Статьи Spring framework Grails. Как не допустить ошибок при использовании GORM (Часть 2)

- 2.5
- 0
- 3

- 14 сентября 2010 г.
- sanya

- java ee,
- spring,
- Grails,
- GORM,
- Spring framework

Перевод статьи "GORM Gotchas (Part 2)" Питера Ледбрука от 2 июля 2010 года с официального сайта www.grails.org о том, как не допустить ошибок при использовании базового плагина GORM (объектно ориентированное отображение данных в Grails).

В первой части серии статей я познакомил тебя с некоторыми тонкостями сохранения доменных классов в GORM. На этот раз, я буду рассказывать о реляционных отношениях с акцентом на использование hasMany и belongsTo.

GORM предоставляет только несколько основных элементов для определения отношений между доменными классами, но их достаточно для описания большинства задач. Когда я даю учебные курсы по Grails, то всегда удивляюсь, как мало слайдов о реляционных отношениях. Как ты можешь себе представить, это кажущаяся простота скрывает в себе тонкости поведения в которых можно запутаться. Давай начнем с самых основных отношений: многие-к-одному.

Многие к одному (Many-to-one)

Скажем, я имею два доменных класса:

```
class Location {
    String city
}

class Author {
    String name
    Location location
}
```

Когда ты видишь доменный класс Author (Автор), ты просто знаешь, что в предметной области где-то должен быть доменный класс Book (Книга). Это правильно, далее будет еще и

домен Book, но сейчас давай пока просто сосредоточимся на двух доменных классах и реляционном отношении многие-к-одному (many-to-one).

Оно выглядит просто, не так ли? И это так. Просто вставляешь свойство location типа Location и получаешь в доменом классе Author реляционное отношение с доменом Location. Но посмотри что происходит, когда запускаешь следующий код в Grails консоли:

```
def a = new Author(name: "Niall Ferguson", location: new Location(city:
"Boston"))
a.save()
```

получаешь исключение. Если посмотреть сообщение исключения, то увидишь "not-null property references a null or transient value: Author.location" (Свойство исключающее null имеет значения null или ссылается на transient переменную). Что же происходит? Немного о том что значит для Hibernate "transient", так как это ключевой момент. Переменная transient, это та, которая не подключена к Hibernate сессии. Из кода видно, что переменной Author.location присваивается новый экземпляр доменного класса Location, который не связан с Hibernate-сессией. Таким образом, раз домен Location не в сессии, значит он transient. Исправления очевидны, чтобы присоединить экземпляр Location к сессии, нужно сохранить его:

```
def l = new Location(city: "Boston")
save()

def a = new Author(name: "Niall Ferguson", location: l)
a.save()
```

Так почему же наше свойство с отношением многие к одному не восприняло значение new Location(city: "Boston"), хотя многие примеры GORM пестрят такой техникой присвоения значений? Все это потому что обычно доменные классы используют свойство belongsTo в подобных ситуациях.

Каскадные операции с belongsTo

Всякий раз, когда имеешь дело с отношениями в Hibernate, ты должен иметь хорошее представление о том, что имеется в виду под каскадными операциями. Это справедливо и для GORM. Каскадные операции определяют какой тип действий применить к доменному классу в зависимости от его реляционных отношений. Например, с учетом модели выше: сохранить ли Location с Author? удалить ли Location, если мы удаляем Author? а что делать если удалить Location? удалить ли с Location и Author?

Сохранение и удаление - наиболее распространенные действия связанные с каскадными операциями, и это единственное, что тебе действительно нужно понять. Так что если вернуться к предыдущему разделу, то поймешь, что экземпляр Location не был сохранен с Author, так как каскадное отношение между Author -> Location не установлено. Если мы теперь изменим домен Location на:

```
class Location {
    String city
    static belongsTo = Author
}
```

ты можешь увидеть, что исключение исчезло и Location сохраняется вместе с Author. Строка с BelongsTo гарантирует, что Author сохранится каскадно вместе с Location. Как говорится в документации, теперь и удаление будет происходить каскадно, так что если ты удалишь Author, теперь связанный с ним Location будет тоже удален. Однако, следует помнить что теперь сохранение и удаление Location не произойдет без сохранения и удаления Author.

Какие бывают belongsTo?

Одна вещь, которая часто смущает людей это то, что belongsTo поддерживает два различных синтаксиса. Синтаксис использованный выше просто определяет каскадную связь между двумя классами, в то время как в альтернативном синтаксисе добавляется обратная ссылка, автоматически настраивая двунаправленное реляционное отношение:

```
class Location {
    String city
    static belongsTo = [ author: Author ]
}
```

В этом случае свойство Author добавляется к домену Location во время определения каскадной связи. Преимущество такого синтаксиса ещё и в том, что ты можешь определить несколько каскадных связей.

Ниже представлен интеграционный тест в котором в предпоследней строке выполняется проверка на истинность утверждения, что при использовании синтаксиса static belongsTo = [author: Author] из домена Location доступно свойство author, а в случае использования static belongsTo = Author, свойство author не доступно:

```
void testBidirectional() {
    Author a = new Author(name: "Test", location: new Location(city: "City"))
    a.save()

    assert a.location.city == "City" // true

    def l = Location.get(a.location.id)

    // static belongsTo = [ author: Author ] - true
    // static belongsTo = Author - false

    assert l.author.id == a.id
}
```

Одну вещь, которую нужно запомнить если использовать синтаксис static belongsTo = [author: Author], когда сохраняется новый Author вместе с новым Location, Grails автоматически сохраняет свойство location. Иными словами, обратная ссылка инициализируется в не явном виде. Прежде чем я перейду к коллекциям, я хотел бы сказать последнюю вещь о многих-к-одному. Иногда люди думают, что добавление обратной ссылки, как мы сделали выше, превращает отношения в один-к-одному. На самом деле, это технически не один-к-одному, вот если добавить ограничение уникальности на одной стороне отношения к другой, например:

```
class Author {
    String name
    Location location

    static constraints = {
        location(unique: true)
    }
}
```

то тогда получим один к одному. Конечно, этот пример не имеет смысла в нашей предметной области, так как исключает отношение Location с Author, я привел его для того, чтобы отобразить как технически верно определить отношение один к одному. Отношение многие-к-одному это очень просто, если поймешь как работает belongsTo. С другой стороны, отношения с участием коллекций может создать несколько неприятных сюрпризов, если ты ещё не привык к Hibernate.

Коллекции (один к одному / многие ко многим)

Коллекции естественным образом моделируют отношение один-ко-многим в объектно-ориентированном языке и GORM делает работу с ними очень простой, учитывая то, что происходит за кулисами. Тем не менее, это одна из областей, где несоответствие между объектно-ориентированным языком и реляционной базой данных поднимает свою уродливую голову. Для начала, ты должен запомнить, что твои данные в памяти Hibernate-сессии могут отличатся от тех, что в базе данных.

Коллекции экземпляров доменов против записей в базе данных

Если у тебя есть коллекция экземпляров доменов, то ты имеешь дело с объектами в памяти. Это означает, что ты работаешь с ней, как и с любым другим набором объектов. Ты можете производить итерации и изменять значения. В какой-то момент ты захочешь сохранить изменения в базу данных, то есть коллекцию объектов которую ты изменил. Я вернусь к этому в ближайшее время, но сначала я хотел бы продемонстрировать некоторые тонкости, связанные с разрывом между коллекцией объектов и фактическими данными. Чтобы сделать это, я собираюсь представить класс Book:

```
class Book {
    String title

    static constraints = {
        title(blank: false)
    }
}

class Author {
    String name
    Location location

    static hasMany = [ books: Book ]
}
```

Здесь создается однонаправленная связь (Book не имеет обратной ссылкой на Author) один-ко-многим, где Author имеет ноль или более книг. Теперь я выполняю этот код в Grails консоли (замечательный инструмент для экспериментов с GORM):

```
def a = new Author(name: "Niall Ferguson", location: new Location(city:
"Boston"))
a.save(flush: true)

a.addToBooks(title: "Colossus")
a.addToBooks(title: "Empire")

println a.books*.title
println Book.list()*.title
```

Результат будет выглядеть следующим образом:

[Empire, Colossus]

[ ]

Таким образом, ты можешь напечатать коллекцию Book, но книги ещё не в базе данных. Можно даже вставить a.save () после второго a.addToBooks (), что тоже не даст видимого

эффекта. Вспомни из предыдущей статьи, я рассказывал что save () не гарантирует немедленное сохранение данных. Если хочешь увидеть новые книги в запросе, нужно добавить flush:

```
...
a.addToBooks(title: "Colossus")
a.addToBooks(title: "Empire")

a.save(flush: true) // <---- Это строка добавлена

println a.books*.title
println Book.list()*.title
```

Два оператора println выведут те же книги, хотя и не обязательно в том же порядке. Подтверждение в расхождении между коллекцией в памяти и в базе данных можно получить с помощью следующей строки:

```
println a.books*.id
```

При использовании save() (без явного вызова flush), будет напечатан null. И только при явном сохранении Hibernate-сессии дочерние доменные классы в множестве получат свои уникальные идентификаторы . Это отличается от того, что ты видел при использовании реляционного отношения многие к одному, тогда не было нужды использовать flush при сохранении свойства Location домена Author! Важно понимать существование этой разницы, в противном случае, для тебя настанут тяжелые времена. Небольшое отступление, в случае если ты следующие примеры запускаешь в Grails консоли, помни, что все что ты сохранишь при запуске сценария в консоли будет доступно, даже если выполнишь следующий сценарий. Данные очищаются только при перезагрузке консоли. Кроме того, сессия всегда сохраняется после завершения сценария.

Хорошо, теперь вернемся к коллекциям. Приведенные выше примеры показывают некоторое интересное поведение, которое определяет следующий вопрос. Почему Book сохраняется в базе данных, хотя я не определил belongsTo?

Каскадные отношения

Как и в реляционных отношениях, освоение коллекций в GORM подразумевает овладение каскадным поведением. Прежде всего следует отметить, что сохранения всегда каскадные от родителя к своим детям, даже если не указан belongsTo. Если это так, то стоит ли использовать belongsTo? Конечно да.

Рассмотрим, что произойдет если запустить этот код в консоли после того, как добавишь Author и его книги Book(s):

```
def a = Author.get(1)
a.delete(flush: true)

println Author.list()*.name
println Book.list()*.title
```

Вывод выглядит следующим образом:

[ ]

[Empire, Colossus]

Иными словами, автор был удален, но его книги нет. Вот где необходим belongsTo, он

гарантирует, что удаление произойдет каскадно, так же как и сохранение. Просто добавить строку static belongsTo = Author в Book, и приведенный выше код будет печатать пустые списки для автора и книг. Просто, не так ли? В этом случае, да, но настоящее веселье только начинается.

Посмотри, как мы заставили Hibernate-сессию сохраниться в приведенном выше примере? Если не использовать flush:true, то тогда Author.list() мог бы отобразить имена только что удаленных авторов, только потому что изменения не попали бы в точку сохранения Hibernate-сессии.

Каскадное удаление подчиненных объектов

Удаляешь что-то вроде экземпляра Author и GORM просто автоматически удаляет каскадно его подчиненные объекты. Но что делать, если просто хочешь удалить одну или несколько книг автора, а не самого автора? Можешь попробовать следующее:

```
def a = Author.get(1)
a.books*.delete()
```

Думаешь этот код приведет к удалению всех книг? На самом деле этот код сгенерирует исключение:

org.springframework.dao.InvalidDataAccessApiUsageException: deleted object would be re-saved by cascade (remove deleted object from associations): [Book#1]; ...

at org.springframework.orm.hibernate3.SessionFactoryUtils.convertHibernateAccessException(SessionFactoryUtils.java:657)

at org.springframework.orm.hibernate3.HibernateAccessor.convertHibernateAccessException(HibernateAccessor.java:412)

at org.springframework.orm.hibernate3.HibernateTemplate.doExecute(HibernateTemplate.java:411)

at org.springframework.orm.hibernate3.HibernateTemplate.executeWithNativeSession(HibernateTemplate.java:374)

at org.springframework.orm.hibernate3.HibernateTemplate.flush(HibernateTemplate.java:881)

at ConsoleScript7.run(ConsoleScript7:3)

Caused by: org.hibernate.ObjectDeletedException: deleted object would be re-saved by cascade (remove deleted object from associations): [Book#1]

Ух ты, какие полезные сообщения! Да, проблема в том, что книги все еще находятся в коллекции автора, поэтому, когда Hibernate-сессия будет сохранена, они будут заново созданы для автора. Помни, что не только подчинённые объекты сохраняются каскадно, но и изменения экземпляров домена автоматически сохраняются (из-за грязной проверки в Hibernate).

Возможно это решение поможет обойти исключение удаления книг из коллекции:

```
def a = Author.get(1)
a.books.clear()
```

Однако, это не является решением, потому что книги все еще находятся в базе данных. Они

просто больше не связаны с автором. Итак, ты должен удалить их также:

```
def a = Author.get(1)

a.books.each { book ->
    a.removeFromBooks(book)
    book.delete()
}
```

К сожалению, сейчас получаешь исключение «ConcurrentModificationException», потому что удаляешь из книги автора коллекции во время итерации по авторам. Здесь стандартная Java ошибка. Можешь сделать шаг в сторону для исправления ситуации, создав копию коллекции, как в следующем примере:

```
def a = Author.get(1)

def l = []
l += a.books

l.each { book ->
    a.removeFromBooks(book)
    book.delete()
}
```

и снова получишь ошибку "not-null property references a null or transient value: Book.author". Я потом объясню, почему автору книги был присвоен null. Так как свойство author не может содержать null, то триггер валидности сработал на присвоение null. Этого достаточно, чтобы кто-нибудь сошел с ума!

Не боись, ибо есть решение - добавь правила мапинга в домен Author:

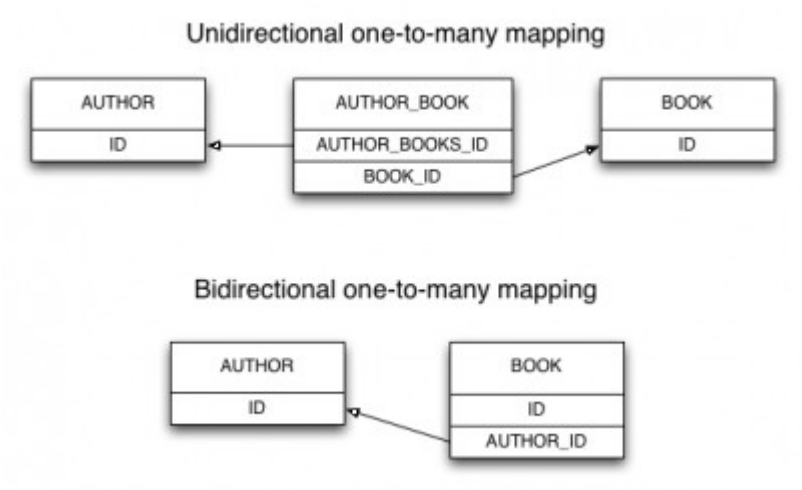
```
static mapping = {
    books cascade: "all-delete-orphan"
}
```

Теперь любая книга, которая удалена у автора будет автоматически удалена с помощью GORM. Последний пример, где удаляются все книги из коллекции, теперь будет работать. На самом деле, если связь однонаправленная ты можешь существенно уменьшить код:

```
def a = Author.get(1)
a.books.clear()
```

Это позволит удалить все книги одним махом! Мораль этой истории проста: если используешь belongsTo, то четко установи каскадный тип "all-delete-orphan" в блоке отображения родителя. На самом деле, есть веские основания для принятия этого поведения по умолчанию для belongsTo в реляционной связке один-ко-многим в GORM.

В этой связи возникает интересный вопрос: почему clear () метод не работает в двунаправленной связи? Я не уверен на 100%, я считаю, это потому, что книги сохраняют обратную ссылку на автора. Чтобы понять, почему это влияет на поведение clear(), ты должен сначала понять, что GORM строит таблицы базы данных по разному для однонаправленной и двунаправленной связи один-ко-многим. Для однонаправленных отношений, GORM создает промежуточную таблицу, поэтому когда удаляешь записи с книгами, то из промежуточной таблицы просто удаляются записи. Двунаправленная связь отображается с использованием внешнего ключа дочерней таблицы, то есть в доменном классе книга (Book) добавляется внешний ключ на автора (Author). Схема должна объяснить все более ясно:



Когда удаляешь у автора коллекцию книг, то внешний ключ все еще остается в таблице AUTHOR_BOOK, потому что GORM не удаляет значение свойства автора (author). Отсюда получается, как будто коллекцию не удаляли.

Далее ещё про коллекции. О методах addTo*() и removeFrom*().

addTo*() против <<

В моих примерах я использовал addTo*() и removeFrom*() динамические методы предоставляемые GORM. Для чего? В конце концов, если работаешь со стандартными коллекциями Java, то можно просто использовать такой код:

```
def a = Author.get(1)
a.books << new Book(title: "Colossus")
```

Конечно можно, но есть некоторые подводные камни в GORM. Рассмотрим следующий код:

```
def a = new Author(name: "Niall Ferguson", location: new Location(city:
"Boston"))
a.books << new Book(title: "Colossus")
a.save()
```

На первый взгляд кажется что всё в порядке в том, что написано выше. И все же, если запустить код, получишь исключение «NullPointerException», потому что коллекция книг еще не инициализирована. Это в корне отличается от поведения которое ты видел в предыдущем примере, когда получил автора из базы данных с помощью get(). В случае с get(), ты можешь спокойно добавлять книги, а с проблемой столкнуться только при создании нового автора. Проблема решается при использовании addTo*() вместо оператора <<, этот метод null-безопасный (null-safe) так как инициализирует коллекцию книг.

Теперь рассмотрим пример, в котором выберем автора используя get() перед добавлением новой книги в коллекцию книг автора. Если отношение является двунаправленным, будешь получать исключение "property not-null or transient", потому что свойство Author не было установлено в домене Book. Если использовать оператор <<, тогда нужно инициализировать обратную ссылку вручную. С addTo * () методом, это делать не нужно.

Еще одна возможность addTo*() метода неявно до создавать корректный класс домена. Обратите внимание на примеры, как просто вставить значение в коллекцию книг, которая ещё не была инициализирована. Это потому, что метод addTo*() может получить из свойства hasMany тип коллекции содержащейся в нем. Здорово, да?

RemoveFrom * () метод является менее полезным, но он корректно вычищает обратные ссылки. Конечно, это лучше всего работает с "all-delete-orphan" опцией, как я говорил ранее.

Последний тип реляционных отношений это многие ко многим.

Многие-ко-многим

Если ты хочешь, можешь передать управление реляционным отношением многие-ко-многим GORM. Есть несколько вещей, которые нужно знать:

№1 Удаляет не каскадно, и точка.

№2 На одной стороне отношения должен быть belongsTo, обычно не имеет значения, с какой стороны он есть.

№3 BelongsTo влияет только на направление каскадных операций сохранения и не влияет на каскадные операции удаления.

№4 Объеденяющая таблица используется всегда, но ты не можете хранить какую либо информацию в ней.

Извени за точку в №1 о каскадном удалении, но очень важно понять, что здесь поведение существенно отличается от многих к одному и один-ко-многим. Важно также понимать последний момент, многие-ко-многим связывают информацию. Например, пользователь может иметь много ролей, а роль может содержать в себе многих пользователей. Но пользователь может иметь различные роли в различных проектах, следовательно, проект имеет собственные реляционные связи. В таких случаях лучше управлять самостоятельно реляционным отношением многие-ко-многим.

Резюме

Ну, наверное, эта самая длиная статья, которую я написал. Мы подошли к концу. Поздравляю! Не беспокойся, если тебе не удалось переварить все в один присест, всегда можно вернуться к тем частям, которые не понятны.

Я думаю, GORM делает большую работу по абстрагированию работы с базой данных объектно-ориентированным стилем, но как ты видел в конечном счете в глубине всего лежит база данных. Вооруженный информацией, представленной в этой статье, ты не должен теперь иметь каких-либо проблем с GORM коллекциями. Надеюсь это означает, что теперь ты будешь наслаждаться кодированием приложений с реляционными связями.

Можешь не поверить, но я еще не рассказал все, что нужно знать о коллекциях. Есть еще некоторые интересные вопросы, которые покрывают lazy-loading (ленивую загрузку данных), я буду рассказывать об этом в следующей статье.

До следующей встречи!