Hibernate: Truly Understanding the Second-Level and Query Caches

At 1:22 AM on Sep 26, 2005, R.J. Lorimer wrote:

I've written multiple articles here at Javalobby on the Hibernate O/R mapping tool, and they are usually met with a large degree of popularity. Hibernate isn't just a popular kid on the block, however; it is actually a very powerful, consistent, and reliable database mapping tool. Mapping between objects in Java to relational databases has many facets that you must be aware of. Hibernate does a particularly good job of making the process simple to start, and providing the facilities to allow it to scale well and meet exceedingly complex mapping demands.

One of the primary concerns of mappings between a database and your Java application is performance. One of the common concerns of people who haven't spent much time working with Hibernate in particular, is that O/R mapping tools will limit your ability to make performance-enhancing changes to particular queries and retrievals. Today I want to discuss two facets of the Hibernate infrastructure that are implemented to handle certain performance concerns - the second-level cache and the query cache.

The Second Level Cache

The second-level cache is called 'second-level' because there is already a cache operating for you in Hibernate for the duration you have a session open. From the Hibernate documentation:

A Hibernate Session is a transaction-level cache of persistent data. It is possible to configure a cluster or JVM-level (SessionFactory-level) cache on a class-by-class and collection-by-collection basis. You may even plug in a clustered cache. Be careful. Caches are never aware of changes made to the persistent store by another application (though they may be configured to regularly expire cached data).

As implied above, this 'second-level' cache exists as long as the session factory is alive. The second-level cache holds on to the 'data' for all properties and associations (and collections if requested) for individual entities that are marked to be cached.

I'm not here to re-hash the details provided clearly by the Hibernate documentation, so for detailed information about selecting a cache provider and configuring Hibernate to use it, look at Section 20.2 of the Hibernate documentation .

Suffice it to say, the important part is the 'cache' element which you add to your mapping file:

```
<cache usage="transactional|read-write|nonstrict-read-write|read-only" />
```

For most readers, what I'm describing here is probably nothing new. Bear with me, I'll get to the goodies in a minute.

The Query Cache

The other cache that is available is the query cache. The query cache effectively holds on to the identifiers for an individual query. As described in the documentation:

Note that the query cache does not cache the state of the actual entities in the result set; it caches only identifier values and results of value type. So the query cache should always be used in conjunction with the second-level cache.

Configuration of the query cache is described in Section 20.4 of the Hibernate Documentation .

Once enabled via the configuration of Hibernate, it is simply a matter of calling `setCacheable(true)` on your `Query` or `Criteria` object.

Now, on to the inner workings.

How The Second-Level Cache Works

One of the keys to understanding how the caches can help you is to understand how they work internally (at least conceptually). The second-level cache is typically the more important to understand, particularly when dealing with large, complex object graphs that may be queried and loaded often. The first thing to realize about the second-level cache is that it doesn't cache instances of the object type being cached; instead it caches the individual values for the properties of that object. So, conceptually, for an object like this:

```
public class Person {
 private Person parent;
 private Set<Person> children;

 public void setParent(Person p) { parent = p; }
 public void setChildren(Set<Person> set) { children = set; }

 public Set<Person> getChildren() { return children; }
 public Person getParent() { return parent; }
}
```

...with a mapping like this:

```
<class name="org.javalobby.tnt.hibernate.Person">

<cache usage="read-write"/>

  <id name="id" column="id" type="long">
   <generator class="identity"/>
  </id>
  <property name="firstName" type="string"/>
  <property name="middleInitial" type="string"/>
  <property name="lastName" type="string"/>
  <many-to-one name="parent" column="parent_id" class="Person"/>
  <set name="children">
   <key column="parent_id"/>
   <one-to-many class="Person"/>
  </set>
</class>
```

Hibernate will hold on to records for this class conceptually like this:

```
*----------------------------------------*
|          Person Data Cache             |
|----------------------------------------|
| 1 -> [ "John" , "Q" , "Public" , null ] |
| 2 -> [ "Joey" , "D" , "Public" ,  1   ] |
| 3 -> [ "Sara" , "N" , "Public" ,  1   ] |
*----------------------------------------*
```

So, in this case, Hibernate is holding on to 3 strings, and one serializable identifier for the 'many-to-one' parent relationship. Let me reiterate that Hibernate is *not* holding on to actual instances

of the objects. Why is this important? Two reasons. One, Hibernate doesn't have to worry that client code (i.e. your code) will manipulate the objects in a way that will disrupt the cache, and two, the relationships and associations do not become 'stale', and are easy to keep up to date as they are simply identifiers. The cache is not a tree of objects, and can instead just be a conceptual map of arrays. I continually use the word conceptual, because Hibernate does much more behind the scenes with this cache but, unless you plan on implementing your own provider, you don't need to worry about it. Hibernate calls this state that the objects are in 'dehydrated', as it is all the important bits of the object, but in a more controllable form for Hibernate. I'll use the terms 'dehydrate' and 'hydrate' to describe the process of converting between this broken apart data into an object of your domain (hydrating being the process of creating an instance of your domain and populating it with this data, dehydrating being the inverse).

Now, the astute of you may have noticed that I have omitted the 'children' association all-together from the cache. This was intentional. Hibernate adds the granularity to allow you to decide which associations should be cached, and which associations should be re-determined during the hydration of an object of the cached type from the second-level cache. This provides control for when associations can potentially be altered by another class that doesn't explicitly cascade the change to the cached class.

This is a very powerful construct to have. The default setting is to not cache associations; and if you are not aware of this, and simply turn on caching quickly without really reading into how caching works in Hibernate, you will add the overhead of managing the cache without adding much of the benefits. After all, the primary benefit of caching is to have complex associations available without having to do subsequent database selects - as I have mentioned before, n+1 database queries can quickly become a serious performance bottleneck.

In this case we're dealing just with our person class, and we know that the association will be managed properly - so let's update our mapping to reflect that we want the 'children' association to be cached.

```
  <set name="children">

<cache usage="read-write"/>

   <key column="parent_id"/>
   <one-to-many class="Person"/>
```

Now, here is an updated version of our person data cache:

```
*-----------------------------------------------------*
|                 Person Data Cache                   |
|-----------------------------------------------------|
| 1 -> [ "John" , "Q" , "Public" , null , [ 2 , 3 ] ] |
| 2 -> [ "Joey" , "D" , "Public" ,  1   , []        ] |
| 3 -> [ "Sara" , "N" , "Public" ,  1   , []        ] |
*-----------------------------------------------------*
```

Once again let me point out that all we are caching is the ID of the relative parts.

So, if we were to load the person with ID '1' from the database (ignoring any joining for the time being) without the cache, we would have these selects issued:

```
select * from Person where id=1 ; load the person with id 1
select * from Person where parent_id=1 ; load the children of 1 (will return 2, 3)
```

```
select * from Person where parent_id=2 ; load any potential children of 2 (will
return none)
select * from Person where parent_id=3 ; load any potential children of 3 (will
return none)
```
With the cache (assuming it was fully loaded), a direct load will actually issue **NO** select statements, because it can simply look up based on an identifier. If, however we didn't have the associations cached, we'd have these SQL statements invoked for us:

```
select * from Person where parent_id=1 ; load the children of 1 (will return 2, 3)
select * from Person where parent_id=2 ; load any potential children of 2 (will
return none)
select * from Person where parent_id=3 ; load any potential children of 3 (will
return none)
```
That's nearly the same number of SQL statements as when we didn't use the cache at all! This is why it's important to cache associations whenever possible.

Let's say that we wanted to lookup entries based on a more complex query than directly by ID, such as by name. In this case, Hibernate must still issue an SQL statement to get the base data-set for the query. So, for instance, this code:

```
Query query = session.createQuery("from Person as p where p.firstName=?");
query.setString(0, "John");
List l = query.list();
```
... would invoke a single select (assuming our associations were cached).

```
select * from Person where firstName='John'
```
This single select will then return '1', and then the cache will be used for all other lookups as we have everything cached. This mandatory single select is where the query cache comes in.

How the Query Cache Works

The query cache is actually much like the association caching described above; it's really little more than a list of identifiers for a particular type (although again, it is much more complex internally). Let's say we performed a query like this:

```
Query query = session.createQuery("from Person as p where p.parent.id=? and
p.firstName=?");
query.setInt(0, Integer.valueOf(1));
query.setString(1, "Joey");
query.setCacheable(true);
List l = query.list();
```
The query cache works something like this:

```
*-------------------------------------------------------------------------------
------*
|                                  Query Cache
|
|-------------------------------------------------------------------------------
------|
| ["from Person as p where p.parent.id=? and p.firstName=?", [ 1 , "Joey"] ] -> [
2 ] ] |
*-------------------------------------------------------------------------------
------*
```

The combination of the query and the values provided as parameters to that query is used as a key, and the value is the list of identifiers for that query. Note that this becomes more complex from an internal perspective as you begin to consider that a query can have an effect of altering associations to objects returned that query; not to mention the fact that a query may not return whole objects, but may in fact only return scalar values (when you have supplied a select clause for instance). That being said, this is a sound and reliable way to think of the query cache conceptually.

If the second-level cache is enabled (which it should be for objects returned via a query cache), then the object of type Person with an id of 2 will be pulled from the cache (if available in the cache), it will then be hydradated, as well as any associations.

I hope that this quick glance into how Hibernate holds on to values gives you some idea as to how information can be cached by Hibernate, and will give you some insight into how you can manage your mappings so that you can squeeze the maximum performance possible out of your application.

Until next time,

R.J. Lorimer
Contributing Editor – rj -at- javalobby.org
Author – http://www.coffee-bytes.com
Software Consultant – http://www.crosslogic.com