Hibernate/JPA

Instituto de Matemática e Estatística
Universidade de São Paulo

14 de abril de 2009

Conteúdo

Introdução

- Grande maioria das aplicações utiliza bancos de dados relacionais para armazenar seus dados
- Dados são modelados de maneira diferente em sistemas orientados a objetos e em bancos de dados relacionais
- JDBC e JDO eram "fracos"

Introdução

- Criação de frameworks que lidavam com a diferença da modelagem de dados
- Exemplos: Hibernate, TopLink

JPA

- Especificação de framework que cria o mapeamento objeto-relacional
- Solução mais simples e eficiente que as existentes até o momento
- Divulgado no Java EE 5 como parte do EJB 3

JPA

- Frameworks que motivaram a criação da JPA, adaptaram-se para implementar esta especificação

Annotations básicas

```
@Entity
public class Funcionario implements Serializable {
        @Id
        private int id;
        private String nome;
        @Temporal(TemporalType.DATE)
        private Date nascimento;
        // Construtor default, getter e setters
}
```

Annotations básicas

```
@Entity
public class Funcionario implements Serializable {
        private int id;
        private String nome;
        private String nascimento;
        @Id
        public int getId() {
          return id;
        }
        @Temporal(TemporalType.DATE)
        public Date getNascimento() {
          return nascimento;
        }
        // Outros getters e setters
}
```

@Id - Define qual campo que é a chave primária.

```
@Id
private int id;
```

@GeneratedValue - Define como que o JPA gera os valores para a chave primária. Pode ser de quatro tipos: AUTO (o default), IDENTITY, SEQUENCE e TABLE.

```
@Id
@GeneratedValue
private int id;
```

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int id;
```

strategy=GenerationType.AUTO - O JPA tenta descobrir qual é a melhor estratégia.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

strategy=GenerationType.IDENTITY - O banco de dados usa uma coluna de autoincremento.

```
@Id
@SequenceGenerator(name="Funcionario_Seq",
                   sequenceName="Func_Gen")
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="Funcionario_Seq")
private int id;
```

strategy=GenerationType.SEQUENCE - Uma sequence é usada para gerar a chave primária. Essa sequence é descrita em uma annotation @SequenceGenerator.

```
@TableGenerator(
    name="Funcionario_Seq",
    table="COntadores",
    pkColumnName="Nome",
    valueColumnName="Proximo",
    pkColumnValue="Funcionario"
)
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
                    generator="Funcionario_Seq")
private int id;
```

strategy=GenerationType.TABLE - Uma outra tabela é usada para gerar a chave primária. Essa tabela é especificada em uma annotation @TableGenerator.

Para tipos de data, hora e data/hora:

- java.sql.Date mapeia para "data".
- java.sql.Timestamp mapeia para "data/hora".
- java.sqlTime mapeia para "hora".
- java.util.Date e java.util.Calendar mapeiam de acordo com a annotation @Temporal.

```
@Temporal(TemporalType.DATE)
private Date dataNascimento;
```

Campos ou getters de tipos enum devem ter a anotação @Enumerated. @Enumerated tem dois valores: ORDINAL, que é o padráo e diz que o método ordinal() do enum será usado para persistí-lo, ou STRING que diz que o método name() será usado para persistí-lo.

```
@Enumerated(EnumType.STRING)
private Estado estado;
```

```
@Lob
@Basic(optional=true, fetch=FetchType.LAZY)
private char[] fotografia;
```

FetchType.EAGER - Indica que o campo sempre é carregado para a memória junto com a entidade. FetchType.LAZY - Indica que o campo só é carregado para a memória quando esse é referenciado.

```
@Entity(name="truck")
@Table(name="tb_caminhoes")
public class Caminhao implements Serializable {
```

```
@Entity
public class Carro implements Serializable {
@Id
@Column(name="carro_placa")
private String placa;

@Column(unique=true, nullable=true)
private String renavam;
```

E como fazer os relacionamentos entre classes?

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

```
@Entity
public class Empresa implements Serializable {
@OneToMany
private Collection<Funcionario> funcionarios;

@ManyToMany
private Collection<Cliente> clientes;
... blablabla ...
}
```

- cascade
- fetch
- targetEntity

@JoinColumn ao resgate!

```
@Entity
public class Veiculo implements Serializable {
    @ManyToOne
    @JoinColumn(name="cod_pessoa")
    private Pessoa proprietario;

    ... blablabla ...
}
```

Relacionamentos N-N

Quando @ManyToMany é usado, o JPA tenta deduzir o nome da tabela intermediária e dos respectivos campos.

```
@Entity
public class Pessoa implements Serializable {
  @Id
  private long cpf;

  @ManyToMany
  @JoinTable(name="visitas",
      joinColumns={@JoinColumn(name="cpf_visitante")},
      inverseJoinColumns={@JoinColumn(name="cod_museu")}
  )
  private Collection<Museu> museusVisitados;
```

Entity Manager

Funcionalmente, divide-se em:

- Transaction Association
- Entity Lifecycle Management
- Entity Identity Management
- Cache Management
- Query Factory
- Closing

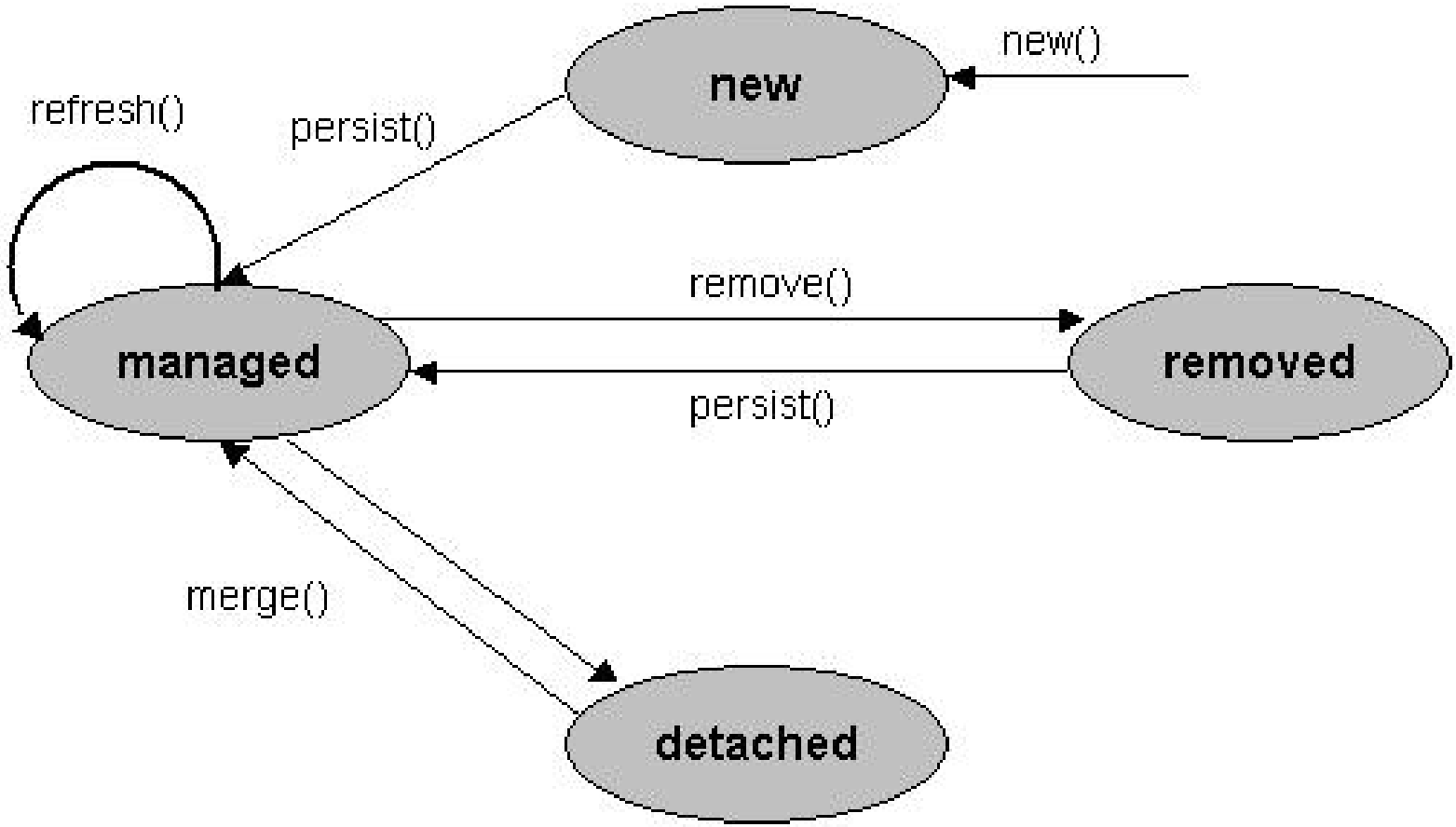
Transaction Association

```
public EntityTransaction getTransaction();
```

Entity Lifecycle Management

```
public void persist(Object entity);

public void remove(Object entity);

public void refresh(Object entity);

public Object merge(Object entity);

public void lock (Object entity, LockModeType mode);
- READ
- WRITE
```

Entity Lifecycle Management



Entity Identity Management

```
public <T> T find(Class<T> cls, Object oid);

public <T> T getReference(Class<T> cls, Object oid);

public boolean contains(Object entity);
```

Cache Management

```
public void flush();

public FlushModeType getFlushMode();

public void setFlushMode(FlushModeType flushMode);
- COMMIT
- AUTO

public void clear();
```

Query Factory

```
public Query createQuery(String query);

public Query createNamedQuery(String name);

public Query createNativeQuery(String sql);
public Query createNativeQuery(String sql, Class resultCls)
public Query createNativeQuery(String sql, String resultMap
```

Closing

```
public boolean isOpen();
public void close();
```

JPQL

- Similar a SQL
- Usado para definir buscas de entidades
- Independente do mecanismo usado para persistência
- Estende EJB QL
- Declaradas estaticamente ou dinamicamente

Comandos

Os comandos podem ser:

Seleção

```
SELECT campos FROM entidade [WHERE
condição] [GROUPBY clausula] [HAVING
condição] [ORDERBY clausula]
```

Mudança de valor

```
UPDATE entidade SET campo_atualizado =
valor [WHERE condição]
```

Remoção

```
DELETE entidade [WHERE condição]
```

Comandos

Os comandos podem ser:

Seleção

```
SELECT campos FROM entidade [WHERE
condição] [GROUPBY clausula] [HAVING
condição] [ORDERBY clausula]
```

Mudança de valor

```
UPDATE entidade SET campo_atualizado =
valor [WHERE condição]
```

Remoção

```
DELETE entidade [WHERE condição]
```

Comandos

Os comandos podem ser:

Seleção

```
SELECT campos FROM entidade [WHERE
condição] [GROUPBY clausula] [HAVING
condição] [ORDERBY clausula]
```

Mudança de valor

```
UPDATE entidade SET campo_atualizado =
valor [WHERE condição]
```

Remoção

```
DELETE entidade [WHERE condição]
```

Comandos

Os comandos podem ser:

Seleção

```
SELECT campos FROM entidade [WHERE
condição] [GROUPBY clausula] [HAVING
condição] [ORDERBY clausula]
```

Mudança de valor

```
UPDATE entidade SET campo_atualizado =
valor [WHERE condição]
```

Remoção

```
DELETE entidade [WHERE condição]
```

Para os exemplos a seguir

```
@Entity
    @Table(name="CUSTOMER_TABLE")
    public class Customer implements Serializable{
        public enum CustomerStatus {FULL_TIME, PART_
        @Id
        @Column(name="ID")
            private Integer customerId;
        @Column(name="CITY")
            private String city;
        @Column(name="NAME")
            private String name;
        @Enumerated(ORDINAL)
            @Column(name="STATUS")
            private CustomerStatus status;
        @OneToMany(mappedBy="customer")
            private Collection<Order> orders;
```

```
@Entity
    @Table(name="ORDER_TABLE")
    public class Order implements Serializable {
        @Id
        @Column(name="ID")
            private Integer orderId;
        @Column(name="QUANTITY")
            private int quantity;
        @Column(name="TOTALPRICE")
            private float totalPrice;
        @ManyToOne()
            @JoinColumn(name="CUST_ID")
            private Customer customer;
    }
```

Definindo e Executando Queries
Criadas Dinamicamente

Definindo a query

```
String ejbql =
"SELECT c FROM Customer c WHERE c.name = 'Joao'";
```

Executando

```
Query query = em.createQuery(ejbql);
List result = query.getResultList();
```

Definindo e Executando Queries
Criadas Dinamicamente

Definindo a query

```
String ejbql =
"SELECT c FROM Customer c WHERE c.name = 'Joao'";
```

Executando

```
Query query = em.createQuery(ejbql);
List result = query.getResultList();
```

Definindo e Executando Queries

Criadas em Named Queries

Definindo a Named Query

```
@NamedQuery (
    name="findCustomerByName",
    query="SELECT c FROM Customer "
    + " c WHERE c.name = :name"
)
})
```

Executando

```
List customers =
  em.createNamedQuery("findCustomerByName")
  .setParameter("name", "Joao")
```

Definindo e Executando Queries

Criadas em Named Queries

Definindo a Named Query

```
@NamedQuery (
    name="findCustomerByName",
    query="SELECT c FROM Customer "
    + " c WHERE c.name = :name"
)
})
```

Executando

```
List customers =
   em.createNamedQuery("findCustomerByName")
  .setParameter("name", "Joao")
```

Exemplos
Select

SELECT SIMPLES

```
"SELECT c FROM Customer c WHERE c.name = 'Joao'
```

SELECT COM JOIN IMPLICITO

```
"SELECT o FROM Order o WHERE o.costumer.name = 'Joao'
```

SELECT COM COLLECTION

```
SELECT costumer
FROM Costumer costumer,
IN(costumer.orders) order
WHERE order.totalPrice > 10
```

Exemplos
Select

SELECT SIMPLES

```
"SELECT c FROM Customer c WHERE c.name = 'Joao'
```

SELECT COM JOIN IMPLICITO

```
"SELECT o FROM Order o WHERE o.costumer.name = 'Joao'
```

SELECT COM COLLECTION

```
SELECT costumer
FROM Costumer costumer,
IN(costumer.orders) order
WHERE order.totalPrice > 10
```

Exemplos
Select

SELECT SIMPLES

```
"SELECT c FROM Customer c WHERE c.name = 'Joao'
```

SELECT COM JOIN IMPLICITO

```
"SELECT o FROM Order o WHERE o.costumer.name = 'Joao'
```

SELECT COM COLLECTION

```
SELECT costumer
FROM Costumer costumer,
IN(costumer.orders) order
WHERE order.totalPrice > 10
```

Hibernate

- Um dos principais frameworks que originou JPA
- Criado a partir de iniciativa open source
- Hoje, é mantido pela JBoss, divisão da Red Hat

Utilizando a API do Hibernate

- E possivel utilizar a API exclusiva do Hibernate mesmo quando o utilizamos como *provider* para o JPA
- O método getDelegate() retorna uma instancia que implementa a interface *org.hibernate.Session*

Hibernate Query API

- **Criteria Query API**
  - Maneira mais simples de obter dados
  - Queries sao feitas programaticamente atraves de uma API Java
- Hibernate Query Language
  - Sintaxe e funcionalidades bastante parecidas com JPQL
  - http://javaplace.blogspot.com/2007/09/hibernate-query-language-hql-e-ejb.html
- SQL Nativo

Como utilizar a Criteria API

- Crie um objeto *org.hibernate.Criteria* atraves do *factory method* createCriteria(...) da *Session*
  - Passe a classe do objeto persistente ou o nome da entidade para o metodo *createCriteria(...)*
  - Chame o metodo *list()* do objeto *Criteria*

```
//Get all instances of Person class and its subclasses
Criteria crit = sess.createCriteria(Person.class);
List results = crit.list();
```

Paginacao

- Hibernate cuida da paginacao
  - Retornar um numero fixo de objetos
- Dois metodos da classe Criteria
  - setFirstResult() - seta a primeira linha do resultado
  - setMaxResult() - seta a quantidade de linhas a retornar

```
Criteria crit = sess.createCriteria(Person.class);
crit.setFirstResult(2);
crit.setMaxResults(50);
List results = crit.list();
```

Restrictions

- Utilizada para obter objetos que satisfacam determinadas condicoes
  - Exemplo: Objetos do tipo Pessoa cuja idade seja maior do que 20 anos
- Adicione restricoes para o objeto Criteria com o metodo add()
  - O metodo add() recebe um objeto *org.hibernate.criterion.Criterion* que representa uma unica restricao
- Podem existir varias restrictions para uma Criteria
  - O metodo add() devolve o proprio objeto Criteria (chaining)

Metodos da classe Restriction

- Restrictions.eq("name", "Shin")
- Restrictions.ne("name", "NoName")
- Restrictions.like("name", "Sa%")
- Restrictions.ilike("name", "sa%")
- Restrictions.isNull("name");
- Restrictions.gt("price",new Double(30.0))
- Restrictions.between("age", new Integer(2), new Integer(10))
- Restrictions.or(criterion1, criterion2)
- Restrictions.conjuntion()
- Restrictions.disjunction()

Exemplo de Restriction

```
// Resgata objetos do tipo Pessoa
// cujo nome segue um padrao e cuja
// idade e 10 ou null

List people = sess.createCriteria(Person.class)
 .add(Restrictions.like("name", "Shin%"))
 .add(Restrictions.or(
  Restrictions.eq("age", new Integer(10)),
  Restrictions.isNull("age")))
 .list();
```

Ordenando resultados

- E possivel ordenar resultados utilizando *org.hibernate.criterion.Order*

```
List cats = sess.createCriteria(Cat.class)
  .add( Restrictions.like("name", "F%")
  .addOrder( Order.asc("name") )
  .addOrder( Order.desc("age") )
  .setMaxResults(50)
  .list();
```

Associacoes

- E possivel especificar *constraints* em entidades relacionadas utilizando *createCriteria(...)*

```
 List cats = sess.createCriteria(Cat.class)
.add(Restrictions.like("name", "F%")
.createCriteria("kittens")
     .add(Restrictions.like("name", "F%"))
.list()
```

Projecoes e Aggregate Functions

- A classe *org.hibernate.criterion.Projections* e um *factory* para instancias de *Projection*
- Aplique uma projecao em uma query atraves do metodo *setProjection()* da classe *Criteria*

Projecoes e Aggregate Functions

- rowCount()
- avg(String propertyName)
- count(String propertyName)
- countDistinct(String propertyName)
- max(String propertyName)
- min(String propertyName)
- sum(String propertyName)

Exemplo

```
// Retorna uma lista com um array de Object
// como primeiro elemento. O array de Object contem
// todos os valores em ordem

Criteria crit = sess.createCriteria(Product.class);
ProjectionList projectList = Projections.projectionList(
projectList.add(Projections.avg("price"));
projectList.add(Projections.sum("price"));
crit.setProjection(projectList);
List results = crit.list();
```

Query by Example

- Prove um outro "estilo"de busca
- Como realizar uma QBE
  - Parcialmente popule uma instancia de um objeto
  - Deixe o Hibernate construir uma *Criteria* utilizando a instancia como exemplo
- A classe *org.hibernate.criterion.Example* implementa a interface *Criterion*
  - Voce pode usa-la como qualquer outra restricao
- Utilize *Example.create(...)* para criar uma restricao

Exemplo

```
// Recupera objetos do tipo Pessoa
// atraves de um exemplo

Criteria crit = sess.createCriteria(Person.class);
Person person = new Person();
person.setName("Shin");
Example exampleRestriction = Example.create(person);
crit.add(exampleRestriction);
List results = crit.list();
```

Conclusao

- Facil de usar, poderosa e elegante
- Especialmente util quando e necessario utilizar Dynamic Query Generation
- Por outro lado, queries estaticas externas possuem algumas vantagens
  - Queries externalizadas podem ser auditadas e otimizadas (se necessario) pelo DBA
  - Named queries em arquivos de mapeamento concentram as queries em apenas um lugar
  - Named queries sao faceis de guardar em cache (se necessario)

PERGUNTAS

(NÃO OUSEM!!!)

PERGUNTAS

(NÃO OUSEM!!!)