15. Program a databáze

Obsah

- Spojení do databáze
- Použití SQL v programu
- Práce s daty
- Objektově relační mapování (ORM)
- Transakce nad objekty
- Vhodné užití, úskalí a efektivita
- Příklady

Spojení do databáze

Spojení do databáze je prvním krokem při práci s databází v aplikaci. Program zůstává stejný i při změně databáze. Nejčastěji se používá JDBC v Javě nebo obdobné knihovny v jiných jazycích.

JDBC (Java Database Connectivity)

JDBC je standardní API pro připojení k relačním databázím v Javě.

```
// Základní připojení pomocí JDBC
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/databaze",
    "uzivatel",
    "heslo"
);
```

Connection Pooling

Connection pooling udržuje několik připojení k databázi v "poolu", což zlepšuje výkon aplikace.

```
// Pseudokód pro connection pooling
ConnectionPool pool = new ConnectionPool(
    "jdbc:mysql://localhost:3306/databaze",
    "uzivatel",
    "heslo"
);
Connection conn = pool.getConnection();
// ...použití spojení...
conn.close(); // Vrátí spojení do poolu, neuzavře ho skutečně
```

Použití SQL v programu

Přímé vykonávání SQL příkazů s JDBC

```
// Vykonání jednoduchého dotazu
Connection conn = getConnection();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT id, jmeno, prijmeni FROM uzivatele");

// Zpracování výsledků
while (rs.next()) {
    int id = rs.getInt("id");
    String jmeno = rs.getString("jmeno");
    String prijmeni = rs.getString("prijmeni");
    System.out.println(id + ": " + jmeno + " " + prijmeni);
}
```

Prepared Statements

Prepared Statements poskytují: - Ochranu proti SQL injection - Lepší výkon při opakovaném vykonávání - Lepší práci s datovými typy

```
// Použití prepared statement
Connection conn = getConnection();
String sql = "SELECT * FROM uzivatele WHERE prijmeni = ? AND vek > ?";
PreparedStatement pstmt = conn.prepareStatement(sql);

// Nastavení parametrů
pstmt.setString(1, "Novák");
pstmt.setInt(2, 30);

// Vykonání dotazu
ResultSet rs = pstmt.executeQuery();
```

Batch Processing

Batch processing umožňuje odeslat více příkazů najednou, což zlepšuje výkon.

```
// Batch processing s JDBC
Connection conn = getConnection();
conn.setAutoCommit(false); // Vypnutí automatických commitů

PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO uzivatele (jmeno, prijmeni) VALUES (?, ?)"
);

// Přidání záznamů do dávky
pstmt.setString(1, "Jan");
pstmt.setString(2, "Novák");
pstmt.addBatch();

pstmt.setString(1, "Petr");
pstmt.setString(2, "Svoboda");
pstmt.addBatch();

// Vykonání dávky
int[] counts = pstmt.executeBatch();
conn.commit();
```

Práce s daty

Mapování výsledků dotazu na objekty

V reálných aplikacích obvykle mapujeme výsledky dotazů na objekty.

```
// Pseudokód pro mapování výsledků na objekty
List<User> getAllUsers() {
    List<User> users = new ArrayList<>();
    Connection conn = getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT id, jmeno, prijmeni, vek FROM uzivatele");

    while (rs.next()) {
        User user = new User();
        user.id = rs.getInt("id");
        user.jmeno = rs.getString("jmeno");
        user.prijmeni = rs.getString("prijmeni");
```

```
            user.vek = rs.getInt("vek");
            users.add(user);
        }

        return users;
    }
```

Data Access Object (DAO)

DAO je návrhový vzor, který poskytuje abstraktní rozhraní k datovému úložišti.

```
// Příklad rozhraní DAO
interface UserDAO {
    User findById(int id);
    List<User> findAll();
    void save(User user);
    void update(User user);
    void delete(int id);
}

// Implementace DAO s JDBC
class JdbcUserDAO implements UserDAO {
    @Override
    public User findById(int id) {
        Connection conn = getConnection();
        PreparedStatement pstmt = conn.prepareStatement(
            "SELECT * FROM uzivatele WHERE id = ?"
        );
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            return mapUserFromResultSet(rs);
        }
        return null;
    }

    // další metody...
}
```

Objektově relační mapování (ORM)

ORM převádí data mezi relační databází a objektově orientovaným programováním. Interpretace: Model, Třída = Tabulka, Objekty = Řádky

Výhody ORM

- Zjednodušuje přístup k datům
- Minimalizuje množství kódu
- Poskytuje abstrakci nad SQL
- Spravuje spojení, transakce a cachování

Nevýhody ORM

- Může být pomalejší než přímé SQL dotazy
- Učící křivka
- Může vést k neefektivním dotazům (N+1 problém)
- Omezená kontrola nad generovanými SQL dotazy

Hibernate (Java)

Hibernate je populární ORM framework pro Javu.

```java
// Příklad entity v Hibernate
@Entity
@Table(name = "uzivatele")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String jmeno;
    private String prijmeni;
    private int vek;

    // gettery a settery...
}

// Příklad použití Hibernate
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

// Vytvoření uživatele
User user = new User();
user.setJmeno("Jan");
user.setPrijmeni("Novák");
user.setVek(30);
session.save(user);

// Získání uživatele
User retrievedUser = session.get(User.class, 1);

// Dotazování pomocí HQL (Hibernate Query Language)
List<User> users = session.createQuery("FROM User WHERE vek > 30").list();

tx.commit();
session.close();
```

JPA (Java Persistence API)

JPA je standardní API pro ORM v Javě.

```java
// Použití JPA
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
User user = new User("Jan", "Novák", 30);
em.persist(user);
em.getTransaction().commit();

// Dotazování pomocí JPQL
TypedQuery<User> query = em.createQuery(
    "SELECT u FROM User u WHERE u.vek > :vek", User.class
);
query.setParameter("vek", 30);
List<User> results = query.getResultList();
```

```
em.close();
emf.close();
```

Transakce nad objekty

Transakce zajišťují ACID vlastnosti (Atomicita, Konzistence, Izolace, Trvalost) při práci s databází.

JDBC Transakce

```
// Transakce s JDBC
Connection conn = getConnection();
try {
    conn.setAutoCommit(false); // Začátek transakce

    // Provedení několika SQL příkazů
    PreparedStatement pstmt1 = conn.prepareStatement(
        "UPDATE ucty SET zustatek = zustatek - ? WHERE id = ?"
    );
    pstmt1.setDouble(1, 1000);
    pstmt1.setInt(2, 1);
    pstmt1.executeUpdate();

    PreparedStatement pstmt2 = conn.prepareStatement(
        "UPDATE ucty SET zustatek = zustatek + ? WHERE id = ?"
    );
    pstmt2.setDouble(1, 1000);
    pstmt2.setInt(2, 2);
    pstmt2.executeUpdate();

    conn.commit(); // Potvrzení transakce
} catch (SQLException e) {
    conn.rollback(); // Odvolání transakce v případě chyby
    throw e;
} finally {
    conn.setAutoCommit(true);
    conn.close();
}
```

Transakce v ORM

ORM frameworky poskytují vysokoúrovňové API pro práci s transakcemi.

```
// Hibernate transakce
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();

    // Provedení operací
    Account account1 = session.get(Account.class, 1);
    account1.withdraw(1000);

    Account account2 = session.get(Account.class, 2);
    account2.deposit(1000);

    tx.commit();
} catch (Exception e) {
    if (tx != null) tx.rollback();
    throw e;
```

```
} finally {
    session.close();
}
```

Deklarativní transakce (Spring)

Spring Framework umožňuje deklarativní správu transakcí pomocí anotací.

```
// Deklarativní transakce ve Spring
@Service
public class BankService {
    @Autowired
    private AccountRepository accountRepository;

    @Transactional
    public void transfer(int fromId, int toId, double amount) {
        Account from = accountRepository.findById(fromId);
        from.withdraw(amount);
        accountRepository.save(from);

        Account to = accountRepository.findById(toId);
        to.deposit(amount);
        accountRepository.save(to);
    }
}
```

Vhodné užití, úskalí a efektivita

Kdy použít JDBC přímo

- Pro jednoduché aplikace s malým počtem dotazů
- Když potřebujete maximální kontrolu nad SQL dotazy
- Pro specializované dotazy, které ORM neumí efektivně mapovat
- Pro dávkové operace s velkým množstvím dat

Kdy použít ORM

- Pro komplexní aplikace s mnoha entitami a vztahy
- Když je důležitější rychlost vývoje než absolutní výkon
- Pro týmové projekty, kde ORM poskytuje konzistentní přístup
- Když není potřeba psát specializované SQL dotazy

Časté problémy a jejich řešení

1. N+1 problém Problém: Pro každý záznam v kolekci se provede samostatný dotaz.

```
// Problematický kód (pseudokód)
List<Order> orders = orderRepository.findAll(); // 1 dotaz
for (Order order : orders) {
    Customer customer = order.getCustomer(); // N dodatečných dotazů
    // ...
}

// Řešení: Eager fetching nebo join fetch
List<Order> orders = orderRepository.findAllWithCustomers(); // 1 dotaz s JOIN
```

2. Neuzavřená spojení Problém: Neuvolněná databázová spojení mohou vést k únikům paměti.

```
// Správné uzavírání zdrojů s try-with-resources
try (
```

```
    Connection conn = getConnection();
    PreparedStatement pstmt = conn.prepareStatement(sql);
    ResultSet rs = pstmt.executeQuery()
) {
    while (rs.next()) {
        // zpracování výsledků
    }
}
```

3. SQL Injection Problém: Neošetřené vstupy mohou vést k SQL injection útokům.

```
// Nebezpečný kód
String sql = "SELECT * FROM uzivatele WHERE jmeno = '" + userInput + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);

// Bezpečné řešení
String sql = "SELECT * FROM uzivatele WHERE jmeno = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, userInput);
ResultSet rs = pstmt.executeQuery();
```

Optimalizace výkonu

1. Connection Pooling Opakované vytváření spojení je nákladné. Použití connection poolingu výrazně zvyšuje výkon.

2. Dávkové operace Pro velké množství operací používejte batch processing.

3. Indexy Ujistěte se, že tabulky mají vhodné indexy pro vaše dotazy.

4. Lazy vs. Eager Loading V ORM zvažte, kdy použít lazy loading (načítání dat až když jsou potřeba) a kdy eager loading (načítání souvisejících dat předem).

5. Cachování Použijte cachování pro často čtená, zřídka měněná data.

Příklady

Příklad 1: Jednoduchá aplikace pro správu uživatelů s JDBC

```
// Třída pro reprezentaci uživatele
class User {
    private int id;
    private String jmeno;
    private String prijmeni;
    private int vek;

    // konstruktory, gettery, settery...
}

// DAO třída pro práci s uživateli
class UserDAO {
    private Connection getConnection() throws SQLException {
        return DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/databaze",
            "uzivatel",
            "heslo"
```

```
        );
    }

    // Získání všech uživatelů
    public List<User> getAllUsers() throws SQLException {
        List<User> users = new ArrayList<>();
        String sql = "SELECT id, jmeno, prijmeni, vek FROM uzivatele";

        try (
            Connection conn = getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql)
        ) {
            while (rs.next()) {
                User user = new User();
                user.setId(rs.getInt("id"));
                user.setJmeno(rs.getString("jmeno"));
                user.setPrijmeni(rs.getString("prijmeni"));
                user.setVek(rs.getInt("vek"));
                users.add(user);
            }
        }

        return users;
    }

    // Přidání nového uživatele
    public void addUser(User user) throws SQLException {
        String sql = "INSERT INTO uzivatele (jmeno, prijmeni, vek) VALUES (?, ?, ?)";

        try (
            Connection conn = getConnection();
            PreparedStatement pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)
        ) {
            pstmt.setString(1, user.getJmeno());
            pstmt.setString(2, user.getPrijmeni());
            pstmt.setInt(3, user.getVek());

            pstmt.executeUpdate();

            try (ResultSet generatedKeys = pstmt.getGeneratedKeys()) {
                if (generatedKeys.next()) {
                    user.setId(generatedKeys.getInt(1));
                }
            }
        }
    }
}
```

Příklad 2: Transakce s JDBC

```
// Převod peněz mezi účty
public void transferMoney(int fromAccountId, int toAccountId, double amount) throws SQLException {
    Connection conn = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
```

```
            // Odečtení peněz z prvního účtu
            String updateSql1 = "UPDATE ucty SET zustatek = zustatek - ? WHERE id = ?";
            PreparedStatement pstmt1 = conn.prepareStatement(updateSql1);
            pstmt1.setDouble(1, amount);
            pstmt1.setInt(2, fromAccountId);
            pstmt1.executeUpdate();

            // Kontrola dostatečného zůstatku
            String checkSql = "SELECT zustatek FROM ucty WHERE id = ?";
            PreparedStatement checkStmt = conn.prepareStatement(checkSql);
            checkStmt.setInt(1, fromAccountId);
            ResultSet rs = checkStmt.executeQuery();

            if (rs.next() && rs.getDouble("zustatek") < 0) {
                throw new SQLException("Nedostatek prostředků na účtu");
            }

            // Přičtení peněz na druhý účet
            String updateSql2 = "UPDATE ucty SET zustatek = zustatek + ? WHERE id = ?";
            PreparedStatement pstmt2 = conn.prepareStatement(updateSql2);
            pstmt2.setDouble(1, amount);
            pstmt2.setInt(2, toAccountId);
            pstmt2.executeUpdate();

            conn.commit();
        } catch (SQLException e) {
            if (conn != null) {
                try {
                    conn.rollback();
                } catch (SQLException ex) {
                    // Log error
                }
            }
            throw e;
        } finally {
            if (conn != null) {
                try {
                    conn.setAutoCommit(true);
                    conn.close();
                } catch (SQLException e) {
                    // Log error
                }
            }
        }
    }
```

Příklad 3: Jednoduchá implementace ORM s JDBC

```
// Jednoduchá anotace pro označení primárního klíče
@interface PrimaryKey {}

// Anotace pro mapování sloupce
@interface Column {
    String name() default "";
}

// Třída pro mapování tabulky uživatelů
class User {
```

```
    @PrimaryKey
    @Column(name = "id")
    private int id;

    @Column(name = "jmeno")
    private String jmeno;

    @Column(name = "prijmeni")
    private String prijmeni;

    @Column(name = "vek")
    private int vek;

    // gettery a settery...
}

// Jednoduchý ORM mapper
class SimpleORM {
    private Connection conn;

    public SimpleORM(Connection conn) {
        this.conn = conn;
    }

    public <T> T findById(Class<T> clazz, int id) throws Exception {
        String tableName = clazz.getSimpleName().toLowerCase();
        String sql = "SELECT * FROM " + tableName + " WHERE id = ?";

        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            T instance = clazz.getDeclaredConstructor().newInstance();

            for (Field field : clazz.getDeclaredFields()) {
                if (field.isAnnotationPresent(Column.class)) {
                    field.setAccessible(true);
                    String columnName = field.getAnnotation(Column.class).name();
                    if (columnName.isEmpty()) {
                        columnName = field.getName();
                    }

                    if (field.getType() == int.class) {
                        field.set(instance, rs.getInt(columnName));
                    } else if (field.getType() == String.class) {
                        field.set(instance, rs.getString(columnName));
                    }
                    // další typy...
                }
            }

            return instance;
        }

        return null;
    }
}
```

Shrnutí

- **JDBC** poskytuje přímý přístup k databázi v Javě, je vhodný pro jednodušší aplikace nebo specifické požadavky.
- **Prepared statements** jsou nezbytné pro zabezpečení proti SQL injection a zvýšení výkonu.
- **DAO vzor** poskytuje abstraktní rozhraní k datovému úložišti a odděluje logiku přístupu k datům od obchodní logiky.
- **ORM** zjednodušuje mapování mezi objekty a relační databází, ale může mít výkonnostní dopady.
- **Transakce** zajišťují integritu dat i v případě chyb nebo souběžného přístupu.
- Pro optimální výkon je důležité správně používat connection pooling, dávkové operace a dbát na uzavírání zdrojů.
- Výběr mezi přímým SQL a ORM závisí na komplexnosti aplikace, požadavcích na výkon a preferencích vývojářů.