## 对象关系映射（下）

### JPA 的应对之道

#### 对象模式的阻抗不匹配

符合面向对象设计原则的领域模型，其中一个重要特征是建立了高内聚低耦合的对象图。要做到这一点，就需得将具有高内聚关系的概念 **封装** 为一个类，通过显式的类型体现领域中的概念，这样既提高了代码的可读性，又保证了职责的合理分配，避免出现一个庞大的实体类。领域驱动设计更强调这一点，并因此还引入了值对象的概念，用以表现那些无需身份标识却又具有内聚知识的领域概念。因此，一个设计良好的领域模型，往往会形成一个具有嵌套层次的对象图模型结构。

虽然嵌套层次的领域模型与扁平结构的关系数据模型并不匹配，但通过 JPA 提供的 @Embedded 与 @Embeddable 标注可以非常容易实现这一嵌套组合的对象关系，例如 Employee 类的 address 属性和 email 属性：

@Entity  
@Table(name="employees")  
public class Employee extends AbstractEntity<EmployeeId> implements AggregateRoot<Employee> {  
 @EmbeddedId  
 private EmployeeId employeeId;  
  
 private String name;  
  
 @Embedded  
 private Email email;  
  
 @Embedded  
 private Address address;  
}  
  
@Embeddable  
public class Address {  
 private String country;  
 private String province;  
 private String city;  
 private String street;  
 private String zip;  
  
 public Address() {  
 }  
}  
  
@Embeddable  
public class Email {  
 @Column(name = "email")  
 private String value;  
  
 public String value() {  
 return this.value;  
 }  
}

以上定义的领域类，都是 Employee 实体的值对象。注意，为了支持 JPA 实现框架通过反射创建对象，若为值对象定义了带参的构造函数，就需要显式定义默认构造函数，如 Address 类的定义。

对比 EmployeeId 类的定义，你会发现该类的定义仍然属于值对象的范畴，只是由于该类型在数据模型中作为主键，故而应将该字段声明为 @EmbeddedId 标注。

无论是 Address、Email 还是 EmployeeId 类，它们在领域对象模型中虽然被定义为独立的类，但在数据模型中却都是 employees 表中的列。其中 Email 类仅仅是表中的一个列，定义为类的目的是体现电子邮件的领域概念，并有利于封装对邮件地址的验证逻辑； Address 类封装了多个内聚的值，体现为 country、province 等列，以利于维护地址概念的完整性，同时也可以实现对领域概念的重用。创建 employees 表的 SQL 脚本如下所示：

CREATE TABLE employees(  
 id VARCHAR(50) NOT NULL,  
 name VARCHAR(20) NOT NULL,  
 email VARCHAR(50) NOT NULL,  
 employeeType SMALLINT NOT NULL,  
 gender VARCHAR(10),  
 salary DECIMAL(10, 2),  
 currency VARCHAR(10),  
 country VARCHAR(20),  
 province VARCHAR(20),  
 city VARCHAR(20),  
 street VARCHAR(100),  
 zip VARCHAR(10),  
 mobilePhone VARCHAR(20),  
 homePhone VARCHAR(20),  
 officePhone VARCHAR(20),  
 onBoardingDate DATE NOT NULL  
 PRIMARY KEY(id)  
);

如果一个值对象在数据模型中被设计为一个独立的表，但由于它无需定义主键，需要依附于一个实体表，因此在领域模型中依旧标记为 @Embeddable。这既体现了面向对象的封装思想，又表达了一对一或一对多的关系。SalariedEmployee 聚合中的 Absence 值对象就遵循了这样的设计原则。

面向对象的封装思想体现了对细节的隐藏，正确的封装还体现为对职责的合理分配。遵循“信息专家模式”，无论是领域模型中的实体，还是值对象，都应该从它们拥有的数据出发，判断领域行为是否应该分配给这些领域模型类。如 HourlyEmployee 实体类的 payroll(Period) 方法、Absence 值对象的 isIn(Period) 与 isPaidLeave() 方法，乃至于 Salary 值对象的 add(Salary) 等方法，都充分体现了对领域行为的合理封装，避免了贫血模型的出现：

public class HourlyEmployee extends AbstractEntity<EmployeeId> implements AggregateRoot<HourlyEmployee> {  
 public Payroll payroll(Period period) {  
 if (Objects.isNull(timeCards) || timeCards.isEmpty()) {  
 return new Payroll(this.employeeId, period.beginDate(), period.endDate(), Salary.zero());  
 }  
  
 Salary regularSalary = calculateRegularSalary(period);  
 Salary overtimeSalary = calculateOvertimeSalary(period);  
 Salary totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(this.employeeId, period.beginDate(), period.endDate(), totalSalary);  
 }  
}  
  
public class Absence {  
 public boolean isIn(Period period) {  
 return period.contains(leaveDate);  
 }  
  
 public boolean isPaidLeave() {  
 return leaveReason.isPaidLeave();  
 }  
}  
  
public class Salary {  
 public Salary add(Salary salary) {  
 throwExceptionIfNotSameCurrency(salary);  
 return new Salary(value.add(salary.value).setScale(SCALE), currency);  
 }  
  
 public Salary subtract(Salary salary) {  
 throwExceptionIfNotSameCurrency(salary);  
 return new Salary(value.subtract(salary.value).setScale(SCALE), currency);  
 }  
  
 public Salary multiply(double factor) {  
 return new Salary(value.multiply(toBigDecimal(factor)).setScale(SCALE), currency);  
 }  
  
 public Salary divide(double multiplicand) {  
 return new Salary(value.divide(toBigDecimal(multiplicand), SCALE, BigDecimal.ROUND\_DOWN), currency);  
 }  
}

这充分证明领域模型对象既可以作为持久化对象，搭建起对象与关系表之间的桥梁；又可以体现丰富的包含领域行为在内的领域概念与领域知识。合二者为一体的领域模型对象被定义在领域层，位于基础设施层的资源库实现可以访问它们，避免定义重复的领域模型与数据模型。

对象模式中的继承更为特殊，因为关系表自身不具备继承能力，这与对象之间的组合关系不同。若仅仅为了重用而使用继承，那么在数据模型中只需保证关系表的列无需重复定义即可。因此，可以简单地将继承了父类的子类看做是一张关系表，父类与所有子类对应的字段都放在这一张表中，就好似对集合求并集一般。这种策略在 ORM 中被称之为 Single-Table 策略。为了区分子类，这一张单表必须额外定义一个列，作为区分子类的标识列，在 JPA 中被定义为 @DiscriminatorColumn。例如，如果需要为 Employee 建立继承体系，则它的标识列就是 employeeType 列。

若子类之间的差异太大，采用 Single-Table 策略实现继承的方式会让表的冗余显得格外明显。因为有的子类并没有这些列，却不得不为属于该类型的行记录提供这些列的存储空间。要避免这种冗余，可以采用 Joined-Subclass 策略实现继承。采用这种策略时，继承关系中的每一个实体类，无论是具体类还是抽象类，数据库中都有一个单独的表与之对应。子实体对应的表无需定义从根实体继承而来的列，而是通过共享主键的方式进行关联。

由于 Single-Table 策略是 ORM 默认的继承策略，若要采用 Joined-Subclass 策略，需要在父实体类的定义中显式声明其继承策略，如下所示：

@Entity   
@Inheritance(strategy=InheritanceType.JOINED)   
@Table(name="employees")   
public class Employee {}

采用 Joined-Subclass 策略实现继承时，数据模型中子实体表与父实体表之间的关系实则是一对一的连接关系，这可以认为是为了解决对象模式阻抗不匹配的无奈之举，毕竟用连接关联关系表达继承，怎么看都显得有些别扭。当领域模型中继承体系的子类较多时，这一设计还会影响查询效率，因为它可能牵涉到多张表的连接。

如果既不希望产生不必要的数据冗余，又不愿意表连接拖慢查询的速度，则可以采用 Table-Per-Class 策略。采用这种策略时，继承体系中的每个实体类都对应一个独立的表，其中，父实体对应的表仅包含父实体的字段，子实体对应的表不仅包含了自身的字段，同时还包含了父实体的字段。这相当于用数据表样式的冗余来避免数据的冗余，用单表来避免不必要的连接。如果子类之间的差异较大，我更倾向于采用 Table-Per-Class 策略，而非 Joined-Subclass 策略。

继承的目的绝不仅仅是为了重用，甚至可以说重用并非它的主要价值，毕竟“聚合/合成优先重用原则”已经成为了面向对象设计的金科玉律。继承的主要价值在于支持多态，这样就能利用 Liskov 替换原则，子类能够替换父类而不改变其行为，并允许定义新的子类来满足功能扩展的需求，保证对扩展是开放的。在 Java 或 C# 这样的语言中，由于受到单继承的约束，定义抽象接口以实现多态更为普遍。无论是继承多态还是接口多态，都应站在领域逻辑的角度，思考是否需要引入合理的抽象来应对未来需求的变化。在采用继承多态时，需要考虑对应的数据模型是否能够在对象关系映射中实现继承，并选择合理的继承策略来确定关系表的设计。至于接口多态是对领域行为的抽象，与领域模型的持久化无关，在定义抽象接口时，无需考虑领域模型与数据模型之间的映射。

#### 与持久化无关的领域模型

并非所有的领域模型对象都需要持久化到数据表，一些领域概念之所以定义为值对象，仅仅是为了封装领域行为，表达一种高内聚的领域概念，以便于领域对象更好地分配职责，隐藏实现细节，支持良好的行为协作。例如，与 HourlyEmployee 聚合根交互的 Period 类，其作用是体现一个结算周期，作为薪资计算的条件：

public class Period {  
 private LocalDate beginDate;  
 private LocalDate endDate;  
  
 public Period(LocalDate beginDate, LocalDate endDate) {  
 this.beginDate = beginDate;  
 this.endDate = endDate;  
 }  
  
 public Period(YearMonth yearMonth) {  
 int year = yearMonth.getYear();  
 int month = yearMonth.getMonthValue();  
 int firstDay = 1;  
 int lastDay = yearMonth.lengthOfMonth();  
  
 this.beginDate = LocalDate.of(year, month, firstDay);  
 this.endDate = LocalDate.of(year, month, lastDay);  
 }  
  
 public Period(int year, int month) {  
 if (month < 1 || month > 12) {  
 throw new InvalidDateException("Invalid month value.");  
 }  
  
 int firstDay = 1;  
 int lastDay = YearMonth.of(year, month).lengthOfMonth();  
  
 this.beginDate = LocalDate.of(year, month, firstDay);  
 this.endDate = LocalDate.of(year, month, lastDay);  
 }  
  
 public LocalDate beginDate() {  
 return beginDate;  
 }  
  
 public LocalDate endDate() {  
 return endDate;  
 }  
  
 public boolean contains(LocalDate date) {  
 if (date.isEqual(beginDate) || date.isEqual(endDate)) {  
 return true;  
 }  
 return date.isAfter(beginDate) && date.isBefore(endDate);  
 }  
}

结算周期必须提供成对儿的起止日期，缺少任何一个日期，就无法正确地进行薪资计算。将 beginDate 与 endDate 封装到 Period 类中，再利用构造函数限制实例的创建，就能避免起止日期任意一个值的缺失。引入 Period 类还能封装领域行为，让对象之间的协作变得更加合理。由于这样的类没有声明 @Entity，因此是一种 POJO 类。因为它并不需要持久化，为示区别，可称呼这样的类为瞬态类（Transient Class）。对应的，倘若在一个支持持久化的领域类中，需要定义一个无需持久化的字段，可称呼这样的字段为瞬态字段（Transient Field）。JPA 定义了 @Transient 标注用以显式声明这样的字段，例如：

@Entity  
@Table(name="employees")  
public class Employee extends AbstractEntity<EmployeeId> implements AggregateRoot<Employee> {  
 @EmbeddedId  
 private EmployeeId employeeId;  
  
 private String firstName;  
 private String middleName;  
 private String lastName;  
  
 @Transient  
 private String fullName;  
}

Employee 类对应的数据表定义了 firstName、middleName 与 lastName 列，为了调用方便，该类又定义了 fullName 字段，该值并不需要持久化到数据库中，因此需声明为瞬态字段。

理想的领域模型类应该如瞬态类这样的 POJO 类，这也符合整洁架构的思想，即处于内部核心的领域类不依赖任何外部框架。由于需要为领域模型与数据模型建立关系映射，就必须通过某种元数据机制对其进行表达，ORM 框架才能实现对象与关系的映射。在 Java 语言中，可供选择的元数据机制就是 XML 或标注（Annotation）。XML 因其冗长繁杂与不直观的表现力等缺陷，在相对大型的产品或项目开发中，已被渐渐摒弃，因而更建议使用标注。由于 JPA 是 Oracle（Sun）为持久化接口制定的规范，我们也可自我安慰地说，这些运用到领域模型类上的标注仍然属于 Java 语言的一部分，不算是违背整洁架构的设计原则。

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