**JPQL и CRITERIA**

<https://dou.ua/lenta/articles/jpa-fetch-types/>

Понимание стратегий загрузки коллекций в JPA и Hibernate является ключевым для производительности приложения, использующего ORM.

Отношениям один-ко-многим или многие-ко-многим между таблицами реляционной базы данных в объектном виде соответствуют свойства сущности типа List или Set, размеченные аннотациями @OneToMany или @ManyToMany. При работе с сущностями, которые содержат коллекции других сущностей, возникает проблема известная как «N+1 selects». Первый запрос выберет только корневые сущности, а каждая связанная коллекция будет загружена отдельным запросом. Таким образом, ORM выполняет N+1 SQL запросов, где N — количество корневых сущностей в результирующей выборке запроса.

В данной статье будут рассмотрены детали различных типов и стратегий загрузки коллекций в JPA, а в следующей части — режимы загрузки коллекций в Hibernate.

Два типа загрузки

В JPA есть 2 типа загрузки (FetchType): EAGER and LAZY.

EAGER загрузка заставляет ORM загружать связанные сущности и коллекции сразу, вместе с корневой сущностью. LAZY загрузка означает, что ORM загрузит сущность или коллекцию отложено, при первом обращении к ней из кода.

FetchType в JPA говорит когда мы хотим, чтоб связанная сущность или коллекция была загружена. По умолчанию JPA провайдер загружает связанные коллекции (отношения один-ко-многим и многие-ко-многим) отложено (lazy loading). В большинстве случаев отложенная загрузка — оптимальный вариант. Нет смысла инициализировать все связанные коллекции, если к ним не будет обращений.

Метод *FetchType*определяет две стратегии извлечения данных из базы данных:

* ***FetchType.EAGER*:**Поставщик персистентности должен загрузить соответствующий

аннотированное поле или свойство.

* ***FetchType.LAZY*:**Поставщик данных должен загружать данные, когда

к нему первый доступ, но его можно загрузить с нетерпением.

**Для JPA по умолчанию:**

***EAGER*** для аннотированных полей ***@Basic****,* ***@ManyToOne***и ***@OneToOne***.

***LAZY*** для аннотированных полей ***@OneToMany****,* ***@ManyToMany*** и ***@ElementCollection***.

**Для Hibernate по умолчанию:**

все загрузки только ***LAZY***

Две стратегии загрузки связанных коллекций

JPA предоставляет две основных стратегии загрузки: SELECT и JOIN.

1. Стратегия загрузки SELECT является стратегией “по-умолчанию”. Она загружает связанные коллекции отдельными SQL запросами. Иногда эта стратегия может негативно повлиять на производительность, особенно, когда в результирующей выборке большое количество элементов. Эту проблему часто называют «N+1 selects».

**2. Стратегия JOIN указывает ORM, что загружать связанные коллекции необходимо в одном SQL запросе с корневой сущностью, используя оператор LEFT JOIN в сгенерированном SQL запросе**.

Часто эта стратегия лучше с точки зрения производительности, особенно, когда в результирующей выборке большое количество элементов. Конечно, при условии, что в дальнейшем к загруженным коллекциям будут обращения в коде.

Есть несколько способов указать ORM **использовать стратегию загрузки JOIN**: **в** **JPQL** это оператор**JOIN FETCH**, **в** JPA **Criteria** это метод **fetch** класса Root , + **entity graph** добавленные в JPA 2.1.

У стратегии загрузки JOIN есть и недостатки:

- JPQL и JPA Criteria запросы со стратегией загрузки JOIN возвращают декартово произведение (cartesian product). Это значит, что если корневая сущность содержит связанную коллекцию с 3-мя элементами, результирующая выборка будет иметь размер 3. Оператор DISTINCT может использоваться, чтобы этого избежать. Он уберет все дублирующиеся строки из результирующей выборки. Но, если результат может содержать дубликаты и это ожидаемо, оператор DISTINCT все равно уберет их.

- Только одна связанная коллекция, которая загружается стратегией JOIN может быть типа java.util.List, остальные коллекции должны быть типа java.util.Set. В обратном случае, будет выброшено исключение:

HibernateException: cannot simultaneously fetch multiple bags

- При использовании стратегии загрузки JOIN методы setMaxResults и setFirstResult не добавят необходимых условий в сгенерированный SQL запрос. Результат SQL запроса будет содержать все строки без ограничения и смещения согласно firstResult/maxResults. Ограничение количества и смешение строк будет применено в памяти. Также будет выведено предупреждение:

**WARN HHH000104:** firstResult/maxResults specified with collection fetch; applying in memory!

Пример

Давайте для примера рассмотрим следующую модель. Сущность Book владеет отношениями многие-ко-многим с сущностями Author и Category(т.е. будут созданы еще 2 промежуточные таблицы со связкой по “id” для каждого отношения). Пример целиком доступен [на Github](https://github.com/evgeniy-khist/examples/tree/master/jpa-hibernate-fetching-strategies/jpa-fetching-strategies).

@Entity

public **class** **Book** **implements** **Serializable** {

@Id

@GeneratedValue

**private** Long id;

**private** String isbn;

**private** String title;

@Temporal(TemporalType.DATE)

**private** Date publicationDate;

@ManyToMany(fetch = FetchType.EAGER)//тут я говорю, чтобы сущность Author загрузилась сразу

вместе с сущностью Book

**private** List<Author> authors = **new** ArrayList();

@ManyToMany

**private** List<Category> categories = **new** ArrayList();

/\*...\*/

}

@Entity

public **class** **Author** **implements** **Serializable** {

@Id

@GeneratedValue

**private** Long id;

**private** String fullName;

@ManyToMany(mappedBy = "authors")

**private** List<Book> books = **new** ArrayList();

/\*...\*/

}

@Entity

public **class** **Category** **implements** **Serializable** {

@Id

@GeneratedValue

**private** Long id;

**private** String name;

**private** String description;

/\*...\*/

}

Давайте добавим тестовых данных.

Category softwareDevelopment = **new** Category();

softwareDevelopment.setName("Software development");

em.persist(softwareDevelopment);

Category systemDesign = **new** Category();

systemDesign.setName("System design");

em.persist(systemDesign);

Author martinFowler = **new** Author();

martinFowler.setFullName("Martin Fowler");

em.persist(martinFowler);

Book poeaa = **new** Book();

poeaa.setIsbn("007-6092019909");

poeaa.setTitle("Patterns of Enterprise Application Architecture");

poeaa.setPublicationDate(df.parse("2002/11/15"));

poeaa.setAuthors(asList(martinFowler));

poeaa.setCategories(asList(softwareDevelopment, systemDesign));

em.persist(poeaa);

Author gregorHohpe = **new** Author();

gregorHohpe.setFullName("Gregor Hohpe");

em.persist(gregorHohpe);

Author bobbyWoolf = **new** Author();

bobbyWoolf.setFullName("Bobby Woolf");

em.persist(bobbyWoolf);

Book eip = **new** Book();

eip.setIsbn("978-0321200686");

eip.setTitle("Enterprise Integration Patterns");

eip.setPublicationDate(df.parse("2003/10/20"));

eip.setAuthors(asList(gregorHohpe, bobbyWoolf));

eip.setCategories(asList(softwareDevelopment, systemDesign));

em.persist(eip);

Тесты будут запускаться на WildFly 8.2.1.Final с JPA 2.1 провайдером Hibernate 4.3.7.Final.

Поиск по первичному ключу

При поиске сущности по первичному ключу будет использована стратегия загрузки JOIN для коллекций с типом загрузки EAGER. Коллекции с типом загрузки LAZY будут загружены при первом обращении к ним в коде.

Book eip = em.find(Book.class, eipId);

Сгенерированный SQL:

**select**

book0\_.id **as** id1\_1\_0\_,

book0\_.isbn **as** isbn2\_1\_0\_,

book0\_.publicationDate **as** publicat3\_1\_0\_,

book0\_.title **as** title4\_1\_0\_,

authors1\_.books\_id **as** books\_id1\_1\_1\_,

author2\_.id **as** authors\_2\_2\_1\_,

author2\_.id **as** id1\_0\_2\_,

author2\_.fullName **as** fullName2\_0\_2\_

**from**

Book book0\_

**left** **outer** **join**

Book\_Author authors1\_

**on** book0\_.id=authors1\_.books\_id

**left** **outer** **join**

Author author2\_

**on** authors1\_.authors\_id=author2\_.id

**where**

book0\_.id=?

JPQL и JPA Criteria запросы со стратегией «select»

В JPQL запросах стандартной является стратегия загрузки SELECT. Для каждой сущности из списка результатов JQPL запроса будет выполнен дополнительный SQL запрос для загрузки каждой связанной коллекции. Коллекции с типом загрузки LAZY будут загружены при первом обращении к ним в коде.

List<Book> books = em.createQuery("select b from Book b order by b.publicationDate")

.getResultList();

assertEquals(2, books.size());

JPA Criteria запросы по умолчанию имеют такое же поведение, как и JQPL запросы.

CriteriaQuery<Book> cr = cb.createQuery(Book.**class**);

Root<Book> root = cr.from(Book.**class**);

cr.orderBy(cb.asc(root.get(Book\_.publicationDate)));

TypedQuery<Book> q = em.createQuery(cr);

List<Book> books = q.getResultList();

assertEquals(2, books.size());

//*или вместо последних 3-х строк можно*

em.createQuery(cr).getResultList().forEach(System.out::println);

Сгенерированный SQL:

**//1-й запрос по корневой таблице**

**select**

book0\_.id **as** id1\_1\_,

book0\_.isbn **as** isbn2\_1\_,

book0\_.publicationDate **as** publicat3\_1\_,

book0\_.title **as** title4\_1\_

**from**

Book book0\_

**order** **by**

book0\_.publicationDate

**//2-й запрос по связанной, автоматичски созданной, промежуточной таблице со связкой по “id”**

**select**

authors0\_.books\_id **as** books\_id1\_1\_0\_,

authors0\_.authors\_id **as** authors\_2\_2\_0\_,

author1\_.id **as** id1\_0\_1\_,

author1\_.fullName **as** fullName2\_0\_1\_

**from**

Book\_Author authors0\_

**inner** **join**

Author author1\_

**on** authors0\_.authors\_id=author1\_.id

**where**

authors0\_.books\_id=?

**/3-й запрос по связанной таблице “Autor”**

**select**

authors0\_.books\_id **as** books\_id1\_1\_0\_,

authors0\_.authors\_id **as** authors\_2\_2\_0\_,

author1\_.id **as** id1\_0\_1\_,

author1\_.fullName **as** fullName2\_0\_1\_

**from**

Book\_Author authors0\_

**inner** **join**

Author author1\_

**on** authors0\_.authors\_id=author1\_.id

**where**

authors0\_.books\_id=?

JPQL и JPA Criteria запросы с «join fetch»

Чтобы использовать стратегию загрузки JOIN в JQPL запросах, используйте оператор JOIN FETCH. Корневые сущности со связанными коллекциями будут загружены в одном SQL запросе. Результатом запроса будет декартово произведение (cartesian product). Вместо 2 элементов в результирующей выборке, запрос с JOIN FETCH вернет 3, потому что книга «Enterprise Integration Patterns» имеет двух авторов, поэтому будет дважды встречаться в результатах запроса.

List<Book> books = em.createQuery("select b from Book b left join fetch b.authors order by b.publicationDate").getResultList();

assertEquals(3, books.size());

,где “left join fetch” это аналог “OUTER JOIN” в SQL (*загружаются все строки, даже не имеющие соответствия в другой таблице*). “INNER JOIN” ЭТО ПРОСТО “join fetch”.

JPA Criteria запрос, как и JQPL запрос, вернет 3 результата из-за декартова произведения. Чтобы установить стратегию загрузки LEFT JOIN в JPA Criteria запросах необходимо использовать метод fetch **c** JoinType.LEFT. А стратегию загрузки JOIN необходимо использовать просто метод fetch **без** JoinType.LEFT.

CriteriaQuery<Book> cr = cb.createQuery(Book.**class**);

Root<Book> root = cr.from(Book.**class**);

book.fetch(Book\_.authors, JoinType.LEFT);

cr.orderBy(cb.asc(root.get(Book\_.publicationDate))); //где .asc/.desc – по возрастанию/убыванию

TypedQuery<Book> q = em.createQuery(cr);

List<Book> books = q.getResultList();

assertEquals(3, books.size());

Сгенерированный SQL:

**//все в одном запросе**

**select**

book0\_.id **as** id1\_1\_0\_,

author2\_.id **as** id1\_0\_1\_,

book0\_.isbn **as** isbn2\_1\_0\_,

book0\_.publicationDate **as** publicat3\_1\_0\_,

book0\_.title **as** title4\_1\_0\_,

author2\_.fullName **as** fullName2\_0\_1\_,

authors1\_.books\_id **as** books\_id1\_1\_0\_\_,

authors1\_.authors\_id **as** authors\_2\_2\_0\_\_

**from**

Book book0\_ //корневая таблица

**left** **outer** **join**

Book\_Author authors1\_ //автоматичски созданная промежуточная таблица со связкой по “id”

**on** book0\_.id=authors1\_.books\_id

**left** **outer** **join**

Author author2\_ //связанной таблица “Autor”

**on** authors1\_.authors\_id=author2\_.id

**order** **by**

book0\_.publicationDate

JPQL и JPA Criteria запросы с «distinct» и «join fetch»

Оператор DISTINCT удаляет дубликаты из результатов запроса. В этом примере результат JPQL запроса с оператором DISTINCT будет содержать 2 элемента. Это хороший «workaround», когда декартово произведение, которое возвращает JPQL запрос с JOIN FETCH является проблемой.

List<Book> books = em.createQuery("select distinct b from Book b left join fetch b.authors order by b.publicationDate")

.getResultList();

assertEquals(2, books.size());

В JPA Criteria чтобы удалить дубликаты из результатов запроса, используется метод CriteriaQuery#distinct(boolean).

CriteriaQuery<Book> cr = cb.createQuery(Book.**class**);

Root<Book> book = cr.from(Book.**class**);

cr.distinct(true);

book.fetch(Book\_.authors, JoinType.LEFT);

cr.orderBy(cb.asc(book.get(Book\_.publicationDate)));

TypedQuery<Book> q = em.createQuery(cr);

List<Book> books = q.getResultList();

assertEquals(2, books.size());

Сгенерированный SQL:

**select**

**distinct** book0\_.id **as** id1\_1\_0\_,

author2\_.id **as** id1\_0\_1\_,

book0\_.isbn **as** isbn2\_1\_0\_,

book0\_.publicationDate **as** publicat3\_1\_0\_,

book0\_.title **as** title4\_1\_0\_,

author2\_.fullName **as** fullName2\_0\_1\_,

authors1\_.books\_id **as** books\_id1\_1\_0\_\_,

authors1\_.authors\_id **as** authors\_2\_2\_0\_\_

**from**

Book book0\_

**left** **outer** **join**

Book\_Author authors1\_

**on** book0\_.id=authors1\_.books\_id

**left** **outer** **join**

Author author2\_

**on** authors1\_.authors\_id=author2\_.id

**order** **by**

book0\_.publicationDate

JPQL запрос с entity graph (Он моментально (EAGER) загружает любые выбранные (LAZY) поля на уровне запроса)

\* - детально про entity graph смотри ниже

В JPA 2.1 был добавлен новый способ управления JOIN стратегией загрузки — entity graph.

EntityGraph<Book> fetchAuthors = em.createEntityGraph(Book.**class**);

fetchAuthors.addSubgraph(Book\_.authors); //тут я создаю **граф** в котором говорю моментально выгрузить все поля, связанной с полем «authors» *(в сущности Book)* , сущности *(для понимания см. п.4.2 дальше по тексту)*

List<Book> books = em.createQuery("select b from Book b order by b.publicationDate")

.setHint("javax.persistence.fetchgraph", fetchAuthors) //применяю написанный сверху **граф**

.getResultList();

assertEquals(3, books.size());

Сгенерированный SQL:

**select**

book0\_.id **as** id1\_1\_0\_,

author2\_.id **as** id1\_0\_1\_,

book0\_.isbn **as** isbn2\_1\_0\_,

book0\_.publicationDate **as** publicat3\_1\_0\_,

book0\_.title **as** title4\_1\_0\_,

author2\_.fullName **as** fullName2\_0\_1\_,

authors1\_.books\_id **as** books\_id1\_1\_0\_\_,

authors1\_.authors\_id **as** authors\_2\_2\_0\_\_

**from**

Book book0\_

**left** **outer** **join**

Book\_Author authors1\_

**on** book0\_.id=authors1\_.books\_id

**left** **outer** **join**

Author author2\_

**on** authors1\_.authors\_id=author2\_.id

**order** **by**

book0\_.publicationDate

JPQL запрос с «join fetch» нескольких коллекций

Следующее исключение возникнет, если несколько коллекций типа java.util.List загружаются одновременно. Только одна коллекций, которая загружается со стратегией JOIN может быть типа java.util.List, остальные коллекции, которые загружаются стратегией JOIN должны быть типа java.util.Set.

List<Book> books = em.createQuery("select b from Book b left join fetch b.authors left join fetch b.categories")

.getResultList();

Будет выброшено исключение:

org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple bags

Обратите внимание, что загружать несколько коллекций стратегией JOIN — это не всегда оптимальный вариант. Если обе коллекции будут иметь по 100 элементов, SQL запрос вернет 10000 строк. Иногда вместо этого более эффективно выполнить 2 запроса: первый, загружающий первую коллекцию, и второй, загружающий вторую коллекцию. Это значительно уменьшит суммарное количество строк в результатах запросов.

JPQL запрос с «join fetch» и «max results»

Когда используется стратегия загрузки JOIN, методы setMaxResults и setFirstResult не добавят соответствующих условий в сгенерированный SQL запрос. Запрос вернет все строки без ограничений и смещений, указанных в firstResult/maxResults. Вместо этого, ограничения будут применены в памяти. Если фильтрация в памяти вызывает проблемы, **не используйте setFirsResult, setMaxResults и getSingleResult со стратегией загрузки JOIN.**

List<Book> books = em.createQuery("select b from Book b left join fetch b.authors order by b.publicationDate")

.setFirstResult(0)

.setMaxResults(1)

.getResultList();

assertEquals(1, books.size());

Будет выведено предупреждение:  
WARN [org.hibernate.hql.internal.ast.QueryTranslatorImpl] HHH000104: firstResult/maxResults specified with collection fetch; applying in memory!

Сгенерированный SQL:

**select**

book0\_.id **as** id1\_1\_0\_,

author2\_.id **as** id1\_0\_1\_,

book0\_.isbn **as** isbn2\_1\_0\_,

book0\_.publicationDate **as** publicat3\_1\_0\_,

book0\_.title **as** title4\_1\_0\_,

author2\_.fullName **as** fullName2\_0\_1\_,

authors1\_.books\_id **as** books\_id1\_1\_0\_\_,

authors1\_.authors\_id **as** authors\_2\_2\_0\_\_

**from**

Book book0\_

**left** **outer** **join**

Book\_Author authors1\_

**on** book0\_.id=authors1\_.books\_id

**left** **outer** **join**

Author author2\_

**on** authors1\_.authors\_id=author2\_.id

**order** **by**

book0\_.publicationDate

**JPA Entity Graph**

1. Обзор

В JPA 2.1 введена функция Entity Graph, как метод загрузки необходимых, полей/ссылочных полей на связанные сущности, классов-сущностей.

Он моментально (EAGER) загружает любые выбранные (LAZY) поля на уровне запроса.

Я так понимаю, что эта технология используется для моментальной EAGER загрузки полей, вместо *(чаще всего установленной в классах-сущностях)* LAZY загрузки. При чем это делается в коде сразу перед запросом, без необходимость лезть в класс-сущность и менять все там на EAGER, а потом опять назад на LAZY. И выполняется все это в одним запросом (*как при Join Fetch*).

2. Определение модели

Прежде чем мы начнем изучать Entity Graph, нам нужно определить модели объектов, с которыми мы работаем. Допустим, мы хотим создать блог-сайт, где пользователи могут комментировать и делиться постами.

Итак, сначала у нас будет сущность *User*:

@Entity

**public class User {**

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private Long id;**

**private String name;**

**private String email;**

//...

}

Пользователь может делиться различными сообщениями, поэтому нам также нужна сущность *Post*:

@Entity

**public class Post {**

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private Long id;**

**private String subject;**

@OneToMany(mappedBy = "post")

**private List<Comment> comments = new ArrayList<>();**

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn

**private User user;**

//...

}

Пользователь также может комментировать общие записи, поэтому, наконец, мы добавим сущность *Comment*:

@Entity

**public class Comment {**

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private Long id;**

**private String reply;**

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn

**private Post post;**

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn

**private User user;**

//...

}

Как мы видим, сущность *Post*имеет связь с сущностями *Comment*и *User*. Сущность *Comment*имеет связь с сущностями *Post*и *User*.

Цель состоит в том, чтобы загрузить следующий граф различными способами:

Пост -> пользователь: Пользователь

      -> комментарии: список <комментарий>

            комментарии[0]: Комментарий -> пользователь: Пользователь

            комментарии[1]: Комментарий -> пользователь: Пользователь

3. Загрузка связанных сущностей с помощью *FetchType*Strategies

Метод *FetchType*определяет две стратегии извлечения данных из базы данных:

* ***FetchType.EAGER*:**Поставщик персистентности должен загрузить соответствующий

аннотированное поле или свойство. Это поведение по умолчанию для аннотированных полей *@ Basic, @ManyToOne*и *\_ @ OneToOne \_*.

* ***FetchType.LAZY*:**Поставщик данных должен загружать данные, когда

к нему первый доступ, но его можно загрузить с нетерпением. Это поведение по умолчанию для аннотированных полей *@ OneToMany, @ManyToMany*и *\_ @ ElementCollection - \_*.

Например, когда мы загружаем сущность *Post*, связанные сущности *Comment*не загружаются по умолчанию, поскольку у *@ OneToMany* *FetchType*равен *LAZY.*

Мы можем переопределить это поведение, изменив *FetchType*на *EAGER:*

**@OneToMany(mappedBy = "post", fetch = FetchType.EAGER)**

**private List<Comment> comments = new ArrayList<>();**

Для сравнения, когда мы загружаем сущность *Comment*, его родительская сущность *Post* загружается сразу, т.к. режим по умолчанию для *@ManyToOne это which EAGER.*

*Мы также можем не загружать сущность Post , изменив эту аннотацию на LAZY: \_*

**@ManyToOne(fetch = FetchType.LAZY)**

**@JoinColumn(name = "post\_\_id")**

**private Post post;**

* Обратите внимание, что, поскольку *LAZY*не является обязательным требованием, поставщик сохраняемости может по-прежнему сразу загружать сущность *Post*, если хочет. \*\* Поэтому для правильного использования этой стратегии мы должны вернуться к официальной документации соответствующего поставщика сохраняемости.

Теперь, поскольку мы использовали аннотации для описания нашей стратегии выборки, **наше определение статично и нет способа переключаться с *LAZY*на *EAGER*во время выполнения**.

Это то место где Entity Graph вступает в игру, как мы увидим в следующем разделе.

**4. Определение графа сущностей**

Чтобы определить Entity Graph, мы можем либо использовать аннотации к сущности, либо мы можем программно работать с использованием JPA API.

**4.1. Именованные графы -** **определение графа сущностей с аннотациями (используется если известно, что именно так такие поля будут многократно вызываться. Иначе проще вариант 4.2)**

* Аннотация @ *NamedEntityGraph*позволяет указывать атрибуты, которые необходимо включить, когда мы хотим загрузить объект и связанные с ним ассоциации. \*\*

Итак, давайте сначала определим Entity Graph для сущности Post, который загружает необходимые поля:

**@NamedEntityGraph(**

**name = "post-entity-graph",**//наименование графа для данной сущности

**attributeNodes = {** //в нем дальше написаны имена полей, которые будут загружены если выбрать стратегией загрузки этот граф

**@NamedAttributeNode("subject"),**

**@NamedAttributeNode("user"),**

**@NamedAttributeNode("comments"),**

**}**

**)**

**@Entity**

**public class Post {**

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

**private List<Comment> comments = new ArrayList<>();**

**//...**

**}**

В этом примере мы использовали *@ NamedAttributeNode*для определения связанных сущностей, которые будут загружены при загрузке корневого объекта.

Давайте теперь определим более сложный Entity Graph, где мы будем загружать поля “subject”, “user” класса Post, а также поле "user" из класса Comment, используя для связки поле "comments" (**value = "comments"**) класса Post.

Для этой цели мы будем использовать атрибут *@ NamedAttributeNode subgraph.****Это позволяет ссылаться на именованный подграф, определенный с помощью аннотации****@ NamedSubgraph*:

**@NamedEntityGraph(**

**name = "post-entity-graph-with-comment-users",**

**attributeNodes = {**

**@NamedAttributeNode("subject"),**

**@NamedAttributeNode("user"),**

**@NamedAttributeNode(value = "comments", subgraph = "comments-subgraph"),**

**},**

**subgraphs = {**

**@NamedSubgraph(**

**name = "comments-subgraph",**

**attributeNodes = {**

**@NamedAttributeNode("user")**

**}**

**)**

**}**

**)**

**@Entity**

**public class Post {**

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

**private List<Comment> comments = new ArrayList<>();**

**//...**

}

Определение аннотации *@ NamedSubgraph*аналогично @ *NamedEntityGraph*и позволяет указывать атрибуты связанной ассоциации. Поступая так, мы можем построить полный граф.

В приведенном выше примере с помощью определенного графа ‘ *post-entity-graph-with-comment-users'*мы можем загрузить *Post,*связанное *User, Comments*и *\_User , связанные с Comments.\_*

Наконец, обратите внимание, что мы можем альтернативно добавить определение Entity Graph, используя дескриптор развертывания *orm.xml*:

**<entity-mappings>**

**<entity class="com.baeldung.jpa.entitygraph.Post" name="Post">**

**...**

**<named-entity-graph name="post-entity-graph">**

**<named-attribute-node name="comments"/>**

**</named-entity-graph>**

**</entity>**

**...**

**</entity-mappings>**

**4.2. Динамические графы - определение графа сущностей в коде, непосредственно перед запросом, с помощью JPA API**

Мы также можем определить *Entity Graph* через *API EntityManager*, вызвав метод *createEntityGraph ():*

**EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);**

Чтобы указать переменые корневого объекта которые будут загружены сразу, мы используем метод *addAttributeNodes ():*

**entityGraph.addAttributeNodes("subject");**

**entityGraph.addAttributeNodes("user");**

, и я так понял, что если эти две последние строчки кода не писать, то все переменные класса Post будут загружены как EAGER

Точно так же как и в п.4.1., чтобы включить атрибуты из связанной сущности, мы используем *addSubgraph ()*для создания встроенного графа сущностей, а затем мы *\_addAttributeNodes () \_*, как мы делали выше:

**entityGraph.addSubgraph("comments")**

**.addAttributeNodes("user");**

Теперь, когда мы увидели, как создать Entity Graph, мы рассмотрим, как использовать его в следующем разделе.

**5. Использование Entity Graph**

Итак давайте начнем с использования обычного метода *EntityManager.find*()и потом просмотрим журнал:

**Post post = entityManager.find(Post.class, 1L);**

Вот журнал, предоставленный реализацией Hibernate:

**select**

**post0\_\_.id as id1\_\_1\_\_0\_\_,**

**post0\_\_.subject as subject2\_\_1\_\_0\_\_,**

**post0\_\_.user\_\_id as user\_\_id3\_\_1\_\_0\_\_**

**from**

**Post post0\_\_**

**where**

**post0\_\_.id=?**

Как мы видим из журнала, классы-сущности *User*и *Comment* не загружены т.к. в классе-сущности Post тип загрузки ссылочных переменных **comments** и **user** обозначен как LAZY.

Мы можем получить Entity Graph различными способами. Далее представлены 3 возможных типов загрузок графов сущностей:

**1).** Мы можем переопределить это поведение по умолчанию, вызвав перегруженный метод *find ()*, который принимает подсказки как *Map.*Затем мы можем **предоставить тип графа, который мы хотим загрузить:**

**EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph"); //где "post-entity-graph" это имя именованного графа, созданного в п.4.1**

**Map<String, Object> properties = new HashMap<>();**

**properties.put("javax.persistence.fetchgraph", entityGraph);**

**Post post = entityManager.find(Post.class, id, properties);**

Если мы посмотрим снова в журнале, мы увидим, что эти объекты теперь загружены и только в одном запросе выбора:

**select**

**post0\_\_.id as id1\_\_1\_\_0\_\_,**

**post0\_\_.subject as subject2\_\_1\_\_0\_\_,**

**post0\_\_.user\_\_id as user\_\_id3\_\_1\_\_0\_\_,**

**comments1\_\_.post\_\_id as post\_\_id3\_\_0\_\_1\_\_,**

**comments1\_\_.id as id1\_\_0\_\_1\_\_,**

**comments1\_\_.id as id1\_\_0\_\_2\_\_,**

**comments1\_\_.post\_\_id as post\_\_id3\_\_0\_\_2\_\_,**

**comments1\_\_.reply as reply2\_\_0\_\_2\_\_,**

**comments1\_\_.user\_\_id as user\_\_id4\_\_0\_\_2\_\_,**

**user2\_\_.id as id1\_\_2\_\_3\_\_,**

**user2\_\_.email as email2\_\_2\_\_3\_\_,**

**user2\_\_.name as name3\_\_2\_\_3\_\_**

**from**

**Post post0\_\_**

**left outer join**

**Comment comments1\_\_**

**on post0\_\_.id=comments1\_\_.post\_\_id**

**left outer join**

**User user2\_\_**

**on post0\_\_.user\_\_id=user2\_\_.id**

**where**

**post0\_\_.id=?**

**2).** Давайте посмотрим, как мы можем достичь того же, используя JPQL:

//Именованный граф

**EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph-with-comment-users");**

//или Динамический граф (*выбираем поля для моментальной (не LAZY) загрузки тут-же, а не загружаем написанный граф из класса*):

**EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);**

**entityGraph.addAttributeNodes("subject");**

**entityGraph.addAttributeNodes("user");**

**entityGraph.addSubgraph("comments").addAttributeNodes("user");**

**Post post = entityManager.createQuery("select p from Post p where p.id = :id", Post.class)**

**.setParameter("id", id)**

**.setHint("javax.persistence.fetchgraph", entityGraph)**

**.getSingleResult();**

**3).** И, наконец, давайте рассмотрим пример API *Criteria*:

//Именованный граф

**EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph");**

//или выбираем поля для моментальной (не LAZY) загрузки тут-же, а не загружаем написанный граф из класса (Динамический граф):

**EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);**

**entityGraph.addAttributeNodes("subject");**

**entityGraph.addAttributeNodes("user");**

**entityGraph.addAttributeNodes("comments");**

**CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();**

**CriteriaQuery<Post> criteriaQuery = criteriaBuilder.createQuery(Post.class);**

**Root<Post> root = criteriaQuery.from(Post.class);**

**criteriaQuery.where(criteriaBuilder.equal(root.<Long>get("id"), id));**

**TypedQuery<Post> typedQuery = entityManager.createQuery(criteriaQuery);**

**typedQuery.setHint("javax.persistence.loadgraph", entityGraph);**

**Post post = typedQuery.getSingleResult();**

В каждой из загрузок **тип графика указан как подсказка**. В то время как в первом примере мы использовали *Map,*в двух последующих примерах мы использовали метод *setHint ()*.

6. Заключение

В этой статье мы рассмотрели использование графического объекта JPA для динамического извлечения *Entity*и его ассоциаций.

Решение принимается во время выполнения, в котором мы выбираем загружать или нет связанную ассоциацию.

Очевидно, что производительность является ключевым фактором, который необходимо учитывать при проектировании объектов JPA. Документация JPA рекомендует использовать стратегию *FetchType.LAZY*, когда это возможно, и Entity Graph, когда нам нужно загрузить связь.

Выводы

Как вы могли убедиться, в JPA существует множество нюансов, связанных со стратегиями загрузки. JPA 2.1 предоставляет множество способов управления загрузкой связанных коллекций.

*deprecated* Но самая популярная реализация JPA, Hibernate предоставляет еще больше способов управления загрузкой отношений один-ко-многим и многие-ко-многим. FetchMode в Hibernate говорит как мы хотим, чтоб связанные сущности или коллекции были загружены: используя по дополнительному SQL запросу на коллекцию, в одном запросе с корневой сущностью, используя JOIN, или в дополнительном запросе, используя SUBSELECT. Об этом и других средствах загрузки связанных коллекций, которые предоставляет Hibernate, поговорим в следующей части.

[**[https://s.dou.ua/img/avatars/40x40_me21_moUEx5F.jpg](https://dou.ua/users/sergey-morenets/)Sergey Morenets**](https://dou.ua/users/sergey-morenets/) Java evangelist[24.07.2016 03:01](https://dou.ua/lenta/articles/jpa-fetch-types/#958470)

Есть интересный аспект, который остался неотвеченным. Если вы укажете тип загрузки как LAZY, и попытаетесь обратиться к коллекции при закрытой сессии, то получите исключение. Интересно было бы узнать, как автор борется с исключением либо как обходит его.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_15032270_207028116404678_7812941592273916947_n.jpg](https://dou.ua/users/evgeniykhist/)Evgeniy Khyst**](https://dou.ua/users/evgeniykhist/) Java Solution Architect[27.07.2016 17:53](https://dou.ua/lenta/articles/jpa-fetch-types/#961040)

Хороший вопрос, полноценный ответ на который заслуживает отдельной статьи.

Например, есть свойство hibernate.enable\_lazy\_load\_no\_trans, которое можно добавить в persistence.xml. Добавив это свойство, даже после коммита транзакии и закрытия сессии, в которой был загружен объект, к его коллекциям с типом загрузки LAZY можно будет обращаться.

Пример persistence.xml:

<**persistence** version="2.1" xmlns="..." xsi:schemaLocation="...">

<**persistence-unit** name="myexample">

<**jta-data-source**>java:/myexampleDS</**jta-data-source**>

<**properties**>

<**property** name="hibernate.hbm2ddl.auto" value="update"/>

<**property** name="hibernate.enable\_lazy\_load\_no\_trans" value="true"/>

</**properties**>

</**persistence-unit**>

</**persistence**>

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_me21_moUEx5F.jpg](https://dou.ua/users/sergey-morenets/)Sergey Morenets**](https://dou.ua/users/sergey-morenets/) Java evangelist[27.07.2016 20:07](https://dou.ua/lenta/articles/jpa-fetch-types/#961123)

Спасибо. А в чем принципиальное отличие этого решения от OpenEntityManagerInViewFilter?

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_15032270_207028116404678_7812941592273916947_n.jpg](https://dou.ua/users/evgeniykhist/)Evgeniy Khyst**](https://dou.ua/users/evgeniykhist/) Java Solution Architect[27.07.2016 23:34](https://dou.ua/lenta/articles/jpa-fetch-types/#961282)

[OpenEntityManagerInViewFilter](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/support/OpenEntityManagerInViewFilter.html) — это класс из Spring Framework, соответственно в Java EE приложениях без Spring его использовать не получится. Более того, на мой взгляд, свойство в persistece.xml выглядит менее инвазивно, чем специальный фильтр.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_167107.jpg](https://dou.ua/users/vitaliy-rozumyak/)Vitaliy Rozumyak**](https://dou.ua/users/vitaliy-rozumyak/) Android developer[08.12.2016 16:31](https://dou.ua/lenta/articles/jpa-fetch-types/#1028424)

Ось такого посту дуже не вистачало. Проблема у тому що ці та інші граблі доволі часті для розробників початківців, а ресурси часто дають лише шаблонні уривки з документації, а моменти на які явно варто звернути увагу — опускають.  
На рахунок

hibernate.enable\_lazy\_load\_no\_trans

 — пишуть що можливі проблеми, але як зрозумів більше для версій 4.x.x. Більшість пофікшені. Хтось взагалі називає Антипаттерном. Користуюсь hibernate 5.1.2. Є якісь критичні моменти з використанням цієї функції?  
Спасибі, Євген!

[Ответить](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_254652.jpg](https://dou.ua/users/igor-dmitriev-3/)Igor Dmitriev**](https://dou.ua/users/igor-dmitriev-3/) Engineer Manager в [SPD-Ukraine](https://jobs.dou.ua/companies/spd-ukraine/)[21.07.2016 15:13](https://dou.ua/lenta/articles/jpa-fetch-types/#957158)

Хорошая статья, хочу добавить от себя:  
1. EAGER — полное зло и его лучше избегать. EAGER приводит к неявным N+1 в HQL запросах. Так же плох тем что если у тебя есть две ситуации: в одной ты хочешь получить Book с authors то все хорошо, но если тебе просто нужен обьект book, а у тебя всегда будут подтягиваться авторы, когда тебе это не нужно в твоей ситуации. Я уже не говорю о том что если в авторе есть еще EAGER поля и в них тоже могут быть EAGER, и получается что достается большой граф обьектов. Проблема в том что с LAZY в EAGER можно сделать, но наоборот — к сожалению нет, а так же управление фетчингом на уровне ентити это глобальная конфигурация и вы не можете поменять ее в рантайме, в отличии от EntityGraph, где можно динамически управлять стратегий фетчинга на уровне запроса.  
2. оператор DISTINCT на самом деле убирает дубликаты на стороне сервера, ведь данные запроса летят на сервер в любом случае, хибернейт когда маршалит обьекты убирает дубликаты, это так же важно понимать. А так же можно обернуть результат в Set.  
3. Хорошо что были показаны сравнения HQL и Criteria Builder API, но не показали что будет если использовать Hibernate Criteria, а на самом деле результат отличается от Criteria Builder. Так как если в Book есть EAGER поля то Hibernate Criteria для запроса списка книг сразу сгенерирует JOIN в отличие от Criteria Builder, где EAGER поля подтянутся отдельными запросами.  
4. И круто было бы если показать решения проблемы HHH000104: firstResult/maxResults specified with collection fetch; applying in memory! А оно есть.  
5. Так же есть совет чтоб избежать Cartesian Product — нужно следовать простому правилу и все будет хорошо — нельзя делать join fetch или entity graph на больше чем одну коллекцию, на many-to-one и one-to-one можно делать много джоинов. Следуя этому простому правилу все будет хорошо.  
6. Так же полезно знать про Batch Fetching и FetchMode Subselect.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/25x25_15032270_207028116404678_7812941592273916947_n.jpg](https://dou.ua/users/evgeniykhist/)Evgeniy Khyst**](https://dou.ua/users/evgeniykhist/) Java Solution Architect[27.07.2016 17:19](https://dou.ua/lenta/articles/jpa-fetch-types/#961013)

Я не могу рекомендовать никогда не использовать EAGER. Все зависит от конкретной ситуации. Мой подход — знать все варианты и проанализировав ситуацию выбрать оптимальный.

Я специально добавил примеры на GitHub, чтоб каждый смог убедиться, что JPQL оператор DISTINCT транслируется в SQL оператор DISTINCT и сам Hibernate никакие дубликаты уже не удаляет, так как SQL запрос возвращает только уникальные записи. Можно обернуть результат в Set, но тогда вы должны не забыть реализовать equals и hashcode и будете получать по сети, если база данных на другом узле в сети, дубликаты.

[Ответить](javascript:;)

[Поддержать](javascript:;)

[**[https://s.dou.ua/img/avatars/40x40_23_YhkdNag.jpg](https://dou.ua/users/nazar-sobchuk/)Nazar Sobchuk**](https://dou.ua/users/nazar-sobchuk/) Atlassian Jira Expert[21.07.2016 14:08](https://dou.ua/lenta/articles/jpa-fetch-types/#957104)

Спасибо за статью :)

LAZY загрузка означает, что ORM загрузит сущность или коллекцию отложено, при первом обращении к ней из кода.

Это не всегда так. JPA может вернуть mock объект, а саму entity подгрузить именно тогда, когда на mock объекте будет вызван метод, а не при первом обращении к сущности / коллекции.