Java objektumok leképzése relációs adatbázisokra OJB-vel

Viczián István (viczus@freemail.hu)

Előszó

E cikk olyan haladó programozóknak nyújt segítséget, kik tisztában vannak a Java nyelvvel, és többször is használták a JDBC technológiát adatok relációs adatbázisban való tárolásához. Az ingyenes ObJectRelationalBridge (OJB) tulajdonképpen egy köztes réteg az alkalmazás és az adatbázis meghajtó között, mely kiküszöböli SQL parancsok használatát, az objektumok és táblák közötti kétirányú leképzést végzi. A cikk segítséget nyújt az első lépésekben, objektumok, táblák készítésében, a leképzés leírásában és az API megismerésében.

Bevezetés

Aki használt már relációs adatbázist Java programból, az tudja, hogy legegyszerűbben ez a JDBC technológiával oldható meg, de sajnos ebben az esetben elkerülhetetlenné válik, hogy a Java kód keveredjen az SQL utasításokkal, hiszen a metódusoknak `String` paraméterként kell megadni magát az SQL utasítást.

Ez a megoldás nem illik bele az objektum-orientált szemléletbe, két nyelvet kever, ami nem ajánlott, sem ebben az esetben, mikor Java és SQL keveredik, sem pl. webes alkalmazások fejlesztésekor, a Java és HTML esetén, sőt lehetőség van mindhárom keverésére. Ez csökkenti az átláthatóságot, és nehezebben is módosítható. Erre a problémára több megoldást is létezik, egyrészt az SQL parancsok külön választhatóak külön fájlba, a J2EE esetén ha CMP (Container Managed Persistence) Enterprise JavaBeans-eket használunk, akkor azok perzisztenciáját az un. EJB konténer kezeli. Fejlesztők rendelkezésére állnak már objektum-orientált adatbázisok is, melyek kiforrottsága még nem áll olyan magas szinten, mint a relációs adatbázisoknak. Egyszerűbb megoldás J2SE alkalmazások írása esetén egy objektum-relációs híd használata. Persze J2EE alkalmazások írásakor is használhatjuk ezt, ha BMP (Bean Managed Persistence) Enterprise JavaBeans-eket használjuk, hiszen akkor a bean-eknek saját maguknak kell a perzisztenciájukról gondoskodniuk.

Az ObJectRelationalBridge (OJB) az Apache Software Foundation támogatásával, az The Apache DB project (korábban Jakarta) keretein belül fejlesztett alprojekt. Alapvető célja Java objektumok leképzése relációs adatbázisokra. A piacon rengeteg ilyen köztes réteg található, mind az ingyenes, mind az üzleti szférában. Választásom azért esett mégis erre a szoftverre, mert az Apache név már bizonyított az ingyenes segédeszközök, keretrendszerek terén, illetve bíztam az általam használt többi Jakarta project keretein belül fejlesztett szoftverrel való együttműködésében (pl. Ant, Log4J).

Az OJB nem csak saját API-val rendelkezik (PersistenceBroker API), hanem erre épülnek rá magasabb szintű, szabványos interfészek is, úgymint az ODMG 3.0, JDO és illetve Object Transaction Manager (OTM), mely tartalmazza az előző kettő közös jellemzőit.

A perzisztencia teljesen transzparens, hiszen a tárolni kívánt osztályoknak nem kell speciális metódusokat implementálni, és nem kell egy kijelölt őstől származnia. Az objektumok és táblák, illetve mezők és oszlopok közti leképzést egy XML mapping állományban kell megadni.

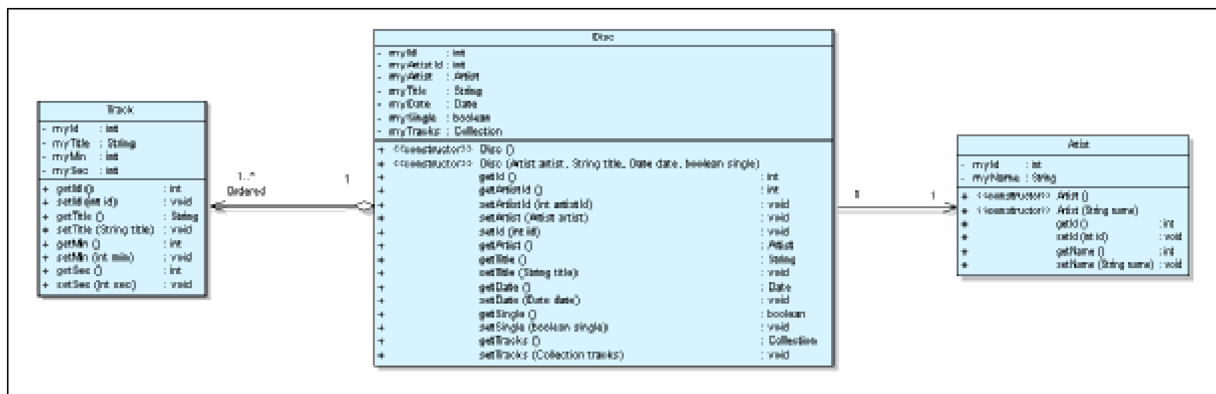
Példa

Az OBJ működését egy egyszerű példán fogom bemutatni, ami három osztályt és táblát tartalmaz. Szerepelnek benne cd lemezek, a hozzá tartozó előadó és az lemezen található számok.

Objektum – relációs adatbázis leképzés

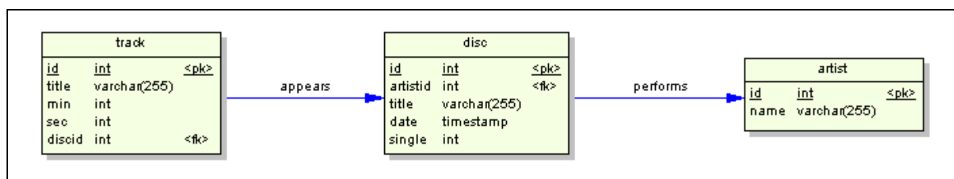
Vegyünk három egyszerű osztályt, `Artist`, `Disc`, `Track` néven, mely ábrázol egy előadót, egy lemezt és a lemezen lévő számokat.

Minden lemezhez legalább egy szám tartozik, és az egyszerűség kedvéért pontosan egy előadó. Azzal sem foglalkozunk, hogy egy előadó több lemezt is kiadhat. Az osztályok UML diagrammja a következő ábrán található.



1. ábra Osztályok UML diagrammja

Ehhez az osztályokhoz meglehetősen egyszerű táblák tartoznak, melyeket a következő ábra mutatja.



2. ábra Táblák

Az OJB-nek az egyedi azonosítók kezeléséhez és a zárolás kezeléséhez szüksége van két saját táblára is, melyeket neve `OJB_SEQ` és `OJB_HL_SEQ`.

```
CREATE TABLE OJB_SEQ
(
    TABLENAME VARCHAR (175) not null,
    FIELDNAME VARCHAR (70) not null,
    LAST_NUM integer,
    PRIMARY KEY(TABLENAME,FIELDNAME)
);

CREATE TABLE OJB_HL_SEQ
(
    TABLENAME VARCHAR (175) not null,
    FIELDNAME VARCHAR (70) not null,
    MAX_KEY integer,
    GRAB_SIZE integer,
    PRIMARY KEY(TABLENAME,FIELDNAME)
);
```

Fontos, hogy minden tárolni kívánt osztálynak adjunk meg paraméter nélküli konstruktort is, hiszen az OJB azzal példányosítja, majd a `setXYZ()` metódusokat használja a mezők

beállításakor. A `getXYZ()` metódusokat használja az objektumok tárolásakor, így minden tárolni kívánt mezőhöz deklarálnunk kell mindkét accessor metódust. Végig vigyázzunk a kis és nagybetűk közti különbségre, hiszen a Java nyelv megkülönbözteti azokat, így az OJB mapping állományban is ehhez kell igazodnunk. Ez természetesen nem vonatkozik a táblák és oszlopok neveire, hiszen az adatbázis-kezelők nem különböztetik meg a kis és nagybetűket.

A mapping állomány a `repository.xml`, mely hivatkozik két másik XML fájlra is, egyik a saját táblái és objektumai közötti leképzést írja le, a másik a saját leképzéseink leírására való. Nézzük meg, hogy az előbb megalkotott osztályaink és tábláink közötti leképzést hogyan kell leírni! Kezdjük a `Track` osztállyal:

```
<class-descriptor
      class="Track"
      table="track"
>
   <field-descriptor id="1"
      name="id"
      column="id"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="true"
   />
   <field-descriptor id="2"
      name="title"
      column="title"
      jdbc-type="VARCHAR"
   />
   <field-descriptor id="3"
      name="min"
      column="min"
      jdbc-type="INTEGER"
   />
   <field-descriptor id="4"
      name="sec"
      column="sec"
      jdbc-type="INTEGER"
   />
   <field-descriptor id="5"
      name="discId"
      column="discid"
      jdbc-type="INTEGER"
   />
```

Ekkor a `Track` osztályt a `track` táblához rendeltük, és az `id`, `title`, `min` és `sec` mezőit az azonos nevű oszlopokhoz. Az azonosítóról meg kellett mondani, hogy `INTEGER` JDBC típusú (vigyázzunk, nagybetűsnek kell lennie), elsődleges kulcs, és értékét növelni kell minden új sor beillesztésekor. Ezzel még a platformfüggetlenséget is biztosítottuk, amit a JDBC nem, ugyanis minden adatbázis-kezelő más módon oldja meg az egyes mezők automatikus növelését. Az OJB az összetett kulcsokat is támogatja. Most nézzük az előadó leképzését!

```
<class-descriptor
      class="Artist"
      table="artist"
>
   <field-descriptor id="1"
      name="id"
      column="id"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="true"
   />
   <field-descriptor id="2"
      name="name"
      column="name"
      jdbc-type="VARCHAR"/>
```

Az OJB támogatja az 1:1, 1:n, n:m kapcsolatok leírását és az öröklődést is. Minden lemezhez tartozik egy előadó, illetve minden lemezhez tartozik legalább egy szám is, amiket az lemez objektum egy `Colletion` mezőben tárol. Elvárható, hogy az lemez betöltésekor töltse be a hozzá tartozó előadót és számokat is. Ezeket az állításokat a következőképpen kell leírni:

```
<class-descriptor
      class="Disc"
      table="disc"
>
   <field-descriptor id="1"
      name="id"
      column="id"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="true"
   />
   <field-descriptor id="2"
      name="artistId"
      column="artistid"
      jdbc-type="INTEGER"
   />
   <field-descriptor id="3"
      name="title"
      column="title"
      jdbc-type="VARCHAR"
   />
   <field-descriptor id="4"
      name="date"
      column="date"
      jdbc-type="TIMESTAMP"
      conversion="org.apache.ojb.broker.accesslayer.
            conversions.JavaDate2SqlTimestampFieldConversion"
   />
   <field-descriptor id="5"
      name="single"
      column="single"
      jdbc-type="INTEGER"
      conversion="org.apache.ojb.broker.accesslayer.
            conversions.Boolean2IntFieldConversion"
   />
    <reference-descriptor
      name="artist"
      class-ref="Artist"
   >
      <foreignkey field-id-ref="2"/>
   </reference-descriptor>
   <collection-descriptor
      name="tracks"
      element-class-ref="Track"
      auto-retrieve="true"
      auto-update="true"
      auto-delete="true"
      orderby="id"
   >
      <inverse-foreignkey field-id-ref="5" />
   </collection-descriptor>
```

Ez a leképzés az OJB több tulajdonságát is megmutatja. Először is látható, hogy a `date` és `single` mezőkhöz egy un. conversion van megadva. Vannak beépített konverziók, és lehet saját konverziós osztályt is írni. Az első a `java.util.Date` objektumot képzi le `TIMESTAMP` JDBC típusra, a másik a `boolean` egyszerű típust `INTEGER` JDBC típusra. A `reference-descriptor` tag írja le az 1:1 leképzést, a `collection-descriptor` az 1:n leképzést. Az 1:1 esetén meg kell adni, hogy milyen osztályú a hivatkozott objektum, illetve meg kell adni, hogy melyik mező tartalmazza annak elsődleges kulcsát, azaz melyik a külső kulcs. Jelen esetben ez a 2-es számú, azaz az `artistId` mező. Az 1:n leképzés esetén meg kell adni a `Collection` nevét, mely tartalmazni fogja a hivatkozott objektumokat

(`tracks` mező), meg kell adni a hivatkozott objektumok osztályát (`Track`), illetve meg kell adni, hogy a hivatkozott objektum mely mezője a külső kulcs. Jelen esetben ez az 5-ös számú, ami a `Track` leírásánál a `discId` mező. Itt lehet megadni, hogy a szülő objektum példányosításakor, frissítésekor és törlésekor automatikusan a gyermek objektumok is az akciónak megfelelően változzanak. Meg lehet adni azt is, hogy a sorrend mi szerint legyen, jelen esetben az azonosító szerint.

Abban az esetben, ha n:m kapcsolatot akarunk reprezentálni, akkor azt megtehetjük kapcsoló objektummal, de akár a nélkül is. Persze kapcsoló táblára mindig szükség lesz. Az első esetben felbontjuk az n:m kapcsolatot két 1:n kapcsolatra, és annak megfelelően készítjük el a leképzést, az utóbbi esetben a `collection-descriptor` tag esetén attribútumként meg kell adni a kapcsoló tábla nevét (`indirection-table`), illetve belső tag-ekkel meg kell adni a két külső kulcsot: `fk-pointing-to-this-class`, `fk-pointing-to-element-class`.

Az OJB képes öröklődési hierarchiák, poliformizmus letárolására, illetve az alapján történő lekérdezésre is. Abban az esetben, ha egy osztály implementál egy interfészt, vagy kiterjeszt egy másik osztályt, akkor az őst is le kell írni a konfigurációs állományban (absztrakt osztály vagy interfész esetén is, de akkor nem kell megadni hozzá mezőt), és definiálni kell azokat az osztályokat, melyek implementálják vagy kiterjesztik. A konfigurációs fájlban ezt a következőképpen kell leírni interfész kiterjesztése esetén:

```
<class-descriptor class="DiscInterface">
      <extent-class class-ref="Disc" />
</class-descriptor>
```

Abban az esetben, ha kiterjesztésről beszélünk:

```
<class-descriptor class="Disc">
      <extent-class class-ref="Album" />
      <extent-class class-ref="Promo" />
      …
</class-descriptor>
```

Ilyenkor szerepeltetni kell az Album és a Promo osztály osztályleíróját is a konfigurációs fájlban. Lehetőség van az összes objektum egyazon táblában való eltárolására is.
Ezekben az esetekben a lekérdezést nem csak a konkrét osztállyal adhatjuk meg, hanem interfésszel vagy ősosztállyal is. Az OJB biztosítja a gyermekek azonosítójának egyediségét.

Alkalmazás

Az OJB konfigurálható egy `OJB.properties` fájlal, melyet alapértelmezésben a `CLASSPATH`-ban keres az OJB, de az `OJB.properties` property beállítással is megadhatjuk annak teljes elérési útvonalát. Ez a konfigurációs állomány tartalmazza a leképzést végző fájl nevét és elérési útját, pool méretét, loggolással kapcsolatos beállításaokat, és még sok egyéb paramétert.

A futtatáshoz szükség van még az OJB JAR fájlára (a cikk írásakor a `jakarta-ojb-0.9.5.jar` fájlt használtam, akkor még a Jakarta project része volt, de azóta kijött az 1.0 RC1 verzió is). Tipikus használatkor a következő osztályokat kell importálni a forrás fájlunkban:

```
import org.apache.ojb.broker.PersistenceBroker;
import org.apache.ojb.broker.PersistenceBrokerFactory;
import org.apache.ojb.broker.PersistenceBrokerException;
import org.apache.ojb.broker.query.Criteria;
import org.apache.ojb.broker.query.Query;
import org.apache.ojb.broker.query.QueryFactory;
```

Tárolás és törlés

A következő kódrészlet végzi egy lemez tárolását:

```
PersistenceBroker broker = null;
 try
 {
     broker = PersistenceBrokerFactory. defaultPersistenceBroker();
 }
 catch (Throwable t)
 {
     t.printStackTrace();
 }
 try
 {
     broker.beginTransaction();
     broker.store(aDisc);
     broker.commitTransaction();
 }
 catch (PersistenceBrokerException ex)
 {
     broker.abortTransaction();
     ex.printStackTrace();
 }
```

A kódból is látható, hogy az OJB támogatja a tranzakciókat, szinkronizációs pont kijelölésére a `PersistenceBroker.beginTransaction()`, commithoz a `PersistenceBroker.commitTransaction()`, rollbackhez a `PersistenceBroker. abortTransaction()` metódus használatos. A `store()` metódust nem csak új objektum tárolásakor, hanem már létező objektum módosításakor is használjuk. Objektum törlésekor a `store()` helyett a `delete()` metódus alkalmazandó.

Lekérdezés

Objektumok lekérdezésekor `Criteria` objektumokat kell összeállítanunk a feltételek meghatározására. A lekérdezést maga egy `Query` objektum végzi, melyet a `QueryFactory`-tól kell kérni, és meg kell adni az osztályát az objektumoknak, amiket le akarunk kérdezni, és magát a feltételt.

A következő kódrészlet azokat a lemezeket kérdezi le, melyek kislemezek a megjelenés sorrendjében.

```
Criteria criteria = new Criteria();
Criteria.addEqualTo("single", Boolean.TRUE);
criteria.addOrderByDescending("date");

Query query = QueryFactory.newQuery(Disc.class, criteria);

Collection discs = null;

 try
 {
        discs = broker.getCollectionByQuery(query);
 }
 catch (PersistenceBrokerException ex)
 {
     System.out.println(ex.getMessage());
     ex.printStackTrace();
 }
```

Természetesen az OJB támogatja az összes lehetőséget, melyet az SQL lekérdező nyelv biztosít. Ilyenek az összehasonlító operátorok, `LIKE` feltétel, `BETWEEN`, `IN` kulcsszó. A `Criteria` objektumokat logikai kapcsolatba lehet egymással állítani. Megengedett a sorba rendezés és csoportosítás is. Végső esetben konkrét SQL feltételt is meg lehet adni, hiszen az

előzőekből is az OJB SQL feltételt generál (minek kiírására debug célból megvan a lehetősége).

Rendelkezik olyan speciális képességekkel, mint objektum cache-elés, adatbázis kapcsolatok automatikus pool-ozása, laza kötés virtuális proxy-kon keresztül (egy objektum betöltésekor nem feltétlenül kerül betöltésre a hozzá kapcsolódó objektum, csak akkor, ha arra szükség van), illetve elosztott zárolás-kezelés.

Amint az a Java nyelven írt Apache projektektől elvárható, rendelkezik Log4J támogatással, a build környezet Ant-tal van felépítve és több, mint 120 JUnit teszt esetet tartalmaz a regressziós teszthez, és a disztribúcióban benne foglaltatik a részletes dokumentáció, API-dokumentáció, tutoriálok és példa programok.

Az OJB tetszőlegesen skálázható, hiszen azon kívül, hogy a PersistenceBroker egy JVM-ben fut magával az alkalmazással, lehetőség van elosztott rendszereknél arra, hogy több kliensalkalmazás csatlakozzon több szerverhez, és a terheléselosztás érdekében a kliens mindig a legkevésbé terhelt szerverhez forduljon.

A mapping manuális beállítása helyett folyamatban van olyan eszközök megírása, melyet ezt automatikusan elvégzik. Több megközelítés is lehetséges:

- Java osztályból (vagy UML modellből) SQL DDL (Data Definition Language) utasítást készít, mely automatikusan legenerálja a táblákat.
- SQL DDL-ből (vagy konkrét adatbázisból) legenerálja a Java osztályokat.
- Felület a mapping elvégzésére, ha az adatbázis és az osztályok adottak.

Ezen eszközök egyike sem adott még, mindegyik fejlesztés alatt áll.

Konklúzió

Az OJB egy kitűnő eszköz objektumok leképzésére relációs adatbázisokra. Egyszerűbb alkalmazások esetén sikeresen kiváltja az SQL alkalmazását, és tökéletesen beleillik az objektumorientált szemléletbe.

Bonyolultabb alkalmazások esetén azonban felléphetnek olyan esetek, ahol a relációs adatbázis nem szabványos lehetőségeit, mélyebb funkcióit is kihasználjuk, hogy mégis szükség van az OJB megkerülésére, esetleg módosítására. Szerencsére ez is lehetséges, hiszen forráskódja is hozzáférhető. Arról azonban ne feledkezzünk meg, hogy ez egy újabb, teljesen Java nyelven írt réteg az alkalmazásunkban, mely nagy mennyiségű adatok lekérdezésekor, objektumok példányosításakor jelentős lassulást okozhat.

Budapest, 2003. március 14.

Hivatkozások

The Apache Software Foundation http://www.apache.org
The Apache DB Project http://db.apache.org
ObJectRelationalBridge http://db.apache.org/ojb/
Object Relation Tool Comparison
http://c2.com/cgi-bin/wiki?ObjectRelationalToolComparison

Szerzőről

Viczián István a Debreceni Egyetem programtervező matematikus szakán végzett 2001 nyarán, ahol most levelező PhD. hallgató az elosztott rendszerek, middleware-ek témakörében. Jelenleg szoftver fejlesztőként dolgozik Budapesten, melynek keretében middleware szoftverekkel, alkalmazásintegrációval, webes technológiákkal valamint application mining-gal és reverse enginering-gel foglalkozik. Szabadidejében internetes portált fejleszt, és Java technológiákkal ismerkedik, valamint Java blog-ot ír (JTechLog).
E-mail: viczus@freemail.hu
Honlap: http://dragon.unideb.hu/~vicziani