**HQL и JPQL**

Hibernate Query Language (HQL) и Java Persistence Query Language (JPQL) оба являются объектно-ориентированными языками запросов, схожими по природе с SQL. JPQL - это подмножество HQL. JPQL-запрос всегда является допустимым HQL-запросом, однако обратное неверно.

И HQL, и JPQL не являются безопасными для типов способами выполнения операций запроса. Запросы по критериям предлагают безопасный для запросов подход к типу запросов. См. [Критерии](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/query/criteria/Criteria.html#criteria) для получения дополнительной информации.

!!! Одновременно JPQL c HQL в одном файле использовать нельзя т.к. класс Query, используемый для запросов:

- для HQL использует “import org.hibernate.query.Query”;

- для JPQL использует “import javax.persistence.Query”,

а "org.hibernate" перебивает "javax.persistence" !!!

**в JPQL, я так понял, что создание/удаление/изменение информации в БД требует begin + commit. А выборка – нет.**

**Оглавление**

Примеры доменной модели

PA Query API (JPQL)

Получение результатов в JPQL

Hibernate Query API (HQL)

Получение результатов в HQL

Прокрутка запросов

SELECT

UPDATE

DELETE

INSERT

FROM

Идентификационные переменные (алиасы)

Ссылки на корневые объекты

Явные объединения

Неявные объединения (выражения пути)

DISTINCT

Ссылки коллекций на свои коллекции ()

Особый случай - квалифицированные выражения пути

Полиморфизм

Выражения

Идентификационная переменная

Выражения пути

литералы

арифметика

Конкатенация (операция)

Агрегатные функции

Скалярные функции

Стандартизированные функции JPQL

HQL-функции

Нестандартизированные функции

Выражения, связанные с коллекцией

TYPE

CASE / COALESCE выражения (аналог switch/case/default в java)

Выражения NULLIF

SELECT

Предикаты: Реляционные сравнения, ALL, NULL, LIKE, BETWEEN, IN, Exists, IS [NOT] EMPTY, [NOT] MEMBER [OF], NOT, AND, Оператор предиката ИЛИ

WHERE/HAVING

GROUP BY

ORDER BY (сортировать по)

Режим “Только для чтения” **(**для JPQLsetHint или [@QueryHint](http://docs.oracle.com/javaee/7/api/javax/persistence/QueryHint.html)**,** для HQL еще иsetReadOnly**)**

**Примеры доменной модели**

Чтобы лучше понять дальнейшие примеры HQL и JPQL, пришло время познакомиться с сущностями модели предметной области, которые используются во всех функциях примеров в этой главе.

Пример 1. Примеры доменной модели

@NamedQueries({

@NamedQuery( //именованный запрос, который каждый раз будет передаваться по сети

name = "get\_person\_by\_name",

query = "select p from Person p where name = :name"

)

,

@NamedQuery(

name = "get\_read\_only\_person\_by\_name",

query = "select p from Person p where name = :name",

hints = {

@QueryHint( //для установления разных режимов работы запроса, в данном случае сущность полученная в результате запроса будет доступна только для чтения "readOnly"

name = "org.hibernate.readOnly",

value = "true"

)

}

)

})

@NamedStoredProcedureQueries( //именованная процедура которая будет лежать и использоваться

//только на самом сервере

@NamedStoredProcedureQuery(

name = "sp\_person\_phones",

procedureName = "sp\_person\_phones",

parameters = {

@StoredProcedureParameter(

name = "personId",

type = Long.class,

mode = ParameterMode.IN

),

@StoredProcedureParameter(

name = "personPhones",

type = Class.class,

mode = ParameterMode.REF\_CURSOR

)

}

)

)

@Entity

@Inheritance(strategy = InheritanceType.JOINED) //определяет тип построения таблиц при

//последующем наследования этой сущности

public class **Payment** {

@Id

@GeneratedValue

private Long id;

private BigDecimal amount;

private boolean completed;

@ManyToOne

private Person person;

//Getters and setters are omitted for brevity

}

@Entity

public class **CreditCardPayment** extends Payment {

}

@Entity

public class **WireTransferPayment** extends Payment {

}

@Entity

public class **Person** {

@Id

@GeneratedValue

private Long id;

private String name;

private String nickName;

private String address;

@Temporal(TemporalType.TIMESTAMP ) //для указания, что это поле будет

//java.util.Date или java.util.Calendar

private Date createdOn;

@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)

@OrderColumn(name = "order\_id")

private List<Phone> phoneS = new ArrayList<>();

@ElementCollection // использутся для создания/загрузки коллекции *(не сущности), которая*

*//будет сохранена в отдельной таблице с ключами для связки с основной таблицей*

@MapKeyEnumerated(EnumType.STRING) //говорит, что тип ключа будет Java Enum

private Map<AddressType, String> addresses = new HashMap<>();

@Version //для уставки оптимистической блокировки всех полей сущности

private int version;

//Getters and setters are omitted for brevity

}

public enum AddressType {

HOME,

OFFICE

}

@Entity

@Table(name = "phone\_call")

public class **Call** {

@Id

@GeneratedValue

private Long id;

@ManyToOne

private Phone **phone**;

@Column(name = "call\_timestamp")

private Date timestamp;

private int duration;

//Getters and setters are omitted for brevity

}

@Entity

public class **Phone** {

@Id

private Long id;

@ManyToOne(fetch = FetchType.LAZY)

private Person person;

@Column(name = "phone\_number")

private String number;

@Enumerated(EnumType.STRING)

@Column(name = "phone\_type")

private PhoneType type;

@OneToMany(mappedBy ="**phone**", cascade =CascadeType.ALL, orphanRemoval =true) //”o-

//phanRemoval =true” указывает, что все объекты “Phone”, которые не имеют ссылки на Call, должны быть удалены из БД.

private List<Call> calls = new ArrayList<>( );

@OneToMany(mappedBy = "**phone**")

@MapKey(name = "timestamp")// мы не просто получаем из связанной сущности все ее

//переменные, а сразу группируем их по переменной "timestamp", используемой в виде ключа в Map.

@MapKeyTemporal(TemporalType.TIMESTAMP )//говорим, что тип ключа будет java.util.Date

// или java.util.Calendar.

private Map<Date, Call> callHistory = new HashMap<>();

@ElementCollection

private List<Date> repairTimestamps = new ArrayList<>( );

//Getters and setters are omitted for brevity

}

public enum PhoneType {

LAND\_LINE,

MOBILE;

}

@Entity

public class **Partner** {

@Id

@GeneratedValue

private Long id;

private String name;

@Version

private int version;

//Getters and setters are omitted for brevity

}

**JPA Query API (JPQL)**

В JPA запрос представлен javax.persistence.Query или javax.persistence.TypedQuery полученый из EntityManager. Чтобы создать встроенный Query или TypedQuery, вам нужно использовать EntityManager#createQuery метод. Для именованных запросов нужен метод EntityManager#createNamedQuery.

TypedQuery используется если запрос предполагается использовать для извлечения сущностей, при этом вызывается createQuery() с передачей типа результата.

TypedQuery дает вам возможность указать тип объекта при создании запроса, и следовательно, любая последующая операция не требует явного приведения к предполагаемому типу. В то время как обычный QueryAPI не возвращает точный тип объекта, который вы ожидаете и который вам нужно преобразовать.

!!! А вот HQL сам понимает возвращаемый тип объекта !!!

Пример 2. Получение JPA Query или TypedQuery ссылки

EntityManagerFactory emf = Persistence.createEntityManagerFactory(“ru.javarush”); //где “ru.javarush” это имя из <persistence-unit name="ru.javarush" transaction-type="RESOURCE\_LOCAL"> из настроечного файла “persistence.xml который подходит для любой реализации JPA, а не только для Hibernate”

EntityManager entityManager (или “em”) = emf.createEntityManager();

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name"

);

TypedQuery<Person> typedQuery = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name", Person.class

);

entityManager.close();

emf.close();

Пример 3. Получение JPA Query или TypedQuery ссылки для именованного запроса

@NamedQuery(

name = "get\_person\_by\_name",

query = "select p from Person p where name = :name"

)

Query query = entityManager.createNamedQuery( "get\_person\_by\_name" );

TypedQuery<Person> typedQuery = entityManager.createNamedQuery(

"get\_person\_by\_name", Person.class

);

Hibernate предлагает специальную [@NamedQuery](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NamedQuery.html) аннотацию, которая предоставляет способы настройки различных функций запросов, таких как режим сброса, кэшируемость, интервал ожидания.

Пример 4. Получение Hibernate Query или TypedQuery ссылки для именованного запроса

@NamedQueries({

@NamedQuery(

name = "get\_phone\_by\_number",

query = "select p " +

"from Phone p " +

"where p.number = :number",

timeout = 1,

readOnly = true

)

})

Phone phone = entityManager

.createNamedQuery( "get\_phone\_by\_number", Phone.class )

.setParameter( "number", "123-456-7890" )

.getSingleResult();

Также Query интерфейс можно использовать для управления выполнением запроса. Например, мы можем указать время ожидания выполнения или контролировать кэширование.

Пример 5. Основное Query использование JPA

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

// timeout - in milliseconds

.setHint( "javax.persistence.query.timeout", 2000 )

// очищать только во время коммита

.setFlushMode( FlushModeType.COMMIT );

Для получения полной информации см. Query [Javadocs](http://docs.oracle.com/javaee/7/api/javax/persistence/Query.html) . Многие параметры, управляющие выполнением запроса, определены как подсказки. JPA определяет некоторые стандартные подсказки (например, время ожидания в примере), но большинство из них зависят от поставщика. **Использование определенных подсказок поставщика в некоторой степени ограничивает переносимость ваших приложений.**

javax.**persistence**.query.**timeout** - определяет время ожидания запроса в миллисекундах.

javax.**persistence**.**fetchgraph** - определяет *выборку* EntityGraph. Атрибуты, явно указанные как AttributeNodes, обрабатываются как FetchType.EAGER(посредством выборки соединения или последующего выбора). Для получения дополнительной информации см. Обсуждения EntityGraph в [Fetching](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/fetching/Fetching.html#fetching) .

javax.**persistence**.**loadgraph** - определяет *загрузочный* граф EntityGraph. Атрибуты, явно указанные как AttributeNodes, обрабатываются как FetchType.EAGER(посредством выборки соединения или последующего выбора). Атрибуты, которые не указаны, обрабатываются как FetchType.LAZY или FetchType.EAGER в  зависимости от определения атрибута в метаданных. Для получения дополнительной информации см. Обсуждения EntityGraph в [Fetching](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/fetching/Fetching.html#fetching) .

org.**hibernate**.**cacheMode** - определяет CacheModeдля использования. .

org.**hibernate**.**cacheable** - определяет, является ли запрос кэшируемым. истина / ложь.

org.**hibernate**.**cacheRegion** - для запросов, которые кешируются, определяет конкретную область кеша для использования.

org.**hibernate**.**comment** - определяет комментарий для применения к сгенерированному SQL.

org.**hibernate**.**fetchSize** - определяет размер выборки JDBC для использования.

org.**hibernate**.**flushMode** - определяет Hibernate-специфичный FlushMode для использования. Если возможно, используйте javax.persistence.Query#setFlushMode вместо этого.

org.**hibernate**.**readOnly** - определяет, что сущности и коллекции, загруженные этим запросом, должны быть помечены как доступные только для чтения.

Последнее, что должно произойти перед выполнением запроса, - это связать значения для любых определенных параметров. JPA определяет упрощенный набор методов привязки параметров. По сути, он поддерживает установку значения параметра (по имени / позиции) и специализированную форму для Calendar/ Date типов, дополнительно принимающих  TemporalType.

Пример 6. Привязка параметра имени JPA “ **:**\*\*\* ”

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" );

// Для общих типов временных полей (например, `java.util.Date`,` java.util.Calendar`)

// нам также нужно предоставить связанный `TemporalType`

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.createdOn > :timestamp" )

.setParameter( "timestamp", timestamp, TemporalType.DATE );

JPQL стиле позиционные параметры объявляются с помощью вопросительного знака , за которым следует порядковый номер - ?1, ?2. Порядковые номера начинаются с 1. Как и с именованными параметрами, позиционные параметры также могут появляться в запросе несколько раз.

Пример 7. Связывание позиционного параметра JPA “ **?1, ?2 ?- - -** “

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like ?1" )

.setParameter( 1, "J%" );

|  |  |
| --- | --- |
|  | Рекомендуется не смешивать формы в данном запросе. |

**Получение результатов в JPQL**

С точки зрения выполнения, JPA Query предлагает 2 различных метода для получения набора результатов:

* Query#**getResultList()** - выполняет запрос выбора и возвращает список результатов.
* Query#**getSingleResult()**- выполняет запрос выбора и возвращает один результат. Если было более одного результата, выдается исключение.

Пример 8. getResultList(),если в таблице больше 1 столбца

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.getResultList();

Пример 9. JPA getSingleResult(),если в таблице 1 столбец

Person person = (Person) entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.getSingleResult();

**Hibernate Query API (HQL)**

В Hibernate HQL-запрос представлен как org.hibernate.query.Query полученный из a Session. Если HQL является именованным запросом, Session#getNamedQuery будет использоваться; иначе нужно Session#createQuery.

Пример 10. Получение гибернации Query

Session session = sessionFactory.openSession();

session.beginTransaction();

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name"

);

session.getTransaction().commit();

session.close();

Пример 11. Получение Queryссылки Hibernate для именованного запроса

org.hibernate.query.Query query = session.getNamedQuery( "get\_person\_by\_name" );

|  |  |
| --- | --- |
|  | Мало того, что синтаксис JPQL был в значительной степени вдохновлен HQL, но многие из API-интерфейсов JPA были также вдохновлены Hibernate. Два Query контракта очень похожи. |

Также интерфейс Query можно использовать для управления выполнением запроса. Например, мы можем указать время ожидания выполнения или контролировать кэширование.

Пример 12. Основное использование запроса - Hibernate

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

// timeout - in seconds

.setTimeout( 2 )

// писать в L2 кеши, но не читать из них

.setCacheMode( CacheMode.REFRESH )

// assuming query cache was enabled for the SessionFactory

//при условии, что кеш запросов был включен для SessionFactory

.setCacheable( true )

// добавить комментарий к сгенерированному SQL, если он включен через свойство конфигурации hibernate.use\_sql\_comments

.setComment( "+ INDEX(p idx\_person\_name)" );

Для получения полной информации см. [Запрос](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Query.html) Javadocs.

|  |  |
| --- | --- |
|  | Подсказки запросов здесь являются подсказками запросов к базе данных. Они добавляются непосредственно в сгенерированный SQL согласно Dialect#getQueryHintString.  С другой стороны, понятие JPA о подсказках к запросу относится к подсказкам, нацеленным на провайдера (Hibernate). Поэтому, даже если их называют одинаковыми, имейте в виду, что у них совсем другое предназначение. Также имейте в виду, что **подсказки запросов Hibernate обычно делают приложение непереносимым между базами данных, если только код, добавляющий их, сначала не проверяет диалект.** |

Промывка подробно описана в [разделе Промывка](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/flushing/Flushing.html#flushing) . Блокировка подробно описана в [разделе Блокировка](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/locking/Locking.html#locking) . Концепция состояния «только для чтения» рассматривается в [контексте постоянства](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/pc/PersistenceContext.html#pc) .

Hibernate также позволяет приложению подключаться к процессу построения результатов запроса через org.hibernate.transform.ResultTransformer контракт. Посмотрите его [Javadocs,](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/transform/ResultTransformer.html) а также реализации, предоставленные Hibernate для получения дополнительной информации.

Последнее, что должно произойти, прежде чем мы сможем выполнить запрос, - это связать значения для любых параметров, определенных в запросе. Запрос определяет много перегруженных методов для этой цели. Наиболее общая форма принимает значение, а также Hibernate Type.

Пример 13. Привязка параметра имени Hibernate

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%", StringType.INSTANCE );

Обычно Hibernate понимает ожидаемый тип параметра с учетом его контекста в запросе. В предыдущем примере, поскольку мы используем параметр при LIKE сравнение со строковым атрибутом, Hibernate автоматически определит тип; поэтому вышесказанное можно упростить.

Пример 14. Привязка параметра имени Hibernate (предполагаемый тип)

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" );

Существуют также сокращенные формы для связывания общих типов, таких как строки, логические значения, целые числа и т. д.

Пример 15. Привязка параметра имени Hibernate (краткие формы)

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name " +

" and p.createdOn > :timestamp" )

.setParameter( "name", "J%" )

.setParameter( "timestamp", timestamp, TemporalType.TIMESTAMP);

Позиционные параметры в стиле HQL следуют синтаксису позиционных параметров JDBC. Они объявлены с использованием ? без следующего порядкового номера. Невозможно связать два таких позиционных параметра как «одинаковые», кроме привязки одного и того же значения к каждому.

Пример 16. Связывание позиционного параметра Hibernate

org.hibernate.query.Query query = session.createQuery(

"select p " +

"from Person p " +

"where p.name like ? " )

.setParameter( 0, "J%" );

|  |  |
| --- | --- |
|  | **Эта форма считается устаревшей и может быть удалена в ближайшее время**. |

**Получение результатов в HQL.**

С точки зрения исполнения, Hibernate предлагает 4 различных метода получения результатов. 2 наиболее часто используемых

* Query#**list** - выполняет запрос выбора и возвращает список результатов.
* Query#**uniqueResult**- выполняет запрос выбора и возвращает единственный результат. Если было более одного результата, выдается исключение.

Пример 17. list()Результат гибернации

List<Person> persons = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.list();

Также возможно извлечь единственный результат из Query.

Пример 18. Спящий режим uniqueResult()

Person person = (Person) session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.uniqueResult();

|  |  |
| --- | --- |
|  | Если уникальный результат используется часто, а атрибуты, на которых он основан, являются уникальными, вы можете рассмотреть возможность сопоставления естественного идентификатора и использования API загрузки естественного идентификатора. Посмотрите [Natural Ids](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/domain/natural_id.html#naturalid) для получения дополнительной информации по этой теме. |

**Прокрутка запросов**

Hibernate предлагает дополнительные специализированные методы для прокрутки запроса и обработки результатов с использованием серверного курсора.

Query#scroll работает в тандеме с JDBC-понятием прокрутки ResultSet.

Query#scroll метод перегружен:

* Затем основная форма принимает один аргумент типа, org.hibernate.ScrollMode который указывает тип прокрутки, которая будет использоваться. См. [Javadocs](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/ScrollMode.html) для деталей на каждом.
* Вторая форма не принимает аргументов и будет использовать ScrollMode обозначенный Dialect#defaultScrollMode.

Query#**scroll** **возвращает** org.hibernate.**ScrollableResults** который оборачивает базовый JDBC (прокручиваемый) ResultSet и предоставляет доступ к результатам. В отличие от типичного только вперед ResultSet, он ScrollableResults позволяет вам перемещаться по ResultSet в любом направлении.

Пример 19. Прокрутка через ResultSet содержащие сущности

try ( ScrollableResults scrollableResults = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.scroll()

) {

while(scrollableResults.next()) {

Person person = (Person) scrollableResults.get()[0];

process(person);

}

}

|  |  |
| --- | --- |
|  | Поскольку эта форма содержит ResultSet открытый JDBC , приложение должно указывать, когда оно будет выполнено ScrollableResults с  помощью вызова своего close()метода (как унаследовано, java.io.Closeable чтобы ScrollableResults работать с блоками [try-with-resources](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) ).  Если приложение не будет закрыто, Hibernate автоматически закроет базовые ресурсы (например, ResultSet и PreparedStatement), используемые внутри ScrollableResults к моменту окончания текущей транзакции (принятие или откат). Тем не менее, это хорошая практика, чтобы закрыть ScrollableResults явно. |
|  | Если вы планируете использовать Query#**scroll** выборки из коллекции, важно, чтобы ваш запрос явно упорядочил результаты так, чтобы результаты JDBC содержали связанные строки последовательно. |

Hibernate также поддерживает Query#**iterate**, который предназначен для загрузки сущностей, **когда известно, что загруженные записи уже сохранены в кэше второго уровня**. Идея повторения заключается в том, что в запросе SQL будут получены только совпадающие идентификаторы. Из них идентификаторы разрешаются путем поиска в кэш-памяти второго уровня. Если поиск по кэшу второго уровня завершится неудачно, к базе данных необходимо будет выполнить дополнительные запросы.

|  |  |
| --- | --- |
|  | Эта операция может выполнять значительно лучше для загрузки большого количества объектов, которые наверняка уже существуют в кэше второго уровня. В тех случаях, когда многие объекты не существуют в кэше второго уровня, эта операция почти наверняка будет выполняться хуже. |

Iterator Вернулся из Query#iterateфактически специально набранный Итератор: org.hibernate.engine.HibernateIterator. Он специализирован для предоставления close()метода (опять же, унаследованного от java.io.Closeable). Когда вы закончите с этим, Iterator вы должны закрыть его, либо приведя к HibernateIterator или Closeable, либо позвонив [Hibernate#close(java.util.Iterator)](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Hibernate.html#close-java.util.Iterator-).

Начиная с Stream версии 5.2, Hibernate предлагает поддержку для возврата ResultSet, который впоследствии может быть использован в stream() для преобразования базового уровня .

Внутренне stream()поведение ведет себя как  Query#scroll а основной результат поддерживается ScrollableResults.

Извлечение прогноза с использованием Query#stream метода может быть сделано следующим образом:

Пример 20. Спящий режим stream()с использованием типа результата проекции

try ( Stream<Object[]> persons = session.createQuery(

"select p.name, p.nickName " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.stream() ) {

persons

.map( row -> new PersonNames(

(String) row[0],

(String) row[1] ) )

.forEach( this::process );

}

При извлечении одного результата, такого как Person сущность, а не Stream<Object[]>, Hibernate выяснит фактический тип, поэтому результатом будет a Stream<Person>.

Пример 21. Спящий режим stream()с использованием типа результата объекта

try( Stream<Person> persons = session.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.stream() ) {

Map<Phone, List<Call>> callRegistry = persons

.flatMap( person -> person.getPhones().stream() )

.flatMap( phone -> phone.getCalls().stream() )

.collect( Collectors.groupingBy( Call::getPhone ) );

process(callRegistry);

}

|  |  |
| --- | --- |
|  | Как и в случае с ScrollableResults, вы всегда должны закрывать Hibernate Stream либо явно, либо используя блок [try-with-resources](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) . |

**Чувствительность к регистру**

За исключением имен классов и свойств Java, запросы не чувствительны к регистру. SeLeCT так  же, как sELEct и так же, как и SELECT, но есть org.hibernate.eg.FOO и org.hibernate.eg.Foo разные, как есть foo.barSet и foo.BARSET.

|  |  |
| --- | --- |
|  | Эта документация использует строчные ключевые слова как соглашение в примерах. |

**Типы выписок**

Оба HQL и JP позволяют SELECT, UPDATE и DELETE заявление должны быть выполнены. **HQL дополнительно допускает INSERT операторы в форме, аналогичной SQL INSERT FROM SELECT.**

|  |  |
| --- | --- |
|  | Следует внимательно следить за тем, когда выполняется оператор UPDATE или DELETE.  Следует соблюдать осторожность при выполнении массовых операций обновления или удаления, поскольку они могут привести к несоответствиям между базой данных и объектами в контексте активного постоянства. Как правило, массовые операции обновления и удаления должны выполняться только внутри транзакции в новом постоянном контексте или перед извлечением или доступом к объектам, состояние которых может быть затронуто такими операциями. |

**SELECT**

[BNF](https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form) для SELECT операторов в HQL является:

select\_statement :: =

[select\_clause]

from\_clause

[where\_clause]

[groupby\_clause]

[having\_clause]

[orderby\_clause]

Простейшее из возможных SELECT операторов HQL имеет вид:

List<Person> persons = session.createQuery(

"from Person" )

.list();

|  |  |
| --- | --- |
|  | Оператор select в JPQL точно такой же, как и для HQL, за исключением того, что JPQL требует a select\_clause, а HQL - нет.  List<Person> persons = entityManager.createQuery(  "select p " +  "from Person p", Person.class )  .getResultList();  **Несмотря на то, что HQL не требует присутствия a select\_clause, обычно рекомендуется включать его. Для простых запросов цель ясна, но в более сложные запросы это не всегда так. Кроме того приложения, заинтересованные в переносимости JPA должны реализовывать явное задание select.** |

**UPDATE**

BNF для UPDATE операторов одинаков в HQL и JPQL:

update\_statement ::=update\_clause [where\_clause]

update\_clause ::= UPDATE entity\_name [[AS] identification\_variable]SET update\_item {, update\_item}\*

update\_item ::= [identification\_variable.]{state\_field | single\_valued\_object\_field} = new\_value

new\_value ::= scalar\_expression | simple\_entity\_expression | NULL

UPDATE операторы по умолчанию не влияют version на timestamp значения атрибута или для затронутых объектов.

Однако вы можете заставить Hibernate установить значения атрибута version или timestamp с помощью versioned update. Это достигается путем добавления VERSIONED ключевого слова после UPDATE ключевого слова. **Это особенность Hibernate, и она не переносится**.

|  |  |
| --- | --- |
|  | Пользовательские типы версий, org.hibernate.usertype.UserVersionTypeне допускаются в сочетании с update versionedоператором. |

Оператор UPDATE выполняется с использованием executeUpdate()либо org.hibernate.query.Query или javax.persistence.Query. Метод известен тем , кто знаком с JDBC executeUpdate()на java.sql.PreparedStatement.

Значение int, возвращаемое executeUpdate()методом указывает число лиц , осуществленных с помощью операции. Это может или не может соотноситься с количеством строк в базе данных. Массовая операция HQL может привести к выполнению нескольких реальных операторов SQL (например, для присоединенного подкласса). Возвращенное число указывает на количество реальных объектов, затронутых оператором. **Используя иерархию наследования JOINED, удаление из одного из подклассов может фактически привести к удалению не только таблицы, в которую отображается этот подкласс, но также «корневой» таблицы и таблиц «между ними».**

Пример 22. Операторы запроса UPDATE

jpql

int updatedEntities = entityManager.createQuery(

"update Person p " +

"set p.name = :newName " +

"where p.name = :oldName" )

.setParameter( "oldName", oldName )

.setParameter( "newName", newName )

.executeUpdate();

hql

int updatedEntities = session.createQuery(

"update Person " +

"set name = :newName " +

"where name = :oldName" )

.setParameter( "oldName", oldName )

.setParameter( "newName", newName )

.executeUpdate();

int updatedEntities = session.createQuery(

"update versioned Person " +

"set name = :newName " +

"where name = :oldName" )

.setParameter( "oldName", oldName )

.setParameter( "newName", newName )

.executeUpdate();

|  |  |
| --- | --- |
|  | Ни операторы, UPDATE ни DELETE операторы не допускают неявных объединений. Их форма запрещает явные объединения. |

**DELETE**

BNF для DELETE операторов одинаков в HQL и JPQL:

delete\_statement ::= delete\_clause [where\_clause]

delete\_clause ::= DELETE FROM entity\_name [[AS] identification\_variable]

Утверждение DELETE также выполняется с использованием executeUpdate()метода либо org.hibernate.query.Query или javax.persistence.Query.

**INSERT**

HQL добавляет возможность определения INSERT операторов.

|  |  |
| --- | --- |
|  | Нет никакого JPQL эквивалента этому. |

BNF для оператора HQL INSERT:

insert\_statement ::= insert\_clause select\_statement

insert\_clause ::= INSERT INTO entity\_name (attribute\_list)

attribute\_list ::= state\_field[, state\_field ]\*

attribute\_list является аналогом column specification в SQL INSERT заявления. Для объектов, участвующих в отображенном наследовании, в атрибуте могут использоваться только атрибуты, непосредственно определенные в именованном объекте attribute\_list. Свойства суперкласса недопустимы, а свойства подкласса не имеют смысла. Другими словами, INSERT заявления по своей природе являются неполиморфными.

select\_statement может быть любым допустимым запросом выбора HQL с оговоркой, что возвращаемые типы должны соответствовать типам, ожидаемым вставкой. В настоящее время это проверяется во время компиляции запроса, а не позволяет проверке переходить в базу данных. Это может вызвать проблемы между Hibernate Types, которые *эквивалентны,* а не *равны* . Например, это может привести к проблемам с несоответствиями между атрибутом, сопоставленным как a, org.hibernate.type.DateTypeи атрибутом, определенным как a org.hibernate.type.TimestampType, даже если база данных может не проводить различий или может обрабатывать преобразование.

Для атрибута id оператор вставки дает вам две опции. Вы можете либо явно указать свойство id в attribute\_list, в этом случае его значение берется из соответствующего выражения выбора, либо опустить его в, attribute\_listв котором используется сгенерированное значение. Эта последняя опция доступна только при использовании генераторов идентификаторов, которые работают «в базе данных»; Попытка использовать эту опцию с любыми генераторами типа «в памяти» вызовет исключительную ситуацию при разборе.

Для атрибутов оптимистичной блокировки оператор вставки снова дает вам два варианта. Вы можете либо указать атрибут, в этом attribute\_listслучае его значение берется из соответствующих выражений выбора, либо опустить его attribute\_listв случае, когда используется seed valueопределение, соответствующее соответствующему org.hibernate.type.VersionType.

Пример 23. Операторы запроса INSERT

int insertedEntities = session.createQuery(

"insert into Partner (id, name) " +

"select p.id, p.name " +

"from Person p ")

.executeUpdate();

**FROM**

FROM отвечает за определение области действия типов объектной модели, доступных для остальной части запроса. Он также отвечает за определение всех «идентификационных переменных», доступных для остальной части запроса.

**Идентификационные переменные (алиасы)**

Идентификационные переменные часто называют псевдонимами. Ссылки на классы объектной модели в FROM предложении могут быть связаны с переменной идентификации, которая затем может использоваться для ссылки на этот тип в остальной части запроса.

В большинстве случаев объявление идентификационной переменной является необязательным, хотя обычно рекомендуется объявлять их.

Идентификационная переменная должна соответствовать правилам достоверности идентификатора Java.

Согласно JPQL идентификационные переменные должны рассматриваться как регистрозависимые. **Хорошая практика говорит, что вы должны использовать один и тот же регистр в запросе для ссылки на данную идентификационную переменную.** Другими словами, JPQL говорит, что они *могут быть* нечувствительными к регистру, и поэтому Hibernate должен иметь возможность обращаться с ними как таковыми, но это не делает это хорошей практикой.

**Ссылки на корневые объекты**

Ссылка на корневую сущность, или то, что JPA называет range variable declaration, является ссылкой на сопоставленный тип сущности из приложения. Он не может называть компоненты / встраиваемые типы. А ассоциации, в том числе коллекции, обрабатываются по-другому, ято будет показано позже.

BNF для ссылки на корневой объект:

root\_entity\_reference ::= entity\_name [AS] identification\_variable

Пример 24. Простой пример запроса

List<Person> persons = entityManager.createQuery(

"select p " +

"from org.hibernate.userguide.model.Person p", Person.class )

.getResultList();

Мы видим, что запрос определяет ссылку корневого объекта на org.hibernate.userguide.model.Person тип объектной модели. Кроме того, он объявляет псевдоним(алиас)  p этой ссылки.

**Обычно** ссылка на корневую сущность представляет собой не org.hibernate.userguide.model.Person, а просто entity name FQN (а не полное имя) сущностного класса. По умолчанию имя сущности является неквалифицированным именем класса сущности, здесь Person

Пример 25. Простой запрос с использованием имени объекта для ссылки на корневой объект

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p", Person.class )

.getResultList();

Можно также указать несколько ссылок на корневую сущность, даже если присвоить имя одной и той же сущности.

Пример 26. Простой запрос с использованием нескольких корневых ссылок на сущности

List<Object[]> persons = entityManager.createQuery(

"select distinct pr, ph " +

"from Person pr, Phone ph " +

"where ph.person = pr and ph is not null", Object[].class)

.getResultList();

List<Person> persons = entityManager.createQuery(

"select distinct pr1 " +

"from Person pr1, Person pr2 " +

"where pr1.id <> pr2.id " +

" and pr1.address = pr2.address " +

" and pr1.createdOn < pr2.createdOn", Person.class )

.getResultList();

**Явные объединения**

Предложение FROM также может содержать явные объединения отношений с использованием join ключевого слова. Эти объединения могут быть inner или left outer объединениями стиля.

Пример 27. Явные примеры *внутреннего* соединения

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"join pr.phones ph " +

"where ph.type = :phoneType", Person.class )

.setParameter( "phoneType", PhoneType.MOBILE )

.getResultList();

// тот же запрос, но явно указывающий тип соединения как «внутренний»

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"inner join pr.phones ph " +

"where ph.type = :phoneType", Person.class )

.setParameter( "phoneType", PhoneType.MOBILE )

.getResultList();

Пример 28. Примеры явного *внешнего* левого соединения

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"left join pr.phones ph " +

"where ph is null " +

" or ph.type = :phoneType", Person.class )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.getResultList();

// функционально тот же запрос, но с использованием фразы 'left external'

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"left outer join pr.phones ph " +

"where ph is null " +

" or ph.type = :phoneType", Person.class )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.getResultList();

HQL также определяет WITH предложение для определения условий соединения.

|  |  |
| --- | --- |
|  | Это специфично для HQL.  JPQL определяет *ON* предложение для этой функции. |

Пример 29. Пример WITH соединения предложения HQL

List<Object[]> personsAndPhones = session.createQuery(

"select pr.name, ph.number " +

"from Person pr " +

"left join pr.phones ph with ph.type = :phoneType " )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.list();

Пример 30. Пример ON соединения предложения JPQL

List<Object[]> personsAndPhones = entityManager.createQuery(

"select pr.name, ph.number " +

"from Person pr " +

"left join pr.phones ph on ph.type = :phoneType " )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.getResultList();

|  |  |
| --- | --- |
|  | Важным отличием является то, что в сгенерированном SQL условия WITH/ON условия становятся частью ON предложения в сгенерированном SQL, в отличие от других запросов в этом разделе, где условия HQL / JPQL становятся частью WHERE предложения в сгенерированном SQL, |

Различие в этом конкретном примере, вероятно, не так существенно. with clause иногда необходимо для более сложных запросов.

Явные объединения могут ссылаться на ассоциативные или компонентные / встроенные атрибуты. В случае с компонентами / встроенными атрибутами соединение является просто логическим и не соотносится с физическим (SQL) соединением. Для получения дополнительной информации о коллекционных ссылочных ссылках см. [Ссылки коллекционных членов](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-collection-valued-associations) .

Важный вариант использования явных объединений состоит в том, чтобы определить, **FETCH JOINS** для загрузки основной и всех связанных таблиц одним SQL запросом *(а не каждую связанную своим SQL запросом)*. В качестве примера, с учетом сущности, названной Person с именованной ассоциацией phones, JOIN FETCH коллекция также загрузит дочернюю коллекцию в тот же SQL-запрос:

Пример 31. Пример получения соединения

// functionally the same query but using the 'left outer' phrase

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"left join fetch pr.phones ", Person.class )

.getResultList();

Как видно из примера, соединение выборки определяется путем введения ключевого слова fetch после ключевого слова join. В этом примере мы использовали левое внешнее соединение, потому что мы также хотели вернуть клиентов, у которых нет заказов.

Внутренние объединения также могут быть получены, но внутренние объединения отфильтровывают корневую сущность. В этом примере использование вместо этого внутреннего соединения привело бы к тому, что клиенты не отфильтровывали бы заказы из результата.

|  |  |
| --- | --- |
|  | Соединения извлечения недопустимы в подзапросах.  Следует соблюдать осторожность, когда извлечение вступает в ассоциацию с ценным набором, которая каким-либо образом ограничивается (извлекаемая коллекция также будет ограничена). По этой причине обычно рекомендуется не назначать переменную идентификации для извлеченных объединений, кроме как для указания вложенных объединений извлечения.  Соединения извлечения не должны использоваться в постраничных запросах (например, setFirstResult()или setMaxResults()), а также не должны использоваться с функциями scroll()или iterate(). |

**Неявные объединения (выражения пути)**

Другим способом добавления в область действия типов объектных моделей, доступных для запроса, является использование неявных соединений или выражений пути.

Пример 32. Пример простого неявного соединения

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"where ph.person.address = :address ", Phone.class )

.setParameter( "address", address )

.getResultList();

// такой же как

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"join ph.person pr " +

"where pr.address = :address ", Phone.class )

.setParameter( "address", address)

.getResultList();

Неявное соединение всегда начинается с символа identification variable, за которым следует оператор навигации ( .), за которым следует атрибут для типа объектной модели, на который ссылается инициал identification variable.  В этом примере инициалом identification variable является ph ссылка на Phone объект. Затем ph.person ссылка ссылается на person атрибут Phone объекта.  Person это тип ассоциации, поэтому мы перейдем к его атрибуту age.

|  |  |
| --- | --- |
|  | Если атрибут представляет ассоциацию объекта (не коллекцию) или компонент / внедренный, эта ссылка может быть далее перемещена. Базовые значения и ассоциации, связанные с коллекциями, не подлежат дальнейшей навигации. |

Как показано в примере, неявные объединения могут появляться вне FROM clause. Тем не менее, они влияют на FROM clause.

|  |  |
| --- | --- |
|  | Неявные объединения всегда рассматриваются как внутренние объединения.  Несколько ссылок на одно и то же неявное соединение всегда ссылаются на одно и то же логическое и физическое (SQL) соединение. |

Пример 33. Повторное неявное соединение

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"where ph.person.address = :address " +

" and ph.person.createdOn > :timestamp", Phone.class )

.setParameter( "address", address )

.setParameter( "timestamp", timestamp )

.getResultList();

// такой же как

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"inner join ph.person pr " +

"where pr.address = :address " +

" and pr.createdOn > :timestamp", Phone.class )

.setParameter( "address", address )

.setParameter( "timestamp", timestamp )

.getResultList();

Как и в случае явных объединений, неявные объединения могут ссылаться на ассоциации или атрибуты компонента / встраиваемого объекта. Для получения дополнительной информации о коллекционных ссылочных ссылках см. [Ссылки коллекционных членов](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-collection-valued-associations) .

В случае с компонентами / встроенными атрибутами соединение является просто логическим и не соотносится с физическим (SQL) соединением. Однако, в отличие от явных объединений, неявные объединения могут также ссылаться на поля основного состояния, пока на этом заканчивается выражение пути.

**DISTINCT**

Для JPQL и HQL DISTINCT имеет два значения:

1. Его можно передать в базу данных, чтобы дубликаты были удалены из набора результатов
2. Его можно использовать для фильтрации тех же ссылок на родительские сущности при соединении, извлекающих дочернюю коллекцию.

**Использование DISTINCT с проекциями SQL**

Для проекций SQL их DISTINCT необходимо передать в базу данных, поскольку дублированные записи необходимо отфильтровать перед возвратом в клиент базы данных.

Пример 34. Использование DISTINCT с примером проекционных запросов

List<String> lastNames = entityManager.createQuery(

"select distinct p.lastName " +

"from Person p", String.class)

.getResultList();

При выполнении вышеуказанного запроса Hibernate генерирует следующий запрос SQL:

SELECT DISTINCT

p.last\_name as col\_0\_0\_

FROM person p

Для этого конкретного случая использования DISTINCT правильная передача ключевого слова из JPQL/HQL в базу данных.

**Использование DISTINCT с запросами сущностей**

DISTINCT также может использоваться для фильтрации ссылок на объекты сущностей при извлечении дочерней ассоциации вместе с родительскими сущностями.

Пример 35. Пример неправильного использования DISTINCT с запросами сущностей

List<Person> authors = entityManager.createQuery(

"select distinct p " +

"from Person p " +

"left join fetch p.books", Person.class)

.getResultList();

В этом случае DISTINCT используется, потому что может быть несколько Books объектов, связанных с данным Person. Если в базе данных есть 3 Persons и у каждого человека есть 2 Books, без DISTINCT запрос с “fetch” вернется 6 Persons,так как размер набора результатов дается числом соединенных Book записей.

Однако ключевое слово DISTINCT также передается в базу данных:

SELECT DISTINCT

p.id as id1\_1\_0\_,

b.id as id1\_0\_1\_,

p.first\_name as first\_na2\_1\_0\_,

p.last\_name as last\_nam3\_1\_0\_,

b.author\_id as author\_i3\_0\_1\_,

b.title as title2\_0\_1\_,

b.author\_id as author\_i3\_0\_0\_\_,

b.id as id1\_0\_0\_\_

FROM person p

LEFT OUTER JOIN book b ON p.id=b.author\_id

В этом случае ключевое слово DISTINCT нежелательно, поскольку оно выполняет избыточную сортировку набора результатов, как описано [в этом сообщении в блоге](http://in.relation.to/2016/08/04/introducing-distinct-pass-through-query-hint/). Чтобы исправить эту проблему, Hibernate 5.2.2 добавил поддержку **HINT\_PASS\_DISTINCT\_THROUGH** подсказки запроса сущности:

Пример 36. Пример правильного использования DISTINCT с запросами сущностей

List<Person> authors = entityManager.createQuery(

"select distinct p " +

"from Person p " +

"left join fetch p.books", Person.class)

.setHint( QueryHints.HINT\_PASS\_DISTINCT\_THROUGH, false )

.getResultList();

С помощью этой подсказки запроса сущности Hibernate не будет передавать ключевое слово DISTINCT в запрос SQL:

SELECT

p.id as id1\_1\_0\_,

b.id as id1\_0\_1\_,

p.first\_name as first\_na2\_1\_0\_,

p.last\_name as last\_nam3\_1\_0\_,

b.author\_id as author\_i3\_0\_1\_,

b.title as title2\_0\_1\_,

b.author\_id as author\_i3\_0\_0\_\_,

b.id as id1\_0\_0\_\_

FROM person p

LEFT OUTER JOIN book b ON p.id=b.author\_id

При использовании HINT\_PASS\_DISTINCT\_THROUGH подсказки запроса сущности Hibernate может по-прежнему удалять дублированные сущности родительской стороны из результата запроса.

**Ссылки коллекций на свои коллекции ()**

Пример 37. Пример ссылки коллекции на свою внутреннюю коллекцию

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Person pr " +

"join pr.phones ph " +

"join ph.calls c " +

"where pr.address = :address " +

" and c.duration > :duration", Phone.class )

.setParameter( "address", address )

.setParameter( "duration", duration )

.getResultList();

// альтернативное написание

List<Phone> phones = session.createQuery(

"select ph " +

"from Person pr, " +

"in (pr.phones) ph, " +

"in (ph.calls) c " +

"where pr.address = :address " +

" and c.duration > :duration" )

.setParameter( "address", address )

.setParameter( "duration", duration )

.list();

В этом примере переменная идентификации на ph самом деле относится к типу объектной модели Phone, который является типом элементов Person#phones ассоциации.

В этом примере также показан альтернативный синтаксис для указания объединений ассоциации коллекции с использованием IN синтаксиса. Обе формы эквивалентны. Какую форму приложение выберать для использования - вопрос вкуса.

**Особый случай - квалифицированные выражения пути**

Пример 38. Пример квалифицированной коллекции ссылок когда ключ таблицы является MAP-ом

@OneToMany(mappedBy = "phone")

@MapKey(name = "timestamp")

@MapKeyTemporal(TemporalType.TIMESTAMP )

private Map<Date, Call> callHistory = new HashMap<>();

// выбрать все звонки (значение карты) для данного Phone

List<Call> calls = entityManager.createQuery(

"select ch " +

"from Phone ph " +

"join ph.callHistory ch " +

"where ph.id = :id ", Call.class )

.setParameter( "id", id )

.getResultList();

// такой же как выше

List<Call> calls = entityManager.createQuery(

"select value(ch) " +

"from Phone ph " +

"join ph.callHistory ch " +

"where ph.id = :id ", Call.class )

.setParameter( "id", id )

.getResultList();

// выбрать все the Call timestamps (the map **key**) для данного Phone

// Выбирает только ключи

List<Date> timestamps = entityManager.createQuery(

"select **key**(ch) " +

"from Phone ph " +

"join ph.callHistory ch " +

"where ph.id = :id ", Date.class )

.setParameter( "id", id )

.getResultList();

// select all the Call and their timestamps (the 'Map.**Entry**') for a given Phone

// Выбирает ключ+значение

List<Map.Entry<**Date, Call**>> callHistory = entityManager.createQuery(

"select **entry**(ch) " +

"from Phone ph " +

"join ph.callHistory ch " +

"where ph.id = :id " )

.setParameter( "id", id )

.getResultList();

// Sum all call durations for a given Phone of a specific Person

Long duration = entityManager.createQuery(

"select sum(ch.duration) " +

"from Person pr " +

"join pr.phones ph " +

"join ph.callHistory ch " +

"where ph.id = :id " +

" and index(ph) = :phoneIndex", Long.class )

.setParameter( "id", id )

.setParameter( "phoneIndex", phoneIndex )

.getSingleResult();

VALUE

Можно не указывать. Полезно, чтобы явно показать намерение. Действительно для любого типа коллекционной ссылки.

KEY

Действительно **только** для **Maps**. Относится к ключу карты. Если ключ сам по себе является сущностью, его можно перемещать дальше.

ENTRY

Действителен **только** для **Maps**. Относится к логическому java.util.Map.Entry кортежу карты (комбинации его ключа и значения). ENTRY допустимо только в качестве конечного пути и применимо только к SELECT предложению.

ИНДЕКС

Согласно правилам HQL, это справедливо для обоих, Maps и в Lists них указывается javax.persistence.OrderColumn аннотация для ссылки на Map ключ или List позицию (или OrderColumn значение). JPQL, однако, резервирует это для использования List и добавляет KEY для Map случая. Приложения, заинтересованные в переносимости JPA-провайдеров, должны знать об этом различии.

Посмотрите [выражения, связанные](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-collection-expressions) с коллекцией, для получения дополнительной информации о выражениях, связанных с коллекцией.

**Полиморфизм**

Запросы HQL и JPQL по своей природе полиморфны.

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p ", Payment.class )

.getResultList();

Этот запрос Payment явно называет сущность. Однако все подклассы Payment также доступны для запроса. Таким образом, если сущности CreditCardPayment и WireTransferPayment расширяют сущность Payment, все три типа будут доступны для запроса объекта и запрос будет возвращать экземпляры всех троих.

Это поведение может быть изменено двумя способами:

* ограничивая запрос, чтобы выбрать только из сущности подкласса
* используя либо аннотацию org.hibernate.annotations.Polymorphism (глобальную, и специфичную для Hibernate). См. [@PolymorphismРаздел для получения дополнительной информации об этом сценарии использования](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/domain/inheritance.html#entity-inheritance-polymorphism) .

|  |  |
| --- | --- |
|  | HQL-запрос from java.lang.Object полностью действителен (хотя и не очень практичен с точки зрения производительности)**!** Он возвращает каждый объект каждого типа сущности, определенный вашими сопоставлениями приложения. |

**Выражения**

По сути, выражения - это ссылки, которые разрешают базовые значения или значения кортежей.

**Идентификационная переменная**

См [В FROMстатью](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-from-clause) .

**Выражения пути**

Снова, см [. FROMПункт](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-from-clause) .

**литералы**

Строковые литералы заключены в одинарные кавычки. Чтобы избежать одиночной кавычки в строковом литерале, используйте двойные одинарные кавычки.

Пример 39. Примеры строковых литералов

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like 'Joe'", Person.class)

.getResultList();

// Escaping quotes

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like 'Joe''s'", Person.class)

.getResultList();

Числовые литералы допускаются в нескольких различных формах.

Пример 40. Числовые примеры

// simple integer literal

Person person = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.id = 1", Person.class)

.getSingleResult();

// simple integer literal, typed as a long

Person person = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.id = 1L", Person.class)

.getSingleResult();

// decimal notation

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration > 100.5", Call.class )

.getResultList();

// decimal notation, typed as a float

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration > 100.5F", Call.class )

.getResultList();

// scientific notation

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration > 1e+2", Call.class )

.getResultList();

// scientific notation, typed as a float

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration > 1e+2F", Call.class )

.getResultList();

|  |  |
| --- | --- |
|  | В форме научной нотации Eрегистр не учитывается.  Конкретная типизация может быть достигнута с использованием того же суффиксного подхода, который указан в Java. Итак, L обозначает длинный, D обозначает двойной, F обозначает плавающий. Фактический суффикс не зависит от регистра.  Логические литералы TRUE и FALSE, опять же, без учета регистра.  На перечисления можно даже ссылаться как на литералы. Необходимо использовать полное имя класса enum. HQL также может обрабатывать константы таким же образом, хотя JPQL не определяет это как поддерживаемое.  Имена сущностей также могут использоваться как буквальные. Смотрите [тип объекта](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-entity-type-exp) .  Литералы даты / времени могут быть указаны с использованием escape-синтаксиса JDBC:   * {d 'yyyy-mm-dd'} для дат * {t 'hh:mm:ss'} на времени * {ts 'yyyy-mm-dd hh:mm:ss[.millis]'} (по желанию миллис) для отметок времени.   Эти литералы даты / времени работают только в том случае, если драйверы JDBC их поддерживают. |

**арифметика**

Арифметические операции также представляют допустимые выражения.

Пример 41. Числовые арифметические примеры

// select clause date/time arithmetic operations

Long duration = entityManager.createQuery(

"select sum(ch.duration) \* :multiplier " +

"from Person pr " +

"join pr.phones ph " +

"join ph.callHistory ch " +

"where ph.id = 1L ", Long.class )

.setParameter( "multiplier", 1000L )

.getSingleResult();

// select clause date/time arithmetic operations

Integer years = entityManager.createQuery(

"select year( current\_date() ) - year( p.createdOn ) " +

"from Person p " +

"where p.id = 1L", Integer.class )

.getSingleResult();

// where clause arithmetic operations

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where year( current\_date() ) - year( p.createdOn ) > 1", Person.class )

.getResultList();

Следующие правила применяются к результату арифметических операций:

* Если один из операндов равен Double/ double, результатом является Double
* иначе, если один из операндов равен Float/ float, результатом является Float
* иначе, если любой из операндов есть BigDecimal, результат BigDecimal
* иначе, если любой из операндов есть BigInteger, результат BigInteger *(кроме деления, в этом случае тип результата больше не определяется)*
* иначе, если любой из операндов равен Long/ long, результатом является Long *(за исключением деления, в этом случае тип результата больше не определяется)*
* иначе, (при условии, что оба операнда имеют целочисленный тип), результатом является Integer *(за исключением деления, в этом случае тип результата не определяется далее)*

Арифметика даты также поддерживается, хотя и более ограниченным образом. Частично это связано с различиями в поддержке базы данных и частично с отсутствием поддержки INTERVAL определения в самом языке запросов.

**Конкатенация (операция)**

HQL определяет оператор конкатенации в дополнение к поддержке функции concatenation ( CONCAT). Это не определено в JPQL, поэтому переносимые приложения должны избегать его использования. Оператор конкатенации берется из оператора конкатенации SQL (например ||).

Пример 42. Пример операции конкатенации

String name = entityManager.createQuery(

"select 'Customer ' || p.name " +

"from Person p " +

"where p.id = 1", String.class )

.getSingleResult();

См. [Скалярные функции](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-exp-functions) для деталей о concat()функции

**Агрегатные функции**

Агрегатные функции также являются допустимыми выражениями в HQL и JPQL. Семантика такая же, как у их SQL-аналога. Поддерживаемые агрегатные функции:

COUNT (включая отдельные / все квалификаторы)

Тип результата всегда Long.

AVG

Тип результата всегда Double.

MIN

Тип результата совпадает с типом аргумента.

MAX

Тип результата совпадает с типом аргумента.

SUM

Тип результата SUM()функции зависит от типа суммируемых значений. Для целочисленных значений (кроме BigInteger) тип результата - Long.

Для значений с плавающей запятой (кроме BigDecimal) тип результата Double. Для BigInteger значений, тип результат BigInteger. Для BigDecimal значений, тип результат BigDecimal.

Пример 43. Примеры агрегатных функций

Object[] callStatistics = entityManager.createQuery(

"select " +

" count(c), " +

" sum(c.duration), " +

" min(c.duration), " +

" max(c.duration), " +

" avg(c.duration) " +

"from Call c ", Object[].class )

.getSingleResult();

Long phoneCount = entityManager.createQuery(

"select count( distinct c.phone ) " +

"from Call c ", Long.class )

.getSingleResult();

List<Object[]> callCount = entityManager.createQuery(

"select p.number, count(c) " +

"from Call c " +

"join c.phone p " +

"group by p.number", Object[].class )

.getResultList();

Агрегации часто появляются с группировкой. Для получения информации о группировке см. [Группировка по](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-group-by) .

**Скалярные функции**

И HQL, и JPQL определяют некоторые стандартные функции, которые доступны независимо от используемой базовой базы данных. HQL также может понимать дополнительные функции, определенные Диалектом, а также приложением.

**Стандартизированные функции JPQL**

Вот список функций, определенных как поддерживаемые JPQL. Приложения, заинтересованные в сохранении переносимости между провайдерами JPA, должны придерживаться этих функций.

CONCAT

Функция конкатенации строк. Переменная длина аргумента 2 или более строковых значения, которые будут объединены вместе.

List<String> callHistory = entityManager.createQuery(

"select concat( p.number, ' : ' , cast(c.duration as string) ) " +

"from Call c " +

"join c.phone p", String.class )

.getResultList();

SUBSTRING

Извлекает часть строкового значения. Второй аргумент обозначает начальную позицию, где 1 - первый символ строки. Третий (необязательный) аргумент обозначает длину.

List<String> prefixes = entityManager.createQuery(

"select substring( p.number, 1, 2 ) " +

"from Call c " +

"join c.phone p", String.class )

.getResultList();

ВЕРХНИЙ

Верхний регистр указанной строки

List<String> names = entityManager.createQuery(

"select upper( p.name ) " +

"from Person p ", String.class )

.getResultList();

НИЖНИЙ

Строчные буквы указанной строки

List<String> names = entityManager.createQuery(

"select lower( p.name ) " +

"from Person p ", String.class )

.getResultList();

ОТДЕЛКА

Следует семантике функции обрезки SQL.

List<String> names = entityManager.createQuery(

"select trim( p.name ) " +

"from Person p ", String.class )

.getResultList();

ДЛИНА

Возвращает длину строки.

List<Integer> lengths = entityManager.createQuery(

"select length( p.name ) " +

"from Person p ", Integer.class )

.getResultList();

LOCATE

Находит строку в другой строке. Третий аргумент (необязательный) используется для обозначения позиции, с которой начинается поиск.

List<Integer> sizes = entityManager.createQuery(

"select locate( 'John', p.name ) " +

"from Person p ", Integer.class )

.getResultList();

ABS

Вычисляет математическое абсолютное значение числового значения.

List<Integer> abs = entityManager.createQuery(

"select abs( c.duration ) " +

"from Call c ", Integer.class )

.getResultList();

MOD

Вычисляет остаток от деления первого аргумента на второй.

List<Integer> mods = entityManager.createQuery(

"select mod( c.duration, 10 ) " +

"from Call c ", Integer.class )

.getResultList();

SQRT

Вычисляет математический квадратный корень из числового значения.

List<Double> sqrts = entityManager.createQuery(

"select sqrt( c.duration ) " +

"from Call c ", Double.class )

.getResultList();

CURRENT\_DATE

Возвращает текущую дату базы данных.

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.timestamp = current\_date", Call.class )

.getResultList();

CURRENT\_TIME

Возвращает базу данных текущего времени.

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.timestamp = current\_time", Call.class )

.getResultList();

CURRENT\_TIMESTAMP

Возвращает текущую временную метку базы данных.

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.timestamp = current\_timestamp", Call.class )

.getResultList();

**HQL-функции**

Помимо стандартизированных функций JPQL, HQL предоставляет некоторые дополнительные функции независимо от используемой базовой базы данных.

BIT\_LENGTH

Возвращает длину двоичных данных.

List<Number> bits = entityManager.createQuery(

"select bit\_length( c.duration ) " +

"from Call c ", Number.class )

.getResultList();

**cast**

Выполняет приведение SQL. Целевой объект должен назвать используемый тип отображения Hibernate. См. Главу о [типах данных](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/domain/basic_types.html#basic-provided) для получения дополнительной информации.

List<String> durations = entityManager.createQuery(

"select cast( c.duration as string ) " +

"from Call c ", String.class )

.getResultList();

**extract**

Выполняет извлечение SQL для значений даты и времени. Извлечение извлекает части даты и времени (например, год).

List<Integer> years = entityManager.createQuery(

"select extract( YEAR from c.timestamp ) " +

"from Call c ", Integer.class )

.getResultList();

Смотрите сокращенные формы ниже.

ГОД

Сокращенная форма выдержки для извлечения года.

List<Integer> years = entityManager.createQuery(

"select year( c.timestamp ) " +

"from Call c ", Integer.class )

.getResultList();

МЕСЯЦ

Сокращенная форма выписки для извлечения месяца.

ДЕНЬ

Сокращенная форма выдержки для извлечения дня.

ЧАС

Сокращенная форма выдержки для извлечения часа.

МИНУТЫ

Сокращенная форма выдержки для извлечения минут.

ВТОРОЙ

Сокращенная форма выдержки для извлечения второй.

STR

Сокращенная форма для приведения значения в качестве символьных данных.

List<String> timestamps = entityManager.createQuery(

"select str( c.timestamp ) " +

"from Call c ", String.class )

.getResultList();

List<String> timestamps = entityManager.createQuery(

"select str( cast(duration as float) / 60, 4, 2 ) " +

"from Call c ", String.class )

.getResultList();

**Нестандартизированные функции**

Диалекты Hibernate могут регистрировать дополнительные функции, которые, как известно, доступны для этого конкретного продукта базы данных. Эти функции также доступны в HQL (и JPQL, хотя, конечно, только при использовании Hibernate в качестве поставщика JPA). Однако они будут доступны только при использовании этой базы данных Диалект. Приложения, нацеленные на переносимость базы данных, должны избегать использования функций этой категории.

Разработчики приложений также могут предоставить свой собственный набор функций. Обычно это будут либо пользовательские функции SQL, либо псевдонимы для фрагментов SQL. Такие объявления функций создаются с использованием addSqlFunction()метода org.hibernate.cfg.Configuration.

**Выражения, связанные с коллекцией**

Существует несколько специализированных выражений для работы с коллекционными ассоциациями. Как правило, это просто сокращенные формы или другие выражения для краткости.

**size**

Рассчитать размер коллекции. Приравнивается к подзапросу!

MAXELEMENT

Доступно для использования на коллекциях базового типа. Относится к максимальному значению, определенному путем применения maxагрегации SQL.

MAXINDEX

Доступно для использования в индексированных коллекциях. Относится к максимальному индексу (ключ / позиция), который определяется путем применения maxагрегации SQL.

MINELEMENT

Доступно для использования на коллекциях базового типа. Относится к минимальному значению, определенному путем применения minагрегации SQL.

MININDEX

Доступно для использования в индексированных коллекциях. Относится к минимальному индексу (ключ / позиция), который определяется путем применения minагрегации SQL.

ЭЛЕМЕНТЫ

Используется для ссылки на элементы коллекции в целом. Разрешено только в предложении where. Часто используется в сочетании с ALL, ANY или SOME ограничениями.

**indices**

Аналогично elements, только indices индексы коллекций (ключи / позиции) в целом.

Пример 44. Примеры связанных выражений

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where **maxelement**( p.calls ) = :call", Phone.class )

.setParameter( "call", call )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where **minelement**( p.calls ) = :call", Phone.class )

.setParameter( "call", call )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where **maxindex**( p.phones ) = 0", Person.class )

.getResultList();

// the above query can be re-written with member of

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where :phone **member** of p.phones", Person.class )

.setParameter( "phone", phone )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where :phone = some elements ( p.phones )", Person.class )

.setParameter( "phone", phone )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where **exists** elements ( p.phones )", Person.class )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where **current\_date()** > key( p.callHistory )", Phone.class )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where **current\_date()** > all elements( p.repairTimestamps )", Phone.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where 1 in **indices**( p.phones )", Person.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where **size**( p.phones ) = 2", Person.class )

.getResultList();

Элементы индексированных коллекций (массивы, списки и карты) могут указываться оператором индекса.

Пример 45. Примеры операторов индекса

// indexed lists

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.phones[ 0 ].type = 'LAND\_LINE'", Person.class )

.getResultList();

// maps

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.addresses[ 'HOME' ] = :address", Person.class )

.setParameter( "address", address)

.getResultList();

//max index in list

List<Person> persons = entityManager.createQuery(

"select pr " +

"from Person pr " +

"where pr.phones[ maxindex(pr.phones) ].type = 'LAND\_LINE'", Person.class )

.getResultList();

См. Также [Особые выражения с указанием пути,](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-collection-qualification) поскольку существует много совпадений.

**TYPE**

Мы также можем ссылаться на тип объекта как выражение через type. Это в основном полезно при работе с иерархиями наследования объектов. Тип может быть выражен с использованием TYPE функции, используемой для ссылки на тип идентификационной переменной, представляющей сущность. Имя объекта также служит способом ссылки на тип объекта. Кроме того, тип объекта может быть параметризован, и в этом случае ссылка на класс Java объекта будет привязана в качестве значения параметра.

Пример 46. Примеры выражений типа сущностей

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p " +

"where type(p) = CreditCardPayment", Payment.class )

.getResultList();

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p " +

"where type(p) = :type", Payment.class )

.setParameter( "type", WireTransferPayment.class)

.getResultList();

|  |  |
| --- | --- |
|  | HQL также имеет устаревшую форму ссылки на тип сущности, хотя эта устаревшая форма считается устаревшей в пользу TYPE. Устаревшая форма использовалась бы p.classв примерах, а не type(p). Упоминается только для полноты. |

**CASE / COALESCE выражения**

**(аналог switch/case/default в java)**

Поддерживаются как простые, так и искомые формы, а также две определенные в SQL сокращенные формы ( **NULLIF** и **COALESCE**)

**Простые выражения CASE / COALESCE**

Простая форма имеет следующий синтаксис:

CASE {operand} WHEN {test\_value} THEN {match\_result} ELSE {miss\_result} END

Пример 47. Пример простого выражения

List<String> nickNames = entityManager.createQuery(

"select " +

" case p.nickName " +

" when 'NA' " +

" then '<no nick name>' " +

" else p.nickName " +

" end " +

"from Person p", String.class )

.getResultList();

// аналогичный верхнему

List<String> nickNames = entityManager.createQuery(

"select coalesce(p.nickName, '<no nick name>') " +

"from Person p", String.class )

.getResultList();

**CASE / COALESCE выражения для поиска**

Форма поиска имеет следующий синтаксис:

CASE [ WHEN {test\_conditional} THEN {match\_result} ]\* ELSE {miss\_result} END

Пример 48. Пример искомого выражения

List<String> nickNames = entityManager.createQuery(

"select " +

" case " +

" when p.nickName is null " +

" then " +

" case " +

" when p.name is null " +

" then '<no nick name>' " +

" else p.name " +

" end" +

" else p.nickName " +

" end " +

"from Person p", String.class )

.getResultList();

// аналогичный верхнему

List<String> nickNames = entityManager.createQuery(

"select coalesce( p.nickName, p.name, '<no nick name>' ) " +

"from Person p", String.class )

.getResultList();

**Выражения NULLIF**

NULLIF - это сокращенное выражение CASE, которое возвращает NULL, если его операнды считаются равными.

Пример 49. Пример NULLIF

List<String> nickNames = entityManager.createQuery(

"select nullif( p.nickName, p.name ) " +

"from Person p", String.class )

.getResultList();

// equivalent CASE expression

List<String> nickNames = entityManager.createQuery(

"select " +

" case" +

" when p.nickName = p.name" +

" then null" +

" else p.nickName" +

" end " +

"from Person p", String.class )

.getResultList();

**SELECT**

Предложение SELECT определяет, какие объекты и значения возвращать в качестве результата запроса. Все выражения, обсуждаемые в [выражениях,](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-expressions) являются допустимыми выражениями выбора, если не указано иное. См. Раздел [Hibernate Query API](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-api) для получения информации об обработке результатов в зависимости от типов значений, указанных в SELECT предложении.

Существует определенный тип выражения, который действителен только в предложении select.

**NEW**. Hibernate называет это «динамической реализацией». JPQL поддерживает некоторые из этих функций и называет их «выражением конструктора».

Поэтому вместо того, чтобы иметь дело с Object[](опять же, см. [Hibernate Query API](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-api) ), мы заключаем значения в типобезопасный Java-объект, который будет возвращен как результат запроса.

Пример 50. Пример создания динамического HQL и JPQL

public class **CallStatistics** {

private final long count;

private final long total;

private final int min;

private final int max;

private final double abg;

public CallStatistics(long count, long total, int min, int max, double abg) {

this.count = count;

this.total = total;

this.min = min;

this.max = max;

this.abg = abg;

}

// Геттеры и сеттеры опущены для краткости

}

**CallStatistics** callStatistics = entityManager.createQuery(

"select new org.hibernate.userguide.hql.**CallStatistics** (" +

" count(c), " +

" sum(c.duration), " +

" min(c.duration), " +

" max(c.duration), " +

" avg(c.duration)" +

") " +

"from Call c ", **CallStatistics**.class )

.getSingleResult();

|  |  |
| --- | --- |
|  | Ссылка на класс должна быть полностью квалифицированной и иметь соответствующий конструктор. |
|  | Класс здесь не нужно отображать. Если он действительно представляет сущность, полученные экземпляры возвращаются в состоянии NEW (не управляется!). |

HQL поддерживает дополнительные функции "динамической реализации". Во-первых, в запросе можно указать возвращаемые List, а не Object[]скалярные результаты:

Пример 51. Пример динамического создания - список

List<List> phoneCallDurations = entityManager.createQuery(

"select new list(" +

" p.number, " +

" c.duration " +

") " +

"from Call c " +

"join c.phone p ", List.class )

.getResultList();

Результаты этого запроса будут List<List>в отличие от List<Object[]>

HQL также поддерживает перенос скалярных результатов в Map.

Пример 52. Пример динамического создания - карта

List<Map> phoneCallTotalDurations = entityManager.createQuery(

"select new map(" +

" p.number as phoneNumber , " +

" sum(c.duration) as totalDuration, " +

" avg(c.duration) as averageDuration " +

") " +

"from Call c " +

"join c.phone p " +

"group by p.number ", Map.class )

.getResultList();

Результаты этого запроса будут a, List<Map<String, Object>>, а не a List<Object[]>. Ключи карты определяются псевдонимами, заданными в выражениях выбора. Если пользователь не назначает псевдонимы, ключом будет индекс каждого конкретного столбца набора результатов (например, 0, 1, 2 и т. Д.).

**Предикаты**

Предикаты составляют основу предложения where, предложения has и выражений поиска. Они являются выражениями, которые разрешают к значению истины, как правило, TRUE или FALSE, хотя логические сравнения, включающие, NULL как правило, разрешают к UNKNOWN.

**Реляционные сравнения**

Сравнения включают один из операторов сравнения: =, >, >=, <, <=, <>. HQL также определяет != в качестве оператора сравнения, синоним <>. Операнды должны быть одного типа.

Пример 53. Сравнительные примеры сравнения

// numeric comparison

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration < 30 ", Call.class )

.getResultList();

// string comparison

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like 'John%' ", Person.class )

.getResultList();

// datetime comparison

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.createdOn > '1950-01-01' ", Person.class )

.getResultList();

// enum comparison

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where p.type = 'MOBILE' ", Phone.class )

.getResultList();

// boolean comparison

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p " +

"where p.completed = true ", Payment.class )

.getResultList();

// boolean comparison

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p " +

"where type(p) = WireTransferPayment ", Payment.class )

.getResultList();

// entity value comparison

List<Object[]> phonePayments = entityManager.createQuery(

"select p " +

"from Payment p, Phone ph " +

"where p.person = ph.person ", Object[].class )

.getResultList();

Сравнения могут также включать в себя подзапросы классификаторов: ALL, ANY, SOME.  SOME и ANY являются синонимами.

Классификатор **ALL** принимает значение true, если сравнение истинно **для всех значений** в результате подзапроса. Он принимает значение false, если результат подзапроса пуст.

Пример 54. Пример квалификатора сравнения всех подзапросов

// выбрать всех людей со всеми звонками короче 50 секунд

List<Person> persons = entityManager.createQuery(

"select distinct p.person " +

"from Phone p " +

"join p.calls c " +

"where 50 > all ( " +

" select duration" +

" from Call" +

" where phone = p " +

") ", Person.class )

.getResultList();

Спецификатор ANY/ SOME разрешается в true, если сравнение истинно для некоторых (хотя бы одного) значений в результате подзапроса. Он принимает значение false, если результат подзапроса пуст.

**NULL**

Проверяет значение на null. Может применяться к базовым ссылкам на атрибуты, ссылкам на сущности и параметрам. HQL дополнительно позволяет применять его к компонентам / встраиваемым типам.

Пример 55. Примеры проверки на null

// select all persons with a nickname

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.nickName is not null", Person.class )

.getResultList();

// select all persons without a nickname

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.nickName is null", Person.class )

.getResultList();

**LIKE**

Выполняет аналогичное сравнение строковых значений. Синтаксис:

like\_expression ::=

string\_expression

[NOT] LIKE pattern\_value

[ESCAPE escape\_character]

Семантика совпадает с семантикой SQL-подобного выражения.  pattern\_value Это шаблон , чтобы попытаться матч в string\_expression. Так же, как SQL, pattern\_value можно использовать “**\_**“ и “**%**“ в качестве подстановочных знаков. Значения одинаковы.  Символ \_ соответствует любому символу и % совпадает с любым количеством символов.

Пример 56. Подобные предикатные примеры

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like 'Jo%'", Person.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name not like 'Jo%'", Person.class )

.getResultList();

Необязательный escape 'escape character' используется для указания escape-символа, используемого для экранирования специального значения \_и % в pattern\_value. Это полезно , когда необходимость искать на образцах , включая либо \_ или %.

Синтаксис формируется следующим образом: **'like\_predicate' escape 'escape\_symbol'**(типа like 'Dr|\_%' escape '|') Итак, если ”|”это escape-символ и мы хотим сопоставить все хранимые процедуры с префиксом Dr\_, аналогичные критерии становятся следующими 'Dr|\_%' escape '**|**':

Пример 57. Как с escape-символом

// find any person with a name starting with "Dr\_"

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like 'Dr|\_%' escape '|'", Person.class )

.getResultList();

**BETWEEN**

Аналогично BETWEEN выражению SQL , он проверяет, находится ли значение в границах. Все операнды должны иметь сопоставимые типы.

Пример 58. Между примерами предикатов

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"join p.phones ph " +

"where p.id = 1L and index(ph) between 0 and 3", Person.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.createdOn between '1999-01-01' and '2001-01-02'", Person.class )

.getResultList();

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"where c.duration between 5 and 20", Call.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name between 'H' and 'M'", Person.class )

.getResultList();

**IN**

Предикат IN выполняет проверку того, что конкретное значение находится в списке значений. Его синтаксис:

in\_expression ::=

single\_valued\_expression [NOT] IN single\_valued\_list

single\_valued\_list ::=

constructor\_expression | (subquery) | collection\_valued\_input\_parameter

constructor\_expression ::= (expression[, expression]\*)

Типы single\_valued\_expression и отдельные значения в single\_valued\_list должны быть согласованы.

**JPQL ограничивает допустимые типы здесь строковым, числовым, датой, временем, отметкой времени и перечислением, а в JPQL single\_valued\_expression может ссылаться только на:**

* **«поля состояния», что является термином для простых атрибутов. В частности, это исключает связь и атрибуты компонента / внедренного.**
* **выражения типа сущности. Смотрите**[**тип объекта**](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-entity-type-exp)

В HQL single\_valued\_expression может ссылаться на гораздо более широкий набор типов выражений. Допускаются однозначные ассоциации, а также атрибуты компонента / встраивания, хотя эта функция зависит от уровня поддержки кортежа или «синтаксиса конструктора значения строки» в базовой базе данных. Кроме того, HQL никоим образом не ограничивает тип значения, хотя разработчики приложений должны знать, что разные типы могут иметь ограниченную поддержку в зависимости от поставщика базы данных. Это в значительной степени причина ограничений JPQL.

Список значений может быть из разных источников. В constructor\_expression и collection\_valued\_input\_parameter список значений не должен быть пустым; оно должно содержать хотя бы одно значение.

Пример 59. В примерах предикатов

List<Payment> payments = entityManager.createQuery(

"select p " +

"from Payment p " +

"where type(p) **in** ( CreditCardPayment, WireTransferPayment )", Payment.class )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where type **in** ( 'MOBILE', 'LAND\_LINE' )", Phone.class )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select p " +

"from Phone p " +

"where type **in** :types", Phone.class )

.setParameter( "types", Arrays.asList( PhoneType.MOBILE, PhoneType.LAND\_LINE ) )

.getResultList();

List<Phone> phones = entityManager.createQuery(

"select distinct p " +

"from Phone p " +

"where p.person.id **in** (" +

" select py.person.id " +

" from Payment py" +

" where py.completed = true and py.amount > 50 " +

")", Phone.class )

.getResultList();

// Not JPQL compliant!

List<Phone> phones = entityManager.createQuery(

"select distinct p " +

"from Phone p " +

"where p.person in (" +

" select py.person " +

" from Payment py" +

" where py.completed = true and py.amount > 50 " +

")", Phone.class )

.getResultList();

// Not JPQL compliant!

List<Payment> payments = entityManager.createQuery(

"select distinct p " +

"from Payment p " +

"where ( p.amount, p.completed ) in (" +

" (50, true )," +

" (100, true )," +

" (5, false )" +

")", Payment.class )

.getResultList();

**Exists**

Выражения Exists проверяют наличие результатов подзапроса. Утвердительная форма возвращает true, если результат подзапроса содержит значения. Отрицательная форма возвращает true, если результат подзапроса пуст.

**IS [NOT] EMPTY**

Выражение IS [NOT] EMPTY относится к коллекции многозначных выражений пути. Он проверяет, имеет ли конкретная коллекция какие-либо связанные значения.

Пример 60. Примеры выражения пустой коллекции

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.phones is empty", Person.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.phones is not empty", Person.class )

.getResultList();

**[NOT] MEMBER [OF]**

Выражение [NOT] MEMBER [OF] относится к коллекции многозначных выражений пути. Он проверяет, является ли значение членом указанной коллекции.

Пример 61. Примеры выражений членов коллекции

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where 'Home address' member of p.addresses", Person.class )

.getResultList();

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where 'Home address' not member of p.addresses", Person.class )

.getResultList();

**NOT**

Оператор NOT используется для отрицания предиката , который следует за ним. Если этот следующий предикат равен true, NOT преобразуется в false.

|  |  |
| --- | --- |
|  | Если предикат истинен, NOT разрешается в ложь. Если предикат неизвестен (например NULL), NOT также разрешается в неизвестное. |

**AND**

Оператор AND используются для объединения 2 предикатных выражений. Результат выражения AND является истинным тогда и только тогда, когда оба предиката преобразуются в истинные. Если один из предикатов разрешается в неизвестное, выражение AND также разрешается в неизвестное. В противном случае результат будет ложным.

**Оператор предиката ИЛИ**

Оператор OR используются для объединения 2 предикатных выражений. Результат выражения OR равен true, если один предикат разрешается в true. Если оба предиката разрешаются в неизвестное, выражение ИЛИ разрешается в неизвестное. В противном случае результат будет ложным.

**WHERE/HAVING**

Предложение WHERE запроса состоит из предикатов, которые утверждают, соответствуют ли значения в каждой потенциальной строке текущим критериям фильтрации. Таким образом, предложение where ограничивает результаты, возвращаемые запросом select, и ограничивает объем запросов на обновление и удаление.

**GROUP BY**

Предложение GROUP BY позволяет создавать агрегированные результаты для различных групп значений. В качестве примера рассмотрим следующие запросы:

Пример 62. Группировка по примеру

Long totalDuration = entityManager.createQuery(

"select sum( c.duration ) " +

"from Call c ", Long.class )

.getSingleResult();

List<Object[]> personTotalCallDurations = entityManager.createQuery(

"select p.name, sum( c.duration ) " +

"from Call c " +

"join c.phone ph " +

"join ph.person p " +

"group by p.name", Object[].class )

.getResultList();

//It's even possible to group by entities!

List<Object[]> personTotalCallDurations = entityManager.createQuery(

"select p, sum( c.duration ) " +

"from Call c " +

"join c.phone ph " +

"join ph.person p " +

"group by p", Object[].class )

.getResultList();

Первый запрос возвращает полную сумму всех заказов. Второе извлекает итоговое значение для каждого клиента, сгруппированное по каждому клиенту.

В групповом запросе предложение where применяется к неагрегированным значениям (по сути, оно определяет, попадут ли строки в агрегацию).

Предложение **HAVING** также ограничивает результаты, но оно работает с агрегированными значениями. В [Группе по примеру](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html#hql-group-by-example) мы получили Call итоги продолжительности для всех людей. Если это окажется слишком большим объемом данных для обработки, мы можем ограничить результаты, ориентируясь только на клиентов с общей суммой более 1000:

Пример 63. Имея пример

List<Object[]> personTotalCallDurations = entityManager.createQuery(

"select p.name, sum( c.duration ) " +

"from Call c " +

"join c.phone ph " +

"join ph.person p " +

"group by p.name " +

"having sum( c.duration ) > 1000", Object[].class )

.getResultList();

Предложение HAVING следует тем же правилам, что и WHERE предложение, и также состоит из предикатов. **HAVING** применяется **после того, как группирование и агрегация были выполнены,** в то время как**WHERE** предложение применяется **до**.

**ORDER BY (сортировать по)**

Предложение ORDER BY используется для указания выбранных значений, которые будут использоваться для упорядочения результата. Типы выражений, которые считаются действительными как часть ORDER BY предложения, включают:

* поля состояния
* атрибуты компонента / встраиваемые
* скалярные выражения, такие как арифметические операции, функции и т. д.
* переменная-идентификатор, объявленная в предложении select для любого из предыдущих типов выражений

**Кроме того, JPQL говорит, что все значения, указанные в ORDER BY предложении, должны быть названы в SELECT предложении**. HQL не предписывает это ограничение, но приложения, которым требуется переносимость базы данных, должны знать, что не все базы данных поддерживают ссылки на значения в ORDER BY предложении, на которые нет ссылки в предложении select.

Отдельные выражения в порядке упорядочения могут быть квалифицированы как ASC(по возрастанию) или DESC(по убыванию), чтобы указать желаемое направление упорядочения. Нулевые значения могут быть помещены в начале или в конце отсортированного набора с помощью NULLS FIRST или NULLS LAST предложения соответственно.

Пример 64. Сортировка по примеру

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"order by p.name", Person.class )

.getResultList();

List<Object[]> personTotalCallDurations = entityManager.createQuery(

"select p.name, sum( c.duration ) as total " +

"from Call c " +

"join c.phone ph " +

"join ph.person p " +

"group by p.name " +

"order by total", Object[].class )

.getResultList();

**Режим “Только для чтения” (**для JPQLsetHint или [@QueryHint](http://docs.oracle.com/javaee/7/api/javax/persistence/QueryHint.html)**,** для HQL еще иsetReadOnly**)**

Как объяснено в разделе [неизменяемости сущностей](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/chapters/domain/immutability.html#entity-immutability) , выборка сущностей в режиме только для чтения намного эффективнее, чем выборка сущностей для чтения и записи. Даже если объекты являются изменяемыми, вы все равно можете получить их в режиме только для чтения и получить выгоду от уменьшения занимаемой памяти и ускорения процесса очистки.

Одним из параметров запроса указываем setHint, в котором прописываем "org.hibernate.readOnly":

Пример 65. Пример запроса сущностей только для чтения

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"join c.phone p " +

"where p.number = :phoneNumber ", Call.class )

.setParameter( "phoneNumber", "123-456-7890" )

.setHint( "org.hibernate.readOnly", true )

.getResultList();

calls.forEach( c -> c.setDuration( 0 ) );

SELECT c.id AS id1\_5\_ ,

c.duration AS duration2\_5\_ ,

c.phone\_id AS phone\_id4\_5\_ ,

c.call\_timestamp AS call\_tim3\_5\_

FROM phone\_call c

INNER JOIN phone p ON c.phone\_id = p.id

WHERE p.phone\_number = '123-456-7890'

Как видите, SQL UPDATE не выполняется.

Вы также можете передать подсказку только для чтения именованным запросам, используя JPA аннотацию  [@QueryHint](http://docs.oracle.com/javaee/7/api/javax/persistence/QueryHint.html), в которую в параметре *“name”* прописать "org.hibernate.readOnly".

Пример 66. Извлечение объектов только для чтения с использованием именованного запроса и подсказки только для чтения

@NamedQuery(

name = "get\_read\_only\_person\_by\_name",

query = "select p from Person p where name = :name",

hints = {

@QueryHint(

name = "org.hibernate.readOnly",

value = "true"

)

}

)

API Hibernate предлагает собственный Query#setReadOnly метод в качестве альтернативы использованию подсказки запроса JPA:

Пример 67. Пример собственного запроса только для чтения сущностей

List<Call> calls = entityManager.createQuery(

"select c " +

"from Call c " +

"join c.phone p " +

"where p.number = :phoneNumber ", Call.class )

.setParameter( "phoneNumber", "123-456-7890" )

.unwrap( org.hibernate.query.Query.class )

.setReadOnly( true )

.getResultList();

Последнее обновление 2018-11-16