13 способов улучшить производительность JPA и Hibernate

[26/09/2016](http://akorsa.ru/2016/09/13-sposobov-uluchshit-proizvoditelnost-jpa-i-hibernate/)  [admin](http://akorsa.ru/author/admin/)  [Hibernate](http://akorsa.ru/category/hibernate/)

Введение

Hibernate является самой популярной имплементацией [JPA](https://en.wikipedia.org/wiki/Java_Persistence_API). Эта популярность принесла несколько преимуществ всем пользователям. Уже написано множество статей в блогах, вопросов и ответов на популярных форумах и успешных советов по улучшению работы Hibernate. В этой статье я хочу суммировать некоторые лучшие практики для JPA и Hibernate, которые помогут Вам избежать распространненых ошибок и создать лучшее приложение.

Итак, поехали!

Лучшие практики

**1.Используйте проецирования, которые подходят под Ваш случай**

Когда Вы пишите выражение SQL SELECT, Вы явно задаете в выборке столбцы, которые необходимо для данного случая. И это должно работать также и в Hibernate. К сожалению, большинство разработчиков только выбирают entity из БД, не обращая внимания на то, подходит ли такая выборка им или нет.

JPA и Hibernate поддерживают больше типов проецирования, чем просто сущности. Существует 3 различных типа проецирования, и каждая имеет свои преимущества и недостатки.

**1.1 Сущности**

Сущности — самая распространенная проекция. Вы должны использовать ее, когда хотите **получить все атрибуты сущности и для операции обновления или удаления**, которые оказывают влияние только на малое количество сущностей.

|  |  |
| --- | --- |
| 1 | em.find(Author.class, 1L); |

**1.2 POJO**

Проецирование POJO схоже с проецированием сущности, но оно позволяет создавать представление записи в БД, которое специфично для данного случая. Это особенно полезно, если Вам**необходимо получить только небольшой набор атрибутов сущности** или если Вам нужно получить атрибуты из нескольких связанных сущностей.

|  |  |
| --- | --- |
| 1  2  3 | List<BookPublisherValue> bookPublisherValues = em.createQuery(    “SELECT new org.thoughts.on.java.model.BookPublisherValue(b.title, b.publisher.name) FROM Book b”,    BookPublisherValue.class).getResultList(); |

**1.3 Скалярные значения**

Скалярные значения не очень популярный тип проецирования, потому что он представляет значения как Object[]. Вы должны использовать этот тип, только**в случае выборки малого количества атрибутов и напрямую обработать их в Вашей бизнес логике**. Проецирование POJO наиболее лучший способ, когда Вам нужно выбрать большое количество атрибутов или если Вы хотите передать результат запроса в разные подсистемы.

|  |  |
| --- | --- |
| 1  2 | List<Object[]> authorNames = em.createQuery(    “SELECT a.firstName, a.lastName FROM Author a”).getResultList(); |

2. Используйте подходящий тип запроса

JPA и Hibernate предоставляют множество явных и неявных опций для создания запроса. Ни один из них не подходит для всех случаев, и Вы должны быть уверены, что выбрали тот самый, который лучше всего подходит.

**2.1 EntityManager.find()**

Метод EntityManager.find() не только самый легкий способ получить сущность по ее первичному ключу, но также **предоставляет преимущества в производительности и безопасности**:

* Hibernate проверяет кэши первого и второго уровня, прежде чем выполнить SQL запрос для чтения сущности из БД
* Hibernate генерирует запрос и вставляет первичный ключ как параметр для избегания уязвимостей SQL инъекций.

|  |  |
| --- | --- |
| 1 | em.find(Author.class, 1L); |

**2.2 JPQL**

[Java Persistence Query Language](http://docs.oracle.com/cd/E12839_01/apirefs.1111/e13946/ejb3_langref.html)определен стандартом JPA и очень похож на SQL. Он оперирует сущностями и их связями вместо таблиц БД. Вы **можете использовать его для создания запросов низкой и средней сложности**.

|  |  |
| --- | --- |
| 1  2  3 | TypedQuery<Author> q = em.createQuery(    “SELECT a FROM Author a JOIN a.books b WHERE b.title = :title”,    Author.class); |

**2.3 Criteria API**

[Criteria API](http://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html)предлагает Вам удобный для использования API для динамически заданных запросов в рантайме. Вы должны использовать этот способ, если **структура запроса зависит от ввода информации пользователем**.

Вы можете увидеть пример для такого запроса в нижеприведенном коде. Если атрибут заголовка введенного объекта содержит не пустую String, сущность Book соединится с сущностью Author и вернется в результат, в случае если поле title будет равно введенному значению.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CriteriaBuilder cb = em.getCriteriaBuilder();  CriteriaQuery<Author> q = cb.createQuery(Author.class);  Root<Author> author = q.from(Author.class);  q.select(author);    if (!input.getTitle().isEmpty()) {    SetJoin<Author, Book> book = author.join(Author\_.books);    q.where(cb.equal(book.get(Book\_.title), input.getTitle()));  } |

**2.4 Нативные запросы**

Нативные запросы дают Вам шанс написать и выполнить нативное SQL выражение. Такой способ часто является лучшим подходом для очень сложных запросов и, если Вы хотите использовать премущества СУБД, наподобие типа данных  PostgreSQLs JSONB.

|  |  |
| --- | --- |
| 1  2  3 | MyEntity e = (MyEntity) em.createNativeQuery(    “SELECT \* FROM myentity e WHERE e.jsonproperty->’longProp’ = ‘456’“,    MyEntity.class).getSingleResult(); |

3. Используйте привязанные (bind) параметры

Вы должны использовать привязку параметров к запросу вместо добавления значений напрямую к строке запроса. Это дает следующие преимущества:

* Вы можете не волноваться по поводу SQL инъекций
* Hibernate маппирует Ваши параметры запросы на корректный тип
* Hibernate может делать внутренние оптимизации для обеспечения лучшей производительности.

JPQL, Criteria API и нативные SQL запросы используют один и тот же интерфейс [Query](http://docs.oracle.com/javaee/7/api/javax/persistence/Query.html), который предоставляет метод [setParameter](http://docs.oracle.com/javaee/7/api/javax/persistence/Query.html" \l "setParameter-int-java.lang.Object-) для позиционного и именованного привязывания параметров. Hibernate поддерживает привязывание параметров по имени для нативных запросов, но это не определено JPA спецификацией. Поэтому я рекомендую **использовать только позиционные параметры для нативных запросов**. Они отмечены как «?» и их нумерация начинается с 1.

|  |  |
| --- | --- |
| 1  2  3 | Query q = em.createNativeQuery(“SELECT a.firstname, a.lastname FROM Author a WHERE a.id = ?”);  q.setParameter(1, 1);  Object[] author = (Object[]) q.getSingleResult(); |

Hibernate и JPA поддерживают привязывание параметров по имени для JPQL и Criteria API. Это позволяет задать имя для каждого параметра и передать значения параметров в метод setParameter(). Название параметра чувствительно к регистру и должно иметь префикс в виде «:» символа.

|  |  |
| --- | --- |
| 1  2  3 | Query q = em.createNativeQuery(“SELECT a.firstname, a.lastname FROM Author a WHERE a.id = :id”);  q.setParameter(“id”, 1);  Object[] author = (Object[]) q.getSingleResult(); |

4. Используйте static String для именованных запросов и имен параметров

Это маленькое улучшение, но оно намного больше облегчает работу с именованными запросами и их параметрами. Я предпочитаю задавать их как атрибуты сущности, но Вы можете также создать класс, который будет содержать все запросы и имена параметров.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @NamedQuery(name = Author.QUERY\_FIND\_BY\_LAST\_NAME,  query = “SELECT a FROM Author a WHERE a.lastName = :” + Author.PARAM\_LAST\_NAME)  @Entity  public class Author {      public static final String QUERY\_FIND\_BY\_LAST\_NAME = “Author.findByLastName”;    public static final String PARAM\_LAST\_NAME = “lastName”;      …    } |

Вы можете потом использовать эти строки для инициализации именованного запроса и установки параметра.

|  |  |
| --- | --- |
| 1  2  3 | Query q = em.createNamedQuery(Author.QUERY\_FIND\_BY\_LAST\_NAME);  q.setParameter(Author.PARAM\_LAST\_NAME, “Tolkien”);  List<Author> authors = q.getResultList(); |

5. Используйте JPA Metamodel при работе с Criteria API

Criteria API предоставляет удобный способ для определения запроса. Для этого необходимо сделать ссылки между сущностями и их атрибутами. Лучший способ сделать это использовать статическую [JPA Metamodel](http://docs.oracle.com/javaee/6/tutorial/doc/gjiup.html). Вы можете автоматически генерировать статический metamodel класс для каждой сущности во время билда. Этот класс содержит статические атрибуты для каждого атрибута сущности.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Generated(value = “org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor”)  @StaticMetamodel(Author.class)  public abstract class Author\_ {      public static volatile SingularAttribute<Author, String> firstName;    public static volatile SingularAttribute<Author, String> lastName;    public static volatile SetAttribute<Author, Book> books;    public static volatile SingularAttribute<Author, Long> id;    public static volatile SingularAttribute<Author, Integer> version;    } |

Вы можете использовать этот metamodel класс для ссылки на атрибуты сущности в запросе Criteria. Я использую его в пятой строке приведенного кода для ссылки на атрибут lastName Author сущности.

|  |  |
| --- | --- |
| 1  2  3  4  5 | CriteriaBuilder cb = em.getCriteriaBuilder();  CriteriaQuery<Author> q = cb.createQuery(Author.class);  Root<Author> author = q.from(Author.class);  q.select(author);  q.where(cb.equal(author.get(Author\_.lastName), lastName)); |

6. Используйте суррогатные ключи и позвольте Hibernate генерировать новые значения

Главным преимуществом суррогатного первичного ключа (или технического ID) это то, что он представляет собой одно простое число, а не комбинацию множества атрибутов как большинство естественных ключей. Все вовлеченные системы, главным образом Hibernate и СУБД, управляет ключами эффективно. Hibernate может также использовать существующие преимущества СУБД, такие как последовательности (sequences) или автоинкрементные столбцы для генерации уникальных значений для новых сущностей.

|  |  |
| --- | --- |
| 1  2  3  4 | @Id  @GeneratedValue  @Column(name = “id”, updatable = false, nullable = false)  private Long id; |

7. Указывайте естественные идентификаторы, там, где это необходимо

Вы должны указывать естественные идентификаторы, даже если Вы решили использовать суррогатный ключ в качестве первичного ключа. Естественный идентификатор, тем не менее, идентифицирует запись в БД и объект в реальном мире. Множество вариантов использования предполагает использование их, вместо искусственных, суррогатных ключей. Поэтому хорошей практикой является моделировать их как уникальные ключи в Вашей БД. Hibernate также предлагает Вам моделировать их как натуральный идентификатор сущности и обеспечивает дополнительным API для получения их из БД.

Единственная вещь, которую нужно сделать, это добавить к нужному атрибуту аннотацию [@NaturalId](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NaturalId.html).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Entity  public class Book {    @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column(name = “id”, updatable = false, nullable = false)  private Long id;    @NaturalId  private String isbn;    …  } |

8. Используйте SQL скрипты для создания схемы БД.

Hibernate может использовать информацию маппинга Ваших сущностей для генерации схемы БД, Это самый легкий вариант и вы можете увидеть множество примеров в интернете. Это может и ОК для маленьких тестовых приложений, но Вы не должны использовать его для бизнес приложений. **Схема БД имеет огромное влияние на производительность и размер Вашей БД**. Поэтому Вы должны сами спроектировать и оптимизировать схему БД под свои нужны и экспортировать ее как SQL скрипт. Вы можете запустить этот скрипт внешним инструментом, как [Flyway](https://flywaydb.org/) или можете использовать Hibernate для инициализации БД во время старта. Следующий сниппет кода показывает как настроить persistence.xml, который укажет Hibernate запускать create.sql скрипт для создания БД.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <?xml version=”1.0″ encoding=”UTF-8″ standalone=”yes”?>  <persistence xmlns=”<http://xmlns.jcp.org/xml/ns/persistence>” xmlns:xsi=”<http://www.w3.org/2001/XMLSchema-instance>” version=”2.1″ xsi:schemaLocation=”<http://xmlns.jcp.org/xml/ns/persistence> <http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd>”>    <persistence-unit name=”my-persistence-unit” transaction-type=”JTA”>      <description>My Persistence Unit</description>      <provider>org.hibernate.ejb.HibernatePersistence</provider>      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>      <exclude-unlisted-classes>false</exclude-unlisted-classes>        <properties>        <property name=”hibernate.dialect” value=”org.hibernate.dialect.PostgreSQLDialect”/>          <property name=”javax.persistence.schema-generation.scripts.action” value=”create”/>        <property name=”javax.persistence.schema-generation.scripts.create-target” value=”./create.sql”/>      </properties>    </persistence-unit>  </persistence> |

9. Логируйте и анализируйте все запросы в процессе разработки

Очень большое количество выполняемых запросов является самой главной причиной для проблем с производительностью Hibernate. Это часто вызвано проблемой n + 1 запросов, но это не является единственным способом для получения большего числа SQL выражений, чем ожидается.

**Hibernate скрывает все взаимодействия с БД под его API , и часто сложно угадать, сколько запросов будет выполнено в том или ином случае**. Лучший способ управлять этой проблемой это**логировать все SQL выражения в процессе** разработки и анализировать их до того как вы закончите этап разработки. Вы можете это сделать, настроив уровень логирования:

org.hibernate.SQL = DEBUG

10. Не используйте FetchType.EAGER

Стратегия фетчинга данных EAGER другая общая причина для проблем с производительностью Hibernate.  Стратегия указывает Hibernate инициализировать все связи сущности, когда он извлекает сущность из БД.

|  |  |
| --- | --- |
| 1  2 | @ManyToMany(mappedBy = “authors”, fetch = FetchType.EAGER)  private Set<Book> books = new HashSet<Book>(); |

Стратегия извлечения связанных сущностей из БД зависит от связи и заданного режима FetchMode. Но это не главная проблема. **Главная проблема это то, что Hibernate будет извлекать связанные сущности в независимости от того нужны ли или нет для конкретного случая**. Это создает лишнюю нагрузку, которая замедляет работу приложения и часто является причиной проблем с производительностью. **Вы должны использовать режим FetchType.LAZY** и извлекать связанные сущности только те, которые необходимо в данном случае.

|  |  |
| --- | --- |
| 1  2 | @ManyToMany(mappedBy = “authors”, fetch = FetchType.LAZY)  private Set<Book> books = new HashSet<Book>(); |

11. Инициализируй нужные lazy связи с помощью исходного запроса

Как я объснил ранее, FetchType.LAZY указывает Hibernate извлекать только необходимые связанные сущности. Это помогает Вам избежать определенных проблем с производительностью и LazyInitializationException  и проблему выборки n + 1, которая возникает, когда Hibernate выполняет дополнительный запрос для инициализации связи каждой из выбранных n сущностей.

Лучший способ борьбы избежать вышеуказанные проблемы это извлекать сущность вместе со связями, которые нужны Вам в данный момент. Опцией, которая делает это,  **является использование JPQL запроса с выражением JOIN FETCH**.

|  |  |
| --- | --- |
| 1  2  3 | List<Author> authors = em.createQuery(  “SELECT DISTINCT a FROM Author a JOIN FETCH a.books b”,  Author.class).getResultList(); |

12. Избегайте использования каскадного удаления для больших связей

Большинство разработчиков (в том числе и я) нервничают, когда они видят CascadeType.REMOVE для связи. Это указывает Hibernate также удалить связанные сущности, при удалении  самой сущности. Всегда присутствует страх, что связанные сущности также используют каскадное удаление для некоторых из их связей и Hibernate может удалить больше записей в БД, чем подразумевалось. **Каскадное удаление делает сложным понимание, что на самом деле произойдет при удалении сущности**. И этого нужно избегать.

Если Вы внимательней взглянете на то, как Hibernate удаляет связанные сущности, Вы найдете другую причину не использовать каскадное удаление. Hibernate выполняет 2 SQL выражения для каждой связанной сущности: первое выражение SELECT извлекает сущность из БД и второе выражение DELETE  удаляет ее. Это может быть нормально в случае одной или двух связанных сущностей, но создает проблемы, если их количество большое.

13. Используйте @Immutable где возможно

Hibernate регулярно выполняет «грязные» проверки всех сущностей, ассоциированных с текущим PersistenceContext для обнаружения изменений. Это нужная вещь для всех изменяемых сущностей. Но не все сущности должны быть изменяемы. Сущности могут также отображены как read-only view БД или таблицы. Выполнение проверок над этими сущностями является излишеством, которого Вы должны избегать.

Вы можете сделать это с помощью аннотации @Immutable. Hibernate будет игнорировать сущности для «грязных» проверок и не запишет никаких изменений в БД.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Entity  @Immutable  public class BookView {    …    } |

Заключение

Я представил широкий ряд лучших практик, которые помогут Вам имплементировать приложение быстрее и избежать общеизвестных ошибок с производительностью. Я сам следую им для избегания проблем и они мне очень помогли.