Java для молодых специалистов Банка ВТБ

Урок 12

Hibernate. Часть 2

Аннотации

Давайте рассмотрим список аннотаций, применяемые в Hibernate.

1. Каждый класс хранимой сущности должен иметь аннотацию **@Entity**;

2. Аннотации **@Table** позволяет задать имя таблицы, в которую будут отображаться объекты данной сущности, и настроить индексы;

```
@Table(name = "demo_annotated", indexes = {
        @Index(name = "name_idx", columnList = "name"),
        @Index(name = "id_name_idx", columnList = "id, name"),
        @Index(name = "unique_name_idx", columnList = "name", unique = true)
})
```

3. Аннотации **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany** обозначают связи между сущностями;

4. **@Column** отвечает за настройки столбца в таблице. С помощью параметра **name** указывается имя столбца в которое будет записано значение размеченного поля. Для “ручного” формирования запроса с помощью которого будет построен столбец можно воспользоваться параметром **columnDefinition**. Здесь же можно указать ограничение NOT NULL (**nullable** = false). Чтобы запретить изменение значение какого-либо столбца параметр **updatable** переводится в false.

```
@Column(name = "manual_def_str", columnDefinition = "VARCHAR(50) NOT NULL
UNIQUE CHECK (NOT substring(lower(manual_def_str), 0, 5) = 'admin')")
String manualDefinedString;

@Column(name = "short_str", nullable = false, length = 10) // varchar(10)
String shortString;

@Column(name = "created_at", updatable = false)
LocalDateTime createdAt;
```

5. Для автогенерации времени создания объекта в базе данных и времени его обновления используются аннотации **@CreationTimestamp** и **@UpdateTimestamp**;

6. При реализации связи **@ManyToOne**, для создания внешнего ключа указывается следующая форма аннотации **@JoinColumn**

```
@ManyToOne
@JoinColumn(
        name = "product_id",
        nullable = false,
        foreignKey = @ForeignKey(name = "FK_PRODUCT_ID")
)
Product product;
```

6. Любой класс хранимой сущности обязан иметь идентифицирующий атрибут - поле, помеченное

аннотацией **@Id**. Аннотация **@GeneratedValue** по умолчанию указывает что генерация будет выполнятся автоматически, и позволяет указать способ генерации.

```
public class Product {
    @Id
    @GeneratedValue
    @Column(name = "id")
    Long id;
```

8. **@Version** поле используется для версионирования, о котором речь пойдет ниже.

Оптимистическое управление параллельным доступом

Оптимистическое управление параллельным доступом подходит для случаев, когда изменения вносятся редко и в рамках одной транзакции допустимо позднее обнаружение конфликтов. Для оптимистического управления необходимо включить версионирование. При таком подходе будет “побеждать” первая подтвержденная транзакция.

Для добавления версионирования достаточно добавить в классы, помеченные аннотацией @Entity поле с аннотацией @Version.

```
@Entity
@Table(name = "items")
public class Item {
    @Id
    @GeneratedValue
    @Column(name = "id")
    Long id;

    @Column(name = "val")
    int val;

    @Column(name = "junkField")
    @OptimisticLock(excluded = true)
    int junkField;

    @Version
    long version;

    public void setVal(int val) {
        this.val = val;
    }

    public long getVersion() {
        return version;
    }
```

```
    public Item() {
    }

    public Item(int val) {
        this.val = val;
    }

    @Override
    public String toString() {
        return String.format("Item [ id = %d, val = %d, version = %d ]", id,
val, version);
    }
}
```

В таком случае, каждому экземпляру данной сущности будет присваиваться версия, которая отображается на отдельный столбец в таблицы базы данных. Для поля version можно добавить геттер, но ни в коем случае не должно быть сеттера, так как изменением этого поля занимается сам Hibernate. По сути, версия - это просто счетчик. Давайте посмотрим на следующий пример:

```
// ... тут стандартный запуск SessionFactory
session = factory.getCurrentSession();
session.beginTransaction();
Item item = session.find(Item.class, 1L);
System.out.println(item.getVersion()); // <- 1
item.setVal(20);
session.flush();
System.out.println(item.getVersion()); // <- 2
item.setVal(30);
session.flush();
System.out.println(item.getVersion()); // <- 3
session.getTransaction().rollback();

session = factory.getCurrentSession();
session.beginTransaction();
item = session.find(Item.class, 1L);
System.out.println(item.getVersion()); // <- 1
session.getTransaction().commit();
session.close();
// … а тут завершение работы
```

При внесении изменений в состояние item, Hibernate накапливает эти изменения, но не посылает запросы в базу данных. Выполнение session.flush() выталкивает контекст хранения, выполняя запросы в БД. В результате, после каждого flush() версия растет. Если выполняется rollback(), то само собой изменения версии не подтверждаются.

Для проверки работы такого варианта оптимистической блокировки, можно воспользоваться следующим кодом.

```
// ...
```

```
new Thread(() -> {
   System.out.println("Thread #1 started");
   Session session = factory.getCurrentSession();
   session.beginTransaction();
   Item item = session.get(Item.class, 1L); // <- version = 1
   item.setVal(100);
   uncheckableSleep(1000);
   session.save(item);
   session.getTransaction().commit(); // version увеличивается до 2
   System.out.println("Thread #1 committed");
   if (session != null) {
       session.close();
   }
   countDownLatch.countDown();
}).start();

new Thread(() -> {
   System.out.println("Thread #2 started");
   Session session = factory.getCurrentSession();
   session.beginTransaction();
   Item item = session.get(Item.class, 1L); // <- version = 1
   item.setVal(200);
   uncheckableSleep(3000);
   try {
       session.save(item);
       session.getTransaction().commit(); // в момент подтверждения транзакции
во втором потоке производится сравнение версии при старте транзакции (1) и
текущим значением версии (2)
       System.out.println("Thread #2 committed");
   } catch (OptimisticLockException e) {
       session.getTransaction().rollback();
       System.out.println("Thread #2 rollback");
       e.printStackTrace();
   }
   if (session != null) {
       session.close();
   }
   countDownLatch.countDown();
}).start();
try {
   countDownLatch.await();
} catch (InterruptedException e) {
   e.printStackTrace();
}
// ...
```

** unckeckableSleep(int ms) метод, выполняющий Thread.sleep(), с перехватом InterruptedException.*

Два потока параллельно пытаются изменить состояние item. При старте, в каждой транзакции выполняется “запоминание версии объекта”, которая равна 1. Транзакция может быть успешно

завершена только в том случае, если версия объекта на момент начала и подтверждения транзакции совпадают.

Из кода видно, что первый поток подтвердит транзакцию раньше второго, при этом будет произведено сравнение версий, и поскольку **1 == 1**, транзакция удачно завершится и версия увеличится на 1. Через пару секунд, второй поток попытается также завершить транзакцию, но версии будут отличаться **1 != 2**, и в этом случае будет сгенерировано исключение OptimisticLockException, после перехвата которого выполняется rollback().

Если изменение какого-либо поля не должно влиять на версию объекта, то такое поле можно пометить как `@OptimisticLock(excluded = true)`, в примере кода из начала пункта, такой аннотацией было помечено поле int junkField.

Пессимистические блокировки

Давайте рассмотрим случай пессимистической блокировки.

```
// ...
session = factory.getCurrentSession();
session.beginTransaction();
int sumValue = 0;
List<Item> items = session.createQuery("SELECT i FROM Item i;", Item.class)
        .setLockMode(LockModeType.PESSIMISTIC_READ)
        .getResultList();
for (Item o : items) {
   sumValue += o.getVal();
}
session.getTransaction().commit();
// ...
```

Допустим мы хотим просуммировать значения всех item'ов в нашей таблице, и сделать это надо именно на стороне нашего приложения, а не базы данных. В таком случае, при получении списка объектов из базы данных, мы не можем давать другим транзакциям изменять значения этих элементов. Для этого может быть установлена пессимистическая блокировка с помощью метода setLockMode(). При выборе в качестве аргумента LockModeType.PESSIMISTIC_READ, полученные записи будут доступны другим транзакциям только для чтения, такой режим аналогичен блокировке PosetgreSQL "FOR SHARE". Если выбрать LockModeType.PESSIMISTIC_WRITE, то строки заблокируются как для чтения, так и для записи, что в PostgreSQL аналогично "FOR UPDATE". Блокировки будут сняты только по завершению текущей транзакции.

Практическое задание

1. Создайте таблицу items (id serial, val int, ...), добавьте в нее 40 строк со значением 0;

2. Запустите 8 параллельных потоков, в каждом из которых работает цикл, выбирающий случайную строку в таблице и увеличивающий val этой строки на 1. Внутри транзакции необходимо сделать Thread.sleep(5). Каждый поток должен сделать по 20.000 таких изменений;

3. По завершению работы всех потоков проверить, что сумма всех val равна соответственно 160.000;

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.