

# 领域驱动设计浅谈

---

小 组： 官网商城开发组

分 享 人： 袁 超

# CONTENTS

01

**什么是领域驱动设计**

02

**为什么需要领域驱动设计**

03

**领域驱动设计的战略设计**

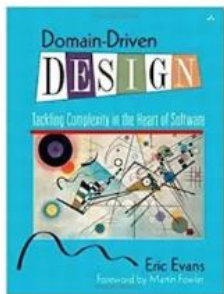
04

**领域驱动设计的战术设计**

05

**总结**

# DDD简史



2003.08(2016.02)  
领域驱动设计



2013.02  
实现领域驱动设计



2015.04  
领域驱动设计设计模式  
原理与实践

**2003** 年 Eric Evans 发表《Domain-Driven Design –Tackling Complexity in the Heart of Software》（**领域驱动设计**）简称 Evans DDD；

**2013** 年,Vaughn Vernon 发表《**实现领域驱动设计**》、《**领域驱动设计精粹**》；

**2015**年， Jimmy Nilsson 结合了《领域驱动设计》和《企业应用架构模式》，写了《**领域驱动设计与模式实战**》；

# 什么是领域驱动设计

一种模型驱动的软件设计方式



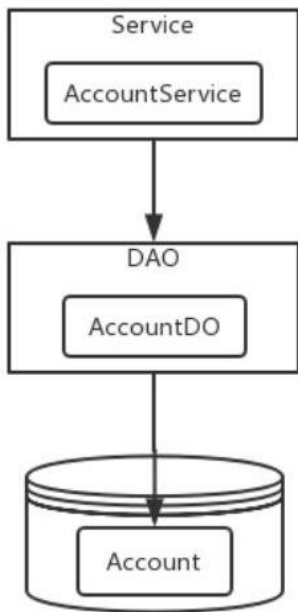
领域就是**问题域**，有边界，领域中有很多问题;

通过建立领域模型来解决领域中的核心问题，模型驱动的思想;

通过领域模型驱动代码的实现，确保代码让领域模型落地，代码最终能解决问题；

## 事务脚本（Transaction Script、贫血模型）的编程模型

事务脚本就是经典的Service-->DAO-->DB的编程模式，传统J2EE等事务性编程模型只关心数据，这些数据对象除了简单setter/getter方法外，没有任何业务方法，也被比喻成贫血模型，所有的业务逻辑都不包含在内而是放在Business Logic层。



事务脚本实现很简单，不需要对业务进行深入的抽象，也不用领域建模，比较适合简单的业务场景，但是对于复杂业务场景，事务脚本将会使代码变得很复杂，难以维护。

以银行转账为例，在事务脚本的实现中，其转账的业务逻辑都被写在了AccountService的实现里面了，而Account仅仅是getters和setters的贫血数据结构

## DDD中的编程模型

- 采用DDD的设计思想，业务逻辑不再集中在几个大型的类上，而是由大量相对小的领域对象(类)组成，这些类具备自己的状态和行为，每个类是相对完整的独立体，并与现实领域的业务对象映射。
- 领域模型就是由这样许多的细粒度的类组成。基于领域驱动的设计，保证了系统的可维护性、扩展性和复用性，在处理复杂业务逻辑方面有着先天的优势。

以银行账号Account为案例，Account有“存款”和“取款”等业务行为，如果用DDD的方式实现，Account实体除了账号属性之外，还包含了行为和业务逻辑，比如包含“存款”credit()和“取款”debit()方法；

```
public class Account {  
    // @Id  
    private String id;  
    private double balance;  
    private OverdraftPolicy overdraftPolicy;  
    . . .  
    public double balance() { return balance; }  
    public void debit(double amount) {  
        this.overdraftPolicy.preDebit(this, amount);  
        this.balance = this.balance - amount;  
        this.overdraftPolicy.postDebit(this, amount);  
    }  
    public void credit(double amount) {  
        this.balance = this.balance + amount;  
    }  
}
```

## DDDD中领域模型的编程模型

透支策略OverdraftPolicy被抽象成包含了业务规则并采用了策略模式的对象

```
public interface OverdraftPolicy {
    void preDebit(Account account, double amount);
    void postDebit(Account account, double amount);
}

public class NoOverdraftAllowed implements OverdraftPolicy {
    public void preDebit(Account account, double amount) {
        double newBalance = account.balance() - amount;
        if (newBalance < 0) {
            throw new DebitException("Insufficient funds");
        }
    }
    public void postDebit(Account account, double amount) {
    }
}

public class LimitedOverdraft implements OverdraftPolicy {
    private double limit;
    . . .
    public void preDebit(Account account, double amount) {
        double newBalance = account.balance() - amount;
        if (newBalance < -limit) {
            throw new DebitException(
                "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
        }
    }
    public void postDebit(Account account, double amount) {
```

Domain Service只需要调用Domain Entity对象完成业务逻辑

```
public class MoneyTransferServiceDomainModelImpl
    implements MoneyTransferService {
    private AccountRepository accountRepository;
    private BankingTransactionRepository bankingTransactionRepository;
    . . .
    @Override
    public BankingTransaction transfer(
        String fromAccountId, String toAccountId, double amount) {
        Account fromAccount = accountRepository.findById(fromAccountId);
        Account toAccount = accountRepository.findById(toAccountId);
        . . .
        fromAccount.debit(amount);
        toAccount.credit(amount);
        BankingTransaction moneyTransferTransaction =
            new MoneyTransferTransaction(fromAccountId, toAccountId, amount);
        bankingTransactionRepository.addTransaction(moneyTransferTransaction);
        return moneyTransferTransaction;
    }
}
```



---

## 二，为什么要关注领域驱动设计



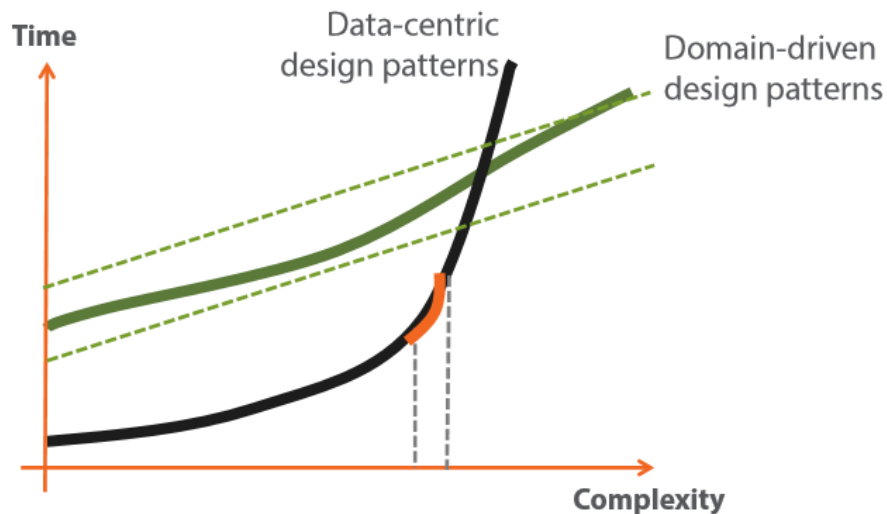


## 我们为什么关注领域驱动设计---软件系统复杂性应对

从EricEvans的《领域驱动设计：软件核心复杂应对之道》一书的书名就可以看出这一方法论是为了解决软件核心复杂性的

解决复杂和大规模软件的武器可以被粗略地归为：  
抽象、分治

DDD提供了这样的手段，让我们知道  
如何抽象出限界上下文以及如何去分治



NOTE: Adapted from Martin Fowler's PoEAA



### **三. DDD的战略设计**

# DDD的战略设计

---

DDD有战略设计和战术设计之分；

**战略设计**主要从高层“俯视”我们的软件系统，关注限界上下文的划分和关系；

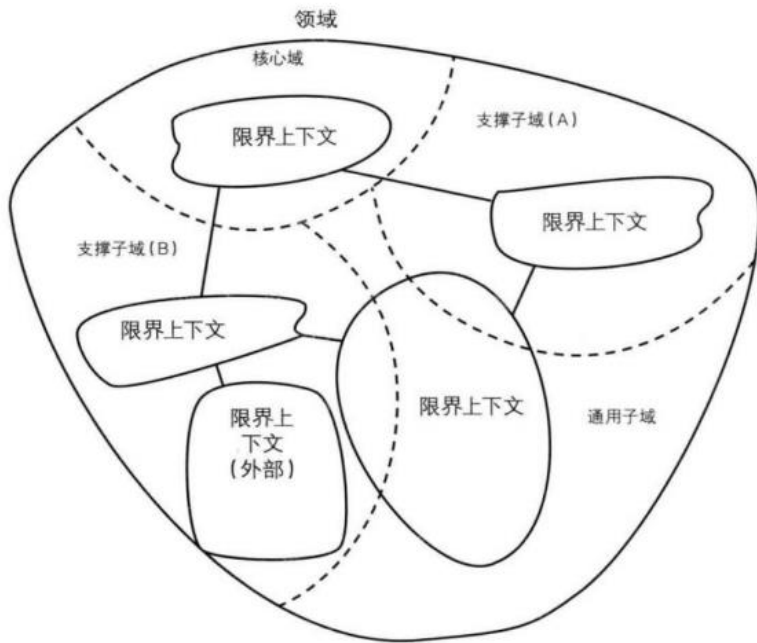
**战略建模-Strategic Modeling：**

**限界上下文** ( Bounded Context )

**上下文映射图** ( Context Mapping )

# 战略建模 --> 限界上下文

限界上下文：限的意思就是划分、规定，界就是界限、或者一个边界，上下文就是业务的整个流程。限界上下文定义了领域模型的边界，限界上下文的目的就是理清子域，然后区分这些子域那些是核心域、支撑子域和通用子域

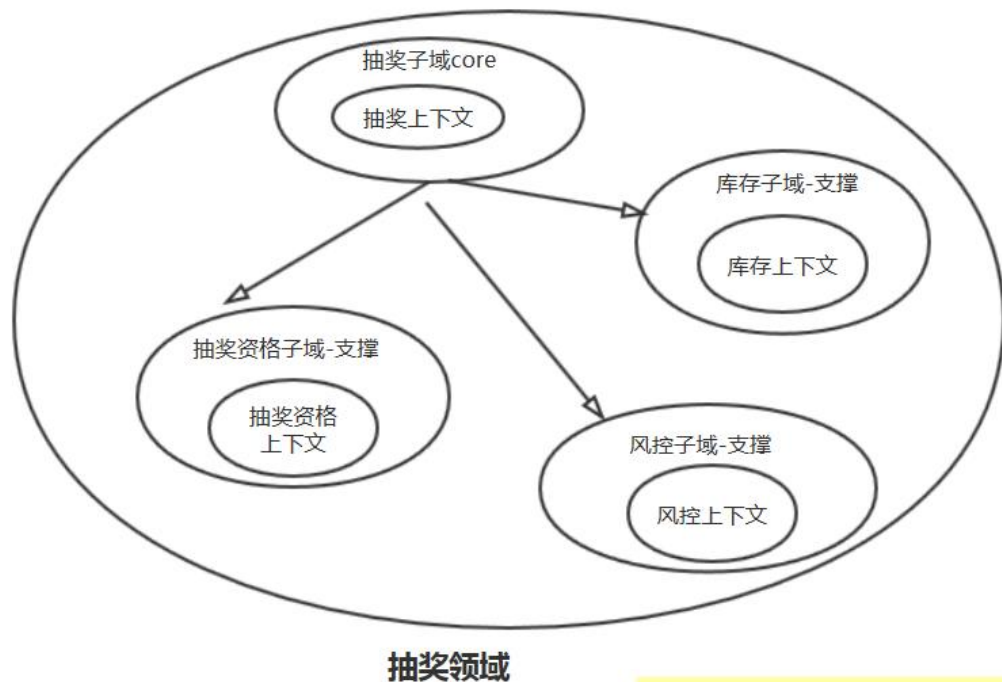


观察角度的不同，限界上下文划定的边界也有所不同，大致分为：

**从领域逻辑的角度**，即根据业务知识进行划分，比如将电商项目划分为订单、商品、促销、物流、售后等限界上下文。

**从技术实现角度**，比如为了解决系统的高并发，决定引入缓存，那就可以考虑将缓存服务抽离为一个独立的限界上下文作为支撑。

## 战略建模 --> 举例-划分限界上下文



抽奖的需求描述如下：

1. 抽奖活动有活动限制，例如用户的抽奖次数限制，抽奖的开始和结束的时间等；
2. 一个抽奖活动包含多个奖品，可以针对一个或多个用户群体；
3. 奖品有自身的奖品配置，例如库存量，被抽中的概率等，最多被一个用户抽中的次数等等；
4. 活动具有风控配置，能够限制用户参与抽奖的频率。

## 战略建模 --> 限界上下文映射图

多个系统之间会发生关系，存在交互，这也必然会在各自的限界上下文有所表现。上下文映射图（Context Map）便是表示各个系统之间关系的总体视图。

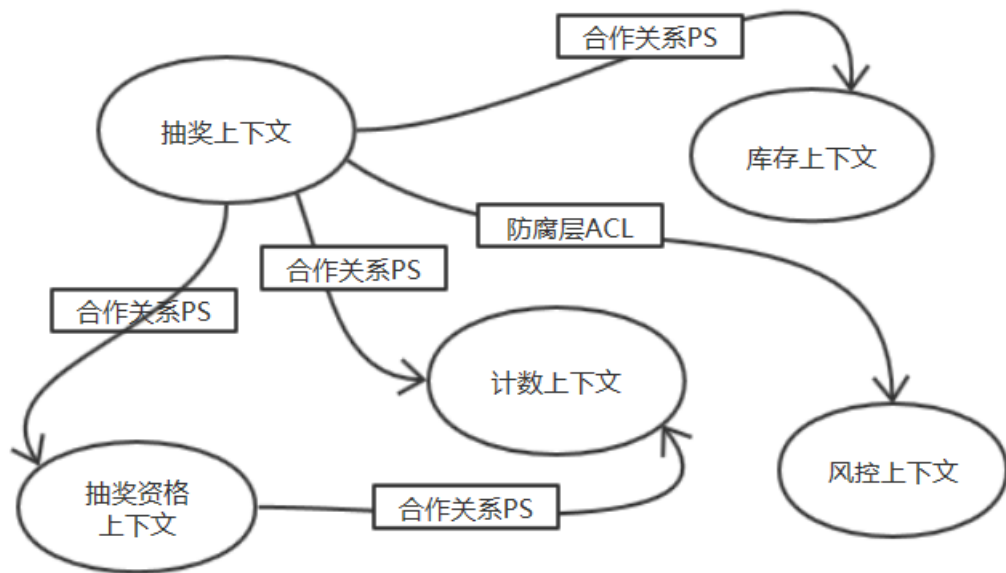
在进行上下文划分之后，我们还需要进一步梳理上下文之间的关系。在实践中，使用得最多的应当是：防腐层 + 公开主机服务的搭配使用；

**合作关系**（Partnership）：两个上下文紧密合作的关系，一荣俱荣，一损俱损。

**防腐层**（Anticorruption Layer）：一个上下文通过一些适配和转换与另一个上下文交互。

**开放主机服务**（Open Host Service）：定义一种协议来让其他上下文来对本上下文进行访问。

## 战略建模 --> 限界上下文映射图举例



使用前面抽奖的例子，进行限界上下文映射；由于抽奖，抽奖资格，奖品，计数五个上下文都处于抽奖领域的内部，之间符合“一荣俱荣，一损俱损”的**合作关系**（Partnership，简称PS）。

抽奖上下文通过防腐层（Anticorruption Layer，ACL）对风控上下文进行了隔离，而风控上下文通过**开放主机服务**（Open Host Service）作为发布语言（Published Language）对抽奖上下文提供访问机制。

## 限界上下文代码设计

在实际开发中，我们一般会采用模块来表示一个领域的界限上下文，模块（Module）是DDD中明确提到的一种控制限界上下文的手段，比如：

```
import com.company.team.bussiness.lottery.*;//抽奖上下文
import com.company.team.bussiness.riskcontrol.*;//风控上下文
import com.company.team.bussiness.qualification.*;//抽奖资格上下文
import com.company.team.bussiness.inventory.*;//库存上下文
```

如代码中所示，一般的工程中包的组织方式为{com.公司名.组织架构.业务.上下文.\*}，这样的组织结构能够明确的将一个上下文限定在包的内部。





## 四， DDD的战术设计

# DDD的战术设计

---

- **战术设计** 关注具体，使用建模工具来细化和填充上下文；

## 战术建模-Tactical Modeling：

- **聚合**-Aggregate
- **实体**-Entity
- **值对象**-Value Objects
- **资源库**-Repository
- **领域服务**-Domain Services

## 领域对象：实体（Entity）

### 实体：

当一个对象由其标识（而不是属性）区分时，这种对象称为实体（Entity），这个标识必须就有不变性和唯一性。

### 实体设计原则：

在定义实体时还需要清楚，哪些行为是属于这个实体的，哪些职责是本实体应该具备的。

### 误区：实体标识与数据库主键区别

数据库主键是一个完全独立的概念，与DDD或域建模无关。  
但实体需要持久保存并跟踪，而数据库中的主键有助于实现系统中所有标识的唯一性，所以通常可以用主键来生成实体标识。

## 领域对象：值对象（Value Object）

### 值对象：

在领域中，并不是每一个事物都必须有一个唯一标识，也就是说我们不关心对象是哪个，而只关心对象是什么，当你只关心某个对象的属性时，该对象便可作为一个值对象。

### 值对象特性：

值对象没有唯一标识，判断是否是同一个对象时是通过它们的所有属性是否相同；  
值对象所有属性都是只读的，可以被安全的共享；需要保证值对象创建后就不能被修改，即不允许外部再修改其属性。

### 误区：

值对象很重要，在习惯了使用数据库的数据建模后，很容易将所有对象看作实体。使用值对象，可以更好地做系统优化、精简设计

## 领域对象：聚合根 ( Aggregate Root )

### 聚合根

Aggregate(聚合) 定义了一组具有内聚关系的相关对象的集合，作为一个整体被外界访问，聚合根 ( Aggregate Root ) 是这个聚合的根节点。聚合由值对象和实体组成。

### 聚合设计原则

聚合 != 大对象，相反聚合应尽量设计的小

聚合是用来封装真正的不变性，而不是简单的将对象组合在一起

## 仓储 ( Repository )

### 仓储：

领域对象需要资源存储，存储的手段可以是多样化的，常见的无非是数据库，分布式缓存，本地缓存等。资源库 ( Repository ) 的作用，就是对领域的存储和访问进行统一管理的对象

### 仓储与DAO：

DAO是关系型数据库和应用之间的契约，它封装了Web应用中的数据库CRUD操作细节；资源库是一个独立的抽象，它与DAO进行交互，并提供到领域模型的业务接口；

## 领域服务 ( Domain Service )

### 领域服务：

领域中的一些概念不太适合归类到实体对象或值对象，因为它们本质上就是一些操作，一些动作。这些操作或动作往往会涉及到多个领域对象，并且需要协调这些领域对象共同完成这个操作或动作

### 理解：

领域服务是无状态的，它存在的意义就是协调领域对象共完成某个操作，所有的状态还是都保存在相应的领域对象中。

## 战术建模 --- 细化上下文

战术建模很重要的一步就是区分出整个限界上下文中的实体、值对象、聚合、领域服务，战术建模其实对应了Entity、ValueObject、Aggregate、Domain Service。

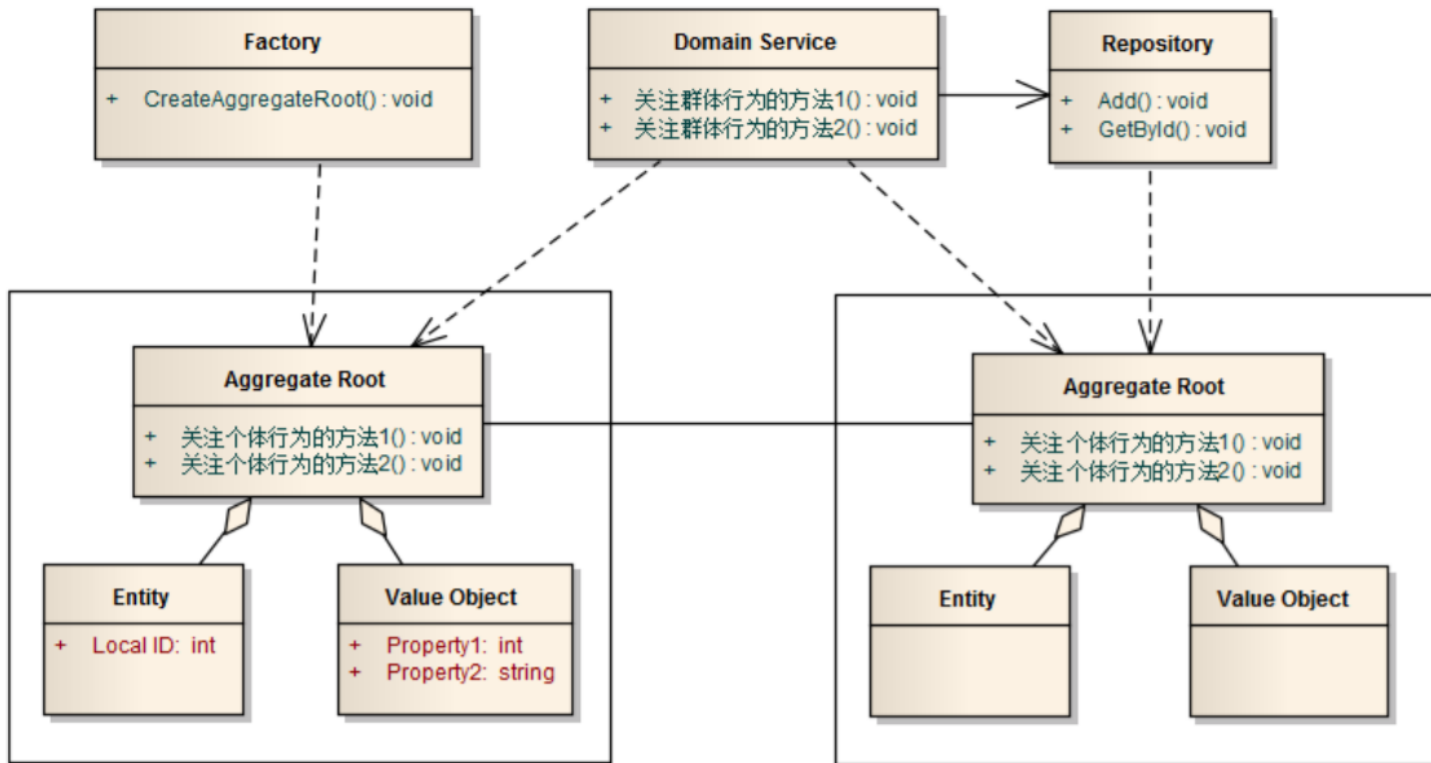
对于模块内的组织结构，一般情况下我们是按照领域对象、领域服务、领域资源库、防腐层等组织方式定义的，如下所示：


```
import com.company.team.bussiness.lottery.domain.valobj.*;//领域对象-值对象
import com.company.team.bussiness.lottery.domain.entity.*;//领域对象-实体
import com.company.team.bussiness.lottery.domain.aggregate.*;//领域对象-聚合根
import com.company.team.bussiness.lottery.service.*;//领域服务
import com.company.team.bussiness.lottery.repo.*;//领域资源库
import com.company.team.bussiness.lottery.facade.*;//领域防腐层
```



# DDD领域模型各构造块关系

DDD领域模型各构造块关系图





## 五，总结

## DDD到底能给工作带来什么

---

- 领域驱动设计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规范过程，使得你的**设计思路能够更加清晰**，设计过程更加规范。
- 领域驱动设计尤其善于处理与领域相关的高复杂度业务的产品研发，通过它可以为你的产品建立一个核心而**稳定的领域模型内核**，有利于领域知识的传递与传承。
- 领域驱动设计的思想、原则与模式有助于提高团队成员的**面向对象设计能力与架构设计能力**。

## 怎么去使用DDD

---

DDD它是针对复杂系统设计的一套软件工程方法：把系统分割为一个个有边界的上下文（Bounded Context）；通过实体、值对象、聚合等来处理领域的信息；通过领域服务与他领域耦合；通过上下文地图来表述系统的宏观结构

DDD是一系列分析方法，在遇到具体情况的时候完全可以借鉴其中一部分，最终目的还是解决问题，至于是不是DDD，并不一定要套用。



# 谢谢

---

Thank you