JAVA JDBC
CACHED ROW SET

Zlatko Sirotić, univ.spec.inf.

ISTRA TECH d.o.o.

Pula

Neki autorovi radovi zadnjih godina

- JavaCro 2016: Java paralelizacija II - Streams
- HrOUG 2015a: Povratak u Prolog
- HrOUG 2015b: Kada Oracle naredba nije serijabilna
- CASE 2015: Višestruko nasljeđivanje – san ili Java 8?
- JavaCro 2015: Java paralelizacija
- HrOUG 2014: Nasljeđivanje je dobro, naročito višestruko - Eiffel, C++, Scala, Java 8
- CASE 2014: Trebaju li nam distribuirane baze u vrijeme oblaka?
- JavaCro 2014: Da li postoji samo jedna "ispravna" arhitektura web poslovnih aplikacija

Teme

- Java Database Connectivity (JDBC)
- Result set (interface ResultSet)
- Konekcije, connection pool i proxy user
- Updatable result set
- Row set (interface RowSet)
- Cached row set (interface CachedRowSet)

Java Database Connectivity (JDBC)

- JDBC je aplikacijsko programsko sučelje (API) za programski jezik Java, koje definira kako klijent može koristiti bazu podataka. JDBC je dio Java Standard Edition (JSE) platforme, koja je sada vlasništvo Oracle korporacije.
- JDBC se prvi put pojavio 1997. godine, kada je Sun Microsystem izbacio drugu verziju Jave, tj. Java Development Kit (JDK) 1.1. Od tada nadalje, JDBC je dio JSE. Trenutačno je zadnja verzija JDBC 4.2, koja je uključena u Java SE 8.
- JDBC je od početka podržavao tzv. result set (klasa ResultSet), pomoću kojeg u JDBC-u klijent pristupa podacima dobivenim iz baze. Kroz specifikaciju JSR 114, uveden je i tzv. row set, koji je nadograđen nad result setom, i koji pruža dodatne mogućnosti manipulacije podacima kroz JDBC.

Result set (interface ResultSet)

```
/* 1.Primjer: ResultSet sa SELECT naredbom */ ...
public class P1ResultSet {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query =
    "select naziv from m_artikli order by naziv";
  try (Connection conn =
          DriverManager.getConnection(url, shema, lozinka);
       Statement stmt = conn.createStatement();
       ResultSet rs = stmt.executeQuery(query);
      ) // try-with-resources, postoji od Jave 7
  {while (rs.next()) System.out.println(rs.getString(1));}
} // main
} // class P1ResultSet
```

Konekcije, connection pool i proxy user

- Konekcije baze podataka u današnjim web i mobilnim aplikacijama rijetko se kreiraju i zatvaraju kod svake upotrebe. Razlog je taj što kreiranje konekcije traje neko vrijeme, koje nije zanemarivo (reda je npr. stotinjak milisekundi, ili više).
- Zbog toga se, za razliku od 1. primjera, konekcije često koriste kroz connection pool. Nakon upotrebe, konekcije se stvarno ne zatvaraju, već se vraćaju u connection pool (zatvaraju se logički, a ne i fizički).
- 2. primjer Java programa pokazuje korištenje connection poola (u konkretnom slučaju je to Oracle Universal Connection Pool, koji je raspoloživ od Oracle baze 11g). U primjeru se kreira connection pool koji ima maksimalno 4 konekcije, a inicijalno se kreiraju 3 konekcije.

Konekcije, connection pool i proxy user

- U 2. primjeru se koristi i tzv. proxy user. Naime, problem koji se može pojaviti kod connection poola jeste taj da svaki korisnik baze podataka dobiva svoj connection pool.
- To obično nije problem, jer današnje web aplikacije najčešće rade kroz samo jednog korisnika baze podataka (iako to nije baš najbolje za sigurnost baze podataka).
- Ako u nekoj aplikaciji želimo zadržati (npr. zbog sigurnosti) odnos jedna osoba = jedan korisnik na bazi, tada svakako želimo izbjeći kreiranje connection poola za svakog korisnika. Uobičajeno se to radi kroz proxy usera. On nam služi kao "vlasnik" connection poola, a "pravi" korisnici baze ulaze preko proxy usera u bazu.

Konekcije, connection pool i proxy user

- Primijetimo da (barem u Oracle bazi) **konekcija baze podataka i sesija baze podataka nisu istovrsni pojmovi!**
- **Jedna konekcija baze podataka može imati nula, jednu ili više sesija baze podataka**. Istina, najčešće se kroz jednu konekciju koristi samo jedna sesija.
- No, baš u 2. primjeru to nije tako. Vidimo da se npr. na početku otvore tri konekcije i tri sesije za proxy usera, uproxy. Međutim, kada kroz takvu konekciju uđemo kao pravi korisnik u1 (kroz proxy usera), vidimo da broj konekcija ostaje isti, a broj sesija se povećava za jedan, tj. kroz jednu konekciju sada idu dvije sesije, za korisnike uproxy i u1.
- Istina, korisnička sesija uproxy je u ovom slučaju nepristupačna, ali u nekim drugim slučajevima zaista možemo imati dvije aktivne sesije kroz istu konekciju.

Konekcije, connection pool i proxy user

```
/* 2.Primjer: Universal Connection Pool i proxy user */ ...
public class P2UCPProxy {
public static void main(String args[])
  throws SQLException, Exception {
  try {
    //Create pool-enabled data source instance.
    PoolDataSource pds =
      PoolDataSourceFactory.getPoolDataSource();
    //set the connection properties on the data source.
    pds.setConnectionFactoryClassName
      ("oracle.jdbc.pool.OracleDataSource");
    pds.setURL("jdbc:oracle:thin:@localhost:1521:ORCL");
    pds.setUser("uproxy");
    pds.setPassword("uproxy");
    //Override any pool properties.
    pds.setInitialPoolSize(3);
    pds.setMaxPoolSize(4);
```

Konekcije, connection pool i proxy user

```
    System.out.println("Prije otvaranja logicke konekcije\n");
    Thread.sleep (5_000);
    //Get a database connection from the datasource.
    Connection conn1 = pds.getConnection();
... // vide se tri konekcija uproxy
    Properties prop1 = new Properties();
    prop1.put(OracleConnection.PROXY_USER_NAME, "u1");
    prop1.put(OracleConnection.PROXY_USER_PASSWORD, "u1");
    ((OracleConnection)conn1).openProxySession
      (OracleConnection.PROXYTYPE_USER_NAME, prop1);
... // vide se tri sesije uproxy i jedna u1,
    // ali samo su tri procesa (konekcije)
    Connection conn2 = pds.getConnection();
    ((OracleConnection)conn2).openProxySession
      (OracleConnection.PROXYTYPE_USER_NAME, prop1);
... // vide se tri sesije uproxy i dvije u1,
    // ali samo su tri procesa (konekcije)
```

Konekcije, connection pool i proxy user

```
    Connection conn3 = pds.getConnection();
    ((OracleConnection)conn3).openProxySession
      (OracleConnection.PROXYTYPE_USER_NAME, prop1);
... // vide se tri sesije uproxy i tri u1,
    // ali samo su tri procesa (konekcije)
    Connection conn4 = pds.getConnection();
    ((OracleConnection)conn4).openProxySession
      (OracleConnection.PROXYTYPE_USER_NAME, prop1);
... // vide se ČETIRI sesije uproxy i četiri u1,
    // ali samo su četiri procesa (konekcije)
    ((ValidConnection) conn4).setInvalid();
    conn4.close(); conn4 = null;
... // ostaju tri procesa, svi kao uproxy
    ((ValidConnection) conn3).setInvalid();
    conn3.close(); conn3 = null;
... // ostaju dva procesa, svi kao uproxy
```

Konekcije, connection pool i proxy user

```
    ((ValidConnection) conn2).setInvalid();
    conn2.close(); conn2 = null;
... // ostaje jedan proces, kao uproxy
    ((ValidConnection) conn1).setInvalid();
    conn1.close(); conn1 = null; ... // nema više procesa
  } catch (Exception e) {System.out.println(e.toString());}
} // main

public static void printStatistics(final PoolDataSource pds) {
 JDBCConnectionPoolStatistics statistics = pds.getStatistics();
 System.out.println(statistics.getAvailableConnectionsCount());
 System.out.println(statistics.getBorrowedConnectionsCount());
 System.out.println(statistics.getConnectionsClosedCount());
} // printStatistics

} // class P2UCPProxy
```

Updatable result set

- Result set ne mora služiti samo za čitanje, već i za izmjene.
- Prvo, postoje tri tipa result set kurzora:

TYPE_FORWARD_ONLY (default): kurzor ide samo naprijed;

TYPE_SCROLL_INSENSITIVE: može se micati naprijed ili nazad, ili na određenu poziciju; no, takav kurzor ne vidi promjene na bazi – otuda riječ INSENSITIVE;

TYPE_SCROLL_SENSITIVE: može se micati naprijed, nazad, ili na određenu poziciju; dodatno, promjene na bazi (ali ne sve) reflektiraju se na result set, tj. kurzor ih vidi - otuda riječ SENSITIVE; rijetki su JDBC driveri koji ga implementiraju.

- Dalje, što se tiče ažuriranja, postoje dva tipa result seta:

CONCUR_READ_ONLY (default);

CONCUR_UPDATABLE: result set može se mijenjati.

Updatable result set

- U 3. primjeru Java programa koristi se result set koji je TYPE_SCROLL_SENSITIVE i CONCUR_UPDATABLE.

- Primijetimo dvije važne stvari:

- jako je bitno da se na početku transakcije kaže **conn.setAutoCommit(false);**
  ako to ne kažemo, JDBC default ponašanje je da nakon svake DML naredbe radi automatski COMMIT;
- kad napravimo **rs.updateRow();**
  izmjena retka (UPDATE; isto vrijedi za DELETE i INSERT) **odmah se šalje na bazu**, što kao posljedicu ima i to da je za drugu sesiju baze podataka taj redak odmah zaključan, i ostaje zaključan do kraja transakcije, tj. do COMMIT ili ROLLBACK.

Updatable result set


```
/* 3.Primjer: ResultSet sa UPDATE */ ...
public class P3UpdateResultSet {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query =
    "select sifra, naziv from m_artikli order by naziv";
  try (Connection conn =
         DriverManager.getConnection(url, shema, lozinka);
       Statement stmt =
         conn.createStatement
          (ResultSet.TYPE_SCROLL_SENSITIVE,
           ResultSet.CONCUR_UPDATABLE);
       ResultSet rs = stmt.executeQuery(query);
      )
```

Updatable result set


```
    conn.setAutoCommit(false);
    while (rs.next()) {
      System.out.println
        ("Prije UPDATE " + rs.getString("naziv"));
      rs.updateString("naziv", "XXX");
      rs.updateRow(); // nakon ovoga, "vanjski UPDATE čeka"
      System.out.println
        ("Nakon UPDATE " + rs.getString("naziv"));
      Thread.sleep(5_000);
    }
    conn.rollback();
  } catch (Exception e) {
    System.out.println(e.toString());
  }
} // main

} // class P3UpdateResultSet
```

Row set (interface RowSet)

- JDBC row set nije nova stvar. Zapravo, pojavio se već u JDBC 2.0 (kada su se pojavili npr. i scrollable result set i updatable result set), ali kao opcionalni paket.
- Implementacija za row set standardizirana je kroz JDBC RowSet Implementations Specification (u JSR-114), kao ne-opcionalni paket, od Java SE 5.0 dalje. Row set se naslanja na result set, tj. RowSet sučelje nasljeđuje ResultSet sučelje.
- Neke prednosti row seta u odnosu na result set su sljedeće:

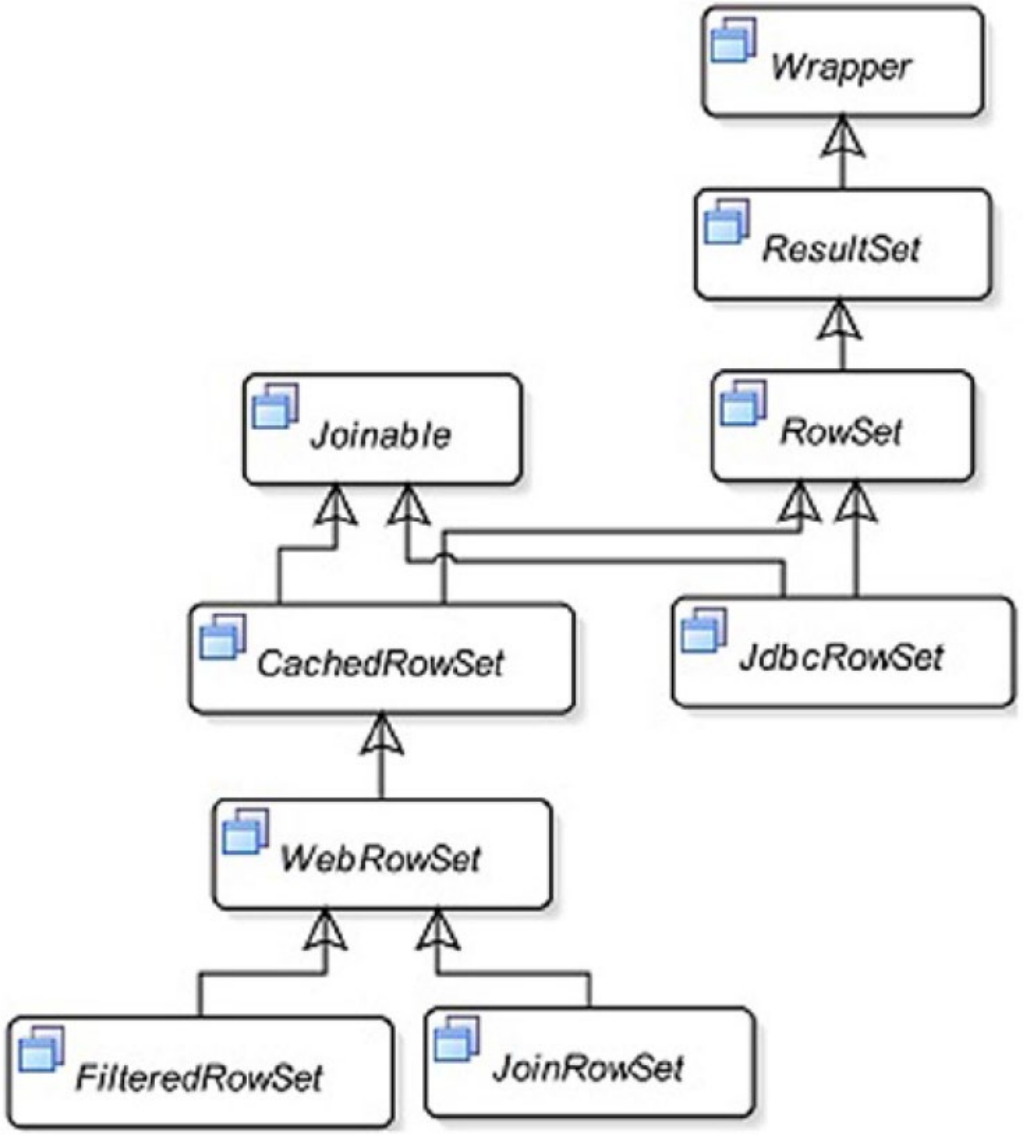
- programiranje sa row setom je jednostavnije; kada se koristi result set, mora se eksplicitno raditi s konekcijama (Connection) i naredbama (Statement), koji su kod row seta skriveni od programera;
- row set je serijabilan (Serializable; result set nije), pa **može biti poslan preko mreže, ili snimljen na disk za kasniju upotrebu;**

Row set (interface RowSet)

- result set mora uvijek biti spojen na izvor podataka, dok row set može biti i odspojen; **row set se može spojiti na bazu podataka npr. samo onda kada čita ili piše u bazu;**
- result set koristi bazu podataka kao izvor podataka; row set može koristiti i druge izvore podataka koji vraćaju podatke u tabularnom obliku;
- **serijabilnost row seta i mogućnost da radi i kada nije spojen na izvor podataka, čine row set vrlo korisnim za web ili mobilne aplikacije.**

❖ Row set može imati i nedostataka. Budući da se kod nekih row set klasa podaci keširaju u memoriji klijenta, može se javiti veliko zauzeće memorije (ako se na klijent učita velika količina podataka). Dalje, javlja se veća vjerojatnost nekonzistencije između podataka na klijentu i na izvoru.

Row set (interface RowSet)



Cached row set (interface CachedRowSet)

- Kako je prikazano prethodnoj slici, CachedRowSet ima i podklase (zapravo, sučelja):

- **WebRowSet**: omogućava čitanje i pisanje podataka / metapodataka kao XML dokumenta;
- **FilteredRowSet**: omogućava filtriranje podataka na klijentskoj strani;
- **JoinRowSet**: omogućava spajanje (join) dva ili više row seta u jedan row set, na klijentskoj strani.

- Sljedeći Java program pokazuje korištenje sučelja CachedRowSet. **Nakon čitanja podataka sa baze (nakon crs.execute();), klijent se automatski odspaja.**
- Dok je klijent odspojen, nad lokalnim podacima radi se UPDATE (crs.updateRow();), DELETE (crs.deleteRow();) i INSERT (crs.insertRow();).

Cached row set (interface CachedRowSet)

- **Nakon što se promjene pošalju na bazu (crs.acceptChanges(); - pritom se klijent automatski konektira na bazu), stvarno se ažurira baza podataka.**
- Kako je označeno u programskom kodu (kao komentar), može se desiti sljedeće:

- ako netko u međuvremenu na bazi mijenja ili izbriše (i napravi COMMIT) redak koji se lokalno mijenjao, lokalna izmjena neće imati efekta, ali se na klijentu neće javiti greška;
- ako netko u međuvremenu na bazi izbriše (i napravi COMMIT) redak koji se lokalno brisao, na klijentu se neće javiti greška;
- ako netko u međuvremenu na bazi unese (i napravi COMMIT) redak koji se lokalno unio, na bazi će se možda javiti greška - ako je zbog toga npr. narušen UK (unique key).

Cached row set (interface CachedRowSet)


```
/* 4.Primjer: Cached Row Set sa UPDATE */
public class P4CachedRowSetTest {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query =
    "select sifra, naziv from m_artikli order by naziv";
  try {
    OracleCachedRowSet crs = new OracleCachedRowSet();
    crs.setUrl(url);
    crs.setUsername(shema);
    crs.setPassword(lozinka);
    crs.setCommand(query);
    crs.execute();
    crs.next(); // idemo na 1.redak
```

Cached row set (interface CachedRowSet)


```
System.out.println("Prije UPDATE:"+crs.getString("naziv"));
Thread.sleep(5_000);
crs.updateString("naziv", "XXX");
crs.updateRow();
System.out.println("Nakon UPDATE:"+crs.getString("naziv"));
Thread.sleep(5_000);
// ako se izvana promijeni redak i da commit,
// ova izmjena se neće napraviti, ali ne desi se greška

crs.absolute(3);
System.out.println("Prije DELETE:"+crs.getString("naziv"));
Thread.sleep(5_000);
crs.deleteRow(); // brišemo 3. redak
System.out.println("Nakon DELETE ");
Thread.sleep(5_000);
// ako se izvana izbriše redak i da commit,
// ne desi se greška
```

Cached row set (interface CachedRowSet)


```
    crs.moveToInsertRow();
    crs.updateString("sifra", "9");
    crs.updateString("naziv", "A9");
    crs.insertRow();
    crs.moveToCurrentRow();
    System.out.println("Nakon INSERT i prije accept");
    Thread.sleep(5_000);
    // ako se izvana unese redak sa istim PK / UK i da commit,
    // desi se greška na bazi (nakon crs.acceptChanges())

    crs.acceptChanges(); // odmah je commit
    System.out.println("Nakon acceptChanges");
    Thread.sleep(5_000);
  } catch (Exception e) {System.out.println(e.toString());}
} // main

} // class P4CachedRowSetTest
```

Cached row set (interface CachedRowSet)

- Kako je već rečeno, kod cached row seta mogu se javiti konflikti između lokalnih ažuriranja i ažuriranja koje je netko drugi u međuvremenu napravio u bazi podataka (ili drugom izvoru podataka).
- Kada se detektira konflikt kod izvršavanja metode acceptChanges(), dolazi do **SyncProviderException** iznimke. Tada se može koristiti synchronization resolver objekt (instanca od SyncResolver) za rješavanje konflikta, kako prikazuje sljedeći isječak koda:

```
catch (SyncProviderException spe) {
  // Kada acceptChanges() detektira neki konflikt
  SyncResolver resolver = spe.getSyncResolver();
  // Procedura ispisuje detalje o konfliktu
  printConflicts(resolver, cachedRs);
}
```

Cached row set (interface CachedRowSet)

- SyncResolver objekt omogućava navigaciju kroz sve konflikte, te omogućava da izmijenimo retke u row setu:

```
public static void printConflicts ...
  try {
    while (resolver.nextConflict()) {
      int status = resolver.getStatus();
      String operation = "None";
      if (status == INSERT_ROW_CONFLICT) operation = "insert";
...   // Čitamo person ID sa baze
      Object oldPerId = resolver.getConflictValue("per_id");
      // Čitamo person ID iz cached row seta
      int row = resolver.getRow();
      cachedRs.absolute(row);
      Object newPersonId = cachedRs.getObject("per_id");
      // Koristimo setResolvedValue() metodu
      // da postavimo resolved value za stupac
      // resolver.setResolvedValue(columnName, resolvedValue);
```

Zaključak

- Row set je serijabilan (result set nije), pa može biti poslan preko mreže, ili snimljen na disk za kasniju upotrebu.
- Dalje, dok result set mora uvijek biti spojen na izvor podataka, row set može biti i odspojen. Row set se može spojiti na bazu podataka npr. samo onda kada čita ili piše podatke u bazu.
- To čini row set vrlo korisnim za web ili mobilne aplikacije. Klijent ne mora imati niti JDBC driver, jer može pokupiti podatke (ili ih ažurirati) sa srednjeg sloja, koristeći odgovarajući row set.
- Mogu se javiti konflikti između lokalnih ažuriranja i ažuriranja koje je netko drugi u međuvremenu napravio u bazi podataka (ili drugom izvoru podataka). No, ti se konflikti mogu programski detektirati (kroz SyncProviderException iznimku) i rješavati (pomoću synchronization resolver objekta).

Literatura

- Boyarsky, J., Selikoff, S. (2015): OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809, Sybex
- Menon, R., M. (2005): Expert Oracle JDBC Programming, Apress
- Sharan, K. (2014): Java 8 APIs, Extensions and Libraries, Apress
- Sierra K., Bates B. (2015): OCA/OCP Java SE 7 Programmer I & II Study Guide, McGraw-Hill Education
- Oracle priručnik (2013): Oracle Database JDBC Developer's Guide 12c Release 1 (12.1) E17657-14
- Oracle priručnik (2013): Oracle Universal Connection Pool for JDBC Developer's Guide 12c Release 1 (12.1) E17659-10