Objektově relační mapování

3. listopadu 2020

Ve světe programování velkých softwarových celků historicky vykrystalizovaly dva de facto standardy: (i) objektově orientované programování pro tvorbu programu a (ii) relační databáze pro uložení dat. Objektově relační mapování se snaží tyto dva odlišné světy spojit a vytvořit pojící vrstvu tak, aby programátory zbavilo rutinních úkolů, které jsou spojeny s ukládáním a načítáním dat z relační databáze.

1 Úvod

V praxi se velice často setkáváme s tím, že je aplikace rozdělena do dvou (případně více) vrstev, kde jedna vrstva představuje aplikační logiku a je naprogramována v Javě a druhá vrstva se stará o uložení dat a využívá k tomu relační databázi. Toto lze řešit standardním způsobem s využitím JDBC a jazyka SQL, kdy si nadefinujeme jednotlivé operace, které nám budou objekty ukládat a načítat z databáze. Výhodou tohoto řešení je, že máme plnou kontrolu nad tím, jak jsou data ukládána a načítána. Nevýhodou je, že tento přístup znamená vytvořit velké množství kódu s prvky duplicity.

Nabízí se proto vytvořit vrstvu, která bude schopna objekty ukládat jako řádky tabulek v rel. databázi a načítat je zpět jako objekty. Tato činnost se označuje jako *objektově relační mapování* (ORM) a na platformě Java je realizována pomocí Java Persistence API (JPA). Implementací JPA existuje hned několik, nejznámější je asi Hibernate, a dalé se používá EclipseLink, OpenLink nebo DataNuclues.

2 Mapování

Z pohledu aplikační vrstvy (jazyka Java) jsou entity[1], se kterými aplikace pracuje, objekty uložené v paměti. Identita objektu[2] je dána jeho adresou v paměti a vztahy mezi jednotlivými objekty jsou vyjádřeny pomocí odkazů.

Z pohledu uložení dat v relační databázi jsou entity vyjádřeny jako n-tice (viz relace nad relačním schématem), resp. jako řádky tabulky. Identita jednotlivých entit je v relační databázi určena hodnotami v atributech klíče. Vazby mezi entitami jsou řešeny pomocí klíčů a cizích klíčů.

Aby se eliminovaly tyto rozdíly, je potřeba definovat způsob, jak jednotlivé objekty a řádky tabulek mezi sebou vzájemně mapovat.

Jednou z možností je použít POJO (Plain Old Java Object) a definovat v konfiguračním XML souboru

[1] např. faktura, adresa, dodavatel

[2] to, co objekt jednoznačně identifikuje

mapování mezi objekty dané třídy a databází. Druhou, pohodlnější možností, je použít AJO (Annotated Java Object) a definovat mapování pomocí anotací.

K určení vztahu třídy k tabulce v relačním databázi lze využít následující anotace:

- `@Entity` – označuje, že objekty budou spravovány JPA (ukládany a načítány z relační databáze);
- `@Table(name = "")` – určuje tabulku, ve které jsou objekty dané třídy;
- `@Id` – označuje, který atribut bude brán jako primární klíč;
- `@GeneratedValue` – označuje, že klíč je automaticky generovaná hodnota;
- `@Column` – umožňuje specifikovat vlastnosti atributu (resp. sloupce), jméno, velikost.

Důležité je, aby jednotlivé třídy, implementovaly metody `equals` a `hashCode` v souladu se zvoleným klíčem, jinak se dá očekávat nekonzistentní chování. V praxi často třídy implementují i rozhraní `Serializable`.

2.1 Příklad mapování

Použití ORM si budeme demonstrovat na jednoduché aplikaci pro evidenci faktur.

Začneme třídou reprezentující položky na skladě:

```
@Entity
@Table(name = "stock_items")
public class StockItem implements Serializable {

    @Id
    @Column(name = "item_name", length = 50)
    private String itemName;

    @Column(length = 10)
    private String unit;

    @Column(precision = 12, scale = 3)
    private BigDecimal unitPrice;

    // konstruktory
    // gettery a settery
    // hashCode & equals

}
```

Anotacemi třídy jsme určili, že se jedná o entitu, třídu spravovanou JPA, a určili i tabulku, kde budou jednotlivé objekty uloženy. Nadefinovali jsme atributy pro název položky, měrnou jednotku a cenu za měrnou jednotku. Anotace `@Id` určuje, že primárním klíčem bude atribut `itemName`. Anotace `@Column`

specifikují dodatečné vlasnosti sloupce na straně databáze, jako je jméno sloupce, jeho délka nebo přesnost uložených dat. Pokud se anotace `@Column` nepoužije, odvodí se jméno a vlasnosti sloupce z deklarace atributu. Vynechané části kódu naleznete v přiloženém souboru s projektem.

2.2 Vztahy mezi objekty

Předchozí příklad byl jednoduchý v tom, že jednotlivé atributy třídy byly atomické hodnoty. Běžně potřebujeme definovat vztahy mezi jednotlivými třídami, např. faktura–položky, faktura–odběratel.

Tyto vztahy lze opět vyjádřit pomocí anotací, kterými jsou:

- `@Embedded` a `Embeddable` – indikuje, že objekt je součástí řádku nějaké entity;
- `@OneToOne` – popisuje vztah 1:1;
- `@ManyToOne` – vztah N:1 (společně s anotací `@JoinColumn`, určuje cizí klíč);
- `@OneToMany` – vztah 1:N (`mappedBy` – atribut);
- `@ManyToMany` – vztah M:N.

2.3 Příklady vazeb mezi objekty

Uvažujme, že u každého odběratele chceme mít (i) jméno, (ii) daňové identifikační číslo a (iii) adresu, přičemž v databázi chceme mít tyto atributy uložené v jednom řádku, ale v Javě chceme s adresou pracovat jako se samostatným objektem.

Budeme postupovat podobně jako v předchozím příkladu, jen u atributu `address` použijeme anotaci `@Embedded`.

```
@Entity
@Table(name = "customers")
public class Customer implements Serializable {

    private String name;

    @Id
    private String itin;

    @Embedded
    private Address address;

    // konstruktory
    // gettery a settery
    // hashCode & equals
}
```

Třídu reprezentující adresu označíme anotací `@Embeddable` následovně.

```
@Embeddable
public class Address implements Serializable {

    private String street;
    private String city;
    private String zip;

    // konstruktory
    // gettery a settery
    // hashCode & equals
}
```

Vztahy typu 1:N a N:1 si ukážeme na reprezentaci faktur, kde každá faktura má (i) identifikátor, (ii) datum, (iii) odběratele a (iv) seznam položek. Každá položka faktury má svůj (i) identifikátor, (ii) fakturu, (iii) skladovou položku a (iv) množství.

Položku faktury je možné reprezentovat následující třídou.

```
@Entity
@Table(name ="invoice_items")
public class InvoiceItem implements Serializable {

    @Id
    @GeneratedValue
    private int id;

    @ManyToOne
    @JoinColumn(name = "invoice_id")
    private Invoice invoice;

    @ManyToOne
    @JoinColumn(name = "stock_item")
    private StockItem stockItem;

    private BigDecimal amount;
    // konstruktory
    // gettery a settery
    // hashCode & equals
}
```

Kromě atomických atributů `id` a `amount` zde máme odkazy na fakturu (`invoice`), kam položka patří, a odkaz na skladovou položku (`stockItem`). Protože více položek může být na jedné faktuře a skladová položka se může objevit na více fakturách, jedná se o vztah N:1, a je na místě použít anotaci `@ManyToOne`.

Současně s těmito anotacemi je použita i anotace `@JoinColumn`, která udává název sloupce s cizím klíčem. Tím nám vznikne potřebná vazba na straně databáze.

Třída reprezentující fakturu může vypadat následovně:

```
@Entity
public class Invoice implements Serializable {

    @Id
    @GeneratedValue
    private int id;

    @Column(name = "issue_date")
    @Temporal(TemporalType.DATE)
    private Date date;

    @ManyToOne(optional = false)
    private Customer customer;

    @OneToMany(mappedBy = "invoice", cascade = CascadeType.ALL)
    private List<InvoiceItem> items = new ArrayList<>();

    // konstruktory
    // gettery a settery
    // hashCode & equals
}
```

Z pohledu vztahu mezi třídami zde máme jednak odkaz na odběratele a seznam položek. Vzhledem k tomu, že odběratel může mít přiřazených více faktur, používame zde vazbu N:1. Vztah faktura–položka je typu 1:N, a proto použijeme anotaci `@OneToMany`. Součástí této anotace je i parametr `mappedBy`, který určuje atribut v položce faktury, který odpovídá dané faktuře. Vzniká tak oboustranná vazba.

V posledním příkladu jsem použili ještě další anotace a jejich parametry, které si zaslouží vysvětlení.

- `@Temporal` – udává, že se jedná o atribut pracující s časem,
- `optional = false` – indikuje, že daná hodnota je povinná,
- `cascade = CascadeType.ALL` – indikuje, že operace s daným objektem (např. uložení) se mají promítnout i do odkazovaných objektů, případně lze specifikovat, které operace se mají provést.

U jednotlivých anotací indikujicích vztahy mezi třídami lze uvést parametr `fetch` s hodnotami `FetchType.EAGER` nebo `FetchType.LAZY`, který udává, jestli se mají odkazované objekty načítat okamžitě (implicitní chování) nebo líně, tj. v momentě, kdy se k objektu přistupuje.

3 Manipulace s objekty

Objekty (entity) spravované JPA se mohou nacházet ve čtyřech stavech:

- *new* – nově vytvořený objekt, nemající protipól v databázi,
- *managed* – objekt má přesný protipól v databázi (vznikne voláním `persist`),
- *detached* – objekt není ve stavu, kdy by měl protipól v DB (v důsledku aktualizace objektu); do stavu *managed* se dostane volaním `merge`,
- *removed* – objektu byl odstraněn protipól v DB.

O práci s entitami se stará třída `EntityManager`[3], která poskytuje například metody:

- `void persist(Object entity)` pro uložení entity,
- `void merge(Object entity)` pro uložení změn entity,
- `void refresh(Object entity)` pro obnovení stavu entity,
- `void remove(Object entity)` pro odstranění entity,
- `<T> find(Class<T> entityClass, Object primaryKey)` pro nalezení objektu dané třídy pomocí primárního klíče,
- `Query createQuery(String query)` pro vyhledání objektů v jazyce JPQL.

Práce s objekty je přímočará. Chceme-li například uložit fakturu postupujeme následovně:

```
entityManager.getTransaction().begin();
entityManager.persist(invoice);
entityManager.getTransaction().commit();
```

Nejdříve zahájíme transakci. Následně provedeme operace s objekty, které potřebujeme. Na závěr transakci potvrdíme. Podobně postupujeme u odstraňovaní nebo aktualizace stavu objektů a dalších operací. Viz přiložený kód.

Pro získání objektů může buď použít metodu `find` nebo dotaz v jazyce JPQL, což je jazyk imitující SQL.

Například seznam všech faktur získáme následovně.

```
(List<Invoice>) entityManager.createQuery("select inv from Invoice inv").getResultList();
```

Dotaz čteme následovně:

1. `from Invoice` – chceme vybrat objekty třídy `Invoice`,
2. `from Invoice inv` v rámci dotazu se na faktury budeme odkazovat proměnnou `inv`,
3. `select inv` určuje, že chceme vrátit objekty v proměnné `inv`.

Jedná se o určitou obdobu `select * from invoice`. Dotazování se budeme podrobněji věnovat v následujícím semináři.

[3] přesněji řečeno třída implementující rozhraní `EntityManager`

4 Konfigurace systému

V předchozích částech jsme si ukázali, jak definovanovat mapování mezi třídami a tabulkami a jak provádět základní operace. Zbývá popsat, jak získat objekt `EntityManageru` a nakonfigurovat databázi.

JPA se typicky používá ve spojení s Java EE, ale není problém jej poouživat i s Java SE. Ukážeme si konfiguraci, kterou lze použít v rámci obou platforem.

K získání `EntityManegeru` potřebujeme nejdříve získat `EntityManagerFactory`, takže postupujeme následovně.

```
private final EntityManagerFactory emfactory =
    Persistence.createEntityManagerFactory("lecture04PU");
private final EntityManager entityManager = emfactory.createEntityManager();
```

Metoda `createEntityManagerFactory` má jako svůj argument název *persistence unit* (`lecture04PU`), což je jednotka zastřešující práci s danou množinou entit, typicky se vztahuje k jedné databázi.

Konfiguraci persistance unit najdete v souboru `persistance.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persiste
  <persistence-unit name="lecture04PU" transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>jdbc/jpa01</non-jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Tento soubor definuje název persistence unit a jako typ transakce uvádíme `RESOURCE_LOCAL`, což znamená, že za správu transakcí přebíráme zodpovědnost.[4]

Element `non-jta-data-source` definuje, ke které databázi se budeme připojovat. Ve vlasnostech persistence unit ještě uvádíme, že pokud neexistují odpovídající tabulky, mají se vytvořit.

4.1 Připojení k databázi

Posledním pomyslným dílem skládačky je připojení k databázi. Správu připojení obstarává aplikační server. Ten si udržuje otevřená spojení k databázovému serveru v tzv. *connection pool*u, ze kterého jsou jednotlivá spojení podle potřeby vybírána. Tím se optimalizuje přístup k databázi. Dále, aby byla oddělena

[4] Tento přístup se typicky používá v Java SE. V některém z dalších seminářů se k tomuto vrátíme a zodpovědnost budeme delegovat na aplikační server.

konfigurace od implementace, není dané spojení do databáze odkazováno přímo ale nepřímo pomocí tzv. zdrojů (*resources*).

Konfigurace spojení je očekávatelně realizována na úrovni aplikační serveru, tj. najdete ji na adrese `http://localhost:4848`.[5].

4.2 Vytvoření connection poolu

Začneme tím, že si vytvoříme databázi, ke které se můžeme připojit přes rozhraní JDBC, tj. spustíme databázový server, vytvoříme uživatele pro přístup k databázi a vytvoříme tomuto uživateli databázi. Pro jakou databázi se rozhodnete není úplně důležité.

V konfiguraci aplikačního serveru (`http://localhost:4848`) v části *Resources* → *JDBC* → *JDBC Connection Pools*, vytvoříme nový connection pool. Zvolíme vhodný název charakterizující danou databázi, jako typ zdroje použijeme `javax.sql.DataSource` a zvolíme odpovídající ovladač. V dalším kroku konfigurace nastavíme vlasnosti připojení, zejména jméno databáze, adresu, port, jméno a heslo uživatele, případně další parametry. Po vytvoření můžeme tlačítkem *Ping* ověřit, jestli je vše nastaveno správně.

Může se stát, že aplikační server nemá k dispozici ovladač pro databázi, se kterou chcete pracovat. Ten je možné doplnit následujícím příkazem.

```
asadmin add-library jdbc-ovladac.jar
```

4.3 Vytvoření databázového zdroje

Poslední krok je identifikovat přípojení k databázi jako zdroj, aby na něj mohlo být odkazováno.

K tomu slouží volba *Resources* → *JDBC* → *JDBC Resources*, kde vytvoříme nový zdroj a přiřadíme mu jméno, v našem případě `jdbc/jpa01`, které jsme použili v nastavení persistence unit, a zvolíme námi vytvořený connection pool. Bývá zvykem, že databázové zdroje začínají prefixem `jdbc/`.

5 Závěr

V tomto semináři jsme si ukázali zakladní použití objektově relačního mapovaní. V následujícím semináři se zaměříme na dotazování a další vlasnosti ORM.

[5]Popis odpovída AS GlassFish nebo Payara