## 对象关系映射（上）

### 领域模型的持久化

领域驱动设计强调对领域建模来应对业务复杂度，通过分层架构来隔离业务复杂度与技术复杂度，这就使得我们在考虑领域逻辑时，尽量规避对领域模型持久化的考虑，引入抽象的资源库正是为了解决这一问题。领域驱动设计的驱动力是领域逻辑而非数据库样式，因此是先有领域模型，然后再根据领域模型定义数据模型，此为领域模型驱动设计与数据模型驱动设计的根本区别。

### 对象关系映射

领域模型是面向对象的，数据模型是面向关系表的。倘若采用领域模型驱动设计，领域模型一方面充分地表达了系统的领域逻辑，同时它还将映射为数据模型，成为操作数据库的持久化对象。这就是采用面向对象设计编写基础设施层的持久化功能时，无法绕过的对象关系映射（Object Relationship Mapping，ORM）。

#### 对象关系阻抗不匹配

如果持久化的数据库为关系数据库，就会出现所谓“对象关系阻抗不匹配”的问题。这种阻抗不匹配主要体现为以下三个方面：

* 类型的阻抗不匹配：例如不同关系型数据库对浮点数的不同表示，字符串类型在数据库的最大长度约束等，又例如 Java 等语言的枚举内建类型本质上仍然属于基础类型，关系数据库中却没有对应的类型来匹配。
* 样式的阻抗不匹配：领域模型与数据模型不具备一一对应的关系。领域模型是一个具有嵌套层次的对象图结构，数据模型在关系数据库中却是扁平的关系结构，要让数据库能够表示领域模型，就只能通过关系来变通地映射实现。
* 对象模式的阻抗不匹配：面向对象的封装、继承与多态无法在关系数据库得到直观体现。通过封装可以定义一个高内聚的类来表达一个细粒度的基本概念，但数据表往往不这么设计；数据表只有组合关系，无法表达对象之间的继承关系；既然无法实现继承关系，就无法满足 Liskov 替换原则，自然也就无法满足多态。

ORM 框架正是为了解决这些阻抗不匹配问题应运而生，这个问题如此的重要，因此 Java 语言甚至定义了持久化的规范，用以指导面向对象的语言要素与关系数据表之间的映射，如 SUN 在 JDK 5 中引入的 JPA（Java Persistence API），作为 JCP 组织发布的 Java EE 标准，就起到了在 Java 社区指导 ORM 技术实现的规范。

### JPA 的应对之道

顾名思义，ORM 框架的目的是在对象与关系之间建立一种映射。为了满足这一目标，往往通过配置文件或者在领域模型中声明元数据来表现这种映射关系。JPA 作为一种规范，它全面地考虑了各种阻抗不匹配的情形，然后规定了标准的映射元数据，如 @Entity、@Table 和 @Column 等 Java 标注。一旦领域模型声明了这些标注，具体的JPA框架如 Hibernate 等就可以通过反射识别这些元数据，获得对象与关系之间的映射信息，从而实现领域模型的持久化。

#### 类型的阻抗不匹配

针对类型的阻抗不匹配，JPA 元数据通过 @Column 标注的属性来指定长度、精度还有对 null 的支持；通过 Lob 标注来表示字节数组；通过 @ElementCollection 等标注来表达集合。至于枚举、日期和 ID 等特殊类型，JPA 也针对性地给出了元数据定义。

**枚举类型**

关系数据库的内建类型没有枚举类型。如果领域模型的字段被定义为自定义的枚举，通常会在数据库中将相应的列定义为 smallint 类型，然后通过 @Enumerated 表示枚举的含义，例如：

public enum EmployeeType {  
 Hourly, Salaried, Commission  
}  
  
public class Employee {  
 @Enumerated  
 @Column(columnDefinition = "smallint")  
 private EmployeeType employeeType;  
}

使用 smallint 表示枚举类型虽然能够体现值的有序性，但在管理和运维数据库时，查询得到的枚举值却是没有任何业务含义的数字，这不利于对数据的理解。这时，可以将这样的列定义为 VARCHAR，而在领域模型中声明为：

public enum Gender {  
 Male, Female  
}  
  
public class Employee {  
 @Enumerated(EnumType.STRING)  
 private Gender gender;  
}

通过在字段上标注 @Enumerated(EnumType.STRING)，可以将枚举类型转换为字符串。注意，数据库的字符串应与枚举类型的字符串值以及大小写保持一致。

**日期类型**

针对 Java 的日期和时间类型进行映射，处理要相对复杂一些。因为 Java 定义了多种日期和时间类型，包括：

* 用以表达数据库日期类型的 java.sql.Date 类和表达数据库时间类型的 java.sql.Timestamp 类
* Java 库用以表达日期、时间与时间戳类型的 java.util.Date 类或 java.util.Calendar 类
* Java 8 引入的新日期类型 java.time.LocalDate 类与新时间类型 java.time.LocalDateTime 类

当领域模型对象的日期或时间字段被定义为 java.sql.Date 或 java.sql.Timestamp 类型时，由于数据库支持这一类型，因此无需做任何特别的配置。通过 columnDefinition 属性值，甚至可以设置默认值，例如：

@Column(name = "START\_DATE", columnDefinition = "DATE DEFAULT CURRENT\_DATE")  
private java.sql.Date startDate;

如果字段被定义为 java.util.Date 或 java.util.Calendar 类型，JPA 定义了 @Temporal 标注将其映射为日期、时间或时间戳，例如：

@Temporal(TemporalType.DATE)  
private java.util.Calendar birthday;  
  
@Temporal(TemporalType.TIME)  
private java.util.Date birthday;  
  
@Temporal(TemporalType.TIMESTAMP)  
private java.util.Date birthday;

如果字段被定义为 Java 8 新引入的 LocalDate 或 LocalDateTime 类型时，情况稍显复杂，需要取决于 JPA 的版本。JPA 2.2 版本已经支持 Java 8 日期时间 API 中除 java.time.Duration 外的其他日期和时间类型。因此，若选择了这个版本的 JPA，无需再为 JDK 8 的日期或时间类型做任何设置，与诸如 String、int 等类型一视同仁。

如果 JPA 的版本是 2.1 及以下版本，由于这些版本发布在 Java 8 之前，因此无法直接支持这两种类型，需要为其定义 AttributeConverter。例如为 LocalDate 定义转换器：

import javax.persistence.AttributeConverter;  
import javax.persistence.Converter;  
import java.sql.Date;  
import java.time.LocalDate;  
  
@Converter(autoApply = true)  
public class LocalDateAttributeConverter implements AttributeConverter<LocalDate, Date> {  
 @Override  
 public Date convertToDatabaseColumn(LocalDate locDate) {  
 return locDate == null ? null : Date.valueOf(locDate);  
 }  
  
 @Override  
 public LocalDate convertToEntityAttribute(Date sqlDate) {  
 return sqlDate == null ? null : sqlDate.toLocalDate();  
 }  
}

**主键类型**

在关系数据库中，每个表的主键都是至为关键的列，通过它可以标注每一行记录的唯一性。主键还是建立表关联的关键列，通过主键与外键的关系可以间接支持领域模型对象之间的导航，同时也保证了关系数据库的完整性。无论是单一主键还是联合主键，主键作为身份标识（Identity），只要能够确保它在同一张表中的唯一性，原则上可以定义为各种类型，如 BigInt、VARCHAR 等。在数据表定义中，只要某个列被声明为 PRIMARY KEY，在领域模型对象的定义中，就可以使用 JPA 提供的 @Id 标注。这个标注还可以和 @Column 标注组合使用：

@Id  
@Column(name = "employeeId")  
private int id;

主流的关系数据库都支持主键的自动生成，JPA 提供了 @GeneratedValue 标注说明了该主键是自动生成的。该标注还定义了 strategy 属性用以指定自动生成的策略。JPA 还定义了 @SequenceGenerator 与 @TableGenerator 等特殊的 ID 生成器。

在建立领域模型时，我们强调从领域逻辑出发考虑领域类的定义。尤其对于实体类而言，ID 代表的是实体对象的身份标识。它与数据表的主键有相似之处，例如都要求唯一性，但二者的本质完全不同：前者代表业务含义，后者代表技术含义。前者用于对实体对象生命周期的管理与跟踪，后者用于标记每一行在数据表中的唯一性。因此，领域驱动设计往往建议定义 Identity 值对象作为实体的身份标识。一方面，值对象类型可以清晰表达该身份标识的业务含义；另一方面值对象类型的封装也有利于应对未来主键类型可能的变化。

Identity 值对象的定义，体现了面向对象的封装思想，JPA 定义了一个特殊的标注 @EmbeddedId 来建立数据表主键与身份标识值对象之间的映射。例如，为 Employee 实体对象定义了 EmployeeId 值对象，则 Employee 的定义为：

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

JPA 对主键类有两个要求：相等性比较与序列化支持。这就需要 EmployeeId 实现 Serializable 接口，并重写 Object 的 equals() 与 hashcode() 方法，同时在类定义之上声明 Embeddable 标注：

@Embeddable  
public class EmployeeId implements Identity<String>, Serializable {  
 @Column(name = "id")  
 private String value;  
  
 private static Random random;  
  
 static {  
 random = new Random();  
 }  
  
 // 必须提供默认的构造函数  
 public EmployeeId() {  
 }  
  
 private EmployeeId(String value) {  
 this.value = value;  
 }  
  
 @Override  
 public String value() {  
 return this.value;  
 }  
  
 public static EmployeeId of(String value) {  
 return new EmployeeId(value);  
 }  
  
 public static Identity<String> next() {  
 return new EmployeeId(String.format("%s%s%s",  
 composePrefix(),  
 composeTimestamp(),  
 composeRandomNumber()));  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 EmployeeId that = (EmployeeId) o;  
 return value.equals(that.value);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.hash(value);  
 }  
}

使用时，可以直接传入 EmployeeId 对象作为主键查询条件：

Optional<Employee> optEmployee = employeeRepo.findById(EmployeeId.of("emp200109101000001"));

#### 样式的阻抗不匹配

样式（Schema）的阻抗不匹配，实则就是对象图与关系表之间的不匹配。要做到二者的匹配，就需要做到图结构与表结构之间的互相转换。在领域模型的对象图中，一个实体组合了另一个实体，由于两个实体都有各自的身份标识，因此在数据库中可以通过主外键关系建立关联。这些关联关系分别体现为一对一、一对多或者多对一、多对多。

例如，在领域模型中，HourlyEmployee 聚合根实体与 TimeCard 实体之间的关系可以定义为：

@Entity  
@Table(name="employees")  
public class HourlyEmployee extends AbstractEntity<EmployeeId> implements AggregateRoot<HourlyEmployee> {  
 @OneToMany  
 @JoinColumn(name = "employeeId", nullable = false)  
 private List<TimeCard> timeCards = new ArrayList<>();  
}  
  
@Entity  
@Table(name = "timecards")  
public class TimeCard {  
 private static final int MAXIMUM\_REGULAR\_HOURS = 8;  
  
 @Id  
 @GeneratedValue  
 private String id;  
 private LocalDate workDay;  
 private int workHours;  
  
 public TimeCard() {  
 }  
}

在数据模型中，timecards 表则通过外键 employeeId 建立与 employees 表之间的关联：

CREATE TABLE employees(  
 id VARCHAR(50) NOT NULL,  
 ......  
 PRIMARY KEY(id)  
);  
  
CREATE TABLE timecards(  
 id INT NOT NULL AUTO\_INCREMENT,  
 employeeId VARCHAR(50) NOT NULL,  
 workDay DATE NOT NULL,  
 workHours INT NOT NULL,  
 PRIMARY KEY(id)  
);

如果对象图的组合关系发生在一个实体和值对象之间，并形成一对多的关联。由于值对象没有唯一的身份标识，它的数据模型也没有主键，而是将实体表的主键作为外键，由此来表达彼此之间的归属关系。这时，领域模型仍然通过集合来表达一对多的关联，但使用的标注却并非 @OneToMany，而是 @ElementCollection。例如，领域模型中的 SalariedEmployee 聚合根实体与 Absence 值对象之间的关系可以定义为：

@Embeddable  
public class Absence {  
 private LocalDate leaveDate;  
  
 @Enumerated(EnumType.STRING)  
 private LeaveReason leaveReason;  
  
 public Absence() {  
 }  
  
 public Address(String country, String province, String city, String street, String zip) {  
 this.country = country;  
 this.province = province;  
 this.city = city;  
 this.street = street;  
 this.zip = zip;  
 }  
}  
  
@Entity  
@Table(name="employees")  
public class SalariedEmployee extends AbstractEntity<EmployeeId> implements AggregateRoot<SalariedEmployee> {  
 private static final int WORK\_DAYS\_OF\_MONTH = 22;  
  
 @EmbeddedId  
 private EmployeeId employeeId;  
  
 @Embedded  
 private Salary salaryOfMonth;  
  
 @ElementCollection  
 @CollectionTable(name = "absences", joinColumns = @JoinColumn(name = "employeeId"))  
 private List<Absence> absences = new ArrayList<>();  
  
 public SalariedEmployee() {  
 }  
}

@ElementCollection 说明了字段 absences 是 SalariedEmployee 实体的字段元素，类型为集合；@CollectionTable 标记了关联的数据表以及关联的外键。其数据模型如下 SQL 语句所示：

CREATE TABLE employees(  
 id VARCHAR(50) NOT NULL,  
 ......  
 PRIMARY KEY(id)  
);  
  
CREATE TABLE absences(  
 employeeId VARCHAR(50) NOT NULL,  
 leaveDate DATE NOT NULL,  
 leaveReason VARCHAR(20) NOT NULL  
);

数据表 absences 没有自己的主键，employeeId 列是 employees 表的主键。注意，在 Absence 值对象的定义中，无需再定义 employeeId 字段，因为 Absence 值对象并不能脱离 SalariedEmployee 聚合根单独存在。

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