Mapeamento Objeto Relacional

Spring Data - JPA

Definição

Mapeamento Objeto Relacional é “persistir de maneira automática e transparente, os objetos de um aplicativo para tabelas em um banco de dados relacional. Em essência, transforma dados de uma representação para a outra”.

No desenvolvimento de um sistema, muitas vezes o programador dedica boa parte do tempo de desenvolvimento construindo comandos de instruções SQL para realizar a persistência dos dados no banco de dados relacional. O aplicativo precisará de uma camada de mapeamento objeto relacional, que irá traduzir as estruturas e operações do sistema orientado a objetos para o banco de dados relacional.

Persistência

Capacidade de preservar os dados entrados pelo usuário após o programa ter sido fechado;

Em Java, várias formas:

- Escrita direta em arquivo (texto ou binário);
- Serialização (com ou sem framework);
- Banco de dados relacionais (SGBDR) com JDBC;
- Banco de dados orientado a objetos (SGBDOO);
- Etc.

Em sistemas de informação, o uso de SGBDRs é o mais comum.

Persistência SGBD Relacional

Comunicação por meio de sentenças SQL:

- Criação e alteração de tabelas;
- Inserção, atualização e exclusão de dados;
- Restrições, projeções e junções;
- Agrupamentos, ordenação e agregação;
- Etc.

Conexão ao SGBD por meio de um driver;

Padronização da API via JDBC;

Tarefa tediosa e propensa a erros.

Aplicativos usando SGBD Relacional

Softwares simples podem ser construídos sobre o acesso JDBC:

- Lógica de negócio trabalha com linhas e colunas.

Softwares mais complexos possuem um modelo de domínio:

- Classes que representam objetos do domínio do problema;
- Utilização de conceitos OO como polimorfismo;
- Lógica de negócio trabalha com objetos.

Aproximadamente 30% do código é feito para manipular dados via SQL/JDBC;

Estruturas são repetidas em comandos INSERT, UPDATE DELETE e SELECT;

O modelo de objetos geralmente é “torcido” para se adequar ao modelo de dados;

Software de difícil manutenção.

Aplicativos usando SGBD Relacional

Divisão em camadas é senso comum;

Alternativas para camada de persistência:

SQL/JDBC codificado à mão	Desperdício de esforço, baixa produtividade e manutenção, desempenho possivelmente inferior às soluções já existentes.
Serialização	Acesso ao grafo como um todo, não permite buscas, problemas de concorrência.
EJBs de Entidade	Torce o modelo de objetos, sem suporte a polimorfismo e herança, não são portáveis na prática, não são serializáveis, modelo intrusivo que dificulta testes unitários.
Bancos de Dados OO	Baixa aceitação pelo mercado, padrão imaturo.

Solução para ODBC (JDBC)

Também conhecida como:

- Object/Relational Mapping (ORM);
- Gateway-based Object Persistence (GOP).

> *Persistência automática e transparente de objetos de um aplicativo Java para tabelas em um banco de dados relacional, utilizando meta-dados que descrevem o mapeamento entre os objetos e o banco de dados. Em essência, transforma dados de uma representação para a outra.*
>
> Hibernate in Action

Componentes de uma solução ORM

API para efetivação de operações CRUD;

Linguagem ou API para construção de consultas que se refiram às classes ou suas propriedades;

Mecanismo de especificação dos meta-dados de mapeamento;

Técnicas de interação com o SGBDR, incluindo:

- Verificação de objetos sujos (dirty checking) – Sofreu alteração;
- Associações recuperadas sob demanda (lazy/eagerz association fetching);
- Outras funções de otimização.

Problemas trabalhados por ORM

- Como definir as classes persistentes e os meta-dados
- Como mapear hierarquias de classes
- Relacionamento da identidade de objeto e de linhas em tabelas
- O ciclo de vida de um objeto persistente
- Recuperar dados de associações de forma eficiente
- Gerenciar transações, cache e concorrência?

Persistência dos Objetos

A persistência de objetos por meio da técnica de **MOR** permite que se contorne a impedância existente entre as metodologias **OO** e **Relacional**, aproveitando assim os benefícios de ambas metodologias.

Persistência transparente refere-se à habilidade de se manipular os dados armazenados em um **SGBDR** diretamente pela linguagem orientada a objeto, e é o objetivo do **MOR**.

Isso retoma aos seguintes aspectos desejáveis em um sistema de persistência de objetos:

Persistência Ortogonal:

Válida para todos os objetos do sistema, independente de seu tipo.

Persistência Transitiva:

Se um objeto é persistente, todos os objetos referenciados por esse objeto devem ser promovidos a objetos persistentes.

Camada de Persistência

Objetivo: isolar os objetos de mudanças no mecanismo de armazenamento.

Se um SGBD diferente tiver que ser utilizado pelo sistema, por exemplo, somente a camada de persistência é modificada; os objetos de domínio permanecem intactos.

Diminuição do acoplamento entre os objetos e a estrutura do BD torna:

Mais flexível (pode ser modificado para se adaptar a novos requisitos)

Portátil (pode ser transportado para outras plataformas de HW ou SW)

Desvantagens:

Intermediação traz uma sobrecarga de processamento ao sistema, o que pode diminuir o seu desempenho.

Pode aumentar a complexidade da realização de certas operações que seriam triviais com o uso direto de SQL.

Entretanto, as vantagens adquiridas pela utilização de uma camada de software, principalmente em sistemas complexos, geralmente compensam as desvantagens.

Vantagens e Desvantagens de ORM

- **Vantagens**:
  - Produtividade:
    - Elimina a maior parte do código de infra-estrutura.
  - Manutenibilidade:
    - Menos linhas de código, menos manutenção;
    - Alterações nos dados não são tão impactantes.
  - Desempenho:
    - Mais tempo para implementar otimizações;
    - Maior conhecimento dos detalhes dos SGBDRs.
  - Independência de fornecedor:
    - Uso de dialetos de SQL.
- **Desvantagens**
  - Não são fáceis de aprender;
  - Para seu bom uso, é preciso dominar SQL e a tecnologia de bancos de dados relacionais;
  - Problemas decorrentes do seu uso são bastante complexos e difíceis de solucionar;
  - Não são a “bala de prata” da persistência!

Técnicas de ORM

A principal tarefa do **MOR** envolve a identificação das construções da orientação a objetos que se deseja extrair do esquema relacional, entre elas a identificação das classes e dos relacionamentos.

As principais técnicas de mapeamento de objetos em **SGBDR** podem ser descritas como:

1.Mapeamento Classe – Tabela

- Mapeamento de uma classe em uma ou mais tabelas, ou de uma tabela para uma ou mais classes, e mapeamento de herança.

2. Mapeamento Atributo – Coluna

- Mapeamento de tipos em atributos.

3. Mapeamento Relacionamento – Chave estrangeira

- Mapeamento dos relacionamentos OO em relacionamentos entre tabelas.

Estratégias do Mapeamento Classe-Tabelas

Mapeamento de Subset

- Onde os atributos da classe persistente representam algumas ou todas colunas de uma tabela.
- Esta estratégia convém para casos onde todos os atributos de uma classe persistente são mapeados a uma mesma tabela, e onde não há preocupação de incluir as colunas que não fazem parte do modelo de negócios. Pode referir-se também à herança de tabelas simples.

- **Mapeamento de Superset**
  - Onde os atributos da classe persistente são derivados de colunas de múltiplas tabelas.
  - Este tipo de mapeamento é usado para criar “classes de visão”, que ocultam o modelo físico de dados, ou para mapear uma árvore de herança de classes utilizando o mapeamento vertical.

Atributos de uma classe

Atributos Primitivos

- Atributo de uma classe que é mapeado a uma coluna de uma tabela. Valor de um tipo de dados específico (int, float, double, dentre outros).

Atributos de Referência

- Atributos que representam relacionamentos com outras classes. Atributos cujo tipo é uma referência a outro objeto ou conjunto de objetos (composição).
- As relações entre objetos são implementadas explicitamente através de atributos de referência, enquanto as relações entre tabelas são realizadas através de associações e chaves estrangeiras.
- O mapeamento de associações preocupa-se, basicamente, com algumas categorias de relacionamentos entre objetos.
- Uma baseia-se na multiplicidade, outra baseia-se em dois tipos de direcionalidade (Relacionamentos Unidirecionais e Relacionamentos Bidirecionais).
- Outra forma de relacionamento entre objetos são os Relacionamentos Recursivos (por exemplo, um time pode ser integrante de outros times).
- Um aspecto essencial da POO é a herança. A herança permite que dados e comportamentos de uma superclasse sejam reaproveitados por subclasses. Bancos de dados relacionais não possuem o conceito de herança.
- Entidades não podem herdar atributos de outras entidades.

Herança

Mapeamento Distribuído de Herança (Horizontal e Vertical)

- Forma mais difícil de ser implementada quando coleções heterogêneas de objetos precisam ser recuperadas. Cada subclasse é mapeada em uma tabela separada, e todos os atributos herdados são replicados na tabela.

Mapeamento de Filtro de Herança

- Forma mais rápida de mapeamento. Custo de manutenção e espaço. As classes são representadas em uma única tabela. Cada registro da tabela utiliza atributos pertinentes à sua subclasse, enquanto os outros atributos são mantidos nulos.

Mapeamento de Herança

Segundo Scott Ambler, existem fundamentalmente três estratégias para mapear herança em um banco de dados relacional

- **Uma tabela por hierarquia**: Mapear toda a hierarquia de classes para uma tabela, onde todos os atributos das classes da hierarquia são armazenados nesta única tabela. A desvantagem desta estratégia é que toda vez que um objeto da hierarquia for persistido no banco, é necessário persistir também os valores das demais classes vazios, causando uma grande quantidade de campos inutilizados. Entretanto o acesso ao banco para a manipulação dos dados é mais rápido, uma vez que todos os dados estão em somente uma tabela. É adicionada uma coluna (Object Type) na tabela que referência qual o tipo do objeto, ou seja, de qual classe aqueles dados pertencem;

- **Uma tabela por classe concreta**: Cada classe concreta mapeada reflete uma tabela com todos os atributos herdados das super classes abstratas. A vantagem desta estratégia é a facilidade de manipulação de dados, uma vez que todos os dados de cada classe estão em apenas uma única tabela. Como desvantagem, destaca-se que quando se modifica uma classe abstrata, é necessário modificar todas as tabelas geradas pelas classes filhas no modelo relacional;

- **Uma tabela por classe**: Cada hierárquica mapeada reflete uma tabela, relacionadas através do mecanismo de especialização padrão do banco de dados relacional (utilização de chaves estrangeiras). Segunda esta modalidade de mapeamento, tenta-se ao máximo manter a normalização de dados, de forma que a estrutura final das tabelas fica bastante parecida com a hierarquia das classes representada na UML. Esta é a técnica que mais naturalmente mapeia objetos para banco de dados relacionais.

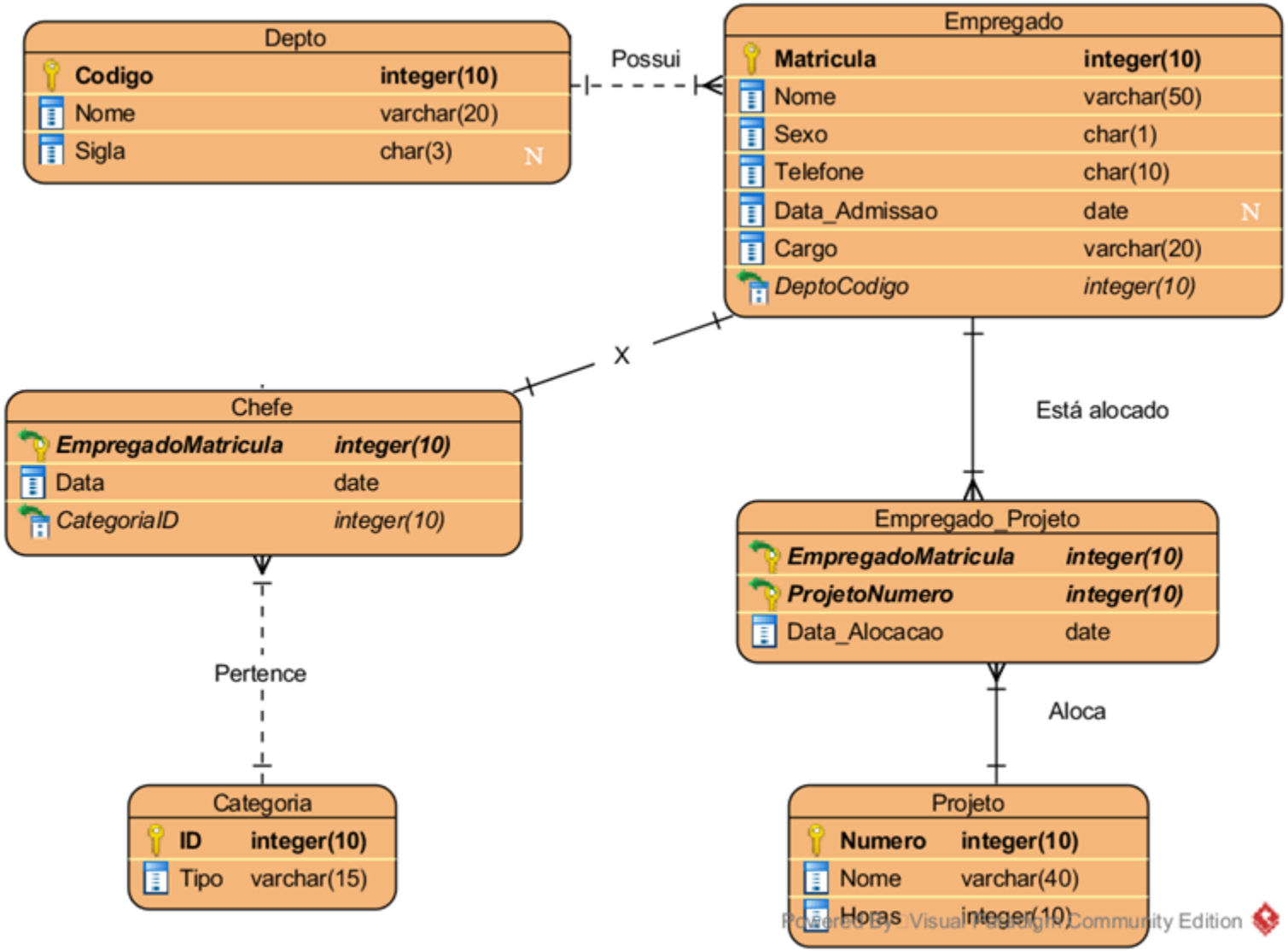
Spring Data

O Spring Data é o modelo de programação dentro do Spring Framework para acesso e manipulação de dados.

Chegando com a intenção de facilitar a configuração e utilização com o seu banco de dados, seja ele relacional ou não, o Spring Data traz vários recursos bacanas para acelerar o nosso desenvolvimento. Por exemplo, a configuração padronizada, onde devemos colocar apenas algumas propriedades e ele já vai saber o que fazer.

Outro recurso possível é a criação de query pela assinatura do método, na prática sendo muito simples de utilizar.

Exemplo


Depto
Codigo integer(10)
Nome varchar(20)
Sigla char(3)
N
Possui
Empregado
Matricula integer(10)
Nome varchar(50)
Sexo char(1)
Telefone char(10)
Data_Admissao date
N
Cargo varchar(20)
DeptoCodigo integer(10)
X
Está alocado
Chefe
EmpregadoMatricula integer(10)
Data date
CategoriaID integer(10)
Pertence
Empregado_Projeto
EmpregadoMatricula integer(10)
ProjetoNumero integer(10)
Data_Alocacao date
Aloca
Categoria
ID integer(10)
Tipo varchar(15)
Projeto
Numero integer(10)
Nome varchar(40)
Horas integer(10)
Powered By Visual Paradigm Community Edition

Application.properties

Toda a configuração de conexão com o SGBD é feito por esse arquivo

Exemplo em MS SQL Server

```
##Spring data properties
spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.datasource.url=jdbc:sqlserver://127.0.0.1:1433;databaseName=aulaspringdata;encrypt=true;trustServerCertificate=true
spring.datasource.username=leandro
spring.datasource.password=12345678
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql = true
spring.session.jdbc.initialize-schema: always


## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.SQLServerDialect


# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.properties.hibernate.proc.param_null_passing = true
```

Annotations

```
...
@Entity
@Table(name = "depto")
public class Depto {


@Id
@Column(name = "codigo", nullable = false)
private int codigo;
@Column(name = "nome", length = 20, nullable = false)
private String nome;
@Column(name = "sigla", length = 3, nullable = false)
private String sigla;
}

...
```

Chave Estrangeira

- OneToOne – Relacionamento 1 para 1
- ManyToOne – Relacionamento 1 para N com referência na entidade N
- OneToMany – Relacionamento 1 para N com referência na entidade N (Menos natural, para utilizar em Listas)

```
public class Empregado {
  ... colunas

@ManyToOne(targetEntity = Depto.class, fetch = FetchType.LAZY)
@JoinColumn(name = "deptoCodigo", nullable = false)
private Depto depto;

}
```

LAZY: Este é o FetchType padrão no Hibernate. Isso significa que a entidade associada será buscada somente quando for acessada pela primeira vez. Isto pode melhorar o desempenho nos casos em que a entidade associada não é necessária na maioria das vezes.
Isso pode ser mais eficiente do que buscar a entidade antecipadamente, especialmente se a entidade tiver muitos dados e não for necessária para todos os usos da entidade pai.

EAGER: Este FetchType significa que a entidade associada será buscada junto com a entidade principal quando a entidade principal for buscada no banco de dados. Isto pode ser útil nos casos em que a entidade associada é sempre necessária, mas também pode resultar numa diminuição de desempenho se a entidade associada for grande e/ou tiver muitas associações.
A opção FetchType.EAGER indica que a entidade associada deve ser buscada antecipadamente, o que significa que ela será buscada ao mesmo tempo que a entidade pai.
Usar FetchType.EAGER pode ser mais eficiente do que usar FetchType.LAZY se a entidade associada for necessária para a maioria dos usos da entidade pai, pois evita a necessidade de consultas adicionais ao banco de dados para buscar a entidade associada quando ela for acessada. No entanto, também pode ser menos eficiente se a entidade associada tiver muitos dados e não for necessária para todos os usos da entidade pai, pois sempre será buscada junto com a entidade pai. É uma boa ideia criar um perfil de seu aplicativo para determinar a melhor estratégia de busca para seu caso de uso.

Chave Primaria Composta

Usando @idClass – Uma classe define a PK Composta

```
@Getter
@Setter
@ToString
public class EmpregadoProjetoId implements Serializable {


private static final long serialVersionUID = 1L;
private Empregado empregado;
private Projeto projeto;

}
```

```
@Entity
@Table(name = "empregado_projeto")
@IdClass(EmpregadoProjetoId.class)
public class EmpregadoProjeto {


@Id
@ManyToOne(cascade = CascadeType.ALL, targetEntity = Empregado.class,
            fetch = FetchType.LAZY)
@JoinColumn(name = "empregadoMatricula", nullable = false)
private Empregado empregado;


@Id
@ManyToOne(cascade = CascadeType.ALL, targetEntity = Projeto.class,
            fetch = FetchType.LAZY)
@JoinColumn(name = "projetoNumero", nullable = false)
private Projeto projeto;

@Column(name = "data_alocacao", nullable = false)
private LocalDate dataAlocacao;

}
```

@Transational

A anotação @Transactional trabalha dentro do escopo de uma transação no banco de dados, a transação do banco de dados ocorre dentro do PersistenceContext, que por sua vez, está dentro do EntityManager que é implementado usando Hibernate Session (quando você está usando o Hibernate como container).

O PersistenceContext funciona como um container que armazena os objetos em memória até que eles sejam sincronizados com o banco de dados.

@Transational

```
@Controller
public class ChefeController {
            ... Injeções de Dependência
            @Transactional
            public void opChefe() {
                        Categoria cat = new Categoria(6, "Prog. Sênior");
                        cRep.save(cat);
                        Depto dep = new Depto();
                        dep.setCodigo(1); // Já inserido

                        Chefe ch = new Chefe();
                        ch.setMatricula(12346);
                        ch.setNome("Beltrano de Tal");
                        ch.setSexo("M");
                        ch.setCargo("Chefe");
                        ch.setTelefone("1122009988");
                        ch.setData(LocalDate.of(2022, 4, 18));
                        ch.setDataAdmissao(LocalDate.of(2021, 1, 10));
                        ch.setCategoria(cat);
                        ch.setDepto(dep);
                        chRep.save(ch);
                        List<Chefe> chefes = chRep.findAll();
                        chefes.forEach(chefe -> System.out.println(chefe));

            }
```

Herança

Inheritance Type JOINED – Relação de JOIN entre superconjunto e subconjunto

```
@Entity
@Table(name = "empregado")
@Inheritance(strategy = InheritanceType.JOINED)
public class Empregado {

@Id
@Column(name = "matricula", nullable = false)

... colunas
}
```

```
@Entity
@Table(name = "chefe")
public class Chefe extends Empregado {


@Column(name = "data", nullable = false)
private LocalDate data;


@ManyToOne(targetEntity = Categoria.class, fetch = FetchType.LAZY)
@JoinColumn(name = "categoriaID", nullable = false)
private Categoria categoria;
}
```

Não tem o @Id

NamedNativeQuery e NamedStoredProcedureQuery

- Queries especiais

```
...
@NamedNativeQuery(
                              name = "Depto.fn_depto_sigla",
                              query = "SELECT codigo, nome, sigla FROM fn_depto_sigla(?1)",
                              resultClass = Depto.class
)
@NamedStoredProcedureQuery(
                              name = "Depto.sp_depto_sigla",
                              procedureName = "sp_depto_sigla",
                              parameters = {
                                            @StoredProcedureParameter(mode =
                                            ParameterMode.IN, name = "codigo", type = Integer.class),
                                            @StoredProcedureParameter(mode =
                                            ParameterMode.OUT, name = "saida", type = String.class)
                              }
)
public class Depto {

                ...colunas
}
```

```
public interface IDeptoRepositoy extends JpaRepository<Depto, Integer> {
                Depto fn_depto_sigla(int codigo);
                @Procedure(name = "Depto.sp_depto_sigla")
                String sp_depto_sigla(@Param("codigo")int codigo);
}
```

- Caso use Stored Procedure, recomenda-se a Annotation @Transactional para evitar Exceptions