Статьи Spring framework Grails. Как не допустить ошибок при использовании GORM (Часть 1)

- 6
- 1
- 6

- 5 июля 2010 г.
- sanya

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

Перевод статьи Питера Ледбрука от 23 июня 2010 года с официального сайта www.grails.org о том как не допустить ошибок при использовании базового плагина GORM (*объектное отображение данных Grails*).

Ты новичок в Grails? Или возможно ты запустил свое первое приложение с GORM? Если это так, то тебе необходимо прочитать эту серию статей о том как не допустить ошибок при использовании GORM. Эти статьи не только об особенностях которые иногда не учитывают разработчики, но и о том почему GORM ведет себя определенным образом.

Надеюсь, ты уже знаешь, что GORM это базовый плагин (библиотека доступа к базам данных), который поставляется с Grails. Он основан на популярном Java ORM: Hibernate. Как ты возможно знаешь, Hibernate является мощной и гибкой библиотекой, которая дает большие выгоды GORM. Но за все нужно платить: многие из проблем, с которыми пользователи GORM сталкиваются ведут к корням Hibernate. GORM пытается скрыть детали реализации Hibernate, но иногда эти детали дают о себе знать.

В следующей части этой статьи, я опишу основы сохранения объектов в базу данных. Звучит просто, но GORM не всегда работает, так как можно было бы ожидать полагаясь только на базовые принципы. Итак, начнем:

Когда я вызываю save(), я имею в виду сохранить!

Проблема с сохранением объектов предметной области (domain class), вероятно, первая из проблем с которой разработчики сталкиваются. Многие из нас прошли через "Я сохранил его, так почему же не в базу данных"? Слыша эту фразу ты начинаешь чувствовать себя лучше, потому что ты не один! Почему это происходит? Есть несколько причин.

Не забывайте о проверке!

Каждый раз, когда ты сохраняешь домен (domain class), Grails проверяет его с помощью ограничений, которые ты определил. Если какие-либо значения домена нарушают ограничения, то объект не сохраняется, а ошибки об ограничениях присоединяются к домену. Решение проблемы валидации решается простой проверкой переменной на null, которая возвращается методом save() или вызовом функции hasErrors() после вызова метода save().

Когда ты связываешь данные пользователя с доменом, то как правило, ты ожидаешь что данные будут в домене. И нет смысла удивляться, если пользователь ввел данные не в соответствии с ограничениями. В данном случае выбросить исключение просто не уместно, особенно когда это веб-приложение с большим количеством пользователей. В этих условиях всегда лучше проверить возвращаемое значение метода save() и соответствующим образом отреагировать (возвращает null, если сохранить не удалось, в противном случае возвращается экземпляр класса домена), например:

```
def book = new Book(params)
if (!book.save()) {
    // Ошибка сохранения! Отобразить ошибки пользователю.
    ...
}
```

С другой стороны, когда ты сам загружаешь тестовые данные в BootStrap или в Grails консоле, ты обычно ожидаешь, что все сохранится. Если в твоих данных есть ошибки валидации, то это означает, что ты допустил ошибку в формировании данных. В таких случаях не хочется возиться с проверкой возвращаемых значений и было бы намного лучше, если бы Grails выбросил исключение для не соответствующих данных. Это поведение возможно не по умолчанию, и ты можешь легко включить его с помощью failOnError аргумента:

```
book.save(failOnError: true)
```

Если ты желаешь, чтобы такое поведение было по умолчанию, то можешь просто установить grails.gorm.failOnError значение true в grails-app/conf/Config.groovy. И не забывай, что всем доменным свойствам неявно уставлено ограничение nullable: false, что является, наиболее частой причиной не сохранения доменов. Так что же дальше?

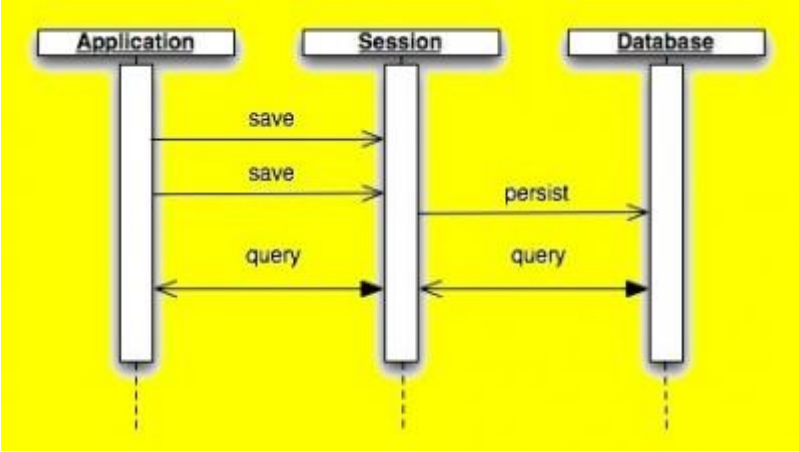
Hibernate-сессия

В редких случаях, ты можешь заметить, что после сохранения не можешь найти экземпляр твоего доменного класса по запросу, даже если твой доменный класс прошел успешно проверку. Это является симптомом более широкого вопроса, связанного с использованием Hibernate под капотом.

Hibernate основан на сессионном объектном отображении данных (ORM).

Это очень важный момент, и кто желает по-настоящему комфортно владеть GORM должен понимать, что такое сессия и какое влияние она оказывает на приложения. Так что же такое Hibernate-сессия? Это в основном кэш объектов в памяти, которые опираются на данные в базах данных. Когда ты сохраняешь новый экземпляр домена, он неявно привязывается к сессии, т.е. он добавляется в кэш-память и становится Hibernate-управляемым объектом, но экземпляр домена может быть не сохранен в базу данных на момент его создания!

Следующая диаграмма иллюстрирует это поведение:



При сохранении экземпляра доменного класса, он сразу же доступен в сессии. Но Hibernate использует его по своему усмотрению и решает самостоятельно когда сохранять в базу данных данные экземпляра, для того чтобы оптимизировать порядок запросов к базе данных. Обычно ты не заметишь какую-либо разницу, поскольку Grails и Hibernate заботится об порядке, но иногда необходимо прямое вмешательство.

Как и следовало ожидать, Grails позволяет тебе контролировать, когда данные фактически сохраняются в базе данных. Никогда не видел такой код в примерах?

```
book.save(flush: true)
```

flush: true сообщает Hibernate сохранить все изменения сразу в базе данных. Это соответствует *сохранению сессии*.

Опасность знания об немедленном сохранении данных экземпляра доменого класса в том, что ты будешь писать flush: true везде. Не надо этого делать. Пусть Hibernate делает свою работу, сохраняй сессию, только когда нужно или по крайней мере, только в конце пакета обновлений. Ты должен использовать flush: true, только когда ты не видишь данные в базе данных, но они уже должны быть там. Я знаю, что воспользоваться моим советом не всегда возможно, так как многое зависит и от реализации базы данных. Одна из областей, где нужно использовать flush: true, это когда ты взаимодействуешь с другим приложением или внутренней службой, которые обращаются к той же базе данных, но за пределами текущего сеанса.

Если ты все еще туманно себе представляете работу Hibernate сессии, не волнуйся - мы будем возвращаться к Hibernate сессии снова и снова, потому что она имеет основополагающее значение для многих ошибок, связанных с GORM. И следующая проблема, тоже связана с ней же.

Когда я не вызываю save(), я имею в виду не сохранять!

В противном случае бывает, когда домен сохраняется, когда ты не вызывал save(). Давай рассмотрим эту ситуацию. Если ты не сталкивался с данным поведением, то я гарантирую, что обязательно столкнешься. Так почему так происходит?

Hibernate поддерживает концепцию грязных проверок (dirty-checking). Это означает, что Hibernate проверяет имеются ли какие-либо изменения значений в свойствах экземпляра домена и если что-либо изменилось, то эти изменения сохраняются в базе данных автоматически. Следующий пример разъяснит ситуацию: предположим, что есть домен класса Book с свойствами `title и author и следующий код отображает`

событие в контролере:

```
def b = Book.findByAuthor(params.author)
b.title = b.title.reverse
```

Обрати внимание, что вызова save() в коде нет. Когда код отработает, то ты увидишь, что новое название книги было сохранено в базе данных без явного сохранения. Это потому, что:

1. Book находится в Hibernate сессии (так как домен получен через запрос);
2. так как свойство title сохраняемое (все свойства доменов по умолчанию сохраняемые до тех пор пока не установить transient);
3. значение свойства было изменено после чего сессия закрылась.

Давайте рассмотрим пример более детально. Во-первых, я уже упоминал, что объекты могут быть "привязанными" к сессии, то есть быть под управлением Hibernate, но как же объект был привязан к сессии? Если вы используете GORM для получения домена например, через get() метод или любой тип запроса, то объект автоматически ассоциируется с сессией. Если ты только что создал новый экземпляр через new, то объект *не подключен* к сессии, пока не вызвать метод save().

Во-вторых, свойства домена класса являются сохраняемыми по умолчанию, то есть они имеют соответствующие столбцы в базе данных, где их значения хранятся. Можешь сделать свойства не сохраняемыми, добавив transient, что значит их значения не будут хранится в базе данных.

И наконец, я отметил, что изменения сохраняются, если сессия закрывается. Что я имел в виду? Для того, чтобы работать с базой данных через Hibernate, ты должен открыть сессию. После того как сессия закрыта, ты больше не можешь использовать её для доступа к базе данных. Кроме того сессия сохраняется до того как закроется, именно поэтому изменения сохраняются в конце события нашего контроллера (Grails автоматически открывает сессию в начале запроса и закрывает ее в конце).

Можно ли предотвратить такие автоматические изменения? Конечно. Одним из вариантов является вызов save() до закрытия сессии для того, чтобы валидировать изменения, к тому же данные могут быть не сохранены в случае ошибки возникающей при проверке ограничений. Конечно, если значения в порядке, но ты не хочешь сохранять данные, то вызови discard() до закрытия сессии. Этот вызов не сбросит значения свойств экземпляра домена, но это гарантирует, что данные не будут сохранены в базе данных.

Надеюсь, первая часть статьи поможет тебе избежать некоторые ошибки. Ключ к пониманию того как их не допустить кроется в том как сессия Hibernate влияет на сохранение доменов. Даже если у тебя сейчас не все уложилось в голове, в будущих статьях о GORM я предоставлю ещё более подробную информацию и примеры, которые помогут в усвоении материала.

В общем, я рекомендую, чтобы ты всегда использовал явно save() для сохранения объектов, не полагаясь на грязную-проверку (dirty-checking) Hibernate. Это позволяет сделать код более понятным, потому что ты сообщаешь о том, что хочешь сохранить объект и проверить его на ограничения. Я также рекомендую всегда проверять возвращаемое значение save() в коде приложения, вместо использования failOnError: true аргумента.

Если ты по прежнему опасаешься работать с GORM на данном этапе, то это ты зря. GORM действительно позволяет работать с базой данных играючи и весело, полученная информация из этой серии статей поможет тебе наработать доверие для решения любых проблем, которые могут возникнуть при использовании GORM.

До следующей встречи!