OBJEKTOVĚ - RELAČNÍ MAPOVÁNÍ S VYUŽITÍM TECHNOLOGIE JDO

Vít Stinka

ČZU, Provozně ekonomická fakulta, Kamýcká, 160 00 Praha 6

Abstrakt

Tématem příspěvku je technologie JavaDataObjects a možnosti jejího využití při perzistentním mapováním objektů do relační databáze. Příspěvek na jednoduchém příkladě probírá rozdíly mezi standardním rozhraním JDBC a JDO.

1. Úvod – vzor MVC a role modelu

Jedním z klíčových návrhových vzorů, o který se v dnešní době při vývoji informačních systémů můžeme opřít, je paradigma *Model-View-Controller.* Jedná se o obecný pohled na strukturu aplikací, který nám říká: „aplikační komponenty spadají do třech hlavních, logicky oddělených skupin":

- *Model,* představující entity, se kterými aplikace pracuje a jejich (business) logiku.
- *View,* tvořící uživatelský pohled na model (nejčastěji grafické uživatelské rozhraní).
- *Controller*, řídící chování celé aplikace tím, že slučuje Model a View komponenty. Obsahuje tedy logiku na úrovni „volání" spouštěnou nejčastěji událostí, vyvolanou uživatelem.

Systém, který rozumně vyžívá přístupu MVC, dosahuje relativně vysokého stupně znovupoužitelnosti a flexibility (např. komponenty modelu a view mohou být ve vztahu 1:N) a navíc přirozeně konverguje ke stavu „high cohesion – low coupling", který je nutnou podmínkou k rozšiřitelnosti aplikací.

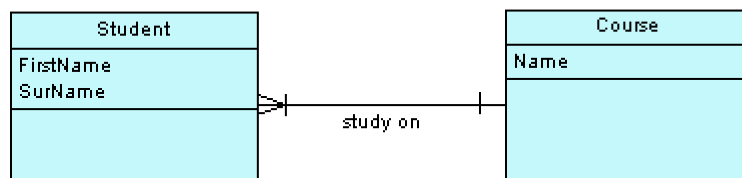
Vývoj podle vzoru MVC (zde uvažujeme Javovské prostředí) však není zcela triviální. Většinou k němu potřebujeme určitý základ (framework) – komponenty, které nám umožní „oddělit logiku od stánek" nebo alespoň část těchto komponent. Pokud bychom tyto komponenty neměli k dispozici, pravděpodobně bychom strávili jejich vývojem většinu času, který máme na projekt k dispozici.

Uvažujme nyní podrobněji o komponentách modelu: dojdeme k závěru, že v jejich kompetenci jsou dvě hlavní oblasti:

- *Aplikační logika* („business metody"). V nich je zanesena hlavní přidaná hodnota aplikace.
- *Data* nutná pro vykonání aplikační logiky. Se správou dat je však spojena nemalá režie - většinou jsou uložena v persistentním úložišti (např. relační databáze, XML, …) a komponenty modelu musí zajistit jejich *mapování* do tohoto úložiště. Otázkou (a tématem dalších odstavců) nyní je, kolik námahy je spojeno s tímto mapováním v MVC aplikaci a jak lze pro tuto úlohu využít technologii JDO.

2. Klasický způsob mapování a jeho nevýhody

Uvažujme nyní triviální aplikaci, která má 2 entity modelu *student* a *obor,* spojené ve vztahu N:1 (na oboru studuje více studentů). Tyto entity jsou přirozeně *perzistentní* (což *nemusí* být u komponent modelu pravidlem), lze si tedy představit, že jejich instance bude třeba mapovat do perzistentního úložiště (zde uvažujeme relační databázi).



Obr. 1 : Příklad entit modelu

Nyní se zamysleme nad možnostmi řešení úlohy „založení studenta „Jan Novák“ studujícího na oboru „Informatika“ (který je třeba rovněž založit).

První možností je využití standardního rozhraní JDBC. Kód by pak vypadal následovně (pro zjednodušení jsou identifikátory entit použity přímo):

```
Connection conn = ...;
Statement stmt = null;
String crSQL = "insert into course values (1,
\"Informatika\")";
String stuSQL = "insert into student values (2,  \"Jan\",
\"Novák\", 1)";
stmt = conn.createStatement();
stmt.executeUpdate(crSQL);
stmt.executeUpdate(stuSQL);
conn.commit();
```

Tento zápis, ač jinak velmi jednoduchý, zdaleka neodpovídá našim požadavkům na způsob persistentního ukládání (a následně načítání) entit. Je tomu tak proto, že JDBC je pouze prostředek na *spouštění SQL příkazů* z tříd Javy. Další aspekty s tímto spojené již neřeší. To vede ke stavu, kdy při ukládání nebo načítání perzistentních entit postupujeme „jinak“, než jsme v objektovém prostředí zvyklí.

Cílem je, abychom s persistentními entitami pracovali standardním způsobem. Entita bude např. uložena pokud jí zašleme zprávu „store” – a možnost zaslání této zprávy bude součástí rozhraní entity a nikoli, jako ve výše uvedeném příkladě, nějaké jiné „pomocné“ třídy. Důsledkem mimo jiné bude (viz níže), že se zcela odstíníme od zápisu destruktivních operací (insert/update/delete) ve formě SQL.

Při dosahování výše uvedeného cíle máme několik možností: můžeme vynaložit značné úsilí na vyvinutí vlastního frameworku na objektově relační mapování. Lepší možností však je využít již existující, pokud možno standardní řešení. Java (ale pouze v J2EE edici) nám základní podporu poskytuje – jsou jí techniky Container-Managed-Persistence a Bean-Managed-Persistence jako součást EJB. Jejich použití a konfigurace ale nepatří k nejjednodušším – a navíc v nich některé potřebné vlastnosti stále chybí. Proto se v poslední

době začaly objevovat další, modernější a z pohledu programátora jednodušší způsoby. Jedním z nich je i standard JDO.

3. Technologie JDO

3.1 Základní princip

Technologie JavaDataObjects je interface-standard definovaný firmou Sun jako framework pro *objektovou persistenci* (která zdaleka nezahrnuje jen mapování do relační databáze). Potřebná rozhraní a několik základních tříd jsou součástí volně dostupné package *javax.jdo.* JDO tedy není jedna konkrétní sada komponent – existuje řada komerčních a v poslední době i volně dostupných [3] implementací (tzv. *JDO vendor*s) od různých výrobců, které se od sebe mohou vzájemně (v některých případech dosti značně) odlišovat (odlišnosti mohou být způsobeny právě orientací na typ úložiště). Základní myšlenkou je *transparentnost* persistence, jejímž cílem je „maximálně neodlišovat" práci s persistentními a transientními entitami.

Způsob použití JDO lze v základním přehledu shrnout takto:

1. Na začátku existuje libovolná Java třída. U této třídy požadujeme schopnost persistence.
2. Po kompilaci třídy je na tuto aplikován speciální proces *enhacenmentu* - „rozšíření", na jehož konci třída začne implementovat interface *PersistentCapable,* který je součástí specifikace JDO. Modifikace třídy je provedena (pomocí ant-tasku) až na úrovni byte-kódu, proto původní zdrojový kód zůstane zcela nedotčen.
3. Aby mohl proběhnout bod 2, je třeba vytvořit pro každou třídu *persistence descriptor* (ve formátu XML), který obsahuje „konfiguraci" persistence, např. navázání atributů třídy na sloupec v databázi. Celý framework obsahuje dále několik dalších konfiguračních souborů.
4. Následně je možno definovaným způsobem transparentně mapovat (ukládat / načítat) třídu do persistentního úložiště.

3.2 Použití JDO

Výše uvedený příklad lze s využitím technologie JDO implementovat takto:

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory(…);
PersistenceManager pm = pmf.getPersistenceManager();

Transaction tr=pm.currentTransaction();
tr.begin();

Course course = new Course();
course.setName("Informatika");
pm.makePersistent(course);

Student student = new Student();
student.setFirstName("Jan");
student.setSurName("Novák");
student.setCourse(course);
pm.makePersistent(student);
```

```
tr.commit();
```

Persistence deskriptor pro třídu student může vypadat následovně:

```
<jdo>
 <package name="jdotest">
  <class name="Student" identity-type="datastore">
   <extension vendor-name="jdo" key="jdbc-table-name"
value="Student" />
   <field name="FirstName">
   <extension vendor-name="jdo" key="jdbc-column">
    <extension vendor-name="jdo" key="jdbc-column-name"
value="FirstName"/>
   </extension>
   </field>
   <field name="SurName">
   <extension vendor-name="jdo" key="jdbc-column">
    <extension vendor-name="jdo" key="jdbc-column-name"
value="SurName" />
   </extension>
   </field>
  </class>
 </package>
</jdo>
```

Course a student jsou běžné datové „set/get“ třídy, které byly rozšířeny frameworkem JDO pro persistentní použití. Persistence descriptor (jehož formát může být v rámci jednotlivých implementací odlišný) definuje, které třídy a atributy budou mapovány na konkrétní sloupce v relační databázi. Klíčovou třídou v JDO je PersistenceManager, který provádí základní operace s „PersistentCapable“ třídami (ukládání, mazání, vyhledávání podle ID, atd.).

Jednoduchým způsobem lze pracovat i s třídami v asociačním vztahu – systém dokáže přijmout jako argument „set“ metody jinou instanci JDO a standardním způsobem (v databázi přes cizí klíč) ji uložit. Z příkladu je vidět, že pro destruktivní operace a jednoduché vyhledávání nepotřebujeme vůbec používat jazyk SQL. Systém dále poskytuje sofistikovaný systém práce s transakcemi (optimistické / pesimistické transakce, atd.)

3.3 JDO a dotazování

Dotazování lze v JDO realizovat třemi způsoby:

- Pomocí metody *getObjectById()* na třídě PersistentManager.
- Pomocí objektu *Extent,* který poskytuje iterátor pro přístup ke všem uloženým instancím třídy.
- Pomocí vlastního jazyka *JDOQL*, který umožňuje extenty dále omezovat pomocí selekčních podmínek, deklarovat parametry, atd. Konstrukce lze zapisovat přímo na úrovni Javy.

V následujícím příkladě je uveden parametrizovaný dotaz hledající studenty podle příjmení:

```
Extent studentExtent = pm.getExtent(Student.class, true);
String filter =  "SurName == searchSurName";
Query q = pm.newQuery(studentExtent, filter);
q.setOrdering("SurName ascending");
Collection c = (Collection)q.execute("Novák");
```

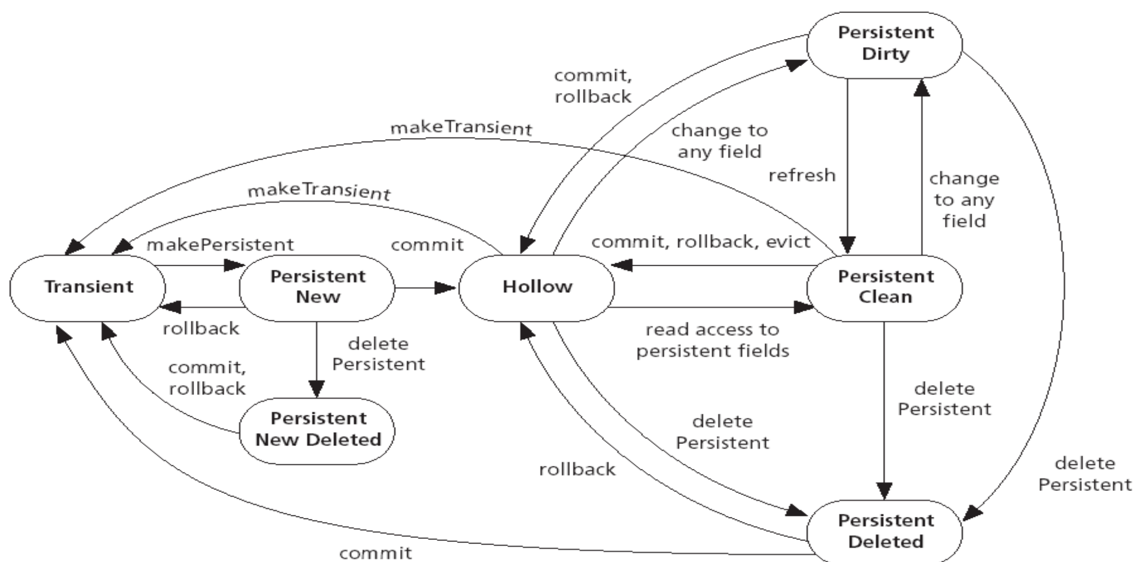
Konstrukce jazyka JDOQL obsahují podporu pro všechny základní typy dotazů (vč. podpory pro operace join, group-by, atd). Cílem je poskytnout jednoduchý způsob psaní dotazů bez znalosti SQL, který se navíc nezmění při změně typu úložiště (databáze → XML).

3.4 Životní cyklus JDO instancí

Každá instance JDO se dostává v průběhu svého životního cyklu do definovaných stavů.
Základní stavy instancí jsou:

- Transient - instance nemající persistentní reprezentaci (typicky instance po vytvoření).
- Hollow – instance mající persistentní reprezentaci, jejíž aktuální hodnoty však nejsou načtené do aplikační vrstvy.
- Persistent – instance s persistentní reprezentací – tento typ je dále rozdělen na další podtypy (např. po zavolání metody deletePersistent() se instance dostane do stavu PersistentDeleted, následně se (po potvrzení transakce) stane opět transientní).

Přechody mezi stavy JDO instancí jsou zobrazeny na následujícím schématu [1]:



Obr. 2 : Životní cyklus JDO instance

4. Závěr

Na uvedeném příkladě není přínos technologie JDO pravděpodobně možno zcela docenit – její možnosti jsou pochopitelně podstatně rozsáhlejší. [1] Navíc se i zde jistě najdou některé nedostatky (např. implementace dostupná autorovi, neposkytovala rozumný způsob mapování dědičnosti). Už dnes se však již najdou oblasti, kde by její využití stálo přinejmenším za zvážení.

Literatura:

1. ROOS, R. JavaDataObjects. Addison-Wesley, 2003, 264p., ISBN 0-321-12380-8.
2. http://www.java.sun.com
3. http://jakarta.apache.org