Objektově relační mapování (dokončení)

4. listopadu 2025

V minulém semináři jsme si představili základní principy objektově relačního mapování (ORM) a Java Persistence API (JPA). V tomto semináři úvod do problematiky dokončíme a představíme si možnosti dotazování a několik způsobů, jak řešit dědičnost.

1 Dotazování

Java Persistence API nabízí hned několik způsobů, jak přistupovat k datům. První dva způsoby jsme si ukázali již v minulém semináři.

1.1 Vyhledání podle klíče

Nejjednodušší způsob, jak získat entitu z databáze, je vyhledat ji pomocí metody

```
<T> T EntityManager.find(Class<T> entityClass, Object primaryKey)
```

a následně procházet graf objektů.

1.2 Ad-hoc dotazy

Druhý, co do vyjadřovací síly silnější, způsob pro přístup k entitám je pomocí ad-hoc dotazů a jazyka JPQL. V minulém semináři jsme si ukázali zakládní variantu ve tvaru.

```
Query q = em.createQuery("select item from Invoice item");
List<Invoice> invoices = q.getResultList();
```

Pokud v dotazu potřebujeme specifikovat nějakou hodnotu, podporuje dotazovácí jazyk i rozhraní JPA bezpečné vložení parametrů do dotazu.

```
Query q = em.createQuery("select cust from Customer cust where cust.itin = :itin");
q.setParameter("itin", value);
return q.getSingleResult();
```

Pozor, v žádném případě není dobrý nápad sestavovat dotaz pomocí spojování řetězců. Jinak hrozí, že do programu zaneseme bezpečnostní chybu, která je ekvivalentem *SQL injection attack*.

Všimněme si, že v poslední ukázce jsme použili metodu `Query.getSingleResult()`, která je určená pro vrácení právě jedné entity. Ať už v kódu potřebujeme metodu `getResultList` nebo `getSingleResult`, je možné použít i typově bezpečnější variantu, která umožňuje vyhnout se implicitním, případně explicitním, typovým konverzím:

```
TypedQuery<Invoice> q = em.createQuery("select item from Invoice item", Invoice.class);
List<Invoice> invoices = q.getResultList();
```

Vedle toho jazyk JPQL umožňuje zpracovat i složitější dotazy, kde výsledkem je obecná n-tice reprezentovaná jako pole hodnot typu `Object[]`. Praktický příklad by mohl vypadat následovně:

```
List<Object[]> res = em.createQuery("SELECT c.name, COUNT(inv.id)"
      + " FROM Customer c, Invoice inv"
      + " WHERE inv.customer = c "
      + " GROUP BY c.name").getResultList();
```

Součástí jazyka JPQL je podpora modifikace dat, což je vhodné v situacích, kdy potřebujeme hromadně aktualizovat data. Použití je velmi podobné jako v případě bežných dotazů, jen se pro provedení změn volá metoda `Query.executeUpdate()`, jak ukazují následující dva příklady.

```
Query q = em.createQuery("UPDATE StockItem it SET it.unitPrice = it.unitPrice * 1.1");
int updated = q.executeUpdate();
```

```
Query q = em.createQuery("DELETE FROM StockItem it WHERE it.itemName = :name");
q.setParameter("name", ...);
int updated = q.executeUpdate();
```

Podrobnosti k jazyku JPQL včetně příkladů můžete nalézt na
`https://javaee.github.io/tutorial/persistence-querylanguage.html`.

1.3 Pojmenované dotazy

Určitým ekvivalentem k připraveným dotazům (*prepared statements*) v rozhraní JDBC jsou v případě JPA *pojmenované dotazy*, které jsou svázány s konkrétní entitou pomocí anotací `@NamedQuery` a `@NamedQueries`.

Následující příklad ukazuje deklaraci entity, pro kterou existuje jeden pojmenovány dotaz.

```
@NamedQuery(
   name = "allInvoicesForCustomer",
   query = "select i from Invoice i where i.customer = :cust")
public class Invoice implements Serializable { /* ...  */ }
```

Tento dotaz je následně možné použít pomocí metody `EntityManager.createNamedQuery`, jak ukazuje následující kód.

```
Query q = em.createNamedQuery("allInvoicesForCustomer");
q.setParameter("cust", customer);
List<Invoice> invoices = q.getResultList();
```

Pokud bychom chtěli mít pro danou třídu více pojmenovaných dotazů, je nutné je spojit pomocí anotace `@NamedQueries`, například následovně:

```
@NamedQueries({
   @NamedQuery(
   name = "allInvoices",
   query = "select i from Invoice i"),
   @NamedQuery(
   name = "allInvoicesForCustomer",
   query = "select i from Invoice i where i.customer = :cust"),
   @NamedQuery(
   name = "allInvoicesForCustomerItin",
   query = "select i from Invoice i where i.customer.itin = :custItin")
})
@Entity
public class Invoice implements Serializable { /* ...  */ }
```

1.4 Criteria API

Dotazování v jazyce JPQL má některé nedostatky, zejména co se týká typové bezpečnosti. Dalším problematickým místem je dynamická tvorba dotazů, tj. pokud neznáme dotaz předem, musíme jej sestavit spojováním řetězců, což může vést k bezpečnostním problémům, viz již jednou zmíněný SQL injection attack.

Reakcí na tyto nedostatky byl vznik Criteria API, které umožňuje dynamicky a typově bezpečně sestavit dotaz.

Criteria API využívá tří základních kamenů:

1. meta-model – třídy popisující model databáze, slouží k tomu vygenerované třídy s příponou _, např. pro entitu `Foo` je vygenerována třída `Foo_`,
2. `CriteriaBuilder` – třída sestavující dotaz,
3. `CriteriaQuery` – třída představující dotaz, avšak ten je před samotným provedením nutné převést na dotaz EntityManageru.

Vytvoření dotazu vypadá následovně:

```
1 CriteriaBuilder cb = em.getCriteriaBuilder();
2 CriteriaQuery cq = cb.createQuery(Invoice.class);
3
```

```
4 Root<Invoice> invoice = cq.from(Invoice.class);
5 cq.select(invoice);
6
7 TypedQuery<Invoice> q = em.createQuery(cq);
8 List<Invoice> invoices = q.getResultList();
```

Nejdříve vytvoříme objekt třídy `CriteriaBuilder` a následně vytvoříme dotaz (řádky 1 a 2), kde určíme, že výsledkem mají byt hodnoty typu `Invoice`. Dále sestavíme dotaz podobně jako bychom sestavovali strom nějakého výrazu (řádky 4 a 5). V našem případě tento strom má jen dva uzly, kořen obsahující operaci `from` a operaci `select`. Na závěr `EntityManager` vytvoří standardní dotaz a získáme výsledek (řádky 7 a 8).

Pokud bychom chtěli pracovat s atributy, například provést restrikci, použijeme odkaz do meta-modelu.

```
Root<Invoice> invoice = cq.from(Invoice.class);
cq.where(cb.equal(invoice.get(Invoice_.customer), customer));
cq.select(invoice);
```

V tomto příkladu metoda `cb.equal` reprezentuje operaci porovnání a atribut `Invoice_.customer` odkazuje na atribut v meta-modelu.

V podobném duchu podporuje CriteriaAPI i aktualizaci dat. Následující příklad ukazuje, jak vynásobit hodnotu atributu `StockItem.unitPrice` dvěma.

```
CriteriaBuilder update = entityManager.getCriteriaBuilder();
CriteriaUpdate<StockItem> cq = update.createCriteriaUpdate(StockItem.class);
Root<StockItem> item = cq.from(StockItem.class);

cq.set(StockItem_.unitPrice,
   update.prod(item.get(StockItem_.unitPrice), new BigDecimal(2)));

Query q = entityManager.createQuery(cq);
q.executeUpdate();
```

Metoda `update.prod` odpovídá operaci násobení.

2 Dědičnost

Důležitým (i když ne nutným) rysem objektově orientovaného programování je dědičnost. Objektově relační mapovaní na to musí pochopitelně reagovat. Jelikož relační model dat postráda ekvivalent k dědičnosti, je to místo, kde dochází k rozporům mezi dobře navrženým modelem tříd a modelem databáze.

JPA nabízí tři způsoby, jak řešit dědičnost. Všechny jsou vyznačeny pomocí anotace `@Inheritance` u dané entity.

2.1 SINGLE_TABLE

První řešení, které si ukážeme, předpokládá, že existuje jedna tabulka pro celou hierarchii tříd, kde máme jeden sloupec vyčleněný pro rozlišení jednotlivých tříd. Samozřejmě může nastat situace, kdy nějaký atribut není ve třídě definován. V takovém případě se u něj v tabulce nastaví příznak `NULL`.

Příklad použití, kdy máme abstraktní entitu faktura, která má dvě podtřídy národní a mezinárodní faktura ukazuje následující kód.

```
/**
 * abstraktni trida reprezentujici fakturu
 */
@Entity
@Inheritance
@DiscriminatorColumn(name="INVOICE_TYPE")
@ForceDiscriminator
public abstract class Invoice {
    @Id
    private long id;
    /* ... */
}

/**
 * trida reprezentujici narodni fakturu
 */
@Entity
@DiscriminatorValue("NAT")
public class NationalInvoice extends Invoice {
    private String eetNo;
    /* ... */
}

/**
 * trida reprezentujici mezinarodni fakturu
 */
@Entity
@DiscriminatorValue("INT")
public class InternationalInvoice extends Invoice {
    private String iban;
    /* ... */
}
```

V abstraktní třídě pomocí anotace `@DiscriminatorColumn` definujeme sloupec, který určuje o jaký typ faktury se bude jednat. V potomcích pak pomocí anotací `@DiscriminatorValue` určujeme hodnoty, které

jednotlivé typy faktur rozliší.

Toto řešení je problematické ze dvou důvodů. (i) Předpokládá, že v tabulce existuje sloupec, který je schopen rozlišit jednotlivé třídy. To je problém v situaci, kdy programujeme proti existující databázi, kterou nemůžeme měnit. (ii) U jednotlivých atributů nelze nastavit `NOT NULL`.

2.2 JOINED

Druhou možností, jak řešit dědičnost, je vytvořit pro každou entitu tabulku s atributy, které jsou v ní definované. Potomoci a jejich tabulky musí obsahovat atribut, primární klíč, který je definovaný v kořenové třídě. Za těchto podmínek JPA může jednotlivé třídy/tabulky spojit pomocí operace spojení.

Toto řešení je relativně přirozené i z pohledu relačního modelu. Určitým problémem může být, že některé implementace JPA vyžadují použití sloupce pro rozlišení tříd.

Samotné použití je podobné jako v předchozí podkapitole, jen je změněna anotace na:

```
@Inheritance(strategy=InheritanceType.JOINED)
```

2.3 TABLE_PER_CLASS

Třetí řešení, které si ukážeme, předpokládá, že pro každou třídu existuje právě jedna tabulka, která obsahuje všechny atributy včetně atributů předka. U tohoto řešení je problematické, že komplikuje sestavení dotazu. Je nutné provádět vícenásobné dotazy nebo sjednocení pomocí `UNION`, problematické je i spojení a řazení záznamů. Výhodou je, že není vyžadován sloupec určující typ entity.

Použití tohoto přístupu ilustruje následující kód:

```
/**
 * abstraktni trida reprezentujici fakturu
 */
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Invoice {
   @Id
   private long id;
   /* ... */
}

/**
 * trida reprezentujici narodni fakturu
 */
@Entity
@Table(name = "INVOICE_NAT")
public class NationalInvoice extends Invoice {
    private String eetNo;
    /* ... */
```

```
}

/**
 * trida reprezentujici mezinarodni fakturu
 */
@Entity
@Table(name = "INVOICE_INT")
public class InternationalInvoice extends Invoice {
    private String iban;
    /* ... */
}
```

Určitou alternativou k dědičnosti typu TABLE_PER_CLASS je použití anotace `@MappedSuperclass`, kdy se takto označená třída nevysktytuje v modelu databáze, ale každý potomek má vlastní tabulku se všemi atributy.

Komentář závěrem

Autor tohoto semináře považuje ORM koncepčně za jeden velký omyl. Jednak ho k tomu vede skutečnost, že opravdu dobře propojit dva odlišné světy (OOP a RM) není prakticky možné a vzniká zde řada třecích ploch. Tento problém je v literatuře označován jako *object-relational impedance mismatch*. Dále ho k tomu vede osobní zkušenost z několika projektů, kdy sice na začátku projektu ORM umožnilo rychlejší vývoj, ale s postupem času vyžadovalo stále více péče a „magie" nutné k tomu, aby řešení bylo dostatečně rychlé a odpovídalo dobrému (požadovanému) návrhu databáze.

K problematice ORM je velmi vhodné přečíst článek, *The Vietnam of Computer Science*,[1] případně smířlivější *OrmHate* od Martina F:owlera.[2]

I když autor semináře nepovažuje ORM za dobré řešení, je na místě, aby tato problematika byla probrána v rámci tohoto předmětu, protože v řadě situací je toto řešení dostatečně dobré a používá se v celé řadě projektů a firem.

[1] `https://web.archive.org/web/20220823105749/http://blogs.tedneward.com/post/the-vietnam-of-computer-science/`
[2] `https://martinfowler.com/bliki/OrmHate.html`