DDD系列第四讲：领域层设计规范

原创 殷浩 淘系技术 2020-12-28

收录于话题
#后端技术分享

29个

在一个DDD架构设计中，领域层的设计合理性会直接影响整个架构的代码结构以及应用层、基础设施层的设计。但是领域层设计又是有挑战的任务，特别是在一个业务逻辑相对复杂应用中，每一个业务规则是应该放在Entity、ValueObject 还是 DomainService是值得用心思考的，既要避免未来的扩展性差，又要确保不会过度设计导致复杂性。今天我用一个相对轻松易懂的领域做一个案例演示，但在实际业务应用中，无论是交易、营销还是互动，都可以用类似的逻辑来实现。

初探龙与魔法的世界架构

背景和规则

平日里看了好多严肃的业务代码，今天找一个轻松的话题，如何用代码实现一个龙与魔法的游戏世界的（极简）规则?

基础配置如下：

- 玩家（Player）可以是战士（Fighter）、法师（Mage）、龙骑（Dragoon）
- 怪物（Monster）可以是兽人（Orc）、精灵（Elf）、龙（Dragon），怪物有血量
- 武器（Weapon）可以是剑（Sword）、法杖（Staff），武器有攻击力
- 玩家可以装备一个武器，武器攻击可以是物理类型（0），火（1），冰（2）等，武器类型决定伤害类型

攻击规则如下：

1. 兽人对物理攻击伤害减半

2. 精灵对魔法攻击伤害减半
3. 龙对物理和魔法攻击免疫，除非玩家是龙骑，则伤害加倍

OOP实现

对于熟悉Object-Oriented Programming的同学，一个比较简单的实现是通过类的继承关系（此处省略部分非核心代码）：

```
public abstract class Player {
      Weapon weapon
}
public class Fighter extends Player {}
public class Mage extends Player {}
public class Dragoon extends Player {}

public abstract class Monster {
    Long health;
}
public Orc extends Monster {}
public Elf extends Monster {}
public Dragoon extends Monster {}

public abstract class Weapon {
    int damage;
    int damageType; // 0 - physical, 1 - fire, 2 - ice etc.
}
public Sword extends Weapon {}
public Staff extends Weapon {}
```

而实现规则代码如下：

```
public class Player {
    public void attack(Monster monster) {
        monster.receiveDamageBy(weapon, this);
    }
}
```

```
6
7  public class Monster {
8      public void receiveDamageBy(Weapon weapon, Player player) {
9          this.health -= weapon.getDamage(); // 基础规则
10     }
11 }
12
13 public class Orc extends Monster {
14     @Override
15     public void receiveDamageBy(Weapon weapon, Player player) {
16         if (weapon.getDamageType() == 0) {
17             this.setHealth(this.getHealth() - weapon.getDamage() / 2); /
18         } else {
19             super.receiveDamageBy(weapon, player);
20         }
21     }
22 }
23
24 public class Dragon extends Monster {
25     @Override
26     public void receiveDamageBy(Weapon weapon, Player player) {
27         if (player instanceof Dragoon) {
28             this.setHealth(this.getHealth() - weapon.getDamage() * 2); /
29         }
30         // else no damage, 龙免疫力规则
31     }
32 }
```

然后跑几个单测：

```
1  public class BattleTest {
2
3      @Test
4      @DisplayName("Dragon is immune to attacks")
5      public void testDragonImmunity() {
6          // Given
7          Fighter fighter = new Fighter("Hero");
8          Sword sword = new Sword("Excalibur", 10);
```

```
        fighter.setWeapon(sword);
        Dragon dragon = new Dragon("Dragon", 100L);

        // When
        fighter.attack(dragon);

        // Then
        assertThat(dragon.getHealth()).isEqualTo(100);
    }

    @Test
    @DisplayName("Dragoon attack dragon doubles damage")
    public void testDragoonSpecial() {
        // Given
        Dragoon dragoon = new Dragoon("Dragoon");
        Sword sword = new Sword("Excalibur", 10);
        dragoon.setWeapon(sword);
        Dragon dragon = new Dragon("Dragon", 100L);

        // When
        dragoon.attack(dragon);

        // Then
        assertThat(dragon.getHealth()).isEqualTo(100 - 10 * 2);
    }

    @Test
    @DisplayName("Orc should receive half damage from physical weapons"
    public void testFighterOrc() {
        // Given
        Fighter fighter = new Fighter("Hero");
        Sword sword = new Sword("Excalibur", 10);
        fighter.setWeapon(sword);
        Orc orc = new Orc("Orc", 100L);

        // When
        fighter.attack(orc);

        // Then
        assertThat(orc.getHealth()).isEqualTo(100 - 10 / 2);
```

```
    }

    @Test
    @DisplayName("Orc receive full damage from magic attacks")
    public void testMageOrc() {
        // Given
        Mage mage = new Mage("Mage");
        Staff staff = new Staff("Fire Staff", 10);
        mage.setWeapon(staff);
        Orc orc = new Orc("Orc", 100L);

        // When
        mage.attack(orc);

        // Then
        assertThat(orc.getHealth()).isEqualTo(100 - 10);
    }
}
```

以上代码和单测都比较简单，不做多余的解释了。

分析OOP代码的设计缺陷

编程语言的强类型无法承载业务规则

以上的OOP代码可以跑得通，直到我们加一个限制条件：

- 战士只能装备剑
- 法师只能装备法杖

这个规则在Java语言里无法通过强类型来实现，虽然Java有Variable Hiding（或者C#的new class variable），但实际上只是在子类上加了一个新变量，所以会导致以下的问题：

```
@Data
public class Fighter extends Player {
    private Sword weapon;
}

```

```
@Test
public void testEquip() {
    Fighter fighter = new Fighter("Hero");

    Sword sword = new Sword("Sword", 10);
    fighter.setWeapon(sword);

    Staff staff = new Staff("Staff", 10);
    fighter.setWeapon(staff);

    assertThat(fighter.getWeapon()).isInstanceOf(Staff.class); // 错误了
}
```

在最后，虽然代码感觉是setWeapon(Staff)，但实际上只修改了父类的变量，并没有修改子类的变量，所以实际不生效，也不抛异常，但结果是错的。

当然，可以在父类限制setter为protected，但这样就限制了父类的API，极大的降低了灵活性，同时也违背了Liskov substitution principle，即一个父类必须要cast成子类才能使用：

```
@Data
public abstract class Player {
    @Setter(AccessLevel.PROTECTED)
    private Weapon weapon;
}

@Test
public void testCastEquip() {
    Fighter fighter = new Fighter("Hero");

    Sword sword = new Sword("Sword", 10);
    fighter.setWeapon(sword);

    Player player = fighter;
    Staff staff = new Staff("Staff", 10);
    player.setWeapon(staff); // 编译不过，但从API层面上应该开放可用
}
```

最后，如果规则增加一条：

- 战士和法师都能装备匕首（dagger）

BOOM，之前写的强类型代码都废了，需要重构。

对象继承导致代码强依赖父类逻辑，违反开闭原则Open-Closed Principle（OCP）

开闭原则（OCP）规定“对象应该对于扩展开放，对于修改封闭”，继承虽然可以通过子类扩展新的行为，但因为子类可能直接依赖父类的实现，导致一个变更可能会影响所有对象。在这个例子里，如果增加任意一种类型的玩家、怪物或武器，或增加一种规则，都有可能需要修改从父类到子类的所有方法。

比如，如果要增加一个武器类型：狙击枪，能够无视所有防御一击必杀，需要修改的代码包括：

- Weapon
- Player和所有的子类（是否能装备某个武器的判断）
- Monster和所有的子类（伤害计算逻辑）

```
public class Monster {
    public void receiveDamageBy(Weapon weapon, Player player) {
        this.health -= weapon.getDamage(); // 老的基础规则
        if (Weapon instanceof Gun) { // 新的逻辑
            this.setHealth(0);
        }
    }
}

public class Dragon extends Monster {
    public void receiveDamageBy(Weapon weapon, Player player) {
        if (Weapon instanceof Gun) { // 新的逻辑
                    super.receiveDamageBy(weapon, player);
        }
        // 老的逻辑省略
    }
}
```

在一个复杂的软件中为什么会建议“尽量”不要违背OCP？最核心的原因就是一个现有逻辑的变更可能会影响一些原有的代码，导致一些无法预见的影响。这个风险只能通过完整的单元测试覆盖来保障，但在实际开发中很难保障单测的覆盖率。OCP的原则能尽可能的规避这种风险，当新的行为只能通过新的字段/方法来实现时，老代码的行为自然不会变。

继承虽然能Open for extension，但很难做到Closed for modification。所以今天解决OCP的主要方法是通过Composition-over-inheritance，即通过组合来做到扩展性，而不是通过继承。

Player.attack(monster) 还是 Monster.receiveDamage(Weapon, Player)?

在这个例子里，其实业务规则的逻辑到底应该写在哪里是有异议的：当我们去看一个对象和另一个对象之间的交互时，到底是Player去攻击Monster，还是Monster被Player攻击？目前的代码主要将逻辑写在Monster的类中，主要考虑是Monster会受伤降低Health，但如果是Player拿着一把双刃剑会同时伤害自己呢？是不是发现写在Monster类里也有问题？代码写在哪里的原则是什么？

多对象行为类似，导致代码重复

当我们有不同的对象，但又有相同或类似的行为时，OOP会不可避免的导致代码的重复。在这个例子里，如果我们去增加一个“可移动”的行为，需要在Player和Monster类中都增加类似的逻辑：

```
public abstract class Player {
    int x;
    int y;
    void move(int targetX, int targetY) {
        // logic
    }
}

public abstract class Monster {
    int x;
    int y;
    void move(int targetX, int targetY) {
        // logic
    }
}
```

一个可能的解法是有个通用的父类：

```
public abstract class Movable {
    int x;
    int y;
    void move(int targetX, int targetY) {
        // logic
    }
}

public abstract class Player extends Movable;
public abstract class Monster extends Movable;
```

但如果再增加一个跳跃能力Jumpable呢？一个跑步能力Runnable呢？如果Player可以Move和Jump，Monster可以Move和Run，怎么处理继承关系？要知道Java（以及绝大部分语言）是不支持多父类继承的，所以只能通过重复代码来实现。

问题总结

在这个案例里虽然从直觉来看OOP的逻辑很简单，但如果你的业务比较复杂，未来会有大量的业务规则变更时，简单的OOP代码会在后期变成复杂的一团浆糊，逻辑分散在各地，缺少全局视角，各种规则的叠加会触发bug。有没有感觉似曾相识？对的，电商体系里的优惠、交易等链路经常会碰到类似的坑。而这类问题的核心本质在于：

- 业务规则的归属到底是对象的“行为”还是独立的”规则对象“？
- 业务规则之间的关系如何处理？
- 通用“行为”应该如何复用和维护？

在讲DDD的解法前，我们先去看看一套游戏里最近比较火的架构设计，Entity-Component-System（ECS）是如何实现的。

Entity-Component-System（ECS）架构简介

ECS介绍

ECS架构模式是其实是一个很老的游戏架构设计，最早应该能追溯到《地牢围攻》的组件化设计，但最近因为Unity的加入而开始变得流行（比如《守望先锋》就是用的ECS）。要很快的理解ECS架构的价值，我们需要理解一个游戏代码的核心问题：

- 性能：游戏必须要实现一个高的渲染率（60FPS），也就是说整个游戏世界需要在1/60s（大概16ms）内完整更新一次（包括物理引擎、游戏状态、渲染、AI等）。而在一个游戏中，通常有大量的（万级、十万级）游戏对象需要更新状态，除了渲染可以依赖GPU之外，其他的逻辑都需要由CPU完成，甚至绝大部分只能由单线程完成，导致绝大部分时间复杂场景下CPU（主要是内存到CPU的带宽）会成为瓶颈。在CPU单核速度几乎不再增加的时代，如何能让CPU处理的效率提升，是提升游戏性能的核心。
- 代码组织：如同第一章讲的案例一样，当我们用传统OOP的模式进行游戏开发时，很容易就会陷入代码组织上的问题，最终导致代码难以阅读，维护和优化。
- 可扩展性：这个跟上一条类似，但更多的是游戏的特性导致：需要快速更新，加入新的元素。一个游戏的架构需要能通过低代码、甚至0代码的方式增加游戏元素，从而通过快速更新而留住用户。如果每次变更都需要开发新的代码，测试，然后让用户重新下载客户端，可想而知这种游戏很难在现在的竞争环境下活下来。

而ECS架构能很好的解决上面的几个问题，ECS架构主要分为：

- Entity：用来代表任何一个游戏对象，但是在ECS里一个Entity最重要的仅仅是他的EntityID，一个Entity里包含多个Component
- Component：是真正的数据，ECS架构把一个个的实体对象拆分为更加细化的组件，比如位置、素材、状态等，也就是说一个Entity实际上只是一个Bag of Components。
- System（或者ComponentSystem，组件系统）：是真正的行为，一个游戏里可以有很多个不同的组件系统，每个组件系统都只负责一件事，可以依次处理大量的相同组件，而不需要去理解具体的Entity。所以一个ComponentSystem理论上可以有更加高效的组件处理效率，甚至可以实现并行处理，从而提升CPU利用率。

ECS的一些核心性能优化包括将同类型组件放在同一个Array中，然后Entity仅保留到各自组件的pointer，这样能更好的利用CPU的缓存，减少数据的加载成本，以及SIMD的优化等。

一个ECS案例的伪代码如下：

```
public class Entity {
  public Vector position; // 此处Vector是一个Component, 指向的是MovementSys
}

public class MovementSystem {
```

```
  List<Vector> list;

  // System的行为
  public void update(float delta) {
    for(Vector pos : list) { // 这个loop直接走了CPU缓存，性能很高，同时可以用S1
      pos.x = pos.x + delta;
      pos.y = pos.y + delta;
    }
  }
}

@Test
public void test() {
  MovementSystem system = new MovementSystem();
  system.list = new List<>() { new Vector(0, 0) };
  Entity entity = new Entity(list.get(0));
  system.update(0.1);
  assertTrue(entity.position.x == 0.1);
}
```

由于本文不是讲解ECS架构的，感兴趣的同学可以搜索Entity-Component-System或者看看Unity的ECS文档等。

ECS架构分析

重新回来分析ECS，其实它的本源还是几个很老的概念：

组件化

在软件系统里，我们通常将复杂的大系统拆分为独立的组件，来降低复杂度。比如网页里通过前端组件化降低重复开发成本，微服务架构通过服务和数据库的拆分降低服务复杂度和系统影响面等。但是ECS架构把这个走到了极致，即每个对象内部都实现了组件化。通过将一个游戏对象的数据和行为拆分为多个组件和组件系统，能实现组件的高度复用性，降低重复开发成本。

行为抽离

这个在游戏系统里有个比较明显的优势。如果按照OOP的方式，一个游戏对象里可能会包括移动代码、战斗代码、渲染代码、AI代码等，如果都放在一个类里会很长，且很难去维护。通过将通用逻辑抽离出来为单独的System类，可以明显提升代码的可读性。另一个好处则是抽离了一些和对象代码无关的依赖，比如上文的delta，这个delta如果是放在Entity的update方法，则需要作为入参注入，而放在System里则可以统一管理。在第一章的有个问题，到底是应该 Player.attack(monster) 还是 Monster.receiveDamage(Weapon, Player)。在ECS里这个问题就变的很简单，放在CombatSystem里就可以了。

数据驱动

即一个对象的行为不是写死的而是通过其参数决定，通过参数的动态修改，就可以快速改变一个对象的具体行为。在ECS的游戏架构里，通过给Entity注册相应的Component，以及改变Component的具体参数的组合，就可以改变一个对象的行为和玩法，比如创建一个水壶+爆炸属性就变成了“爆炸水壶”、给一个自行车加上风魔法就变成了飞车等。在有些Rougelike游戏中，可能有超过1万件不同类型、不同功能的物品，如果这些不同功能的物品都去单独写代码，可能永远都写不完，但是通过数据驱动+组件化架构，所有物品的配置最终就是一张表，修改也极其简单。这个也是组合胜于继承原则的一次体现。

ECS的缺陷

虽然ECS在游戏界已经开始崭露头角，我发现ECS架构目前还没有在哪个大型商业应用中被使用过。原因可能很多，包括ECS比较新大家还不了解、缺少商业成熟可用的框架、程序员们还不够能适应从写逻辑脚本到写组件的思维转变等，但我认为其最大的一个问题是ECS为了提升性能，强调了数据/状态（State）和行为（Behaivor）分离，并且为了降低GC成本，直接操作数据，走到了一个极端。而在商业应用中，数据的正确性、一致性和健壮性应该是最高的优先级，而性能只是锦上添花的东西，所以ECS很难在商业场景里带来特别大的好处。但这不代表我们不能借鉴一些ECS的突破性思维，包括组件化、跨对象行为的抽离、以及数据驱动模式，而这些在DDD里也能很好的用起来。

基于DDD架构的一种解法

领域对象

回到我们原来的问题域上面，我们从领域层拆分一下各种对象：

实体类

在DDD里，实体类包含ID和内部状态，在这个案例里实体类包含Player、Monster和Weapon。Weapon被设计成实体类是因为两把同名的Weapon应该可以同时存在，所以必须要有ID来区分，同时未来也可以预期Weapon会包含一些状态，比如升级、临时的buff、耐久等。

```
public class Player implements Movable {
    private PlayerId id;
    private String name;
    private PlayerClass playerClass; // enum
    private WeaponId weaponId; //  (Note 1)
    private Transform position = Transform.ORIGIN;
    private Vector velocity = Vector.ZERO;
}

public class Monster implements Movable {
    private MonsterId id;
    private MonsterClass monsterClass; // enum
    private Health health;
    private Transform position = Transform.ORIGIN;
    private Vector velocity = Vector.ZERO;
}

public class Weapon {
    private WeaponId id;
    private String name;
    private WeaponType weaponType; // enum
    private int damage;
    private int damageType; // 0 - physical, 1 - fire, 2 - ice
}
```

在这个简单的案例里，我们可以利用enum的PlayerClass、MonsterClass来代替继承关系，后续也可以利用Type Object设计模式来做到数据驱动。

Note 1: 因为 Weapon 是实体类，但是Weapon能独立存在，Player不是聚合根，所以Player只能保存WeaponId，而不能直接指向Weapon。

值对象的组件化

在前面的ECS架构里，有个MovementSystem的概念是可以复用的，虽然不应该直接去操作Component或者继承通用的父类，但是可以通过接口的方式对领域对象做组件化处理：

```
public interface Movable {
    // 相当于组件
    Transform getPosition();
    Vector getVelocity();

    // 行为
    void moveTo(long x, long y);
    void startMove(long velX, long velY);
    void stopMove();
    boolean isMoving();
}

// 具体实现
public class Player implements Movable {
    public void moveTo(long x, long y) {
        this.position = new Transform(x, y);
    }

    public void startMove(long velocityX, long velocityY) {
        this.velocity = new Vector(velocityX, velocityY);
    }

    public void stopMove() {
        this.velocity = Vector.ZERO;
    }

    @Override
    public boolean isMoving() {
        return this.velocity.getX() != 0 || this.velocity.getY() != 0;
    }
}

@Value
public class Transform {
    public static final Transform ORIGIN = new Transform(0, 0);
    long x;
```

```
    long y;
}

@Value
public class Vector {
    public static final Vector ZERO = new Vector(0, 0);
    long x;
    long y;
}
```

注意两点：

- Moveable的接口没有Setter。一个Entity的规则是不能直接变更其属性，必须通过Entity的方法去对内部状态做变更。这样能保证数据的一致性。
- 抽象Movable的好处是如同ECS一样，一些特别通用的行为（如在大地图里移动）可以通过统一的System代码去处理，避免了重复劳动。

装备行为

因为我们已经不会用Player的子类来决定什么样的Weapon可以装备，所以这段逻辑应该被拆分到一个单独的类里。这种类在DDD里被叫做领域服务（Domain Service）。

```
public interface EquipmentService {
    boolean canEquip(Player player, Weapon weapon);
}
```

在DDD里，一个Entity不应该直接参考另一个Entity或服务，也就是说以下的代码是错误的：

```
public class Player {
    @Autowired
    EquipmentService equipmentService; // BAD: 不可以直接依赖

    public void equip(Weapon weapon) {
        // ...
    }
}
```

这里的问题是Entity只能保留自己的状态（或非聚合根的对象）。任何其他的对象，无论是否通过依赖注入的方式弄进来，都会破坏Entity的Invariance，并且还难以单测。

正确的引用方式是通过方法参数引入（Double Dispatch）：

```
public class Player {

    public void equip(Weapon weapon, EquipmentService equipmentService)
        if (equipmentService.canEquip(this, weapon)) {
            this.weaponId = weapon.getId();
        } else {
            throw new IllegalArgumentException("Cannot Equip: " + weapor
        }
    }
}
```

在这里，无论是Weapon还是EquipmentService都是通过方法参数传入，确保不会污染Player的自有状态。

Double Dispatch是一个使用Domain Service经常会用到的方法，类似于调用反转。

然后在EquipmentService里实现相关的逻辑判断，这里我们用了另一个常用的Strategy（或者叫Policy）设计模式：

```
public class EquipmentServiceImpl implements EquipmentService {
    private EquipmentManager equipmentManager;

    @Override
    public boolean canEquip(Player player, Weapon weapon) {
        return equipmentManager.canEquip(player, weapon);
    }
}

// 策略优先级管理
public class EquipmentManager {
```

```
    private static final List<EquipmentPolicy> POLICIES = new ArrayList<
    static {
        POLICIES.add(new FighterEquipmentPolicy());
        POLICIES.add(new MageEquipmentPolicy());
        POLICIES.add(new DragoonEquipmentPolicy());
        POLICIES.add(new DefaultEquipmentPolicy());
    }

    public boolean canEquip(Player player, Weapon weapon) {
        for (EquipmentPolicy policy : POLICIES) {
            if (!policy.canApply(player, weapon)) {
                continue;
            }
            return policy.canEquip(player, weapon);
        }
        return false;
    }
}

// 策略案例
public class FighterEquipmentPolicy implements EquipmentPolicy {

    @Override
    public boolean canApply(Player player, Weapon weapon) {
        return player.getPlayerClass() == PlayerClass.Fighter;
    }

    /**
     * Fighter能装备Sword和Dagger
     */
    @Override
    public boolean canEquip(Player player, Weapon weapon) {
        return weapon.getWeaponType() == WeaponType.Sword
                || weapon.getWeaponType() == WeaponType.Dagger;
    }
}

// 其他策略省略，见源码
```

这样设计的最大好处是未来的规则增加只需要添加新的Policy类，而不需要去改变原有的类。

攻击行为

在上文中曾经有提起过，到底应该是Player.attack(Monster)还是Monster.receiveDamage(Weapon, Player)？在DDD里，因为这个行为可能会影响到Player、Monster和Weapon，所以属于跨实体的业务逻辑。在这种情况下需要通过一个第三方的领域服务（Domain Service）来完成。

```
public interface CombatService {
    void performAttack(Player player, Monster monster);
}

public class CombatServiceImpl implements CombatService {
    private WeaponRepository weaponRepository;
    private DamageManager damageManager;

    @Override
    public void performAttack(Player player, Monster monster) {
        Weapon weapon = weaponRepository.find(player.getWeaponId());
        int damage = damageManager.calculateDamage(player, weapon, monst
        if (damage > 0) {
            monster.takeDamage(damage); //  (Note 1) 在领域服务里变更Monste
        }
        // 省略掉Player和Weapon可能受到的影响
    }
}
```

同样的在这个案例里，可以通过Strategy设计模式来解决damage的计算问题：

```
// 策略优先级管理
public class DamageManager {
    private static final List<DamagePolicy> POLICIES = new ArrayList<>()
    static {
        POLICIES.add(new DragoonPolicy());
        POLICIES.add(new DragonImmunityPolicy());
```

```
        POLICIES.add(new OrcResistancePolicy());
        POLICIES.add(new ElfResistancePolicy());
        POLICIES.add(new PhysicalDamagePolicy());
        POLICIES.add(new DefaultDamagePolicy());
    }

    public int calculateDamage(Player player, Weapon weapon, Monster mor
        for (DamagePolicy policy : POLICIES) {
            if (!policy.canApply(player, weapon, monster)) {
                continue;
            }
            return policy.calculateDamage(player, weapon, monster);
        }
        return 0;
    }
}

// 策略案例
public class DragoonPolicy implements DamagePolicy {
    public int calculateDamage(Player player, Weapon weapon, Monster mor
        return weapon.getDamage() * 2;
    }
    @Override
    public boolean canApply(Player player, Weapon weapon, Monster monste
        return player.getPlayerClass() == PlayerClass.Dragoon &&
                monster.getMonsterClass() == MonsterClass.Dragon;
    }
}
```

特别需要注意的是这里的CombatService领域服务和3.2的EquipmentService领域服务，虽然都是领域服务，但实质上有很大的差异。上文的EquipmentService更多的是提供只读策略，且只会影响单个对象，所以可以在Player.equip方法上通过参数注入。但是CombatService有可能会影响多个对象，所以不能直接通过参数注入的方式调用。

单元测试

```
@Test
```

```
@DisplayName("Dragoon attack dragon doubles damage")
public void testDragoonSpecial() {
    // Given
    Player dragoon = playerFactory.createPlayer(PlayerClass.Dragoon, "Da
    Weapon sword = weaponFactory.createWeaponFromPrototype(swordProto, '
    ((WeaponRepositoryMock)weaponRepository).cache(sword);
    dragoon.equip(sword, equipmentService);
    Monster dragon = monsterFactory.createMonster(MonsterClass.Dragon, 1

    // When
    combatService.performAttack(dragoon, dragon);

    // Then
    assertThat(dragon.getHealth()).isEqualTo(Health.ZERO);
    assertThat(dragon.isAlive()).isFalse();
}

@Test
@DisplayName("Orc should receive half damage from physical weapons")
public void testFighterOrc() {
    // Given
    Player fighter = playerFactory.createPlayer(PlayerClass.Fighter, "My
    Weapon sword = weaponFactory.createWeaponFromPrototype(swordProto, '
    ((WeaponRepositoryMock)weaponRepository).cache(sword);
    fighter.equip(sword, equipmentService);
    Monster orc = monsterFactory.createMonster(MonsterClass.Orc, 100);

    // When
    combatService.performAttack(fighter, orc);

    // Then
    assertThat(orc.getHealth()).isEqualTo(Health.of(100 - 10 / 2));
}
```

具体的代码比较简单，解释省略

移动系统

最后还有一种Domain Service，通过组件化，我们其实可以实现ECS一样的System，来降低一些重复性的代码：

```
public class MovementSystem {

    private static final long X_FENCE_MIN = -100;
    private static final long X_FENCE_MAX = 100;
    private static final long Y_FENCE_MIN = -100;
    private static final long Y_FENCE_MAX = 100;

    private List<Movable> entities = new ArrayList<>();

    public void register(Movable movable) {
        entities.add(movable);
    }

    public void update() {
        for (Movable entity : entities) {
            if (!entity.isMoving()) {
                continue;
            }

            Transform old = entity.getPosition();
            Vector vel = entity.getVelocity();
            long newX = Math.max(Math.min(old.getX() + vel.getX(), X_FEN
            long newY = Math.max(Math.min(old.getY() + vel.getY(), Y_FEN
            entity.moveTo(newX, newY);
        }
    }
}
```

单测：

```
@Test
@DisplayName("Moving player and monster at the same time")
public void testMovement() {
    // Given
```

```
    Player fighter = playerFactory.createPlayer(PlayerClass.Fighter, "My
    fighter.moveTo(2, 5);
    fighter.startMove(1, 0);

    Monster orc = monsterFactory.createMonster(MonsterClass.Orc, 100);
    orc.moveTo(10, 5);
    orc.startMove(-1, 0);

    movementSystem.register(fighter);
    movementSystem.register(orc);

    // When
    movementSystem.update();

    // Then
    assertThat(fighter.getPosition().getX()).isEqualTo(2 + 1);
    assertThat(orc.getPosition().getX()).isEqualTo(10 - 1);
}
```

在这里MovementSystem就是一个相对独立的Domain Service，通过对Movable的组件化，实现了类似代码的集中化、以及一些通用依赖/配置的中心化（如X、Y边界等）。

DDD领域层的一些设计规范

上面我主要针对同一个例子对比了OOP、ECS和DDD的3种实现，比较如下：

- 基于继承关系的OOP代码：OOP的代码最好写，也最容易理解，所有的规则代码都写在对象里，但是当领域规则变得越来越复杂时，其结构会限制它的发展。新的规则有可能会导致代码的整体重构。
- 基于组件化的ECS代码：ECS代码有最高的灵活性、可复用性、及性能，但极具弱化了实体类的内聚，所有的业务逻辑都写在了服务里，会导致业务的一致性无法保障，对商业系统会有较大的影响。
- 基于领域对象 + 领域服务的DDD架构：DDD的规则其实最复杂，同时要考虑到实体类的内聚和保证不变性（Invariants），也要考虑跨对象规则代码的归属，甚至要考虑到具体领域服务的调用方式，理解成本比较高。

所以下面，我会尽量通过一些设计规范，来降低DDD领域层的设计成本。领域层里的Value Object（Domain Primitive）设计规范请参考我之前的文章。

实体类（Entity）

大多数DDD架构的核心都是实体类，实体类包含了一个领域里的状态、以及对状态的直接操作。Entity最重要的设计原则是保证实体的不变性（Invariants），也就是说要确保无论外部怎么操作，一个实体内部的属性都不能出现相互冲突，状态不一致的情况。所以几个设计原则如下：

创建即一致

在贫血模型里，通常见到的代码是一个模型通过手动new出来之后，由调用方一个参数一个参数的赋值，这就很容易产生遗漏，导致实体状态不一致。所以DDD里实体创建的方法有两种：

constructor参数要包含所有必要属性，或者在constructor里有合理的默认值。

比如，账号的创建：

```
public class Account {
    private String accountNumber;
    private Long amount;
}

@Test
public void test() {
    Account account = new Account();
    account.setAmount(100L);
    TransferService.transfer(account); // 报错了，因为Account缺少必要的Accou
}
```

如果缺少一个强校验的constructor，就无法保障创建的实体的一致性。所以需要增加一个强校验的constructor：

```
public class Account {
    public Account(String accountNumber, Long amount) {
```

```
        assert StringUtils.isNotBlank(accountNumber);
        assert amount >= 0;
        this.accountNumber = accountNumber;
        this.amount = amount;
    }
}

@Test
public void test() {
    Account account = new Account("123", 100L); // 确保对象的有效性
}
```

使用Factory模式来降低调用方复杂度

另一种方法是通过Factory模式来创建对象，降低一些重复性的入参。比如：

```
public class WeaponFactory {
    public Weapon createWeaponFromPrototype(WeaponPrototype proto, String
        Weapon weapon = new Weapon(null, newName, proto.getWeaponType(),
        return weapon;
    }
}
```

通过传入一个已经存在的Prototype，可以快速的创建新的实体。还有一些其他的如Builder等设计模式就不一一指出了。

尽量避免public setter

一个最容易导致不一致性的原因是实体暴露了public的setter方法，特别是set单一参数会导致状态不一致的情况。比如，一个订单可能包含订单状态（下单、已支付、已发货、已收货）、支付单、物流单等子实体，如果一个调用方能随意去set订单状态，就有可能导致订单状态和子实体匹配不上，导致业务流程走不通的情况。所以在实体里，需要通过行为方法来修改内部状态：

```
@Data @Setter(AccessLevel.PRIVATE) // 确保不生成public setter
```

```
public class Order {
    private int status; // 0 - 创建，1 - 支付，2 - 发货，3 - 收货
    private Payment payment; // 支付单
    private Shipping shipping; // 物流单

    public void pay(Long userId, Long amount) {
        if (status != 0) {
            throw new IllegalStateException();
        }
        this.status = 1;
        this.payment = new Payment(userId, amount);
    }

    public void ship(String trackingNumber) {
        if (status != 1) {
            throw new IllegalStateException();
        }
        this.status = 2;
        this.shipping = new Shipping(trackingNumber);
    }
}
```

【建议】在有些简单场景里，有时候确实可以比较随意的设置一个值而不会导致不一致性，也建议将方法名重新写为比较“行为化”的命名，会增强其语意。比如setPosition(x, y)可以叫做moveTo(x, y)，setAddress可以叫做assignAddress等。

通过聚合根保证主子实体的一致性

在稍微复杂一点的领域里，通常主实体会包含子实体，这时候主实体就需要起到聚合根的作用，即：

- 子实体不能单独存在，只能通过聚合根的方法获取到。任何外部的对象都不能直接保留子实体的引用
- 子实体没有独立的Repository，不可以单独保存和取出，必须要通过聚合根的Repository实例化
- 子实体可以单独修改自身状态，但是多个子实体之间的状态一致性需要聚合根来保障

常见的电商域中聚合的案例如主子订单模型、商品/SKU模型、跨子订单优惠、跨店优惠模型等。很多聚合根和Repository的设计规范在我前面一篇关于Repository的文章中已经详细解释

过，可以拿来参考。

不可以强依赖其他聚合根实体或领域服务

一个实体的原则是高内聚、低耦合，即一个实体类不能直接在内部直接依赖一个外部的实体或服务。这个原则和绝大多数ORM框架都有比较严重的冲突，所以是一个在开发过程中需要特别注意的。这个原则的必要原因包括：对外部对象的依赖性会直接导致实体无法被单测；以及一个实体无法保证外部实体变更后不会影响本实体的一致性和正确性。

所以，正确的对外部依赖的方法有两种：

1. 只保存外部实体的ID：这里我再次强烈建议使用强类型的ID对象，而不是Long型ID。强类型的ID对象不单单能自我包含验证代码，保证ID值的正确性，同时还能确保各种入参不会因为参数顺序变化而出bug。具体可以参考我的Domain Primitive文章。
2. 针对于“无副作用”的外部依赖，通过方法入参的方式传入。比如上文中的equip(Weapon，EquipmentService）方法。

如果方法对外部依赖有副作用，不能通过方法入参的方式，只能通过Domain Service解决，见下文。

任何实体的行为只能直接影响到本实体（和其子实体）

这个原则更多是一个确保代码可读性、可理解的原则，即任何实体的行为不能有“直接”的”副作用“，即直接修改其他的实体类。这么做的好处是代码读下来不会产生意外。

另一个遵守的原因是可以降低未知的变更的风险。在一个系统里一个实体对象的所有变更操作应该都是预期内的，如果一个实体能随意被外部直接修改的话，会增加代码bug的风险。

领域服务（Domain Service）

在上文讲到，领域服务其实也分很多种，在这里根据上文总结出来三种常见的：

单对象策略型

这种领域对象主要面向的是单个实体对象的变更，但涉及到多个领域对象或外部依赖的一些规则。在上文中，EquipmentService即为此类：

- 变更的对象是Player的参数
- 读取的是Player和Weapon的数据，可能还包括从外部读取一些数据

在这种类型下，实体应该通过方法入参的方式传入这种领域服务，然后通过Double Dispatch来反转调用领域服务的方法，比如：

```
Player.equip(Weapon, EquipmentService) {
    EquipmentService.canEquip(this, Weapon);
}
```

为什么这种情况下不能先调用领域服务，再调用实体对象的方法，从而减少实体对领域服务的入参型依赖呢？比如，下面这个方法是错误的：

```
boolean canEquip = EquipmentService.canEquip(Player, Weapon);
if (canEquip) {
    Player.equip(Weapon); // ✘，这种方法不可行，因为这个方法有不一致的可能性
}
```

其错误的主要原因是缺少了领域服务入参会导致方法有可能产生不一致的情况。

跨对象事务型

当一个行为会直接修改多个实体时，不能再通过单一实体的方法作处理，而必须直接使用领域服务的方法来做操作。在这里，领域服务更多的起到了跨对象事务的作用，确保多个实体的变更之间是有一致性的。

在上文里，虽然以下的代码虽然可以跑到通，但是是不建议的：

```
public class Player {
    void attack(Monster, CombatService) {
        CombatService.performAttack(this, Monster); // ✘，不要这么写，会导致
    }
}
```

而我们真实调用应该直接调用CombatService的方法：

```
public void test() {
    //...
    combatService.performAttack(mage, orc);
}
```

这个原则也映射了4.1.5 的原则，即Player.attack会直接影响到Monster，但这个调用Monster又没有感知。

通用组件型

这种类型的领域服务更像ECS里的System，提供了组件化的行为，但本身又不直接绑死在一种实体类上。具体案例可以参考上文中的MovementSystem实现。

策略对象（Domain Policy）

Policy或者Strategy设计模式是一个通用的设计模式，但是在DDD架构中会经常出现，其核心就是封装领域规则。

一个Policy是一个无状态的单例对象，通常需要至少2个方法：canApply 和 一个业务方法。其中，canApply方法用来判断一个Policy是否适用于当前的上下文，如果适用则调用方会去触发业务方法。通常，为了降低一个Policy的可测试性和复杂度，Policy不应该直接操作对象，而是通过返回计算后的值，在Domain Service里对对象进行操作。

在上文案例里，DamagePolicy只负责计算应该受到的伤害，而不是直接对Monster造成伤害。这样除了可测试外，还为未来的多Policy叠加计算做了准备。

除了本文里静态注入多个Policy以及手动排优先级之外，在日常开发中经常能见到通过Java的SPI机制或类SPI机制注册Policy，以及通过不同的Priority方案对Policy进行排序，在这里就不作太多的展开了。

加餐 - 副作用的处理方法 - 领域事件

在上文中，有一种类型的领域规则被我刻意忽略了，那就是"副作用"。一般的副作用发生在核心领域模型状态变更后，同步或者异步对另一个对象的影响或行为。在这个案例里，我们可以增加一个副作用规则：

- 当Monster的生命值降为0后，给Player奖励经验值

这种问题有很多种解法，比如直接把副作用写在CombatService里：

```
public class CombatService {
    public void performAttack(Player player, Monster monster) {
        // ...
        monster.takeDamage(damage);
        if (!monster.isAlive()) {
            player.receiveExp(10); // 收到经验
        }
    }
}
```

但是这样写的问题是：很快CombatService的代码就会变得很复杂，比如我们再加一个副作用：

- 当Player的exp达到100时，升一级

这时我们的代码就会变成：

```
public class CombatService {
    public void performAttack(Player player, Monster monster) {
        // ...
        monster.takeDamage(damage);
        if (!monster.isAlive()) {
            player.receiveExp(10); // 收到经验
            if (player.canLevelUp()) {
                player.levelUp(); // 升级
            }
        }
    }
}
```

如果再加上“升级后奖励XXX”呢？“更新XXX排行”呢？依此类推，后续这种代码将无法维护。所以我们需要介绍一下领域层最后一个概念：领域事件（Domain Event）。

领域事件介绍

领域事件是一个在领域里发生了某些事后，希望领域里其他对象能够感知到的通知机制。在上面的案例里，代码之所以会越来越复杂，其根本的原因是反应代码（比如升级）直接和上面的事件触发条件（比如收到经验）直接耦合，而且这种耦合性是隐性的。领域事件的好处就是将这种隐性的副作用“显性化”，通过一个显性的事件，将事件触发和事件处理解耦，最终起到代码更清晰、扩展性更好的目的。

所以，领域事件是在DDD里，比较推荐使用的跨实体“副作用”传播机制。

领域事件实现

和消息队列中间件不同的是，领域事件通常是立即执行的、在同一个进程内、可能是同步或异步。我们可以通过一个EventBus来实现进程内的通知机制，简单实现如下：

```
// 实现者：瑜进 2019/11/28
public class EventBus {

    // 注册器
    @Getter
    private final EventRegistry invokerRegistry = new EventRegistry(this

    // 事件分发器
    private final EventDispatcher dispatcher = new EventDispatcher(Execu

    // 异步事件分发器
    private final EventDispatcher asyncDispatcher = new EventDispatcher

    // 事件分发
    public boolean dispatch(Event event) {
        return dispatch(event, dispatcher);
    }

    // 异步事件分发
    public boolean dispatchAsync(Event event) {
        return dispatch(event, asyncDispatcher);
    }

```

```
    // 内部事件分发
    private boolean dispatch(Event event, EventDispatcher dispatcher) {
        checkEvent(event);
        // 1.获取事件数组
        Set<Invoker> invokers = invokerRegistry.getInvokers(event);
        // 2.一个事件可以被监听N次，不关心调用结果
        dispatcher.dispatch(event, invokers);
        return true;
    }

    // 事件总线注册
    public void register(Object listener) {
        if (listener == null) {
            throw new IllegalArgumentException("listener can not be nul
        }
        invokerRegistry.register(listener);
    }

    private void checkEvent(Event event) {
        if (event == null) {
            throw new IllegalArgumentException("event");
        }
        if (!(event instanceof Event)) {
            throw new IllegalArgumentException("Event type must by " + 
        }
    }
}
```

调用方式：

```
public class LevelUpEvent implements Event {
    private Player player;
}

public class LevelUpHandler {
    public void handle(Player player);
}

```

```
public class Player {
    public void receiveExp(int value) {
        this.exp += value;
        if (this.exp >= 100) {
            LevelUpEvent event = new LevelUpEvent(this);
            EventBus.dispatch(event);
            this.exp = 0;
        }
    }
}
@Test
public void test() {
    EventBus.register(new LevelUpHandler());
    player.setLevel(1);
    player.receiveExp(100);
    assertThat(player.getLevel()).equals(2);
}
```

目前领域事件的缺陷和展望

从上面代码可以看出来，领域事件的很好的实施依赖EventBus、Dispatcher、Invoker这些属于框架级别的支持。同时另一个问题是因为Entity不能直接依赖外部对象，所以EventBus目前只能是一个全局的Singleton，而大家都应该知道全局Singleton对象很难被单测。这就容易导致Entity对象无法被很容易的被完整单测覆盖全。

另一种解法是侵入Entity，对每个Entity增加一个List:

```
public class Player {
  List<Event> events;

  public void receiveExp(int value) {
        this.exp += value;
        if (this.exp >= 100) {
            LevelUpEvent event = new LevelUpEvent(this);
            events.add(event); // 把event加进去
            this.exp = 0;
        }
```

```
    }
}

@Test
public void test() {
    EventBus.register(new LevelUpHandler());
    player.setLevel(1);
    player.receiveExp(100);

    for(Event event: player.getEvents()) { // 在这里显性的dispatch事件
        EventBus.dispatch(event);
    }

    assertThat(player.getLevel()).equals(2);
}
```

但是能看出来这种解法不但会侵入实体本身，同时也需要比较啰嗦的显性在调用方dispatch事件，也不是一个好的解决方案。

也许未来会有一个框架能让我们既不依赖全局Singleton，也不需要显性去处理事件，但目前的方案基本都有或多或少的缺陷，大家在使用中可以注意。

总结

在真实的业务逻辑里，我们的领域模型或多或少的都有一定的“特殊性”，如果100%的要符合DDD规范可能会比较累，所以最主要的是梳理一个对象行为的影响面，然后作出设计决策，即：

- 是仅影响单一对象还是多个对象，
- 规则未来的拓展性、灵活性，
- 性能要求，
- 副作用的处理，等等

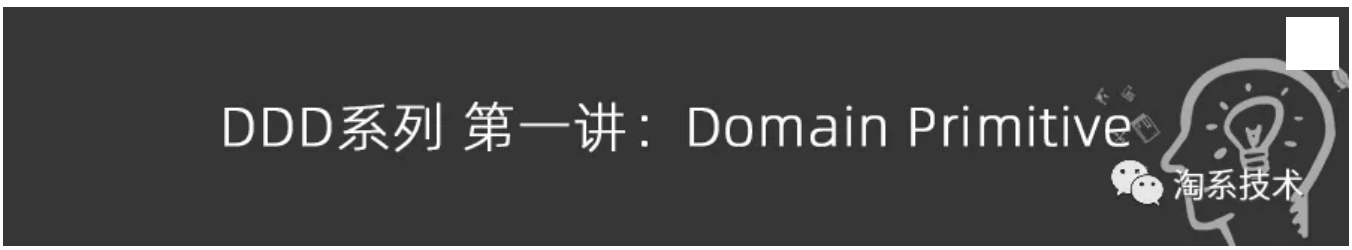
当然，很多时候一个好的设计是多种因素的取舍，需要大家有一定的积累，真正理解每个架构背后的逻辑和优缺点。一个好的架构师不是有一个正确答案，而是能从多个方案中选出一个最平衡的方案。

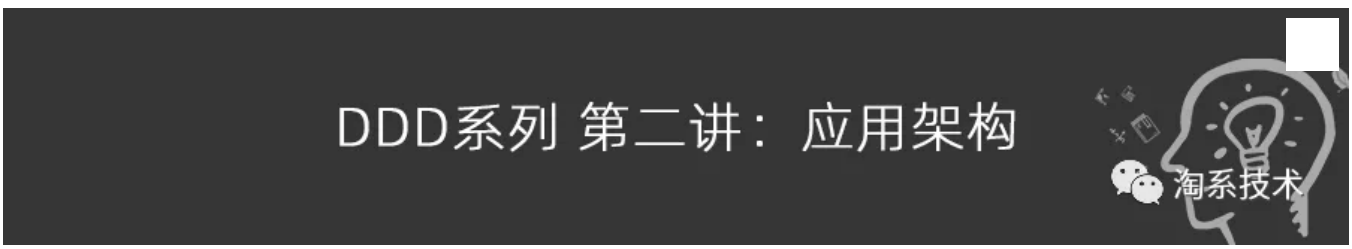
求案例，求简历

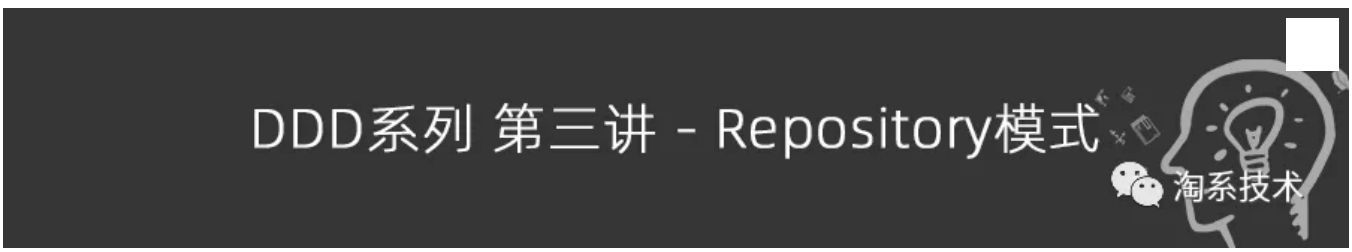
最后，求看到这里的读者能给我一些输入，我后续想写一下如何让一个程序员不要总写CRUD代码的主题，希望能从一些明显CRUD的代码案例中，能通过DDD架构的设计产出可扩展、可维护的架构来。目前我还缺少一些真实案例，希望读者同学们能将一些案例通过邮件的方式给我，包括（脱了敏的）代码和业务描述等。我保证会尽可能的回复每一个案例，并且会把一些经典案例放在文章中。我的邮箱：guangmiao.lgm@alibaba-inc.com，也可以加我的钉钉号：luangm（殷浩）

同时，我们团队也在持续招聘。我负责淘系的行业和导购团队，我们团队负责天猫和淘宝的四大行业（服饰、快消、消电、家装）的日常业务需求和创新业务（3D/AR、搭配、定制、尺码导购等）、前台场（iFashion、全球购、亲宝贝、美妆学院、酷玩星球等），以及手淘的横向导购场（有好货、好店、大赏、品牌导购等），总DAU（日均访问用户数）大概3000W左右。今年我们团队的核心目标是去重构行业的导购体验，给消费者带来不一样的、更人性化、有更强交互性、更能体现每个细分行业差异化特性、文化的导购场。欢迎感兴趣的同学加盟。

✿ 拓展阅读







作者|殷浩

编辑|橙子君

出品|阿里巴巴新零售淘系技术