Grails Persistence Tips and Gotchas

Boston Grails Users' Group

August 5, 2009

Burt Beckwith

My Background

- Java Developer for over 10 years
- Background in Spring, Hibernate, Spring Security
- Full-time Grails developer since February 2008
- Regular contributor on the Grails User mailing list
- Primary developer of Spring Security (Acegi) Grails plugin
- Created UI Performance, Datasources, Twitter, Spring MVC, and CodeNarc Grails plugins
- Technical Editor of Grails in Action
- 2008 Groovy Award winner
- http://burtbeckwith.com/blog/
- http://twitter.com/burtbeckwith

Standard Grails One-to-Many

Library has many Visits:

```
class Library {

        String name

        static hasMany = [visits: Visit]
}
```

```
class Visit {

        String personName
        Date visitDate = new Date()

        static belongsTo = [library: Library]
}
```

Standard Grails One-to-Many (cont.)

Usage:

```
Library library = new Library(name: 'Carnegie').save()
...
library.addToVisits(new Visit(personName:'me'))
library.save()
...
library.addToVisits(new Visit(personName:'me2'))
library.save()
```

- object-oriented approach to recording each visit to a library

- convenience method addToVisits() (along with corresponding removeFromVisits()) handles adding to/removing from mapped collection and cascading the save/delete

Standard Grails One-to-Many (cont.)

DDL (use “grails schema-export” to generate):

```
create table library (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        name varchar(255) not null,
        primary key (id)
);

create table visit (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        library_id bigint not null,
        person_name varchar(255) not null,
        visit_date timestamp not null,
        primary key (id)
);

alter table visit add constraint FK6B04D4B4AEC8BBA
foreign key (library_id) references library;
```

So What's the Problem?

- “hasMany = [visits: Visit]” creates a Set (org.hibernate.collection.PersistentSet) in Library – the “inverse” collection in traditional Hibernate

- Adding to the Set requires loading all instances from the database to guarantee uniqueness, even if you know the new item is unique

- Likewise for a mapped List – Hibernate pulls the entire collection to maintain the correct order, even if you're adding to the end of the list

- In traditional Hibernate you could map the collection as a Bag, which is just a regular Collection with no ordering or uniqueness guarantees, but Grails doesn't support Bags

So What's the Problem? (cont.)

• You get a false sense of security since it's a lazy-loaded collection (by default); loading a Library doesn't load all Visits, but that's only partially helpful

• Works fine in development when you only have a few Visits, but imagine when you deploy to production and you have 1,000,000 Visits and want to add one more

• Risk of artificial optimistic locking exceptions; altering a mapped collection bumps the version, so simultaneous Visit creations can break but shouldn't

Demo

Ok, So What's the Solution?

Remove the collection:

```
class Library {
        String name
}
```

```
class Visit {
        String personName
        Date visitDate = new Date()
        Library library
}
```

How does that affect usage?

```
Library library = new Library(name: 'Carnegie').save()
...
new Visit(personName:'me', library: library).save()
...
new Visit(personName:'me2', library: library).save()
```

- Different syntax for persisting a Visit
- No cascading; to delete a Library you need to delete its Visits first
- If you want to know all Visits for a Library, use a custom finder:
  - def visits = Visit.findAllByLibrary(library)
- This is actually significantly more convenient since you can query for the 10 most recent, just last month's visits, etc.:
  - def last10 = Visit.executeQuery(
    "from Visit v order by visitDate desc", [max: 10])

How does this affect DDL?

Not at all, both approaches set visit.library_id:

```
create table library (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        name varchar(255) not null,
        primary key (id)
);

create table visit (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        library_id bigint not null,
        person_name varchar(255) not null,
        visit_date timestamp not null,
        primary key (id)
);

alter table visit add constraint FK6B04D4B4AEC8BBA
foreign key (library_id) references library;
```

Standard Grails Many-to-Many

User has many Roles, Roles have many Users:

```
class User {
	static hasMany = [roles: Role]

	String username
}
```

```
class Role {

	static belongsTo = User
	static hasMany = [users: User]

	String name
}
```

Standard Grails Many-to-Many (cont.)

Usage:

```
Role roleUser = new Role(name: 'ROLE_USER').save()
Role roleAdmin = new Role(name: 'ROLE_ADMIN').save()
…
User user1 = new User(username:'user1')
user1.addToRoles(roleUser)
user1.save()
...
User user2 = new User(username:'user2')
user2.addToRoles(roleAdmin)
user2.save()
```

- object-oriented approach to assigning a Role to a User
- syntax is very similar to One-to-many
- convenience method addToRoles() (along with corresponding removeFromRoles()) handles adding to/removing from mapped collection and cascading the save/delete

Standard Grails Many-to-Many (cont.)

DDL (use “grails schema-export” to generate):

```
create table role (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        name varchar(255) not null,
        primary key (id)
);

create table user (
        id bigint generated by default as identity (start with 1),
        version bigint not null,
        username varchar(255) not null,
        primary key (id)
);

create table user_roles (
        user_id bigint not null,
        role_id bigint not null,
        primary key (user_id, role_id)
);

alter table user_roles add constraint FK7342994952388A1A foreign key (role_id) references role;
alter table user_roles add constraint FK73429949F7634DFA foreign key (user_id) references user;
```

So What's the Problem?

• “hasMany = [users: User]” and “hasMany = [roles: Role]” create a Set (org.hibernate.collection.PersistentSet) in both User and Role

• Adding a Role to a User's 'roles' Set requires loading all instances of the User's Roles (for uniqueness check) AND all other Users who already have that Role from the database

• This is because Grails automatically maps both collections for you and populates both - “user.addToRoles(role)” and “role.addToUsers(user)” are equivalent because it's bidirectional

• Works fine in development when you only have a few Users, but imagine when you deploy to production and you have 1,000,000 registered site users with ROLE_MEMBER and want to add one more

• Risk of artificial optimistic locking exceptions; altering a mapped collection bumps the version, so simultaneous Role grants can break but shouldn't

Demo

Ok, So What's the Solution?

Remove the collections and map the join table:

```
class User {
	String username
}
```

```
class Role {
	String name
}
```

```
class UserRole implements Serializable {
	User user
	Role role
	static mapping = {
		table 'user_roles'
		version false
		id composite: ['user', 'role']
	}
}
```

Ok, So What's the Solution? (cont.)

Plus we can add in some helper methods to UserRole:

```
class UserRole implements Serializable {

        ...

        static UserRole create(User user, Role role, boolean flush = false) {
                UserRole userRole = new UserRole(user: user, role: role)
                userRole.save(flush: flush, insert: true)
                return userRole
        }

        static boolean remove(User user, Role role, boolean flush = false) {
                UserRole userRole = UserRole.findByUserAndRole(user, role)
                return userRole ? userRole.delete(flush: flush) : false
        }

        static void removeAll(User user) {
                executeUpdate(
                        "DELETE FROM UserRole WHERE user=:user", [user: user])
        }
}
```

Ok, So What's the Solution? (cont.)

and restore a 'roles' pseudo-collection back in User:

```
class User {

	String username

	Set<Role> getRoles() {
		UserRole.findAllByUser(this).collect { it.role } as Set
	}

	boolean hasRole(Role role) {
		return UserRole.countByUserAndRole(this, role) > 0
	}
}
```

How does that affect usage?

```
User user = …
Role role = …
UserRole.create user, role
 – or –
UserRole.remove user, role
```

- Different syntax for granting a Role

- No cascading; to delete a User you you need to delete (disassociate) its Roles first (use UserRole.removeAll(User user))

- In the unlikely case that you want to know all Users with a Role, use a custom finder:
  - def users = UserRole.findAllByUser(role).collect { it.user } as Set

So Never Use Mapped Collections?

- No, you need to examine each case

- The standard approach is fine if the collections are reasonably small – both sides in the case of Many-to-Many

- The collections will contain proxies, so they're smaller than real instances until initialized, but still a memory concern

Using Hibernate 2nd Level Cache

DataSource.groovy:

```
dataSource {
        pooled = true
        driverClassName = 'com.mysql.jdbc.Driver'
        url = ...
        username = ...
        password = ...
        dialect = org.hibernate.dialect.MySQL5InnoDBDialect
}

hibernate {
        cache.use_second_level_cache = true
        cache.use_query_cache = true
        cache.provider_class = 'org.hibernate.cache.EhCacheProvider'
}
```

Mapping in Domain Class

```
class Book {
        …
        static mapping = {
                cache true
        }
}
```

```
class Country {
        …
        static mapping = {
                cache usage: 'read-only'
        }
}
```

```
class Author {
        static hasMany = [books:Book]
        static mapping = {
                books cache: true
        }
}
```

Usage Notes

- The 1st level cache is the Hibernate Session

- Can significantly reduce database load by keeping instances in memory

- Can be distributed between multiple servers to let one instance load from the database and share updated instances, avoiding extra database trips

- “cache true” creates a read-write cache, best for read-mostly objects since frequently-updated objects will result in excessive cache invalidation (and network traffic when distributed)

- “cache usage: 'read-only'” creates a read-only cache, best for lookup data (e.g. Countries, States, Zip Codes, Roles, etc.) that never change

- DomainClass.get() always uses the 2nd-level cache

- By default nothing else always uses the cache but can be overridden

Configuring with ehcache.xml

• If you don't create an ehcache.xml file in the classpath (either grails-app/conf or src/java) EhCache will use built-in defaults and you'll see warnings at startup

```
<ehcache>

    <diskStore path="java.io.tmpdir" />

    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        ...
    />

    <cache name="com.foo.bar.Thing"
        maxElementsInMemory="10000"
        eternal="false"
        overflowToDisk="false"
        maxElementsOnDisk="0"
    />
```

Configuring with ehcache.xml (cont.)

```
    <cache name="com.foo.bar.Zipcode"
        maxElementsInMemory="100000"
        eternal="true"
        overflowToDisk="false"
        maxElementsOnDisk="0"
    />

    <cache name="org.hibernate.cache.StandardQueryCache"
        maxElementsInMemory="50"
        eternal="false"
        overflowToDisk="false"
        maxElementsOnDisk="0"
        timeToLiveSeconds="120"
    />
    <!-- timestamps of the most recent updates to queryable tables -->
    <cache name="org.hibernate.cache.UpdateTimestampsCache"
        maxElementsInMemory="5000"
        eternal="true"
        overflowToDisk="false"
        maxElementsOnDisk="0"
    />
</ehcache>
```

Query Cache

Criteria queries:

```
def criteria = DomainClass.createCriteria()
def results = criteria.list {
        cacheable(true)
}
```

HQL queries:

```
DomainClass.withSession { session ->
        return session.createQuery(
                "select ... from … where ...")
                .setCacheable(true)
                .list()
        }
}
```

In dynamic finders (new in 1.1)

```
def person = Person.findByFirstName("Fred", [cache:true])
```

Hibernate query cache considered harmful?

- Most queries are not good candidates for caching; must be same query and same parameters

- DomainClass.list() is a decent candidate if there aren't any (or many) updates and the total number isn't huge

- Great blog post by Alex Miller (of Terracotta)
http://tech.puredanger.com/2009/07/10/hibernate-query-cache/

2nd Level Cache API

- evict one instance
  - sessionFactory.evict(DomainClass, id)
- evict all instances
  - sessionFactory.evict(DomainClass)
- evict one instance's collection
  - sessionFactory.evictCollection('DomainClass.collectionName', id)
- evict all of DomainClass' collections
  - sessionFactory.evictCollection('DomainClass.collectionName')
- Map cacheEntries = sessionFactory.statistics
  .getSecondLevelCacheStatistics(regionName)
  .entries

2nd Level Cache API (cont.)

- sessionFactory.statistics (org.hibernate.stat.Statistics) methods:
  - statistics.queryCacheHitCount
  - statistics.queryCacheMissCount
  - statistics.queryCachePutCount
  - statistics.secondLevelCacheHitCount
  - statistics.secondLevelCacheMissCount
  - statistics.secondLevelCachePutCount
  - statistics.secondLevelCacheRegionNames

2nd Level Cache API (cont.)

• statistics.getSecondLevelCacheStatistics(cacheName) (org.hibernate.stat.SecondLevelCacheStatistics) methods:

- cacheStatistics.elementCountInMemory
- cacheStatistics.elementCountOnDisk
- cacheStatistics.hitCount
- cacheStatistics.missCount
- cacheStatistics.putCount
- cacheStatistics.sizeInMemory