## 实体

“实体（Entity）”这个词被我们广泛使用，甚至被我们过分使用。设计数据库时，我们用到实体，例如《数据模型资源手册》就说：“实体是一个重要的概念，企业希望建立和存储的信息都是关于实体的信息。”在分解系统的组成部分时，我们用到实体，例如《系统架构》就说：“实体也称为部件、模块、例程、配件等，就是用来构成全体的各个小块”。在徐昊提出的四色建模法中，实体又变为“代表参与到流程中的人/组织/地点/物品”。

前面搬来亚里士多德的理论，说明实体是我们要描述的 **主体** ，而另一个古希腊哲学家巴门尼德则认为实体是 **不同变化状态的本体** 。这两个颇为抽象的论断差不多可以表达领域驱动设计中“实体”这个概念，那就是能够以 **主体类型** 的形式表达领域逻辑中具有个性特征的概念，而这个 **主体的状态在相当长一段时间内会持续地变化** ，因此需要有一个 **身份标识（Identity）** 来标记它。

如果我们认同范畴理论中“其他范畴必须内居于一主体”的论断，则说明 **实体必须包括属性与行为** ，而这些属性往往又由别的次要主体（同样为实体）或表示数量、性质的值对象组成。这一设计原则体现了“分而治之”的思想，也满足单一职责原则。在一些复杂的商业项目中，现实世界对应的主题概念往往具有几十乃至上百个属性。若遵循“将数据与行为封装在一起”的面向对象设计原则，固然可以避免贫血模型，但也会导致一个实体类承担了太多的职责，从而变得过于臃肿。注意，将实体的高内聚属性进一步封装为实体或值对象，并不同于前面提及的“分离具体使用的行为”，而是让对象的粒度变得更细，同时履行自己应有的职责，符合自治对象的“最小完备”特性。

一个典型的实体应该具备三个要素：

* 身份标识
* 属性
* 领域行为

### 身份标识

身份标识（Identity，或简称为 ID）是实体对象的必要标志，换言之， **没有身份标识的领域对象就不是实体** 。实体的身份标识就好像每个公民的身份证号，用以判断相同类型的不同对象是否代表同一个实体。身份标识除了帮助我们识别实体的同一性之外，主要的目的还是为了管理实体的生命周期。实体的状态是可以变更的，这意味着我们不能根据实体的属性值进行判断，如果没有唯一的身份标识，就无法跟踪实体的状态变更，也就无法正确地保证实体从创建、更改到消亡的生命过程。

一些实体只要求身份标识具有唯一性即可，如评论实体、博客实体或文章实体的身份标识，都可以使用自动增长的 Long 类型或者随机数与 UUID、GUID，这样的身份标识并没有任何业务含义。有些实体的身份标识则规定了一定的组合规则，例如公民实体、员工实体与订单实体的身份标识就不是随意生成的。遵循业务规则生成的身份标识体现了领域概念，例如公民实体的身份标识其实就是“身份证号”这一领域概念。定义规则的好处在于我们可以通过解析身份标识获取有用的领域信息，例如解析身份证号，可以直接获得该公民的部分基础信息，如籍贯、出生日期、性别等，解析订单号即可获知该订单的下单渠道、支付渠道、业务类型与下单日期等。

在设计实体的身份标识时，通常可以将身份标识的类型分为两个层次：通用类型与领域类型。

通用类型提供了系统所需的各种生成唯一标识的类型，如基于规则的标识、基于随机数的标识、支持分布式环境唯一性的标识等。这些类型都将放在 **系统层** 代码模型的 domain 包中，可以作为整个系统的共享内核。例如，我们定义一个通用的 Identity 接口：

public interface Identity<T> extends Serializable {  
 T value();  
 boolean isEmpty();  
 T emptyId();  
}

随机数的身份标识则如下接口所示：

public interface RandomIdentity<T> {  
  
 T next();  
  
}

如果需要按照一定规则生成身份标识，而其唯一性的保证则由随机数来承担，则可以定义 RuleRandomIdentity 类。它同时实现了 Identity 与 RandomIdentity 接口：

@Immutable  
  
public class RuleRandomIdentity implements RandomIdentity<String>, Identity<String> {  
  
 private String value;  
  
  
 private String prefix;  
  
 private int seed;  
  
 private String joiner;  
  
 private static final int DEFAULT\_SEED = 100\_000;  
  
 private static final String DEFAULT\_JOINER = "\_";  
  
 private static final long serialVersionUID = 1L;  
  
 public RuleRandomIdentity() {  
  
 this("", DEFAULT\_SEED, DEFAULT\_JOINER);  
  
 }  
  
 public RuleRandomIdentity(int seed) {  
 this("", seed, DEFAULT\_JOINER);  
 }  
  
  
 public RuleRandomIdentity(String prefix, int seed) {  
  
 this(prefix, seed, DEFAULT\_JOINER);  
  
 }  
  
  
 public RuleRandomIdentity(String prefix, int seed, String joiner) {  
  
 this.prefix = prefix;  
  
 this.seed = seed;  
  
 this.joiner = joiner;  
  
  
 this.value = compose(prefix, seed, joiner);  
  
 }  
  
  
 @Override  
  
 public final String value() {  
  
 return this.value;  
  
 }  
  
  
 @Override  
  
 public final boolean isEmpty() {  
  
 return value.isEmpty();  
  
 }  
  
  
 @Override  
  
 public final String emptyId() {  
  
 return "";  
  
 }  
  
  
 @Override  
  
 public final String next() {  
  
 return compose(prefix, seed, joiner);  
  
 }  
  
  
 private String compose(String prefix, int seed, String joiner) {  
  
 long suffix = new Random(seed).nextLong();  
  
 return String.format("%s%s%s", prefix, joiner, suffix);  
  
 }  
  
}

UUID 可以视为一种特殊的随机数，同时实现了 Identity 与 RandomIdentity 接口：

@Immutable  
public class UUIDIdentity implements RandomIdentity<String>, Identity<String> {  
 private String value;  
  
 public UUIDIdentity() {  
 this.value = next();  
 }  
  
 private static final long serialVersionUID = 1L;  
  
 @Override  
 public String next() {  
 return UUID.randomUUID().toString();  
 }  
  
 @Override  
 public String value() {  
 return value;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return value.isEmpty();  
 }  
  
 @Override  
 public String emptyId() {  
 return "";  
 }  
}

这些基础的身份标识类应具备序列化的能力，同时还应保证它们的不变性。注意，包括 UUID 在内的随机数并不能支持分布式环境的唯一性，它需要特殊的算法，例如采用 SnowFlake 算法来避免在分布式系统内产生身份标识的碰撞。

定义通用类型的身份标识是为了重用，只有领域类型的身份标识才与各个限界上下文的实体对象有关，例如为 Employee 定义 EmployeeId 类型，为 Order 定义 OrderId 类型。在定义领域类型的身份标识时，可以选择恰当的通用类型身份标识作为父类，然后在自身类的定义中封装生成身份标识的领域逻辑。例如，EmployeeId 会根据企业的要求生成具有统一前缀的标识，就可以让 EmployeeId 继承自 RuleRandomIdentity，并让企业名称作为身份标识的前缀：

public final class EmployeeId extends RuleRandomIdentity {  
 private String employeeId;  
 private static final String COMPANY\_NAME = "topddd";   
  
 public EmployeeId(int seed) {  
 super(COMPANY\_NAME, seed);  
 }  
}

领域类型的身份标识往往具备领域知识和业务逻辑。它是实体的身份标识，但它自身却应该被定义为值类型，保持值的不变性，同时提供属于身份标识的常用方法，隐藏生成身份标识值的细节，以便于应对未来可能的变化。

### 属性

实体的属性用来说明其主体的 **静态特征** ，并通过这些属性持有数据与状态。通常，我们会依据粒度的大小将属性分为 **基本属性** 与 **组合属性** 。简单说来，定义为开发语言内置类型的属性就是所谓的“基本属性”，如整型、布尔型、字符串类型等；与之相反，组合属性则通过自定义类型来表现。例如 Product 实体的属性定义：

public class Product extends Entity<ProductId> {  
 private String name;  
 private int quantity;  
 private Category category;  
 private Weight weight;  
 private Volume volume;  
 private Price price;  
}

其中，Product 实体的 name、quantity 属性属于基本属性，分别被定义为 String 与 Int 类型；而 category、weight、volume、price 等属性则为组合属性，类型为自定义的 Category、Weight、Volume 和 Price 类型。

这两种属性之间是否存在什么分界线？例如说，难道我们就不能将 category 定义为 String 类型，将 weight 定义为 Double 类型吗？又或者，难道我们不能将 name 定义为 Name 类型，将 quantity 定义为 Quantity 类型吗？我认为，划定这条边界线的标准是： **该属性是否存在约束规则、组合因子或属于自己的领域行为？**

先来看 **约束规则** 。相较于产品名而言，产品的类别具有更强的约束性，避免出现分类无休止地增长，过多离散细小的分类反而不利于产品的管理。更何况，从业务规则来看，产品的类别可能还存在一个复杂的层次结构，单单靠一个字符串是没法表达如此丰富的约束条件与层次结构的。当然，如果需求对产品名也有明确的约束，为其定义一个 Name 类也未尝不可；只是相比较而言，定义 Category 的必要性更有说服力罢了。

再看 **组合因子** 。这其实是看属性的不可再分性。我们看 Weight 与 Volumn 两个对象，就有非常明显的特征：它们都需要值与计数单位共同组合。如果只有一个值，会导致计算结果的不匹配，概念也会出现混乱，例如 2kg 与 2g 显然是两个不同的值，不能混为一谈。至于 quantity 属性之所以被设计为基本属性，是假定它没有计数单位的要求，因而无需与其他值组合。当然，这样的设计取决于业务场景。如果需求说明商品数量的单位存在个位数、万位数、亿位数的变化，又或者以箱、盒、件等不同的量化单位区分不同的商品，作为基本属性的 quantity 就缺乏业务的表现能力，必须将其定义为组合属性。

最后来看 **领域行为** 。由于多数语言不支持为基本类型扩展自定义行为（C# 的扩展方法，Scala 的隐式转换支持这种扩展，但这种扩展意味着基础类型的扩展，而非对应领域概念的扩展），让若需要为属性添加完全属于自己的领域行为，就只能选择组合属性。例如 Product 的 Price 属性，需要提供运算行为。这种运算并非普通数值类型的四则运算，而是与价格计算的领域逻辑绑定在一起的。如果不将其定义为 Price 类型，就无法为其封装自定义行为。

由于实体的组合属性是一个自定义类型，而它们又并不需要唯一的身份标识，因此在领域设计建模阶段，这些组合属性其实都可以定义为值对象类型。

设计实体时，应该遵循 **保持实体专注于身份** 这一设计原则，让实体只承担符合它身份的业务行为，而把内聚性更强的属性分解为单独的值对象，并运用“信息专家模式”将操作了值对象属性的业务行为推向值对象，让值对象成为高内聚的体现领域逻辑的对象。这样的设计符合面向对象设计思想的“ **职责分治** ”原则，即依据各自持有的数据与状态以及和领域概念之间的粘度来分配职责，保证了实体类的单一职责。

### 领域行为

实体拥有领域行为，可以更好地说明其主体的 **动态特征** 。一个不具备动态特征的对象，是一个哑对象，一个蠢对象。这样的对象明明坐拥宝山（自己的属性）而不自知，反而去求助他人帮他操作自己的状态，不是愚蠢是什么？为实体定义表达领域行为的方法，与前面讲到组合属性需要封装自己的领域行为是一脉相承的，都是“职责分治”的设计思想体现。

实体的领域行为依据不同的特征，可以分为：

* 变更状态的领域行为
* 自给自足的领域行为
* 互为协作的领域行为

**变更状态的领域行为**

一个实体对象的状态是由属性持有的，与 **值对象** 不同，实体对象是允许调用者更改其状态的。许多语言都支持通过 get 与 set 方法（或类似的语法糖）来访问状态。然而，领域驱动设计强调 **代码模型也是领域模型的一部分** ，因此代码模型中的类名、方法名都需要以业务角度去表达领域逻辑，甚至希望领域专家也能够参与到编程元素的命名讨论上。至少，我们应该让这些命名遵循团队共同制定的 **统一语言** 。因此，单从命名看，我们并不希望遵循 Java Bean 的规范，单纯地将这些变更状态的方法定义为 set 方法。例如，修改产品价格的领域行为就应该定义为 changePriceTo(newPrice) 方法，而非 setPrice(newPrice)：

public class Product extends Entity<ProductId> {  
 public void changePriceTo(Price newPrice) {  
 if (!this.price.sameCurrency(newPrice)) {  
 throw new CurrencyException("Cannot change the price of this product to a different currency");  
 }  
 this.sellingPrice = newPrice;  
 }  
}

准确地说，我们将变更状态的方法认为是实体拥有的领域行为，这就突破了 set 方法的范畴，使得我们定义的实体变得更加智能，符合面向对象的特征。

**自给自足的领域行为**

既然是自给自足，就意味着实体对象只能操作自己的属性，而不外求于别的对象。这种领域行为最容易管理，因为它不会和别的实体对象产生依赖。即使发生了变化，只要定义好的接口无需调整，就不会将变化传递出去。例如，航班实体对象 Flight 定义了计划飞行时间、估算飞行时间与实际飞行时间等属性，领域逻辑需要获得这三者之间的统计值：

public class Flight extends Entity<FlightId> {  
 private FlightTimePeriod scheduleFlight;  
 private FlightTimePeriod estimateFlight;  
 private FlightTimePeriod actualFlight;  
  
 public FlightStatistic analyze() {  
 long scheduleElapsedSeconds = scheduleFlight.elapsedSeconds();  
 long estimateElapsedSeconds = estimateFlight.elapsedSeconds();  
 long actualElapsedSeconds = actualFlight.elapsedSeconds();  
 return new FlightStatistic(schedulElapsedSeconds - actualElapsedSeconds, estimateElapsedSeconds - actualElapsedSeconds);  
 }  
}  
  
public class FlightTimePeriod {  
 private LocalDateTime takeOffTime;  
 private LocalDateTime landingTime;  
  
 public FlightTimePeriod(LocalDateTime takeOffTime, LocalDateTime landingTime) {  
 if (takeOffTime.after(landingTime)) {  
 throw new FlightTimePeroidException("Take off time should not be later than landing time".)  
 }  
 this.takeOffTime = takeOffTime;  
 this.landingTime = landingTime;  
 }  
  
 public long elapsedSeconds() {  
 Duration duration = Duration.between(takeOffTime, landingTime);  
 return duration.toMillis() \* 1000;   
 }  
}

变更状态的领域行为由于要改变实体的状态，意味着该操作往往会产生副作用。自给自足的领域行为则有所不同，它主要是对实体已有的属性值（包括调用该实体组合属性定义的方法返回的值，如前面代码中 FlightTimePeriod 的方法 elapsedSeconds() 返回的值）进行计算，返回调用者希望获得的结果。

对象不可能做到完全的自给自足，有时也需要调用者提供必要的信息。这时，就可以通过方法参数传入外部数据。若方法参数为其他的领域对象，就变为领域对象之间互为协作的领域行为。

**互为协作的领域行为**

除了操作属于自己的属性，实体也可以调用别的对象，形成一种协作关系。要注意区分实体属性与外部对象。如果实体对象操作的是自己的属性对象，就不属于互相协作的范畴。因此，参与协作的对象通常作为方法的外部参数传入。例如，在 Rental 实体中，如果需要根据客户类型计算每月的租金，就需要与 CustomerType 对象进行协作：

public class Rental extends Entity<RentalId> {  
 public Price monthlyAmountFor(CustomerType customerType) {  
 if (customerType.isVip()) {  
 return this.amount.multiple(1 - DISCOUNT);  
 }  
 return this.amount;  
 }  
}

参与协作的 CustomerType 扮演了数据提供者角色，Rental 类会根据它提供的客户类型执行不同的运算规则。这种协作方式是对象协作的低级形式。正如我在第 1-15 课《领域模型与对象范式（中）》所讲的那样：“对象之间若要默契配合，形成良好的协作关系，就需要通过行为进行协作，而不是让参与协作的对象成为数据的提供者。”这要求参与协作的对象皆为操作自身信息的自治对象，无论是实体、值对象，都是各自履行自己的职责，然后基于业务场景进行行为上的协作。

例如，要计算订单实际应缴的税额，首先应该获得该订单的纳税额度。这个纳税额度等于该订单所属的纳税调节额度汇总值减去手动调节纳税额度的值。在得到订单的纳税额度后，乘以订单的总金额，即为订单实际应缴的税额。订单的纳税调节为另一个实体对象 OrderTaxAdjustment。由于一个订单存在多个纳税调节，因此可以引入一个容器对象 OrderTaxAdjustments，它分别提供了计算纳税调节额度汇总值和手动调节纳税额度值的方法：

public class OrderTaxAdjustments {  
 private List<OrderTaxAdjustment> taxAdjustments;  
 private BigDecimal zero = BigDecimal.ZERO.setScale(taxDecimals, taxRounding);  
  
 public BigDecimal totalTaxAdjustments() {  
 return taxAdjustments  
 .stream  
 .reduce(zero, (ta, agg) -> agg.add(ta.getAmount()));  
 }  
  
 public BigDecimal manuallyAddedTaxAdjustments() {  
 return taxAdjustments  
 .stream  
 .filter(ta -> ta.isManual())  
 .reduce(zero, (ta, agg) -> agg.add(ta.getAmount()));  
 }  
}

Order 实体对象计算税额的领域行为实现为：

public class Order {  
 public BigDecimal calculateTatalTax(OrderTaxAdjustments taxAdjustments) {  
 BigDecimal tatalExistingOrderTax = taxAdjustments.totalTaxAdjustments();  
 BigDecimal tatalManuallyAddedOrderTax = taxAdjustments.manuallyAddedTaxAdjustments();  
 BigDecimal taxDifference = tatalExistingOrderTax.substract(tatalManuallyAddedOrderTax).setScale(taxDecimals, taxRounding);  
  
 return totalAmount().multiply(taxDifference).setScale(taxDecimals, taxRounding);  
 }  
}

Order 与 OrderTaxAdjustments 根据自己拥有的数据各自计算自己的税额部分，从而完成合理的职责协作。这种协作方式体现了职责的分治，如此设计出来的领域对象满足了“自治”特征。

在领域逻辑中，还有一种特殊的领域行为，就是针对实体（包括值对象）进行 **增删改查** 的操作，分别对应增加、删除、修改与查询这四个操作。从对象的角度考虑，这四个操作其实都是对对象生命周期的管理。如果我们将创建的对象放到一个资源库（Repository）中进行管理，则增删改查操作其实就是访问资源库。在领域驱动设计中，针对实体的增删改查操作都分配给了专门的资源库对象。换言之，在领域驱动的设计模型中，实体往往并不承担增删改查的职责。至于本节提及的“变更状态的领域行为”，仅仅针对对象的内存状态进行修改。

除此之外，还有创建行为。领域驱动设计引入了工厂类封装复杂的创建行为，有时候，也可能由实体类扮演工厂角色，提供创建实体对象的能力。无论是增删改查，还是对象的创建，都属于一个对象的生命周期，我会在《领域模型对象的生命周期》一节中专门讲解。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持