上下文，才需要创建领域模型，并使用模型驱动完成开发。

1.2.3　领域模型

在软件开发过程中，使用多种模型对开发的软件进行描述，比如实体-关系模型用来描述系统的数据结构，对象模型用来描述软件结构等。这些模型大多是高层的抽象模型，并不规定具体的实现方式，比如使用实体-关系模型绘制的 E-R 图，并不规定使用的数据库类型；同一个 E-R 图，既可以使用 MySQL，也可以使用 Oracle 创建对应的数据结构。使用 UML 描述的类图也是如此，同一个类图，既可以使用 C#实现，也可以使用 Java 实现。这些抽象的高层模型具有相同的特征，就是必须通过某种转换才能得到具体的可执行代码，这就决定了这些模型是不能直接使用和测试的。领域模型与这些模型不同，它不是抽象模型，而是使用具体编程语言编写的代码，这些代码可以被测试，可以被项目的其他部分引用，作为可执行系统的一部分参与运行。

领域模型是领域驱动设计的核心。领域模型用代码描述领域问题，使用的概念包括实体、值对象、领域事件、领域服务等。领域模型与诸如持久化等实现技术无关，这些部分在领域模型中被定义为接口（比如存储库接口），将具体的实现分离到技术层面，通常这些技术层面的代码叫作基础设施。也就是说领域模型是远离技术复杂性的。如果你使用 C#，那么领域模型应该是基本的 C#类，也就是 POCO；如果你使用 Java，那么领域模型应该是 POJO。领域模型不涉及数据库类型、授权与认证、消息发送方式等，如果你发现在领域模型中引用了与这些技术相关的类库，那么这个领域模型大概率是需要进行重构了。

在设计领域模型时，经常会用到前面提到的高层抽象模型以及相应的建模方法，这些建模方法与领域驱动设计没有矛盾，在构建领域模型时，经常需要使用这些方法进行辅助设计。需要注意的是，领域模型的最终交付物是可使用的代码，而不是类图等描述文档。虽然这些文档化的模型可以帮助用户理解系统，但最终的交付物是使用代码实现的可测试、可使用的领域模型。

1.2.4　通用语言

前面提到了，软件的开发是由领域外行（软件开发人员）和软件开发外行（领域专家）合作完成的，两种外行之间的交流存在复杂性，通用语言就是用于这两种外行之间交流的工具。通用语言中的“通用”是指在项目中通用，在项目划定的上下文中只使用这一种语言。通用语言是严格规定了的自然语言，其中涉及的术语和词汇需要严格定义，没有二义性。通用语言与领域模型是一致的，通用语言描述的业务在领域模型中有对应的代码描述，并且是可以通过测试验证的。

在学习领域驱动设计时，一定要正确理解通用语言的概念，理解它与其他建模语言（如统一建模语言）之间的区别。虽然它们都叫“语言”，但内涵不同，通用语言强调的是交流的属性，并不具备统一建模语言所有的严格语义定义和语言的完备性。通用语言中的“通用”强调的是在单一项目上下文中的通用，是上下文中业务术语的通用，与具体的业务密切相关，脱离了相关的上下文，该“通用语言”就失效了。而使用统一建模语言却可以为几乎所有的软件类型建模，因为它具有完整的语义并且与具体的业务无关。

1.3　领域驱动设计使用中的难点

与所有分析设计方法一样，将理论应用到实践总会遇到各种各样的问题。很多问题是相同的，

例如将某种方法作为包治百病的万灵药（英文中一般称 Silver Bullet，即银弹）、团队的组织问题、与用户的沟通问题等。对领域驱动设计来说也有特定的问题，在参考文献[1][2][3]中也列出了很多，不再重复，这里只列出笔者在实践中遇到的一些问题。

1.3.1 对软件复杂性理解的偏差

可能是由于 Eric Evans 著名的《领域驱动设计——软件核心复杂性应对之道》这本书的书名的原因，导致领域驱动设计被认为是开发复杂软件的方法，只有架构师或者高级开发人员才能涉足，大多数开发人员要么觉得它过于高深而望而却步，要么觉得所开发的软件不是那么复杂，没有必要使用。实际上，软件的复杂性不单决定于规模，更决定于所要实现的业务。在实践中，需要编程完成的项目都不是“简单”的：一方面由于所处的领域不同，对用户或业务专家而言是简单的问题，对于开发人员来说却是复杂的；另一方面，很多最终“复杂”的项目往往是从起初被认为是“简单”的项目演化而来的。很多情况是，当意识到软件的复杂性时，项目已经进行一大半了，只能硬着头皮做下去。

1.3.2 术语的理解

领域驱动设计引入了很多术语，其中有些术语与其他设计方法中的术语在字面上是相同的，但实际含义不同，实体、值对象、聚合这些术语在不同的上下文中的含义也是不同的，这就对领域驱动的学习和使用造成了障碍。下面通过几个例子进行简单的说明，在后面各章有详细介绍。

- 通用语言：在软件开发的语境中提到“语言”，给人的第一感觉是一种完备的编程语言或者建模语言，比如“统一建模语言（UML）”。而“通用语言”所强调的是在项目团队中的“通用”：参与项目的领域专家和开发者使用同一种语言进行交流，在设定的范围内（限界上下文），语言中的术语没有二义性。因此，“通用语言”应该理解为“限界上下文内的专用语言”。
- 实体：它是一个使用广泛的术语，在很多设计方法中都有使用，这里简单说明“实体-关系”模型和 UML 中的实体与领域驱动设计中实体的区别。在“实体-关系”模型中，实体是数据的抽象和概念表示；而在领域驱动设计中，实体不仅包含数据抽象（属性），还包含行为抽象（方法和函数）。在 UML 中，将类分为 3 种——“实体类”“控制类”和“边界类”。这里的“实体类”强调的是数据，虽然也包括方法，但主要用于操作数据，执行逻辑一般在“控制类”中完成；而在领域驱动设计中，没有“控制类”和“边界类”的概念，执行逻辑在实体或者领域服务中实现。
- 聚合：“聚合”在 UML 中表示对象间的一种关联关系；而在领域驱动设计中，“聚合”表示若干有密切关系的实体和值对象，这些实体和值对象之间的关系可能是 UML 中的“聚合”关系，也可能是“组合”关系。

还有很多术语，这里不一一说明。正确理解领域驱动设计中术语的含义是用好领域驱动设计的前提。

1.3.3　技术框架问题

任何软件开发方法在实现时都离不开技术框架的支撑，领域驱动设计也不例外。一方面，如果缺乏技术框架的支撑，很多设计思想就无法实现，可以想象一下如果没有依赖注入框架，那该如何实现软件分层架构；另一方面，技术框架的使用或多或少会引入框架依赖，而这种依赖可能会破坏原本应该遵守的设计原则。

1. 缺乏技术框架的支撑

前面提到了，领域驱动设计早在 2003 年就已经提出了，但并不是立刻就得到大规模的使用，并且经常被认为会增加架构的复杂性。导致这种问题的原因是缺乏技术框架的支撑，特别是缺乏符合领域驱动设计理念的对象-关系映射框架（ORM）。在.Net Framework 的早期版本中，访问数据库采用的是数据访问组件模式，所提供的框架是 ADO.Net，这个框架可以将数据库中的数据模型映射为数据集（DataSet）和数据表（DataTable），应用代码可以操作数据集和数据表，如果映射到对象，则需要编写复杂的开箱装箱代码。由于缺乏成熟的 ORM 框架，因此需要手工编写这些代码，这大大增加了系统开发的工作量和复杂程度。当然，到了现在，这个问题已经基本解决了，EF Core 等框架天然支撑领域驱动设计的开发。

2. 技术框架的约束

这个问题与前面一个问题有一定的关系。实际项目开发时，我们不会从零开始创建项目，而是选择一个已完成的类似项目作为样板，或者选择一个脚手架模板创建项目的基础框架。这样做的好处是充分利用了已有的技术积累，可以缩短开发周期，降低技术风险。由于代码基架中已经隐含了某种设计方法，因此带来的问题是很难引入新的设计方法。

框架本身是基于特定的设计理念或者设计模式，并采用了特定的技术。有些框架会明确指出所基于的设计模式（比如 Python 世界中的 Django 就明确说明基于 Activate Record 模式），而有些框架虽然没有明示，但潜在包含某种设计理念。基于框架进行开发就需要按照框架的设计理念将业务逻辑进行分解，并映射到框架的相应部分。很多早期的框架仍然采用表模式，并使用 ADO.Net 等数据库访问组件。在这些框架上很难应用领域驱动设计的方法。

3. 技术框架的侵入

理想状态下，领域模型应独立于具体的实现框架，但在实际中要完全做到却并不那么容易。检查领域模型是否依赖于某种框架很容易，只需查看领域模型所在类库的依赖关系即可。如果依赖于某特定的框架，就可能存在框架侵入。

框架侵入带来的一个问题是，领域模型必须依赖于这种框架，如果框架发生了变化，领域模型也必须变化，尽管其描述的业务没有发生改变。另一个问题是很难进行框架的替换，由于框架植入了系统的核心，采用其他框架进行替换的代价非常高昂。

下面列举两个技术框架侵入的例子。

常见的一个例子是在领域模型中使用数据描述标签。很多 ORM 框架（如 EF、Dapper 和 PetaPOCO）都支持在类中使用标签描述对应的数据库结构，比如下面的代码：

```
[Table("Player_tb")]
public class Player
```

```
{
    [Key]
    public int Id { get; set; }
    public string UserName{ get; set; }
    public string NickName { get; set; }
}
```

上面的代码说明 Player 对应的数据库表是 Player_tb，Id 作为这个表的关键字。

这种方式的好处是可读性强，可以简化数据库和对象之间的对应关系。缺点有两个：一是领域模型必须依赖相应的框架，二是数据库的结构被硬编码在代码中（想象一下表的名称需要改变的情况）。

还有一个例子是在领域事件定义中使用特定框架的接口。在.Net 生态中，MediatR 是最流行的消息传递框架，如果使用 MediatR 实现事件机制，所定义的事件必须实现特定的接口，示例如下：

```
public class Ping : INotification
{ }
```

如果在领域模型中使用 MediatR 定义领域事件，就会导致 MediatR 对领域模型的侵入。

在开发领域模型时，需要避免框架侵入，尽量将对框架的依赖放到接口的实现中完成。比如前面的第一个例子，可以使用 FluntAPI 代替数据标签实现相同的功能。第二个例子就稍微复杂一些，需要对领域事件在实现层进行二次封装，这个问题在后面会详细讨论。

如果无法完全避免框架入侵，或者避免框架入侵的代价过大，就需要对依赖的技术框架进行评估，并明示可能的风险，从而将影响降到最低。

1.3.4 英语障碍

英语障碍来源于两方面：

一方面是很多术语是由英文翻译过来的，属于外来语，很多术语本身就让人费解；而有些障碍则是翻译过程中人为制造的，笔者高度怀疑有些翻译者故弄玄虚，把简单的词语翻译得高深莫测。这个障碍不仅存在于领域驱动设计中，在其他很多地方都可以遇到。例如，上中学学习物理时，对“功”的概念感觉很高深，不好理解，等到后来发现是从“work”翻译过来的，就觉得这个概念其实不那么复杂；上大学时，有个“鲁棒性”的概念，感觉很神秘，后来发现是从“robustness”音译过来的，就觉得为什么不译成“健壮性”，这样一看就懂。领域驱动设计的翻译中倒是没有这些难以理解的翻译，却存在很多不一致性。比如 Ubiquitous Language，有翻译为统一语言的，有翻译为通用语言的；再比如 Repository，有翻译为资源库的，有翻译为存储库的。这些术语的障碍只能通过标准化来解决，可是没有这方面的权威机构进行这种标准化工作，目前也只能以现状为基础，将现状作为事实标准。

英语障碍的另一方面是编程语言和交流语言的不一致。我们采用以英文作为基础的计算机语言编写代码，这至少在目前是无法改变的现状。而现代编程技术更强调代码的人类可读性，很多编程的技术都要求命名具有人类可读性，包括函数的名称、测试用例的名称、变量的命名等，都要求其描述性易于人类理解。领域驱动设计中领域模型和通用语言也是基于这种理念提出的，这样确保代码与交流语言（包括文档）的一致性，而所有这一切所基于的基础是代码与文档使用同一种语言——英语，这对于使用汉语进行交流使用英语进行开发的我们来说，增加了一重挑战。我们不仅需要将业务语言变

为计算机语言，还要在中文与英文之间频繁切换，这对于英文差强人意的开发人员（比如笔者）来说一直是挑战。不仅领域驱动设计如此，很多建模方法都有类似的问题，这就需要我们自己摸索和建立适合国情的方法。

1.4　学习和使用领域驱动设计的一些体会

对于如何学习和使用领域驱动设计，文献[1][2][3][4][5][6][7]中有很多建议可供参考，不再重复，本节只结合笔者的一些体会，提出一些建议。

1.4.1　理解领域驱动设计的精髓

领域驱动设计与技术框架不同，不能像学习大多数技术框架那样按照示例照猫画虎，而应该在实际项目中边学边用。领域驱动设计与项目所使用的软件开发技术和采用的软件生命周期模型密切相关，如果对相关概念没有完整的理解，只是实现某些技术相关的内容，就达不到预期的效果。

下面简单概括一下领域驱动设计的原则。领域驱动设计是打通软件全开发过程交流障碍的开发方法，采用的手段是将业务复杂性和技术复杂性分开。在“战略设计”阶段完成限界上下文的划分，每个限界上下文有通用语言和相应的领域模型。模型、术语和通用语言在限界上下文之间不重叠，限界上下文之间的集成通过“映射”完成。领域模型是贯穿整个开发过程的核心，在设计领域模型时，遵循持久性忽略（Persistence Ignorance）和基础结构忽略（Infrastructure Ignorance）原则。领域模型与通用语言是一致的，可以使用通用语言驱动领域模型的迭代。在实践中，可以尝试建立自己的通用语言，使用通用语言驱动开发，并且经常检查通用语言和模型的一致性。

如果使用领域驱动设计进行开发，可以对照上述原则对正在开发的项目进行检查，看是否有违反这些原则的地方。下面是需要检查的几个例子：

- 项目中领域模型、应用服务和基础设施等是否可以分离到独立的软件模块中？
- 是否存在共用代码库？共用代码库中是否包含模型？
- 包含领域模型的模块是否引用了持久化框架？

如果对第一个问题的回答是“否”，对后面两个问题的回答是“是”，那就要审查项目的代码结构了。

1.4.2　使用“战略设计”规划项目

拿到初始需求后如何下手？首先是明确系统的业务目标，然后了解为了完成业务目标软件需要实现的功能，以及这些功能之间的关系，在此基础上，对项目进行整体设计。整体设计的目的就是将系统进行分解，划分为相对独立的子系统或者模块，这些子系统或模块可以独立进行开发。如果系统很大（涉及众多业务目标），例如面向整个企业的管理系统，那么需要先进行业务级别的建模和规划，逐级进行分解，并分解到每一个子系统承担明确可验收的业务目标后，再进行子系统级别的设计。

在项目规划或者说是整体设计的过程中，可以使用领域驱动设计的“战略设计”。“战略设计”

中的领域、子域以及限界上下文的概念不仅适用于领域驱动设计，也可以用于其他开发方法。使用“战略设计”划分的子系统处于自己的上下文，在这个上下文中，业务术语的含义保持一致性，而这个业务术语在其他上下文中可能有不同的含义。

当使用限界上下文的概念审视以往的项目规划时，往往会发现很多之前没有发现但现在看来是显而易见的问题。这里举一个例子，很多应用系统中都有“基础信息维护”或者类似的模块，这个模块的作用是为其他业务子系统或模块提供公用的基础信息，有些是通用的信息，例如组织机构、人员等，有些是与应用相关的专用信息，例如“设备管理系统”中可能有设备类型、故障类型、设备档案等。然而，被忽略的问题是，很多基础信息代表的概念在不同的子系统或模块中可能具有不同的属性，甚至具有不同的业务含义，尽管所指向的客观实体相同。如果使用限界上下文的概念进行术语确定，会发现大部分的基础信息需要分解到各个限界上下文中，因为在不同的限界上下文中，这些基础信息具有不同的业务含义。

需要说明的是，使用“战略设计”完成限界上下文的划分和映射后，不一定所有的限界上下文都使用领域驱动设计的“战术设计”来完成，甚至有可能完全不使用。每个限界上下文选择的开发方法要依据实际情况确定。因此，领域驱动设计中的“战略设计”更为通用也更为重要。

1.4.3 在开发过程中使用“战术设计”

学习领域驱动设计的最好方法是在开发过程中使用领域驱动设计，将这种设计方法融入开发过程中。

当系统相关的限界上下文都确定了以后，就可以开始针对每个限界上下文确定相应的通用语言，同时开始领域模型的设计了。也就是说从这时起，编码的工作就开始了。这是与很多传统开发方式不同的地方，很多开发方式要求需求分析文档固定下来后才开始编码，而在领域驱动设计中，领域模型是采用代码表达的，编码是分析设计方法的一部分。这样的好处是，作为领域模型的代码是最终项目的一部分，可以避免需求与代码的不一致。需要注意的是，在领域模型编写的过程中，不要涉及具体的技术实现，也不要引入特定的框架。可以创建.Net 类库项目作为领域模型的载体，采用单元测试或者行为驱动测试来验证领域模型。如果涉及业务用例，可以编写服务层代码，用来进行业务编排。服务层也采用.Net 类库作为载体，独立于领域模型，同样采用单元测试进行验证；如果需要进一步的验证，可以编写交互简单的控制台应用程序。如果在领域模型设计中需要使用特定的技术，比如数据库访问、通信等，可以将这些技术涉及的功能抽象为接口，在领域模型外部进行实现。在领域模型创建过程中，技术实现要尽可能简单，能够验证业务过程即可。

当领域模型基本完成后，可以参考软件框架模型搭建应用系统，可选择的模型包括分层模型、六边形框架、洋葱圈框架等，软件框架在第 17 章有详细的介绍。在软件框架中，领域模型处于核心的位置，系统的其他部分引用领域模型，并实现领域模型中定义的接口，所有的业务用例在应用层进行组织，表示层通过调用应用层实现用户与系统的交互。

上面描述的是开发的大致过程，实际上在这个过程中系统的各个部分需要反复迭代、重构和优化。在迭代的过程中，领域模型与通用语言需要保持一致，领域模型需要始终做到与特定实现技术无关，并且尽量避免框架的侵入，虽然这在很多情况下是不可避免的。为了管理这个过程，需要使用各种管理工具和技术，这在后面的章节会有介绍。

1.4.4　在学习中尽量尝试各种技术，在实践中保持简洁

分清楚学习场景和实际应用场景。如果处于学习场景，可以使用复杂的技术和方法实现简单的需求，这样可以在业务复杂度锁定的情况下，练习掌握技术实现。反过来也是一样，可以使用相同的技术反复解决各种假定的业务问题。在学习时，尽量从零开始构建项目，而不是使用项目模板或者项目基架。虽然这样会花费一些时间，但可以深入了解软件各组成部分之间的内在关系。

而在实际项目中，一定要牢记简洁原则（KISS，Keep It Simple，Stupid），避免过度设计。这种原则的体现首先表现在技术路线的选择上，项目中能用一种技术解决的问题，就不要用多种技术，这样可以降低开发成本，增加项目的可维护性。如果是企业级的项目，这一点更加重要。从大的方面来说，技术选型包括使用的关系数据库类型、消息中间件类型、认证服务类型等；从小的方面来说，技术选型包括项目中使用的依赖注入框架、持久化框架等。在开发中，简洁原则首先体现在标准化的解决方案上，开发人员使用标准化的解决方案去处理实际问题，不需要花时间重复解决同样的问题。

1.4.5　实事求是，避免将理论当作教条

在实际项目开发中，一定要结合实际情况运用理论知识，不要将理论作为教条生搬硬套，更不能片面地理解某些示例，在项目中不分场合地套用示例中的模式。

实际项目中经常遇到的一个问题就是如何处理领域模型与持久化架构的关系。理论上讲，领域模型应该对持久化架构无感，可在实际中，很多持久化架构需要在实体类中使用特定的标记来标注实体类与数据库中表或者字段的关系，这种标记的使用已经使领域模型与特定的架构绑定在一起了。这种情况，在 1.3.5 节已经讨论过，在第 10 章介绍关系数据库的存储库时会进行详细介绍。如果由于各种原因，这种方式是唯一的选择，那么在实际中只能接受，同时需要评估对框架的依赖会对项目带来何种影响，并明确说明可能的影响。

实际中经常遇到的另一个问题是应用层和领域层的界限问题，例如某些服务不太好确定是应用服务还是领域服务。很多资料中提到，领域模型负责细粒度的业务，应用服务负责调用编排这些业务，领域模型与用例无关，应用服务完成某个用例。而在实际中，业务编排本身就包含业务含义，如果仅仅将业务编排按照应用服务来设计，就会导致在领域模型中存在的是缺乏主线的碎片化的业务，在应用层中出现大量的复杂的负责编排的服务，从更高层的角度看，整个软件更像是使用“贫血模型”设计的。这时，更多的可能是业务分析不彻底，需要将业务过程模型化，在基础的领域模型之上形成更高层次的领域模型，而复杂的应用服务很可能是一个新的限界上下文。因此，在实际中，需要注意：

①应用层经常是跨限界上下文的。

②如果应用层过于复杂，那么很有可能这是一个候选的限界上下文。

1.5　本 书 概 况

1.5.1　本书的目标和结构

本书的目标是介绍在.Net 环境下使用领域驱动设计开发软件系统的基本方法。本书力求低门槛，

只要对.Net 环境和 C#语言有一定的了解就可以。

本书以一个实例为线索，说明采用领域驱动设计的方法和技术构建软件的过程。本书分为 3 个部分：

第 1 部分介绍领域模型和如何创建领域模型，第 2 部分介绍与领域驱动设计实现相关的.Net 技术，第 3 部分介绍如何以领域模型为核心构造各种类型的应用系统，以及如何实现项目的升级和演化。建议读者跟随本书从零开始学习，完成从创建领域模型到构造可运行的软件这一升级过程。

1.5.2 为什么选择.Net

领域驱动设计不是基于某种具体的编程语言或者平台的，但在实际项目中却需要使用具体的编程语言和平台来实现，特别是很多实现细节，更加依赖所使用的平台。因此，学习领域驱动设计需要依托某一种具体的实现技术，本书选择.Net 平台。

选择.Net 平台的原因有两个。一是.Net 平台是有生命力的发展迅速的平台。现在的.Net 平台摆脱了.Net Framework 的束缚，经过.Net Core 的若干次迭代，已经成为广泛用于构建新式应用和强大的云服务的跨平台框架。二是目前介绍领域驱动设计的书籍[1][2][3][6]，基本上都将 Java 语言和其相关的框架作为实现平台，文献[5]虽然以 C#作为编程语言，但所基于的框架仍是过时的.Net Framework，基于.Net 平台的十分少见。

基于上述两个原因，本书选择 C#作为编程语言，选择.Net 平台作为依托框架，所开发的示例是可以运行在 Docker 容器中的跨平台的前后端分离的分布式应用。

1.5.3 本书选择的示例

领域驱动设计的对象是领域，也就是某种业务，而不是某种计算机技术实现。所以学习领域驱动设计，必须选择一个业务场景作为示例。业务场景过于简单，读者就体会不到使用领域驱动设计的必要性。业务场景复杂，领域驱动设计可以发挥很大的作用，但这需要读者首先学习相关领域的背景知识，而这会导致偏离学习领域驱动设计的初衷。因此，本书需要选择既易于理解，又有一定复杂度的业务场景作为示例。

还有一个问题就是，领域驱动设计用于解决领域专家与开发人员的交流问题，而读者在阅读时是独自一人，这就使得读者具有领域专家和开发者的双重身份，并且易于切换。

本书选择将编写诗词游戏作为示例。首先，诗词游戏易于理解，不需要过多的业务背景，这样，读者的注意力会集中在对业务复杂性的处理上，而不必花过多的时间理解业务本身的复杂性。

其次，所设计的系统具有一定的复杂度：需要玩家登录认证、需要对诗词进行管理、游戏的类型应该可以进行扩展、玩家可以在游戏中进行社交活动等。系统具有很大的弹性，可以做得比较简单，也可以做得很复杂，可以让读者体会应用系统的演化过程。

最后，诗词游戏作为中文为主的文字游戏，可以在很大程度上体现领域驱动设计落地或者说本地化的难点，使读者可以理解领域模型与通用语言之间保持一致性的重要性和难度。

1.5.4 本书使用的开发环境

本书大部分内容采用 C#语言，面向.Net 平台（采用稳定的.Net 6），示例大部分使用 Visual Studio 2022 社区版进行构建。前端部分采用 Visual Studio Code 进行构建，采用 JavaScript 语言，使用的技

术架构包括 Vue.js、Vant 等。

本书中使用的其他工具和环境包括：

- Docker：容器环境。
- Git：源代码管理工具。
- Nuget：.Net 的包管理工具。
- xUnit：单元测试框架。
- SpecFlow：行为驱动设计测试框架。
- MongoDB：NoSQL 文档数据库管理系统。
- MS SQL Server：关系数据库管理系统。
- RabbitMQ：消息中间件。
- Kafka：消息中间件。
- Identity Server 4：基于 OIDC 的认证服务。
- Jenkins：持续集成框架。

在附录 A 中有这些环节和工具的安装与使用的简单介绍。

1.5.5　本书中的数据和代码

本书选取《全唐诗》的一部分作为测试数据。简化起见，很多示例使用了 XML 文件作为数据源。《全唐诗》收录了唐代一千七百余位诗人的四万三千多首诗，作为试验项目，不需要使用所有数据，示例数据中只包括了李白、杜甫、白居易、李商隐和杜牧的五千余首诗，这些示例数据以 XML 文件和 Sqlite 数据库文件的形式包含在示例项目中。

本书的代码分为下面几个部分：

- 诗词游戏上下文诗中的领域模型
- 诗词服务上下文
- 诗词管理上下文
- 人机诗词游戏应用
- 多人诗词游戏应用

需要注意的是，上面的代码是结果代码，而书中所列的代码大部分是过程代码，笔者希望读者可以从过程代码中理解软件的构建过程，而不是只看最后的结果。

1.6　本 章 小 结

领域驱动设计的目的是应对软件的复杂性，方法是使用领域模型驱动软件的开发。本章概要介绍了领域驱动设计的基本概念，包括限界上下文、领域模型和通用语言等，这些概念在后面各章会展开介绍。

针对领域驱动设计学习和使用的难点，本章给出了一些建议。在后面各章中，会将 C#作为实现语言，将.Net 平台作为依托，将诗词游戏开发作为示例进行具体说明。

第 2 章

从零开始构建诗词游戏

从本章开始，将从零开始构建示例应用——诗词游戏，在构建过程中逐步引入领域驱动设计相关的概念。

2.1 需 求 概 述

我们希望开发一个诗词游戏的应用，用户可以通过各种终端（Web、客户端、移动应用）参与游戏，并进行社交活动。该游戏是围绕古诗词设计的文字游戏，可以是单人与计算机进行，也可以多人参与。参与者轮流作答或者抢答，答错或答不出者淘汰，答对加分，最后没有被淘汰的参与者为赢家。游戏类型包括对诗、飞花令、接龙等，游戏类型可以增加。我们将不同类型游戏的初始条件称作游戏条件。下面是几种有代表性的游戏介绍。

- 对诗：对诗就是“上句接下句”，比如一个玩家起头“床前明月光”，接下来对“疑是地上霜”，再下来是“举头望明月”，这里的游戏条件就是“床前明月光”。
- 飞花令：飞花令需要作答的诗句中带有规定的字或词，比如游戏设定“花”，那么“花间一壶酒”“黄四娘家花满蹊”都是符合要求的作答，这里的游戏条件就是“花”。
- 接龙：接龙就是上一句的尾，是下一句的开头，比如“小时不识月”可以接“月里青山淡如画”，这里的游戏条件就是“小时不识月”。

游戏的类型可以扩展，扩展的形式有多种。常见的扩展形式是在现有类型上增加限制条件，而常见的限制条件包括限定某个或某几个作者，或者限定在“唐诗三百首”范围内等。另一种扩展是改变现有游戏类型的规则，超级飞花令就是这种扩展的一个实例，将词句中包含某个字或词改变为可以包含某种类型的字或词，比如“四季”“颜色”等。还有一种扩展是新的玩法，比如“唐诗集对”，用唐诗对对子。游戏应用对游戏类型的扩展应该是开放的，新类型的加入不应该影响已有的类型。

游戏可以由玩家创建，其他的玩家可以加入没有开始的游戏，游戏开始前也可以退出（创建者不能退出），创建者可以决定何时开始游戏。游戏也可以由系统自动创建，系统可以从希望参与游戏的玩家中挑选若干组成游戏，创建完成后就自动开始。游戏创建时需要设定游戏的类型（对诗、

接龙、飞花令等）、游戏上下文以及作答的顺序（轮流作答还是抢答）。

游戏开始后，玩家根据游戏的类型和游戏上下文作答，回答正确加分，回答错误出局，然后轮到下一个玩家作答，最后回答正确的玩家是赢家，游戏结束。

玩家可以进行社交活动，比如交友、创建群组、对游戏进行评论等，参见一般的社交系统功能。

玩家的注册、登录、管理可以参照一般流程。

游戏使用的诗词资源是《全唐诗》，可以使用诗词管理进行扩展。

2.2　领域、子域与限界上下文

在开发应用系统时，首先要解决的问题是从何处下手。不管使用什么开发方法，在确定了系统目标之后要做的事情都是将系统进行逐级分解，分解到可以进行编程的粒度。如果一次分解后粒度仍然过大，就在第一次分解的基础上再次分解，直到可以编程实现为止。不同的开发方法分解系统的方法不同，比如传统结构化开发中的 IDEF 方法[17]等。使用领域驱动设计同样需要将系统进行划分，所使用的概念是领域和子域，划分的结果是限界上下文。

一个领域代表一定的业务范围，所谓领域就是这个业务范围的边界，边界以内属于领域内，边界以外属于其他领域。领域内的业务可以继续划分为子域，子域内部的业务元素内聚性高，彼此联系紧密，与子域外部的元素耦合性低，仅存在少量确定的关联关系。子域仍可以继续划分，直到分解为单一职责的业务单元。子域按照在业务中的作用可以分为“核心子域”“支撑子域”和“通用子域”。

- “核心子域”是业务的核心部分，具有很强的个性化业务。
- “通用子域”可以在很多业务中使用，如“用户管理”就是常见的通用子域，用于管理用户信息，可以和很多“核心子域”一起使用。
- “支撑子域”对系统起支撑作用，为核心子域提供必要的服务。

子域划分完成，表示对问题域的分解基本完成，需要使用恰当的解决方案处理这些问题，这就是限界上下文的划分。一个限界上下文中可以包含若干子域，但最好是一个限界上下文只包含一个子域。在限界上下文中，业务元素的名称是唯一的，与其他限界上下文中同名的业务元素表示不同的业务概念。限界上下文规定了业务范围，并将这个范围带到领域模型中，从而贯穿项目的始终。在限界上下文中，使用通用语言描述业务，通用语言需要与领域模型保持一致。领域模型是对象模型，既包括功能也包括数据，是功能模型和信息模型的统一。

现在来分析一下诗词游戏的需求，确定这个应用的子域和限界上下文。

根据 2.1 节的需求描述，可以初步确定系统所涉及的要素：首先是游戏，然后是参与游戏的用户，也就是玩家。由于游戏限定为诗词的文字游戏，因此诗词也是一个要素。可以使用思维导图的方式列出这些要素的关系，具体如图 2-1 所示。

从图 2-1 中可以看出，“用户子域”是通用子域，因为涉及的用户管理和认证可以使用第三方的应用实现，同时也是一个支撑子域，为“游戏子域”和“社交子域”提供支撑；“游戏子域”和“社交子域”是核心子域，是系统的核心业务部分；“诗词服务”和“游戏管理”属于支撑子域，为核心子域提供支撑。

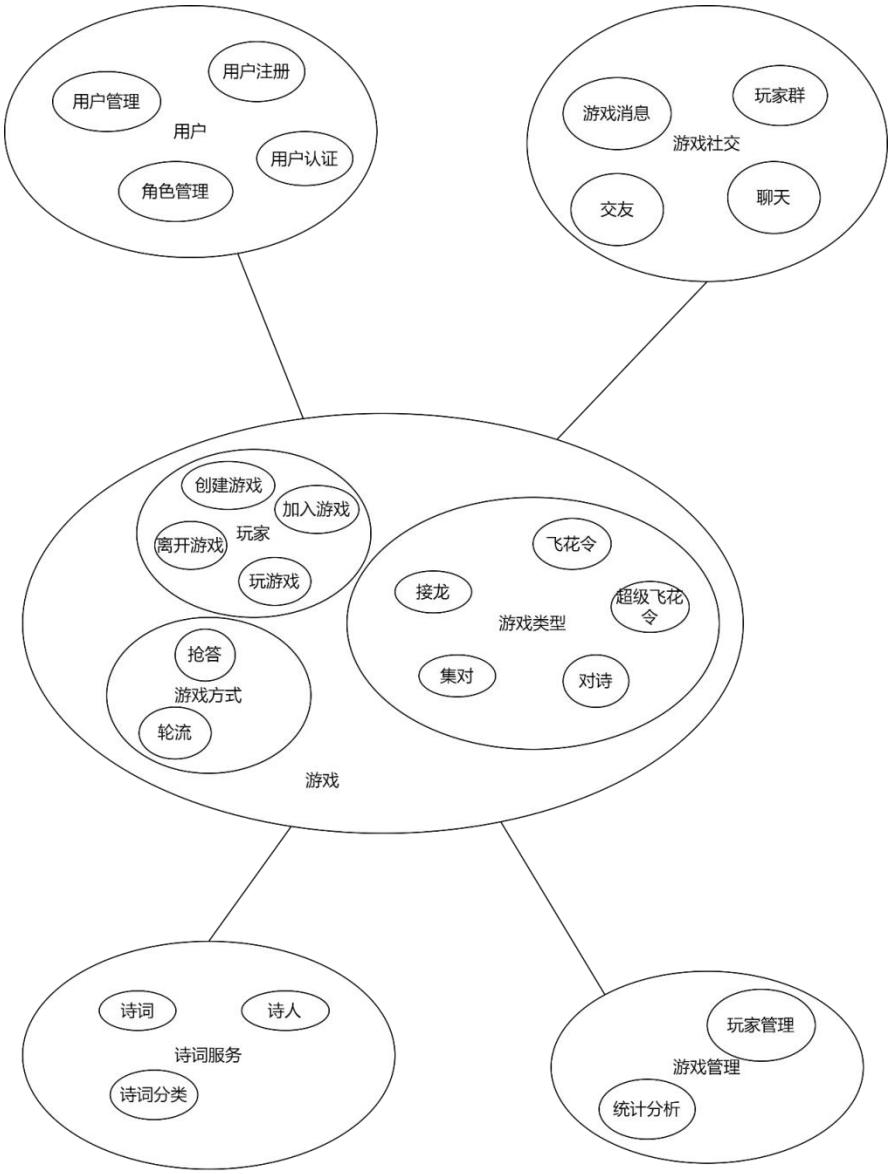

图 2-1　子域的初步划分

2.3　限界上下文的初步确定

在系统构建之初，前面通过对各种涉及的要素进行领域划分，可以帮助我们进行思考，但这毕竟不是语义严格的模型，而且我们的目标系统比较简单，需求易于理解，采用这种方式得出的结果偏差不会太大。在实际中，这种方法只能用在项目初期，用于抛砖引玉。随着对业务理解的加深，需要使用更具体的模型来进行分析。

回到诗词游戏，可以使用类图来描述概念模型，如图 2-2 所示。

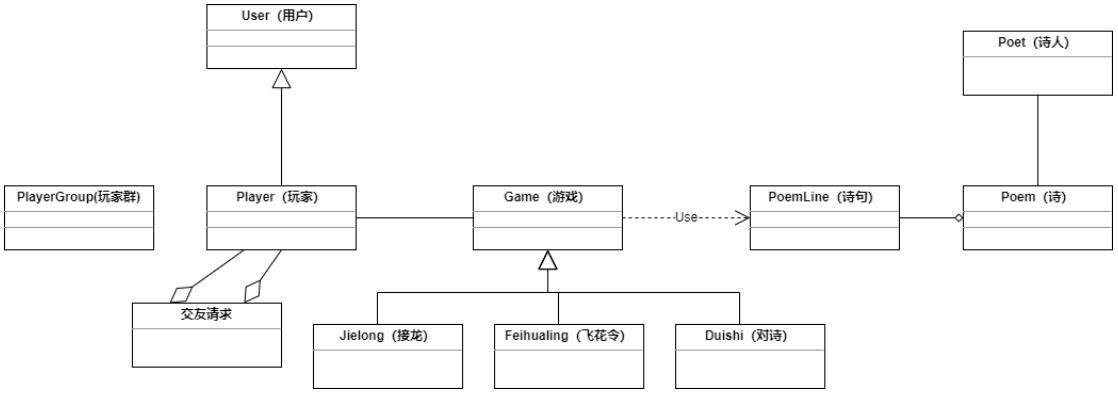

图 2-2　简化的概念模型

上面的类图是初步的概念模型，将系统中的业务要素以类图的方式进行描述，确定这些要素之间的关系。由于是概念模型，不需要细化到属性和方法，因此只需描述出关键概念之间的关系即可。

现在结合需求对初步的概念模型进行分析。可以将思维导图中对系统的直观划分方法应用到初步概念模型中，在图中设置边界，如图 2-3 所示。

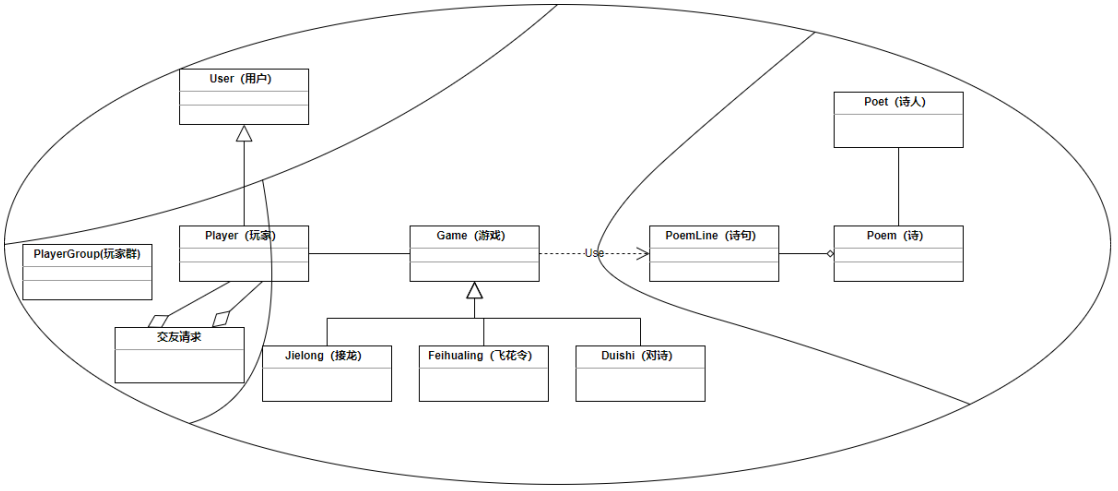

图 2-3　在概念模型上增加边界

从上面的分析模型可以看出，在游戏和社交中，不需要关心玩家的来历，换句话说，即使没有玩家和用户之间的关系，游戏和社交也照样可以存在。如果去掉玩家和用户之间的关系会如何呢？没有太大的影响，因此可以通过一套独立的系统完成用户注册登录，然后采用某种方式将用户映射到游戏系统的玩家。这样，就得到了一个独立的限界上下文，可以称之为用户认证上下文。继续分析会发现，游戏中的玩家和社交中的玩家具有不同的特性和方法，在游戏中，需要改变玩家的积分和等级，而在社交中，会有玩家的朋友、参与的群组等。因此，需要将游戏和社交分开为两个不同的限界上下文，在这两个上下文中都存在玩家，但含义却不相同。最后看一下诗词和游戏的关系，进一步分析发现，诗词只是文字形式的一种，如果将诗词换为成语或者其他文字形式，游戏的规则和结构没有太大的变化。因此，诗词服务也是一个独立的上下文。划分后的上下文以及它们之间的关系如图 2-4 所示。

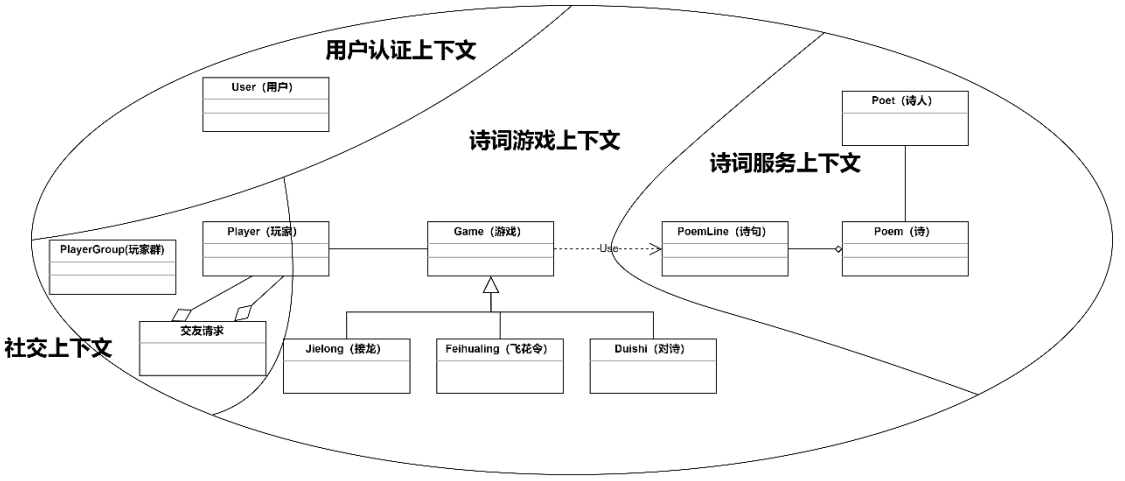

图 2-4　初步划分的限界上下文

2.3.1　用户认证上下文

从上面的分析可以看出，用户认证可以作为独立的上下文存在，用户认证作为独立的系统进行部署，可以完成用户注册、第三方平台登录、认证、用户管理等功能。用户认证系统与游戏系统可以不在同一个进程。当玩家访问游戏系统时，首先要重定位到用户认证系统进行认证，认证完成后，返回到游戏系统。如果用户在游戏系统中不存在，则在游戏系统中创建相同用户名的玩家。认证过程如图 2-5 所示。

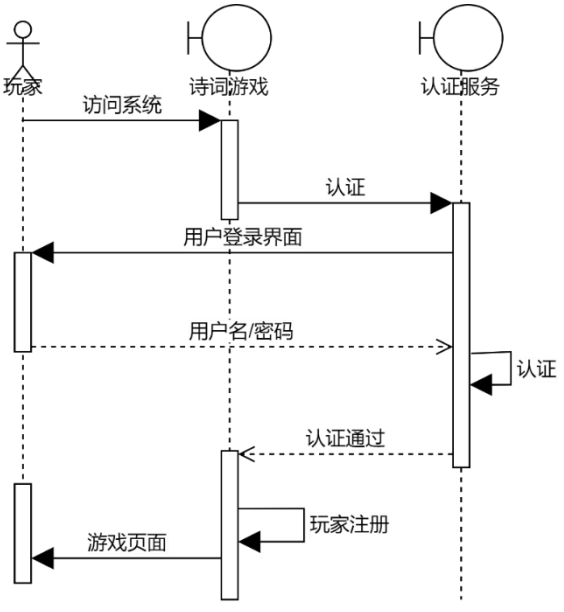

图 2-5　用户认证过程

第三方认证系统只要满足上面的要求，就都可以作为用户认证上下文的实现集成到系统中。

2.3.2　诗词游戏上下文

诗词游戏上下文是系统的核心限界上下文，也是需要重点关注的部分。后面各章主要以诗词游戏上下文为例，说明领域模型的创建与验证。

社交上下文所涉及的主要是用户之间的关系，可以认为是与诗词游戏上下文平行的部分，不作为本书重点。

2.3.3　诗词服务上下文

前面已经分析了，诗词在这里只是一种结构化文字，只是作为游戏进行的条件。因此，只要规定了访问接口，通过什么方式获得诗词的内容对于游戏来说就不重要了，可以从 XML 文件、数据库或者远程主机访问获得。诗词游戏不需要关心如何维护和管理这些诗词。本书示例使用《全唐诗》作为游戏的文字来源，保存在 XML 文件和数据库中，诗词数据的维护和管理可以认为是与诗词服务无关的独立部分，属于另一个限界上下文。

2.3.4　游戏管理上下文

诗词游戏在运行过程中会逐渐积累数据，进而产生对这些数据进行查询、统计和管理的需求。玩家希望查询自己参与的游戏过程，数据分析师希望统计哪些诗句被引用得最多，管理员希望可以限制恶意玩家、删除过时的游戏等。积累的数据本身就是财富，随着时间的推移，这些功能会越来越重要。游戏管理上下文依赖诗词游戏上下文，是诗词游戏上下文的下游。

2.4　限界上下文映射

前面提到了，领域驱动设计将领域内的业务划分为子域，逐级划分后，形成若干相对独立的限界上下文，这些限界上下文可以独立进行编程开发。然而，用户最终需要的是作为整体运行的系统，而不是一个个孤立的软件模块。因此，这些相对独立的限界上下文必须集成起来，才能形成有机的系统。限界上下文之间的集成方式在领域驱动设计中叫作限界上下文映射。

可以使用图形的方式表示限界上下文之间的映射，如图 2-6 所示，使用 U 代表上游，使用 D 代表下游。

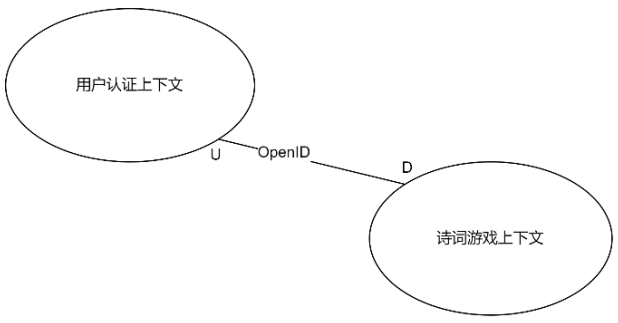

图 2-6　限界上下文之间的关系

在图中为什么使用“上游（U）”和“下游（D）”，而不使用箭头表示呢？因为带箭头的线段在各种模型中有不同的含义，很容易引起歧义。比如，在 UML[23]中，带箭头的实线表示引用关系，带箭头的虚线表示依赖关系，如果箭头是三角形，那么实线就是继承关系，虚线就是实现关系；在数据流图中，箭头方向表示数据的流向；在流程图中，箭头方向表示执行过程。在限界上下文关系图中，如果使用带箭头的线段，则很容易产生歧义，有人可能认为是依赖关系，有人可能认为是数据流向。使用明示的标记可用避免歧义的产生。

需要注意的是，限界上下文映射不仅是指生成的软件产品之间的集成关系，还包括负责上下文开发的团队之间的集成关系和使用的通用语言之间的转换关系。由于开发过程是动态的，限界上下文的映射也不是一成不变的，随着开发过程的进行，限界上下文的映射关系会发生变化，为了避免映射关系变化带来的不一致问题，需要使用持续集成对系统进行验证。在使用限界上下文的映射关系时，一方面要了解限界上下文对应的软件产品之间的集成关系，另一方面也要了解负责限界上下文开发的团队之间的关系。常见的限界上下文映射有如下一些关系，映射模式的名称来源于文献[3]。

2.4.1 各行其道

各行其道（或者叫作各自为政）方式下，各个限界上下文是独立的，不互相集成。上下文之间如果需要数据交互，则需要通过人工完成。导致这种情况的原因很多，原因之一是安全等非技术因素，一个最典型的实例是物理隔离的内网和外网系统之间的集成。有很多组织出于安全等考虑，需要内网系统和外网系统物理隔离，不能互相访问。这种情况下，如果数据需要集成，就只能采用人工导入导出的方式进行。

2.4.2 已发布语言

已发布语言可以是某种协议或者标准，也可以是某种 DSL（领域特定语言）。不同的限界上下文之间可以根据已发布语言开发与自己内部的通用语言对应的转换工具。已发布语言的形式有很多种，可以是某种格式化的文档，比如 XML 格式、JSON 格式的数据交换文件，也可以是某种协议，或者是 C#编写的 DSL 类库。

在诗词游戏示例中，认证服务采用 OIDC 协议，这就是一种已发布语言。因为认证的过程以及在这个过程中传递的数据格式都符合协议规定，所以只要符合协议的要求，就可以设计和编写相关的代码。

在实际项目中，应尽量采用“已发布语言”实现限界上下文之间的映射。采用通用的标准协议作为“已发布语言”是最优选择。也可以制订组织内部的“已发布语言”在开发中加以应用。

“已发布语言”通常与“开放主机服务”和“客户–供应商”模式一起使用。

2.4.3 开放主机服务

开放主机服务指限界上下文定义了一套协议或者接口，使其可以被当作服务来使用。这个协议是“开放的”，有详细的说明文档，很容易被其他上下文使用。开放主机服务经常使用 Web API 或者 gRPC 等方式提供服务。开放主机服务也经常同已发布语言一起组成对外服务。

在诗词游戏示例中，诗词服务可以采用开放主机服务模式，提供基于 Web API 或者 gRPC 的诗

词查询接口。

在开放主机服务中，提供服务的一方规定了访问协议和数据格式。比如诗词服务 Web API 提供了根据诗句进行查询的功能，这个功能的接口和返回的数据格式是诗词服务上下文确定的，使用这个功能的其他限界上下文需要访问这个接口，并能够解析接收的数据。为了减少下游上下文对上游的依赖，可以创建“防腐层”，将从服务中获取的数据转换为需要的格式。这样，当上游发生变化时，只会影响到“防腐层”，不会影响下游的其他部分。

2.4.4　客户-供应商

客户-供应商描述的也是一种上下游的关系，客户向供应商定制需要的内容，供应商可以根据定制发布客户需要的内容。与开放主机服务不同的是，客户不是从供应商主动获取内容，而是供应商向客户推送内容。

在诗词游戏中，游戏上下文和社交上下文之间就是客户-供应商关系。诗词游戏中产生的各种消息（游戏创建、玩家加入游戏、玩家进行游戏等），通过消息中间件发布给社交上下文中的“游戏信息动态”。所发布的内容由游戏上下文决定，但内容的格式等需要两个上下文的团队协商制定。

如果客户无法参与决定供应商发布的内容格式，那么客户就成为供应商的“跟随者”。为了减少客户对供应商的依赖，就需要设置“防腐层”。

2.4.5　跟随者

跟随者也是指上游和下游之间的关系，但规则由上游制定，下游必须遵守，不存在商量的余地。使用第三方服务和产品一般都是这种情况。需要注意的是，使用“开放主机服务”或者“客户-供应商”模式时，下游都有可能是上游的“跟随者”。

诗词游戏上下文和游戏管理上下文之间就是这种关系。游戏管理的数据完全来源于诗词游戏，诗词游戏的类型、条件、参与方式等决定了游戏的数据结构，当有新的游戏类型加入时，就会有新的游戏数据结构。向游戏管理上下文提供什么样的数据，完全由诗词游戏上下文决定，游戏管理完全是被动的，所以游戏管理是诗词游戏的跟随者。

2.4.6　防腐层

在跟随者模式中，如果使用的第三方系统发生了变化，或者希望替换第三方系统，那么现有的部分势必需要更改。为了控制这种情况，将可能的更改限制在最小的范围内，可以使用“防腐层”模式，在下游与上游上下文之间建立一套翻译机制，将上游上下文的通用语言翻译为本上下游的通用语言，这种翻译机制就是防腐层。如果上游发生了变化，那么只需改变这种翻译机制就可以了。

2.4.7　合作方式

合作方式是指两个上下文之间存在合作关系，表现在工作内容上的重叠。这种关系有可能是由于上下文划分不合理导致的，也有可能是组织工作分配不合适导致的。总之，如果两个上下文之间存在合作方式，那么就需要考虑是否将这两个上下文合并，或者以某种方式进行重新划分。在很多情况下，合作方式可以由其他低耦合的方式实现。

2.4.8 共享内核

共享内核指不同的上下文共用领域模型的一部分，也可以认为是两个上下文中有领域模型重叠的部分。很多情况下，这是合作方式的一种解决方案：将需要合作的部分识别出来，以共享内核的方式存在。虽然共享内核导致了两个上下文之间的耦合，但还是为两个上下文之间的合作给出了明确的定义。

2.5 诗词游戏上下文的通用语言

现在可以创建诗词游戏上下文的通用语言了，规定在这个上下文中所涉及的词汇的含义。这些词汇是构建领域模型中各种元素的基础，并且和领域模型一致。也就是说，如果领域模型进行了改进，所涉及的通用语言的词汇也需要改进；反过来，如果通用语言发生了变化，领域模型也需要进行相应的调整。由于通用语言使用中文，而领域模型使用英语或者拼音，因此需要将这些词汇进行对应，形成词表。在关键的描述中，需要在词汇后面加上带括号的说明，例如“玩家（Player）”。我们尝试将需求使用通用语言进行描述，在描述中尽量使用英文，但对于无法翻译为英文的，比如飞花令，使用汉语拼音；某些词可以翻译为英文，比如“接龙”和“对诗”，但这些英文词过于生僻，不好理解，所以也使用拼音；某些词比如“抢答”，采用缩写 FCFA（First Come, First Answer）。

我们使用 PoemGame 表示诗词游戏上下文，在这个上下文中，游戏（Game）有多个玩家（Player）参与，游戏规则由游戏的类型（GameType）和条件（GameCondition）确定，游戏类型（GameType）包括飞花令（Feihualing）、接龙（Jielong）和对诗（Duishi），游戏类型应该可以扩展。游戏说明如下：

- 飞花令（Feihualing）：在游戏条件（GameCondition）中指定了作答（Answer）中必须包含的字或词。例如，如果在游戏条件（GameCondition）中的设置为“花”，那么“花间一壶酒”“黄四娘家花满蹊”都是符合要求的作答。在同一游戏中，答案（Answer）不能重复。
- 接龙（Jielong）：接龙就是上一句的尾是下一句的开头。例如“小时不识月”可以接“月里青山淡如画”。在游戏条件（GameCondition）中指定了起头的诗词，这里的游戏条件（GameCondition）是“小时不识月”，在同一游戏中，答案（Answer）不能重复。
- 对诗（Duishi）：对诗就是“上句接下句”。在游戏条件（GameCondition）中设置第一句。比如，游戏条件（GameCondition）是“床前明月光”，那么答案（Answer）应该是“疑是地上霜”。

由于在很多游戏类型中都需要判断玩家的作答是否重复，因此需要使用游戏记录（PlayRecord）来保存游戏的过程。

玩家（Player）可以加入游戏（JoinGame）或者离开游戏（LeaveGame）。玩家在游戏中轮流（Inturn）作答（Play）或者抢答（FCFA），在游戏创建时确定该游戏采用轮流作答还是抢答的方式。如果是人-计算机形式的单机游戏，就只有轮流作答方式。

诗词游戏上下文通过访问诗词服务上下文（PoemService）对答案进行判断，不同类型的游戏判断的算法不同。

现在有了初步的通用语言，下一步在此基础上创建领域模型的第一个版本。

2.6　创建第一个版本

现在创建第一个版本，这个版本中没有引入任何领域驱动设计的概念，只是按照面向对象的基本方法进行设计。

使用 Visual Studio 2002 创建一个空的解决方案，在解决方案中添加两个解决方案文件夹，名称分别为 PoemGame 和 PoemService，用来保存游戏上下文和诗词服务上下文的相关项目；在这两个文件夹中创建 tests 文件夹，用来保存测试项目。

在 PoemGame 中创建类库 PoemGame.Domain 和 PoemGame.Domain.Feihualing，这两个类库分别为游戏的基础模型和针对飞花令的模型。在 PoemService 中创建 PoemService.Shared，在这里定义诗词服务的接口，然后创建 PoemService.Xml 项目，使用 XML 数据源实现诗词服务接口。最后，创建一个控制台应用，名称为 PoemGame.Console，用来模拟游戏过程。项目的结构如图 2-7 所示。

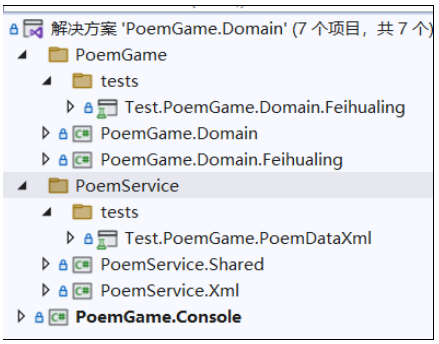

图 2-7　第一版诗词游戏解决方案的结构

在 PoemGame.Domain 中定义了 Game 和 Player，在 Game 中实现游戏的过程，Game 的代码架构如下：

```
public abstract class Game
{
    public Guid Id { get; set; }
    private List<Player> Players { get; set; } = new List<Player>();
    private List<Player> ActivatePlayers = new List<Player>();

    protected List<PlayRecord> Records { get; set; } = new List<PlayRecord>();
    public GameStatus Status { get; private set; }
    public string GameCondition { get; set; }

    public Player GetCurrentPlayer()
    {
        return ActivatePlayers[currentPlayerIndex];
    }

    private int currentPlayerIndex;
```

```
public Game()
{
    Status = GameStatus.Ready;
}
public void AddPlayer(Player player)
{
    if (Status != GameStatus.Ready) throw new Exception("不能加入");
    Players.Add(player);
    ActivatePlayers.Add(player);
}


public GamePlayResult Play(string answer)
{
    var res=new GamePlayResult();

    if (string.IsNullOrEmpty(answer))
    {
        res.Message = "不能输入空值，请重新输入";
        res.IsAnswerCorrect = false;
        return res;

    }
    var IsProperly = CheckAnswer(answer);
    var record = new PlayRecord
    {
        Answer = answer,
        PlayDateTime = DateTime.Now,
        PlayerId = GetCurrentPlayer().Id
    };
    Records.Add(record);
    if (IsProperly)
    {
        currentPlayerIndex++;
        if (currentPlayerIndex == ActivatePlayers.Count)
                                currentPlayerIndex = 0;
        res.Message = "回答正确";
    }
    else
    {
        var current = GetCurrentPlayer();
        ActivatePlayers.Remove(current);

        if (ActivatePlayers.Count == 1)
        {
            currentPlayerIndex = 0;
            Status= GameStatus.Done;
        }
        else if (currentPlayerIndex == ActivatePlayers.Count)
            currentPlayerIndex = 0;
```

```
                res.Message = current.UserName + "出局,";
            }
            return res;
        }
        protected abstract bool CheckAnswer(string answer);
    }
```

在 Game 中保存了参与游戏的玩家，根据玩家的加入顺序确定回答问题的顺序。在游戏进行过程中，将玩家的答案保存在 PlayRecord 中。具体的游戏类中实现了判断答案是否正确的方法（CheckAnswer）。在这个例子中，FeihualingGame 实现了 CheckAnswer 方法：

```
    public class FeihualingGame : Game
    {
        private readonly IPoemService poemService;
        public FeihualingGame(IPoemService _poemService,string gamecontext)
        {
            GameCondition = gamecontext;
            poemService = _poemService;
        }
        protected override bool CheckAnswer(string answer)
        {
            foreach(var record in Records)
            {
                if (record.Answer == answer) return false;
            }
            if (!poemService.IsPoemLineExist(answer)
           || !answer.Contains(GameCondition)) return false;
            return true;
        }

    }
```

判断方法很简单——在诗词库中有符合要求的答案并且答案不能重复。CheckAnswer 方法中使用诗词服务来判断诗句是否存在。这个版本的诗词服务的定义也很简单，只有判断诗句是否存在这一个方法：

```
    public interface IPoemService
    {
        bool IsPoemLineExist(string line);
     }
```

这里省略了诗词服务的实现，留待下一章介绍。

Player 的代码如下：

```
    public class Player
    {
        public Guid Id { get; set; }
        public string UserName { get; set; }
        public string NickName { get; set; }
        public int Score { get; set; }
        public Guid? CurrentGameId { get; private set; }
```

```
        public void JoinGame(Game generalGame)
        {
            if (generalGame.Status != GameStatus.Ready) throw new Exception("不能加入已开始或已完成的游戏");

            CurrentGameId = generalGame.Id;

            generalGame.AddPlayer(this);
        }
    }
```

在控制台中编写模拟代码对这个模型进行测试：

```
var game = new FeihualingGame(new PoemServiceXml(), "花");

var player1 = new Player { UserName = "张三" };
var player2 = new Player { UserName = "李四" };
var player3 = new Player { UserName = "王五" };

player1.JoinGame(game);
player2.JoinGame(game);
player3.JoinGame(game);

var note = "请输入一句唐诗，带“花”";
Console.WriteLine(note);
do
{
    Console.WriteLine("当前用户:"+game.GetCurrentPlayer().UserName);
    var answer = Console.ReadLine();
    var res = game.Play(answer);
    Console.WriteLine(res.Message);
} while (game.Status != GameStatus.Done);
Console.WriteLine("赢家:" + game.GetCurrentPlayer().UserName);
Console.WriteLine("游戏结束");
```

运行效果如图 2-8 所示。

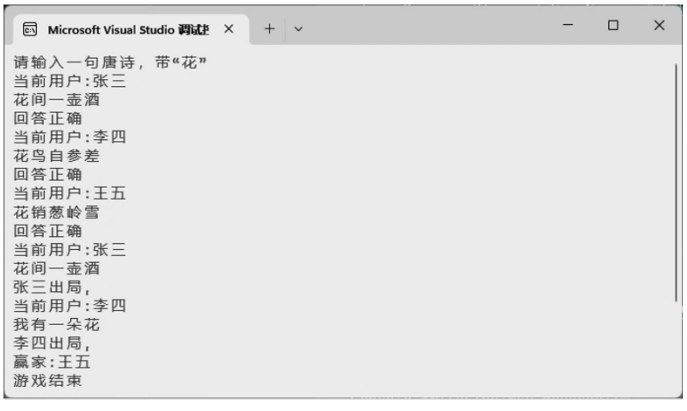

图 2-8　模拟游戏运行效果

现在看一下 PoemGame.Domain 中定义的类，分别是 Game、Player、PlayRecord 和 GamePlayResult，从形式上看，它们都是 C#的 class 类型，但从作用上看似乎又有些区别：GamePlayResult 的作用是输出结果，只有属性并没有方法；PlayRecord 用于记录游戏的过程，由于历史是不能修改的，因此 PlayRecord 的实例一旦创建就不能修改，并且游戏的记录与游戏密切相关，只能通过游戏访问；Game 和 Player 具有属性和方法，并且在运行时状态会发生改变。如果模型变得复杂，就有必要引入新的概念为不同的类型进行命名，便于模型的理解，这些概念在领域驱动设计中有相应的定义。在后面各章中，会引入领域模型的概念，在这个项目的基础上，进行重构和迭代。

2.7　本 章 小 结

领域、子域、限界上下文这些概念属于领域驱动设计的战略设计部分。“战略”这个词总是感觉有些大，对应系统开发时常用的术语应该是“总体设计”“总体规划”或者诸如此类的东西。到了限界上下文内部，就属于战术设计的范畴了。不管叫什么，这个阶段所解决的问题是一样的，就是系统以某种形式拆解为可以由小团队独立开发的部分，并能够将这些部分以某种形式集合起来，完成系统所需要的所有功能。在领域驱动设计中，划分的结果是子域，子域可以分为“核心子域”“支撑子域”和“通用子域”。一般情况下，“通用子域”一定是“支撑子域”。子域和限界上下文的划分往往凭经验进行，没有可以操作的简单方法，本章提出了一种在概念建模的基础上进行限界上下文划分的方法。需要注意的是，子域和限界上下文的划分是动态的迭代过程，因为不管使用什么方法划分，都有合理性和不合理性，需要在后续的开发中，随着对业务的深入理解而进行调整。不会有一开始就有的、完美的、解决一切的方案，只要大的方向不错，其他的部分可以在迭代中不断完善。