Стратегии загрузки графа объектов в JPA (Часть 2 и вывод)

 02/11/2016

[В первой части](http://akorsa.ru/2016/10/strategii-zagruzki-grafa-obektov-v-jpa-chast-1/) мы рассмотрели статические стратегии загрузки графа объектов. Теперь рассмотрим динамические и сделаем выводы.

Примеры и тесты доступны на [гитхаб](https://github.com/Akorsa/jpa-object-graphs).

Стратегии динамической загрузки

Стратегии динамической загрузки различаются тем, насколько они динамичны на самом деле. Вообще говоря, все эти стратегии позволяют Вам **решить в рантайме какие из частей графа нужно выгрузить**, в то время как вышеупомянутые стратегии не поддерживают этого, потому что они сконфигурированы через маппинг во время компиляции или через константные запросы в виде строк.

**Именованные графы сущностей**

Единственный способ для определения нескольких стратегий динамической загрузки это использовать NamedEntityGraph, который может быть сконфигурирован как аннотация над сущностью, например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @NamedEntityGraph(name = "user.cars", attributeNodes = @NamedAttributeNode("cars")  @Entity  @Table(name = "usr")  public class User {  } |

Граф сущности имеет два важных атрибута: уникальное имя и attributeNodes, который описывает части графа объектов, которые нужно загрузить. Данный граф задает то, что всякий раз когда загружается сущность user, его машины должны быть тоже выгружены. Почему это динамически происходит? Я должен задать граф как аннотацию и он не может измениться в рантайме. Теперь я покажу, как использовать этот граф. Вы можете использовать его на каждой операции по разному:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test  public void testInitCarsByEntityGraph() {      User user = entityManager.find(User.class, 1L,      Collections.singletonMap("javax.persistence.fetchgraph",          entityManager.getEntityGraph("user.cars")));      user.getCars().stream().forEach(System.out::println);    } |

Есть свойство javax.persistence.fetchgraph позволяющая Вам передать данный граф сущностей для использования в операции find. В этом примере, мы использовали метод find класса EntityManager. Вы так же можете установить граф, когда используете запрос JPQL:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Test  public void testInitByNamedEntityGraphInJPQL() {      TypedQuery<User> loadUserQuery = entityManager.createQuery("select usr from User usr         where usr.id = :id", User.class);      loadUserQuery.setParameter("id", 1L);      loadUserQuery.setHint("javax.persistence.fetchgraph",           entityManager.getEntityGraph("user.cars"));      User user = loadUserQuery.getSingleResult();      user.getCars().stream().forEach(System.out::println);    } |

Граф в этом примере передается через метод setHint, который задан в классе TypedQuery.

Вы так же можете иметь несколько разных графов и выбрать нужный граф, в зависимости от некоторых условий, или Вы можете передать имя графа как параметр для репозитория или DAO. Именно поэтому я категоризирую эти графы как динамические.

Однако, именованные графы все еще имеют лимит, потому что я могу использовать только один граф на одну операцию одновременно.

Не позволяется смешивать их. Следовательно, Вы потенциально можете столкнуться с огромным количеством разных графов, представляющих все комбинации ребер графа, которые должны быть загружены. Если у Вас именно такой случай, то Вы должны использовать динамические графы сущностей.

Динамические графы сущностей

Динамические графы объектов могут быть проинициализированы в рантайме. Если Ваше приложение должно быть очень гибким при загрузке разных графов, Вам нужно выбрать эту концепцию. Следующий тест покажет, как использовать их:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test  public void testInitDynamicEntityGraph() {      EntityGraph<User> graph = entityManager.createEntityGraph(User.class);          graph.addAttributeNodes("cars");      User user = entityManager.find(User.class, 1L,      Collections.singletonMap("javax.persistence.fetchgraph", graph));      user.getCars().stream().forEach(System.out::println);    } |

В этом примере мы создаем новый EntityGraph через EntityManager.createEntityGraph. Для этого графа мы добавляем attributeNodes  = cars. Инстанс затем передается в javax.persistence.fetchgraph свойство опять, как и в примере именованного графа сущностей.

Criteria API

Самый мощный и гибкий способ задавать запросы с JPA это использовать JPA Criteria API. Criteria API позволяет Вам задавать целый запрос динамически в рантайме. Типичный пример использования — фильтр для поиска. Представьте, что у Вас есть разные атрибуты фильтра и всякий раз когда Вы записываете какое-то значение в форму для фильтра Вы хотите добавить тест, в котором этот атрибут проверяется на равенство заданному значению из части «where» запроса. Самый очевидный подход связан  с конкатенации строк. Но это не безопасно и вероятно неэффективно. Следовательно, Вы можете использовать JPA Criteria API, как показано в примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Test  public void testInitByExplicitFetchJoinInJPACriteria() {      CriteriaBuilder builder = entityManager.getCriteriaBuilder();      CriteriaQuery<User> query = builder.createQuery(User.class);      Root<User> root = query.from(User.class);      root.fetch("cars", JoinType.LEFT);      CriteriaQuery<User> criteriaQuery =          query.select(root).where(builder.and(builder.equal(root.get("id"), 1L)));      User user = entityManager.createQuery(criteriaQuery).getSingleResult();      user.getCars().stream().forEach(System.out::println);    } |

В этом примере, Вы сперва создаете объект CriteriaBuilder. Этот объект позволяет Вам создавать новый CriteriaQuery объект. Далее, создаем источник запроса (query root). На этом объекте вызываем fetch для машин. Затем Вы задаете что нужно выбрать и как должно задаваться where. Для того чтобы сделать запрос безопасным Вы можете использовать также каноническую мета модель. Это позволит заменить строковые «cars» и «id» на безопасные выражения как root.get(User\_.id) или root.get(User\_.cars). Для того чтобы динамически загружать объекты из графа Вы можете добавлять дальнейшие части графа, например, root.fetch("cars", JoinType.LEFT).fetch("axis", JoinType.LEFT) соединяя вызовы метода fetch.

C другой стороны, Вы можете даже комбинировать именованные графы или динамически графы с criteria API:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Test  public void testInitByNamedEntityGraphInJPACriteria() {      CriteriaBuilder builder = entityManager.getCriteriaBuilder();      CriteriaQuery<User> query = builder.createQuery(User.class);      Root<User> root = query.from(User.class);      CriteriaQuery<User> criteriaQuery =          query.select(root).where(builder.and(builder.equal(root.get("id"), 1L)));      User user = entityManager.createQuery(criteriaQuery)          .setHint("javax.persistence.fetchgraph",      entityManager.getEntityGraph("user.cars")).getSingleResult();      user.getCars().stream().forEach(System.out::println);    } |

Как выбрать стратегию?

Резюмируя, я хочу обсудить преимущества и недостатки представленных стратегий и гайд для выбора подходящей стратегии.

**Lazy Loading**

Преимущества:

* Вы выгружаете только те объекты, которые прямо запросили
* Вы не должны думать о конкретной стратегии выгрузки
* Ваш корневой узел загружается быстро

Недостатки:

* Можно получить большие задержки при выгрузки объектов, которые имеют длинный путь в графе
* Нельзя использовать джойны SQL для загрузки частей графа, Вы производите несколько выражений select
* Вы не гибки, потому что Ваша стратегия задана глобальна на уровне маппинга

Нужно использовать если:

* Ваш фронтенд работает на той же самой JVM, что и бэкенд с Hibernate
* Вы не можете предвидеть какие из частей графа нужны в каждой ситуации
* Вы хотите, чтоб приложение стартовало быстро и распределяете нагрузку во времени между последующими действиями пользователя

Вы не должны использовать если:

* Ваш фронтенд вызывает бэкенд Hibernate удаленно
* очевидно какие именно части графа должны быть загружены
* Вы можете пренебречь временем запуска приложения и предзагрузить большинство данных

**Eager Loading**

Преимущества:

* Вы явно загружаете все данные, которые необходимы
* Вы можете использовать JPA для оптимальной стратегии выгрузки (батчи, джойны, селекты)
* Вы не имеете дело с закрытыми сессиями

Недостатки:

* Вы должны думать, какие части должны быть заранее выгружены и какие нет
* Вы будете иметь высокую задержку для загрузки корневого объекта из графа, потому что также другие части графа должны быть загружены
* Вероятно Вы выгрузите ненужные части графа
* Вы не гибки из-за того что стратегия глобальна и задается на уровне маппинга

Нужно использовать если:

* ясно какие объекты из графа всегда выгружаются рядом
* Вам нужно сделать предзагрузку для быстрого доступа позже

Вы не должны использовать если:

* не ясно какие части графа требуются выгружать
* граф выгрузки очень большой

**Explicit join**

Преимущества:

* Вы решаете на уровне операций что нужно выгружать, без выгрузки ненужных объектов
* Вы можете предложить разные варианты Вашей операции (например, и lazy и eager одновременно) используя разные стратегии join для одной и той же операции

Недостатки:

* Более сложно определить что выгружать на уровне запроса
* Опреление что нужно загружать на уровне запроса делает запрос менее переиспользуемым

Нужно использовать если:

* для определенной операции, всегда одни и теже данные должны быть выгружены

Вы не должны использовать если:

* Методы с джойном немного менее эффективны, потому что Вы получаете декартово произведение

**Именованные графы**

Преимущества:

* Вы решаете на уровне операции что нужно выгрузить
* Вы можете предложить разные варианты операции, позволяя использовать разные графы загрузки
* Увеличивается переиспользование, потому что Вы используете тот же самый граф для разных операций

Недостатки:

* Синтаксис многословный, если граф становится сложней
* Именованные графы не могут быть совмещены

Нужно использовать если:

* у Вас есть несколько стратегий загрузки, которые должны поддерживаться и могут переиспользованы для разных операций в репозитории/DAO

Вы не должны использовать если:

* Есть очень много стратегий или комбинаций стратегий

**Динамические графы**

Преимущества:

* Вы решаете на уровне операции что нужно выгрузить
* Потребитель операции репозитория/DAO может сам решить что выгружать, это может уменьшить количество операций предлагаемых Вашими репозиториями/DAO

Недостатки:

* Сложно задавать
* Сложно переиспользовать, потому что они действительны только для конкретного случая

Нужно использовать если:

* есть множество разных стратегий загрузки или комбинаций

Вы не должны использовать если:

* Вам нужно малое количество стратегий в Вашем DAO

**Criteria API**

Преимущества:

* Позволяет писать полностью динамические запросы, включая динамические стратегии
* Может быть комбинирован с графами

Недостатки:

* Более сложен, труден для чтения и сложно задавать

Нужно использовать если:

* у Вас есть множество вариантов одного и того же запроса
* Вы используете конкатенации String для создания запросов

Вы не должны использовать если:

* Вы можете сделать то же самое одним статическим запросом