БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ и ИНФОРМАТИКИ
Кафедра многопроцессорных систем и сетей

Рафеенко Е.Д.

Web- программирование

Технология Java Persistence API

Содержание

- Java Persistence Query Language
- Использование Criteria API для создания запросов
- Управление параллельным доступом

The Java Persistence Query Language

Интерфейс **Query** позволяет работать с запросами (статическими и динамическими).

- Статические или именованные запросы объявляются перед классом-сущностью с помощью аннотации @**NamedQuery**. Атрибуты аннотации: имя запроса (name) и его текст (query). Создается именованный запрос вызовом метода createNamedQuery.
- Динамические запросы создаются методом createQuery(String qlString).

Запросы должны быть написаны на языке JPA-QL. Основное его отличие от стандартного языка заключается в том, что вместо имен таблиц и столбов указывают имена сущностей и их свойств.

The Java Persistence Query Language

- В запрос можно передать параметры, либо по имени (тогда в запросе они указаны как :parameterName) либо по порядковому номеру. Для этого используется метод **setParameter**.

- Для получения результата выполнения запроса используется метод **getResultList** для списка выбранных объектов или **getSingleResult** для единственного объекта.

- Существует возможность выборки не всех записей, а только ограниченного их числа. Для этого нужно в запросе установить индекс первой записи (setFirstResult) и максимальное количество записей (setMaxResult).

The Java Persistence Query Language

- Запросы с параметрами:

именованные параметры :name

позиционные параметры ?1

Пример - простой запрос с параметром:

```
@NamedQuery(name = "studentByLastName",
   query = "select st " +
  "from Student st " +
      "where st.lastName = :lastName"),
```

The Java Persistence Query Language

- Запросы к связанным сущностям:

```
@NamedQuery(name = "studentsInCourse",
  query = "select st " +  "from Student st, " +  "in  (st.courses)  c "
```

или

```
@NamedQuery(name = "studentsInCourse",  query = "select st " +
 "from Student st  join st.courses  c "
```

с параметром:

```
@NamedQuery(name = "studentsInCourse",
  query = "select st " +
 "from Student st  in (st.courses) c " +
    "where c.name = :courseName"
```

Criteria API

- Criteria API - дополнение к JPQL.
- Позволяет динамически создавать типобезопасные JPA запросы.

https://www.objectdb.com/java/jpa/query/jpql/structure

Criteria API

Пример Criteria запроса:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Student> cq = cb.createQuery(Student.class);
Root<Student> student = cq.from(Student.class);
cq.select(student);
TypedQuery<Student> q = em.createQuery(cq);
List<Student> allStudents = q.getResultList();
```

Metamodel API

Управляемая сущность:

```
@Entity
public class Student {
    @Id
  protected Long id;
    protected String name;
    @OneToMany
    protected List<Course> courses;
    ...
}
```

Класс метамодели:

```
@Static Metamodel(Student.class)
public class Student _ {
    public static volatile SingularAttribute< Student,
Long> id;
    public static volatile SingularAttribute< Student,
String> name;
    public static volatile ListAttribute< Student, Course>
courses;
}
```

Для использования в Criteria queries необходимо сгенерировать или создать вручную статические классы метамодели;

Metamodel API

Запрос к связанным сущностям:

```
CriteriaQuery< Student > cq = cb.createQuery(Student .class);
Root<Student> student = cq.from(Student .class);
Join<Student, Course > course= student.join(Student_. courses);
```

Объекты **Path** (это может быть корневая сущность) используются объектом CriteriaQuery в SELECT и WHERE clauses. Метод Path.*get* используется для доступа к атрибутам сущности.

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Student> student = cq.from(Student.class);
cq.select(student.get(Student_.name));
```

Metamodel API

Метод CriteriaQuery.where ограничивает результаты запроса аналогично WHERE оператору в JPQL запросе:

```
CriteriaQuery<Student> cq = cb.createQuery(Student.class);
Root<Student> student = cq.from(Student.class);
Date someDate = new Date(...);
cq.where(cb.gt(student.get(Student_.birthday), date));
```

Метод CriteriaQuery.groupBy разбивает результаты запроса по группам.
Метод CriteriaQuery.orderBy упорядочивает результаты запроса.

Metamodel API

Для подготовки запроса к выполнению создается объект TypedQuery<T>

```
CriteriaQuery<Student> cq = cb.createQuery(Student.class);
...
TypedQuery<Student> q = em.createQuery(cq);
Student result = q.getSingleResult();
```

```
CriteriaQuery<Student> cq = cb.createQuery(Student.class);
...
TypedQuery<Student> q = em.createQuery(cq);
List<Student> results = q.getResultList();
```

Управление параллельным доступом

Оптимистическое блокирование:
Перед тем, как подтвердить изменение данных , провайдер контролирует, что сущность не изменилась с тех пор как она была считана из БД другими транзакциями.

Пессимистическое блокирование:
Провайдер создает транзакцию, которая получает длительную блокировку на данные (до завершения транзакции). Другие транзакции не могут модифицировать или удалить данные пока блокировка не будет снята.

Оптимистическое блокирование

Аннотация jakarta.persistence.Version используется, чтобы пометить сохраняемое поле как версионный атрибут сущности.
Провайдер читает и обновляет это свойство, когда экземпляр сущности модифицируется во время транзакции.

```
@Version
protected int version;
```

Режимы блокирования (Lock Modes)

OPTIMISTIC - Получить optimistic read lock для сущностей с атрибутом @Version.
OPTIMISTIC_FORCE_INCREMENT - Получить optimistic read lock для сущностей с атрибутом @Version и прирастить значение атрибута.
PESSIMISTIC_READ – Немедленно получить long-term read lock для предотвращения модификации и удаления данных. Другие транзакции могут читать данные.
PESSIMISTIC_WRITE - Немедленно получить long-term write lock для предотвращения чтения, модификации и удаления данных.
PESSIMISTIC_FORCE_INCREMENT - Немедленно получить long-term lock для предотвращения чтения, модификации и удаления данных и прирастить значение атрибута @Version.

Режимы блокирования (Lock Modes)

Способы указания режима блокирования:

```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);

EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
LockModeType.PESSIMISTIC_WRITE);

EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

Режимы блокирования (Lock Modes)

Способы указания режима блокирования (продолжение):

```
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

Информационные ресурсы
Организационные вопросы