5 Herança

Objetivos

- Apresentar as estratégias para herança;
- Identificar estratégias de herança em anotação JPA;
- Exemplificar o tratamento da herança nas tabelas geradas no banco.

Introdução

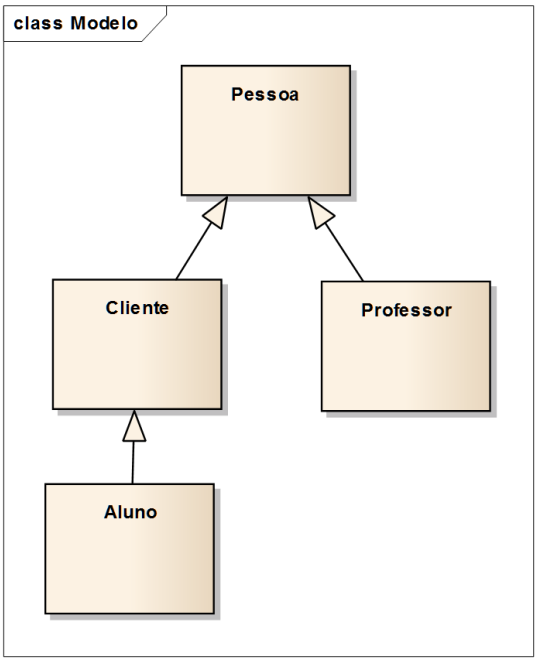
Bancos de dados relacionais não suportam herança de forma nativa, forçando o mapeamento das estruturas de herança do modelo conceitual dentro do próprio modelo físico de dados. Há certa flexibilidade em relação à ampla utilização de estruturas de herança por parte de desenvolvedores OO, devido em parte ao problema fragilização e acoplamento da hierarquia de classes. No entanto, este problema pode ser atribuído, na sua maior parte, a práticas pobres do encapsulamento entre objetos colaboradores do que ao conceito de herança propriamente dito.

Mesmo levando-se em conta o fato que é necessário um trabalho considerável para mapear uma hierarquia de objetos em um banco de dados relacional, isto não deve dissuadir o desenvolvedor a utilizar herança quando julgá-la apropriada.

A especificação JPA fornece três formas diferentes para se mapear uma hierarquia de herança para uma base de dados relacional:

- Uma única tabela por hierarquia de classes: Uma tabela terá todas as propriedades de todas as classes na hierarquia.
- Uma tabela por classe concreta: Cada classe terá uma tabela correspondente, com todas as suas propriedades e as propriedades de suas superclasses mapeadas para essa tabela. **Esta implementação é opcional na JPA, ou seja, dependendo do framework ela não esta disponível.**
- Uma tabela por subclasse: Cada classe terá sua própria tabela. Cada tabela terá apenas as propriedades que são definidas naquela classe particular. Essas tabelas não terão propriedades de nenhuma superclasse ou subclasse.

Os exemplos desse capítulo irão considerar a hierarquia de classes abaixo.



Uma Tabela por Hierarquia de Classes

Nessa estratégia, uma tabela representa todas as classes de uma hierarquia. Considerando a hierarquia de herança da figura acima, as classes Pessoa, Cliente e Empregado são representadas em uma mesma tabela:

```
// contém propriedades de Pessoa, Cliente e Empregado
create table PESSOA_HIERARQUIA (
     id integer primary key not null,
     ...
     DISCRIMINADOR varchar(31) not null
);
```

A tabela Pessoa_Hierarquia conterá, então, as propriedades de todas as classes dessa hierarquia. Mas, essa estratégia de mapeamento requer também uma coluna adicional discriminadora na tabela, que identifica o tipo de entidade armazenada em uma dada linha da mesma. Assim, para mapear essa estratégia, as classes da hierarquia devem usar determinadas anotações:

```
@Entity
@Table(name="PESSOA_HIERARQUIA")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINADOR",
  discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Pessoa")
public class Pessoa {
   ....
   ....
}

@Entity
@DiscriminatorValue("Cliente")
public class Cliente extends Pessoa {
   ....
}

// usa o valor default do discriminator (nome da classe)
@Entity
public class Empregado extends Cliente {
   ....
   ....
}
```

A anotação @javax.persistence.Inheritance é usada para definir a estratégia de persistência para o relacionamento de herança:

```
package javax.persistence;
```

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Inheritance {
  InheritanceType strategy() default SINGLE_TABLE;
}

public enum InheritanceType {
  SINGLE_TABLE, JOINED, TABLE_PER_CLASS
}
```

O atributo strategy() define o mapeamento de herança que será usado, podendo assumir um dos tipos definidos por InheritanceType. A anotação @Inheritance precisa ser usada apenas na raiz da hierarquia de classes.

```
package javax.persistence;
@Target(TYPE) @Retention(RUNTIME)
public @interface DiscriminatorColumn {
  String name() default "DTYPE";
  DiscriminatorType discriminatorType() default STRING;
  String columnDefinition() default "";
  int length() default 10;
}
```

Como apenas uma tabela representa toda a hierarquia de classes, o gerenciador de persistência precisa de alguma forma identificar qual classe cada linha da tabela representa e ele pode determinar isto lendo o valor da coluna discriminadora. A anotação @javax.persistence.DiscriminatorColumn é que identifica esta coluna. O atributo name() identifica o nome da coluna e o atributo discriminatorType() especifica qual tipo ela será. O tipo pode ser STRING, CHAR ou INTEGER.

```
package javax.persistence;
@Target(TYPE) @Retention(RUNTIME)
public @interface DiscriminatorValue {
  String value()
}
```

A anotação @DiscriminatorValue define qual valor a coluna discriminadora terá para as linhas que armazenam uma instância da classe onde a anotação foi especificada. Você pode não especificar esse valor, nesse caso, será gerado um valor automaticamente. Quando é usado o tipo STRING, o valor gerado será o nome da classe.

Vantagens

A estratégia de mapeamento SINGLE_TABLE é a mais simples de implementar e executa melhor que todas as outras estratégias. Há apenas uma tabela para administrar.

Desvantagens

Uma grande desvantagem desse método é que todas as colunas de propriedades de subclasses devem poder ficar sem valores. Então, se você precisa ou quer ter restrições NOT NULL nestas colunas, você não pode. Isto porque estas colunas podem não ser usadas. Essa estratégia não é normalizada.

Uma Tabela por Classe Concreta

Nessa estratégia, uma tabela é definida para cada classe concreta da hierarquia. Além disso, cada tabela tem colunas representando suas propriedades e todas as propriedades de qualquer superclasse.

```
// contém propriedades apenas da classe Pessoa
create table Pessoa (
  id integer primary key not null,
  ....
);
// contém propriedades da classe Cliente e de Pessoa
create table Cliente (
  id integer primary key not null,
  ....
);
// contém propriedades da classe Empregado, Cliente e Pessoa
create table Empregado (
  id integer primary key not null,
  ....
);
```

A maior diferença entre essa estratégia e a SINGLE_TABLE é que não é necessária uma coluna discriminadora na base de dados. Além disso, cada tabela contém todas as propriedades da hierarquia a serem persistidas. Essa estratégia é mapeada com as anotações seguintes:

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Pessoa {
  ....
}

@Entity
public class Cliente extends Pessoa {
  ....
}

@Entity
public class Empregado extends Cliente {
  ....
}
```

O único meta-dado requerido é o InheritanceType e ele precisa ser especificado apenas para a classe base Pessoa.

Vantagens

A vantagem desse método sobre o SINGLE_TABLE é que você pode definir restrições para propriedades de subclasses. Além disso, ele pode tornar mais fácil o mapeamento de um sistema de modelo de dados legado.

Desvantagens

Essa estratégia não é normalizada porque possui colunas redundantes em suas tabelas, referentes a propriedades de classes base. Ainda, para suportar esse tipo de mapeamento, o gerenciador de persistência tem que fazer algumas operações custosas. Devido a isto, não é recomendado usar essa estratégia, exceto em situações em que os dados não ficariam duplicados em diferentes tabelas.

Uma Tabela por Subclasse

Nessa estratégia, cada subclasse tem sua própria tabela, mas esta tabela contém apenas as propriedades que foram definidas naquela classe em particular. Ou seja, é similar à estratégia TABLE_PER_CLASS, exceto porque a estrutura é normalizada. Esse método é também chamado de estratégia JOINED.

```
// contém propriedades apenas da classe Pessoa
create table Pessoa (
  id integer primary key not null,
  ....
);
// contém propriedades apenas da classe Cliente
create table Cliente (
  id integer primary key not null,
  ....
);
// contém propriedades apenas da classe Empregado
create table Empregado (
  EMP_PK integer primary key not null,
  ....
);
```

Quando o gerenciador de persistência carrega uma entidade que é uma subclasse, ele faz um join SQL em todas as tabelas da hierarquia. Neste mapeamento, deve haver uma coluna em cada tabela que pode ser usada para fazer o join destas. No exemplo considerado aqui, as tabelas Pessoa, Cliente e Empregado compartilham os mesmos valores de chave primária. As anotações deste mapeamento são simples:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa {
  ....
}

@Entity
public class Cliente extends Pessoa {
  ....
}

@Entity
@PrimaryKeyJoinColumn(name="EMP_PK")
public class Empregado extends Cliente {
  ....
}
```

O gerenciador de persistência precisa saber quais colunas em cada tabela serão usadas para executar um join ao carregar uma entidade com essa estratégia de herança. A anotação @javax.persistence.PrimaryKeyJoinColumn pode ser usada para descrever esse meta-dado:

```
package javax.persistence;
@Target({TYPE, METHOD, FIELD})
public @interface PrimaryKeyJoinColumn {
  String name() default "";
  String referencedColumnName() default "";
  String columnDefinition() default "";
}
```

O atributo name() identifica a coluna em que você executará um join na tabela atual e seu default é a coluna de chave primária da tabela da superclasse. Já o atributo referencedColumnName() indica a coluna que será usada para executar um join a partir da tabela da superclasse. Ele pode indicar qualquer coluna da tabela da superclasse, mas seu default é a coluna de chave primária. Se o nome da coluna de chave primária é exatamente o mesmo na superclasse e nas subclasses, então essa anotação não é necessária. Por exemplo, a classe Cliente não precisa ter a anotação @PrimaryKeyJoinColumn. Mas, a classe Empregado, que tem o nome da coluna de chave primária diferente daquele das tabelas de suas superclasses, precisa ter a anotação.

Se a hierarquia de classes usa uma chave composta, a anotação @PrimaryKeyJoinColumns pode ser usada para indicar múltiplas colunas para a execução do join.

Vantagens

Embora essa estratégia não seja tão rápida como a SINGLE_TABLE, você pode definir restrições NOT NULL em qualquer coluna de qualquer tabela e ainda, seu modelo é normalizado. Esse método também é melhor que o TABLE_PER_CLASS por duas razões: o modelo da base de dados relacional é totalmente normalizado e executa melhor que o TABLE_PER_CLASS.

Desvantagens

Não executa tão rápido como o SINGLE_TABLE.

Combinação de Estratégias: Classes Bases que Não São Entidades

As estratégias de mapeamento de herança anteriores referem-se a uma hierarquia de classes de entidades. No entanto, algumas vezes, você precisa herdar de uma superclasse que não representa uma entidade. Esta superclasse pode ser uma classe existente no modelo de domínio, mas que você não quer tornar uma entidade. A anotação @javax.persistence.MappedSuperclass permite definir esse tipo de mapeamento. Para exemplificar, vamos tornar a classe Pessoa uma classe mapeada:

```
@MappedSuperclass
public class Pessoa {
   private int id;
   private String primeiroNome;
   private String ultimoNome;
   ....
}
@Entity
@Table(name="CLIENTE")
@Inheritance(strategy=InheritanceType.JOINED)
@AttributeOverride(name="ultimoNome", column=@Column(name="sobrenome"))
public class Cliente extends Pessoa {
   ...
}
@Entity
@Table(name="EMPREGADO")
@PrimaryKeyJoinColumn(name="EMP_PK")
public class Empregado extends Cliente {
   ....
}
```

Como a superclasse não é uma entidade, ela não terá uma tabela associada. Todas as subclasses herdam as propriedades de persistência da classe base e podem sobrescrever qualquer propriedade da classe mapeada usando a anotação @javax.persistence.AttributeOverride. Então, a estrutura da base de dados do exemplo ficaria assim:

```
// contém propriedades da classe Cliente e de Pessoa
create table CLIENTE (
   id integer primary key not null,
   primeiroNome varchar(255),
   sobrenome varchar(255),
   .....
)

// contém propriedades da classe Empregado
create table EMPREGADO (
   EMP_PK integer primary key not null,
   ...
)
```

Como você pode ver, a classe Cliente herda as propriedades id, primeiroNome e ultimoNome. No entanto, como a anotação @AttributeOverride foi usada, o nome da

coluna na tabela Cliente para ultimoNome foi trocado para sobrenome. Esta estratégia é útil quando você quer ter uma classe base para uma entidade, mas não quer que essa classe base tenha uma tabela correspondente.

Classes que não recebem a anotação @Entity e nem @MappedSuperclass são completam